What does an Operating System do?

- manages hardware
    - fairly allocates hardware among the applications
    - ensure privacy and enable controlled sharing
    - oversee program execution and handle errors
- abstract the bare hardware
    - make it easier to use, make the applications platform independent
- new abstractions to enable applications
    - powerful features beyond the bare hardware
- Always in control of the hardware
    - first software loaded when the machine boots
    - continues running while apps come and go (must be able to run for a long time)
- Parts of it have complete access to hardware
    - privileged instructions, all memory and devices
    - mediates application access to the hardware

Privileged Instructions

- most CPU instructions can be used by anyone: arithmetic, logical, etc
- some instructions are privileged because they need to ensure privacy

What does an OS look like?

- while CPU supports data types and operations, OS supports richer objects, higher operations, files, processes…,create, destroy, read, write, signal
- much of what the OS does is behind the scenes
- plug and play, power management, fault handling, domain services, upgrade management

Functionality in OS

- OS code is very expensive to develop and maintain
- useful to distinguish OS from kernel (more on that later)
- functionality must be in the OS if it
    - requires the use of privileged instructions (ex: changing address spaces)
    - requires the manipulation of OS data structures
    - requires for security, trust, or security integrity

Complexity Management

- Layered/Hierarchical Structure
    - can be understood progressively, piece at a time
- Modularity and Functional Encapsulation
    - hiding complexity and simplifying interfaces
- Generalizing and Unifying Abstractions
    - high-level organizing concepts, readable solution paradigms
- Indirection, Federation, and Deferred Binding
    - TBD plugins for TBD problems
- Appropriate Abstraction
    - functionality well-suited for intended users

S/W Principles from this Course

- Mechanism/Policy: to meet a wide range of evolving needs
- Interfaces as contracts: implementations are not interfaces
- Dynamic equilibrium: Robust adaptive resource allocation
- Fundamental Role of Data Structures: Robust adaptive resource allocation
- Iterative Solutions/Progressive Refinement: incremental improvements

Readings
Complexity Management Principles

- Layered Structure and Hierarchical Decomposition
    - Hierarchical Decomposition is the process of decomposing in a top-down fashion
    - Operating System is not always strictly hierarchical
        - communication between different parts increasing
    - UC System example
        - UC System can be broken into Regents, the Office of the President, and the campuses
        - then from campuses, we can divide into each individual campus
        - one campus (UCLA) can be divided into different administrations
        - administrations can be divided into personnel, etc.
- Modularity and Functional Encapsulation
    - Each group/component (at each level) must have a coherent purpose
    - Most functions can be performed entirely within that group/component
    - The union of the groups is able to achieve the system's purpose
    - As long as the component is able to effectively fulfill its responsibilities, external managers and clients can ignore the internal structures and operating rules
    - When it is possible to understand and use the functions of a component without understanding its internal structure, we call that component a **module**, and say that its implementation is **encapsulated** by that module
    - Smaller and simpler components are easier to understand than larger and more complex ones
        - We want a larger amount of smaller and simpler components
    - Since many functions are closely related, if an operation on a single resource is implemented in separate components, there is a danger that that component may break another
        - Combine closely related functionality into a single module! + we want smallest possible modules consistent with the co-location of closely related functionality
        - This is called **cohesion**
    - If most operations can be accomplished entirely within a single component, they are likely to be accomplished more efficiently. Exchanges of services can lead to
        - Overhead of communication may reduce efficiency
        - Increased number of interfaces increases the complexity of the system
        - Increased dependencies between components increase likelihood of errors

- Appropriately Abstracted Interfaces and Information Hiding
    - Faucet Example: Traditionally, faucets have been implemented with a hot and cold valve, but this design is not necessarily the best one
        - you must guess what the rate of flow and temperature of the water is
        - A better solution would be to create valves for the temperature and flow rate
    - Even if the implementation (internal) was well suited, exposing implementations could result in
        - exposing more complexity to the client, making the interface difficult to describe/learn
        - limit the provider; flexibility to change the interface in the future
    - An appropriately abstracted interface that does not reveal the underlying implementation is said to be opaque and display good information hiding
- Powerful Abstractions: One that can be applied to many instructions
    - we can draw on a library of tools, where new tools can radically alter the way we approach problems and solve them
    - virtually all the tools and resources in operating systems are abstract concepts from the imaginations of system architects. A powerful abstraction is one that can be profitably applied to many situations
        - common paradigms (e.g. lock granularity, cache coherency, bottlenecks) that enable us to more easily understand a wide range of phenomena
        - common architectures (e.g. federations of plug-in modules treating all data sources as files) that can be used as fundamental models for new solutions
        - common mechanisms (e.g. registries, remote procedure calls) in terms of which we can visualize and construct solutions)
    - Sufficiently powerful abstractions give us tools to understand, organize, and control systems which would otherwise be extremely complex.
- Interface Contracts
    - An interface specification is a contract: a promise to deliver certain results, in specific form, in response to the request
    - These contracts also represent a commitment for what will be done in the future
    - An operating system is used for a long time, and built upon thousands of people's codes and contributions
    - However, it is impossible for every developer to test every change with every combination of components from every other developer. Interface contracts are our first line of defense against incompatible changes.
- Progressive Refinement
    - Because it is difficult to estimate the work, anticipate the problems, and get the requirements right in a large project, we want to add new functionality in smaller, one feature at a time, increments. We want to deliver these new functionalities as quickly as possible and add on as much as possible

Architectural Paradigms

- Mechanism/Policy Separation
    - The basic concept is that the mechanisms for managing resources should not dictate or greatly constrain the policies according to which the policies are used
        - Therefore, resource managers should be designed in two logical parts
            - The mechanisms that keep track of resources and give/revoke client access to them
            - A configurable plug-in policy engine that controls which clients get which resources when
    - Card-key lock system example
        - Each door has a card-key reader and a computer-controlled lock
        - each user is issued a personal card-key
        - to get access to a room, a user swipes a card-key past the reader
        - a controlling computer will lookup the registered owner of the card-key in an access control database in order to decide whether or not the user has access
        - In this case, the physical mechanisms are the card-keys, readers, locks, and the controlling computer. The policy is represented by rules in the access control database, which can be configured to support a wide range of policies
            - the mechanisms itself impose very few constraints on who should be allowed to enter which rooms when
            - Because the policy is independent from the mechanisms, we could also move on to a very different mechanism implementation and continue to support the exact same access policies
- Indirection, Federation, and Deferred Bindings
    - Powerful unifying abstractions often give rise to general object classes with multiple implementations
    - Consider a file system: there are many different kinds of file systems but they all have similar functionality (ex: DOS/FAT on SD cards, ISO 9660 file systems on CD ROMs, etc), but they all implement rather similar functionalities
    - One could write an operating system that understood all possible file system formats, but the number and range file system formats and rate of evolution dooms that approach to failure.
        - Realistically, you want to find a way to add support for new file systems independently from the operating system to which they will be connected
    - The most common way to accommodate multiple implementations of similar functionality is with plug-in modules that can be added to the system as needed. Typically, these implementations share these features
        - A common abstraction (class interface) to which all plug-in modules adhere
        - the implementations are not built-in to the operating system, but accessed indirectly

        - The indirection is often achieved through some sort of federation framework that registers available implementations and enables clients to select the desired implementation
        - the binding of a client to a particular implementation does not happen when the software is built but rather deferred until the client actually needs to access it
        - deferred binding: The implementing module may be dynamically discovered, and dynamically loaded. It need not be known or loaded into the OS until a client requests it
- Dynamic Equilibrium
    - Most complex systems have numerous tunable parameters that can be used to optimize behavior for a given load
    - Unfortunately, a good set of tunable parameters is not enough
        - tuning parameters tend to be highly tied to a particular implementation, and proper configuration often requires understanding of complex processes
        - it is possible to build automated management agents that continually monitor system behaviors, and promptly adjust accordingly
    - Stability of a complex natural system is often the result of dynamic equilibrium, where the state of the system is the net result of opposing forces. Any external event that pertubes the equilibrium automatically triggers a reaction in the opposite direction
    - ex: the amount of water in a sealed pot is the net result of evaporation and condensation
    - If resource allocation can be driven by opposing forces, we may be able to design systems that continuously and automatically adapt
- Criticality of Data Structures
    - our data structure designs determine which operations are fast and slow, which operations are simple and complex, the locking requirements for each operation, and the speed of error recovery
    - Often, when confronted with a difficult performance, robustness, or correctness problem, the solution is found in the right data structure

Reading Notes (Chapter 2)
What happens when a program runs?

- A running program executes instructions
    - Many millions of times every second, the processor fetches an instruction from memory, decodes it, and executes it. After it is done with this instruction, the processor moves on to the next set of instructions
- Operating system does things such as allowing programs to share memory, enables programs to interact with devices, etc.

Virtualization: The OS takes a physical resource and transforms it into a more general, powerful, and easy-to-use virtual form of itself. Sometimes, the OS is referred to as a **virtual machine**

- In order to allow users to tell the OS what to do and thus make use of the features of the virtual machine, the OS also provides some APIs that you can call.
- Because the OS provides these system calls to run programs, access memory, and devices, we also sometimes say that the OS provides a standard library to applications
- The OS is also sometimes known as a resource manager because it allows programs to access devices, instructions, and data
    - Each of the CPU, memory, and disk is a resource of the system. It is thus the operating system's role to manage those resources

Virtualizing the CPU (2.1)

- Ex: Run the spin() function as 4 times
    - It appears as if the programs are running at the same time
    - The operating system, with some help from the hardware, turns a single CPU into seemingly infinite number of CPUs.
        - This process is called virtualizing the CPU
    - You might also notice that the ability to run multiple programs at once raises all sorts of questions
        - If two programs want to run at a particular time, which should run?
            - This question can be answered by the policy of the OS; policies are used in many different places within an OS to answer these kinds of questions
            - Basic mechanisms that operating systems implement must not constrain these policies

Virtualizing Memory (2.2)

- The model of physical memory presented by modern machines is very simple
    - Memory is just an array of bytes
        - to read memory, one must specify the address to be able to access the data stored there
        - to write/update memory, one must also specify the data to be written to the given address
- Ex: a program that mallocs a certain amount of memory, prints out the address of the memory, and then also prints out the process identifier
    - If we run two instances of this program at the same time, the program seems to be updating the value at the same address space independently, as if the program has its own private memory instead of sharing the same physical memory with other programs
    - Each process accesses its own private virtual address space, which the OS maps onto the physical memory of the machine
    - A memory reference within one running program does not affect the address space of other processes.

    - The physical memory is a shared resource, managed by the operating system

Concurrency (2.3)

- Because the OS has to juggle so many things at once, the problems of concurrency (running things at the same time occur)
- These problems of concurrency can be seen in modern multi-threaded programs as well
    - Race conditions can occur depending on how many times you iterate through a program, global variables, etc.
    - The reason for this is because of how the instructions are executed. Because sometimes instructions do not execute atomically, strange things happen, leading to the problem of concurrency.

Persistence (2.4)

- In system memory, devices such as DRAM store values in a volatile manner, meaning that data can be easily lost
    - when power goes away or the system crashes, any data in memory is lost. Therefore, we need hardware and software to be able to store data persistently.
    - The hardware comes in the form of some kind of input/output or I/O device; in modern systems, a hard drive is a common repository for persistence, although solid-state drivers (SSDs) are also making head-way
- The software in the OS that usually manages the disk is called the file system; it is thus responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system
- The OS does not create a private, virtualized disk for each application
    - Rather, it is assumed that users want to share information that is within files
- Functions such as open(), creat(), close(), or write() are called system calls that are routed to the part of the OS called the file system, which then handles the request and returns some kind of error code to the user
- Device Driver: Some code in the OS that knows how to deal with a specific device

Design Goals (2.5)

- AN OS takes physical resources, such as CPU, memory, or disk, and virtualizes them. It handles tough and tricky issues related to concurrency, and stores files persistently, thus making them safe over the long term
- One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use. Abstractions make it possible to write a large program by dividing it into small and understandable pieces. Ex: write programs in high-level languages such as C without thinking of assembly, to write code in assembly without thinking about logic gates, etc.
- Another goal is provide high performance through minimizing the overhead that the OS has
- Another goal is to provide protection between applications, as well as between the OS and the applications.

    - Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behavior of one does not harm others;
    - Protection is at the heart of one of the main principles of an OS, which is isolation
        - isolation: isolating processes from one another is the key to protection
- The operating system must also run non-stop; when it fails, all applications running on the system will fail as well. Therefore, operating systems must have high reliability
- Energy-efficiency is also important in our increasingly green world
- Security against malicious attacks are also critical
- Mobility is also important as OSs are run on smaller and smaller devices