Advanced Operating Systems (CS 202)

Synchronization

(some cache coherence slides adapted from Ian Watson)

Classic Example

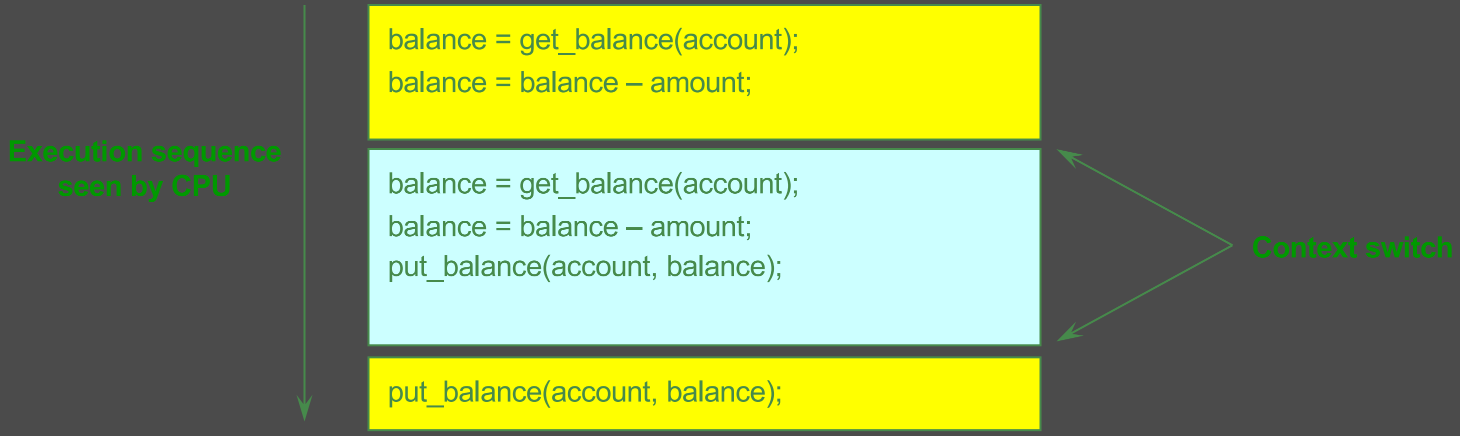
- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
        balance = get_balance(account);
        balance = balance – amount;
        put_balance(account, balance);
        return balance;
}
```

- Now suppose that you and your father share a bank account with a balance of $1000
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:



Execution sequence seen by CPU

```
balance = get_balance(account);
balance = balance – amount;
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

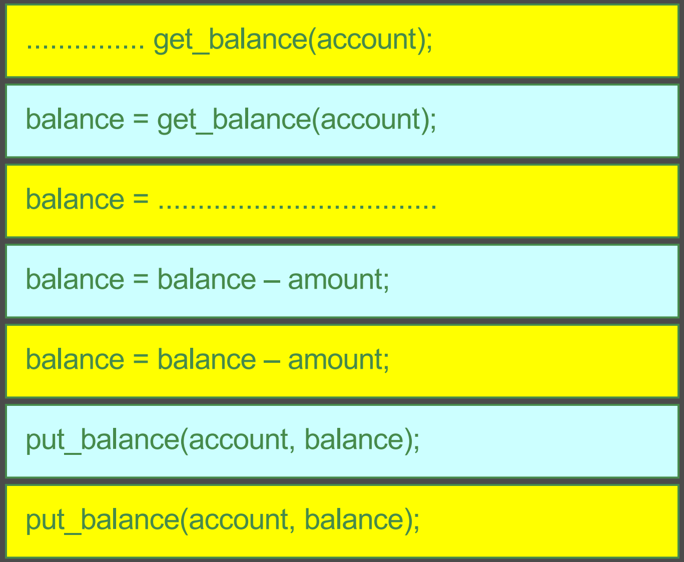
Context switch

- What is the balance of the account now?

How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever



```
............... get_balance(account);
balance = get_balance(account);
balance = ..................................
balance = balance – amount;
balance = balance – amount;
put_balance(account, balance);
put_balance(account, balance);
```

Mutual Exclusion

- Mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
  - What does atomic mean?

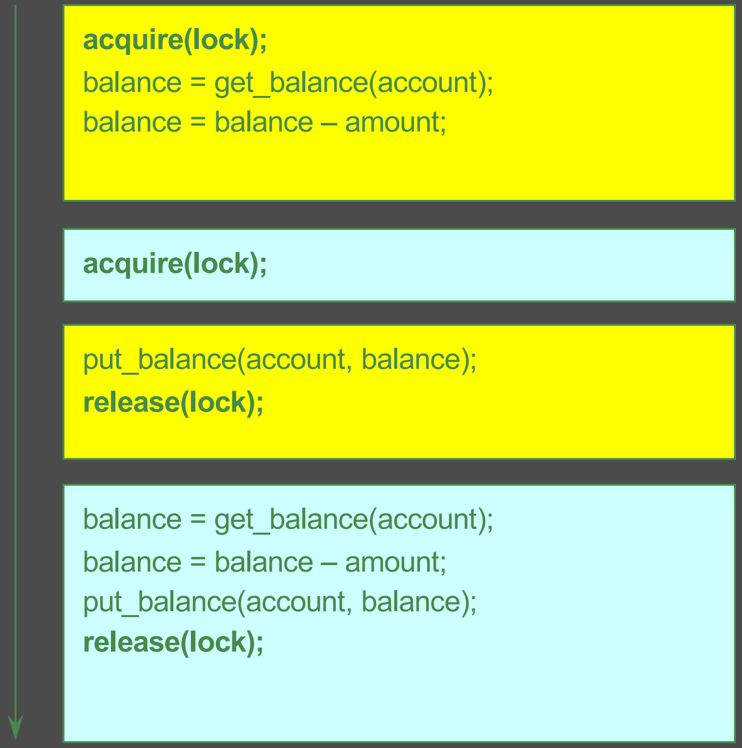
- Code that uses mutual called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others

- What requirements would you place on a critical section?

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section



```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;
```

```
acquire(lock);
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

- Why is the “return” outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

Stepping back

- What does the OS need to support?
  - And why? Isnt this an application/programming problem?
- Synchronization is hard – why?
- Synchronization can be a performance problem – why?
- Other semantics than mutual exclusion possible.

Implementing locks

- Software implementations possible
  - You should have seen Dekker's algorithm and possibly Peterson's algorithm
  - They are difficult to get right
  - They make assumptions on the system that may no longer hold
    - (e.g., memory consistency as we will see shortly)
- Most systems offer hardware support

Using Test-And-Set

- Here is our lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?

Overview

- Before we talk deeply about synchronization
  - Need to get an idea about the memory model in shared memory systems
  - Is synchronization only an issue in multi-processor systems?
- What is a shared memory processor (SMP)?
- Shared memory processors
  - Two primary architectures:
    - Bus-based/local network shared-memory machines (small-scale)
    - Directory-based shared-memory machines (large-scale)

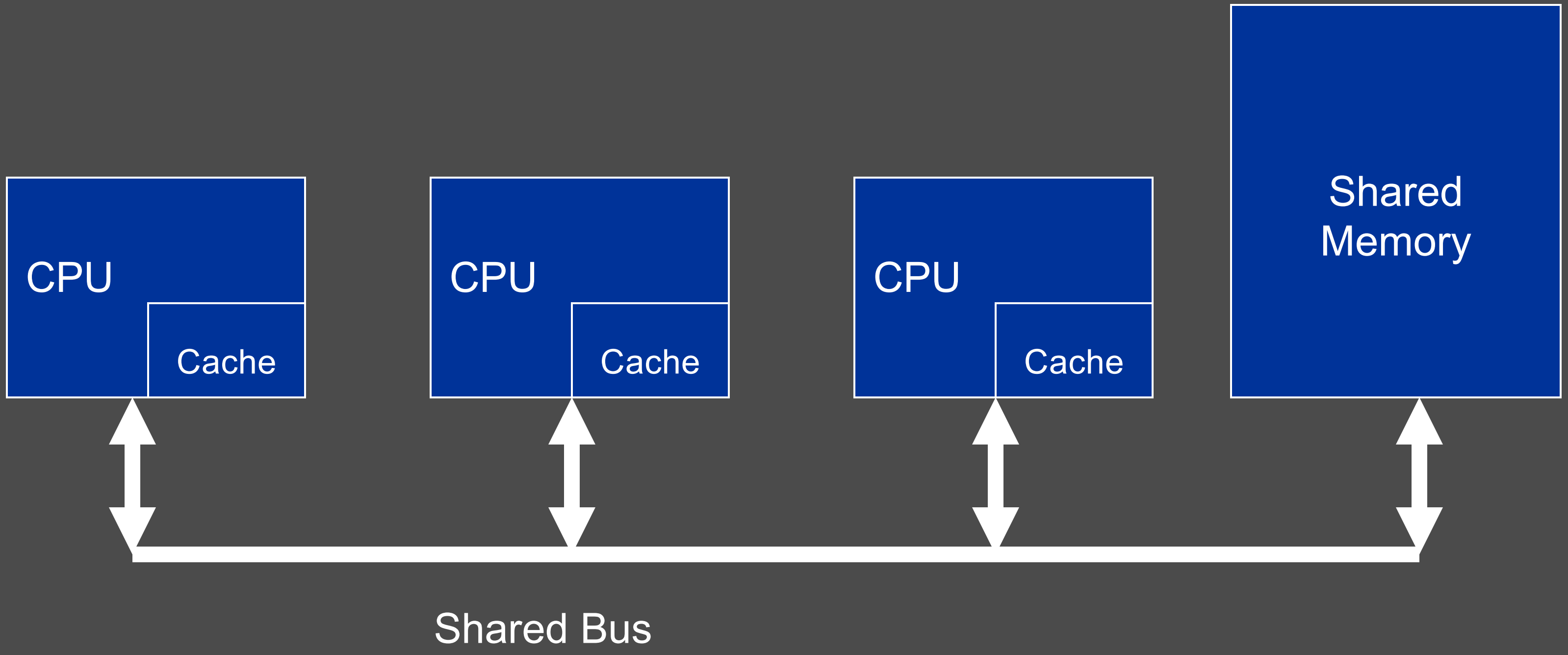
Plan…

- Introduce and discuss cache coherence
- Discuss basic synchronization, up to MCS locks (from the paper we are reading)
- Introduce memory consistency and implications
- Is this an architecture class???
  - The same issues manifest in large scale distributed systems

CRASH COURSE ON CACHE COHERENCE

Bus-based Shared Memory Organization

Basic picture is simple :-



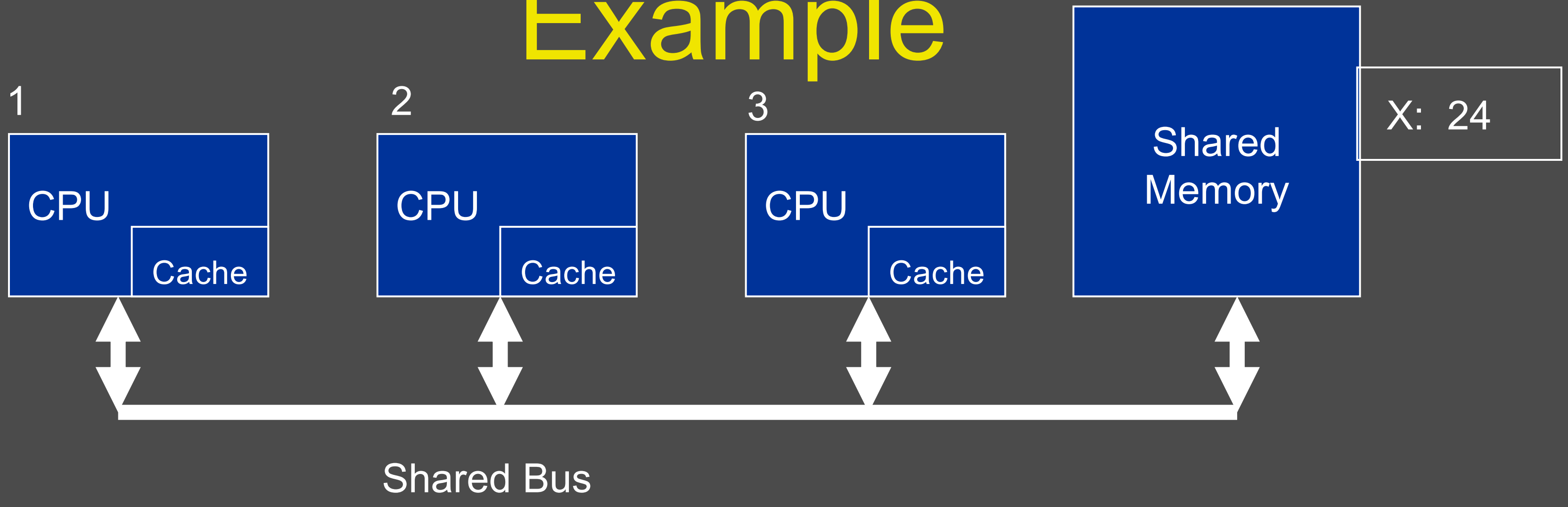
Organization

- Bus is usually simple physical connection (wires)
- Bus bandwidth limits no. of CPUs
- Could be multiple memory elements
- For now, assume that each CPU has only a single level of cache
- Other organizations (e.g., with a network) have NUMA issues

Problem of Memory Coherence

- Assume just single level caches and main memory
- Processor writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough
- What happens if two updates happen at (nearly) the same time?
    - Can two different processors see them out of order?

Example



Processor 1 reads X: obtains 24 from memory and caches it
Processor 2 reads X: obtains 24 from memory and caches it
Processor 1 writes 32 to X: its locally cached copy is updated
Processor 3 reads X: what value should it get?
Memory and processor 2 think it is 24
Processor 1 thinks it is 32

Notice that having write-through caches is not good enough

Cache Coherence

- Try to make the system behave as if there are no caches!
- How? Idea: Try to make every CPU know who has a copy of its cached data?
    - too complex!
- More practical:
    - Snoopy caches
        - Each CPU snoops memory bus
        - Looks for read/write activity concerned with data addresses which it has cached.
            - What does it do with them?
        - This assumes a bus structure where all communication can be seen by all.
- More scalable solution: 'directory based' coherence schemes

Snooping Protocols

- Write Invalidate
  - CPU with write operation sends invalidate message
  - Snooping caches invalidate their copy
  - CPU writes to its cached copy
    - Write through or write back?
  - Any shared read in other CPUs will now miss in cache and re-fetch new data.

Snooping Protocols

- Write Update
  - CPU with write updates its own copy
  - All snooping caches update their copy
- Note that in both schemes, problem of simultaneous writes is taken care of by bus arbitration - only one CPU can use the bus at any one time.
- Harder problem for arbitrary networks

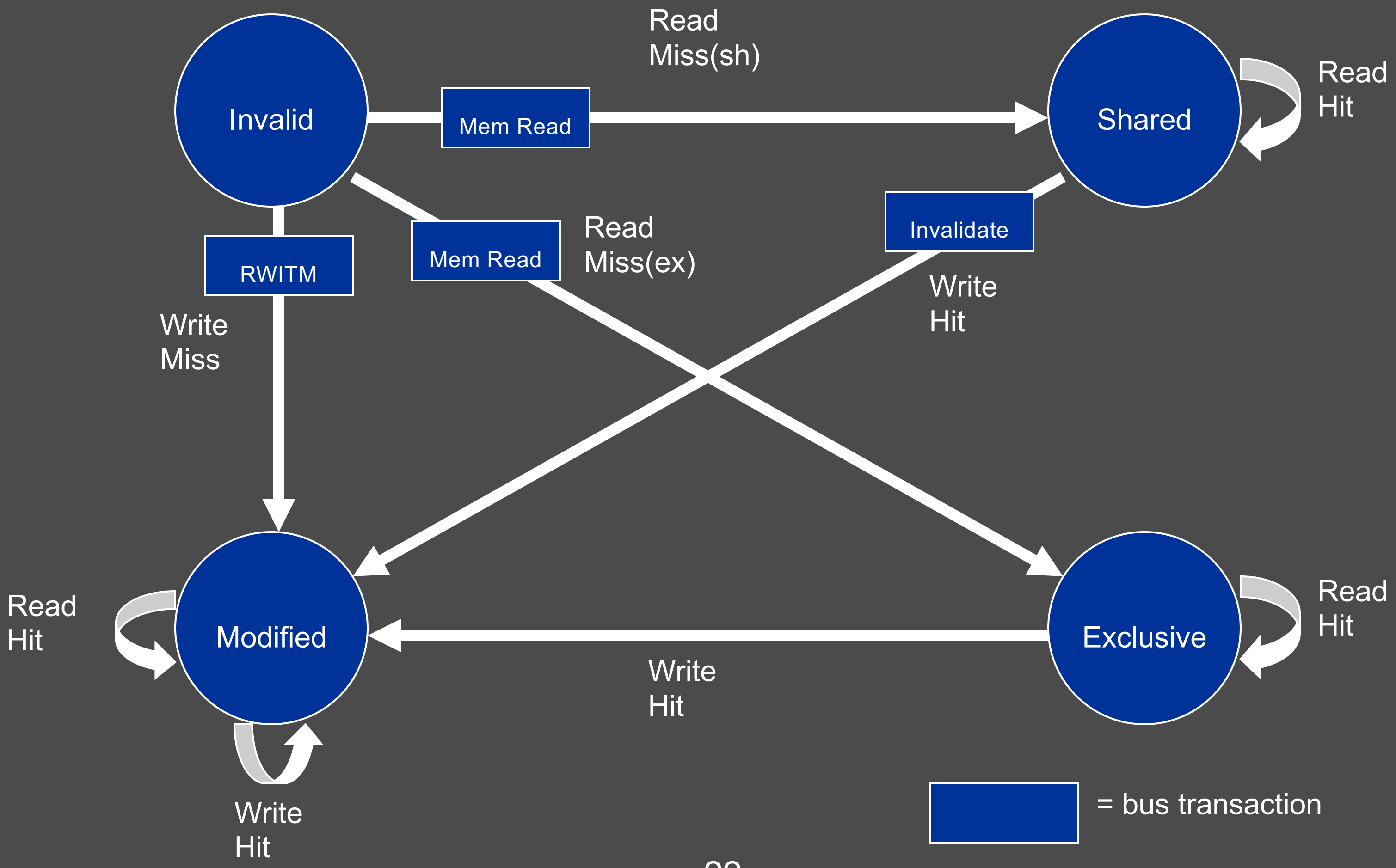
Update or Invalidate?

- Which should we use?
- Bus bandwidth is a precious commodity in shared memory multi-processors
  - Contention/cache interrogation can lead to 10x or more drop in performance
  - (also important to minimize false sharing)
- Therefore, invalidate protocols used in most commercial SMPs

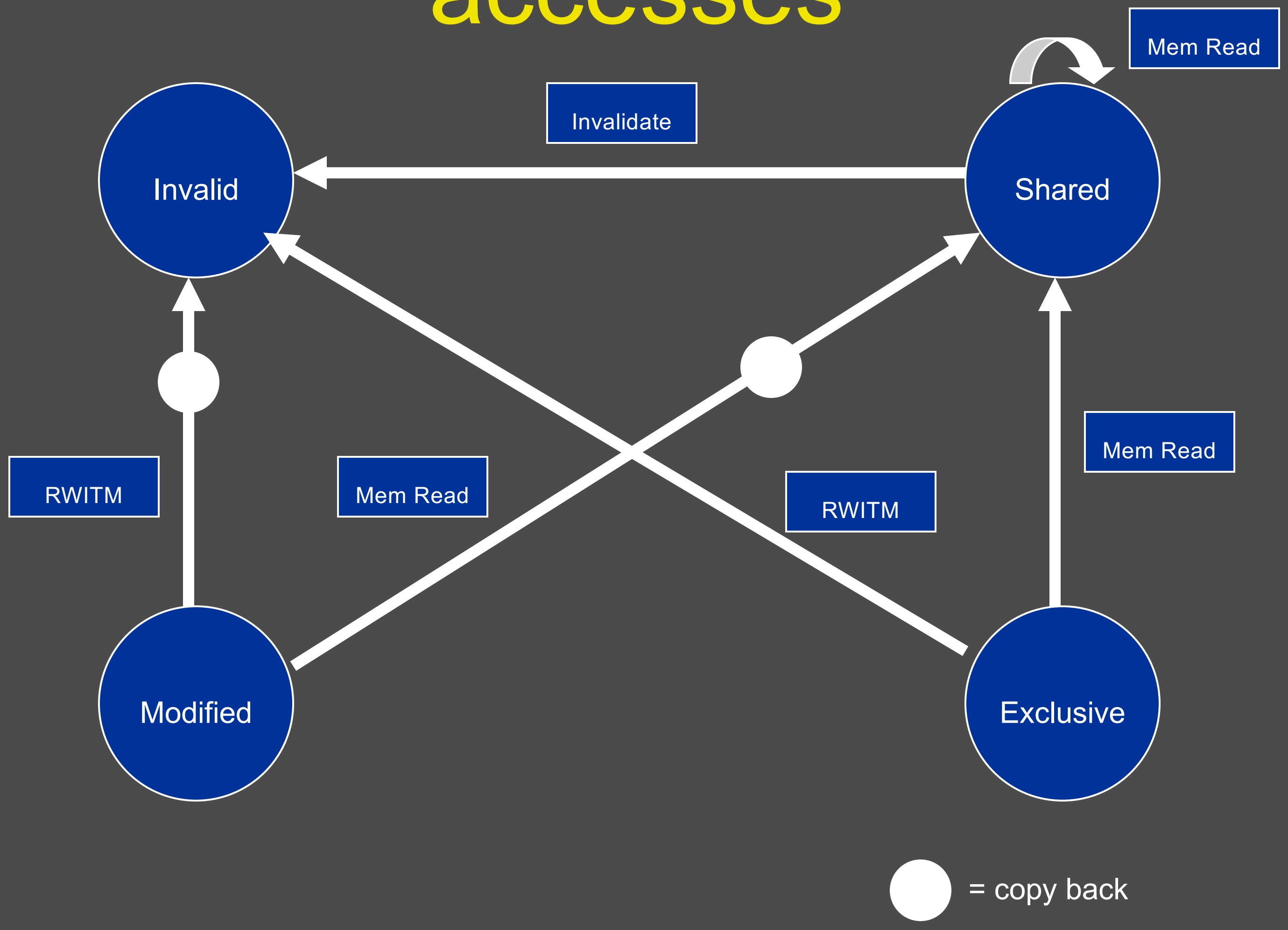
Implementation Issues

- In both schemes, knowing if a cached value is not shared (copy in another cache) can avoid sending any messages.
- Invalidate description assumed that a cache value update was written through to memory. If we used a 'copy back' scheme other processors could re-fetch old value on a cache miss.
- We need a protocol to handle all this.

MESI – locally initiated accesses



MESI – remotely initiated accesses



Mem Read

Invalidate

Invalid

Shared

RWITM

Mem Read

Mem Read

RWITM

Modified

Exclusive

● = copy back

MESI notes

- There are other protocols and minor variations (particularly to do with write miss)
- Normal ‘write back’ when cache line is evicted is done if line state is M
- Multi-level caches
  - If caches are inclusive, only the lowest level cache needs to snoop on the bus
    - Most modern CPUs have inclusive caches
    - But they don’t perform as well as non-inclusive caches

Cache Coherence summary

- Reads and writes are atomic
  - What does atomic mean?
    - As if there is no cache

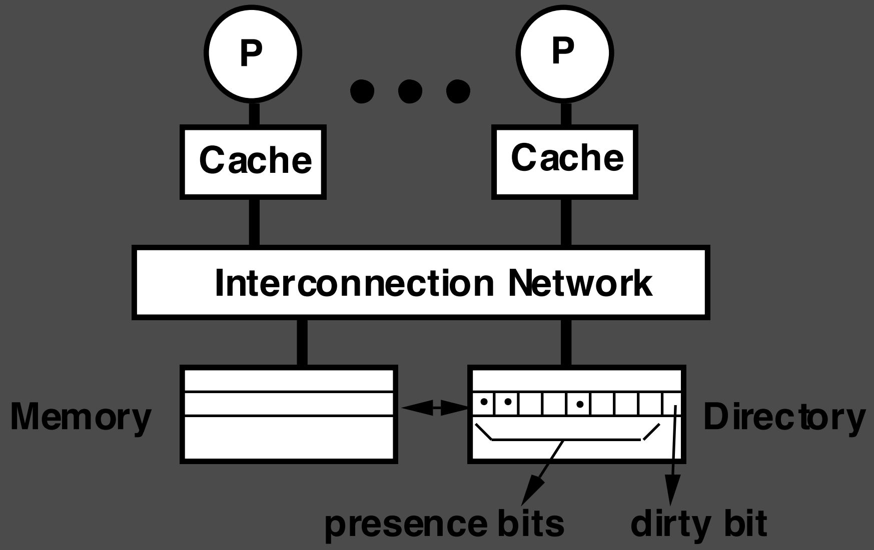
- Some magic to make things work
  - Have performance implications
  - …and therefore, have implications on performance of programs

Directory Schemes

- Snoopy schemes do not scale because they rely on broadcast

- Directory-based schemes allow scaling.
  - avoid broadcasts by keeping track of all PEs caching a memory block, and then using point-to-point messages to maintain coherence
  - they allow the flexibility to use any scalable point-to-point network

Basic Scheme (Censier & Feautrier)



- Assume "k" processors.
- With each cache-block in memory: k presence-bits, and 1 dirty-bit
- With each cache-block in cache: 1valid bit, and 1 dirty (owner) bit

- Read from main memory by PE-i:
  - If dirty-bit is OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to PE-i; }
- Write to main memory:
  - If dirty-bit OFF then { send invalidations to all PEs caching that block; turn dirty-bit ON; turn P[i] ON; ... }
  - ...

Key Issues

- Scaling of memory and directory bandwidth
  - Can not have main memory or directory memory centralized
  - Need a distributed memory and directory structure
- Directory memory requirements do not scale well
  - Number of presence bits grows with number of PEs
  - Many ways to get around this problem
    - limited pointer schemes of many flavors
- Industry standard
  - SCI: Scalable Coherent Interface