

CS61B Lecture #25

Administrative:

Today:

- Threads
- Communication between threads
- Synchronization
- Mailboxes
- Coroutines

Coming Up: *Data Structures, Chapter 12, Graph Structures*

Threads

- So far, all our programs consist of single sequence of instructions.
- Each such sequence is called a *thread* (for “thread of control”) in Java.
- Java supports programs containing *multiple* threads, which (conceptually) run concurrently.
- Actually, on a uniprocessor, only one thread at a time actually runs, while others wait, but this is largely invisible.
- To allow program access to threads, Java provides the type `Thread` in `java.lang`. Each `Thread` contains information about, and controls, one thread.
- Simultaneous access to data from two threads can cause chaos, so are also constructs for controlled communication, allowing threads to *lock* objects, to *wait* to be notified of events, and to *interrupt* other threads.

But Why?

- Typical Java programs always have > 1 thread: besides the main program, others clean up garbage objects, receive signals, and other stuff.
- When programs deal with asynchronous events, is sometimes convenient to organize into subprograms, one for each independent, related sequence of events, one for other things.
- Threads allow us to insulate one such subprogram from another.
- GUIs often organized like this: application is doing some computation or I/O, another thread waits for mouse clicks (like 'Stop'), another pays attention to updating the screen as needed.
- And, of course, sometimes *do* have a real multiprocessor.

Java Mechanics

- To specify the actions "walking" and "chewing gum":

```
class Chewer1 implements Runnable {
    public void run () { while (true) ChewGum(); }
}
class Walker1 implements Runnable {
    public void run () { while (true) Walk(); }
}
```

- To walk and chew gum:

```
Thread chomp = new Thread (new Chewer1 ());
Thread clomp = new Thread (new Walker1 ());
chomp.start (); clomp.start ();
```

- Alternative:

```
class Chewer2 extends Thread {
    // NOTE: Thread implements Runnable.
    public void run () { while (true) ChewGum(); }
}
class Walker2 extends Thread {
    public void run () { while (true) Walk(); }
}
```

```
-----
Thread chomp = new Chewer2 (), clomp = new Walker2 ();
chomp.start (); clomp.start ();
```

Avoiding Interference

- When one thread has data for another, one must wait for the other to be ready.
- Likewise, if two threads use the same data structure, generally only one should modify it at a time; other must wait.
- E.g., what would happen if two threads simultaneously inserted an item into a linked list at the same point in the list?
- A: Both could conceivably execute

```
p.next = new ListCell(x, p.next);
```

with the *same* values of `p` and `p.next`; one insertion is lost.

- Can arrange for only one thread at a time to execute a method on a particular object with either of

<pre>void f (...) {</pre>		<pre>synchronized void f (...) {</pre>
<pre> synchronized (this) {</pre>		<pre> <i>body of f</i></pre>
<pre> <i>body of f</i> }</pre>		
<pre> }</pre>		
<pre>}</pre>		

Communicating the Hard Way

- Communicating data is tricky: the faster party must wait for the slower.
- Obvious approaches don't work: suppose thread1 wants to give data to thread2 by calling `DataExchanger.putData (...)`:

```
class DataExchanger {
    static volatile Object value;
    Object getData () {
        Object r; r = null;
        while (r == null) { r = value; }
        value = null;
        return r;
    }
    void putData (Object data) {
        while (value != null) { }
        value = data;
    }
}
```

- ...and thread2 receives the data with `DataExchanger.getData()`.
- *Lots* of ways this doesn't work! One thread can monopolize machine while waiting; two threads executing `getData` or `putData` simultaneously cause chaos.

Primitive Java Facilities

- wait method on Object makes thread wait (not using processor) until notified by notifyAll, unlocking the Object while it waits.
- E.g., ucb.util.mailbox has something like this (simplified):

```
interface Mailbox {
    void deposit (Object msg) throws InterruptedException;
    Object receive () throws InterruptedException;
}
```

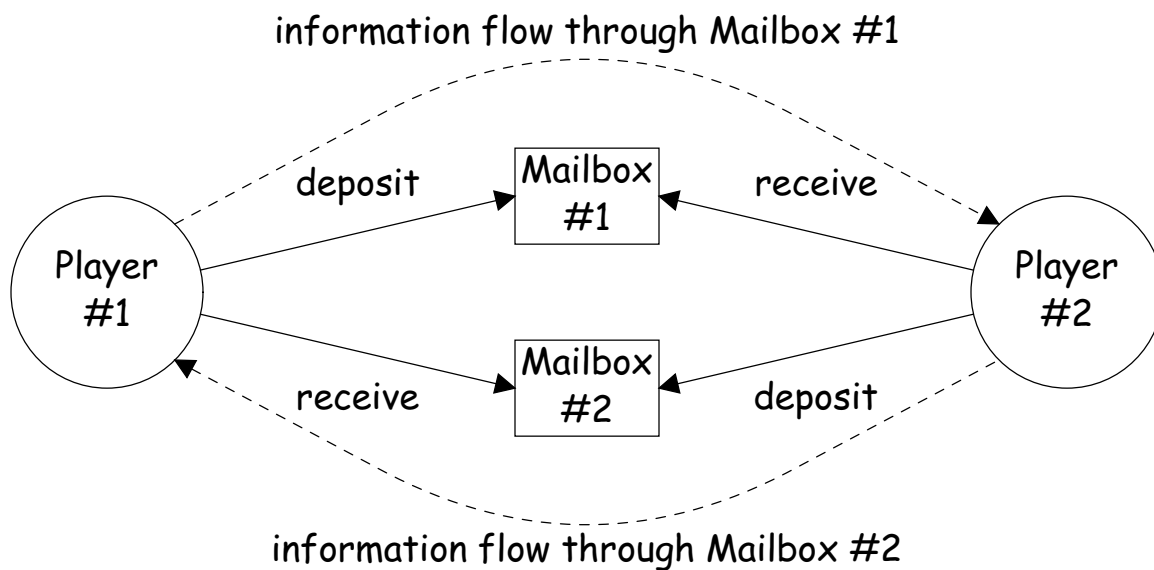
```
class QueuedMailbox implements Mailbox {
    private List queue = new LinkedList ();

    public synchronized void deposit (Object msg) {
        queue.add (msg);
        this.notifyAll (); // Wake any waiting receivers
    }
```

```
    public synchronized Object receive ()
        throws InterruptedException {
        while (queue.isEmpty ())
            wait ();
        return queue.remove (0);
    }
}
```

Message-Passing Style

- Use of Java primitives very error-prone. Wait until CS162.
- We will just use mailboxes and be happy.
- They allow the following sort of program structure:



- Where each Player is a thread that looks like this

```
if (IGoFirst ())
    myMove = computeMyMove ();
while (!gameOver ()) {
    outBox.deposit (myMove);
    hisMove = inBox.receive ();
    myMove = computeMyMove (hisMove);
}
```

More Concurrency

- Previous example can be done other ways, but mechanism is very flexible.
- E.g., suppose you want to think during opponent's move:

```
if (IGoFirst ())
    myMove = computeMyMove ();
while (!gameOver ()) {
    outBox.deposit (myMove);
    hisMove = null;
    while (hisMove == null) {
        thinkAheadForAWhile ();
        hisMove = inBox.receiveIfPossible ();
    }
    myMove = computeMyMove (hisMove);
}
```

- **receiveIfPossible doesn't wait; returns null if no message yet.**

```
public synchronized Object receiveIfPossible ()
    throws InterruptedException {
    if (queue.isEmpty ())
        return null;
    return queue.remove (0);
}
```

Use In GUIs

- Java library handles things like buttons in Graphical User Interfaces with *callbacks*:

```
JMenuItem item = new JMenuItem ("Quit");
item.addActionListener (
    new ActionListener () {
        public void actionPerformed (ActionEvent e) {
            System.exit (0);
        }
    })
```

- *ActionListener*, like *Comparator*, functions as a kind of lambda expression here. The above tells the system to exit the program whenever the user pushes a certain menu item labeled 'Quit'.
- Is a special thread that does nothing but wait for *events* like mouse clicks, pressed keys, mouse movement, etc., look to see if there is an *ActionListener* waiting for that event, and if so, to call its *actionPerformed*.
- Another special thread periodically performs drawing actions that program has requested.

Interrupts

- An *interrupt* is an event that disrupts the normal flow of control of a program.
- In many systems, interrupts can be totally *asynchronous*, occurring at arbitrary points in a program.
- The Java developers considered this unwise; arranged that interrupts would occur only at controlled points.
- In Java programs, one thread can interrupt another to inform it that something unusual needs attention:

```
otherThread.interrupt ();
```

- But like sending to a mailbox, `otherThread` does not receive the interrupt until it waits: methods `wait`, `sleep` (wait for a period of time), `join` (wait for thread to terminate).
- Interrupt causes these methods to throw `InterruptedException`, so typical use is like this:

```
try {  
    wait ();  
} catch (InterruptedException e) {  
    HandleEmergency ();  
}
```

Remote Mailboxes (A Side Excursion)

- RMI: Remote Method Interface allows one program to refer to objects in another program.
- We use it to allow mailboxes in one program be received from or deposited into in another.
- To use this, you define an *interface* to the remote object:

```
import java.rmi.*;
interface Mailbox extends Remote {
    void deposit (Object msg)
        throws InterruptedException, RemoteException;
    Object receive ()
        throws InterruptedException, RemoteException;
    ...
}
```

- On machine that actually will contain the object, you define

```
class QueuedMailbox ... implements Mailbox {
    Same implementation as before, roughly
}
```

Remote Objects Under the Hood

```
// On machine #1:
```

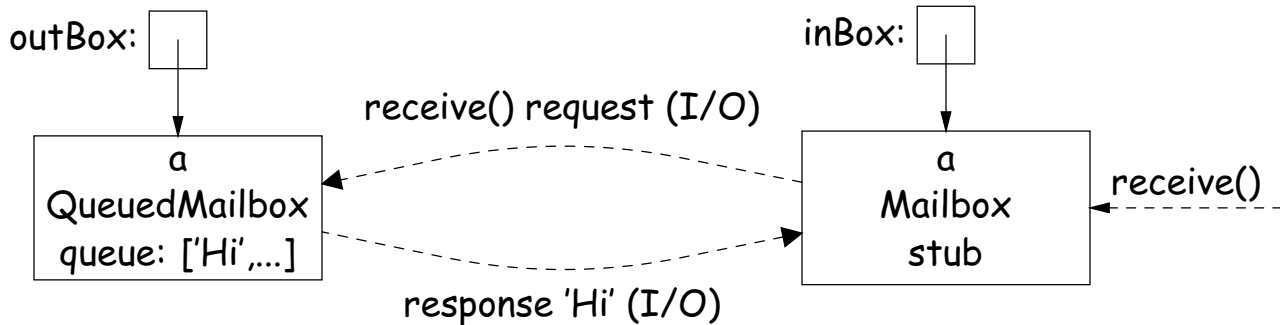
```
Mailbox outBox
```

```
= new QueuedMailbox ();
```

```
// On Machine #2:
```

```
Mailbox inBox
```

```
= get outBox from machine #1
```



- Because `Mailbox` is an interface, hides fact that on Machine #2 doesn't actually have direct access to it.
- Requests for method calls are relayed by I/O to machine that has real object.
- Any argument or return type OK if it also implements `Remote` or can be *serialized*—turned into stream of bytes and back, as can primitive types and `String`.
- Because I/O involved, expect failures, hence every method can throw `RemoteException` (kind of `IOException`).