13	14 Exam 2	15 PROJ: Project 4 Ou	16 Threads	17	18	19
20	21 Threads	22	23 Disjoint Sets HW: HW 6 Out	24	25	26
27	28 Graphs	29 PROJ: Project 4 Du PROJ: Project 5 Ou	30 HW: Hw 6 Due	May 1	2	3
4	5 Graphs	6	7	8	9	10
11	12 TBD	13 PROJ: Project 5 Du	14 EMAIL GRACE DAYS EMAIL GRACE DAYS NO CLASS	15	16	17
18	19	20	21 10:30am CMSC 341,	22	23	24

Parallel Programming

A brief intro to:

Parallelism, Threads, and Concurrency

Our Goals

- Appreciate the 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

Keep in mind

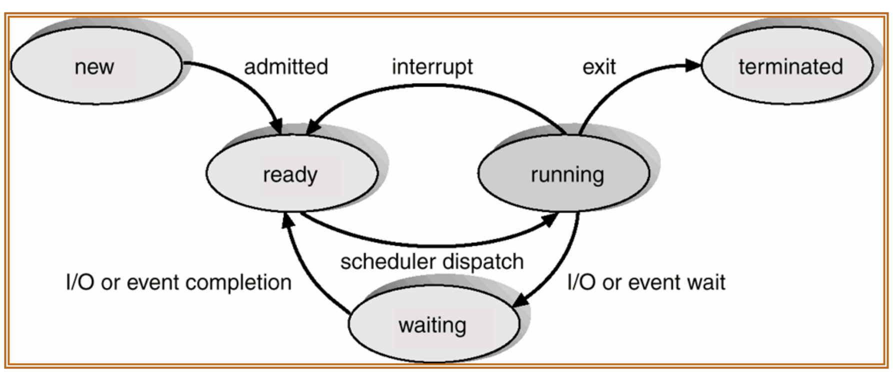
- An area of rapid change!
  - 1990s: parallel computers were expensive
  - Now: 4 core machines are commodity
- Variations between languages
- Evolving frameworks, models, etc.
  - E.g. Java's getting Fork/Join in Java 1.7 (summer 11)
  - MAP/REDUCE

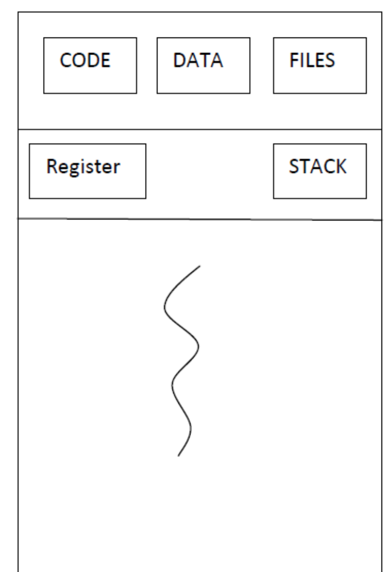
(Multi)Process vs (Multi)Thread

- Assume a computer has one CPU

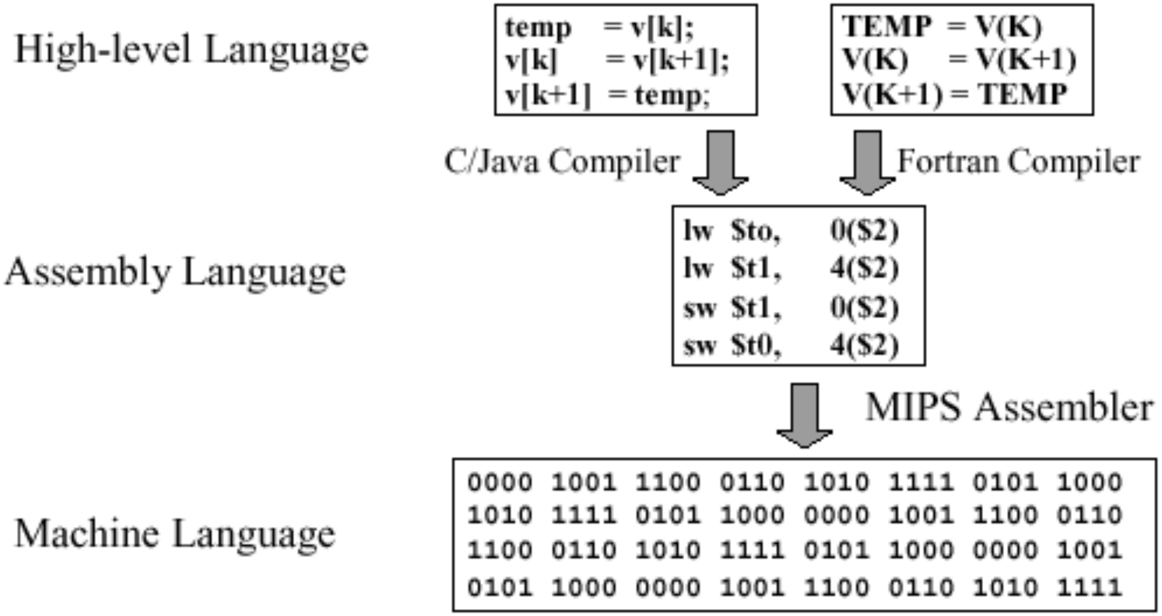
- Can only execute one statement at a time
  - Thus one program at a time

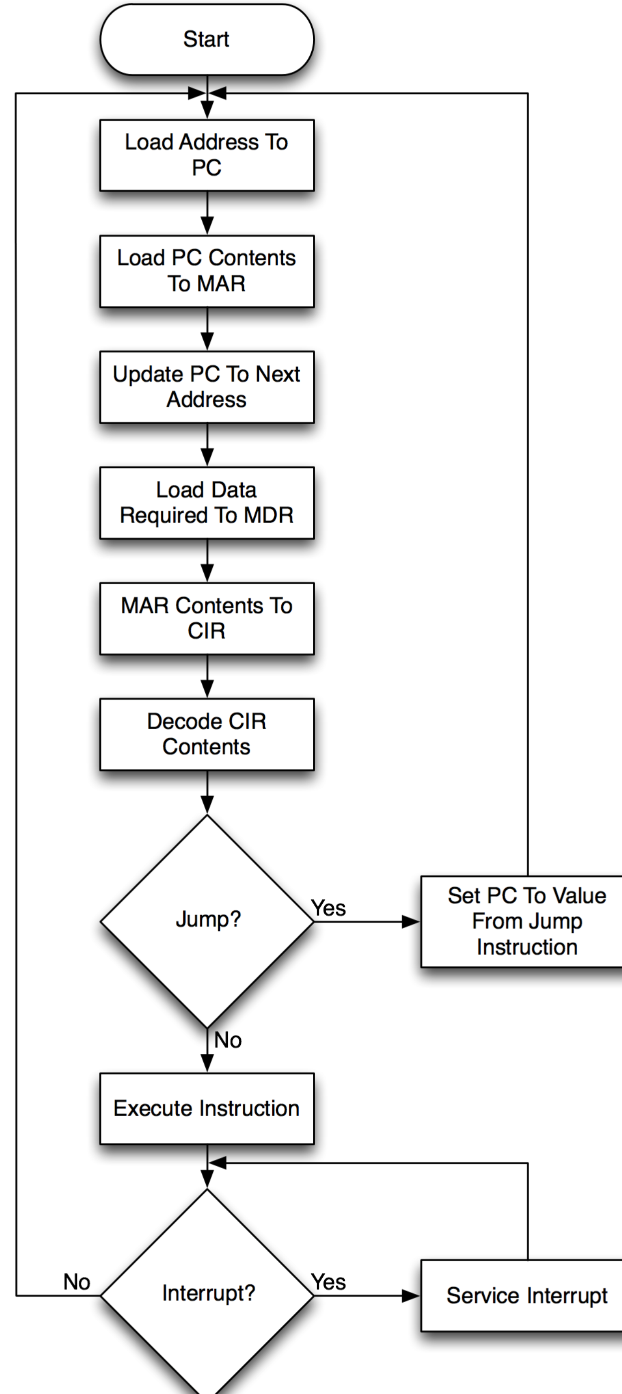
- Process: an operating-system level "unit of execution"





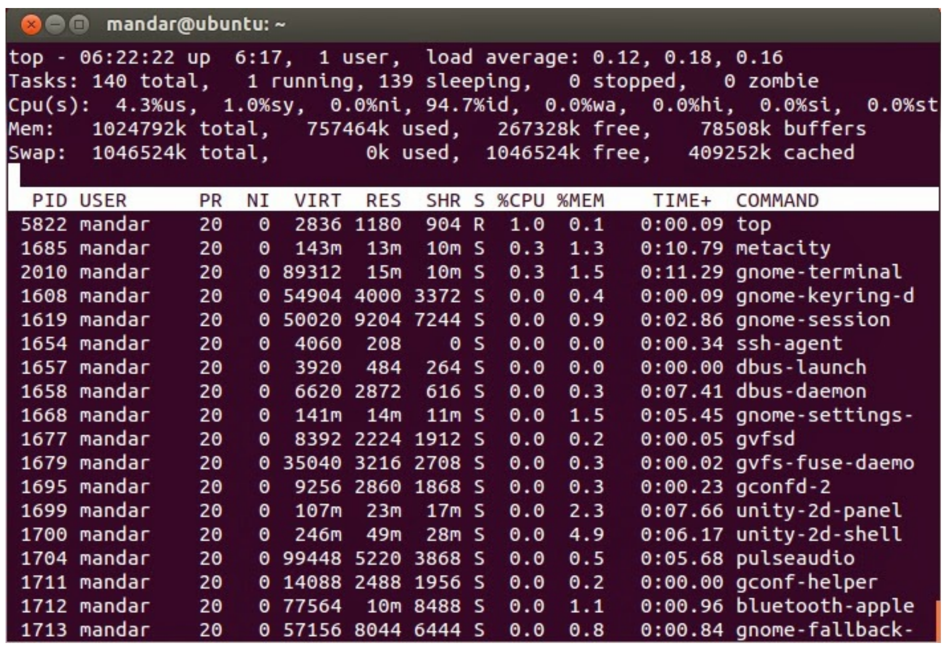
Single threaded process


High-level Language
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
TEMP = V(K)
V(K) = V(K+1)
V(K+1) = TEMP
C/Java Compiler
Fortran Compiler
Assembly Language
lw $to, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
MIPS Assembler
Machine Language
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111


Start
Load Address To PC
Load PC Contents To MAR
Update PC To Next Address
Load Data Required To MDR
MAR Contents To CIR
Decode CIR Contents
Jump?
Yes
Set PC To Value From Jump Instruction
No
Execute Instruction
Interrupt?
No
Yes
Service Interrupt

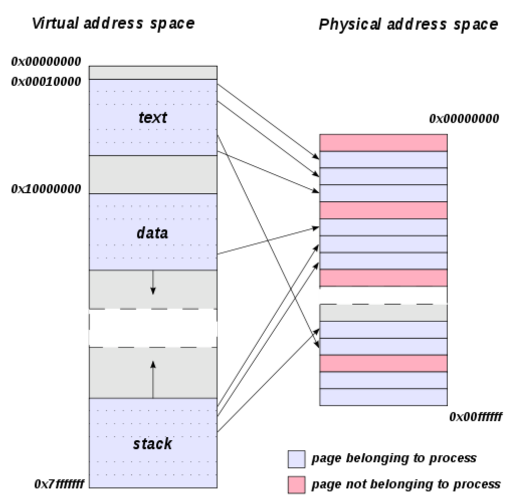
(Multi)Process vs (Multi)Thread

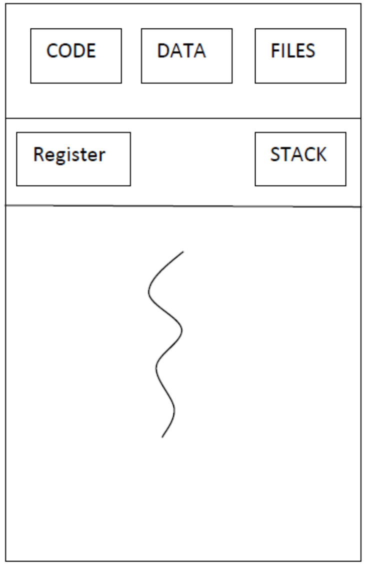
- Multi-processing
  - Op. Sys. “time-slices” between processes
  - Computer <u>appears</u> to do more than one program (or background process) at a time



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

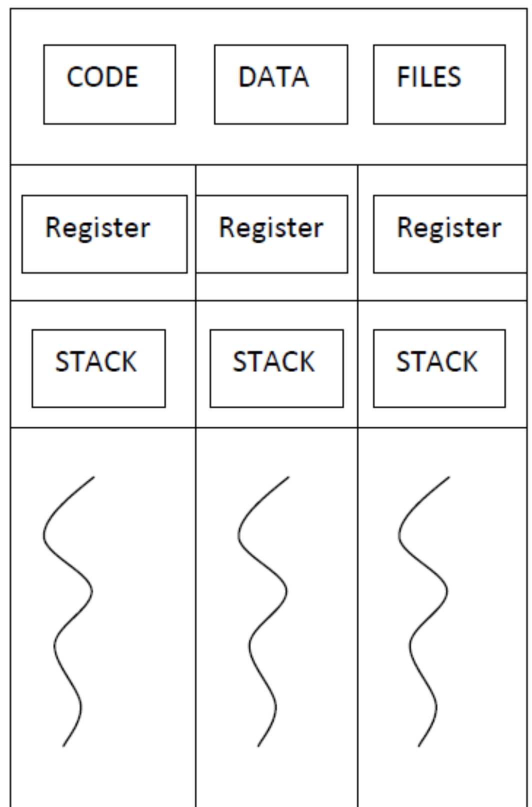




Single threaded process

Tasks and Threads

- A single program can be multi-threaded
  - Time-slicing done just like in multiprocessing
  - Repeat: the threads share the same memory



Multi threaded process

Task

- A **task** is an abstraction of a series of steps
  - Might be done in a separate thread

- In Java, there are a number of classes / interfaces that basically correspond to this
  - Example: 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 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()

Possible Needs for Task Objects

- Can you think of any?

Possible Needs for Task Objects

- Can you think of any?

- Storing tasks for execution later
  - Re-execution
- Undo and Redo
- Threads

Undo Operations

- A task object should:
  - Be able to execute and undo a function
  - Therefore will need to be able to save enough state to "go back"

- When application executes a task:
  - Create a task object and make it execute
  - Store that object on a undo stack

- Undo
  - Get last task object stored on stack, make it undo

Calculator App Example

- We had methods to do arithmetic operations:

```
public void addToMemory(double inputVal) {
     memory = memory + inputVal;
    }
```

- Instead:

```
public void addToMemory(double inputVal) {
    AddTask task = new AddTask(inputVal);
   task.run();
   undoStack.add(task);
  }
```

Stack, Undo Stack

- A **Stack** is an important ADT
  - A linear sequence of data
  - Can only add to the end, remove item at the end
  - LIFO organization: “last in, first out”
  - Operations: push(x), pop(), sometimes top()

- Stacks important for storing delayed things to return turn
  - Run-time stack (with activation records)
  - An undo stack (and a separate redo stack)

Nested class for Adding

```
private class AddTask implements UndoableRunnable {
    private double param;
    public AddTask(double inputVal) {
        this.param = inputVal;
    }
    public void run() {  // memory is field in CalcApp
        memory = memory + this.param;
    }
    public boolean undo() {
        memory = memory - this.param;
        return true;
    }
}
```

Undo operation

- In the Calc app:

```
public boolean undo() {
   boolean result = false;
   int last = undoStack.size()-1;
   if ( last >= 0 ) {
     UndoableRunnable task = undoStack.get(last);
     result = task.undo();
     undoStack.remove(last);
     }
   return result;
}
```

Example: MyTimerTask

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.

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()

```
class PrimeThread extends Thread {
         long minPrime;
         PrimeThread(long minPrime) {
             this.minPrime = minPrime;
         }

         public void run() {
             // compute primes larger than minPrime
              . . .
         }
     }
```

```
PrimeThread p = new PrimeThread(143);
     p.start();
```

Java’s Thread Class

- 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
- We won’t do this! 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
    - Several options here!

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

```
class PrimeRun implements Runnable {
         long minPrime;
         PrimeRun(long minPrime) {
             this.minPrime = minPrime;
         }

         public void run() {
             // compute primes larger than minPrime
              . . .
         }
     }
```

```
PrimeRun p = new PrimeRun(143);
     new Thread(p).start();
```

Join (not the most descriptive word)

- 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()

Thread Join Example