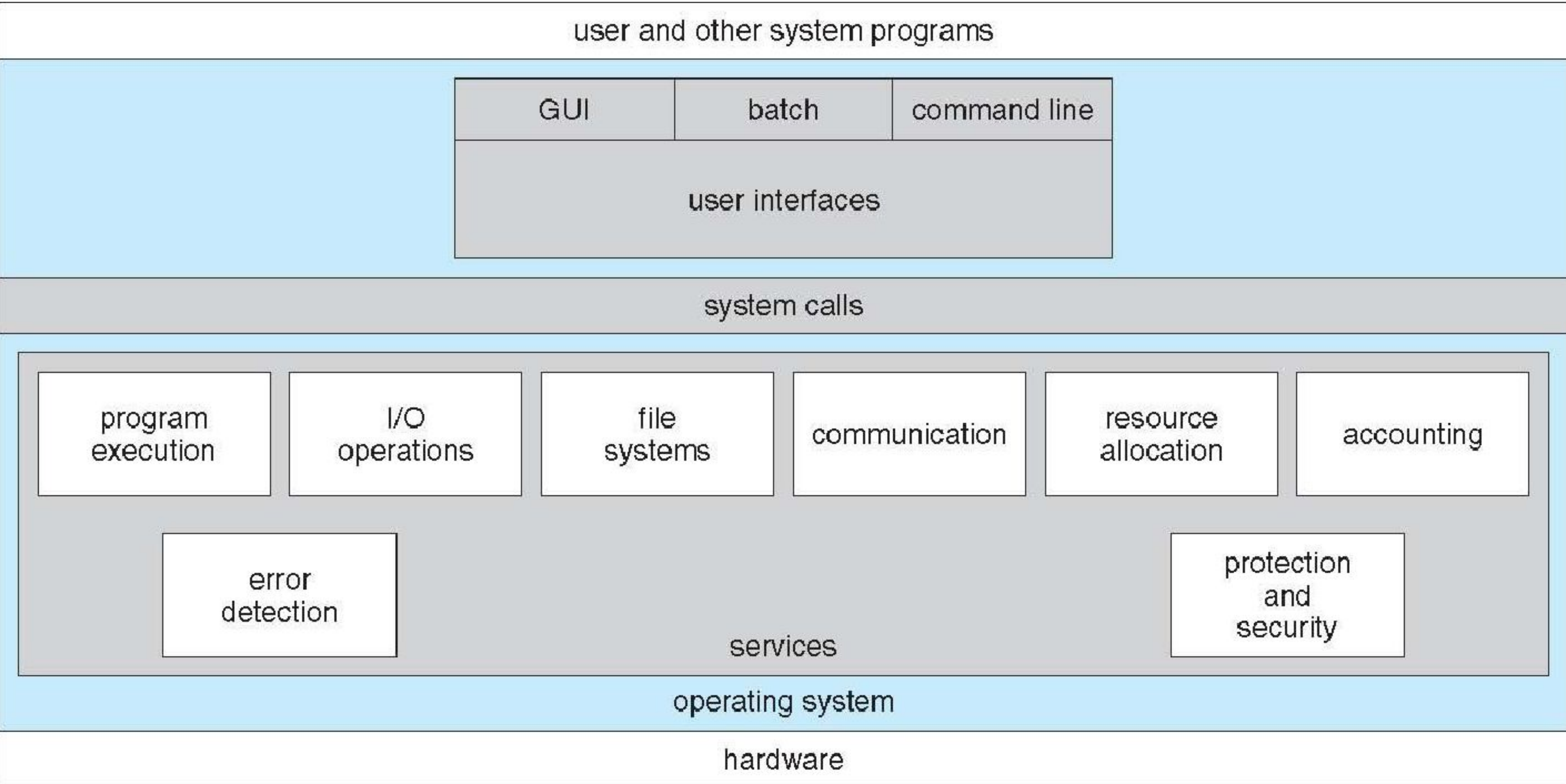
Operating System Structures

Objectives

- Describe
  - Services
    - Users
    - Processes
    - Other Systems
- Discuss Structure
- Explain
  - Installation
  - Customization
  - Booting

OS Services

- Provide environment for execution of programs and services
  - Client: user or other programs
- Efficient operation of the system
  - Resource sharing


user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

OS Services – Execution Environment

- User interface
  - Command Line Interpreter (CLI)
  - Graphical User Interface (GUI)
  - Batch
- Program execution
  - Load into memory, run, terminate
- I/O operations
- File Operation
  - Open/close

OS Services – Execution Environment

- File manipulation
  - Read/write
  - Create/delete
  - Search
  - Permissions
  - List information
- Process communication
  - Within system box or across a network
  - Message passing vs. shared memory

OS Services – Execution Environment

- Error detection
  - CPU errors
  - Memory errors
  - I/O errors
  - Program errors

OS Services – Resource Sharing

- Resource allocation
  - Multiple users
  - Multiple (concurrent processes)
  - Resources include
    - CPU cycles, memory, I/O devices, file storage
- Accounting
  - Auditing and debugging
- Protection and security

OS-UIs – Command-Line Interface (CLI)

- Text based command
- Goal
  - Fetch and execute next user command
- Implementation
  - Kernel – single program handles all commands
  - System programs – each command is a program name
    - More flexible
- Multiple flavours
  - Shells
- Shell scripts – sequence of commands in a file (batch mode)
  - frequently used sequence of commands

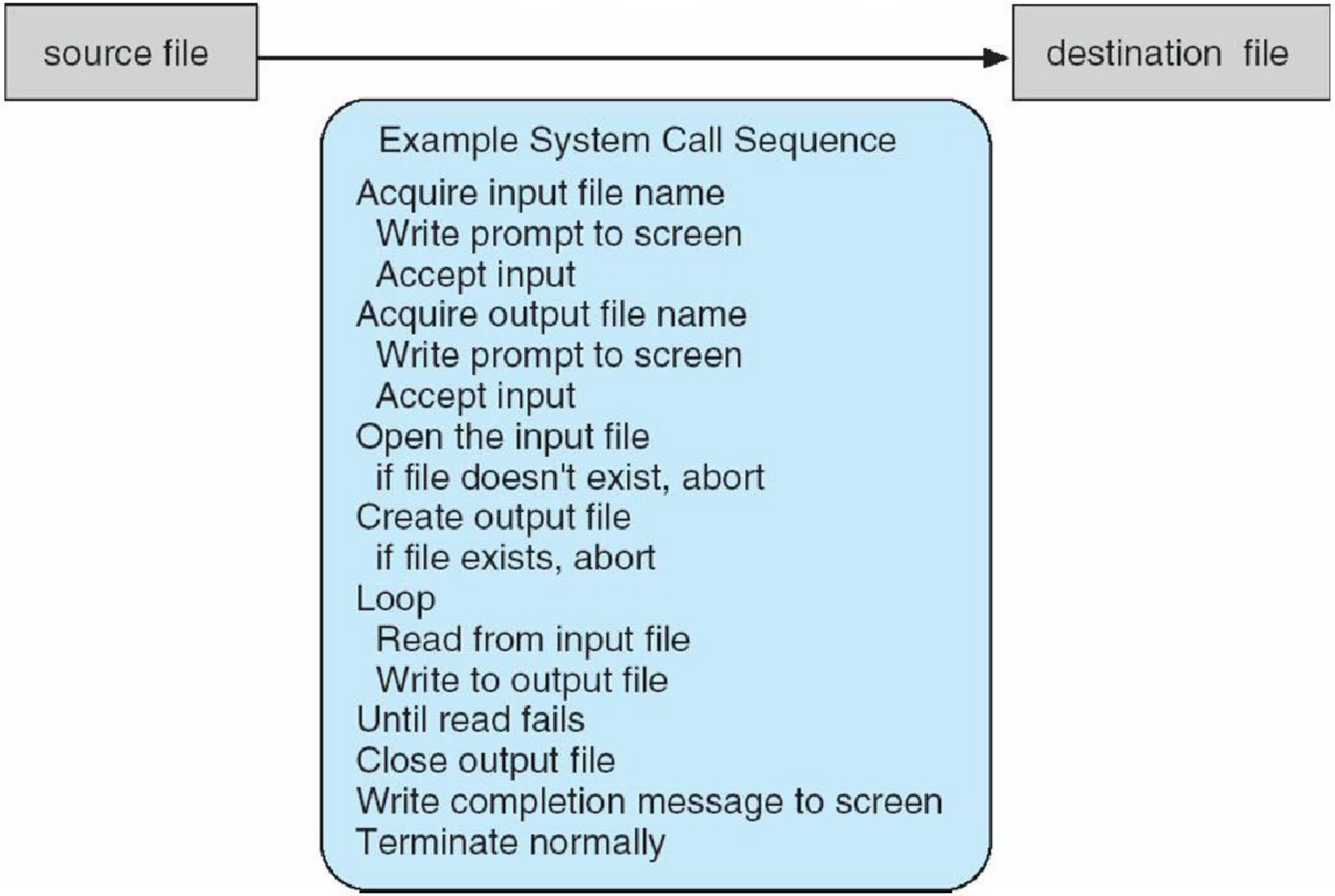
Graphical User Interface (GUI)

- Windows with desktop metaphor
    - icons, mouse pointer, menus
    - point-and-click, drag-and-drop
- Touch screen
    - Smart phones and tablets - gestures
- Hybrid systems
    - Both GUI and CLI
        - E.g. Windows 10
        - Linux (GUI optional)

System Calls

- Programming interface to services provided by the OS
  - Written in a high-level language (typically C/C++)
- Application Program Interface (API)
  - Indirect use of system calls
- Examples
  - Win32 API
  - POSIX API (UNIX, Linux, Mac OS X)
  - JAVA API (JVM)
- Why not make system calls directly?

System Call Example



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)
```

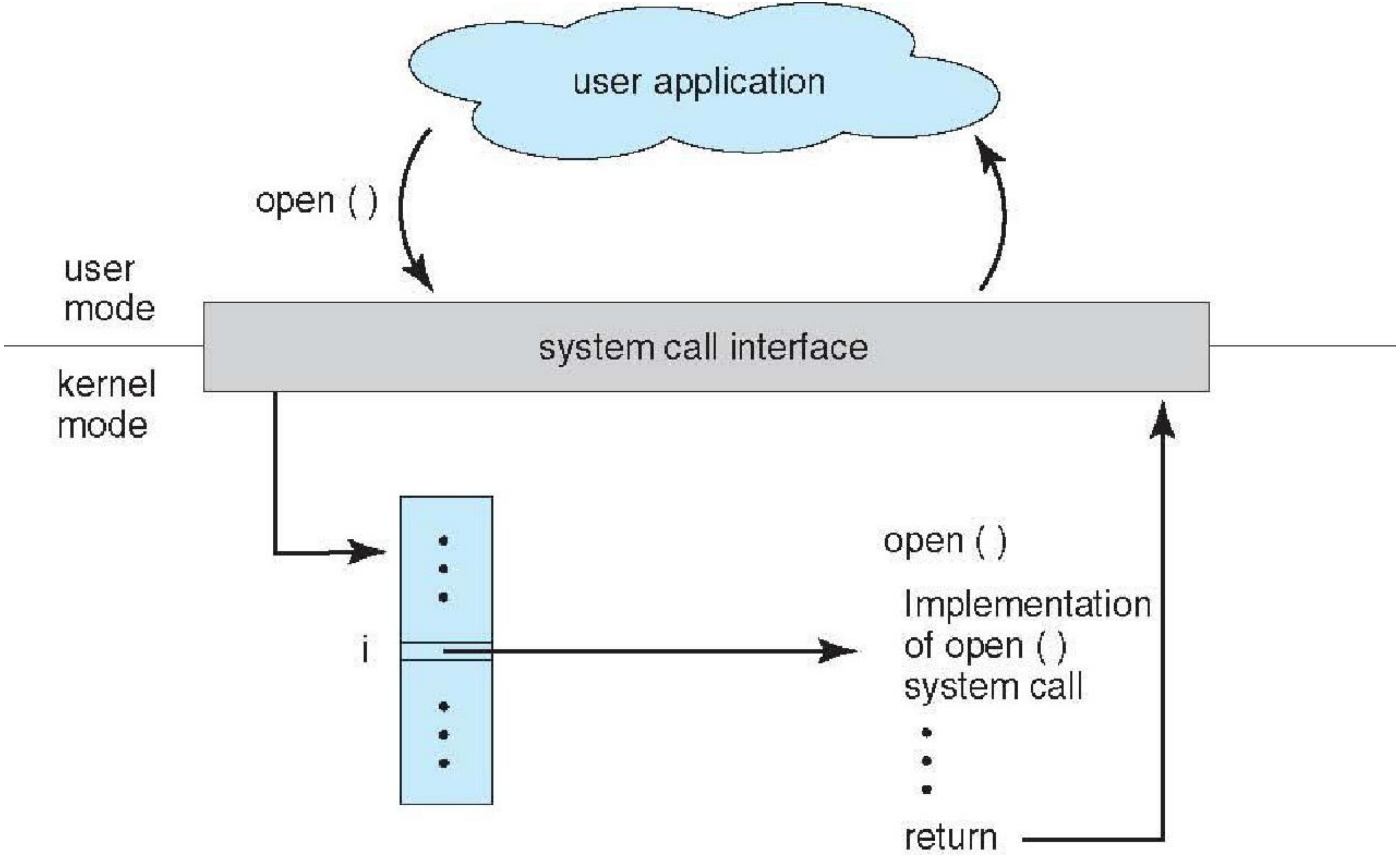
return value | function name | parameters

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.

API – System Call – OS Relationship

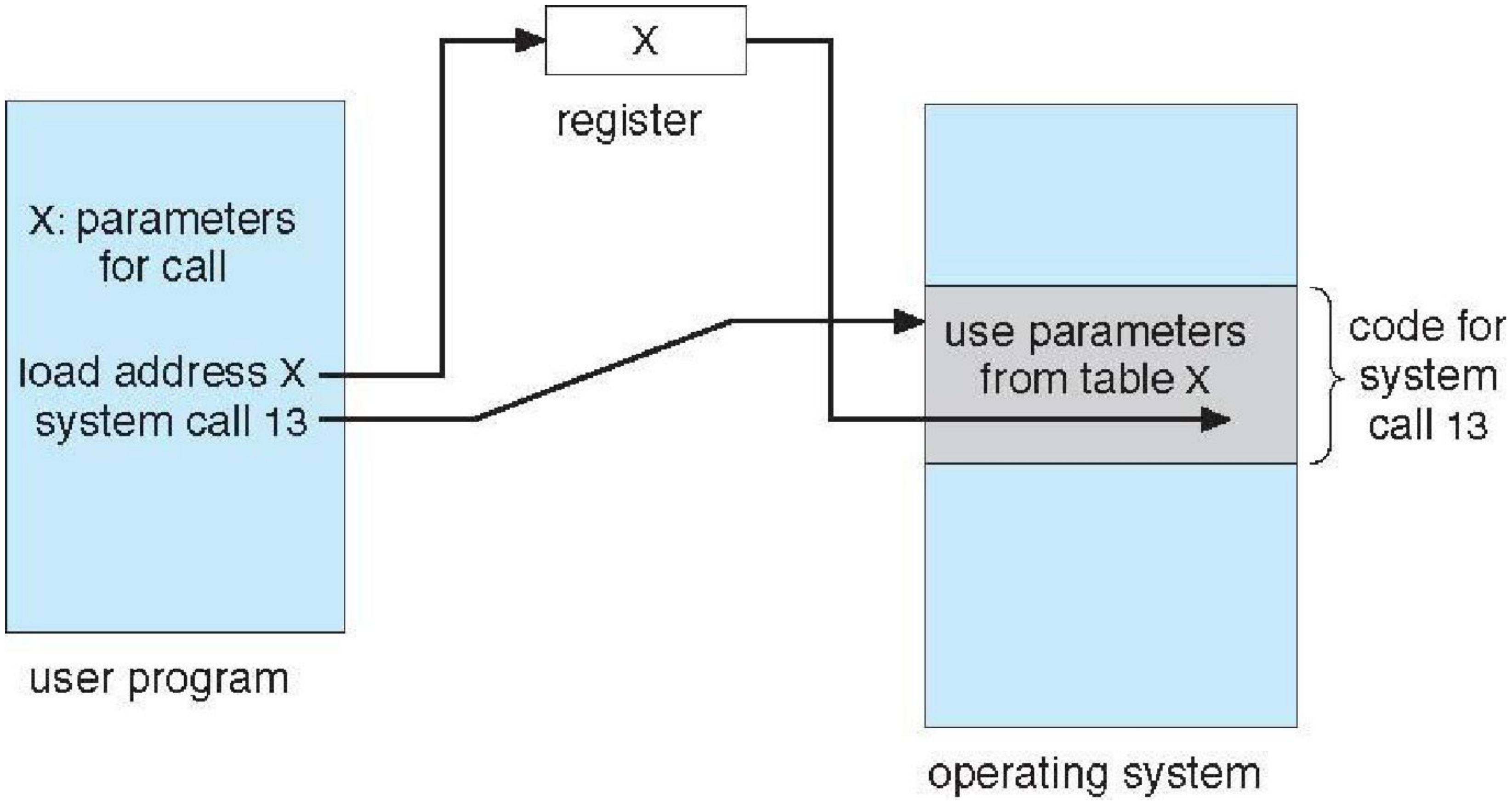


System Call Parameter Passing

- Three methods
  - Registers – store parameters into registers
  - Memory block
    - Address in register
  - Stack
    - Program pushes
    - OS pops

Large parameters

Parameter Passing Using 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
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls

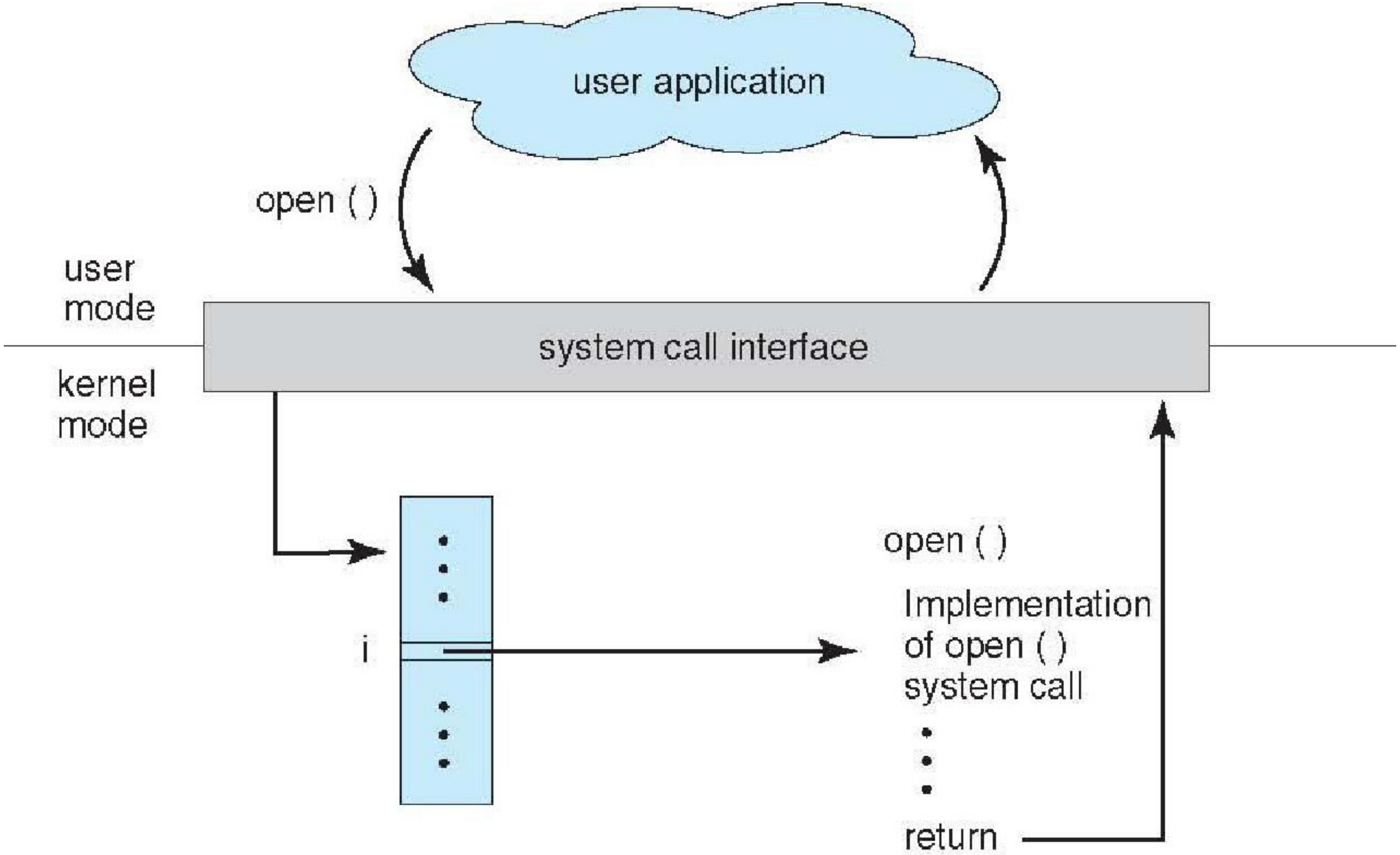
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
    - **Message passing model**
  - create and gain access to memory regions
    - **Shared-memory model**
  - transfer status information
  - attach and detach remote devices

Types of System Calls

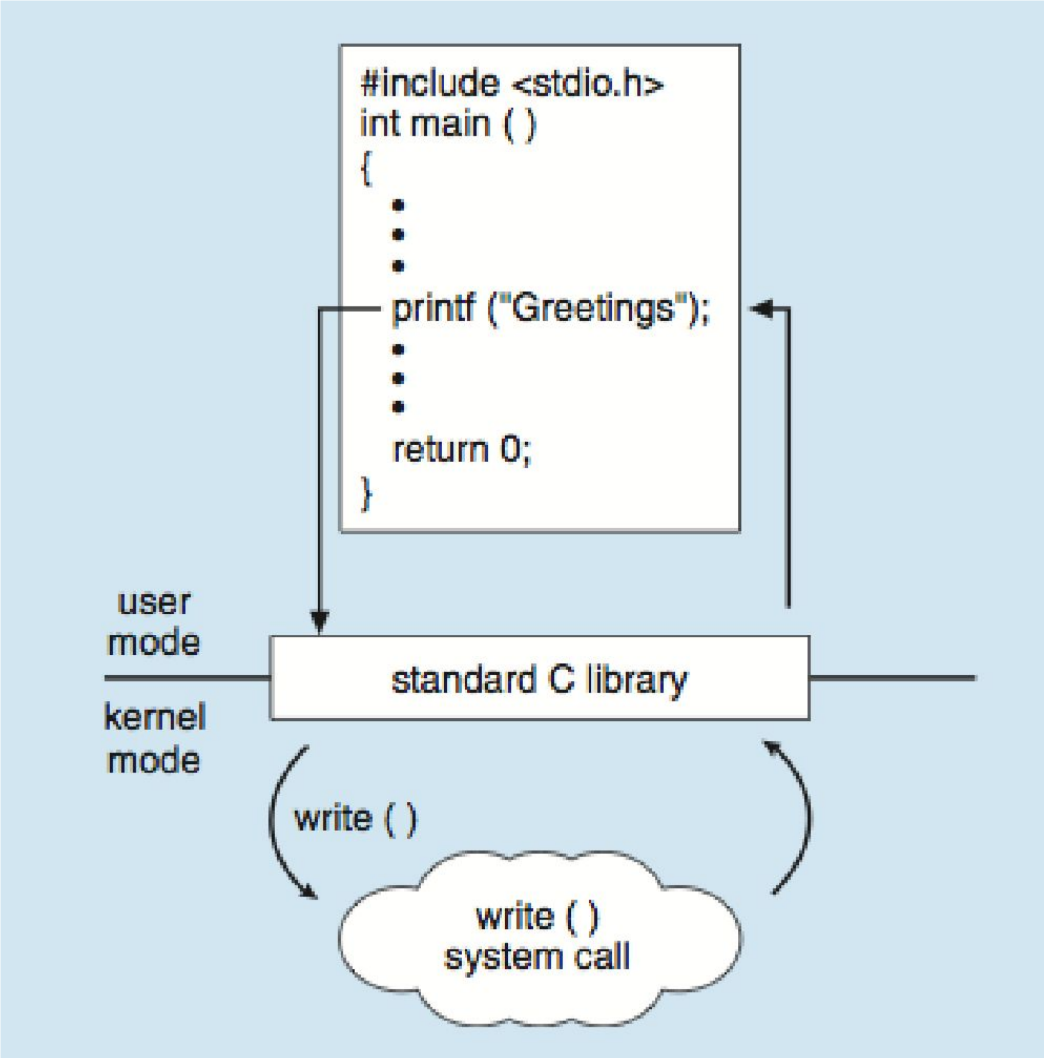
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

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

API – System Call – OS Relationship



Standard C Library Example



System Programs

- Convenient environment
  - Program development
  - Program execution
- Most users’ view of the OS
- Can have system programs for
  - File manipulation
  - Status information
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

OS Design and Implementation

- Complex problem
  - No “one size fits all” solution
- Design goals
  - User goals
    - Convenient, easy to learn, reliable, safe and fast
  - System goals
    - Easy to design, implement and maintain
    - Flexible
    - Reliable
    - Error-free
    - efficient

OS Design

- Separation of concerns
  - Policy
    - What to do?
  - Mechanism
    - How to do it?
- Separation facilitates flexibility if policy changes

OS Implementation

- Possibilities
  - Assembly language
    - Early OSes
  - System programming languages
    - E.g. Algol, PL/1
  - C/C++
    - State of the art
- Usually a mixture of language
  - Assembly – for lowest level operations
  - C – form for main body
  - C/C++, PERL, Python, shell script for system programs

OS Implementation

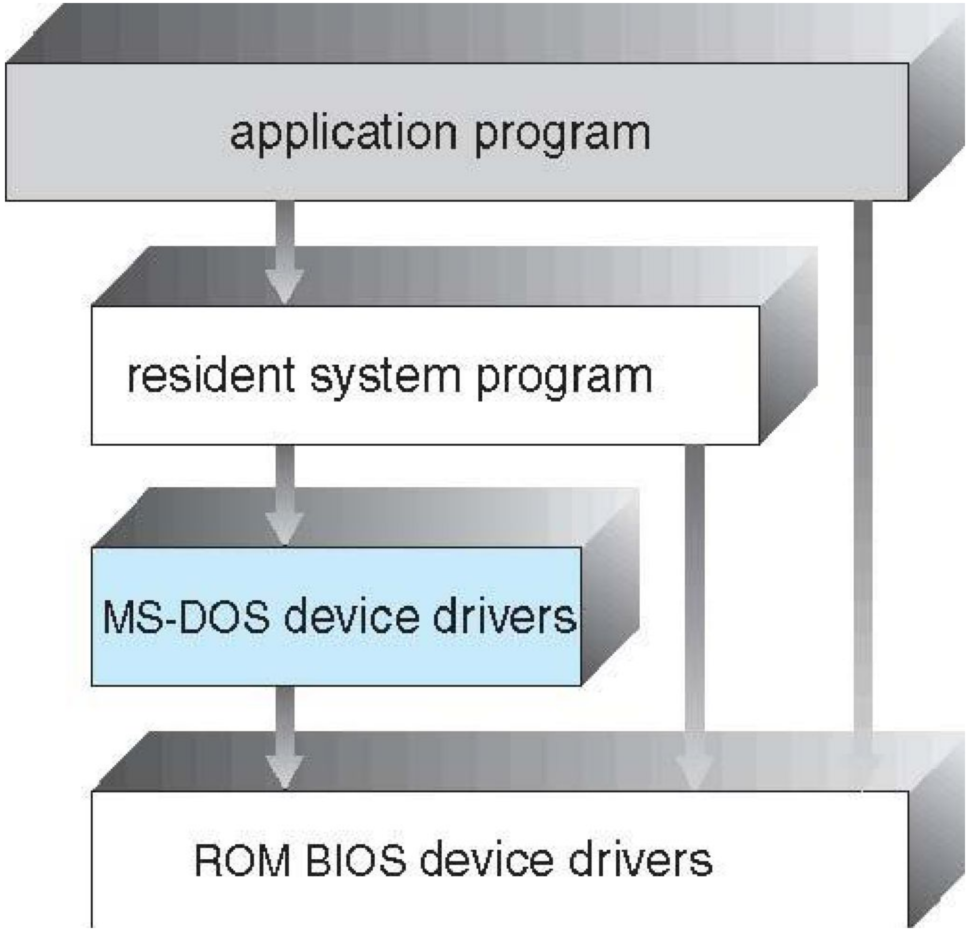
- High-level language implementation
  - Easy to port to other hardware
  - But, slower
- Emulation
  - Enables running an OS on non-native hardware
  - Again, slower

OS Structure

- General purpose OS is a very large program
- Various ways to structure
  - Examples in next slides

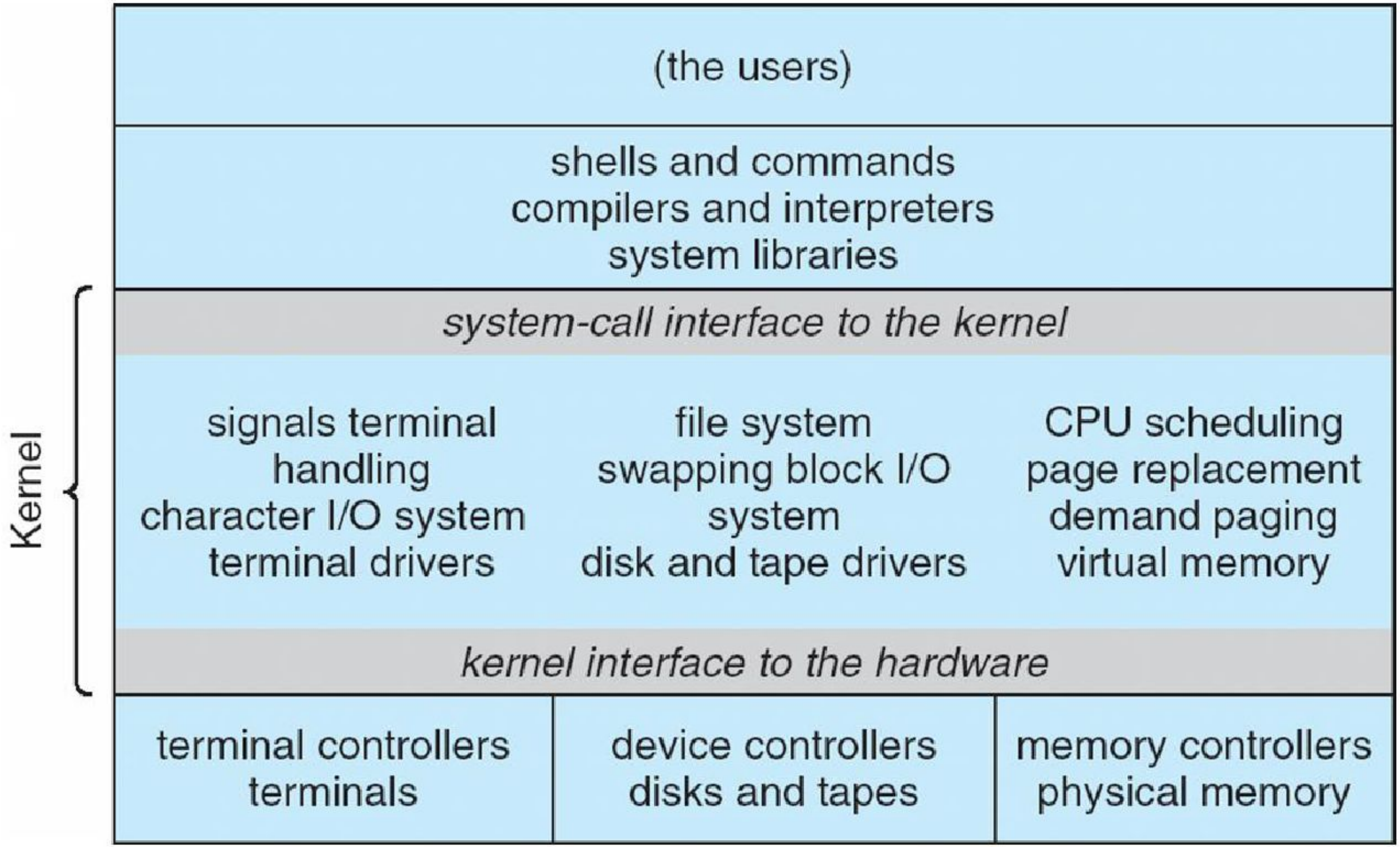
OS Structure – Simple Structure

- Eg. MS-DOS

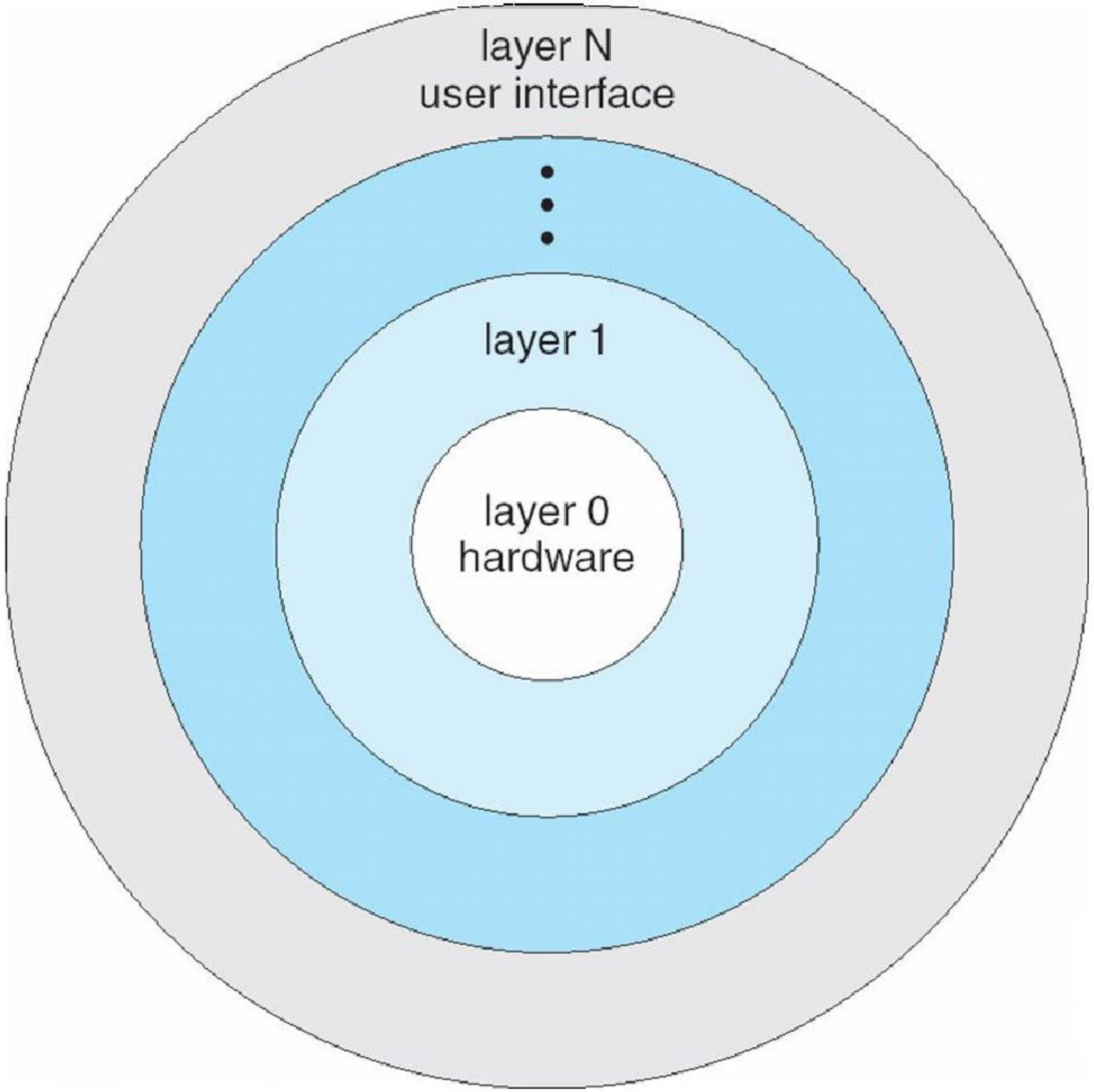


OS Structure - UNIX

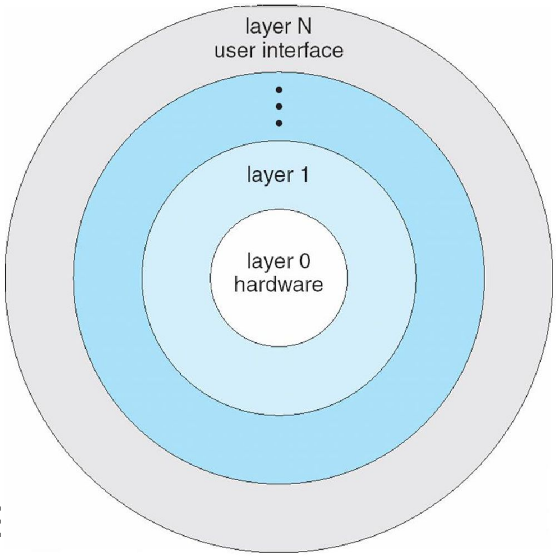
- Two separate parts
  - Kernel
  - System programs


(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal handling
character I/O system
terminal drivers
file system
swapping block I/O system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory
Kernel

OS Structure – Layered Approach

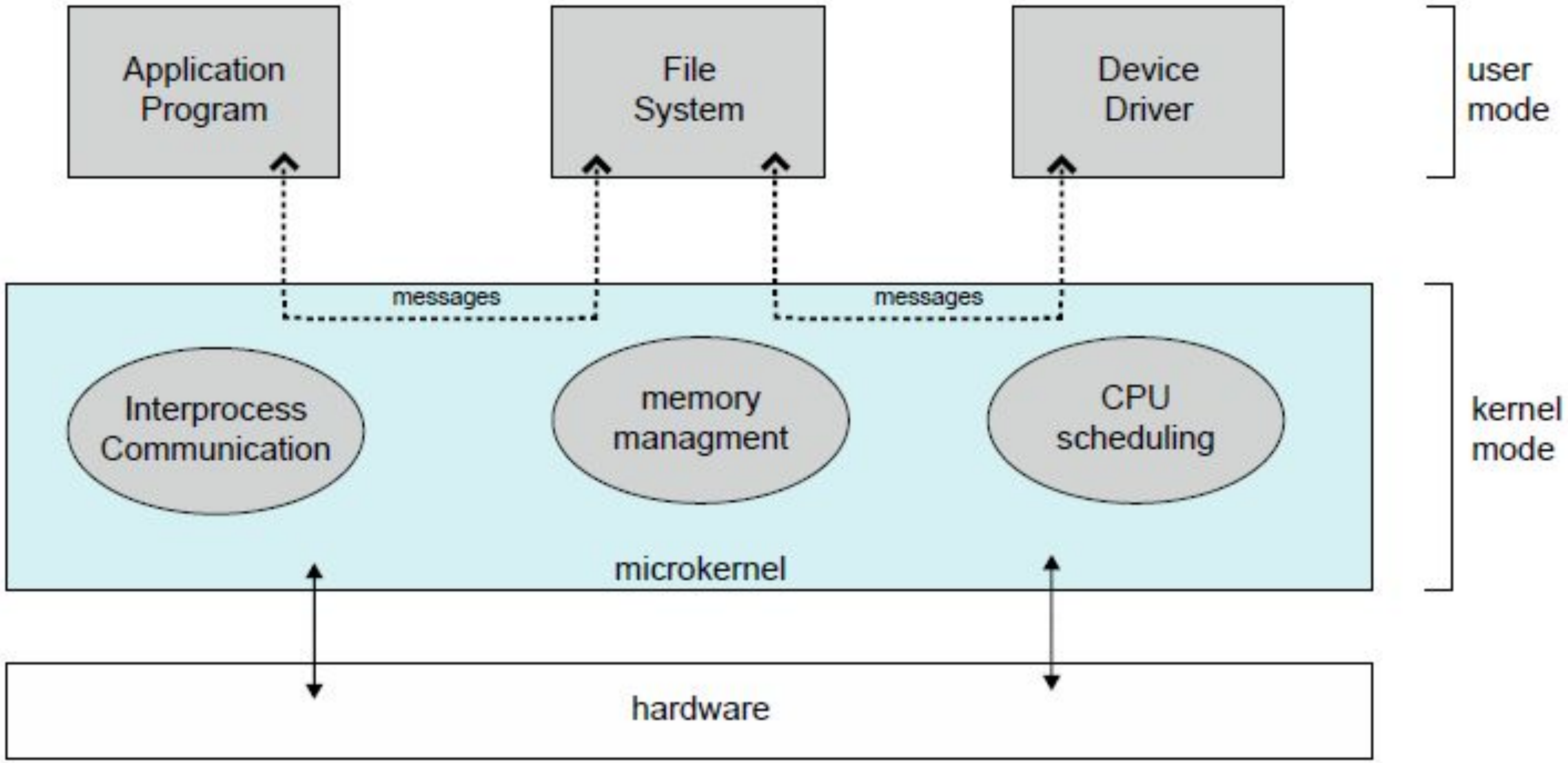


OS Structure – Layered Approach



- Lower layer provides service to upper layer
- Design of layer is not a trivial exercise
  - e.g. CPU scheduling layer vs backing store device driver la
    - CPU scheduling below backing store device driver
    - CPU scheduling above backing store device driver allows CPU to use swapping when memory is not large enough
- Weakness – function call overhead

OS Structure - Microkernel



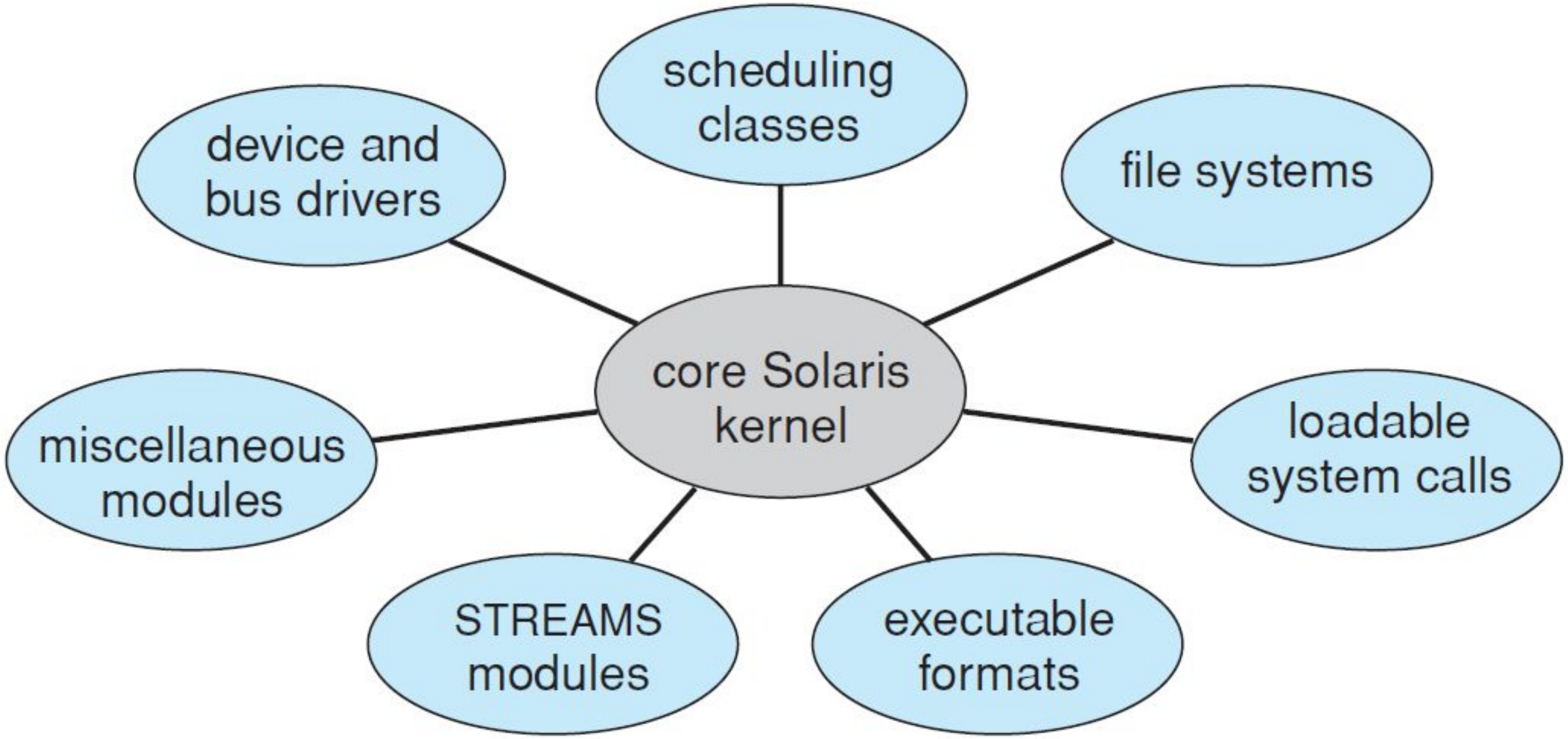
OS Structure - Microkernel

- Advantages
  - Easier to extend
  - Enhanced portability
  - Increases reliability
  - More secure
- Disadvantage
  - Increased performance overhead
    - Increased communication between user space and kernel

OS Structure – Loadable Kernel Modules

- Loadable kernel modules
- Object-oriented principles
- Core component
  - Is separate
  - Talks to others through an interface
  - Is loadable as needed within the kernel
- E.g.
  - Linux, Solaris

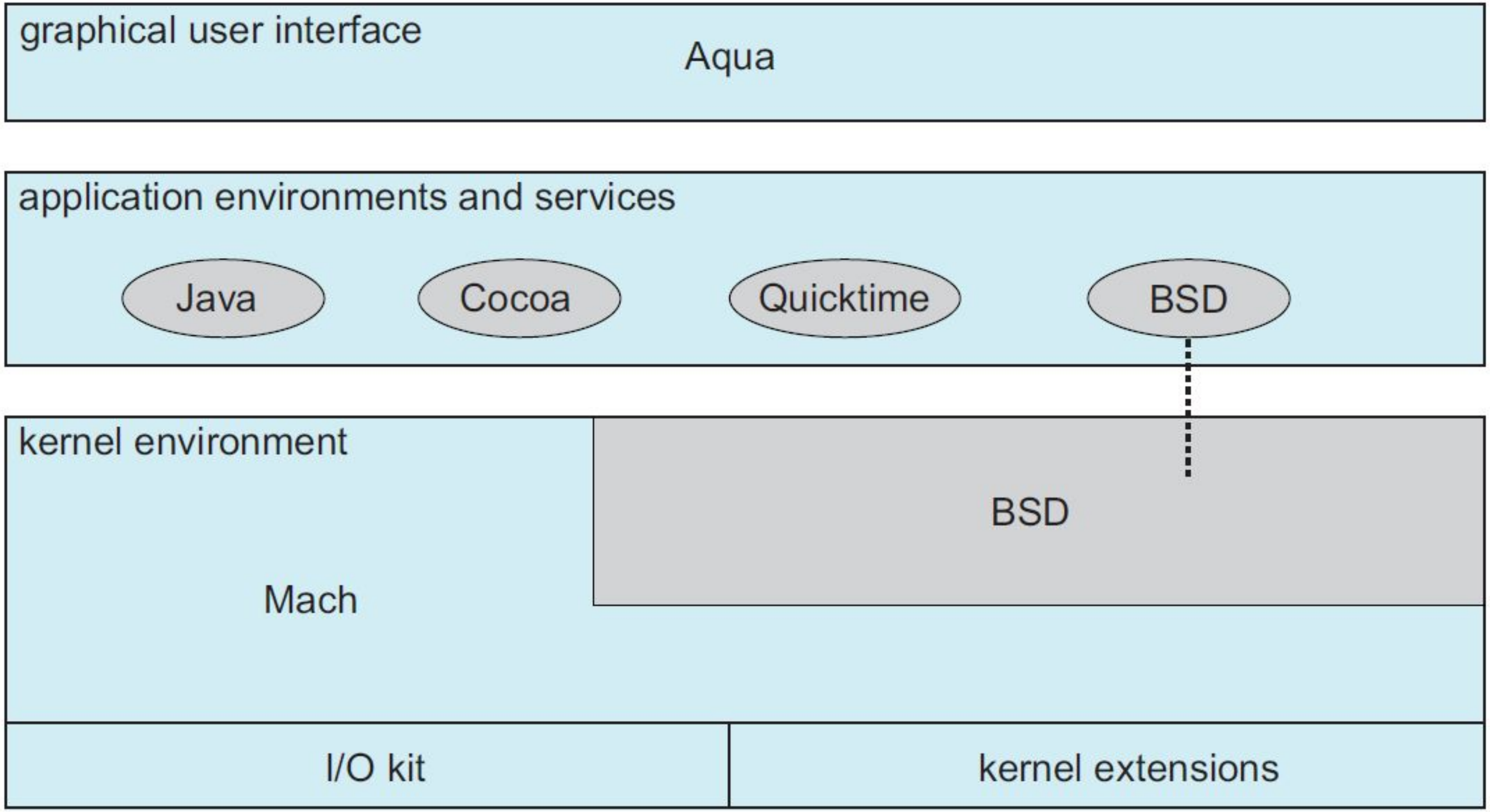
Solaris architecture – loadable kernel modules



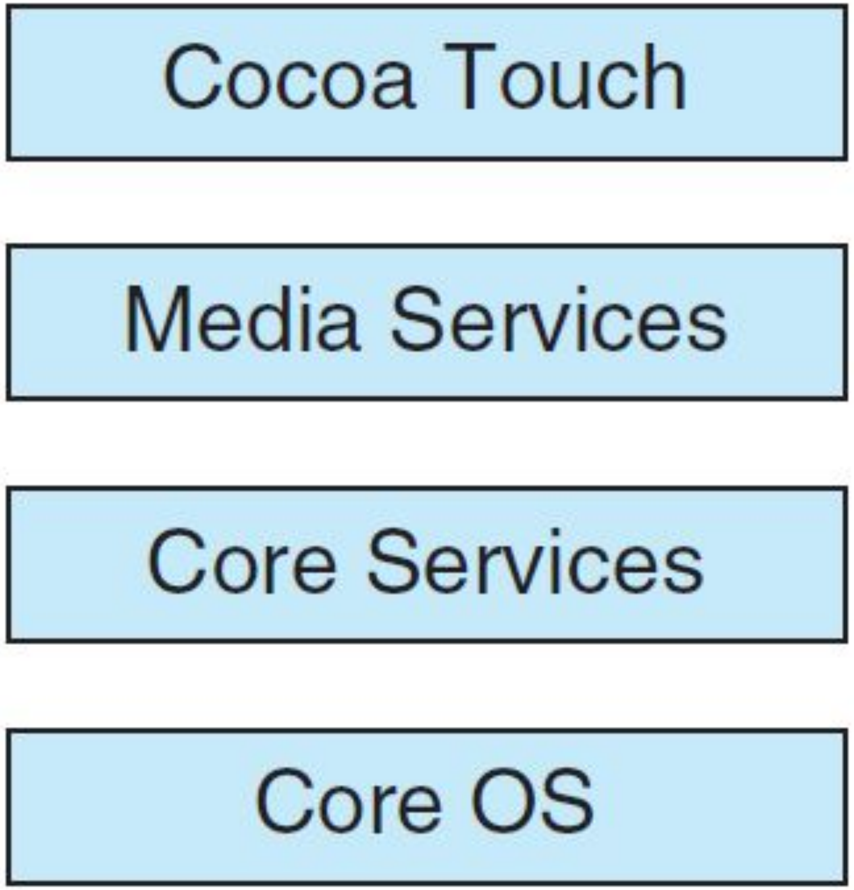
OS Structure - hybrid

- Most OSs use a hybrid architecture
  - A largely monolithic core + loadable modules
    - e.g. Linux and Solaris
  - Windows – monolithic core + microkernel + loadable modules

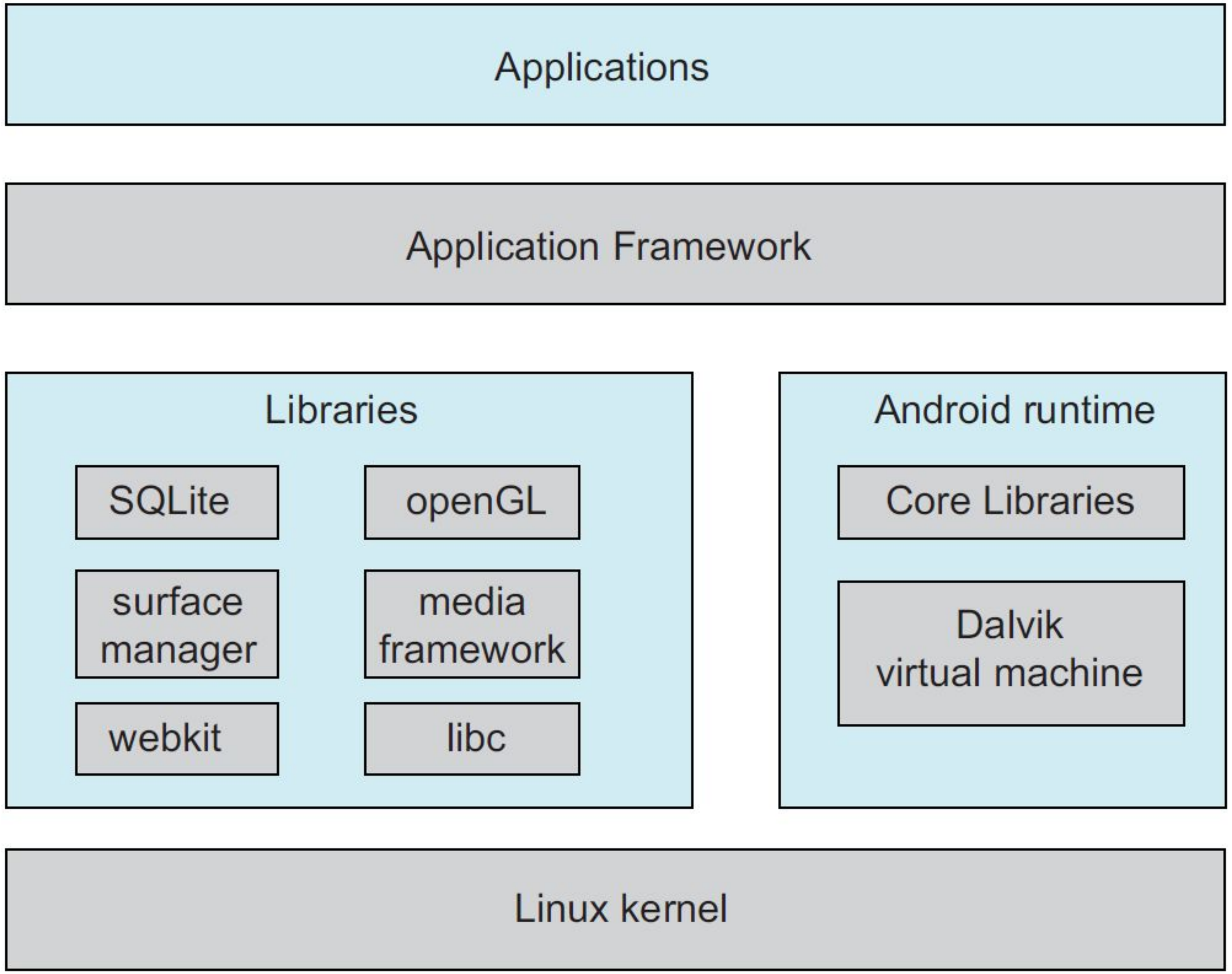
Mac OS X Structure



iOS structure



Android structure

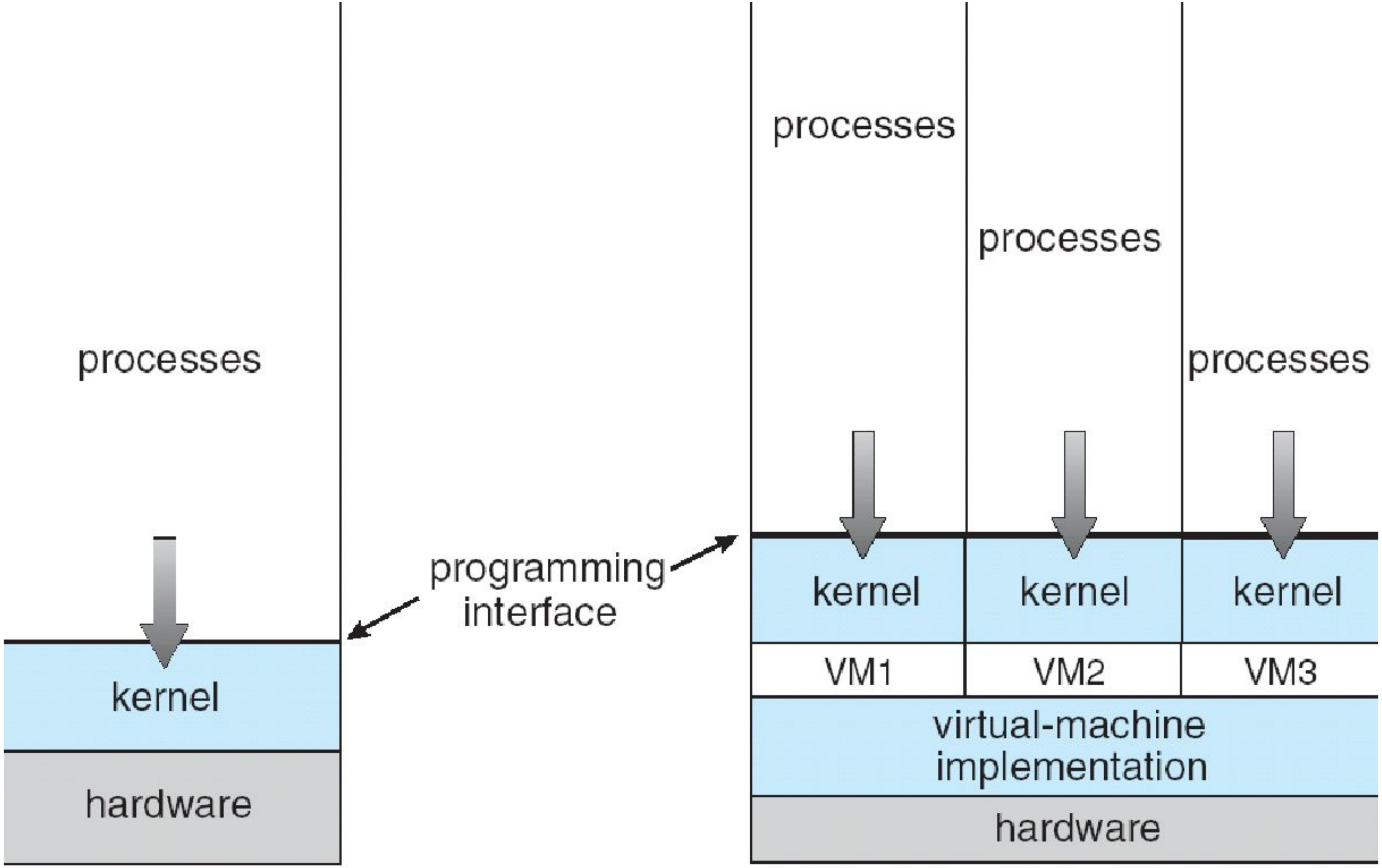


OS Structure Virtual Machines

- Virtual machine
  - Interface identical physical hardware
    - Processor
    - Memory
  - Host OS
  - Guest OS
- Multiple OSs sharing same hardware
  - Protection from each other
  - Controlled file sharing
  - Communication via networking
  - Useful for development and testing

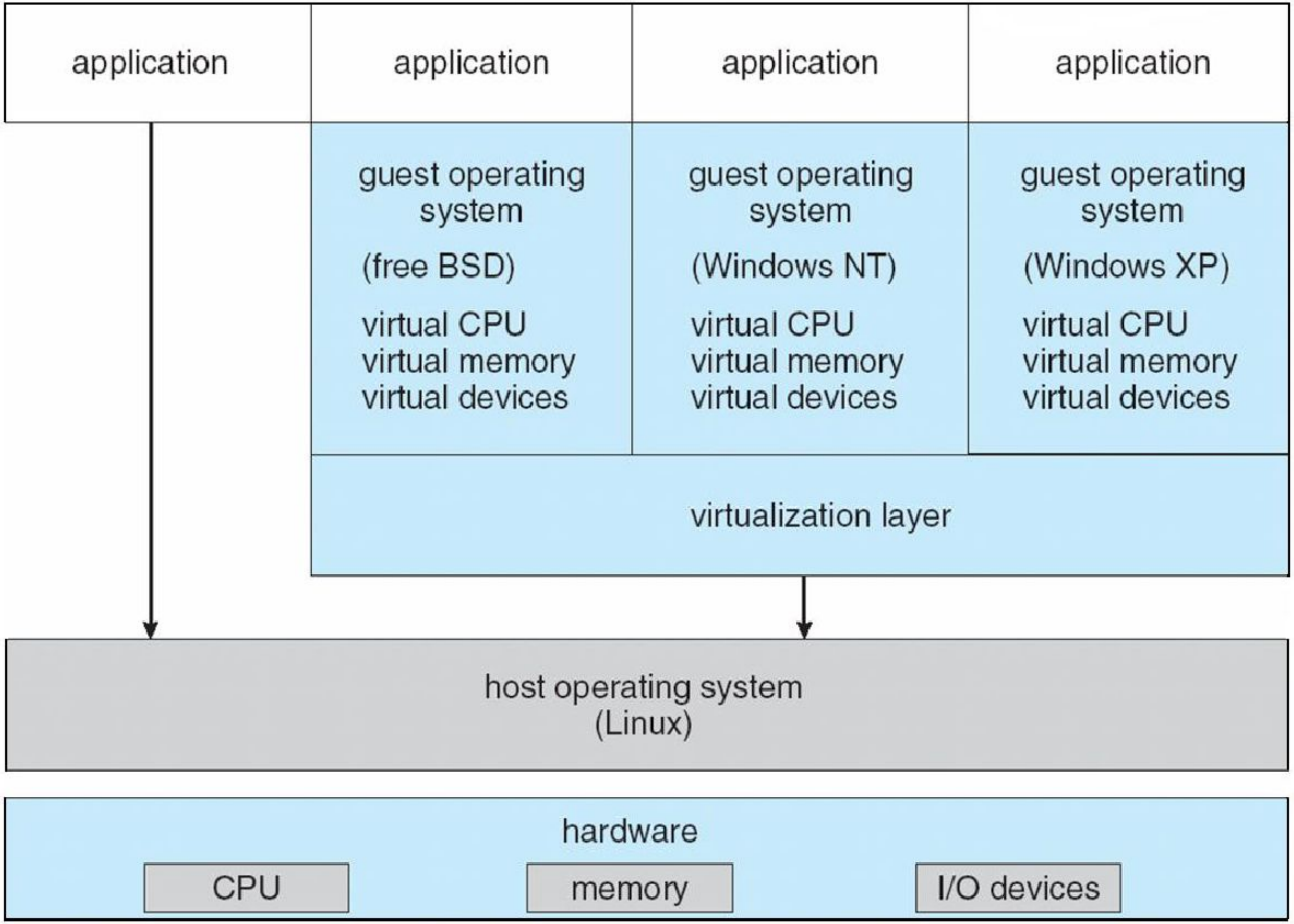
OS Structure Virtual Machines

- Consolidation
  - Many low-resource use systems into fewer busier systems
- Open Virtual format
  - Standard format for VMs
  - Allows a VM to run on may different VM platforms


processes
programming
interface
kernel
hardware
processes
processes
processes
kernel
kernel
kernel
VM1
VM2
VM3
virtual-machine
implementation
hardware

(a)

(b)


application
application
application
application
guest operating system
(free BSD)
virtual CPU
virtual memory
virtual devices
guest operating system
(Windows NT)
virtual CPU
virtual memory
virtual devices
guest operating system
(Windows XP)
virtual CPU
virtual memory
virtual devices
virtualization layer
host operating system
(Linux)
hardware
CPU
memory
I/O devices

OS Debugging

- Finding and fixing bugs
- Log files
- Core dump
  - Snapshot of memory for a failed process
- Crash dump
  - Snapshot of kernel memory
- Performance tuning
  - Trace listings – eg task manager (windows)
  - Profiling tools

OS Generation

- Configuration for specific machine and site
- SYSGEN
  - Used to build system specific compiled kernel
  - More efficient code that general kernel

System Boot

- Bootstrap Loader
  - Stored in ROM/EPROM
- Two-stage
  - ROM to Boot Block
  - Boot block has bootstrap loader