Overview

Three Easy Pieces

- OS is a resource abstractor and resource allocator
  - resource abstractor
    defines set of logical resources that correspond to hardware resources and well defined operations on it
  - resource allocator
    decided who gets how much and when
- **Why do we want virtualization?**
  To make the computer easier to program and to provide each program the illusion that it is alone on the computer going through the Fetch Decode Execute cycle
- Concurrency
  juggling multiple things at once
- Persistence
  ability to store data that survives a program termination/ computer shutdown
  done by the file system

Kernel

- OS Kernel
  the core part of the software (OS is software) and is in charge of implementing resource abstraction and allocation
  A code and data that always resides in RAM
  - A challenging software development endeavors with multiple concerns
    not to use too much memory, speed, cannot use standard libraries
  - uses non portable compiler directives
  - Debugging is hard and a kernel bug in the file system makes generating a crash dump problematic

Process of Turning On a Computer

1. POST (Power-On Self-Tests) are performed by the BIOS in firmware/ROM (Read-Only Memory)
2. Booting: The BIOS runs a first program: the bootstrap program
3. The bootstrap program initializes the computer (register contents, device controller contents, ...)
4. Then it loads another program in RAM, the bootstrap loader, and runs it
5. The bootstrap loader loads (whole or part of) the kernel (i.e., some code and associated data) into RAM at a known/fixed address
6. At some point, a bootstrap loader creates and starts the first program (called init on Linux, launchd on OS X)
7. Once all this has been done... nothing happens until an event occurs

Booted OS

- kernel code and data reside in memory at a specified address, as loaded by the bootstrap program
- Kernel's footprint has to be small
- each program is loaded in RAM
- In RAM 2 kinds of code
  - User Code
    written by user and library developers
  - Kernel Code
    written by kernel devleopers
- process can run user code and by doing a system call it can run the kernel code

  - Memory Protection
    each process think its alone and processes never step on each other's toes in the RAM

Kernel: Event Handler

- event handler as in all entries into the kernel code occur as a result of an event
- kernel defines a handler for each event type
- 2 Kinds of OS Events
  Interrupts (Asynchronous): some device controller saying some hardware thing happened and generated in real time from the outside world
  Traps (Synchronous)
  caused by instruction executed by a running program and synchronous because generated as a part of fetch decode execute cycle from inside world
- System Call
  a trap by which user programs get the kernel to do some work on their behalf

Interfaces

- Batch: Historical Human Interface
- GUI
- CLI
  can start the system programs and user programs where the distinction between them is a matter of perspective
  system program is not in the OS but pre installed

System Call API

- "Lowest-level" interface to OS services and GUI, CLI are built on top of this API
- Every program uses it
- On Linux, the strace tool makes it possible to spy on how a program uses the System Call API
- On UNI-ish systems, the time tool makes it possible too time system calls
- each call is identified with a number stored internal table named the syscall table
- makes it possible to access virtualized hardware resource
- calls are expensive and should be done wisely

Main OS Services

- Process Management
  The OS is responsible for
  - Creating and deleting processes;
  - Suspending and resuming processes;
  - Providing mechanisms for process synchronization;
  - Providing mechanisms for process communication;
  - Providing mechanisms for deadlock handling.
- Memory Management
  determines what is in memory when the kernel is always in memory
  The OS is responsible for
  - Keeping track of which parts of memory are currently being used and by which process
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

  OS is not responsible for
  - for memory caching, cache coherency, etc. as they are managed by hardware

- Storage Management
  the OS provides uniform, logical view of information storage
  OS operates File System Management
  - Creating and deleting files and directories
  - Manipulating files and directories
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile) storage media
  - Free-space management
  - Storage allocation
  - Disk scheduling
- I/O Management
  OS hides peculiarities of hardware devices from the user and the OS is responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), spooling (the overlapping of output of one job with input of other jobs), etc.
  - General device-driver interface
  - Drivers for specific hardware devices
- Protection and Security
  - Protection
    mechanisms for controlling access of processes to resources defined by the OS
  - Security
    defense of the system against internal and external attacks
    OS provides
    * Memory protection
    * Device protection
    * User IDs associated to processes and files
    * Group IDs for sets of users
    * Definition of privilege levels