CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 2

OS Structure: Intro
TDD, BDD, and all that

January 27th, 2014
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Finish up our discussion about OS basics
- Test-Driven Design, Behavior-Driven Design
- Where are we going next?

Interactive is important!
Ask Questions!

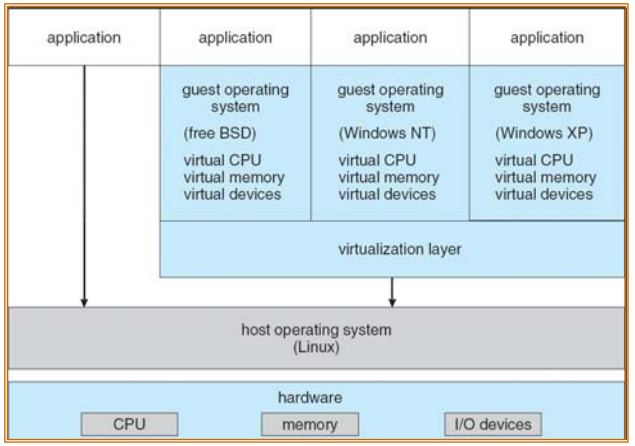
Note: Some slides and/or pictures in the following are adapted from slides ©2013

Review: Virtual Machines

- Software emulation of an abstract machine
  - Make it look like hardware has features you want
  - Programs from one hardware & OS on another one
- Programming simplicity
  - Each process thinks it has all memory/CPU time
  - Each process thinks it owns all devices
  - Different Devices appear to have same interface
  - Device Interfaces more powerful than raw hardware
    - » Bitmapped display ⇒ windowing system
    - » Ethernet card ⇒ reliable, ordered, networking (TCP/IP)
- Fault Isolation
  - Processes unable to directly impact other processes
  - Bugs cannot crash whole machine
- Protection and Portability
  - Stability of POSIX interface between systems
  - Limits to what Users programs are allowed to do

Review: Virtual Machines (con't): Layers of OSs

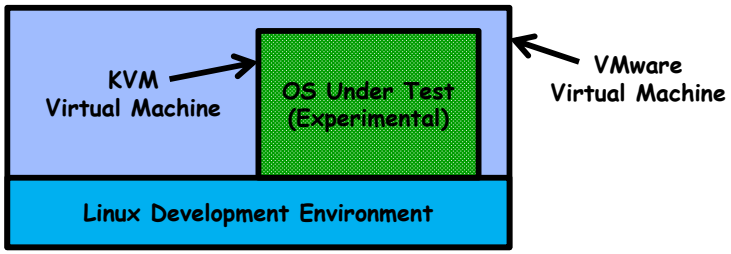
- Useful for OS development
  - When OS crashes, restricted to one VM
  - Can aid testing programs on other OSs



Review: How to work on OSes easily?

- Traditional:
  - Sit at serial console,
  - Upload new OS image somehow
  - Reboot and possibly crash ("Panic")
  - Debug with very limited tools

- How we will do it in this class: Virtual Machines!
- In fact – *Nested Virtual Machines:*



What does an Operating System do?

- Silerschatz and Gavin:
  "An OS is Similar to a government"
  - Begs the question: does a government do anything useful by itself?
- Coordinator and Traffic Cop:
  - Manages all resources
  - Settles conflicting requests for resources
  - Prevent errors and improper use of the computer
- Facilitator:
  - Provides facilities that everyone needs
  - Standard Libraries, Windowing systems
  - Make application programming easier, faster, less error-prone
- Some features reflect both tasks:
  - E.g. File system is needed by everyone (Facilitator)
  - But File system must be Protected (Traffic Cop)

What is an Operating System,… Really?

- Components:
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Communications? (Does Email belong in OS?)
  - Multitasking/multiprogramming?
- What about?
  - File System?
  - Multimedia Support?
  - User Interface?
  - Internet Browser? ☺
- Is this only interesting to Academics??

Operating System Definition (Cont.)

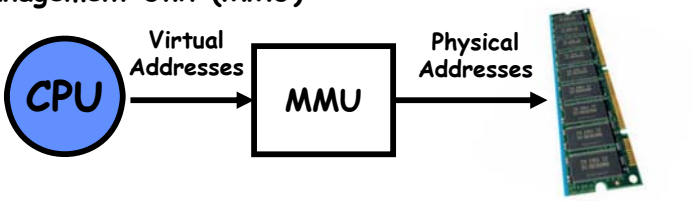
- No universally accepted definition
- "Everything a vendor ships when you order an operating system" is good approximation
  - But varies wildly
- "The one program running at all times on the computer" is the kernel.
  - Everything else is either a system program (ships with the operating system) or an application program
- Studying OSes is really about the Hardware/Software interface (API)
  - Thus, we will hope to give you enough knowledge to:
    - » Understand this interface
    - » Modify this interface
    - » Change the support underneath the interface

Review: Protecting Processes from Each Other

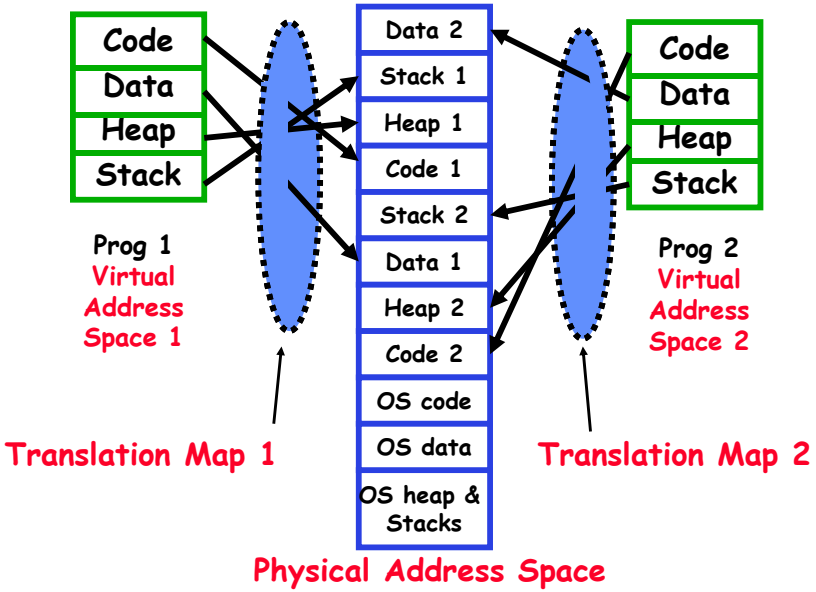
- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Address Translation
  - Dual Mode Operation
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Address Translation

- Address Space
  - A group of memory addresses usable by something
  - Each program (process) and kernel has potentially different address spaces.
- Address Translation:
  - Translate from Virtual Addresses (emitted by CPU) into Physical Addresses (of memory)
  - Mapping *often* performed in Hardware by Memory Management Unit (MMU)

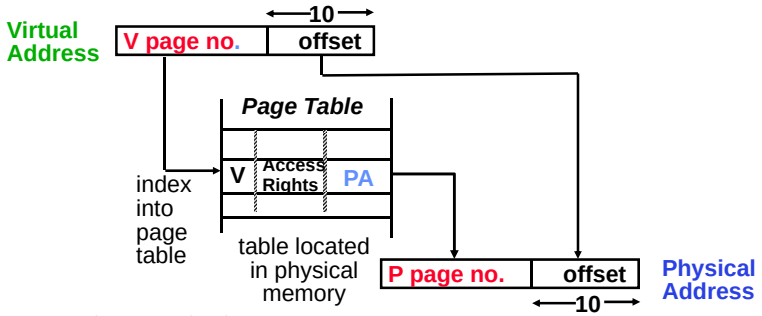


Example of Address Translation



Address Translation Details

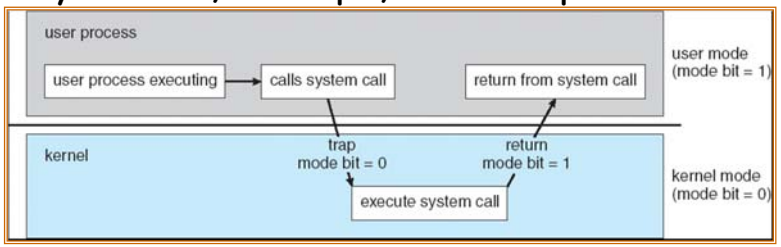
- For now, assume translation happens with table (called a Page Table):



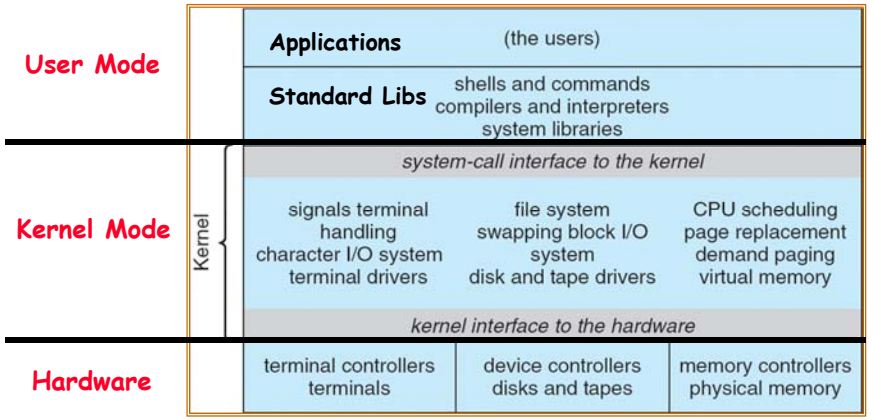
- Translation helps protection:
  - Control translations, control access
  - Should Users be able to change Page Table???

Dual Mode Operation

- Hardware provides at least two modes:
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode: Normal programs executed
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify page tables in user mode
    - » Attempt to modify ⇒ Exception generated
- Transitions from user mode to kernel mode:
  - System Calls, Interrupts, Other exceptions



UNIX System Structure



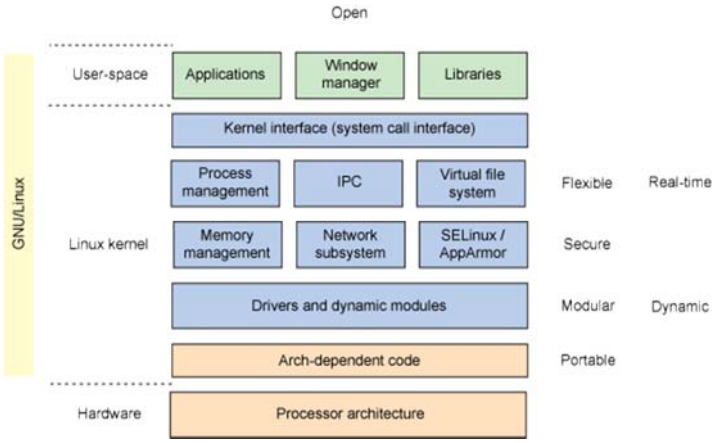
Administrivia

- If you don't have an account form, get one from Vedant or Palmer today:
  - Need it get VMware license
  - Log into your email account on http://inst.eecs.berkeley.edu
- For Redmine accounts:
  - Click on Google logo and use your XXX@berkeley.edu address!
  - Sometimes Google Chrome doesn't give you a chance to choose which google account you are using unless you logout first.
  - If you get a comment indicating that a new account has been created for you, you have probably logged in incorrectly.
- Redmine project development site
  - We are using a Redmine project development site for all resource control, bug tracking, etc
  - Your course account gives you access to the server
    - » Log in right away and update your name/email
    - » Generate an ssh key and upload that to the server
    - » Remote access to git repositories.

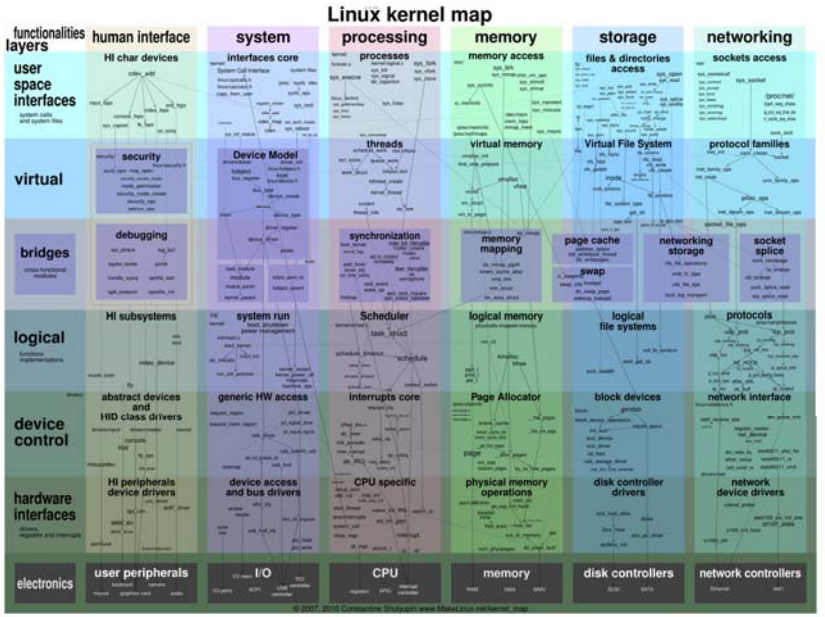
Administrivia (Con't)

- You should be well on the way to reading the Cucumber book!
  - Reading suggestions are up on the lecture page
  - In fact, I suggest that you work through the calculator example in the book!
- Get moving on the other suggested readings as well
  - I'll put up readings in the lecture schedule for the day we talk about them (look ahead!)
- Don't worry about groups until Monday
  - I will put up the official web form by then
- You should have already started on Project 0!
  - Project 0 is due on Wednesday (in two days!)
  - Project 0.5 starts on Wednesday (in two days!)
  - Lots of "stuff" to set up about your environment
- Why do we start with Project 0/0.5 (individually)?
  - We want everyone to be productive with the tools so that we can get down to the good stuff
  - We want to shake out the bugs in the infrastructure
- Check out "Resources" link off the home page
  - It gives some additional resources that might be useful

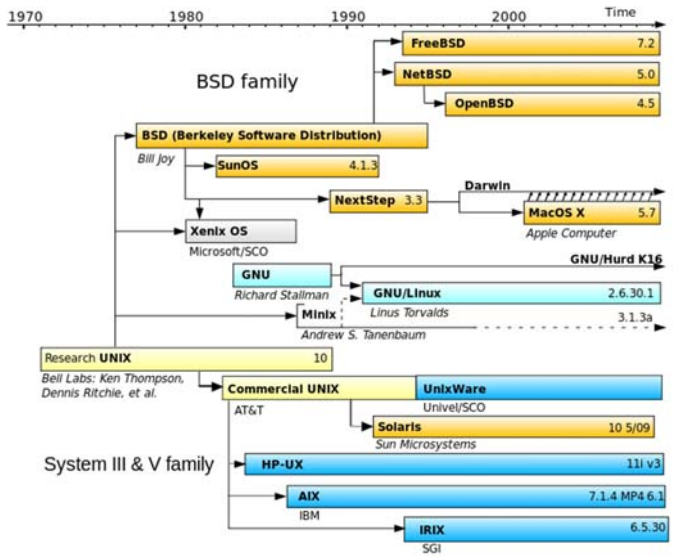
Linux Structure (More as term progresses!)



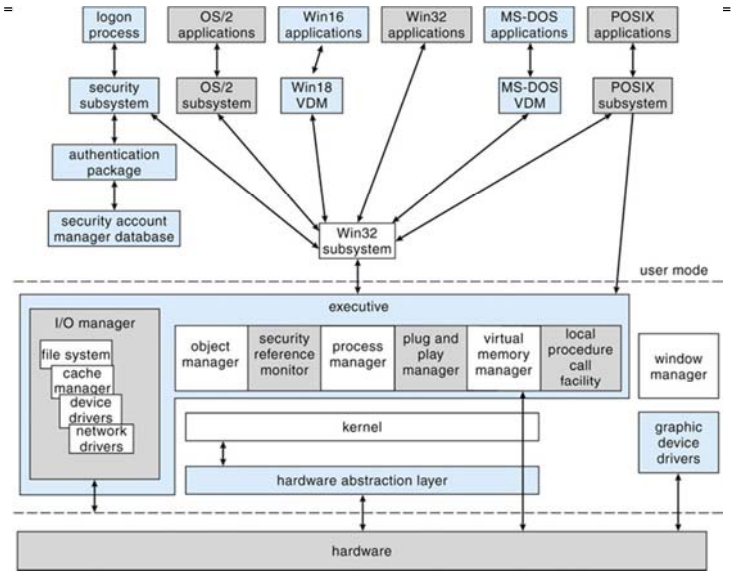
Map of Linux Components (http://www.makelinux.net/kernel_map/)



History of Unix



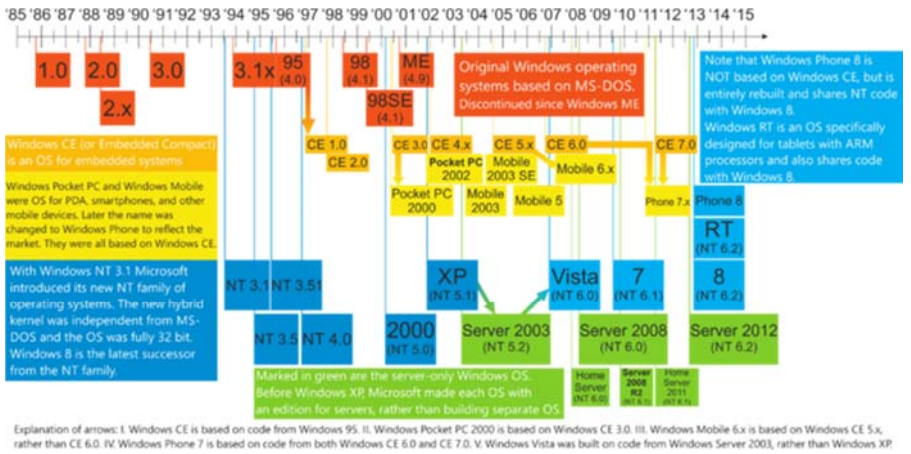
Microsoft Windows Structure



Major Windows Components

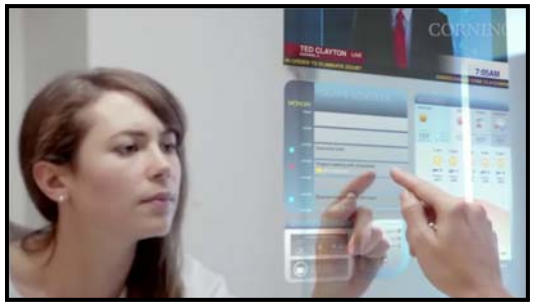
- Hardware Abstraction Layer
  - Hides hardware chipset differences from upper levels of OS
- Kernel Layer
  - Thread Scheduling
  - Low-Level Processors Synchronization
  - Interrupt/Exception Handling
  - Switching between User/Kernel Mode.
- Executive
  - Set of services that all environmental subsystems need
    - » Object Manager
    - » Virtual Memory Manager
    - » Process Manager
    - » Advanced Local Procedure Call Facility
    - » I/O manager
    - » Cache Manager
    - » Security Reference Monitor
    - » Plug-and-Plan and Power Managers
    - » Registry
    - » Booting
- Programmer Interface: Win32 API

History of Windows



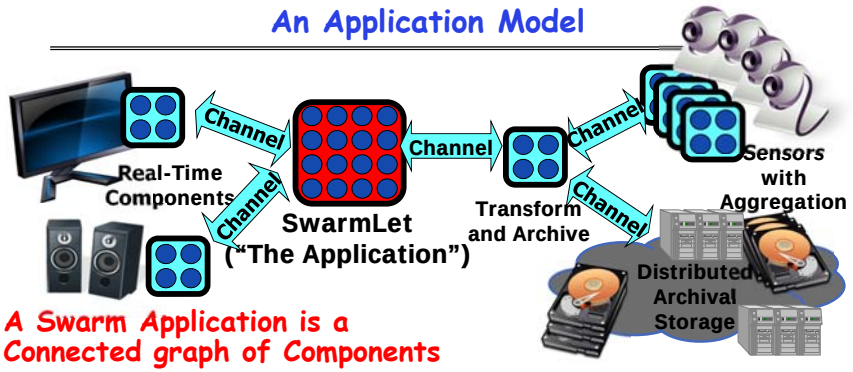
Brave New World: The Swarm?
Example: a Smart Space

- Displays Everywhere
  - Walls, Tables, Appliances, Smart Phones, Google Glasses….
- Audio Output Everywhere
- Inputs Everywhere
  - Touch Surfaces
  - Cameras/ Gesture Tracking
  - Voice
- Context Tracking
  - Who is Where
  - What do they want
  - Which Inputs map to which applications



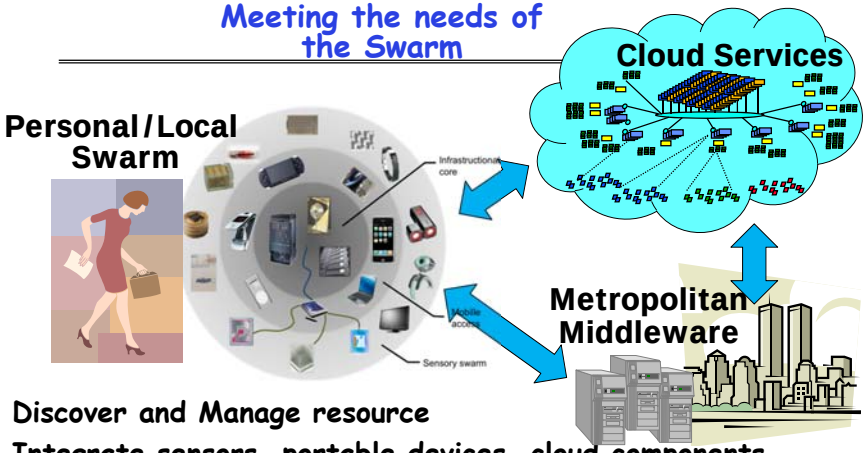
- How do we hope to organize this complexity?
  - Not via Stovepipe solutions! Today's typical solution!
  - Need something more global!

An Application Model



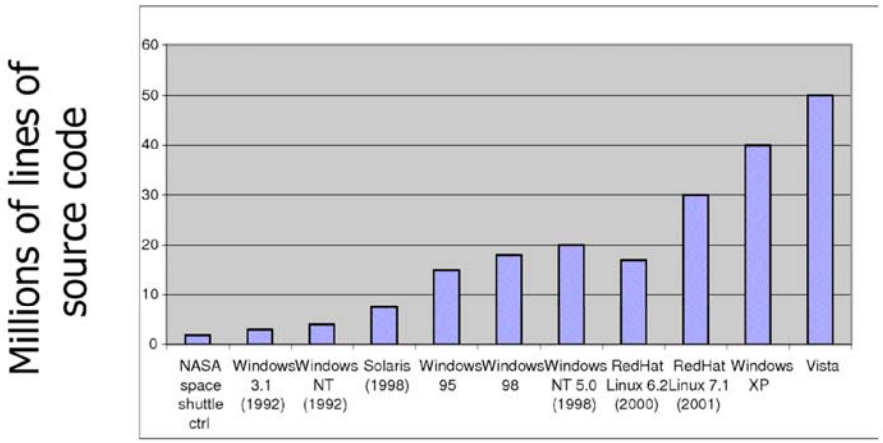
- A Swarm Application is a Connected graph of Components
  - Globally distributed, but locality and QoS aware
  - Avoid Stovepipe solutions through reusability
- Many components are *Shared Services* written by programmers with a variety of skill-sets and motivations
  - Well-defined semantics and a managed software version scheme
  - Service Level Agreements (SLA) with micropayments
- Many are "Swarmlets" written by *domain programmers*
  - They care *what* application does, not *how* it does it

Meeting the needs of the Swarm



- Discover and Manage resource
- Integrate sensors, portable devices, cloud components
- Guarantee responsiveness, real-time behavior, throughput
- Self-adapt to failure and provide performance predictability
- Secure, high-performance, durable, available information
- Monetize resources when necessary: micropayments

Recall: Increasing Software Complexity



From MIT's 6.033 course

What are typical problems with development?

- Delivering the "Wrong Thing"
  - Spend lots of time designing things only to discover that your solution doesn't do what you need!
  - Development process often divorced from "stakeholders", namely the people who know what the software actually needs to do
- Unstable in Production
  - Works great when you experiment with it
  - Doesn't work well in the field
- Costly to maintain
  - Software is brittle and each new feature causes you to break previous features
  - Unexpected regressions are common

What is wrong with traditional development?

- Rigid framework for development
  - Planning phase:
    - » How many people, what resources, etc?
  - Analysis phase:
    - » Try to articulate in detail problem trying to solve
    - » Ideally without prescribing how it should be solved (almost never happens)
  - Design phase:
    - » Think about design and architecture
    - » Standards to use, large and small technical decisions
    - » Decompose problem into manageable chunks to produce functional specifications
  - Coding phase:
    - » Write the software according to the spec
    - » In theory, all the "hard thinking" already done
    - » (Why programming and testing often sent off-shore to third parties)
  - Testing phase:
    - » We save testing until late in the process
  - Deployment
    - » Here we finally send the product out to users

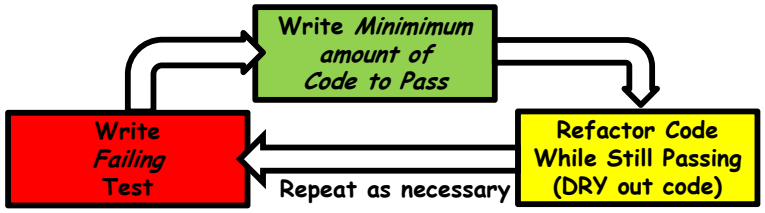
In reality, this doesn't really work well

- Why so much structure and ceremony?
  - Because traditionally the later in the process that we discover a problem, the more expensive it is to fix!
  - Each piece done by different team, thus phases represent handoff from one set of people to another
- In reality, there is much back and forth between analysis, design, and coding
  - As design or coding proceeds, problems are discovered forcing redesign of major components
  - These changes force increasingly complex communication between teams
- This complex interaction makes it increasingly unlikely that changes will happen through official channels
  - So, work done outside process
  - Documents get out of sync with software itself
  - Testing gets squeezed

Why is software designed this way?

- Perhaps in analogy with Civil projects
  - Need to be really sure you are doing the right thing when building a bridge – hard to go back a redesign the support structure after you poured the cement!
- But – Software is SOFT
  - Perhaps a development process that reflects the nature of software is better?
  - Software is brittle only if it is design with a rigid process!
- The Agile Manifesto instead:
  - We have come to value:
    - » Individuals and interactions over processess and tools
    - » Working software over comprehensive documentation
    - » Customer collaboration over contract negotiation
    - » Responding to change over following a plan
  - While there is value in the things on the right, we value the things on the left more

One step forward: Test-Driven Development (TDD)



- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle:
  - First the developer writes an (initially failing) automated test case that defines a desired improvement or new function,
  - Then produces the minimum amount of code to pass that test, and
  - Finally refactors the new code to acceptable standards.
- Key thing – *Tests* come before *Code*

From TDD ⇒ BDD

- What is problem with bare-bones TDD?
  - Where to start?
  - What to test or not test?
  - What tests to write?
  - How much to test at a time?
  - What should the test files be called?
- Much better: Check for *Behavior* rather than *Testing*
  - Now, the checking that is done is to show that a particular behavior wanted by someone is happening
    - » Should be performed at higher level (APIs, User Interface, etc) and independent of implementation
- Need a language (DSL) to express Behavior and way to automate verification of behavior
  - For instance – Cucumber!
  - Expresses human-readable analysis and *executable acceptance tests*
  - Write the minimum amount of code required to meet your behavioral checks
    - » Don't write code you don't need!
    - » If behavior isn't specified, don't bother writing code for it

A Ubiquitous Language for Analysis

- Need a framework for analyzing the process:
  - As a [X]
  - I want [Y]
  - so that [Z]
- Then, need a way of expressing the *acceptance criteria* in terms of *scenarios:*
  - Given some initial context (the givens),
  - When an event occurs,
  - Then ensure some outcomes
- Example in cucumber (called, say "valid_card_withdrawal.feature")

```
Feature: The Customer tries to withdraw cache using valid ATM card
     As a customer,
     I want to withdraw cache from an ATM
     so that I don't have to wait in line at the bank
  scenario: Successful Cache Withdrawal
     Given I have an ATM card that is owned by me
     When I request $40
     and my account has enough money
     Then I will receive $40
  scenario: Unsuccessful Cache Withdrawal
     Given I have an ATM card that is owned by me
     When I request $40
     And my account does not have enough money
     Then I will receive an error
```


Acceptance Criteria Should Be Executable!

- Cucumber provides an execution environment for Acceptance tests:

```
development_directory/features:
      *.feature                    # Cucumber Files
      step_definitions/*.rb        # Step Definitions
      support/*.rb                 # Supporting code
```

- How does this all connect?
  - Files in 'support' get loaded early, set up environment before starting execution and hooks to execute before and after each scenario
  - Files in 'step_definitions' are global definitions of what to do when a particular step (*Given, When, Then, And, But)* happens
- Step definitions should be treated like you are designing your own language!
  - They also connect the high-level language of feature files to the actual implementation
  - They may need to tweak the design in interesting ways
- Step definitions typically call code in the implementation before it has been implemented!
  - Write the code "you wish you had"

What do Step definitions look like?

- What do these steps translate into?

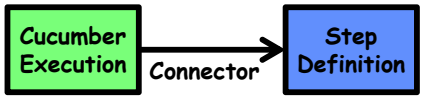
```
Given I have an ATM card that is owned by me
When I request $40
and my account has enough money
Then I will receive $40
```

- Answer: Regular expressions in a step file:

```
Given /^I have an ATM card that is owned by me$/ do
    # Set up  machine with card and valid PIN
     @my_account ||= Account.new
end

When /^I request \$(\d+)$/ do |amount|
     @my_request = amount
end

And /^my account has enough money$/
     @my_account.balance.should <= @my_request
end

Then /^I will receive \$(\d+)$/ do |amount|
     @my_account.request_money(@my_request).should = amount
end
```

- Steps interact with actual implementation
  - Reference code you "wish you had", not "code you already have"

How does this work in practice?



- Can use Step definitions to call out across interfaces:
  - Project 0 –
    - » Before and After hooks in 'support/hooks.rb' start up and shut down virtual machine; For autograder, will perform "git pull" and "make" on your kernel as well!
    - » Step definitions use custom protocol across serial interface to communicate with virtual machine
  - Cucumber-cpp
    - » Uses "Cucumber wire protocol" to send steps across TCP/IP channel to Step definitions *written in C++!*
    - » *"GIVEN, WHEN, THEN"* are c-preprocessor macros!
  - Cucumber+Capybara
    - » Adds special DSL to Steps for talking with various web browsers (can speak about content returned, etc)

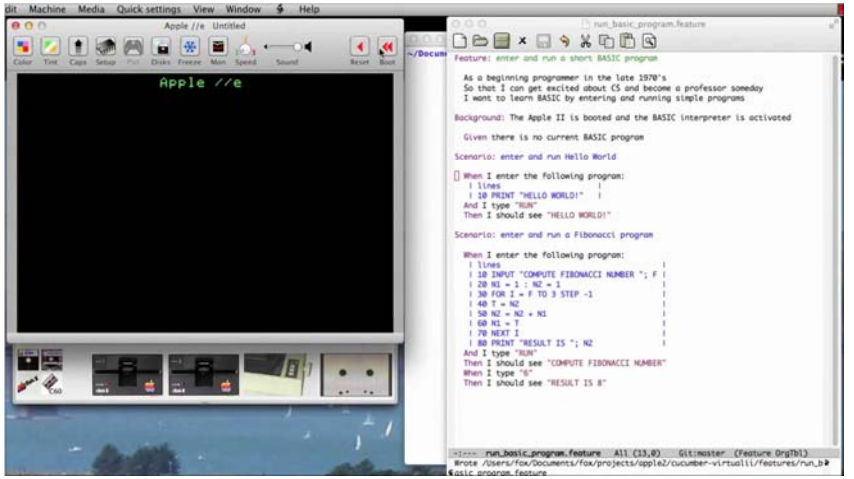
Amusing example: Verify Apple-II

- Start with Apple-II Emulator, then add BDD testing with Cucumber (Thanks to Armando Fox):
  https://github.com/armandofox/cucumber-virtualii

```
Feature: enter and run a short BASIC program

  As a beginning programmer in the late 1970's
  So that I can get excited about CS and
  become a professor someday
  I want to learn BASIC by entering and running
  simple programs

Background: The Apple II is booted and the
      BASIC interpreter is activated
   Given there is no current BASIC program

Scenario: enter and run Hello World
  When I enter the following program:
    | lines                    |
    | 10 HOME                  |
    | 20 PRINT "HELLO WORLD!"  |
  And I clear the screen
  And I type "RUN"
  Then I should see "HELLO WORLD!"
```

```
Scenario: enter and run a Fibonacci program
  When I enter the following program:
    | lines                                    |
    | 10 INPUT "COMPUTE FIBONACCI NUMBER "; F  |
    | 20 N1 = 1 : N2 = 1                       |
    | 30 FOR I = F TO 3 STEP -1                |
    | 40 T = N2                                |
    | 50 N2 = N2 + N1                          |
    | 60 N1 = T                                |
    | 70 NEXT I                                |
    | 80 PRINT "RESULT IS "; N2                |
  And I type "RUN"
  Then I should see "COMPUTE FIBONACCI NUMBER"
  When I type "6"
  Then I should see "RESULT IS 8"
```


Cucumber tests for Apple II: Video by Armando Fox



Verification Methodology

- Need for both User Stories (Behaviors) and Component Tests (Unit testing)
  - Behavioral Tests represent desired behavior from standpoint of stakeholders and involve whole code base
    - » Executable documentation!
    - » Slower, whole-system acceptance testing
    - » Run after every change
  - Unit testing frameworks (Like Rspec, CUnix, CPPSpec, etc) thoroughly test modules
    - » Fast execution
    - » Only run tests when change actual module
- Behavioral tests
  - High-level description independent of implementation
  - Test files named for behaviors being tested
    - » When failures happen, know where to start looking
  - Always in sync with code: tests run after every change
  - JBehave, Cucumber, etc
- Unit tests
  - Express individual details of implementation
  - Consider writing one or more unit test for every module
  - Can use CPPSpec, Cunit, etc.
  - Can be systematic, catch corner cases, etc

How Agile Methods Address Project Risks

- No longer Delivering Late or Over Budget
  - Deliver system in tiny, one- or two-week iterations (or mini-projects)
  - Always have a working release
  - Know exactly how much it costs
- No Longer Delivering the Wrong Thing
  - Can demonstrate new features to stakeholders and make any tweaks or correct any misunderstandings while work fresh in developer's minds
- No Longer Unstable in Production
  - Deliver something on every iteration
  - Must get good at building and deploying the application
    - » Releasing to production or testing hardware just another build to just another environment
    - » Rely on software automation to manage this
  - Application servers automatically configured, database schemas automatically updated, code automatically built, assembled, and deployed
  - All types of tests automatically executed to ensure system working
- No Longer Costly to Maintain
  - With first iteration –team is effectively in maintenance mode!
  - Adding code to a working system, so they have to be very careful

Conclusion

- Studying OSes is really about the Hardware/Software interface (API)
  - Thus, we will hope to give you enough knowledge to:
    - » Understand this interface
    - » Modify this interface
    - » Change the support underneath the interface
- Test-Driven Development (TDD)
  - Write tests first, then write code, then refactor
- Behavior-Driven Development (BDD)
  - Instead of Tests, think about writing Executable Behavior specifications
  - Cucumber for Integration Behaviors, Unit tests for implementation.