(Latest Revision: Dec 25, 2020)
[2020/12/25: typo correction]
[2020/02/09: small changes in notes on 2.8.4 and 2.9.2]

Chapter Two -- Operating-System Structures -- Lecture Notes

- **2.0 Chapter 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

- **2.1 Operating-System Services**

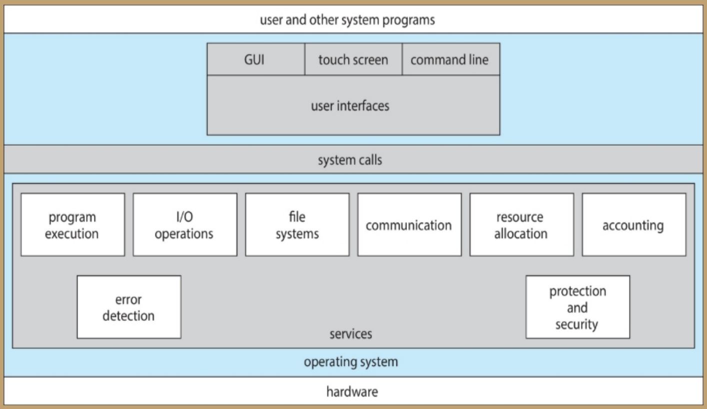


Figure 2.1: A view of operating system services

Operating systems provide many services. Several are listed here.

The authors say that these OS services are mainly concerned with helping users:

- **User Interface:** There are different kinds, like touchscreen, GUI, and command-line.
- **Program Execution:** (Execute programs for users)
- **I/O operations:** It is much too difficult for users to operate the I/O hardware correctly without help.

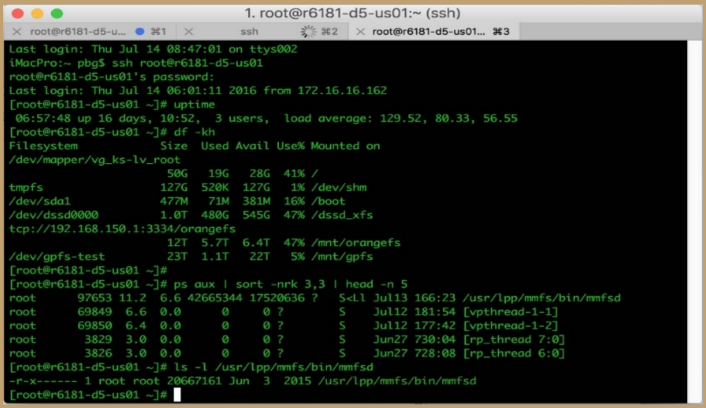
- **File System Manipulation:** The OS helps us store, organize, manage, and protect our information.
- **Communications:** Users need their processes to exchange information. OSs help. The two main ways to do it are *with shared memory* and *by message passing*.
- **Error Detection:** An OS continually checks to see if something is going wrong. The OS is programmed to take appropriate action.

The authors say that the following OS services are more for ensuring efficient operation of the system itself:

- **Resource Allocation:** For example, allocation of CPU time, main memory, and online file space
- **Logging:** An OS keeps a lot of records for many purposes, such as accounting, fault detection, protection, and security.
- **Protection and Security:** It is necessary to prevent user processes from harming each other, or harming the OS.

2.2 User and Operating-System Interface

Operating Systems often include user-interface system programs. The text lists three main types. These interfaces are NOT normally considered a direct function of the operating system. Therefore, our text does not discuss user and operating-system interfaces very much.



```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0      0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0      0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829  3.0  0.0      0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826  3.0  0.0      0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

Figure 2.2: The bash shell command interpreter in macOS

- **2.2.1 Command Interpreters:** The text-based interfaces on unix-type systems are the most common examples nowadays. The user types commands with a keyboard. The commands are displayed in an on-screen window or terminal. Results produced by the commands are also displayed the same way. Commands can be *internal* or *external*.
- **2.2.2 Graphical User Interface:** These are screen, mouse, and keyboard interfaces. There are multiple windows and menus arranged according to a

desktop metaphor. The KDE and GNOME environments are open software that runs under Linux.

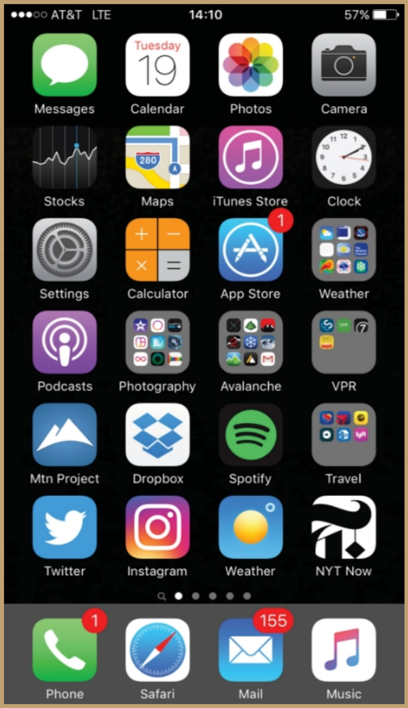


Figure 2.3: The iPhone touch screen

- **2.2.3 Touch-Screen Interface:** These interfaces are common on mobile devices. Users tap and swipe the screen to make selections. Some interaction is audio-based. Users can type on a keyboard simulated on-screen.
- **2.2.4 Choice of Interface:** Command Interpreters are powerful for doing many challenging tasks, but require rather a long time to learn. GUIs and Touch-Screen Interfaces make it easy to learn how to perform commonly-needed tasks.

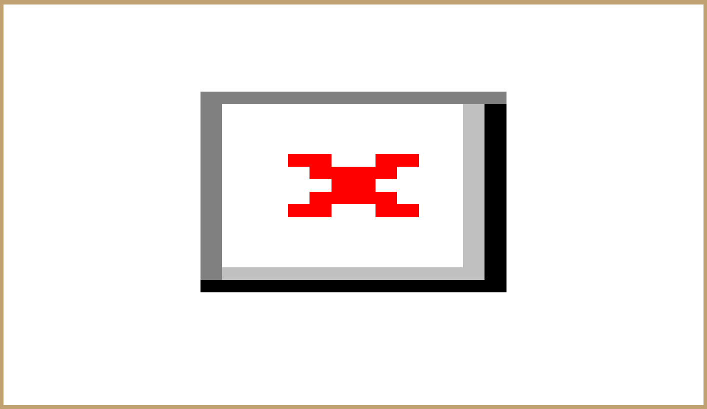
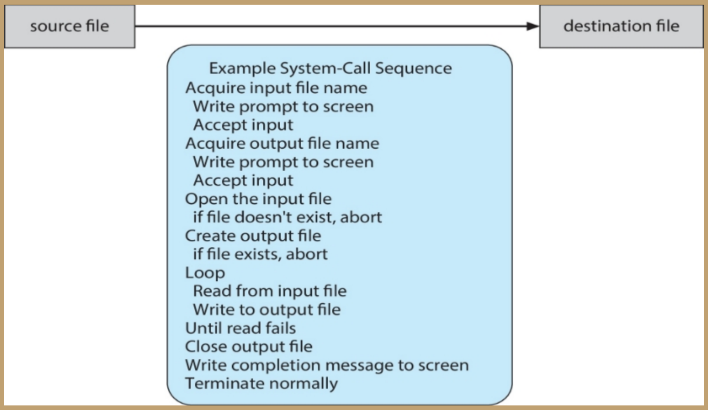


Figure 2.4: The macOS GUI

- **2.3 System Calls**

  A system call is like a function call. It is a request for an OS service. ***System calls are very important.*** "System calls" is very much the answer to the question, "What does the operating system do?"



```
Example System-Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Figure 2.5: Examples of how system calls are used

  - **2.3.1 Example:** As an example, the text describes how a simple program that makes a copy of a file has to make a large number of system calls, involving keyboard, screen, and disk I/O.

  - **2.3.2 Application Programming Interface:** Even low-level *system programmers* usually do NOT put system calls in their program code. Instead they use an application programming interface (API) which provides a library of functions. When a process calls an API function, the API function, in turn, makes whatever system calls are necessary. One of the benefits of using APIs is that the programs are easier to port from one kind of computer to a different kind. Also, system calls often have difficult-to-understand syntax, and use of the API functions is simpler and easier. The three most common APIs are the Windows API, the POSIX API, and the Java API.

    The run-time support functions in libraries provide the system-call interface for many programming languages - including C and Java.

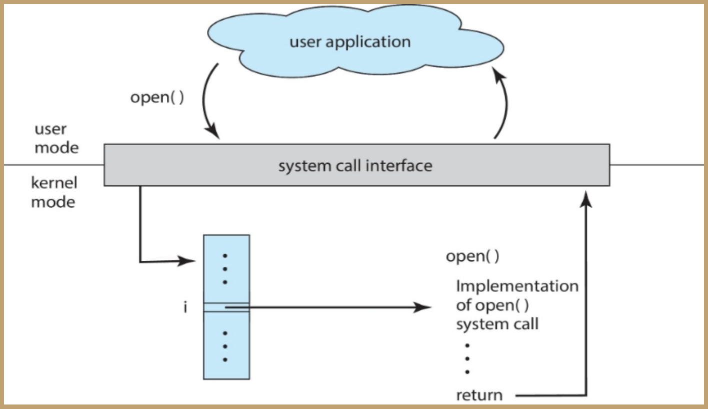


Figure 2.6: The handling of a user application invoking the open() system call

Typically, there is a number associated with each system call. To implement a system call, the requesting process executes an instruction that causes a trap. Of course, the trap puts an OS handler into the CPU. The OS handler looks up the system call number in a table maintained by the OS. The table entry tells the OS which piece of code it has to execute to perform the system call, and the OS then executes that code. (See figure 2.6.)

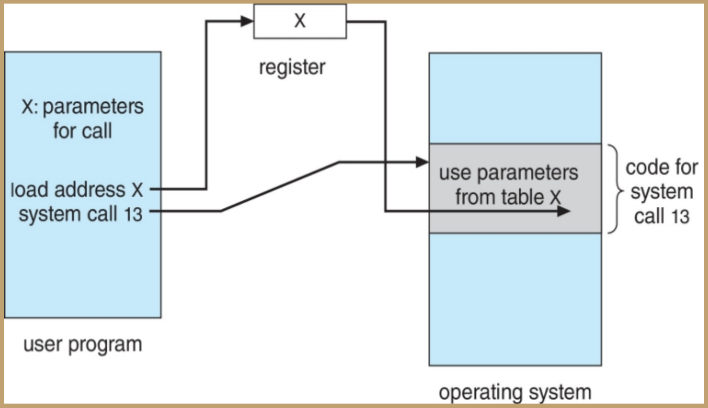


Figure 2.7: Passing of parameters as a table

important detail: **A process can pass parameters to a system call by leaving them in CPU registers, by leaving them in a RAM block (pointed to by an address in a register), or by pushing them onto the runtime stack.**

- **2.3.3 Types of System Calls**

Much of what we study in the rest of the text will be about how to implement the system calls listed here.

- **2.3.3.1 Process Control:** Examples are create process, terminate process, load process, execute process, get process attributes, set process attributes, wait for event, signal event, allocate memory, and free memory.

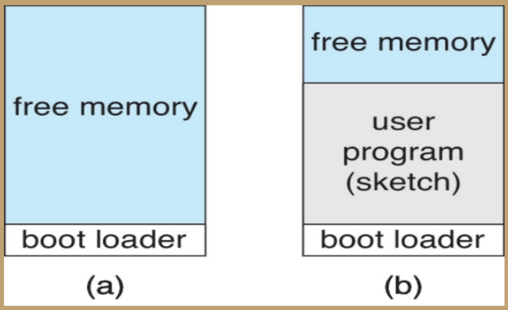


Figure 2.9: Arduino execution (a) At system startup (b) Running a sketch

A parent process can create a child process and control its actions. We will see a lot of discussion of that later on.

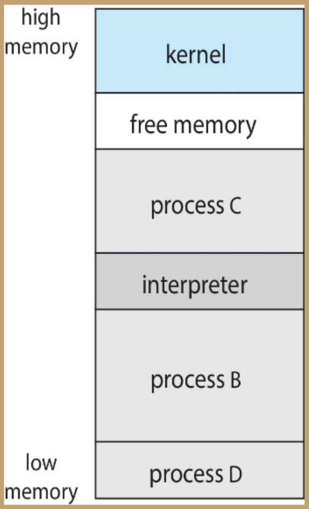


Figure 2.10: FreeBSD running multiple programs

- **2.3.3.2 File Management:** Examples of file management system calls are create file, delete file, open file, read file, write file, reposition file, get file attributes, and set file attributes. Typically we need the same or similar system calls for directories (folders).

        - **2.3.3.3 Device Management:** Examples of device management system calls are request (exclusive access to) device, release device, read device, write device, and reposition device. Processes use devices in a manner that is similar to how processes use files.

        - **2.3.3.4 Information Maintenance:** Examples are get current time and date, get version number, get amount of free memory, get amount of free disk space, dump memory, and get time profile. Mentioned here, as well as under Process Control system calls, are get process attributes and set process attributes.

        - **2.3.3.5 Communication:**

          Examples of system calls for the message-passing model of process communication might be get host ID, get process ID, open connection, close connection, accept connection, wait for connection, read message, write message, and close connection.

          Examples of system calls for the shared-memory model of process communication are shared memory attach, read memory, and write memory.

        - **2.3.3.6 Protection:** System calls such as get permission and set permission inspect and modify the permission settings on resources such as files and disks. The permission settings are intended to control which processes or other entities are allowed access to the resources. Specific users may be authorized or denied access by system calls with names like allow_user and deny_user.

- **2.4 System Services**
    - Most computer scientists think about an OS mainly as a kernel, which is the part of the computing system 'from the system call level down to the hardware.'
    - So the basic idea is that user programs make requests for OS services by using system calls, and the real 'guts' of the OS is the software that runs to perform the system call.
    - There is *system service* software, which computer scientists often see as a layer that is technically outside the kernel, yet somehow closer to the kernel than software that we normally think of as 'user apps.' Examples:
        - some software for manipulating and modifying files and directories (including searching and editing software);
        - system performance monitoring software;
        - programming support like compilers, assemblers, and debuggers;
        - services for loading parts of programs into memory and making linkages among them;
        - software that provides the means for processes to communicate, possibly via a network;

        - long-running specialized server processes, like a subsystem (daemon) that accepts jobs for a printer, or daemons that provide access to remote file systems.
    - Most users feel as if "the OS" is high-level system services, accessed with touch screens, GUIs, or command-line interfaces. However, to computer scientists "the OS" usually means "the kernel."

- **2.5 Linkers and Loaders**

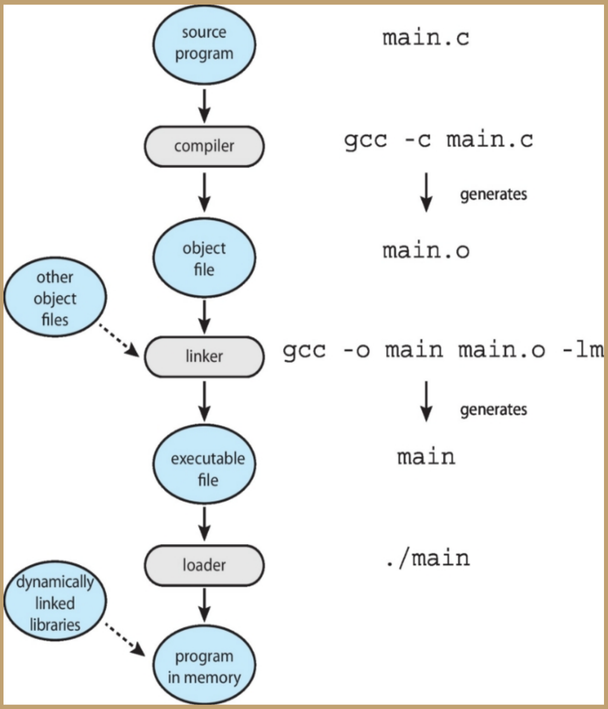


Figure 2.11: The role of the linker and loader

    - It is the job of an OS to load programs into the memory of the computer and execute them.
    - The job is complex.
        - One reason is that most programs are really made of different pieces of code written at different times and stored in different files.
        - Another reason is that it is often necessary to move pieces of running programs from one place in memory to a different place.
        - Also, a particular part of a program may not be needed every time the program executes. So the OS may delay loading some parts until they are needed.

- **2.6 Why Applications Are Operating-System Specific**
    - Clearly, if we want to run a program on a computer, we have to write the instructions in the machine language understood by the computer.

- Also, programs need to make system calls, and different operating systems have different APIs and/or different sets of system calls.
- Besides, executable files have different formats on different computers, and different operating systems utilize different data structures and algorithms for similar tasks.
- For those and other reasons, there is no way to write one program that will run on dissimilar computing systems.

- **2.7 Operating-System Design and Implementation**
  - **2.7.1 Design Goals:** Generally speaking, users want a system that is convenient, reliable, safe, and fast. Developers want to satisfy users, but want the system to be easy to build, implement, and maintain.
  - **2.7.2 Mechanisms and Policies:** *Policy* refers to *what* we decide to do. *Mechanism* refers to *how* we do what we do. Policies change over time, so the computing system tends to be more flexible if we build in mechanisms that can support a range of policies. The text uses the example of a CPU scheduler that can give high priority to programs that use the CPU intensively, but the same scheduler can be 'tweaked' by changing a few numbers in some tables, and then that scheduler will give priority to the opposite kind of program, those that use the CPU sparingly.

  - **2.7.3 Implementation:**
    - The *implementation* of the operating system, that is the manner in which the ideas of the design are written in programming language(s).
    - Typically a few parts of an OS have to be written in assembly language.
    - However **modern operating systems are written almost entirely in a high-level programming language such as C.** Actually several different high level languages may be used to implement various parts of an operating system. Advantages: code is developed faster; is smaller, easier to understand, debug and change; and easier to port.
    - There are some things that just can't be done with high-level language statements, for example some device-driver code and instructions that save and restore register values.
    - There was a time when writing more of the code in assembly could be justified because assembly programmers could use 'tricks' that made routines run faster by some constant factor compared with compiled high level code. Modern optimizing compilers have eliminated that advantage almost entirely, although there are still a few bottlenecks in systems that are 'hand-coded' in assembly language in order to achieve 'the absolute maximum' efficiency - e.g. the authors mention interrupt handlers, memory managers, and CPU schedulers in this regard.
    - As the authors say: "As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code."

- **2.8 Operating-System Structure**
    - Principles of good software development that one learns in a data structures course apply to operating systems, like modularity and information hiding.

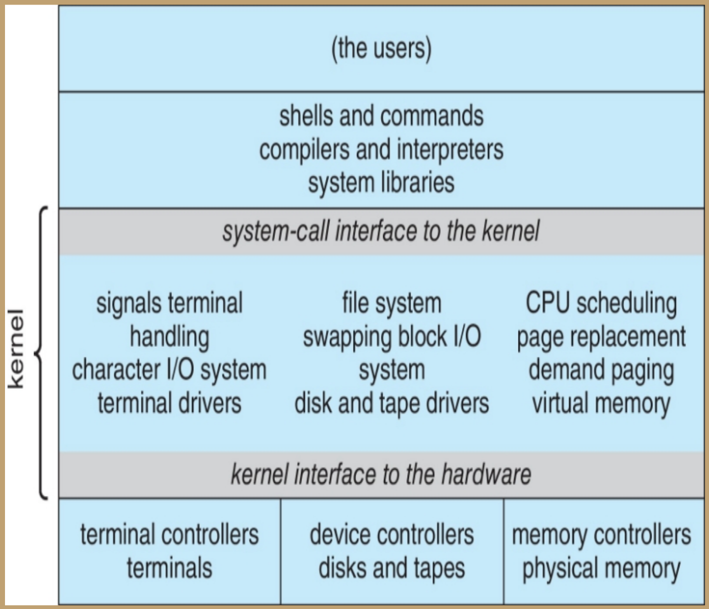


Figure 2.12: Traditional unix system structure

- **2.8.1 Monolithic Structure:** Because users of operating systems place great value on speed and efficiency, many OS designs feature a *monolithic kernel,* which is "a single, static, binary file that runs in a single address space." A monolithic kernel can have some modular design aspects, but structural limitations make them difficult to implement and modify.

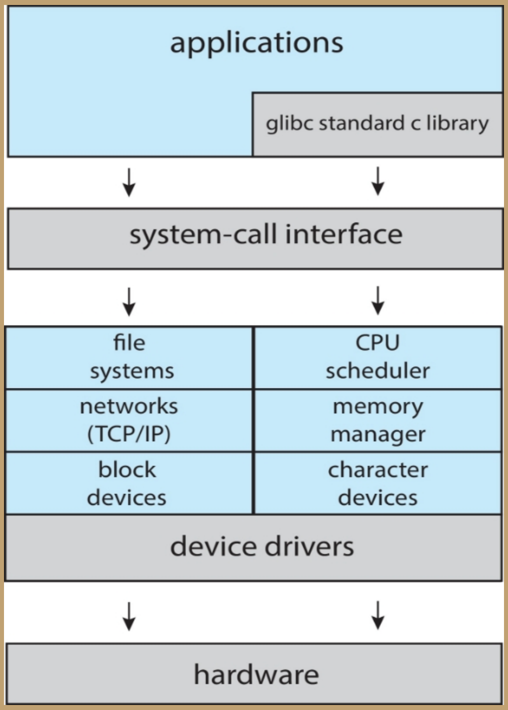


Figure 2.13: Linux system structure

- **2.8.2 Layered Approach:**

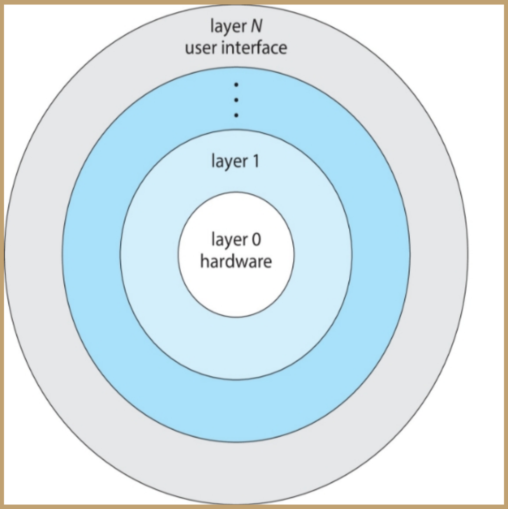


Figure 2.14: A layered operating system

- In a layered approach, we make implementation and modification easier, by designing the operating system as a set of very distinct modules called layers.

    - Each layer is a well-defined abstract data structure (aka, object) with specific data members and operations (methods).
    - The design of the layers requires that they be numbered from lowest to highest, with the lowest being the hardware, and the highest being the user interface.
    - Code in one layer, can make calls to functions in other layers, but only if those functions are *in the same or lower layers*.
    - Because of the rules, once the system is designed, it's relatively easy to build without errors, starting with the hardware, and continuing with the next highest layer only after the current layer has been implemented and tested.
    - It can be quite difficult to design operating system layers in a way that absolutely assures that a lower layer will never need to use a function in a higher layer.
    - Also, a command that originates in a higher layer can result in a cascade of calls to functions in lower and lower levels. If this travels through several layers, it can cause delay, which tends to make the system slow and inefficient.
    - Many OS designs use the idea of layering, but, to protect the efficiency of the system, there tend to be just a few large layers, rather than many small layers.

  - **2.8.3 Microkernels**

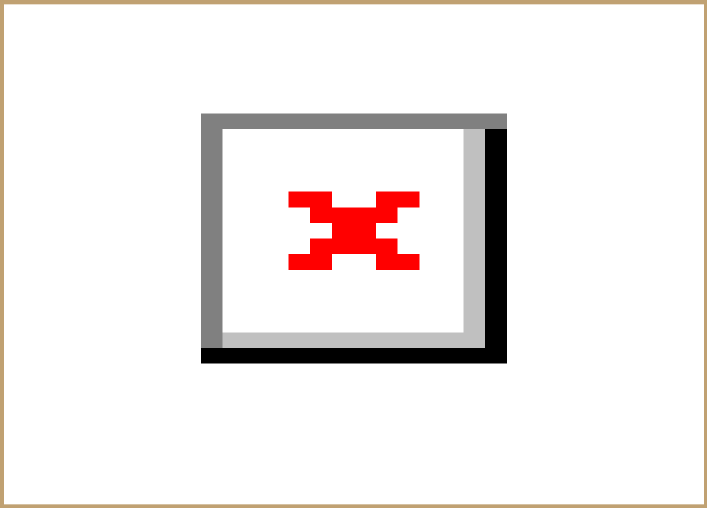


Figure 2.15: Architecture of a typical microkernel

    - Over years, as designers added more and more functionality to operating systems, it became increasingly difficult to implement monolithic systems.
    - Researchers at Carnegie-Mellon University pioneered *microkernel design*. The monolithic kernel is replaced with a microkernel that provides just a

    few essential services, and the rest of the operating system is turned into separate user-level programs.
    - It is possible for things like file systems and device drivers to be user-level programs.
    - Microkernels typically contain some core process and memory management functions, as well as a message passing service.
    - When a user program needs a service from the OS, it typically uses the microkernel's message-passing service. The microkernel makes sure that the request reaches the necessary part of the OS.
    - Microkernel architectures are very modular. They are usually relatively reliable and easy to secure, maintain, modify, and port.
    - Microkernel architectures can suffer performance problems due to excessive time spent on message-passing work.
    - Because of such performance problems, designers tend to just use some of the ideas of microkernel approach, without building 'pure' microkernel systems.

- **2.8.4 Modules**
    - "Modules" is a design style that is currently quite popular.
    - It has similarities to other models. For example it is similar to the microkernel model, except that it employs dynamically loadable kernel modules (LKMs) that are *not* constrained to communicate with other modules through the core part of the kernel. There is also a similarity with the layered model, which has very clear and separated modules.
    - The LKMs may be loaded at boot time, or loaded when needed while the system is running. They can also be deleted from memory if no longer needed.
    - An example would be a device driver support module loaded when a new device is plugged into the computer, and when the device is unplugged, the module is deleted because it is not needed any more.

- **2.8.5 Hybrid System** Most real operating systems are not pure examples of any of the operating system design models discussed above. Real systems are usually a mix of different design models.

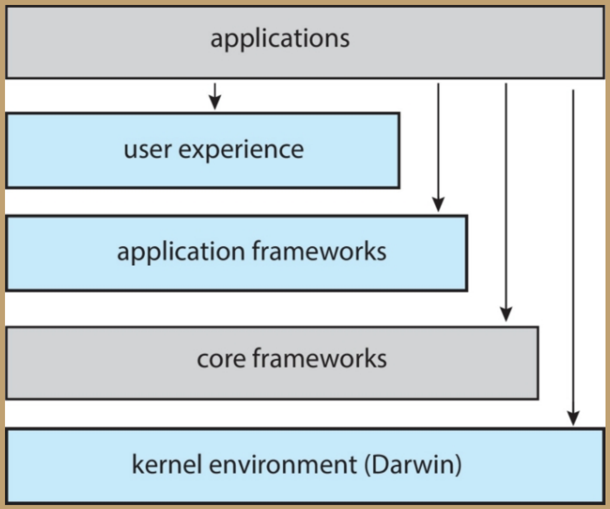


Figure 2.16: Architecture of Apple's macOS and iOS operating systems

- **2.8.5.1 MacOS and iOS** MacOS and iOS systems have a kernel environment known as Darwin. Darwin is an example of a hybrid system - a mix of different designs. It has a Mach micro-kernel and a BSD Unix kernel implemented over the Mach micro-kernel. Programs can make Mach system calls, and they can also make BSD system calls.

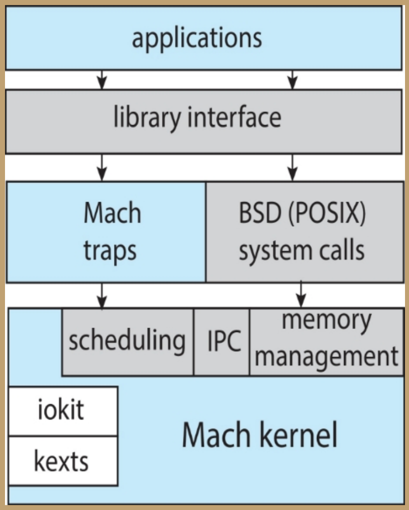


Figure 2.17: The structure of Darwin

- **2.8.5.2 Android** Android systems have a modified Linux kernel.

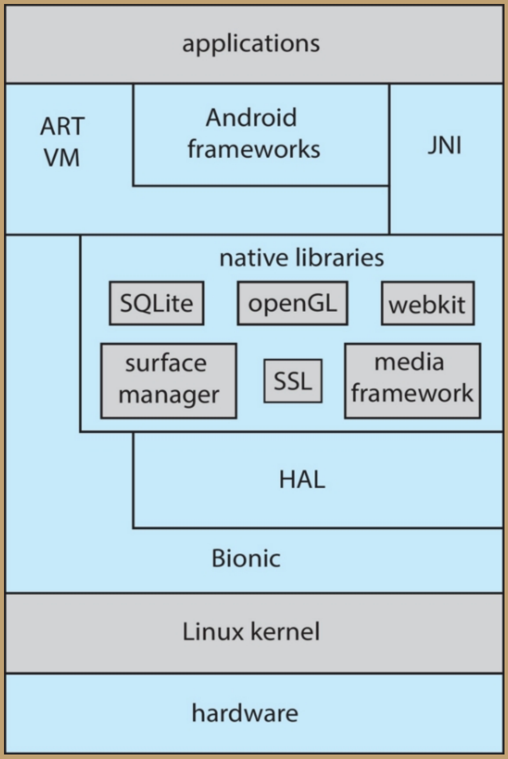


Figure 2.18: Architecture of Google's Android

- **Windows Subsystem for Linux** Windows has some subsystems that emulate other operating systems. For example, there is a subsystem for Linux that allows users to run native Linux applications on a Windows machine. When an application running in the Linux environment makes a system call, it is (basically) translated into Windows system calls.

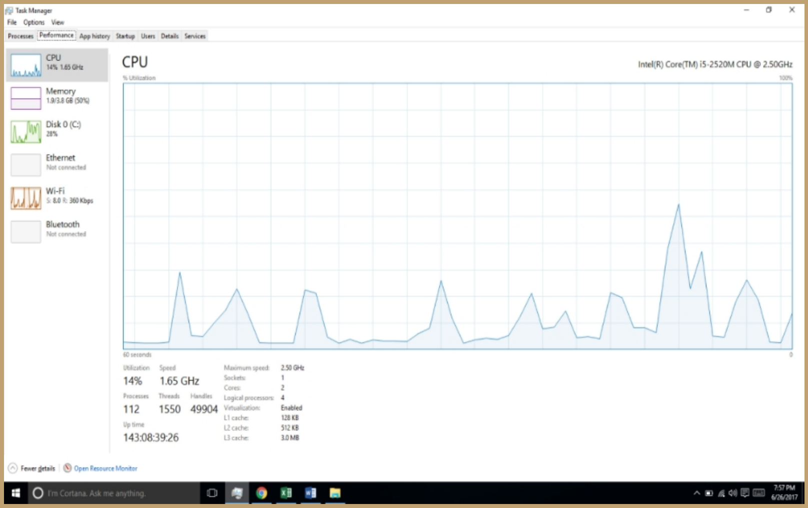
- **2.9 Building and Booting an Operating System**
  - **2.9.1 Operating-System Generation**
    - Systems have to be configured for the specific site to which they are targeted. We perform system generation when we put a new host online, providing at that time, for example, a hostname, host IP address, router IP address.
    - Other SYSGEN operations possible: disk partitioning, installation of device drivers, choice of CPU-scheduling algorithm, and setting of the maximum number of processes to be supported.
    - The SYSGEN information may be incorporated into the source code for the OS, and a custom version of the OS then compiled.
    - Another option, common in PCs, laptops, and mobile devices, is for the system to be pre-compiled and the SYSGEN information saved in the form of tabular information in files, accessed and utilized by the pre-compiled OS in order to achieve the desired customization effects. For

        example the system can read its hostname and default router from a configuration file. Scripts written during the SYSGEN can be executed each time the computer boots in order to perform custom setups.
        - Tables can determine which of certain dynamically loadable modules are to be loaded at boot time.
        - SYSGEN may be followed by a linking, but not compiling step, so that certain modules such as drivers are incorporated into a permanent executable image of the kernel.

    - **2.9.2 System Boot**
        - Modern computers have ROM that begins executing automatically at system start-up time. The program may be called BIOS (Basic Input Output System) or UEFI (Unified Extensible Firmware Interface).
        - The ROM boot program will typically run diagnostics, initialize the hardware, and load and execute another program from secondary memory, or from a network. That program could be the OS, but it could also be another program that in turn loads the OS, or there might be more steps in the series.
        - In some small devices, the entire OS may be in ROM or EPROM.

- **2.10 Operating-System Debugging**
    - **2.10.1 Failure Analysis**
        - Operating systems keep logs of "interesting events." That includes errors made by processes.
        - The OS may save a copy of a failed process to a location on secondary memory. This is called a *core dump*. (At one time the technology that implemented main memory was called "magnetic core.") In other words the OS may save all the *process context* at the time the error is detected. The *process context* is all the information about the process that is in the main memory.
        - When the kernel is the process that fails, we often say that the OS *crashed*, and the dump may be called a *crash dump*.
        - Users, programmers, and computer administrators can use a software package called a debugger to help figure out what the errors are in a process. It's possible to examine a running process or a core dump with a debugger.

    - **2.10.2 Performance Monitoring and Tuning** There is some discussion here of tools used in an OS to monitor the performance of the system. People can use the information to get ideas for improving performance.



CPU 14% 1.65 GHz
Memory 1.9/3.8 GB (50%)
Disk 0 (C:) 28%
Ethernet Not connected
Wi-Fi S: 8.0 R: 360 Kbps
Bluetooth Not connected

CPU Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz

```
Utilization   Speed                 Maximum speed:        2.50 GHz
14%           1.65 GHz              Sockets:              1
                                    Cores:                2
Processes     Threads   Handles     Logical processors:   4
112           1550      49904       Virtualization:       Enabled
                                    L1 cache:             128 KB
Up time                             L2 cache:             512 KB
143:08:39:26                        L3 cache:             3.0 MB
```

Figure 2.19: The Windows 10 task manager

- **2.10.2.1 Counters**
  - Some of the monitoring involves just keeping counts of the number of times certain operations are executed.
  - Linux tools such as ps, top, vmstat, netstat, and iostat use information from counters.
  - Many flavors of unix use a special in-memory file system (/proc) for storing statistics about each current process.
  - Windows has a tool called the Windows Task Manager that gives information about current applications and processes, CPU and memory utilization, and networking statistics.

- **2.10.3 Tracing**
  One can use tracing tools to gather data about important "system events."

  For example `strace` traces the system calls a process makes, `gdb` functions as a source-level debugger that traces things the program does, like reading or changing a variable, `perf` is a collection of Linux performance tools, `tcpdump` collects network packets.

- **2.10.4 BCC**

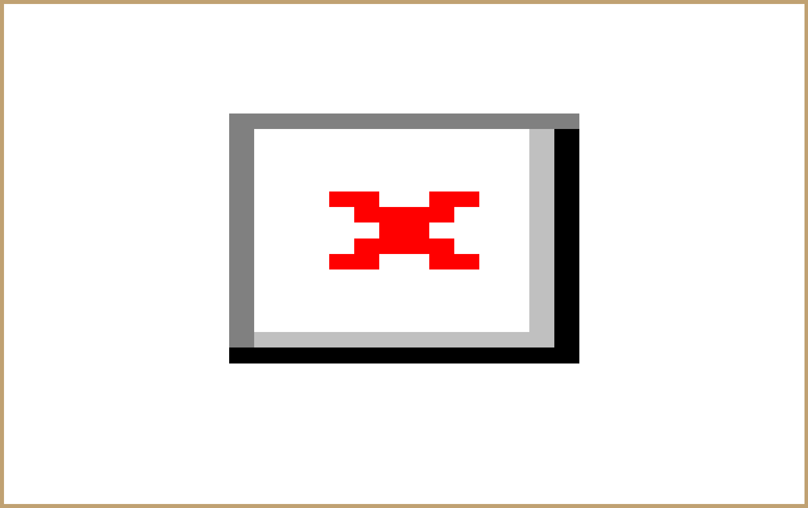


Figure 2.20: The BCC and eBPF tracing tools

BCC is a tracing and debugging toolkit used on Linux systems, useful for examining interactions between user-level and kernel code. Through its use, special instructions are dynamically inserted into a running Linux system, where they are able to sense and record specific events, such as the invocation of a specific system call, or the time currently required to perform each disk I/O.

There is a very wide variety of kinds of monitoring that one can do with BCC. The authors say that it is possible to safely trace virtually any area of a Linux system, which is very useful for finding performance bottlenecks and security exploits.