This chapter outlines different views of operating systems from users and systems, presents a high-level description of the OS interfaces that programmers or users interact with including **system calls** and **system programs**, and describes various approaches used in designing operating systems.

**Operating System Services**

- An operating system provides an environment for program execution by offering various services to programmers and users.
- There are three primary approaches for interacting with an OS: (1) command interpreters (CLI), (2) graphical user interfaces, and (3) touch-screen interfaces.
- Operating systems offer ***two general categories*** of services and functions. One category of services is to enforce protection among different processes running *concurrently* in the system. For instance, processes are allowed to access only those memory that are associated with their own address space; a process is not allowed to access hardware devices directly without operating system assistance and intervention. The second category of services is to provide functionalities that are not supported directly by the underlying hardware, for instance memory management and file systems.
- Operating system provides a wide range of services, such as user interface (UI), program execution, I/O and file system management, communications, logging, and error detection. This can also be in the forms of resource allocation, accounting, security and protection.
- At the low level or programming level, OS provides **system calls**, which, through *Application Programming Interfaces* or **APIs**, allow running programs to request services from operating systems. For example, user can use **`printf()`**, an API, which invokes the system call **`write()`** to perform the actual operations – write to files or I/O devices.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- At the high level, OS provides **system programs** for users to issue requests without the need for writing programs; for instance, creating and deleting a file. Most users' view of an operation system is "defined" by system programs.
- The types of user requests vary in different levels. The system-call level requests provide basic functions such as process control, file and device manipulation. Higher-level requests, satisfied by command interpreter or system programs, can possibly be translated into a sequence of system calls.

**System Calls**

- **System calls** are generally available as functions in C and C++, although certain low-level tasks (e.g., access hardware) written in assembly language instructions.
- The **system call interface** of a programming language library serves as a "link" to the system calls. This interface intercept function calls in APIs and invokes the necessary system calls within an operating system. For example, the standard C

library or libc provides the system-call interface for UNIX and Linux systems.

- There is a reason for separating API and the underlying system call, (1) program portability by using API, and (2) to hide the complex details in system calls.
- There are **three** general methods used to pass parameters to the OS when using system calls. The simplest approach is to pass parameters in ***registers***. If there are more parameters than the number of registers, the parameters are generally stored in a ***block***, or ***table of memory***, and the address of the block is passed as a parameter in a register. Parameters can also be placed, or pushed, onto a ***stack*** by programs and popped out of the stack by the OS.
- The run-time environment (**RTE**) contains the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software such as libraries and loaders. The system-call interface is part of RTE.

**Linker and Loader**

- A **linker** combines relocatable object modules (compiler generated) into a single binary executable file stored on a secondary storage device. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specified with the flag -lm).
- A **loader** loads the executable file into memory, possibly supporting *dynamically linked libraries* (**DLLs**), where it becomes eligible to run on an available CPU. The benefit of DLLs is that it avoids linking and loading libraries that may end up not being used into an executable file. Moreover, this makes it possible for multiple processes to share DLLs, resulting in significant savings in memory use.

**Protection and Security**

- Protection is concerned with controlling user or process access of the resources (hardware and software) in computer systems. An **access right** is a permission of a user to perform an operation on an object. A **domain** is a set of access rights. Processes execute in domains and may use any of the access rights in the domain to access/manipulate objects. This also implies that processes cannot perform an operation on an object if the domain does not specify.
- The role of security is to defend the system against internal or external attacks. There are a wide range of security threats such as denial-of-service, worms, viruses, and etc.

**Policy and Mechanism**

- The **policy** defines what needs to be done, while the **mechanism** specifies how it is actually implemented.
- This separation is necessary for flexibility so as to minimize the changes needed in the mechanism when the policy is changed.
- An operating system is designed with specific goals, which determines the policies, and these policies are implemented through specific mechanisms.

**Operating System Design**

- A **monolithic** operating system has little or no structure; all functionalities are provided in a single, static binary file that runs in a single *address space*. Although such systems are difficult to modify and debug, their primary benefit is *efficiency*.
- **Modularity** is an important technique in the design of OS or any complex software systems. It divides functions into different **modules**, which simplifies the debugging and verification. The OS functionalities can be divided into different modules. Interactions among those modules also need to be specified.
- **Layered approach** is a common type of modular design, in which each layer only interacts with two layers, utilizing the services provided by the lower layer except the bottom layer and providing services for the upper services except the top layer while implementing certain functions required for the specific layer. In operating systems, the bottom layer is usually the hardware interface and the highest layer is the user interface.
- A **microkernel approach** removes all non-essential components from the kernel, and implements them as system- or user-level programs. This results in a significantly smaller kernel (typically only including functions such as process, memory management, and IPC) that can be more easily ported from one hardware platform to another.
- The microkernel design uses a minimal kernel; most services run as user-level applications. Communications take place via **message passing**. Its performance might suffer due to increased system-function overhead, esp., inter-process communications or IPC between user and kernel space.
- Many OSes now support **loadable kernel modules**, which allow adding modules of different functionality to an operating system while it is executing.
- Generally, modern OS adopts a hybrid approach that combines several different types of structures for performance, security, and usability. For instance, Apple Mac OS X is a hybrid operating system; it is layered, the top is GUI (*Aqua*), the next layer is application environment and services including *Cocoa* programming environment. The bottom layer is the kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and other dynamically loadable modules. Linux is monolithic, with the operating system in a single address space provides very efficient performance. However, it also modular, so that new functionality can be dynamically added to the kernel.

**Microkernel**

- There are several advantages using a microkernel design: (a) adding a new service does not require modifying the kernel, (b) it is more secure as more operations are done in user mode than in kernel mode, and (c) a simpler kernel design and functionality typically results in a more reliable operating system.
- User programs and system services interact in a microkernel architecture by using interprocess communication (IPC) mechanisms - ***message passing***. The primary disadvantages of the microkernel architecture are that the overheads associated with interprocess communications and the frequent use of the

operating system's messaging functions in order to enable the user process and the system service to interact with each other.

**Loadable Kernel Modules or LKMs**

- LKM approach may be the best approach in contemporary operating system design, which combines the benefits of both layered and microkernel design techniques. In a modular design, the kernel needs only to maintain the capability to perform the required functions and know how to communicate with other modules. If more functionalities are required in the kernel, users can dynamically load corresponding modules into the kernel. The kernel has sections with well-defined, protected interfaces, a feature found in layered systems. This is more flexible in that each module can communicate with another module.
- As it is difficult to predict what features an operating system will need when it is being designed or/and implemented. The advantage of using **loadable kernel modules** is that functionality can be added to and removed from the kernel while it is running. There is no need to either recompile or reboot the kernel.

**Mac OS X and iOS**

- They are similar in architecture with user interface, programming or language support, graphics and media services, and the kernel environment - ***Darwin*** includes the Mach microkernel and the BSD UNIX kernel.
- They are different in, (1) Mac uses *Aqua* UI for a mouse or trackpad, iOS uses *Springboard* UI for touch devices; (2) Mac uses *Cocoa programming environment* with an API for the Objective-C, and iOS uses *Cocoa Touch* Objective-C API; (3) Mac uses *Core frameworks* supporting graphics and media services, while iOS uses *Media services* for graphics, audio, video, and *Core services* for cloud computing and databases.
- The kernel environment - Darwin consists of Mach microkernel and BSD UNIX kernel – a hybrid system with two system-call interfaces – (1) Mach system for memory management, CPU scheduling, and IPC facilities; (2) BSD system calls for POSIX functionality for networking, security, and programming language.

**iOS and Android**

- Similarities: (1) Both are based on the existing kernels (Linux and Mac OS X). (2) Both adopt an architecture that uses software stacks. (3) Both provide frameworks for developers.
- Differences: (1) iOS is closed-source, and Android is open-source. (2) iOS applications are developed in Objective-C, Android in Java. (3) Android uses a virtual machine, and iOS executes code natively.

**OS implementation**

- There are several advantages of using high-level languages to implement OSes. The code can be written faster, is more compact and easier to understand and debug. In addition, advances in compiler technique improve the generated

(binary) code for the entire operating system by simple recompilation. Finally, the OS is far easier to port — to move to some other hardware platform.