# Today's Plan

## Upcoming:

- Quiz 1 Monday!
- Assignment 1

## Last time:

- Operating system duties

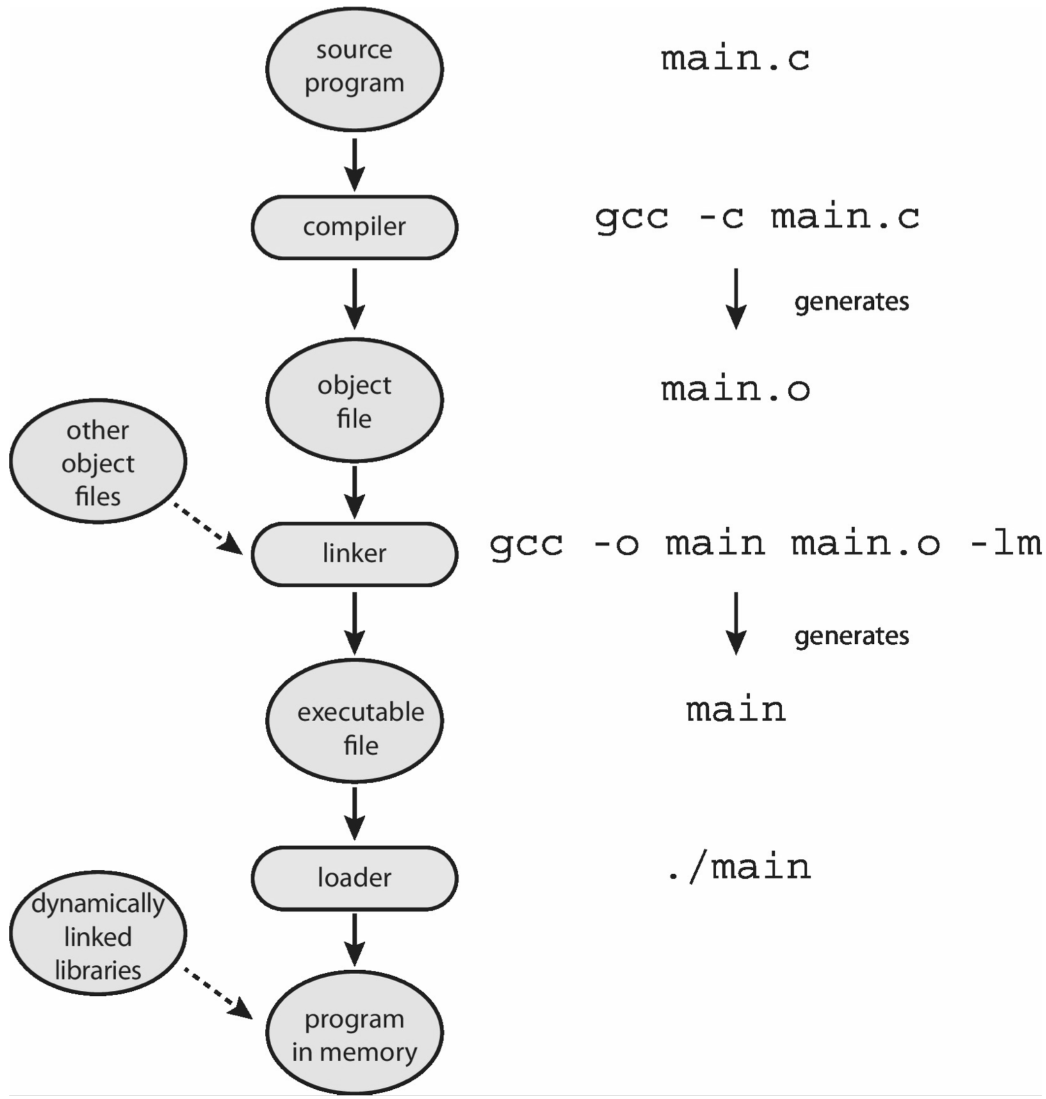
## Today's topics:

- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Introduction to Processes

# System Programs

- System programs provide a convenient environment for program development and execution. Examples:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of the operating system is defined by system programs, not the actual system calls.

# Example: The Linker and Loader



# Operating System Design & Implementation

- *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
  - Flexible, reliable, error-free
  - <u>Efficient!</u>

# Operating System Design & Implementation

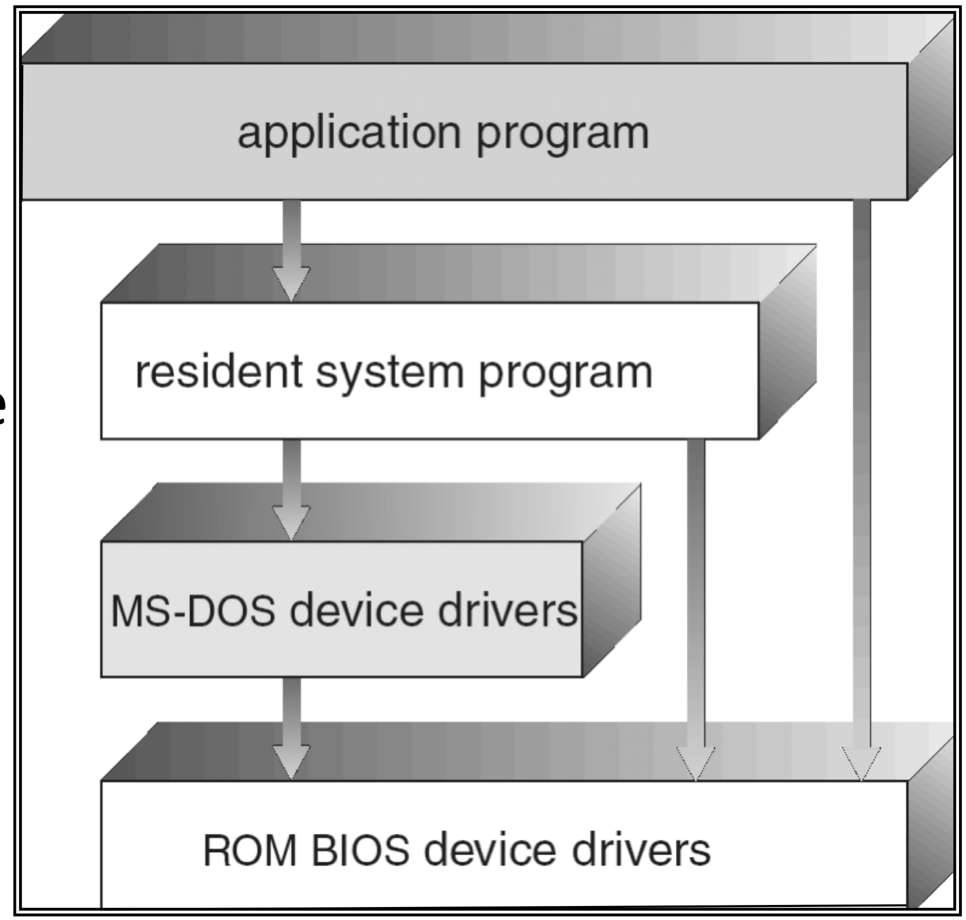
- Important principle to separate
  - **Policy:** What will be done?
  - **Mechanism:** How to do it?
- Why have this separation?
  - Allows maximum flexibility if either policy or mechanism need to be changed later

# Operating System 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
- High-level languages easier to port to other hardware
  - But slower
- Emulation can allow an OS to run on non-native hardware

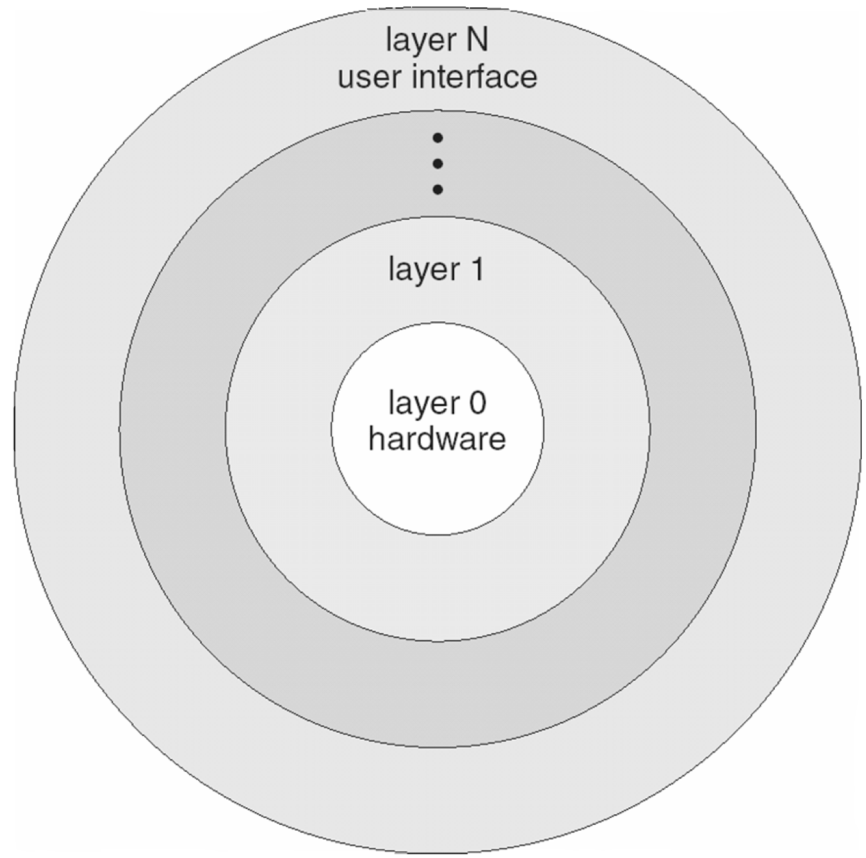
# Simple Structure

- MS-DOS – written to provide the most functionality in the least space
    - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



# Layered Approach

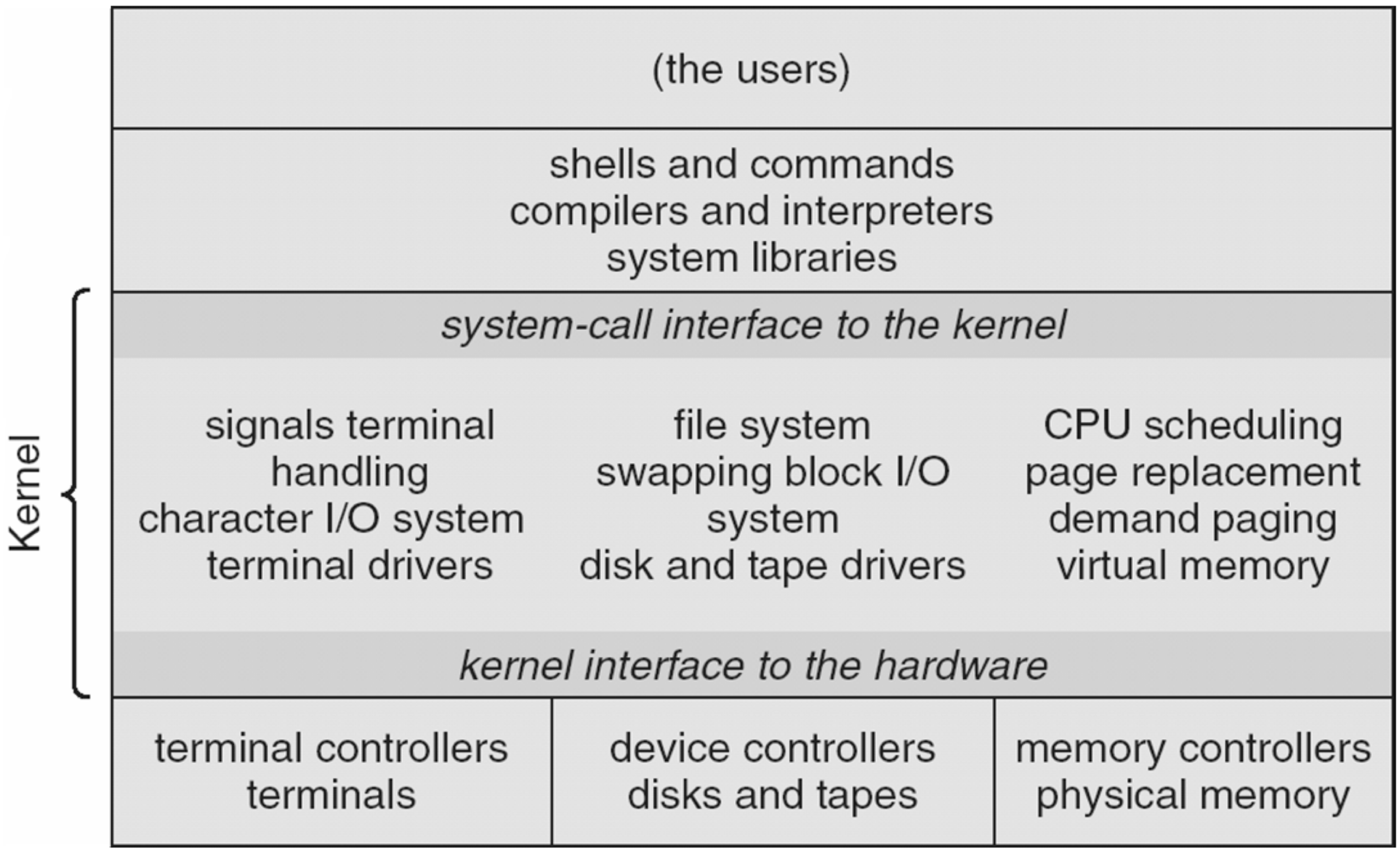
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions and services of only lower-level layers



# UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system was a *monolithic kernel*.
- 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 Structure

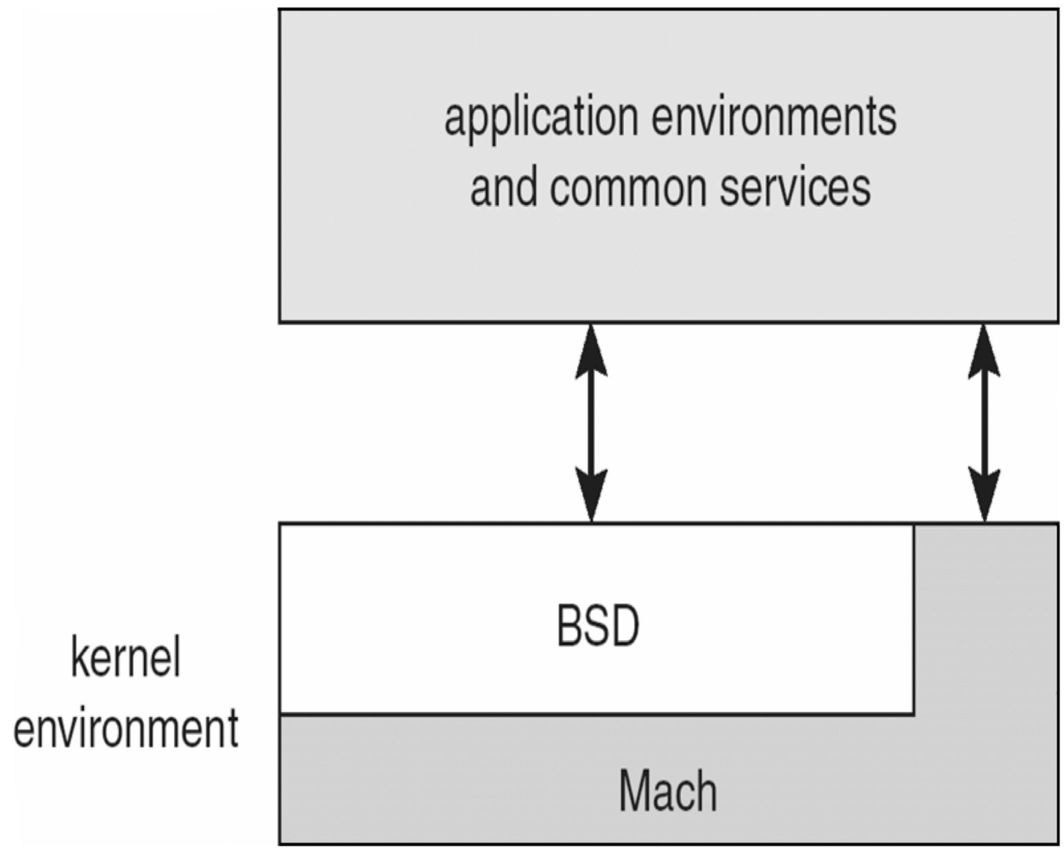


# Microkernel System Structure

- Moves as much from the kernel into "*user*" space
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the OS to new architectures
  - More reliable
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

# Monolithic vs. Microkernel Structure

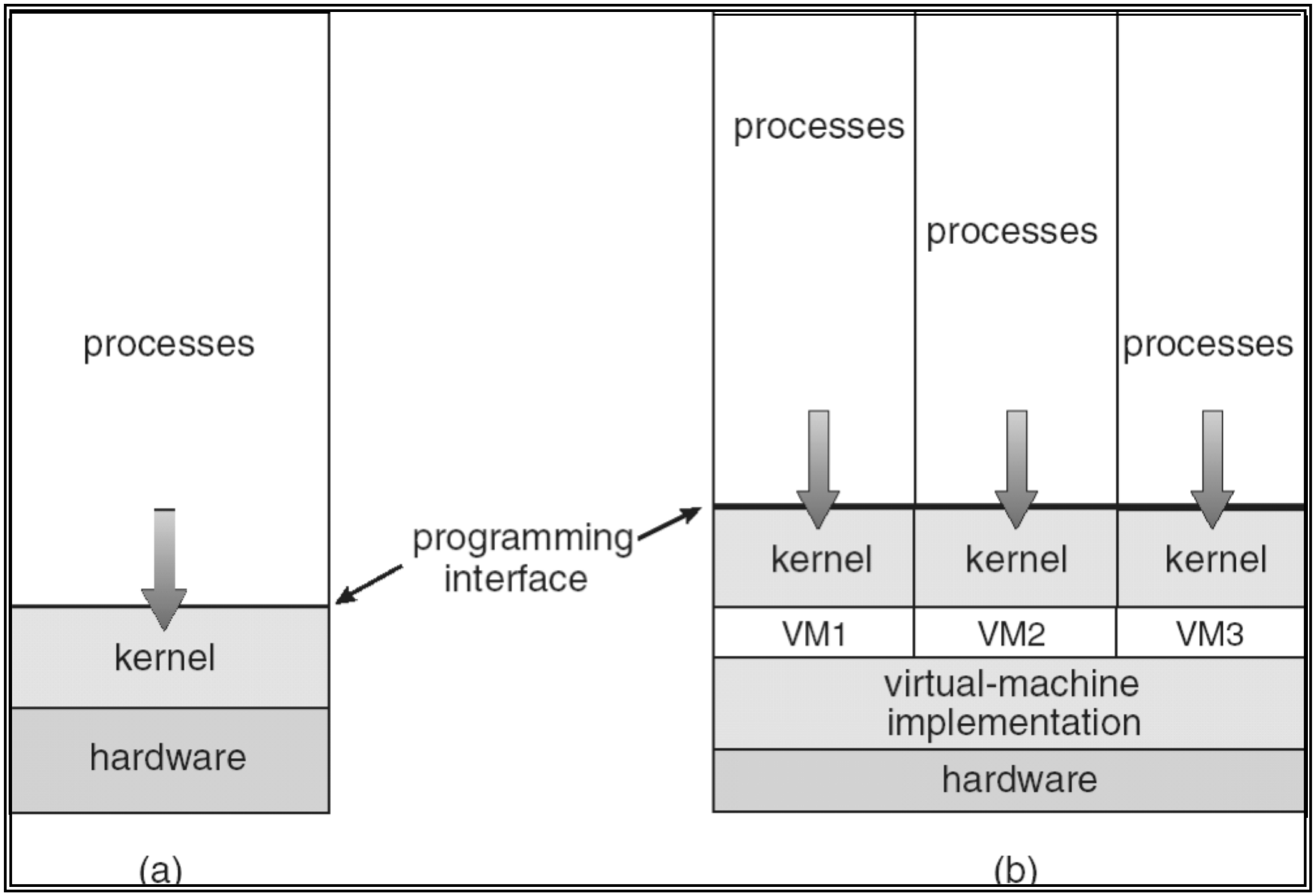
- Most popular modern OSes are actually hybrids of the monolithic and microkernel structures:
  - Many functions are moved into "user" space
  - Some are kept in the kernel for performance reasons
  - E.g. Mac OS/X structure:



# Virtual Machines

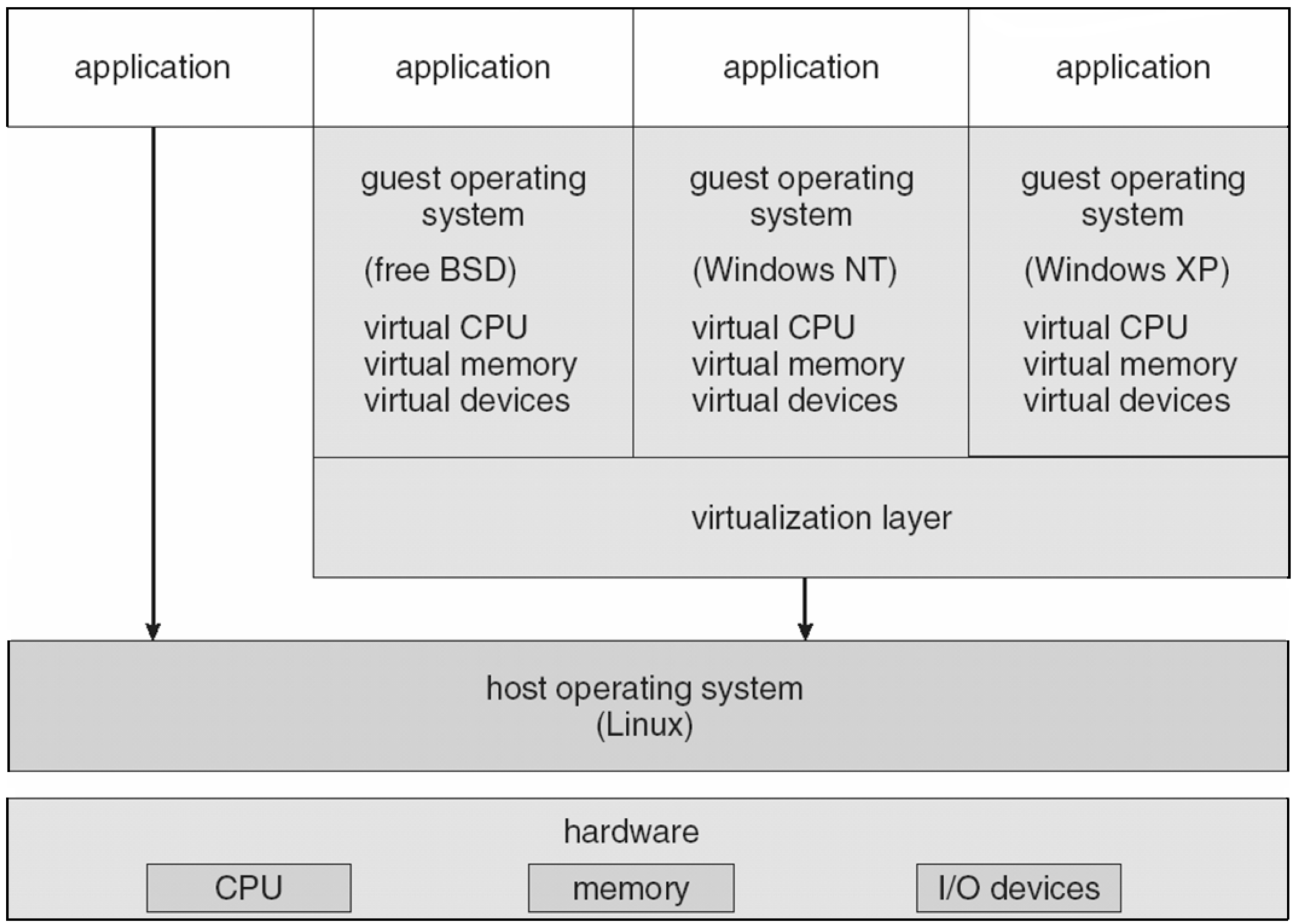
- A *virtual machine* takes the layered approach to its logical conclusion.  It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying hardware
- The operating system host creates the illusion that a process has its own processor and memory
- Each guest is provided with a (virtual) copy of underlying computer

# Virtual Machines

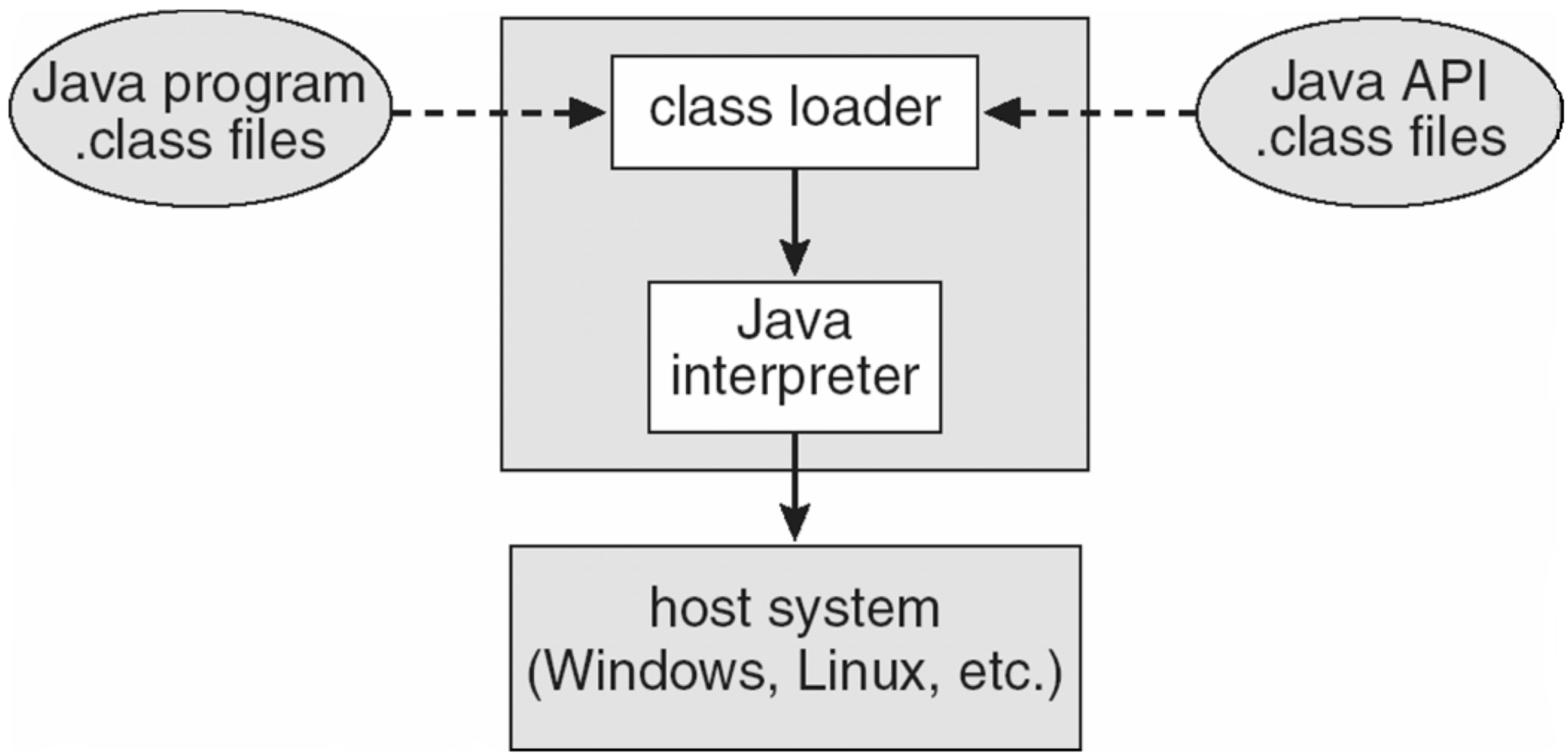



(a) Nonvirtual machine (b) virtual machine

# VMWare Architecture



# Java Virtual Machine



# Operating-System Debugging

- *Debugging* is finding and fixing errors, or bugs
- OSes generate log files containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Kernighan's Law: "*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*"

# Processes

- Process Concept
- Concurrency
- Race Conditions
- Process Creation
- Interprocess Communication
- Examples of IPC Systems

# What is a Process?

- Fundamental building block of modern operating systems is the notion of a *process*
- A process is a running program (a program in execution). This includes:
  - All programs running on behalf of users (application programs)
  - Some operating system functions are also implemented using processes
- A process is a single thread of execution under control of the OS

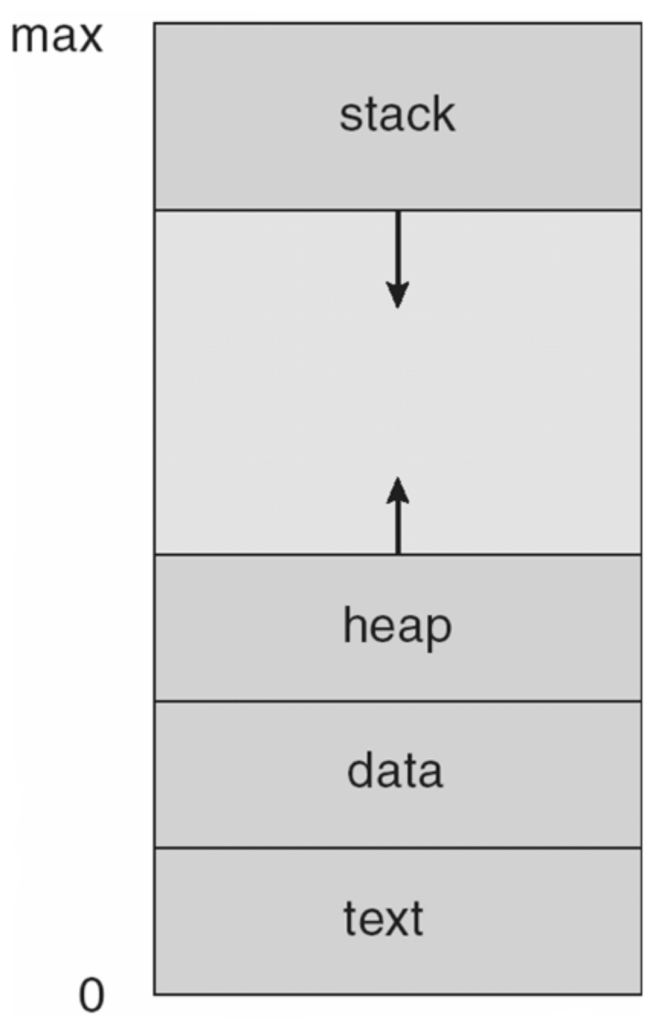
# Process Details

- Much of the functionality of a modern OS is the work required to manage processes
- OS may have hundreds of processes active at the same time
  - Although only a small number of them executing at a given time on a multi-core CPU system
- Processes are not found in the operating system *kernel*

# What is not a Process?

- A program by itself is not a process
- There is no one-to-one correspondence between programs and processes
  - E.g. there may be 10 people using emacs at the same time, i.e. 10 processes running emacs, but only one copy of the emacs program on disk
  - E.g. there may be many programs on disk that are not executing at the present time → these are not processes!
  - Programs are passive entities, while processes are active

# A Process in Memory




- **Text**: the instructions that make up the program
- **Data**: the data the program uses
- **Heap**: used for dynamic memory
- **Stack**: used for function calls