Parameters for Evaluation of CPU scheduling Algo-

① **CPU utilization** – Percentage of time CPU is busy in execution / doing useful work.

$$CPU\% = \frac{\text{useful time}}{\text{total time}} \times 100$$

② **Response time** → Time that a process has to wait in Ready Queue.

Response time = Process start time − Arrival time

③ **Turnaround Time** →

TAT = Process completion − Process submission (Arrival time)

④ **Waiting time** →

WT = TAT − Burst time

→ In Response time, we consider only the first time CPU has been allocated.

⑤ **Throughput** → $TP = \frac{\text{No. of processes completed}}{\text{Total CPU time}}$

1 FCFS (First come First serve) →

→ non-preemptive scheduling Algo.
→ criteria is Arrival time

Example →

1

Process	AT	BT	ST	CT ~~TAT~~	TAT	WT
P_1	0	4	0	4	4	0
P_2	2	3	9	12	10	7
P_3	1	5	4	9	8	3
P_4	4	2	12	14	10	8

P_1	P_3	P_2	P_4

0   4   9   12   14

Gantt's chart

Response time,
$P_1 - 0-0 = 0$
$P_2 - 9-2 = 7$
$P_3 - 4-1 = 3$
$P_4 - 12-4 = 8$

Avg. Response time $= \frac{0+7+3+8}{4} = \frac{18}{4} = 4.5$

Avg. TAT $= \frac{4+10+8+10}{4} = \frac{32}{4} = 8$

Avg. WT $= \frac{0+7+3+8}{4} = \frac{18}{4} = 4.5$

CPU% $= \frac{14}{14} \times 100 = 100\%$ [utilization]

throughput $= \cancel{\frac{14}{4}} = \frac{4}{14} = 0.28$

* If two process have same AT, then process with small PID would be selected.

②

	PID	A.T.	B.T	C.T.	R.T.	T.A.T.	W.T.
P_1	1	3	3	14	8	11	8
P_2	2	2	4	6	0	4	0
P_3	3	1	1	2	0	1	0
P_4	4	2	5	11	4	9	4
P_5	5	4	6	20	10	16	10

	P_3	P_2	P_4	P_1	P_5
0 – 1	1 – 2	2 – 6	6 – 11	11 – 14	14 – 20

Avg. R.T. $= \frac{8+0+0+4+10}{5} = \frac{22}{5}$

Avg. TAT $= \frac{11+4+1+9+16}{5} = \frac{41}{5}$

Avg. W.T $= \frac{8+0+0+4+10}{5} = \frac{22}{5}$

CPU % $= \frac{19}{20} \times 100 = 95\%$

Throughput $= \frac{5}{20} = 0.25$

③

Process	A.T.	B.T
P_1	0	4
P_2	3	2
P_3	7	6
P_4	10	3
P_5	2	1
P_6	3	1

Avg. TAT = 5.16

Avg. WT = 2.3

Avg. RT = 2.3

CPU = $(17/20) \times 100\%$

throughput $= \frac{6}{20}$

convoy effect → In FCFS, if the 1st process is having large burst time then it will have a huge impact on Avg. waiting time of all the remaining processes.

(2) SJF (shortest Job first) →

→ ~~Pr~~ non-preemptive.
→ criteria is the burst time.

Example –

Process	AT	BT	CT	RT	TAT	WT
P_1	1	7	8	0	7	0
P_2	2	5	16	9	14	9
P_3	3	1	9	5	6	5
P_4	4	2	11	5	7	5
P_5	5	8	24	11	19	11

	P_1	P_3	P_4	P_2	P_5
0	1	8	9	11	16	24

$$CPU\% = \frac{23}{24} \times 100 \qquad Avg.\ TAT = \frac{53}{5} = 10.6$$

$$throughput = \frac{5}{24} \qquad Avg\ WT = \frac{30}{5} = 6$$

* if there are two process with same burst time, then schedule the process as per Arrival time.

(3) Pre-emptive SJF (SRTF — shortest Remaining time first) →

Example —

Process	A.T.	B.T.	C.T.	RT	TAT	WT
P_1	0	~~7~~ 6	19	0	19	12
P_2	1	~~5~~ 4	13	0	12	7
P_3	2	~~3~~ ~~2~~ ~~1~~ 0	6	0	4	1
P_4	3	~~1~~ 0	4	0	1	0
P_5	4	~~2~~ 0	9	3	5	3
P_6	5	~~1~~ 0	7	1	2	1

P_1	P_2	P_3	P_4	P_3	P_3	P_6	P_5	P_2	P_1

0 1 2 3 4 5 6 7 9 13 19

Avg. TAT = 7.16

Avg WT = 4

Avg RT = $\frac{4}{6}$ = 0.67

CPU% = 100%

Throughput = $\frac{6}{19}$

(2)

Process	A.T.	B.T.	CT	TAT	WT
P_1	3	~~1~~ 0	13	10	6
P_2	4	~~1~~ 0	9	5	3
P_3	5	~~1~~ 0	7	2	1
P_4	2	~~6~~ 0	19	17	11
P_5	1	~~8~~ 7	26	25	17
P_6	2	~~4~~ ~~3~~ ~~2~~ ~~1~~ 0	6	4	0

Avg TAT = $\frac{63}{6}$

Avg WT = $\frac{38}{6}$

	P_5	P_6	P_6	P_6	P_6	P_3	P_2	P_1	P_4	P_5

0 1 2 3 4 5 6 7 9 13 19 26

4. Round-Robin scheduling →

→ Pre-emptive

→ Time-Quantum / slice is the criteria.

Example — Quantum = 2

Process	A.T.	B.T.	CT	TAT	WT
P_1	0	~~4~~ ~~2~~ 0	13	13	9
P_2	1	~~5~~ ~~3~~ ~~1~~ 0	19	18	13
P_3	2	~~2~~ 0	6	4	2
P_4	3	~~1~~ 0	7	4	3
P_5	4	~~6~~ ~~4~~ ~~2~~ 0	21	17	11
P_6	6	~~3~~ ~~1~~ 0	18	12	9

P_1	P_2	P_3	P_4	P_5	P_6	P_1	P_2	P_5	P_6	P_2	P_5

0 2 4 6 7 9 11 13 15 17 18 19 21

$$\text{Avg TAT} = \frac{13+18+4+4+17+12}{6} = \frac{68}{6} = 11.33$$

$$\text{Avg WT} = \frac{9+13+2+3+11+9}{6} = \frac{47}{6} = 7.83$$

2. TQ=3

Process	AT	BT	CT	RT	TAT	WT
P_1	5	~~5~~ ~~2~~ 0	32	10	27	22
P_2	4	~~6~~ ~~3~~ 0	27	5	23	17
P_3	3	~~7~~ ~~4~~ 1	33	3	30	23
P_4	1	~~4~~ ~~1~~ 0	30	0	29	20
P_5	2	~~2~~ 0	6	2	4	2
P_6	6	~~3~~ 0	21	12	15	12

ART = 5.33
ATAT = 21.33
AWT = 16
CPU% = 96.96%
Throughput = 0.18

	P_4	P_5	P_3	P_2	P_4	P_1	P_6	P_3	P_2	P_4	P_1	P_3

0 1 4 6 9 12 15 18 21 24 27 30 32 33

RQ : ~~P1~~, ~~P5~~, ~~P3~~, ~~P2~~, ~~P4~~, ~~P1~~, ~~P6~~, ~~P3~~, ~~P2~~, ~~P4~~, ~~P1~~, ~~P5~~

(2)

Process	AT	BT	CT	RT	TAT	WT
P_1	0	~~4~~ ~~2~~ 0	8	0	8	4
P_2	1	~~5~~ ~~3~~ ~~1~~ 0	18	1	17	12
P_3	2	~~2~~ 0	6	2	4	2
P_4	3	~~1~~ 0	9	5	6	5
P_5	4	~~6~~ ~~4~~ ~~2~~ 0	21	5	17	11
P_6	6	~~3~~ ~~1~~ 0	19	7	13	10

$\boxed{TQ = 2}$

RQ : ~~P1~~, ~~P2~~, ~~P3~~, ~~P1~~, ~~P4~~, ~~P5~~, ~~P2~~, ~~P6~~, ~~P5~~, ~~P2~~, ~~P6~~, P5

P_1	P_2	P_3	P_1	P_4	P_5	P_2	P_6	P_5	P_2	P_6	P_5
0–2	2–4	4–6	6–8	8–9	9–11	11–13	13–15	15–17	17–18	18–19	19–21

Avg RT = 20/6   CPU% = 100%

Avg TAT = 65/6   throughput = 6/21

Avg WT = 44/6

* In Round-Robin,
  - → If TQ decreases, no. of context switch increases and average RT also increases.
  - → If TQ increases too much, then it will start becoming

5 Longest Job first (LJF) →

→ non-preemptive
→ criteria is Burst-time.

Ex-

1

Process	AT	BT	CT	RT	TAT	WT
P_1	0	3	3	0	3	0
P_2	1	~~4~~5	14	8	13	8
P_3	2	4	9	3	7	3
P_4	3	~~5~~2	5	10	2	0
P_5	4	6	20	10	16	10

ART = 21/5
ATAT = 41/5
AWT = 21/5
CPU% = 100%
throughput = 5/20

P_1	P_4	P_3	P_2	P_5
0 – 3	3 – 5	5 – 9	9 – 14	14 – 20

2

Process	A.T.	B.T.
P_1	3	3
P_2	1	1
P_3	2	3
P_4	4	2
P_5	6	6
P_6	2	4

ATAT = 55/6
AWT = 36/6

6 Longest Remaining time first (LRTF) →

→ preemptive version of LJF.
→ criteria is remaining burst time.

Ex - 1

Process	AT	BT	RT	CT	TAT	WT
P_1	1	~~2~~ ~~1~~ 0	0	18	17	15
P_2	2	~~4~~ ~~3~~ ~~2~~ ~~1~~ 0	0	19	17	13
P_3	3	~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0	0	20	17	11
P_4	4	~~8~~ ~~7~~ ~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0	0	21	17	9

	P_1	P_2	P_3	P_4	P_3	P_4	P_3	P_4	P_2	P_3
0–1	1–2	2–3	3–4	4–7	7–8	8–9	9–10	10–11	11–12	12–13

P_4	P_2	P_3	P_4	P_1	P_2	P_3	P_4
13–14	14–15	15–16	16–17	17–18	18–19	19–20	20–21

Avg RT = 0

Avg TAT = 17

Avg WT = 12

CPU % = $\frac{20}{21} \times 100 = 95.23\%$

throughput = $4/21 = 0.19$

2

Process	AT	BT
P_1	0	2
P_2	2	4
P_3	7	5

P_1	P_1	P_2		P_3
0–1	1–2	2–6	6–7	7–12

(7) Highest Response Ratio Next (HRRN) →

[non-Preemptive]

$$\text{Response Ratio} = \frac{w+s}{s}$$

w — waiting time of a process at time 't'
s — Burst time of that process

* when Burst time is less / or, waiting time is large, then Response Ratio is more. so, it favours shorter Jobs as well as limits the waiting time.

Ex —

Process	AT	BT	CT	RT	TAT	WT
P_1	0	4	4	0	4	0
P_2	3	8	12	1	9	1
P_3	6	6	18	6	12	6
P_4	9	7	28	12	19	12
P_5	12	3	21	6	9	6

ART = 5
ATAT = 10.6
AWT = 5
CPU% = 100%
Throughput = $\frac{5}{28}$

P_1	P_2	P_3	P_5	P_4
0–4	4–12	12–18	18–21	21–28

At t=12

$$RR[P_3] = \frac{6+6}{6} = 2 \quad [6 - \text{waiting time } [AT - 6, \text{ we are at } 12]]$$

$$RR[P_4] = \frac{3+7}{7} = \frac{10}{7} = 1.42$$

$$RR[P_5] = \frac{0+3}{3} = 1$$

At t=18

$$RR[P_4] = \frac{9+7}{7} = \frac{16}{7} = 2.28$$

$$RR[P_5] = \frac{6+3}{3} = \frac{9}{3} = 3$$

Ques.

Process	AT	BT
P_1	1	1
P_2	4	2
P_3	5	6
P_4	6	2
P_5	7	4

ATAT = 5.8

AWT = 2.8

	P_1		P_2	P_3	P_4	P_5
0 – 1	1 – 2	2 – 4	4 – 6	6 – 12	12 – 14	14 – 18

---

(8) Priority scheduling (non-preemptive) →

→ criteria is priority.

→ mode is non-preemptive.

[Assumption
Highest priority is
largest no.]

Ques.

Process	Priority	AT	BT
P_1	4	0	4
P_2	5	1	5
P_3	7	2	1
P_4	2	3	2
P_5	1	4	3
P_6	6	5	6

P_1	P_3	P_6	P_2	P_4	P_5
0 – 4	4 – 5	5 – 11	11 – 16	16 – 18	18 – 21

* If priority is same, then use Arrival time.

Ques.

Process	Priority	AT	BT	CT
P_1	4	4	6	17
P_2	5	6	3	11
P_3	6	3	4	8
P_4	6	2	2	4
P_5	7	1	1	2
P_6	3	2	2	19

AWT = 4.16
ATAT = 7.16
CPU% = 94.73%
throughput = 6/17

//////	P_5	P_4	P_3	P_2	P_1	P_6

0 1 2 4 8 11 17 19

$$ATAT = \frac{13+5+5+2+1+17}{6} = \frac{43}{6} = 7.16$$

$$AWT = \frac{7+2+1+0+0+15}{6} = \frac{25}{6} = 4.16$$

---

(9) Pre-emptive priority scheduling →

Ques.

Process	Priority	AT	BT	CT	TAT	WT
P_1	4	1	~~4~~ 3	18	17	13
P_2	5	2	~~1~~ 0	14	12	10
P_3	7	2	~~3~~ ~~1~~ 0	10	8	5
P_4	8	3	~~5~~ 0	8	5	0
P_5	5	3	~~1~~ 0	15	12	11
P_6	6	4	~~2~~ 0	12	8	6

ATAT = 10.33
AWT = 7.5
CPU% = 94.44%
throughput = $\frac{6}{18}$

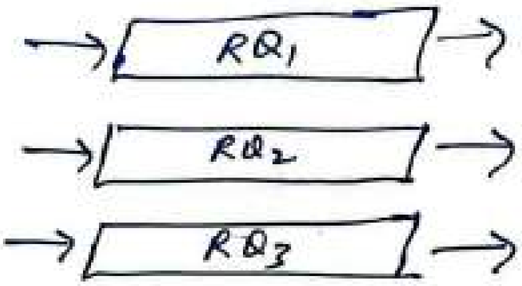
//////	P_1	P_3	P_4	P_3	P_6	P_2	P_5	P_1

0 1 2 3 8 10 12 14 15 18

(10) multi-level ~~feedback~~ Queue scheduling →

→ multiple Ready Queue are maintained.

→ Generally, High priority processes are kept in top-level ready Queue and low priority in low level RQ.

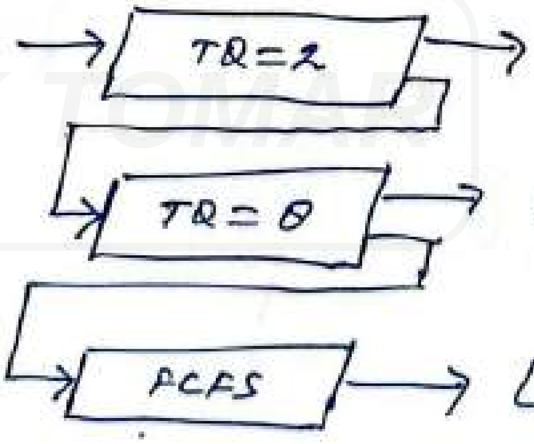


→ only after top-level RQ completion, further level Queue will be processed.

→ In Ready Queue, any other previous strategy can be used.

→ If this strategy is to be used then bottom-level RQ processes will suffer from starvation.

(11) multilevel feedback Queue scheduling →



[few process would complete here]

[Processes with higher req. will be shifted here]

[most of the processes would be completed before coming to this Queue]

→ This will avoids starvation and at the same time would prefer high-priority processes.

Burst Time prediction

→ difficult part to predict time.
→ can be predicted with some accuracy.

① static techniques →

ⓐ on the basis of process type

- system processes → generally less interactive so allocate more cpu time
- user processes
  - interactive [less cpu time]
  - less-interactive (Background process) [more cpu time]

ⓑ on the basis of process size

suppose, P1 is there with size 100 kb and taken 10 ms to complete so allocate the same time to same sized processes.

② dynamic techniques →

ⓐ simple averaging method →

suppose, P_1, P_2 and P_3 have taken total

21 ms then we will assign 21/3 = 7 ms to the new process P_4.. $\left[\tau_{n+1} = \left(\sum_{i=1}^{n} t_i\right) \times \frac{1}{n} \right]$

(2) Exponential averaging →

$$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha)\, \tau_n$$

τ_n = Estimated CPU burst for n^{th} process
τ_{n+1} = " " " " $(n+1)^{th}$ process
t_n = Actual Burst time for n^{th} process
α — smoothing factor (0–1)

Ques. $\alpha = 0.6$, $\tau_1 = 10$ ms
$[t_1 = 5,\ t_2 = 9,\ t_3 = 5]$ Actual burst time of P1, P2, P3

$$\boxed{\tau_4 = \alpha \cdot t_3 + (1-\alpha)\, \tau_3}$$

$\tau_2 = \alpha t_1 + (1-\alpha)\tau_1 = 0.6 \times 5 + 0.4 \times 10 = 7.0$
$\tau_3 = \alpha \cdot t_2 + (1-\alpha)\tau_2 = 0.6 \times 9 + 0.4 \times 7 = 8.2$

$\tau_4 = \alpha \cdot t_3 + (1-\alpha)\tau_3 = 0.6 \times 5 + 0.4 \times 8.2$

$$\boxed{\tau_4 = \boxed{6.28 \text{ ms}}}$$

Principle of concurrency → In multiprogramming environment, we may have multiple processes that executes concurrently. Processes executing concurrently in the OS can be categorize into two categories—

1) Independent processes
2) cooperating processes

1) Independent processes → A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data or program code with any other process is independent.

2) cooperating processes → A process is cooperating if it can affect or be affected by other processes executing in the system. Clearly, any process that shares data or program code with other processes is a cooperating process.

* so, in multiprog. systems, one of the goal is to provide an environment (Interprocess communication mechanism) because of —

1) Information sharing → since many processes are interested in same information (e.g. shared file), we must allow concurrent access.

2) computation speedup → if we want a particular task to run faster, it is to be break into subtasks. these subtasks needs to be executed concurrently.

3) convenience → using the concurrent execution, a user can be allowed to perform multiple tasks.

Interprocess communication (IPC) → cooperating processes requires an IPC mechanism, which allows them to share data, program code and resources. An OS must implement IPC for supporting concurrent execution of cooperating processes.

there are basically two fundamental models for IPC—

1.) shared memory model
2.) message passing model

* Both of the above models are common in OS. some OS implement both these models.

* In the shared memory model, a shared memory region is established and processes can then exchange info. by reading and writing data to the shared region. It allows convenience of commⁿ, as the kernel is only required to establish shared memory regions. It is also faster than message passing model.

* In message passing model, commⁿ b/w cooperating processes takes place by means of exchanging msgs. It is useful for exchanging smaller amount of data. It is slow as compared to shared memory models as these are implemented using system calls and thus requi
—res intervention of kernel on a regular basis.

1) <u>message passing model</u> → This model is useful in distributed environment, where the processes can reside on diff. computers connected through a network. In this, processes communicates only by sending & Receiving mess
—ages and they do not share address space.

* A msg passing facility provides two operations—

send(destination, msg)

Receive (source , msg)

* if two processes P1 & P2 wants to communicate, they must send & Receive msgs. e.g ⇒ In a chat program, processes communicates through msgs.

A commⁿ link should be there b/w the communicating processes.

There are basically three ways in which msg passing model can be implemented—

a) Direct or Indirect communication.

b) synchronous or Asynchronous commⁿ.

c) Bounded or unbounded buffered commⁿ.

a) <u>Direct or Indirect commⁿ</u> → In Direct commⁿ, the processes that wants to communicate must explicity name the other process. i.e. if P1 wants to send a msg to P2 then —

→ send (P2, msg) - send a msg to P2.

→ receive (P1, msg) - Receive a msg from P1.

The commⁿ link will have the following properties—

1) link is associated with only two processes, that know each other's identity.

2) Between two processes, there is only exacty one link.

* The disadvantage of this scheme is changing the identifier of one process, all the refferences to this identifier is to be identified & updated.

* In Indirect commⁿ, the msgs are send and received to and from mailboxes. A mailbox is an abstract object where processes can place their msgs. The o/n will be—

→ send (A, msg) — send msg to mailbox A.

→ Receive (A, msg) - Receive msg from mailbox A.

* A mailbox may be owned either by a process (as a part

of the add. space of process) or by the OS (independent)
If the mailbox is a part of the OS, then OS must provide following operations—

1) create & delete mailbox.
2) send & Receive msg to and from mailbox.

* In Indirect commn, processes commn link has following properties—

1) link may be associated with more than two processes.
2) link can be established, when processes shares a mailbox.
3) there may be multiple links associated with a process.

2) Synchronous and Asynchronous communication → the commn b/w two processes can be synchronous or asynchronous.

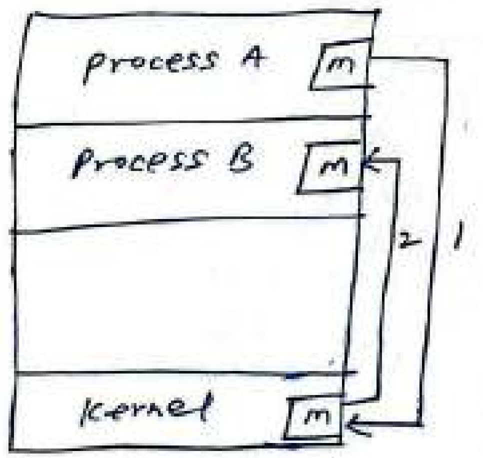
* synchronous commn means blocking send and blocking Receive.
* Asynchronous commn means non-blocking send and non-blocking Receive.

a) Blocking send → sender process is blocked until the msg is Received.
b) non-Blocking send → sender process sends the msg and Resumes operation.
c) blocking Receive → Receiver process is blocked until a msg is available.
d) non-Blocking Receive → Receiver Receives either a valid msg or a null.

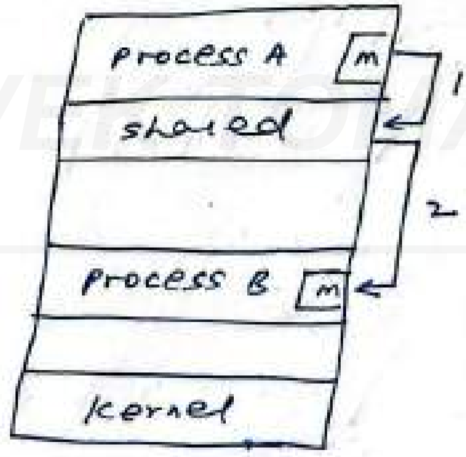
c) Bounded or unbounded buffered commⁿ → If the commⁿ is asynchronous (direct or Indirect), msg exchanged b/w processes resides in a Queue. These queues can be impl-emented in three ways –

1) Zero capacity → These queues are used in synchrono-us commⁿ.
2) Bounded capacity → these queues have limited capacity.
3) unbounded capacity → these Queues are used in asyn-chronous commⁿ. The sender never blocks.

2) shared memory model → In this model, a shared memory region is required. Typically, this region resi-des in the address space of the process, that starts the commⁿ. other process that takes part in the commⁿ must attach itself to this address space.



a) message passing



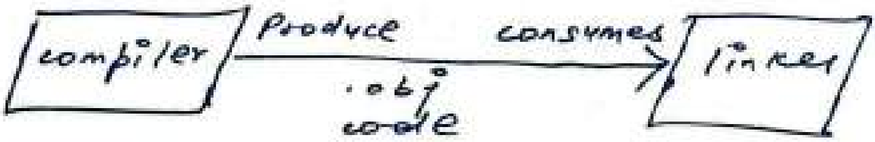
b) shared memory

* to illustrate the concept of shared memory commⁿ, let us take the example of producer-consumer problem.

Producer – consumer Problem → In this, there are two types of processes, i.e.

A: Producer and A consumer.

A. Producer produces information that is consumed (used) by the consumer. e.g. => A compiler produces code, which is consumed by linker and linker produces the code which is consumed by loader.



* one soln to the problem uses shared memory model. To allow producer & consumer to run concurrently, we must have a buffer, where producer can produce the info. and consumer can consume info. from there. The producer and consumer must be synchronized in a way that consumer does not try to consume the item, that has not been produced yet.

* To achieve this, we can have two types of buffers—

1) unbounded buffer -> In this, consumer has to wait for the arrival of new items, but the producer can always produce.

2) Bounded buffer -> The bounded buffer is implemented as a circular array with two logical pointers — in and out. in points to the next free position and out points to the first full position.

-> (in == out) — Buffer is empty

-> ((in+1) % Buffer_size) == out — Buffer is full.

Producer implementation ->

while (1)

```
{  while(((in+1) % buffer_size) == out)
        /* do nothing */
   buffer [in] = next produced item;
   in = (in+1) % buffer_size;
}
```

consumer implementation →

```
while (1)
{
     while (in==out)
        /* do nothing */
  next consumed item = buffer [out];
  out = (out+1) % buffer_size;
}
```

---

critical section problem → when there are cooperating processes in the system, they may be sharing data, program code and resources.

critical section → A critical section of a process is the segment of code, in which the process may be changing common variables, updating a table, writing into a shared file and so on.

suppose two or more processes are trying to access a single resource like printer or trying to write to a shared file. this type of resource is reffered as critical resource and the portion of program which is reffering to this resource is critical section of the program.

* thus, at any moment, only one process can execute in its cs otherwise the race condition will occur.

<u>Race condition</u> → A Race condition occurs when multiple processes read and write data items so that the final result is dependent on the execution of instructions. In this, all the processes tries to write the data at the last and the final result becomes unstable and non-pred-ictable.


```
do
{
      [entry section]
            critical section
      [exit section]
            Remainder section
} while(1);
```

* the critical section problem is about designing a protocol that can be used by cooperating processes. Each process must take permission to entry into its critical section. the section of code implementing this request is entry section. the critical section is to be followed by exit section. the remaining code i.e. in which process is not trying to access any shared data/resource is in remainder section.

A solution to the critical section problem must satisfy the following three requirements—

1) <u>mutual exclusion (MUTEX)</u> → if we have 'n' cooperating

processes ($P_1, P_2, -- P_n$) and process P_i is executing its critical section, then no other process is executing its critical section at that time.

That means the execution of critical section by the cooperating processes should be mutually exclusive.

2) Progress → if no process is executing in its CS and some process wish to execute its CS, then only those processes that are not executing their remainder sect-ion can participate in the decision (If there is a single process, then straightway it is able to execute).

3) Bounded waiting → There should be a limit on the number of times that other processes are allowed to execute their critical sections after a process has made a request to execute its critical section.

e.g. ⇒ suppose that three processes (P1, P2, P3) are cooperating. suppose P1 exits its CS, either P2 and P3 should be allowed. ~~Assume~~ that OS grants access to P3 and then again P1 requests for CS. if the OS again grants access to P1, then P2 may have to wait indefinitely. this condition is known as starvation.

---

solutions to critical section problem → there were various solutions implemented for CS problem.

let us assume, there are only two processes P1 and P2. then, one of the possible solution is —

1) solution 1 →

Algorithm →

var : turn

```
begin : turn = 1
  do
  {
    while (turn==2)
    /* do nothing */
    /* critical section */
    turn=2;
    /* Remainder section */
  } while(1);
```

Process P1

```
  do
  {
    while (turn==1)
    /* do nothing */
    /* critical section */
    turn=1;
    /* Remainder section */
  } while(1);
```

Process P2

* The shared variable turn is used to indicate which process can execute CS. If turn=1, P1 can straightway execute its CS and then it sets turn to '2', so enabling process P2 to execute its CS. (mutual exclusion fullfilled)
* This solⁿ also ensures Bounded waiting, since P1 has to wait for P2 to go through the CS exactly once.
* But, the problem comes in progress condition. suppose P1 is in its CS and P2 is in remainder section. If P1 exits from CS, finishes remainder cycle and wishes to enter the CS once again, but it would encounter a busy wait until P2 uses the CS.

2) Dekker's solution → In dekker's solⁿ, we uses three variables flag1, flag2 and turn. flag1 and flag2 will be enabled when P1 and P2 wish to execute their CS. if (flag1=1), then P1 checks for flag2. if flag2 is false, then P1 enters straightway. if flag2 is also

true, then P1 consults turn.

Dekker's Algorithm →

var : flag1, flag2, turn

begin : turn=1

```
do
{
  flag1 = 1 ;
  while( flag2 == 1)
  {
    if ( turn == 2)
    {
      flag1 = 0 ;
      while ( turn == 2)
      /* do nothing */
      flag1 = 1 ;
    }
  }
  /* critical section */
  turn = 2 ;
  flag1 = 0 ;
  /* remainder section */
} while (1);
```

Process P1

```
do
{
  flag2 = 1 ;
  while ( flag1 == 1)
  {
    if( turn == 1)
    {
      flag2 = 0 ;
      while ( turn == 1)
      /* do nothing */
      flag2 = 1 ;
    }
  }
  /* critical section */
  turn = 1 ;
  flag2 = 0 ;
  /* remainder section */
} while (1);
```

Process P2

* this Algo. ensures all the conditions for critical section. we will examine all the conditions one by one —

1) mutual exclusion → if P1 and P2, both wants to execute their cs. then flag1 = flag2 = 1. now, if turn is 1, then after examining the value of flag1

P2 will examine the value of turn. as the turn is 1, it will set flag2 as '0' and enables P1 to enter into its CS. P2 will remain waiting for the turn. after completion, P1 will set flag1 as '0' and turn as 2, enabling P2 to execute its CS.

2) <u>Progress</u> → Any process that wants to enter into its CS can straightway enter, if the other process do not wish to enter.

3) <u>Bounded waiting</u> → The process (P1/P2) have to wait only for once for other process.

---

3) <u>Peterson's solution</u> → Dekker's Algo. solves the critical section problem, but with a rather complex approach.

<u>Peterson's Algo.</u> →

var : flag1, flag2, turn

begin : turn=1

```
do
{
  flag1 = 1;
  turn = 2;
  while( flag2 && turn==2)
  /* do nothing */
  /* critical section */
  flag1 = 0;
  /* remainder section */
} while(1);
```

Process P1

```
do
{
  flag2 = 1;
  turn = 1;
  while( flag1 && turn==1)
  /* do nothing */
  /* critical section */
  flag2 = 0;
  /* remainder section */
} while(1);
```

Process P2

now, we will examine this Algo. for all three conditions—

1) <u>mutual exclusion</u> → since the turn is a Boolean var, so at any moment of time, its value can be either '1' or '2'. since, flag1 = flag2 = 1, so at any time the only one while cond^n can be false.

2) <u>progress</u> → if only P1 wants to execute its cs, then the value of flag2 will be 0. then, P1 will set flag1 as 1 and turn as 2, so the condition of while loop will false (flag2 = 0) and straightway P1 can execute its cs.

3) <u>Bounded waiting</u> → since, any process (P1/P2) will set its flag to '0', after executing its cs, so other process can enter into cs. so, other process has to wait maximum for once.

<u>solution of critical section problem for multiprocessor systems using test and set operations</u> → seeing the above solutions, we can say that any solution to the critical section problem requires lock. i.e. a process aquires a lock before entering into its cs and then releases the lock.

many modern computer system provides special H/W instructions testandset(), which we can use for critical section problem sol^n.

<u>Algorithm</u> →

```
boolean Testandset (boolean *target)
{
    boolean x = *target;
    *target = True;
```

```
    return x;
}
```

Definition of TestandSet() instruction

```
do
{
    waiting[i] = TRUE;
    key = TRUE;
    while( waiting[i] && key)
        key = TestandSet(&lock); /* getting value for
                                    lock variable */
    waiting[i] = FALSE;

    /* critical section */

    j = (i+1)%n;      /* n is the total no. of processes */
    while((j!=i) && (waiting[j]==FALSE))
        j = (j+1)%n;
    if(j==i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    /* Remainder section */

} while(1);
```

* This solⁿ satisfies all the requirements for a critical section problem solⁿ. but, unfortunately it is an difficult task to implement this solⁿ on multiprocessors systems is a difficult task.

semaphores → The main concern with the design of OS as a collection of cooperating sequential processes and devel-oping effecient tool for supporting cooperation.

Two or more processes can cooperate by means of simple signal, such that a process can be forced to stop at a specified place until it has received a specific signal. for signalling, specific variables called semaphores are used.

* A semaphore 's' is an integer variable, upon which only three atomic (indivisible) operations are defined:-

(1) Initialization → A semaphore may be initialized to a non-negative value.

(2) wait → wait operation decrements the semaphore value. if the value becomes negative, the process that has invoked wait o/n will be blocked.

(3) signal → signal o/n increments the value of sema-phore. if the value is less than or equals to zero, the blocked process will be unblocked.

* The definition for wait(s) and signal() is —

```
wait(s)                          signal(s)
{                                {
  while(s<=0)                       s++;
     /* no operation */
  s--;                           }
}
```

* All the modification to the value of semaphore can only be done by only wait() and signal() and these operations must be executed indivisibly.

```
do
{
    waiting (mutex);
    /* critical section */
    signal (mutex);
    /* Remainder section */
} while(1);
```

* In the above given code for the wait() and signal(), suppose we have two processes P1 and P2. the value of the 's' will be initialized to 1. if P1 wants to execute its CS, then it will invoke ~~sig~~ wait operation on 's'. then, it will decrement the semaphore value to '0' and execute its CS. In the meantime, if P2 also wants to execute its CS, then it will be in busy waiting state, till the P1 executes signal operation.

Types of semaphores → There are basically two types of semaphores—

1) Binary semaphores.
2) counting / general semaphores.

1) Binary semaphores → A Binary semaphore is the Restricted version. It may have only two values i.e. 0 and 1. on some systems, these are also known as mutex locks.

(a) A Binary semaphore is initialized to 1.

(b) the wait o/n checks the semaphore value. if the value is zero, then the process executing wait o/n is blocked. if the value is one, it is decremented to zero, and the process continues execution.

(c) the signal o/n checks if any process is blocked. If any, then it is unblocked. if no process is blocked, then the value of 's' is set to one.

② general / counting semaphore → It is a non-binary and non-restricted version of semaphore. The value of the counting semaphore can be over an unrestricted domain. these can be used to control access to a given resource consisting of a finite number of instances.

ⓐ semaphore value is initialized to the no. of instances of Resource.

ⓑ wait o/n decrements the value of semaphore by 1. if the value becomes zero, then after that processes requesting will be blocked.

ⓒ signal o/n increments the value of semaphore. if the value is zero, it unblocks a blocked process.

Implementation of semaphores → The main disadvantage of the semaphore definition given before is busy waiting. Busy waiting wastes cpu cycle as processes are in continous loop during the time in which other process is in its cs. These type of semaphores are known as spinlocks.

To overcome this, we can modify the definition of wait & signal as –

```
struct semaphore
{    int value;
     struct process *list;

};
```

```
wait(semaphore *s)
{
   s->value--;
   if(s->value < 0)
   {
      add the process to s->list;
      block();
   }
}
```

```
signal(semaphore *s)
{
   s->value++;
   if(s->value <= 0)
   {
      remove process from
                  s->list;
      wakeup(P);
   }
}
```

* In this, when a process executes a wait o/n and find the value as negative, it will block itself. The block o/n will put the process in a waiting queue associated with the semaphore, and the process state is switched to waiting.
* A process which is blocked is restarted, when some other process executes a signal o/n. The process is re-started by wakeup o/n, which will change its state to ready.
* The magnitude of the negative value of semaphore will tells the no. of processes waiting on that semaphore.
* The list of waiting processes can be implemented by the link field in PCB.

<u>strong and weak semaphores</u> → for both counting & Binary semaphores, a queue is used to hold processes waiting on that semaphore.

1) <u>strong semaphore</u> → If the processes are removed from that queue using some predefined

policy like FIFO, this semaphore is a strong semaphore.

* this type of semaphore does not lead to starvation.

2) weak semaphore → If there isn't any predefined policy and processes are removed randomly from the Queue, this will be a weak semaphore.

* this can lead to starvation of processes.

---

classical problems of synchronization →

① The Bounded-Buffer problem → In this problem, we assume that the buffer-size is bounded, lets say 'n'. There will be three semaphores associated with the buffer pool i.e. full, empty and mutEX. mutEX semaphore provides mutual exclusion for accesses to the buffer pool. The empty and full semaphores gives the no. of empty and full locations.

soln of Bounded-Buffer problem using semaphores →

var : full, empty, mutEX

Initial : full = 0, empty = buffer-size, mutEX = 1

```
do
{
  // produce an item in nextp
  ______;
  wait (empty);
  wait (mutex);
  // add nextp to buffer
  signal (mutex);
  signal (full)
} while(1);
```

```
do
{
  wait (full);
  wait (mutex);
  // remove an item from
  buffer.
  signal (mutex);
  signal (empty);
} while(1);
```

② The Readers-writers Problem → It is a other synchronization problem used to test synchronization tools.

In this, there is a data area shared among no. of processes. The data area could be a file, or a block of memory. The processes can be either readers (that reads the data area) or writers (writes to the data area).

The conditions that must be satisfied —

1) Any no. of readers may simultaneously read the file.
2) only one writer at a time may write a file.
3) if a writer is writing to the file, no reader may read the file.

* The readers-writers problem has many variations.
The first reader-writer problem requires that no reader will be kept waiting, unless a ~~writer~~ writer has already obtained a permission (i.e. readers has priority). In this, the writers may starve.
The second reader-writer problem requires that once a writer is ready, it performs writing as soon as possible (i.e. writer has priority). In this case, readers may starve.

* for these reasons, other variants of the problem have been proposed.

<u>solution of the first reader-writer problem</u> →

```
var : readcount, mutex, x
begin: readcount = 0, mutex = x = 1
 do
 {
    wait(x);
    readcount++;
```

```
 do
 {
    wait(mutex);
```

```
if(readcount == 1)
    wait(mutex);
signal(x);
/* performs reading */
wait(x);
readcount--;
if(readcount == 0)
    signal(mutex);
signal(x);
} while(1);
```

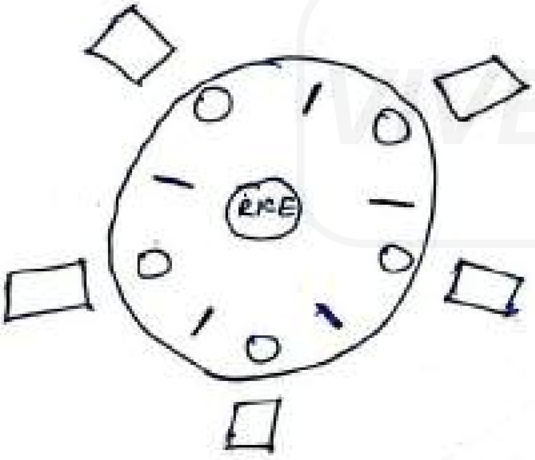
Reader process

```
/* performs writing */
signal(mutex);
} while(1);
```

writer process

* In the above solⁿ, the semaphore mutex is used to enforce mutual exclusion. As long as one writer is accessing the shared area, no other reader or writer can access it. However, to allow multiple readers, we require that when there are no readers reading, the first reader should wait on mutex. When there is atleast one reader reading, subsequent reader need not wait before entering. The global variable readcount is used to keep track of the no. of readers and the semaphore x is used to assure that readcount is updated properly.

3)· **The Dining philospher problem** → In this, five philospher live in a house, where a table is laid for them. The life of each philospher consists of thinking and eating. The philosphers share a single circular table surrounded by five chairs. In the centre of table is a bowl of rice and there are five chopsticks.



* When a philospher thinks, she does not interact with other philosphers. From time to time, a philospher gets hungry and tries to pick the two chopsticks closest to her. Obviously, she cannot pick the chopstick which is already in the hand of neighbour.

• When a philospher has two chopsticks, she eats without releasing the chopsticks.

The Dining philospher problem is a example of concurrency control problems. It is a simple representation of the need to allocate several resources among several processes in a starvation & deadlock free manner.

Solution using semaphores → one solⁿ is to represent each chopstick with a semaphore. A philospher tries to grab chopstick by invoking wait o/n and releases it by invoking signal operation.

Thus, the shared data are—

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1.


```
do
{
    wait ( chopstick[i] );
    wait ( chopstick([i+1]%5] );

    /* eat */

    signal ( chopstick[i] );
    signal ( chopstick[ (i+1)%5] );

    /* think */

} while (1);
```

<u>structure of philospher i</u>

* Although this soln gurantees that no two neighbours are eating simultaneously, but it can create deadlock suppose all five philospher becomes hungry simultaneously and all grabs their left chopsticks. When all philospher tries to grab their right chopstick, they will be delayed forever because all semaphores are now '0'.

* Some possible soln are—

1) Allow atmost four philosphers to sit simultaneously.
2) Allow a philospher to pick her chopsticks only if both chopsticks are available.
3) use an asymmetric soln like an odd philospher picks her first left chopstick and then right.