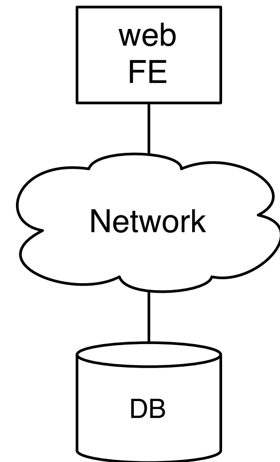
Overview of coherence and consistency

Coherence - motivation

Imagine we're building a web service that has a two-tier architecture:



- a web FE that receives requests from clients and generates HTML
- a DB that the web FE interrogates in order to populate the HTML with data

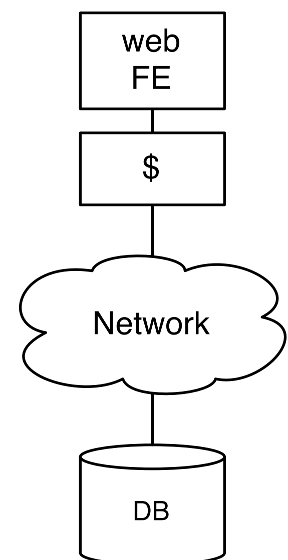
Advantages of this design:

- FE can be stateless; means that it is easy to recover from failure just by restarting it
- DB provides you with strong durability and consistency guarantees

Disadvantages:

- DB is over the network, so may become a performance bottleneck
    - the latency of a lookup is impacted by the network path and the complexity of the database, especially if involves hitting the disk
    - the bandwidth of the service is impacted by the network bandwidth, and the bandwidth of the database

Idea: introduce a cache on the same machine as the web FE



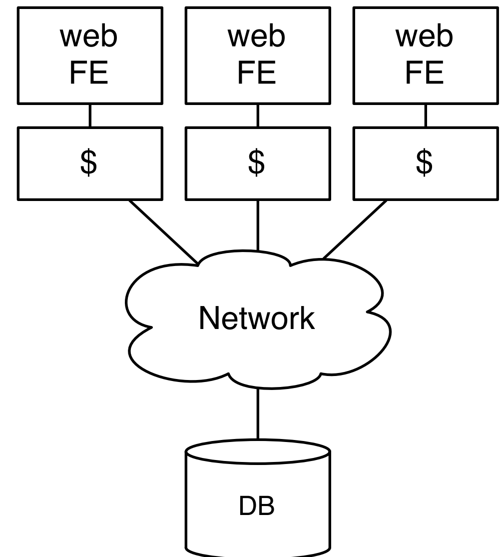
- cache stores data in memory, not on disk, so is fast
- cache is resident on the same machine as the FE, so has low latency and high bandwidth
- all reads and writes go through the cache; assuming there is locality in the workload, cache capacity can be less than DB capacity and things are still great

Issues that are created:

- writes eventually need to propagate from the cache to the database for durability
    - write-through: slow but safe
    - write-back: fast but unsafe
- need high hit rate to be effective
    - e.g., cache needs to warm up after restart
- consistency problems? no – reads see most recent write
- need an eviction policy

Eventually, the single FE will become a bottleneck of the system, so want to "scale" up by adding more FEs. Also get the fringe benefit of some fault tolerance; clients can contact any FE to get service.

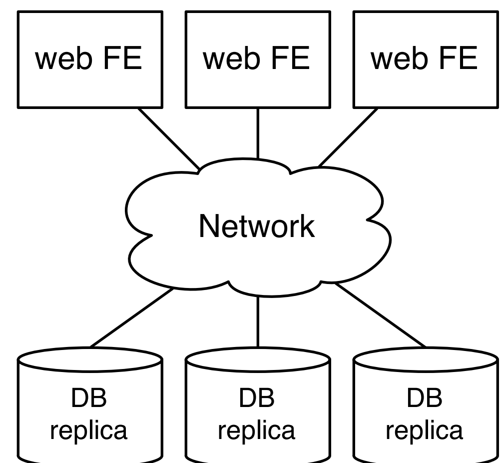
Issues that are created:



- entries in a cache can become stale if you're not careful, causing a read to see old data
    - write-through: most recent write is always in DB
    - write-back: most recent write could be in somebody else's cache
- multiple operations can show up at DB concurrently
    - need to make writes atomic – prevent two writes to the same record from resulting in a garbled record
- if propagate writes across caches, operations can show up in caches in different orders
    - FE 1 sees update A then update B
    - FE 2 sees update B then update A

Sidebar: keeping the truth durably in one place (a single DB) creates a single point of failure. Idea: replicate the DB for fault tolerance and higher capacity!

Issues that are created:



- a write operation needs to propagate to multiple replicas – writes are expensive
- need to preserve the order of updates at the replicas
    - otherwise they will diverge, leading to confusion. **Basically the same issue as in the caching case.**
- if you require all the replicas to be available when a write is satisfied, the overall reliability of the system goes down!
    - more likely to have a component in a failed state the more components you have
    - but, if you don't, then some replicas will miss updates
        - consensus protocols to solve – topic for a later day

Coherence

The classic problem is to maintain coherence of caches

- multiprocessor/multicore, and multiple memory caches
- multiple client caches for a shared network filesystem
- web browsers cache web pages
- DNS caches cache DNS entries

Basic issue: a write needs to propagate out to other caches

A memory system is coherent iff:

- a read must return the most recent write (no matter which processor reads)

Thus:

- every write must eventually be accessible via a read (unless overwritten)
- writes to a given location are seen in the same order by all processors

Strawman approach*: writethrough with broadcast invalidation*

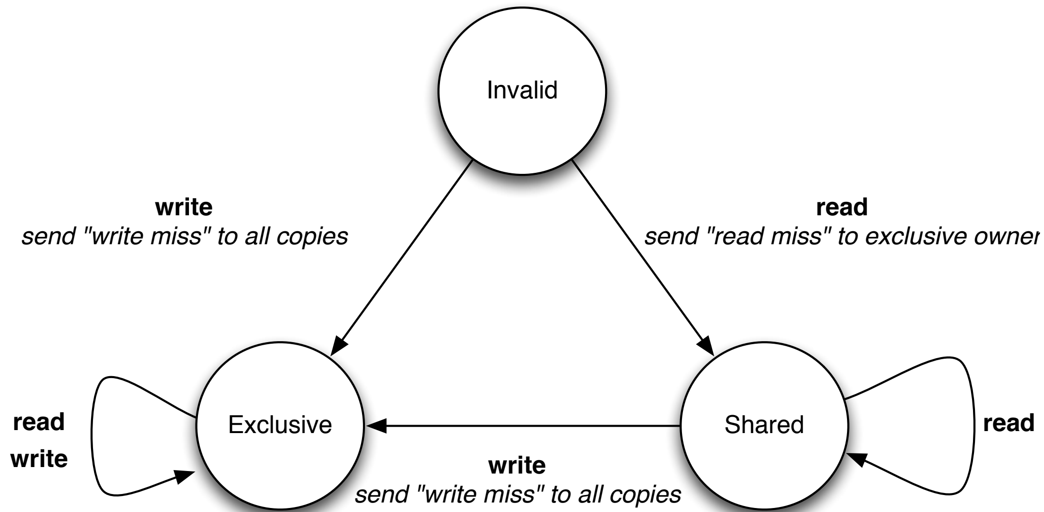
- On a read miss, fetch data from memory
- On a read hit, serve data from cache
- On a write
  - (a) write data through cache to memory
  - (b) broadcast an invalidation to all other caches
- Does it work?
  - Not if (a) and (b) are separate steps. Why? Nothing preventing some caches from reading old values and others from reading new values in between (a) and (b), depending on whether they have the old value cached or not.
  - Must:
    - lock the bus before starting (a)
    - receive an ACK from all caches during (b)
    - release the bus lock after (b) is finished
- Is it efficient?
  - No! Locked bus and broadcast on EVERY write – hugely expensive.

Writeback with exclusive ownership approach

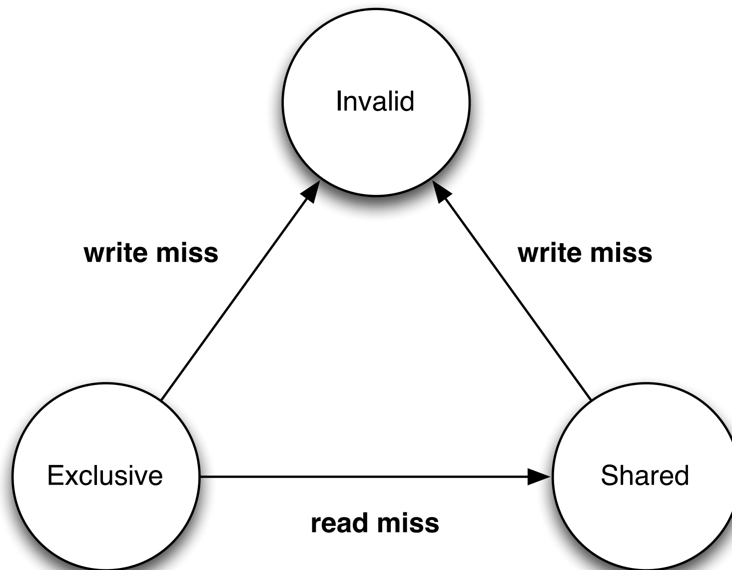
Idea: allow at most one cache to contain dirty data – designate that cache to be in an "exclusive" state, and only it can read/write data. Can have read sharing with no dirty data; all shared caches can read simultaneously.

State machines:

Messages arriving from CPU



Messages arriving from bus



Does this work?

- Great news: repeated writes are absorbed by the exclusive cache
    - bad news: if concurrent write sharing, cached item will bounce back and forth between caches

- Yes, but need to be careful about ordering and making transitions atomic. e.g., two caches may experience CPU-driven write misses at the same time.
    - the "send write miss to all copies" needs to be atomic and acknowledged, and must include the propagation of the data from other cache/memory
- Yes, but need to keep track of all of the copies of data (either shared or exclusive) so that you can propagate invalidations or downgrades

- how do you do this?
    - locks, data structures, and acknowledged messages in IVY
    - shared bus and snooping in multiprocessor caches

Observation: implementing cache coherence means keeping track of the state of caches

- TTL-based coherence – associate a time-to-live for a data item in a cache
    - invalidate the item when TTL expires
    - DNS works this way, as does NFS to some degree
- Advantage:
    - server doesn't need to keep track of state!

Consistency – motivation

Consider this:

```
    Initially all pointers = null, all integers = 0.

P1                                   P2, P3, ..., Pn

while (there are more tasks) {       while (MyTask == null) {
  Task = GetFromFreeList();            Begin Critical Section
  Task → Data = ...;                   if (Head != null) {
  insert Task in task queue              MyTask = Head;
}                                        Head = Head → Next;
Head = head of task queue;             }
                                       End Critical Section
                                     }
                                     ... = MyTask → Data;
```

Will this code work?

- for it to work, require that when P2 reads "Mytask → data", it sees the value that P1 wrote in there.
- What guarantees this will happen?
  - nothing – it depends on the memory consistency model!
    - note that this is not a coherence issue – its about the order in which reads/writes to differing locations in memory become visible
  - on some commercial machines, this will work, and on others, it won't

Another one to consider; mutual exclusion algorithm:

```
    Initially Flag1 = Flag2 = 0

P1                     P2

Flag1 = 1              Flag2 = 1
if (Flag2 == 0)        if (Flag1 == 0)
  critical section       critical section
```

Does it work?

- again, only if you assume something about the ordering of writes and reads to the two variables.

So, we have to reason about the memory consistency model. There is a rich spectrum.

Linearizability (or strict consistency)

- Every processor sees updates in the same order that they actually happened according to "real time"

```
P1:        W(x)1
-----------------------------    [is linearizable]
P2:  R(x)0       R(x)1
```

```
P1:  W(x)1
------------------------
P2:         R(x)2 R(x)2
------------------------       [is linearizable]
P3:    W(x)2
```

```
P1:  W(x)1
------------------------
P2:         R(x)1 R(x)1
------------------------      [is not linearizable]
P3:    W(x)2
```

```
P1:  W(x)1
-----------------------     [is not linearizable]
P2:        R(x)0 R(x)1
```

- Simple mental model: its as though all processes are timesliced on a single processor with a single memory
  - o reads/writes are ordered in memory according to when they happened in real time

- Subtlety: what if reads and writes aren't instantaneous?
  - o and a read and write overlap?
  - o Linearizability says "its as though the operation happened instantaneously sometime between the begin and end"

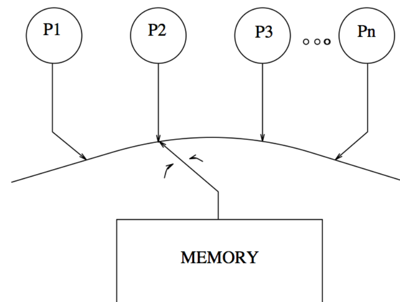
- Extremely simple model to think about, but extremely expensive to implement. Basically need to coordinate on every read and write, so that concurrent read/writes resolve according to real time.

Sequential consistency

Two conditions:

1. As if the operations of all the processors were executed in some sequential order, and
2. Operations of each individual processor appear in this sequence in the order specified by its program.

Mental model:




```
P1:  W(x)1
------------------------       [is sequential consistency; sliding beads on a string.]
P2:        R(x)0 R(x)1
```

```
P1:        W(x)1
------------------------       [is sequential consistency; neither processor can disprove]
P2:  R(x)1
```

```
P1:  W(x)1
------------------------
P2:         R(x)1 R(x)2
------------------------       [is sequential consistency]
P3:         R(x)1 R(x)2
------------------------
P4:     W(x)2
```

```
P1:  W(x)2
------------------------
P2:         R(x)1 R(x)2
------------------------       [is sequential consistency, but not linearizable]
P3:         R(x)1 R(x)2
------------------------
P4:     W(x)1
```

(The above two are examples of a “data race” – two memory operations on different processors with no intervening synchronization operation.)

```
P1:  W(x)1
-----------------------
P2:        R(x)1 R(x)2
-----------------------     [is not sequentially consistent; P2 & P3 order writes difft]
P3:        R(x)2 R(x)1
-----------------------
P4:    W(x)2
```

Implementing sequential consistency is possible, but has complications:

- **Program order requirement**
  - a processor must ensure that its previous memory operation is complete before proceeding with its next memory operation in program order.
    - ensuring completeness means getting an acknowledgement back from cache or memory
  - in a cache-based system, a write must generate invalidate or update messages for all cached copies, and the write can be considered complete only when the generated invalidates and updates are acknowledged by the target caches.

- **Write atomicity requirement**
  - writes to the same location must be serialized (i.e., writes to the same location be made visible in the same order to all processors)
  - the value of a write cannot be returned by a read until all invalidates or updates generated by the write are acknowledged (i.e., the write is visible to all processors)

There are some processor-level and compiler-level optimizations that sequential consistency makes impossible. So, right off the bat, we see there is a tradeoff between the ease of understanding of a memory model, and the efficiency of the memory system.

Causal consistency

- A read returns a causally consistent recent version of the data
  - If I received a message A from a node (or indirectly through another node), I will see all updates that node made prior to A
- Can something be causally consistent but sequentially inconsistent?
  - Yes – concurrent writes can appear in different orders at different processors!!
  - Means that causally unrelated writes don't need to be coordinated
  - Fast, fast, fast!

```
P1:  W(x)1
-----------------------
P2:        R(y)2 R(x)0     [is causally consistent]
-----------------------
P3:    W(y)2


P1:  W(x)1
--------------------------------
P2:                R(y)2 R(y)0    [is not causally consistent]
--------------------------------
P3:     R(x)1 W(y)2
```

Release consistency

- A process must acquire a block of data before being able to modify it. Modifications only need to be written back before a release.
- Acquires and releases are sequentially consistent.

Eventual consistency

- A read returns a recent version of the data
  - o but not necessarily the most recent
  - o and not necessarily consistent with another CPUs read

- If no updates happen for long enough, all CPUs eventually agree on the state of memory