**4.5- Write-Update and Write-Back**

 This protocol is similar to the previous one except that instead of writing through to the memory whenever a shared block is updated, memory updates are done only when the block is being replaced. The block states and protocol are summarized in TABLE -11.

TABLE -11 Write-Update Write-Back Protocol

|  |  |
| --- | --- |
| **State** | **Description** |
| Valid Exclusive[VAL-X] | This is the only cache copy and is consistent with global memory. |
| Shared Clean[SH-CLN] | There are multiple cache copies shared. |
| Shared Dirty[SH-DRT] | There are multiple shared cache copies. This is the last one being updated. (Ownership.) |
| Dirty [DIRTY] | This copy is not shared by other caches and has been updated. It is not consistent with global memory. (Ownership.) |
| **Event** | **Action** |
| Read-Hit | Use the local copy from the cache. State does not change. |
| Read-Miss | If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared Clean. If the supplying cache copy was in a Valid Exclusion or Shared Clean, its new state becomes Shared Clean. If the supplying cache copy was in a Dirty or Shared Dirty state, its new state becomes Shared Dirty. |
| Write-Hit | If the sate was Valid Exclusive or Dirty, perform the write locally and set the state to Dirty. If the state is Shared Clean or Shared Dirty, perform update and change state to Shared Dirty. Broadcast the updated block to all other caches. These caches snoop the bus and update their copies and set their state to Shared Clean. |
| Write-Miss | The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update, set the state to Shared Dirty, and broadcast the updated block to all other caches. Other caches snoop the bus, update their copies, and change their state to Shared Clean. If the copy comes from memory, perform the write and set the state to Dirty. |
| Block replacement | If a copy is in a Dirty or Shared Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive, no write back is needed when a block is replaced. |

**Example 6** Consider the shared memory system of Figure -6 and the following operations: (1) P reads X; (2) P updates X; (3) Q reads X; (4) Q updates X; (5) Q reads X; (6) Block X is replaced in Q’s cache; (7) P updates X; (8) Q updates

X. TABLE -12 shows the contents of memory and the two caches after the execution of each operation when Write-Update Write-Back was used for cache coherence.

 The table also shows the state of the block containing X in P’s cache and Q’s cache.

TABLE -12 Example 6 (Write-Update Write-Back)



**5- Directory Based Protocols**

 Owing to the nature of some interconnection networks and the size of the shared memory system, updating or invalidating caches using snoopy protocols might become unpractical. For example, when a multistage network is used to build a large shared memory system, the broadcasting techniques used in the snoopy protocols becomes very expensive. In such situations, coherence commands need to be sent to only those caches that might be affected by an update. This is the idea behind directory-based protocols. Cache coherence protocols that somehow store information on where copies of blocks reside are called directory schemes. A directory is a data structure that maintains information on the processors that share a memory block and on its state. The information maintained in the directory could be either centralized or distributed. A Central directory maintains information about all blocks in a central data structure. While Central directory includes everything in one location, it becomes a bottleneck and suffers from large search time. To alleviate this problem, the same information can be handled in a distributed fashion by allowing each memory module to maintain a separate directory. In a distributed directory, the entry associated with a memory block has only one pointer one of the cache that requested the block.

**5.1- Protocol Categorization**

 A directory entry for each block of data should contain a number of pointers to specify the locations of copies of the block. Each entry might also contain a dirty bit to specify whether or not a unique cache has permission to write this memory block. Most directory-based protocols can be categorized under three categories: full-map directories, limited directories, and chained directories.

**Full-Map Directories** In a full-map setting, each directory entry contains N pointers, where N is the number of processors. Therefore, there could be N cached copies of a particular block shared by all processors. For every memory block, an N-bit vector is maintained, where N equals the number of processors in the shared memory system. Each bit in the vector corresponds to one processor. If the i th bit is set to one, it means that processor i has a copy of this block in its cache. Figure -7 illustrates the fully mapped scheme. In the figure the vector associated with block X in memory indicates that X is in Cache C0 and Cache C2. Clearly the space is not utilized efficiently in this scheme, in particular if not many processors share the same block.

**Limited Directories** Limited directories have a fixed number of pointers per directory entry regardless of the number of processors. Restricting the number of simultaneously cached copies of any block should solve the directory size problem that might exist in full-map directories. Figure -8 illustrates the limited directory scheme. In this example, the number of copies that can be shared is restricted to two

Figure -7 Fully mapped directory.

This is why the vector associated with block X in memory has only two locations. The vector indicates that X is in Cache C0 and Cache C2.



Figure -8 Limited directory (maximum sharing = 2).

**Chained Directories** Chained directories emulate full-map by distributing the directory among the caches. They are designed to solve the directory size problem without restricting the number of shared block copies. Chained directories keep track of shared copies of a particular block by maintaining a chain of directory pointers.

 Figure -9 shows that the directory entry associated with X has a pointer to Cache C2, which in turn has a pointer to Cache C0. That is, block X exists in the two Caches C0 and Cache C2. The pointer from Cache C0 is pointing to terminator (CT), indicating the end of the list.



Figure -9 Chained directory.

**5.2- Invalidate Protocols**

**Centralized Directory Invalidate** When a write request is issued, the central directory is used to determine which processors have a copy of the block.

 Invalidating signals and a pointer to the requesting processor are forwarded to all processors that have a copy of the block. Each invalidated cache sends an acknowledgment to the requesting processor. After the invalidation is complete, only the writing processor will have a cache with a copy of the block. Figure -10 shows a write-miss request from Cache C3. Upon receiving the request, the memory sends invalidating signals and a pointer to the Cache C3 to Cache C0 and Cache C2. These caches invalidate themselves and send invalidation acknowledgment to Cache C3. After the invalidation is done, Cache C3 will have exclusive read-write access to X.



Figure -10 Centralized directory invalidation.

**Scalable Coherent Interface (SCI)** The scalable coherent interface (SCI) protocols are based on a doubly linked list of distributed directories. Each cached block is entered into a list of processors sharing that block. For every block address, the memory and cache entries have additional tag bits. Part of the memory tag identifies the first processor in the sharing list (the head). Part of each cache tag identifies the previous and following sharing list entries. Without counting the number of bits needed in the local caches for the pointers, the directory size in memory equals the number of memory blocks times log2 (number of caches).

 Initially memory is in the uncached state and cached copies are invalid. A read request is directed from a processor to the memory controller. The requested data is returned to the requester’s cache and its entry state is changed from invalid to the head state. This changes the memory state from uncached to cached. When a new requester directs its read request to memory, the memory returns a pointer to the head. A cache-to-cache read request (called Prepend) is sent from the requester to the head cache. On receiving the request, the head cache sets its backward pointer to point to the requester’s cache. The requested data is returned to the requester’s cache and its entry state is changed to the head state. The head of the list has the authority to purge other entries in the list to obtain an exclusive (read-write) entry. The initial transaction to the second sharing list entry purges that entry and returns its forward pointer. The forward pointer is used to purge the next entry and so on. Entries can also delete themselves from the list when they are needed to cache other block addresses. Figure -11 shows the sharing list addition and removal operations in SCI.

Figure -11 Scalable coherent interface (a) sharing list addition (SCI); and (b) head purging other entries (SCI).

**Stanford Distributed Directory (SDD)** The Stanford distributed directory (SDD) protocol is based on a singly linked list of distributed directories. Similar to the SCI protocol, memory points to the head of the sharing list. Each processor points only to its predecessor. The sharing list additions and removals are handled differently from the SCI protocol.

 On a read-miss, a new requester sends a read-miss message to memory. The memory updates its head pointers to point to the requester and send a read-missforward signal to the old head. On receiving the request, the old head returns the requested data along with its address as a read-miss-reply. When the reply is received, at the requester’s cache, the data is copied and the pointer is made to point to the old head.

 On a write-miss, a requester sends a write-miss message to memory. The memory updates its head pointers to point to the requester and sends a write-miss-forward signal to the old head. The old head invalidates itself, returns the requested data as a write-miss-reply-data signal, and send a write-miss-forward to the next cache in the list. When the next cache receives the write-miss-forward signal, it invalidates itself and sends a write-miss-forward to the next cache in the list. When the write-miss-forward signal is received by the tail or by a cache that no longer has a copy of the block, a write-miss-reply is sent to the requester. The write is complete when the requester receives both write-miss-reply-data and write-miss-reply. Figure -12 shows the sharing list addition and removal operations in SDD.

Figure -12 Stanford distributed directory (a) sharing list addition (SDD); and (b) write miss sharing list removal (SDD).

**6- Shared Memory Programming**

 Shared memory parallel programming is perhaps the easiest model to understand because of its similarity with operating systems programming and general multiprogramming.

 Shared memory programming is done through some extensions to existing programming languages, operating systems, and code libraries. In a shared memory parallel program, there must exist three main programming constructs: (1) task creation, (2) communication, and (3) synchronization.

**6.1- Task Creation**

 At the large-grained level, a shared memory system can provide traditional timesharing. Each time a new process is initiated, idle processors are supplied to run the new process. If the system is loaded, the processor with least amount of work is assigned the new process. These large-grained processes are often called heavy weight tasks because of their high overhead. A heavy weight task in a multitasking system like UNIX consists of page tables, memory, and file description in addition to program code and data. These tasks are created in UNIX by invocation of fork, exec, and other related UNIX commands. This level is best suited for heterogeneous tasks.

 At the fine-grained level, lightweight processes makes parallelism within a single application practical, where it is best suited for homogeneous tasks. At this level, an application is a series of fork-join constructs. This pattern of task creation is called the supervisor–workers model, as shown in Figure -13.



Figure -13 Supervisor–workers model used in most parallel applications on shared memory systems.

**6.2- Communication**

 In general, the address space on an executing process has three segments called the text, data, and stack. The text is where the binary code to be executed is stored; the data segment is where the program’s data are stored; and the stack is where activation records and dynamic data are stored. The data and stack segments expand and contract as the program executes. Therefore, a gap is purposely left in between the data and stack segments.

 Serial processes are assumed to be mutually independent and do not share addresses. The code of each serial process is allowed to access data in its own data and stack segments only. A parallel process is similar to the serial process plus an additional shared data segment. This shared area is allowed to grow and is placed in the gap between private data and stack segments. Figure -14 shows the difference between a serial process and a parallel process.



Figure -14 Serial process vs. parallel process.

 Communication among parallel processes can be performed by writing to and reading from shared variables in the shared data segments as shown in Figure -15.

**6.3- Synchronization**

 Synchronization is needed to protect shared variables by ensuring that they are accessed by only one process at a given time (mutual exclusion). They can also be used to coordinate the execution of parallel processes and synchronize at certain points in execution. There are two main synchronization constructs in shared memory systems: (1) locks and (2) barriers. Figure -16a shows three parallel processes using locks to ensure mutual exclusion. Process P2 has to wait until P1 unlocks the critical section; similarly P3 has to wait until P2 issues the unlock statement. In Figure -16b, P3 and P1 reach their barrier statement before P2, and they have to wait until P2 reaches its barrier. When all three reach the barrier statement, they all can proceed.



Figure -15 Two parallel processes communicate using the shared data segment.



Figure -16 Locks and barriers.