CS193J: Programming in Java
Summer Quarter 2003

Lecture 10
Thread Interruption, Cooperation (wait/notify), Swing Thread, Threading conclusions

Manu Kumar
sneaker@stanford.edu

Handouts

- 1 Handout for today!
  - #21: Threading 3
  - #22: HW3a: ThreadBank
  - #23: HW3b: LinkTester

Homework #2 feedback

- What did you think?
  - SCPD students are again encouraged to email their comments to me at sneaker@stanford.edu

Recap

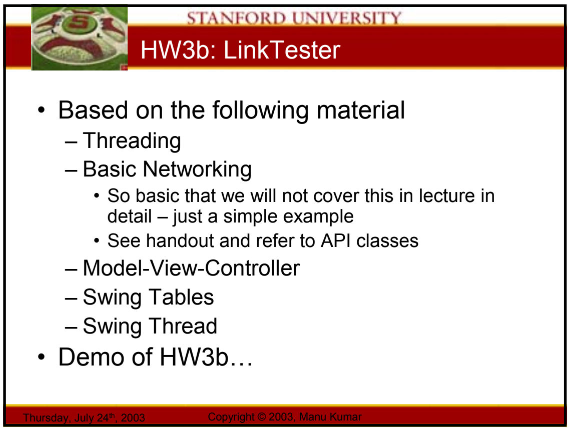
- Last Time
  - Review Introduction to Threading
    - Java threads
      - Simple Thread Example
  - Threading 2
    - Race Conditions
      - Reader/Writer Conflict
      - Writer/Writer Conflict
    - Locking
    - Synchronized Method
      - Synchronized method example

Today

- Homework #3 overview
  - ThreadBank
    - demo
  - LinkTester
    - demo
- Thread Interruption
- Cooperation
  - Wait/notify
- Swing Thread
- Threading conclusions

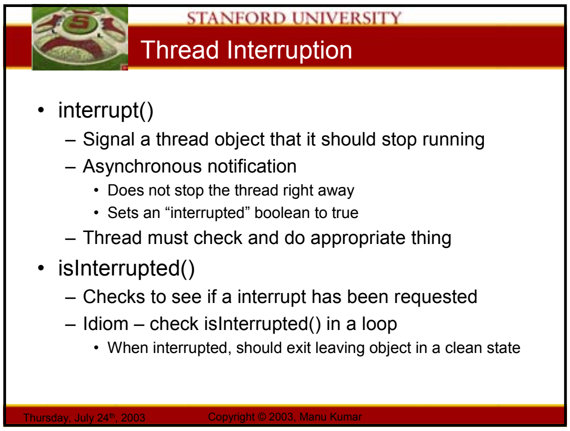
HW3a: ThreadBank

- Small assignment
  - Intended to cover mostly material covered in lecture this week
    - Java Threads
    - Synchronization
    - Cooperation (today)
- Recommendation
  - Finish this assignment this week!
    - It is small, the material if fresh in your mind
    - Part 3b is more involved…



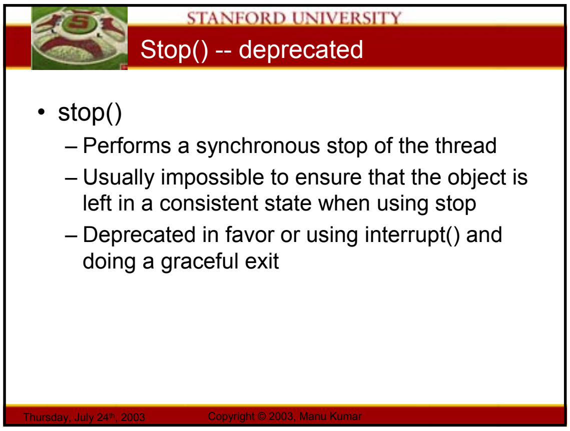
HW3b: LinkTester

- Based on the following material
  - Threading
  - Basic Networking
    - So basic that we will not cover this in lecture in detail – just a simple example
    - See handout and refer to API classes
  - Model-View-Controller
  - Swing Tables
  - Swing Thread
- Demo of HW3b…



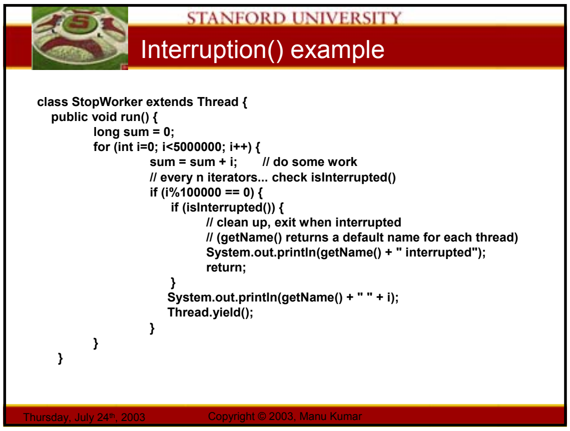
Thread Interruption

- interrupt()
  - Signal a thread object that it should stop running
  - Asynchronous notification
    - Does not stop the thread right away
    - Sets an “interrupted” boolean to true
  - Thread must check and do appropriate thing
- isInterrupted()
  - Checks to see if a interrupt has been requested
  - Idiom – check isInterrupted() in a loop
    - When interrupted, should exit leaving object in a clean state



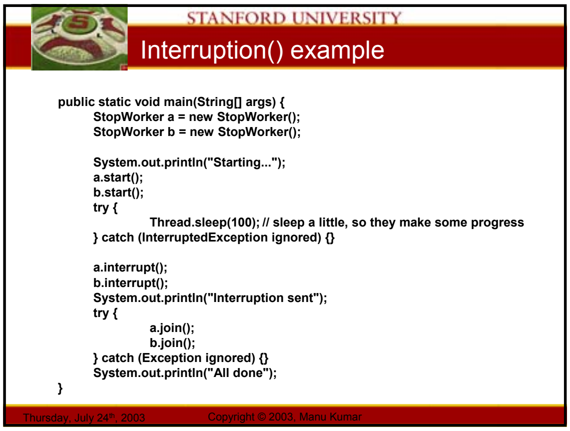
Stop() -- deprecated

- stop()
  - Performs a synchronous stop of the thread
  - Usually impossible to ensure that the object is left in a consistent state when using stop
  - Deprecated in favor or using interrupt() and doing a graceful exit



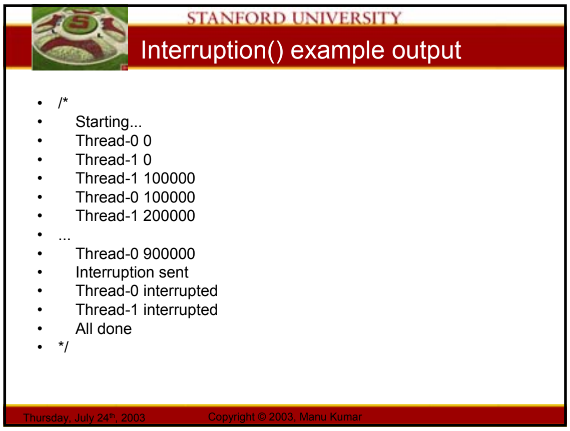
Interruption() example

```
class StopWorker extends Thread {
   public void run() {
            long sum = 0;
            for (int i=0; i<5000000; i++) {
                    sum = sum + i;      // do some work
                    // every n iterators... check isInterrupted()
                    if (i%100000 == 0) {
                        if (isInterrupted()) {
                             // clean up, exit when interrupted
                             // (getName() returns a default name for each thread)
                             System.out.println(getName() + " interrupted");
                             return;
                        }
                        System.out.println(getName() + " " + i);
                        Thread.yield();
                    }
            }
    }
```



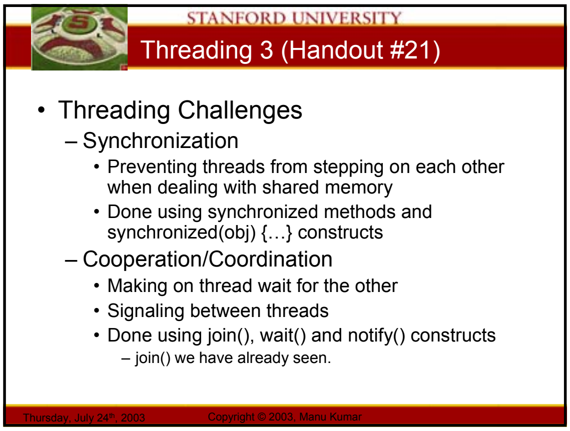
Interruption() example

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

        System.out.println("Starting...");
        a.start();
        b.start();
        try {
                Thread.sleep(100); // sleep a little, so they make some progress
        } catch (InterruptedException ignored) {}

        a.interrupt();
        b.interrupt();
        System.out.println("Interruption sent");
        try {
                a.join();
                b.join();
        } catch (Exception ignored) {}
        System.out.println("All done");
}
```



Interruption() example output

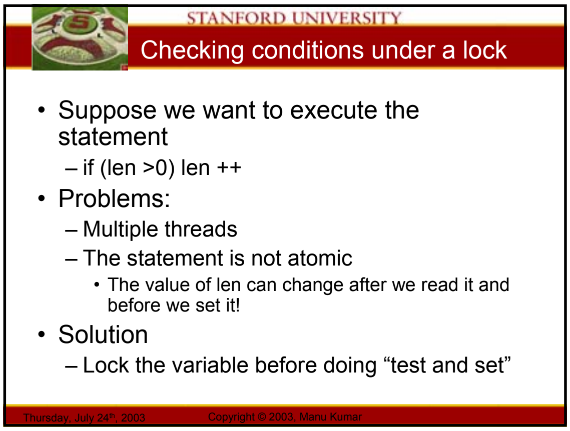
```
• /*
•    Starting...
•    Thread-0 0
•    Thread-1 0
•    Thread-1 100000
•    Thread-0 100000
•    Thread-1 200000
• …
•    Thread-0 900000
•    Interruption sent
•    Thread-0 interrupted
•    Thread-1 interrupted
•    All done
• */
```



STANFORD UNIVERSITY

Threading 3 (Handout #21)

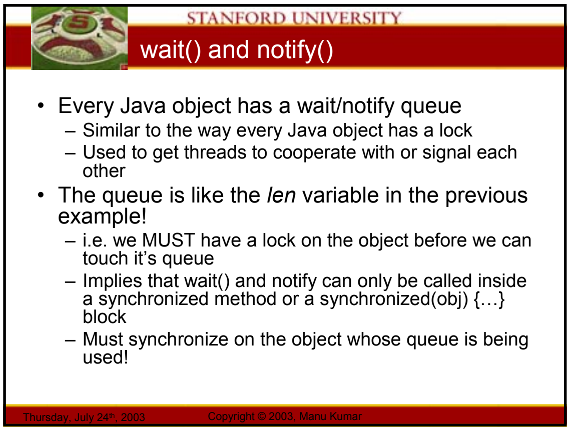
- Threading Challenges
  - Synchronization
    - Preventing threads from stepping on each other when dealing with shared memory
    - Done using synchronized methods and synchronized(obj) {…} constructs
  - Cooperation/Coordination
    - Making on thread wait for the other
    - Signaling between threads
    - Done using join(), wait() and notify() constructs
      - join() we have already seen.



STANFORD UNIVERSITY

Checking conditions under a lock

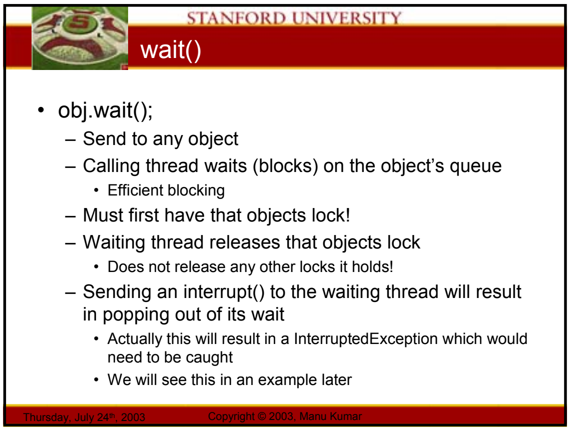
- Suppose we want to execute the statement
  - if (len >0) len ++
- Problems:
  - Multiple threads
  - The statement is not atomic
    - The value of len can change after we read it and before we set it!
- Solution
  - Lock the variable before doing “test and set”



STANFORD UNIVERSITY

wait() and notify()

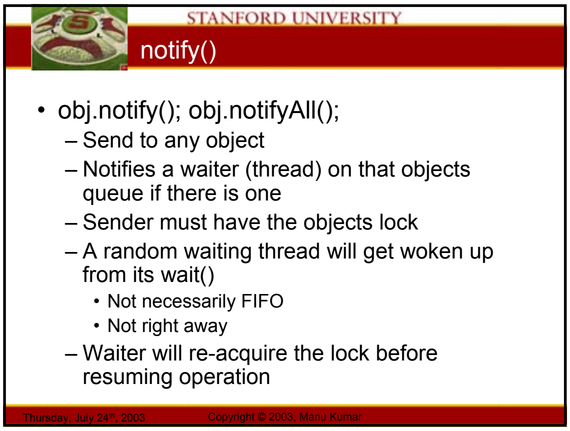
- Every Java object has a wait/notify queue
  - Similar to the way every Java object has a lock
  - Used to get threads to cooperate with or signal each other
- The queue is like the *len* variable in the previous example!
  - i.e. we MUST have a lock on the object before we can touch it’s queue
  - Implies that wait() and notify can only be called inside a synchronized method or a synchronized(obj) {…} block
  - Must synchronize on the object whose queue is being used!



STANFORD UNIVERSITY

wait()

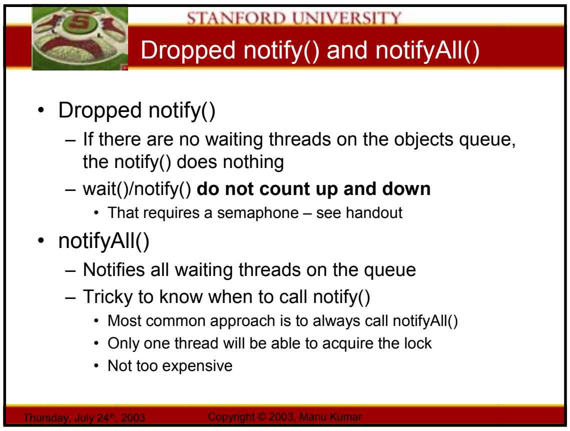
- obj.wait();
  - Send to any object
  - Calling thread waits (blocks) on the object’s queue
    - Efficient blocking
  - Must first have that objects lock!
  - Waiting thread releases that objects lock
    - Does not release any other locks it holds!
  - Sending an interrupt() to the waiting thread will result in popping out of its wait
    - Actually this will result in a InterruptedException which would need to be caught
    - We will see this in an example later



STANFORD UNIVERSITY

notify()

- obj.notify(); obj.notifyAll();
  - Send to any object
  - Notifies a waiter (thread) on that objects queue if there is one
  - Sender must have the objects lock
  - A random waiting thread will get woken up from its wait()
    - Not necessarily FIFO
    - Not right away
  - Waiter will re-acquire the lock before resuming operation



STANFORD UNIVERSITY

Dropped notify() and notifyAll()

- Dropped notify()
  - If there are no waiting threads on the objects queue, the notify() does nothing
  - wait()/notify() **do not count up and down**
    - That requires a semaphore – see handout
- notifyAll()
  - Notifies all waiting threads on the queue
  - Tricky to know when to call notify()
    - Most common approach is to always call notifyAll()
    - Only one thread will be able to acquire the lock
    - Not too expensive

Monitor Exception

- Java.lang.IllegalMonitorStateException: current thread not owner
  - This is the exception thrown if a thread tries a wait/notify on a object without first holding its lock!
  - You will get these while writing your code!
    - Make sure you are synchronizing on the correct object before calling wait or notify!

While (cond) wait() idiom

- When the waiting thread is woken up from the wait it holds the lock
  - But the condition it was waiting for may not be true any more!
  - It may have become false again in between when the notify happened and when the wait/return happened
  - Necessary to check the condition again before proceeding further
- Idiom

```
try {
        while (<condition>) wait();
} catch (InterruptedException e) {}
```

Wait/notify example

- Producer/Consumer problem with wait/notify
  - "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.

Wait/notify example

- Strategy:
  - The AddRemove object is the common object between the threads
    - they use its lock and its wait/notify queue.
  - add() does a notify() when it adds an element
  - remove() does a wait() if there are no elements
  - Eventually, an add() thread will put an element in and do a notify()
  - Each adder adds 10 times, and each remover removes 10 times, so it balances in the end.

Wait/Notify example code

- Code walk through
  - In emacs…

Dropped notify() problem…

- Notify() does not count the number of notifies!
  - It is instantaneous
    - If there are waiters waiting they will be signaled
    - If a waiter comes after the notify, it is not signaled
- wait/notify() is simpler than a semaphore
  - Semaphores count
    - Classic CS locking construct
    - Possible to build semaphore using wait/notify

DroppedNotify Example

- Code walkthrough
  - In emacs…

Swing/GUI Threading

- Problem: Swing vs. Threads
  - Modifying the GUI state while it is being drawn
    - Typical reader/writer conflict problem
  - Example
    - paintComponent() while another thread changes the component geometry
    - Send mouseMoved() notification to an object, but another thread deletes the object!

Solution: Swing Thread

- Swing Thread: a.k.a One Big Lock!
  - One official designated “Swing thread”
  - Does all Swing/GUI notifications using the Swing thread, one at a time
    - paintComponent() – always on Swing Thread
    - All notifications: action events, mouse events – sent on the Swing Thread
  - System keeps a queue of “Swing jobs”
    - When the Swing Thread is done with its current job it moves on to the next one
  - Only the Swing Thread is allowed to edit the state of the GUI
    - Since the Swing thread is the only one allowed to touch the Swing state there is in effect a big lock over all the Swing State

Programmer Rules

- On the swing thread – edit ok
  - Allowed to edit the Swing state when you are on the Swing Thread
    - Container.add(), setPreferredSize(), setLayout()
- Don’t hog the Swing Thread
  - Do not to time-consuming operations on the Swing Thread
    - If you hold the Swing Thread, no Swing/GUI processing will happen till you release it!
  - Fork off a worker thread to do a time-consuming operation
- Not on the Swing Thread – no edit
  - A thread which is not the swing thread may not send messages that edit the Swing state
  - Use invokeLater() to run code on the swing thread
  - Repaint() is an exception – since it only schedules a call to paintComponent() which is called by the Swing Thread
  - Another exception is modifying state before the component has been made visible
    - For example in a constructor

Swing Thread: Results

- In your notifications (paintComponent(), actionPerformed()) you are on the Swing Thread
  - Feel free to send Swing messages
- There is only one Swing Thread
  - When you have it, no other Swing activity is happening
    - Do not hog the Swing Thread

SwingUtilities

- Built in utility method to allow you to “post” some code to the Swing Thread to run later
  - Uses Runnable interface
    - public void run()
  - SwingUtilities.invokeLater(Runnable)
    - Queue up the given runnable
    - Will execute when the Swing Thread gets to it
  - SwingUtilities.invokeAndWait(Runnable)
    - Same as above, but also block current thread till the runnable has completed

SwingUtilities Client Example

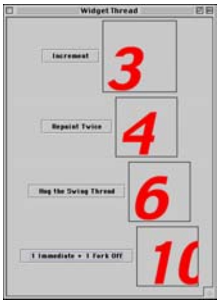
```
class MyFrame extends JFrame {
    private JLabel label;
    // Typical GUI code down here creates and starts the worker
    public MyFrame() {
        // standard Frame ctor stuff, create buttons...
        button.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Worker worker = new Worker();
                worker.start();
            }
        });
        …
    }
```

SwingUtilities Client Example

```
class Worker extends Thread {
    public void run() {
        // The worker does some big computation
        final String answer = <something>;
        // We want to call setText() to send the answer to the GUI.
        // We must go through SwingUtilities.invokeLater()
        SwingUtilities.invokeLater(
            new Runnable() {   // create a runnable on the fly
                public void run() {
                    label.setText(answer);
                }
            }
        );
    }
  }
}
```

SwingThread Demo

- Demo and code walkthrough
  - In emacs…



Threading Conclusions

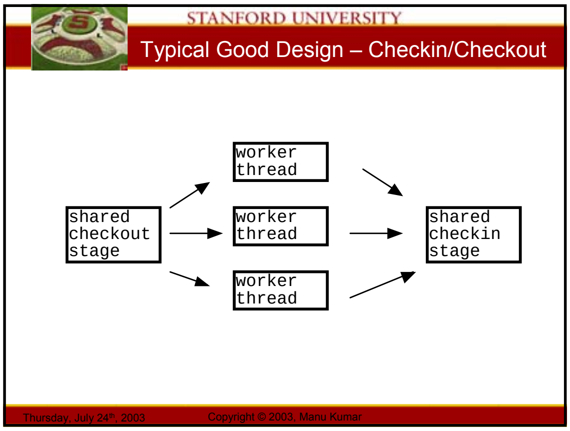
- Java uses an OOP Concurrency style
  - Objects store state
  - Getters and setters are synchronized
  - Intuitive extension to how threading is handled
    - Not just a translation from C/C++
- Compile Time “Structured” style
  - Lock/unlock structure is specified at compile time
    - synchronized(obj) {…}
    - Impossible to write code where lock/unlock don’t balance
      - Much better than lock() and unlock() constructs in other languages
- Java uses “monitor” style locking
  - Not as flexible, but easier and less error prone

When to use Threading?

- Hardware
  - To take advantage of increasingly parallel hardware
- GUI
  - To keep the GUI responsive
- Networking
  - Use thread to support multiple connections
  - Speed up by pipelining slow operations

In general…

- Single Threaded is easier!
  - There are cases when this is the best use of your time
- Design for concurrency
  - By default, do not put much effort in to making your class support concurrency
    - Should only be deliberately added when it makes sense
    - It is not trivial to support concurrency
      - Performance tradeoff
      - Complexity tradeoff



STANFORD UNIVERSITY

Typical Good Design – Checkin/Checkout

shared checkout stage

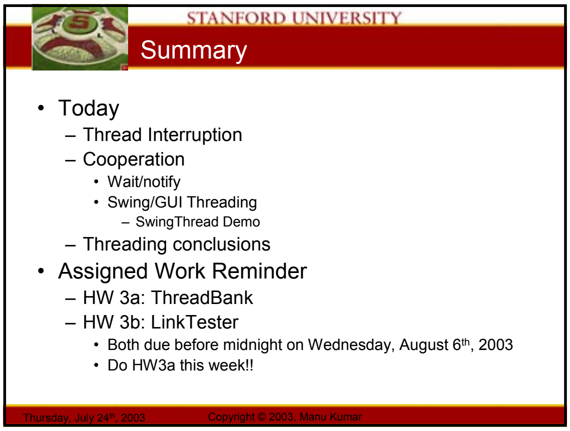
worker thread

worker thread

worker thread

shared checkin stage

Thursday, July 24th, 2003



STANFORD UNIVERSITY

Summary

- Today
  - Thread Interruption
  - Cooperation
    - Wait/notify
    - Swing/GUI Threading
      - SwingThread Demo
  - Threading conclusions
- Assigned Work Reminder
  - HW 3a: ThreadBank
  - HW 3b: LinkTester
    - Both due before midnight on Wednesday, August 6th, 2003
    - Do HW3a this week!!

Thursday, July 24th, 2003