Lecture Overview

- Operating system software introduction
  - OS components
  - OS services
  - OS structure

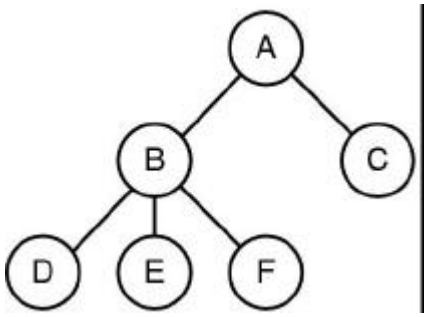
Operating System Components

- Process management
- Memory management
- Secondary storage management
- I/O system management
- File management
- Protection system
- Networking
- Command interpreter

Process Management

- A *process* is the unit of work in an operating system
  - Processes are active with a *program counter*, programs are not
  - The execution of a process is sequential
- A process uses resources provided by the OS, including CPU time, memory, files, and I/O devices, to accomplish its task
- The operating system is responsible for
  - Process creation and deletion
  - Process suspension and resumption
  - Providing mechanisms for
    - Process synchronization and communication

Process Management



All processes form a process tree

- A is a child of a special system process, *init*
- A has two child processes, B and C
- B has three child processes, D, E, and F

Memory Management

- Memory is a large array of words or bytes, each with its own address; it is a quickly accessible storage repository shared by the CPU and I/O devices
- Main memory is a *volatile* storage device; it loses its contents in the case of system failure
- The operating system is responsible for
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes to load when memory space becomes available
  - Allocating and deallocating memory space as needed

Secondary Storage Management

- Main memory (*primary storage*) is volatile and too small to accommodate all data and programs, the computer system must provide nonvolatile *secondary storage* to back up main memory
- Most modern computer systems use disks as the principle online storage medium, for both programs and data
- The operating system is responsible for
  - Free space management
  - Storage allocation
  - Disk scheduling

I/O System Management

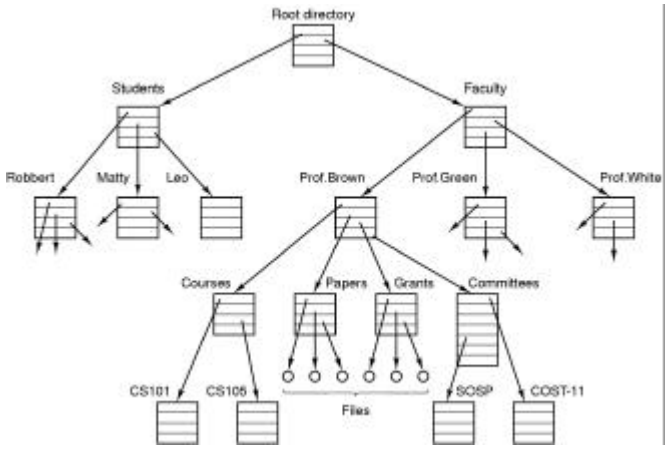
- The OS must hide all of the peculiarities of the underlying I/O devices
- The I/O system consists of
  - A general device-driver interface
  - A buffer-caching system
  - Drivers for specific hardware devices

File Management

- A file is a collection of related information defined by its creator; files represent programs and data
- Files are grouped into directories; directories may can directories as well as file, forming a hierarchy
- The operating system is responsible for
  - File creation and deletion
  - Directory creation and deletion
  - Support of primitives for manipulating files and directories
  - Mapping files onto secondary storage
  - File backup on stable (nonvolatile) storage media

File Management

An example of a file system hierarchy



Protection System

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources
- The protection mechanism must
  - Distinguish between authorized and unauthorized usage
  - Specify the controls to be imposed
  - Provide a means of enforcement

Networking

- A *distributed* system is a collection processors that do not share memory or a clock
- The processors in the system are connected through a communication network
- A distributed system provides user access to various system resources
- Access to a shared resource allows
  - Computation speed-up
  - Increased data availability
  - Enhanced reliability

Command Interpreter

- The operating system must provide some mechanism for users to submit commands for
  - Process creation and management
  - I/O handling
  - Secondary-storage management
  - Main-memory management
  - File-system access
  - Protection
  - Networking
- The program that reads and interprets commands is the *command-line interpreter* or *shell*
- The shell is not typically part of the operating system

Operating System User Services

- User/program services (mirrors system components to a large degree)
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communications
  - Error detection
- System services
  - Resource allocation
  - Accounting
  - Protection

How does the OS provide programs access to services?

System Calls

- *System calls* provide the interface between a running program and the operating system
  - Generally available as assembly-language instructions, but also available as functions in high-level languages
- Three general methods are used to pass parameters between a running program and the operating system
  - Pass parameters in *registers*
  - Store the parameters in a table in memory, and the table address is passed as a parameter in a register
  - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system

Classes of System Calls

- Process management
  - Create, terminate, wait for process
- File management
  - Create, delete, read, write files
- Device management
  - Read, write, access devices
- Information maintenance
  - Get/set time, system data, file/process attributes
- Communication
  - Create communication connections; send, receive messages; shared memory

POSIX System Calls

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

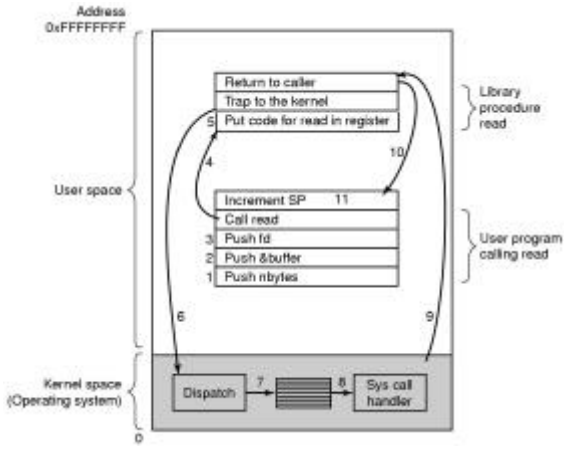
Win32 System Calls

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

API versus system call - APIs may not map one-to-one to OS system calls, in fact, they may be completely different

Performing a System Call

Consider the `read(fd, buffer, nbytes)` system call



Operating System Design

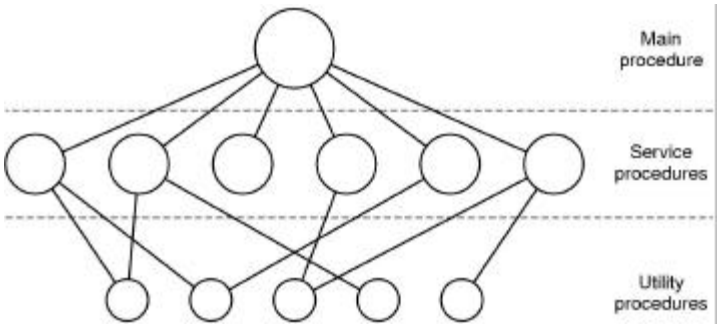
- Like all programs, operating systems are built using some architectural design approach
- The architectural design of an OS affects its overall strengths and weaknesses
- Due to the complexity of an OS, choosing a good design approach is important
  - "Good" is relative to your goals

Monolithic Operating System Design

Monolithic approach (this includes Linux)

- "The Big Mess" [Tanenbaum]
- A collection of procedures in a single object/executable file
- No information hiding

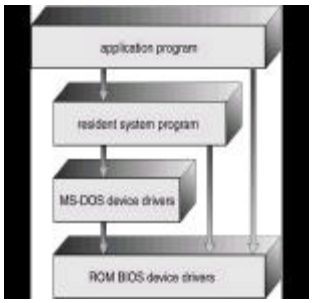
The generic structure of a monolithic OS



Monolithic Operating System Design

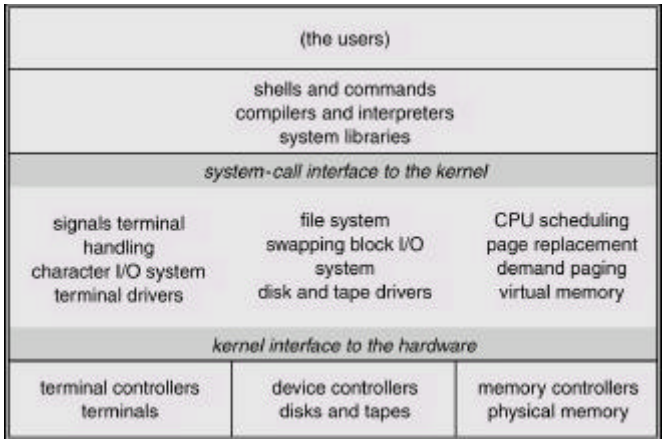
MS-DOS was written to provide the most functionality in the least space

- Not divided into modules
- Interfaces and levels of functionality are not well separated
- Limited by its hardware (Intel 8088)
- Victim of its own success



Monolithic Operating System Design

Unix is typically separated into two parts, the kernel and the system programs



Layered Operating System Design

- The operating system is divided into a number of layers (levels), each built on top of lower layers
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Structure of the THE operating system

MULTICS also used a similar layered approach

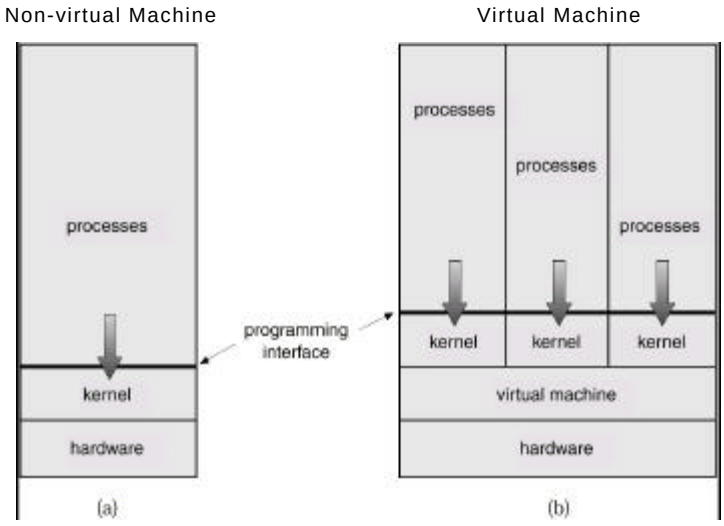
Virtual Machine OS Design

- A timesharing OS provides multiprogramming and an extended machine with a convenient interface; it is possible to separate these two functions
- A *virtual machine* provides multiprogramming only by providing an exact virtual copies of the bare hardware
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory
- Each virtual machine can run any OS on top of it

Virtual Machine OS Design

- The resources of the physical computer are shared to create the virtual machines
  - CPU scheduling creates the appearance that users have their own processor
  - Resource allocation and sharing creates the appearance of virtual devices for I/O, files, etc.
- This is Java's approach, it is also similar to processor emulation modes

Virtual Machine OS Design



Virtual Machine OS Design

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

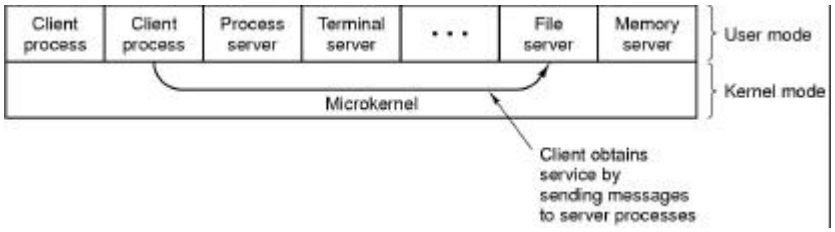
Client-Server Operating System Design

Microkernels/client-server model

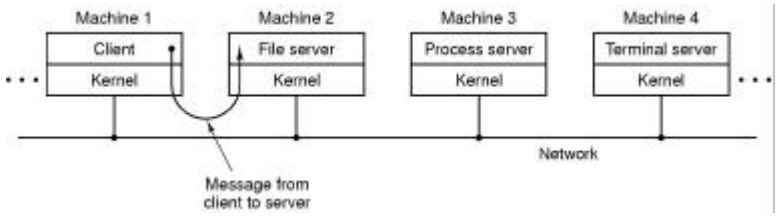
- Key concept: the kernel should be as small as possible (e.g., process, memory, and communication management), everything else should be in user space
- Communication among modules is provided via message passing using a client-server paradigm
- Allows for better protection since services cannot affect the rest of the operating system
- Very adaptable for use in distributed systems
- It is not always possible to have a clean separation between a service and the kernel
- This approach adds overhead
- Examples: Mach, Mac OS X, GNU Hurd

Client-Server Operating System Design

Microkernel approach in a local setting



Microkernel approach in a distributed setting



Linux Approach

- Linux is monolithic but…
  - It has the notions of modules
  - A *module* is an object file whose code can be linked to (and unlinked from) the kernel at runtime
  - The module executes in kernel mode
  - Modules export symbols and depend on other modules
  - Modules are managed with reference counting
  - Modules can be linked on demand
- Modules provide many of the same benefits as the microkernel approach, but it mostly offers a degree of modularity and performance, not protection or applicability to distributed systems