2.1 Operating-System Services

* An operating system provides an environment for the execution of programs.
* It provides certain services to programs and users to make the programming task easier.



One set of operating system services provides functions that are helpful to the user

• **User interface:** take several forms. One is a **command-line interface (CLI)**, which uses text commands . Another is a **batch interface**, in which commands and directives and most commonly, a **graphical user interface (GUI).**

• **Program execution:** The program must be able to end its execution, either normally or abnormally (indicating error).

• **I/O operations: for** efficiency and protection, users usually cannot control I/O devices directly.

Therefore, the OS must provide a means to do I/O.

• **File-system manipulation**. Some OS include permissions management to allow or deny access to files or directories based on file ownership.

• **Communications**. There are many circumstances in which one process needs to exchange information with another process.

Communications may be implemented via **shared memory**, or **message passing.**

• **Error detection**. Errors may occur in the:

1. CPU and memory hardware (as a memory error or a power failure).
2. in I/O devices. (Such as a parity error on disk).
3. 3-and in the user program (arithmetic overflow).

* For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing (halt the system-terminate an error-causing ...ETC).

**Another set of operating system functions for ensuring the efficient operation of the system itself.**

• **Resource allocation**. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.

**Operating system manages:**

1-recourses have special allocation (CPU cycles, main memory, and file storage).

2- Recourses have much more general request and release code (I/O devices).

* operating systems have CPU-scheduling routines to manage these factors

• **Accounting**. Record how much and what kinds of computer resources are used. accumulating usage Statistics.

• **Protection and security**. Protection involves ensuring that all access to system resources is controlled.

Security requires authentication .It extends to defending external I/O devices, from invalid access attempts and to recording all such connections for detection of break-ins.

2.2 User and Operating-System Interface

Several ways for users to interface with the operating system:

1-command-line interface, or **command interpreter**, allows users to directly enter commands to be performed by the OS.

2- Allows users to interface with the operating system via a **graphical user interface**, or GUI.

**2.2.1 Command Interpreters**

* Some operating systems include the command interpreter in the kernel.
* Others, treat the command interpreter as a special program that is running when a job is initiated .EX: Windows and UNIX.
* On systems with multiple command interpreters, the interpreters are known as shells. EX: UNIX and LINUX different shells (Bourne shell – Corn shell…etc).
* Most shells provide similar functionality.
* The main function of the command interpreter is to get and execute the next user-specified command.

**These commands can be implemented in two general ways:**

|  |  |
| --- | --- |
| 1. The command interpreter itself contains the code to execute the command. | 1-The command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. used by UNIX |
| Example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. | The UNIX command to delete a file **rm file.txt** would search for a file called rm, load the file into memory, and execute it with the parameter file.txt. |
| In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code. | In this way, programmers can add new commands to the system easily. |

**2.2.2 Graphical User Interfaces**

Users employ a mouse-based window and- menu system characterized by a **desktop** metaphor, **icons**, **folder**—or pull down a menu that contains commands.

History:

* The first GUI appeared on the Xerox Alto computer in 1973.
* The advent of Apple Macintosh computers in the 1980s.
* the Aqua interface that appeared with Mac OS X.
* Microsoft’s first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS OS.
* Cosmetic changes in the appearance of the GUI
* Smartphones and handheld tablet computers typically use a touchscreen interface with **gesture**s.
* Traditionally, UNIX systems have been dominated by command-line interfaces.
* Significant development in GUI designs from various open-source projects, such as K Desktop Environment (or KDE) and the GNOME desktop by the GNU project which are used by LINUX and UNIX.

**2.2.3 Choice of Interface**

\*the choice is personal preference.

* **System administrators** and **power users** use the **command-line interface**:

\*\*more efficient, giving them faster access. \*\*make repetitive tasks easier, in part because they have their own programmability- \*\*the program is not compiled into executable code but rather is interpreted by the command-line interface**: shell scripts.**

* Most Windows users use the Windows GUI environment and almost never use the MS-DOS shell interface.
* Mac OS X (which is in part implemented using a UNIX kernel), the operating system now provides both an Aqua interface and a command-line interface.

2.3 System Calls

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

* Available as routines written in C and C++.
* Simple programs may make heavy use of the operating system. Look p62
* Application developers design programs according to an **application programming interface (API)**.

**Three of the most common APIs:** the Windows API, the POSIX API for POSIX-based systems, and the Java API.

A programmer accesses an API via a library of code provided by the operating system (In C language, the library is called **libc**).

Each operating system has its own name for each system call.

The functions that make up an API typically invoke the actual system calls .

EX: Function CreateProcess () (create a new process) actually invokes the NTCreateProcess () system call in the Windows kernel.

**Why would an application programmer prefer programming according to an API rather than invoking actual system calls?**

One benefit concerns program **portability**.

Actual system calls are more detailed and difficult to work with than the API.

The run-time support system (a set of functions built into libraries included with a compiler) provides a **system-call** **interface** that serves as the link to system calls made available by the OS.

* Three general methods are used to pass parameters to the operating system:

1-pass the parameters in registers.

2-Parameters are more than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register. This approach taken by Linux and Solaris.

3-Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the OS.

* Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

2.4 Types of System Calls

System calls can be grouped roughly into six major categories: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, **communications**, and **protection**.

**2.4.1 Process Control**

* A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()).
* Terminating abnormally:

A dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid (help) the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem.

A process or job executing one program may want to load () and execute () another program.

**THIS** section is really important I couldn’t make a summary for it. Please look it in textbook.

**2.4.2 File Management**

We first need to be able to: create () and delete () open () read (), write (), or reposition () (rewind or skip to the end of the file, for example). Close () the file.

* For either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them.
* File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, get file attributes () and set file attributes (), are required for this function.
* Some operating systems provide move () and copy ().

**2.4.3 Device Management**

The various resources (needed to execute a process) controlled by the operating system can be thought of as devices (physical devices (disk drives), and abstract or virtual devices (files)).

* A system with multiple first **request** () a device, to ensure exclusive use of it then release it.These functions is similar to the **open** () and **close** () system calls for files. We can **read**(),**write** (), and (possibly) **reposition**() the device, just as we can with files.
* The similarity led to merge the two into a combined file-device structure with system calls used on both.

**2 .4.4 Information Maintenance**

* Many system calls exist for transferring information between the user program and the operating system. EX: system call to return the current **time** () and **date** ().
* Another set of system calls is helpful in debugging a program.

EX: system calls to **dump** () memory.

* A program **trace** lists each system call as it is executed.

Microprocessors provide a **single step**, in which a **trap** is executed by the CPU after every instruction. The trap is usually **caught** by a debugger.

* Many operating systems provide a time profile (time that the program executes at a particular location) of a program.
* At every **occurrence of the timer interrupt**, the value of the program counter is **recorded**.
* Calls are also used to **reset** the process information (**get-process-attributes** () and **set-process**\_**attributes** ()).

**2.4.5 Communication**

Two common models of interprocess communication: the message passing model and the shared-memory model.

* In the **message-passing model,** the communicating processes exchange messages with one another to transfer information **directly** or **indirectly** through a common **mailbox**.

Each computer in a network has a **host name** which must hasnetwork **identifier (IP**).

Each process has a **process name**, and this name is translated into an identifier.

* The **get\_hostid** () and **get\_processid** () system calls do this translation.
* Depending on the system’s model of communication, identifiers are then passed to the **open**() and **close**() calls or **open\_connection**() and **close\_connection**().
* The recipient process usually must give its permission with an **accept\_connection** () call.
* **Receiving connections processes** are special-purpose **daemons:**
* They execute a **wait\_for\_connection**() call and are awakened when a connection is made.
* The ***source of the communication***, known as the **client**, and the ***receiving daemon***, known as a **server**, then exchange messages by using **read\_message**() and **write\_message**() system calls. The **close connection** () call terminates the communication.
* In the **shared-memory model**, processes use shared memory **create** () and shared memory **attach** () system calls to create and access to regions of memory owned by other processes.

2.4.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system.

System calls providing protection include **set\_permission** () and **get\_Permission** (), which **manipulate** the permission settings of resources such as files and disks.

The **allow\_user** () and **deny\_user** () system calls: **access** to certain resources.

2.5 System Programs

**System programs**, also known as **system utilities**, provide a convenient **environment** for program development and execution.

**They can be divided into these categories:**

• **File management.**

• **Status information.** Some ask the system for the date, time, or similar status information.

* Others are more complex, providing detailed performance, logging, and debugging information.
* These programs format and print the output to the output devices.
* Some systems also support a registry, which is used to store and retrieve configuration information.

**• File modification.** text editors to create and modify the content of files stored on disk or other storage devices.

**• Programming-language support.** Compilers, assemblers, debuggers, and

Interpreters provided with the operating system or available as a separate download.

• **Program loading and execution.** The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders and debugging.

**• Communications**. The mechanism for creating virtual connections among processes, users, and computer systems.

• **Background services**. Methods for launching certain system-program processes at boot time.

* Running system-program processes are known as **services**, **subsystems**, or **daemons.**
* EX: process schedulers, system error monitoring services, and print servers.
* Operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities**.**

2.6 Operating-System Design and Implementation

Problems we face in designing and implementing an operating system.

**2.6.1 Design Goals**

* The first problem in designing a system is to define goals and specifications.
* At the highest level, it will be affected by the choice of hardware and the type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose.
* The requirements then will be harder to specify but generally divided into two basic groups: user goals and system goals.
* User and Programmers want the system: easy, reliable and fast,,, etc. But this is not useful(vague) in the system design .
* no unique solution to the problem of defining the requirements for an operating system.

**2.6.2 Mechanisms and Policies**

**Mechanisms** determine how to do something; **policies** determine what will be done.

* Policies **change** across places or over time.
* In the worst case, each **change** in policy would require a change in the underlying mechanism.
* If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy
* Microkernel-based operating systems (Section 2.7.3) take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building **blocks**. These blocks are almost policy **free**, allowing more advanced mechanisms and policies to be added via user-created kernel modules or user programs themselves.
* Making the scheduling mechanism general purpose **allows** vast policy changes to be made with a single load-new-table command.
* At the other extreme is a system such as Windows, in **which both mechanism and policy are encoded** in the system to enforce a global look and feel.
* **Policy** decisions are important for all **resource allocation**.

**2.6.3 Implementation**

Once an operating system is designed, it must be implemented.

It is difficult to make general statements about how they are implemented.

2. 7 Operating-System Structure

A common approach is to partition the task into small (well-defined) components, or **modules**, rather than have one **monolithic** system.

Many operating systems do not have well-defined structures. They started as small, simple, and limited systems and then grew.

EX: M5-DOS so it was not carefully divided into modules.

**2.7.2 Layered Approach**

* With proper hardware support, operating systems can be broken into smaller pieces.
* The OS can then retain much greater control over the computer and application.
* A system can be made modular in many ways.
* One method is the **layered approach**, in which the operating system is broken into a number of layers (levels):
* The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

The main advantage of the layered approach:

* Simplicity of construction and debugging.

Each layer:

* Selected so that it uses functions (operations) and services of only lower-level layers.
* Implemented only with operations provided by lower-level layers.
* Needs to know only what these operations do not how they are implemented.
* Hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach: is **defining** the various layers. Because a layer can use only lower-level layers, careful planning is necessary.

* A final problem with layered implementations is that they tend to be less efficient than other types.

**2.7.3 Microkernels**



Researchers at Carnegie Mellon University developed an OS called Mach that modularized the kernel using the **microkemel** approach.

**Microkernels method:** structures the OS by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.

Micro kernels provide minimal process and memory management, and communication facility.

**Main function of the microkemel** is to provide communication through message passing between the client program and the various services running in user space.

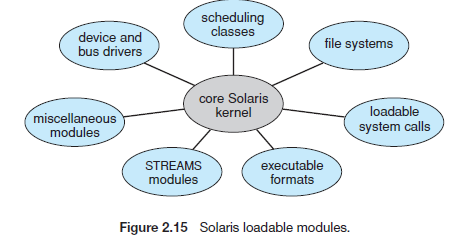
**Benefit of the microkernel:**

* Makes extending the operating system easier (new services are added to user space and consequently do not require modification of the kernel.)
* The microkernel also provides more security and reliability, since most services are running as user- rather than kernel- processes.
* Ex: The Mac OS X kernel (also known as **Darwin**) is partly based on the Mach microkernel.

Unfortunately, the performance of microkernels can **suffer** due to increased system-function overhead. EX: Windows NT which corrected later (Windows 95. Windows NT 4.0).

By the time Windows XP was designed, Windows architecture had become more **monolithic** than microkernel.

**2.7.4 Modules**



The best current methodology for operating-system design involves using **loadable kernel modules**.

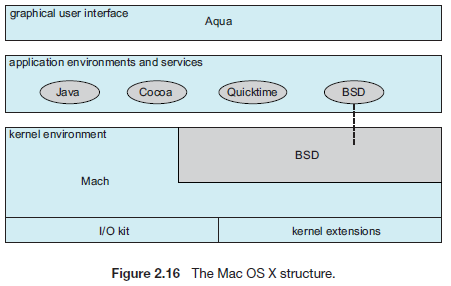
**Loadable kernel modules**: the kernel has a set of core components and links in additional services via modules, either at boot time or during run time.

Common in implementation of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows.

**2.7.5 Hybrid Systems**

* Few operating systems adopt a single, strictly defined structure.
* Instead, they combine different structures, resulting in **hybrid** systems that address performance, security, and usability issues.
* **Both Linux and Solaris are:**
* Monolithic, because having the operating system in a single address space provides very efficient performance.
* Modular, so that new functionality can be dynamically added to the kernel.
* Windows is largely
* monolithic,
* Some behavior of microkernel systems.
* Support for separate subsystems (known as operating-system **personalities**) that run as user-mode processes.
* Support for loadable kernel modules.

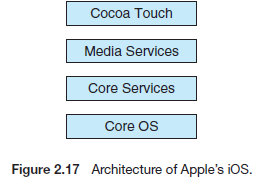
**2.7.5.1 Mac OS X**



The Apple Mac OS X operating system uses a hybrid structure, the **Cocoa** environment for the Objective-C programming language, used for writing Mac OS X applications.

* The top layers include the **Aqua user interface** (Figure 2.4) and a set of application environments and services.
* Below these layers is the **kernel environment**, which consists primarily of the **Mach microkernel** and the BSD UNIX kernel.
* **Mach** provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling.
* **The BSD** component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.
* Kernel environment provides an I/O kit for development of device drivers and loadable modules (which Mac OS X refers to as **kernel extensions**).

**2.7.5.2 iOS**



* iOS is a mobile operating system run on Apple smartphones, the IPad, the iPhone.
* iOS is structured on the Mac OS X operating system, with added functionality pertinent (related) to mobile devices, but does not directly run Mac OS X applications.
* Close-sourced.

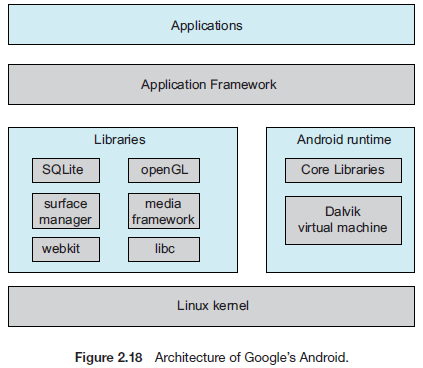
**1-Cocoa Touch** is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices.

**Difference between Cocoa, and Cocoa Touch** is that Cocoa Touch provides support for hardware features unique to mobile devices, such as touch screens.

**2-Media services** layer provides services for graphics, audio, and video.

3-The **core services** layer provides a variety of features, including support for cloud computing and databases. 4-The **bottom layer** represents the core operating system, which is based on the kernel environment.

**2.7.5.3 Android**



* Designed by the Open Handset Alliance (led primarily by Google).
* Android runs on a variety of mobile platforms and is open-sourced.
* Similar to iOS, **layered stack** of software that provides a rich set of framework.
* At the bottom is the **Linux kernel**, used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management.
* The Android **runtime environment** includes a core set of libraries as well as the Dalvik virtual machine.
* Applications in the Java language. The Java class files are first compiled to Java bytecode and then translated into an executable file that runs on the **Dalvik virtual machine**.
* The **set of libraries** includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia.
* The **libc library** is similar to the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

**2.8 Operating-System Debugging**

* **Debugging** is the activity of finding and fixing errors in a system, both in hardware and in software.
* **Performance Problems** are considered **bugs**.
* Debugging can also include performance tuning, which seeks to improve performance by removing processing **bottlenecks**.

**2.8.1 Failure Analysis**

|  |  |
| --- | --- |
| If a process fails,   * Most operating systems write the error information to a **log file**. | Failure in the kernel is called a **crash.** |
| The OS can also take a **core (memory) dump**—a capture of the memory of the process— and store it in a file for later analysis. | When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**. |
| Debugging user-level process code is a challenge. | * Operating-system kernel debugging is even more complex than debugging user-level process because of:   **The size and complexity of the kernel**, **its control of the hardware**, and **the lack of user-level debugging tools**. |
| such strategies would be unnecessary for debugging ordinary user-level processes. | \*A common technique is to **save the kernel’s memory state to a section of disk**.  \*If the kernel detects an **unrecoverable** error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area.  \* When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis. |

**2.8.2 Performance Tuning**

**Performance tuning** seeks to improve performance by removing processing **bottlenecks**.

* **In a number of systems**, the OS does tuning by producing ***trace listings*** of system behavior.

1. All interesting events are logged with their time and important parameters and are written to a file.
2. An analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies.
3. These same traces can be **run as input** for a simulation of a suggested improved system.
4. Traces also can help people to **find errors** in operating-system behavior.

* **Another approach to performance tuning** uses **single-purpose, interactive tools** that allow users and administrators to question the state of various system components to look for bottlenecks.
* EX:

1- One such tool employs the **UNIX** command top to display the resources used on the system, as well as a sorted list of the “top” resource-using processes.

1. Other tools display the state of disk I/O, memory allocation, and network traffic.
2. The Windows Task Manager is a similar tool for Windows systems. The task manager includes information for current applications as well as processes, CPU and memory usage, and networking statistics.

**2.8.3 DTrace**

**DTrace** is **a facility that dynamically adds probes to a running system**, both in user processes and in the kernel.

* These probes can be queried via the D programming language to **determine** an astonishing amount about the **kernel**, the **system state,** and **process activities.**
* Example and more important info at page 87-88.

2.9 Operating-System Generation

* OS are designed specifically for **one machine at one site.**
* More commonly, operating systems are designed to run **on any of a class of machines** at **a variety of sites.**
* **The system that configured or generated for each specific computer site, a process for this sometimes known a system generation SYSGEN**.
* The operating system is normally **distributed** on **disk**, on **CD-ROM or DVD-ROM**, or as an “ISO” image, **which is a file in the format of a CD-ROM or DVD-ROM**.

**To generate a system**,

* The **SYSGEN program** reads from a given file, or asks the operator of the system for information:

EX: What CPU is to be used? How will the boot disk be formatted? What devices are available?

* Once this information is determined, it can be used in several ways.

2.10 System Boot

After an operating system is generated, it’s available for use by the hardware.

* **The procedure of starting a computer by loading the kernel is known as booting the system.**
* A small piece of code known as the **bootstrap program** or **bootstrap loader locates** the kernel, **loads** it into main memory, and **starts** its execution.
* Computer systems, such as PCs, use a **two-step process** in which a simple bootstrap loader **fetches** a more complex boot program from disk, which in turn **loads** the kernel.
* The **bootstrap** program can perform a variety of tasks. Usually, one task is to **run diagnostics** to determine the state of the machine.
* If the **diagnostics** pass, the program can continue with the **booting steps**. It can also **initialize** all aspects of the system, Sooner or later, it **starts** the operating system.

Some systems—such as cellular phones, tablets, and game consoles—**store the entire operating system in ROM.**

Storing the operating system in **ROM** is suitable for **small** operating systems, **simple** supporting hardware, and **rugged** operation.

* **A problem with this approach** is that changing the bootstrap code requires changing the ROM hardware chips.
* Some systems **resolve** this problem by using **erasable programmable read-only memory (EPROM)**, which is read only except when explicitly given a command to become writable.

**All forms of ROM** are also known as **firmware**, since their characteristics fall somewhere **between** those of hardware and those of software.

* **A problem with firmware**: is that executing code there is **slower** than executing code in RAM.
* **Solution**: Some systems store the operating system in firmware and copy it to RAM for fast execution.
* **A final issue with firmware** is that it is relatively **expensive**, so usually only **small** amounts are available.
* For large operating systems the **bootstrap loader** is stored in **firmware**, and the operating system is on disk. In this case,

The bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that boot block.

* **GRUB** is an example of an open-source bootstrap program for Linux systems.
* All of the **disk-bound bootstrap**, and the OS itself, can be easily changed by writing new versions to disk.

A disk that has a **boot partition** is called a **boot disk** or **system disk**.

* Now that the full bootstrap program has been **loaded**, it can **traverse** the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be **running**.