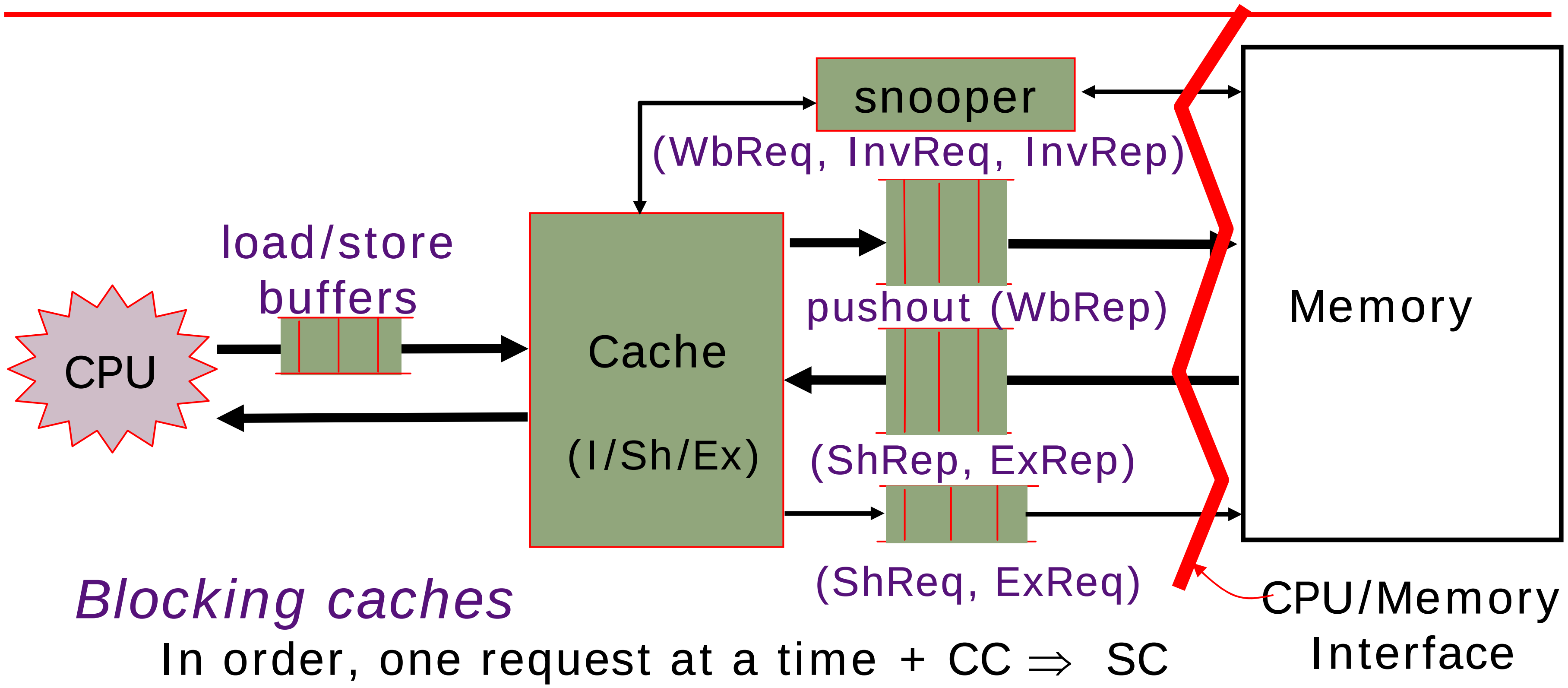
Cache Coherence Protocols for Sequential Consistency

Arvind
Computer Science and Artificial Intelligence Lab
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic

Systems view



Blocking caches

In order, one request at a time + CC ⇒ SC

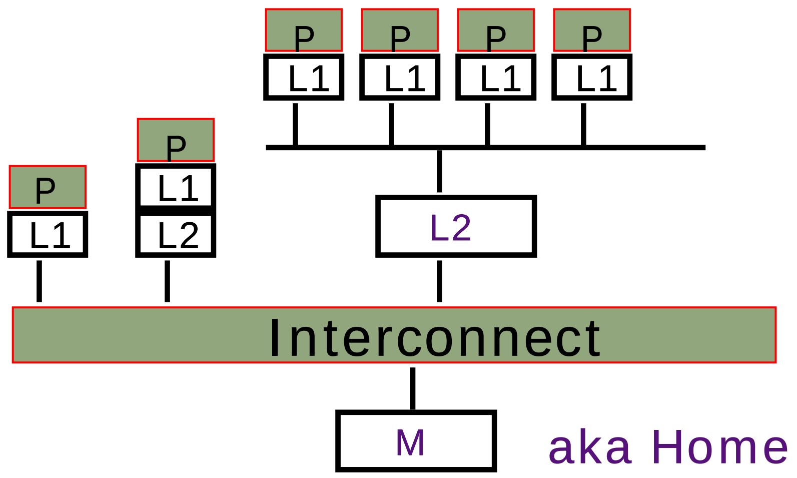
Non-blocking caches

Multiple requests (different addresses) concurrently + CC

⇒ Relaxed memory models

CC ensures that all processors observe the same order of loads and stores to an address

A System with Multiple Caches



Assumptions: Caches are organized in a hierarchical manner

- Each cache has exactly one parent but can have zero or more children
- Only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

$$a \in L_i \quad \Rightarrow \quad a \in L_{i+1}$$

Maintaining Cache Coherence

Hardware support is required such that

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

⇒

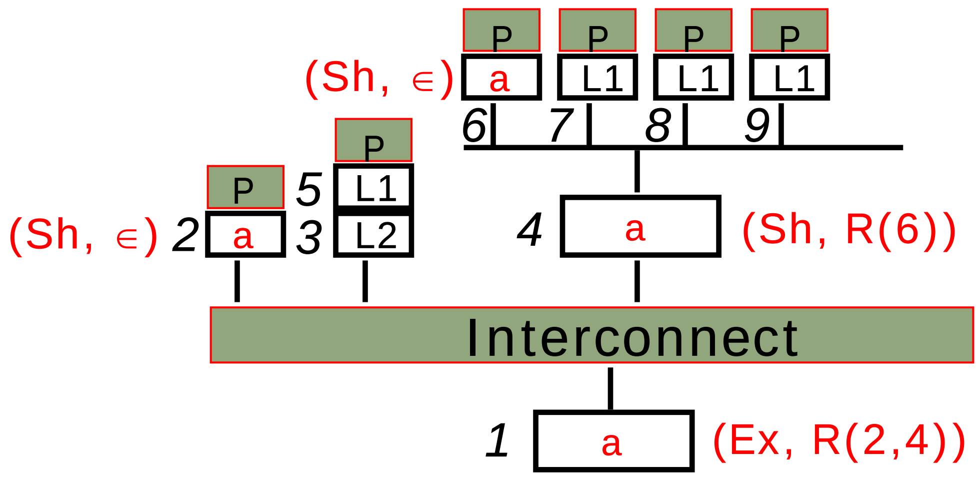
write request:

The address is *invalidated* in all other caches *before* the write is performed

read request:

If a dirty copy is found in some cache, a write-back is performed before the memory is read

State Encoding



Each address in a cache keeps two types of state info

- *sibling info:* do my siblings have a copy of address a
  - Ex (means no), Sh (means may be)
- *children info:* has this address been passed on to any of my children
  - W(id) means child id has a writable version
  - R(dir) means only children named in the *directory* dir have copies

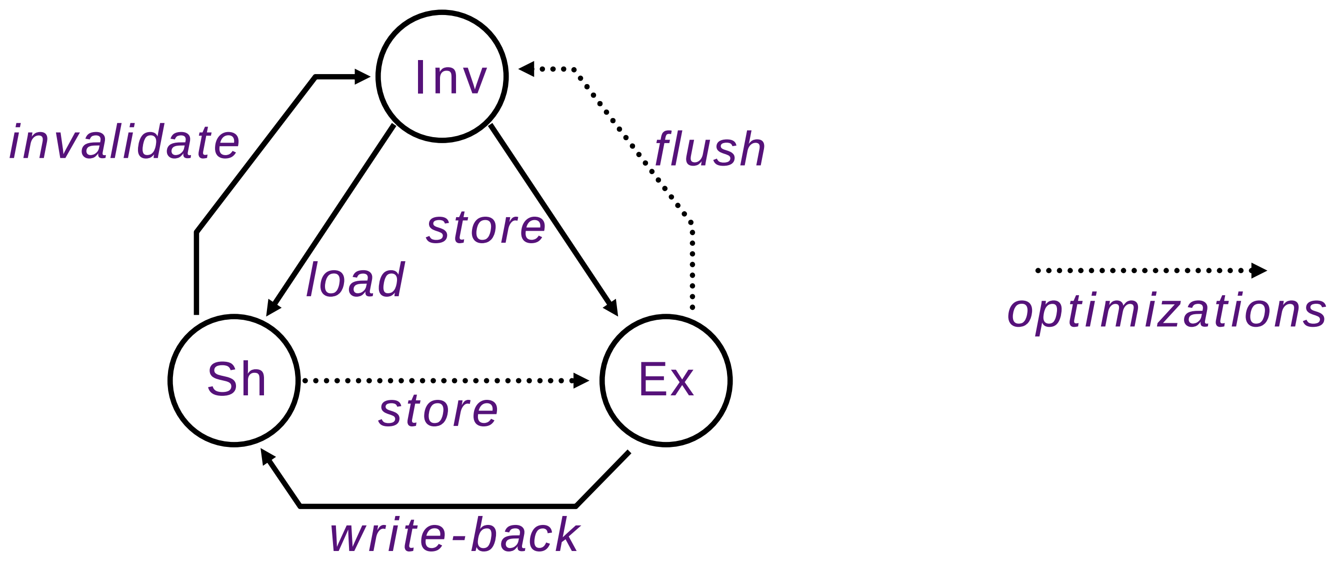
Cache State Implications

Sh ⇒ cache's siblings and decedents can only have Sh copies

Ex ⇒ each ancestor of the cache must be in Ex
⇒ *either* all children can have Sh copies
or one child can have an Ex copy

- Once a parent gives an Ex copy to a child, the parent's data is considered stale
- A processor cannot overwrite data in Sh state in L1
- By definition all addresses in the home are in the Ex state

Cache State Transitions



This state diagram is helpful as long as one remembers that each transition involves cooperation of other caches and the main memory.

High-level Invariants in Protocol Design

Guarded Atomic Actions

- Rules specified using guarded atomic actions:

  <guard predicate>

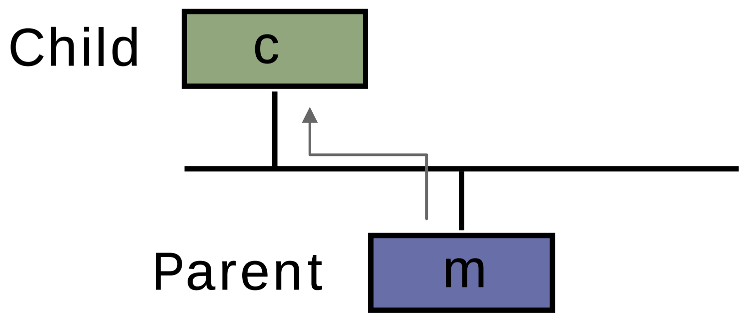
  $\rightarrow$ {set of state updates that must occur atomically with respect to other rules}

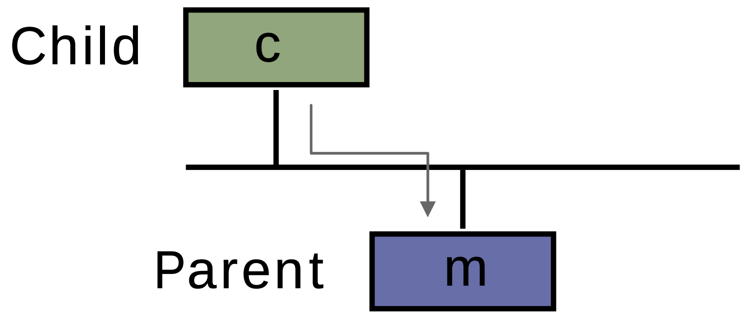
- E.g.:

  m.state(a) == R(dir) & $id_c \notin$ dir

  $\rightarrow$ m.setState(a, R(dir+ id_c)),
  c.setState(a, Sh); c.setData(a, m.data(a));

Data Propagation Between Caches





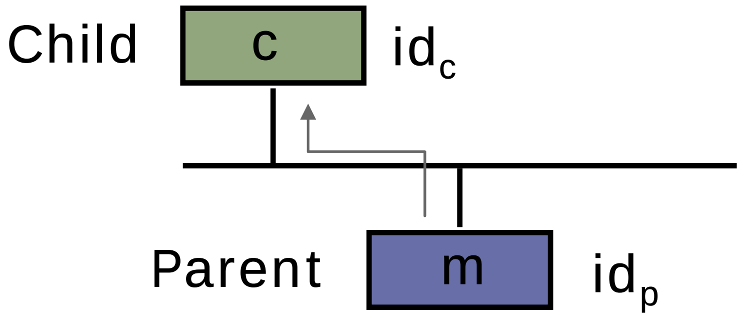
Caching rules

- *Read caching rule*
- *Write caching rule*

De-caching rules

- *Write-back rule*
- *Invalidate rule*

Caching Rules: *Parent to Child*



- Read caching rule

  $R(dir) == m.state(a) \quad \& \quad id_c \notin dir$

  $\rightarrow$ m.setState(a, R(dir+ id_c))

  c.setState(a, Sh); c.setData(a, m.data(a));

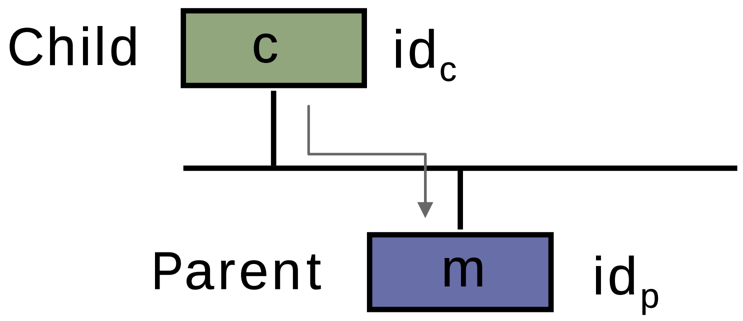
- Write caching rule

  $\varepsilon == m.state(a)$

  $\rightarrow$ m.setState(a, W(id_c))

  c.setState(a, Ex); c.setData(a, m.data(a));

De-caching Rules: *Child to Parent*



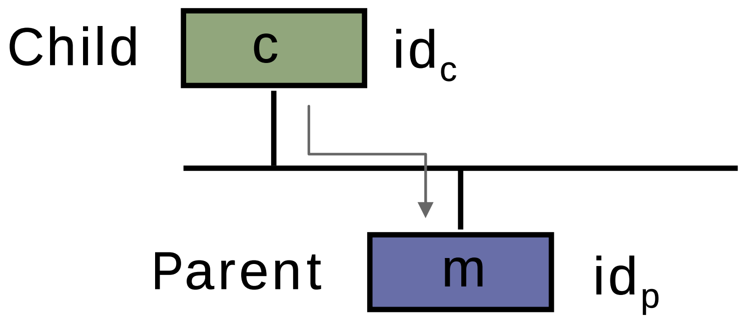
- Writeback rule

```
    W(id_c) == m.state(a)    &  Ex == c.state(a)
→   m.setState(a, R({id_c}))
    msetData(a, c.data(a));
    c.setState(a, Sh);
```

- Invalidate rule

```
    R(dir) == m.state(a) &  id_c ∈ dir &  Sh == c.state(a)
→   m.setState(a, R(dir - id_c))
    c.invalidate(a);
```

Making the Rules *Local & Reactive*

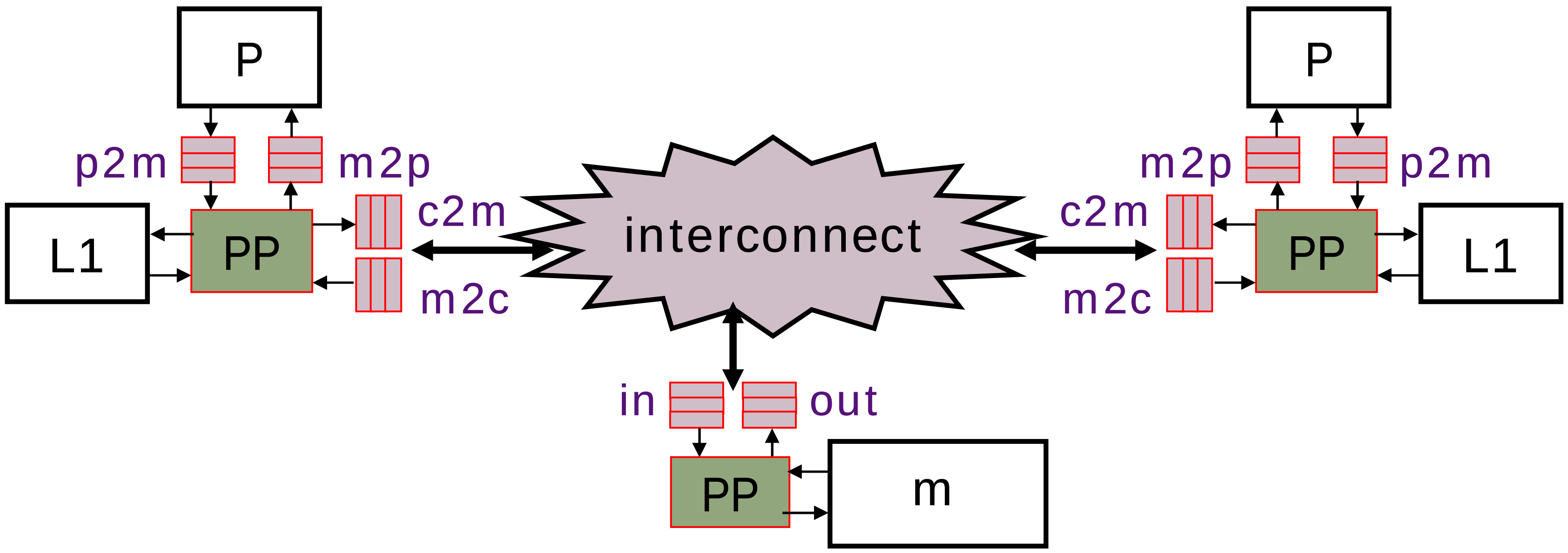


- Some rules require observing and changing the state of multiple caches simultaneously (atomically).
  - very difficult to implement, especially if caches are separated by a network
- Each rule must be triggered by some action
- Split rules are into multiple rules – "request for an action" followed by "an action and an ack".
  - ultimately all actions are triggered by some processor

Protocol Design

Note
We will not be able to finish this part today.
(The rest of the material will be covered during the next lecture.)

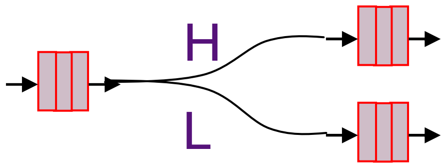
Protocol Processors *an abstract view*



- Each cache has 2 pairs of queues
  - one pair (c2m, m2c) to communicate with the memory
  - one pair (p2m, m2p) to communicate with the processor
- Messages format:
  msg(idsrc,iddest,cmd,priority,a,v)
- FIFO messages passing between each (src,dest) pair except a Low priority (L) msg cannot block a high priority (H) msg

H and L Priority Messages

- At the memory unprocessed requests cannot block the result messages. Hence all messages are classified as H or L priority.
  - all messages carrying results are classified as high priority
- Accomplished by having separate paths for H and L priority
  - In Theory: separate networks
  - In Practice:
    - Separate Queues
    - Shared buses for both networks



A Protocol for a system with two memory levels (L1 + M)

Cache states: Sh, Ex, Pending, Nothing

Memory states: R(dir), W(id), T_R (dir), T_W(id)

If dir is empty then R(dir) and T_R(dir) represent the same state

Messages:

Cache to Memory requests: ShReq, ExReq

Memory to Cache requests: WbReq, InvReq, FlushReq

Cache to Memory responses: WbRep(v), InvRep, FlushRep(v)

Memory to Cache responses: ShRep(v), ExRep(v)

Operations on cache:

cache.state(a) – returns state s

cache.data(a) - returns data v

cache.setState(a,s), cache.setData(a,v), cache.invalidate(a)

inst = first(p2m); msg= first(m2c); mmsg = first(in)

Voluntary rules: Cache must be able to evict values to create space

It would be good to have “silent drops” but difficult in a directory-based protocol

Invalidate rule

```
        cache.state(a) is Sh
→       cache.invalidate(a)
        c2m.enq (Msg(id, Home, InvRep, a))
```

Flush rule

```
        cache.state(a) is Ex
→       cache.invalidate(a)
        c2m.enq (Msg(id, Home, FlushRep, a, cache.data(v))
```

Writeback rule

```
        cache.state(a) is Ex
→       cache.setState(a, Sh)
        c2m.enq (Msg(id, Home, WbRep, a, cache.data(v)))
```

This rule may be applied if the cache/processor knows it is the “last store” operation to the location.

Such voluntary rules can be used to construct adaptive protocols.

Voluntary rules: Memory should be able to send more values than requested

Cache Rule

```
        m.state(a) is R(dir) & id ∉ dir
→       m.setState(a, R(id+dir))
        out.enq(Msg(Home,id,ShRep, a,m.data(a)))
```

It is a rule like this that allows us to fetch locations a+1, a+2, ... when a processor requests address a.

Five-minute break to stretch your legs

Load Rules (at cache)

- Load-hit rule

```
      Load(a)==inst
   &  cache.state(a) is Sh or Ex
→     p2m.deq
      m2p.enq(cache.data(a))
```

- Load-miss rule

```
      Load(a)==inst
   &  cache.state(a) is Nothing
→     c2m.enq(Msg(id, Home, ShReq, a)
      cache.setState(a,Pending)
```

This is blocking cache because the Load miss rule does not remove the request from the input queue (p2m) ... *more later*

Store Rules (at cache)

- Store-hit rule

```
      Store(a,v)==inst
  &   cache.state(a) is Ex
→     p2m.deq;
      m2p.enq(Ack)
      cache.setData(a, v)
```

- Store-miss rules

```
      Store(a,v)==inst
  &   cache.state(a)  is Nothing
→     c2m.enq(Msg(id, Home, ExReq, a);
      cache.setState(a,Pending)

      Store(a,v)==inst
  &   cache.state(a)  is Sh
→     c2m.enq(Msg(id, Home, InvRep, a);
      cache.setState(a,Nothing)
```

Already covered by the Invalidate voluntary rule

Processing ShReq Messages (at Home)

Uncached or Outstanding Shared Copies

```
        Msg(id,Home,ShReq,a) == m msg
    &   m.state(a) is R(dir) & id $\notin$ dir
$\rightarrow$       in.deq;
        m.setState(a, R(dir+{id}));
        out.enq(Msg(Home,id,ShRep, a,m.data(a)))
```

Outstanding Exclusive Copy

```
        Msg(id,Home,ShReq,a) == m msg
    &   m.state(a) is W(id') & (id' is not id)
$\rightarrow$       m.setState(a, $T_W$(id'));
        out.enq(Msg(Home,id',WbReq, a))
```

Processing ExReq Messages (at home)

Uncached or cached only at the requester cache

```
     Msg(id,Home,ExReq,a) ==m msg
  &  m.state(a) is R(dir) &  (dir is empty or has only id)
  →  in.deq
     m.setState(a, W(id))
     out.enq(Msg(Home,id,ExRep, a, m.data(a))
```

Outstanding Shared Copies

```
     Msg(id,Home,ExReq,a) ==m msg
  &  m.state(a) is R(dir) & !(dir is empty or has only id)
  →  m.setState(a, T_R(dir-{id}))
     out.enq(multicast(Home,dir-{id},InvReq, a)
```

Outstanding Exclusive Copy

```
     Msg(id,Home,ExReq,a) ==m msg
  &  m.state(a) is W(id')  & (id' is not id)
  →  m.setState(a, T_W(id'))
     out.enq(Msg(Home,id',FlushReq, a)
```

Processing Reply Messages (at cache)

ShRep
```
        Msg(Home, id, ShRep, a, v) == msg
    --  cache.state(a) must be Pending or Nothing
→       m2c.deq
        cache.setState(a, Sh)
        cache.setData(a, v)
```

ExRep
```
        Msg(Home, id, ExRep, a, v) == msg
    --  cache.state(a) must be Pending or Nothing
→       m2c.deq
        cache.setState(a, Ex)
        cache.setData(a, v)
```

-- In general only a part of v will be overwritten by the Store instruction.

Processing InvReq Message (at cache)

InvReq

```
        Msg(Home,id,InvReq,a) == msg
    &   cache.state(a) is Sh
→       m2c.deq
        cache.invalidate(a)
        c2m.enq (Msg(id, Home, InvRep, a))

        Msg(Home,id,InvReq,a) == msg
    &   cache.state(a) is Nothing or Pending
→       m2c.deq
```

Processing WbReq Message (at cache)

WbReq

```
        Msg(Home,id,WbReq,a) == msg
    &   cache.state(a) is Ex
→       m2c.deq
        cache.setState(a, Sh)
        c2m.enq (Msg(id, Home, WbRep, a, cache.data(v)))

        Msg(Home,id,WbReq,a) == msg
    &   cache.state(a) is Sh or Nothing or Pending
→       m2c.deq
```

Processing FlushReq Message (at cache)

FlushReq

```
        Msg(Home,id,FlushReq,a) == msg
    &   cache.state(a) is Ex
→       m2c.deq
        cache.invalidate(a)
        c2m.enq (Msg(id, Home, FlushRep, a, cache.data(v)))

        Msg(Home,id,FlushReq,a) == msg
    &   cache.state(a) is Sh
→       m2c.deq
        cache.invalidate(a)
        c2m.enq (Msg(id, Home, InvRep, a))

        Msg(Home,id,FlushReq,a) == msg
        &   cache.state(a) is Nothing or Pending
. →     m2c.deq
```

Processing Reply InvRep Messages (at home)

InvRep

```
           Msg(id,Home,InvRep,a) == mmsg
       &   m.state(a) is $T_R$(dir)
  →        deq mmsg;
           m.setState(a, $T_R$(dir-{id}))

           Msg(id,Home,InvRep,a) == mmsg
       &   m.state(a) is R(dir)
. →        deq mmsg;
           m.setState(a, R(dir-{id}))
```

Processing Reply WbRep Messages (at home)

WbRep

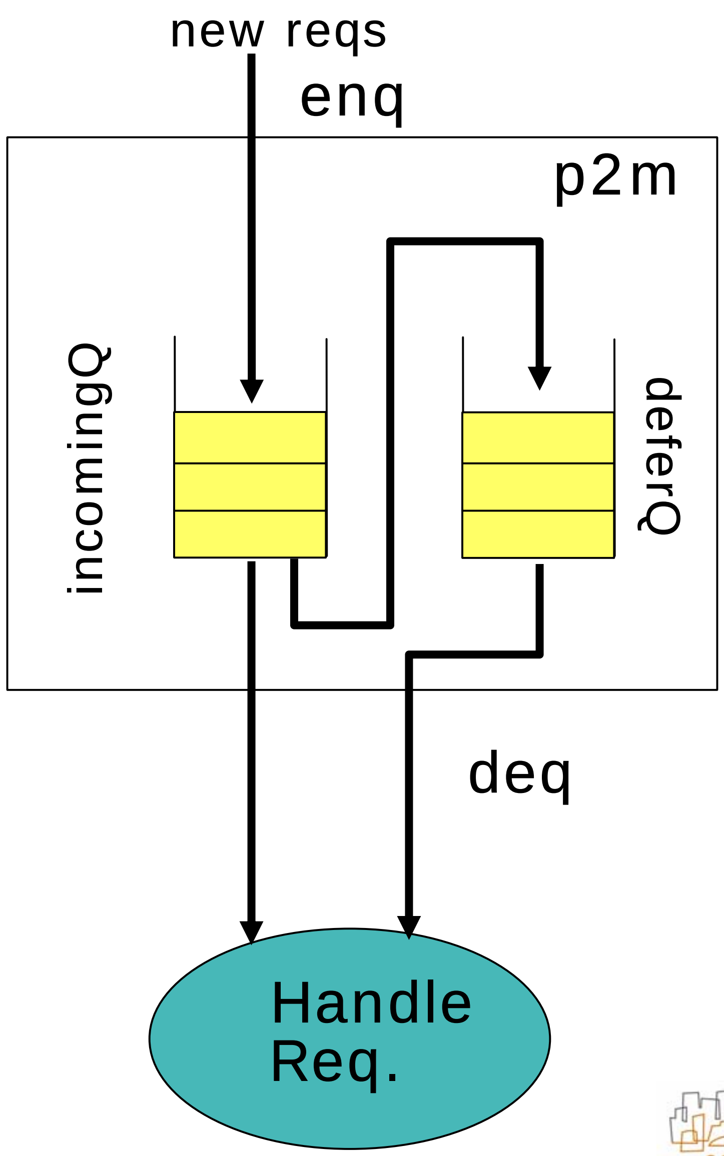
```
        Msg(id,Home,WbRep,a,v) == mmsg
    --  m.state(a) must be T_W(id) or W(id)
→       deq mmsg;
        m.setState(a, R(id))
        m.setData(a,v)
```

FlushRep

```
        Msg(id,Home,FlushRep,a,v) == mmsg
    --  m.state(a) must be T_W(id) or W(id)
→       deq mmsg;
        m.setState(a, R(Empty))
        m.setData(a,v)
```

Non-Blocking Caches

- Non-blocking caches are needed to tolerate large memory latencies
- To get non-blocking property we implement p2m with 2 FIFOs (deferQ, incomingQ)
- Requests moved to deferQ when:
  - address not there
  - needed for consistency

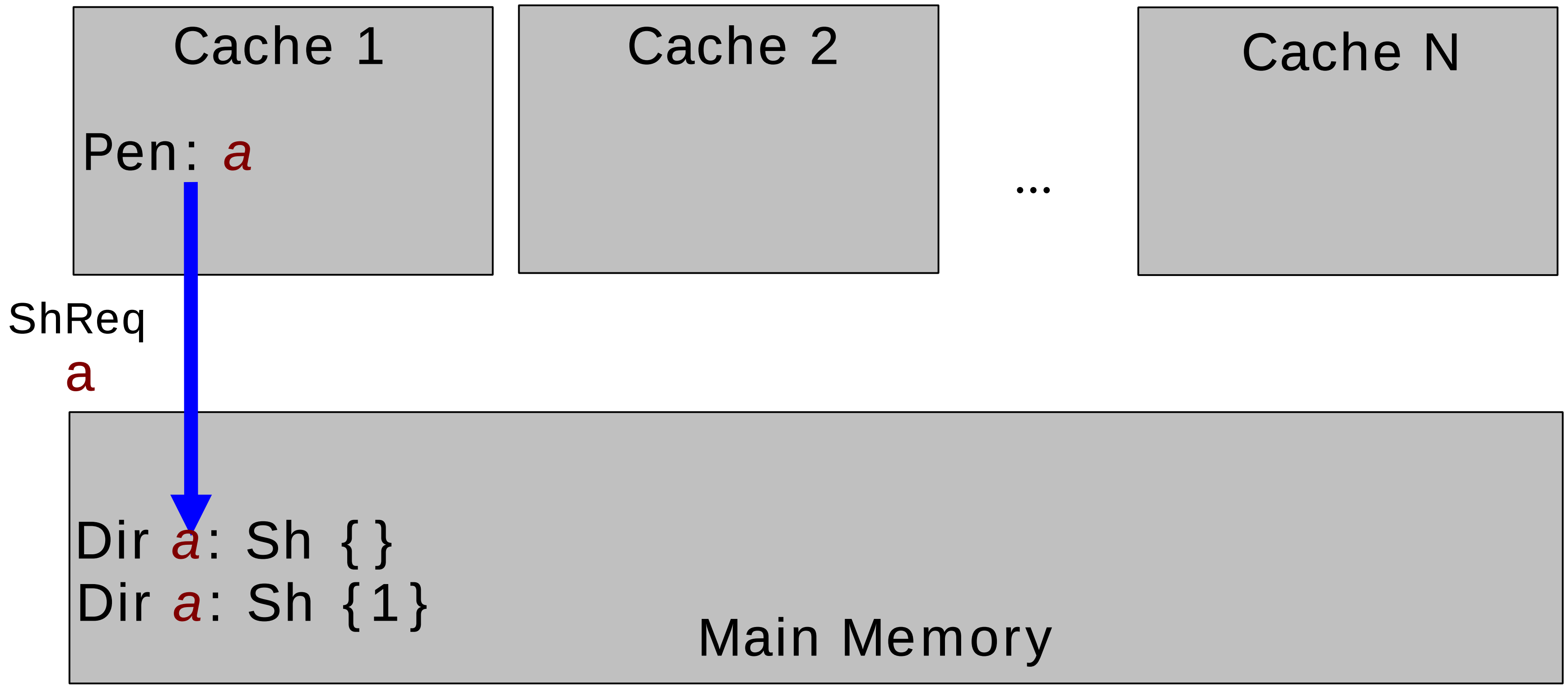


Conclusion

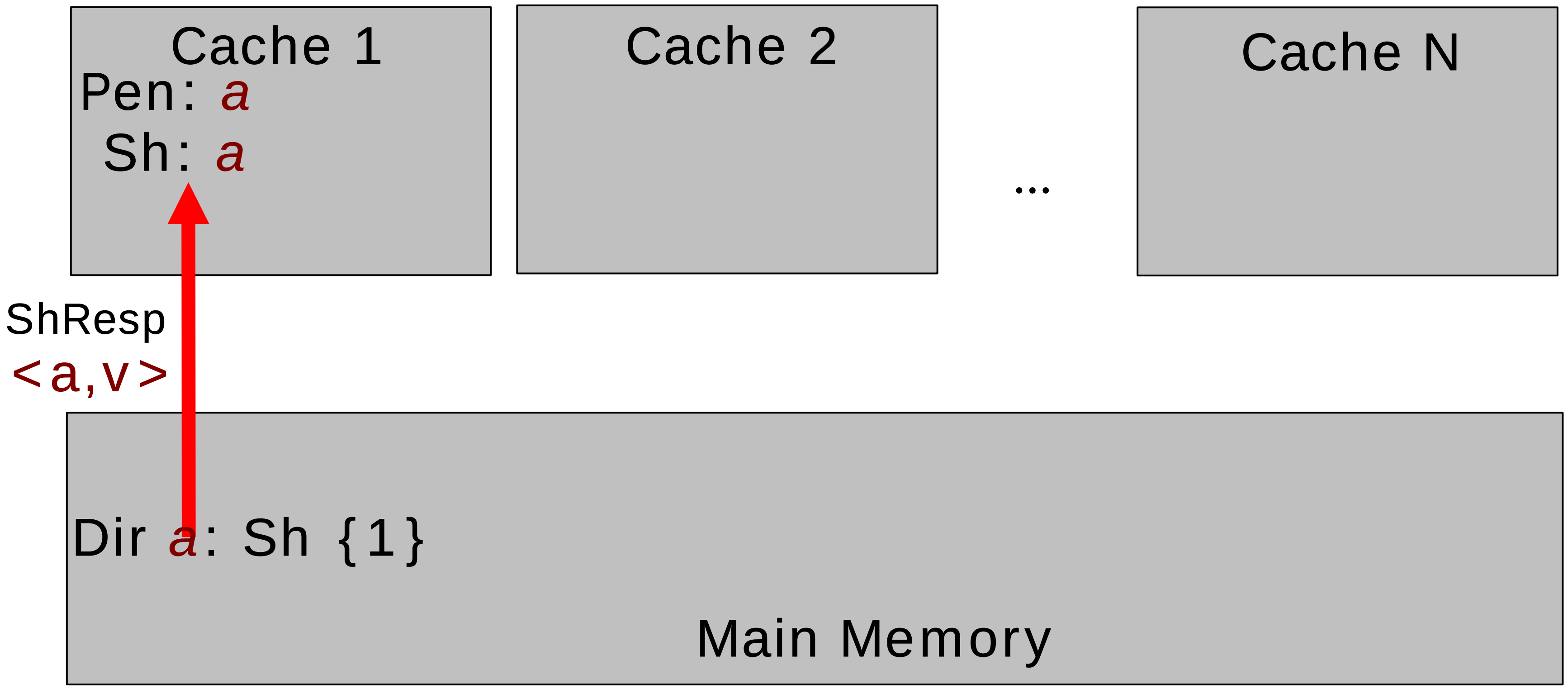
- This protocol with its voluntary rules captures many other protocols that are used in practice.
  - we will discuss a bus-based version of this protocol in the next lecture
- We need policies and mechanisms to invoke voluntary rules to build truly adaptive protocols.
  - search for such policies and mechanisms in an active area of research
- Quantitative evaluation of protocols or protocol features is extremely difficult.

Thank you !

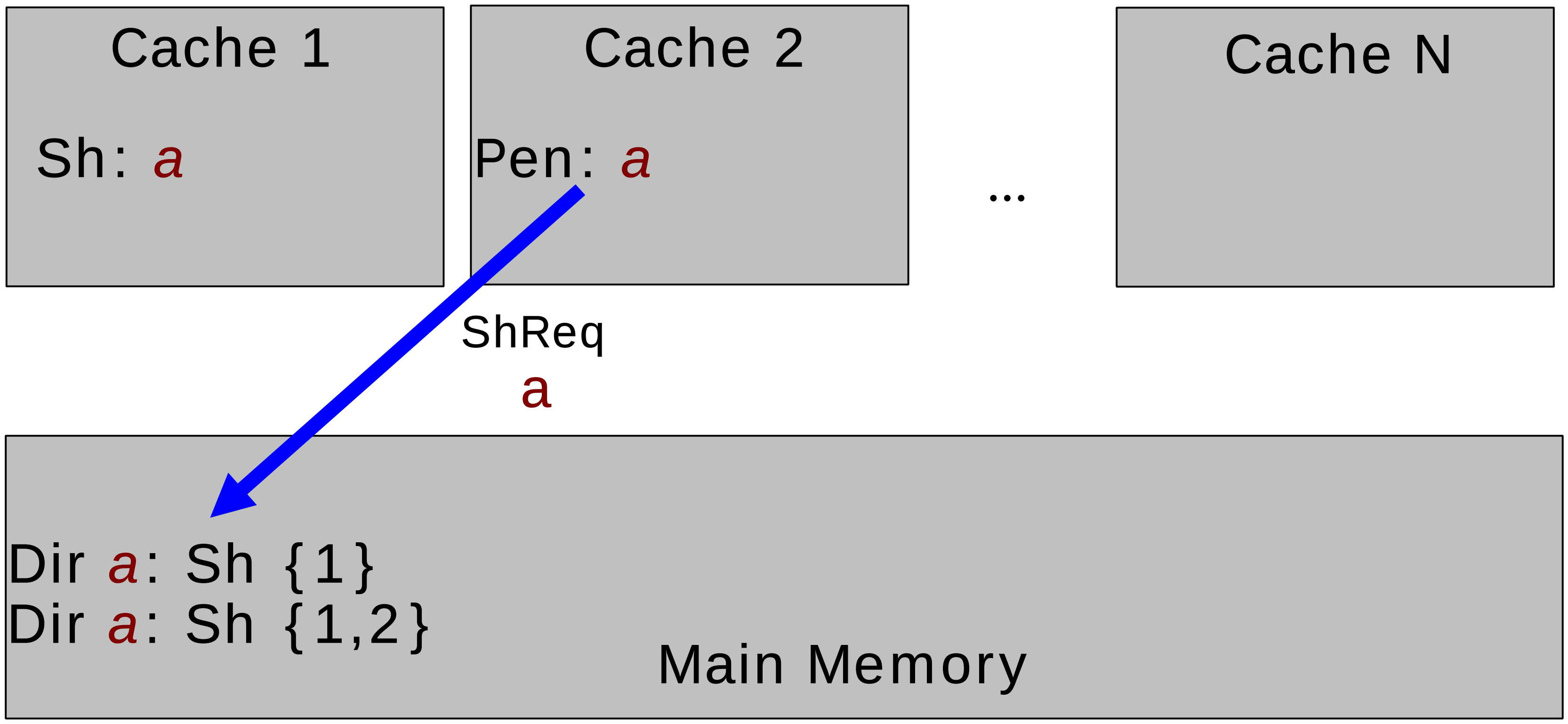
Protocol Diagram



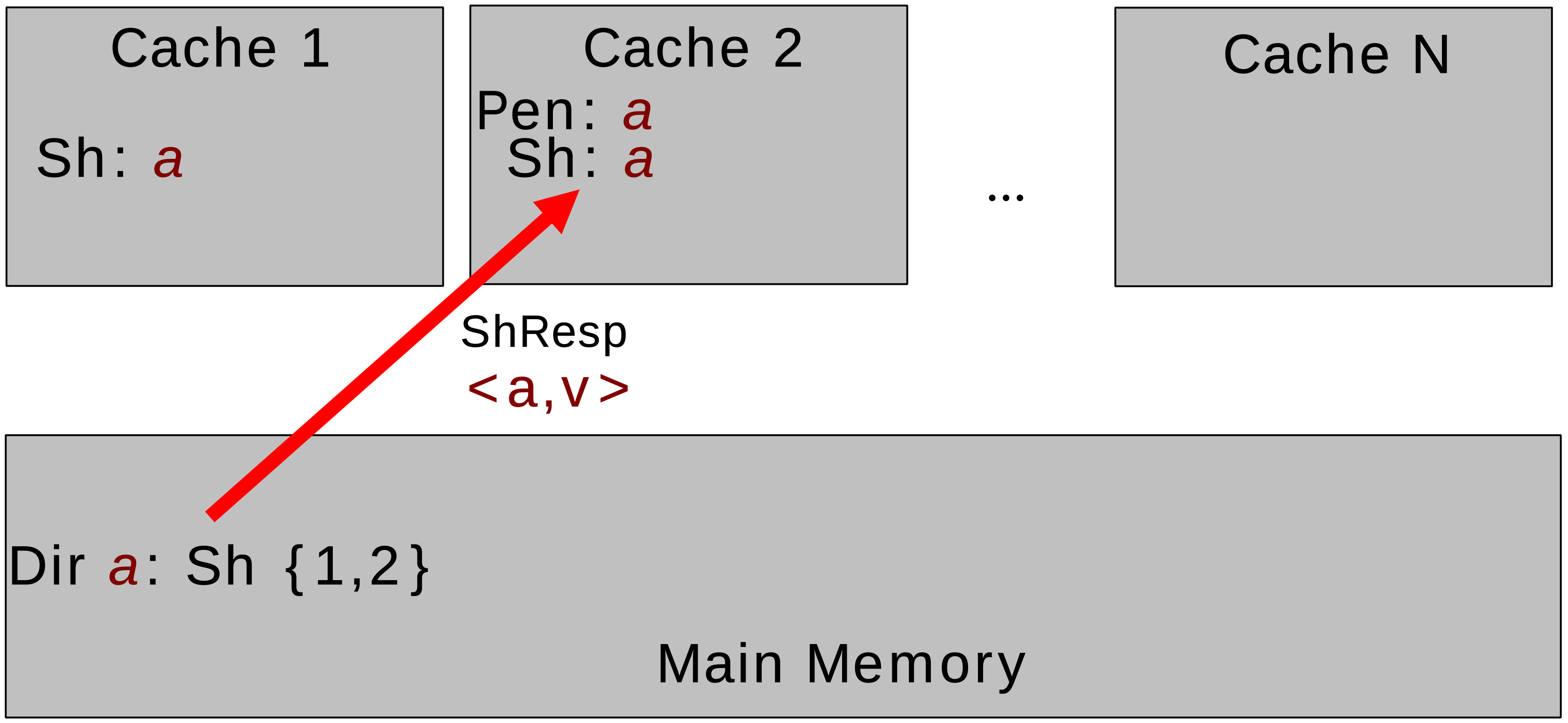
Protocol Diagram



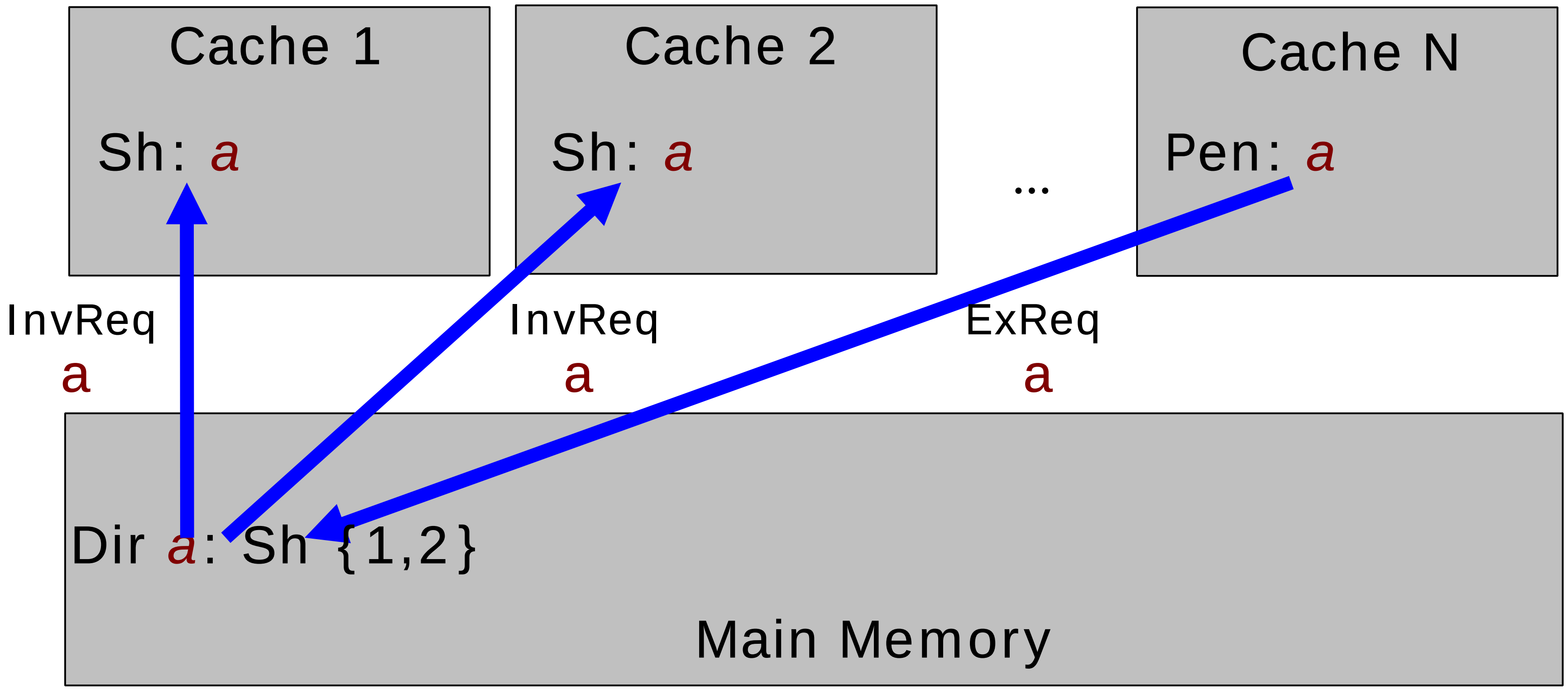
Protocol Diagram



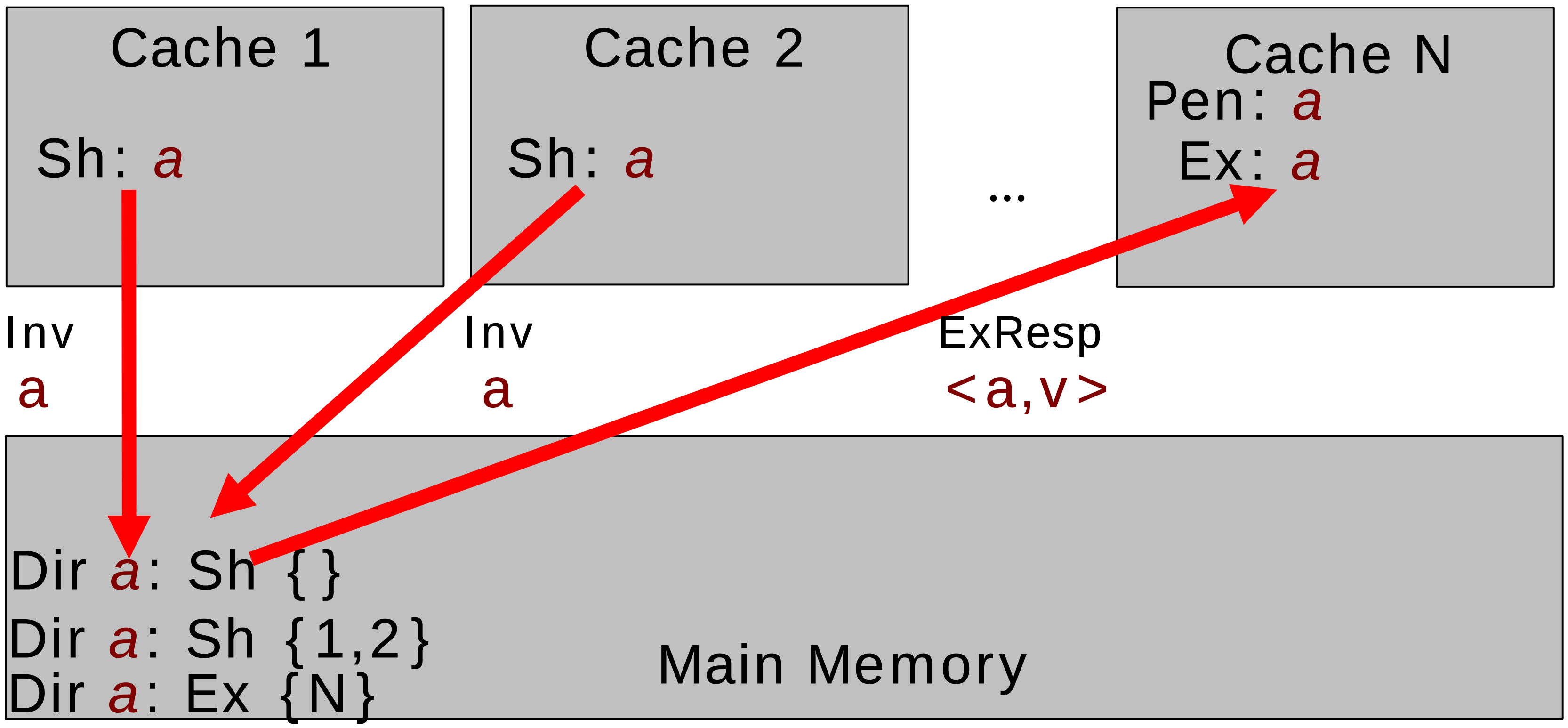
Protocol Diagram



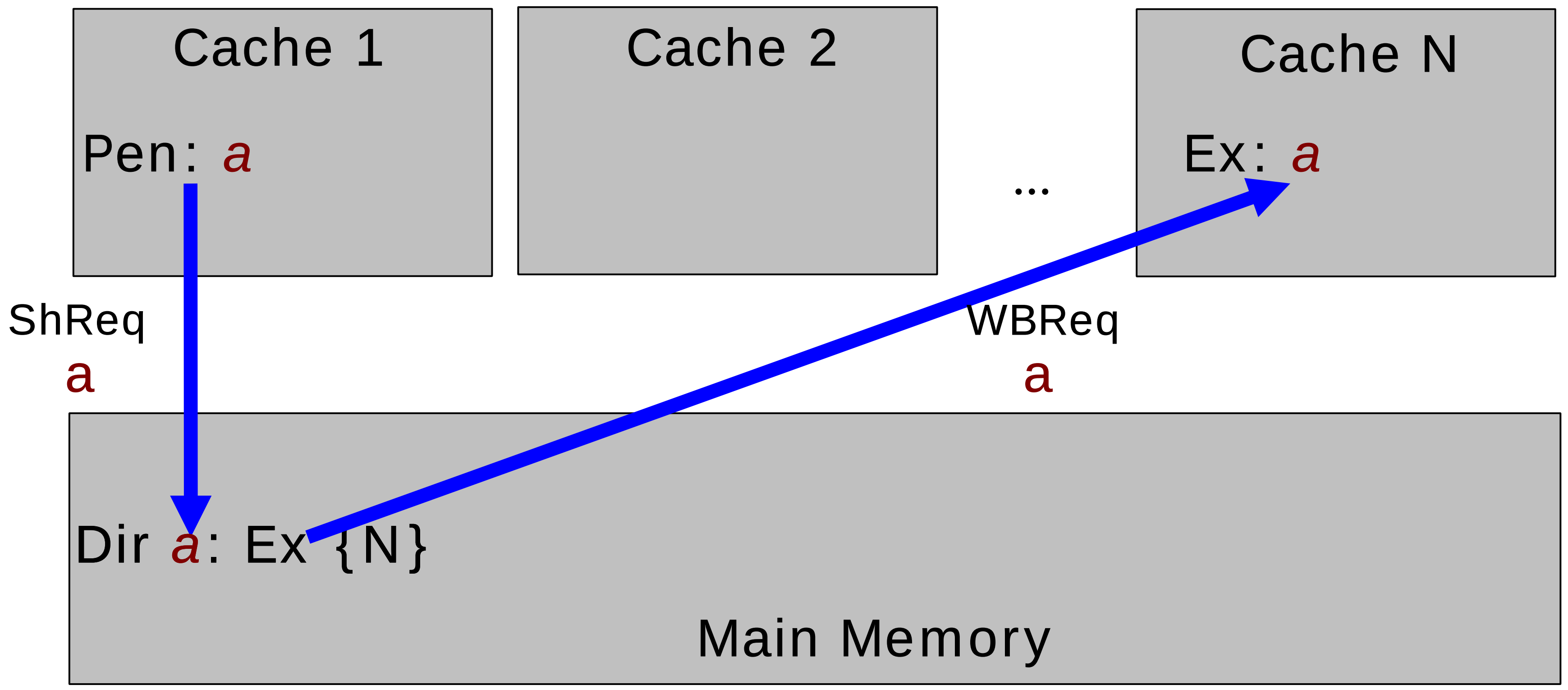
Protocol Diagram



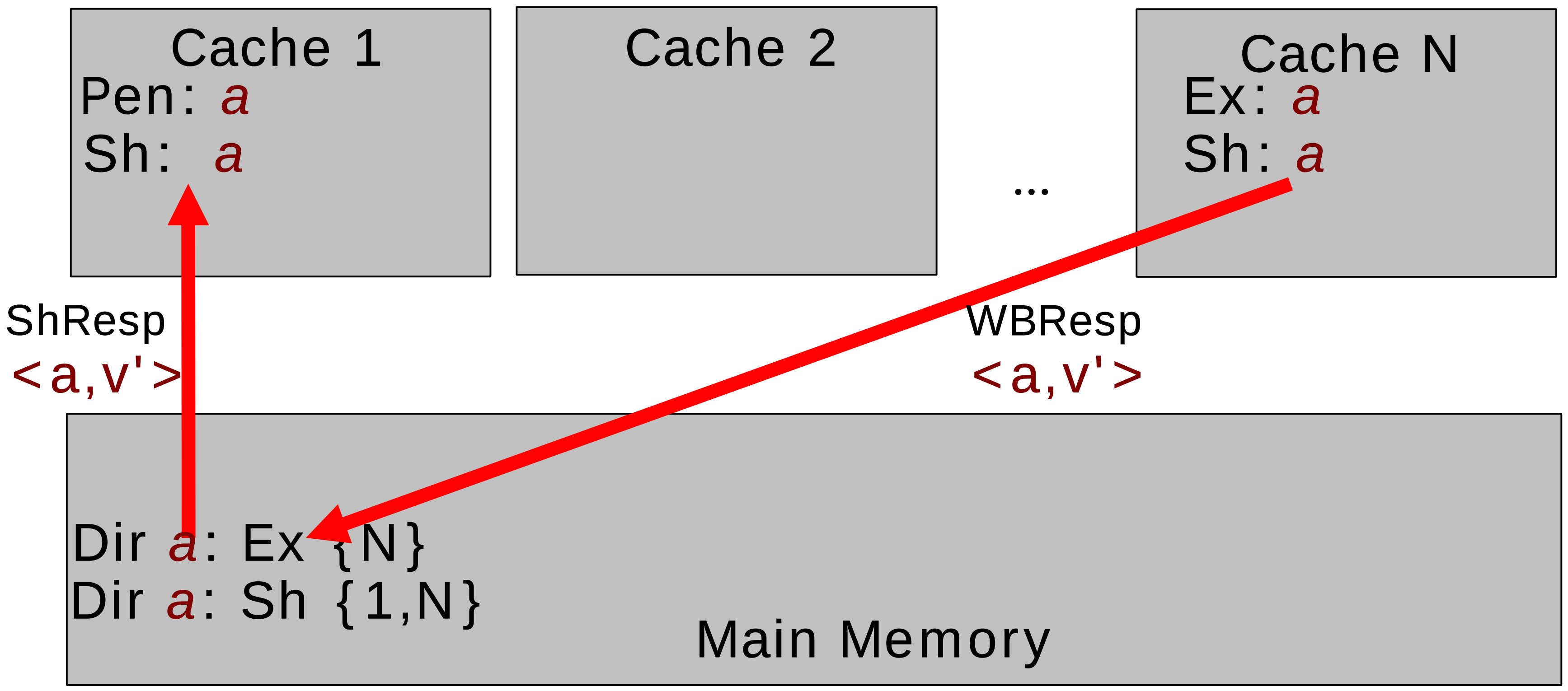
Protocol Diagram



Protocol Diagram



Protocol Diagram


Cache 1
Pen: a
Sh: a
Cache 2
…
Cache N
Ex: a
Sh: a
ShResp
<a,v'>
WBResp
<a,v'>
Dir a: Ex {N}
Dir a: Sh {1,N}
Main Memory