CS M151B:
Computer Systems Architecture

Week 10

Cache Coherence

- Consider a two processor shared memory multi-processor each running the following snippets.

P0 runs:
addi $t0, $zero, 10
sw $t0, A

P1 runs:
lw $t0, A

- What is in P1's $t0 register after execution of this code?

Cache Coherence

- If your answer was... “it depends”, then you're right. Given no other restrictions, we can make no assumptions about what ordering these instructions are executed.
- This is not the problem of cache coherence.
- Consider a specific ordering.

```
P0:
addi $t0, $zero, 10
sw $t0, A
P1:
lw $t0, A
```

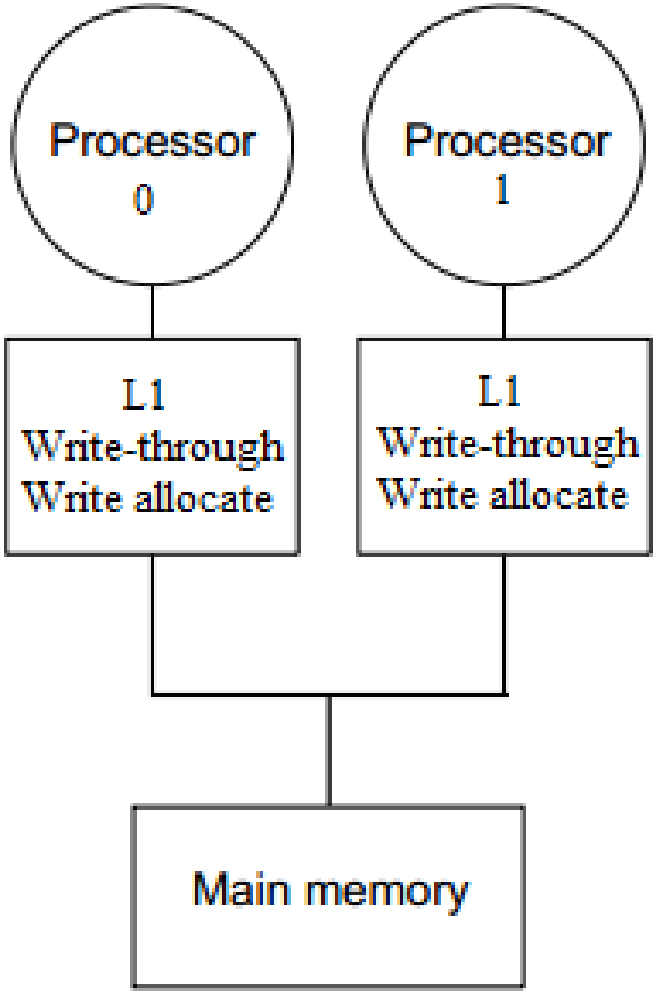
Cache Coherence

P0:
addi $t0, $zero, 10
sw $t0, A
P1:
lw $t0, A

- Now we have a specific ordering. We expect P0 to set it's $t0 register to 10 and then store that at address A. Then P1 reads from address A. P1's $t0 register should have 10.
- How does this look from the processors?

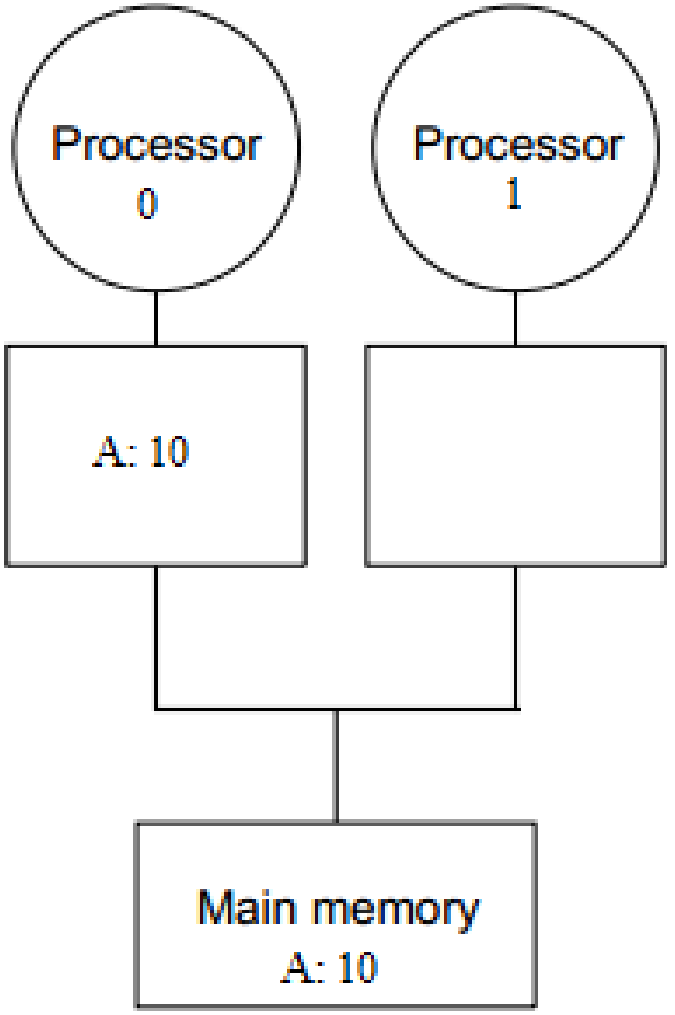
Cache Coherence

- Assume this is our setup and that the L1 caches are empty.
- P0.$t0 is 10.
- P0:
  - sw $t0, A



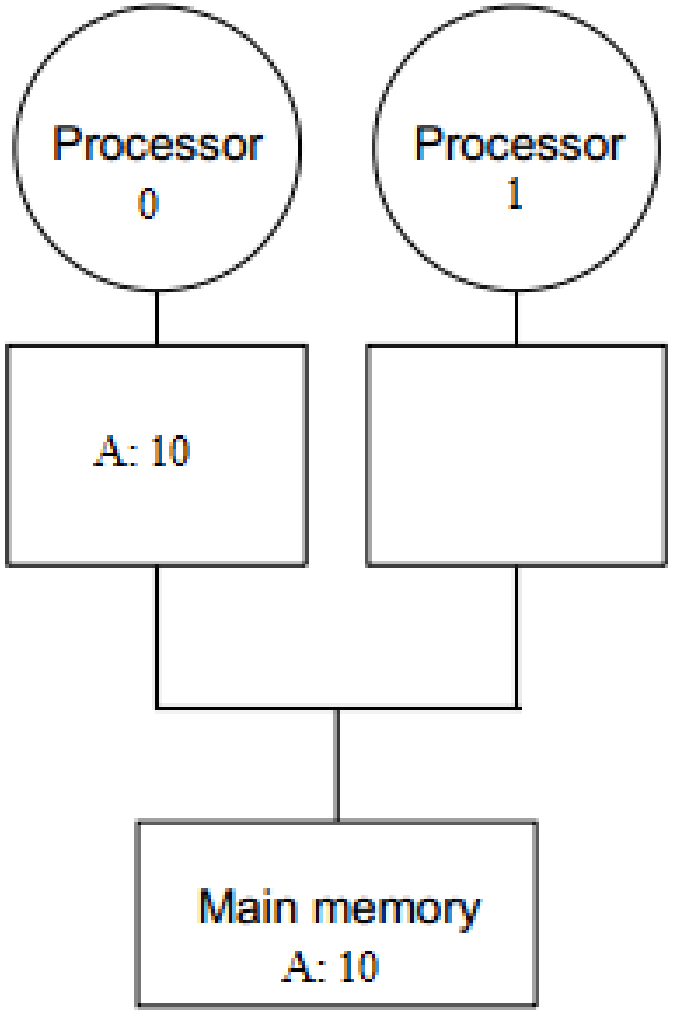
Cache Coherence

- P0:
  - sw $t0, A
- Because write allocate and write through, pull A from memory and write to both cache and main memory.



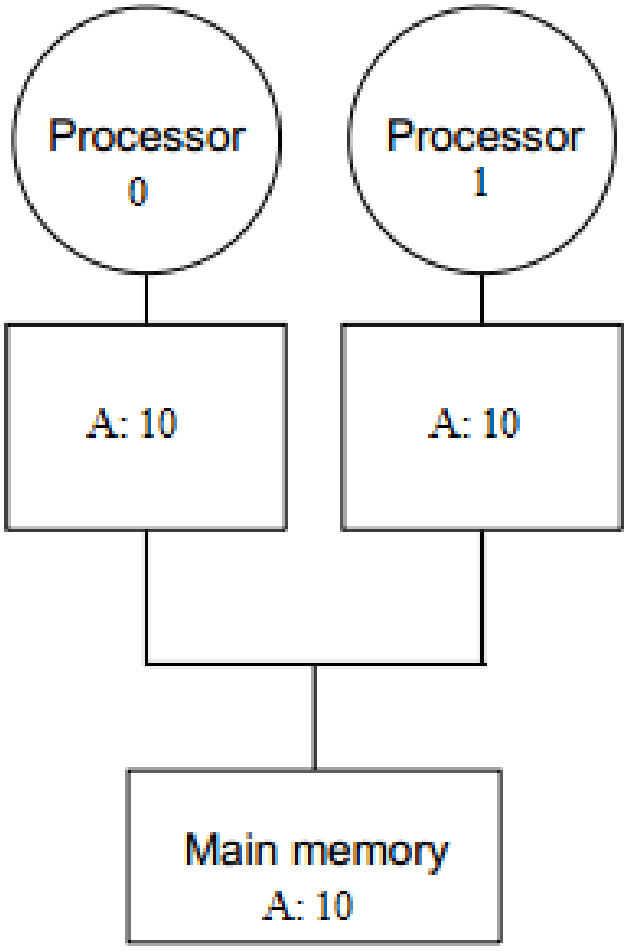
Cache Coherence

- P1:
  - lw $t0, A



Cache Coherence

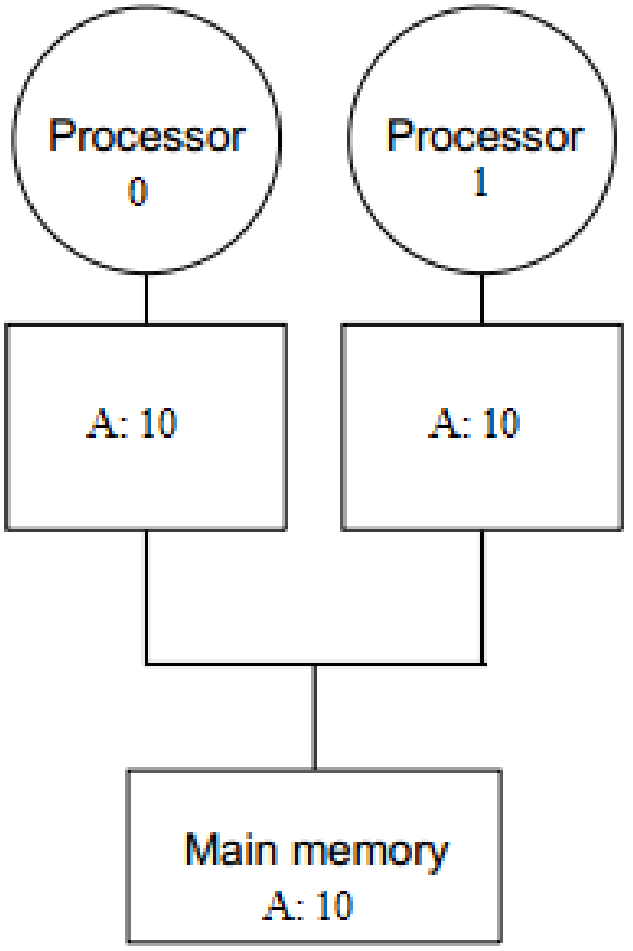
- P1:
  - lw $t0, A
- Now P1.$t0 has 10 and A is cached in P1's L1 cache.
- No problem!



Cache Coherence

- Let's repeat this with:

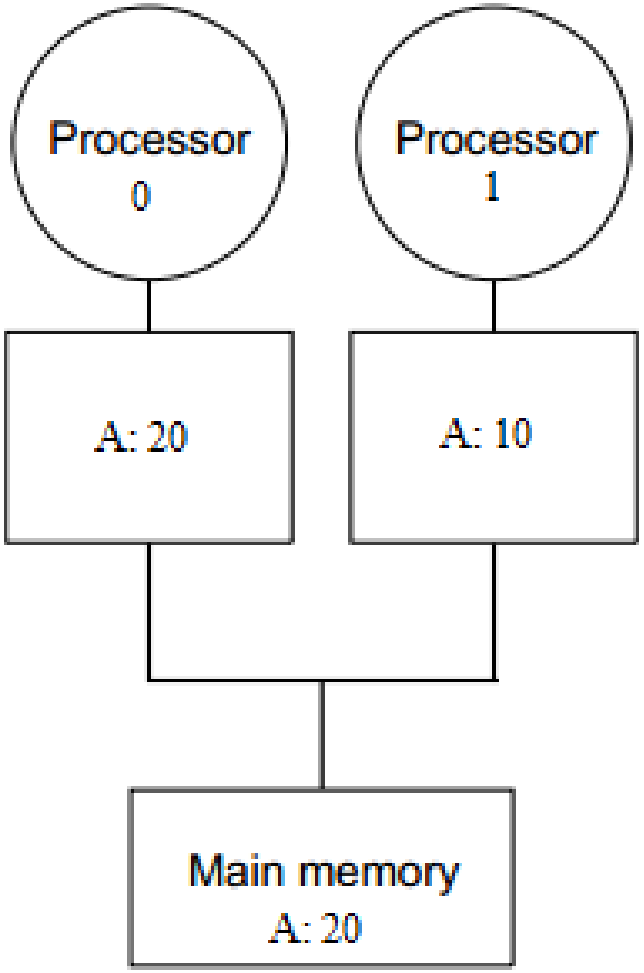
  P0:
  addi $t0, $zero, 20
  sw $t0, A
  P1:
  lw $t0, A



- Assume now $t0 has 20.
- First, P0:
  - sw $t0, A

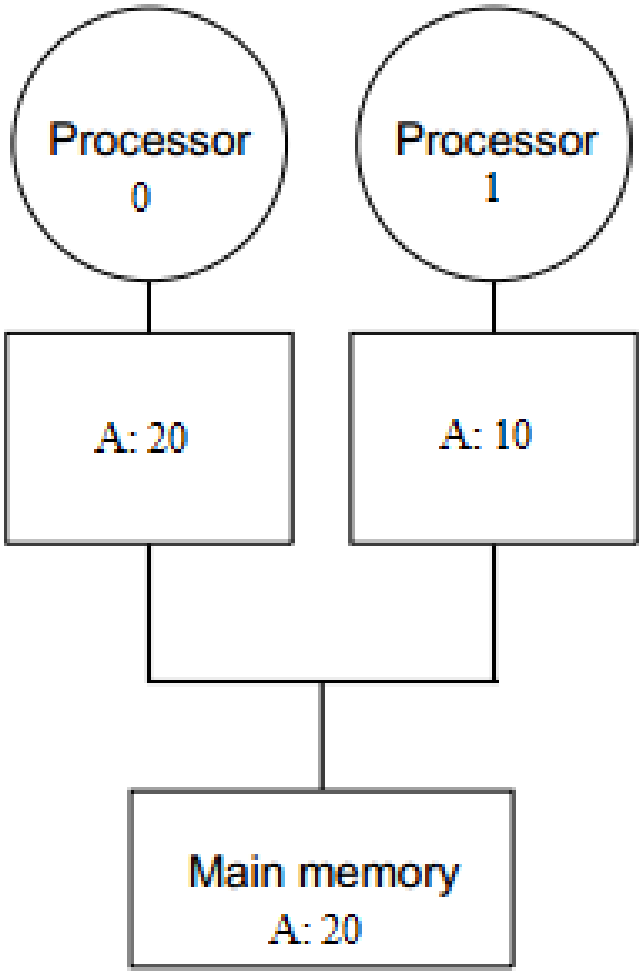
Cache Coherence

- P0:
  - sw $t0, A
- Because write-through, A is set to 20 in both memory and the P0 L1 cache.
- Now, P1:
  - lw $t0, A



Cache Coherence

- P1:
  - lw $t0, A
- Uh oh... no change. We completed a write to A, but P1 didn't see it because of that pesky cache.
- THIS is the problem of cache coherence.



Cache Coherence

- To be more precise, a system like this lacks “coherence”.
- Coherence means maintaining:
  - Program order: If P1 writes to X and no other processor writes to X, then if P1 reads from X, it will return the value written by P1.
  - Write propagation: If P1 writes to X and no other writes occur to X, then if P2 reads from X (after enough time), P2 will read the value written by P1.
  - Write serialization: Writes to the same location will be seen in the same order by all processors.

Cache Coherence

- In a nutshell, coherence means every processor sees the most recent write to a location.
- Every processor sees the same orders of writes.

Cache Coherence

- Currently, each block in the cache only has two states, valid or invalid.
- Most solutions to cache coherence require maintaining more state for each block in cache, depending on the method.
  - Modified (only one cached copy of block that is different than in memory, ex. when a cache writes to memory)
  - Shared (multiple copies of blocks, up-to-date in memory, ex. when multiple processors are reading but not writing to a shared block)
  - Invalid (invalid)
  - Exclusive (only one cached copy of block, *up-to-date*)
  - Owned (multiple copies of blocks, not up-to date)

Cache Coherence

- Two popular implementation methods:
  - Snooping caches.
  - Directory based.
- Two popular policies:
  - Write-invalidate
  - Write-update

Snooping Caches

- Each request made by a processor to the shared memory is heard by all of the other processors.
- For example, if processor 0 issues a store to memory address A, processor 1 hears this and reacts accordingly.
- Each local cache now has a snoop controller to listen to the bus

Snooping Caches

- Write-invalidate snooping
- Only one processor can have a modified block in cache at any point in time (modified state).
- Multiple processors can have an unmodified block in cache (shared state).

Snooping Caches

- Write-invalidate snooping
- Some example state transitions:
- If P0 and P1 have unmodified (shared) block A in cache and P0 writes to A:
  - P0 cache hit. Write-hit protocol takes over. The block in P0 becomes “modified”.
  - P0 issues an invalidate.
  - Any processors with a copy of A will invalidate that copy.

Snooping Caches

- Write-invalidate snooping
- If P1 has an unmodified (shared) block A in cache and P0 writes to A:
  - P0 cache miss. Write-miss protocol takes over. If using a write-allocate, the block is cached in P0 as “modified”.
  - P0 issues a write-miss.
  - Any processors with a copy of A will invalidate that copy.

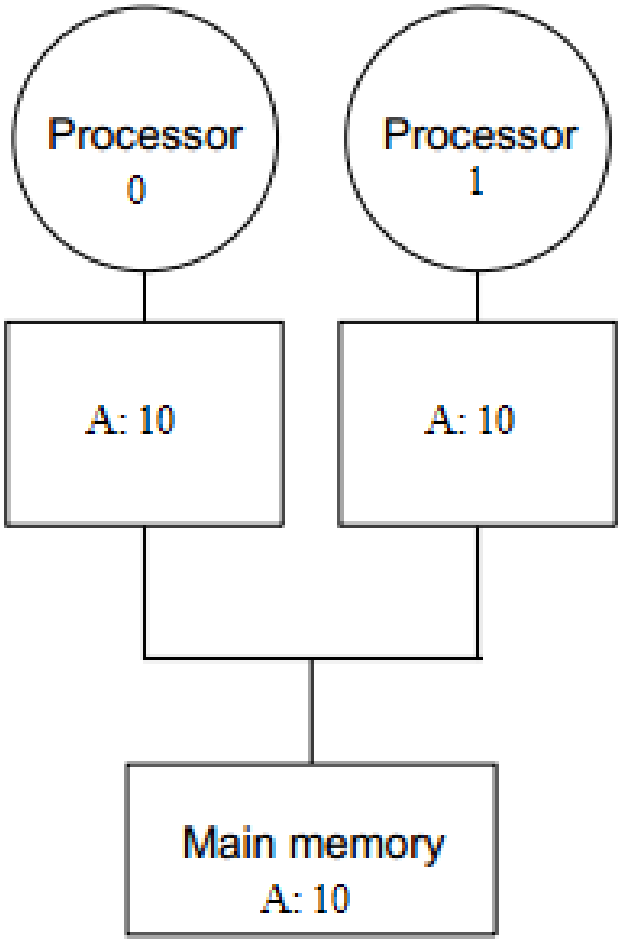
Snooping Caches

- Write-invalidate snooping
- If P1 has modified block A in cache and P0 writes to A:
  - P0 cache miss.
  - P0 issues a write miss.
  - If write-back P1 will have to flush it's block to memory for P0 to read it, and also invalidate.
- ...and so on.

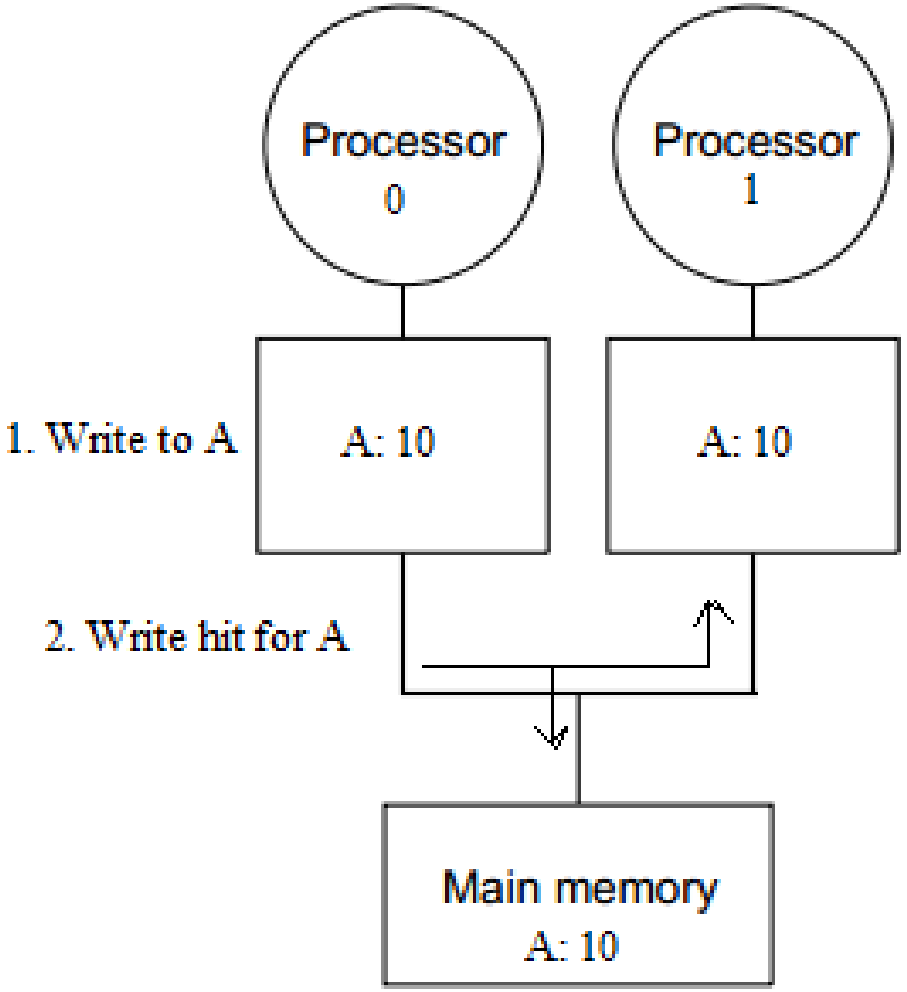
Snooping Caches

- Consider:
  - Write-through, write-allocate
  - P0 has cache block A with value 10
  - P1 has cache block A with value 10
  - Memory has value 10 in memory location A
- P0 writes 20 to A.

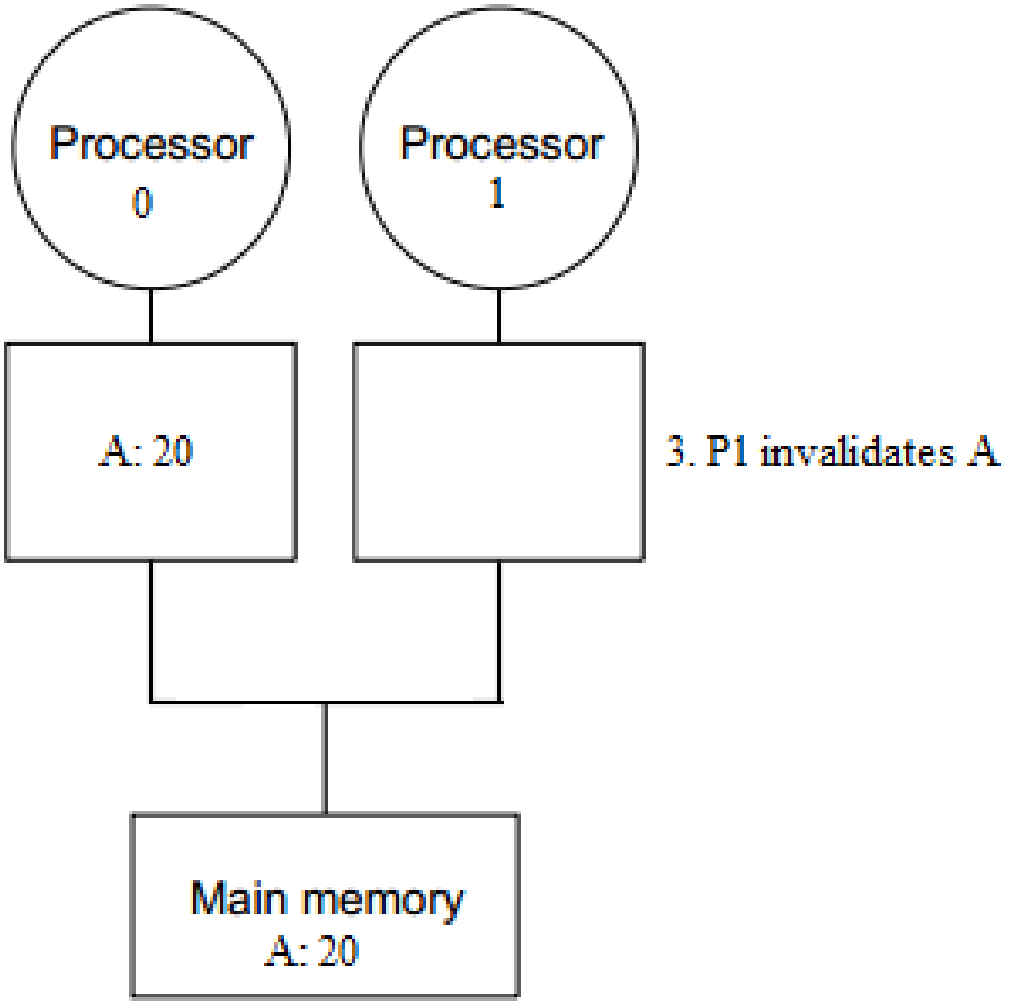
Snooping Caches



Snooping Caches



Snooping Caches



Snooping Caches

- Now, if P1 reads A, it will be a cache miss, but at least it will force A to fetch the correct value from memory.
- What if the cache were write-back?
  - P0 would write 20 to A in the cache block, but memory would have A as being 10.
  - If P1 reads from A, unless it gets the value from P0, it's not going to get a fresh value.

Snooping Caches

- What if the cache were write-back?
  - Can we have a system where when we read, we make sure that we get the up-to-date version of the block, be it from memory or another cache?
  - With the power of snooping, sure. A request made by a processor will be heard by all other processors.
  - Does this lead to other complications?
  - You bet!
  - Welcome to the world of cache coherence.

Snooping Caches

- Notice in this scheme a write to shared data will invalidate other cached blocks. This could hamper performance
- An alternative would be to use a write-update policy.
- Most operations are the same, except when you write to shared data, you must also broadcast the new data so that other caches can see and update their own values.

Snooping Caches

- Snooping seems like a reasonable way to implement cache coherence.
- Snooping is falling out of favor however...
- Why (aka, tradeoffs)?

Snooping Caches

- Tradeoffs:
  - The whole system only works under the condition that any processor sees any request issued by any processor. Generally speaking this necessitates the use of a bus to interconnect.
  - Buses are slow and impractical because it is a shared resource where only one processor can use it at any time.
  - This also leads to scaling problems. The more processors attached to the bus, the more of a bottleneck it becomes.

Snooping Caches

- Tradeoffs:
  - It also requires adding a snoop controller and a snoop port to the cache. Adding ports are costly because the number of wires required grow quadratically.
  - Despite being “simple”, snooping is troublesome enough where it's worth looking to other solutions.

Directory Based Caches

- Another solution: Directory based caches
  - Snoopy caches are “simple” because every processor can overhear the actions of every other processor.
  - Because each cache knows exactly what happens, each cache can be responsible for the state of its own blocks
  - However, this “simplicity” is only because of the specific design constraint: that we use a bus or broadcast to all nodes (which can be rough).
  - Bus architectures end up being slow and thus small (8-16 processors)

Directory Based Caches

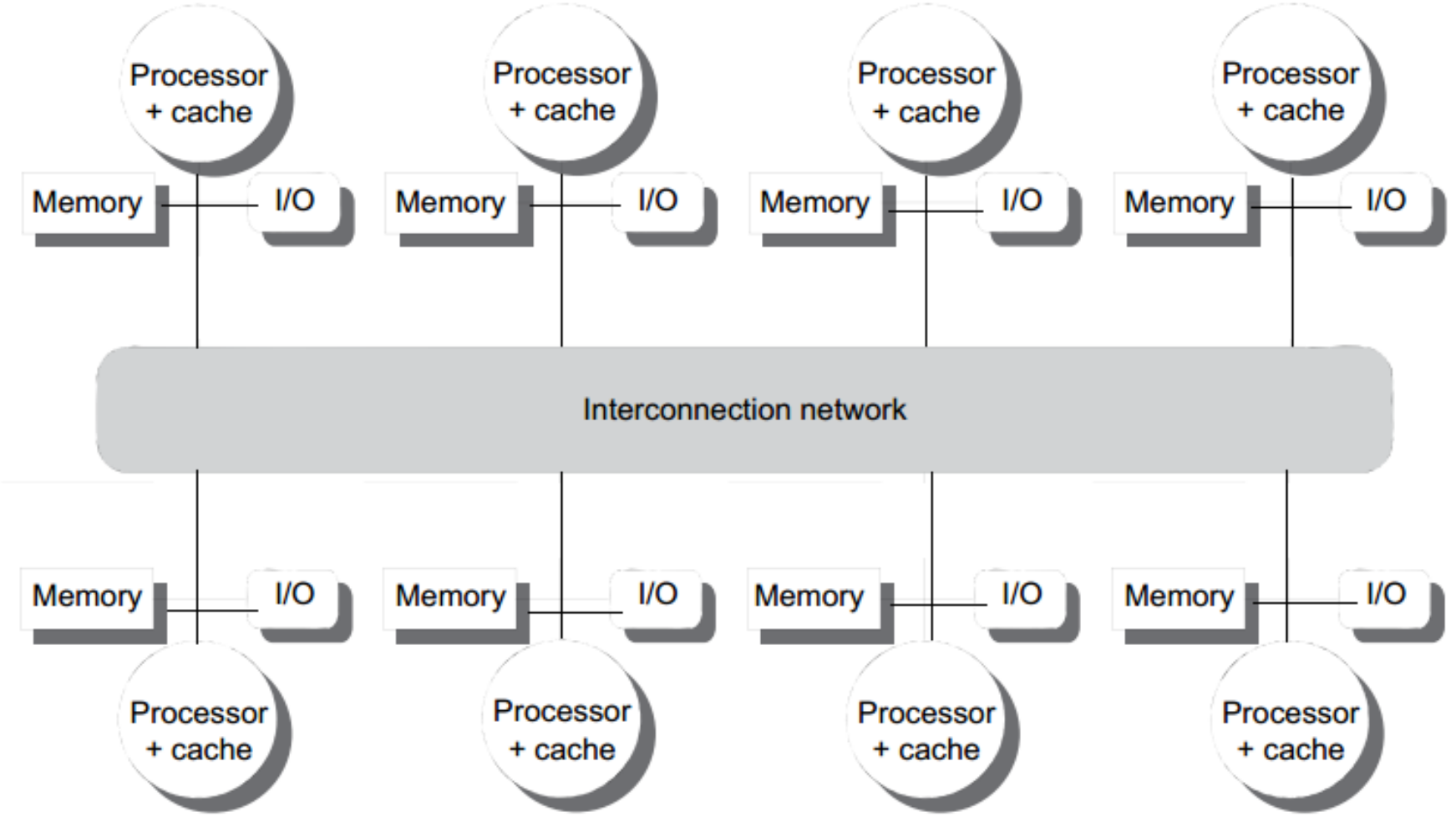
- Say our access to memory is not based on a bus, but an interconnection network, how do we maintain coherence?
- We need a solution that doesn't require every processor to hear every request.
- We need a solution that doesn't require every cache to be solely responsible for maintaining their own coherence.

Directory Based Caches

- Additionally, consider the even more complicated case of distributed memory.
- How can we maintain coherence now?

Directory Based Caches

- Consider a distributed memory system:



Directory Based Caches

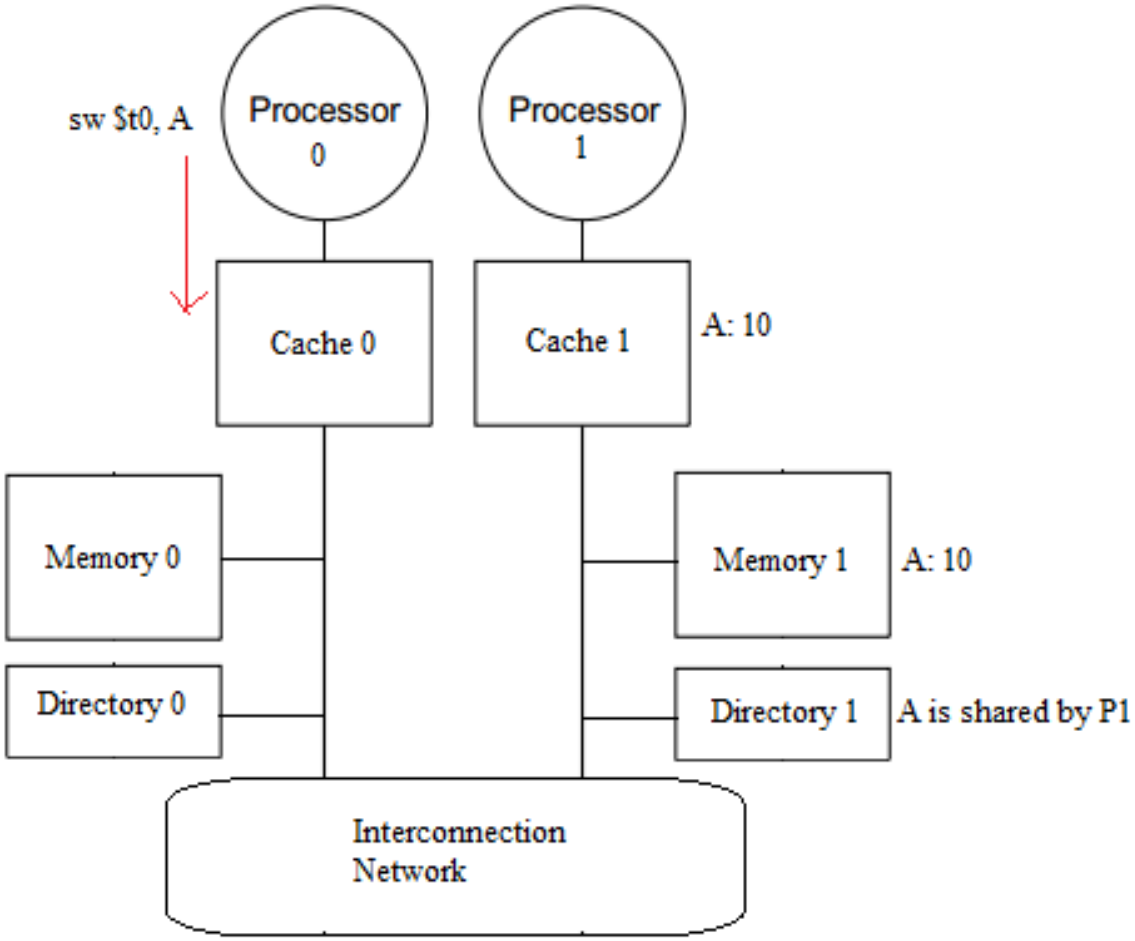
- Each processor node is responsible for a part of memory.
- If a processor makes an access to memory that “belongs” to a different processor, it will have to go through the interconnection network to find the correct node.
- This means, each node will know all of the requests made to that block in memory.

Directory Based Caches

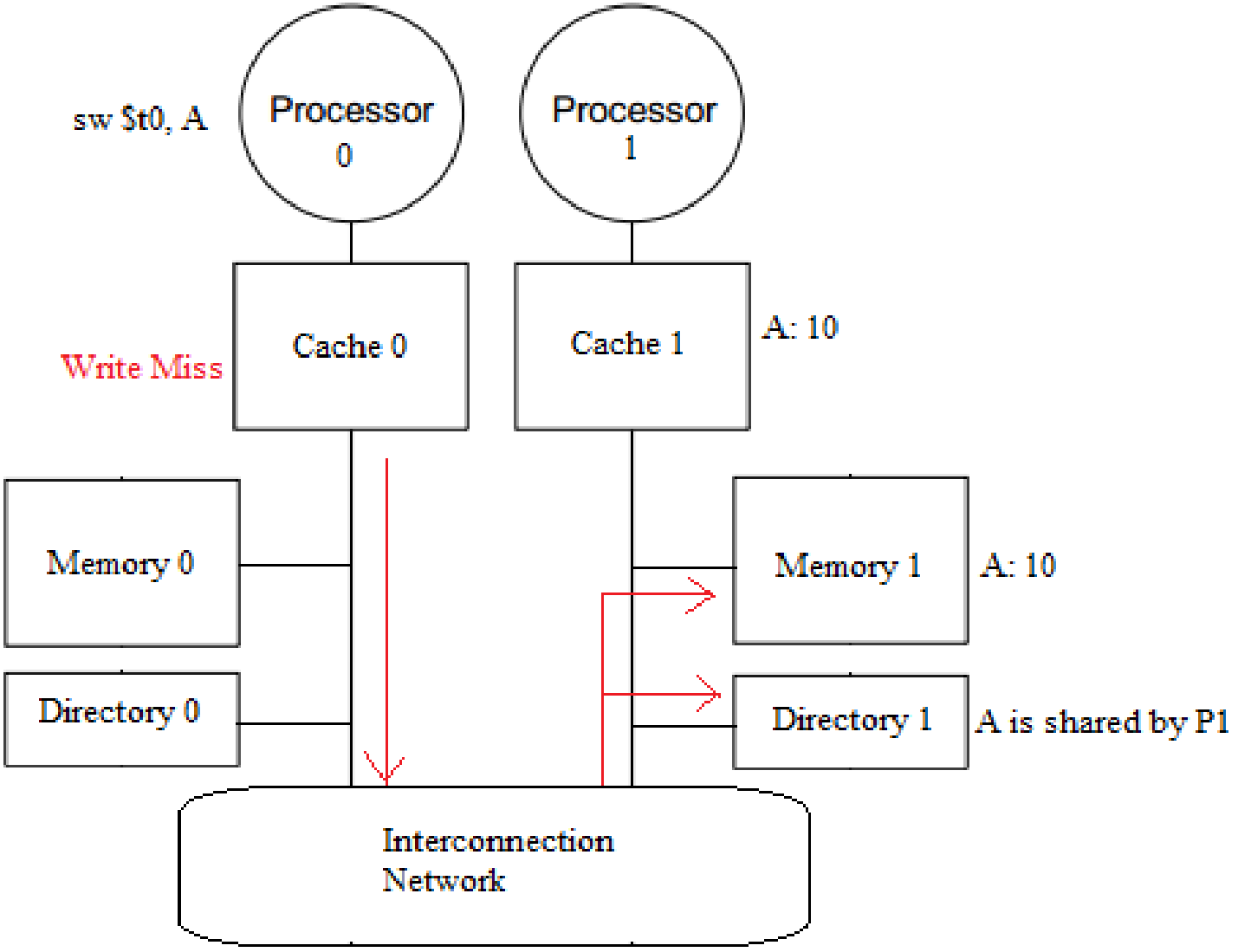
- The idea:
  - Associated with each memory unit is a directory.
  - Associated with each cache is a “directory”.
  - Each memory directory is responsible for keeping that track of the state of each block in that memory unit
    - Who has copies?
    - Could it be it modified?
    - Is it locked?
  - Each cache keeps track of the state of each block in cache.

Directory Based Caches

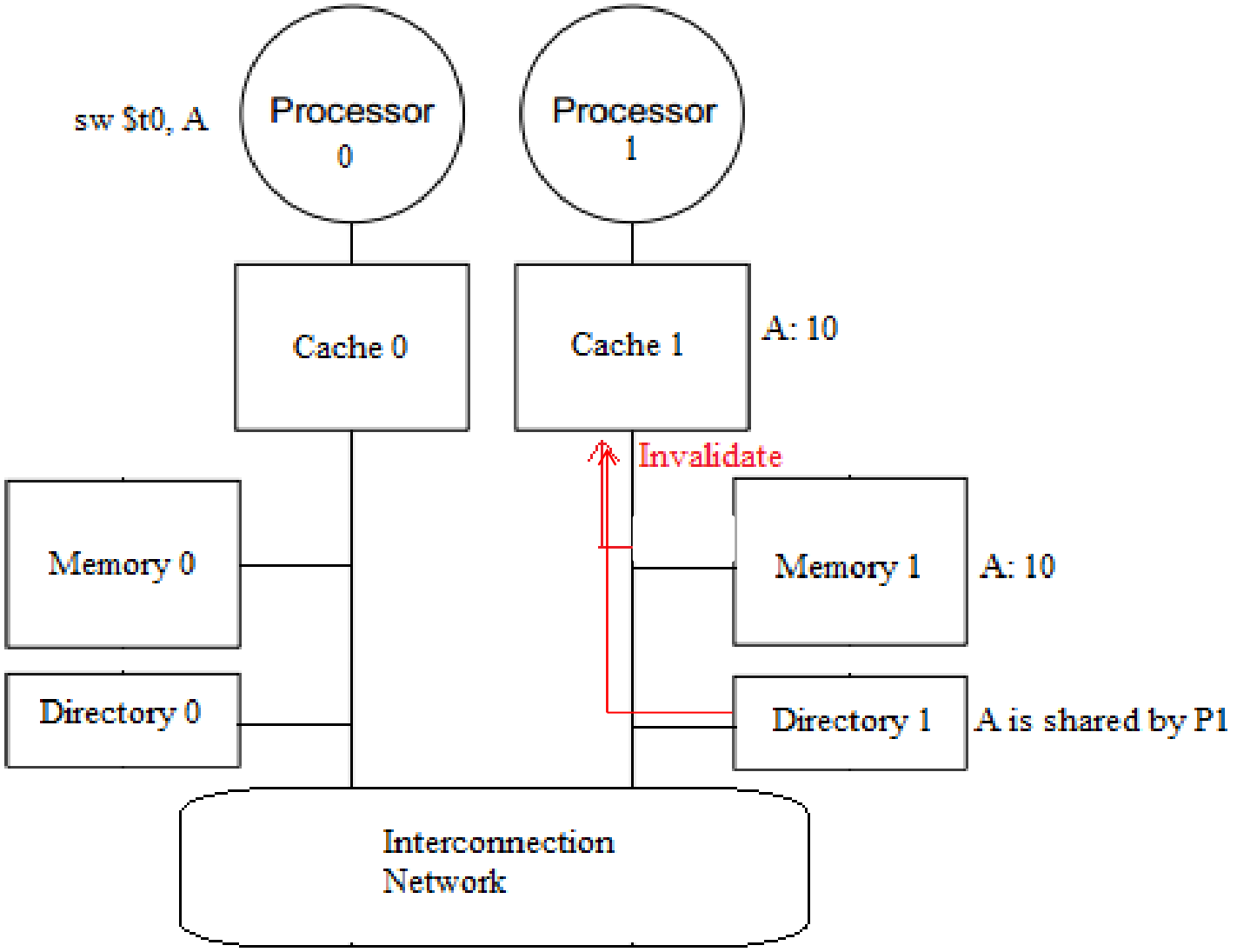
- Consider:
  - Write invalidate
  - A is in memory attached to P1
  - P1 has up-to-date version of A (the directory knows this)
  - P0 writes to A



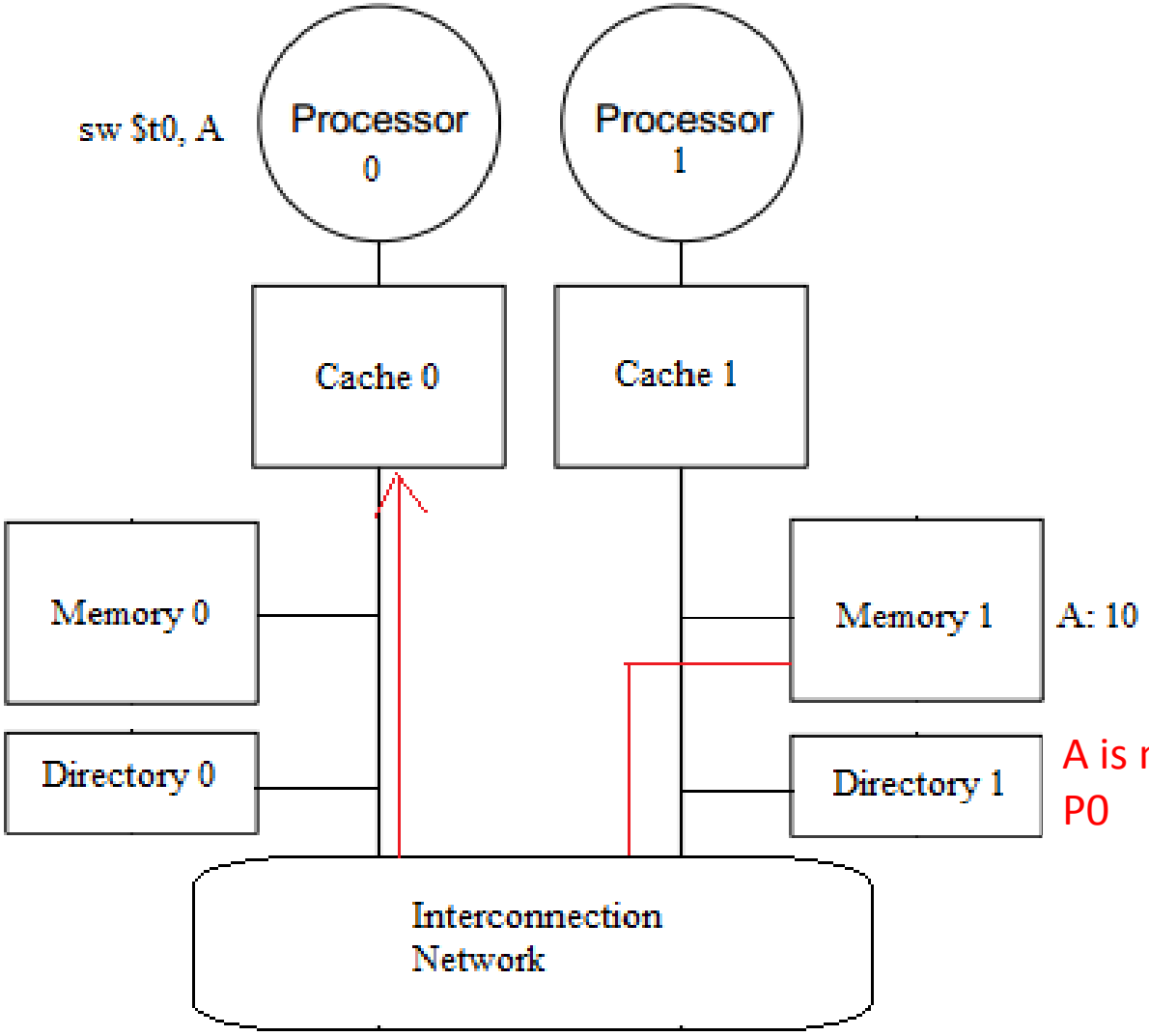
Directory Based Caches



Directory Based Caches



Directory Based Caches



A is modified by P0

Directory Based Caches

- This is merely an example of what a request in a directory based cache might look like.
- Things become complicated very quickly.
  - How many sharers?
  - How to store directory? How large can we allow?
  - Multiple state transition patterns for both cache and directory blocks.
  - Needs explicit acknowledgment when operation is completed.
  - Plus all of the issues with snoopy cache.

End of

The Final Week

-The Exam Remains-

Dawn of
A New Day