Chapter-4

Multiprocessors and Thread-Level Parallelism

We have seen the renewed interest in developing multiprocessors in early 2000:
- The slowdown in uniprocessor performance due to the diminishing returns in exploring
  instruction-level parallelism.
- Difficulty to dissipate the heat generated by uniprocessors with high clock rates.
- Demand for high-performance servers where thread-level parallelism is natural.
For all these reasons multiprocessor architectures has become increasingly attractive.

A Taxonomy of Parallel Architectures
The idea of using multiple processors both to increase performance and to improve
availability dates back to the earliest electronic computers. About 30 years ago, Flynn
proposed a simple model of categorizing all computers that is still useful today. He
looked at the parallelism in the instruction and data streams called for by the instructions
at the most constrained component of the multiprocessor, and placed all computers in one
of four categories:
1. Single instruction stream, single data stream
   (SISD)—This category is the uniprocessor.

   2. Single instruction stream, multiple data streams
(SIMD)—The same instruction is executed by multiple processors using different data streams. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. Vector architectures are the largest class of processors of this type.

3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date, but may be in the future. Some special purpose stream processors approximate a limited form of this (there is only a single data stream that is operated on by successive functional units).
4. **Multiple instruction streams, multiple data streams** (MIMD)—Each processor fetches its own instructions and operates on its own data. The processors are often off-the-shelf microprocessors. This is a coarse model, as some multiprocessors are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.

2. MIMDs can build on the cost/performance advantages of off-the-shelf microprocessors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers. With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. Recall from the last chapter, that a process is an segment of code that may be run independently, and that the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of the processes on other processors.

It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called *threads*. 
Today, the term thread is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space. To take advantage of an MIMD multiprocessor with \( n \) processors, we must usually have at least \( n \) threads or processes to execute. The independent threads are typically identified by the programmer or created by the compiler. Since the parallelism in this situation is contained in the threads, it is called *thread-level parallelism*.

Threads may vary from large-scale, independent processes—for example, independent programs running in a multiprogrammed fashion on different processors—to parallel iterations of a loop, automatically generated by a compiler and each executing for perhaps less than a thousand instructions. Although the size of a thread is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction is that such parallelism is identified at a high-level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel. In contrast, instruction level parallelism is identified by primarily by the hardware, though with software help in some cases, and is found and exploited one instruction at a time.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictate a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization, because what constitutes a small or large number of processors is likely to change over time.

The first group, which we call *centralized shared-memory architectures*
Centralized shared memory architectures have at most a few dozen processors in 2000. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By replacing a single bus with multiple buses, or even a switch, a centralized shared memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable, sharing a centralized memory, even organized as multiple banks, becomes less attractive as the number of processors sharing it increases.

Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are often called symmetric (shared-memory) multiprocessors (SMPs), and this style of architecture is sometimes called UMA for uniform memory access. This type of centralized shared-memory architecture is currently by far the most popular organization.

The second group consists of multiprocessors with physically distributed memory. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor’s memory bandwidth requirements, the scale of multiprocessor for which distributed memory is preferred over a single, centralized memory continues to
decrease in number (which is another reason not to use small and large scale). Of course, the larger number of processors raises the need for a high bandwidth interconnect.

Distributed-memory multiprocessor

Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth, if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory bandwidth and lower memory latency. The key disadvantage for a distributed memory architecture is that communicating data between processors becomes somewhat more complex and has higher latency, at least when there is no contention, because the processors no longer share a single centralized memory. As we will see shortly, the use of distributed memory leads to two different paradigms for interprocessor communication.

Typically, I/O as well as memory is distributed among the nodes of the multiprocessor, and the nodes may be small SMPs (2–8 processors). Although the use of multiple processors in a node together with a memory and a network interface is quite useful from the cost-efficiency viewpoint.

Challenges for Parallel Processing

- Limited parallelism available in programs
  - Need new algorithms that can have better parallel performance
- Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?
Data Communication Models for Multiprocessors

- shared memory: access shared address space implicitly via load and store operations.
- message-passing: done by explicitly passing messages among the processors
  - can invoke software with Remote Procedure Call (RPC)
  - often via library, such as MPI: Message Passing Interface
  - also called "Synchronous communication" since communication causes synchronization between 2 processes

Message-Passing Multiprocessor

- The address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor

- The same physical address on two different processors refers to two different locations in two different memories.

Multicomputer (cluster):

- can even consist of completely separate computers connected on a LAN.
- cost-effective for applications that require little or no communication.

Symmetric Shared-Memory Architectures
Multilevel caches can substantially reduce the memory bandwidth demands of a processor.

This is extremely

- Cost-effective
- This can work as plug in play by placing the processor and cache sub-system on a board into the bus backplane.

Developed by

- IBM – One chip multiprocessor
- AMD and INTEL- Two –Processor
- SUN – 8 processor multi core

Symmetric shared – memory support caching of

- Shared Data
- Private Data

**Private data:** used by a single processor

When a private item is cached, its location is *migrated* to the cache Since no other processor uses the data, the program behavior is identical to that in a uniprocessor.

**Shared data:** used by multiple processor

When shared data are cached, the shared value may be *replicated* in multiple caches

advantages: reduce access latency and memory contention induces a new problem: cache coherence.

**Cache Coherence**

Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values.

I.e, If two different processors have two different values for the same location, this difficulty is generally referred to as cache coherence problem
Cache coherence problem for a single memory location

- **Informally:**
  - “Any read must return the most recent write”
  - Too strict and too difficult to implement

- **Better:**
  - “Any write must eventually be seen by a read”
  - All writes are seen in proper order (“serialization”)

- **Two rules to ensure this:**
  - “If P writes x and then P1 reads it, P’s write will be seen by P1 if the read and write are sufficiently far apart”
  - Writes to a single location are serialized: seen in one order
    - Latest write will be seen
    - Otherwise could see writes in illogical order (could see older value after a newer value)

The definition contains two different aspects of memory system:

- Coherence
- Consistency

A memory system is coherent if,

- Program order is preserved.
- Processor should not continuously read the old data value.
- Write to the same location are serialized.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache contents for CPU A</th>
<th>Cache contents for CPU B</th>
<th>Memory contents for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU A stores 0 into X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The above three properties are sufficient to ensure coherence. *When a written value will be seen is also important.* This issue is defined by memory consistency model. Coherence and consistency are complementary.

**Basic schemes for enforcing coherence**

Coherence cache provides:

- migration: a data item can be moved to a local cache and used there in a transparent fashion.
- replication for shared data that are being simultaneously read.
- both are critical to performance in accessing shared data.

To overcome these problems, adopt a hardware solution by introducing a protocol to maintain coherent caches named as Cache Coherence Protocols.

These protocols are implemented for tracking the state of any sharing of a data block.

Two classes of Protocols

- Directory based
- Snooping based

**Directory based**

- Sharing status of a block of physical memory is kept in one location called the directory.
- Directory-based coherence has slightly higher implementation overhead than snooping.
- It can scale to larger processor count.

**Snooping**

- Every cache that has a copy of data also has a copy of the sharing status of the block.
- No centralized state is kept.
- Caches are also accessible via some broadcast medium (bus or switch)
- Cache controller monitor or snoop on the medium to determine whether or not they have a copy of a block that is represented on a bus or switch access.
Snooping protocols are popular with multiprocessor and caches attached to single shared memory as they can use the existing physical connection- bus to memory, to interrogate the status of the caches. Snoop based cache coherence scheme is implemented on a shared bus. Any communication medium that broadcasts cache misses to all the processors.

**Basic Snoopy Protocols**

- **Write strategies**
  - Write-through: memory is always up-to-date
  - Write-back: snoop in caches to find most recent copy

- **Write Invalidate Protocol**
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and **invalidate** any copies
    - Read miss: further read will miss in the cache and fetch a new copy of the data.

- **Write Broadcast/Update Protocol (typically write through)**
  - Write to shared data: broadcast on bus, processors snoop, and **update** any copies
    - Read miss: memory/cache is always up-to-date.

- **Write serialization**: bus serializes requests!
  - Bus is single point of arbitration

**Examples of Basic Snooping Protocols**

**Write Invalidate**
An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

**Write Update**

An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

Assume neither cache initially holds X and the value of X in memory is 0

**Example Protocol**

- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

**Example Write Back Snoopy Protocol**

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state:
  - Clean in all caches and up-to-date in memory *(Shared)*
  - OR Dirty in exactly one cache *(Exclusive)*
  - OR Not in any caches
- Each cache block is in one state (track these):
  - *Shared*: block can be read
  - OR *Exclusive*: cache has only copy, its writeable, and dirty
  - OR *Invalid*: block contains no data (in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses

**Write-Back State Machine – CPU**

**State Transitions for Each Cache Block is as shown below**
- CPU may read/write hit/miss to the block
- May place write/read miss on bus
- May receive read/write miss from bus

Cache Coherence State Diagram
### Request | Source | State of block | Function and explanation
---|---|---|---
Read hit | processor | shared or exclusive | Read data in cache.
Read miss | processor | invalid | Place read miss on bus.
Read miss | processor | shared | Address conflict miss: place read miss on bus.
Read miss | processor | exclusive | Address conflict miss: write back block, then place read miss on bus.
Write hit | processor | exclusive | Write data in cache.
Write hit | processor | shared | Place write miss on bus.
Write miss | processor | invalid | Place write miss on bus.
Write miss | processor | shared | Address conflict miss: place write miss on bus.
Write miss | processor | exclusive | Address conflict miss: write back block, then place write miss on bus.
Read miss | bus | shared | No action; allow memory to service read miss.
Read miss | bus | exclusive | Attempt to share data: place cache block on bus and change state to shared.
Write miss | bus | shared | Attempt to write shared block; invalidate the block.
Write miss | bus | exclusive | Attempt to write block that is exclusive elsewhere: write back the cache block and make its state invalid.
**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 => Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- Shared medium serializes writes => Write consistency

**Implementation Complications**
- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - Bus transaction is not atomic:
      can have multiple outstanding transactions for a block
• Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
• Must track and prevent multiple misses for one block
• Must support interventions and invalidations

Performance Measurement
• Overall cache performance is a combination of
  – Uniprocessor cache miss traffic
  – Traffic caused by communication – invalidation and subsequent cache misses
• Changing the processor count, cache size, and block size can affect these two components of miss rate
• Uniprocessor miss rate: compulsory, capacity, conflict
• Communication miss rate: coherence misses
  – True sharing misses + false sharing misses

True and False Sharing Miss
• True sharing miss
  – The first write by a PE to a shared cache block causes an invalidation to establish ownership of that block
  – When another PE attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred
• False sharing miss
  – Occur when a block a block is invalidate (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written to
  – The block is shared, but no word in the cache is actually shared, and this miss would not occur if the block size were a single word
• Assume that words x1 and x2 are in the same cache block, which is in the shared state in the caches of P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss or a false sharing miss.

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write x1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Read x2</td>
</tr>
<tr>
<td>3</td>
<td>Write x1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write x2</td>
</tr>
<tr>
<td>5</td>
<td>Read x2</td>
<td></td>
</tr>
</tbody>
</table>

**Example Result**

• True sharing miss (invalidate P2)
• 2: False sharing miss
  – x2 was invalidated by the write of P1, but that value of x1 is not used in P2
• 3: False sharing miss
  – The block containing x1 is marked shared due to the read in P2, but P2 did not read x1. A write miss is required to obtain exclusive access to the block
• 4: False sharing miss
• 5: True sharing miss

**Distributed Shared-Memory Architectures**

Distributed shared-memory architectures

• Separate memory per processor
  – Local or remote access via memory controller
  – The physical address space is statically distributed
Coherence Problems

• Simple approach: uncacheable
  – shared data are marked as uncacheable and only private data are kept in caches
  – very long latency to access memory for shared data

• Alternative: directory for memory blocks
  – The directory per memory tracks state of every block in every cache
    • which caches have a copies of the memory block, dirty vs. clean, ...
  – Two additional complications
    • The interconnect cannot be used as a single point of arbitration like the bus
    • Because the interconnect is message oriented, many messages must have explicit responses

To prevent directory becoming the bottleneck, we distribute directory entries with memory, each keeping track of which processors have copies of their memory blocks

Directory Protocols

• Similar to Snoopy Protocol: Three states
  – **Shared**: 1 or more processors have the block cached, and the value in memory is up-to-date (as well as in all the caches)
  – **Uncached**: no processor has a copy of the cache block (not valid in any cache)
  – **Exclusive**: Exactly one processor has a copy of the cache block, and it has written the block, so the memory copy is out of date
    • The processor is called the owner of the block

• In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared (usually a bit vector for each memory block: 1 if processor has copy)

• Keep it simple(r):
  – Writes to non-exclusive data => write miss
- Processor blocks until access completes
- Assume messages received and acted upon in order sent

**Messages for Directory Protocols**

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Message contents</th>
<th>Function of this message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>local cache</td>
<td>home directory</td>
<td>P, A</td>
<td>Processor P has a read miss at address A; request data and make P a read sharer.</td>
</tr>
<tr>
<td>Write miss</td>
<td>local cache</td>
<td>home directory</td>
<td>P, A</td>
<td>Processor P has a write miss at address A; request data and make P the exclusive owner.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Invalidate a shared copy of data at address A.</td>
</tr>
<tr>
<td>Fetch</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.</td>
</tr>
<tr>
<td>Fetch/invalidate</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; invalidate the block in the cache.</td>
</tr>
<tr>
<td>Data value reply</td>
<td>home directory</td>
<td>local cache</td>
<td>D</td>
<td>Return a data value from the home memory.</td>
</tr>
<tr>
<td>Data write back</td>
<td>remote cache</td>
<td>home directory</td>
<td>A, D</td>
<td>Write back a data value for address A.</td>
</tr>
</tbody>
</table>

- local node: the node where a request originates
- home node: the node where the memory location and directory entry of an address reside
- remote node: the node that has a copy of a cache block (exclusive or shared)

**State Transition Diagram for Individual Cache Block**
• Comparing to snooping protocols:
  – identical states
  – stimulus is almost identical
  – write a shared cache block is treated as a write miss (without fetch the block)
  – cache block must be in exclusive state when it is written
  – any shared block must be up to date in memory
• write miss: data fetch and selective invalidate operations sent by the directory controller (broadcast in snooping protocols)

**Directory Operations: Requests and Actions**
• Message sent to directory causes two actions:
  – Update the directory
  – More messages to satisfy request
• Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
– Read miss: requesting processor sent data from memory & requestor made only sharing node; state of block made Shared.

– Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

• Block is Shared => the memory value is up-to-date:

– Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.

– Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

• Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:

– Read miss: owner processor sent data fetch message, causing state of block in owner’s cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor.
Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.

– Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.

– Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

**Synchronization : The Basics**
Synchronization mechanisms are typically built with user-level software routines that rely on hardware– supplied synchronization instructions.

- Why Synchronize?
  Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction to Fetch and Update Memory
- Atomic exchange: interchange a value in a register for a value in memory
  0 ⇒ synchronization variable is free
  1 ⇒ synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    0 if you succeeded in setting the lock (you were first)
    1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - 0 ⇒ synchronization variable is free
- Hard to have read & write in 1 instruction: use 2 instead
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
- Example doing atomic swap with LL & SC:
User Level Synchronization—Operation Using this Primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock
  
  ```
  li     R2,#1
  ```
  
  ```
  lockit: exch R2,0(R1) ; atomic exchange
  ```
  
  ```
  bnez R2,lockit ; already locked?
  ```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables

- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):

  ```
  try: li R2,#1
  ```
  
  ```
  lockit: lw R3,0(R1) ; load var
  ```
  
  ```
  bnez R3,lockit ; ≠ 0 ⇒ not free ⇒ spin
  ```
  
  ```
  exch R2,0(R1) ; atomic exchange
  ```
  
  ```
  bnez R2,try ; already locked?
  ```
Memory Consistency Models

• What is consistency? When must a processor see the new value? e.g., seems that
  P1: A = 0;   P2: B = 0;
      ......     ......
      A = 1;     B = 1;
L1: if (B == 0) ...  L2: if (A == 0) ...

• Impossible for both if statements L1 & L2 to be true?
  – What if write invalidate is delayed & processor continues?

• Memory consistency models:
  what are the rules for such cases?

• Sequential consistency: result of any execution is the same as if the accesses of
each processor were kept in order and the accesses among different processors
were interleaved ⇒ assignments before ifs above
  – SC: delay all memory accesses until all invalidates done

• Schemes faster execution to sequential consistency

• Not an issue for most programs; they are synchronized
  – A program is synchronized if all access to shared data are ordered by
    synchronization operations
    write (x)

... release (s) \{unlock\}

... acquire (s) \{lock\}

... read(x)

• Only those programs willing to be nondeterministic are not synchronized: “data
race”: outcome f(proc. speed)

• Several Relaxed Models for Memory Consistency since most programs are
synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW
to different addresses
Relaxed Consistency Models: The Basics

- **Key idea**: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent
  - By relaxing orderings, may obtain performance advantages
  - Also specifies range of legal compiler optimizations on shared data
  - Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program

- 3 major sets of relaxed orderings:
  1. W→R ordering (all writes completed before next read)
     - Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called **processor consistency**
  2. W → W ordering (all writes completed before next write)
  3. R → W and R → R orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering

- Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written