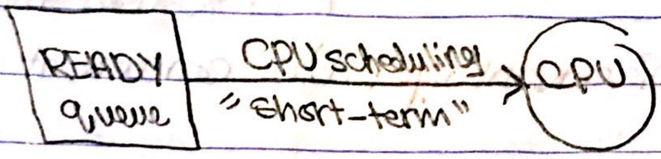
Chapter #5:

CPU Scheduling.

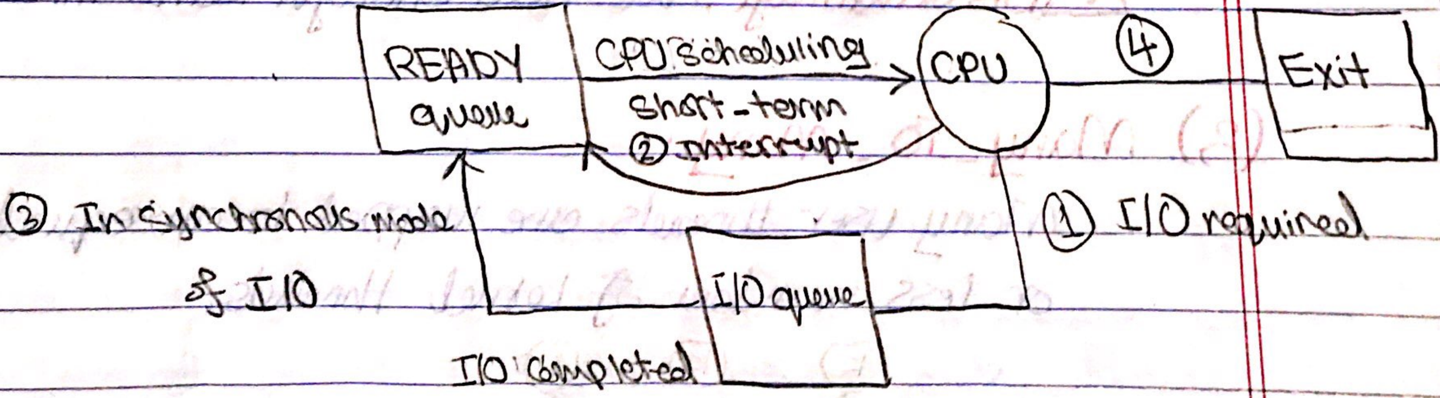
CPU scheduling:

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 CPU scheduling?

- I/O required.
- Interrupt.
- Process I/O is completed.
  → (In synchronous Mode)
- Process terminated.



In Cases:

- 1 & 4 Non-Preemptive. "تخلت عن ال CPU طوعية"
- 2 & 3 Preemptive: "مع طرد أو إخلاء ، أخذت ال CPU عنوة"

→ 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 finishs execution.

(4) Waiting Time: (min).
it's the time the process spends in the READY queue.

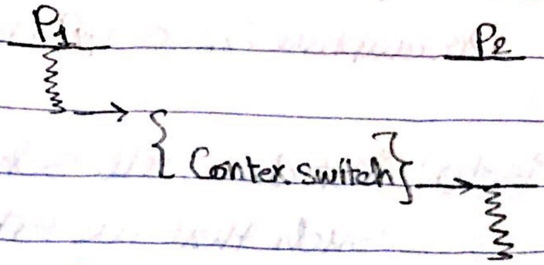
(5) Response Time:
it's the time from submitting job until you see the first response from the computer.

Weighted Turnaround Time = (minimum is better)
Turnaround Time / service (CPU) Time.

→ Every switching from processes Ⓟi to Ⓟj needs 2 Context switch.

lecture #12

March-6-2018
Tuesday



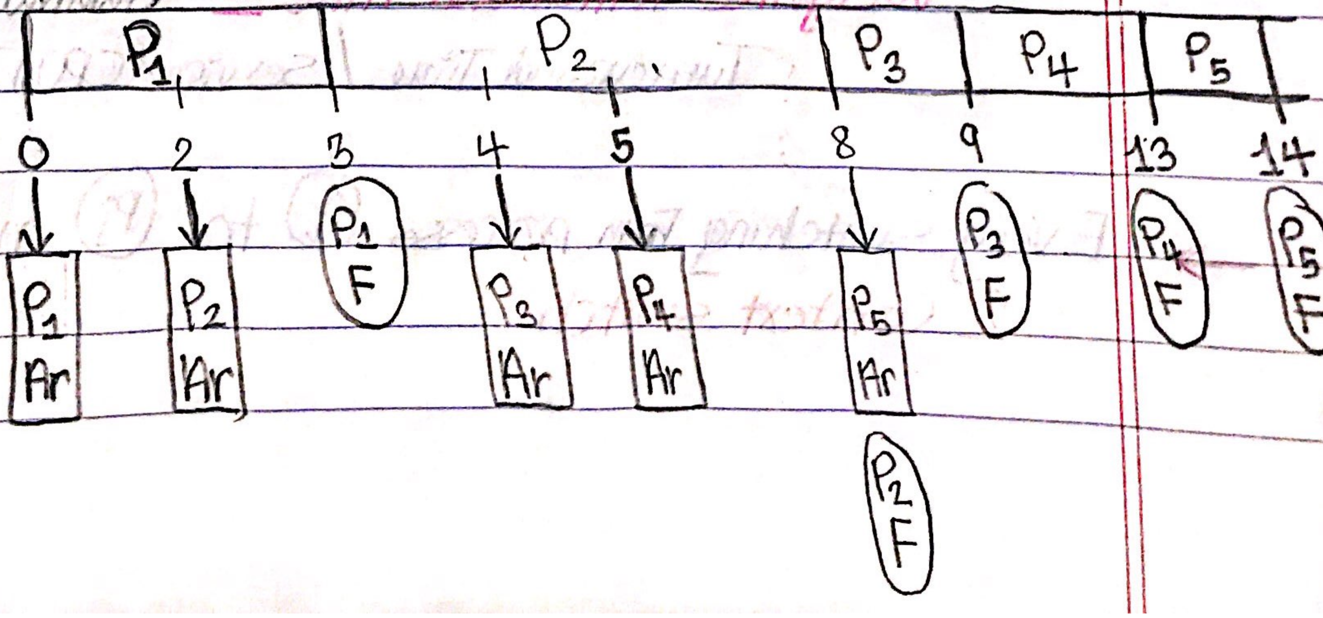
[1] FCFS → First Come First Serve :

example: given the following Ready queue:

Process	Arrival time	Service time (CPU burst)
P_1	0	3
P_2	2	5
P_3	4	1
P_4	5	4
P_5	8	1

Compute the average turnaround time & average waiting time.

→ We use Gantt diagram



- Turn around = Finish Time _ Arrival time.
- Waiting time = Turn around time _ Service (CPU) time.

- Average turn around time =

$$\frac{(3-0) + (8-2) + (9-4) + (13-5) + (14-8)}{5} = 5.6 \text{ unit.}$$

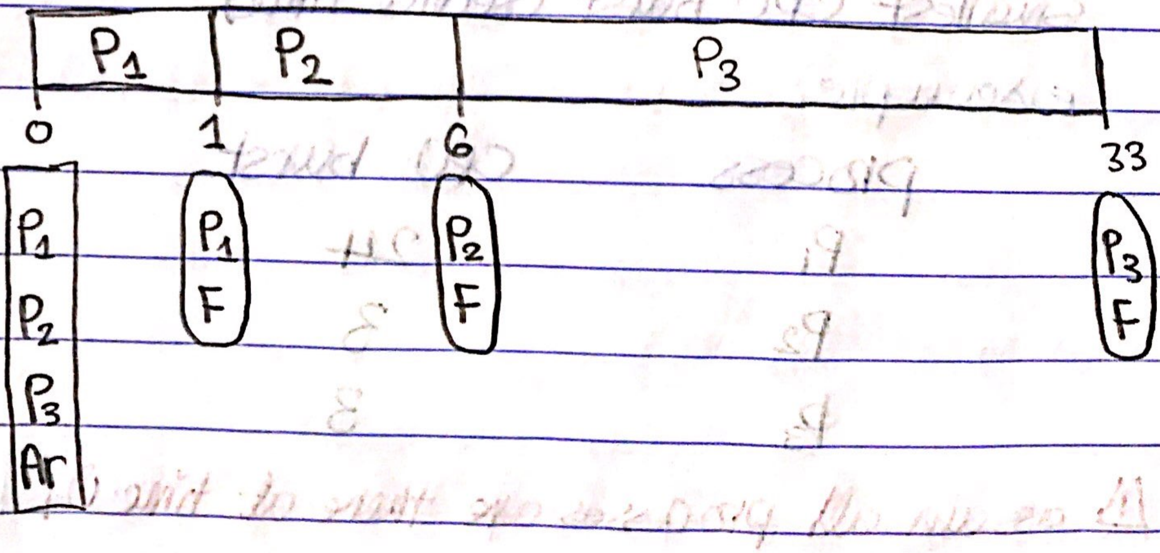
- Average waiting time =

$$\frac{(3-0-3) + (8-2-5) + (9-4-1) + (13-5-4) + (14-8-1)}{5} = 2.8$$

FCFS: Convoy problem "كيفية وصول الموكب"

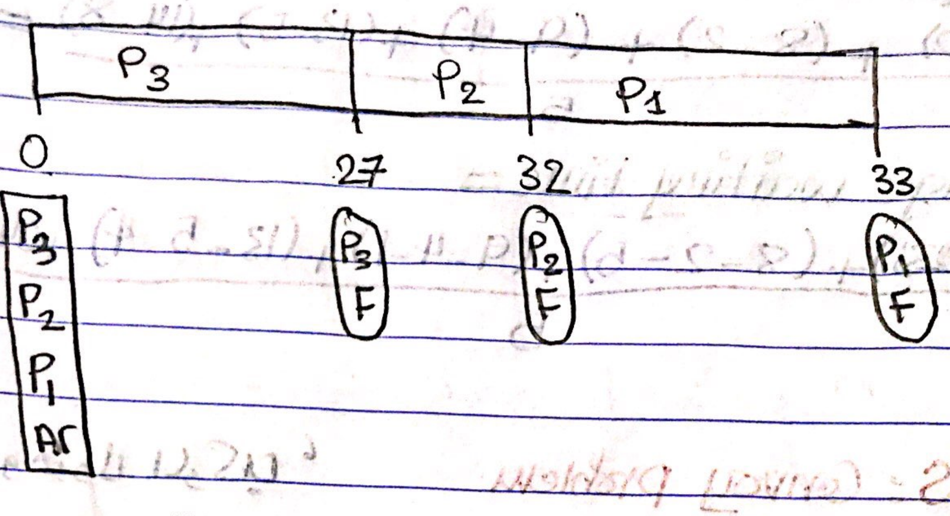
example:-

Process	CPU burst
P_1	1
P_2	5
P_3	27



- ATT = (1-0) + (6-0) + (27-0)/3 = (40/3)
- AWT = (1-0-1) + (6-0-5) + (33-0-27)/3 = (7/3)

Process	CPU burst
P_3	27
P_2	5
P_1	1



— $ATT = (27-0) + (32-0) + (33-0)/3 = (92/3)$

— $AWT = (27-0-27) + (32-0-5) + (33-0-1)/3 = (59/3)$

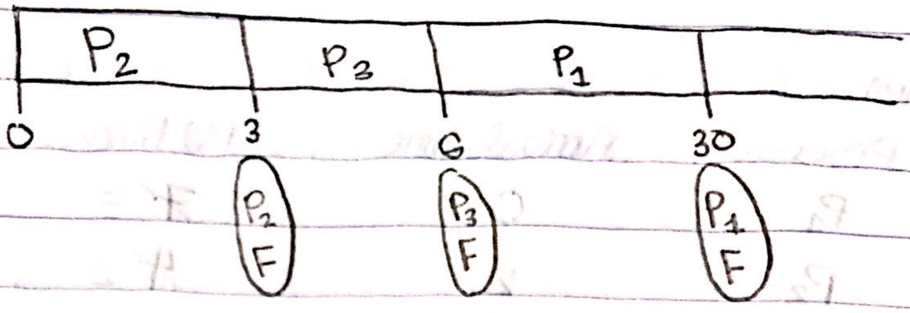
[2] Shortest Job First:

The CPU is given to the process with the smallest CPU burst (service time)

example:

process	CPU burst
P_1	24
P_2	3
P_3	3

⚠ assum all processes are there at time 0, and arrive in the same order.



$$* \text{ ATT} = \frac{3+6+30}{3} = \frac{39}{3} = 13$$

$$* \text{ AWT} = \frac{0+3+6}{3} = \frac{9}{3} = 3$$

⚠ **Note:** Shortest job First gives the minimum (optimal) Solution, that is it gives the minimum waiting time.

→ There are two versions of SJF:

(1) Preemptive: if a 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.

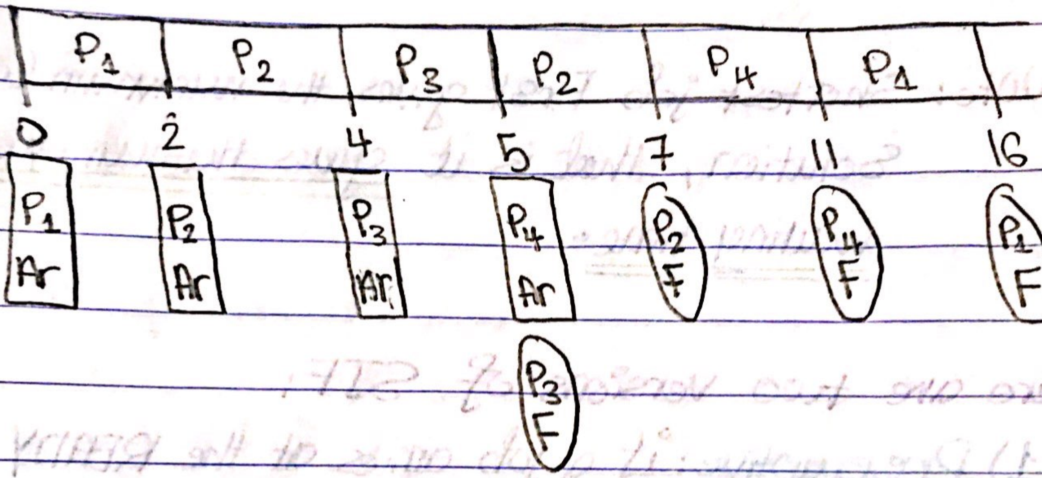
Shortest Remaining Time First (SRTF)

(2) Non-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 the running process & then switches to the new arriving process.

example:

Process	Arrival time	CPU time
P_1	0	~~7~~ 5
P_2	2	~~4~~ 2
P_3	4	~~1~~ 0
P_4	5	4

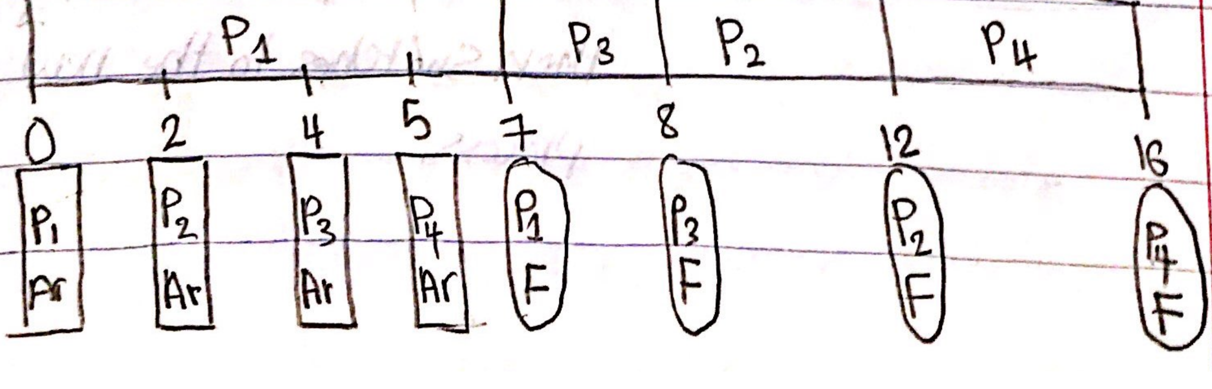
(a) Preemptive:



$$* \; ATT = \frac{(16-0) + (7-2) + (5-4) + (11-5)}{4} = \frac{12}{4} = 3$$

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

(b) Non-Preemptive:



$$* \; ATT = \frac{(7-0)+(12-2)+(8-4)+(16-5)}{4} = 8$$

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

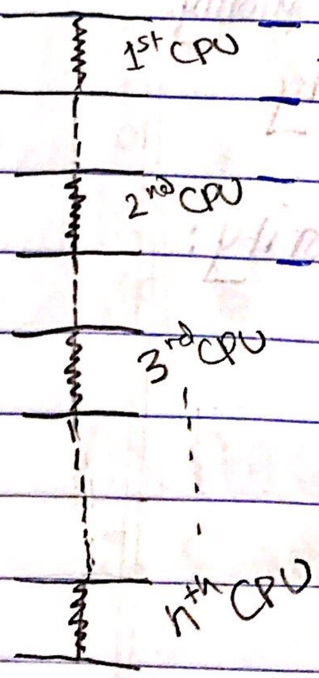
⚠ Problem : Starvation ∴ Solution: Aging

→ Aging: as time progresses, 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 the nth CPU burst.
- Y_n = estimated Length of the nth CPU burst.
- take a constant $0 \leq w \leq 1$
- Define the formula:

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

$$w = 0 \rightarrow Y_{n+1} = Y_n$$

$$w = 1 \rightarrow Y_{n+1} = T_n$$

→ Let us expand the equation:

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

$$+ (1-w) * [\_\_\_\_$$

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

$$+ (1-w)^3 w * T_{n-3} \_\_\_\_$$

* Substitute $w = 1/2$

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

lecture #13

Wednesday
March.7.2018

[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)

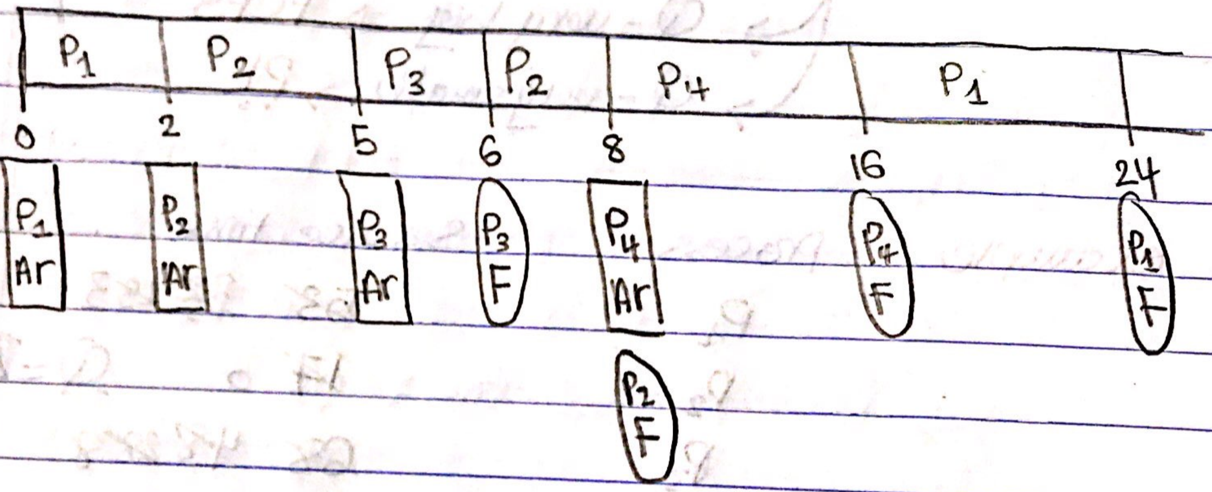
- There are Two versions of priority:
  - (a) Preemptive
  - (b) Non-preemptive

example: given the queue as follows:

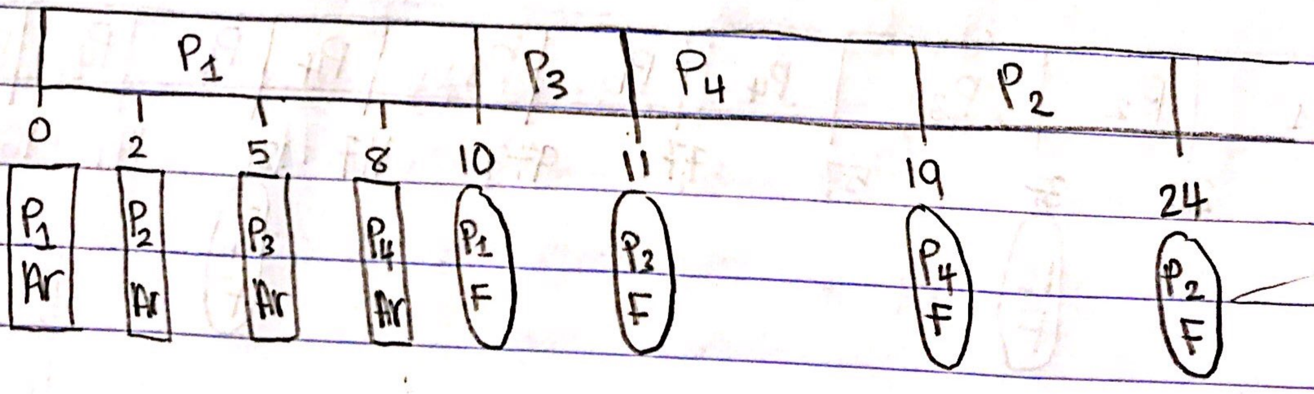
process	CPU burst	priority	arrival time.
P1	~~10~~ 8	1	10:00
P2	~~5~~ 2	2	10:02
P3	10	5	10:05
P4	8	4	10:08

⚠ high # = high priority.

(a) preemptive



(b) Non-preemptive



⚠ Problem: Starvation.

∴ Solution: Aging.

↳ As time progresses, increase the priority.

[4] Round Robin (RR):

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

If

→ Q = very Big → FCFS

→ Q = very small → ?!

example

Process	Service time
P_1	~~53~~ ~~33~~ ~~13~~ 3
P_2	~~17~~ 0
P_3	~~68~~ ~~48~~ ~~28~~ 8
P_4	~~24~~ 4

Q = 20

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_1	P_3
20	37	57	77	97	117	121	141	161	164	172

P_2 F

P_4 F

P_1 F

P_3 F

[5] Multi_Level Queues:

— Ready queue is divided in to 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:

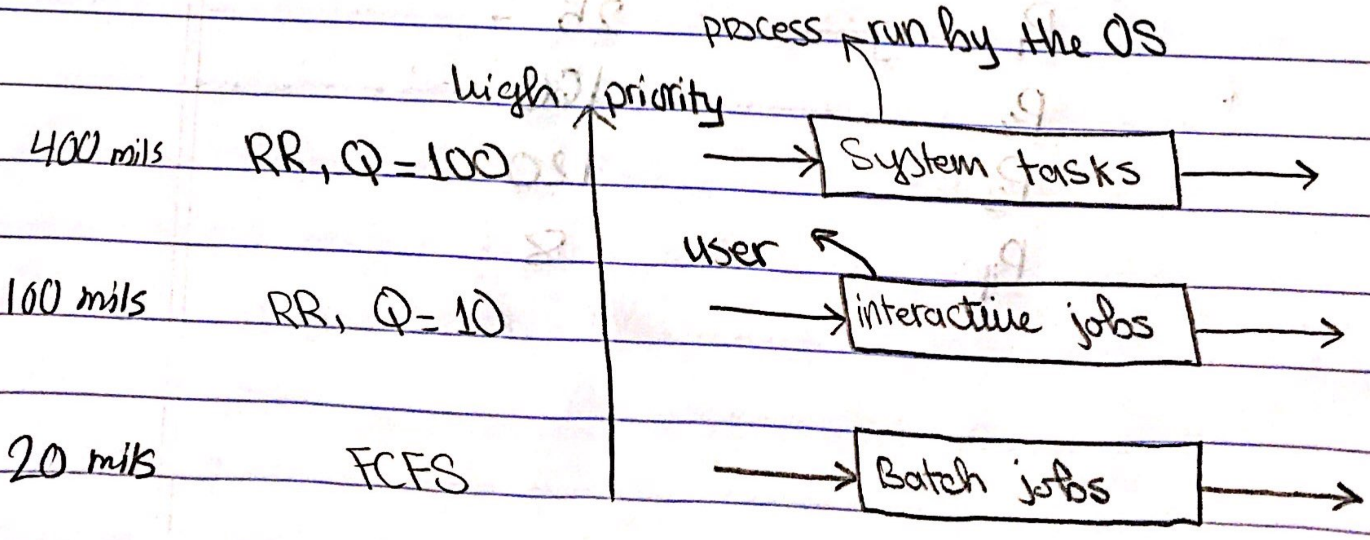
(1) time slice:

Each queue is assigned a chunk of time slice of CPU time, which scheduled among its processes.

(2) Fixed priority:

—serve all jobs in "system tasks".
—Then serve all jobs in "interactive"
—Then serve all jobs in "Batch"

↳ Preemptive.
↳ Non-Preemptive.



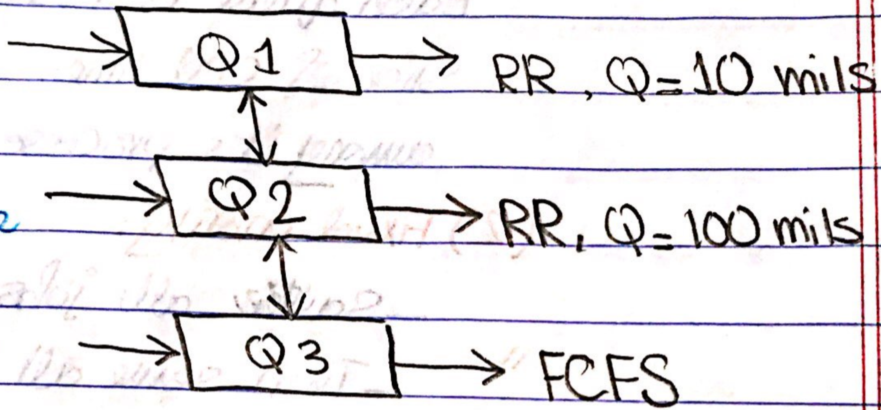
⚠ Problems starvation.

March 21.2018

[6] Multi-Level Feedback Queues

- Ready queue is divided into several queues.
- The process can move up & down between queues.

example:

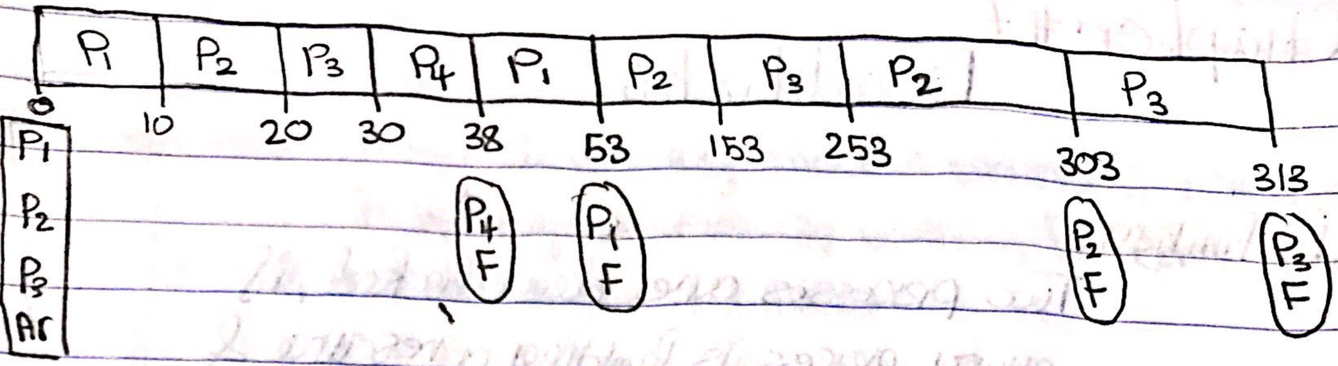


- Preemptive
- non_preemptive

→ new process enters Q1.

example:

Process	CPU_burst
P1	25
P2	160
P3	120
P4	8



☐ Algorithm Evaluation:-

(1) Deterministic model 'poor'
(2) Queuing Theory 'theoritical'
(3) Simulation. 'good'
* (4) Implementation 'best Algorithm for evaluation'

Chapter 6

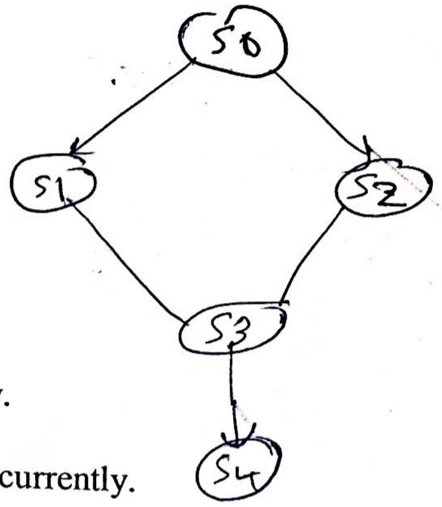
Concurrent Processes and Process Synchronization

Concurrent Processes

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

Precedence Graph:

using Precedence:



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,

statements (3) & { (1) or (2) }can't executed concurrently.
(4) & (3) can't executed concurrently.
(4) & { (1) or (2) or (3) } can't executed concurrently.

- But statements (1) & (2) can be executed concurrently.

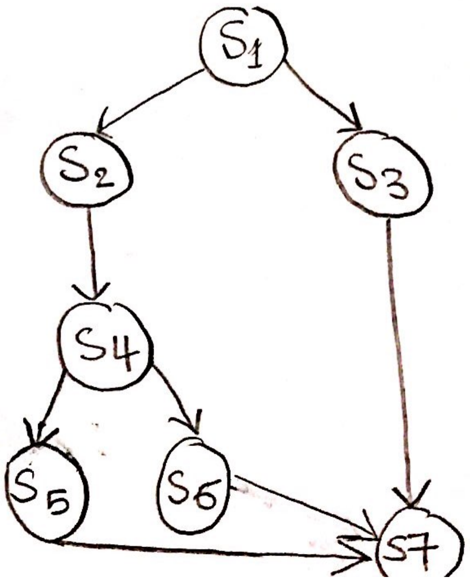
- So if we have multiple functional units in our CPU such as adders or we have multiprocessor system then statements (1) & (2) can be executed concurrently (in parallel).

(1)

Definition: A precedence graph is a directed graph whose nodes correspond to statements. An edge from 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 executed 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 and produce the same result?
- **Define:**
  $R(S_i) = \{a_1, a_2, \ldots, a_m\}$ 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.
  $W(S_i) = \{b_1, b_2, \ldots, b_n\}$ 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:

- $S : \overset{W}{c} = \overset{R}{a - b}$
  $R(S) = \{a, b\}$
  $W(S) = \{c\}$

- $S : w = c + 1$
  $R(S) = \{c\}$
  $W(S) = \{w\}$

- $S: x = x + 2$
  $R(S) = \{x\}$
  $W(S) = \{x\}$

- $S:$ read(a)
  $R(S) = \{a\}$
  $W(S) = \{a\}$

* $S:$ read(a)
$R(S) = \{a\}$
$W(S) = \{\ \}, \emptyset$

The Bernstein's conditions for concurrent statements are:

Given the statements S_1 & S_2 , then S_1 & S_2 can be executed concurrently if:

$$R(S_1) \cap W(S_2) = \emptyset$$
$$W(S_1) \cap R(S_2) = \emptyset$$
$$W(S_1) \cap W(S_2) = \emptyset$$

Example:

Given, $S_1 : a = x + y$
$S_2 : b = z + 1$

$R(S_1) = \{x, y\}$
$W(S_1) = \{a\}$
$R(S_2) = \{z\}$
$W(S_2) = \{b\} \leftarrow b$
$\{x, y\} \cap \{b\} = \emptyset$
$\{z\} \cap \{a\} \rightarrow a = \emptyset$
$\{a\} \cap \{b\} = \emptyset$

Example:

Given,

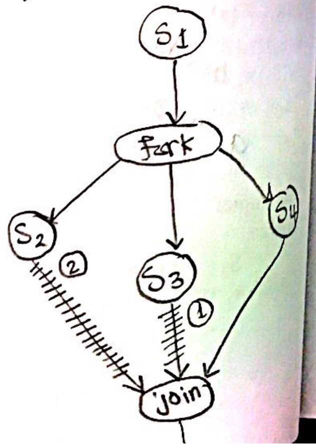
$S_3 : c = a - b$

$R(S_3) \cap W(S_2) = \{a, b\} \cap \{b\} \neq \emptyset$

Count = # of computations to join;

```
function join ;
{
    :
    Count = count - 1;
    if (count != 0)
      Quit (stop) computation;
}
```

Count = ~~3~~ ~~2~~ ~~1~~ 0



Fork & Join Constructs:

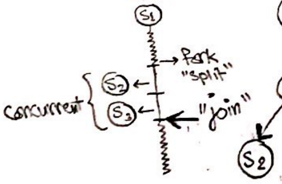
- Precedence graph is difficult to use in Programming Languages, so other means must be provided to specify precedence relation.

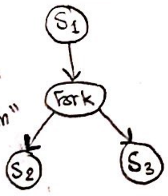
- The **Fork L** instruction produces two concurrent executions.
  - One starts at statement labeled L.→ label.
  - Other, the continuation of the statement following the fork instruction

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

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

Concurrent

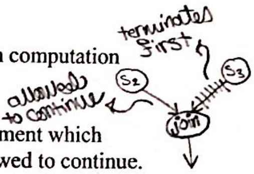




(*) When the fork L statement is executed, a new computation is started at S_3 which is executed concurrently with the old computation, which continues at S_2. That is, the fork statement splits one single ~~corporation~~ Computation into two independent computation; hence the name Fork

- The join instruction recombine two concurrent computation. Each computation must ask to be joined.

  Since the two computations executes at different speeds, the statement which executes the join **first** is terminated first, while the second in allowed to continue.



- For 3 computations, two in terminated while the third continues.

- If **count** is number of computations to join, then the execution of the **join** has the effect

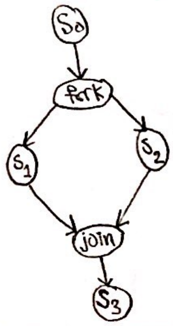
  count = count – 1;
  If count ≠ 0 then quit (quit this computation)

→ join instruction (function) Can't be executed concurrently, but, one process at a time.

The join statement for two computations is executed atomically, i.e. can't be executed concurrently but in a sequential manner, because this might affect **count** giving a wrong result.
For example, if both decrement **count** at same time then count = 0, and the computation **dues not quit.**

- For two processes:

  Count =2
  Fork L₁;



Very important Note:- join statement must be executed atomically. that is, one process at a time, that is, can't be executed concurrently.

```
      :
      S1 ;
     goto L2;
  L1 : S2
  L2 : join count
```

Concurrent

(2)

- Let us go back to out four statements in the beginning of this chapter. Using fork & join, this will look lila:

```
       count =2;
       Fork  L1;
      a = x+y;
      goto  L2;
     L1 : b=z+1;
     L2 : joint count;  -> join
       c  =a-b;
       w  =c+1;
```

Concurrent

fork → a= x+y ; b= z+1 ; } Concurrent.
c= a-b ;
w= c+1 ;

- For the precedence graph earlier:

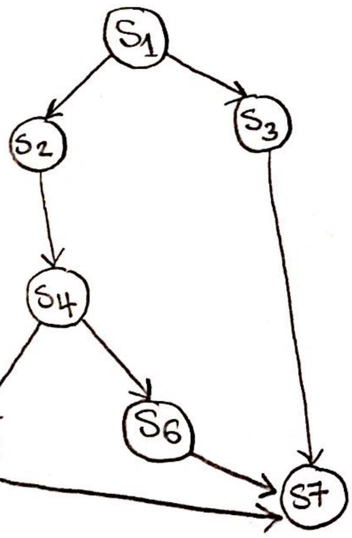
```
      S1 ;
      count  = 3
      Fork L1; -> Concurrent
      S2;
      S4
      Fork  L2;
      S5;
      goto L3;
   L2 : S6;
      goto L3;
   L1 : S3;
   L3 : join count ;
      S7;
```

Concurrent

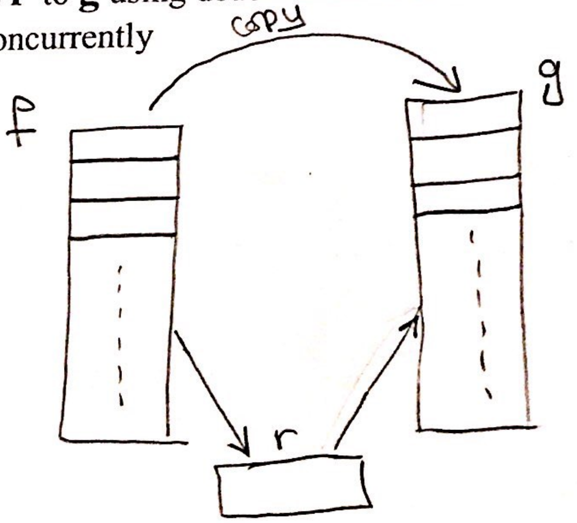
Concurrent

Fork L1 S3

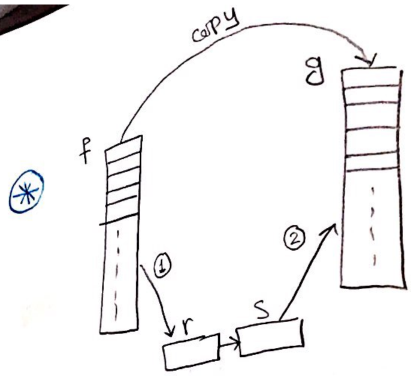
Fork L2 S6



- 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)
```



* two records r & s:

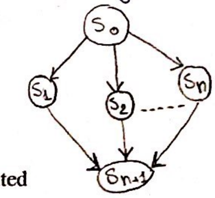
```
read (f,r);
while (not eof (f)) do
      begin
        count = 2;
        s: = r;
        Fork L1;
        [Write (g, s);  ⎫
        [goto L2;       ⎬ concurrent statements.
Concurrent [L1: read (f,r);  ⎭
        L2: join count;
          End;
        Write (g,r);
End;
```

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 (3)

Example:

```
S0;
Parbegin
    S1;  ⎫ these statements are executed concurrently.
    S2;  ⎪
     :   ⎪
    Sn;  ⎭
Parend; →
Sn+1;
```



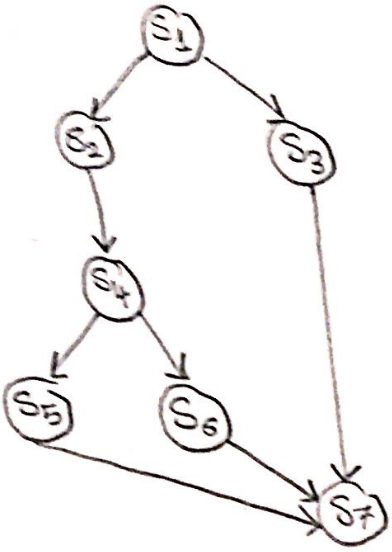
- All statements enclosed between parbegin and parend can be executed concurrently

(*) In our pervious example,

```
parbegin
   a  = x+y;  } concurrently.
   b  = z+1;
parend ;
   c =a-b;   } not concurrently.
   w =c+1;
```

(*) In the example:

```
$S_1$;
parbegin
   $S_3$;
   begin
      $S_2$;
      $S_4$;
      parbegin
        $S_5$;
        $S_6$;  > Concurrent .
      parend;
   end;
parend;
$S_7$;
```



(*) For the files copying files :

```
begin
   reset (f);
   read (f, r);
   while (not eof (f)) do
      begin
        S = r;
   ->   parbegin
           write (g, s);  } concurrent
           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 . . ; //can be of 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 the 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.

Bounded-Buffer

Data Structure used:

```
item . . ; //can be of any data type
item buffer[n], nextp , nextc;
int in = 0, out = 0;
int counter = 0;
```

* with Counter

* with Counter

Producer:

```
do
 {  ...
    produce an item in nextp
    ...                       buffer is full
    while (counter == n)
       no-op; // busy waiting
    buffer[in] = nextp;
    in = (in + 1) % n;
    counter = counter + 1;
 }
while true;
```

Critical section

Consumer:

```
do                             buffer is empty
{  while (counter == 0)
      no-op; // busy waiting
   nextc = buffer[out];
   out = (out + 1)% n;
   counter = counter - 1;
   ...
   consume the item in nextc
   ...
}
while true;
```

Critical section

- Counter = counter + 1; could be implemented as

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

Producer

Concurrently

- Counter = counter - 1; could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

consumer

- 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 {count = 6 }
S5: consumer execute counter = register2 {count = 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.

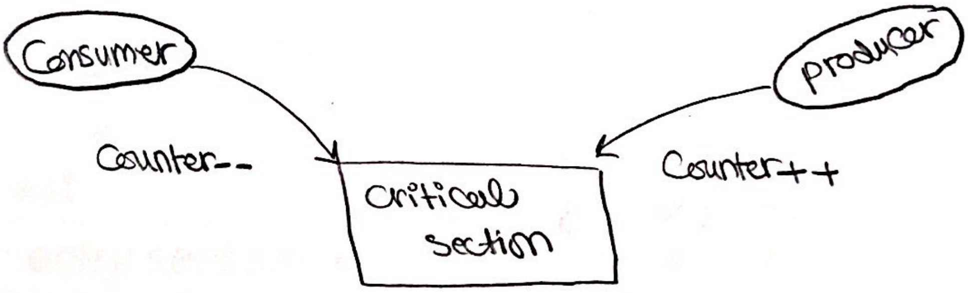
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

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

must be executed *atomically*.

to access a shared data Concurrently the shared data must be accessed 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 resources.* (i.e: Counter in producer_consumer)
- *If there is no controlled access to shared data, it is possible to obtain an inconsistent view of this data.*
- *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.

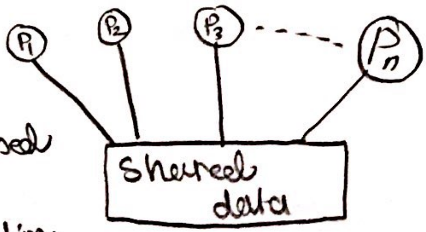
- *n* processes all competing to use some shared data

- Each process has a 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          <-- شرط دخول ال critical section
        critical section
    exit section           <-- ما العمل عند الخروج من ال critical section
        remainder section
until false;
```



Shared data is Accessed Atomically

↳ one process at a time.

Solution Requirements:

① **Mutual Exclusion.** If process Pi 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 exist 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 the critical section and 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 sections after a process has made a request to enter its critical section and before that request is granted.

"there's a bound for each process on the amount of time it needs to get the critical section "

- Assume that each process executes at a nonzero speed. ← حتى لا تحجز ال critical section حالك فيه، تبقى تعيق ال Process waiting.
- No assumption concerning relative speed of the n processes.

Solution to Critical Section Problem

Types of Solutions

- *Software solutions* Programming
    - 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* Ready functions to support the programmer
    - Extending hardware solutions to provide some functions and data structure support to the programmer.

SOFTWARE SOLUTION

- Only 2 processes, P_0 and 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 a value of either 0 or 1
             //if turn = i, P(i) can enter it's critical
section
```

Process P_i → works based on turns so concurrency isn't used.

```
do
{
 while (turn != i) /*do nothing*/ ;
   critical section
 turn = j;
   remainder section
}
while (true)
```

busy waiting

Process P_j

```
do
{
  while(turn!=j)
    do nothing ;
  critical section;
  turn = i ;
  reminder Section
}
```

- **Mutual exclusion** ok

- **Bounded waiting** ok - each only waits at most 1 go.

- **Progress not good** - 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.

Algorithm 2

- *Shared variables*

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

Process P_i

```
do
  {  flag[i]= true;
       while (flag[j]) /*do nothing*/ ;

    critical section

  flag[i]=false;

   remainder section
 }
while (true)
```

```
Process Pj
do
{ Flag[j] =true;
  while (Flag[i])
    do nothing;
 }
```

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

- Infinite loop

Algorithm 3

Combined shared variables of algorithms 1 and 2.

```
  int turn;        //turn can have a value of either 0 or 1
  boolean flag[2];       flag[0] = flag[1] = false;
// if flag[i] == true, P(i) ready to  enter its  critical      section
process Pi
  do
   { flag[i]= true;
     turn = j;
     while (flag[j] && turn==j) /*do nothing*/ ;

     critical section

    flag[i]=false;

     remainder section
  }
  while (true)
```

Process P_0 → Concurrent ← **Process P_1**

```
do
{ flag[0]= true;
  turn = 1;
  while (flag[1]  && turn==1)
      /*do nothing*/  ;

critical section

flag[0]=false;

remainder section

} while (true)
```

```
do
{flag[1]= true;
 turn = 0;
 while flag[0]  && turn==0)
     /*do nothing*/ ;

critical section

flag[1]=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).

Bakery Algorithm

→ generalization of the solution for ⓝ processes.

Introduction

This algorithm solves the critical section problem for ***n*** processes in software. The basic idea is that of a bakery; customers take numbers, and whoever has the lowest number gets service next. Here, of course, "service" means entry to the critical section.

Critical section for *n* processes

- Generalization for n processes.
- Each process has an id. Ids are ordered.
- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P $_i$ and P $_j$ receive the same number, if *i* < *j* , then P $_i$ is served first; else P $_j$ is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <= lexicographical order (ticket #, process id #)
    - (a,b) < (c,d) if a < c or if a = c and b < d
    - max(a_0, . . . , a_{n-1}) is a number, *k* , such that *k* >= a_i for i = 0, . . . , n - 1
- Shared data

```
1  boolean choosing[n]; //initialise all to false
2  int number[n];  //initialise all to 0

3  do
4  { choosing[i] = true;
5  number[i] = max(number[0], number[1], ...,number[n-1]) + 1;
6  choosing[i] = false;
7  for(int j = 0; j < n; j++)
8     { while (choosing[j]== true)
9            /*do nothing*/
10       while ((number[j]!=0) && (number[j],j)< (number[i],i))
11            /*do nothing*/
12    }

13       critical section

14 number[i] = 0;

15       remainder section

   } while (true)
```

Comments

lines 1-2: Here, *choosing*[*i*] is true if P_i is choosing a number. The number that P_i will use to enter the critical section is in *number*[*i*]; it is 0 if P_i is not trying to enter its critical section.

lines 4-6: These three lines first indicate that the process is choosing a number (line 4), then try to assign a unique number to the process P_i (line 5); however, that does not always happen. Afterwards, P_i indicates it is done (line 6).

lines 7-12: Now we select which process goes into the critical section. Pi waits until it has the lowest number of all the processes waiting to enter the critical section. If two processes have the same number, the one with the smaller name - the value of the subscript - goes in; the notation "(a,b) < (c,d)" means true if a < c or if both a = c and b < d (lines 9-10). Note that if a process is not trying to enter the critical section, its number is 0. Also, if a process is choosing a number when P_i tries to look at it, P_i waits until it has done so before looking (line 8).

line 14: Now P_i is no longer interested in entering its critical section, so it sets *number*[*i*] to 0.

Drawbacks of Software Solutions

- **Complicated to program**
- **Busy waiting (wasted CPU cycles**
- **It would be more efficient to *block* processes that are waiting (just as if they had requested I/O).**

HARDWARE SOLUTION

Hardware Solution <u>*Disable Interrupts*</u>

On a uni-processor, you can get mutual exclusion by locking out interrupts. Observations:

- You can only afford to do this for a little while, so you don't lose any interrupts (of course in general you don't want to protect expensive things with spin locks).
- Nothing else works if you're sharing memory with a device you sure can't use a spin lock! (DEADLOCK).
- Correct solution for a uni-processor machine, but this doesn't work on multiprocessors, the solution is not correct.
- During critical section multiprogramming is not utilized - performance penalty.

```
Repeat
  disable interrupts
  critical section
  enable interrupts
  remainder section
Forever
```

Hardware Solution <u>*Test and Set*</u> → must be executed Atomically

Use better (more powerful) atomic operations:

- **Test and modify the content of a word <u>atomically</u>.**

```
boolean Test_and_Set(Boolean & target)

 {boolean test = target;
   target = true;
  return test;
 }
```

→ Call by reference to return the value of target.

- Shared data: `boolean lock = false;`

Process P_i

```
do
{ while (Test-and-Set(lock))
        /*do nothing*/ ;
        critical section
   lock = false;
       remainder section
}while (true)
```

OPERATING SYSTEM SOLUTION

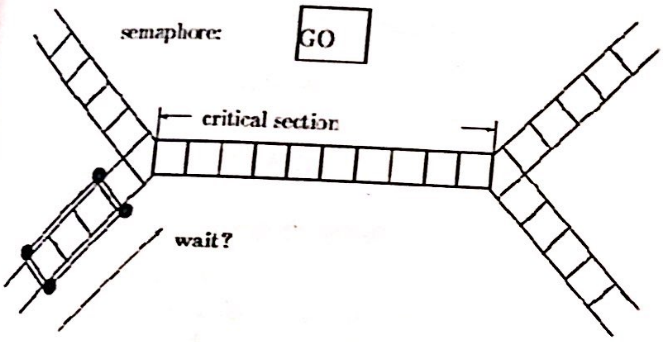
Semaphores

→ checks if the critical section is Empty or not
"close"

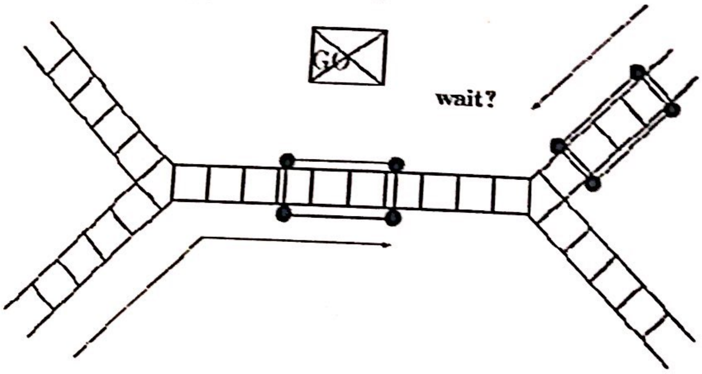
Semaphore: wait and signal

↳ "opens the critical section"

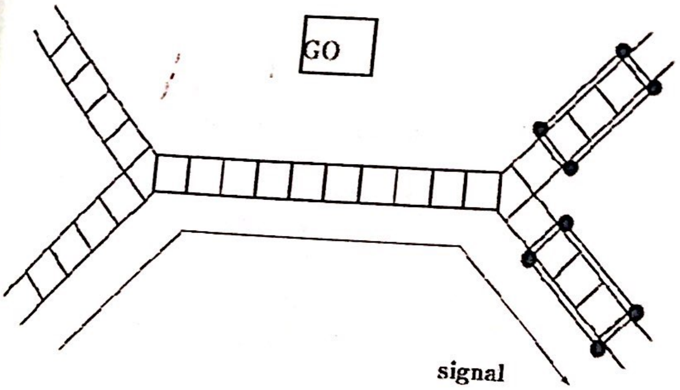
(1) Train 1 Arrives



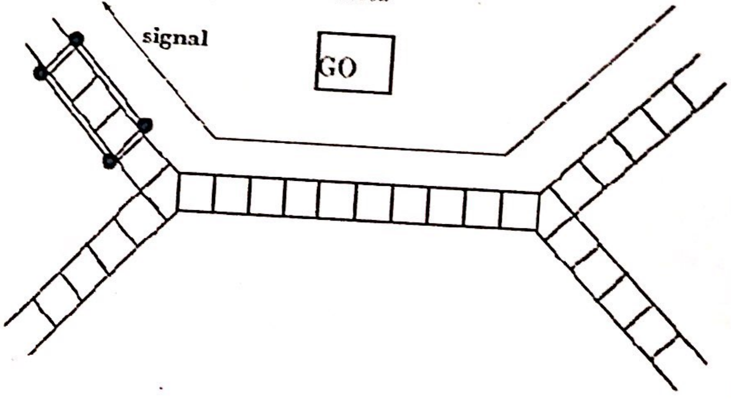
(2) Train 2 Arrives



(3) Train 1 Leaves



(4) Train 2 Leaves



Semaphore S - integer variable
↳ int S=1; - can only be accessed via two indivisible (**atomic**) operations

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

signal(S)  : S = S + 1;
```

mutual exclusion

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

Note:- The wait and signal instruction must be executed atomicaly.

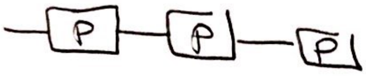
Problem (semaphore): busy waiting.

Semaphore Implementation

- Define a semaphore as a record/structure

```
struct semaphore
 { int value;
   List *L;    //a list of processes
 }
```

↳ Pending



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

```
wait(S)
{ S.value = S.value -1;
  if (S.value <0)
    { add this process to S.L;
      block;
    }
}

signal(S)
{ S.value = S.value + 1;
  if (S.value <= 0)
    { remove a process P from S.L;
      wakeup(P);
    }
}
```

Classical Problems of Synchronization

- Bounded Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

Bounded Buffer Problem

- Shared data

```
char item;                    // could be any data type
char buffer[n];
semaphore full = 0;           // counting semaphore
semaphore empty = n;          // counting semaphore
semaphore mutex = 1;          // binary semaphore
char nextp, nextc;
```

mutual execlusion.

- Producer process

```
do
{ produce an item in nextp
  wait (empty);  → checks if Buffers is 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;
}
```

Readers-Writers Problem

- Shared data

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

- Writer process

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

- Reader process

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

Dining Philosopher Problem

- Shared data

```
semaphore chopstick[5];
chopstick[] = 1;
```

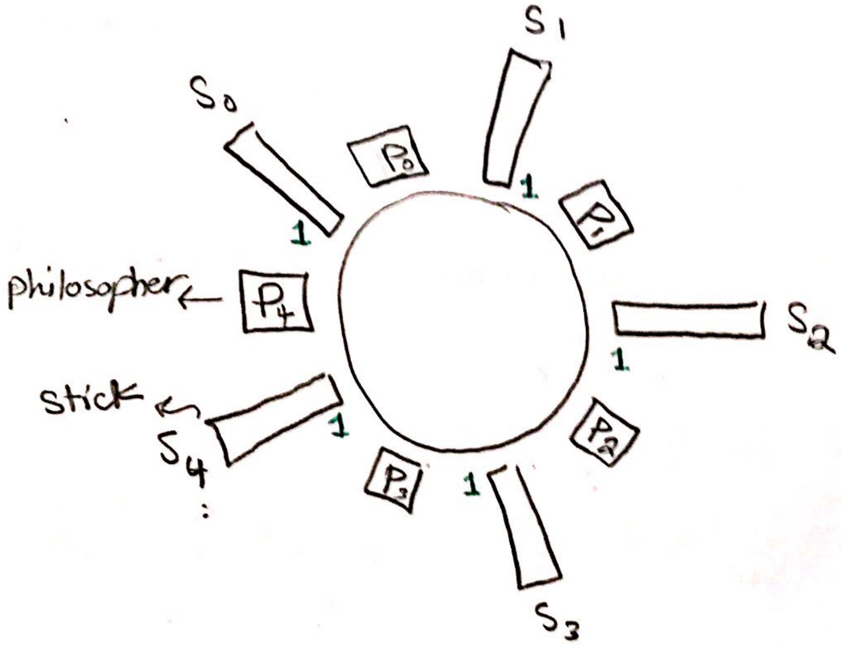
↳ available

- Philosopher *i*:

```
do
{ wait (chopstick[i]);
  wait (chopstick[i+1 mod 5]);
     eat;
  signal (chopstick [i]);
  signal (chopstick [i+1 mod 5]);
     think;

}
while (true)
```

1: available



⚠ Problems:

(1) Deadlock.

(2) Starvation.

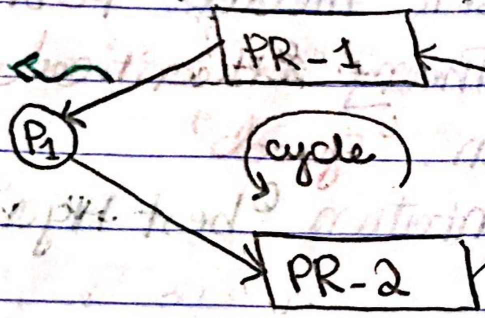
Chapter #7

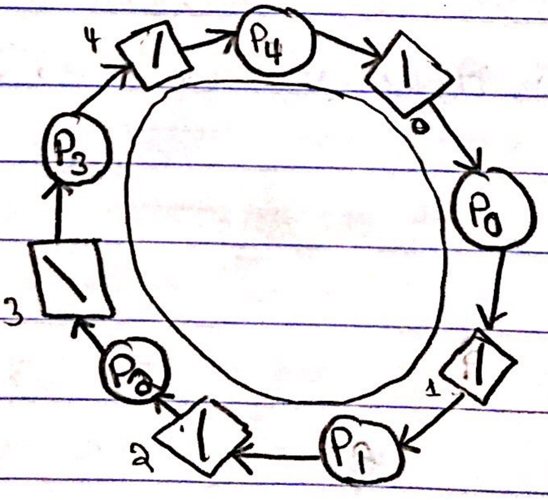
DeadLocks.

Definition:

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

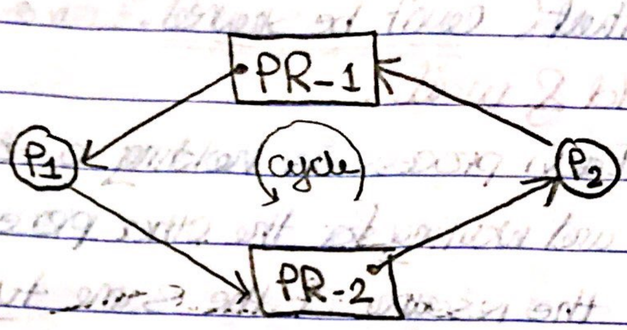
Printer is allocated to the process





Monday
April 16. 2018

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 resource types $\overset{HD}{R_0}, \overset{tapes}{R_1}$ ___, $\overset{printers}{R_{n-1}}$

— we have W_j instances of each resource type

$\overset{5}{W_0}, \overset{2}{W_1}$ ______, $\overset{3}{W_n}$

— Each process use the resources in the following order:

* Requests the resources.
* uses the resources.
* Releases the resources.

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.

[i] Necessary Conditions:

4 necessary conditions must hold simulatanously in order for a deadlock to occur.

(1) Mutual Exclusion:

The resource type must be used exclusively that's can't be shared. "for more than one process at a time"

(2) Hold & wait:

Each process is holding a resource type and waiting for the other process to release the resource of the same type.

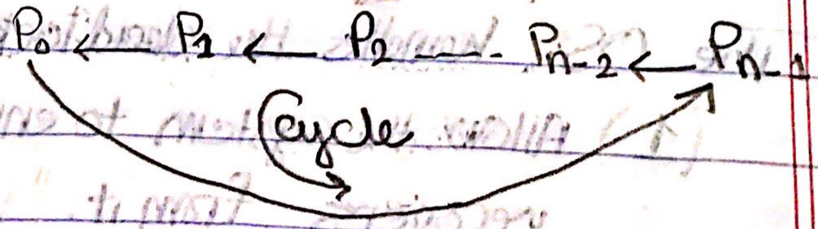
(3) No Pre-emption:

Can't remove any of the resources.

(4) Circular wait: "cycle"

There exists a sequence of processes $\langle P_0, P_1, P_2, \text{---}, P_{n-1} \rangle$ such that:

- P_0 is waiting for P_1 to release its resources.
- P_1 is waiting for P_2 to release its resources.
- P_{n-2} is waiting for P_{n-1} to release its resources.



Resource Allocation Graph:

Generally, a graph $G = (V, E)$

V = set of vertices.

E = set of edges.

→ In the deadlock case:

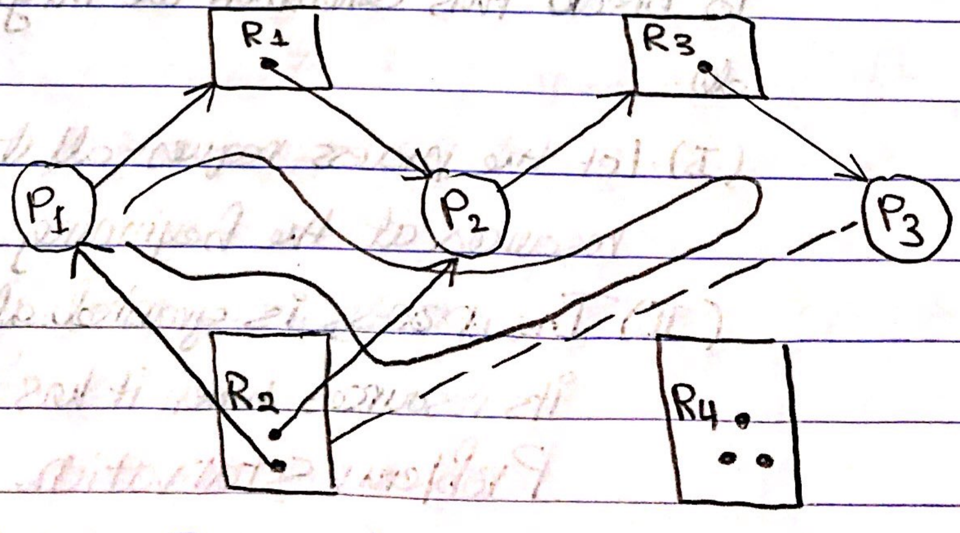
$V = \{$ P: Process.
R: resource type.

$E = \{$ (P_i, R_j): Process P_i is requesting one instance of resource type R_j
(R_j, P_i): one instance of resource type R_j is allocated or given to process P_i.

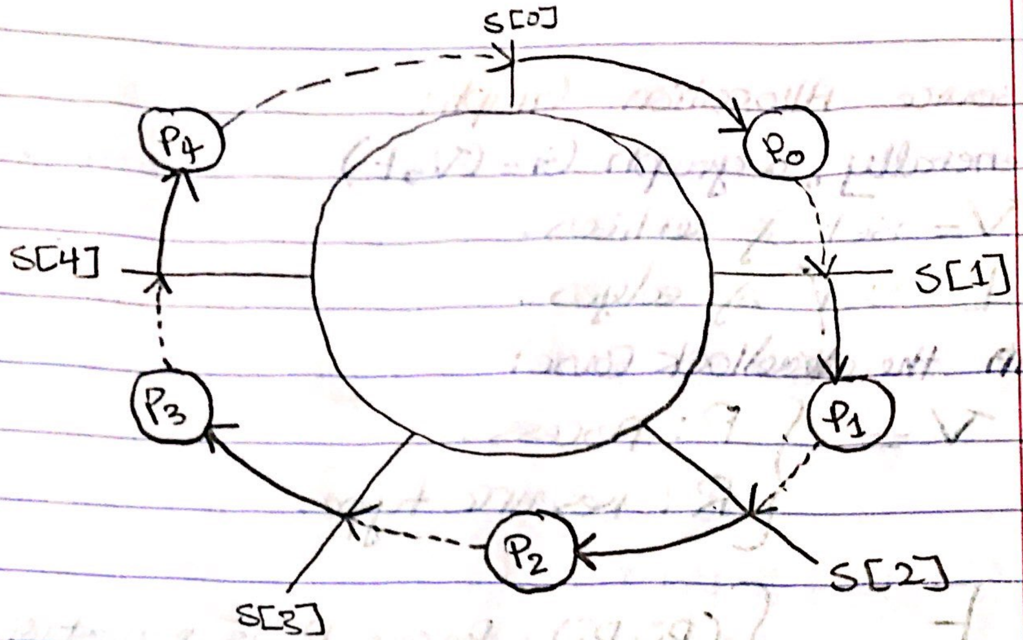
Example: $P = \{P_1, P_2, P_3\}$

$R = \{R_1(1), R_2(2), R_3(1), R_4(3)\}$

$E = \{(P_1, R_1), (P_2, R_3), (R_1, P_2), (R_2, P_2), (R_2, P_1)$
$(R_3, P_3)\}$



→ Assume P_3 demands an instance of R_3.



Deadlock prevention:-

To make sure at least one of the four 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 requests a resource which is not available, it must release the resources it has.

Problem: low system utilization. "poor performance", in addition to starvation.

4- **Circular wait:**

① Card reader.
② Hard disk.
③ Tape.
④ Printer.

```
Process Pi ;
Semaphor int s[i] = { 1 , 1 , 1 , 1 }
Repeat {
        Think ;
        wait(si) ; → wait(S min(i, ((i+1)%5)))
        wait( s((i+1)%5) ) ; → wait(Smax(i,((i+1)%5)))
        Eat ;
        Signal (s((i+1)%5) ) ;
        Signal (si);
      }until False.
```

4 Deadlock Avoidance:

Definition: A system is in a safe state, if there exists a sequence of processes $<P_0, P_1, P_2, ---, P_{n-1}>$ such that:

P_0 can take all available resources, execute & finish.

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

P_2 can take all available resources, & resources released by P_0, P_0, execute & Finish.

⋮

P_{n-1} can take all available resources and resources released by $P_0, P_1, P_2 -- P_{n-2}$, execute & Finish.

Definition: If there's such a sequence, then the system is safe, NO deadlock.

example: A system with 12 tape units and 3 processes, A snapshot at the system looks like:

Process	max needs	allocated	current needs
P_0	10	5	5
P_1	4	2	2
P_2	9	~~2~~ 3	~~7~~ 6

the available at this time: $\boxed{3}$ → 12 - Σallocated(9)

- Is the system safe.

available: ~~3~~ ~~4~~ ⑤ <P1, P0, P2>

~~5~~ ~~0~~ ⑩

~~10~~ ~~3~~ ⑫

Safe ✓

- Assume, process 2 demanded extra tape & the OS granted the request. is the system safe?

the available at this time: $\boxed{2}$

available: ~~2~~ 4 <P1, ??

ⓧ No safe sequence - deadlock

Process	Allocation			max			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	3 ~~0~~	~~0~~ 3	2	4	3	3	~~4~~ 1	~~3~~ 0	1

- IS the System Safe ?! Is there a safe sequence?!

Available

~~3~~ 3 2

~~5~~ ~~3~~ ~~2~~

~~7~~ ~~4~~ ~~3~~

~~7~~ ~~5~~ ~~3~~

~~10~~ ~~5~~ ~~5~~

$\boxed{10 \quad 5 \quad 7}$

A	B	C	
10	5	7	total

<P1, P3, P0, P2, P4>

- Assum process 4 requested (3, 3, 0), Is the system Safe?!

available

0 0 2 ⚠ Not Safe.

Banker's Algorithm:-

- We will Consider only one resource type for simplicity.
- Each process declares its maximum needs at the Beginning.
- When a process request a resource it might has to wait.
- When the process gets all resources it must release them in a finite time.

➡ Data Structure used :

- Array max,

  Max[i] = j, means Pi will need a max of j units of the resource.

- 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,

Available = W , W is the [illegible] number of units available of the resource.

→ Algorithm (Bankers)

```
1- let W = available ;
2- Define an array K[i] = 1 ∀ i = 0,1, ... , n-1
3- Find an 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 ∀ i [illegible] then
       System is SAFE
   else
       System is UnSAFE.
```

K
1
1
1
1
1
1

example:

Process	max	Allocation	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

available (W = 3)

~~3~~ ~~5~~ ~~10~~ 12

K
~~1~~ 0
~~1~~ 0
~~1~~ 0

Chapter #8

Memory Management.

'Ordinary Memory Management'

↳ means: all programs must be admitted (allocated to memory before execution starts)

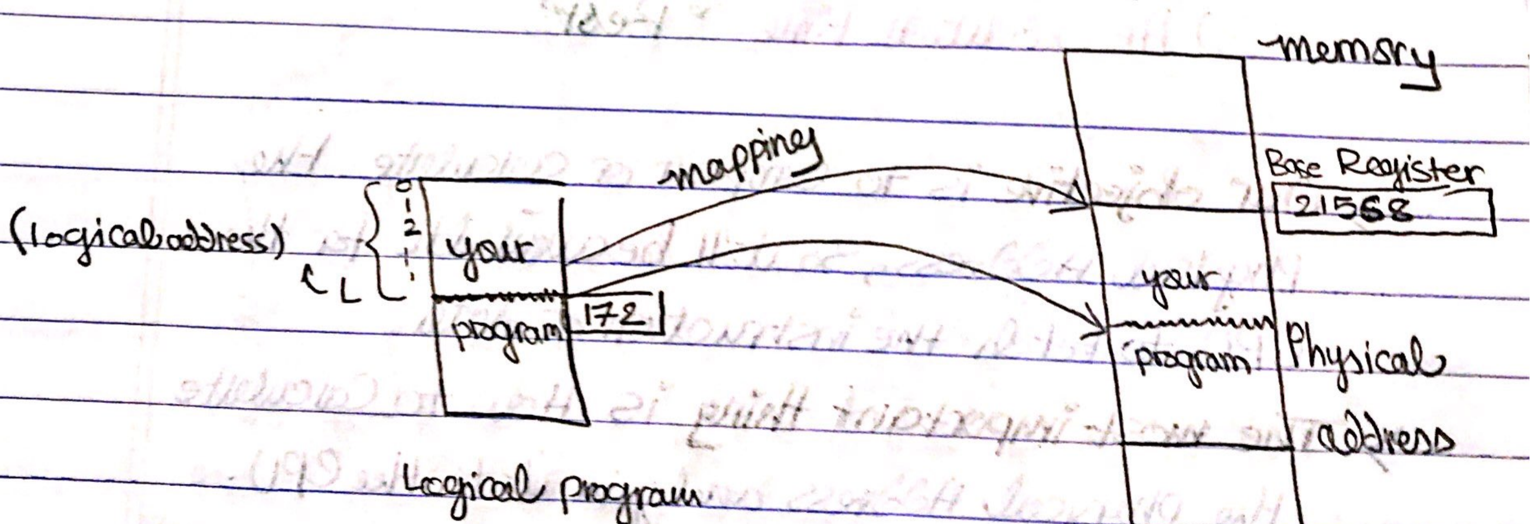
☐ Logical Address VS. Physical Address:-

* **Logical Address:**

The address seen in your program, It's the offset of the address in the program.

* **Physical Address:**

It's the actual address in memory.



$PA = LA + \text{Base Register}$

$PA = 172 + 21568$

$= 21740.$

⧉ Binding Times:

When the OS determines the physical addresses?

(1) At Compilation time.

The PAs are assigned at the beginning, Which means the program must be Loaded into memory every time at the same location. Also notice that the program can't change its location during execution.

(2) At Loading time.

The PAs are decided when the program is Loaded into memory.

⚠ Problem: the program can't be moved during execution.

(3) At execution time. "Best"

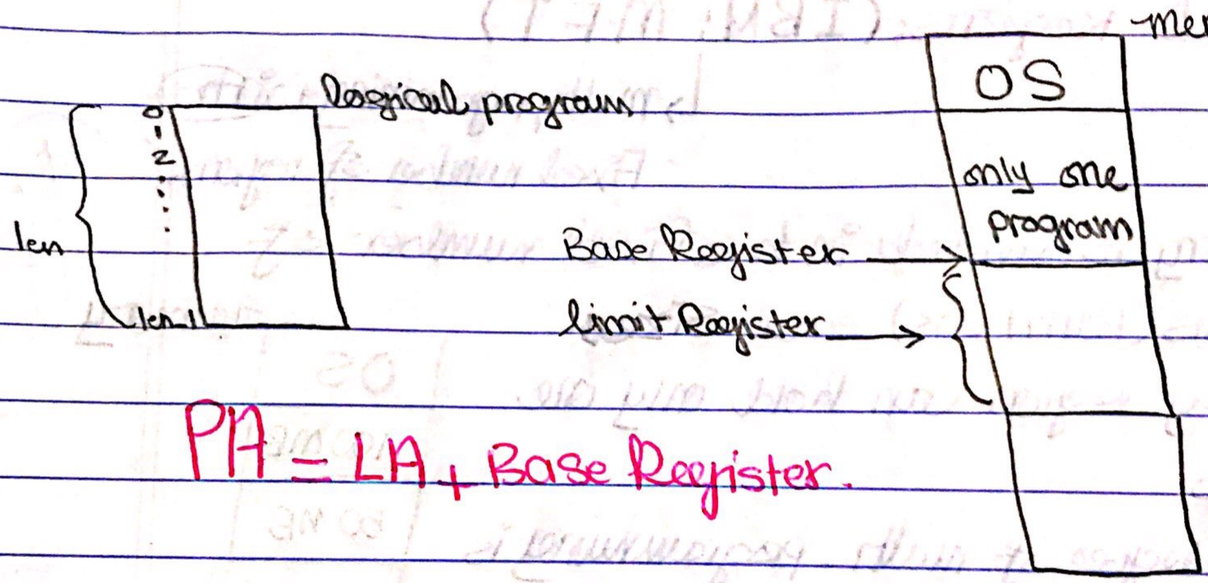
→ Our objective is to Computer or calculate the Physical Address, so it'll be available for the CPU to fetch the instruction or data.

→ The most important thing is How to Calculate the physical Address and give it to the CPU. in memory management

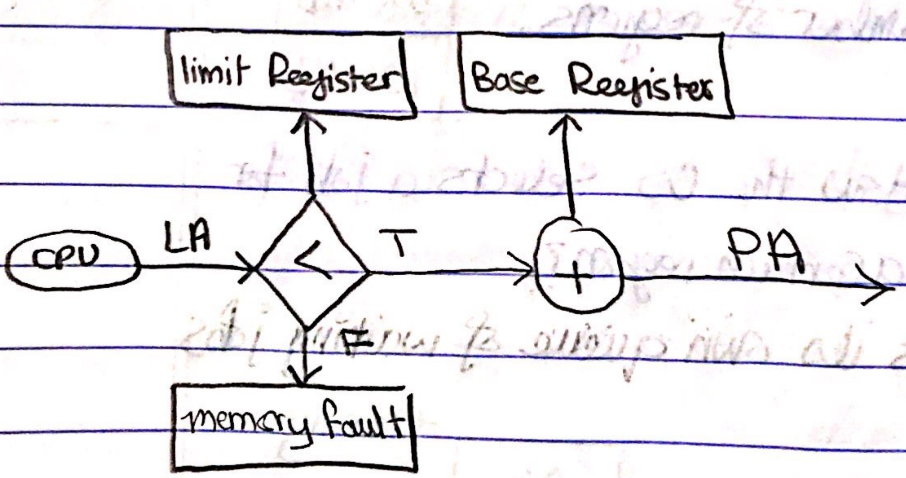
تابع

[1] Contiguous Allocation _ 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 into it.
- **Hardware support means:** What data structure we need to compute the PA ?!
  Answer: we need Base & limit Registers.



PA = LA + Base Register.



There are two various from this MM algorithms:-
(1) Fixed Regions.
(2) Dynamic (variable) Regions.

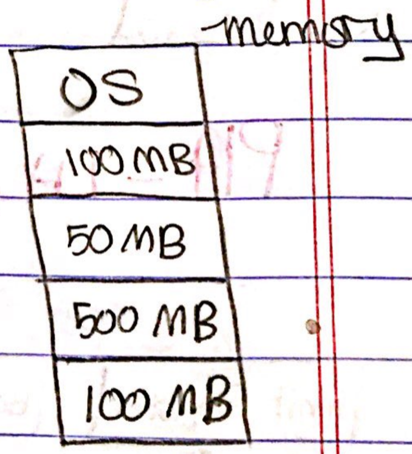
Multiple partitions:
- Base & Limit.
- PA = Base + LA.

} Revision.

(1) Fixed Regions: (IBM: MFT)

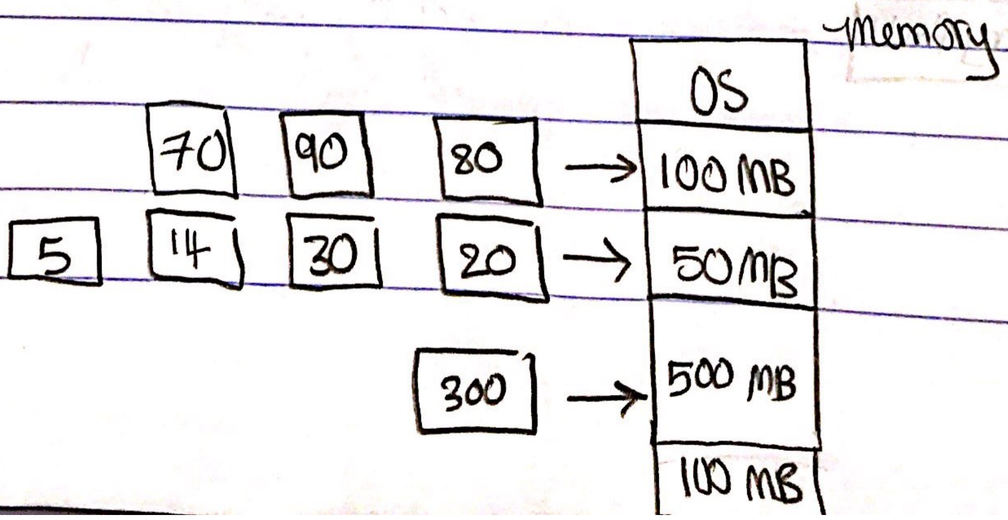
↳ multiprogramming with Fixed number of regions.

- memory is divided in to a fixed number of regions (Partitions) and Sizes.
- every region can hold only one Job
- The degree of multi-programming is bounded By the number of regions.

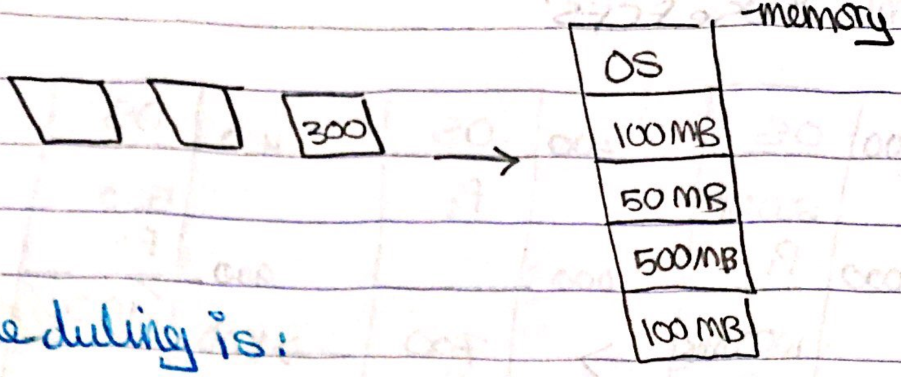


Job scheduling, How the OS selects a job for a Certain region?!

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



(b) There's only one queue of waiting jobs.



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

⚠ Problems: **Internal Fragmentation:**
The remaining unused memory inside the region.

External Fragmentation: The unused region which is small to fit any available job.

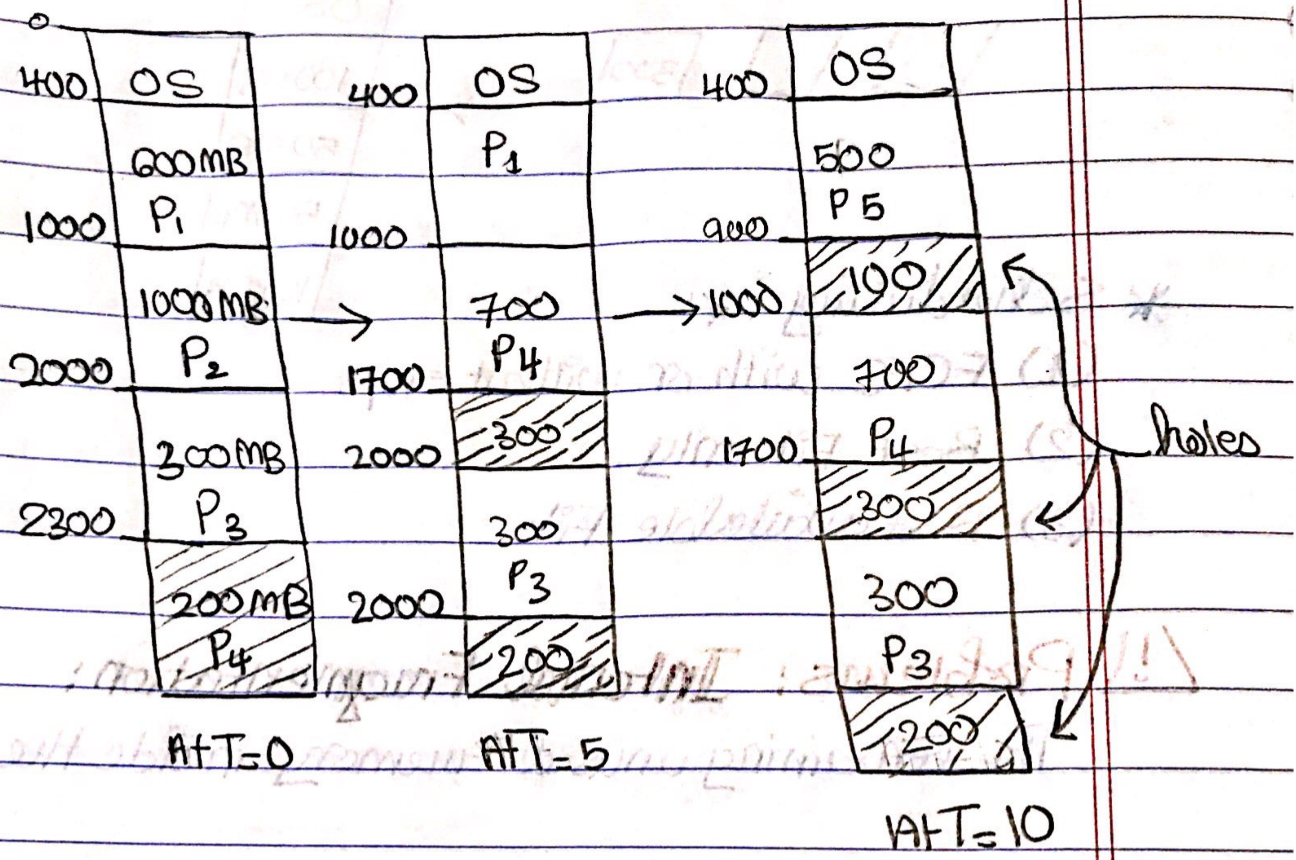
(2) Dynamic (variable) regions:

example:

assume we have the following queue of jobs:

Process	memory needed	time in memory
P_1	600 MB	10
P_2	1000 MB	5
P_3	300 MB	20
P_4	700 MB	8
P_5	500 MB	15

Assume we have memory 2500 MB, OS is reserving 400 MB. "Use FCFS"



- after a while, memory will contain:
  - allocated regions.
  - Set of holes "Eternal Fragmentation"

* Job Scheduling: How we select a hole "variable region" for a process?!

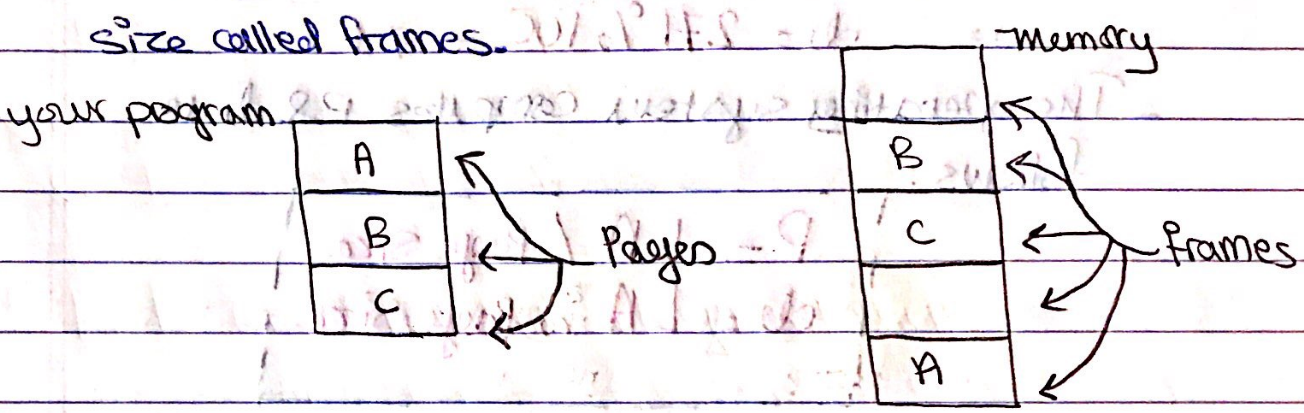
(1) First Fit
(2) Best Fit.
(3) Worst Fit.

⚠ Problem: external Fragmentation. "holes"

∴ Solution: Compaction.

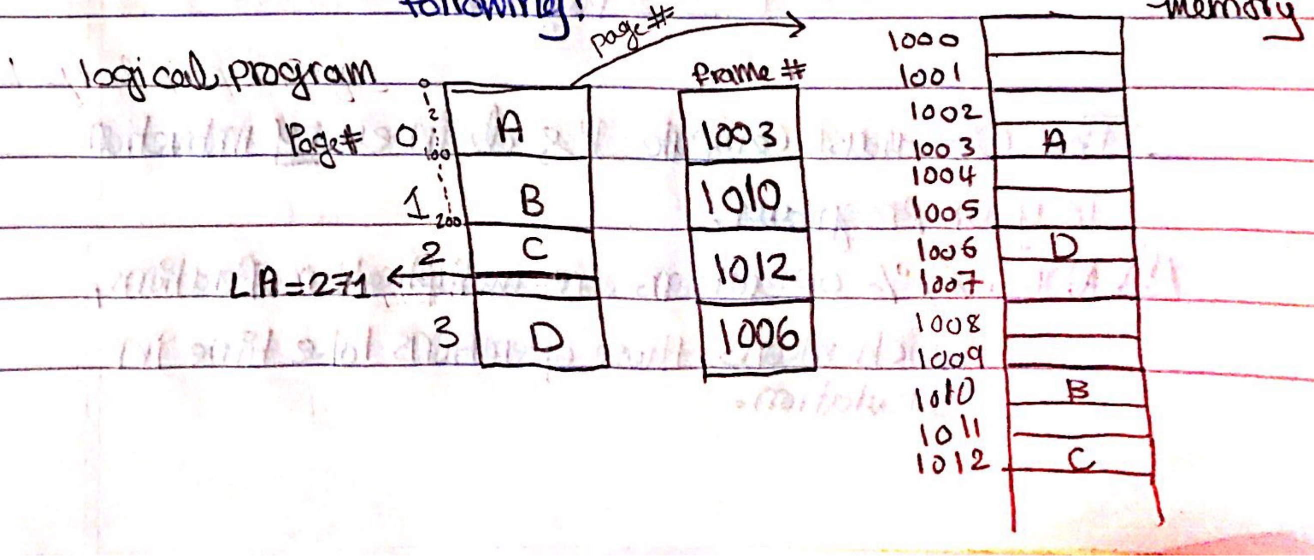
[2] Non-Contigious - Paging:

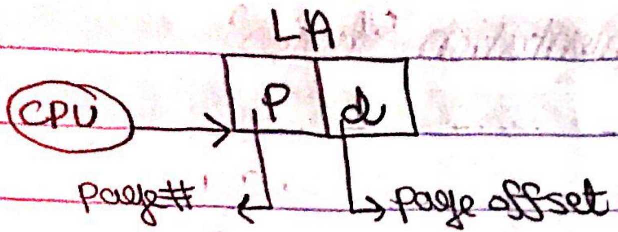
- The logical program is divided into equal size partitions called pages.
- Memory is divided in to partitions of the same size called frames.



- Hardware support needed to compute the PA: we need what's called "page table", which is a table that contains the "frame numbers" of the pages in the logical program.

example: Assume page size = 100 bytes, given the following:





example: take the LA = 271

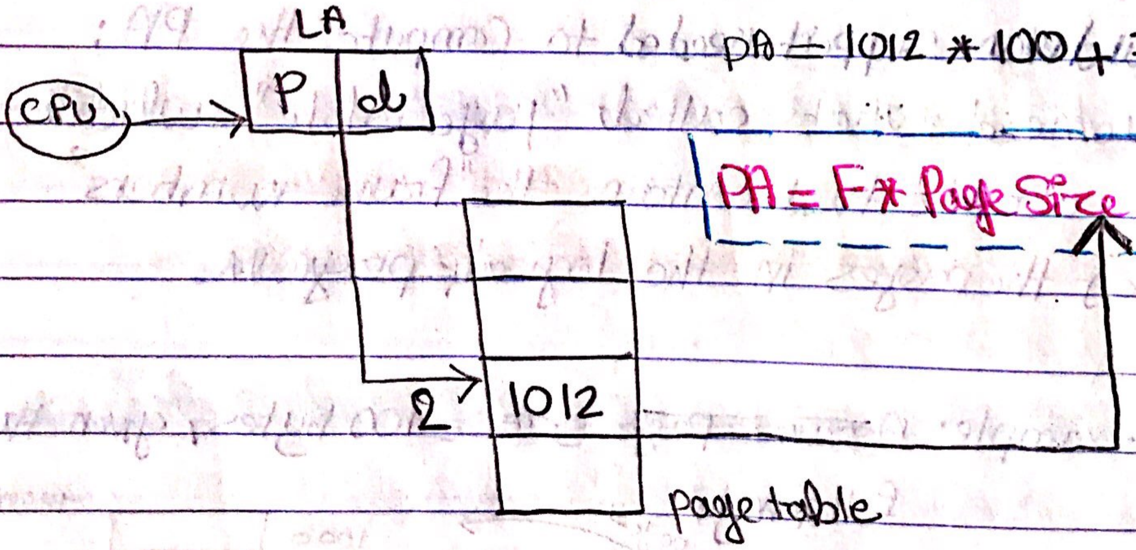
LA = 271 ⇒ { P = 2, d = 71,

P = 271 / 100 = 2

d = 271 % 100 = 71

- The operating system computes P & d as follows:

P = LA / Page Size
d = LA % Page Size

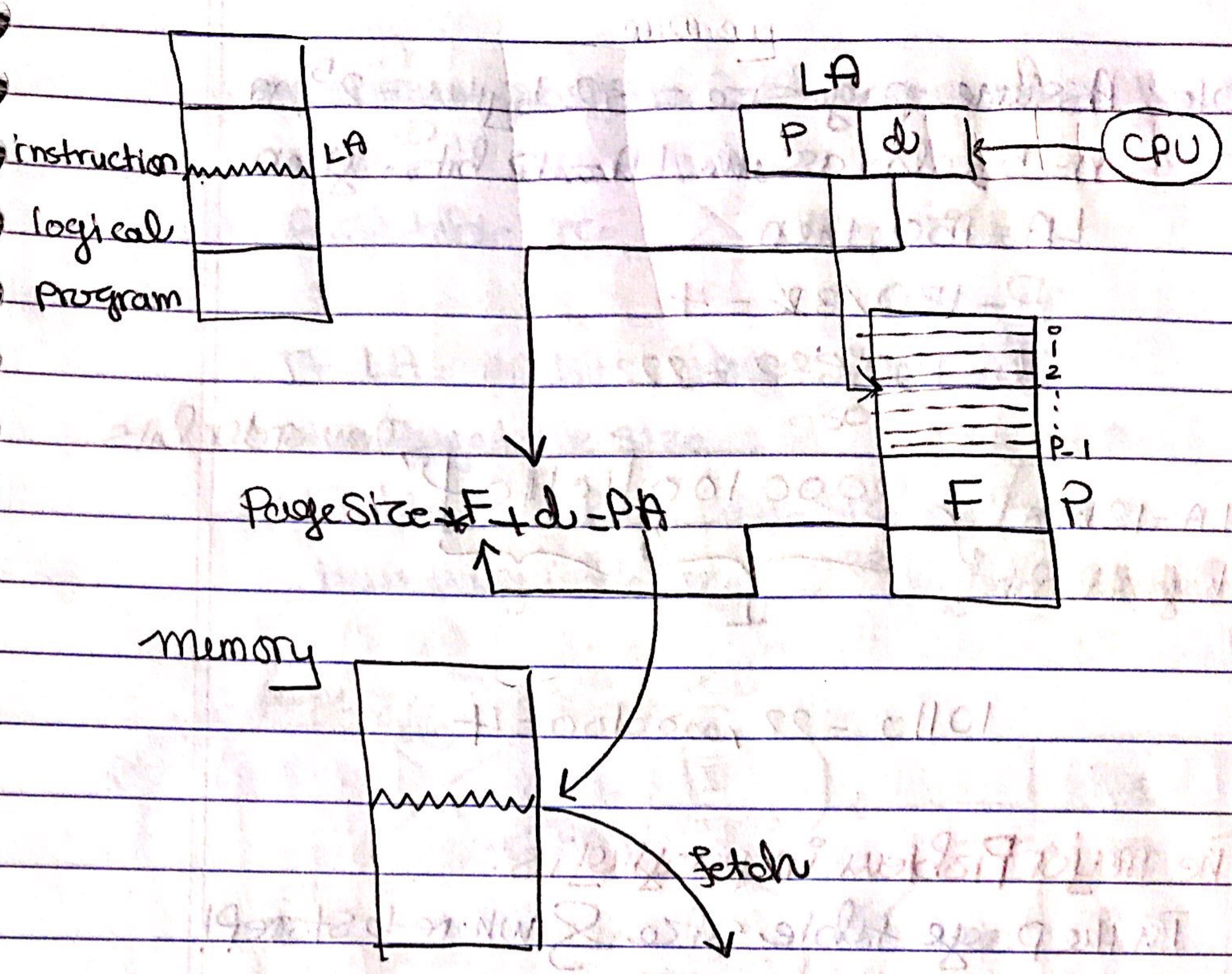


PA = 1012 * 100 + 71 = 101271

PA = F * Page Size + d

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

⚠ Note: /, % operations are multiplication operations, which means, those operations take time in calculation.



— Does the OS, in real life performs those two operations?

→ NO

⚠ Note: the page size in practice is always 2^n bytes.

Generally, $\underset{2^{10}}{1024} \le \text{page size} \le \underset{2^{13}}{8192}$

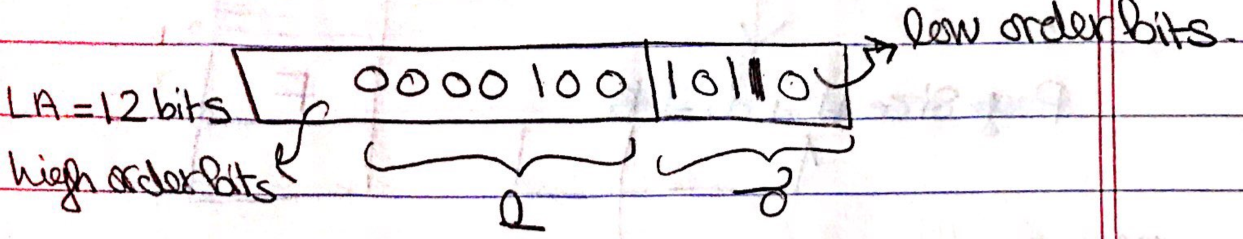
most of the times, Page size = $4096 = 2^{12}$

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 assum LA = 12 bits, given LA=150, then:

$P = 150/32 = 4$

$d = 150\%32 = 22$

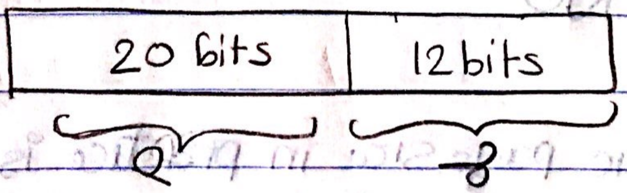
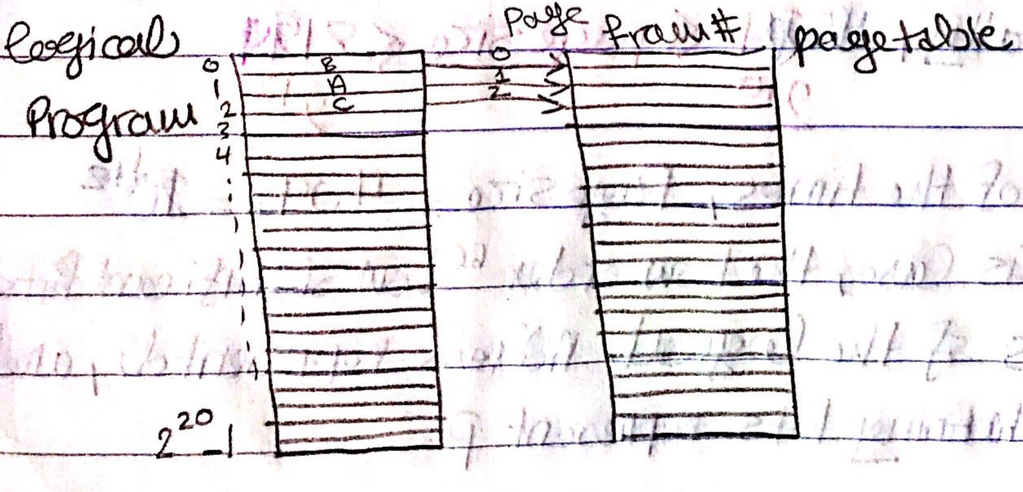


$10110 = 22$, $0000100 = 4$.

⚠ The major Problem in paging is:

In the page table size & where to store?!

example: Assum LA=32 bits, page size = 4096 = 2^{12}

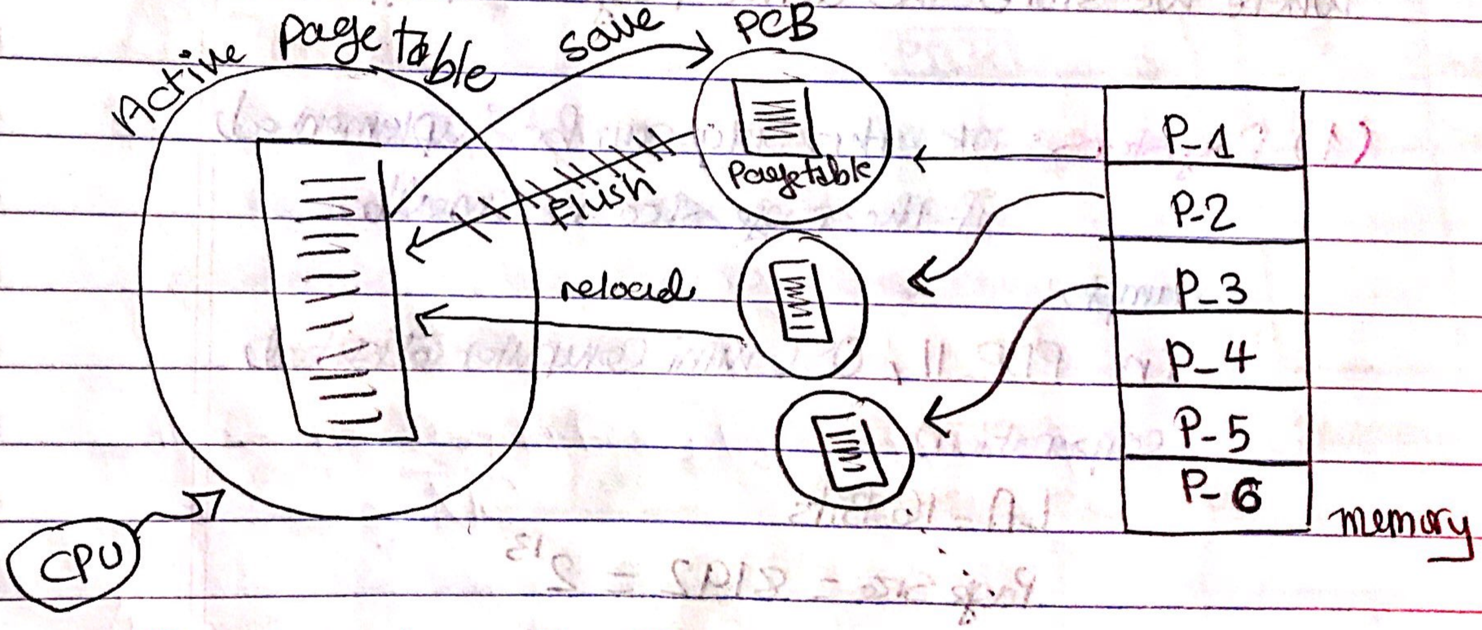


⊞ maximum program can be executed on this machine $= 2^{20}$

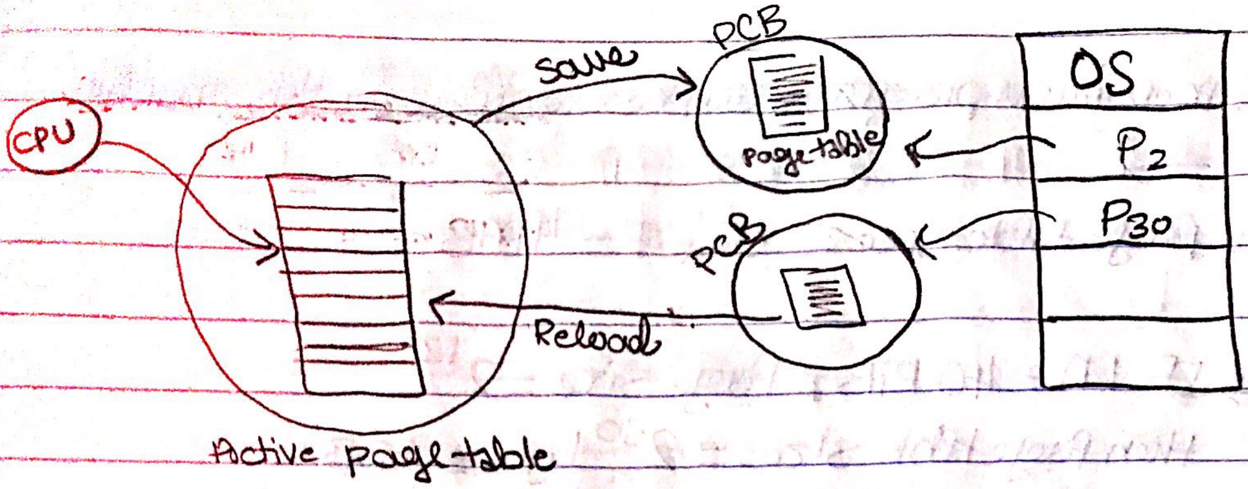
⊞ Page table size $= 2^{20} * 4 = 4MB$

→ If LA = 40 Bits, Page size $= 2^{12}$
then Page table size $= 2^{30}$ byte = 1 GB

→ If LA = 48 Bits, Page Size $= 2^{12}$
Then page table Size $= 2^{38}$ bytes = 256 GB



⊞ 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?!

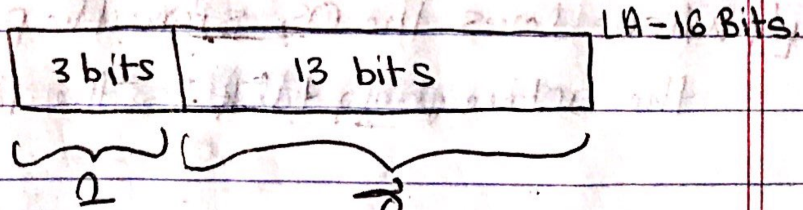
(1) Registers: ok, this only can be implemented if the page size is small.

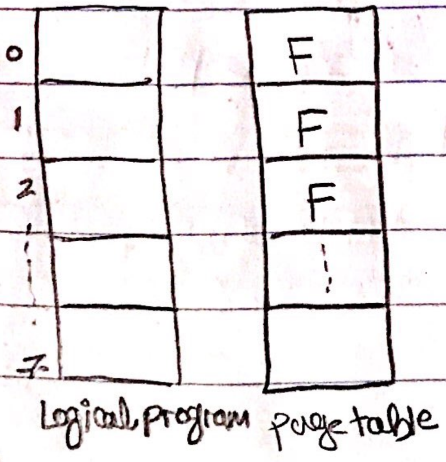
example:

In PDP 11, OS(mini Computer digital corporation)

LA = 16 Bits

Page Size = 8192 = 2^{13}



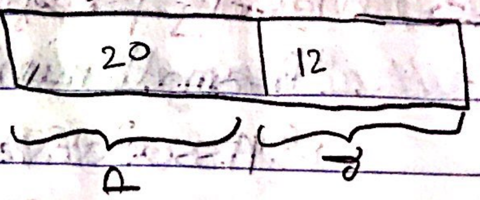


* max program can be executed containing only $2^3 = 8$ pages

* Page table size = $2^3 *$ Frame number size $= 2^3 * 4 = 32$ bytes

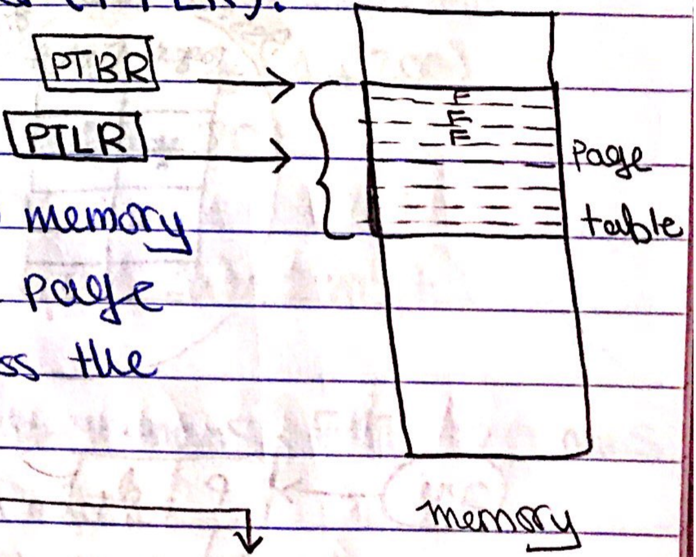
But, If LA = 32 bits

Page size = 4096 = 2^{12}



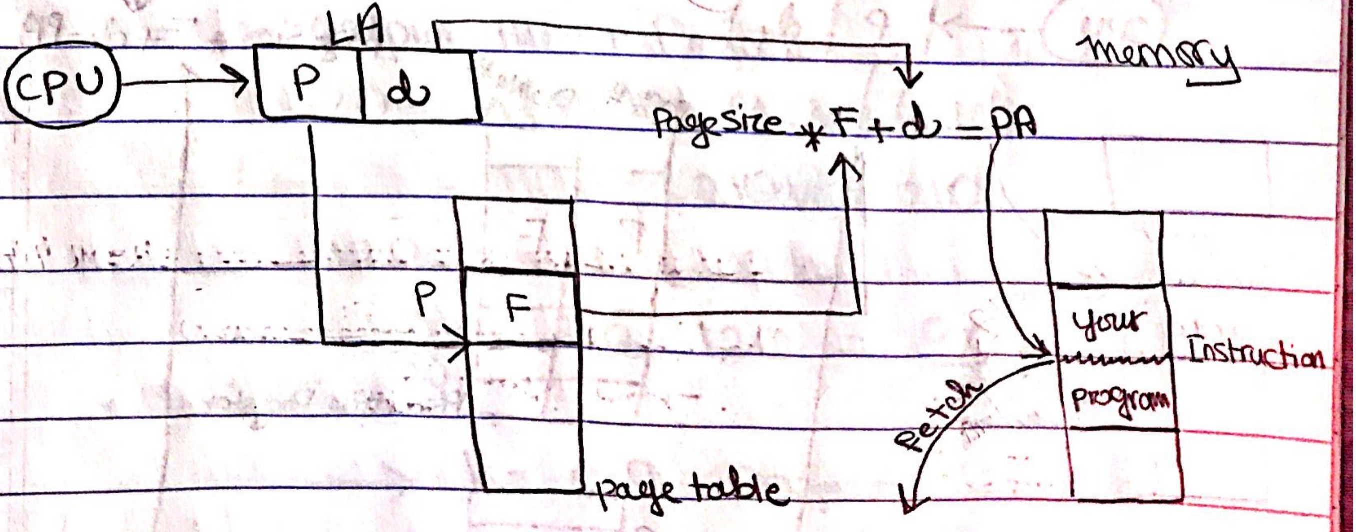
∴ size of page table = $2^{20} * 4 = 4\text{MB}$

(2) Memory: Keep the page table in memory identified by the Page table base register (PTBR) & Page table Limit register (PTLR).



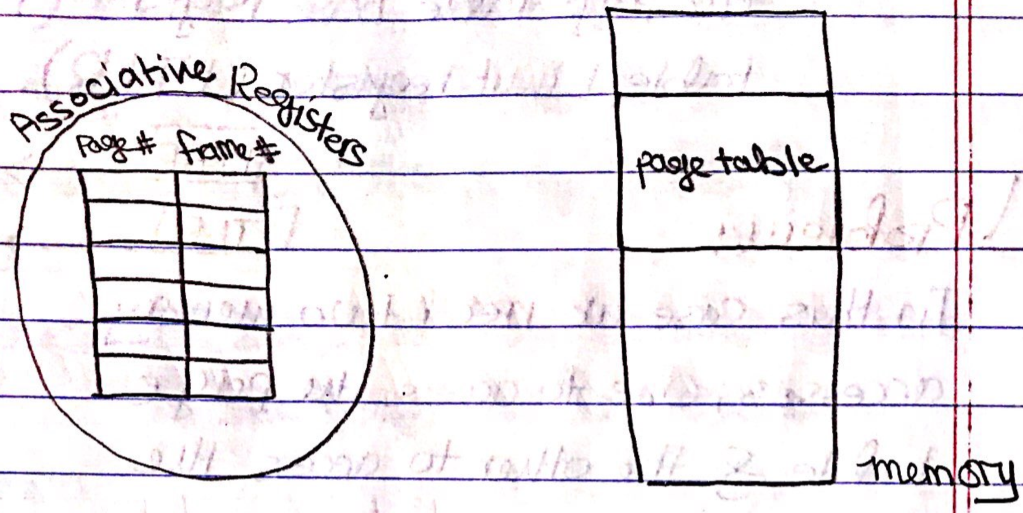
⚠ Problem:

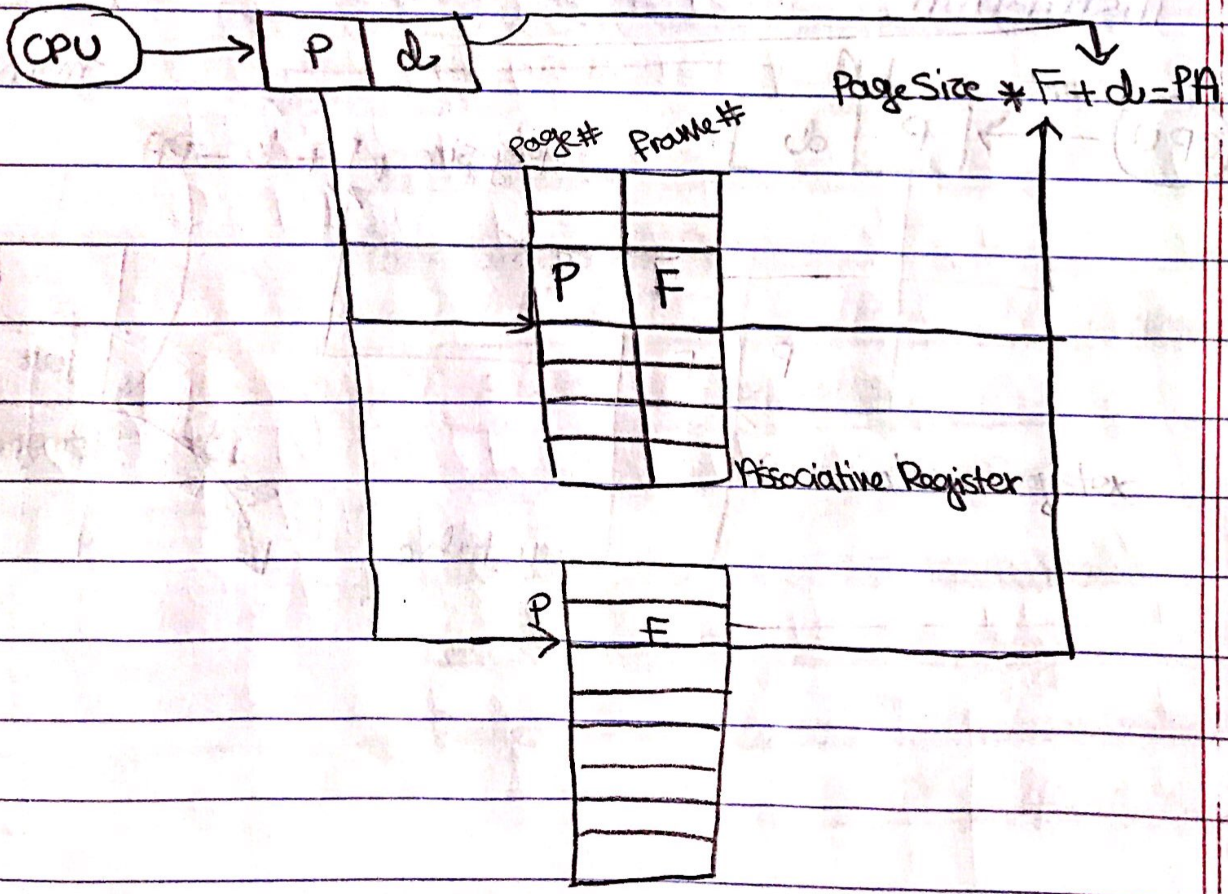
In this case we need two memory accesses, One to access the page table & the other to access the instruction.



(3) Memory + Registers

- A small number of registers called 'Associative Registers' or 'translation Look aside buffers' are assigned or dedicated for a small page table whose entry contains (Page#, Frame#)





– Performance of Associative Register depends on Hit Ratio (h)

– Hit Ratio: probability that the desired page ~~[illegible]~~ in the associative Register.

example:

① – memory access = 100 nanS → m

– Search time in associative register = 1 nanS → t

– Hit Ratio (h) = 0.95

→ Effective Access Time (EAT)

$= 0.95 * (1 + 100) + 0.05 * (1 + 200)$

$= 0.95 * 101 + 0.05 * 201$

$= 106$ nanS.

→ $EAT = h * (m + t) + (1 - h)(2m + t)$

② m = 100 nanS, t = 10 nanS, EAT = 120 nanS

compute the hit Ratio:

$120 = h(110) + (1-h)(210)$

$= 110h - 210h + 210$

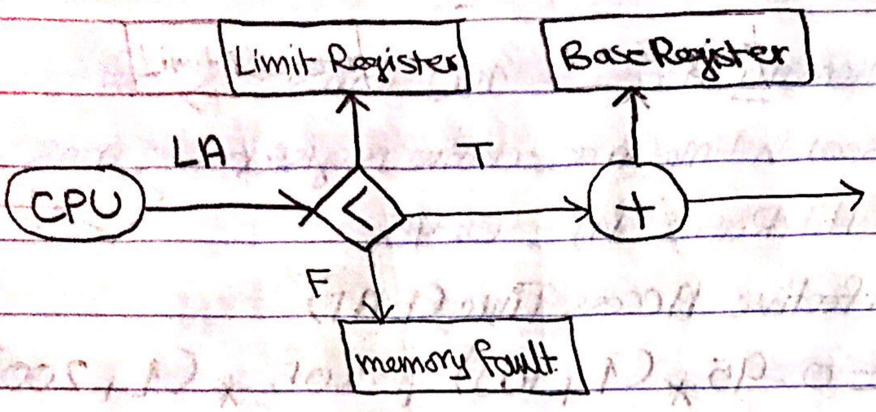
$120 = 210 - 100h$

$\frac{100h}{100} = 210 - 120 = \frac{90}{100}$

→ $h = 0.9$ perfect.

[4] Memory Protection:

In multiple partitions, memory protection is performed using base & Limit Registers.



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

(1) Legal / ILLegal Bits:

assume my program is 4 pages.

1: legal page

0: ILLegal Page

(2) R/W Bit (Read only Bit)

1: R/W Page.

0: Read only page

A
B
C
D

logical program

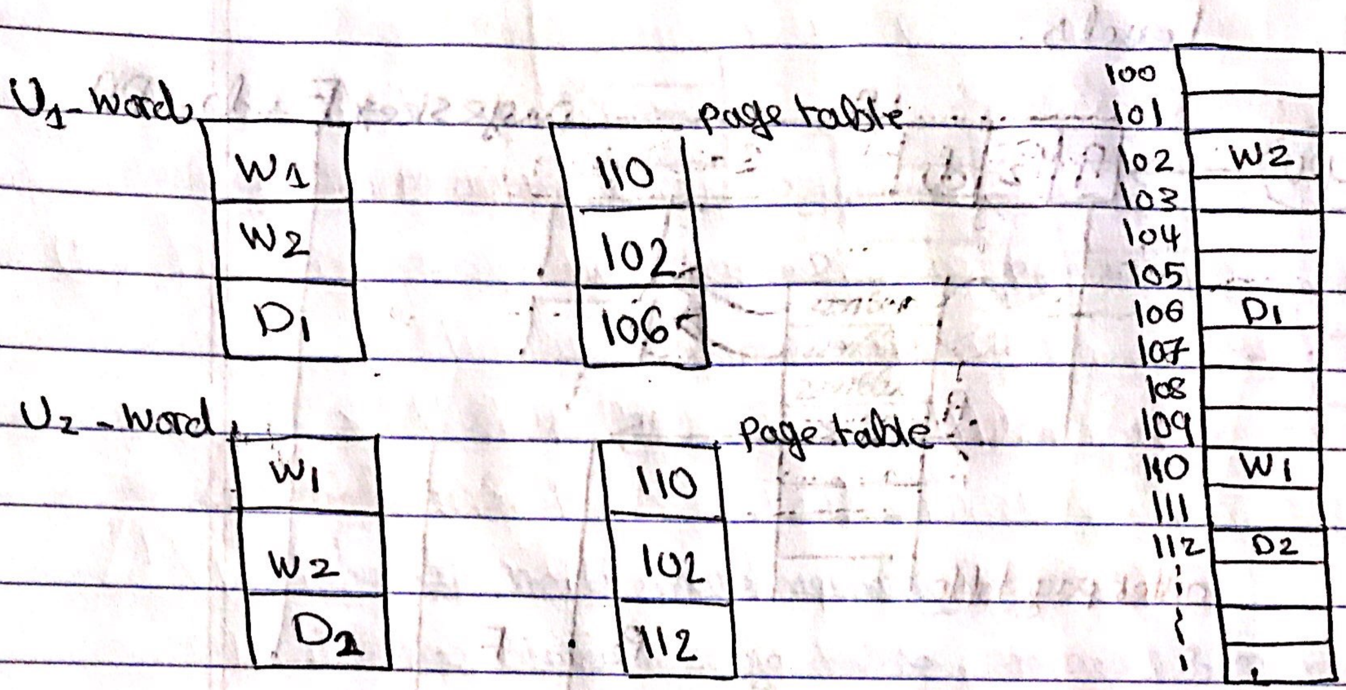
:

	L/IL	R/W
12510	1	1
7415	1	1
8001	1	1
120	0	0
	0	0
	0	0

page table

Paging Advantages:

‘advantages of sharing pages’



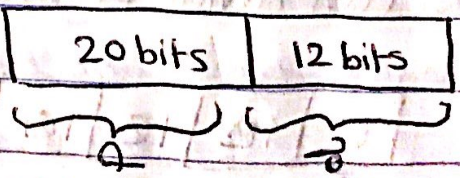
Disadvantage:

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

Multi-Level page table:

① Assume LA = 32 bits.

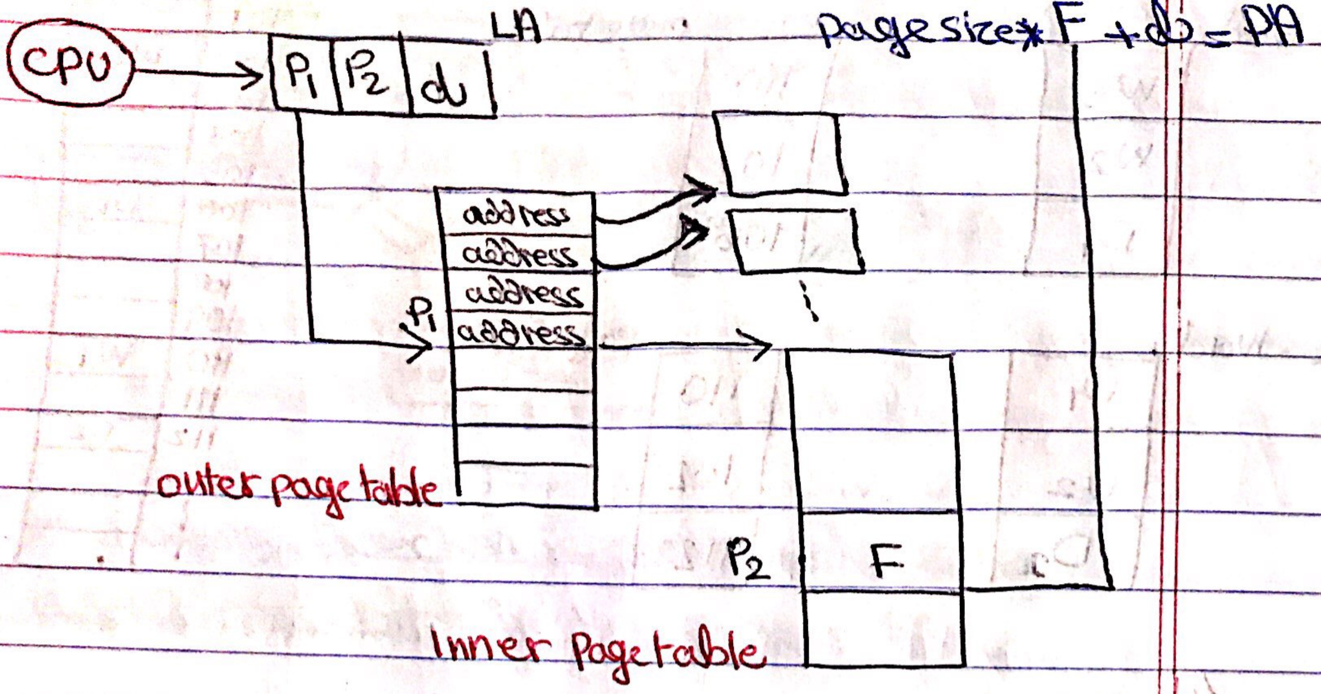
Page size = 4096 = 2^{12}



∴ Page table size = $2^{20} * 4 = 4$ MB

→ This is big to store contiguously in memory.

→ Let's divide the page table into two Levels.



— In our example, assume $P_1 = 8$ bits

$P_2 = 12$ bits

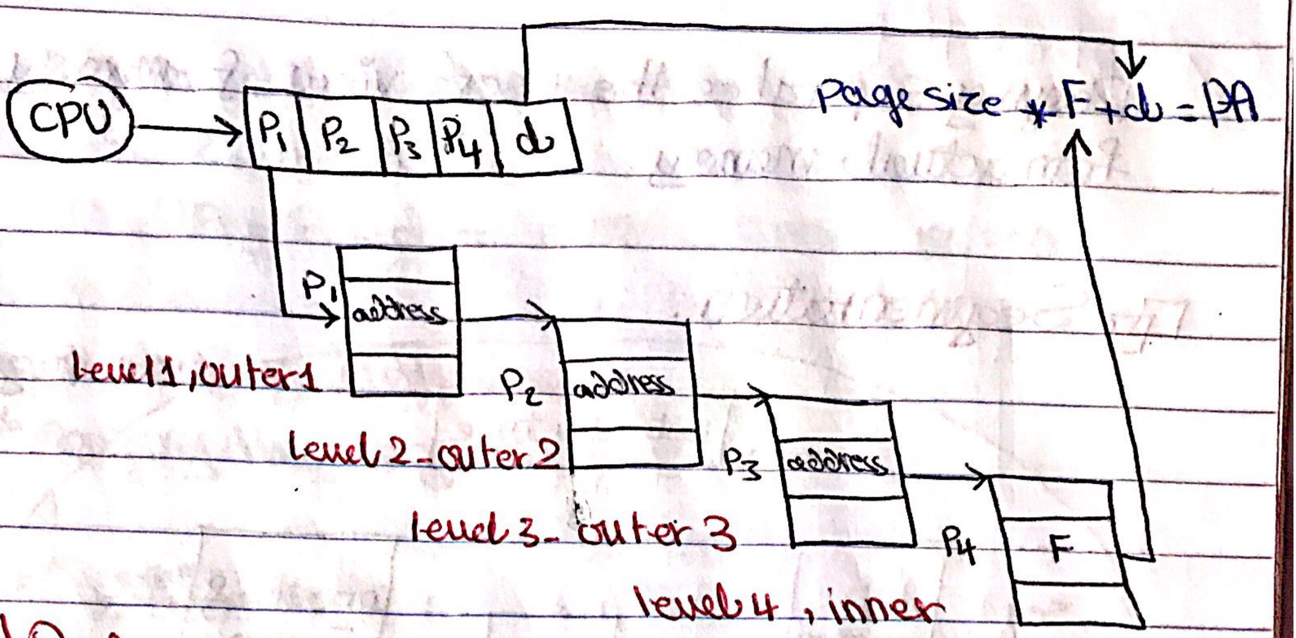
— outer page table size $= 2^8 * 4 = 2^{10} = 1KB$

— Inner page table size $= 2^{12} * 4 = 2^{14} = 16KB$

② LA = 48 bits , Page size = 2^{12} bytes

OS can divide LA into 4 levels

6	10	10	10	12
P_1	P_2	P_3	P_4	d



⚠ Problem:

↳ We need more memory accesses.

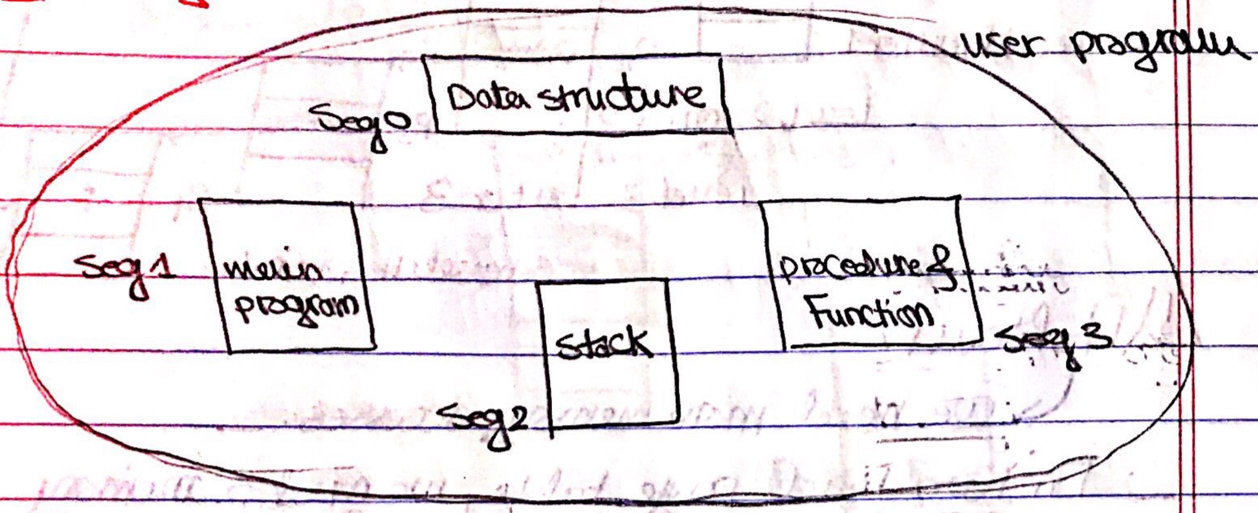
- In two level page table, we need 3 memory accesses.
  - One to get address (P_1).
  - One to get F.
  - One to fetch instruction.
- In 4-Level page table, we need 5 memory accesses. But, with good associative Register Algorithm, the performance will be as well good.

example: assume,
- memory access time = 100 nans
- associative register Search time = 1 nanS
- assume 4 levels of Page table with hit ratio (h) = 0.98

$$EAT = 0.98(100+1) + 0.02(500+1)$$
$$= 0.98 * 101 + 0.02 * 501 = 109$$

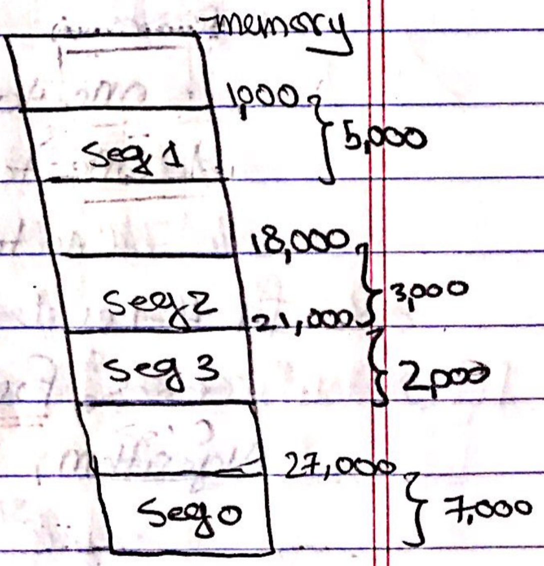
- Paging Seperates the user's view of memory from actual memory.

Segmentation:



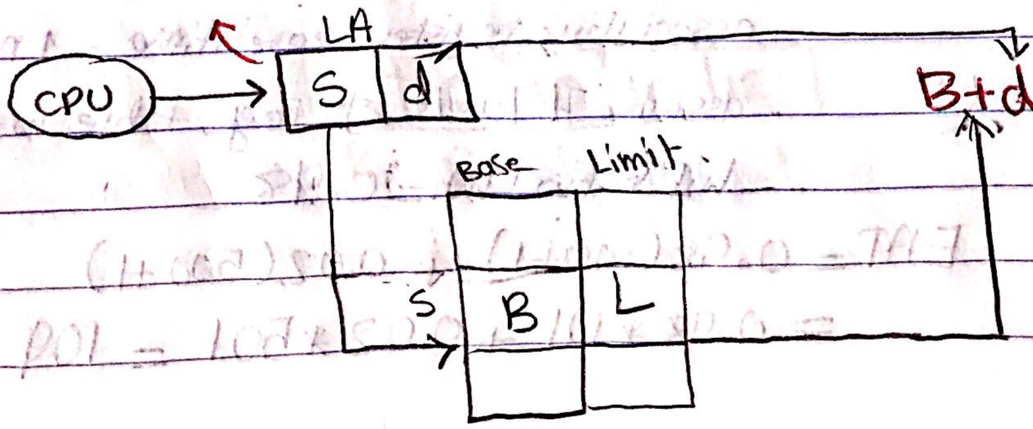
	Segment Base Register	segment Length
0	27,000	7,000
1	1,000	5,000
2	18,000	3,000
3	21,000	2,000

Segment table



s: Segment #

d: LA offset.



$B+d=PA$

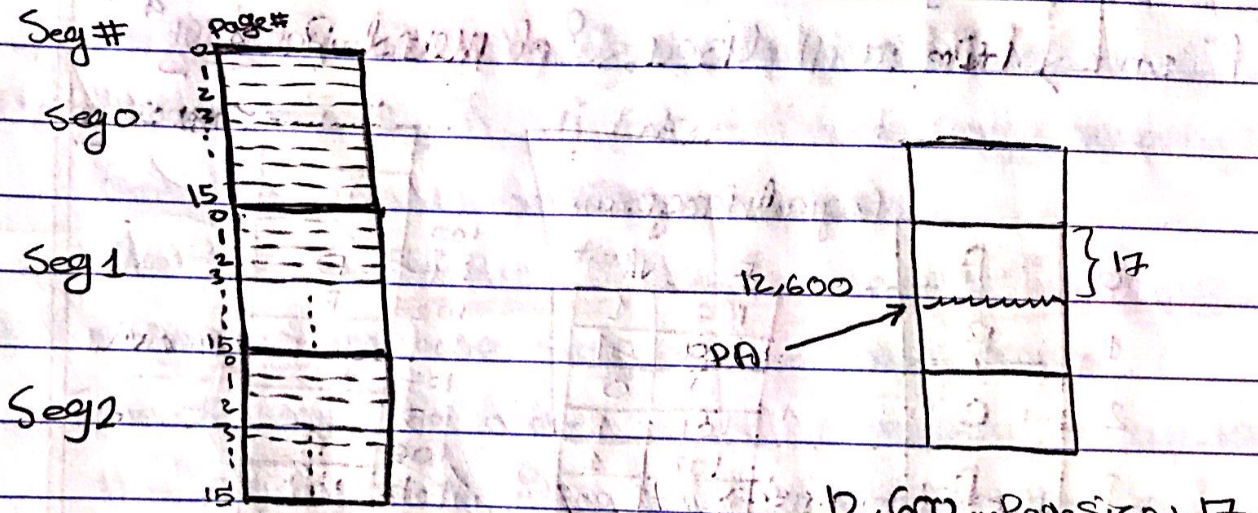
example: given LA which generates:

$S = 2, d = 350$

$PA = B + d = 18,000 + 350 = 18,350$

☐ Segmentation with Paging

"paging the Segments"

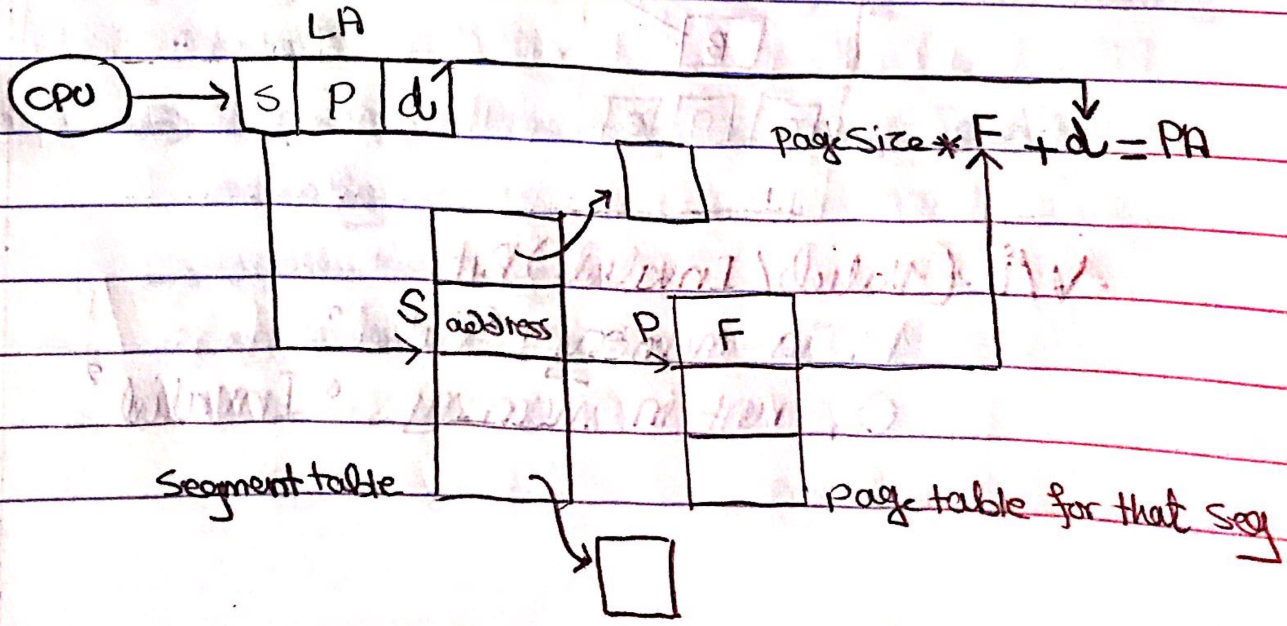


12,600 * PageSize + 17

If Seg size = 64K.

↳ page size = 4K.

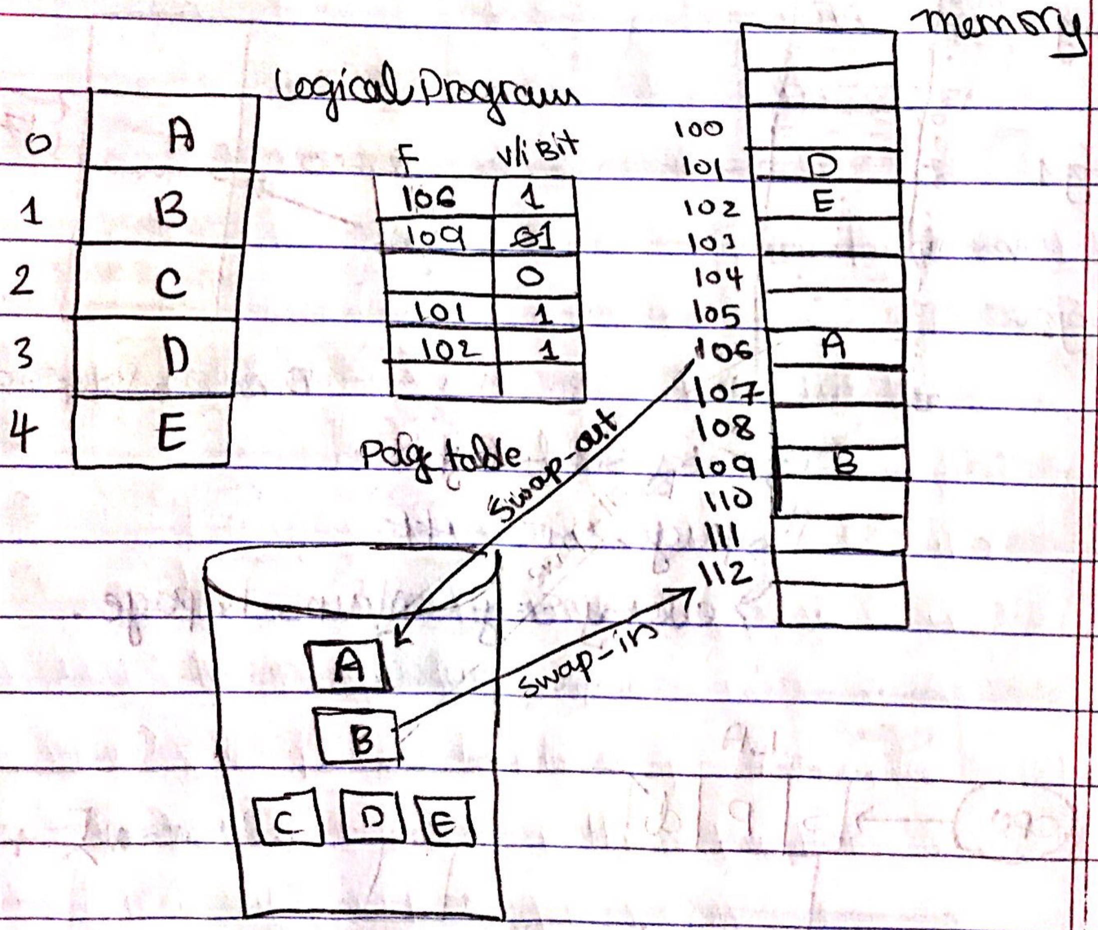
↳ every Seg contains 16 page.



Chapter#9:

Virtual Memory Management.

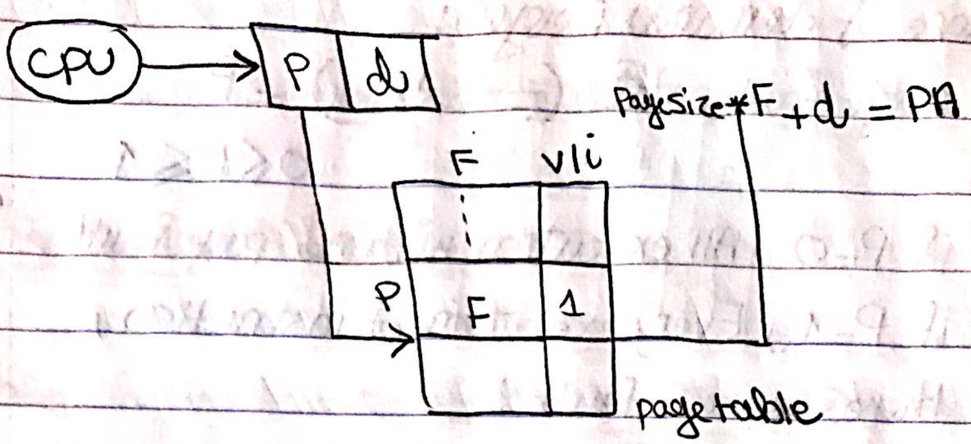
- How to run bigger ~~pro~~ programs on Smaller memory.
- No need for the whole program to be Loaded into memory, only part of it will do.
- We will discuss "demand Paging"



v/i (Valid/Invalid) Bits

1; in memory. "valid"

0; not in memory. "Invalid"



- The LA is checked through the page table if it's a valid address, that's, if v/i Bit = 1 (the page memory) that it continue execution as usual.
- What if the v/i Bit = 0, that's the page isn't in memory, then we say a page fault occurs.
- The OS looks for a free frame in memory, swap-in the required page from HD, it updates the page table & resumes execution.
- What if there's no free frame?
  - ⇒ It selects a victim frame
  - ⇒ may be, 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:

- Assum page fault rate (probability) = P
  $0 \leqslant P \leqslant 1$
  - → If P = 0, All execution with no page fault. "perfect"
  - → If P = 1, Every execution of instruction, there's a page fault.
- Assum memory access = m

$$EAT = (1 - P) * m + P * \underbrace{[\text{swap in Page} + \text{maybe swap\_out page} + m]}_{\text{Page fault overhead}}$$

Example: Assum memory access (m) = 10 mics

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

$$EAT = (1 - P) * 10 + P * [\overset{\text{swap\_in}}{10} * 1000 + 0.4 * 10 * 1000 + 10]$$

$$= 10 - \cancel{10P} + 10{,}000P + 4{,}000P + \cancel{10P}$$

$$= 10 + 14{,}000P \simeq 14{,}000P$$

⚠ تحويل الوحدات الأكبر لوحدات أصغر

mils, ~~mo~~ 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 = page is modified.
0 = page isn't 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.

1 2 3 4 1 2 5 1 2 3 4 5

1	1	1	4	4	4	5	✓	✓	5	5	✓
	2	2	2	1	1	1			3	3	
		3	3	3	2	2			2	4	

9 page faults

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	✓	✓	5	5	5	5	4	4
	2	2	2			2	1	1	1	1	5
		3	3			3	3	2	2	2	2
			4			4	4	4	3	3	3

10 page faults

∴ Belady's Anomaly!

2 Optimal Replacement:

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

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	✓	✓	1	✓	✓	3	3	✓
	2	2	2			2			2	4	
		3	4			5			5	5	

7 Page faults

⚠ 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.

1 2 3 4 1 2 5 1 2 3 4 5

1	1	1	4	4	4	5	✓	✓	3	3	3
	2	2	2	1	1	1			1	4	4
		3	3	3	2	2			2	2	5

10 page faults "3 Frames"

1	1	1	1	✓	✓	1	✓	✓	1	1	5
	2	2	2			2			2	2	2
		3	3			5			5	4	4
			4			4			3	3	3

8 page faults "4 frames"

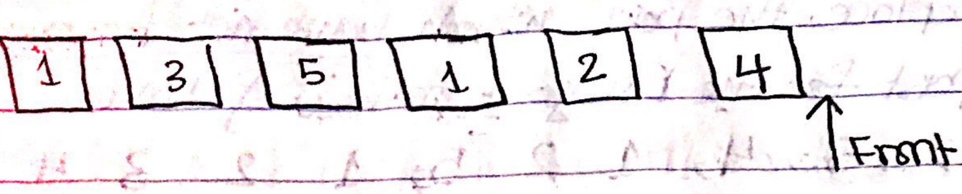
* Implementation:

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

	time of reference
	10:3:20 → Integer
	10:3:11

page table

(2) Keep a queue of referenced pages



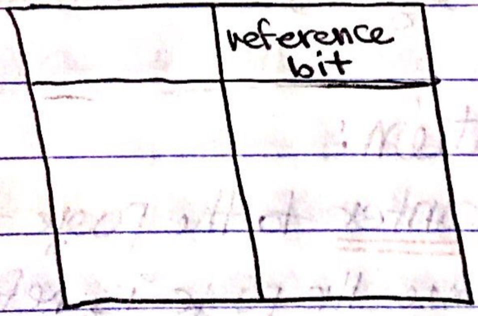
→ or we can use a stack, Every time we use a page, push it into stack.
↳ the stack top is the latest used page

LRU approximation:

Add one bit to the page table called reference Bit

↳ 1 ≡ The page is referenced (used), Read or Written.

↳ 0 ≡ Page isn't referenced.



Page table.

→ LRU approximation, Replace the page whose reference Bit is 0.

Second chance:

→ replace 1 0

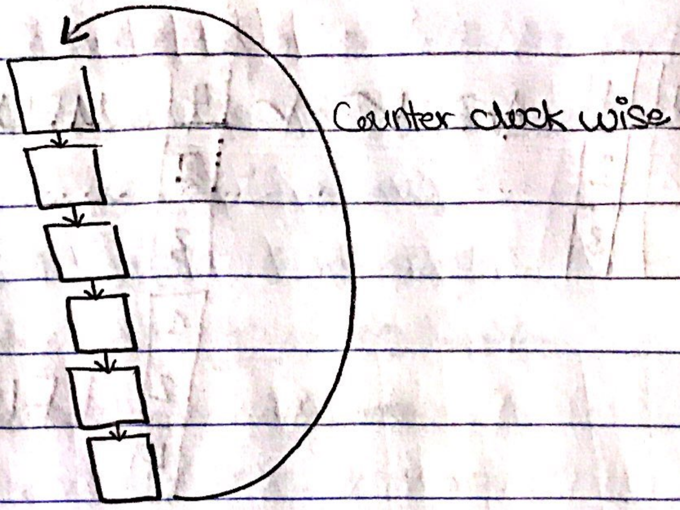
→ no change 0 1

→ nochange 0 1

→ replace 1 0

→ nochange 0 1

→ no change 0 1



Enhanced Second chance:

Use the reference bit & the dirty bit. (Reference bit, dirty bit)

best → (0,0) ≡ Page isn't referenced & not modified. ①

→ (0,1) ≡ Page isn't referenced but modified. ③

→ (1,0) ≡ Page is referenced but not modified. ②

worst → (1,1) ≡ Page is referenced & modified. ④

★ Counting Algorithms:

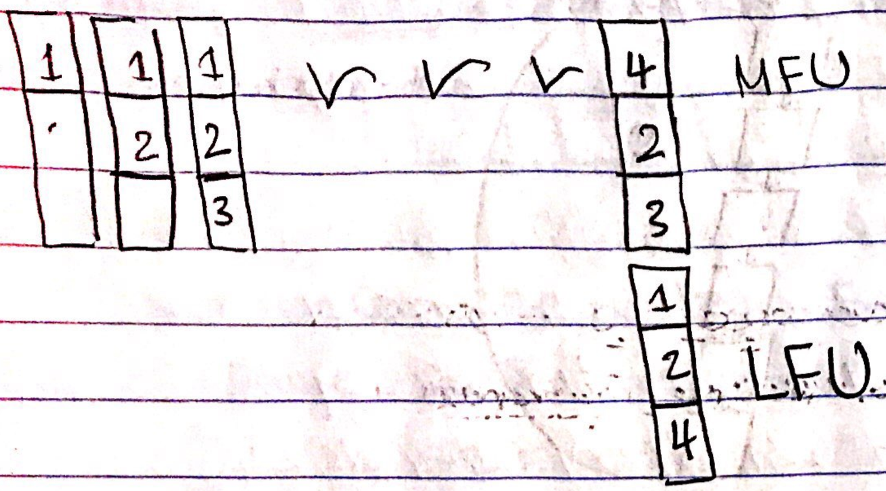
1_ MFU (Most Frequently Used)

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 # of all times.

1 2 3 1 2 1 4 3 4 5



Global vs. Local Replacement

May. 19. 2018

Allocation of Frames for processes :-

(1) Equal Allocation : Each process gets the same number of pages (frames).

example: memory 100 frames, 5 processes

- Every process gets $\frac{100}{5} = 20$ frames.
- unfair → not good performance.

(2) Proportional Allocation according to size :

example: Assume the size of process $P_i = S_i$

Assume we have m frames of memory

Process P_i gets $\frac{S_i}{\sum S_i} * m$

- Assum memory 100 frames, 3 processes with sizes 100, 400, 700 KB.
- $P_1 = \frac{100}{1200} * 100 \simeq 8$ frames
- $P_2 = \frac{1400}{31200} * 100 \simeq 34$ frames
- $P_3 = \frac{700}{1200} * 100 \simeq 58$ frames.

(3) Proportional according to priority:

example:

process	priority
P_1	2
P_2	3
P_3	7 → highest priority

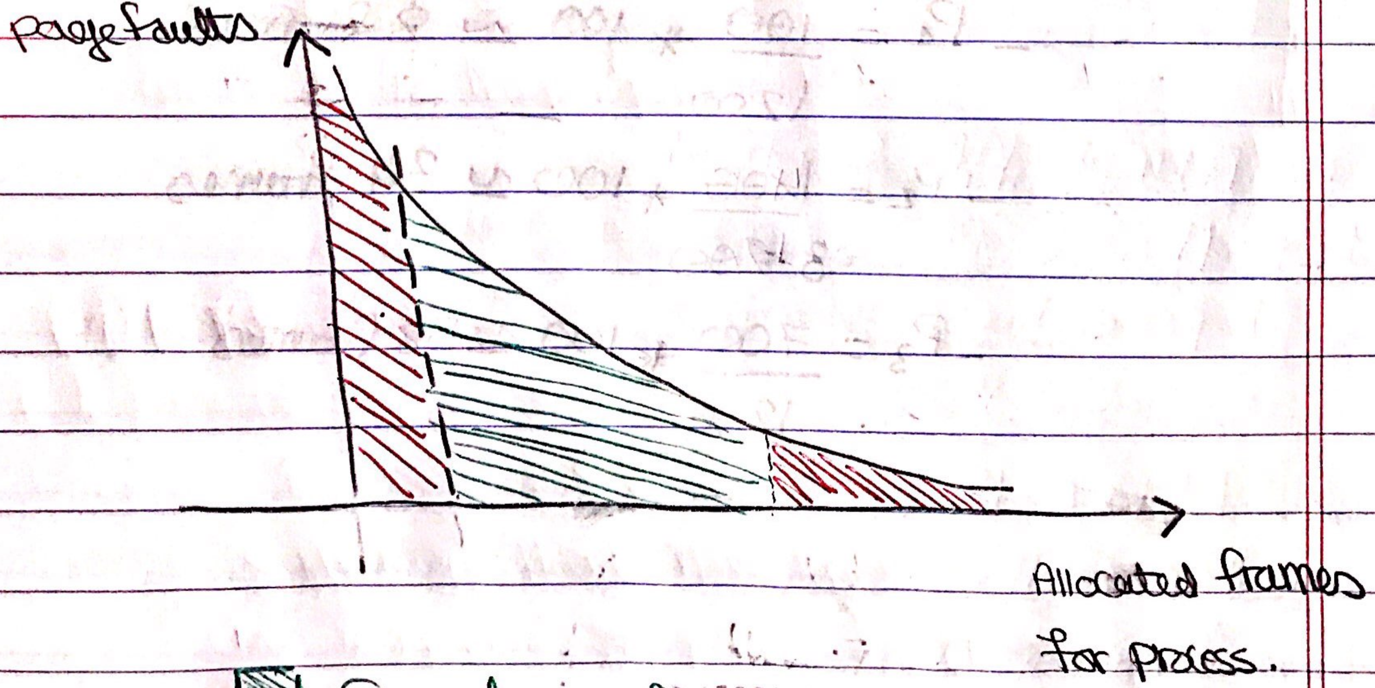
- If we have 100 frames of memory

- $P_1 = \frac{2}{12} * 100 = \frac{1}{6} * 100$

- $P_2 = \frac{3}{12} * 100 = \frac{1}{4} * 100 = 25$ frames

- $P_3 = \frac{7}{12} * 100$

☐ Thrashing: The OS is busy swapping page in & out.



▨ Good performance.

▨ Poor performance.

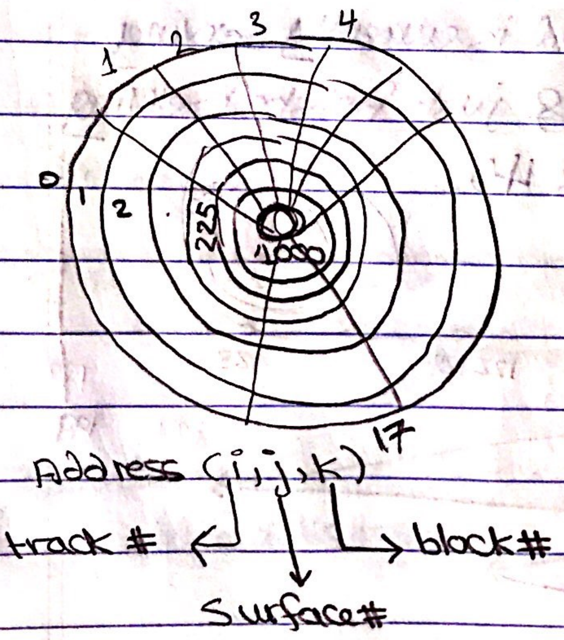
- Sometimes the number of Allocated frames for the process is low, which leads to poor performance (low system utilization).

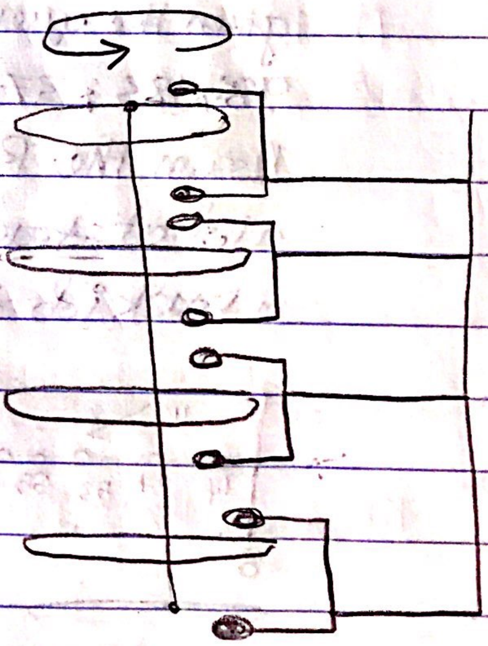
The OS thinks that the degree of multiprogramming is low,

→ OS increases the degree of multi-programming.

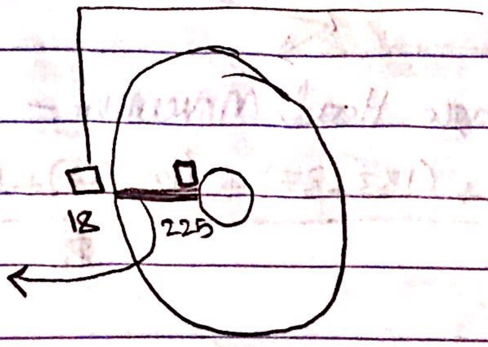
→ The system gets wors.

→ Most of the time the ~~system~~ OS is doing swapping of page.





(225, 3, 17)



The time to move from track 18 → 225 is called "seek time" mechanical move.

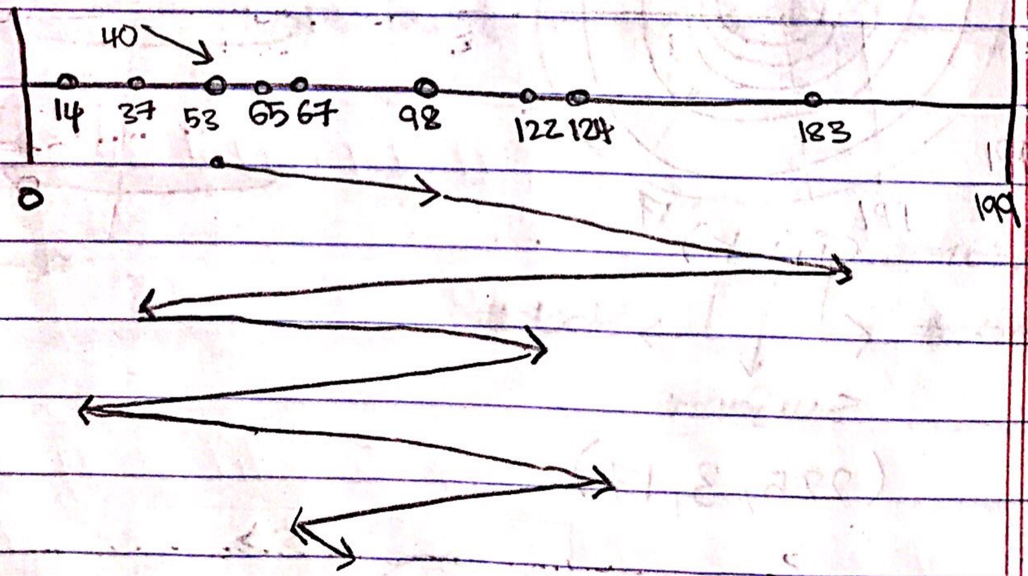
تقنية سرعة للدوران

HD access time = Seek time + Latency time
+ transfer time.

We have no control

→ we could improve the seek time by selecting a good disk scheduling algorithm. (minimize seek time)

example: Assum 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.

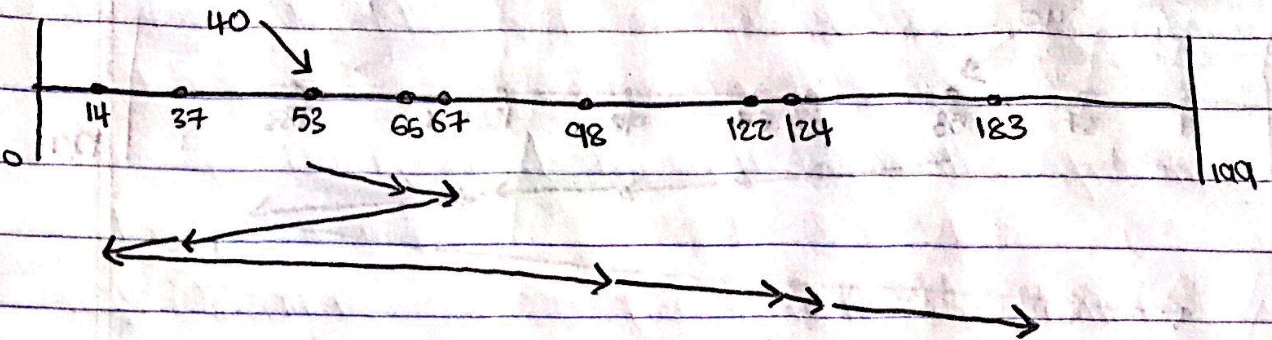


(1) FCFS

→ Average Head Movement =

$$\frac{(183-53)+(183-37)+(122-37)+(122-14)+(124-65)+(67-65)}{8}$$

(2) Shortest SeekTime First (SSTF)

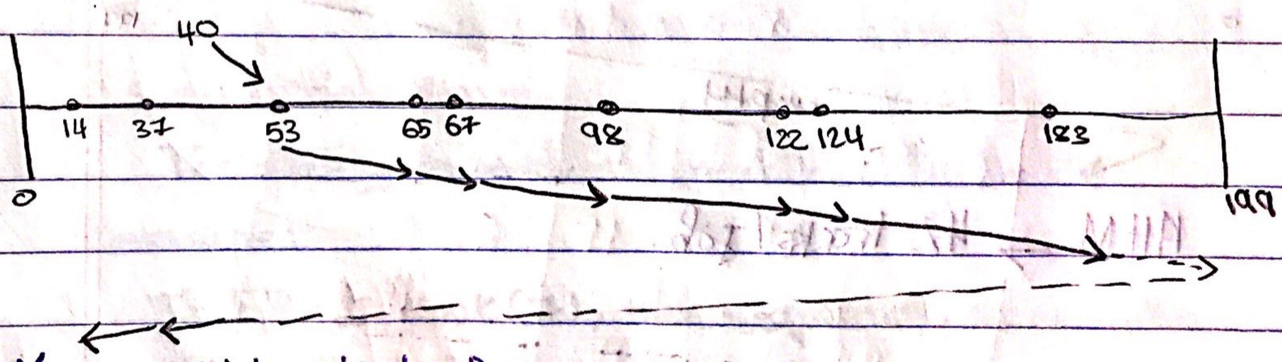


AHM → 29 tracks / Job.

SSTF gives the <u>optimum</u> solution
minimum

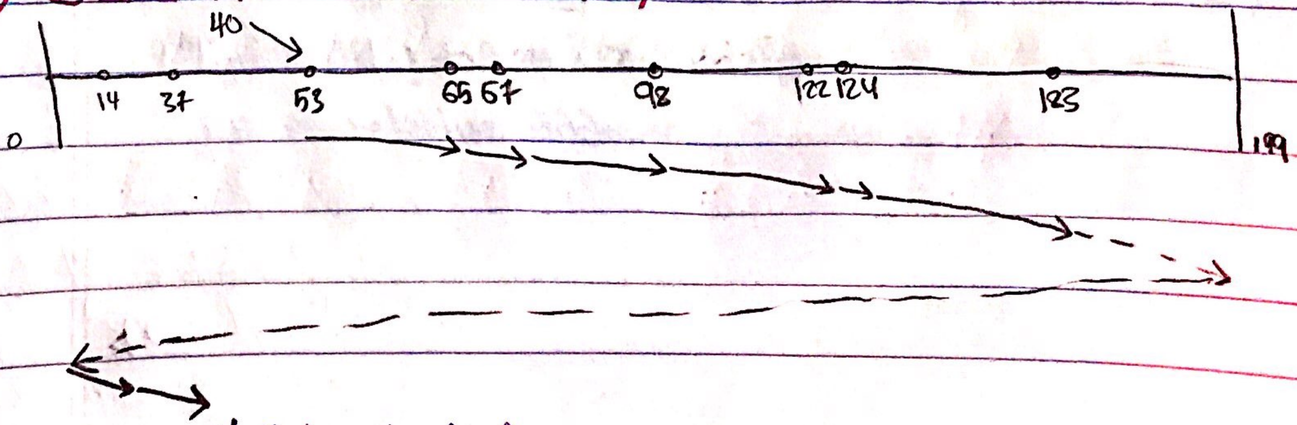
⚠ Problem: starvation

(3) SCAN (Elevator)



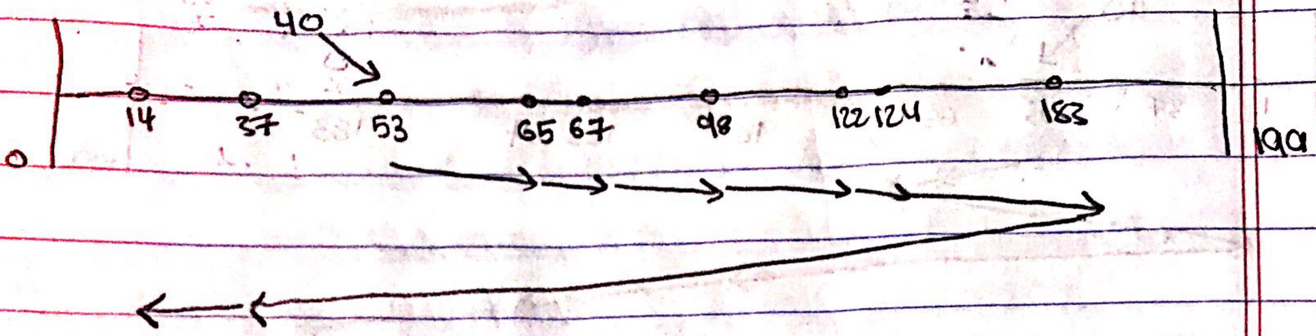
AHM → 41 tracks / Job.

(4) C_SCAN (Circular SCAN)



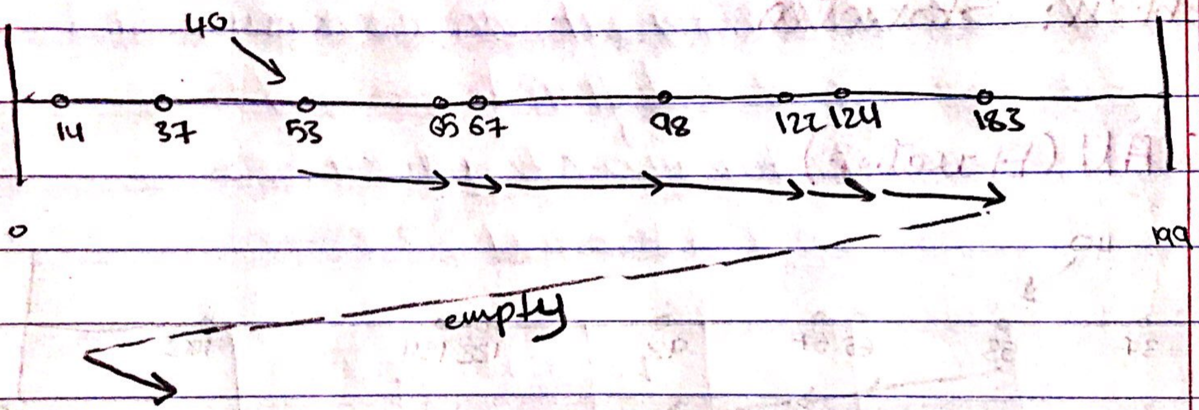
AHM → 47 tracks / Job.

(5) Look:

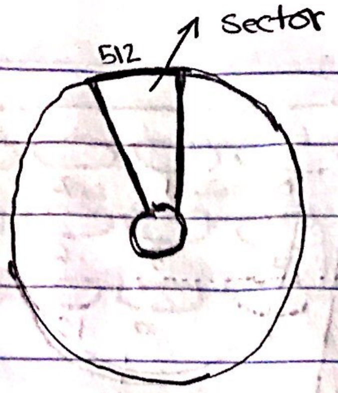


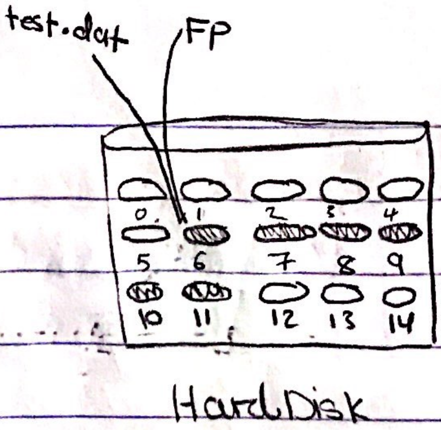
AHM → 37 tracks/ Job.

(7) C-Look:



AHM → 42 tracks / Job.





File Access:

(1) Sequential Access.

(2) Direct Access.

- **Location**: Address of First block in the file (constant, can't be changed)
- **File Pointer**: The current location where the file is located.

→ Sequential Access:

The ability to read the next Block in the file.

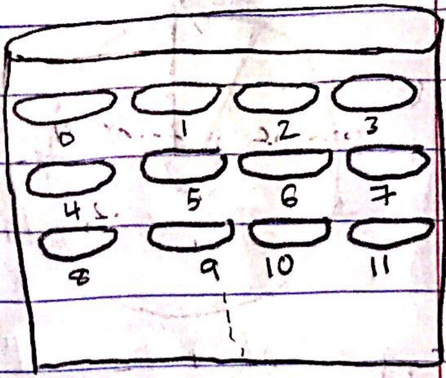
example:

IF FP at Block (7), then sequential access is the ability to read/write Block (8).

→ Direct Access:

The ability to read/write Block # in the file n

(n is relative Address 'رقم Block في الملف'



Allocation Methods:

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

⚠ Note: Every storage device, has "device directory" which contains the information about the files stored on that device.

Generally, it's a "hash table".

— Device directory —

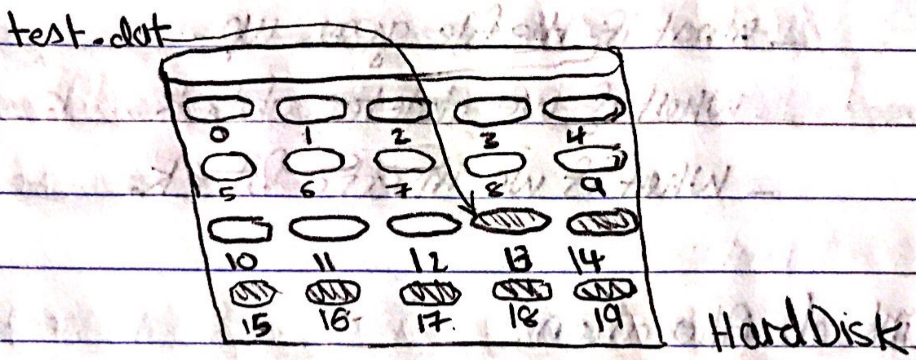
Name	Location	FP	Size	
Sam.txt	11	11	6	
All files on the device				

(1) Contigious Allocation:

- The OS selects contigious blocks of the file.

Device Directory

Name	Location	Size	FP
test.dat	13	7	13



∴ Advantages:

It supports both sequential & direct access easily.

→ Sequential Access:

If FP at address X,

∴ The next block address is X+1.

→ Direct Access:-

Assum we want to read block #5 in the file (5 relative)

- The physical ~~of~~ address of block #5 in the file = Location + 5 - 1 = 17

∴ To access Block #n,

$PA = Location + n - 1$

⚠ Disadvantages:

(1) Fragmentation (holes).

↳ Solution: Compaction (defrag).

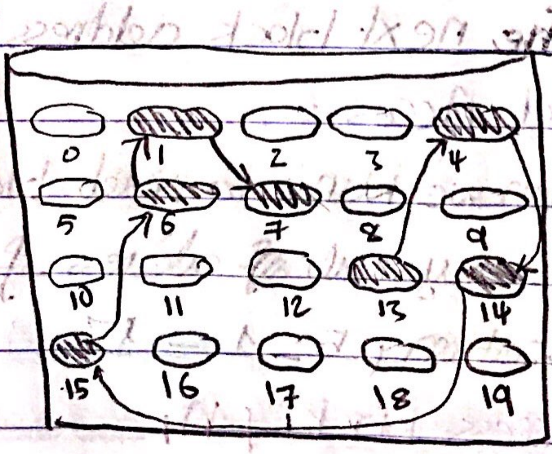
(2) "Major Disadvantage"

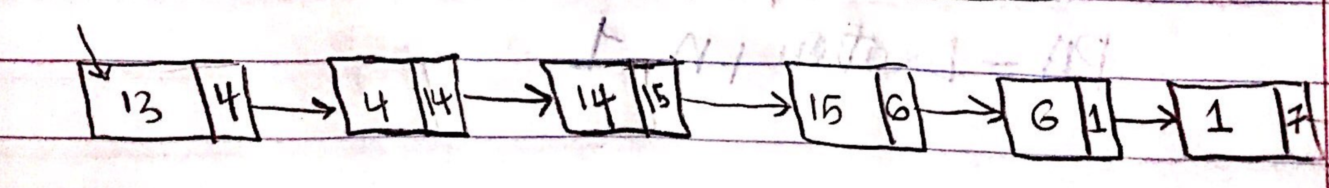
- what if the file grows up.
- what if we delete a block.
- what if we insert a block.

(2) Linked Allocation:

- The file Blocks are selected as a linked list of Blocks.

Name	Location	Size	
test.dat	13	7	





∴ Advantages:

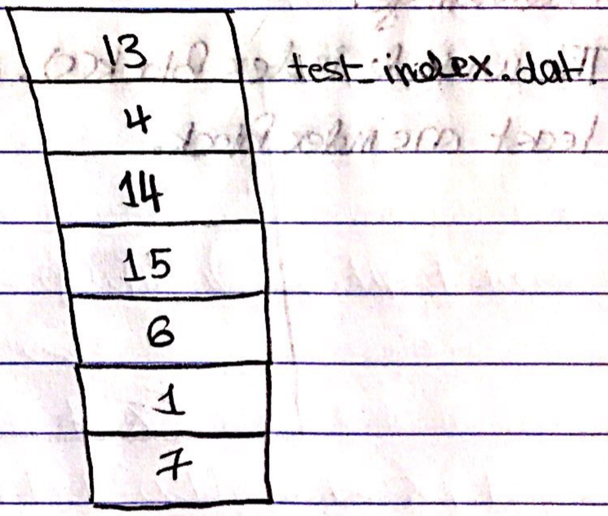
No fragmentation

_File can grow, insert, delete as you like.

⚠ Disadvantage: "Major"

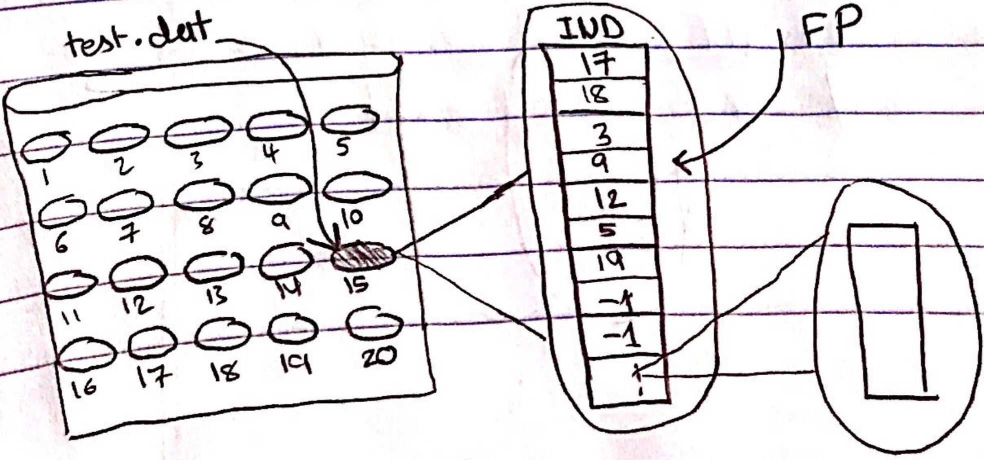
_It only supports Sequential Access easily.

_Direct Access isn't ~~sa~~ supported & needs other tools such as an "index file".



(3) Indexed Allocation:

_Each file has at least one index block which contains the addresses of the file data blocks.



∴ Advantages:

It supports both sequential & Direct Access.

→ Sequential Access:

IF FP at IND[n];

next block address equals IND[n++];

→ Direct Access:

To access block # n in the file, its PA = IND[n].

⚠ Disadvantage:

The wasted index Block(s). each file needs at least one index Block.