OS Structures

Bojan Nokovic

Based on: "Operating Systems Concepts", 10th Edition Silberschatz Et al.

Jan. 2021

Operating System Services

OS services provides functions that are helpful to the user:

1. User interface - CLI, GUI, touch-screen, Batch

Operating System Services

OS services provides functions that are helpful to the user:

1. User interface - CLI, GUI, touch-screen, Batch
2. Program execution - load a program into memory and to run that program, end execution, either normally or abnormally

Operating System Services

OS services provides functions that are helpful to the user:

1. User interface - CLI, GUI, touch-screen, Batch
2. Program execution - load a program into memory and to run that program, end execution, either normally or abnormally
3. I/O Operation - may involve a file or an I/O device

Operating System Services

OS services provides functions that are helpful to the user:

1. User interface - CLI, GUI, touch-screen, Batch
2. Program execution - load a program into memory and to run that program, end execution, either normally or abnormally
3. I/O Operation - may involve a file or an I/O device
4. File-system manipulation - read and write files and directories, create and delete them, search them, list file Information, permission management

Operating System Services

5. Communications - via shared memory or through message passing (packets moved by the OS)

Operating System Services

5. Communications - via shared memory or through message passing (packets moved by the OS)
6. Error detection - on CPU, memory hardware, I/O devices, user program; each type of error, OS should take the appropriate action

Operating System Services

5. Communications - via shared memory or through message passing (packets moved by the OS)
6. Error detection - on CPU, memory hardware, I/O devices, user program; each type of error, OS should take the appropriate action
7. Resource allocation - when multiple users or multiple jobs running concurrently, resources must be allocated to each of them

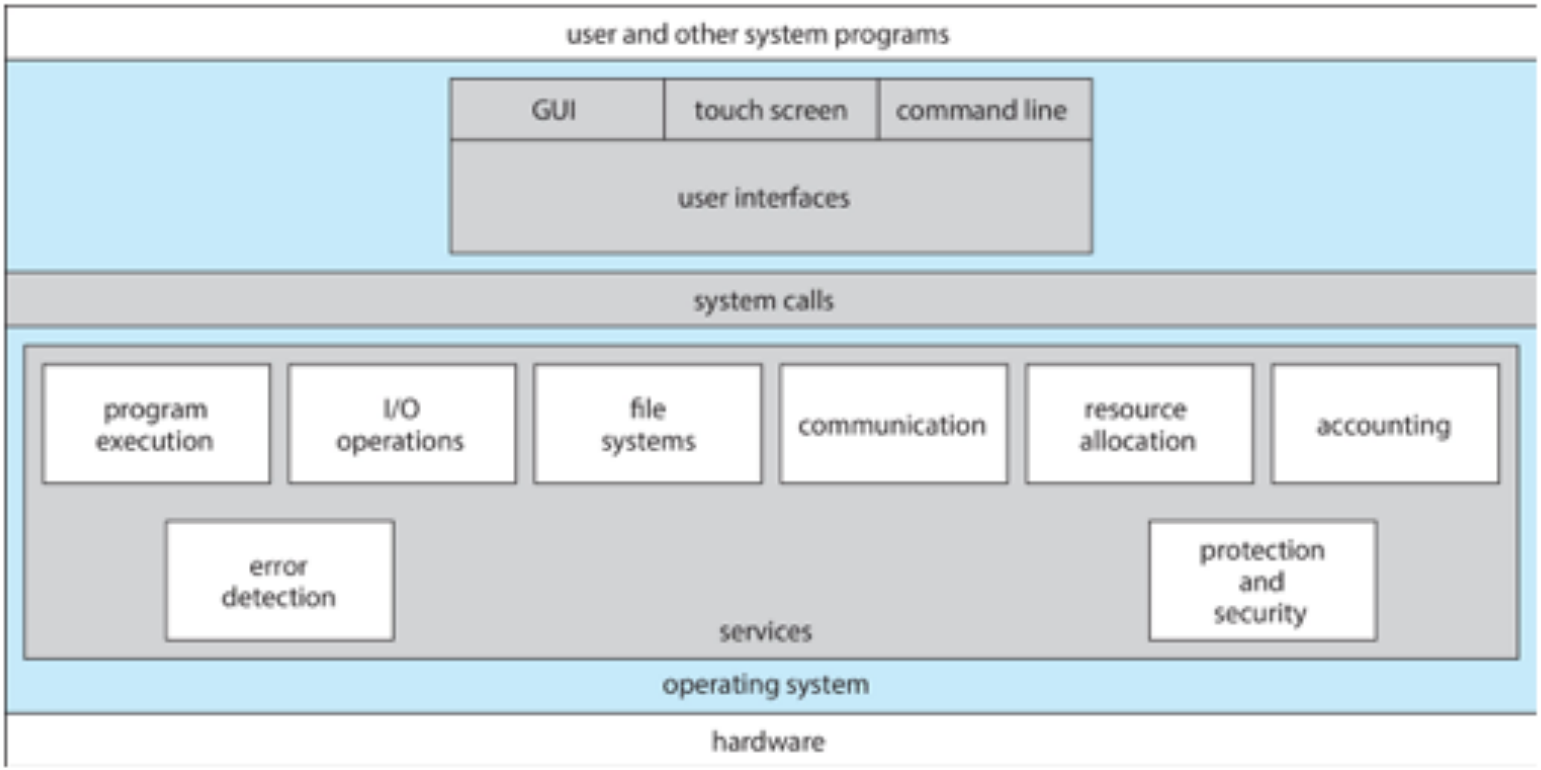
Operating System Services

5. Communications - via shared memory or through message passing (packets moved by the OS)
6. Error detection - on CPU, memory hardware, I/O devices, user program; each type of error, OS should take the appropriate action
7. Resource allocation - when multiple users or multiple jobs running concurrently, resources must be allocated to each of them
8. Logging - keep track of user's computer resources usage

Operating System Services

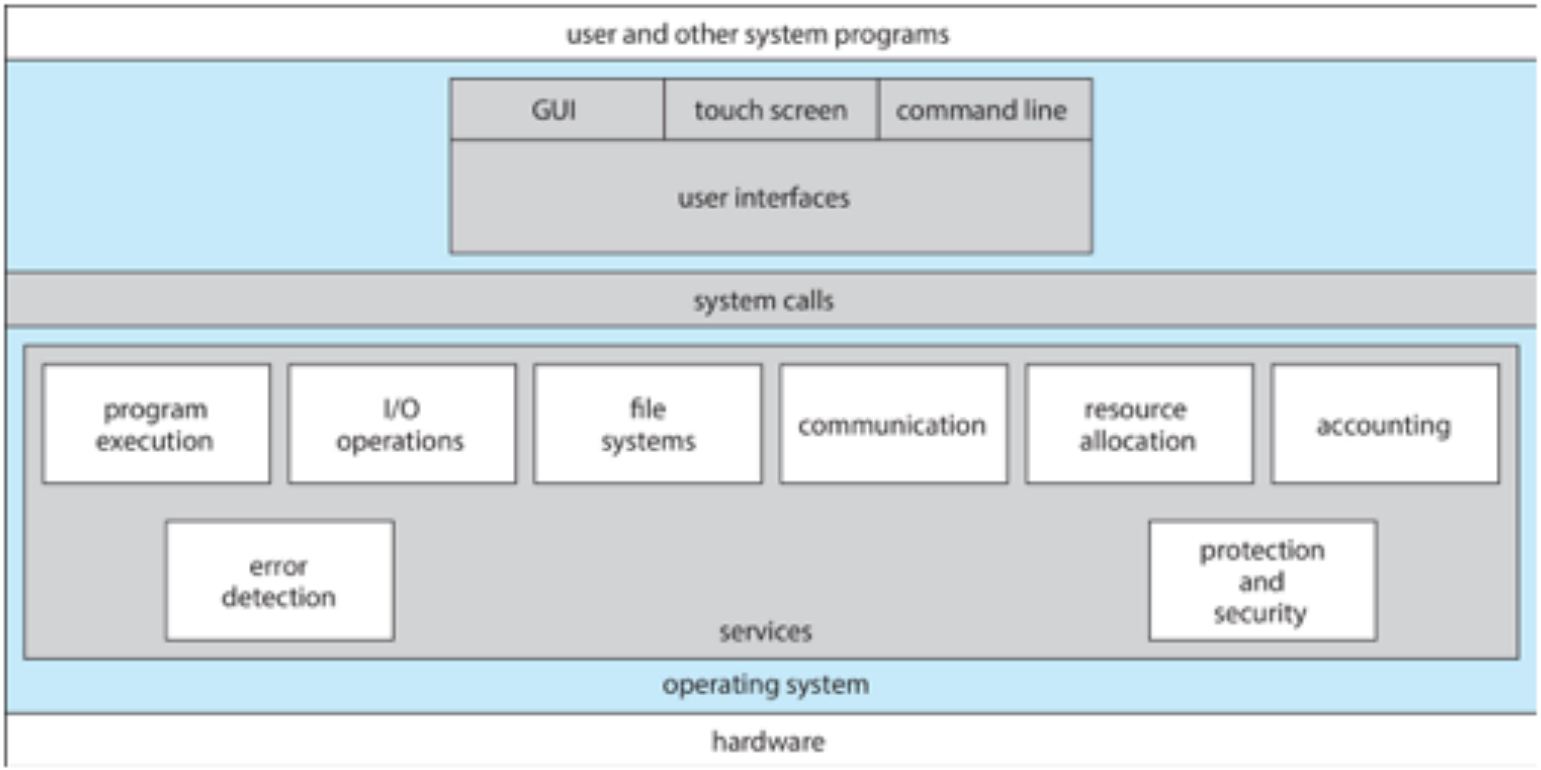
5. Communications - via shared memory or through message passing (packets moved by the OS)
6. Error detection - on CPU, memory hardware, I/O devices, user program; each type of error, OS should take the appropriate action
7. Resource allocation - when multiple users or multiple jobs running concurrently, resources must be allocated to each of them
8. Logging - keep track of user's computer resources usage
9. Protection and security - ensures all access to system resources is controlled, and requires user authentication

A View of Operating System Services



What is accounting service?

A View of Operating System Services

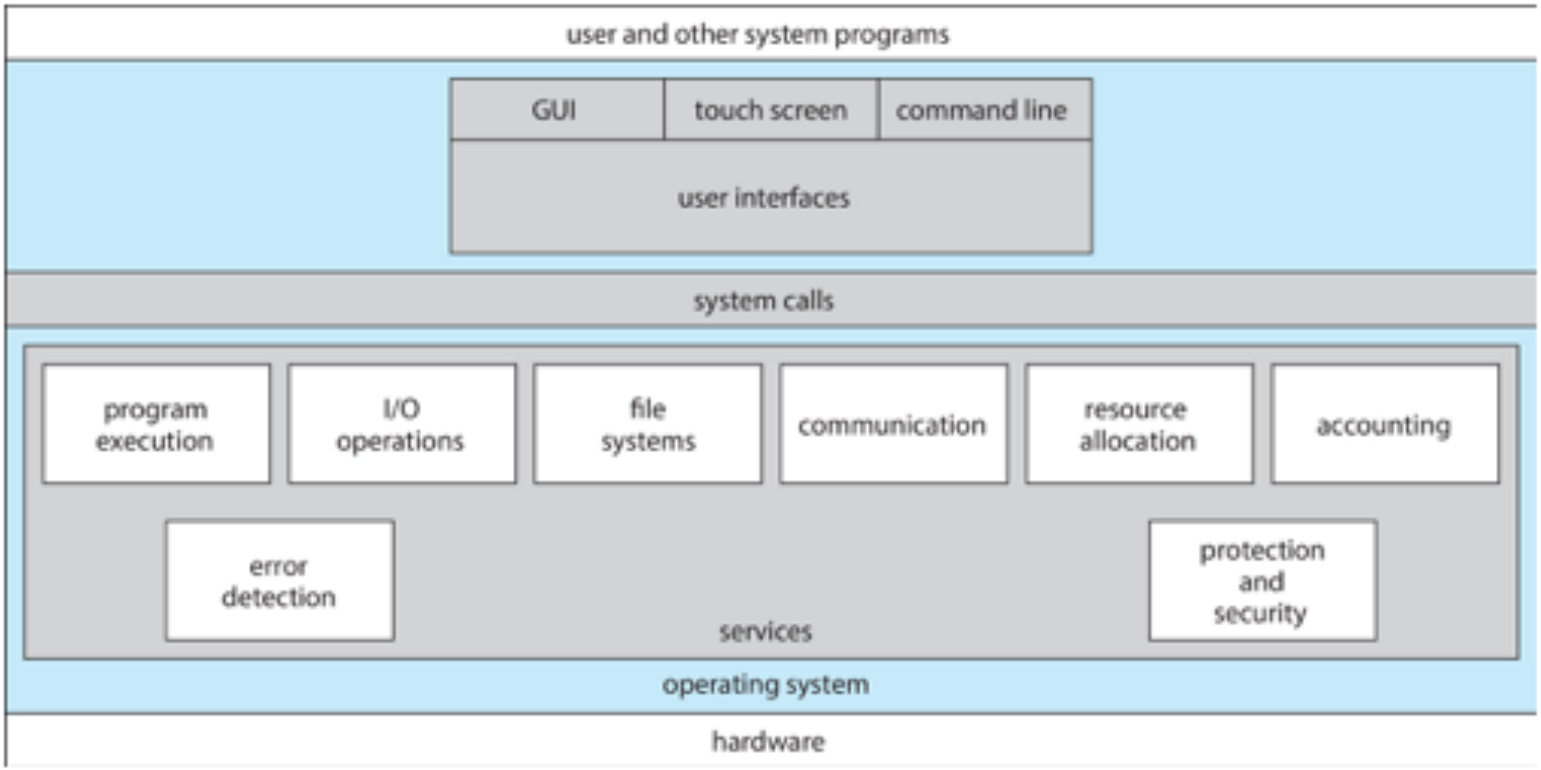


What is accounting service?

Record keeping may be used for accounting (so that users can be billed)

What are system calls?

A View of Operating System Services



What is accounting service?

Record keeping may be used for accounting (so that users can be billed)

What are system calls?

System calls provide an interface to the services made available by an operating system.

System Calls

Programming interface to the services provided by the OS

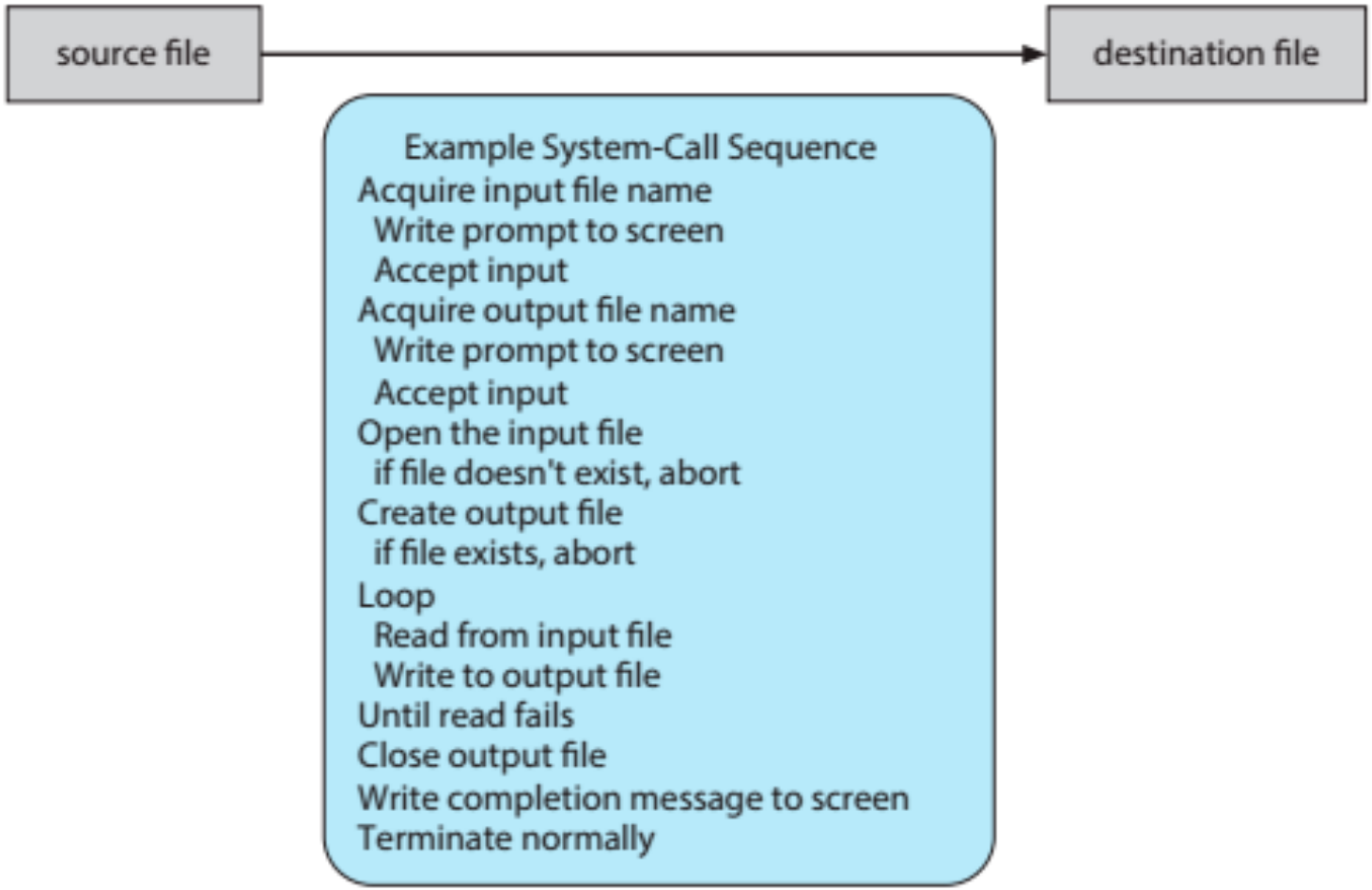
Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use

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)
- Java API for the Java virtual machine (JVM)

Example of System Calls

The UNIX command: `cp in.txt out.txt`



Example of API - Reading Data Into Buffer

The following example reads data from the file associated with the file descriptor `fd` into the buffer pointed to by `buf`.

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_r;
int fd;
...
nbytes = sizeof(buf);
bytes_r = read(fd, buf, nbytes);
...
```

On Linux and other Unix-like operating systems `man` is the interface used to view the system's reference manuals.

`http://pubs.opengroup.org/onlinepubs/9699919799/`

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

- Q: What is an abstraction in SE?

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

- Q: What is an abstraction in SE?
  A: Purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure.

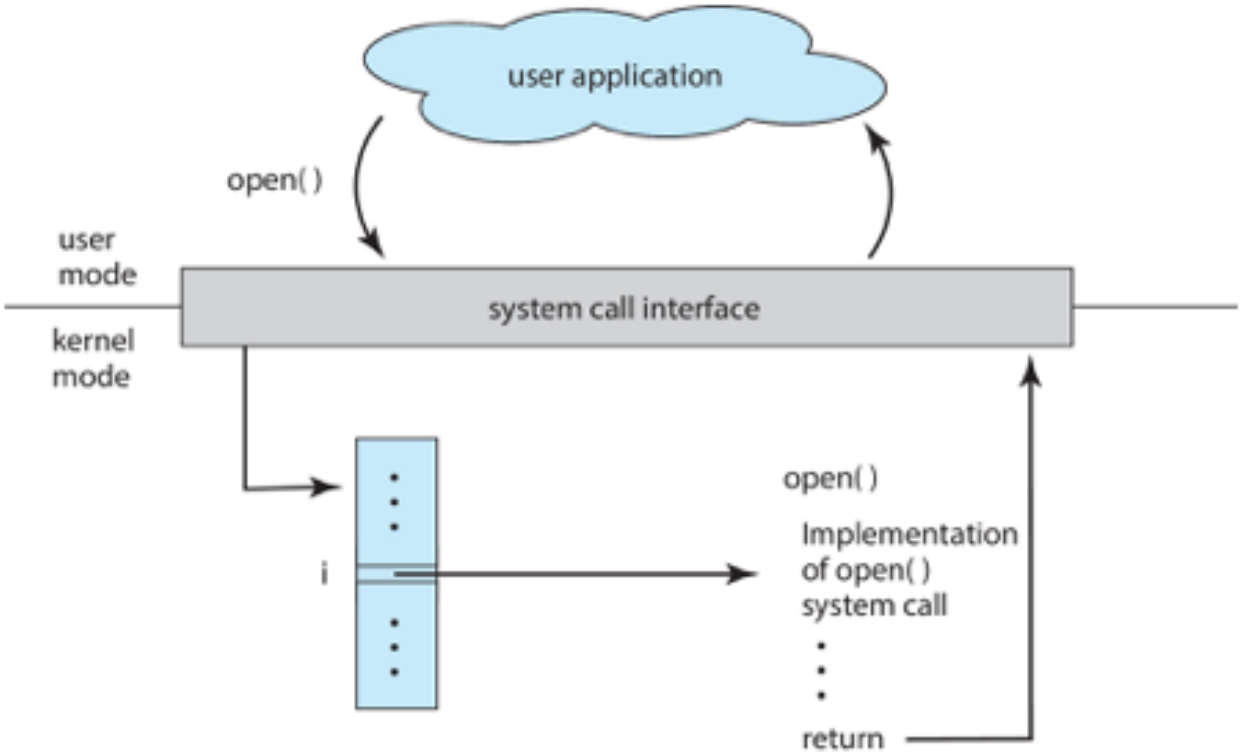
The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. - Edsger W. Dijkstra

System Call Implementation

The caller needs to know nothing about how the system call is implemented

- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface is 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



"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts." - Benjamin C. Pierce

System Call Parameter Passing

Three general methods used to pass parameters to the OS

1. Register - pass the parameters in registers
2. Block - parameters stored in a block in memory, and address of block passed as a parameter in a register
3. Stack - parameters pushed the stack by the program and popped off the stack by the operating system

A restriction: Practically block and stack methods do not limit the number or length of parameters being passed.

Types of System Calls

1. Process control
2. File management
3. Device management
4. Information maintenance
5. Communications
6. Protection

Types of System Calls: 1

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

Unix system calls: `fork(), exit(), wait(),..`

Types of System Calls: 2, 3

File management

- create, delete, open, close file
- read, write, reposition
- get and set file attributes

Unix: `open(), read(), write(), close(),..`

Device management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes, logically attach or detach devices

Unix: `ioctl(), read(), write(),...`

Types of System Calls: 4, 5

Information maintenance

- get time or date, set time or date, get system data, set system data
- get and set process, file, or device attributes

Unix: `getpid(), alarm(), sleep(),..`

Communications

- create, delete communication connection, transfer status information, attach and detach remote devices
- send, receive messages if message passing model; create and gain access to memory regions in shared-memory model

Unix: `pipe(), shm_open(), mmap(),...`

Types of System Calls: 6

Protection

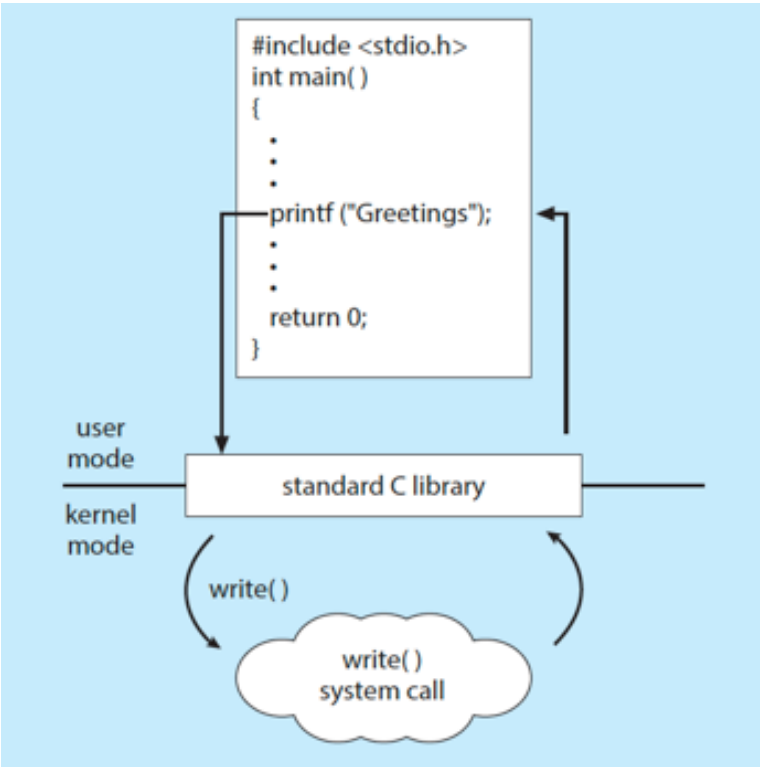
- Control access to resources
- Get and set permissions
- Allow and deny user access

Unix: `chmod()`, `umask()`, `chown()`,..

System Call Interface

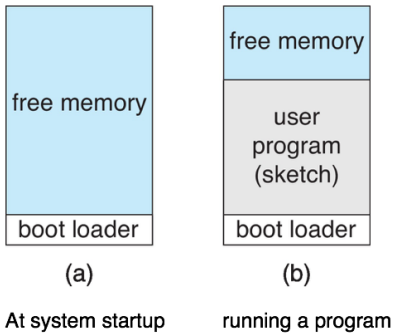
The standard C library provides a portion of the system-call interface.

Example: C program invoking `printf()` library call, which calls `write()` system call



Example Arduino

- Single-tasking, Single memory space
- No operating system, Boot loader loads program (sketch) loaded via USB into flash memory



$39.95

Microcontroller ATmega328
Operating Voltage 5V
Input Voltage (recommended) 7-12V
Input Voltage (limits) 6-20V
Digital I/O Pins 14 (of which 6 provide PWM output)
Analog Input Pins 6
DC Current per I/O Pin 40 mA
DC Current for 3.3V Pin 50 mA
Flash Memory 32 KB (ATmega328) of which 0.5 KB used by bootloader
SRAM 2 KB (ATmega328)
EEPROM 1 KB (ATmega328)
Clock Speed 16 MHz

1KB = 1024 bytes

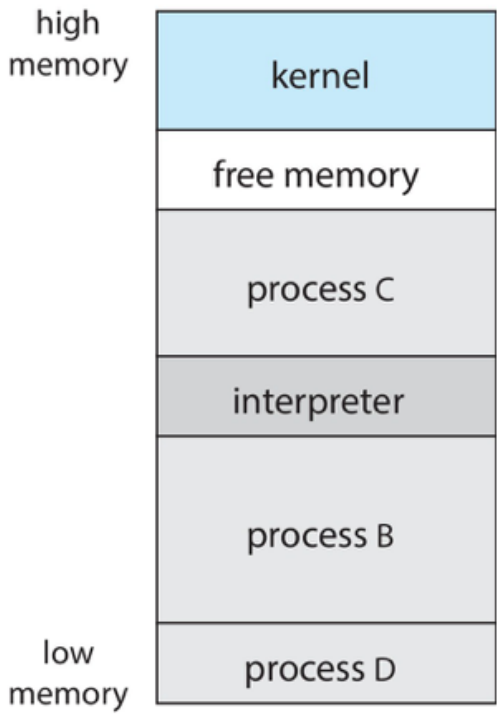
Example FreeBSD

Unix-like OS via Berkeley Software Distribution

Shell executes `fork()` to create process

- Executes `exec()` to load program into process
- Shell waits for process to terminate or continues with user commands

Process exits with:
(1) code = 0 - no error
(2) code > 0 - error code



Why Applications are Operating System Specific?

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) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc

Operating System Design Challenges

Internal structure of different Operating Systems can vary widely

Affected by choice of hardware, type of system

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

Implementation

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 Monolithic Structure

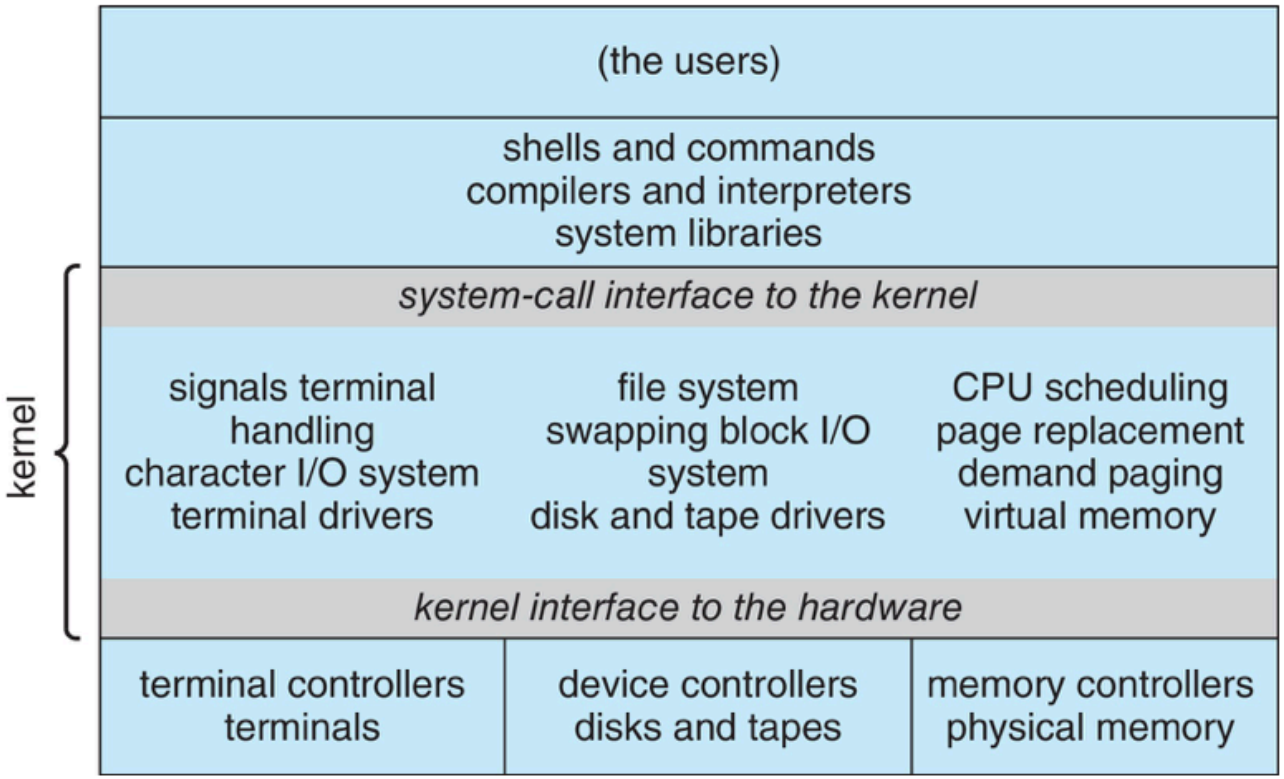
The original UNIX OS consists of two separable parts:

1) Systems programs

2) 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

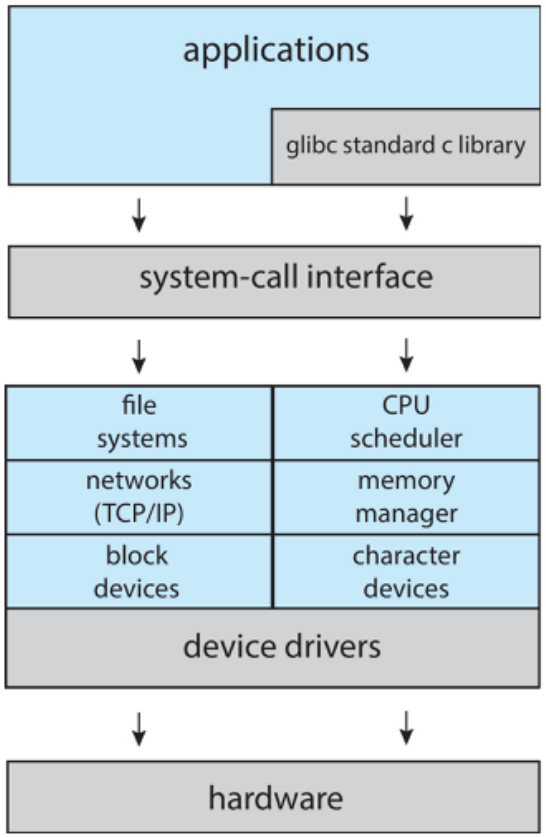
Beyond simple but not fully layered!



- ☹ Difficult to implement and extend.
- ☺ Little overhead in the system-call interface.
- ☺ Communication within the kernel is fast.

Linux System Structure

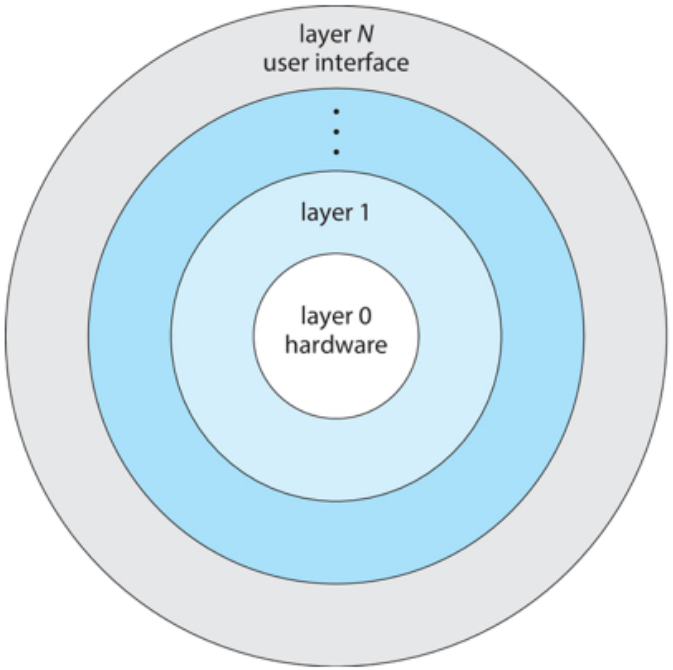
Monolithic plus modular design.



☺ Changes in one component affect only that component.

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
- Each layer uses functions (operations) and services of only lower-level layers



☺ Simplifies debugging and system verification.
☺ Each layer hides the existence of certain data structures, operations, and hardware from higher level layers.
☹ Performance of such systems is poor.

Microkernels

- Moves as much from the kernel into user space. Mach is example of microkernel; Mac OS X kernel Darwin is partly based on Mach
- Communication takes place between user modules using message passing

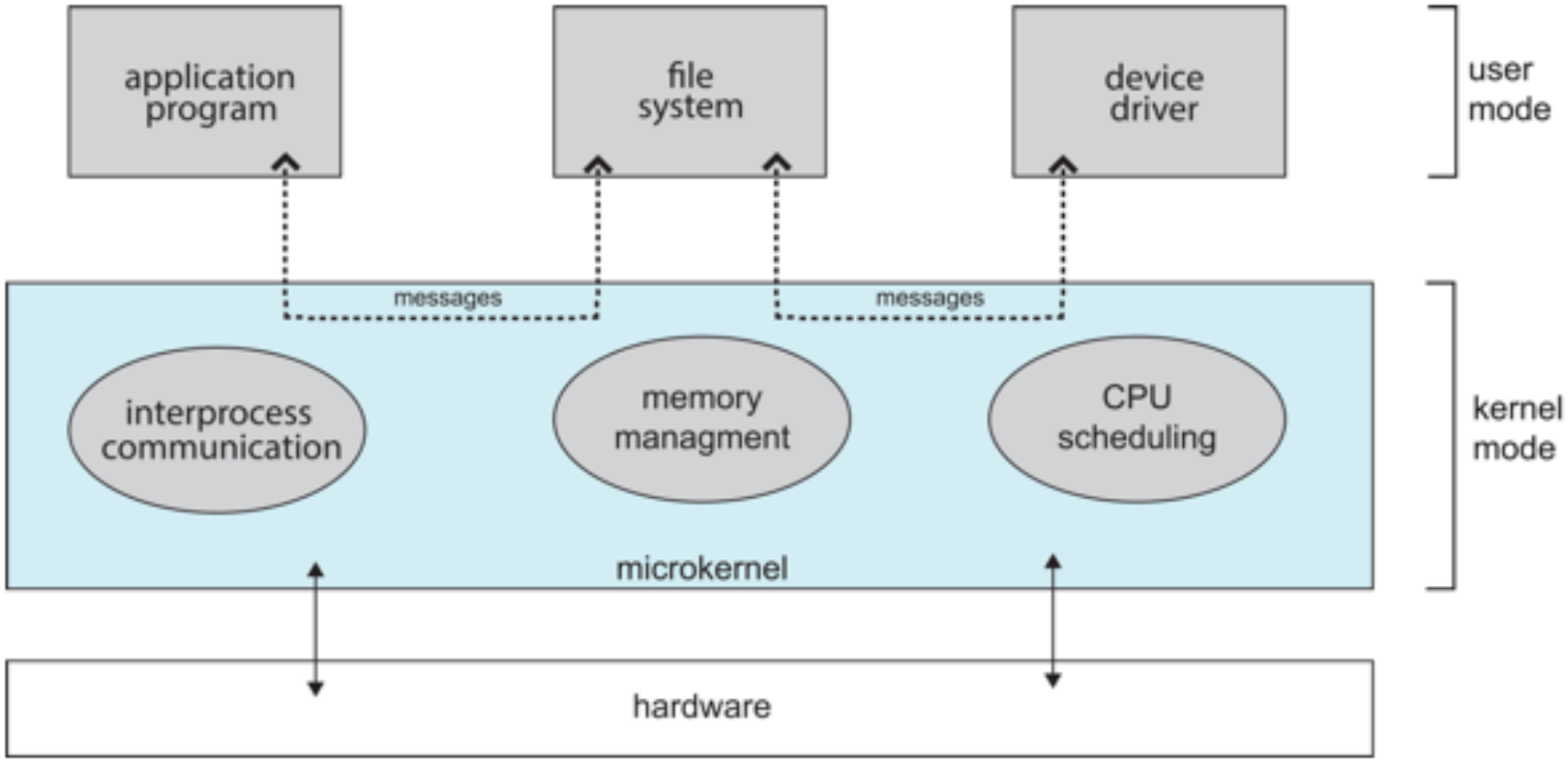
Benefits ☺

- (1) Easier to extend a microkernel (2) easier to port the operating system to new architectures (3) more reliable-less code is running in kernel mode (4) more secure

Detriments ☹

- Performance overhead of user space to kernel space communication

Microkernel System Structure



Assigns only a few essential functions to the kernel.

Nonessential components from the kernel implemented as user-level programs.

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.

This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.

Hybrid Systems

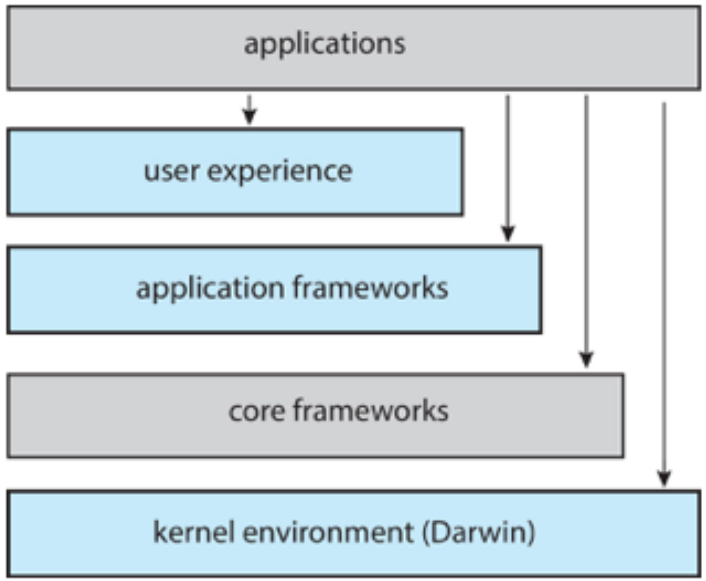
Most modern operating systems are actually not one pure model

- 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

Hybrid Systems

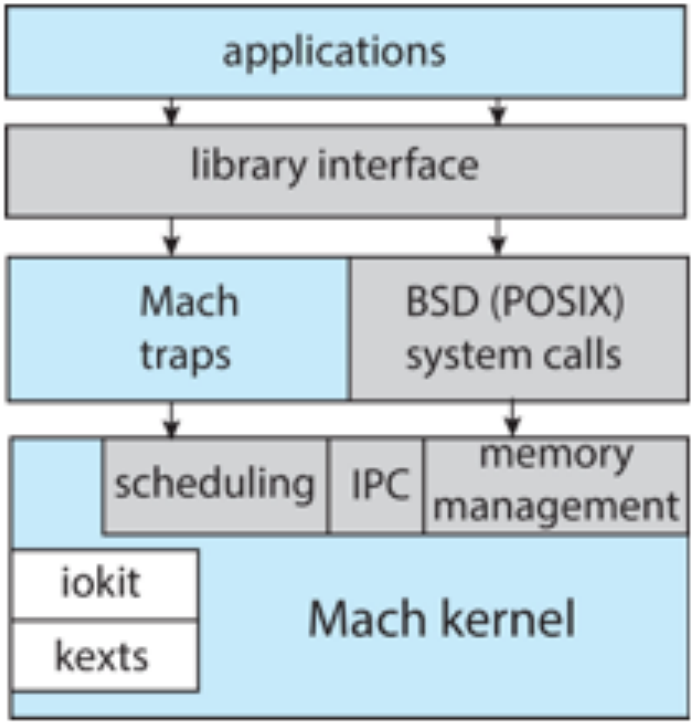
Kernel consisting of Mach-based Darwin microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)



- macOS - Aqua, iOS - Springboard
- App. framework Cocoa provide an API for the Objective-C and Swift programming languages
- Core frameworks support graphics and media - Quicktime, OpenGL

Darwin

Darwin consists primarily of the Mach microkernel and the UNIX kernel.

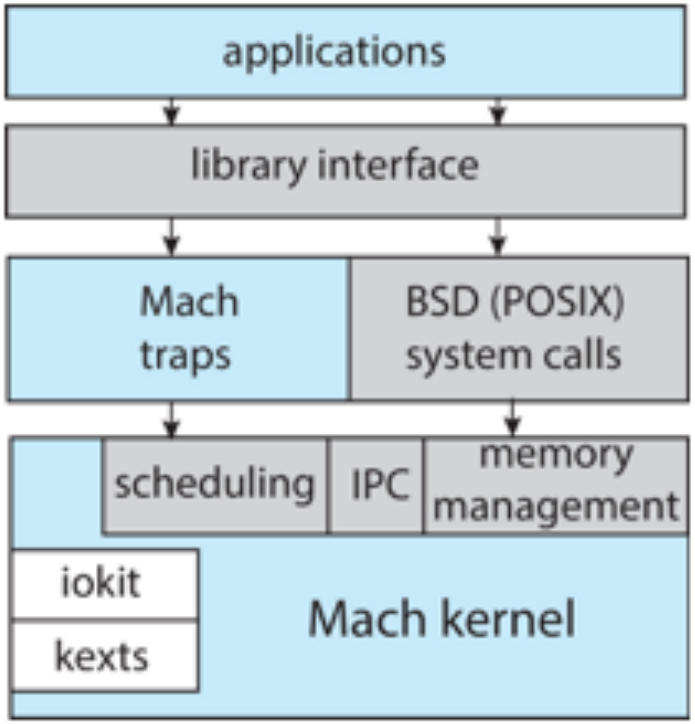


Two system-call interfaces: Mach system calls (trap) and BSD UNIX system calls.

Q: What is IPC?

Darwin

Darwin consists primarily of the Mach microkernel and the UNIX kernel.



Two system-call interfaces: Mach system calls (trap) and BSD UNIX system calls.

Q: What is IPC? A: Interprocess communication

Android

Developed by Open Handset Alliance (mostly Google) - open source

Similar stack to iOS (Apple mobile OS for iPhone, iPad)

Based on Linux kernel

- Provides process, memory, device-driver management
- Adds power management

Android

Originally runtime environment included core set of libraries and Dalvik virtual machine (VM).

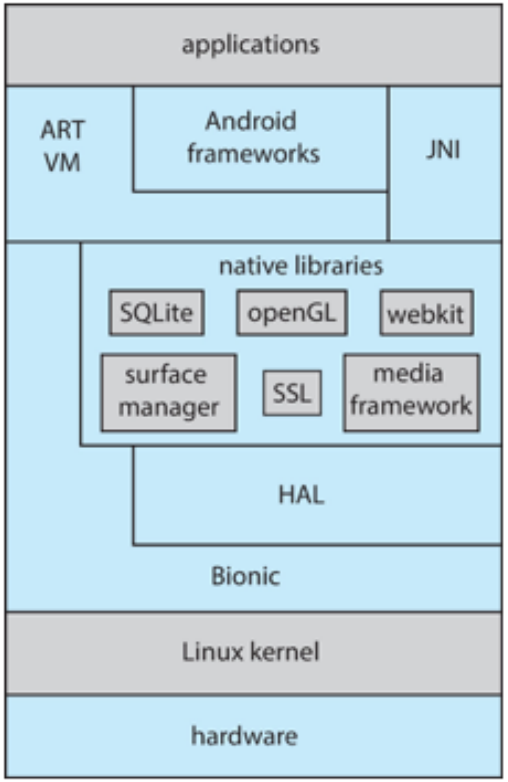
Since 2015 the Dalvik runtime is no longer maintained or available. Its byte-code format is now used by Android Run Time ART VM.

Apps developed in Java plus Android API

- Java class files compiled to Java bytecode then translated to executable than runs in ART VM

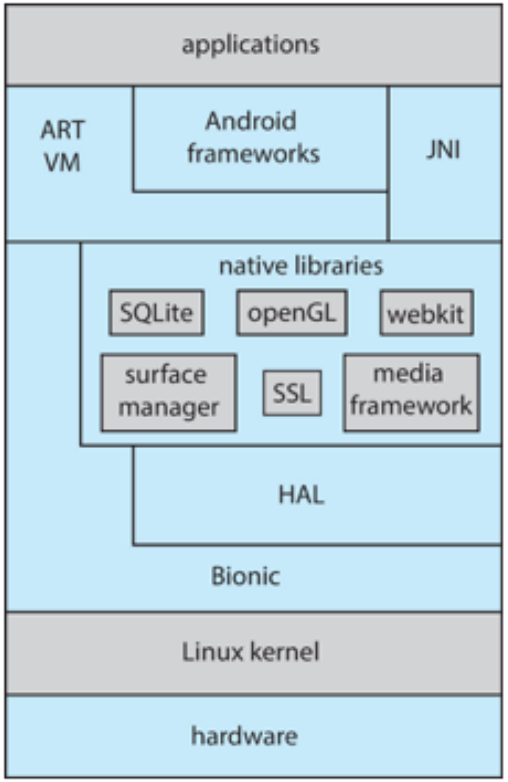
Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android



Q: What is HAL?

Android



Q: What is HAL?
A: Hardware Abstraction Layer - maps between generic hardware commands and responses and those unique to a specific platform. By abstracting all hardware, such as the camera, GPS chip, the HAL provides applications with a consistent view independent of specific hardware.

Android Runtime

Q: What is JNI?

Every Android application runs in its own process with its own instance of the ART VM

ART VM executes files in the Dalvik Executable (.dex) format

Component includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language

To execute an operation the ART VM calls on the corresponding C/C++ library using the Java Native Interface (JNI)

Power Management

Android addes two features to the Linux kernel to enhance the ability to perform power managemnet

1) Alarms
   - Implemented in the Linux kernel and is visible to the app developer through the AlarmManager in the RunTime core libraries
   - Is implemented in the kernel so that an alarm can trigger even if the system is in sleep mode
2) Wakelocks
   - Prevents an Android system from entering into sleep mode
   - These locks are requested through the API whenever an application requires one of the managed peripherals to remain powered on

Thank you !

Operating Systems are among the most complex pieces of software ever developed !