Process Synchronization (Moderate)

Need of Synchronization

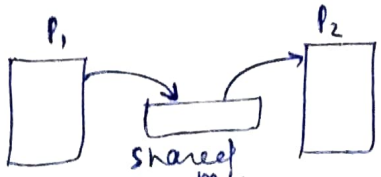
Each process allows to work in its own environment.
Environment means every process can only access its own PCB elements

But if two process communicate with each other means interprocess communication then it might create some problems. So we need of synchronisation and security provided by OS.

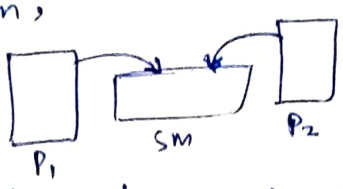
→ In CN two process were present in different host.

→ In OS process are present in same host.

→ Scenio is in such a way that, P_2 is reading the output of P_1 so, first P_1 will occur and then P_2

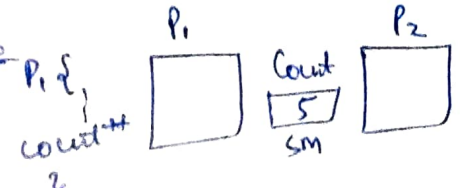


But if both process try to access at the same time then, it is a problem,



→ If two process try to compete or co-operate there may arise consistency.

Ex



Count [5]

Count ++ ;	Count --
1) MOV Count, R_0	1) MOV Count, R_0
2) INC R_0	2) DEC R_0
3) MOV R_0, Count	3) MOV R_0, Count

Assume P_1 scheduled first and executes P_1: 1, 2, 3 then

Count = 6.

and P_2 scheduled second then

P_1: 1, 2, 3 and

Count = 5

If both process occurs one after other then there is consistency.

Case-2 If P_1 executes but $R_0 = 6$

P_1: 1, 2 Count = 5

and P_1 got preempted and P_2 got scheduled and executes

P_2: 1, 2, 3

Count = 4

because P_1 doesn't change the value of count.

and if P_1 comes to execute 3 then it move 6 to count.

P_1: 1, 2 | P_2: 1, 2 | P_1: 3 | P_2: 3
$R_0 = 6$ $R_1 = 4$ Count = 6 Count = 4

→ So value depends on order of execution, of order called as 'race condition'.

Part of the code where process are using shared variable or memory called as 'critical section'.

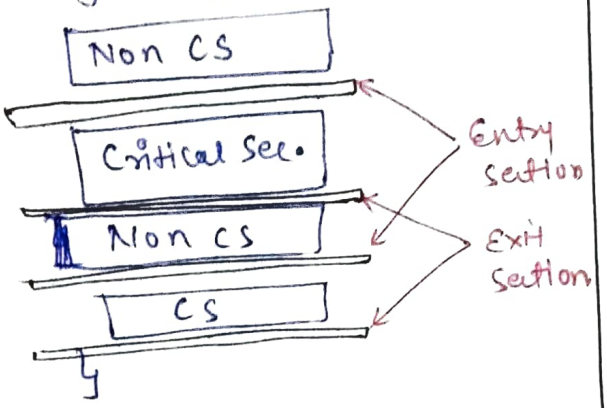
Ex → Count variable

→ So we have to ensure that only one process access a critical section at a time, to provide Synchronization.

Introduction to Synchronization mechanism:

Ex→ Critical section (CS)

Program {



→ If speed of process defines output also called race condition.

→ Problem arises when process with critical section get preempted.

→ We can allow or disallow access to critical section using entry/exit points.

→ Only one process is allow to enter CS at a time, called synchronization mechanism (solution to race condition).

→ But synchronization mechanism might also have problem like—

① Extended CPU time
② Produce Deadlock
③ Doesn't support mutual exclusion

Condition for Synchronization mechanism:

Requirements

① Mutual Exclusion] Primary
② Progress

→ If one process in critical section then other is not allowed called Mutual Exc.

→ Progress should be guaranteed. It means if P_1 is scheduled but it doesn't want to access critical section then must allow P_2 to enter it.

③ Bounded waiting] Seconday
④ Portability or architectural neutrality

→ You should be able to say after how much times other process will get chance to enter in CS, to avoid starvation: Bounded waiting.

→ Solutions or mechanism should be hardware independent.

→ In bounded waiting it doesn't matter's after how much time other process get CPU but it must get it after decided time.

Synchronization mechanism

- Busy waiting (BW)
- No Busy waiting (NBW)

BW: Process is in CPU and stuck in infinite loop without producing any kind of output.

NBW: ~~If fail~~ it doesn't falls in infinite loop.

Lock Variables:

- Software mechanism implemented in user mode.
- Busy waiting solution
- Can be used even for more than two processes.

```
entry section   while (Lock != 0);   -(1)
                Lock = 1             -(2)
                Critical Section     -(3)
exit section    Lock = 0             -(4)
```

- Simplest synchronization mechanism.

Lock = 0 : vacant CS

Lock = 1 : Occupied CS

while (Lock != 0); ' ' → semi-colon means if Lock ≠ 1 process is in infinite waiting. You can break it by making Lock = 0.
↳ (loop)

Assume P_1 executed P_1 : 1, 2, 3 and get preempted without execute (4) and if P_2 get a chance but it will fall in infinite loop because Lock = 1 and as P_2 is in CPU without producing output called busy waiting.

P_1 : 1, 2, 3	P_2 : 1 (infite loop)	P_1 : 4	P_2 : 1, 2, 3

→ This method wasn't providing mutual exclusion so, it gets failed to provide Synchronization.

Lock = 0 → initially.

On assembly lang. code:

```
1) Load Lock, R0
2) CMP  R0, 0
3) JNZ  Step 1
4) Store 1, LOCK
   CS
5) Store 0, Lock
```

Algorithm of assembly code

Preempted

P_1 : 1 | P_2 : 1, 2, 3, 4, CS

→ P_1 : 2, 3, 4 CS

P_1 will not check it again b/c it already checked last time. So, at a time all the process can enters in a CS.

TSL

We can update the Lock Variable concept as -

1) LOAD Lock R0
2) Store 1, LOCK
3) CMP R0, 0
4) JNZ step 1

[CS]

Above code is different from the code we discuss in Lock variable section.

In the previous code preemption can occur at anywhere in between or after 1), 2), 3) but in above code mutual exclusion disturbs if preemption occur after step 1) but still there is chance that mechanism shouldn't follow mutual exclusion.

So to avoid this problem it is necessary to run atleast 2 instruction in a consequtive manner.

[1) Load Lock, R0
2) Store 1, Lock] Atomic

Above 2 instruction must be occur before preemption to maintain Mutual Exclusion.

TSL (Test Set Lock)

TSL Lock, R0 is the single instruction to substitude instead of above 2 instruction.

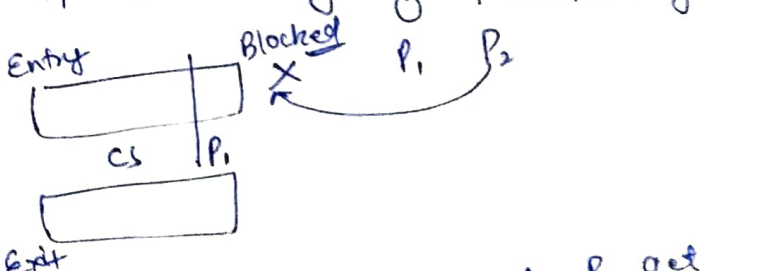
Process that execute TSL instruction first change the value of Lock to 1. this will 'definitely guarantee the mutual exclusion.'

If process doesn't wants to enter then it have no need to execute TSL, this way we can maintain the Progress in synchronization mechanism.

→ Bounded waiting is not guaranteed.
→ It is not portable b/c 2 platform must execute TSL instruction.

Priority Inversion

If P_1 is in critical section but P_2 with higher priority want to execute the same critical section. In this case scheduler allow P_2 to access CPU but entry point doesn't allow P_2 to enter b/c P_1 is already inside



So, CPU is with P_2 but P_1 get preempted inside CS. Hence both process get 'spin locked'.

So, chosing synchronization mechanism depends upon the scheduling algorithm.

- ~~TSL solution is deadlock free.~~
- ~~solution is not starvation free.~~
- ~~The process enter cs doesn't follow FIFO.~~
- ~~More than one process can't enter at the same time.~~

```
void enter_cs (x) {  x=0
    while(test-and-set(x));
}
void leave_cs(x){
    x=0;
}
```

↳ Perform TSL operation

~~On TSL if FIFO guaranteed then there is no starvation.~~

Soln. Gate 2012 Ques: (9mp) → Must to do and analyze

On fetch_and_add (x,i)

suppose X=10, i=2

then fetch and add (10,2) returns 10 and update the value to 12.

L=0 → Initially.

Assume P_1 enters with L=0 and made L=1 but doesn't enters in while loop.

But if P_2 comes it found L=1 and get stuck into while loop making L=1 and 2 continuously (verify the running algo.)

So, only process can enter inside cs and other get fall in infinite loop when P_1, Release lock as L=0 then other process will get enter.

→ Process is mutually exclusive but having starvation for other process

To avoid starvation you can add all the process in a queue to ensure that each process get chance to execute.

But in question there is no queue given.

Option1 (Correct)

P_1	P_2	P_3	P_4	... P_n
L=0	L=1	L=2	L=3	L= range
L=1	L=2	L=3	L=4	L= range++
cs				

→ If process get preempted without executing L=1 then value of L can overflow.

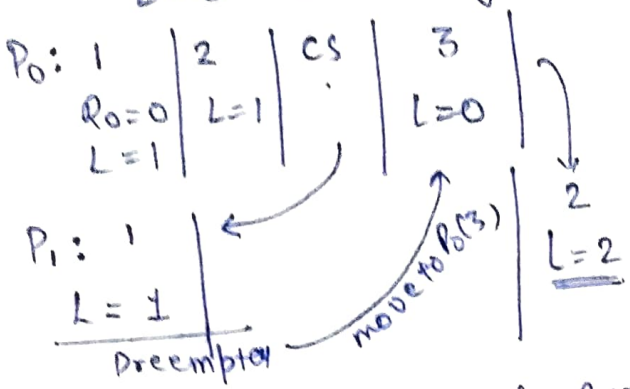
option2 → Correct bcoz if Lock is available then L=0 (Explain at last)

option-3 and option4 ↳ Incorrect bcoz not working and starvation too.

```
while (fetch_and_add(L,1))  -①
    L=1;  -②
    CS
Release lock(L) {
}   L=0;  -③
```

We give a number to each line of code.

$L = 0$ → Initially



Even the value of $L = 0$ by $P_0(3)$ by P_1 knows $L = 1$ due to $P_1(1)$ preemption and fall in infinite loop.

So, option 2 is also correct in question

Ans – 1, 2.

Gate-2010 (P_1 and P_2 Question)

S_1	S_2		
T	T	→ P_2	Mutual Exclusion Guaranteed
T	F	→ P_1	
F	T	→ P_1	
F	F	→ P_2	

Assume case is $S_1 = S_2 = T$ then P_2 is allow to access CS but P_2 is busy in another task. So if P_1 want to access and CPU is free bcoz P_2 is not there. Even after that P_1 is not allowed to enter.

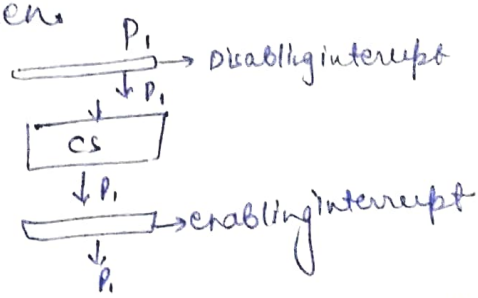
So it not follow progress.

If P_2 doesn't want to enter in CS then P_1 will get starved.

Disabling Interrupts

→ Assume that process is in way to get inside the critical section at the same time another process with higher priority comes at entry point and first process get preempted.

→ To stop above condition we can provide a facility of disabling interrupt so that another process can't interrupt the execution in between.



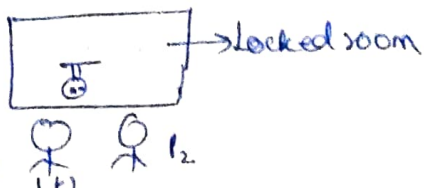
So, disabling interrupt guaranteed –

1. Mutual Exclusion
2. Progress
3. Bounded waited is not guaranteed.
4. Architecture dependent (System must provide support to interrupt)

But this method of interrupt is rarely used bcoz we can't give control to user process.

Turn Variable and Strict Alteration Method

Consider a scenario that there is a locked room that allows only two people (one by one) to get inside.



→ Key किसी एक के पास होगी।

→ Person with key will enter first and come back after completing Critical Section then lock the room and handover the key to another person.

→ So, mutual exclusion is guaranteed bcoz only one can enter at a time.

→ But if person with key doesn't wants to enter inside room then another process will get starved.

→ So, progress doesn't guaranteed.

→ Even if person with key doesn't want to enter in cs, फिर भी उसे room में जाकर वापस आना होगा to transfer the key.

→ Also bounded waiting guaranteed.

→ Points to remember:

① Software mechanism Implemented at user mode.

② Busy waiting solution

③ Only for 2 process, called in any orders

`Int turn = 0;` Global Variable

for P_0	for P_1
Non CS	Non CS
while (turn != 0);	while (turn != 1);
CS	CS
turn = 1;	turn = 0;
Non CS	Non CS

→ Busy waiting (Infinite loop)

Turn will either get 0 or 1 so, only one process will get inside.

→ Mutual exclusion guaranteed

Both the processes are inside the cs alternatively in 'normal execution.'

→ Mechanism is architectural neutral.

→ So, turn variable is better than lock variable except progress.

Assume for P_0 and P_1 we have Interested[0] and Interested[1] respectively.

P_0	P_1
Non CS	Non CS
Interested[0] = True while (Interested[1] == True);	Interested[1] = True while (Interested[0] == True);
CS	CS
Interested[0] = false	Interested[1] = false;

→ And this way we can guaranteed,
→ Mutual Exclusion
→ Progress is also guaranteed here.
→ Bounded Waiting not possible bcoz of deadlock

P_0	P_1	P_0	P_1	W→ While loop
$I_0 = T$ CS	$I_1 = T$ W2	$I_0 = F$ $I_0 = T$ W2	W2	

→ Architectual Neutral.
→ So, both turn and interested variable have its own pros and cons. But peterson combine both to give a unique solution.

Peterson's Solution

→ Software mechanism at user mode.
→ Busy waiting solution
→ 2 process solution
→ It uses turn & interested variable.
→ No overhead for OS.

```
# define N 2 → Two process
# define TRUE 1
# define FALSE 0
int interested [N] = false] → Initially both process are false
int turn;
void Entry_Section (int process)
{1) int other ;
2) other = 1 - process ;
3) interested [process] = True
4) turn = process ;
5) While ( interested [other] == True &&
          turn == other);
}
void Exit_section (int process){
6) interested [process] = FALSE ;
}
```

→ Tracing of Algorithm

interested 0 F
1 T

turn 1

P_0: 1 2 3 4 5 [CS] | Preempted
P_1: 1 2 3 4 5 (stuck in loop)
P_0: 6 | Preempted
P_1: 5 [CS]

→ As long as other process is not interested another process will get into CS, so progress and mutual exclusion is guaranteed.

→ Architectural Neutral

P_0: 1 2 3 | P_1: 1 2 3 4 5 (stuck in loop)
↳ Preempted (1)

I 0 F T
1 F T

turn 1 → 0

→ So, process that set interested variable first get chance to access CS, is wrong one.

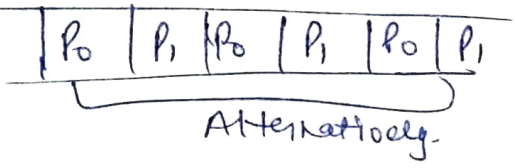
P_0: 4 5 (stuck in loop) | P_1: 5 (not stuck in loop)

on P_1: 5 turn ≠ Process, (so not stuck in loop)

→ Process that access turn variable first will enter in CS first.

We can access it using queue Process that access turns first insert it in queue and then do alternatively.

Ex→ Assume P_0 access turn first



→ Bounded waiting guaranteed.

So, this Algo. supports all four properties of mechanism.

→ Disadvantage → for 2 process only.

So, peterson's Algo. is the best known Algo. at user level without taking help of hardware.

Synchronisation mechanism without busy waiting

To avoid busy waiting we use sleep and wakeup mechanism

Due to S & W mechanism CPU will never be idle and process will never be in priority inversion.

Sleep() and wake_up() are systems calls. Using sleep() process get sleep and using wake_up() a sleep process wake up.

Ex→ Producer-Consumer Problem

→ Consumer reads the output written by producer.
→ Also called Reader/writer problems
→ Output produce store in buffer.

```
#define N 100 // slots in buffer
#define count=0 // no. of items in buffer
void producer (void)
{
 in temp;
 while(True) {
 temp = produce_item();
 if (count==N){sleep();}
  insert_item (temp);
  count = count + 1;
 if (count==1)
   wake_up (Consumer);
}}
void consumer (void) {
 int item;
while(true) {
    if (count==0){sleep();}
item= remove_item ();
    count= count-1;
 if (count == N-1){
      wake_up (producer);}
 consume_item (item);
}}
```

Both the producer & consumer might get into deadlocks.

→ Assume that consumer checks the count value as 0 and before going to sleep it gets preempted. and Producer comes in action. When producer comes at wake-up (consumer) this step will get wasted bcoz consumer is not sleeping

→ As the buffer get completely filled producer will get to sleep() and consumer comes under action from last preempted state and it also gets sleep().

→ So, both get sleep() at the same time.

So To avoid this, use a bit to ensure that consumer will not go to sleep().

If that bit is set and consumer comes to sleep then he will think that someone comes to wake me up in my absence and I have no need to sleep.

But this will work only for one consumer and producer. bcoz bit is set of producer. to avoid missing of wake-up() calls.

For multiple producer and consumer we have semaphores

→ Semaphores needs support from OS.

Semaphores: We can't use it in user mode bcoz at the same time both the user and OS can try to access it that create inconsistency.

→ Variables on which read, modify and update perform automatically in kernal mode (no preemption.

① Counting semaphore
② Binary semaphore (Mutexes)

Counting Semaphores

It should have some integer value either positive, -ve or 0.

Suppose int value = 3 it means at a time 3 process can access the critical section simultaneously.

value = 3 2 1 0 -1 -2

P_0 P_1 P_2 P_3 P_4

3 inside critical section

2 inside Queue

P_4 change the value -1 to -2 and get enters in Queue.

So, -2 means there are 2 processes inside the Queue 'Q'.

As process enters in CS decrease the value by 1 and if process came out of Queue increase value by 1.

Process came out of cs, check the value magnitude if it is negative then it wake up the process in queue.

→ Using queue bounded waiting is guaranteed.

→ If value = 1 then mutual exclusion is guaranteed.

→ Also progress is guaranteed.

Symbols Used for

Decrement	Increment
Down	UP
P	V
Wait	Signal

Here we use Down & UP,

```
Struct Semaphore {
        int value;
        Queue type L;
}
```

(-ve) value denotes no. of process blocked.

Queue contains PCB's.

```
Down (Semaphore s) {
    S.value = S.value - 1;
    if (S.value < 0) {
        put process (PCB) in L;
        Sleep(); }
    else return;
}
```

```
UP (Semaphore S)
{ S.value = S.value + 1;
  if (S.value <= 0) {
    select a process from L;
    wake up();
  }
}
```

Q A counting semaphore was initialized to 10. Then 6P (wait) and 4V (signal) operation are computed. What is the result?

→ 10 − 6 + 4 = 8 Ans

Q S = 7, then 20P 8 15V. Find final S?

→ S = 7 − 20 + 15 = 2 Ans

Binary Semaphores or mutexes

→ It just use for operation not for counting.

```
Struct Bsemaphore {
    Enum value (0,1);
    Queue type L;
}
```

L → Contains all PCB's corresponding to process get blocked, while performing down operation unsuccessfully

→ Allows only two process at a time, then support mutual exc.

```
Down (Semaphore S) {
  if (S.value == 1) {
      S.value = 0; }
  else {
  Put the process in S.L;
    Sleep();
} }
```

```
V(Semaphore S){
  if (S.L is empty) {
    S.value = 1; }
  else {
  select a process from SL;
    wakeup();
  }
}
```

To do → Gate 1992 Question
While practice or revision

→ Generally scheduler used to give execution order to process but we can also maintain a order using Mutex too.

Ex:-

```
mutex a=1, b=0;

P0                      P1
while(true){            while(true){
  P(a);                   P(b);
  Print("1");             Print("0");
  V(b);                   V(a);
}                       }
a=0, b=1                b=0, a=1
```

Output → 1 0 | 0 | 0 - - -

Order will always as above sequence if P_0 execute first but if P_1 execute first then, due to P(b) P_1 get blocked bcoz b=0 in initial so you can't get perform down on b. and whenever P_0 comes it will start the same order of execution as previous.

Ex → On Mutex

```
mutex a, b; a=1, b=1;

P0                      P1
while(true){            while(true){
1) P(a);                5) P(a);
2) P(b);                6) P(b);
   <cs>                    <cs>
3) V(a);                7) V(a);
4) V(b);                8) V(b);
}                       }
```

→ Mutual exclusion is possible bcoz process that perform P(a) and P(b) first gets entry to cs.

P_0: 1 | P_1: 5 | P_0: 2 cs 3 | P_1: 6

(P_1: 5 → Blocked; P_1: 6 → Blocked)

P_0: 4 | P_1: cs 7 8 |

→ In above two process we can acheive mutual exclusion using one mutex only.

→ If we can acheive it using one mutex then we can acheive it using 2, 3, ... n mutex too.

If we exchange the steps 5) & 6) as

5) P(b); 6) P(a);

then both process might get in deadlock as—

P_0: 1 | P_1: 5 | P_0: 2 | P_1: 6

(P_0: 2 → Blocked; P_1: 6 → Blocked)

→ Both process get blocked.

→ But deadlock is not always possible.

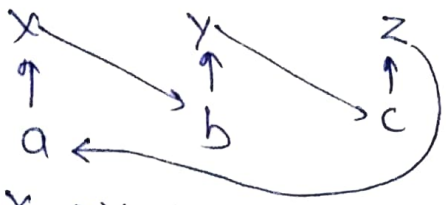
Gate 2013 [Must read soln from other source too]

Option a)

X: P(a) | Y: P(b) | Z: P(c)

X: P(b) | Y: P(c) | Z: P(a) P(a)

Blocked



X → Y → Z Loop found so there is a deadlock.

Option(b)

X: P(b) | Y: P(b) | Z: P(a)

Blocked

X: P(a) | Y: P(b) | Z: P(c)

Blocked

X: P(a) | Y: Blocked | Z: P(a)

Blocked

X | P(a)

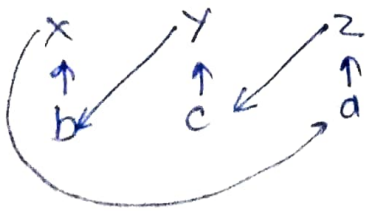
→ Not in deadlock X and Y will execute after completion of Z.

Option(c).

X: P(b) | Y: P(c) | Z: P(a)

X: P(a) | Y: P(b) | Z: P(c)

Blocked



Cycle found so deadlock.

Option(d)

~~X: P(a) | Y: P(c) | Z: P(c)~~

Blocked

~~X: P(b) | Y: P(b)~~

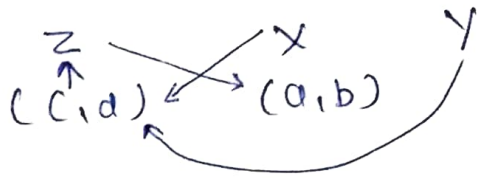
Blocked

Z: P(c) P(a) | X: P(a) P(b)

Y: P(c) | Z: P(a) | X P(c)

Blocked

→ Preempte each process after 1 step and find there is a deadlock or not



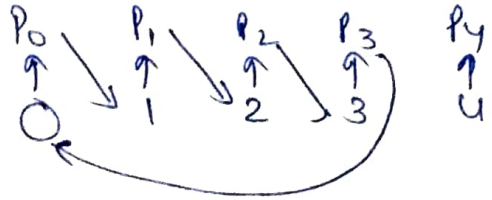
In cycle only Z and X are there. So Z and X are in deadlock and Y is suffering.

Gate 2000

P_0 :	P(m[0])	P(m[1])
P_1 :	P(m[1])	P(m[2])
P_2 :	P(m[2])	P(m[3])
P_3 :	P(m[3])	P(m[0])
P_4 :	P(m[4])	P(m[1])

P ↔ wait both are same.

If we preempt each process after one step then.



$P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3$

Cycle so deadlock

Mutual exclusion is not guaranteed because two process can enter in CS at a same time.

Ex → P_0 and P_2 because both have no mutex in common,

Similarly, (P_1 and P_3) or (P_2 & P_4)
(P_4, P_3)

→ Max. 2 process possible in CS at same time because there is no more than 2 process with all different mutexes.

Question on Dining Philosopher Problem

P_i = 0, 1, 2, 3 M_i = 0, 1, 2, 3

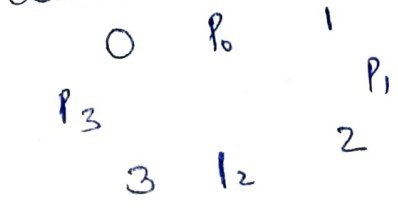
```
Pi:  P(Mi);
     P(Mi+1 mod 4);
     CS
     V(Mi);
     V(Mi+1 mod 4);
```

P_0 : P(M_0) P(M_1)
P_1 : P(M_1) P(M_2)
P_2 : P(M_2) P(M_3)
P_3 : P(M_3) P(M_0)

→ deadlock possible.

To avoid deadlock we allow $P_0 \to P_2$ follow above Algo. and for P_3 to follow

P_3: P(M_0) P(M_3) to avoid deadlock.



→ It means take one process out of order to avoid deadlock

→ Must read about dining philospher problem and take notes with imp. point.

→ Every philospher takes left chopstick first and then right one.

Gate 2003: Imp to do while revision.

→ In these kind of questions verify in the option that allows only one process to access critical section at a time.

Gate 2006 (Problem on Barrier)

Barrier → It is a busy-wait Solution.

Barrier is a point where all the processes are tends to meet to go forward.

Cont -

Implementation

```
void barrier {
1) P(S);
2) process_arived ++;
3) V(S);
4) while (process_arived != 3);
5) P(S);
6) process_left ++;
7) if(process_left == 3)
8) {  process_arived = 0;
9)    process_left=0;
10) }
11) V(S); }
```

→ Above algo. is designed for 3 processes.

→ Every process must have to follow the or execute the complete algo.

In given options

→ a) is wrong bcoz counting semaphore will allow or disable the mutual exclusion.

b) is right bcoz assume that process_arived = (PA) say
process_left = (PL) say
when first 3 process enters then PA = 3 and assume process 1 lefts with PL = 1
जितनी देर में बाकी दो Process PL की value को increment करें उससे पहले ही P_1 को Process लौट कर Barrier पर आगई means
PA = 4 और उसके बाद PL = 3 हुआ

अब PL = 3 के बाद 8), 9) step की वजह से PA = PL = 0 but Process 1 तो वापस आ चुकी है और P_2 और P_3 का wait कर रही है। अब जब P_2 और P_3 आएँगे तो PA की value 2 करेंगे और while loop में infinite time के लिए stuck कर जाएँगे क्योंकि all the process are arrived but value is still 2.

This way deadlock occurs.

→ This problem can be solved as ~~the~~ at the beggining of barrier waits until the process_arrived becomes '0' before proceeding to execute P(S).

Gate-2008: Implementing counting semaphores using mutexes.
very good Question to understand, so must watch its video again after 5:00 mins. (But first do it by self)

→ Value of X_b can't be 1 initially bcoz both process get blocked.

So, (0,0 and 0,1) not possible.

Ans, (1,0) Explanation ↓

Each counting semaphore have two parts ① int value, ② Queue L.

X_b {0, 1} → is a mutex allow only single process to access Counting semaphore 'S' at a time.

and L_x is a list that contain process which gets docon bcoz X_b is only allowing one process but counting semaphore can allow more than one process, so all the other process waiting gets inside L_x.

Y_b {0, 1} → is used to give access to only one process from L_y.

L_y → is a list contains process get down bcoz value of 'S' was negative.

Assume S=4 and P_0 comes and perform down on 'S' and enters and then P_1 come and make S=2 and get enters in L_x Similarly P_2, P_3 come and get inside L_x with S>0

and after this P_4 comes and make S=-1 get blocked and get inside L_y.

→ So, L_x and L_y both containing different kind of process.

Gate 2010: In ques given that $S_0 = 1$, $S_2 = 0$ but no info about S_1, so assume

$S_0=1, S_1=\overset{0}{\cancel{1}}, S_2=0$

P_0: 1 2 3 4 | P_1: 1 2

↓ $S_1=0$

P_0: 1 2 3 4 | P_2: 1, 1 2

P_0: 1 2 3 4 | P_0: 1 2 Blocked

o/p 000

$S_0=1, S_1=0, S_2=0$

Not complete solution watch videos to get solution if required.

Ans- atleast 2

Imp Questions to do while revision and also given in videos

① Gate 2013

② Gate 1995 and 1993.

Deadlock (Easy)

A set of process are said to be in deadlock, if they wait for happening of an event caused by others in the same set.

→ Starvation is long waiting.

→ Deadlock is infinite waiting.

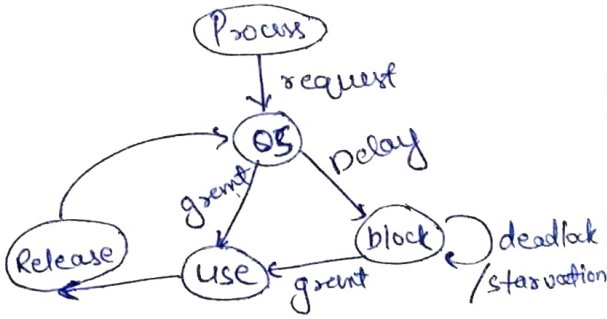
Necessary Condition

→ Mutual Exclusion

→ Hold & Wait

→ No premption

→ Circular wait



Questions:-

Single instance and multiple instance resource. Instance defines no. of processes can access resource at a time.

A system is having 3 user process each requires 2 units of resource. the minimum no. of units of R such that no deadlock occur

→ 3 process x 2 resource = 6

→ max. 6 resources requires but here asking for mino

No of Resources	P_1	P_2	P_3
2	1	1	Blocked
3	1	1	1
4	1	1	1

All three waiting for 2nd resource and no one get so deadlock

free resource ← (1)

Process that get free resource will execute first and transfer it to another one.

So starvation might possible but no deadlock.

So, min resource = 4

Assume P_1, P_2, P_3 requires 2, 3, 4 resources respectively at a time then, how many min. resource units require.

No. of resource	P_1	P_2	P_3
"3"	1	1	1
"4"	1	1	1
		1	
"5"	1	1	1
		1	1
"6"	1	1	1
		1	1
"7"	1	1	1
		1	1

(1) → free resource

Here each process will access free resource execute and transfer it to another

So min resource = 7.

Trick

P_1	P_2	P_3
2	3	4

→ find max. no. of resource that can provide deadlock as (resource required − 1)

P_1	P_2	P_3
(1)	(2)	(3)

Total resource at max are not sufficient to avoid deadlock = 1+2+3 = 6

and add 1 to it as

6+1 = 7

So, 7 resource min. required to avoid deadlock.

1 → resource is free resource.

(3)

P_1	P_2	P_3	---	P_n
x_1	x_2	x_3	---	x_n

→ Min. resource required

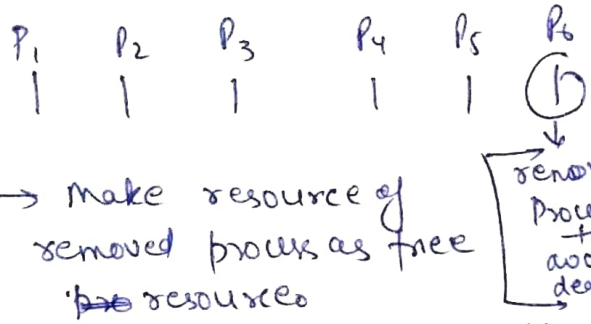
$$[(x_1+x_2+x_3+\cdots+x_n) - n] + 1$$

$$\left[\sum_{i=1}^{n} x_i - n\right] + 1$$

(4) Given, resource has 6 instances each process needs 2 resources then how many process can work without deadlock.

→ Allot 1 resource to each process then deadlock occur but if we take 1 resource

back from any process and make it free then no deadlock occur but that process get Starved

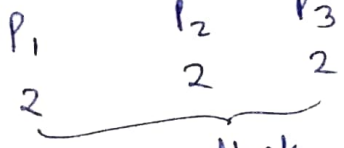


→ make resource of removed process as free ~~free~~ resources

So, max. 5 process are allowed

→ Min. 6 process required for deadlock.

→ Resource = 6, each process require 3 resources find max. of process without deadlock.



So, ~~only~~ 2 process or max. 2 process can exist without deadlock.

→ Resources = 100, P_i = 2 instance

P_1 ---- P_{100}
1 1 ---- 1

→ 99 process at max

Trick → find min. process to perform deadlock and then substract '1' from the number.

→ R = 100 P_i = 3 instance

P_1 ---------- P_{50}
(2) ---------- (2)

→ 49 process as max. without deadlock.

→ To find min. process to perform deadlock allot all resources to some no. of process so that each process get (required instance − 1)

→ In above example we provide 100 resources to 'n' process so that each process get (3−1) = 2 resources

So, $n = \frac{100}{2} = 50$.

Ans → (50 − 1) = 49.

→ R = 100, P_i = 4, n = ?

⇒ $n = \left\lceil \frac{R}{P_i - 1} \right\rceil = 34$

→ for min. use ceil function and for max. use floor function.

Ans = 34 − 1 = 33

Gate 1992

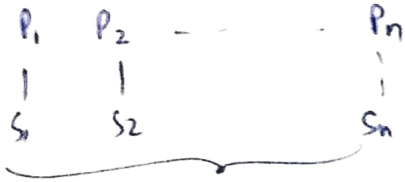
$n = \left[\frac{6}{3-1}\right] - 1$

$= 6/2 - 1$

$= 3 - 1 = 2$ Ans//

max 2 Process for deadlock free

Gate 2005: To see the question.



$$(S_1 + S_2 + \dots + S_n) - n + 1 = m$$

$$\sum_{i=1}^{n} S_i - (n+m) + 1 = 0$$

$$\sum_{i=1}^{n} S_i = (n+m) - 1$$

$$\Rightarrow \sum_{i=1}^{n} S_i < (m+n) \text{ to avoid deadlock}$$

Gate 2006. (very imp) [Must solve]

→ Solve it byself and watch video if required

Ans → $(x_p + x_q) \geq \min_{k \neq p,q} (y_k)$

→ If process that have no need of more resource, then we can use those resource to distribute among other process

P_1	P_2	P_3	--	P_p	P_q	...	P_n	
2	x_2	x_3		x_p	x_q		x_n	→ alloted
y_1	y_2	y_3		0	0		y_n	→ required

x_p and x_q needs no more resources, so after Komplettion of P_p and P_q we can use all resources $(x_p + x_q)$ to satisfy the min. of (y_k).

→ This way we can avoid deadlock.

Deadlock Handling

1. Deadlock Ignorance
2. Deadlock prevention
3. Deadlock avoidance
4. Deadlock detection and recovery.

→ Windows and Linux follows deadlock ignorance.

→ Since, deadlock is a very rare problem, so we generally ignore it.

→ In deadlock prevention we generally try to disable any one of the necessary condition for deadlock from
1. Mutual Exc. 2. Hold and wait
3. No premption. 4. Circular wait

→ In deadlock avoidance, while allocating resource we verify that this allocation doesn't leads to deadlock.

→ Deadlock detection & recovery is very popular methods we allow deadlock to happen and then recover it.

Deadlock prevention

Condition	Approach
Mutual Exclusion	Spool Everything
Hold & wait	Request all the resources initially

Condition	Approach
No Premption	Take resource away
Circular Wait	Order resources numerically

To see

Spooling:- Simultaneous peripheral operations online.

Disabling mutual exclusion is not 100% practical solution, it is possible upto some extent only.

→ ! (Hold & Wait)

⇒ ! Hold or ! Wait

If you don't want to wait then request all the resources in the begging.

→ But prous don't know about the needed resources in the begging.

→ So this condition can't be disable in practical easily.

Similarly preemption ~~and circular wait~~ is also not practically possible so, we generally can't perform deadlock prevention.

→ Circular wait is sometime practically possible.

→ Every proces. need ask for resource with higher index.

Ex 

→ If P1 have hold on R3 then it will ask for resources R4, R5, R6 not, R1 and R2.

→ So, only circular wait can be implemented practically.

Safe, Unsafe, Deadlock Avoidance and Banker's Algorithm

Tape drivers → Resources

Prous	R_1	R_2	R_3	R_4	
A	3	0	1	1	→ Resources Assigned.
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	1	

A	1	1	0	0	→ Resources Still needed
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

3,4

	R_1	R_2	R_3	R_4	
E' (	6	3	4	2)	Vectors
P (	5	3	2	2)	
A' (	1	0	2	0)	A' = (E' − P)

A' → Available after allocation

E' → Total available

P → Total allocated / assigned

→ Above configuration is called a State of system.

So, we have to provide A' vector to a process in such a way all the process get completed in some order. If such order exist then it is in safe state otherwise unsafe.

We have A' = 1 0 2 0

Needed					
A =	1	1	0	0	✗
B =	0	1	1	2	✗
C =	3	1	0	0	✗
D =	0	0	1	0	✓

So A' can satisfy the need of process D and allow the required resources to grant D. And after completion D will release all its resources So, new A' vector becomes

$$A' = 1\ 0\ 2\ 0$$
$$+\ D = 1\ 1\ 0\ 1 \text{ (assigned)}$$
$$(A')_{new} = 2\ 1\ 2\ 1$$

Again we do the same procedure for all process other than D.

$$A' = 2\ 1\ 2\ 1$$
$$+\ A = 3\ 0\ 1\ 1 \text{ (assigned)}$$
$$5\ 1\ 3\ 2$$

Other than A we could also satisfy the need of process E. So we can take process in any order.

$$A' = 5\ 1\ 3\ 2$$
$$B = 0\ 1\ 0\ 0$$
$$5\ 2\ 3\ 2$$

$$A' = 5\ 2\ 3\ 2$$
$$C = 1\ 1\ 0\ 0$$
$$6\ 3\ 4\ 2$$

and finally satisfy E.

$$A' = 6\ 3\ 4\ 2$$
$$E = 0\ 0\ 0\ 0$$
$$6\ 3\ 4\ 2$$

$(A')_{final}$ should be equals to $(E')_{initial}$.

We allot the the resources in order

D A B C E → (safe sequence)

Consider the previous both state tables and assume that B demands for resources as B(0 0 1 0) means B needed (0 1 1 2) but at that instant demand is only for (R_3). then OS acts as if it alloted R_3 to B and then check the system is in safe state or not if system will not in safe state then it take back the resources or doesn't allot it.

→ B = (0 0 1 0) (needed)

then, some changes occur.

assigned $(B)_{table} = (0\ 1\ 1\ 0)$

$(P)_{new} = (5\ 3\ 3\ 2)$

$(B)_{still\ needy} = (0\ 1\ 0\ 2)$

$(A')_{new} = (6\ 3\ 4\ 2)$

Now try to find the safe sequence with A' = (1 0 1 0)

A' = 1 0 1 0
D = 1 1 0 1
$(A')_{N_1}$ = 2 1 1 1
A = 3 0 1 1
$(A')_{N_2}$ = 5 1 2 2
B = 0 1 1 0
$(A')_{N_3}$ = 5 2 3 2
C = 1 1 1 0
$(A')_{N_4}$ = 6 3 4 2
E = 0 0 0 0
$(A')_{N_5}$ = (E') = 6 3 4 2

So, Safe sequence = D A B C E →
and we will allot (0 0 1 0) to B_0

And, if we are not obtaining the safe state then we will not allow.

→ Complete process we discuss above is banker's Algo. or deadlock avoidance algo.

→ Proposed by Dijkstra.

Gate-2007

Safe Sequence
$(P_1 \rightarrow P_0 \rightarrow P_2)$

Total = (5 5 5)
Allocated = (5 4 3)
Available = (0 1 2)

To do → Do 5 question on Banker's Algo. during revision other than practice (other than GATE)

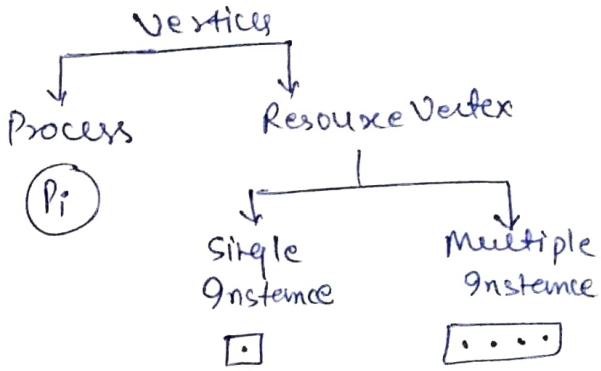
→ If needed matrix is not given find it first using (max − allocation)

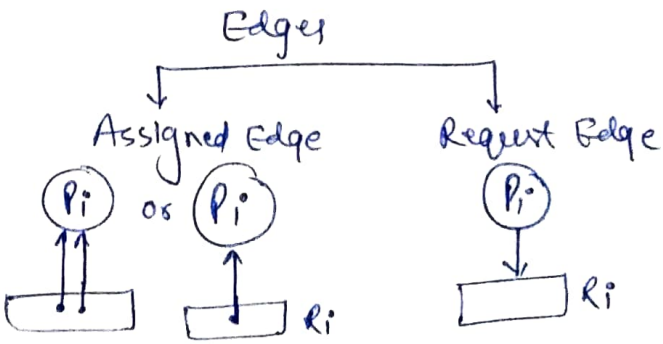
Resource Allocation Graph

→ we can identify deadlock easily, just by looking cycle in graph.

→ Meanwhile table is better than graph in case of large no. of process and resources

In graph



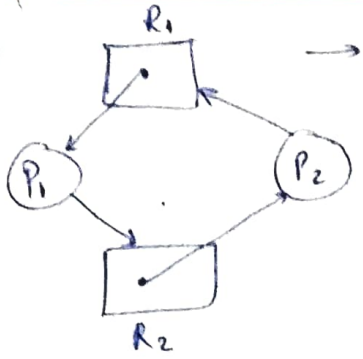


→ Using graph we can identify the state of system.

Examples

Cont --

①



→ single instance resource

P_1 already have R_1 and waiting for R_2, on the same P_2 already have R_2 and waiting for R_1. so deadlock.

Imp sufficient condition

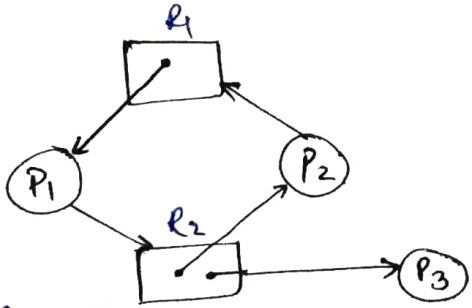
In single instance resource allocation graph if cycle formed then deadlock occur.

Graph to table

	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0
	↑ Allocation		↑ Request	

Available = (0 0) bcoz of single instance. so we can't satisfy any request so, deadlock here.

②



	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0
P_3	0	1	0	0
	(Allocation)		(Resources)	

Availability = (0 0) so process P_3 will satisfy

(New Avail) = 0 0
P_3 = 0 1
(0 1)

Now satisfy P_1 (0,1)

(New Avail) = 0 1
1 0
(1 1)

Now satisfy P_2 (1 0)

So, safe sequence → P_3 P_1 P_2

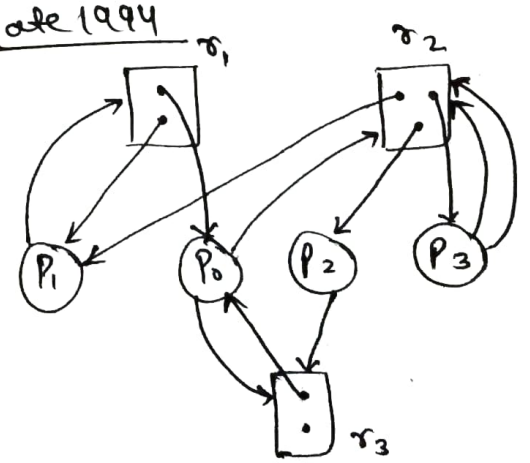
System is safe with no deadlock.

In multi resource graph 'cycle is not a sufficient condition' for deadlock.

But for consider the deadlock cycle must be there.

So, cycle is necessary but not sufficient for deadlock in multi instance resource graph.

Gate 1994



→ Deadlock or not.

	r_1	r_2	r_3	r_1	r_2	r_3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	0	2	0

Available (0 0 1)

first satisfy P_2 requires (001) and after completion of P_2

Available = 0 0 1
P_2 = 0 1 0
0 1 1

Now satisfy P_0 (0 1 1)

Available = 0 1 1
P_0 = 1 0 1
1 1 2

Now satisfy P_1 (1 0 0)

Available 1 1 2
1 1 0
2 2 2

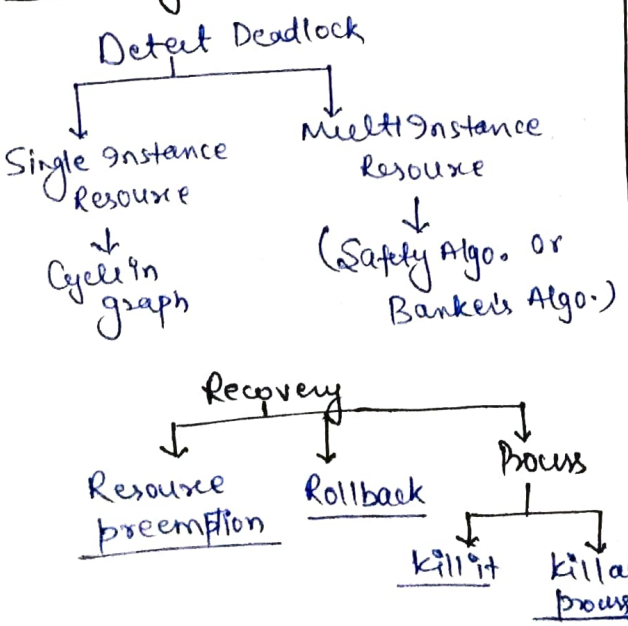
Now satisfy P_3 (0, 2, 0) and final available

→ 2 2 2
P_3 0 1 0
Available → 2 3 2

Safe sequence → P_2 P_0 P_1 P_3

So no deadlock is here.

→ Deadlock detection and recovery



for rollback we need checkpoint to get the previous safe state. If we get deadlock then we get back to previous safe state using checkpoints.

→ In rollback and resource preemption we deals with resources.

→ (kill it) means kill only one process causing deadlock or part of cycle. Also called as 'select a victim'

Memory Management (Tough)

Note → i) Most important
ii) 100% sure
iii) 3-4 Marks.

Need for multiprogramming & m/m management

CPU can directly access -

Very Costly
1. Registers
2. Main memory (RAM)
3. Cache (if present)

Larger main m/m → more process → more efficiency

But, there is a problem if main memory have only 1 process at a time.

Assume, size of process = 4MB
main m/m = 4MB

So, only one process can run at a time.