This chapter presents the high-level description of the operating-system interfaces that users or programmers actually see including system calls. There are several important concepts including layered design, virtual machines, system design and implementation, system generation, and the policy/mechanism difference.

Operating System Services

- Operating systems offer two categories of services and functions. One class of services is to enforce protection among different processes running concurrently in the system. For instance, processes are allowed to access only those memory locations that are associated with their address space; a process is not allowed to access devices directly without operating system intervention. The second class of services is to provide functionalities that are not supported directly by the underlying hardware. Virtual memory and file system are two of such examples.
- Operating system provides a wide range of services, such as user interface (UI), program execution, I/O and file system management, communications, and error detection. This can also be in the forms of resource allocation, accounting, security and protection.
- At the low level, it provides **system calls**, which, through API, allow a running program to request services from operating system directly. For example, user can use printf(), high-level API, which invokes the system call write().
- At the high level, it provides **system programs** for users to issue requests without the need for writing programs.
- The types of user requests vary according to different level. The system-call level provides the basic functions, such as process control, file and device manipulation. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls.

System Calls

- The **system call interface** of a programming language serves as a link to system calls made available by the OS. This interface intercepts function calls in the API and invokes the necessary system call within the operating system.
- There is a need 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 during system calls. The simplest approach is to pass the parameters in *registers*. In some cases, there may be more parameters than registers, in which 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 the *stack* by the program and popped out of the stack by the OS.

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 to minimize the changes needed when either policy or mechanism is changed.

Operating System Design

- **Modularity** is an important technique in the design of OS, or for any complex software systems. It divides functions into different modules, which simplifies the debugging and system verification. The OS functionalities are 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 at most 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 by implementing certain functions required for the specific layer.
- **Microkernel** removes all non-essential components from the kernel, and implements them as user-level programs. This results in a smaller kernel that can be more easily ported from one hardware platform to another, but its performance might suffer due to the increased system-function overhead.
- Many OS now supports dynamically loaded modules, which allow adding functionality to an operating system while it is executing.
- Generally, modern OS adopt a hybrid approach that combines several different types of structures. 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 service including *Cocoa* programming environment. The bottom is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and other dynamically loadable modules.

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 mechanisms such as **message passing**. The primary disadvantages of the microkernel architecture are the overheads associated with interprocess communication 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.

Modules

- Modular kernel approach may be the best of the current operating system design techniques, which combines the benefits of both the layered and microkernel design techniques. In a modular design, the kernel needs only to have the capability to perform the required functions and know how to communicate between modules. However, if more functionality is required in the kernel, then the user can dynamically load modules into the kernel. The kernel can have sections with well-defined, protected interfaces, a desirable property found in layered systems. More flexibility can be achieved by allowing the modules to communicate with one another.
- As it is difficult to predict what features an operating system will need when it is being designed. 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.

iOS and Android

- Similarities: (1) Both are based on existing kernels (Linux and Mac OS X). (2) Both have 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 language to implement an OS. The code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, the OS is far easier to port — to move to some other hardware.

Virtualization

- **Emulation** is used when the source CPU type is different from the target CPU type. It allows applications complied for the source CPU to run on the target CPU.
- **Virtualization** is a technology that allows an operating system to run as an application within another operating system.
- The **virtual machine** or **VM** creates an illusion for multiple processes in that each process “thinks” it runs on a dedicate CPU (processor) with its own memory.
- There are a number of advantages in using virtual machine, software emulation of an abstract machine, programming simplicity, fault isolation (e.g., easy debugging as errors do not crash the whole machine), protection and portability (e.g., Java across multiple platforms).