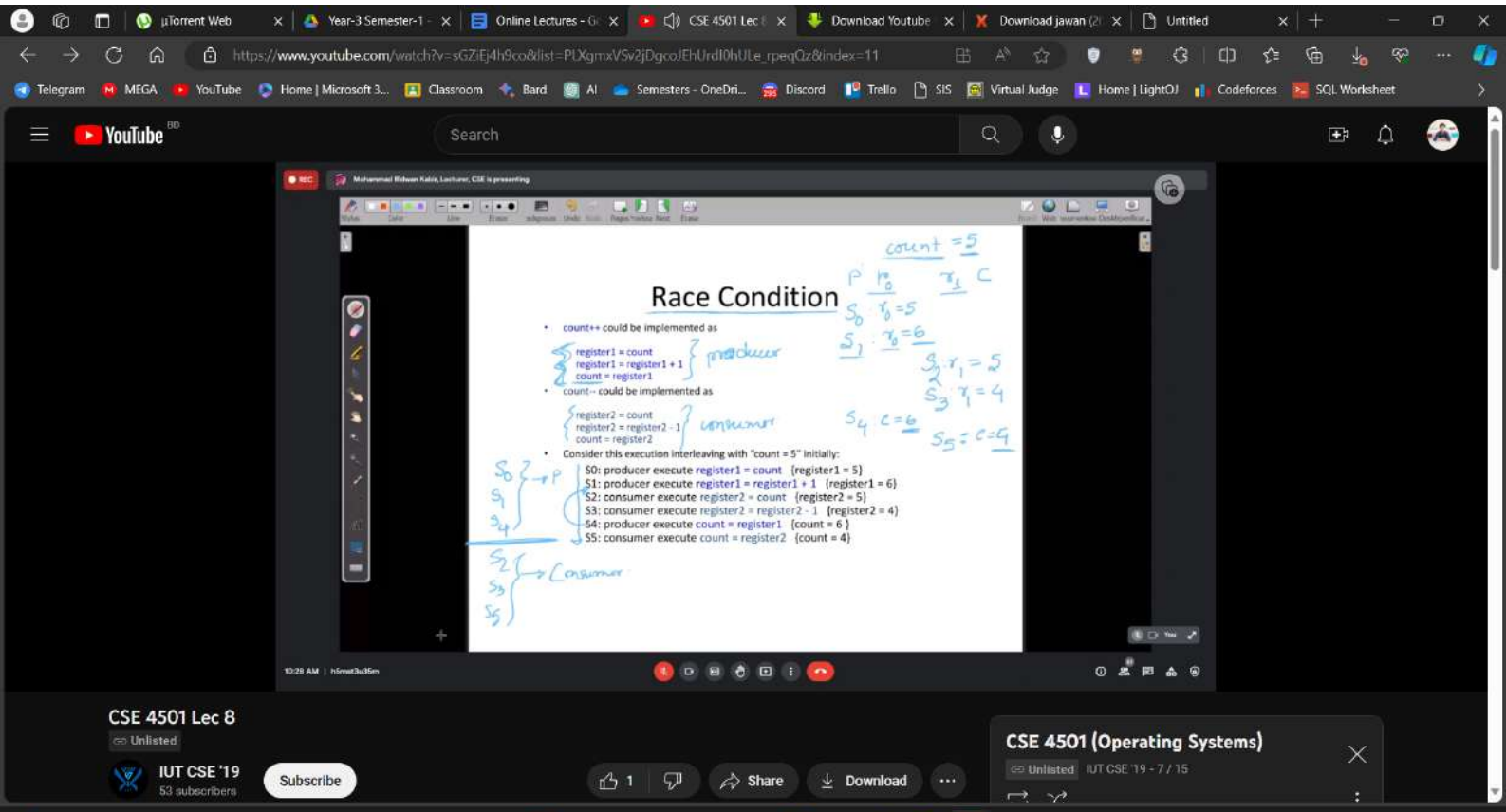


CSE 4501 Lec 8

Unlisted

IUT CSE '19

53 subscribers

Subscribe

1 Share Download

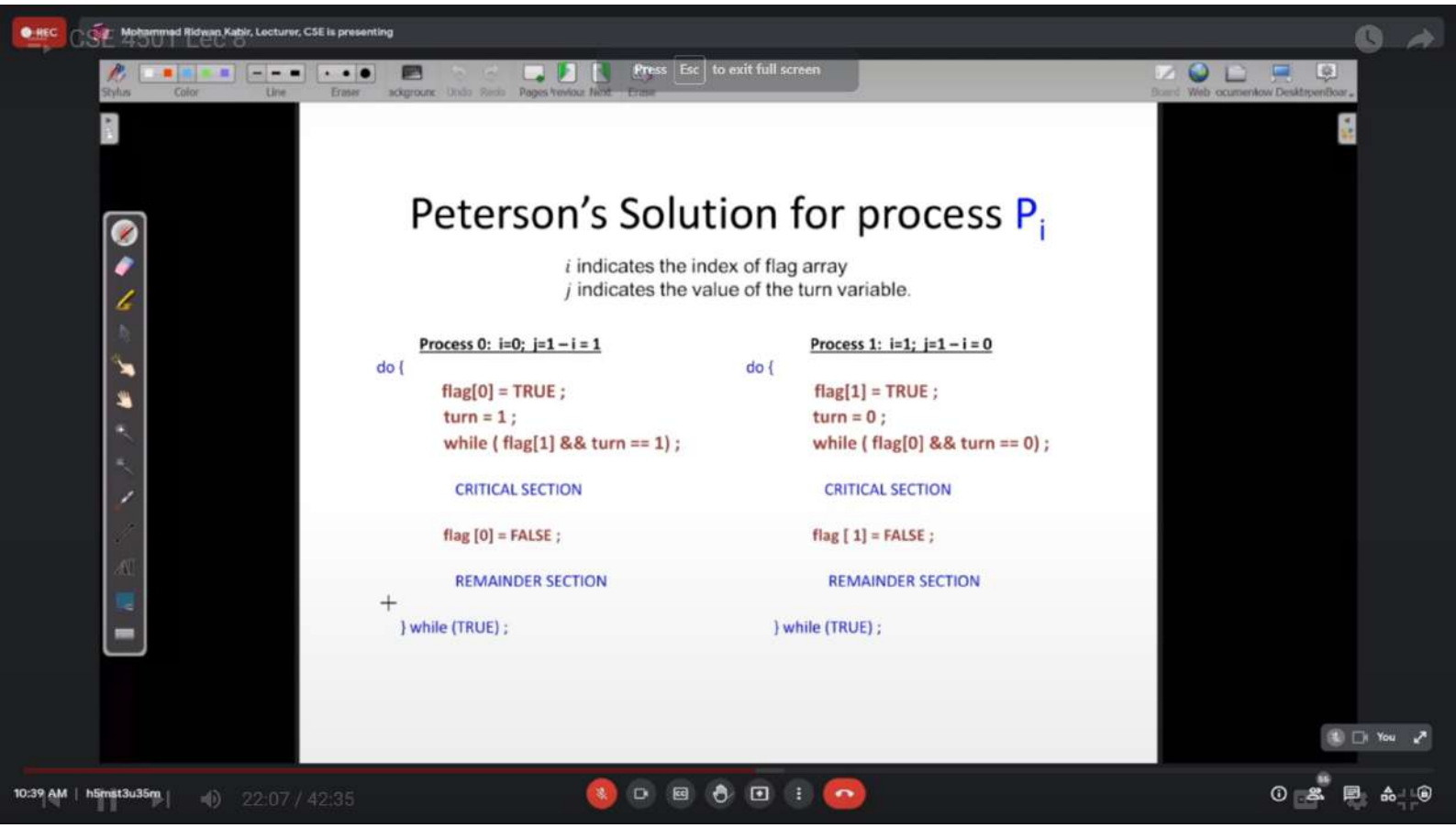
CSE 4501 (Operating Systems)

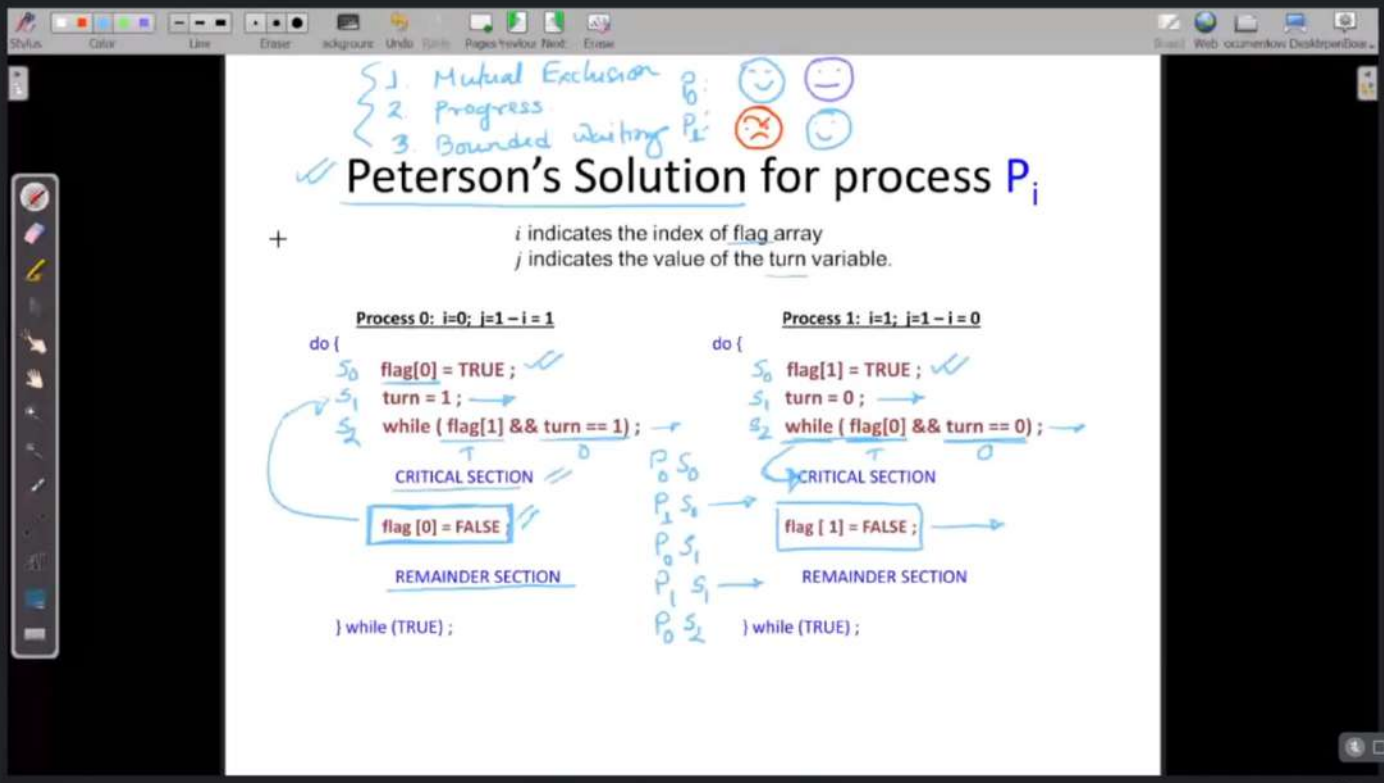
Unlisted IUT CSE '19 - 7 / 15

25°C Haze

Search

10:29 PM 10/29/2023


Mohammad Ridwan Kabir, Lecturer, CSE is presenting
Press Esc to exit full screen
Peterson's Solution for process P_i
i indicates the index of flag array
j indicates the value of the turn variable.
Process 0: i=0; j=1 – i = 1
do {
flag[0] = TRUE ;
turn = 1 ;
while (flag[1] && turn == 1) ;
CRITICAL SECTION
flag [0] = FALSE ;
REMAINDER SECTION
} while (TRUE) ;
Process 1: i=1; j=1 – i = 0
do {
flag[1] = TRUE ;
turn = 0 ;
while (flag[0] && turn == 0) ;
CRITICAL SECTION
flag [1] = FALSE ;
REMAINDER SECTION
} while (TRUE) ;



CSE 4501 Lec 8

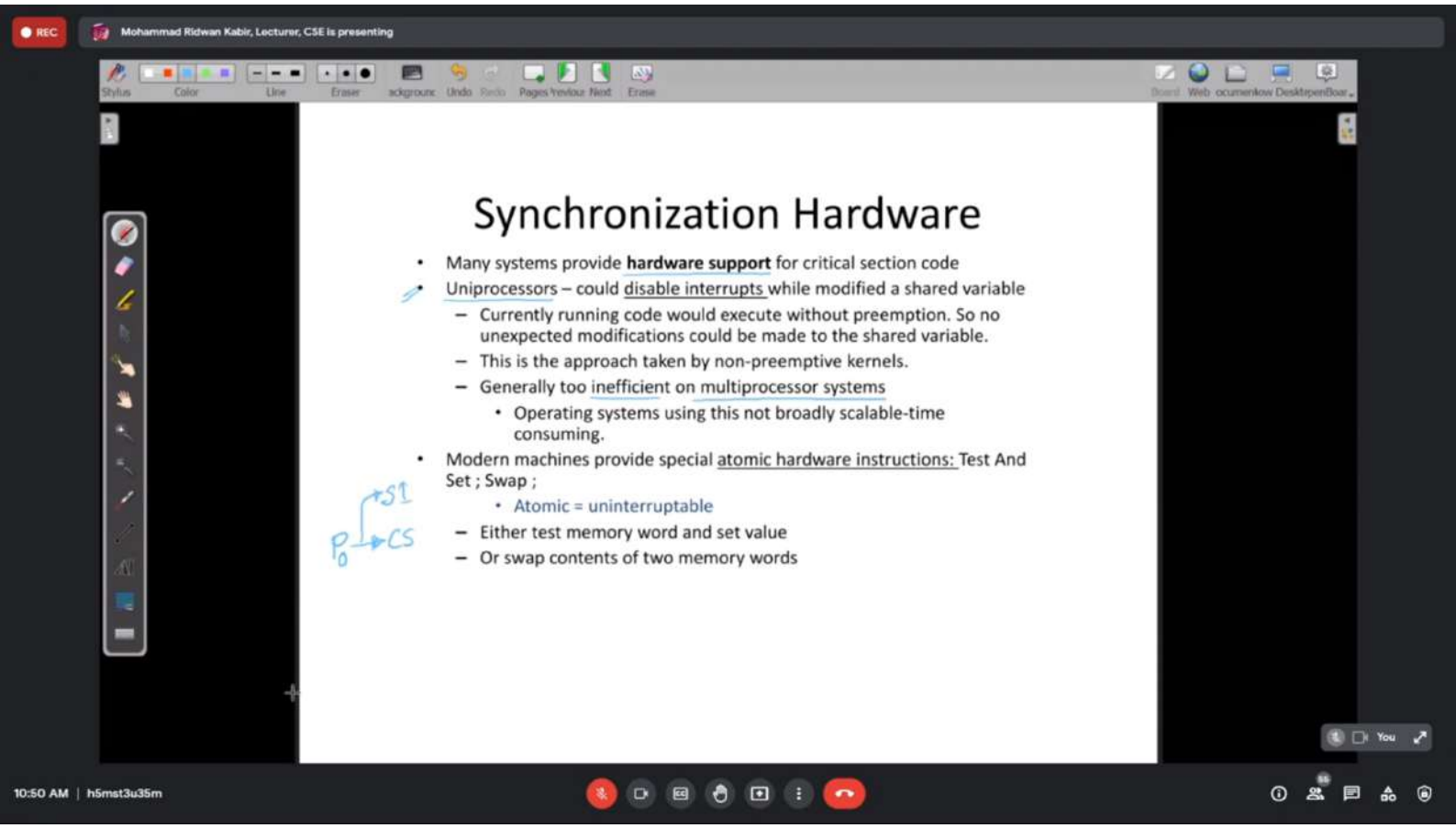
CS solution with locks using h/w

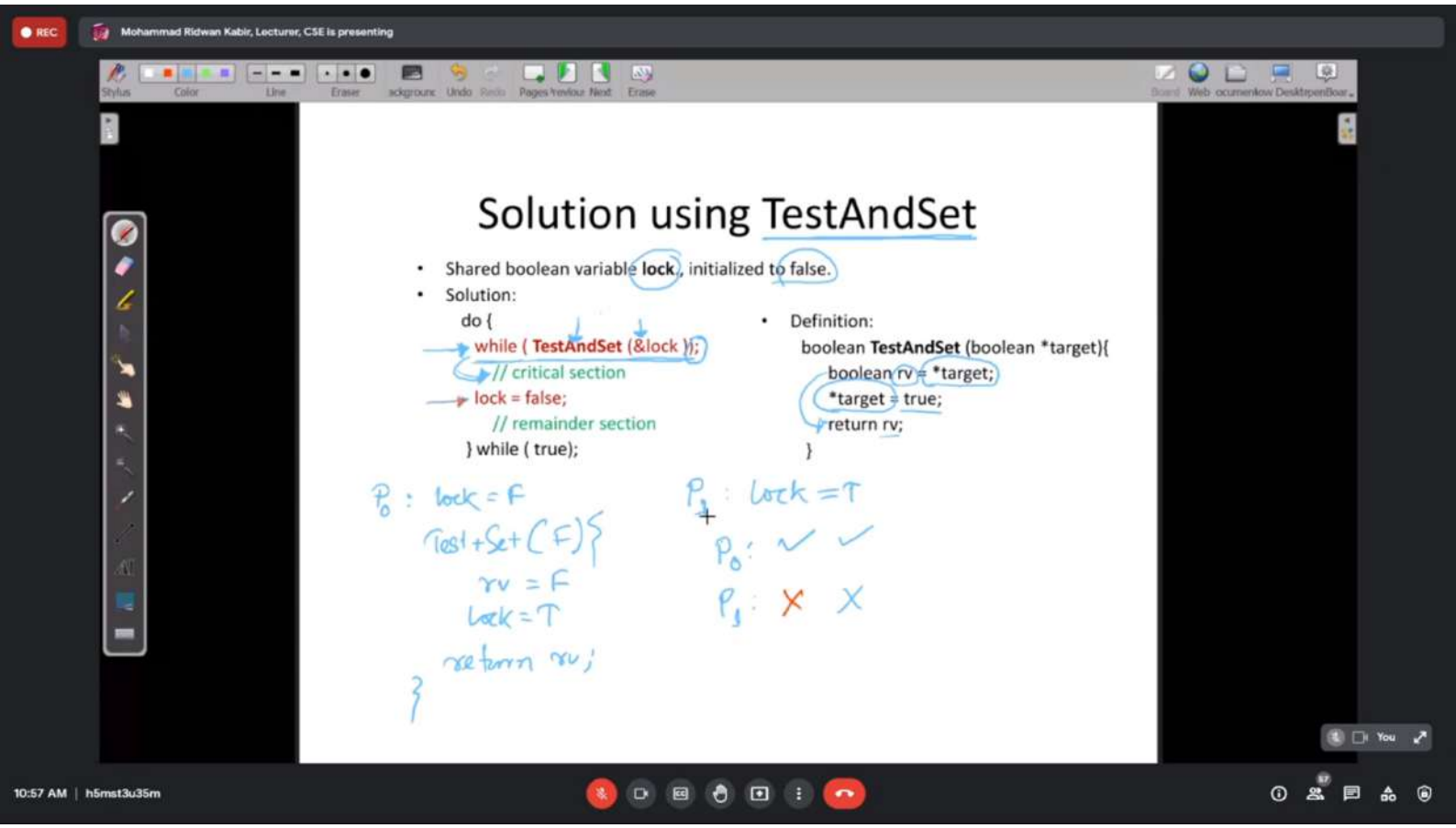
- A simple tool requires-lock
- Race conditions are prevented by requiring that critical section be protected by locks.

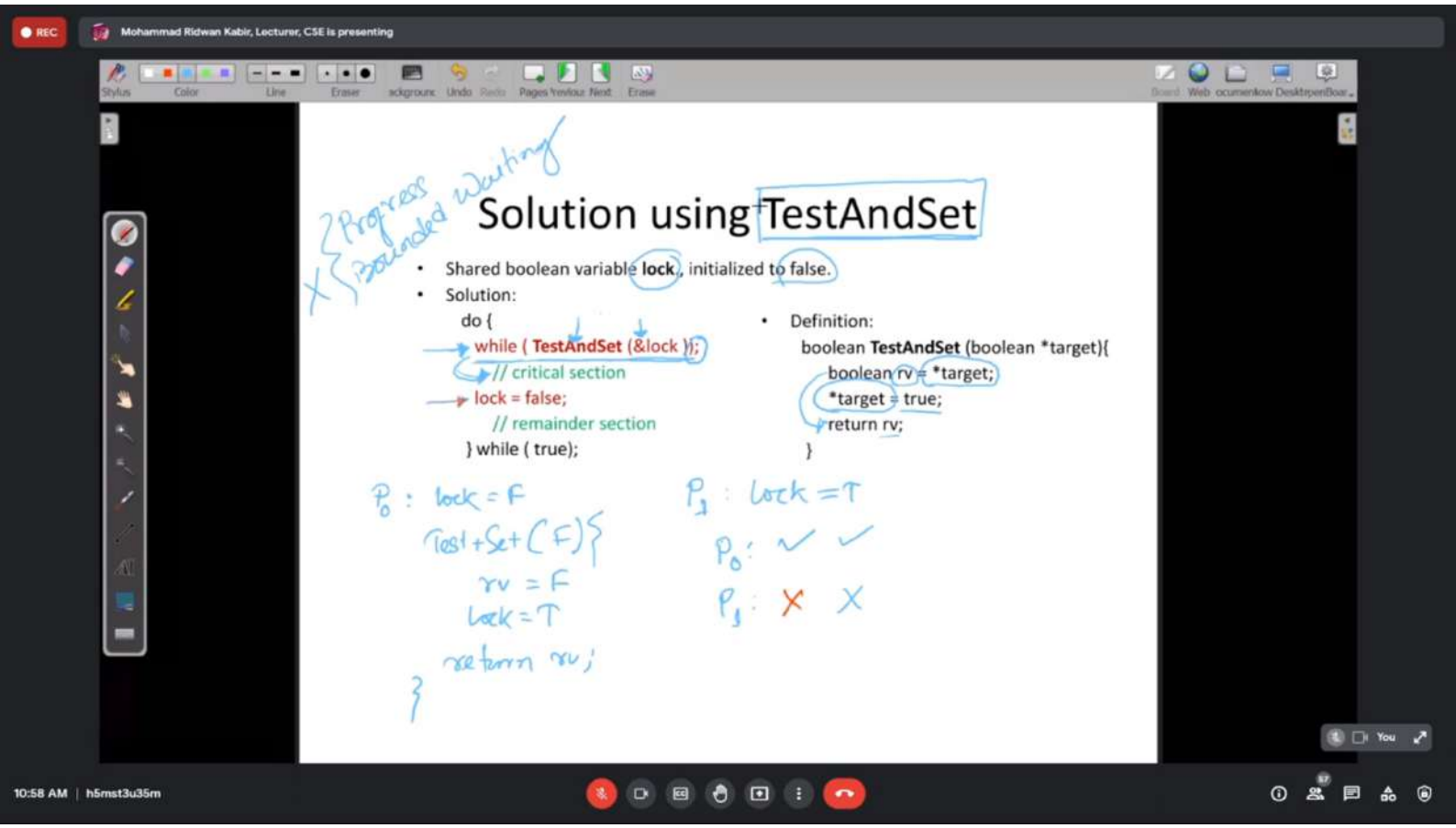
```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

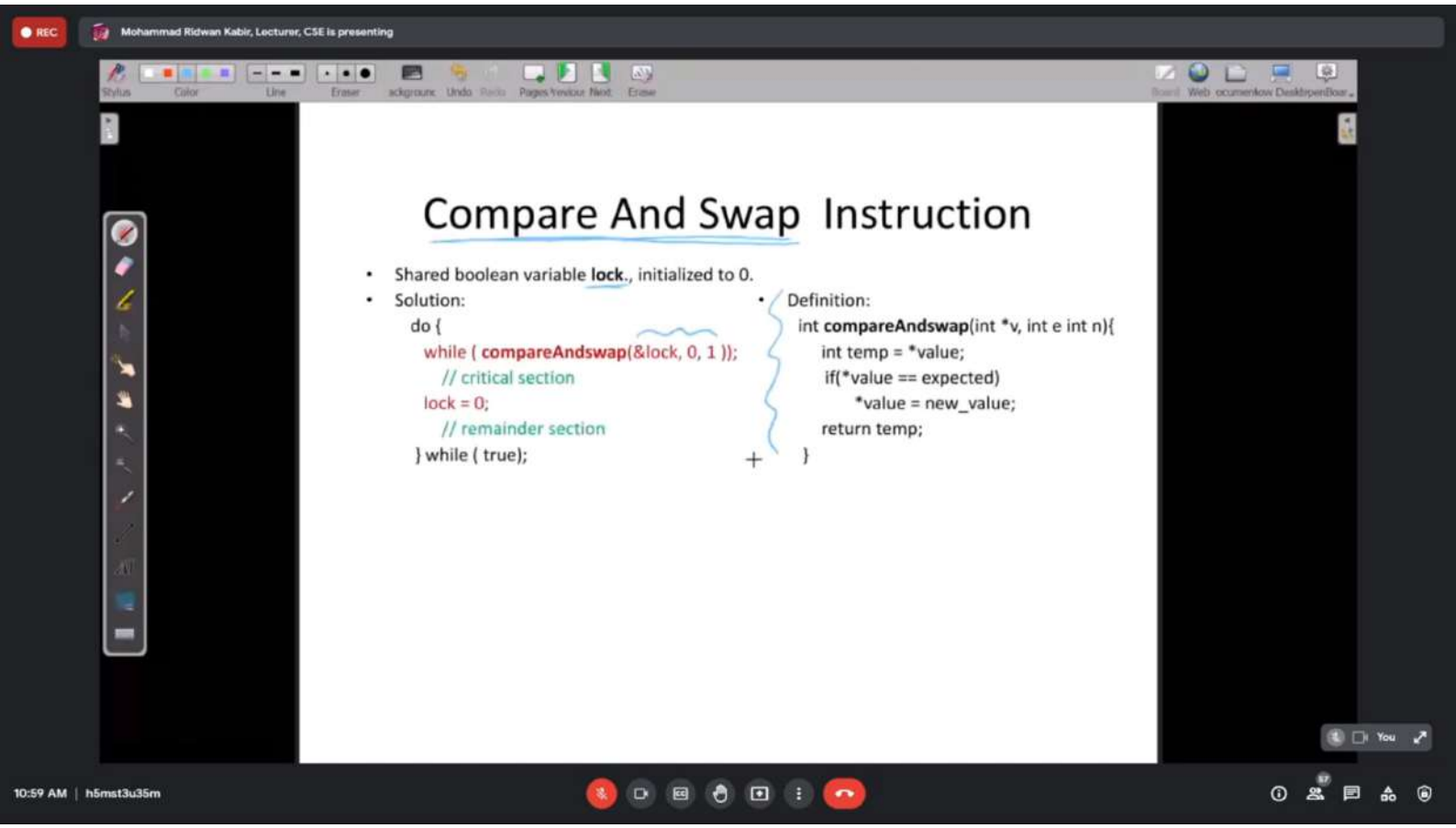
Figure 6.3 Solution to the critical-section problem using locks.

32:27 / 42:35


REC
Mohammad Ridwan Kabir, Lecturer, CSE is presenting
Synchronization Hardware
Many systems provide hardware support for critical section code
Uniprocessors – could disable interrupts while modified a shared variable
Currently running code would execute without preemption. So no unexpected modifications could be made to the shared variable.
This is the approach taken by non-preemptive kernels.
Generally too inefficient on multiprocessor systems
Operating systems using this not broadly scalable-time consuming.
Modern machines provide special atomic hardware instructions: Test And Set ; Swap ;
Atomic = uninterruptable
Either test memory word and set value
Or swap contents of two memory words
You
10:50 AM | h5mst3u35m


REC
Mohammad Ridwan Kabir, Lecturer, CSE is presenting
Solution using TestAndSet
Shared boolean variable lock, initialized to false.
Solution:
do {
while (TestAndSet (&lock));
// critical section
lock = false;
// remainder section
} while (true);
Definition:
boolean TestAndSet (boolean *target){
boolean rv = *target;
*target = true;
return rv;
}
P0 : lock = F
TestAndSet(F){
rv = F
lock = T
return rv;
}
P1 : lock = T
P0 : ✓ ✓
P1 : X X
You
10:57 AM | h5mst3u35m


REC
Mohammad Ridwan Kabir, Lecturer, CSE is presenting
Solution using TestAndSet
Shared boolean variable lock, initialized to false.
Solution:
do {
while (TestAndSet (&lock));
// critical section
lock = false;
// remainder section
} while (true);
Definition:
boolean TestAndSet (boolean *target){
boolean rv = *target;
*target = true;
return rv;
}
Progress
Bounded Waiting
10:58 AM | h5mst3u35m


REC
Mohammad Ridwan Kabir, Lecturer, CSE is presenting
Compare And Swap Instruction
Shared boolean variable lock., initialized to 0.
Solution:
do {
while (compareAndswap(&lock, 0, 1));
// critical section
lock = 0;
// remainder section
} while (true);
Definition:
int compareAndswap(int *v, int e int n){
int temp = *value;
if(*value == expected)
*value = new_value;
return temp;
}
You
10:59 AM | h5mst3u35m

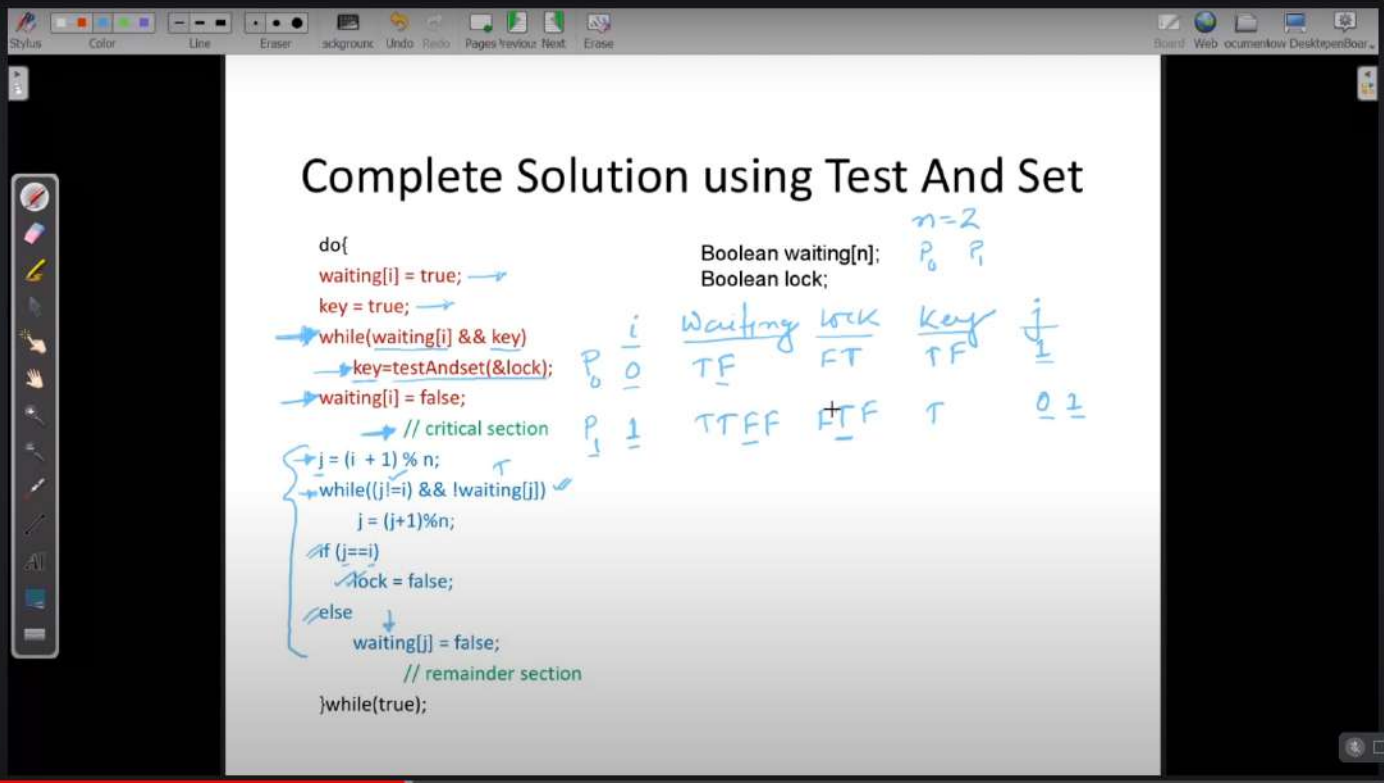
Complete Solution using Test And Set

```
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
    key=testAndset(&lock);
waiting[i] = false;
        // critical section
j = (i  + 1) % n;
while((j!=i) && !waiting[j])
      j = (j+1)%n;
if (j==i)
     lock = false;
else
     waiting[j] = false;
          // remainder section
}while(true);
```

```
Boolean waiting[n];
Boolean lock;
```

n=2 P_0 P_1

	i	Waiting	lock	Key	j
P_0	0	TF	FT	TF	
P_1	1	T_	FT	T	


Stylus Color Line Eraser Undo Redo Pages Previous Next Erase
Board Web Documents Desktop
Complete Solution using Test And Set
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
key=testAndset(&lock);
waiting[i] = false;
// critical section
j = (i + 1) % n;
while((j!=i) && !waiting[j])
j = (j+1)%n;
if (j==i)
lock = false;
else
waiting[j] = false;
// remainder section
}while(true);
Boolean waiting[n];
Boolean lock;
n=2
P0 P1
i
P0 0
P1 1
Waiting Lock Key j
T F F T T F 1
T T F F F T F T 0 1
You

Complete Solution using Test And Set

```
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
    key=testAndset(&lock);
waiting[i] = false;
        // critical section
j = (i + 1) % n;
while((j!=i) && !waiting[j])
      j = (j+1)%n;
if (j==i)
    lock = false;
else
    waiting[j] = false;
        // remainder section
}while(true);
```

```
Boolean waiting[n];
Boolean lock;
```

n=2

P0 P1

	i	Waiting	lock	Key	j
P0	0	T F	F T	T F	1
P1	1	T T F F	F T F	T	0 1

Complete Solution using Test And Set

```
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
    key=testAndset(&lock);
waiting[i] = false;
        // critical section
j = (i + 1) % n;
while((j!=i) && !waiting[j])
    j = (j+1)%n;
if (j==i)
    lock = false;
else
    waiting[j] = false;
        // remainder section
}while(true);
```

Boolean waiting[n];
Boolean lock;

n=2 P_0 P_1

	i	Waiting	lock	Key	j
P_0	0	T F	F T	T F	1
P_1	1	T T F F	F T F	T	0 1

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

Complete Solution using Test And Set

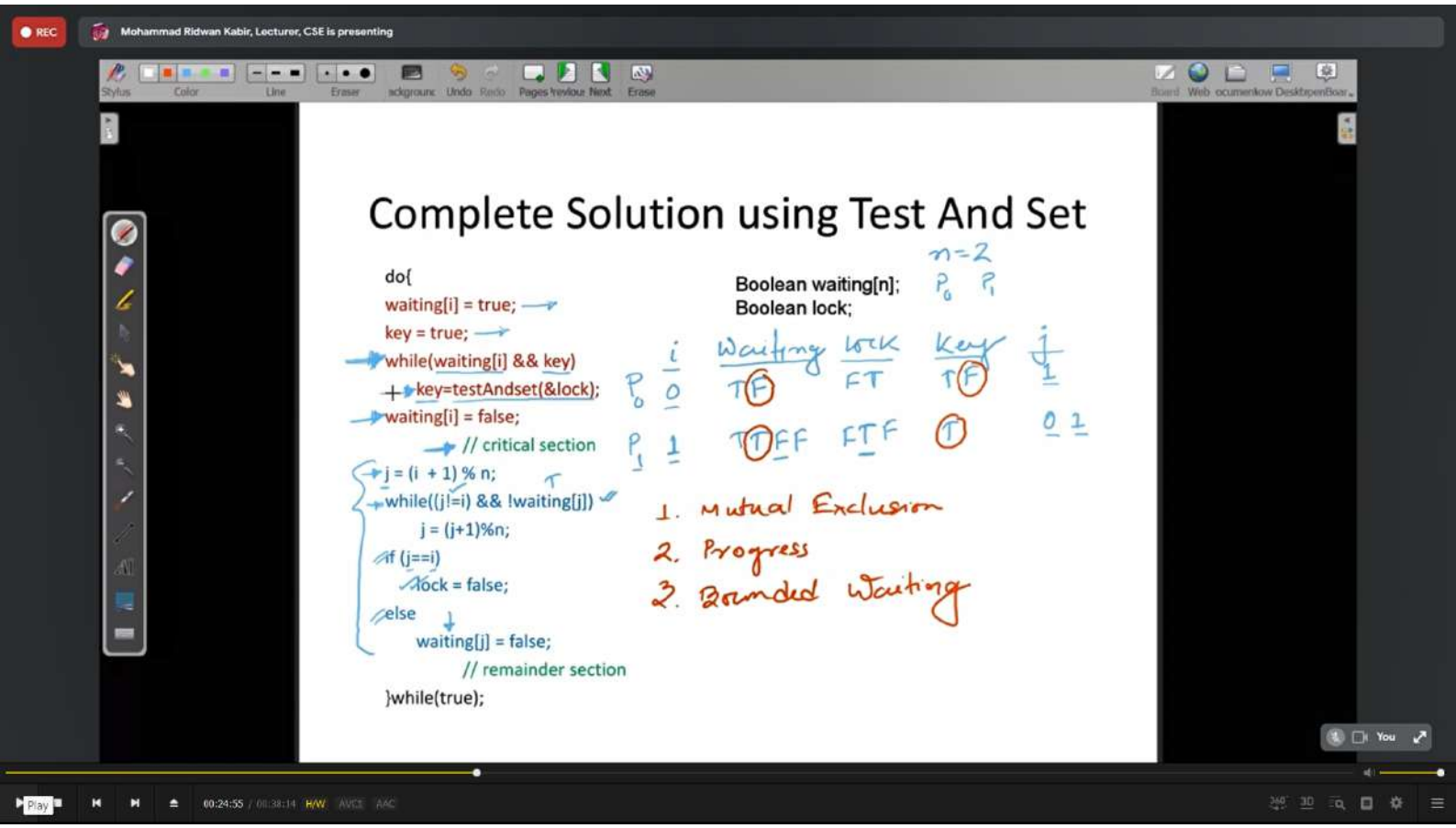
```
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
    key=testAndset(&lock);
waiting[i] = false;
        // critical section
j = (i + 1) % n;
while((j!=i) && !waiting[j])
    j = (j+1)%n;
if (j==i)
    lock = false;
else
    waiting[j] = false;
        // remainder section
}while(true);
```

Boolean waiting[n];
Boolean lock;

n=2 ; P_0 P_1

	i	Waiting	lock	Key	j
P_0	0	T F	F T	T F	1
P_1	1	T T F F	F T F	T	0 1

1. Mutual Exclusion
2. Progress
3. Bounded Waiting


REC
Mohammad Ridwan Kabir, Lecturer, CSE is presenting
Complete Solution using Test And Set
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
key=testAndset(&lock);
waiting[i] = false;
// critical section
j = (i + 1) % n;
while((j!=i) && !waiting[j])
j = (j+1)%n;
if (j==i)
lock = false;
else
waiting[j] = false;
// remainder section
}while(true);
Boolean waiting[n];
Boolean lock;
n=2
1. Mutual Exclusion
2. Progress
3. Bounded Waiting
You
Play
00:24:55 / 01:38:14

Mutex Locks

- Calls to either acquire() or release() must be done atomically.
- Main disadvantage is busy waiting.
- When one process is in its critical section, others must loop continuously in the call to acquire().
- This type of mutex lock is also known as spin lock as the process literally spins while waiting for the lock to become available.
  - Spin lock is a problem in real multiprogramming situation, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other processes might be able to use productively.
  - One advantage of spin lock is that, no context switching is required when a process must wait on a lock. Thus, when locks are expected to be held for a short time, spin locks are useful.
  - They are often employed on multi-processor systems where one thread can spin on one processor while another thread performs its critical section on another processor.

Semaphore Usage

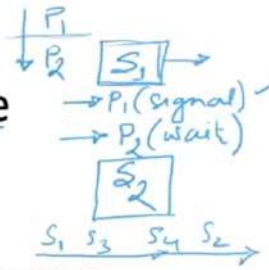
- Consider two concurrently running processes
  - P_1 with statement S_1
  - P_2 with statement S_2
- Suppose we require S_2 to be executed only when S_1 has finished execution.
- To do this, we allow P_1 and P_2 to share a semaphore synch initialized to 0 and the processes P_1 and P_2 are defined as follows.

```
P1(){
   S1;
   signal(synch);
}
```

```
P2(){
   wait(synch);
   S2;
}
```

- Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

Semaphore Usage



- Consider two concurrently running processes
  - P_1 with statement S_1
  - P_2 with statement S_2
- Suppose we require S_2 to be executed only when S_1 has finished execution.
- To do this, we allow P_1 and P_2 to share a semaphore synch initialized to 0 and the processes P_1 and P_2 are defined as follows.

```
P1(){
    S1;
    signal(synch);
}
```

```
P2(){
    wait(synch);
    S2;
}
```

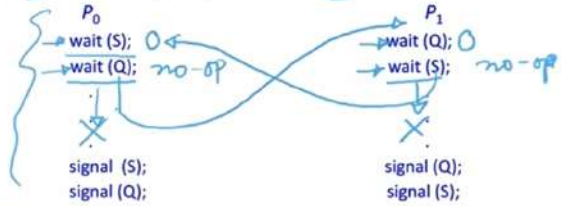
- Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

while (S<=0);

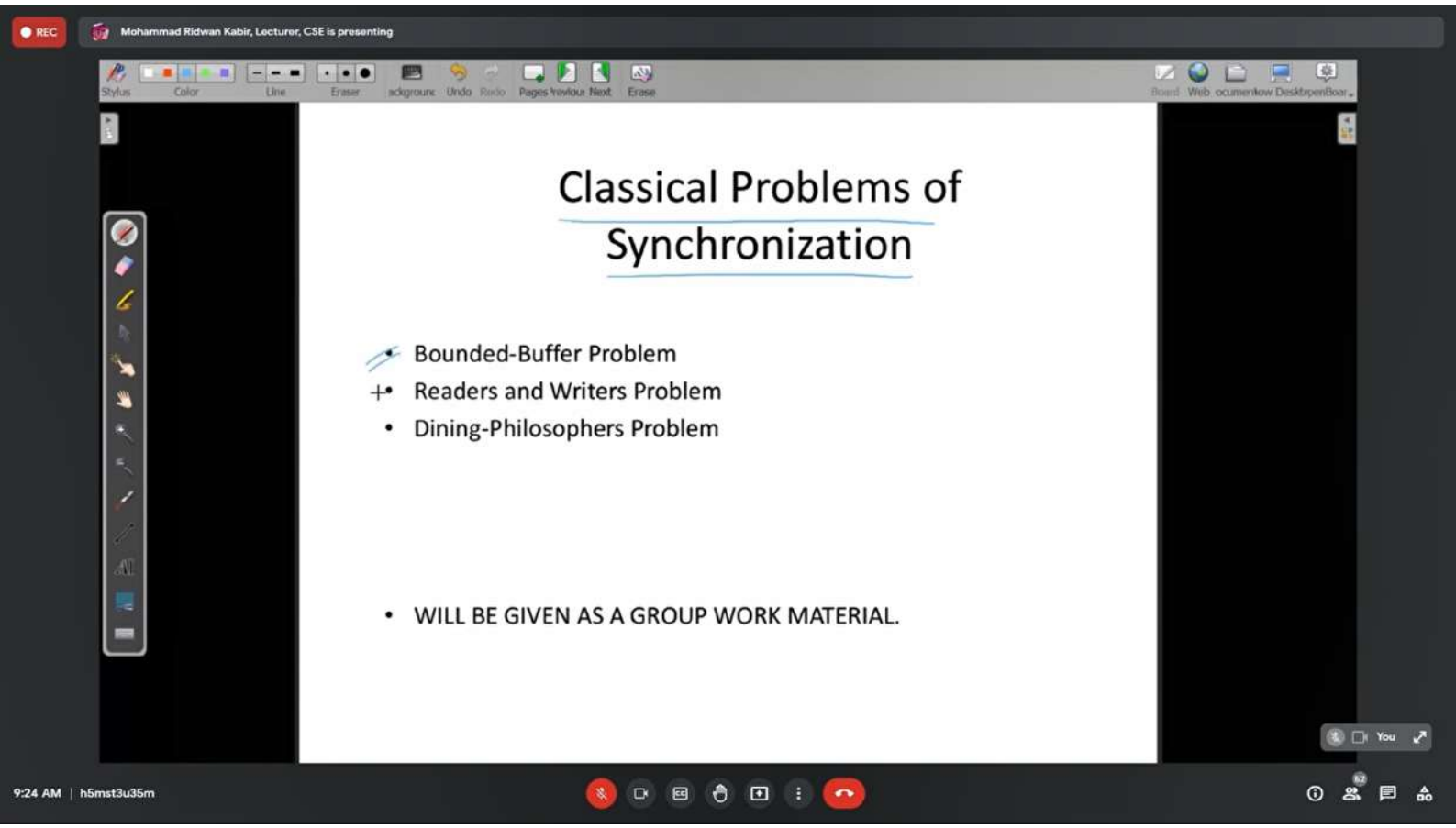
S--
S++

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1



- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.


REC
Mohammad Ridwan Kabir, Lecturer, CSE is presenting
Classical Problems of Synchronization
Bounded-Buffer Problem
Readers and Writers Problem
Dining-Philosophers Problem
WILL BE GIVEN AS A GROUP WORK MATERIAL.
You
9:24 AM | h5mst3u35m