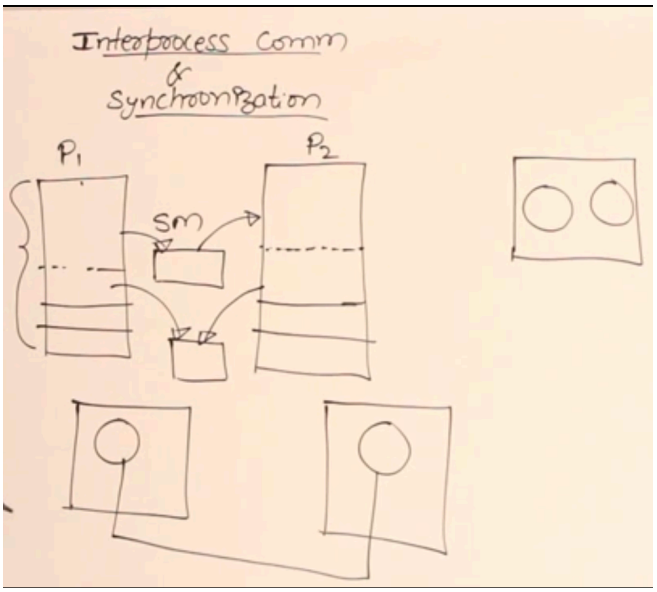
Lecture #1: Need for synchronisation

INTERPROCESS COMMUNICATION AND SYNCHRONISATION

- Process protection: a feature of an OS which ensures that one process's memory space is protected by other processes i.e. a process cannot manipulate the memory space of any other process
- If an operating system provides you with process protection, then there is no problem i.e. when all the processes are running independently
- But there can be a problem when two processes are trying to communicate with each other
- Processes can communicate with each other when
  i) they are in different computer/host
  ii) they are in same computer/host
- When they are in different computer/host, they will use protocol stacks like ISOSI, TCPIP for this interprocess communication
- In OS, we will not be using different computers. Communication will happen between processes when they are in the same host/computer
- There are two reasons for processes in the same computer to communicate with each other
  - 1. They want to cooperate: for example, if we have process p1 and p2, then the completion/ output of p2 will depend on p1's output
  - 2. They want to compete: for example, if p1 and p2 are trying to simultaneously access the printer
- So whenever processes want to communicate or compete, they need to communicate and whenever they communicate they are going to do it through ***shared memory (SM)***
- As long as the processes are not running parallelly, there wouldn't be any problem. But if they are, for example: p1 is writing the output in the shared memory and simultaneously p2 is trying to read it there could be some problem. Alternatively, if p2 is writing the data instead of reading it, then it could overwrite some data written by p1 and there could occur a lose of data
- Similarly, if both of them are trying to write something to the printer at the same time, then there could be a problem
- So whenever processes compete or cooperate, they need to communicate, and they do so by shared memory, and whenever this happens, there could arise data inconsistencies or other



errors

- Example: suppose we have two processes p1 and p2 which trying to do different tasks
- Somewhere between p1's code, it is trying to increment a variable named **count** to **count+1** whose initial value is 5
- Similarly, somewhere between p2's code it is trying to decrement the same variable **count** to **count -1**

- The entire program of p1 & p2 won't access shared memory, only a part of the program will access the shared memory and that part of the program is called the ***critical section*** of the program
- The count++ operation is implemented into assembly language code as follows:

1. mov count, R_0 // move count to register R_0
2. Inc R_0 // increment register R_0
3. mov R_0, count // move register R_0 to count

- Similarly, the count-- operation is implemented in assembly language as follows:

1. mov count, R_1 // move count to register R_1
2. Dec R_1 // Decrement register R_1
3. mov R_1, count // move register R_1 to count

- the register above can also be R_0, but for the sake of generality, we're assuming it's some other register R_1
- When we have such a sequence of operations, things can go wrong when we have a scheduling algorithm implemented in preemptive mode
- Let $p_{1,\,i}$ be the line number i of p_1 in the assembly code written above
- $\therefore$ $p_{1,\,1}$ will be "mov count, R_0", $p_{1,\,2}$ will be "Inc R_0" and so on
- Similarly, let $p_{2,\,i}$ also be the line number i of p_2 assembly code written above
- Let us denote preemption of a process by the symbol "|", execution of next instruction as "→" and execution of next process (and completion of current process) as "⇒"
- Now, if the following scheduling of process takes place

$p_{1,\,1} \rightarrow p_{1,\,2} \rightarrow p_{1,\,3} \Rightarrow p_{2,\,1} \rightarrow p_{2,\,2} \rightarrow p_{2,\,3}$ // count = 5

- then count = 5, and everything is consistent i.e. p_2 is scheduled only and only after complete execution of p_1 and vice-versa (No preemption case)
- But if p_1 gets preempted in the middle of its execution of critical section and follows the following sequence of scheduling

$p_{1,\,1} \rightarrow p_{1,\,2} \mid\ p_{2,\,1} \rightarrow p_{2,\,2} \rightarrow p_{2,\,3} \Rightarrow p_{1,3}$ // count = 6

Then there is a problem, because when p_2 is preempted, only register R_0 has value equal to 6, and count still has not been updated by instruction $p_{1,\,3}$

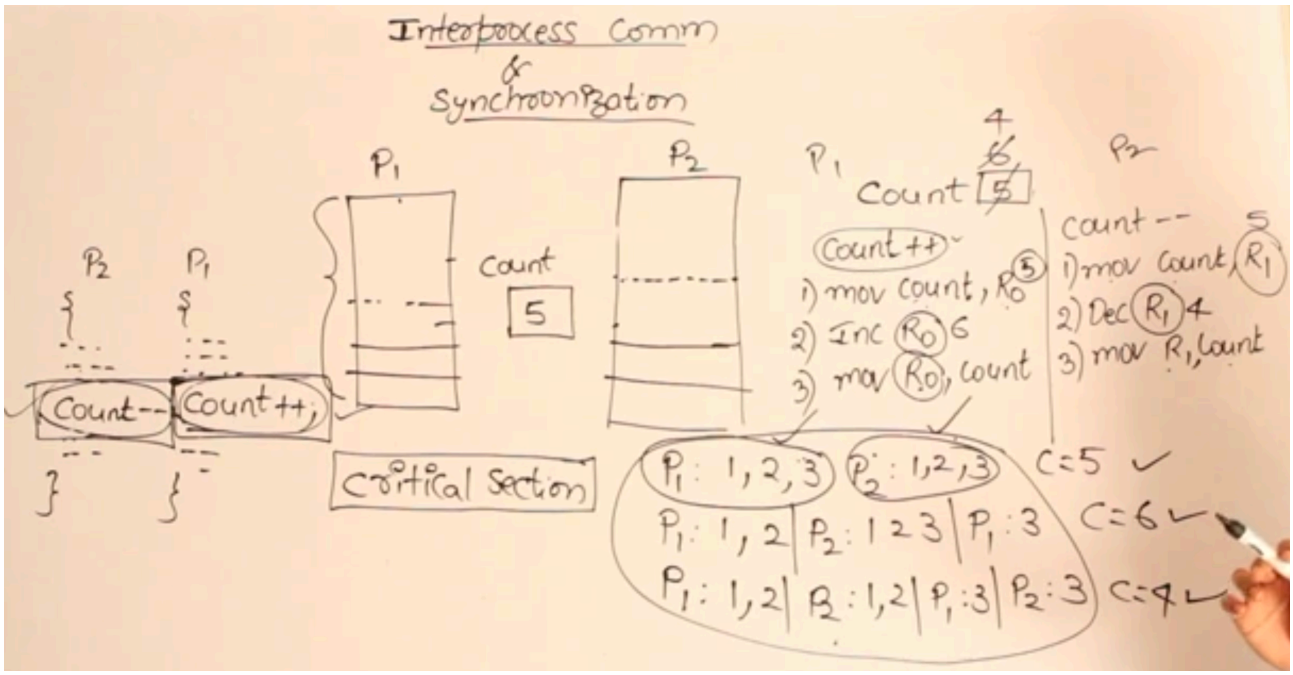
- $\therefore$ when p_2 gets completely executed while taking the value of count as 5, and at the end of p_2's execution, count takes value 4
- After the full execution of p_2, p_1 again gets scheduled and executes $p_{1,\,3}$, which moves the value of R_0 which was 6 to count which was recently updated to value 4
- $\therefore$ **at the end of p_1 & p_2's execution, the value of count becomes 6**, which is inconsistent as the value of count should remain 5 after incrementing and decrementing it
- We get inconsistent with the following sequence of scheduling also:

$p_{1,\,1} \rightarrow p_{1,\,2} \mid\ p_{2,\,1} \rightarrow p_{2,\,2} \mid p_{1,3} \Rightarrow p_{2,\,3}$ // count = 4

As in the end we would get the value of count as 4

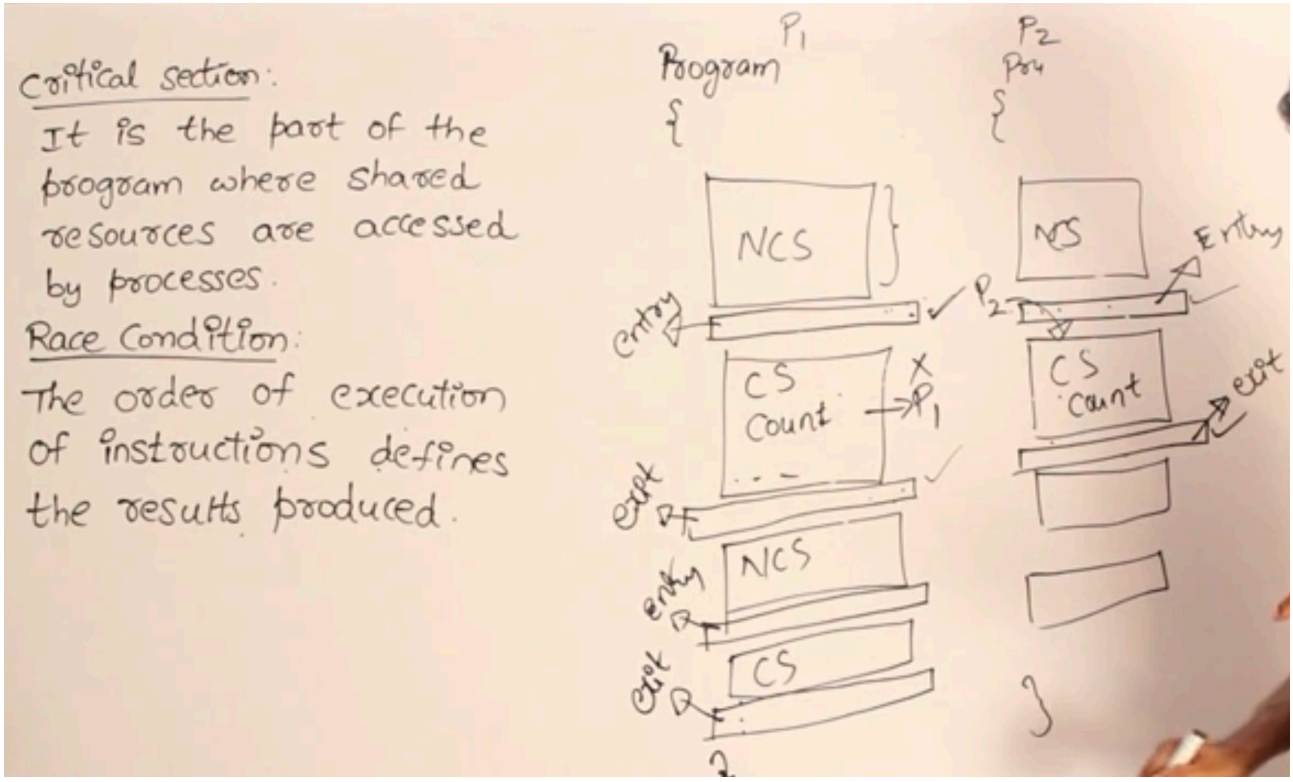
- $\therefore$ we can get the value of count as 4, 5 or 6 depending on the order of execution of the process
- This is called ***Race Condition***, i.e. the final output depends on the order in which the processes are being executed and preempted
- The part of the code where the processes are using a common resource or memory is called ***critical section***
- For example:

    - In process p_1 & p_2, the part where the two programs increment and decrement the value of count would be the critical section
- The term “Race” comes from the following thought: The output is depending on which process finishes its execution first, implying a notion of race between the process for output
- The solution to this problem is to have some kind of synchronisation between processes such that only a single process accesses the critical section at a time, without letting any other processes access the same critical section, so that no race condition arises



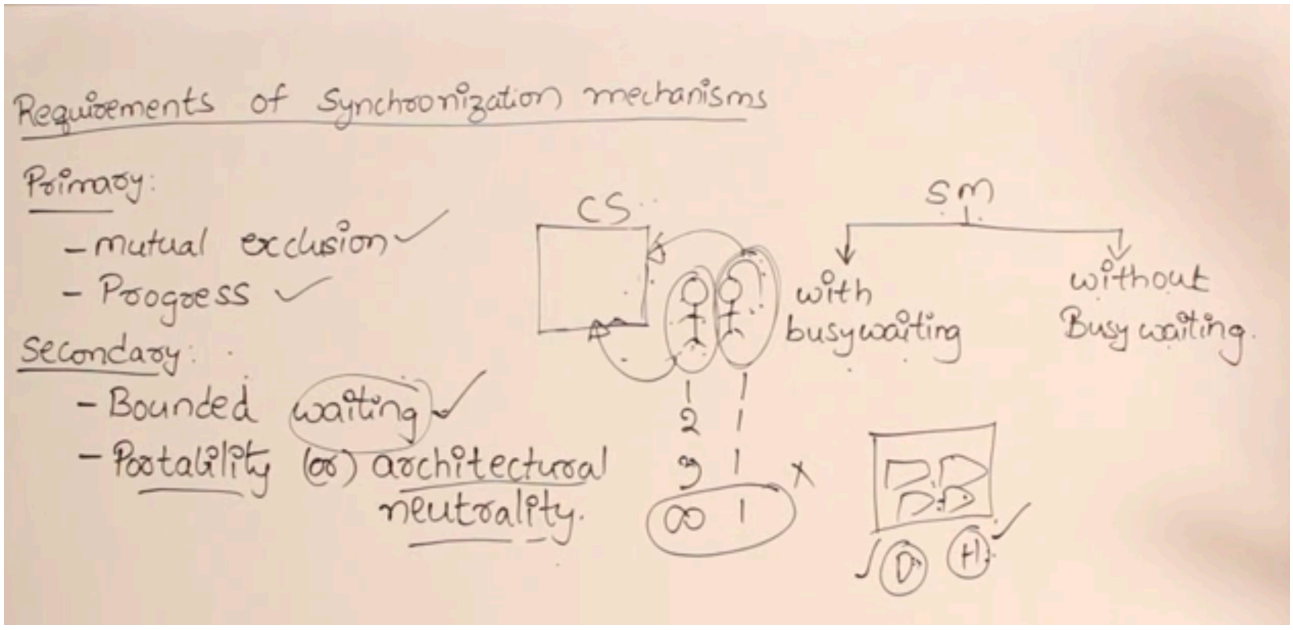
Lecture #2: Introduction to Synchronisation Mechanisms

- In order to avoid race conditions, synchronisation processes were devised
- Placing a small “Entry” code before entering the critical section and a small “Exit” code after exiting the critical sections, wherever they are in the program, which would help us in avoid race conditions to occur
- These sections of code are called ***Entry section*** and ***Exit sections*** respectively
- If placing these codes achieve the desired results, then they are called synchronisation mechanism
- There are various kinds of synchronisation mechanisms along with some requirements, so that they do not create more problems when we place them in our program
- Some problems that can arise when including the synchronisation mechanisms are
    - Wastage of CPU when including entry and exit sections
    - mechanism being unable to the race conditions to occur (called ***mutual exclusion***)
    - Inducing ***deadlocks*** into the program (Deadlock - all processes getting blocked due to entry sections forever)
- there are various synchronisation mechanisms have their own set of advantages and disadvantages
- Though synchronisation mechanism were invented to solve the race conditions, they have been extended in such a way that, we are even able to order the execution of processes i.e. we could use the synchronisation mechanisms to make the processes behave in a certain way

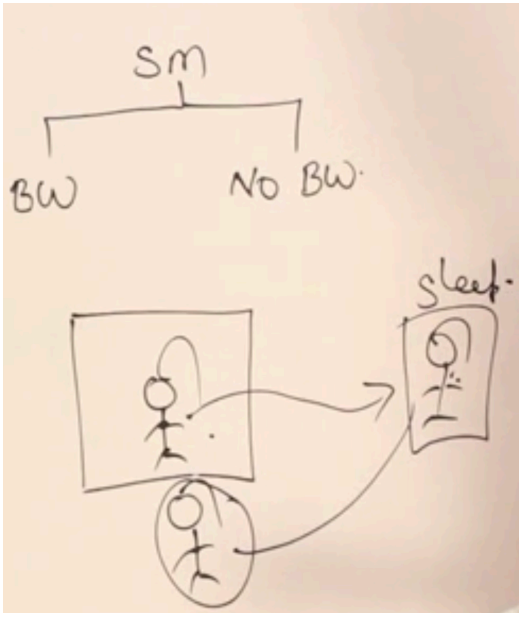


Lecture #3: Conditions for Synchronisation Mechanisms

- ***Mutual Exclusion***: If one process is in the critical section, then the other section should not be in the critical section at any time. This condition is called mutual exclusion.
- An analogy for critical section and processes is dressing room and people respectively i.e. we can think of critical sections as dressing room and processes as people
- In this analogy, mutual exclusion will be the condition that no two person should be allowed in the dressing room at the same time
- ***Progress***: If a process does not want to get into the critical section, then it should not stop other processes to get into the critical section
- In our analogy, this would be: A person not wanting to get into the dressing room should not stop other person to go into the dressing room
- These two conditions are primary conditions i.e. we will not consider to implement the synchronisation mechanism if it does not follow these two condition
- Other two conditions are secondary i.e. they are desirable but could be considered for implementation even without them
- These secondary conditions are
  - ***Bounded waiting***: A process should not wait for execution for long time or someone should not stop a person for a long time.
  - Bounded waiting therefore means being able to tell after how many times of execution of a particular process some other process will get a chance to be executed
  - If there is no bounded waiting, then there might be starvation of process
  - ***Portability*** or ***architectural neutrality***: We should not make any assumptions that our solution should only work on this particular OS or hardware
  - Sometimes it happens that our proposed solution is too dependent on a particular instruction provided by the OS or supported by the hardware, and implementation of the same solution on other OS or hardware becomes impossible due to this hardware or OS dependency of our solution. So we should try to keep our solutions architecturally neutral.
- Synchronisation mechanisms can be classified into two types, mechanisms:
  - With busywaiting
  - Without busywaiting



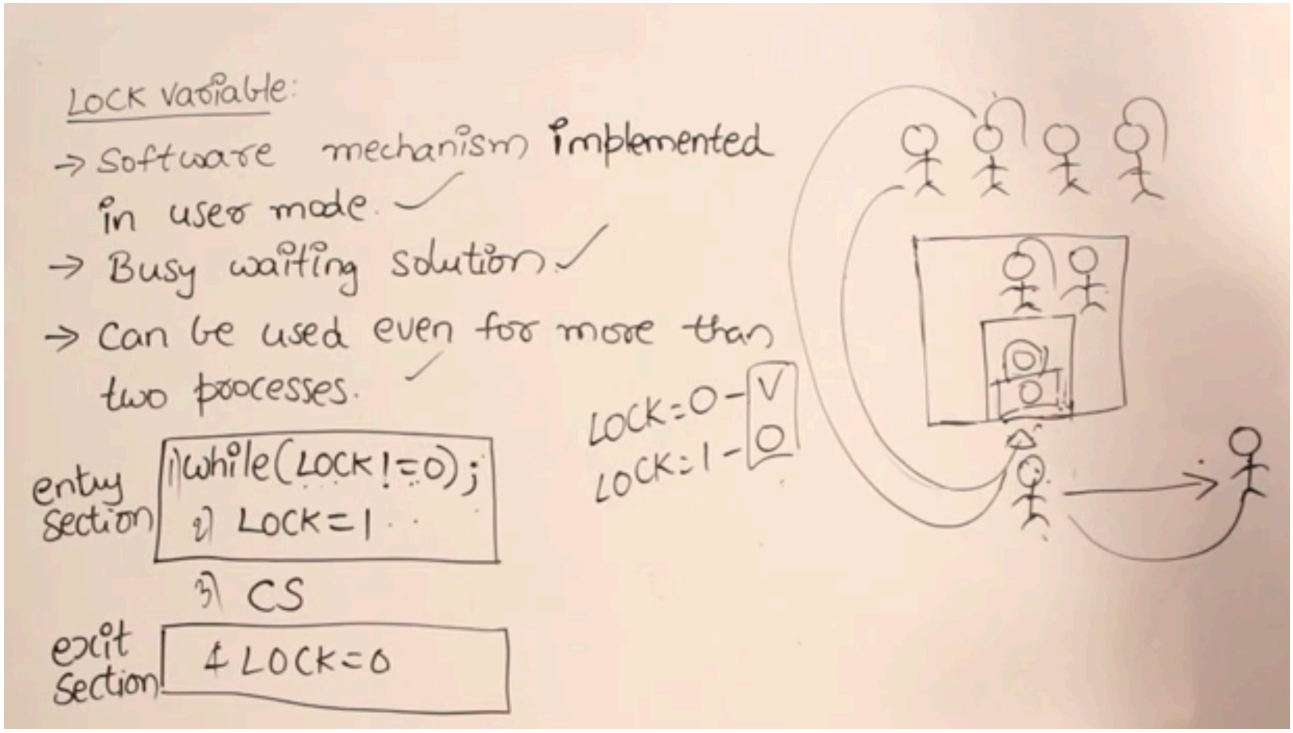
- Busywaiting - in our analogy of dressing room, if a person is waiting and someone else is already inside the room, then that person will be standing on the door and continuously keep knocking the door, asking whether the person is finished and is still inside
- This continuous knocking in terms of CPU is falling into an infinite loop, which would take a lot of CPU time, and once the process is over, it is scheduled
- Without busywaiting - The person will look once if the room is occupied and if it is, then the person goes to sleep. Then when the person in the dressing room is free, they will wake the person who is sleeping to go into the dressing room
- So the difference between these two is that, in busywaiting, a lot of CPU time will be wasted, while without it, we will save a lot of CPU time



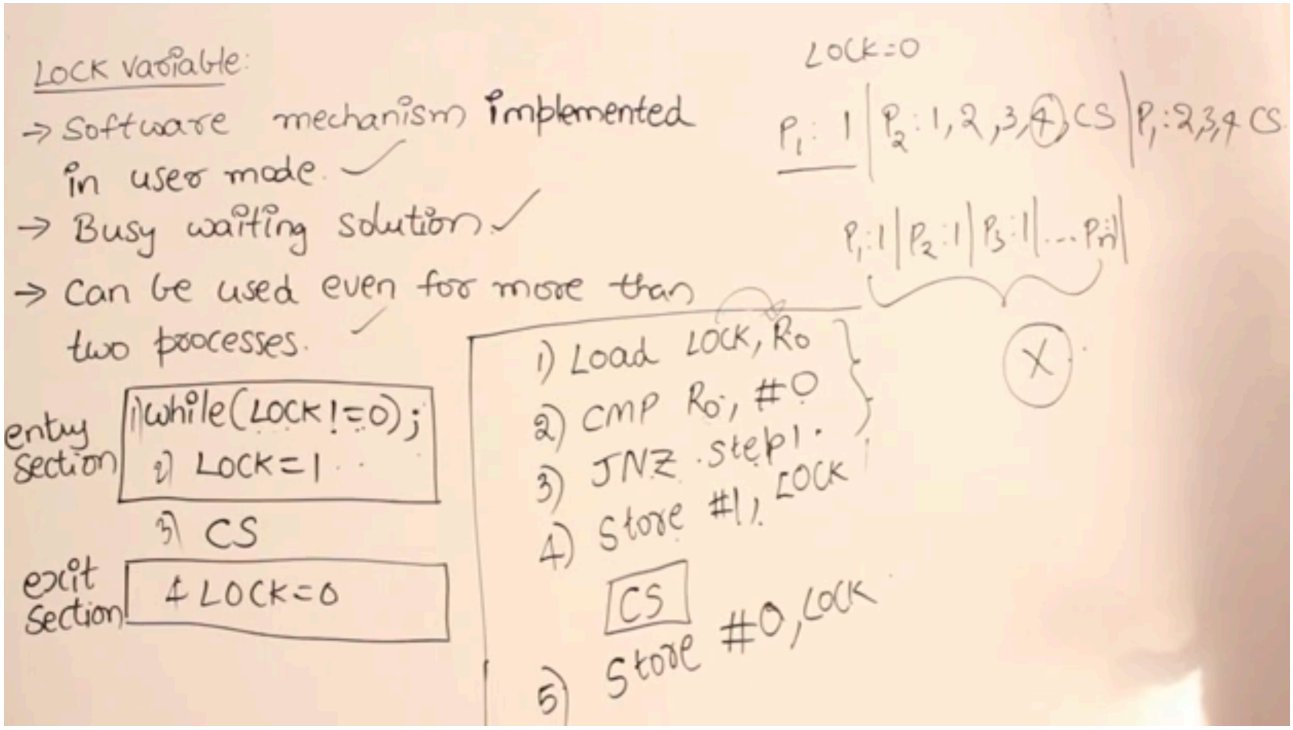
Lecture #4: Lock variables

- This is the simplest synchronisation mechanism
- Even though the name says lock, it is actually not lock
- It works like a “tag”
- Software mechanism; no hardware support needed; implemented in user mode
- There are two kinds of modes: user mode & kernel mode

- If we need any support from the operating system, we need to go the operating system code, which we call to be ***kernel mode***
- If we don't want/need any interference from the operating system, then that is called ***user mode***
- It is a busywaiting solution
- Many solutions work for only two processes, but this solution works for more than two processes
- It has entry section as follows: while(LOCK != 0);LOCK = 1;
- And its exit section is as follows: LOCK = 0;
- This has problems, because a process might get preempted when it has just 'seen'[1] the value of LOCK as 0, and when some other process that is scheduled will see the value of LOCK as 0, so it will go into the critical section, and during this time if it gets switched to the original process, it will not again check to see if LOCK is actually equal to 1 and directly enter the critical section (as it knows from before the value of LOCK is 0 before it was preempted), ultimately failing to provide mutual exclusion
- $\therefore$ we do not consider this mechanism as viable as it fails our primary condition of providing mutual exclusion
- This could be easily explained if we write the Entry section code in (pseudo) assembly language and try to preempted the processes in a certain way
- If we preempt all the processes just after they've 'seen' the value of LOCK as 0, then schedule all the processes one by one, then **all the processes could be in the critical section at once**, once again failing to provide mutual exclusion
- Some (pseudo) assembly language code/command :-
  - LOAD: loading variable content to register or vice-versa
  - CMP: comparing values in the register to some other value, return 1 if equal else 0
  - JNZ step X: Jump if Not Zero to step number X
  - STORE: store/load a value to some variable
- we initially initialise LOCK to 0 as no process is in the critical section in the beginning



[1] 'seen' or seeing here means loading the value of LOCK to register R_0



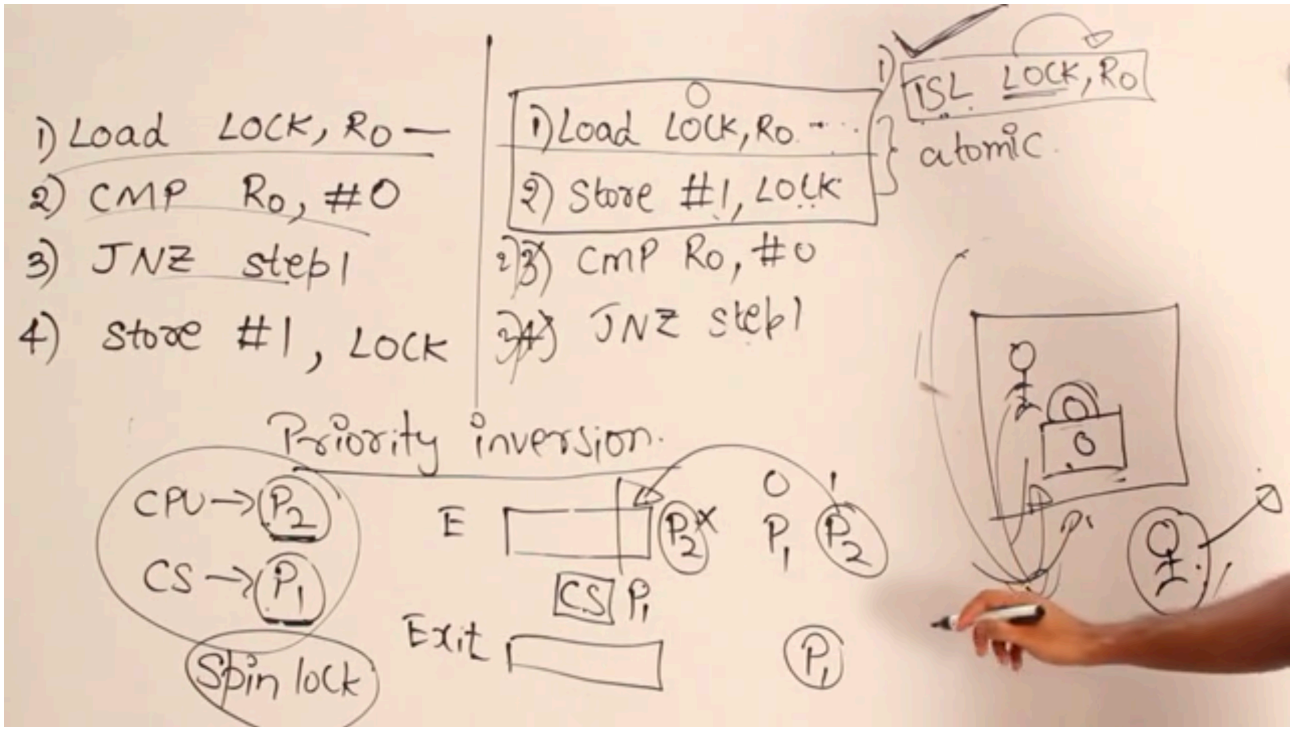
Lecture #5: TSL

- the entry section code in assembly language might look like this:

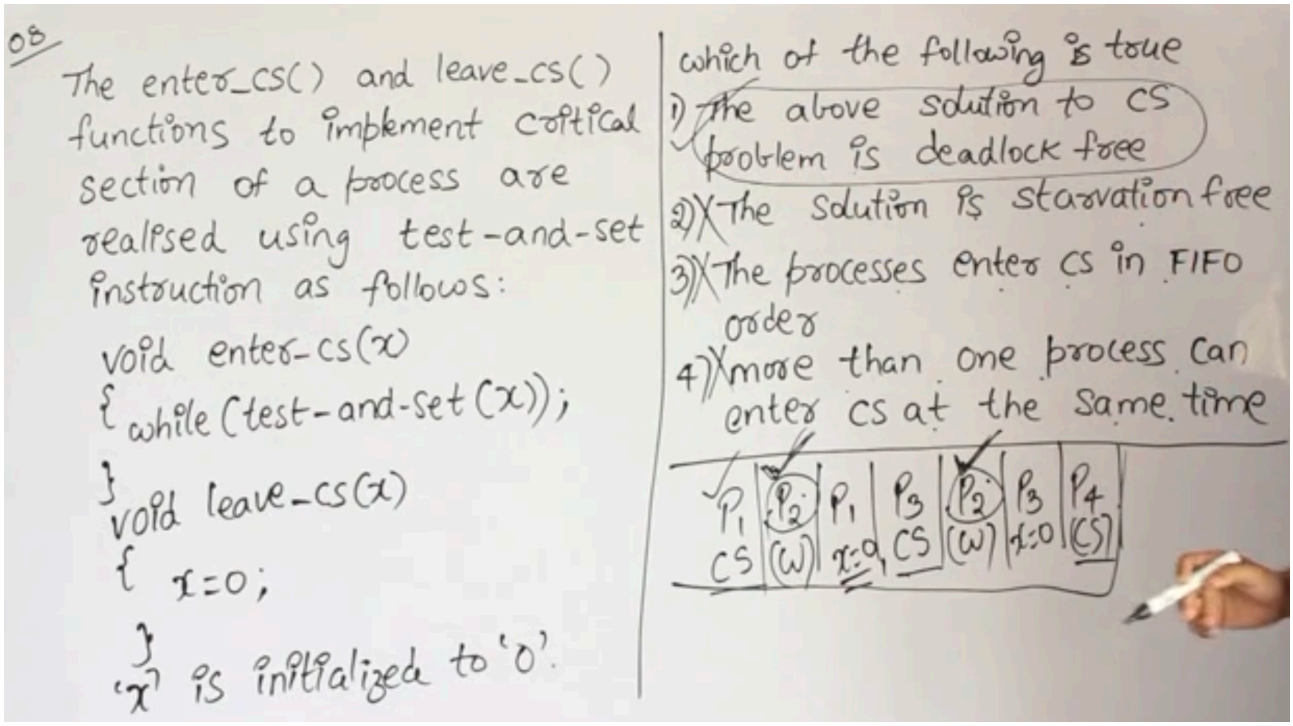
1. Load LOCK, R_0 //load value of LOCK to register R_0
2. CMP R_0, #0 //Compare value of R_0 to value 0
3. JNZ step 1 //Jump to step 1 if it is not equal to zero
4. Store #1, LOCK //store value 1 in LOCK

Critical section starts

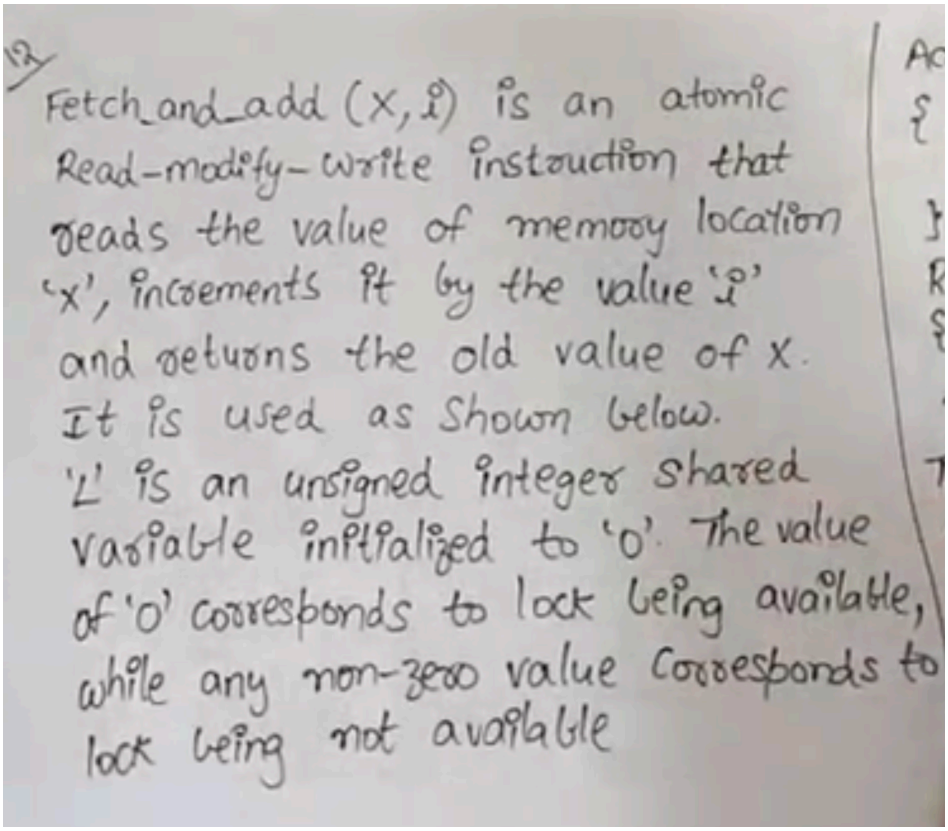
- Now, if any process gets preempted after executing step 1, 2 or 3



Lecture #6: Gate 2008 TSL question



Lecture #7: Gate 2012 TSL question



```
AcquireLock(L)
{ while (Fetch_and_Add(L,1))
      L=1;
}
ReleaseLock(L)
{
  L=0;
}
```

This implementation

a) Fails as 'L' overflows
b) Fails as 'L' can take non-zero value when lock is available
c) works correctly but starves
d) works correctly without starvation.

12/ Fetch_and_add (X, i) is an atomic

L=0

P_1	P_2	P_3	P_4	P_5	..... P_n
L=0	L=1	L=2	L=3	L=4	
L=1	L=2	L=3	L=4	L=5	L=n
CS					

00...0
000 1
000 10
111 11
000 00

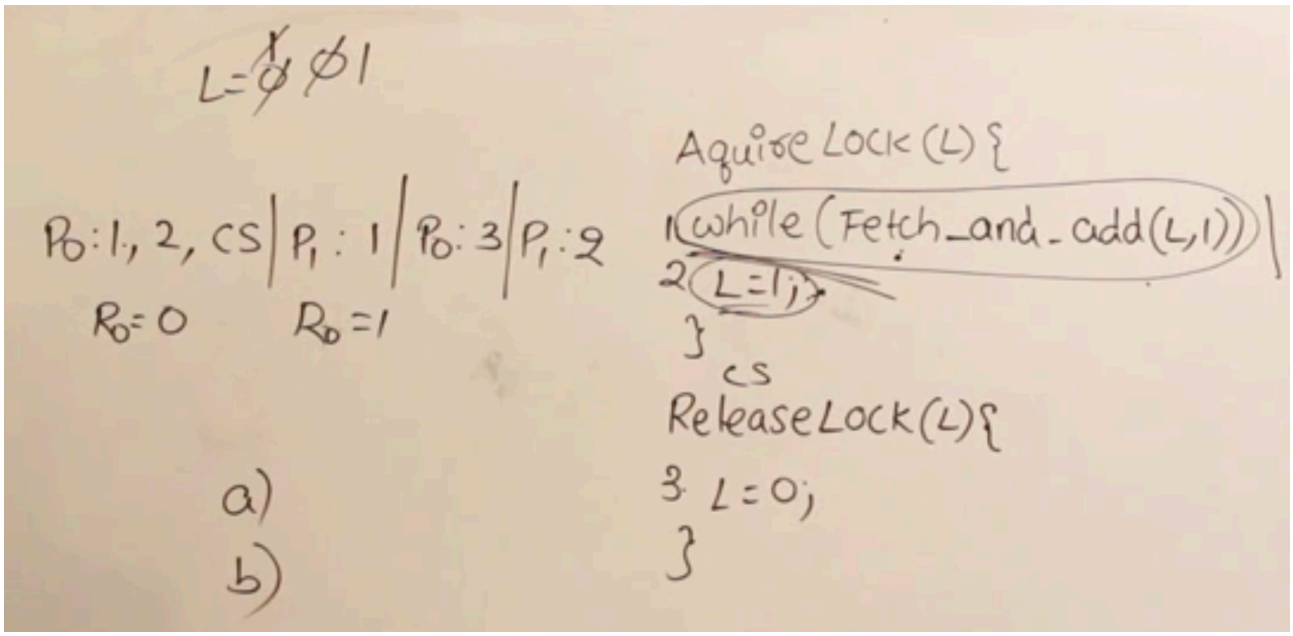
>1

```
AcquireLock(L)
{ while (Fetch_and_Add(L,1))   (1) -> (2)
      L=1;
}
ReleaseLock(L)
{
  L=0;
}
```

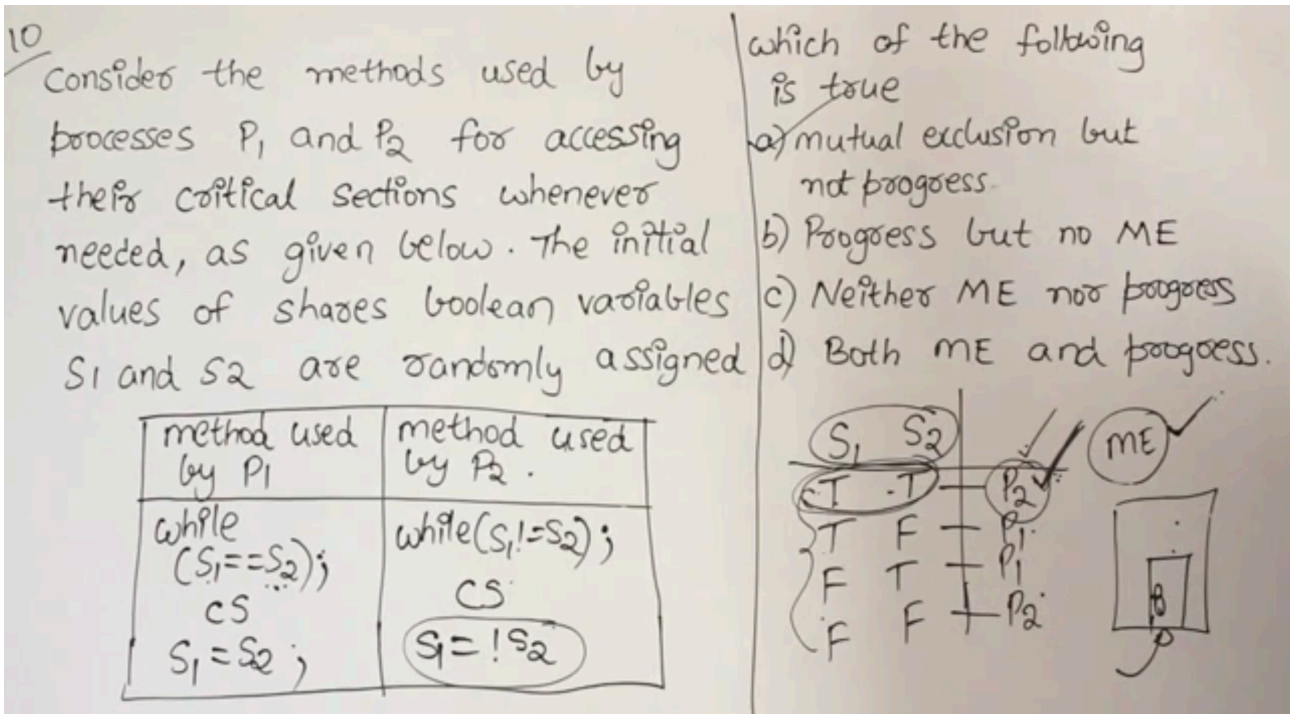
This implementation

a) Fails as 'L' overflows
b) Fails as 'L' can take non-zero value when lock is available
c) works correctly but starves
d) works correctly without starvation

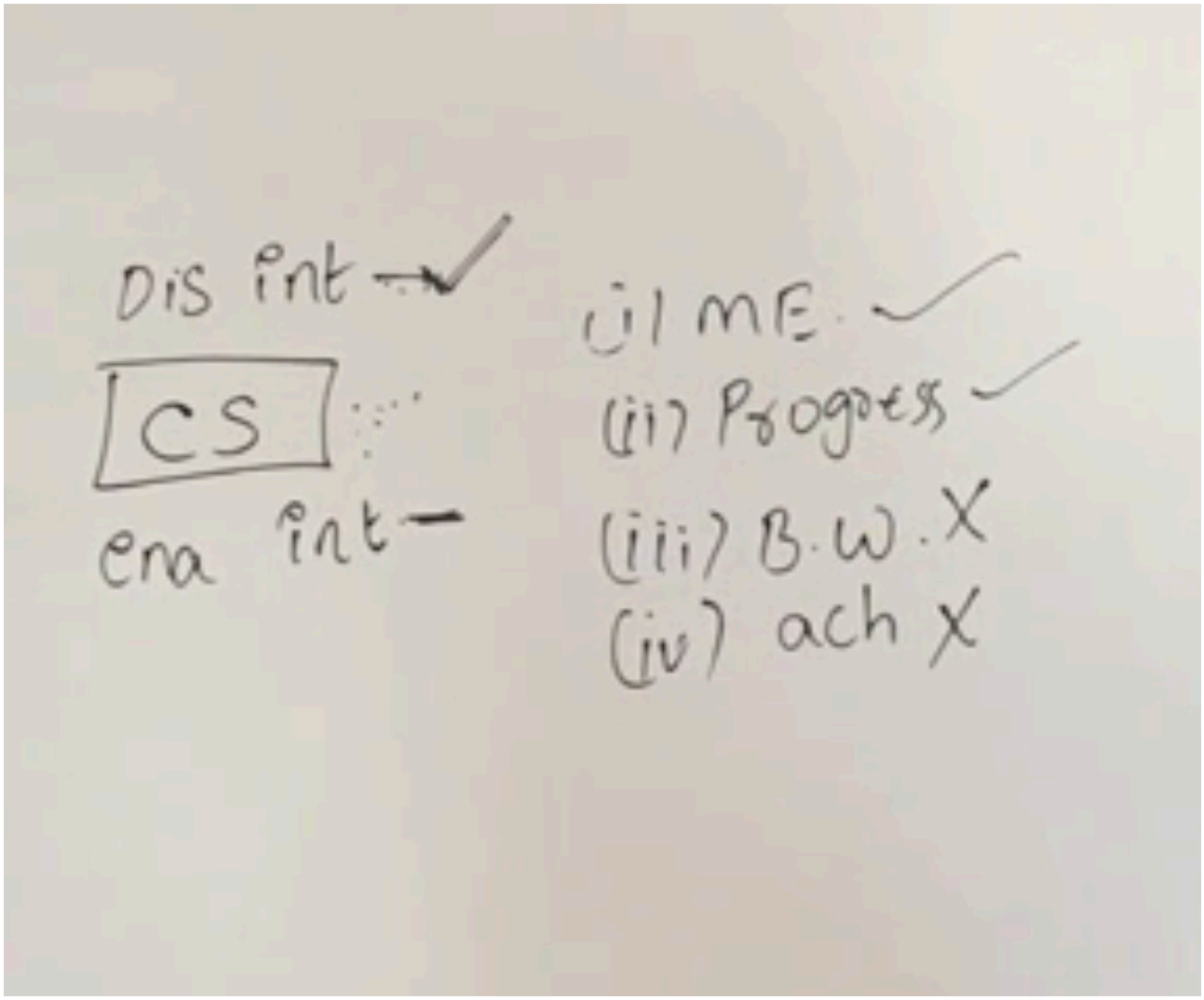
Lecture #8: Gate 2012 TSL Question Continuation



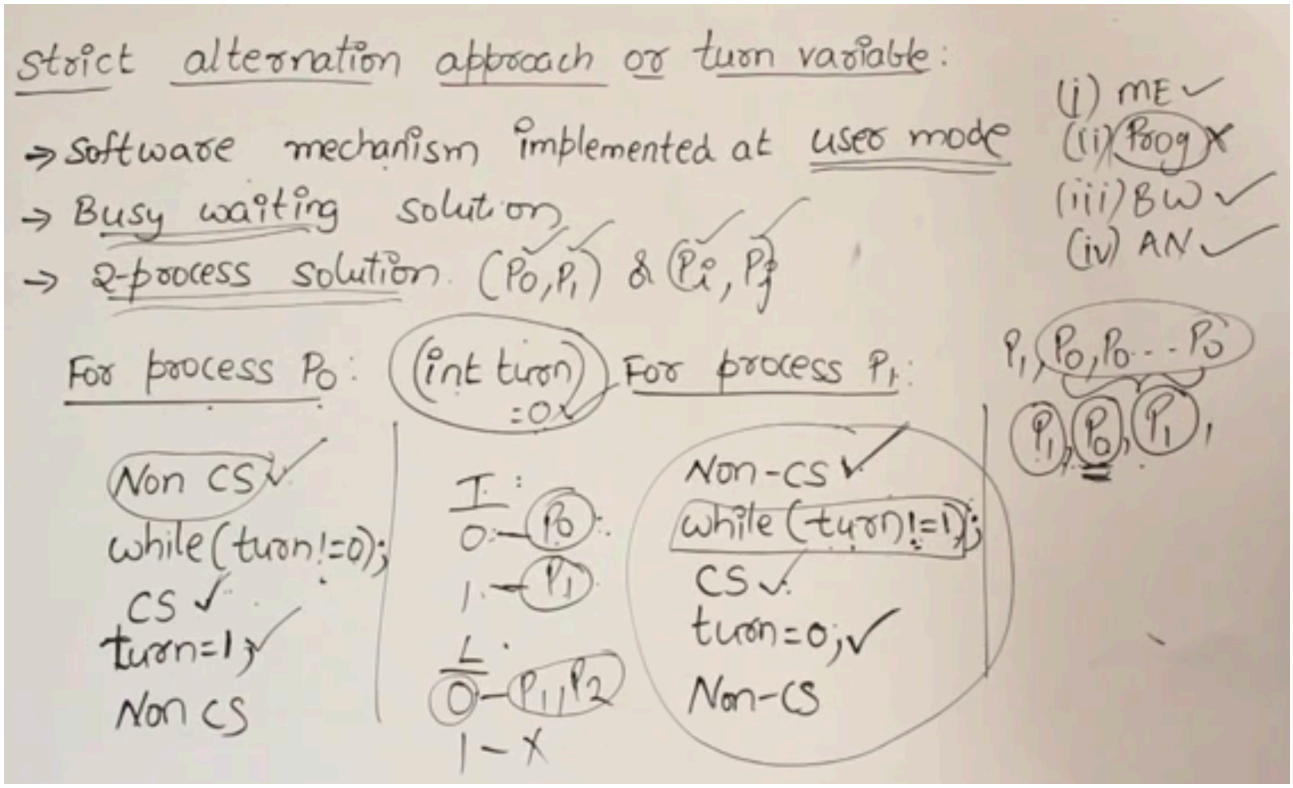
Lecture #9: Gate 2010 TSL question



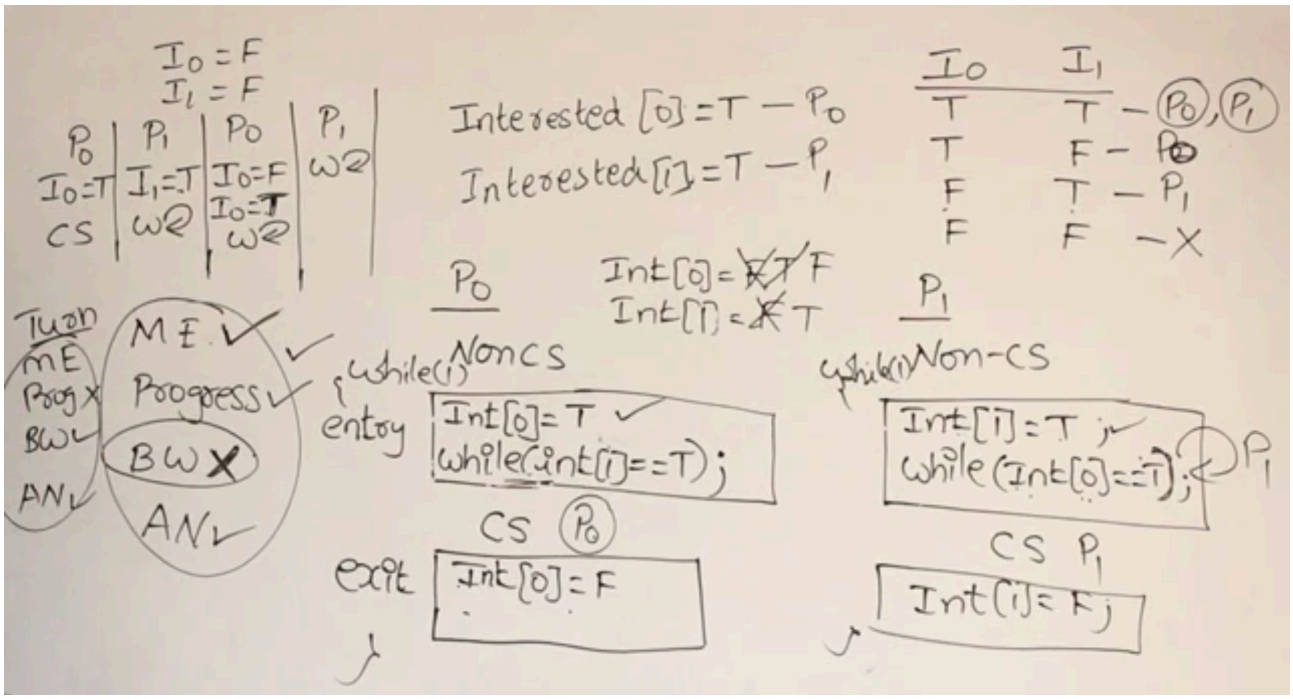
Lecture #10: Disabling interrupts



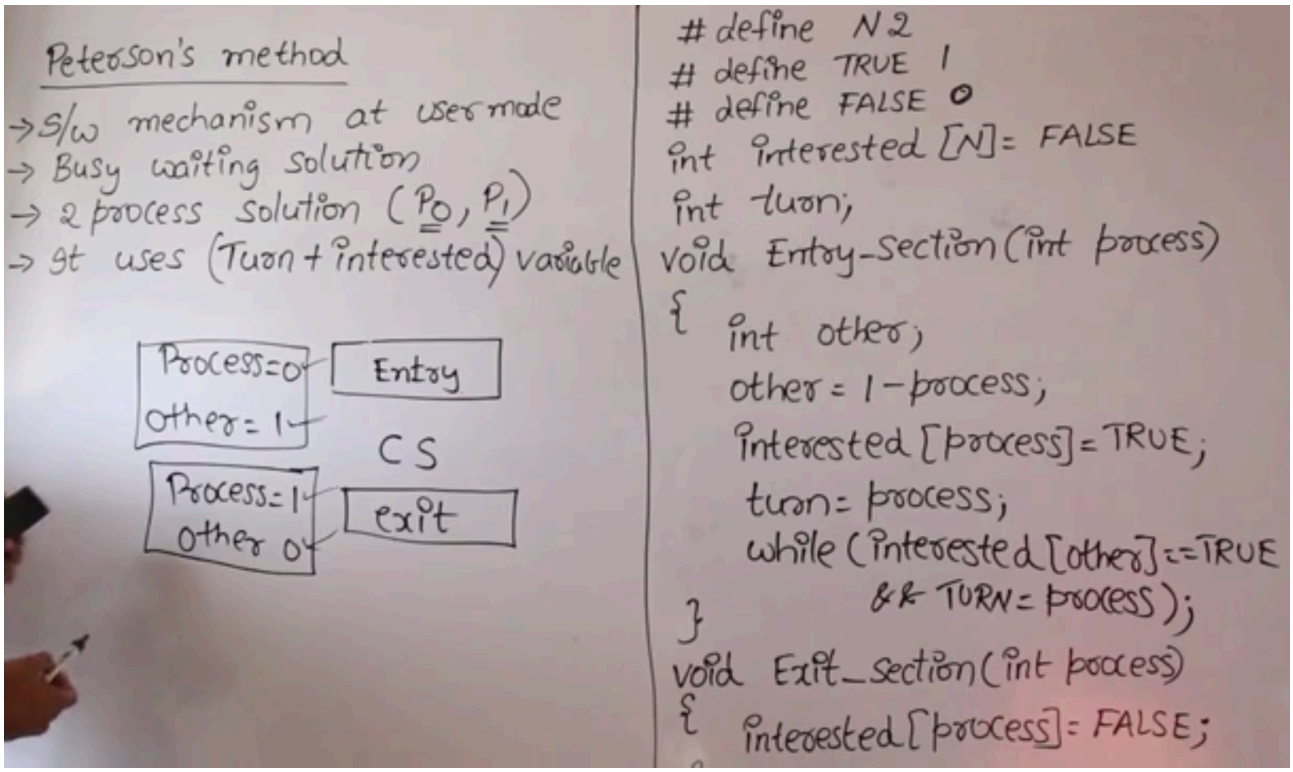
Lecture #11: Turn Variable or Strict Alteration Method



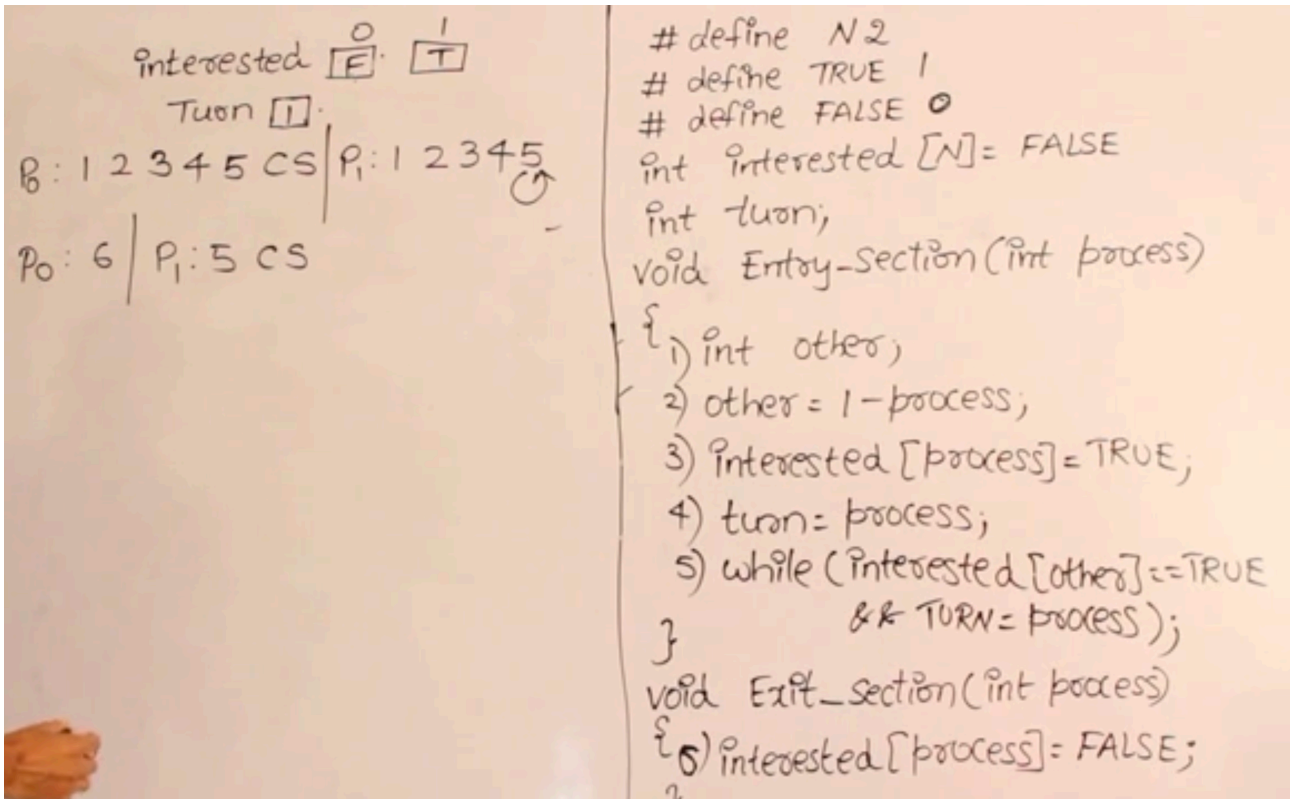
Lecture #12: Interested Variable

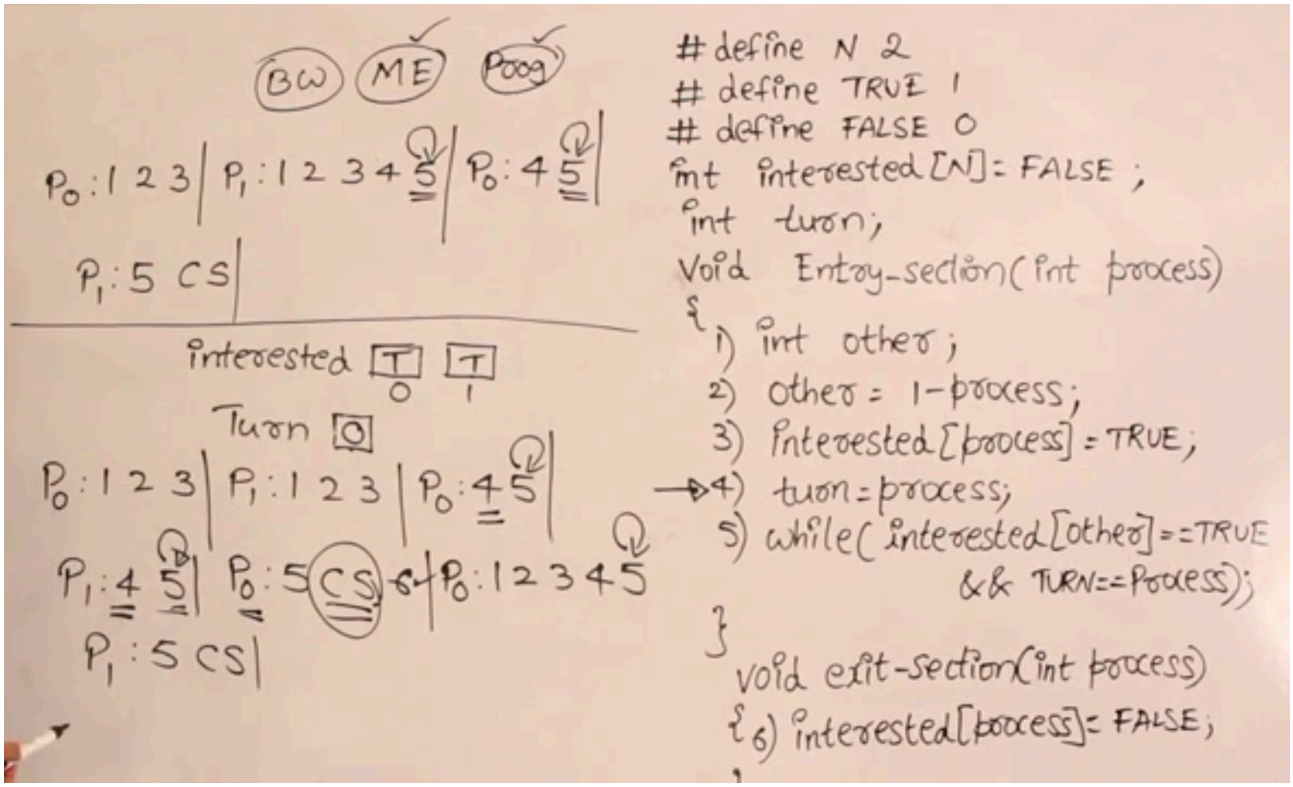


Lecture #13: Peterson's solution



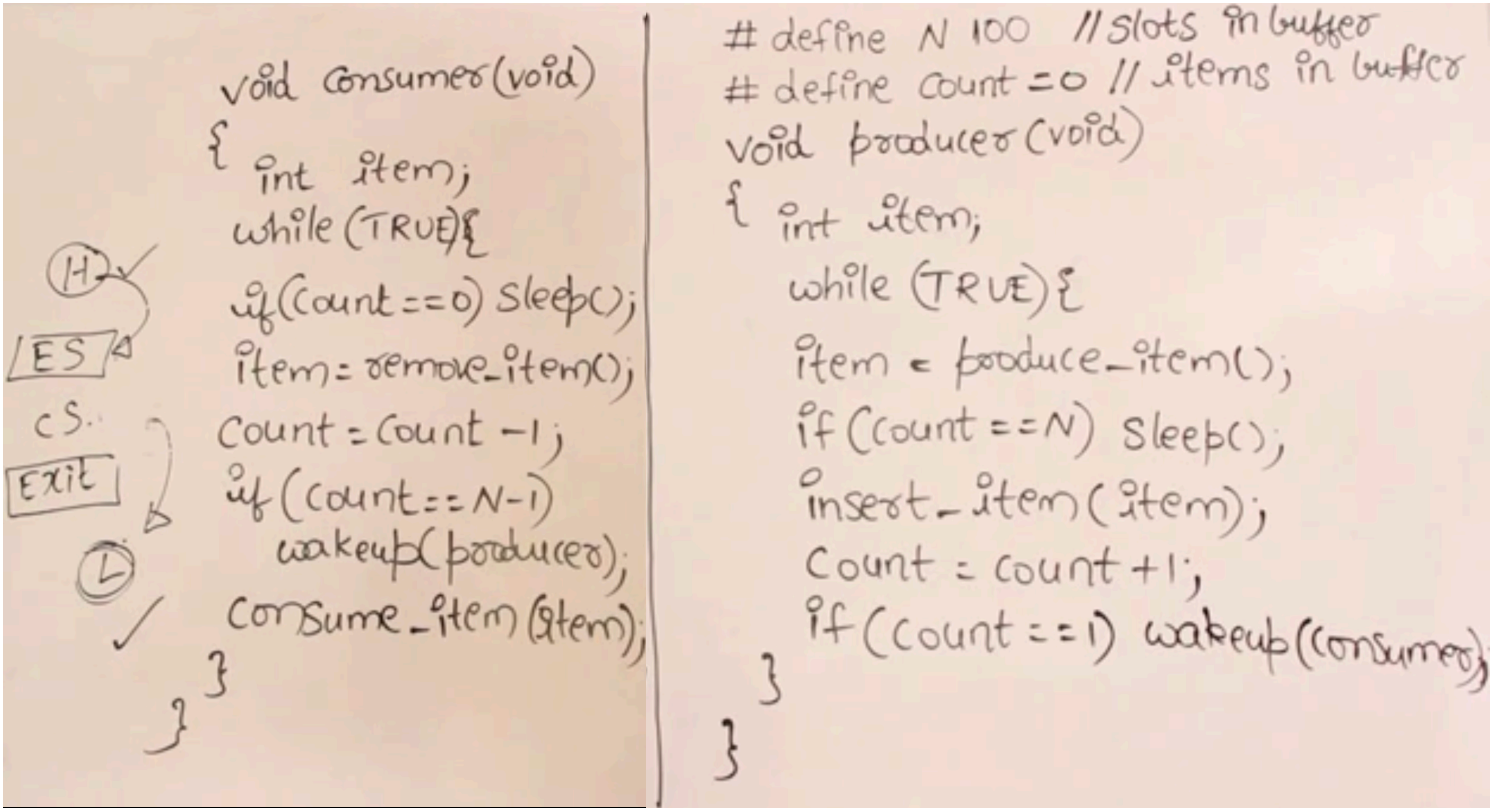
Lecture #14: Tracing Peterson's solution





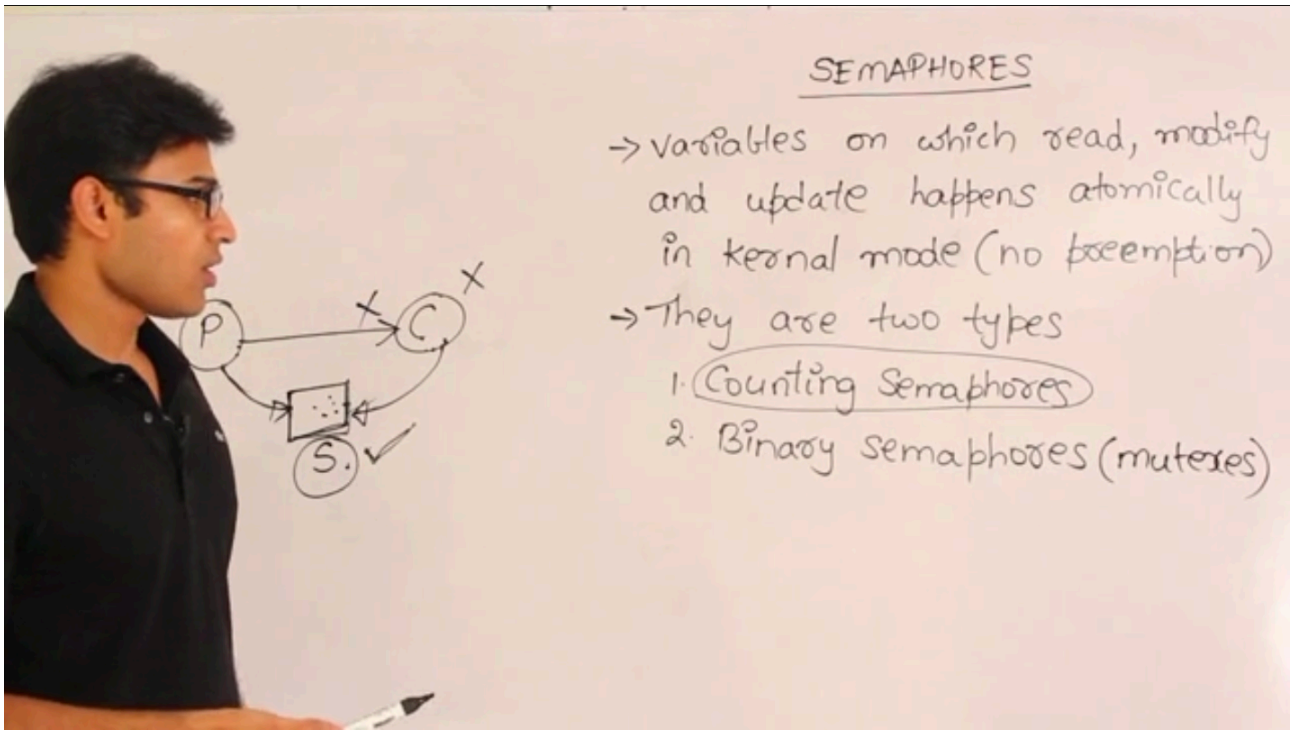
- All four conditions satisfied in Peterson's solution

Lecture #15 & #16: Synchronization Mechanisms Without Busy Waiting



- Till now, every solution that we have explored is busywaiting solutions, which suffers from two major disadvantages
  - CPU Time Wastage
  - Priority Inversion
- Sleep and wake-up mechanism is better as it does not suffer from the above two problems
- Using producer consumer problem, we will examine sleep and wake-up mechanism
- The process which writes is called writer or producer and the process which reads is called reader or consumer
- Both will use the shared memory called buffer where they read and write
- The code in the image might pose a scenario in which both producer and consumer eventually go to sleep when preempted at a certain time, and therefore it suffers from deadlock.
- In the analogy of rooms, when the producer tries to wake the consumer, he should check if he is in the room sleeping, and if he's not sleeping, he should put a tag outside of the door saying don't sleep so that when the consumer does come to sleep, he sees the tag of don't sleep and then doesn't sleep
- This solution of putting a tag outside the door can be implemented by maintaining a bit which records if the sleep call has been made by the producer or the consumer before starting
- But this solution is valid only for 1 producer and consumer. $\therefore$ for N producers and consumers we need to maintain an integer which records the number of sleep calls made by producers and consumers.
- This variable integer is called semaphore coined by Dijkstra in 1965
- Implementing semaphore requires support from OS
- All modern OS support semaphores

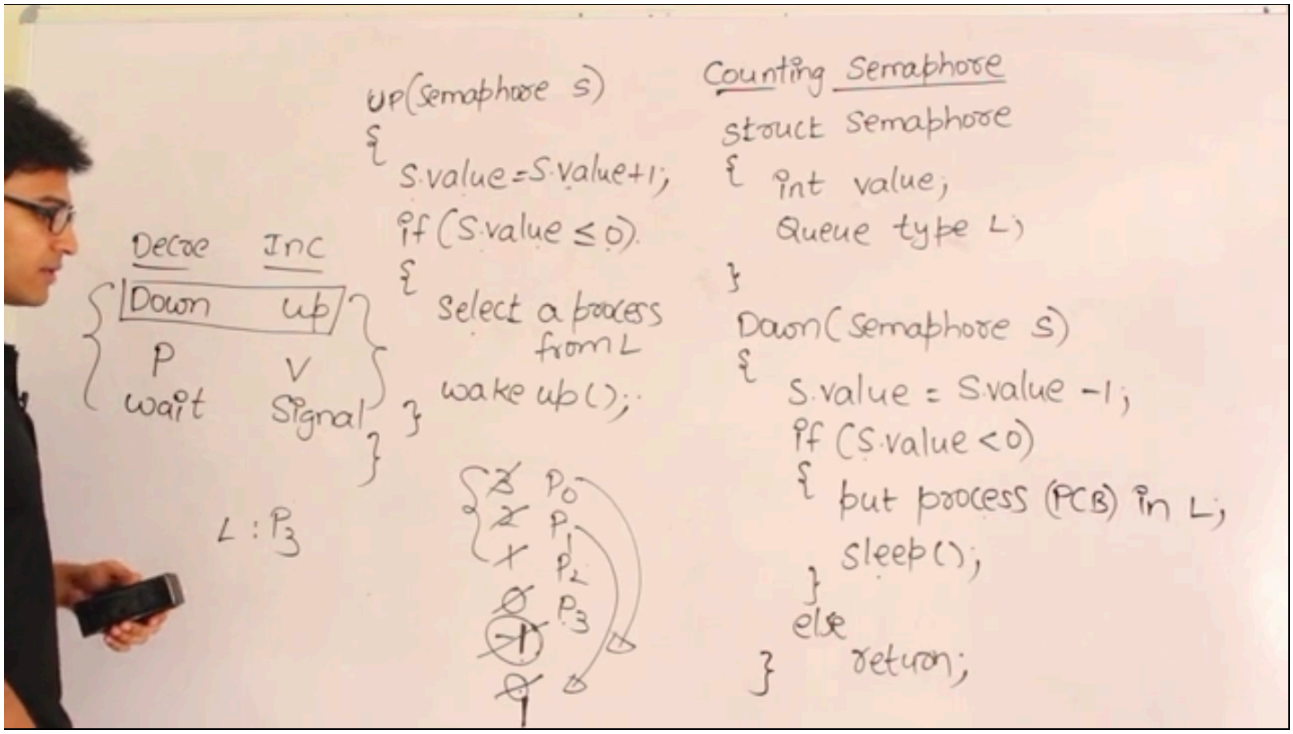
Lecture #17: Introduction to Semaphores



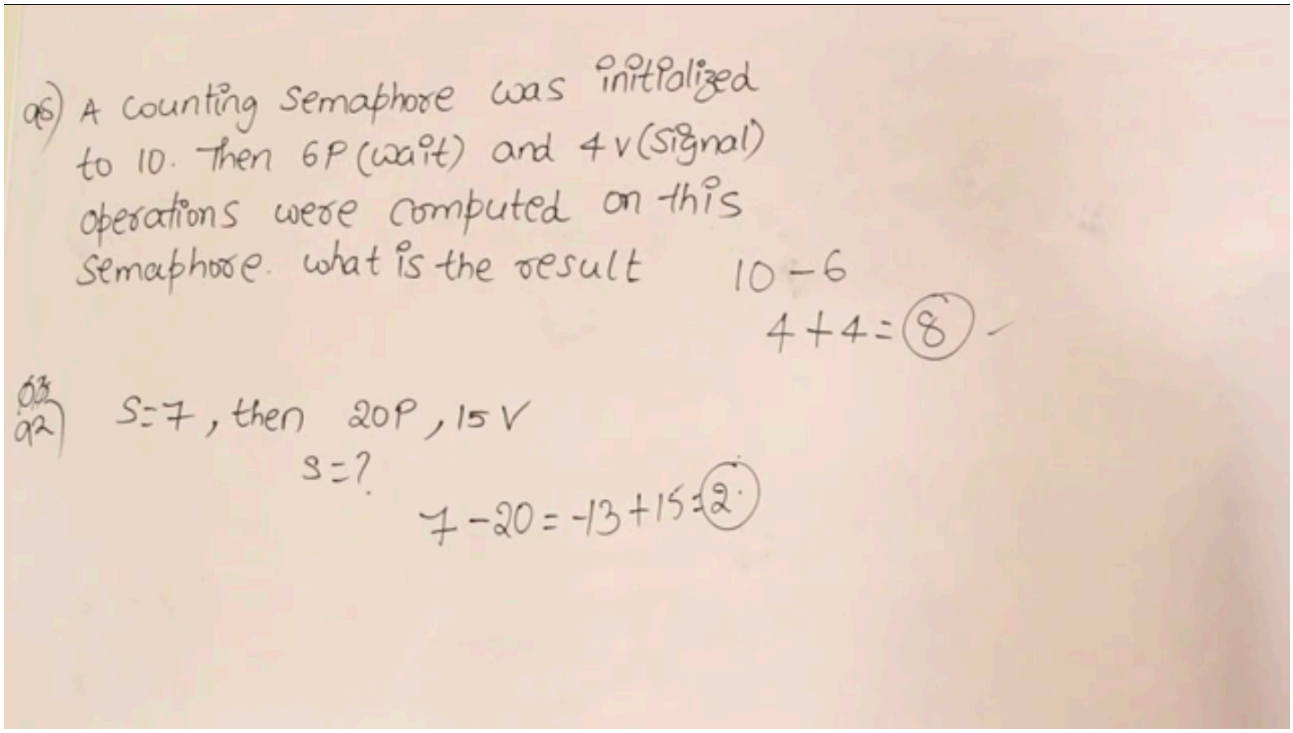
- Binary semaphore (Mutex) supports mutual exclusion
- Counting semaphore will set an upper bound on the number of processes to be entered into the critical section

Lecture #18: Counting semaphore

- positive values in counting semaphore (CoS) indicate how many processes can enter into the critical section
- Negative values indicates the number of sleeping processes
- 0 indicates no more processes can enter into the critical section
- When entering, we should decrement the CoS, and when leaving we should increment it
- This incrementing and decrementing should be atomically, which will be provided by the OS
- Setting CoS as 1 will provide us mutual exclusion and ∴ act as binary semaphore or mutex
- Implementing it using queue also guarantees bounded waiting
- Decrementing variable - Decre, Down, P or Wait
- Incrementing variable - Incre, Up, V or Signal
- When entering CS, use UP and when exiting, use DOWN

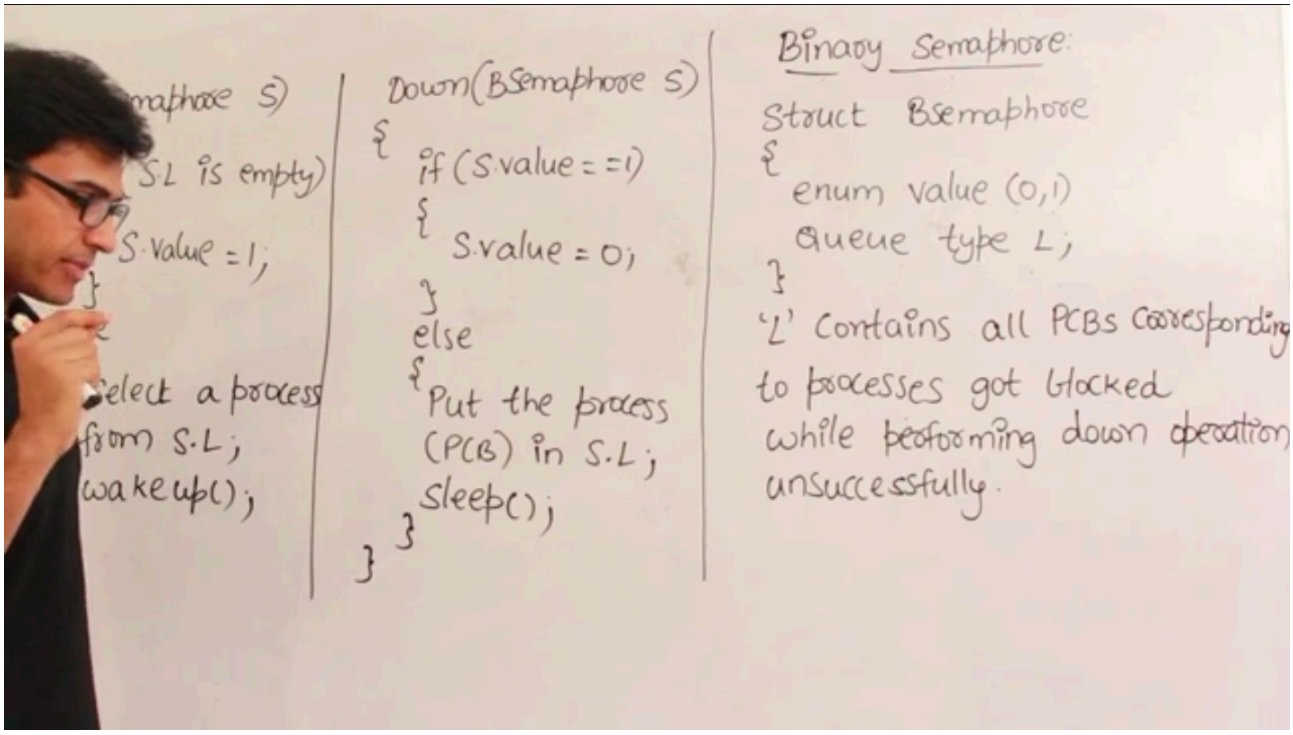


Lecture #19: Problems on Counting Semaphores

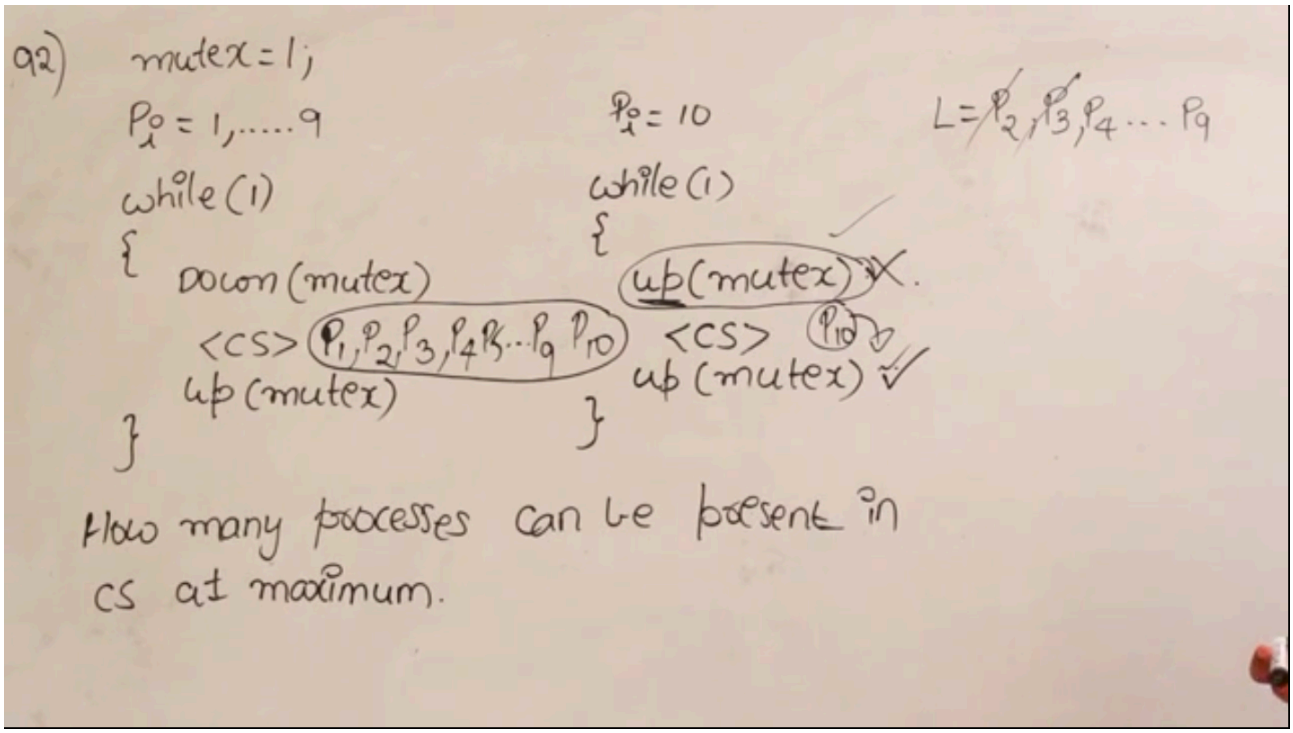


Lecture #20: Binary semaphores or Mutexes

- enum variable can take only enumerated values

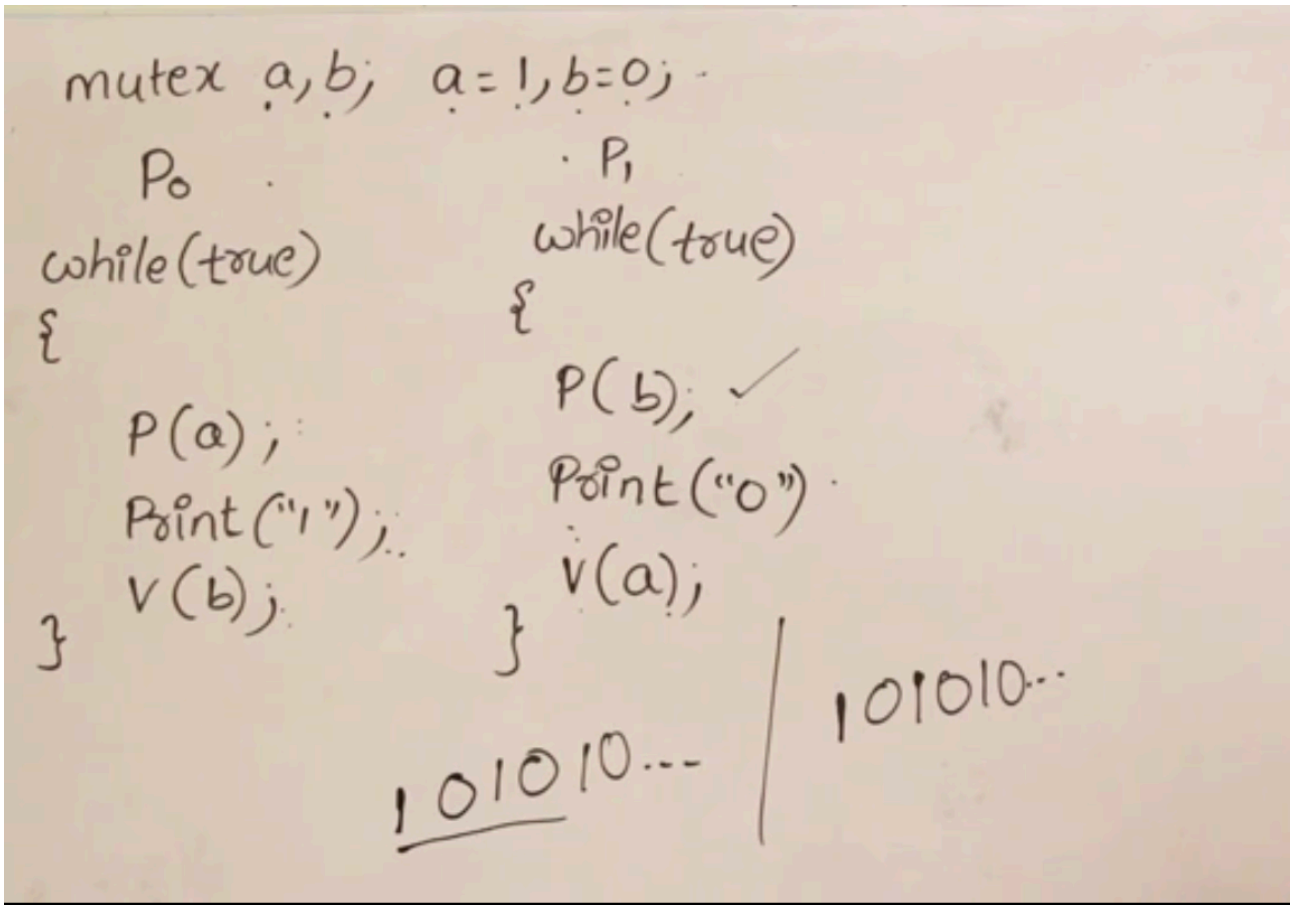


Lecture #21: Gate 92 question on mutex



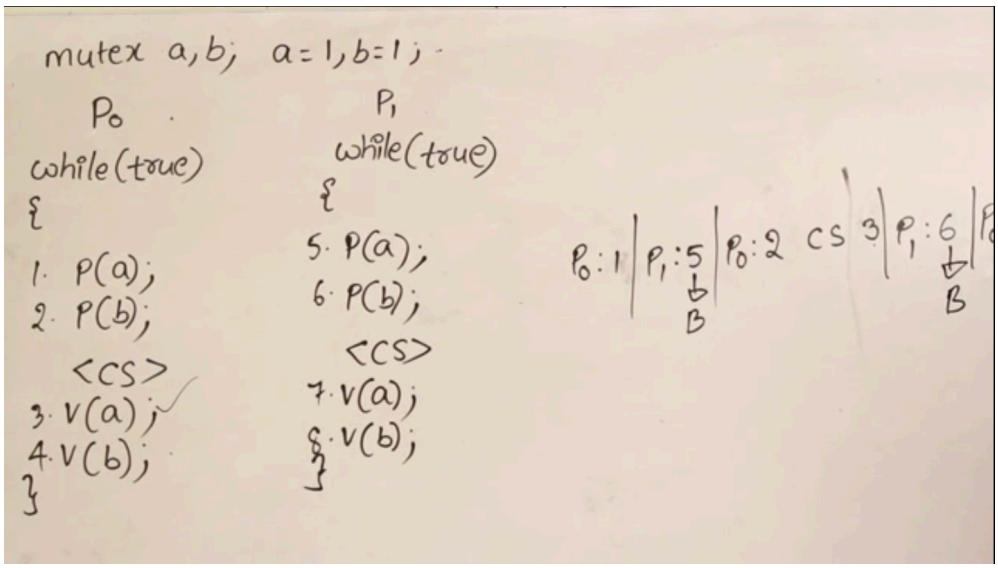
- 10 processes can be present inside due to incorrect implementation

Lecture #22: Mutex Example

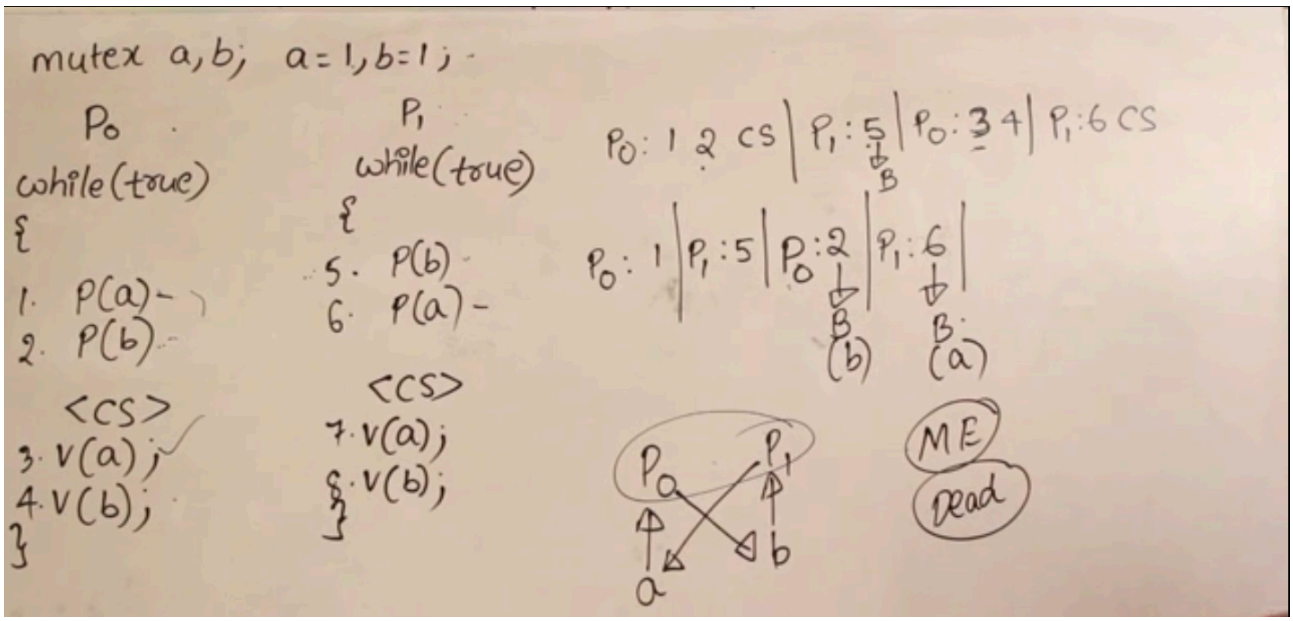


- the placement of the mutexes will guarantee that strict alteration is followed
- ∴ mutexes can not only provide mutual exclusion, but also be used to execute the order in which code gets executed

Lecture #23: Mutex Example 1



- is mutual exclusion Guaranteed? Yes.



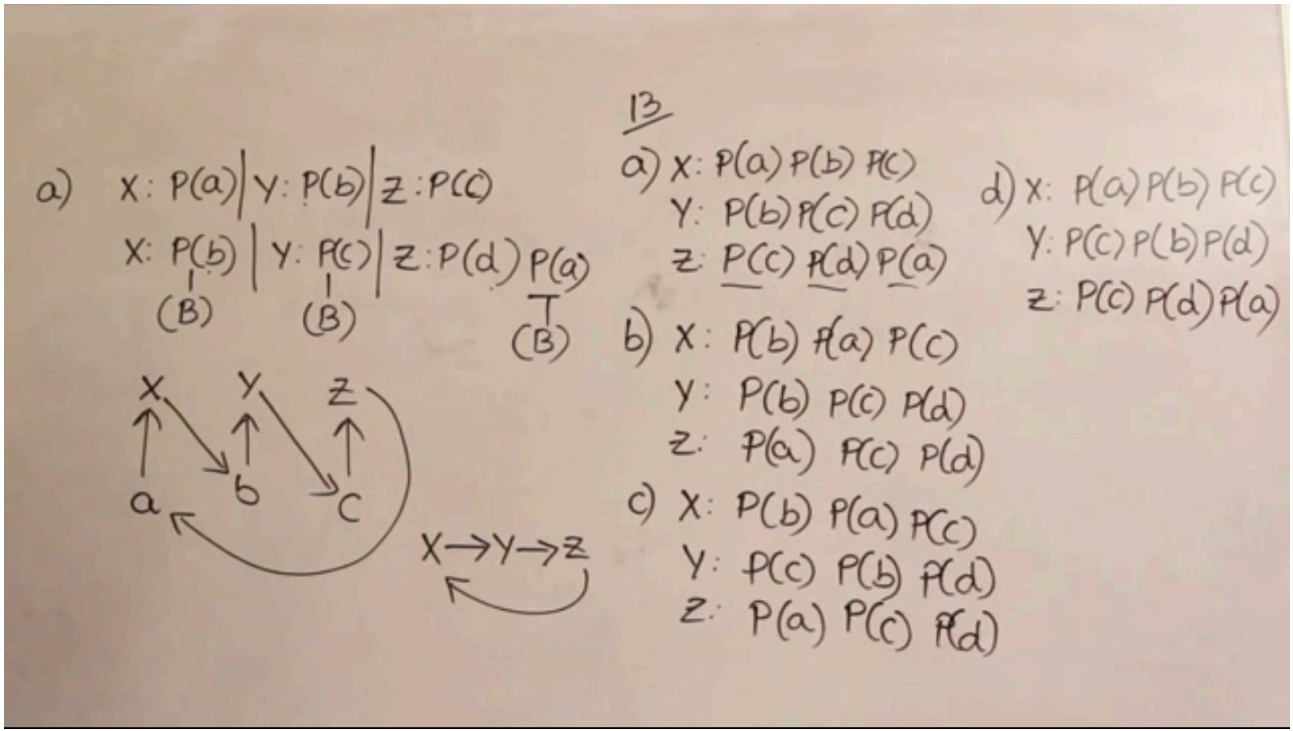
- Is mutual exclusion guaranteed? Yes.
- is deadlock possible? Yes.
- Is deadlock always possible? No.
- But if something can go wrong, it will go wrong (Murphys law) ∴ we won't use this solution.

Lecture #24: Gate 2013 Question on mutex

There are 3 processes X, Y and Z and three binary semaphores a, b, c and d.

Given they perform different sequences of DOWN on the given semaphore, in which sequence is deadlock not possible?

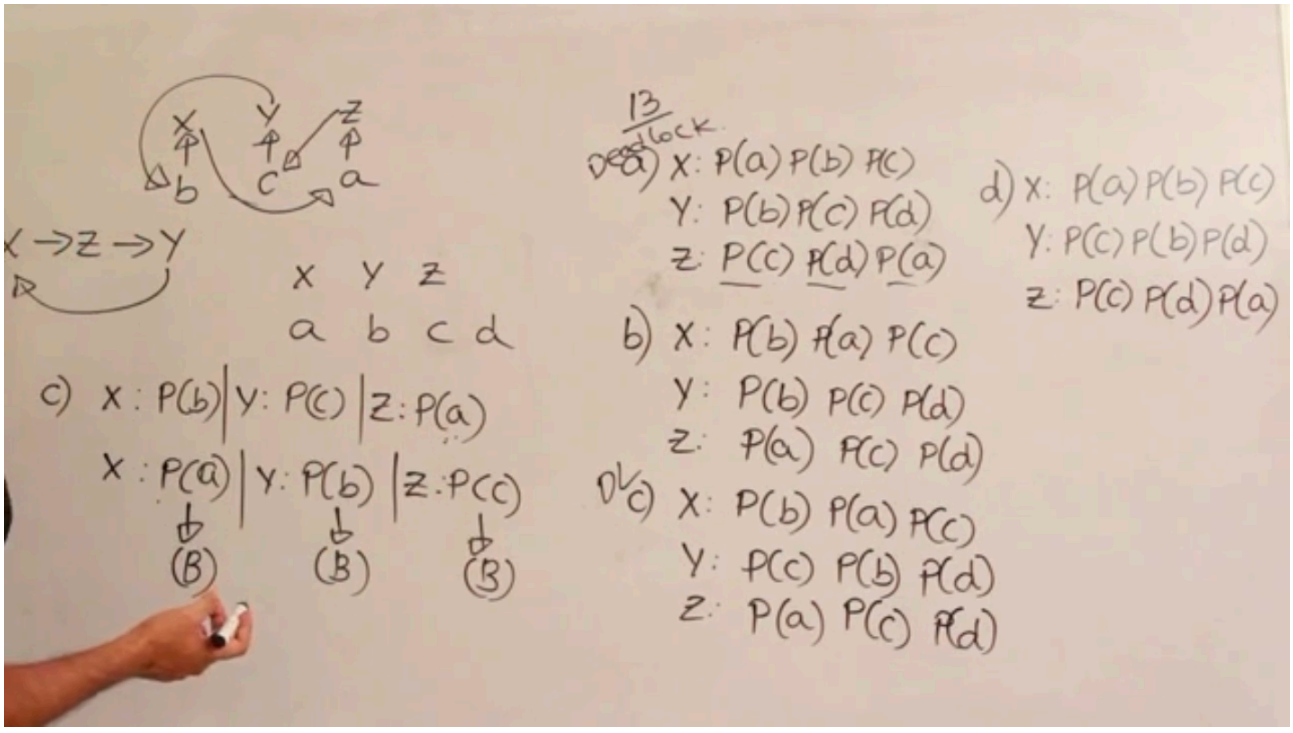
- The general rule to follow while solving these kinds of questions is as follows:
- Try to perform 1 DOWN operation on each semaphore by each process and see if deadlock is possible

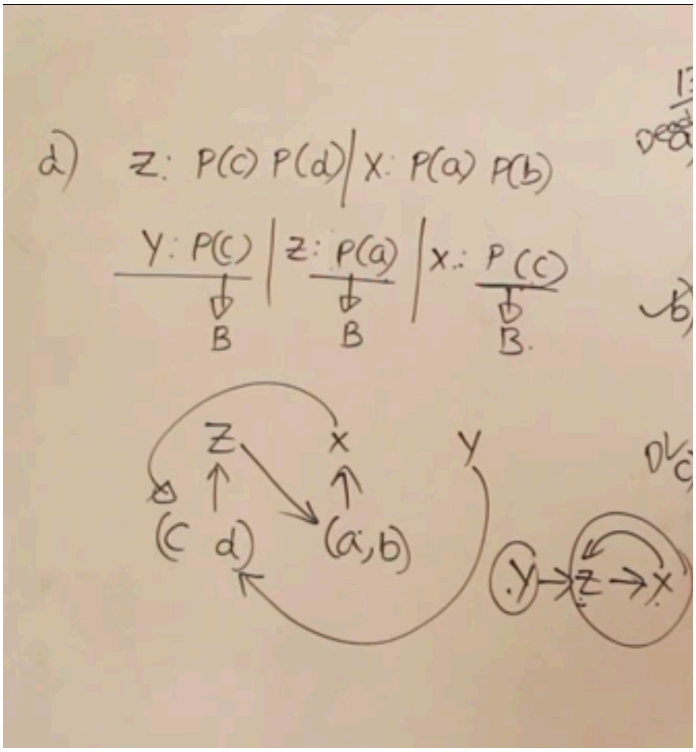


- If giving 1 mutex to each process is not possible, try giving two mutex to each process and then see if deadlock is possible

Whenever there is a cycle, meaning a set of processes waiting for each other, then it means there is deadlock.

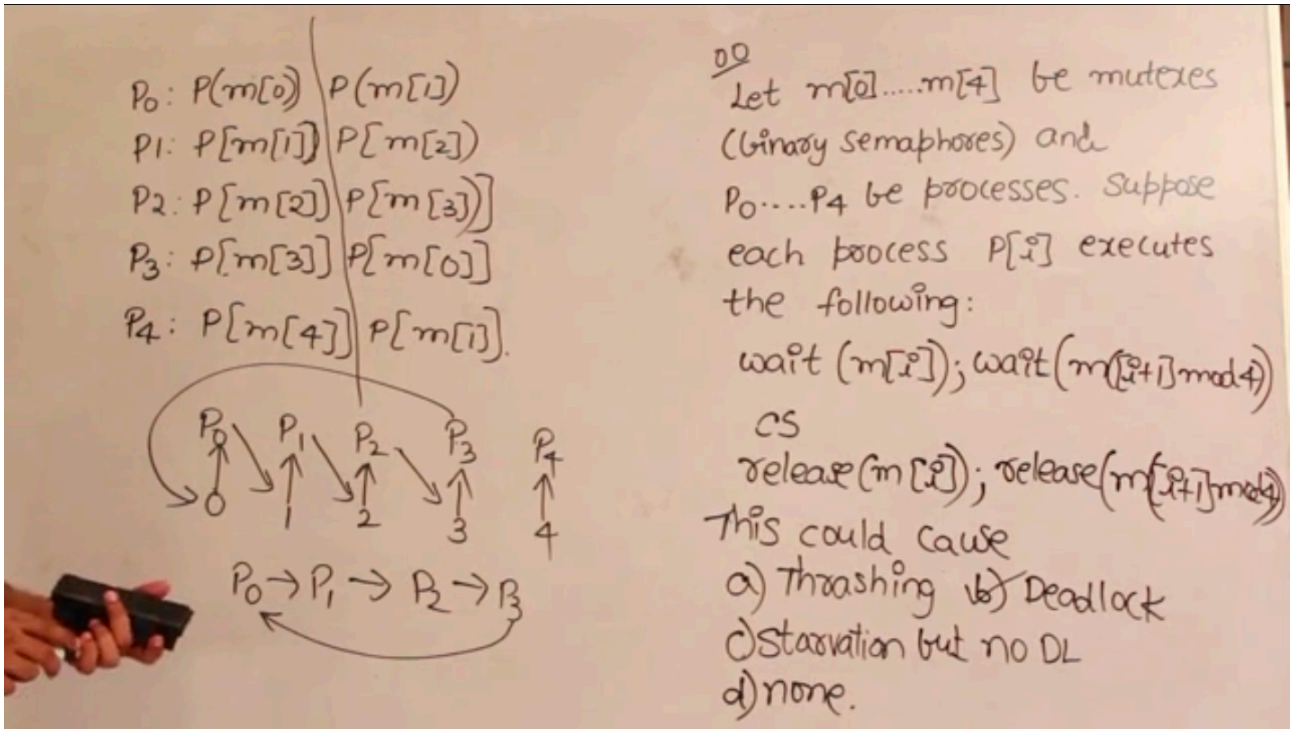
When you cannot find a deadlock in a sequence, then hold it for some time and try to find the deadlock in some other sequence. If you can find the deadlock in other sequences, you can eliminate them and eventually find the correct.





Since deadlock is possible in a b and d, ∴ c is the correct answer.

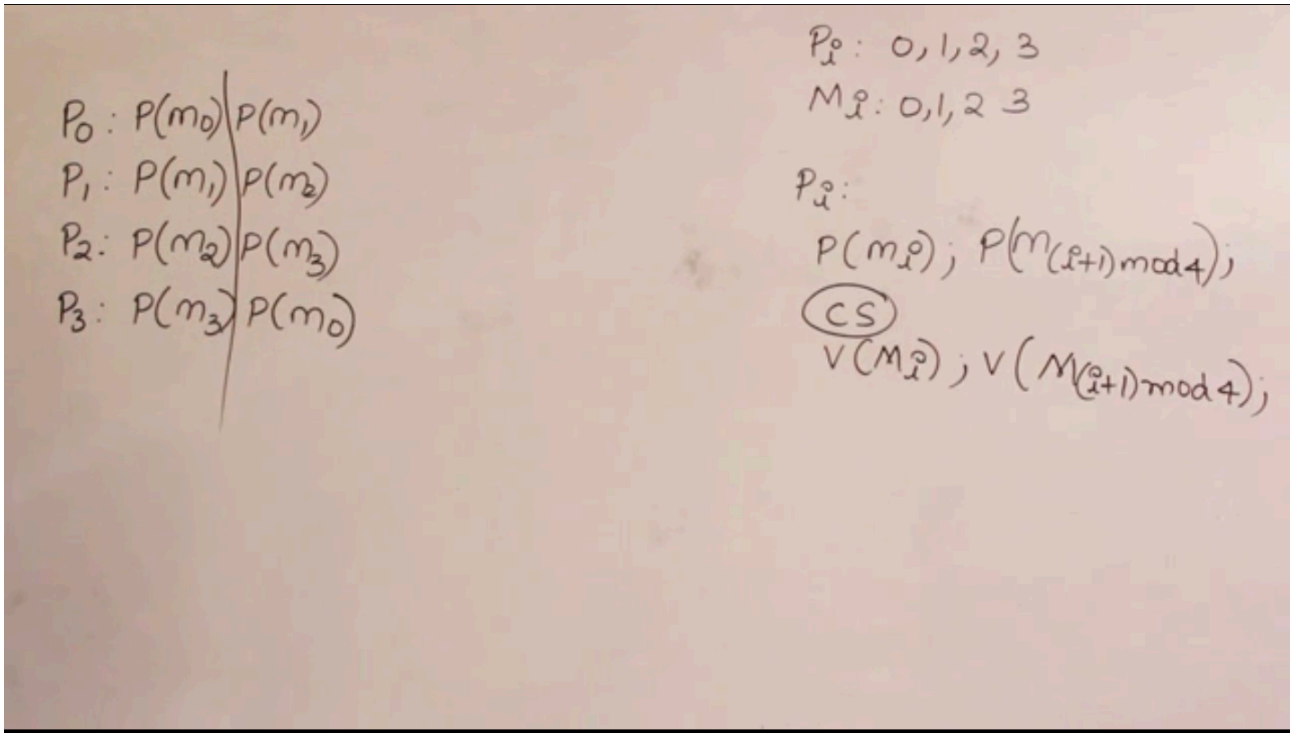
Lecture #25: Gate 2000 Question mutex



Deadlock is definitely possible, but is mutual exclusion possible? No.

How many processes can enter into the critical section at once? 2.

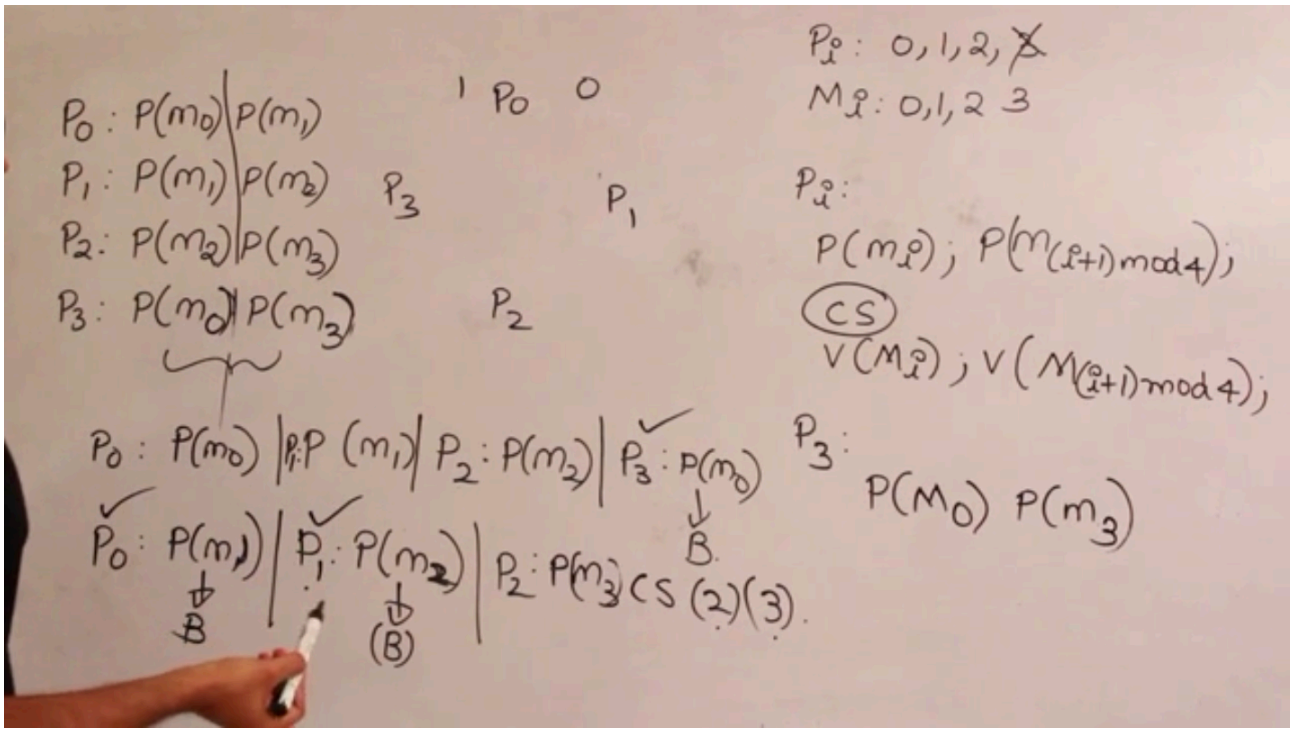
Lecture 26: Gate 96 Question on dining philosophers problem



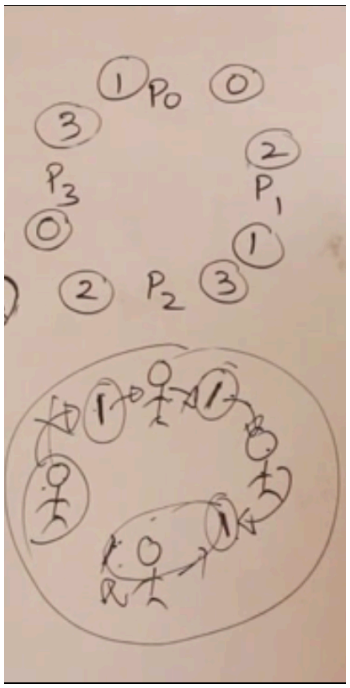
In this arrangement, as seen before, deadlock is definitely possible.

What can we do in order to break the deadlock? This is the dining philosophers problem.

What we can do is that change one of the process's semaphore order!



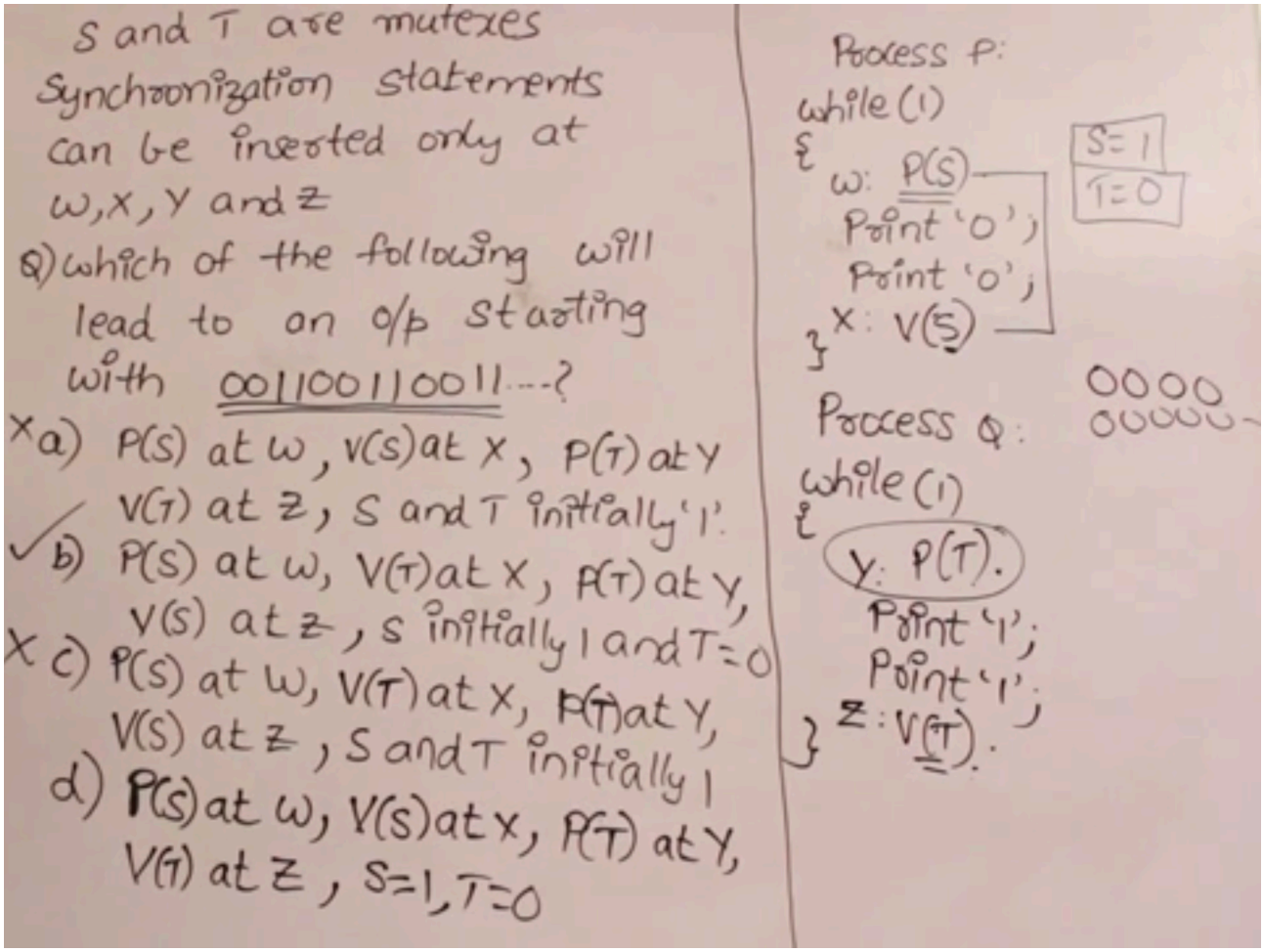
This way, by making one process's *semaphore out of order*, the deadlock is broken.



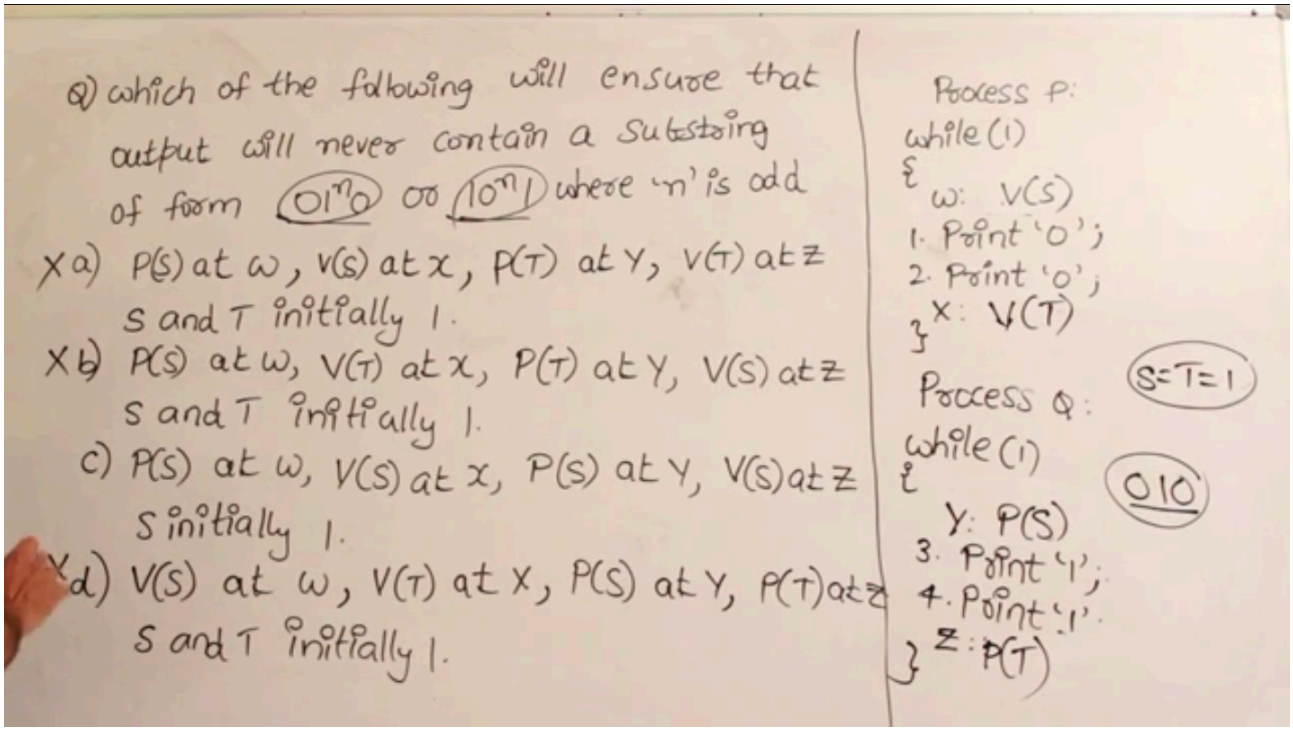
Dining philosophers problem: There are 4 philosophers who are eating spaghetti on a round table. And there are 4 forks shared between them. Each philosopher requires two forks two eat fork. The philosophers either eat the spaghetti or muse. Each philosophers take the left fork first and then the right fork. The way in which they are eating, no one will be able to finish their meal. How can we break this deadlock?

Answer: Make one philosopher take the right fork first and then the left one. This will ensure there is no deadlock and the philosophers finish their meal.

Lecture #27: Gate 2003 Question (part 1)



Lecture #28: Gate 2003 Question (part 2)

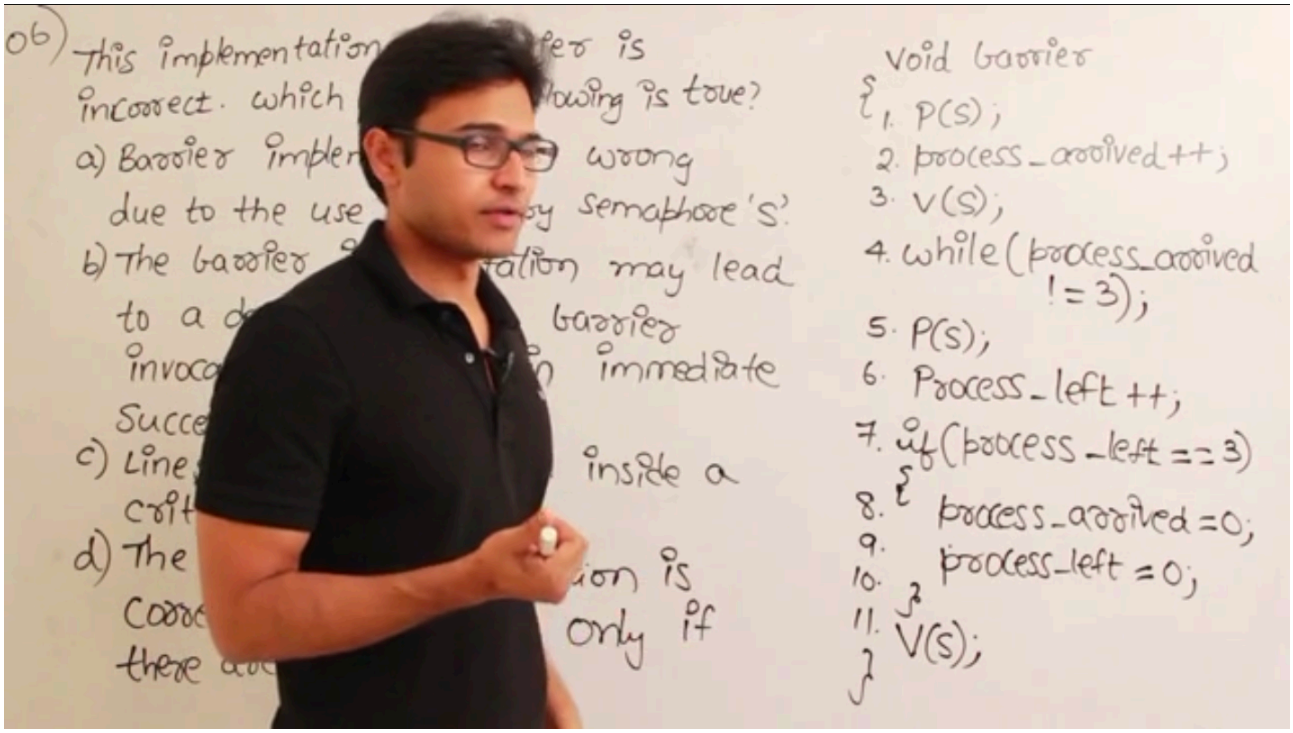


Lecture #29: Gate 06 Question on Barrier(part 1)

Barrier means every process will wait until all process reach to a certain point called barrier.
- it is busywaiting solution

Analogy - A cricket team of 11 people waiting to go to the cricket ground and waiting for all the teammates to come, and when everyone has arrived, all one by one jump the wall beyond which there is cricket ground.

There is a paper on the wall, and once a person arrives, he updates the number of the paper and sits waiting for other people to come. Then after everyone has arrived, they start leaving and start updating another paper which shows how many people have jumped the wall, so that once all the people have left, the last one can reset the papers which counted the number of people arrived and number of people left, so that the papers could be of use if needs to be used again.

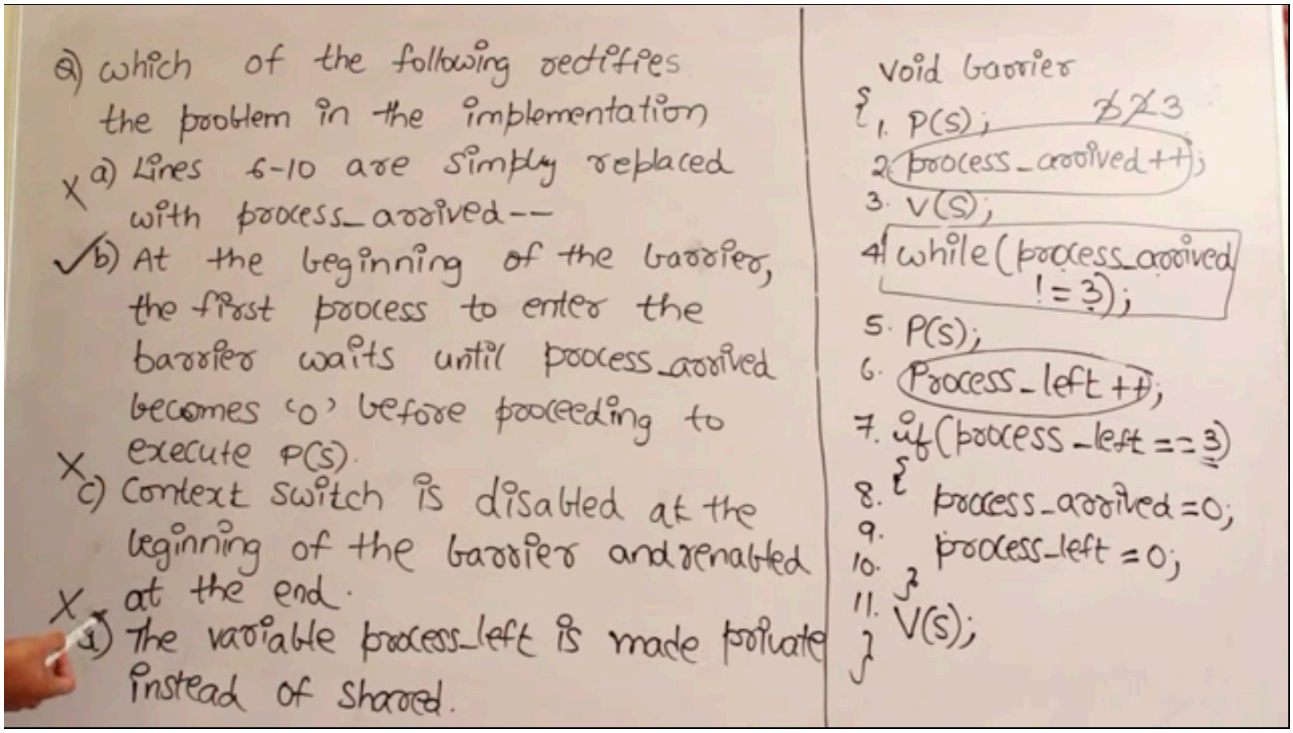


This implementation is incorrect if it used in immediate succession. This is because if used in immediate succession, the counter to count the number of people arrived isn't capped to max number of people, and therefore it can be more than that. ∴ if there are three people, the third one can jump and then immediately again come back, increasing the counter to 4, which will eventually be reset by the last person to 0 and 0. Then if everyone comes again, they will keep on waiting for that one person to arrive as one count will be lost and even after everyone has arrived, the count can be at most 2, not more than that. ∴ it is wrong.

Lecture #30: Gate 06 Question on Barrier(part 2)

Q. How to fix the barrier implementation in the previous question?

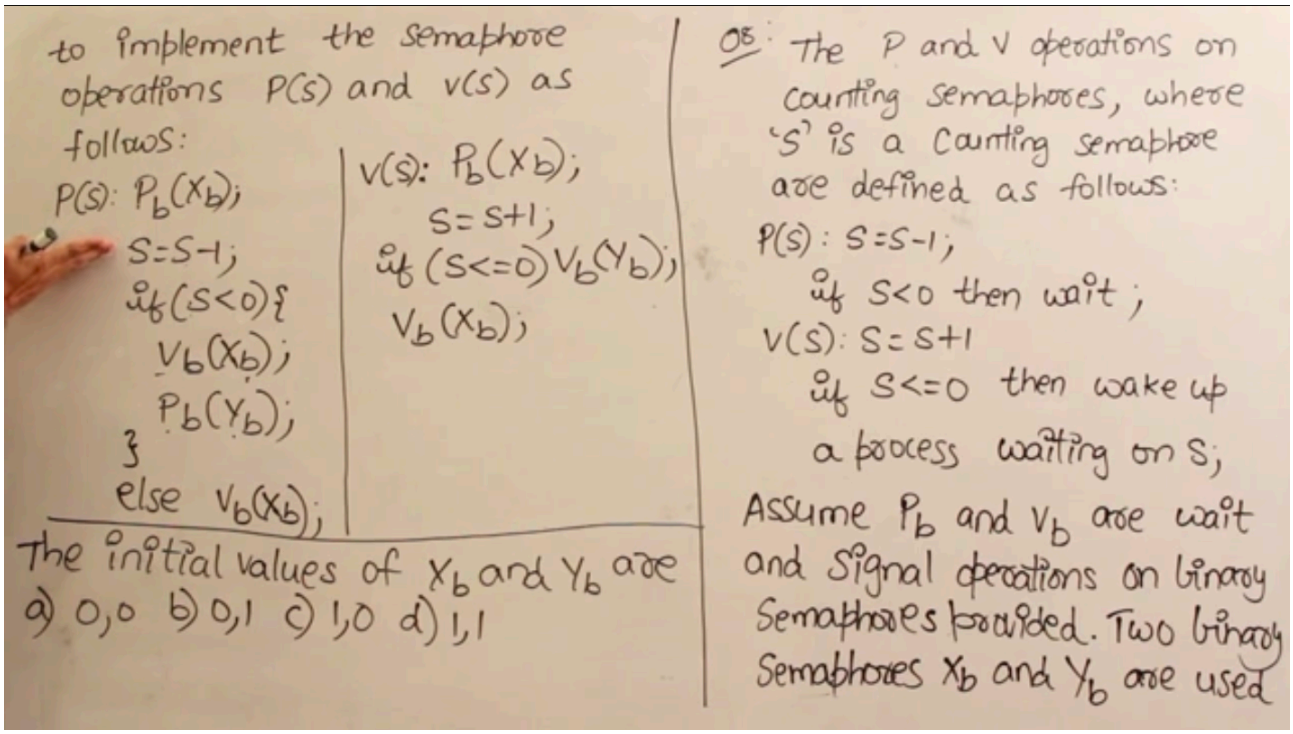
Answer: Let the process (or person) which immediately comes back after crossing the barrier/ jumping the wall to wait until both counters have turned to 0.

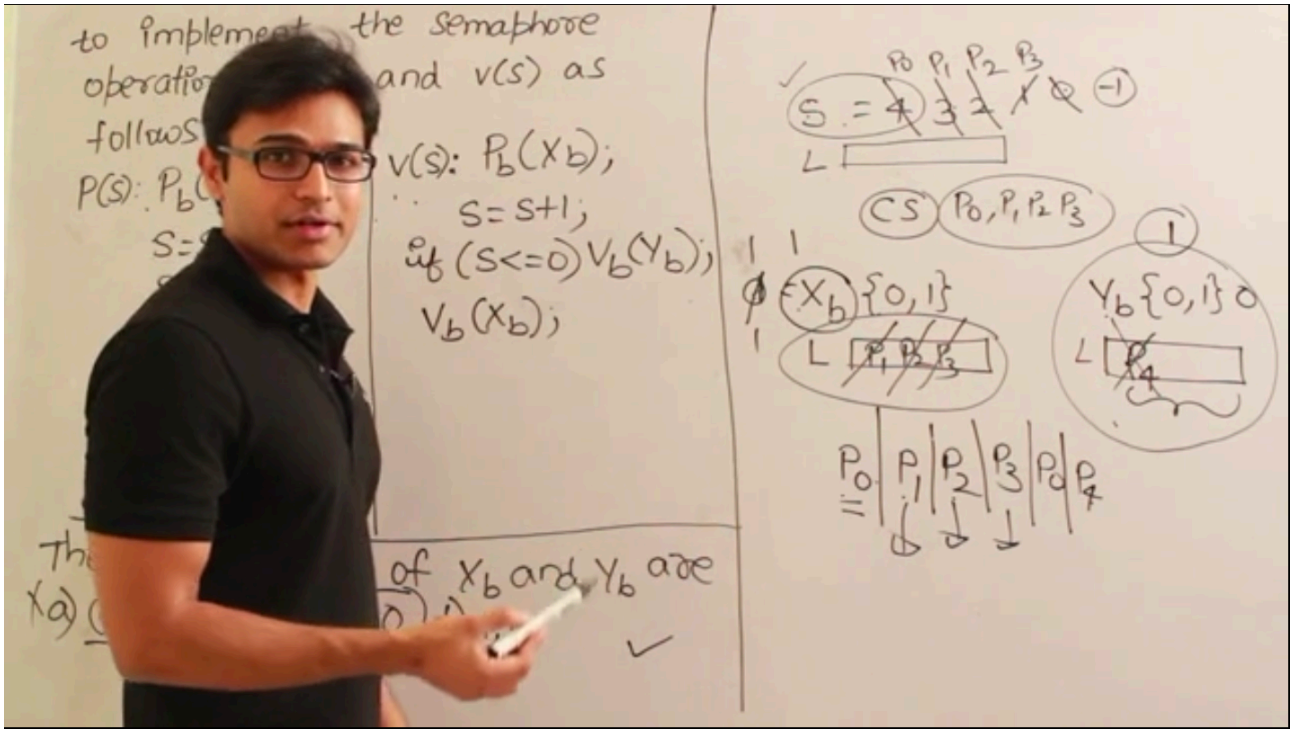


Lecture #31: Gate 2008 Question on implementing Counting Semaphore using Mutexes

To implement counting semaphore using binary semaphore or mutexes, we will use two mutexes X_b & and Y_b.

- X_b will take care of implementing mutual exclusion when incrementing the the count
- Y_b will be used to maintain the queue of those processes who have performed DOWN operation unsuccessfully.





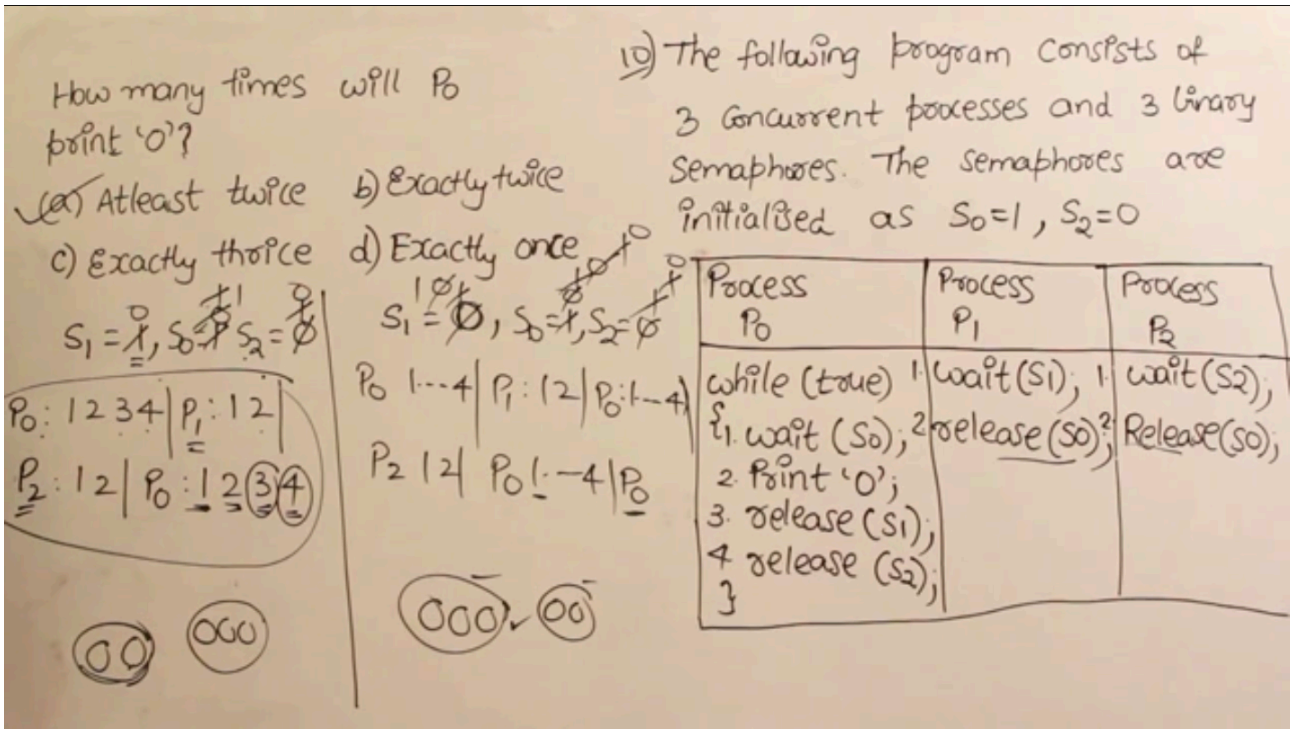
Both X_b and Y_b will have two independent queues related to them.

Queue linked to X_b will store the processes which are trying to perform a DOWN operation on S **WHILE** some other process is performing it.

And the queue linked to Y_b will store the processes which actually **HAVE** unsuccessfully performed DOWN on S.

The initial values of both the mutexes are $X_b = 1$ and $Y_b = 0$ for the implementation to properly work.

Lecture #32: Gate 2010 Question on Mutexes

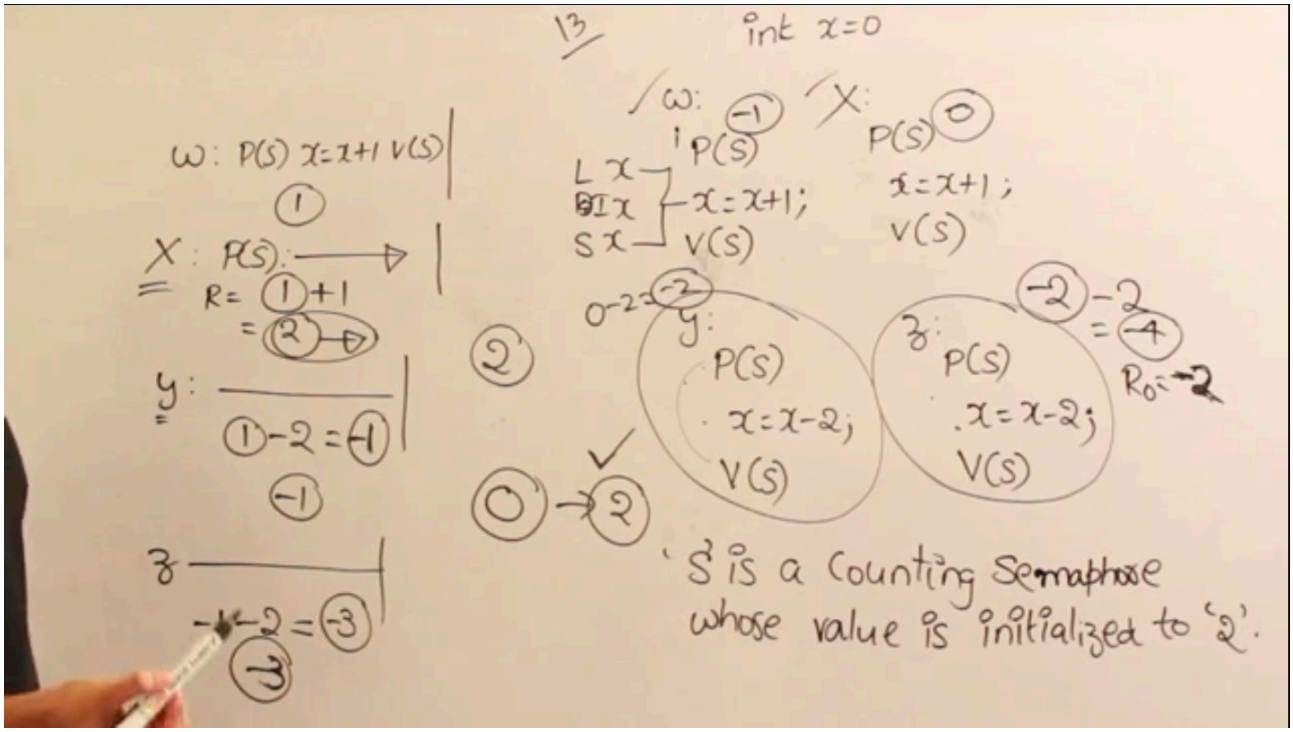


Lecture #33: Gate 2013 Question on Counting Semaphores

If X is a counting semaphore initialised to 2, then what is the maximum value of x possible after processes w, X, y, z run?

Incrementing a variable has three parts - load, increment, store and preemption can occur at any stage while running the program.

The maximum value following some sequence of preemption for both program will be 2 and the



minimum will be -4.

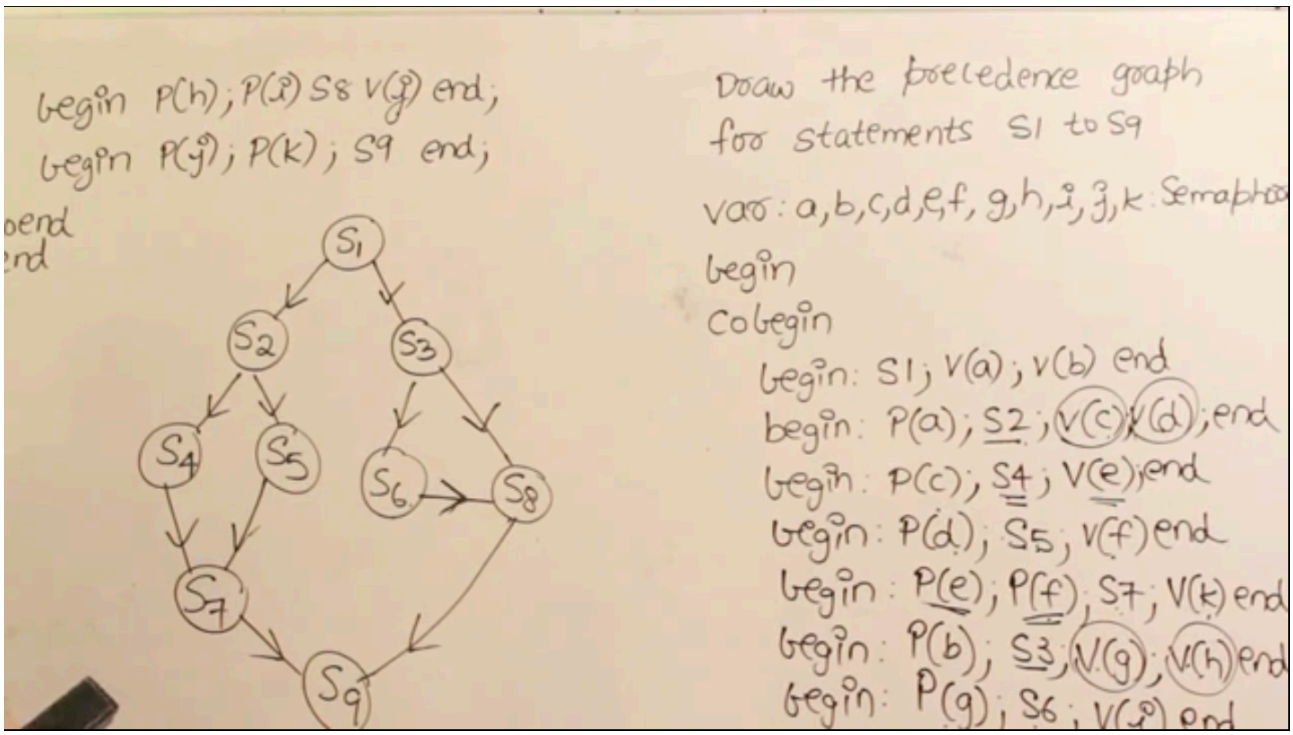
Lecture #34: Gate 1995 Question on Concurrent Processes and Semaphores

cobegin and coend means concurrent begin and concurrent end

The statement written between cobegin and coend can be executed in any other.

We have to draw the precedency graph, which will show us in which order can the statements be executed.

All the mutexes from a to k are initialised to 0.



Lecture #35:Gate 1993 Question on Concurrent Processes and Semaphores 2

Here we are given the precedency graph, and we have to write the statements between cobegin and coend.

Depending on the number of incoming edges, we can decide on the number of down operations that we have to put before the statements, and depending on the number of outgoing edges, we can decide the number of up operations we have to place after the statement ends.