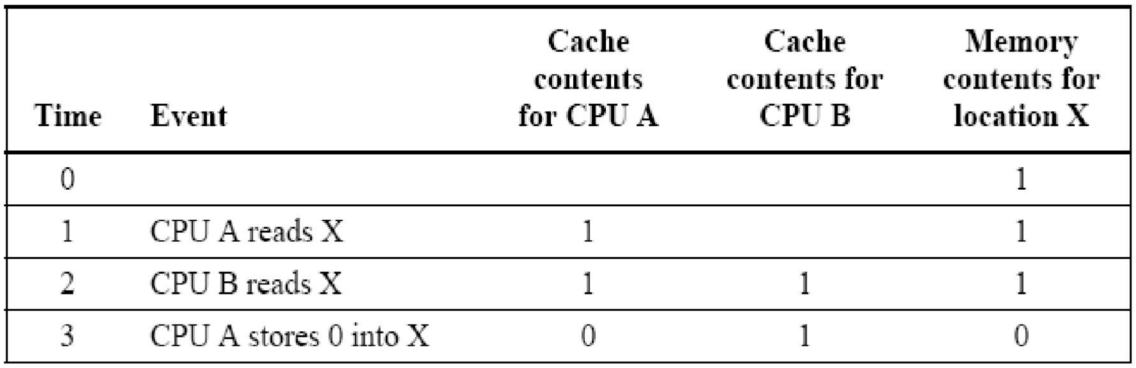
Cache Coherence

Figure below illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty s generally referred to as the *cache-coherence* problem.



Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

The cache-coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 1!

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs.

The first aspect, called *coherence*, defines what values can be returned by a read.

The second aspect, called *consistency*, determines when a written value will be returned by a read. Let's look at coherence first.

A memory system is coherent if

1.A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

3. Writes to the same location are *serialized*: that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

The first property simply preserves program order—we expect this property to be true even in uniprocessors. The second property defines the notion of what it means to have a coherent view of memory: If a processor could continuously read an old data value, we would clearly say that memory was incoherent. The need for write serialization is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processor could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely. The simplest way to avoid such difficulties is to serialize writes, so that all writes to the same location are seen in the same order; this property is called *write serialization*.

Although the three properties just described are sufficient to ensure coherence, the question of when a written value will be seen is also important. To see why, observe that we cannot require that a read of X instantaneously see the value written for X by some other processor. If, for example, a write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly *when* a written value must be seen by a reader is defined by a *memory consistency model*.

Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations. For simplicity, and because we cannot explain the problem in full detail at this point, assume that we require that a write does not complete until all processors have seen the effect of the write and that the processor does not change the order of any write with any other memory access. This allows the processor to reorder reads, but forces the processor to finish a write in program order.

Basic Schemes for Enforcing Coherence

The coherence problem for multiprocessors and I/O, although similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible— a program running on multiple processors will normally have copies of the same data in several caches. In a coherent multiprocessor, the caches pr ovide both *migration* and *replication* of shared data items.

Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion. This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

Coherent caches also provide replication for shared data that is being simultaneously read, since the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item. Supporting this migration and replication is critical to performance in accessing shared data. Thus, rather than trying to solve the problem by avoiding it in software, small-scale multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.

The protocols to maintain coherence for multiple processors are called *cachecoherence protocols*. Key to implementing a cache-coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols, which use different techniques to track the sharing status, in use:

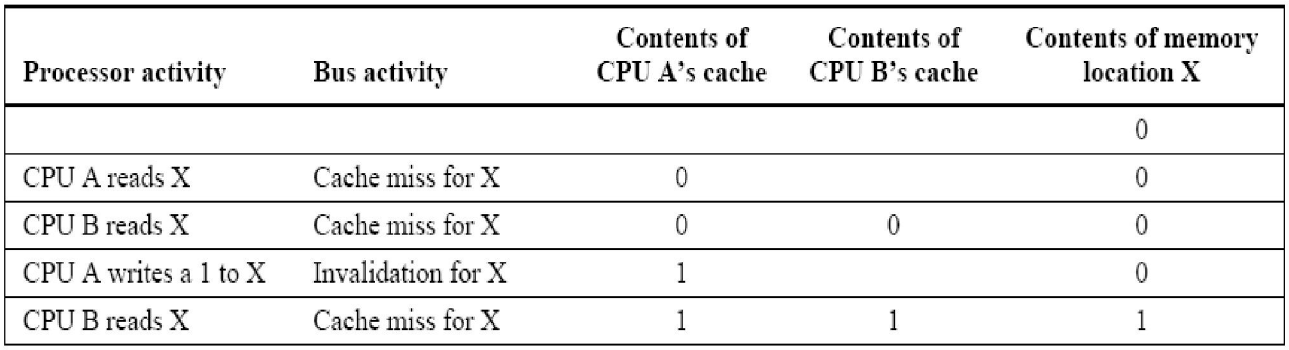
1. *Directory based*—The sharing status of a block of physical memory is kept in just one location, called the *directory*when we discuss scalable shared-memory architecture.
2. *Snooping*—Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared-memory bus, and all cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of a block that is requested on the bus.

Snooping protocols became popular with multiprocessors using microprocessors and caches attached to a single shared memory because these protocols can use a preexisting physical connection—the bus to memory—to interrogate the status of the caches.

Snooping Protocols:

There are two ways to maintain the coherence requirement described in the previous subsection. One method is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a

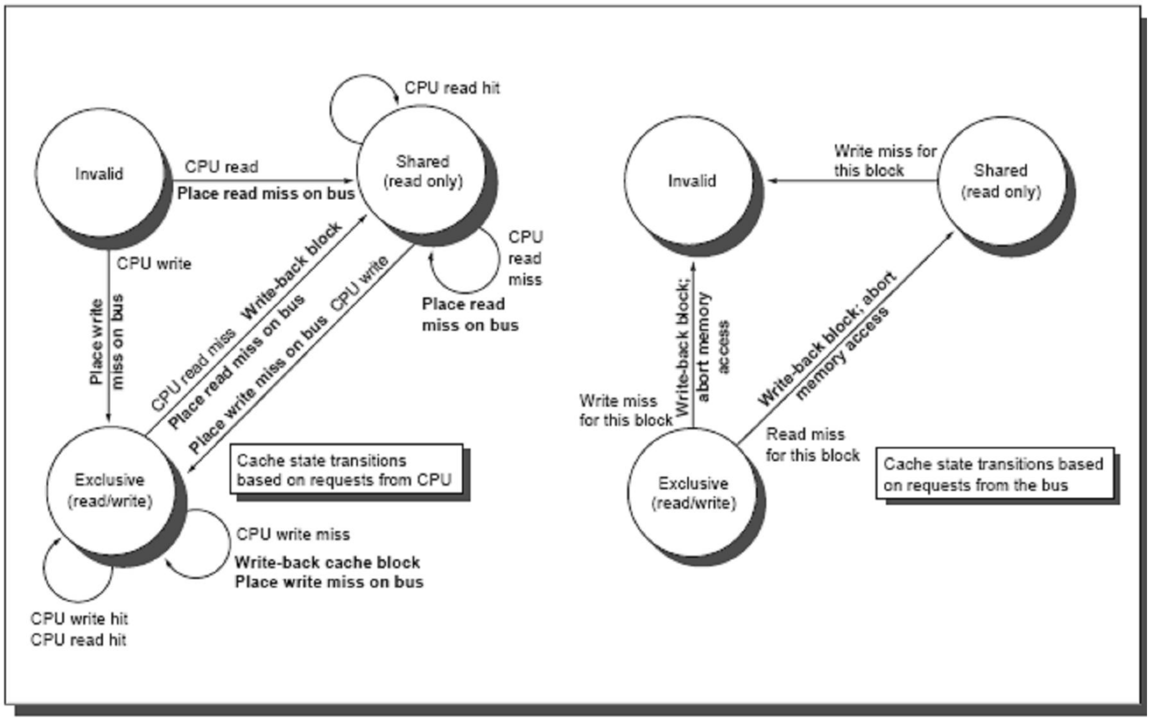
write. It is by far the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.
Figure below shows an example of an invalidation protocol for a snooping bus with write-back caches in action To see how this protocol ensures coherence, consider a write followed by a read by another processor: Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data simultaneously, one of them wins the race (we'll see how we decide who wins shortly), causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol enforces write serialization.



Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

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

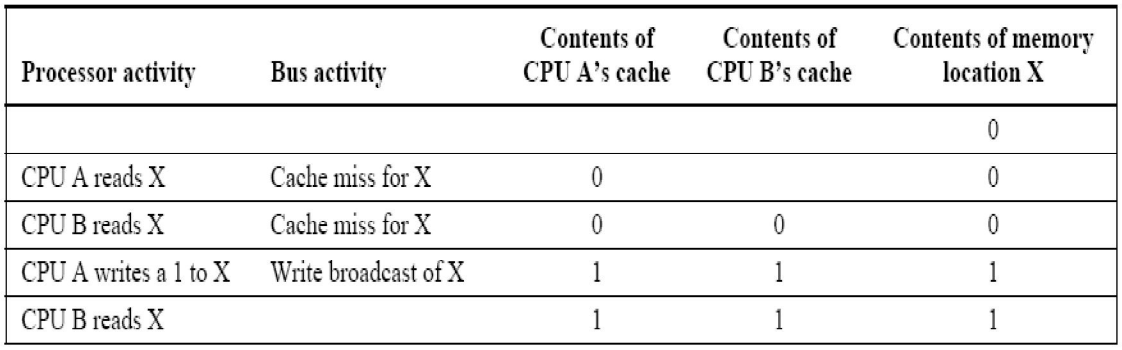
We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, is typical in most protocols and simplifies the protocol.



A write-invalidate, cache-coherence protocol for a write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles with any access permitted by the CPU without a state transition shown in parenthesis under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the CPU associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the CPU does not match the address in the cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state always generates a miss, even if the block is present in the cache, since the block must be made exclusive. Whenever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory provides data on a read miss for

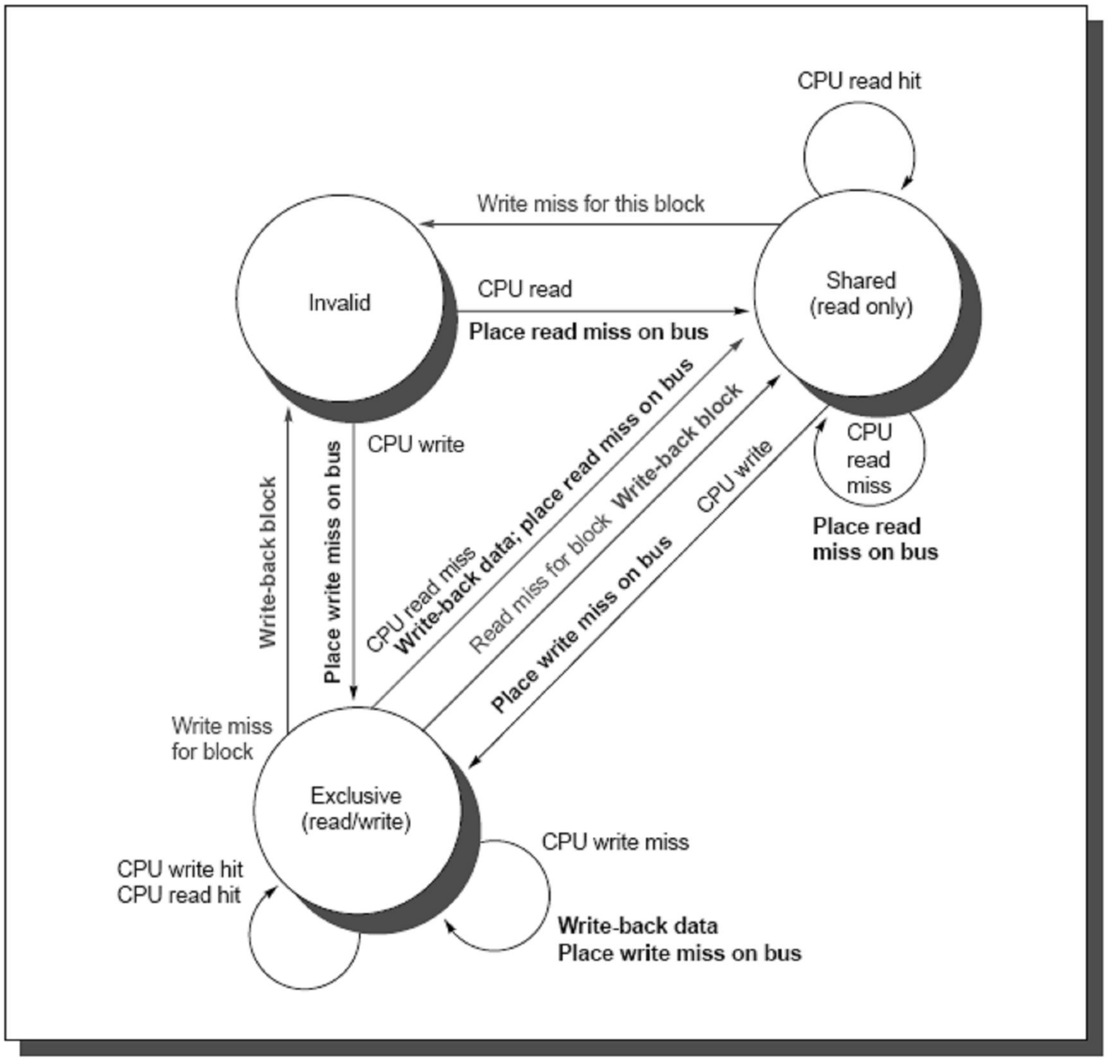
a block that is clean in all caches. In actual implementations, these two sets of state diagrams are combined.
This protocol is somewhat simpler than those in use in existing multiprocessors.

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a *write update* or *write broadcast* protocol. To keep the bandwidth requirements of this protocol under control it is useful to track whether or not a word in the cache is shared— that is, is contained in other caches. If it is not, then there is no need to broadcast or update any other caches. Figure 6.8 shows an example of a write update protocol in operation. In the decade since these protocols were developed, invalidate has emerged as the winner for the vast majority of designs. To understand why, let's look at the qualitative performance differences. The performance differences between write update and write invalidate protocols arise from three characteristics:



Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

An example of a write update or broadcast protocol working on a snooping bus for a single cache
block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0.
The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates
no activity or no copy cached. When CPU A broadcasts the write, both the cache in CPU B and the memory location of X
are updated.



Cache-coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray.

The actions in gray in Figure, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the memory.