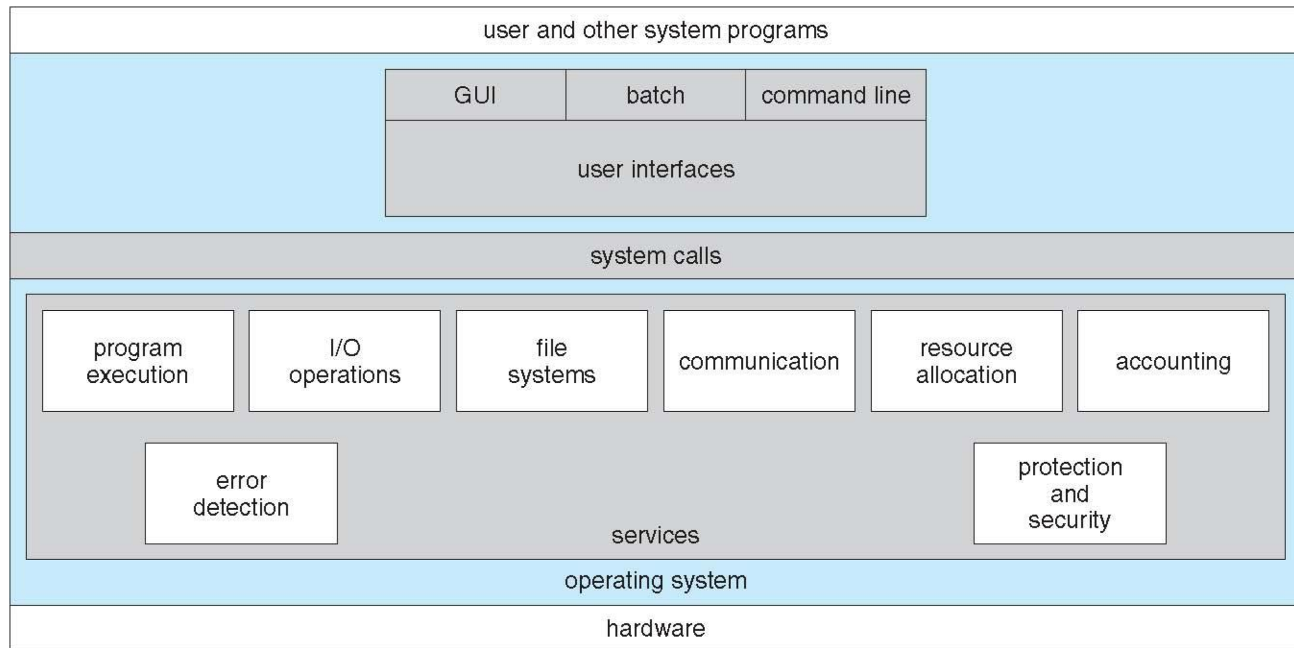
Services

Interface

Structure

# Operating system services (1)



# Operating system services (2)

- Functions that are helpful to the user
  - User interface
    - Command line interpreter
    - Batch interface
    - Graphical user interface (GUI)
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communications
    - Shared memory or message passing
  - Error detection
    - Hardware and software errors

# Operating system services (3)

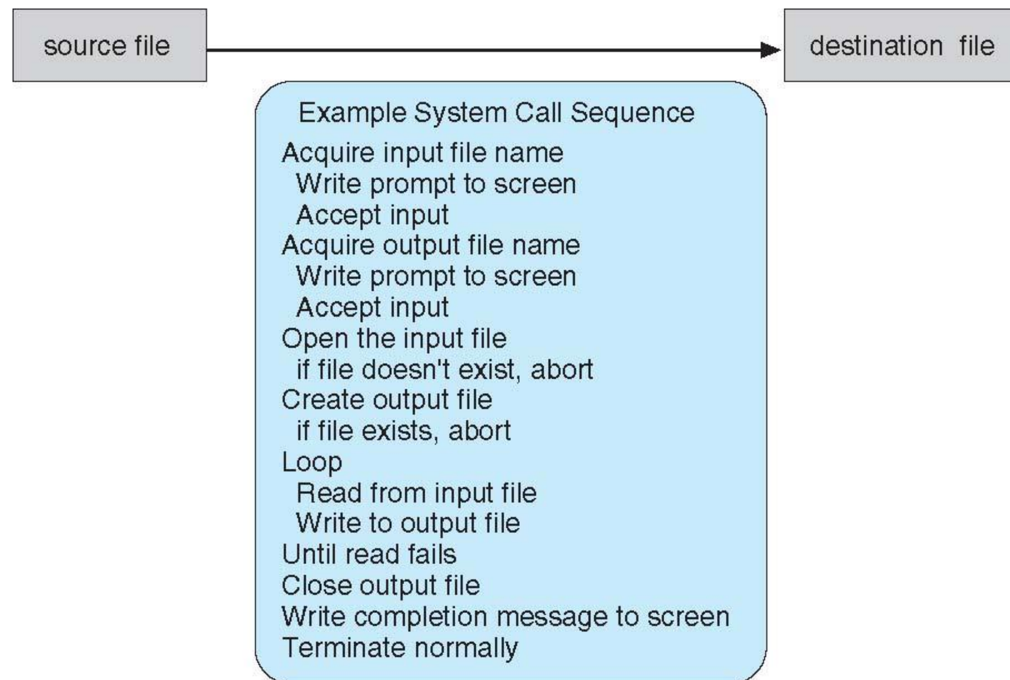
- Functions for the efficient operation of the system
  - Resource allocation
    - Special or generic allocation code
  - Accounting
  - Protection and security
    - Access control
    - Security based on weakest link principle

# OS user interface

- Command interpreter – shells
  - Implemented in kernel or as a system program
  - Built-in commands versus external system programs
- GUI
  - Desktop metaphor with icons
- Many systems include both!

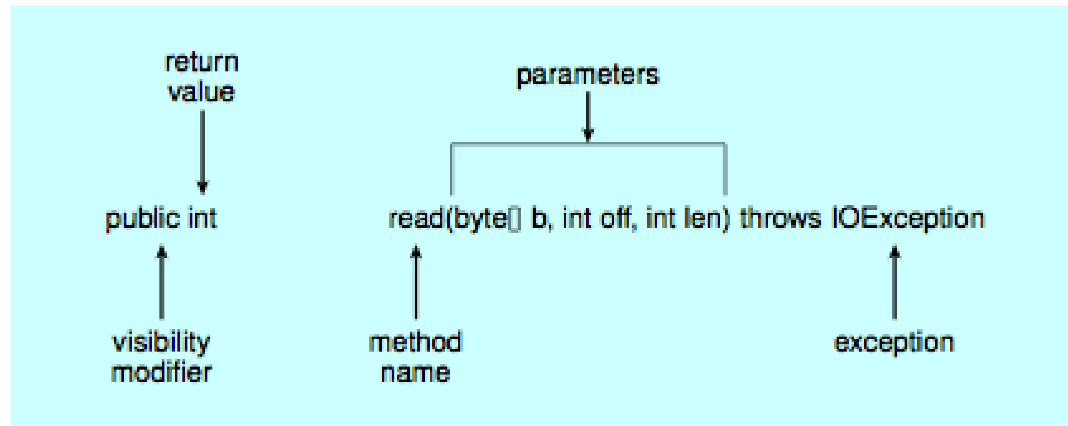
# System calls (1)

- An interface to the services of the OS
  - Routines (methods) in C or C++
- System call use: copy file



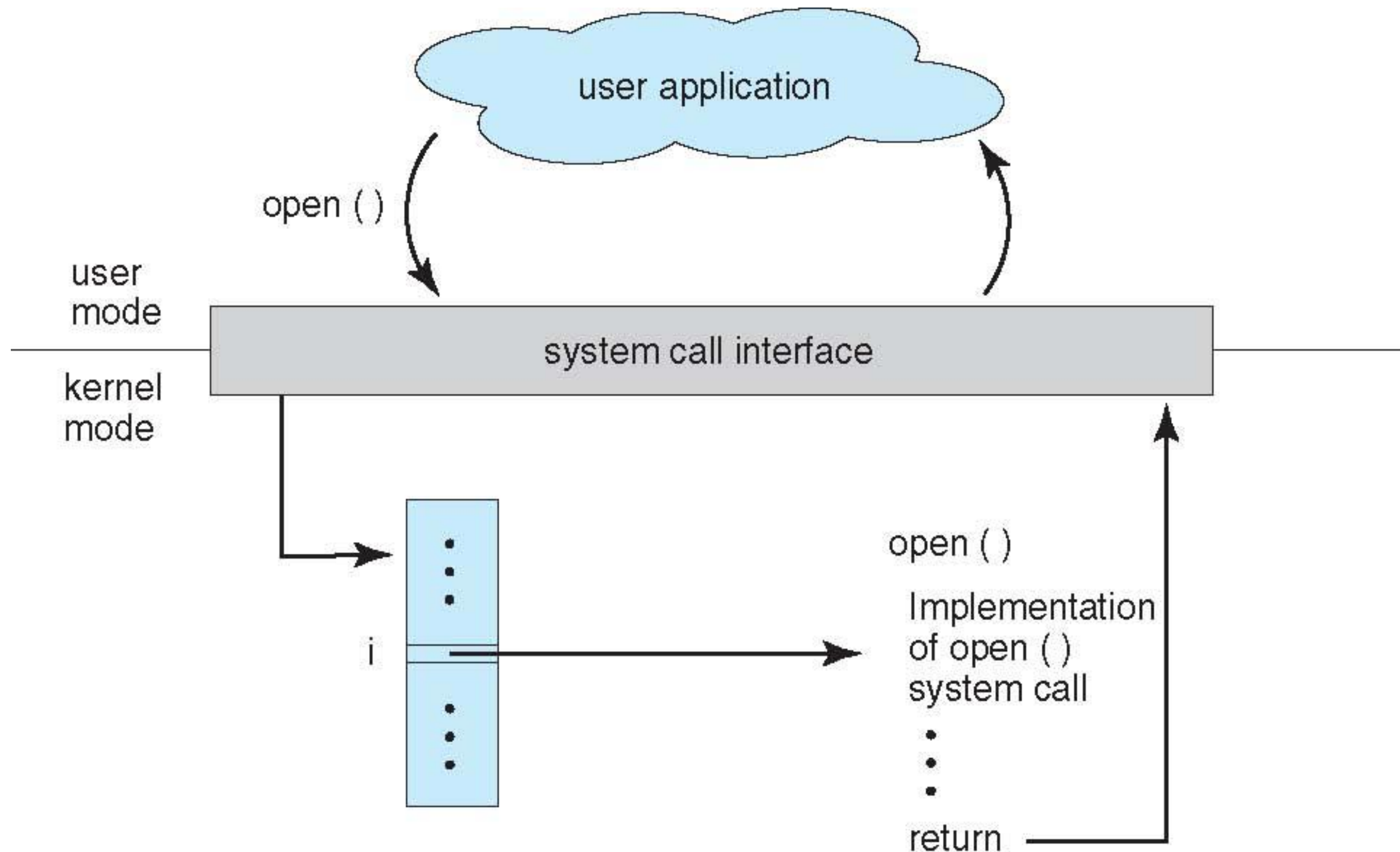
# System calls (2)

- Application Programming Interface (API)
  - Examples
    - Win32 for Windows
    - POSIX API for POSIX-based systems (Unix, Linux, Mac OS X)
    - Java API for Java Virtual Machine
  - Key advantages: portability and ease of use



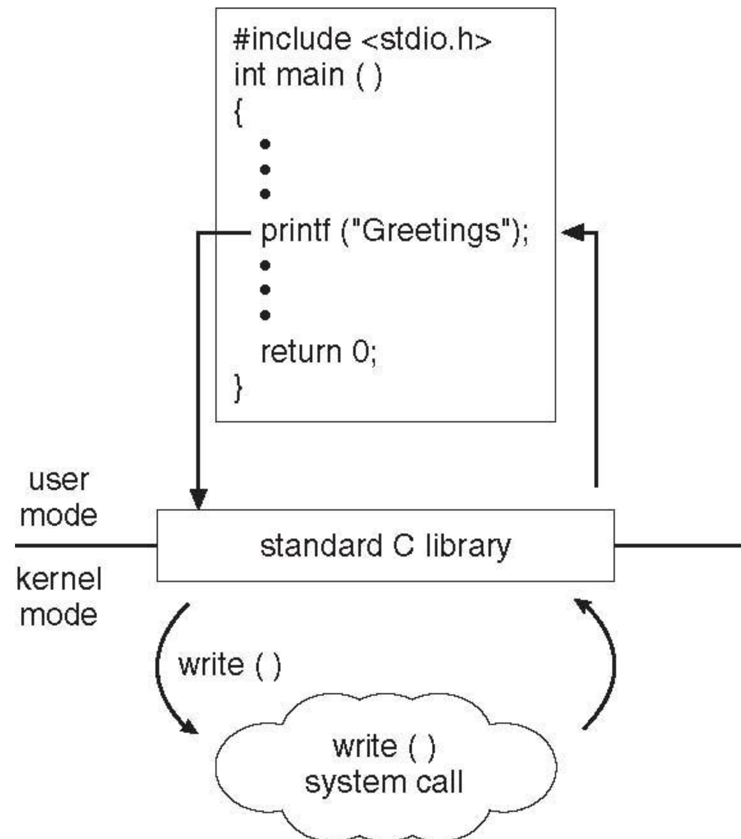
# System calls (3)

- System call handling: open()



# System calls (4)

- C program invoking printf() library call, which calls write() system call



# System calls (5)

- Parameter passing
  - Simplest: pass the parameters in registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - E.g. 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
- In Java it is not possible to directly make systems calls
  - Java Native Interface (JNI)

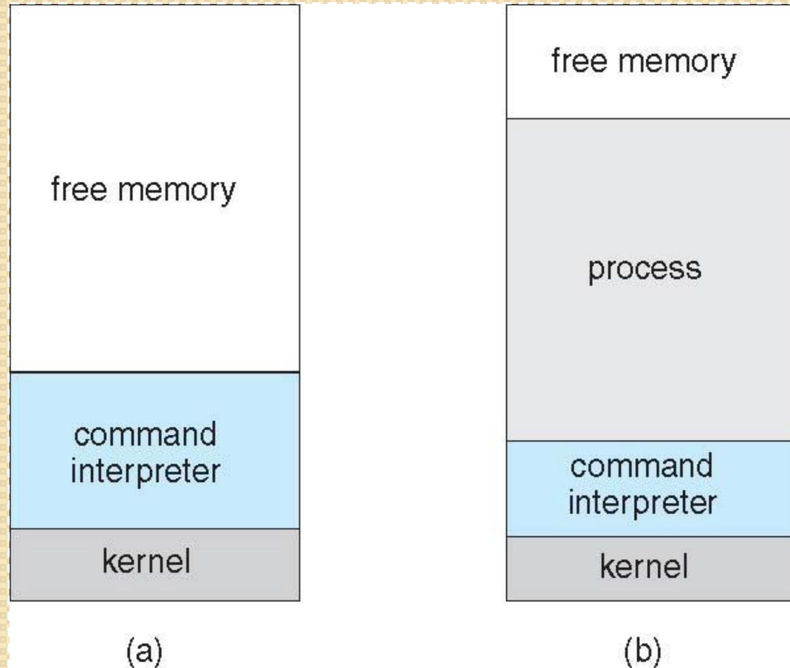
# Types of system calls (1)

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

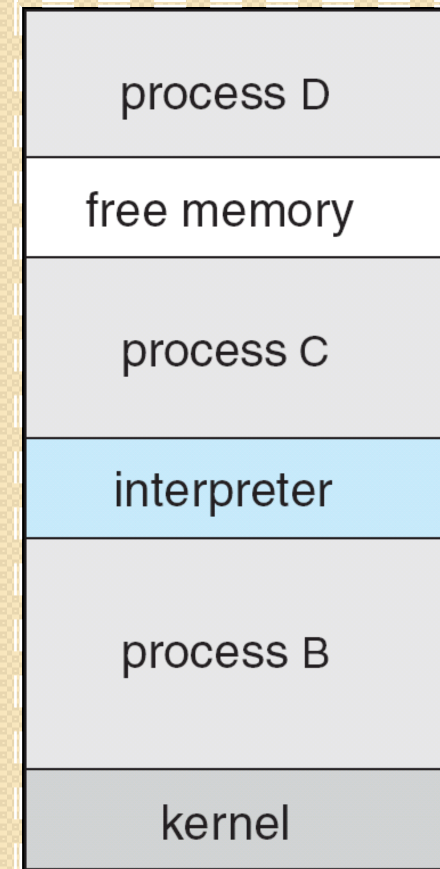
| | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File Manipulation | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device Manipulation | SetConsoleMode()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

# Types of system calls (2)

Single-task OS – MS-DOS




Multi-task OS – FreeBSD



# Types of system calls (3)

- Process control
  - end, abort
  - load, execute
  - create/terminate process
  - get/set process attributes
  - wait for time/event, signal event
  - allocate free memory
- File management
  - create/delete file
  - open, close
  - read, write, reposition
  - get/set file attributes
- Device management
  - request/release device
  - read, write, reposition
  - get/set device attributes
  - logically attach/detach devices
- Information maintenance
  - get/set time or date
  - get/set system data
  - set process, file or device attributes
- Communications
  - create/delete communication connection
  - send/receive messages
  - transfer status information
  - attach/detach remote devices
- Protection
  - get/set permission
  - allow/deny user

# System programs

- A convenient environment for program development and execution
  - File management
  - Status information (Registry)
  - File modification
  - Programming-language support
    - Compilers, assemblers, debuggers, interpreters
  - Program loading and execution (Loaders, linkage editors)
  - Communications
    - Web browsers, Email clients, Remote log in, File transfer
- Application programs
- These define the users' view of the OS

# OS Design & Implementation (1)

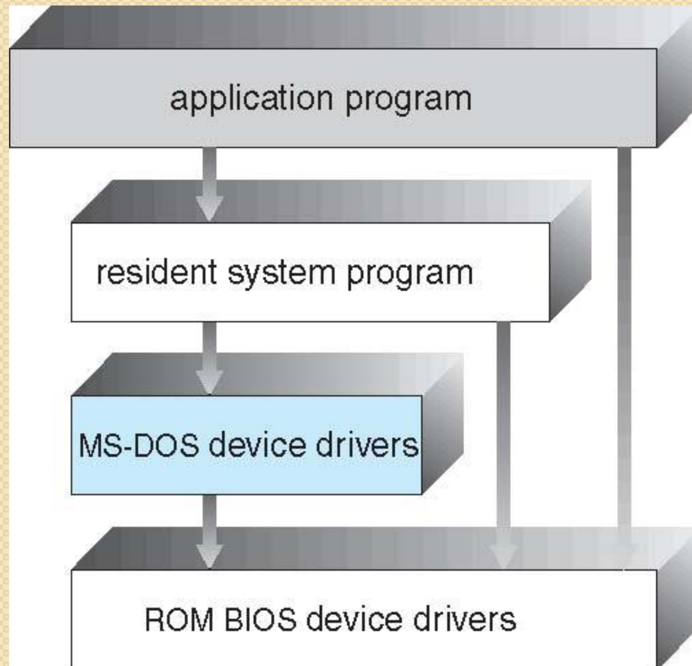
- No definitive answers
- Design goals
  - User goals: convenient to use, easy to learn, reliable, safe and fast
  - System goals: easy to design, implement, maintain, flexible, reliable, error free and efficient
  - General software engineering principles help
- Separation of policy from mechanism
  - Separate how (policy) from what (policy)
  - Example: timer
  - Separation is important for flexibility
  - Policy decisions are important for all resource allocation

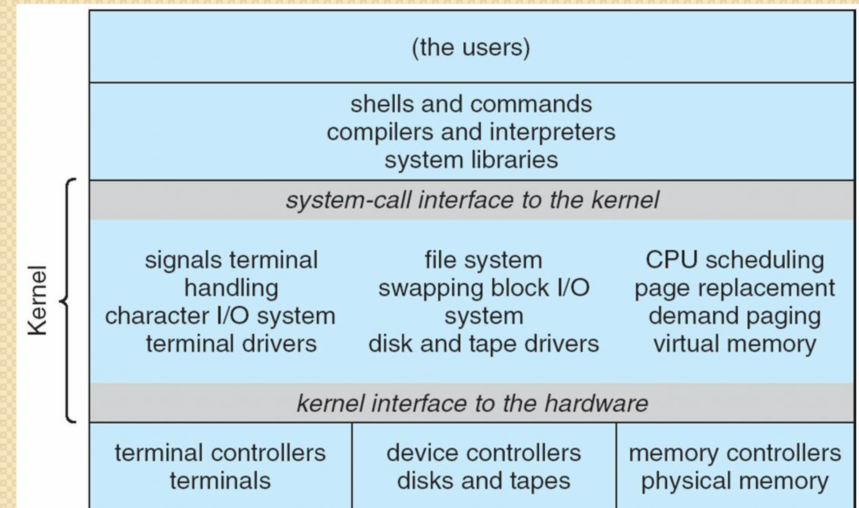
# OS Design & Implementation (2)

- Implementation
  - Mostly written in C or C++ with certain parts in assembly language
    - Advantages
      - Faster coding, more compact, easier to understand and debug
      - Improvements in compilation will improve the code
      - Easier to port
    - Disadvantages
      - Reduced speed, increased storage requirements
        - Modern compiler are better than most developers at code optimisation
        - Better data structure and algorithms are more likely to deliver performance improvements
        - You can always identify and fix performance bottlenecks after correctness has been established

MS-DOS layers

Traditional Unix

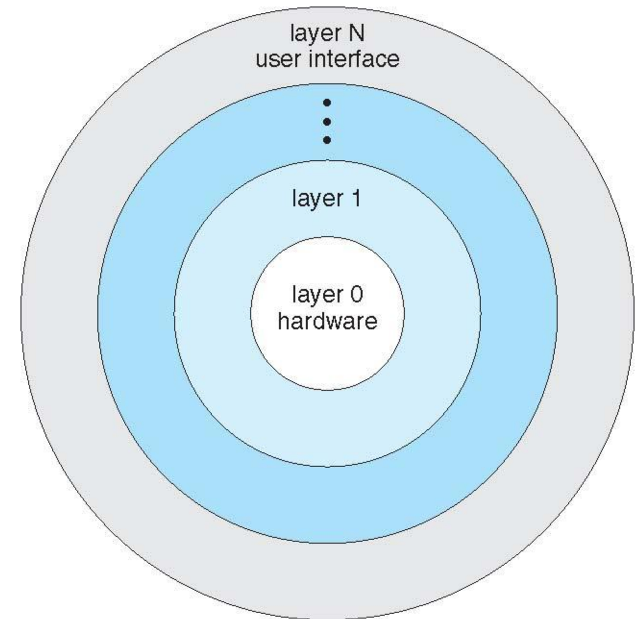



# OS Structure (1)

# OS Structure (2)

- Layered approach
  - Top-down approach & information hiding
  - Open versus closed
  - Advantages
    - Simplicity of construction and debugging
  - Disadvantages
    - Difficulty in defining layers
    - Less efficient
  - Trend: fewer layers with more functionality

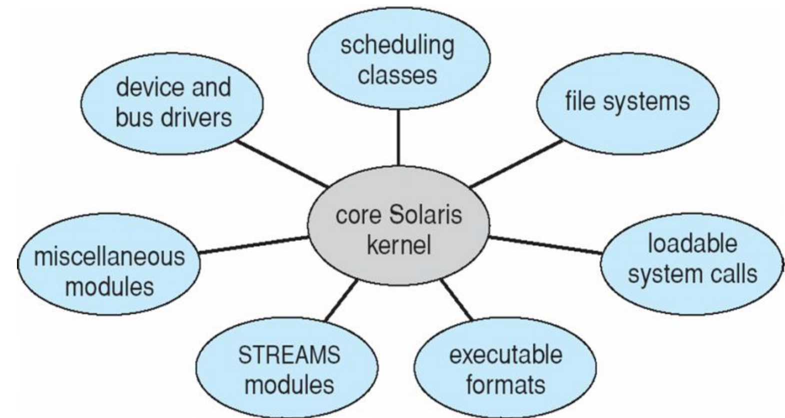


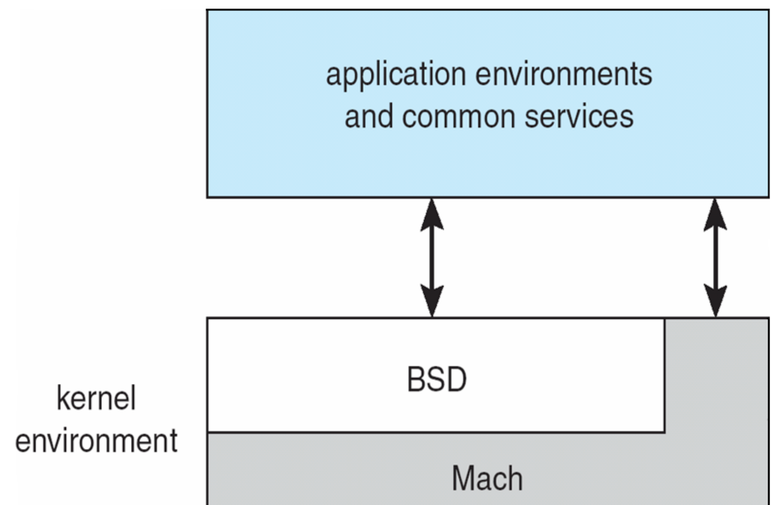
# OS Structure (3)

- Micro-kernels
  - Remove everything non-essential out of the kernel
  - Module communication with message-passing through the kernel
  - Advantages
    - Ease of extension, easier to port, more secure and reliable
  - Disadvantage
    - Performance decrease

# OS Structure (4)

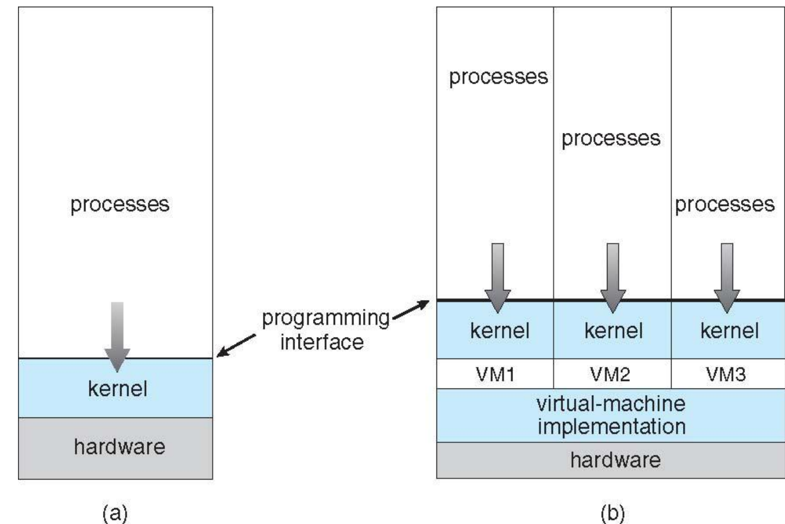
- Modules
  - Best current approach: use object-oriented techniques to create a modular kernel
    - Module interfaces
  - Dynamically loadable modules
  - More flexible than layered
  - More efficient than micro-kernel



# Virtual machines (I)

- The logical conclusion of the layered approach
- Abstract single computer hardware into several execution different environments
    - CPU scheduling and virtual memory techniques give the illusion of a processor and memory for its environment
- Host versus guest OS

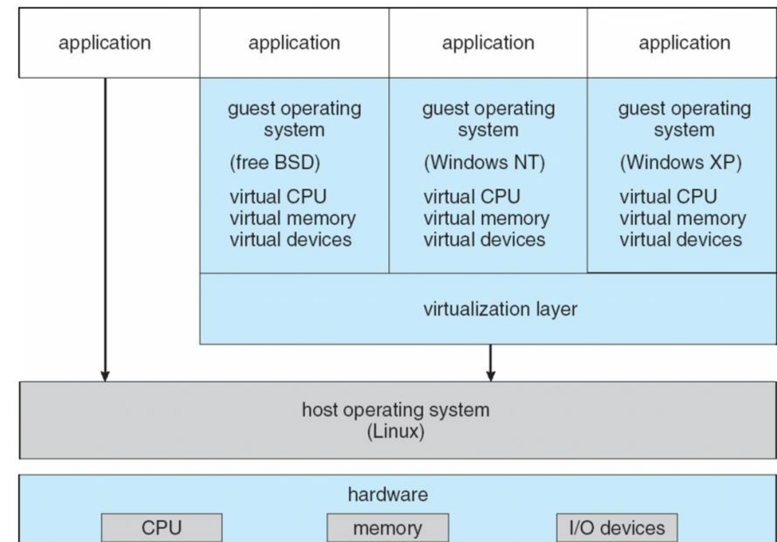


# Virtual machines (2)

- Benefits
  - Share hardware between different execution environments
  - Protect host from VMs and isolate VMs from each other
    - Sharing either through shared file-system or over virtual communication network
  - A perfect vehicle for OS research and development
    - System development time reduction
  - Rapid porting and testing of programs in multiple environments
  - Key for Cloud Computing
    - System consolidation
    - Easier system management
      - VM images – Open Virtual Machine Format

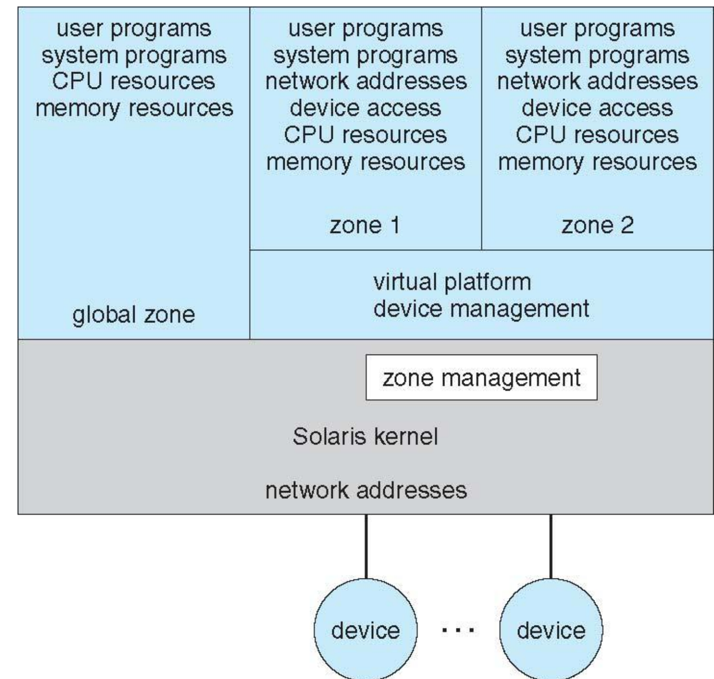
# Virtual machines (3)

- Implementation
  - Challenging to provide an exact duplicate of the underlying machine
  - Virtual user and virtual kernel mode
  - Performance issues
    - Virtual I/O – spooled or interpreted
    - Multiprogramming
  - Requires hardware support



# Virtual machines (4)

- Alternatives
  - Simulation/emulation
  - Para-virtualisation
  - Container or zones

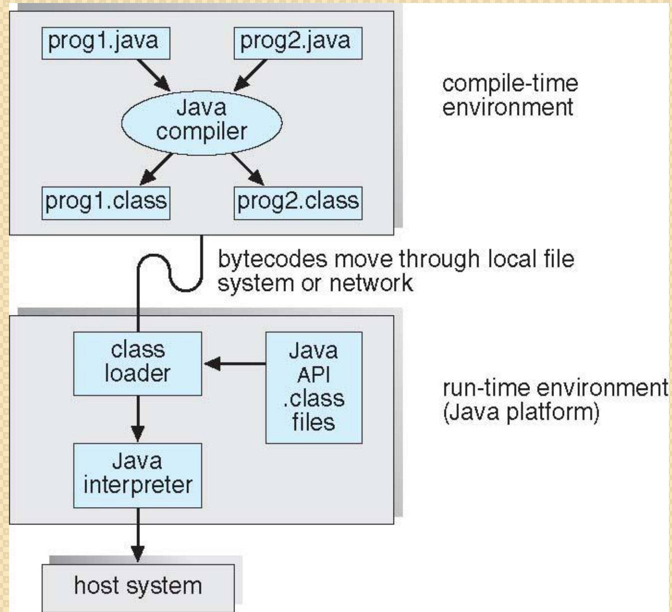


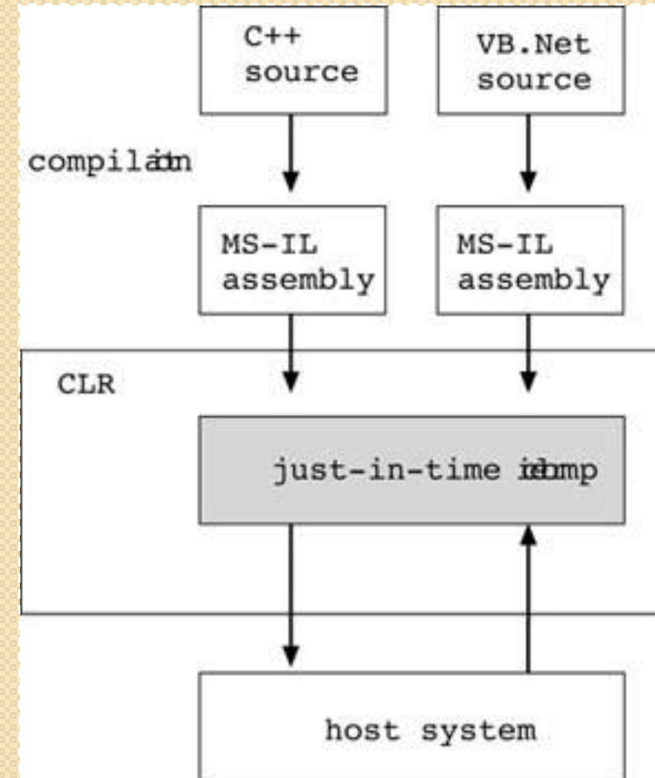
# Java (1)

- Java technology
  - Programming language specification
  - Virtual machine specification
- Java language
  - Object-oriented
  - Each class compiled into bytecode
  - High-level support for networking and distributed objects
  - Multi-threaded language
  - Secure
  - Java standard edition API (micro-edition, enterprise edition)
- Java virtual machine
  - Class loader and Java interpreter
  - Garbage collection
  - Just-in-time (JIT) compiler
  - Java programs do not interact directly with the OS, the JVM does
- Java development kit
  - Development tools (compiler, debugger)
  - Runtime environment (JRE)

JDK

.NET




# Java (2)

# For contemplation

- What is the purpose of the command interpreter? Why is it usually separate from the kernel?
- What is the main advantage of the layered approach to system design? What are the disadvantages of using the layered approach?
- How could a system be designed to allow a choice of OS from which to boot? What would the bootstrap program need to do?
- Describe three general methods for passing parameters to the operating system.
- What are the advantages and disadvantages of using the same system call interface for manipulating both files and devices?
- Why is the separation of mechanism and policy desirable?
- What are the main advantages of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?
- In what ways is the modular kernel approach similar to the layered approach? In what ways does it differ from the layered approach?
- What is the main advantage for an OS designer of using a virtual machine architecture? What is the main advantage for a user?
- What is the relationship between a guest operating system and a host operating system in a system like VMware? What factors need to be considered in choosing the host operating system?
- The experimental Synthesis operating system has an assembler incorporated within the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and to system-performance optimization.