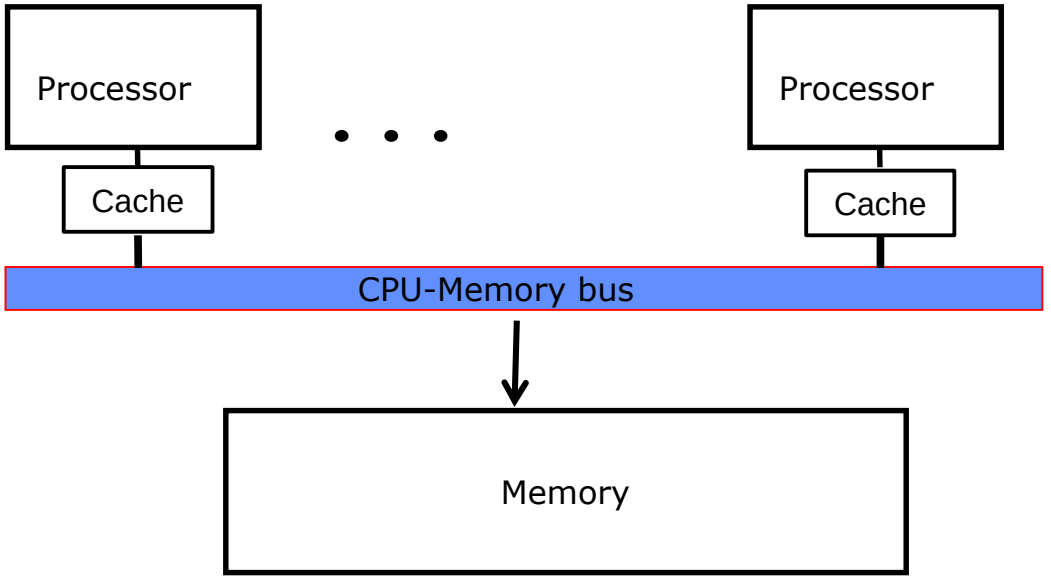
Parallel Computer Architecture
Fall 2018

Distributed Shared Memory Architectures
&
Directory-Based Memory Coherence

Nikos Bellas

Computer and Communications Engineering Department
University of Thessaly

Unified Shared Memory Multiprocessors

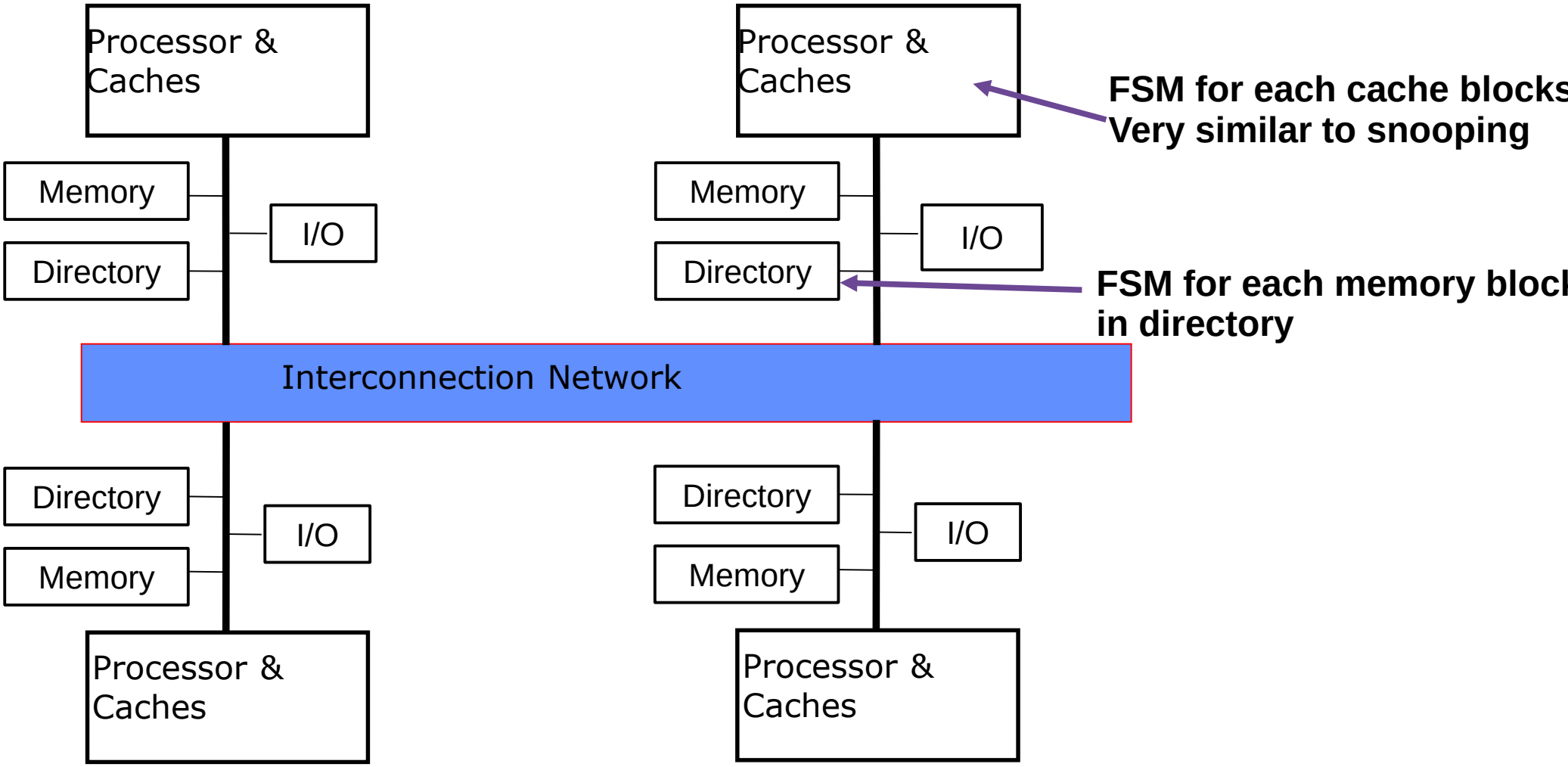


Not scalable

Assume a UMA system with 16 processors, each processor with a L1-L2 Cache hierarchy, with L2 global miss rate 1%.
The block size of the L2 Cache is 64 bytes, and the processors are clocked at 4 GHz.
Assume that the processors can sustain IPC=2, and each application has on average 30% load/store instructions. What is the max bandwidth required by all processors?

Each application has on average 30% load/store instructions. Therefore, 0.3 data references per instruction or 0.3*2 data references each cycle (for IPC=2).
This amounts to 0.3*2*4*16 = 38.4 data references per ns for all 16 processors.
These data references create 38.4 * 0.01 = 0.384 misses/ns which require 0.384*64=24.576 bytes/ns bandwidth to memory = **24.576 GB/sec**
On the other hand, the memory bandwidth of the highest-performance UMA 16-way multiprocessor in 2006 was **2.4 GB/sec**

Distributed Shared Memory (DSM or NUMA)



Distributed Shared Memory (DSM or NUMA)

- Each processor has its own physical memory
  - and potentially I/O facilities like DMA
- No symmetric memory
- However, the memories are shared and can be accessed with load/store instructions by any processor
- Every memory module has associated directory information (one directory per processor)
  - It only keeps information for the blocks of the memory module of that processor – i.e. the sharing status for each block is in a single known location
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate ONLY with the nodes that have copies if necessary
  - **No broadcasting** – Only **point to point** network transactions

Directory Protocol

- No broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Typically 3 agents involved
  - Local node (processor+cache) where a request originates
  - Home node (directory controller) where the memory location of an address resides
  - Remote node (processor+cache) has a copy of a cache block, whether exclusive or shared

Directory Protocol

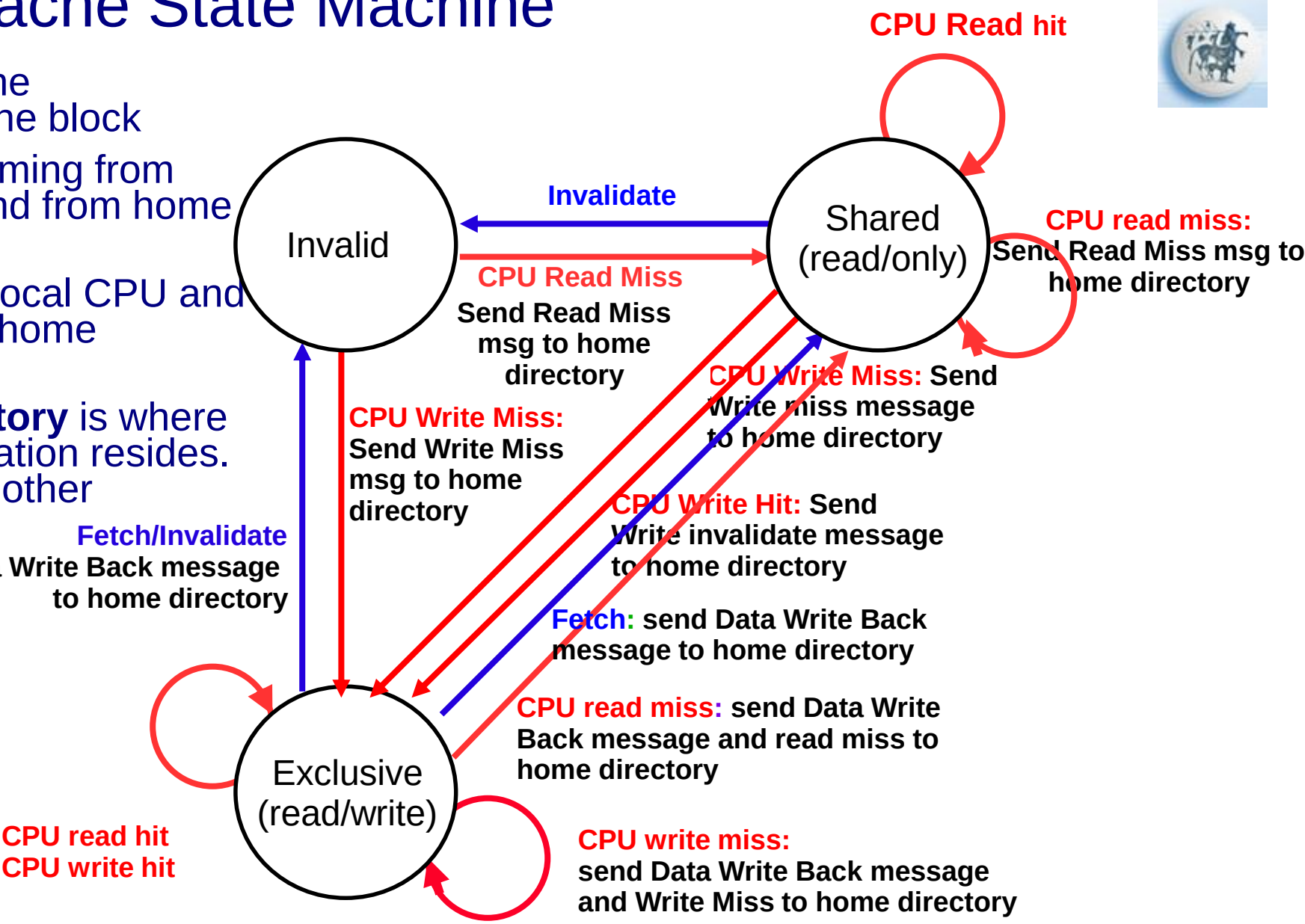
- Two different FSMs
  - One for each cache block (in the cache)
  - One in directory for each memory block of the memory module
- The cache block FSM is similar to snoopy protocol
- The directory FSM has also three states:
  - Uncached: no processor has it; not valid in any cache, only in memory
  - Shared: ≥ 1 processors have memory block, memory up-to-date
  - Exclusive: 1 processor (owner) has data; memory out-of-date

Directory Protocol

- In addition to sharing state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple:
  - Writes to non-exclusive data
    $\Rightarrow$ write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

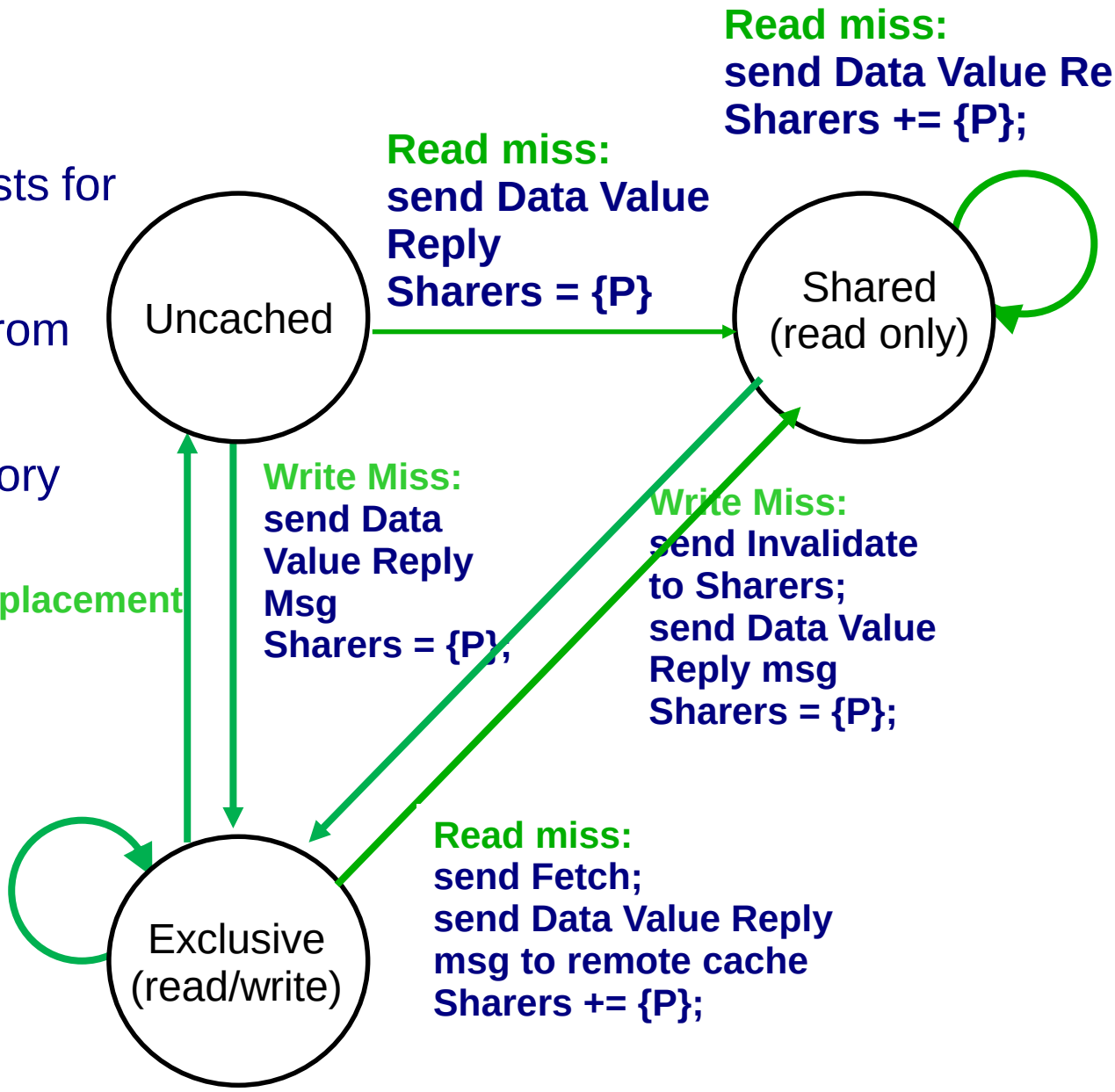
CPU -Cache State Machine

- State machine for each cache block
- Requests coming from local CPU and from home directory
- Red is from local CPU and Blue is from home directory
- **Home directory** is where the mem location resides. Can be in another processor



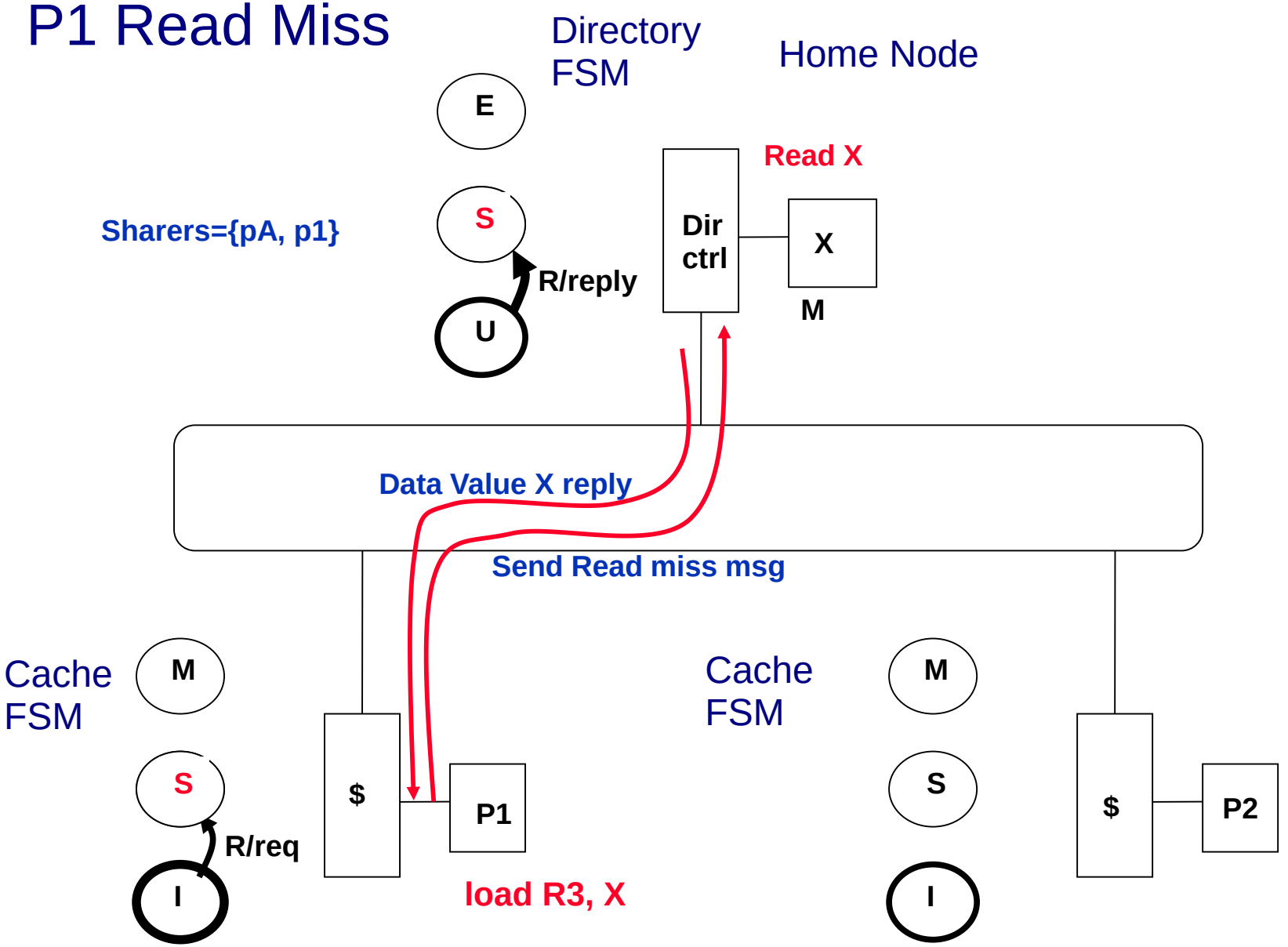
Directory State Machine of Home Node

- State machine for *Directory* requests for each memory block
- Requests coming from cache controllers
- Uncached state if data only in memory

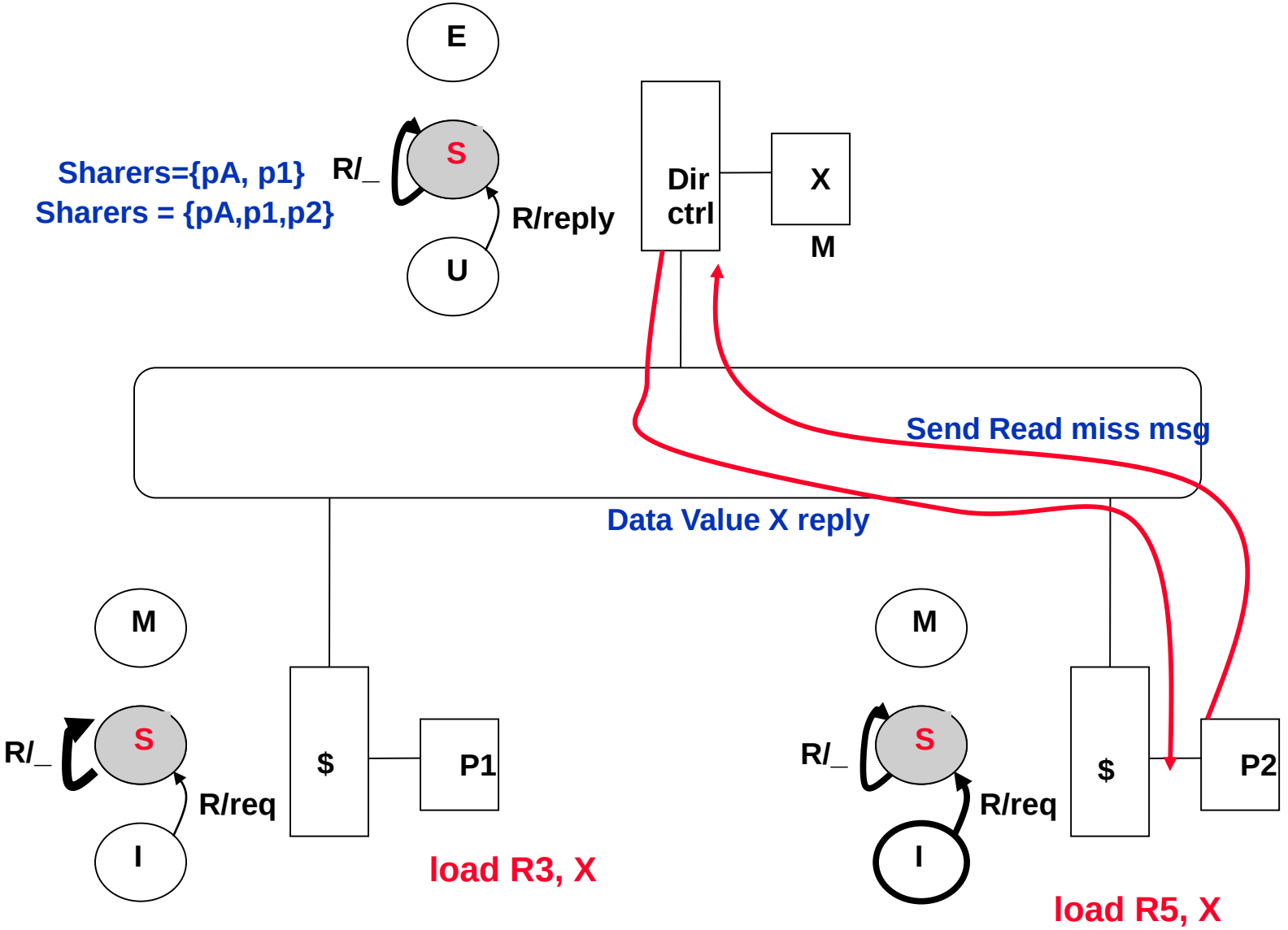


Example Directory Protocol

P1 Read Miss

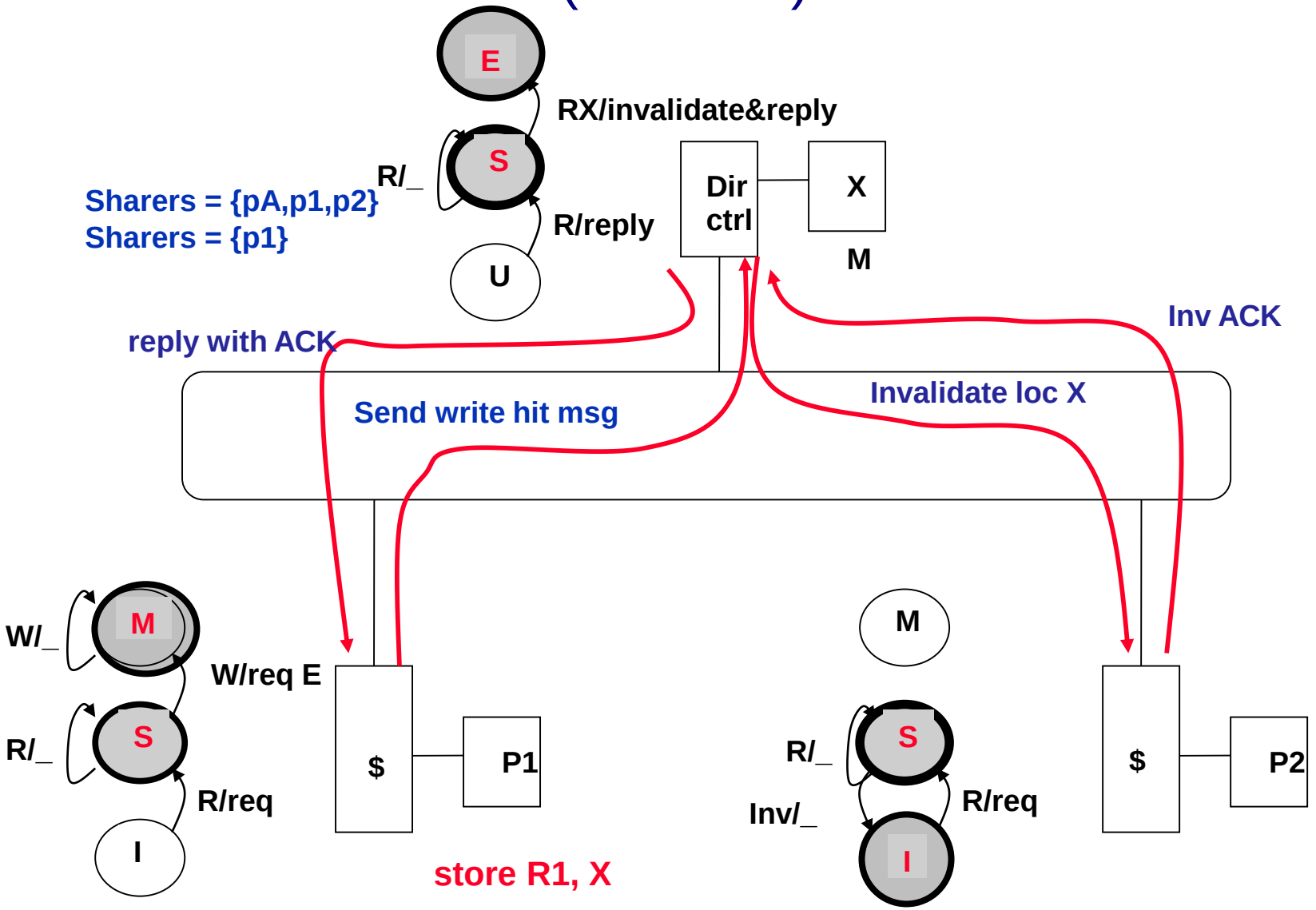


Example Directory Protocol
P2 Read Miss



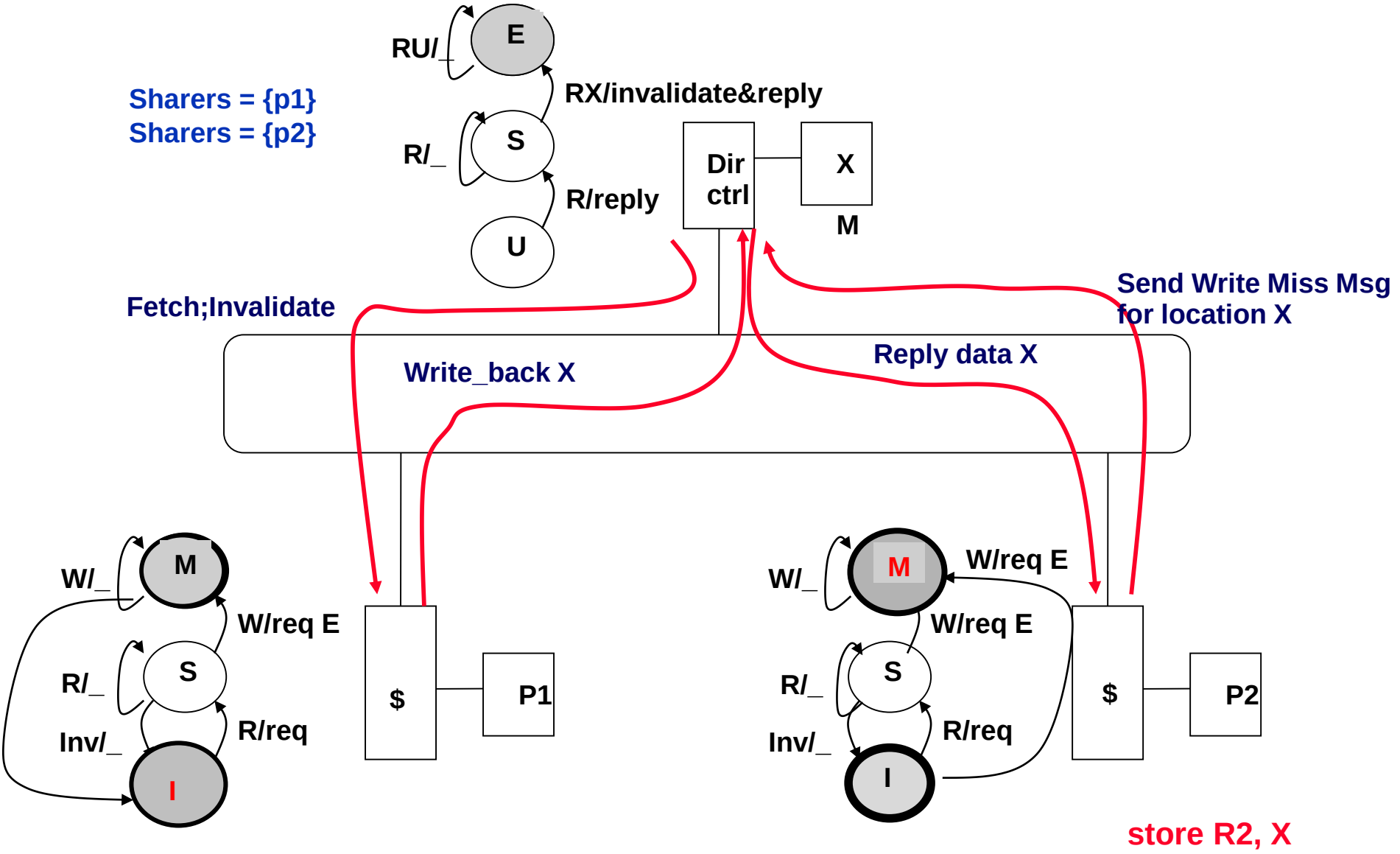
Example Directory Protocol

P1 writes to shared (Write Hit)



Example Directory Protocol

P2 writes to Exclusive (Write Miss)



Discussion

- All these mechanisms are hardware based
- BUT, programmer should be concerned about performance
- A frequently accessed shared data structure can easily thrash such a system
- Data placement near access points is still critical

Synchronization

Synchronization

- Question:
- How are all these hardware enhancements used in software to implement synchronization primitives?
  - The key ability to implement synchronization is a set of hardware primitives with the ability to atomically read and modify a memory location
  - Typically, part of the ISA
  - Higher level structures (synchronization libraries) based on these primitives
  - Locks, barriers, etc...

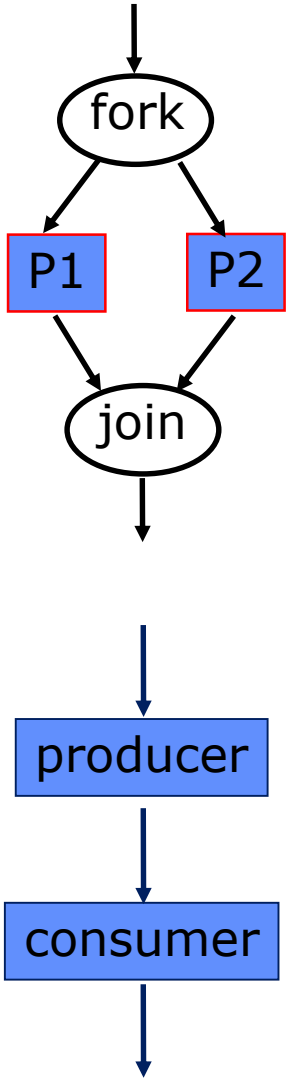
Synchronization

The need for synchronization arises whenever there are concurrent processes in a system
(even in a uniprocessor system)

Exclusive use of a resource: Operating system has to ensure that only one process uses a resource at a given time

Barriers: In parallel programming, a parallel process may want to wait until several events have occurred

Producer-Consumer: A consumer process must wait until the producer process has produced data



Locks or Semaphores

E. W. Dijkstra, 1965

A *semaphore s* is a non-negative integer, with the following operations:

P(s): *if s>0, decrement s by 1 and enter critical section, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P's and V's must be executed atomically, i.e., without

- *interruptions* or
- *interleaved accesses to s* by other processors

```
Process i
P(s)
    <critical section>
V(s)
```

initial value of *s* determines the maximum no. of processes in the critical section

Use of Semaphores
Locks

Example: *m is a memory location, R is a register*

```
AtomicExchange (m), R:
Rt =  M[m];
M[m] = R;
R = Rt;
```

m is a shared location called Lock.
When Lock==0, we can enter the Critical Region
When Lock==1, we cannot enter the Critical Region
To enter a critical section, processors
need to capture the lock by setting it 0 --> 1
Unlocking does not need to be atomic

```
Lock (m), R  /* R is local */
lock: R = 1;
      AtomicExchange (m), R;
       BNEZ R, lock;
       <critical section>
       R = 0;
       M[m] = R;  /* unlock */
```

Implementation of Atomic Exchange
Load Linked & Store Conditional Instructions

Examples: *m is a memory location, R is a register*
If (m) was changed by another processor or a context switch happened between the execution of LL and SC, the SC fails

```
LL  R, (m) :
  R = M[m];

SC  R, (m) :
     if M[m] has changed since last LL then
          R = 0;
     else if the processor context switched since last LL then
          R = 0;
     else
          M(m) = R;
           R = 1;
```

Note: LL and SC go in pairs

```
Lock (m), R :  /* initialize R with 1 */
try : R3 = R;
       LL R2, (m);    /* (m) <--> R */
       SC  R3, (m);
       BEQZ R3, try; /* was the exchange atomic? */
       BNEZ R2, try;   /* if yes, check for lock availability */
       < critical section>
```

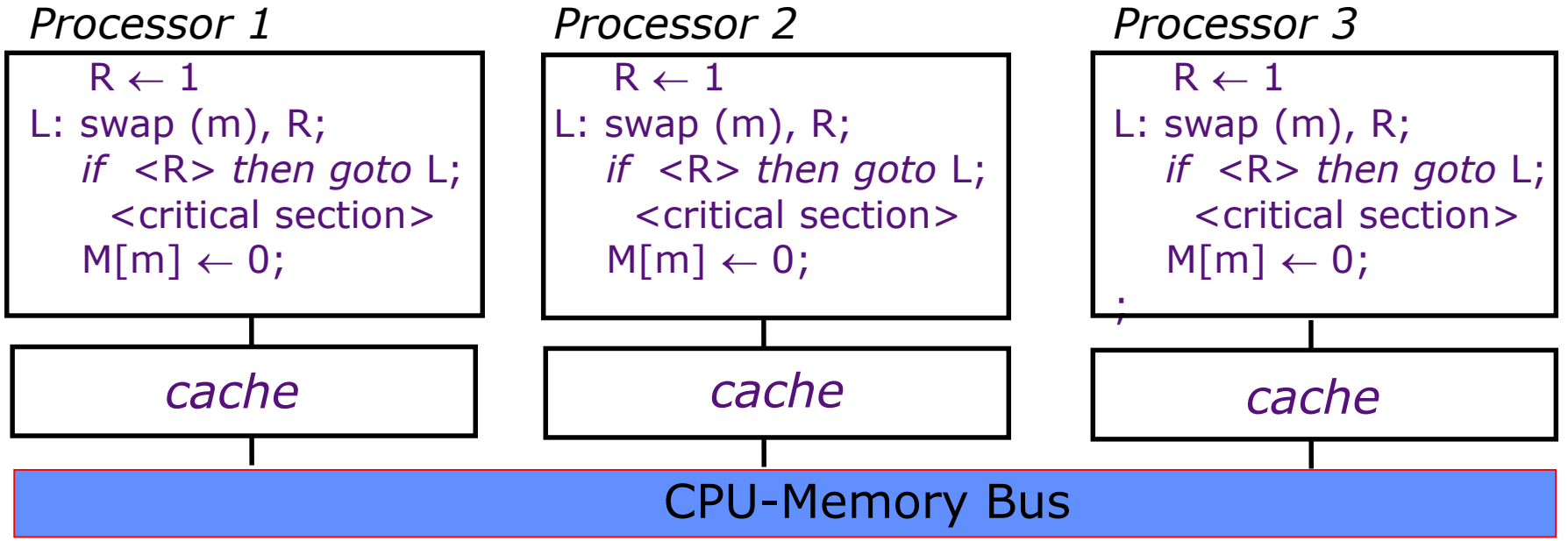
Load Linked & Store Conditional Instructions

LL R1, (m)
SC R2, (m)

LL and SC implementation in a multiprocessor :

- keep track of the address *m* in a link register R_l
  $R_l \leftarrow m$
- The LL instruction causes the data in (m) to be loaded in the local cache of CPU A
- If an interrupt occurs, or if the cache block that corresponds to memory location *m* is invalidated (by a write from another CPU B to *m*), the link register is cleared
- The *SC* instruction checks if the address m matches the value of the link register

Synchronization and Caches

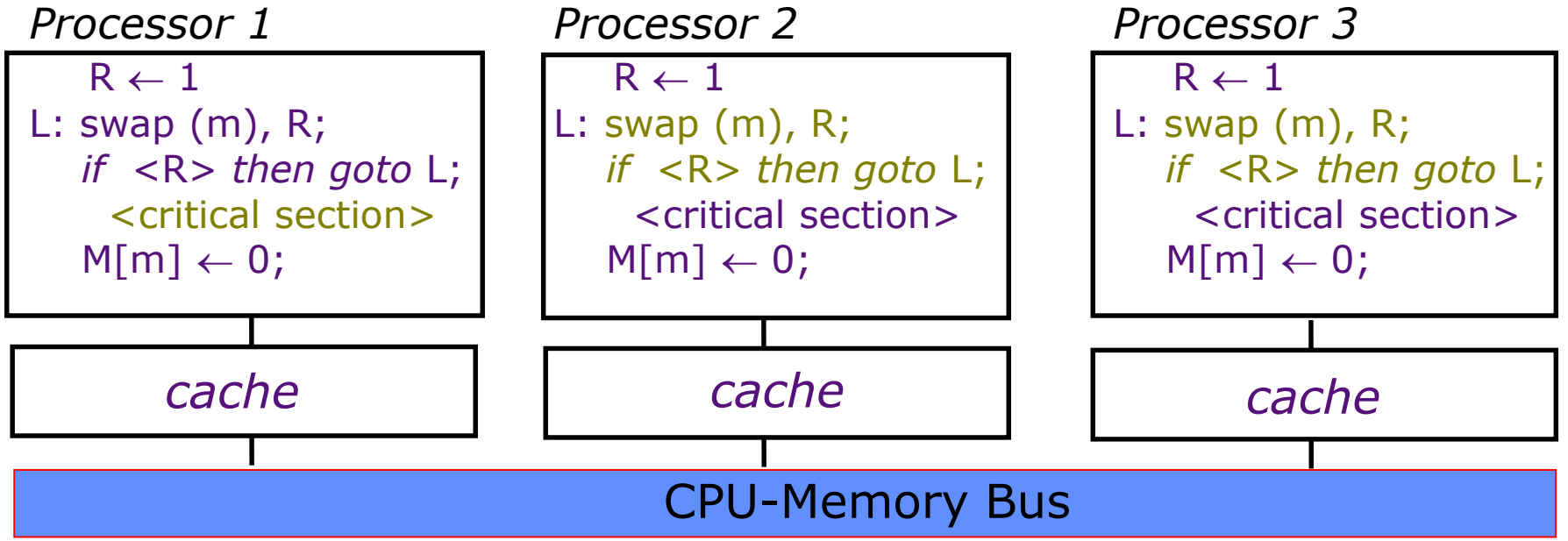


Cache-coherence protocols will cause lock to ping-pong between caches.

Ping-ponging can be reduced by first reading the lock location *(non-atomically)* and executing a swap only if it is found to be zero. For example, if we use *LL, SC* instructions

```
swap(m), R {
try:
        LL R1, (m)
        SC R, (m)
        BEQZ R, try
        R = R1
}
```

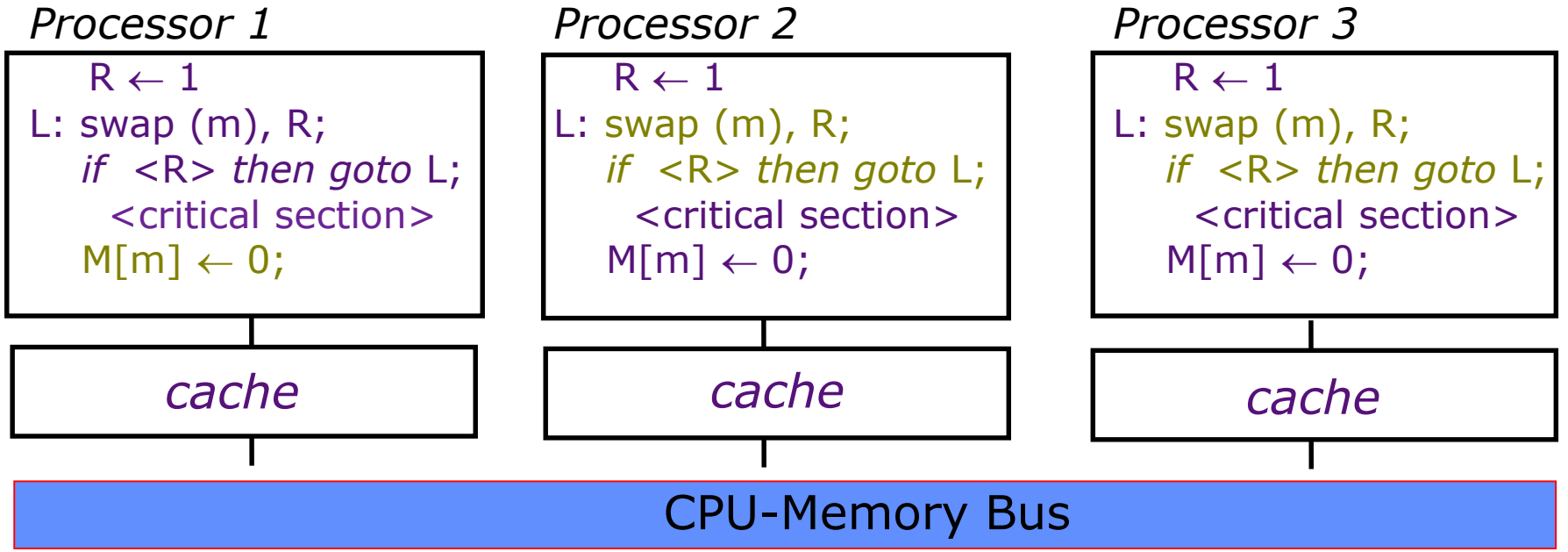
Synchronization and Caches



P1 has lock, P2 and P3 spin testing if lock is 0, lock is shared
Assume that status of cache line that contains the lock is (I, S, S),

```
swap(m), R {
try:
        LL R1, (m)
        SC R, (m)
        BEQZ R, try
        R = R1
}
```

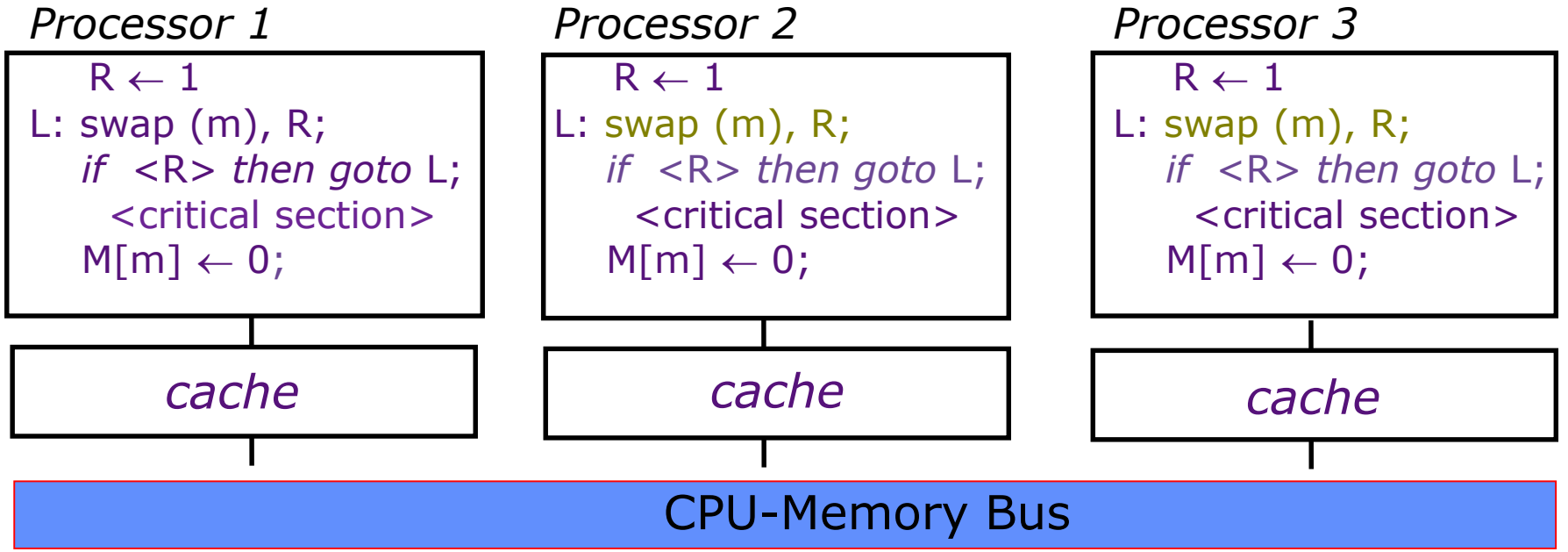
Synchronization and Caches



P1 sets lock to 0, P2 and P3 receive invalidate,
lock becomes Exclusive to P0
Status of cache line that contains the lock becomes (M, I, I)

```
swap(m), R {
try:
        LL R1, (m)
        SC R, (m)
        BEQZ R, try
        R = R1
}
```

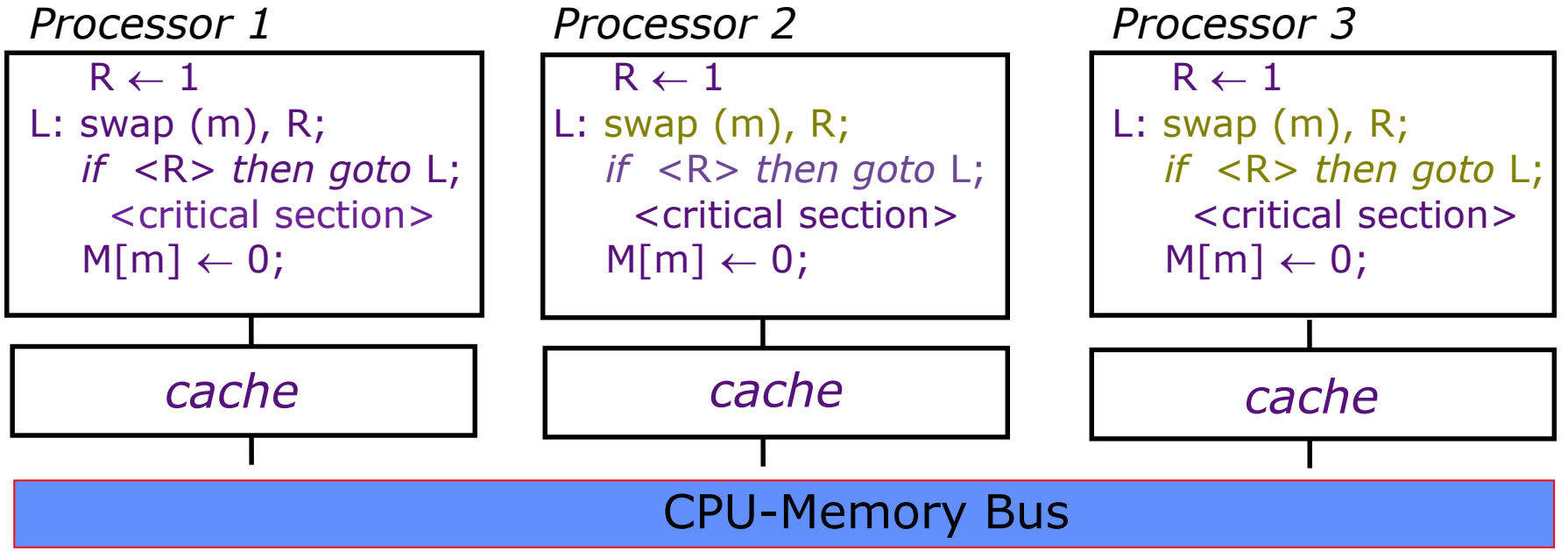
Synchronization and Caches



P2 and P3 race to read the lock (read miss)
Status of cache line that contains the lock becomes (S, S, S)

```
swap(m), R {
try:
        LL R1, (m)
        SC R, (m)
        BEQZ R, try
        R = R1
}
```

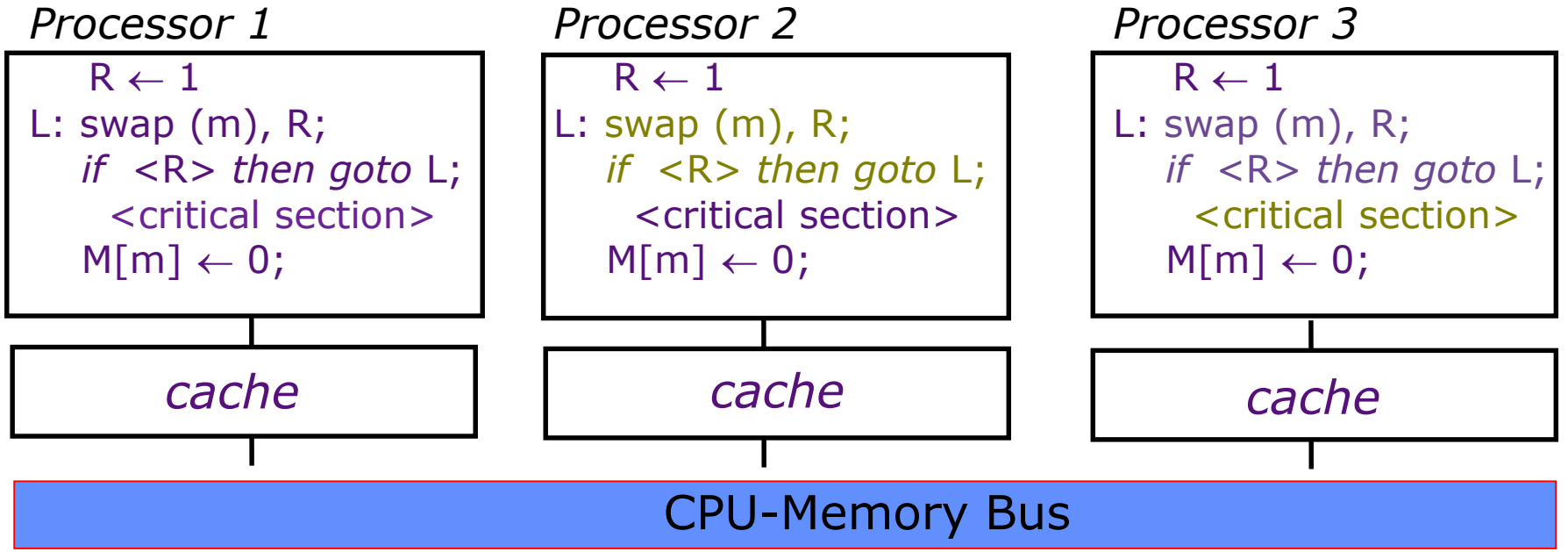
Synchronization and Caches




```
swap(m), R:
   Rt =  M[m];
   M[m] = R;
    R = Rt;
```

P3 is faster to test for 0 and to write 1 to the lock (write hit) and invalidate the other copies in P1,P2
lock becomes Exclusive to P3
Status of cache line that contains the lock becomes (I, I, M)

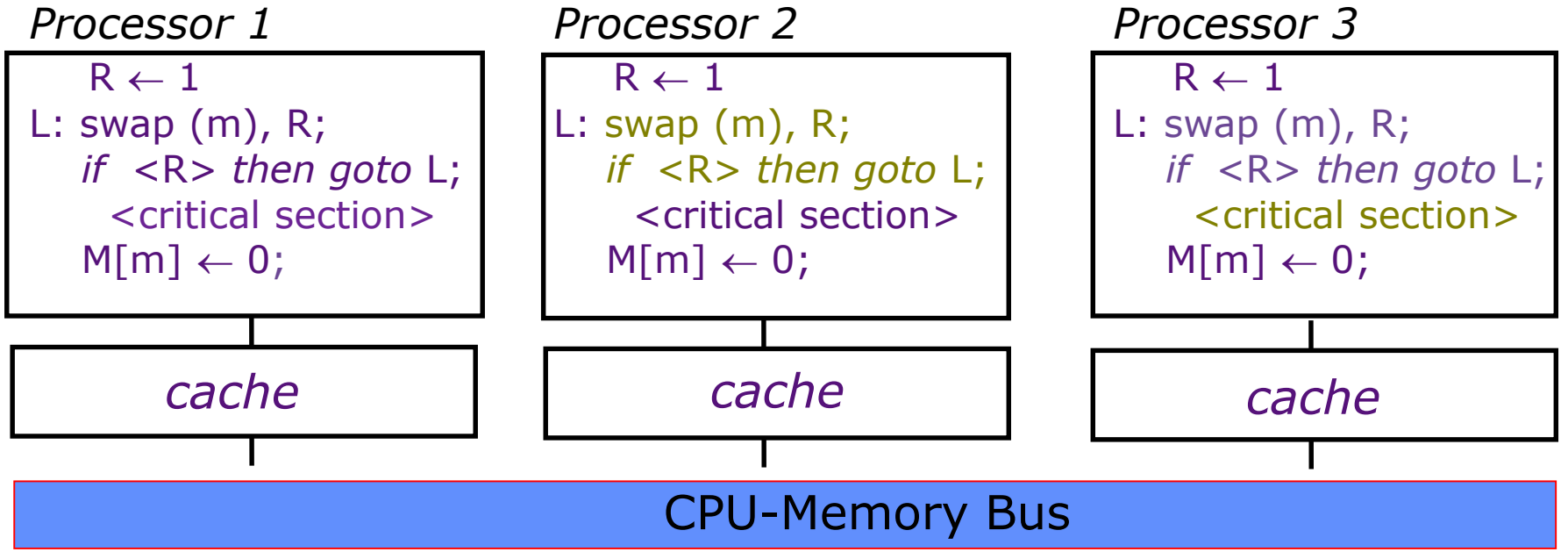
Synchronization and Caches



swap(m), R:
R_t = M[m];
M[m] = R;
R = R_t;

P3 enters critical region
P2 attempts to read the lock again (read miss), and gets the value 1. It sets the lock to 1 (write hit)
and lock becomes Exclusive to P2
Status of cache line that contains the lock becomes (I, M, I)

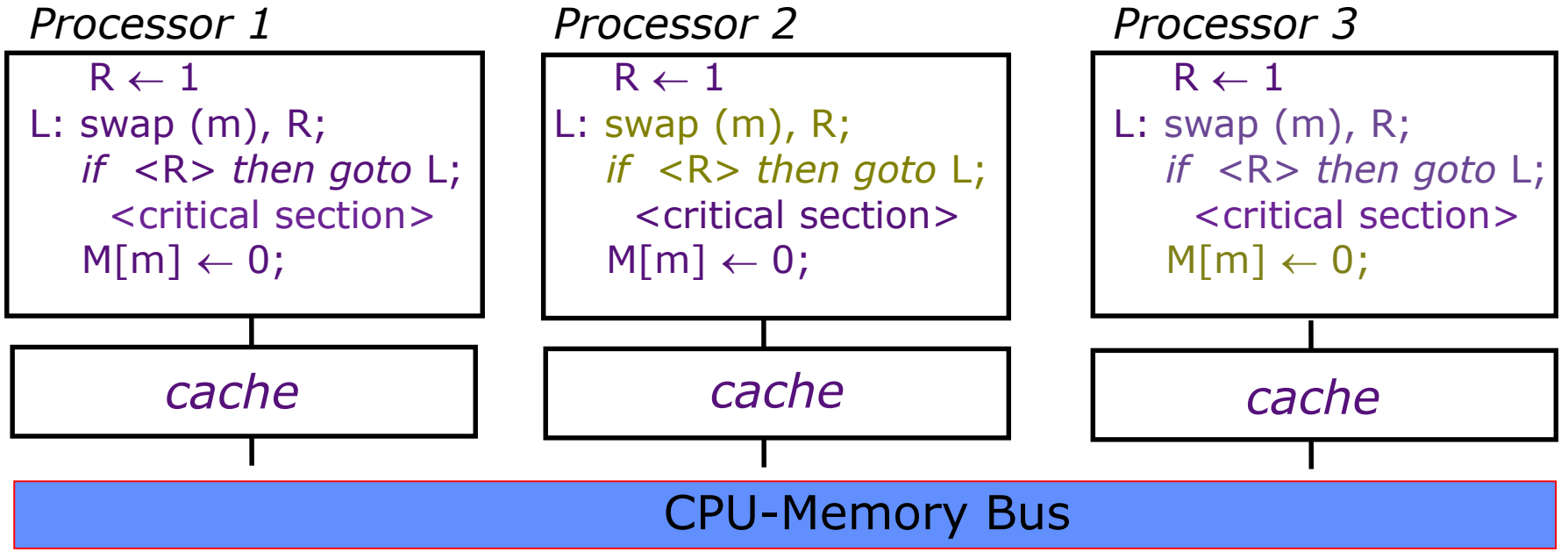
Synchronization and Caches



P2 spins testing if lock is 0. The lock remains Exclusive to P2
Status of cache line that contains the lock remains (I, M, I)

```
swap(m), R {
try:
        LL R1, (m)
        SC R, (m)
        BEQZ R, try
        R = R1
}
```

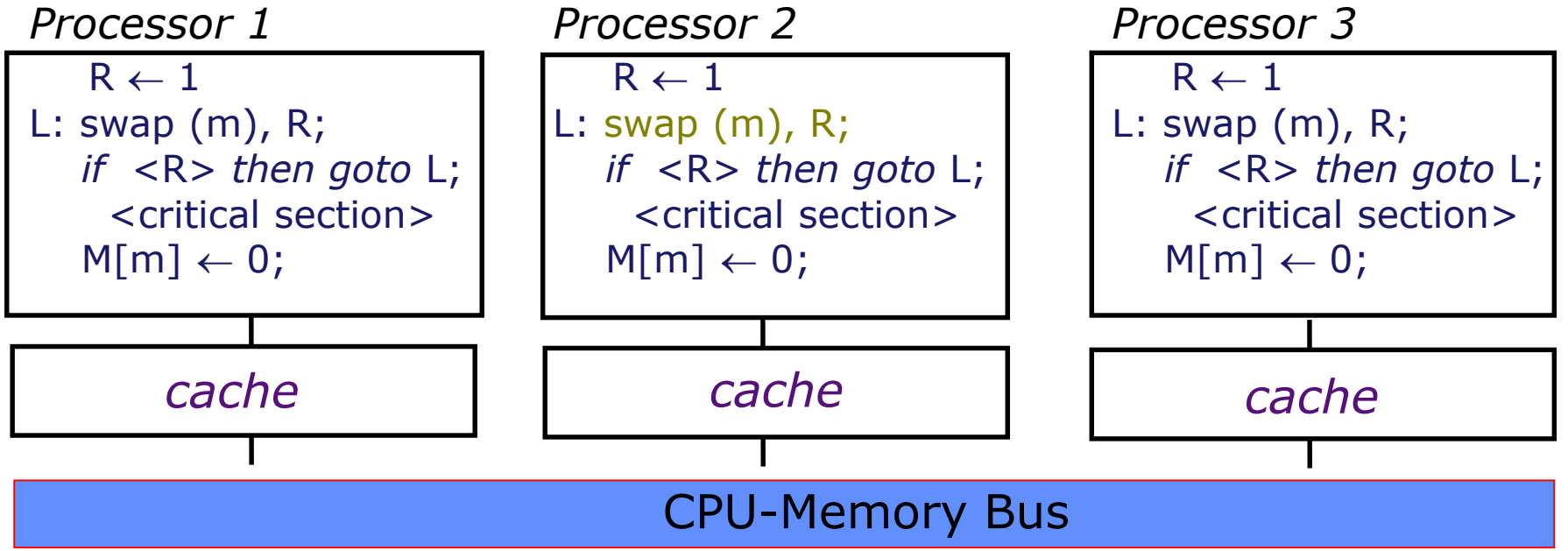
Synchronization and Caches



P3 sets lock to 0, P2 receives invalidate, lock becomes Exclusive to P3
Status of cache line that contains the lock becomes (I, I, M)

```
swap(m), R {
try:
        LL R1, (m)
        SC R, (m)
        BEQZ R, try
        R = R1
}
```

Synchronization and Caches



P2 reads the lock (I, S, S), find it equal to 0, and sets it to 1 (I, M, I). P2 enters the critical region.

```
swap(m), R {
try:
        LL R1, (m)
        SC R, (m)
        BEQZ R, try
        R = R1
}
```