vibhu
PAGE No
DATE

Process Synchronization

- As there are many process running in CPU, there may be some processes which are using the common resource, so the common resource should be acquired only by a single process at a time, it is called as process synchronization

Why synchronization is needed?

→ When two process works together & share some common part with each other, there might be a case when both process alter values of the common part. & due to this the wrong value is updated.

- So using synchronization one process can use the common part at a time so that the value is not altered wrongly.

Race Condition

When two or more process are accessing some part if resultant value is dependent on order of execution of process, this situation is called Race condition.

Critical Section : It is any resource which is being shared between two or more resources. It can be hardware, variable, etc.

Modes of execution of process

serial | parallel

Process Synchronization

Process

Cooperative	Independent
(Execution of one process) affects other bcz they share something it can be variable, hardware, memory, code, resources etc. (cpu, printer, hardware)	• Execution of one process does not affect other, they are running independently

→ The cooperative processes needs to be synchronized, otherwise they can create some problems.

eg. int shared = ~~5~~ 6 4

• suppose P_1 start first.

Race Condition

	P_1			P_2	
1)	int x = shared	x = 5	1)	int y = shared	y = 5
2)	x++	x = 6	2)	y --	y = 4
3)	sleep(1)		3)	sleep(1)	Process sleeps/pre empt
4)	shared = x	Shared = 6	4)	Shared = y	Shared = 4

Process sleeps & CPU can't be idle so it executes P_2

Here initially shared = 5 in P_1 we incremented it by 1 & in P_2 we decremented it by 1

∴ the result should be same as shared = 5

but as both process shares some resource (variable) & the processes are not synchronized therefore the result is wrong

This problem is called **Race Condition**

∴ We needs some synchronization for this

Producer Consumer Problem

```
int count = 0;

void consumer()                          void producer()
{                                        {
    int itemc;                               int itemp;
    while (true)                             while (true);
    {                                        {
        while (count == 0);                      Produce_item (itemp);
        itemc = Buffer[out];                     while (count == n);
        out = (out+1) mod n;                     Buffer[in] = itemp
        count = count - 1;                       in = (in+1) mod n;
        Process_item (itemc);                    count = count + 1
    }                                        }
}
```

Buffer

0	
1	X_2
2	X_3
3	X_4
4	
5	
6	
7	

out: 1

in: 4

count: 2

consumer:
1) Load R_c, m[count]
2) DECR R_c
3) Store m[count], R_c

producer:
1) Load R_p, m[count]
2) INCR R_p
3) Store m[count], R_p

Case 1 (Normal Execution) : Suppose : count = 0

R_p=0, R_p=1, count=1 — R_c=1 R_c=0 count=0

Producer 1 2 3 Consumer 1 2 3

Case 2 : Producer 1 2 Consumer 1 2 producer 3 Consumer 3

R_p=3 R_p=4 R_c=3 R_c=2 count=4 count=2

Count = 2

(produced)

∴ Here initially, 3 items were present + 1 item added, 1 item consumed, so the count should be 3 but the count reduced to 2, therefore it is problem which needs synchronization.

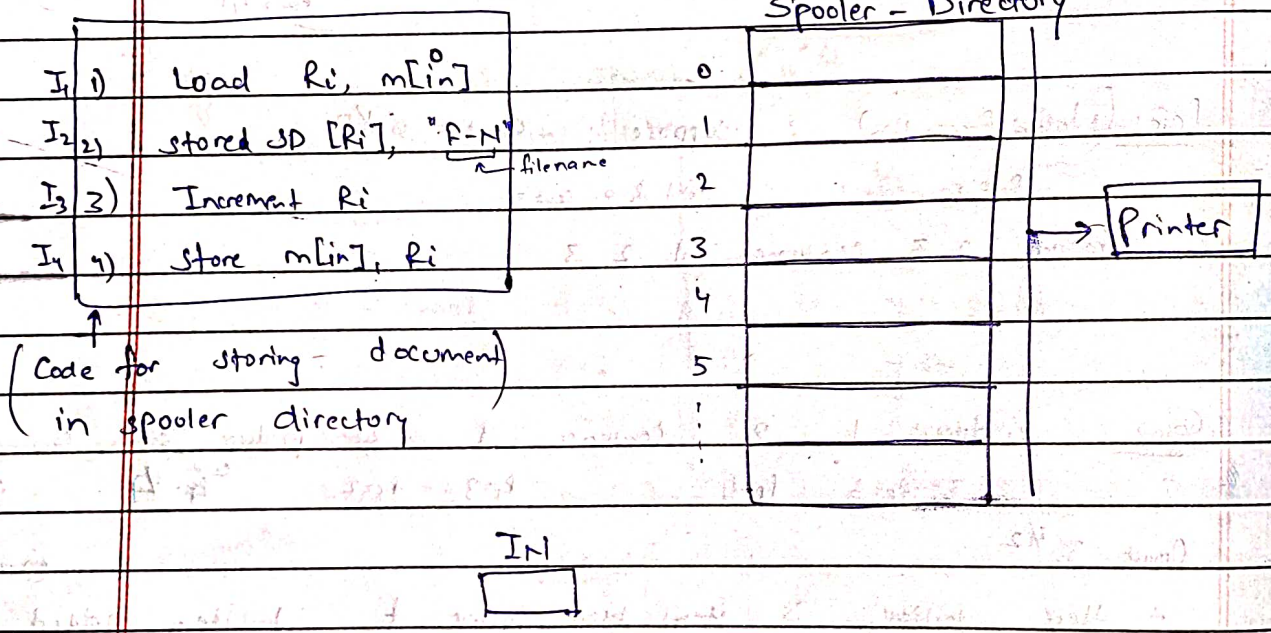
Printer - Spooler Problem

(Program)

In this problem we are having a network where many users are working & only 1 printer is present & these users want are printing their documents.

As we know printer is peripheral device & is very slow compared to CPU., so we maintain a program called Spooler

→ Spooler contains a queue where the ordered sequence of documents is present which has to be printed.



- Here "IN" is a shared variable which tells where next document has to be stored in spooler Directory (SD)

1) Normal Case : in = 0

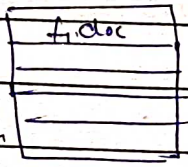
f-N → f_1.doc → S.D

i) R_1 = m[0] ⇒ R_1 = 0

2) SD[R_i] = f1doc

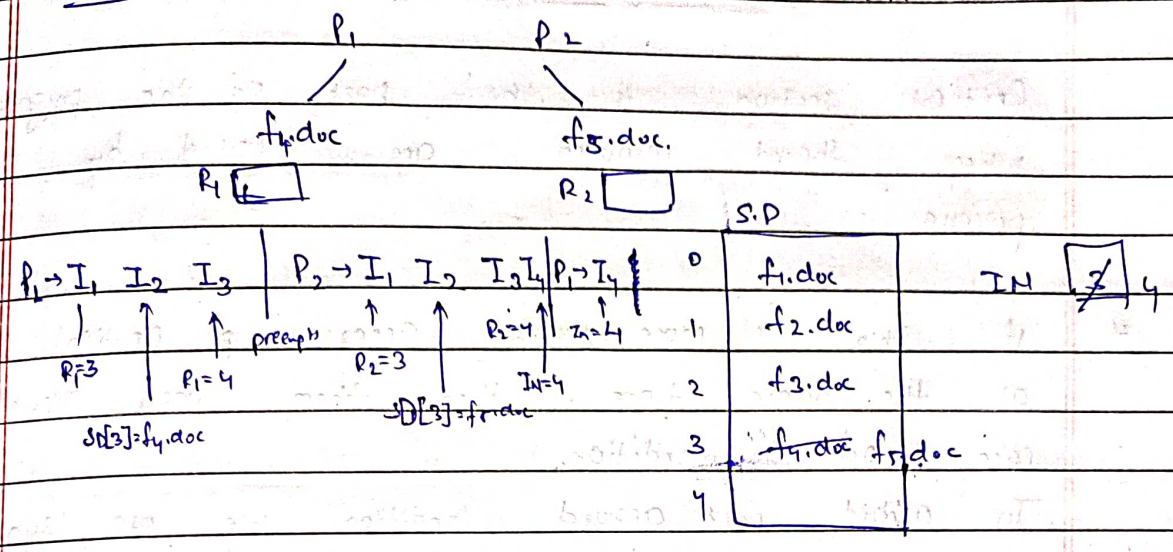
3) R1 = 1

4) in = 1



After this f_1.doc will be given to printer & it will be print

Case 2 :→ Two - process. Came at same time



- Due to Non - Synchronization both process stored data at same memory location & due to this data loss occurs.

Critical Section

Critical section is the part of the program where shared resources are accessed by various processes. (cooperating)

* If two or more process access a critical section at the same time, then there may occur race around condition.

To avoid race around condition we use synchronization.

```
main()
{
    Non Critical Section
    Entry section
    Critical Section
    Exit section
}
```

Synchronization codes

for achieving Synchronization Mechanism we are required to follow 4 conditions/Rules.

1) Mutual Exclusion
2) Progress
3) Bounded Waiting
4) No assumption related to H/w, speed

} Primary Conditions & Mandatory

} Secondary Condition

1) Mutual Exclusion :- if One process is using critical section then at same time another process must not be able to use critical section.

2) Progress :- If Critical section is not used by any process & one process does not want to use critical section, then it should not block/stop another process from entering into critical section.

3) Bounded Waiting :- If P_1 & P_2 required to use the CS but only P_i is using it, then there is no bounded waiting.

→ There should

A bound must exist on the no of times that Other processes are allowed to enter their CS after a process has made a request to enter its Critical section and before it is granted.

4) No assumption related to H/w & speed: The synchronization must not depend on Hardware & it should be portable (can work on diff. OS)

Lock Variable

Critical section solution using lock

* Execute in User Mode.
* Multiprocess solution
* No mutual exclusion guaranted

Lock = 0 → Vacant
1 → occupied.

Basic idea :→
```
do{
    aquire Lock
    CS
    Release lock
}
```

Pseudo Code :-

```
1. while(Lock==1) ;
2. Lock = 1 ;
3. Critical section
4. Lock = 0
```

Case 1 :→ $P_1 \to 1\ 2\ 3$ | $P_2 \to 1$ | $P_1 \to 4$ | $P_2 \to 1\ 2\ 3\ 4$

(P_1 preempts; Now P_1 comes back; P_2 terminated)

[P_1] in CS, [1] Lock — Infinite loop bcz lock=1

Case 2 :→ P_1 P_2 / Lock=1 Lock = 0 1

P_1: 1 (executed) ✱ | P_2: 1 2

P_1 preempts

P_2 starts execution

P_1 comes back, it will not see the Lock=1 & directly enters in CS bcz it has already executed that instruction

[P_2 P_1] in CS

Now in CS 2 process are present which fails Mutual exclusion.

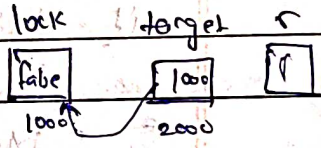
vibhu
PAGE No
DATE

Test And Set Instruction

In lock variable ~~while execut~~ the problem occurs, when a process executes statement 1 and then it preempts and another process enters into critical section, after some time process P_1 returns and it also enters critical section & mutual exclusion fails.

- Now the Test and set instruction solved this problem by combining the statement 1 & 2 and making it atomic.

```
while ( test_and_set (& lock));
CS
lock = false;
```




```
boolean test_and_set ( boolean *target)
{   boolean r = *target;
    *target = true;
    return r;
}
```

* This method provides mutual exclusion & progress.

Turn Variable (strict Alteration)

* It is a process solution
* Run in user mode.

Process P_0	Process P_1
1) Entry code → while(turn != 0);	1) while (turn != 1);
2) Critical section.	2) Critical section
3) Exit code → turn = 1;	3) turn = 0;

int turn = 0

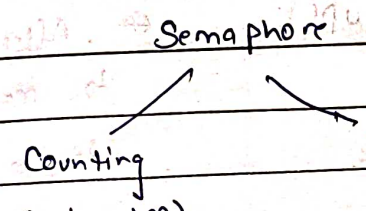
If initially turn = 0 the process "P_0" will run first
if initially turn = 1 the process "P_1" will run first

- Mutual Exclusion is present
- Progress is not guarenteed.
  bcz suppose turn = 0 and Critical section is empty then at that time both process can execute suppose 'P_1' tries to execute first but turn = 0 then it went into infinite loop in statement 1 (i.e. while (turn != 1);)
  therefore the progress is not guaranteed
- Bounded waiting present
- Independent of software & hard ware

Semaphore

- It is a method or tool which is used to prevent race conditions.

Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve synchronization.



During synchronization we need to perform various operation (these operation are ~~are~~ performed in entry section and exit section)

These operation are -

1) ~~Down~~ P() / Down / Wait / — Entry section
2) V() / Up / Signal / Post / Release

Initially when CS is empty and multiple processes wants to execute CS then it needs to execute the entry section code (i.e. Down)

```
Down ( Semaphore S)
{ SValue = S Value - 1;
  if (S Value < 0)
  {
    put process (PCB) in
    Suspended list / sleep();
  }
  else
    return;
}
```

Entry section

Suppose initially S = 3 & we have process P_1 P_2 P_3 P_4

CS	Blocked	S
P_1 P_2 P_3	P_4	~~0~~ ~~-1~~ -1

∴ S = -1 indicates that there is 1 blocked out process

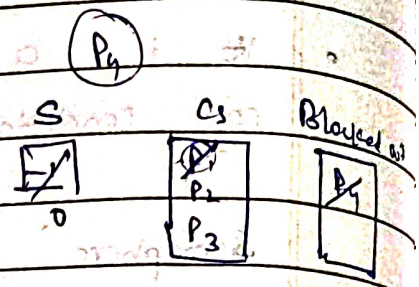
S = 1 indicates 1 more process can enter CS.

While exiting the critical section the process has to execute the exit section code.

Exit Section

```
Up (Semaphore S)
{
    SValue = SValue + 1;
    if (SValue <= 0)
    {
        Select a process
        from suspended list
        wake up();
    }
}
```

← After waking up it goes to ready queue & again try to enter in CS.



if $S > 0$ & then we perform Down operation then it will be an successful operation.

Q) If $S = 10$ & we perform 8 P operation & 4 V operation then what is the final value of Semaphore

→ $S = 10$

After 8P ∴ $S = 4$

After 4V ∴ $S = 4 + 4 = 8$

∴ $S = 8$

Uniprocessing → 1 CPU

- Uniprocessing
- Multi programming (Pipelining & Chaining)
- Multi tasking / Time Sharing

1 CPU — FCFS, SJF, RR, PrioScheduling

Multiprocessing → 2 or more CPU (Duplication of Hardware)

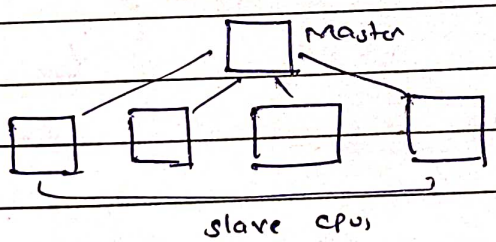
→ How to distribute CPU properly among process?

- Using "Load Balancing"

Multiprocessing

- Symmetric Multi processing
- Asymmetric Multiprocessing (AMP) → Here 1 CPU is master and other CPUs are its slaves

, work of masterCPU → proper distribution of process among slaves



Also known as → Client server architecture

Advantage :- • Efficient utilization of CPU

Disadvantage :- If master got destroyed then whole architecture got idle

Solution → • Replication of server
• Symmetric multi processing

Migration :-

- Push
- Pull
  - ↳ If some CPU is free or having less load then it goes to other CPU which CPU having more load then it takes some process from it.
- ↳ If CPU having more/high load then it pushes some processes to other CPU with min load / less load

* In practical we use hybrid migration combination of both push and pull.

Binary Semaphore

It can have two values either 0 or 1.

Two operations can be performed.

1) Down, P, wait
2) Up, V, Signal.

Down Operation

```
Down (Semaphore S)
{   if (sValue == 1)
    {
        sValue = 0;
    }
    else
    {
        Block the process
        and place in
        suspend list,
        sleep();
    }
}
```

- if initially S = 1 & we perform down operation then it is successful operation
- If initially S = 0 & we perform down operation then it is unsuccessful operation and process will be block.

Up Operation

```
Up (Semaphore S)
{   if (suspend list is Empty)
    {
        sValue = 1;
    }
    else
    {
        select a process from
        suspended list and
        wake up();
    }
}
```

- If initially S = 0 & suspend list is empty then S = 1 & process using CS will move out of CS successfully.

Ex:-

P_1	P_2
Down(s)	Down(s)
CS	CS
Up(s)	Up(s)

P_1 & P_2 are cooperative process & using same CS.

* If initially $s = 0$ then P_1 & P_2 will be blocked so we have to take s value carefully.

Suppose $s = 1$ & P_2 executes. (at same time) P_1 (one)

s | 1 | 0

(Down) $P_2 \rightarrow$ CS | $P_1 \xrightarrow{\text{Down}}$ suspended | $P_2 \Rightarrow$ CS $\xrightarrow{\text{UP}}$ executed & wakeup P_1

Q) Each Process P_i ($i = 1$ to 9) execute the following code

```
repeat
{
    P(mutex)
    CS
    V(mutex)
forever
```

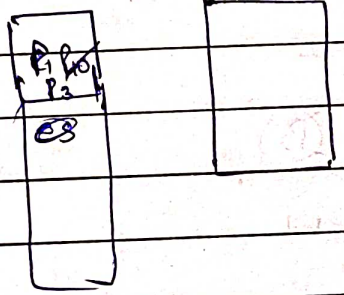
Process P_{10} execute the following code.

```
repeat
    V(mutex)
    CS
    V(mutex)
forever
```

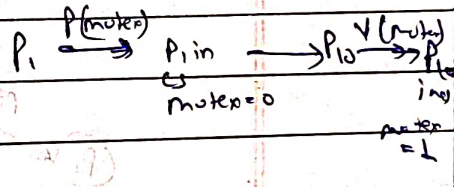
mutex
1

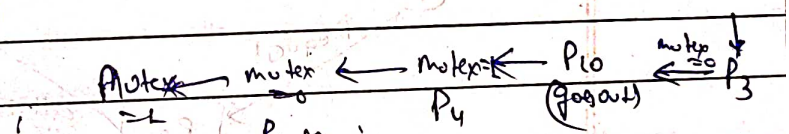
- What is the maximum no. of process that may present in CS at any point of time?

P_1 P_{10} suspended.



mutex
0 0 0





Similarly the all the process can enter CS therefore max. no. of process that may enter CS = 10.