Shared Memory Multiprocessors

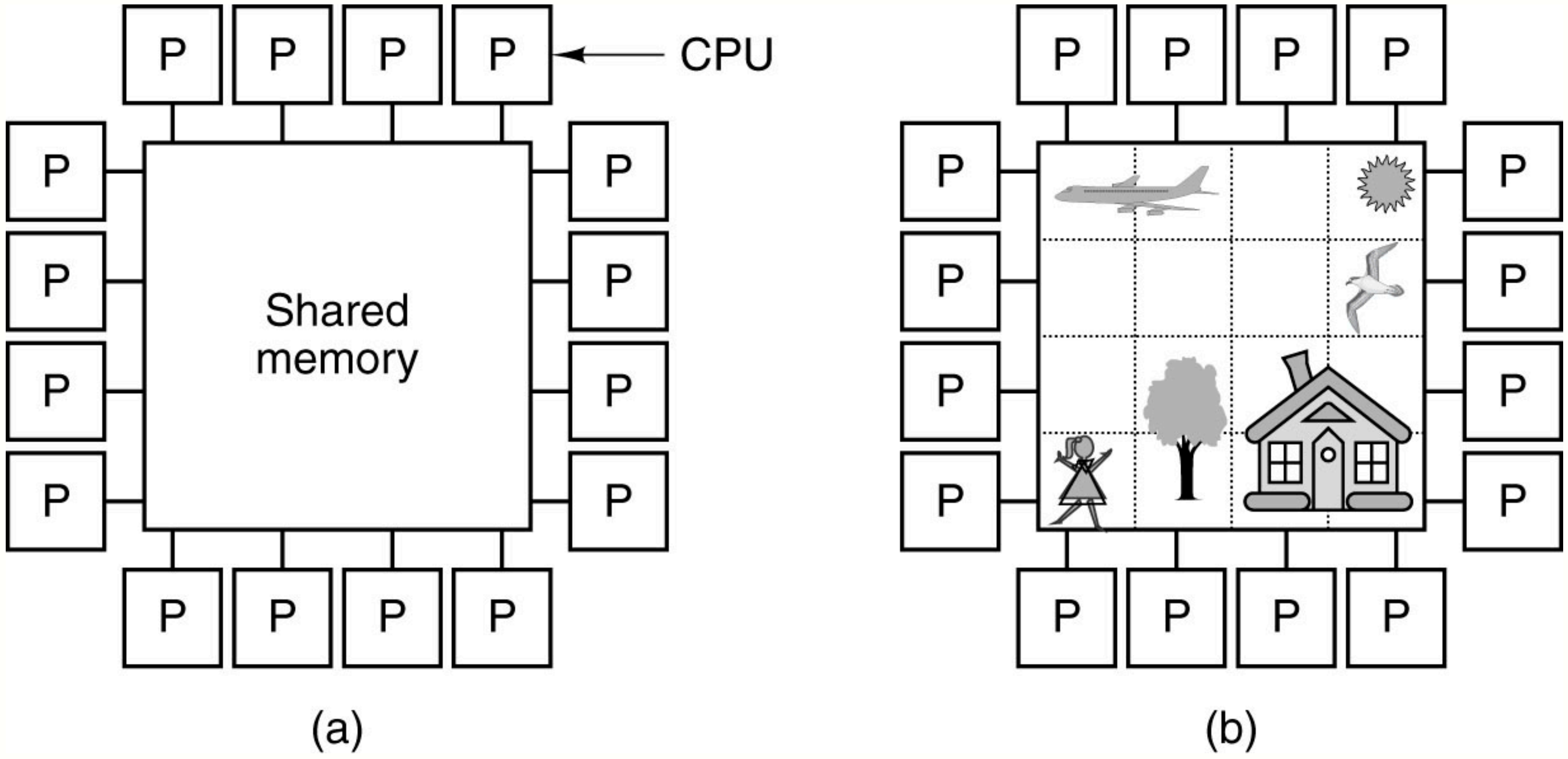
- Introduction
- UMA systems
- NUMA systems
- COMA systems
- Cache coherency
- Process synchronization
- Models of memory consistency

Shared memory multiprocessors

- A system with multiple CPUs "sharing" the same main memory is called **multiprocessor**.
- In a multiprocessor system all processes on the various CPUs share a ***unique logical address space***, which is mapped on a physical memory that can be distributed among the processors.
- Each process can read and write a data item simply using *load* and *store* operations, and *process communication* is through ***shared memory***.
- It is the hardware that makes all CPUs access and use the same main memory.

Shared memory multiprocessors

- This is an architectural model simple and easy to use for programming; it can be applied to a wide variety of problems that can be modeled as a set of tasks, to be executed in parallel (at least partially) (Tanenbaum, Fig. 8.17).



Shared memory multiprocessors

- Since all CPUs share the address space, only a single instance of the operating system is required.
- When a process terminates or goes into a wait state for whichever reason, the O.S. can look in the process table (more precisely, in the ready processes queue) for another process to be dispatched to the idle CPU.
- On the contrary, in systems with *no shared memory*, each CPU must have its own copy of the operating system, and processes can only communicate through *message passing*.
- The basic issue in shared memory multiprocessor systems is memory itself, since the larger the number of processors involved, the more difficult to work on memory efficiently.

Shared memory multiprocessors

- All modern OS (Windows, Solaris, Linux, MacOS) support **symmetric multiprocessing**, (**SMP**), with a scheduler running on every processor (a simplified description, of course).
- "ready to run" processes can be inserted into a single queue, that can be accessed by every scheduler, alternatively there can be a "ready to run" queue for each processor.
- When a scheduler is activated in a processor, it chooses one of the "ready to run" processes and dispatches it on its processor (with a single queue, things are somewhat more difficult, can you guess why?)

Shared memory multiprocessors

- A distinct feature in multiprocessor systems is **load balancing**.
- It is useless having many CPUs in a system, if processes are not distributed evenly among the cores.
- With a single “ready-to-run” queue, load balancing is usually automatic: if a processor is idle, its scheduler will pick a process from the shared queue and will start it on that processor.

Shared memory multiprocessors

- Modern OSs designed for SMP often have a separate queue for each processor (to avoid the problems associated with a single queue).
- There is an explicit mechanism for load balancing, by which a process on the wait list of an overloaded processor is moved to the queue of another, less loaded processor.
- As an example, SMP Linux activates its load balancing scheme every 200 ms, and whenever a processor queue empties.

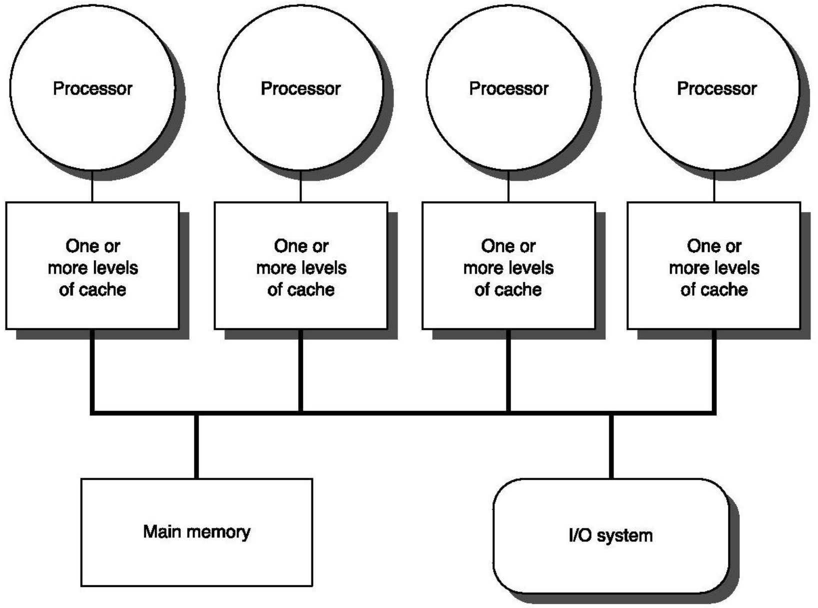
Shared memory multiprocessors

- Migrating a process to a different processor can be costly when each core has a private cache (can you guess why?).
- This is why some OSs, such as Linux, offer a system call to specify that a process is tied to the processor, independently of the processors load.
- There are three classes of multiprocessors, according to the way each CPU *sees* main memory:

Shared memory multiprocessors

1. **Uniform Memory Access (UMA)**: the name of this type of architecture hints to the fact that all processors share a unique centralized primary memory, so *each CPU has the same memory access time.*

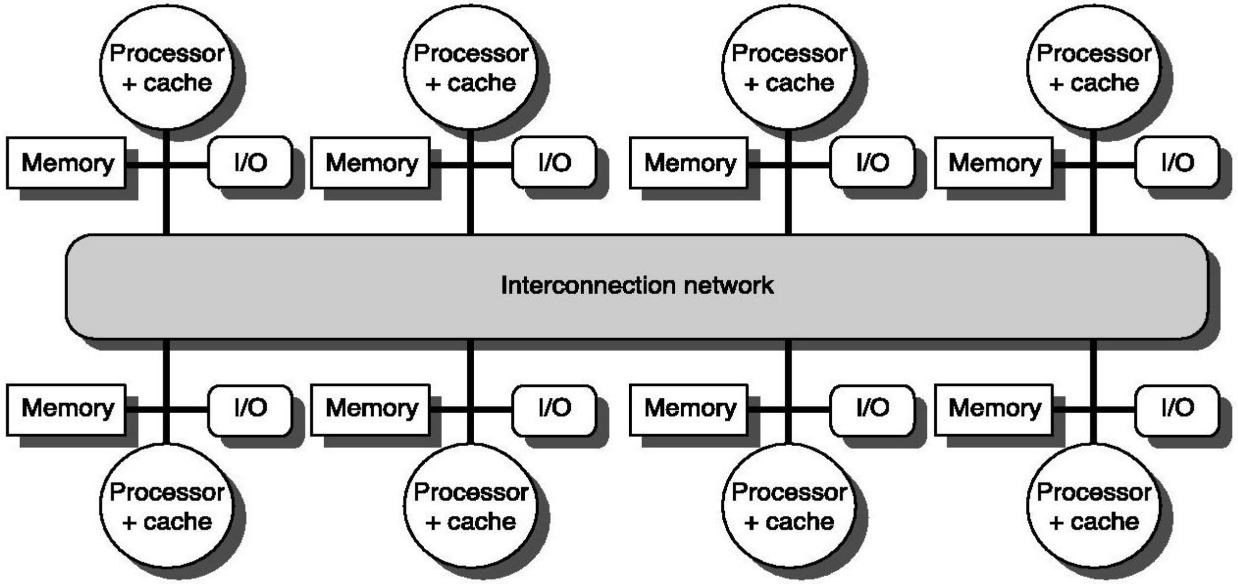
- Owing to this architecture, these systems are also called *Symmetric Shared-memory Multiprocessors (SMP)* (Hennessy-patterson, Fig. 6.1)


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
Main memory
I/O system

Shared memory multiprocessors

2. **Non Uniform Memory Access (NUMA)**: these systems have a shared logical address space, but physical memory is *distributed* among CPUs, so *that access time to data depends on data position, in local or in a remote memory* (thus the NUMA denomination)

- These systems are also called *Distributed Shared Memory (DSM)* architectures (Hennessy-Patterson, Fig. 6.2)



Shared memory multiprocessors

3. **Cache Only Memory Access (COMA)**: data have no specific "permanent" location (no specific memory address) where they stay and whence they can be read (copied into local caches) and/or modified (first in the cache and then updated at their "permanent" location).

- Data can migrate and/or can be replicated in the various memory banks of the central main memory.

UMA multiprocessors

- The *simplest* multiprocessor system has a *single bus* to which connect at least two CPUs and a memory (shared among all processors).
- When a CPU wants to access a memory location, it checks if the bus is free, then it sends the request to the memory interface module and waits for the requested data to be available on the bus.
- **Multicore processors** are small UMA multiprocessor systems, where the first shared cache (L2 or L3) is actually the communication channel.
- Shared memory can quickly become a bottleneck for system performances, since all processors must synchronize on the single bus and memory access.

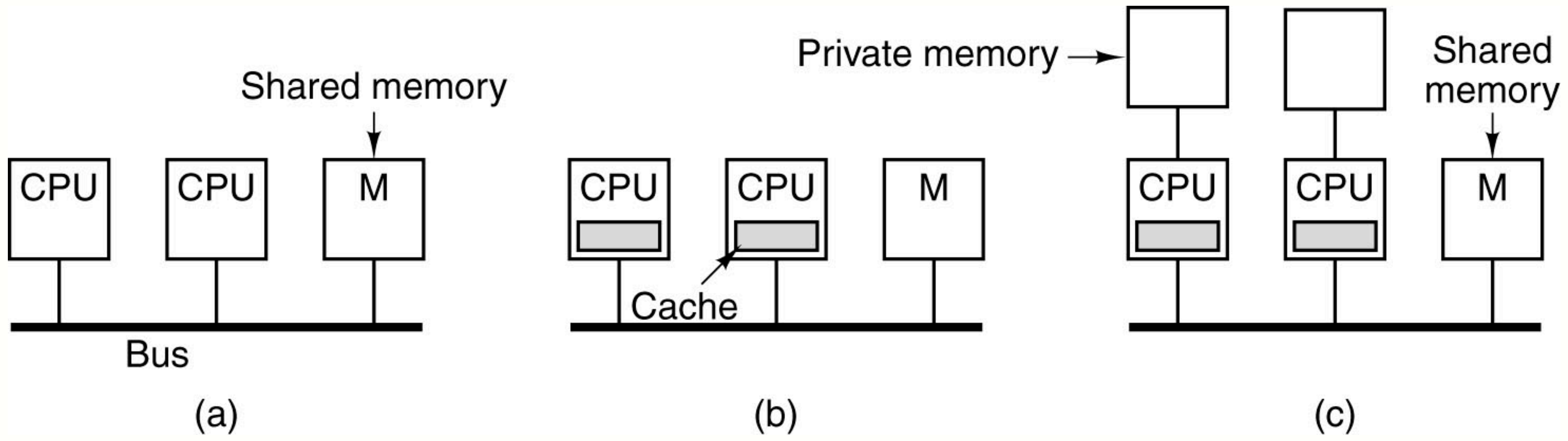
UMA multiprocessors

- *Larger* multiprocessor systems (>32 CPUs) cannot use a *single bus* to interconnet CPUs to memory modules, because bus contention becomes un-manegeable.

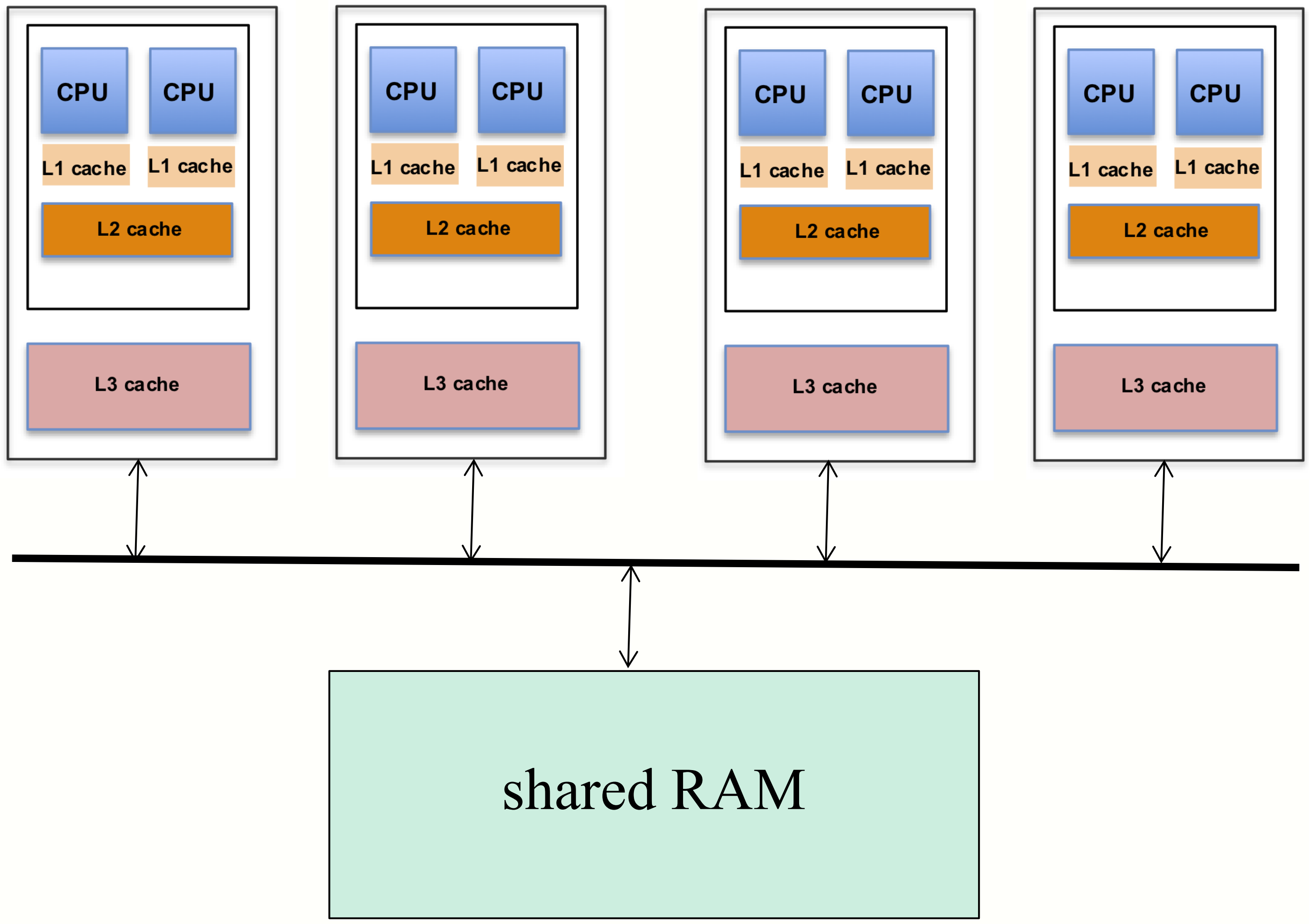
- CPU – memory is realized through an interconnection network (in jargon “fabric”).

UMA multiprocessors

- Caches local to each CPU alleviate the problem, furthermore each processor can be equipped with a private memory to store data of computations that need not be shared by other processors. Traffic to/from shared memory can reduce considerably (Tanenbaum, Fig. 8.24)



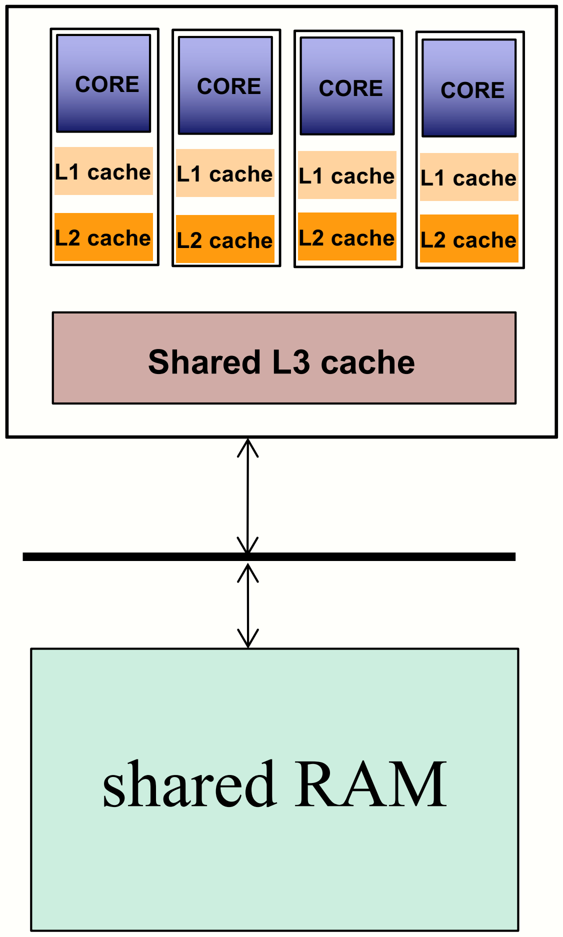
UMA multiprocessors

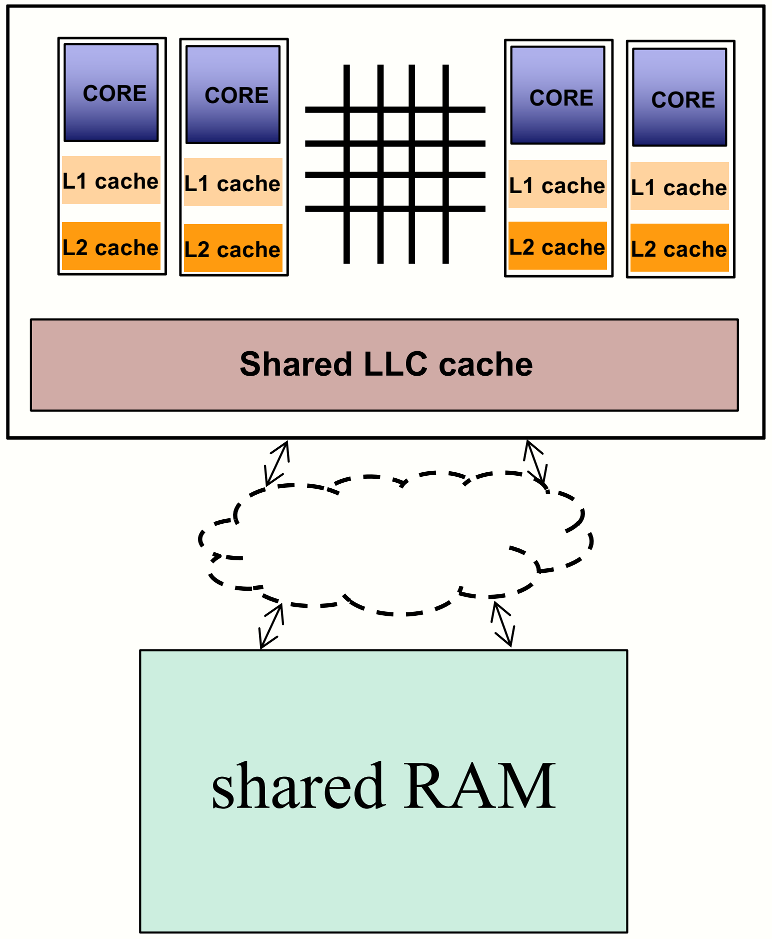


UMA multicores - manycores

multicores: 2 ÷ 22

manycores: ~ 70





Caches and memory in multiprocessors

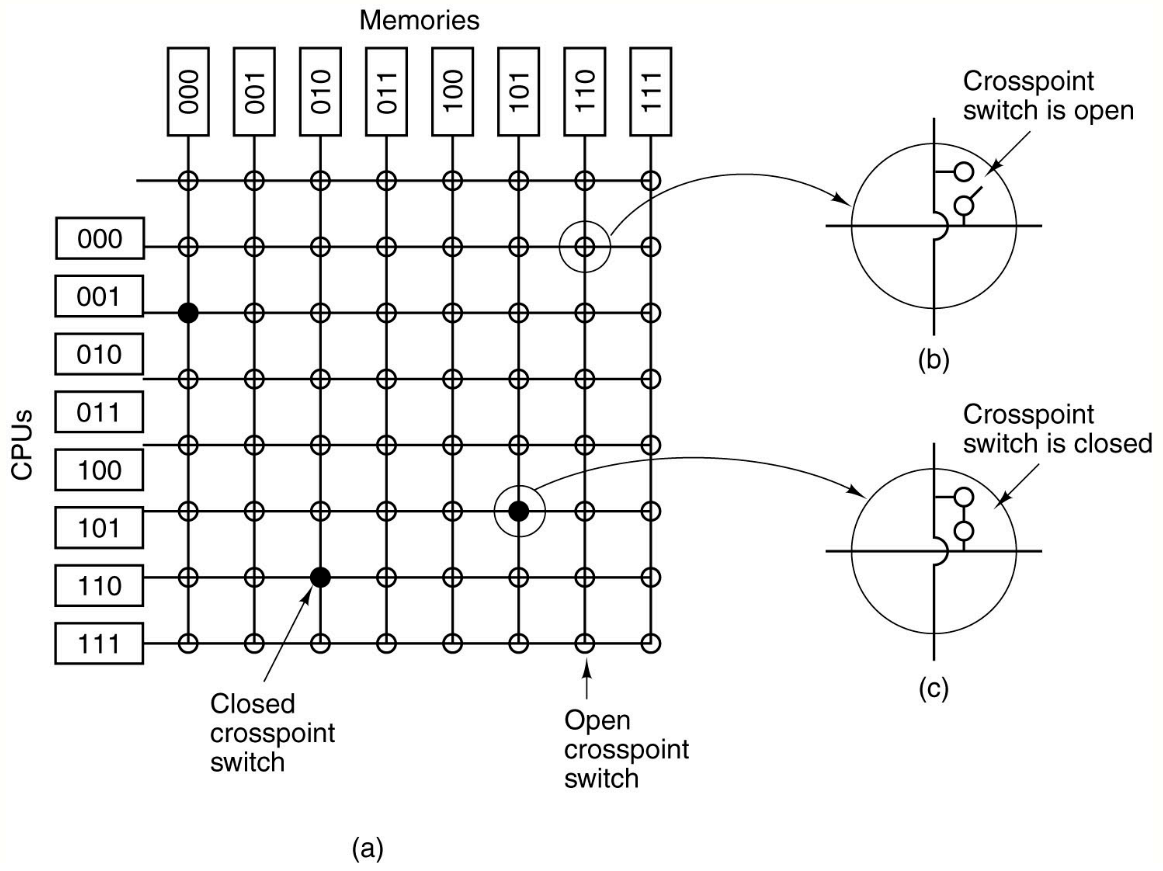
- Memory (and the memory hierarchy) in multiprocessors poses two different problems:
- Coherency: whenever the address space is *shared* – the same memory location can have multiple instances (cached data) at different processors
- Consistency: whenever different access times can be seen by processors – write operations from different processors require some model for guaranteeing a sound, consistent behaviour (the *when* issue – namely, the *ordering* of writes)

Crossbar switch UMA systems

- Even with a protocol like MESI, a single bus to interface all processors with memory limits the dimension of UMA multiprocessor systems, and usually 32 CPUs (cores) is considered a maximum.
- Beyond this limit, it is necessary to resort to another CPU-RAM interconnection system. The simplest scheme to interconnect *n* CPU with *k* memory is a *crossbar switch*, an architecture similar to that used for decades in telephone switching.

Crossbar switch UMA systems

- A switch is located at each crosspoint between a vertical and a horizontal line, allowing to connect the two, when required.
- In the figure, three switches are closed, thus connecting CPU-memory pairs (001-000), (101-101) and (110-010). (Tanenbaum, Fig. 8.27)

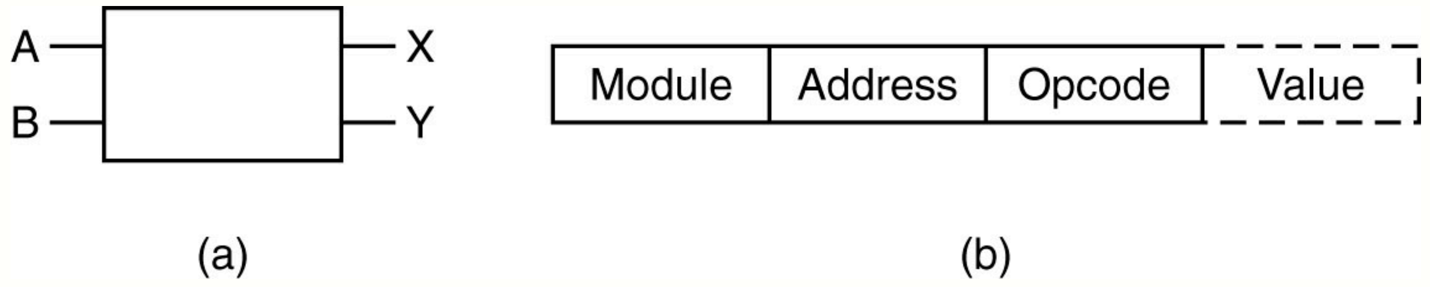


Crossbar switch UMA systems

- It is possible to configure the switches so that each CPU can connect to each memory bank (and this makes the system UMA)
- The number of switches for these scheme scales with the number of CPUs and memories; *n* CPU and *n* memories require $\boldsymbol{n^2}$ switches.
- This pattern fits well medium scale systems (various multiprocessor systems from Sun Corporation use this scheme); certainly, a 256-processor system cannot use it (256^2 switches would be required !!).

Multi-stage crossbar switch UMA

- To interconnect many CPUs, a solution is using a network set up with simple bi-directional switches with two inputs and two outputs: in these switches, each input can be redirected to each output (Tanenbaum, Fig. 8.28):



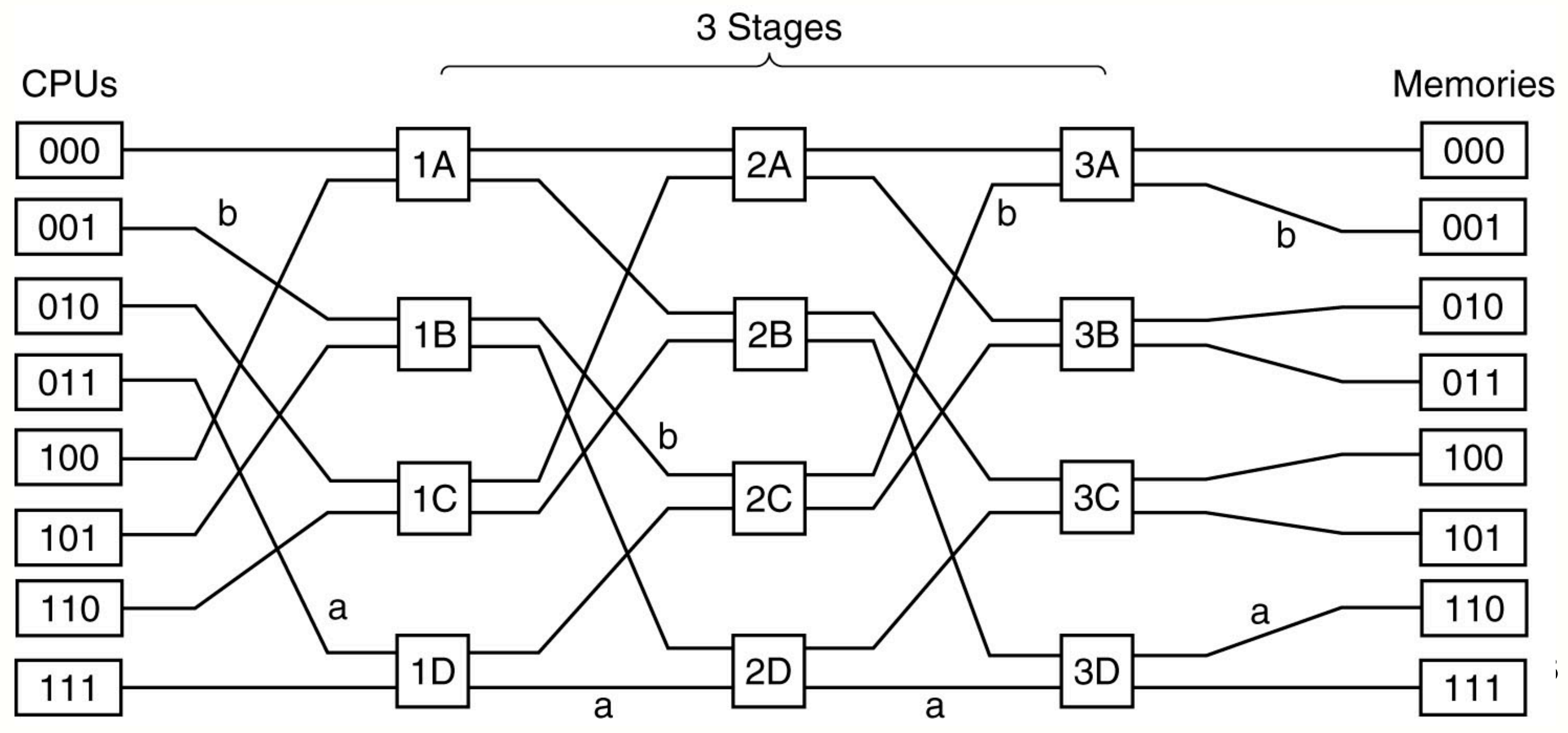
Multi-stage crossbar switch UMA

- Messages between CPU and memory consist of four parts:
- **Module**: *which memory block is requested – which CPU is requestor*
- **Address**: *address within memory block*;
- **Opcode**: *operation to carry out (READ or WRITE);*
- **Value** (optional): *value to be written (for a WRITE).*

- The switch can be programmed to analyse *Module* and to establish the output to forward the message to.

Multi-stage crossbar switch UMA

- 2 x 2 switches can be used in many ways to set up multi-stage interconnection networks. One simple case is **omega network** (Tanenbaum, Fig. 8.29):



Multi-stage crossbar switch UMA

- 8 CPUs are connected to 8 memories, using 12 switches laid out in three stages. Generalizing, n CPUs and n memories require $\log_2 n$ stages and $n/2$ switch per stage, giving a total of $(n/2)\log_2 n$ switches: much better than crossbar switches (n^2).
- Let us see how the network works. CPU 011 wants to read a data item in RAM block 110. the CPU sends a READ request to switch 1D with *Module* = 110 -- 011.
- The switch analyses the most significant (leftmost) bit and uses it for routing: 0 routes on the upper exit, 1 on the lower one.
- In this case, the request is routed to 2D.

Multi-stage crossbar switch UMA

- Switch 2D does the same: analyses the second-most significant bit (the central one) and routes the request to 3D.
- Finally, the least significant bit is used for the last routing, to block 110 (path *a* in the drawing)
- At this point, the block is read and must be sent back to CPU 011: its“address” is used, but bits are analysed from right to left (least to most significant).
- Concurrently, CPU 001 wants to execute a WRITE in block 001. The process is similar (path *b* in the drawing). Since paths *a* and *b* do not use the same switches, the two requests proceed in parallel and do not interfere with each other.

Multi-stage crossbar switch UMA

- Let us analyse what happpens if CPU 000 accesses block 000. This request clashes with that by CPU 001 at switch 3A: either request must wait.
- Contrary to crossbar switch networks, **omega networks are blocking**: not all sequences of requests can be served concurrently.
- Conflicts can arise in using a connection or a switch, in accessing a memory block or in answering a CPU request.
- A variety of techniques can be used to minimize the probability of conflicts meanwhile maximizing CPU-memory parallelism.

Caches and coherency

- Local caches pose a fundamental issue: each processor sees memory though its own cache, thus two processors can see different values for the same memory location (Hennessy-Patterson, Fig. 6.7)

Time	Event	Cache A	Cache B	RAM location for 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

Caches and coherency

- This issue is **cache coherency**, and if not solved, it prevents using caches in the processors, with heavy consequencies on performances.
- Many **cache coherency protocols** have been proposed, all of them designed to prevent different versions of the same cache block from being present in two or more caches (**false sharing**).
- All solutions are at the hardware level: the controller at each cache is capable of monitoring all memory requests on the bus coming from other CPUs, and, if necessary, the coherency protocols is activated.

Caches and coherency

Cache coherency is tackled with differents approaches in UMA and NUMA multiprocessors (and many-core processors).

- UMA multiprocessors have a processors-to-memory pathway that favors bus interconnection, so that cache coherency is based on **bus snooping**

- NUMA multiprocessors rely on complex interconnection networks for processor-to-memory communication, and the only vieble solution is based on **cache directories**

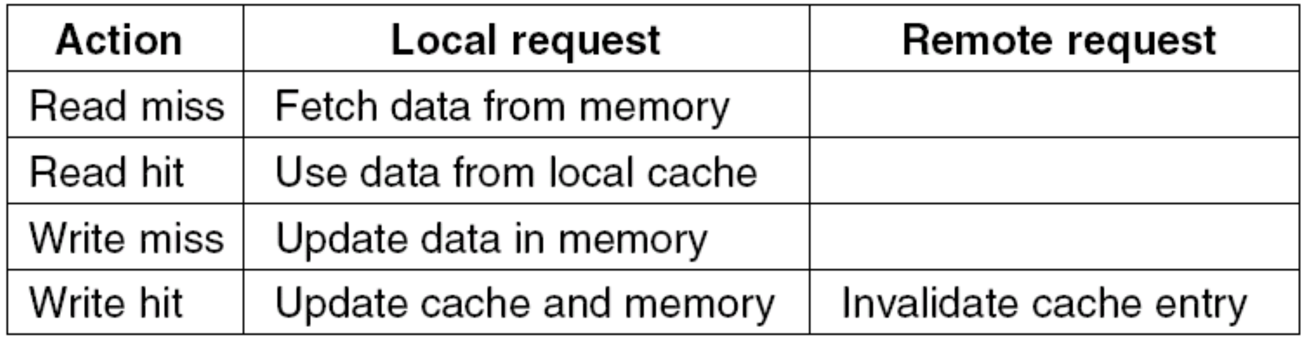
- Mixed mode approaches are emerging for MANY-CORE multiprocessors, with chips hosting a fairly large numbers of cores with on die shared caches.

Snooping Caches

- A simple cache coherency protocol is **write through.** Let us review events that happen between a processor accessing data, and its cache:
- **read miss**: the CPU cache controller fetches from RAM the missing block and load its into the cache. Subsequent reads of the same data will be solved in the cache (**read hit**).
- **write miss**: the modified data are written directly in RAM: prior to this, the block containing the data is *not* loaded into local cache.
- **write hit**: the cache block is updated and the update is propagated to RAM.
- Write operations are propagated to RAM, whose content is always updated .

Snooping Caches

- Let us consider now the operations on the side of the snooper in another CPU (right column in table). *cache A* generates read/write ops., *cache B* is the snooping cache (Tanenbaum, Fig. 8.25).
- **read miss**: *cache B* sees *cache A* fetch a block from memory but does nothing (in case of **read hit** *cache B* sees nothing at all)
- **write miss/hit**: *cache B* checks if it holds a copy of the modified data: if not, it takes no action. However, if it does hold a copy, the block containing it in flagged as invalid in *cache B*.



Action	Local request	Remote request
Read miss	Fetch data from memory	
Read hit	Use data from local cache	
Write miss	Update data in memory	
Write hit	Update cache and memory	Invalidate cache entry

Snooping Caches

- Since all caches snoop on all memory actions from other caches, when a cache modifies a data item, the update is carried out in the cache itself (if necessary) and in memory; the “old” block is removed from all other caches (actually, it is simply flagged as invalid).
- According to this protocol, no cache can have inconsistent data.
- There are variations to this basic protocol. As an example, “old” blocks could be updated with the new value, rather than being flagged as invalid (“replicating writes”).
- This version requires more work, but prevents future cache misses.

Snooping Caches

- The nicest feature of this cache coherency protocol is simplicity.
- The basic *disadvantage* of **write-through based** protocols is inefficiency, since each write operation is propagated to memory, and the communication bus is likely to become the bottleneck.
- To alleviate the problem, in these protocols not all write operations are immediately propagated to RAM: a bit is set in the cache block, to signal that the block is up-to-date, while memory is "old".
- Sooner or later, the modified block will be forwarded to RAM, possibly after more updates (not after each of them).

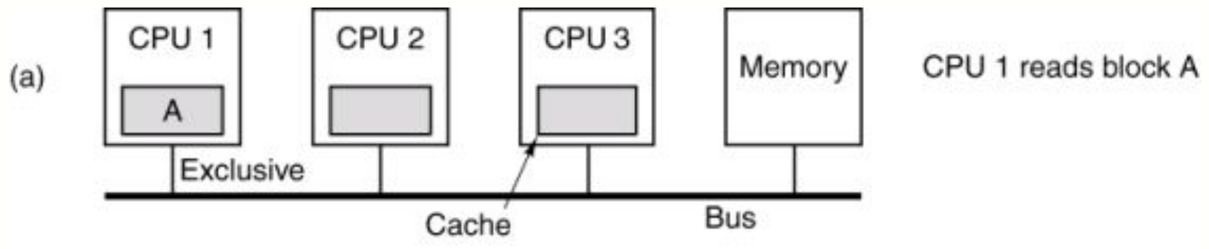
The MESI protocol

- One of the most common **write-back** cache coherency protocoll, used in modern processors, is **MESI**, where each cache entry can be in one of 4 possible states:

1.	**Invalid**	the cache entry does not contain valid data
2.	**Shared**	Multiple caches can hold the block, RAM is updated.
3.	**Exclusive**	No other caches holds the block, RAM is updated.
4.	**Modified**	The block is valid, RAM holds an old copy of the block, no other copies exist.

The MESI protocol

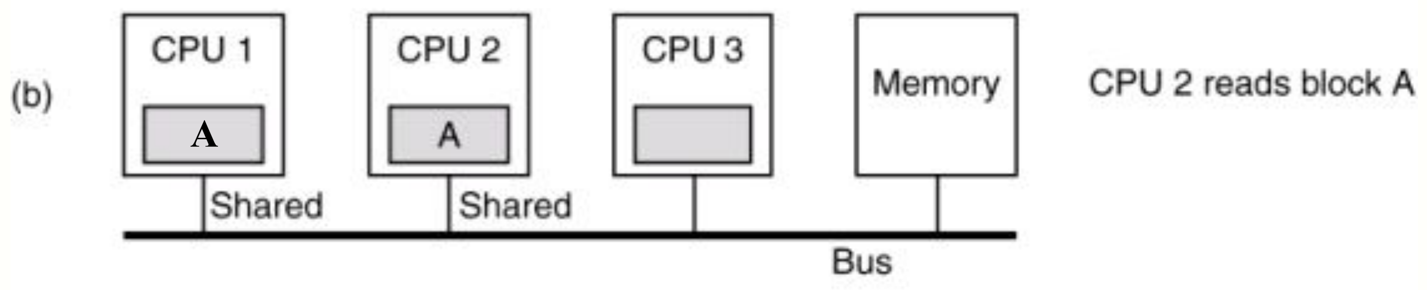
- At system startup, all cache entries are flagged I: Invalid.
- The first time a block is read into the cache of CPU 1, it is flagged E: Exclusive, because the cache is the exclusive owner of the block.



- Subsequent reads of the data item from the same CPU will hit in the cache and will not involve the bus. (Tanenbaum, Fig. 8.26a)

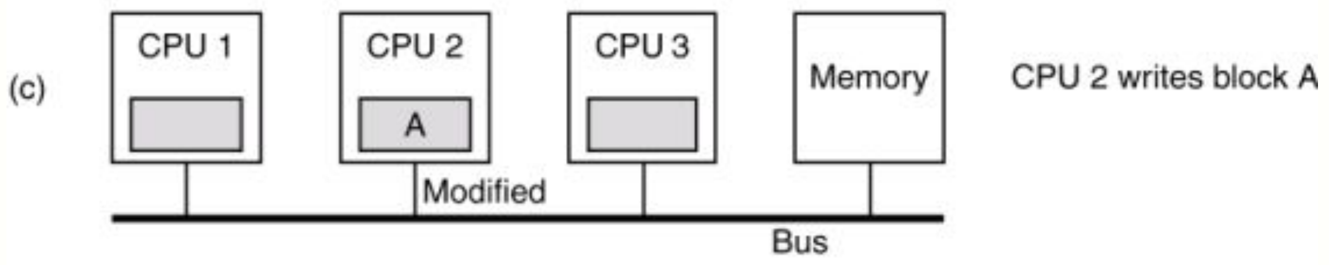
The MESI protocol

- If CPU 2 reads the same block, the snooper in CPU 1 detects the read and signals over the bus that CPU 1 holds a copy of the same buffer. Both entries in the caches are flagged S: Shared.
- Subsequent reads in the block from CPU 1 or CPU 2 will hit in the appropriate cache, with no access to the BUS (Tanenbaum, Fig. 8.26b)



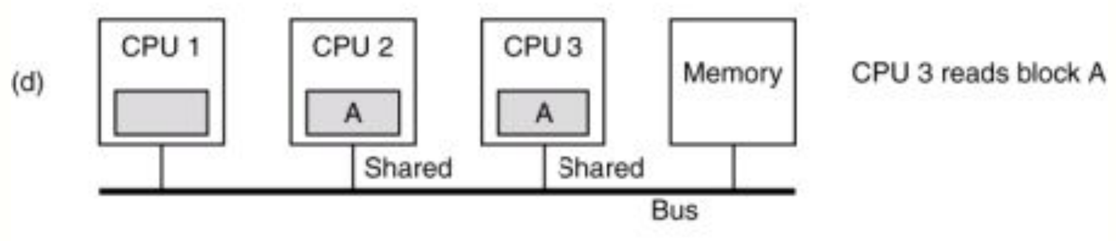
The MESI protocol

- If CPU 2 modifies a block flagged S, it sends over the bus an invalidate signal, so that other CPUs can invalidate their copy. The block is flagged M: Modified, and *it is not written to RAM* (if the block is flagged E, no signal is sent to other caches, since there are no other copies of block in other caches). (Tanenbaum, Fig. 8.26c).



The MESI protocol

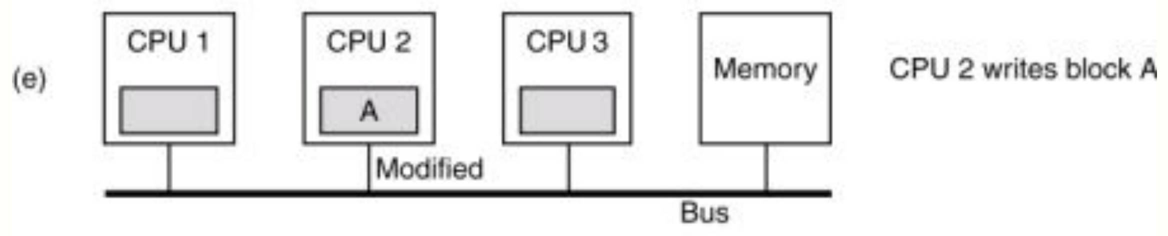
- What happens if CPU 3 tries to read the same block? The snooper in CPU 2 detects this, and holding the unique valid copy of block, it sends a wait signal to CPU 3, meanwhile updating the stale memory with the valid block.



- Once memory has been updated, CPU 3 can fetch the required block, and the two copies of the block are flagged S, shared, in both caches (Tanenbaum, Fig. 8.26.d).

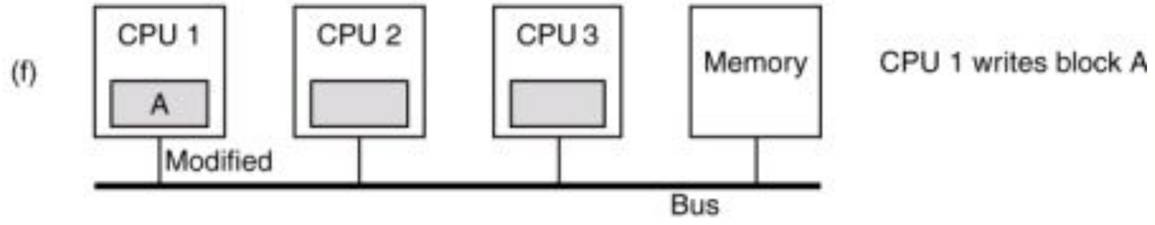
The MESI protocol

- If CPU 2 modifies the block again, in its cache, it will send again an invalidate signal over the bus, and all other copies (such as that in CPU 3) will be flagged I: Invalid. Block in CPU 2 is flagged again M: modified. (Tanenbaum, Fig. 8.26e).



The MESI protocol

- Finally, if CPU 1 tries to write into the block, CPU 2 detects the attempt, and sends a signal over the bus to make CPU 1 wait while the block is written to memory. At the end CPU 2 flags its copy as Invalid, since another CPU is about to modify it.
- At this point CPU 1 is writing a block that is stored in no cache.
- With a **write-allocate** policy, will be loaded in the cache of CPU 1 and flagged M (Tanenbaum, Fig. 8.26f).
- If no write-allocate policy is active, the write directly acts on RAM, and the block continues to be in no cache.



The MESIF protocol

- Developed by Intel for cache coherent non-uniform memory architectures. The protocol is based on five states, Modified (M), Exclusive (E), Shared (S), Invalid (I) and *Forward* (F).
- The F state is a specialized form of the S : it designates the (unique) responder for any requests for the given line, and it prevents overloading the bus due to multiple responder arbitration.
- This allows the requestor to receive a copy at cache-to-cache speeds, while allowing the use of as few multicast packets as the network topology will allow.
- Developed for supporting multi-core in a die, an adopted in recent (2015) many-core processors