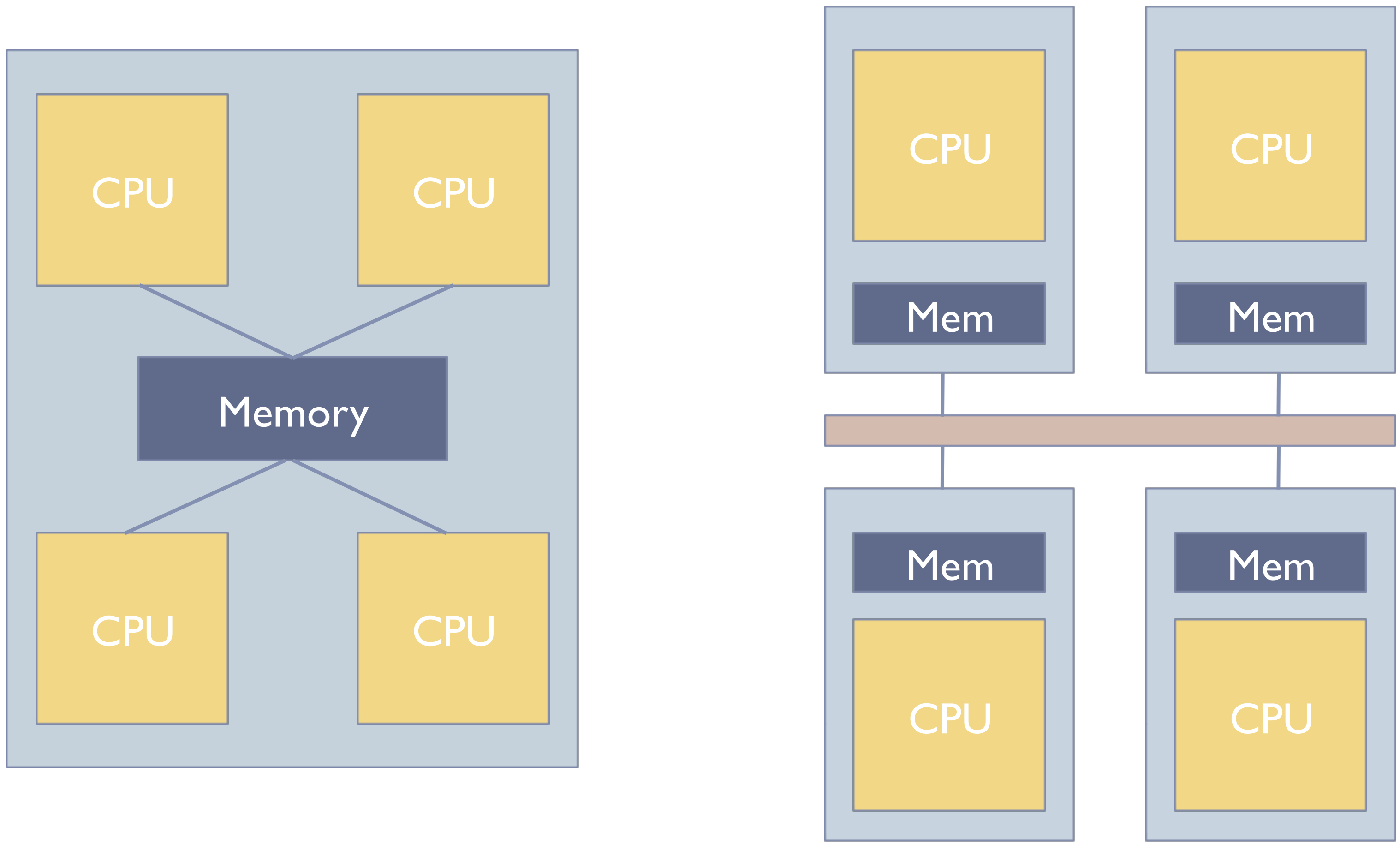
Distributed Systems

Shared vs Distributed Memory

- Standard systems you encounter in your daily life come in one of two forms: shared or distributed memory.
- **Shared memory**: A relatively small number of processors are attached to a common memory subsystem.
  - Any processor can address any location in memory directly.
  - Memory subsystem deals with certain important concurrency control issues, such as cache coherence.
- On the other hand, in a distributed system, we typically have **distributed memory**.
  - Memory owned by a subset of processors that can directly address it.
  - Other processors must send request to processor that owns a block of memory so it can perform memory accesses on behalf of the requestor.

Shared vs. distributed memory



Shared memory

- Shared memory is attractive. Why?
  - It is easy to write code that runs very fast.
  - Distributed systems have to suffer performance hits due to network latency and bandwidth.
  - Distributed systems require two classes of operations for local versus remote data.
  - The primary difficulty in shared memory comes from concurrency control to deal with correctness.
- Performance is also hard to predict in a distributed environment due to potential external sources of resource consumption.
- Distributed systems also have to tolerate link or node failures.
  - These are very rare in shared memory systems, and typically mean you have bigger problems with your computer.

Distributed Shared Memory

- Shared memory is programmable – we all accept that. Just write code in your favorite language, addressing memory just like you always have.
- DSM brings this abstraction to the distributed world. Treat remote nodes as holding memory in a deeper level of the memory hierarchy than you do in the single machine.
- Hide the message passing or other techniques for doing remote addressing from the user under a runtime library or compiler.
  - This also means that by hiding it from you, it can be implemented better than had you coded it by hand.

Target Audience

- When is DSM a good choice?
- Typically when you want to run a single program in a distributed system without explicitly dealing with the layer that binds the distinct nodes together.
  - Furthermore, when you want to be able to run the same code in a non-distributed system.
  - Either single CPU or in a shared memory system.
- Programs targeting parallel shared memory machines map easily onto DSM.

UMA, NUMA, and DSM

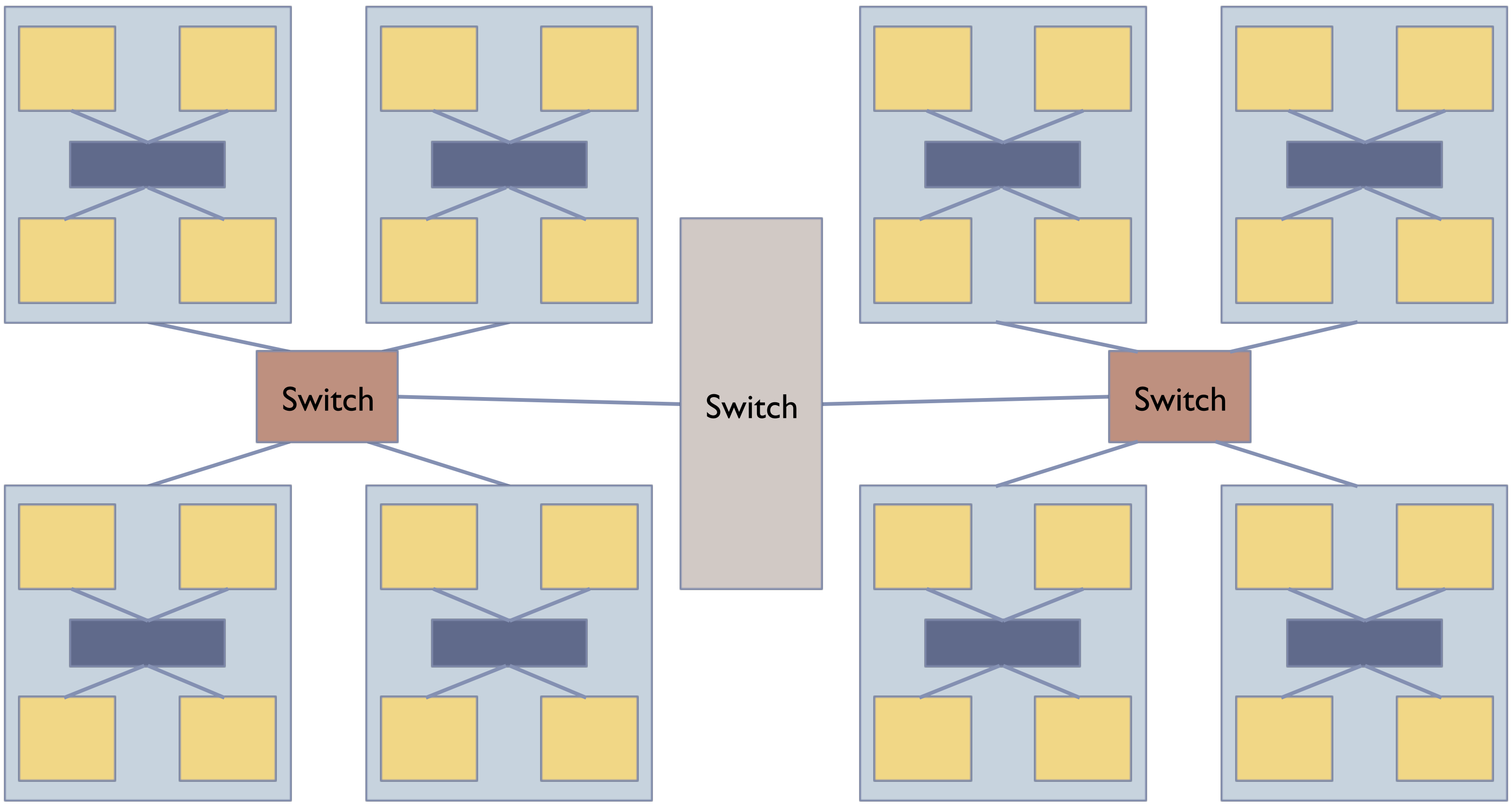
- In small SMPs, memory access times are uniform. All memory accesses to main store take the same amount of time.
- As SMPs get large, the cost for hardware to support UMA becomes expensive, so memory access time becomes non-uniform. Hierarchy of memory: tightly coupled processors have UMA, groups of those have slightly higher latency, groups of these have even more, and so on.
- DSM is similar to NUMA – remote machines are just a very high latency memory.

SGI Origin 2000
128 CPUs in NUMA configuration.

Cray-link interconnect supporting NUMA.

NUMA

- NUMA architectures. Indicated switches that bridge distributed memories into a single NUMA memory are very expensive as CPU count scales.



NUMA and performance

- Performance optimization is hard on NUMA machines.
- The same issues arise in DSM.

- Locality!

- Best performance is when the memory you access most frequently is nearby.
- Caching can help, but as we'll see later, granularity of caching can cause false sharing and poor performance.
  - Sometimes worse than performance with no cache at all!
- Also difficult to anticipate when memory access pattern is a runtime property, not a static one.

Message passing and DSM

- Programming typically is asynchronous. "get" and "put" are one sided.
  - Intended to be similar to operations like "x=a[4]" and "a[4]=y" respectively, where "a" is a DSM shared variable, and "x" and "y" are local to the accessor.
- In DSM, variables may be shared. This makes some correctness issues arise, as the owner of the data may see the data modified without their consent.
  - One sided operations don't require explicit participation of the user app. code on the other side – runtime layer takes care of that behind the scenes, for better or worse.
  - Requires protection, just like in threading with critical sections and mutual exclusion.

Efficiency

- The book states that DSM programs can be made to perform as well as the equivalent explicit message passing program.
- This is likely true, but only for specific cases. In general, DSM does not scale well.
  - This is one of the reasons you don't see it used frequently in practice.

- One of the primary reasons for this is that DSM systems typically try to make the programmer as unaware as possible of the distribution of memory underneath their program.
  - This means coherence protocols in software, and other consistency mechanisms. Those can be very network intensive, and will not scale well.

Implementations

- NUMA can be considered a hardware form of DSM.
  - These perform well, but at the cost of $$$. They can be expensive.
- Paged virtual memory uses the VM system to hide remote memory behind a well defined region of the address space of each process.
  - This provides transparency to the app, but requires support at the OS level.
- Middleware solutions are the most portable and least intrusive on the platform. No OS or hardware support necessary.
  - Sometimes can take advantage of hardware features though, such as DMA from network devices.

DSM and abstraction

- One of the key features of a DSM system is abstraction.
- Provide the same abstraction to the programmer as the existing language that they are working in.
- This is intended to address usability from the perspective of the programmer.
  - Easier to manage parts of a program if all data is addressed equally, instead of some via variables and others via explicit send/receive calls.

Types of DSMs

- Byte-oriented
  - Distributed shared memory addressed at the byte level, just like any other variable in a language like C.
  - Page-based schemes typically support this.

- Object-oriented
  - Objects are shared and their contents manipulated by external processes via get/set methods, and possibly higher level abstractions (such as queue push/pop methods).
  - RMI would be an example of this, although acquisition of the remote objects requires special calls that only apply to remote objects.
    - Not all objects treated equally.

Types of DSMs (2)

- Immutable data
  - Linda!
  - The tuple space is the shared memory, and it is viewable by all participants.
  - Operations are the "put" and "take" (out/in) operations of Linda.
  - Data in the tuple space is never modified.
    - If a process wants to modify an element, it must extract the tuple from the tuple space and put another in it's place with the modified data.

  - You implemented a tuple space where the space existed within a single process. The Linda model doesn't prohibit the tuple space from being distributed itself, as long as the semantics of the tuple space are maintained.

Synchronization

- Like threads, when multiple execution contexts share a common store, concurrency control mechanisms are necessary to prevent detrimental non-determinism.

- So, most DSM systems provide abstractions such as locks and semaphores so programmers can use standard locking disciplines to protect critical sections and data.

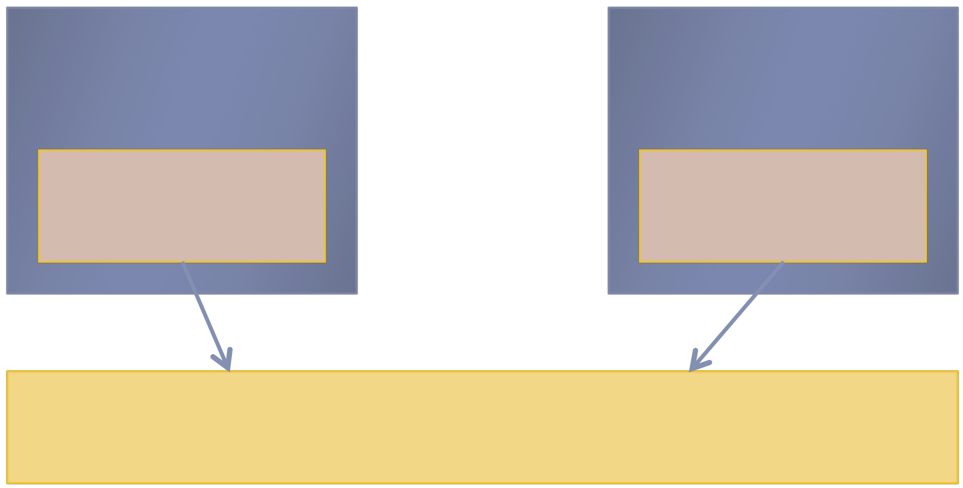
- Virtual memory based systems can deal with atomic instructions such as **`testAndSet`**, but at a potentially high performance cost.

Consistency

- Consistency is all about ensuring that a set of concurrently executing processes have a view of the world that makes sense.
  - A system that provides consistency prevents concurrent processes having conflicting views of the state.
- The issues that arise are the same as those we saw for replication schemes. We may desire sequential consistency or linearizability.
- One interesting weaker consistency model that we may desire is called *coherence*. This is what is provided in SMPs with respect to the cache.

Caches and hierarchical memories.

- Think about a simple SMP. Typically we have a single shared memory, but distinct caches on each processor.



- This isn't very different from a distributed shared memory. The caches are local to each processor, and the shared memory is remote relative to the caches.

Caches and hierarchical memories

- What issue arises?

- Say a processor P1 reads a memory from the shared store, and then modifies it.
- This modification occurs in the local cache of the processor. Only when an operation that invalidates the cache line occurs does the modified data get flushed to main memory.
- Now, what if another processor reads the address that is cached at P1? We would want that data to be accessed by P2.

Caches and hierarchical memory

- Clearly we would want to have P2 see the updated version that P1 holds in it's cache.
- On the other hand, if P2 reads an address that P1 has never seen, P1 never should care if P2 reads or writes to it if P1 never accesses it.
  - Reads or writes to distinct addresses that don't reside on a common cache line don't have any constraint on ordering.

- A coherence scheme provides this. It is weaker than sequential consistency, as it focuses only on ordering of writes to the same place in memory by multiple processors.
  - Writes to distinctly different places in memory are independent.
- SMPs implement this in hardware.

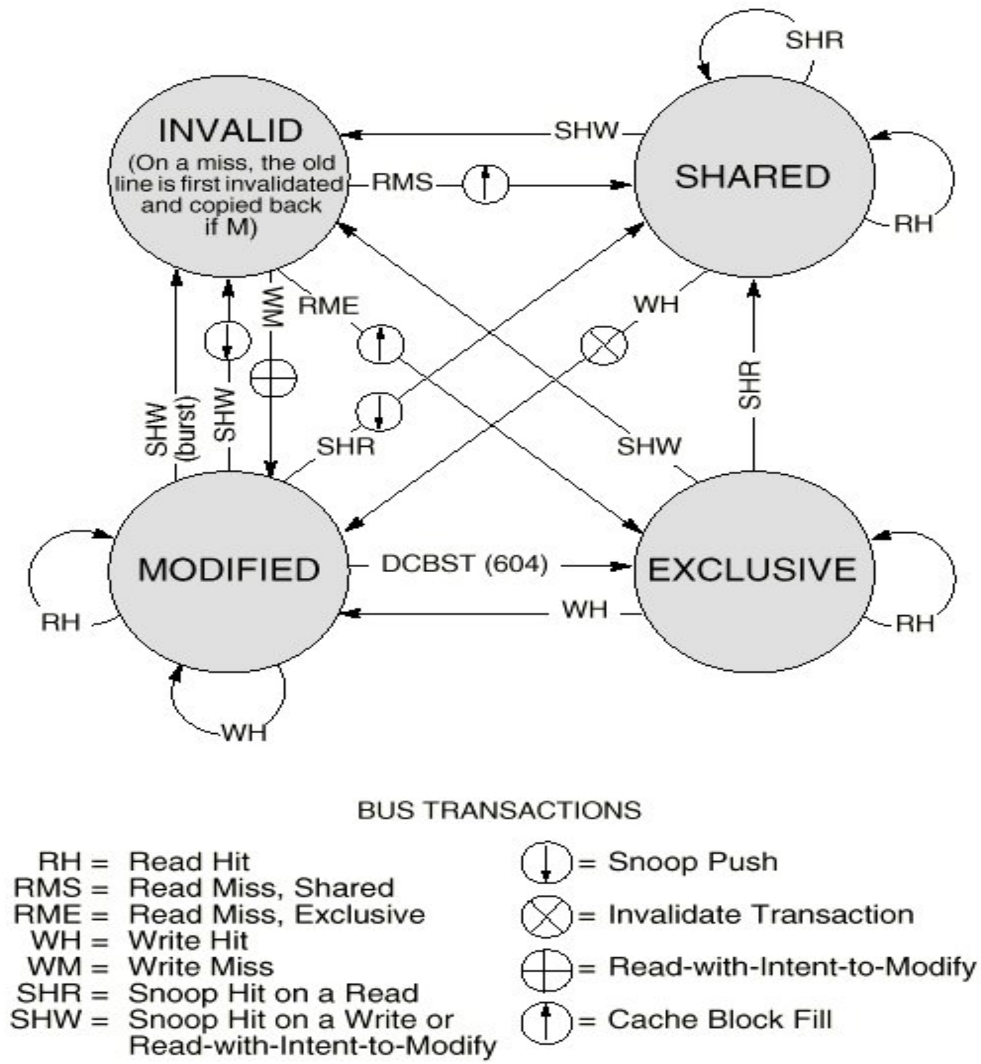
Coherence in SMPs

- We mentioned this briefly earlier in the term.

- Cache coherent SMPs use protocols in hardware to maintain coherence.
- Cached data decorated with a few bits of state, beyond the simpler clean/dirty required in a single processor cache.
- Coherence protocol defines how these states change and when memory moves from cache to memory and memory to cache based on observed transactions on the shared memory bus.

- For large systems, such as NUMA SMPs, CC-NUMA protocols require more sophisticated schemes (such as directory-based protocols) when no shared memory bus exists that each cache can snoop on.

Coherence protocol: MESI example



Weak consistency

- If the DSM system is aware of synchronization used to protect data, then it can relax it's consistency model.
- Say a data element is protected by a lock. Then assuming the other processes obey the locking discipline, there is no reason to propagate updates to other participants until the lock protecting it is released.
- Schemes that are unaware of synchronization primitives must be paranoid and propagate updates right away.

Weak consistency (2)

- The "weak" in this model means that updates don't propagate immediately, but at the end of critical sections.

- So, memory can be inconsistent when a processor is in a critical section, but the locking discipline means that other accessors won't see the inconsistency since the synchronization scheme prevents them from accessing critical sections anyways.

- I see this as having the same spirit as a transaction – briefly allow part of the system to become inconsistent, but in a very controlled manner that has a well defined mechanism to restore everything to a consistent state.

Updates

- The critical component of a DSM system (as in a replication system) is how to propagate updates.
- Two schemes are most common:

- Write-update:
  - When writes occur, the updated data is propagated.
- Write-invalidate:
  - When writes occur, a notification that any other version of the data is invalid is sent.
  - Propagation only occurs when reads occur. Multiple writes may occur before propagation actually happens.

Virtual memory: 1 page refresher

- Apps get illusion of flat, contiguous address space. Under the covers, the system maps this address space all over memory, possibly to disk or into other apps.
  - e.g.: mapping a file into memory.
- Memory partitioned into pages (possibly a few K each)
- Page faults occur when an app requests an address that is not in physical memory, and the system replaces an existing page that isn't in use with the one requested.
  - High overhead, especially if the paged data is out on disk.
- VM support provided in operating system.
- DSM works by mapping part of the address space to hold distributed data.
  - Faults, paging out, and read/write permissions on a page related to the underlying get/put protocol.

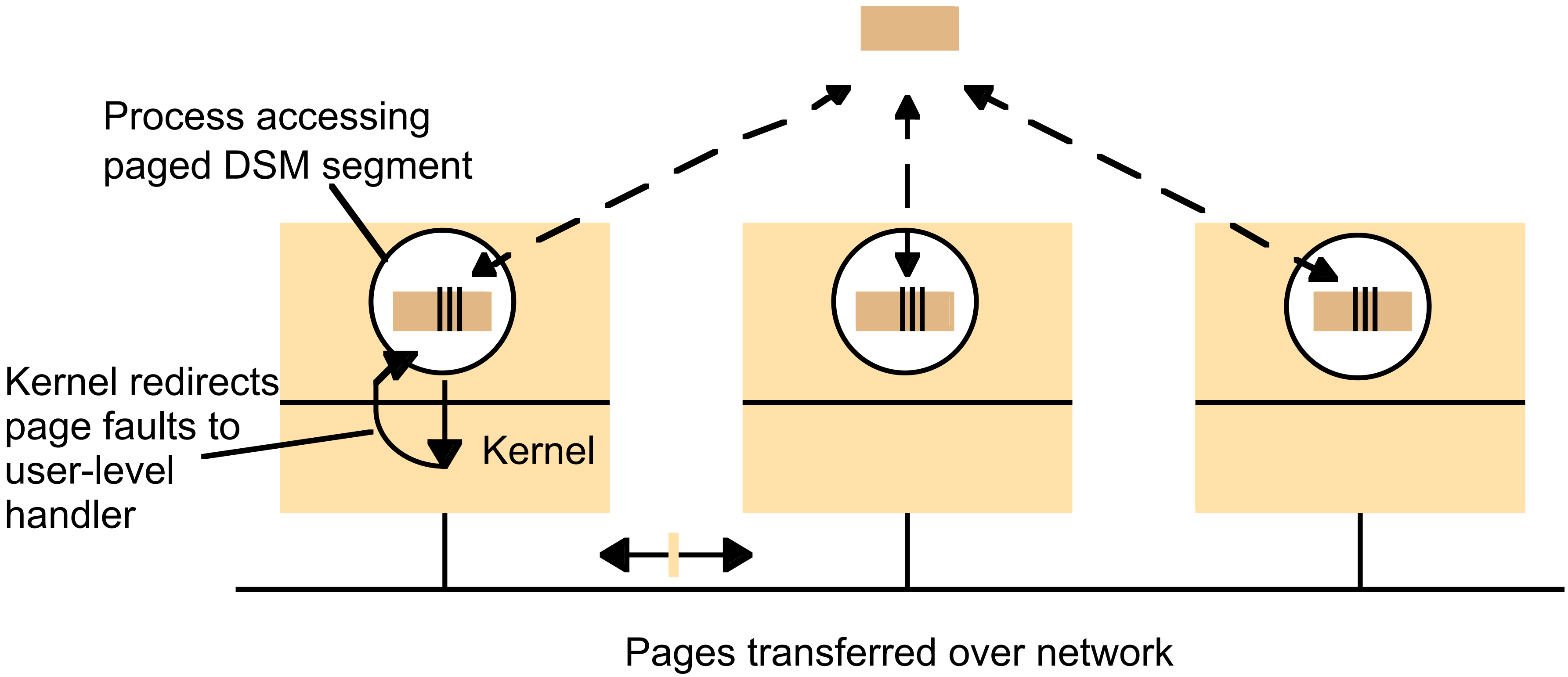
Granularity

- Like cache-based memories, we care about the granularity of memory regions that operations like invalidations apply to.
- In a cache, these apply to cache lines.
- In a page-based DSM, these apply to pages. Pages can be quite large.
- It is entirely possible that two processes will work with memory that resides in the same page, yet do not actually conflict.

Granularity (2)

- Too coarse of a granularity can cause invalidations to result unnecessarily, even if processes aren't conflicting on the shared memory.
- This is known as *false sharing*.
  - Can result in thrashing.
- This isn't unique to distributed systems.
- Tightly coupled programs written poorly in a cache coherent SMP can see this at the cache line level.
  - The need to avoid this is part of the `folklore' of parallel programming. Parallel programmers typically learn to write programs in a form that avoids this.
  - Writing code to avoid this isn't hard really. It's just something a first-time SMP programmer can accidentally wander into.

Page-based scheme

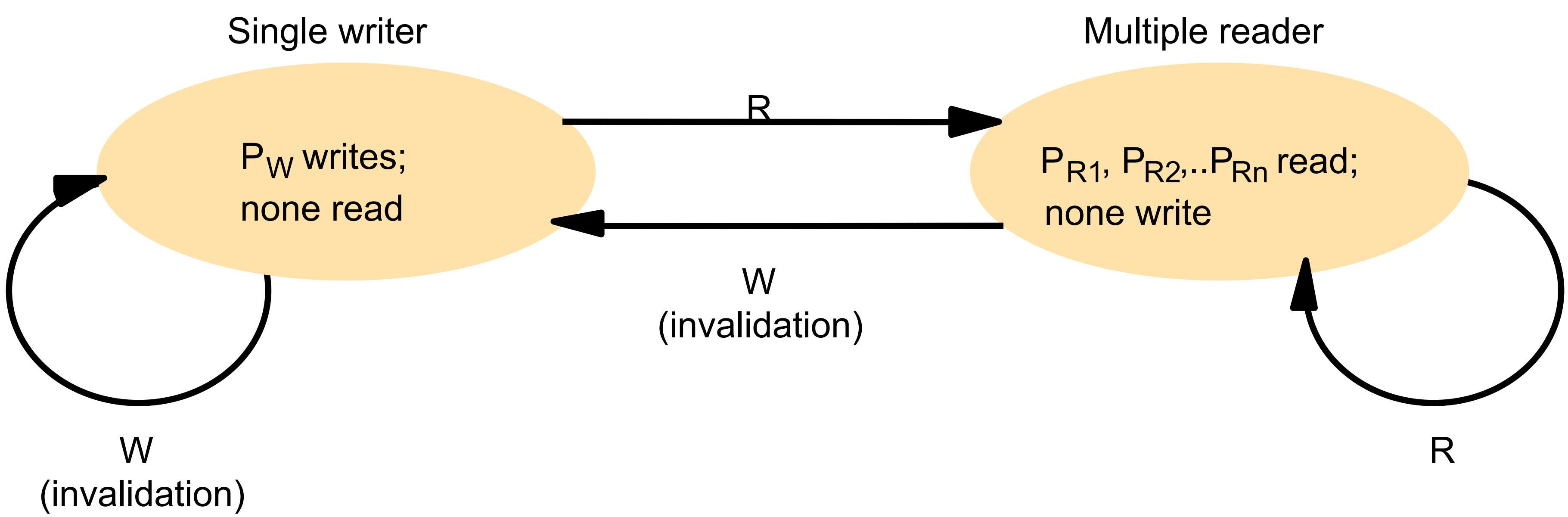


Pages and writes

- Write updates are high overhead for paging schemes, so page fault schemes don't mesh well with write-update.
  - From a performance perspective.
- Write-invalidate is more compatible, as are buffered write-updates.
  - Buffering means multiple writes may occur before an update is propagated.

Write-invalidate protocol

- Looks like a simple cache coherence protocol actually.



Note: R = read fault occurs; *W* = write fault occurs.

Write invalidation

- Updating processes have read/write permission to a page. No other process may read or write to it.
- Reading pages, processes have read-only permissions to a page.
- State transition diagram shows how writes and reads transition between the states.
- Detail is how to achieve this.

Invalidation protocols

- How do we invalidate a page?
- One approach is to have a centralized manager that knows the mapping of pages to the owners of them.
- Client that is writing to a page contacts manager to acquire the copy set for the page.
  - Copy set is the set of other clients that have read the page.
- The client then multicasts the invalidation request to the clients in the copy set.
- Centralized manager sets owner of page to the client that first makes it in to get the copy set.
  - This prevents situations where two clients try to write and invalidate at the same time.

Invalidation schemes

- Centralized manager is easy, but has a bottleneck.
- Distributed algorithms have been proposed, in which the set of processes help find who owns a page.
  - Remove performance bottleneck.
  - Penalty is complexity in DSM system for determining ownership and performing invalidations.
  - Distributed algorithms can also be built to avoid dependence on multicast.
    - Good for platforms without multicast support.

Release consistency

- Sequential consistency allows the system to behave the way programmers expect, but at a cost.
- Release consistency relaxes this to reduce the overhead.
- Exploits knowledge of synchronization primitives.
  - Semaphores
  - Locks
  - Barriers
- Using this, the system can reason about what possible operations can occur assuming all processes obey the locking discipline.
  - If a process uses memory without properly locking it, all bets are off. This is considered a bug that the programmer is responsible for, not the DSM system.

Release consistency

- Accesses are distinguished as competing vs non-competing.
  - Competing accesses are those that may occur concurrently where one is a write.
  - Two reads are non-competing.
  - Writes on data protected by locks are also considered non-competing, as the competition would have occurred in lock acquisition.
- Lock acquisitions are considered competing operations.
  - These are further divided into acquire vs release operations.
- Dividing up accesses into special classes assists the DSM system in knowing when high-overhead consistency operations must be performed.
  - Allows the DSM system to avoid overhead in a pure invalidate model.

A key observation

- Constraining overlapping operations can yield executions equivalent to sequential consistency without requiring the system to strictly obey sequential consistency.
- Rules:
  - RC1: Before read/write on a process, all previous acquire operations by the process must be performed.
  - RC2: Before a release on a process, all previous read/write operations by the process must be performed.
  - RC3: Acquire and release operations are sequentially consistent relative to each other.

- So, we require SC with synchronization primitives, but not reads and writes. This reduces the overhead, as sync. primitives will occur less frequently.

Hardware support

- DSM is a useful abstraction, but the overhead from maintaining consistency in software can cause the overall performance to be very poor or unpredictable.

- The goal is to provide a coherent view of a set of disjoint memory spaces. Hardware standards were created to push some of the work into the network layer of commodity machines to overcome this software performance problem.

- Examples: IEEE SCI (Scalable Coherent Interface), InfiniBand.
  - InfiniBand is one of the dominant high performance cluster interconnection network technologies today.

Concluding remarks

- DSM is attractive because it gives a shared memory programming model in a distributed system.
- Performance is difficult due to consistency constraints.
- You may encounter software layers that are somewhere between DSM and message passing.
  - Example: ARMCI – Aggregate Remote Memory Copy Interface.