

# Chapter 2: Operating-System Services

---





# Objectives

---

- Identify **services provided by an operating system**
- Illustrate **how system calls are used** to provide operating system services
- Compare and contrast **monolithic, layered, microkernel, modular, and hybrid strategies** for designing operating systems
- Apply tools for **monitoring operating system performance**
- Design and implement **kernel modules for interacting with a Linux kernel**





# Operating System Services

- Operating systems provide an **environment for execution of programs and services to programs and users**
- 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**).
    - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **touch-screen**, **Batch**
  - **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
  - **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.





# Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **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

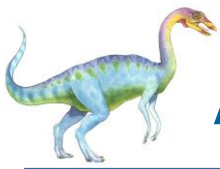




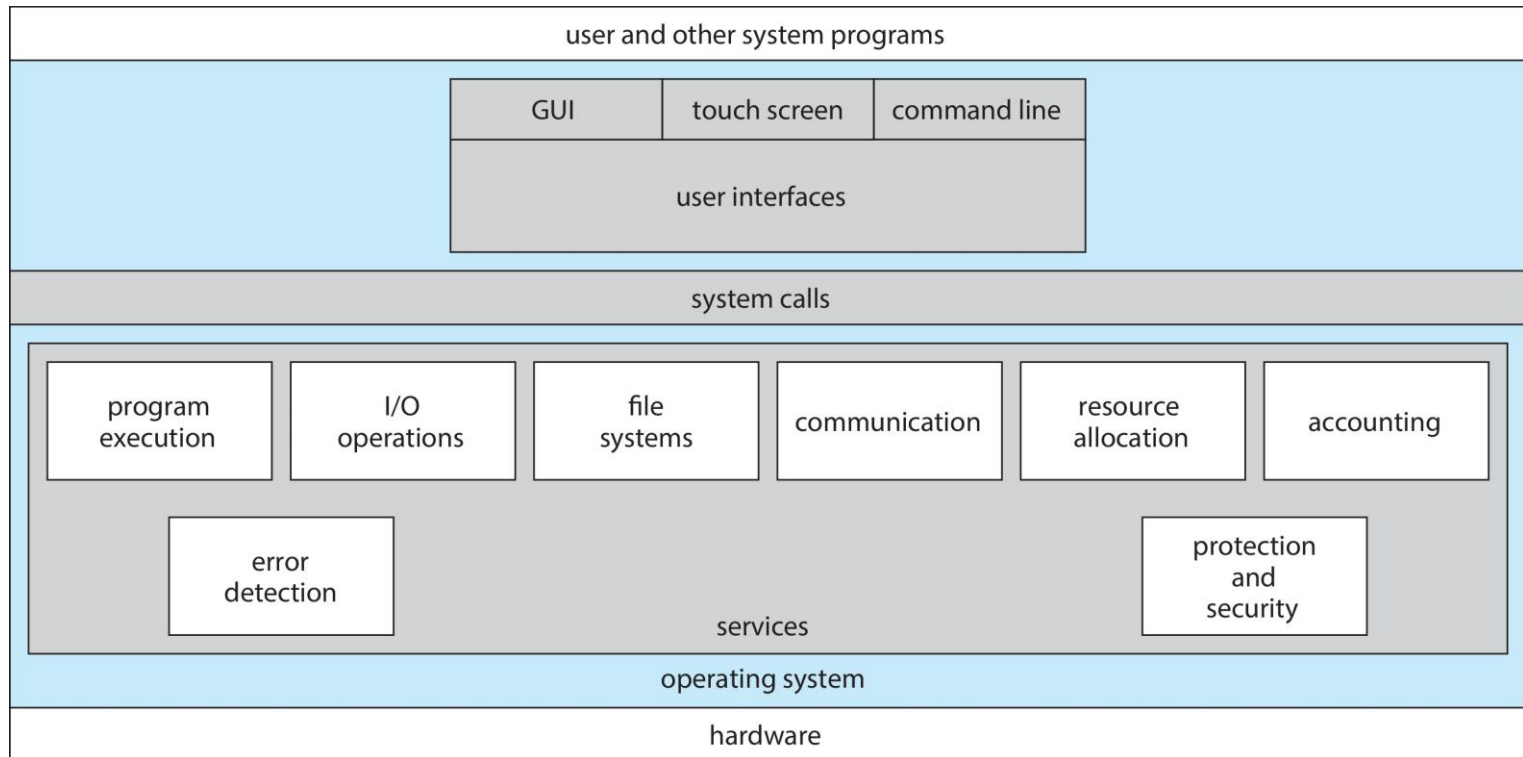
# Operating System Services (Cont.)

- 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 of them
    - ▶ Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Logging** - 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





# A View of Operating System Services



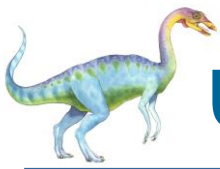


# Command Line interpreter

---

- Command interpreter as a special program that is running when a process is initiated or when a user first logs on
  - ***C shell, Bourne-Again (bash) shell, Korn shell***
- CLI allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn't require shell modification





# User Operating System Interface - 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 i.e., **folder**)
  - Invented at Xerox PARC
- Many systems now include **both CLI and GUI interfaces (Why??)**
  - 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)

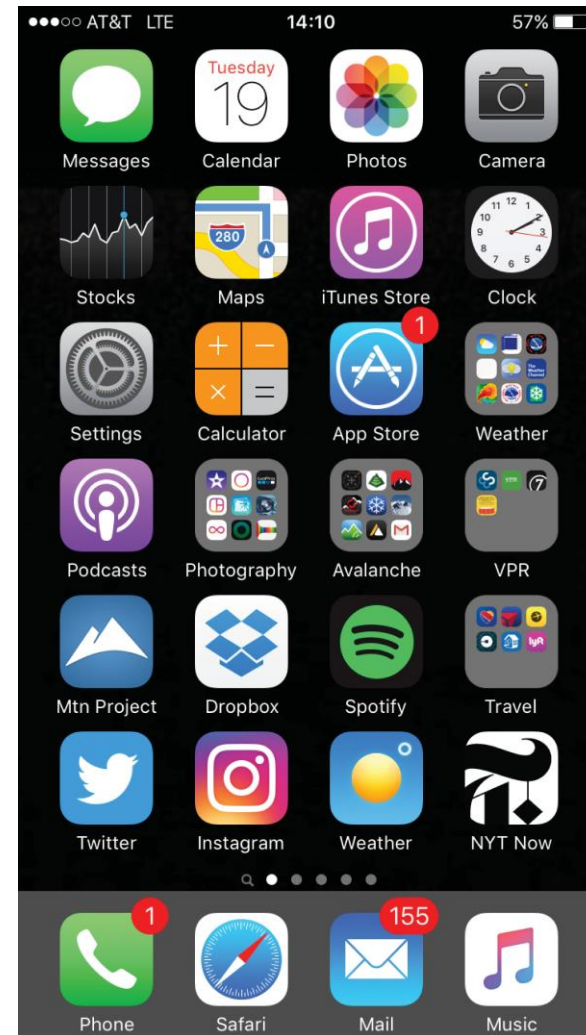






# Touchscreen Interfaces

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





# System Calls

---

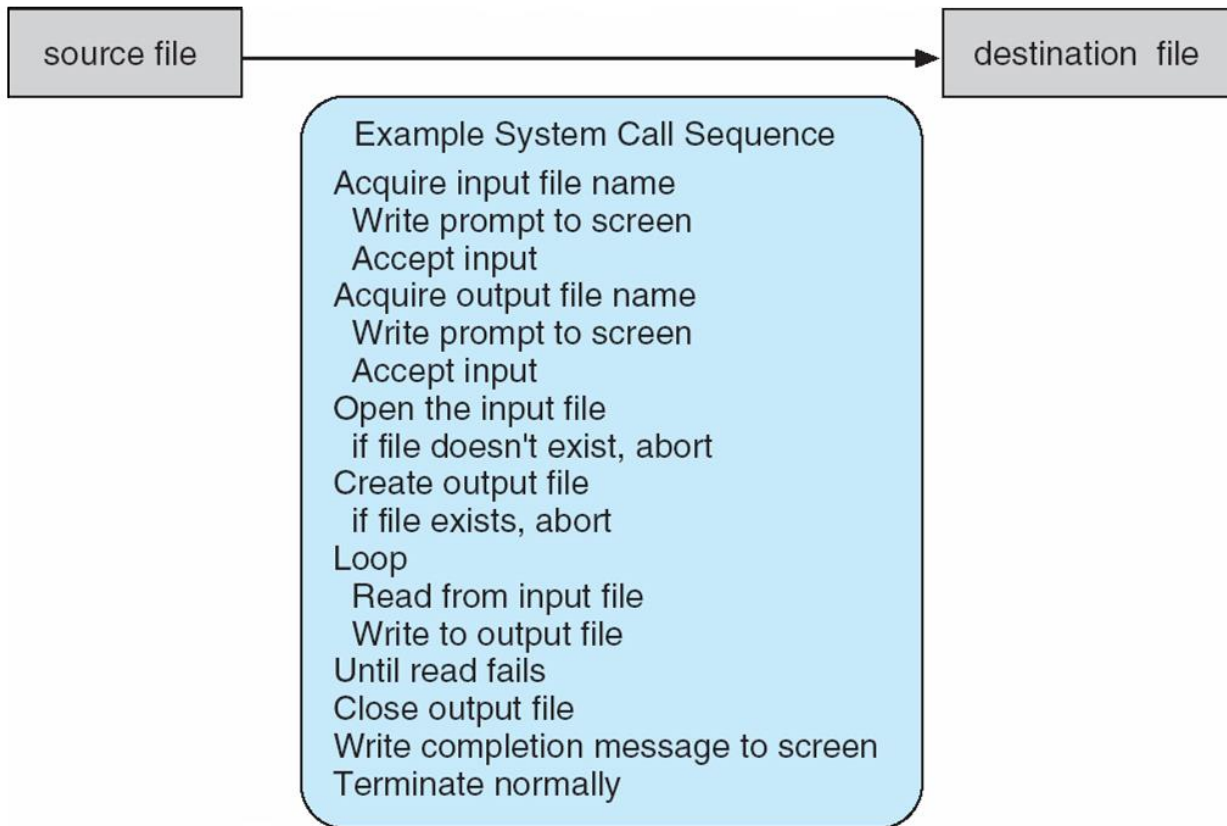
- **Programming interface** to the services provided by the OS
- Typically written in a **high-level language** (C or C++)
  - certain low-level tasks may be written using **assembly-language instructions**
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather 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)





# Example of System Calls

- System call sequence to copy the contents of one file to another file
- `cp in.txt out.txt`





# 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:

<pre>#include &lt;unistd.h&gt;</pre>		
<pre>ssize_t</pre>	<pre>read(int fd, void *buf, size_t count)</pre>	
return	function	parameters
value	name	

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 into which the data will be read
- `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`.





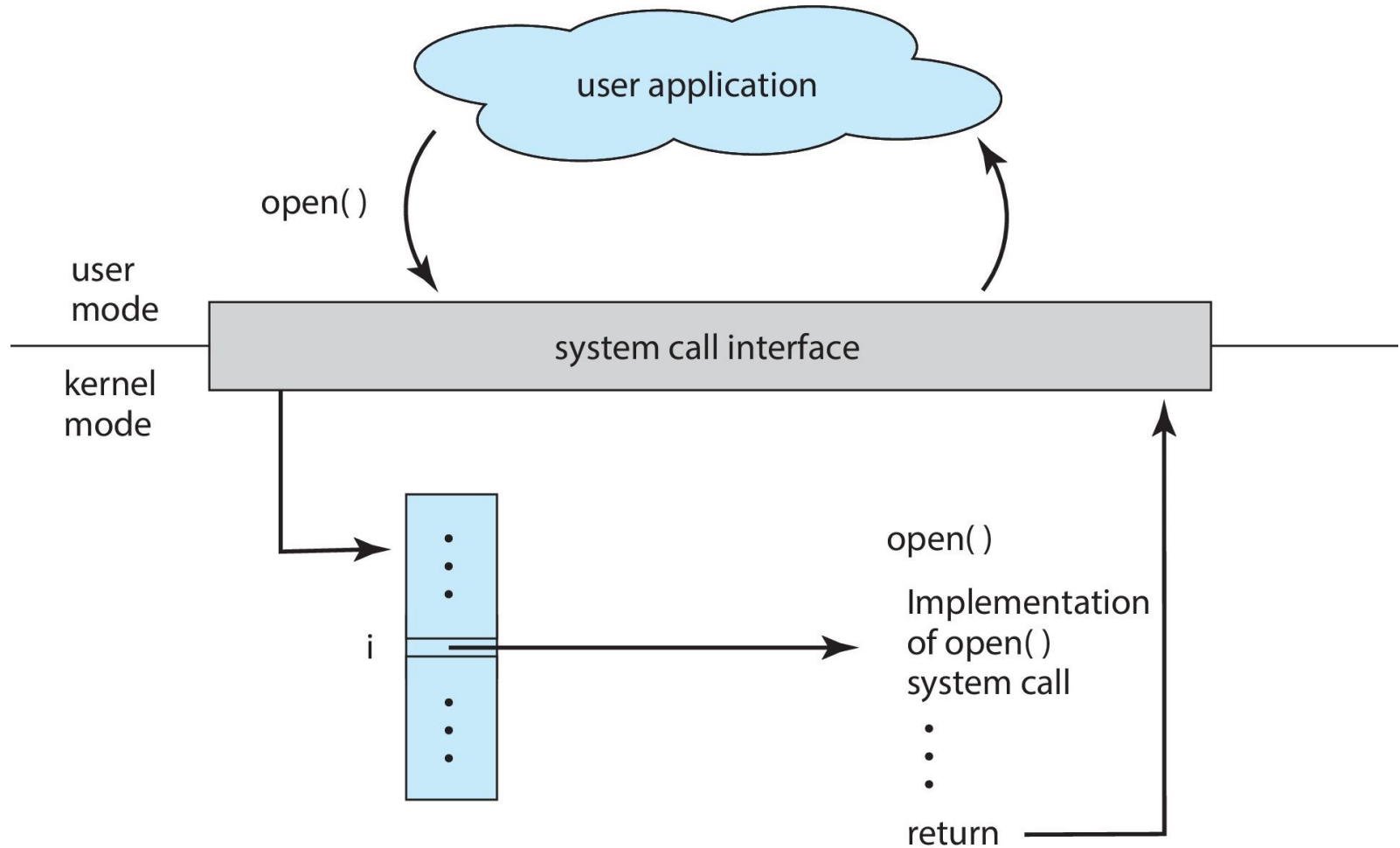
# System Call Implementation

- Typically, a number is 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





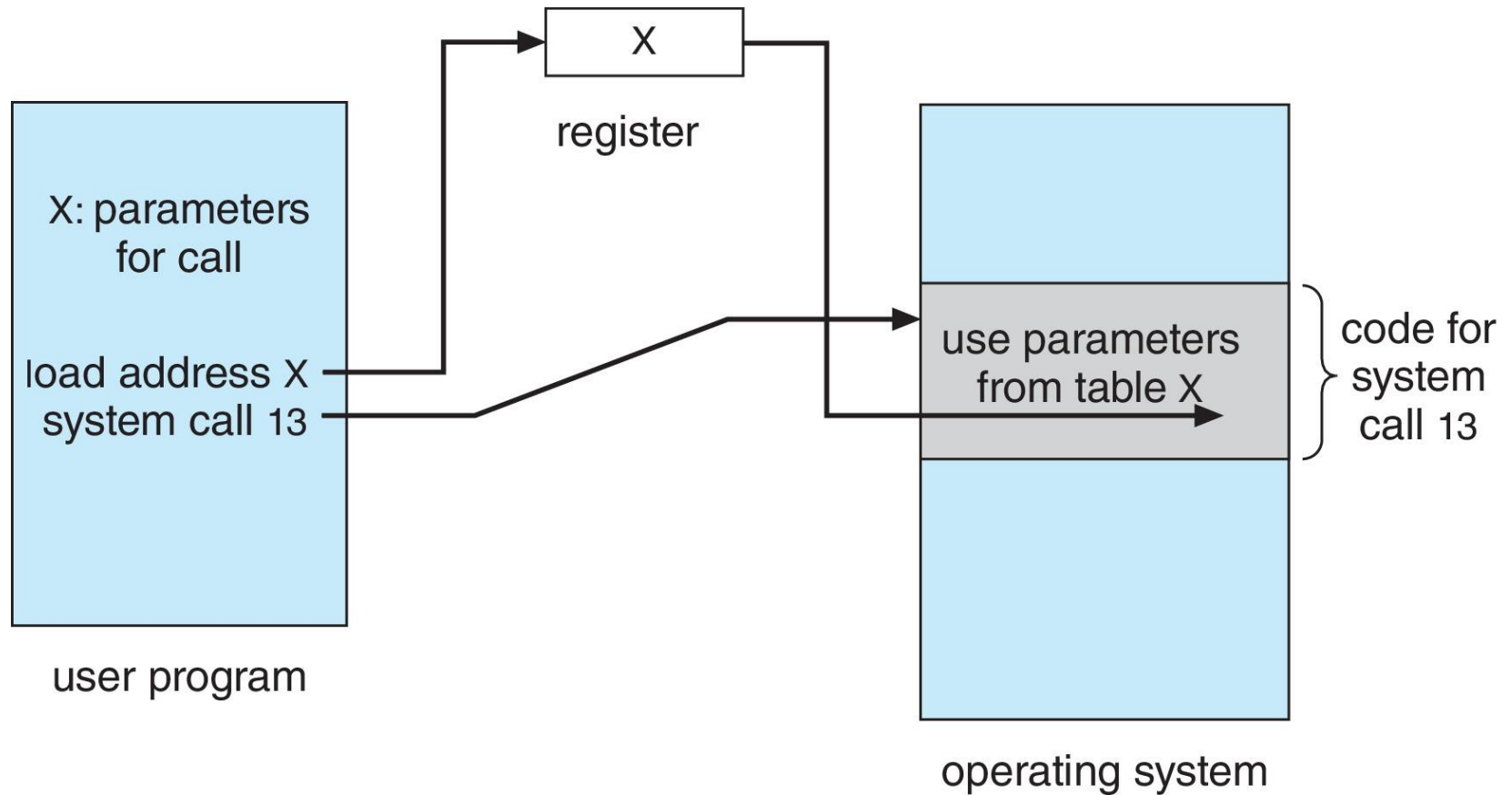
# System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Why parameter passing?
- 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 *create\_process()*, terminate process *terminate process()*
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory





# Types of System Calls (Cont.)

---

- 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**
    - ▶ From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices





# Types of System Calls (Cont.)

---

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





# Examples of Windows and Unix System Calls

## EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

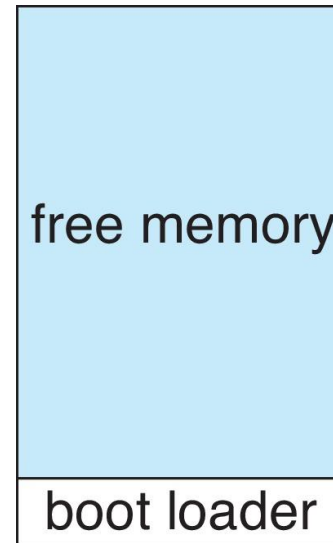
	Windows	Unix
<b>Process control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File management</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device management</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Information maintenance</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Communications</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protection</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()





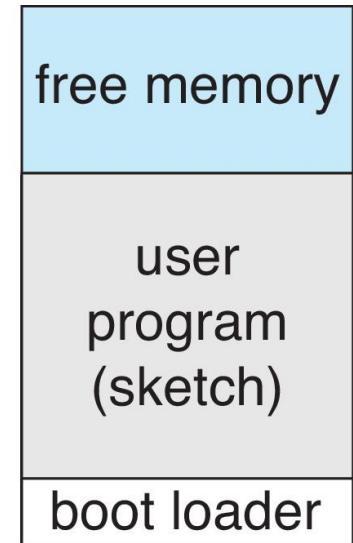
# Example: Arduino

- Arduino is a **simple hardware platform** consisting of a microcontroller along with **input sensors that respond to a variety of events**.
- the Arduino provides no user interface **beyond hardware input sensors**.
- Single-tasking
- No operating system
- **Boot loader loads** program
- Programs (sketch) loaded via **USB into flash memory**
- Single memory space
- Program exit -> shell reloaded



(a)

At system startup



(b)

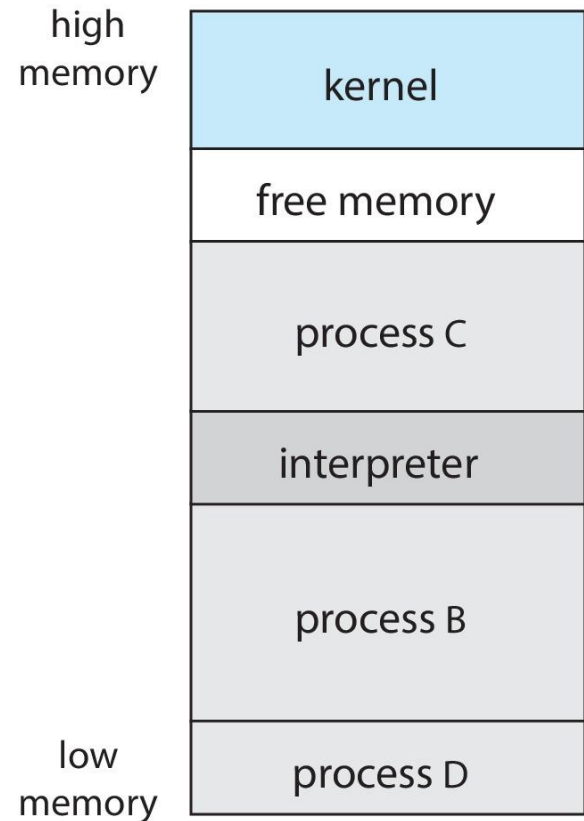
running a program





# 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 memory
  - Depending on how the command was issued,
    - ▶ the shell either waits for the process to finish
    - ▶ or runs the process “in the background.”
  - Process exits with:
    - ▶ `code = 0` – no error
    - ▶ `code > 0` – error code





# Linkers and Loaders

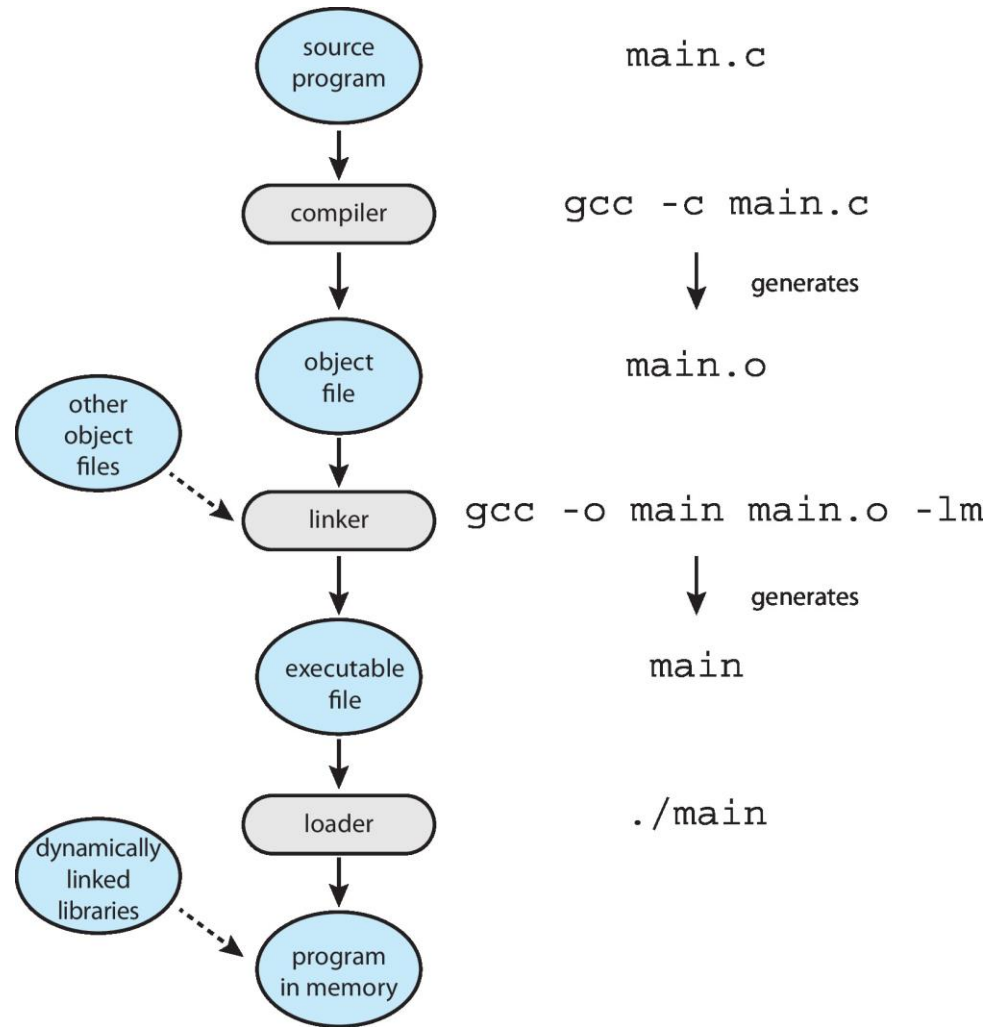
- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable file**
  - Also brings in libraries
- Program resides on **secondary storage as binary executable**
- Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and **adjusts code and data in program to match those addresses**
- Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, **shared by all that use the same version of that same library** (loaded once)







# The Role of the Linker and Loader





# Why Applications are Operating System Specific

- Apps compiled on one system usually **not executable on other operating systems**
- Each operating system provides its **own unique system calls**
  - **Own file formats**, etc.
- Apps can be multi-operating system
  - Written in **interpreted language** like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a **VM containing the running app** (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- The general lack of application mobility has several causes, all of which still make developing **cross-platform applications a challenging task**.





# Design and Implementation

---

- Design and Implementation of OS is not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- User goals and System goals
  - 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
- Specifying and designing an OS is highly creative task of software engineering





# Policy and Mechanism

---

- Important design principle: **Separation of policy from mechanism**
- **Policy:** **What** needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism:** **How** to do something?
  - Example: timer
- The separation of policy from mechanism is a very important principle, **it allows maximum flexibility** if **policy decisions are to be changed later**.
  - Example: change 100 to 200





# Implementation

- Operating systems are **collections of many programs**, written by many people over a long period of time
- Much variation
  - Early OSes in assembly language; Now C, C++
- Actually usually a mix of languages; Example Android
  - Kernel is written mostly in **C with some assembly language**
  - system libraries are written in **C or C++**
  - application frameworks, which provide the developer interface to the system, are written **mostly in Java**
- More high-level language (because of interpreter) easier to **port** to other hardware
  - But slower





# Operating System Structure





# Monolithic Structure – Original UNIX

---

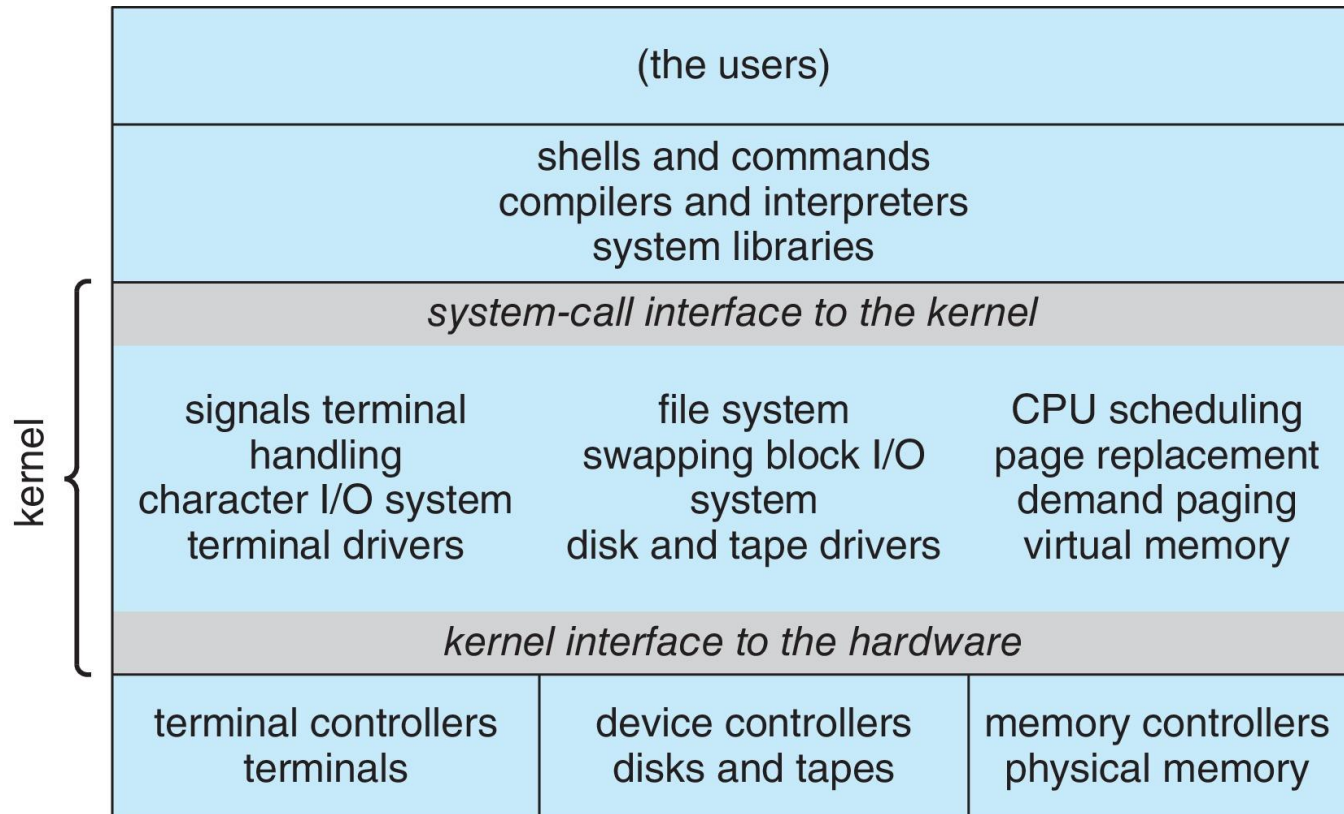
- The simplest structure for organizing an operating system is **no structure at all**.
  - place all of the functionality of the kernel into a **single, static binary file** that runs in a **single address space**. (**Any Advantage ??**)
- The UNIX OS consists of two separable parts
  - Systems programs
  - 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





# Traditional UNIX System Structure

Beyond simple **but not fully layered**

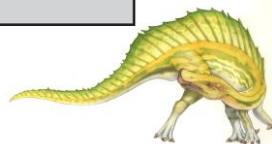
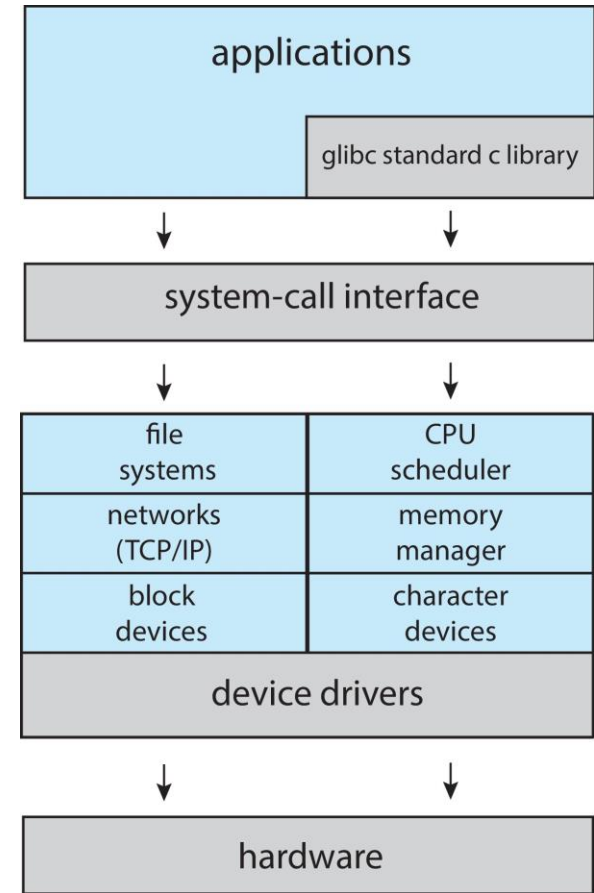






# Linux System Structure

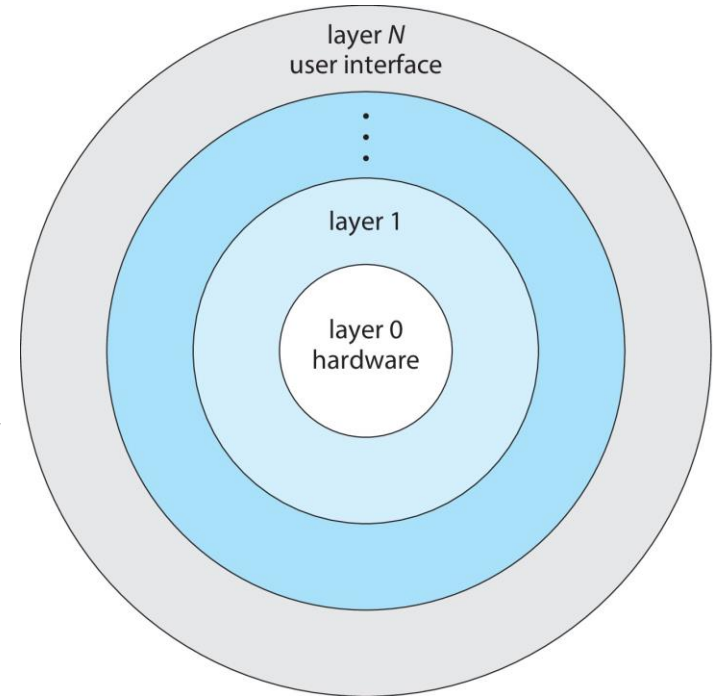
- Monolithic plus **modular design**
- The Linux kernel runs entirely in kernel mode in a single address space





# Layered Approach

- monolithic approach->**tightly coupled**
  - changes to one part of the system can have **wide-ranging effects on other parts**.
- 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**
- layer  $M$ —consists of **data structures and a set of functions** that can be invoked by higher-level layers. Layer  $M$ , in turn, can **invoke operations on lower-level layers**.
- Simplicity of construction and debugging



- Not used for full design; used within several important **OS subsystems** e.g., networking





# Microkernels

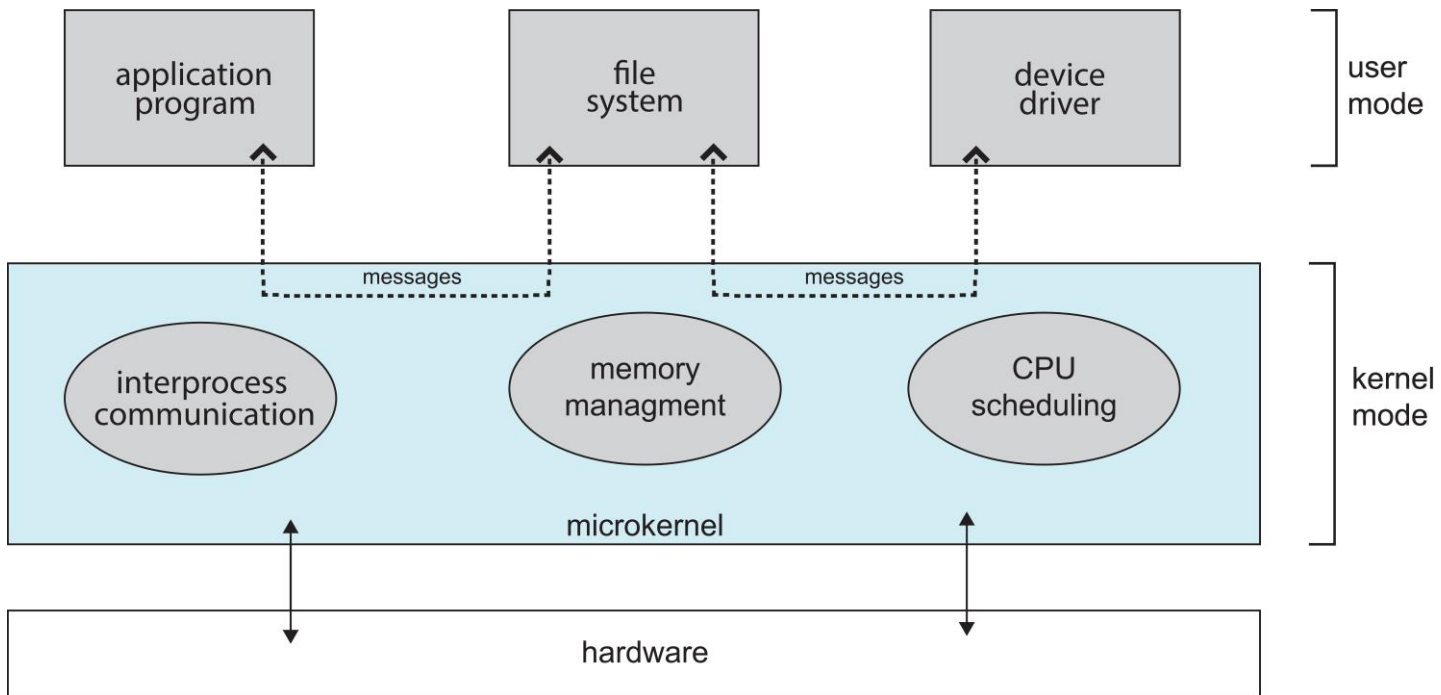
- **Theme:** Moves as much **from the kernel into user space**
- **Mach** is an example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between **user modules using message passing**
- Benefits:
  - Easier to extend a microkernel (All new services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





# Microkernel System Structure

- Microkernel is to provide communication between the **client program** and the **various services** that are also running in user space.





# Modules

---

- Many modern operating systems implement **loadable kernel modules (LKMs)**
  - 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
- This type of **design is common in modern implementations** of UNIX, such as Linux, macOS, and Solaris, as well as Windows.
- Overall, similar to layers but with more flexible





# Hybrid Systems

---

- Most modern operating systems are not one pure model
  - 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
  - Below is kernel consisting of **Mach microkernel and BSD Unix parts**, plus I/O kit and dynamically **loadable modules** (called **kernel extensions**)





# macOS and iOS Structure

---

- Read by yourself





# Darwin

---

- Read by yourself







# Android

---

- Read by yourself





# Building and Booting an Operating System

---

- Operating systems generally designed to run on a class of systems with variety of peripherals
- Commonly, operating system already installed on purchased computer
  - But can **build and install some other operating systems**
  - If generating an operating system from scratch
    - ▶ Write the operating system source code
    - ▶ Configure the operating system for the system on which it will run
    - ▶ Compile the operating system
    - ▶ Install the operating system
    - ▶ Boot the computer and its new operating system





# Building and Booting Linux

---

- Download Linux source code (<http://www.kernel.org>)
- Configure kernel via “make menuconfig”
- Compile the kernel using “make”
  - Produces `vmlinuz`, the kernel image
  - Compile kernel modules via “make modules”
  - Install kernel modules into `vmlinuz` via “make modules\_install”
  - Install new kernel on the system via “make install”





# Operating-System Debugging

---

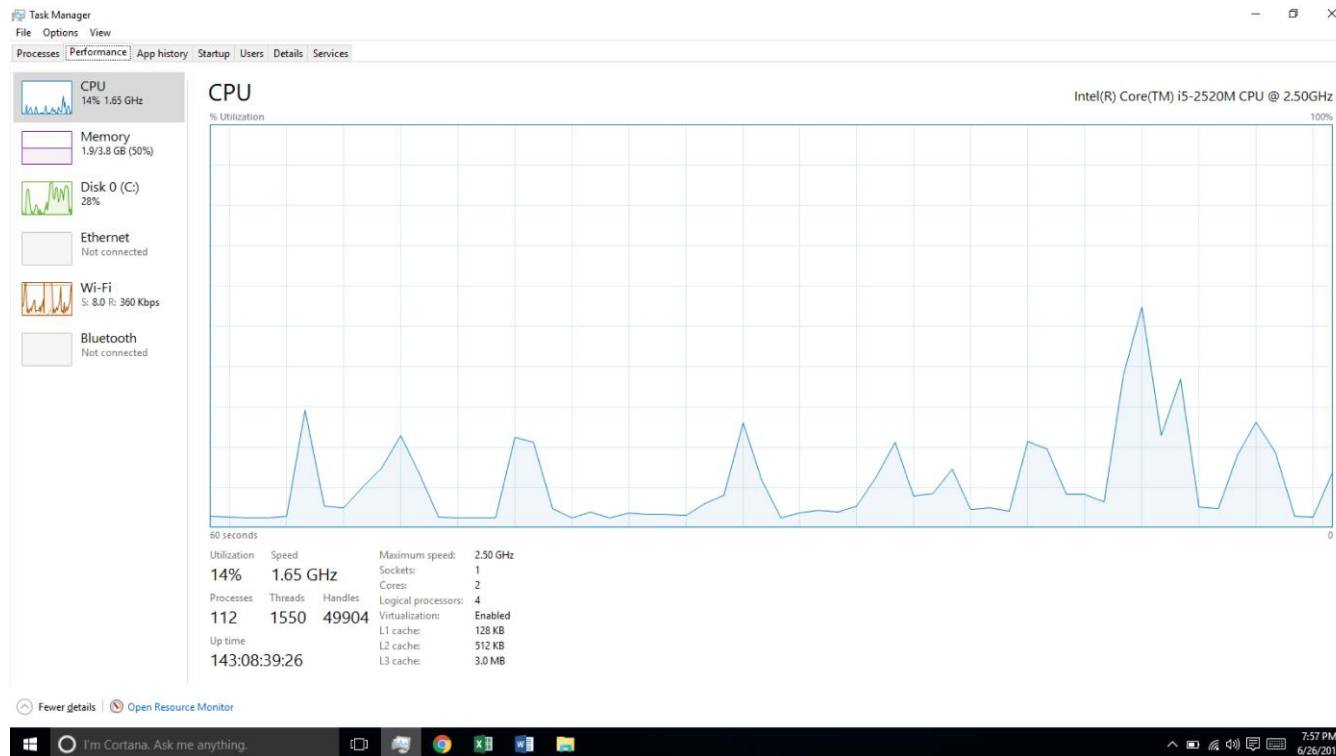
- **Debugging** is finding and fixing errors, or **bugs**
- Also **performance tuning**
- OS generate **log files** containing error information
- **Failure of an application** can generate **core dump** file **capturing** memory of the process
- **Operating system failure** can generate **crash dump** 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





# Performance Tuning

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





# Tracing

---

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets





# BCC

- Debugging **interactions** between **user-level and kernel code** nearly impossible without toolset **that understands both an instrument their actions**
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
- For example, **disksnoop.py** traces disk I/O activity

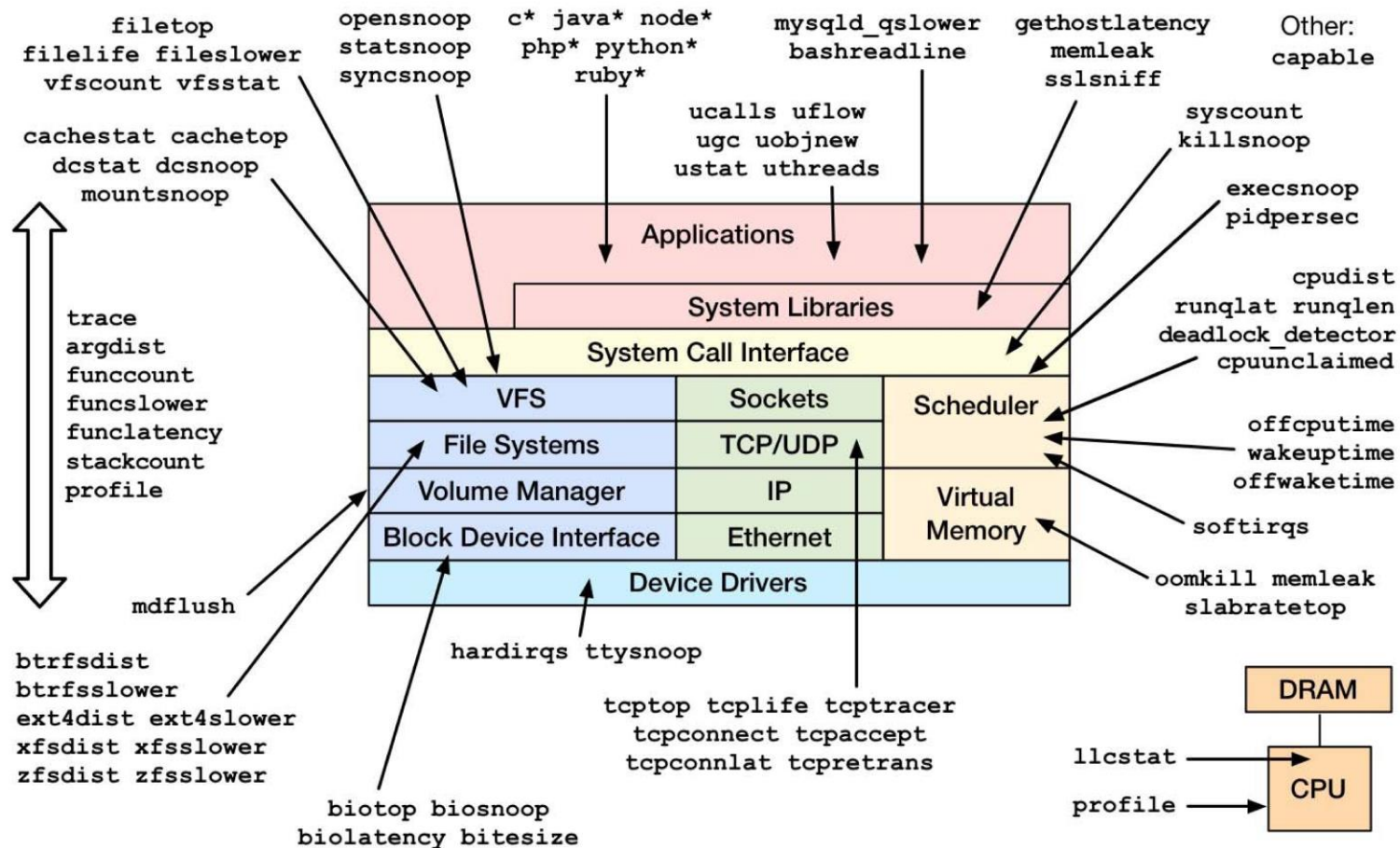
TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35





# Linux bcc/BPF Tracing Tools

## Linux bcc/BPF Tracing Tools



<https://github.com/iovisor/bcc#tools> 2017





# End of Chapter 2

---

