CHAPTER 1:

INTRODUCTION

* Operating System
  + OS is a program
  + It manages Computer’s H/W
  + OS provides basis for Application programs
  + Also act as an intermediary between computer user and computer H/W.
  + An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.
  1. What OSs do

Components of computer system:

1. Computer H/W: provides the basic computing resources for the system like CPU, memory, I/O devices.
2. OS: The operating system controls the hardware and coordinates its use among the various application programs for the various users.
3. System and Application Programs: Define the ways in which these resources are used to solve users’ computing problems.
4. Users
   * 1. User View of OS
     2. System Side view of OS
     3. Defining OS

The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. **The common functions of controlling and allocating resources are then brought together into one piece of software: i.e. the operating system.**

**A more common definition, and the one that we usually follow, is that the operating system**

**is the one program running at all times on the computer—usually called the kernel. (Along with the kernel, there are two other types of programs: system programs, which are associated with the operation of operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the operating system.)**

Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple’s iOS and Google’s Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name a only few).

* 1. Computer-System Organization

Before we can explore the details of how computer systems operate, we need general knowledge of the **structure of a computer system**.

* + 1. Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory.

The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (ROM) or **electrically erasable programmable read-only memory (EEPROM),** known by the general term **firmware**.

The bootstrap program must locate the operating-system **kernel** and load it into memory for OS execution.

Once the kernel is loaded and executing, it can start providing **services** to the system and its users. Some services are provided outside of the kernel, by **system programs** that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running. On UNIX, the first **system process is “init,”** and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time **by sending a signal to the CPU, usually by way of the system bus.** **Software may trigger an interrupt by executing a special operation of OS called a system call (also called a monitor call).**

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a **fixed location**. The fixed location usually contains the starting address where the service routine for the interrupt is located. The **interrupt service routine** executes; on completion, the CPU resumes the interrupted computation.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common.

**Since only a predefined number of interrupts is possible**, **a table of pointers** to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This **array, or interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.

* + 1. Storage Structure

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from **rewritable memory**, called **main memory** (also called random-access memory, or RAM). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM).**

All forms of memory provide an array of bytes. **Each byte has its own address**. Interaction is achieved through a **sequence of load or store instructions** to specific **memory addresses**. The **load instruction** moves a byte or word from main memory to an internal register within the CPU, whereas the **store instruction** moves the content of a register to main memory. Aside from

explicit loads and stores, the CPU automatically loads instructions from main memory for execution.

A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register/cache**. The instruction is then **decoded** and may cause operands to be fetched from memory and stored in some **internal register**. After the instruction on the operands has been executed, the result may be stored back in memory.

Notice that the memory unit sees only a **stream of memory addresses**. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore how a memory address is generated by a program. **We are interested only in the sequence of memory addresses generated by the running program.**

Due to volatile nature of Main memory thus, most computer systems provide secondary storage as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The wide variety of storage systems can be organized in a hierarchy according to speed, cost, size and volatility as follows…

Register > Cache > Main Memory > Solid State Disk > Magnetic Disk > Optical Disk > Magnetic Tapes

In this Storage Systems top 4 are constructed using Semiconductor Technology /Memory. In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile.

Caches can be installed to improve performance where a large disparity in access time or

Transfer rate exists between two components.

* + 1. I/O Structure

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI) controller**.

A device controller maintains some **local buffer storage and a set of special-purpose registers**. The device controller is responsible for moving the data between the **peripheral devices that it controls and its local buffer storage**. Typically, operating systems have a **device driver** for each device controller. **This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device**.

**To start an I/O operation**, the device driver loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information.

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA) is used**. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use **switch** rather than **bus architecture**. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective.

* 1. Computer-System Architecture

A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

* + 1. Single-Processor Systems

On a singleprocessor system, there is one main CPUcapable of executing a general-purpose instruction set, including instructions from user processes.

Almost all singleprocessor systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system. All of these special-purpose processors run a limited instruction set and do not run user processes.

The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

* + 1. Multiprocessor Systems

Within the past several years, multiprocessor systems (also known as parallel systems or multicore systems) have begun to dominate the landscape of computing. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor systems have three main advantages:

1. Increased throughput
2. Economy of scale
3. Increased reliability

If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

Increased reliability of a computer system is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. Some systems go beyond graceful degradation and are called fault tolerant, because they can suffer a failure of any single component and still continue operation.

Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected. The HP NonStop (formerly Tandem) system uses both hardware and software duplication to ensure continued operation despite faults.

The multiple-processor systems in use today are of two types.

Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A boss processor controls the system; the other processors either look to the boss for instruction or have predefined tasks. This scheme defines a boss–worker relationship. The boss processor schedules and allocates work to the worker processors.

The most common systems use **symmetric multiprocessing (SMP),** in which **each processor performs all tasks** within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided **if the processors share certain data structures.** A multiprocessor system of this form will allow processes and resources—such as memory— to be shared dynamically. Virtually all modern operating systems—including Windows, Mac OS X, and Linux—now provide support for SMP.ly among the various processors and can lower the variance among the processors.

Either way, multiprocessing can cause a system to change its memory access model from uniform memory access (UMA) to non-uniform memory access (NUMA). UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With NUMA, some parts of memory may take longer to access than other parts, creating performance penalty. (Operating systems can minimize the NUMA penalty through resource management).

A recent trend in CPU design is to include multiple computing cores on a single chip. Such multiprocessor systems are termed multicore. They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips. It is important to note that while multicore systems are multiprocessor systems, not all multiprocessor systems are multicore, these multicore CPUs appear to the operating system as N standard processors.

Finally, blade servers are a relatively recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

* + 1. Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs. The generally accepted definition is that **clustered computers share storage and are closely linked via a local-area network LAN** (as described in Chapter 17) or a faster interconnect, such as **InfiniBand**. Clustering is usually used to provide high-availability service—that is, service will continue even if one or more systems in the cluster fail. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of

the applications see only a brief interruption of service.

Clustering can be structured asymmetrically or symmetrically. In asymmetric clustering, one machine is in hot-standby mode while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server. In symmetric clustering, two or more hosts are running

applications and are monitoring each other. This structure is obviously more efficient, as it use all of the available hardware. However it does require that more than one application be available to run.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide high-performance computing environments. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as parallelization, which divides a program into separate components that run in parallel on individual computers in the cluster.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Chapter 17). Parallel clusters allow multiple hosts to access the same data on shared storage.

Cluster technology is changing rapidly. Some cluster products support dozens of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs),** as described in Section 10.3.3, which allow many systems to attach to a pool of storage. If the **applications and their data are stored on the SAN,** then the **cluster software** can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.8 depicts the general structure of a clustered system.

* 1. OS Structure

An operating system provides the environment within which programs are executed. One of the most important aspects of operating systems is the ability to multiprogramming. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory. Multi programmed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively.

Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

**Time sharing requires an interactive computer system**, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the response time should

be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user .As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

**A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.** Each user has at least one separate program in memory. A program loaded into memory and executing is called a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Input, for example, may be bounded by the user’s typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision **involves job scheduling.**

In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**.

Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage,

and memory management. We discuss these considerations throughout the text. In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through swapping, whereby processes are swapped in and out of main memory to the disk. Amore common method for ensuring reasonable response time is virtual memory, a technique that allows the execution of a process that is not completely in memory.

The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory. Further, it abstracts main memory into a large, uniform array of storage, separating logical memory as viewed by the user from physical memory.

This arrangement frees programmers from concern over memory-storage limitations.

* 1. OS Operations
     1. Dual-Mode and Multimode Operations

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code.

At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program’s behalf. Accordingly, most contemporary operating systems—such as Microsoft Windows 7, as well as Unix and Linux—take advantage of this dual-mode feature and provide greater protection for the operating system.

* + 1. Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a timer.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

* 1. Process Management

Computing resources are either given to the process when it is created or allocated to it while it is running. We emphasize that a program by itself is not a process. A program is a passive entity, like the contents of a file stored on disk, whereas a process is an active entity. A single-threaded process has one program counter specifying the next instruction to execute. (Threads are covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

The operating system is responsible for the following activities in connection with process management:

• Scheduling processes and threads on the CPUs

• Creating and deleting both user and system processes

• Suspending and resuming processes

• Providing mechanisms for process synchronization

• Providing mechanisms for process communication

* 1. Memory Management

Main memory is a repository of quickly accessible data shared by the CPU and I/O devices.

The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). As noted earlier, the main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from

disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them. For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses.

To improve both the utilization of the CPU and the speed of the computer’s response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the hardware design of the system. Each algorithm

requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

• Keeping track of which parts of memory are currently being used and who is using them

• Deciding which processes (or parts of processes) and data to move into and out of memory

• Allocating and deallocating memory space as needed.

* 1. Storage Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. The operating system maps files onto physical media and accesses these files via the storage devices.

* + 1. File-System Management

Each storage medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

The operating system implements the abstract concept of a file by managing mass-storage media, such as tapes and disks, and the devices that control them.

The operating system is responsible for the following activities in connection with file management:

• Creating and deleting files

• Creating and deleting directories to organize files

• Supporting primitives for manipulating files and directories

• Mapping files onto secondary storage

• Backing up files on stable (nonvolatile) storage media

* + 1. Mass-Storage Management

The operating system is responsible for the following activities in connection with disk management:

• Free-space management

• Storage allocation

• Disk scheduling

The entire speed of operation of a computer may hinge on the speeds of the disk

Subsystem and the algorithms that manipulate that subsystem.

Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

* + 1. Caching

Internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they

are outside the control of the operating system. Because caches have limited size, cache management is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance. Various replacement algorithms for software-controlled caches are discussed in Chapter 9.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system. In a hierarchical storage structure, the same data may appear in different levels of the storage system.

In multiprocessor environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called cache coherency, and it is usually a hardware issue (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible.

* + 1. I/O Systems

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O subsystem.

The I/O subsystem consists of several components:

• A memory-management component that includes buffering, caching, and spooling

• A general device-driver interface

• Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

* 1. Protection and Security

Will go through latter…

* 1. Kernel Data Structures

We turn next to a topic central to operating-system implementation: the way data are structured in the system. In this section, we briefly describe several fundamental data structures used extensively in operating systems.

* + 1. Lists, Stacks and Queues

Just go through the book part as well as DATASTRUCTURES part in Studytonight portal.

* + 1. Trees

Just go through the book part as well as DATASTRUCTURES part in Studytonight portal.

* + 1. Hash Functions and Maps

Just go through the book part.

* + 1. Bitmaps

Just go through the book part.

* 1. Computing Environments

We turn now to a discussion of how operating systems are used in a variety of computing environments.

* + 1. Traditional Computing

In the latter half of the 20th century, computing resources were relatively scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive. Batch systems processed jobs in bulk, with predetermined input from files or other data sources. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Today, traditional time-sharing systems are uncommon. The same scheduling technique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time. Even a web browser can be composed of multiple processes, one for each website currently being visited, with time sharing applied to each web browser process.

* + 1. Mobile Computing

Mobile computing refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight.

Many developers are now designing applications that take advantage of the unique features of

mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. Power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

* + 1. Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface’s device driver. Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system’s utility and popularity.

A network, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP, including all general-purpose ones. Some systems support proprietary protocols to suit their needs. To an operating system, a network protocol simply needs an interface device—a network adapter, for example—with a device driver to manage it, as well as software to handle data. These concepts are discussed throughout this book.

Networks are characterized based on the distances between their nodes. A local-area network (LAN) connects computers within a room, a building, or a campus. A wide-area network (WAN) usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols. The

continuing advent of new technologies brings about new forms of networks. For example, a metropolitan-area network (MAN) could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a personal-area network (PAN) between a phone and a headset or a smartphone and a desktop computer

Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A network operating system is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all

other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network.

* + 1. Client-Server Computing

As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs and mobile devices. Correspondingly, user-interface functionality once handled directly by centralized systems is increasingly being handled by PCs, quite often through a web interface. As a result, many of today’s systems act as server systems to satisfy requests

generated by client systems. This form of specialized distributed system, called a client–server system, has the general structure depicted in Figure 1.18.

Server systems can be broadly categorized as compute servers and file servers:

• The compute-server system provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that responds to client requests for data is an example of such a system.

• The file-server system provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.

* + 1. Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but

in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

Determining what services are available is accomplished in one of two general ways:

• When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.

• An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a discovery protocol must be provided

that allows peers to discover services provided by other peers in the network.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another. The Napster system used an approach similar to the first type described above: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used

a technique similar to the second type: a client broadcasted file requests to other nodes in the system, and nodes that could service the request responded directly to the client. The future of exchanging files remains uncertain because peer-to-peer networks can be used to exchange copyrighted materials (music, for example) anonymously, and there are laws governing the distribution of copyrighted material. Notably, Napster ran into legal trouble for copyright infringement and its services were shut down in 2001.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as voice over IP (VoIP). Skype uses a hybrid peer-to- peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

* + 1. Virtualization

Virtualization is a technology that allows operating systems to run as applications within other operating systems. Broadly speaking, virtualization is one member of a class of software that also includes emulation. Emulation is used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called “Rosetta,” which allowed applications compiled for the IBM CPU to run on the Intel CPU.

That same concept can be extended to allow an entire operating system written for one platform to run on another. Emulation comes at a heavy price, however. Every machine-level

Instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code can run much slower than the native code.

A common example of emulation occurs when a computer language is not compiled to native code but instead is either executed in its high-level form or translated to an intermediate form. This is known as interpretation. Some languages, such as BASIC, can be either compiled or interpreted. Java, in contrast, is always interpreted. Interpretation is a form of emulation in that the high-level language code is translated to native CPU instructions, emulating not another CPU but a theoretical virtual machine on which that language could run natively. Thus, we can run Java programs on “Java virtual machines,” but technically those virtual machines are Java emulators.

With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently. Running multiple virtual machines allowed (and still allows) many users to run tasks on a system

designed for a single user. Later, in response to problems with running multiple Microsoft Windows XP applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on XP. That application ran one or more guest copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.20.) Windows was the host operating system, and the VMware application was the

virtual machine manager VMM. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

Even though modern operating systems are fully capable of running multiple applications reliably, the use of virtualization continues to grow. On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host. For example, an Apple laptop running Mac OS

X on the x86 CPU can run a Windows guest to allow execution of Windows applications. Companies writing software for multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware, ESX, and Citrix XenServer no longer run on host operating systems but rather are the hosts. Full details of the features and implementation of virtualization are found in Chapter 16.

* + 1. Cloud Computing

Please go through the book for this component.

* + 1. Real time embedded Systems

Please go through the book for this component.

* 1. Open-Source Operating Systems

Open-source operating systems are those available in source-code format rather than as compiled binary code. Linux is the most famous opensource operating system, while Microsoft Windows is a well-known example of the opposite closed-source approach. Apple’s Mac OS X and iOS operating systems comprise a hybrid approach. They contain an open-source kernel

named Darwin yet include proprietary, closed-source components as well.

* + 1. History

Please go through the book for this component.

* + 1. Linux

Please go through the book for this component.

* + 1. BSD UNIX

Please go through the book for this component.

* + 1. Solaris

Please go through the book for this component.

* + 1. Open-Source Systems as learning Tools

Please go through the book for this component.

* 1. Summary

Please go through the book for this component.

CHAPTER 2:

OPERATING SYSTEM STRUCTURES

2

We can view an operating system from several vantage points. One view focuses on the **services** that the system provides; another, on the **interface** that it makes available to users and programmers; a third, on its **components** and their **interconnections**.

2.1 Operating-System Services

The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating system services are provided for the convenience of the programmer, to make the programming task easier. Figure 2.1 shows one view of the various operating-system services and how they interrelate.

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

* User Interface:

Almost all operating systems have a user interface (UI). This interface can take several forms. One is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a batch interface, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.

* Program Execution

Program execution. The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

* I/O Operations

A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly.

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

* File System Manipulation

The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and

list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific

features or performance characteristics.

* Communications

There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied

Together by a computer network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which **packets of information in predefined formats** are moved between processes by the operating system.

* Error Detection

The operating system needs to be detecting and correcting errors constantly. Errors may occur in the **CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time).** For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

Q) What is Parity?

Another set of operating system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

* Resource Allocation:

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have **CPU-scheduling routines** that take into account the **speed of the CPU**, the **jobs that must be executed**, the **number of registers available**, and **other factors**. **There may also be routines to allocate printers, USB storage drives, and other peripheral devices.**

* Accounting

We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

* Protection and Security

The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating

System itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain

access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and to recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

2.2 User and OS Interface

2.2.1 Command Interpreters

Some operating systems include the command interpreter in the kernel. Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX and Linux systems, a user may choose among several different shells, including the Bourne shell, C shell, Bourne-Again shell, Korn shell, and others. Third-party shells and free user-written shells are also available.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way.

These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For 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. 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.

An alternative approach—used by UNIX, among other operating systems —implements most commands through system programs. In this case, 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.

2.2.2` GUI

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system (Mac OS) has undergone various changes over the years, the most significant being the adoption of 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 operating system.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however. These include the Common Desktop Environment (CDE) and X-Windows systems, which are common on commercial versions of UNIX, such as Solaris and IBM’s AIX system. In addition, there has been 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. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.

2.2.3 Choice of Interface

It typically User interface is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system. In this book, we concentrate on the fundamental problems of providing adequate service to user programs. **From the point of view of the operating system, we do not distinguish between user programs and system programs.**

2.3 System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OSX), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called libc. Each operating system has its own name for each system call.

Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function CreateProcess() (which unsurprisingly is used to 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?

There are several reasons for doing so…

1. One benefit concerns program portability. An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences often make this more difficult than it may appear).
2. Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer.
3. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

For most programming languages, 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 operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.

Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.

The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time

support library.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and

length of the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call. Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. In some cases,

however, there may be more parameters 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 (Figure 2.7). This is the approach taken by Linux and Solaris. Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the

operating system. 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

2.4.2 File Management

2.4.3 Device Management

2.4.4 Information Maintenance

2.4.5 Communication

2.4.6 Protection

2.5 System Programs

2.6 OS Design and Implementation

2.6.1 Design goals

2.6.2 Mechanisms and Policies

2.6.3 Implementation

2.7 OS Structure

2.7.1 Simple Structure

2.7.2 Layered Approach

2.7.3 Microkernels

2.7.4 Modules

2.7.5 Hybrid Systems

2.7.5.1 Mac OS X

2.7.5.2 iOS

2.7.5.3 Android

2.8 OS Debugging

2.8.1 Failure Analysis

2.8.2 Performance Tuning

2.8.3 DTrace

2.9 OS Generation

2.10 System Boot

2.11 Summary