Concurrent Programming

class Thread
interface Runnable
class Object

Multiprocessing
– Using Multiple Processors.

Multiprogramming (Batch)
– Switch and execute different jobs simultaneously, to improve CPU utilization.

Multitasking (Interactive)
– Perform more than one task at a time for each user, to improve response time.

Multithreading
– Threads in a user task (process) can share code, data, and system resources, and access them concurrently.
– Each thread has a separate program counter, registers, and a run-time stack.

Multitasking

ls
more a.txt
emacs -nw a.java
gcc a.c ; a.out
…
logout

ls
xclock &
netscape &
emacs a.java &
javac a.java ; java a
…

Browser : A Multithreaded Application

• Concurrent Activities:
  – Scrolling a page.
  – Downloading an applet/image over the internet.
  – Playing animation and sound simultaneously.
  – Printing a page in the background.
  – Updating Stock Quotes automatically.
  – FTP-ing a remote file.
  – ...

• Browser is a mini-OS!
Threads in Java

- A single sequential flow of control (of execution of expressions and statements) is called a thread.
- A thread independently executes Java code, but may share address space with other threads. (cf. Process)

Java primitives for concurrent programming are based on Hoare’s monitors.

Motivation

Application independent interleaving of threads.

- Support Concurrent Activities
  - Multimedia Applications
  - Advertisements, Sports/stocks ticker
- Improve user response
  - User input in interactive programs
  - I/O in networked programs “in the background”
- Improve CPU utilization
  - Demand-driven thread-scheduling

Overall Game Plan

- How to create and execute multiple threads?
  - Interleaved computations
  - Indiscriminate interleaving interferes with application semantics
- Introduce Mutual Exclusion constructs
  - Example: Printing documents on a shared printer
- Introduce Synchronization constructs
  - Example: Buffering on a shared printer

Java Primitives

- Mutual exclusion
  - synchronized methods in a class cannot be run concurrently on behalf of different threads.
  - synchronized statement locks an object.
    - “Object” is an instance of a class (incl. class Class).
- Synchronization / Cooperation
  - Methods wait(), notify(), notifyAll() etc.
    in class Object enable threads to communicate and regulate each other’s progress.
Creating and Running Threads

- Define subclass of class Thread.
  
  ```java
  public void run() {}
  ```

- Create an instance of the subclass, and invoke `start()` on it. Now `run()` method executes in an independent thread.

Example using class Thread

```java
class Echo extends Thread {
    int id;
    Echo(int i) {
        id = i;
    }
    public void run() {
        while (true) {
            System.out.println(id +" ABC ”);
            // yield();
        }
    }
    public static void main (String[] args)
    {
        new Echo(1) . start();
        new Echo(2) . start();
    }
}
```

```bash
> javac Echo.java
> java   Echo
```

<table>
<thead>
<tr>
<th>Solaris</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
<tr>
<td>1 ABC</td>
<td>1 ABC</td>
</tr>
</tbody>
</table>

With `yield()` in both OS...
Alternative: Creating and Running Threads

- Threads can also be created from an instance of a class implementing the interface Runnable.
  - Required when the class is defined by extension.

Example using interface Runnable

```java
class Echo implements Runnable {
    int id;
    Echo(int i) {
        id = i;
    }
    public void run() {
        while (true) {
            System.out.println(id + " ABC ");
            // yield();
        }
    }
    public static void main(String[] args) {
        new Thread(new Echo(1)).start();
        new Thread(new Echo(2)).start();
    }
}
```

Example in C# using System.Threading;

```csharp
class Echo {
    int id;
    Echo(int i) {
        id = i;
    }
    public void run() {
        while (true)
            System.Console.WriteLine(id + " ABC ");
    }
    public static void Main () {
        new Thread ( new ThreadStart(new Echo(1).run)) . Start();
        new Thread ( new ThreadStart(new Echo(2).run)) . Start();
    }
}
```

Alternate Rendition in Java

```java
class Echo implements Runnable {
    public void run() {
        try {
            while (true) {
                Thread.sleep(1000);
                System.out.println("ABC");
            }
        } catch (InterruptedException e) {
            return;
        }
    }
    public static void main(String[] args) {
        Runnable r = new Echo();
        new Thread(r) . start();
    }
}
```
Threads and Applets

A Simple Bouncing Ball Animation

```java
class Ball {
    static final int TOP = 10, BOTTOM = 150;
    int incr = 2;
    int ypos = TOP;
    void paint(java.awt.Graphics g) {
        if (ypos < TOP || ypos > BOTTOM)
            incr = -incr;
        ypos += incr;
        g.setColor(java.awt.Color.cyan);
        g.fillOval(10, ypos, 10, 10);
    }
}
```

```java
// <applet code=Bounce.class height=200 width=50></applet>
public class Bounce extends java.applet.Applet {
    Ball b;
    public void init() {
        b = new Ball();
        setBackground(java.awt.Color.red);
    }
    public void paint(java.awt.Graphics g) {
        b.paint(g);
        repaint(15);
        /* try  { while (true) {
                Thread.sleep(15);
                repaint();
            } catch (InterruptedException e) {} */
    }
}
```

Threaded Applet

```java
public class ThreadedBounce extends java.applet.Applet implements Runnable {
    Ball b;
    Thread t;
    public void init() {
        b = new Ball();
        setBackground(java.awt.Color.red);
        t = new Thread(this);
        t.start();
    }
}
```
Another version of Threaded Applet

```java
public class ThreadedColorBounce extends java.applet.Applet implements Runnable {
    Ball b;
    Thread t;
    public void init() {
        b = new Ball();
        setBackground(java.awt.Color.red);
    }
    public void start() {
        if (t == null) {
            t = new Thread(this);
            t.start();
        }
    }
    public void stop() {
        t = null;
    }
    public void paint(java.awt.Graphics g) {
        b.paint(g);
    }
    public void run() {
        try {
            while (true) {
                Thread.sleep(15);
                repaint();
            }
        } catch (InterruptedException e) {};
    }
}
```

Color Changing Ball

```java
class ColorBall extends Thread {
    static final int TOP = 10, BOTTOM = 150;
    int incr = 2;
    java.awt.Color c = java.awt.Color.cyan;
    int ypos = TOP;
    public ColorBall(){
        start();
    }
    void paint(java.awt.Graphics g) {
        if (ypos < TOP || ypos > BOTTOM) {
            ypos += incr;
            g.setColor(c);
            g.fillOval(10, ypos, 10, 10);
        } ...
    }
}
```
public void run() {
    try {
        while (true) {
            sleep(600);
            c = new java.awt.Color(
                (float) Math.random(),
                (float) Math.random(),
                (float) Math.random() );
        } catch (InterruptedException e) {} 
    }
}
Motivation for Mutual Exclusion

Concurrent Access

A:368-369
A:370-371
B:370-371
A:372-373
A:374-375
B:374-375
B:376-377
B:377-378
B:378-379
B:379-380
A:381-382
A:382-383
A:383-384
A:385-386
A:386-387
A:387-388
A:388-389
A:390-391

A:828-829
B:829-830
B:830-831
A:831-832
B:832-833
A:833-834
B:834-835
A:835-836
B:836-837
B:837-838
B:838-839
B:839-840

Interaction of Language Features

run() method case study

Subclassing : Access Control

class Thread { ...
    public void run() {}
}
class SubThread extends Thread {
    public void run() {
        //************ILLEGAL*************/
        void run() { ... }
        //************ILLEGAL*************/
    }
    // Otherwise, run() can be invoked on a SubThread instance via (its reference in) a Thread variable (due to dynamic binding) in another package.
Subclassing: Exception

class Thread {
    public void run() {}
}

class SubThread extends Thread {
    public void run() {
        /*************************/
        ILLEGAL
        *******************/
        public void run() throws InterruptedException {
            sleep(1000);
        }
        /*************************/
        ILLEGAL
        *******************/
        }

    // run() invoked on a SubThread instance via a Thread
    variable (by dynamic binding) can violate the contract that
    run() of class Thread does not throw any exception.

Subclassing: synchronized

- A synchronized method can override or be
  overridden by a non-synchronized method.
  However, this does not alter "synchronized"-status
  of the overridden method accessible using super.
- The locks are released when synchronized
  statements and synchronized method invocations
  complete abruptly by throwing an exception.
- Incorporating threads in the language enables
  dealing with problematic interactions with other
  language features.

synchronized methods

- A method that is not synchronized can run
  concurrently with a synchronized method
  on behalf of different threads.
  - The “locking”-analogy breaks down here.
  - For concreteness, consider an example involving two
    threads: (i) T1 running synchronized method m1 and
    (ii) T2 running ordinary method m2 on the same
    object O.
- A synchronized method can call another
  synchronized method in the same thread.
  - Recursion meaningful
synchronized methods

- A synchronized instance (resp. static) method locks instance (resp. static) fields.
  - A synchronized instance method does not protect static fields, and they can corrupt it through an update.
  - For concreteness, consider an example involving two threads T1 and T2 running synchronized instance methods m1 and m2 on objects O1 and O2 of the same class C. Both m1 and m2 are permitted to concurrently access the static fields of C, and there is potential for corruption. In such situations, static fields must be accessed in the body of instance methods using calls to synchronized static methods.

Synchronized instance methods run on distinct objects in different threads can clash on static fields. Hence, synchronized static methods are necessary to protect static fields from concurrent access.

Thread Scheduling

- A thread is blocked (that is, it is not runnable) if it is:
  - sleeping.
  - suspended.
  - waiting.
  - executing a “blocked” method.
  - “blocked” for I/O.

- Java neither detects nor prevents deadlocks.
Thread Scheduling

- Java runs a *non-blocked* (runnable) highest-priority thread.
- A thread can be *preempted* by a higher priority thread that becomes runnable.
  - The scheduler may however choose to run lower priority thread, to avoid starvation.
- The same (highest) priority threads are run in *round-robin* fashion.

Scheduling Ambiguity

- JVM on *Windows* uses *time-slicing* to schedule same priority threads.
  - (“Fair policy”)
- JVM on *Solaris* chooses one of the same priority threads to run until the thread voluntarily relinquishes the CPU (by sleeping, yielding, or exiting), or until a higher priority thread *preempts* it.

Cooperative Concurrent Threads

*Producer-Consumer Problem*

*Bounded-Buffer Problem*

*(Clients and Servers : Resource Sharing)*

```java
class Buffer {
    Object x;
    // synchronized
    public void put (Object _x, String id) {
        System.out.println("Done << " + id + " puts " + _x);
        x = _x;
    }
    // synchronized
    public Object get (String id) {
        System.out.println("Done >> " + id + " gets " + x);
        return x;
    }
}
```
class Producer extends Thread {
    private Random r = new Random();
    Buffer b;       String id;
    public Producer(Buffer _b, String _id) {
        b = _b;            id = _id;
    }
    public void run () {
        int delay;           Integer ir;
        try {
            while (true) {
                delay = Math.abs(r.nextInt() % 1999) + 50;
                sleep(delay);     ir = new Integer(delay);
                System.out.println("Ready << "+id+" puts "+ ir);
                b.put(ir, id);
            }
        } catch (Exception e) {System.out.println("Exception$ "+ e);};
    }
}

class Consumer extends Thread {
    private Random r = new Random();
    Buffer b;       String id;
    public Consumer(Buffer _b, String _id) {
        b = _b;            id = _id;
    }
    public void run () {
        int delay;         Integer ir = null;
        try {
            while (true) {
                delay = Math.abs(r.nextInt() % 1999) + 50;
                sleep(delay);
                System.out.println("Ready >> "+id+" gets ");
                ir = (Integer) b.get(id);
            }
        } catch (Exception e) {System.out.println("Exception$ "+ e);};
    }
}

public class PCB {
    public static void main(String[] args) {
        Buffer b = new Buffer();
        new Consumer(b,"C1") . start();
        new Producer(b,"P1") . start();
        try {Thread.sleep(1000);}
        catch (Exception e){};
        new Consumer(b,"C2") . start();
        new Producer(b,"P2") . start();
    }
}
Concurrency control: class Buffer

```java
class Buffer {
    Object x;
    boolean empty = true;
    public synchronized void put (Object x, String id) throws Exception {
        while (!empty) wait();
        empty = ! empty;
        System.out.println("Done << " + id + " puts " + x);
        x = x;
        notifyAll();
    }
    ...
    public synchronized Object get (String id) throws Exception {
        while (empty) wait();
        empty = ! empty;
        System.out.println("Done >> " + id + " gets " + x);
        notifyAll();
        return x;
    }
}
```

- Two producers (consumers) cannot put(get) two items in (from) the same buffer slot simultaneously.
- Two producers (consumers) cannot put(get) two items in (from) the same buffer slot without an intervening consumer (producer).
- A consumer (producer) is not permitted to get (put) an item from (into) buffer if empty (full).

Synchronization and Mutual Exclusion

```java
public synchronized void put (Object _x, String id) throws Exception {
    while (!empty) wait();
    empty = ! empty;
    System.out.println("Done << " + id + " puts " + _x);
    x = _x;
    notifyAll();
}
```

- Note that if-then cannot replace while because notifyAll, due to a consumer, can wake up two waiting producers, and only one producer will find the buffer empty ("race condition")

```java
public synchronized Object get (String id) throws Exception {
    while (empty) wait();
    empty = ! empty;
    System.out.println("Done >> " + id + " gets " + x);
    notifyAll();
    return x;
}
```

- Note that notify cannot always be used in place of notifyAll because a consumer may be woken up.
- While required because notifyAll due to a producer does not guarantee that waiting consumers will find an item in the buffer.
Methods in class Object

- public final void wait() throws InterruptedException {...}
  - Current thread is suspended (after atomically releasing the lock on this) until some other thread invokes notify() or notifyAll() or interrupts.
- public final void notify() {...}
  - Notifies exactly one thread waiting on this object for a condition to change. The awakened thread cannot proceed until this thread relinquishes the lock.
- public final void notifyAll() {...}
  - These methods must be invoked inside synchronized code or else IllegalMonitorStateException will be thrown.

Single Cell Buffer in C#

```csharp
using System;
using System.Threading;

class Buffer {
    object x;
    bool empty = true;

    public void put(object _x, string id) {
        lock(this) {
            while (!empty) Monitor.Wait(this);
            empty = !empty;
            Console.WriteLine("Done << " + id + " puts " + _x);
            x = _x;
            Monitor.PulseAll(this);
        }
    }

    public object get(string id) {
        lock(this) {
            while (empty) Monitor.Wait(this);
            empty = !empty;
            Console.WriteLine("Done >> " + id + " gets " + x);
            return x;
        }
    }
}
```

class Producer {
    private Random r = new Random();
    Buffer b;
    string id;

    public Producer(Buffer _b, string _id) {
        b = _b;            id = _id;
    }

    public void run () {
        try {
            while (true) {
                int delay = r.Next(2000) + 50;
                Thread.Sleep(delay);
                Console.WriteLine("Ready << " + id + " puts " + delay);
                b.put(delay, id);
            }
        } catch (Exception e) {Console.WriteLine("Exception << " + e);;}
    }
}

class Consumer {
    private Random r = new Random();
    Buffer b;
    string id;

    public Consumer(Buffer _b, string _id) {
        b = _b;            id = _id;
    }

    public void run () {
        try {
            while (true) {
                int delay = r.Next(2000) + 50;
                Thread.Sleep(delay);
                Console.WriteLine("Ready >> " + id + " gets " + delay);
                delay = (int) b.get(id);
            }
        } catch (Exception e) {Console.WriteLine("Exception >> " + e);;}
    }
}
public class PCB {
    public static void Main(string[] args) {
        Buffer b = new Buffer();
        new Thread( new ThreadStart(new Consumer(b,"C1"),run)).Start();
        new Thread( new ThreadStart(new Producer(b,"P1"),run)).Start();
        try {Thread.Sleep(1000);} catch(Exception e){Console.WriteLine("Exception " + e);}
        new Thread( new ThreadStart(new Consumer(b,"C2"),run)).Start();
        new Thread( new ThreadStart(new Producer(b,"P2"),run)).Start();
    }
}