HPC ARCHITECTURES

Cache Coherency

Cache coherency

- Main difficulty in building multiprocessor systems is the cache coherency problem.
- The shared memory programming model assumes that a shared variable has a unique value at a given time.
- Caching in a shared memory system means that multiple copies of a memory location may exist in the hardware.
- To avoid two processors caching different values of the same memory location, caches must be kept *coherent.*
- To achieve this, a write to a memory location must cause all other copies of this location to be removed from the caches they are in.

Coherence protocols

- Need to store information about sharing status of cache blocks
  - has this block been modified?
  - is this block stored in more than one cache?

- Two main types of protocol

1. Snooping (or broadcast) based
   - every cached copy caries sharing status
   - no central status
   - all processors can see every request
2. Directory based
   - sharing status stored centrally (in a directory)

Snoopy protocols

- Already have a valid tag on cache lines: this can be used for invalidation.
- Need an extra tag to indicate sharing status.
  - can use clean/dirty bit in write-back caches
- All processors monitor all bus transactions
  - if an invalidation message is on the bus, check to see if the block is cached, and if so invalidate it
  - if a memory read request is on the bus, check to see if the block is cached, and if so return data and cancel memory request.
- Many different possible implementations

3 state snoopy protocol: MSI

- Simplest protocol which allows multiple copies to exist
- Each cache block can exist in one of three states:
  - ***M**odified*: this is the only valid copy in any cache and its value is different from that in memory
  - ***S**hared*: this is a valid copy, but other caches may also contain it, and its value is the same as in memory
  - ***I**nvalid*: this copy is out of date and cannot be used.
- Model can be described by a state transition diagram.
  - state transitions can occur due to actions by the processor, or by the bus.
  - state transitions may trigger actions

Processor actions

- read (PrRd)
- write (PrWr)

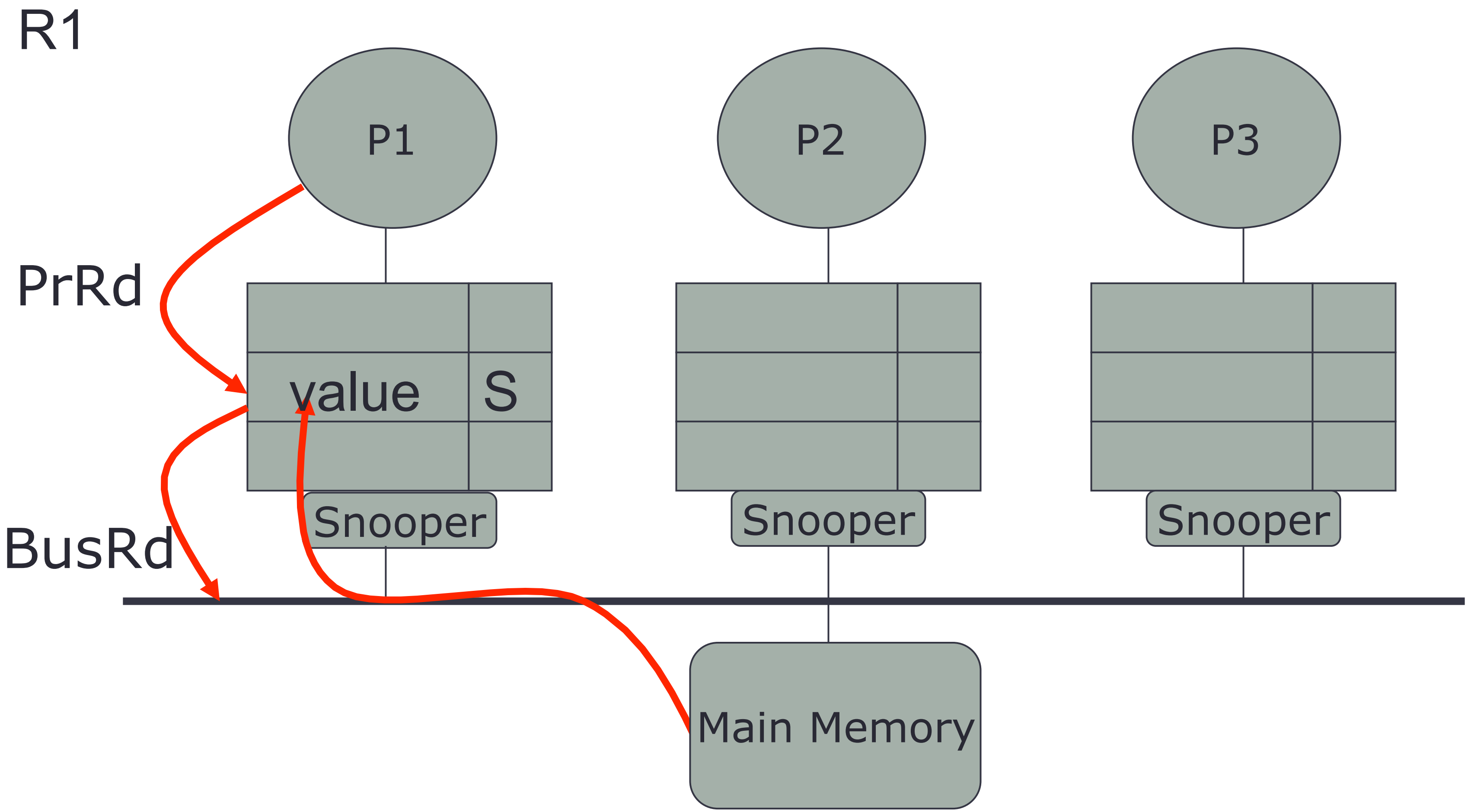
Bus actions

- read (BusRd)
- read exclusive (BusRdX)
- flush to memory (Flush)

MSI Protocol walk through

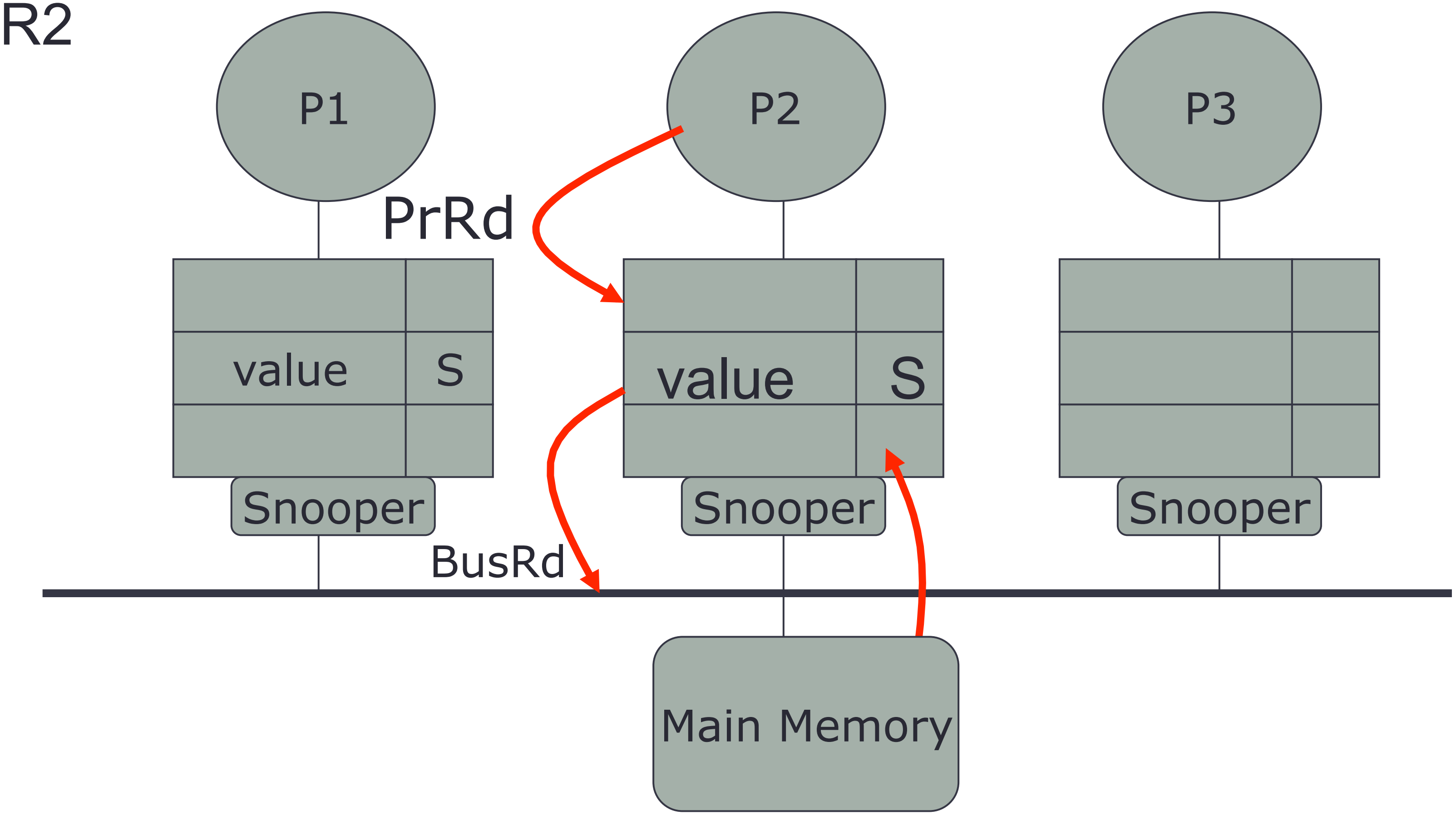
- Assume we have three processors.
- Each is reading/writing the same value from memory where R1 means a read by processor 1 and W3 means a write by processor 3.
- For simplicity sake, the memory location will be referred to as “value.”
- The memory access stream we will walk through is:

R1, R2, W3, R2, W1, W2, R3, R2



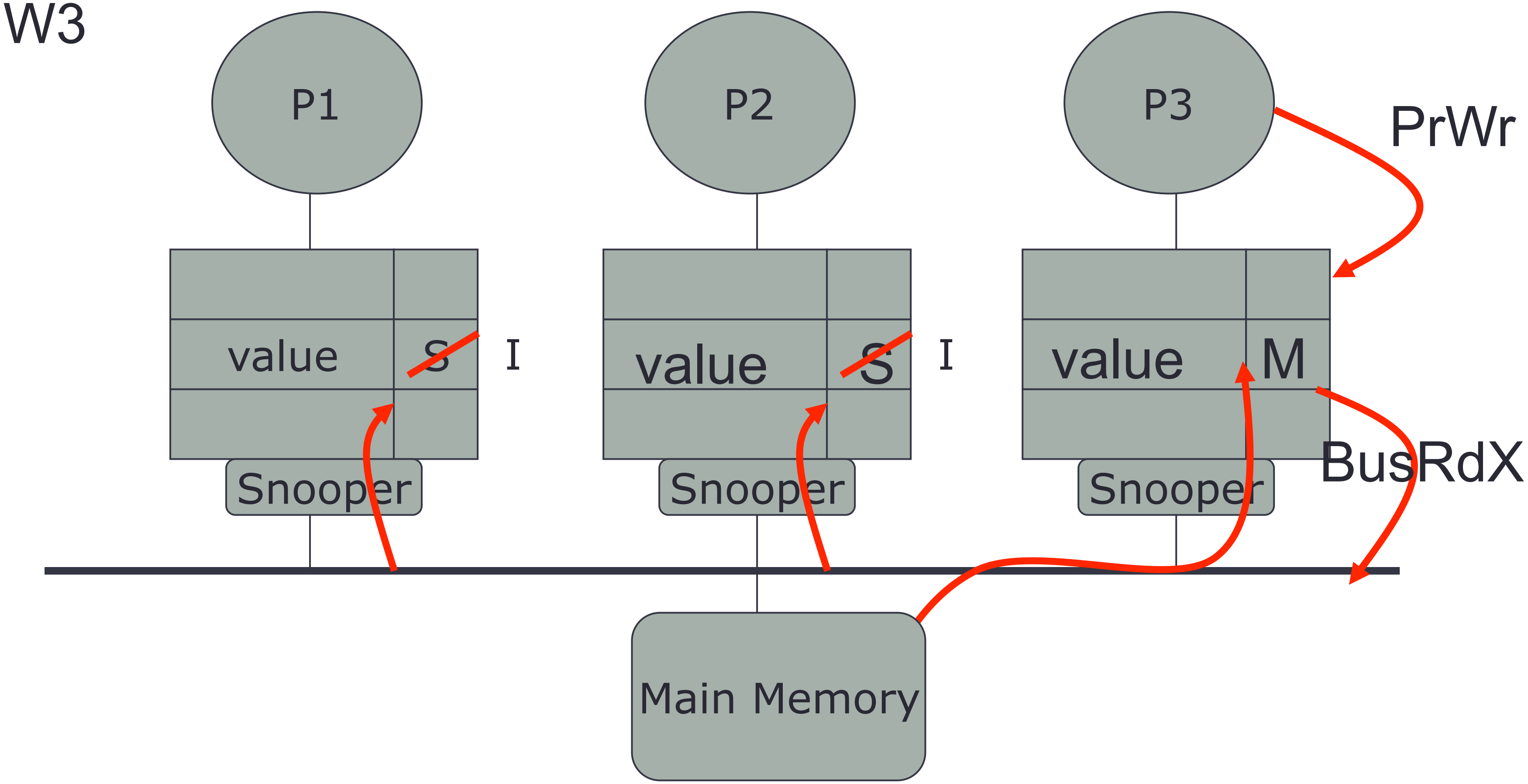
P1 wants to read the value. The cache does not have it and generates a BusRd for the data. Main memory controller provides the data. The data goes into the cache in the shared state.

R2



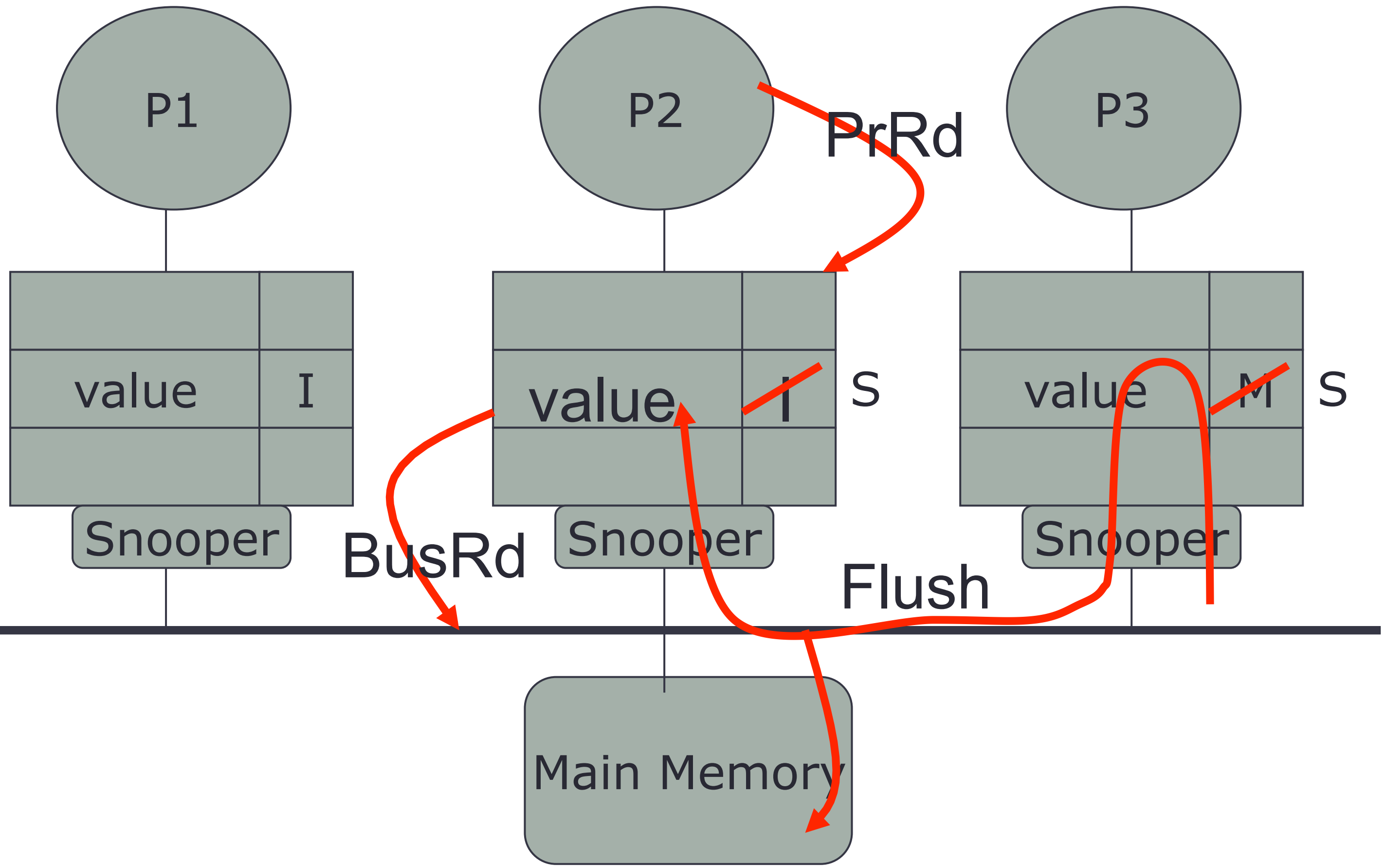
P2 wants to read the value. Its cache does not have the data, so it places a BusRd to notify other processors and ask for the data. The memory controller provides the data.

W3



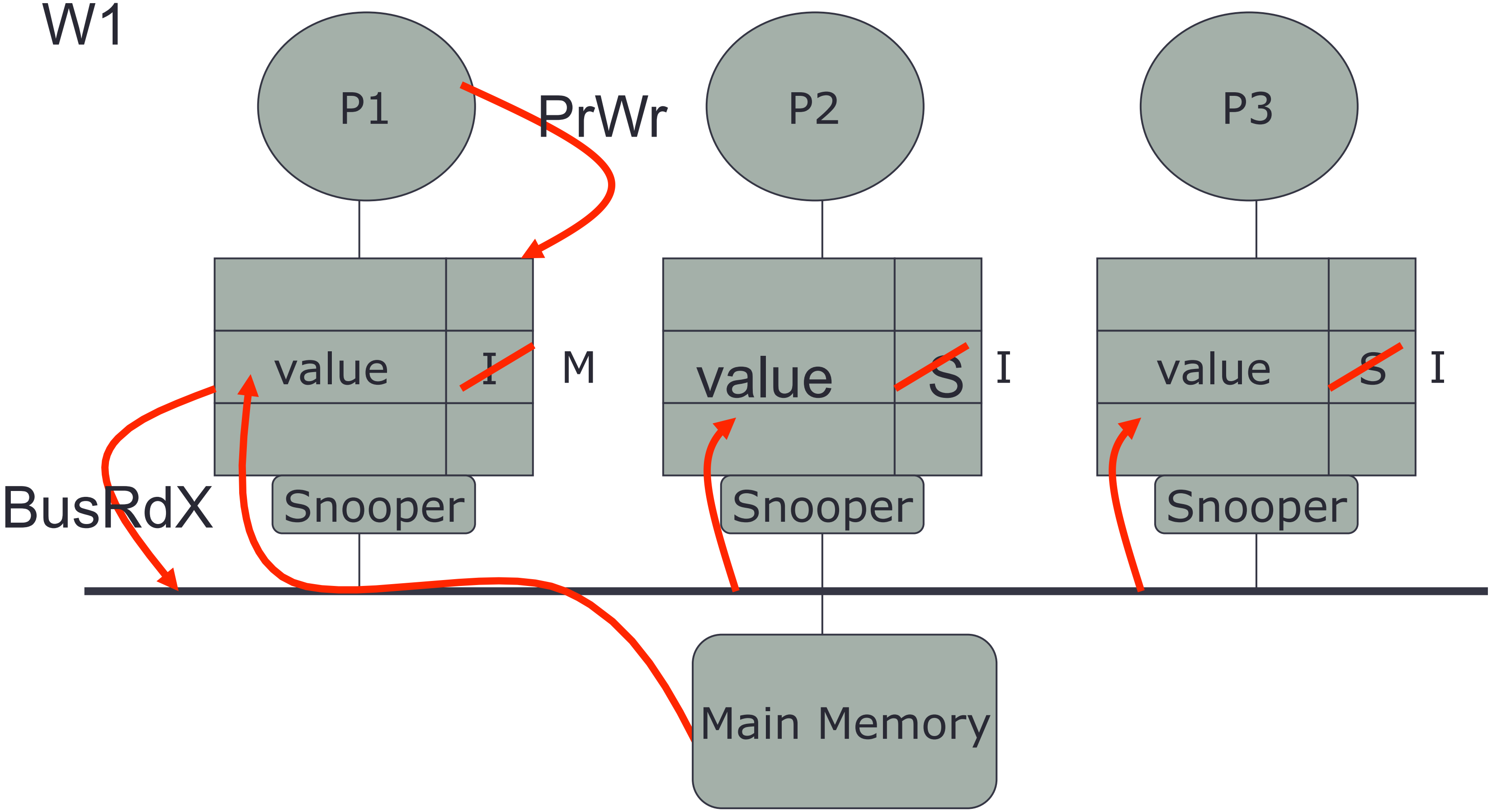
P3 wants to write the value. It places a BusRdX to get exclusive access and the most recent copy of the data. The caches of P1 and P2 see the BusRdX and invalidate their copies. Because the value is still up-to-date in memory, memory provides the data.

R2



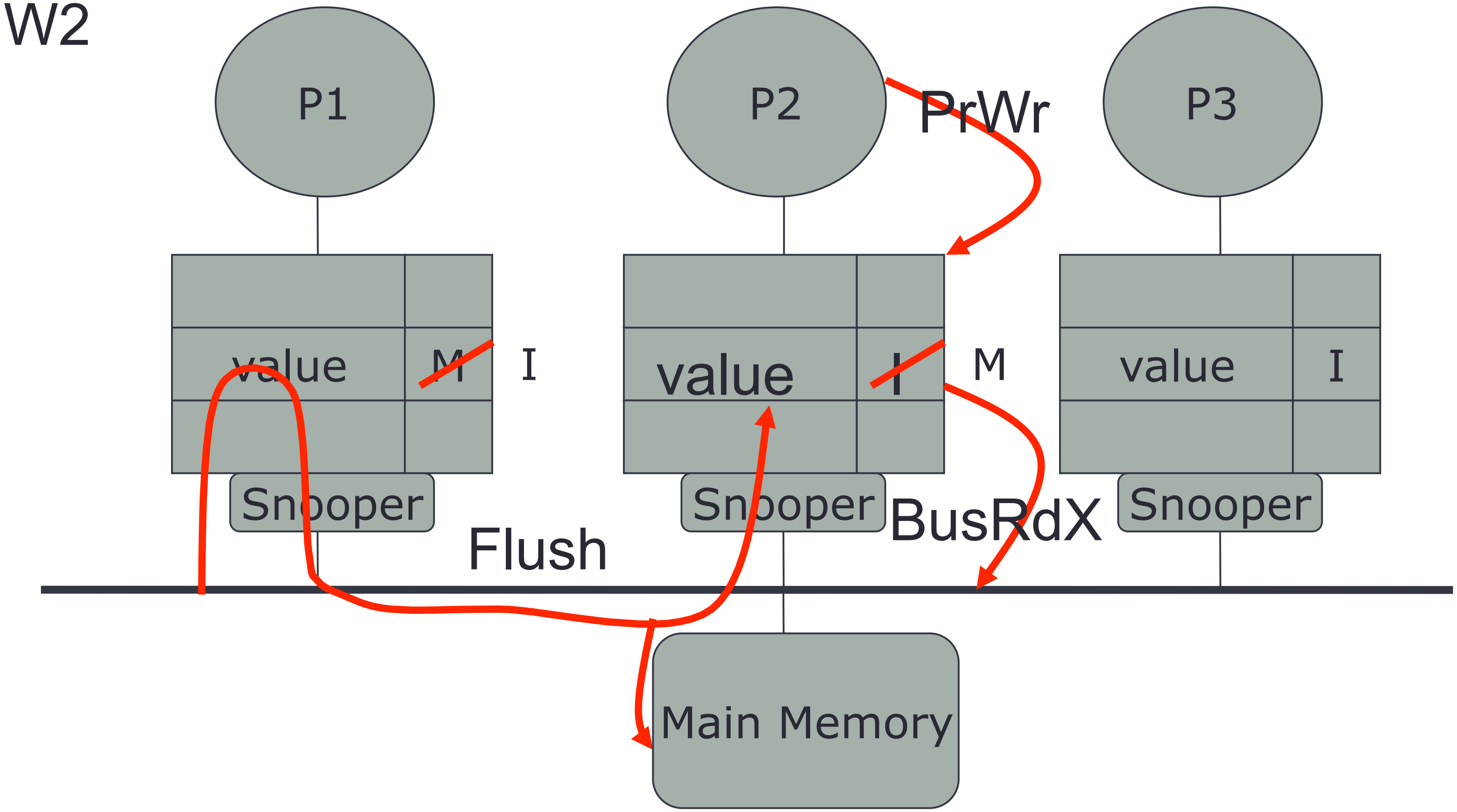
P2 wants to read the value. P3’s cache has the most up-to-date copy and will provide it. P2’s cache puts a BusRd on the bus. P3’s cache snoops this and cancels the memory access because it will provide the data. P3’s cache flushes the data to the bus.

W1



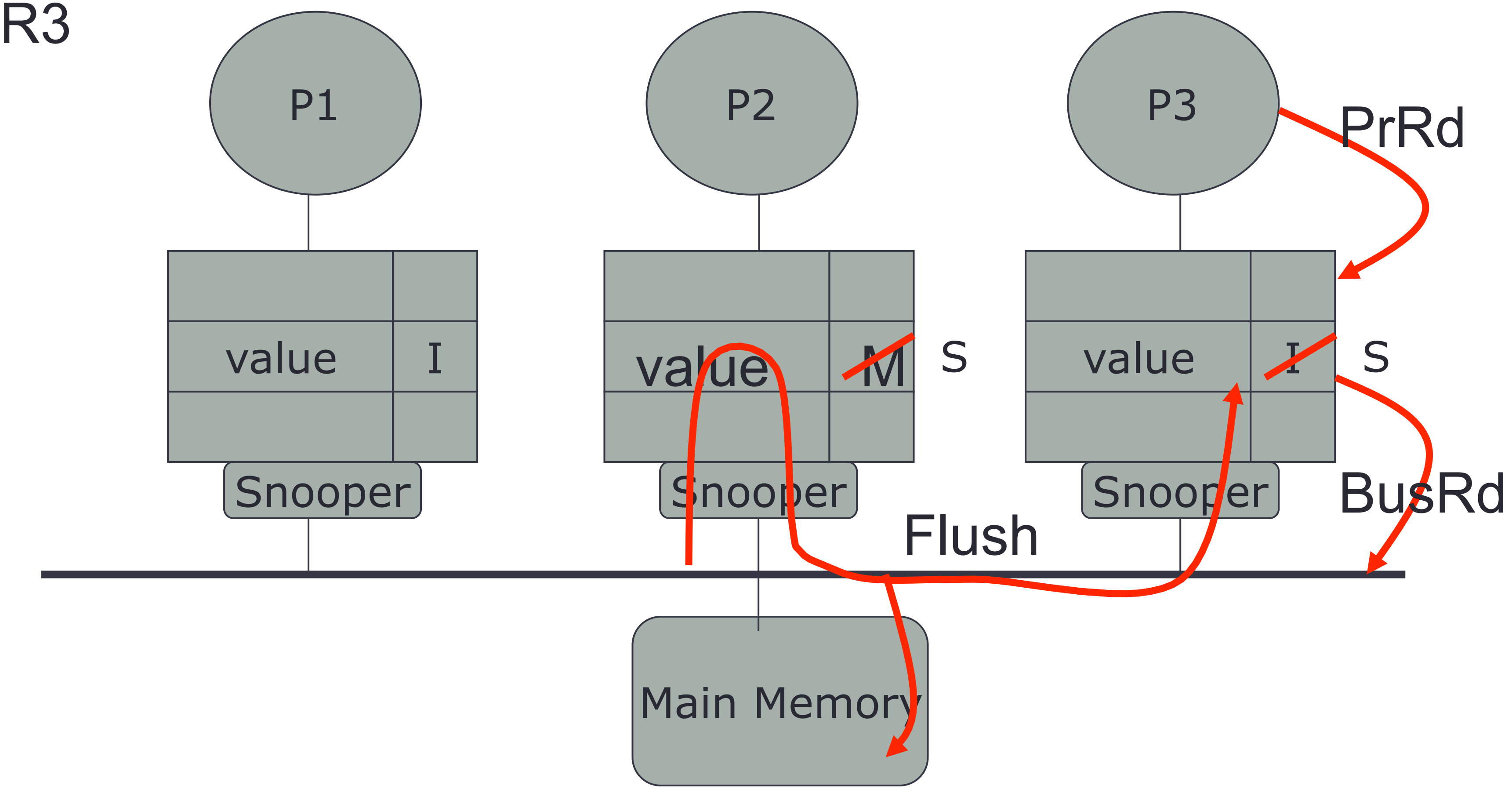
P1 wants to write to its cache. The cache places a BusRdX on the bus to gain exclusive access and the most up-to-date value. Main memory is not stale so it provides the data. The snoopers for P2 and P3 see the BusRdX and invalidate their copies in cache.

W2



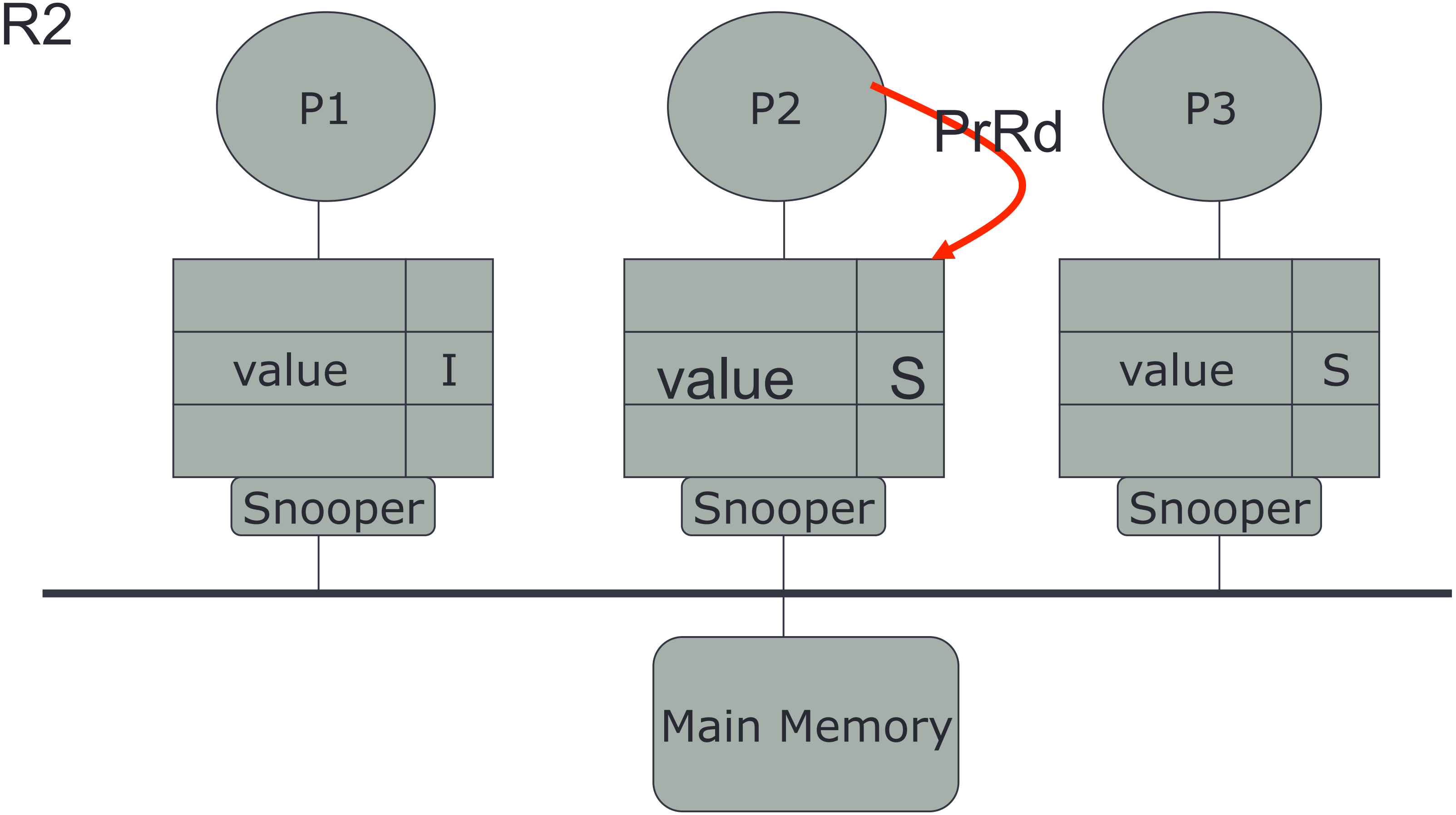
P2 wants to write the value. Its cache places a BusRdX to get exclusive access and the most recent copy of the data. P1's snooper sees the BusRdX and flushes the data to the bus. Also, it invalides the data in its cache and cancels the memory access.

R3



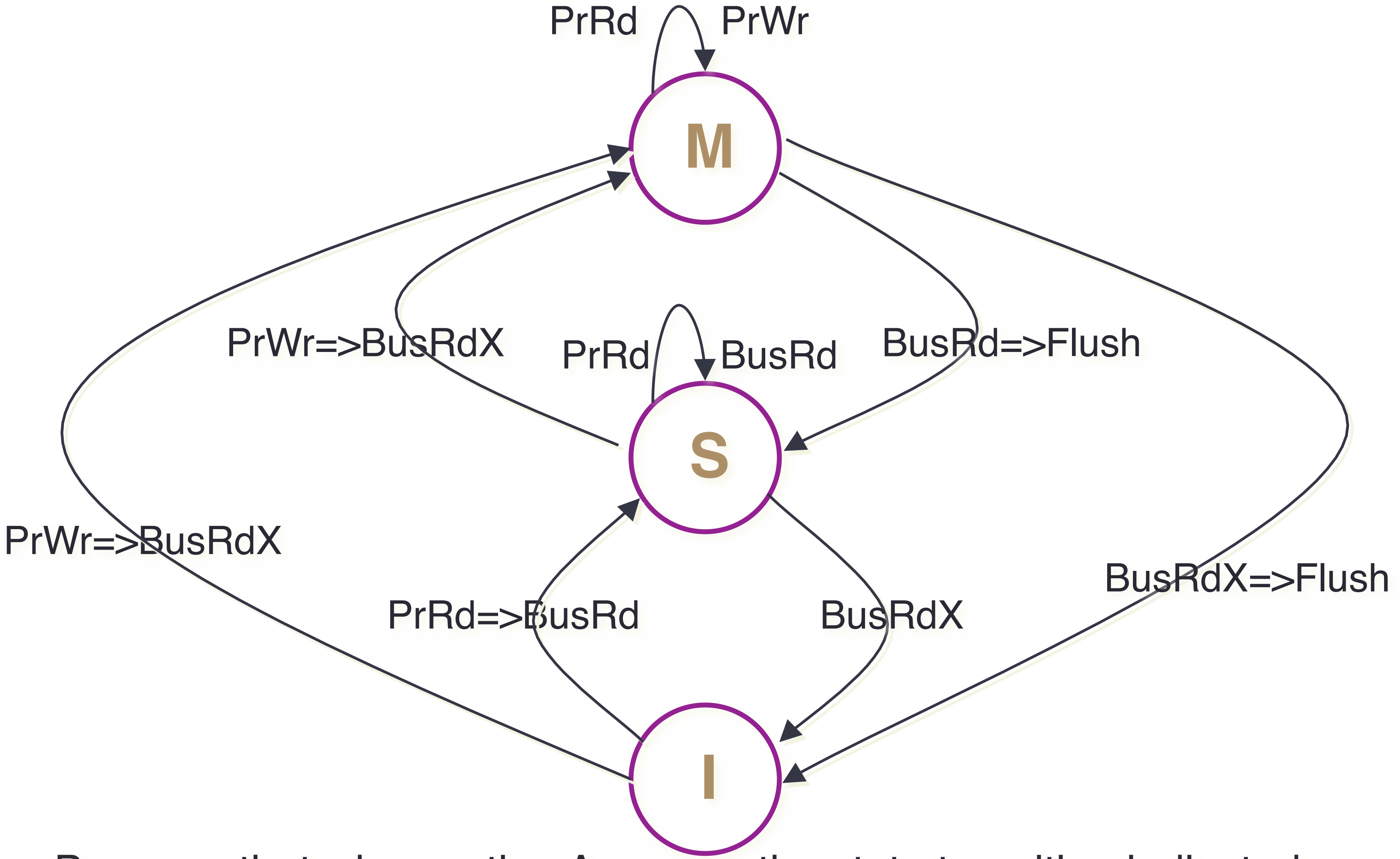
P3 wants to read the value. Its cache does not have a valid copy, so it places a BusRd on the bus. P2 has a modified copy, so it flushes the data on the bus and changes the status of the cache data to shared. The flush cancels the memory access and updates the data in memory as well.

R2



P2 wants to read the value. Its cache has an up-to-date copy. No bus transactions need to take place as there is no cache miss.

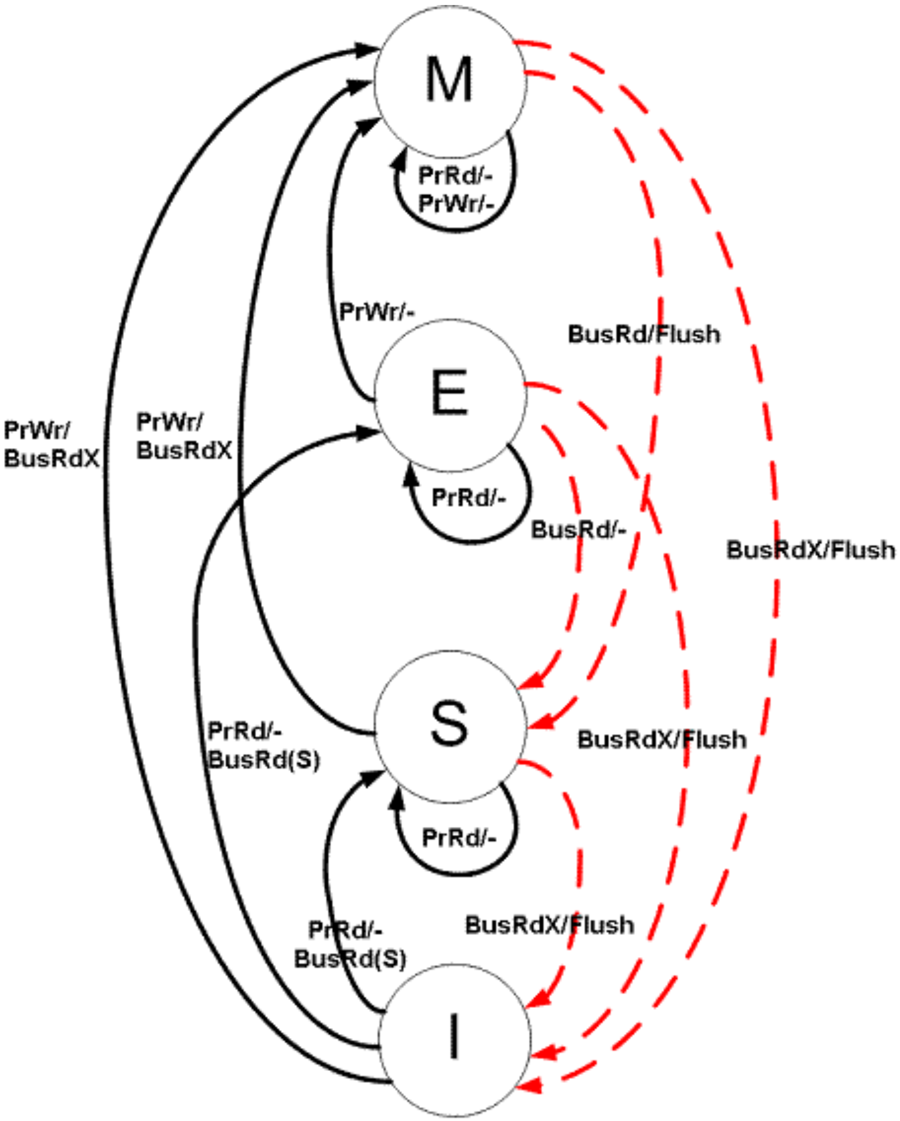
MSI state transitions



A=>B means that when action A occurs, the state transition indicated happens, and action B is generated

Other protocols

- MSI is inefficient: it generates more bus traffic than is necessary
- Can be improved by adding other states, e.g.
  - ***E**xclusive*: this copy has not been modified, but it is the only copy in any cache
  - ***O**wned*: this copy has been modified, but there may be other copies in shared state
- MESI and MOESI protocols are more commonly used protocols than MSI
- MSI is nevertheless a useful mental model for the programmer
- Also possible to update values in other caches on writes, instead of invalidating them


M
PrRd/-
PrWr/-
PrWr/-
E
PrRd/-
BusRd/Flush
BusRd/-
BusRdX/Flush
PrWr/
BusRdX
PrWr/
BusRdX
S
PrRd/-
BusRd(S)
BusRdX/Flush
PrRd/-
PrRd/-
BusRd(S)
BusRdX/Flush
I

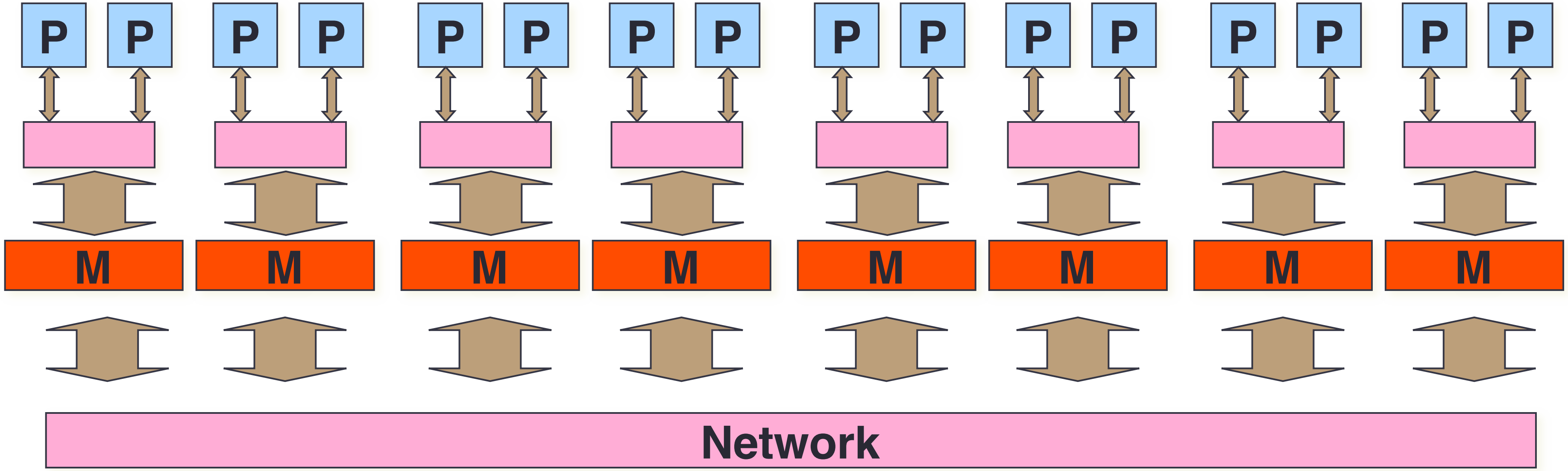
False sharing

- The units of data on which coherency operations are performed are cache blocks: the size of these units is usually 64 or 128 bytes.
- The fact that coherency units consist of multiple words of data gives rise to the phenomenon of *false sharing.*
- Consider what happens when two processors are both writing to *different* words on the *same* cache line.
  - no data values are actually being shared by the processors
- Each write will invalidate the copy in the other processor's cache, causing a lot of bus traffic and memory accesses.
  - same problem if one processor is writing and the other reading
- Can be a significant performance problem in threaded programs
- Quite difficult to detect

Distributed shared memory

- Shared memory machines using buses and a single main memory do not scale to large numbers of processors
  - bus and memory become a bottleneck
- Distributed shared memory machines designed to:
  - scale to larger numbers of processors
  - retain a single address space
- Modest sized multi-socket systems connected with HyperTransport or QPI are, in fact, distributed shared memory

Distributed shared memory



Directory based coherency

- For scalability, there is no bus, so snooping is not possible
- Instead use a directory structure
  - bit vector for every block
  - one bit per processor
  - stored in (distributed) memory
  - bit is set to 1 whenever the corresponding processor caches the block.
- Still some scalability issues:
  - directory takes up a lot of space for large machines
  - e.g. 128 byte cache block, 256 processors: directory is 20% of memory
  - some techniques to get round this

Implementation

- Node where memory (and directory entry) is located is called the home node.
- Basic principal is same as snoopy protocol
  - cache block has same 3 states (modified, shared, invalid)
  - directory entry has modifed, shared and uncached states.
- Cache misses go to the home node for data, and directory bits are set accordingly for read/write misses.
- Directory can:
  - invalidate a copy in a remote cache
  - fetch the data back from a remote cache
- Cache can write back to home node.

cc-NUMA

- We have described a distributed shared memory system where every memory address has a home node.
- This type of system is known a a cache-coherent non-uniform memory architecture (cc-NUMA).
- Main problem is that access to remote memories take longer than to local memory
  - difficult to determine which is the best node to allocate given page on
- OS is responsible for allocating pages
- Common policies are:
  - first touch: allocate on node which makes first access to the page
  - round robin: allocate cyclically

Migration and replication

- Possible for the OS to move pages between nodes as an application is running
- Pages can either be migrated or replicated.
- Migration involves the relocation of a page to a new home node.
- Replication involves the creation of a “shadow” of the page on another node.
    - read miss can go to the shadow page
- Cache coherency is still maintained by hardware on a cache block basis.