# CHAPTER I
# INTRODUCTION

## 1.1 DEFINITION

- ***What is an operating system?***
  - "What stands between the user and the bare machine."
  - More formally, the operating system is the *basic* software required to operate a computer.
  - It plays a similar role to that of the conductor of an orchestra.
- ***What does not belong to an operating system?***
  - All user programs,
  - Programs such as compilers, spreadsheets, word processors, and so forth,
  - Most utility programs (**`mkdir`** is just a user program making the **`mkdir()`** system call),
  - Even the ***command language interpreter*** is not really a part of the operating system: any user can write his or her version of the UNIX shell.
- We will focus on the most essential part of the operating system, namely the part that stays in main memory and controls the execution of all other programs. This part is known as the ***kernel***. It is sometimes called the *monitor*, the *supervisor* or the *executive*.

## 1.2 FUNCTIONS OF AN OPERATING SYSTEM

We can identify three major functions:

### A. To provide a better user interface:

- Accessing directly the hardware would be very cumbersome: we would need to enter manually the code required to read into main memory each

program we want to execute (***boot strapping***).

- Operating systems have made computers easier to use:

  a) ***Batch systems*** allow their users to submit a batches of requests to be processed in sequence; these batch systems include a command language that can be used to specify what to do with the inputs (compile, link edit, execute, attach this file, and so forth)

  b) Systems such as UNIX, MS-DOS and VMS allow users to interact with the OS through their terminals: they have an ***interactive*** command language (UNIX shell, Windows PowerShell) that can also be used to write scripts.

     When the OS allows several interactive users to access the computer at the same time, it is called ***time-sharing***.

  c) More recent systems have ***graphical user interfaces*** (GUIs pronounced *goo-eys*): Macintosh, Windows, X-Windows, Linux.

     These graphical user interfaces were pioneered at XEROX Palo Alto Research Center and popularized by the Macintosh.

     Another area where the OS is providing a better user interface is the ***file system***. Users can create and delete files on the system disks without having to worry about disk allocation: what they are losing is the ability to specify exactly how files are stored on the disk. Some more recent file systems tolerate disk failures (RAID)

## B. To manage the system resources:

- Utilizing a computer efficiently is not an easy task because there is an enormous gap between CPU speeds and disk access times:

- The Memory Hierarchy:

| Level | Device | Access Time |
|---|---|---|
| 1 | Fastest registers (2 GHz CPU) | **0.5 ns** |
| 2 | Main memory (133 MHz average) | **10-70 ns** |
| 3 | Secondary storage (disk) | **7 ms** |
| 4 | Mass storage (CD-ROM library) | **a few s** |

Assume now the same access time ratios at a *human scale*:

**Working on a Term Paper**

| Level | Resource | Access Time |
|---|---|---|
| 1 | Open book on desk | **1 s** |
| 2 | Book on desk | **20-140 s** |
| 3 | Book in library | **162 days** |
| 4 | Book far away | **63 years.** |

The gap will ***widen*****,** as long as processor speeds keep increasing. The OS can use several techniques to handle this huge access time disparity:

(i) The OS can try to maximize the number of bytes exchanged between the main memory and the disk drive during each disk access. It works but:

- If we try to *read too much ahead* of the program, we risk to bring into the main memory data that will never be used;
- If we *delay* writes to write more data in a single I/O operation, we can lose data if the system or the program crashes after the program issued a write but before the data were written to disk.

(ii) The OS can keep in a buffer recently accessed data hoping they will be accessed again. This is known as caching. Caching works very well because we keep accessing again and again the data we are working with. Caching is a fundamental technique of OS and database design

(iii) The OS can implement *multiprogramming*: this allows the CPU to run program while a program waits for an I/O.

This is very important in business applications because many of

these applications use the peripherals much more than the CPU (think about all these old COBOL programs printing monthly bills and paychecks): since the processor was the expensive part of the computer. This is why ***multiprogramming*** was invented.

Multiprogramming also made ***time-sharing*** possible. It remains very important today for all personal computers.

***Multiprogramming:***

- With multiprogramming, a computer lets its CPU divide its time among different tasks: the CPU works for, say, one tenth of a second on a given program, then for another tenth of a second on another one and so forth. Note that a single-core CPU is only working on *one single task* at any given time the.

- The major direct benefit of multiprogramming is that the CPU does not waste any time waiting for the completion of an I/O because it can use the free time to work on another task.

- From time to time, the OS will need to regain control of the CPU either because a task has exhausted its fair share of the CPU time or because something else needs to be done.

- This is done through ***interrupts*****.** An interrupt is a request to interrupt the flow of execution the CPU. It is detected by the CPU hardware after it has executed the current instruction and before it starts the next instruction.

- When an interrupt occurs:

  a) The ***current state of the CPU*** (program counter, program status word, contents of registers, and so forth) is saved, normally on the top of a stack, and

  b) A ***new CPU state*** is fetched.

  This new state includes a new *hardware-defined* value for the program counter. As a result, it is impossible to “hijack” an interrupt.

  The process is totally transparent to the task being interrupted because the OS will always restore the exact state of the CPU before restarting it.

- Interrupts play a central role in modern computer systems:

  a) ***I/O completion interrupts*** notify the OS that an I/O operation has completed,

  b) ***Timer interrupts*** notify the OS that a task has exceeded its quantum of CPU time,

  c) ***Traps*** notify the OS of a *program error* (division by zero, illegal op code, illegal operand address, ...) or a *hardware failure*,

  d) ***System calls*** notify OS that the running task wants to submit a request to the OS.

  *In UNIX, as in many other systems, system calls have the same syntax as function calls: the body of the function being called contains the assembly code preparing the arguments and making the interrupt request.*

- Each interrupt will result into ***two context switches***: one when the running task is interrupted and another when it regains the CPU.

- It might happen that an interrupt request would occur while the system is processing another interrupt. We need to decide if the new request should be allowed to interrupt the current interrupt or should have to wait its turn. This decision is complicated by the fact that all interrupt requests are not equally urgent (as it is also in real life: we can be interrupted by the phone ringing, somebody ringing the doorbell or a cloud of dark smoke coming out of the kitchen!).

  The best solution is to *prioritize* interrupts and to assign to each source of interrupts a ***priority level***. New interrupt requests will be allowed to interrupt lower-priority interrupts but will have to wait for the completion of all pending interrupts with equal or higher priorities. This solution is known as ***vectorized interrupts***.

- It is possible to ***disable*** interrupts and the OS uses this feature while performing short critical tasks that cannot be interrupted. User tasks should be prevented from doing it due to the many risks involved.

- Disk I/O poses a special problem, as the CPU might have to transfer large quantities of data between the disk controller's buffer and the main memory. All modern disk controllers have ***direct memory access*** (or DMA), which allows them to read data from and write data to main memory without any CPU intervention.

## C. To protect users' programs and data:

- Unless the same person always uses the system, we need to prevent its users from accessing, deleting or modifying other people's programs and data. Protection is essential for:

  a) Conventional ***batch*** and ***time-sharing*** systems,

  b) Any computer shared by several users,

  c) Any computer connected to the Internet, especially when remote access is authorized.

- Earlier operating systems for personal computers did not have any protection since they were single-user machines. Windows 2000, Windows XP, Vista and MacOS X are protected.

- The essential resource to protect are user and system ***files***, we must:

  **a)** *Prevent users programs from* ***accessing directly the disk:***

  To achieve this purpose, we need a ***dual-mode CPU***. This kind of CPU will divide instructions into ***two classes***:

  - ***Regular instructions***, which can be safely executed by any process,
  - ***Privileged instructions***, which include ***all physical I/O instructions*** and can be executed only when the CPU is in ***privileged*** or ***supervisor mode***.
  - The process state must therefore include a ***special bit*** stating whether it is in supervisor mode or in user mode.

    User mode will be the default mode for all programs: only the kernel will be allowed to run in supervisor mode.

The only way to switch from user mode to supervisor mode is through an interrupt: An interrupt will always initiate trusted code since the jump address is at a well-defined location in main memory.

**b)** *Prevent user programs from modifying the memory locations containing the OS code:*

We also need special ***memory protection*** hardware checking all memory references from user programs and generating an interrupt every time a user program tries to reference something outside its address space.

One added benefit is that user programs cannot cause other user programs to crash by corrupting their code or their data. Users can safely work on their term paper while running a programming assignment in the background.

***Notes:***

- A system that does not have a dual-mode CPU ***and*** memory protection ***cannot*** be made secure.

- Any decent multiprogramming system must have memory protection even it is a single user system; otherwise one program can cause another to crash.

- To make a system secure, one must also prevent people to reboot the system with a doctored version of the OS.

- If you run Windows XP anyone that can boot your PC with a Linux live CD/DVD can read your NTFS and FAT files unless they are ***encrypted***.

## 1.3 TYPES OF OPERATING SYSTEMS

- We have already discussed ***batch*** systems and ***time-sharing*** systems

## A. Real-Time Operating Systems:

- These systems are designed for applications with strict real-time constraints such as ***process control***, ***guidance systems***, most ***multimedia applications*** and so forth. The key issue is how to guarantee that critical tasks will ***always*** be performed within a specific time frame.

- One can distinguish between:

  **a)** ***Hard real-time systems*** that must guarantee that all deadlines will always be met since any failure could have catastrophic consequences: the reactor could overheat and explode; the rocket could be lost and so on;

  **b)** ***Soft real-time systems*** that guarantee that most deadlines will be met: a DVD decoder that miss a deadline will only spoil our viewing pleasure for a fraction of a second.

- Interactive systems, like Windows or Unix, are not true real time systems: they will respond very quickly to *most* user requests but assume that their users can tolerate some infrequent unexpected delays.

## B. Multiprocessor Operating Systems:

- These systems are designed for multiprocessor architectures, that is, architectures where several processors share the same memory.

- Two major approaches are possible:

  (a) ***master-slave*** system: all system functions are performed by *one* processor; the other processors can only execute user programs; hence the possibility of bottlenecks, and

  (b) ***symmetric*** system: any processor can perform all functions; there can be multiple copies of the OS running in parallel and we must prevent them from interfering with each other.

## C. Distributed Systems:

- These systems consist of integrated networks of computers, such as

***workstations*** sharing common resources (file servers, printers, and so forth)

- Current trend is to leave these systems very loosely coupled: each computer has its own operating system.

- Most distributed systems use the ***client-server model*:** file servers, authentication servers, name servers, and so forth are implemented by processes that wait for requests from client processes and processing them.

- X-window systems are a special case: the X-window server manages windows on a workstation screen for a client program running on the workstation or on a remote machine.

- A shared ***distributed file system*** (DFS) or ***network file system*** (NFS) is an essential part of a distributed system. To reduce the load on the file servers, most DFS allow *client machines* to ***cache*** the portions of the files they are currently using. The key issue is what to do when the same file is cached on different workstations. Updates could be lost and different clients could have inconsistent views of the same file. Each DFS offers its own solution to that problem.

- Other important issues in distributed systems include:

  (a) *Authenticating* users (passwords do not work very well),

  (b) Making distributed systems as *reliable* as stand-alone systems.

  (c) Keeping the clocks of the machines synchronized.

  These topics fail outside the scope on an introductory operating system course.

## 1.4 UNIX

- UNIX started at Bell Labs in the 70's as an attempt to build a sophisticated time-sharing system on a minicomputer.

- UNIX is almost entirely written in C; it was a first.
- It was ported to the VAX architecture in the late seventies at U. C. Berkeley: they added virtual memory, networking (TCP/IP), not to mention many bells and whistles
- It was for a long time the ***de facto standard*** OS for programming and engineering ***workstations***
- Several ***free versions*** exist (FreeBSD, Linux): Since the source code of these free versions is available at no cost, Unix internals are much better known than those of any other operating systems.
- UNIX was the first ***portable*** OS: it has run and still runs on many different hardware platforms.
- It encompasses three major traditions, System V/Solaris, BSD and Linux: no single standard has ever imposed itself.
- It includes *very powerful tools* to manipulate *text files* but lags somewhat behind Macintosh and Windows when GUIs are concerned.

## A Rapid Tour:

- The UNIX kernel is the core of the system and handles the system calls.
- UNIX has several command language interpreters or shells: **`sh, csh`**, **`ksh`**, **`bash`**.
- All commands are documented in the on-line manual:
  - **`man xyz`** displays the manual page for the command **`xyz`**,
  - **`man 2 xyz`** displays the manual page for the *system call* **`xyz()`**,
- Unix has many other utilities (**`passwd`**, **`mail`**, ...), compilers (**`cc`**, **`gcc`**, ...) and text editors (**`vi`**, **`emacs`**).

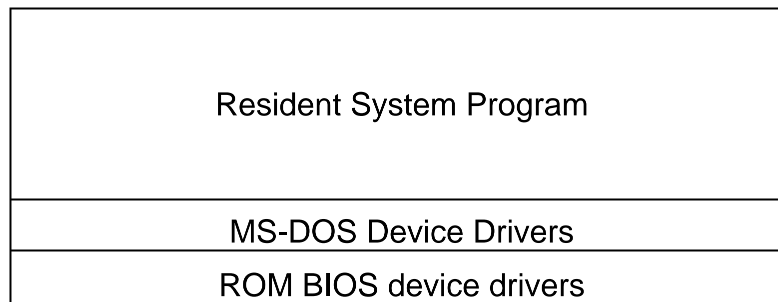
## Most Lasting Impacts:

- UNIX was the *first operating system* that could run efficiently on very different platforms and had its source code made available to its users.
- Its file system was imitated by most of its followers (including MS-DOS and Windows).
- Its system call interface has become an *OS-independent standard* (POSIX).
- Most current OS research performed outside Microsoft Research is done on Linux platforms

## 1.5 O.S. STRUCTURES

### A. Monolithic Kernels:

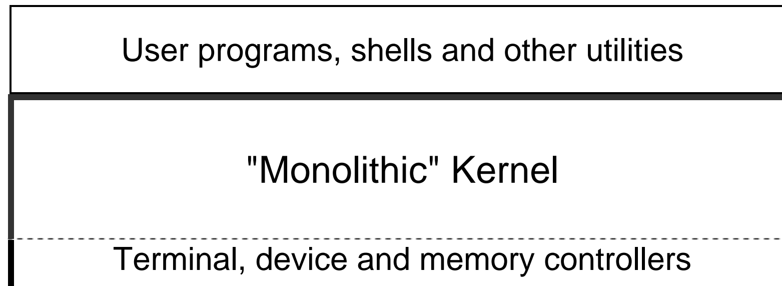
***MS-DOS:***




- BIOS (for Basic Input-Output System) is the program that takes control of the CPU when the system is turned on. It identifies system components (hard disk, DVD drive, …) and initiates the booting of the "real" operating system.
- MS-DOS was developed for a microprocessor lacking dual mode and hardware memory protection: nothing prevents an application program from accessing directly the BIOS.
- Software compatibility has prevented for a long time any significant change in the file system internals: the move from FAT-16 to FAT-32 was the first significant change to the file system internals. (Modern Windows systems use NTFS for their disk file systems and FAT file systems for their flash drives.)

***UNIX:***

- The UNIX kernel is accessible through the ***system call interface*** and the user has to go through the kernel to access the resources managed by the kernel.




- It contains everything that is ***not device-specific*** including the file system, networking code, and so forth.
- The UNIX kernel said to be ***monolithic*** because all the kernel functions share the same address space. This makes the kernel hard to manage, extend and debug.
- Several solutions to the problem have been proposed, among which layered kernels (a *flop*) and microkernels.

## B. Layered Kernel:

- A layered kernel is implemented as a hierarchy of **layers**:
- Each layer defines a new data object hiding from the higher layers some functions of the lower layers and providing in exchange some new functionality.

**Example:** The THE Operating System (E. Dijkstra, T. U. Eindhoven) had six layers (including the hardware and the users programs):

| User Programs |
| --- |
| Buffering for I/O Devices |
| Operator Console Device Driver |
| Memory Management |
| CPU Scheduling |
| Hardware |

- Unfortunately, it is hard to decide which layers should go on the top of which layer. For instance, there are arguments to put the memory management layer:

  (a) On the top of the file system layer (to facilitate swapping) or

  (b) Below it (to allow file system to use memory management).

## C. Microkernels:

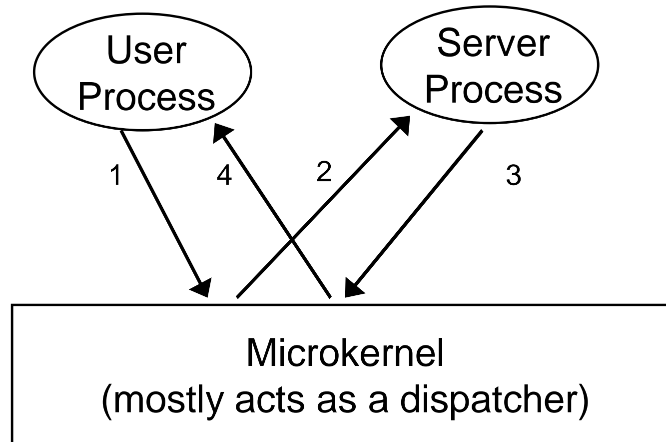
- Microkernels delegate as many tasks as possible to ***servers*** outside the kernel: file servers, networking servers, and so forth

- As a result, the kernel becomes smaller, more manageable and easier to debug. Other advantages include:

  - We can add new services, such a new file system, and modify the existing server processes without touching to the kernel:

  - We can emulate multiple operating systems on the top of a given kernel:

    *One of the versions of the Mach kernel was running a UNIX-emulation server outside the kernel space.*




- Microkernels have one ***major drawback***: System calls involving a server process require ***four*** context switches instead of ***two***:
    1. User process switches context to kernel,
    2. Kernel analyzes request and switches context to the server process,
    3. Server process returns context to kernel
    4. Kernel returns context to user program
- Microkernels would be the ideal solution *if context switches could be made less expensive*:
- MacOS X is Mach-based but has the UNIX emulation inside the kernel.

- **Example: Windows NT:**

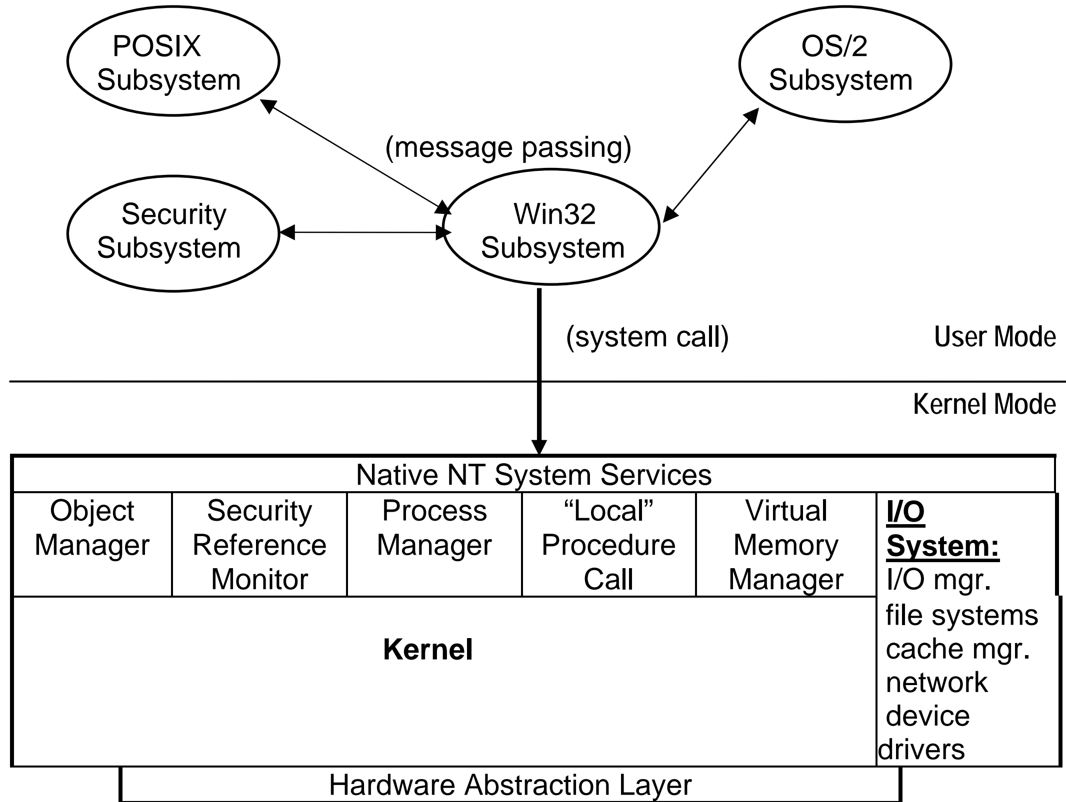
– Windows NT was designed by Microsoft to be "an operating system for the 1990s."

– Its major objectives were: ***extensibility***, ***portability*** to non-INTEL architectures, ***reliability***, ***compatibility*** with user interfaces such as MS-DOS, OS/2 or POSIX (but only in ASCII mode), and ***performance*.**

– It supports symmetric multiprocessing, includes networking primitives and can provide class C-2 level security (that is, the owner of each resource can specify which users can access it).

– It includes a *layered executive* and *user-level servers* called *subsystems* (four in version 3.5, three in 4.0):

– The servers allow the support of multiple application programming interfaces (API's); the *security subsystem* is used at log-on time.




- *Performance reasons have led to the* ***reintegration*** *of the Win32 subsystem* ***into the kernel*** *in version 4.0.*

- MS-DOS applications and 16-bit window code run within the context of separate *virtual MS-DOS machines* (VDM) that are directly supported by the Win32 subsystem.

- The *object manager* creates manages and deletes the abstract data types used to represent NT resources (*executive objects*).

- The *security reference monitor* performs run-time object protection and auditing.

- The *local procedure call facility* implements a local version of *remote procedure calls* (RPC, see section 4.3)

- The *virtual memory manager* allocates and deallocates the main memory (see chapter 8).

- The *kernel* handles interrupts and exceptions, schedules the processors, synchronizes their activities and provides services for the rest of the NT executive.

- The *I/O system* includes:

  1. The *I/O manager*, which implements device independent I/O facilities;
  2. The *file system,* which translates *file-oriented I/O requests* into I/O requests;
  3. The *network drivers*;

4. The NT executive *device drivers*, which directly manipulate the I/O device;
5. The *I/O cache manager*.

– The *hardware abstraction layer* consists of a set of routines that hide hardware-dependent details from the NT executive.

## D. Modular Kernels:

- Linux achieves the same level of flexibility as microkernels through ***modules***.
- A ***module*** is an object file whose contents can be linked to—and unlinked from—the kernel at any time while the system is running.

  *The most common use of modules is to add to the kernel **device drivers** for new devices.*

- Unlike user-level servers, modules do not run as separate processes but are executed inside the kernel just as any other kernel function,
- The main advantages of modular kernels are:
  1. ***Extensibility:*** new features can be added to the kernel without having to modify the kernel source code; in many cases, the process is completely transparent to the user since linking and unlinking can be performed automatically by the kernel.
  2. ***Lack of performance penalty:*** since modules are executed in the kernel address space, calling a module function is as fast as calling any other kernel function.
- The main disadvantage of modular kernels is their lower reliability: a poorly written module can corrupt the whole kernel and crash the system.
- The problem is compounded by the fact that most device driver writers are not as experienced as other kernel writers. In Windows XP, device drivers account for 85% of recently reported failures.
- To learn more, you might want to read:

  M. M. Swift, B. N. Bershad, H. M. Levy, "Improving the reliability of commodity operating systems," *Proc. 19th ACM Symp. on Operating Systems Principles*, Oct. 2003, pp. 207-222.

  (in the COSC 6360 reading list)

## 1.6 KEY CONCEPTS

Be sure that you understand and can explain the following concepts:

*batch system*
*client/server model*
*context switch*
*distributed file system*
*distributed operating system*
*dual mode CPU*
*graphical user interface*
*hard deadline*
*interrupt*
*kernel*
*layered kernel*
*master-slave multiprocessing*
*memory protection*
*microkernel*
*monolithic kernel*
*multiprocessor operating system*
*multiprogramming*
*on-line manual*
*portable operating system*
*privileged instruction*
*real-time operating system*
*server processes*
*supervisor mode*
*symmetric multiprocessing*
*system call*
*timer interrupt*
*time-sharing*
*trap*
*user mode*
*vectorized interrupts*