chapter 2

services: Predefined comfort routine → user interface → CLI

└→ GUI

└→ batch (shell script) executable files

└→ Program execution

└→ I/O operations

└→ File system

└→ Communication

└→ error detection

Program execution → load Programs into memory

└→ run that Program

└→ stop it (normaly or abnormaly (error))

I/O → user can not usually control I/O devices directly

└→ a running Program may require I/O

file system → read and write files and dictionaries

└→ create and delete files

└→ Permission management

Communication → information exchange between Processes

└→ ① shared memory

└→ ② message Passing

└→ Packets of info in Predefined format

└→ they can be on different computers

error detection → detect and correct errors

└→ in CPU, memory hardware, I/O devices, user Program

└→ Halt the system or Terminate the Program or Pass error code

services for efficient operation → resource allocation

└→ accounting

└→ Protection and security

accounting → usage statics

└→ billing users based on how much and what computer sources they use security → Control access to system resources

↳ user authentication

↳ ...

Command interpreters:

kernel Based or System Program → windows or UNIX

↳ running when a job is initiated
or
when the user logs on

multiple command interpreter → shell

get and execute user-specified ~~Program~~ command

↳ ① internal codes

↳ ② system Program → rm file.txt

System calls → Provide an interface to O.S. services

↳ C, C++, assembly

↳ API → Application Programming Interface

↳ a set of ~~fun~~ available function, & their Parameters and return values

↳ Java, POSIX, windows

↳ accessed via a library of code Provided by the O.S.

↳ Portability, actual system calls are difficult.

↳ system call interface: link to system calls

numbers system calls, Puts them in a table and invokes them

↳ Parameter passing → registers

↳ register Pointing to memory block (stack)

* Communications Mechanism → message Passing

↳ shared memory

types of system calls :

- Process Control
  - end, abort
  - load, execute
  - create, terminate
  - get and set attributes
  - wait for time, event (signal event)
  - allocate and free memory
- File management
  - creat, delete
  - open, close
  - read, write, reposition
  - get and set file attributes
- Device Manegement
  - request or release device
  - read, write, reposition
  - get and set device attributes
  - logically attach or detach devices
- Information Maintnance
- Communications
- Protections

* O.S design

Goals : user vs System

Mechanism (how) vs Policy (what)

* O.S. structure

* simple structure or monolithic → the most common organization

↳ OS. is a large single Program in kernel mode

kernel = everything after system call interface and before hardware

Problems : crash, difficult to understand

adv : very little overhead in system call interface

example = MS-DOS

Beyond simple but not fully layered = traditional UNIX

traditional UNIX kernel = device drivers + interfaces

* layered approach

bottom layer = hardware   top layer = user interface

layers = ~~data~~ a implementation of data and operations that can manipulate that data | can be invoked by higher layers

↳ each layer hides data and... from higher levels

adv : simplicity of construction and debugging

⇒ in debugging each layer we're only concerned with it's lower layers

disadv : Problem with appropiately defining layers

not efficient : system calls are needed for communication between layers ⇒ overhead

⇒ Fewer layers with more functionality ✓

Scanned by CamScanner

* Microkernel

Moves as much from the kernel to "user" space

=> smaller kernel $\overset{\text{result}}{=>}$ { minimal process and memory management
communication facility (client program and user-space services)
⇓
message passing

interprocess communication
memory management
+ CPU scheduling

---

micro kernel

adv: easier to ~~[illegible]~~ extend (add new services to user space)
easy to port from one hardware to another
more secure (most services are running as user processes)
more reliable (less code is running in kernel)

disadv: overhead due to user space to kernel space communications

* Modules

kernel's core components + additional services via modules
↓
loadable kernel modules

result: we don't have to recompile the kernel after every change

* any module can call any other module => more flexible than layered
↳ no need for message passing => more efficient

* uses object-oriented approach

* ~~a~~ each core component is seperate

* hybrid = combining different structures
↳ better to ~~a~~ address { reliability
security
usability