Chapter 2: System Structures

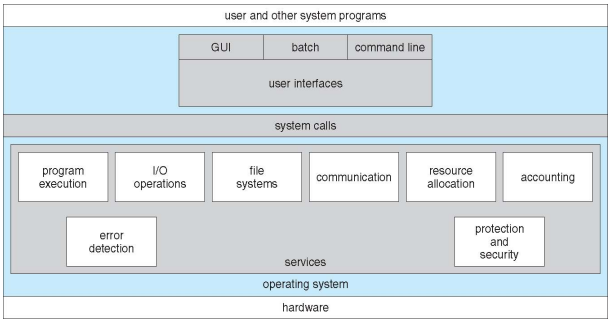
Chapter 2: Operating System Structures

- Operating System Services
- System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines

Objectives

- To describe the services that an operating system provides to users
- To discuss the various ways of structuring an operating system

A View of Operating System Services



Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), batch interface (file execution)
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device. User can not access I/O device directly
  - File-system manipulation - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Operating System Services (Cont)

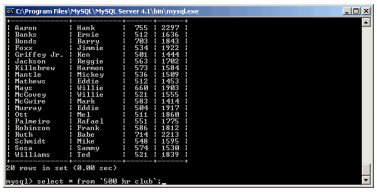
- One set of operating-system services provides functions that are helpful to the user (Cont):
  - Communications – Processes may exchange information between processes, on the same computer or different computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - Error detection – OS needs to detect and correct errors constantly
    - May occur in the *CPU* and *memory hardware* (memory error or power failure), in *I/O devices* (parity error on a disk, network connection problem, lack of papers on a printer), in *user programs* (arithmetic overflow, an attempt to access illegal memory location)
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing

Operating System Services (Cont)

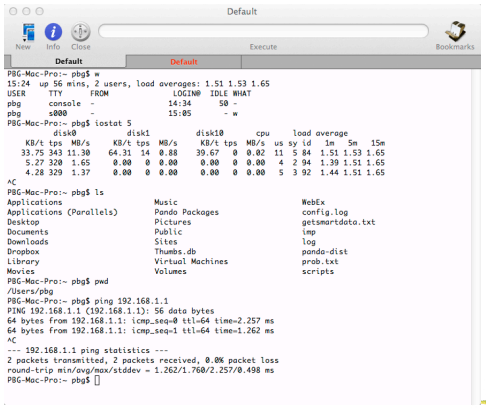
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - Resource allocation - When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - Accounting - To keep track of which users use how much and what kinds of computer resources
  - Protection and security - The owners of information stored may want to control use of that information, concurrent processes should not interfere with each other
    - Protection involves ensuring that all access to system resources is controlled
    - Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

User Operating System Interface - CLI

- Command Line Interface (CLI) or command interpreter allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple command interpreters to choose from – **shells**
  - Primarily fetches a command from user and executes it
  - UNIX, Linux and MS DOS



Bourne Shell Command Interpreter



User Operating System Interface - GUI

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions such as invoke a program, select a file or directory, or pull down a menu that contains commands
  - Invented at Xerox PARC in earlier 1970s, and widely used in Apple Macintosh computers in the 1980s
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available (CLI)
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

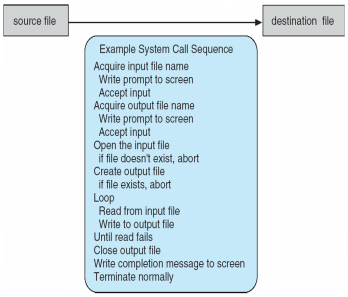
The Mac OS X GUI

Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++), certain low-level tasks (i.e., accessing hardware) may have to be in assembly languages
- An example – system call sequence to copy the contents of one file to another file
- Simple programs make heavy use of the OS. Frequently, systems execute thousands of system calls per second
- Hide this level of details from programmers



System Calls - API

- Application Program Interface (API) specifies a set of functions that are available to an application programmer, including the parameters passed to the function and return values it expects
- The functions that make up an API typically invoke the actual system calls on behalf of the application programmer.
- Three most common APIs
  - Win32 API for Windows systems
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)
- Why use APIs rather than invoking system calls?
  - Hide the complex details of the system call from users
  - Program portability

Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t   read(int fd, void *buf, size_t count)
```

return value | function name | parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

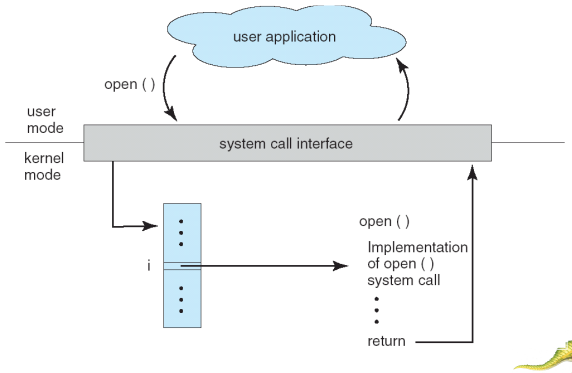
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns –1.

System Call Implementation

- For most programming languages, the run-time support system (a set of functions built into libraries) provide a system call interface that serves as the link to system calls of OS
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface intercepts function calls in the API, invokes the necessary system call in OS kernel, and returns status of the system call and return value(s) if any
- The caller need know *nothing* about how the system call is implemented
  - needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API

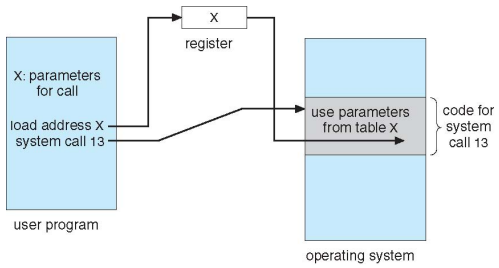
API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply the identity of desired system call
  - The exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

Types of System Calls (Cont.)

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes

Types of System Calls (Cont.)

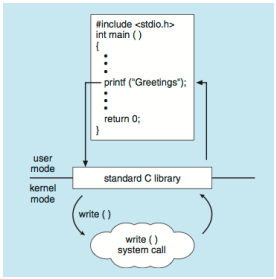
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Standard C Library Example

- The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux.
- C program invokes printf() statement, the C library intercepts this call and invokes the necessary system call write(), takes the return value and pass to the user program



System Programs

- System programs, or system utilities, provide a convenient environment for program development and execution. They can be divided into:
  - **File manipulation**: create, delete, copy, rename, print, dump, list. …
  - **Status information**: time/date, memory usage, logging ad debugging info.
  - **File modification**; file editor for example
  - **Programming-language support**: compiler, assemblers, debuggers
  - **Program loading and execution**
  - **Communications**
  - **Background services**: for example constant running system program processes known as services, subsystems, or daemons.
  - **Application programs**: Web browsers, word processors, spreadsheets, database systems, games. ..
- Most users' view of an operation system is defined by system programs, not the actual system calls

System Programs (Cont.)

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information

System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Programs (Cont.)

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke
  - Web browsers, word processors, spreadsheets, database systems, games

Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven to be successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system (batch, time sharing, single user, multiuser, distributed, real time, or general purpose)
- User goals and System goals – much harder to specify
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont.)

- Important principle to separate
  Policy: ***What*** will be done?
  Mechanism: ***How*** to do it?
- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later
- Specifying and designing OS is highly creative task of **software engineering**
- Policy decisions are important for all resource allocations

Implementation

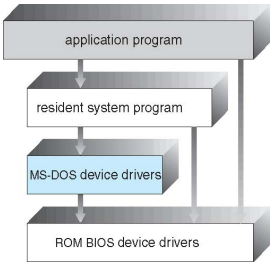
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now most operating systems are written in C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware
  - The code can be written faster, more compact, easier to debug
  - But slower and might require more storage

Operating System Structure

- General-purpose OS is a very large program, must be carefully engineered to function properly and be modified easily
- A common approach is to partition the task into small components, or modules, rather than have one monolithic system.
  - Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions
- Various ways to structure one as follows

Simple Structure

- Such Oses do not have well-defined structure, usually started as small, simple and limited systems
- MS-DOS – written to provide the most functionality in the least space
  - Not carefully divided into modules
  - Although it has some structure, interfaces and levels of functionality are not well separated – i.e., app programs can access I/O directly
  - Written for Intel 8088 with no dual mode and no hardware protection

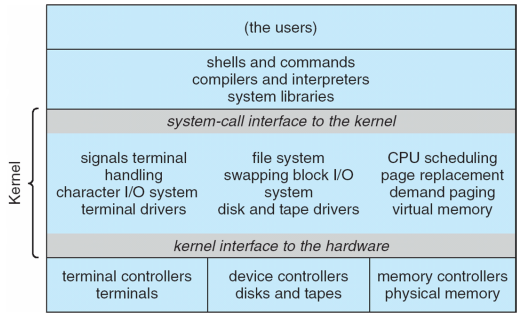


UNIX

- UNIX – initially limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts: the kernel and system programs
- The kernel is further separated into a series of interfaces and device drivers, which have been expanded over the years.
- In traditional UNIX, the kernel consists of everything below the system -call interface and above the physical hardware
  - It provides the file system, CPU scheduling, memory management, and other operating-system functions; an enormous amount of functionality combined into one level. This monolithic structure makes it difficult to implement and maintain
  - A distinct performance advantage is that there is little overhead in the system call interface or in communication with the kernel

Traditional UNIX System Structure

Beyond simple but not fully layered

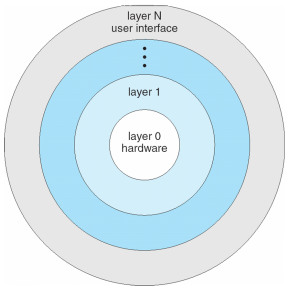


Layered Approach

- One type of modular approaches
- 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
- The main advantage of a layered approach the simplicity of construction and debugging. The layers are selected such that each uses functions (operations) and services of only lower-level layers
- The major difficulty involves appropriately defining the various layers. Also this tends to be less efficient as each layer adds overhead – Few layers with more functionality in recent years.

Layered Approach (Cont.)

- An OS layer is an implementation of an abstract object made up of data and operations manipulating those data
- A layer M consists of data structure and a set of routines that can be involved by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers
- Information hiding: a layer does not need to know how the lower-layer operations are implemented, only what these operations do.



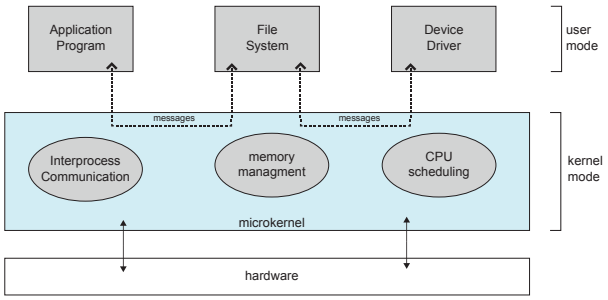
Microkernel System Structure

- The kernel became large and difficult to manage
- Removing all nonessential components from the kernel and implementing them as system or user-level programs
- This results in a smaller kernel. Yet, there is little consensus regarding which services should remain in the kernel and which should be implemented in user space
  - Typically, microkernels provide minimal process and memory management, in addition to a communication facility.
- Mach, developed at CMU in mid-1980s, is an example of microkernel
- The main function of the microkernel is to provide communication between client program and various services also running in user space
- Communication is provided through message passing

Microkernel System Structure (Cont.)

- Benefits:
  - Easier to extend a microkernel OS, as all new services are added to user space without modification on the kernel
  - When the kernel has to be modified, the changes are fewer, as the kernel is much smaller
  - Easier to port the operating system to new architectures (hardware)
  - More reliable and reliable (less code is running in kernel mode), since most services are running as user – not kernel processes. If a service fails, the rest of the OS remains untouched
  - Mac OS X kernel (also known as Darwin) is partly based on Mach kernel
- Detriments:
  - Performance of microkernels can suffer due to increased system -function overhead, user space to kernel space communication
  - Earlier Window NT had a layered microkernel, now more monolithic

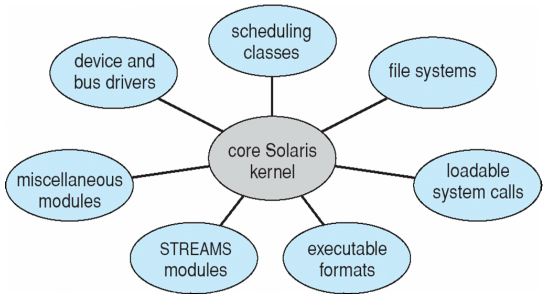
Microkernel System Structure



Modules

- Perhaps the best current methodology for OS design involves using **loadable kernel modules**
- It provides core services while other services are dynamically implemented as the kernel is running
  - Recompiling the kernel is required each time new features are added
  - For example, the kernel has CPU scheduling and memory management algorithms, and adds support for different file systems by way of loadable modules
- This resembles a layered system in that each kernel section has defined, protected interface, but it is more flexible as any module can call any other module (no higher or lower layer relationship)
- It is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but more efficient because modules do not need to invoke message passing in order to communicate
- This type of design is common in modern implementation of UNIX, such as Solaris, Linux, and Mac OS x, as well as Windows

Solaris Modular Approach



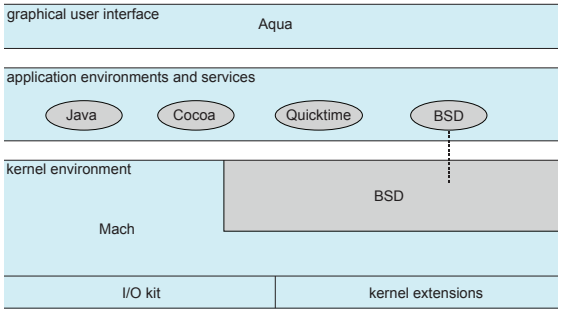
Hybrid Systems

- In practice, very few operating systems adopt a single, strictly defined structure. Instead they combine different structures, resulting in hybrid systems that address performance, security, usability needs
  - Both Linux and Solaris are monolithic, because having the OS in a single address space provides very efficient performance. They are also modular for dynamic loading of new functionality
  - Windows largely monolithic (primarily performance reason), but retains some behavior typical of microkernel systems, including providing support for different subsystems (known as OS personalities) that run as user -mode processes. Windows systems also provide support for dynamically loadable kernel modules.

Mac OS X

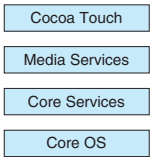
- Apple Mac OS X uses a hybrid structure.
- It is a layered system
- The top layers include **Aqua** user interface and a set of application environments and services. **Cocoa** environment specifies an API for the Objective-C language, used for writing Mac OS X applications
- Below is *kernel environment*, consisting of the Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called *kernel extensions*)
  - MACH microkernel provides memory management support for RPC, IPC
  - BSD components provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads

Mac OS X Structure



iOS

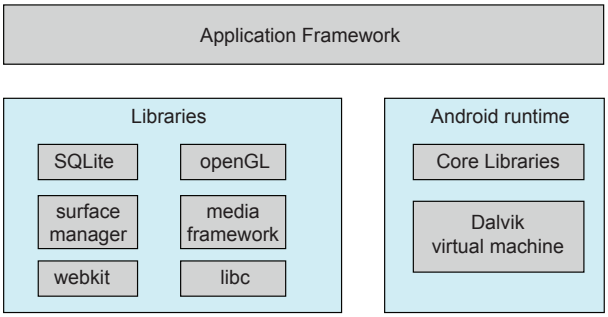
- Apple mobile OS for ***iPhone***, ***iPad*** (close-sourced)
  - Structured on Mac OS X, added functionality pertinent to mobile devices
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps (touch screen features)
  - **Media services** layer for graphics, audio, video
  - **Core services** provides support for cloud computing, databases
  - **Core OS** represents the core operating system, which is based on Mac OS X kernel environment



Android

- Developed by Open Handset Alliance (led primarily by Google)
  - Open-source
- Similar stack to iOS in that it is a layered stack of software
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java (non-standard Java API, based on Android API), run on Dalvik virtual machine
  - Dalvik optimized for mobile devices with limited memory and CPU
- Libraries include frameworks for web browser (webkit), database (SQLite), and multimedia,
- The libc library is similar to standard C library, but much smaller, designed for slower CPUs in mobile devices

Android Architecture



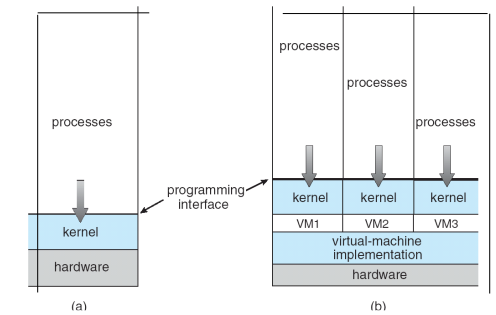
Virtual Machines

- The virtual machine creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory
- Software emulation of an abstract machine
  - Make it look like hardware that 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
- Fault Isolation
  - Processes unable to directly impact other processes
  - Bugs cannot crash whole machine
- Protection and Portability
  - Java virtual machine: Java interface safe and stable across many platforms

Virtual Machines (Cont)

- Allows operating systems to run as applications within other OSes
- **Emulation** used when source CPU type different from target type
  - Generally slowest method, as each machine-level instruction must be translated into equivalent instruction on the target system
  - For example, Apple moved from IBM PowerPC to Intel x86 CPU, it included an emulation facility that allowed applications complied for the IBM CPU to run on the Intel CPU
- **Virtualization** – OS running as guest OS within another OS (host)
  - VMM or Virtual Machine Manager provides virtualization services. It has more privileges than user processes, but fewer than the kernel (more than the dual-mode). VMM runs the guest operating system, manages their resource use, and protect each guest fom others
  - Consider VMware running multiple WinXP guests, each running applications, in which Windows is the host OS, VMware application is the VMM.

Virtual Machines (Cont)



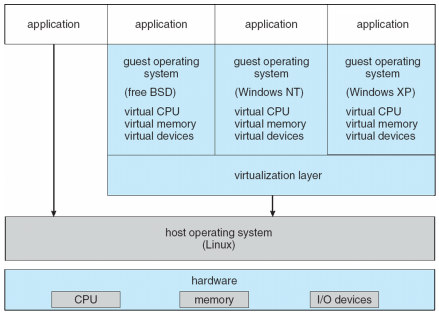
(a) Nonvirtual machine (b) virtual machine

Virtual Machines (Cont)

- Use cases involve laptops and desktops running multiple OSes for exploration or to run applications written for operating systems other than the native host (machine)
  - Apple laptop running Mac OS X host, Windows as a guest to allow execution of Windows application
  - Multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging.
  - Virtualization has become a common method of executing and managing compute environments within data centers

Virtual Machines (Cont): Layers of OS

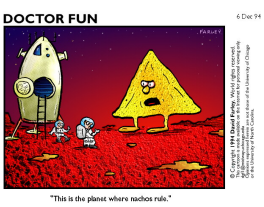
- Useful for OS development
  - When OS crashes, restricted to one VM
  - Can aid testing programs on other OS



Nachos: Virtual OS Environment

You will be working with **Nachos**

- Instructional software allows to study and modify OS functions. It runs as a UNIX process, while a real operating system runs on hardware
- Nachos simulates the general low-level facilities of typical hardware, including interrupts, virtual memory and interrupt-driven I/O
- To test the concepts you learn about thread, multiprogramming, virtual memory, file systems and networking
- Machos written in C++ and well organized, making it easier to understand the operation of a typical operating system



End of Chapter 2