Parallel & Concurrent Programming:
Concurrency in Java

Emery Berger
CMPSCI 691W - Spring 2006
Concurrency in Java

- Built-in OO-style support
- New concurrency operations
  - Concurrency patterns
Java Object Model

- Every Java object has a lock
  - Lock word
  - Recursive:
    - Thread ID
    - Count
- Condition variables:
  - Wait queues
- Space overhead, but convenient
  - Can always lock an object with synchronized
Synchronized Example

class AtomicCounter {
    private int count;
    AtomicCounter (int n) {
        count = n;
    }
    public void increment() {
        synchronized (this) {
            count++;
        }
    }
    public int getCount() {
        int c;
        synchronized (this) {
            c = count;
        }
        return c;
    }
}

Synchronized Example

class AtomicCounter {
    private int count;
    AtomicCounter (int n) {
        count = n;
    }
    synchronized public void increment() {
        count++;
    }
    synchronized public int getCount() {
        int c;
        c = count;
        return c;
    }
}
Condition Variables

- Also built-in: condition variables
  - `obj.wait()`
  - `obj.wait(long timeout)`
    - Releases monitor, sleeps
  - `obj.notify()`
    - Wakes up one waiting thread
  - `obj.notifyAll()`
    - Wakes up all waiting threads
Java Threads

- extend Thread
- Implement run() method
- Invoke with start()
class CountUp extends Thread {
    public void run () {
        counter.increment();
        System.out.println ("count = " +
            counter.getCount());
    }
    static AtomicCounter counter
        = new AtomicCounter (0);
}

public class testme {
    public static void main (String args[]) {
        for (int i = 0; i < 100; i++) {
            CountUp f = new CountUp();
            f.start();
        }
    }
}
Race Condition

- Previous slide: program has race
- (Example execution)
public class CountUp extends Thread {
    public void run () {
        synchronized (counter) {
            counter.increment();
            System.out.println("count = " +
                counter.getCount());
        }
    }
}

static AtomicCounter counter = new AtomicCounter (0);

public class testme2 {
    public static void main (String args[]) {
        for (int i = 0; i < 100; i++) {
            CountUp f = new CountUp();
            f.start();
        }
    }
}
Thread-Specific Data

- Private local in Thread object = thread-specific data
- Elegant, natural model

```java
class CountUp extends Thread {
    public void run () {
        synchronized (counter) {
            counter.increment();
            System.out.println ("count = " +
                                counter.getCount());
        }
    }
    private AtomicCounter counter
        = new AtomicCounter (0);
}
```
Thread Priority

- Java threads also have **priority**:
  - Unlike UNIX, higher priority value = higher priority
  - If any threads are runnable at priority i, they run instead of any thread at priority $\leq i$
    - Fixed-priority scheduling
    - Caveat: not guaranteed to always hold
Thread Miscellany

- Other Thread methods:
  - `setPriority(int)`
  - `getPriority()`
  - `yield()`
    - Let other threads execute
  - `t.join()`
    - Wait for thread t to complete
Extensive support for concurrency
- Introduced with Java 1.5 ("5")
- Built on Lea’s concurrency library
Old Friends

- Semaphore(int)
  - Ordinary counting semaphore
  - acquire(), tryAcquire(), release()
- Semaphore(int, True)
  - Fair semaphore (FIFO)
Much More...

- Blocking & non-blocking queues
  - Numerous flavors
- Concurrent hash maps
  - “MT-hot”
- Copy-on-write arrays
- Exchanger
- Barriers
- Futures
- Thread pool support
Blocking Queues

- **LinkedBlockingQueue**
  - Blocks on `put()` if full, `poll()` if empty
  - Implement **pipeline** across threads
  - **Producer-consumer** pattern

- Example application:
  - *worker threads*
Blocking Queues 2

- ArrayBlockingQueue
  - Array implementation (bounded buffer)

- Example application:
  - worker threads, without allocation
    - Fixed max number of tasks
**Blocking Queues**

- **SynchronousQueue**
  - Each `put()` waits for `take()`
  - Rendezvous channel

- **Example application:**
  - *worker threads*
  - Same number of threads as tasks
PriorityBlockingQueue
- Unbounded queue, based on heap
- Head = item with lowest “priority”

Example application: concurrent simulation (priority = time)
**Blocking Queues 5**

- **DelayQueue**
  - Time-based scheduling queue
  - Only *expired* items may be removed

- Example applications:
  - Manage objects with timeouts
  - Simulator
Copy-on-write arrays

- CopyOnWriteArrayList
  - Mutations = **copy** entire backing array, update particular item
- Cost?
- When would this be desirable?
Exchanger

- Simple rendezvous
- Each thread gives object to exchanger, and gets other

```java
yours = exchanger.exchange (mine);
```
Barriers

- All threads reach sync point before continuing: **barrier**
- Very common for scientific apps – in loop: do work, reach barrier

```java
for (int i = 0; i < 1000; i++) {
    // do work
    try {
        barrier.await();
    } catch (Exception e) { ... }
}
```
**Futures**

- **FutureTask** – asynchronously executes some function to compute value
- Future operations:
  - `run()` – starts execution
  - `get()` – waits for future to complete,
  - `cancel()` – aborts execution
  - `isDone()` – check if future complete
**Thread Pools**

- Thread invocation & destruction relatively expensive
- Instead: use **pool** of threads
  - When new task arrives, get thread from pool to work on it; block if pool empty
  - Faster with many tasks
  - Limits max threads
- **ThreadPoolExecutor** class
The End