EE 660: Computer Architecture
Parallel Programming and Small Multiprocessors

Yao Zheng

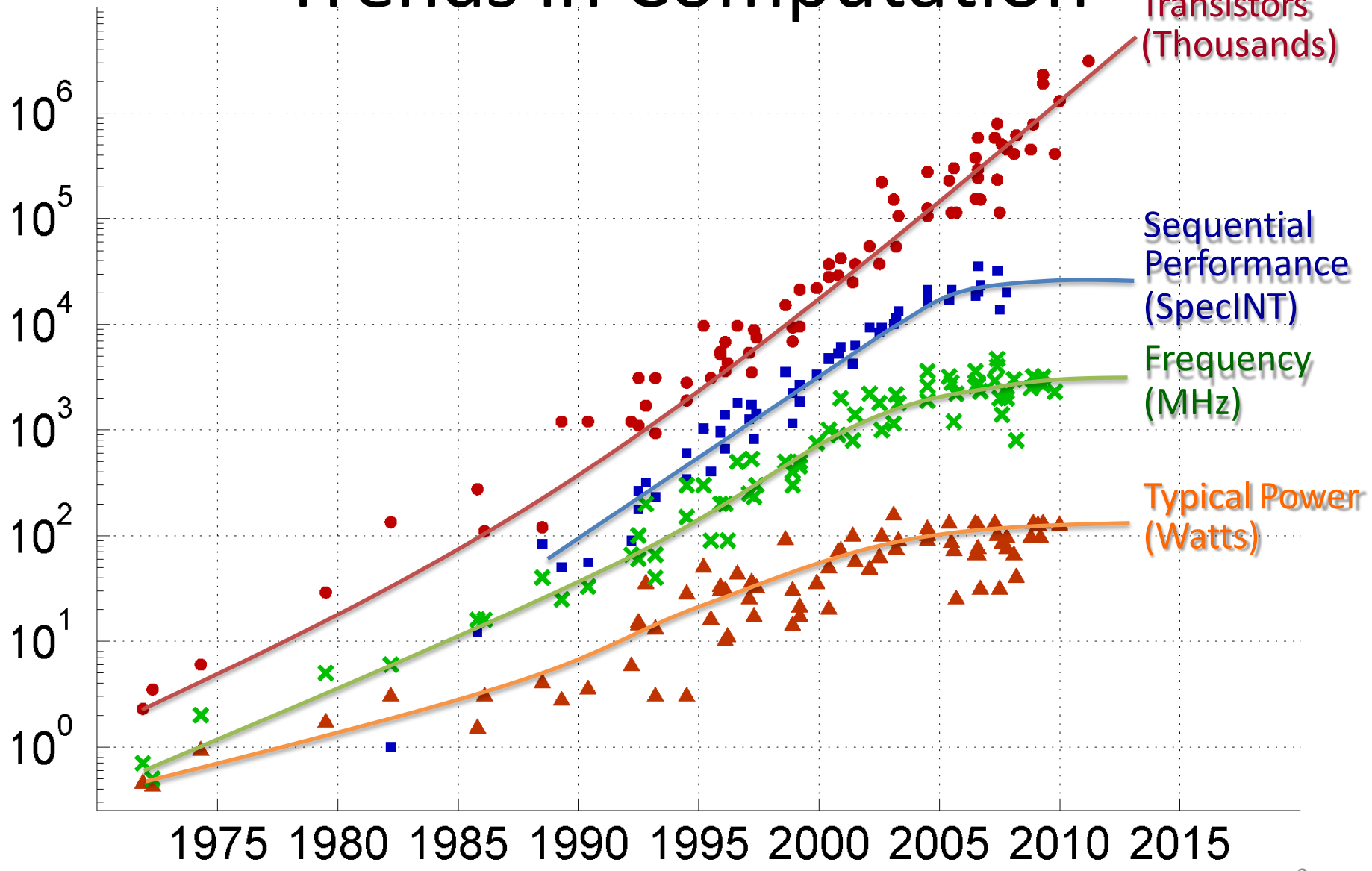
Department of Electrical Engineering

University of Hawai‘i at Mānoa

UNIVERSITY of HAWAI‘I®
MĀNOA

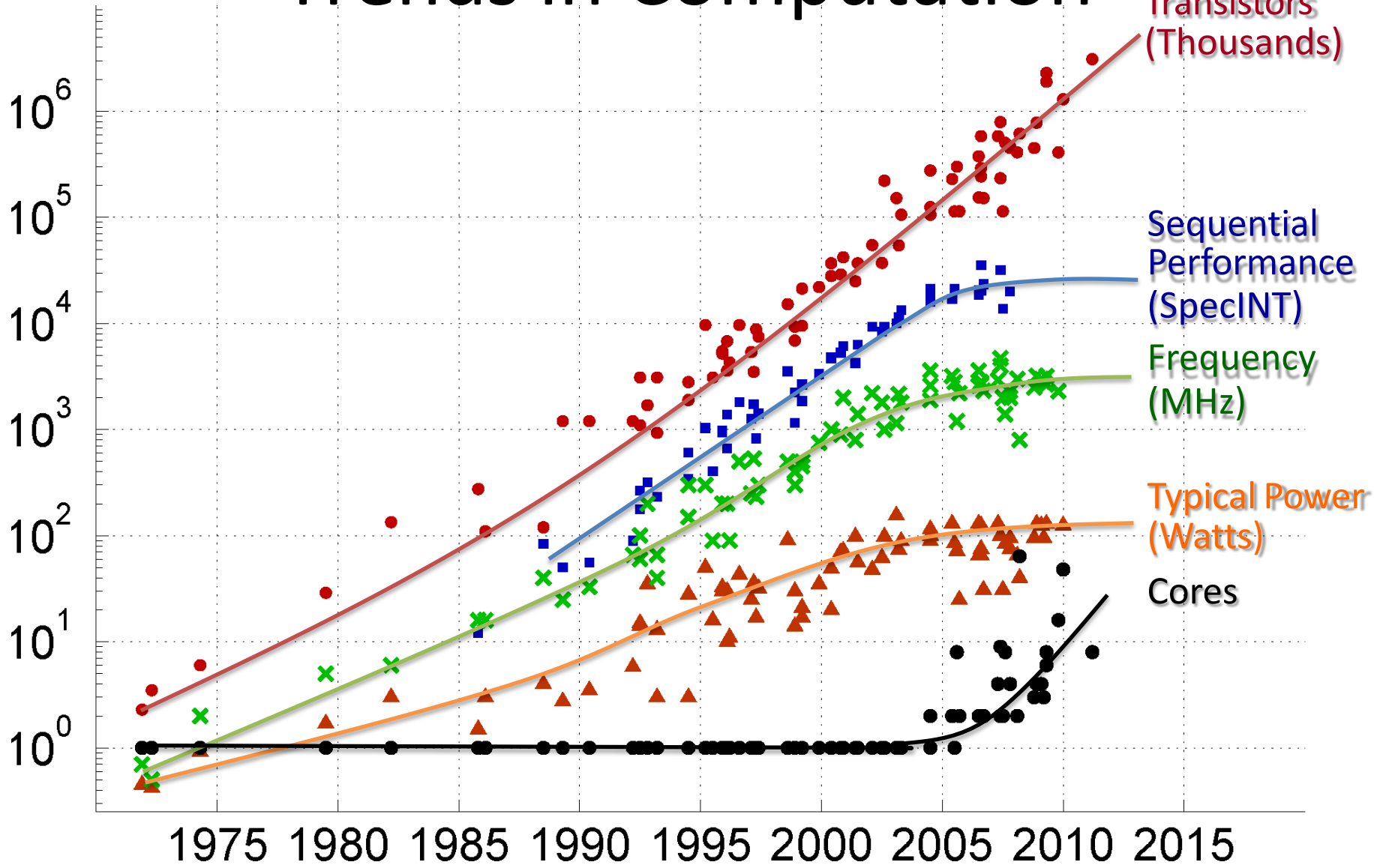
Based on the slides of Prof. David Wentzlaff

Trends in Computation



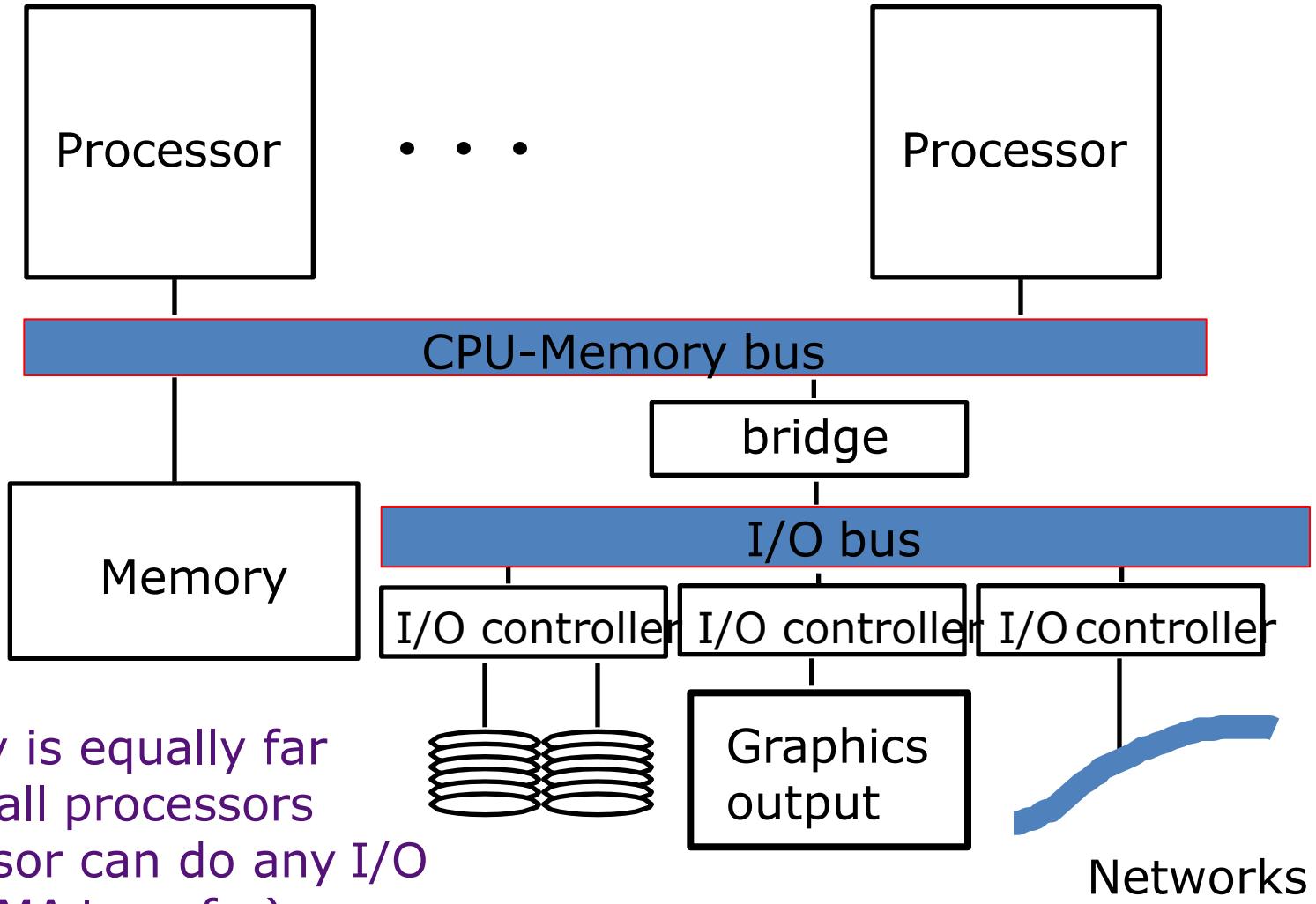
Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and D. We^{2}ntzlaff

Trends in Computation



Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and D. We^{3}ntzlaff

Symmetric Multiprocessors



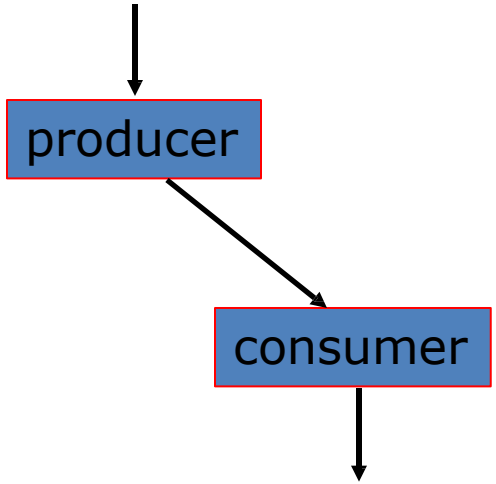
symmetric

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

Synchronization

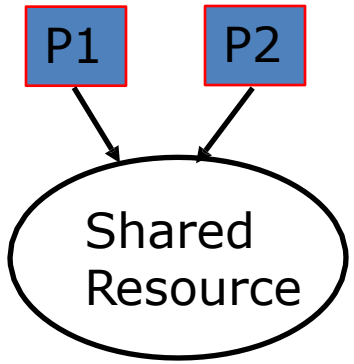
The need for synchronization arises whenever there are concurrent processes in a system

(even in a uniprocessor system)

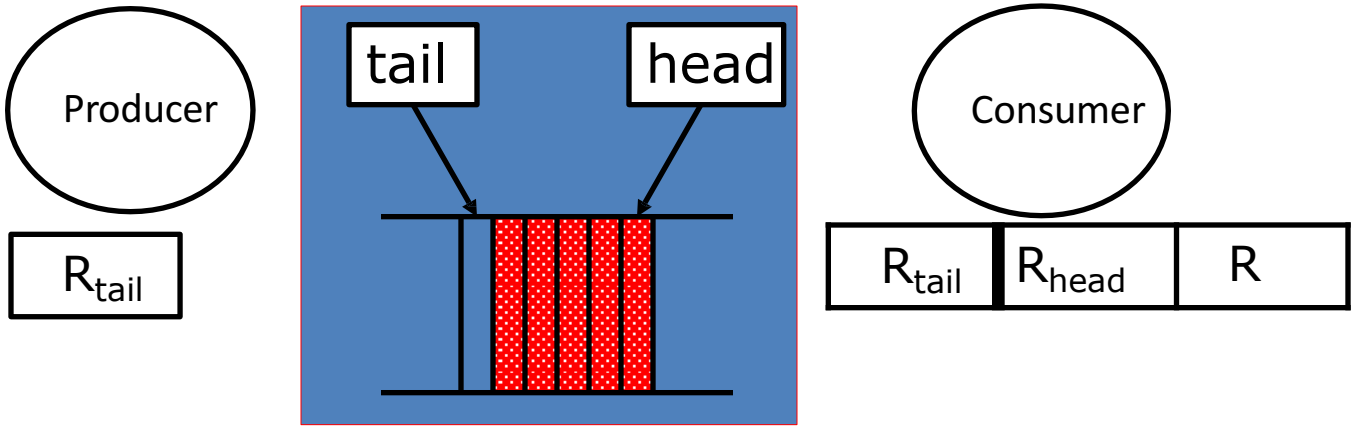


Producer-Consumer: A consumer process must wait until the producer process has produced data

Mutual Exclusion*:* Ensure that only one process uses a resource at a given time



A Producer-Consumer Example



Producer posting Item x:

```
Load R_tail, (tail)
Store x, (R_tail)
R_tail=R_tail+1
Store R_tail, (tail)
```

Consumer:

```
        Load R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head=R_head+1
        Store R_head, (head)
        process(R)
```

The program is written assuming instructions are executed in order.

Problems?

A Producer-Consumer Example

continued

Producer posting Item x:

Load R_{tail}, (tail)

1 Store x, (R_{tail})

R_{tail}=R_{tail}+1

2 Store R_{tail}, (tail)

Consumer:

Load R_{head}, (head)

Load R_{tail}, (tail) *3*

if R_{head}==R_{tail} goto spin

Load R, (R_{head}) *4*

R_{head}=R_{head}+1

Store R_{head}, (head)

process(R)

Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

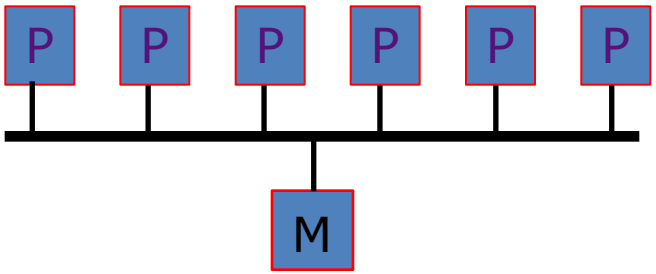
Problem sequences are:

2, 3, 4, 1

4, 1, 2, 3

Sequential Consistency

A Memory Model



" A system 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 the order specified by the program"

Leslie Lamport

Sequential Consistency =

arbitrary *order-preserving interleaving*
of memory references of sequential programs

Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:
Store 1, (X) *(X = 1)*
Store 11, (Y) *(Y = 11)*

T2:
Load R_1, (Y)
Store R_1, (Y') *(Y'= Y)*
Load R_2, (X)
Store R_2, (X') *(X'= X)*

what are the legitimate answers for X' and Y' ?

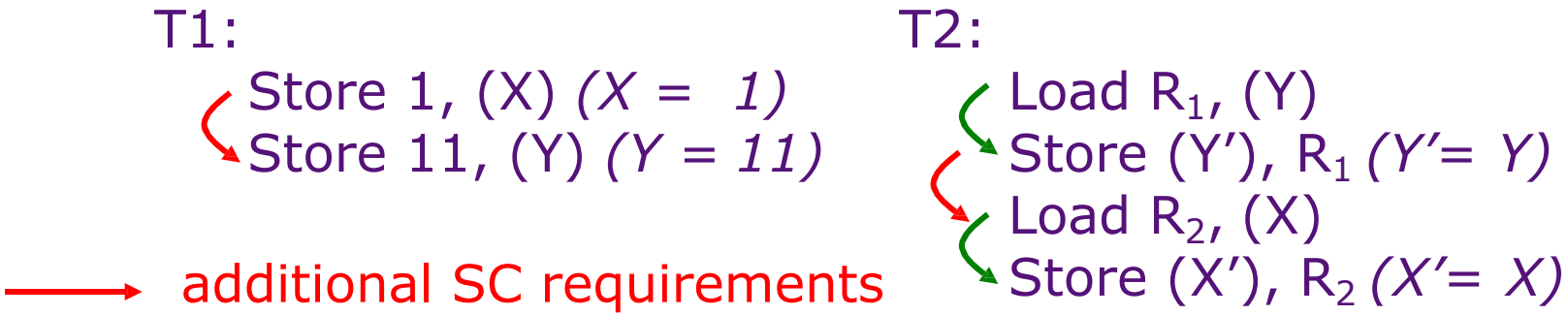
(X',Y') ε{(1,11), (0,10), (1,10), (0,11)} ?

If Y is 11 then X cannot be 0

Sequential Consistency

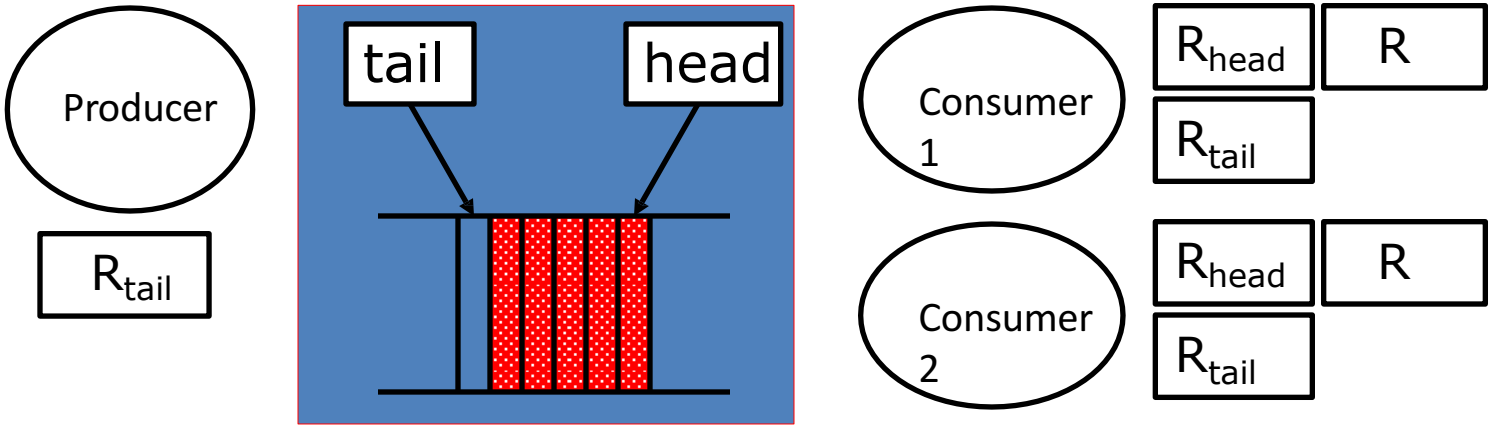
Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (→)

What are these in our example ?



Does (can) a system with caches or out-of-order execution capability provide a *sequentially consistent* view of the memory ?

Multiple Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)
Store x, (R_{tail})
R_{tail}=R_{tail}+1
Store R_{tail} ,(tail)

Consumer:

spin: Load R_{head}, (head)
Load R_{tail}, (tail)
if R_{head}==R_{tail} goto spin
Load R, (R_{head})
R_{head}=R_{head}+1
Store R_{head} , (head)
process(R)

Critical section:
Needs to be executed atomically by one consumer ⇒ locks

What is wrong with this code?

Locks or Semaphores

E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

P(s): *if s>0, decrement s by 1, otherwise wait*

probeer te verlagen, literally ("try to reduce")

V(s): *increment s by 1 and wake up one of the waiting processes*

verhogen ("increase")

P's and V's must be executed atomically, i.e., without

- *interruptions* or
- *interleaved accesses to s* by other processors

```
Process i
    P(s)
        <critical section>
    V(s)
```

initial value of s determines the maximum no. of processes in the critical section

Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

atomic read-modify-write instructions

Examples: *m is a memory location, R is a register*

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
        M[m] ← 1;
```

```
Fetch&Add (m), RV, R:
    R ← M[m];
    M[m] ← R + RV;
```

```
Swap (m), R:
    Rt ← M[m];
    M[m] ← R;
    R ← Rt;
```

Multiple Consumers Example

using the Test&Set Instruction

```
P:      Test&Set (mutex),Rtemp
        if (Rtemp!=0) goto P
        Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rhead=Rhead+1
        Store Rhead, (head)
V:      Store 0, (mutex)
        process(R)
```

Critical Section

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P's and V's

What if the process stops or is swapped out while in the critical section?

Nonblocking Synchronization

```
Compare&Swap(m), Rt, Rs:
  if (Rt==M[m])
      then  M[m]=Rs;
            Rs=Rt ;
            status ← success;
      else  status ← fail;
```

status is an *implicit argument*

```
try:
spin:  Load Rhead, (head)
       Load Rtail, (tail)
       if Rhead==Rtail goto spin
       Load R, (Rhead)
       Rnewhead = Rhead+1
       Compare&Swap(head), Rhead, Rnewhead
       if (status==fail) goto try
       process(R)
```

Load-link & Store-conditional

aka Load-reserve, Load-Locked

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-link R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs' reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-link R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head = R_head + 1
        Store-conditional R_head, (head)
        if (status==fail) goto try
        process(R)
```

Performance of Locks

Blocking atomic read-modify-write instructions

e.g., Test&Set, Fetch&Add, Swap

vs

Non-blocking atomic read-modify-write instructions

e.g., Compare&Swap,

Load-link/Store-conditional

vs

Protocols based on ordinary Loads and Stores

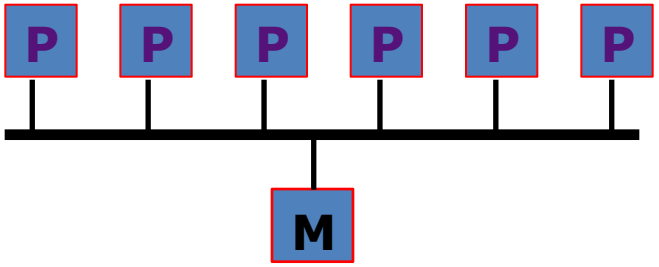
Performance depends on several interacting factors:

degree of contention,

caches,

out-of-order execution of Loads and Stores

Issues in Implementing Sequential Consistency



Implementation of SC is complicated by two issues

- *Out-of-order execution capability*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Caches*

  Caches can prevent the effect of a store from being seen by other processors

SC complications motivate architects to consider *weak* or *relaxed* memory models

Memory Fences

Instructions to sequentialize memory accesses

Processors with *relaxed or weak memory models* permit Loads and Stores to different addresses to be reordered, remove some/all extra dependencies imposed by SC

- LL, LS, SL, SS

Need to provide *memory fence* instructions to force the serialization of memory accesses

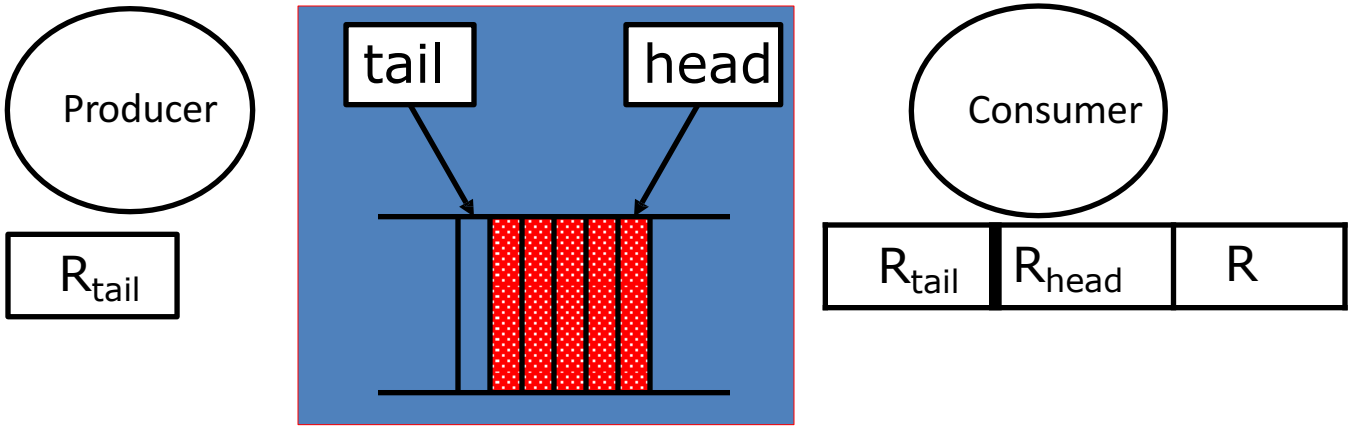
Examples of relaxed memory models:

- Total Store Order: LL, LS, SS, enforce SL with fence
- Partial Store Order: LL, LS, enforce SL, SS with fences
- Weak Ordering: enforce LL, LS, SL, SS with fences

Memory fences are expensive operations – mem instructions wait for all relevant instructions in-flight to complete (including stores to retire – need store acks)

However, cost of serialization only when it is required!

Using Memory Fences



Producer posting Item x:

```
Load R_tail, (tail)
Store x, (R_tail)
MFence_SS
R_tail=R_tail+1
Store R_tail, (tail)
```

ensures that tail ptr is not updated before x has been stored

Consumer:

```
        Load R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        MFence_LL
        Load R, (R_head)
        R_head=R_head+1
        Store R_head, (head)
        process(R)
```

ensures that R is not loaded before equality check

Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2.
Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
    ...
    c1=1;
L:  if c2==1 then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
L:  if c1==1 then go to L
      < critical section>
    c2=0;
```

What is wrong? *Deadlock!*

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

Process 1

```
    ...
L:  c1=1;
    if c2==1 then
        { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
    ...
L:  c2=1;
    if c1==1 then
        { c2=0; go to L}
      < critical section>
    c2=0
```

- Deadlock is not possible but with a low probability a *livelock* may occur.

- An unlucky process may never get to enter the critical section $\Rightarrow$ *starvation*

A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn.
Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2==1 && turn==1
                then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1==1 && turn==2
                then go to L
      < critical section>
    c2=0;
```

- turn == *i* ensures that only process *i* can wait
- variables c1 and c2 ensure *mutual exclusion*
  Solution for n processes was given by Dijkstra and is quite tricky!

N-process Mutual Exclusion

Lamport's Bakery Algorithm

Process i

Initially num[j] = 0, for all j

Entry Code

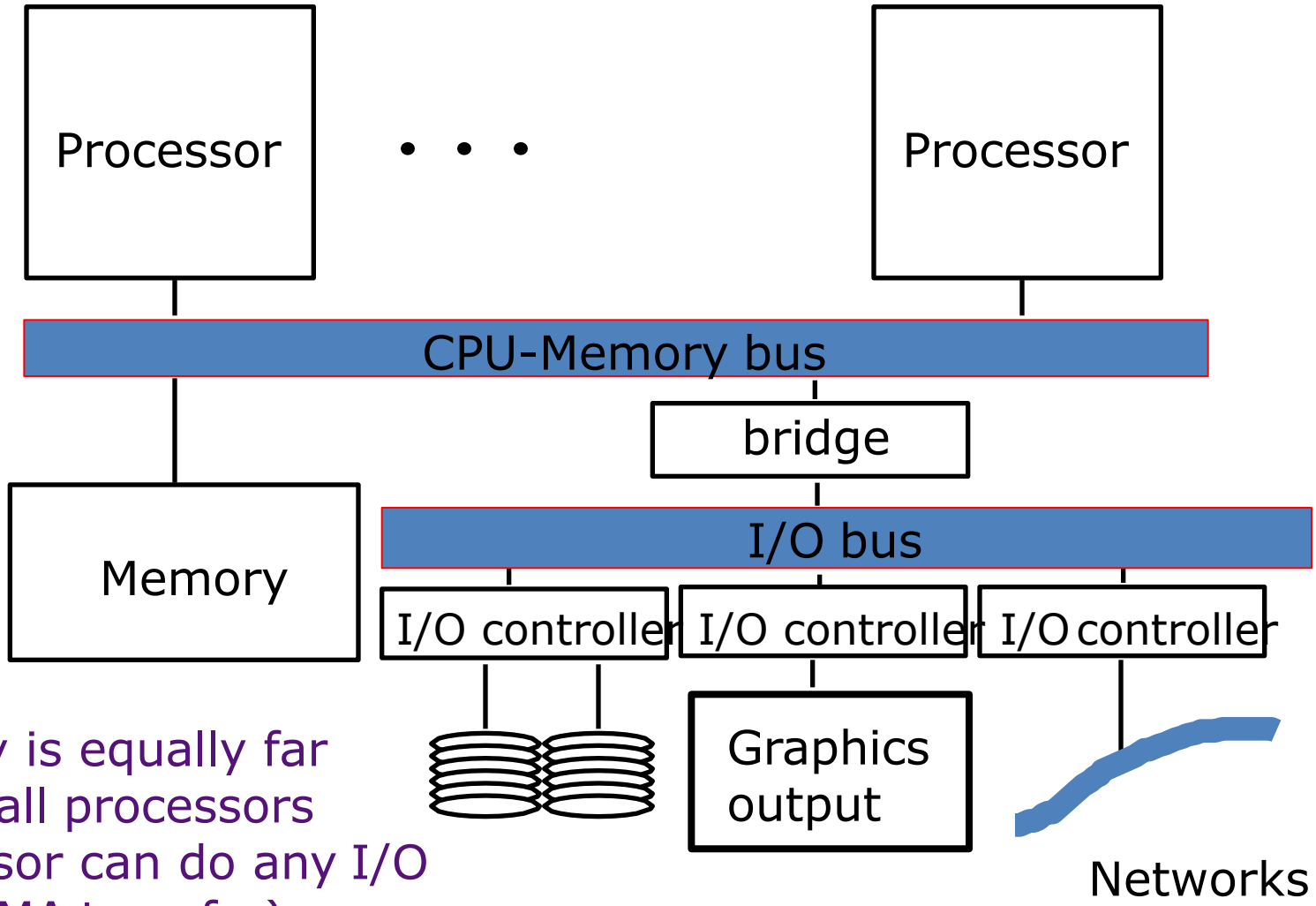
```
choosing[i] = 1;
num[i] = max(num[0], …, num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++) {
    while( choosing[j] );
    while( num[j] &&
           ( ( num[j] < num[i] ) ||
             ( num[j] == num[i] &&  j < i ) ) );
}
```

Exit Code

```
num[i] = 0;
```

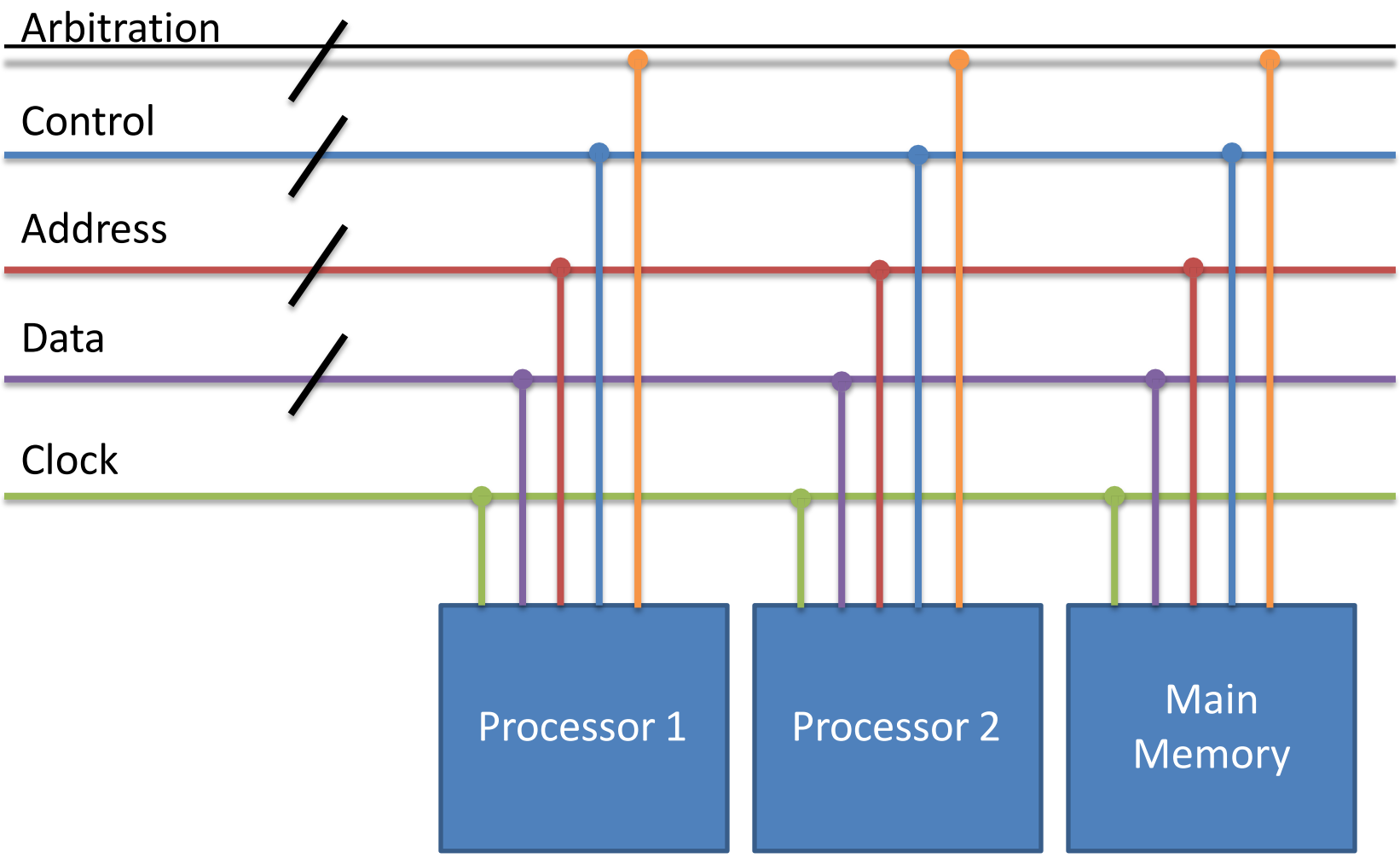
Symmetric Multiprocessors



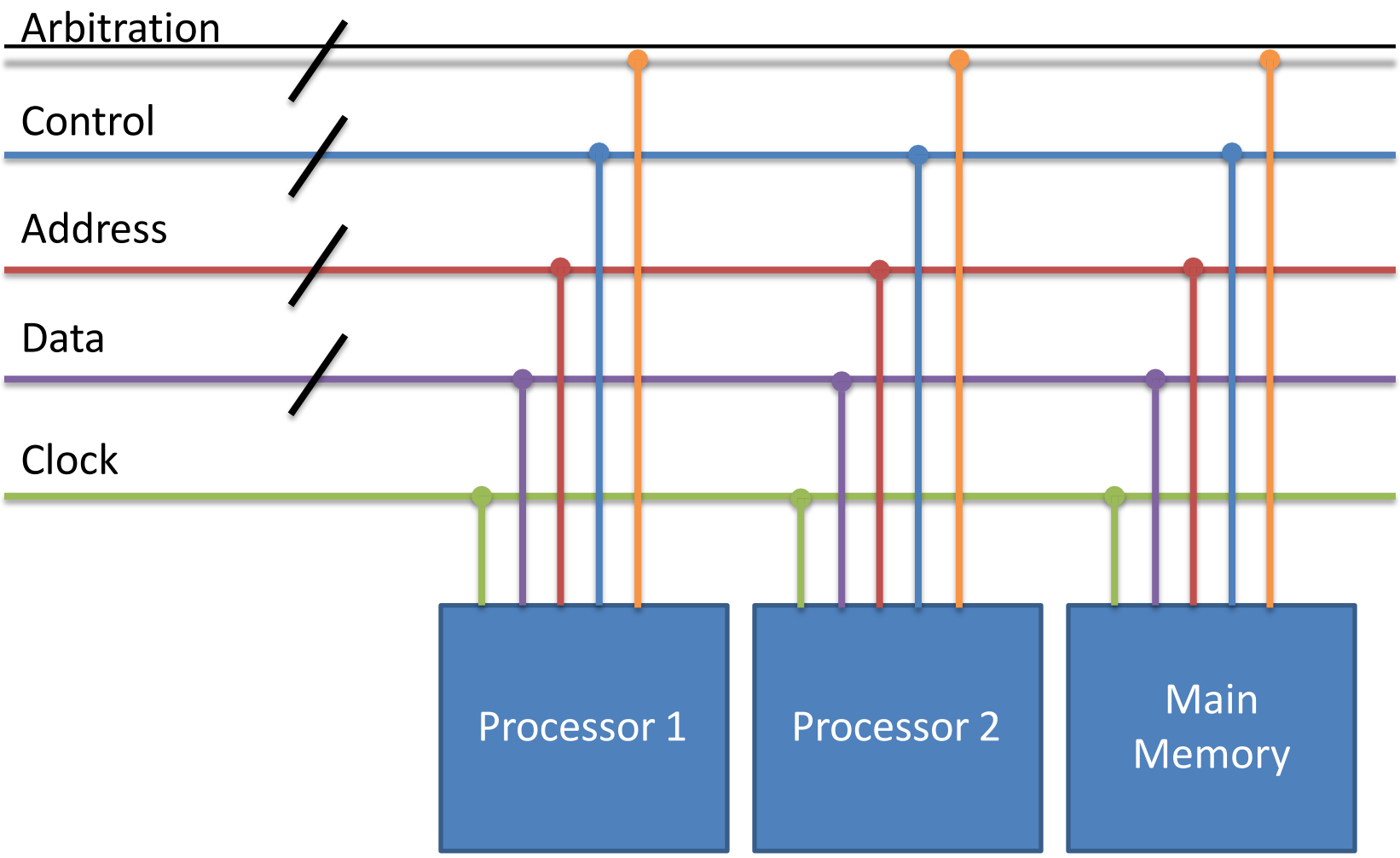
symmetric

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

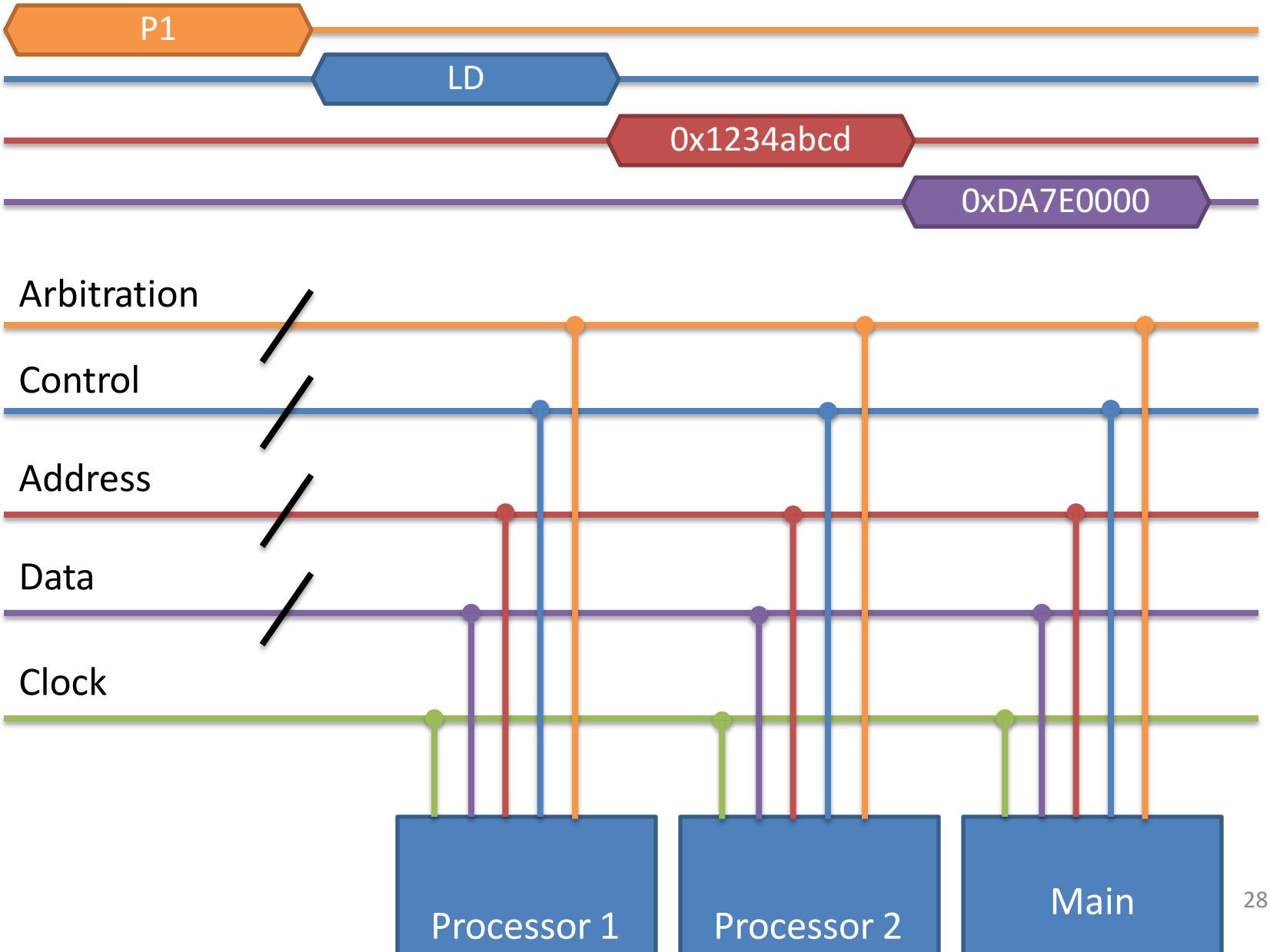
Multidrop Memory Bus



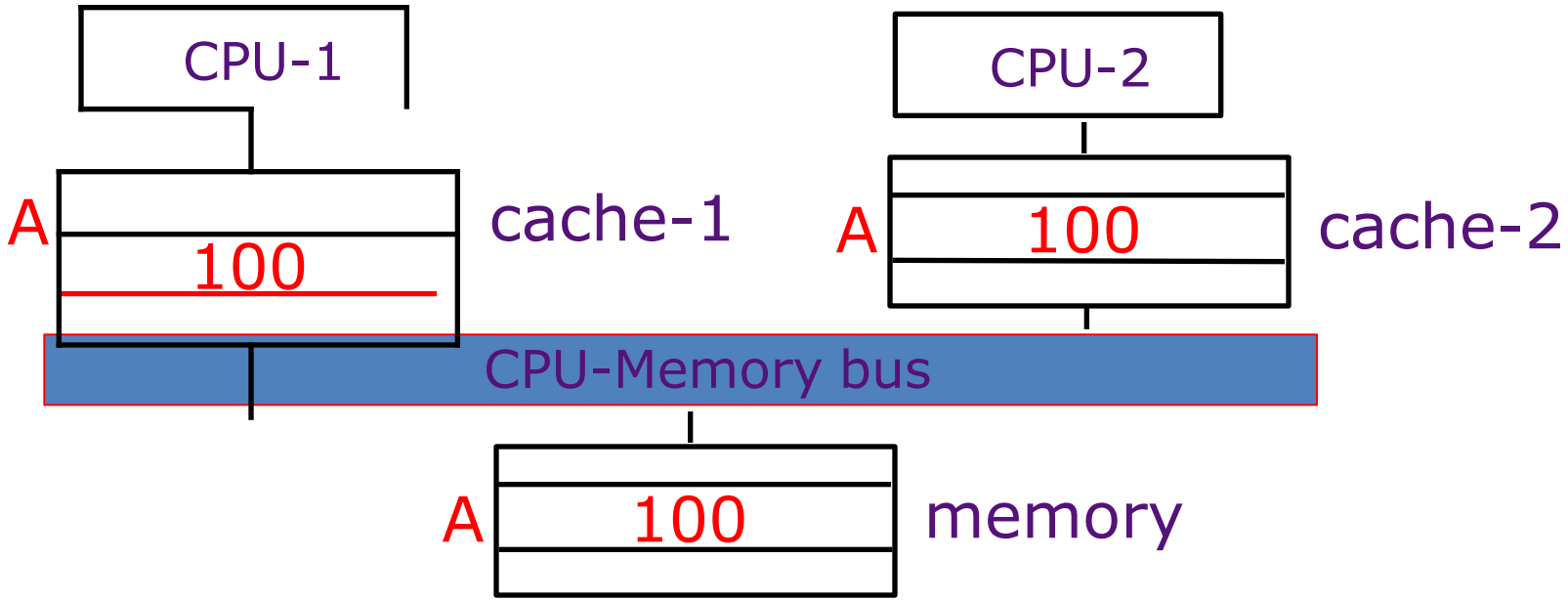
Pipelined Memory Bus



Pipelined Memory Bus



Memory Coherence in SMPs



Suppose CPU-1 updates A to 200.

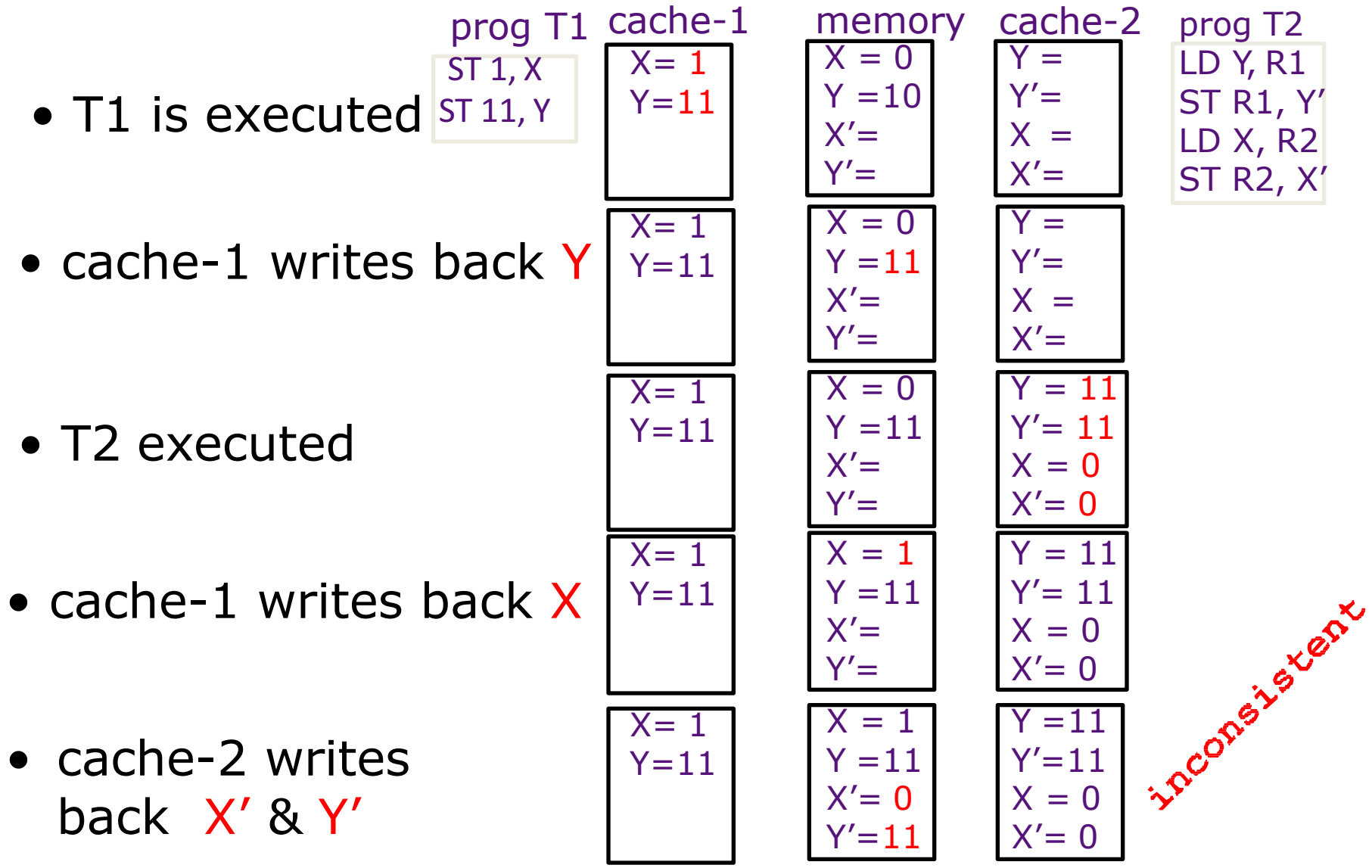
write-back: memory and cache-2 have stale values

write-through: cache-2 has a stale value

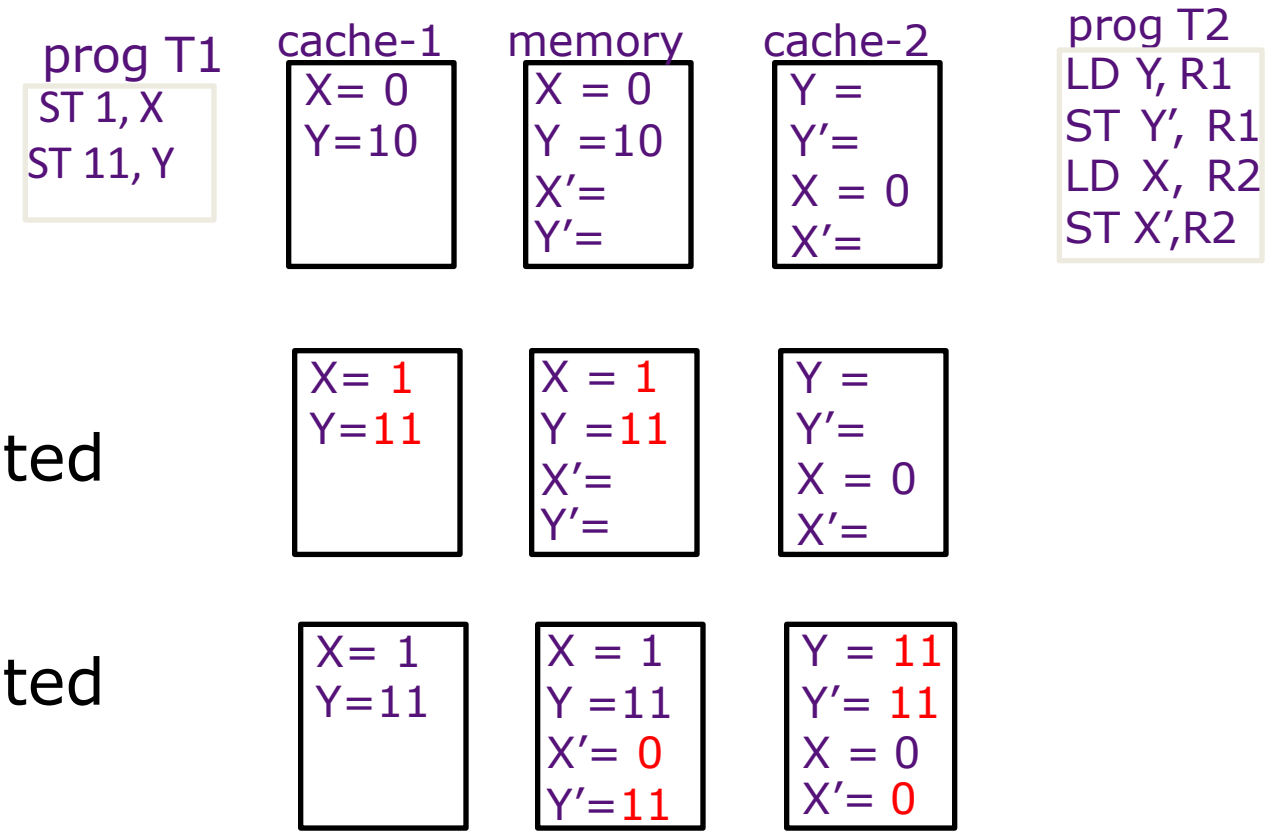
Do these stale values matter?

What is the view of shared memory for programming?

Write-back Caches & SC



Write-through Caches & SC



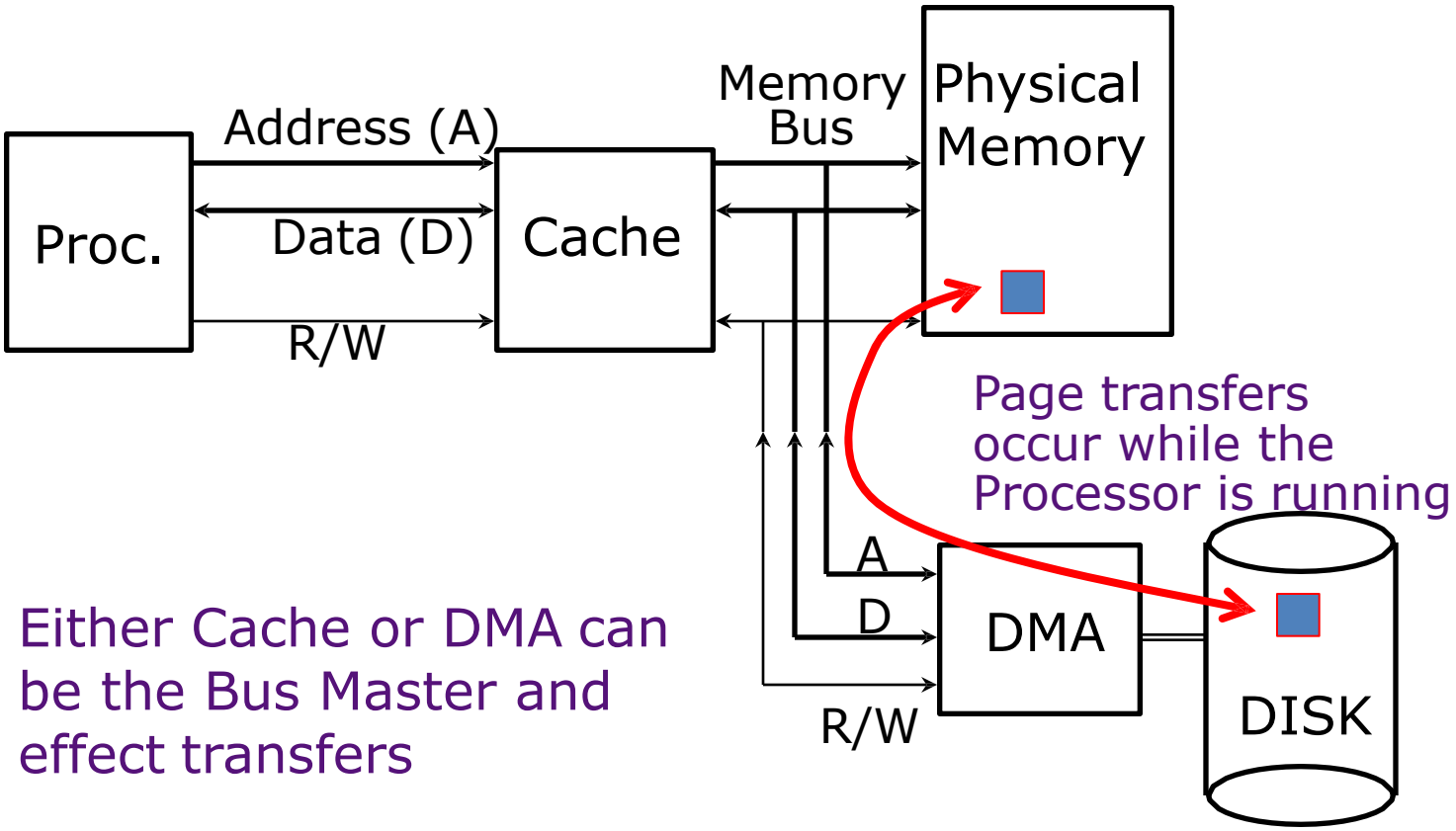
- T1 executed
- T2 executed

Write-through caches don't preserve sequential consistency either

Cache Coherence vs. Memory Consistency

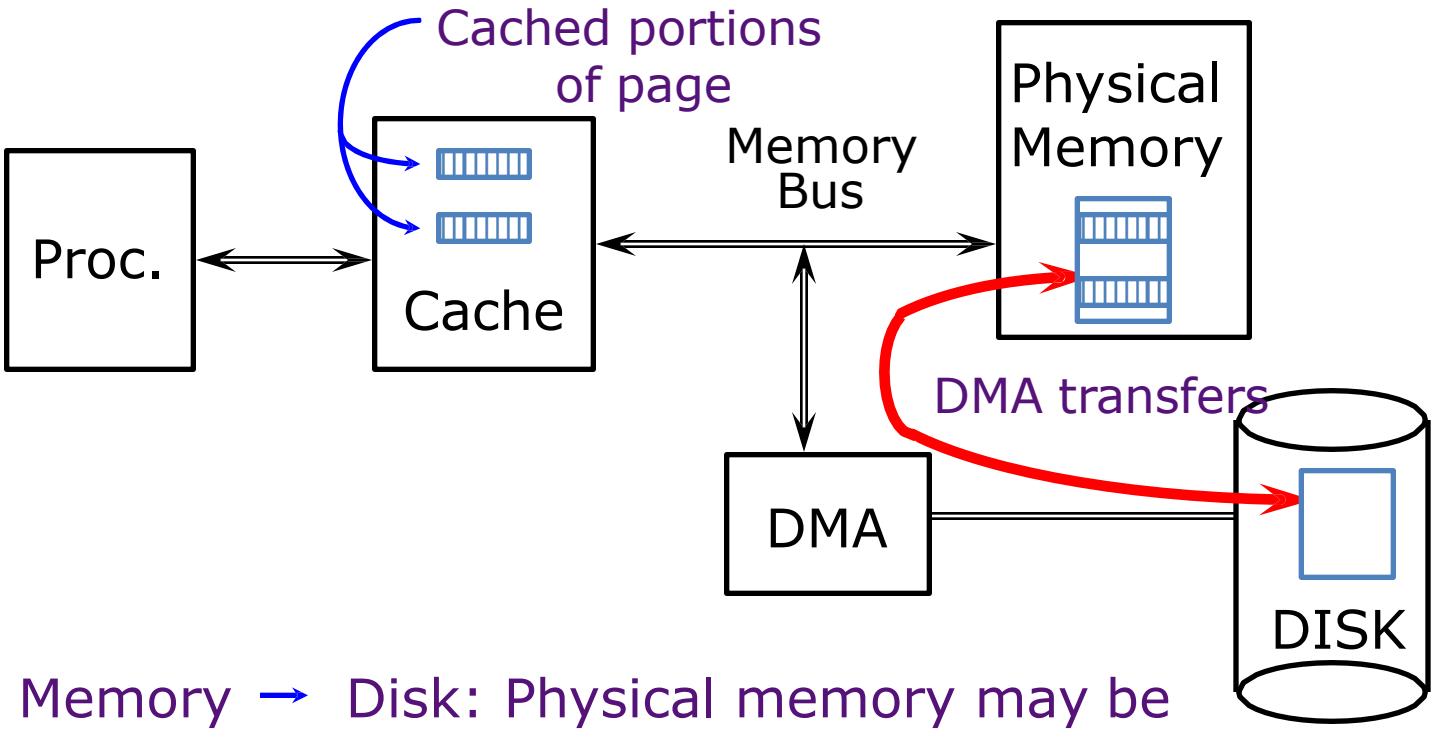
- A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors, for one memory address
  - i.e., updates are not lost
- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another, across different addresses
  - Equivalently, what values can be seen by a load
- A cache coherence protocol is not enough to ensure sequential consistency
  - But if sequentially consistent, then caches must be coherent
- Combination of cache coherence protocol plus processor memory reorder buffer implements a given machine's memory consistency model

Warmup: Parallel I/O



(DMA stands for "Direct Memory Access", means the I/O device can read/write memory autonomous from the CPU)

Problems with Parallel I/O

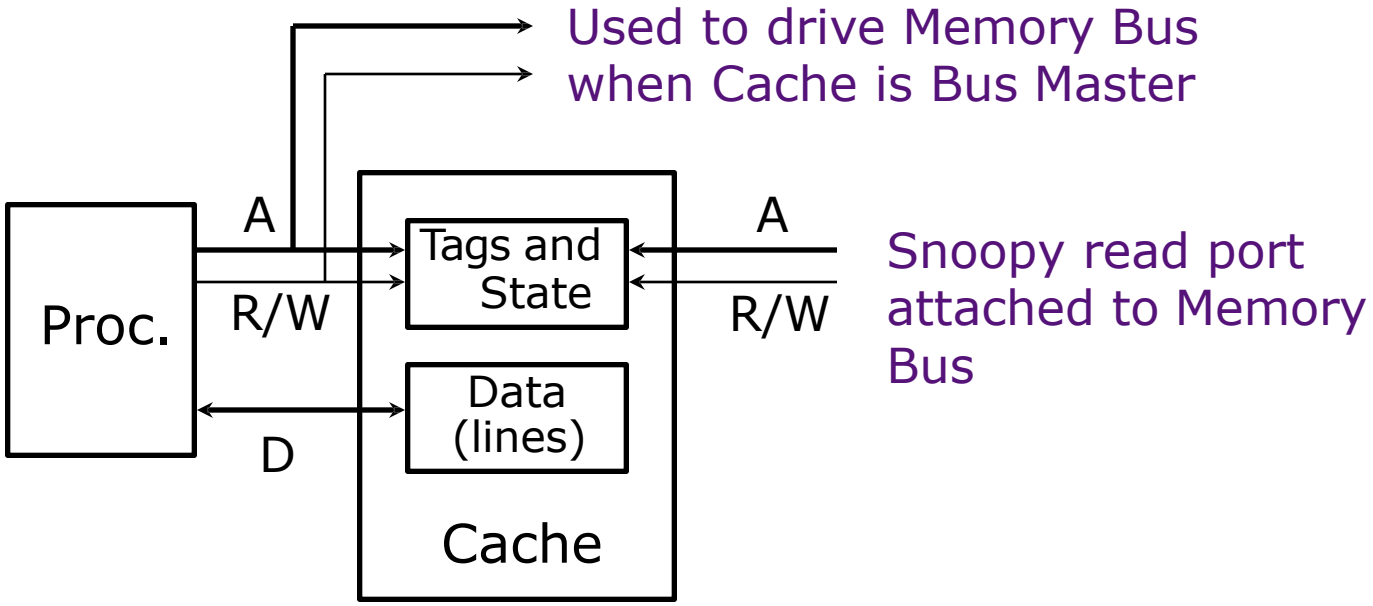


Memory → Disk: Physical memory may be stale if cache copy is dirty

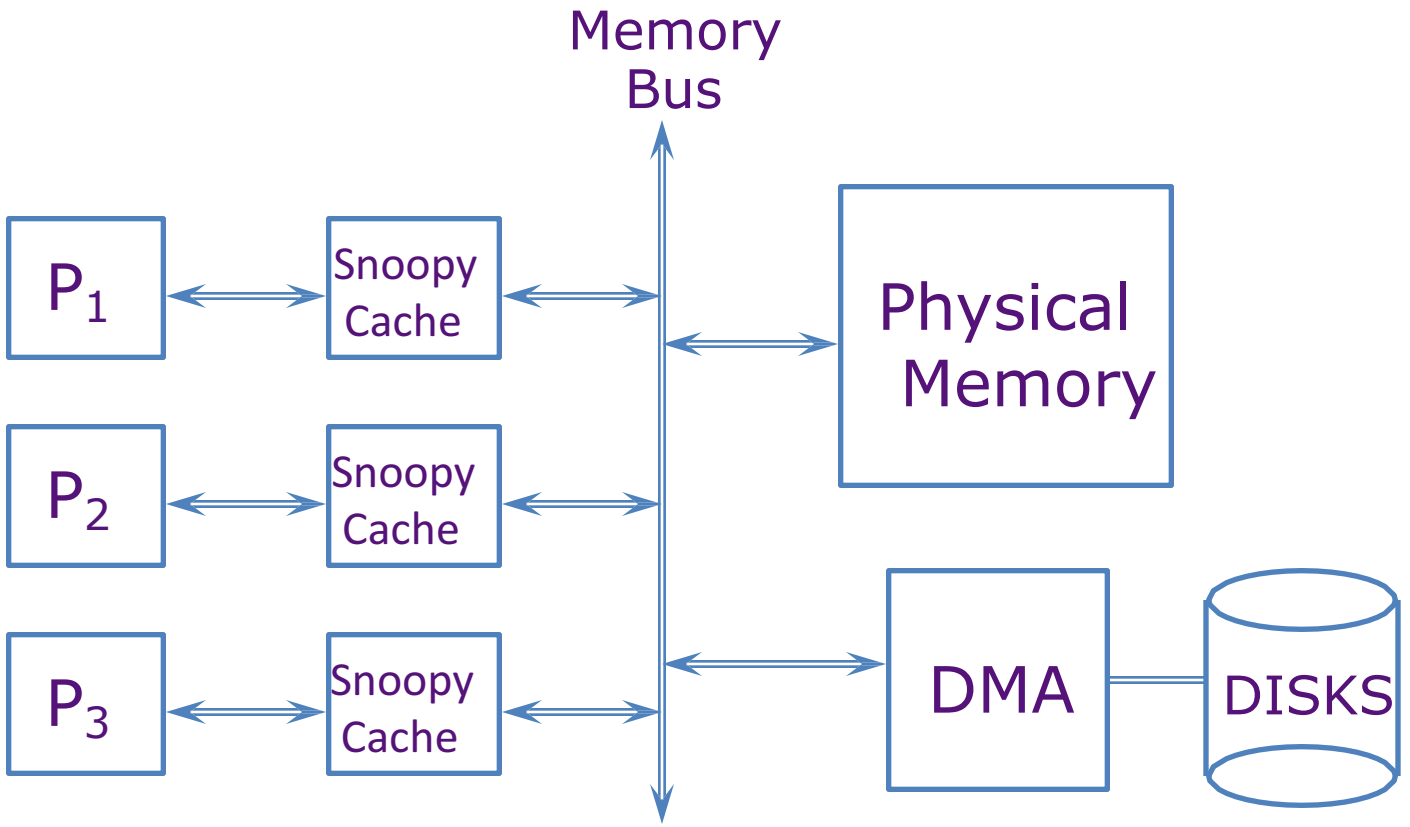
Disk → Memory: Cache may hold stale data and not see memory writes

Snoopy Cache *Goodman & Ravishankar 1983*

- Idea: Have cache watch (or snoop upon) DMA transfers, and then "do the right thing"
- Snoopy cache tags are dual-ported



Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Update(Broadcast) vs. Invalidate Snoopy Cache Coherence Protocols

- Write Update (Broadcast)
  - Writes are broadcast and update all other cache copies
- Write Invalidate
  - Writes invalidate all other cache copies

Write Update (Broadcast) Protocols

write miss:

Broadcast on bus, other processors update copies (in place)

read miss:

Memory is always up to date

Write Invalidate Protocols

write miss:

the address is *invalidated* in all other caches *before* the write is performed

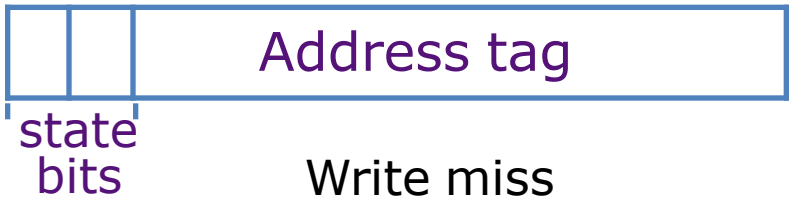
read miss:

if a dirty copy is found in some cache, a write-back is performed before the memory is read

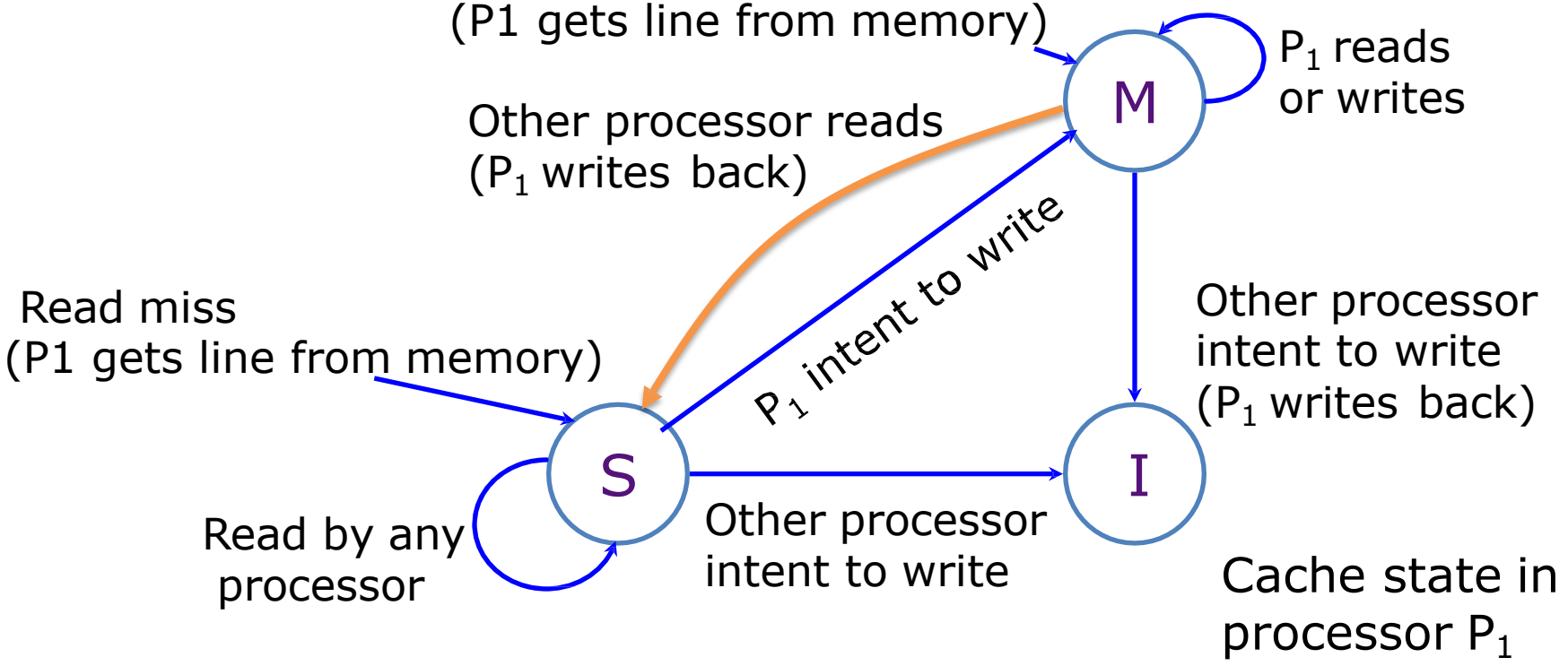
Cache State Transition Diagram

The MSI protocol

Each cache line has state bits



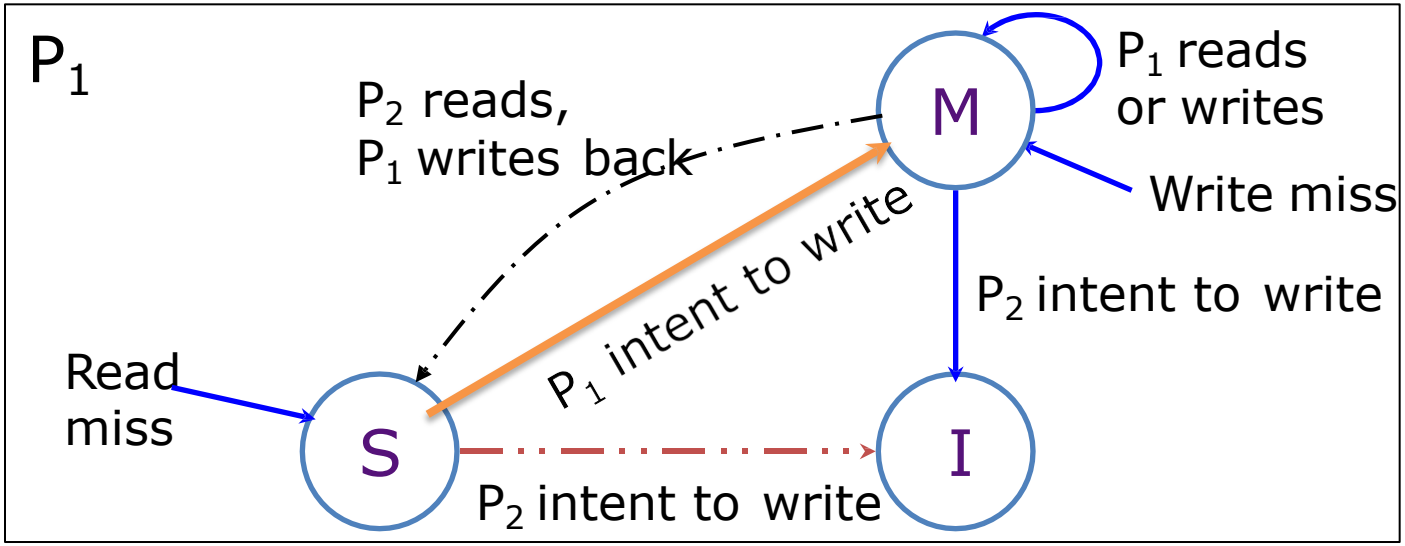
M: Modified
S: Shared
I: Invalid

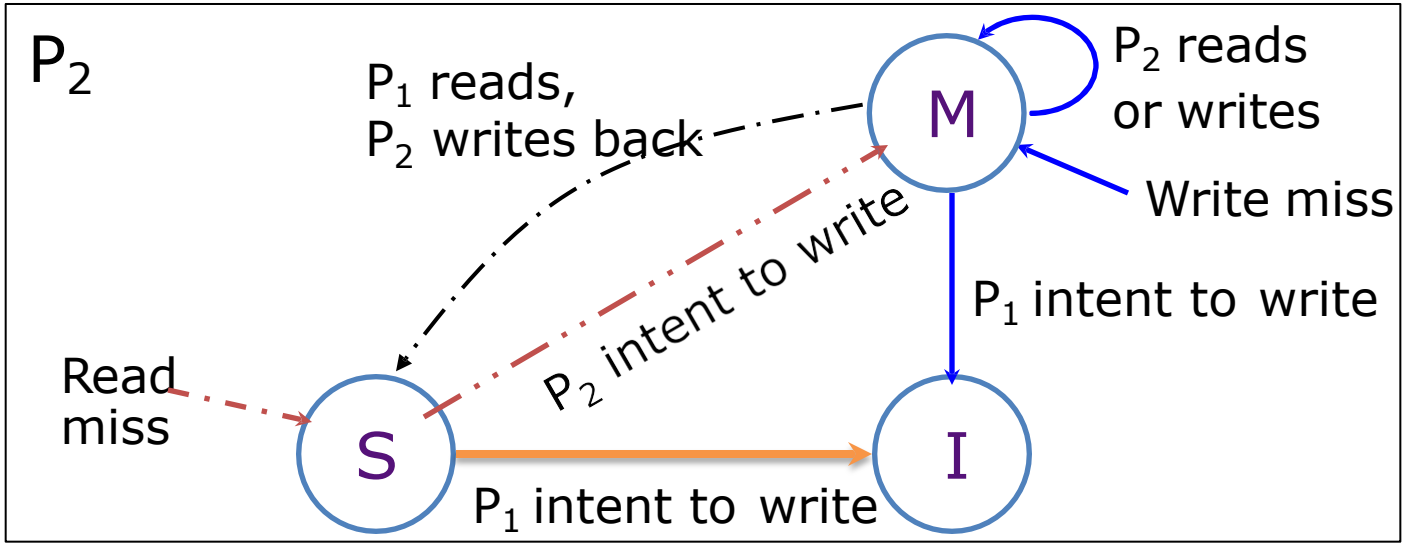


Two Processor Example

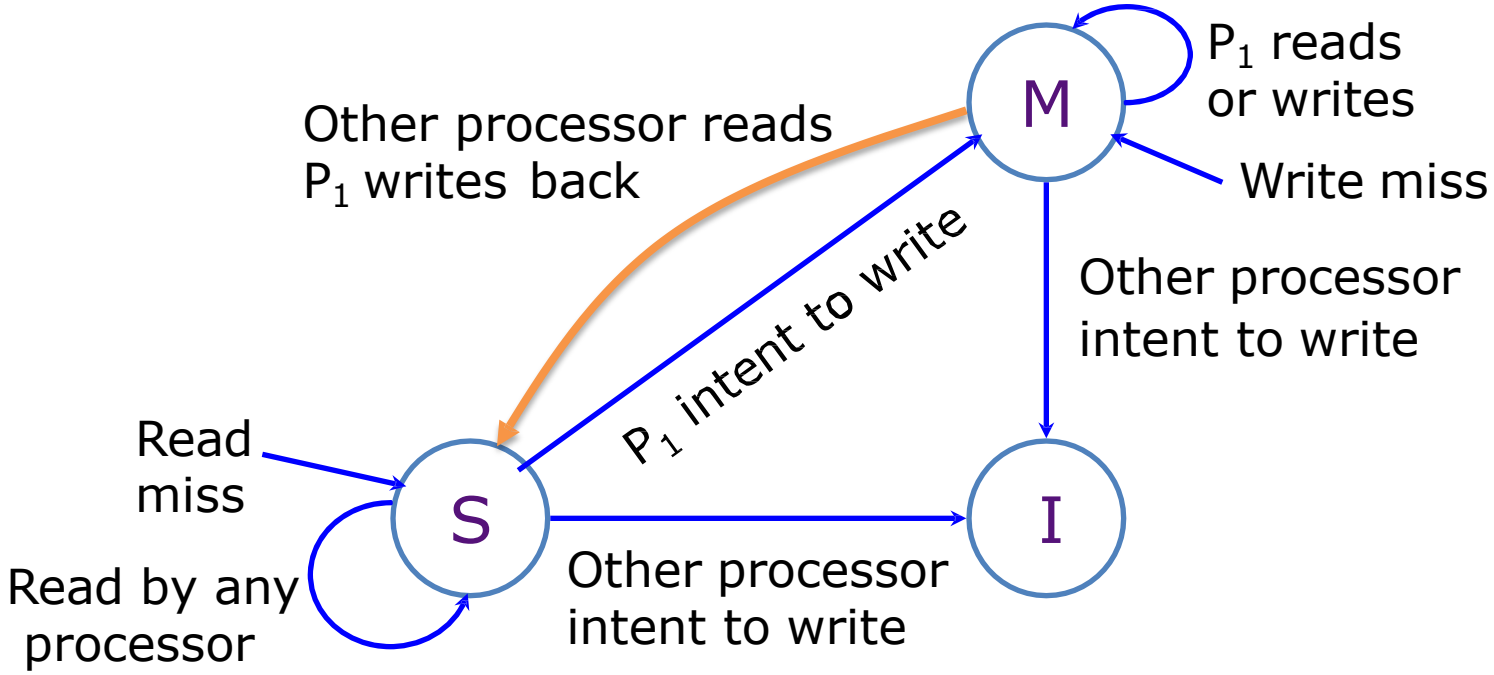
(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes





Observation

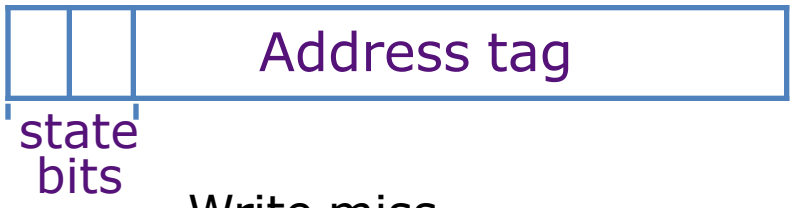


- If a line is in the M state then no other cache can have a copy of the line!
  - Memory stays coherent, multiple differing copies cannot exist

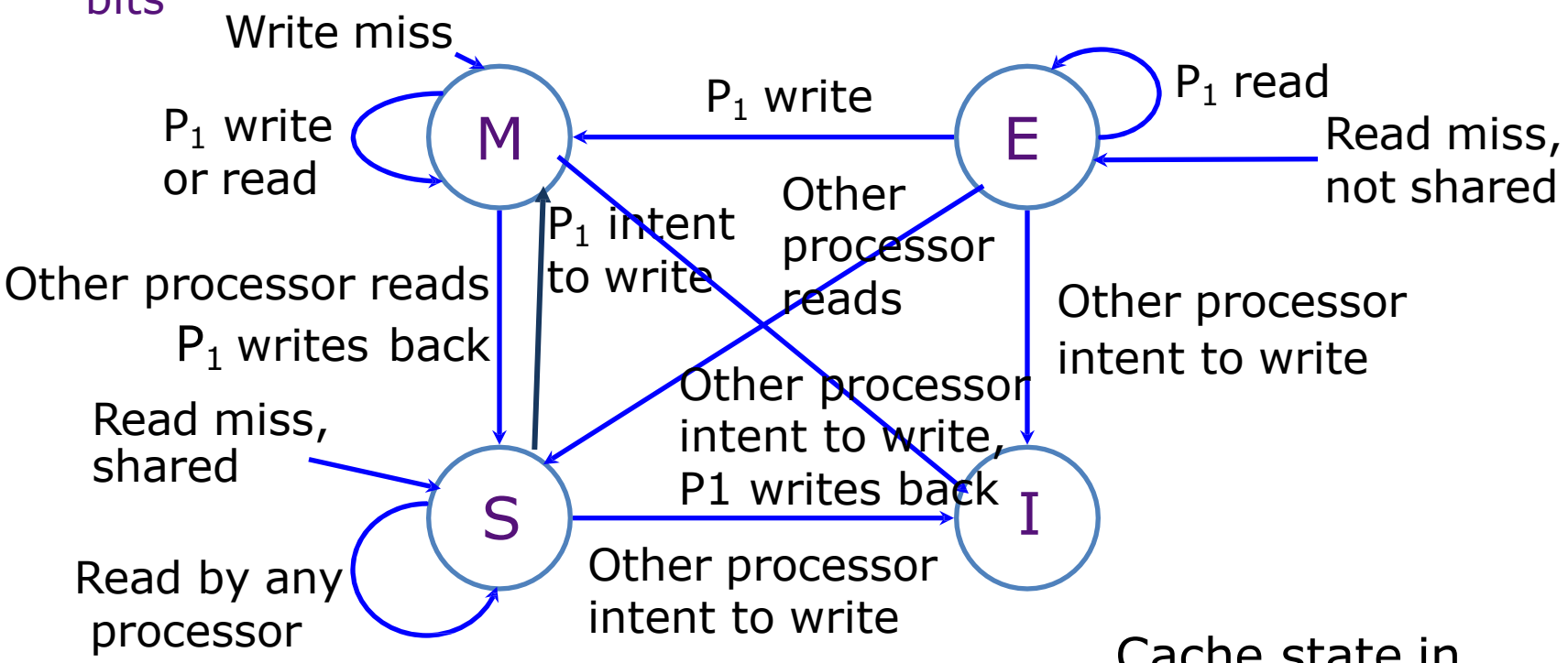
MESI: An Enhanced MSI protocol

increased performance for private data (Illinois Protocol)

Each cache line has a tag



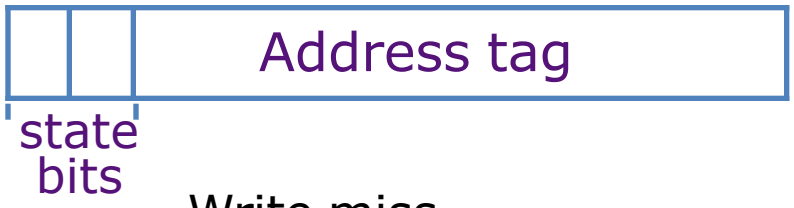
M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



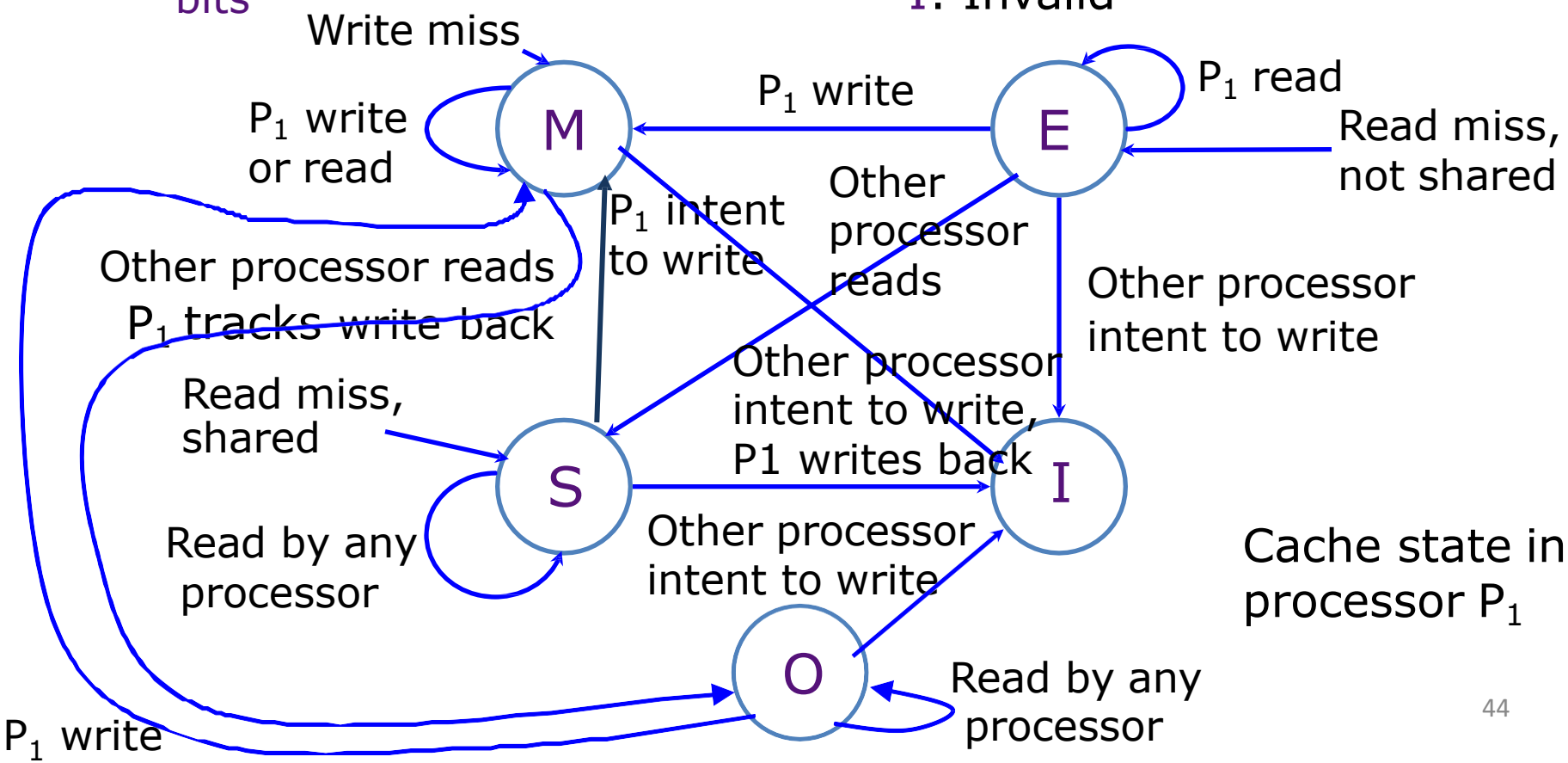
Cache state in processor P_1

MOESI (Used in AMD Opteron)

Each cache line has a tag

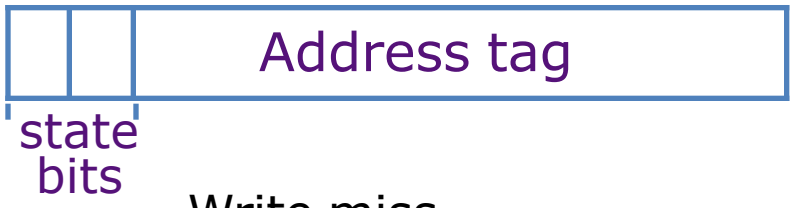


M: Modified Exclusive
O: Owned
E: Exclusive but unmodified
S: Shared
I: Invalid

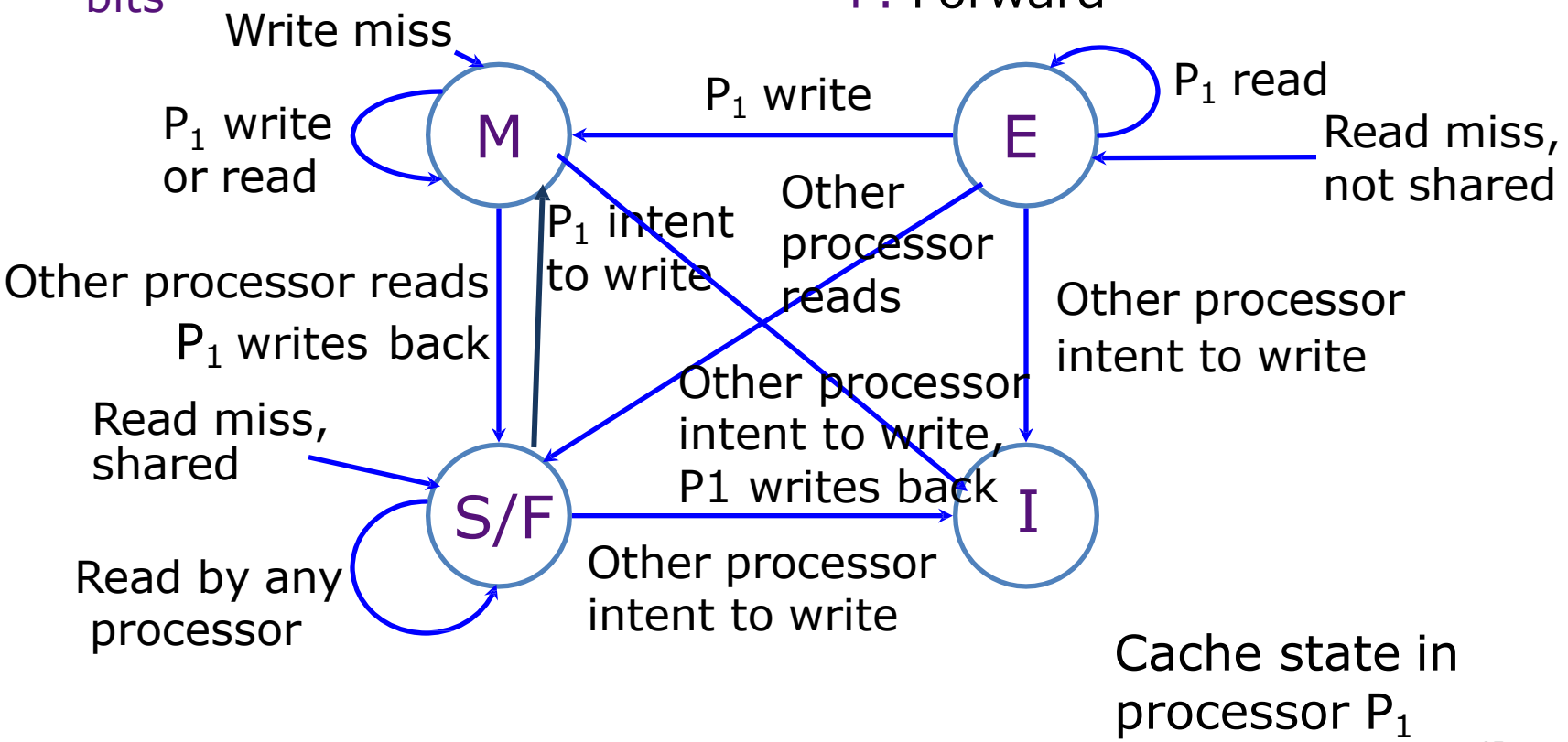


MESIF (Used by Intel Core i7)

Each cache line has a tag



M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid
F: Forward



Scalability Limitations of Snooping

- Caches
  - Bandwidth into caches
  - Tags need to be dual ported or steal cycles for snoops
  - Need to invalidate all the way to L1 cache
- Bus
  - Bandwidth
  - Occupancy (As number of cores grows, atomically utilizing bus becomes a challenge)

False Sharing

A cache block contains more than one word

Cache-coherence is done at the block-level and not word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same block address.

What can happen?

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)

- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750

Blackboard Example: Sequential Consistency

P1	P2
1	5
2	6
3	7
4	8

Valid

1	1	5
2	2	6
5	3	7
3	4	1
6	5	2
7	6	3
8	7	4
4	8	8

Not Valid

5
1
3
2
4
6
7
8

Analysis of Dekker's Algorithm

Scenario 1

```
...                    Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
                       then go to L
   < critical section>
c1=0;
```

```
...                    Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
                       then go to L
   < critical section>
c2=0;
```

Scenario 2

```
...                    Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
                       then go to L
   < critical section>
c1=0;
```

```
...                    Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
                       then go to L
   < critical section>
c2=0;
```