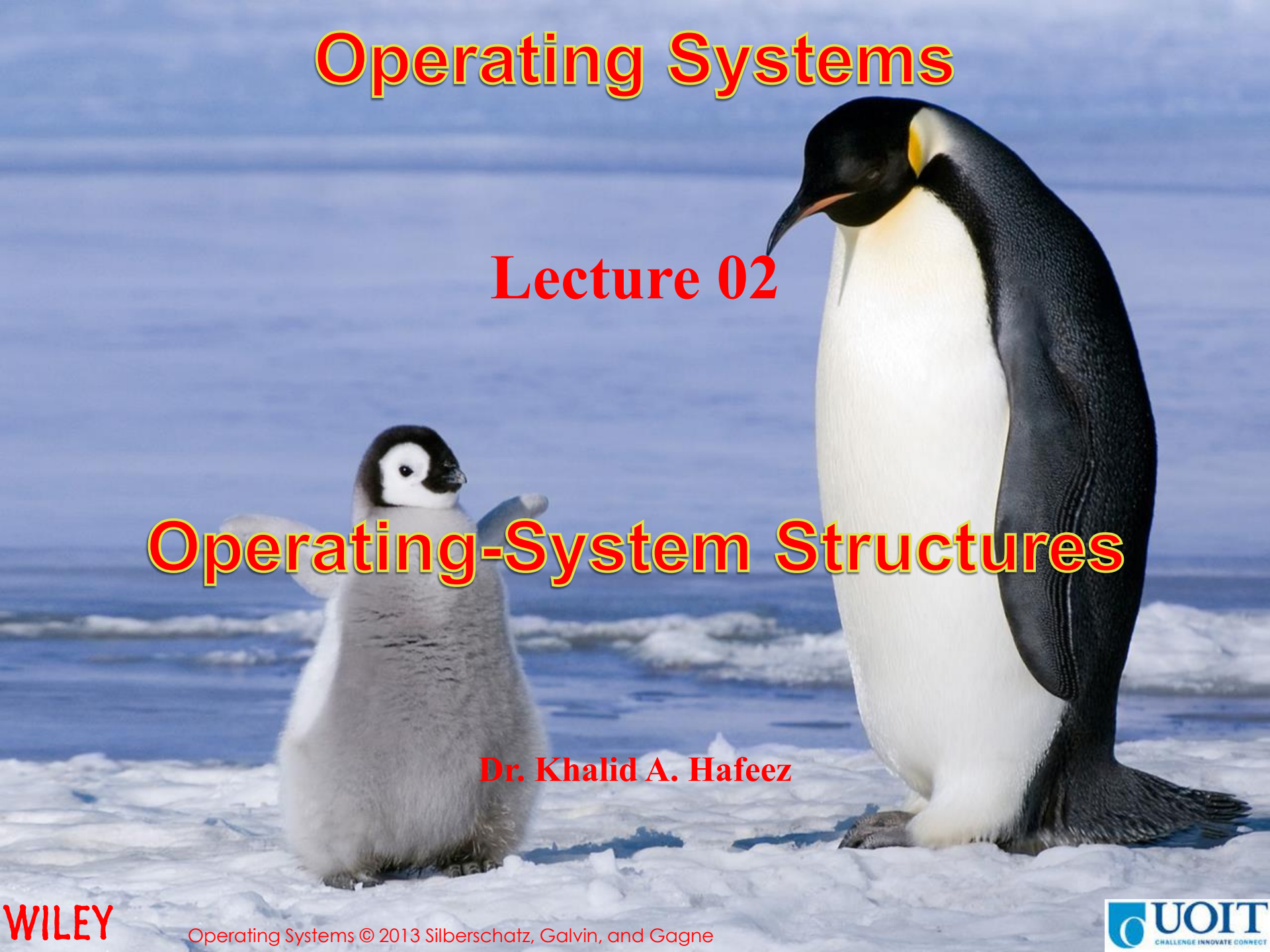
Operating Systems

Lecture 02

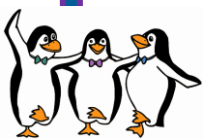
Operating-System Structures

Dr. Khalid A. Hafeez



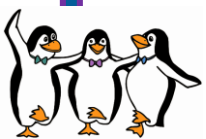
Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot



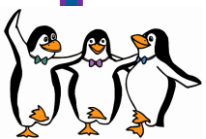
Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot



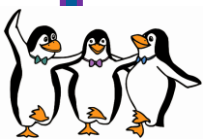
Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- **The OS can be studied from three different angles:**
  - The services the OS provides
  - The interface that the OS makes available to users and programmers
  - The OS components and their interconnections.



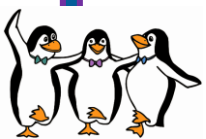
Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - **User interface:** Almost all operating systems have a user interface (**UI**).
    - It has several forms:
      - **Command-Line (CLI),**
      - **Graphics User Interface (GUI),**
      - **Batch interface:** commands entered into files, to be executed.
  - **Program execution:** The system must be able to load a program into 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



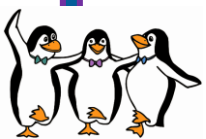
Operating System Services

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **File-system manipulation:** The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
  - **Communications**: Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - **Error detection:** OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

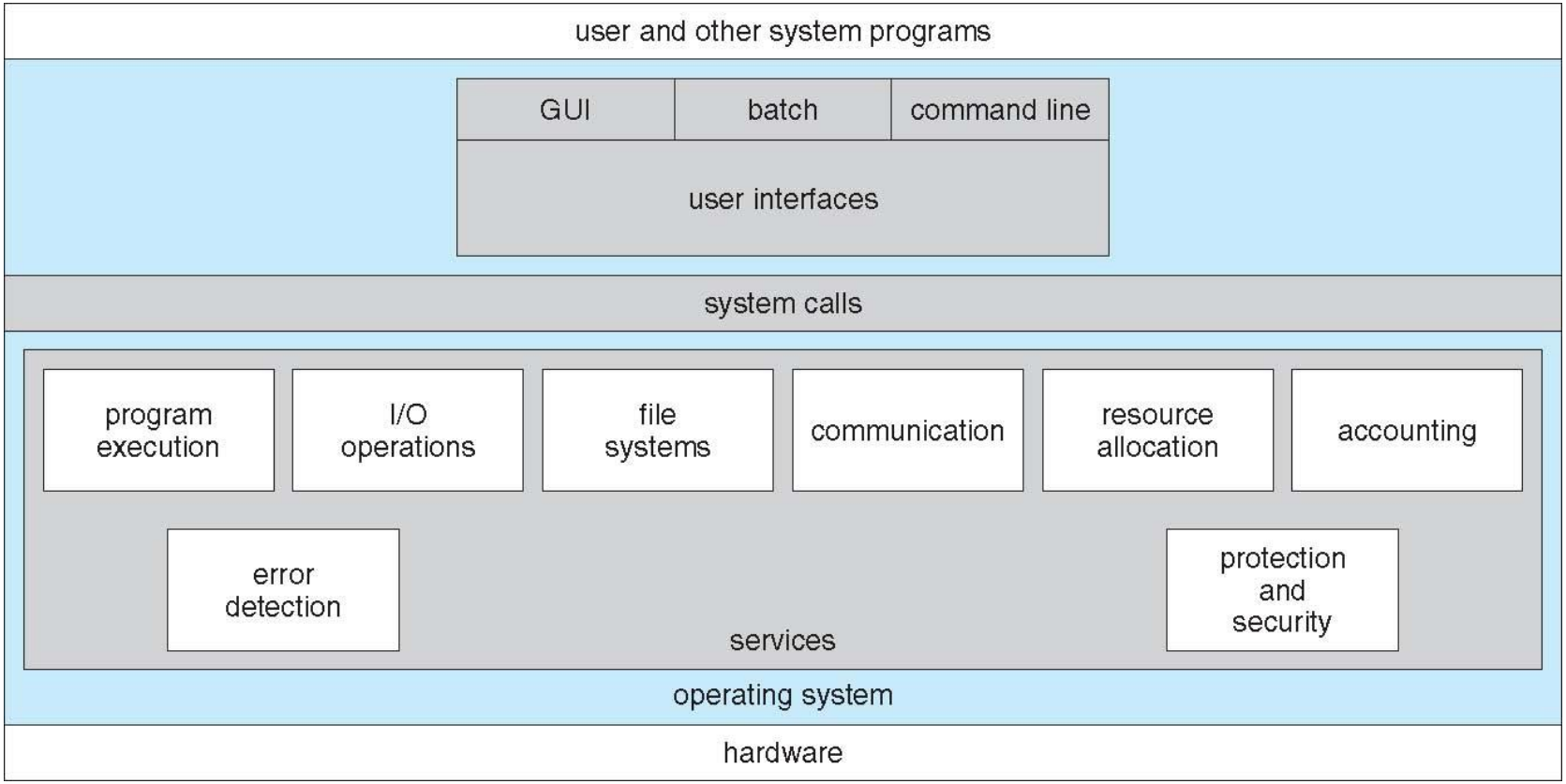


Operating System Services

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing:
  - **Resource allocation:** When multiple users or multiple jobs running concurrently, resources must be allocated to each one of them
    - Many types of resources: CPU cycles, main memory, file storage, I/O devices.
  - **Accounting:** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security:** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** controlling the access to system resources
    - **Security** securing the system from outsiders requires user authentication, and defending external I/O devices from invalid access attempts



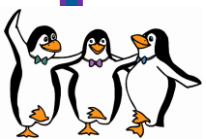
Operating System Services



A view of operating system services

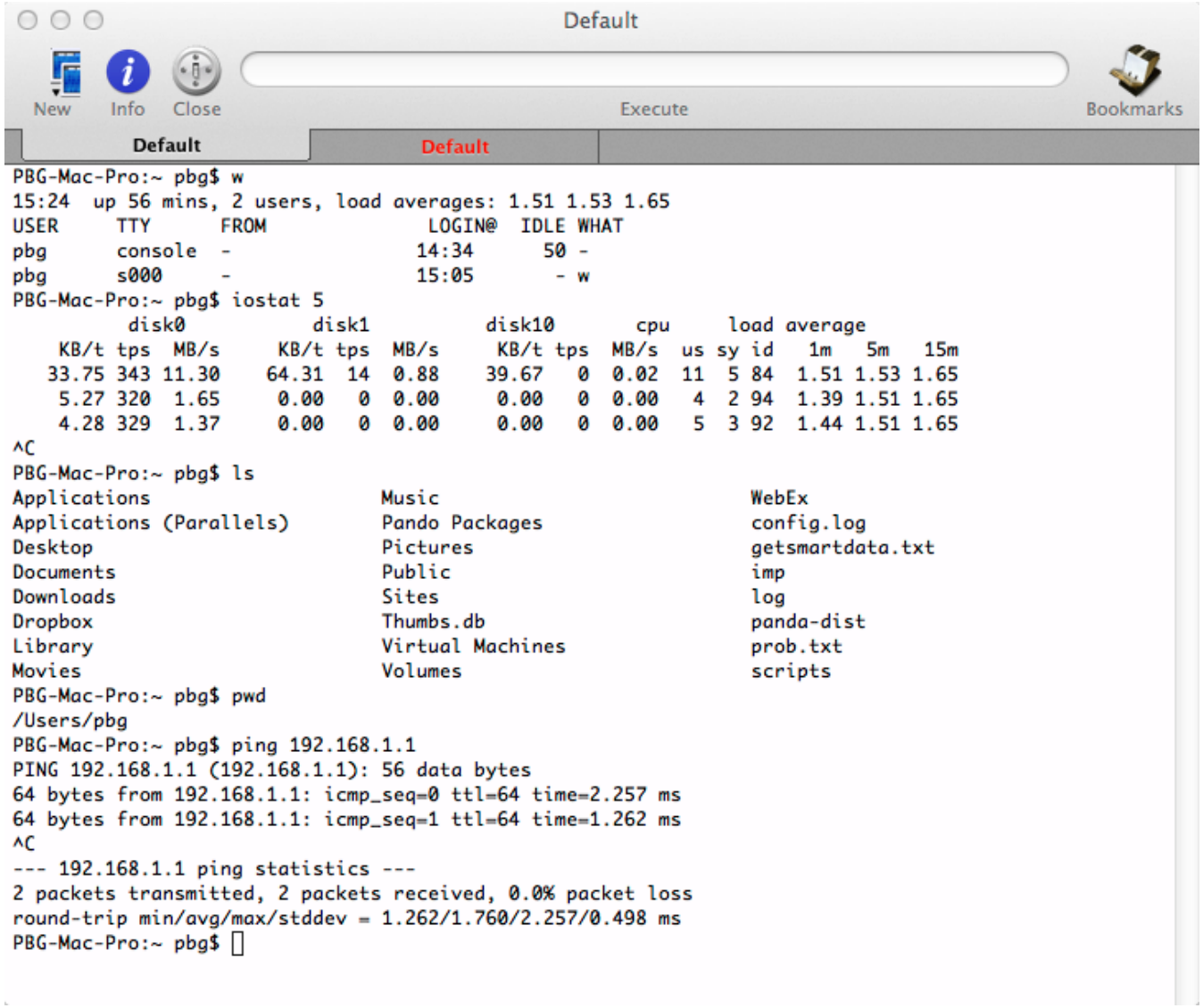
User and Operating System Interface

- Command interpreters
  - CLI or **command interpreter** allows direct command entry
    - Sometimes implemented in kernel, sometimes by system program
    - Sometimes multiple flavors implemented called **shells**
    - Primarily fetches a command from user and executes it
    - Sometimes commands built-in, sometimes just names of programs
      - If the latter, adding new commands doesn't require shell modification
    - In UNIX and Linux systems, there are different shells to choose from:
      - Bourne shell,
      - C shell,
      - Bourne-Again shell,
      - Korn shell,



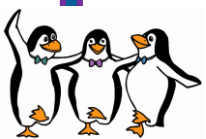
User and Operating System Interface

- The Bourne Shell Command Interpreter in Solaris 10.



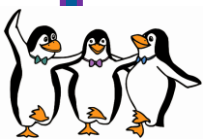
User and Operating System Interface

- Graphical User Interfaces (GUI)
  - User-friendly **desktop** metaphor interface
    - Usually mouse, keyboard, and monitor
    - **Icons** represent files, programs, actions, etc
    - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
    - Invented at Xerox PARC in 1973
  - 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
    - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)



User and Operating System Interface

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



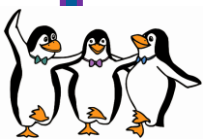
User and Operating System Interface

- The Mac OS X (which is in part implemented using a UNIX kernel), provides both a Aqua interface and a command-line interface.

System Calls

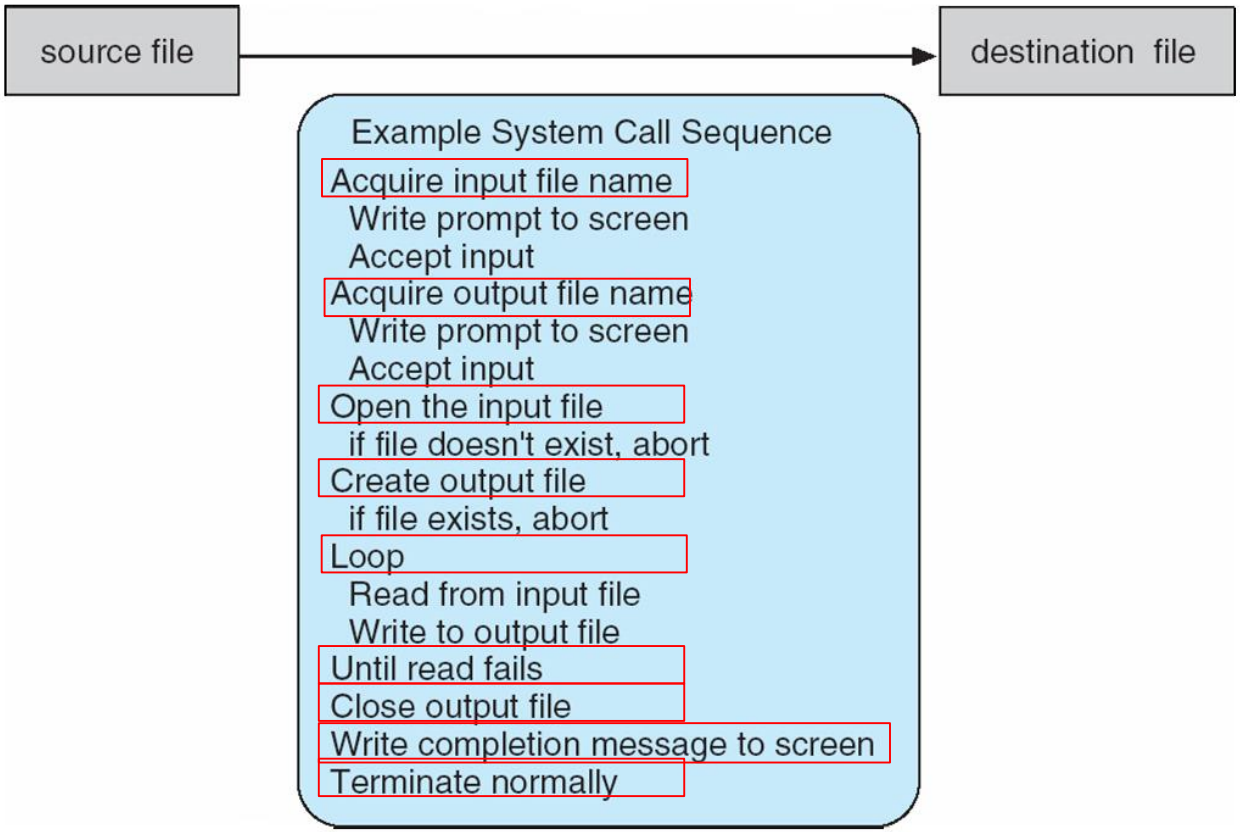
System Calls

- They provide an interface to the services made available by an operating system.
- Typically written in a high-level language (C or C++), some by assmbly
- Mostly accessed by programmers (or programs) via a high-level **Application Programming Interface (API)** rather than direct system call use
  - Programmers access the API through a library (called libc in UNIX/Linux)
- Three most common APIs are:
  - Win32 API for Windows,
  - POSIX API for POSIX-based systems (includes most versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)
- *Note that the system-call names used throughout this course are **generic***



System Calls

- Example:
  - System calls sequence to copy the contents of one file to another file



System Calls

- **Example of Standard API**

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:

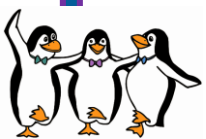
```
#include <unistd.h>

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

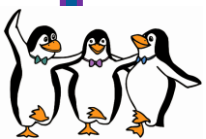
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.



System Calls

System Call Implementation

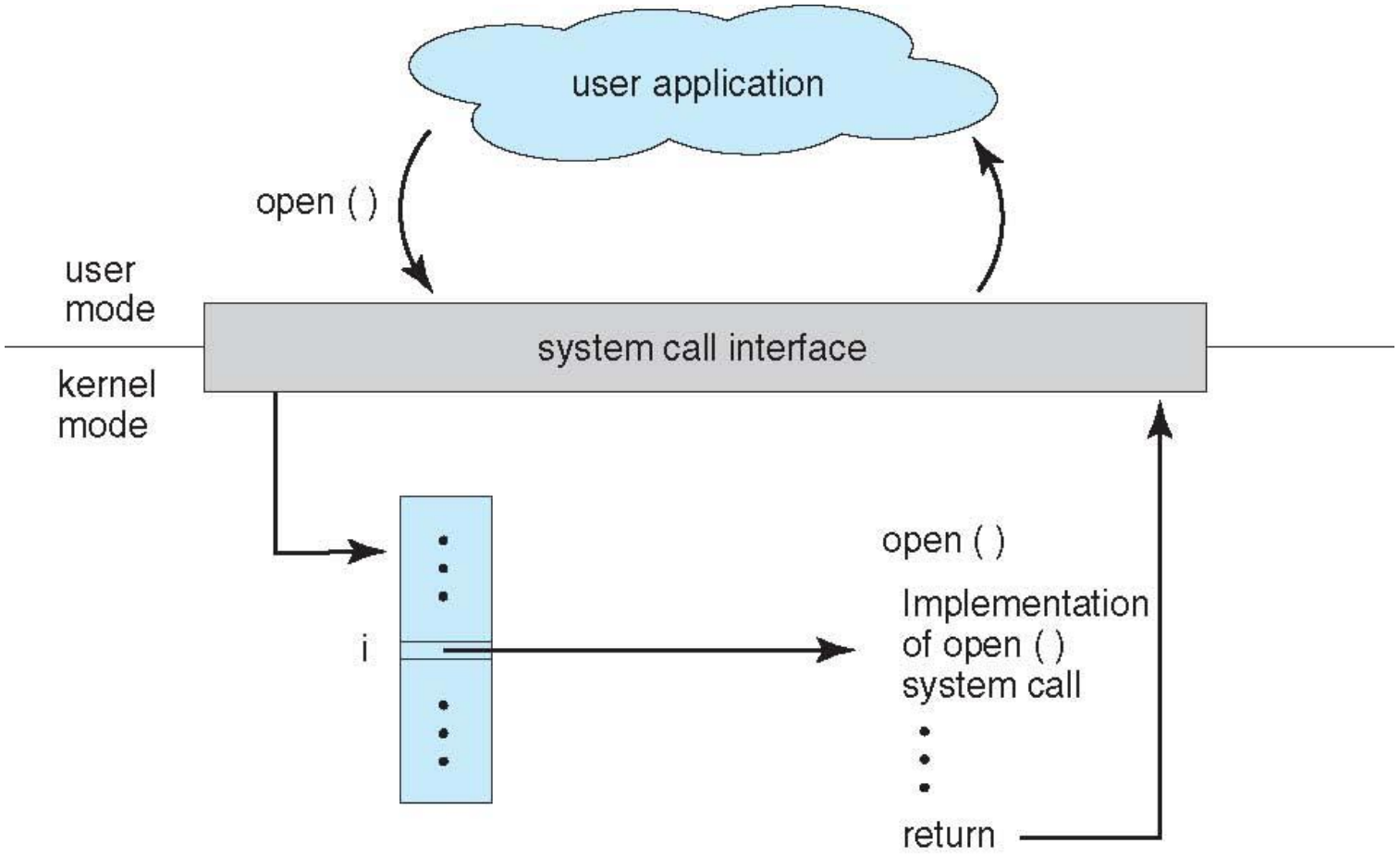
- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
  - Example: the Windows function CreateProcess() actually invokes the NTCreateProcess() system call in the Windows kernel.
- The caller need to 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
    - Managed by run-time support library (set of functions built into libraries included with compiler)



System Calls

- The figure shows the relationship between an API, the system-call interface, and the OS.

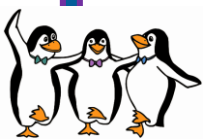
The handling of a user application invoking the open() system call



System Calls

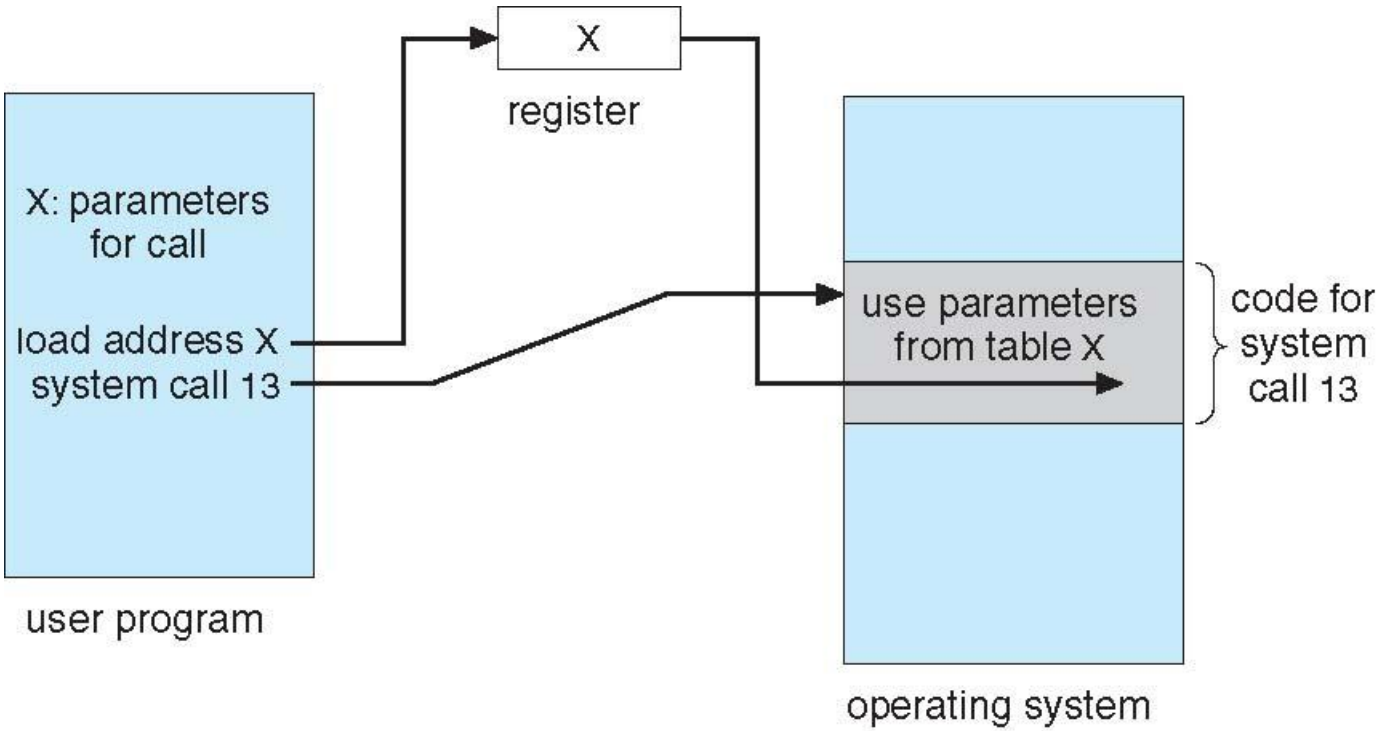
System Call Parameter Passing

- Often, more information is required than simply the identity of the desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Passing the parameters in registers
    - It is the simplest but it may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - Such as in 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



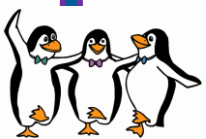
System Calls

- Parameter Passing via Table



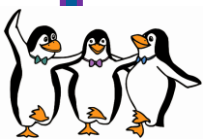
Types of System Calls

- System calls can be grouped into six major categories:
  1. Process control,
  2. File manipulation,
  3. Device manipulation,
  4. Information maintenance,
  5. Communications,
  6. Protection.



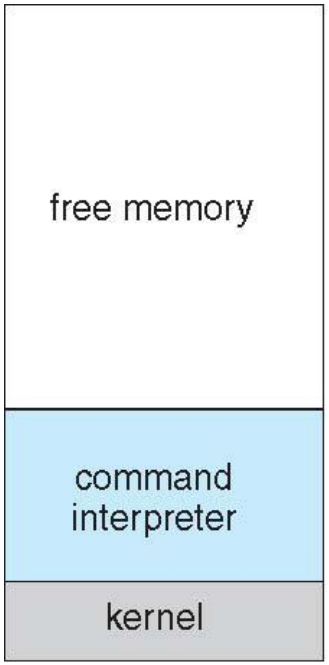
Types of System Calls

1. Process control system calls
   - create process, terminate process
   - end, abort
   - load, execute
   - 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
     - Such as acquire_lock() and release_lock()



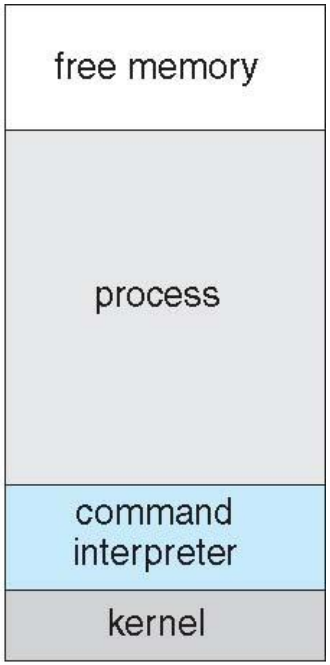
Types of System Calls

- Example: MS-DOS
  - It is single-tasking system
  - Shell (command interpreter) invoked when system booted
  - It uses a simple method to run program and does not create a new process
  - It uses a single memory space
  - Loads program into memory, overwriting itself but not the kernel
  - When program exit, the remaining from the shell reloads back the rest from the hard disk and then ready to run new program



(a)

At system startup

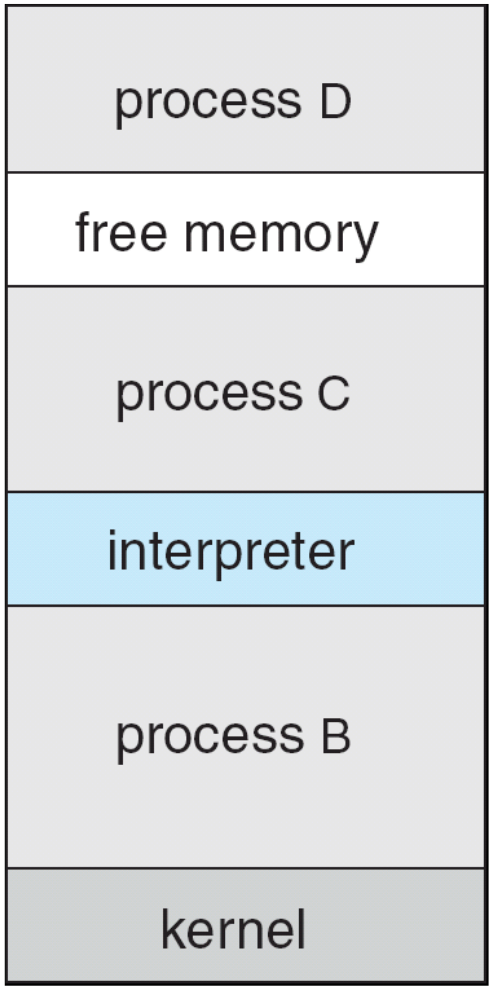


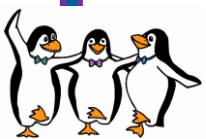
(b)

running a program

Types of System Calls

- Example: FreeBSD (**B**erkeley **S**oftware **D**istribution)
  - It is Unix variant
  - It is a multitasking system
  - When user login, the system invokes user's choice of shell (many shells to choose from)
  - To start a new process, the shell executes a *fork()* system call
  - Then it loads the selected program into memory by executing the *exec()* system call, and the program is executed
  - Then Shell waits for process to terminate or continues with other user commands
  - When a process is done, it executes *exit()* system call to terminate and returns either:
    - code = 0 → no error
    - code > 0 → error code

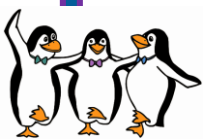




Types of System Calls

2. File management
   - create file, delete file
   - open, close file
   - read, write, reposition, move, copy
   - get and set file attributes
     - File name, file type, protection codes, accounting information
   - Operations for directories

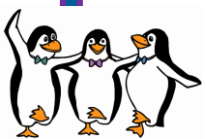
3. Device management:
   - A process may need several resources to execute such as main memory, disk drives, access to files, …
     - request device, release device
     - read, write, reposition
     - get device attributes, set device attributes
     - logically attach or detach devices



Types of System Calls

4. Information maintenance

- They are needed for transferring information between the user program and the operating system
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes



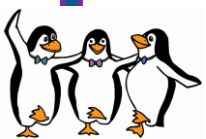
Types of System Calls

5. Communications

- create, delete communication connection
- transfer status information
- attach and detach remote devices
- There are two common models of interprocess communication:
  - The **message-passing model**: for exchanging smaller amounts of data
    - send, receive messages to host name or process name
      - From client to server
  - The **shared-memory model**: for intercomputer communication
    - create and gain access to memory regions

6. Protection

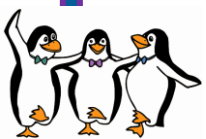
- To control access to the resources
  - Get and set permissions
  - Allow and deny user access



Types of System Calls

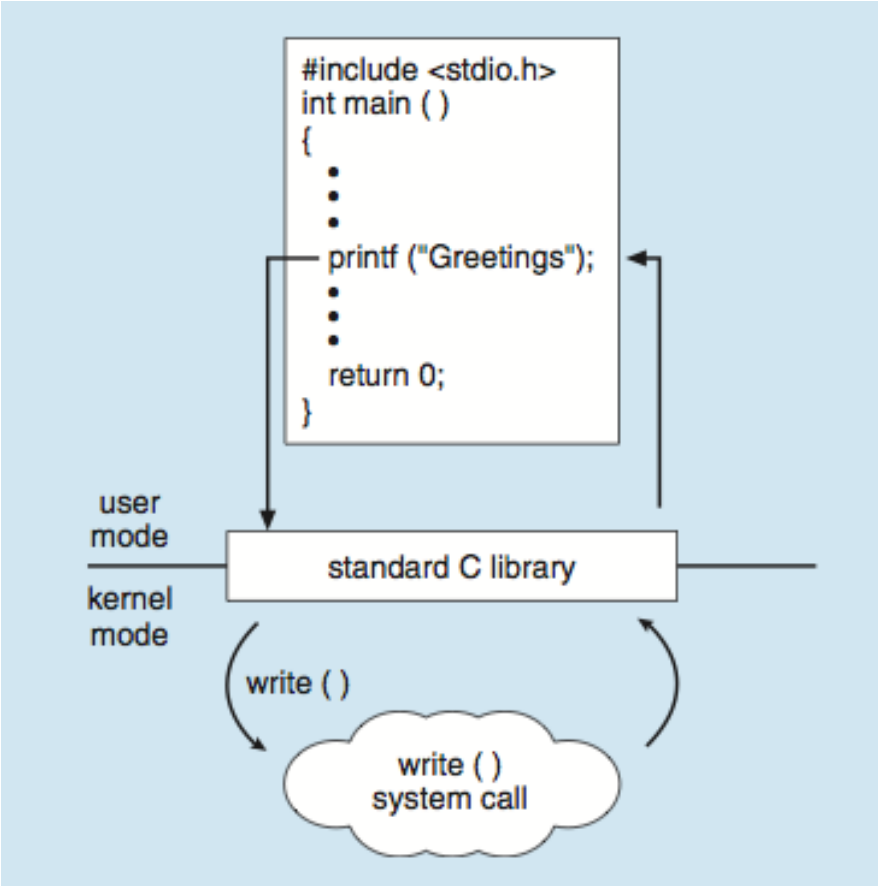
- Examples of Windows and Unix System Calls

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



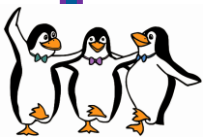
Types of System Calls

- Example of Standard C Library
  - C program invoking ***printf()*** library call, which calls ***write()*** system call



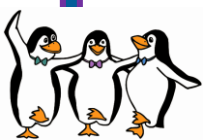
System Programs

- System programs (system utilities) provide a convenient environment for program development and execution.
- Some of system programs are user interfaces to system calls while others are more complex programs.
- **They can be divided into:**
  - File management
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs
- Most users' view of the operating system is defined by the application and system programs, rather than by the actual system calls.



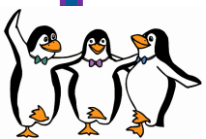
System Programs

- **File management**
  - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some programs that ask the system for info about date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - These programs are typically format and print the output to the terminal or other output devices
  - Some systems implement a **registry** which is used to store and retrieve configuration information
- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text



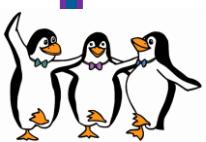
System Programs

- **Programming-language support**
  - Compilers, assemblers, debuggers and interpreters sometimes provided for common programming languages (such as C, C++, Java, and PERL)
- **Program loading and execution**
  - The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications**
  - Provide the mechanism for creating virtual connections among processes, users, and computer systems
    - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another



System Programs

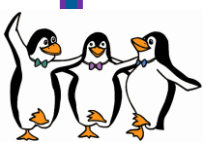
- **Background Services**
  - Launched at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown (called daemons or services)
      - Provide facilities like disk checking, process scheduling, error logging, printing, …
      - Run in user context not in kernel context
      - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke
  - Such as Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.



Operating System Design and Implementation

- Specifying and designing an OS is a highly creative task of **software engineering,**
- **Principles for designing an OS are:**

1. **Design goals:**
   - Internal structure of different Operating Systems can vary widely
   - Start the design by defining the goals and specifications
     - Which are affected by choice of hardware, and type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose
   - The requirements are hard to define but they can be divided into:
     - User goals:
       - The OS should be convenient to use, easy to learn, reliable, safe, and fast
     - Designers goals:
       - The OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

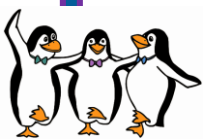


Operating System Design and Implementation

2. Separation of mechanisms and policies

- **Policy:** ***What*** will be done?
- **Mechanism:** ***How*** to do it?

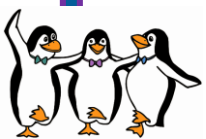
- The separation of policy from mechanism is a very important principle, since it allows maximum flflexibility if policy decisions are to be changed later
  - Example:
    - The timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.



Operating System Design and Implementation

3. Implementation

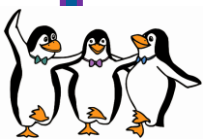
- Much variation
  - Early OSes are implemented in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually we use a mix of languages
  - Lowest levels are written in assembly
  - Main body is written in C
  - Systems programs are implemented in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language is easier to **port** to other hardware
  - But it is slower and increases the storage requirements
- **Emulation** can allow an OS to run on non-native hardware
- After the OS been implemented correctly, bottleneck routines can be identified and replaced with assembly-language equivalents.



Operating System Structure

- General-purpose OS is a very large program that must be engineered carefully
- A common approach is to partition the task into small modules, rather than have one monolithic system

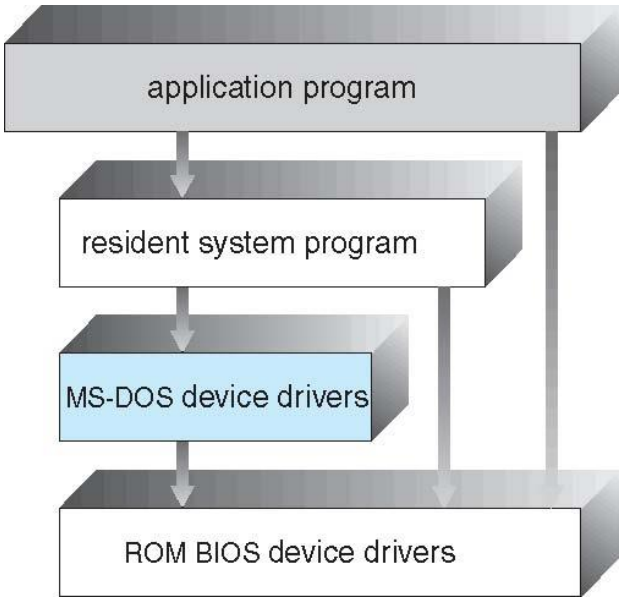
- **There are various ways to structure the OS:**
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstraction
  - Microkernel -Mach



Operating System Structure

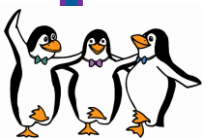
Simple Structure -- MS-DOS

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
    - Application programs are able to access the basic I/O routines to write directly to the display and disk drives
      - This makes the system more vulnerable to malicious programs, and crashes



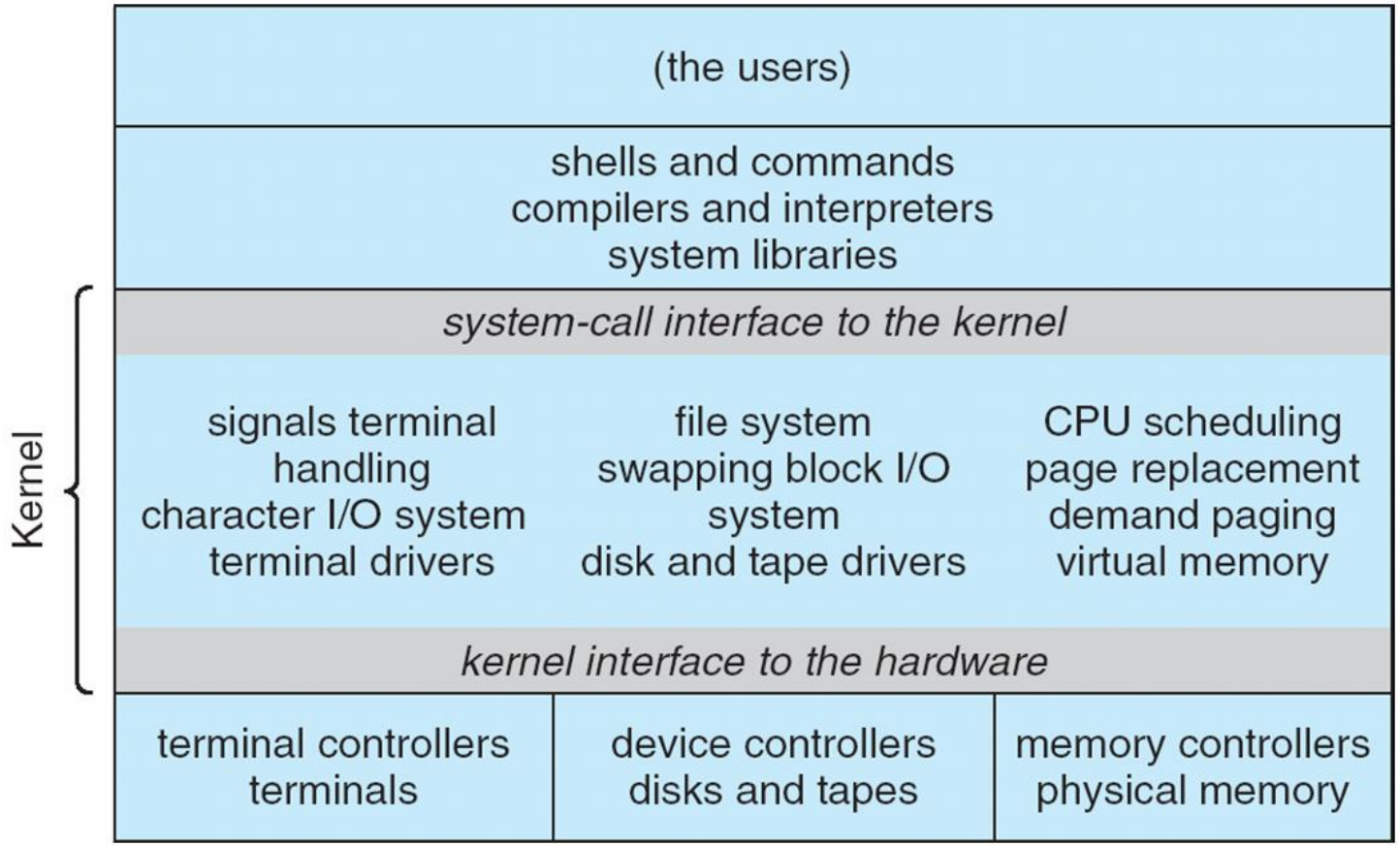
Operating System Structure

- Traditional UNIX
  - Not a simple structure and not fully layered
  - The original UNIX is limited by hardware functionality, the original UNIX operating system had limited structuring.
  - The UNIX OS consists of two separable parts:
    - System 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



Operating System Structure

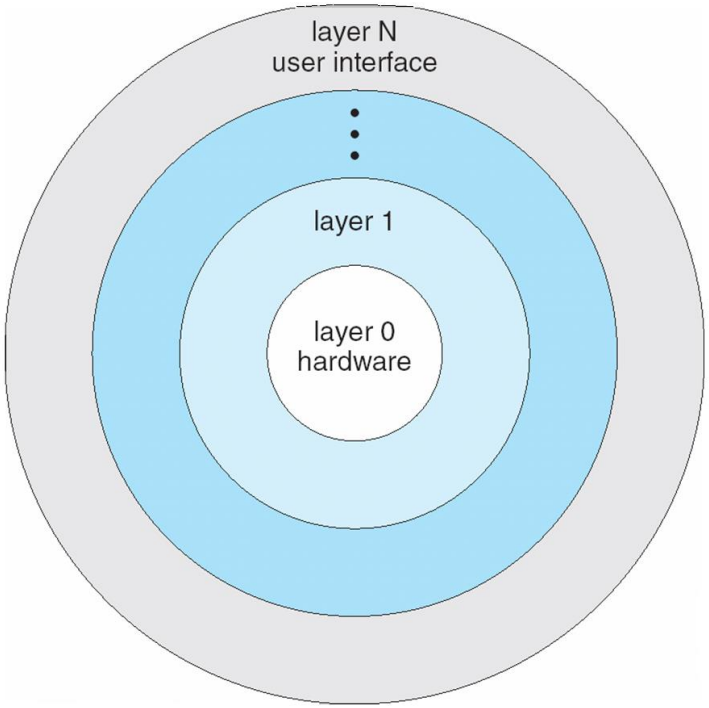
- Traditional UNIX System Structure
  - Beyond simple but not fully layered



Operating System Structure

Layered Approach

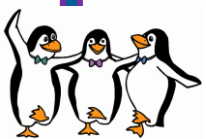
- The operating system is divided into a number of layers (levels), each built on top of lower layers.
  - The bottom layer (layer 0), is the hardware;
  - The highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only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



Operating System Structure

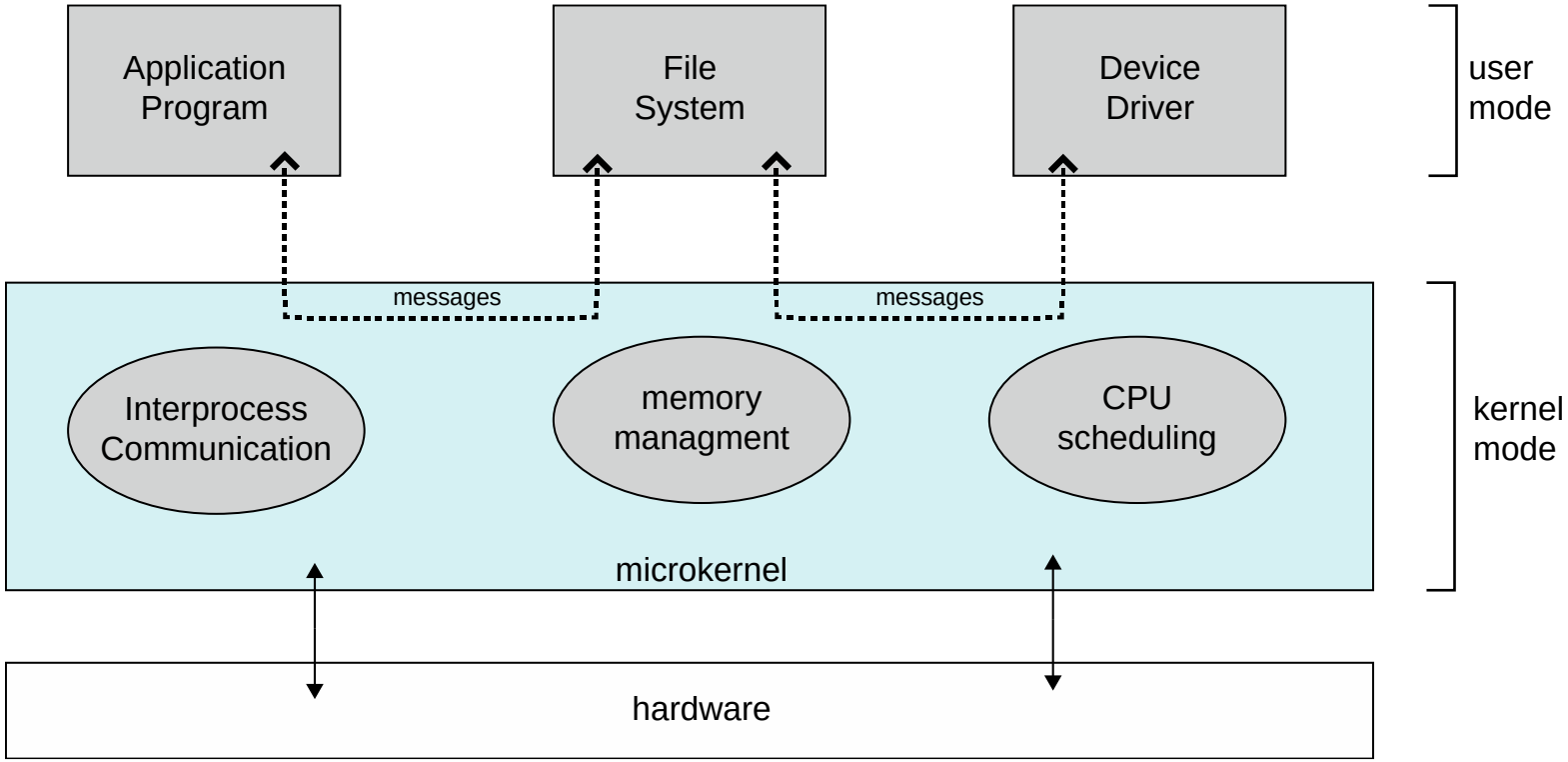
Microkernel System Structure

- Moves all nonessential components from the kernel and implementing them as system and user-level programs.
- **Mach** is an example of **microkernel** developed by Carnegie Mellon Univ.
  - Mac OS X kernel (**Darwin**) is partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel by adding new features to the user space
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure and reliable
- Disadvantage:
  - Performance overhead of user space to kernel space communication



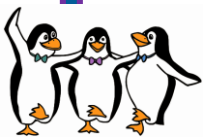
Operating System Structure

- Microkernel System Structure



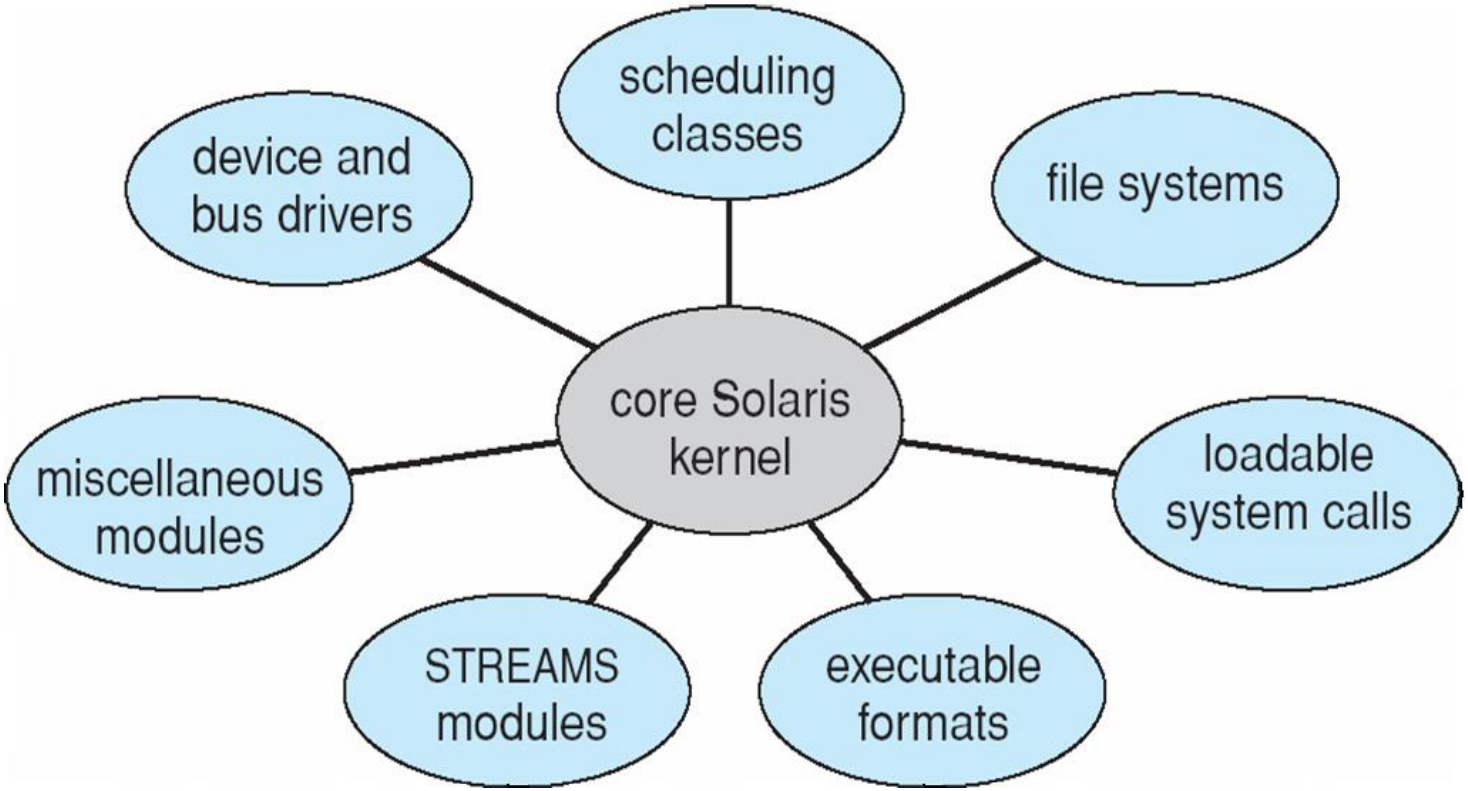
Operating System Structure

- Modules
  - Many modern operating systems implement **loadable kernel modules**
    - Uses object-oriented approach
    - Each core component is separate
    - Each talks to the others over known interfaces
    - Each is loadable as needed within the kernel
  - Overall, similar to layers but with more flexibility
    - Linux, Solaris, Mac OS X, Windows, etc



Operating System Structure

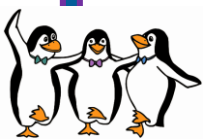
- Modules
  - Solaris Modular Approach
    - It is organized around a core kernel with seven types of loadable kernel modules:



Operating System Structure

Hybrid Systems

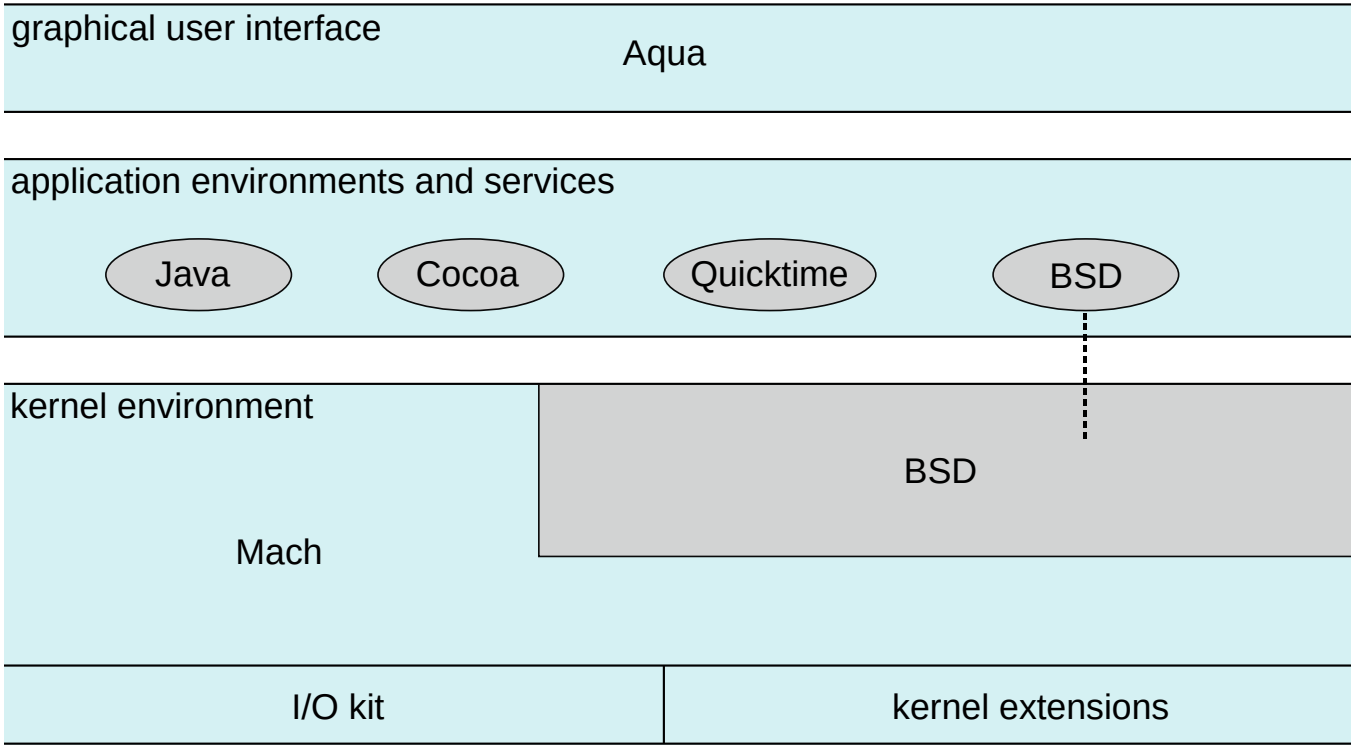
- Most modern operating systems are actually not one pure model
- But they are hybrid combines multiple approaches to address performance, security, and usability needs
- **Examples**:
  - Linux and Solaris kernels are monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem known as ***personalities***



Operating System Structure

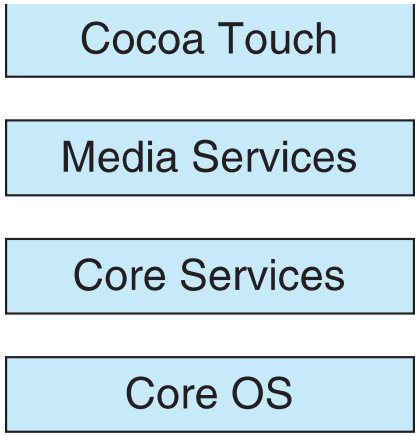
Hybrid Systems

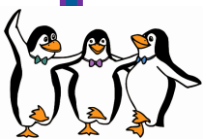
- Apple Mac OS X is hybrid, layered, uses **Aqua** UI plus **Cocoa** programming environment
  - The kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)
  - Mac OS X Structure



Operating System Structure

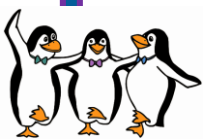
- Hybrid Systems
  - **iOS:** Apple mobile OS for ***iPhone***, ***iPad***
    - Structured on Mac OS X, with added functionality for mobile devices
      - But it does not run OS X applications natively
      - Also runs on different CPU architecture (ARM vs. Intel)
    - **Cocoa Touch** it is an API for Objective-C for developing apps that suits the mobile devices with touch screan
    - **Media services** layer for graphics, audio, video
    - **Core services** provides support for cloud computing, databases
    - **Core operating system**, is based on Mac OS X kernel





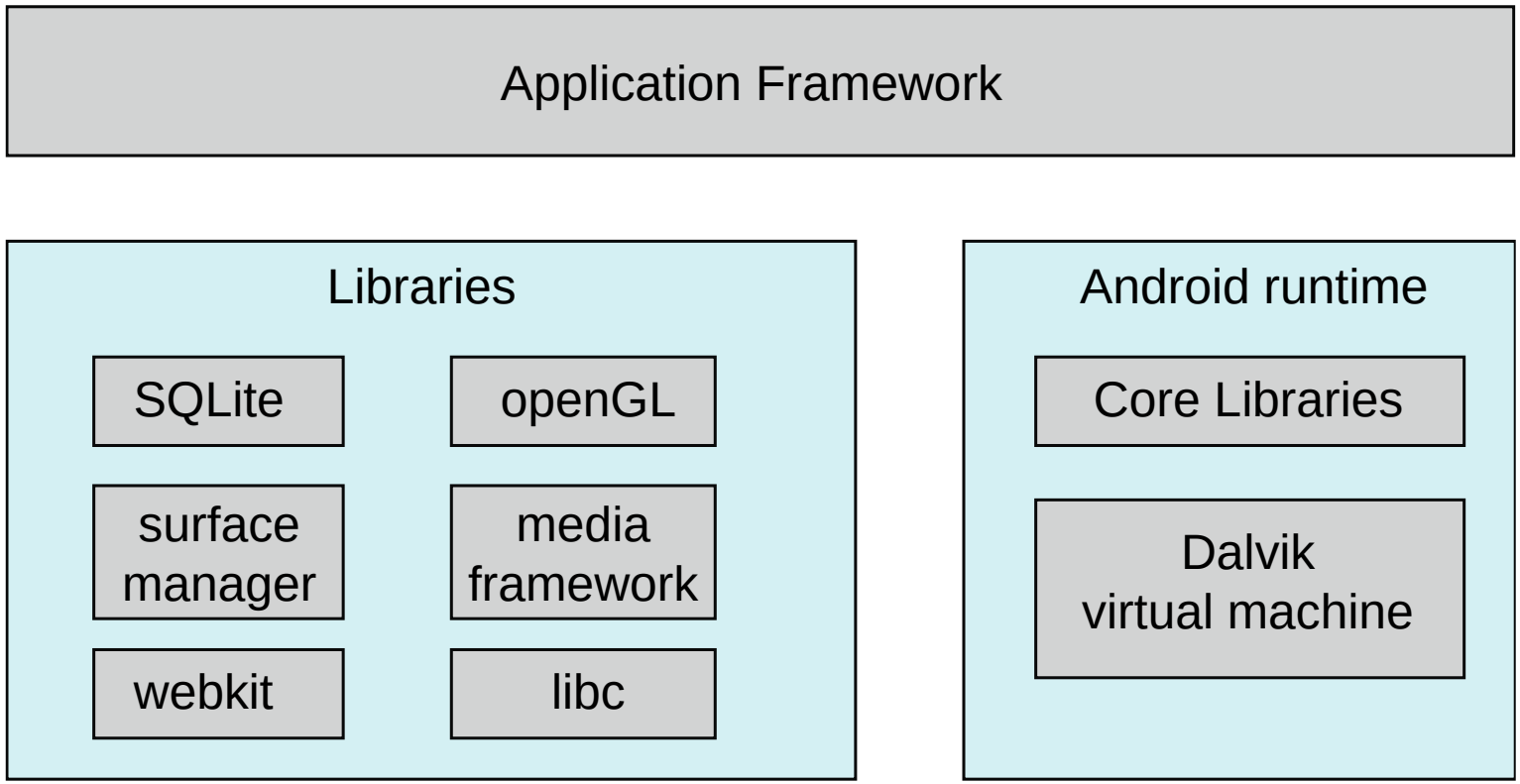
Operating System Structure

- Hybrid Systems
  - **Android:** Developed by Open Handset Alliance (led primarily by Google)
    - It is an open Source
    - It has similar to iOS in its layered structure
    - It is based on Linux kernel but modified to
      - Provide process, memory, device-driver management, and power management
    - The Android runtime environment includes core set of libraries and Dalvik virtual machine
      - Apps developed in Java plus Android API
        - Java class files compiled to Java bytecode then translated to executable then runs in Dalvik VM
    - Libraries include frameworks for web browser (webkit), database support (SQLite), multimedia



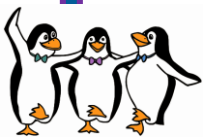
Operating System Structure

- Hybrid Systems
  - Android structure



Operating-System Debugging

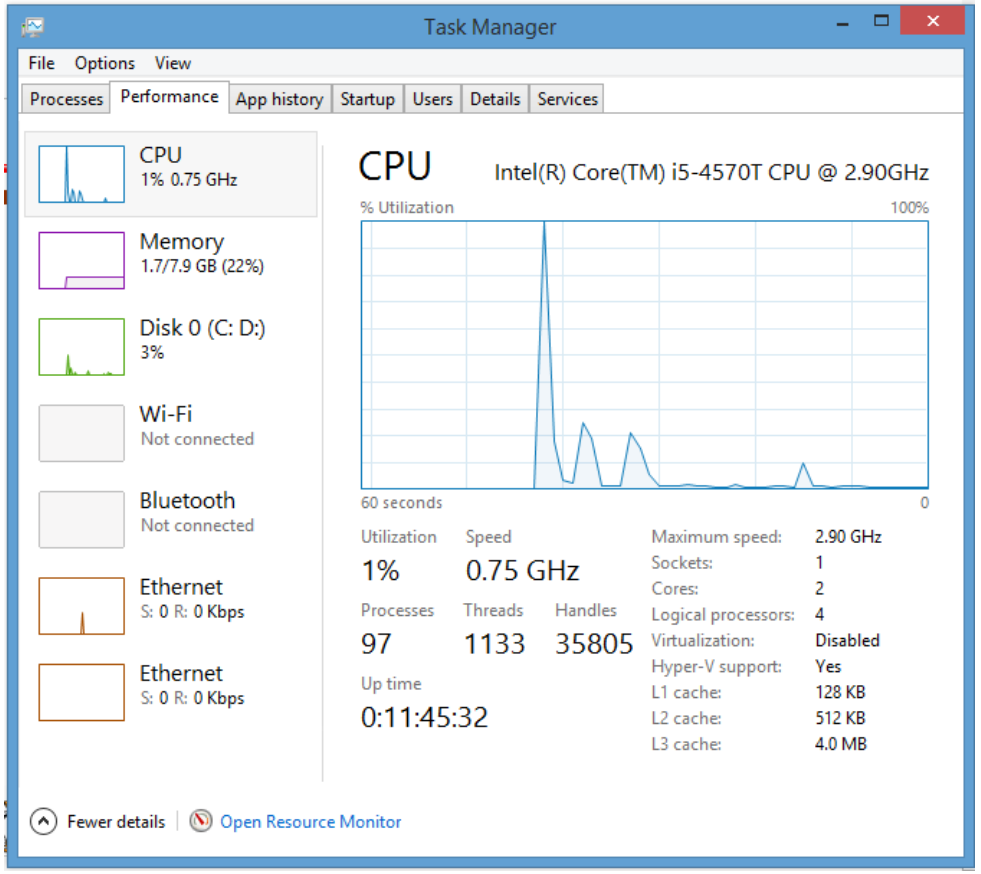
- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends
- 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.*"

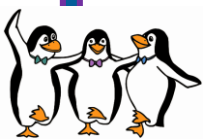


Operating-System Debugging

Performance Tuning

- Improve performance by removing the bottlenecks
- OS provides means of computing and displaying measures of system behavior
- For example,
  - "top" is a unix command to display the resources used on the system, as well as a sorted list of the "top" resource-using processes.
  - Windows Task Manager includes information about current applications and processes, CPU and memory usage, and networking statistics.



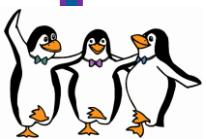


Operating-System Debugging

Dtrace tool

- DTrace is a tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- DTrace is a facility that dynamically adds probes to a running system, both in user processes and in the kernel.
- **Probes** fire when code is executed within a **provider**, capturing state data and sending it to **consumers** of those probes
- Example of following XEventsQueued system call move from libc library to kernel and back

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                       U
  0   -> _XEventsQueued                    U
  0     -> _X11TransBytesReadable          U
  0     <- _X11TransBytesReadable          U
  0     -> _X11TransSocketBytesReadable    U
  0     <- _X11TransSocketBytesreadable    U
  0     -> ioctl                           U
  0       -> ioctl                         K
  0         -> getf                        K
  0           -> set_active_fd             K
  0           <- set_active_fd             K
  0         <- getf                        K
  0         -> get_udatamodel              K
  0         <- get_udatamodel              K
...
  0         -> releasef                    K
  0           -> clear_active_fd           K
  0           <- clear_active_fd           K
  0           -> cv_broadcast              K
  0           <- cv_broadcast              K
  0         <- releasef                    K
  0       <- ioctl                         K
  0     <- ioctl                           U
  0   <- _XEventsQueued                    U
  0 <- XEventsQueued                       U
```



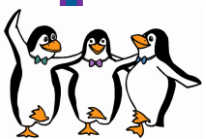
Operating-System Debugging

- Dtrace
  - DTrace code (as a D language code) to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds

```
sched:::on-cpu
uid == 101
{
   self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
   @time[execname] = sum(timestamp - self->ts);
   self->ts = 0;
}
```

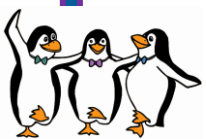
```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
        gnome-settings-d              142354
        gnome-vfs-daemon              158243
        dsdm                          189804
        wnck-applet                   200030
        gnome-panel                   277864
        clock-applet                  374916
        mapping-daemon                385475
        xscreensaver                  514177
        metacity                      539281
        Xorg                         2579646
        gnome-terminal               5007269
        mixer_applet2                7388447
        java                        10769137
```

Figure 2.21 Output of the D code.



Operating System Generation

- Operating systems are designed to run on any of a class of machines;
- Therefore, the system must be configured for each specific computer site
- **System generation SYSGEN** is a program that obtains information concerning the specific configuration of the hardware system
  - This SYSGEN program determines what components are in the computer either (reads from a file, or asks the operator about the hardware system, or probes the hardware directly).
  - The following kinds of information must be determined:
    - What CPU is to be used?
    - How will the boot disk be formatted?
    - How much memory is available?
    - What devices are available?
    - What operating-system options are desired, or what parameter values are to be used? how many buffers, what CPU-scheduling algorithm, what the maximum number of processes to be supported is, ...



System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, 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, which will load the kernel
- Common bootstrap loader for Linux, **GRUB** (GNU GRand Unified Bootloader), allows selection of kernel from multiple disks, versions, kernel options
- When the Kernel is loaded, the system is said to be **running**