Lecture 17

Arturo Gangwar to Everyone 7:15 PM

S

Hello Sir , How are you doing??

```
void RecordingReminder ( )
if (Recording == false)
    {
   recording == true ;
    }
else
    {
     ignore this remainder ;
    }
```

Collapse All 2 8

Understand the requirement to solve CS Problem.

→ mutual exclusion

→ Progress

→ Bounded waiting.

Critical-Section Problem Solution

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

1. **Mutual exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Source: Galvin

Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

↳ multiple

Progress : there should be some Progress in critical section

Arturo Gangwar to Everyone 9:32 PM

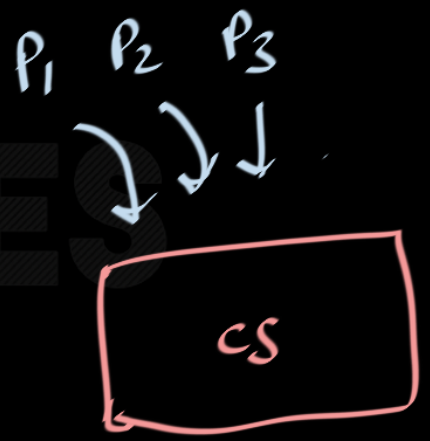
S

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

>, 1

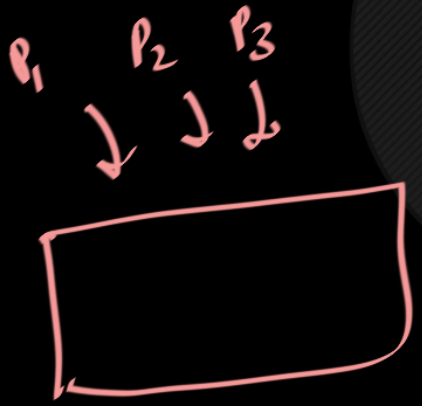
→ if P wants to enter in CS, and no one else wants to enter in cs then P should get enter.

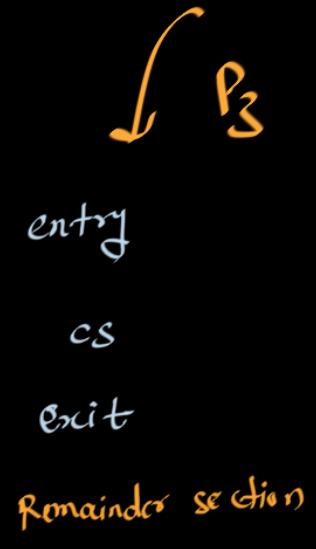
→ if multiple process wants to enter then ATLEAST one should get enter (in other words no deadlock)



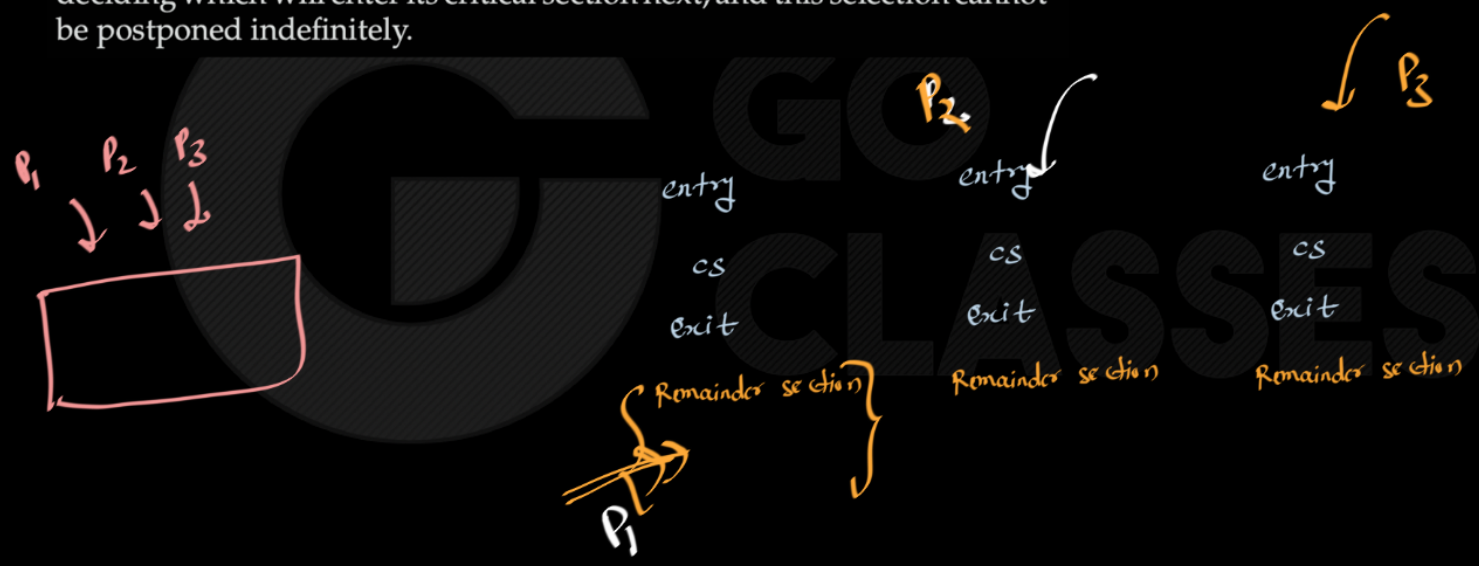
Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

→ No deadlock





Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.



Only P2, P3 decides who will go in cs next. (P1 won't)

Remainder section

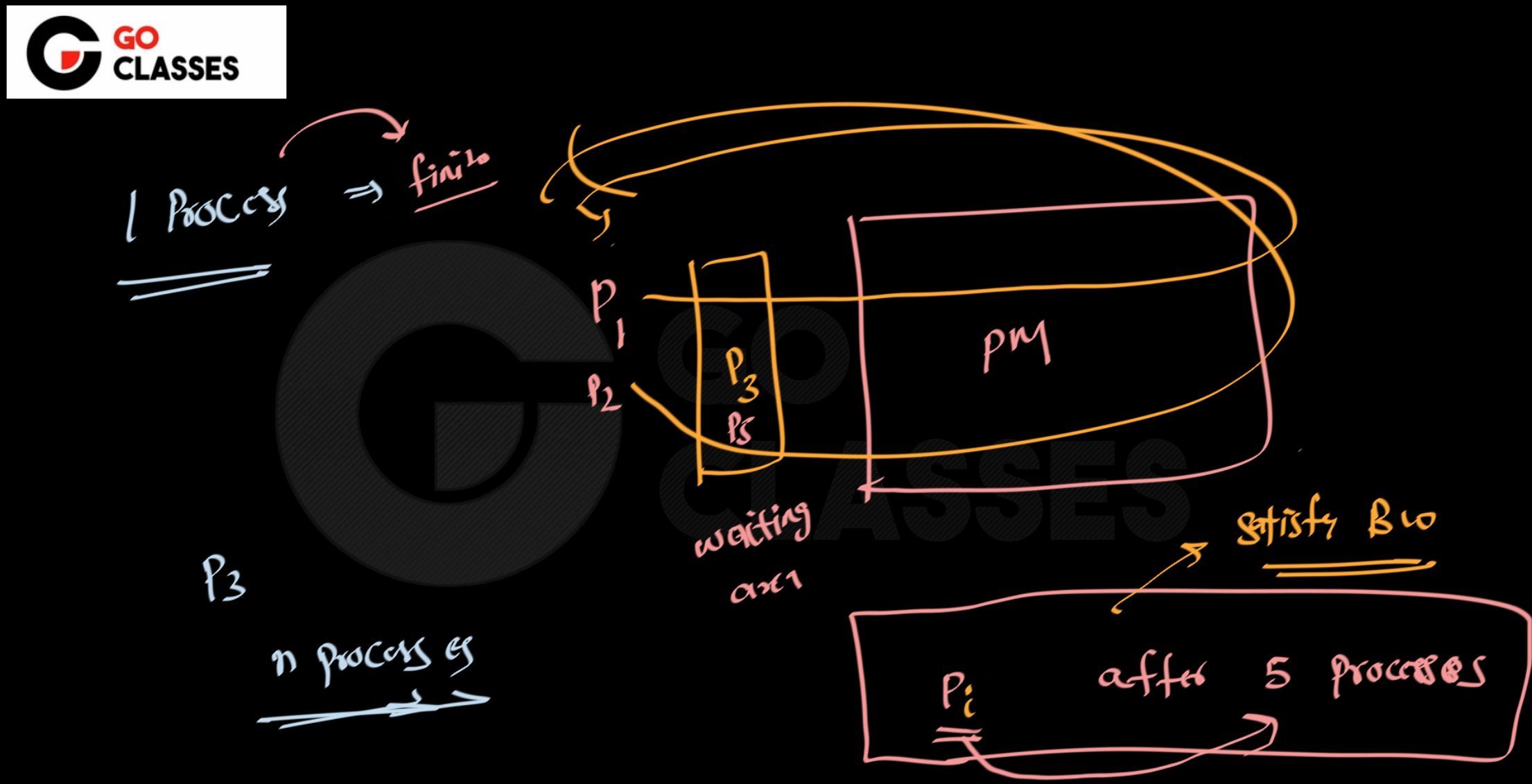
GO CLASSES

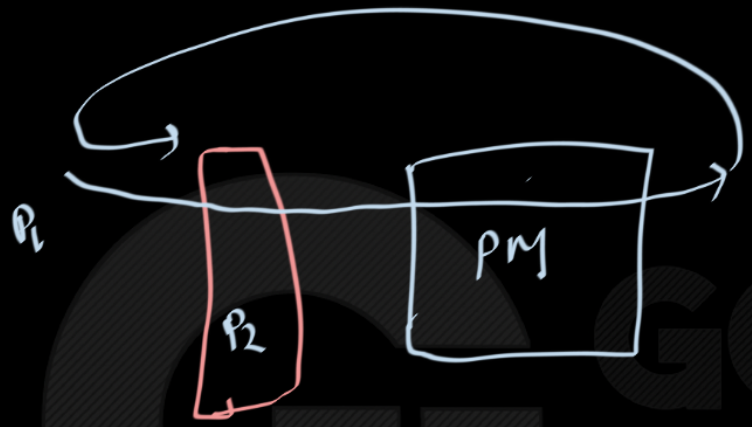
→ i don't want P_i to re enter immediately to CS while others are waiting.

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

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

→ 'it is in the waiting area


1 Process ⇒ finite
P1
P2
P3
P5
PM
waiting area
P3
n Process eg
satisfy B.C
Pi
after 5 processes



⇒ Re enter immediately won't be the issue to BW if we are giving a guarantee on the no. of times it can re enter immediately (or sequentially)

GO CLASSES

in layman terms

1) ME : allow atmost one

2) Progress : allow atleast one

3) BW : Re enter immediately is not possible

(You can reenter but there should be a bound)

That's ok,

but how to solve questions ?

GO CLASSES

$\forall$ executions ME, Progress, and BW is satisfied then it is acceptable Solution

we want to come up with some set of rules that covers all possibilities.

How to check these three in questions ?

1. **Mutual exclusion**

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

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

Operating Systems

THE UNIVERSITY OF UTAH®

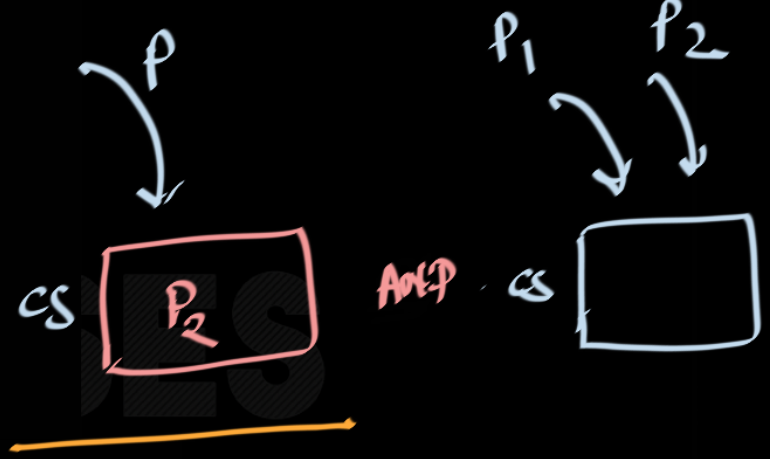
1. **Mutual exclusion**
   a. One process in critical section, another process tries to enter → Show that second process will block in entry code
   b. Two (or more) processes are in the entry code → Show that at most one will enter critical section
2. **Progress (== absence of deadlock)**
   a. No process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

Operating Systems

THE UNIVERSITY OF UTAH®

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



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;
```

Implementation: First try (remember . . .)

Shared variables:

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

busy = 1 → 0;

initial setup for BW

Thread 1:

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

P1

Thread 2:

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

P2

Shared variables:

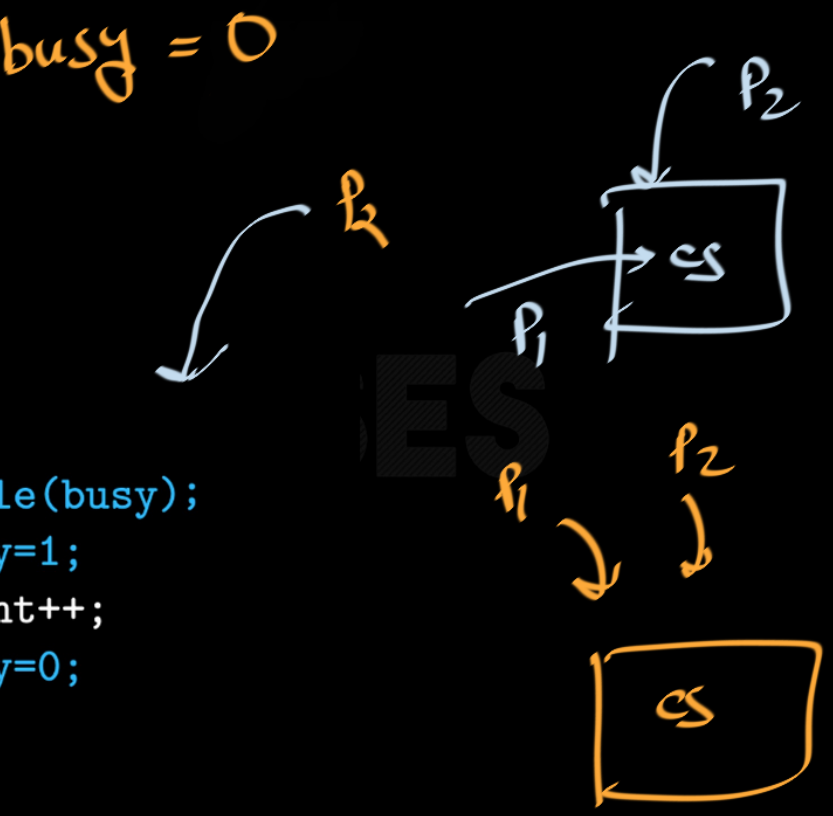
```
int count=0;
int busy=0;
```

Thread 1:

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

Thread 2:

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



busy = 1; 0

Shared variables:

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

↳ interested

Thread 1:

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

P_1

Thread 2:

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

P_2

Since P_1 can re enter hence Bw is not satisfied

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;
```

x { ME : Not satisfied

Progress : Satisfied

x { BW : Not satisfied

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 → No deadlock
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

if busy is initialised to 1 then Progress is also not satisfied

Another Try


```
int turn = 1  // shared variable
```

Code for Process 0

```
while (turn!=0);

CS

turn = 1;
```

Code for Process 1

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

CS

want[0] = False;
```

Another Try

ME ✓

Progress ✗

```
int turn = 1  // shared variable
```

Code for Process 0

```
while (turn!=0);
CS
turn = 1;
```

initial setup

P_0 ←

P_1 →

BW ✓

Code for Process 1

```
while (turn!= 1);
CS
turn = 0;
```

P_1 P_0 P_1 P_0

Another Try

Called as Strict Alteration

- Mutual exclusion – satisfied
- Progress – not satisfied

BW

```
int turn = 1  // shared variable
```

Code for Process 0

```
while (turn!=0);

CS

turn = 1;
```

Code for Process 1

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

CS

want[0] = False;
```

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;
```

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;
```

ME ✓
Progress ✗ (bcoz of deadlock)

Progress ✗ deadlock ✗

GO CLASSES

'initial Setup for Bw

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

T T

Code for Process 0

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

CS

want[0] = False;
```

P0

Code for Process 1

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

CS

want[1] = False;
```

Pj

{ nloco ask P0 to leave CS and re enter in CS. }

Since P0 can not re enter hence Bw is Satisfied

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

Common Mistake.

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

↓ wrong

Bw Not Satisfied

Code for Process 0

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

CS

want[0] = False;
```

P0

Code for Process 1

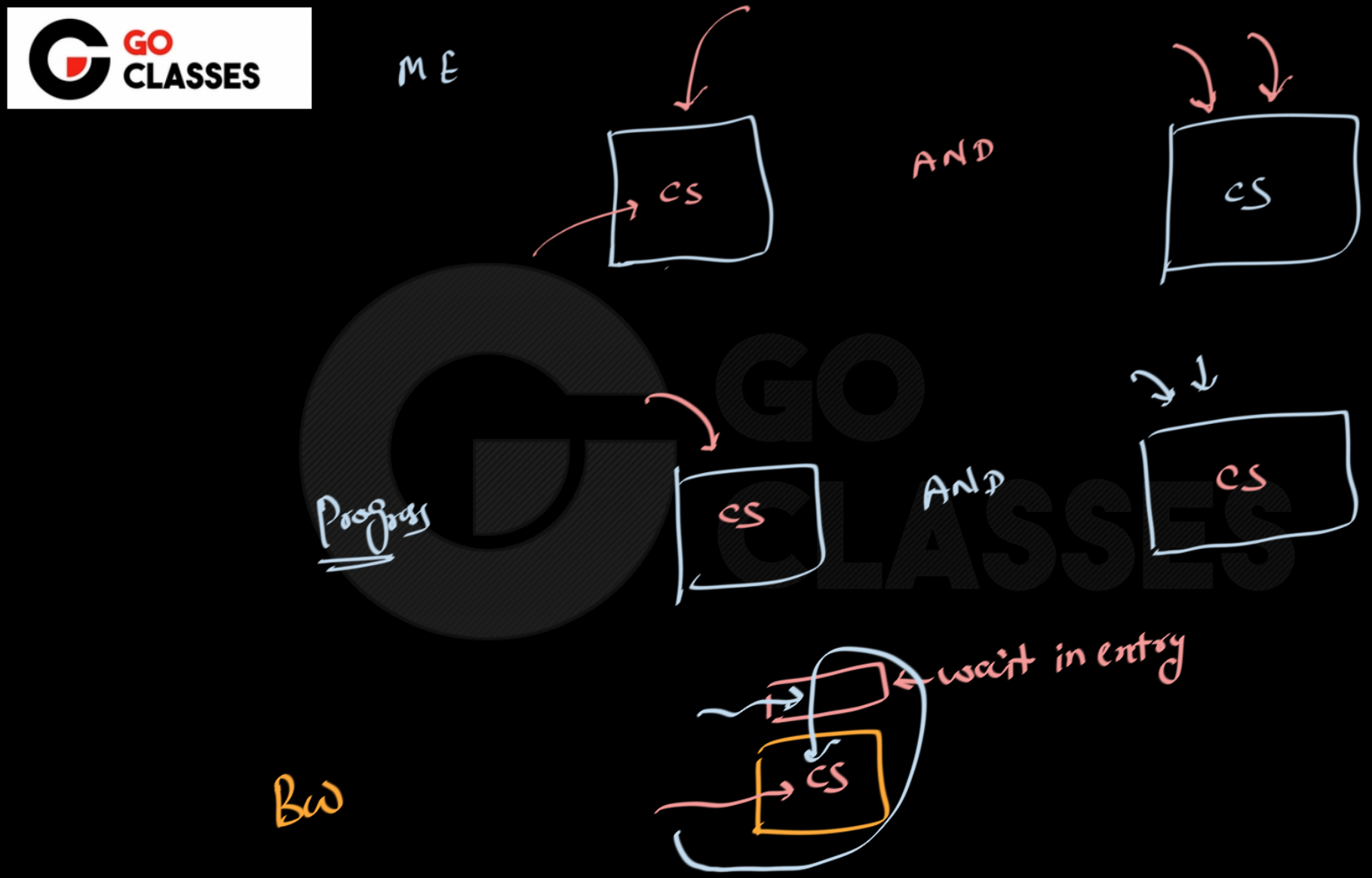
```
want[1] = True;
while (want[0]);

CS

want[1] = False;
```

P1 → Students forget to make P1 in wait area

if Progress is not there then there could be two possible reasons

1) single interested process was not allowed
2) there was a deadlock (ex. strict alt)
   (example - Yet another try)

- deadlock is just one of the reason for progress to not happen.


GO CLASSES
ME
CS
AND
CS
Progress
CS
AND
CS
wait in entry
CS
BW

```
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

BW → Satisfied

Question: 1

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

Question:

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

ME ✓
Prog ✗
BW ✓

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

ME ✓

Progress ✗

Bounded ✓

both conditions can't be true or false at same time.

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

Question: 2

GATE CSE 2007 | Question: 58

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

Question:

GATE CSE 2007 | Question: 58

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

GO CLASSES

```
wants1 = true;
while (wants2 == true);
/* Critical Section */
wants1 = false;
```

```
wants2 = true;
while (wants1 == true);
/* Critical Section */
wants2=false;
```

wants1 = F T F T

wants2 = F T

ME – ✓

Progress – ✗

BW – ✓

75

Best answer

$P1$ can do wants1 $=$ true and then $P2$ can do wants2 $=$ 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 wants1 $=$ 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 wants2 $=$ 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

Question: 3

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;
```

flag[1] = {f, f}

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.

GO CLASSES

flag[i] = { F, F }

P_0

```
repeat
        flag[0]:= true;
        while turn != 0
        do begin
            while (flag[1]);
            turn:=0;
        end

        critical section

        flag[0]:=false;
until false
```

P_1

```
repeat
        flag[1]:= true;
        while turn != 1
        do begin
            while (flag[0]);
            turn:=1;
        end

        critical section

        flag[1]:=false;
until false
```

try with both turn ; if any initialisation says no to some property then no to that property.

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
```

in

Question:

GATE CSE 2015 Set 3 | Question: 10

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

https://gateoverflow.in/8405/gate-cse-2015-set-3-question-10

Answer is (**A**).

Question:

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.

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

Question

Consider two processes P_0 and P_1.

Code for Process P_0

```
while (x==0);
x=0;

CS
```

Code for Process P_1

```
while (x==1);
x=1;

CS
```

Proposed solution satisfy which one of the following ?

- ☐ Mutual exclusion
- ☐ Progress
- ☐ Bounded waiting

Question

Consider two processes P_0 and P_1.

Code for Process P_0

```
while (x==0);
x=0;

CS
```

Code for Process P_1

```
while (x==1);
x=1;

CS
```

Proposed solution satisfy which one of the following ?

❌ Mutual exclusion ❌ Progress ✅ Bounded waiting

Question

Consider two processes P_0 and P_1.

x_1, x_2 = false initially

Code for Process P_0

```
x2 = 0;
while (x1!=1);

CS
x2 = 1;
```

Code for Process P_1

```
x1 = 0;
while (x2 != 1);

CS
x1 = 1;
```

Proposed solution satisfy which one of the following ?

- ☐ Mutual exclusion
- ☐ Progress
- ☐ Bounded waiting

Question

Consider two processes P_0 and P_1.

Code for Process P_0

```
x2 = 0;
while (x1!=1);

CS
x2 = 1;
```

Code for Process P_1

```
x1 = 0;
while (x2 != 1);

CS
x1 = 1;
```

Proposed solution satisfy which one of the following ?

✅ Mutual exclusion ❌ Progress ✅ Bounded waiting

Peterson's Solution

Peterson's Solution


```
// shared variables
int want[2] = {False, False};
int favored = 0;
```

Code for Process 0

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

CS

want[0] = False;
```

Code for Process 1

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

CS

want[1] = False;
```

```
int turn;
boolean flag[2];

do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

GO CLASSES

Hardware Soln ⇒ 40 mins

Semaphore

- Why we need Semaphore
- How to implement it
- advantages of Semaphores (if any)

GO CLASSES

2 more days for Synchronisation

Deadlock ← 2 - 3 days

[Memory management] ← 5 - 6 classes

File System ← 2 - 3 days