Chapter 2: Operating-System Structures

Chapter 2: Operating-System Structures

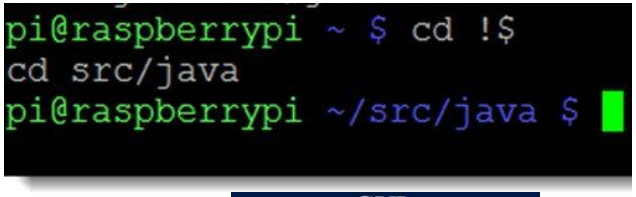
- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot

Objectives

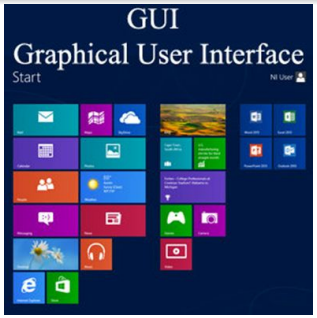
- operating system services
- ways of structuring an operating system
- operating systems installation ,customization and boot process

Operating System Services

- **User interface :**
  - **Command-Line (CLI),**



  - **Graphics User Interface (GUI),**



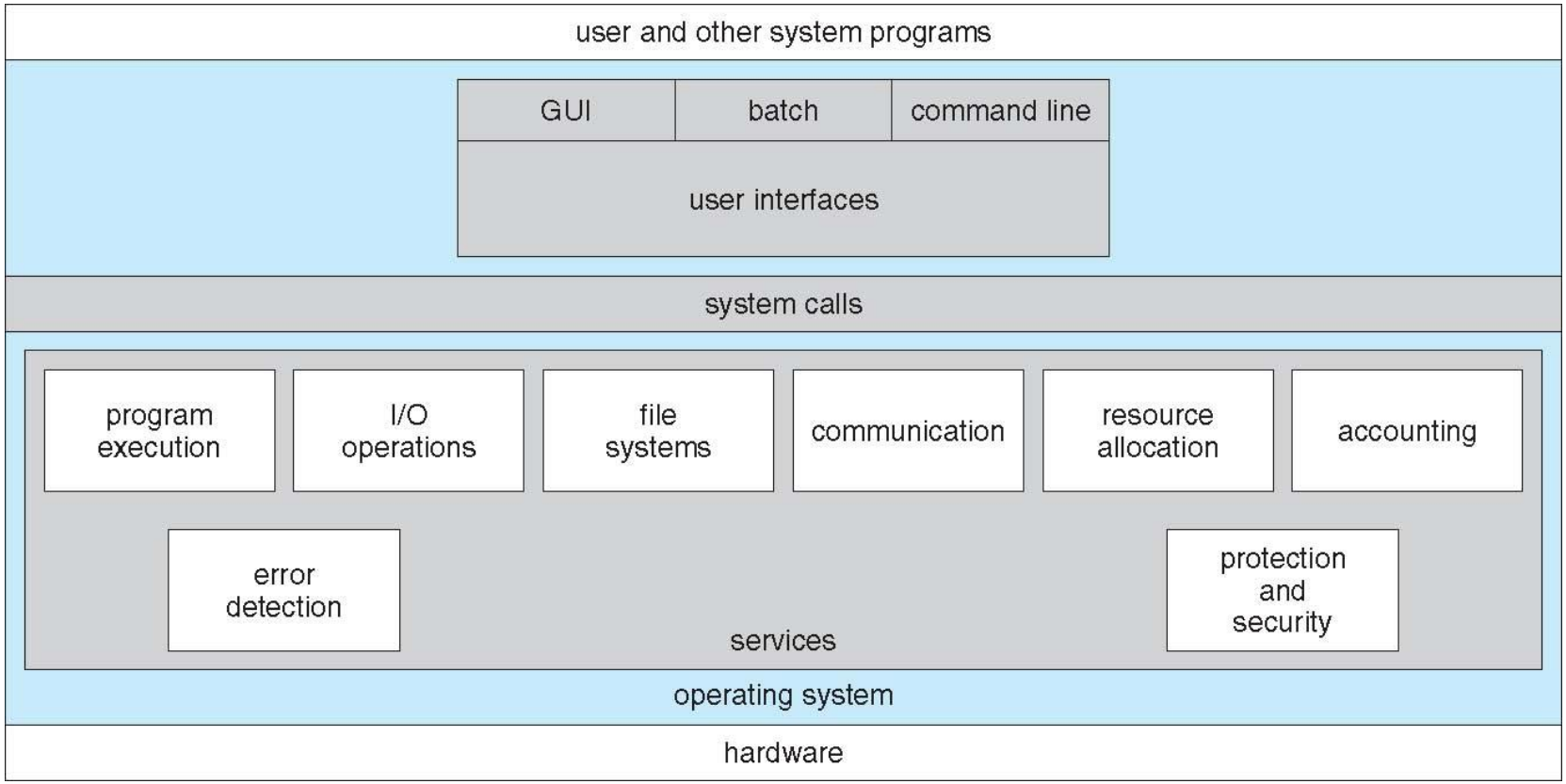
  - **Batch**

- **Program execution**
- **I/O operations**

Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **File-system**
  - **Communications**
    - Processes to process(shared memory or network)
    - via shared memory or through message passing
  - **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

A View of Operating System Services



User Operating System Interface - CLI

CLI or **command interpreter** allows direct command entry, it might be:

- implemented in kernel or by systems program
- multiple flavors implemented – **shells**

- Primarily fetches a command from user to execute it, might be:
  - built-in
  - names of programs: adding new features doesn't require shell modification

Bourne Again Shell Command Interpreter

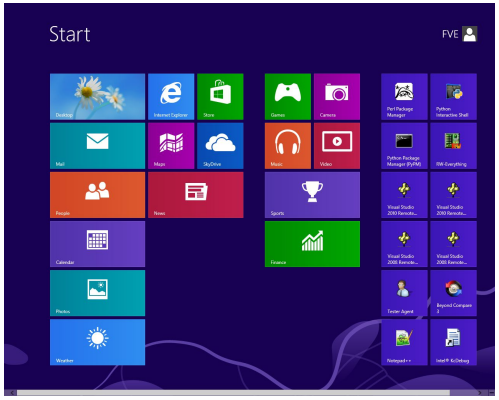
```
venkatesh@DellPC:~/bin$ cat test
#!/bin/bash
#comment
echo "hello world"
venkatesh@DellPC:~/bin$ cat test.sh
#!/bin/bash
#comment
echo "hello world"
```

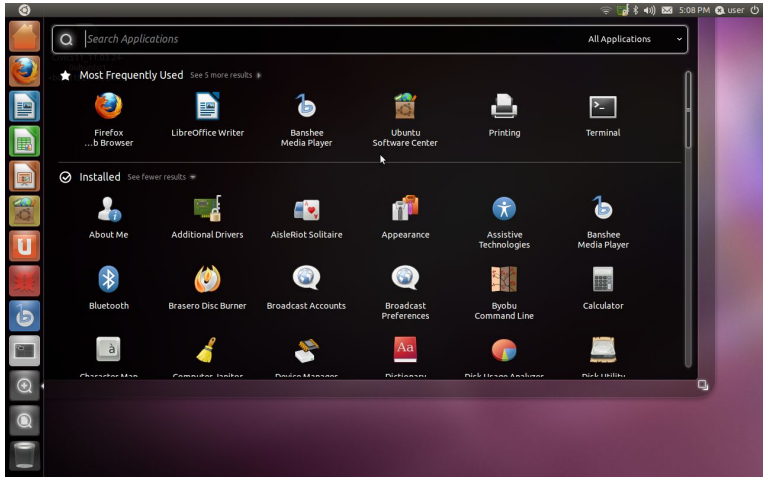
User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons**
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)

systems with 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)





Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - No Mouse required
  - Actions and selection based on gestures
  - Virtual keyboard
  - Voice commands.

The Mac OS X GUI

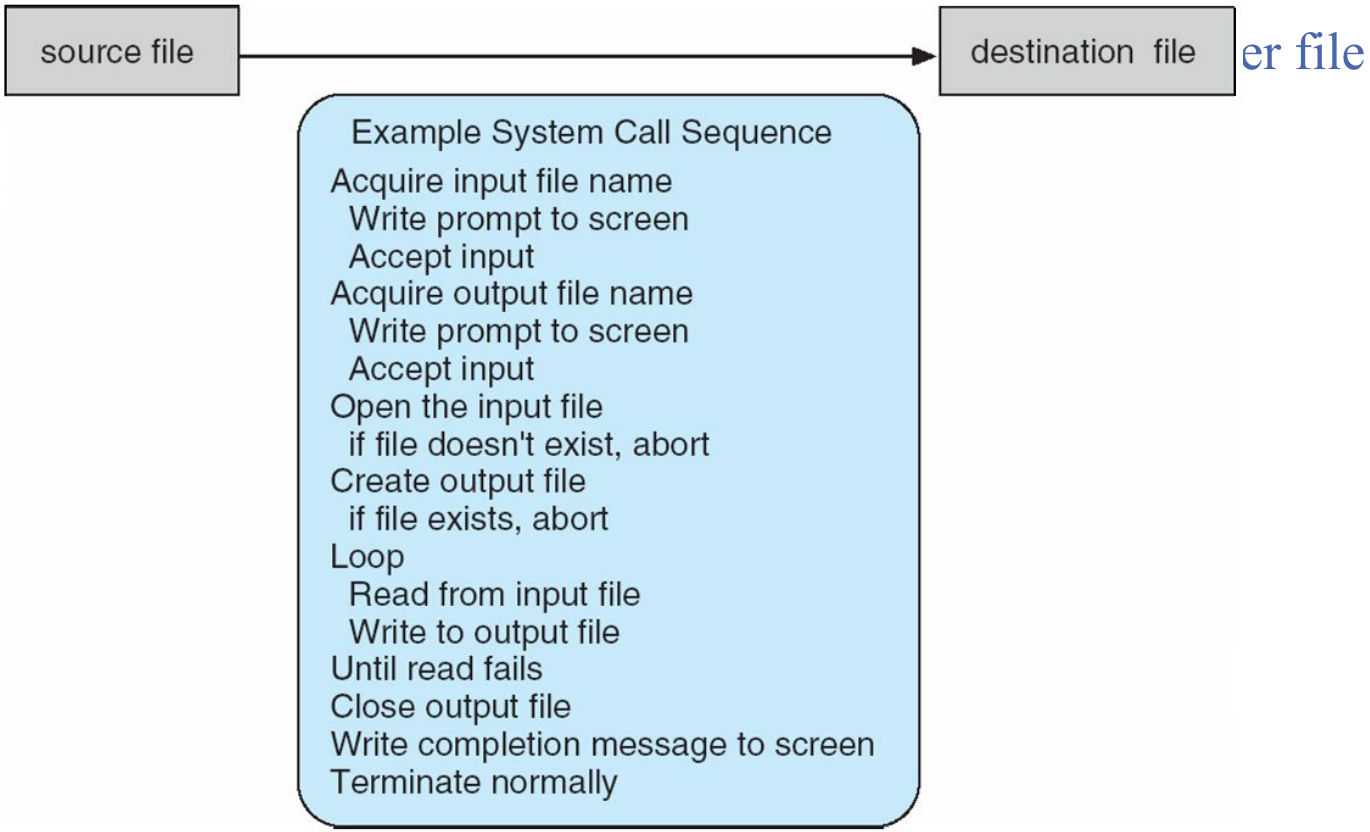
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Note that the system-call names used throughout this text are generic

Example of System Calls

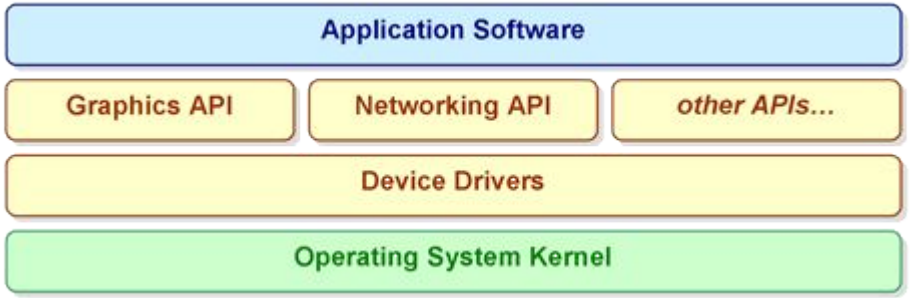
- Sys



er file

API

- The **Application Programming Interface** (API) is the "public" part of the operating system, as far as other applications are concerned. The kernel and device drivers facilitate access to the hardware, and the API software uses that access to provide a "higher-level" method for doing useful things.



Device drivers are special pieces of software which allow the rest of the operating system to interface with specific pieces of computer hardware in an easy-to-use manner. For example, graphics cards, printers, network adapters

Examples of APIs

OpenCL cross-platform API for general-purpose computing for CPUs & GPUs

OpenGL cross-platform graphics API

OpenMP API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on many architectures, including Unix and Microsoft Windows platforms.

Server Application Programming Interface (SAPI)

Simple DirectMedia Layer (SDL)

- ASPI for SCSI device interfacing
- Cocoa and Carbon for the Macintosh
- DirectX for Microsoft Windows
- EHLLAPI
- Java APIs
- ODBC for Microsoft Windows
- OpenAL cross-platform sound API

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:

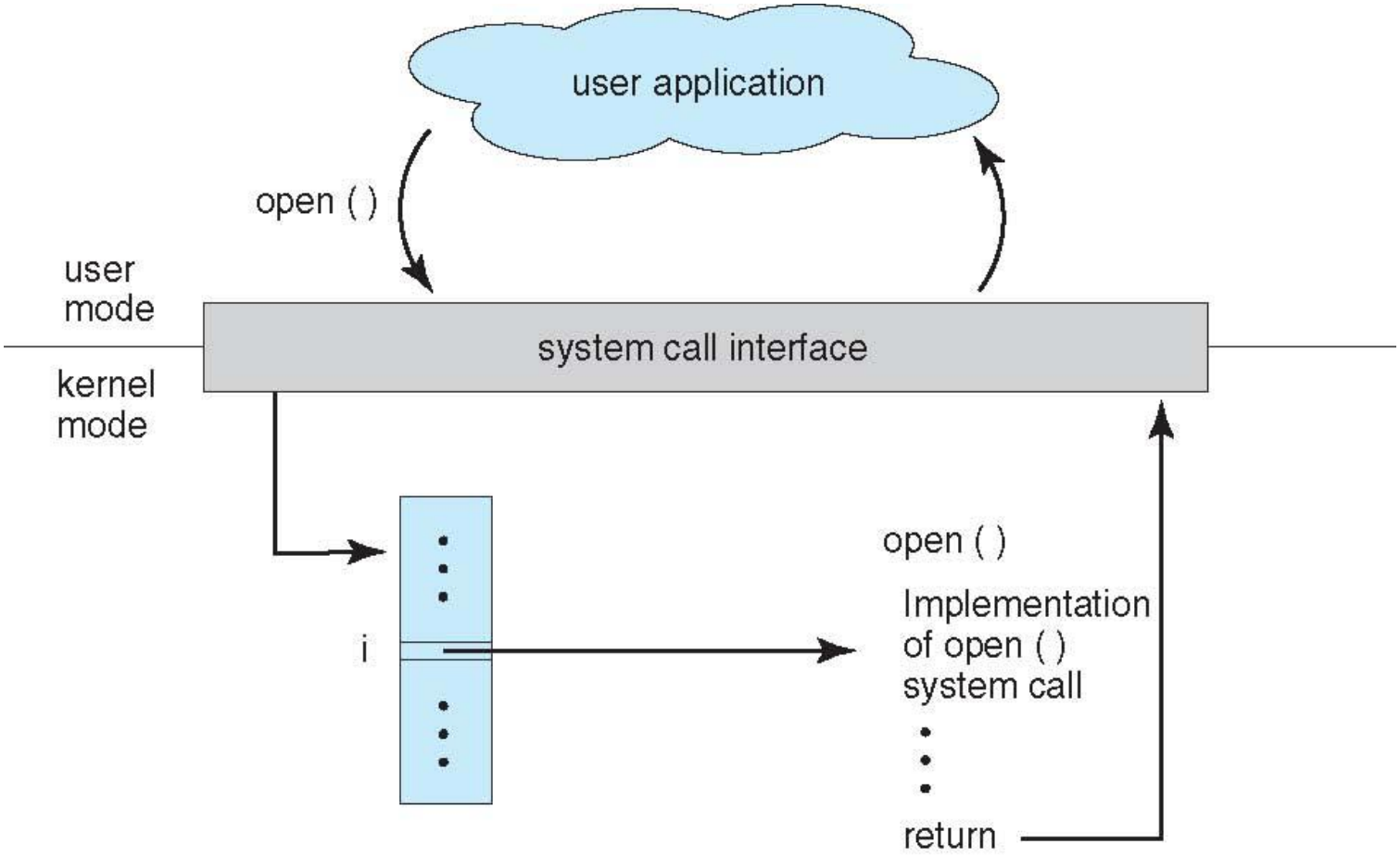
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

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 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
- The caller need 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)

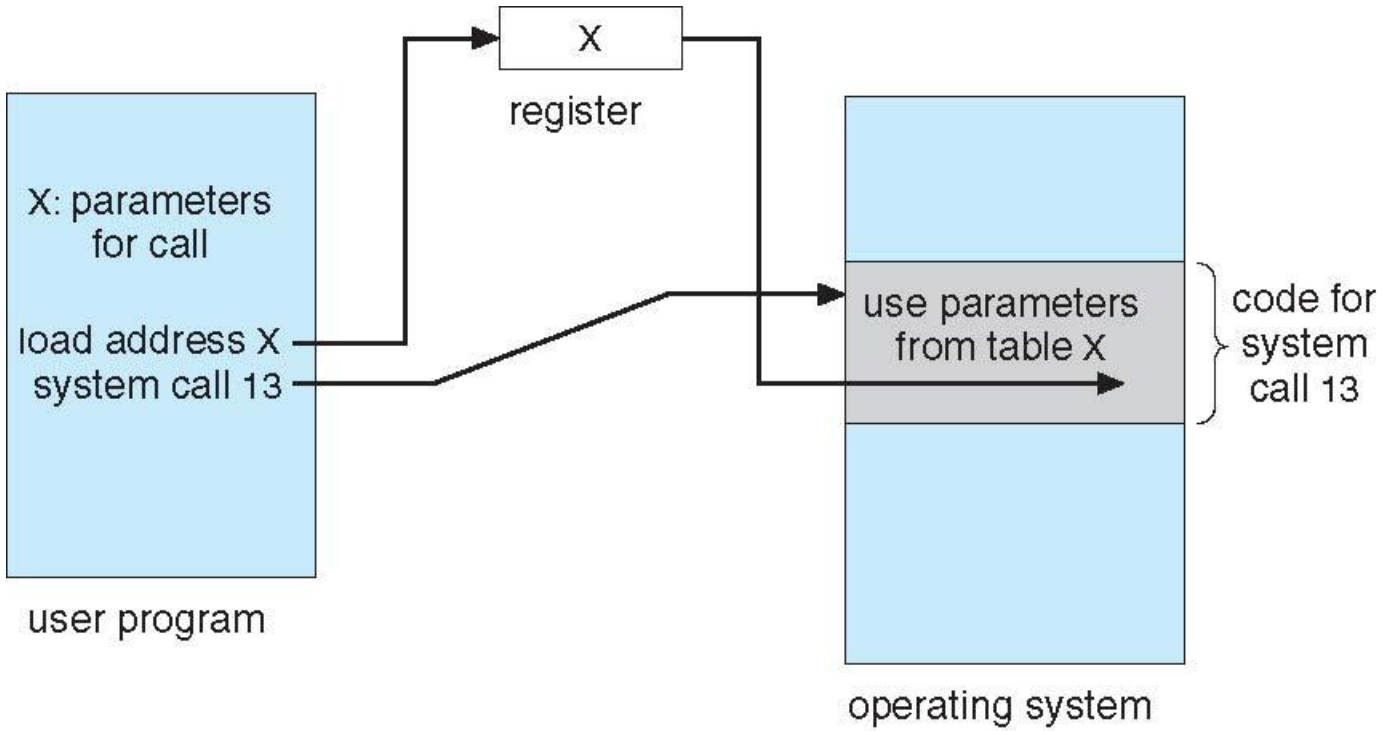
API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of 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
  - Simplest: pass the parameters in registers
    - In some cases, 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
    - This approach taken by 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

Parameter Passing via Table



Types of System Calls

- Process control
  - 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

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 (Cont.)

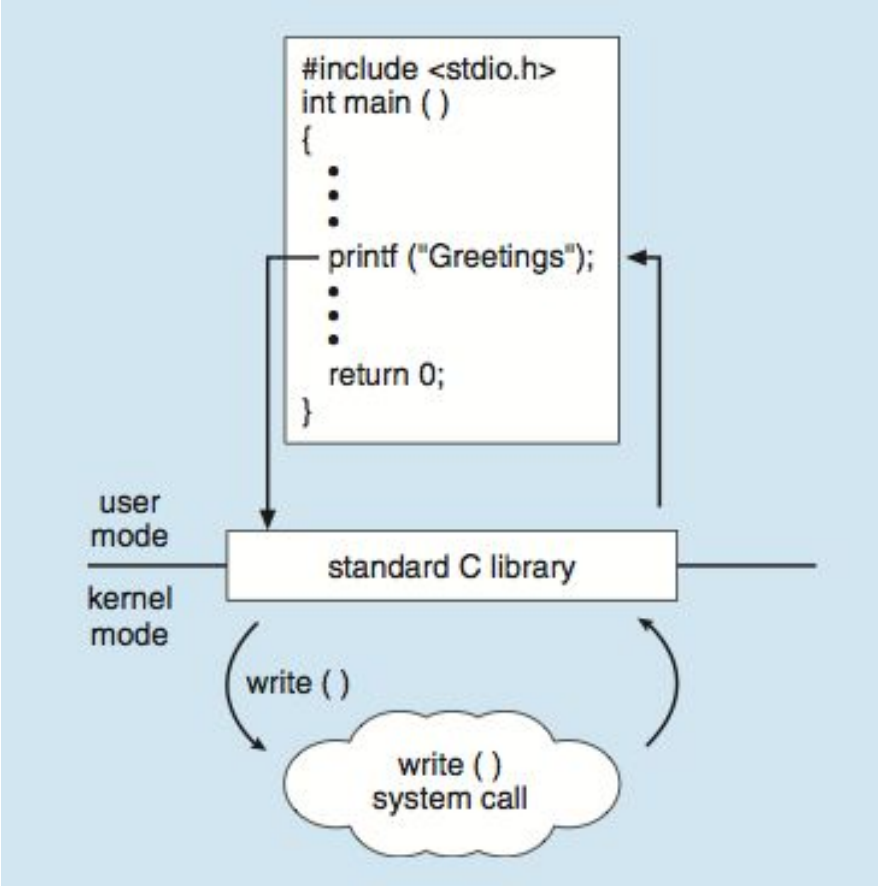
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process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

Examples of Windows and Unix System Calls

	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()

Standard C Library Example

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



SINGLE TASKING VS MULTITASKING EXAMPLE

Arduino example of single task executio

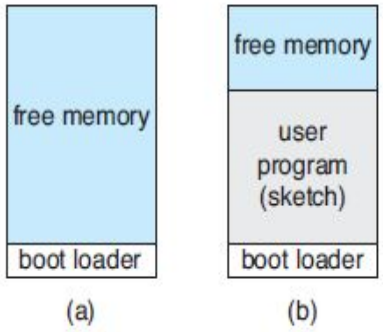
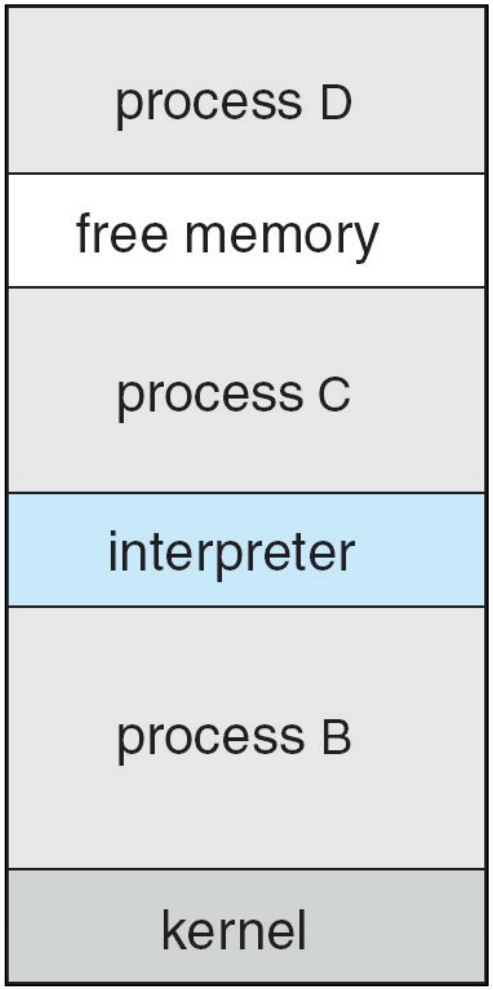


Figure 2.9 Arduino execution. (a) At system startup. (b) Running a sketch.

hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, to just name a few. To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a **sketch**) from the PC to the Arduino's flash memory via a USB connection. The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a **boot loader** loads the sketch into a specific region in the Arduino's memory (Figure 2.9). Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to. For example, if the Arduino's temperature sensor detects that the temperature has exceeded a certain threshold, the sketch may have the Arduino start the motor for a fan. An Arduino is considered a single-tasking system, as only one sketch can be present in memory at a time; if another sketch is loaded, it replaces the existing sketch. Furthermore, the Arduino provides no user interface beyond hardware input sensors.

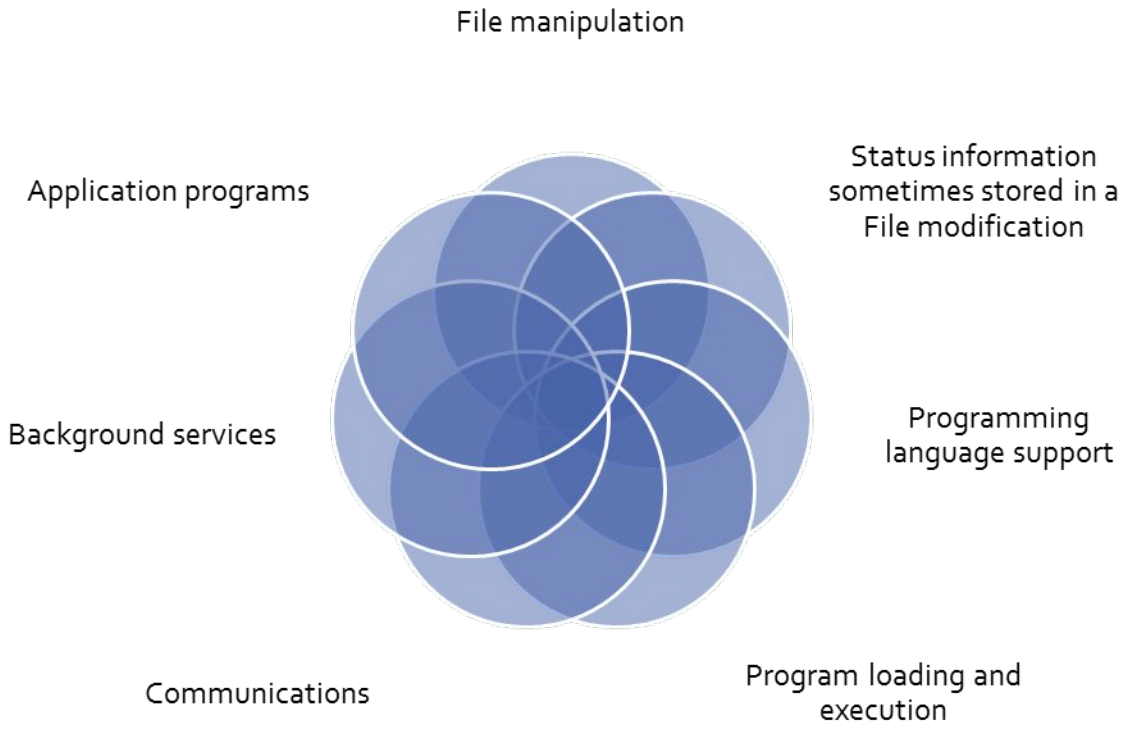
Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



System Programs/services

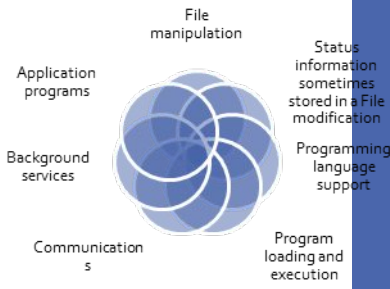
- System services provide a convenient environment for program development and execution. They can be divided into:



Daemons

- Constantly running system-program processes are known as **services**, **subsystems**, or daemons.
- One example is the network daemon : a system needed a service to listen for network connections in order to connect those requests to the correct processes.
- Other examples
  - process schedulers , error monitoring services, and print servers.
- operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

System Programs



- Provide a convenient environment for program development and execution
  - Some are simply user interfaces to system calls; others are considerably more complex

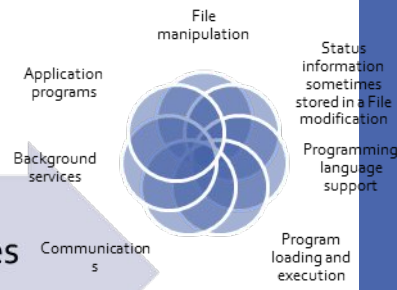
File management

- Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

Status information

- info - date, time, amount of available memory, disk space, number of users
- logging, and debugging information
- output on terminal or other output devices
- Some systems implement a **registry** - used to store and retrieve configuration information

System Services (Cont.)



File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

Programming-language support -

- Compilers, assemblers, debuggers and interpreters sometimes provided

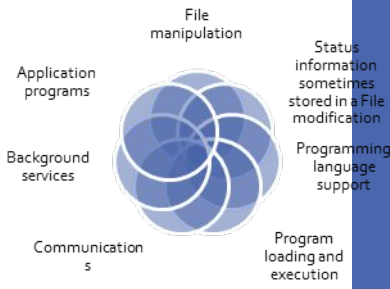
Program loading and execution-

- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

Communications - mechanism for creating virtual connections among processes, users, and computer systems

- users can send messages to one another' s screens,
- browse, e-mail messages, log in remotely, transfer files from one machine to another

System services (Cont.)



Background Services

- Launch at boot time
  - Some for system startup, then terminate
  - Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as **services, subsystems, daemons**

Linkers and loaders

~~command line on UNIX systems—for example~~, `./main`—the shell first creates a new process to run the program using the `fork()` system call. The shell then invokes the loader with the `exec()` system call, passing `exec()` the name of the executable file. The loader then loads the specified program into memory using the address space of the newly created process. (~~When a GUI interface is~~

Source files are compiled into object files that are designed to be loaded
into any physical memory location, a format known as an **relocatable object file**.

Next, the **linker** combines these relocatable object files into a single binary **executable** file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library .

A**loader** is used to load the binary executable file into memory, where it is
eligible to run on a CPU core. An activity associated with linking and loading
is **relocation**, which assigns final addresses to the program parts and adjusts
code and data in the program to match those addresses so that, for example, the
code can call library functions and access its variables as it executes.

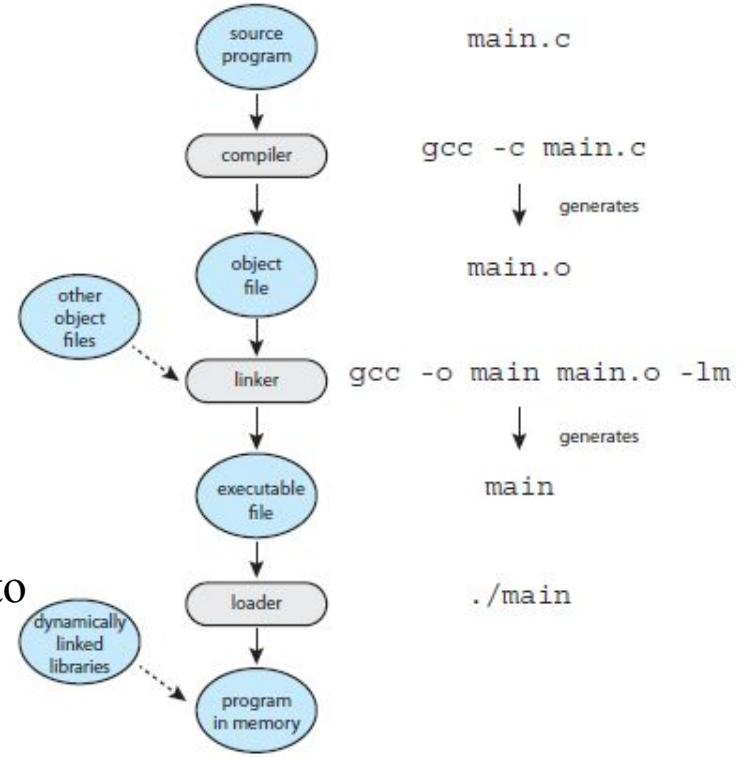


Figure 2.11 The role of the linker and loader.

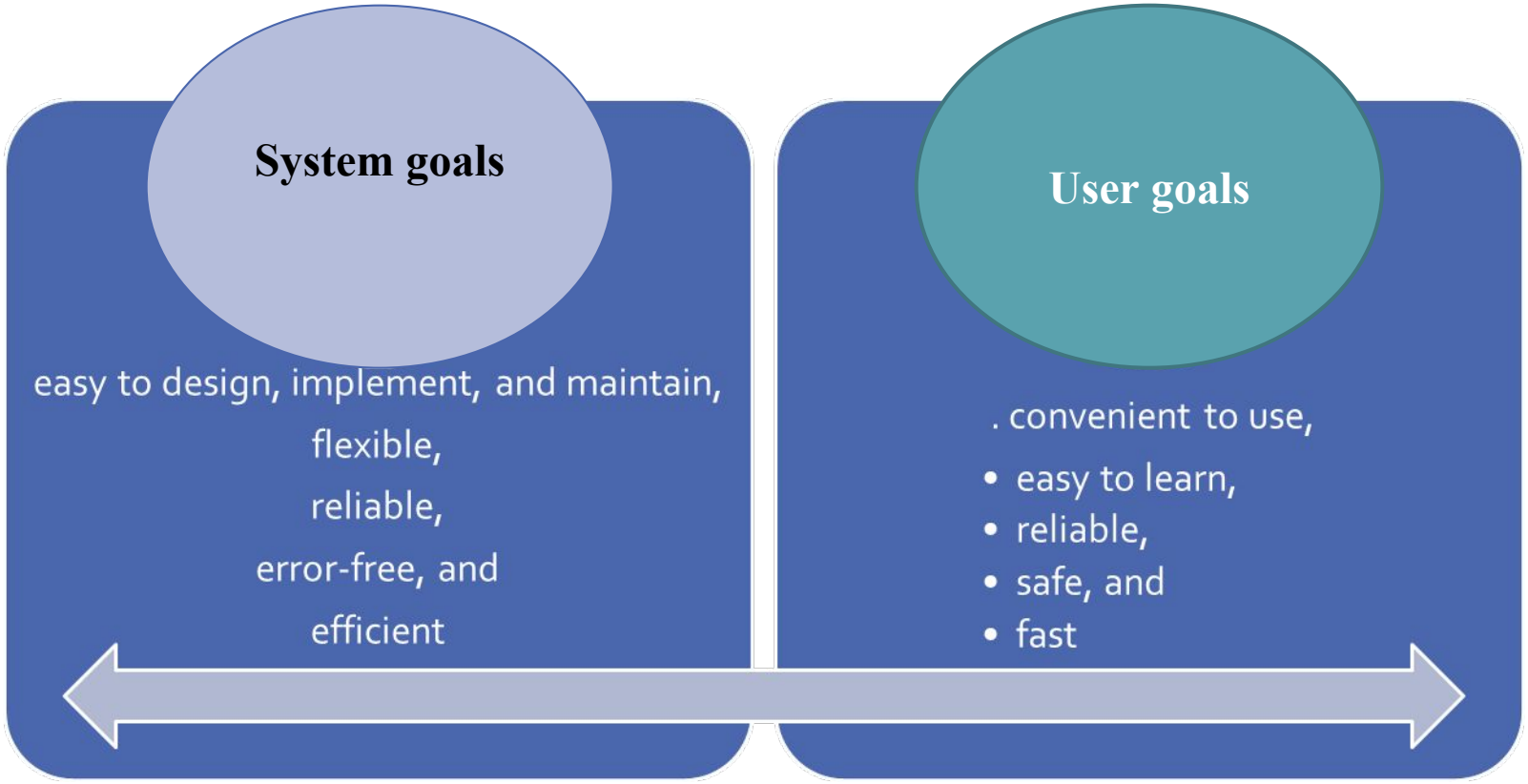
DLL?
ELF?

Why Applications Are Operating-System Specific?

- Reading summary(home task)

OPERATING SYSTEM DESIGN AND IMPLEMENTATION

Operating System Design and Implementation



Policy: ***What*** will be done?and what not
Mechanism: ***How*** to do that what is allowed in policy?

OS Implementation

- usually in a mix of languages
  - Lowest levels in assembly
  - 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
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

Operating System Structure

ways to structure can be:

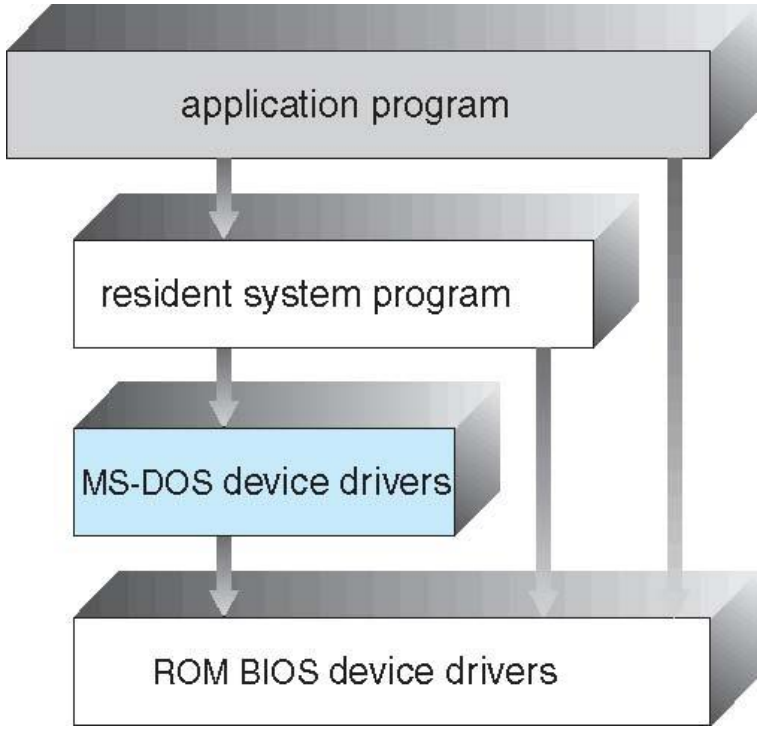
- Simple structure – MS-DOS
- More complex -- UNIX
- Layered – an abstrcation
- Microkernel -Mach

OPERATING SYSTEM STRUCTURE

Simple Structure -- MS-DOS

Not divided into modules

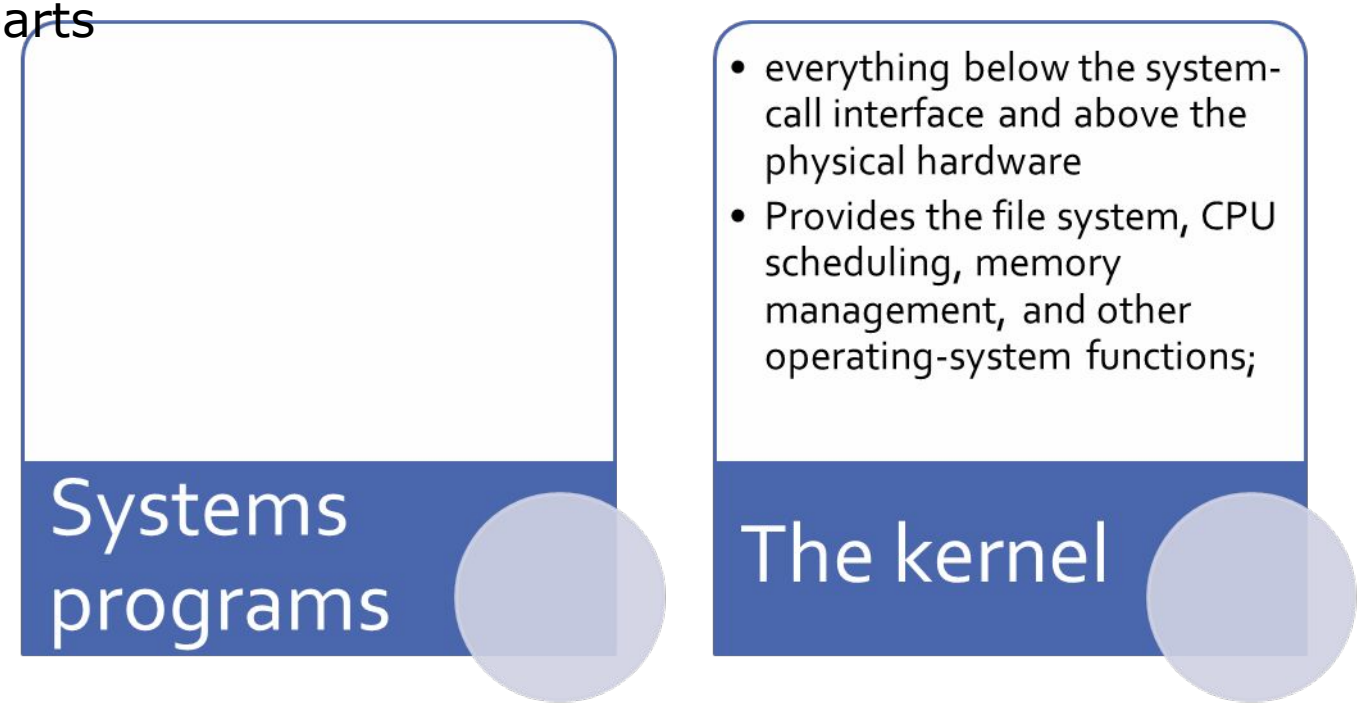
interfaces and levels of functionality are not well separated



Simple Structure -- UNIX

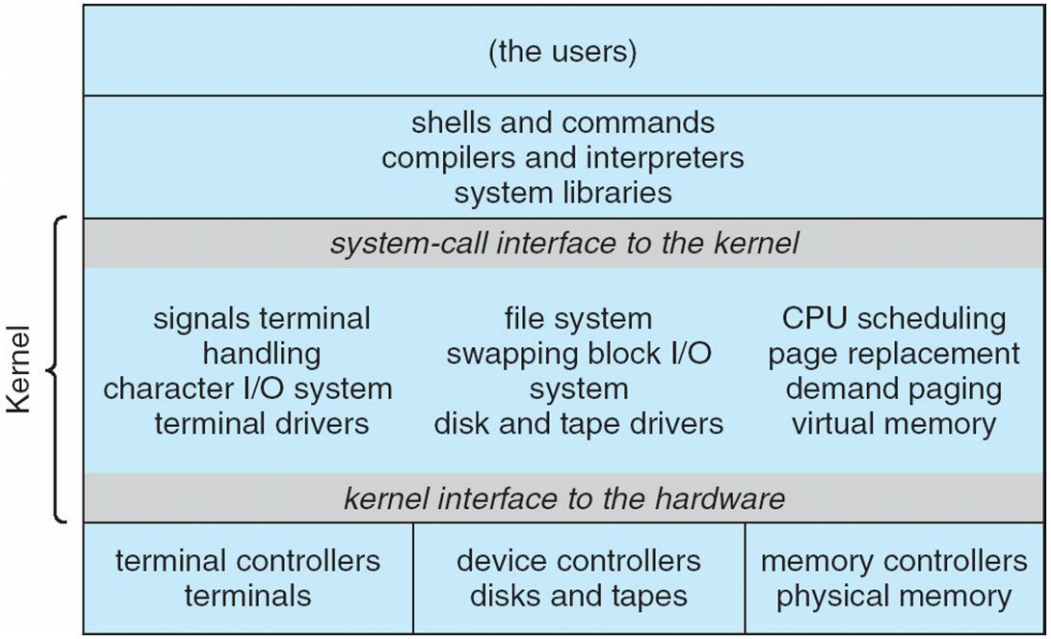
The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a **monolithic** structure—is a common technique for designing operating systems.

Example:The traditional UNIX OS consists of two separable parts



Traditional UNIX System Structure

not fully layered



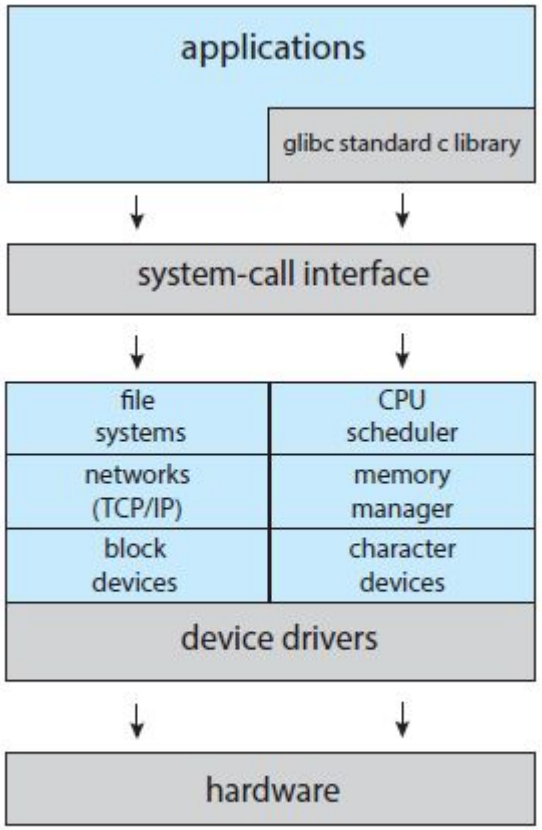
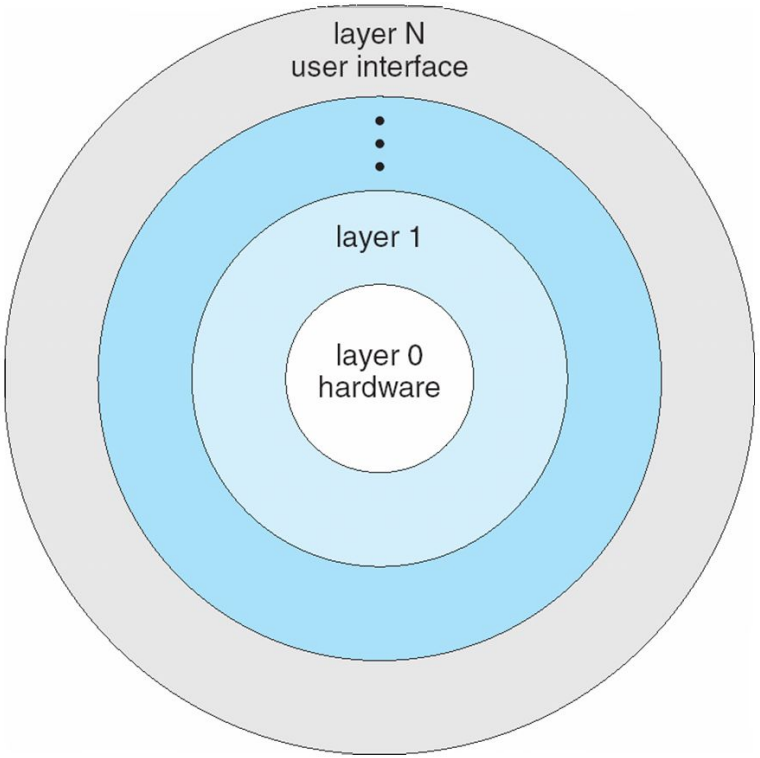


Figure 2.13 Linux system structure.

Layered Approach

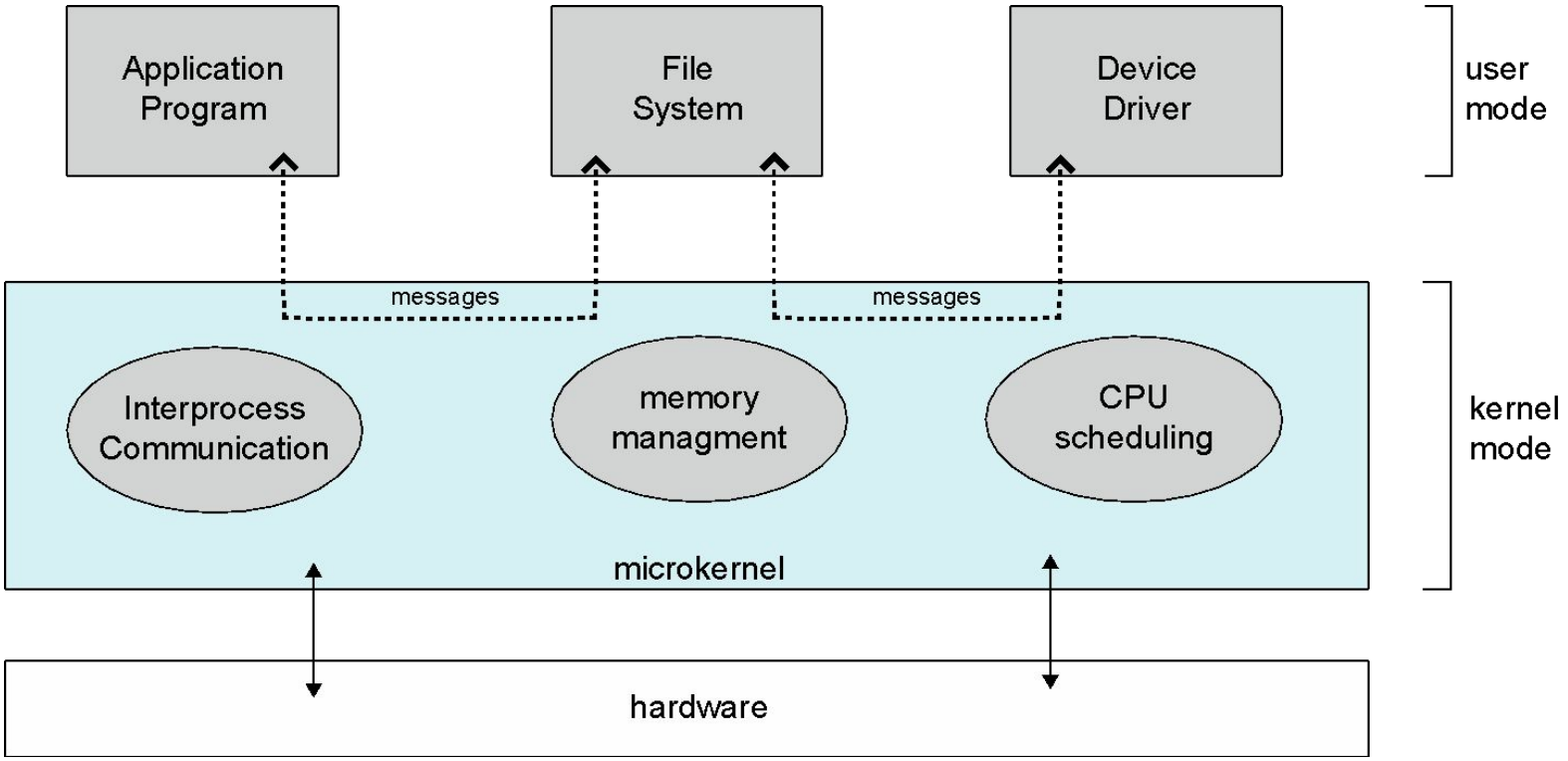
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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



Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

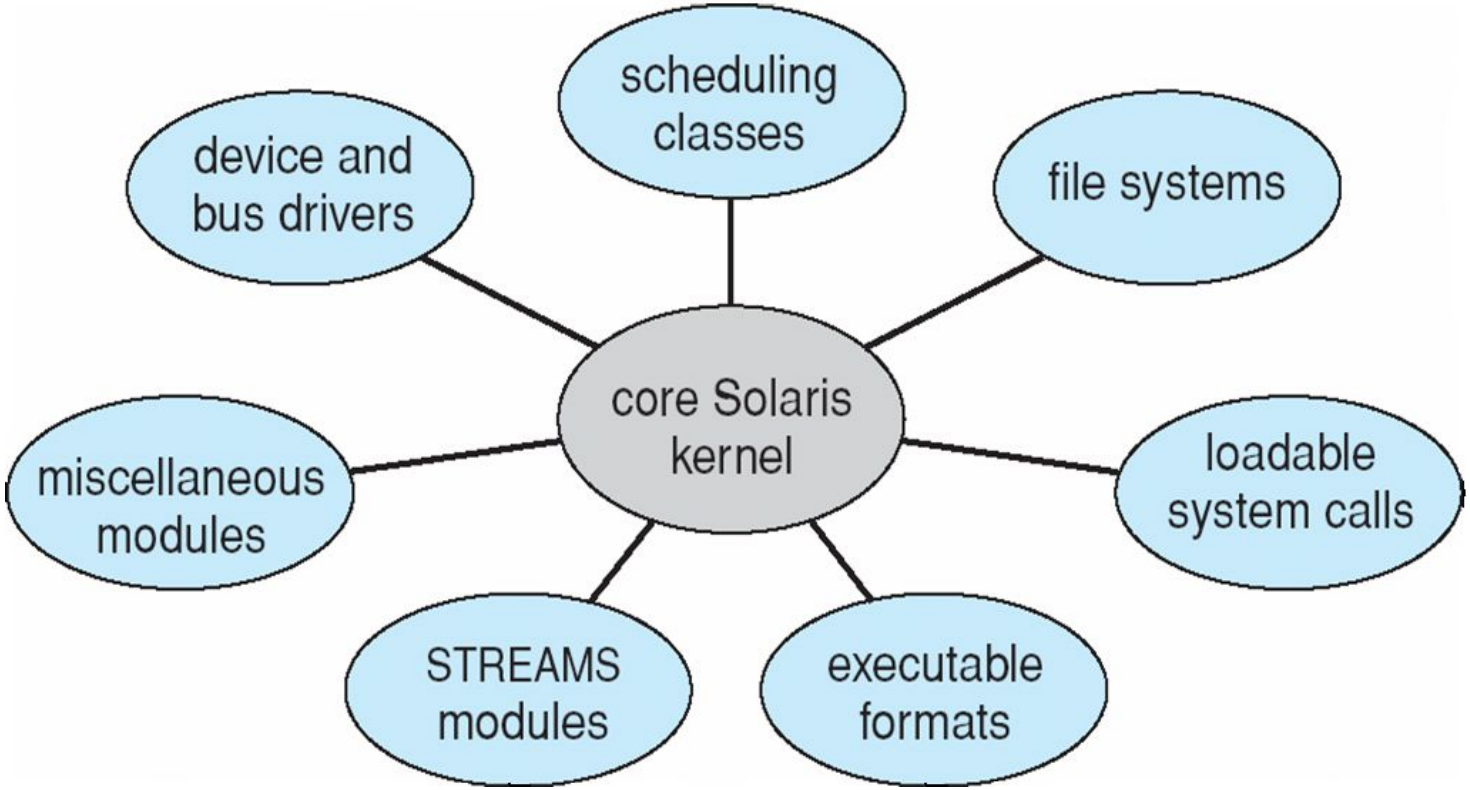
Microkernel System Structure



Modules

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

Solaris Modular Approach

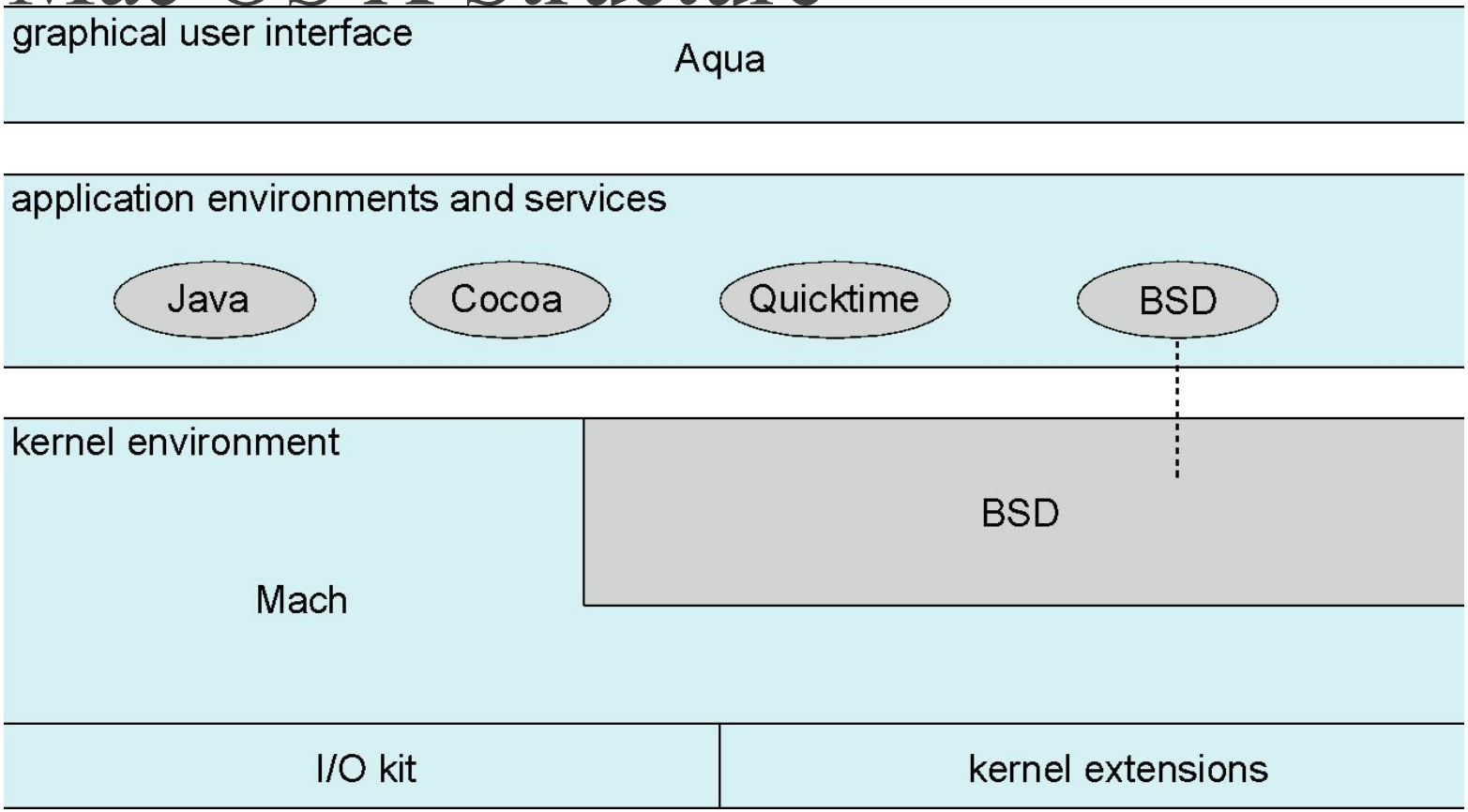


Hybrid Systems

- Hybrid combines multiple approaches to address performance, security, usability needs
- Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
- Windows mostly monolithic, plus microkernel for different subsystem ***personalities***

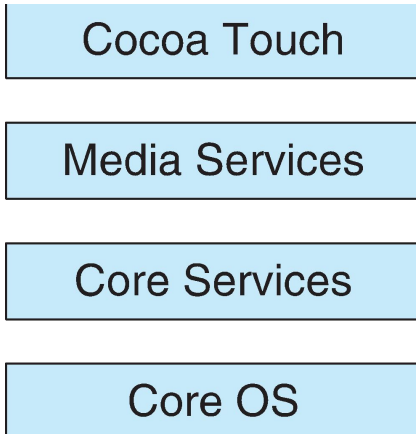
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Mac OS X Structure



iOS

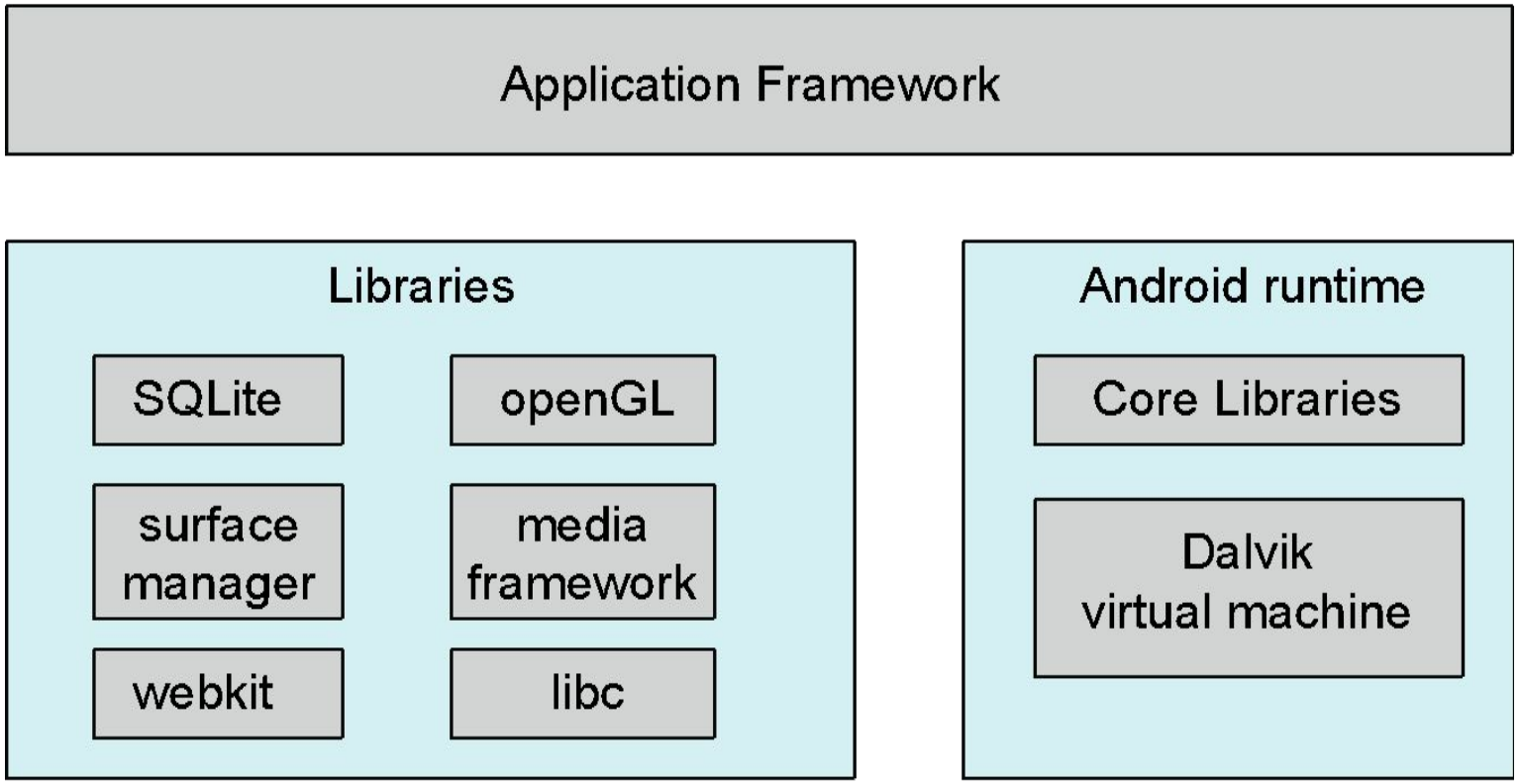
- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- 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 than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



2.9.1 Operating-System Generation

Most commonly, a computer system, when purchased, has an operating system already installed. For example, you may purchase a new laptop with Windows or macOS preinstalled. But suppose you wish to replace the preinstalled operating system or add additional operating systems. Or suppose you purchase a computer without an operating system. In these latter situations, you have a few options for placing the appropriate operating system on the computer and configuring it for use.

If you are generating (or building) an operating system from scratch, you must follow these steps:

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

Kernel configuration

- Configuring the system involves specifying which features will be included, and this varies by operating system.
- Typically, parameters describing how the system is configured is stored in a configuration file of some type, and once this file is created, it can be used in several ways.
- It is to modify a copy of the operating-system source code. Then the operating system is completely compiled (known as a **system build**).
- Data declarations, initializations, and constants, along with compilation, produce an output-object version of the operating system that is tailored to the system described in the configuration file.
- This process allows the library to contain the device drivers for all supported I/O devices, but only those needed are selected and linked into the operating system

Build linux from scratch

1. Download the Linux source code from http://www.kernel.org.
2. Configure the kernel using the "`make menuconfig`" command. This step generates the `.config` configuration file.
3. Compile the main kernel using the "`make`" command. The `make` command compiles the kernel based on the configuration parameters identified in the `.config` file, producing the file `vmlinuz`, which is the kernel image.
4. Compile the kernel modules using the "`make modules`" command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the `.config` file.
5. Use the command "`make modules_install`" to install the kernel modules into `vmlinuz`.
6. Install the new kernel on the system by entering the "`make install`" command.

When the system reboots, it will begin running this new operating system.

System boot

1. A small piece of code known as the bootstrap program or boot loader locates the kernel.
2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

- Some systems have multistage boot process:
  a small boot loader from **BIOS** is run. This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the **boot block.** **2nd boot loader** load the entire operating system into memory and begin its execution.
- Many recent computer systems have replaced the BIOS-based boot process with **UEFI** (Unified Extensible Firmware Interface).
- advantages :
  - better support for 64-bit systems and larger disks.
  - UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

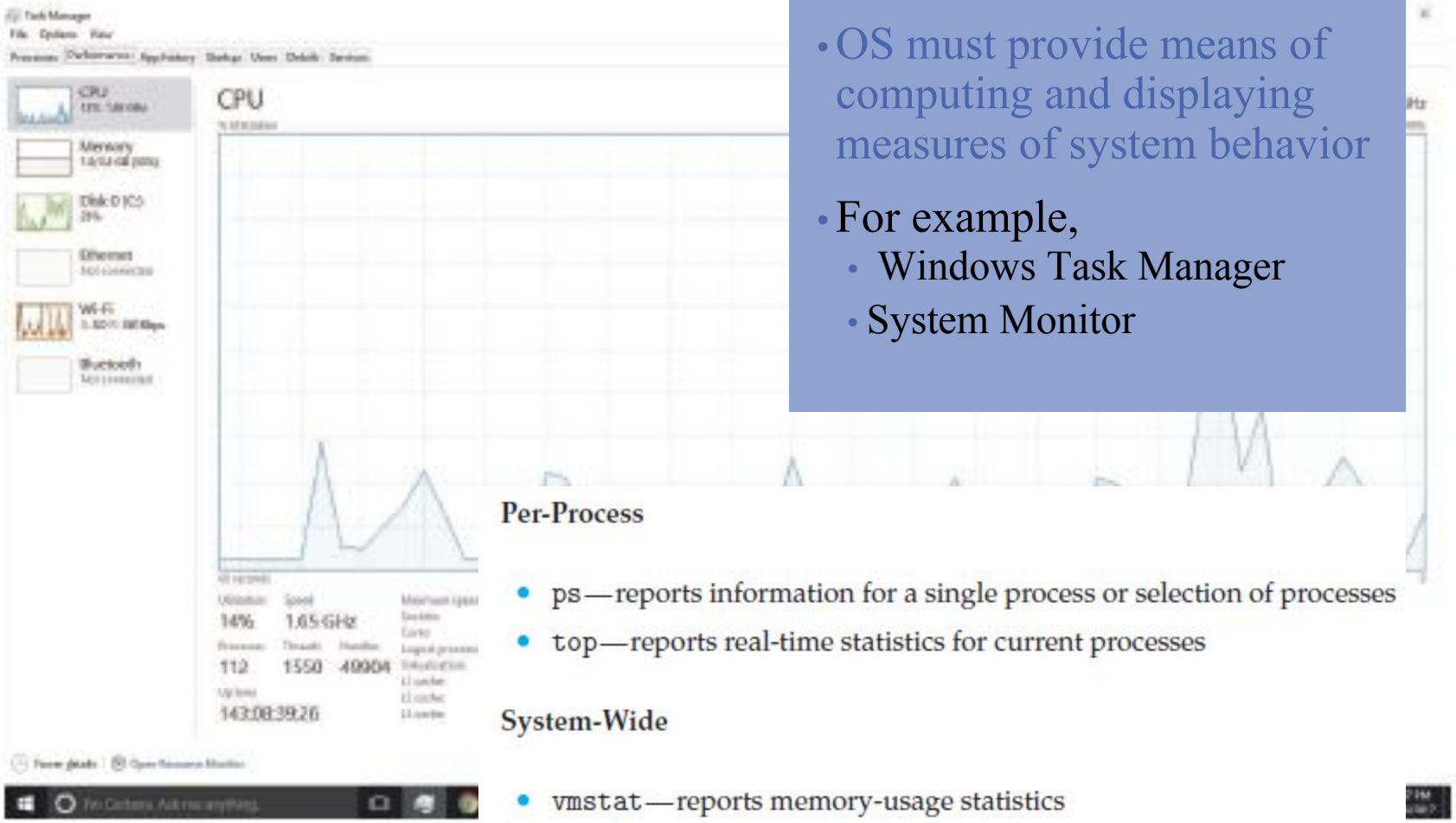
Operating-System Debugging

- **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."

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example,
  - Windows Task Manager
  - System Monitor



Per-Process

- `ps`—reports information for a single process or selection of processes
- `top`—reports real-time statistics for current processes

System-Wide

- `vmstat`—reports memory-usage statistics
- `netstat`—reports statistics for network interfaces
- `iostat`—reports I/O usage for disks

Statistics in Linux

Most of the counter-based tools on Linux systems read statistics from the /proc file system. /proc is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various per-process as well as kernel statistics. The /proc file system is organized as a directory hierarchy, with the process (a unique integer value assigned to each process) appearing as a subdirectory below /proc. For example, the directory entry /proc/2155 would contain per-process statistics for the process with an ID of 2155. There are /proc entries for various kernel statistics as well. In both this chapter and Chapter 3, we provide programming projects where you will create and access the /proc file system.

End of Chapter 2