# Handout 3 – Multiprocessor and thread level parallelism

# Outline

- **Review**
- **MP Motivation**
- **SISD v. SIMD (SIMT) v. MIMD**
- **Centralized vs. Distributed Memory**
- **MESI and Directory Cache Coherency**
- **Synchronization and Relaxed Program Order**
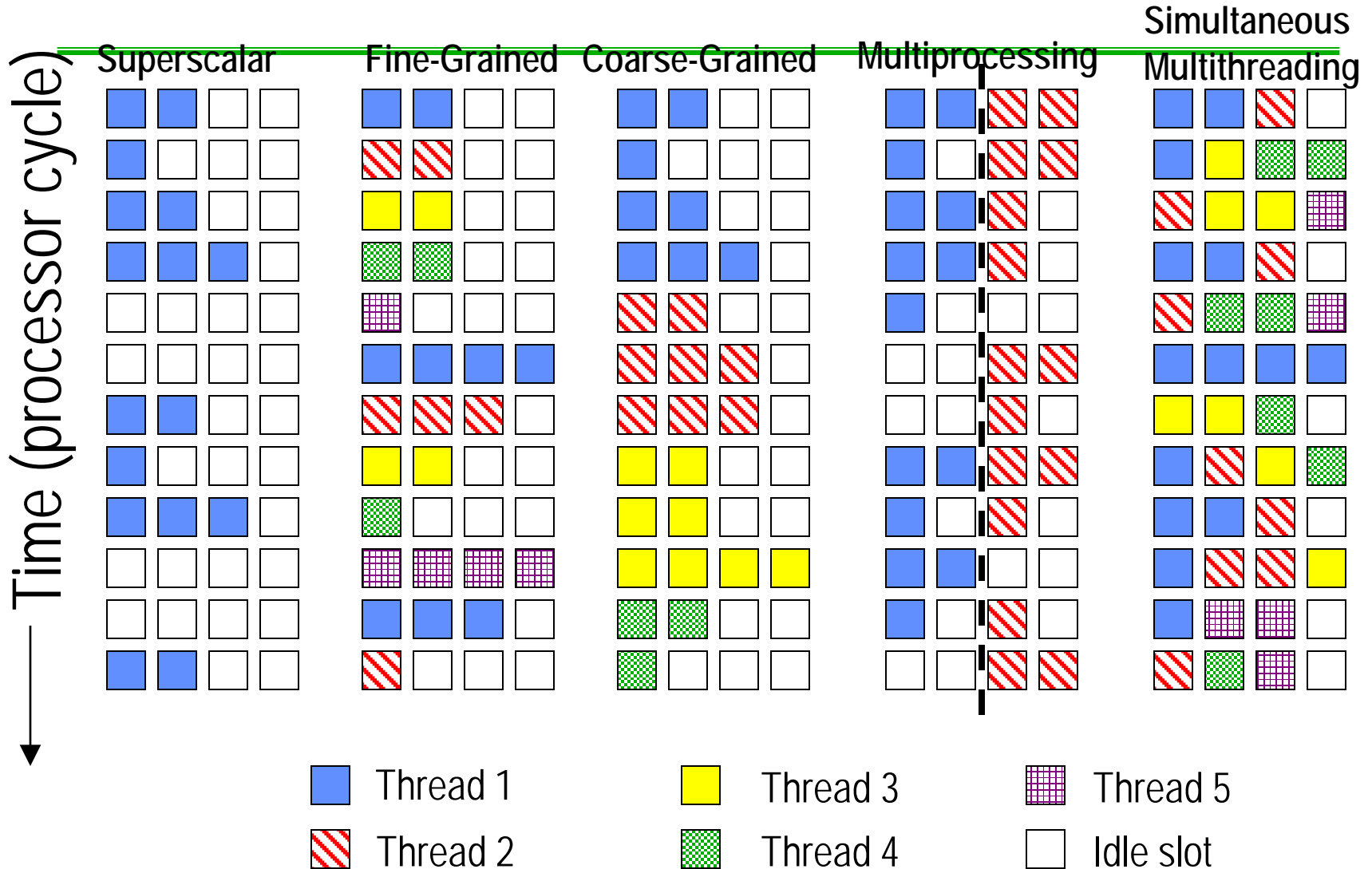- **Conclusion**

# Speculation: Register Renaming vs. ROB

- **Alternative to ROB is a larger physical set of registers combined with register renaming**
  - Extended registers replace function of both ROB and reservation stations

- **Instruction issue maps names of architectural registers to physical register numbers in extended register set**
  - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
  - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits

- **Most Out-of-Order processors today use extended registers with renaming**
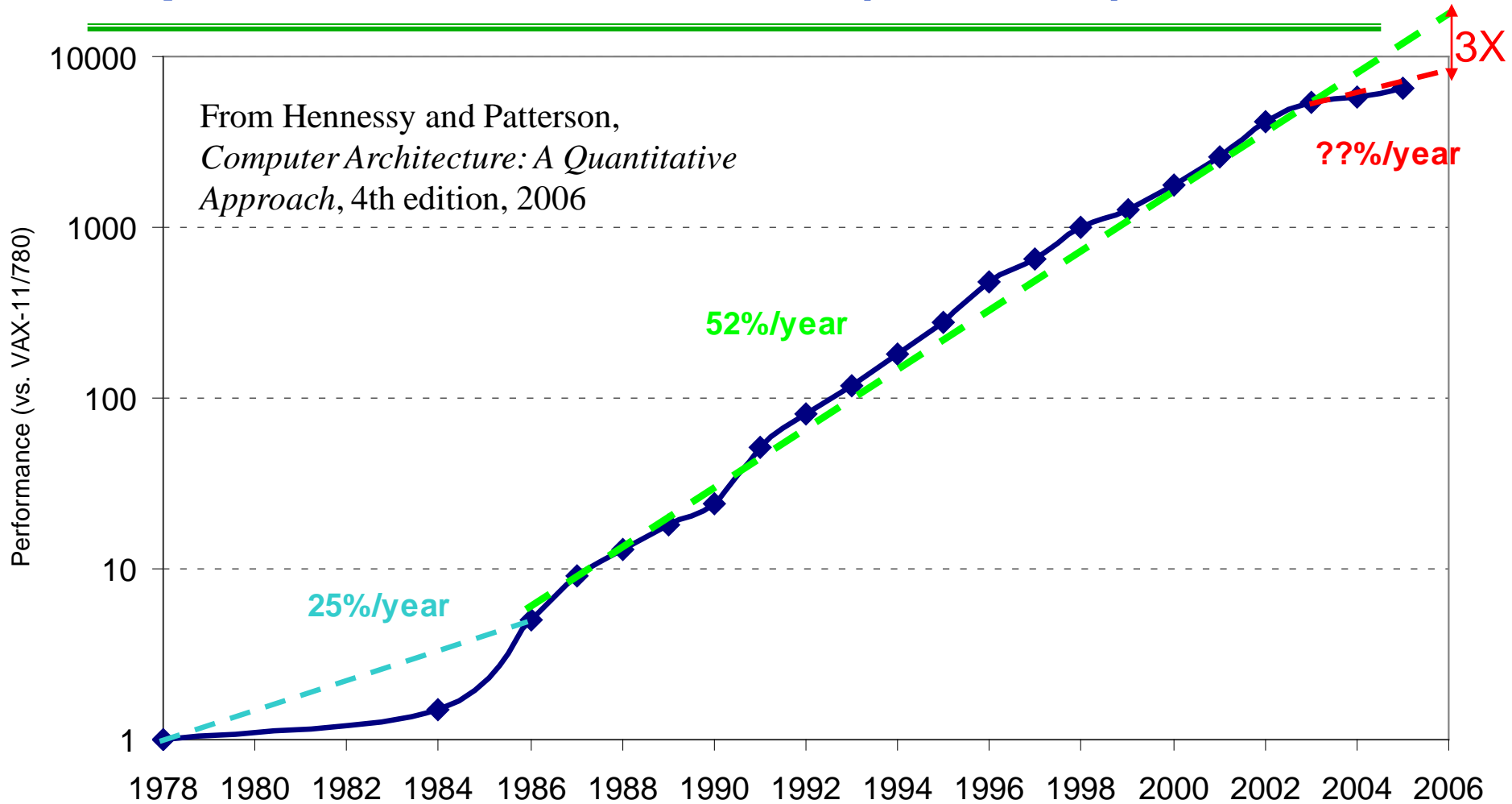
# Terminology:  Barrel threading

- **Interleaved multi-threading**
- Cycle i+1: an instruction from thread B is issued
- Cycle i+2: an instruction from thread C is issued
- The purpose of this type of multithreading is to remove all <u>data dependency</u> stalls from the execution <u>pipeline</u>. Since one thread is relatively independent from other threads, there's less chance of one instruction in one pipe stage needing an output from an older instruction in the pipeline.
- Conceptually, it is similar to <u>pre-emptive</u> multi-tasking used in operating systems. One can make the analogy that the time-slice given to each active thread is one CPU cycle.
- **Terminology**
- This type of multithreading was first called *Barrel processing,* in which the staves of a barrel represent the pipeline stages and their executing threads. *Interleaved* or *Pre-emptive* or ***Fine-grained*** or *time-sliced* multithreading are more modern terminology.

# Multithreaded Categories



Time (processor cycle)

| Superscalar | Fine-Grained | Coarse-Grained | Multiprocessing | Simultaneous Multithreading |
| --- | --- | --- | --- | --- |

Thread 1  Thread 3  Thread 5

Thread 2  Thread 4  Idle slot

# Uniprocessor Performance (SPECint)



From Hennessy and Patterson,
*Computer Architecture: A Quantitative Approach*, 4th edition, 2006

52%/year

25%/year

??%/year

3X

Performance (vs. VAX-11/780)

- **VAX        : 25%/year 1978 to 1986**
- **RISC + x86: 52%/year 1986 to 2002**
- **RISC + x86: ??%/year 2002 to present**

# Other factors that favor Multiprocessors

- **Growth in data-intensive applications**
  - Data bases, file servers, …

- **Growing interest in servers, server perf.**

- **Increasing desktop perf. less important**
  - Outside of graphics

- **Improved understanding in how to use multiprocessors effectively**
  - Especially server where significant natural TLP

- **Advantage of leveraging design investment by replication**
  - Rather than unique design

# Flynn's Taxonomy

- **Flynn classified by data and control streams in 1966**

| Single Instruction Single Data (SISD) <br><br> (Uniprocessor) | Single Instruction Multiple Data **SIMD** <br><br> (single PC: Vector, CM-2) |
|---|---|
| Multiple Instruction Single Data (MISD) <br><br> (????) | Multiple Instruction Multiple Data **MIMD** <br><br> (Clusters, SMP servers) |

- **SIMD $\Rightarrow$ Data Level Parallelism or SIMT**

- **MIMD $\Rightarrow$ Thread Level Parallelism**

- **MIMD popular because**
  - **Flexible: N pgms and 1 multithreaded pgm**
  - **Cost-effective: same MPU in desktop & MIMD**

# Back to Basics

- **"A parallel computer is a collection of processing elements that <u>cooperate</u> and communicate to solve large problems fast."**

- **Parallel Architecture = Computer Architecture + Communication Architecture**

- **2 classes of multiprocessors WRT memory:**

1. **Centralized Memory Multiprocessor**
   - **< few dozen processor chips (and < 100 cores) in 2006**
   - **Small enough to share single, centralized memory**

2. **Physically Distributed-Memory multiprocessor**
   - **Larger number chips and cores than 1.**
   - **BW demands $\Rightarrow$ Memory distributed among processors**

# Centralized vs. Distributed Memory
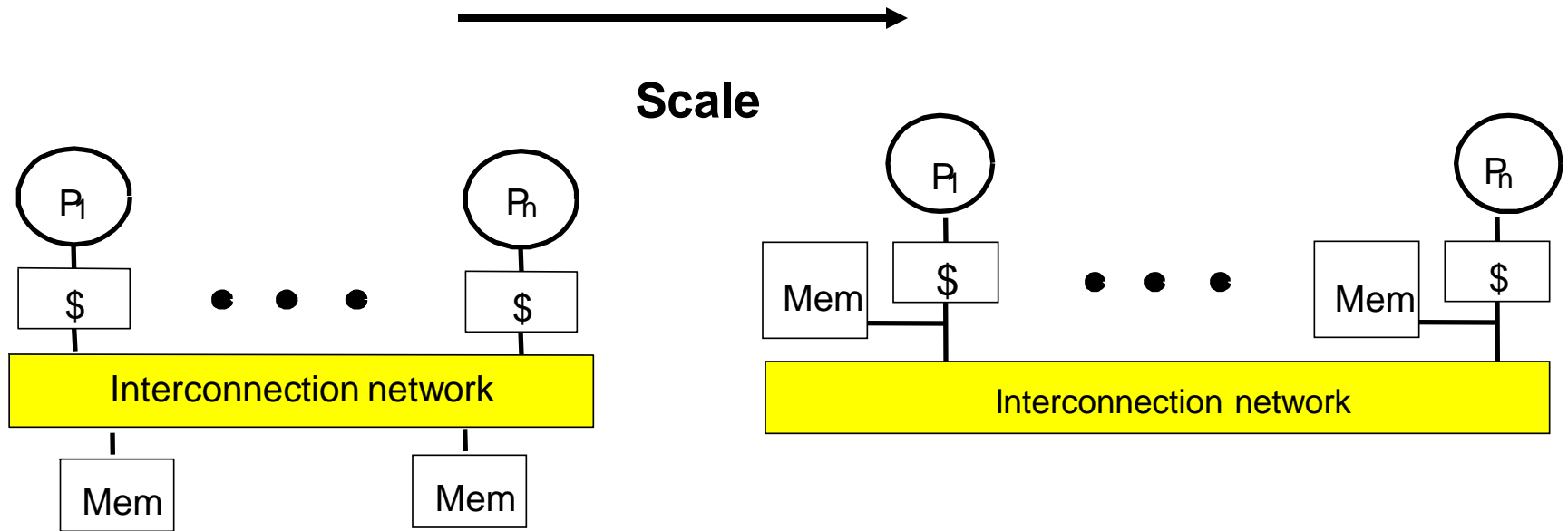


**Centralized Memory**            **Distributed Memory**

# Centralized Memory Multiprocessor

- **Also called <u>symmetric multiprocessors (SMPs)</u> because single main memory has a symmetric relationship to all processors**

- **Large caches $\Rightarrow$ single memory can satisfy memory demands of small number of processors**

- **Can scale to a few dozen processors by using a switch and by using many memory banks**

- **Although scaling beyond that is technically conceivable, it becomes less attractive as the number of processors sharing centralized memory increases**

# Distributed Memory Multiprocessor

- **Pro: Cost-effective way to scale memory bandwidth**

    - **If most accesses are to local memory**

- **Pro: Reduces latency of local memory accesses**

- **Con: Communicating data between processors more complex**

- **Con: Must change software to take advantage of increased memory BW**

# 2 Models for Communication and Memory Architecture

1. **Communication occurs by explicitly passing messages among the processors: message-passing multiprocessors**

2. **Communication occurs through a shared address space (via loads and stores): shared memory multiprocessors either**

   - **UMA (Uniform Memory Access time) for shared address, centralized memory MP**

   - **NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory MP**

   - **In past, confusion whether "sharing" means sharing physical memory (Symmetric MP) or sharing address space**

# Symmetric Shared-Memory Architectures

- **From multiple boards on a shared bus to multiple processors inside a single chip**
- **Caches both**
  - **Private data are used by a single processor**
  - **Shared data are used by multiple processors**
- **Caching shared data**
  **$\Rightarrow$ reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth**
  **$\Rightarrow$ cache coherence problem**

# Example Cache Coherence Problem
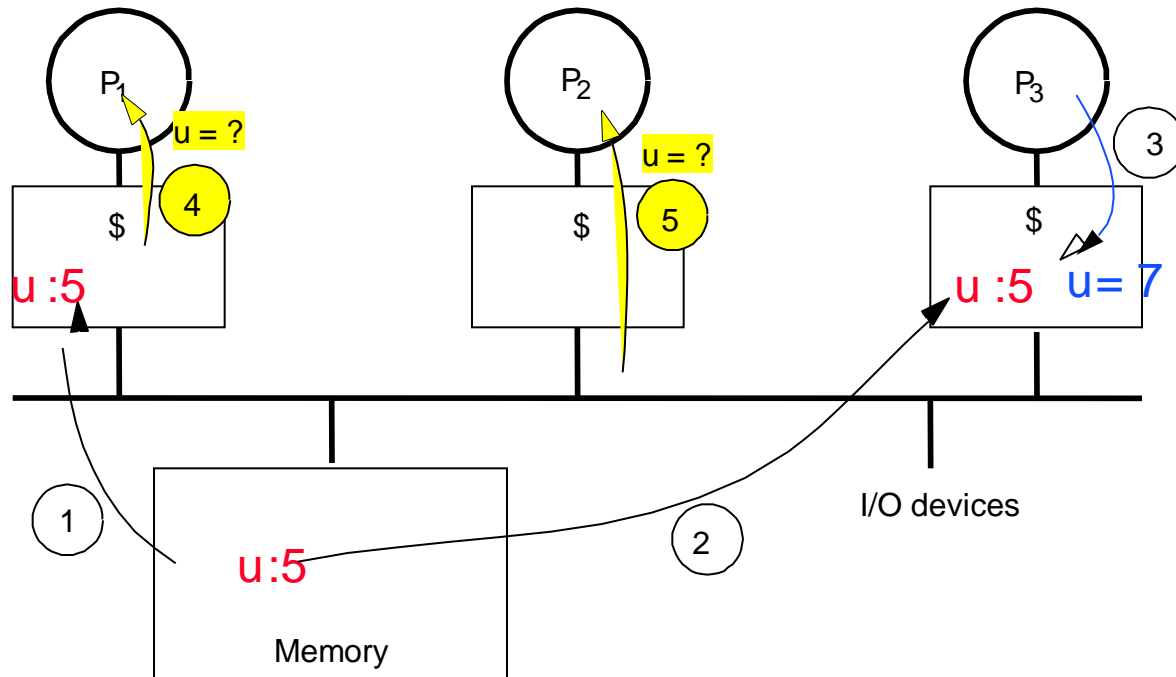


- **Processors see different values for u after event 3**
- **With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when**
    - » **Processes accessing main memory may see very stale value**
- **Unacceptable for programming, and it's frequent!**

# Basic Schemes for Enforcing Coherence

- **Program on multiple processors will normally have copies of the same data in several caches**
  - Unlike I/O, where it's rare

- **Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches**
  - Migration and Replication key to performance of shared data

- **Migration - data can be moved to a local cache and used there in a transparent fashion**
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory

- **Replication – for shared data being simultaneously read, since caches make a copy of data in local cache**
  - Reduces both latency of access and contention for read shared data

# Two Classes of Cache Coherence Protocols

1. **Directory based** — **Sharing status of a block of physical memory is kept in just one location, the directory**

2. **Snooping** — **Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept**

   - **All caches are accessible via some broadcast medium (a bus or switch)**

   - **All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access**

# Snoopy Cache-Coherence Protocols
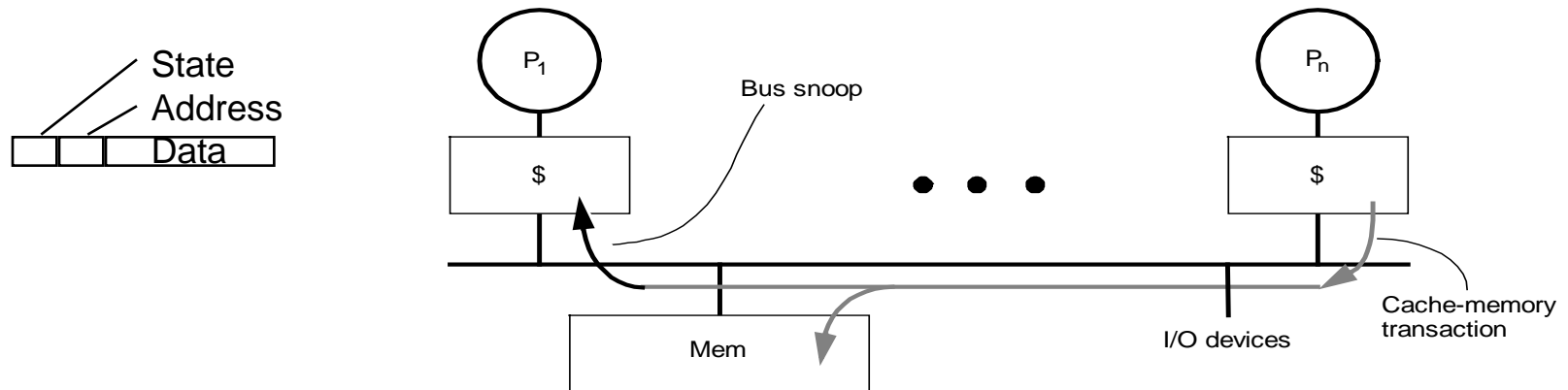


- **Cache Controller "snoops" all transactions on the shared medium (bus or switch)**
  - <u>relevant transaction</u> **if for a block it contains**
  - **take action to ensure coherence**
    - » **invalidate, update, or supply value**
  - **depends on state of the block and the protocol**
- **Either get exclusive access before write via write invalidate or update all copies on write**

# Example: Write-thru Invalidate
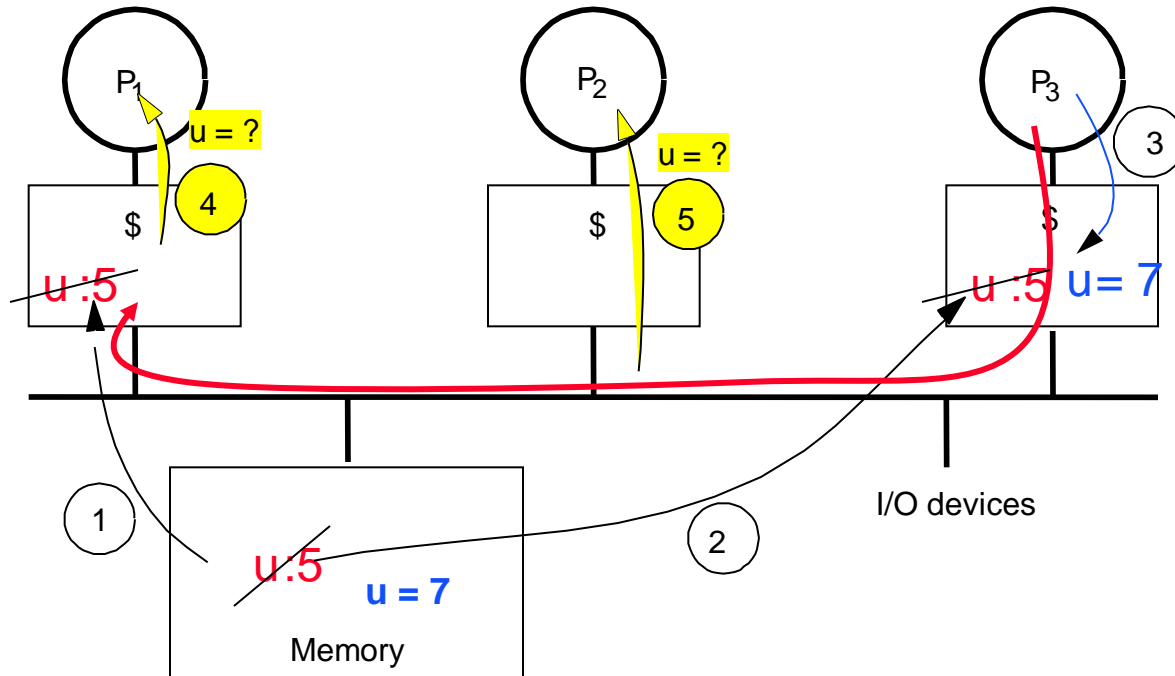


- **Must invalidate before step 3**
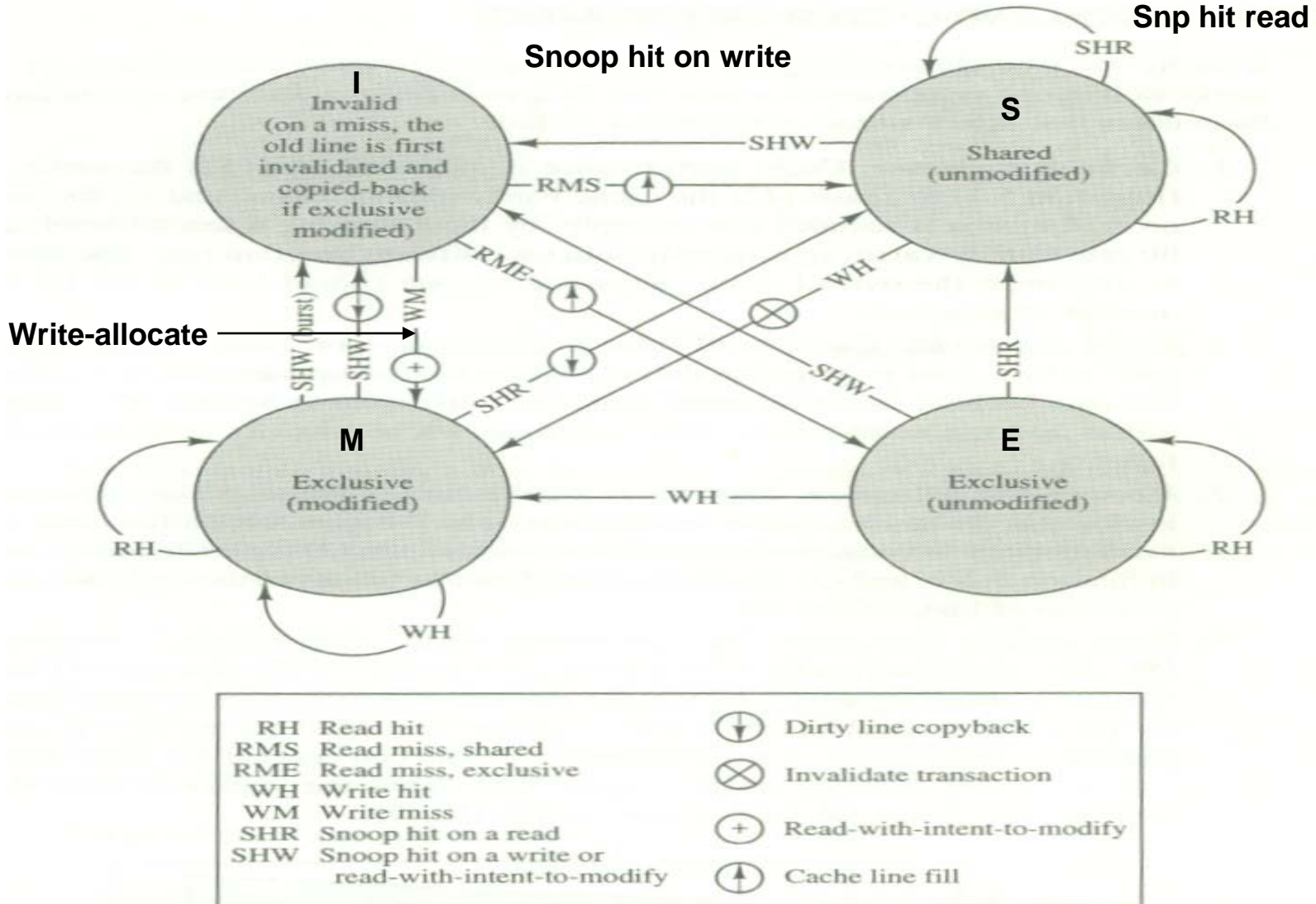- **Write update uses more broadcast medium BW ⇒ all recent MPUs use write invalidate**

# Architectural Building Blocks

- **Cache block state transition diagram**
  - **FSM specifying how disposition of block changes**
    - » **invalid, valid, dirty, shared**

- **Broadcast Medium Transactions (e.g., bus)**
  - **Fundamental system design abstraction**
  - **Logically single set of wires connect several devices**
  - **Protocol: arbitration, command/addr, data**
  - $\Rightarrow$ **Every device observes every transaction**

- **Broadcast medium enforces serialization of read or write accesses $\Rightarrow$ Write serialization**
  - **1st processor to get medium invalidates others copies**
  - **Implies cannot complete write until it obtains bus**
  - **All coherence schemes require serializing accesses to same cache block**

- **Also need to find up-to-date copy of cache block**

# MESI Protocol

**Each cache line will be associated with one of 4 states.
Invalid (I); Modified (M); Exclusive (E) or Shared (S)**



**Snp hit read**

**Snoop hit on write**

**Write-allocate**

**I** — Invalid (on a miss, the old line is first invalidated and copied-back if exclusive modified)

**S** — Shared (unmodified)

**M** — Exclusive (modified)

**E** — Exclusive (unmodified)

| | |
|---|---|
| RH | Read hit |
| RMS | Read miss, shared |
| RME | Read miss, exclusive |
| WH | Write hit |
| WM | Write miss |
| SHR | Snoop hit on a read |
| SHW | Snoop hit on a write or read-with-intent-to-modify |

- ⊕ Dirty line copyback
- ⊗ Invalidate transaction
- ⊕ Read-with-intent-to-modify
- ⊕ Cache line fill

(b) Diagram [4] (Reprinted by permission of Intel Corporation, Copyright Intel Corporation, 1991.)

21

# Locate up-to-date copy of data

- **Write-through: get up-to-date copy from memory**
  - **Write through simpler if enough memory BW**
- **Write-back harder**
  - **Most recent copy can be in a cache**
- **Can use same snooping mechanism**
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity from retrieving cache block from a processor cache, which can take longer than retrieving it from memory

- **Write-back needs lower memory bandwidth**
  **⇒ Support larger numbers of faster processors**
  **⇒ Most multiprocessors use write-back**

# Cache behavior in response to bus

- **Every bus transaction must check the cache-address tags**
  - **could potentially interfere with processor cache accesses**

- **A way to reduce interference is to duplicate tags**
  - **One set for caches access, one set for bus accesses**

- **Another way to reduce interference is to use L2 tags**
  - **Since L2 less heavily used than L1**
  - ⇒ **Every entry in L1 cache must be present in the L2 cache, called the <span style="color:red">inclusion property</span>**
  - **If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor**

# Scalable Approach: Directory

- **Every memory block has associated directory information**
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions

- **Many alternatives for organizing directory information**

# Directory Protocol

- **Similar to Snoopy Protocol: Three states**
  - **<u>Shared</u>: ≥ 1 processors have data, memory up-to-date**
  - **<u>Uncached</u> (no processor has it; not valid in any cache)**
  - **<u>Exclusive</u>: 1 processor (owner) has data;**
                              **memory out-of-date**

- **In addition to cache state, must track <u>which processors</u> have data when in the shared state (usually bit vector, 1 if processor has copy)**

# Scalable Approach: Directory
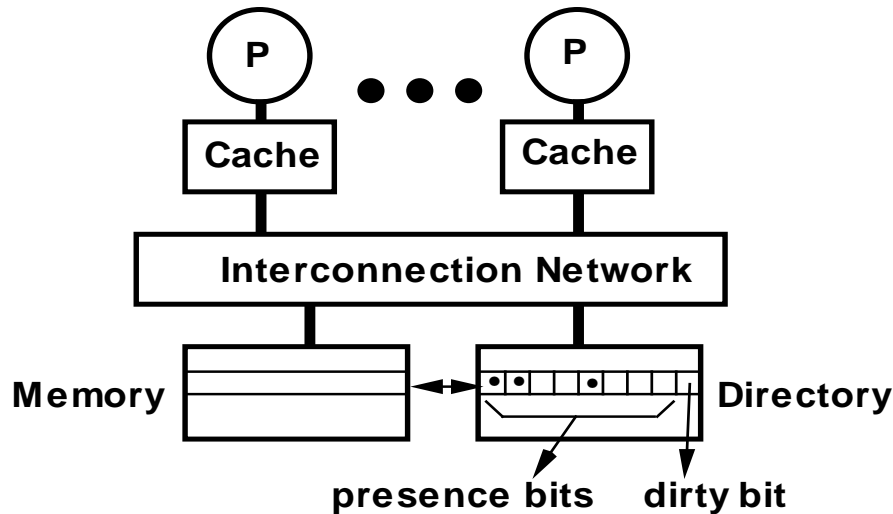


- k processors.
- With each cache-block in memory:
  k  presence-bits, 1 dirty-bit
- With each cache-block in cache:
  1 valid bit, and 1 dirty (owner) bit

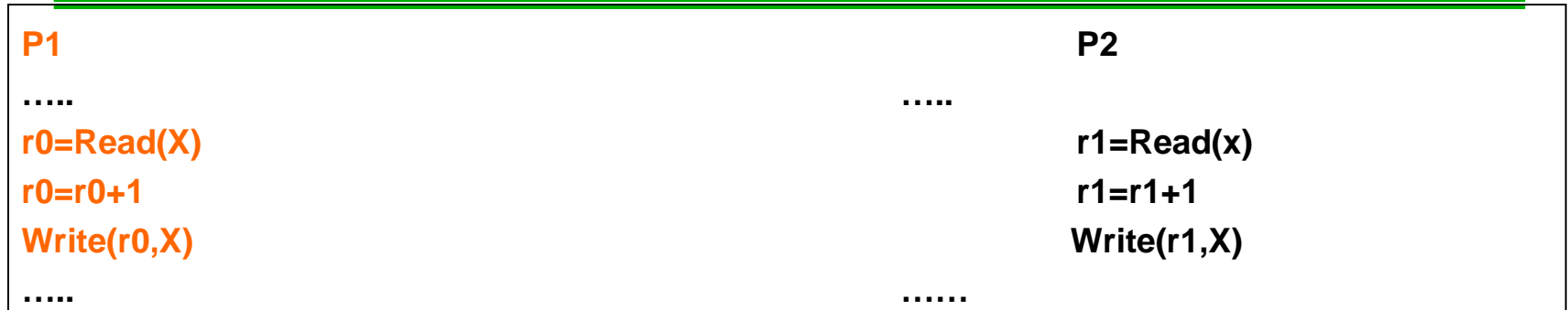- **Read from main memory by processor i:**

  - **If dirty-bit OFF then { read from main memory; turn p[i] ON; }**

  - **if dirty-bit ON   then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}**

- **Write to main memory by processor i:**

  - **If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }**

    - **...**

# Understanding Program Order

- **Initially X = 2**

| P1 | P2 |
|---|---|
| ….. | ….. |
| r0=Read(X) | r1=Read(x) |
| r0=r0+1 | r1=r1+1 |
| Write(r0,X) | Write(r1,X) |
| ….. | …… |

## Possible execution sequences:

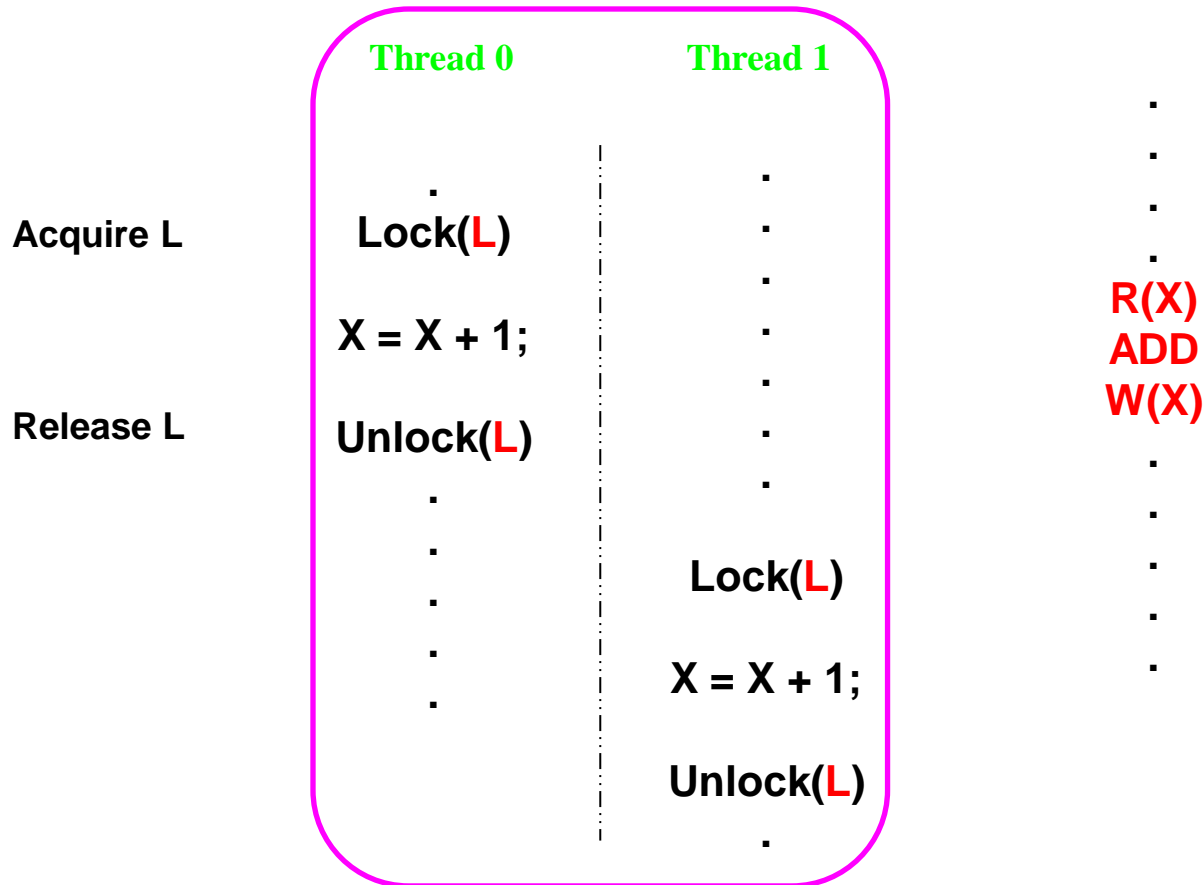| | | |
|---|---|---|
| P1:r0=Read(X) | P2:r1=Read(X) | P1: r0 = R(x) |
| P2:r1=Read(X), r1=2 | P2:r1=r1+1        , r1 = 3 | P1: r0 = r0+1 |
| P1:r0=r0+1        ,3=2+1 | P2:Write(r1,X) | P1: W(r0, x) |
| P1:Write(r0,X) | P1:r0=Read(X)    , r0 = 3 | |
| P2:r1=r1+1    , r1=2+1 =3 | P1:r0=r0+1        , r0 = 3+1=4 | P2: r1=R(x) |
| P2:Write(r1,X) | P1:Write(r0,X) | P2: r1 = r1 +1 |
| x=3      in memory | x=4  in memory | P2: W(r1, x) |
| | | X = 4 in memory |

**Data race:**

there is a variable modified by more than one process in a way such that the results depend on who gets there first.

**Non-mutual exclusive access to shared item X results in different outcome- Data race**
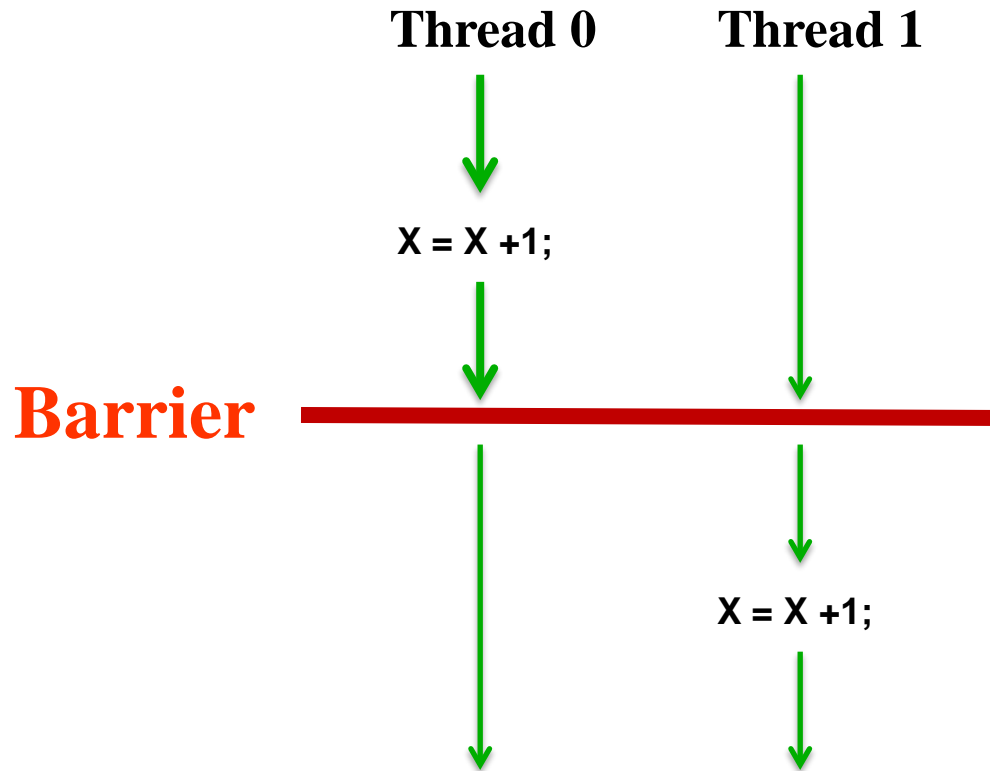
# Lock(L):   L is a Lock at an address

- **Two threads access one shared variable**

|  | Thread 0 | Thread 1 |  |
|---|---|---|---|
|  |  |  | . |
|  |  |  | . |
|  |  |  | . |
|  | . | . | . |
| Acquire L | Lock(L) | . | |
|  |  | . | **R(X)** |
|  | X = X + 1; | . | **ADD** |
|  |  | . | **W(X)** |
| Release L | Unlock(L) | . | |
|  | . | . | . |
|  | . | Lock(L) | . |
|  | . |  | . |
|  | . | X = X + 1; | . |
|  |  |  | . |
|  |  | Unlock(L) | |
|  |  | . | |

# Synchronization through a barrier

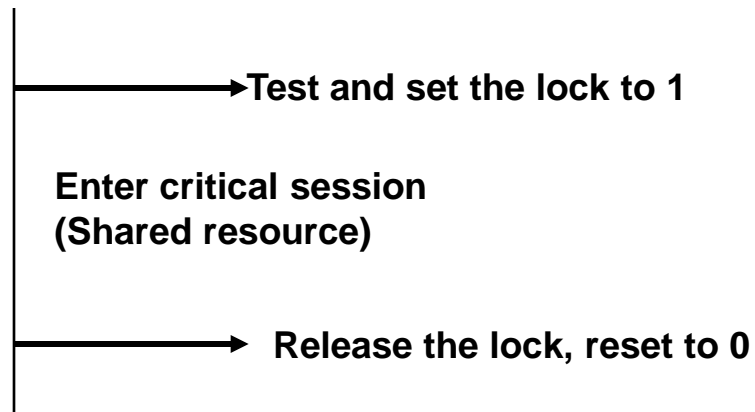Thread 0          Thread 1

X = X +1;

**Barrier**

X = X +1;

# Synchronization

- **Why synchronization ?**
  - **Processors communicate through shared variables.**
- **Hardware supports for synchronization**
  - **Instruction that performs read-modify-write in an atomic way.**
- **Example instructions**
  - **0 : lock unclaimed, 1: taken**
  - **Test and set**
  - **Fetch-and-increment** → **Test and set the lock to 1**
  - **Exchange**

**Enter critical session
(Shared resource)**

**Release the lock, reset to 0**

# Spin lock

- **A spin lock whose address is in R1**
  - **R2 = 1, write 1 to Mem[R1], read Mem[R1] back to R2 (done before the write).**

- **Problem**
  - **Spin lock ties up the processor.**
  - **Spin lock does not scale.**
  - **In cache coherence multiprocessor, a write of getting the lock, will generate a write miss (each processor wants to get the lock executing exch.) This may cause huge traffic in the interconnection.**

```
                DADDUI     R2,R0,#1
lockit:         EXCH       R2,0(R1)        ;atomic exchange
                BNEZ       R2,lockit       ;already locked?
```

# Spin on Local Copy

- **Read it first. This causes the copy to reside in the local cache (shared state).**

```
lockit:  LD       R2,0(R1)     ;load of lock
         BNEZ     R2,lockit    ;not available-spin
         DADDUI   R2,R0,#1     ;load locked value
         EXCH     R2,0(R1)     ;swap
         BNEZ     R2,lockit    ;branch if lock wasn't 0
```

# Coherency vs Consistency

- **A consistent memory means that the entire address space as seen by a program (that may be running as multiple processes on a single processor or on multiple processors) will be consistent.**

- **What does consistent mean?**

- **All processes/processor will see the entire memory containing the same values at all times.**
  - **You see what I see.**

- **Coherency is related to reads/writes of a single data item.** Cache coherence only requires a locally (*i.e.* per-location) consistent view.

- **If every single data item is coherent, then the memory is consistent.**
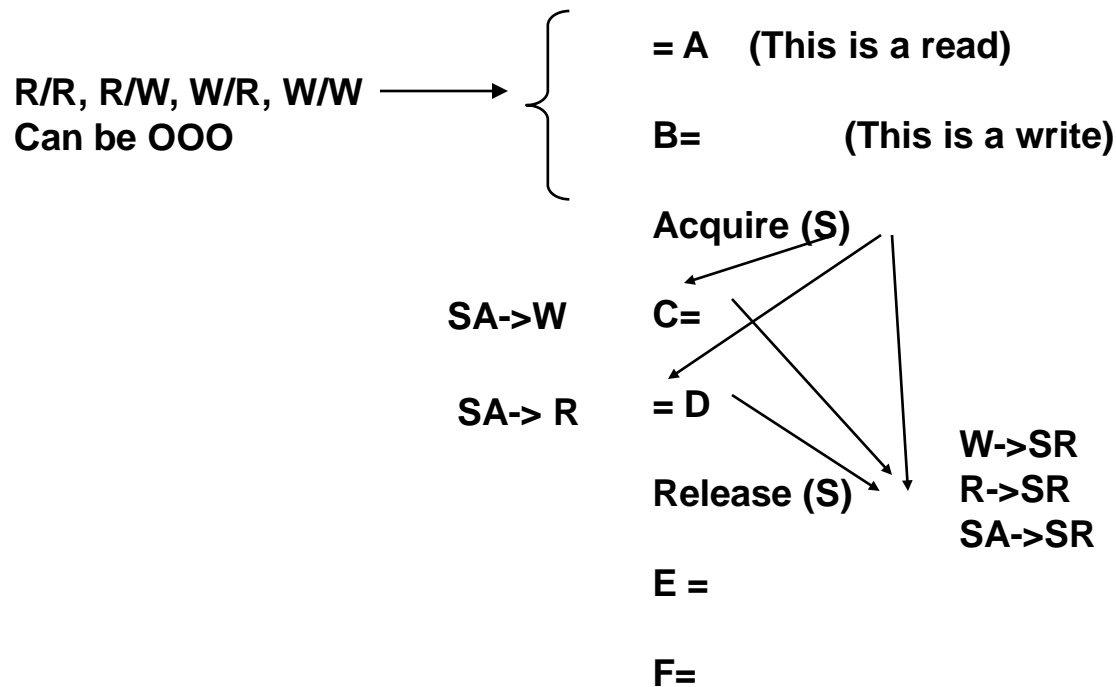
# Relaxing Program Orders

- **Divide memory operations into data operations and synchronization operations**

- **Synchronization operations act like a fence:**
  - **All data operations before synch in program order must complete before synch is executed**
  - **All data operations after synch in program order must wait for synch to complete**
  - **Synchs are performed in program order**

- **Implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed**

# Relaxed Consistency Model

- ## Release consistency

R/R, R/W, W/R, W/W
Can be OOO

= A   (This is a read)

B=        (This is a write)

Acquire (S)

SA->W     C=

SA-> R     = D

Release (S)

W->SR
R->SR
SA->SR

E =

F=

# And in Conclusion

- "End" of uniprocessors speedup => Multiprocessors
- Parallelism challenges: % parallalizable, long latency to remote memory
- Centralized vs. distributed memory
  - Small MP vs. lower latency, larger BW for Larger MP
- Message Passing vs. Shared Address
  - Uniform access time vs. Non-uniform access time
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data $\Rightarrow$ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- Shared medium serializes writes
  $\Rightarrow$ Write consistency