Chapter 2: Operating-System Services

Outline

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

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
- Illustrate the process for booting an operating system
- 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.

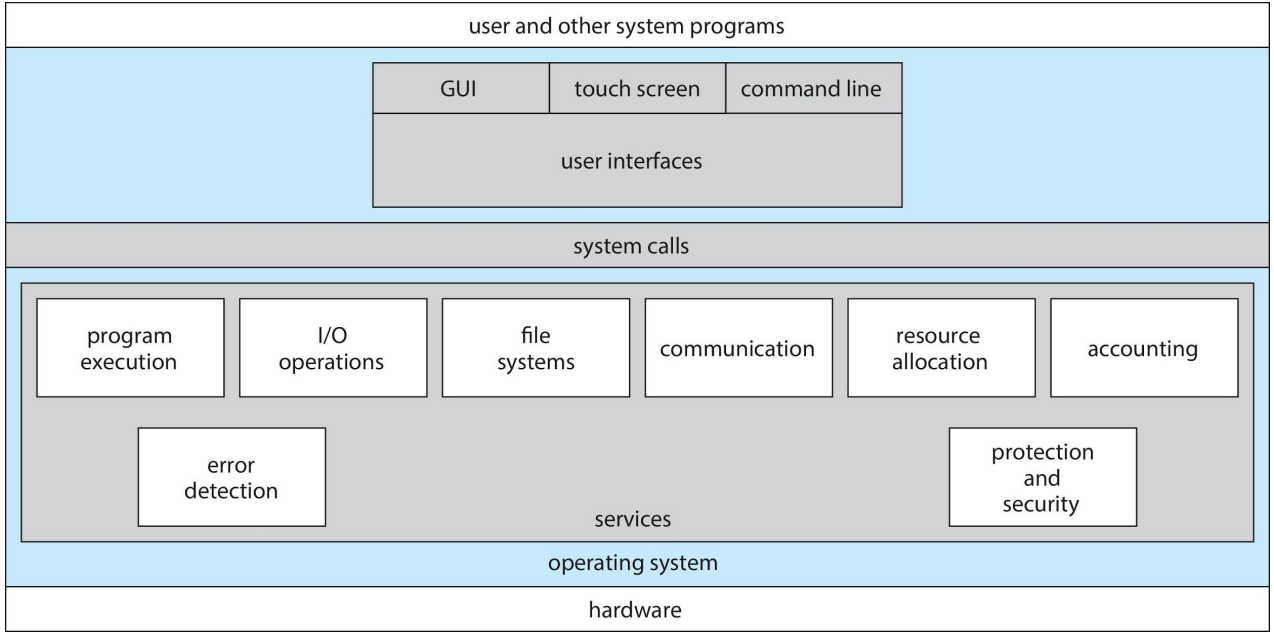
Operating System Services (Cont.)

- 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
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services

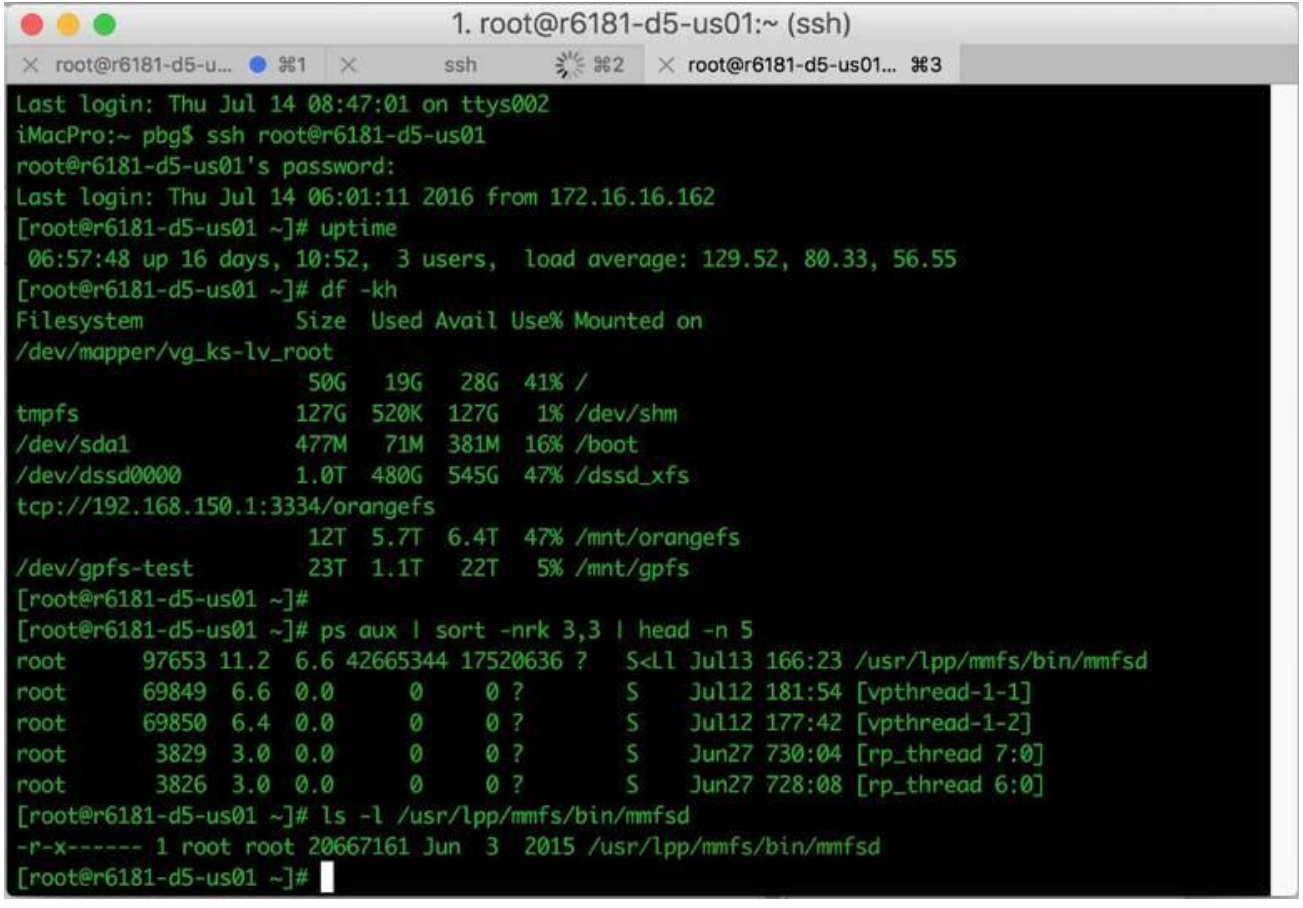


Command Line interpreter

- 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
    - **$rm file.txt**

- shell searches for the file rm,
- loads it into the memory,
- then executes it with the argument file.txt

Bourne Shell Command Interpreter



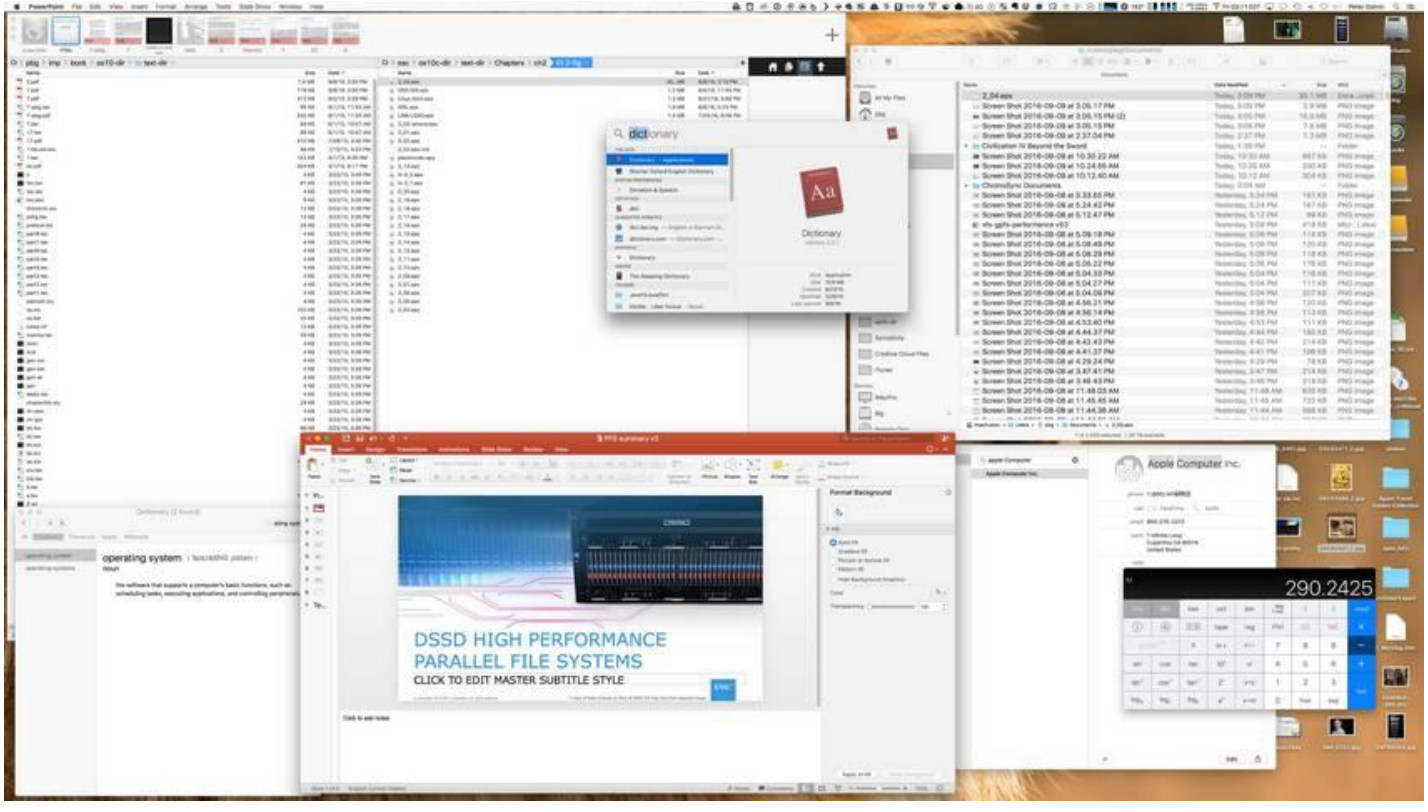
User Operating System Interface - GUI

- 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 (known as a **folder**)
  - Invented at Xerox PARC
- 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)

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

The Mac OS X GUI



System Calls (from system programming lecture)

System calls

User processes

- cannot perform privileged operations

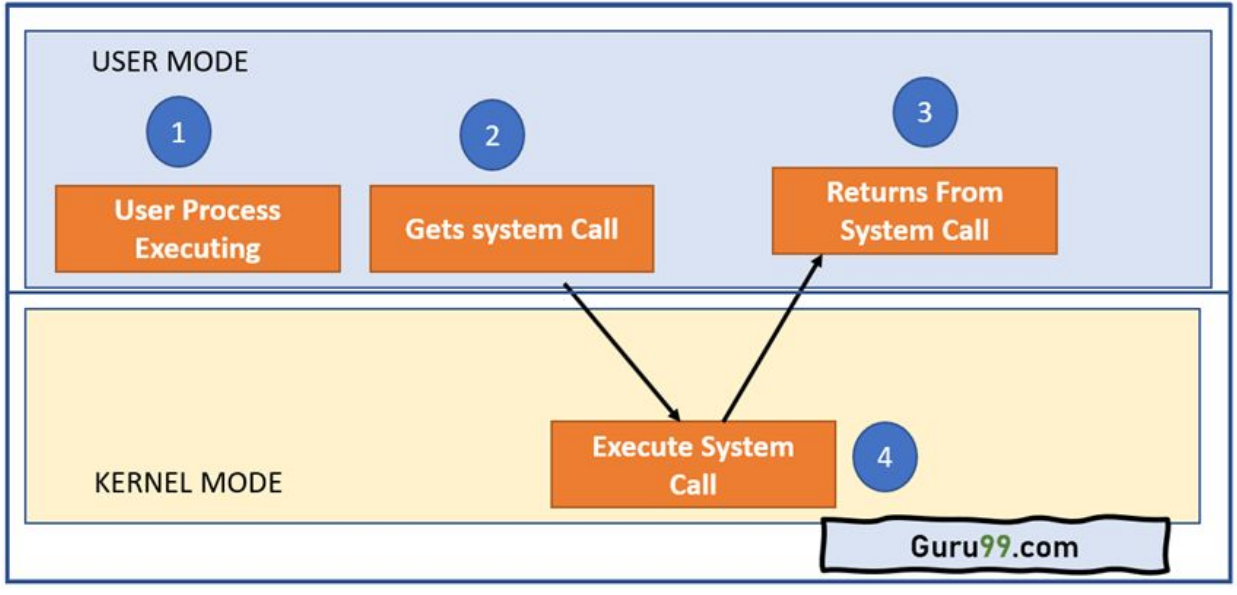
- request OS to do so on their behalf by issuing **system calls**

A **system call** is a request made by **an *active process (through traps or software interrupts)*** for a service performed by the kernel.

Examples:

- input/output (i.e., any movement of information to or from the combination of the CPU and main memory)
- creation of a new process.
- And many more https://man7.org/linux/man-pages/man2/syscalls.2.html

Why do we need a system call?



(Privileged mode)

https://www.guru99.com/system-call-operating-system.html#2

How do we switch(transition) to kernel mode?

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	-	-	
arm/OABI	swi NR	-	r0	-	-	2
arm/EABI	swi 0x0	r7	r0	r1	-	
arm64	svc #0	w8	x0	x1	-	
blackfin	excpt 0x0	P0	R0	-	-	
i386	**int $0x80**	**eax**	**eax**	**edx**	**-**	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
m68k	trap #0	d0	d0	-	-	
microblaze	brki r14,8	r12	r3	-	-	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	-	r7	
…						
x86-64	**syscall**	**rax**	**rax**	**rdx**	**-**	**5**
x32	syscall	rax	rax	rdx	-	5
xtensa	syscall	a2	a2	-	-	

https://man7.org/linux/man-pages/man2/syscall.2.html

```c
#include <unistd.h>

int main(int argc, char *argv[]) {
   write(1, "Hello World\n", 12); /* write "Hello World" to stdout */
   _exit(0);                      /* exit with error code 0 (no error) */
}
```

x86-32 equivalent assembly code

```
_start:
   movl $4, %eax    ; use the `write` [interrupt-flavor] system call
   movl $1, %ebx    ; write to stdout
   movl $msg, %ecx  ; use string "Hello World"
   movl $12, %edx   ; write 12 characters
   int $0x80        ; make system call

   movl $1, %eax    ; use the `_exit` [interrupt-flavor] system call
   movl $0, %ebx    ; error code 0
   int $0x80        ; make system call
```

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

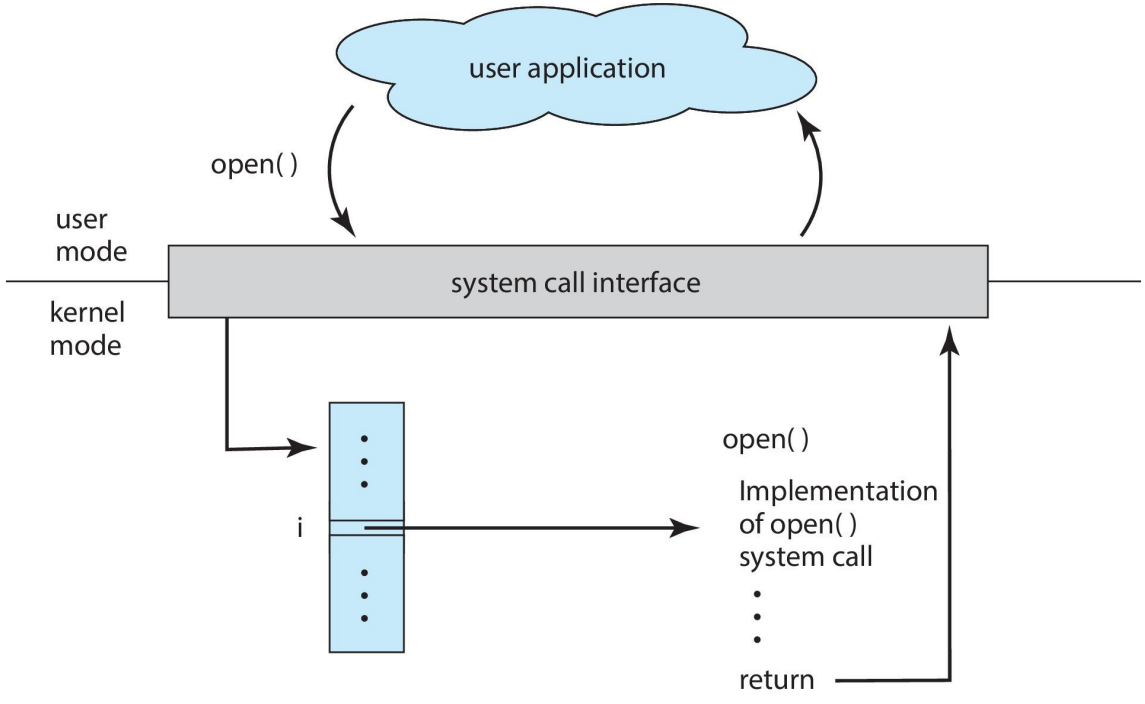
```c
#include <unistd.h>

int main(int argc, char *argv[]) {
   write(1, "Hello World\n", 12); /* write "Hello World" to stdout */
   _exit(0);                      /* exit with error code 0 (no error) */
}
```

x86-64 equivalent assembly code

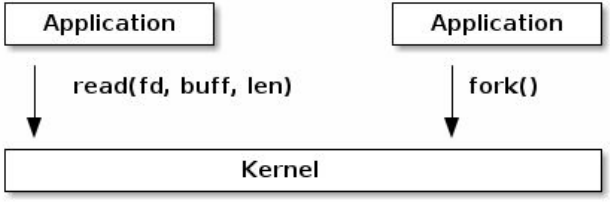
```
  _start:
     movq $1, %rax    ; use the `write` [fast] syscall
     movq $1, %rdi    ; write to stdout
     movq $msg, %rsi  ; use string "Hello World"
     movq $12, %rdx   ; write 12 characters
     syscall          ; make syscall

     movq $60, %rax   ; use the `_exit` [fast] syscall
     movq $0, %rdi    ; error code 0
     syscall          ; make syscall
```

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

API – System Call – OS Relationship

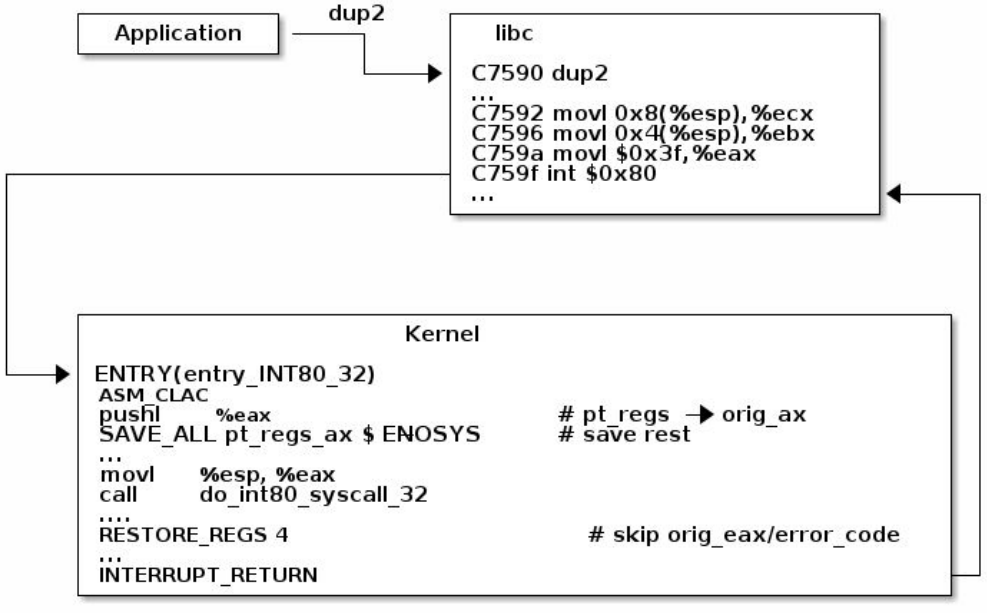


Example dup2() on x86



System calls are specific assembly instructions that

- setup information to identify the system call and its parameters
- trigger a kernel mode switch
- retrieve the result of the system call



https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html

Some definitions

Interrupt

A signal to the kernel from a hardware or a software that indicates an event need “immediate” attention

- **Hardware interrupts:** the pressing of a key on the keyboard, a movement of the mouse.
- **Software interrupts (e.g. int 0x80)**
  - requests by an application program to operating systems
  - each interrupt is associated with an interrupt handler

★ CPU switch control to an interrupt handler
★ The execution of the program resumes after the interrupt is handled

trap vs exception vs interrupt

An exception is an unexpected event occurs because of **the execution of an instruction** (synchronous event).

- Division by zero
- **CPU detects this:**
- **Exception Types**
  - Traps,
    - Breakpoint, Trap, Syscalls, division by zero
  - faults,
    - Memory faults(Page fault),
  - aborts

Hardware interrupt is an unexpected event from outside the processor.

Software interrupt (e.g., int 0x80) is generated by the application programs(similar to traps).

trap vs exception vs interrupt
see https://stackoverflow.com/questions/3149175/what-is-the-difference-between-trap-and-interrupt

Summary and warnings

```
int file_fd = open(...);
```

- We got access to a file descriptor object.

```
write(file_fd, "Hello!", 6);
```

- writes some bytes to the file descriptor object that represents a file

A system call is an operation that the kernel carries out.

- First, the operating system prepares a system call.
- Next, the kernel executes the system call to the best of its ability in kernel space (a privileged operation).

★ System calls are expensive.
  - Their cost in terms of time and CPU cycles has recently been decreased, but try to use them as sparingly as possible.

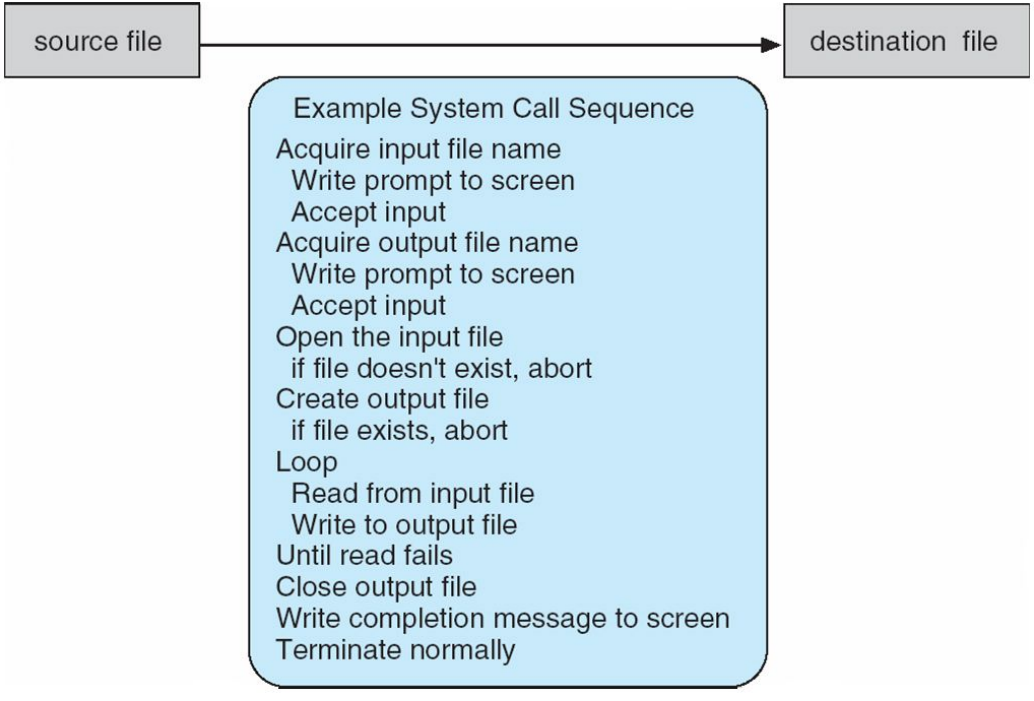
★ this operation could fail for several reasons:
  - the file is no longer valid,
  - the hard drive failed,
  - the system was interrupted etc.

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)

Example of System Calls

- System call sequence to copy the contents of one file to another file



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:

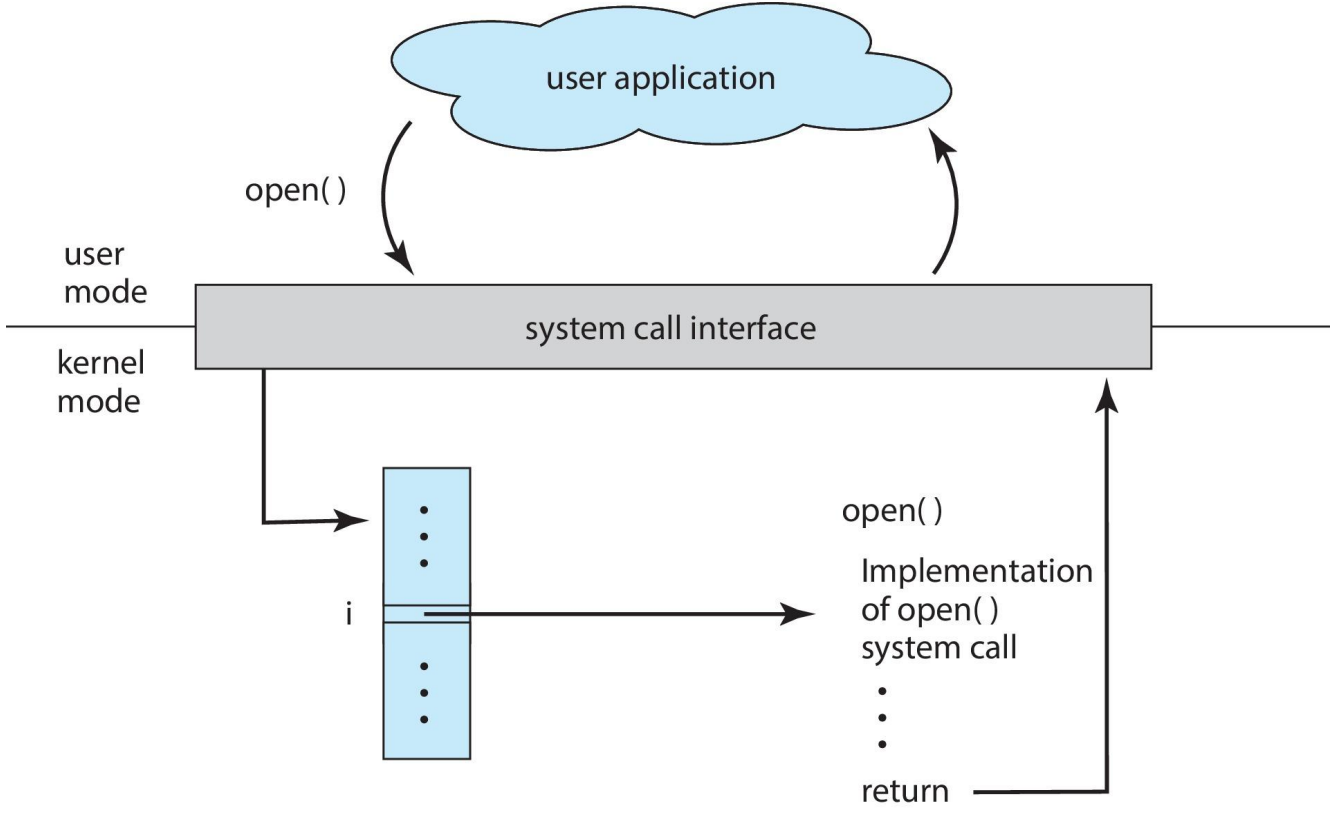
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

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)

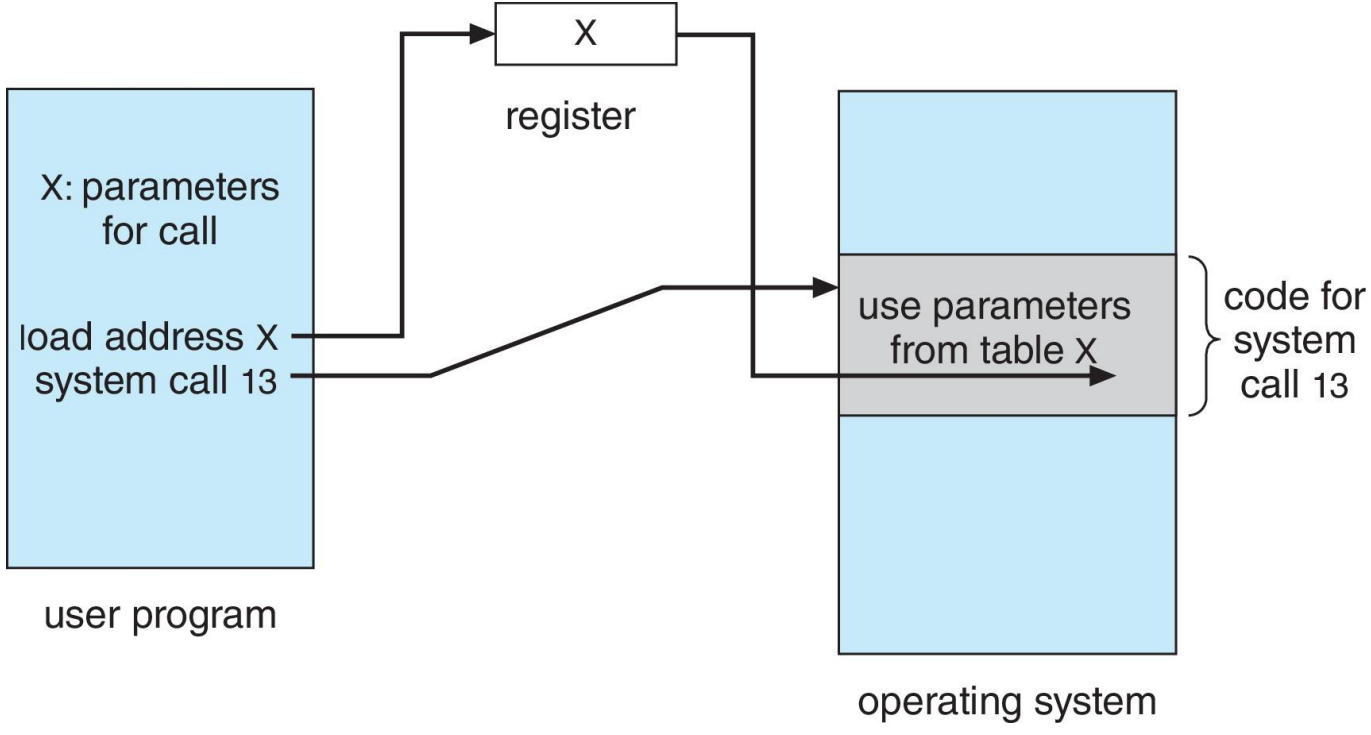
API – System Call – OS Relationship



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
- 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

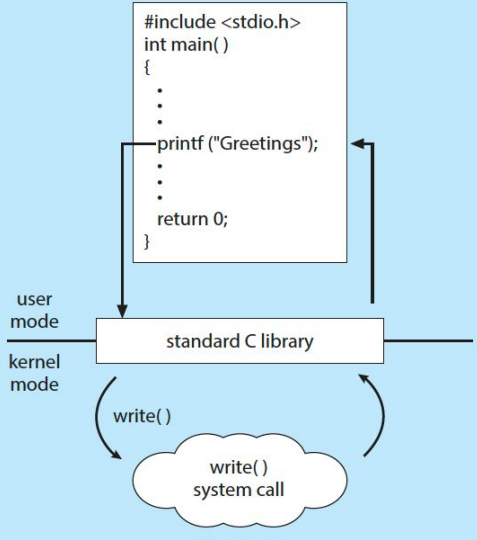


Standard C Library Example

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

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



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 (Cont.)

- File management
  - create, delete, open, close
  - 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

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

[Windows API index - Win32 apps | Microsoft Learn]

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

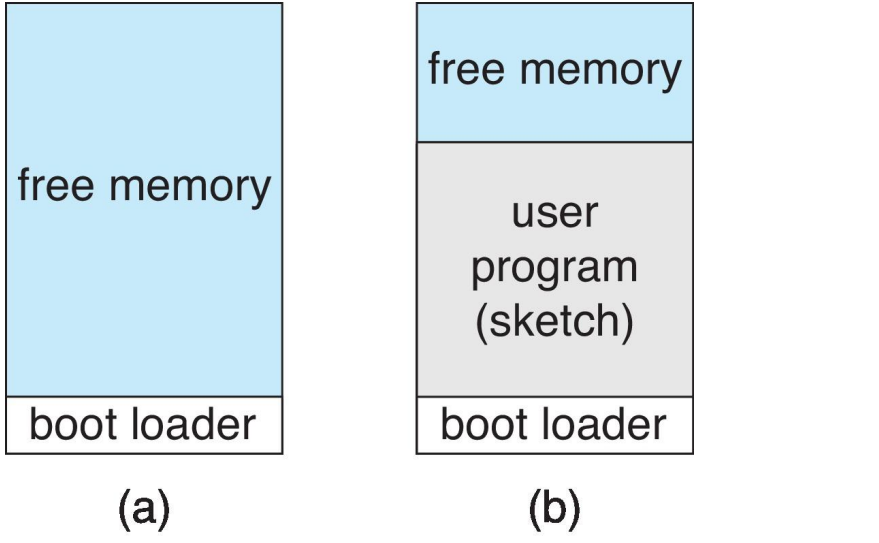
Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

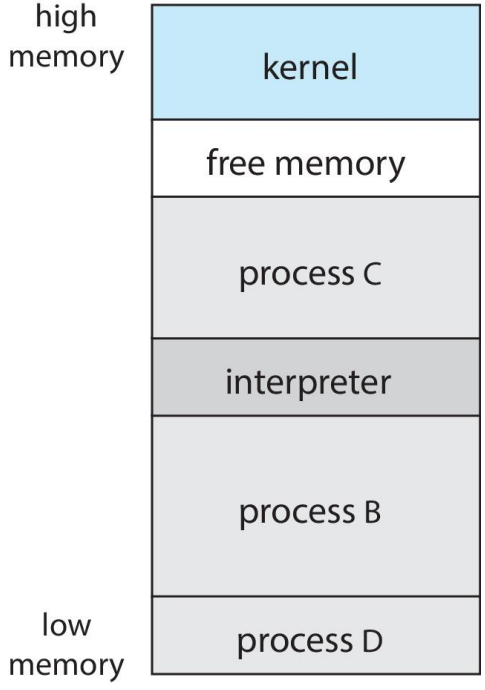
Example: Arduino

- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Bootloader (a small piece of software in arduino) loads program
  - **There is no OS.**
- Program exit -> shell reloaded



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 Services (system utilities)

- System programs provide a convenient environment for program development and execution.
- They can be divided into:
  - **File manipulation**
    - programs to create, delete, copy, rename, print, list,and generally access and manipulate files and directories.
  - **File modification**
    - Text editors to create and modify files
    - Special commands to search contents of files or perform transformations of the text

System Services (system utilities)

- System programs provide a convenient environment for program development and execution.
  - **Status information** sometimes stored in a file
    - the system for the date, time, amount of available memory or disk space, number of users, or similar
    - more complex, providing detailed performance, logging, and debugging information
    - Some systems also support a registry, which is used to store and retrieve configuration information.
  - **Programming language support**
    - Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python)
    - often provided with the operating system or available as a separate download.

System Services (system utilities)

- Program loading and execution
  - The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders.
  - Debugging systems for either higher-level languages or machine language are needed as well.

- Communications
  - These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.
  - They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another...

System Services (system utilities)

- Background services
  - Constantly running system-program processes are known as services, subsystems, or daemons
  - the network daemon, process schedulers, system error monitoring services, and print servers, etc.
  - 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
- **Application programs**
  - web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games…
  - Don't pertain to system, Run by users, Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

- Most users' view of the operating system is defined by system programs, not the actual system calls

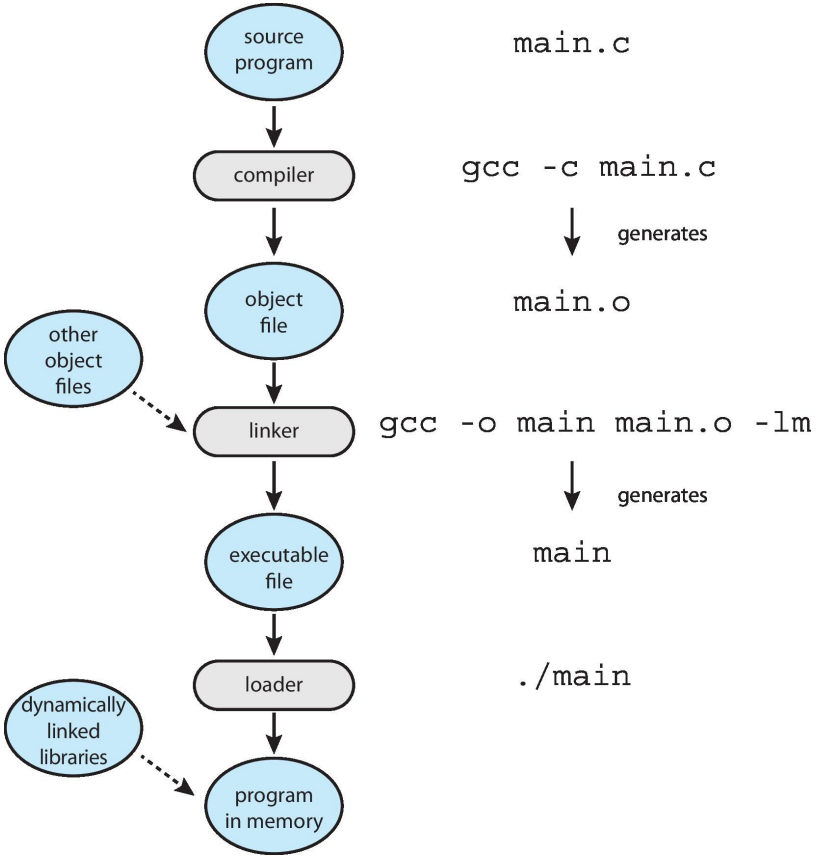
Why Applications are Operating System Specific

1st reason: system call APIs are different!

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)
- Object, executable files have standard formats, so operating system knows how to load and start them

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
- **Application Binary Interface** (**ABI**)
  - binary level interface between two softwares
  - defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

Design and Implementation of an OS

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

Design and Implementation

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

- **Policy**: **What** needs to be done?
  - Example: Interrupt after every 100 seconds

- **Mechanism**: **How** to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- 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

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually 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

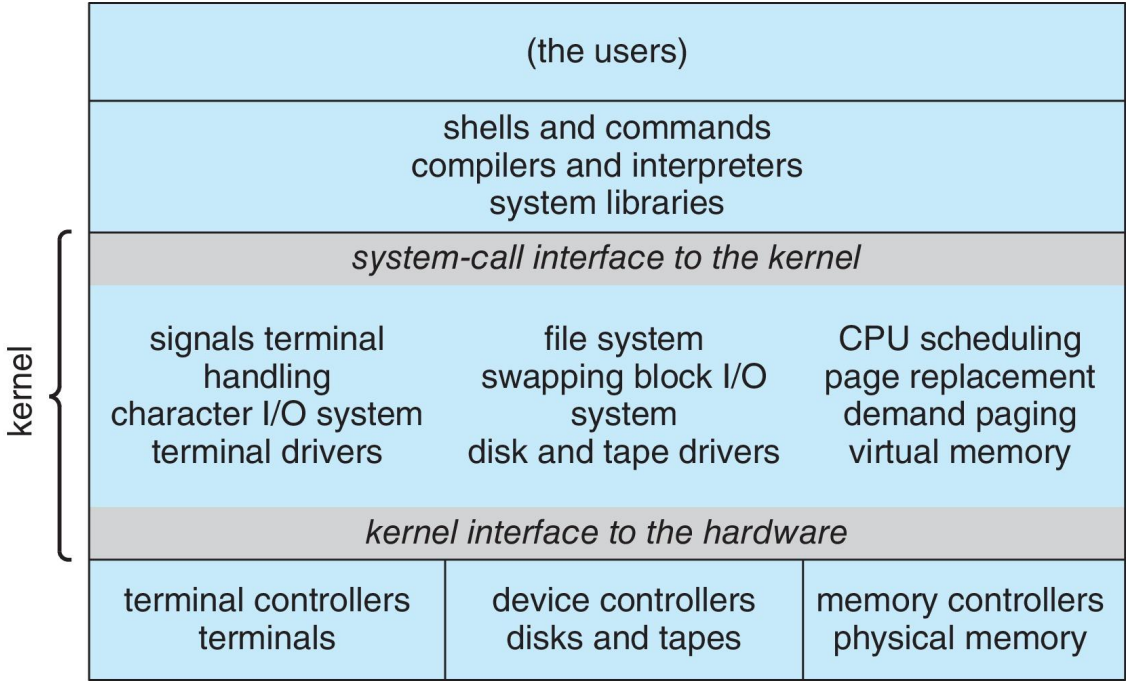
- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- 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

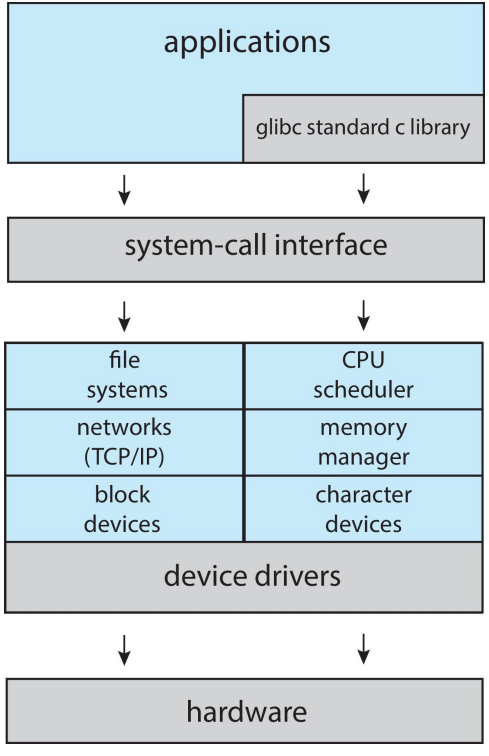
Traditional UNIX System Structure

Beyond simple but not fully layered



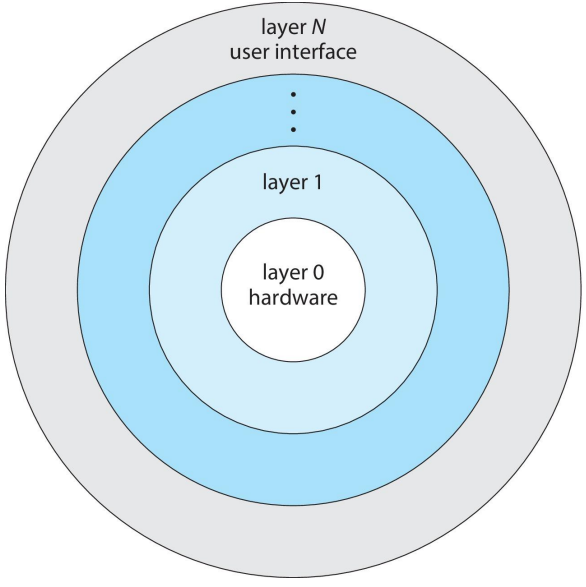
Linux System Structure

Monolithic(very early UNIX) plus modular design



Layered Approach

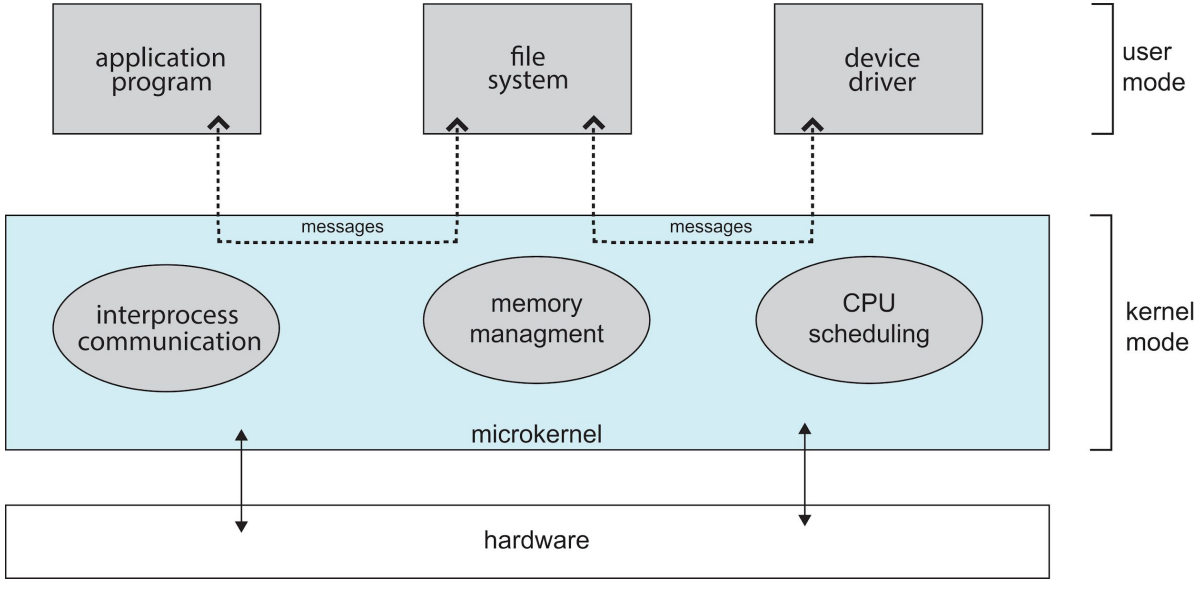
- **A system can be made modular in many ways.**
- One method is layered approach
  - The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
  - With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



Microkernels

- Moves all nonessential components 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
  - 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



- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.
- Communication is provided through message passing

❖ **Obtaining a service through a system call more expensive than monolithic kernel**
❖ **Considered more secure!**
➢ **e.g. kernel is minimal trusted computing base (TCB)**

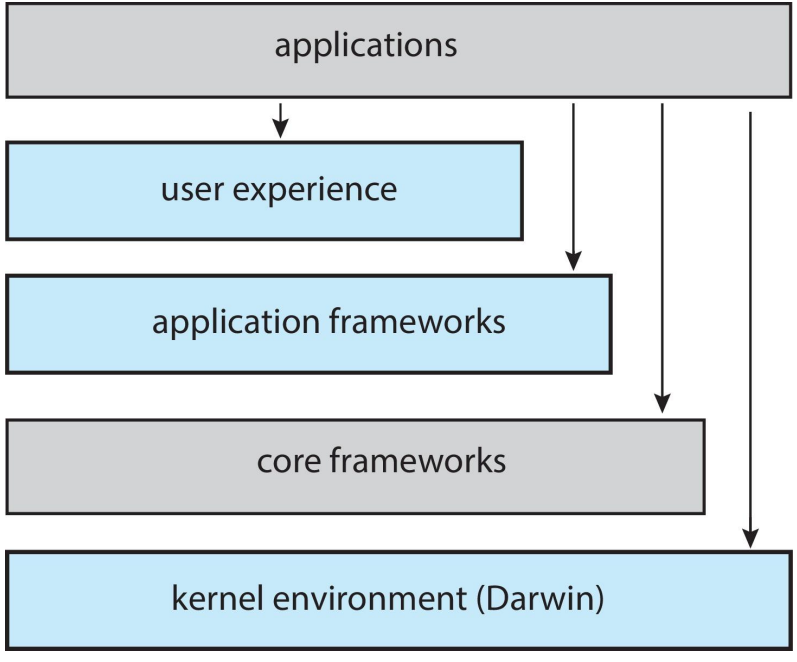
Modules

- 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
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.

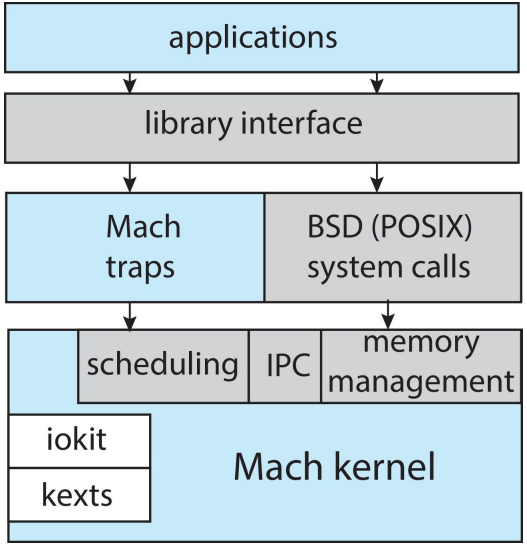
Hybrid Systems

- 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, 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**)

macOS and iOS Structure

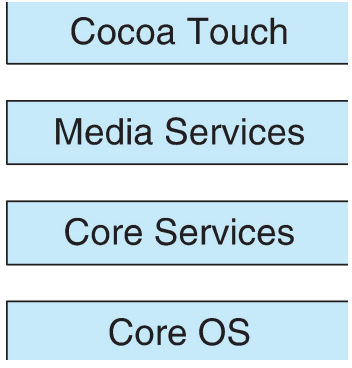


Darwin

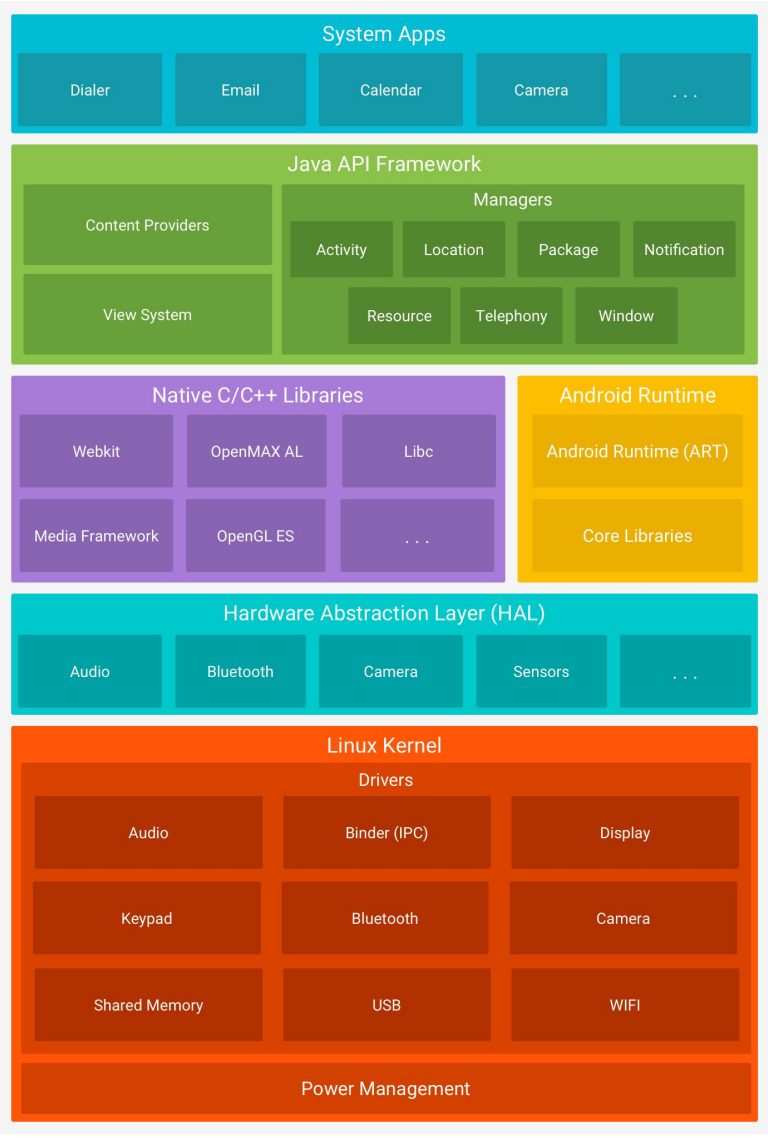

applications
library interface
Mach traps
BSD (POSIX) system calls
scheduling
IPC
memory management
iokit
kexts
Mach kernel

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

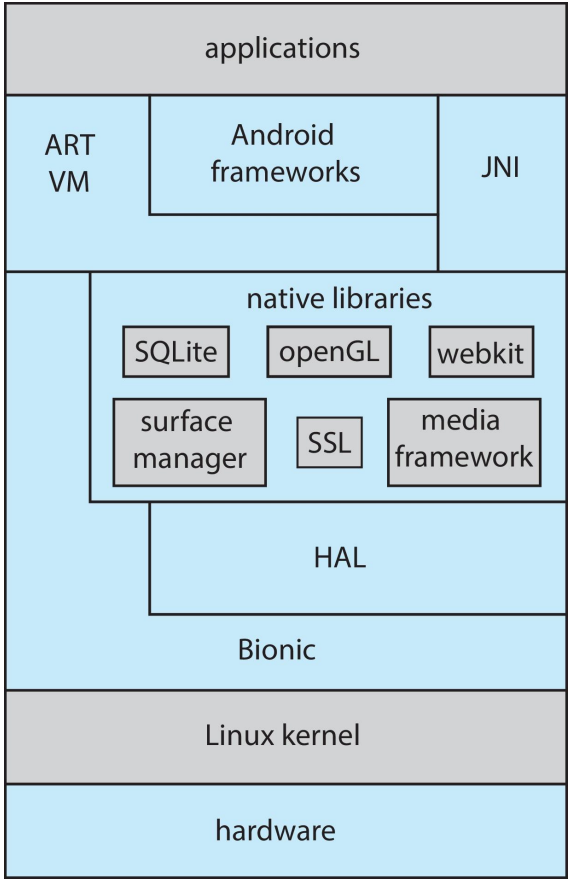


Android



https://developer.android.com/guide/platform

Android Architecture



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

- 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 "**`$make menuconfig`**"
  - generates `.config` file
- Compile the kernel using "`make`"
  - compiles kernel with the parameters defined in `.config`
  - Produces `vmlinuz`, the kernel image
  - Compile kernel modules via "**`$make modules`**"
    - with the parameters defined in `.config`
  - Install kernel modules into `vmlinuz` via "**`$make modules_install`**"
  - Install new kernel on the system via "**`$make install`**"

- To install linux
  - Download the Ubuntu ISO image from https://www.ubuntu.com/
  - Instructed the virtual machine software VirtualBox to use the ISO as the bootable medium and booted the virtual machine
  - Answered the installation questions and then installed and booted the operating system as a virtual machine

System Boot

- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, **BIOS**, 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
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface** (**UEFI**)
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode

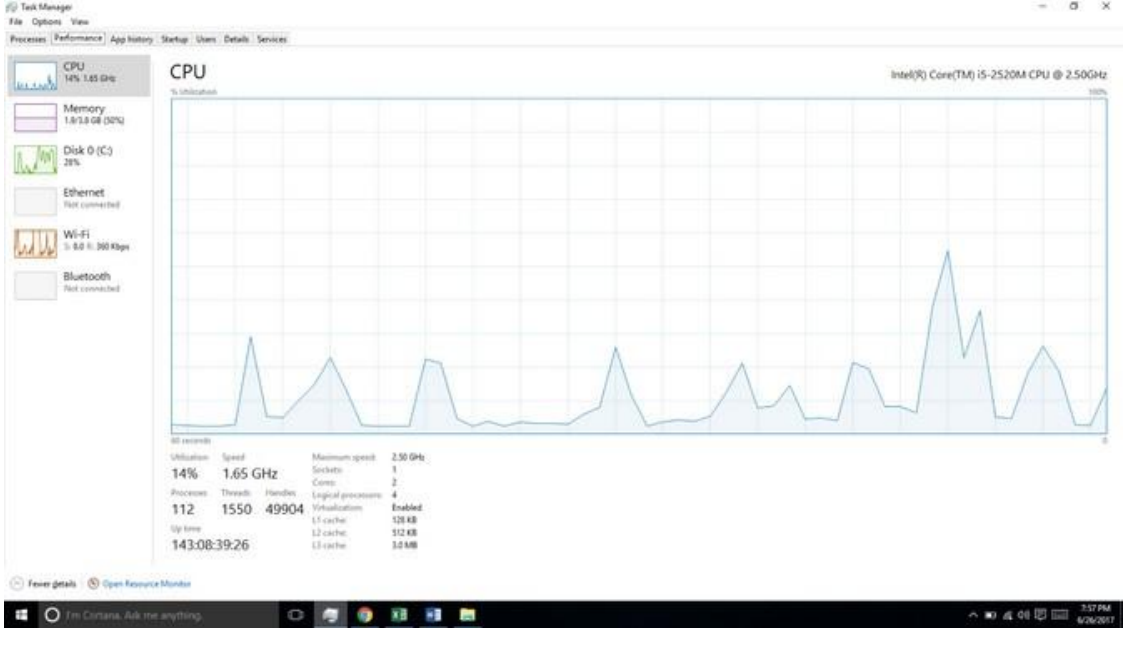
Operating-System Debugging

- **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

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, “top” program or 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
    - truss-freeBSD
    - dtrace, dtruss
  - ktrace-enable kernel tracing for processes
  - gdb – source-level debugger
  - perf – collection of Linux performance tools Perf Wiki
  - tcpdump – collects network packets

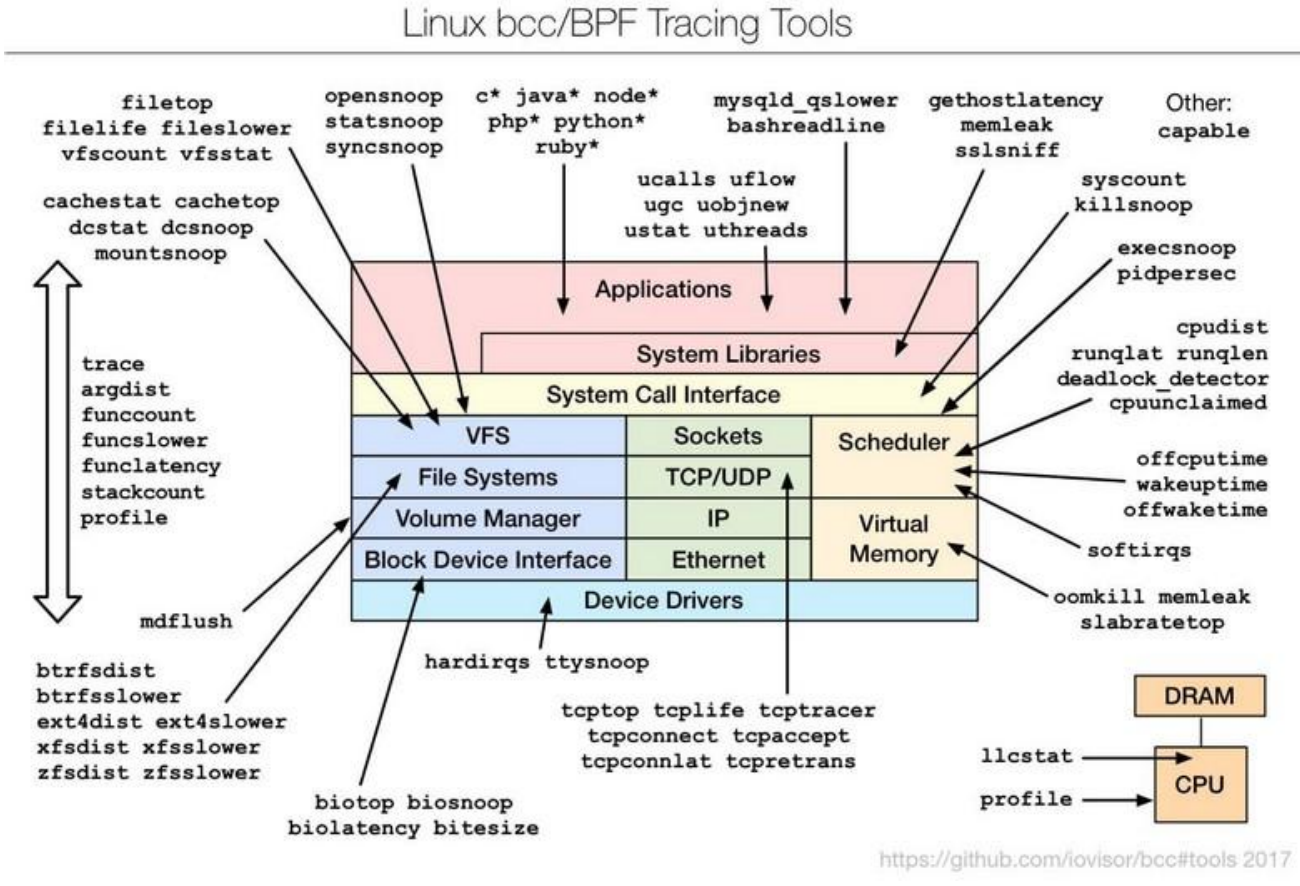
BCC

- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and an instrument their actions
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
  - See also the original DTrace
- 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
```

- Many other tools (next slide)

Linux bcc/BPF Tracing Tools



Linux kernel test

Linux Test Project

- Example system call testing (you may need in the hws):
  - https://linux-test-project.readthedocs.io/en/latest/developers/test_case_tutorial.html

Autotest

Kernel Testing Guide

End of Chapter 2