Lecture 16

Next Topic:
Synchronization

- "The *too much milk* problem"

time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!

Question:

There are two processor named producer and consumer.

And their code are also given below.

Code for Producer

```
counter++;
```

Code for Consumer

```
counter--;
```

Analyze what can happen If we run these two processes concurrently.

`counter++;`

$register_1$ = `counter`
$register_1 = register_1 + 1$
`counter` = $register_1$

`counter--;`

$register_2$ = `counter`
$register_2 = register_2 - 1$
`counter` = $register_2$

`counter++;`

$register_1$ = `counter`
$register_1 = register_1 + 1$
`counter` = $register_1$

`counter--;`

$register_2$ = `counter`
$register_2 = register_2 - 1$
`counter` = $register_2$

T_0:	*producer*	execute	$register_1$ = `counter`	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2$ = `counter`	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	`counter` = $register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	`counter` = $register_2$	$\{counter = 4\}$

Operation	counter = counter+1;	counter = counter+1;
on CPU	reg_1 = counter; reg_1 = reg_1 + 1; counter = reg_1;	reg_2 = counter; reg_2 = reg_2 + 1; counter = reg_2;
Interleaved execution	reg_1 = counter; reg_1 = reg_1 + 1;	
		reg_2 = counter; reg_2 = reg_2 + 1;
	counter = reg_1;	
		counter = reg_2;

`counter++;`

$register_1$ = `counter`
$register_1 = register_1 + 1$
`counter` = $register_1$

`counter--;`

$register_2$ = `counter`
$register_2 = register_2 - 1$
`counter` = $register_2$

This is called Race Condition

T_0:	*producer*	execute	$register_1$ = `counter`	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2$ = `counter`	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	`counter` = $register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	`counter` = $register_2$	$\{counter = 4\}$

Race Condition:

A situation where several processes access and manipulate the same data concurrently and the outcome of execution depends on the particular order in which the access takes place, is called a Race Condition.

The Critical-Section Problem

- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section.


ENGAGED

The Critical-Section Problem

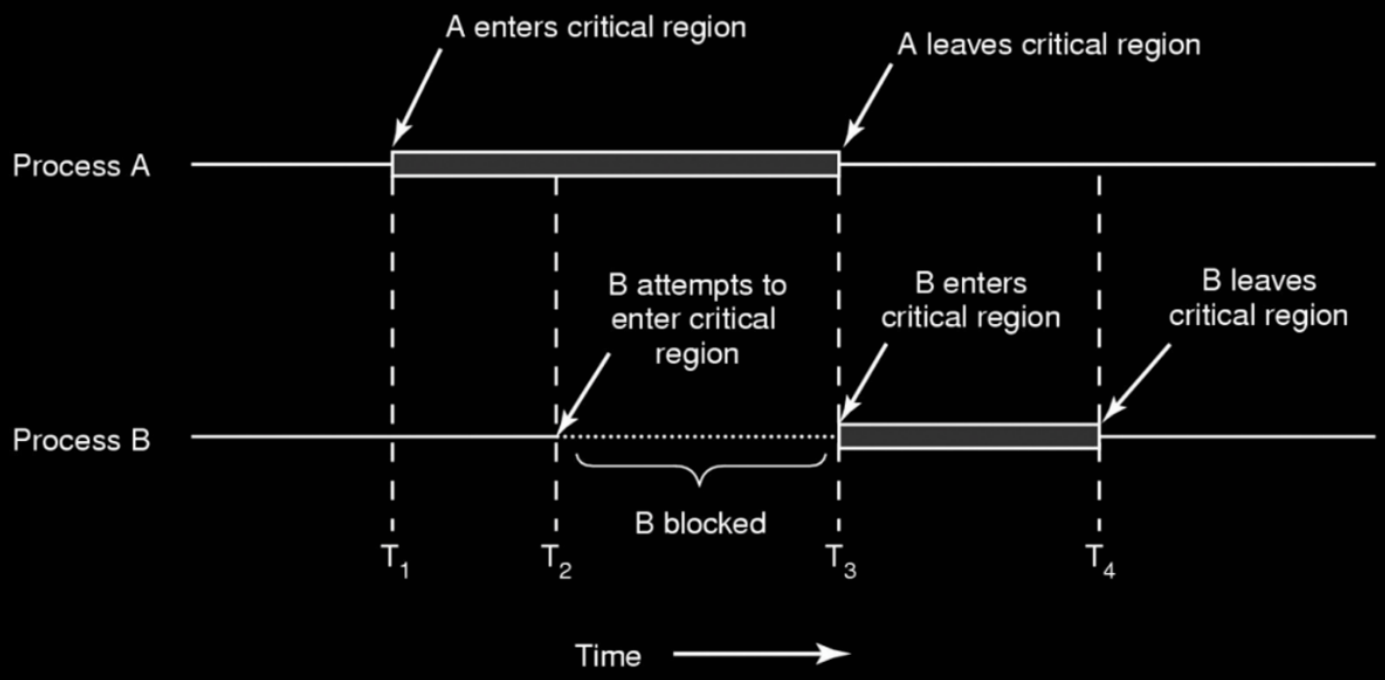
Informally, a critical section is a code segment that accesses shared variables and has to be executed as an atomic action.

The critical section problem refers to the problem of how to ensure that at most one process is executing its critical section at a given time.

Important: Critical sections in different threads are not necessarily the same code segment!

General structure of process P_i

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```



This is what we actually want.

Question

What is the final value of count?

```
int count = 0;
```

min :

max :

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```

Question

What is the final value of count?

```
int count = 0;
```

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```

min : 2

max : 20

when we are not loosing any update (serial execution)

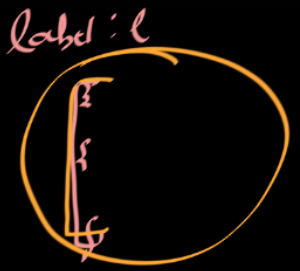
count=0

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```



idea → we can just lose all of the T2 updates

```
load  R1, 1000
Add   R1, 1
Store R1, 1000
```

1) T_1 is doing just one instruction
2) T2 start and finish completely
3) T1 resume and finish

⇒ final count = 10

count=0

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

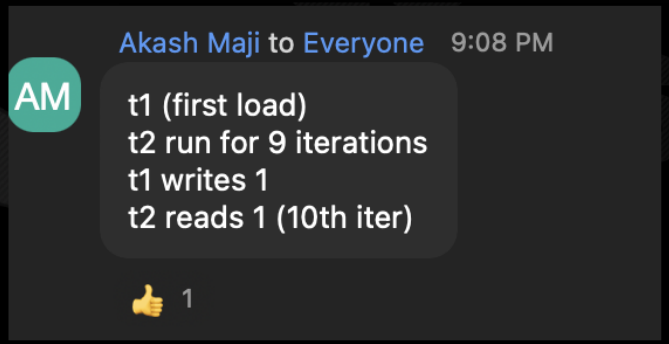
Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```

load R1, 1000

Add R1, 1

Store R1, 1000



Read from memory

modify locally

write to memory

GO CLASSES

count=0

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```

AM **Akash Maji** to Everyone 9:08 PM

t1 (first load)
t2 run for 9 iterations
t1 writes 1
t2 reads 1 (10th iter)

👍 1

a **anish** to Everyone 9:12 PM

T1 -> First Instruction
T2 -> All instructions 9 times
T1 -> Second and Third Instruction
T2 -> First Instruction
T1 -> All Instructions 9 times
T2 -> Second and Third Instruction

👍 1

T_1 reads 0

T_2 finish 9 instructions

T_1 modify and write to memory

T_2 will start the 10th iteration
- read 1
- Add

T1 finish completely

T2 will write 2.

GO CLASSES

- we are asking T_1 to read minimum possible value (T_1 will read 0)
- T_1 Preempt, T_2 scheduled (finish all except last iteration)
- T_1 to write (here T_1 has finished one iteration)
- T_2 to just read "1" (only read)
- T_1 scheduled and finish
- T_2 (last iteration) will write 2.

The Critical-Section Solution

GO CLASSES

Soln :

Mutual exclusion : we want at most one process in critical section (at a time).

GO CLASSES

Soln :

Mutual Exclusion : we want at most one process in critical section

Thread :

```
Enter CS;
        ← CS
Leave CS;
```

T_1

{ while (1) ;

CS: [Count++ ;]

T_2

{ while (1) ;

CS: [Count++ ;]

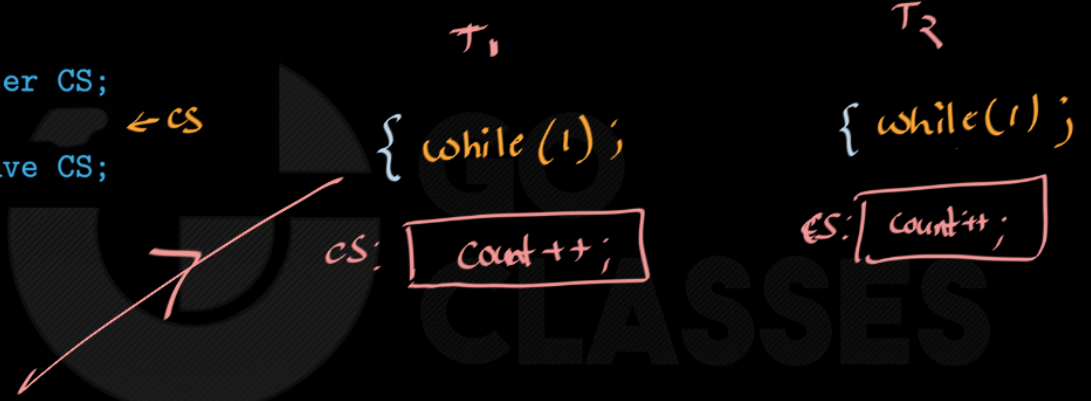
GO CLASSES

Soln :

Mutual exclusion : we want at most one process in critical section

Thread :

```
Enter CS;
        ← CS
Leave CS;
```



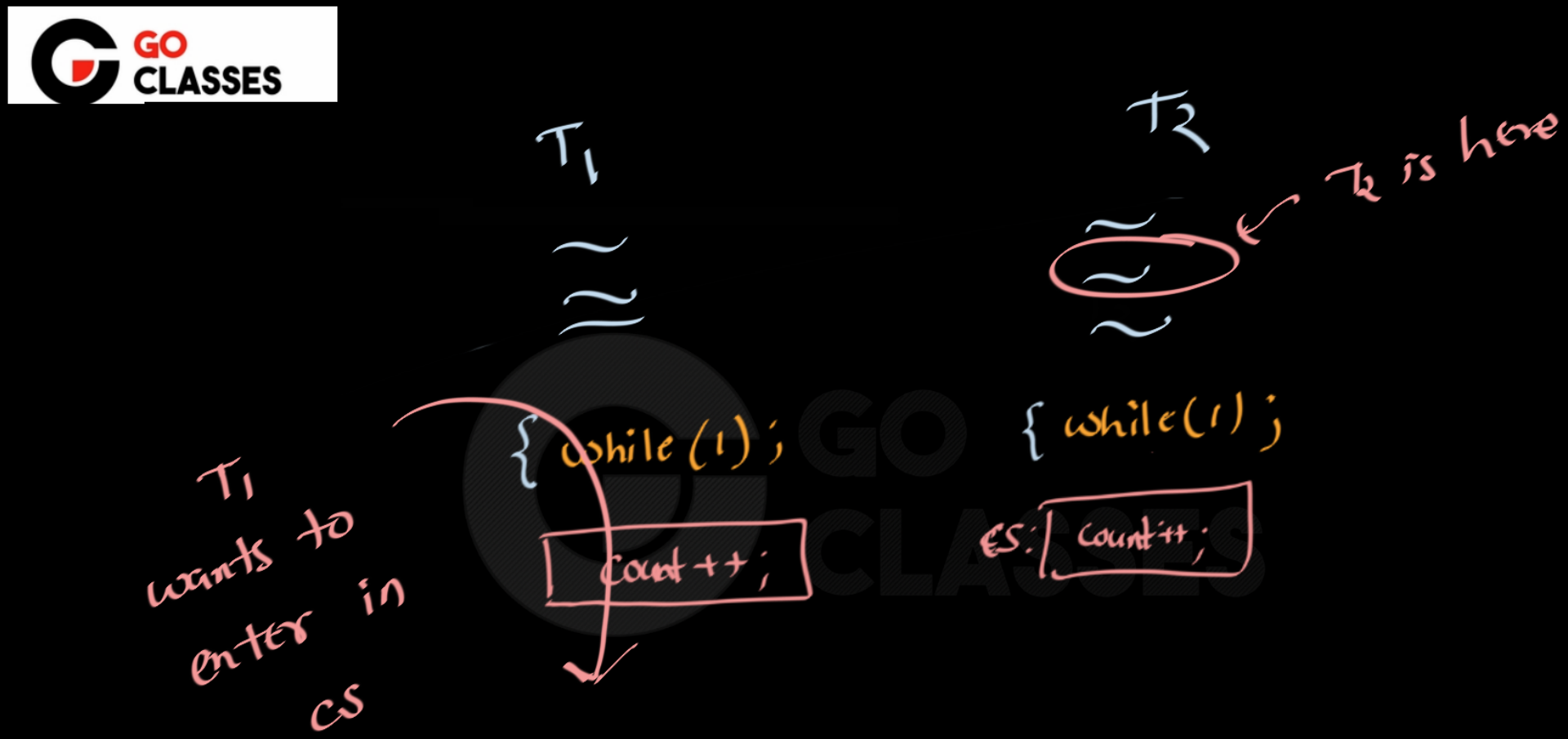
[this satisfy mutual exclusion but it is a horrible soln.]

GO CLASSES

Mutual exclusion : we want at most one process in critical section (at a time).

+

Some extra conditions to solve CS problem.


GO CLASSES
T1
T2
T2 is here
{ while (1) ;
count++;
{ while (1) ;
CS: count++;
T1 wants to enter in CS

GO CLASSES

Progress : there should be some Progress in critical section

Arturo Gangwar to Everyone 9:32 PM

S

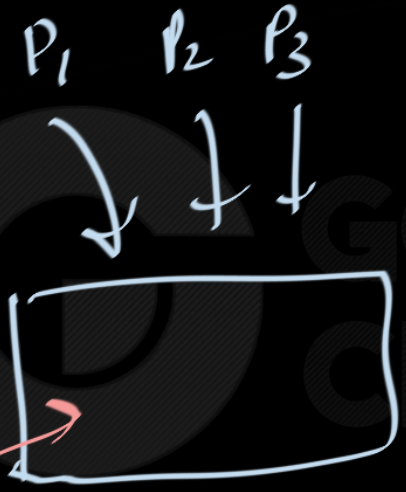
if cs is empty and if someone come which is eligible to enter then that should to allowed to enter in the cs

→ if P wants to enter in CS, and no one else wants to enter in cs then P should get enter.

→ if multiple Process wants to enter then AT LEAST one should get enter (in other words No deadlock)

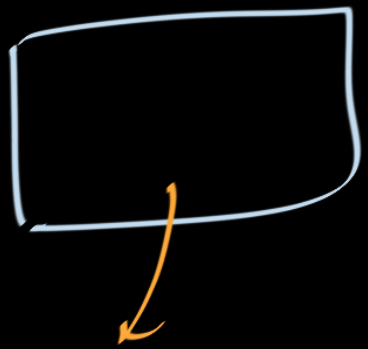
GO CLASSES

Progress



allow at least 1

only one process wants to enter



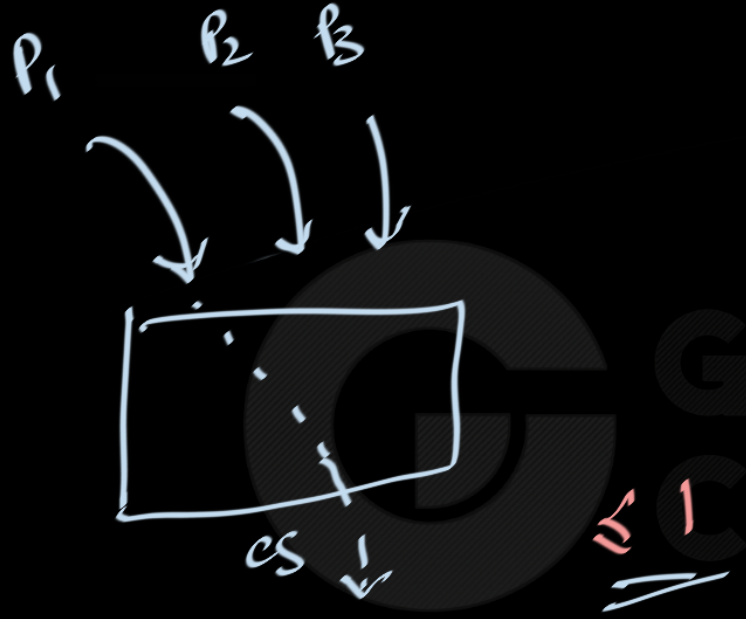
allow it

Arturo Gangwar to Everyone 9:57 PM

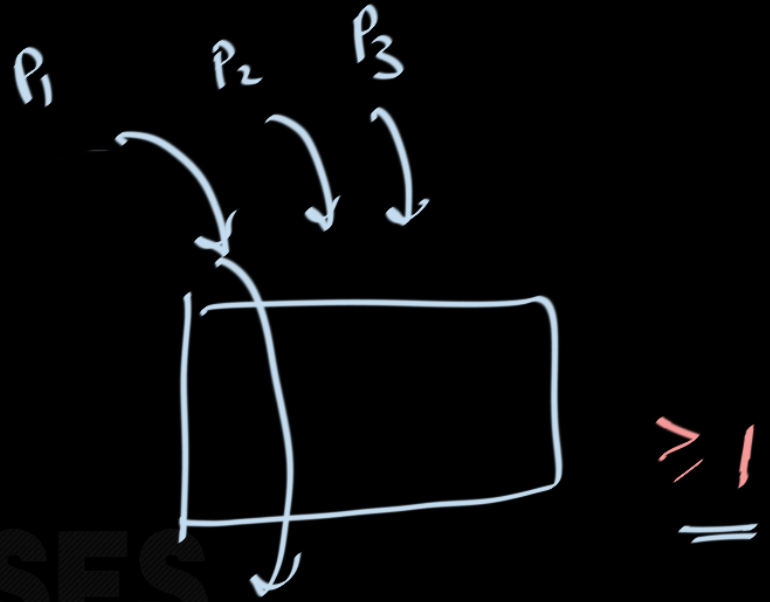
S

if P1 is alone wants to enter then P1 be like " mai nhi to kaun ?"

1 2 1



Mutual exclusion



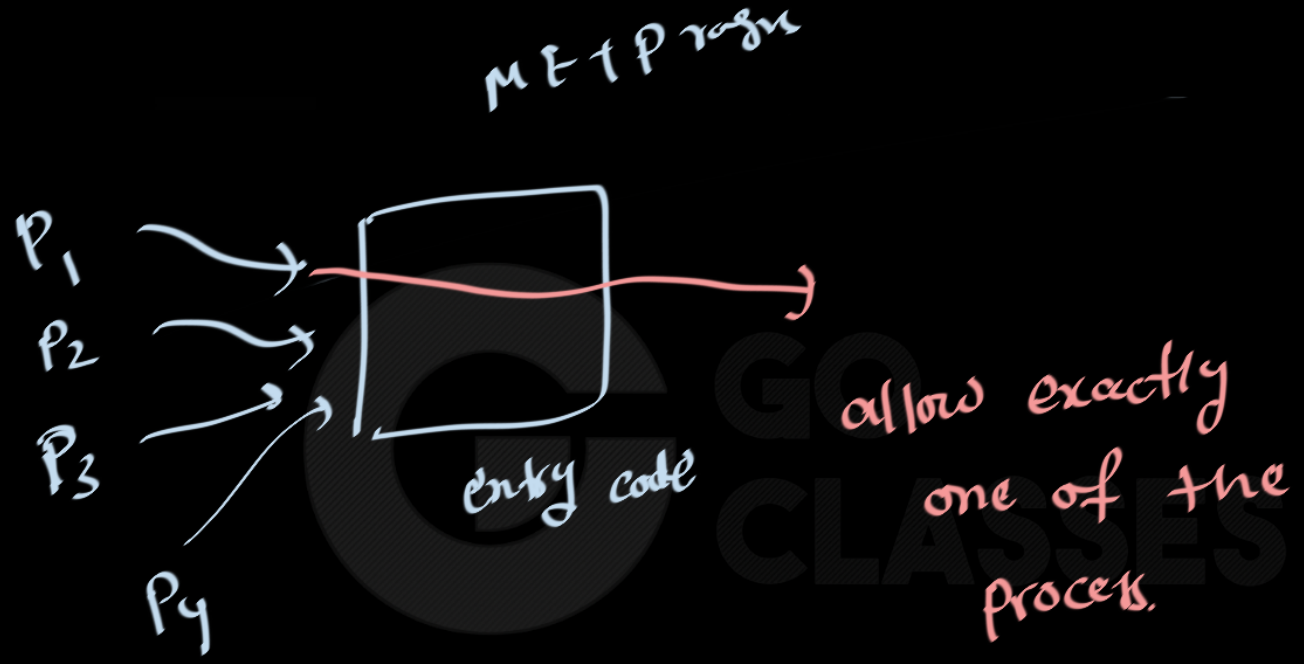
Progress

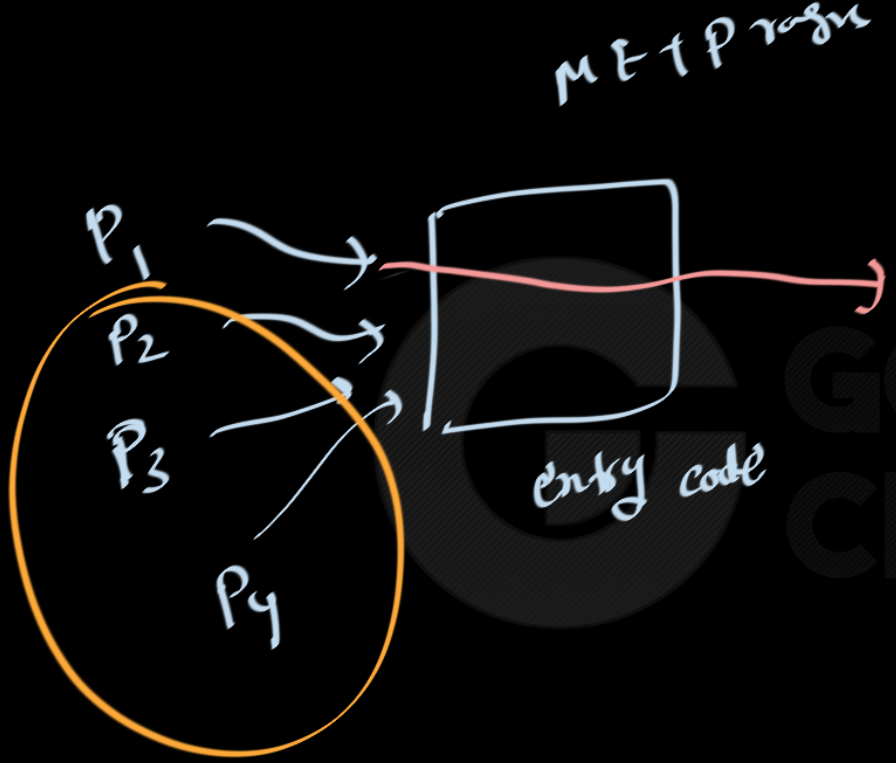
GO CLASSES

ME + Progress

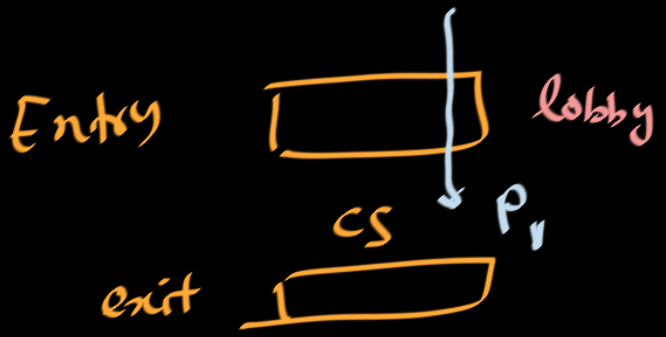
exactly one Process is allowed.

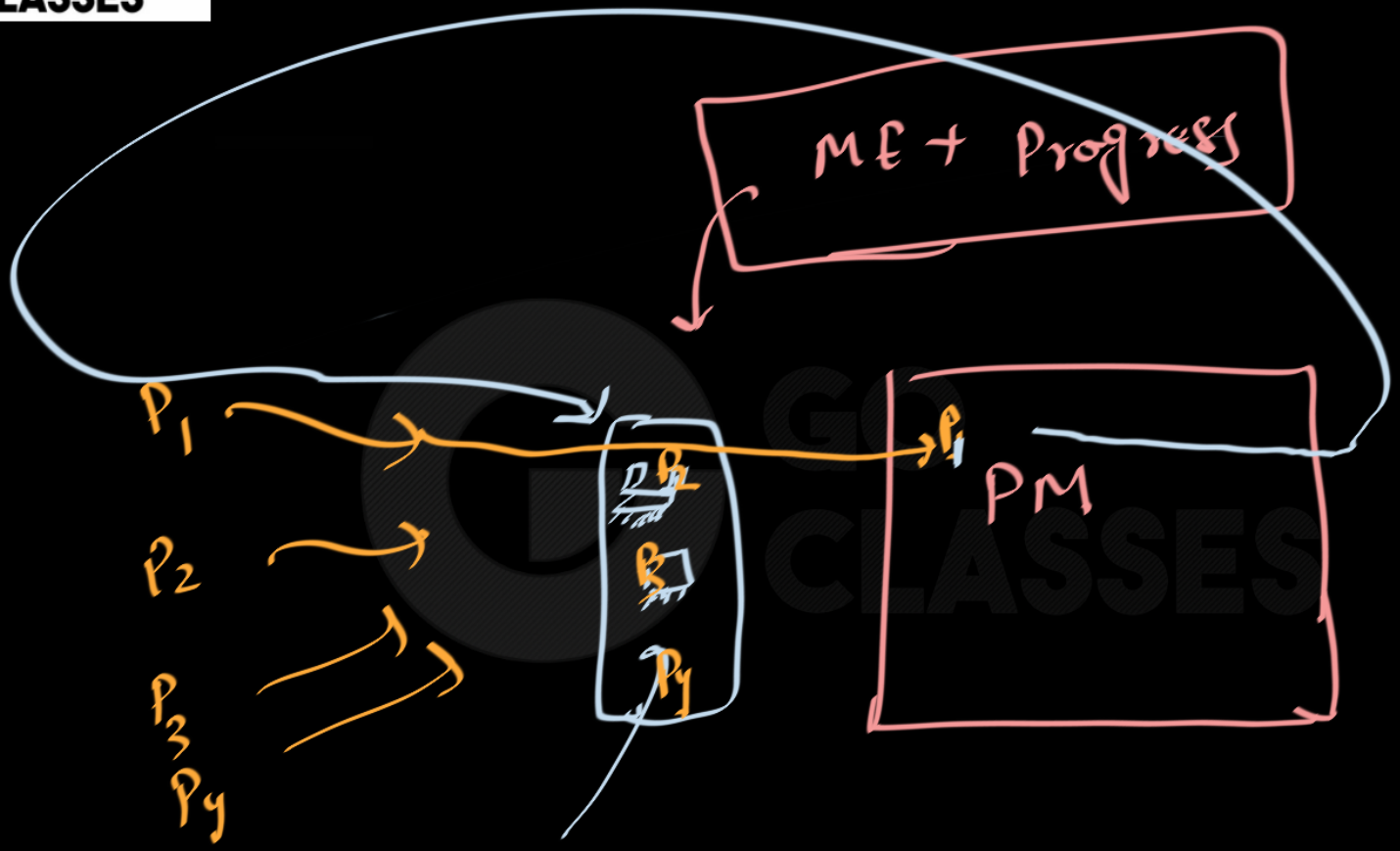
Do you think we may require the extra condition(s) along with ME + Progress


MEP region
P1
P2
P3
P4
entry code
allow exactly one of the process.



allow exactly one of the process.





When P_1 is meeting with PM, P_2, P_3, P_4 are waiting

GO CLASSES

i don't want P_i to re enter immediately to CS while others are waiting.

GO CLASSES

→ i don't want P_i to re enter immediately to CS while others are waiting.

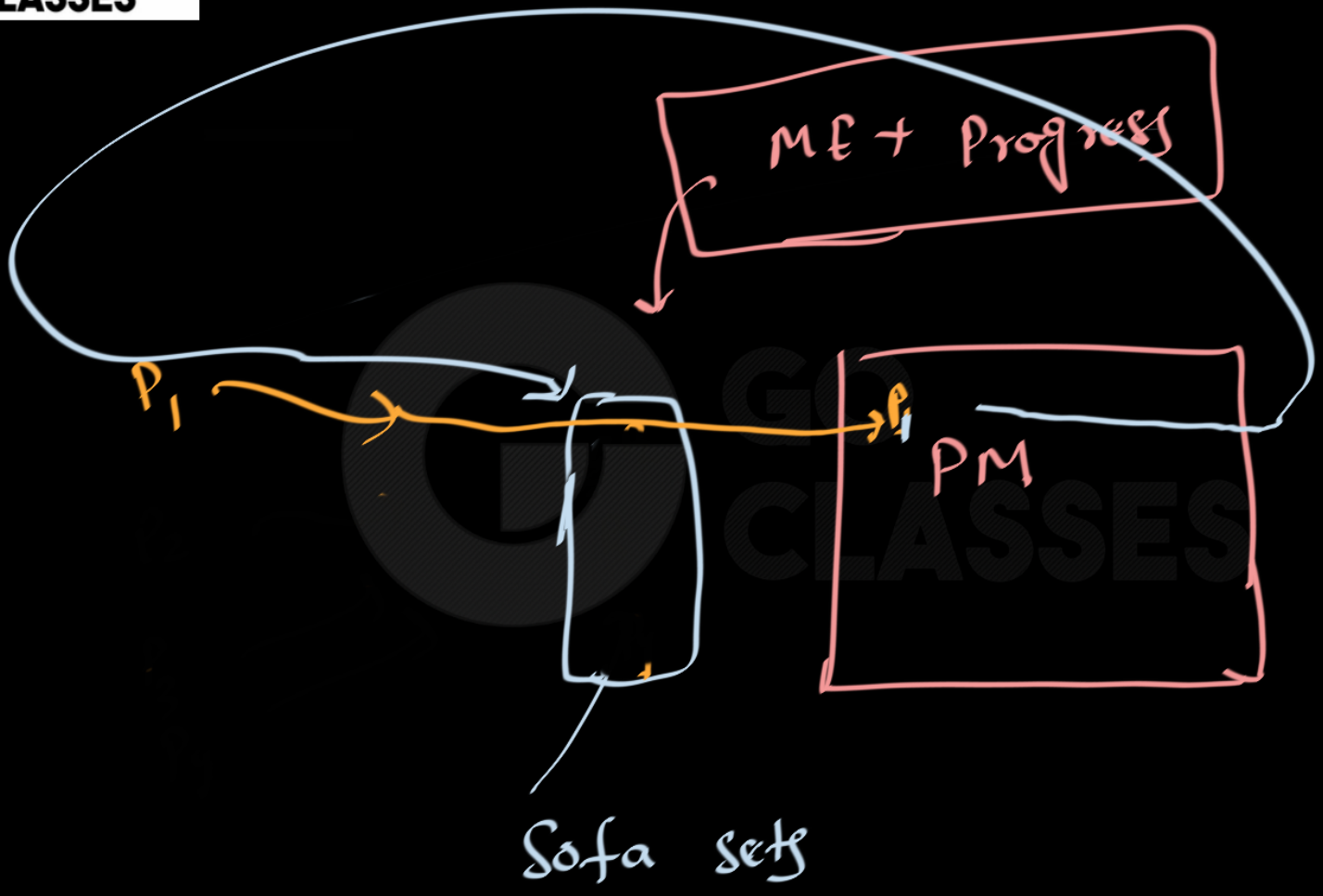
[→ if others are not waiting then i should allow to re enter immediately.]

GO CLASSES

- i don't want P_i to re enter immediately to CS while others are waiting.

- [→ if others are not waiting then i should allow to re enter immediately.]

→ this is already in the defination of Progress


ME + Progress
P1
PM
Sofa sets

GO CLASSES

ME : allow at most one. ≤ 1

+

Progress : allow at least one ≥ 1

Exactly one

GO CLASSES

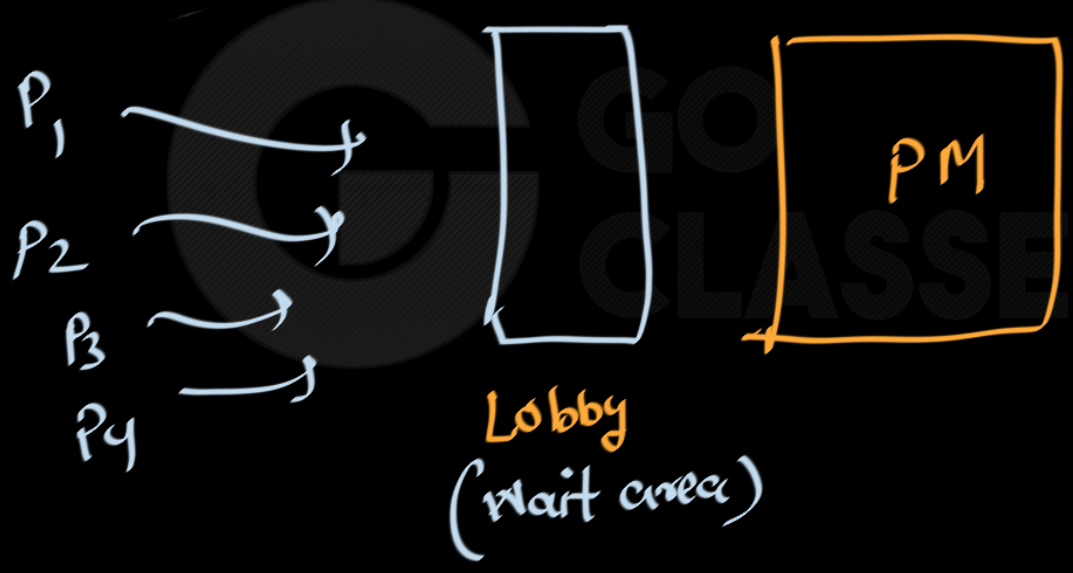
ME : allow at most one. ≤ 1

Progress : allow at least one ≥ 1

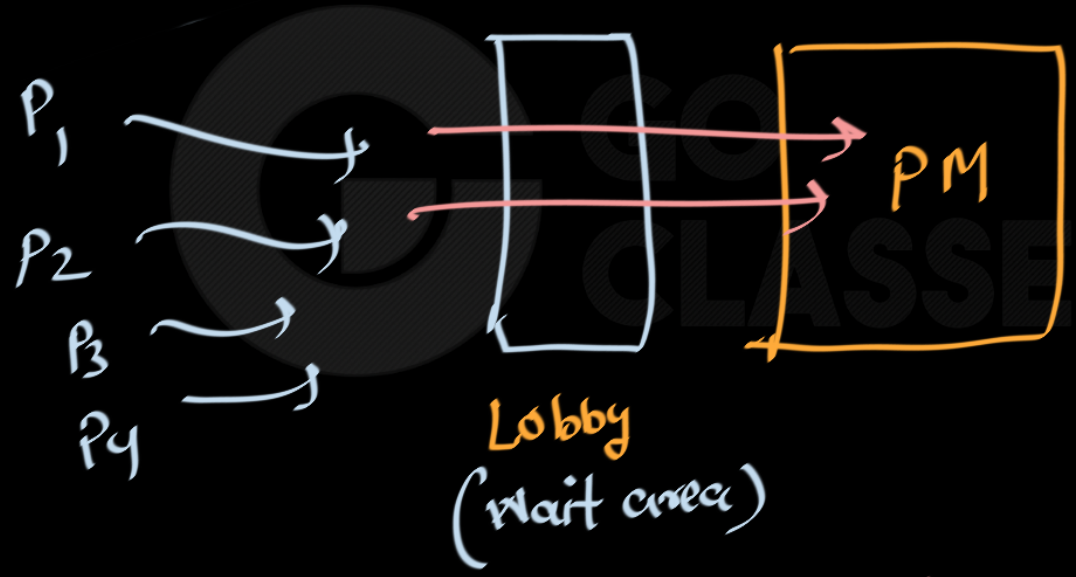
Bounded waiting : Don't allow re enter immediately if others are waiting

GO CLASSES

ME + Progress + BW

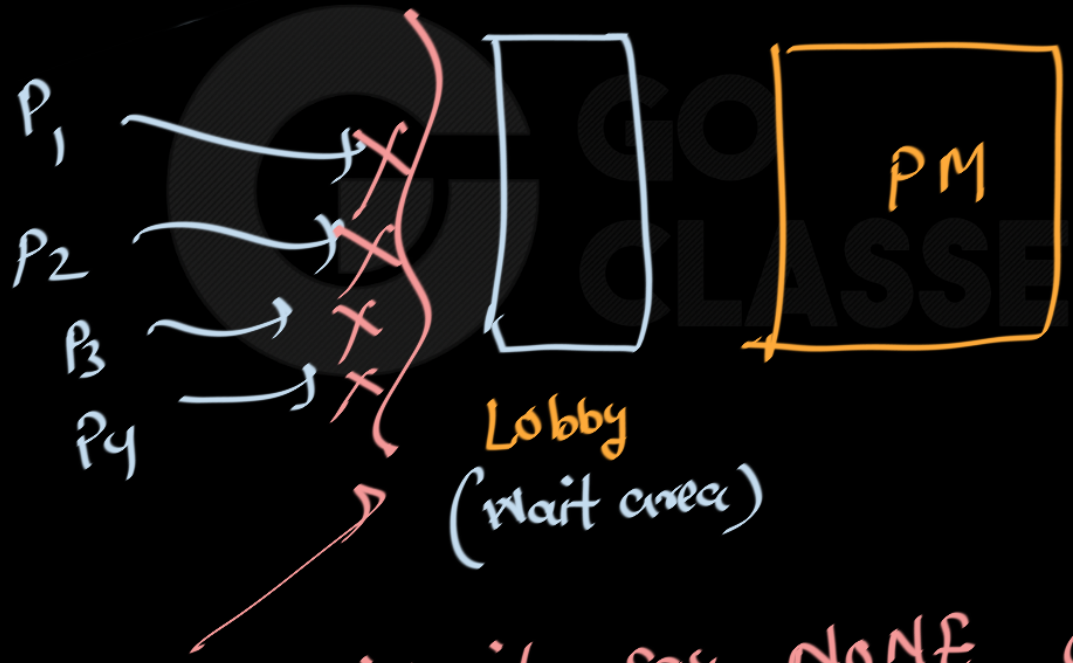


ME + Progress + BW



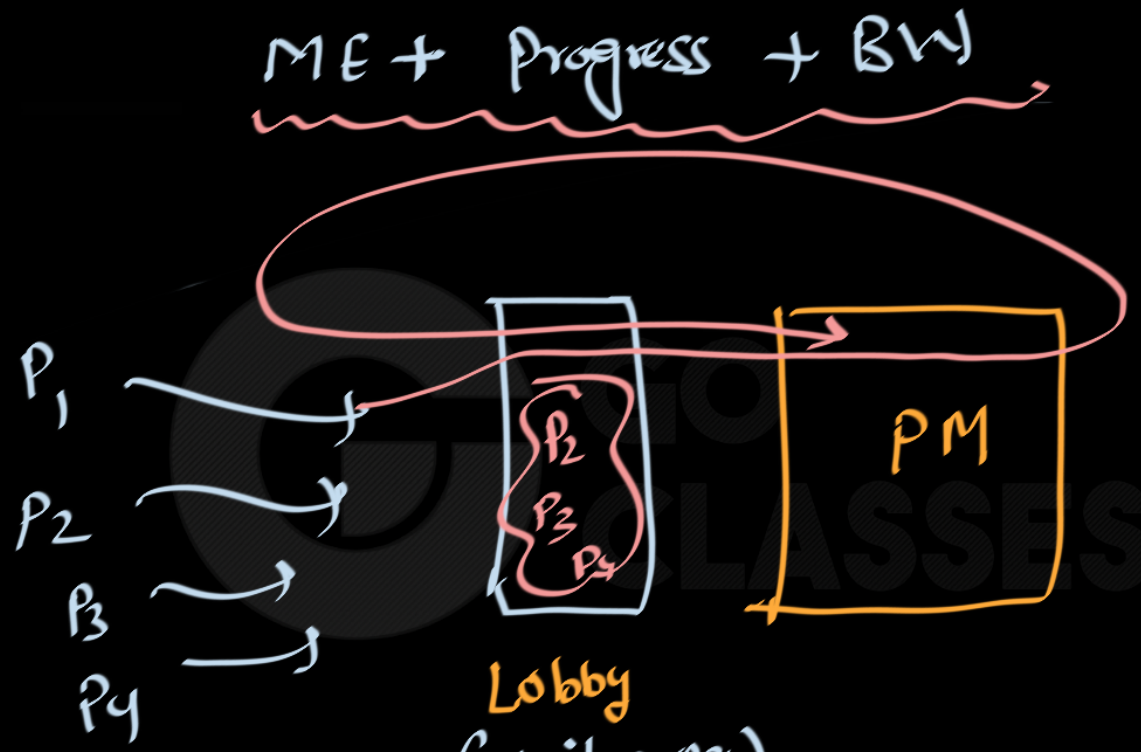
multiple persons won't be allowed to meet

ME + Progress + BW



will it say NONE can enter ⇒ NO

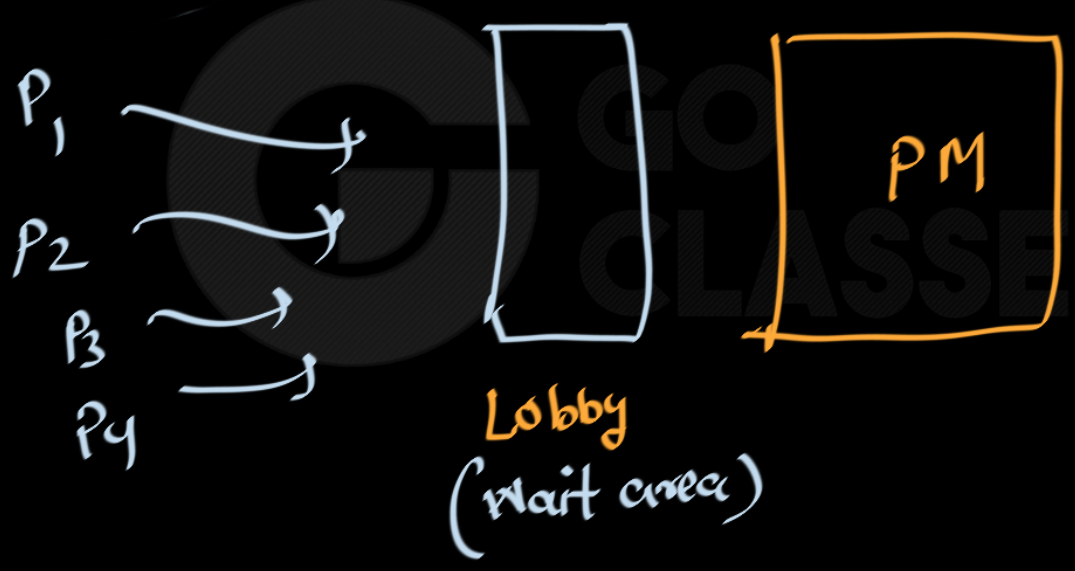
Progress says atleast allow one



if P_2, P_3, P_4 are waiting and P_1 is in cs $\Rightarrow$ i won't allow P_i to reenter

ME + Progress + BW

if any sol^n follows all of this then it is a CS Solution



Critical sections

Thread 1:

```
Enter CS;
count++;
Leave CS;
```

Thread 2:

```
Enter CS;
count++;
Leave CS;
```

How to implement Enter CS and Leave CS?

The Critical-Section Solution

The Critical-Section Solution

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

1. **Mutual exclusion.**
2. **Progress.**
3. **Bounded waiting.**

Implementing Mutual Exclusion

How do we do it?

→ as a user, we can write some code

→ we can OS (using System call) to help us

→ we can take hardware help also.

Implementing Mutual Exclusion

How do we do it?

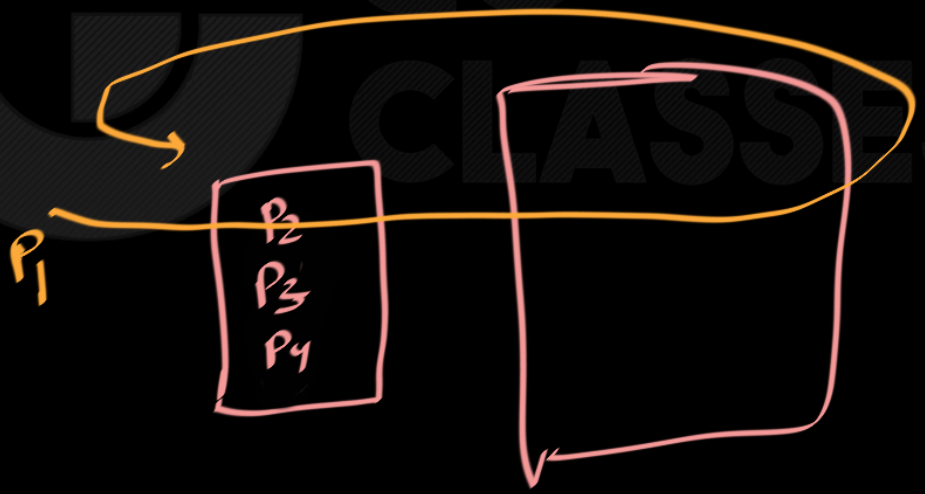
- via software: entirely by user code
- via OS support: OS provides primitives via system call
- via hardware: special machine instructions

Progress meaning in layman terms

The purpose of this condition is to make sure that either some process is currently in the CS and doing some work or, if there was at least one process that wants to enter the CS, it will and then do some work. In both cases, some work is getting done and therefore all processes are making progress overall.

Bounded wait meaning in layman terms

Any process p is not bypassed by some process p'



Bounded wait meaning in layman terms

Any process p is not bypassed by some process p'

Interestingly, Bounded waiting doesn't say if a process can actually enter.
It only says that a process should not be bypassed.

That's ok,

but how to solve questions ?

GO CLASSES

Attempt 1;

We want to write a code that solve CS problem

↓

satisfy ME

progress BW

GO CLASSES

we want to write a code that Solve CS problem

↓

Satisfy ME

Progress

BW

Attempt 1 :

P_1

Entry {

CS

Exit {

P_2

{

CS

{

GO CLASSES

we want to write a code that Solve CS problem

↓

Satisfy ME

Progress

BW

Attempt 1;

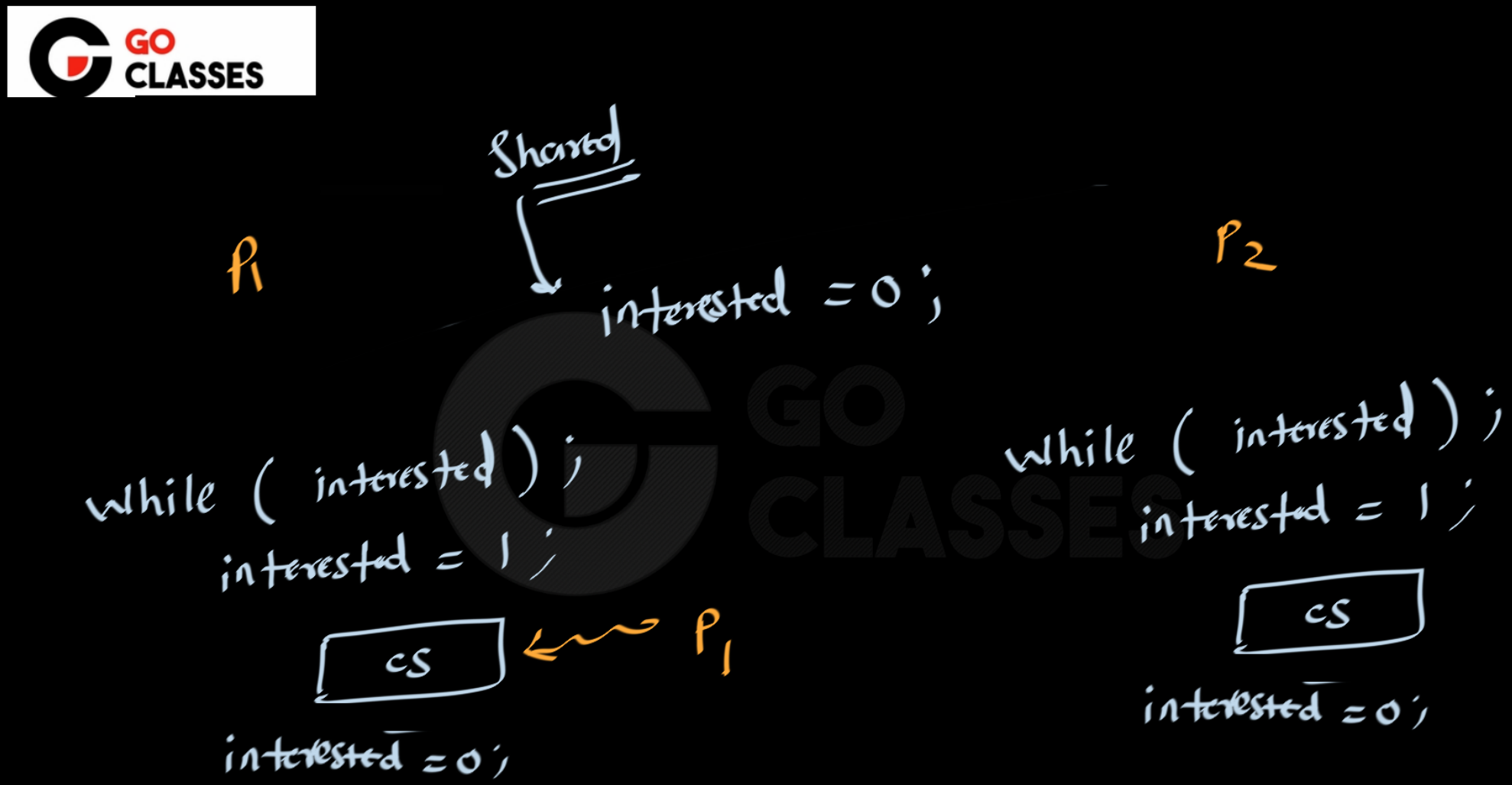
Shared variable

```
int count = 0;        ← shared data
int interested = 0;
```

```
while ( interested );
interested = 1;

CS

interested = 0;
```


Shared
interested = 0;
P1
P2
while (interested) ;
interested = 1 ;
CS
interested = 0 ;
while (interested) ;
interested = 1 ;
CS
interested = 0 ;
P1

GO CLASSES

Shared

interested = 0 / 1 ;

P_1

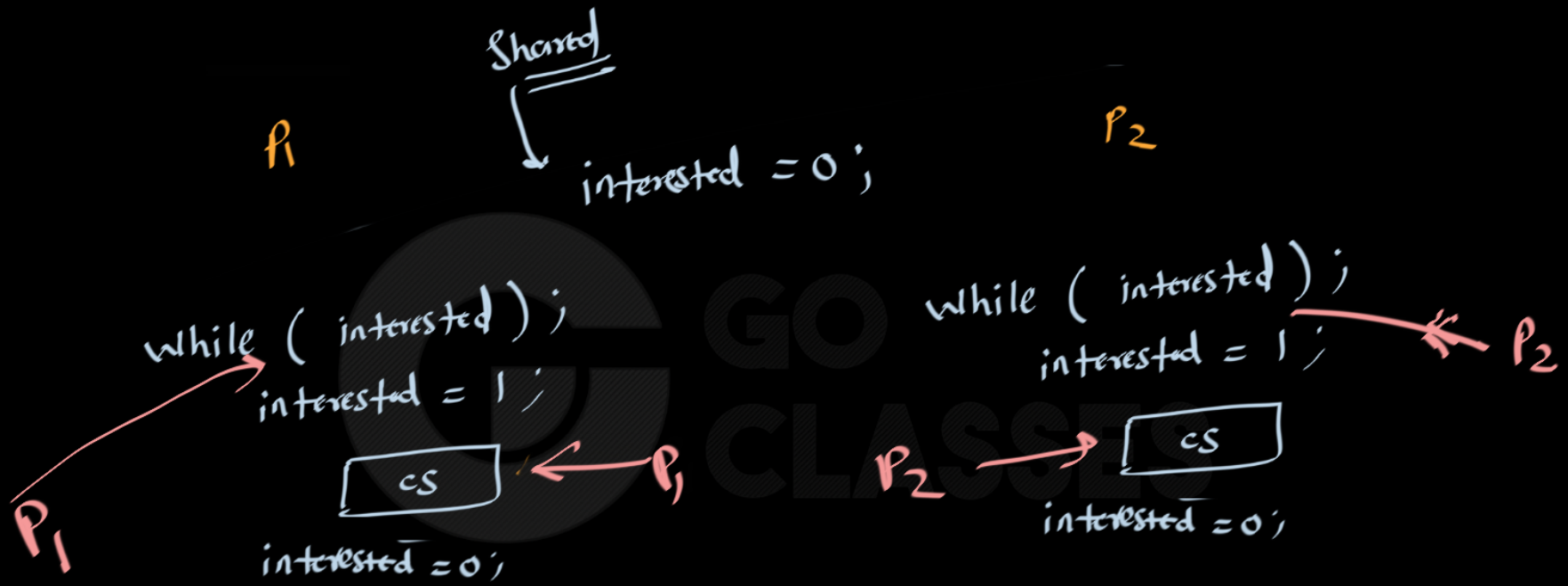
```
while ( interested ) ;
interested = 1 ;
CS
interested = 0 ;
```

P_1

P_2

```
while ( interested ) ;
interested = 1 ;
CS
interested = 0 ;
```

if P_1 is in CS, get preempted, and P_2 wants to enter then can it enter? ⇒ No.



it doesn't satisfy mutual exclusion.

GO CLASSES

```
while ( interested ) ;   ≡   while ( interested == 1 ) ;
```

if "interested" is 1 then

stuck in loop.

How to check these three in questions ?

1. **Mutual exclusion**

2. **Progress (== absence of deadlock)**

3. **Bounded waiting (== fairness)**

1. **Mutual exclusion**
   a. One process in critical section, another process tries to enter → Show that second process will block in entry code
   b. Two (or more) processes are in the entry code → Show that at most one will enter critical section
2. **Progress (== absence of deadlock)**
   a. No process in critical section, P1 arrives → P1 enters
   b. Two (or more) processes are in the entry code → Show that at least one will enter critical section
3. **Bounded waiting (== fairness)**

   One process in critical section, another process is waiting to enter → show that if first process exits the critical section and attempts to re-enter, show that waiting process will be get in

https://pubweb.eng.utah.edu/~cs5460/slides/Lecture08.pdf

Implementation: First try (remember . . .)

Shared variables:

```
int count=0;
int busy=0;
```

Thread 1:

```
while(busy);
busy=1;
count++;
busy=0;
```

Thread 2:

```
while(busy);
busy=1;
count++;
busy=0;
```

Another Try

Code for Process 0

```
while (turn != 0);

CS

turn = 1;
```

Code for Process 1

```
while (turn != 1);

CS

turn = 0;
```

Yet Another Try...

```
int want[2] = {False, False}; // shared variable
```

Code for Process 0

```
want[0] = True;
while (want[1]);

CS

want[0] = False;
```

Code for Process 1

```
want[1] = True;
while (want[0]);

CS

want[1] = False;
```

```
int want[2] = {False, False}; // shared variable
```

Code for Process 0

```
want[0] = True;
while (want[1]);

CS

want[0] = False;
```

Code for Process 1

```
want[1] = True;
while (want[0]);

CS

want[1] = False;
```

This does enforce Mutual Exclusion.

But now both processes can set flag to true, then loop for ever waiting for the other.

This is deadlock

- Mutual exclusion – satisfied
- Progress – not satisfied

GATE CSE 2010 | Question: 23

13,211 views

35

Consider the methods used by processes $P1$ and $P2$ for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables $S1$ and $S2$ are randomly assigned.

Method used by P1	Method used by P2
while (S1 == S2); Critical Section S1 = S2;	while (S1 != S2); Critical Section S2 = not(S1);

Which one of the following statements describes the properties achieved?

A. Mutual exclusion but not progress
B. Progress but not mutual exclusion
C. Neither mutual exclusion nor progress
D. Both mutual exclusion and progress

https://gateoverflow.in/2202/gate-cse-2010-question-23

82

Best answer

Answer is (**A**). In this mutual exclusion is satisfied,only one process can access the critical section at particular time but here progress will not satisfied because suppose when $S1 = 1$ and $S2 = 0$ and process $P1$ is not interested to enter into critical section but $P2$ want to enter critical section. $P2$ is not able to enter critical section in this as only when $P1$ finishes execution, then only $P2$ can enter (then only $S1 = S2$ condition be satisfied).

Progress will not be satisfied when any process which is not interested to enter into the critical section will not allow other interested process to enter into the critical section. When $P1$ wants to enter the critical section it might need to wait till $P2$ enters and leaves the critical section (or vice verse) which might never happen and hence progress condition is violated.

answered Oct 29, 2014 • edited Jun 21, 2021 by **Lakshman Patel RJIT**

comment Follow share this

neha pawar

44

Two processes, $P1$ and $P2$, need to access a critical section of code. Consider the following synchronization construct used by the processes:

```
/*  P1   */
while (true) {
  wants1 = true;
while (wants2 == true);
 /* Critical Section */
  wants1 = false;
      }
/* Remainder section */
```

```
/*  P2   */
while (true) {
  wants2 = true;
 while (wants1 ==
    true);
  /* Critical
 Section */
  wants2=false;
      }
/* Remainder section
      */
```

Here, wants1 and wants2 are shared variables, which are initialized to false.

Which one of the following statements is TRUE about the construct?

A. It does not ensure mutual exclusion.

B. It does not ensure bounded waiting.

C. It requires that processes enter the critical section in strict alteration.

D. It does not prevent deadlocks, but ensures mutual exclusion.

https://gateoverflow.in/1256/gate-cse-2007-question-58

75

Best answer

$P1$ can do wants$1 =$ true and then $P2$ can do wants$2 =$ true. Now, both $P1$ and $P2$ will be waiting in the while loop indefinitely without any progress of the system - deadlock.

When $P1$ is entering critical section it is guaranteed that wants$1 =$ true (wants2 can be either true or false). So, this ensures $P2$ won't be entering the critical section at the same time. In the same way, when $P2$ is in critical section, $P1$ won't be able to enter critical section. So, mutual exclusion condition satisfied.

So, **D** is the correct choice.

Suppose $P1$ first enters critical section. Now suppose $P2$ comes and waits for CS by making wants$2 =$ true. Now, $P1$ cannot get access to CS before $P2$ gets and similarly if $P1$ is in wait, $P2$ cannot continue more than once getting access to CS. Thus, there is a bound (of 1) on the number of times another process gets access to CS after a process requests access to it and hence bounded waiting condition is satisfied.

---

https://cs.stackexchange.com/questions/63730/how-to-satisfy-bounded-waiting-in-case-of-deadlock

answered Apr 12, 2015 • edited Jun 26, 2018 by **Milicevic3306**

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Useful


Arjun

GATE CSE 1988 | Question: 10iib

9

Given below is solution for the critical section problem of two processes P_0 and P_1 sharing the following variables:

```
var flag :array [0..1] of boolean; (initially false)
turn: 0 .. 1;
```

The program below is for process $P_i(i = 0 \text{ or } 1)$ where process $P_j(j = 1 \text{ or } 0)$ being the other one.

```
repeat
        flag[i]:= true;
        while turn != i
        do begin
            while (flag[j]);
            turn:=i;
        end

        critical section

        flag[i]:=false;
until false
```

Determine of the above solution is correct. If it is incorrect, demonstrate with an example how it violates the conditions.

12

Best answer

the above solution for the critical section isn't correct because it **satisfies Mutual exclusion and Progress** but it **violates the bounded waiting**.

Here is a sample run

```
suppose turn =j initially;
```

Pi runs its first statement then Pj runs its first statement then P_i run $2, 3, 4$ statement, It will block on statement 4

Now P_j start executing its statements goes to critical section and then flag[j] = false

Now suppose P_j comes again immediately after execution then it will again execute its critical section and then flag[j] = false

Now if P_j is coming continuously then process P_i will suffer starvation.

---

the correct implementation (for Bounded waiting) is, at the exit section we have to update the turn variable at the exit section.

```
repeat
    flag[i]:= true;
    while turn != i
    do begin
        while flag [j] do skip
        turn:=i;
    end
    critical section
    flag[i]:=false;
    turn=j;
until false
```

57

Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both the processes

Process X	Process Y

/* other code for process X*/
while (true)
{
varP = true;
while (varQ == true)
{
/* Critical Section */
varP = false;
}
}
/* other code for process X */
	/* other code for process Y */
	while (true)
	{
	varQ = true;
	while (varP == true)
	{
	/* Critical Section */
	varQ = false;
	}
	}
	/* other code for process Y */

Here *varP* and *varQ* are shared variables and both are initialized to false. Which one of the following statements is true?

A. The proposed solution prevents deadlock but fails to guarantee mutual exclusion
B. The proposed solution guarantees mutual exclusion but fails to prevent deadlock
C. The proposed solution guarantees mutual exclusion and prevents deadlock
D. The proposed solution fails to prevent deadlock and fails to guarantee mutual exclusion

Answer is (**A**).

GATE CSE 2016 Set 2 | Question: 48

39

Consider the following two-process synchronization solution.

PROCESS 0	Process 1
Entry: loop while (turn == 1); (critical section) Exit: turn = 1;	Entry: loop while (turn == 0); (critical section) Exit turn = 0;

The shared variable turn is initialized to zero. Which one of the following is TRUE?

A. This is a correct two- process synchronization solution.
B. This solution violates mutual exclusion requirement.
C. This solution violates progress requirement.
D. This solution violates bounded wait requirement.

https://gateoverflow.in/39600/gate-cse-2016-set-2-question-48

68

There is strict alternation i.e. after completion of process 0 if it wants to start again.It will have to wait until process 1 gives the lock.
This violates progress requirement which is, that no other process outside critical section can stop any other interested process from entering the critical section.
Hence the answer is that it violates the progress requirement.

Best answer

The given solution does not violate bounded waiting requirement.

Bounded waiting is : There exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted.

Here there are only two processes and when process 0 enters CS, next entry is reserved for process 1 and vice-versa (strict alteration). So, bounded waiting condition is satisfied here.

Correct Answer: C

GO CLASSES

Plan of next lecture:

> formal method or step by step method to check ME, Progress, and BW

> we will try our own Soln's

> we will see GATE PYQs (where they are failing)

> Actual Working Soln of CS Problem.