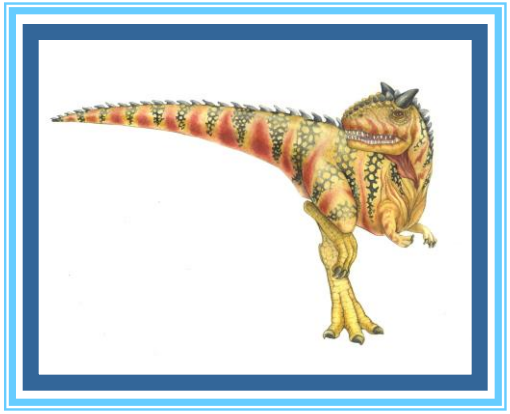
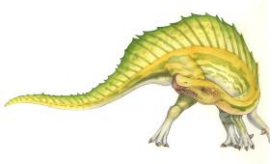
Chapter 2b: Operating-System Design and Implementation



Outline

- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

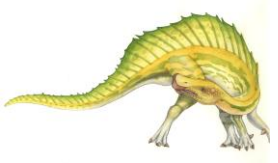


Objectives

- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems
- Illustrate the process for booting an operating system
- Apply tools for monitoring operating system performance
- Design and implement kernel modules for interacting with a Linux kernel

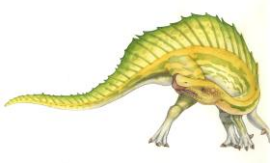
Design and Implementation

- Design and Implementation of OS is not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - 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
- Specifying and designing an OS is highly creative task of **software engineering**



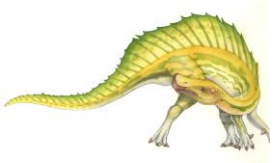
Policy and Mechanism

- **Policy**: **What** needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism**: **How** to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.
  - Example: change 100 to 200



Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now 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
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

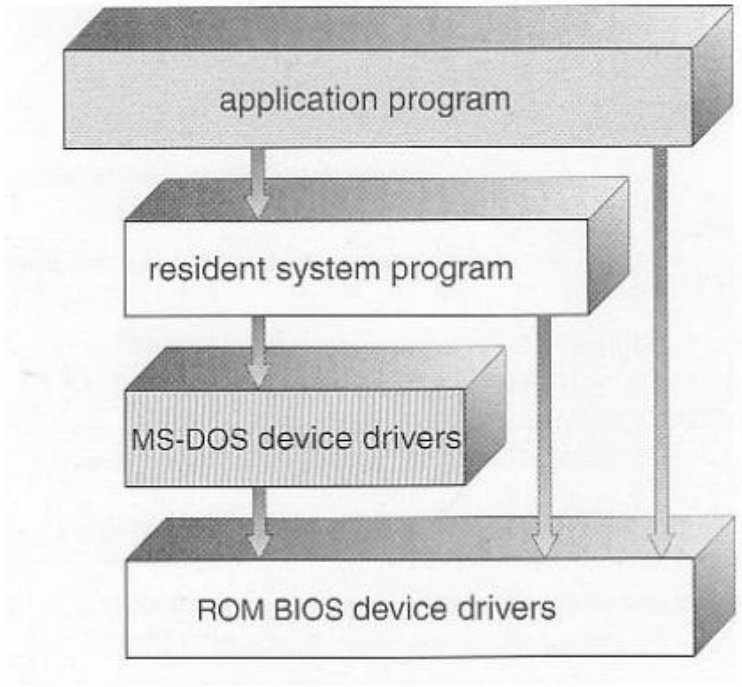


Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

Simple Structure

- In MS-DOS, the interfaces and levels of functionality are not well separated.
- Application programs can access basic I/O routines to write directly to the display and disk drives.
- Such freedom leaves MS-DOS in bad state and the entire system can crash down when user programs fail.



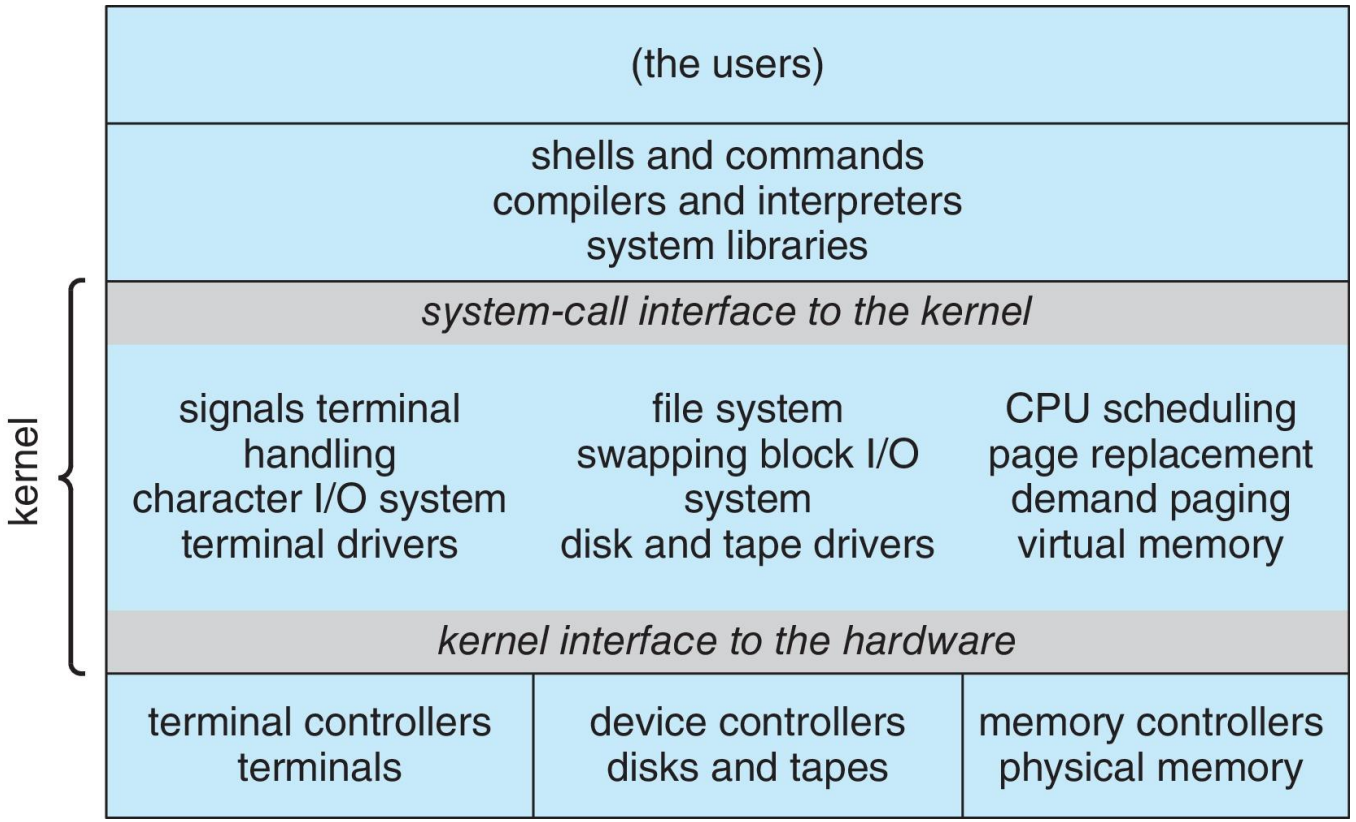
MS-DOS Layer Structure

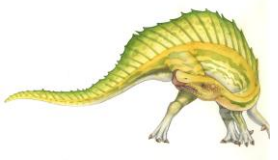
Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Traditional UNIX System Structure

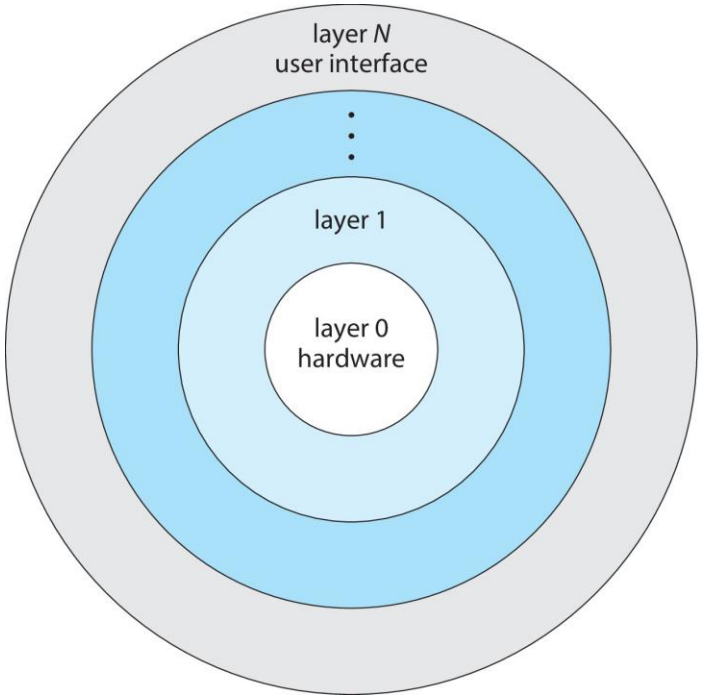
Beyond simple but not fully layered

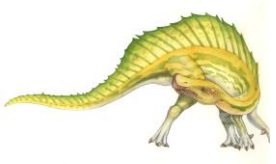




Layered Approach

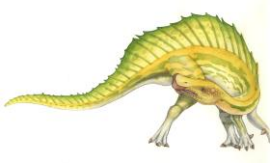
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Merits: simpicity in construction and debugging. The implementation details are hidden from higher layers.
- Demerits: defining laters and tasks, overhead of passing through intermediary layers.



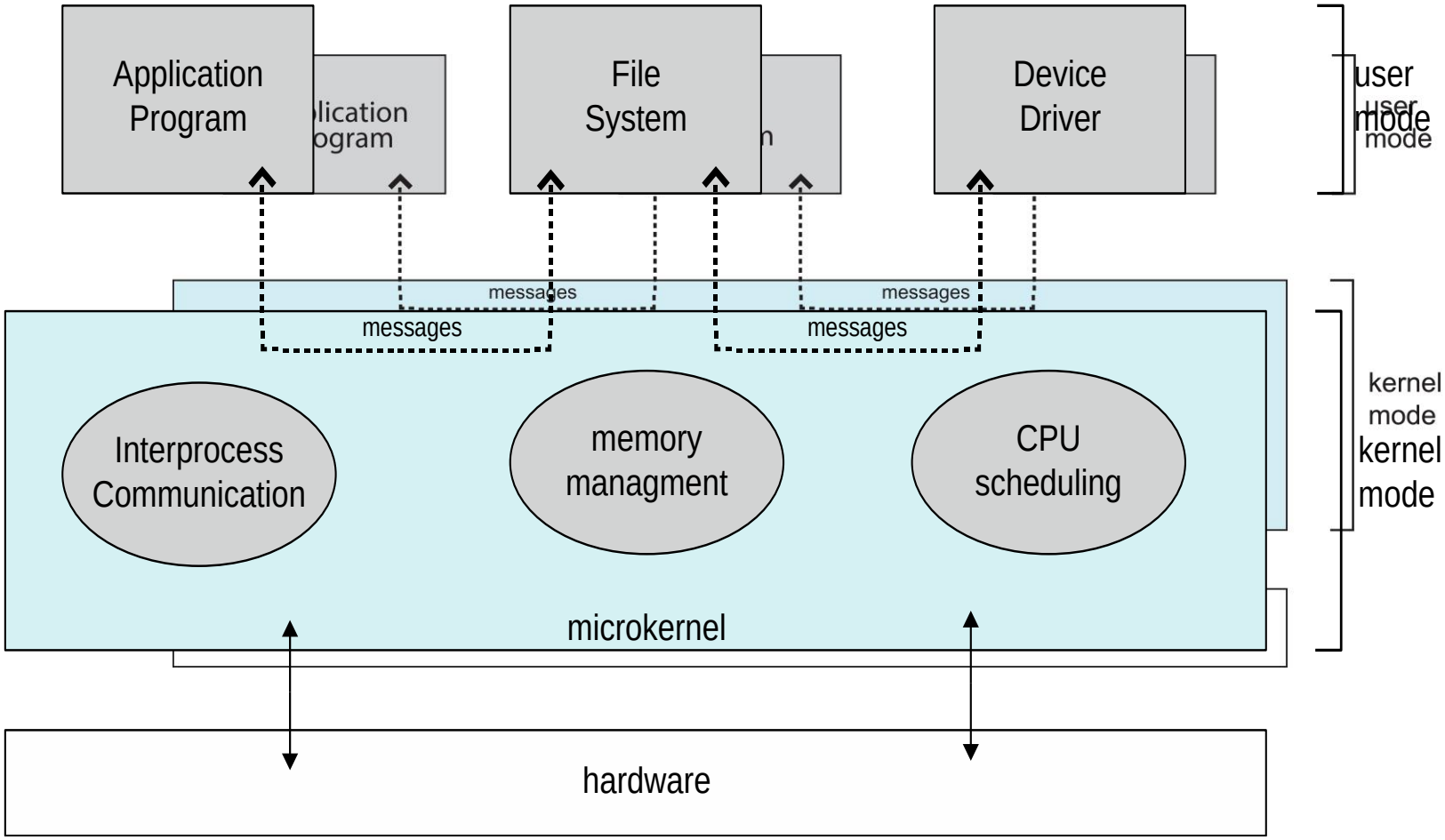


Microkernels

- Basic idea: **remove all non-essential services** from the kernel
- Making the kernel as small and efficient as possible
- Moves as much from the kernel into user space and implemented as system applications
- Provide basic services like process and memory management with meassage passing between other services
- **Mach** is widely used **microkernel** for Mac OS X
- Benefits:
  - Easier to extend a microkernel - adding more system applications and rebuilding a new kernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode), secure
- Detriments:
  - Performance overhead of user space to kernel space communication

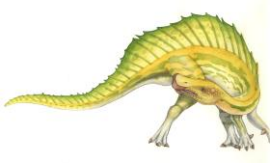


Microkernel System Structure



Modules

- Many modern operating systems implement **loadable kernel modules** (**LKMs**)
  - Uses object-oriented approach with a relatively small core kernel and set of modules which can be linlked dynamically
  - Modules are similar to layers with well defined tasks and interfaces
  - But, unlike layered approach each module can communicate with any other module without going through intermediary layers
  - Each module talks to the others over known interfaces hence kernel odes not have to implement message passing
  - Each module is loadable as needed within the kernel
  - Resembles Microkernel with primary module having core functions and the knowledge of how to load and communicate with other modules
- Overall, similar to layers but with more flexible



Modules

Eg. Linux, Solaris, etc.

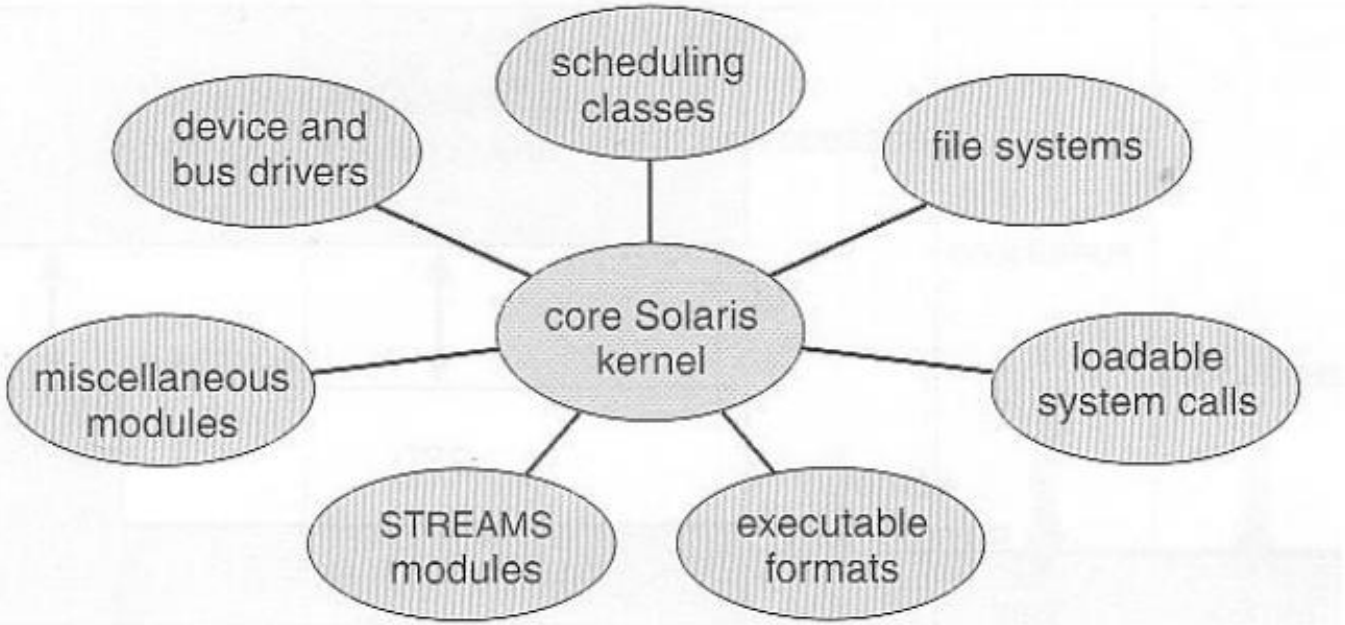
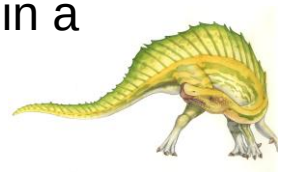


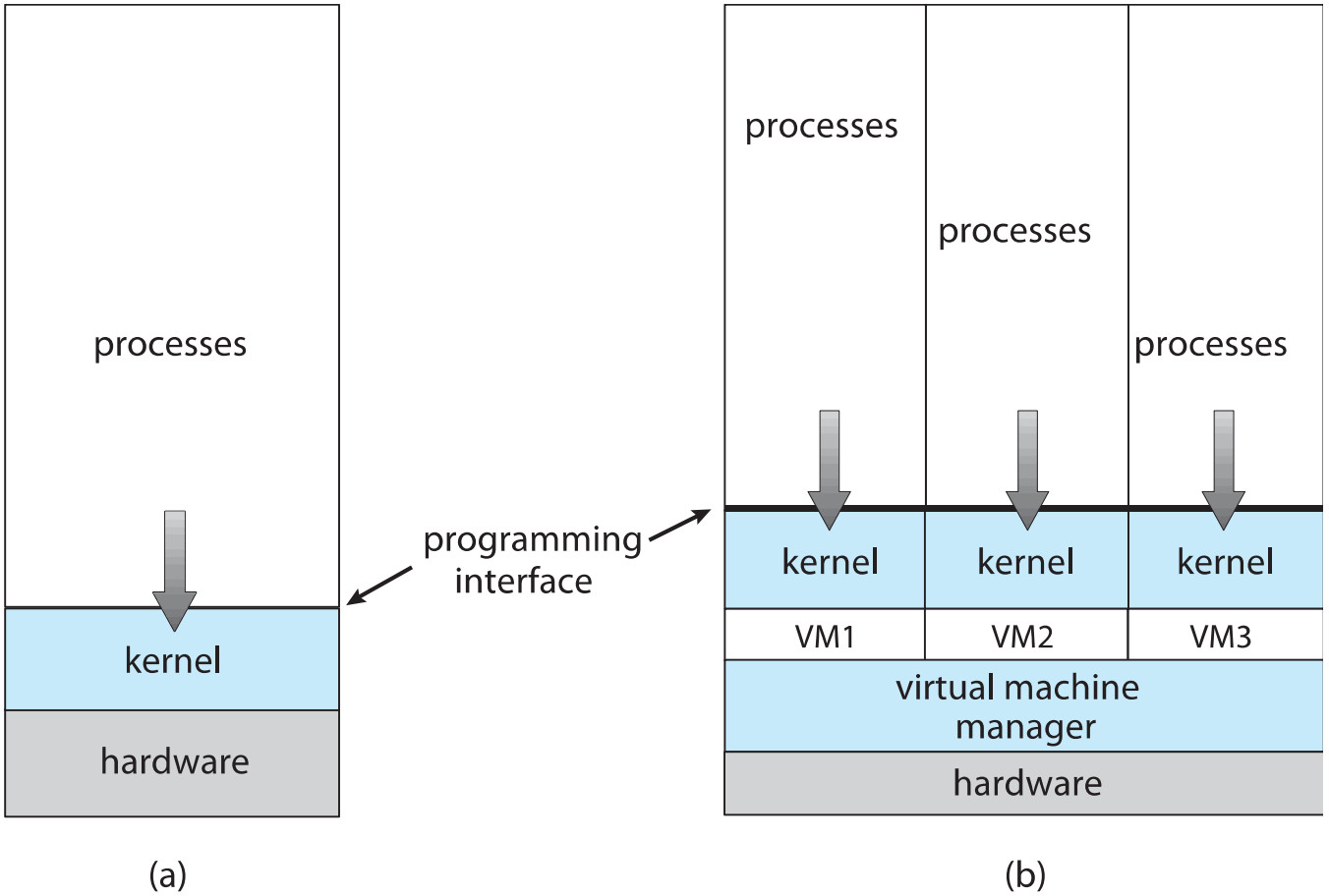
Figure 2.15 Solaris loadable modules

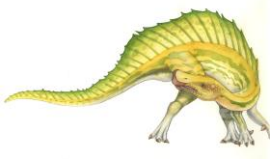
Virtual Machine

- The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.
- Creates an illusion that a process has its own processor with its own memory.
- Host OS is the main OS installed in system and the other OS installed in the system are called guest OS.
- Benefits:
  - Share the same hardware and run several different execution environments(OS)
  - Host system is protected from the virtual machines and the virtual machines are protected from one another
  - Developing apps for multiple OSes without having multiple systems
  - System consolidation: two or more systems are made to run in a single system



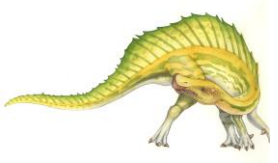
Virtual Machines





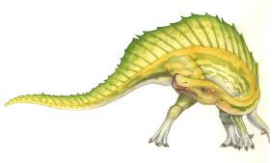
Building and Booting an Operating System

- Operating systems generally designed to run on a class of systems with variety of perpherals
- Commonly, operating system already installed on purchased computer
  - But can build and install some other operating systems
  - If generating an operating system from scratch
    - Write the operating system source code
    - Configure the operating system for the system on which it will run
    - Compile the operating system
    - Install the operating system
    - Boot the computer and its new operating system



System Boot

- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface** (**UEFI**)
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode



System Boot

- Upon start-up, the BIOS (Bootstrap program) goes through the following sequence:
- detects physical resources and does Power-on self-test (POST) of H/W
- Detect the video card's (chip's) BIOS and execute its code to initialize the video hardware
- Detect any other device BIOSes and invoke their initialize functions
- Display the BIOS start-up screen
- Perform a brief memory test (identify how much memory is in the system)
- Set memory and drive parameters
- Configure Plug & Play devices (traditionally PCI bus devices)
- Assign resources (DMA channels & IRQs)
- Identify the boot device
- When the BIOS identifies the boot device (typically one of several disks that has been tagged as the bootable disk), it reads block 0 (containing larger boot program) from that device into memory location 0x7c00 and jumps there.

End of Chapter 2b