Multiprocessor Systems

Multiprocessor - computer system containing more than one processor.

Principal motive is to increase the speed of execution of the system.

Sometimes other motives, such as fault tolerance and matching the application.

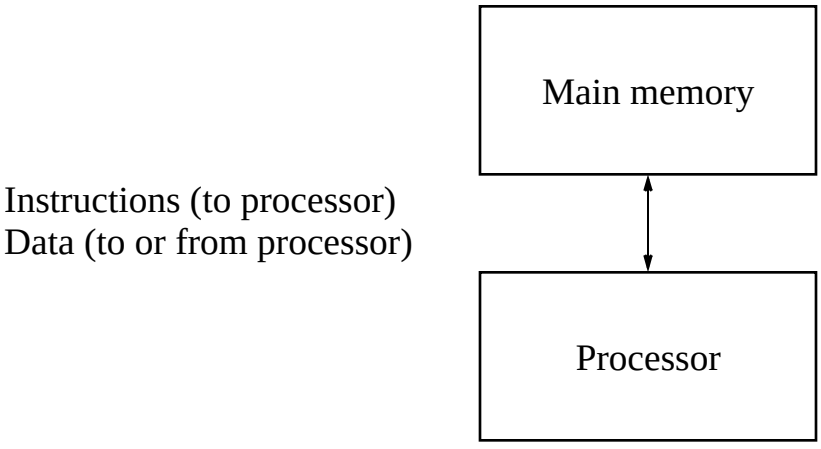
Apparent that increased speed should result when more than one processor operates simultaneously.

Types of multiprocessor systems
(where each processor executes its own program)

Shared memory multiprocessor system - a natural extension of a single processor system in which all the processors can access a common memory.

Distributed memory multicomputer system -multiple interconnected computers where each computer has its own memory.

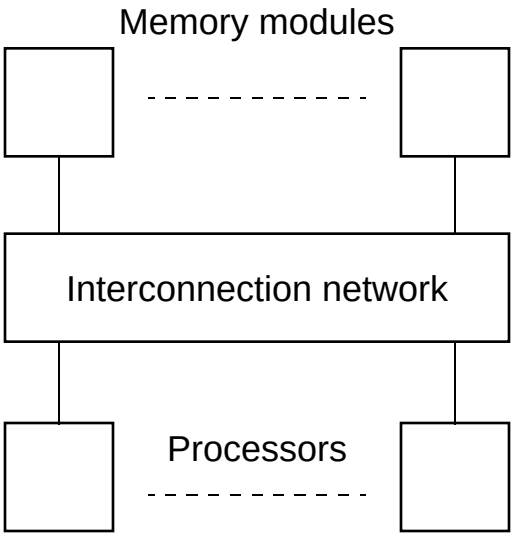
Conventional Computer



Each main memory location in the memory located its *address.* Addresses start at 0 and extend to $2^n - 1$ when there are n bits in the address.

Shared Memory Multiprocessor System

Each processor can access any memory location. One address space.



Interconnection networks

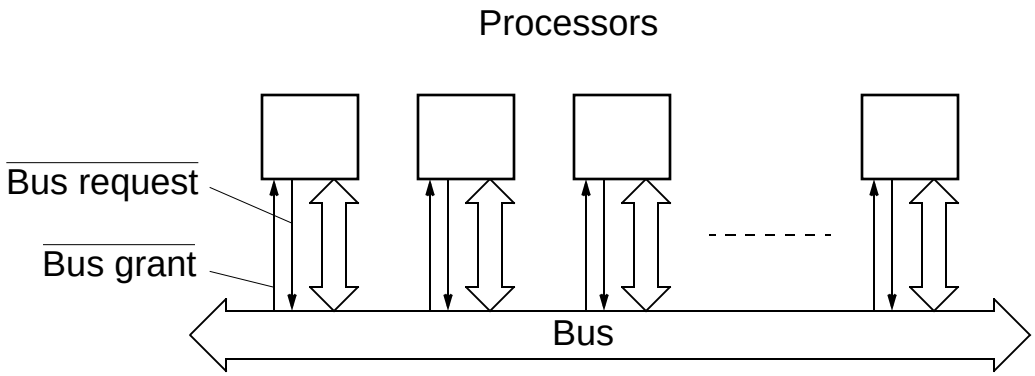
Various possible networks:

- Single bus
- Multiple buses (not much used)
- Rings
- Mesh
- Hypercube (popular in the 1980's, not any more)
- Multistage interconnection networks (MINs)

Single bus approach used in small multiprocessor systems, for example quad Pentium systems.

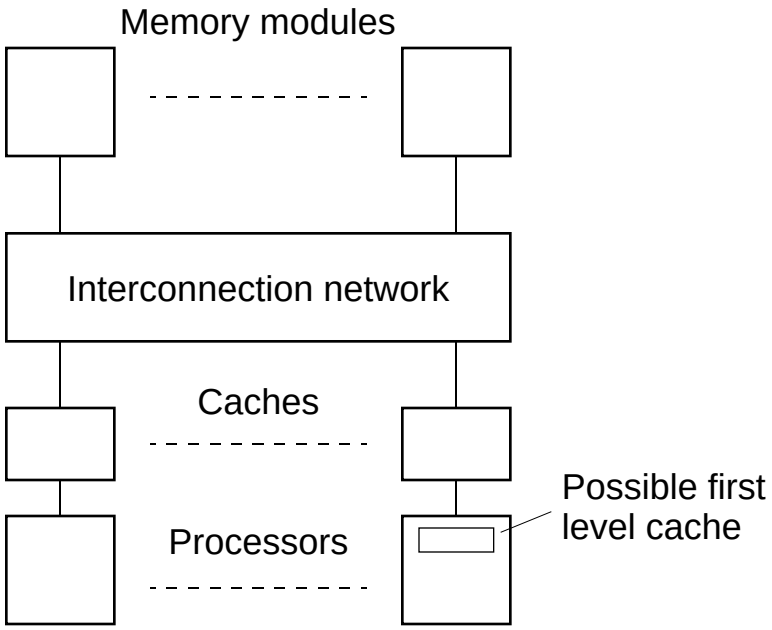
Shared bus multiprocessor system

A natural extension to a single bus microprocessor systems.



Shared memory multiprocessor system with caches

Natural to apply caches to a shared memory multiprocessor system.



Cache Coherence

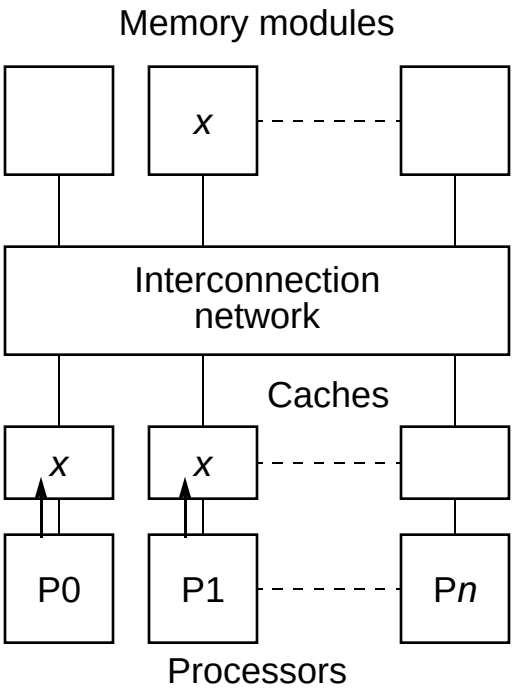
Significant additional factors to consider in using cache memory in a multiprocessor environment, in particular maintaining accurate copies of data in the multiple caches in the system.

Maintaining copies in all the caches the same is known as *cache coherence* Any read should obtain the most recent value written. (Actually more complicated that this.)

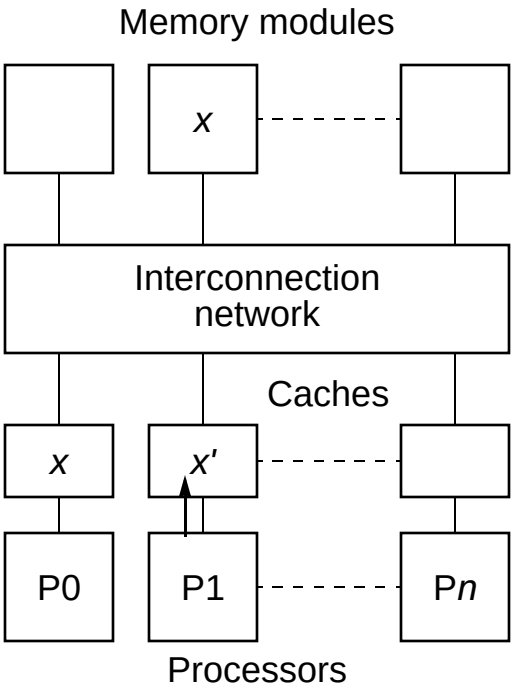
Write policy

Write-through is not sufficient, or even necessary, for maintaining cache coherence, as more than one processor writing-through the cache does not keep all the values the same and up to date.

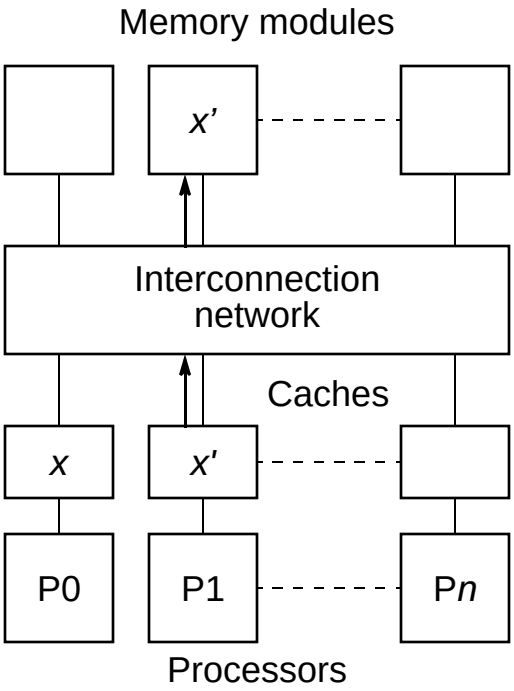
Inconsistency with write through policy copy



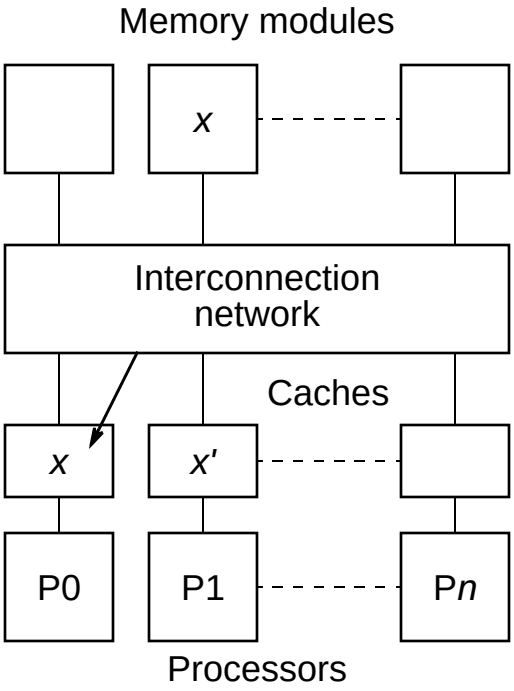
(a) Processors accessing *x*



(b) Processor 1 updating *x*



(c) After write-through



(d) Invalidating or updating copy

Two possible solutions

1.Update copy in the cache of processor 0, or

2.Invalidate copy in the cache of processor 0

both of which require access to the cache of processor 0.

Update

Update writes all cached copies with the new value of *x*..

Not usually implemented because of the overhead of the update.

In any event, it may be not completely necessary because not all processors may access the location again.

Invalidation

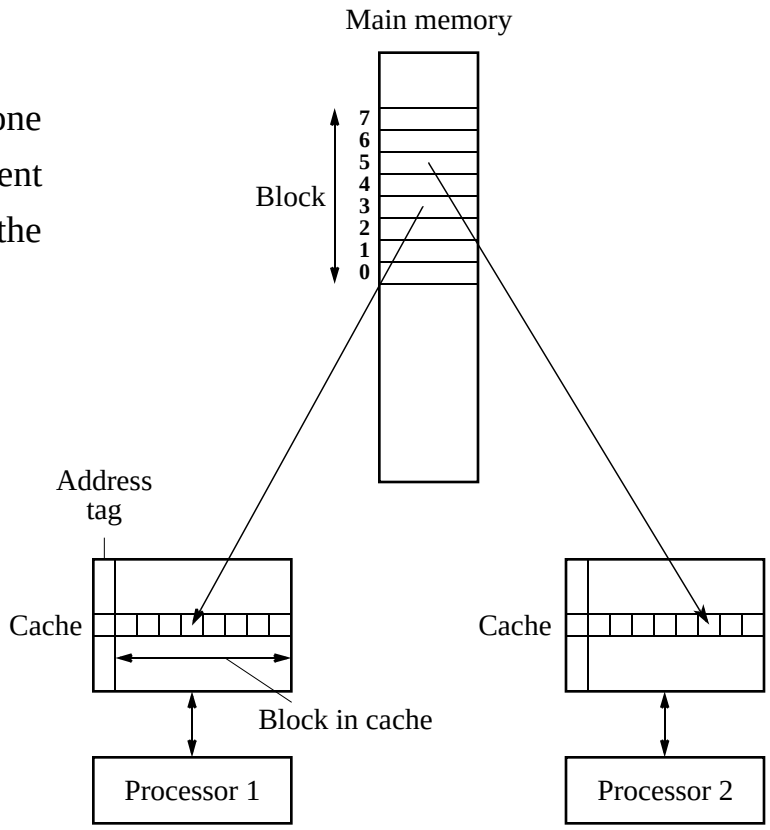
Done by resetting the valid bit associated with *x* in the cache. Now processor 0 must access the main memory if it references *x* again, to bring a new copy of *x* back into its cache. If copies existed in caches apart from the cache of processor 1, these copies would also need to be invalidated.

Numerous variations of invalidate and update protocols developed in the research community. We will describe used by manufacturers.

With invalidation, write back may be practiced rather than write-through to reduce the memory traffic. Then there is only one valid copy in one cache, and one processor has *ownership* of this copy.

False sharing

When more than one processor accesses different parts of a line but not the actual data items.



False sharing can result in significant reduction in performance because, in maintaining cache coherence, the smallest unit considered is the line.

False sharing can be reduced by distributing the data into different lines if sharing is expected.

A task for the compiler, and requires both knowledge of the use of the data and the architectural arrangements of the caches.

Alter the layout of the data stored in the main memory, separating data only altered by one processor into different blocks.

May be difficult to satisfy in all situations.

Example

```
forall (i = 0; i < 5; i++)
    a[i] = 0;
```

is likely to create false sharing as the elements of **a**, **a[0]**, **a[1]**, **a[2]**, **a[3]**, and **a[4]**, likely to be stored in consecutive locations in memory.

Would need to place each element in a different block, which would create significant wastage of storage for a large array.

forall is a high level language construct that says do the body with each value of **i** simultaneously.

Methods of Achieving Cache Coherence

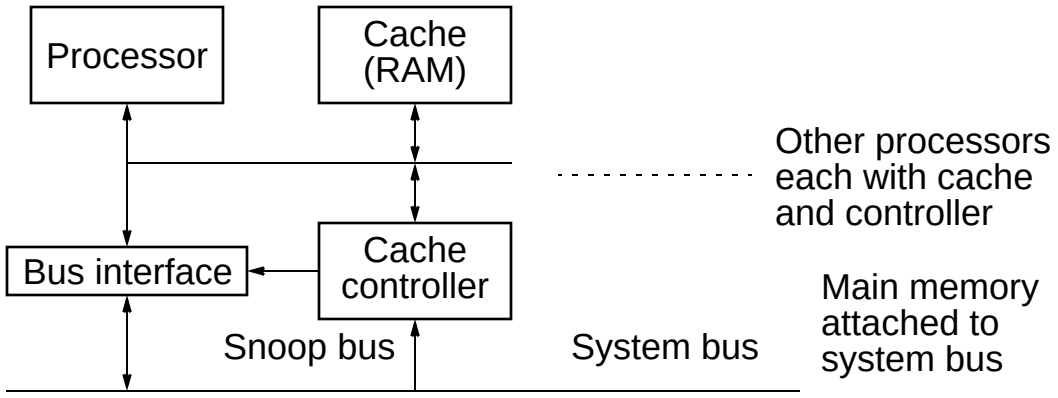
For a single bus structure, snoop bus mechanism often used.

Snoop bus mechanism

In the *snoop bus* mechanism, a "bus watcher" unit with each processor/ cache observes the transactions on the bus and in particular monitors all memory write operations. If a write is performed to a location which is cached locally, this copy is invalidated - needs a protocol -see later.

Could invalid based upon only index (not compare tags).

Snoop bus mechanism



Four-state MESI (Modified/Exclusive/Shared/Invalid) invalidate protocol

Perhaps the most popular snoop protocol with microprocessor manufacturers.

Can be found in the internal data cache of Intel Pentium, the second level Pentium cache controller, the Intel 82490 (Intel, 1994c), the Intel i860 processor (Intel, 1992b), and Motorola MC88200 cache controller (Motorola, 1988b), among others.

Each line in the cache can be in one of four states:

1. Modified (exclusive) – The line is only in this cache and has been modified (written) with respect to memory. Copies do not exist in other caches *or in memory.*
2. Exclusive (unmodified) – The line is only in this cache and has not being modified. It is consistent with memory. Other copies do not exist in other caches.
3. Shared (unmodified) – This line potentially exists in other caches. It is consistent with memory. To stay in this state, access to line can only be for reading.
4. Invalid – This line has been invalidated and does not contain valid data.

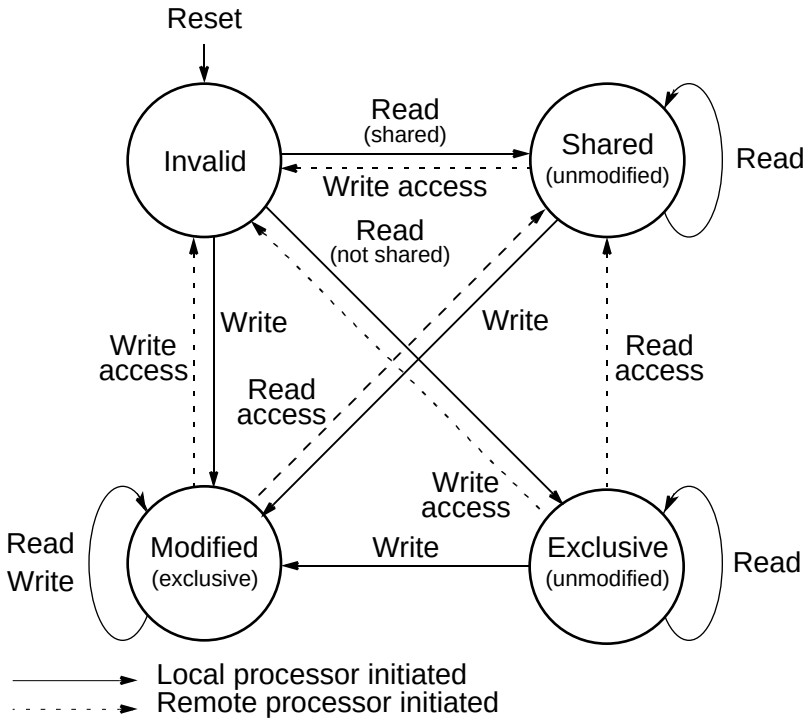
Two bits can be associated with each line to indicate the state of the line.

The modified (exclusive) and exclusive (unmodified) states are used to indicate that the processor has the only copy of the cache line.

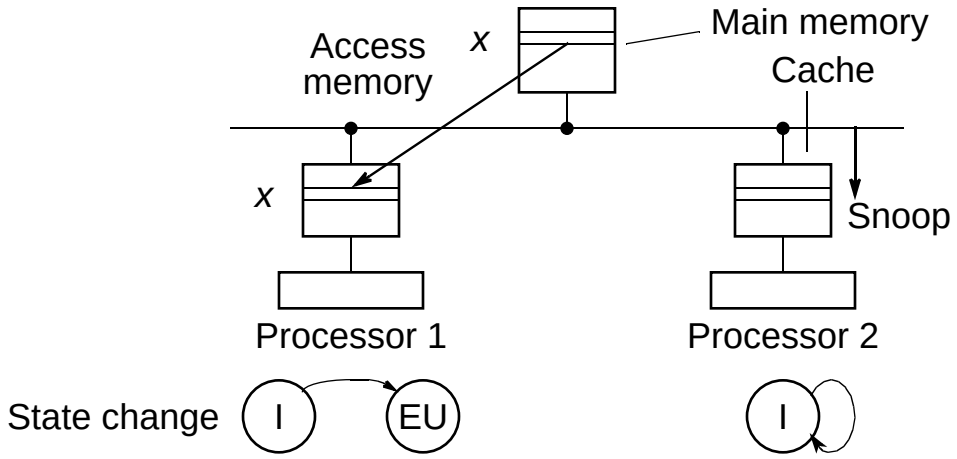
In the modified (exclusive) state, the processor has altered the contents of the line from that kept in the main memory and hence a valid copy does not even exist in the main memory. It will be necessary to write back the line before any other cache can use the line.

Lines enter the invalid state by being invalidated by other processors, i.e. this is an invalidate protocol.

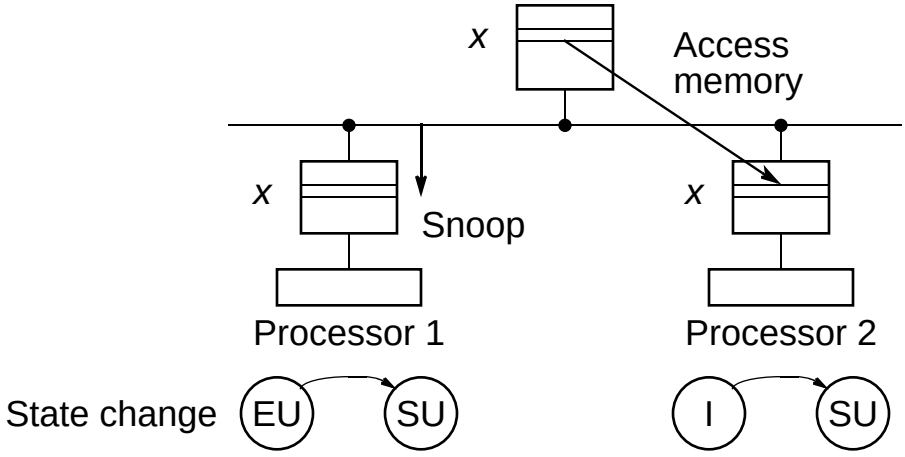
MESI protocol – major transitions without write-once



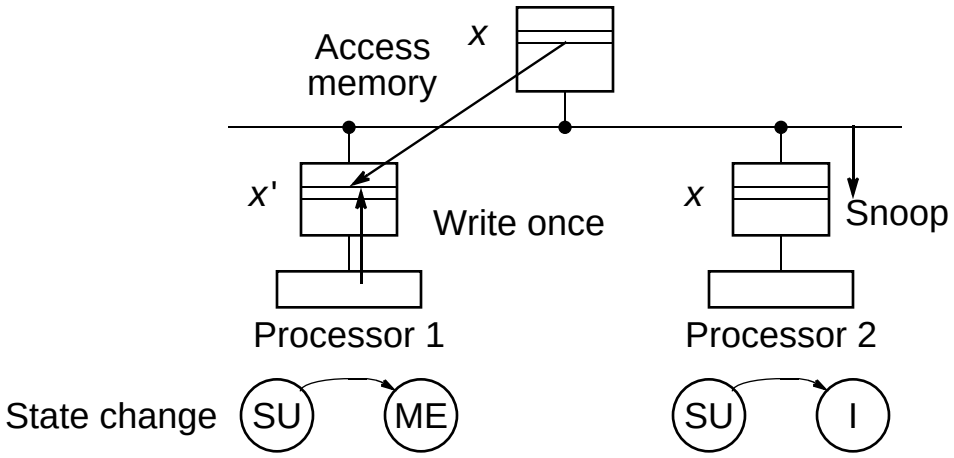
Example Sequence of MESI Protocol State Changes



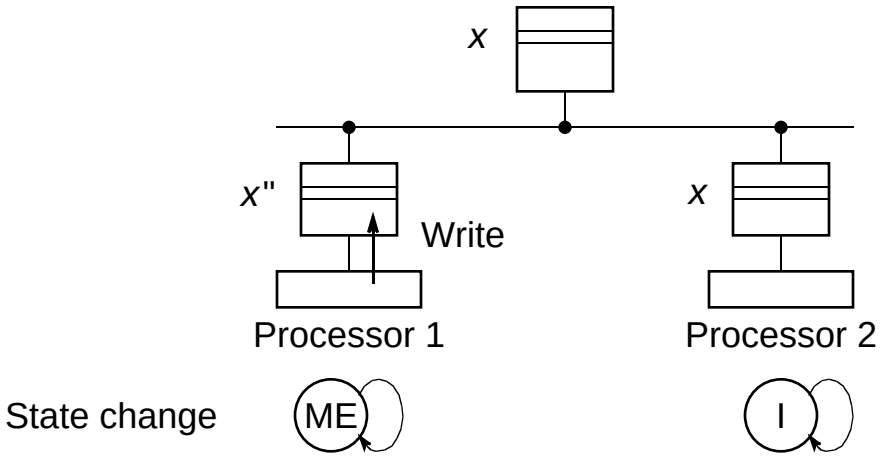
(a) Processor 1
reads *x*



(b) Processor 2
reads *x*

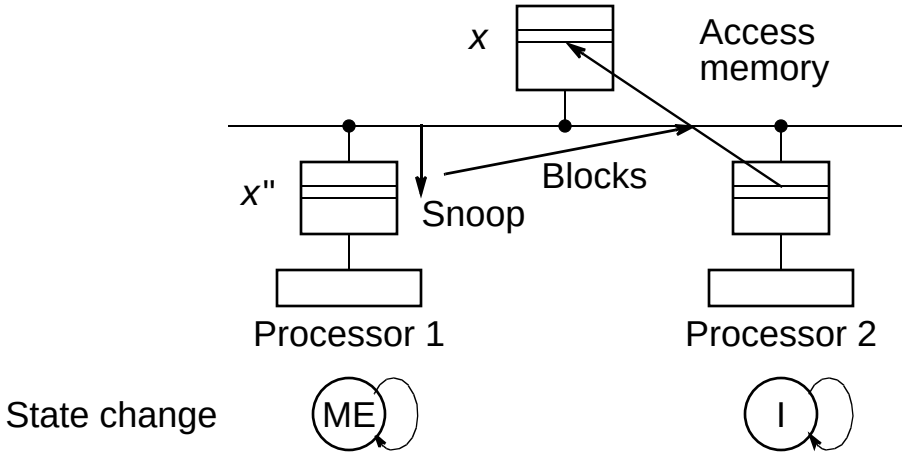


(c) Processor 1 writes to *x*

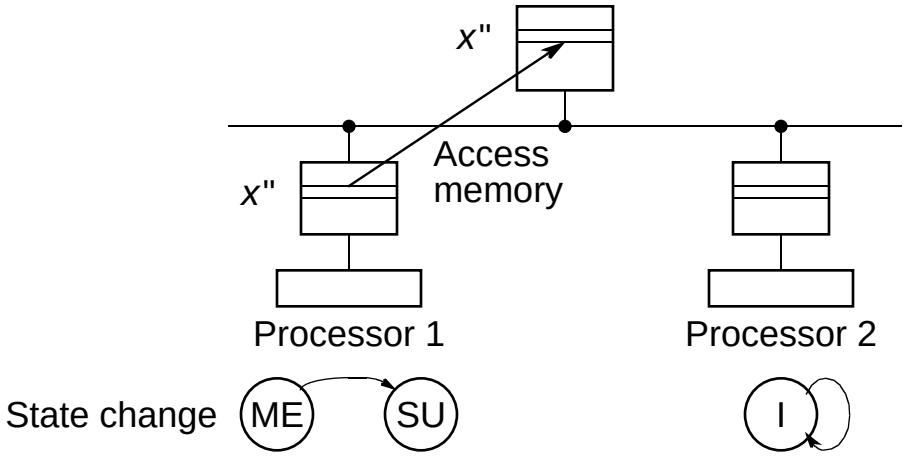


(d) Processor 1 writes to *x*

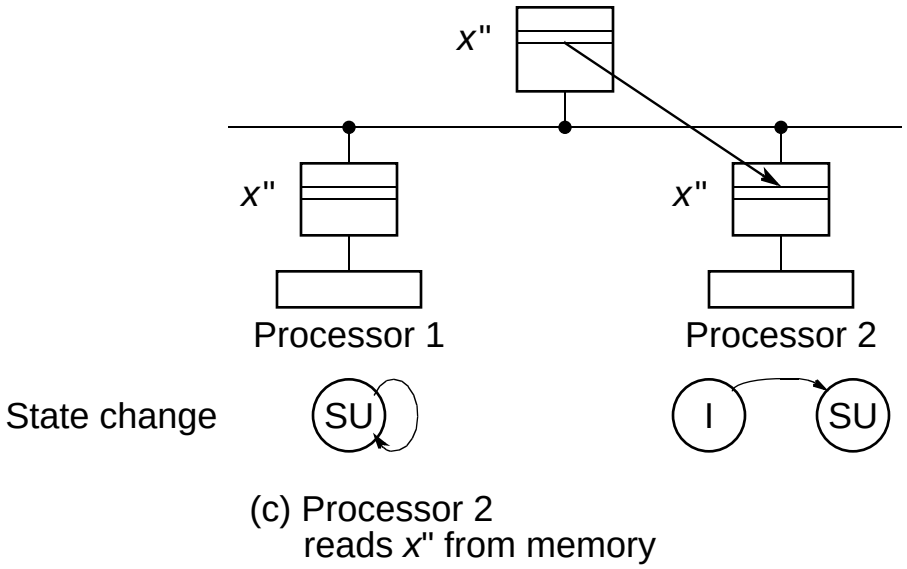
MESI protocol state changes from exclusive ownership to shared



(a) Processor 2 reads *x*



(b) Processor 1 writes back x"



(c) Processor 2
reads *x*" from memory

Performance of Single Bus Network

A key factor in any interconnection network is the bandwidth - the average number of requests accepted in a bus cycle.

Bandwidth and other performance figures can be found by one of four basic techniques:

1. Using analytical probability techniques.
2. Using analytical Markov queuing techniques.
3. By simulation.
4. By measuring an actual system performance.

Probabilistic Techniques

Principal assumptions:

1. The system is synchronous and processor requests are only generated at the beginning of a bus cycle.

2. All processor requests are random and independent of each other.

3. Requests which are not accepted are rejected, and requests generated in the next cycle are independent of rejected requests generated in previous cycles.

Assumption 2

Ignores characteristic that programs normally exhibit referential locality. However, requests from different processors normally independent.

Assumption 3

Rejected requests are ignored and not queued for the next cycle. This assumption is not generally true. Normally when a processor request is rejected in one cycle, the same request will be resubmitted in the next cycle. However, the assumption substantially simplifies the analysis and makes very little difference to the results.

Bandwidth

Probability that a processor makes a (random) request for memory = r.

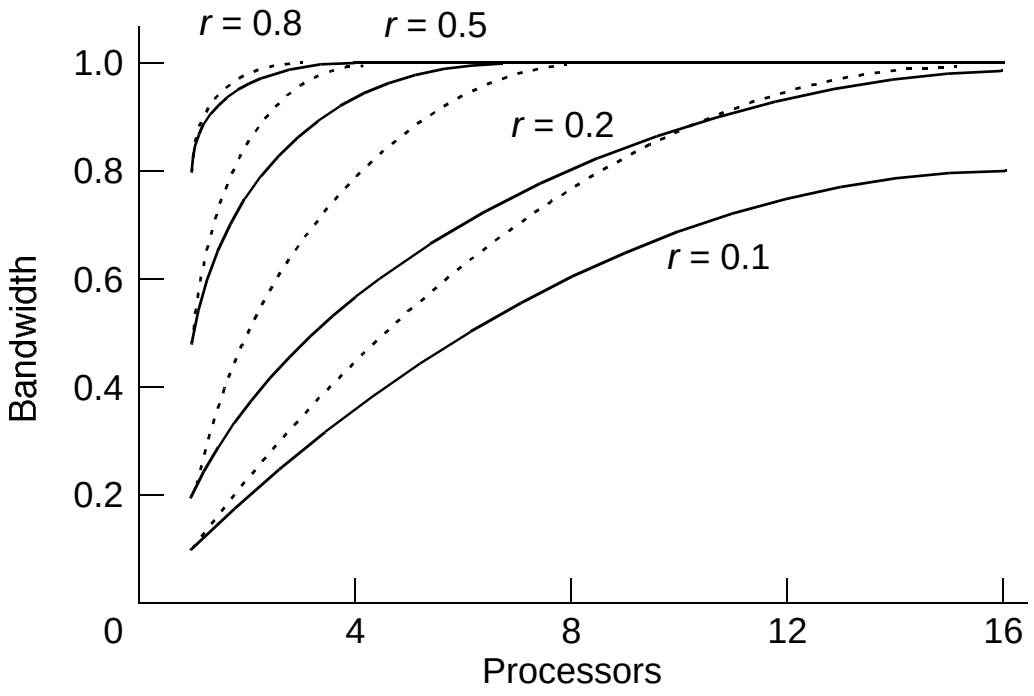
Probability that the processor does not make a request = $1 - r$.
Probability that no processors make a request for memory = $(1 - r)^p$ where there are p processors.
Probability that one or more processors make a request= $1 - (1 - r)^p$.

Since only one request can be accepted at a time in a single bus system, the average number of requests accepted in each arbitration cycle (the bandwidth, BW) is given by:

$$BW = 1 - (1 - r)^p$$

Bandwidth of a single bus system

(- - - using more accurate rate adjusted equations, see textbook)



Key Observation,

Bus saturates - at about 8 processors with r = 0.5.

Not be that bad with cache memory as then r is much less.

Still, a single bus is only suitable for a small system.