ACT

College of Computer Studies

COURSE NOTES iN

OPESYS:

Operating Systems

Course Description:

OPESYS deals with the key concepts and the general principles of the study of operating systems. This course focuses on the implementation of various operating systems from their structures, modules, components, and development. Students will learn to manipulate the special features of an operating system, configure the OS settings, create batch files and apply maintenance techniques in an operating system. The course consists of lectures and learning activities. Upon completion of this course, the students are expected to create their own batch files, manipulate and maintain an operating system.

Learning Outcomes:

- Compare and contrast the characteristics and features of various operating systems suited to solve complex problems through evaluation various case studies;
- Implement and simulate basic modules or components of an operating system;
- Participate in developing real-world systems that will suit the needs of the organization.

Tools or Application to Use:

- Student Achievement Monitoring System (SAMS)
- Schoology (web application)
- Messenger
- Other Applications

Mode of Assessment:

- Online Quiz
- Activities
- Recitation during synchronous class
- Presentation

References:

- Operating System Concepts, Silberschatz 10th Edition

1 WHAT OPERATING SYSTEMS DO

An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

An amazing aspect of operating systems is how they vary in accomplishing these tasks.

- Mainframe operating systems are designed primarily to optimize the utilization of hardware.
- Personal computer (PC) operating systems support complex games, business applications, and everything in between.
- Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs.

WHAT OPERATING SYSTEMS DO

A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and the ***users.***

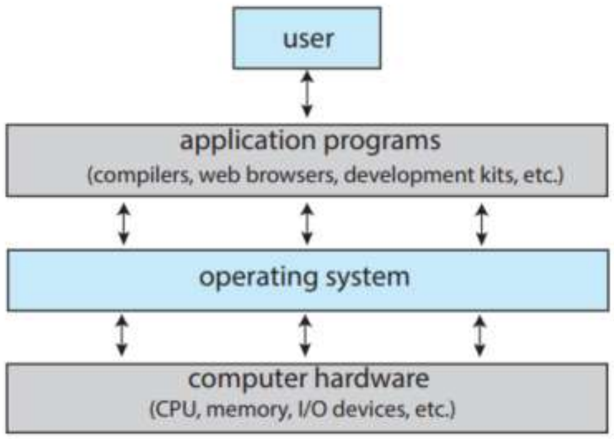
The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an ***environment*** within which other programs can do useful work.

User View

The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and security and none paid to **resource utilization**— how various hardware and software resources are shared.

Increasingly, many users interact with mobile devices such as smartphones and tablets — devices that are replacing desktop and laptop computer systems for some users. These devices are typically connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a **voice recognition** interface, such as Apple's **Siri**.


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

Some computers have little or no user view. For example, **embedded computers** in-home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems and applications are designed primarily to run without user intervention.

System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. 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.

Defining Operating Systems

How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute 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: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. 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 operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

The matter of what constitutes an operating system became increasingly important as personal computers became more widespread and operating systems grew increasingly sophisticated. Today, however, if we look at operating systems for mobile devices, we see that once again the number of features constituting the operating system is increasing. 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 only a few).

In summary, for our purposes, the operating system includes the always running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while it is running. Most of this text is concerned with the kernel of general-purpose operating systems, but other components are discussed as needed to fully explain operating system design and operation.

Hi I'm Flashee!

You have reached the end of the lesson. Be sure to answer the corresponding **activity of this lesson** on the activities folder of our class materials in the file server.

2 COMPUTER SYSTEM ORGANIZATION

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 between components and shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display). Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect. 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. 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.

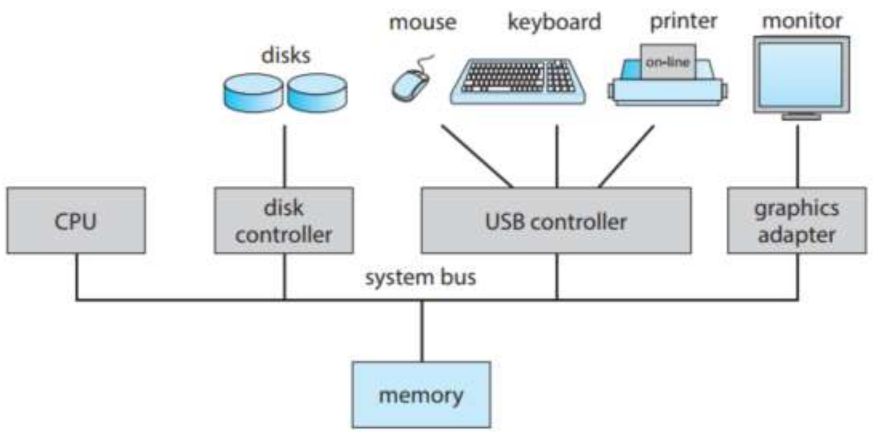


Figure 1.2 A typical PC computer system.

INTERRUPTS

Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in 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 that it has finished its operation. The device driver then gives control to other parts of 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 such as "write completed successfully" or "device busy". But how does the controller inform the device driver that it has finished its operation? This is accomplished via an **interrupt**.

Overview

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. (There may be many buses within a computer system, but the system bus is the main communications path between the major components.) Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

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. A timeline of this operation is shown in Figure 1.3.

Interrupts are an important part of computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for managing this transfer would be to invoke a generic routine to examine the interrupt information.

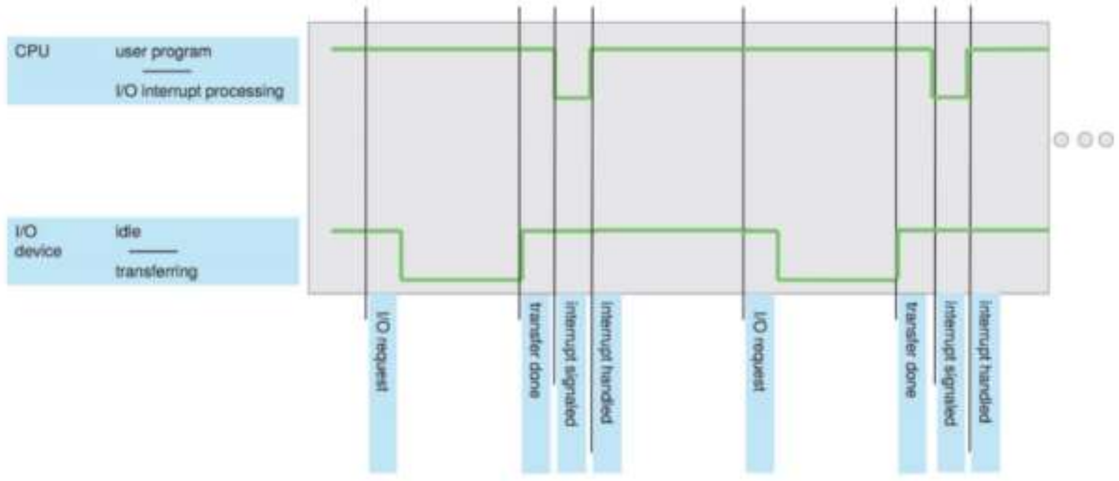


Figure 1.3 Interrupt timeline for a single program doing output.

The routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly, as they occur very frequently. 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 number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupt in this manner.

The interrupt architecture must also save the state information of whatever was interrupted so that it can restore this information after servicing the interrupt. If the interrupt routine needs to modify the processor state — for instance, by modifying register values — it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

Implementation

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector. It then starts execution at the address associated with that index. The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the interrupt handler, and the handler clears the interrupt by servicing the device. Figure 1.4 summarizes the interrupt-driven I/O cycle.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and the **interrupt-controller hardware.**

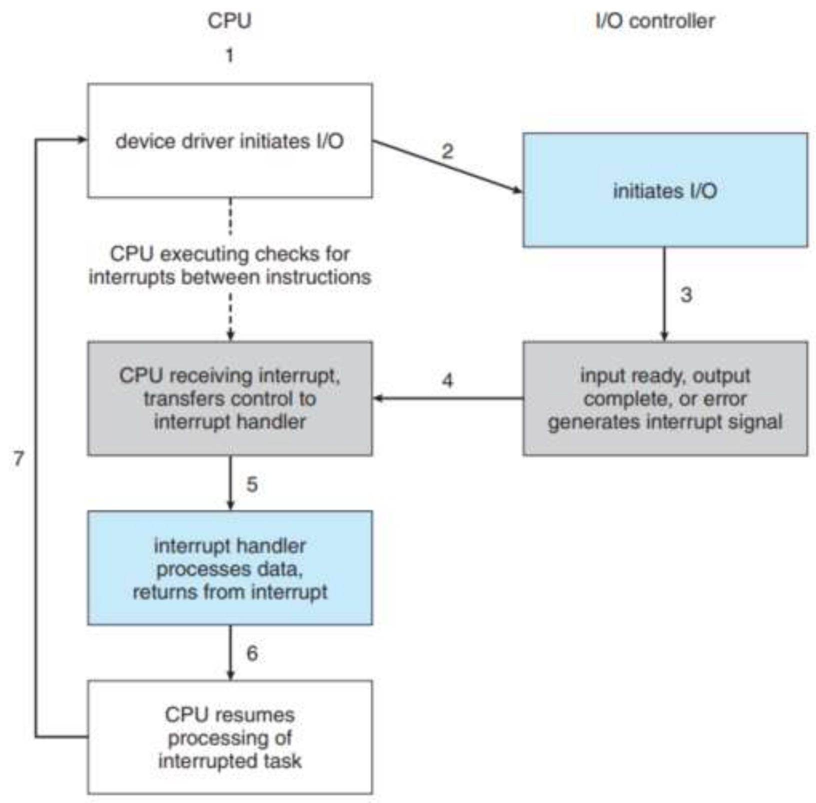

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

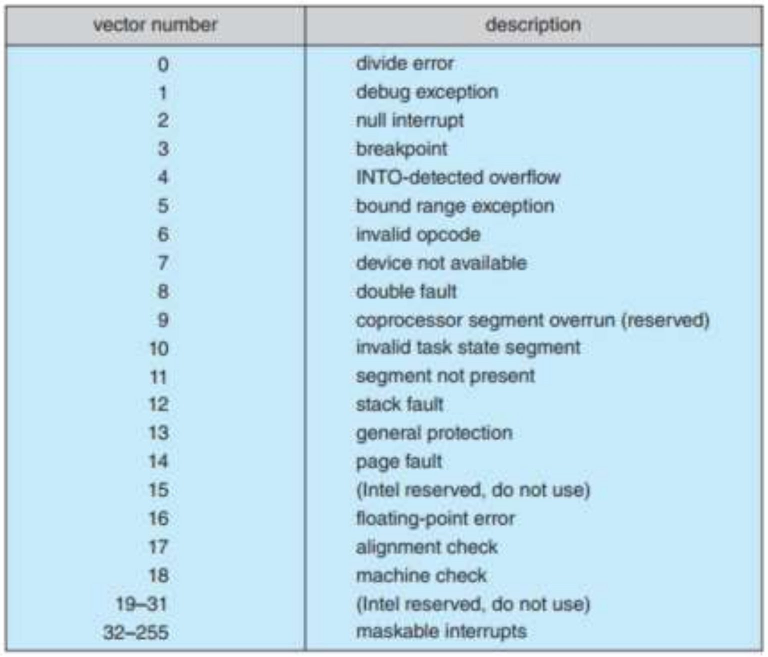
Figure 1.4 Interrupt-driven I/O cycle.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

Recall that the purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 1.5 illustrates the design of the interrupt vector for Intel processors. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and make it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 1.5 Intel processor event-vector table.

STORAGE STRUCTURE

The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. 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**).

Computers use other forms of memory as well. For example, the first program to run on computer power-on is a **bootstrap program**, which then loads the operating system. Since RAM is **volatile**— loses its content when power is turned off or otherwise lost — we cannot trust it to hold the bootstrap program. Instead, for this and some other purposes, the computer uses electrically erasable programmable read-only memory (EEPROM) and other forms of **firmware** — storage that is infrequently written to and is nonvolatile. EEPROM can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

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 from the location stored in the program counter.

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**. 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.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible on most systems for two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory, as mentioned, is volatile — it loses its contents when power is turned off or otherwise lost.

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 most common secondary-storage devices are **hard-disk drives (HDDs)** and **nonvolatile memory (NVM) devices**, which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing. Secondary storage is also much slower than main memory.

In a larger sense, however, the storage structure that we have described — consisting of registers, main memory, and secondary storage — is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or Blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes — to store backup copies of material stored on other devices, for example — are called **tertiary storage**. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.6) according to storage capacity and access time.

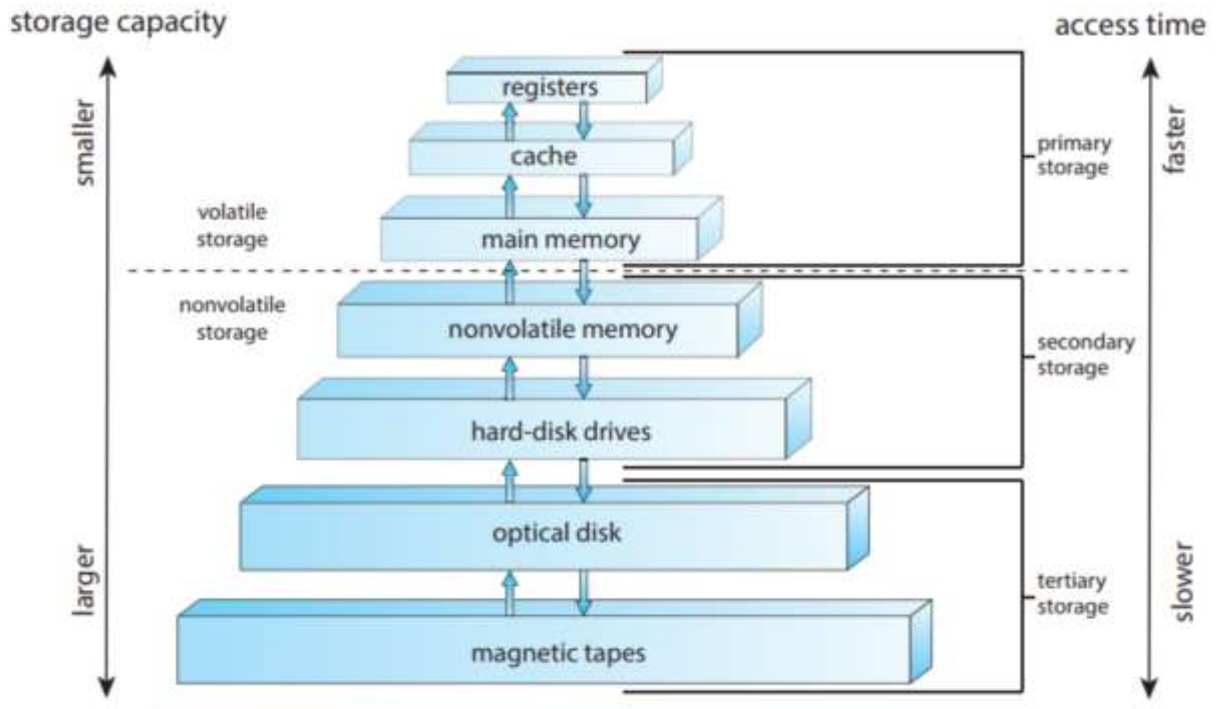


Figure 1.6 Storage-device hierarchy.

As a general rule, there is a trade-off between size and speed, with smaller and faster memory closer to the CPU. As shown in the figure, in addition to differing in speed and capacity, the various storage systems are either volatile or nonvolatile. Volatile storage, as mentioned earlier, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

The top four levels of memory in the figure are constructed using **semiconductor memory**, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

Since storage plays an important role in operating-system structure, we will refer to it frequently in the text. In general, we will use the following terminology:

• Volatile storage will be referred to simply as **memory**. If we need to emphasize a particular type of storage device (for example, a register), we will do so explicitly.

• Nonvolatile storage retains its contents when power is lost. It will be referred to as **NVS**. The vast majority of the time we spend on NVS will be on secondary storage. This type of storage can be classified into two distinct types:

◦ **Mechanical**. A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape. If we need to emphasize a particular type of mechanical storage device (for example, magnetic tape), we will do so explicitly.

◦ **Electrical**. A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD. Electrical storage will be referred to as NVM. If we need to emphasize a particular type of electrical storage device (for example, SSD), we will do so explicitly.

Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

The design of a complete storage system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile storage as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

I/O STRUCTURE

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

Recall from the beginning of this section that a general-purpose computer system consists of multiple devices, all of which exchange data via a common bus. The form of interrupt-driven I/O described in Section 1.2.1 is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as NVS 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 the device and main 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. Figure 1.7 shows the interplay of all components of a computer system.

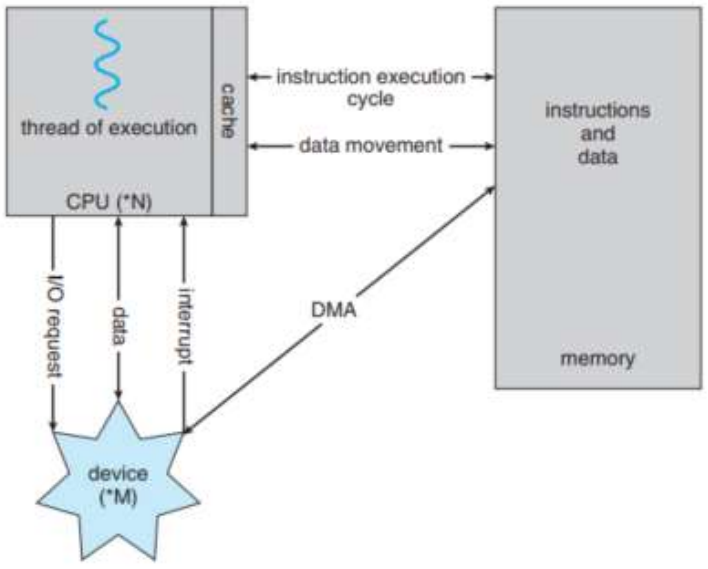


Figure 1.7 How a modern computer system works.

Hi I'm Flashee!

You have reached the end of the lesson. Be sure to answer the corresponding **activity of this lesson** on the activities folder of our class materials in the file server.

3 OPERATING SYSTEM SERVICES

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

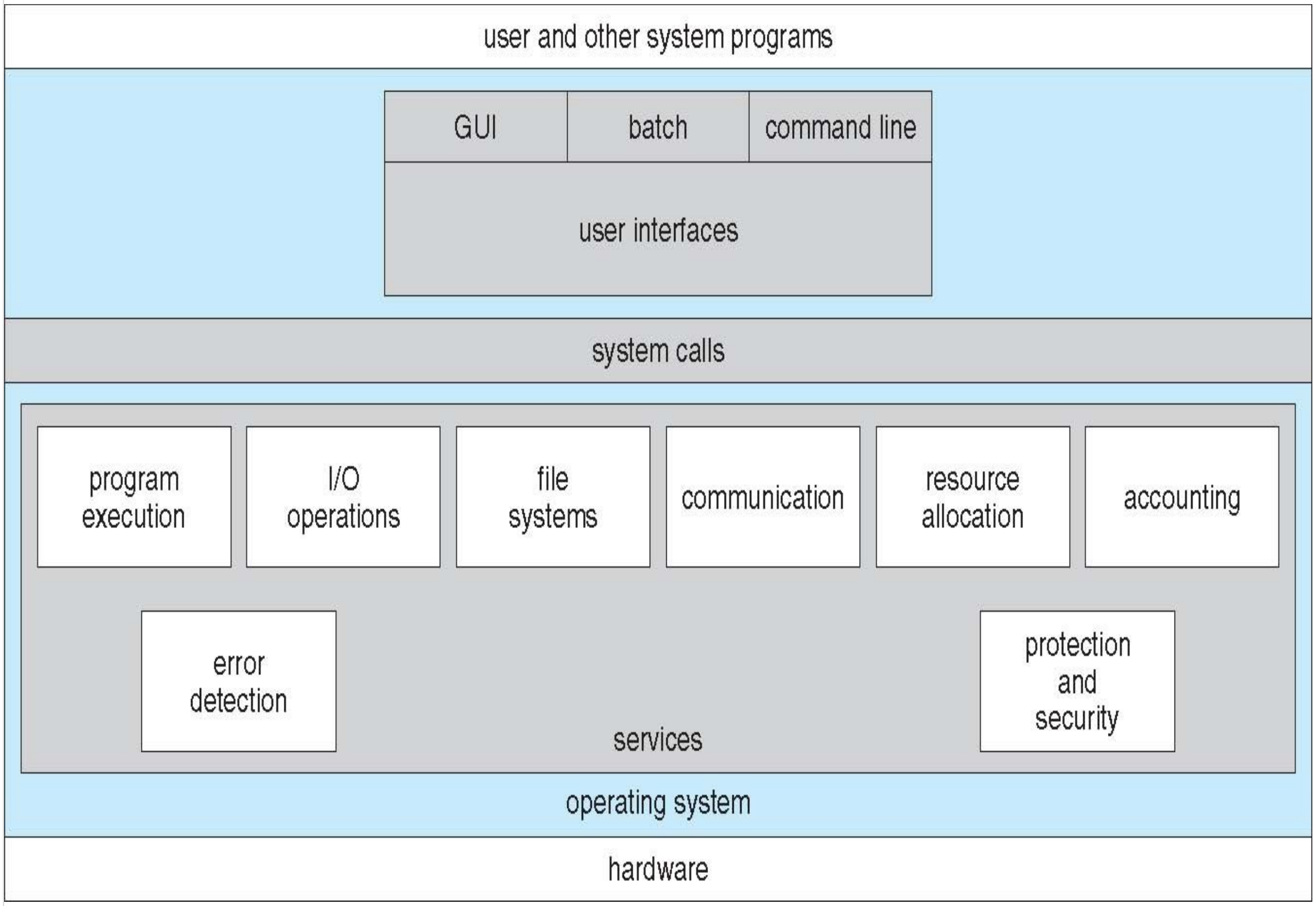
We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. We consider what services an operating system provides, how they are provided, how they are debugged, and what the various methodologies are for designing such systems.

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- 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**).
        - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
    - **Program execution** - The system must be able to load a program into memory and to run that program, end 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
- One set of operating-system services provides functions that are helpful to the user (Cont.):
    - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
    - **Communications** – Processes may exchange information, on the same computer or between computers over a network
        - Communications may be via shared memory or through message passing (packets moved by the OS)
    - **Error detection** – OS needs to be constantly aware of possible errors
        - May occur in the CPU and memory hardware, in I/O devices, in user program
        - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
        - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system
- Another set of OS function exists for ensuring the efficient operation of the system itself via resource sharing
    - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
        - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
    - **Accounting -** To keep track of which users use how much and what kinds of computer resources

- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services



Note:
Read our reference book to learn more about this lesson.

Hi I'm Flashee!

You have reached the end of the lesson. Be sure to answer the corresponding **activity of this lesson** on the activities folder of our class materials in the file server.

4 USER OPERATING SYSTEM INTERFACE

User Operating System Interface - CLI

- CLI or **command interpreter** allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs
    - If the latter, adding new features doesn’t require shell modification

User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc.
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

Touchscreen Interfaces

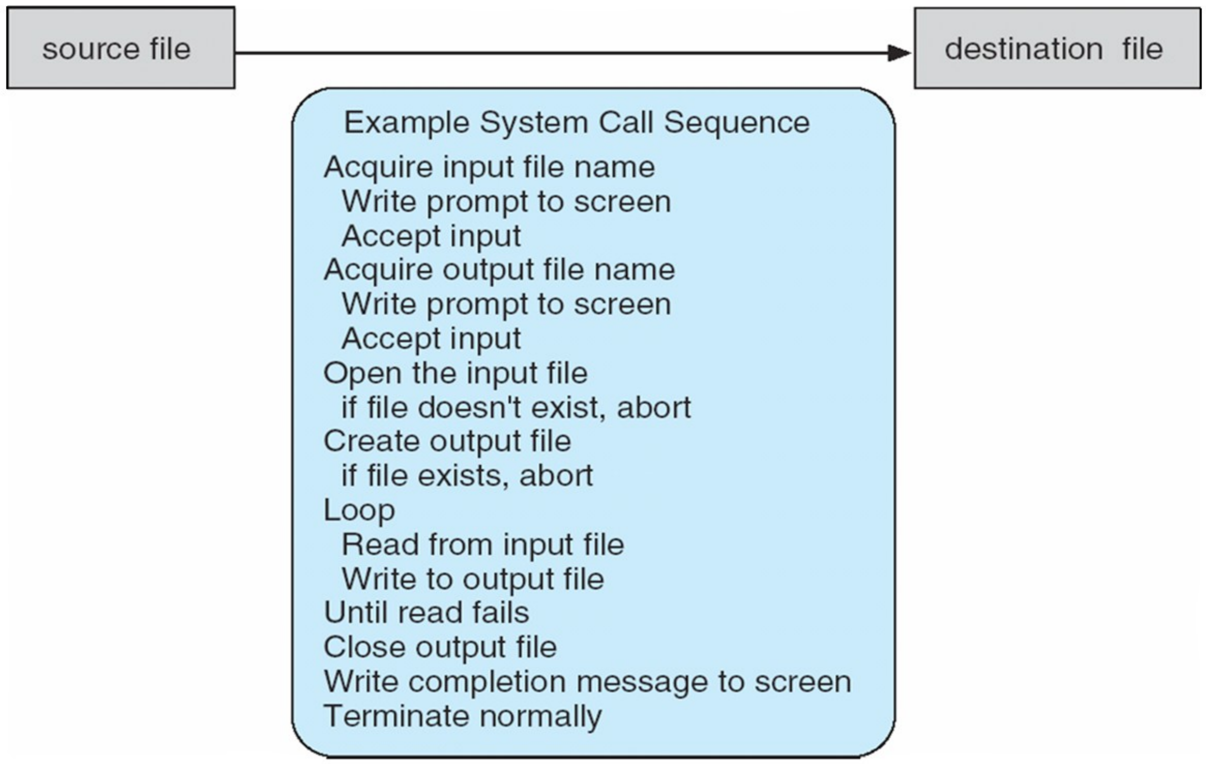
- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of System Calls

- System call sequence to copy the contents of one file to another file



Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

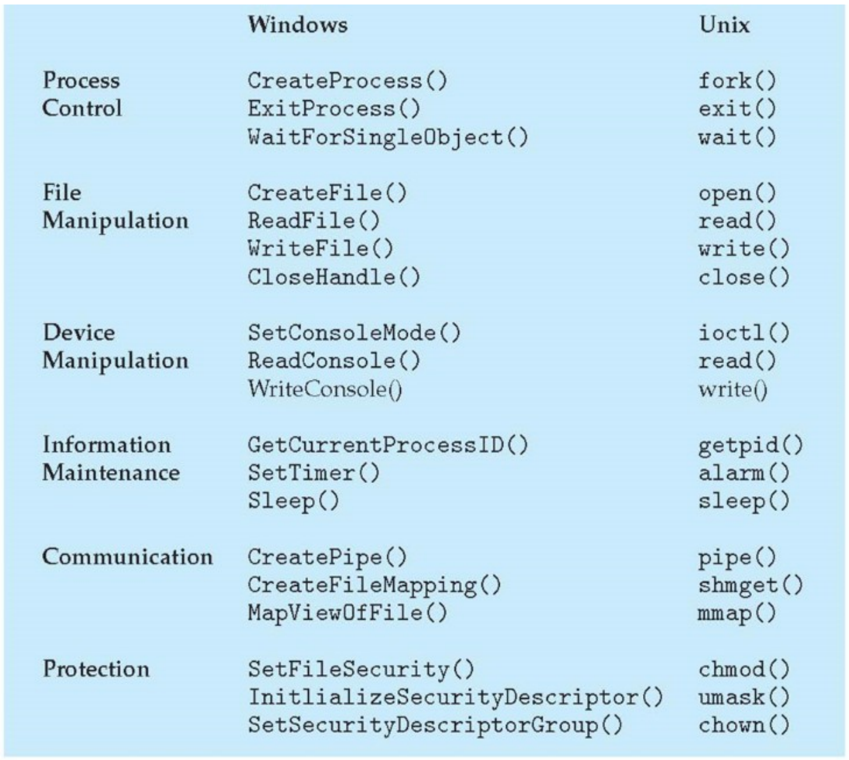
- Device management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices

- Information maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get and set process, file, or device attributes

- Communications
    - create, delete communication connection
    - send, receive messages if **message passing model** to **host name** or **process name**
        - From **client** to **server**
    - **Shared-memory model** create and gain access to memory regions
    - transfer status information
    - attach and detach remote devices

- Protection
    - Control access to resources
    - Get and set permissions
    - Allow and deny user access

Examples of Windows and Unix System Calls



	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
    - File manipulation
    - Status information sometimes stored in a File modification
    - Programming language support
    - Program loading and execution
    - Communications
    - Background services
    - Application programs
- Most users' view of the operating system is defined by system programs, not the actual system calls
- Provide a convenient environment for program development and execution
    - Some of them are simply user interfaces to system calls; others are considerably more complex

- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

- **Status information**
    - Some ask the system for info - date, time, amount of available memory, disk space, number of users
    - Others provide detailed performance, logging, and debugging information
    - Typically, these programs format and print the output to the terminal or other output devices
    - Some systems implement a **registry** - used to store and retrieve configuration information

- **File modification**
    - Text editors to create and modify files
    - Special commands to search contents of files or perform transformations of the text

- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided

- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
    - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

- **Background Services**
    - Launch at boot time
        - Some for-system startup, then terminate
        - Some from system boot to shutdown
    - Provide facilities like disk checking, process scheduling, error logging, printing
    - Run in user context not kernel context
    - Known as **services**, **subsystems**, **daemons**

- **Application programs**
    - Don't pertain to system
    - Run by users
    - Not typically considered part of OS
    - Launched by command line, mouse click, finger poke

Note:

Read our reference book to learn more about this lesson.

Hi I'm Flashee!

You have reached the end of the lesson. Be sure to answer the corresponding **activity of this lesson** on the activities folder of our class materials in the file server.

5 COMMAND LINE INTERFACE

File System

- A *file* is a collection of bytes of information treated as a single unit.
- It is given a *name* to make it easy to find and use later.
- The *file system* keeps track of where a file is actually resident on a disk.
- A disk (hard disk, floppy, optical disk) is subdivided into *directories* or *folders*.
- The top-level folder on a disk is known as the *root*.
- The root is generally subdivided into *subfolders*.
- Any folder or subfolder can contain files and other folders.
- The *fully-qualified filename* includes the name of the file and the *path* to the folder in which it resides: c:\courses\061\os\chapter4.ppt

Windows Registered File Types

- A particular file extension can be registered and associated with a particular program.
  - .docx files are associated with MS Word
  - .xlsx files are associated with MS Excel
  - .txt files are associated with Notepad
  - .html files are associated with IE
- This is how Windows knows what to do when you double-click a file in My Computer.

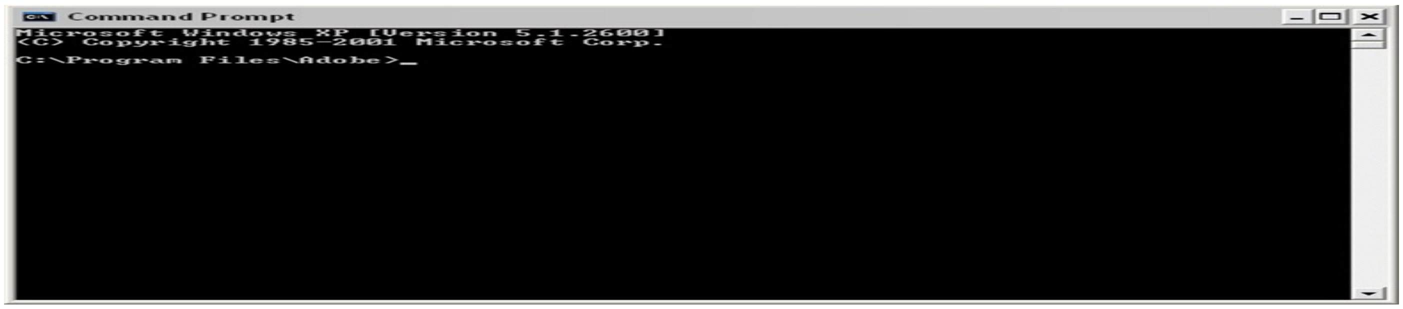
File Attributes

- Each file has four attributes which can be viewed or set.
  - **Read Only** – File may be viewed, copied, executed (if appropriate), but not changed. It can be deleted.
  - **Hidden** – File will not be displayed in normal list of files.
  - **System** – File is identified to belong to system, should not be messed with.
  - **Archive** – File is (or is not) a candidate for backup.

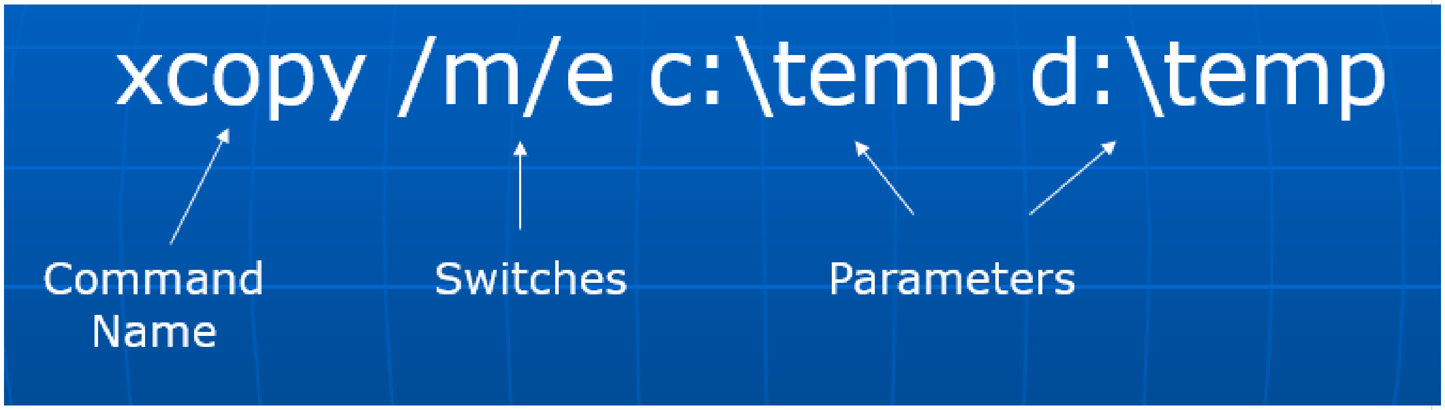
DOS Commands

- Two types – Internal and External
  Internal commands are resident in the main kernel file: command.com (or cmd.exe)
  External commands are separate little programs.
- It's important to learn DOS commands because you can write scripts to execute a set of commands automatically.

The Command Prompt


```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Program Files\Adobe>_
```

Three Parts of a DOS Command



DOS Wildcard Characters

- The characters ? and * can be used to affect multiple files with a single command.
    - The ? means any single character.

      copy c:\temp\notes??.doc d:\temp

      means copy any Word file that begins with the word "notes" with exactly two other characters, like "notes01.doc", "notesAB.doc", etc.

    - The "*" wildcard replaces any number of characters.

      copy c:\temp\notes.* d:\temp
      (copy all files with the name "notes" and any extension.)

      copy *.doc c:\temp
      (copy all files with a "doc" extension in the current directory.)

At the Command Prompt

- A drive letter and a ":" (e.g. "f:") makes that your current drive.
- CD (Change Directory)
  cd (with no parameters) reminds you what the current directory is.
  cd.. moves you to the parent of the current directory (up one level).
  cd \ moves you to the root of the current drive.
  cd <some directory> makes that your current directory.

- MD – Make directory.
- RD – Remove a directory or an entire directory tree.
- DIR – Display the contents of a directory.
- DEL (or ERASE) – Deletes one or more files.
- COPY – Places a copy of file(s) in a different folder.
- XCOPY – Flexible copy command used for copying large groups of files, commonly used for file backup.
- MOVE – Moves file(s) from one folder to another.
- REN(AME) – Renames file(s).
- ATTRIB – Displays or sets file attributes.
- FORMAT – Formats a disk.
- CHKDSK – Tests the file system on a disk, and reports status.
- DATE and TIME – Display & set the current date & time in the PC.
- TYPE – Displays the contents of a text file.
- CLS – Clears the screen display

Hi I'm Flashee!

You have reached the end of the lesson. Be sure to answer the corresponding **activity of this lesson** on the activities folder of our class materials in the file server.

6 BATCH COMMANDS IMPLEMENTATION

Below is a listing of commands used in a batch file with additional information about each of the commands.

Tip: Just like all commands, all batch file commands are not case sensitive. However, we listed the batch file commands in all caps to help with identification.

@

The at symbol does not echo back text after the symbol. The @ is most often used as **@ECHO OFF** to prevent any of the commands in the batch file from being displayed, just the information outputted by the command.

%1

The percent followed by a numeric value, beginning with one, allows you to add matched variables to a batch file. The line below is an example of what can be used in a batch file.

```
ECHO Hello %1
```

With a batch file containing the above line if you type myname (name of bat file) and then your name, as shown below.

```
myname Bob
```

It would output "Hello Bob" because "Bob" is the first matched text.

Tip: You can keep going to %2, %3, and so on. For example, you could use %2 for a middle name and %3 as the last name.

::

Two colons in front of any line are one of two ways of adding remarks into the batch file without displaying or executing that line when the batch file is run. Unlike REM, this line is not shown regardless if ECHO off is in the batch file.

:LABEL

By adding a colon in front of a word, such as LABEL, you create a category, more commonly known as a label. A label allows you to skip to certain sections of a batch file such as the end of the batch file. Also see GOTO.

CALL

A call is used to run another batch file within a batch file. When the batch file that is called is completed, the remainder of the original batch file is completed. If the batch file does not exist, you get an error.

CHOICE and SET

See our how to use choice and set in a batch file page for an example of how you can create options in your batch file. Further information about each of these commands can also be found on the choice and set page.

CLS

Just like the DOS command would clear your screen. We find it helpful to run the cls command at the top of your batch file to clear any previous commands or output and make any output from the batch file easier to find and read.

ECHO

Echo a message in the batch file. Such as **ECHO Hello World** prints *Hello World* on the screen when executed.

Note: Without **@ECHO OFF** at the beginning of the batch file you'll also get "ECHO Hello World" and "Hello World."

Tip: If you'd just like to create a blank line, type **ECHO.** adding the period at the end creates an empty line.

EXIT

Exits out of the DOS window if the batch file is running from Windows. See the exit command page for further information on this command.

GOTO

Jumps to a label or section of a batch file. The goto can make it easy to jump back to the start or end of a batch file if a condition is met, or an error occurs. See our how to use choice and set in a batch file page for an example of how goto can be used.

IF

Used to check for a certain condition if the condition exists. If that condition exists it performs that function. See the if command for further information on this command.

PAUSE

Prompt the user to press any key to continue.

REM

One of two ways of adding remarks into the batch file without displaying or executing that line when the batch file is run.

SHIFT

The shift command changes the position of replaceable parameters in a batch program. See the shift page for further information on this command.

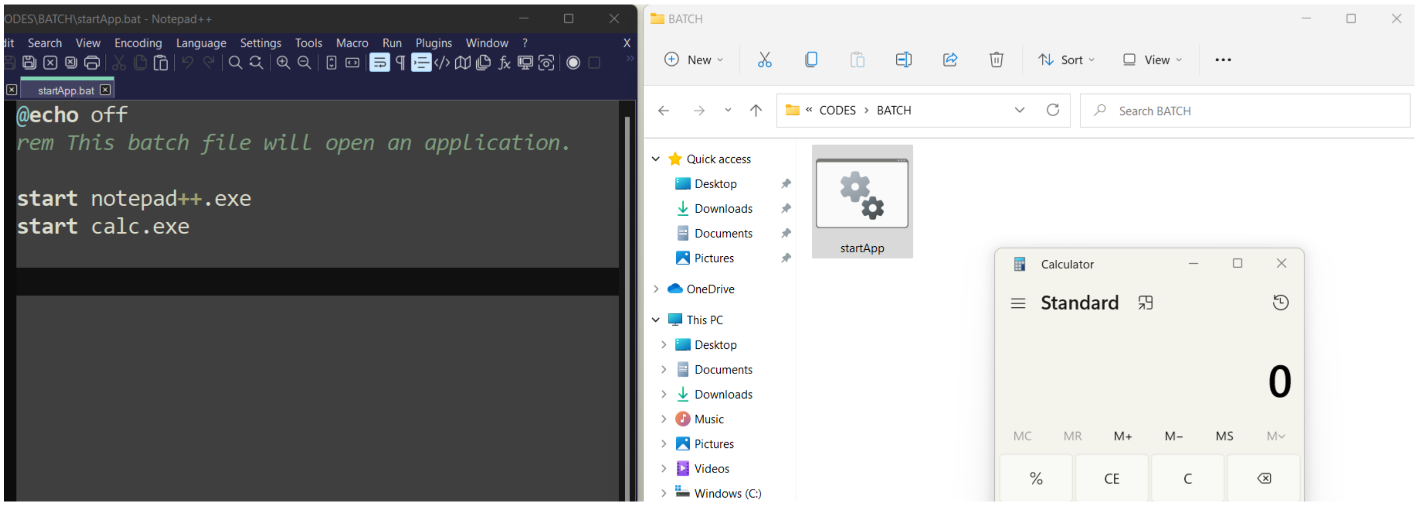
START

Used to open Windows programs. For example, **START C:\WINDOW\CALC** would run the Windows Calculator. The start command can also be used to start any file Windows recognizes. For example, you could start a movie or audio file in a batch file to start your default player for that file.

NOTE: To write the batch scripts, you may use any text editor. To create a batch file, you have to save your batch scripts into .bat

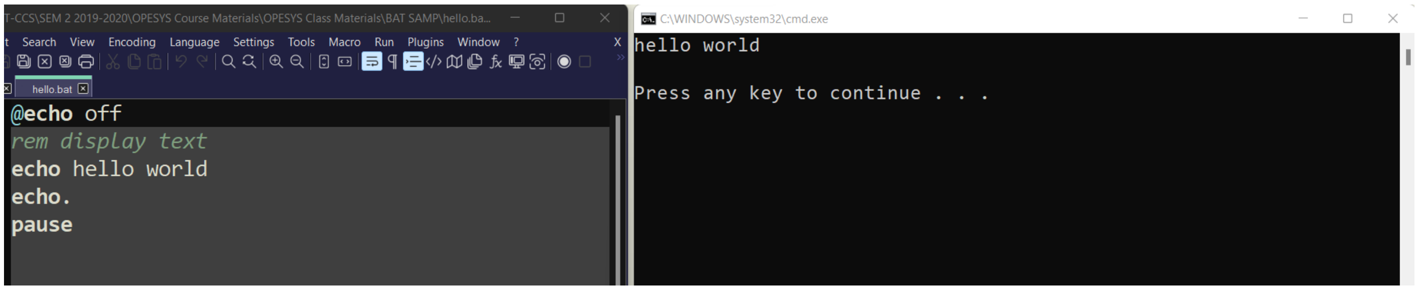
SAMPLE SCRIPTS with OUTPUT

To execute this script, just double-clicked the batch file.



```
@echo off
rem This batch file will open an application.

start notepad++.exe
start calc.exe
```

To execute this script, just double-clicked the batch file.

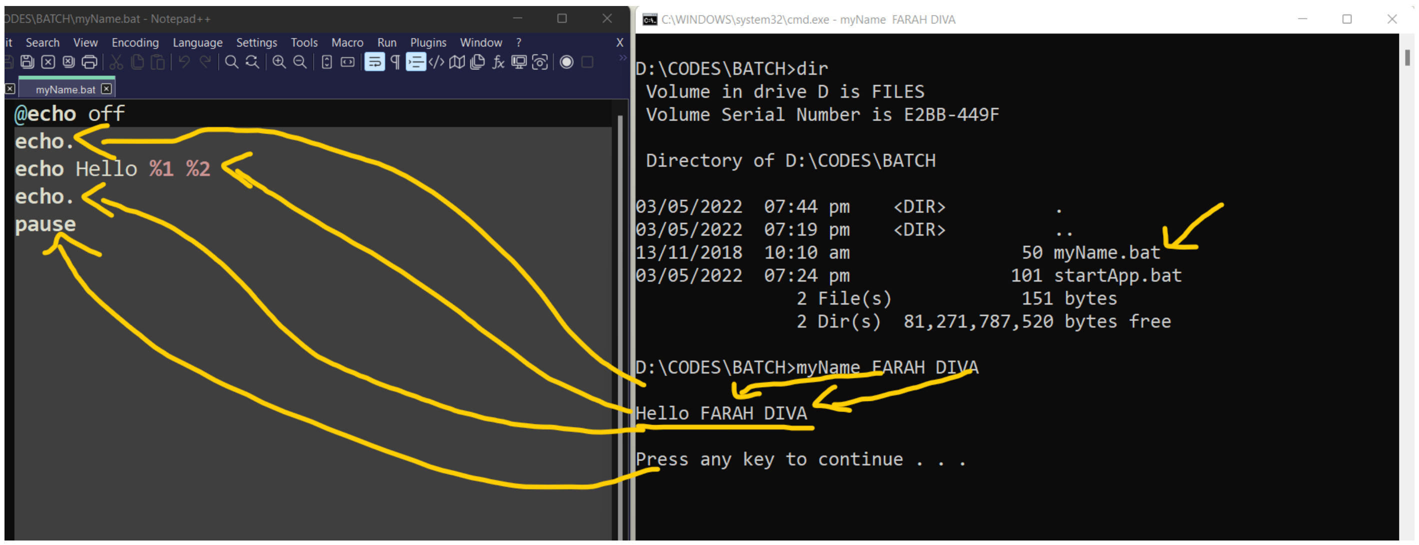


```
@echo off
rem display text
echo hello world
echo.
pause
```

```
hello world

Press any key to continue . . .
```

T-CCS\SEM 2 2019-2020\OPESYS Course Materials\OPESYS Class Materials\BAT SAMP\hello.ba...
hello.bat
C:\WINDOWS\system32\cmd.exe

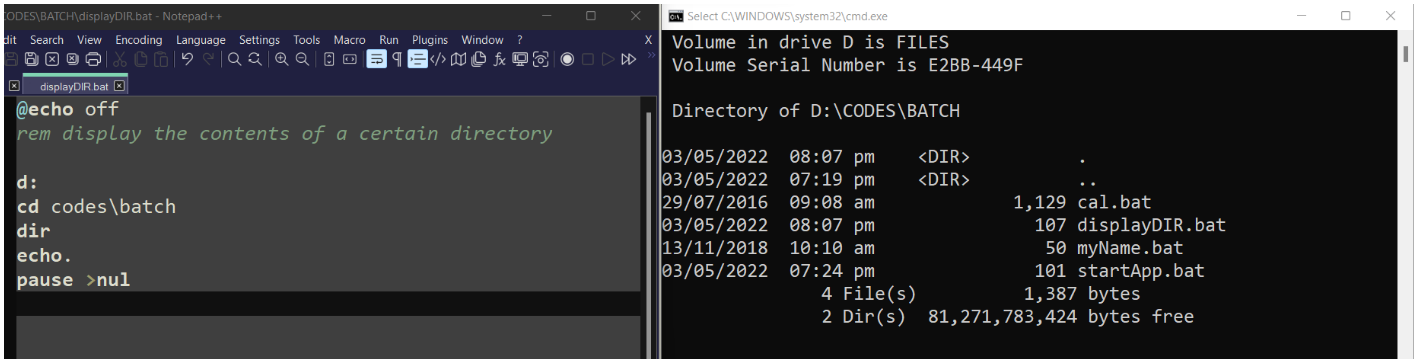
This batch file has to run on the cmd window because there is a need to input values for the generic variables on the scripts.


ODES\BATCH\myName.bat - Notepad++
myName.bat
C:\WINDOWS\system32\cmd.exe - myName FARAH DIVA

```
@echo off
echo.
echo Hello %1 %2
echo.
pause
```

```
D:\CODES\BATCH>dir
 Volume in drive D is FILES
 Volume Serial Number is E2BB-449F

 Directory of D:\CODES\BATCH

03/05/2022  07:44 pm    <DIR>          .
03/05/2022  07:19 pm    <DIR>          ..
13/11/2018  10:10 am                50 myName.bat
03/05/2022  07:24 pm               101 startApp.bat
               2 File(s)            151 bytes
               2 Dir(s)  81,271,787,520 bytes free

D:\CODES\BATCH>myName FARAH DIVA

Hello FARAH DIVA

Press any key to continue . . .
```

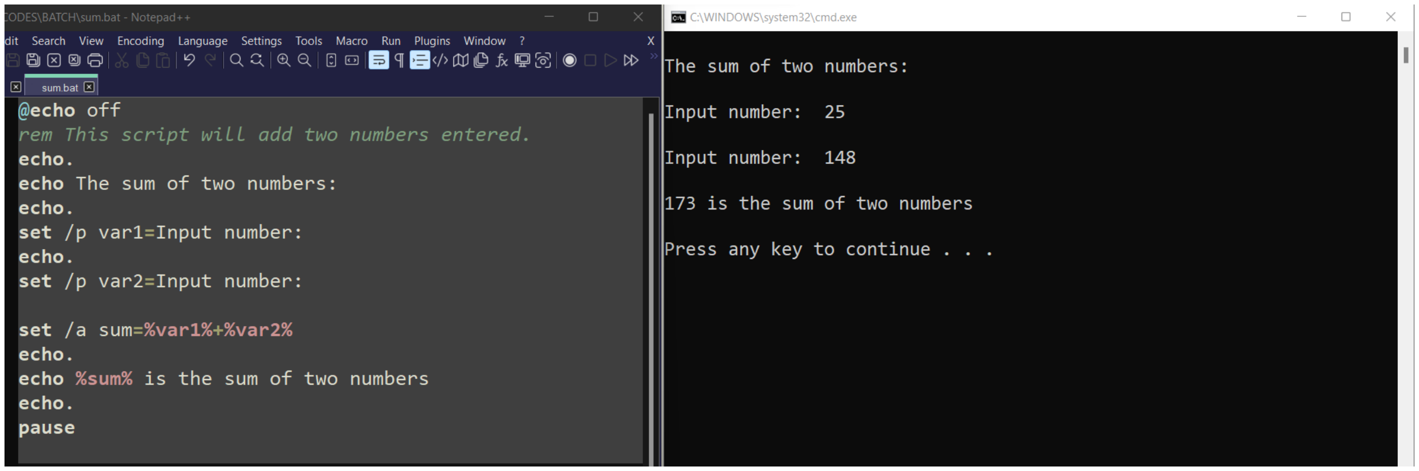
This batch file shows that you can make use of Windows Internal commands combine with the batch commands.


ODES\BATCH\displayDIR.bat - Notepad++
displayDIR.bat
Select C:\WINDOWS\system32\cmd.exe

```
@echo off
rem display the contents of a certain directory

d:
cd codes\batch
dir
echo.
pause >nul
```

```
 Volume in drive D is FILES
 Volume Serial Number is E2BB-449F

 Directory of D:\CODES\BATCH

03/05/2022  08:07 pm    <DIR>          .
03/05/2022  07:19 pm    <DIR>          ..
29/07/2016  09:08 am             1,129 cal.bat
03/05/2022  08:07 pm               107 displayDIR.bat
13/11/2018  10:10 am                50 myName.bat
03/05/2022  07:24 pm               101 startApp.bat
               4 File(s)          1,387 bytes
               2 Dir(s)  81,271,783,424 bytes free
```

This batch file computes the sum of two numbers inputted by the user.



```
@echo off
rem This script will add two numbers entered.
echo.
echo The sum of two numbers:
echo.
set /p var1=Input number:
echo.
set /p var2=Input number:

set /a sum=%var1%+%var2%
echo.
echo %sum% is the sum of two numbers
echo.
pause
```

```
The sum of two numbers:

Input number:  25

Input number:  148

173 is the sum of two numbers

Press any key to continue . . .
```

Hi I'm Flashee!

You have reached the end of the lesson. Be sure to answer the corresponding **activity of this lesson** on the activities folder of our class materials in the file server.

COURSE OUTLINE

Course Code:	**OPESYS**	Credit Units:	**3.0**
Course Name:	**OPERATING SYSTEMS**	Pre-Requisites:	**4th Year Status**

Course Description:

OPESYS deals with the key concepts and the general principles to the study of operating systems. This course focuses on implementation of various operating systems from their structures, modules, components and development. Students will learn to manipulate the special features of an operating system, configure the OS settings, create batch files and apply maintenance techniques in an operating system. The course consists of lectures and learning activities. Upon completion of this course, the students are expected to create their own batch files, manipulate and maintain an operating system.

Learning Outcomes:

- Compare and contrast the characteristics and features of various operating systems suited to solve complex problems through evaluation various case studies;
- Implement and simulate basic modules or components of an operating system;
- Participate in developing real-world systems that will suite to the needs of the organization.

Term	Main Topic	Subtopics
PRELIM	• Introduction to Operating System	• What Operating Systems Do • Computer System Organization
PREMIDTERM	• Operating System Structures	• Operating System Services • User Operating System Interface
MIDTERM	• Command Interpreters	• Command Line Interface
PREFINAL	• Batch Commands	• Batch Commands Implementation
FINAL	• Storage Management	• Storage Device Management