Threading 1

Thanks to Nick Parlante for much of this handout

Concurrency Trends

Faster Computers

- How is it that computers are faster now than 10 years ago?
  - Process improvements -- chips are smaller and run faster
  - Superscalar pipelining parallelism techniques -- doing more than one thing at a time from the one instruction stream.
- Instruction Level Parallelism (ILP)
  - There is a limit to the amount of parallelism that can be extracted from a single, serial stream of instructions.
  - The limit is around 3x or 4x
  - We are well in to the diminishing-returns region of ILP technology.

Hardware Trends

- Moore's law: the density of transistors that we can fit per square mm seems to double about every 18 months -- due to figuring out how to make the transistors and other elements smaller and smaller.
- Here are some hardware factoids to illustrate the increasing transistor budget.
  - The cost of a chip is related to its size in mm^2. It's a super-linear function -- doubling the size of a chip more than doubles its cost.
  - Notice that the chip size has varied around 100-200mm2 while the number of transistors has gone up by a factor of 100.
  - Each chip has a "feature size" its smallest part. As Moore's law progresses, feature size gets smaller. "um" is micrometer -- a millionth of a meter, "nm" is nanometer -- a billionth of a meter
  - 1989: 486 -- 1.0 um -- 1.2M transistors -- 79mm2
  - 1995: Pentium MMX 0.35 um -- 5.5 M transistors -- 128 mm2
  - 1997: AMD Athlon -- 0.25 um -- 22M transistors -- 184mm2
  - 2001: Pentium 4 -- 0.18um -- 42M transistors -- 217 mm2
  - 2004: Prescott Pentium 4 -- 90nm -- 125M transistors -- 112 mm2
  - 2006: Core 2 Duo -- 65nm -- 291M transistors -- 143mm2
  - 2008: Core 2 Penryn -- 45nm -- 410M transistors -- 107mm2
- Q: what do we do with all these transistors?
- A: more cache
- A: more functional units (ILP)
- A: multiple cores, multiple threads on each core (SMT)

1 Billion Transistors

- How do you design a chip with 1 billion transistors?
- What will you do with them all?
- Extract more ILP? -- not really
- More and bigger cache -- ok, but there are limits
- Explicit concurrency -- YES

Hardware vs. Software -- Hard Tradeoff

- **Writing serial, single-thread software is much easier** -- key advice to remember!
- Therefore, hardware thus far has largely been spent in extracting more ILP from a serial stream of instructions.
- That is, we put the burden on the hardware, and keep the software simple. But we are hitting a limit there
- For better performance, we can now move the problem to the **programmers** -- they must write explicitly parallel code. The code is much harder to write, but it can extract much more work from a given amount of hardware.

Hardware Concurrency Trends

- 1. Multiple CPU's -- cache coherency must make expensive off-chip trip
- 2. "Multiple cores" on one chip
  - They can share some on-chip cache
  - A good way to use up more transistors, without doing a whole new design.
- 3. Simultaneous Multi-threading (SMT)
  - One core with multiple sets of registers
  - The core shifts between one thread and another quickly -- say whenever there's an L1 cache miss.
  - Neat feature: hide the latency by overlapping a few active threads -- important if your chip is 10x faster than your memory system.
  - This is called "hyperthreading" by Intel marketing for the P4
- For example, Sun's 2007 Niagara chip has 8 cores per chip, with each core 4-way multithreaded, for a net capacity to run 32 threads. Its performance on a single thread is nothing special, but it can do well with a solution that can be expressed as many threads.

Threads vs. Processes

- Processes
  - Heavyweight-- large start-up costs
  - e.g. Unix process launched from the shell, interacts with other processes through streamed i/o
  - Separate address space
  - Cooperate with simple read/write streams (aka pipes)
  - Synchronization is easy -- typically don't have shared address space (i.e. in some sense, fewer opportunities for bugs)
- Threads
  - Lightweight -- easy to create/destroy
  - All in one address space
  - Can share memory/variables directly (handy)
  - May require more complex synchronization logic to make the shared memory work (potentially hard)

Using Threads

Advantages to multiple threads...

1. Utilize Multiple Hardware Processors

- Re-write the code to use concurrency -- so it can use multiple CPUs. Finish the problem quicker using an 32 processor machine. At present, this is still a little exotic.
- Problem: writing concurrent code is hard, but Moore's law may force us this way as multiple CPU's are the inevitable way to use more transistors. Writing a parallel version will make them most sense for problems where we really care extracting the maximum performance from the hardware.

2. Network/Disk -- Hide The Latency

- Use concurrency to efficiently block when data is not there -- can have hundreds of threads, waiting for their data to come in.
- Even with one CPU, can get excellent results
- The CPU is so much faster than the network, need to efficiently block the connections that are waiting, while doing useful work with the data that has arrived.
- Writing good network code inevitably depends on an understanding of concurrency for this reason. This is no longer an exotic application.

3. Keep the GUI Responsive

- Keep the GUI responsive by separating the "worker" thread from the GUI thread -- this helps an application feel fast and responsive.

Why Concurrency Is Hard

- No language construct yet invented makes the problem go away (in contrast to memory management which has been hugely improved by GC systems). The programmer must be involved. (There is research in the area of compilers that automatically translate serial code to be parallel. Thus far, this does not work for ordinary mainstream code.)
- Counterintuitive -- concurrent bugs are hard to spot in the source code. It is difficult to absorb the proper "concurrent" mindset.
- Because concurrent software is known to be tricky, we will aim for designs that are concurrent but otherwise as simple as we can get away with.
- The easiest bugs are the ones that happen every time.
- In contrast, concurrency bugs show up randomly and sometimes very rarely. They are very machine, VM, and current machine loading dependent, and as a result they are hard to repeat.
- "Concurrency bugs -- the memory bugs of the 21st century."
- Rule of thumb: if you see something bizarre happen, don't just pretend it didn't happen. Note what code was running as best you can.

Java Threads

With Java 5 and 6, higher level threading convenience facilities have been added to the language -- see http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/. However, to work with threads effectively, you need a firm grasp of the fundamentals -- threads, synchronization, race conditions, etc. We will concentrate on those fundamentals, and touch on the higher level facilities just a little.

Current Running Thread

- A thread of execution -- executing statements, sending messages
- Has its own stack, separate from other threads
- Also known as a "thread of control" to distinguish from a java Thread object.
- When have a sequence of statements

```
int i =7;
while (i<10) {
    foo.a();
    ...
}
```

- What we think of as "execute" or "run" -- there is a thread of control that is executing the statements -- the "current running thread".

- A message send, in essence, sends the current running thread over to execute against another object.

static void main(String[] args)

- A Java program begins with a thread executing `main()`, and that one "main" thread executes the whole program.
- We will see how to create and run other threads which will run concurrently.
- In Java, if the main thread reaches the end of `main()`, and no other threads have been started, the program exits. If threads have been started, `main()` exits but the program keeps running on the other threads.

Threads -- Virtual Machine

- Threads in Java are a little easier to deal with than other languages -- there is thread support built in to the language at a low level. Other languages have threads bolted-on to an existing structure.
- The VM keeps track of all the threads and schedules them to get CPU time.
- The scheduling may be preemptive (modern) or cooperative (old, but easier to implement)

Thread Class

- A `Thread` object is just a regular Java object -- it has an address, responds to messages, etc.
- A `Thread` object is a token which represents a thread of control in the VM
- We send messages to the `Thread` object -- the VM interprets these and does the appropriate operations on the underlying thread of control in the VM

Thread.currentThread()

- The static method `Thread.currentThread()` returns a pointer to the `Thread` object that represents the current running thread.

```
int i = 6;
int sum = 7 + 12;      // regular computation

Thread me = Thread.currentThread();
// "me" is the Thread object that represents our thread of
// control (the thread that computed the sum above)
```

Thread Class Use -- Basic Steps

- Here is one way to run some computation in its own thread
- 1. Subclass off `Thread` and implement override the `run()` method
- 2. Create an instance of your `Thread` subclass. It is not running yet, so you can set things up
- 3. Send the thread object the `start()` message -- at this point the VM can allocate a real thread of control, and schedule it to execute the `Thread` object's `run()` method. Do not call `run()` directly -- that's a classic coding error and it does something unintuitive (see example at end of handout).
- 4. A thread of control begins executing the `run()` method of the `Thread` object. The JVM can time-slice between the various threads, running each for a little while.
- 5. Eventually, the thread of control finishes/exits the `run()` method

Thread.sleep(), Thread.yield()

- (static) `Thread.sleep(milliseconds)` blocks the current thread for approximately the given number of milliseconds. May throw an `InterruptedException` if the sleeping thread is interrupted.
- (static) `Thread.yield()` -- voluntarily give up the CPU, so that another thread may run. Just a hint to the VM that perhaps now would be a good time to run a different thread. Old VMs did not switch

threads pre-emptively, so `yield()` made a real difference. However, it is probably not necessary with modern, pre-emptive VMs. However, `yield()` can be used to force more thread switching in an attempt to check for race condition problems. Also, simple VMs (such as on a Palm Pilot), may use a simple, non pre-emptive thread system, in which `yield()` would have some effect.

- The preferred syntax to call these is `Thread.sleep()` or `Thread.yield()`, to emphasize that they are static -- operating on the current running thread.

getName()

- `getName()` on a `Thread` object
- By default, returns something like Thread-0, Thread-1, Thread-2, ... for each thread
- Handy for debugging.
- Alternately, the `Thread` ctor takes a `String` that is used for the name.

Joining

- Suppose you want to wait for another worker thread to complete its `run()`
- Send the `join()` message to the worker thread object -- `worker.join()` -- causes the current running thread to block efficiently until worker finishes its run
- We must handle the `InterruptedException` that `join()` may throw. Java will break us out of the join if we are interrupted by another thread, so really there are two ways to get out of a `join()` -- the other thread finishes, or we are interrupted. In most cases, we will handle `InterruptedException` by doing nothing. (Interruption is a topic of a later lecture).

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

// at this point, two threads may be running -- me and t

// wait for t to complete its run
try {
    t.join();
}
catch (InterruptedException ignored) {}

// now t is done (or we were interrupted)
```

Simple Thread Example

- Strategy: Subclass `Thread`, define the `run()` method

```
/*
 Demonstrates creating a couple worker threads, running them,
 and waiting for them to finish.

 Threads respond to a getName() method, which returns a string
 like "Thread-1" which is handy for debugging.
*/
class FirstWorker extends Thread {
    public void run() {
        long sum = 0;
        for (int i=0; i<10000000; i++) {
            sum = sum + i; // do some work

            // every n iterations, print an update
            // (a bitwise & would be faster -- mod is slow)
            if (i%1000000 == 0) {
                Thread running = Thread.currentThread();
                System.out.println(running.getName() + " " + i);
            }
        }
```

```
    }

    public static void main(String[] args) {
        FirstWorker a = new FirstWorker();
        FirstWorker b = new FirstWorker();

        System.out.println("Starting...");
        a.start();
        b.start();

        // The current running thread (executing main()) blocks
        // until both workers have finished
        try {
            a.join();
            b.join();
        }
        catch (Exception ignored) {}

        System.out.println("All done");
    }
    /*
        Starting...
        Thread-0 0
        Thread-0 10000
        Thread-0 20000
        Thread-1 0
        Thread-1 10000
        Thread-0 30000
        Thread-0 40000
        Thread-1 20000
        Thread-0 50000
        Thread-1 30000
        Thread-0 60000
        Thread-1 40000
        Thread-0 70000
        Thread-1 50000
        Thread-0 80000
        Thread-1 60000
        Thread-0 90000
        Thread-1 70000
        Thread-1 80000
        Thread-1 90000
        All done
    */
}
```

Misc Other Thread Methods

Runnable Interface

- Instead of creating a `Thread` object, you can implement the `Runnable` interface, which defines a single method prototype: `public void run();`
- `run()` defines the code you want to run, and this gives you flexibility to execute the code in different ways -- e.g. pass the `Runnable` to a thread pool to run with one of a set of threads the pool has already going.
- You can pass a `Runnable` to a `Thread` in its constructor, and it will run it.

```
    static void demoRunnable() {
        // Make a runnable.
        Runnable runnable = new Runnable() {
            public void run() {
                System.out.println("Yay Runnable!");
            }
        };

        // Make a thread on the runnable and run it.
        Thread thread = new Thread(runnable);
        thread.start();
    }
```

Priorities

- `getPriority()`/`setPriority()` on `Thread` objects
- Threads have priorities that the scheduler uses to give more time to some threads and less time to others.
- Use priorities to optimize behavior, but not to safeguard critical sections -- priorities are not precise in that way. Use synchronization to protect critical sections no matter what the priorities are.
- There is a school of thought that priorities introduce more complexity than they are worth in a concurrent program, and should never be used. Some VMs ignore priorities anyway.

Complex Thread.currentThread() Example

```
// Demonstrates Thread.currentThread(), and the difference
// betweeen run() and start().
class MiscThread {
    // Prints the name of the thread calling this method.
    public static void whoami() {
        Thread runningMe = Thread.currentThread();
        System.out.println("whoami thread:" + runningMe.getName());
    }

    // Simple worker thread subclass -- run() sleeps
    // and then call whoami().
    public static class MiscWorker extends Thread {
        public void run() {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            MiscThread.whoami();
        }
    }

    public static void main(String[] args) {
        MiscThread.whoami();
        Thread worker = new MiscWorker();
        worker.run();   // Note: never call run() like this!
        worker.start(); // Correct way to start a thread.
        System.out.println("all done");
    }
    /*
     Output:
     whoami thread:main
     whoami thread:main
     all done
     whoami thread:Thread-0
     */
}
```

Mutual Exclusion

Mutual Exclusion / Critical Section

- "Mutual exclusion" -- scheduling threads so that only one thread at a time executes a critical section of code. Also known as "mutex"
- Critical section -- a section of code that manipulates shared memory, and so must not be run by multiple threads at the same time.
- Essentially, keeping the threads from interfering with each other. Avoid reader/writer and writer/writer conflicts on shared memory between threads.
- "Race condition" -- a piece of code with unaddressed concurrency problems, and so depending on how the threads happen to schedule each time, you may get different output.

Coordination

- "Coordination" -- managing the schedules of multiple threads so they can start/pause/etc. to mesh their schedules.
- Typically this centers on handing information from one thread to another, or signaling one thread that another thread has finished doing something.
- We will use `join()` and Semaphores for this (Java also has more primitive `wait()`/`notify()` features that we will not use)

Race Condition Example

```
// Simple class that encapsulates 2 numbers

class Pair {
    private int a, b;

    public Pair() {
        a = 0;
        b = 0;
    }

    // Returns the sum of a and b. (reader)
    public int sum() {
        return(a+b);
    }

    // Increments both a and b. (writer)
    public void inc() {
        a++;
        b++;
    }
}
```

Reader/Writer conflict -- Race Condition

- e.g. thread1 runs `inc()` while thread2 runs `sum()` on the same object.
- In that case, the `sum()` thread could get an incorrect value if the `inc()` is halfway done.
- In part, this happens because the lines of `inc()` and `sum()` interleave
- In fact, even the single statement `a++` is not atomic, so the interleave also happens at a scale finer than a single java statement.
- Java guarantees that 4-byte reads and writes are atomic, but that's it. The statement `a++` actually expands to a non-atomic three-step: read, increment, write.
- Note that this is only a problem if the threads are executing against the same object so they both touch the same memory.

Writer/Writer conflict

- e.g. thread1 runs `inc()` while thread2 runs `inc()` on the same object.
- The two `inc()`'s can interleave to mess up the object state
- `a++` is not atomic -- it can interleave with another `a++` to produce wrong results. This is true in most languages.

Random Interleave Race Condition -- Hard to Observe

- Race conditions depend on two or more threads "interleaving" their execution in just the right way to exhibit the bug. It happens rarely and randomly, but it happens.
- The likelihood of a particular interleave is quite random -- depending on the system load and the number of processors.
- You are more likely to observe the race condition problem on a system with multiple CPUs.

- This is why locating concurrency bugs can be so hard -- they exhibit themselves sporadically.
- Many of the bugs seen in shipping software are concurrency bugs -- they were not seen or tracked down in in-house testing.

Java Locks

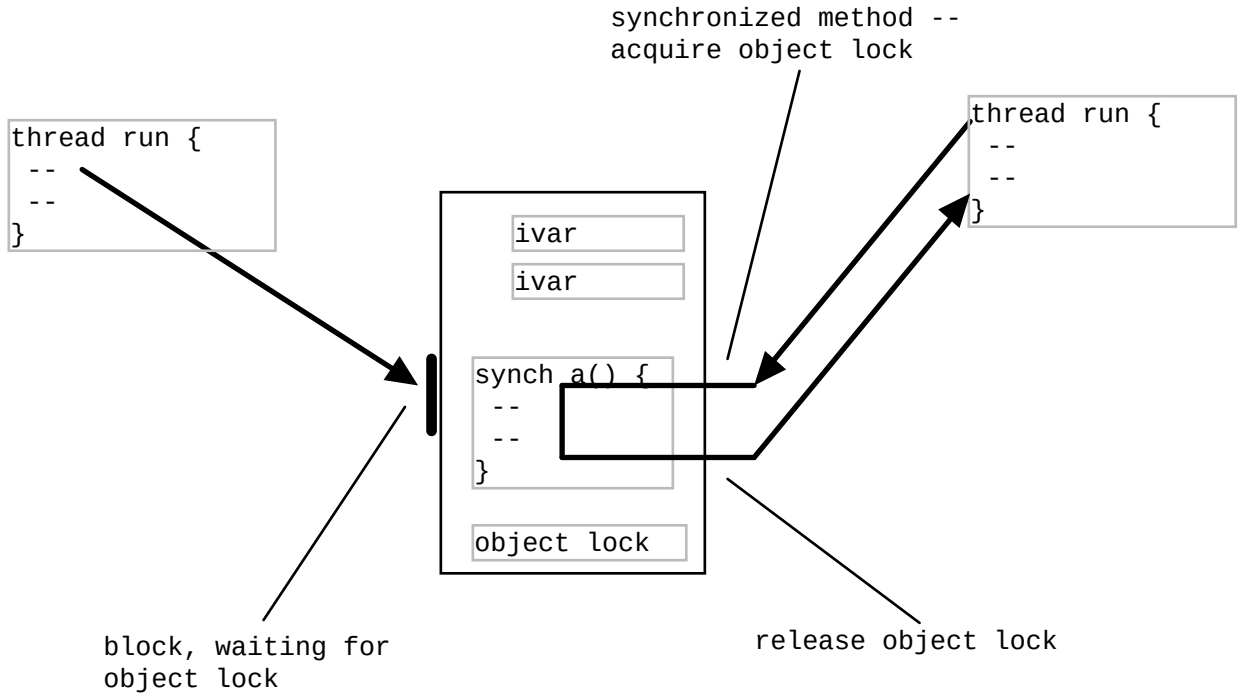
Object Lock + Synchronized Method

- In Java, every object has a "lock"
- A "synchronized" method respects the lock of the receiver object.
- For a thread to execute a synchronized method against a receiver, it first obtains the lock of the receiver.
- The lock is released when the method exits
- If the lock is already held by another thread, the calling thread blocks (efficiently) waiting for the other thread to exit and so make the lock available.
- In this way, multiple threads, in effect, take turns on who can execute against the receiver -- mutual exclusion.

Around The Lock

- The lock is in the receiver object -- it provides mutual exclusion for multiple threads sending messages to **that object**. Other objects have their own locks.
- If a method is not synchronized, it will ignore the lock and just go ahead.

Synchronized Method Picture



Synchronized Method Example

```
/*
 A simple class that demonstrates using the 'synchronized'
 keyword so that multiple threads may send it messages.
 The class stores two ints, a and b; sum() returns
```

```
 their sum, and inc() increments both numbers.

 <p>
 The sum() and incr() methods form a "critical section" --
 they can compute the wrong thing if run by multiple threads
 at the same time. The sum() and inc() methods are declared
 "synchronized" -- they respect the lock in the receiver object.
*/
class Pair {
    private int a, b;

    public Pair() {
        a = 0;
        b = 0;
    }

    // Returns the sum of a and b. (reader)
    // Should always return an even number.
    public synchronized int sum() {
        return(a+b);
    }

    // Increments both a and b. (writer)
    public synchronized void inc() {
        a++;
        b++;
    }
}




/*
 A simple worker subclass of Thread.
 In its run(), sends 1000 inc() messages
 to its Pair object.
 (1000 may not be big enough to exhibit the bug on uniprocessor --
  hardware more like 1000000 may be required).
*/
class PairWorker extends Thread {
    public final int COUNT = 1000;
    private Pair pair;

    // Ctor takes a pointer to the pair we use
    public PairWorker(Pair pair) {
        this.pair = pair;
    }

    // Send many inc() messages to our pair
    public void run() {
        for (int i=0; i<COUNT; i++) {
            pair.inc();
        }
    }


    /*
     Test main -- Create a Pair and 3 workers.
     Start the 3 workers -- they do their run() --
     and wait for the workers to finish.
    */
    public static void main(String args[]) {
        Pair pair = new Pair();
        PairWorker w1 = new PairWorker(pair);
        PairWorker w2 = new PairWorker(pair);
        PairWorker w3 = new PairWorker(pair);

        w1.start();
        w2.start();
        w3.start();
        // the 3 workers are running
        // all sending messages to the same object
```

```
        // we block until the workers complete
        try {
            w1.join();
            w2.join();
            w3.join();
        }
        catch (InterruptedException ignored) {}

        System.out.println("Final sum:" + pair.sum());   // should be 6000
        /*
         If sum()/inc() were not synchronized, the result would
         be 6000 in some cases, and other times random values
         like 5979 due to the writer/writer conflicts of multiple
         threads trying to execute inc() on an object at the same time.
        */
    }
}
```

Multiple acquisition -- ok

- A thread can acquire the same lock multiple times -- that works fine.
- Put another way: a thread does not block waiting for itself. If a thread holds a lock, it can acquire that lock again.
- e.g. `inc()` could call `sum()`, and it will work out right -- the lock will be released only when the thread's lock count goes to 0.
- This is sometimes known as "recursive locks"

Exceptions -- lock release ok

- A thread releases its locks as it returns from methods, no matter how. In particular, an exception terminating the method will release the locks correctly.
- It's nice to have support for this sort of detail built in to the system at a low level.

Synchronized(obj) {...} Block

- A variant of the synchronized method.
- Acquire/Release lock for a specific object. Code looks like...

```
void someOperation(Foo foo) {
    int sum = 0;
    synchronized(foo) {    // acquire foo lock
        sum += foo.value;
    }   // release foo lock
    ...
```

- Similar to synchronized method
  - Uses the same lock as synchronized methods -- the lock in each object.
  - Saying `synchronized (this) { ... }` mimics a synchronized method
- Allows us to specify exactly where to lock, and with what object
- A little slower, a little less readable
- Conclusion: synchronized methods are slightly preferable, but `synchronized(obj)` gives you flexibility -- you can talk about the lock of an object other than the receiver and in places other than the start/end of a method.

Synchronized(obj) {...} Block example

```
/*
 Demonstrates using individual lock objects with the
 synchronized(lock) {...} form instead of synchronizing methods --
 allows finer grain in the locking.
*/
class MultiSynch {
    // one lock for the fruits
```

```
private int apple, bannana;
private Object fruitLock;

// one lock for the nums
private int[] nums;
private int numLen;
private Object numLock;

public MultiSynch() {
    apple = 0;
    bannana = 0;
    // allocate an object just to use it as a lock
    // (could use a string or some other object just as well)
    fruitLock = new Object();

    nums = new int[100];
    numLen = 0;
    numLock = new Object();
}

public void addFruit() {
    synchronized(fruitLock) {
        apple++;
        bannana++;
    }
}

public int getFruit() {
    synchronized(fruitLock) {
        return(apple+bannana);
    }
}

public void pushNum(int num) {
    synchronized(numLock) {
        nums[numLen] = num;
        numLen++;
    }
}

// Suppose we pop and return num, but if the num is negative return
// its absolute value -- demonstrates holding the lock for the minimum time.
public int popNum() {
    int result;
    synchronized(numLock) {
        result = nums[numLen-1];
        numLen--;
    }
    // do computation not holding the lock if possible
    if (result<0) result = -1 * result;
    return(result);
}

public void both() {
    synchronized(fruitLock) {
        synchronized(numLock) {
            // some scary operation that uses both fruit and nums
            // note: acquire locks in the same order everwhere to avoid
            // deadlock.
        }
    }
}
```

Counting Semaphore

- A classic abstraction used in many languages
- Use for co-ordination -- getting threads to co-ordinate their schedules (more complex than plain mutual exclusion)
- Initialize with some `int permit` variable with a value -- 0, 1, ... Each positive `permit` can potentially be acquired

- `acquire()` -- if `permit` if positive, decrement it and continue (take a `permit`), or if `permit` is zero or negative block (wait)
- `release()` -- increments the `permit`, unblocks a blocked thread if `permit` is positive. The newly awoken thread will try to take the `permit`.
- The Semaphore may or may not be "fair" about which thread gets the new `permit`. Being "fair" is more expensive, so generally implementations are random.
- "Barging" -- Suppose a thread, `Alice`, is blocked in `acquire()` trying to get a permit. Some other thread does a `release()`. There may be a window of time where some third thread grabs the new permit before `Alice` can get it, even though `Alice` was waiting first. This is sometimes known as "barging", and it gets back to if the system guarantees FIFO fairness or not. In general, your code should not require fairness to function correctly.
- Can initialize a semaphore to a negative value -- all `acquire()` attempts will block, until enough `release()` calls make the credit positive.
- Note that when blocked in `acquire()`, a thread continues to hold the locks it held at that point.
- Therefore, watch out for deadlock -- if the blocked thread holds a lock needed by the thread that will do the `release()`.

Acquire() and Interruption

- A `Semaphore` class has been added with Java 5 -- previously you had to provide one.
- The Java 5 `Semaphore` is defined so that `acquire()` blocks waiting for a permit, as usual
- However, while the thread is blocked in `acquire()`, it may be "interrupted" by another thread which causes an `InterruptedException`, which is a future lecture topic.
- To account for the interruption case, all calls to `acquire()` must be wrapped in `try/catch` to account for the case where the `acquire()` returns because of interruption.
- For today we are not worrying about interruption, and so the `try/catch` can just do nothing or `printStackTrace()`. Eclipse can insert a default `try/catch` for you (click on the red x). We will revisit that code in the interruption lecture.

acquire(int n), release(int n)

- There are also `acquire(int n)`, and `release(int n)` variants, that work in bulk blocks of n permits. The `acquire(n)` does not gather/hoard the permits 1 at a time. Instead, it waits to grab them in one block.

acquireUninterruptibly()

- Like `acquire`, but without the interrupted exception, so saves you doing the try/catch.

Semaphore Example

```
import java.util.concurrent.*;
/*
Use two Semaphores to get A and B threads to take turns.
This code works.
*/
class TurnDemo {
    Semaphore aGo = new Semaphore(1);     // a gets to go first
    Semaphore bGo = new Semaphore(0);
```

```
    void a() {
        // aGo.acquire() -- wrapped in try/catch
        try {
            aGo.acquire();
        } catch (InterruptedException ignored) {}
        System.out.println("It's A turn, A rules!");
        bGo.release();
    }

    void b() {
        try {
            bGo.acquire();
        } catch (InterruptedException ignored) {}
        System.out.println("It's B turn, B rules!");
        aGo.release();
    }

    /*
     Q: Suppose a() and b() where synchronized -- what would happen?
     A: deadlock! a thread would take the "this" lock and block its its acquire(),
     and the other thread could never get in to do the matching release().
    */
    public void demo() {
        // Use this to wait for the workers
        // using an initial value of -1
        final Semaphore finished = new Semaphore(-1);

        // Thread that calls b() a lot
        new Thread() {
            public void run() {
                for (int i = 0; i< 10; i++) { b(); }
                finished.release();
            }
        }.start();

        // Thread that calls a() a lot
        new Thread() {
            public void run() {
                for (int i = 0; i< 10; i++) { a(); }
                finished.release();
            }
        }.start();

        // wait for both threads (vs. using join)
        try {
            finished.acquire();
        } catch (InterruptedException ignored) {}
        System.out.println("All done!");
    }

    public static void main(String args[]) {
        new TurnDemo().demo();
    }
}

It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
It's A turn, A rules!
It's B turn, B rules!
All done!
```

Producer/Consumer Problem

- Classic pattern with concurrency
- One set of threads produces things, another set consumes those things
- Coordination problem: The consumers need to block when there are no things available
- Optional: also, the producers can block if they get too far ahead of the consumers
- In this case, we use a `canRemove` semaphore to block the consumers when needed

Producer/Consumer Example

```
/*

import java.util.concurrent.*;
/*
Producer/Consumer problem coded with Semaphores.
This code works correctly.

-"len" represents the number of elements in some imaginary array
-add() adds an element to the end of the array. Add() never blocks --
we assume there's enough space in the array.
-remove() removes an element, but can only finish if there
is an element to be removed. If there is no element, remove()
waits for one to be available.

Strategy:
-We have a canRemove Semaphore
-add() does a canRemove.release(), remove() does a canRemove.acquire()
-Each adder adds COUNT times, and each remover removes COUNT times,
so it balances out in the end.
*/
class AddRemove {
    private int len = 0;   // the number of elements we have
    private Semaphore canRemove; // use to block removing
    private static final int COUNT = 100;

    public AddRemove() {
        len = 0;
        canRemove = new Semaphore(0); // initially, can't remove
    }

    public synchronized void add() {
        len++;
        System.out.println(Thread.currentThread().getName() + " add " + (len-1));
        canRemove.release();
    }

    // tricky: this method cannot be synchronized.
    // We may block in the acquire(), and would be holding the lock
    // needed by the adder.
    // Solution: use a sychronized(this) {..} block
    // *after* the acquire.
    public void remove() {
        try {
            canRemove.acquire();
        } catch (InterruptedException ignored) {}

        synchronized(this) {
            // At this point, we have the lock and len>0
            System.out.println(Thread.currentThread().getName() + " remove " + (len-1));
            len--;
        }
    }


    private class Adder extends Thread {
        public void run() {
            for (int i = 0; i< COUNT; i++) {
                add();
                Thread.yield();    // this just gets the threads to switch around more,
```

```
                    // so the output is a little more interesting
            }
        }
    }

    private class Remover extends Thread {
        public void run() {
            for (int i = 0; i< COUNT; i++) {
                remove();
                Thread.yield();
            }
            System.out.println(getName() + " done");
        }
    }


    public void demo () {

        // Make two "adding" threads
        Thread a1 = new Adder();
        Thread a2 = new Adder();


        // Make two "removing" threads
        Thread r1 = new Remover();
        Thread r2 = new Remover();

        // start them up (any order would work)
        a1.start();
        a2.start();
        r1.start();
        r2.start();

        /*
        output
            Add elem 0
            Add elem 1
            Remove elem 1
            Add elem 1
            Add elem 2
            Add elem 3
            Remove elem 3
            Remove elem 2
            Add elem 2
            Add elem 3
            Remove elem 3
            Remove elem 2
            Add elem 2
            ...
            Remove elem 3
            Remove elem 2
            done
            Remove elem 1
            Remove elem 0
            done
        */

    }

    public static void main(String[] args) {
        AddRemove test = new AddRemove();
        test.demo();
    }
}
```