Multiprocessors

Key questions:

- How do parallel processors share data?
- How do parallel processors coordinate?
- How many processors?

Two main approaches to sharing data:

- Shared memory (single address space)
  - Synchronization
  - Uniform/nonuniform memory access
- Message passing
- Clusters (processors connected via LAN)

Two types of connection:

- Single bus
- Network

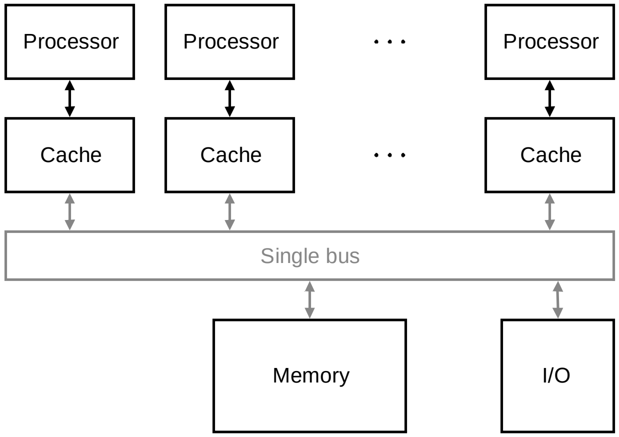
Category	Choice		Number of processors
Sharing data	Shared memory	UMA	2 – 64
		NUMA	8 – 256
	Message passing		8 – 256
Connection	Network		8 – 256
	Bus		2 – 32

Programming multiprocessors

Multiprogramming is difficult:

- Communication problems
- Requires knowledge about the hardware
- All parts of the program should be parallelized

Multiprocessors connected by a single bus



- Each processor is smaller than a multichip processor
- The use of caches can reduce the bus traffic
- There exists mechanisms to keep caches and memory consistent

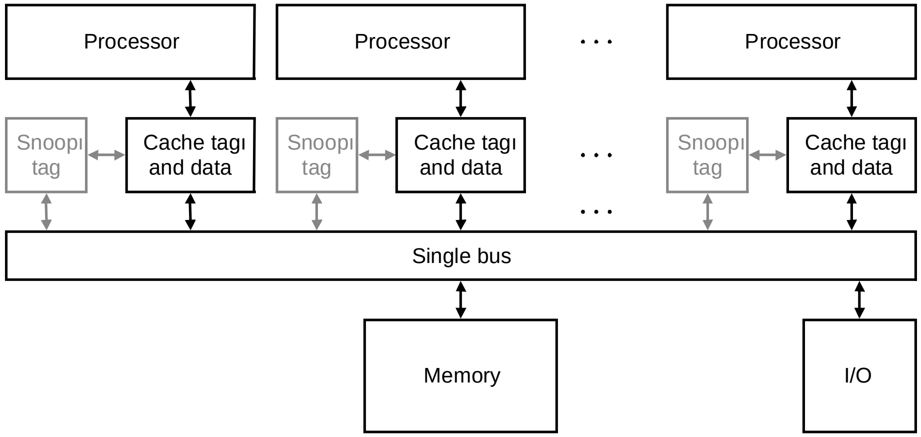
Parallel program

Shared data: A[100000], sum[10]
Private data: i, half

```
sum[Pn]=0;
for (I=10000*Pn; I<10000*(Pn+1); I++)
   sum[Pn]=sum[Pn]+A[I];


half=10
repeat
   sync();  /* wait for partial sum completion - barrier synchronization */
   if (half%2!=0 && Pn==0)
      sum[0]=sum[0]+sum[half-1];
   half=half/2;
   if (Pn<half) sum[Pn]=sum[Pn]+sum[Pn+half];
until (half==1);
```

Multiprocessor cache coherency



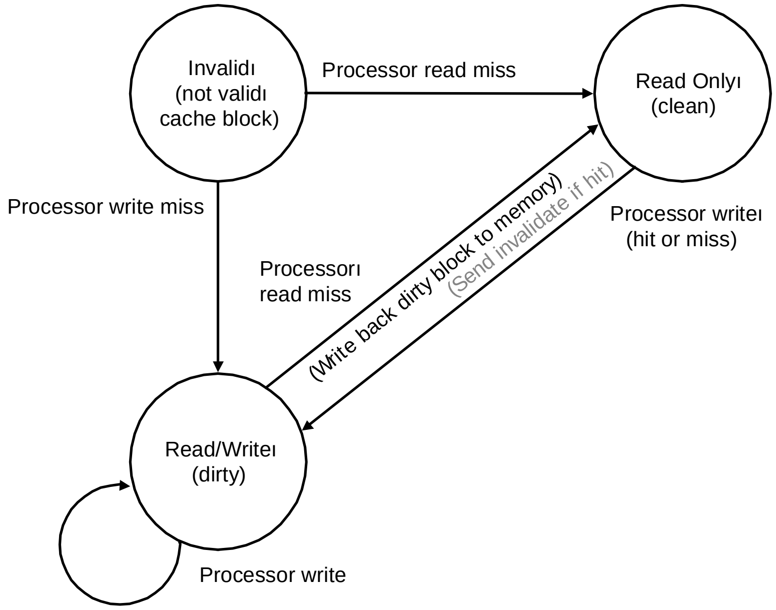
Snooping (monitoring) protocols: locate all caches that share a block to be written. Then:

- *Write-invalidate*: The writing processor causes all copies in other caches to be invalidated before changing its local copy. Similar to write-back.
- *Write-update (broadcast)*: The write processor sends the new data (the word) over the bus. Similar to write-through.
- The role of the size of the block (broadcasting only a word, false sharing).

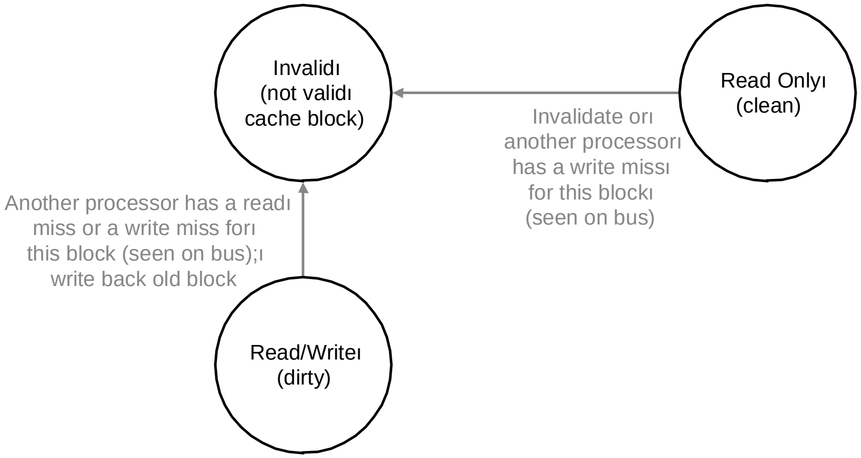
Write-invalidate cache coherency protocol based on a write-back policy

Each cache block is in one of the following states:

- *Read only*: the block is not wtitten and may be shared
- *Read/Write*: the block is written (dirty) and may not be shared
- *Invalid*: the block does not have valid data



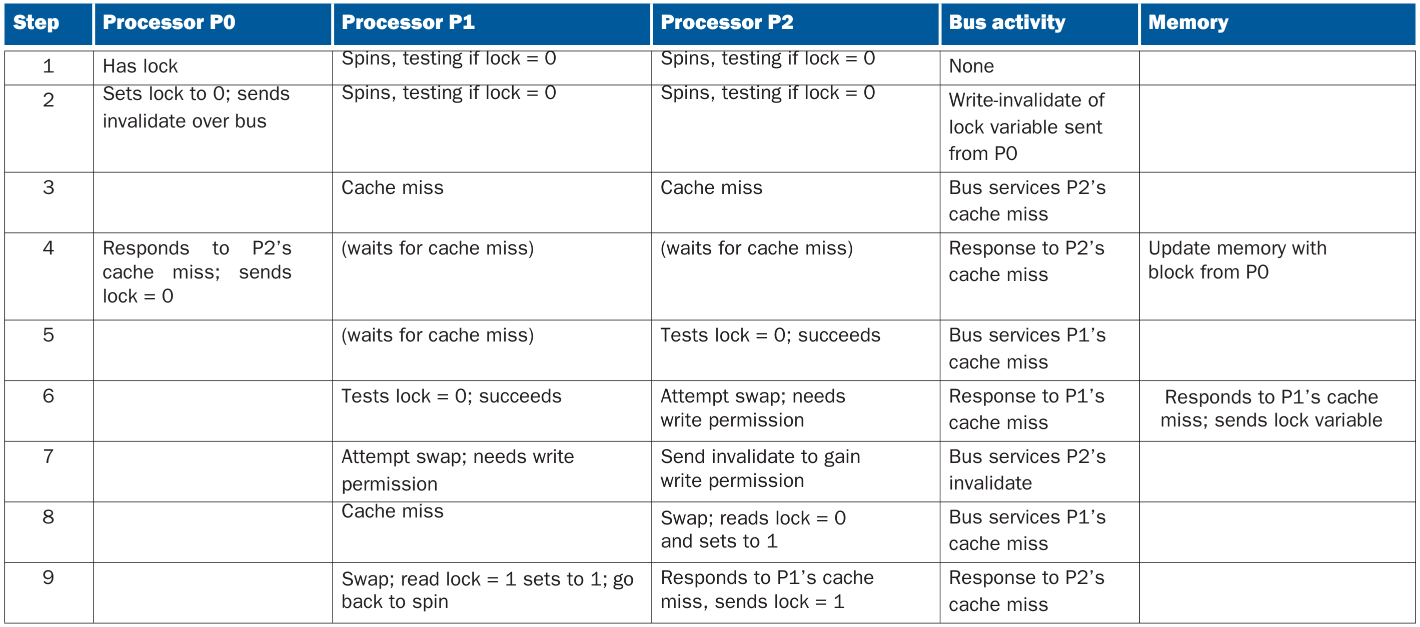
a. Cache state transitions using signals from the processor


Invalidı (not validı cache block)
Read Onlyı (clean)
Invalidate orı another processorı has a write missı for this blockı (seen on bus)
Another processor has a readı miss or a write miss forı this block (seen on bus);ı write back old block
Read/Writeı (dirty)

b. Cache state transitions using signals from the bus

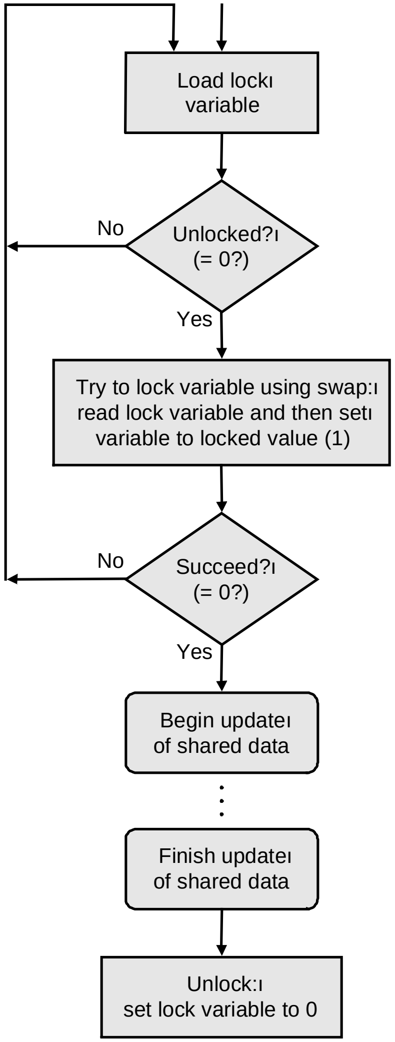
Synchronization using coherency

- Using locks (semaphores)
- Atomic swap operation



Step	Processor P0	Processor P1	Processor P2	Bus activity	Memory
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	None	
2	Sets lock to 0; sends invalidate over bus	Spins, testing if lock = 0	Spins, testing if lock = 0	Write-invalidate of lock variable sent from P0	
3		Cache miss	Cache miss	Bus services P2's cache miss	
4	Responds to P2's cache miss; sends lock = 0	(waits for cache miss)	(waits for cache miss)	Response to P2's cache miss	Update memory with block from P0
5		(waits for cache miss)	Tests lock = 0; succeeds	Bus services P1's cache miss	
6		Tests lock = 0; succeeds	Attempt swap; needs write permission	Response to P1's cache miss	Responds to P1's cache miss; sends lock variable
7		Attempt swap; needs write permission	Send invalidate to gain write permission	Bus services P2's invalidate	
8		Cache miss	Swap; reads lock = 0 and sets to 1	Bus services P1's cache miss	
9		Swap; read lock = 1 sets to 1; go back to spin	Responds to P1's cache miss, sends lock = 1	Response to P2's cache miss	

FIGURE 9.3.6 Cache coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write-invalidate coherency. P0 starts with the lock (step 1). P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the *critical section* (steps 6 and 7), while P1 spins and waits (steps 7 and 8). The "critical section" is the name for the code between the lock and the unlock. When P2 exits the critical section, it sets the lock to 0, which will invalidate the copy in P1's cache, restarting the process.


Load lockı variable
Unlocked?ı (= 0?)
No
Yes
Try to lock variable using swap:ı read lock variable and then setı variable to locked value (1)
Succeed?ı (= 0?)
No
Yes
Begin updateı of shared data
Finish updateı of shared data
Unlock:ı set lock variable to 0