OS Services, Interfaces, Components

CS201 Lecture 2

Jason Hibbeler

University of Vermont

Fall 2019

Basic Role of an OS

Question: why do we have computers?

- and why do we create software for computers?

Basic Role of an OS

Answer: to improve our lives.

Figure 1: A 700 Series Roomba

OS Services

Operating-system services make life more convenient for the application developer

Application program need to interact with the resources that the OS controls

- so it’s only natural that the OS provide efficient and easy-to-use interfaces to these resources

Categories of OS Services

User interface

- the most basic level: how does a user actually communicate with the OS (and thus with the system)?
- command-line interface (CLI), batch, GUI

Program execution

- user needs to be able to load a job into memory and run it

Questions (iClickers out!)

(1) True (A) or false (B)

A big computer can run only one program at a time.

(2) True (A) or false (B):

A laptop computer can run only one program at a time.

(3) True (A) or false (B):

An iPhone can run only one program at a time.

(4) True (A) or false (B):

More than one person at a time can run programs on a computer.

Categories of OS Services

I/O operations

- read and write files (structured data)
- communicate with the keyboard and the screen
- communicate with the network (if available) or Bluetooth
- communicate through USB

File-system management

- the system needs to organize files and directories, including creation and deletion
- the system must provide search capabilities
- and provide security and protection

Categories of OS Services

Communication

- process-to-process communication
- might be through messages or through shared memory

Error detection

- OS needs to detect and correct errors
- memory error, I/O error, user program error
- in the worst case, the OS might halt the system: Blue Screen of Death or kernel panic

OS Services for the System Itself

Resource allocation

- CPU scheduling

Accounting

- who is using what, and how or for how long

Protection and security

- controlling access to the system and its resources

Communicating with the OS

Command-line interpreter

```
jhibbele@zoo ~> who
jmrankin pts/0        2019-01-07 20:58 (65-183-138-199-dhcp.burlingtontelecom.net)
jhibbele pts/1        2019-01-08 16:39 (squall.cems.uvm.edu)
mmsander pts/4        2019-01-08 14:49 (ip040185.uvm.edu)
jtl      pts/5        2019-01-07 09:29 (otter.uvm.edu)
bhimberg pts/6        2019-01-07 09:47 (bras-vprn-shbkpq4086w-lp130-03-174-92-225-100.dsl.bell.ca)
jhenry   pts/8        2019-01-08 09:24 (184-091-196-139.res.spectrum.com)
pjp      pts/10       2019-01-08 13:23 (ip0af51042.int.uvm.edu)
webmster pts/11       2019-01-08 09:31 (ip0af51d6f.int.uvm.edu)
cdanfort pts/15       2019-01-08 10:13 (chaos.uvm.edu)
trustees pts/16       2019-01-08 15:17 (ip0af51042.int.uvm.edu)
87lhowel pts/17       2019-01-08 10:56 (ip074074.uvm.edu)
jhibbele@zoo ~>
```

Communicating with the OS

Command-line interpreter

```
jhibbele@zoo ~> df -k
Filesystem            1K-blocks       Used  Available Use% Mounted on
/dev/mapper/vg0-lv_root
                       13044868   10402532    1985588  84% /
tmpfs                   2097152          0    2097152   0% /dev/shm
/dev/sda1                487652      84722     377330  19% /boot
tmpfs                   4194304        184    4194120   1% /tmp
/dev/mapper/vg0-varlv
                       19544764    1447180   17101656   8% /var
/dev/mapper/vg0-lv_cadence
                       26175772   20757416    4082108  84% /usr/local/cadence
fs01.uvm.edu:/fs01/users
                       20511744    9111552   10351616  47% /fs/users
fsb0.uvm.edu:/fsb0   4227438592 3298604032  714139136  83% /fs/b0
fsa9.uvm.edu:/fsa9   4227434496 3345123328  667563520  84% /fs/a9
fs01.uvm.edu:/fs01     20511744    9111552   10351616  47% /fs/01
etc.
```

Communicating with the OS

Command-line interpreter

```
jhibbele@zoo ~> uptime
 16:42:08 up 1 day, 10:12, 11 users,  load average: 0.04, 0.06, 0.02
jhibbele@zoo ~> whoami
jhibbele
jhibbele@zoo ~> getconf PAGESIZE
4096
jhibbele@zoo ~>
```

Communicating with the OS

Command-line interpreter

- the actual program that reads input and acts on it is called a *shell*

in Unix

- there are various shells: bash, ksh, csh, etc.
- Unix shell is very lightweight: there are a few basic built-ins, but every other "command" essentially tells the shell "go find the file with this name and execute it"
- what's the advantage of this?

```
squall|/users/j/h/jhibbele
> type rm
rm is hashed (/bin/rm)
squall|/users/j/h/jhibbele
> ls -ls /bin/rm
56 -rwxr-xr-x 1 root root 53592 Feb  6  2017 /bin/rm
squall|/users/j/h/jhibbele
>
```

Communicating with the OS

MS-DOS

- the MS-DOS shell actually processes the command (instead of handing off control to another program that processes the command)

GUIs for Operating Systems

Windows, macOS

- whether you use a GUI or a CLI is really just determined by what you want to do, and how you want to do it
- system administrators would probably use the CLI
- with a CLI, we can build “programs” (shell scripts), to do repetitive tasks
- GUI is good for non-power-users
- there is one important exception though, where the GUI is a fundamental part of the OS and the only way to interact with the system (what is this exception?)

System Calls

As we said, the OS mediates and controls all access to system resources

Simple example: we write a user program to read the contents of one file and write those contents to a different file

Sequence of Operations

1. prompt the user for the filename ⇒ system calls to read keyboard input
   - or put up a GUI with filenames and have the user pick one ⇒ many system calls
2. check to see whether the file exists ⇒ system call
3. open the input file and create the (empty) output file ⇒ many system calls
   - if any errors arise, handle them ⇒ many system calls
4. read data from the input file and write it to the output file ⇒ many system calls
   - if any errors arise, handle them ⇒ many system calls
5. when we're done (end-of-file condition ⇒ system call), alert the user ⇒ system calls

A "system call" is code that the operating system is running

- in the case above, it's running all of this code on your behalf—because your program is accessing various system resources

System Calls: through an API

API (application programming interface)

- provides clear, simplified way for an application developer to interact with the OS and with system resources
- provides portability: every Linux system provides the same system calls
- provides abstraction: the application developer doesn't need to know the details of how the low-level stuff is implemented

What are examples of system calls that you've already used?

System Calls: through an API

API (application programming interface)

- provides clear, simplified way for an application developer to interact with the OS and with system resources
- provides portability: every Linux system provides the same system calls
- provides abstraction: the application developer doesn't need to know the details of how the low-level stuff is implemented

What are examples of system calls that you've already used?

- malloc(), printf()

Linux Example: read()

man page

```
NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd
       into the buffer starting at buf.

       If count is zero, read() returns zero and has no other results.
       If count is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE
       On  success, the number of bytes read is returned (zero indicates
       end of file), and the file position is advanced by  this  number.

       [etc.]
```

Linux Example: malloc()

```
NAME
      calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS
      #include <stdlib.h>

      void *calloc(size_t nmemb, size_t size);
      void *malloc(size_t size);
      void free(void *ptr);
      void *realloc(void *ptr, size_t size);

DESCRIPTION
      calloc() allocates memory for an array of nmemb elements of size
      bytes each and returns a pointer to the allocated memory.  The
      memory is set to  zero.  If nmemb or size is 0, then calloc()
      returns either NULL, or a unique pointer value that can later be
      successfully passed to free().

      [etc.]
```

System Calls: the Runtime Library

Application makes a system call

- OS intercepts the call
- in kernel mode, OS performs the low-level function
- OS returns control back to the application

Parameters for the call

- might go into a register
- or into a block of memory that the underlying OS call can read
- and the result code from the call can go into a register that the application code can read

Types of System Calls

1. process control
2. file management
3. device management
4. information maintenance
5. communications
6. protection

Process Control

Need to be able to start a new process

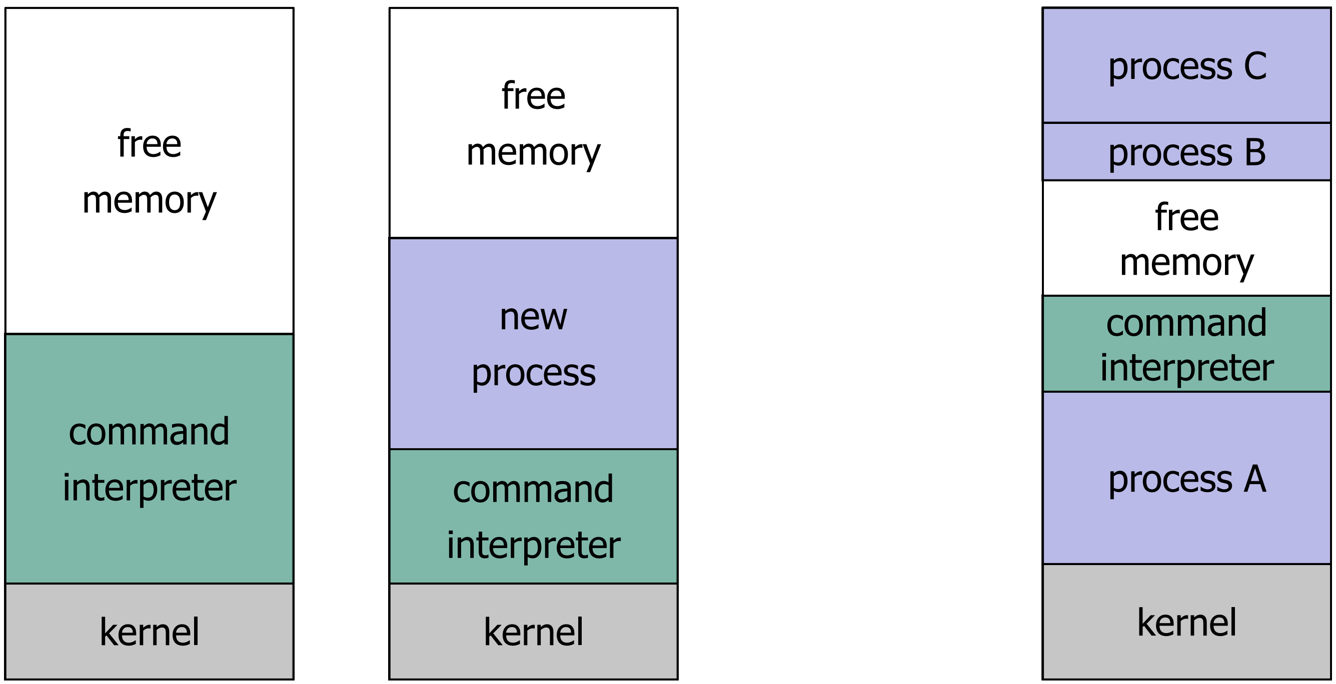
- OS needs to load the code for the program into memory
- OS then transfers control to this new program
- when the new program ends—either normally or abnormally—then the OS needs to clean up after the program

Also, need to set attributes on a new process

- and explicitly wait for it to finish or wait for some specific event to occur

Starting a New Job

single-tasking system vs. multi-tasking system



single-tasking OS (e.g. MS-DOS)

multi-tasking OS (e.g Unix)

File Management

Create and delete files and directories

Read and write to files

Set attributes of files and directories

Device Management

OS must grant exclusive access for us to a device

- why is this?
- why must the access be exclusive?

Then we can read from the device and write to the device (depending on what the device is)

Information Maintenance

OS must mediate for us when we want to read information about the state of the system

- date and time
- amount of memory in system, amount of memory in use
- dump memory dump (useful for debugging the OS itself)
- single-step mode during execution: after each CPU instruction, generate a trap that the user can catch (think about a debugger)
- profiling information
- process table

Communication

Message passing: a process needs to be able to exchange data with a different process

- possibly on a different physical machine
- each machine has a unique hostname and an IP address

Communication

A different communication scheme is through the use of shared memory

- the OS allocates a block of memory that is visible to more than one process
- this way, two or more processes can exchange data by reading and writing to/from the shared memory
- this requires the OS to provide an additional mechanism—what is it?

Communication

A different communication scheme is through the use of shared memory

- the OS allocates a block of memory that is visible to more than one process
- this way, two or more processes can exchange data by reading and writing to/from the shared memory
- this requires the OS to provide an additional mechanism—what is it?

$\Rightarrow$ OS must let more than one process access a particular region of memory

Protection

OS manages who can access what

- users need some limited control over this, which the OS must enable

System Services

Between the OS and the user applications are system programs

- file management and modification
- compilers and debuggers
- communication
- background services (daemons): listen for network requests, run jobs at scheduled times, clean up the system, etc.

OS Specificity

Some kinds of programs are built and can run on only a specific OS

- because of the use of system calls, implemented as part of the OS

Some programs execute in a virtual machine

- such as the JVM

OS Specificity

In general, each OS has specific constraints that describe the structure of executable programs

- and different processor architectures require different instructions in compiled programs

Linux programs are generally hardware independent

- so long as you compile it for the target CPU, it should run

OS Design and Implementation

What are the requirements for an OS?

- does the answer depend on whom we are asking?

OS Design and Implementation

What are the requirements for an OS?

- does the answer depend on whom we are asking?

What do users want from the OS?

What do the people who design and implement the OS want?

OS Design and Implementation

What are the requirements for an OS?

- does the answer depend on whom we are asking?

What do users want from the OS?

- reliability, efficiency, speed, etc.

What do the people who design and implement the OS want?

- ease of maintenance and support, correctness, reliability

Mechanisms vs. Policies

Mechanisms: how the OS performs a task

Policies: what tasks the OS should perform

Requirements vs. implementation

Why should we separate these?

- flexibility
- for much more discussion of requirements vs. implementation, take CS205

this is an advertisement

Mechanisms vs. Policies

Mechanisms: how the OS performs a task

- this is part of how the OS is constructed
- for example, how it keeps me from accessing other people's files

Policies: what tasks the OS should perform

- this involves decisions that the system administrator makes
- for example, a policy that says "a user should by default not be able to access other users' files"

Implementation

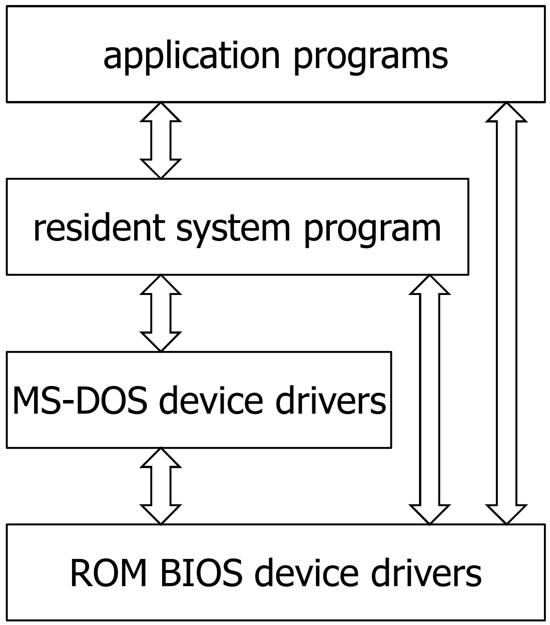
MS-DOS and Unix are implemented mostly in C

It can make sense to implement some parts of the kernel in assembly language (why?)

OS Structure

MS-DOS

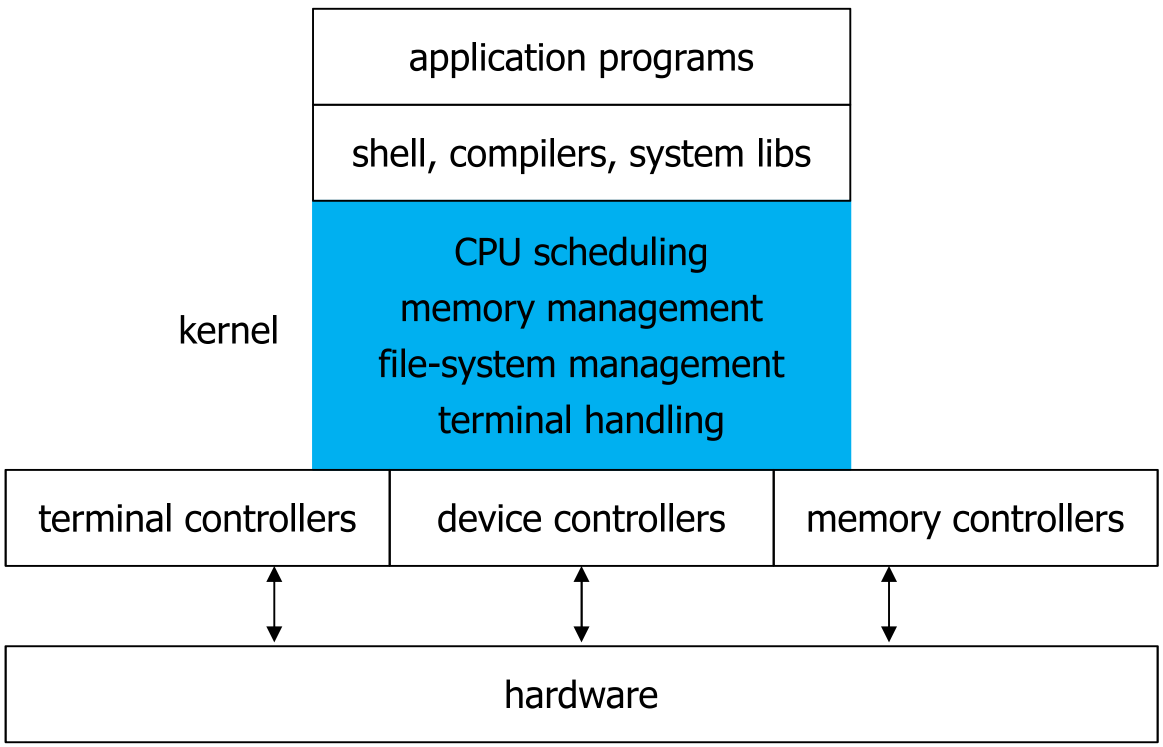
- OS structure was dictated by the hardware; original Intel processors did not have dual mode
- this means that application programs have to be able to access the hardware directly
- this means that the whole system can fail due to incorrect behavior by one process
  - ⇒ not a good situation



OS Structure

Original Unix structure

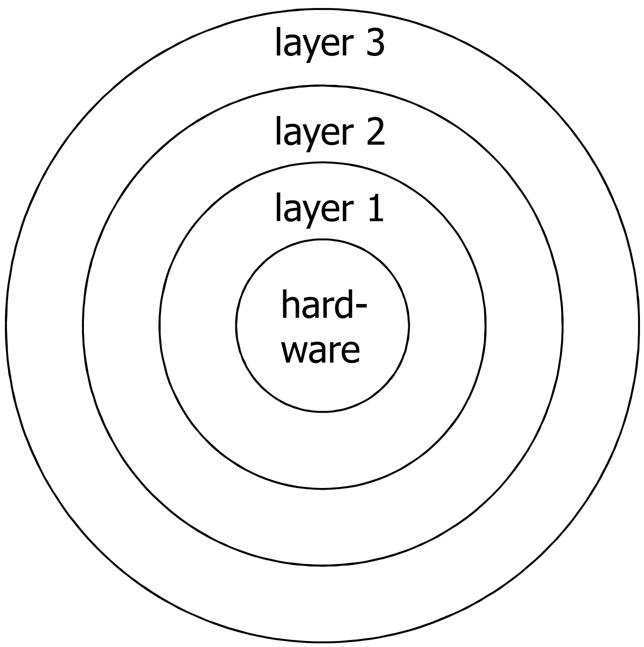
- this looks better, but the problem is the huge, monolithic kernel
- difficult to maintain and enhance



Layered Architecture

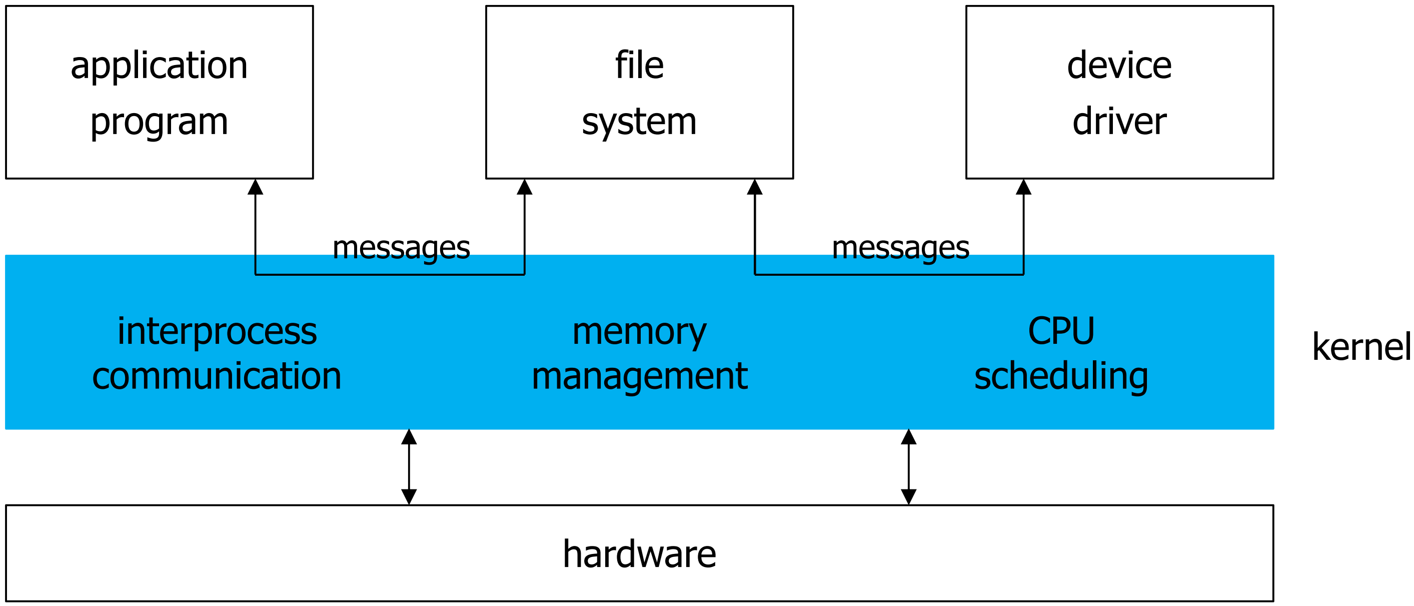
Build the system in layers

- each layer defines an interface: the layer above me can call me, and I call the layer below me



Microkernels

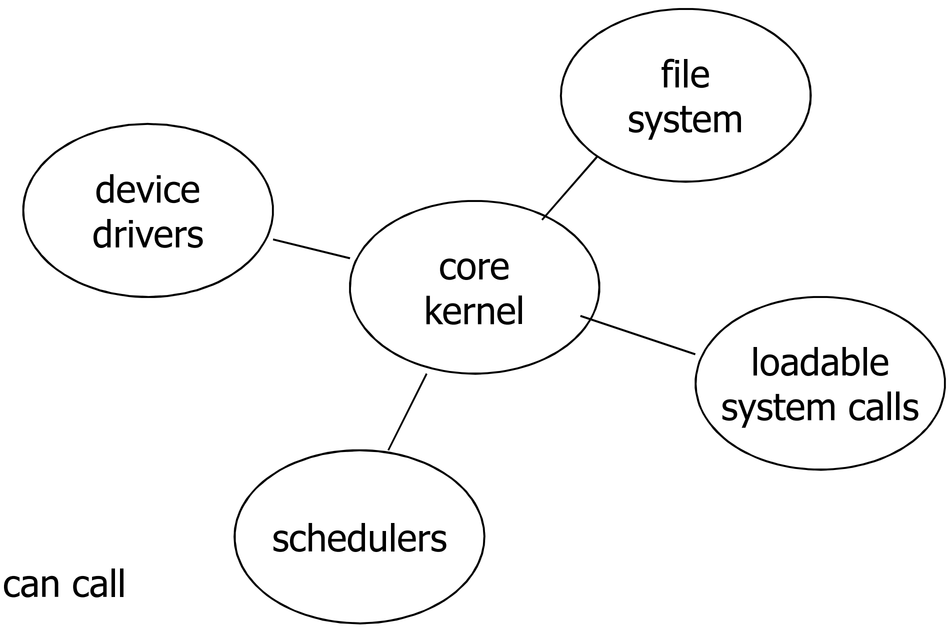
Basic idea: put as little as possible into the kernel



- this looks OK
- the problem can be one of performance: all of the message passing and context switching that's necessary for an application to perform system operations

Modules

Basic idea: put as little as possible into the kernel; put other function in modules that can be loaded as needed



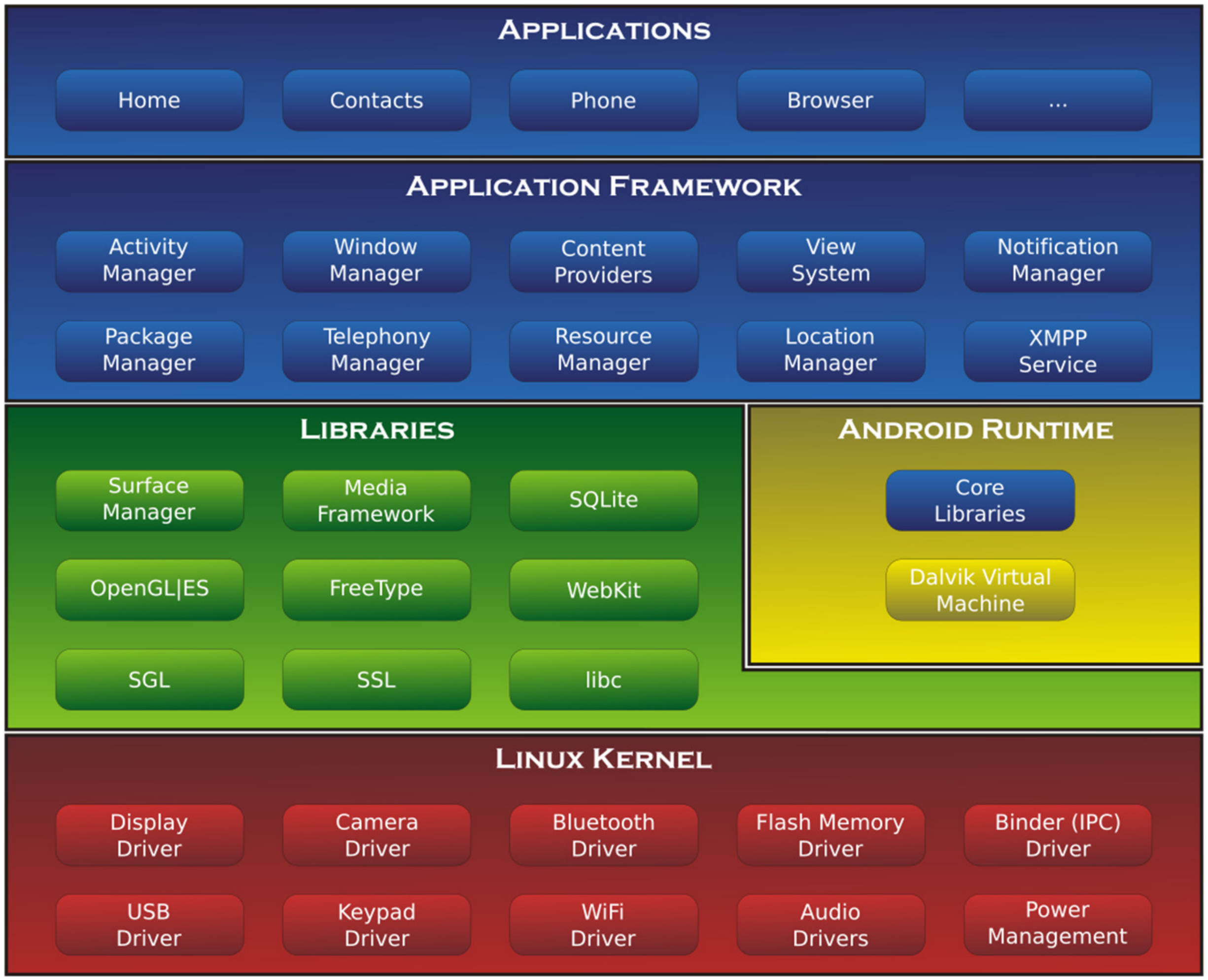
- more efficient: any module can call any other module
- modularity makes the OS easier to maintain and enhance
- this is the structure of Linux

Example Operating Systems

macOS, iOS, Android

- all three have a Unix kernel
- iOS and Android have special components for handling touch-screen interfaces
- iOS is mostly not open source
- Android **is** open source + some proprietary stuff
- iOS and Android have media services built in to the OS
- iOS and Android have additional features pertaining to power management

Android OS



https://commons.wikimedia.org/wiki/File:Android-System-Architecture.svg

Debugging

Debugging user applications is “easy”

Debugging kernel problems is hard

- Blue Screen of Death saves the contents of memory for later examination

OS Installation

For efficiency and correctness, the OS should be tailored to the machine it's running on (i.e., tuned for the machine)

- how much memory is available?
- what devices are present?
- what kind of CPU scheduling should be used?

OS Startup

System boot

- when the system is reset, the CPU starts executing instructions from a specific location in ROM
- this loads a small piece of the kernel (the bootstrap program)
- and the bootstrap program then checks the state of the machine and loads the rest of the OS

ROM (read-only memory)

- slow
- expensive