Welcome!

- Read the Syllabus
  - read the warning about the size of the project
  - make sure you get the 7th edition of the book (if you bought the 6th talk to me)
- Discussion Sections (start Monday)
  - focus on the project

Projects

- The best way to understand is by doing
  - So, we will build an operating system that works on bare hardware by extending the GeekOS academic operating system. Seven projects total.
- Project #0 to be posted Friday
  - It will be due Friday next week.
  - It will get you familiar with the bochs simulator and some of the basics of the initial kernel

Class Grades Server

http://grades.cs.umd.edu

- Get your LinuxLab account from here
  - CS computing cluster. Projects must work and be submitted on these machines.
- Complete grade information
- Interface for requesting regrades on exams and projects (with deadline!)

Forum

- Place for discussion about the projects
  - Preferred place to ask questions; then all students will benefit from the answer
  - Don’t cross the line: do not to provide direct answers that give away main parts of the project! i.e., don’t post code, pseudocode, etc.
- Linked from the course web page

Course Material

- Reading
  - Chapter 1 (today, Monday)
  - Chapter 2 (for Monday)
Why Study OSs?

- Understand computer systems
  - From the hardware up to applications
  - Helpful for understanding performance, security, reliability, and other issues of general interest to computer scientists and engineers

Why Study OSs?

- Understand principles of abstraction
  - OSs are large and complex. How do we manage this complexity?
  - Abstraction:
    - Break each piece into self-contained chunks
    - Plug each piece together
  - Different views of service from view of the provider and view of the user
  - These principles will serve you in other software systems projects
    - E.g., thinking about concurrency

Why Study OSs?

- Understand tradeoffs of system design
  - Many types of users (too many!)
    - real-time, desktop, server, etc...
  - Many possible models and abstractions
    - OS researchers are ‘abstraction merchants’
    - There is no perfect OS!

Other core CS

- While I’m at it, consider
  - Compilers - write a compiler to understand programming languages better, and thus become a better programmer
  - Theory/algorithms - formal models and understanding of computation, helps you approach programming more thoughtfully
  - OSes have the same flavor
    - The subject is not so important as what you can learn along the way

What is an Operating System?

- Resource Manager
  - Resources include: CPU, memory, disk, network
  - OS allocates and de-allocates these resources

- Virtual Machine/Multiplexor
  - provides an abstraction of a larger (or just different) machine
  - Examples:
    - Virtual memory - looks like more memory
    - Java - pseudo machine that looks like a stack machine
    - IBM VM - a complete virtual machine (can boot multiple copies of an OS on it)
    - VMware, Parallels, etc. - like the IBM VM, but in software. These are essentially a kind of OS.

What is an OS (cont)?

- Provider of Services
  - includes most of the things in the above definition
  - provide “common” subroutines for the programmer
    - windowing systems
    - memory management

- The software that is always loaded/running
  - generally refers to the OS kernel.
    - small protected piece of software

- All of these definitions are correct
  - but not all operating have all of these features
Operating System Cousins

- **Hardware**
  - OS is managing hardware resources so needs to know about the ugly details of the hardware
    - interrupt vectors
    - page tables
    - I/O registers
  - some features can be implemented either in hardware or the OS

- **Languages**
  - can you write an OS in any language?
    - No: need to be able to explicitly layout data structures to match hardware

OS Cousins (cont)

- **Language Runtime systems**
  - memory management requirements
    - explicit heap management
    - garbage collection
    - stack layout
    - concurrency and synchronization
    - calling convention (how are parameters passed)

- **Data Structures and Algorithms**
  - efficient access to information in an OS
    - for most things need linear time and space
    - for many things want log or constant time

Operating System Structure

OS Execution Basics

- The OS kernel is **event-driven**:
  - It waits for an event to occur, and then responds to that event

- Events represented by the hardware as **interrupts**
  - Most often from devices
  - But also can arise from software

Abstract View of System

Computer System Structure
Hardware Interrupts

- Indicate an event has occurred
  - when caused by hardware devices
    - indicates data present or hardware available
  - when caused by software
    - Indicates a system call (or trap)
- Interrupt the CPU to execute a handler
  - saves state about what was happening
  - returns where it left off when finished
- Handlers are entry points to the OS

Servicing Interrupts

- Need to know what device interrupted
  - could ask each device (slow!)
  - instead use an interrupt vector
    - array of pointers to functions to handle a specific interrupt
- What happens if an interrupt arrives while we are in an interrupt handler?
  - OS may wish to disable interrupts until handling is complete.

Jobs and Processes

- The OS is responsible for running programs
  - Called jobs, tasks, processes, ...
- Simple approach: batch system
  - Automate running of user jobs
  - Users submit jobs, OS schedules them, run without preemption
  - OS referred to as a resident monitor

Multiprogramming

- Many jobs active “at once”
  - Each job no longer runs to completion before the next can start
  - Many jobs resident in memory at once; the CPU is shared between them, or they run on multiple CPUs.
- Job scheduling
  - which jobs to load into memory

Multiprogramming OSs

- Must manage resources
  - Memory
  - I/O devices
  - CPU
  - Want to be fair and efficient
- CPU scheduling
  - which memory-resident program to run
- Protection
  - Prevent errant job from tainting results of another
Multitasking (Timesharing)

- Like multiprogramming, but interactive
  - Switching between tasks is very fast
- Preemption: interrupting a job to transfer control to another job
  - Quantum: the time slice allocated to a process before it’s preempted. How long should it be?
- Interactivity requires more support
  - Filesystem, virtual memory, synchronization, deadlock avoidance

Desktop System

- Catered to a single user
- Goal: flexibility and responsiveness
  - Support interactive I/O devices
  - May not need to be as efficient as multi-user OS to be more responsive
  - May be less concerned with protection
- Multitasking is very useful here
  - Ensures user responsiveness since background jobs can be preempted

Multiprocessor System

- Single computer system with several CPUs
  - All share memory, clock, devices
- Goals:
  - Better value
    - Increased throughput
    - Lower cost
  - Increased reliability
- Two flavors
  - Symmetric and asymmetric
- Multi-core systems: multiple processors on one chip

Distributed (Clustered) System

- Many machines joined by a network
  - Loosely coupled: each processor has its own memory; communication via the network
  - OS often implements network protocols
- Goals
  - Value
  - Share distributed resources
- Variants: client-server, peer-to-peer

Real-time Systems

- Control device in a dedicated application
  - Medical imaging, industrial control, etc.
- Well-defined fixed time constraints
- Variants
  - Hard: can never miss a deadline
  - Soft: less stringent. Thus permits unpredictable elements like caches, disks, etc.

Handheld Systems

- PDAs, cell phones, etc.
- Issues
  - Limited memory
  - Slow (power-constrained) processors
  - Small display
Migration of OS features

OS Goals: Usability

- Robustness
  - accept all valid input
  - detect and gracefully handle all invalid input
  - should not be possible to crash the OS

- Consistency
  - same operation should mean the same thing
    - read from a file or a network should look the same
    - a “-” flag should be the same in different commands
  - conventions
    - define the convention
    - follow the convention when adding new items

Usability Goals (cont)

- Proportionality
  - simple, common cases are easy and fast
    - good default values
  - complex, rare cases are possible but more complex and slower
    - “rm *” should give a warning
    - formatting the disk should not be on the desktop next to the trash can

Cost Goals

- Good Algorithms
  - time/space tradeoff are important
  - use special hardware where needed
    - smart disk controllers, memory protection

- Low maintenance cost
  - should not require constant attention

- Maintainability
  - most of cost of software is in maintenance so make it easy to maintain the software base

Adaptability Goals

- Tailored to the environment
  - server vs. workstation
  - multi-media vs. data entry

- Changes over time
  - added memory
  - new devices

- Extensible
  - third parties can add new features
    - database vendors often need custom features
    - end customers can extend the system
      - new devices
      - new policies