Lecture 2 - Concurrency and Multithreading

Outline

- Appreciate the (increasing) importance of parallel programming
- Understand fundamental concepts:
  - Parallelism, threads, multi-threading, concurrency, locks, etc.
- See some basics of this is done in Java
- See some common uses:
  - Divide and conquer, e.g. mergesort
  - Worker threads in Swing

Background

- An area of rapid change!
  – 1990s: parallel computers were $$$$
  – Now: 4 core machines are commodity
- Variations between languages
- Evolving frameworks, models, etc.
  – E.g. Java's getting Fork/Join since Java 1.7
  – MAP/REDUCE

(Multi)Process vs (Multi)Thread

- Assume a computer has one CPU
- Can only execute one statement at a time
  - Thus one program at a time
- Process: an operating-system level "unit of execution"
- Multi-processing
  - Op. Sys. "time-slices" between processes
  - Computer <u>appears</u> to do more than one program (or background process) at a time

Multicore vs Multithreading

- Modern CPUs have multiple cores that can execute multiple processes at the same time
- Multiple threads can run even in a single core CPU
- Multithreading can utilize the multiple cores
- Still, it is the job of the operating system to schedule and run the parallel tasks

Advantages of Multithreading

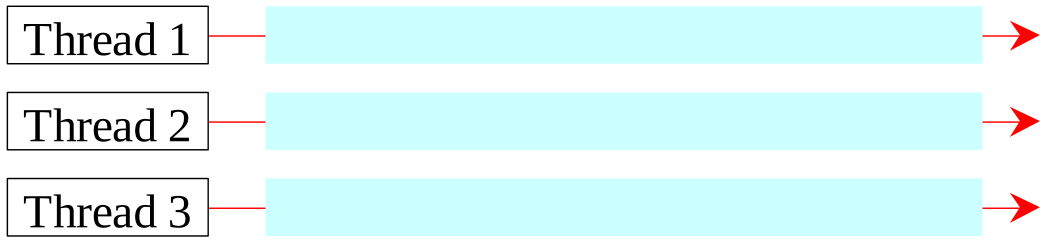
- Reactive systems – constantly monitoring
- More responsive to user input – GUI application can interrupt a time-consuming task
- Server can handle multiple clients simultaneously
- Can take advantage of parallel processing

Multithreading

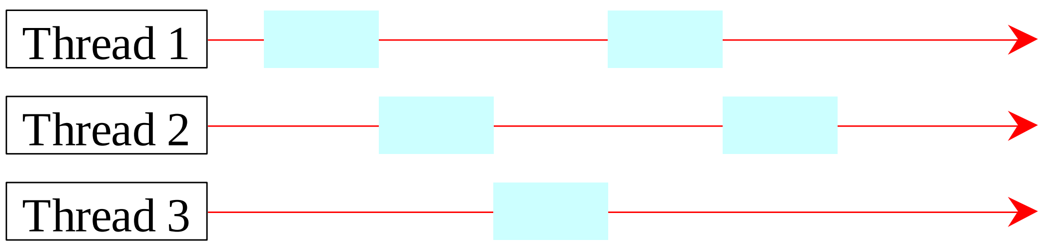
- Different processes do not share memory space.
- A thread can execute concurrently with other threads within a single process.
- All threads managed by the JVM share memory space and can communicate with each other.

Threads Concept

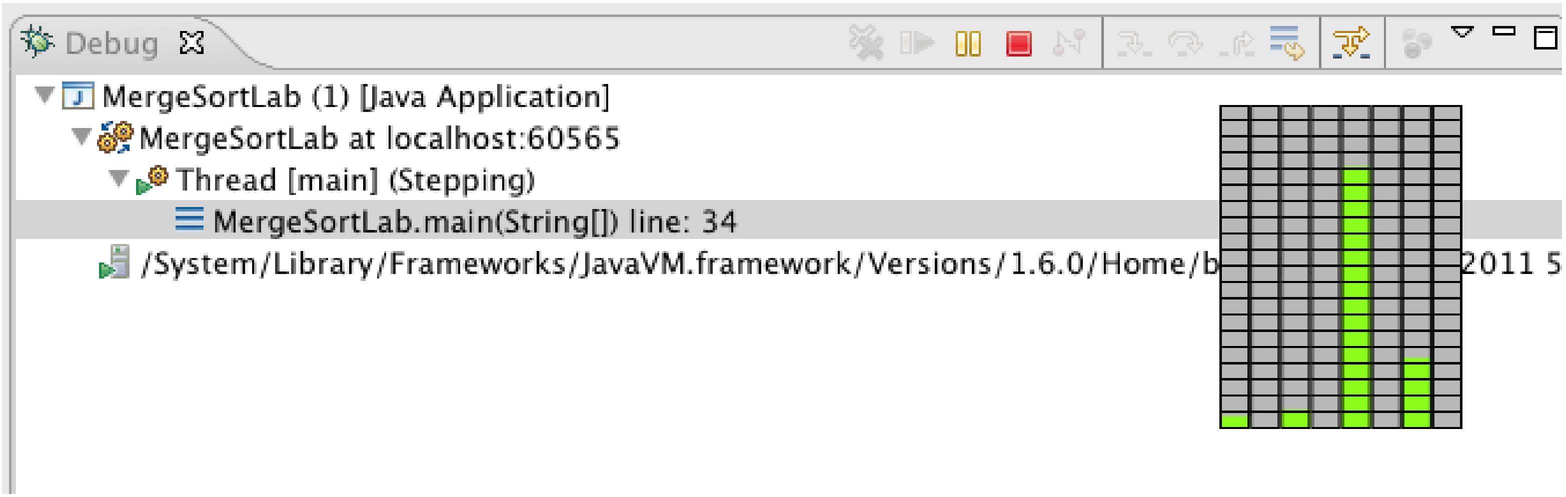
Multiple threads on multiple CPUs

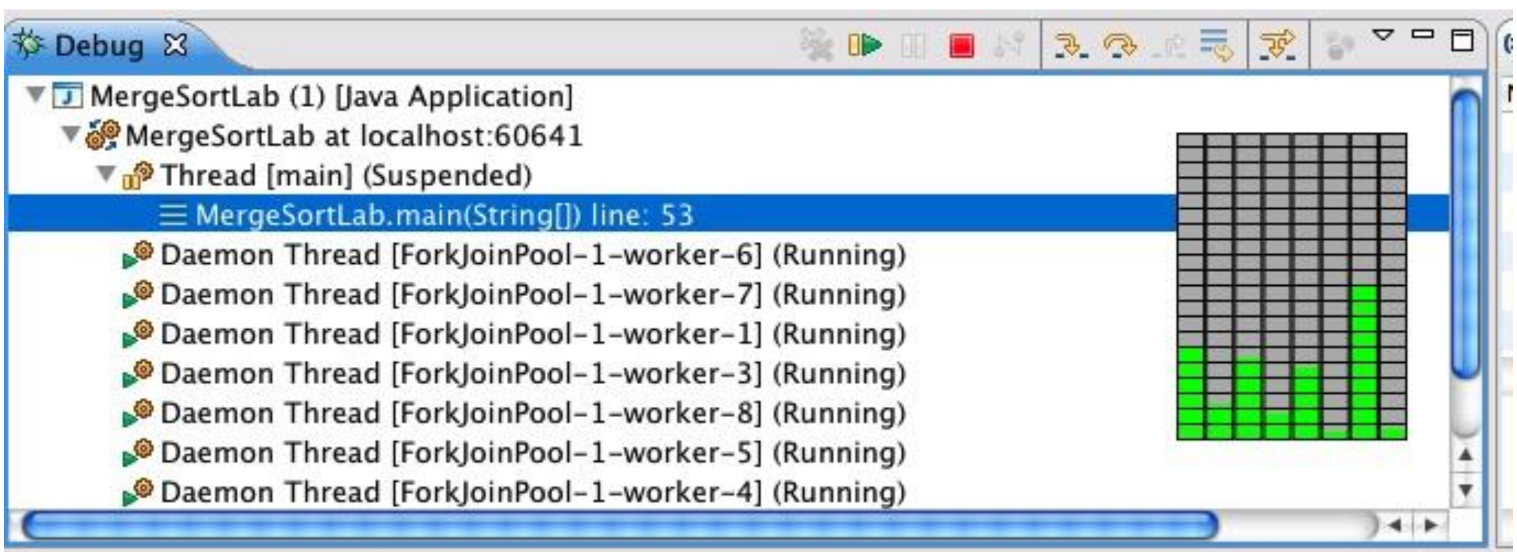


Multiple threads sharing a single CPU



Screenshots: For single- and multi-threaded Mergesort: Threads in Eclipse Debug window, and Mac's CPU usage display

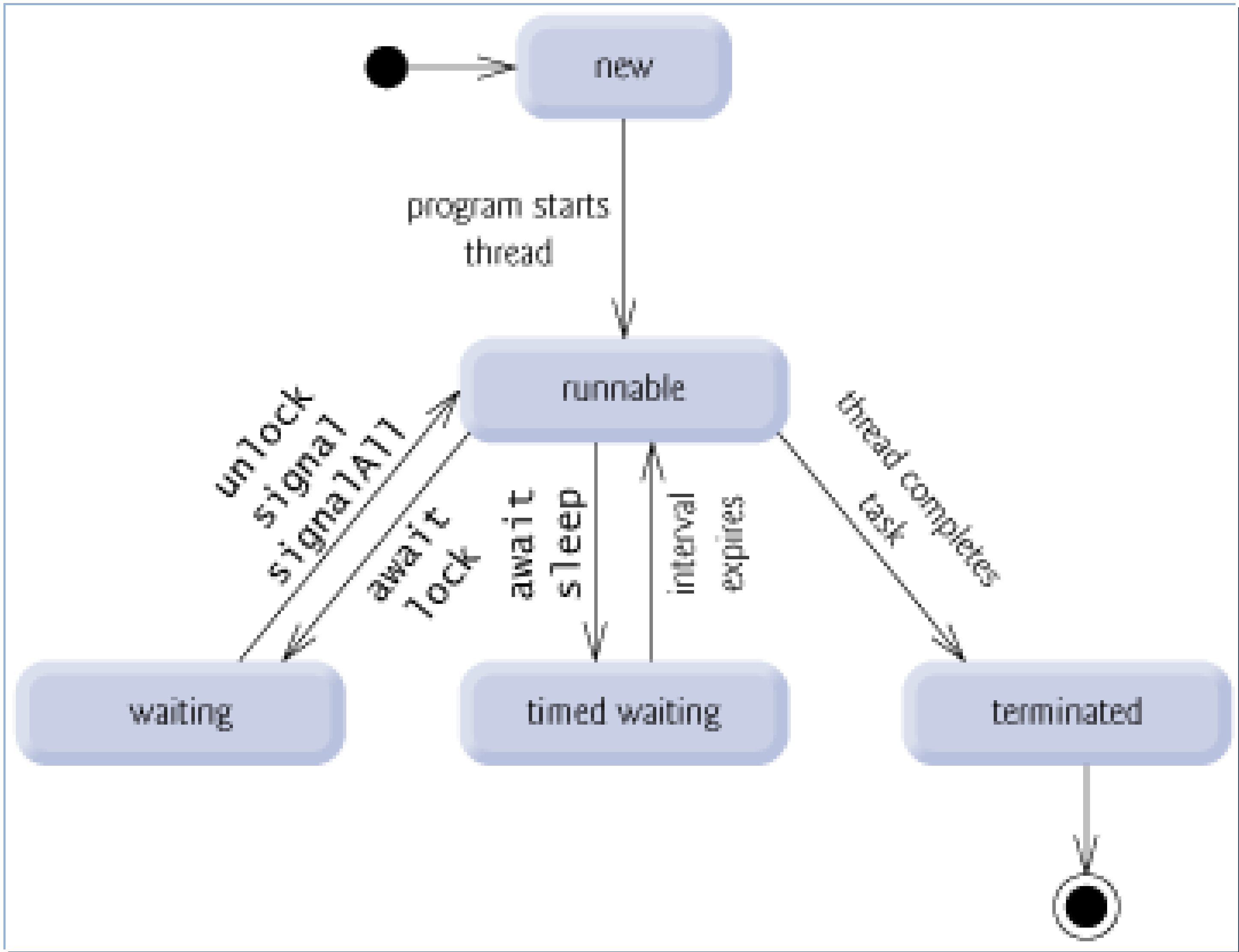
- 



Tasks and Threads

- **Thread**: “a thread of execution”
  - “Smaller”, “lighter” than a **process**
  - smallest unit of processing that can be scheduled by an operating system
  - Has its own run-time call stack, copies of the CPU’s registers, its own program counter, etc.
  - Process has its own memory address space, but threads share one address space
- A single program can be multi-threaded
  - Time-slicing done just like in multiprocessing
  - Repeat: the threads share the same memory

Thread States



Task

- A **task** is an abstraction of a series of steps
  - Might be done in a separate thread
  - Parallelizable
- In Java, there are a number of classes / interfaces that basically correspond to this
  - Example (details soon): Runnable
    - work done by method run()

Java: Statements → Tasks

- Consecutive lines of code:

```
Foo tmp = f1;
f1 = f2;
f2 = tmp;
```

- A method:

```
swap(f1, f2);
```

- A “task” object:

```
SwapTask task1= new SwapTask(f1,f2);
task1.run();
```

Why a task <u>object</u>?

- Actions, functions vs. objects. What’s the difference?

Why a task object?

- Actions, functions vs. objects. What's the difference?
- Objects:
  - Are persistent. Can be stored.
  - Can be created and then used later.
  - Can be attached to other things. Put in Collections.
  - Contain state.
- Functions:
  - Called, return (not permanent)

Java Library Classes for Concurrency

- For task-like things:
  - Runnable, Callable
  - SwingWorker, RecursiveAction, etc.
- Thread class
- Managing tasks and threads
  - Executor, ExecutorService
  - ForkJoinPool
- In Swing
  - The Event-Dispatch Thread
  - SwingUtilities.invokeLater()

Java Thread Classes and Methods

- Java has some “primitives” for creating and using threads
  - Most sources teach these, but in practice they’re hard to use well
  - Now, better frameworks and libraries make using them directly less important.
- But let’s take a quick look

Java's Thread Class

- Class Thread: it's method run() does its business when that thread is run
- But you never call run(). Instead, you call start() which lets Java start it and call run()
- To use Thread class directly (not recommended now):
  - define a subclass of Thread and override run() – not recommended!
  - Create a task as a Runnable, link it with a Thread, and then call start() on the Thread.
    - The Thread will run the Runnable's run() method.

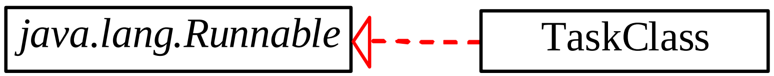
Creating a Task and Thread

- Again, the first of the two “old” ways
- Get a thread object, then call start() on that object
  - Makes it available to be run
  - When it’s time to run it, Thread’s run() is called
- So, create a thread using inheritance
  - Write class that extends Thread, e.g. MyThread
  - Define your own run()
  - Create a MyThread object and call start() on it
- Not good design!

Runnables and Thread

- Use the “task abstraction” and create a class that implements Runnable interface
  - Define the run() method to do the work you want
- Now, two ways to make your task run in a separate thread
  - First way:
    - Create a Thread object and pass a Runnable to the constructor
    - As before, call start() on the Thread object
  - Second way: hand your Runnable to a “thread manager” object

Creating Tasks and Threads




```java
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

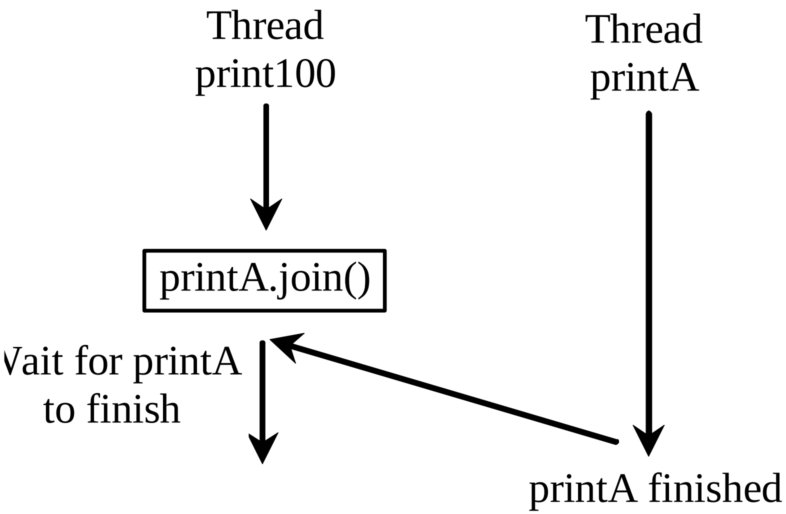
```java
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Join

- The **`Thread`** class defines various primitive methods you could not implement on your own
  - For example: **`start`**, which calls **`run`** in a new thread
- The **`join()`** method is one such method, essential for coordination in this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning its **`run`** returns)
  - E.g. in method foo() running in "main" thread, we call:
    myThread.start();   myThread.join();
  - Then this code waits ("blocks") until myThread's run() completes
- This style of parallel programming is often called "fork/join"
  - Warning: we'll soon see a library called "fork/join" which simplifies things. In that, you never call join()

Join

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



Threading in Swing

- Threading matters a lot in Swing GUIs
  - You know: main’s thread ends “early”
  - JFrame.setvisible(true) starts the “GUI thread”
- Swing methods run in a separate thread called the **Event-Dispatching Thread** (EDT)
  - Why? GUIs need to be responsive quickly
  - Important for good user interaction
- But: slow tasks can block the EDT
  - Makes GUI seem to hang
  - Doesn’t allow parallel things to happen

Thread Rules in Swing

- All operations that update GUI components <u>must</u> happen in the EDT
  - These components are not thread-safe (later)
  - SwingUtilities.invokeLater(Runnable r) is a method that runs a task in the EDT when appropriate
- But execute slow tasks in separate *worker threads*
- To make common tasks easier, use a SwingWorker task

SwingWorker

- A class designed to be extended to define a task for a worker thread
  - Override method **doInBackground()**
    This is like run() – it’s what you want to do
  - Override method **done()**
    This method is for updating the GUI afterwards
    - It will be run in the EDT
- Note you can get interim results too

Java ForkJoin Framework

- Designed to support a common need
  - Recursive divide and conquer code
  - Look for small problems, solve without parallelism
  - For larger problems
    - Define a task for each subproblem
    - Library provides
      - a Thread manager, called a ForkJoinPool
      - Methods to send your subtask objects to the pool to be run, and your call waits until their done
      - The pool handles the multithreading well

The ForkJoinPool

- The “thread manager”
  - Used when calls are made to RecursiveTask’s methods fork(), invokeAll(), etc.
  - When created, knows how many processors are available
  - Pretty sophisticated
    - “Steals” time from threads that have nothing to do

Overview of How To

- Create a ForkJoinPool "thread-manager" object
- Create a task object that extends RecursiveTask
  - Create a task-object for entire problem and call invoke(task) on your ForkJoinPool
- Your task class' compute() is like Thread.run()
  - It has the code to do the divide and conquer
  - First, it must check if small problem – don't use parallelism, solve without it
  - Then, divide and create >1 new task-objects. Run them:
    - Either with invokeAll(task1, task2, …). Waits for all to complete.
    - Or calling fork() on first, then compute() on second, then join()

Same Ideas as Thread But...

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a **`ForkJoinPool`**)

Don’t subclass **`Thread`**	Do subclass **`RecursiveAction<V>`**
Don’t override **`run`**	Do override **`compute`**
Don’t call **`start`**	Do call **`invoke, invokeAll, fork`**
Don’t just call **`join`**	Do call **`join`** which returns answer
or	
	Do call **invokeAll** on multiple tasks

Mergesort Example

- Top-level call. Create "main" task and submit

```
public static void mergeSortFJRecur(Comparable[] list, int first,
   int last) {
   if (last - first < RECURSE_THRESHOLD) {
       MergeSort.insertionSort(list, first, last);
      return;
   }
   Comparable[] tmpList = new Comparable[list.length];
   threadPool.invoke(new SortTask(list, tmpList, first, last));
}
```

Mergesort's Task-Object Nested Class

```
static class SortTask extends RecursiveAction {
    Comparable[] list;
    Comparable[] tmpList;
    int first, last;
    public SortTask(Comparable[] a, Comparable[] tmp,
      int lo, int hi) {
        this.list = a;      this.tmpList = tmp;
        this.first = lo;   this.last = hi;
    }
// continued next slide
```

compute() Does Task Recursion

```
protected void compute() {   // in SortTask, continued from previous slide
    if (last - first < RECURSE_THRESHOLD)
        MergeSort.insertionSort(list, first, last);
    else {
        int mid = (first + last) / 2;
        // the two recursive calls are replaced by a call to
invokeAll
        SortTask task1 = new SortTask(list, tmpList, first, mid);
        SortTask task2 = new SortTask(list, tmpList, mid+1, last);
        invokeAll(task1, task2);
        MergeSort.merge(list, first, mid, last);
    }
}
```

Nice to Have a Thread "Manager"

- If your code is responsible for creating a bunch of tasks, linking them with Threads, and starting them all, then you have muchto worry about:
  - What if you start too many threads? Can you manage the number of running threads?
  - Enough processors?
  - Can you shutdown all the threads?
  - If one fails, can you restart it?

Executors

- An Executor is an object that manages running tasks
  - Submit a Runnable to be run with Executor’s execute() method
  - So, instead of creating a Thread for your Runnable and calling start() on that, do this:
    - Get an Executor object, say called exec
    - Create a Runnable, say called myTask
    - Submit for running: exec.execute(myTask)

How to Get an Executor

- Use static methods in Executors library.
- Fixed “thread pool”: at most N threads running at one time

```
Executor exec =
  Executors.newFixedThreadPool(MAX_THREADS);
```

- Unlimited number of threads

```
Executor exec =
  Executors.newCachedThreadPool();
```

Summary So Far

- Create a class that implements a Runnable to be your "task object"
  - Or if ForkJoin framework, extend RecursiveTask
- Create your task objects
- Create an Executor
  - Or a ForkJoinPool
- Submit each task-object to the Executor which starts it up in a separate thread

Concurrency and Synchronization

- **Concurrency:**
  Multiple-threads/Processes accessing shared data

- **Synchronization:**
  Methods to manage and control concurrent access to shared data by multiple-threads

Possible Bugs in Multithreaded Code

- Possible bug #1

```
i=1; x=10;   x = i + x;   // x could be 12 here
```

- Possible bug #2

```
if ( ! myList.contains(x) )
    myList.add(x);   // x could be in list twice
```

- Why could these cause unexpected results?

How 1 + 10 might be 12

- **Thread 1 executes:**

  (x is 10, i is 1)

  - **Get i (1) into register 1**
  - **Get x (10) into its register 2**

    (other thread has CPU)
  - **Add registers**
  - **Store result (11) into x**

    (x is now 11)

    (other thread has CPU)

    (other thread has CPU)
  - ***Do next line of code***

    (x changes to 12 even though no code in this thread has touched x)

- **Thread 2 executes:**

  (x is 10, i is 1)

  (other thread has CPU)

  (other thread has CPU)

  - **Get i (1) into its register 1**

    (other thread has CPU)

    (other thread has CPU)
  - **Get x (11) into is register 2**
  - **Add registers**
  - **Store result (12) into x**

    (x is now 12)

Synchronization

- Understand the issue with concurrent access to shared data?
  - Data could be a counter (int) or a data structure (e.g. a Map or List or Set)
- A **race condition**: Two threads will access something. They "compete" causing a problem
- A **critical section**: a block of code that can only be safely executed by one thread at a time
- A lock: an object that is "held" by one thread at a time, then "released"

Synchronized Methods

- Common situation: all the code in a method is a critical section
  - I.e. only one thread at a time should execute that method
  - E.g. a getter or setter or mutator, or something that changes shared state info (e.g. a Map of important data)
- Java makes it easy: add synchronized keyword to method signature. E.g.

```
public synchronized void update(…) {
```

```java
public class Counter {
        private int counter;
        public synchronized void increment() {
            counter++;
        }

        public int read() {
            return counter;
         }
    }
```

Synchronization using Locks

- Any object can serve as a lock
  - Separate object: `Object myLock = new Object();`
  - Current instance: the this object
- Enclose lines of code in a *synchronized* block

```
synchronized(myLock) {
     // code here
}
```

- More than one thread could try to execute this code, but one acquires the lock and the others "block" or wait until the first thread releases the lock
- More fine grained than method synchronization (more prone to errors)

Summary So Far

- Concurrent access to shared data
  - Can lead to serious, hard-to-find problems
  - E.g. race conditions
- The concept of a lock
- Synchronized blocks of code or methods
  - One thread at a time
  - While first thread is executing it, others block

Some Java Solutions

- There are some synchronized collections

- Classes like AtomicInteger
  - Stores an int
  - Has methods to operate on it in a thread-safe manner

    int getAndAdd(int delta) instead of i=i+1

Volatile keyword

- Compilers cache variable values from the memory in the registers for faster performance
- This can sometimes lead to errors in execution in a multithreaded program
- Volatile keyword ensures that each variable update is visible to all threads
- However, it does not ensure atomicity of operations (which has to be handled separately)

Cooperation/Communication Among Threads

- Conditions can be used for communication among threads.

- A thread can specify what to do under a certain condition.
- newCondition() method of Lock object.

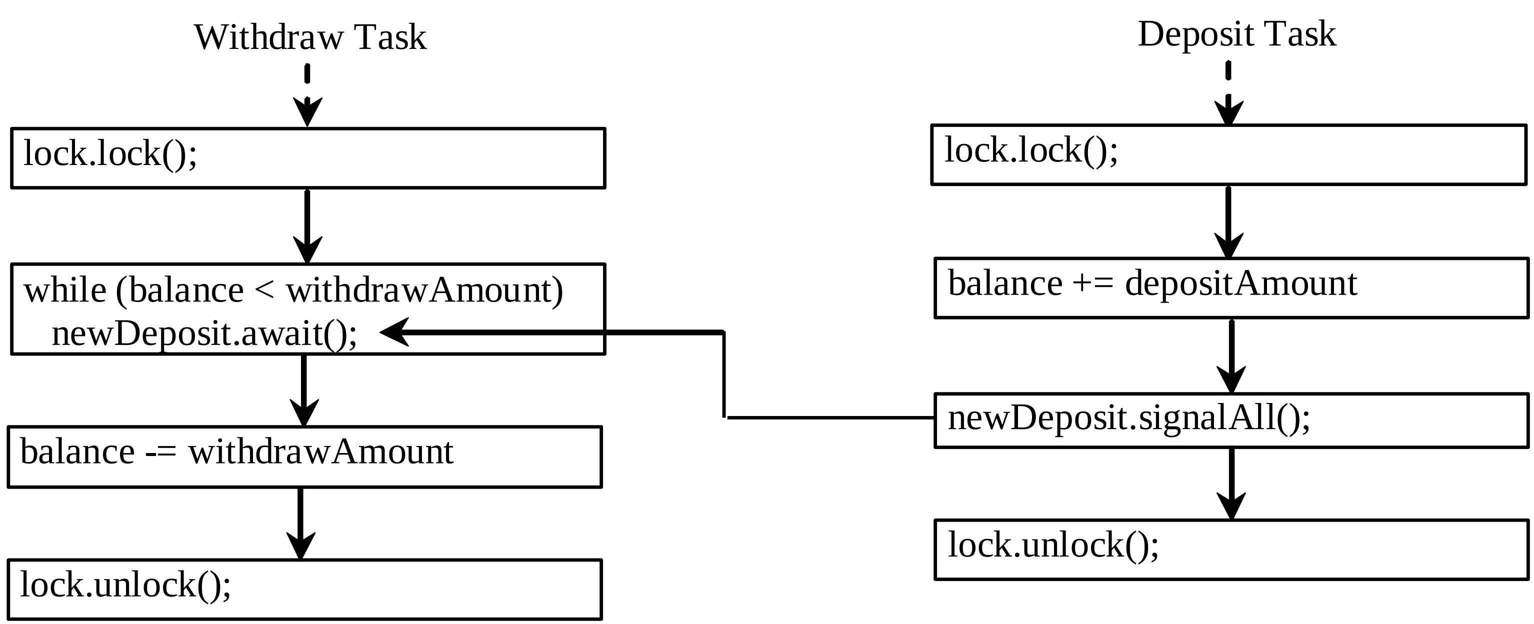
- Condition methods:
  - await() current thread waits until the condition is signaled
  - signal() wakes up a waiting thread
  - signalAll() wakes all waiting threads

«interface» *java.util.concurrent.Condition*	
+*await(): void*	Causes the current thread to wait until the condition is signaled.
+*signal(): void*	Wakes up one waiting thread.
+*signalAll(): Condition*	Wakes up all waiting threads.

Cooperation Among Threads

- Lock with a condition to synchronize operations: <u>newDeposit</u>
- If the balance is less than the amount to be withdrawn, the withdraw task will wait for the <u>newDeposit</u> condition.
- When the deposit task adds money to the account, the task signals the waiting withdraw task to try again.

- Interaction between the two tasks:



Monitor objects

- A *monitor* is an object with mutual exclusion and synchronization capabilities.
- Only one thread can execute a method at a time in the monitor.
- A thread enters the monitor by acquiring a lock (<u>synchronized</u> keyword on method / block) on the monitor and exits by releasing the lock.
- A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.
- *Any object can be a monitor*. An object becomes a monitor once a thread locks it.

wait(), notify(), and notifyAll()

- Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

- The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Example: Using Monitor

Task 1

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

   // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
```

resume

Task 2

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or  anObject.notifyAll();
  ...
}
```

Interrupting threads

- Interrupt()

If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

- The isInterrupt() method tests whether the thread is interrupted.

- Can be used to gracefully stop a waiting thread

```
class TestInterruptingThread1 extends Thread{
public void run(){
        try{
                Thread.sleep(1000);
                System.out.println("task");
        }catch(InterruptedException e){
                throw new RuntimeException("Thread interrupted..."+e);
        }
  }

public static void main(String args[]){
        TestInterruptingThread1 t1=new TestInterruptingThread1();
        t1.start();
        try{
                t1.interrupt();
        }catch(Exception e){System.out.println("Exception handled "+e);}


        }
}
```

The Static sleep(milliseconds) Method

- The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

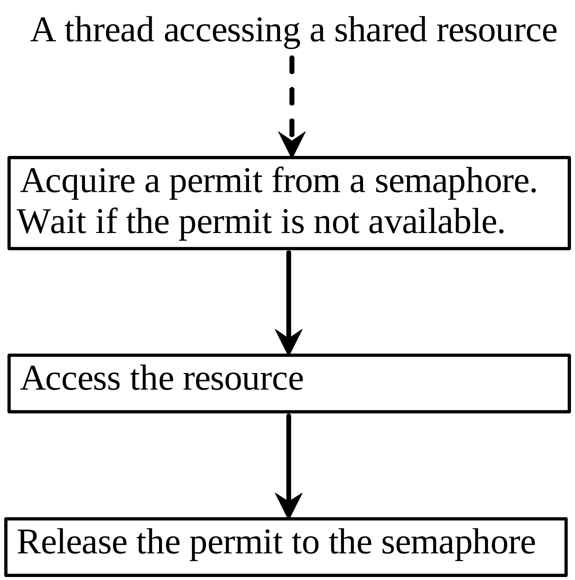
- Every time a number (>= 50) is printed, the <u>print100</u> thread is put to sleep for 1 millisecond.

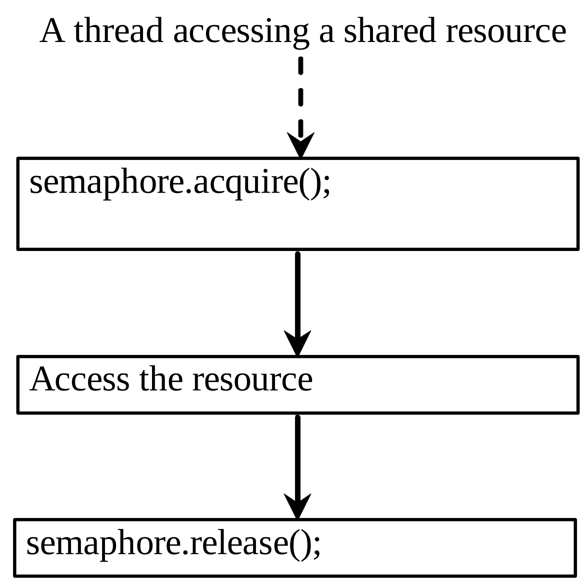
More Advanced Synchronization

- A semaphore object
  - Allows simultaneous access by N threads
  - If N==1, then this is known as a mutex (mutual exclusion)
  - Java has a class Semaphore

Semaphores

- Semaphores can be used to restrict the number of threads that access a shared resource.
- There is no notion of ‘ownership’ in Semaphores (unlike ‘locks’)





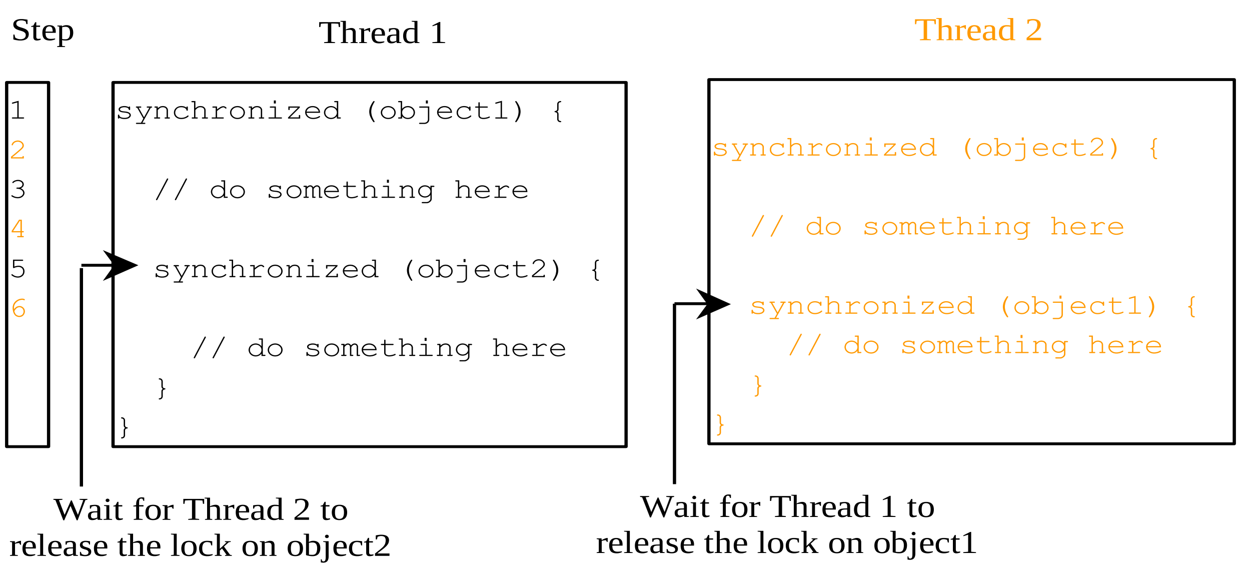
Creating Semaphores

- To create a semaphore, you have to specify the number of permits with an optional fairness policy
- Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1.
- Once a permit is released, the total number of available permits in a semaphore is increased by 1.

java.util.concurrent.Semaphore	
+Semaphore(numberOfPermits: int)	Creates a semaphore with the specified number of permits. The fairness policy is false.
+Semaphore(numberOfPermits: int, fair: boolean)	Creates a semaphore with the specified number of permits and the fairness policy.
+acquire(): void	Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.
+release(): void	Releases a permit back to the semaphore.

Deadlock

- Sometimes two or more threads need to acquire the locks on several shared objects.
- This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.
- In the figure below, the two threads wait for each other to release the in order to get a lock, and neither can continue to run.



Preventing Deadlocks

- Deadlock can be easily avoided by resource ordering.
- With this technique, assign an order on all the objects whose locks must be acquired and ensure that the locks are acquired in that order.
- How does this prevent deadlock in the previous example?

Summary

- Volatile variables are used to avoid synchronization issues due to caching of variables
- Monitors can be used for inter-thread communication
- Interrupts are used to gracefully stop threads
- Sleep() operation is used to suspend a thread for a specified time
- Semaphores control the access to a shared object
- Ordering of locks can avoid deadlocks