Multiprocessors and Thread-Level Parallelism

Multithreading

- Increasing performance by ILP has the great advantage that it is reasonable transparent to the programmer, ILP can be quite limited or hard to exploit in some applications
- When the processor is stalled waiting on a cache miss, the utilization of functional units drops
- Other forms of parallelism – multithreading
- *Multithreading* allows multiple threads to share the functional units of a single processor in an overlapping fashion.
- General method to exploit thread-level parallelism (TLP) is with a multiprocessor that has multiple independent threads operating at once and in parallel.
- Multithreading shares processor core among a set of threads duplicating registers and program counter.

Multithreading

- For sharing, the processor must duplicate the independent state of each thread
- In addition the hardware must support the ability to change to a different thread relatively quickly
- Three main hardware approaches to multithreading
  - Fine-grained multithreading
  - Coarse-grained multithreading
  - Simultaneous multithreading

Fine-grained multithreading

- It switches between threads on each instruction, causing the execution of multiple threads to be interleaved.
- Round-robin fashion – skipping any thread that are stalled at that time
- CPU must be able to switch threads on every clock cycle
- It can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls
- It slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads

Coarse-grained multithreading

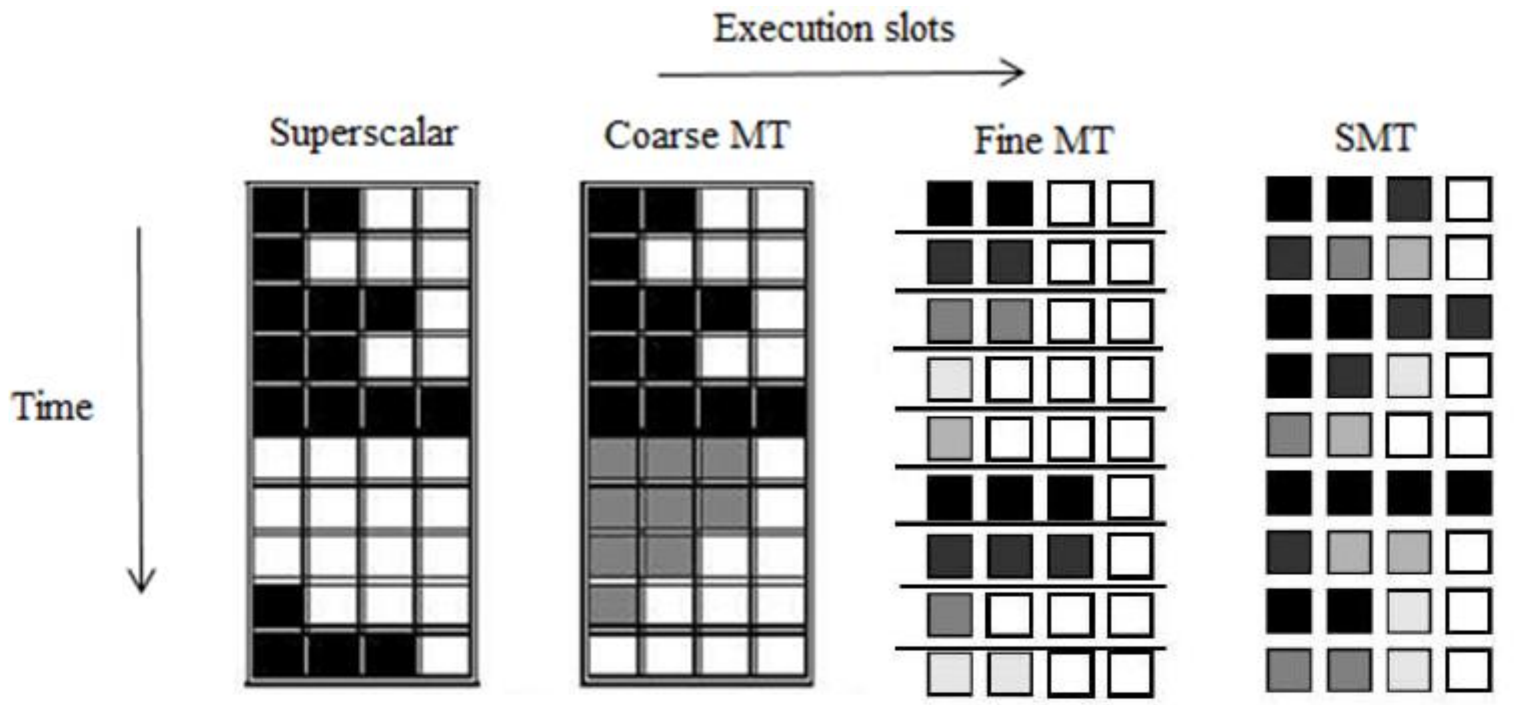
- Switches threads only on costly stalls, such as level two or three cache misses
- Relieves the need to have thread switching
- likely to slow the processor down, since the instructions form other threads will only be issued when a thread encounters a costly stall
- It is limited in its ability to overcome throughput losses, especially from shorter stalls
- There is a pipeline start up cost, because a CPU with CGM issues instruction form a single thread, when a stall occurs, the pipeline must be emptied. Next thread must fill the pipeline before instructions will be able to complete.
- More useful in reducing penalty of high cost stalls where pipeline fill is negligible compared to stall time

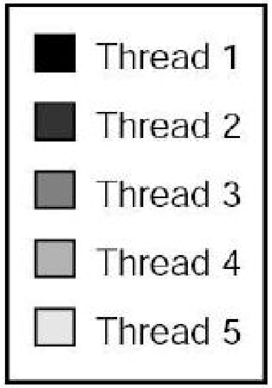
Simultaneous multithreading

- Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple issue, dynamically-scheduled processor to exploit TLP at the same time it exploits ILP.
- Modern multiple- issue processors often have more functional unit parallelism available than a single thread.
- With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Superscalar Processor

- Superscalar with no multithreading
- Superscalar with coarse-grained multithreading
- Superscalar with fine-grained multithreading
- Superscalar with Simultaneous multithreading


Execution slots
Superscalar
Coarse MT
Fine MT
SMT
Time

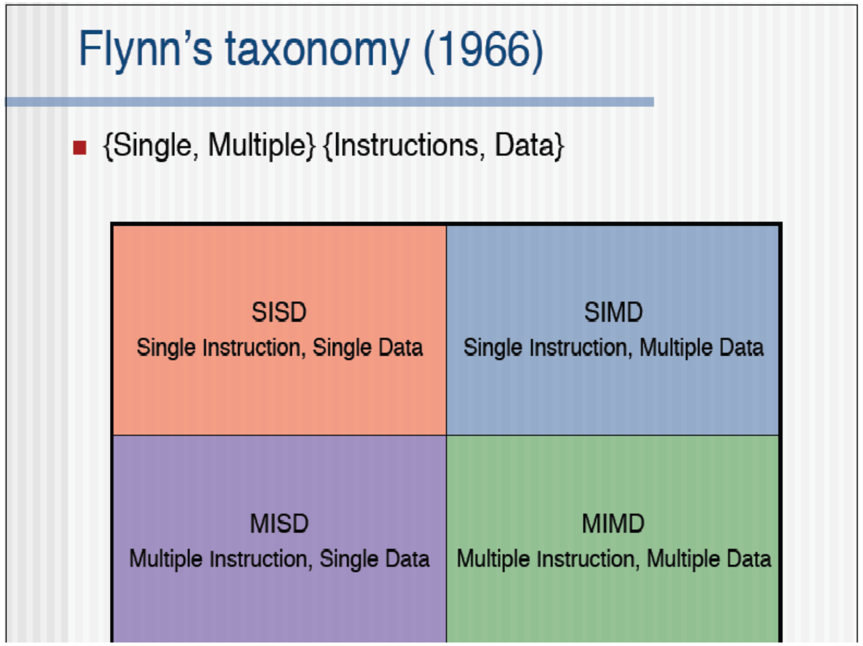

Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

Multiprocessing

- Other than ILP, the only scalable and general–purpose way to increase performance faster is through multiprocessing
- Increased importance of miultiprocessing
  - Growing interest in high-end servers as cloud computing and software-as-a-service become more important
  - Growth in data-intensive applications driven by the availability of massive amounts of data on the internet.
- Multiprocessors exploit thread level parallelism through two different software models
  - Tightly coupled set of threads collaborating on a single task *parallel processing*
  - Execution of multiple, relatively independent processes from one or more users *request level parallelism – such as database responding to queries multiprogramming*

Multiprocessor Architecture

- multiple processors increase performance
- Flynn's proposed a simple model of categorizing all computers
- looked at the parallelism in the instruction and data streams and placed all the computers into one of the 4 categories: (Flynn's Taxonomy)



Flynn's taxonomy (1966)

- {Single, Multiple} {Instructions, Data}

SISD Single Instruction, Single Data	SIMD Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

- SISD
  - uniprocessor
- SIMD
  - The same instruction is executed by multiple processors using different data streams
  - Exploit data level parallelism by applying the same operations to multiple items of data in parallel
  - Each processor has its own data memory, but there is a single instruction memory and control processor, which fetches and dispatches the instruction
  - The multimedia extension, vector architecture are the largest class of SIMD architectures.

– MISD

- No commercial multiprocessor of this type has been built to date

– MIMD

- Each processor fetches its own instruction and operated on its own data
- They exploit TLP, bz. multiple threads operate in parallel
- In general, TLP is more flexible than DLP
- They offers flexibility. With the correct hardware and software, they can function as single user multiprocessors focusing on high performance for one application, as multi programmed multiprocessors running many tasks simultaneously, or as some combination of these functions.

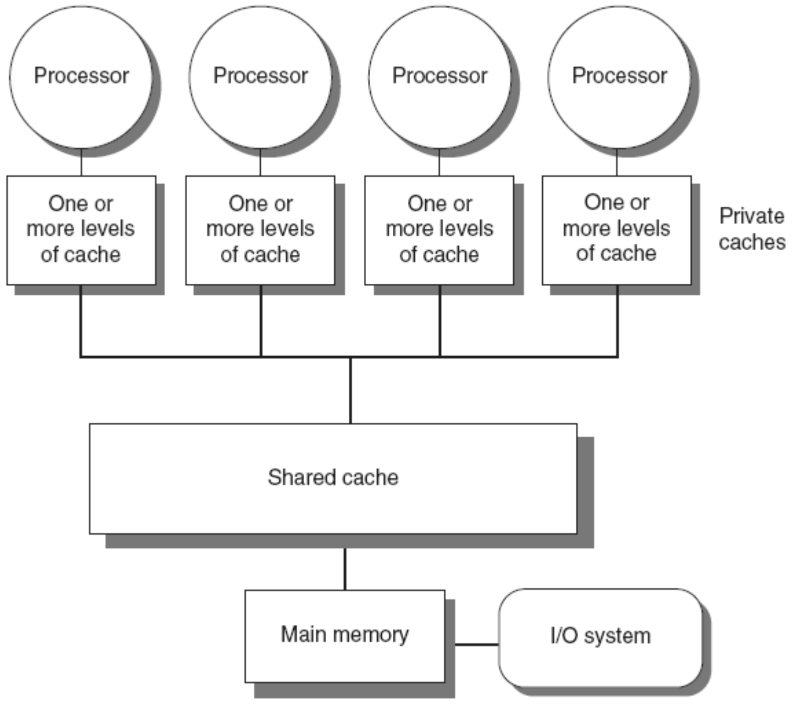
- With MIMD, each processor is executing its own instruction stream or process.
- A process is a segment of code that may be run independently
- The state of the process contains all the information necessary to execute that program on a processor
- It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space
- When multiple process share code and data in this way, they are often called *threads*.
- To take advantage of an MIMD multiprocessors with *n* processors, we must have at least *n* threads or processes to execute.

- The important qualitative distinction of TLP from ILP is that TLP 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.
- Threads can also be used to exploit DLP, but with higher overhead
- The grain size (the amount of computation assigned to a thread) must be sufficiently large to exploit the parallelism efficiently.

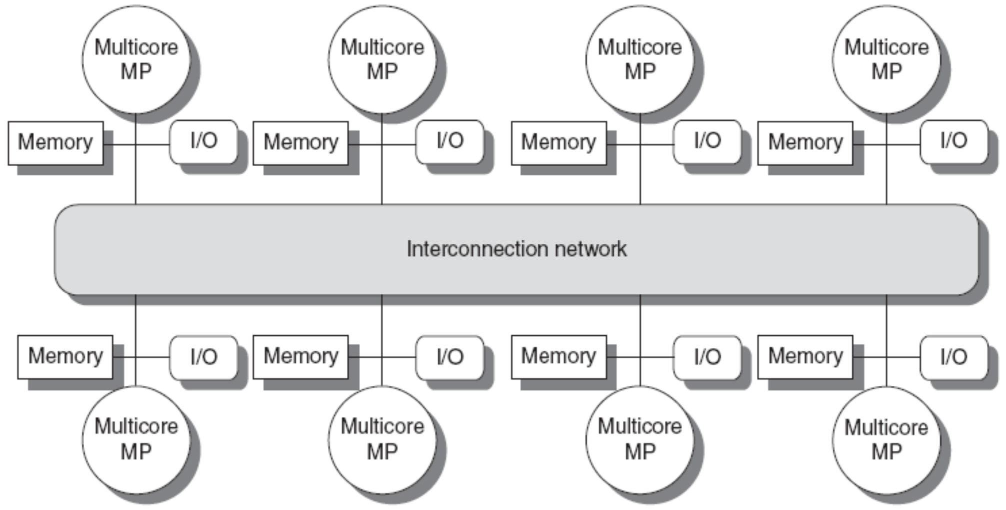
MIMD

- MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy
  - Centralized shared memory architecture
    - Small number of cores
    - Share single memory with uniform memory latency
    - Symmetric relationship to all processors – uniform access time – often called as Symmetric multiprocessors (SMPs)
  - Distributed shared memory architecture
    - Physically distributed memory
    - Non-uniform memory access/latency (NUMA)
    - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks

MIMD


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Centralized shared memory



Distributed shared memory

Distributed shared memory

- Most of the access are to the local memory in the node
- Reduces latency for access to the memory
- Scales memory bandwidth

Models for communication and memory architecture

- A large multi processor must use multiple memories that are physically distributed with the processors
- There are two alternative architectural approaches that differ in the method used for communicating data among processors:

Distributed shared-memory (DSM) architecture

- Communication occurs through a shared address space, as it does in a symmetric shared-memory architecture
- The physically separate memories can be addressed as one logically shared address space
- Memory reference can be made by any processor to any memory location
- Shared-memory refers to the shared address space
- The same physical address on two processors refers to the same location in memory
- Shared memory does not mean that there is a single centralized memory
- In contrast to symmetric shared memory multiprocessors which is also know as UMAs , the DSM multiprocessors are also known as NUMAs, since the access time depends on the location of a data word in memory

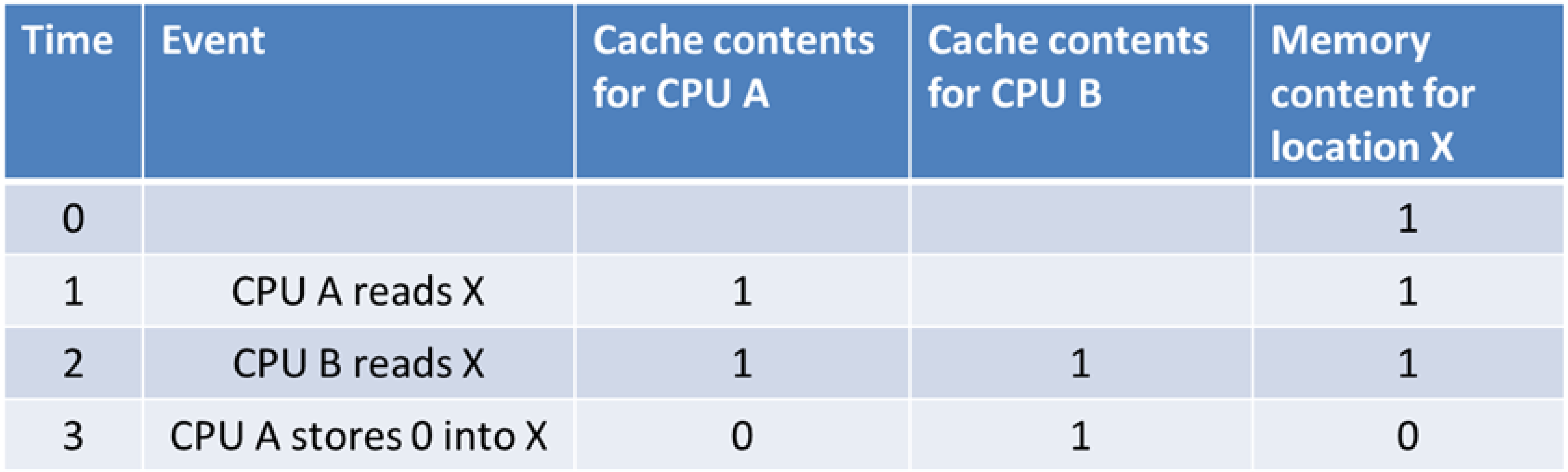
Centralized Shared-Memory Architectures

- Use of large, multilevel caches substantially reduce the memory bandwidth demands of a processor
- If memory bandwidth is reduced, multiple processors may be able to share the same memory
- Because of the small size of the processor and the significant reduction in bus bandwidth achieved by large caches, symmetric processors were extremely cost effective , provided that a sufficient amount of memory bandwidth existed.
- Symmetric shared-memory machines usually support the caching of both shared and private data

- Private data are used by a single processor
- Shared data are used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data
- When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required (uniprocessor)
- When shared data are cached, the shared value may be replicated in multiple caches
- In addition to reduction in access latency and required memory band width, replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously
- Caching of shared data introduces a new problem: ***cache coherence***

Multiprocessor Cache Coherence

- Caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which with out any additional precautions, could end up seeing two different values – *cache coherence problem*



Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory content 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

- Memory system is coherent if any read of a data item returns the most recently written value of that data item.

- Two different aspects of memory system
  - ***Coherence***, defines what values can be returned by a read
  - ***Consistency***, determines when a written value will be returned by a read

- A memory system is coherent if:
  - 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. *preserves program order*
  - A read by a processor to location X that follows a write by another processor to X return the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses. *coherence*
  - Write 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. If the value 1 and 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1. *serialization*

- Question of when a written value will be seen is also important
- 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
- Make following assumptions:
  - A write does not complete until all processors have seen the effect of that write
  - The processor does not change the order of any write w.r.t. any other memory access
- These assumptions means that if a processor write location A followed by B, any processor that sees the new value of B must also see the new value of A
- These restrictions allow the processor to reorder the reads, but forces the processor to finish a write in program order

Basic Schemes for Enforcing Coherence

- A program running on multi processors will have copies of the same data in several caches.
- In a coherent multi processor, the caches provide both migration and replication of shared data items.
- Coherent caches provides migration - data item can be moved to a local cache and used there in a transparent fashion, reducing both the latency and the BW demand on the shared data
- Coherent caches also provide replication for shared data that are being simultaneously read, since the caches make a copy of the data item in the local cache, reducing both latency and contention for a read shared data item.
- Supporting this migration and replication is critical to performance in accessing shared data
- Small scale multi processors adopt a hardware solution by introducing a protocol to maintain coherent caches

Basic Schemes for Enforcing Coherence

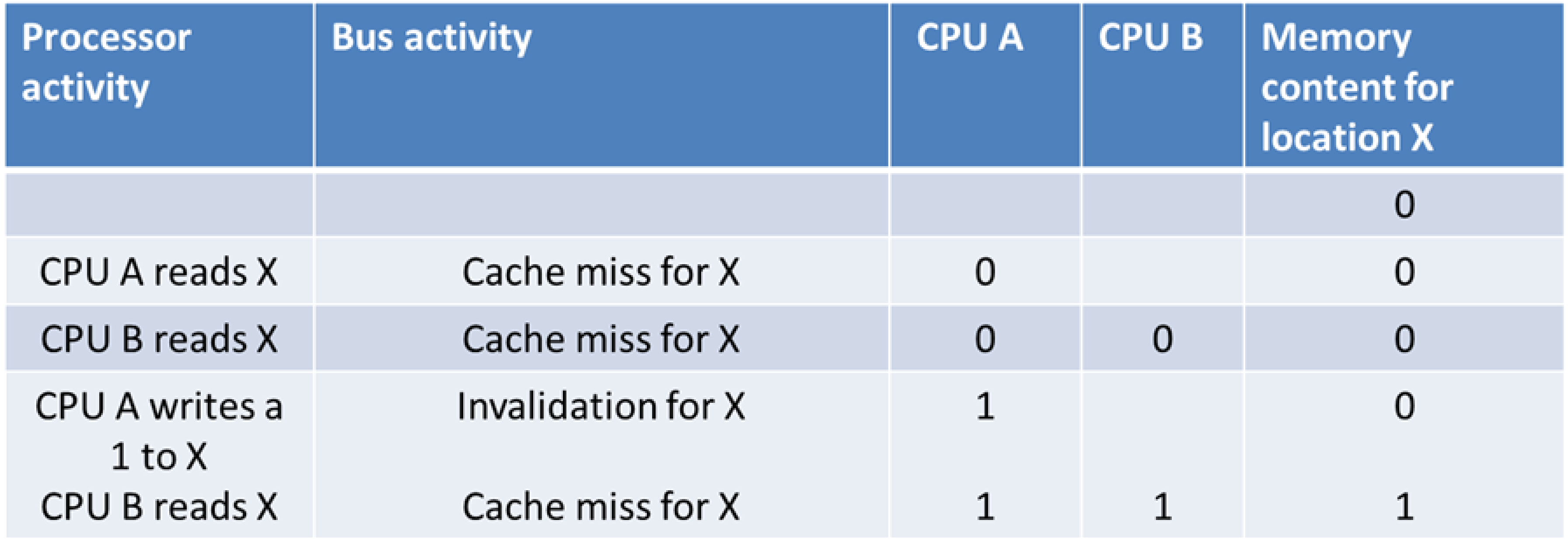
- The protocol to maintain coherence for multiple processors are called cache coherence protocols.
- Key to implementing a cache coherence protocol is tracking the state of any sharing of data block.

- There are two classes of protocols:
  - Directory based - the sharing status of a block of physical memory is kept in just one location, called the directory
  - 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, but no centralized status is kept. These caches are all accessible via some broadcast medium and 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

- Snooping Protocols
  - Write invalidate protocol
  - Write update or write broadcast protocol

Snooping Protocols

- Write invalidate Protocol – to ensure that a processor has exclusive access to a data item before it writes that item because it invalidates other copies on a write. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs. All other cache copies of the item are invalidated.



Processor activity	Bus activity	CPU A	CPU B	Memory content for 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

Snooping Protocols

- 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.
- When the read occurs, it misses in the cache and is forced to fetch the new copy of the data
- For a write, prevent any other processor from being able to write simultaneously
- If two processors try to write the same data simultaneously, one of them will win, 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

Snooping Protocols

- Write update or write broadcast protocol - update all the cached copies of a data item when that item is written.
- A write update protocol must broadcast all writes to shared cache lines, it consumes considerably more bandwidth.
- For this reason, all recent multiprocessors have opted to implement a write invalidate protocol.

Basic Implementation Techniques

- The key to implementing an invalidate protocol is the use of the bus, or another broadcast medium, to perform invalidates.
- To perform an invalidate, the processor acquires bus access and broadcasts the address to be invalidated on the bus
- The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated
- When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation
- If two processors attempt to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus

- The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated
- If the processors were attempting to write to the same block, the serialization enforced by the bus also serializes their writes
- All coherence schemes require some sort of serializing access to the communication medium or another shared structure
- In addition to invalidating outstanding copies of a cache block that is being written into, it is required to locate a data item when a cache miss occurs
- In a write-through cache, it is easy to find the recent value of the data item, since all written data are always sent to the memory, from which most recent value of data can be fetched.
- In a design with adequate memory bandwidth to support the write traffic, using write-through simplifies the implementation of cache coherence

- For a write-back cache, finding the most recent value is harder, since the most recent value can be in a cache rather than in memory
- Write-back cache can use the same snooping scheme both for cache miss and for writes
- Each processor snoops every address placed on the bus
- If a processor finds that it has a dirty copy of the requested cache, it provides that cache block in response to the read request and causes the memory access to be aborted.
- The additional complexity comes from having to retrieve the cache block from the processor’s cache, which often take longer time than retrieving form shared memory if the processor are in separate chips.
- Since write-back caches generate lower requirements for memory bandwidth, they support larger number of faster processors and used in most multiprocessors.

- For a write-back cache, finding the most recent value is harder, since the most recent value can be in a cache rather than in memory
- Write-back cache can use the same snooping scheme both for cache miss and for writes
- Each processor snoops every address placed on the bus
- If a processor finds that it has a dirty copy of the requested cache, it provides that cache block in response to the read request and causes the memory access to be aborted.
- The additional complexity comes from having to retrieve the cache block from the processor’s cache, which often take longer time than retrieving form shared memory if the processor are in separate chips.
- Since write-back caches generate lower requirements for memory bandwidth, they support larger number of faster processors and used in most multiprocessors.