Relaxing all Program Orders

- No program orders are guaranteed by default, except data and control dependences within a process
- Benefit:
  - (1) multiple reads outstanding -> can be bypassed by later write
  - (2) multiple outstanding reads complete OoO -> hide read latency
  - (3) good for dynamically scheduled processors
  - (4) only these models allow key reordering and elimination of accesses -> done by compiler optimisations

    => these compiler optimisation are important + transparent to programmer.
    - Therefore these are the only reasonably high-performance memory models
- 5 models: WO, RC, RMO, PowerPC, Digital-Alpha
- All relax R->R,W ; W->W ; W->R
- All read own write early
- RC and PowerPC: read others' write early
- All models violate SC in (ex-1..6) all examples !

WO : weak ordering

- Memory operations into two categories: Data and Synchronisation
- To enforce program order between two operations, the Programmer has to identify at least one of the memory operation in program as synchronisation operation
- Model's intuition = Reorder memory operation of data region between synch-op does not affect correctness of program
- Operation declared as synch-op provide safety net for enforcing program order. Implementation can be done using counters
- Complete all Rd and Wr before synch-op. Do not issue next-op until synch done
- The Read-Write block has several instructions which can be reordered within that block
- When synch-op are infrequent, as in many parallel programs WO typically provides considerable reordering freedom to hardware and compiler

WO : examples

Lock-unlock pair
To delineate CS to update Head-ptr of list

```
P1, P2, ... Pn
...
lock(TaskQ) // acquire
  newTask -> next = Head;
  if(Head != NULL)
    Head->prev = newTask;
  Head = newTask;
unlock(TaskQ) // release
```

WO : example

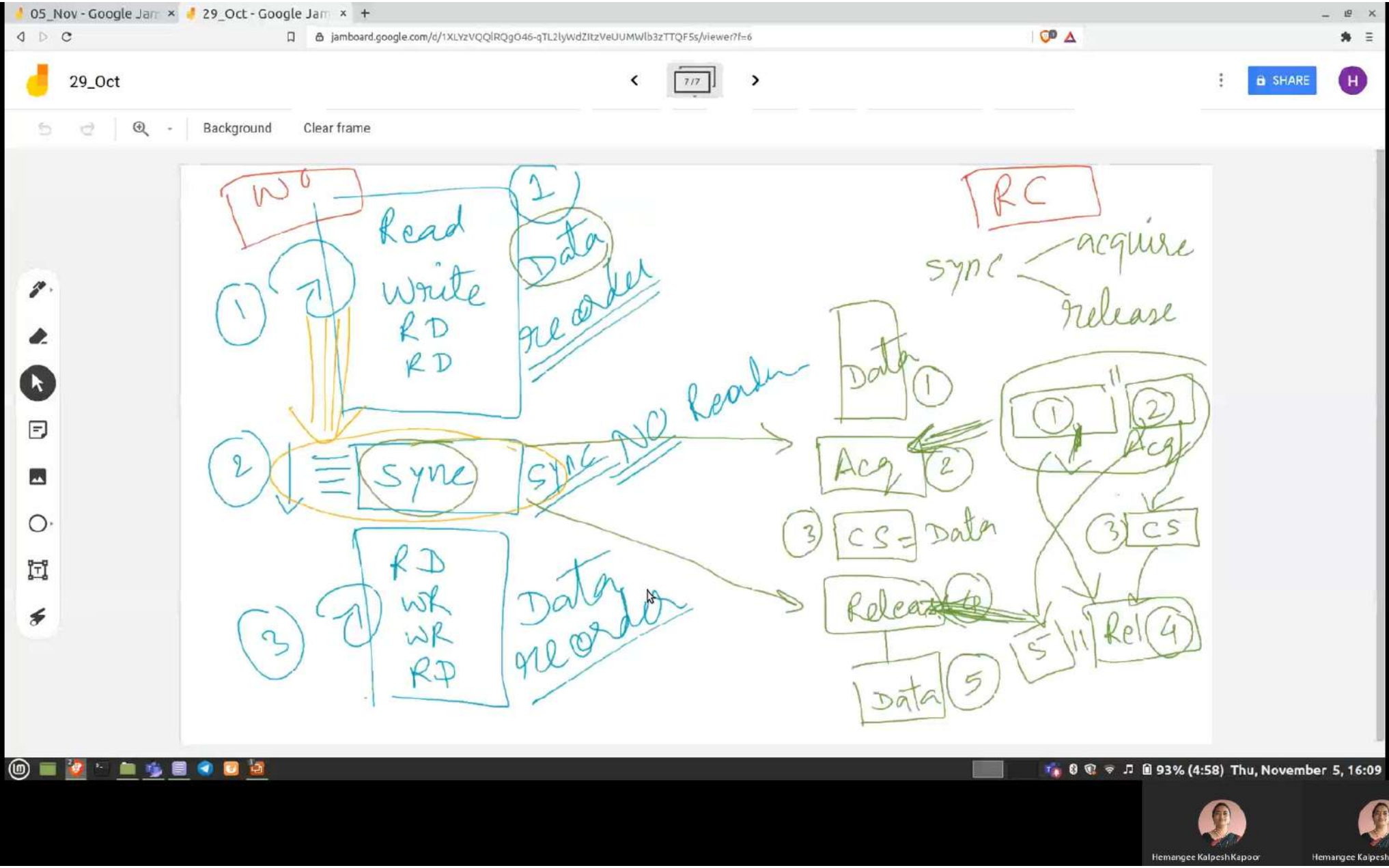
P1

```
TOP:while(flag2 == 0);// acquire
    A = 1;
    U = B;
    V = C;
    D = B*C;
    Flag2 = 0; // release
    Flag1 = 1; // release
    Goto TOP;
```

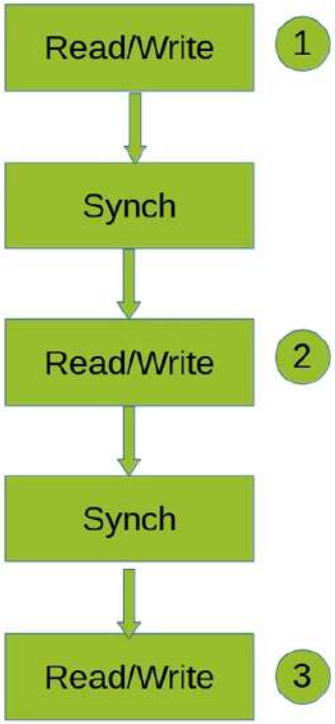
P2

```
TOP:while(flag1 == 0);// acquire
    X = A;
    Y = D;
    B = 3;
    C = D/B;
    Flag1 = 0; // release
    Flag2 = 1; // release
    Goto TOP;
```

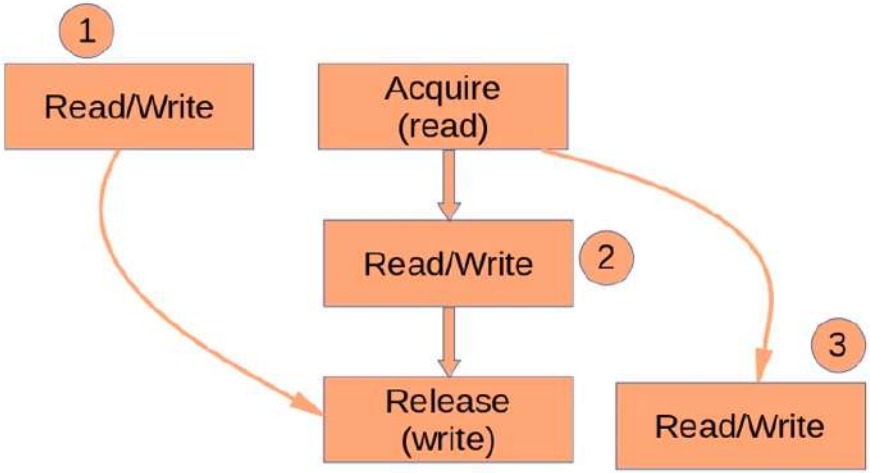
- P1 produces values for A, D and P2 consumes them.
- Flag is used as synch variable
- Therefore program semantics are not violated if we reorder between synch-instr, i.e. reorder write to A,C,V,D in P1
- Flag declared as synch will be mapped by programmer or compiler to appropriate instr in hardware


05_Nov - Google Jam
29_Oct - Google Jam
jamboard.google.com/d/1XLYzVQQlRQgO46-qTL2lyWdZItzVeUUMWlb3zTTQF5s/viewer?f=6
29_Oct
7/7
SHARE
H
Background
Clear frame
W0
Read
Write
RD
RD
1
Data
reorder
2
Sync
Sync NO Reorder
3
RD
WR
WR
RD
Data reorder
RC
sync
acquire
release
Data
Acq
CS=Data
Release
Data
Rel
CS
Hemangee Kalpesh Kapoor
Hemangee Kalpesh Kapoor

Comparison of WO and RC



WO



RC

RC : Release consistency

- Extends WO by distinguishing among synch operations: Acquire, Release
- Acquire = read (or rmw) to gain access to synch variable
  - Ex: lock(), while (flag ==0)
- Release = write (or rmw) to grant permission to another process to gain access
  - Ex: unlock() , flag1 = 1
- Purpose of acquire = delay access after acquire and it is not related to accesses above acquire
  - (Block-1) || (Acquire)
- Purpose of release = grant access to others and not relted to access that follow
  - (Block-3) || (Release)
- Block-1 -> then -> Release
- Acquire –> then -> Block-3

RC : Sufficient conditions

- (1) before issuing an operation labelled release, processor waits until all previous operations are completed in program order
- (2) operations that follow an acquire in program-order are not issued until acquire is complete
- NOTE: Implementation details not part of syllabus.

Memory models for Alpha, Sparc RMO and PowerPC

- WO and RC are specified in terms of using synch operation but they do not say how to implement them
- Alpha, RMO, PowerPC are commercial implementations of memory models that provide no ordering guarantees by default
- They provide specific hardware instructions called memory barriers or fences to enforce orderings
- The WO and RC synch-op are mapped by compiler or programmers to special hardware instructions
- Alpha:
  - Two kinds of fence instructions: MB and WMB
  - MB= waits for all previous memory accesses to complete before issuing new ones
  - WMB= waits for all previous writes to complete. A read issued after a WMB can still bypass a write access issued before the WMB, but writes after WMB must wait

Memory models for Alpha, Sparc RMO and PowerPC

- RMO
  - Has MEMBAR instr in 4 flavours:
  - To specify ordering between R-R, R-W, W-R, W-W
  - If we use MEMBAR to order a WR->RD we do not need to use read-modify-write to achive this order
- Alpha and RMO do not require a safety net for write atomicity
- PowerPC
  - Has single SYNC instruction equivalent to MB in Alpha
  - In PowerPC writes are not atomic: as it allows a write to be seen early by another processor's Read
  - i.e. May need to use rmw operations to make write appear atomic

Portability

- All memory models allow different reorderings and order enforcing operations/instr => portable?
- Ex: A program with enough memory barriers that works correctly on TSO will not necessarily work correctly on an RMO system -> On RMO it will need more special operations
- We need higher-level interfaces that are more convenient to the programmer and portable to the different systems while still enjoying the performance benefits of reorderings

The Programmer's Interface

- All programming interfaces are inspired by the WO and RC models, which are the most relaxed models
- Programmer's contract:
  - Program must include point-to-point event synchronisation using flags, which are explicitly labelled
- System's contract:
  - Compile or run-time library translates these sync-operations into appropriate order-preserving operations (Mem-bar or fences) depending on the system specifications.
  - It re-orders between sync operations
  - OoO, compiler, reorder, etc.
- Thus system = (hardware + compiler) guarantees SC
- Programs that label all synch-events are called synchronised programs
- Programmer has to know which operations are to be synchronised, i.e. which ones to label
- Of course the programmers know which will be synch operations

Identify synch-events

- How to identify?
  - Lock = acquire, unlock = release, arrive at barrier = release, exit barrier = acquire
  - Programmer adds lock, barrier. Language may give labels. Library/compiler maps
- A generic method to identify synch-events: uses following definitions. Used when all else fails
- (1) Conflicting operators
  - Two memory operations from different processes if they access same location and at least one is write
- (2) Competing operators \subset of conflicting operators
  - Two memory operations from different processes compete if they appear next to each other in an SC execution, i.e. one after another, without any interleaving memory operation on shared data between them
- (3) Synchronised program
  - A parallel program is synchornised if all competing memory operations have been labelled as synch-operations. [perhaps labelled using acquire = read, release = write]

05_Nov - Google Jam | 29_Oct - Google Jam | +

jamboard.google.com/d/1ZF-_m_1vpLaM43xjQfb4G-g28VhbHxnaV__rQ1OHg2s/viewer?f=1

05_Nov

2/3

Next frame

SHARE

H

Background Clear frame

Synchronised Program

Conflicting

P1 || P2

Rd A	wr A
wr A	wr A

Competing in SC exe →

P1 : rd A
P2 : wr A

or

P1 wr-A
P2 wr-A

Sync Prog = All competing operations are labelled as synch op

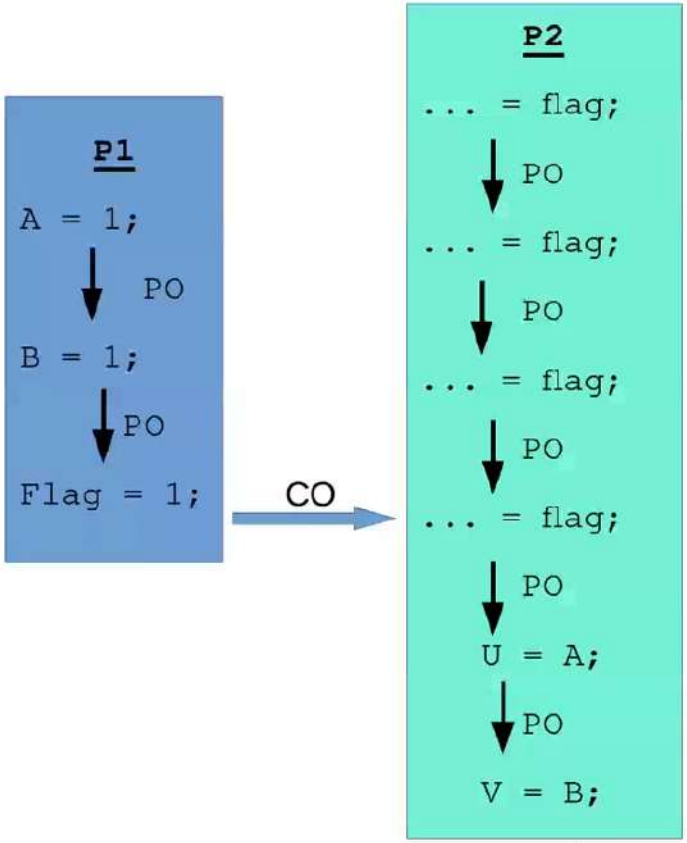
86% (3:51) Thu, November 5, 16:28

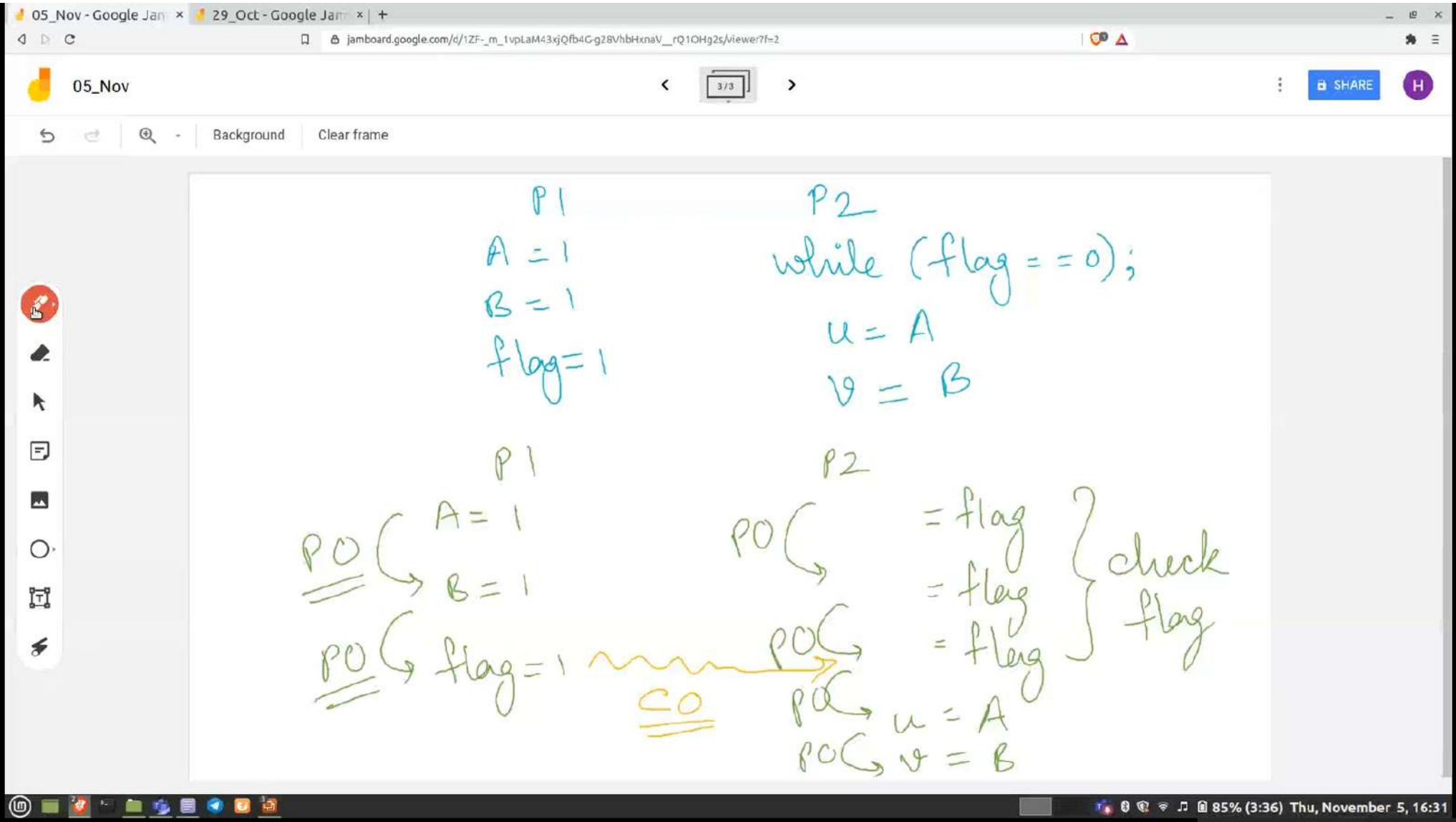
Hemangee Kalpesh Kapoor

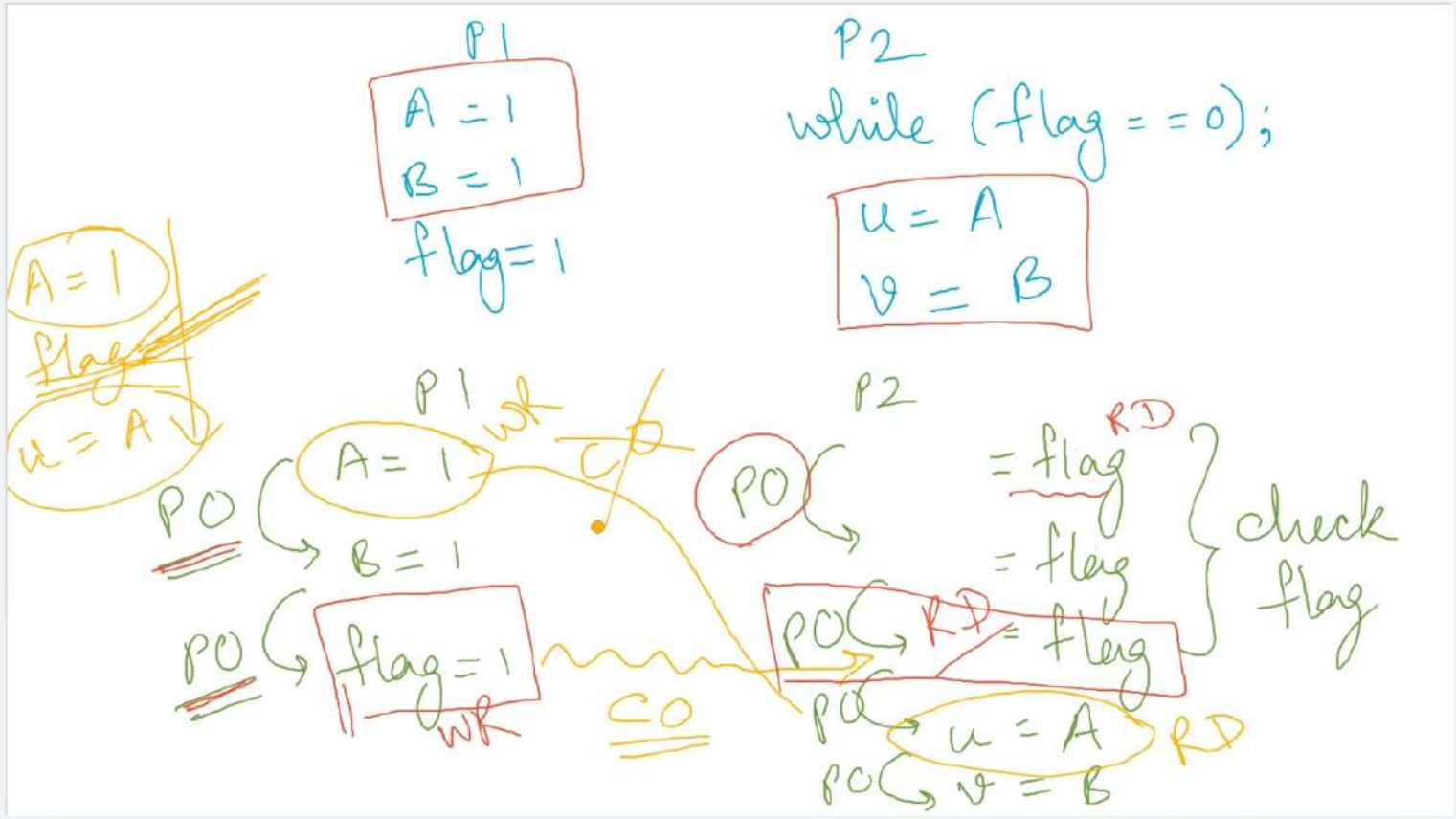
Hemangee Kalpesh Kapoor

Ex: synchronised program

- PO = program order
- CO = conflict order
- Declare flag as synch-op
- Write to A, B, (by P1) and read of A, B in P2 are in 'PO' and separated by 'CO'
- Therefore not required as synch-op
- In the chain of SC accesses, there is at least one link of PO then not competing
- In P1, P2, access to A, B, are conflicting but separated by flag Rd/ Wr. Therefore not competing in chain




05_Nov - Google Jam
29_Oct - Google Jam
jamboard.google.com/d/1ZF-_m_1vpLaM43xjQfb4G-g28VhbHxnaV__rQ1OHg2s/viewer?f=2
05_Nov
3/3
SHARE
H
Background
Clear frame
P1
A = 1
B = 1
flag = 1
P2
while (flag == 0);
u = A
v = B
P1
P0
A = 1
B = 1
P0
flag = 1
C0
P2
P0
= flag
= flag
= flag
check flag
P0
u = A
P0
v = B
85% (3:36) Thu, November 5, 16:31
Hemangee Kalpesh Kapoor
Hemangee Kalpesh Kapoor


P1
A = 1
B = 1
flag = 1
P2
while (flag == 0);
u = A
v = B
A = 1
flag
u = A
P1
wr
A = 1
P0
B = 1
P0
flag = 1
wr
C0
P2
P0
RD
= flag
= flag
= flag
check flag
P0
RD
P0
u = A
RD
P0
v = B

Synchronisation
H&P ch-4
PCA book sec 5.5

Synchronisation

- Why Synchronise?
  - Need to know when it is safe for different processor to use shared data
- Synchronisation mechanisms are built in software programs that rely on hardware supplied synchronisation instructions
- The key hardware primitive is an uninterrupted instr to fetch and update memory (an atomic operation)
- For large-scale multiprocessors, synch can become a bottleneck
  - Therefore techniques are needed to reduce contention and latency of synchronisation
- There is a long history of what hardware support in each industry

Synchronisation Event

- (1) Acquire
  - Method where process tries to get right to synchronise (ex: enter critical section)
- (2) Waiting Algo
  - Waits for synchronisation to become available (ex. Waiting to acquire a locak when lock is not free)
- (3) Release
  - Method to allow another process to proceed past synchronisation (ex. Unlock operation or notify using flag in point-to-point event synch)
- We will see
  - Uniterruptible instructions
  - Spin-locks
  - Point-to-point event synch
  - Global Barrier Synch

Un-interruptible Instr to Fetch & Update memory

- Atomic exchange
  - Interchange a value in a register for a value in memory
    - 0 => sync var free
    - 1 => sync var locked and un-available
    - TASKS (1) set reg=1 and swap
    - (2) value returned in register tells if success in getting lock
    - 0 = success, 1 = some other proc has already locked
- Test & Set
  - Tests a value and set it if the value passes that test
- Fetch & Increment
  - It returns the value of a memory location and atomically increments it. 0 => sync var is free
- Key is that exchange operation is indivisible
- Hard to have read and write in 1 instruction. Use 2 instr instead: LL-SC

05_Nov - Google Jam

jamboard.google.com/d/1ZF-_m_1vpLaM43xjQfb4G-g28VhbHxnaV__rQ1OHg2s/viewer?f=3

05_Nov

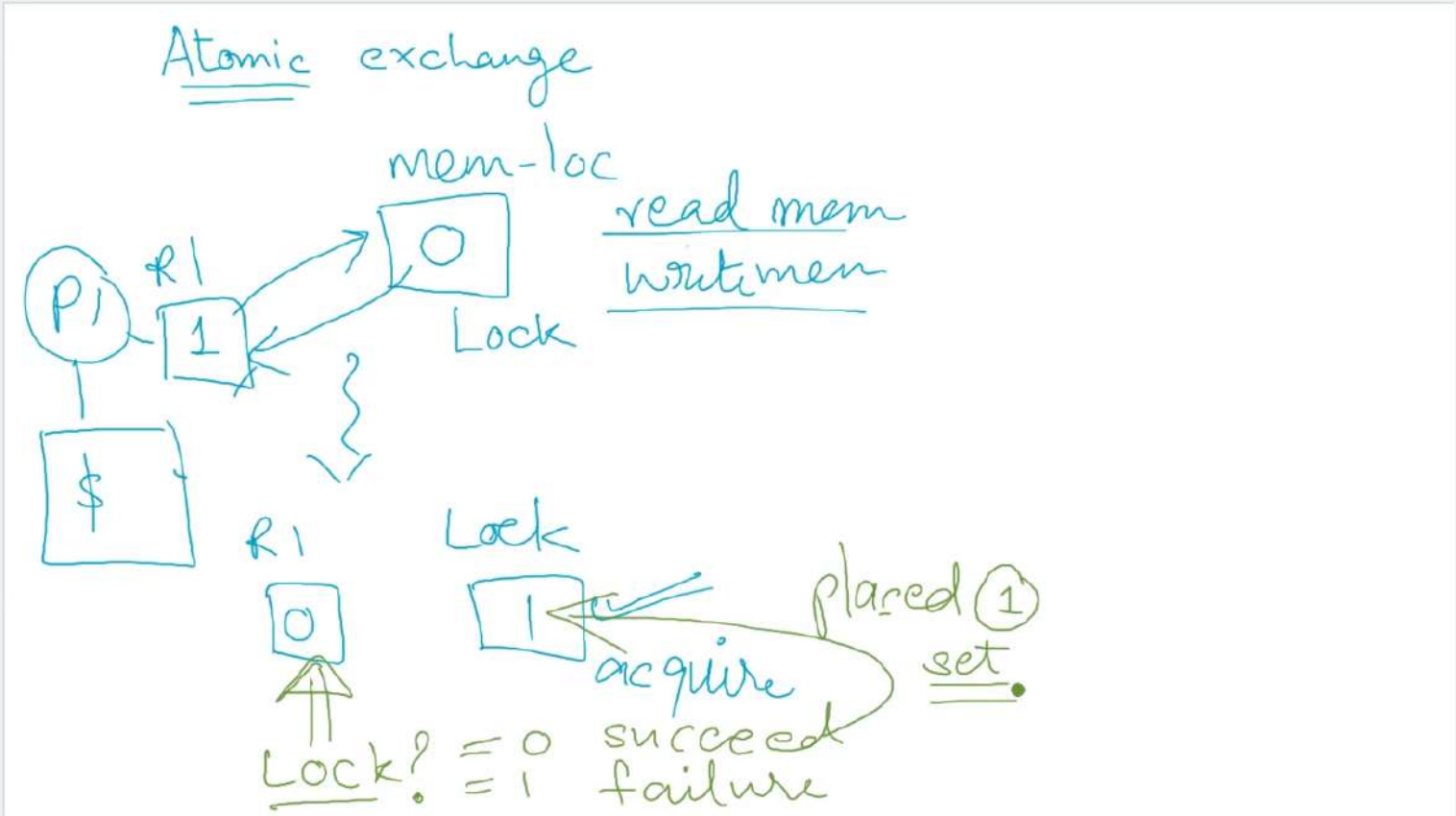
4 / 4

SHARE

H

Background

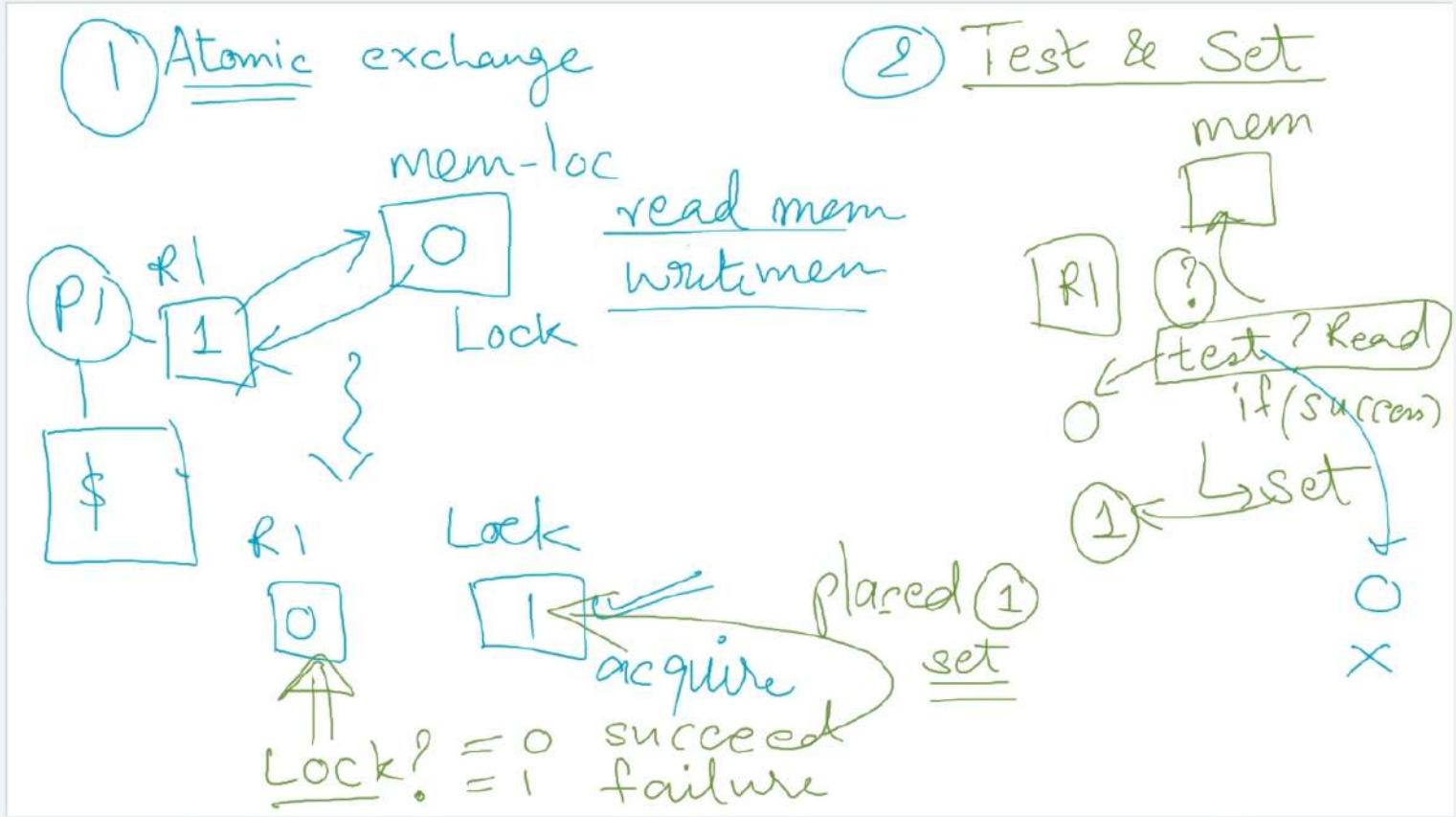
Clear frame

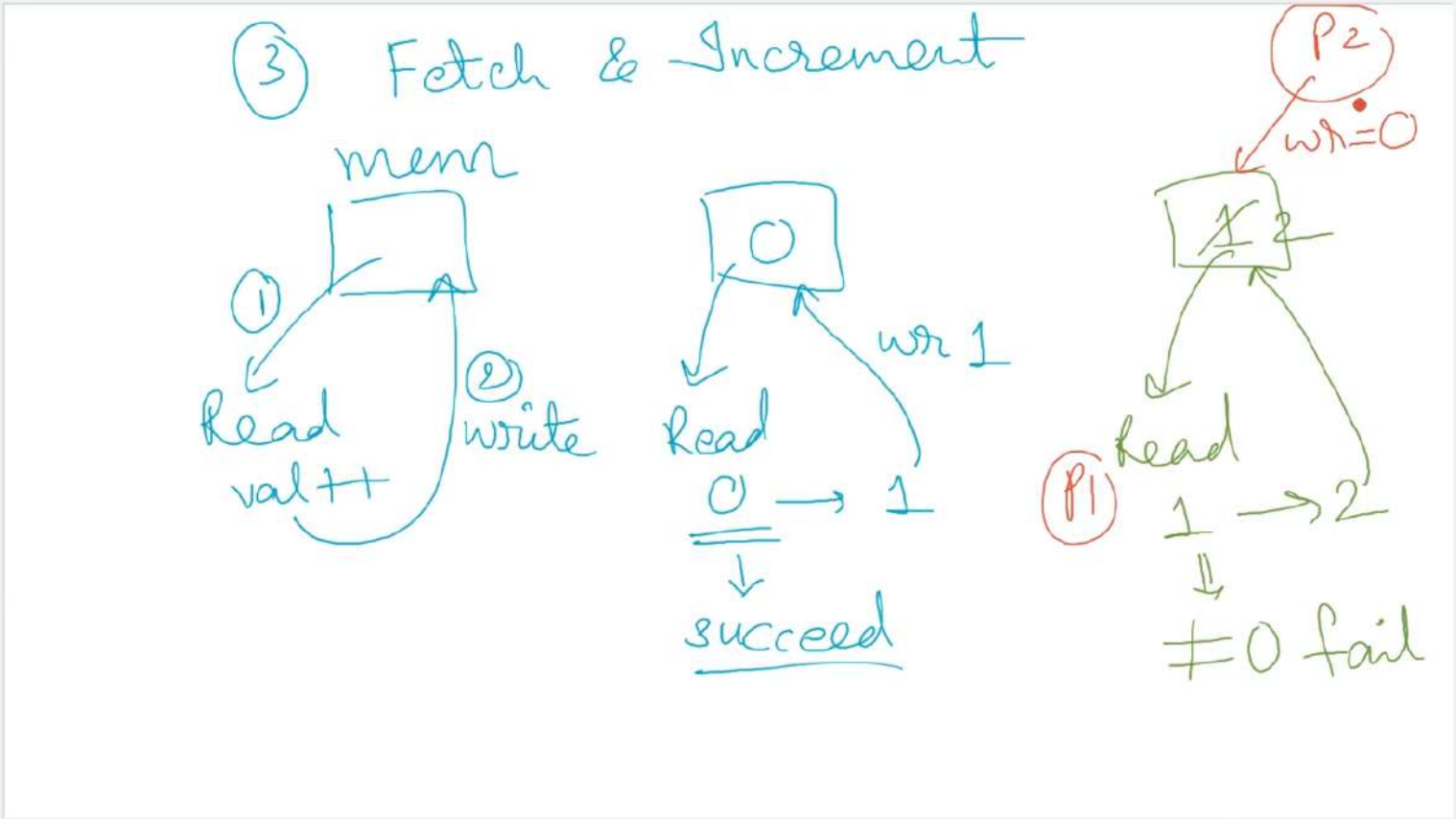


94% (3:20) Thu, November 5, 21:56

Hemangee Kalpesh Kapoor

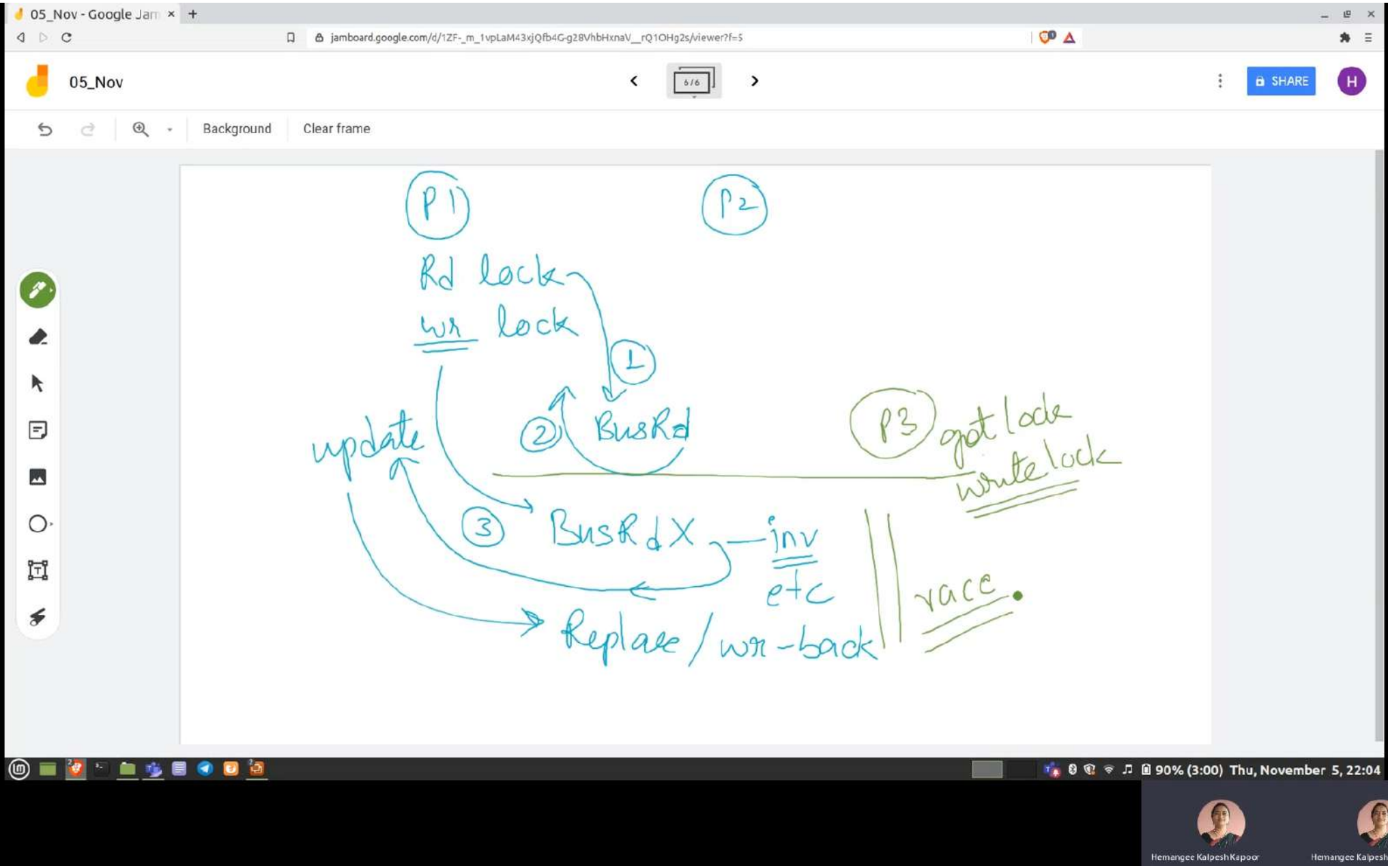
Hemangee Kalpesh Kapoor


1 Atomic exchange
mem-loc
read mem
write mem
P1
R1
1
0
Lock
$
R1
0
Lock
1
acquire
placed 1
set
Lock? =0 succeed
=1 failure
2 Test & Set
mem
R1
?
test ? Read
if (success)
0
1
set
0
X



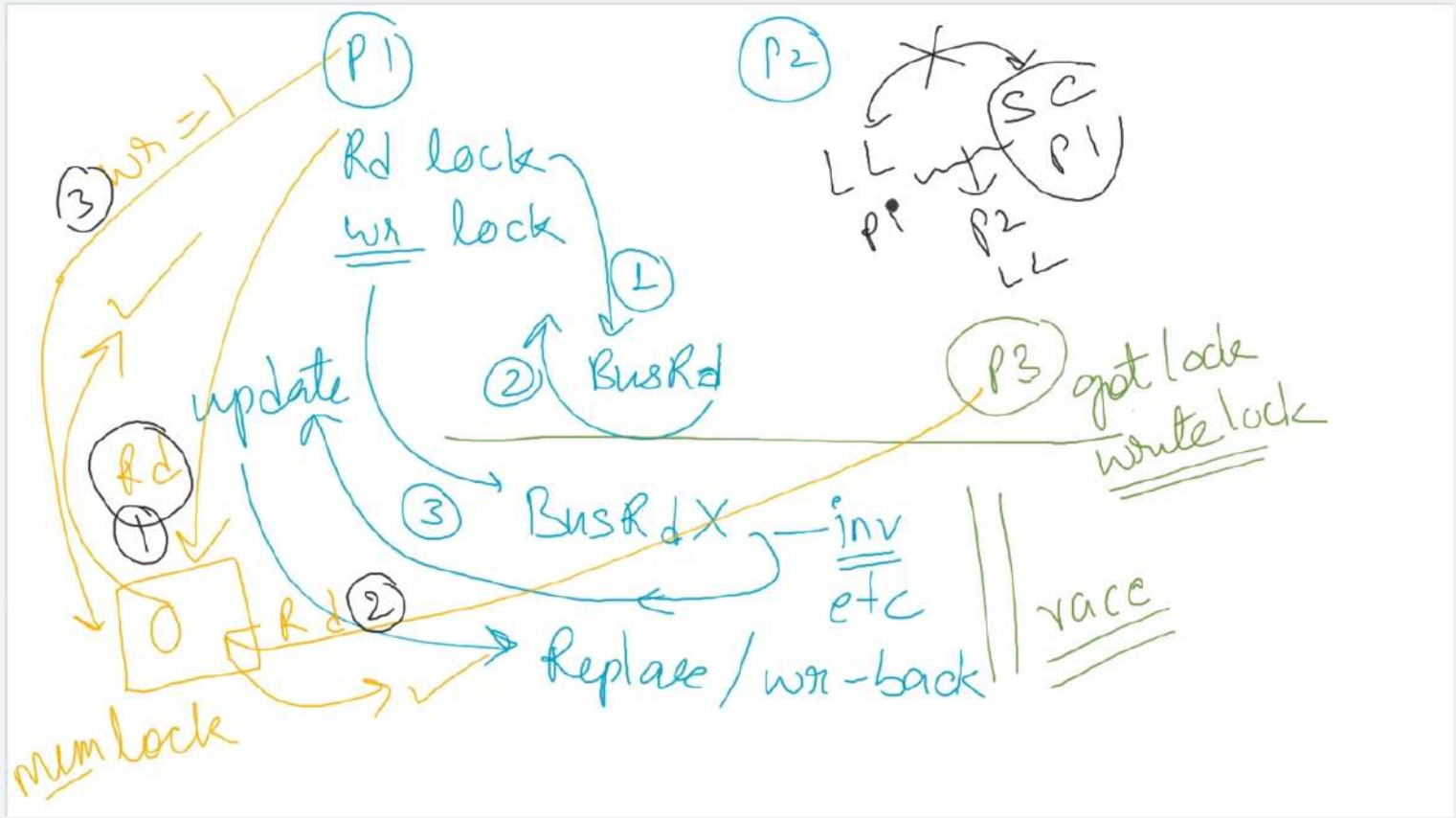
Un-interruptible Instr to Fetch & Update memory

- Atomic exchange
  - Interchange a value in a register for a value in memory
    - 0 => sync var free
    - 1 => sync var locked and un-available
    - TASKS (1) set reg=1 and swap
    - (2) value returned in register tells if success in getting lock
    - 0 = success, 1 = some other proc has already locked
- Test & Set
  - Tests a value and set it if the value passes that test
- Fetch & Increment
  - It returns the value of a memory location and atomically increments it. 0 => sync var is free
- Key is that exchange operation is indivisible
- Hard to have read and write in 1 instruction. Use 2 instr instead: LL-SC


05_Nov - Google Jam
jamboard.google.com/d/1ZF-_m_1vpLaM43xjQfb4G-g28VhbHxnaV__rQ1OHg2s/viewer?f=5
05_Nov
6/6
SHARE
H
Background
Clear frame
P1
P2
Rd lock
wr lock
1
2
BusRd
update
P3
got lock
write lock
3
BusRdX
inv
etc
Replace / wr-back
race
90% (3:00) Thu, November 5, 22:04
Hemangee Kalpesh Kapoor
Hemangee Kalpesh Kapoor

Implementing LL-SC

- Load Linked (or load locked) + Store Conditional
  - Load linked reads memory and returns initial content
  - Store conditional fails if the contents of memory location have changed after the load-linked => Returns 0 = failure
  - SC also fails if processor does context-switch between LL and SC
  - SC success returns = 1
- Implementation
  - Next slide
- What instruction between LL-SC?
  - Simple register operations
  - Number of instr also small so that SC will get chance to execute eventually (in case of scheduling other Mps or processes getting access to synch-var)


P1
P2
SC
P1
LL
wr
P1
P2
LL
Rd lock
wr lock
1
2
BusRd
P3
got lock
write lock
update
Rd
1
3
BusRdX
inv
etc
race
O
Rd
2
Replace / wr-back
mem lock

05_Nov - Google Jam

jamboard.google.com/d/1ZF-_m_1vpLaM43xjQfb4G-g28VhbHxnaV__rQ1OHg2s/viewer?f=6

05_Nov

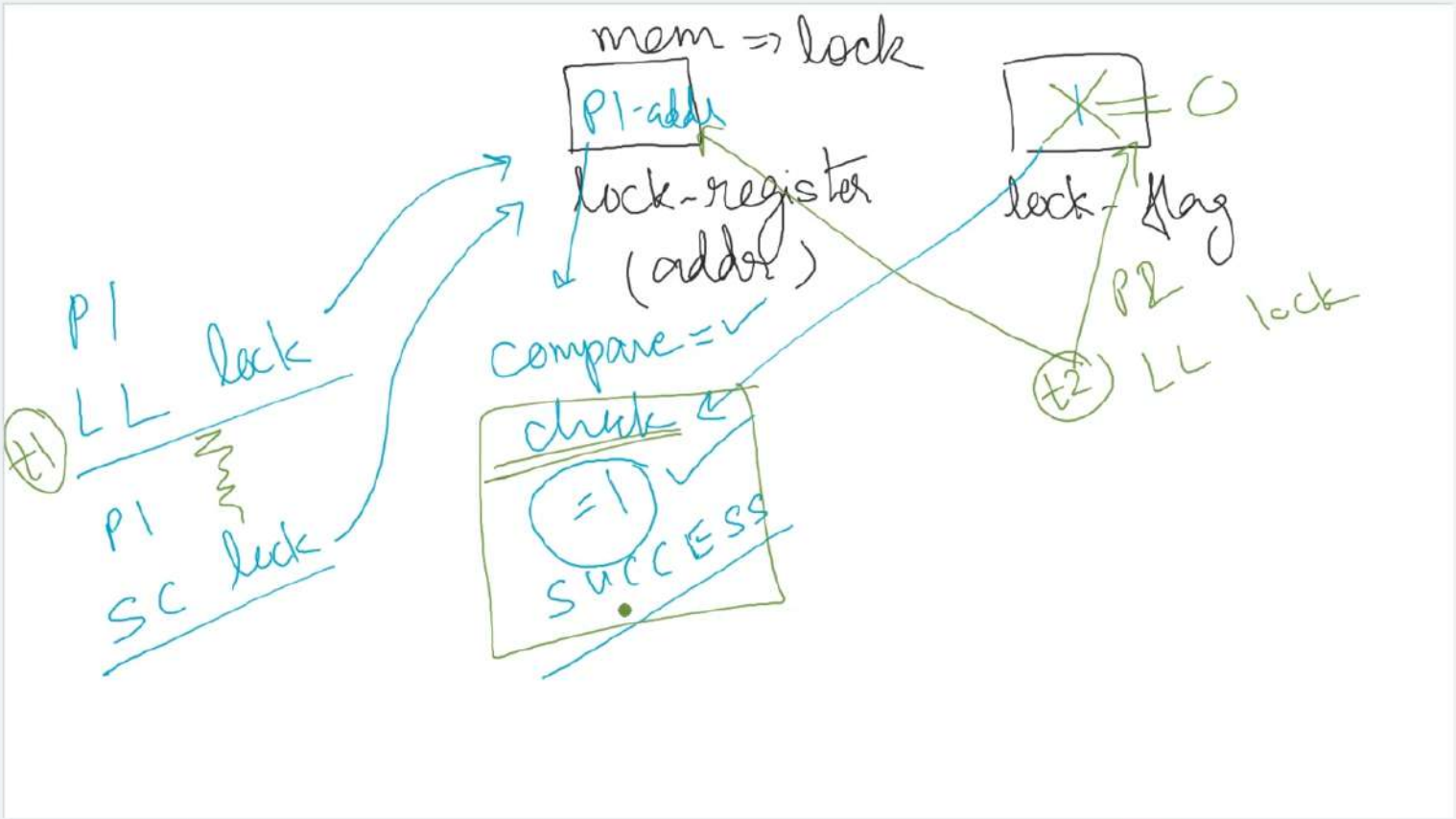
7/7

SHARE

H

Background

Clear frame



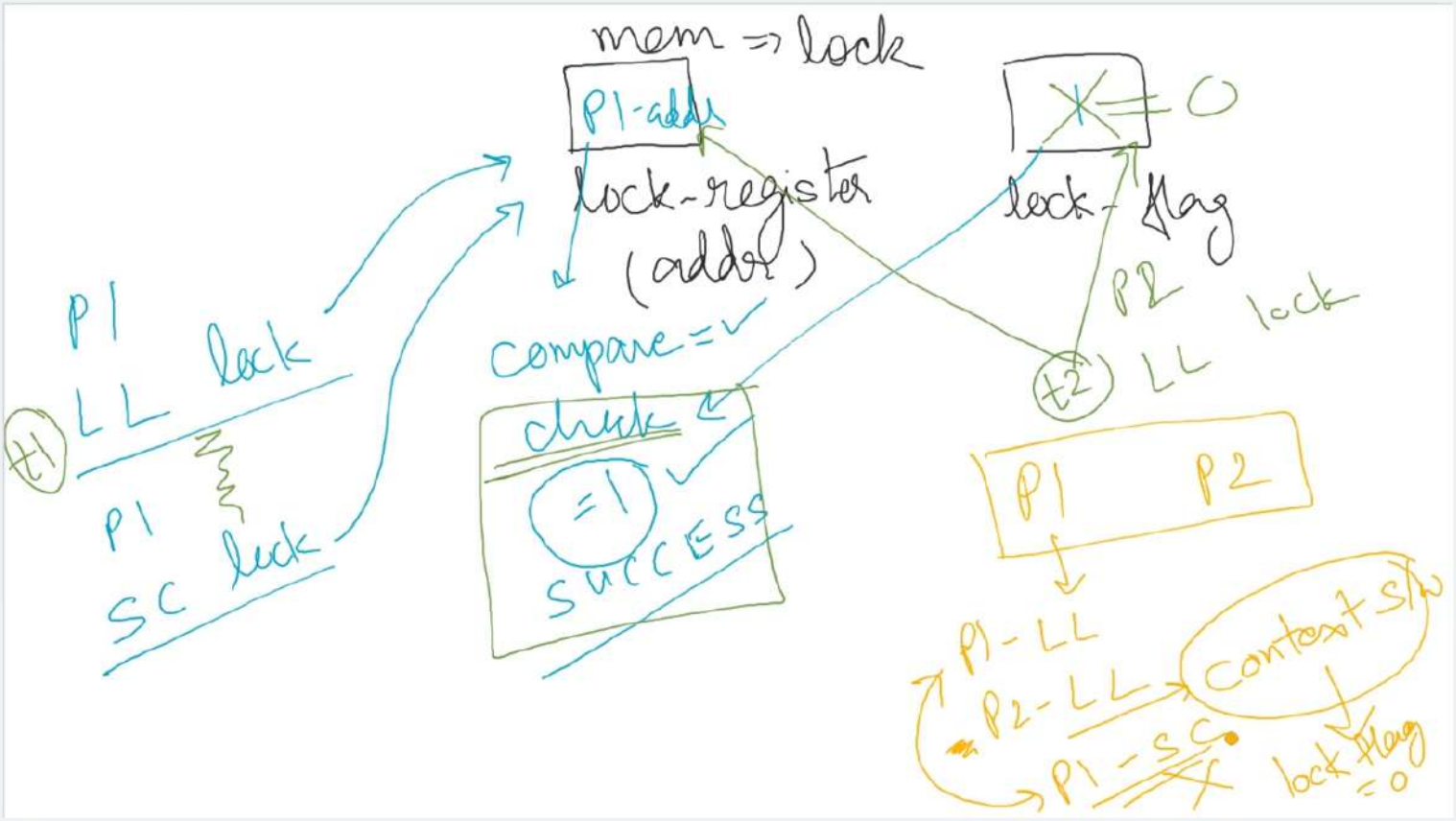
88% (3:02) Thu, November 5, 22:10

Hemangee Kalpesh Kapoor

Hemangee Kalpesh Kapoor

Implementing LL-SC

- Each processor (having cache) has 1 lock-flag + 1 lock-addr-reg
  - On LL instr lock-flag = 1, address = written in register
  - SC will fail if lock flag=0
- 3 cases
- (1) lock flag will be reset on any inv or update going on the bus for the address in lock-addr-reg
- (2) lock flag resets, if lock variable is replaced in the cache. Since then the processor will no longer see inv/upd on that address
- (3) lock flag is reset on context switch. Because after LL, a context switch will cause the SC of another process to succeed because of an LL of old process !
- SC check lock-flag. If '0' SC fails. Indicating an intervening conflicting write has occurred


mem => lock
P1-addr
lock-register
(addr)
lock-flag
X = 0
P1
LL lock
t1
P1
SC lock
compare = 1
check
=1
SUCCESS
t2
P2
LL
lock
P1 P2
P1-LL
P2-LL
P1-SC
context s/w
lock flag
= 0

Livelock situations?

- Subtle livelock situations may arise
- (1) if cache replaces the block holding lock variable
  - Each time LL needs load new block etc.
  - Use split instr/data cache, set-assoc cache, etc.
  - Don't have memory reference instr between LL-SC
- (2) P1 || P2
- If P1-LL, P2-LL, then SC- P1, P2, P1, P2
- If SC is a write then it inv P1-LL and vice versa
- Therefore Livelock
- Hence SC is not normal write
  - It does not generate inv if it fails.
  - i.e. it checks lock flag. If flag=1 -> inv + write. If flag=0 -> return
- Write assmbly code of EXCH. Show cache coherent Bus actions to obtain lock

Ex: using LL-SC do atomic-swap and fetch-&-increment

```
Atomic Swap : R4 <=> [R1]

TRY: MOV R3, R4       // move exchange value
     LL  R2, 0(R1)    // load linked
     SC  R3, 0(R1)    // store conditional
     BEQZ R3, TRY     // branch if store fails (R3 == 0)
     MOV R4, R2       // put load value in R4
```

```
Fetch-and-increment

TRY: LL  R2, 0(R1)       // load linked
     DADDUI  R3, R2, #1  // increment (OK if reg-reg)
     SC  R3, 0(R1)       // store conditional
     BEQZ R3, TRY        // branch if store fails (R3 == 0)
```