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

Sufficient conditions for SC...

- Program order is defined in terms of the source code
- Compiler should not change the order of memory operations that it presents to the hardware (processor)
- Otherwise SC from the programmer's perspective may be compromised even before hardware gets involved
- Unfortunately many compiler optmisations are commonly employed violating SC
- e.g. compilers routinely reorder access to different locations within a process => process may issue accesses out of program order seen by the programmer
- Explicit parallel programs use uniprocessor compilers, which are concerned in preserving orders between accesses to same location
- Advanced optimisations further optimise and even eliminate certain memory operations !
- So stop compilers from optimisations, programmer can insert the keyword telling compilers not to reorder them
- e.g. the **volatile** qualifier before variable declaration prevents the variable getting allocated to register and also prevents it from being reordered with any memory operation before or after it in program order

Implications of SC

- Memory consistency defines the constraints on the order in which memory operations appear to get executed wrt one another
- This enables the programmer to reason about the outcome of the program
- Software that interacts with the next layer must know the memory consistency model
- We will focus on consisteny model as seen by the programmer
  - i.e. interface between the programmer and the rest of the system (= compiler + OS + hardware)
  - e.g. the processor may maintain all the orders (i.e. it does not reorder) but if the compiler has already reordered then the programmer cannot reason by the simple model supported by the hardware
- Consistency model has implications for programming languages, compilers and hardware

Implications of SC ...

- For compiler and hardware it puts restrictions on what they can reorder and what they cannot (which would violate the constraints). What performance optimisations are allowed
- Programming languages must provide mechanisms that help introduce such constraints
- The fewer reorderings that we allow, the easier it becomes for the programmer to reason BUT the puts more constraints on performance optimisations
- Therefore memory consistency model must be able to strike a balance between programming complexity and performance
- Model should be portable = implementable on many platforms while preserving the semantics
- SC gives the programmer an intuitive semantics of program order and interleaving – and can be implemented by satisfying all the sufficient conditions

Drawback of SC

- Preserving strict ordering restricts performance optimisations
- With high cost of memory access, the computer systems can give good performance by hiding the access latency by reordering and overlapping memory accesses
  - Preserving SC clearly does not permit this
- With SC at the programmers interface, the compiler cannot reorder memory accesses even if they are to different locations => disallows critical performance optimisations such as code motion, common-subexpression elimination, software pipelining and register allocation
- If the sufficient conditions are met, a processor has to wait for an access to complete before issuing the next one, so most of the latency suffered by memory references is seen as the processor stall time
- We need to do something about this performance problem !

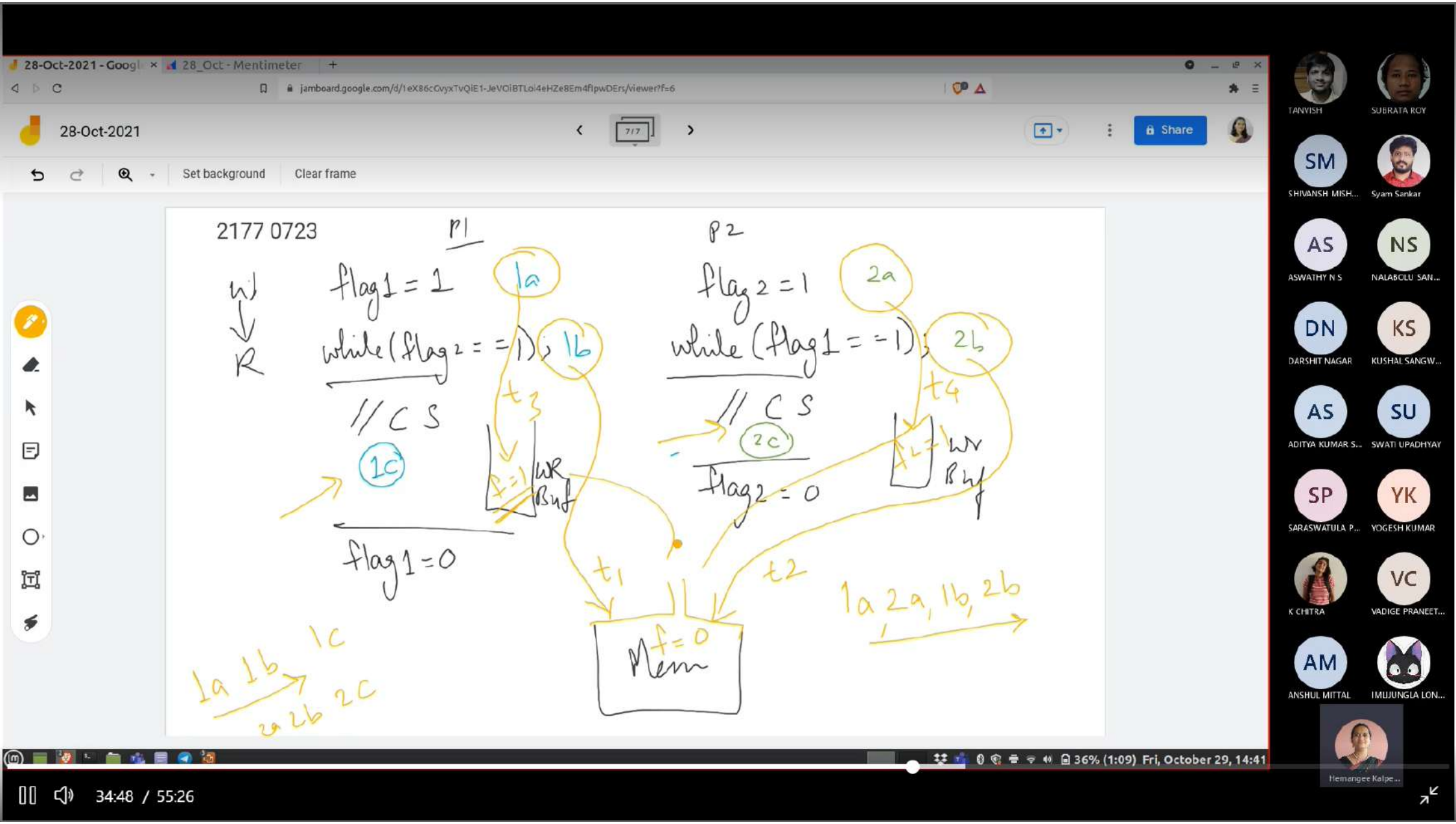
How to hide latency?

- (1) Preserve SC, compiler does not reorder, but hide latency by prefetching data. But actual read write are not issued until previous ones have completed
- (2) Preserve SC but not all the sufficient conditions are satisfied at the programmers interface
  - Compiler can reorder as long as it can guarantee that SC will not be violated in the results. Such compiler algorithms are however very expensive
  - At hardware level, memory operations are issued and executed out-of-program order but are guaranteed to become visible to other processors in program order
    - Suitable in dynamically scheduled processors that have instruction re-order/lookahead buffers where instructions enter in program order, executed in any order but retire (complete) in program order
    - Use branch prediction and speculative execution
    - Have mechanisms to roll-back in case of mis-predictions
    - NOTE that stores cannot be done in lookahead manner ... as stores will become visible immediately
  - All such techniques are very complex to implement in hardware and their advantages are not substantial
  - These techniques work for processors, but do not help the compilers in doing reorderings .. which is critical for optimisation

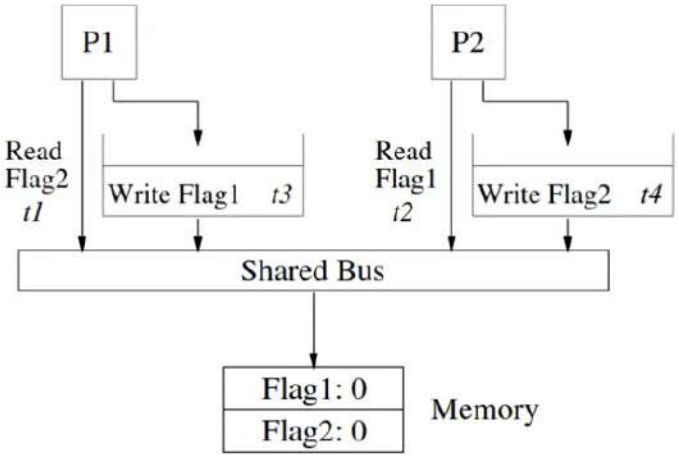
Importance of maintaining W->R

- Problem in SC (no-cache) due to write buffers
- We need to maintain program order between write -> read
- Arch: Bus-based, No-cache, has write buffers and continue next instruction i.e. subsequent reads can go past the write if the location is different
- EX: shows how write buffers violate SC

```
   P1                              P2

flag1 = 1;                     flag2 = 1;

if(flag2 == 0)                 if(flag1 == 0)

  // critical section            // critical section
```


28-Oct-2021 - Googl
28_Oct - Mentimeter
jamboard.google.com/d/1eX86cCvyxTvQlE1-JeVOiBTLoi4eHZe8Em4fIpwDErs/viewer?f=6
28-Oct-2021
7/7
Share
Set background
Clear frame
2177 0723
P1
P2
flag1 = 1
flag2 = 1
while(flag2 == 1);
while(flag1 == 1);
// CS
// CS
1a
1b
1c
2a
2b
2c
t3
t4
t1
t2
WR Buf
flag1 = 0
flag2 = 0
Mem
1a 2a, 1b, 2b
1a 1b 1c
2a 2b 2c
TANVISH
SUBRATA ROY
SHIVANSH MISH...
Syam Sankar
ASWATHY N S
NALABOLU SAN...
DARSHIT NAGAR
KUSHAL SANGW...
ADITYA KUMAR S...
SWATI UPADHYAY
SARASWATULA P...
YOGESH KUMAR
K CHITRA
VADIGE PRANEET...
ANSHUL MITTAL
IMIJUNGLA LON...
Hemangee Kalpe...
36% (1:09) Fri, October 29, 14:41
34:48 / 55:26

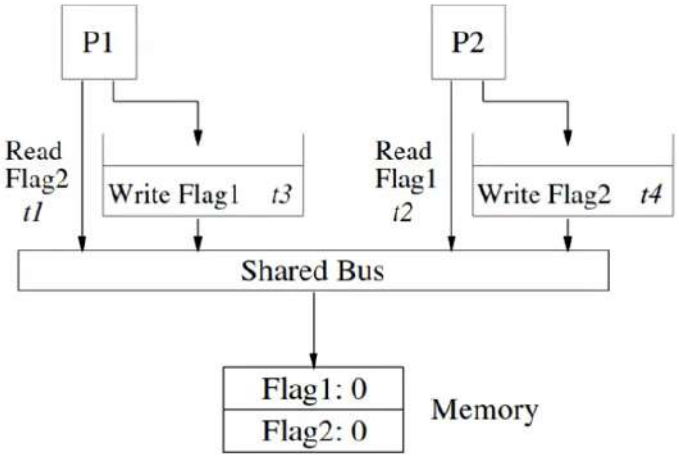
Write buffer



P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a) write buffer

Write buffer



P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a) write buffer

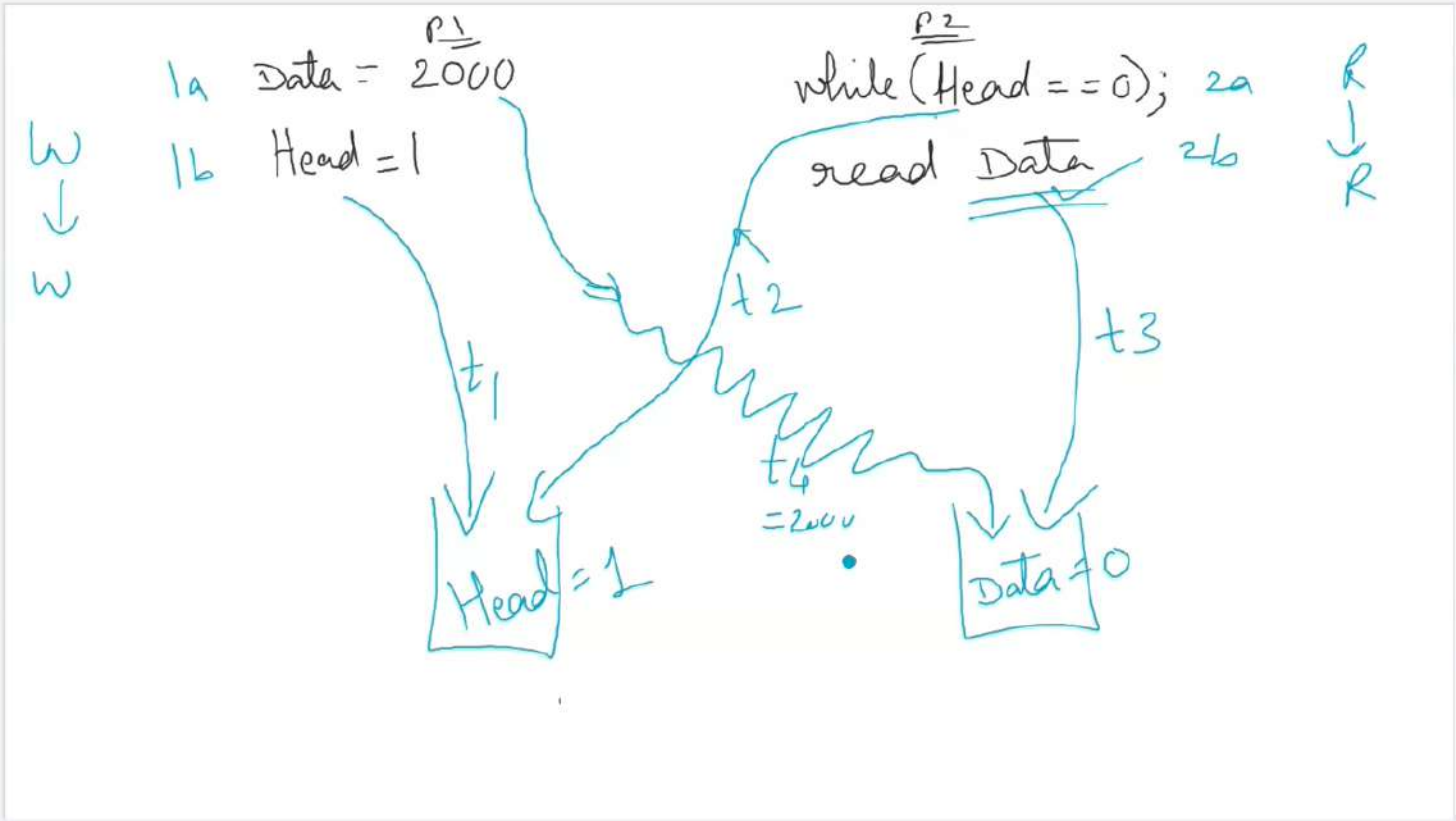
Processor issues in program-order
Optimisation-write-buffer creates problem
Result: flag1 = flag2 = 0 =? Both P1, P2 in critical section
=> violated mutual exclusion !

This optimisation is safe in uniprocessor since bypassing does not lead to violation of uniprocessor data dependence.
However such reorderings can violate SC semantics in a multiprocessor environment

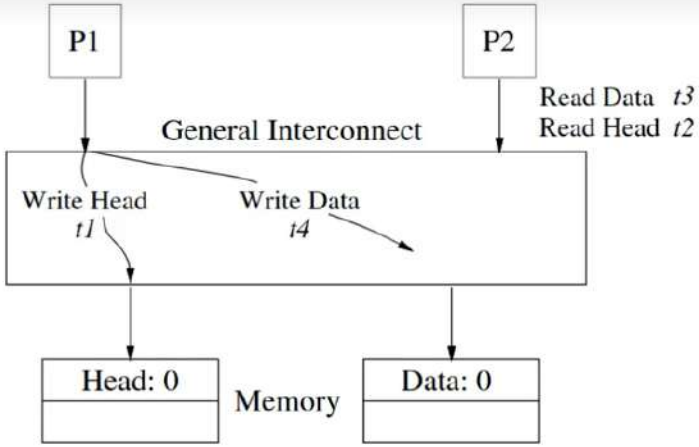
Importance of maintaining W->W

- Non-bus interconnect, Multiple memory modules, Process issues operations in program order
- Multiple writes from same process may be simultaneously serviced by different memory modules
- Ex: show how SC can be violated here

```
   P1                          P2
Data = 2000;                while(Head == 0);
Head = 1;                   read Data;
```


P1
1a Data = 2000
1b Head = 1
W
W
t1
Head = 1
P2
while (Head == 0); 2a
read Data 2b
R
R
t2
t3
t4
=2000
Data = 0

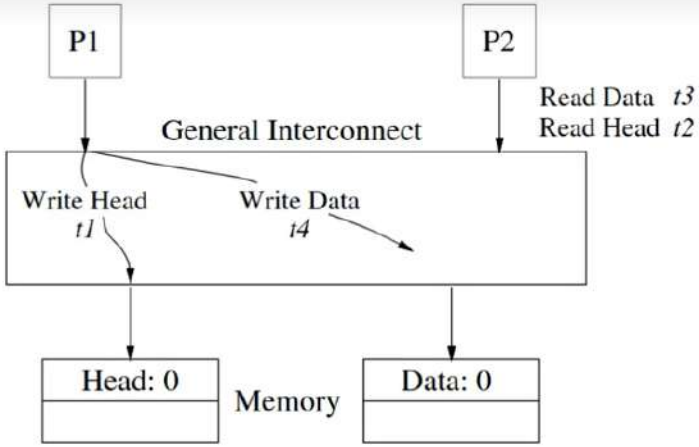
Overlapped writes



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(b) overlapped writes

Overlapped writes



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(b) overlapped writes

SC gives Data value to P2 = 2000
But network delay may cause wr-Head to complete before wr-Data
And P2 may read new-Head but old-Data

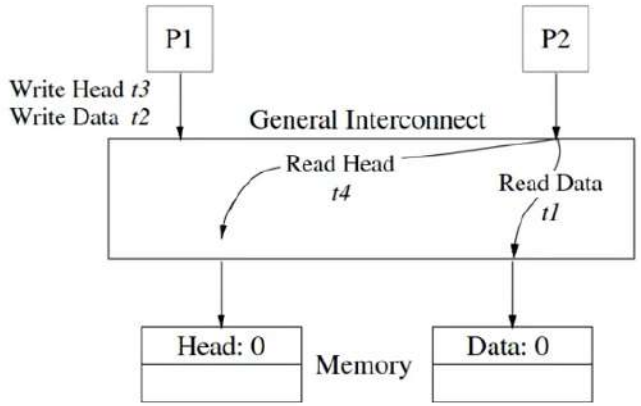
Solution: memory may need to ack P1 when wr-Data done

Importance of maintaining R->W and R -> R

- Processor optimisation of allowing non-blocking read operations:
  - Proceed past the read: lockup-free cache, speculative execution, dynamic scheduling can cause this
- Ex: same as above, P1 does writes in program order and they reach memory in program order
- But P2 allows overlapped reads

```
   P1                          P2

Data = 2000;                while(Head == 0);

Head = 1;                   read Data;
```

Non-blocking reads



P2 reads old-Data and new-Head
Violating SC

P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(c) non–blocking reads

- **Presence of Cache**
  - P1 wr-Data then wr-Head.
  - P2 has 'Data' in its cache
  - Head reaches first + modifies
  - Data is still to reach P2's cache for inv
  - Therefore P2 read new-Head from memory and old-Data from its cache
  - Solution: P1 must wait for P2 to inv Data before doing wr-Head

Solution to SC?

- A completely different way to overcome the performance limitations imposed by SC is to change the consistency model itself
- i.e. not to guarantee ordering constraints but still to retain semantics that are intuitive enough to be useful
- By relaxing the ordering constraints, these relaxed consistency models allow compiler to reorder accesses before presenting them to the hardware
- At hardware level they allow multiple memory references from the same process to be outstanding and also become visible out of program order
- This allows to overlap/hide the memory access latency
- Why does this work?
  - Because SC is overly conservative
  - Many orders that it preserves are not really needed to satisfy programmers intuition
- We will see these relaxed consistency models ... next

Relaxed memory consistency models

- For directory protocols, misses have long latency and collecting acknowledgements can take even longer
- To preserve SC we need to restrict many performance optimisations done by modern compilers and processors
- High cost of memory access motivates overlapped reads/writes or reordering
- Therefore allow reorder up to some extent still preserving semantics

  => Relaxed consistency models

Characterising different memory-consistency models

- Based on two key characteristics:
- (1) how they relax the program order requirement (i, ii, iii), AND
- (2) how they relax the write atomicity requirement (iv)
- (i) Relax W -> R
- (ii) Relax W -> W
- (iii) Relax R -> R, R -> W

Program order:

Applies to
Different locations

- (iv) Read others write early [i.e. before many others have seen inv or update message]
- (v) Relax program order as well as write atomicity = Read own write early [our own write is read by self before other sharers are inv or updated. e.g. Read from write-buffer, or wr-thru-caches (read from cache before write is complete to next level]
  - In a cache based system this allows the read to return the value of the write before the write is serialised with respect to other writes to the same location and before the inv/updates of the write reach any other processor