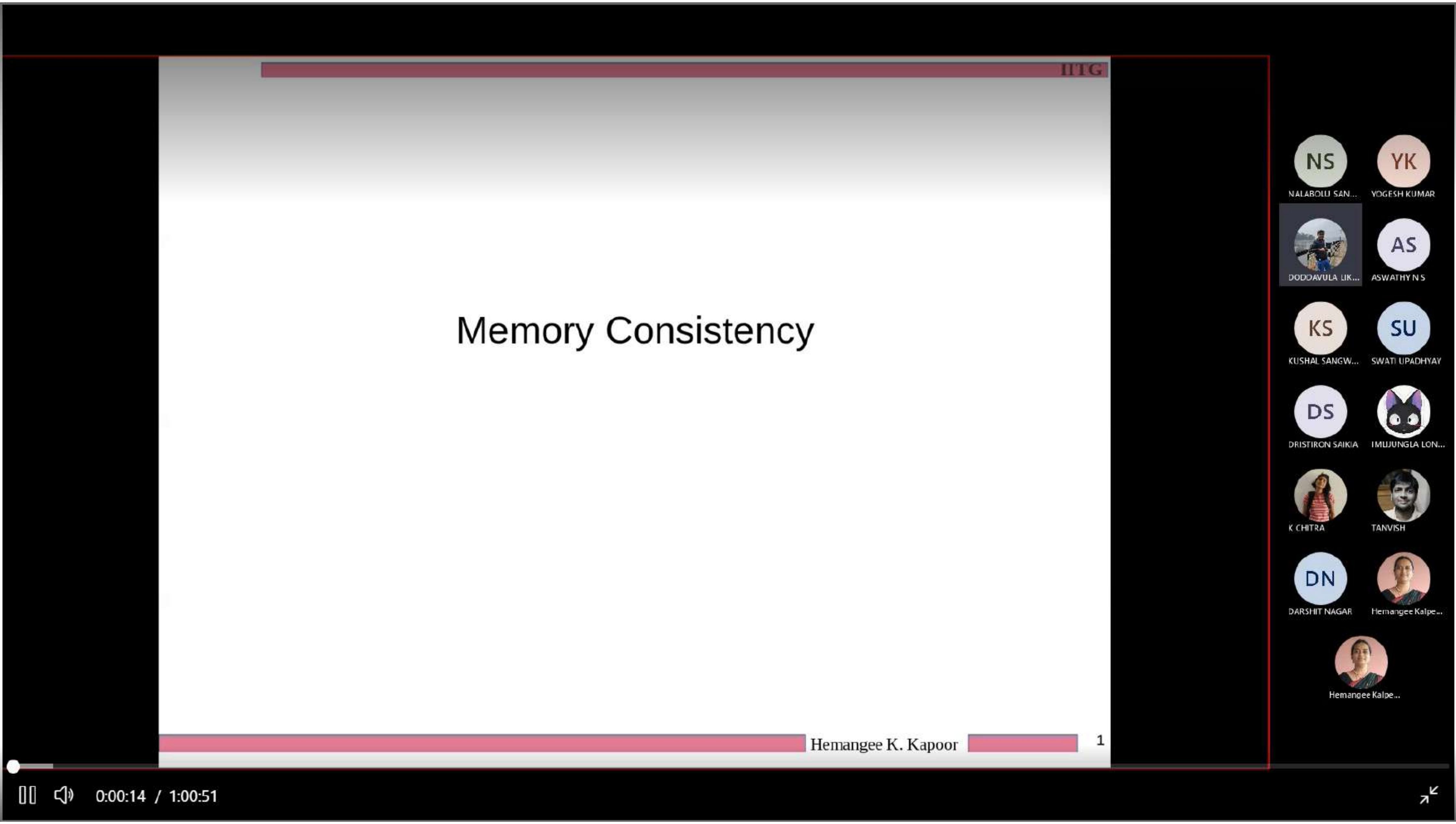
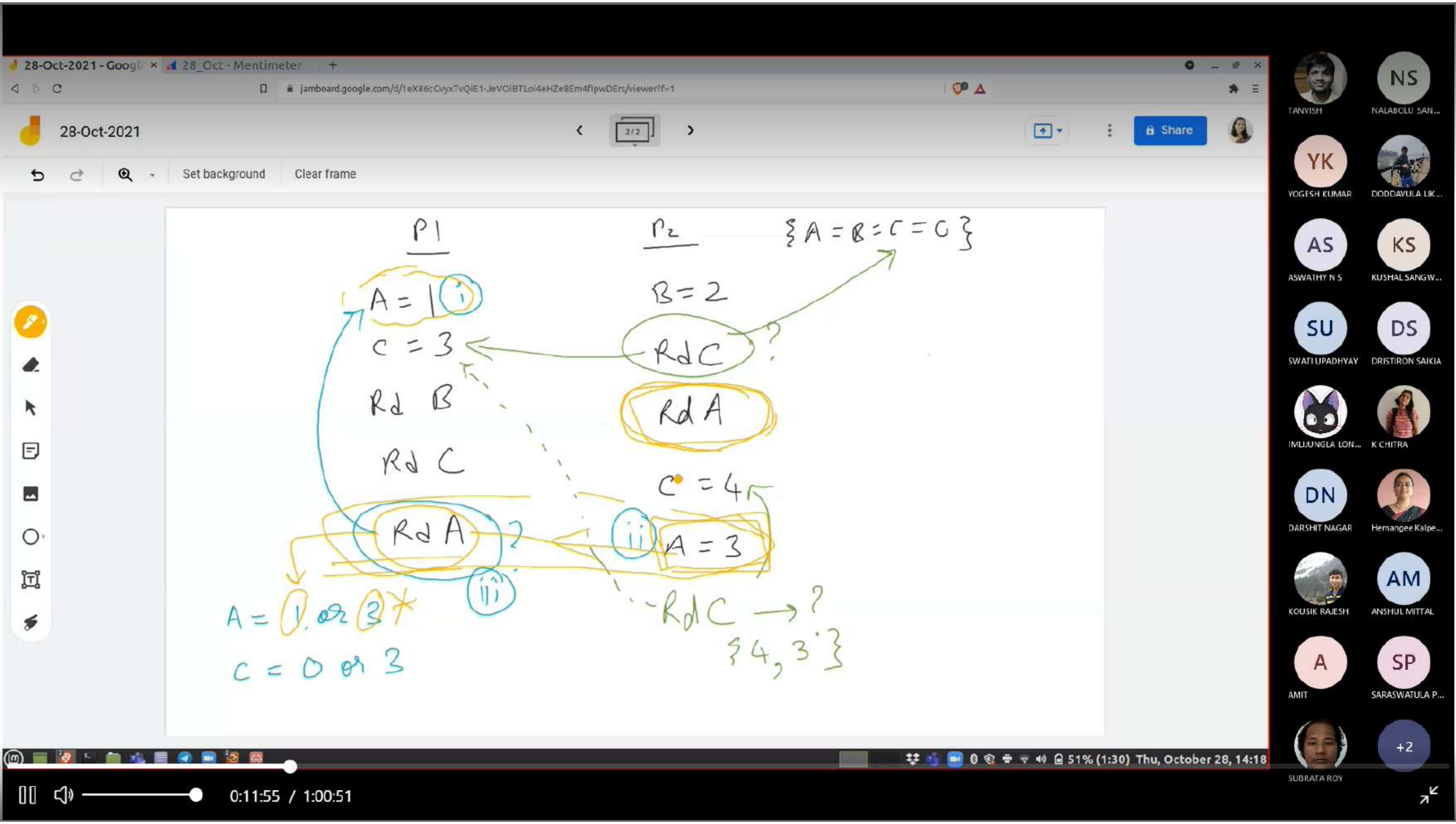
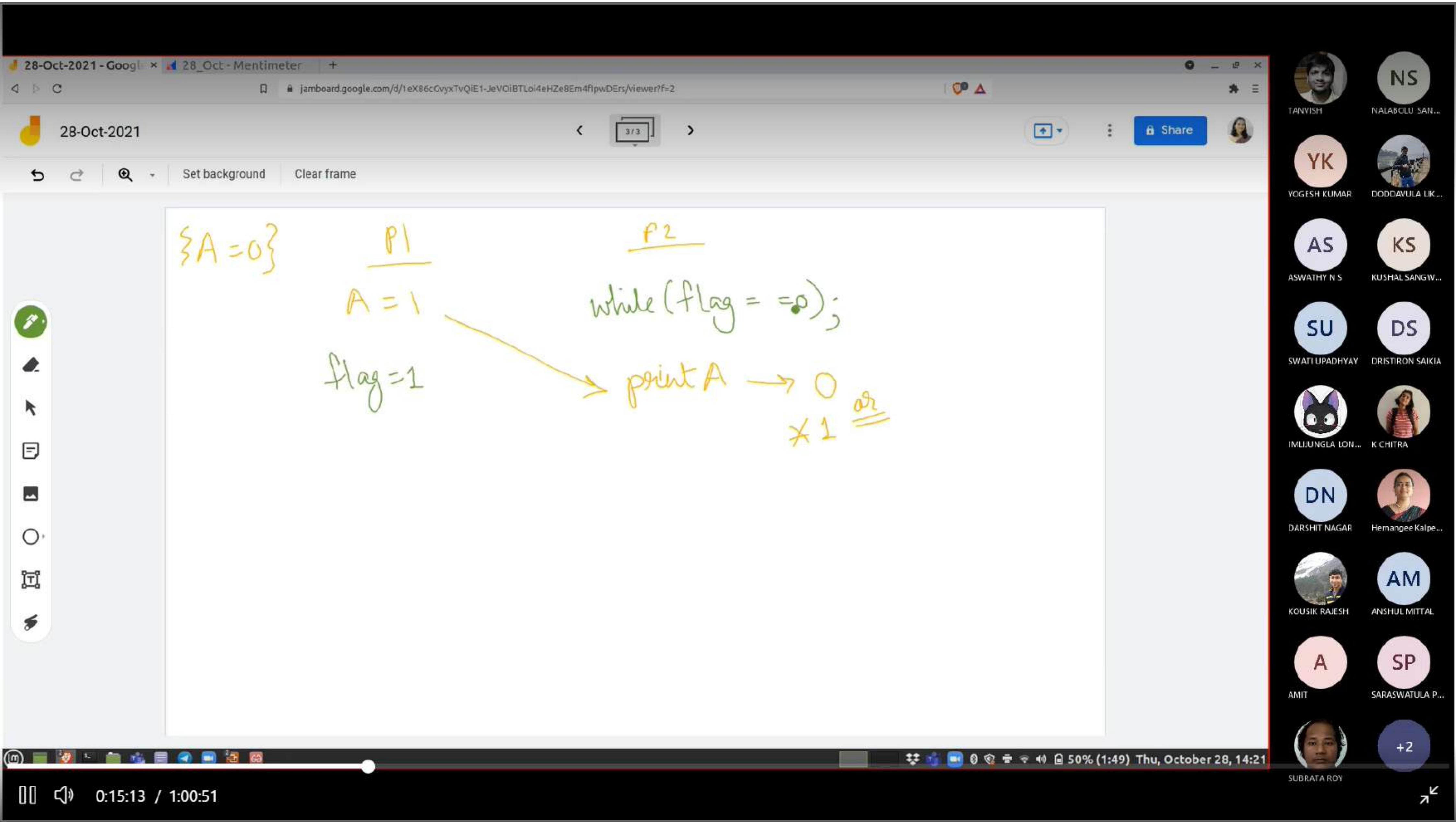


Memory Consistency


28-Oct-2021 - Google
28_Oct - Mentimeter
jamboard.google.com/d/1eX86cGvyxTvQiE1-JeVOIBTLoi4eHZe8Em4flpwDErs/viewer?f=1
28-Oct-2021
2/2
Share
Set background
Clear frame
P1
P2
{A = B = C = 0}
A = 1 (i)
C = 3
Rd B
Rd C
Rd A ? (iii)
B = 2
Rd C ?
Rd A
C = 4
(ii) A = 3
Rd C → ?
{4, 3}
A = 1 or 3 ✗
C = 0 or 3
51% (1:30) Thu, October 28, 14:18
0:11:55 / 1:00:51
TANVISH
NALABOLU SAN...
YOGESH KUMAR
DODDAVULA LIK...
ASWATHY N S
KUSHAL SANGW...
SWATI UPADHYAY
DRISTIRON SAIKIA
IMLIJUNGLA LON...
K CHITRA
DARSHIT NAGAR
Hemangee Kalpe...
KOUSIK RAJESH
ANSHUL MITTAL
AMIT
SARASWATULA P...
SUBRATA ROY
+2



{A = 0}

P1

A = 1

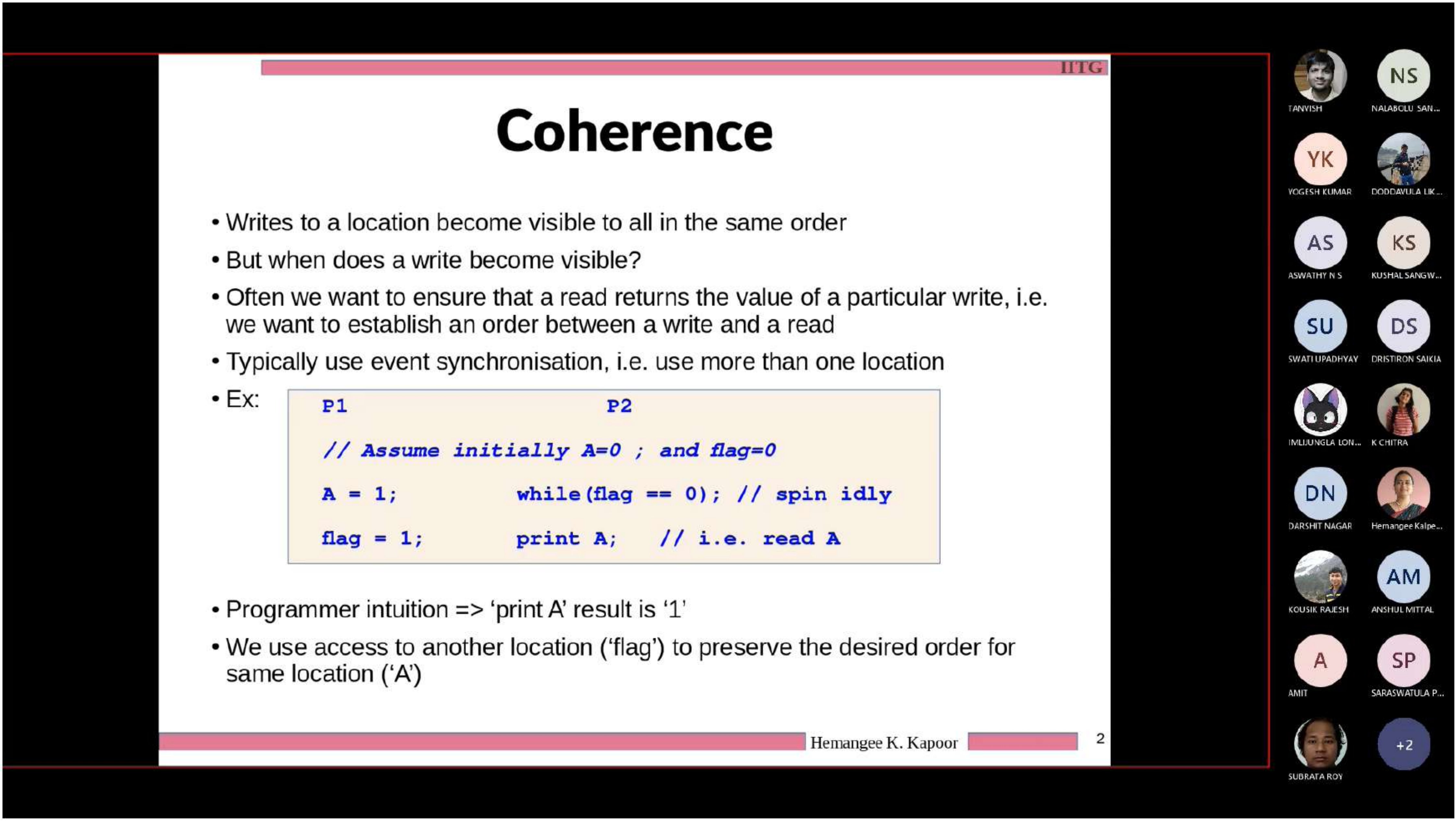
flag = 1

P2

while (flag = = 0);

print A → 0

~~1~~ or

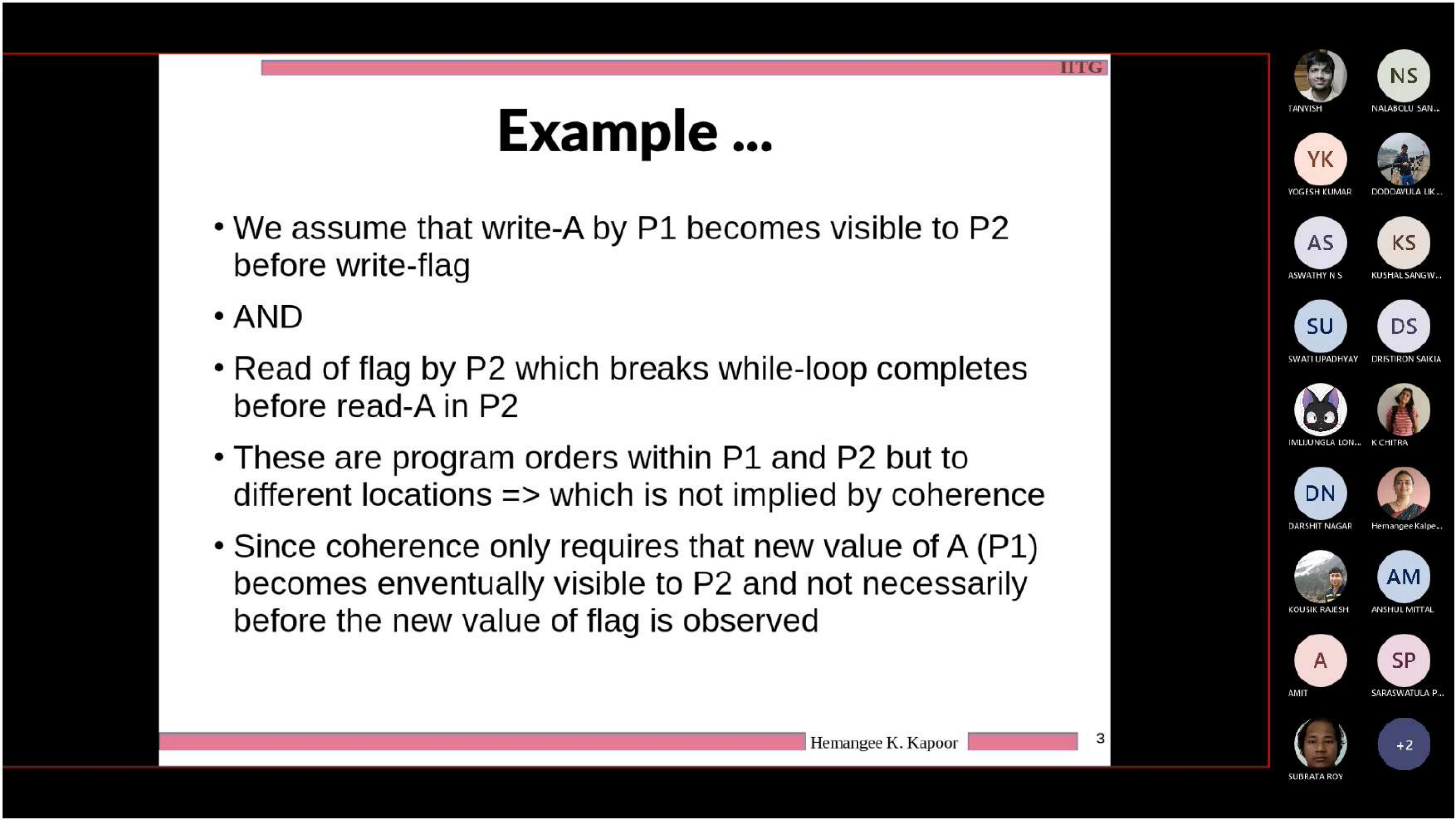


Coherence

- Writes to a location become visible to all in the same order
- But when does a write become visible?
- Often we want to ensure that a read returns the value of a particular write, i.e. we want to establish an order between a write and a read
- Typically use event synchronisation, i.e. use more than one location
- Ex:

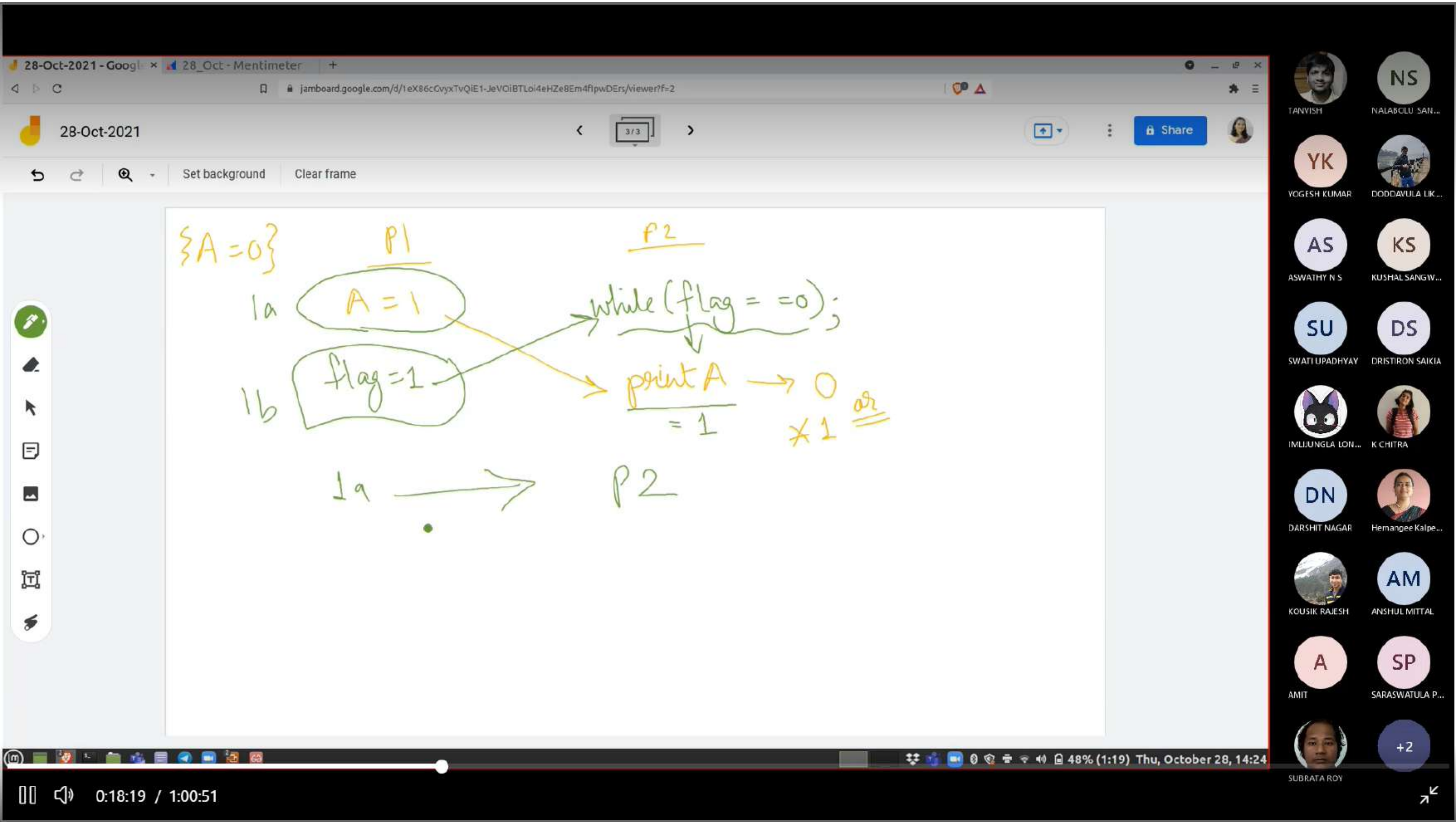
```
P1                     P2
// Assume initially A=0 ; and flag=0
A = 1;          while(flag == 0); // spin idly
flag = 1;       print A;    // i.e. read A
```

- Programmer intuition => ‘print A’ result is ‘1’
- We use access to another location (‘flag’) to preserve the desired order for same location (‘A’)



Example ...

- We assume that write-A by P1 becomes visible to P2 before write-flag
- AND
- Read of flag by P2 which breaks while-loop completes before read-A in P2
- These are program orders within P1 and P2 but to different locations => which is not implied by coherence
- Since coherence only requires that new value of A (P1) becomes enventually visible to P2 and not necessarily before the new value of flag is observed



{A=0}

P1

1a A=1

1b flag=1

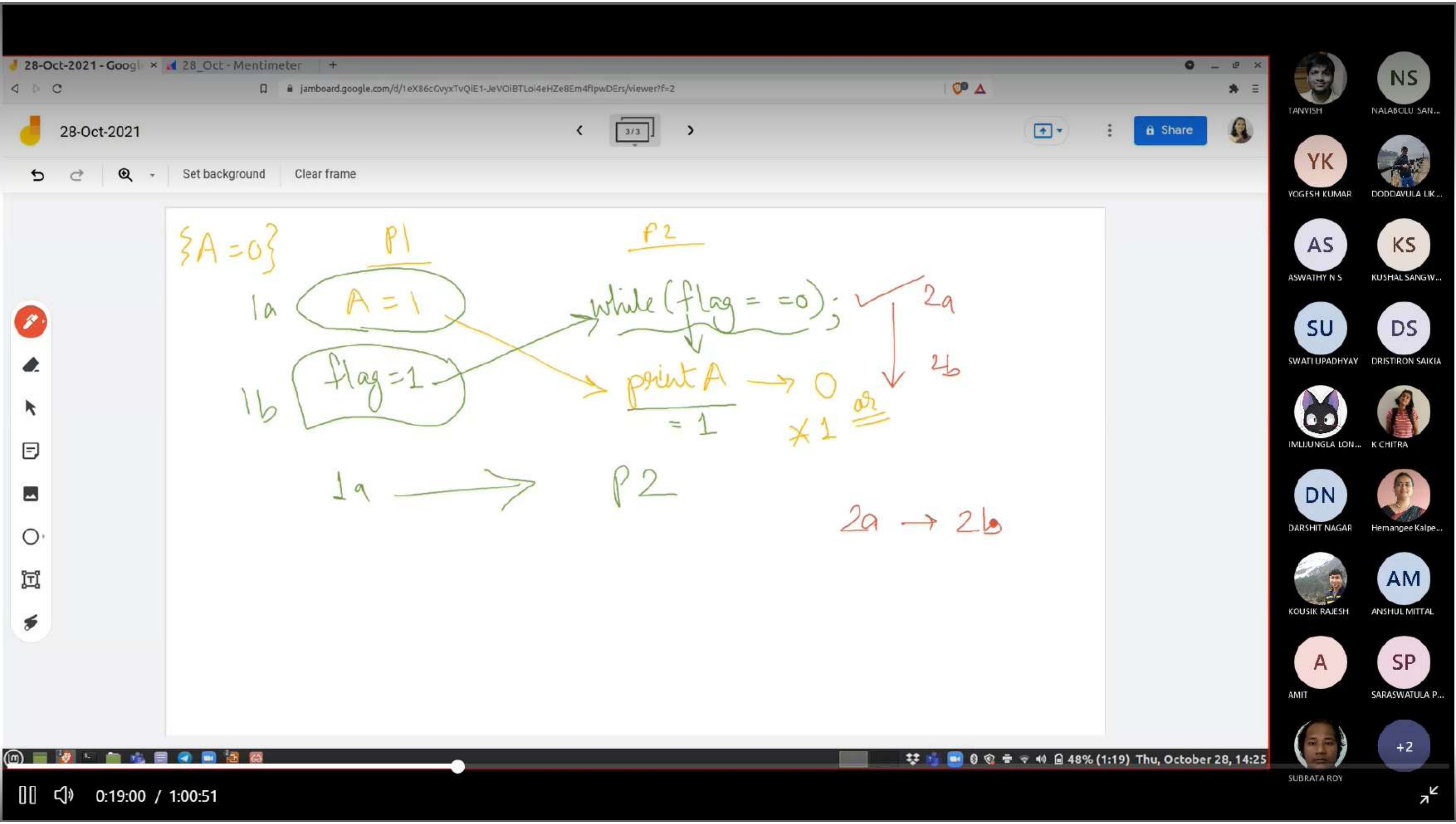
P2

while(flag = =0);

print A → 0 or

= 1 ✗1

1a → P2



{A=0}

P1

1a A=1

1b flag=1

P2

```
while(flag==0);
print A
```

$\text{print } A \rightarrow 0$ or $\cancel{1}$

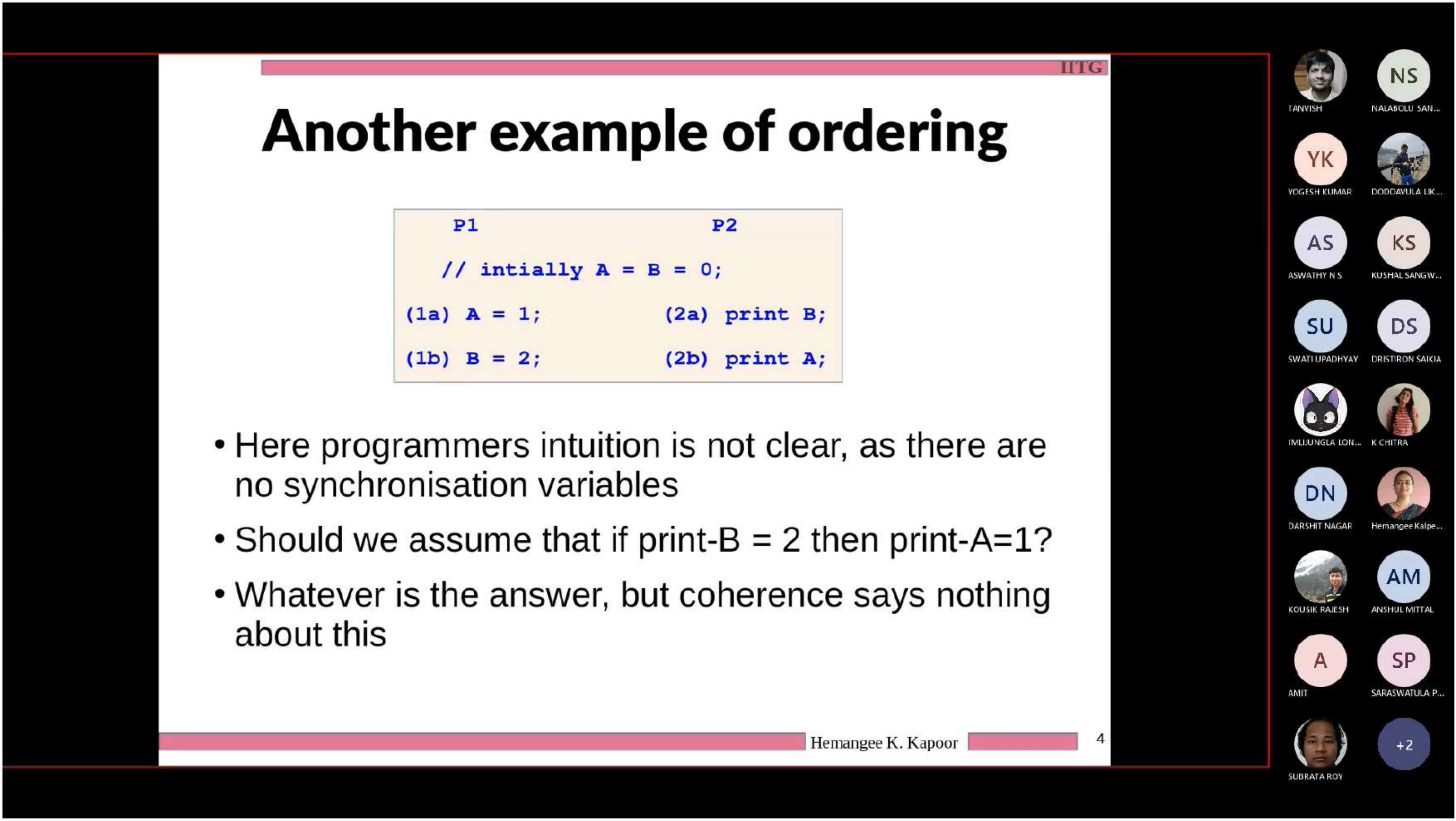
= 1

2a

2b

$1a \longrightarrow P2$

$2a \rightarrow 2b$



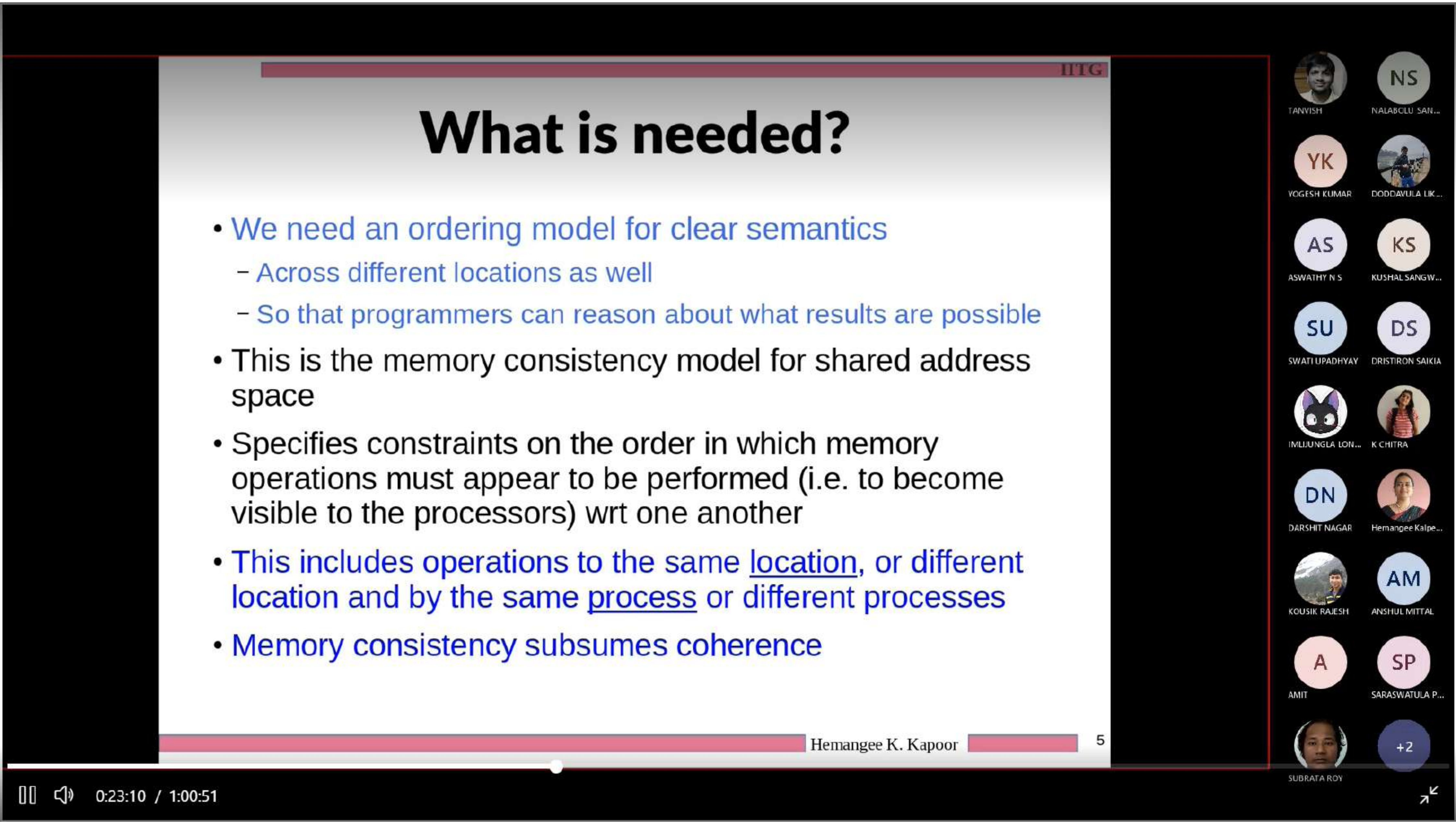
IITG

Another example of ordering

```
    P1                     P2

   // intially A = B = 0;

(1a)  A = 1;            (2a)  print B;

(1b)  B = 2;            (2b)  print A;
```

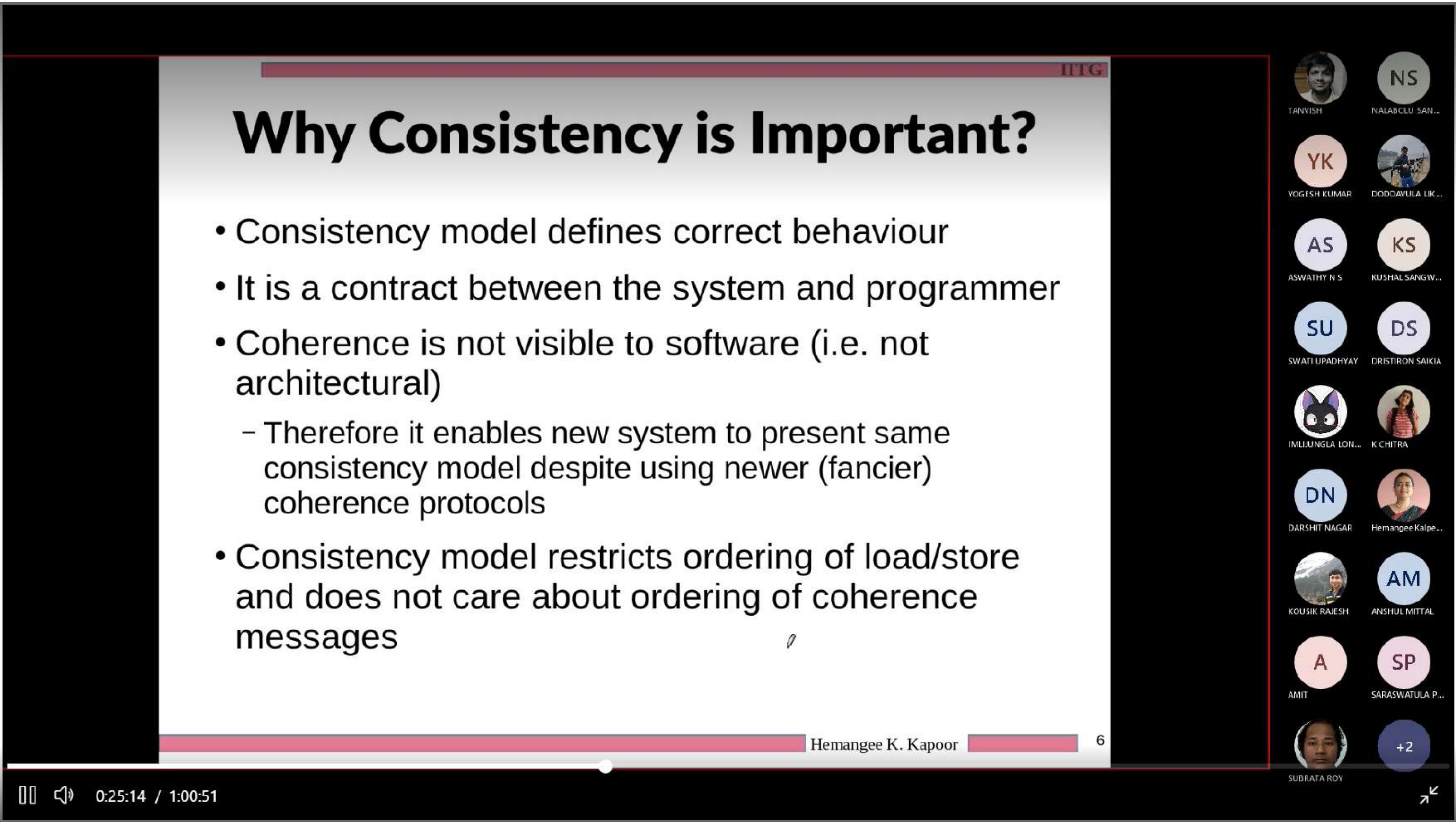
- Here programmers intuition is not clear, as there are no synchronisation variables
- Should we assume that if print-B = 2 then print-A=1?
- Whatever is the answer, but coherence says nothing about this

Hemangee K. Kapoor



What is needed?

- We need an ordering model for clear semantics
  - Across different locations as well
  - So that programmers can reason about what results are possible
- This is the memory consistency model for shared address space
- Specifies constraints on the order in which memory operations must appear to be performed (i.e. to become visible to the processors) wrt one another
- This includes operations to the same location, or different location and by the same process or different processes
- Memory consistency subsumes coherence

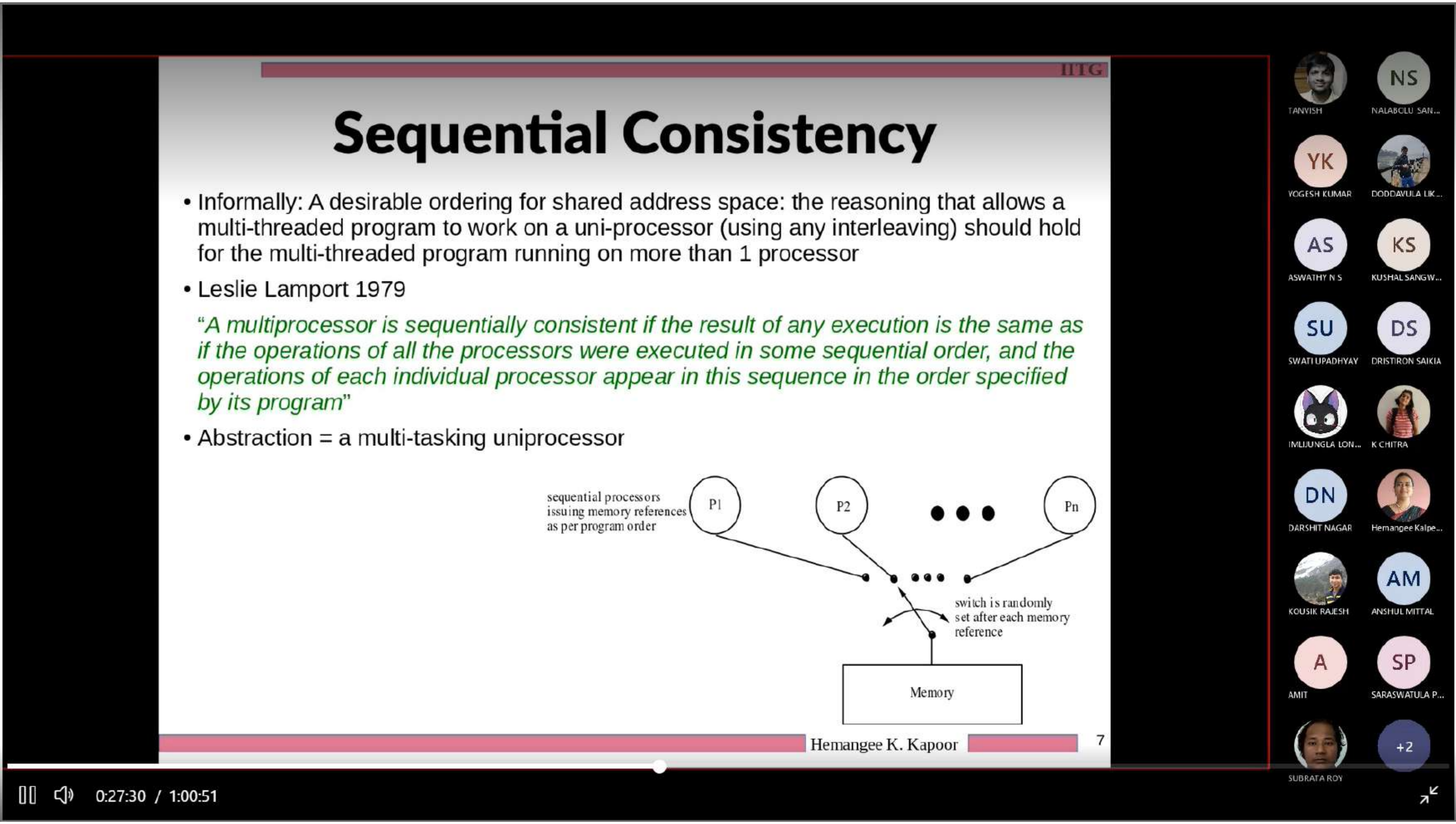


IITG

Why Consistency is Important?

- Consistency model defines correct behaviour
- It is a contract between the system and programmer
- Coherence is not visible to software (i.e. not architectural)
  - Therefore it enables new system to present same consistency model despite using newer (fancier) coherence protocols
- Consistency model restricts ordering of load/store and does not care about ordering of coherence messages

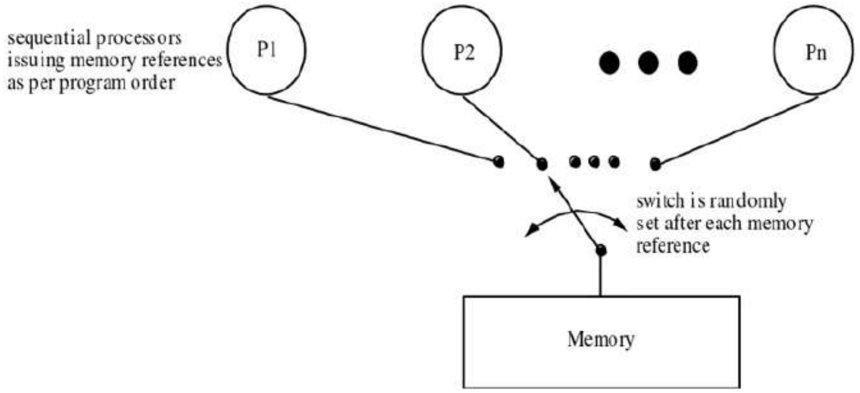
Hemangee K. Kapoor 6

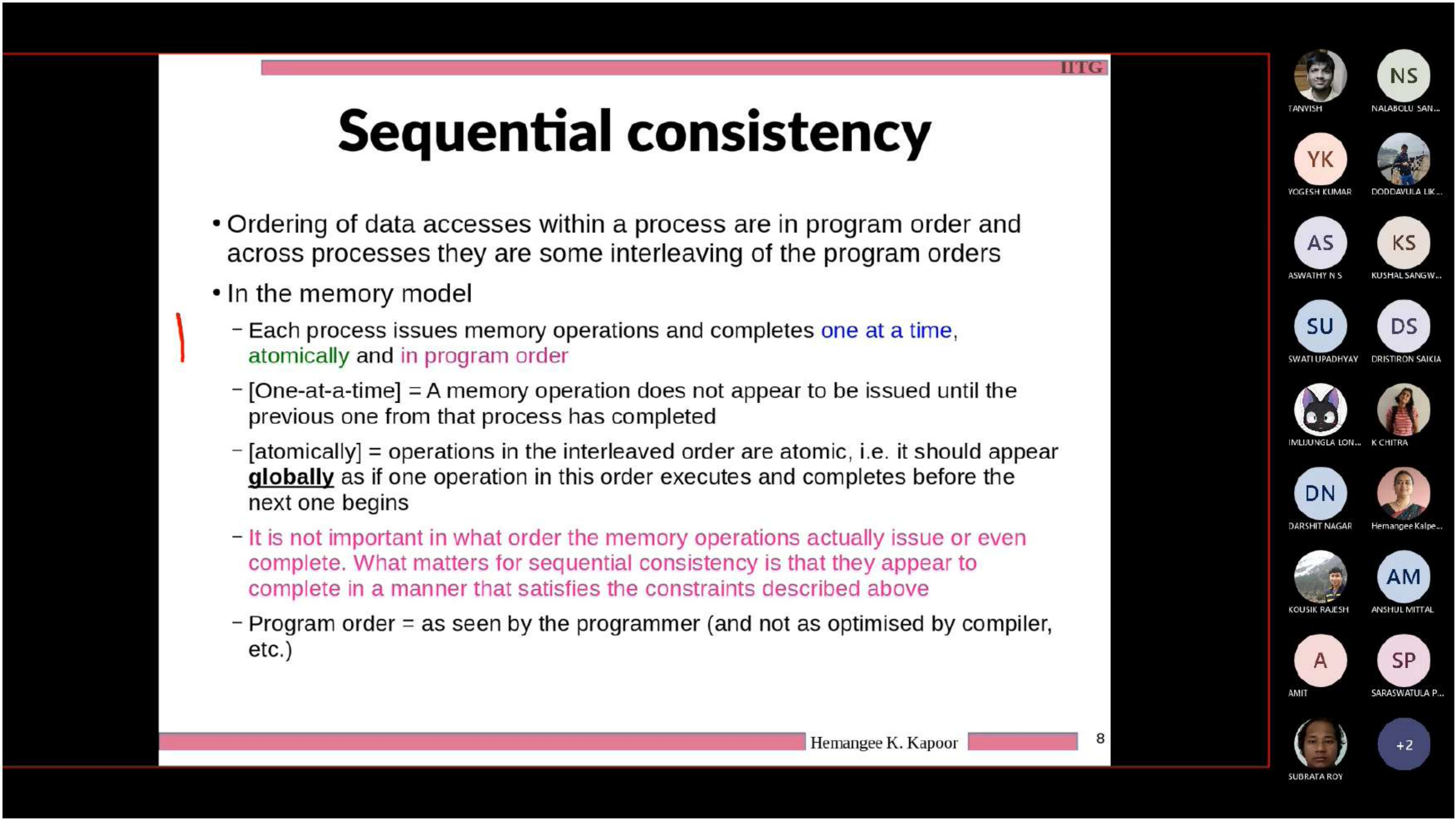


Sequential Consistency

- Informally: A desirable ordering for shared address space: the reasoning that allows a multi-threaded program to work on a uni-processor (using any interleaving) should hold for the multi-threaded program running on more than 1 processor
- Leslie Lamport 1979

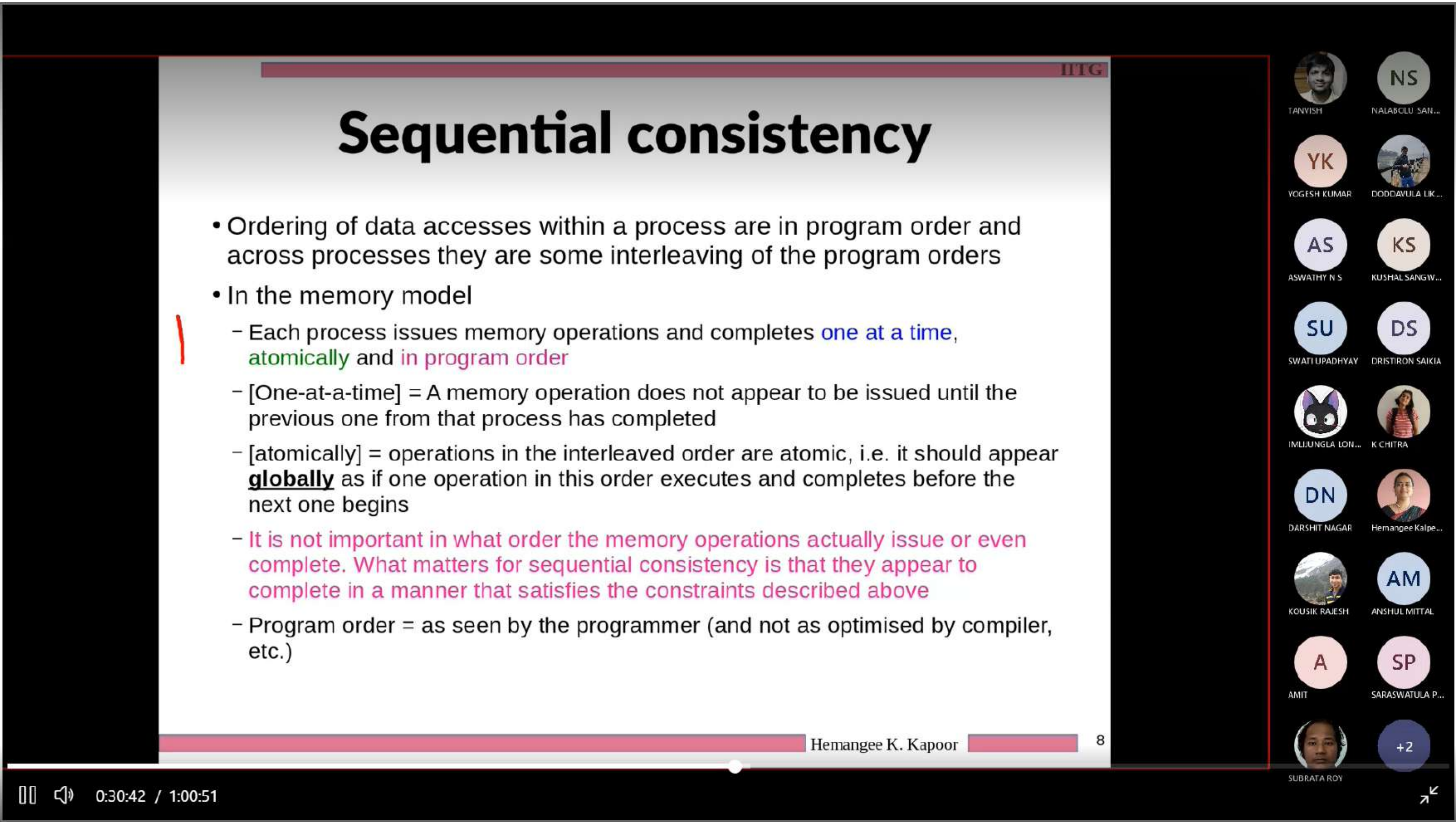
  "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"
- Abstraction = a multi-tasking uniprocessor


sequential processors issuing memory references as per program order
P1
P2
Pn
switch is randomly set after each memory reference
Memory



Sequential consistency

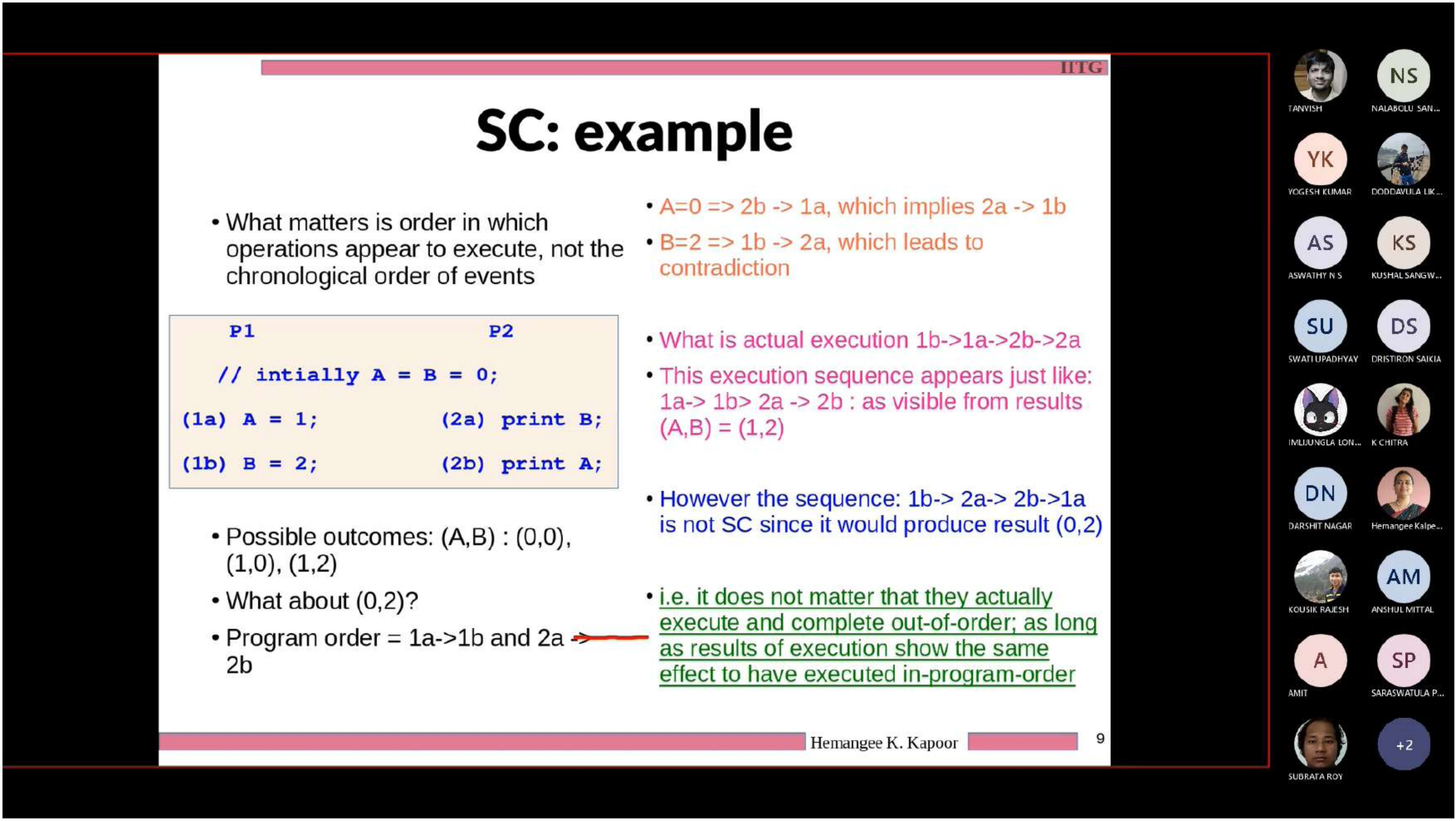
- Ordering of data accesses within a process are in program order and across processes they are some interleaving of the program orders
- In the memory model
  - Each process issues memory operations and completes one at a time, atomically and in program order
  - [One-at-a-time] = A memory operation does not appear to be issued until the previous one from that process has completed
  - [atomically] = operations in the interleaved order are atomic, i.e. it should appear **globally** as if one operation in this order executes and completes before the next one begins
  - It is not important in what order the memory operations actually issue or even complete. What matters for sequential consistency is that they appear to complete in a manner that satisfies the constraints described above
  - Program order = as seen by the programmer (and not as optimised by compiler, etc.)



IITG

Sequential consistency

- Ordering of data accesses within a process are in program order and across processes they are some interleaving of the program orders
- In the memory model
  - Each process issues memory operations and completes one at a time, atomically and in program order
  - [One-at-a-time] = A memory operation does not appear to be issued until the previous one from that process has completed
  - [atomically] = operations in the interleaved order are atomic, i.e. it should appear **globally** as if one operation in this order executes and completes before the next one begins
  - It is not important in what order the memory operations actually issue or even complete. What matters for sequential consistency is that they appear to complete in a manner that satisfies the constraints described above
  - Program order = as seen by the programmer (and not as optimised by compiler, etc.)



SC: example

- What matters is order in which operations appear to execute, not the chronological order of events

```
    P1                  P2

   // intially A = B = 0;

(1a)  A = 1;          (2a)  print B;

(1b)  B = 2;          (2b)  print A;
```

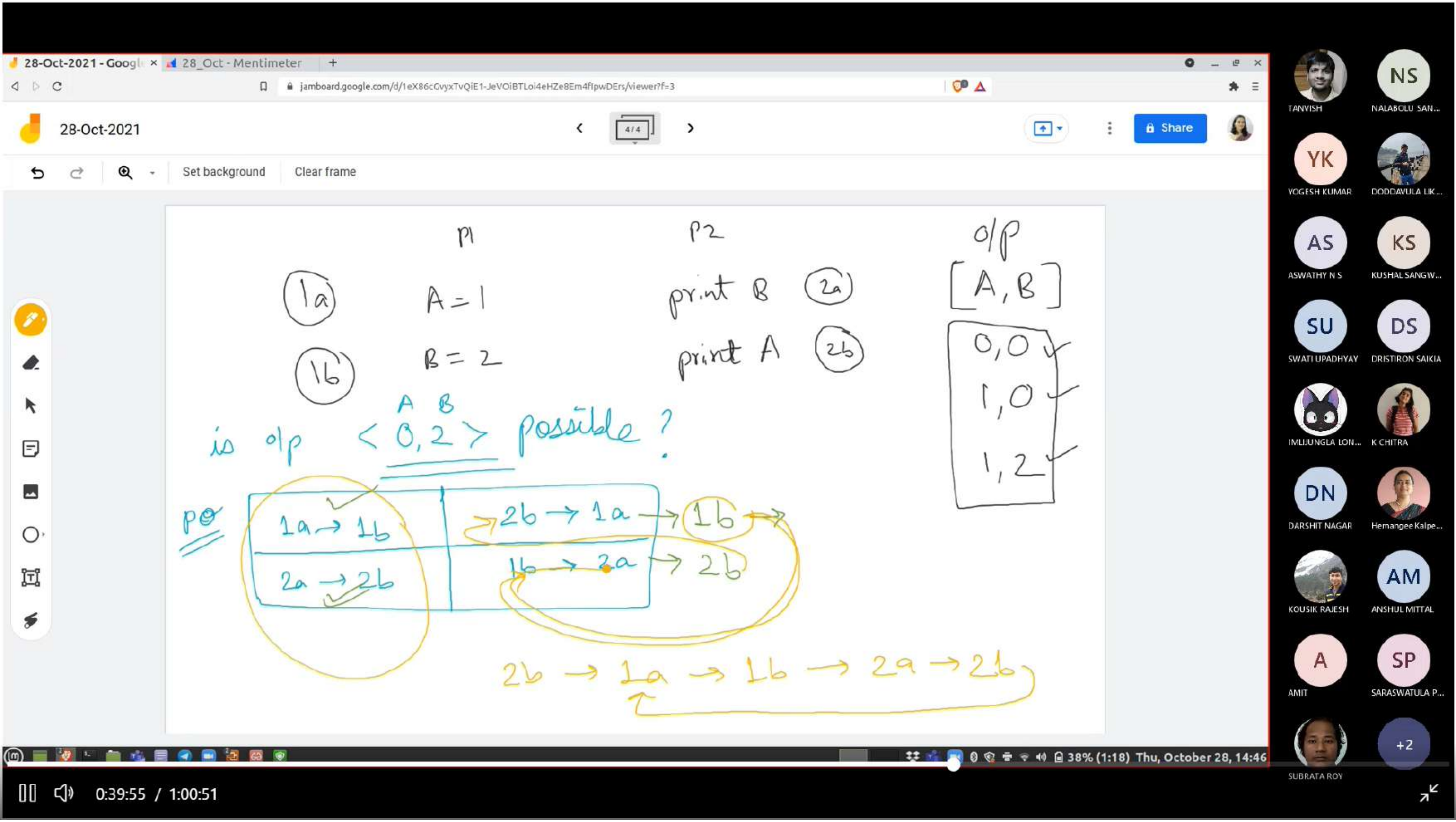
- Possible outcomes: (A,B) : (0,0), (1,0), (1,2)
- What about (0,2)?
- Program order = 1a->1b and 2a -> 2b

- A=0 => 2b -> 1a, which implies 2a -> 1b
- B=2 => 1b -> 2a, which leads to contradiction

- What is actual execution 1b->1a->2b->2a
- This execution sequence appears just like: 1a-> 1b> 2a -> 2b : as visible from results (A,B) = (1,2)

- However the sequence: 1b-> 2a-> 2b->1a is not SC since it would produce result (0,2)

- <u>i.e. it does not matter that they actually execute and complete out-of-order; as long as results of execution show the same effect to have executed in-program-order</u>



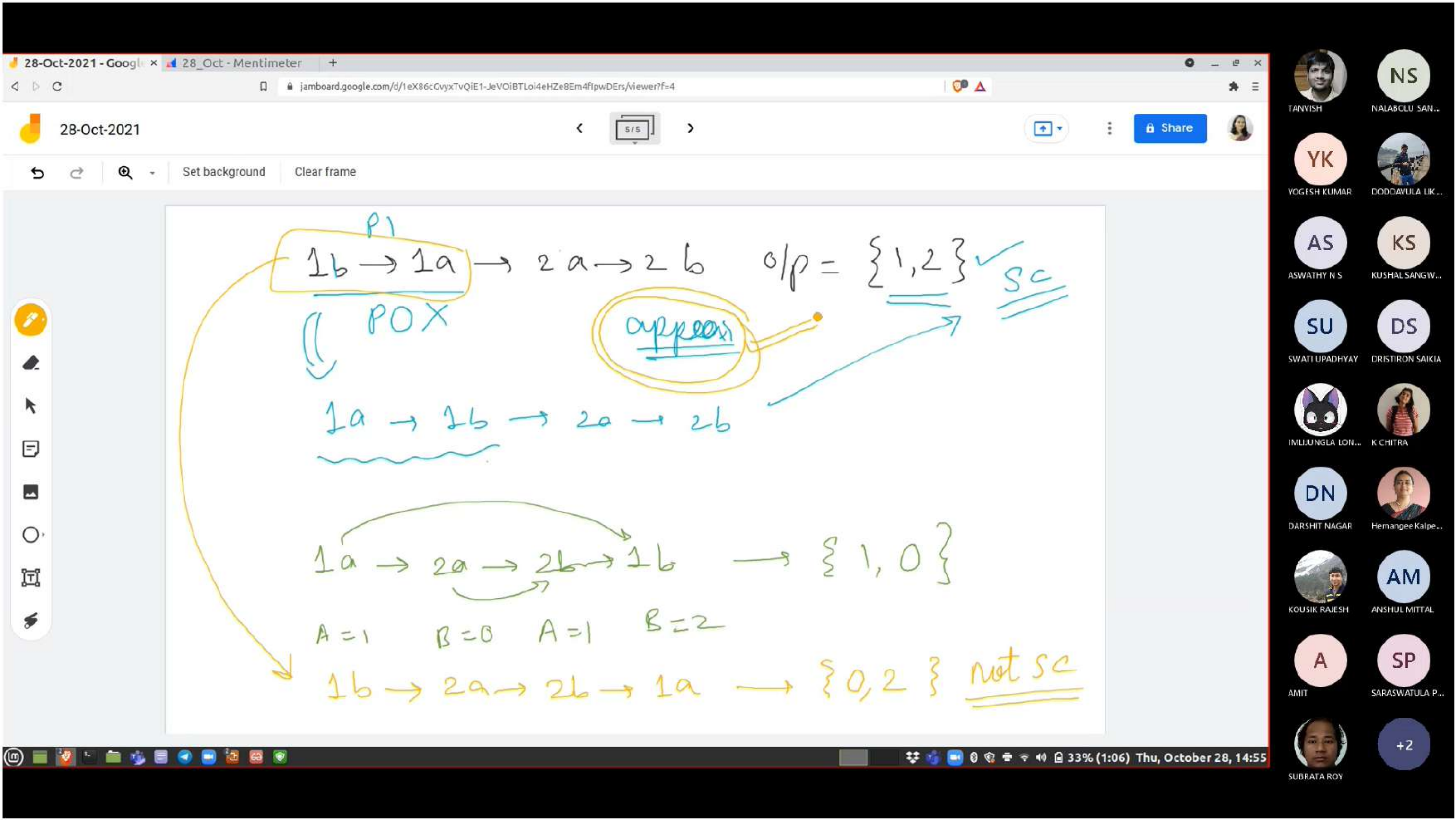
	P1	P2		O/P
1a	A = 1	print B	2a	[A, B]
1b	B = 2	print A	2b	0,0 ✓
				1,0 ✓
				1,2 ✓

is o/p $\langle \overset{A}{0}, \overset{B}{2} \rangle$ possible ?

PO

1a → 1b ✓	→ 2b → 1a → 1b →
2a → 2b ✓	1b → 2a → 2b

2b → 1a → 1b → 2a → 2b



P1

$1b \rightarrow 1a \rightarrow 2a \rightarrow 2b$ o/p = $\{1,2\}$ ✓ SC

POX

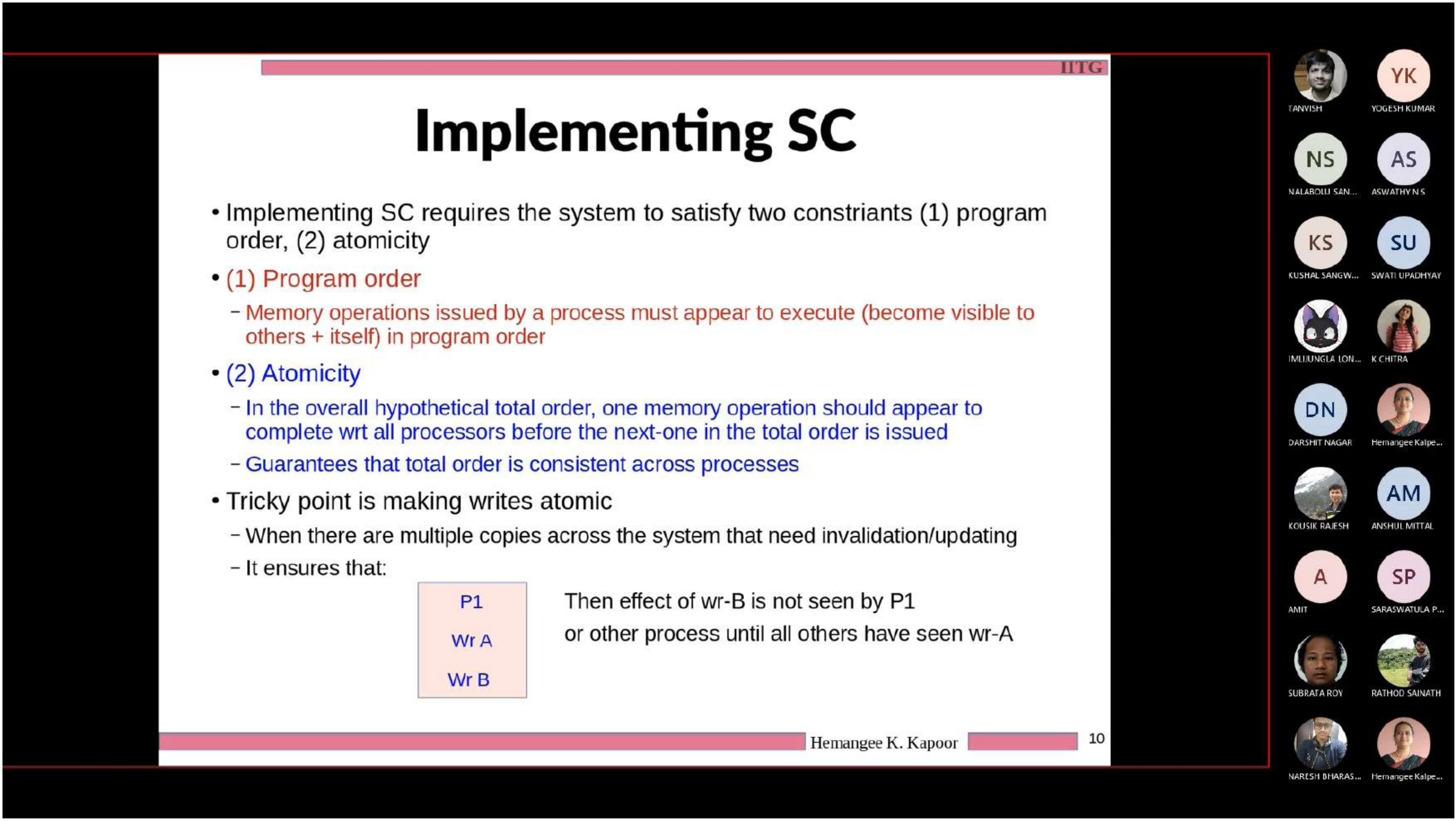
appears

$1a \rightarrow 1b \rightarrow 2a \rightarrow 2b$

$1a \rightarrow 2a \rightarrow 2b \rightarrow 1b \rightarrow \{1, 0\}$

A=1 B=0 A=1 B=2

$1b \rightarrow 2a \rightarrow 2b \rightarrow 1a \rightarrow \{0,2\}$ not SC

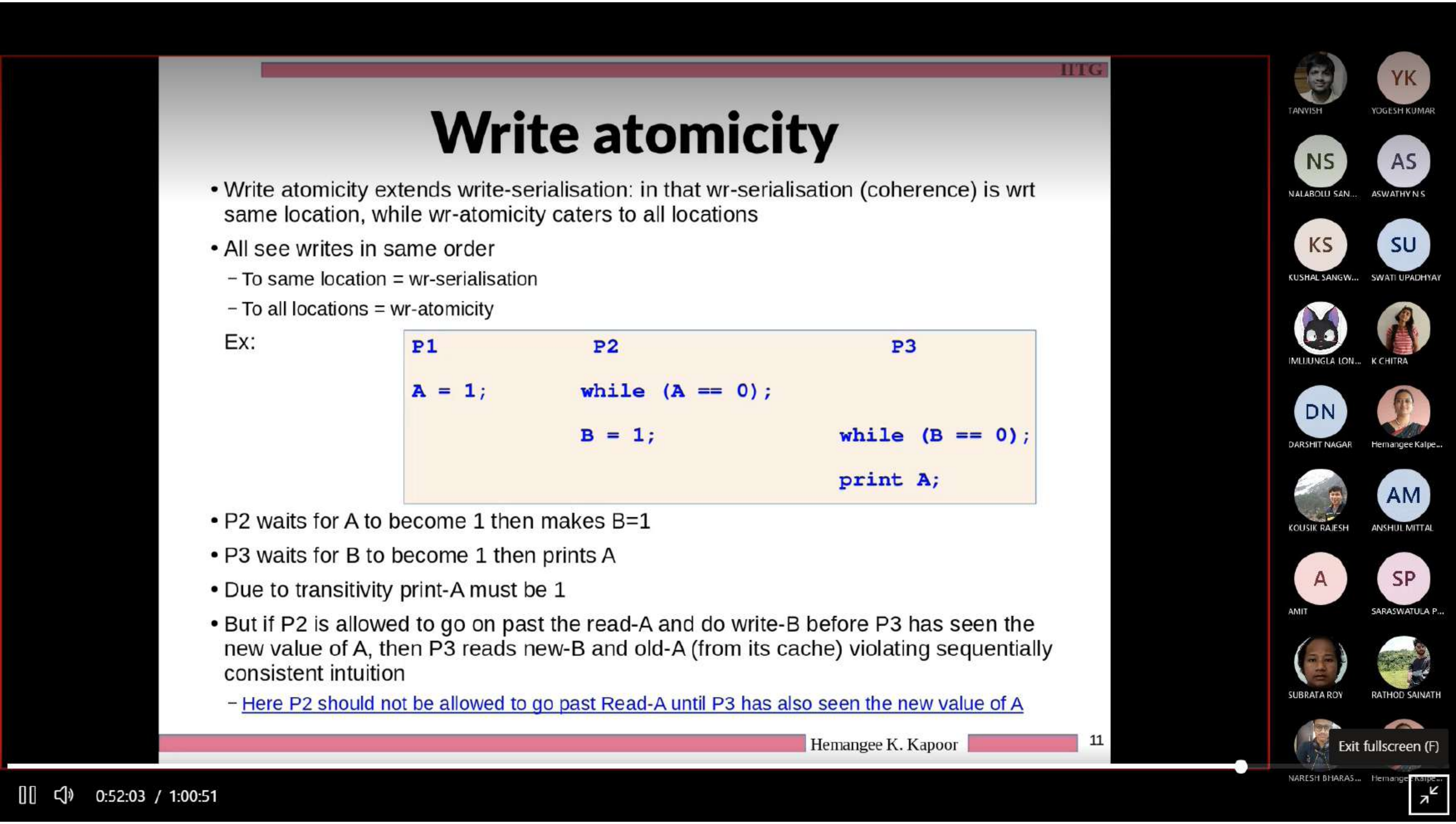


Implementing SC

- Implementing SC requires the system to satisfy two constriants (1) program order, (2) atomicity
- (1) Program order
  - Memory operations issued by a process must appear to execute (become visible to others + itself) in program order
- (2) Atomicity
  - In the overall hypothetical total order, one memory operation should appear to complete wrt all processors before the next-one in the total order is issued
  - Guarantees that total order is consistent across processes
- Tricky point is making writes atomic
  - When there are multiple copies across the system that need invalidation/updating
  - It ensures that:

P1
Wr A
Wr B

Then effect of wr-B is not seen by P1
or other process until all others have seen wr-A



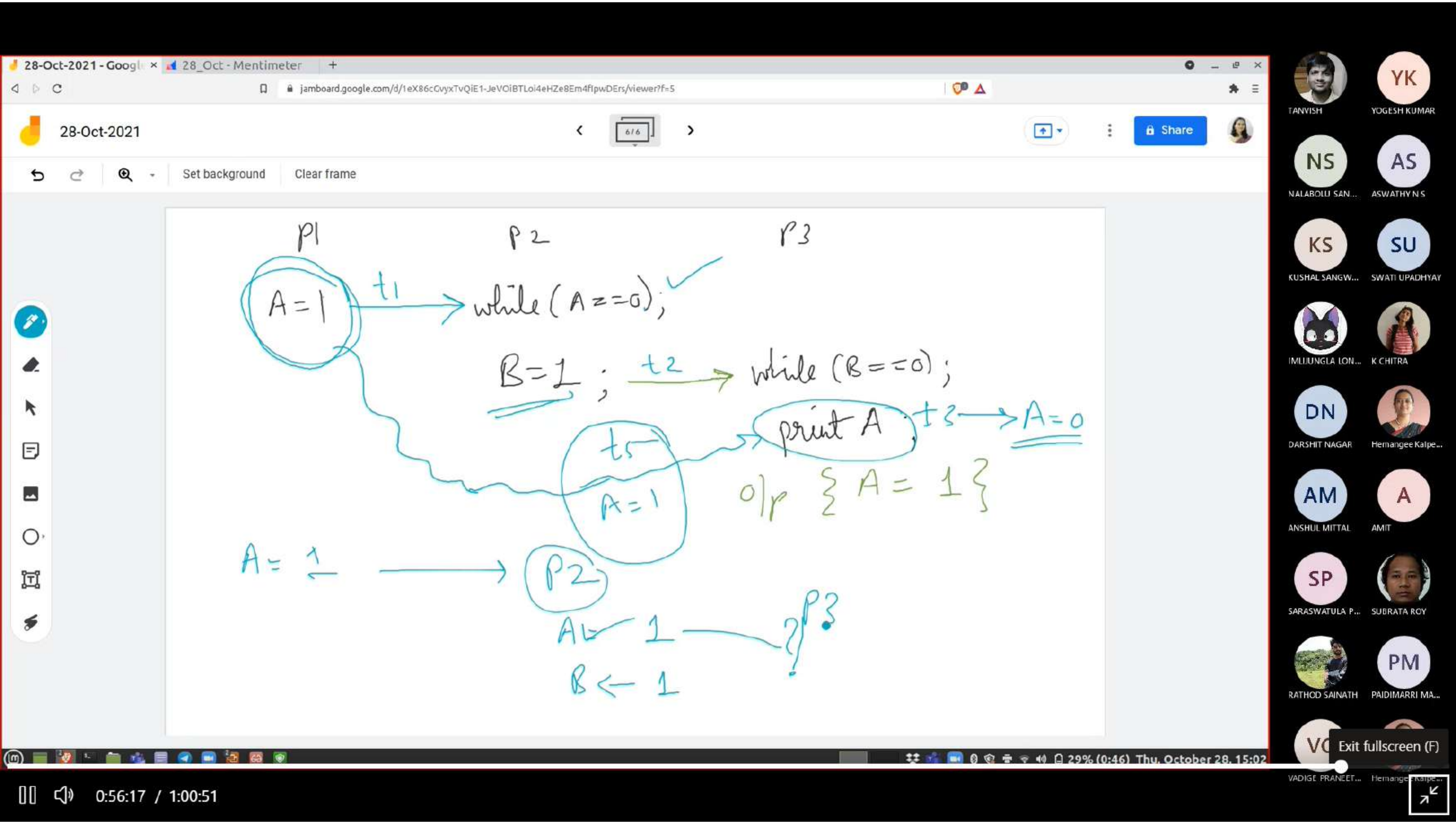
Write atomicity

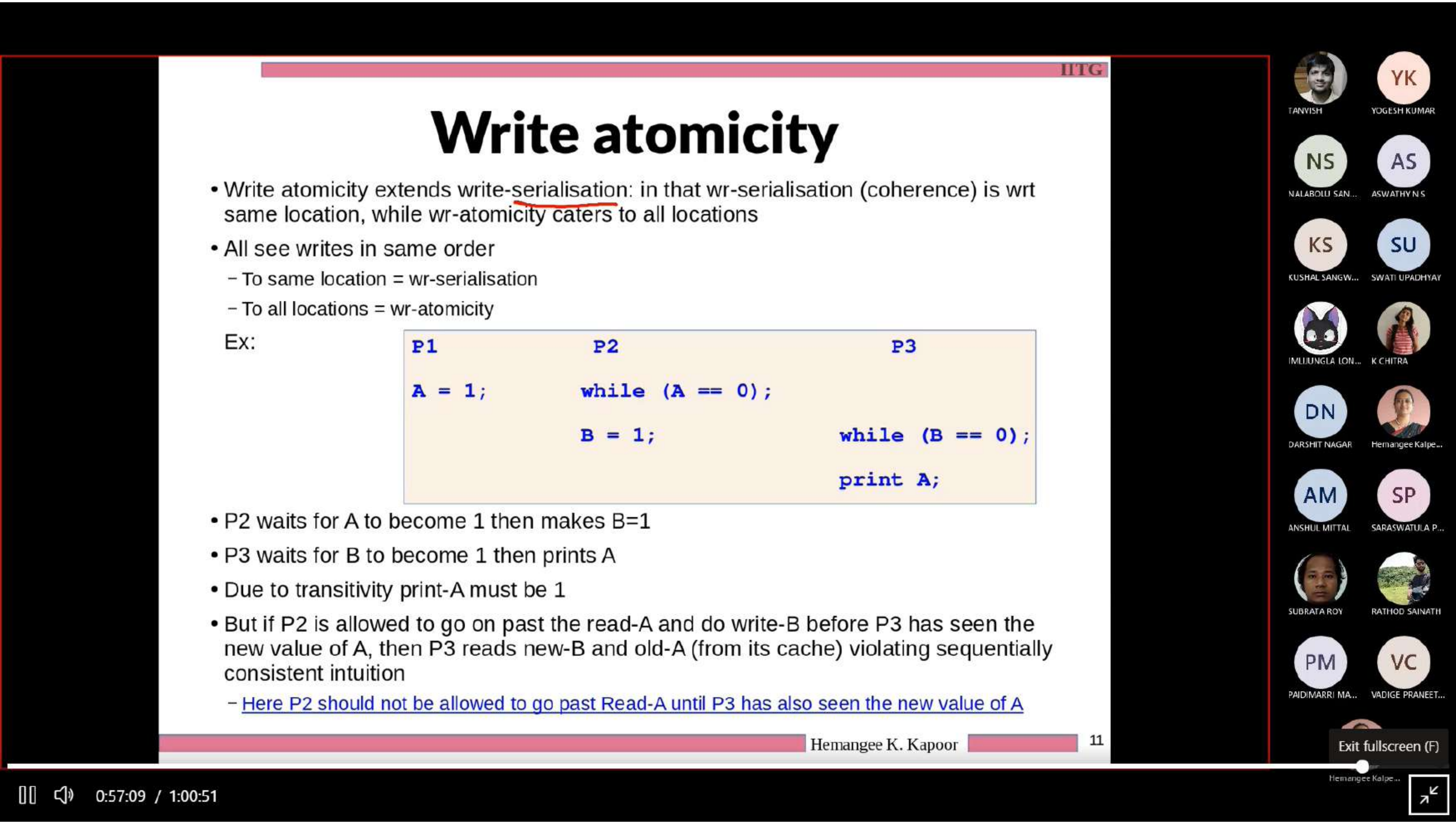
- Write atomicity extends write-serialisation: in that wr-serialisation (coherence) is wrt same location, while wr-atomicity caters to all locations
- All see writes in same order
  - To same location = wr-serialisation
  - To all locations = wr-atomicity

Ex:

```
P1          P2                    P3

A = 1;      while (A == 0);

            B = 1;                while (B == 0);

                                  print A;
```

- P2 waits for A to become 1 then makes B=1
- P3 waits for B to become 1 then prints A
- Due to transitivity print-A must be 1
- But if P2 is allowed to go on past the read-A and do write-B before P3 has seen the new value of A, then P3 reads new-B and old-A (from its cache) violating sequentially consistent intuition
  - Here P2 should not be allowed to go past Read-A until P3 has also seen the new value of A


28-Oct-2021 - Googl
28_Oct - Mentimeter
jamboard.google.com/d/1eX86cGvyxTvQiE1-JeVOiBTLoi4eHZe8Em4flpwDErs/viewer?f=5
28-Oct-2021
6/6
Share
Set background
Clear frame
P1
P2
P3
A = 1
t1
while (A == 0);
B = 1 ;
t2
while (B == 0) ;
print A ;
t3
A = 0
t5
A = 1
o/p { A = 1 }
A = 1
P2
A ← 1
B ← 1
?P3
TANVISH
YOGESH KUMAR
NALABOLU SAN...
ASWATHY N S
KUSHAL SANGW...
SWATI UPADHYAY
IMLIJUNGLA LON...
K CHITRA
DARSHIT NAGAR
Hemangee Kalpe...
ANSHUL MITTAL
AMIT
SARASWATULA P...
SUBRATA ROY
RATHOD SAINATH
PAIDIMARRI MA...
VADIGE PRANEET...
Exit fullscreen (F)
29% (0:46) Thu, October 28, 15:02
0:56:17 / 1:00:51



IITG

Write atomicity

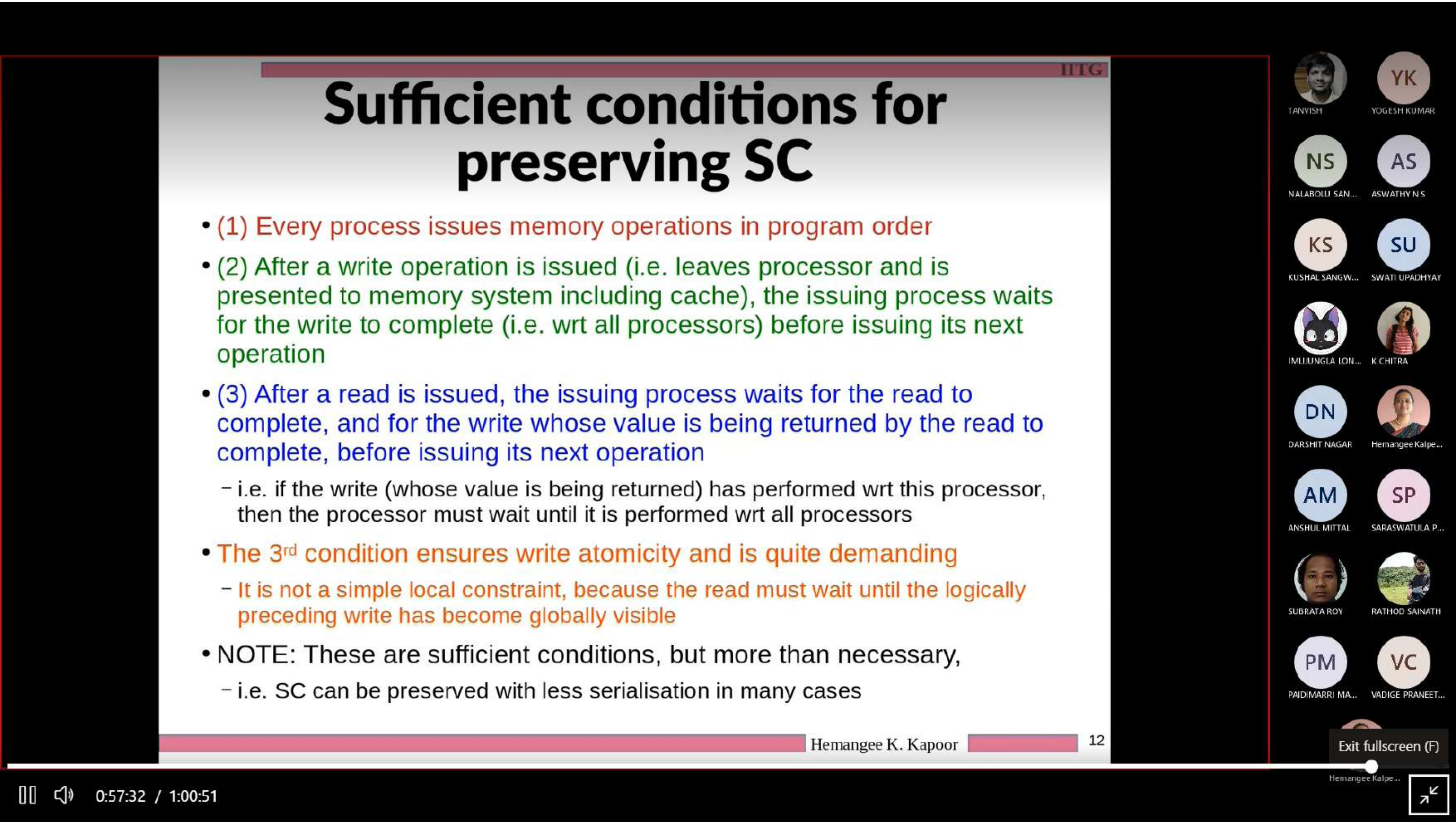
- Write atomicity extends write-serialisation: in that wr-serialisation (coherence) is wrt same location, while wr-atomicity caters to all locations
- All see writes in same order
  - To same location = wr-serialisation
  - To all locations = wr-atomicity

Ex:

```
P1            P2                      P3

A = 1;        while (A == 0);

              B = 1;                  while (B == 0);

                                      print A;
```

- P2 waits for A to become 1 then makes B=1
- P3 waits for B to become 1 then prints A
- Due to transitivity print-A must be 1
- But if P2 is allowed to go on past the read-A and do write-B before P3 has seen the new value of A, then P3 reads new-B and old-A (from its cache) violating sequentially consistent intuition
  - Here P2 should not be allowed to go past Read-A until P3 has also seen the new value of A

Hemangee K. Kapoor 11



IITG

Sufficient conditions for preserving SC

- (1) Every process issues memory operations in program order
- (2) After a write operation is issued (i.e. leaves processor and is presented to memory system including cache), the issuing process waits for the write to complete (i.e. wrt all processors) before issuing its next operation
- (3) After a read is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation
  - i.e. if the write (whose value is being returned) has performed wrt this processor, then the processor must wait until it is performed wrt all processors
- The 3rd condition ensures write atomicity and is quite demanding
  - It is not a simple local constraint, because the read must wait until the logically preceding write has become globally visible
- NOTE: These are sufficient conditions, but more than necessary,
  - i.e. SC can be preserved with less serialisation in many cases