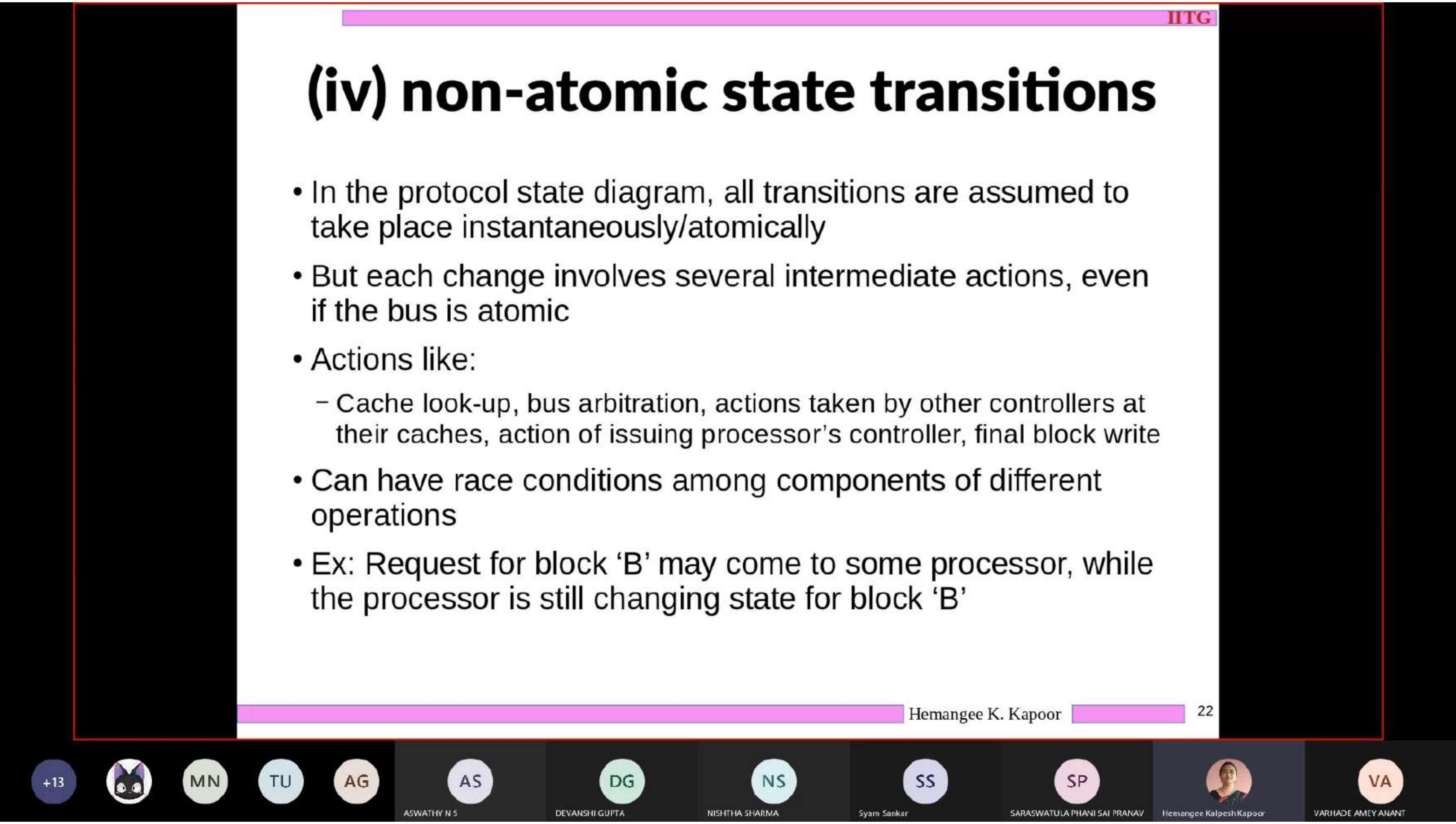
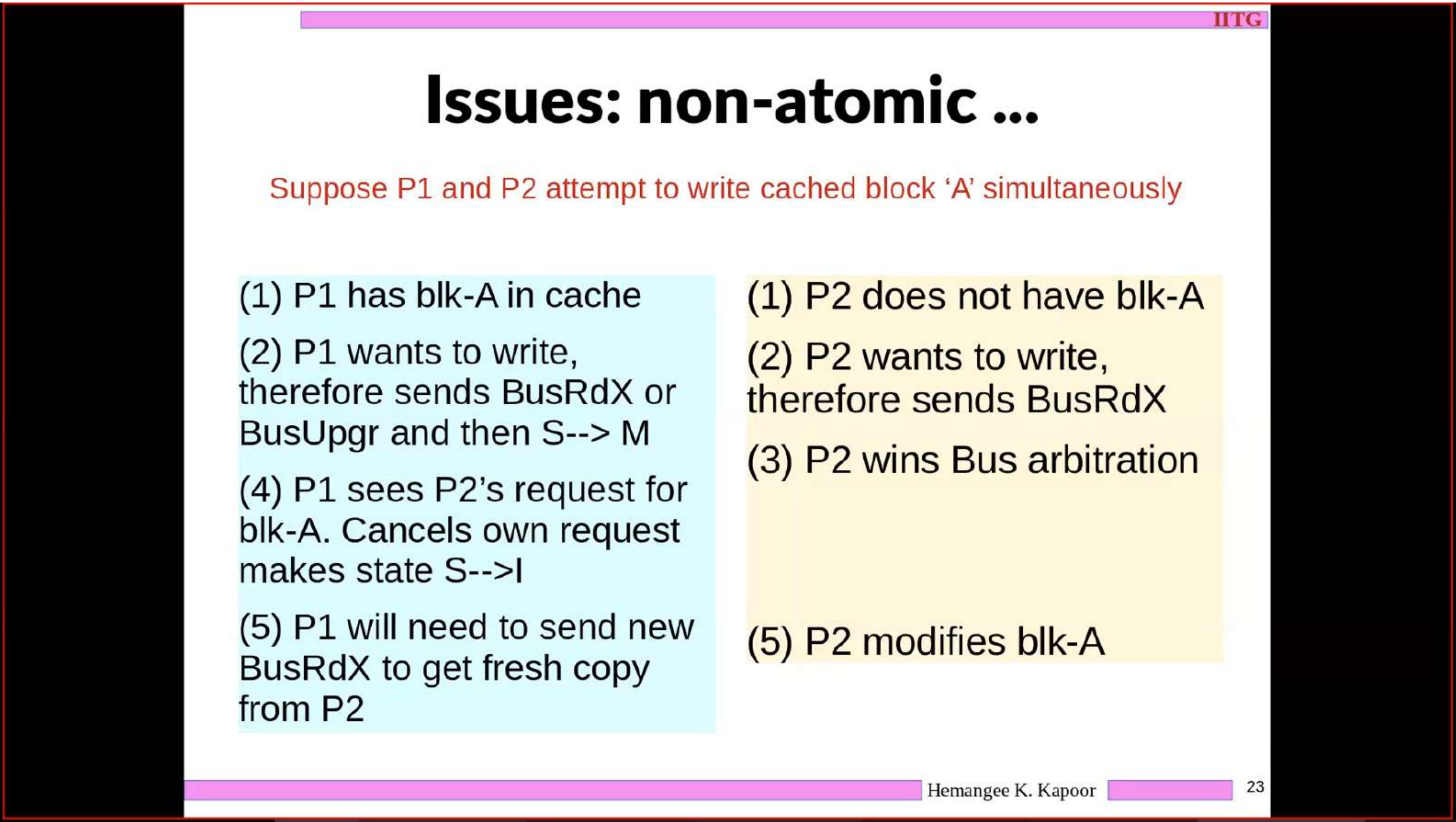


(iv) non-atomic state transitions

- In the protocol state diagram, all transitions are assumed to take place instantaneously/atomically
- But each change involves several intermediate actions, even if the bus is atomic
- Actions like:
  - Cache look-up, bus arbitration, actions taken by other controllers at their caches, action of issuing processor's controller, final block write
- Can have race conditions among components of different operations
- Ex: Request for block 'B' may come to some processor, while the processor is still changing state for block 'B'



Issues: non-atomic …

Suppose P1 and P2 attempt to write cached block ‘A’ simultaneously

(1) P1 has blk-A in cache

(2) P1 wants to write, therefore sends BusRdX or BusUpgr and then S--> M

(4) P1 sees P2’s request for blk-A. Cancels own request makes state S-->I

(5) P1 will need to send new BusRdX to get fresh copy from P2

(1) P2 does not have blk-A

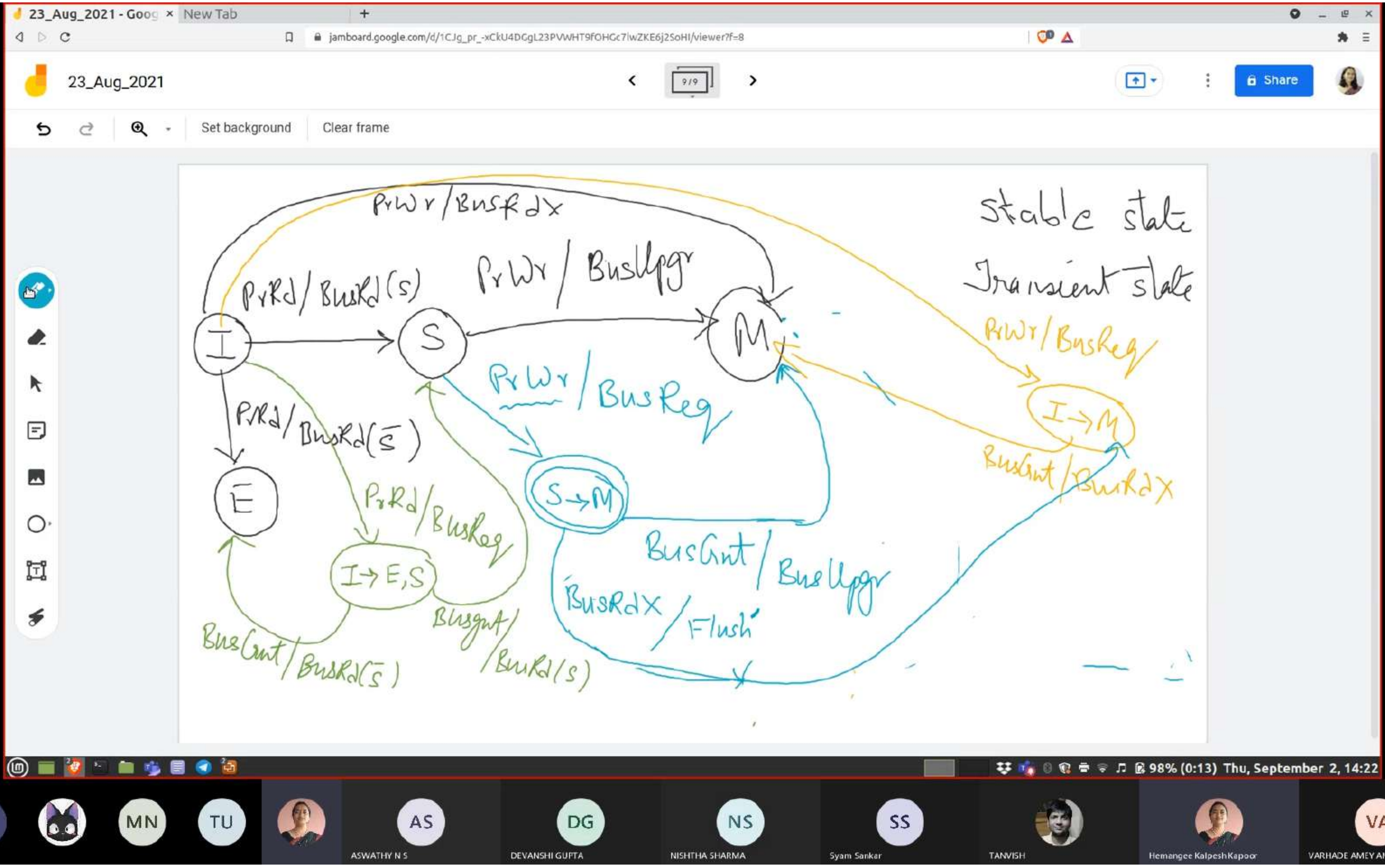
(2) P2 wants to write, therefore sends BusRdX

(3) P2 wins Bus arbitration

(5) P2 modifies blk-A

+16 MN TU AS DG NS SS SP VA

ASWATHY N S DEVANSHI GUPTA NISHTHA SHARMA Syam Sankar SARASWATULA PHANI SAI PRANAV Hemangee Kalpesh Kapoor VARHADE AMEY ANANT


23_Aug_2021
Set background
Clear frame
Share
9/9
Stable state
Transient state
PrWr/BusRdX
PrRd/BusRd(S)
PrWr/BusUpgr
I
S
M
E
PrRd/BusRd(S̄)
PrWr/BusReq
S→M
PrRd/BusReq
I→E,S
BusGnt/BusRd(S̄)
BusGnt/BusRd(S)
BusGnt/BusUpgr
BusRdX/Flush
PrWr/BusReq
I→M
BusGnt/BusRdX
ASWATHY N S
DEVANSHI GUPTA
NISHTHA SHARMA
Syam Sankar
TANVISH
Hemangee Kalpesh Kapoor
VARHADE AMEY ANANT
98% (0:13) Thu, September 2, 14:22

Transient states

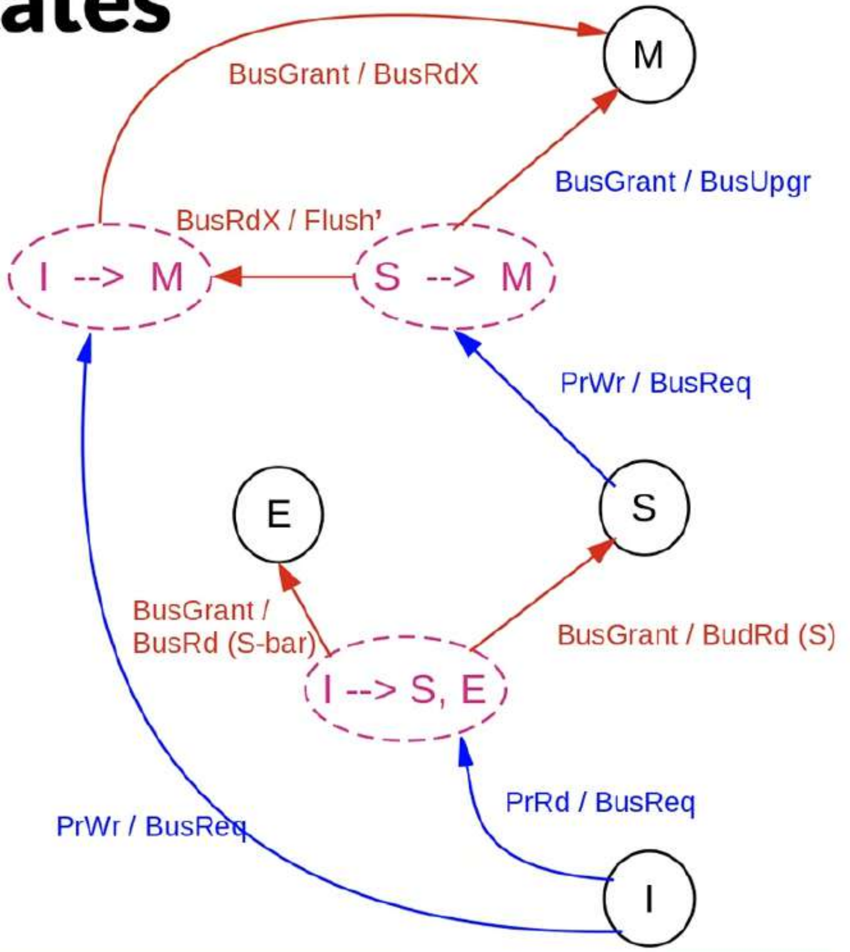
- Changes in MESI:

  I --> S,E

  S --> M

  I --> M

- Shown dotted in the FSM

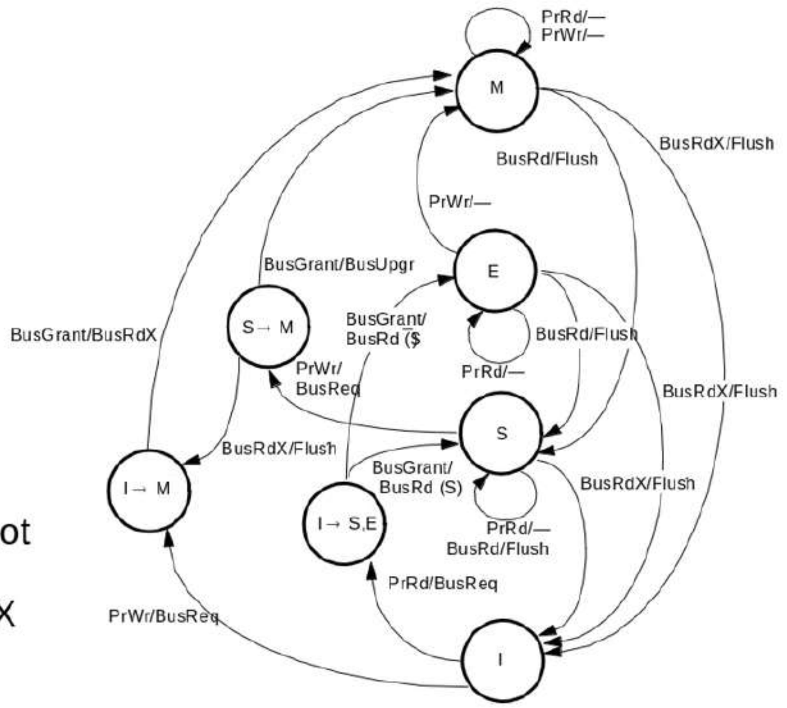


+23 MN TU AS YK DG NS SS VA

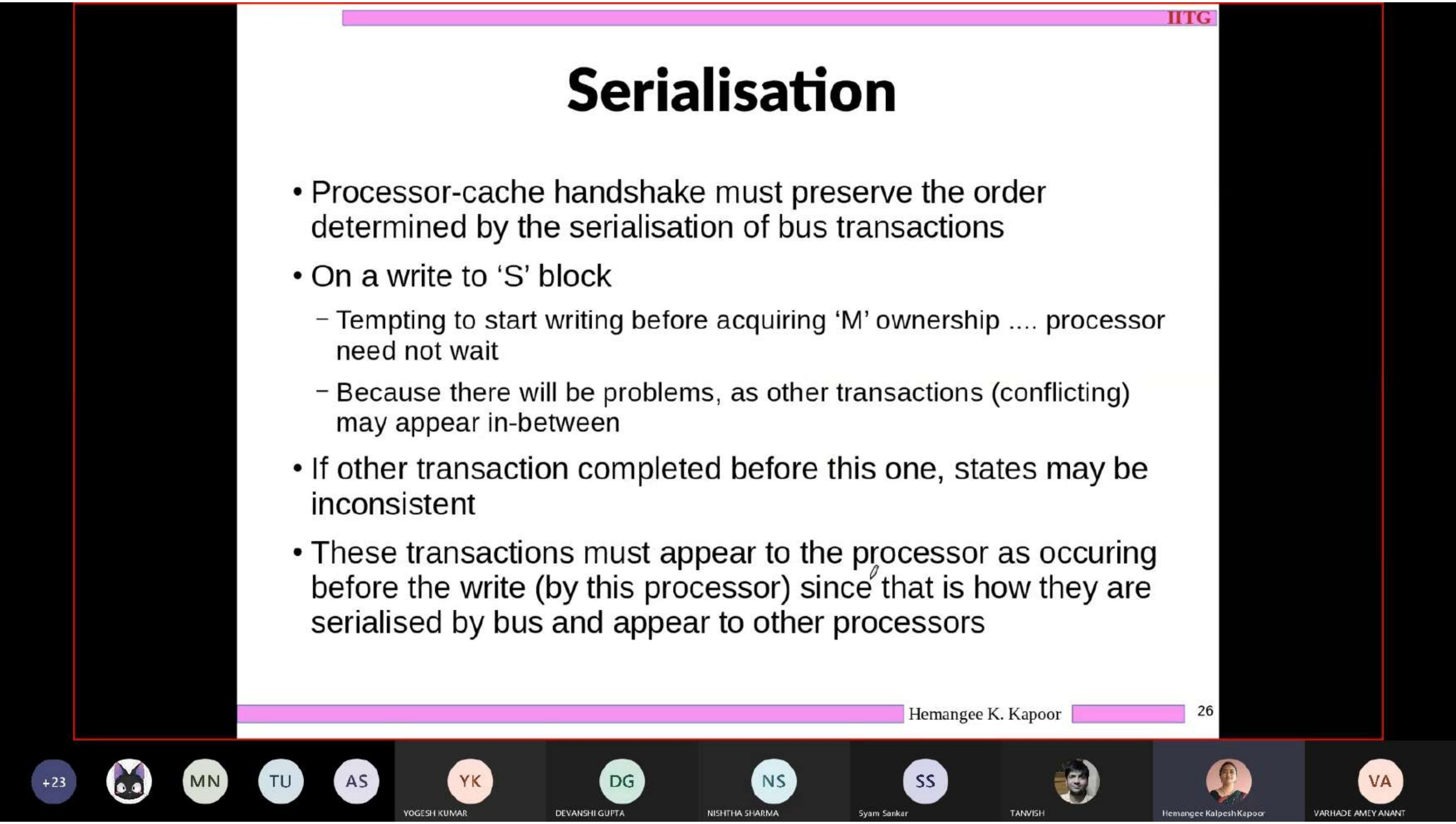
YOGESH KUMAR DEVANSHI GUPTA NISHTHA SHARMA Syam Sankar TANVISH Hemangee Kalpesh Kapoor VARHADE AMEY ANANT

Handling Non-atomicity: Transient States

- 2 types of states
  - Stable (e.g. original MESI)
  - Transient or Intermediate
- Increases complexity
  - e.g. don't send BusUpgr, rather use other mechanisms to avoid data transfer
- If we always use BusRdX then we do not need the intermediate state (S->M), as the output action will always be BusRdX
  - In case of BusUpgr we have to modify the output action to BusRdX and hence one transient state is required.

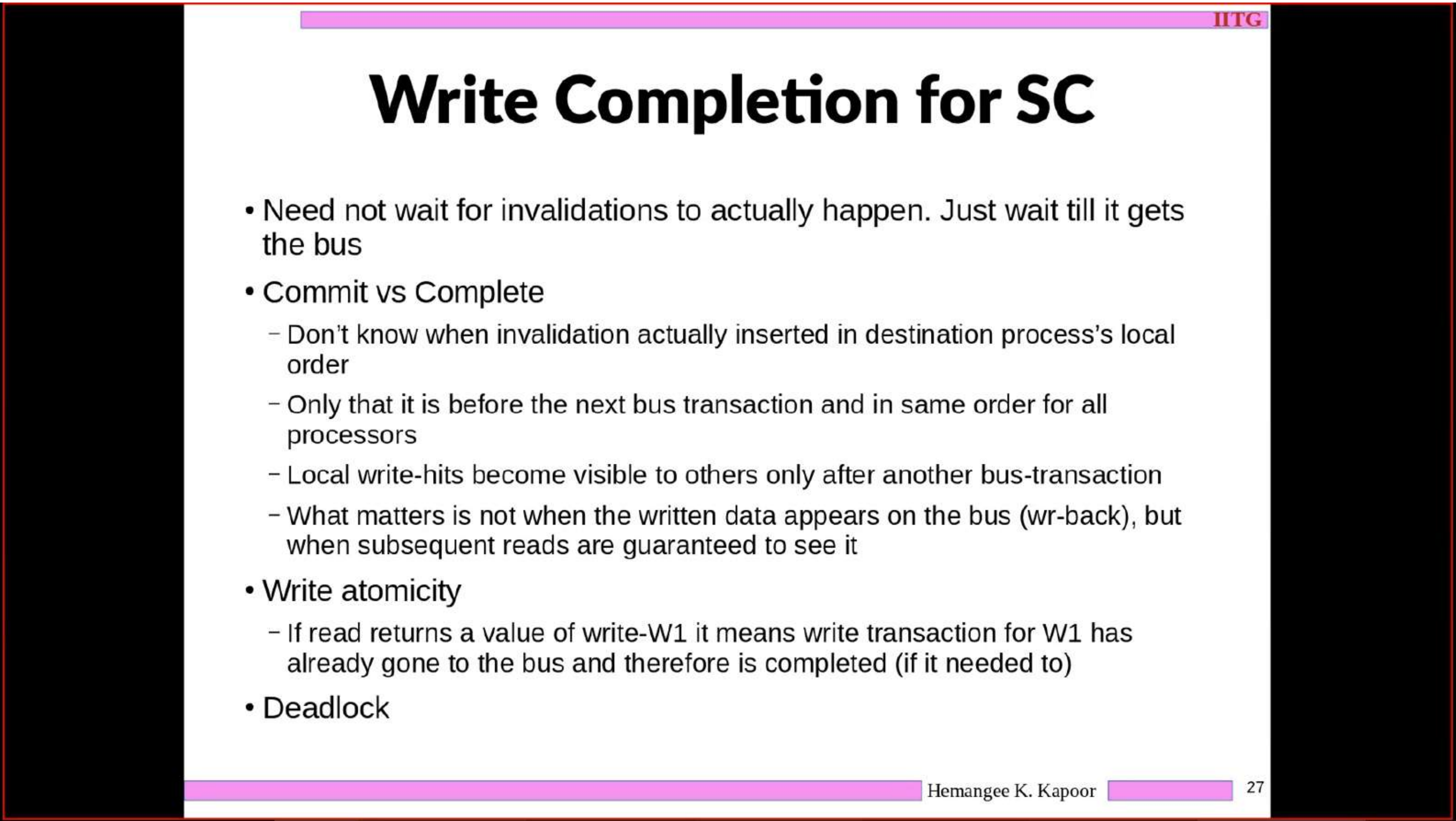


+23 MN TU AS YK DG NS SS VA
YOGESH KUMAR DEVANSHI GUPTA NISHTHA SHARMA Syam Sankar TANVISH Hemangee Kalpesh Kapoor VARHADE AMEY ANANT



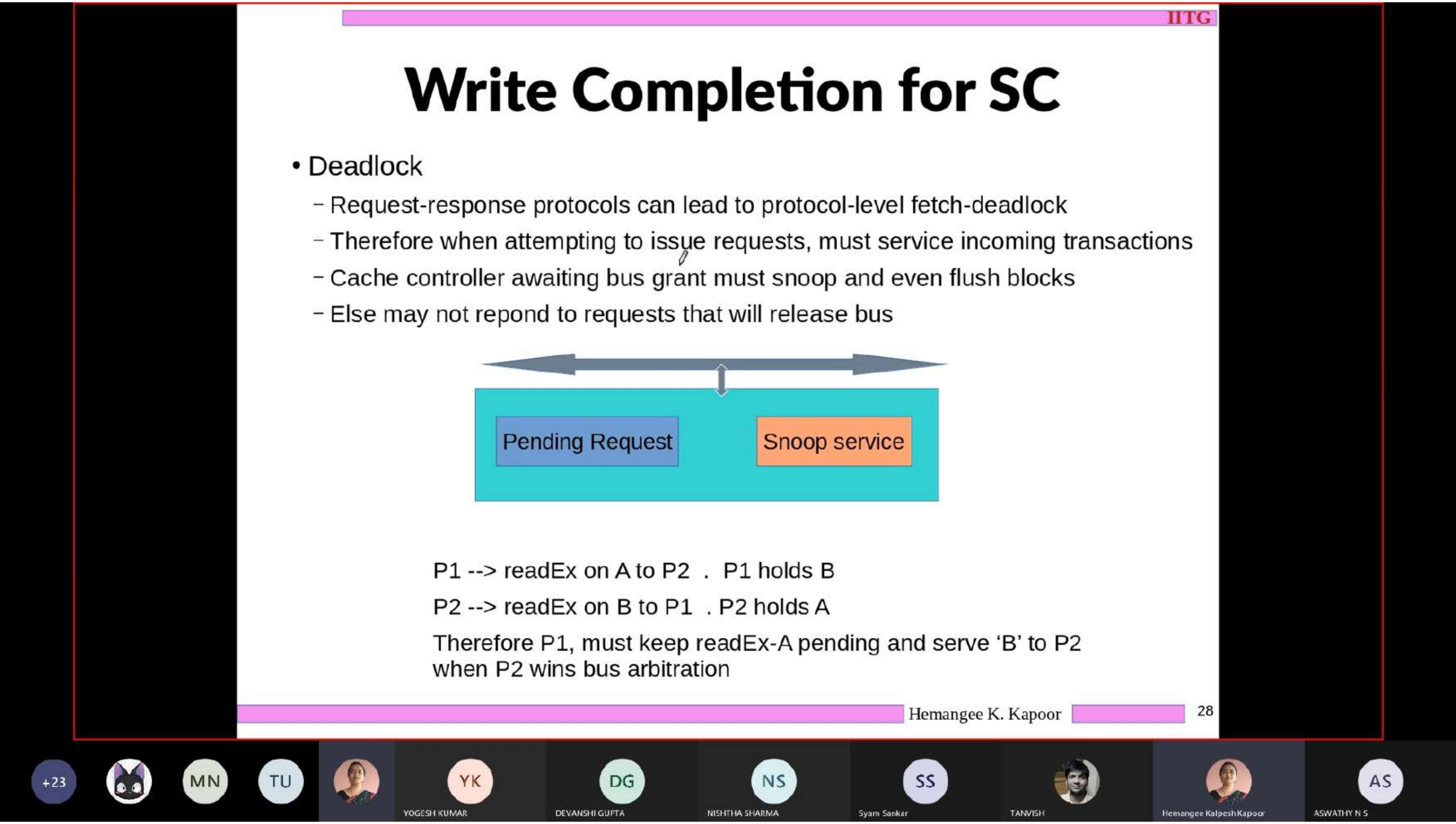
Serialisation

- Processor-cache handshake must preserve the order determined by the serialisation of bus transactions
- On a write to 'S' block
  - Tempting to start writing before acquiring 'M' ownership .... processor need not wait
  - Because there will be problems, as other transactions (conflicting) may appear in-between
- If other transaction completed before this one, states may be inconsistent
- These transactions must appear to the processor as occuring before the write (by this processor) since that is how they are serialised by bus and appear to other processors



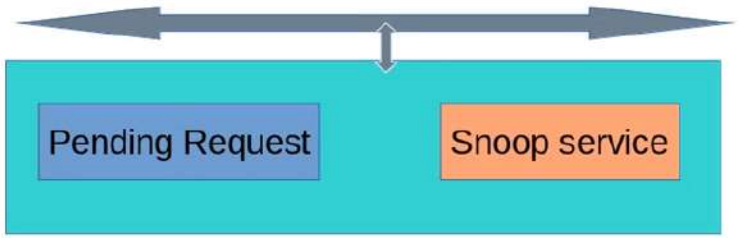
Write Completion for SC

- Need not wait for invalidations to actually happen. Just wait till it gets the bus
- Commit vs Complete
  - Don't know when invalidation actually inserted in destination process's local order
  - Only that it is before the next bus transaction and in same order for all processors
  - Local write-hits become visible to others only after another bus-transaction
  - What matters is not when the written data appears on the bus (wr-back), but when subsequent reads are guaranteed to see it
- Write atomicity
  - If read returns a value of write-W1 it means write transaction for W1 has already gone to the bus and therefore is completed (if it needed to)
- Deadlock



Write Completion for SC

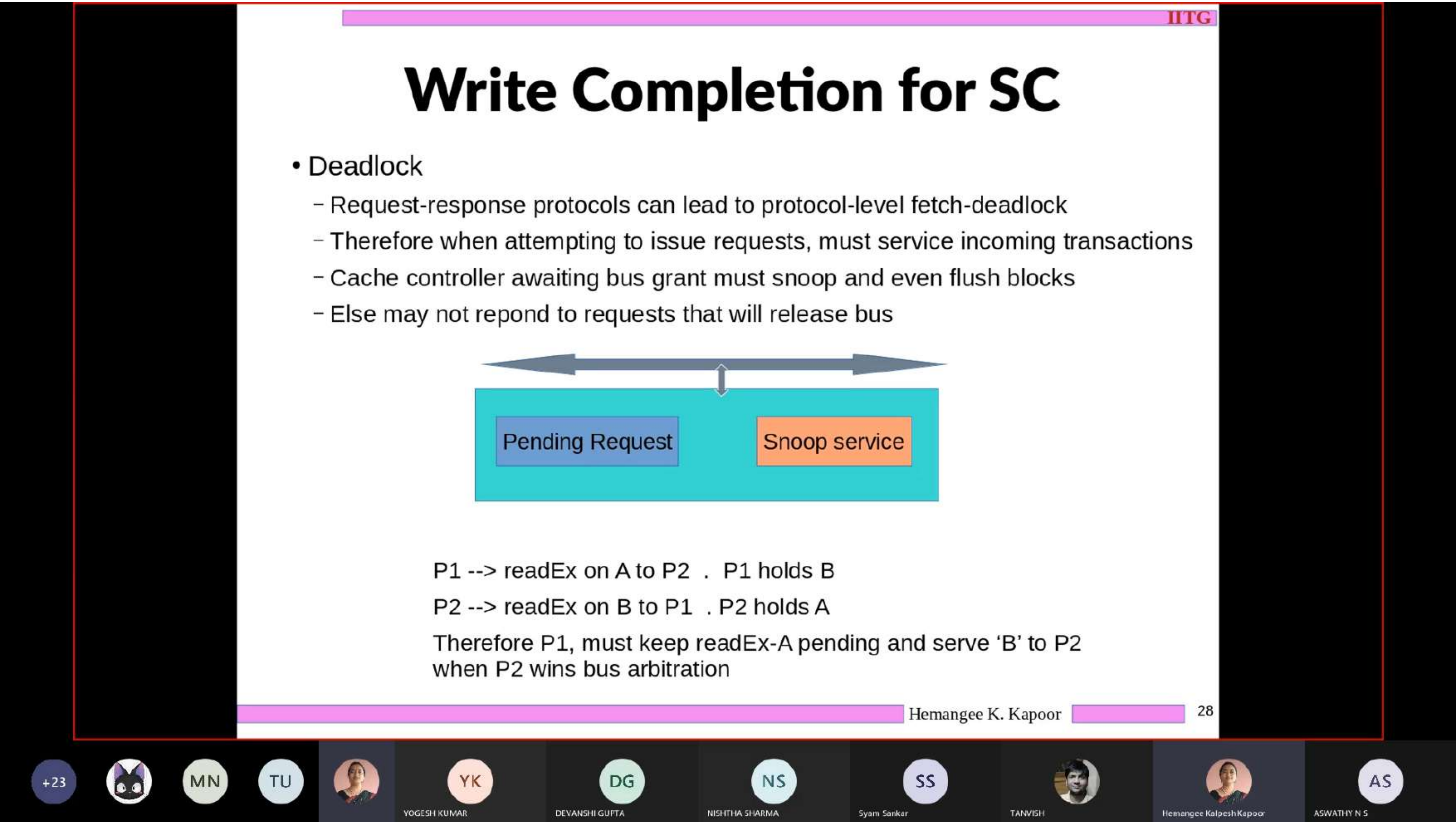
- Deadlock
  - Request-response protocols can lead to protocol-level fetch-deadlock
  - Therefore when attempting to issue requests, must service incoming transactions
  - Cache controller awaiting bus grant must snoop and even flush blocks
  - Else may not repond to requests that will release bus



P1 --> readEx on A to P2 . P1 holds B

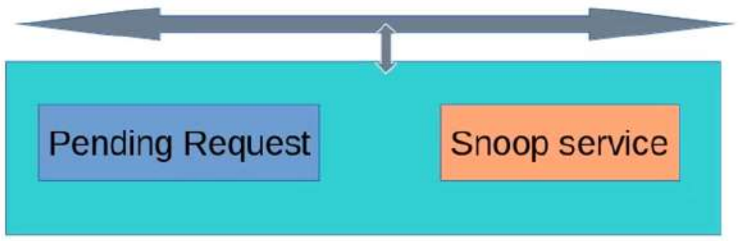
P2 --> readEx on B to P1 . P2 holds A

Therefore P1, must keep readEx-A pending and serve 'B' to P2 when P2 wins bus arbitration



Write Completion for SC

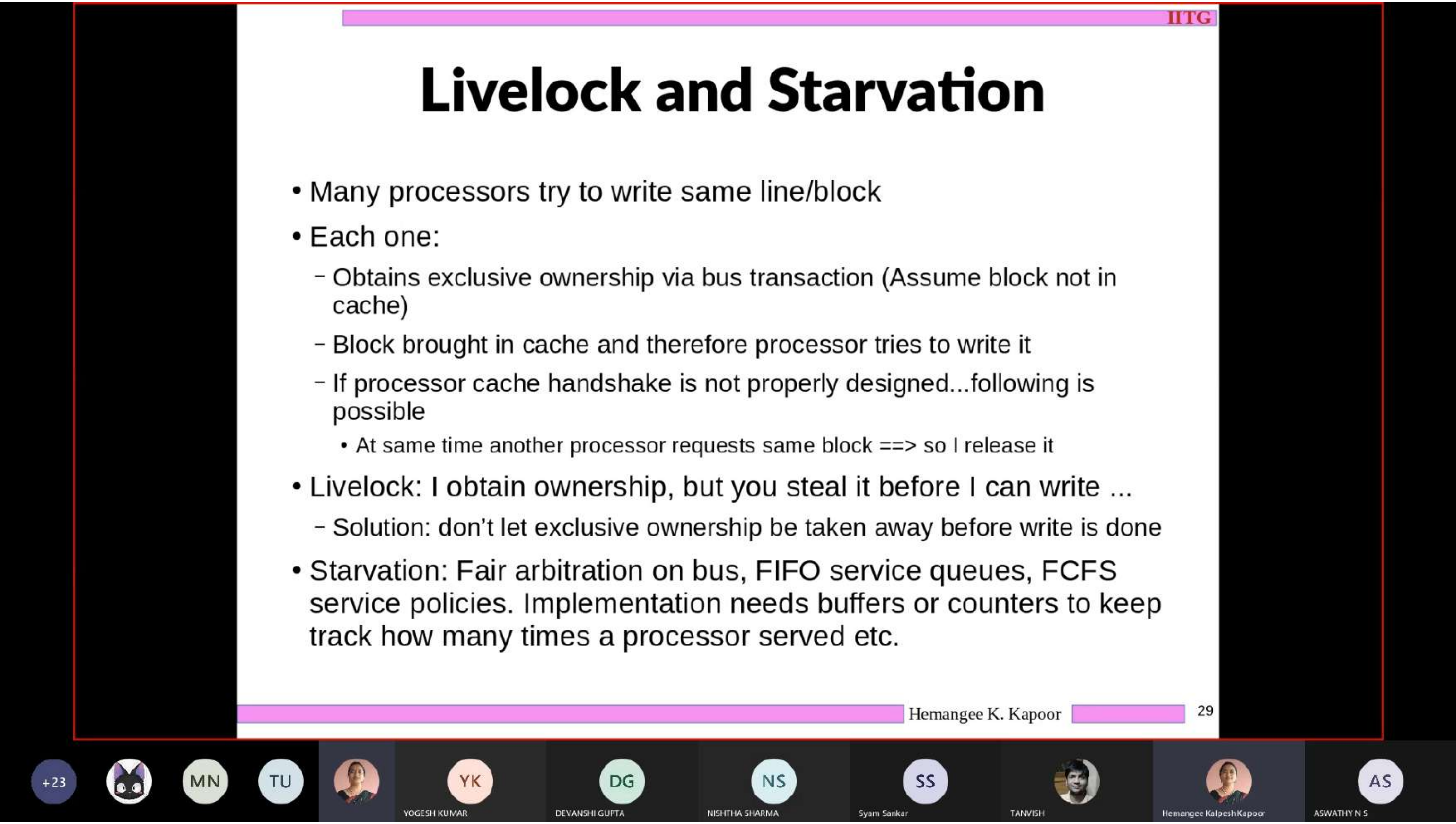
- Deadlock
  - Request-response protocols can lead to protocol-level fetch-deadlock
  - Therefore when attempting to issue requests, must service incoming transactions
  - Cache controller awaiting bus grant must snoop and even flush blocks
  - Else may not repond to requests that will release bus



P1 --> readEx on A to P2 . P1 holds B

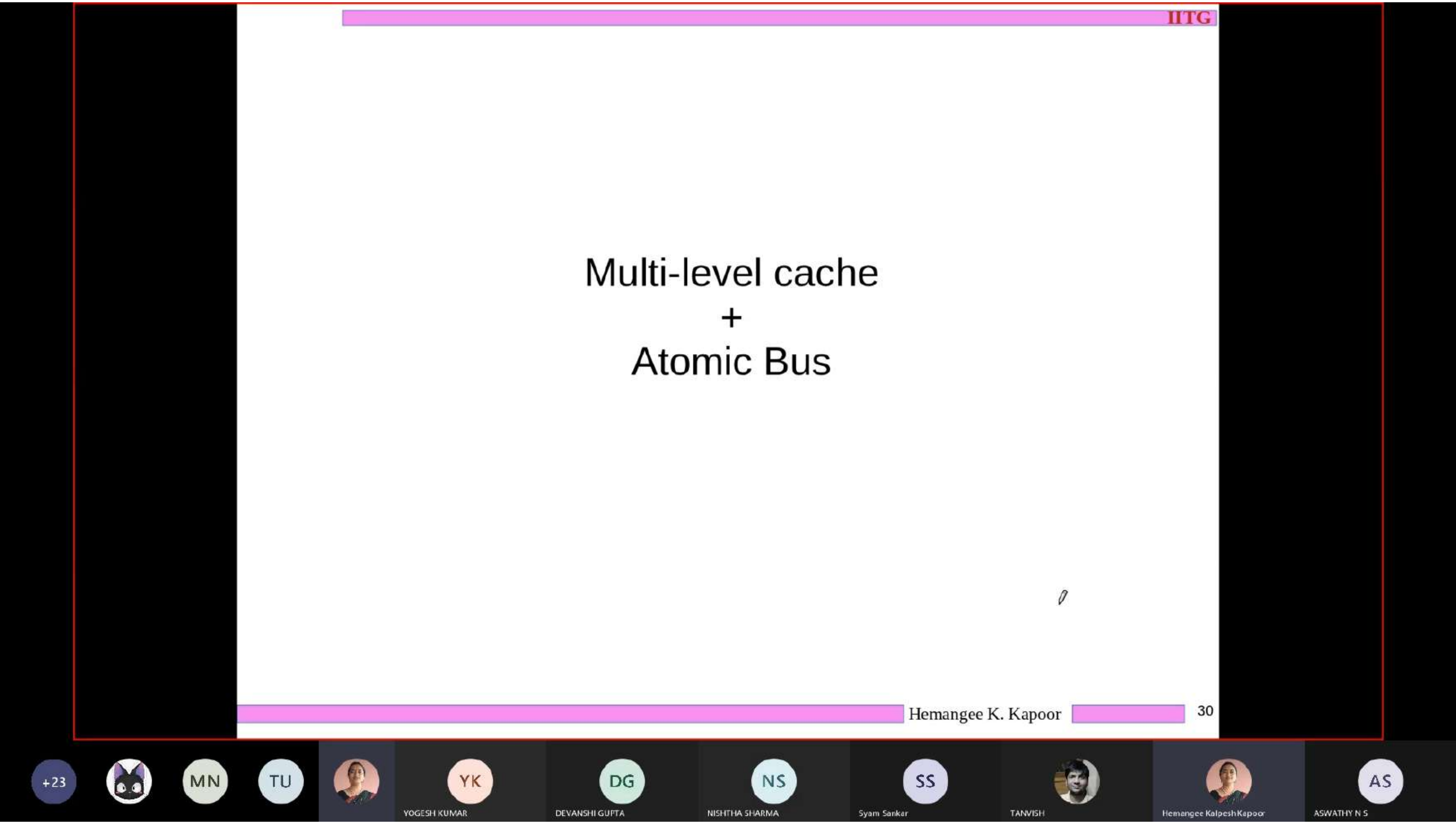
P2 --> readEx on B to P1 . P2 holds A

Therefore P1, must keep readEx-A pending and serve 'B' to P2 when P2 wins bus arbitration

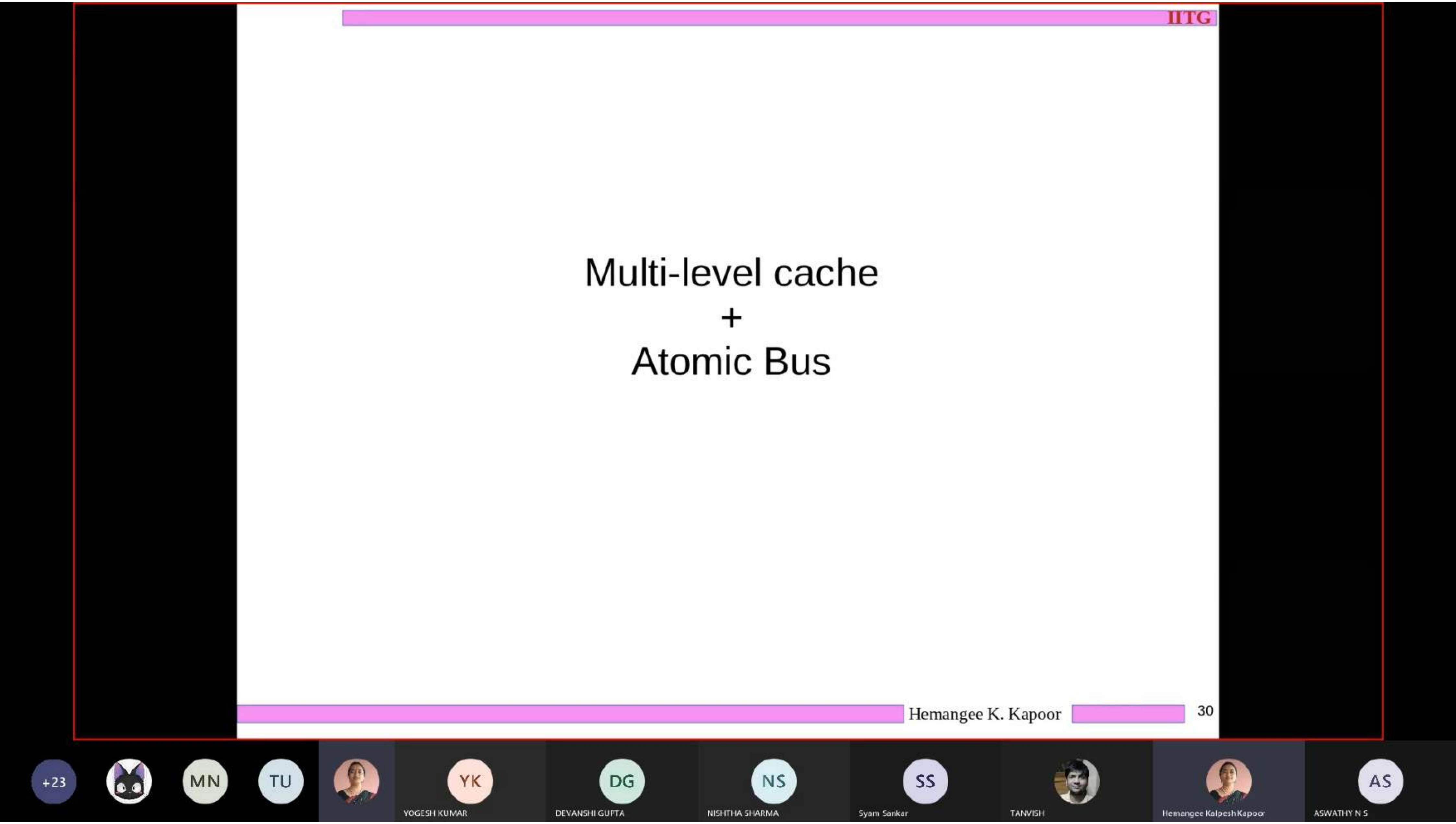


Livelock and Starvation

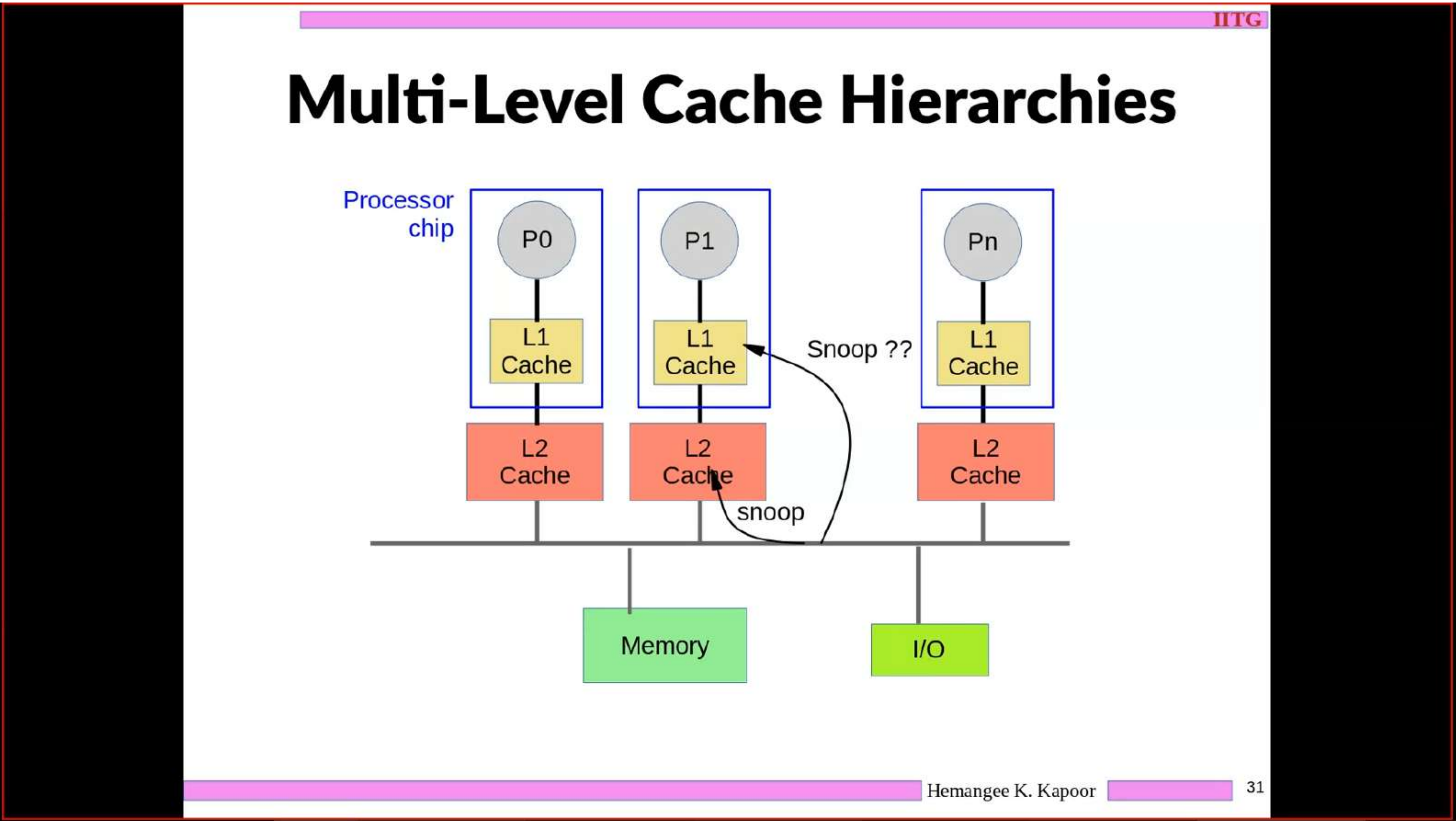
- Many processors try to write same line/block
- Each one:
  - Obtains exclusive ownership via bus transaction (Assume block not in cache)
  - Block brought in cache and therefore processor tries to write it
  - If processor cache handshake is not properly designed...following is possible
    - At same time another processor requests same block ==> so I release it
- Livelock: I obtain ownership, but you steal it before I can write ...
  - Solution: don't let exclusive ownership be taken away before write is done
- Starvation: Fair arbitration on bus, FIFO service queues, FCFS service policies. Implementation needs buffers or counters to keep track how many times a processor served etc.



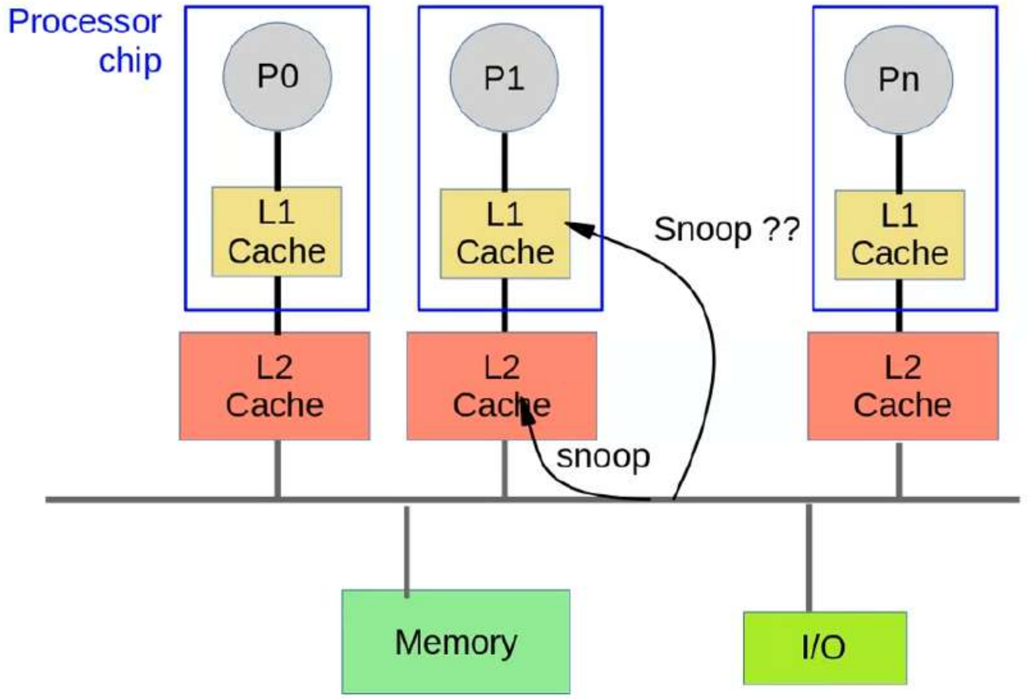
Multi-level cache
+
Atomic Bus

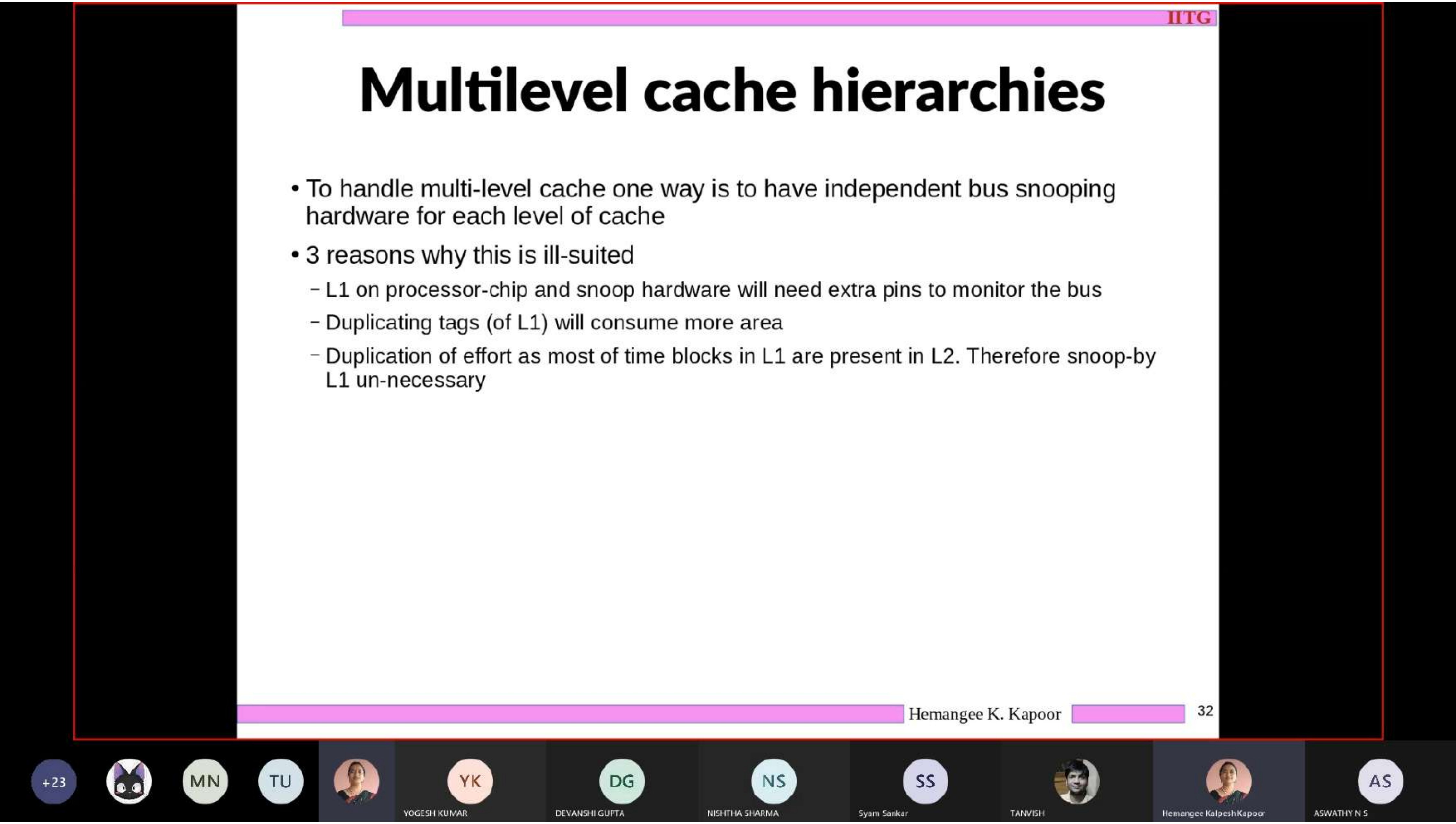


Multi-level cache
+
Atomic Bus



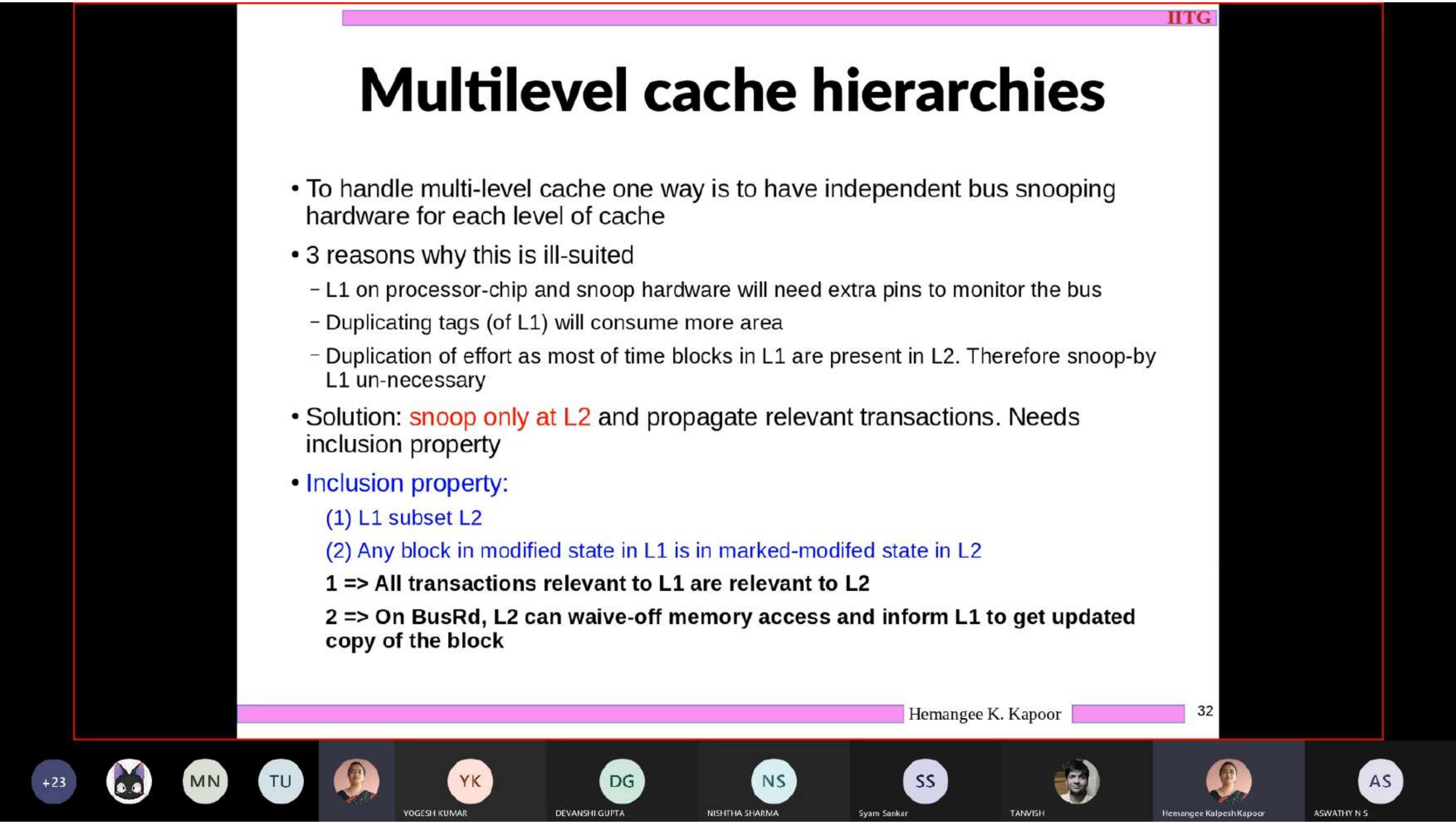
Multi-Level Cache Hierarchies





Multilevel cache hierarchies

- To handle multi-level cache one way is to have independent bus snooping hardware for each level of cache
- 3 reasons why this is ill-suited
  - L1 on processor-chip and snoop hardware will need extra pins to monitor the bus
  - Duplicating tags (of L1) will consume more area
  - Duplication of effort as most of time blocks in L1 are present in L2. Therefore snoop-by L1 un-necessary



Multilevel cache hierarchies

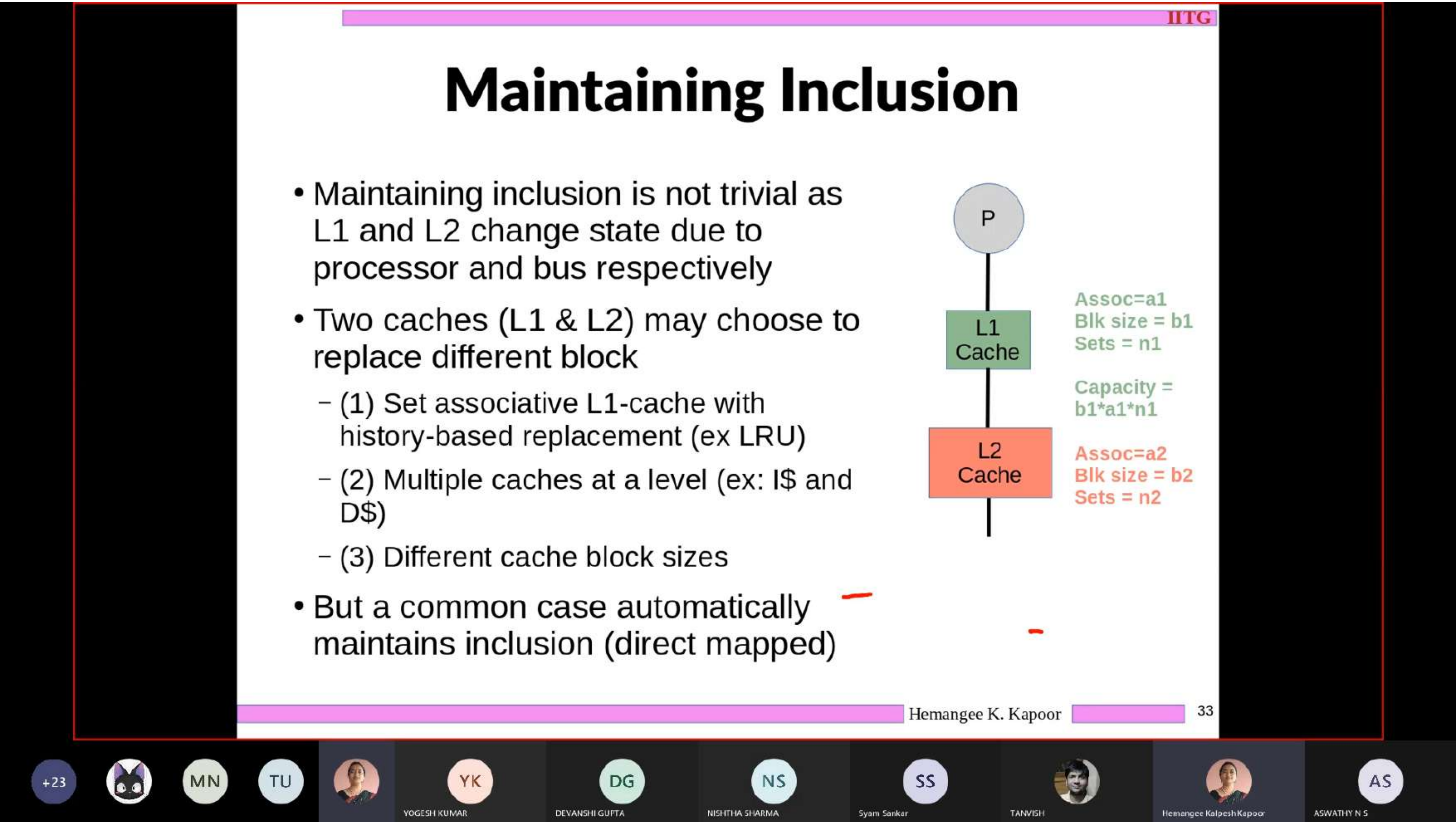
- To handle multi-level cache one way is to have independent bus snooping hardware for each level of cache
- 3 reasons why this is ill-suited
  - L1 on processor-chip and snoop hardware will need extra pins to monitor the bus
  - Duplicating tags (of L1) will consume more area
  - Duplication of effort as most of time blocks in L1 are present in L2. Therefore snoop-by L1 un-necessary
- Solution: snoop only at L2 and propagate relevant transactions. Needs inclusion property
- Inclusion property:

  (1) L1 subset L2

  (2) Any block in modified state in L1 is in marked-modifed state in L2

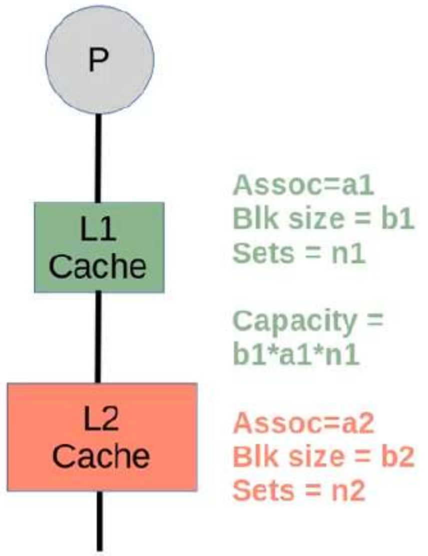
  1 => All transactions relevant to L1 are relevant to L2

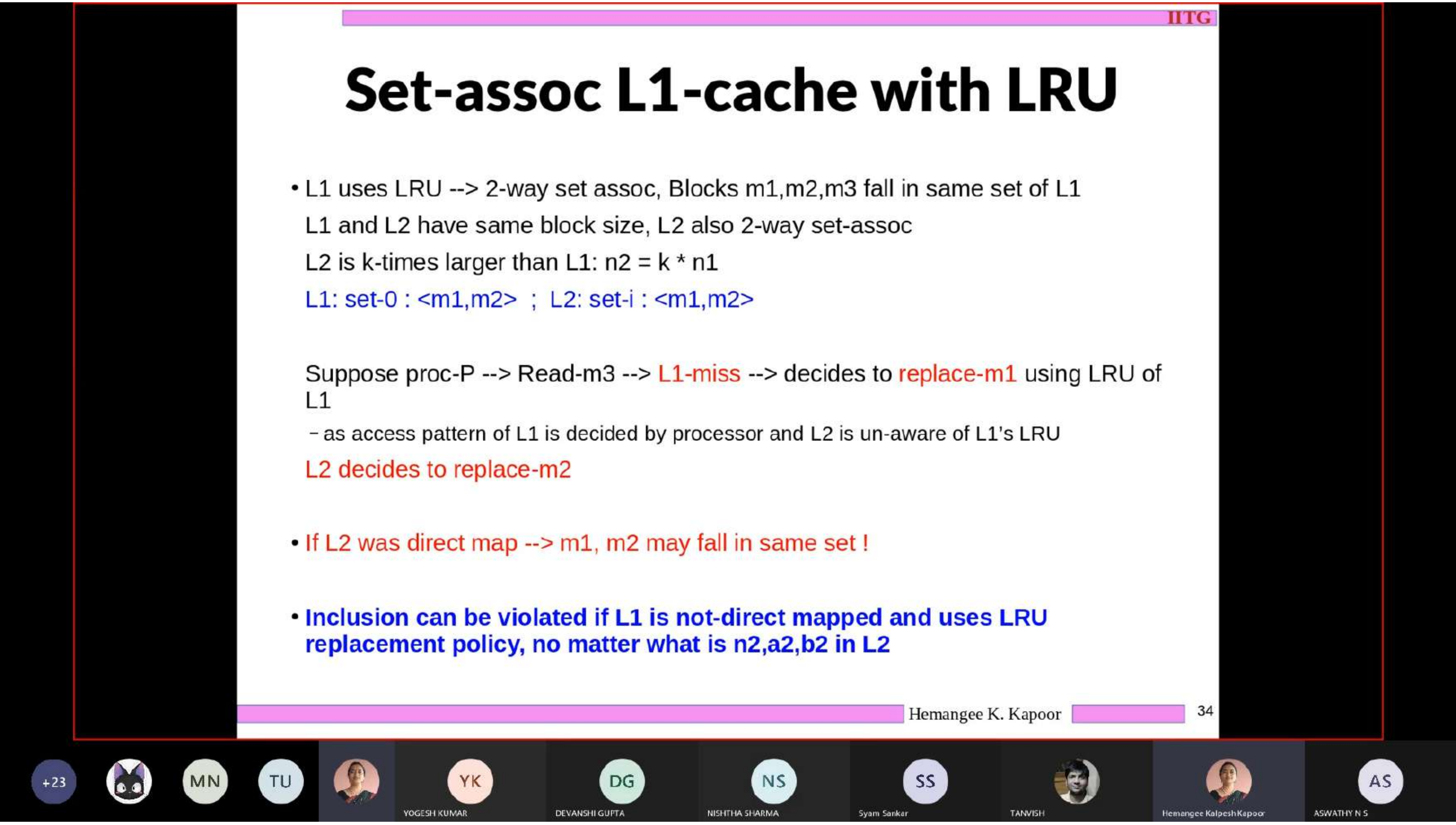
  2 => On BusRd, L2 can waive-off memory access and inform L1 to get updated copy of the block



Maintaining Inclusion

- Maintaining inclusion is not trivial as L1 and L2 change state due to processor and bus respectively
- Two caches (L1 & L2) may choose to replace different block
  - (1) Set associative L1-cache with history-based replacement (ex LRU)
  - (2) Multiple caches at a level (ex: I$ and D$)
  - (3) Different cache block sizes
- But a common case automatically maintains inclusion (direct mapped)





Set-assoc L1-cache with LRU

- L1 uses LRU --> 2-way set assoc, Blocks m1,m2,m3 fall in same set of L1

  L1 and L2 have same block size, L2 also 2-way set-assoc

  L2 is k-times larger than L1: n2 = k * n1

  L1: set-0 : <m1,m2> ; L2: set-i : <m1,m2>

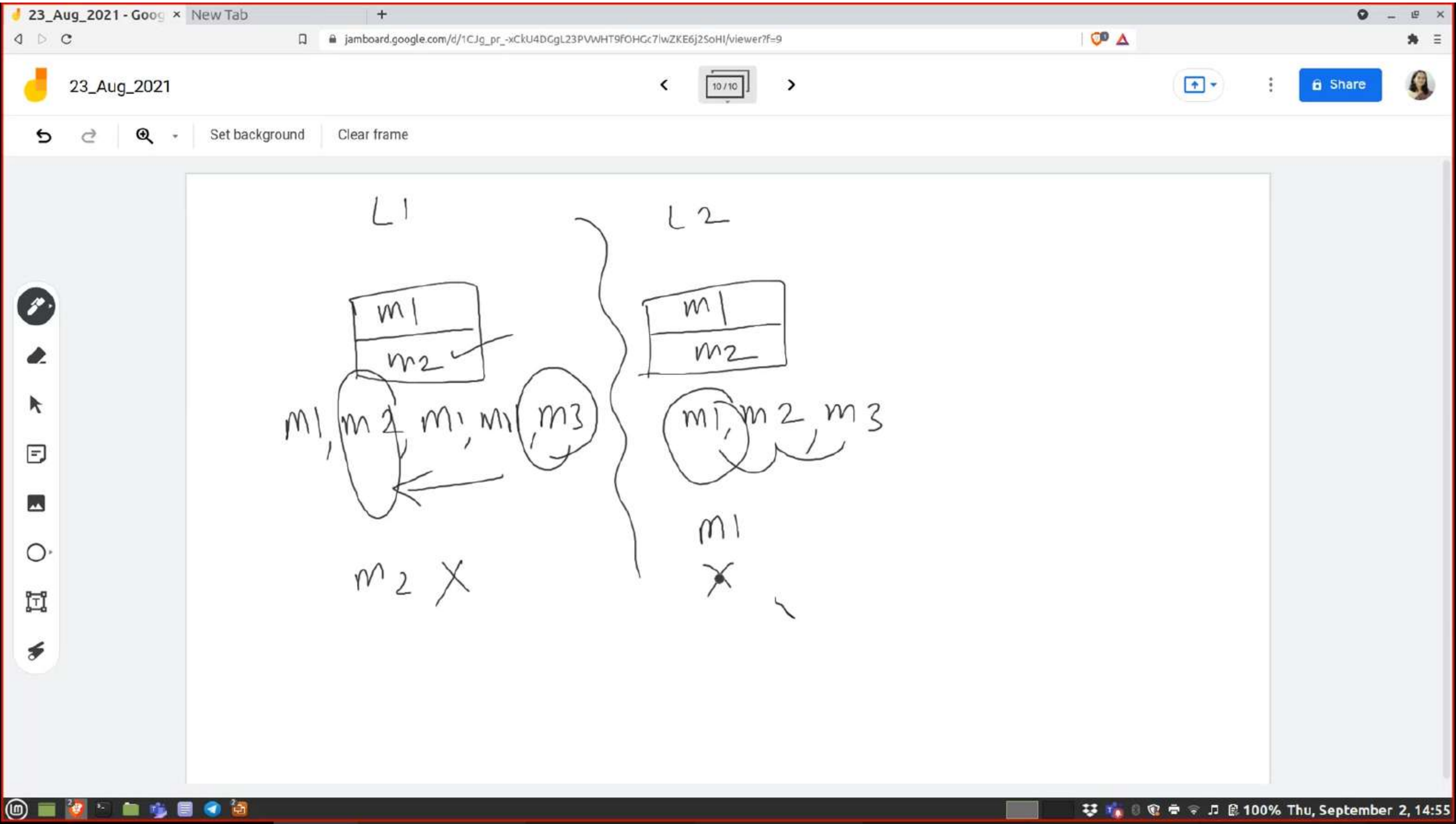
  Suppose proc-P --> Read-m3 --> L1-miss --> decides to replace-m1 using LRU of L1

  - as access pattern of L1 is decided by processor and L2 is un-aware of L1's LRU

  L2 decides to replace-m2

- If L2 was direct map --> m1, m2 may fall in same set !

- **Inclusion can be violated if L1 is not-direct mapped and uses LRU replacement policy, no matter what is n2,a2,b2 in L2**


23_Aug_2021 - Goog × New Tab +

jamboard.google.com/d/1CJg_pr_-xCkU4DGgL23PVWHT9fOHGc7lwZKE6j2SoHI/viewer?f=9

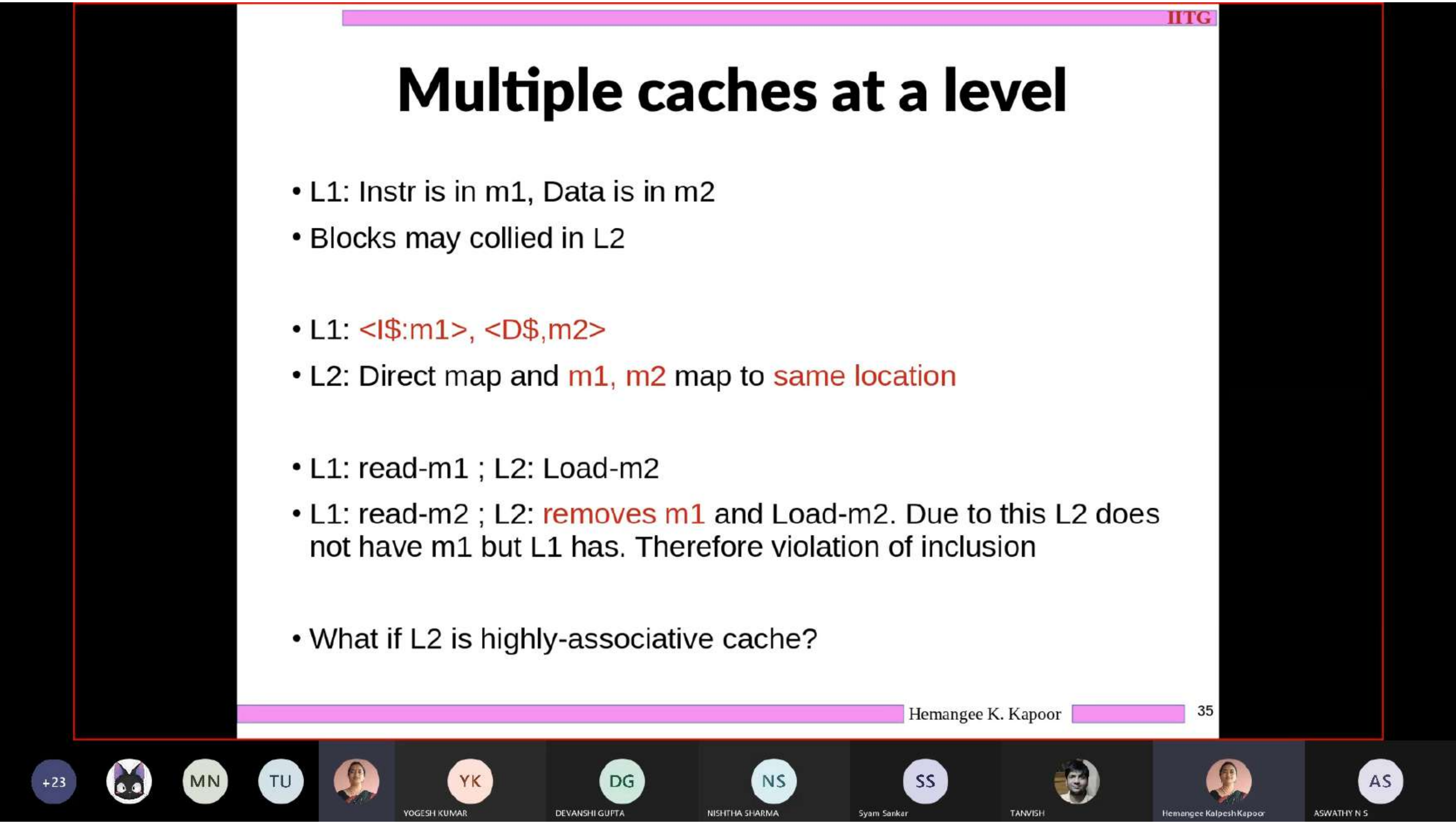
23_Aug_2021

10 / 10

Share

Set background Clear frame

100% Thu, September 2, 14:55



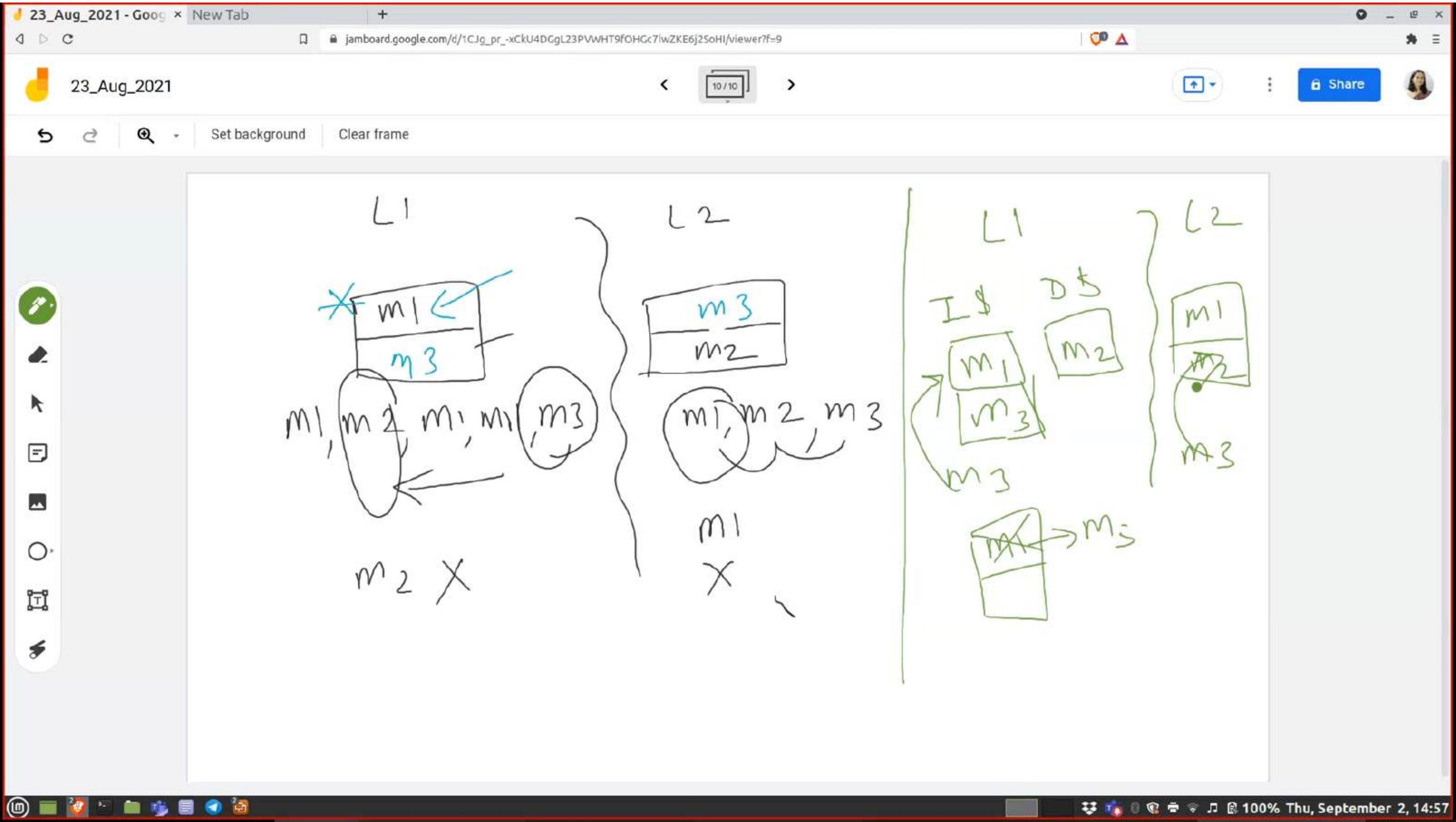
Multiple caches at a level

- L1: Instr is in m1, Data is in m2
- Blocks may collied in L2

- L1: <I$:m1>, <D$,m2>
- L2: Direct map and m1, m2 map to same location

- L1: read-m1 ; L2: Load-m2
- L1: read-m2 ; L2: removes m1 and Load-m2. Due to this L2 does not have m1 but L1 has. Therefore violation of inclusion

- What if L2 is highly-associative cache?


23_Aug_2021 - Goog × New Tab +

jamboard.google.com/d/1CJg_pr_-xCkU4DGgL23PVWHT9fOHGc7lwZKE6j2SoHI/viewer?f=9

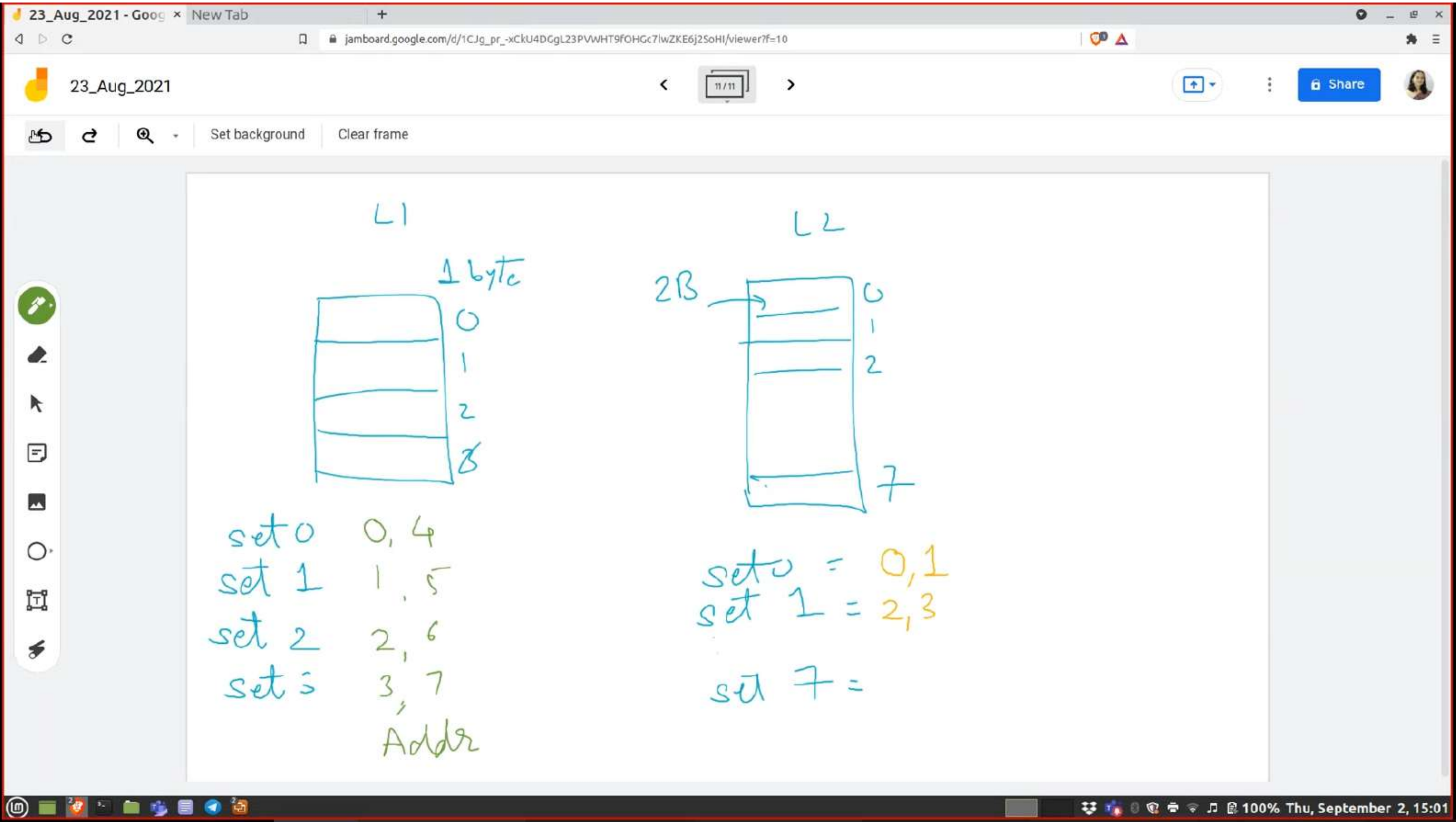
23_Aug_2021

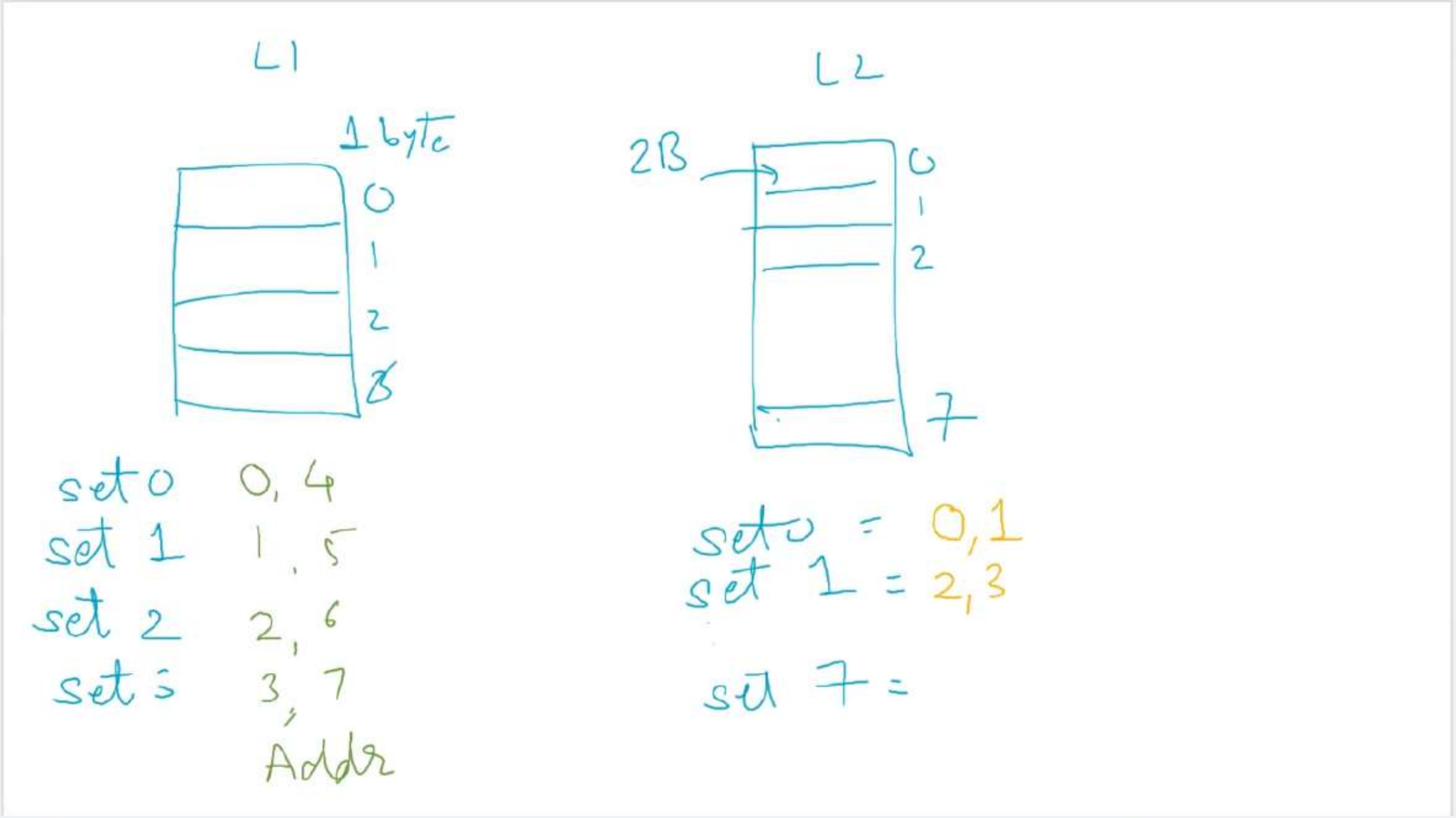
10 / 10

Share

Set background Clear frame

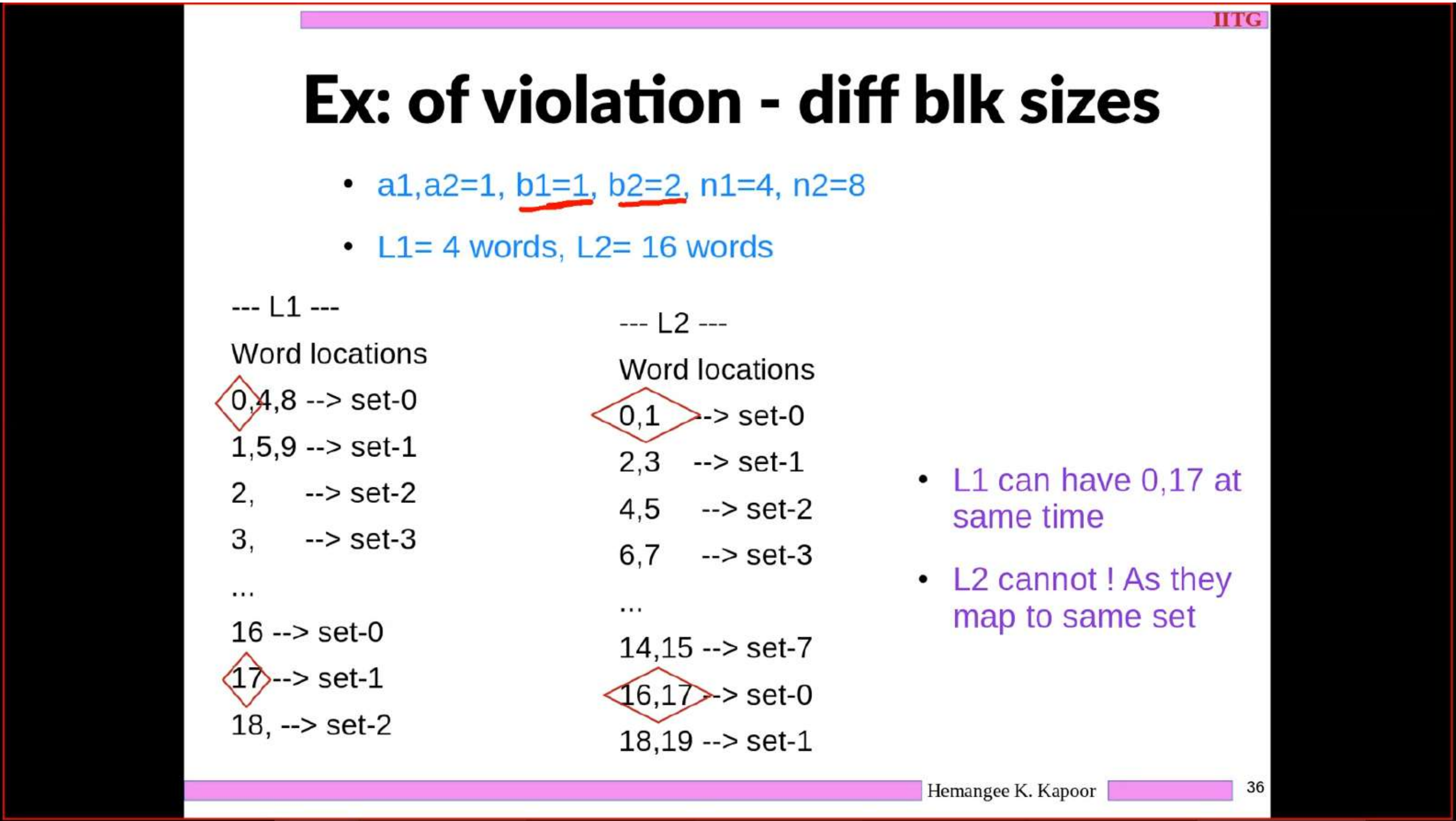
100% Thu, September 2, 14:57


23_Aug_2021 - Goog
New Tab
jamboard.google.com/d/1CJg_pr_-xCkU4DGgL23PVWHT9fOHGc7lwZKE6j2SoHI/viewer?f=10
23_Aug_2021
11 / 11
Share
Set background
Clear frame
100% Thu, September 2, 15:01



set 0 0, 4
set 1 1, 5
set 2 2, 6
set 3 3, 7
Addr

set 0 = 0, 1
set 1 = 2, 3
set 7 =



Ex: of violation - diff blk sizes

- a1,a2=1, b1=1, b2=2, n1=4, n2=8
- L1= 4 words, L2= 16 words

--- L1 ---

Word locations

0,4,8 --> set-0

1,5,9 --> set-1

2, --> set-2

3, --> set-3

...

16 --> set-0

17 --> set-1

18, --> set-2

--- L2 ---

Word locations

0,1 --> set-0

2,3 --> set-1

4,5 --> set-2

6,7 --> set-3

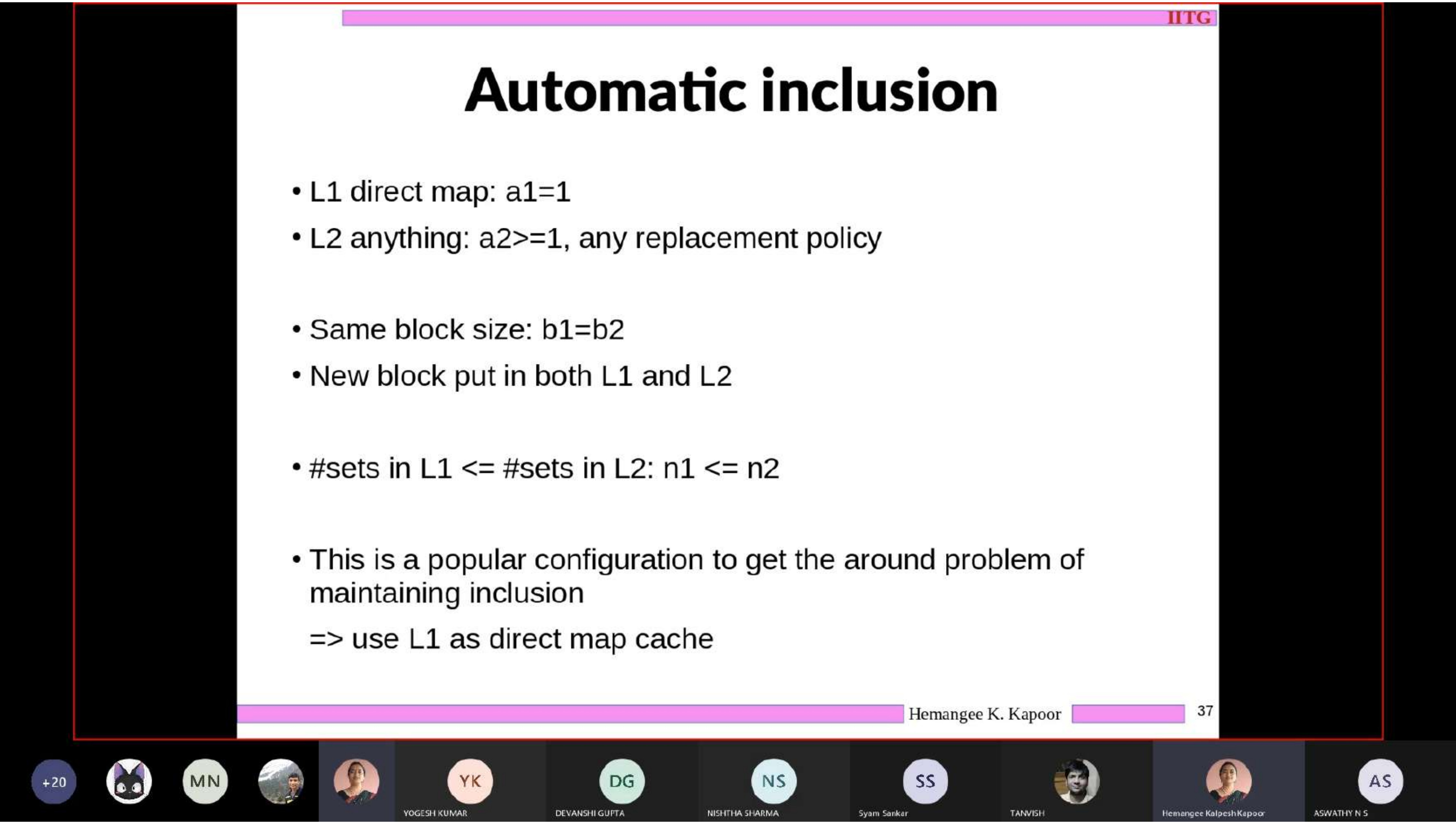
...

14,15 --> set-7

16,17 --> set-0

18,19 --> set-1

- L1 can have 0,17 at same time
- L2 cannot ! As they map to same set



Automatic inclusion

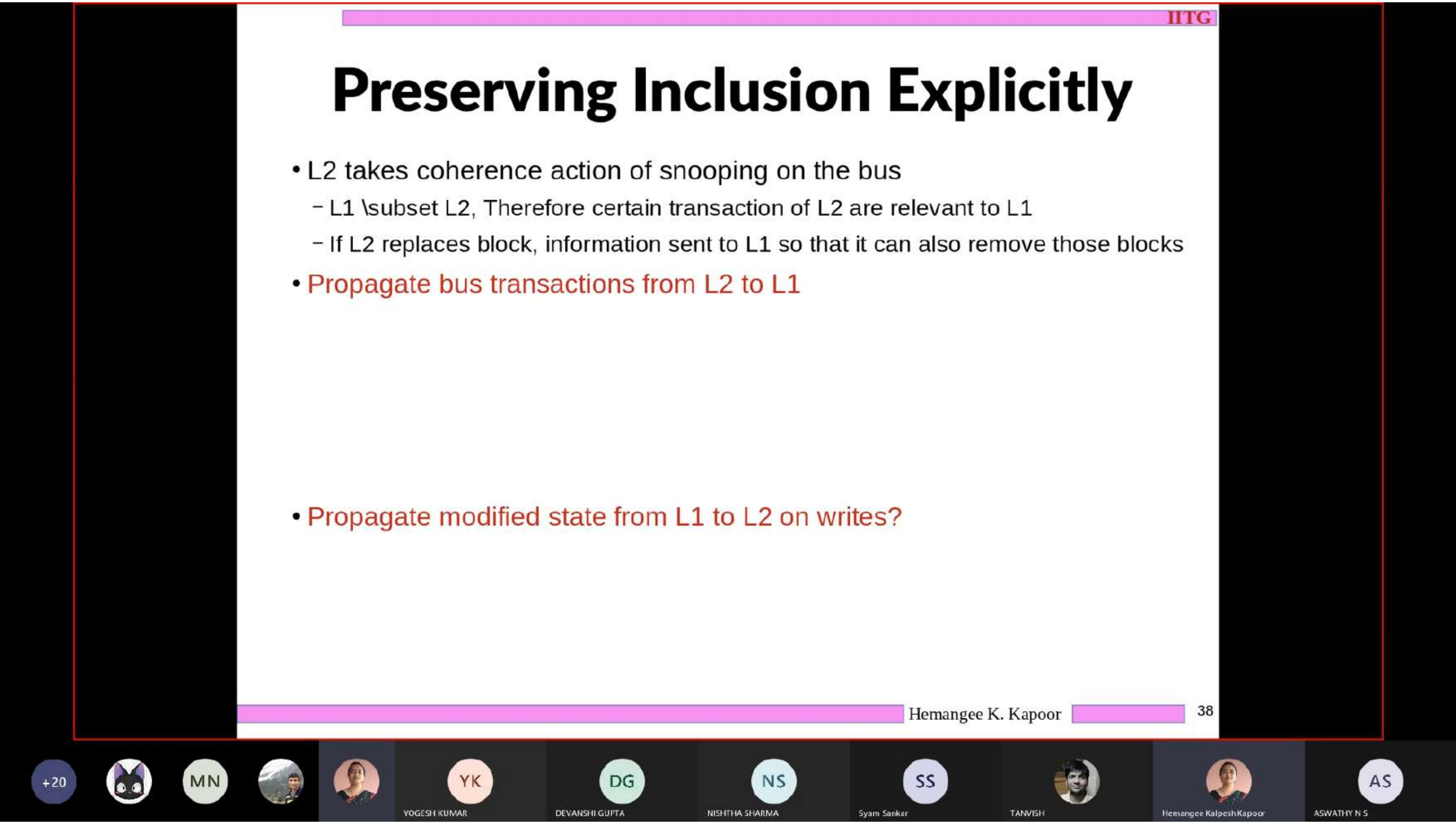
- L1 direct map: a1=1
- L2 anything: a2>=1, any replacement policy

- Same block size: b1=b2
- New block put in both L1 and L2

- #sets in L1 <= #sets in L2: n1 <= n2

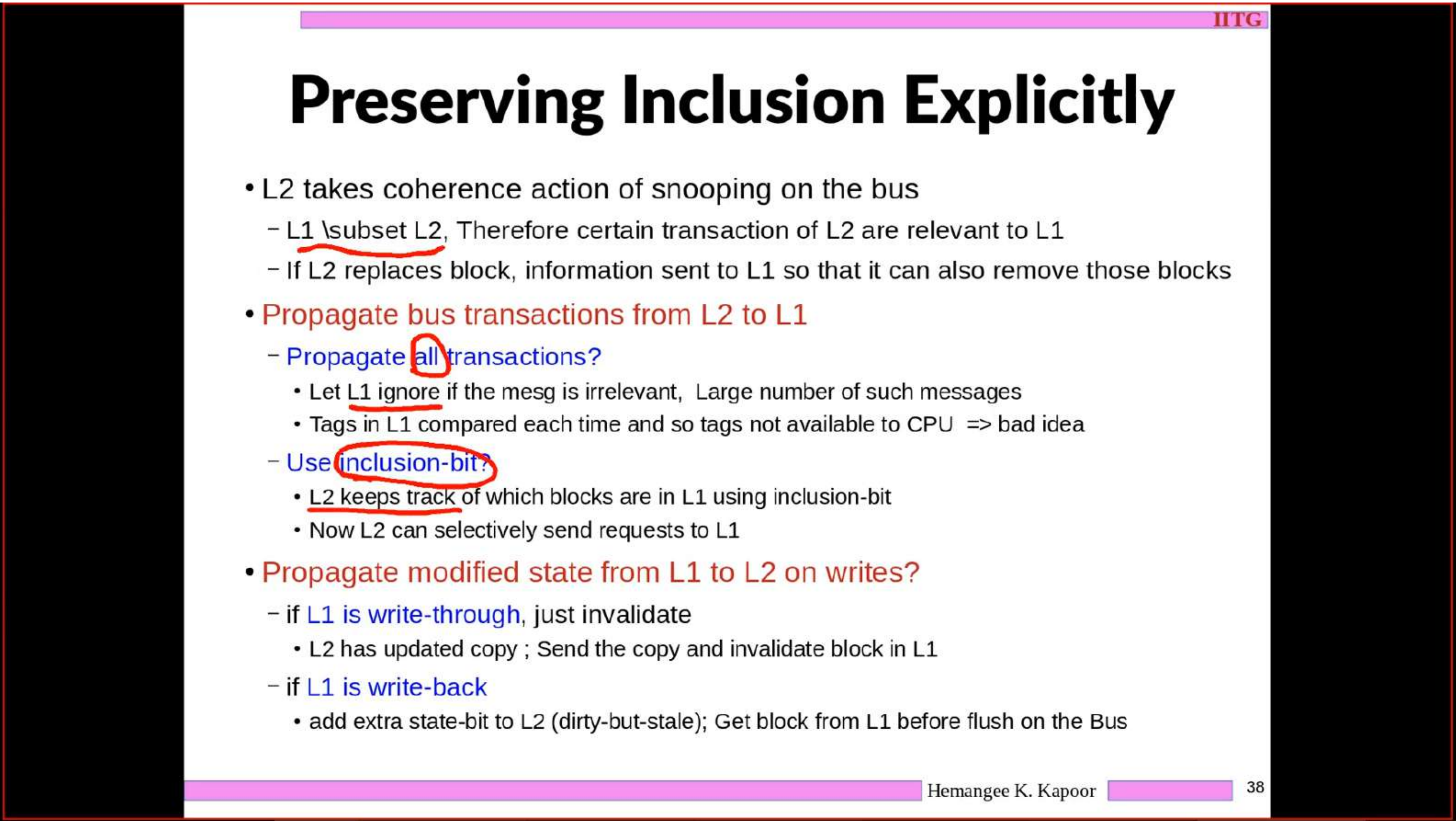
- This is a popular configuration to get the around problem of maintaining inclusion

  => use L1 as direct map cache



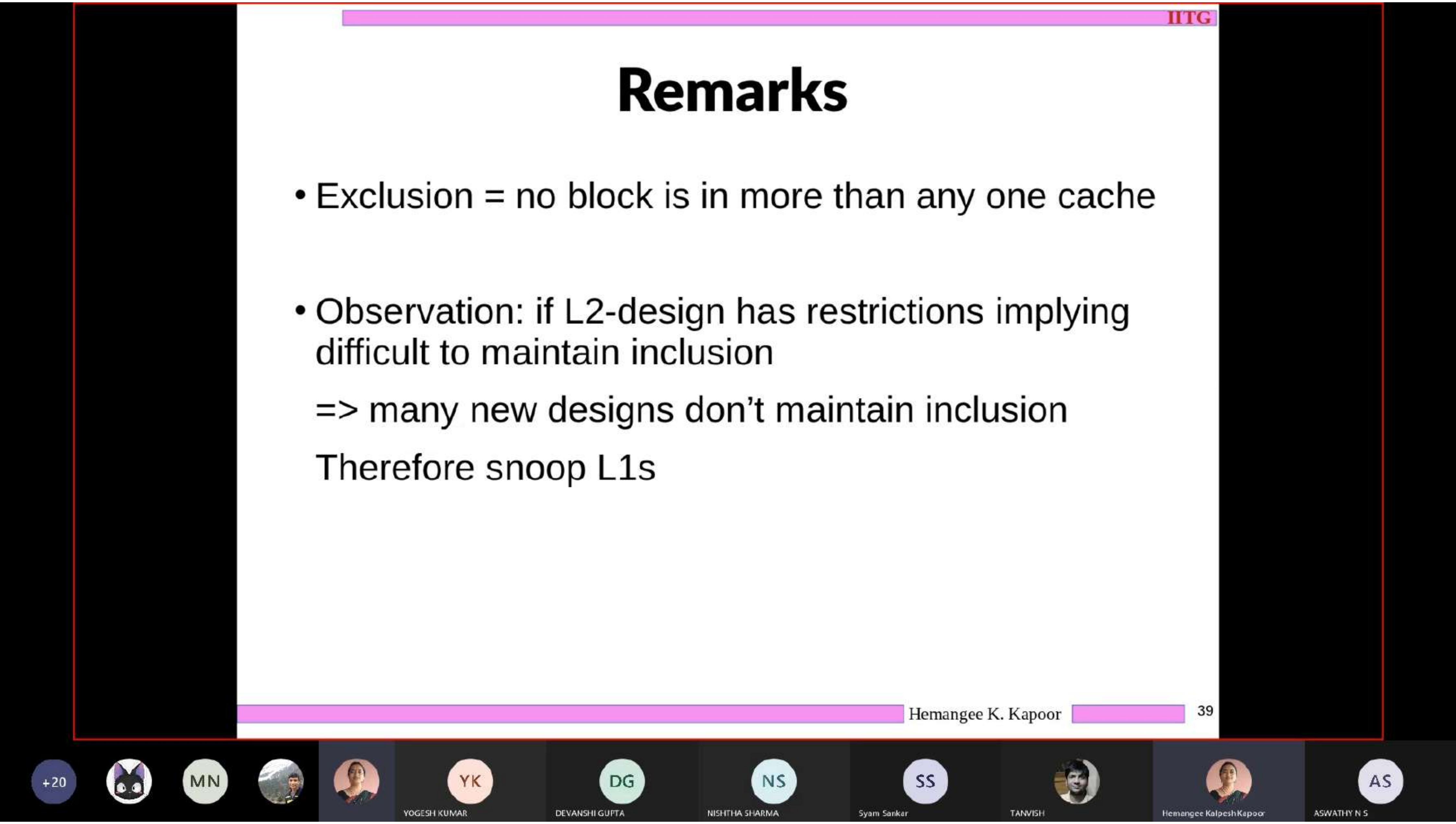
Preserving Inclusion Explicitly

- L2 takes coherence action of snooping on the bus
  - L1 \subset L2, Therefore certain transaction of L2 are relevant to L1
  - If L2 replaces block, information sent to L1 so that it can also remove those blocks
- Propagate bus transactions from L2 to L1
- Propagate modified state from L1 to L2 on writes?



Preserving Inclusion Explicitly

- L2 takes coherence action of snooping on the bus
  - L1 \subset L2, Therefore certain transaction of L2 are relevant to L1
  - If L2 replaces block, information sent to L1 so that it can also remove those blocks
- Propagate bus transactions from L2 to L1
  - Propagate all transactions?
    - Let L1 ignore if the mesg is irrelevant, Large number of such messages
    - Tags in L1 compared each time and so tags not available to CPU => bad idea
  - Use inclusion-bit?
    - L2 keeps track of which blocks are in L1 using inclusion-bit
    - Now L2 can selectively send requests to L1
- Propagate modified state from L1 to L2 on writes?
  - if L1 is write-through, just invalidate
    - L2 has updated copy ; Send the copy and invalidate block in L1
  - if L1 is write-back
    - add extra state-bit to L2 (dirty-but-stale); Get block from L1 before flush on the Bus



Remarks

- Exclusion = no block is in more than any one cache
- Observation: if L2-design has restrictions implying difficult to maintain inclusion

  => many new designs don't maintain inclusion

  Therefore snoop L1s