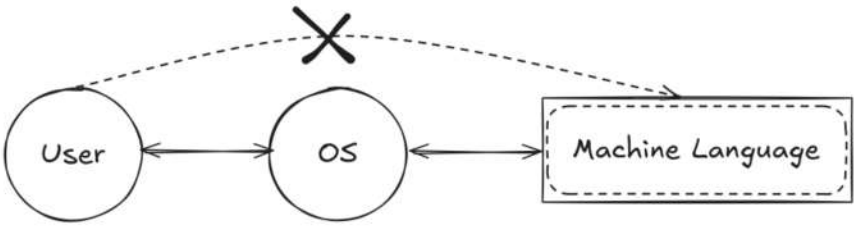
Operating Systems

Chapter 1 # Introduction

❑ Operating Systems Goals

(1) Overall goal: Execute user programs
(2) Primary goal: Conveniency



* It's easier for the user to interact with the OS.

(3) Secondary goal: Efficiency

❑ Computer Resources

(1) CPU
(2) Memory
(3) I/O Devices

* The OS is responsible for managing the resources.

❑ Operating System (OS)

A set of Algorithms that run the computer machine.

- OS manages the **computer resources** and **must** manage them **efficiently**.

Other goal of OS: Utilization of computer resources

- Utilization of CPU: make the CPU as busy as possible.
- Utilization of Memory: To benefit or use the memory as much as possible.
- Utilization of I/O Devices.

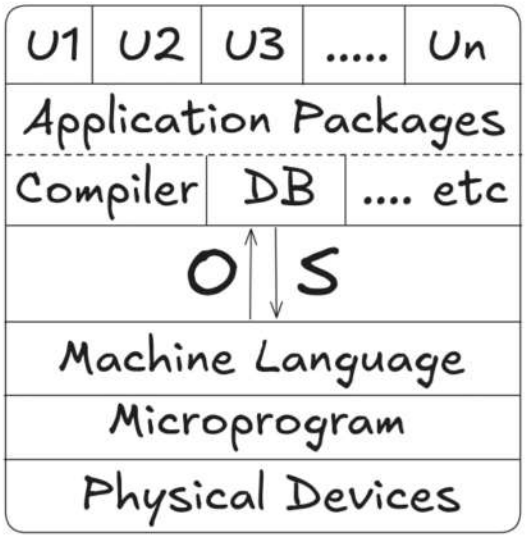
- Computer Performance can be measured ***with throughput.***
- **Throughput**: The number of jobs(programs) that finish execution per unit of time.

❑ Computer

[1] Hardware

- Physical devices: chips, wires, power supplies, ...
- Microprogram: A primitive software that communicates with physical devices which is **an Interpreter** that fetches.
- Fetch: Execute machine language (Assembly Language) instructions.

[2] Operating System



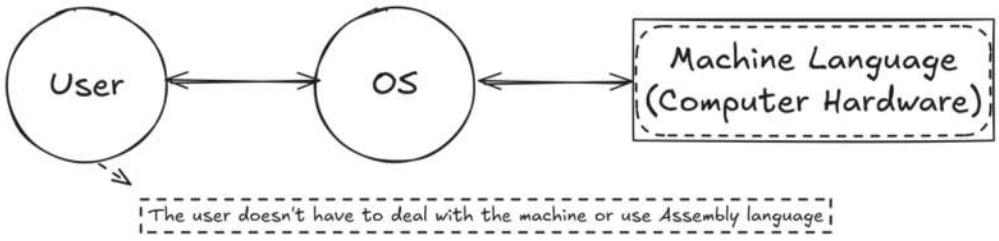
[3] Application Packages

[4] User Programs

❑ Operating System Views

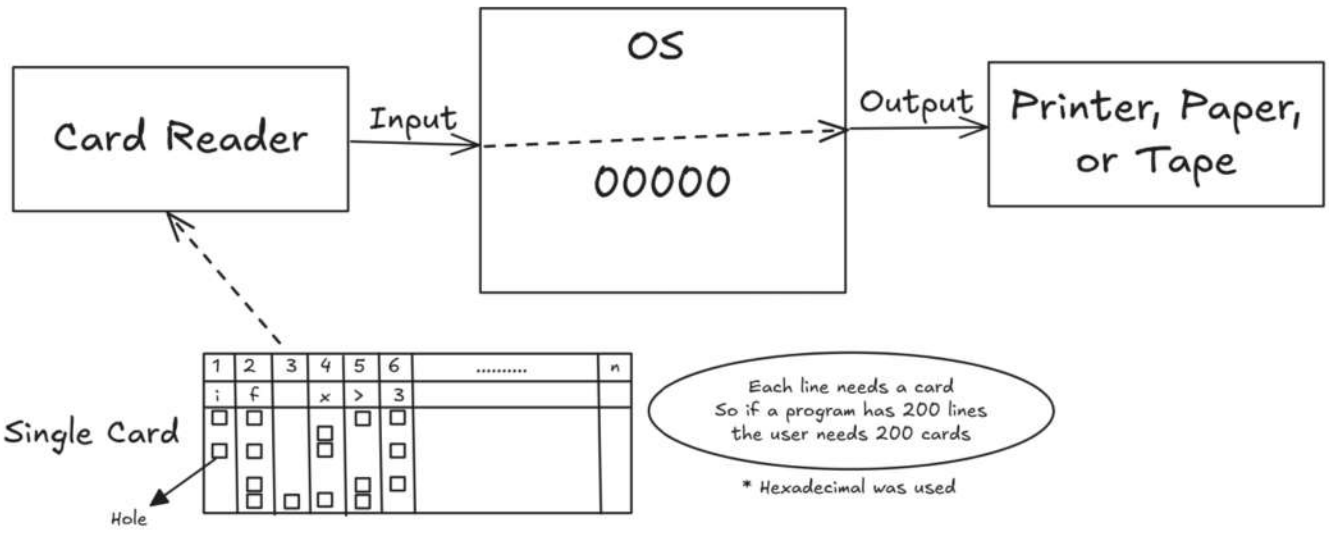
(1) It's central program: It control the execution of all the programs → Overall goal
(2) Extended machine: Extension of the physical machine → Primary goal

* That, it hides all the complexity of system programming and provide the user with simple clean machine to deal with.



(3) Resource Manager: It manages the computer resources efficiently → Secondary goal
(4) Kernal: Part of the OS that's always running and executing instructions.

❑ History & Evolution of OS

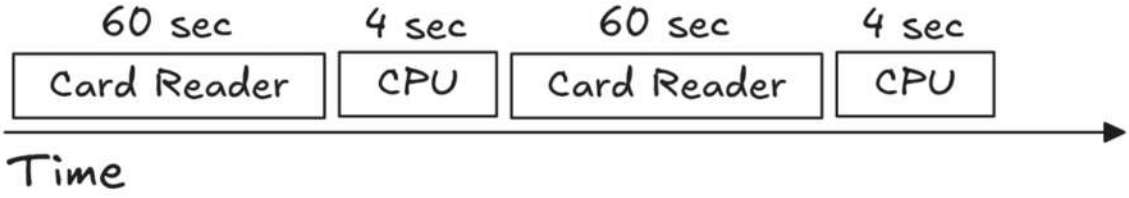


Early Software:

- Machine Language
- Assembly Language (Assemblers)
- Loaders
- Linkers: The Addition of software to the program
- Compilers

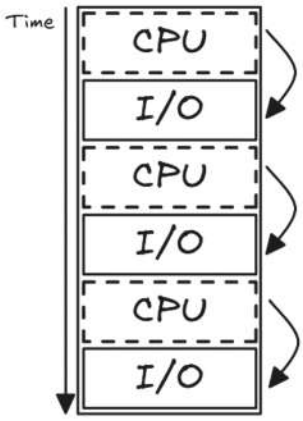
* Poor Performance

- A great portion of time is wasted in set-up time
- No Overlap between I/O & CPU execution
- Low CPU Utilization (due to the big difference between I/O speed & CPU speed)
    - A fast card reader can read 1200 cards/min
    - 1200 / 60 = 20 cards/sec
    - The CPU can process 300 cards/sec



    - Percentage of CPU Utilization = 4/64 = 1/16 = 6%

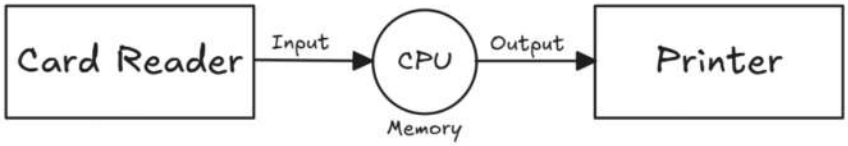
* CPU instruction



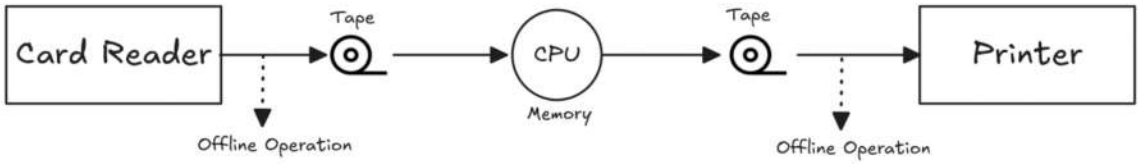
⚠ CPU instruction will keep executing until reaching I/O (Print Output / Read Input)

[A] Offline Operation

(1) Before

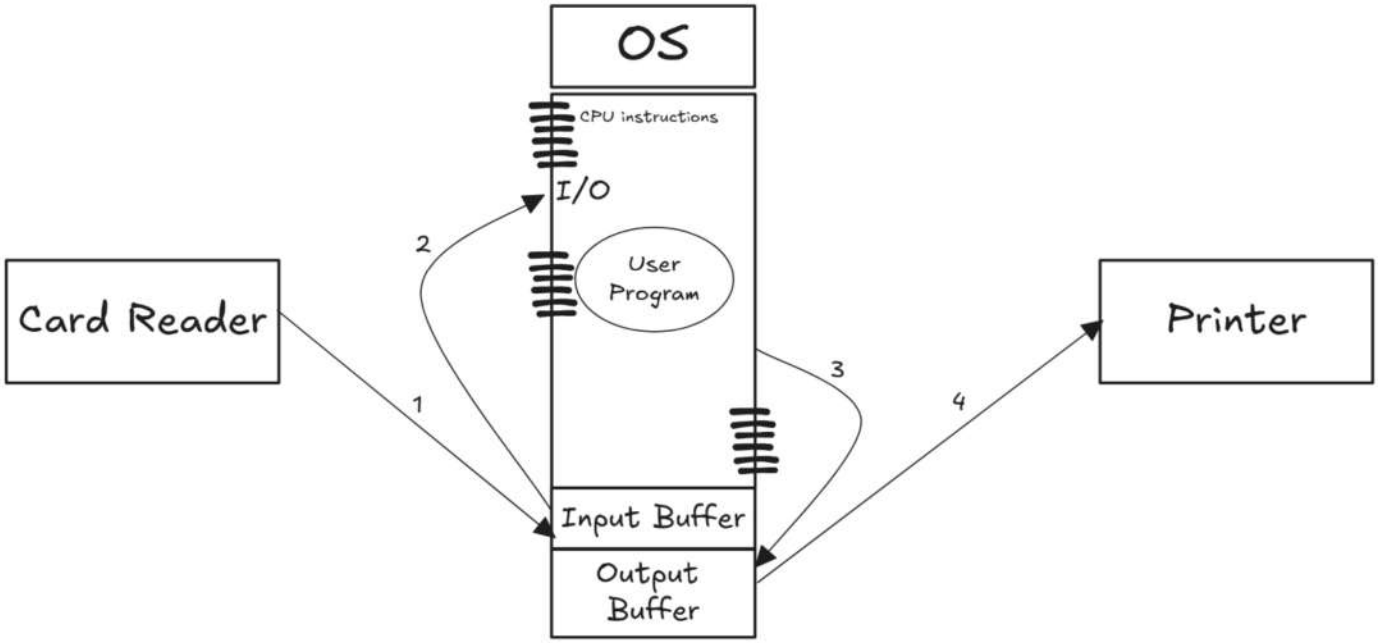


(2) After



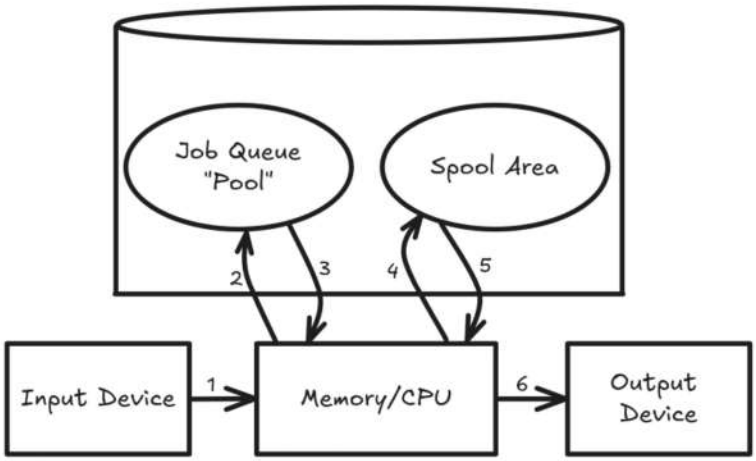
⚠ Tape to Memory is much faster than Card Reader to Memory (Improve Execution)

[B] Buffering



- When CPU reaches I/O it brings data from the buffer not from Card Reader
  ** The **I/O of one job** is *overlapped* with **the execution of the same job**.

[C] Spooling



In Spooling 2 kinds of data-structures were introduced:

(1) Job Queue/Pool: a queue contains the jobs(programs) that demands execution
(2) Spool Area: Contains the jobs that need printing

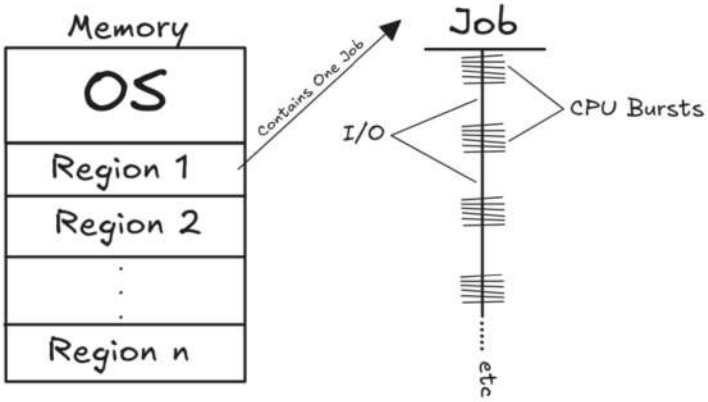
∴ The **I/O of one job** is overlapped with **the execution of another job**

[D] Multiprogramming Batch Systems

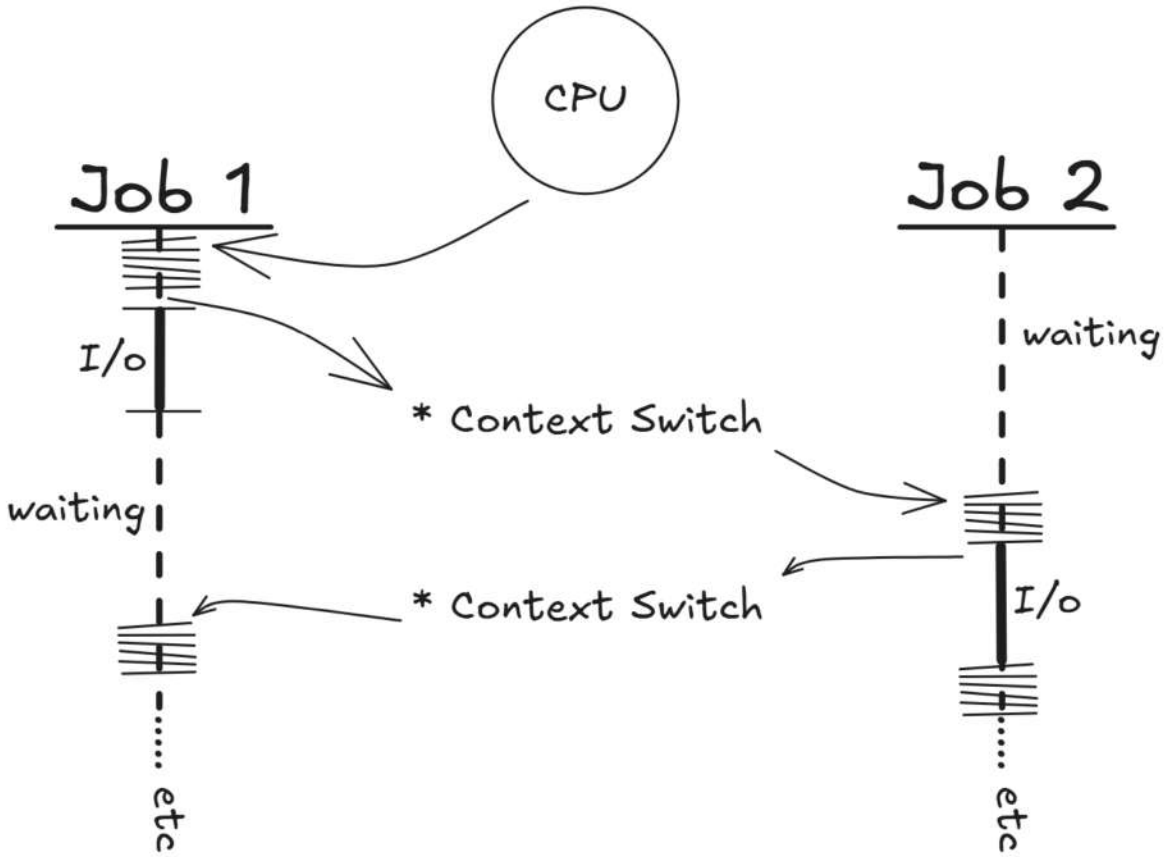
- Memory is divided into regions(partitions)
- Regions sizes normally different
- Each Region contains only one job

Two kinds of jobs:

(1) CPU-bound job: contains **few very long** CPU bursts.
(2) IO-bound job: contains **many very short** CPU bursts.



⚠ Keep in Mind: Any job(process/program) is sequence of **CPU burst** & **I/O wait** and it must start and end with a CPU burst.



** The CPU switches to another job when first one needs I/O

* **Context Switch:** Saves Register for Job 1 & Reloads Register for Job 2

[E] Time Sharing Systems

- Same as multi-programming
- Memory divided into regions
- Several jobs are kept in memory
- Each job is assigned a slice of time called **quantum Q.**
- Each job is executed for this quantum of time & the CPU switches to another job when quantum Q is finished. In addition to normal cases:
    - If the job needs I/O
    - If the job finishes execution
    - High priority process

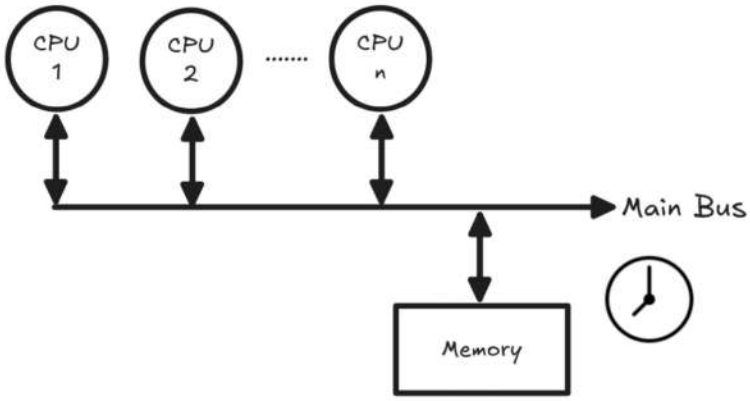
$Multi-programming \neq Time\ Sharing$

- Multi-programming: The CPU switches when the job needs I/O
- Time Sharing: The CPU switches when Quantum Q is finished

[F] Parallel Systems

Multi-processor systems with more than one CPU in-close communication.

(1) **Loosely Coupled Systems:** Ex. Networks, Distributed Systems connected via servers.
(2) **Tightly Coupled Systems:**
Processors share memory, clock, & communication usually takes place in memory



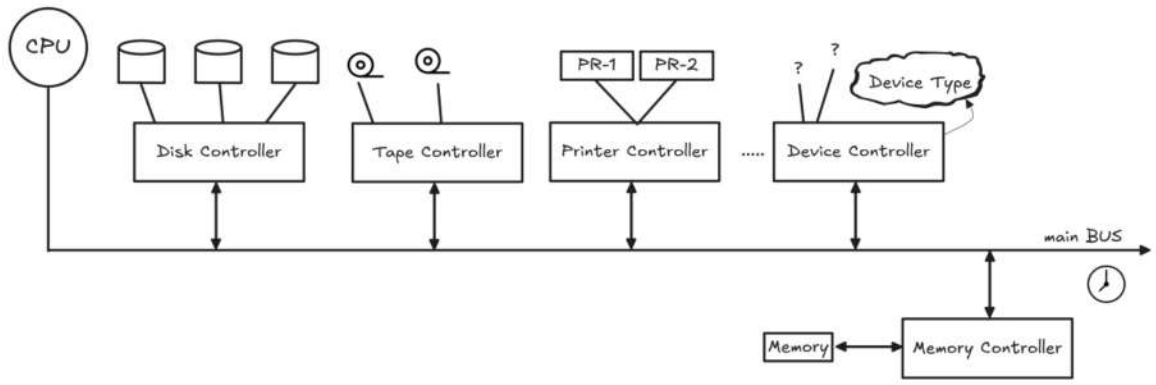
There are two types of processing:

a. **Symmetric multi-processing:** Each CPU has the same identical copy of the OS [Reliable & Simple]
b. **Asymmetric multi-processing:**
   i. There are huge OS that runs this scheme
   ii. There's one CPU called the master CPU which controls other activities of other CPUs
   iii. The relation between the master and other CPUs is called master/slave relationship. Reliable on all cases unless the master CPU is Faulty.

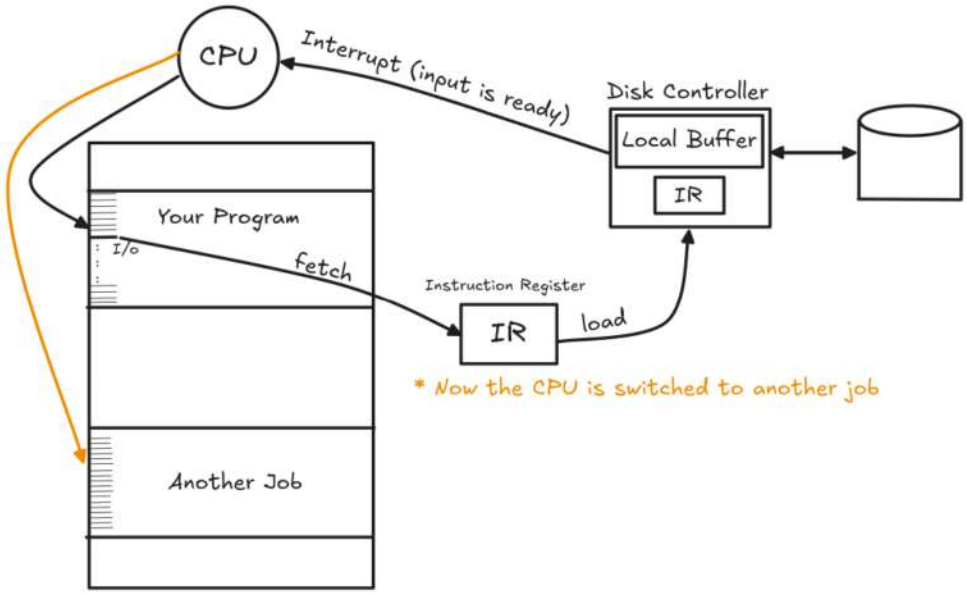
[G] Realtime Systems

- Takes data using sensors
- For systems that needs immediate response

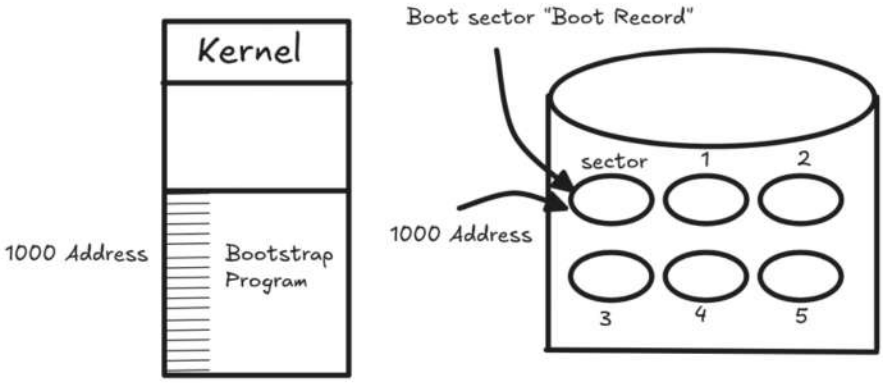
Chapter 2 # Computer System Operations



- I/O can run concurrently with the CPU.
- ⚠ Each device controller is in-charge of particular device.
- ⚠ Each device controller has a local buffer.
- CPU moves data from/to main memory to/from the local buffers.
- I/O is from the device to local buffer of the controller.
- Device controller informs the CPU that it has finished its operation by causing an interrupt.



⚠ Operating Systems are interrupt driven (CPU is interrupted)



❑ Interrupt

A signal sent to the CPU by:

- Hardware
- Software "Trap"

Examples:

- Completion of an I/O. (Hardware Interrupt)
- Division by Zero. (Software Interrupt)
- Invalid Memory Access (Hardware Interrupt)
- Request of an OS service.

OS services that can be asked:

(1) System Program

> format a:

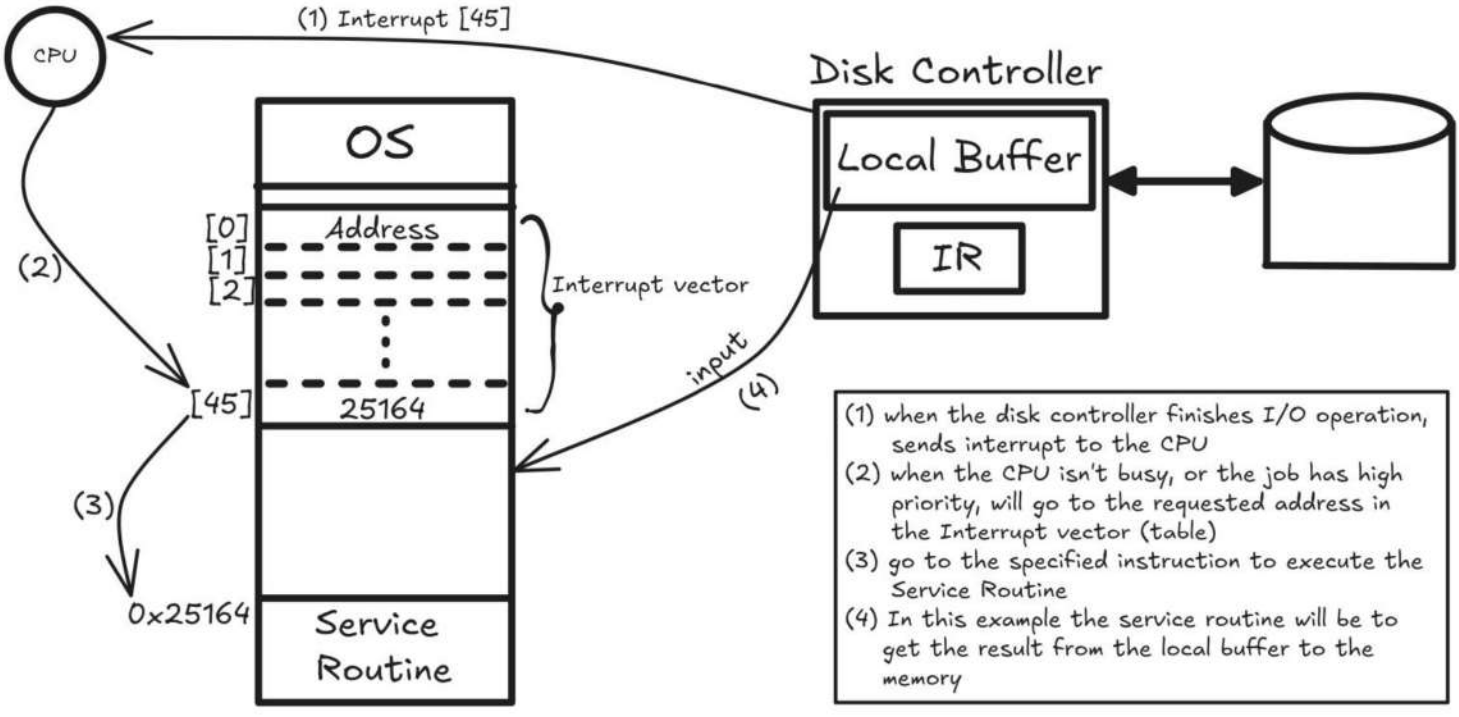
> copy A.dat B.dat

(2) System Call: Its an assembly language instruction.

⬔ Interrupt Handling

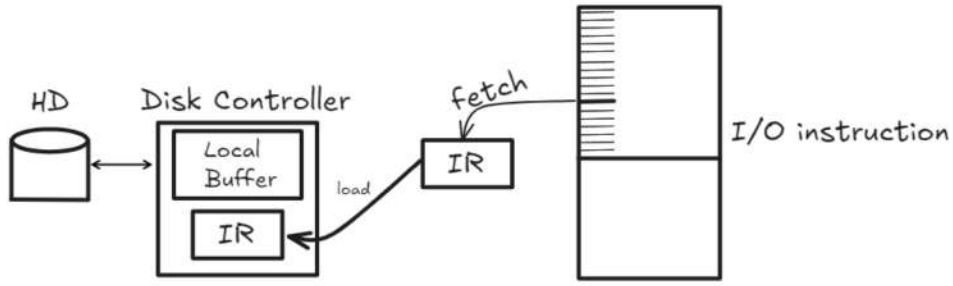
(1) Interrupt Vector (Table)

Assume that the Interrupt which comes from the hard disk informing the CPU the input is completed has number *45*



(2) By Polling

❑ **I/O interrupt structure:**



❑ **Types of I/O:**

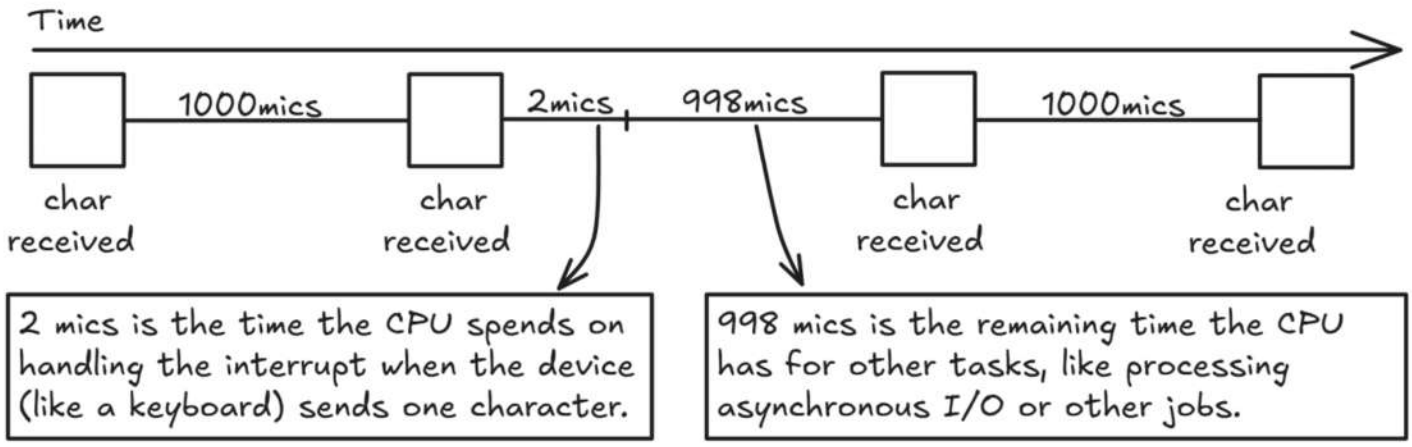
(a) **Synchronize I/O:** after the I/O starts, the control (CPU) returns to the program only upon I/O completion.

- **The CPU waits until the I/O is completed**
- **Wait instruction "CPU is idle"**
- **Loop: jump loop**

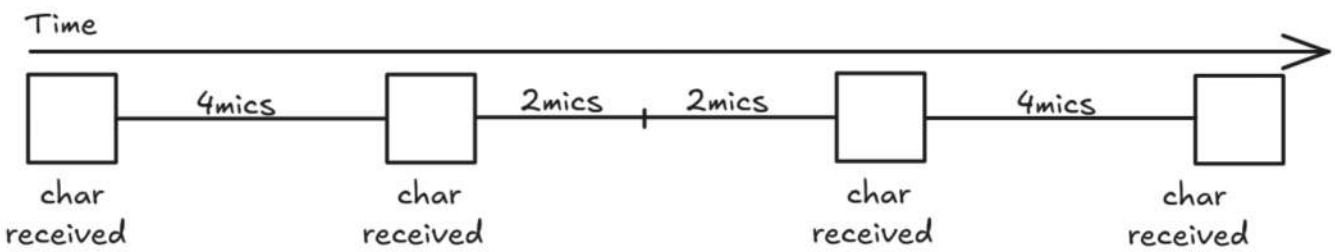
(b) **Asynchronous I/O:** After the I/O starts, the control (CPU) switches to another program without I/O completion.

❑ **Direct Memory Access (DMA):**

- Slow I/O devices such as keyboard, can send one character every 1 milliseconds **1 mils = 1000 mics**.
- The CPU can process (the service routine) to handle the instruction of the character in 2 mics through an I

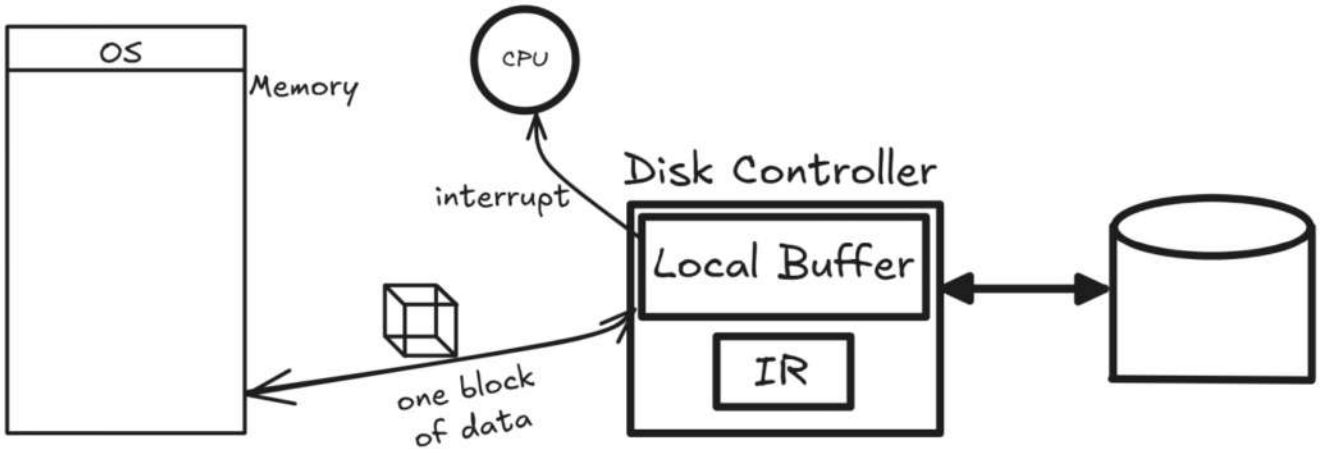


- But, in high speed I/O devices (i.e. Hard Disk) the HD can send/receive char every 4mics



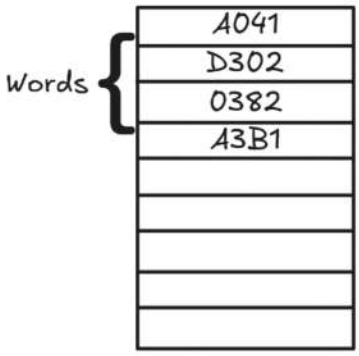
- This leaves the CPU with 4 – 2 = 2mics for Asynchronous multiprocessing **(not enough for switching).**

* In this case (HD) the OS sends one block of data each time & sends an interrupt to the CPU to process this block.

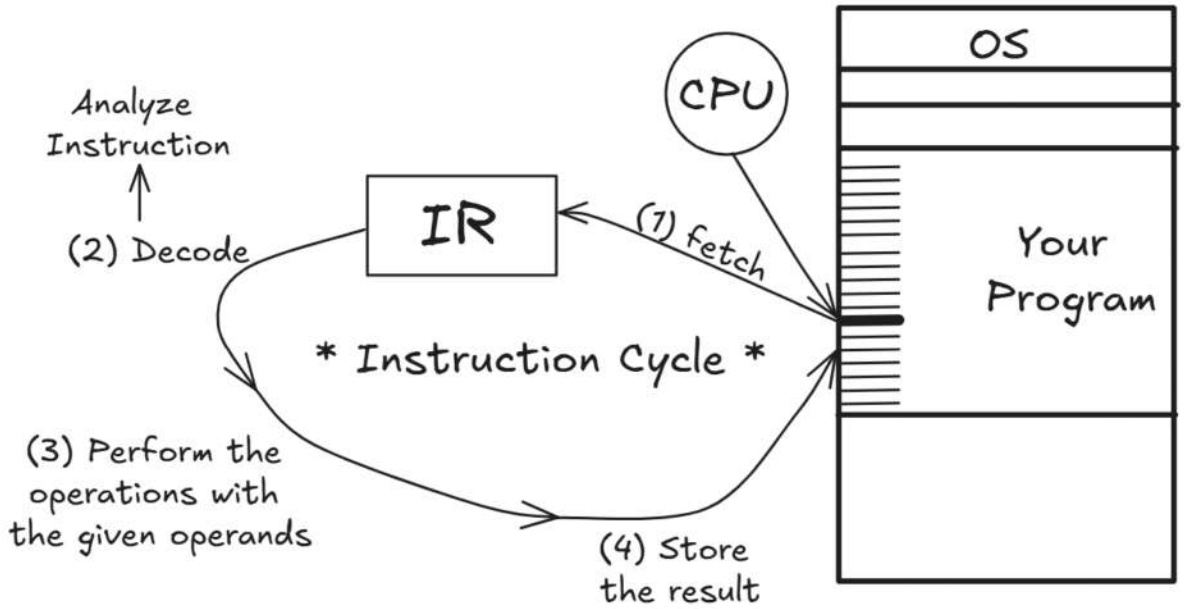


❑ Primary Storage (Memory-RAM) "volatile"

- Memory is the **largest** storage media accessed directly by the CPU
- Memory is an array of words; each has an address.
- Word size = 2-8 byte
  - More common **4 bytes**.
- The CPU performs the following instructions:
  (a) **Load Instruction**: Fetch(get) instruction from the memory to the IR.
  (b) **Store Instruction**: Storing a Register into a memory location.



◥ Instruction Cycle



❑ Secondary Storage

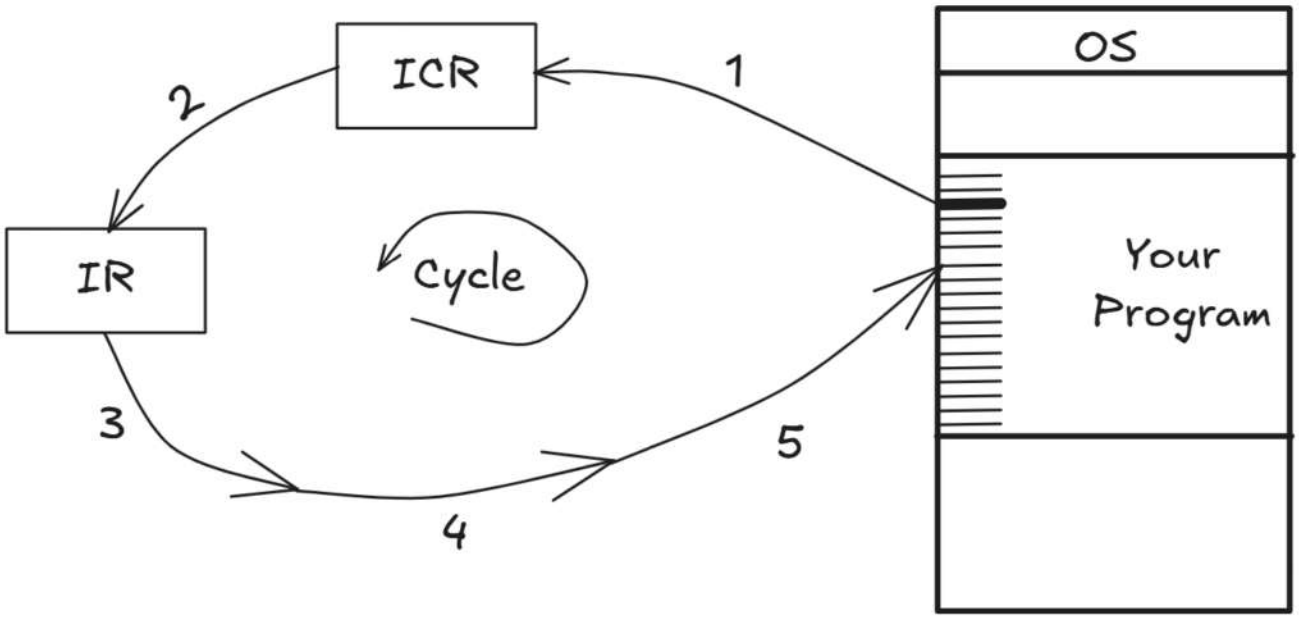
- **Such as**: Hard Disks, Tapes, CDs, Flash Memory, etc.....
- Factors that affect the secondary storge:
  (1) Speed
  (2) Cost
  (3) Volatility. "مؤقت\دائم"
- The fastest storage medium in the System

◥ Cache Memory

Caching: is simply copying data to a fastest storage medium to speedup execution, and ensure good performance.

Examples:

(1) Memory (RAM) is considered as cache for HD.
(2) Registers are considered as cache for Memory.
(3) Instruction Cache Register (ICR).



⚠ 1 & 2 are concurrent (parallel).

- from ICR to IR is much faster than RAM to IR

- L1 Cache is the fastest cache type because it's built with in the CPU
- L2 Bigger but slower
  .
  .
  etc

❑ Hardware Protection

How does the OS protect the Computer Resources (CPU, Memory, I/O Devices)?

⬔ Dual mode of operation:

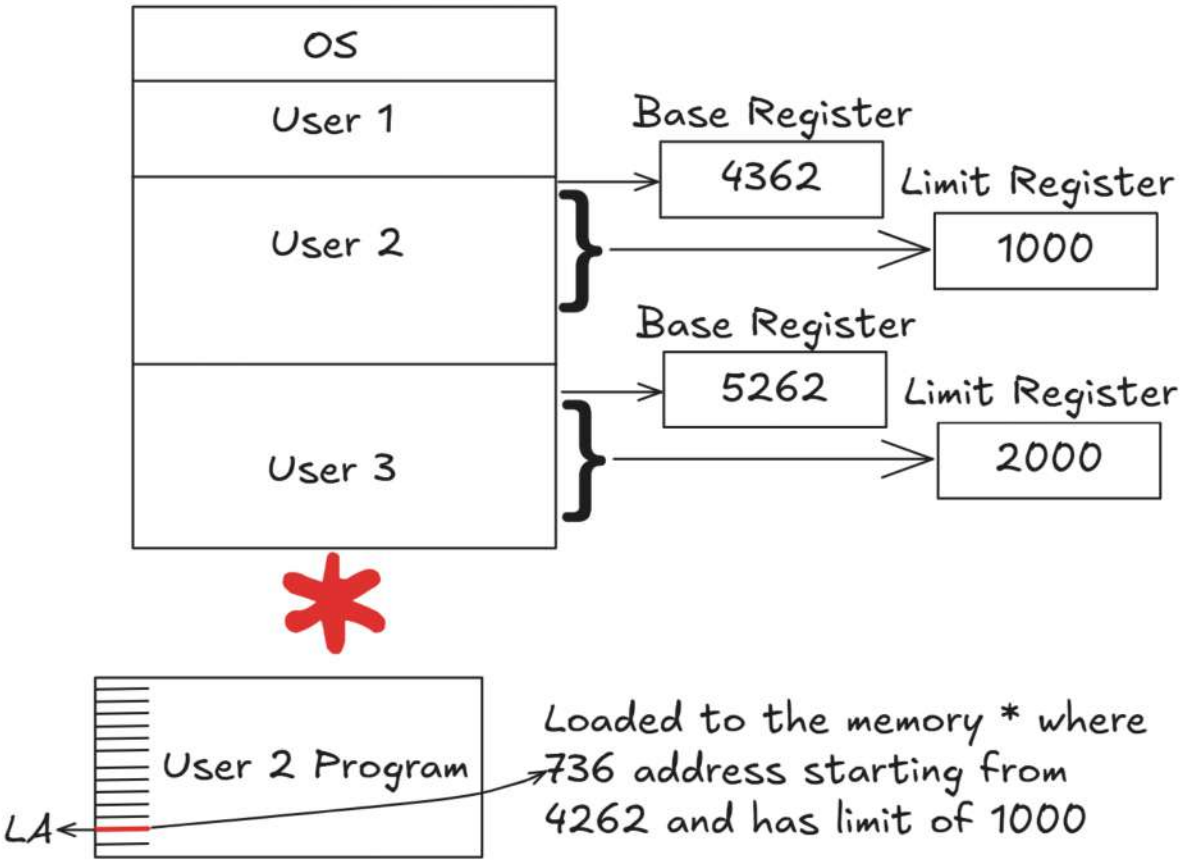
(a) **Monitor (Supervisor) Mode**: In this case the OS execute process on behalf of itself. (i.e. interrupts).
(b) **User Mode**: The OS execute on behalf of the user (it run user programs).

⬔ **Implementation**: One bit is assigned called "mode bit". (1: user mode, 0: monitor mode)

⬔ **Definition**: A **privileged instruction** is an instruction that's execute only in monitor mode.

⬔ **I/O Protection**: All I/O instructions are mode **privileged instruction**.

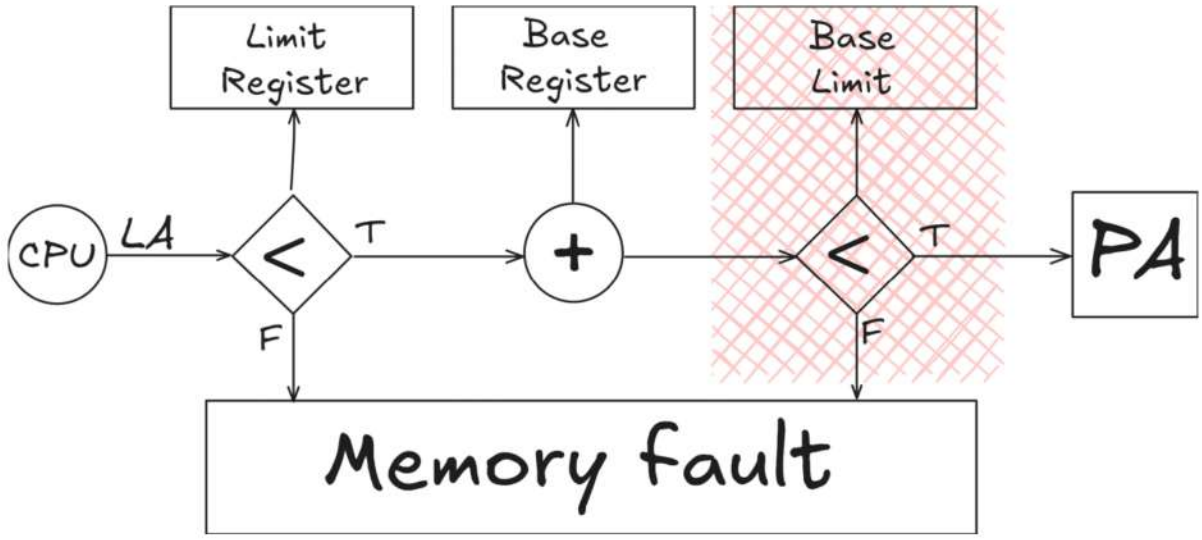
⬔ **Memory Protection**: To protect the memory allocation space of the user program & the OS itself.



* Logical Address (LA): is the offset of the instruction in your program & LA = Address seen in your program.

* Physical Address (PA): is the actual address in memory & PA = LA + Base Register
(i.e.) PA = 736 + 4262 = 4998

➔ **The most important concept; is How the OS compute the PA?**



⬕ **CPU Protection**

- Timer
- With every clock the timer is decremented by 1
- When the timer reaches zero it sends an interrupt to the CPU
- When the CPU receives interruption, it executes the interruption service routine which is responsible for checking the CPU
- Timer can be used in Computer time calculation

⬕ **Operating System Structure**

(1) Process Management
(2) Memory Management
(3) File System (I/O) Management

➔ **The OS services can be provided in two methods:**

(1) System Call: it's an assembly language instruction.
(2) System Programs.

Chapter 3 # Processes

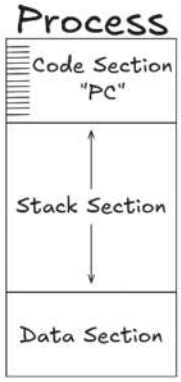
Process Definition: A process is a program in execution. [process ≡ program ≡ job].

❑ Process Types

(1) **Batch Process**: Generally, Batch processes have low priority.
(2) **Time Sharing Process**:
- Users
- Program Development
- Data Entry
- Gaming

(3) **System Tasks**: such as Interrupt

❑ Process Control

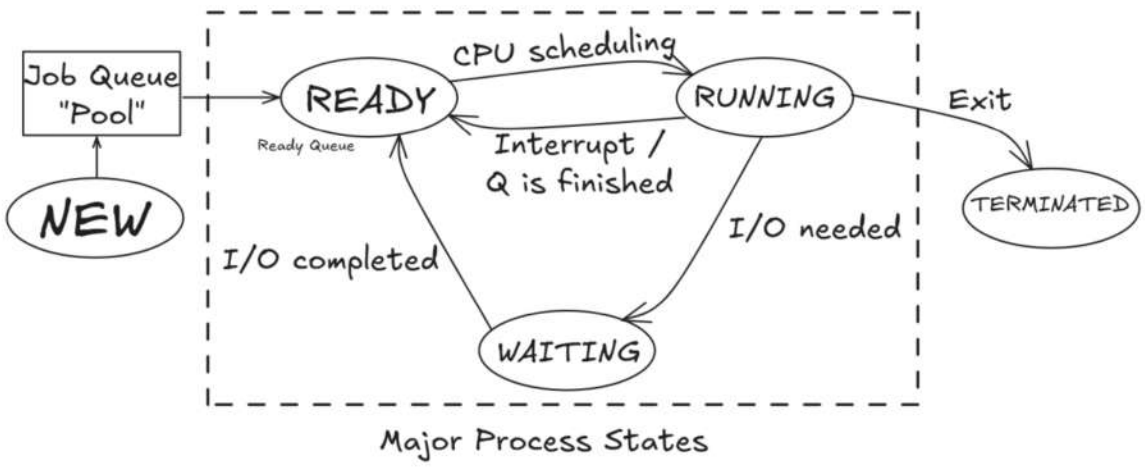


The process contains:

(1) **Code Section** (Program Counter PC)
"PC": is the address of the instruction executed currently.
(2) **Data Section**: Memory allocation were the input & output data is stored.
(3) **Stack Section**: Global variable – static

❑ Process States

(1) **NEW**: the process is newly created
(2) **READY**: the process is loaded into the memory & ready to run and have the CPU.
The process is in **READY** queue waiting to have the CPU to run.
(3) **RUNNING**: The process is having the CPU & execution.
(4) **WAITING**: waiting for some event to happen. Typically, waiting for I/O.
(5) **TERMINATED**: The process finishes execution.

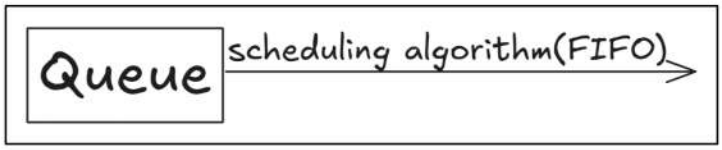


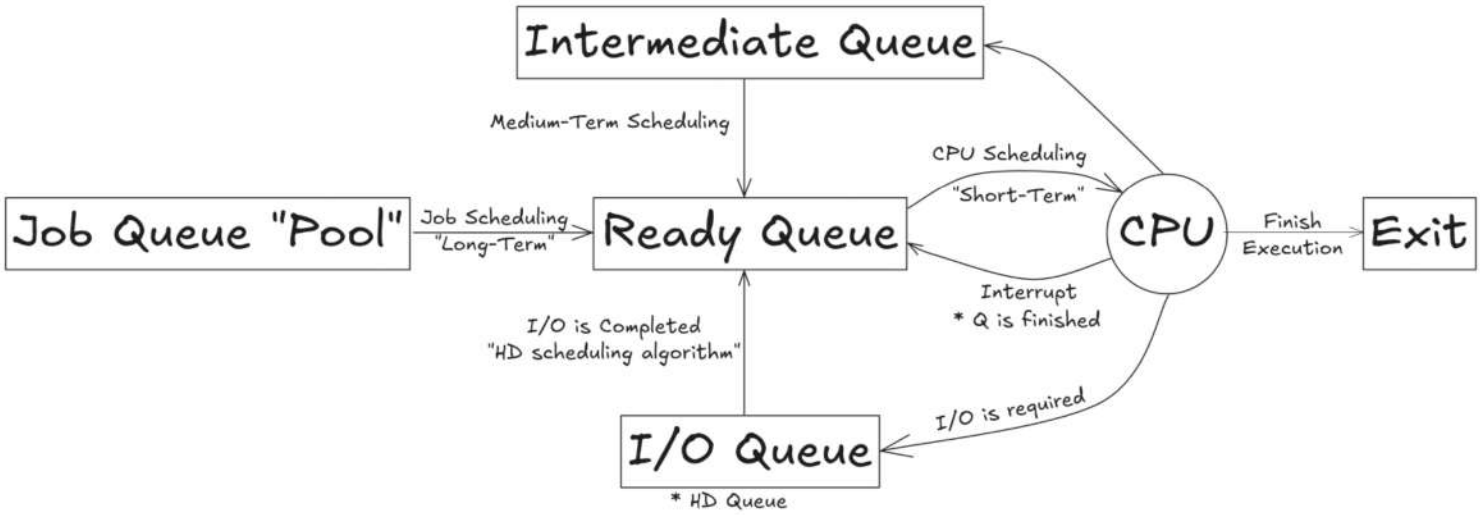
Major Process States

⊙ **Where data about the process is stored?**

The **p**rocess **c**ontrol **b**lock (**PCB**): it's a kind of datastructure, generally it's a table.

❑ Scheduling & System Queues





⬔ Long-Term "Job" scheduling

It's selecting a job from the job queue to be admitted to memory and in term it's added to the READY queue. This selection "**Interrupt**" is invoked(called), this invoked (seconds, minutes). That is, the OS has enough time to decide carefully which job to fetch.

⬔ Short-Term scheduling

It's selecting a process(job) from the REDAY queue to be given to the CPU to RUN.

(Milliseconds, Microseconds, or Nanoseconds), therefore it should be very fast.

➔ **Definition of** Degree of multiprogramming**:** is the # of jobs in the memory "READY Queue". Therefore, Long-Term scheduling controls the degree of multiprogramming.

⬔ Job Scheduling

- If most jobs in memory are **CPU-bound** jobs
  → the CPU will always be busy "High CPU Utilization", But I/O queues are empty.
- If most jobs in memory are **IO-bound** jobs
  → the I/O queues are always full and CPU almost free "Low CPU Utilization".

⚠ In both cases the system **unbalanced**!

- Long-Term Scheduling: selects a mix of CPU & I/O bound jobs, So the system will be reasonably balanced.

⬔ Process Creation

Parent process creates children processes, which in return create other processes.
Forming a **Tree of Processes**

⬔ Resource Sharing: -

- Parent & children share all resources.
- Children share a subset of the parent's resources.
- Parent & children share no resources. "Both compete for all resources".

⬔ Execution: -

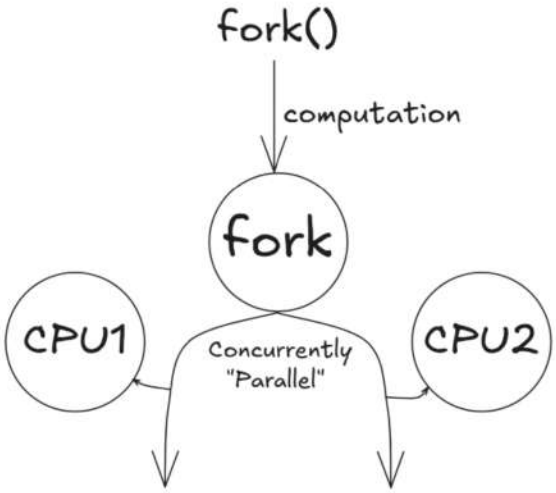
- Parent & children execute concurrently.
- Parent waits until the child terminates.

⬔ Address Space: -

- Child duplicate of parent.
- Child has a program loaded into it.

⬔ Unix examples: -

- **fork** system call creates new process.
- **execve** system used after a fork to replace the process's memory space.



⬔ Process Termination

- Process execute last statement & asks the OS to delete it "exit"
  1. Output data from child to parent 'via fork'
  2. Process's resources are deallocated by OS

⚠ When a process is killed all of its resources are deallocated.

- Parent may terminate execution of children processes 'Abort', when?
  1. Task assigned to child is no longer required.
  2. Child has exceeded allocated resources.
  3. Parent is exiting.

OS doesn't allow child to continue if its parent terminates.

Cascading Termination: If the parent process ends 'exit' all children & sub-children are forced to exit.

⬔ Cooperating Processes

Concurrent processes are either:

1. **Independent process**: Can't affect or be affected by the execution of another process.
2. **Cooperating process**: Can affect or be affected by the execution of another process.

⬔ Advantages of process cooperation:

1. Information sharing
2. Computation speed-up
3. Modularity
4. Convenience

⬔ Producer-Consumer Problem

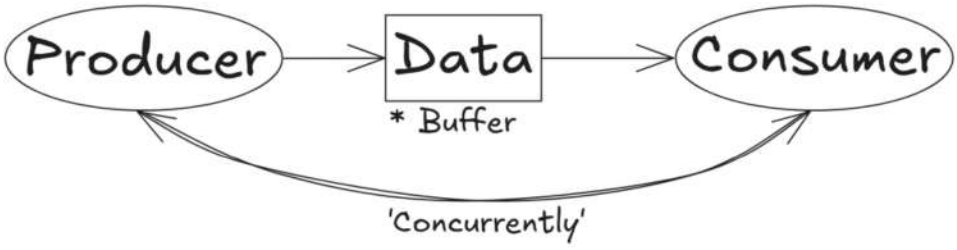
Concurrency requires:

- Cooperation among processes (communication)
- Synchronization of processes action

⊙ To illustrate the idea of cooperation process. Consider the producer-consumer problem:

Producer process produces information.

Consumer process consumes this information.



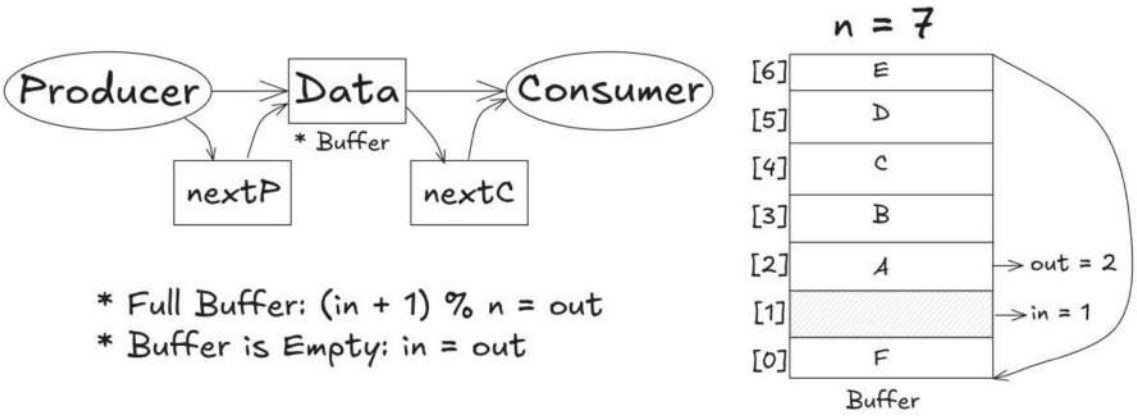
⊙ Examples

1. Print program produces characters consumed by the printer.
2. Compiler produces assembly code consumed by the assembler.
3. Assembler produces machine language code consumed by the Loader.

⊙ Datastructure Required

```
const int n; // Size of buffer
type item; // Item can be char, bit, word...
var int buffer[n];
int in; // Index where we add items in the buffer
int out; // Index where we take items from the buffer
item nextP; // Next produced item
item nextC; // Next consumed item
```

⚠ We will use the Circular Buffer idea in the implementation.



⬔ Producer Process

```
while (true) {
    // Produce an item in nextP
    // ... (implementation of producing an item)
    while ((in + 1) % n == out) {
        // No operation (wait for buffer space)
    }
    buffer[in] = nextP;
    in = (in + 1) % n;
}
```

⬔ Consumer Process

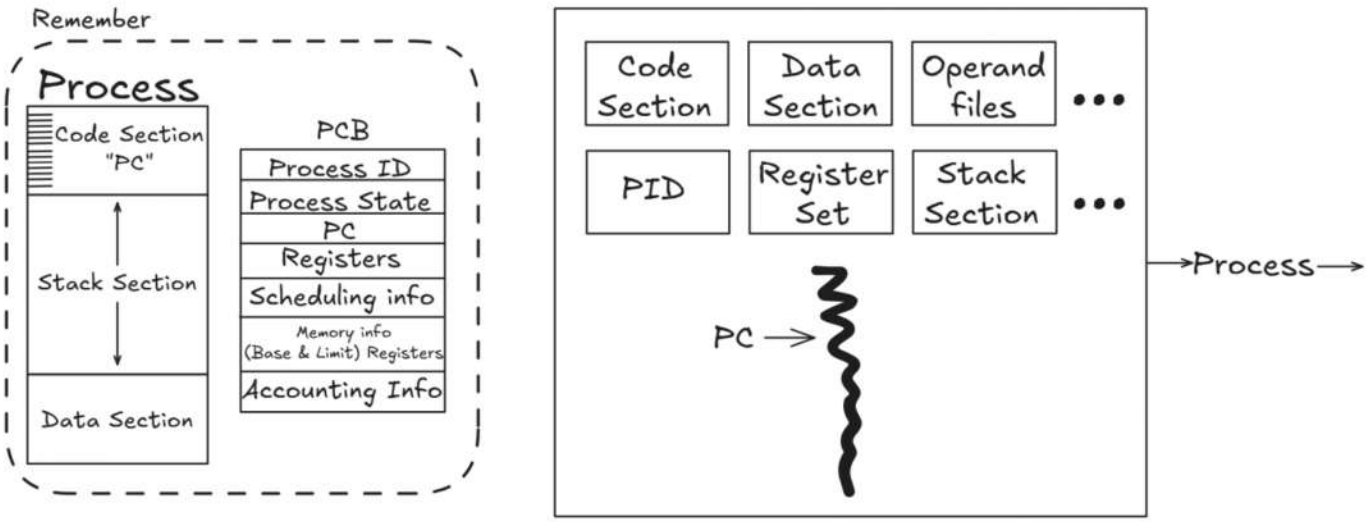
```
while (true) {
    while (in == out) {
        // No operation (busy waiting)
    }
    nextC = buffer[out];
    out = (out + 1) % n;
    // Consume the item
    // ... (implementation of consuming the item)
}
```

⊛ Producer & Consumer processes run concurrently.

⚠ Disadvantages: we only use (n-1) buffer if we have n buffers.

Chapter 4 # Threads

Process (Heavily Weight Process)

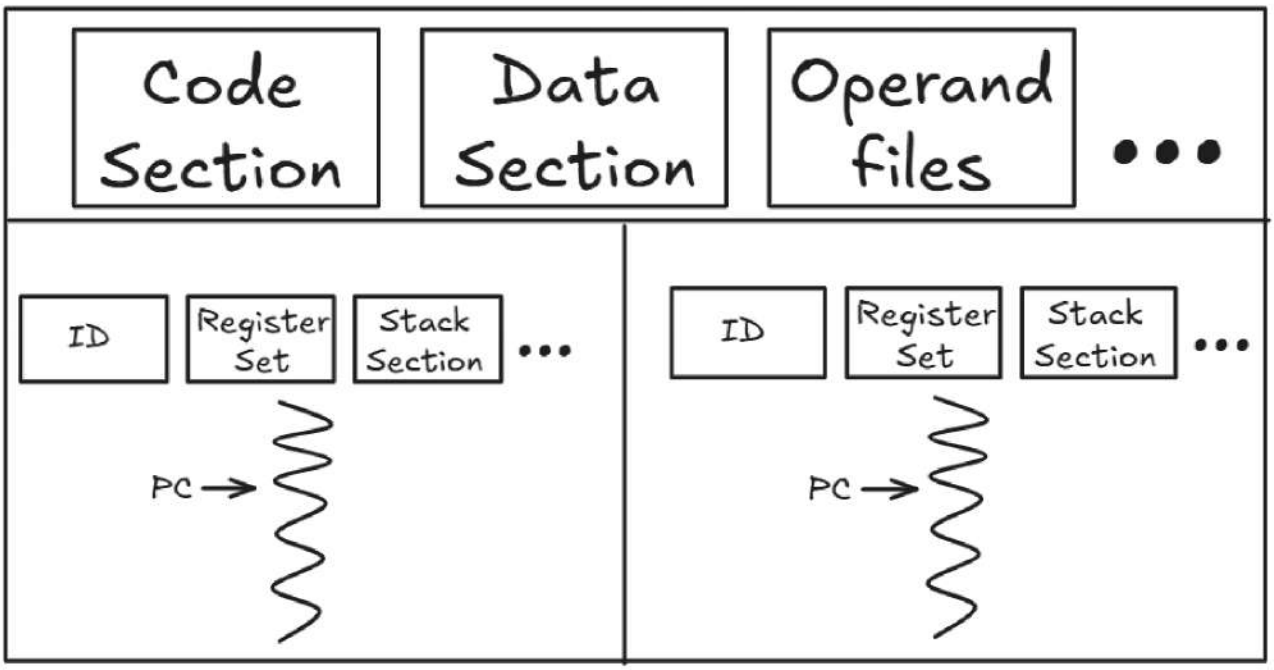


Thread (Light Weight Process) Contains: -

- Program Counter
- Register Set
- Stack Section

➔ All peer threads share: -

- Code Section
- Data Section
- I/O Resources



⊛ Advantages of Threads is: **Concurrency** & **Sharing Resources**

❑ Two kinds of Threads support

1. **At User Level:** It's totally the responsibility of the user (very complex & very difficult).
2. **At kernel Level:** Most OS support this kind of threads.

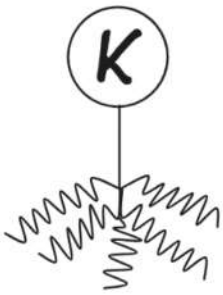
❑ User & Kernel Threads Relationship

➔ Think about user threads as processes(programs)
➔ Think about kernel threads as CPUs

⊛ We have: multi-threads & multi-CPUs

(1) Many-to-One

Many **user threads** are mapped to one **kernel thread**



⚠ Disadvantage: No Concurrent Execution

(2) One-to-One

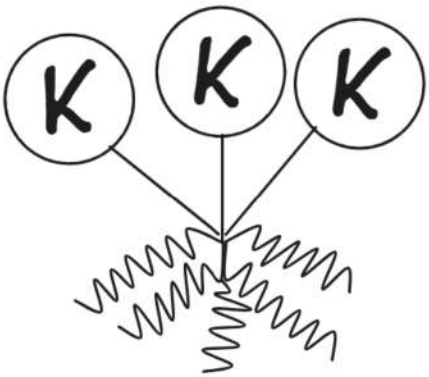
Each **kernel** thread is assigned to one **user** thread

∴ **Main advantage**: this allows concurrent execution

⚠ Disadvantage: we need enough kernel threads

(3) Many-to-Many

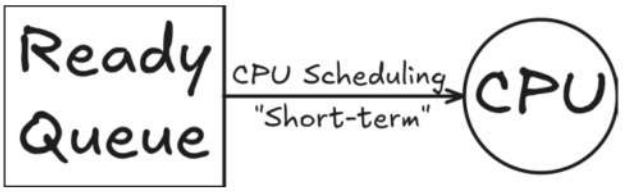
Many user threads are mapped to an equal or a smaller number of kernel threads.



Operating Systems

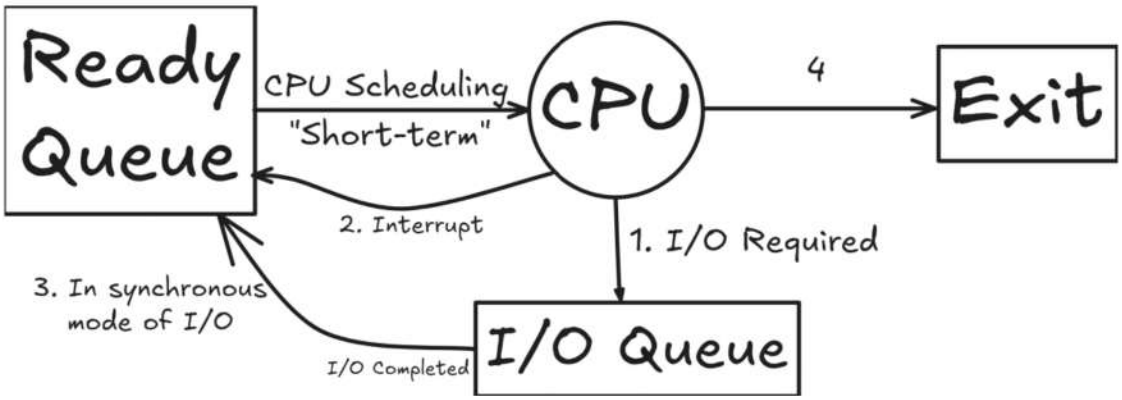
Chapter 5 # CPU Scheduling

CPU Scheduling: Is the process or decision at which process the OS should select from the ready queue and give to the CPU to execute **[short-term scheduling]**



❑ Cases to invoke the CPU Scheduling?

1. I/O required
2. Interrupt
3. Process I/O is Completed → (**In synchronous mode**)
4. Process terminated



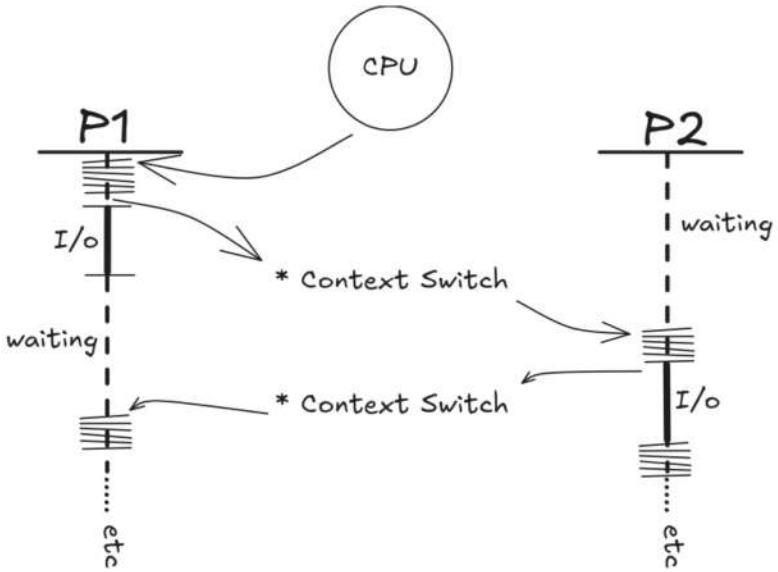
In cases:

- 1 & 4: None-preemptive "CPU تخلت بطوعية عن"
- 2 & 3: Preemptive "CPU أخذت بعنوة ال"

➔ Our Objective is to introduce all scheduling algorithms, such that we take in consideration the following criteria:
(1) CPU utilization **(max)**
(2) Throughput **(max)**
(3) Turnaround time **(min)**: It's the time from submitting job until it finishes execution
(4) Waiting Time **(min)**: It's the time the process spends in the READY queue
(5) Response Time: It's the time from submitting job until you see the first response from the computer

$$Weighted\ Turnaround\ Time = \frac{Turnaround\ Time\ (Minimum\ is\ better)}{Service\ (CPU) Time}$$

➔ Every switching from processes P_i to P_j needs: **2 Context switch**

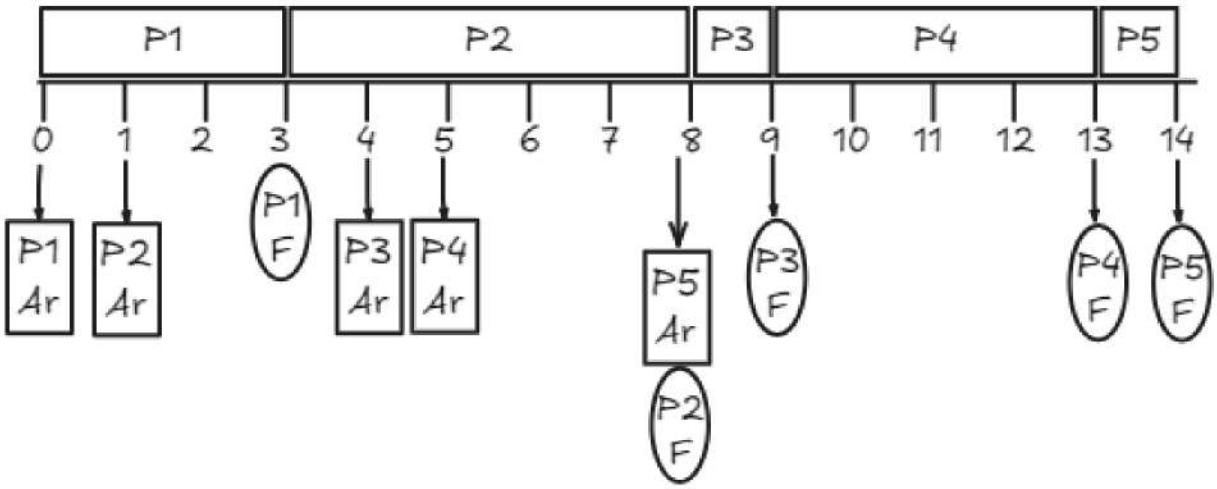


[1] FCFS → First Come First Serve

Example: Given the following READY Queue

Process	Arrival Time	Service Time (CPU Brust)
P1	0	3
P2	2	5
P3	4	1
P4	5	4
P5	8	1

Compute the average turnaround time & average waiting time. → We use Gantt Diagram

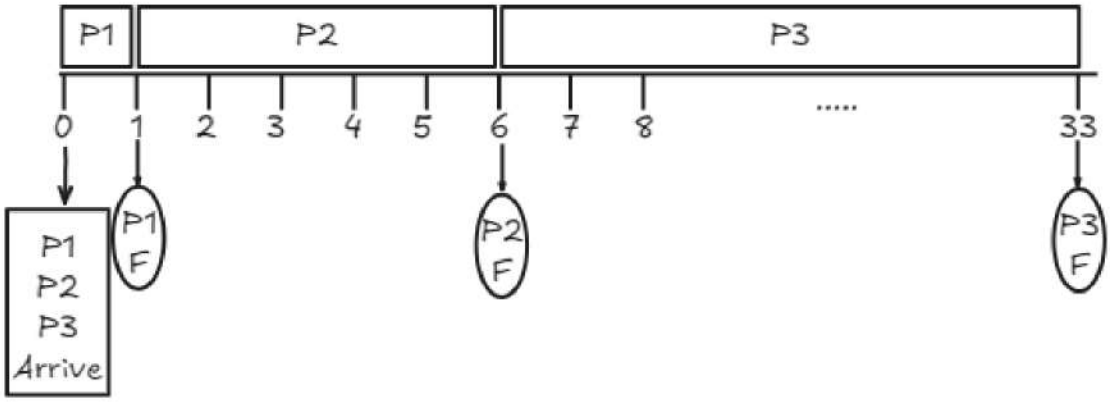


- $Turnaround = Finish\ Time - Arrival\ Time$
- $Waiting\ Time = Turnaround\ Time - Service\ (CPU)\ Time$
- $Avg\ Turnaround\ Time = \frac{(3-0)+(8-2)+(9-4)+(13-5)+(14-8)}{5} = 5.6\ unit$
- $Avg\ Waiting\ Time = \frac{(3-0-3)+(8-2-5)+(9-4-1)+(13-5-4)+(14-8-1)}{5} = 2.8\ unit$

FCFS: Convy Problem

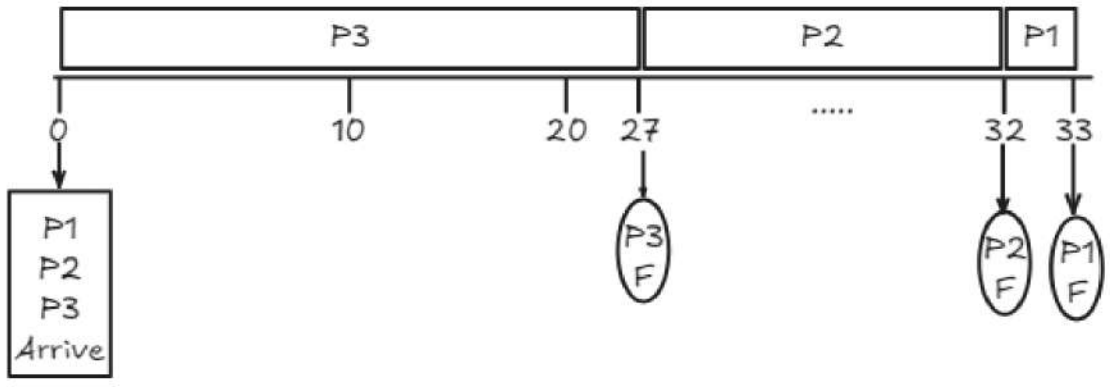
Examples: مشكلة وصول موكب

Process	Arrival Time	Service Time (CPU Brust)
P1	0	1
P2	0	5
P3	0	27



- $ATT = \frac{(1-0)+(6-0)+(33-0)}{3} = 13.\overline{33}\ unit$
- $AWT = \frac{(1-0-1)+(6-0-5)+(33-0-27)}{3} = 2.\overline{33}\ unit$

Process	Arrival Time	Service Time (CPU Brust)
P3	0	27
P2	0	5
P1	0	1

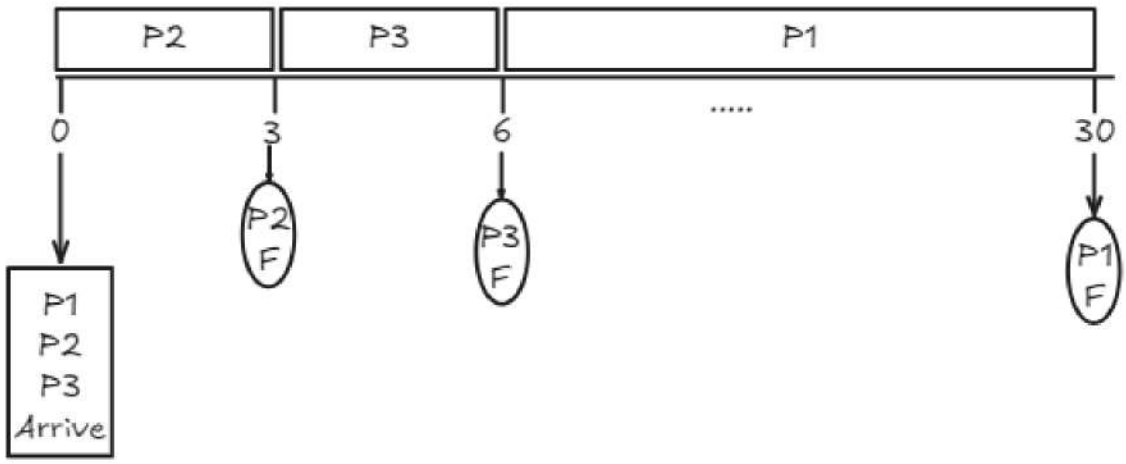


- $ATT = \frac{(27-0)+(32-0)+(33-0)}{3} = 30.\overline{66}\ unit$
- $AWT = \frac{(27-0-27)+(32-0-5)+(33-0-1)}{3} = 19.\overline{66}\ unit$

[2] Shortest Job First

The CPU is given to the process with the smallest CPU burst (Service Time). Example:

Process	Arrival Time	Service Time (CPU Brust)
P1	0	24
P2	0	3
P3	0	3



- $ATT = \frac{(3-0)+(6-0)+(30-0)}{3} = 13\ unit$
- $AWT = \frac{(3-0-3)+(6-0-3)+(30-0-24)}{3} = 3\ unit$

⚠ **Note**: Shortest job first gives the minimum (optimal) solution, That it gives the minimum waiting time.

Two versions of SJF

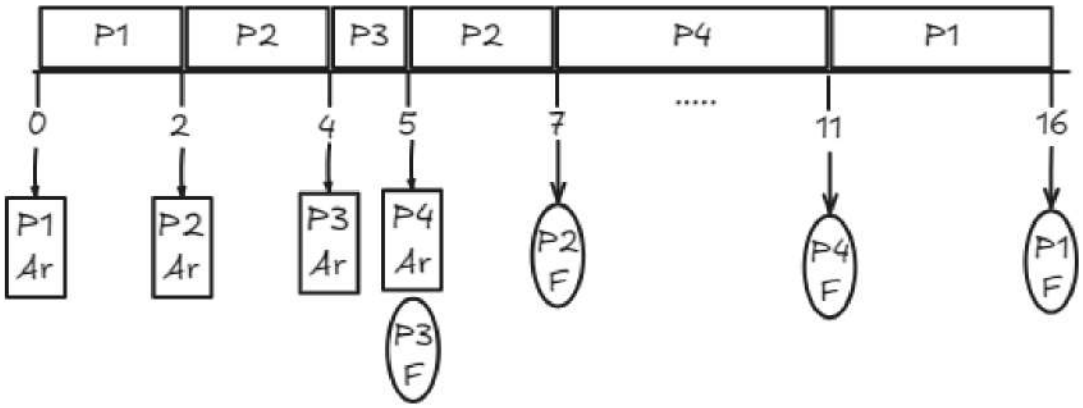
(1) **Preemptive**: if the job arrives at the READY queue with CPU burst less than the remaining of the running process, then the CPU switches to the new arriving process.
Shortest Remaining Time First (SRTF)

(2) **None- Preemptive**: if a job arrives at the READY queue with CPU burst less than the remaining of the running process, then the CPU continues with running process & then switches to the new arriving process.

Example:

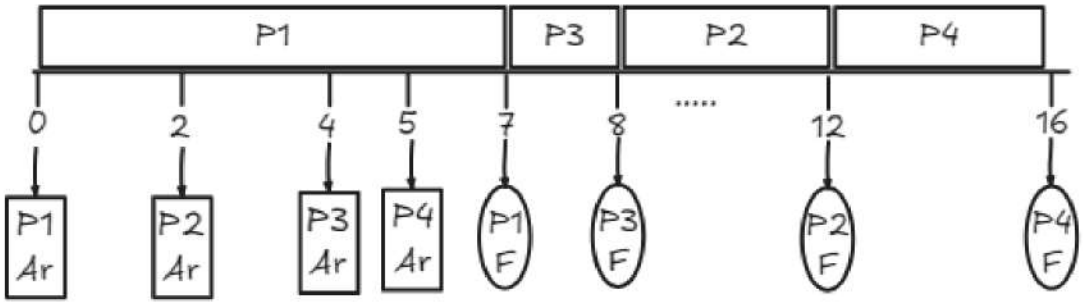
Process	Arrival Time	Service Time (CPU Brust)
P1	0	7
P2	2	4
P3	4	1
P4	5	4

(a) Preemptive



- $ATT = \frac{(16-0)+(7-2)+(5-4)+(11-5)}{4} = 7\ unit$
- $AWT = \frac{(16-0-7)+(7-2-4)+(5-4-1)+(11-5-4)}{4} = 3\ unit$

(b) None-Preemptive



- $ATT = \frac{(7-0)+(12-2)+(8-4)+(16-5)}{4} = 8\ unit$
- $AWT = \frac{(7-0-7)+(12-2-4)+(8-4-1)+(16-5-4)}{4} = \frac{18}{4}\ unit$

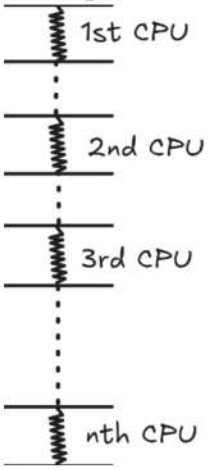
⚠ Problem: Starvation, Solution: Aging

➔ Aging: as time progress, give the process some priority.

⚠ Major Problem: How the OS can decide the length of the next CPU burst (service time)??!

∴ *Solution: the OS can only estimate the length of the next CPU burst.*

Example: Assume



- T_n = Actual length of n^{th} CPU burst
- Y_n = Estimated length of the n^{th} CPU burst
- Take a constant $0 \le w \le 1$
- Define the formula:

$Y_{n+1} = w \times T_n + Y_n \times (1 - w)$

→ $w = 0 \rightarrow Y_{n+1} = Y_n$

→ $w = 1 \rightarrow Y_{n+1} = T_n$

→ Let us expand the equation:

$$Y_{n+1} = w * T_n + (1-w) * [w * T_{n-1} + (1-w) * [w * T_{n-2} + (1-w) * [\ldots$$

$$Y_{n+1} = w * T_n + (1-w) * T_{n-1} + w * (1-w)^2 * T_{n-2} + w * (1-w)^3 * T_{n-3} + \cdots$$

$$Substitute\ w = \frac{1}{2}$$

$$Y_{n+1} = \frac{T_n}{2} + \frac{T_{n-1}}{2^2} + \frac{T_{n-2}}{2^3} + \frac{T_{n-3}}{2^4} + \frac{T_{n-4}}{2^5}$$

[3] Priority

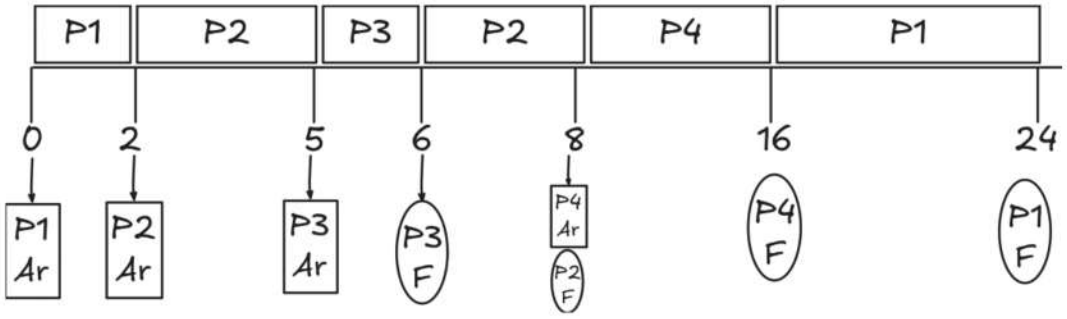
- The CPU is given to the process with the high priority.
- Every process is given a priority number. Generally, low number means low priority
  → System tasks have high priority (i.e. Interrupts)
- There are Two versions of priority:
  (1) **Preemptive**
  (2) **None-preemptive**

Example: Given the queue as follows:

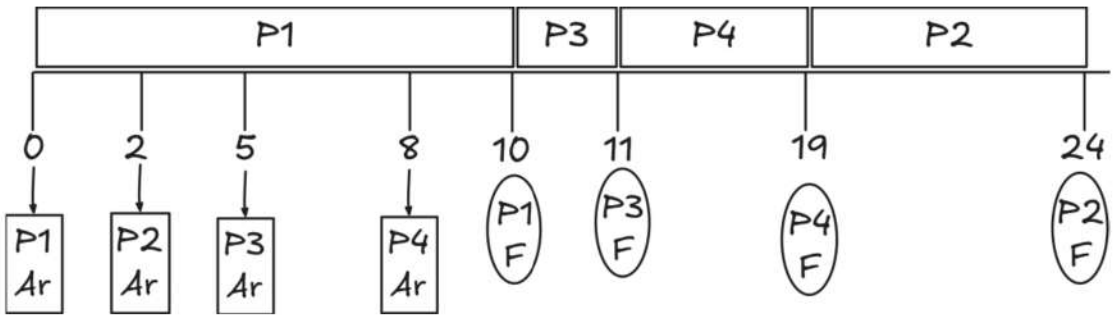
Process	CPU Brust	Priority	Arrival Time
P1	10	1	10:00
P2	5	2	10:02
P3	1	5	10:05
P4	8	4	10:08

⚠ High # = High Priority

(a) Preemptive



(b) None-Preemptive



⚠ Problem: Starvation, Solution: Aging

➔ Aging: as time progress, give the process some priority.

[4] Round Robin (RR)

- It's best designed for time sharing interaction systems
- Each process is assigned a slice of time called Quantum Q. The process runs for this Q & CPU switches to another process on FCFS basis.

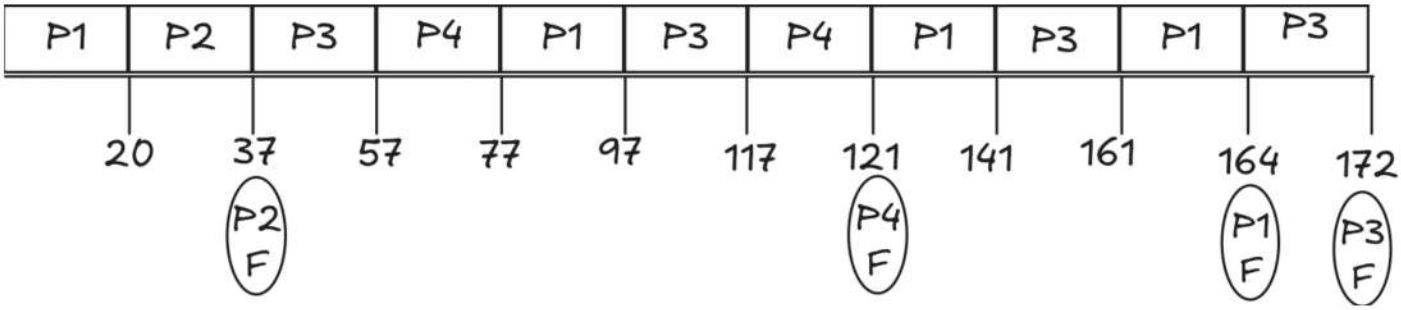
  IF

  → $Q_{very\ big}$ → FCFS

  → $Q_{very\ small}$ → ?!

Example: **Q = 20**

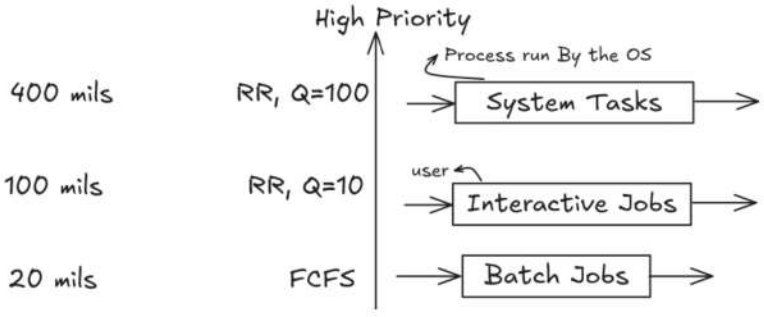
Process	Service Time
P1	53
P2	17
P3	68
P4	24



[5] Multi-Level Queues

- Ready queue is divided into several queues.
- Each queue has its own scheduling algorithm.
- Scheduling between queues, that is How to distribute CPU time among the queues?!
  There are two Algorithms:
  (1) **Time slice**: Each queue is assigned a slice of CPU time, which scheduled among its process.
  (2) **Fixed priority**:
  - Serve all jobs in "System tasks"
  - Then serve all jobs in "interactive"
  - Then serve all jobs in "Batch"

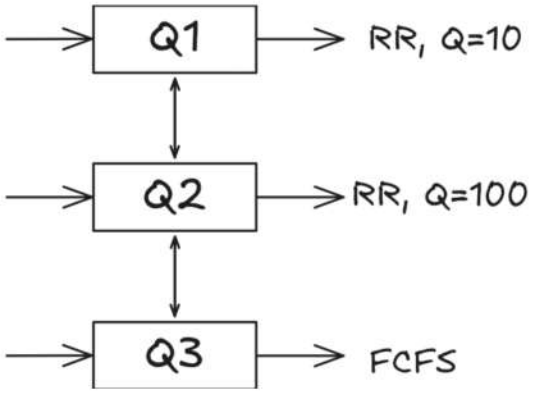
  → Preemptive & None-Preemptive



⚠ Problem: Starvation

[6] Multi-Level Feedback Queues

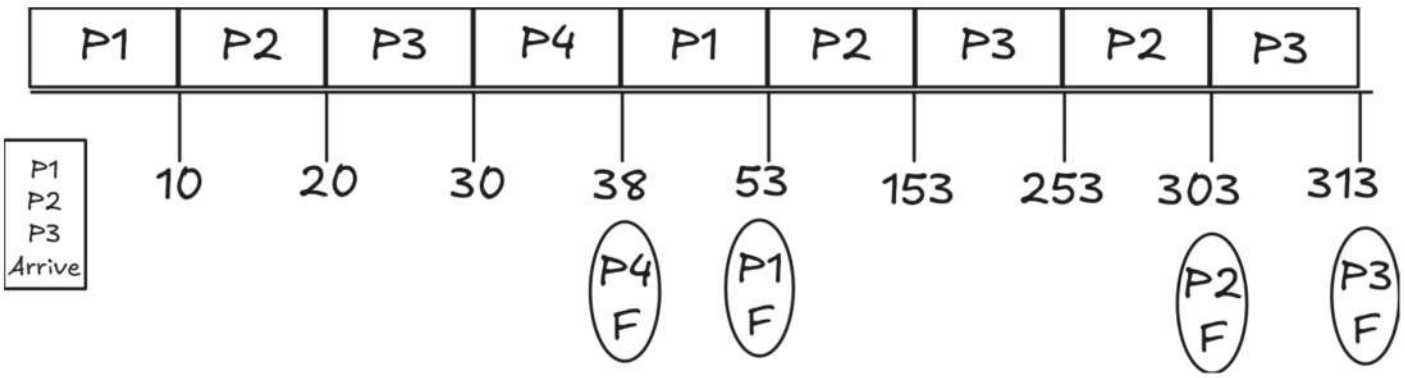
- Ready Queue is divided into several queues.
- The process can move up & down between queues.
- Preemptive & None-Preemptive



New Process enters Q1

Example:

Process	CPU-Burst
P1	25
P2	160
P3	120
P4	8



❑ Algorithm Evaluation

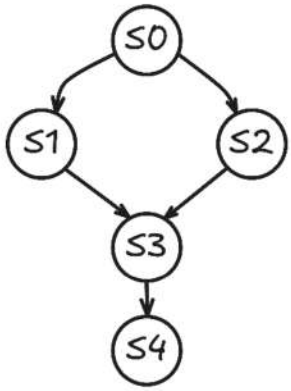
(1) Deterministic mode "poor"
(2) Queuing Theory "Theoretical"
(3) Simulation "good"
(4) Implementation "best algorithm for evaluation"

Chapter 6 # Concurrent Processes and Process Synchronization

Concurrent Processes

- Concurrent process and either independent or cooperating
- Independent process: can't affect or be affected by the processors

Precedence Graph



Given the following statements:

(1) a = x + y (s1)
(2) b = z + 1 (s2)
(3) c = a – b (s3)
(4) w = c + 1 (s4)

Clearly,

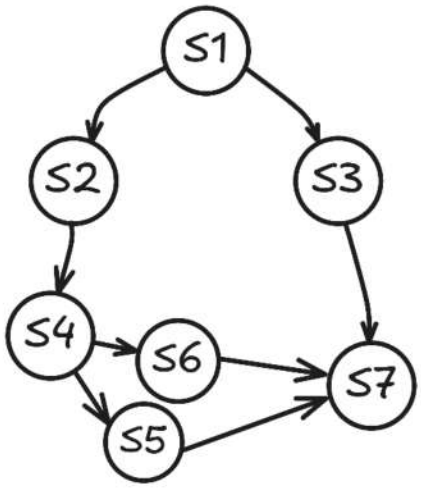
- Statmetns
  - (3) & {(1) & (2)} can't execute concurrentley
  - (4) & (3) can't execute concurrentley
  - (4) & {(1) or (2) or (3)} can't execute concurrentley
- But statements (1) & (2) can be executed concurrentley
- So if we have multiple functional units in our CPU such as address or we have multiprocesser system then the statements (1) & (2) can be executed concurrentley (in parallel).

Definition: A **precedence graph** is a directed graph whose nodes correspond to statements.

An edge form node S_i to node S_j means that S_j is only executed after S_i.

In the given graph:

- S_2 & S_3 can be executed only after S_1 completes
- S_4 can be executéd only after S_2 completes.
- S_5 & S_6 can be executed only after S_4 completes.
- S_7 can be executed only after S_5, S_6, S_3 completes.
- S_3 can be executed concurrently with S_2 S_4 S_5 S_6.



Concurrency Condition

- How do we know if two statements can be executed concurrently & produce the same result?
- **Define**:
  - $R(S_i) = \{a_1, a_2, ..., a_m\}$ be the **READ** set for statement S_i, which is the set of all variables whose values are **referenced** by statement S_i during execution.
  - $W(S_i) = \{b_1, b_2, ..., b_n\}$ be the **WRITE** set for statement S_i, which is the set of all variables whose values are **changed(written)** by the execution of statement S_i.
- **Examples**: Given the statements:

  - **S**: c = a – b
    R(S) = {a, b}
    W(S) = {c}
  - **S**: w = c + 1
    R(S) = {c}
    W(S) = {w}
  - **S**: x = x + 2
    R(S) = {x}
    W(S) = {x}
  - **S**: read(a) **or** scanf(a)
    R(S) = {a}
    W(S) = {a}
  - **S**: print(a) **or** printf(a)
    R(S) = {a}
    W(S) = {} = **Φ**

The Bernstein's conditions for concurrent statements are:

Given the statements S1 & S2, then S1 & S2 can be executed concurrently if:

[$R(S1) \cap W(S2) = \Phi$**] AND [**$W(S1) \cap R(S2) = \Phi$**] AND [**$W(S1) \cap W(S2) = \Phi$**]**

Example 1: Given

S1: a = x + y	R(S1) = {x, y}	$\{x, y\} \cap \{b\} = \Phi$
S2: b = z + 1	W(S1) = {a}	$\{z\} \cap \{a\} = \Phi$
	R(S2) = {z}	$\{a\} \cap \{b\} = \Phi$
	W(S2) = {b}	

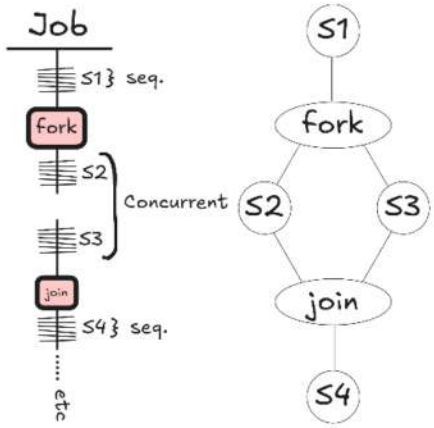
Example 2: Given S3: c = a – b → $R(S3) \cap W(S2) = \{a, b\} \cap \{b\} \neq \Phi$

Fork & Join Constructs

- Precedence graph is difficult to use in Programming Languages, so other means must be provided to specify precedence relation.
- The **Fork L** instruction produces two concurrent executions
  - One starts at statement labeled **L**
  - Other, the continuation of the statement following the fork instruction

Example: the programming segment corresponds to the precedence graph is:

```
S1;
Fork L;
S2;
...
L: S3;
...
```



(*) When the fork L statement is executed, a new computation is started at S_3 which is executed concurrently with the old computation, which continues at S_2. That is, the fork statement splits one single computation into two independent computations; hence the name Fork.

- The join instruction recombines two concurrent computations. Each computation must ask to be joined.
  - Since the two computations executes at different speeds, the statement which executes first the join first is terminated while the second is allowed to continue.
  - For 3 computations, two are terminated while the third continues.
- If **count** is number of computations to join, then the execution of the join has the effect

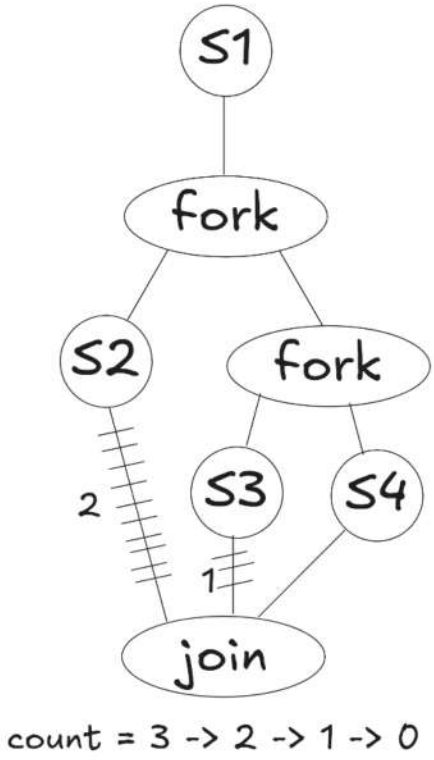
  count = count – 1

  if count ≠ 0 then quit (quit this computation)

The join statement for two computation is executed **atomically.** Can't be executed concurrently but in sequential manner, because this might affect **count** giving a wrong result.

For Example: if both decrement count at the same time, then count = 0, and then computation dues not quit.

```
1. count = # of computations to join
2. function join () {
3. …
4. …
5. count = count - 1
6. if (count != 0) Quit (stop) computation;
7. }
```



- For two processes:

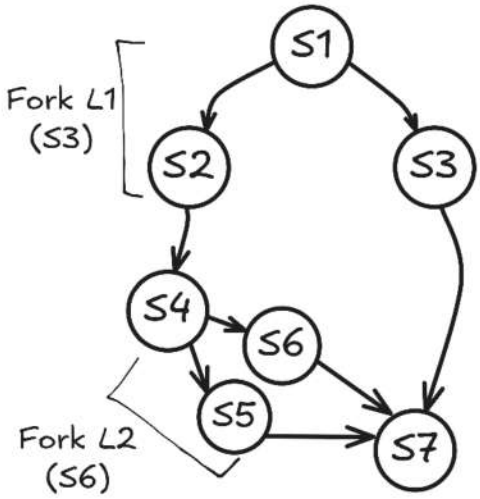
```
1. count = 2
2. Fork L1;
3. …
4. …
5. S1;
6. goto L1;
7. L1: S2
8. L2: join count
```

- Let us go back to our four statements in the beginning of this chapter. Using fork & join:

```
1. count = 2
2. Fork L1;
3. a = x + y;
4. goto L2;
5. L1: b = z+1;
6. L2: join count;
7. c = a - b
8. w = c + 1
```

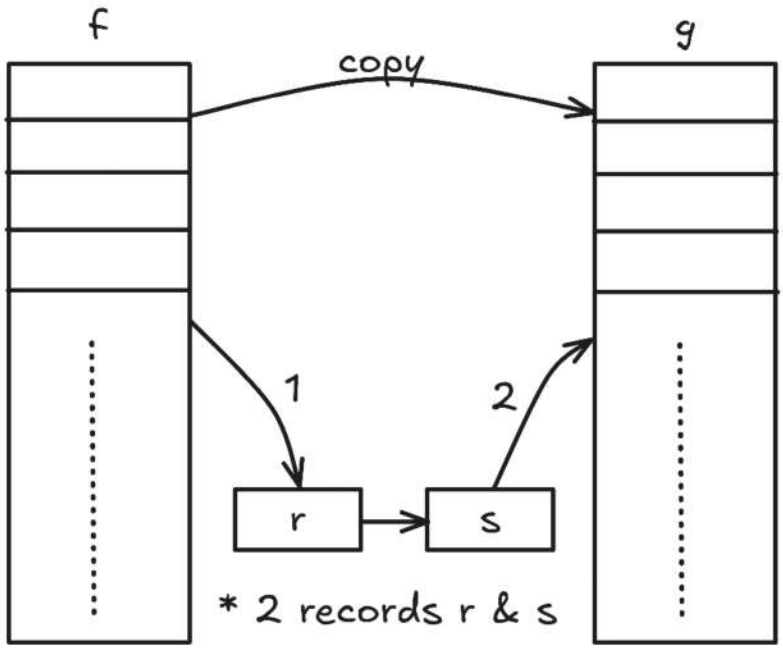
- For precedence graph earlier:

```
  S1;
  count = 3
  Fork L1;
  S2;
  S4;
  Fork L2;
  S5;
  goto L3;
L2: S6;
  goto L3;
L1: S3;
L3: join count;
  S7;
```



- Another example is to copy a sequential file **f** to **g** using double buffers **r** & **s.**
- The program can read from **f** & write to **g** concurrently.

```
T = some-record-type
f, g: file of T;
r, s: T
Begin
  reset(f)
  read(f, r)
  while (not eof(f)) do begin
    count = 2
    s := r
    Fork L1
    write(g, s)
    goto L2
  L1: read(f, r)
  L2: join count
  End
  write(g, r)
End
```

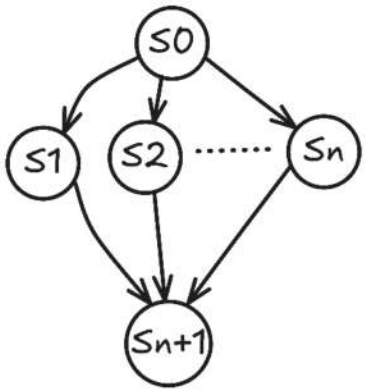


The concurrent statement

- The fork & join instructions are powerful means of writing concurrent programs, unfortunately, it is clumsy and very difficult to keep track, because the fork is similar to goto statements.
- A higher-level language constructs for specifying concurrency due to Dijkstra using the notations parbegin / parend

Example:

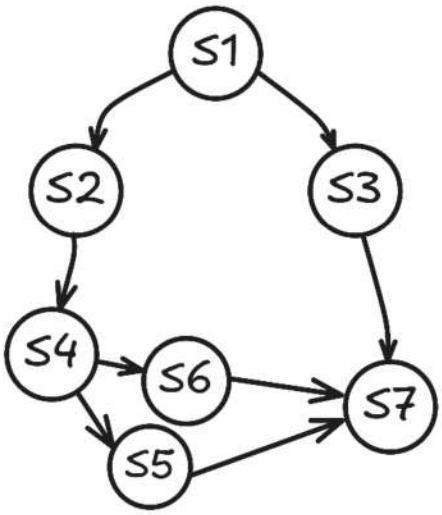
```
S0;
Parbegin
  S1;
  S2;
  …
  Sn;
Parend;
Sn+1
```



- All statements enclosed between parbegin and parend can be executed concurrently.

For precedence graph earlier:

```
S1;
parbegin
  S3;
  begin
    S2;
    S4;
    parbegin
      S5;
      S6;
    parend;
  end
parend;
S7;
```



For the files copying example:

```
begin
  reset(f)
  read(f, r)
  while (not eof(f)) do begin
    s = r
    parbegin
      write(g, s);
      read(f, r);
    parend;
  end;
  write(g, r);
end;
```

Process Synchronization

Background

Process Cooperation

- Information Sharing
- Computation Speedup
- Modularity
- Convenience

Example: Producer-Consumer problem, the bounded buffer problem:

Data Structure used:

```
item …; // Any Data Type
item buffer[n], nextP, nextC;
int in = 0, out = 0;
```

Producer:

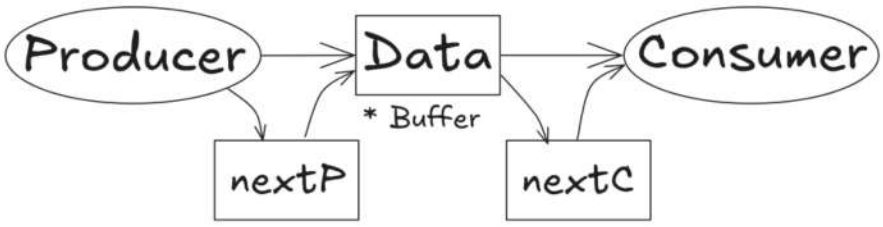
```
do {
      …
      Produce an item in nextP
      …
      while ((in + 1) % n == out)
            no-op; // Full Buffer
      buffer[in] = nextP;
      in = ((in + 1) % n
} while true;
```

Consumer:

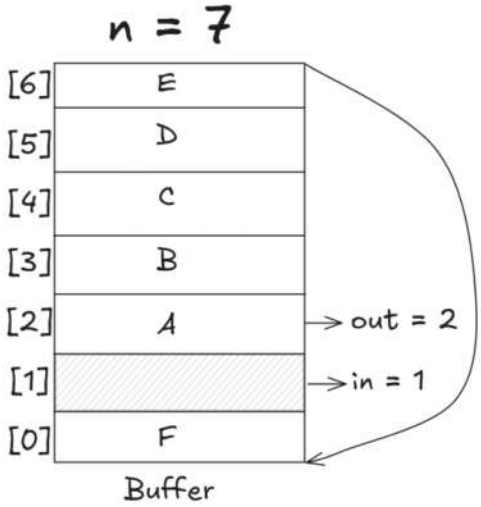
```
do {
      while (in == out)
            no-op; // Empty buffer
      nextC = buffer[out];
      out = (out+ 1) % n
      …
      Consume an item in nextC
      …
} while true;
```

- Shared memory solution to bounded buffer problem discussed before allows at most n - 1 items in buffer at the same time.
- Suppose that we modify the producer consumer code by adding a variable **counter**, initialized to 0 and incremented each time a new item is added to the buffer, and decremented each time an item is taken from the buffer.

Remember:



* Full Buffer: (in + 1) % n = out
* Buffer is Empty: in = out



Bounded Buffer

Data Structure used:

```
item …; // Any Data Type
item buffer[n], nextP, nextC;
int in = 0, out = 0;
int counter = 0;
```

Producer:

```
do {
     …
     Produce an item in nextP
     …
     while (counter == n)
          no-op; // Full Buffer
     buffer[in] = nextP;
     in = ((in + 1) % n;

     counter = counter + 1; *
} while true;
```

Consumer:

```
do {
     while (counter == 0)
          no-op; // Empty buffer
     nextC = buffer[out];

     out = (out+ 1) % n;

     counter = counter - 1; *
     …
     Consume an item in nextC
     …
} while true;
```

* is a **critical section**

Producer **critical section** implementation:

```
register1 = counter;
register1 = register1 + 1;
counter = register1;
```

Consumer **critical section** implementation:

```
register1 = counter;
register1 = register1 - 1;
counter = register1;
```

* Both Producer & Consumer run **Concurrently.**

Consider this execution interleaving:

S0: producer execute register1 = counter {register1 = 5}

S1: producer execute register1 = register1 + 1 {register1 = 6}

S2: consumer execute register2 = counter {register2 = 5}

S3: consumer execute register2 = register2 – 1 {register2 = 4}

S4: producer execute counter = register1 {counter = 6}

S5: consumer execute counter = register2 {counter = 4}

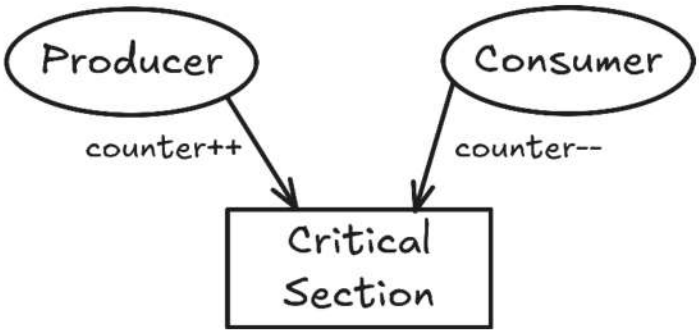
• No problems if there is a strict alternation of the **consumer** and **producer** processes.

Problems with Bounded-Buffer with Counter

- Concurrent access to shared data may result in data **inconsistency**.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

- The statements:
  - counter = counter + 1
  - counter = counter - 1

  must be executed **Atomically**.



Atomically: If one process is modifying counter the other process **must wait**, that is, as if this is executed sequentially.

The Critical Section Problem

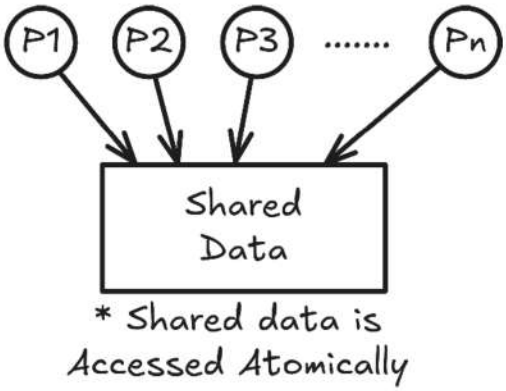
The Problem with Concurrent Execution:

- Concurrent processes (or threads) often need access to shared data and shared resource.
- If there is no controlled access to shared data, it's possible to obtain an inconsistent view of this data.
- Maintaining data consistency require Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Race Condition: A situation in where several processes access and manipulate data concurrently and the outcome of execution depends on the particular order in which the access takes place.

- **N** processes all competing to use some shared data.
- Each process has code segment called **Critical Section**, in which the shared data is accessed.
- Problem: ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process P_i:

```
repeat
     entry section
          critical section
     exit section
          remainder section
until false;
```



Solution Requirements

1. ***Mutual Exclusion***: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections. ***"One process at a time"***
2. ***Progress***: If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next **cannot be postponed indefinitely**.
   "If there are no processes in their critical section & process wants to use the critical section it can get it"
3. ***Bounded Waiting***: A bound must exist on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before the request is granted.

- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of n processes.

Solution to Critical Section Problem

Types of solutions:

- **Software Solutions**
  - Algorithms whose correctness does not rely on any assumptions other than positive processing speed (that may mean no failure).
  - Busy waiting.
- **Hardware Solutions**: Rely on some special machine instructions (System Calls).
- **Operating System Solutions**: Extending hardware solutions to provide functions and data structure support to the programmer.

Software Solution

- Only 2 processes P_0 & P_1
- General structure of process P_i (other process P_j)

```
repeat
  entry section
        critical section
  exit section
        remainder section
until false;
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

Shared variables: -

```
int turn; // turn can have value of 0, 1
// if turn = i, Pi can enter its critical section
```

Process P_i

```
do {
     // Busy waiting
     while (turn != i) no-op;
     critical section
     turn = j;
     remainder section
} while (true)
```

Process P_j

```
do {
     // Busy waiting
     while (turn != j) no-op;
     critical section
     turn = i;
     remainder section
} while (true)
```

- Mutual Exclusion ✓
- Bounded waiting ✓ - each only waits at most 1 go.
- Progress ✗ - each has to wait 1 go. P_0 gone into its (long) remainder, P_1 executes critical and finishes its (short) remainder long before P_0, but still has to wait for P_0 to finish and do critical before it can again.

⚠ Strict alternation not necessarily good - Buffer is actually pointless, since never used! Only ever use 1 space of it.

⚠ Works based on turns so concurrency isn't used.

Algorithm 2

Shared variables: -

```
boolean flag[2];
flag[0] = flag[1] = false;
// if flag[i] == true, Pi ready to enter its critical section
```

Process P_i

```
do {
     flag[i] = true;
     while (flag[j]) no-op;
     critical section
     flag[i] = false;
     remainder section
} while (true)
```

Process P_j

```
do {
     flag[j] = true;
     while (flag[i]) no-op;
     critical section
     flag[j] = false;
     remainder section
} while (true)
```

- Doesn't work at all. Both flags set to true at the start. "After you." "No, after you." "I insist." ...etc.
- Infinite loop.

Algorithm 3

Shared variables: -

```
int turn; // turn can have value of 0, 1
// if turn = i, Pi can enter its critical section
boolean flag[2];
flag[0] = flag[1] = false;
// if flag[i] == true, Pi ready to enter its critical section
```

Process P_i

```
do {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j)
no-op;
     critical section
     flag[i] = false;
     remainder section
} while (true)
```

Process P_j

```
do {
     flag[j] = true;
     turn = i;
     while (flag[i] && turn == i)
no-op;
     critical section
     flag[j] = false;
     remainder section
} while (true)
```

- Meets all three requirements; solves the critical section problem for two processes.
- "flag" maintains a truth about the world - that I am at start/end of critical. "turn" is not actually whose turn it is. It is just a variable for solving conflict if two processes are ready to go into critical. They all give up their turns so that one will win and go ahead.
- e.g. flags both true, turn = 1, turn = 0 lasts, P_0 runs into critical, P_1 waits. Eventually P_0 finishes critical, flag = false, P_1 now runs critical, even though turn is still 0.
  Doesn't matter what turn is, each can run critical so long as other flag is false. Can run at different speeds.
- If other flag is true, then other one is either in critical (in which case it will exit, you wait until then) or at start of critical (in which case, you both resolve conflict with turn).

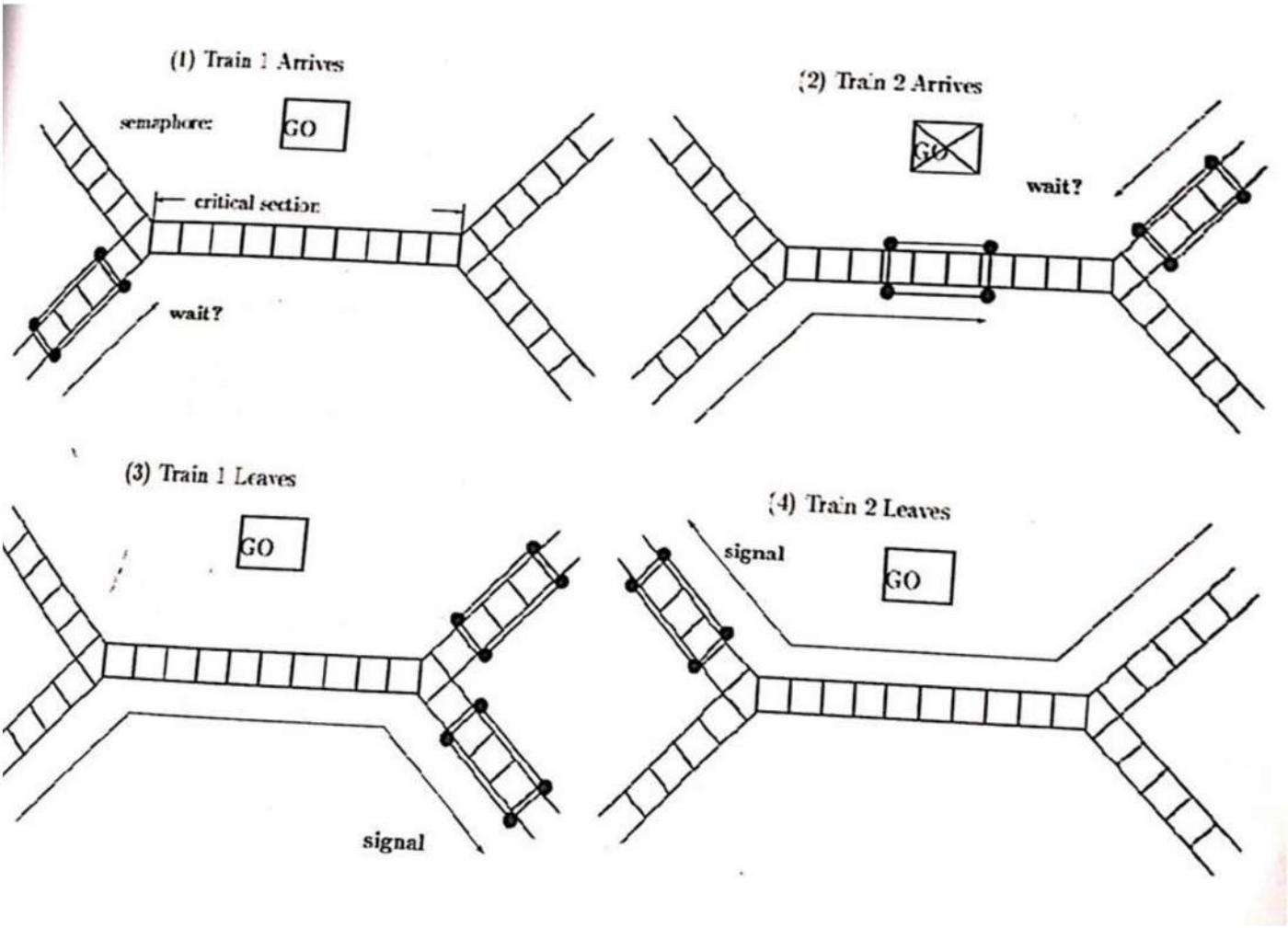
Drawbacks of Software Solutions

1. Complicated to program
2. Busy waiting (wasted CPU cycles)
3. More efficient to *block* processes that are waiting (just as if they had requested I/O)

Operating System Solution

Semaphores (wait & signal)

- Wait: checks if the critical section is empty or not.
- Signal: opens the critical section.



Semaphore S – integer variable which can only be accessed via two indivisible **atomic** operations.

Wait(S):

```
while (s <= 0) { /* do nothing */
}
s = s - 1
```

Signal(S):

```
S = S + 1
```

```
mutex: semaphore = 1;
Repeat
  wait(mutex);
    critical section
  signal(mutex);
    remainder section
Forever
```

Semaphore Implementation

- Define a semaphore as a record/structure

```
struct semaphore {
  int value;
  List *L;
}
```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(P)** resumes the execution of the blocked process P.
- Semaphore operations now defined as:

```
function wait (S) {
  S.value = S.value - 1;
  if (S.value < 0) {
    add this process to S.L;
    block;
  }
}
function signal (S) {
  S.value = S.value + 1;
  if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
  }
}
```

Classical Problems of Synchronization

1. Bounded Buffer Problem

- Shared Data

```
char item;
char buffer[n];
semaphore full = 0;
semaphore empty = n;
semaphore mutex = 1;
char nextp, nextc;
```

- Producer Process

```
do {
  produce an item in nextp
  wait(empty);//if buffer full
  wait(mutex); // counter
  add nextp to buffer;
  signal(mutex);
  signal(full);
} while (true);
```

- Consumer Process

```
do {
  wait(full);
  wait(mutex);
  remove an item from buffer
to nextc;
  signal(mutex);
  signal(empty);
  consume the item in nextc;
} while (true);
```

2. Readers-Writers Problem

- Shared Data

```
semaphore mutex = 1;
semaphore wrt = 1;
int readcount = 0;
```

- Writer Process

```
wait(wrt);
  writing is preformed
signal(wrt);
```

- Reader Process

```
wait(mutex);
readcount = readcount + 1;
if (readcount == 1) wait(wrt);
signal(mutex);
  reading is preformed
wait(mutex);
readcount = readcount - 1;
if (readcount == 0) signal(wrt);
signal(mutex);
```

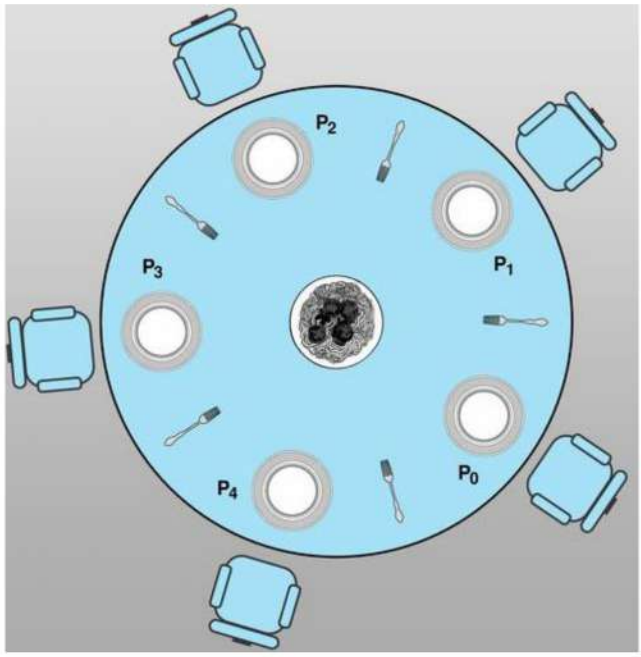
3. Dining Philosopher Problem

- Shared Data

```
semaphore chopstick[5];
chopstick[i] = 1; // 1: available
```

- Philosopher i

```
do {
  wait(chopstick[i]);
  wait(chopstick[(i + 1)% 5]);
  eat();
  signal(chopstick[i]);
  signal(chopstick[(i + 1)% 5]);
  think();
} while (true);
```

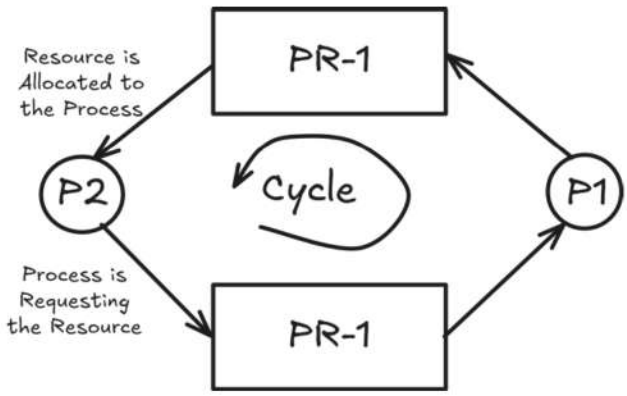


⚠ Problems:

(1) Deadlocks
(2) Starvation

Chapter 7 # Deadlocks

Definition: Two processes are deadlocked if every process is holding a resource & waiting for the other process to release its resource.



❑ Deadlock

A set of waiting (blocked processes), each process is holding a resource & waiting for other processes to release its resources.

❑ System Model

- We have the resources types R_0, R_1, ..., R_{n-1}
- We have Wi instances of each resource type. W_0, W_1, ..., W_n
- Each process uses the resources in the following order:
  - Requests the resource.
  - Uses the resource.
  - Releases the resource.

❑ Deadlock Handling

The OS handles the deadlock in one of two methods:

(1) Allow the system to enter a deadlock and then recovers from it. (**UNIX**)
(2) The OS prevents the system from entering a deadlock state.

❑ Necessary Condition

4 necessary conditions must hold simultaneously in order for the deadlock to occur.

(1) **Mutual Exclusion**: The resource type must be used exclusively that's can't be shared.
(2) **Hold & wait**: Each process is holding a resource type and waiting for the other process to release the resource of the same type.
(3) **Non Pre-emption**: Can't remove any of the resources.
(4) **Circular wait**: "Cycle"

❑ Resource Allocation Graph

$$V = \begin{cases} P: process \\ R: resource \end{cases}$$

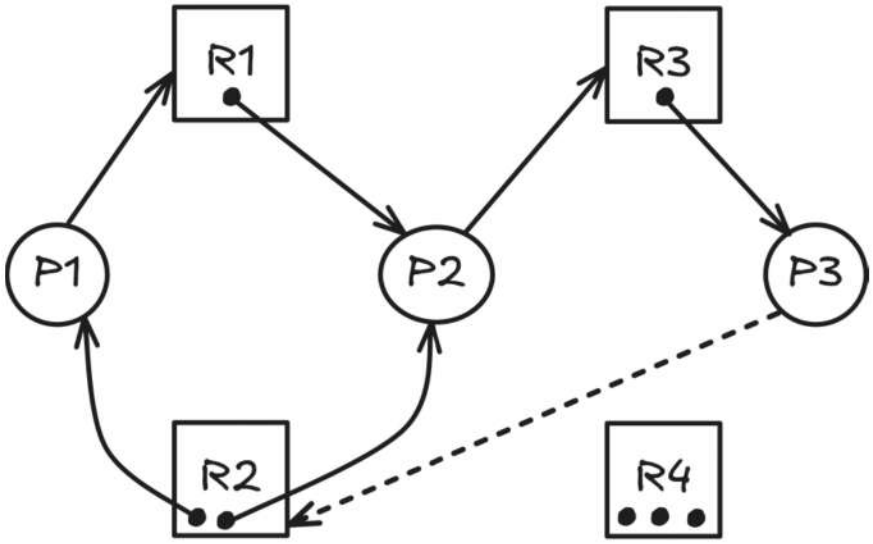
$$E = \begin{cases} (Pi, Rj): Process\ Pi\ is\ requesting\ one\ instance\ of\ resource\ Rj \\ (Rj, Pi): one\ instance\ of\ resource\ Rj\ is\ allocated\ to\ Process\ Pi \end{cases}$$

Example:

P = {P1, P2, P3}

R = {R1: 1, R2: 2, R3: 1, R4: 3}

E = {(P1, R1), (P2, R3), (R1, P2), (R2, P2), (R2, P1), (R3, P3)}



Assume P3 demands an instance of R3.

❑ Deadlock Prevention

To make sure at least one of the necessary conditions don't hold:

1- **Mutual Exclusion**: By default, some resources are mutually exclusive, and we can't do anything about it, such as printers.
2- **Hold & Wait**: To break this condition we might do:
   (i) Let the process request all its resources at the beginning.
   (ii) The process is granted all its resources when it has none.

   ⚠ Problem: Starvation

3- **Non Pre-emption**: If a process request a resource which is not available, it must releases the resources it has.
4- **Circular Wait**:
   (1) Card Reader
   (2) Hard Disk
   (3) Tape
   (4) Printer

❑ Deadlock Avoidence

Definition: a system in a safe state, if there exists a sequence of processes <P_0, P_1, ..., P_{n-1}> such that:

P_0 can take all available resources, execute, and finish.

P_1 can take all available resources, and resources released by P_0, execute, and finish.

P_2 can take all available resources, and resources released by P_0, P_1, execute, and finish.

...

P_{n-1} can take all available resources, and resources released by P_0, P_1, ..., P_{n-2}, execute, and finish.

* If there's such a sequence, then the system is safe, no deadlock.

Example: A system with 12 tapes units and 3 processes, A snapshot at the system looks like:

Process	***Max Needs***	***Allocated***	***Current Needs***
P0	10	5	5
P1	4	2	2
P2	9	~~2~~ 3	~~7~~ 6

The available at this this is: 12 – 9 = 3

- Is the system safe? Yes <P1, P0, P2>

  Available: 3 ~~+~~ **5**

  5 ~~0~~ **10**

  ~~10~~ 3 **12**

- Assume, process 2 demanded extra tape & the OS granted the request.
  The available at this this is: 12 – 10 = 2
  Is the system safe? No safe sequence (deadlock)

  Available: 2 ~~+~~ 4 <P1, ??

Example:

Process	***Allocation***			***Max Needs***			***Current Needs***		
	A	***B***	***C***	***A***	***B***	***C***	***A***	***B***	***C***
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	~~0~~ 3	~~0~~ 3	2	4	3	3	~~4~~ 1	3 0	1

- Is the system safe?
  Available:
  3 3 2 → P1
  5 3 2 → P3
  7 4 3 → P0
  7 5 3 → P2
  10 5 5 → P4
  10 5 7

- Assume process 4 requested (3, 3, 0) more, is the system safe?

  Available:
  0 0 2 → ?? (Not safe)

❑ Banker's Algorithm

Considering only one resource type for simplicity.

- Each process declares its maximum needs at the beginning.
- When a process requests a resource, it might have to wait.
- When the process gets all resources, it must release them in a finite time.

➔ Datastrucre used:

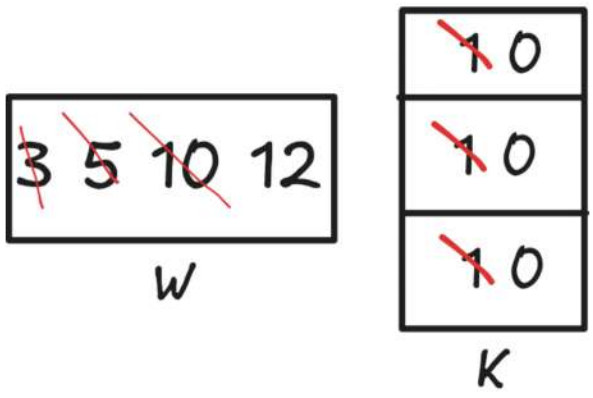
- Array max, max[i] = j; means Pi will need max of j units of the resource.
- Array allocation, allocation[i] = j; means Pi is currently allocated j units of the resource.
- Array NEED, NEED[i] = j; means Pi needs j units of the resource.
- ⚠ NEED[i] = max[i] – allocation[i];
- Available = W, W is the number of units available of the resource.

```
1. let w = available;
2. Define an array K[i] = 1 for all n
3. Find i such that:
     K[i] = 1 & NEED[i] ≤ W
     If no such i exists GoTo step 5
4. W = W + allocation[i]
    K[i] = 0
    GoTo step(3)
5. if K[i] = 0 for all n then safe system
    else not safe system
```

Example:

Process	***Max***	***Allocation***	***Needs***
P0	10	5	5
P1	4	2	2
P2	9	2	7

Available (W = 3)



Chapter 8 # Memory Management

'Ordinary memory system': means all programs must be admitted (allocated to memory before execution starts).

❑ Logical Address vs. Physical Address

* **Logical Address**: The address seen in your program, It's the offset of the address in your program.

* **Physical Address**: It's the actual address in memory.

❑ Binding Times

When the operating system determines the physical address?

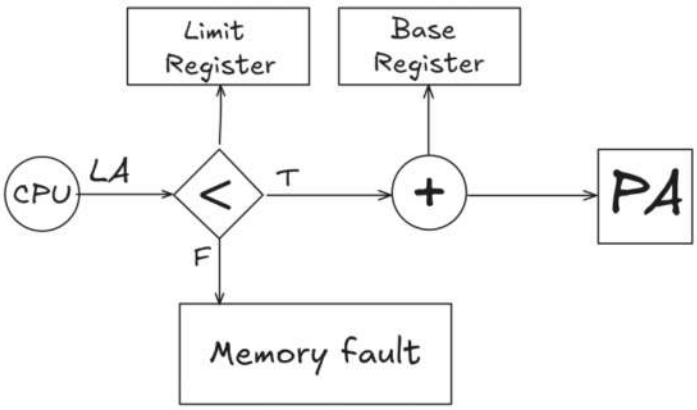
(1) **At compilation time**: The PAs are assigned at the beginning which means the program must be loaded into the memory every time at the same location. Also notice that the program can't change its location during the execution.
(2) **At loading time**: The PAs are decided when the program is loaded in to the memory.
⚠ Problem: The program can't be moved during execution
(3) **At execution time**: 'Best'

➔ **Our objective** is to compute or calculate the physical address, so it'll be available for the CPU to fetch the instruction or data.
➔ **The most important thing** is how to calculate the physical address and give it to the CPU.

[1] Contiguous Allocation

Or multiple partitions (Regions).

- Memory is divided into partitions or regions.
- Every region can hold only one process, when a region or partition becomes free, a new program is loaded in to it.
- Hardware support means: what data structure we need to compute the PA?!
- Answer: Base & Limit Registers



PA = LA + Base Register

❑ Memory Management Algorithms

(1) Fixed Regions
(2) Dynamic (variable) Regions

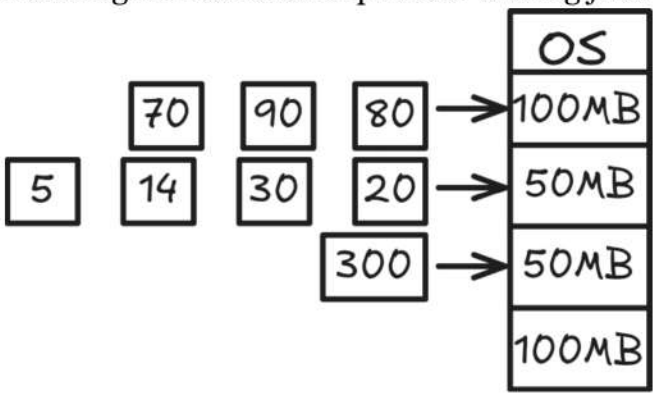
(1) Fixed Regions (IBM: MFT)

Multiprogramming with fixed number of regions.

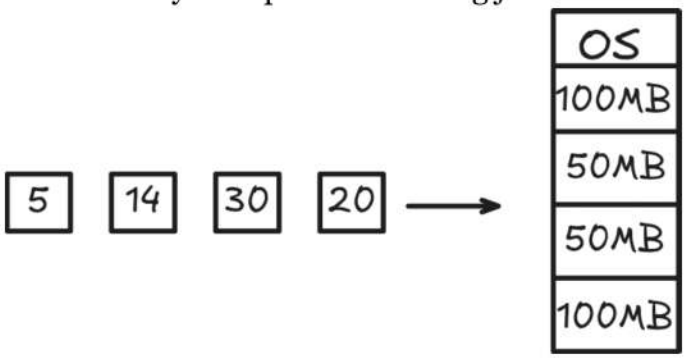
- Memory is divided in to a fixed number of regions (partitions) and sizes.
- Every region can hold one job.
- The degree of multiprogramming is bounded by the number of regions.

Job scheduling, how the OS selects a job for a certain region?!

(a) Each region has its own queue of waiting jobs



(b) There is only one queue of waiting jobs



*** Scheduling is:**
(1) FCFS with or without skip
(2) Best Fit only
(3) Best available Fit

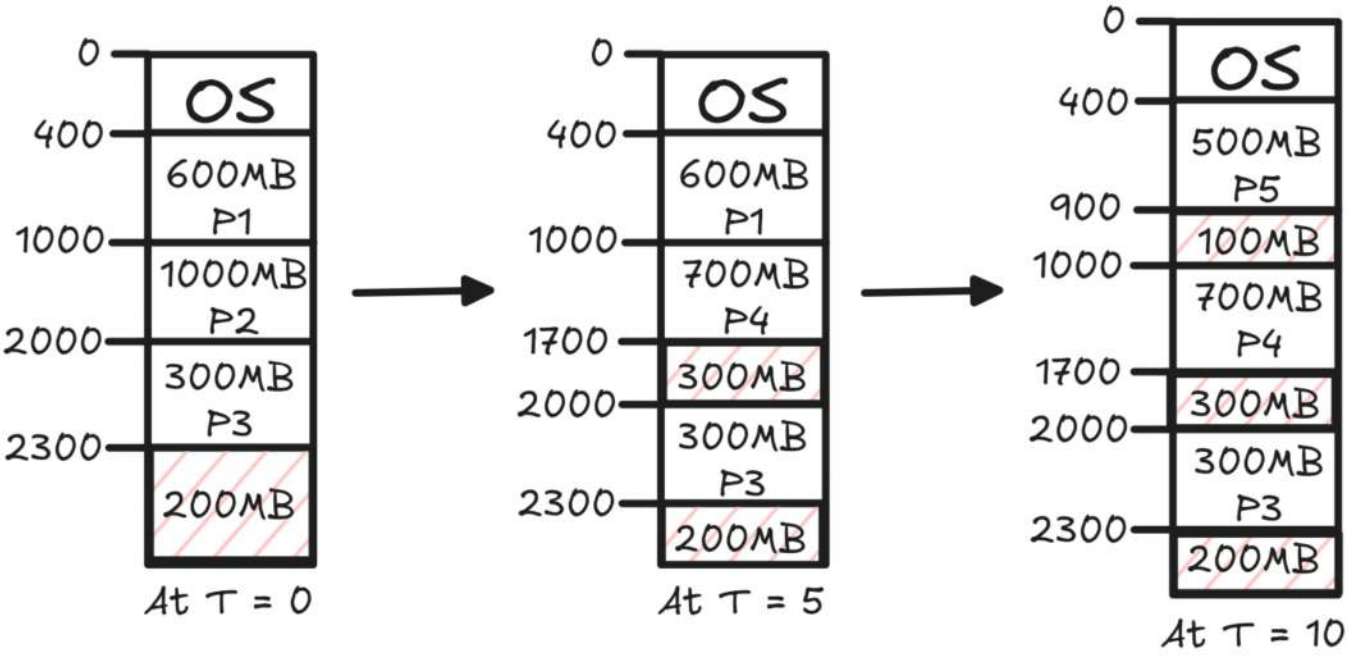
⚠ Problem:
1. Internal Fragmentation: Which is the remaining unused memory inside the region
2. External Fragmentation: The unused regions which is small to fit any available jobs

(2) Dynamic (variables) Regions

Example: Assume we found the following queue of jobs:

Assume we have memory 2500MB & the OS is reserving 400MB. 'Use FCFS'

Process	Memory Needed	Time In Memory
P1	600MB	10
P2	1000MB	5
P3	300MB	20
P4	700MB	8
P5	500MB	15



After a while memory will contain:

- Allocated regions.
- Set of holes 'External Fragmentation'

* Job scheduling: how we select a hole 'variable region' for a process.

(1) First Fit

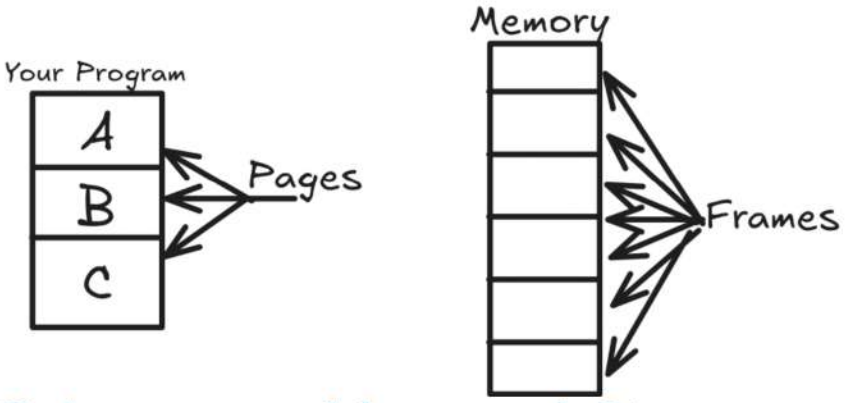
(2) Best Fit

(3) Worst Fit

⚠ Problem: External Fragmentation

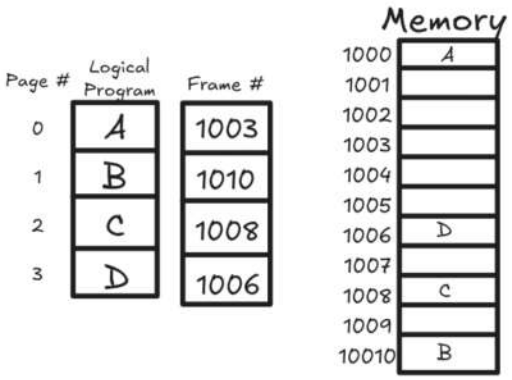
[2] Non-Contiguous / Paging

- The logical program is divided into equal size partitions called pages
- Memory is divided into partitions of the same size called frames



- Hardware support needed to compute the PA:
  We need what's called "Page Table" which is a table that contains the frame members of the pages in the logical program.

Example: Assume page size = 100 byte, given the following:

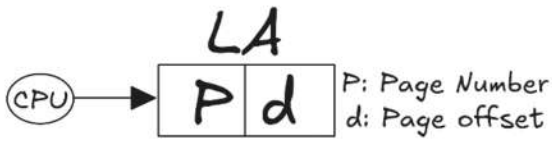


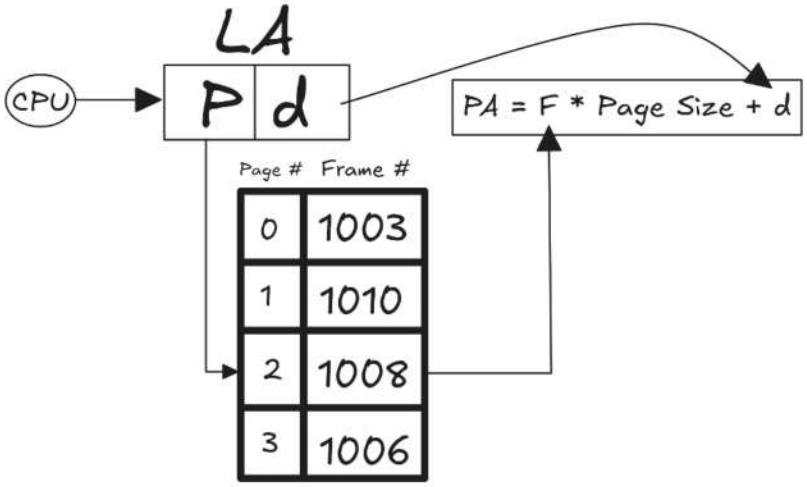
Example: Take the LA = 271 → P = 271 / 100 = 2, d = 271 % 100 = 71

In general:

P = LA / Page Size

D = LA % Page Size





- The OS must compute P & d in every instruction in your program.

- **Note**: /, % operations are multiplication operations, which means these operations take time in execution.

- Does the OS, in real life perform those operations? **NOO!**

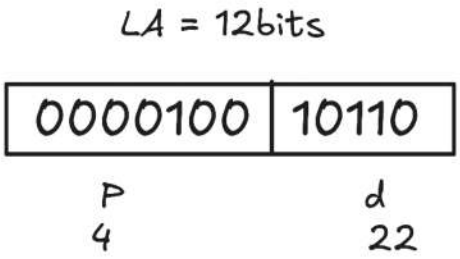
⚠ Note: the page size in practice is always 2^n bytes, generally, ***1024 ≤ page size ≤ 8192***.

Most of the time, Page Size = 4096 = 2^{12}. In this case the low order "Low significant bits" n bits of the logical address represent **d** and the remaining bits represent **P**.

Example: Assume page size = 32 bytes = 2^5,
n = 5, also assume LA = 12 bit, given LA = 150, then:

P = 150 / 32 = 4

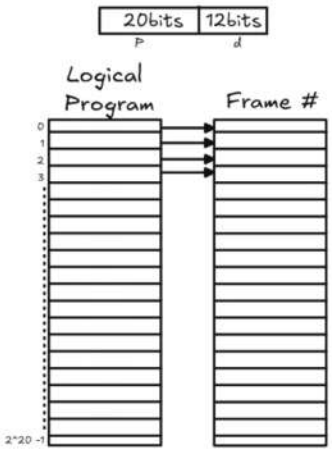
d = 150 % 32 = 22



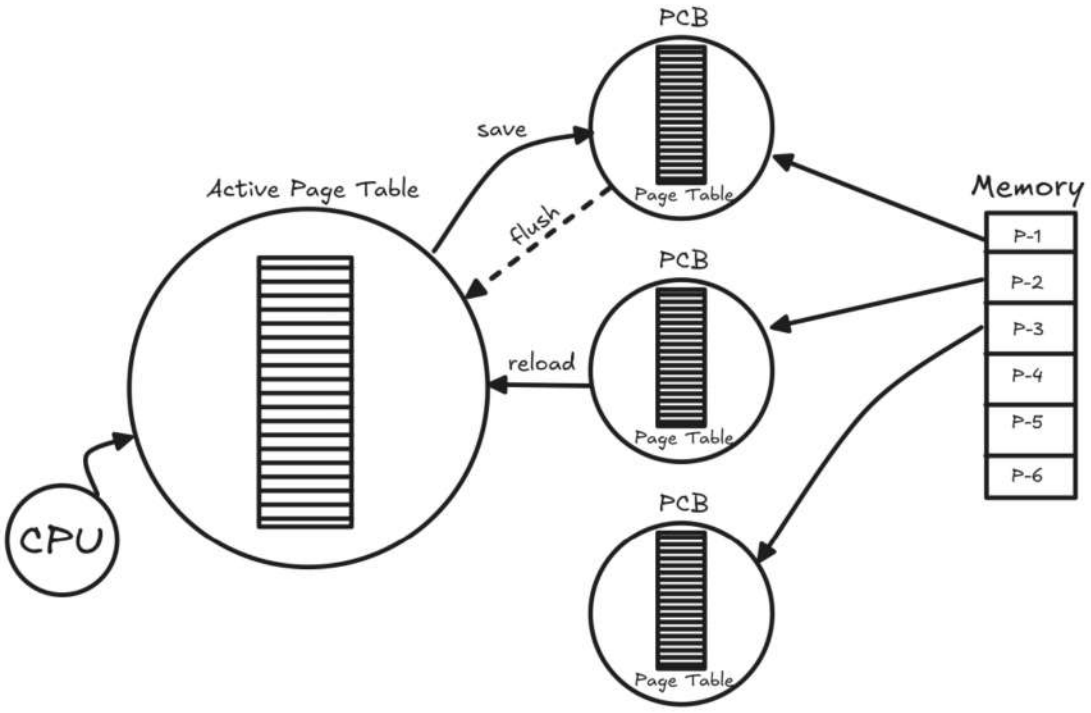
⚠ The major problem in paging is:

In the page size & where to store?

Example: Assume LA = 32bit, page size = 2^{12}



- Maximum program can be executed on this machine = 2^{20}
- Page Table Size = 2^{20} x 4 = 4MB
- If LA = 40 bits, page size 2^{12} then the page table size = $2^{(40-12)}$ x 4 = 1GB
- If LA = 48 bits, page size 2^{12}
  then the page table size = $2^{(48-12)}$ x 4 = 256GB



❑Where does the OS store the page table?!

The active page table is the page table executing.

❑ Implementation of the page table

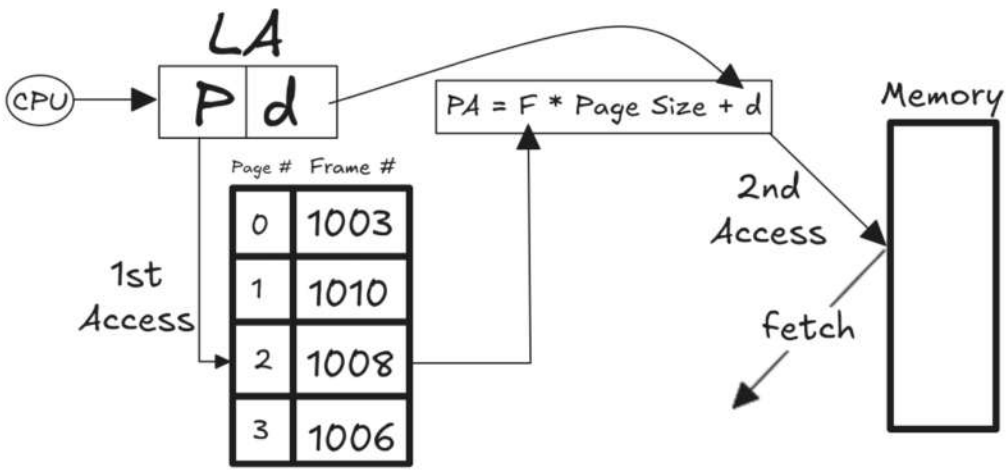
Where we store the active page table?!

(1) **Registers**: ok, this only can be implemented if the page size is small.
Example: In PDP_11, OS (minicomputer digital corporation)
LA = 16bits
Page size = 8192 = 2^{13}
* Maximum program can be executed containing only $\mathbf{2^{(LA-n)} = 2^{(16-13)} = 2^3 = 8}$ **Pages**
* **Page table size = 2^3 * Frame number size (say 4 bytes typical integer)**
= 2^3 * 4 = 32 bytes.

But, if LA = 32bits, page size = 2^{12} → size of page table = 2^{20} * 4 = 4MB

(2) **Memory**: Keep the page table in memory identified by the Page Table Base Register (PTBR) & Page Table Limit Register (PTLR)

⚠ Problem: In this case we need two memory accesses, one to access the page table & the other to access the instruction.



(3) **Memory + Registers**
- A small number of registers called 'Associative Register' or 'translation look aside buffers' are assigned or dedicated for a small page table whose entry contains (Page #, Frame #)
- Performance of Associative Registers depends on Hit Ratio (h)
- Hit Ratio: probability that the desired page in the associative registers.

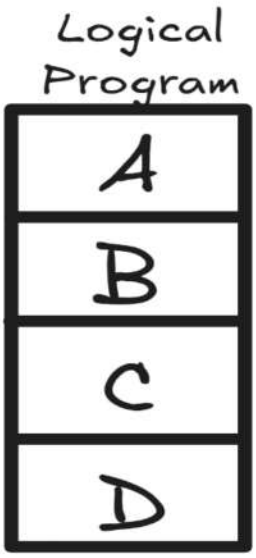
Example: Memory access (m) = **100nans**, search time in associative register (t) = **1nans**, Hit Ratio (h) = **0.95**.
→ **Effective Access Time (EAT) = $h(m+t) + (1-h)(2m+t)$**
EAT = 0.95 (1 + 100) + 0.05 (200 + 1) = 106nans

❑ Memory Protection

In multiple partitions, memory protection is performed using base & limit registers.

While In paging, protection is performed using additional bits in the page table.

(1) **Legal/Illegal Bit**:
0 -> Illegal page
1 -> legal page
(2) **R/W Bit**:
0 -> Read only page,
1 -> Read & Write page

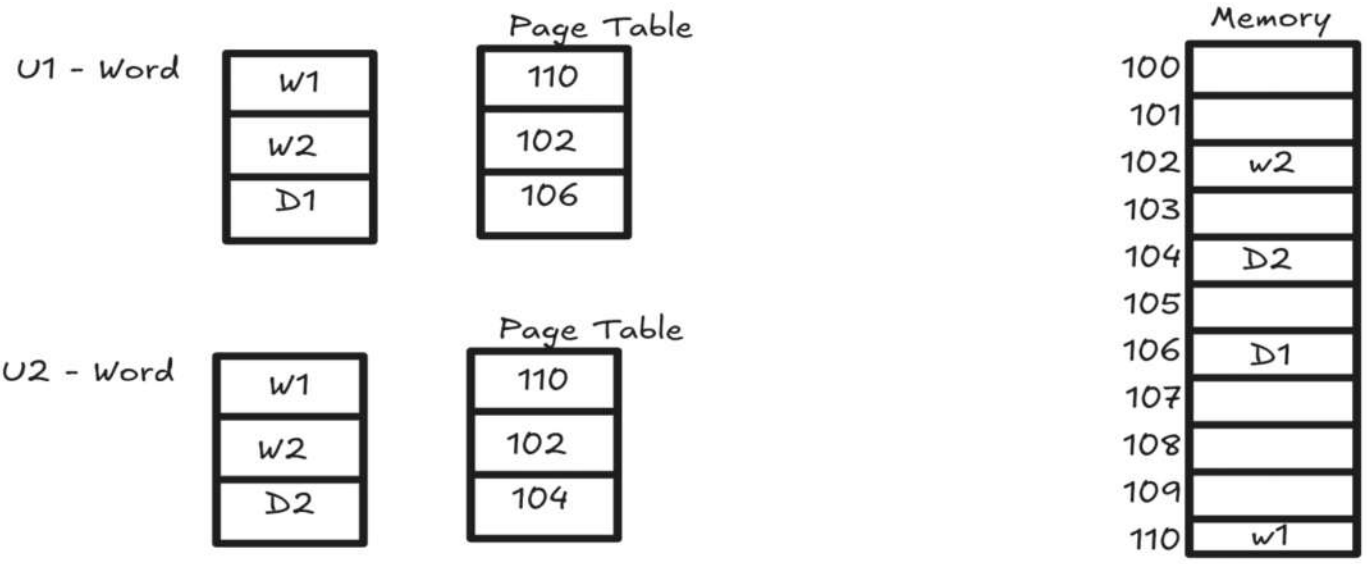


Page Table

Page #	Frame #	L/IL	R/W
0	12501	1	1
1	7415	1	1
2	8001	1	1
3	120	0	0
4		0	0

❑ Paging Advantages

'Advantages of sharing pages'



❑ Paging Disadvantages

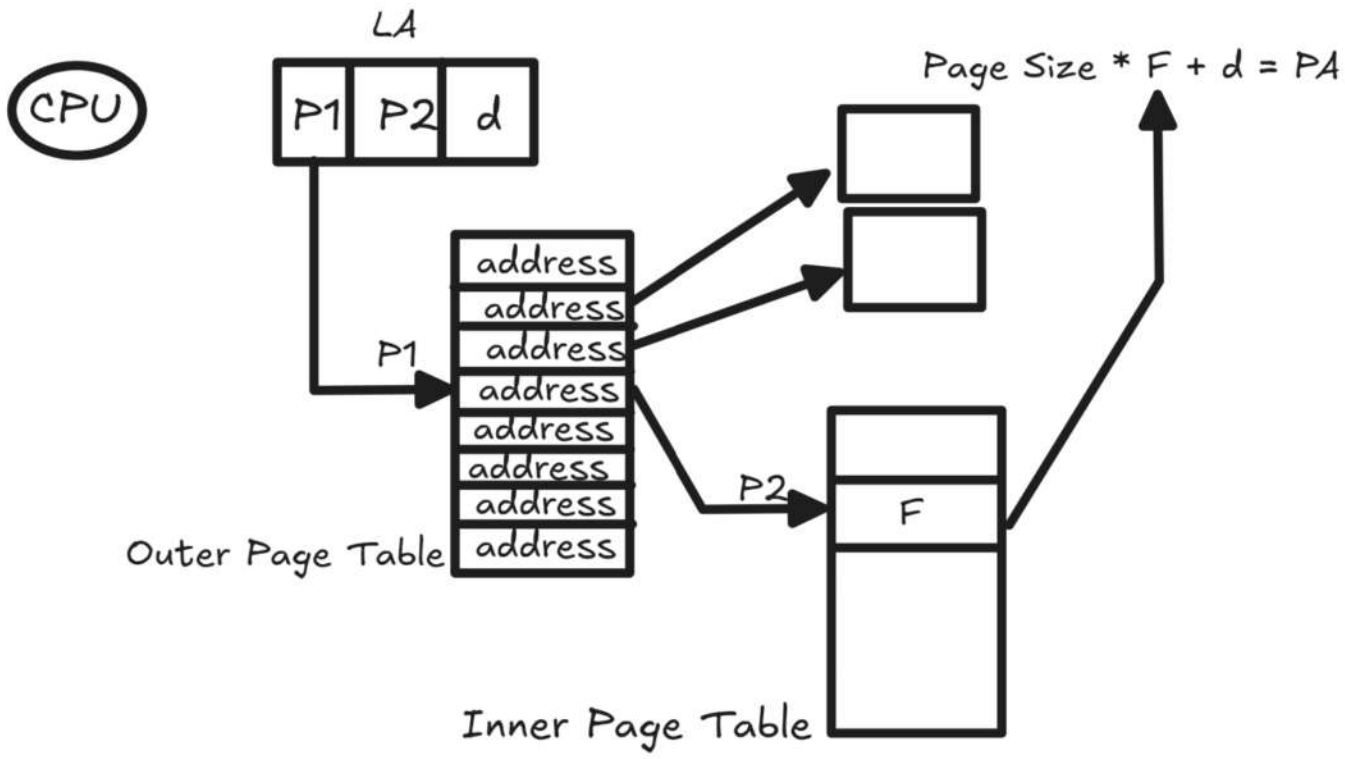
Some people have reservation that the program is divided in too many pieces in memory.

❑ Multi-Level Page Table

Assume LA = 32bit & Page size = 2^{12}

So, page table size = 2^{20} * 4 = 4MB. Which is big to store contiguously in memory.

Let's divide the page table into 2 levels

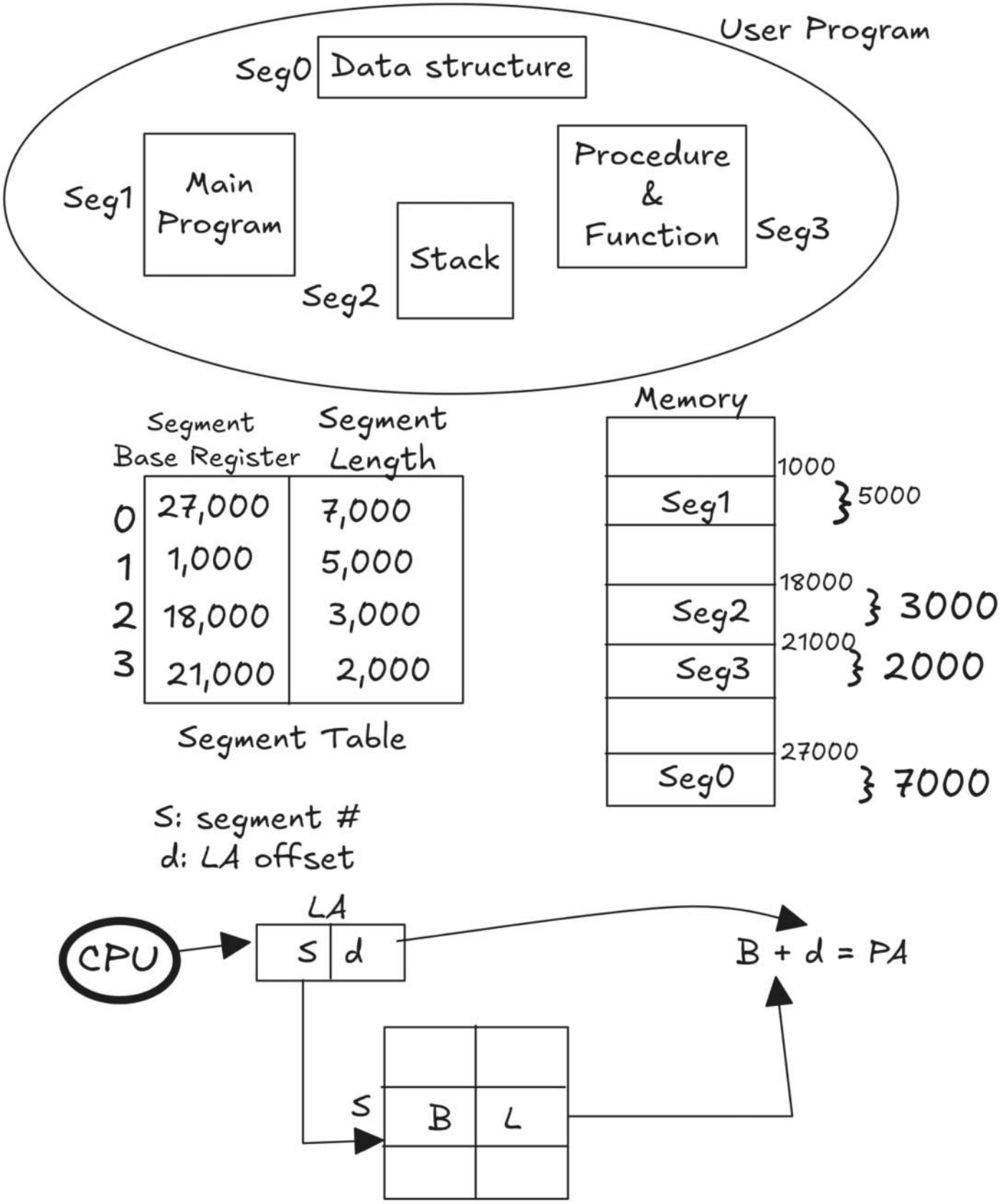


- In our example, assuming P1 = 8 bits & P2 = 12bits
- Outer Page table size = 2^8 * 4 = 1KB
- Inner Page table size = 2^{12} * 4 = 16KB

⚠ Problem: We need more memory access as the LA increases in size.

- In two level page table, we need 3 memory accesses.
  - One to get address (P1)
  - One to get F
  - One to fetch the instruction
- In four level page table, we need 5 memory access. But, with good associative register algorithm, the performance will be as well good
- Paging separates the user's view of the memory from the actual memory

[3] Segmentation

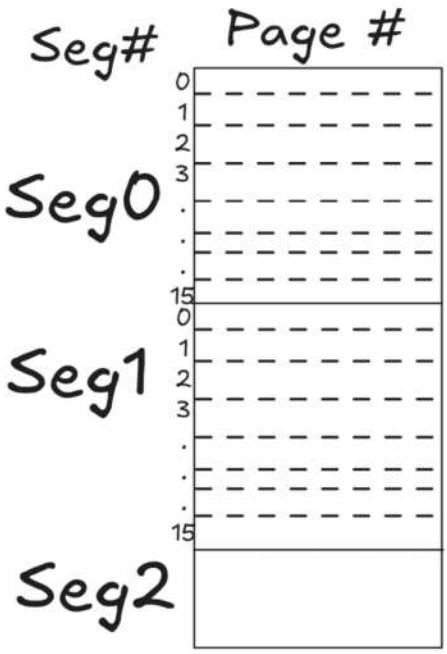


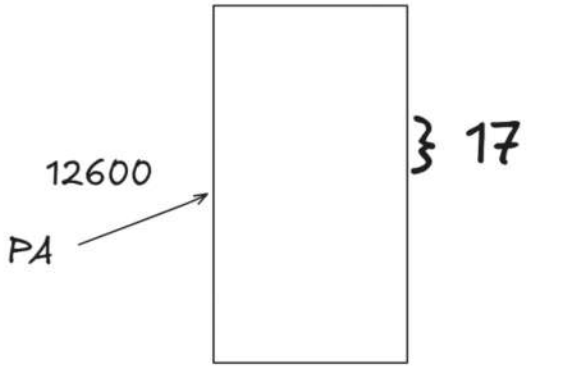
Example: S = 2, d = 350

PA = B + d = 18000 + 350 = 18350

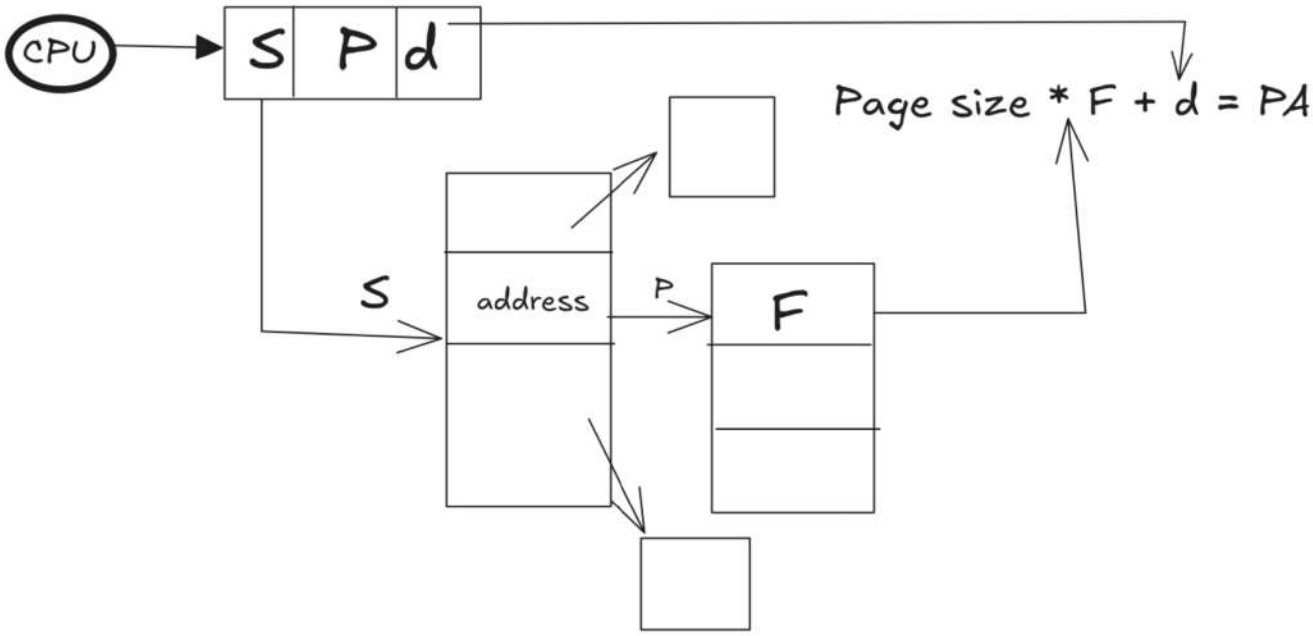
❑ Segmentation with Paging

'Paging with segments'



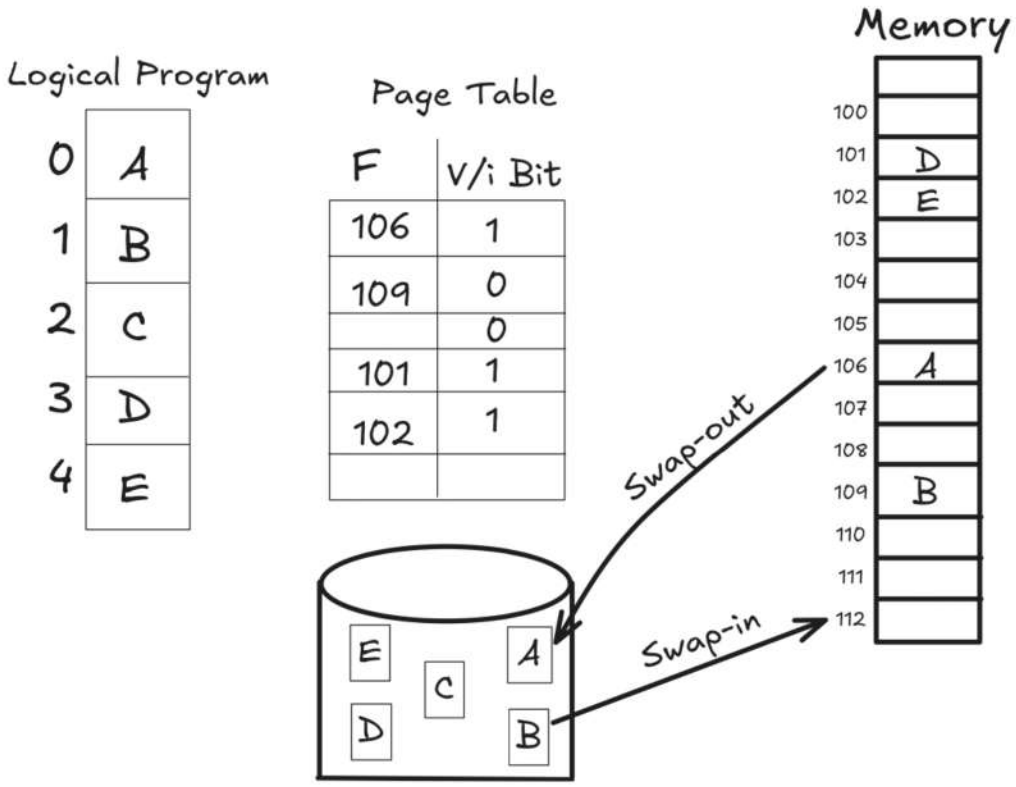


If seg size = 64K
Page size = 4K
every seg contains 16 pages



Chapter 9 # Virtual Memory Management

➔ How to run bigger programs on smaller memory?
➔ No need for the whole program to be loaded into memory, only the part of it will do.
➔ We will discuss '**Demand Paging**'

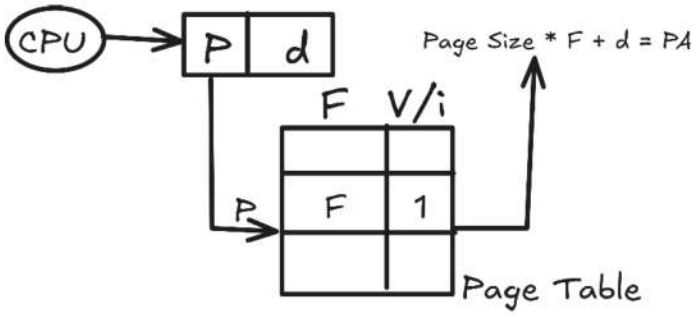


V/i (Valid/Invalid) Bits

1: in memory. 'Valid'

0: not in memory 'Invalid'

- The LA is checked through the page table if its valid address (V/i bit = 1) that it continues the execution as usual.
- If the V/i bit = 0, that's the page isn't in the memory, then we say a page fault occurs.
- The OS looks for free frame in memory, swap in the required page from the HD, it updates the page table & resumes execution.
- If there is **no free frame?**
  - ➢ It selects a victim frame
  - ➢ Maybe, it swap-out the page from memory to HD.
  - ➢ It swap-in the required page from HD to Memory.
  - ➢ Update the page table.
  - ➢ Resumes execution.



❑ Performance of demand pages

- Assume Page fault rate (probability) = **P** $(0 \le P \le 1)$
  - If P = 0, All execution with no page fault. "Perfect"
  - If P = 1, Every execution of instruction, there is a page fault.
- Assume memory access = m

$$\textbf{Page fault overhead} = \textbf{Swap}_{in} + \textbf{Maybe Swap}_{out}\ \textbf{Page} + m$$

$$\textbf{EAT} = (1 - P) * m + P * \textbf{Page fault overhead}$$

Example: Assume memory access = 10mics

- Page fault rate (P)
- Page transfer (swap) time = 10mils
- 40% of the time the page needs to be swap-out

$$\textbf{EAT} = (1 - P) * 10 + P * (10 * 1000 + 0.4 * 10 * 1000 + 10)$$

$$= 10 + 14000P \approx 14000\,P$$

⚠ تحويل الوحدات الأكبر لوحدات أصغر mils, mics → mils -> mics

∴ Result: Performance depends on P (page fault rate).

❑ Objective: Minimize the page fault rate

⚠ Note: A new bit called "**dirty bit**" is added to the page table to indicate if the page is modified.

1: modified or 0: not modified

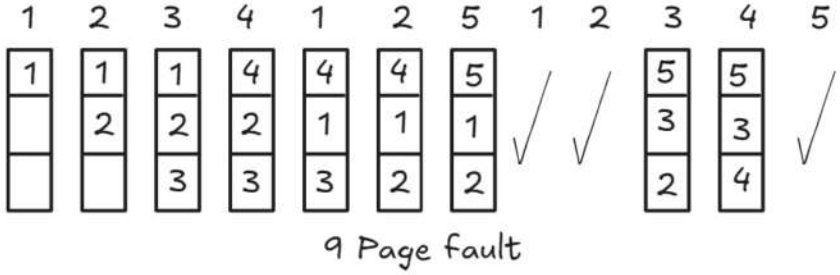
❑ Page Replacement Algorithms

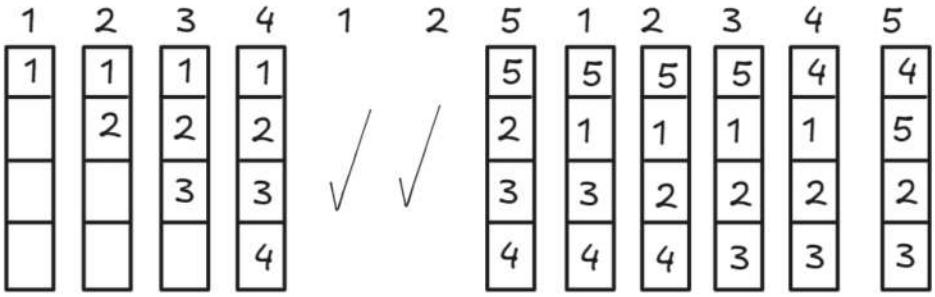
The OS must select the victim frame, so that, to minimize the page fault rate.

[1] FIFO

Replace the page which enters the memory first (Oldest page in memory).

Example: given the following page references.

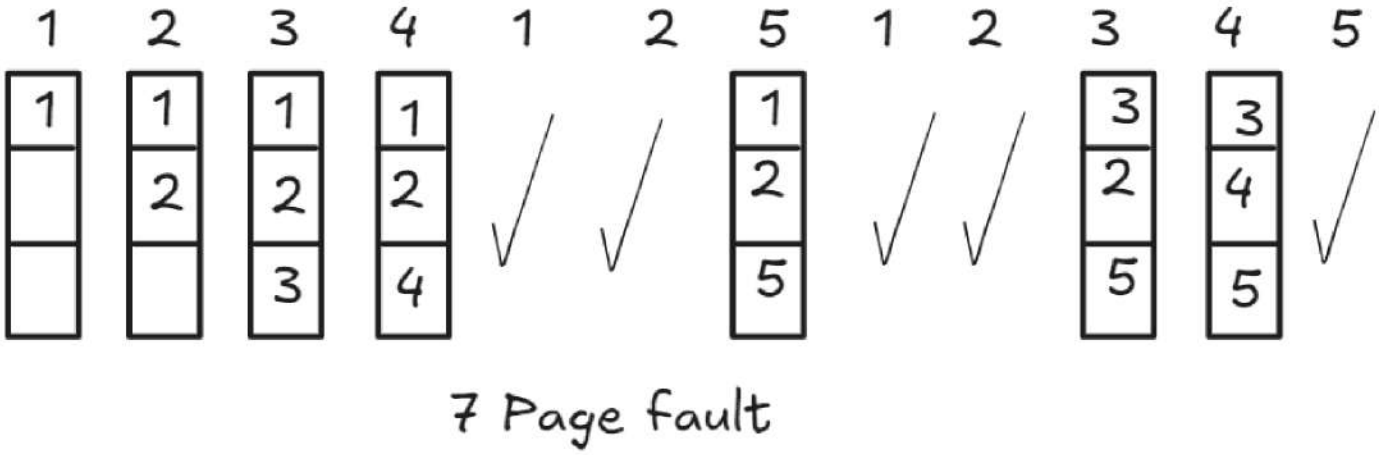




∴ Belady's Anomaly

[2] Optimal Replacement

Replace the page which won't be used in the future for the longest period.

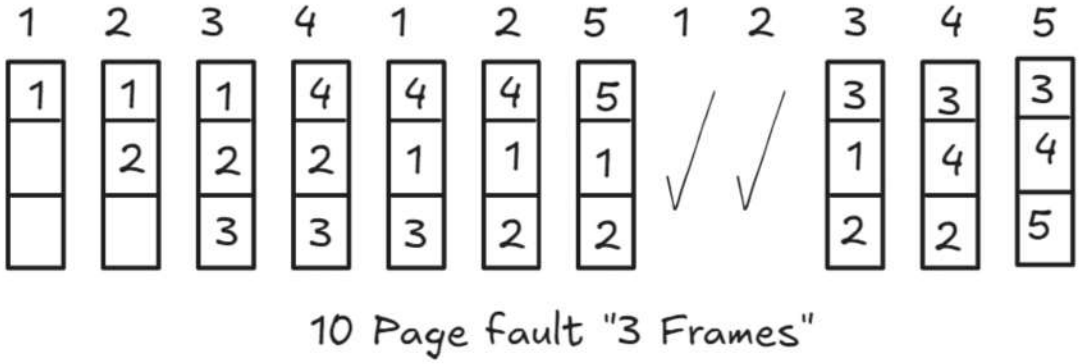


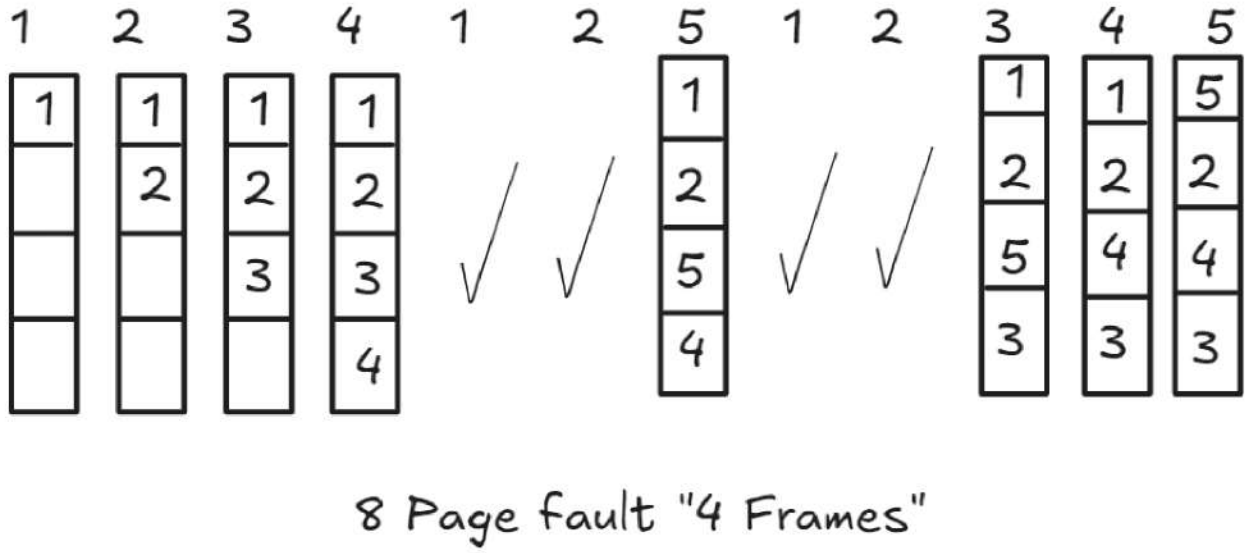
⚠ **Major Problem**: How the OS know in advance the next page in which execution occurs?!

☺ Optimal Replacement is used as a bench mark.

[3] Least Recently Used [LRU]

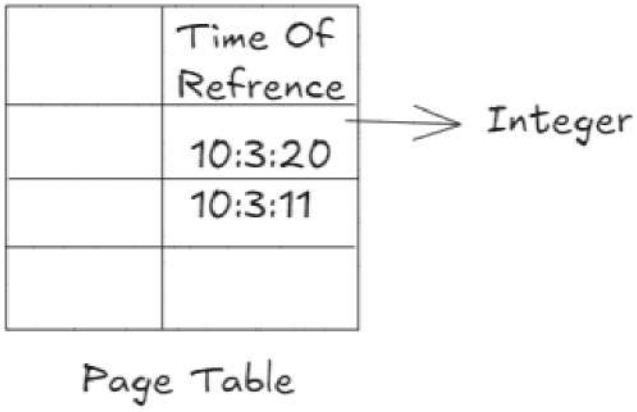
Replace the page which have not been used in the past for the longest period of time.



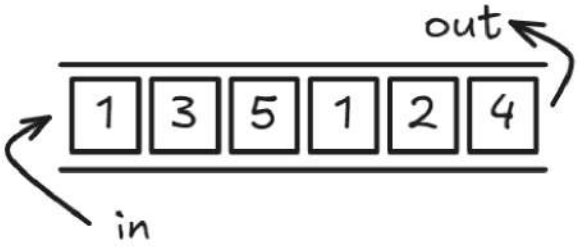


Implementation:

(1) Add a counter to the page table that contains the last time the page is referenced (used).



(2) Keep a queue of referenced pages



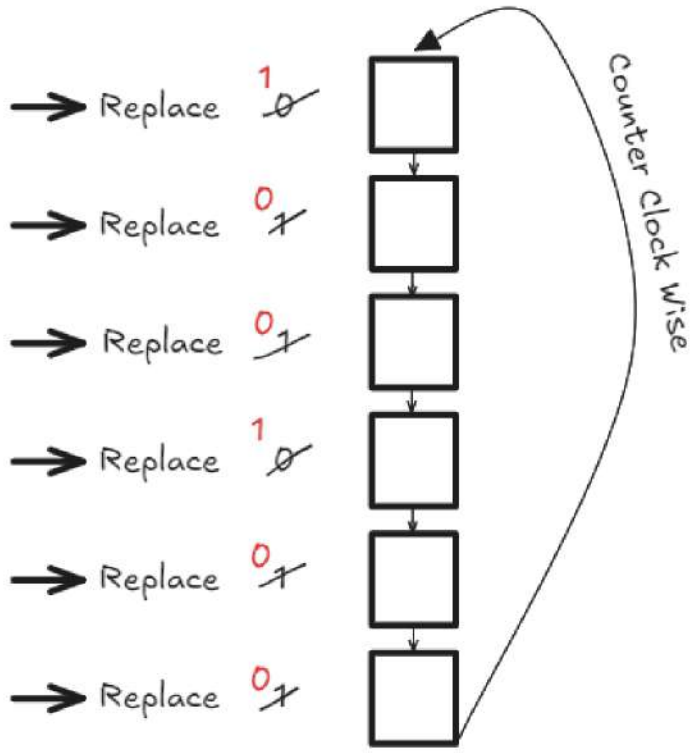
❑ LRU approximation:

Add one bit to the page table called (**reference bit**)

➔ 1: page is referenced (used) READ or WRITE
➔ 0: page isn't referenced.

➔ LRU approximation, replace the page whose **reference bit** is 0.

❑ Second Chance



❑ Enhanced Second Chance

Use the reference bit & the dirty bit (Reference bit, Dirty Bit)

➔ (0, 0) = Page isn't referenced & not modified → BEST
➔ (0, 1) = Page isn't referenced but modified
➔ (1, 0) = Page is referenced but not modified
➔ (1, 1) = Page is referenced & modified → WORST

*** Counting Algorithms**

1- **MFU** (Most Frequently Used)

Replace the page which has been used for the maximum # of all times.

2- **LFU** (Least Frequently Used)

Replace the page which has been used for the minimum number of times.

❑ Global vs. Local Replacement

❑ Allocation of frames for processes

(1) **Equal Allocation**: Each process gets the same number of pages (frames).
Example: memory 100 frames, 5 process.
- Every process gets (100/5) = 20 frames.
- Unfair → not good performance.

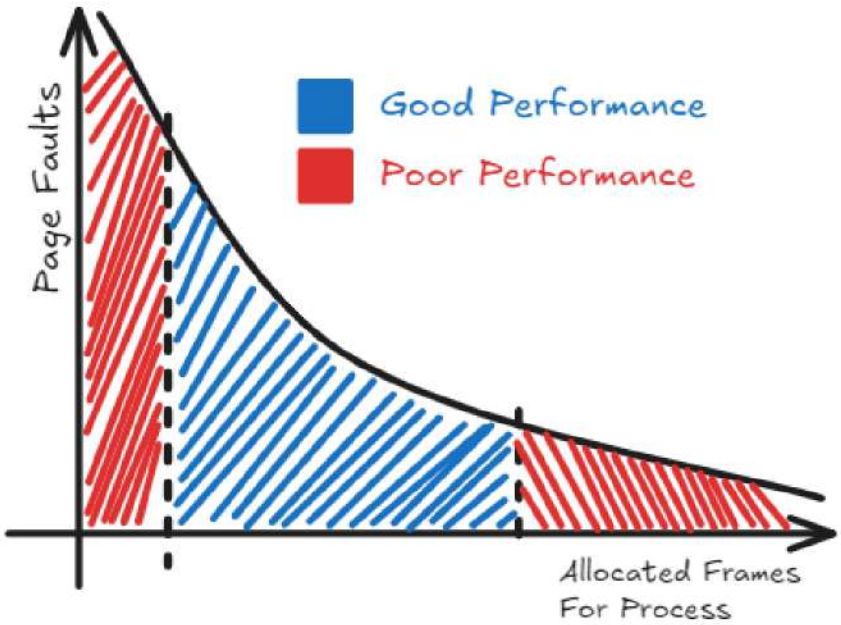
(2) **Proportional Allocation according to size**:
Example: Assume size of process Pi = Si

- Assume we have **m** frames of memory process Pi gets $\frac{Si}{\sum Si} * m$
- Assume memory 100 frames, 3 process with sizes 100, 400, 700 KB
- P1 = $\frac{100}{1200} * 100 \approx 8\ frame$
- P2 = $\frac{400}{1200} * 100 \approx 34\ frame$
- P3 = $\frac{700}{1200} * 100 \approx 58\ frame$

(3) **Proportional according to priority**:
Example:
- Assume we have P1, P2, P3 with priority 2, 3, 7 and we have 100 frames in memory
- P1 = $\frac{2}{12} * 100 \approx 17\ frame$
- P2 = $\frac{3}{12} * 100 \approx 25\ frame$
- P3 = $\frac{7}{12} * 100 \approx 59\ frame$

❑ Thrashing

The OS is busy swapping page in & out.



Some times the number of allocated frames for the process is low, which leads to poor performance (Low Utilization)

The OS thinks that degree of multiprogramming is low

→ OS increases the degree of multiprogramming

→ The system gets worse

→ Most of the time OS is doing swapping of pages.

Chapter 10

❑ File Concept

- **Contiguous logical address space**
- Types:
  - **Data**
    Numeric, characters, binary
  - **Program**

    Source, object (load image)
  - **Documents**

❑ File Attributes

- **Name**: only information kept in human-readable form.
- **Type**: needed for systems that support different types.
- **Location**: Pointer to file location on device.
- **Size**: Current file size.
- **Protection**: Controls who can do reading, writing, executing.
- **Time, date, and user identification**: for protection, security, and usage monitoring.

(***) Information about files are kept in the directory structure, which is maintained on the disk, which is called generally, **Device Directory**.

❑ File Operations

- Create
- Open
- Close
- Write
- Read
- Reposition within the file
- Delete

❑ Access Methods

A- Sequential Access
read next
write next
reset

No read after last writes (generally no rewrite)

B- Direct Access
read n
write n
position to n
read next
write next
rewrite n

n = relative block number

❑ Directory Structure

The general information kept about the directory system are:

- Name
- Type
- Address (Location)
- Current Length
- Maximum Length
- Current Position (File Pointer -- FP)
- Date last accessed (for archival)
- Data last updated (for dump)
- Owner ID
- Protection Information

Operations performed on directory:

- Search file
- Create file
- Delete file
- List a directory
- Rename a file
- Traverse the file system e.g. Date last accessed (for archival)

Directory Systems

1- **Device Directory:** The directory where the **physical information** generally is kept about the files in that device, such as, name, size, date, ... etc.
2- **User File Directory:** The directory where the **logical information** generally is kept about the user files, such as, name, size, date, ...etc.

❑ Device Directory Implementation

Hash Table – Liner list with hash data structure.

- Decreases directory search time
- Collisions – situation where two files' names hash to the same location.

❑ User File Directory Implementation

Organize the directory (logically) to obtain:

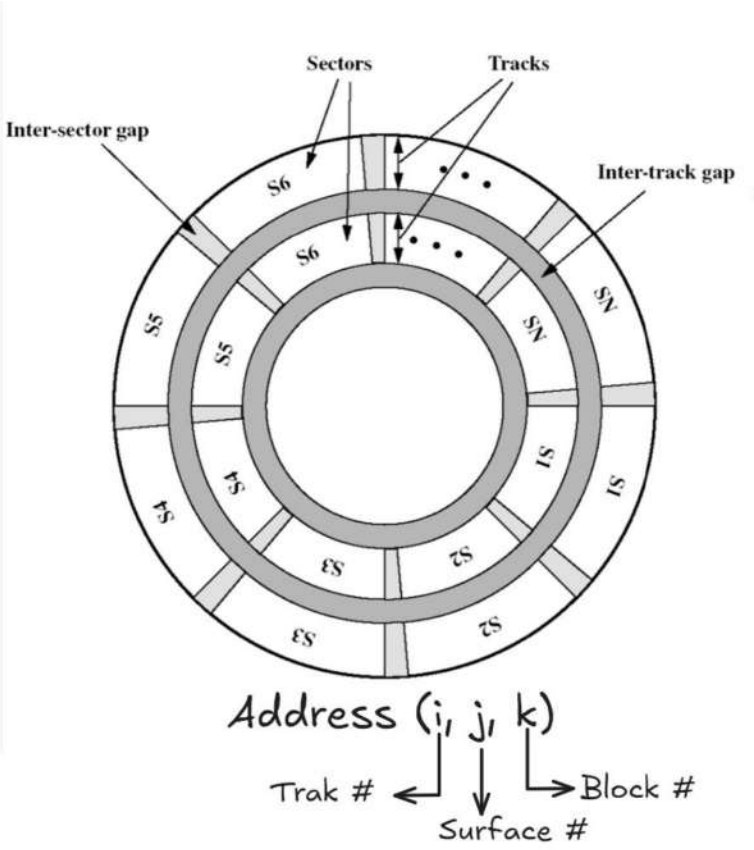
1. Efficiency – locating a file quickly
2. Naming – convenient to users
   - Two users can have the same name for different files.
   - The same file can have several different names (Aliases)
3. Grouping – logical grouping of files by properties, e.g. all C programs files, all Games ...etc.

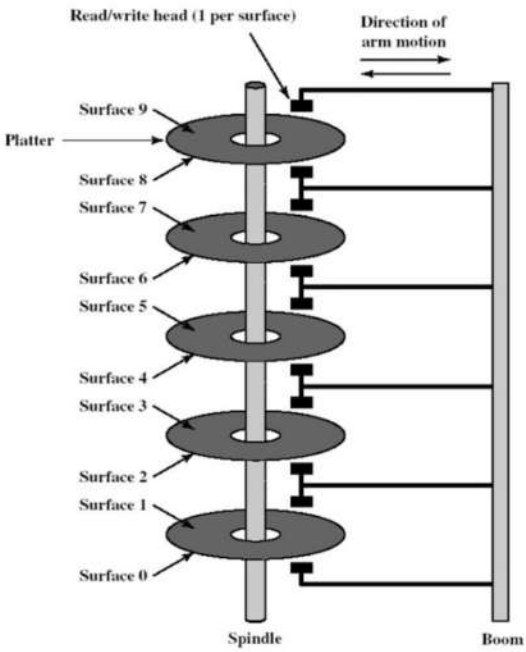
The structure of **UFD:**

Tree-structured directories

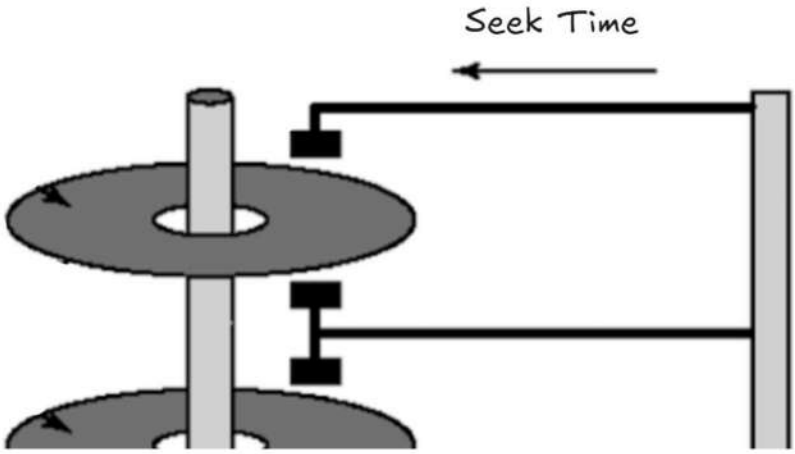
- Efficient searching
- Grouping capabilities
- Current directory (working directory)

Chapter 11 + 12





* The time to move from one track to another is called '**seek time**' which is a mechanical move.



❑ Disk Structure

- A disk can be viewed as an array of blocks
  A **sector (block)**: smallest addressable unit in the disk (track, surface, sector)
- Given the address (i, j, k), then transformation from 3-dim to 1-dim

  b = k + s * (j +i * t)
  where t = number of surfaces (tracks per cylinder)
  s = number of sectors per surface

- **Seek Time**: The time needed to move the R/W head to particular track.
- **Latency Time**: Time to rotate the sector under the R/W head.
- **Access Time**: Seek time + latency time + transfer time.

- **We could improve the seek time by selecting a good disk scheduling algorithm. (minimize seek time)**

❑ Blocking

Packing and unpacking a number of logical records in physical block.

Bloking factor: The number of logical records packed into a physical block.

❑ Free-Space Management

- **Bit Map**: Vector (n blocks)
- **Linked List** (Free List)
- **Counting:** keep the address of the **first free block** and the number **n** of adjacent free blocks. This is best used with contiguous allocation.
- **Grouping**: Store the addresses of **n** free blocks in the first free block.

⚠ **Note**: The logical file must be mapped into the physical storage media (disk).

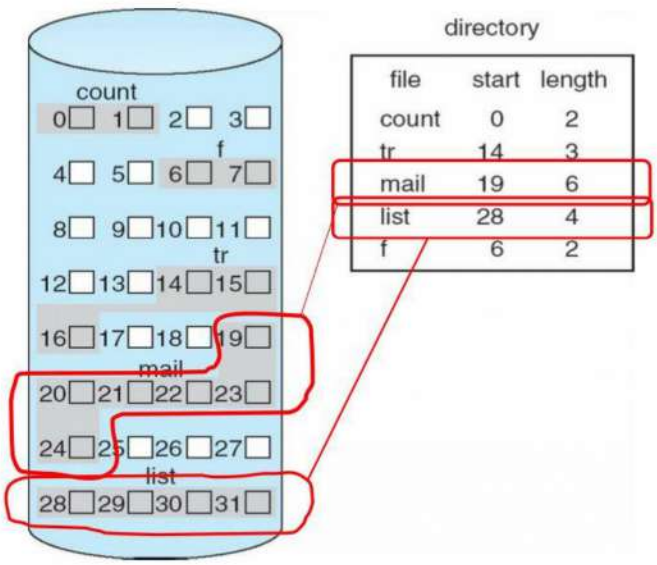
❑ Allocation Methods

How the disk blocks are selected and allocated for a file.

⚠ **Note**: Every storage device has '**device directory**' which contains the information about the file stored on that device, generally it's a Hash Table.

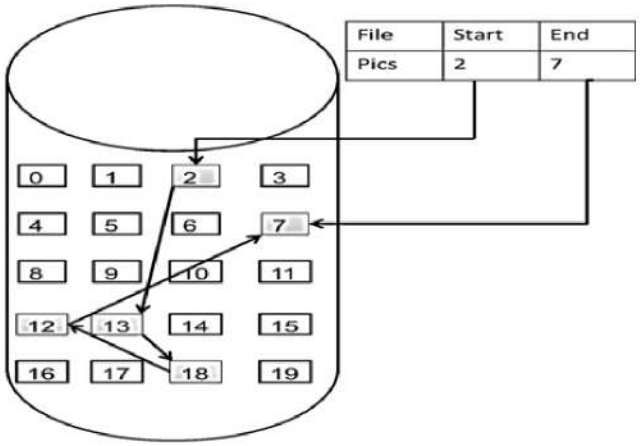
1. Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
- The file is defined by address of the first block and its length.
- No seek time is required to access block (b+1) after block b unless is last block in the cylinder.
- **Random and Sequential** access are supported easily.
- **Problem**: External fragmentation (holes) in the disk.
- **Solution**: Compaction
- **Major Problem**: Files can't grow ☹
- How to find hole for the file: First Fit, Best Fit, Worst Fit.



2. Linked Allocation

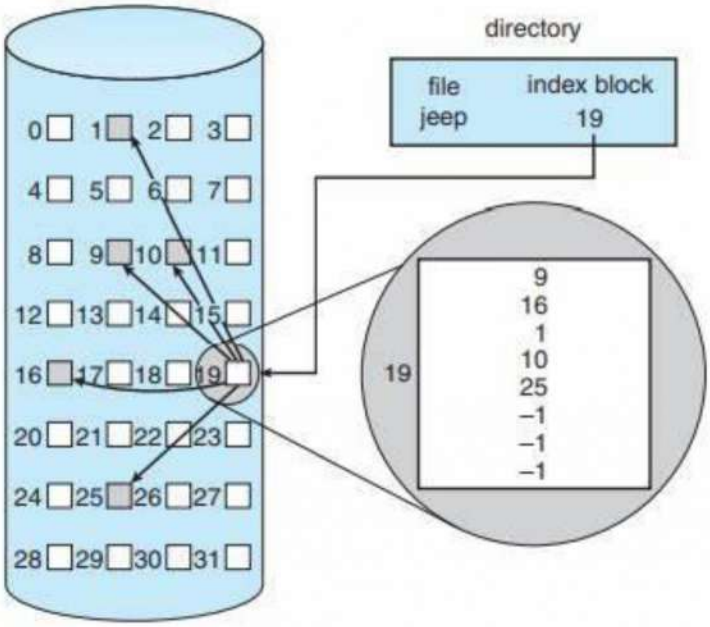
- Each file is a linked list of disk blocks.
- Blocks may be scattered anywhere on the disk.
- Allocate as needed, link together
- **Advantages**:
  - **Simple** – need only starting address and size.
  - Free-space management system – no waste of space. (No External Fragmentation)
  - File can grow ☺
- **Problem**: Address pointer waste.
- **Major Problem**: Supports **only** sequential access.



3. Indexed Allocation

Brings all pointers together into the ***index block***

- Need index table.
- Random access in addition to sequential access.
- Dynamic access without external fragmentation, but have over head of index block.
- Mapping from logical to physical in file of maximum size of 256K words and block of size 512 words. We need only 1 block for index table.



❑ Disk Scheduling

- Disk Requests – Track/Sector
  - Seek
  - Latency
  - Transfer
- Minimize Seek Time
- Seek Time >> Seek Distance
- A number of different **algorithms** exists
  - **FCFS**
  - **SSTF**
  - **SCAN**
  - **LOOK**
  - **C-SCAN**
  - **C-LOOK**

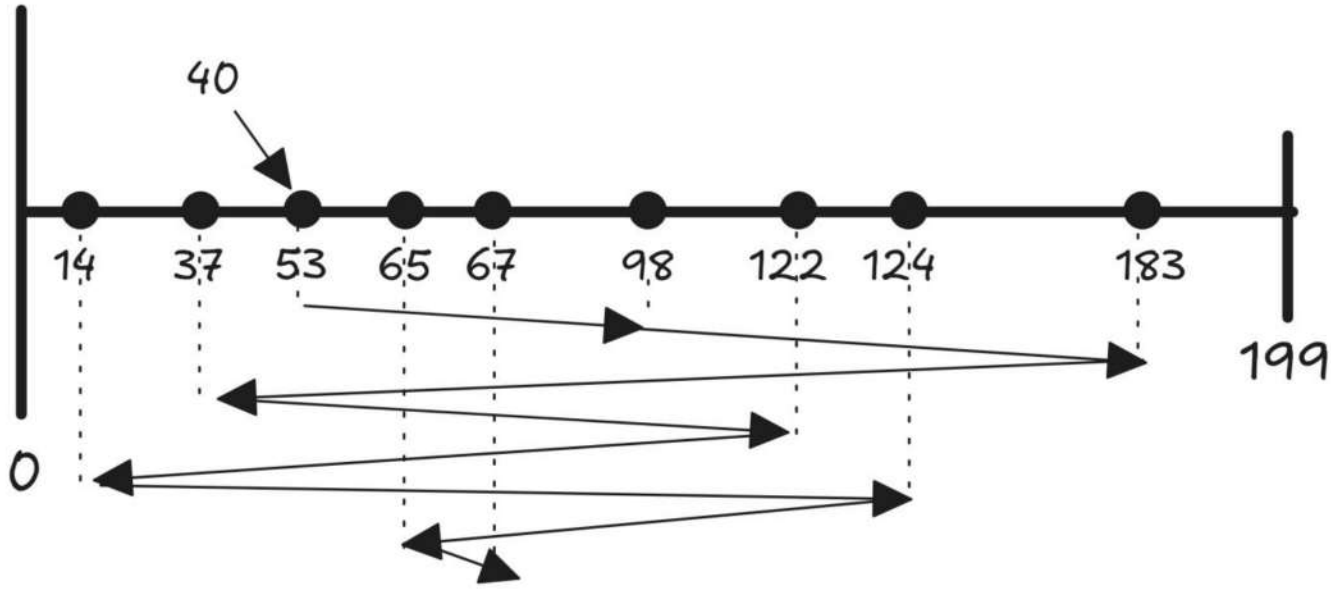
Example

Assume HD has 200 tracks (0 - 199) given the queue of HD requests as follows:

98, 183, 37, 122, 14, 124, 65, 67

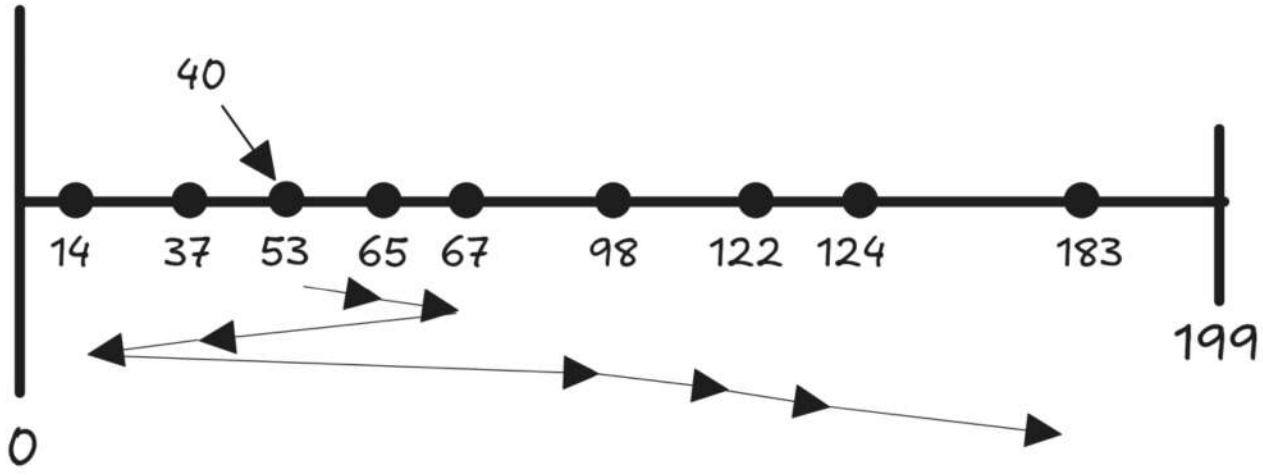
Assum the R/W head is currently serving a job at track **53** & just finished serving a track job at track **40**.

(1) **FCFS**



$$\text{Average Head Movement} = \frac{(183-53)+(183-37)+(122-37)+(122-14)+(124-65)+(67-65)}{8}$$

(2) **Shortest Seek time First (SSTF)**

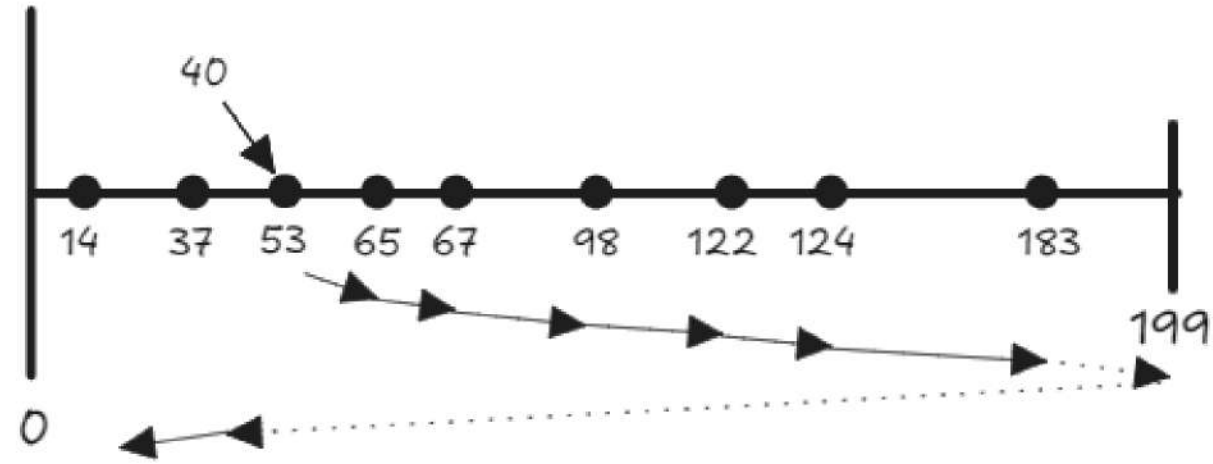


AHM = 29 traks/job

SSTF gives the optimal/minimum solution

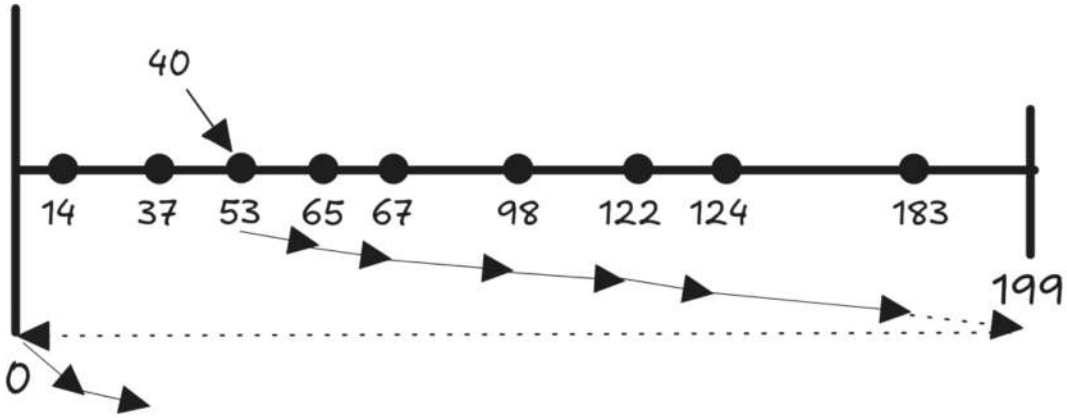
⚠ Problem: Starvation

(3) **SCAN (Elevator)**



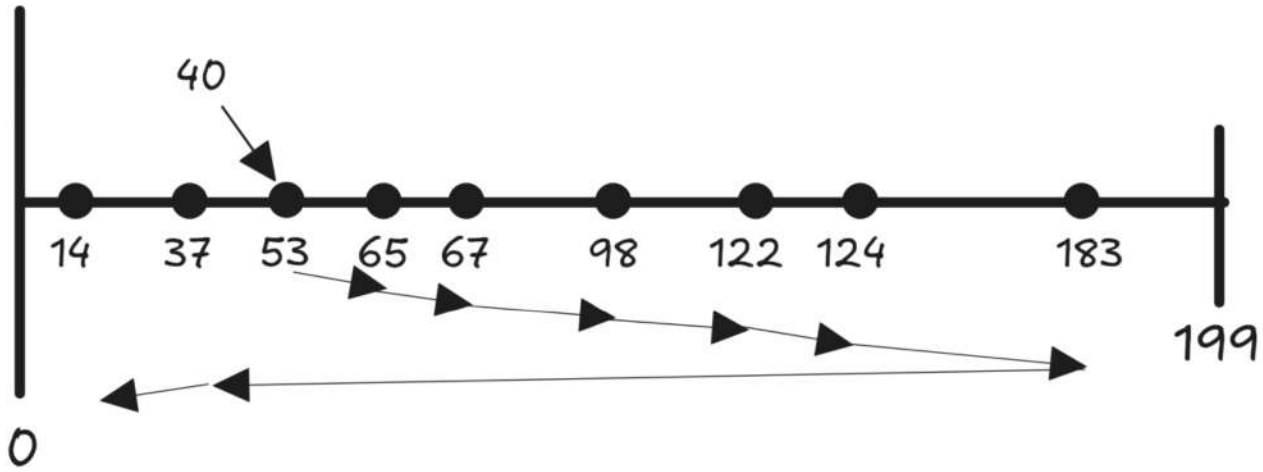
AHM = 41 track/job

(4) **C-SCAN (Circular Scan)**



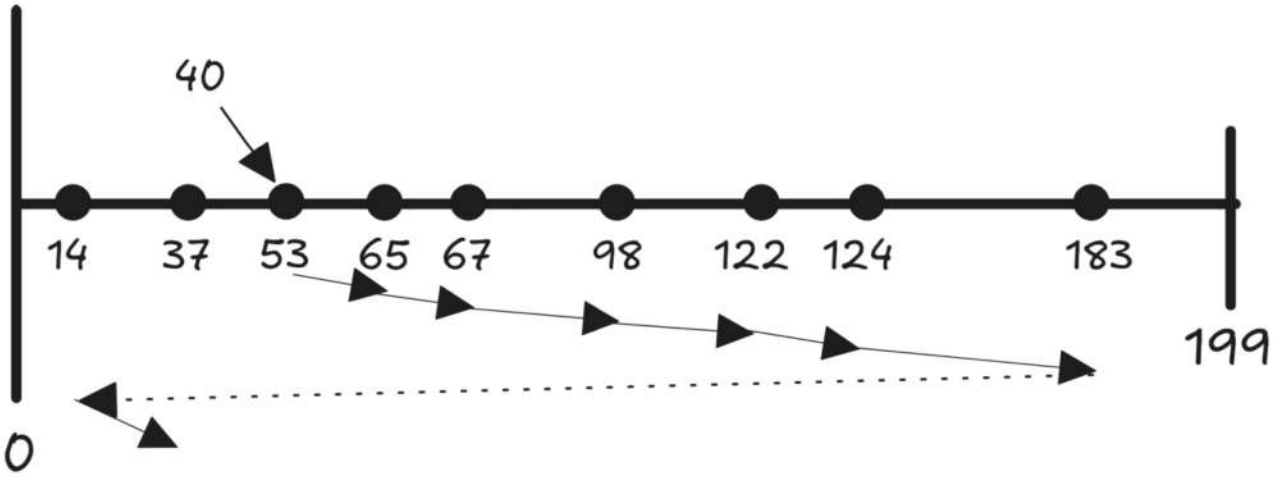
AHM = 47 tracks/job

(5) **Look**



AHM = 37 track/job

(6) **C-Look**



AHM = 42 tracks/job

Formulas

Chapter 5: CPU Scheduling

- $\boldsymbol{Turnaround\ Time = Finish\ Time - Arrival\ Time}$
  - This is the total time it takes for a process to complete, from submission to finish.
- $\boldsymbol{Waiting\ Time = Turnaround\ Time - Service\ (CPU)\ Time}$
  - This is the time a process spends in the READY queue, waiting for the CPU.
- $\boldsymbol{Weighted\ Turnaround\ Time = \frac{Turnaround\ Time}{Service\ (CPU)\ Time}}$
  - This is a ratio of turnaround time to service time.
- $\boldsymbol{Y_{n+1} = w * T_n + Y_n * (1 - w)}$
  - This formula is for estimating the length of the next CPU burst, where:
    - Y_{n+1} is the estimated length of the (n+1)th CPU burst
    - T_n is the actual length of the nth CPU burst
    - Y_n is the estimated length of the nth CPU burst
    - w is a constant between 0 and 1
- $\boldsymbol{EAT = h(m + t) + (1 - h)(2m + t)}$
  - This formula is used to calculate the Effective Access Time with associative registers
    - EAT = Effective Access Time
    - h = Hit Ratio
    - m = Memory Access Time
    - t = Search time in associative register

Chapter 8: Memory Management

- **PA = LA + Base Register**
- **P = LA / Page Size**
- **d = LA % Page Size**

Chapter 9: Virtual Memory Management

- $\boldsymbol{Page\ fault\ overhead = Swap_{in} + (Maybe\ Swap_{out}) + m}$
- **EA = (1 - P) * m + P * page fault overhead**
  - This formula calculates the Effective Access time with demand paging
    - EA = Effective Access time
    - P = Page fault rate
    - m = memory access time
    - page fault overhead = time to swap pages in and out.

Chapter 11 + 12

- **b = k + s * (j + i * t)**
    - This formula is for transforming a 3-dimensional disk address to a 1-dimensional block address, where:
        - b is the block number
        - k is the sector number
        - j is the surface number
        - i is the track/cylinder number
        - s is the number of sectors per surface
        - t is the number of surfaces (tracks per cylinder)