COMP3511 Operating Systems

Topic 2: Operating System Services and Structures

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Acknowledgment: The lecture notes are based on various sources on the Internet

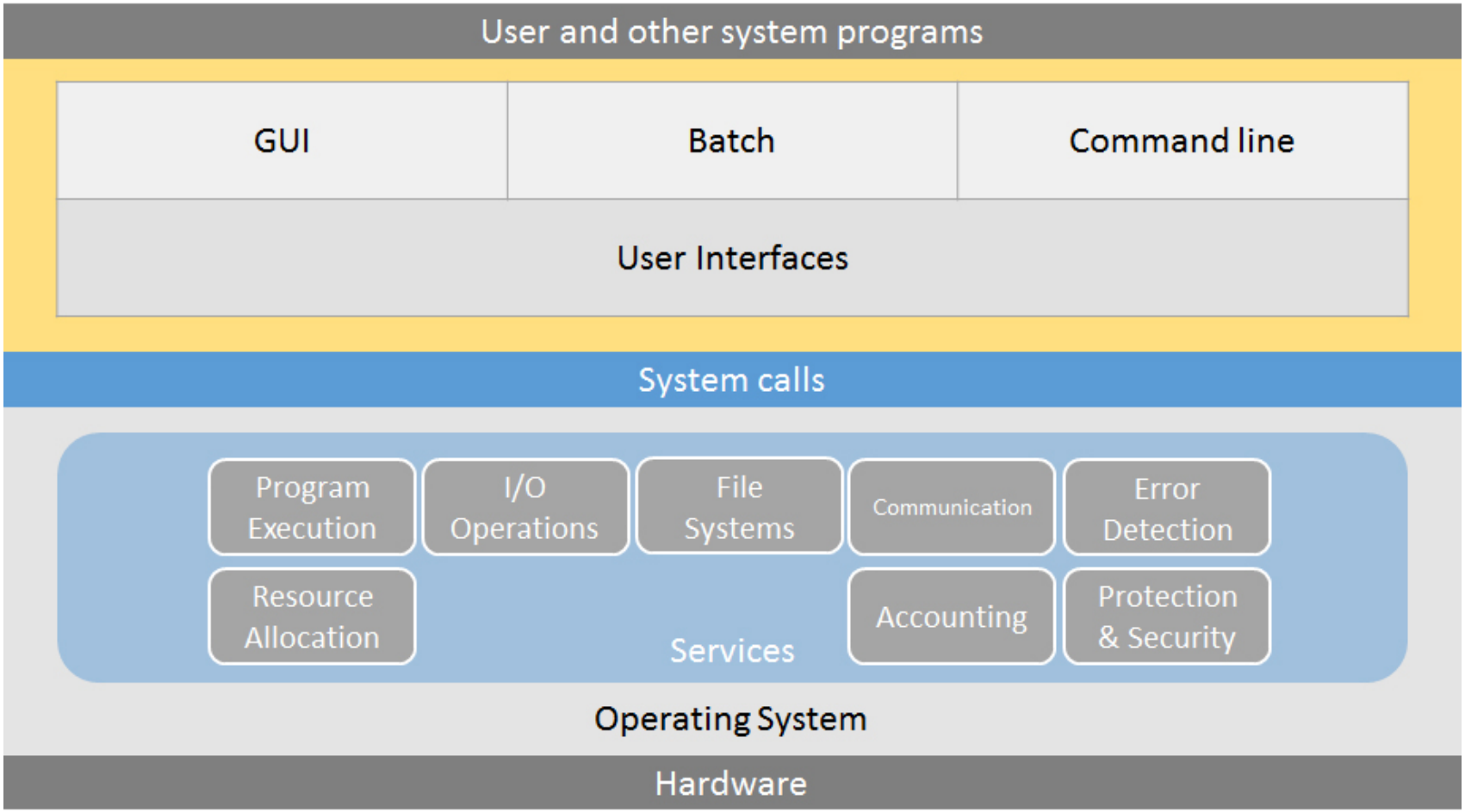
Motivation

- To understand an OS, we need to first look at its components / services and then how they are composed or organized
  - We will come back and look at each of these in detail as the course progresses
- We will also examine and compare different design issues and choices, and present the basic structure of several popular OSes

Operating System Services

- User Operating System Interface: CLI, GUI, Touchscreen Interfaces
- System Calls and Application Program Interface (API)
- System Programs

A View of Operating System Services



Operating System Services

One set of operating-system services provides functions that are useful to the user:

- User interface: Almost all operating systems have a user interface
  - Various between Command-Line (CLI), Graphical User Interface (GUI), batch interface (file execution)
  - Mobile devices may have touchscreen interface
- System calls: Allow computer program to requests services from the kernel of the OS
- Program execution: The system must be able to load a program into the 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 cannot access I/O device directly

Operating System Services (Cont'd)

One set of operating-system services provides functions that are useful to the user:

- File-system manipulation: Programs need to read and write files and directories, create and delete them, search them, list file information, permission management
- Communications: 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 appropriate action to ensure correct and consistent computing

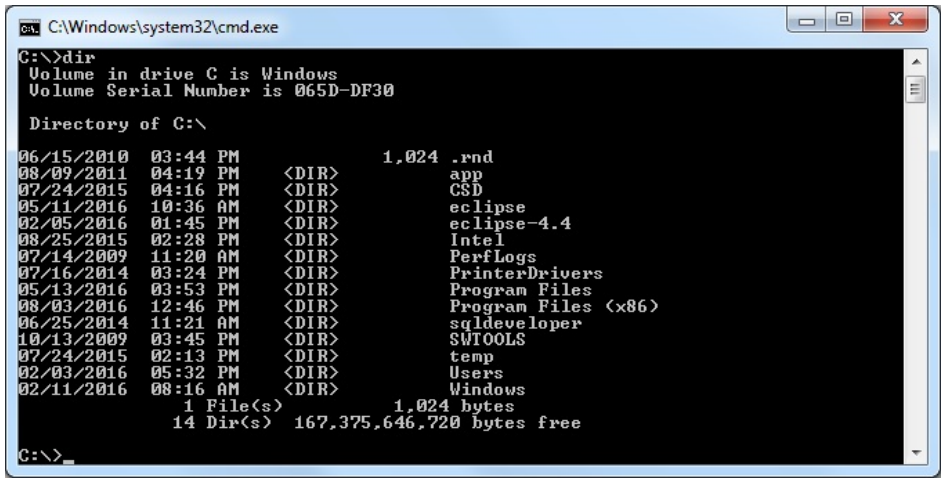
Operating System Services (Cont'd)

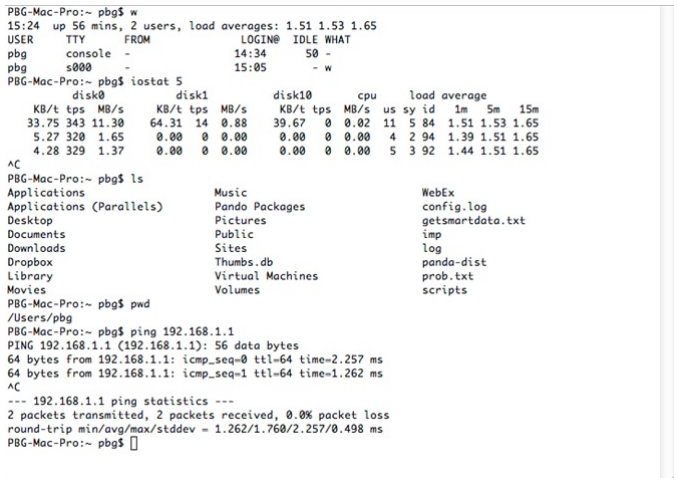
Another set of OS functions exists for ensuring efficient operation of the system itself via resource sharing

- Resource allocation: When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them
  - May types of resources: Some (e.g., CPU cycles, main memory, and file storage) may have special allocation code, others (e.g., 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

User Interface: CLI

- Command Line Interface (CLI) or command interpreter allows direct command entry
- Users can issue commands to the system
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple command interpreters to choose from - shells (e.g. sh, csh, ksh, tcsh, etc.)
  - Primarily fetches a command from user and executes it
  - UNIX, Linux, and MS-DOS





User Interface: GUI

- Graphical User Interface (GUI):
  User-friendly desktop metaphor interface
  - Usually controlled by 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)

User Interface: The Mac OS X GUI

User Interface: Batch Interface

- For batch interface, commands and directives to control those commands are entered into files, and those files are executed

Touchscreen Interfaces

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

System Calls

System Call

A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system its executed on.

- System calls are typically written in a high-level language (C or C++), but certain low-level tasks (i.e., accessing hardware) may have to be in assembly languages
- 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
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Win32 API for Windows systems
  - Java API for the Java virtual machine (JVM)

System Calls - Example of Standard API

- Consider the `read()` function that is available in UNIX and Linux systems

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

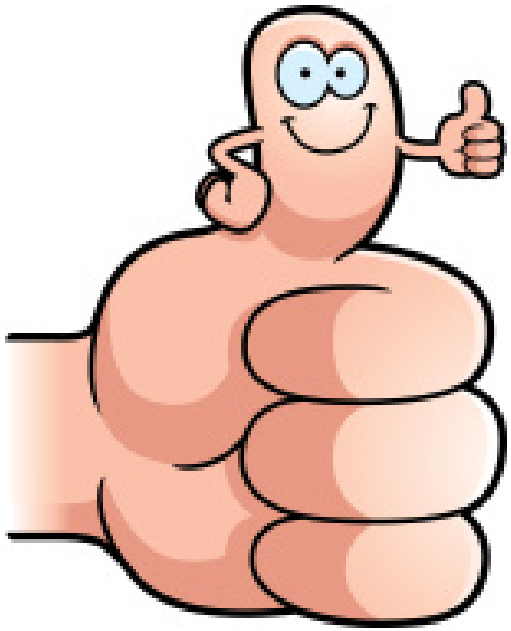
- A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the ssize_t and size_t data type (among other things). The parameter passed to read() are as follows:
  - `int fd` - the file descriptor to be read
  - `void *buf` - a buffer where the data will be read into
  - `size_t count` - the maximum number of bytes to be read into the buffer

- The API for this function is obtained from the man page by involving the command `man read` on the command line.
- System calls used: sys_read(), sys_open(), sys_close()

System Calls - Partial List of POSIX API

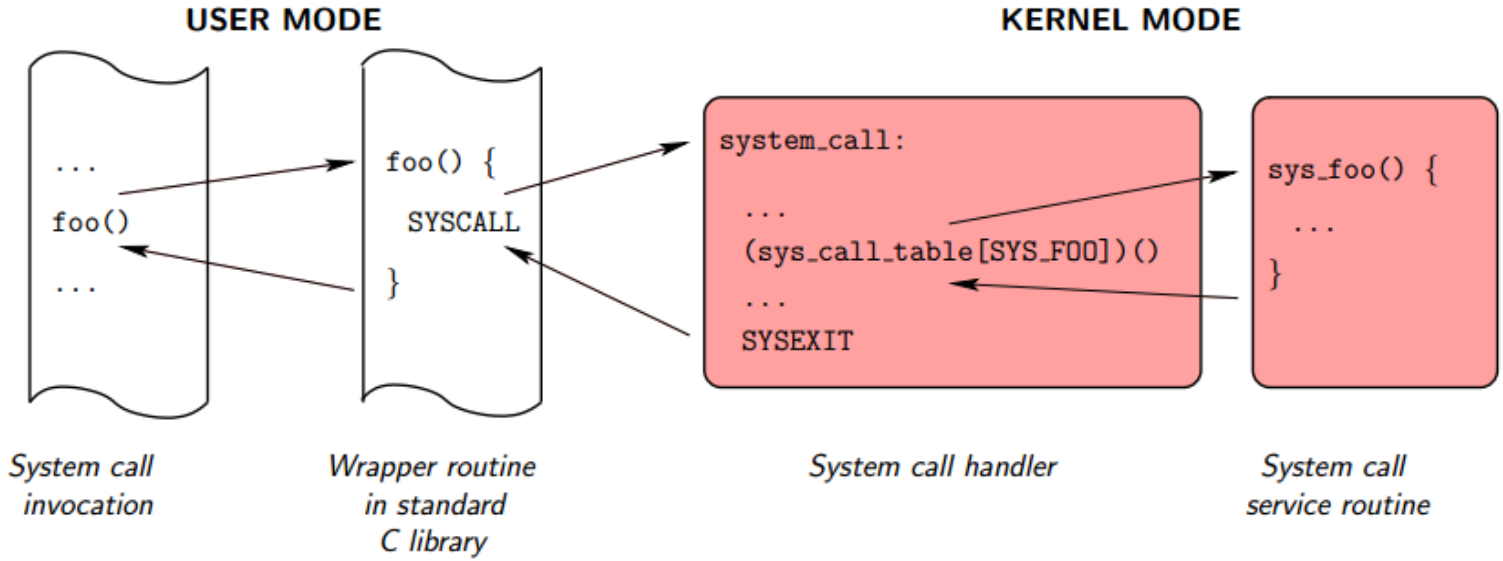
Process management			
pid	=	fork()	Create child process
pid	=	waitpid(pid, &statloc, options)	Wait for child to terminate
s	=	execve(name, argv, environp)	Replace process's image
s	=	kill(pid, signal)	Send
		exit(status)	Terminate process
File management			
fd	=	open(file, how, ...)	Open file for read/write
s	=	close(fd)	Close open file
n	=	read(fd, buffer, nbytes)	Read data from file into the buffer
n	=	write(fd, buffer, nbytes)	Write data from buffer to file
pos	=	lseek(fd, offset, whence)	Move file pointer
s	=	stat(name(name, &buf)	Get file's status information
Directory and file system management			
s	=	mkdir(name, mode)	Create new directory
s	=	rmdir(name)	Remove empty directory
s	=	link(name1, name2)	Create link to file
s	=	unlink(name)	Remove directory entry
s	=	mount(special, name, flag)	Mount file system
s	=	umount(special)	Unmount file sytsem
s	=	chdir(dirname)	Change working directory
s	=	chmod(name, mode)	Change file's protection bits
Other			
secs	=	time(&seconds)	Get elapsed time since 1/1/70

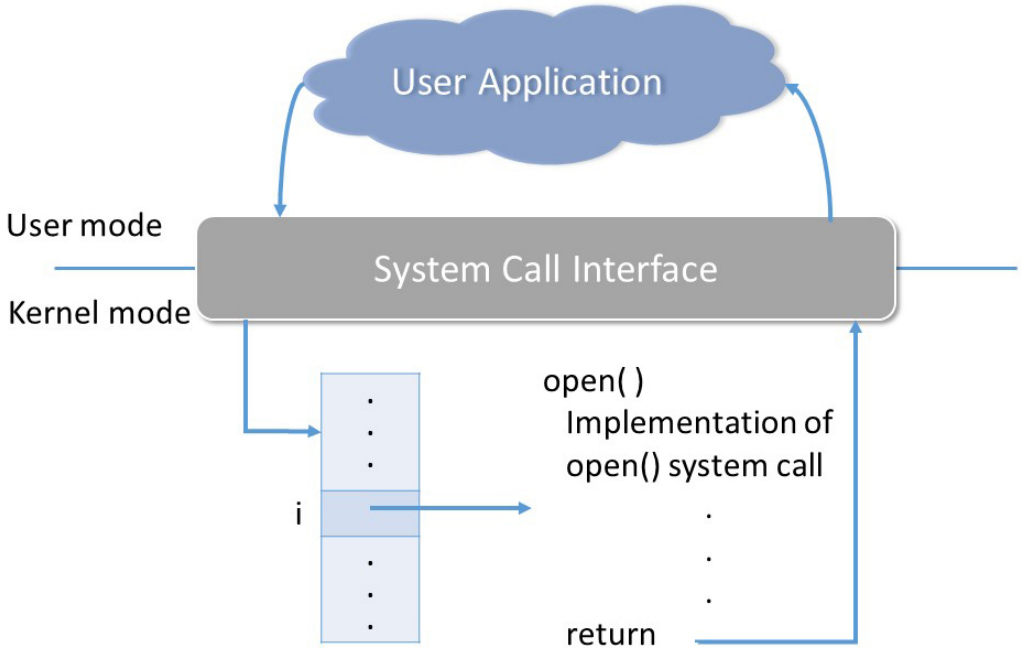
System Call Implementation



- For most programming languages, libraries are available to provide a system call interface that serves as the link to OS system calls
- Typically, a number is associated with each system call and the system-call interface maintains a table indexed according to these numbers
- The callers know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API

System Call Implementation (Cont'd)



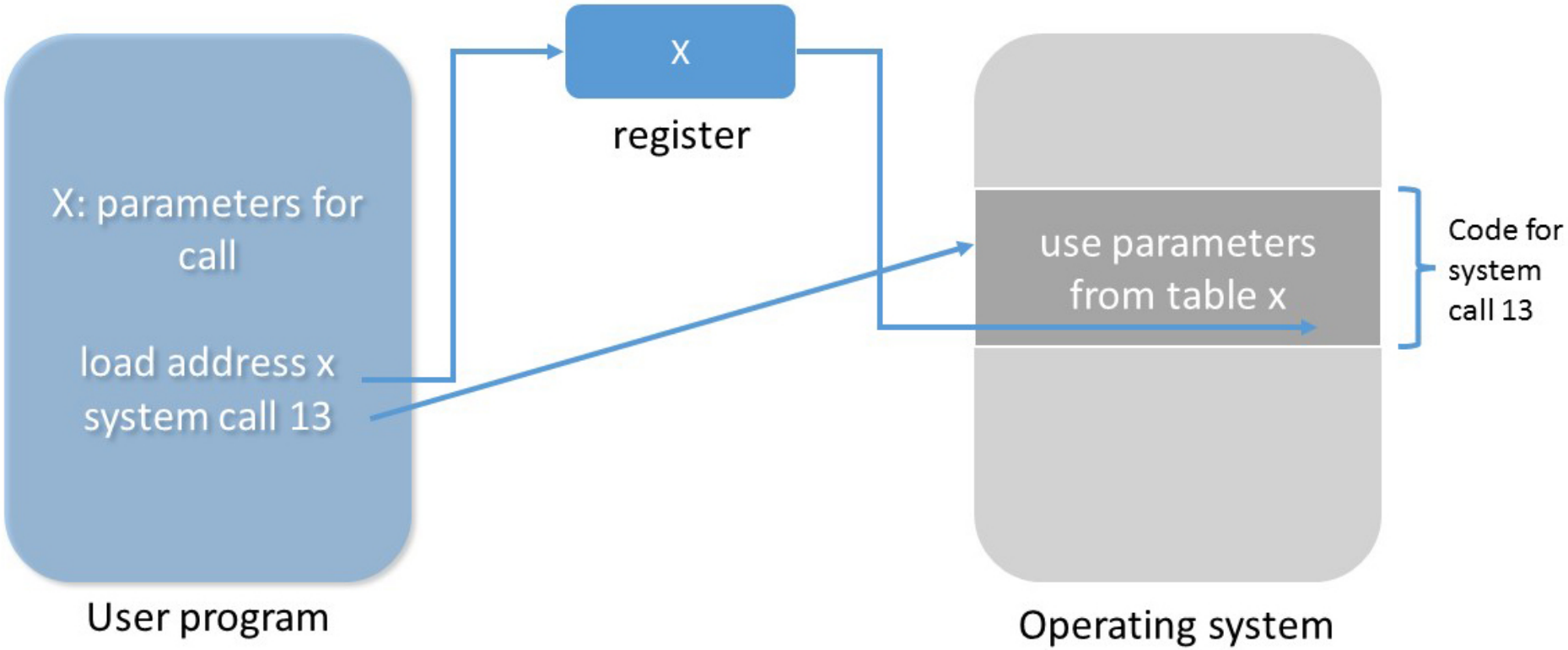


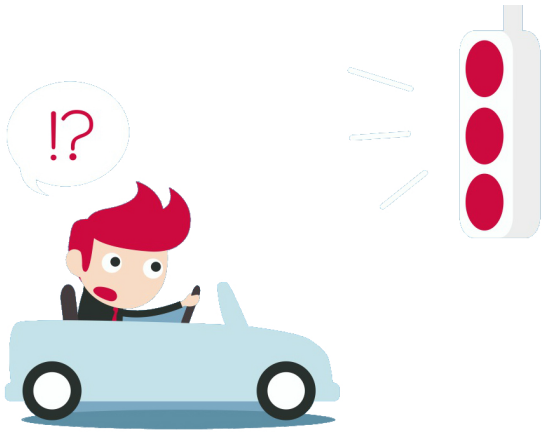
System Call Parameter Passing

- 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
  1. Simplest: Pass the parameters in registers
     (Note: In some cases, may be more parameters than registers)
  2. Parameters stored in a block, or table, in memory, and address of block passed as parameter in a register
     (Note: This approach is taken by Linux and Solaris)
  3. 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 of length of parameters being passed

System Call Parameter Passing - Simplest





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 file, 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
- 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'd)

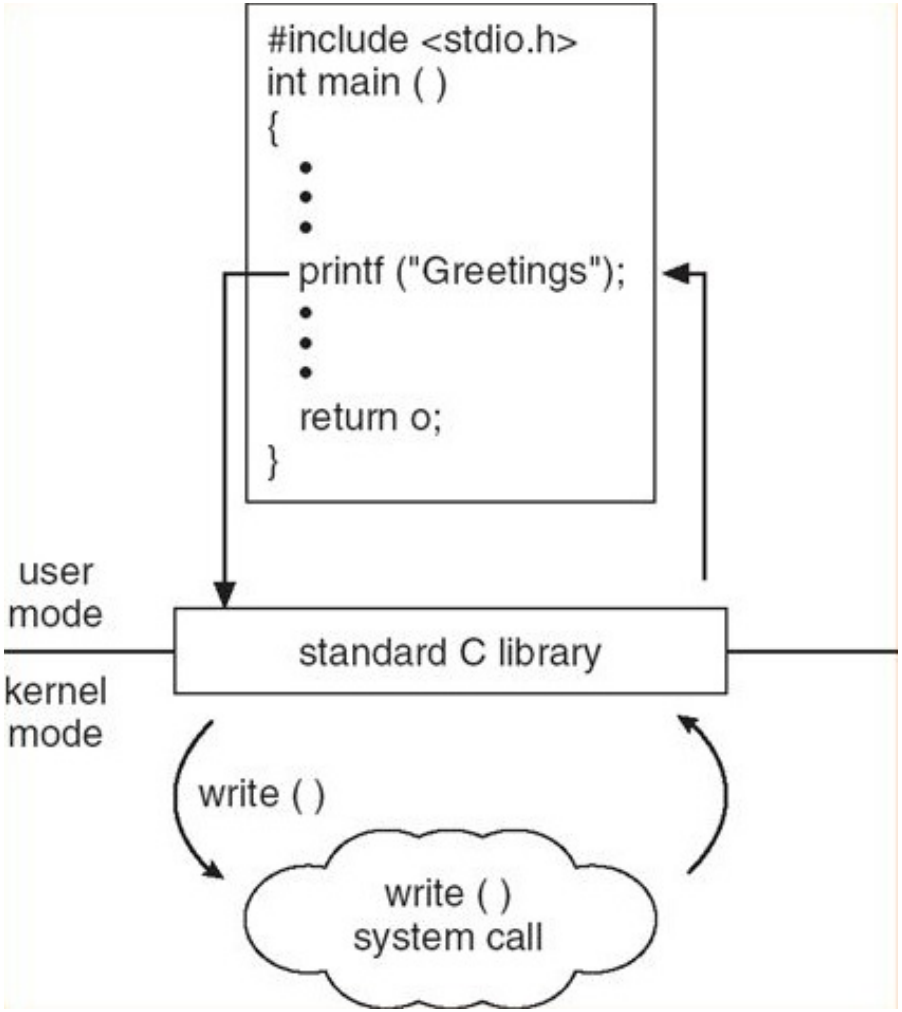
- Communications
  - create, delete communication connection
  - send, receive messages if message passing model to host name or process name from client to server
  - Shared-memory model create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices
- Protection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

Examples of UNIX and Windows System Calls

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

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 then intercepts this call and invokes the necessary system call write(), takes the return value and pass to the user program



System Programs

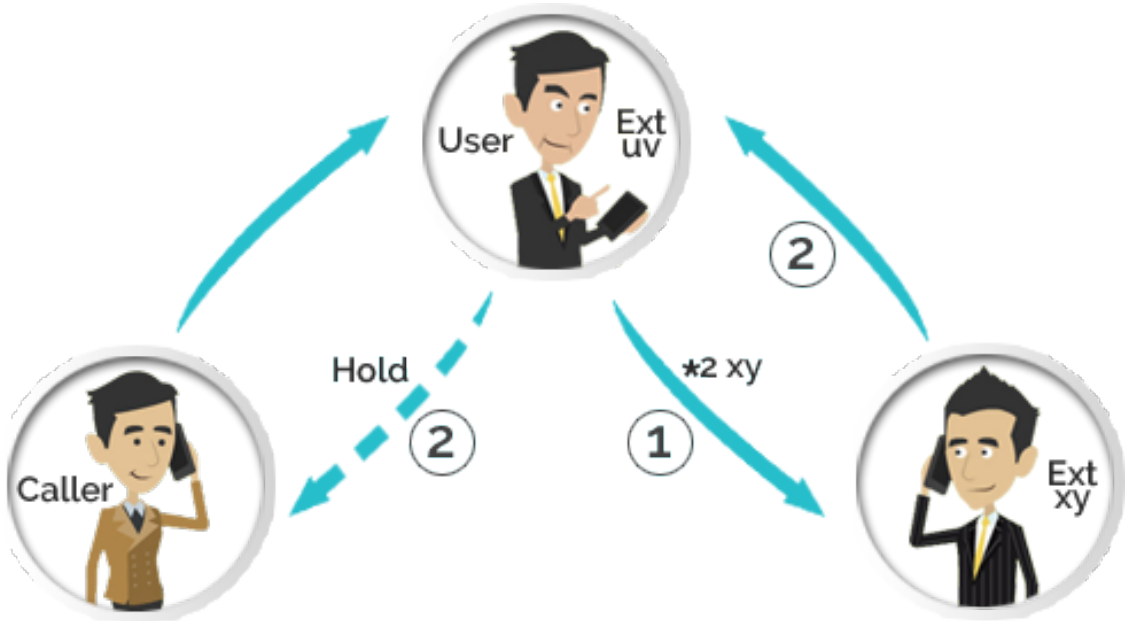
- System programs (or system utilities), provide a convenient environment for program development and execution
- They can be divided into
  - File manipulation: create, delete, copy, rename, print, dump, list, ...
  - Status information: time/date, memory usage, logging and debugging info
  - File modification: file editor for example
  - Programming-language support: compiler, assemblers, debuggers
  - Program loading and execution
  - Communications
  - Background services: 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 operating system programs, not the actual system calls

Operating System Design, Implementation and Structures

- OS Design
- OS Implementation
- OS Structures

Operating System Design

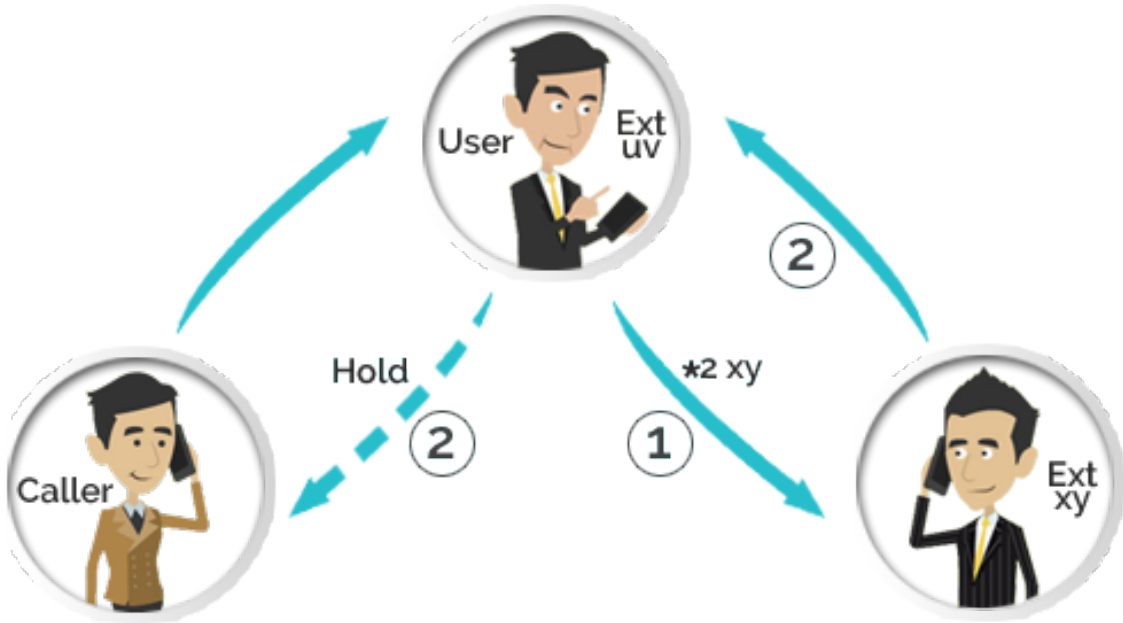
- Internal structure of different operating systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system (batch, time sharing, single user, multi-user, distributed, real time, or general purpose)
- User goals and System goals - much harder to specify
  - 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 (Cont'd)

- Important principle to separate: policy and mechanism
  - Policy: What will be done?
  - Mechanism: How to do it?

  Separation of policy from mechanism allows maximum flexibility if policy decisions are to be changed later
- Specifying and designing OS is highly creative task of software engineering
- Policy decision are important for all resource allocations

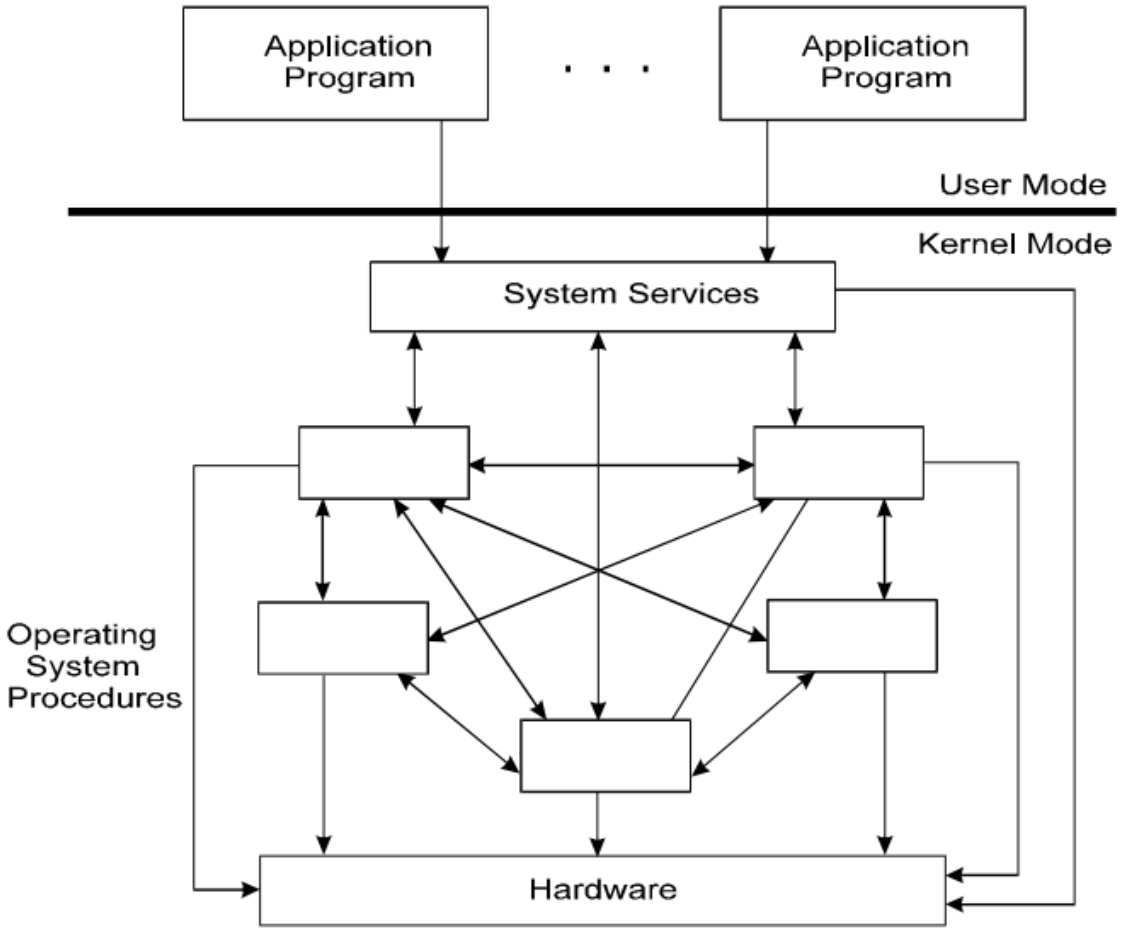


Operating System Implementation

- Much variation in implementation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now most OSes are written in C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly language
  - 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 hard program, must be carefully engineered to function properly and be modified easily
- Traditionally, OS is designed as a monolithic system



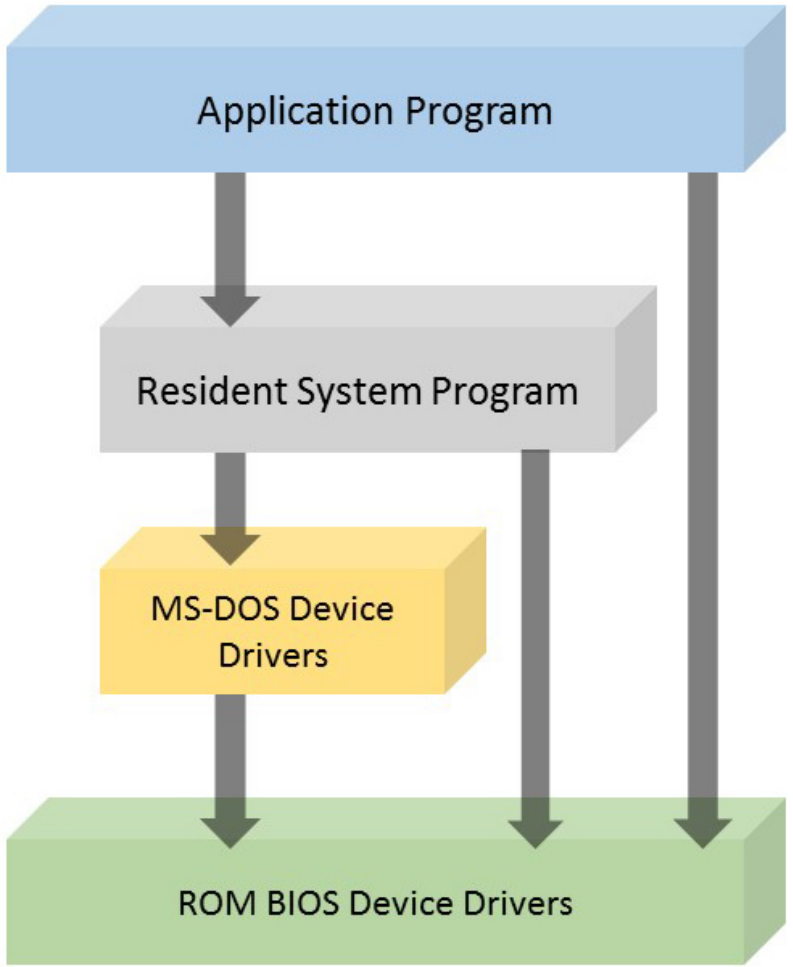
Monolithic system

Operating System Structure (Cont'd)

- Now, 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
  - Layered Approach
  - Micro-kernel System Structure
  - Modular Approach
  - Hybrid Systems
  - ...

Simple Structure

- Such OSes do not have well-defined structure, usually started as small, simple and limited systems
- Example: 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

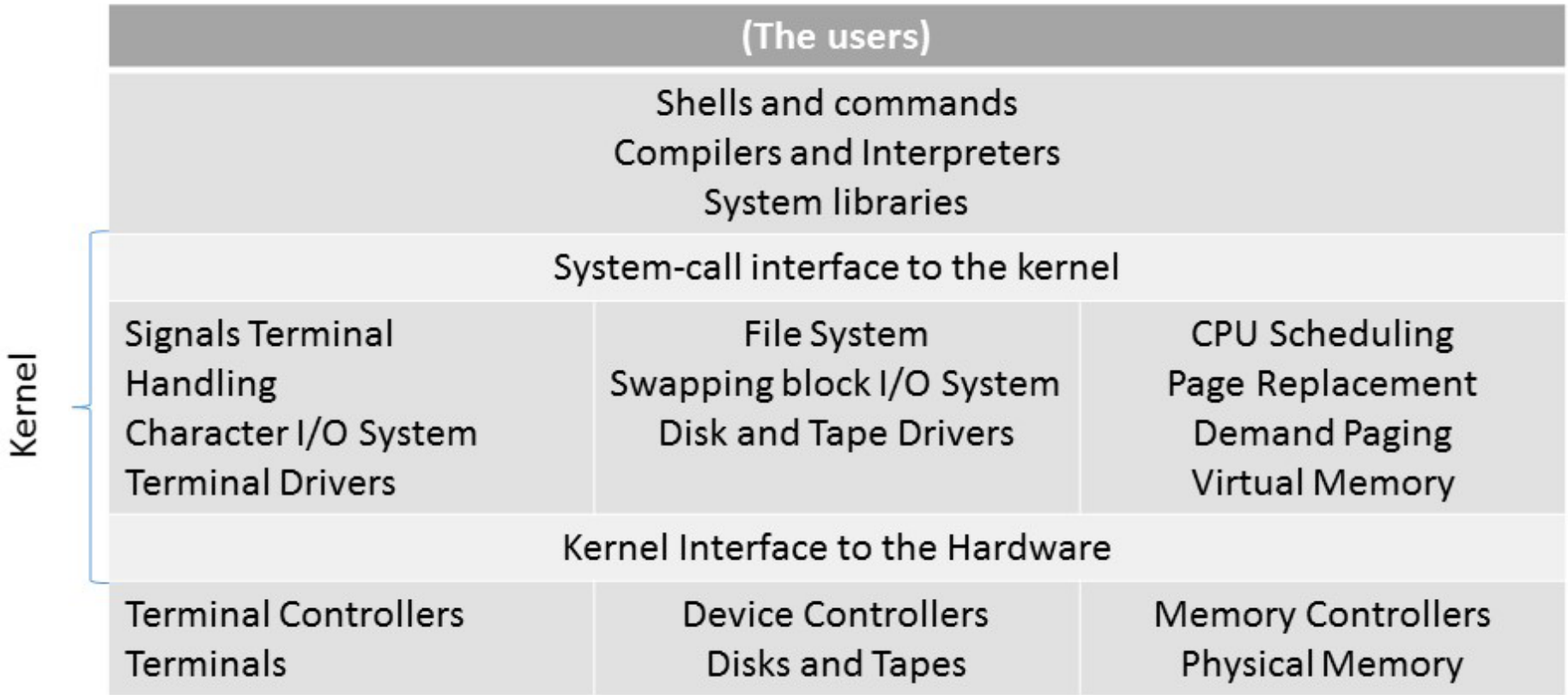


Simple Structure (Cont'd)

- Example: 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

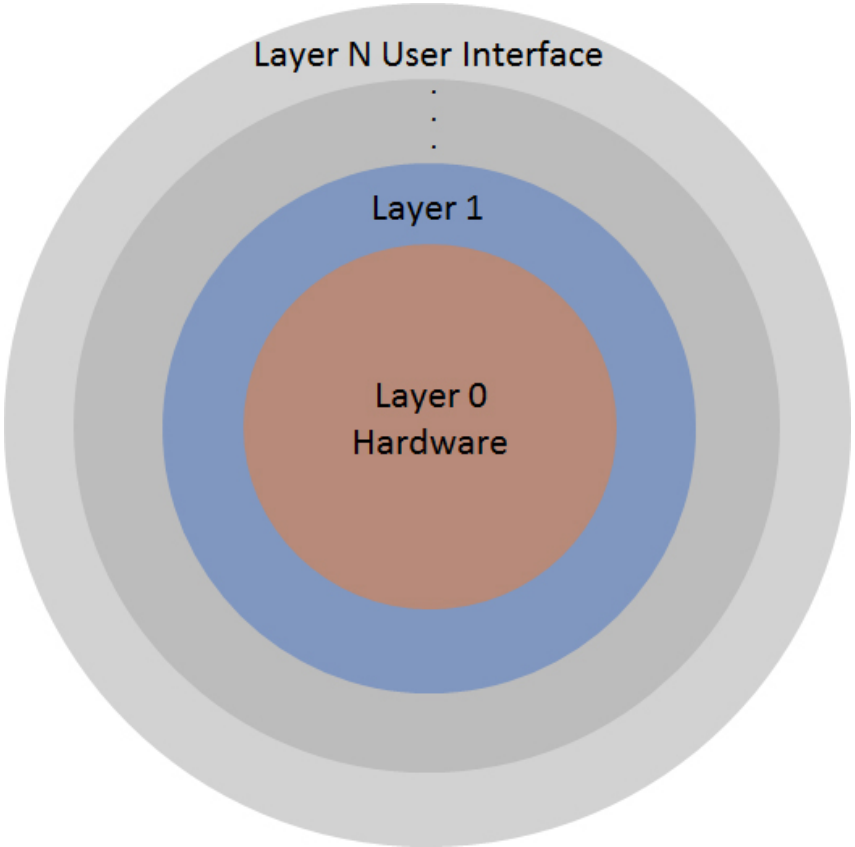


Layered Approach

- 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 (layer N) is the user interface
- The main advantage of a layered approach is the simplicity of construction and debugging. The layers are selected such that each uses function (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'd)

- An OS layer is an implementation of an abstract object made up of data and operations manipulating those data (i.e. class)
- A layer M consists of data structure and a set of routines that can be involved by high-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



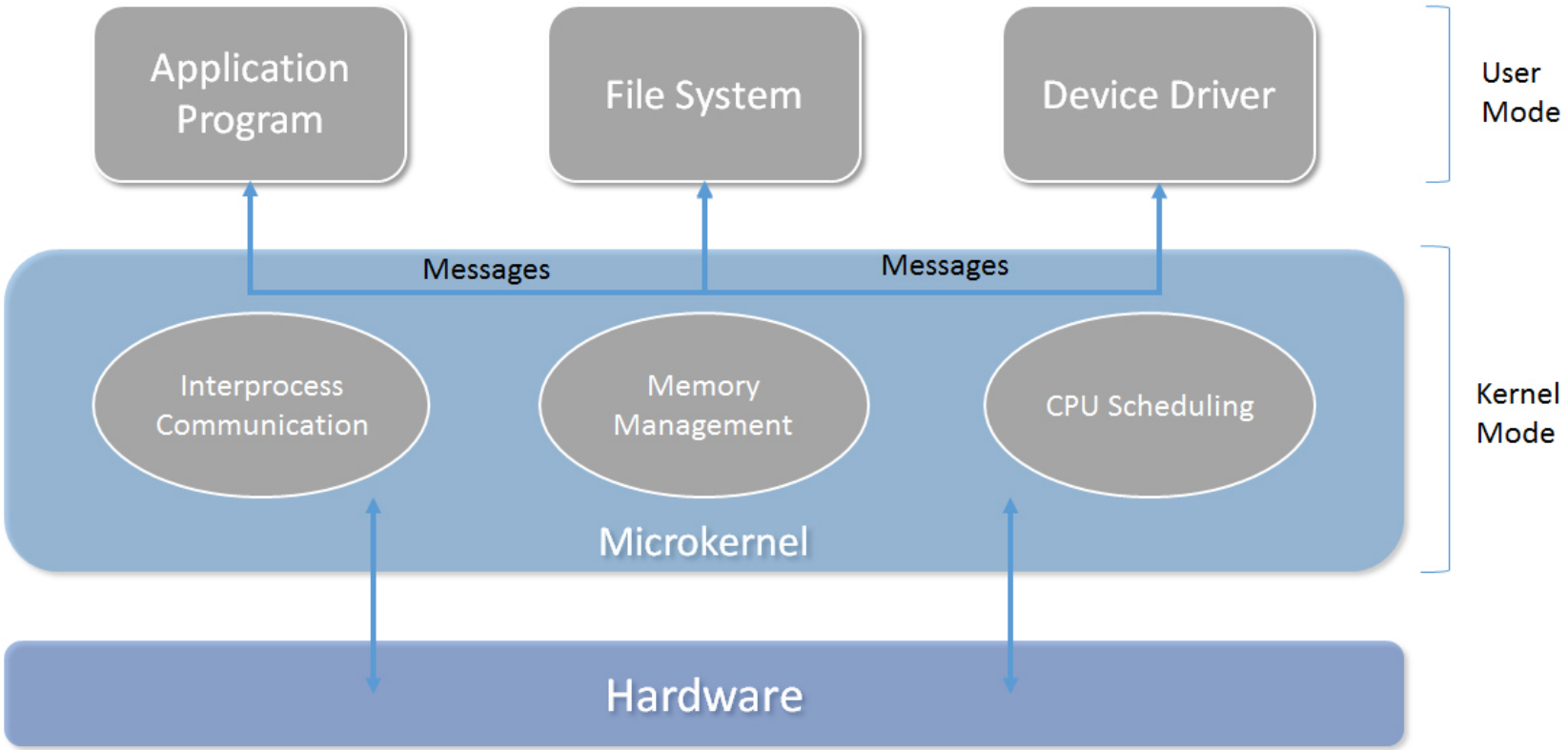
Microkernel System Structure

- The kernel became large and difficult to manage
- Removing all non-essential 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, i.e. using a form of communication between objects

Micokernel System Structure (Cont'd)

- 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 (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 micokernels can suffer due to increased system-function overhead, user space to kernel space communication
  - Earlier Windows NT had a layered microkernel, now more monolithic

Micorkernel System Structure

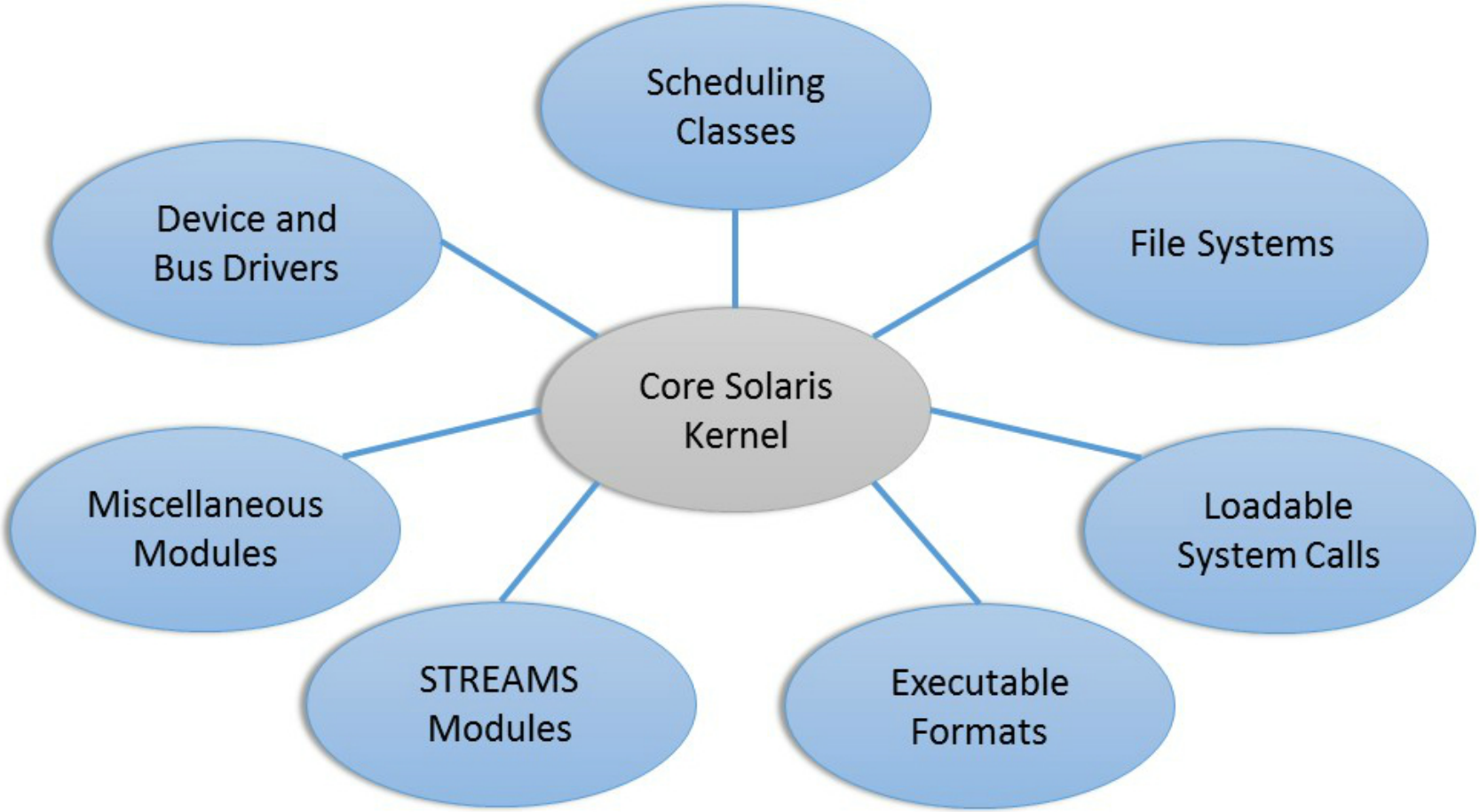


Modules / Modular Approach

- Perhaps the best current technology 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 micokernel 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

Modules / Modular Approach (Cont'd)

- Example: Solaris Modular Approach



Hybrid Systems

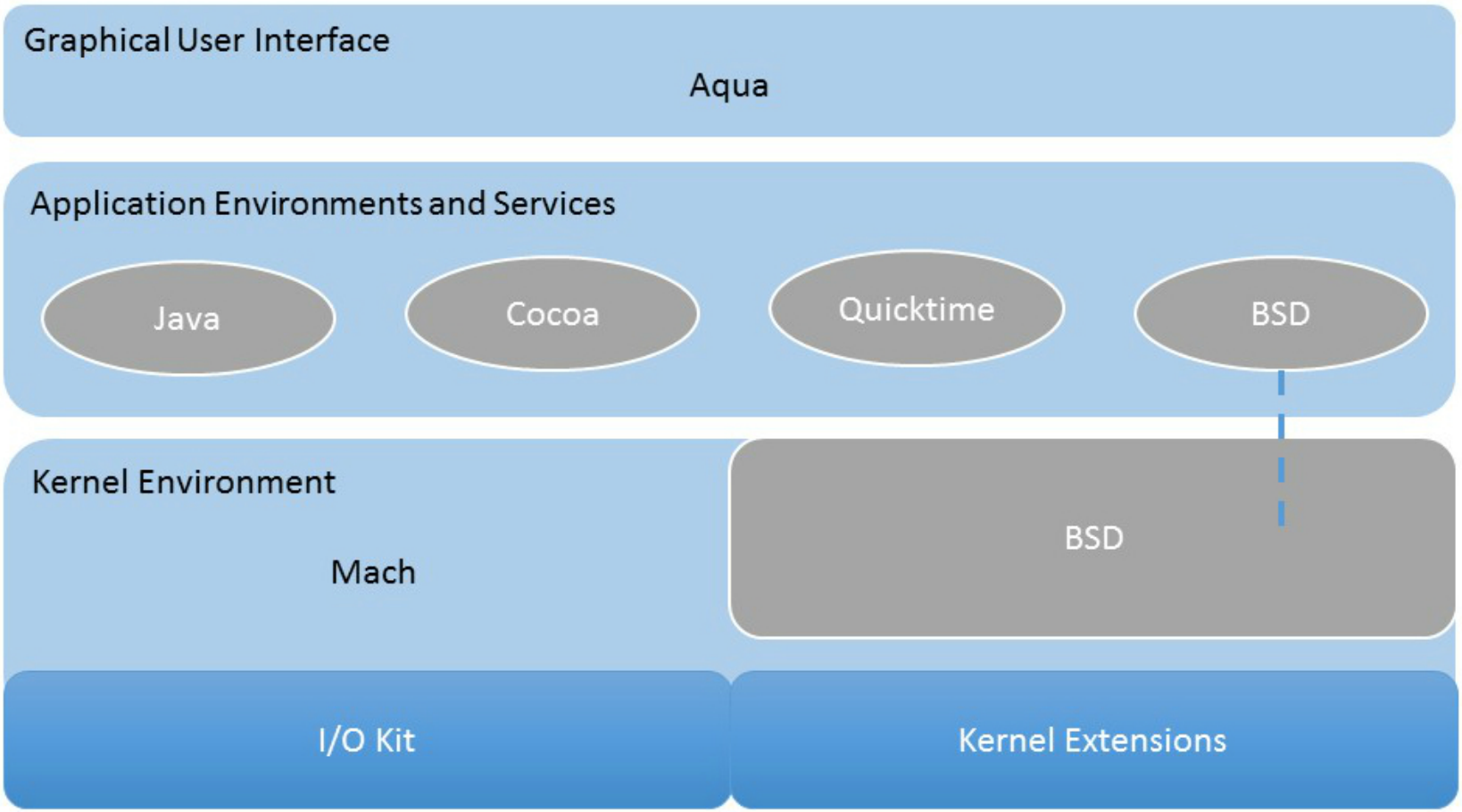
- In practice, very few operating systems adopt a single, strictly defined structure.
- Instead they combine different structures, resulting in hybrid systems that address performance, security, usability needs
  - Linux and Solaris
    - ★ Monolithic, because having the OS in a single address space 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.

Hybrid Systems (Cont'd)

- Example: 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 micokernel and BSD UNIX parts, plus I/O kits and dynamically loadable modules (called kernel extensions)
    - ★ MACH microkernel provides memory management support for RPC, IPC
    - ★ BSD components provide a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads

Hybrid Systems (Cont'd)

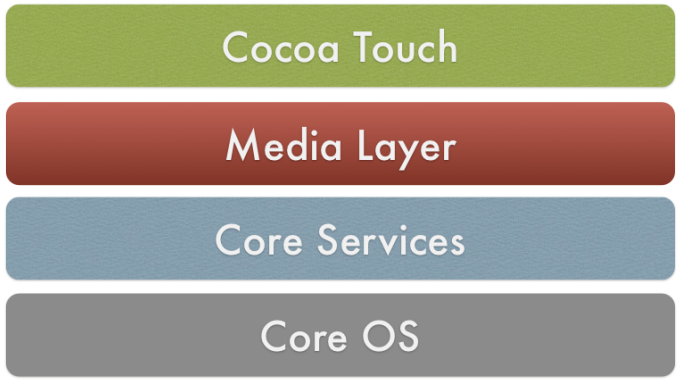
- Mac OS X



Hybrid Systems (Cont'd)

Example: 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 service provides support for cloud computing, databases
- Core OS represents the core operating system, which is based on Mac OS X kernel environment

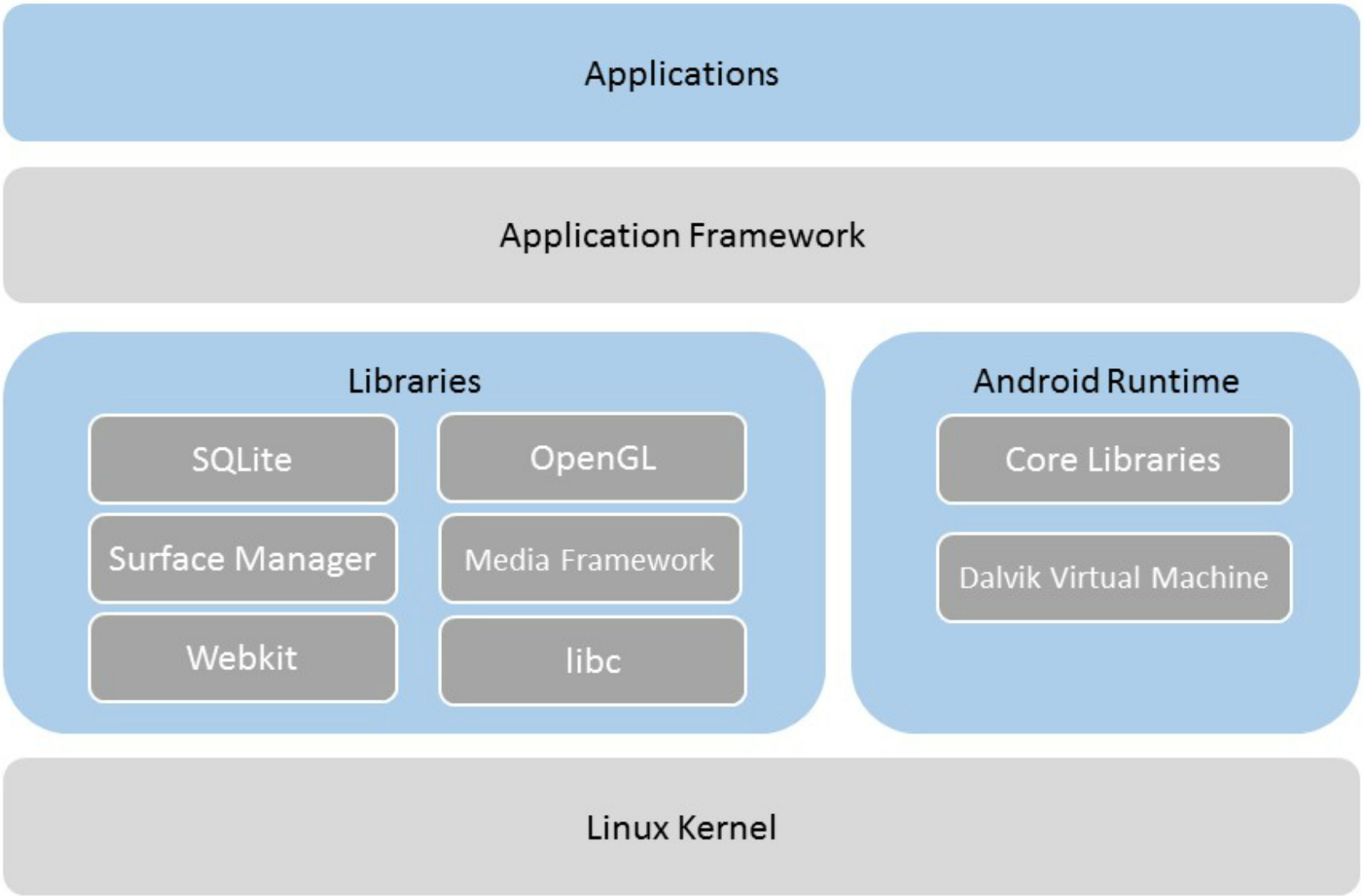
Hybrid Systems (Cont'd)

Example: 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
- Run-time 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

Hybrid Systems (Cont'd)

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
- 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

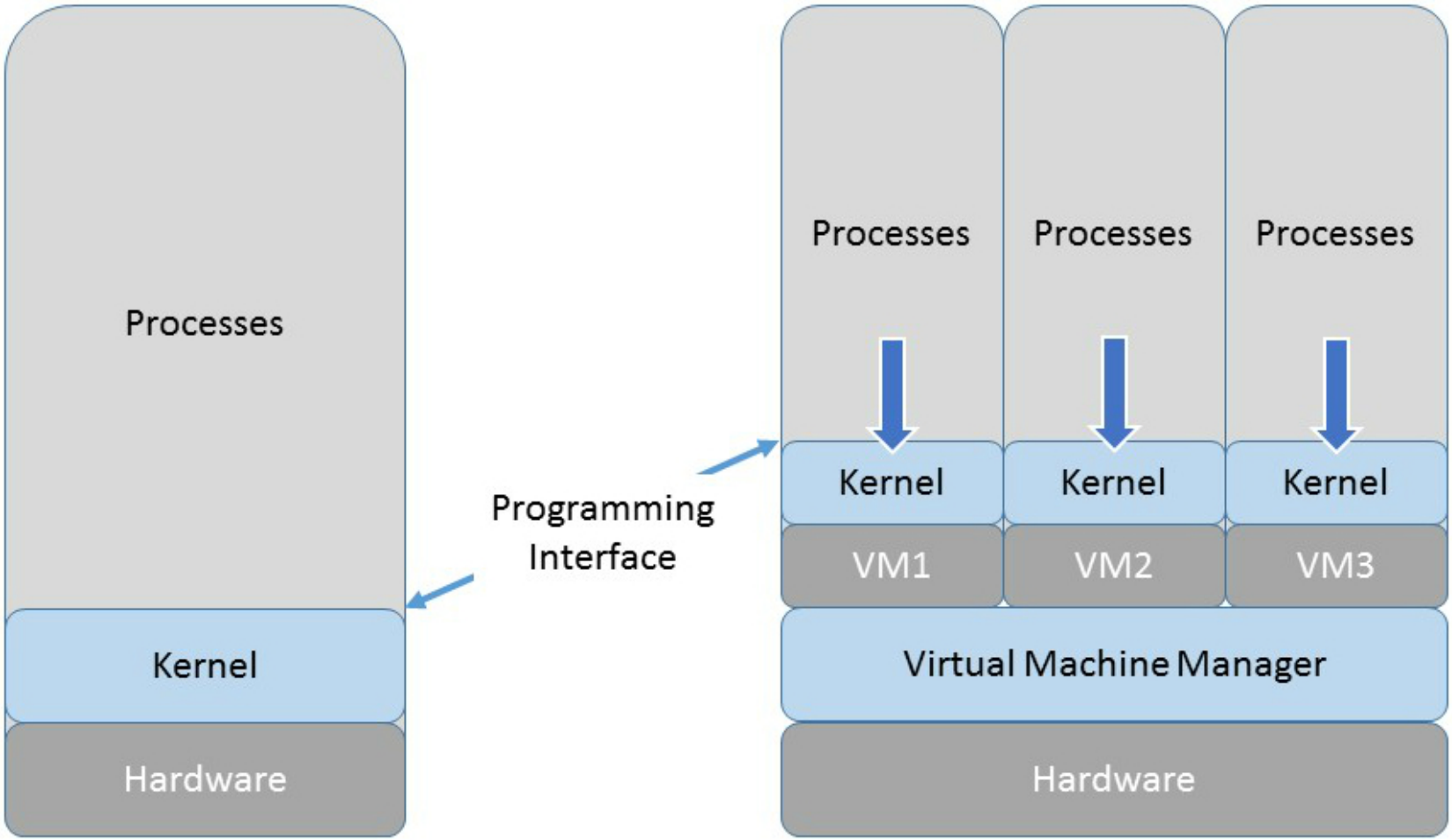
We will discuss more about "process" in the next topic

Virtual Machines (Cont'd)

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 compiled for the IBM CPU to run on the Intel CPU
- Virtualization - OS running as guest OS within another OS (host)
  - Virtual Machine Manager (VMM) 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 from 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'd)



Non-virtual machine

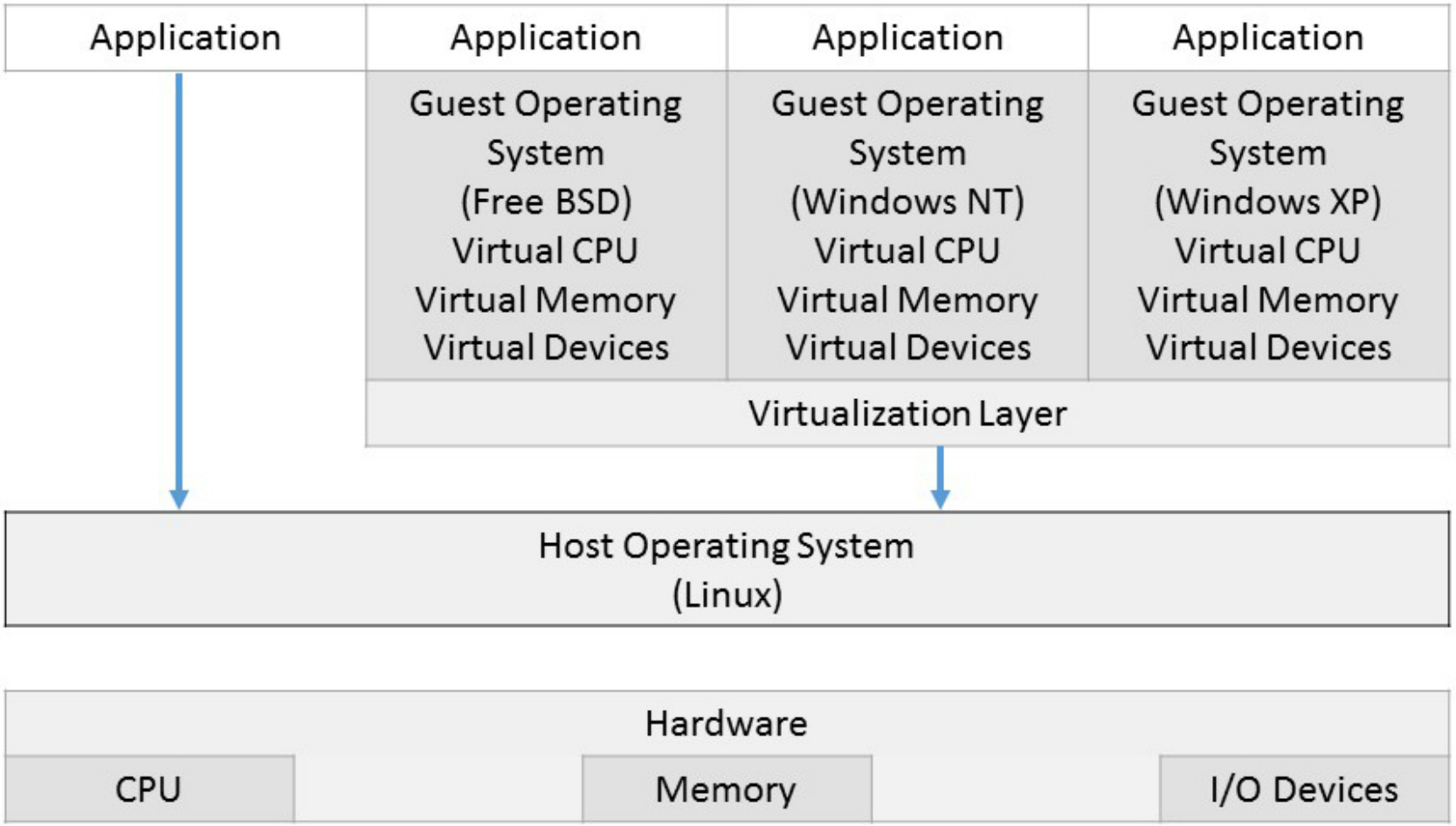
Virtual machine

Virtual Machines (Cont'd)

- Use cases involve laptops and desktops running multiples 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

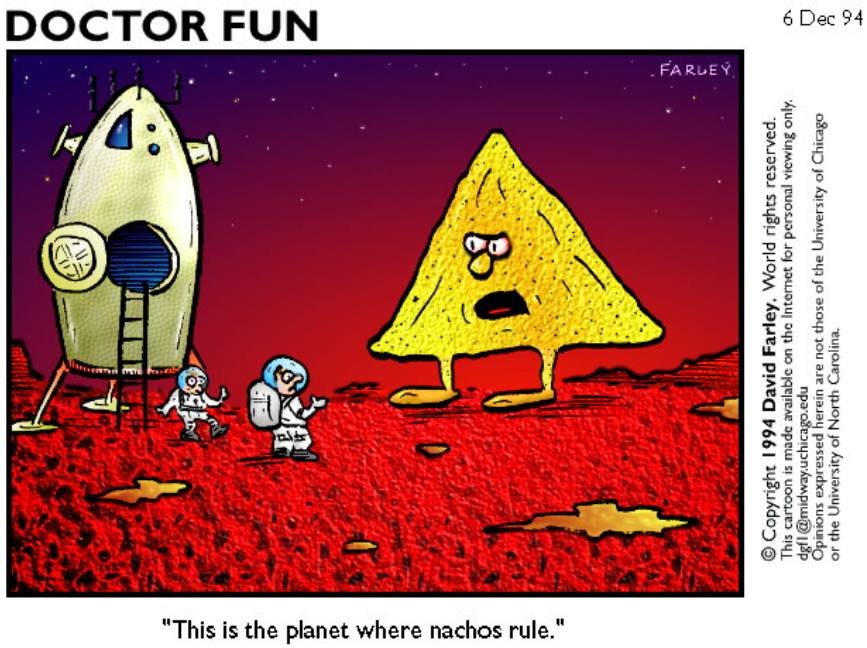
Virtual Machines (Cont'd): Layers of OS

- 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, multiprocessing, virtual memory, file systems and networking
  - Nachos written in C++ and well-organized, making it easier to understand the operation of a typical operating system



That's all!

Any questions?