Chapter 2

- **Operating System Services:**

➢ User interface (UI):

- most commonly:
  - graphical user interface (GUI)
  - command line interface (CLI)

➢ Program execution:

- OS loads programs into memory and runs them

يعني نظام التشغيل يقدر يشغل البرامج في الرام

- a program must be able to end its execution, either normally or abnormally

➢ I/O operations:

▪ user processes cannot execute I/O operations directly ▪ OS must provide processes with some means to perform I/O ➢ File-system manipulation:

▪ OS should allow users to create, read, write, and delete files ➢ Communications:

- OS enables user processes to exchange information, whether on the same computer or on different systems connected by a network
- implemented via shared memory or message passing ➢

Error detection:

i*j

- OS ensures correct computing by: detecting errors in the CPU, memory hardware, I/O devices, or user programs

- **Operating system functions:**
  - ➢ <u>Resource allocation:</u>

- OS allocates resources to the running processes

➢ <u>Logging:</u>

- Recording which process uses how much and what kinds of resources
- This record may be used for account billing or accumulating usage statistics

➢ <u>Protection and security:</u>

- Protection ensures that all access to system resources is controlled
- The security of the system from outsiders is also important

- **System calls:**
  - ➢ provide an interface to the services provided by the OS, allowing user processes to call them
  - ➢ generally available as C / C++ functions or assembly-language instructions
  - ➢ when a process executes a system call, the system traps to the OS
  - ➢ <u>There are three methods for passing parameters to system calls:</u>
    1. Pass the parameter in CPU registers

i*j

2. Store the parameters in a table in memory (block), and pass the address of the block as a parameter in CPU registers
3. Push the parameters onto the stack by the process and pop them later by the OS

➢ The OS provides application programmers with API to invoke services

➢ <u>Types of system calls:</u>

▪ Process control:

- Create, load, execute, terminate, and abort
- Get process attributes and set process attributes
- Wait for time, wait event, signal even, allocate and free memory ▪ File management:
- Create file and delete file
- Open, close, read, write, reposition file • Get file attributes and set file attributes

▪ Device management:

- Request device, release device, read, write, reposition • Get and set device attributes, logically attach or detach devices ▪ Information maintenance:
- Get time/date, set time/date
- Get system data, set system data
- Get process, file, or device attributes, set process, file, or device attributes

▪ Communications:

i*j

- Create, delete communication connection
- Send message, receive message, transfer status information
- Attach remote device, detach remote device

- **System boot:**

➢ booting the system is the process of starting the computer by loading the kernel, it proceeds as follows

1. The bootstrap program locates the kernel
2. The kernel is loaded into memory and started
3. The kernel initializes hardware 4. The root file system is mounted

- **Operating system structure:**

➢ **<u>Monolothic approach:</u>**

- no structure at all
- all of the functionality of the kernel is placed into a single static binary file that runs in a single address space
- tightly coupled system; changes to one part of the system can have effects on other parts
- difficult to implement and extend
- have a performance advantage as:
  - there is very little overhead in the system call interface
  - communication within the kernel is fast
- Several OSs use of this approach, such as UNIX, Linux, and Windows

➢ **<u>Layered approach:</u>**

- divides the OS into a number of layers

i*j

▪ easier to implement and extend

▪ the highest layer: user interface, the bottom layer: the hardware

▪ each layer uses the functions of only the lower-level layers

▪ the kernel components are loosely-coupled, therefore, changes in one component does not affect the others

▪ difficult to define the functionality of each layer

▪ the overall performance of the OS is poor

➢ **Micro-Kernel approach:**

▪ removes all nonessential components from the kernel and implements them as system programs ▪ It is easier to extend the OS as:

- No need to modify of the kernel
- New services are added as system programs

▪ It is easier to modify kernel if needed as it is a smaller kernel ▪ It is easier to port the OS from one hardware design to another ▪ It provides more security and reliability as:

- Most services are running as user rather than kernel processes
- If a service fails, the rest of the operating system remains untouched

➢ **Modules approach:**

▪ The kernel has a set of core components and can link in additional services via Loadable Kernel Modules (LKMs), either at boot time or during run time

▪ Linking additional services dynamically is preferable to recompiling the kernel every time a change is made

i*j

▪ This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows

▪ **Compared to:**

- <u>Layered approach:</u>
  - ◦ Similarity: each kernel section has a defined and protected interface
  - ◦ Difference: more flexible, because any module can call any other module
- <u>Microkernel approach:</u>
  - ◦ Similarity: the primary module has only core functions
  - ◦ Difference: modules do not need to use message passing to communicate

➢ **<u>Hybrid approach:</u>**

▪ very few OSs adopt a single, strictly defined structure

▪ Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues

▪ For example, Linux is both monolithic and modular:

- Monolithic: to provide very efficient performance
- Modular: to add new functionality dynamically to the kernel