

# Lecture 38: Parallelism & Concurrency in Java

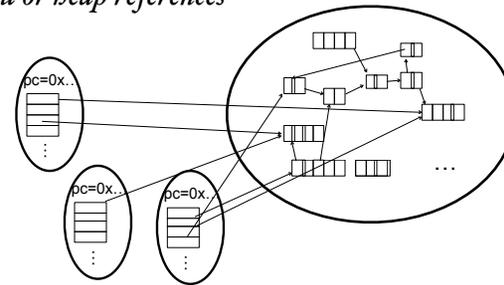
CSC 131  
Fall, 2014

Kim Bruce

## Shared Memory

Threads, each with own unshared call stack and current statement (pc for “program counter”) local variables are numbers/null or heap references

Heap for all objects and static fields

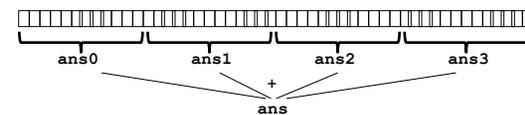


## Parallel Programming in Java

- Creating a thread:
  1. Define a class C extending Thread
    - Override public void run() method
  2. Create object of class C
  3. Call that thread's start method
    - Creates new thread and starts executing run method.
    - Direct call of run won't work, as just be a normal method call

- *Alternatively, define class implementing Runnable, create thread w/it as parameter, and send start message*

## Parallelism Idea



- Example: Sum elements of an array
  - Use 4 threads, which each sum 1/4 of the array
- Steps:
  - Create 4 thread objects, assigning each their portion of the work
  - Call start() on each thread object to actually run it
  - Wait for threads to finish
  - Add together their 4 answers for the final result

## First Attempt

```
class SumThread extends Thread {
    int lo; int hi; int[] arr; //fields to know what to do
    int ans = 0; // for communicating result
    SumThread(int[] a, int l, int h) { ... }
    public void run(){ ... }
}

int sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){// do parallel computations
        ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        ts[i].start(); // use start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}
```

*What's wrong?*

## Correct Version

```
class SumThread extends Thread {
    int lo, int hi, int[] arr; //fields to know what to do
    int ans = 0; // for communicating result
    SumThread(int[] a, int l, int h) { ... }
    public void run(){ ... }
}

int sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){// do parallel computations
        ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        ts[i].start(); // start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
    return ans;
}

See program ParallelSum
```

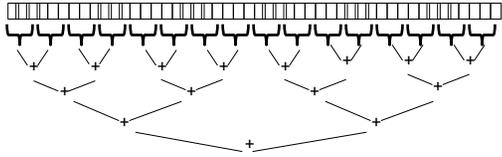
## Thread Class Methods

- void start(), which calls void run()
- void join() -- blocks until receiver thread done
- Style called fork/join parallelism
  - Code on previous slide generates error message as join can throw exception  
InterruptedException
- Some memory sharing: arr field
- Later learn how to protect using synchronized.

## Actually not so great.

- If do timing, it's slower w/ small arrays than sequential!!!
- Want code to be reusable and efficient as core count grows.
  - At minimum, make #threads a parameter.
- Want to effectively use processors available *now*
  - Not being used by other programs
  - Can change while your threads running

## Divide & Conquer



- Divide in half, w/ one thread per half.
  - Each half further subdivided w/ new threads, etc. until down to single elements
  - Depth is  $O(\log n)$ , which is optimal
  - Then total time w/numProc processors

$$O(n/\text{numProc} + \log n)$$

*straight-line code cost  
in step 1*

*each layer is  $O(1)$  in parallel*

## In practice

- Creating all threads and communication swamps savings so
  - use sequential cutoff about 1000
  - Don't create two recursive threads
    - one new and reuse old.
    - Cuts number of threads in half.

*EfficientDivideConquerParallelSum*

## Even Better

- Java threads too heavyweight -- space and time overhead.
- ForkJoin Framework solves problems
- Added in Java 7.

## To Use Library

- Create a ForkJoinPool
- Instead of subclass Thread, subclass RecursiveTask<V> (or RecursiveAction)
- Override compute, rather than run
- Return answer from compute rather than instance vble
- Call fork instead of start
- Call join that returns answer
- To optimize, call compute instead of fork (*rather than run*)
- See *ForkJoinFrameworkDivideConquerParallelSum*

# Handling Concurrency in Java

*See ATM example*

## Synchronized blocks

- Control access w/ synchronized blocks:
  - `synchronized(someObj) { ... }`
  - Must hold lock to access. Release when exit.
- Synchronized methods:
  - Implicitly use “this” as lock on method body

## Shared Variables

- Variables read/written by more than one process are vulnerable to race conditions.
  - Even `++n` is vulnerable, as not atomic
  - But there are “atomic” types like `AtomicInteger`
- If multiple threads access the same mutable state variable you have two options:
  - Make the state variable immutable
  - Use synchronization whenever accessing the state variable

## Shared Variables

- Visibility of changes:
  - If one thread executes synchronized block, and then another thread enters a block with same lock, then current values of variables accessible by first are visible to second when acquires lock
  - Without synchronization, no guarantees!
    - May reorder, may be in cache or register or ...
- If synchronization not necessary, then label vble as `volatile` to force changes to be visible

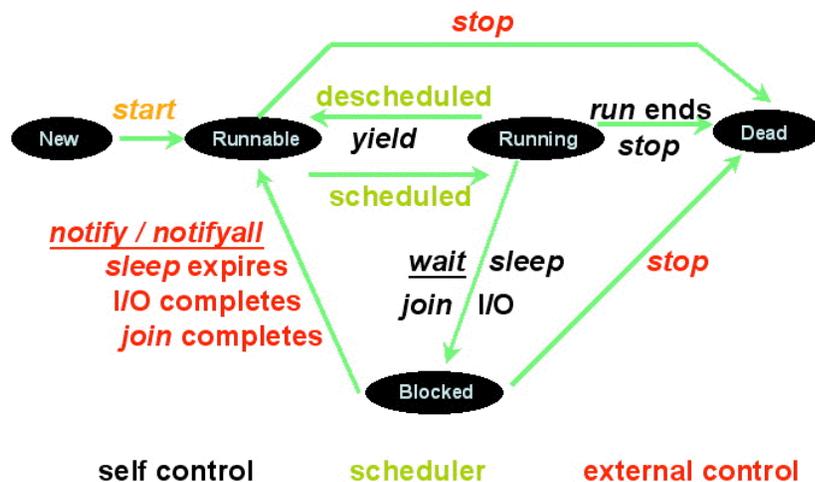
## Conditional Waiting

- Every object has a wait set
- `wait()`: release lock & pause until another thread calls `notify` or `notifyAll`.
- `notify()`, `notifyAll()`: wake up waiting threads, which try to grab lock
  - *Can only be used in synchronized code*
  - *Notify wakes up single thread -- arbitrary choice*
  - *NotifyAll wakes up all waiting threads*
  - *Much better than busy-waiting (spin-locks)*

## Thread States in Java

- New -- declared, but not yet started
- Runnable -- ready to run
- Running -- currently running
- Blocked -- on I/O, wait on monitor, sleep, join
- Dead -- run has ended

## Concurrency in Java



## Monitor in Java

```
public class BoundedBuffer {
    protected int numSlots;
    private int[] buffer;
    private int takeOut = 0, putIn = 0;
    private int count=0;

    public BoundedBuffer(int numSlots) {
        if(numSlots < 0) {
            throw new IllegalArgumentException(
                "numSlots <= 0");
        }
        this.numSlots = numSlots;
        buffer = new int[numSlots];
    }
}
```

*From Mitchell, hwk 14.7*

```
public synchronized void put(int value)
    throws InterruptedException {
    while (count == numSlots) wait();
    buffer[putIn] = value;
    putIn = (putIn + 1) % numSlots;
    count++;
    notifyAll();
}

public synchronized int get()
    throws InterruptedException {
    while (count == 0) wait();
    int value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    count--;
    notifyAll();
    return value;
}
}
```