CS61B Lecture 37

Friday, April 24, 2020

Threads

So far, all of the operations we have done consist of a single sequence of instructions. Each such sequence is called a thread in Java. Java supports programs containing multiple threads, which (conceptually) run concurrently. And even on a uniprocessor (or a single-core processor), only one thread at a time actually runs, while others wait, 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 thread to **lock** objects, to **wait** to be notified of events, and to **interrupt** other threads. This is native to the language, unlike C, which used library imports to achieve a similar effect, but was not native to the language.

This does mean we have to distinct things to talk about: a thread, the sequence, and `Thread`, the Java library type.

Why do we use threads?

Even when we don't specify it, Java programs always have more than one thread: besides the main program, other threads clean up garbage objects, receive signals, update the display, and other stuff.

When programs deal with asynchronous events, it is sometimes convenient to organize into subprograms, one for each independent, related sequence of events. Threads allow us to insulate one such program for another.

GUIs are often structured as such: the application does some computation or I/O, another thread waits for mouse clicks (like 'Stop'), and another pays attention to updating the screen as needed.

Large servers like search engines may be organized this way, with one thread per request. And sometimes we do have a real multi-core processor.

Multi-thread Mechanics

To specify actions with multiple threads, our classes implement an interface called `Runnable`, which has a single method called `run()`. Here is how one might specify actions like walking and chewing gum.

```
class Chewer1 implements Runnable {
    public void run() {
        while (true) chewGum();
    }
}

class Walker1 implements Runnable {
    public void run() {
        while (true) walk();
    }
}
```

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

We can be more concise by using the fact that `Thread` implements `Runnable`.

```
class Chewer2 extends Thread {
    public void run() {
        while (true) chewGum();
    }
}
```

```
Thread chomp = new Chewer2();
chomp.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, while the other must wait. We could cause insertions, deletions, modifications etc. to be lost.

We can arrange for only one thread at a time to execute a method on a particular object with either of the following equivalent definitions:

```
void f(...) {
    synchronized (this) {
        body of f
    }
}

synchronized void f(...) {
    body of f
}
```

Communicating Data

Communicating data is tricky, because the faster party must wait for the slower one. Obvious approaches for sending data from thread to thread don't work:

```
class DataExchanger {
    Object value = null;
```

```
    Object receive() {
        Object r; r = null;
        while (r == null)
            { r = value; }
        value = null;
        return r;
    }
    void deposit(Object data) {
        while (value != null) { }
        value = data;
    }
}

DataExchanger exchanger = new DataExchanger();

// thread1 sends to thread2 with
exchanger.deposit("Hello!");

// thread2 receives from thread1 with
msg = (String) exchanger.receive();
```

A bad thing that could happen: one thread can monopolize machine while waiting; two threads executing deposit or receive simultaneously cause chaos.

Primitive Java Facilities

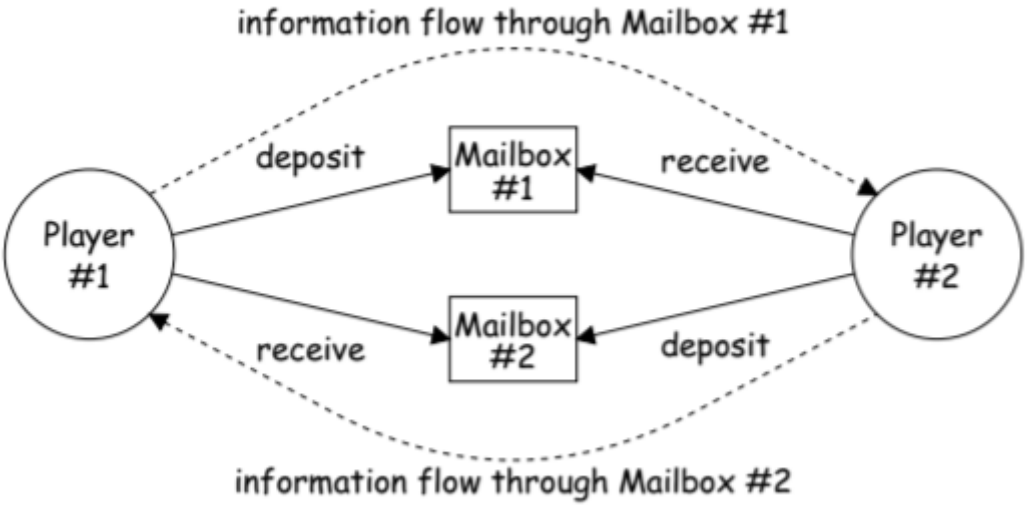
`wait` method on `Object` makes thread wait (not using processor) until notified by `notifyAll`, unlocking the `Object` while it waits.

`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<Object> queue = new LinkedList<Object>();
    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);
    }
}
```

The use of Java primitives is very error-prone, so wait until CS 162. Mailboxes are higher-level, and allow the following program structure:



Where each `Player` is a thread that looks like this:

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

Other Ways

Previous example can be done in other ways, but the mechanism is very flexible. Suppose we want to think during the opponent's move:

```
while (!gameOver()) {
    if (myMove())
        outBox.deposit(computeMyMove(lastMove));
    else {
        do {
            thinkAheadALittle();
            lastMove = inBox.receiveIfPossible();
        } while (lastMove == null);
}
```

`receiveIfPossible` (`written receive(0)` in our actual package) doesn't wait; returns null if no message yet, perhaps like this:

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

Coroutines

A coroutine is a kind of synchronous thread that explicit hands off control to other coroutines so that only one executes at a time, like Python generators. We can get a similar effect with threads and mailboxes:

Here is a recursive, in-order tree iterator:

```
class TreeIterator extends Thread {
    Tree root; Mailbox r;

    TreeIterator(Tree T, Mailbox r) {
        this.root = T; this.dest = r;
    }

    public void run() {
        traverse(root);
        r.deposit(End marker);
    }

    void traverse(Tree t) {
        if (t == null) return;
        traverse(t.left);
        r.deposit(t.label);
        traverse(t.right);
    }
}

void treeProcessor(Tree T) {
    Mailbox m = new QueuedMailbox();
    new TreeIterator(T, m).start();
    while (true) {
        Object x = m.receive();
        if (x is end marker)
            { break; }
        do something with x;
    }
}
```

Use in GUIs

Java's runtime library uses a special thread that does nothing but wait for events. You can designate an object of your choice as a listener; which means that Java's event thread calls a method of that object whenever an event occurs.

As a result, your program can continue to do work while the GUI continues to respond to buttons, menus, etc. Another special thread does all the drawing. You don't even have to be aware when this takes place; just ask the thread to wake up whenever we change something.

Here are some highlights:

```
/** A widget that draws multi-colored lines indicated by mouse. */
class Lines extends JComponent implements MouseListener {
    private List<Point> lines = new ArrayList<Point>();

    Lines() { // Main thread calls this to create one
        setPreferredSize(new Dimension(400, 400));
        addMouseListener(this);
    }
```

```
    public synchronized void paintComponent(Graphics g) { // Paint thread
        g.setColor(Color.white); g.fillRect(0, 0, 400, 400);
        int x, y; x = y = 200;
        Color c = Color.black;
        for (Point p : lines) {
            g.setColor(c); c = chooseNextColor(c);
            g.drawLine(x, y, p.x, p.y); x = p.x; y = p.y;
        }
    }

    public synchronized void mouseClicked(MouseEvent e) // Event thread
        { lines.add(new Point(e.getX(), e.getY())); repaint(); }
    ...
}
```

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

```
try {
    msg = inBox.receive();
} catch (InterruptedException e)
    { HandleEmergency(); }
```

Remote Mailboxes

An **RMI**, Remoted Method Interface, allows one program to refer to objects in another program. We use it to allow mailboxes in one program to be received from or deposited into another.

To use this, you define an interface to the remote object:

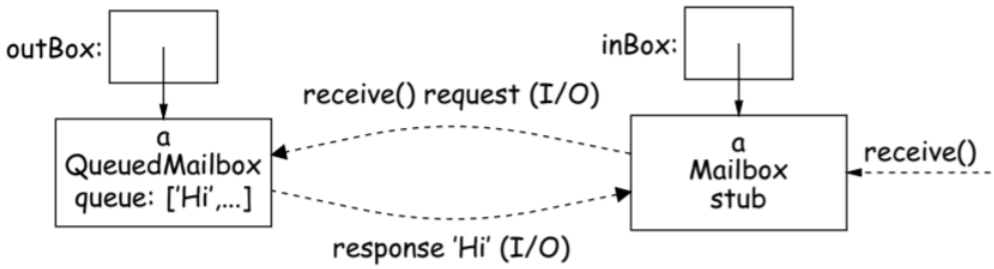
```
import java.rmi.*;
interface Mailbox extends Remote {
    void deposit(Object msg) throws InterruptedException, RemoteException;
    Object receive() throws InterruptedException, RemoteException;
    ...
}
```

On the machine that will contain the object:

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

```
// On machine #1:
Mailbox outBox
  = new QueuedMailbox();
```

```
// On Machine #2:
Mailbox inBox
= get outBox from machine #1
```



Because `Mailbox` is an interface, it hides the fact that Machine 2 doesn't have direct access to it. Requests for method calls are relayed by I/O to the machine with the actual `Mailbox`. Any argument or return type is OK if it also implements `Remote` or can be serialized -- turned into a stream of bytes and back, as can primitive types and `String`.

Because I/O is involved, expect failures, hence every method can throw `RemoteException`, a subtype of `IOException`.

Scope and Lifetime

The scope of a declaration is the portion of program text to which it applies (is visible). It need not be contiguous, and in Java, it is static and independent of data.

Lifetime, or extent, of storage is the portion of a program execution during which it exists. It is always contiguous and generally dynamic, depending on data.

Below are the classes of extent:

- **Static**: entire duration of the program
- **Local or automatic**: duration of call or block execution (local variable)
- **Dynamic**: from time of allocation statement (`new`) to deallocation, if any.

Explicit vs Automatic Freeing

Java has no explicit means to free dynamic storage. However, when no expression in any thread can possibly be influenced by or change another object, it might as well not exist:

```
IntList wasteful() {
    IntList c = new IntList(3, new IntList(4, null));
    return c.tail;
    // variable c now deallocated, so no way to get to first cell
}
```

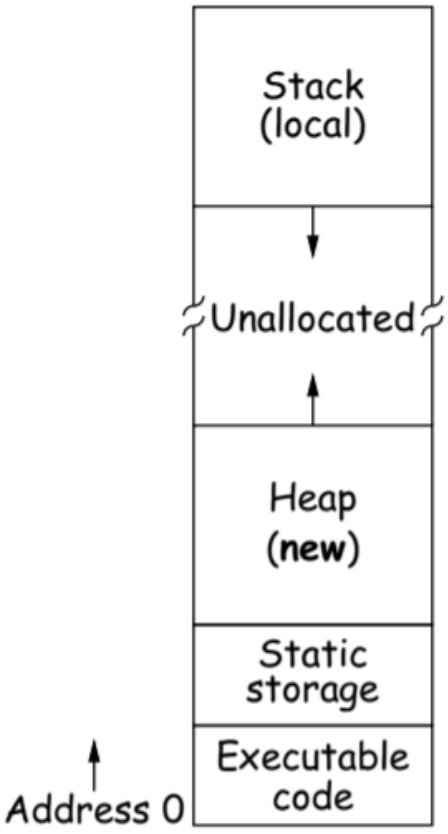
At this point, Java runtime, like Scheme's, recycles the object `c` pointed to: **garbage collection**.

Allocation

Java pointers are represented as integer addresses, and corresponds to the machine's own practice. In Java, you cannot convert integers to pointers, or vice versa. But crucial parts of the Java runtime are implemented in C, or sometimes in machine code, where you *can* do so. Here's a crude allocator in C:

```
char store[STORAGE SIZE]; // Allocated array
size t remainder = STORAGE SIZE;
/** A pointer to a block of at least N bytes of storage */
void* simpleAlloc(size t n) { // void*: pointer to anything
    if (n > remainder) ERROR();
    remainder = (remainder - n) & ~0x7; // Make multiple of 8
    return (void*) (store + remainder);
}
```

Example of Storage Layout in Unix



The OS gives way to turn chunks of unallocated region into heap, and it happens automatically for stack. This is the system that gives you tracebacks on error messages (trace of which lines in your code cause errors).

Explicit Deallocation

In Java, garbage collection is automatic, while C/C++ normally require explicit deallocation, because of:

- Lack of run-time information about what is array
- Possibility of converting pointers to integers.
- Lack of run-time information about unions:

```
union Various {
    int Int;
    char* Pntr;
    double Double;
} X; // X is either an int, char*, or double
```

Java avoids all three problems; automatic collection possible. Explicit freeing can be somewhat faster, but rather error-prone to both memory corruption and memory leaks.

Free Lists

Free lists are a kind of data structure that one can use to keep track of memory you have freed and is available for allocation. The gray areas are unallocated (can be used), and the white areas are allocated. The free list enables us to ask it for the unallocated parts of memory that we can use.