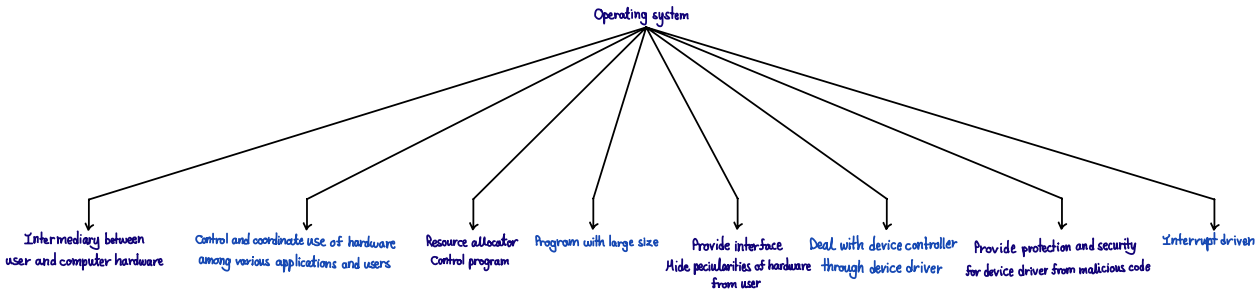
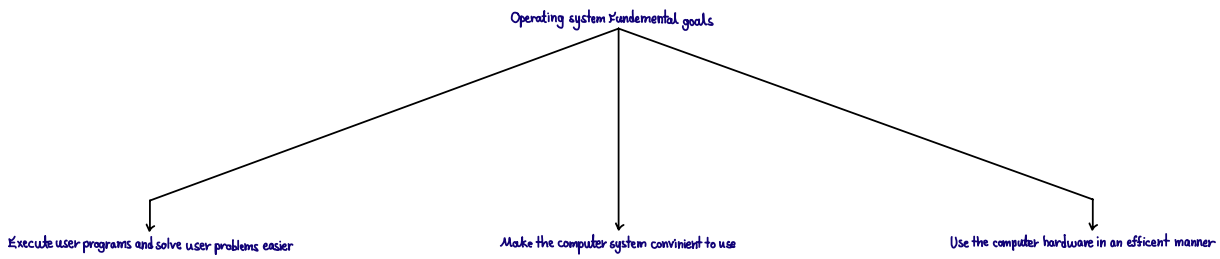
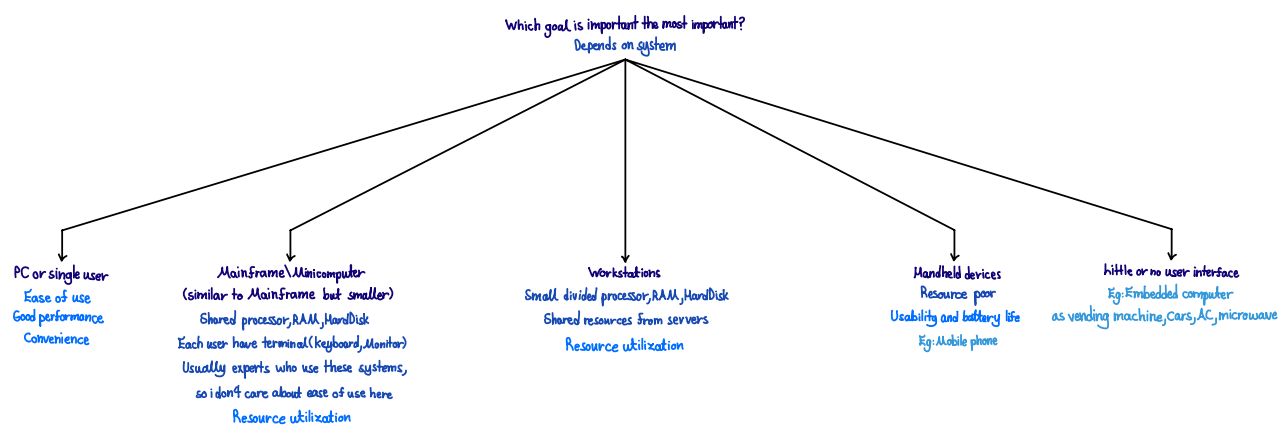
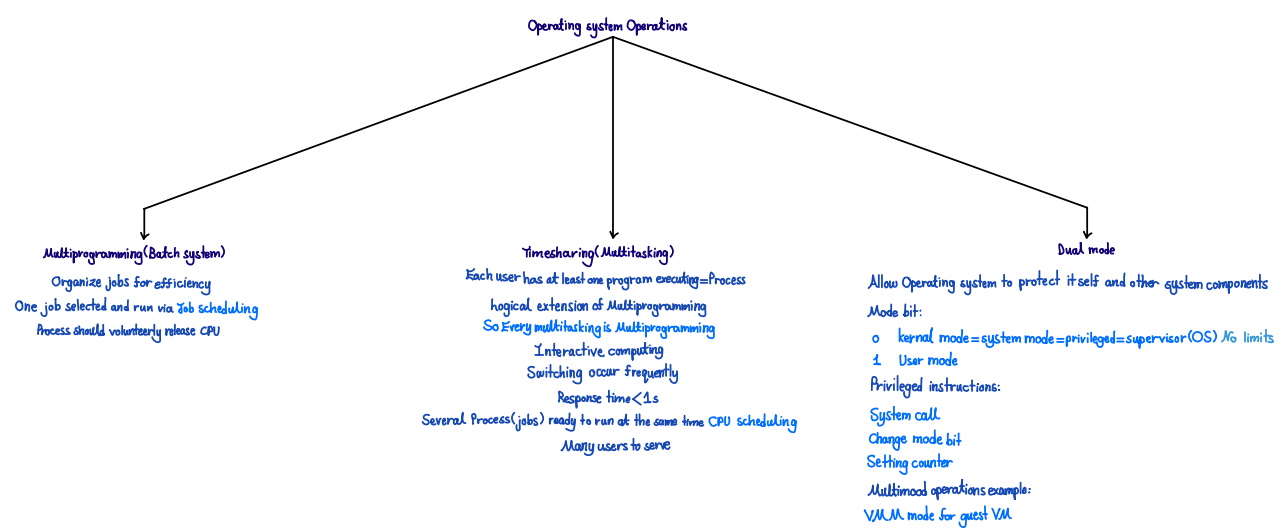
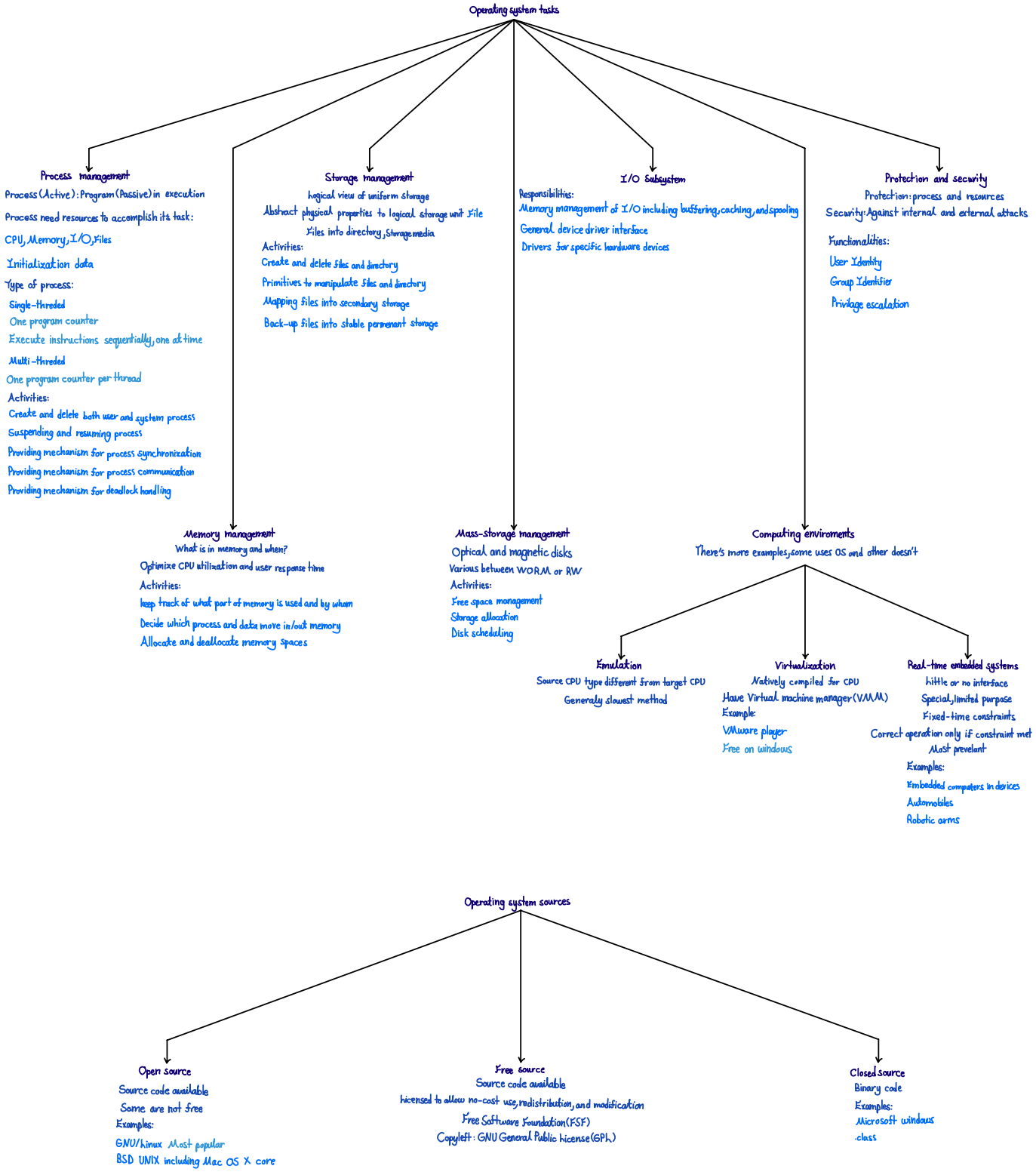

Operating system
Intermediary between user and computer hardware
Control and coordinate use of hardware among various applications and users
Resource allocator
Control program
Program with large size
Provide interface
Hide peculiarities of hardware from user
Deal with device controller through device driver
Provide protection and security
for device driver from malicious code
Interrupt driven

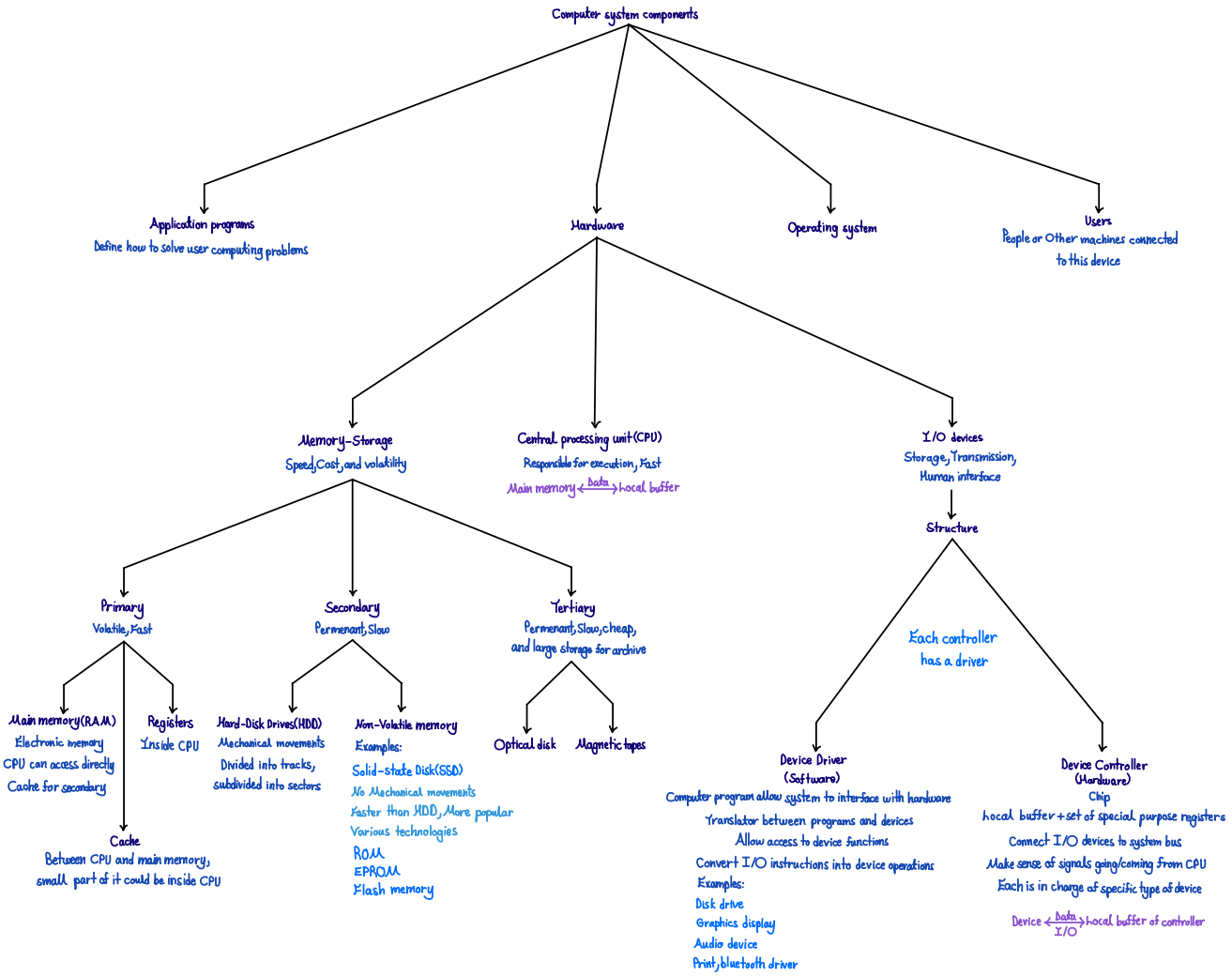

Operating system Fundemental goals
Execute user programs and solve user problems easier
Make the computer system convinient to use
Use the computer hardware in an efficent manner

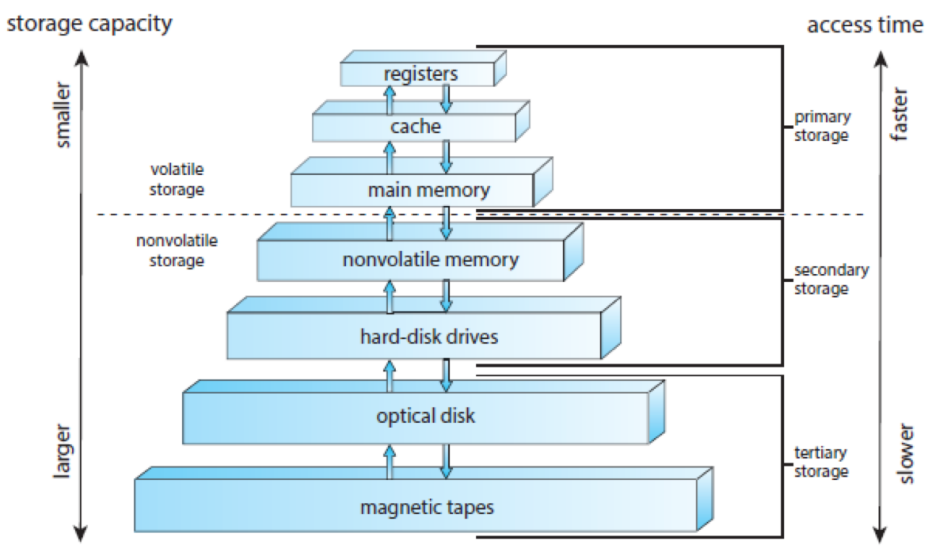

Which goal is important the most important?
Depends on system
PC or single user
Ease of use
Good performance
Convenience
Mainframe\Minicomputer
(similar to Mainframe but smaller)
Shared processor,RAM,HardDisk
Each user have terminal(keyboard,Monitor)
Usually experts who use these systems,
so i don't care about ease of use here
Resource utilization
Workstations
Small divided processor,RAM,HardDisk
Shared resources from servers
Resource utilization
Handheld devices
Resource poor
Usability and battery life
Eg:Mobile phone
little or no user interface
Eg:Embedded computer
as vending machine,Cars,AC,microwave

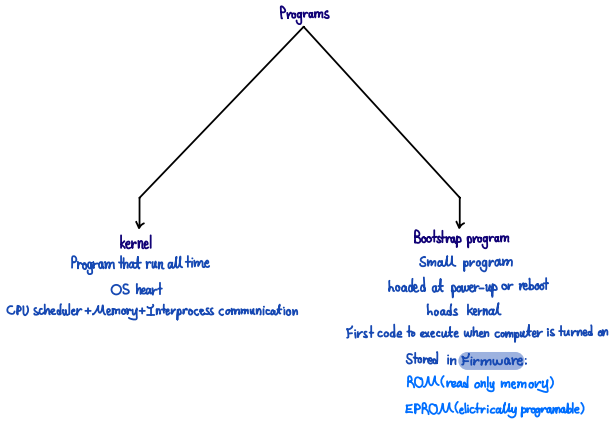

Operating system Operations
Multiprogramming(Batch system)
Organize jobs for efficiency
One job selected and run via Job scheduling
Process should voluntearly release CPU
Timesharing(Multitasking)
Each user has at least one program executing=Process
logical extension of Multiprogramming
So Every multitasking is Multiprogramming
Interactive computing
Switching occur frequently
Response time<1s
Several Process(jobs) ready to run at the same time CPU scheduling
Many users to serve
Dual mode
Allow Operating system to protect itself and other system components
Mode bit:
0 kernal mode=system mode=privileged=supervisor(OS) No limits
1 User mode
Privileged instructions:
System call
Change mode bit
Setting counter
Multimood operations example:
VMM mode for guest VM


Operating system tasks
Process management
Process (Active): Program (Passive) in execution
Process need resources to accomplish its task:
CPU, Memory, I/O, Files
Initialization data
Type of process:
Single-Threaded
One program counter
Execute instructions sequentially, one at time
Multi-Threaded
One program counter per thread
Activities:
Create and delete both user and system process
Suspending and resuming process
Providing mechanism for process synchronization
Providing mechanism for process communication
Providing mechanism for deadlock handling
Storage management
logical view of uniform storage
Abstract physical properties to logical storage unit File
Files into directory, Storage media
Activities:
Create and delete files and directory
Primitives to manipulate files and directory
Mapping files into secondary storage
Back-up files into stable permenent storage
I/O Subsystem
Responsibilities:
Memory management of I/O including buffering, caching, and spooling
General device driver interface
Drivers for specific hardware devices
Protection and security
Protection: process and resources
Security: Against internal and external attacks
Functionalities:
User Identity
Group Identifier
Privilage escalation
Memory management
What is in memory and when?
Optimize CPU utilization and user response time
Activities:
keep track of what part of memory is used and by whom
Decide which process and data move in/out memory
Allocate and deallocate memory spaces
Mass-storage management
Optical and magnetic disks
Various between WORM or RW
Activities:
Free space management
Storage allocation
Disk scheduling
Computing enviroments
There's more examples, some uses OS and other doesn't
Emulation
Source CPU type different from target CPU
Generaly slowest method
Virtualization
Natively compiled for CPU
Have Virtual machine manager (VMM)
Example:
VMware player
Free on windows
Real-time embedded systems
little or no interface
Special, limited purpose
Fixed-time constraints
Correct operation only if constraint met
Most prevelant
Examples:
Embedded computers in devices
Automobiles
Robotic arms


Operating system sources
Open source
Source code available
Some are not free
Examples:
GNU/Linux Most popular
BSD UNIX including Mac OS X core
.java
Free source
Source code available
licensed to allow no-cost use, redistribution, and modification
Free Software Foundation (FSF)
Copyleft: GNU General Public license (GPL)
Closed source
Binary code
Examples:
Microsoft windows
.class


Computer system components
Application programs
Define how to solve user computing problems
Hardware
Operating system
Users
People or Other machines connected to this device
Memory-Storage
Speed, Cost, and volatility
Central processing unit (CPU)
Responsible for execution, Fast
Main memory ← data → local buffer
I/O devices
Storage, Transmission, Human interface
Structure
Primary
Volatile, Fast
Secondary
Permenant, Slow
Tertiary
Permenant, Slow, cheap, and large storage for archive
Each controller has a driver
Main memory (RAM)
Electronic memory
CPU can access directly
Cache for secondary
Registers
Inside CPU
Cache
Between CPU and main memory, small part of it could be inside CPU
Hard-Disk Drives (HDD)
Mechanical movements
Divided into tracks, subdivided into sectors
Non-Volatile memory
Examples:
Solid-state Disk (SSD)
No Mechanical movements
Faster than HDD, More popular
Various technologies
ROM
EPROM
Flash memory
Optical disk
Magnetic tapes
Device Driver (Software)
Computer program allow system to interface with hardware
Translator between programs and devices
Allow access to device functions
Convert I/O instructions into device operations
Examples:
Disk drive
Graphics display
Audio device
Print, bluetooth driver
Device Controller (Hardware)
Chip
local buffer + set of special purpose registers
Connect I/O devices to system bus
Make sense of signals going/coming from CPU
Each is in charge of specific type of device
Device ← data I/O → local buffer of controller


storage capacity
access time
smaller
larger
faster
slower
registers
cache
main memory
volatile storage
nonvolatile storage
nonvolatile memory
hard-disk drives
optical disk
magnetic tapes
primary storage
secondary storage
tertiary storage




# System call

What is system call:

- Function - Software interrupt
- Provide an interface to OS services
- Door that process can access OS services
- written in high level language but sometimes in assembly language

System call interface:

- Indexed table
- Invokes system call and return status + value
- Trap:
  - Change mode from user to kernel

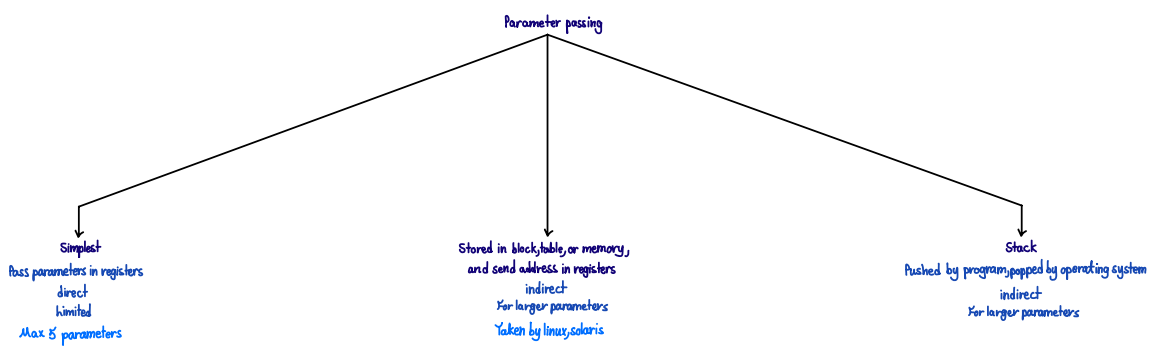
Mostly access by Application Programming Interface (API) rather than system call:

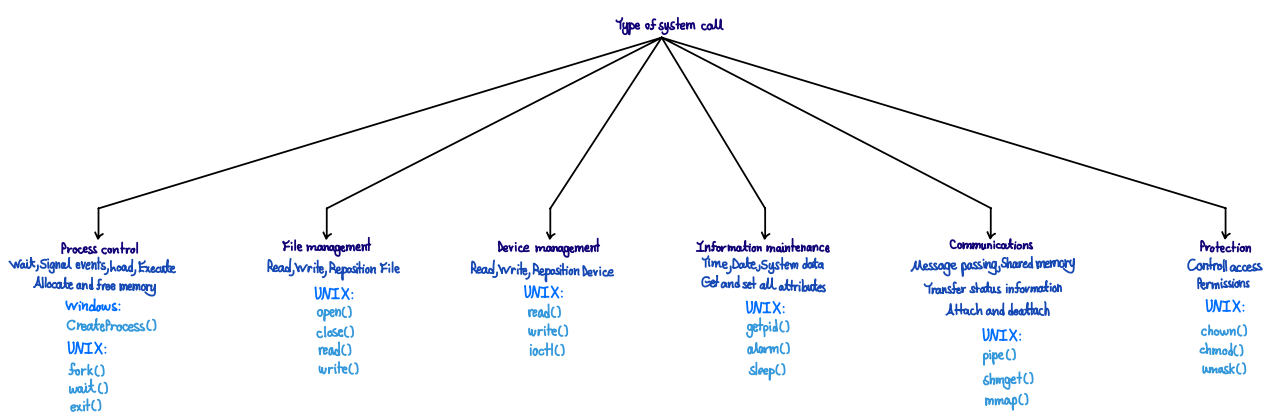
API:

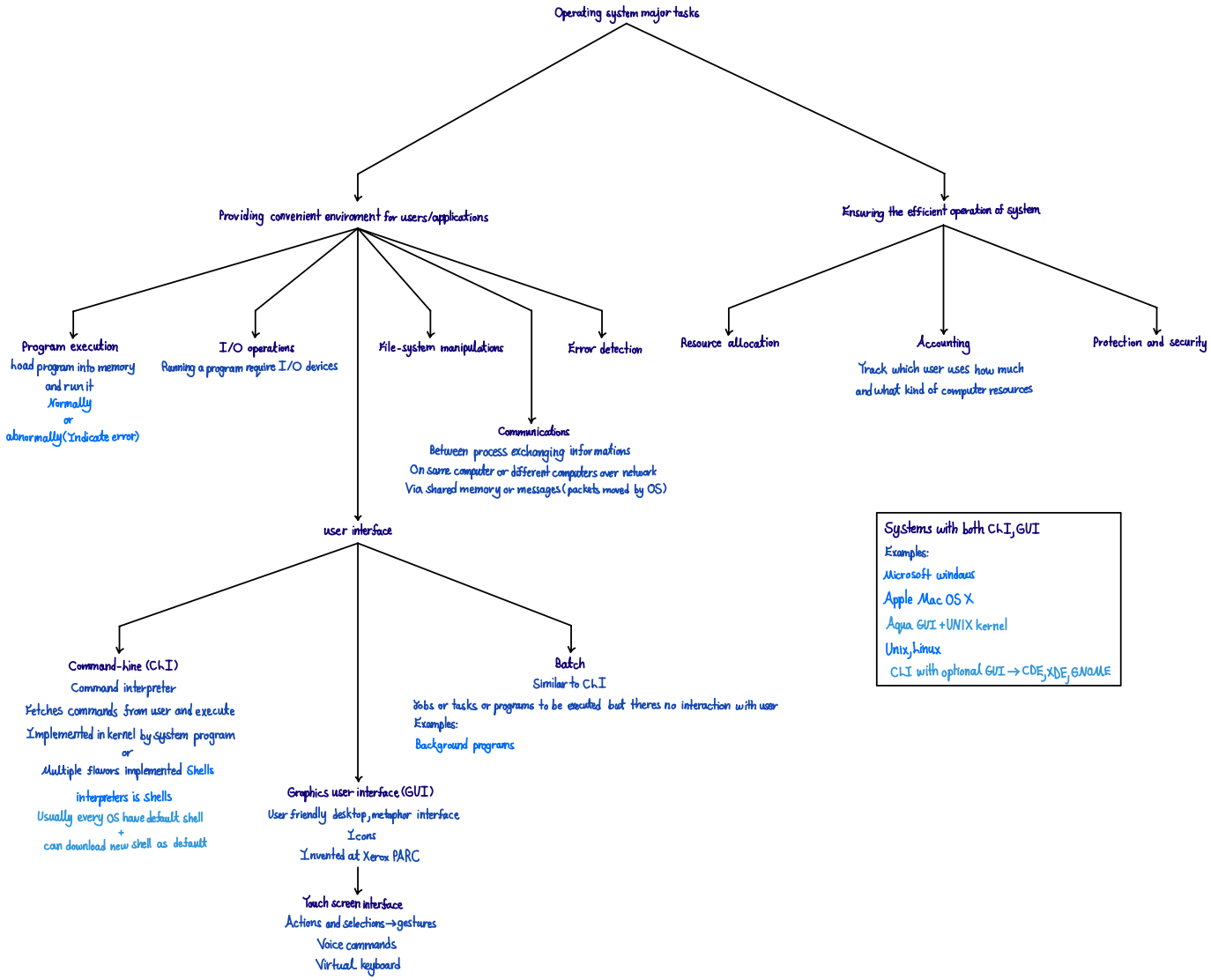
- Wrapper for system call, it call the system call
- Decides interfaces between different systems

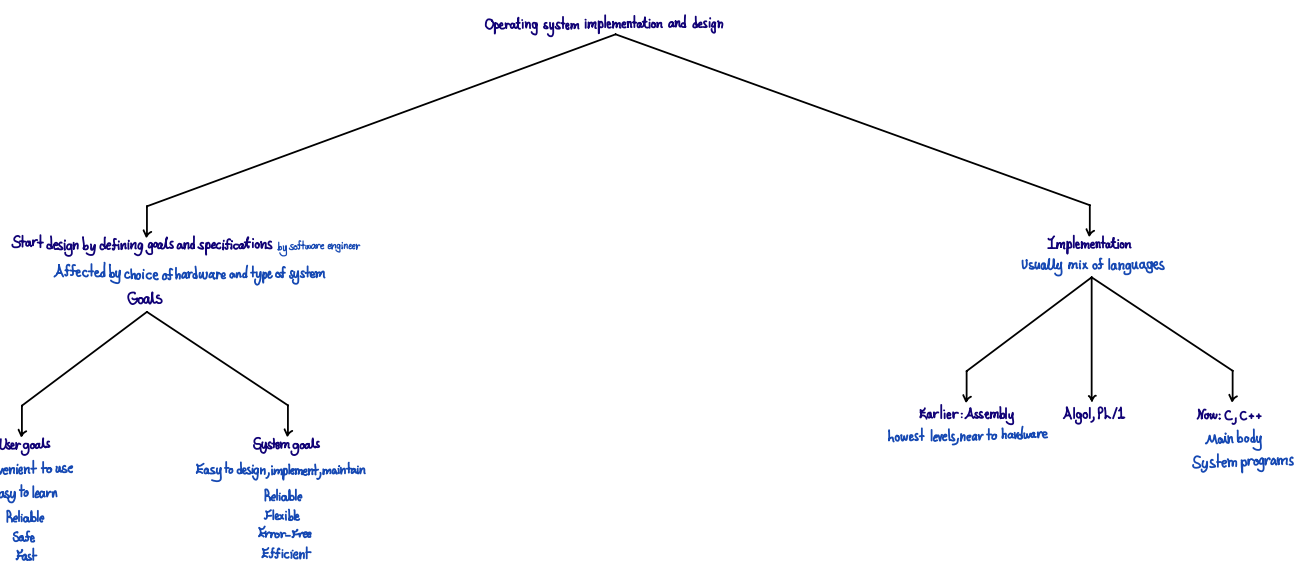
Most common API:

1. Win32 for Windows
2. POSIX for POSIX-based systems: UNIX, linux, Mac OS X
3. Java for JVM




Operating system major tasks
Providing convenient environment for users/applications
Ensuring the efficient operation of system
Program execution
load program into memory and run it
Normally or abnormally (Indicate error)
I/O operations
Running a program require I/O devices
File-system manipulations
Error detection
Communications
Between process exchanging informations
On same computer or different computers over network
Via shared memory or messages (packets moved by OS)
Resource allocation
Accounting
Track which user uses how much and what kind of computer resources
Protection and security
user interface
Command-line (CLI)
Command interpreter
Fetches commands from user and execute
Implemented in kernel by system program or
Multiple flavors implemented Shells
interpreters is shells
Usually every OS have default shell + can download new shell as default
Batch
Similar to CLI
Jobs or tasks or programs to be executed but theres no interaction with user
Examples:
Background programs
Graphics user-interface (GUI)
User friendly desktop, metaphor interface
Icons
Invented at Xerox PARC
Touch screen interface
Actions and selections → gestures
Voice commands
Virtual keyboard
Systems with both CLI, GUI
Examples:
Microsoft windows
Apple Mac OS X
Aqua GUI + UNIX kernel
Unix, linux
CLI with optional GUI → CDE, KDE, GNOME


Operating system implementation and design
Start design by defining goals and specifications by software engineer
Affected by choice of hardware and type of system
Goals
User goals
Convenient to use
Easy to learn
Reliable
Safe
Fast
System goals
Easy to design, implement, maintain
Reliable
Flexible
Error-free
Efficient
Implementation
Usually mix of languages
Earlier: Assembly
lowest levels, near to hardware
Algol, PL/1
Now: C, C++
Main body
System programs

## CPU attention for services

1-Polling
- Turn around one by one
- Slow, not effecient

2-Interrupts
- Urgent call to OS
- Signal from software or hardware

1-Hardware Interrupt
- Sending signal to CPU using system bus

2-Software Interrupt
- System call
- Trap or Exception
- By Error or specific request

## How does interrupt work?

When interrupt request is generated:
- Current execution suspended
- Control transfered to Interrupt Service Routine(ISR)
- through interrupt vector,
- and it initiate Interrupt Handler(IH).

Operating system preserve the process state of CPU by storing:
- Register and Program counter

Interrupt Service Routine(ISR):
- Software
- Part of Operating system

Interrupt vector:
- Contain address of all service routine
- Table of pointers to IH
- Stored in memory resedient
- low memory, fast access
- Indexed with unique device number given with the request

Application ⇄ I/O request ⇄ system call ⇄ Device driver ⇄ Device controller ⇄ Device

Interrupt Request(IRQ)
to inform
+
return value or pointer

## System programs

Ships with operating system
Can be part of kernel

Examples:

Compiler
- Input → source code any high level language
- Checks the syntax, then generate object code
- Works on the whole program

Assembler
- Type of compiler
- Input → assembly language code
- then works as compiler

Interpreter
- Can work on a part of the program
- Eg: python

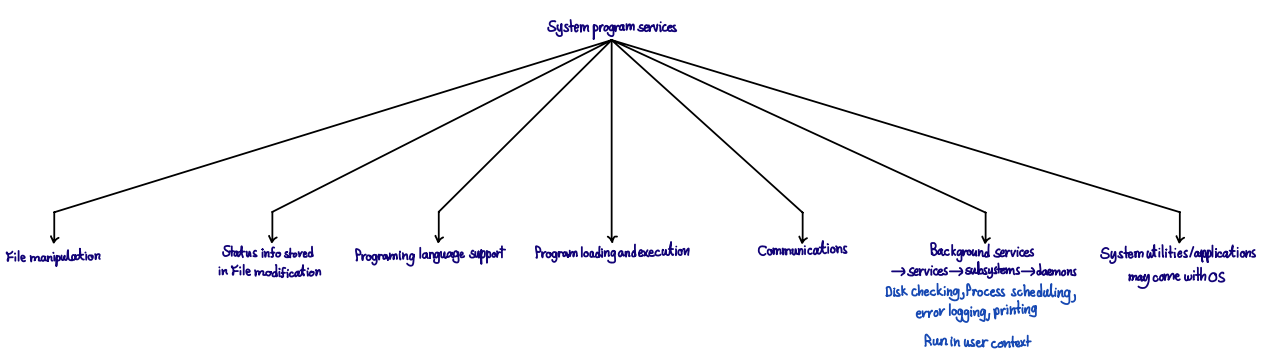
Device driver

Text editors

## Application programs

Don't pertain to system
Run by users

Examples:
- Word processors
- Spreadsheets
- Web browsers
- IDE
- Math solvers
- Games
- Database systems
- Plotting tools



# Operating system structure

## 1- Monolithic structure - Original UNIX

Two parts:

System programs

The kernel

Benefits and limitations:

+Performance - Effecint from communication side of view

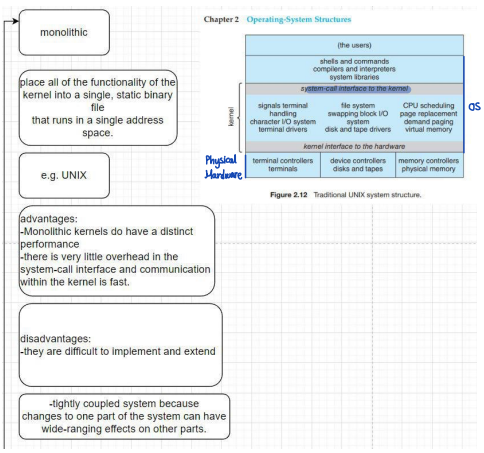
+ hittle overhead

+Speed

- Debugging - Huge line of code (thousands)

- Difficult to implement and extend

- Difficult to change - Tightly coupled




## 2- layered

Abstraction

Divide operating system into layers

Each layer is self-contained → cannot call higher level services

Benefits and limitations:

+Simple to construct, debug, and verify

+Each level hides implementation details of lower layers

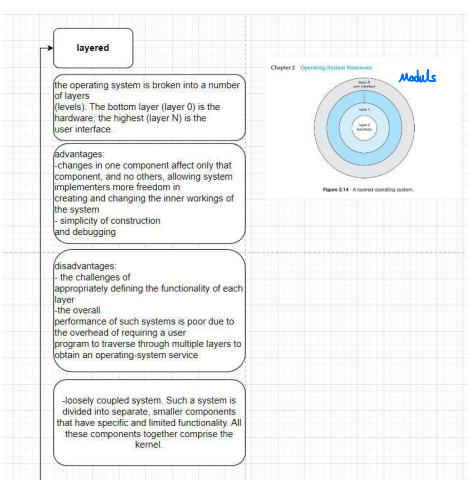
+Easy to change

- hange overhead → Not effecint in moving through layers

- Defining layers difficulty

- hasely coupled

- Performance - overhead




## 3- Microkernel

Moves from kernel into user space Only small piece is still in kernel

Message passing

Example: Mach and Drawin

Benefits and limitations:

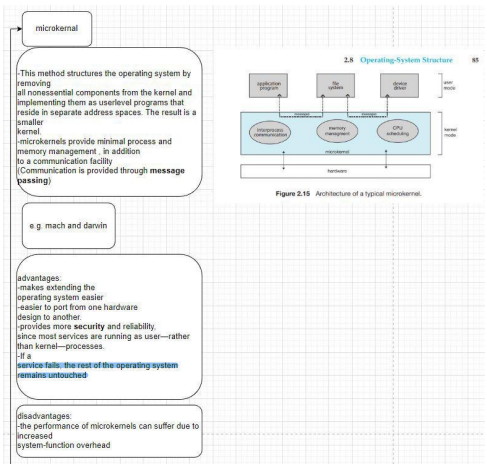
+ More reliable, secure

+Easier to extend

+Easier to port OS to new architecture

- Performance overhead of user and kernel communication

- System function overhead




## 4- Modules The best

loadable kernel Modules

Object-oriented approach

Each core component is separate and talk through interfaces

Similar to layers but more flexible

No message passing

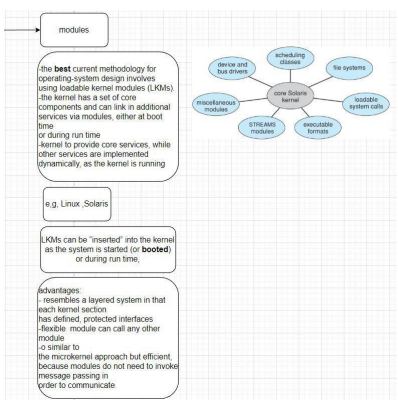
Example: hinux, Solaris

Benefits and limitations:

+Effecint from communication side of view

+Protection

+Flexible




## 5- Hybrid systems

hinux and solaris Monolithic + modular

Windows monolithic + microkernel

Apple Mac OS X hybrid, layered, Aqua UI, Cocoa programming environment

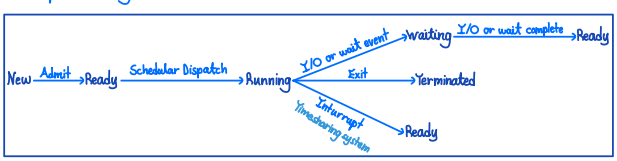
# Process

Process address space, memory image parts:

1- Text section
   Program code
2- Data section
   Global variables
   (Constant size)
3- Stack, Stack pointer
   Temporary data
   Eg: Parameters, return address, local variables
4- Heap
   Memory dynamically allocated during run time
   (Variant size)

Process state:

1- New
2- Ready
   Ready for execution but currently not under execution
3- Running
4- Waiting
5- Terminated
   Completed normally or crashed




Process Control Block (PCB), Task Control Block:

1- Process state
2- Process number
3- Program Counter Register - PC
   Points to the next instruction to be executed
4- CPU Registers
   Instruction Register - IR
   Includes the one instruction under execution
   IR size: one word
5- CPU scheduling information
6- Memory management information
7- Accounting information
8- I/O status information

Context switch:

Saving the old process state (Screenshot), load new process saved state
Pure overhead

Process scheduling:

Multitasking: Process should be scheduled ⟶ Multiprocess ⟶ one CPU

Scheduling queue:

1- Job queue
   Larger than ready queue
   In Hard disk, since Main memory is not enough
2- Ready queue
   In Main memory
3- Devices queues

Schedulers:

1- Short-term schedulers
   CPU scheduler
   Invoked frequently ⟶ ms
2- Long-term schedulers
   Job scheduler
   Invoked infrequently ⟶ s, m
   Control degree of multiprogramming
   Good process mix
3- Medium-term schedulers
   If degree of multiprogramming needs to decrease
   Swapping: Memory ⟷ Disk

Process types:

1- I/O-bound process
   Many-Short CPU bursts
2- CPU-bound process
   Few-Very long CPU bursts

Process types:

1- Independent
2- Cooperating
   Advantages:
   1- Information sharing
   2- Computation speedup
   3- Modularity
   4- Convenience

System mechanism for process:

1- Process creation

Resource options:

1- Parent and child share all resources
2- Child share subset of parent resources
3- Parent and child share no resources

Execution options:

1- Parent and child execute concurrently
2- Parent wait until child terminates

Address space:

1- Child duplicate of parent
   System will create an address space for child in a different space than parent's space
   Child address space is a copy paste from parent's address space
   fork() return value:
   If successfull ⟶ will return new process child ID > 0
   If failure error ⟶ will return < 0
   I am child ⟶ will return 0
2- Child has program loaded into
   System will create an address space for child in a different space than parent's space
   Child address space is a copy paste from parent's address space
   New program is loaded into child

Methods:

linux-Posix, C: fork() Creation
exec() execl() execlp() loading
Windows: CreateProcess() Creation + loading

2- Process termination

exit() - Process terminate itself
Return status data from child to parent and pid using wait()
Process resources deallocated by OS

abort() - Parent process terminate child process
Reasons:
1- Child exceeded allocated resources
2- Task assigned to child no longer required
3- The parent is exiting and OS doesn't allow child to continue

Cascading termination
All child and grandchild are terminated

Orphan
If parent terminated without invoking wait

3- Interprocess communication (IPC)

1- Shared memory
   Cooperating process should be in the same device ⟶ Access to common memory
   Frequent messaging, large amount of data
   Faster as there is no system calls ⟶ One system call
   No kernel interference
   Type of buffers:
   1- Unbounded
   2- Bounded
2- Message passing
   Cooperating process in the same device or different devices
   System calls for every sending and receiving
   Infrequent messaging, small amount of data
   Communication link + Methods
   Methods:
   1- send(message)
   2- receive(message)
   Communication link implementation:
   1- Physical
      1- Shared memory
      2- Hardware bus
      3- Network
   2- Logical
      1- Direct or Indirect
         Indirect methods: create and destroy or delete port, send and receive
      2- Synchronous or Asynchronous
         If both are synchronous ⟶ rendezvous
      3- Automatic or explicit buffering
         1- Zero capacity
         2- Bounded capacity
         3- Unbounded capacity

Operating system execute programs:

1- Batch system - Jobs
2- Time-shared system - User program or task

# Threads

kernels are generally multithreaded
Basic unit of CPU utilization

Each thread has:
1- Thread ID
2- PC
3- Register set
4- Stack

Threads share:
1- Text section
2- Data section
3- OS resources

Threads benefits:
1- Responsiveness
2- Resource sharing
3- Economy
4- Scalability

Computer system architecture:
1- Single processor systems
One cpu, one core
2- Multiprocessors - Parallel systems
Many cpu's each has core
3- Multicore
One cpu, many core's

Multiprocessors and Multicore challenges:
1- Dividing activities
2- Balance
3- Data splitting
4- Data dependency
5- Testing and debugging

Parallelism and concurrency:
Concurrency:
Single processor
Parallelism → Simultaneously:
Multiprocessors and Multicore
1- Data parallelism
Dividing data into cores
All cores are performing same operation
Working on same code and function, but different sections
2- Task parallelism
Every core is performing different operation (Unique) on same data

Amdahl's law:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}} \quad \text{— Parallel}$$

Thread creation categories:
1- Synchronous
2- Asynchronous

Multithreading supported by:
1- User level libraries - User threads
Unknown to kernel
Faster than kernel threads
2- kernel itself
Eg: Operating systems

Multithreading models:
1- Many-to-One
User thread
Effecient but didn't get full benefits of process
Not popular
Examples:
Solaries green, GNU Portable

- No need for kernel support for multithreading (+)
- Thread creation is fast (+)
- Switching between threads is fast; efficient approach (+)
- Blocking system calls defeat the purpose and have to be handled (-)
- Multiple threads will run on a single processor, not utilizing multi-processor machines. (-)

2- One-to-One
kernel thread
Popular, most common
Examples:
Windows, linux

- Provides more concurrency; when a thread blocks, another can run. Blocking system calls are not problem anymore. (+)
- Multiple processors can be utilized as well. (+).
- Need system calls to create threads and this takes time; (-)
- thread switching costly; Number of threads per process sometimes restricted due to overhead (-)
- any thread function requires a system call. (-)

3- Many-to-Many
Between the two previous
Not so sucessfull
User threads => kernel threads

4- Two level
Similar to Many-to-Many, but allow user to bound kernel

Thread main libraries:
Provide API
1- Pthread
Either user or kernel level
POSIX
Specification not implementation
Data declared globally
Common:
UNIX: linux, MAC OS X
Methods:
pthread_attr_init() Default initialization
pthread_create() (4 parameters) Creation, Return 0 if suceed
pthread_exit()
pthread_join()
pthread_cancel()
2- Windows
kernel level
Data declared globally
3- Java thread API

# CPU Scheduling

Ready Queue

Smallest entity to be scheduled in CPU Threads

CPU decisions when:

1-Switch Running → Waiting Nonpreemptive
  Eg: I/O request, wait()
2-Switch Running → Ready Preemptive
  Eg: Interrupt, Timer interrupt
3-Switch Waiting → Ready Preemptive
  Eg: I/O completion
4-Terminates Nonpreemptive

Nonpreemptive and Preemptive:

Nonpreemptive:
Process realeses the CPU
Preemptive:
All OS uses
Consider:
1-Shared data
2-Preemption while in kernel time
  Affect design of OS kernel
3-Interrupts occuring during crucial OS activities

Dispatcher:

1-Context switch < 1o microseconds
  1-Voluntarly
  2-Nonvoluntarly
2-Switch to user mode
3-Jump to the proper location in user program to restart the program

Scheduling critiria:

1-CPU Utilization MAX
  Range 40%-90%
  CPU Utilization = $\frac{\text{Active time CPU}}{\text{All time CPU}}$ = 1-Total CPU idle time
2-Throughput MAX
  Depend on CPU and process types:
  1-Long process low throughput
  2-Short process High throughput
3-Turnaround time MIN
  Completion time-Arrival time
4-Waiting time MIN
  Turnaround time-Burst time
5-Response time MIN
  First response-Arrival time

Scheduling Algorithms:

1-First Come First Serve (FCFS): Nonpreemptive Single processor, CPU, Core
  Special case of priority
  Problem: Average waiting time
  Trouble some to interactive systems
  Convey effect: long process then short process
    One CPU-bound, Many I/O-bound process
    lower CPU and device utilization

2-Shortest Job First (SJF): Nonpreemptive Single processor, CPU, Core
  Special case of priority
  Shortest next CPU bursts
  Optimal-Gives minimum average waiting time
  Estimate length of next CPU burst (Exponential average):
  $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
  $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

  $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts Recent history

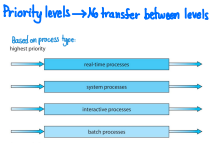
  $\alpha = .5$
  Recent and past history has equal weight

3-Shortest Remaining Time First: Preemtive version of SJF Single processor, CPU, Core

4-Round Robin (RR): Preemtive Single processor, CPU, Core
  Treated as FIFO
  Higher turnaround than SJF but better response
  n process in ready queue → Each will have $\frac{1}{n}$ CPU time
  No process waits more than (n-1)*q
  Time quantum-Time slice (q):
  Usually 10-100 ms
  q large → FIFO
  q small → must be large since context switch will be overhead if q is small

5-Priority scheduling: Preemtive or Nonpreemptive Single processor, CPU, Core
  Priority:
  1-Internally defined
    Based on OS
  2-Externally defined
    Process comes with its number
    Eg: Based on importance, type and amount
  Problem: Starvation
  Solution: Aging

6-Multilevel Queue Single processor, CPU, Core
  Priority levels → No transfer between levels




7-Multilevel Feedback Queue Single processor, CPU, Core

Multilevel-feedback-queue scheduler defined by the following parameters:
- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service

8-Multiple processor scheduling

1-Heterogeneous-Asymmetric
  Master server
2-Homogeneous-Symmetric
  More common since it's distributed
  -load balancing

# Synchronization tools

**Critical section:**
Code segment that access shared data and has to be executed in atomic action

**Critical section problem solution(Conditions):**
1- Mutual exclusion
If process i is executing in critical section, no other process can be executing in critical section
2- Progress
Selection of next process to enter critical section cannot be postponded
3- Bounded waiting
Bound the number of times that the process have asked to wait

**Critical section handling in OS:**
1- Preemtive
Allow process to be preemted will running in kernel mode
More responsive → Suitable for real-time programming
2- Non-Preemtive
Doesn't allow process to be preemted will running in kernel mode
Free from race conditions

**Software solutions:**
1- Global int variable
1- Mutual exclusion yes
2- Progress no
3- Bounded waiting yes
2- Two global boolean variables
1- Mutual exclusion no
2- Progress yes
3- Bounded waiting yes
3- Two global boolean variables → Turn flag before entering loop
1- Mutual exclusion yes
2- Progress no
3- Bounded waiting yes

**Peterson's solution:**
Software-based solution to critical section problem
load,store machine language must be atomic
Not guaranteed to work on modern architecture
Share two variables:
1- int turn
2- boolean flag[2]

**Hardware-based solution to critical section problem:**
Uniprocessor to disable interrupt → Not efficient on multiprocesor systems

**High-level software tools solution to critical section problem:**
1- Mutex (Mutual exclusion)
Simplist
Using locks
Variables: boolean available
Methods: aquire(), release() atomic
Disadvantage: Busy waiting=Spinlock No context switch
2- Semaphore
More robust tool
Provide more sophisticated ways for process to synchronize
Variables: int S, list of process
Methods: wait() → P(), signal() → V()
Semaphore value:
1- Counting
Unrestricted domain
2- Binary
Between 0 and 1
Same as mutex
Semaphore could be with busy waiting or without busy waiting
With busy waiting → waiting queue:
FIFO To insure bounded waiting
Data items: int value + pointer to next in list
Operations:
block() or sleep() Place into waiting queue
wakeup() Waiting state → Ready state

**liveness:**
liveness failure:
Indefinite loop
Deadlock

**Classical problems of synchronization:**
1- Bounded buffer
Shared data:
Semaphore mutex → 1
Semaphore full → 0
Semaphore empty → n
2- Readers and writers
Reader → only read
Writers → read and write
Shared data:
Data set
Semaphore rw_mutex → 1
Semaphore mutex → 1
int read_count → 0
First variation: No reader wait unless writer is writing
Second variation: Once writer is ready → write ASAP
Solution:
Readers-writers locks
3- Dining-Philosphers
Shared data:
Data set
Semaphore chopstick[5] → 1

Possible solutions:

- instead of sleeping for a fixed time, the philosopher would sleep for some random amount of time (i.e sleep(random_time).
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

# Main memory

Memory unit only see:
1- Address, read request
2- Address, data and write request

Register ⟶ One CPU clock or less
Main memory ⟶ Many cycles Causing a stall

Protection provided by:
1- Base (base register=relocation register)
Hold smallest legal physical address
2- limit registers

Address representation different stages:
1- Symbolic
Source code
Variable: count
2- Relocatable
Compiled code
X bytes from the begining of module ⟶ logical
3- Absolute
linker or loader
Address ⟶ Physical

Address binding:
1- Compile time
2- load time
3- Execution time

Dynamic loading:
No special support from OS
Can help by providing libraries to implement dynamic loading
Implemented through program design

linking:
1- Static linking
Combined by loader
2- Dynamic linking

Memory management schemes:
Contiguous allocation:
Old method
1- Fixed partitioning
Variable sizes but will never change
Internal fragmentation + External fragmentation
2- Variable partitioning
OS should maintain:
1- Allocated partitions
2- Free partitions ⟶ Hole
External fragmentation
Non-Contiguous allocation:
1- Paging
Frames:
Fixed size ⟶ Power of 2
Physical memory
Pages:
Fixed size
logical memory
Avoid external fragmentation
Still have internal fragmentation
2- Segmentation
Segments:
Variable size

Fragmentation:
1- Internal fragmentation
2- External fragmentation
Reduced by compaction

Dynamic storage allocation:
1- First-Fit
N blocks allocated ⟶ 0.5 N blocks lost to fragmentation ⟶ 1/3 unusable ⟶ 50 percent rule
2- Best-Fit
3- Worst-Fit

# Virtual memory

Benefits:
More concurrent programs
less I/O needed to swap/load portions of programs
Execute programs larger than RAM size
Easy sharing of address spaces by several process
Allow more effecient process creation

Demand pages:
less I/O needed
less memory needed
Faster response
More users

Page fault activities:
1- Service the interrupt
2- Read the page
3- Restart the process

Page fault rate:

$$\text{Page fault rate} = \frac{\text{Page faults}}{\text{Total memory references}}$$

0 ⟶ no page fault
1 ⟶ Every reference is page fault

Effective access time:
EAT = (1-P) x memory access + P x (page fault overhead + swap page out + swap page in)

What if no free frames:
1- Poor options
2- Prevent over allocation
3- Page replacement

Page replacement algorithm:
1- FIFO algorithm 3
2- Optimal algorithm 1
3- least recently used 2
4- least frequently used Counting algorithm
5- Most frequently used Counting algorithm

Frame allocation:
1- Equal allocation
2- Propotional allocation

$$a_i = \frac{s_i}{\sum s_i} \times \text{total number of frames}$$

Frame allocation:
1- Global allocation
2- local allocation