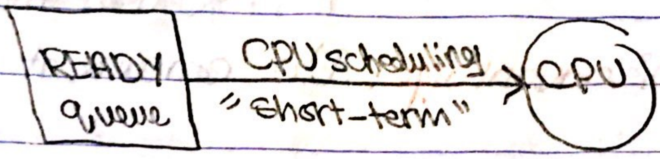
Chapter #5:

CPU Scheduling.

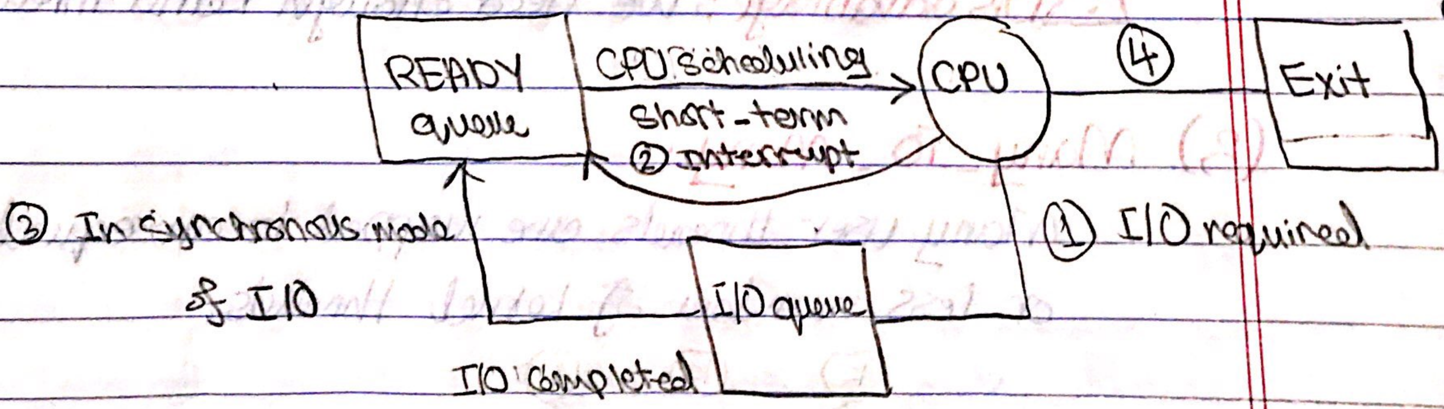
CPU scheduling:

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.

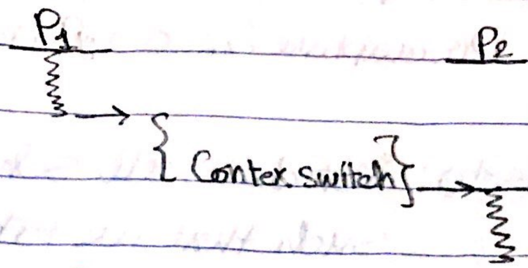
(5) Response Time:
it's the time from submitting job until you see the first response from the Computer.

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

→ Every switching from processes P_i to P_j needs 2 Context switch.

lecture #12

March 6. 2018
Tuesday



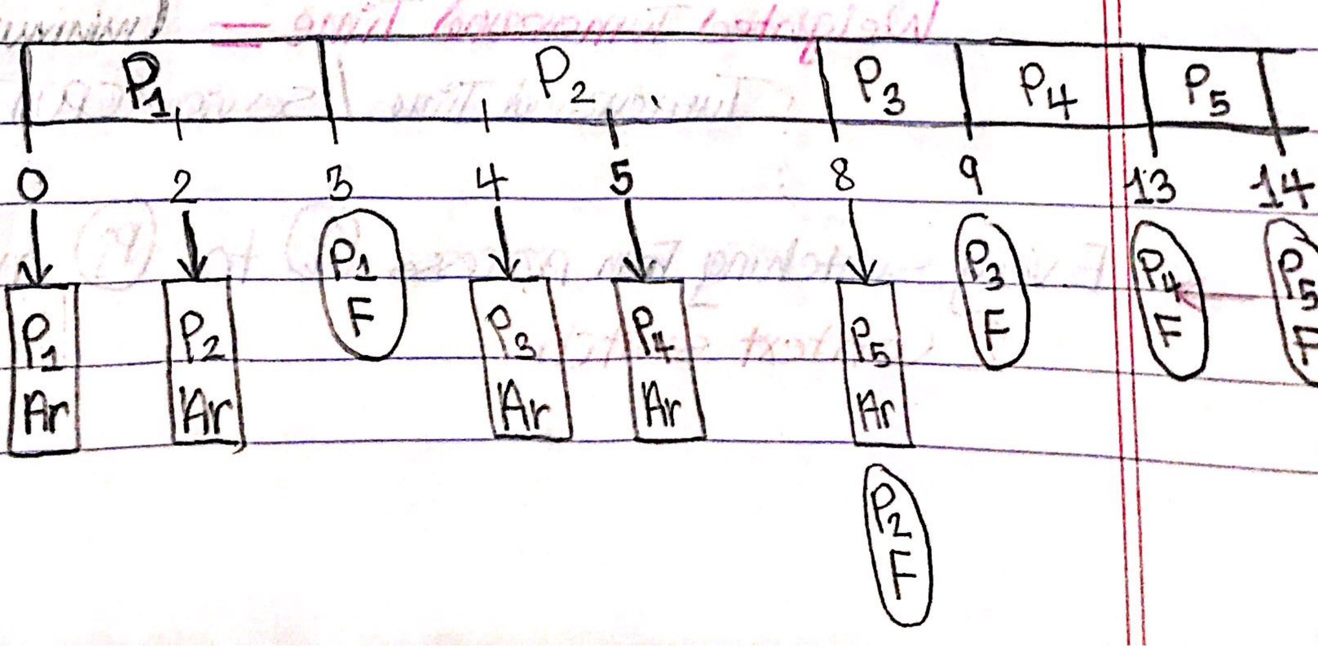
[1] FCFS → First Come First Serve :

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 =

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

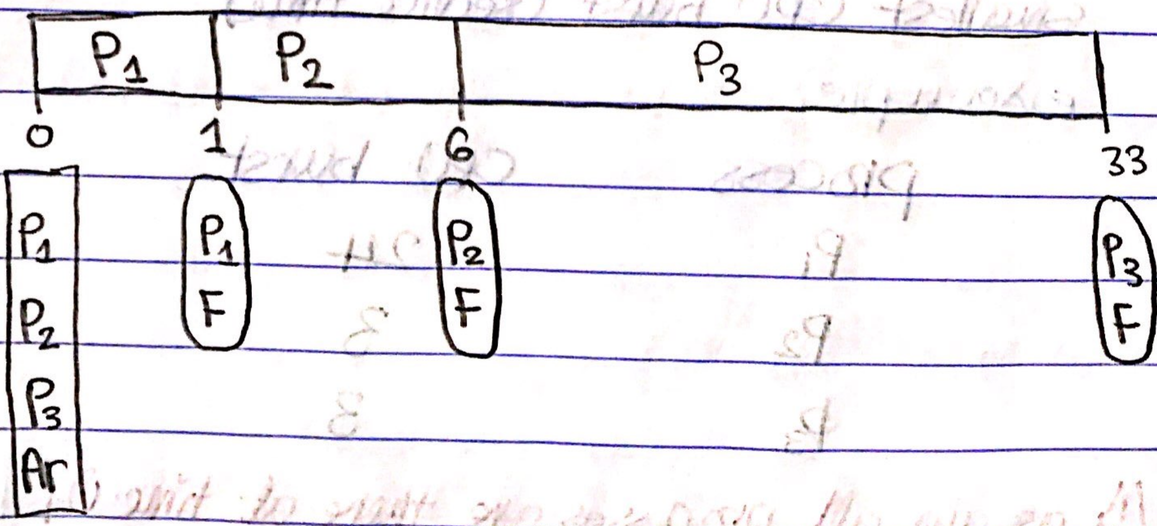
- Average waiting time =

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

FCFS: Convoy problem "كيفية وصول الموكب"

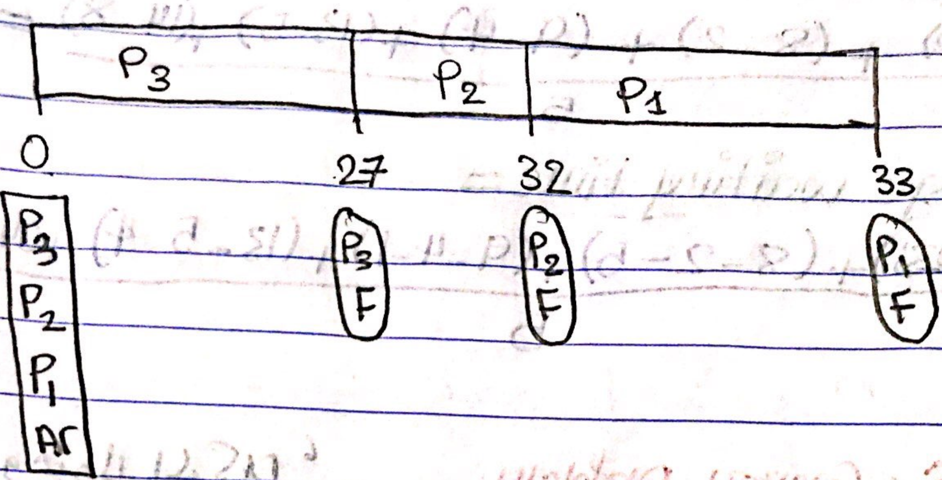
example:-

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.

P_2	P_3	P_1

0 3 6 30

P_2 F (at 3), P_3 F (at 6), P_1 F (at 30)

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

* 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.

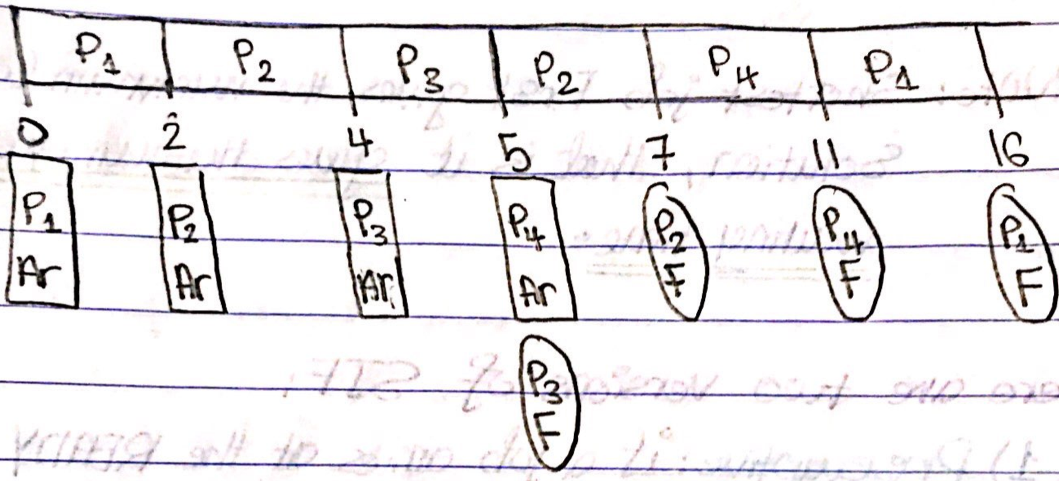
Shortest Remaining Time First (SRTF)

(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

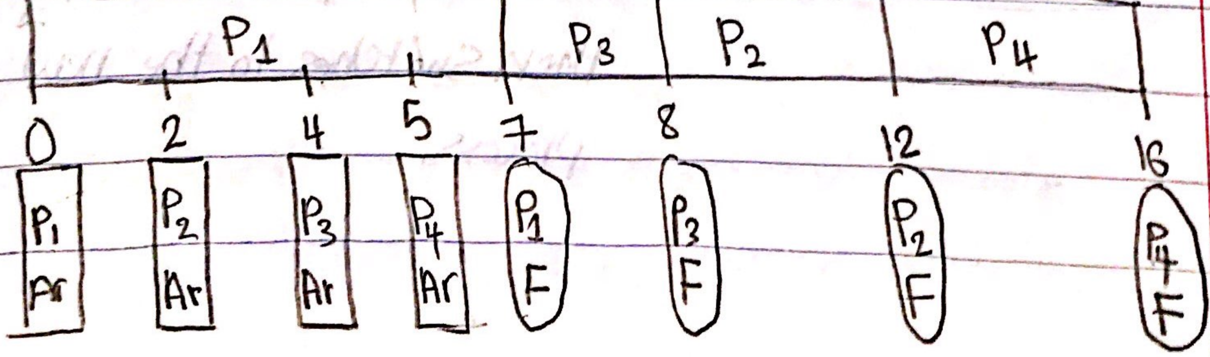
(a) Preemptive:



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

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

(b) Non-Preemptive:



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

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

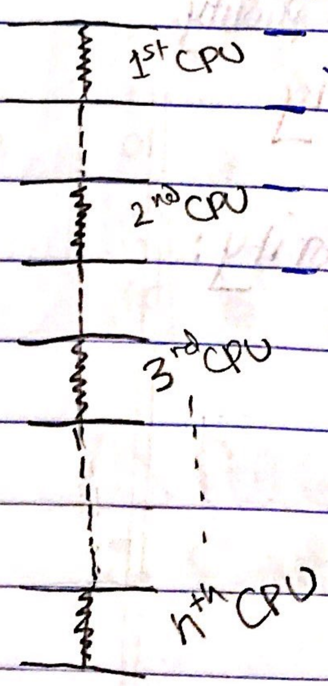
⚠ Problem: Starvation ∴ Solution: Aging

→ 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 n^{th} CPU burst.
- Y_n = estimated Length of the n^{th} 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:

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

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

* 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:

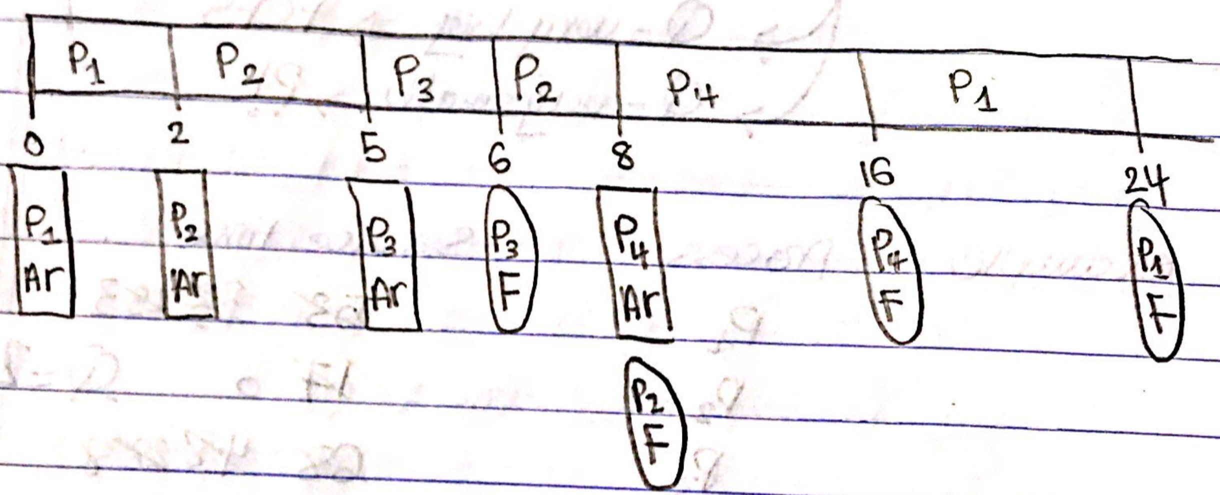
- The CPU is given to the process with the high priority.
- Every process is given a priority number.
  Generally, low number means low priority.
  → System tasks have high priority
  ↘ (i.e. interrupts)
- There are Two versions of priority:
  (a) Preemptive
  (b) Non-preemptive

example: given the queue as follows:

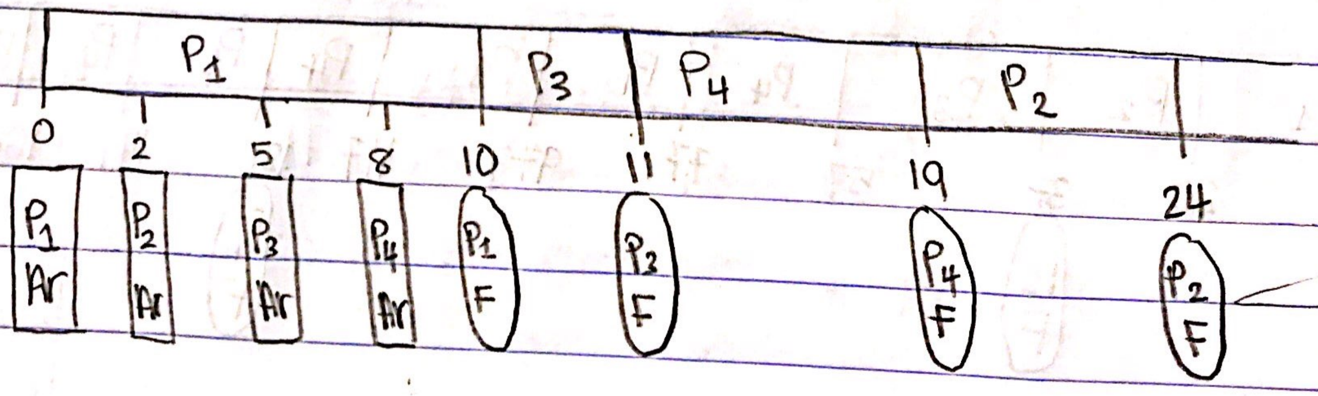
process	CPU burst	priority	arrival time
P_1	~~10~~ 8	1	10:00
P_2	~~5~~ 2	2	10:02
P_3	~~1~~0	5	10:05
P_4	8	4	10:08

⚠ 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~~ ~~28~~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 (at 37), P_4 F (at 121), P_1 F (at 164), P_3 F (at 172)

[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?!

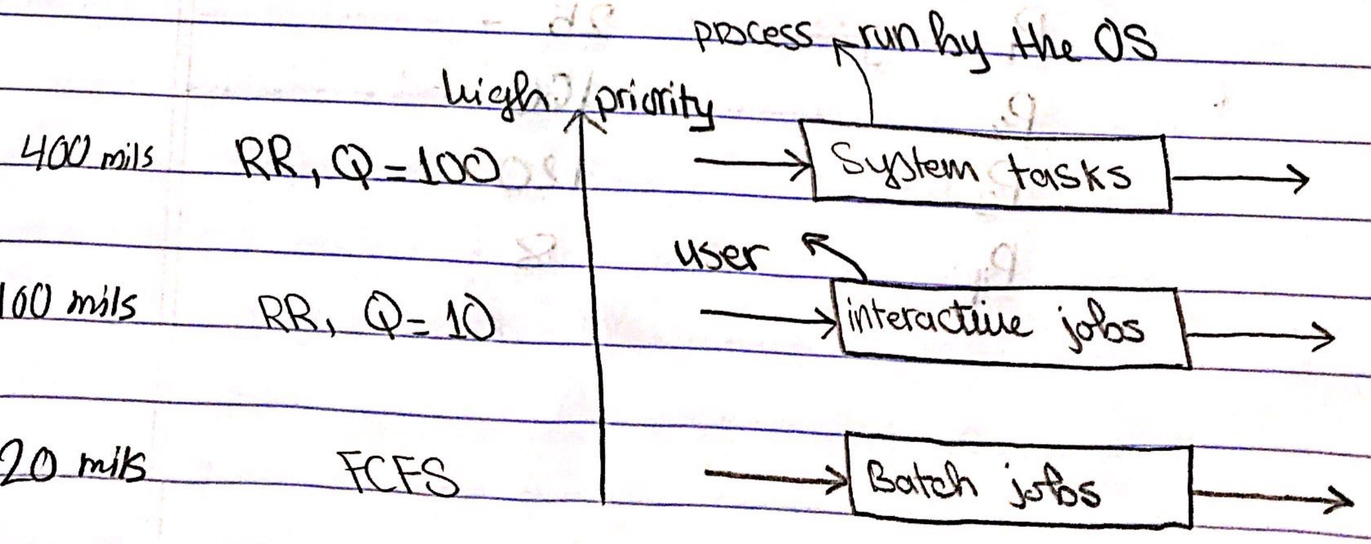
↳ there are two Algorithms:

(1) time slice:

Each queue is assigned a 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.

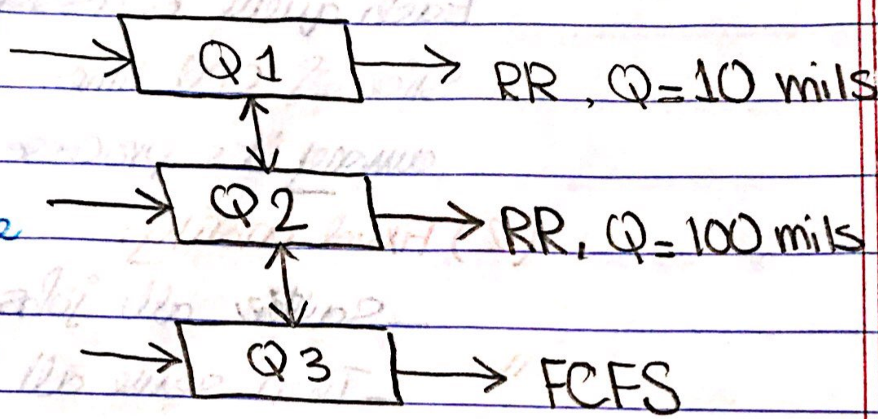
March 21.2018

[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
P_1	25
P_2	160
P_3	120
P_4	8

P1	P2	P3	P4	P1	P2	P3	P2	P3

0, 10, 20, 30, 38, 53, 153, 253, 303, 313

P1
P2
P3
Ar

P4 F (38) — P1 F (53) — P2 F (303) — P3 F (313)

Algorithm Evaluation :-

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

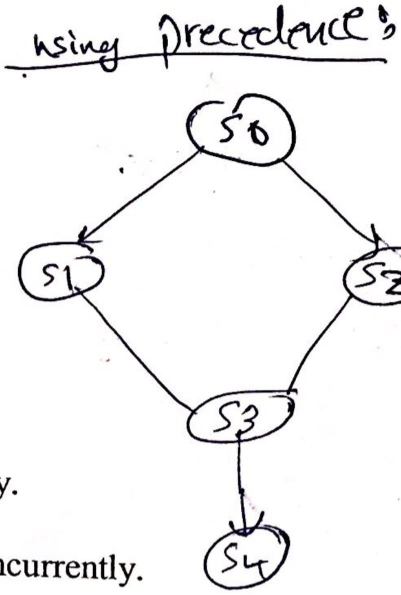
Chapter 6

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:



Given the following statements:

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

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.

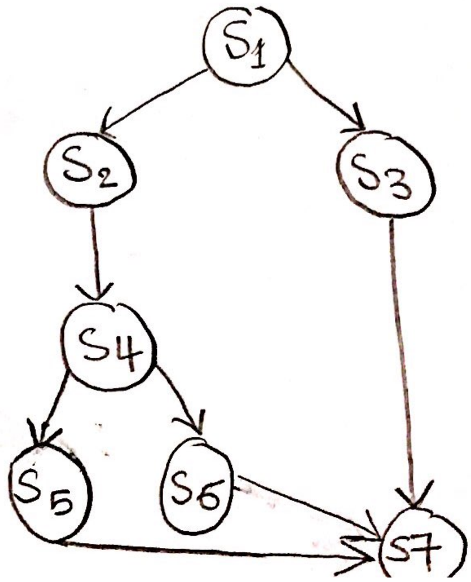
- But statements (1) & (2) can be executed concurrently.

- 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:

W R

- 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)
  R(S) ={a}
  W(S) ={a}

* S: read(a)
R(S) = {a}
W(S) = { }, ∅

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:

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

Example:

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

$R(S_1) = \{x,y\}$
$W(S_1) = \{a\}$
$R(S_2) = \{z\}$
$W(S_2) = \{b\}$ ← b
$\{x,y\} \cap \{b\} = \varnothing$
$\{z\} \cap \{a\} = \varnothing$ → a
$\{a\} \cap \{b\} = \varnothing$

Example:

Given,

$S_3 : c = a - b$

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

Fork & Join Constructs:

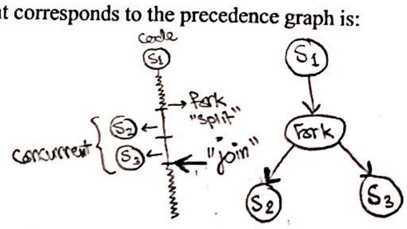
- Precedence graph is difficult to use in Programming Languages, so other means must be provided to specify precedence relation.

- 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:

S_1;
Fork L;
S_2 ;
.
:
L: S_3 ;

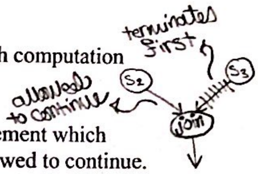
} Concurrent



(*) 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

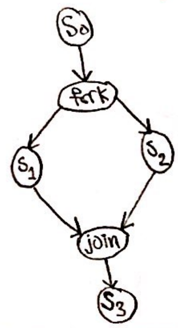
  count = count – 1;
  If count ≠ 0 then quit (quit this computation)

→ 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.**

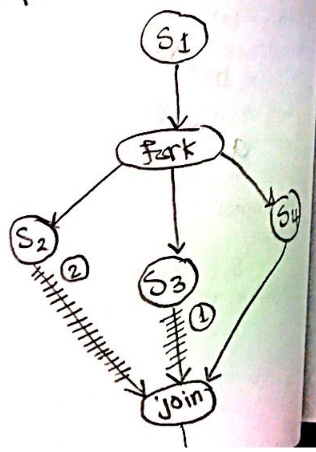
- For two processes:

  Count =2
  Fork L_1;



Count = # of computations to join;
function join ;
{
⋮
Count = Count – 1 ;
if (count != 0)
Quit (stop) Computation;
}

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



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;      ]  > Concurrent
L1 : S2           ]
L2 : join count
```

(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;                 (concurrent)
  [L1 : b=z+1;
  L2 : joint count;  -> (join)
    c  =a-b;
    w  =c+1;
```

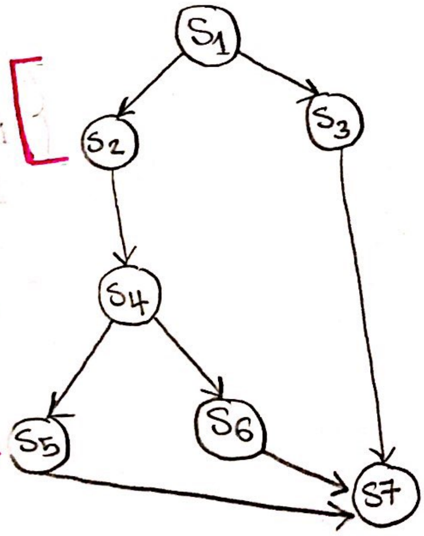
(fork) → a= x+y ; } Concurrent.
→ b= z+1 ;
c= a-b ;
w= c+1 ;

- For the precedence graph earlier:

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

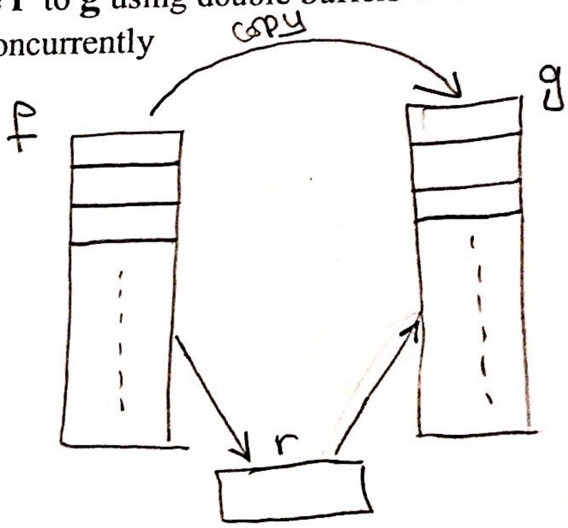
Fork L1 (S3)

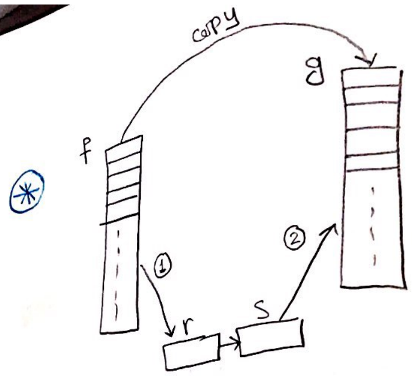
Fork L2 (S6)



- 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;
    L1: read (f,r);
    L2: join count;
     End;
    Write (g,r);
  End;
```

(concurrent statements: Write (g, s); goto L2; / L1: read (f,r);)

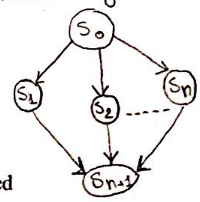
The concurrent statement:

- 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;
    S2;
     :
    Sn;
Parend;
Sn+1;
```

(these statements are executed concurrently.)



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

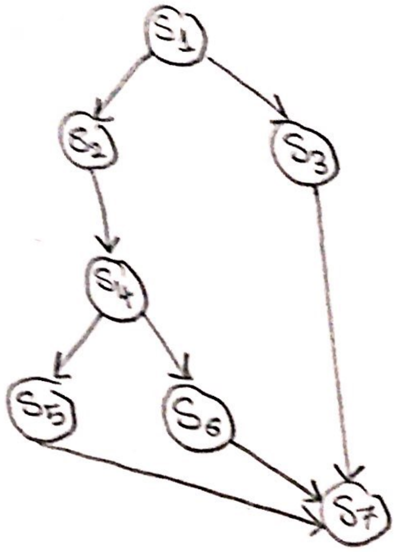
(*) In our pervious example,

```
parbegin
   a  = x+y;
   b  = z+1;
parend ;
   c =a-b;
   w =c+1;
```

(a, b: concurrently. c, w: not concurrently.)

(*) In the example:

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



(*) 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:	Consumer:
do { ... produce an item in nextp ... while ((in+1)%n ==out) no-op; // full buffer buffer[in] = nextp; in = (in + 1) % n; } while true;	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:	**Consumer:**
<pre>do { ... produce an item in nextp ... while (counter == n) → buffer is full no-op; // busy waiting ↺ buffer[in] = nextp; in = (in + 1) % n; counter = counter + 1; } while true; ↓ Critical Section</pre>	<pre>do { while (counter == 0) → buffer is empty no-op; // busy waiting ↺ nextc = buffer[out]; out = (out + 1) % n; counter = counter - 1; ... consume the item in nextc ... } while true; Critical Section</pre>

- 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 — 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 {count = 6 }
S5: consumer execute counter = register2 {count = 4}

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