Process Synchronization

Co-operative process	Independent process
→ One's execution affects other as they share ~~same~~ same or common variable, memory / buffer, code, resources	One's execution doesn't affect other as they have nothing in common

int shared = 5 → 6 → 4

P_1	P_2
x = shared (∵ x = 5)	y = shared (∵ y = 5)
x++ ; x = 5 → 6	y − − ; y = 5 → 4.
Sleep (1)	CS Sleep (1)
Shared = x (∵ Shared = 6)	Shared = y (∵ Shared = 4)
(Terminated)	

↓ gets saved in PCB (Process control block)

As these processes aren't synchronized, they end up producing either 4 or 6, but it should give 5
This is called race-condition

Critical Section

"It is the part of program where shared resources are accessed by various ~~programs~~ processes".

P_1	P_2
#include	#include
main(){	main(){
A, B → non-critical section	X, Y → Uncommon code
count → critical section	count + Common code

main()

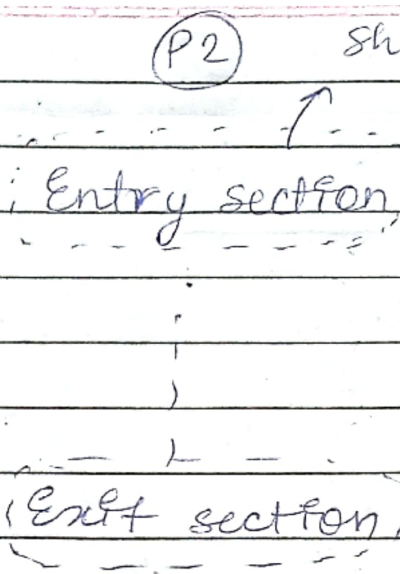
A, B
Non-critical section
(P_1)

Should clear entry section ← Entry section

count ++
critical section

↑ P_1

Exit status



4 conditions for process synchronization:

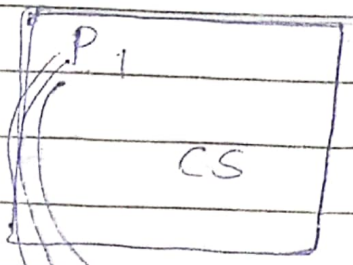
1) Mutual Exclusion } Primary
2) Progress

3) Bounded wait
4) No assumption related to Hardware speed } Secondary

1) Mutual Exclusion: If one process is in critical section, then other process must not be allowed to enter CS.

2) Progress: When both can use common code there is progress
E-g. P_1 can't enter cs due to some code written in P_2 then progress doesn't happen.

3) Bound wait: There musn't be a condition of starvation as one process uses the CPU every time. Let every process use the CPU.



If P_1 only uses the CS then it would go in ~~bounded wait~~ ∞ loop.

	P_1	P_2
→	1	0
→	2	0
	3	0
	⋮	
	∞	

4) Hardware speed: Every solution should be portable & universal. There shouldn't be anything dependent on H/W speed.

Critical section using lock

do {
acquire lock
CS
release lock
}

Entry code

```
1. while (lock == 1)
2. lock = 1
3. Critical section
4. LOCK = 0
```

Exit code

* Execute in user mode
* Multiple process solution
* No mutual exclusion guarantee

Case 1: P1 | P2 ✗
I1 I2 I3 I4 | ✓ lock = 0 ✗ 0

lock = 0 → CS is vacant
lock = 1 → CS is full.

→ Let's assume P1 comes first, so lock is 0 initially & it turns to 1 after 2nd instr. & P1 enters CS
→ So now if P2 comes, it would be stuck on the ∞ while loop.
→ When P1 completes its work it sets lock to 0 again & P2 can enter CS now

Case 2: Let consider P_1 comes first:

L = 0 X 1	P_1	P_2
	$I_1 * I_2 I_3$	$I_1 I_2 I_3$

→ P_1 executes the 1st instr. & gets pre-empted
→ Now the problem is that lock isn't set to 1, so P_2 can enter CS
→ So now P_2 comes & executes I_1 & I_2 & I_3 & enters CS
→ P_2 gets pre-empted
→ Now P_1 comes back & over-writes lock to 1 & enters CS,
→ So, now the real problem arises as both P_1 & P_2 are in CS.

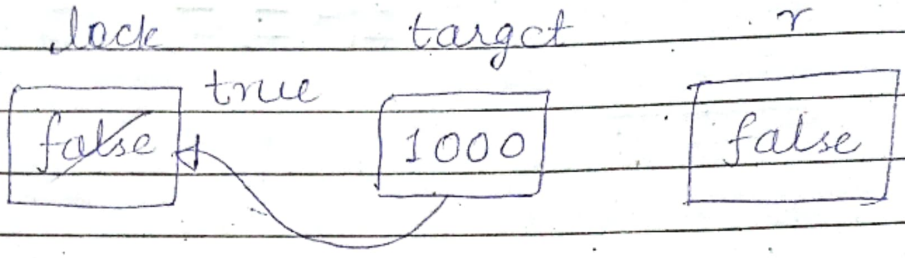
Test & Set Instruction:

It combines I_1 & I_2 of lock

```
while (test and set(&lock));
  CS
  lock=false
```

```
boolean test and set (boolean * target) {
      boolean r = * target;
      * target = TRUE;
      return r;
}
```

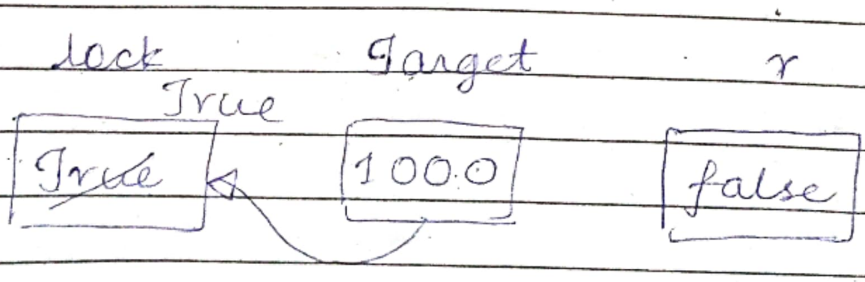
The value of lock is false initially



→ In the while loop the call goes to the actual func^n.
→ & value stored at address of lock is passed to the func^n
→ We store value of target i.e. false in r
→ We set target to True
→ So here we have tested that lock is false, so no process is there in CS & we also set lock to True no other process can enter
→ We return r & as it's false, P_1 enters the CS.

→

→ So now if P_2 wants to enter:



→ So, when the func^n is called, true is passed.

→ r is set to true
→ Lock is set to true
→ ~~r is~~ r is returned which is true, so P_2 gets stuck in a loop

P_1	P_2
$I_1 * I_2$	I_1

Mutual exclusion as well as progress achieved.

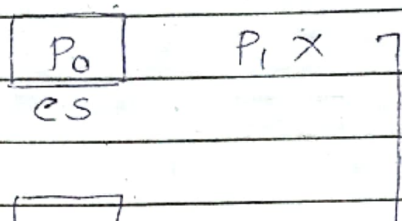
Turn Variable

1) 2 process solution 2) Run in user mode

Process "P_0"	Process "P_1"
~~while~~	
while (turn != 0);	while (turn != 1)
CS	CS
turn = 1;	turn = 0;

→ Turn can take 2 values 0 & 1
if turn = 0, P_0 runs first
if -11- = 1, P_1 runs first
→ While exiting a process turn value is changed so that other processes can enter it.

turn = 0 / 1



1) Mutual Exclusion is satisfied

2) Progress is not there

(if the CS is empty & the process wants to enter then other is stopping it, the other saying I will go first, when the other enters & exits, so the value of turn gets changed & the one which wants to enter can enter.

3) Bounded wait is not there as processes enter one after the other alternatively.

4) Not Hardware Dependent

Semaphore:

(It is a tool to prevent race condition)

Semaphore:
- Counting (-∞ to ∞)
- Binary (0, 1)
- Running co-operative processes side by side

Semaphore is an integer variable which is used in exclusive manner by various concurrent co-operative processes in order to achieve synchronization.

Entry Section	Exit Section
Down (Semaphore S) {	Up (Semaphore S) {
S.value = S.value - 1	S.value = S.value + 1
if (S.value < 0) {	if (S.value ≤ 0) {
put process (PCB) in suspend list sleep();	select a process from suspended list wake up();
}	}
else {	}
return;	
}	
P(), Down(), Wait()	V(), Up(), Signal, Post, Release

Suppose P_1, P_2, P_3, P_4, P_5 are in PCB.

& value of S = 3

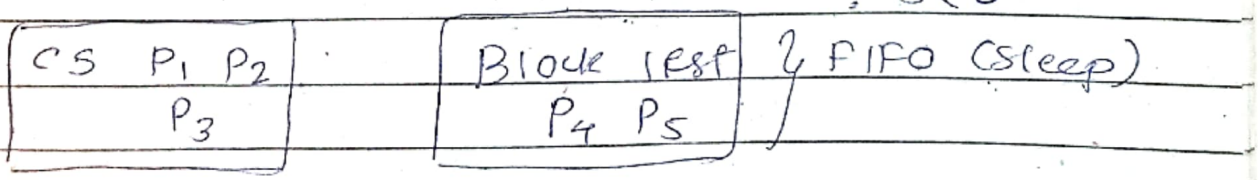
∴ For P_1, S = 3-1 = 2 Enters CS

P_2, S = 2-1 = 1 " "

P_3, S = 1-1 = 0 " "

For P_4 S = 0-1 = -1 No Entry in CS

∴ S < 0



As S value is -2, 2 process are in blocked state

S

-2

If P_1 wants to exit CS then it inc. the value of S from -2 to -1 & P_4 wakes up (It means that the process enters ready queue but hasn't yet woke up)

Again same process happens when P_2 exits & P_5 comes in ready queue

Still no proc. can enter CS as S=0

When P_3 exits, P_4 can enter CS

Up

S

0

No process in suspend list

S

10

How many process can come in CS.

For e.g. S=10, & 6 P() & 4 V() operations

$\therefore S = 10 - 6 = 4$ 6 P()

$S = 4 + 4 = 8$ 4 V()

Page No.:
Date. / /

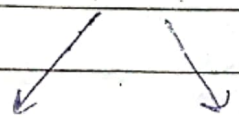
S 5P 3V 1P

17

= 17 - 5 + 3 - 1

= 14

Binary Semaphore:



Down (Semaphore S)	Up (Semaphore S)
{	{
if (S.value == 1) {	if (suspend list is empty) {
S.value = 0;	S.value = 1;
}	}
else {	else {
Block this process & place on suspend list sleep()	Select a process from suspend list & wake up ();
}	}
}	}

S = 1 } Down() S = 0 } Successfull operation

S = 0 } sleep (); block } Unsuccessfull operation

S = 0
Up

S = 1

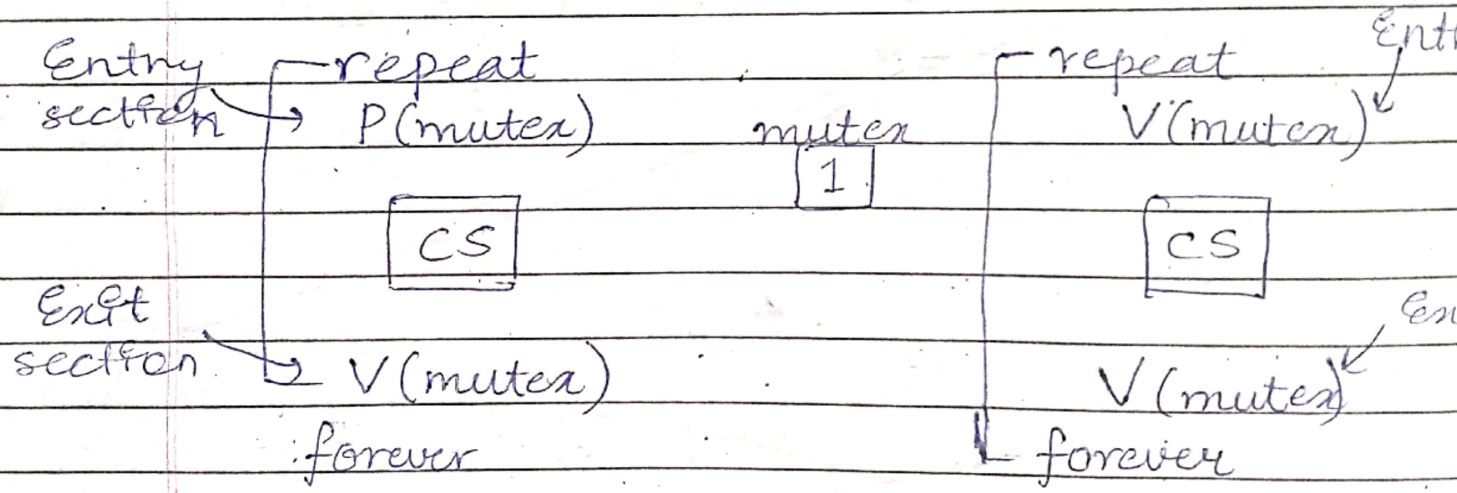
S = 1
Up

S = 1

In up we check whether to block queue is empty or not
If empty (S = 0 1)
If not empty → we bring process from block state to unblock state

Each process P_i (i = 1 to n) execute the foll. code

Process P_{ID} execute the foll. code



What is the max^m no. of process that may be present in CS at any part of time?

Continued:
Soln:

Case-1 P_1 comes mutex turns from 1 to 0

mutex 1 0 X 0 X 0 1

So now not other process can enter except P_{10}

⇒

CS | P_1 P_{10} P_2 P_3 P_{10}

→ P_{10} comes & mutex changes its value from 0 to 1
→ Now any other process can enter for e.g. P_2 enters & mutex goes back to 0
→ Now P_{10} leaves & mutex again goes to 1
→ Now P_3 enters & mutex goes to 0
→ Now P_{10} -11 ——— 11 ——— 1
→ In this way due to the enter & exit of P_{10} at maximum all processes can enter CS.

∴ Max. processes = 10