Concurrent Programming

- Traditional programs do one thing at a time.
- Concurrent programs do several things at once.
- Why do this?
  - exploit parallel hardware
  - defer work ("to-do list")
  - interface to slow device (disk, printer, net)
  - interface with person
  - handle many network clients at once

Expressing Concurrency

- Use a **thread** for each concurrent activity.
- Multiple threads can run within a single program.
  - appear to run at the same time
  - really, they take turns running
  - scheduling happens automatically
- A sequential program is a program with one thread in it.

Starting a Thread: Method 1

```
class MyThread extends java.lang.Thread {
        …
        public void run() {
                // code you want the thread to execute
        }
}
```

```
// start a thread
MyThread t = new MyThread(args);
t.start();
```

Starting a Thread: Method 2

```
class MyClass implements java.lang.Runnable {
        …
        public void run() {
                // code you want the thread to execute
        }
}
```

```
// start a thread
MyClass m = new MyClass(args)
Thread t = new Thread(m);
t.start();
```

Other Thread Operations

- **Thread.currentThread** gets the identity of the currently-running thread.
- **Thread.sleep(millis)** puts the calling thread to sleep for **millis** milliseconds.
- **t.stop()** kills thread **t**.
- thread suicide:
  Thread.currentThread.stop();
  - or return from **run**

Shared Memory

- Threads in the same program share memory.
  - Good: can communicate quickly and easily
  - bad: can stomp each other's data structures
    - » horrible bugs!
- Threads in a program share **static**s and **new**ed up objects.
- Private versions of arguments and locals.

Deferring Work with Threads

```
class DeferredGradeReport extends Thread {
        private String student;

        public DeferredGradeReport(String who) {
                student = who;
                start();
        }

        public void run() {
                int grade = calculateGrade(student);
                sendGradeToStudent(student, grade);
        }
```

Why are Threads Tricky?

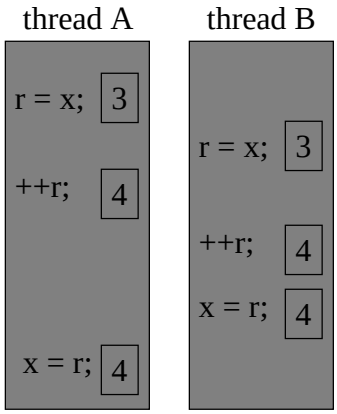
- Single thread: things change only because the program changes them
- multi-threaded: things can change "on their own"

```
x = y;

if(x != y){
        // die horribly
}
```

x = 17;

Another Thread-Related Disaster

```
class C {
        private int x = 3;

        public void increment() {
                int r = x;
                ++r;
                x = r;
        }
}
```



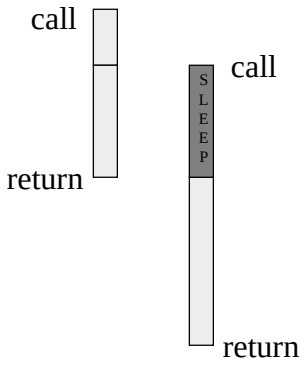
Mutual Exclusion

- protect data from “outside meddling”
- use Java’s synchronized keyword

```
class C {
        private int x = 3;

        public synchronized void increment() {
                int r = x;
                ++r;
                x = r;
        }
}
```

Synchronized: Details

- synchronized method holds a “lock” on the object that the method is invoked on
- if lock is unavailable, you’re put to sleep until you can get it
- lock doesn’t prevent access to data, it only prevents locking



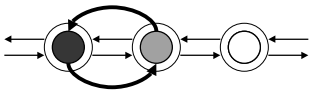
Synchronized: Details

- same thread can acquire the same lock multiple times
  - recursive synchronized methods “work”
- also, synchronized statement in Java

```
C c = new C();
…
synchronized(c){
        ++(c.x);
}
```

Deadlock

```
class LLitem {
  private LLitem next, prev;

   synchronized void remove(){
     synchronized(next){
       next->prev = prev;
     }
     synchronized(prev){
       prev->next = next;
     }
  }
}
```



- A waiting for B; B waiting for A
- we're stuck, forever

Avoiding Deadlock

- key idea: avoid cycles of waiting
- can do use special-case design, then convince yourself it's correct
- usually, follow two simple rules
  - never lock two objects of the same class at the same time
  - if class C was written before class D, code in C never locks a D object (not even indirectly)

Example: Blocking Queue (1.0)

```
public class BlockingQueue extends Queue {
        public synchronized void put(Object o) {
                super.put(o);
        }

        public synchronized Object get() {
                if(super.empty())    return null;
                else    return super.get();
        }
}
```

Using BlockingQueue 1.0

```
// get from BlockingQueue bq
Object o;
do{
        o = bq.get();
}while(o == null);
```

Problem: waste CPU time calling get() over and over

Solution: when queue is empty, get() should block the calling thread

BlockingQueue 2.0

```
public class BlockingQueue extends Queue {
        public synchronized void put(Object o) {
                super.put(o);
        }

        public synchronized Object get() {
                while(super.empty())    Thread.yield();
                return super.get();
        }
}
```

Deadlock!

BlockingQueue 3.0

```
public class BlockingQueue extends Queue {
        // put omitted

        public /*synchronized*/ Object get() {
                while(super.empty())    Thread.yield();
                synchronized(this) {
                        return super.get();
                }
        }
}
```

Sometimes returns null

BlockingQueue 4.0

```
public class BlockingQueue extends Queue {
   public Object get() {
      Object ret;
       do{
          while(super.empty()) Thread.yield();
          synchronized(this) {
             ret = super.get();
          }
       }while(ret == null);
      return ret;
    }
}
```

Works, but inefficient

Solution: wait/notify

- idea: let a thread wait while holding a lock
  - temporarily gives up lock while sleeping
  - awakened when another thread causes something to change
- Java syntax
  - wait(): sleeps temporarily (giving up lock)
  - notify(): wakes up one sleeper
  - notifyAll() wakes up all sleepers

BlockingQueue: Solution

```
public class BlockingQueue extends Queue {
        public synchronized void put(Object o) {
                super.put(o);
                notify();
        }

        public synchronized Object get() {
                while(super.empty())    wait();
                return super.get();
        }
}
```

Wait/Notify Details

- several threads can wait simultaneously
- wait() releases lock, waits for wakeup, then reacquires lock before continuing
- wake-up not immediate, so condition might not be true when thread returns from wait()
  - check for condition in while-loop
- OK to wake up too many threads; deadly to wake up too few
  - when in doubt, notifyAll()