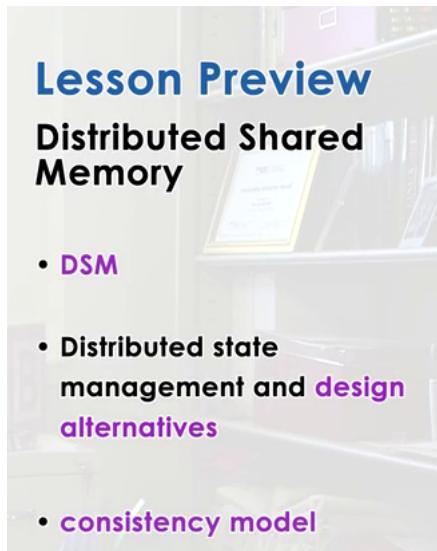


P4L3 - Distributed Shared Memory

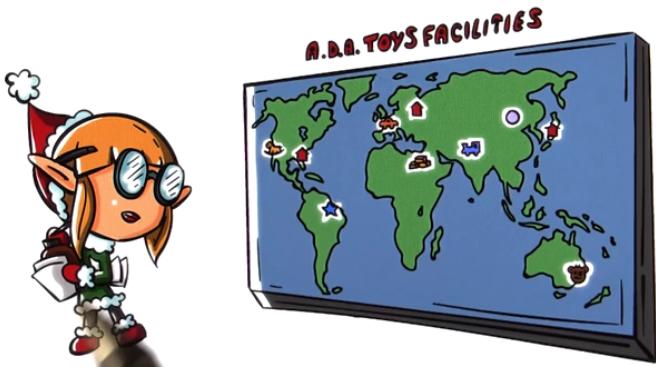
01 - Lesson Preview



Lesson Preview
Distributed Shared Memory

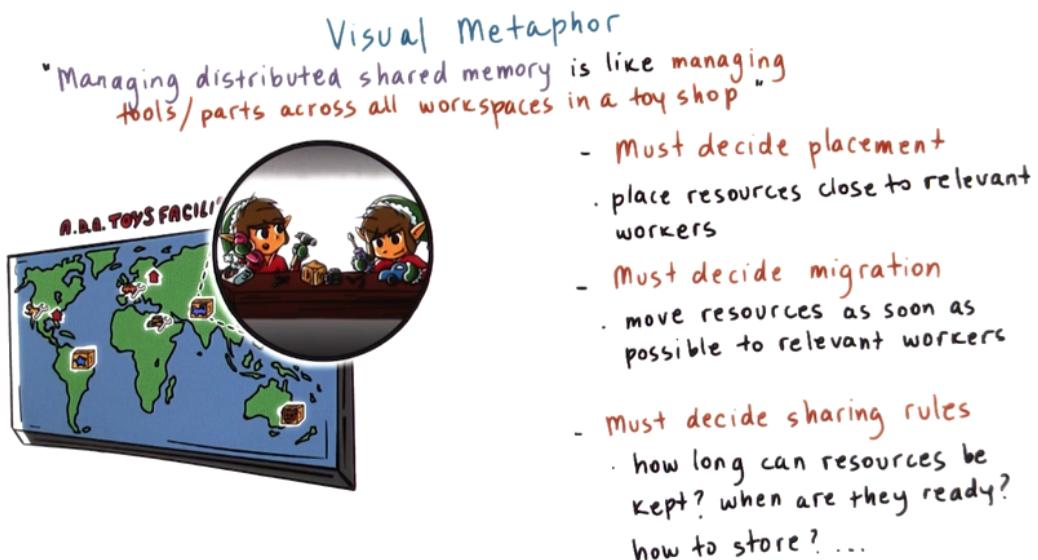
- DSM
- Distributed state management and design alternatives
- consistency model

Visual Metaphor



In this lesson we will talk about distributed shared memory systems for DSMs. These are an example of applications that perform certain distributed state management operations. Now in this lesson a lot of what we will talk about will be in the context of memory management. Still the methods that we will discuss will apply to other types of distributed applications. Specifically we will present certain design alternatives with respect to DSM systems. And we will discuss the trade offs that are associated with those decisions. In addition we will discuss several important consistency models that are commonly used in systems that manage distributed and shared state

02 - Visual Metaphor



Visual Metaphor

Managing distributed shared memory is like managing tools/parts across all workspaces in a toy shop

- Must decide placement
 - place resources close to relevant workers
- Must decide migration
 - move resources as soon as possible to relevant workers
- Must decide sharing rules
 - how long can resources be kept? when are they ready?
how to store? ...

In this lesson we will be talking about another distributed service specifically distributed shared memory. Now recall that in a previous lesson on memory management we made a visual analogy between the memory contents of a process and the parts used by workers sharing a single workspace. Holding off of that analogy managing distributed shared memory is like managing the toy parts or even the tools across all the work spaces in a toy shop. As you can see from this image it is very similar to the one that we used for the distributed file systems because we want to show that there are multiple distributive nodes that are working together towards providing a service. The service that's provided is the sharing of the tools and the toy parts among workers that are working on multiple such work benches. So you imagine even in a single location like this there will be multiple such workbenches with multiple workers using certain toys and toy parts and tools and this is what we want to facilitate the sharing of those tools and parts. For the management of tools and toy parts in a toy shop there are three major decisions that must be made. The first one is how are we going to place all these resources in the distributed environment. Next we must decide how and when to move tools around the different shops or the different workbenches in those shops and finally we must establish some rules how those toy parts and tools will be shared. For instance regarding placement we would probably decide to place those resources close to the workers that are currently needing them to perform their task. When a worker on one work bench is done with a particular resource and another worker on another workbench or potentially in another location needs that resource we should move it as soon as possible. And finally we must have policies on how long can a worker keep a tool on their desk or when and how workers should let others know that a tool or a toy part is ready and is available for use by others. Or for instance when should the tools or the toy parts be placed into a storage facility that others can access as well. For managing distributed shared memory we must make similar types of decisions however clearly the details are going to be very different. For instance when deciding placement we should typically place the contents of memory or the contents of a page in a physical memory that's close to a particular process. This would be the process that either created that content and or will be using that content. When a process needs access to memory that's stored in a page param on a remote node we should come up with some policy how to migrate that content to the local node. For instance we can copy that content that memory page from the remote to the local physical memory. Since the memory page may be concurrently accessed by multiple processes and in addition since we may have multiple copies of that page in different locations in the system it's important to establish some rules how the different operations are ordered. How they're propagated across the different copies so that there is a consistent view of the entire memory.

Visual Metaphor

"Managing distributed shared memory is like managing tools/parts across all workspaces in a toy shop"

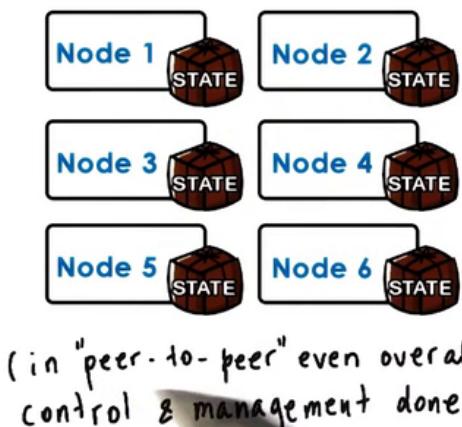
- Must decide placement
 - place memory (pages) close to relevant processes
- Must decide migration
 - when to copy memory (pages) from remote to local
- Must decide sharing rules
 - ensure memory operations are properly ordered
- Must decide placement
 - place resources close to relevant workers
- Must decide migration
 - move resources as soon as possible to relevant workers
- Must decide sharing rules
 - how long can resources be kept? when are they ready?
· how to store? ...

03 - Reviewing DFSs

In the previous lesson we talked about Distributive File Systems. We said that Distributive File Systems were an example of a distributive service where the state the files were stored on some set of nodes the file servers. And then the file servers were accessed by many distributive nodes clients. All of the file access operations open read write on these files. Where then service that was provided by these servers and that these client nodes were requesting from them. Our primary focus of the Distributed File System lesson was the caching mechanisms. These are useful so that server data is cached on the clients. And this is a mechanism that improves the performance that's seen by the clients. The clients can now access these files more quickly. Caching is also a mechanism that enhances the scalability of the distributed file system overall of the servers. Because some of the operations are now performed by the clients the servers don't have to perform every single access and therefore they can scale to support a larger number of clients and serve larger number of files potentially. What we didn't talk about is what happens in the situation when there are multiple file servers? And how do they control and coordinate the access of shared state of shared files among them? We also didn't talk about situations in which all of the nodes in the systems were both servers storing files and allowing access to those files as well as clients sending request to other nodes in the system. In such a scenario there really isn't a clear distinction between client nodes and server nodes. Every single one of the nodes would be performing all of the services. In this is lesson distributed shared memory we will talk about these kinds of scenarios the distributed applications where all the nodes in the distributed environment are both the servers that provide the state and provide the access to that state and also issue requests to other nodes in the system to their peers. So perform some of the client functionality that we saw before.

04 - Peer Distributed Applications

"Peer" Distributed Applications



Each node

- "owns" state
- provides service

=> all nodes are "peers"

Examples: big data analytics, web searches, content sharing, or distributed shared memory (DSM)

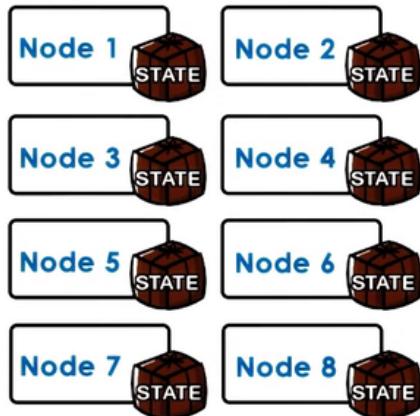
Let's talk now about these cases where there isn't a clear distinction between servers and clients. For instance if you look at all of the big data applications that we hear about today the state of the application is distributed and stored across all nodes. The processors on these nodes run certain applications such as the analytics codes the mining codes searching for patterns or searching in order to answer your Google web search request or some other applications. And they access the state that's distributed across all the other nodes in the system. The key about these applications is that every node in the system owns some portion of the state. By own here we mean that there is some state that's locally stored in a particular node or that's generated by computation that's running on that node. So the overall application state is the union of all of these pieces that are present on every one node. It's as if all the nodes in the system are peers. They all require access to the state anywhere in the system and they all provide access to their local storage state to any of their other peers in the system. For instance think about the large web search applications like Google. The state in that application which is the information about every single webpage in the world is stored on many machines. And at the same time it's also processed on all of them. Whenever you type in a web search it may get processed on a different processor in this distributed system but the statements required for that web search regardless in which processor it hits may be somewhere on some of the other nodes. Applications like Facebook and YouTube also have state that includes billions of images and videos. In these images and videos are a process to detect certain objects to transform them into different formats and sizes. So there's some computation that's performed on that distributed content. The code that performs these processing operations these transformations trans-coding of the videos or the images may run on different nodes in the system but may potentially need to access state that's distributed on other nodes. These applications are examples of applications where every node is responsible

for managing the locally stored state and providing access to the locally stored state. But also at the same times accessing the state that's stored by the remaining nodes in the system. The reason I'm putting the term peer here in quotation marks and not using a popular term that you may have heard of peer to peer systems is that in these kinds of distributed applications it is still possible that there will be some number of designated nodes that will perform some overall management and configuration tasks for the entire system. So there's some portion of control plane tasks or management plane tasks that are performed by some more centralized designated set of nodes that's not evenly distributed among all nodes. In a peer-to-peer system even the overall control and management of this system would be done by all.

05 - Distributed Shared Memory DSM

Distributed Shared Memory (DSM)

Each node ...

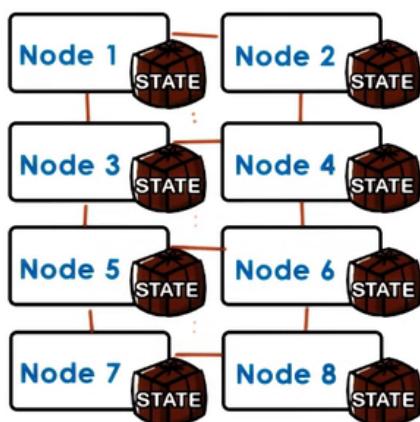


- "owns" state => memory
- provides service
 - memory reads/writes from any node
- consistency protocols

A concrete example of this type of applications is distributed shared memory. Distributed shared memory is a service that manages the memory across multiple nodes so that applications that are running on top will have an illusion that they're running on a shared memory machine like a SMP processor. Each node in the system owns some portion of the physical memory so owns some state and it also provides the operations in that state the reads and writes. They may be originating from any of the other notes in the system. Also each node has to be involved in some consistency protocol so as to make sure that the shared accesses to the state have meaningful semantics. For instance when nodes read and write shared memory locations these reads and writes have to be ordered and observed by the remaining nodes in the system in some meaningful way, ideally in the exact same way that they would have been perceived if this was indeed a shared memory machine. In this lesson using distributed shared memory as an example we will illustrate some of the issues that come up with distributed state management beyond just caching that we already saw in the distributed file system lecture. We will also discuss some meaningful consistency protocols that are useful in these kinds of scenarios. Distributed shared memory mechanisms are important to study because they permit scaling beyond the limitations of how much memory we can include in a single machine. If you

have a multithreaded application or in general an application that was developed with the expectation of shared memory underneath, and all of the sudden you need to support workloads that require more state, you have to add more memory to that system. Now if you look at how the cost of computer systems is affected by the amount of memory they're configured with you will see that beyond a certain limit the cost starts increasing rapidly and machines with very large amounts of memory can be in the order of half million dollar range. Instead with distributed shared memory we can simply add additional nodes and achieve shared memory at a much lower cost. Yes access in remote memory will be slower than access in the local memory. However if you're smart about how data is placed in the first place in the system how it's migrated across the different nodes and what kind of sharing semantics are enforced whenever something gets updated we may hide those access difference from the applications so we may not even perceive there is any kind of slowdown because they're executing and distributed in that environment. One goal of this lecture is to teach you what are some of the opportunities to hide these access differences. Distributed shared memory is becoming more relevant today because commodity interconnect technologies offer really low latency among nodes in a system. For instance these are interconnect technologies that connect the servers in a data center and they offer these RDMA based interfaces where RDMA stands for Remote Direct Memory Access. That provide a really low latency when accessing the remote memories and that really helps address this particular challenge the fact that accessing remote memory is slower. Using these advanced interconnection technologies makes these remote accesses significantly better than what they were before such interconnection opportunities existed. Because of that, distributed shared memory based solutions are becoming more sustainable.

Distributed Shared Memory (DSM)

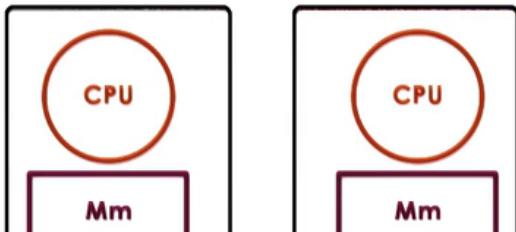


- permits scaling beyond single machine memory limits
- more "shared" memory at lower cost
- slower overall memory access
- commodity interconnect technologies support this (RDMA)

RDMA = Remote Direct Memory Access - provide really low latency when accessing remote memories.

06 - Hardware vs Software DSM

Hardware vs. Software DSM



Hardware-supported (expensive!)

- relies on interconnect
 - OS manages larger physical memory
 - NICs translate remote MM accesses to messages
- ... involved in all aspects of

Distributed shared memory can be supported either in hardware or by software. The key component of the hardware supported distributed shared memories is that they rely on some high end interconnect. The operating system running on each individual node is under the impression that it has much larger physical memory constituting memory that spans multiple memory nodes multiple physical nodes in the system. So the OS is allowed to establish virtual to physical memory mappings that correspond to memory locations that are in other nodes. The memory accesses that reference memory locations that don't correspond to the local physical memory are then passed to the network interconnect card and this is a NIC that corresponds to this advanced high end interconnect technology. So these NICs know how to translate that memory operation into an actual message that gets sent to the correct remote node the correct NIC. The NICs in these nodes will participate in all aspects that are necessary to deal with memory accesses and management of memory consistency and we'll also support certain atomic operations just like the atomics that we saw in shared memory systems. Now while it's very convenient to rely on the hardware to do everything this type of hardware is typically very expensive and it's used only for the ultra high end machines or for the super computing platforms. Instead distributed shared memory systems can be realized in software. Here the software would have to detect which memory accesses are local versus remote to create and send those messages to the appropriate node whenever that's what's necessary. The software would also have to accept messages from other nodes and perform the specific memory operations for them and also be involved in all of the aspects of memory sharing and consistency support. This can be done at the level of the operating system or it can be done with support of a programming language and the runtime product programming language.

07 - Implementing DSM Quiz



Implementing DSM Quiz

According to the paper "Distributed Shared Memory: Concepts and Systems", what is a common task that's implemented in software in hybrid (hardware + software) DSM Implementations?

- prefetch pages
- address translation
- triggering invalidations

Hint: Start reading at Hybrid DSM Implementations heading (page 76 of the paper pdf). Reading the entire paper is highly encouraged =).

Distributed Shared Memory: Concepts and Systems

For a quiz I would like you to take a look at a paper Distributed Shared Memory Concepts and Systems. This was an older survey paper that describes several implementations of distributed shared memory systems and compares them along multiple dimensions that we will discuss in this lesson. The specific question you need to answer is the following. According to the paper Distributed Shared Memory Concepts and Systems what is a common task that's implemented in software in hybrid hardware plus software DSM implementations? The choices are prefetch pages address translation or triggering invalidations. As a hint go straight to page of the original paper PDF that's linked in the instructor's notes and start reading at the hybrid DSM implementations heading. So you don't have to read the entire paper but I do encourage you to do so if you have a chance or at least to look at it in a little bit more detail than just jumping to this hybrid DSM implementation section.

08 - Implementing DSM Quiz



Implementing DSM Quiz

According to the paper "Distributed Shared Memory: Concepts and Systems", what is a common task that's implemented in software in hybrid (hardware + software) DSM Implementations?

well defined and easier to implement in hardware!

prefetch pages

address translation

triggering invalidations

Hint: Start reading at Hybrid DSM Implementations heading (page 76 of the paper pdf). Reading the entire paper is highly encouraged =).

If you read through the hybrid DSM implementation section you will see that it mentions several examples of DSM systems and specifically describes what are the software tasks that those hybrid implementations support. And for every one of those examples prefetching is listed as one of the software tasks. It makes sense that prefetching is a good task to be implemented in software. Whether or not prefetching is useful is going to depend on a particular application on the kind of access pattern that it exhibits. At the same time address translation or triggering invalidations are more concretely defined. And it's easier to implement them with hardware support. For these reasons prefetching pages is the only correct answer to this question.

09 - DSM Design Sharing Granularity

DSM Design: Sharing Granularity

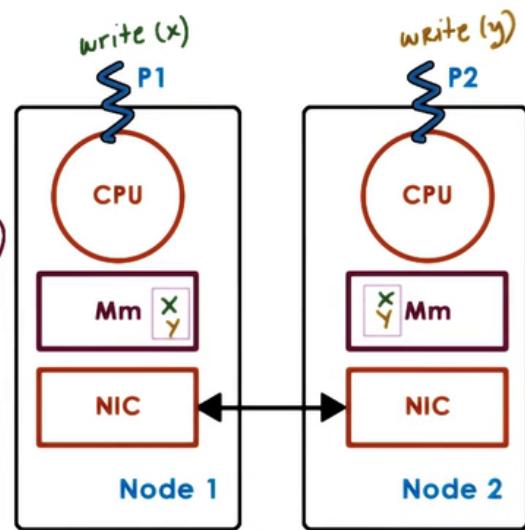
- cache line granularity?
⇒ overheads too high for DSM

✗ variable granularity

✓ page granularity (OS-level)

✓ object granularity (language runtime)

⇒ beware of false sharing
e.g., X and Y are on same page!



Several design points are important to consider when designing a distributed shared memory system. First is the granularity of sharing. In SMP systems the granularity of sharing is a cache line. The hardware tracks concurrent memory accesses at the granularity of a single cache line. And triggers all the necessary coherence actions like invalidations if it detects that a cache line has been modified if that cache line has been shared with other caches. For distributed systems adopting a solution where every single cache line sized write message is being sent to nodes over a network will potentially be too expensive. And it will be hard to justify the use of such system the performance slowdown will be significant and likely it won't be very useful. Instead distributed shared memory designs look at larger granularities of sharing. Some options include variables or pages of virtual memory or entire objects as defined by some higher level programming language. Variables are meaningful from the programmer's perspective so potentially DSM solutions can benefit. Because the programmer can provide some explicit support to tell the distributed shared memory system how and when individual variables should be shared. However this is still potentially too fine granularity. We have a lot of variables that just few bytes long like integers. And in those settings the DSM system would still have very high overheads. Using something larger like an entire page of content or a larger object that begins to make more sense. If the distributed shared memory system is to be integrated at the operating system level the operating system doesn't understand objects. The only thing that it understands is pages of virtual memory. And then the OS tries to map those pages of virtual memory to the underlying physical page frames. So at the operating system level it makes sense to integrate some page-based DSM solutions. The OS would track when pages are modified and then it would trigger all of the necessary messages that need to be exchanged with remote nodes on page modification. Pages are larger. We set a common page sizes for kilobytes in many environments. And so it is possible to then amortize the cost of the remote

access for these larger granularities. With some help from the compiler application level objects can be laid out on different pages and then we can fully just rely on the page base operating system level mechanism. Or we can have a distributed shared memory solution that's actually supported by the programming language and the runtime where the runtime understands which objects are local versus remote objects. And for those objects that are remote ones the runtime will generate all of the necessary communications with remote nodes and all the necessary operations to provide distributed shared memory. In that case the operating system doesn't really need to know anything about the DSM solution. So the benefit of that is that the operating system doesn't have to be modified in this case but this is clearly going to be a less general solution. It will be applicable only for those languages for which there is such a DSM support. Once we start increasing the granularity of sharing one important problem that everyone has to be aware of is what's called false sharing. Consider a page or even a higher level object that internally has two variables x and y. A process in one node is exclusively accessing and modifying x. It doesn't completely care about anything that's stored in the variable y. Similarly a process on another node is exclusively concerned with y. And it has no reference whatsoever to the other variable x. Now when x and y are shared on the same page as in this example here the distributed shared memory system when it's using these larger granularities that's the only thing that it understands. So it understands a shared page. So it will interpret these two write accesses to that shared page as some indication of concurrent accesses to the same location. And it will trigger any necessary coherence operations invalidations updates or any of the other overheads that are associated with maintaining consistency among these two copies. Such coherence overheads won't benefit anyone. P doesn't care what happened to y and also P doesn't care what happened to x. In order to avoid these kinds of situations the programmer must be careful how data is allocated and laid out on pages or how it's grouped in higher level objects. Or the other alternative is to rely on some smart compiler that will be able to understand what is really shared state. And then allocate it within a page or within an object versus what is something that will trigger these false sharing situations.

10 - DSM Design Access Algorithm

What Types of Applications Use DSM?

Application access algorithm

- single reader / single writer (SRSW)
- multiple readers / single writer (MRSW)
- multiple readers / multiple writers (MRMW)



Another important design point in distributed shared memory systems is to understand what are the types of access algorithms that need to be supported on top of that DSM solution? In other words to understand what are the kinds of applications that will be running on top of the DSM layer. The simplest example is single reader/single writer. For these kinds of applications the main role of the DSM layer is to provide additional memory to provide the application with the ability to access remote memory. In this case there really aren't any consistency or sharing related challenges that need to be supported at the DSM layer. The more complex examples are of an application support either multiple readers and single writer or both multiple readers and multiple writers. In those cases it's not just about how to read or write to the correct physical memory location in the distributed system but it's also about how to make sure that the reads return the correctly written the most recently written value of a particular memory location and also that all of the writes that are performed are correctly ordered. This is necessary so as to present the consistent view of the distributed state of the distributed shared memory to all of the nodes in the system. Multiple reader single writer is a special simpler case of the multiple reader multiple writer problem. And so in our discussion in this lesson we will focus on DSM support for multiple readers multiple writers.

11 - DSM Design Migration vs Replication



Performance Considerations

DSM performance metric == access latency

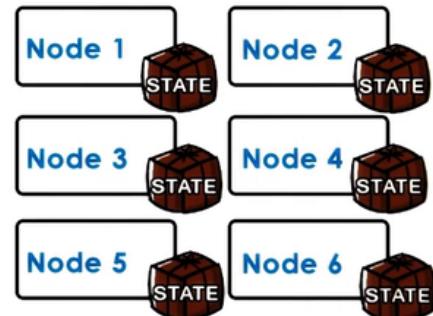
Achieving low latency through...

Migration

- makes sense for SRSW
- requires data movement

Replication (caching)

- more general
- requires consistency management



For a distributed shared memory solution to be useful it must provide good performance to applications. If we think about the core service that's provided by distributed shared memory systems accessing memory locations then it's obvious that the performance metric that's relevant for DSM systems is what is the latency with which processes running on any one of these nodes can perform such remote memory accesses. Clearly accessing local memory is faster than remote memory. So what can we do in order to maximize the number of cases where local memory's accessed versus remote? One way to maximize the number of local accesses and achieve low latency is to use a technique called migration. Whenever a process on another node needs to access remote state we literally copy that state over to the other node. This makes sense for situations where we have a single reader single writer. Since only one node at a time will be accessing this state so it does make sense to move the state over to where that single entity is. However this requires moving the data copying the data over to the remote node and that incurs some overheads. So even for these single reader single writer cases we should be careful when we trigger these types of mechanisms because if it's only going to be a single access that will be performed in this other location then migrating copying over the entire state over to node four it won't be amortized. We won't get much in terms of low latency improvements if we have to copy all this data just for a single read or write access. For the more general case however when there are multiple readers and multiple writers migrating the state

all over the place doesn't make any sense since it needs to be accessed by multiple nodes at the same time. So a mechanism such as replication where the state is copied on multiple nodes potentially on all nodes is a more general mechanism. Use of caching techniques which create a copy of the state on each node where the state was accessed can lead to some similar behavior as what's seen with replication. One problem with this is that it requires consistency management. Now this state will be accessed concurrently on multiple nodes. And we have to make sure we coordinate those operations as we said to order all of the writes propagate the most recent write operation to wherever or whomever is performing the next read operation. This is some overhead. One way to control the overhead is to perhaps limit the number of replicas the number of copies that can exist in this system at any given point of time since the consistency management has overhead that's proportional with the number of copies that need to be maintained consistently.

12 - DSM Performance Quiz



DSM Performance Quiz

If access latency (performance) is a primary concern, which of the the following techniques would be best to use in your DSM design? Check all that apply.

- migration
- caching
- replication



Let's take a quiz about DSM performance. The question is if access latency as a performance metric is a primary concern which of the following techniques would be best suited in your DSM design? The choices are migration caching or replication. And you should check all that apply.

13 - DSM Performance Quiz

Before answering we should consider the access algorithm like single reader single writer multiple reader multiple writer. If we only have single reader single writer then migration is okay but it's not good in all of the other cases. In a more general problem that has multiple readers and multiple writers with migration pages would have to be flipped back and forth between nodes and so migration is really not a good technique. Migration can in fact lead to an increase in latencies for the more general problem. If we look at the other two options caching brings the data on the node where it's accessed and therefore it will definitely improve the latency of the subsequent operations to that data. And similarly replication in general will create copies of the data that are potentially closer to where the data is accessed and therefore can lead to improvements in latency. Now whenever there are multiple concurrent writes in the system caching and replication can also lead to high overheads. If you remember in the distributed file system lecture we mentioned that in the sprite file system whenever it was detected that there are multiple concurrent writers caching or in general the presence of multiple copies of the particular state a file in that case was disabled so as not to have to deal with multiple invalidations or loss of consistency.

14 - DMS Design Consistency Management

P3L4 Synchronization Constructs, Morsel on Cache Coherence

DSM Design: Consistency management

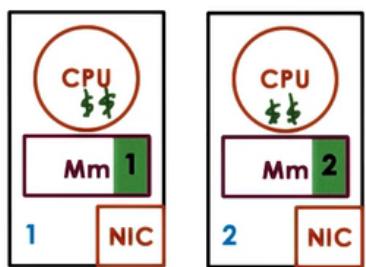
Push invalidations when
data is written to ...

Pull modification info
periodically ...
↳ ...)

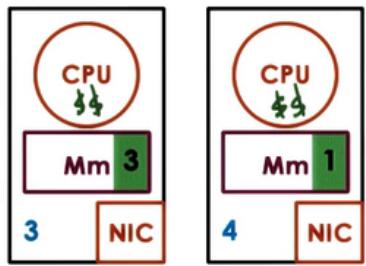
Once we've permitted multiple copies of the same data page or object to be stored in multiple locations the question of maintaining consistency comes up. Since distributed shared memory is intended to behave in a similar manner to shared memory in shared memory multiprocessors let's remind ourselves what we did in shared memory multiprocessors for consistency management. In the lesson one synchronization we explained that in shared memory, multiprocessors consistency is managed using two mechanisms, a write-invalidate or write-update. With write-invalidate, whenever the content particular memory location that's cached to multiple caches is changes on one CPU in one cache. Back via the coherence mechanisms will be propagated to other caches and in the case of write invalidate the other

caches will invalidate their cache content. Or in the event that we have a write update coherence mechanism then the other caches will receive the newly updated copy of that particular memory location. These necessary coherence operations are triggered by the shared memory support in the hardware on every single write operation. And the overhead of supporting that in the distributed shared memory system where the latencies and the costs of performing distributed communication are too high is not going to be justifiable. For these reasons for distributed shared memory we'll look at coherence operations that are more similar to what we discussed in the distributed file systems lecture. One option is to push invalidation messages when a data item is written to. This is similar to the server-based approach that we talked about in the distributed files systems lesson. But remember that the state management in DSM systems is done by all peers. We don't have clients and servers per se in this case. The other option is for the nodes to pull information about any modified state from one or more of the other nodes in the system. This can be done either periodically or purely on demand whenever some process needs to access that state locally it will check with others to see whether it's been modified or not. I intentionally chose the terms push and pull since these are commonly used to distinguish between this more proactive versus this more reactive approach to accomplishing some tasks. In this case maintaining the consistency among two nodes in the distributed system. Another set of terms associated with these types of actions is eager versus lazy. The push-based method is eager since it forces propagation of information immediately eagerly. in contrast the pull method is lazy since it lazily gets the information when it's convenient or when it becomes absolutely necessary. And yet another set of terms to distinguish between these two types of approaches is pessimistic versus optimistic. This push based eager method is pessimistic in that it expects the worst. They expect that the modified state will be needed at other places at other nodes immediately. And so with these methods nodes are in a rush to notify others that a modification happened. In contrast these optimistic methods hope for the best. Here the hope is that the modified state wouldn't be needed elsewhere anytime soon. And that there is plenty of opportunity to accumulate information regarding modifications before anyone has to pay for the cost of sending an invalidation or moving data across the distributed system. Regardless of whether we talk about the push versus pull based methods. When exactly they get triggered whether its after every single data has been modified or whether its with a period of five seconds or ten seconds or one millisecond. Or in some other manner that really is going to depend on the consistency model for the shared state and we will discuss what are the options for consistency models a little bit later in this lecture.

15 - DSM Architecture



DSM Architecture (page-based, OS-supported)
=> each node contributes part of Mm pages to DSM



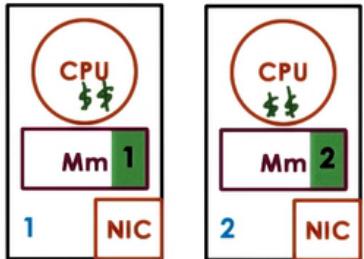
if HMRW...

- need local caches for performance (latency)
- home (or manager) node drives coherence ops.
- all nodes responsible for part of distributed memory (state) management

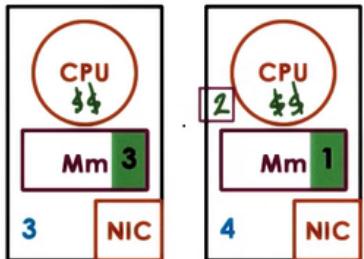
Based on what we described so far let's take a look at how a distributed shared memory system can be designed. This type of system consists of a number of nodes each with their own physical memory. Everyone of the nodes may contribute towards the distributed shared memory system only a portion of their physical memory or it can contribute all of it. Let's assume here that only a portion of the physical memory is contributed towards the DSM service and can

explicitly be addressed, whereas the rest of the memory is used either for caches or for replication or for some metadata that's needed for the DSM layer. The pool of memory regions the memory pages that every single one of these nodes contributes forms the global shared memory that's available for applications running in the system. Every address in this memory pool will be uniquely identified based on the identifier for the node where it's residing as well of the page frame number of that particular physical memory location. The node where our page is located is typically referred to as the home node of that page. Now let's say we're trying to solve the most general case where the system is supposed to support applications that have multiple reader multiple writer requirements. For that reason in order for the system to deliver acceptable performance and achieve low latency with which the memory accesses are performed the DSM layer will incorporate use of caching. Pages will be cached on the nodes where they are accessed and for a state that's cached for memory pages that are stored on these remote nodes. The home node or the manager node will be responsible for driving all of the coherence related operations so it will maintain state that's necessary to track the number of readers the writers whatever cache has been modified et cetera. In this way all of the nodes in the system are responsible for some portion of the management operations for that distributed memory. The information that's maintained at the home or the manager node is similar to the kind of information that we saw that the servers have to maintain in the distributed file system example. Except that in this situation every single one of the nodes is responsible for both providing the service the acting as a server for that particular portion of the shared memory, and also being the client with respect to the other nodes. So every one of the nodes will participate in interactions with a home node in case they are locally caching some of the memory that's stored at a remote site. The home node will have to keep track of states such as what are the pages accessed who is it accessed by whether it's been modified. We may also choose to incorporate mechanisms to dynamically enable/disable caching for. For a similar motivation like what we had in the sprite file system. What are the pages locked that's another useful piece of information. All this information is used in enforcing the shearing semantics that this particular DSM system will implement. One particular page is repeatedly and even exclusively accessed on a node that's not its home node. It would be too expensive to repeatedly contact the home node to perform any necessary state updates so one mechanism that's useful in DSM systems is to separate the notion of home node from the so-called owner. The owner is the node that currently owns the page that's like the exclusive writer for instance. And that's the node that can control all of these state updates and can drive any consistency related operations. So this owner may be different from the home node and in fact the owner may change as whoever is accessing this page migrates throughout the system or new processes new threads require access to this particular page. They may become owners as well. The role of the home node for that page in this case that was node two is to keep track of who is the current owner of that page. In addition to creating page copies via caching in an on demand manner page replicas can be explicitly created for reasons such as load balancing hot spot avoidance or reliability reasons so that the page contents do not disappear if some node in the system fails. For instance in data center environment that have lots of machines where a certain distributed shared state is managed it makes sense to triplicate such shared state on the original machine on a nearby machine for instance in the same rack and then on another remote machine

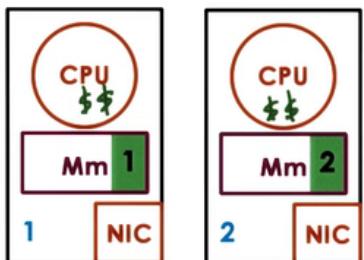
whether it's in another rack or even potentially in another data center. The consistency of these replicas is controlled either by the home node or by some designated management node.



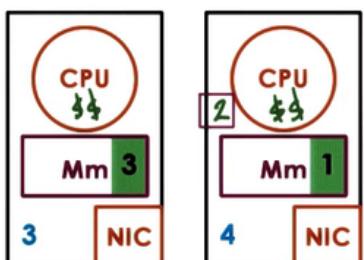
DSM Architecture (page-based, OS-supported)
 => each node contributes part of Mm pages to DSM
 => need local caches for performance (latency)
 => all nodes responsible for part of distributed memory



"Home" node
 - keeps state: pages accessed, modifications, caching enabled/disabled, locked ...
 - current owner (owner may not equal home node)

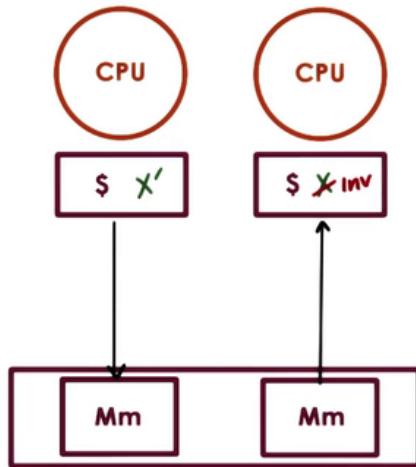


DSM Architecture (page-based, OS-supported)
 => each node contributes part of Mm pages to DSM
 => need local caches for performance (latency)
 => all nodes responsible for part of distributed memory
 => home node manages accesses and tracks page ownership



Explicit replicas
 - for load balancing, performance, or reliability
 - "home"/manager node controls management

DSM Design: Consistency Management



DSM ~ shared memory in SMPs

In SMP

- write-invalidate
- write-update



16

- Summarizing DSM Architecture

Summarizing DSM Architecture

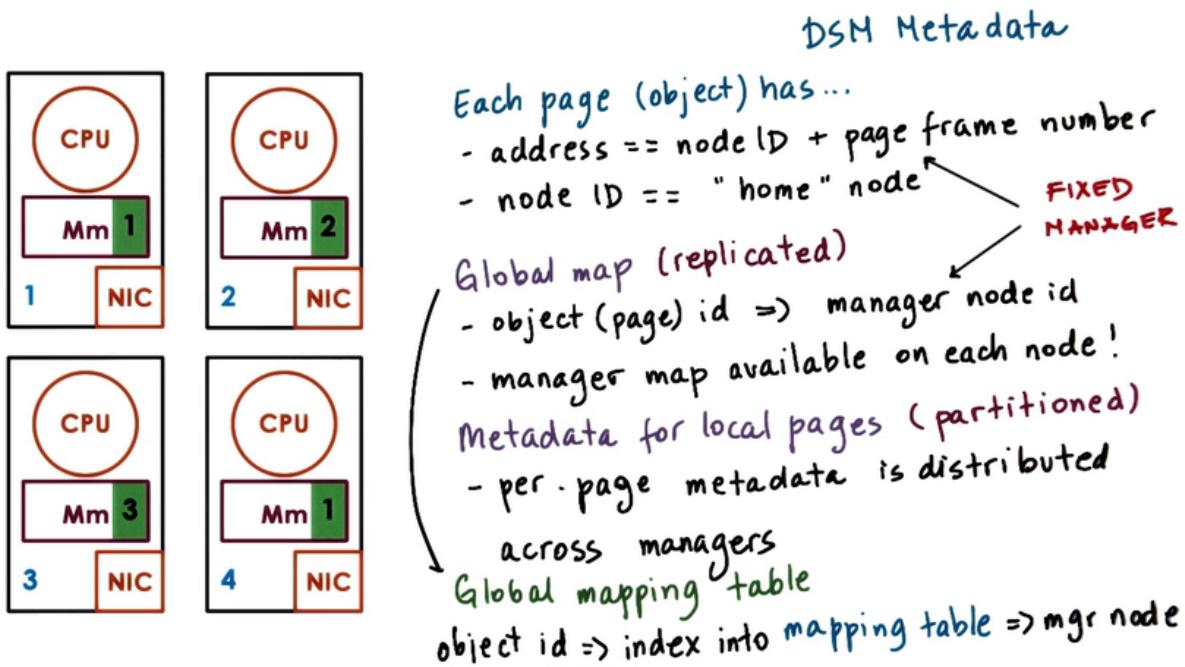
Page-based DSM :

- => each node contributes part of Mm pages to DSM
- => need local caches for performance (latency)
- => all nodes responsible for part of distributed memory
- => home node manages accesses and tracks page ownership
- => explicit replication possible for load balancing, performance or reliability



In summary here's some of the key architectural features of distributed shared memory systems. Specifically we're talking about page based distributed shared memory systems. Every node contributes part of its main memory pages to the DSM layer. Local caches are maintained to improve performance by minimizing the latency of the access to store node modes. All nodes are responsible for some portion of the distributed memory and of its management. Specifically the home node for a page manages the accesses to that page and also tracks who currently owns that page. Who has ownership rights. And finally some explicit replication could be considered in order to assist with load balancing. In order to further improve performance and to address certain reliability concerns.

17 - Indexing Distributed State



One important aspect of distributed shared memory systems is how do we determine where a particular page is? In order to do this the DSM component has to maintain some metadata. First let's see how is a page identified in this system in the first place. Well the page has its address. And that may be some notion of the node identifier plus the page frame number locally at that node. And we said that the node identifier also identifies the home node of the system.

Considering this address format then we see that the address itself directly identifies what is the manager node the home node. That knows everything about that particular page. Every single node in the system just by looking at the address of a particular page. Can uniquely determine what is the manager node for that page. So it's as if the manager information is available on every single node in the system. This could be captured via a global map that has to be replicated that will tell us how for a particular object we find the manager. What that means is that the information about the manager is available on every single node. So it's as if the information about the manager is available on every single node. Since whenever any of the nodes in the system wants to access a particular address they just need to look at the node identifier. And we'll be able to get to the specific manager the specific home node for that address. This information that captures the translation from an identifier of an object a page in this case. To the management node of that object. Represents a global map. And this needs to be replicated. It needs to exist. This information needs to be available on every single one of the nodes. Once we get to a particular manager that's the entity that will have the per-page or per-object metadata. That's necessary in order to perform this specific access to that page or to enforce its consistency. What this means is that in the DSM system the metadata for individual objects or individual pages. It's partitioned across all of the home nodes. All of the management nodes in the system. But in order to quickly find where the manager for a particular data item is.

It is necessary to maintain an additional data structure a global map that will be replicated on every single one of the nodes. That will in some way take an object identifier and map it to that manager. One final note in the example that we discussed so far. We somehow implied that certain bits from the address are used to identify the node identifier. And that means that for every single page there will be a fixed manager uniquely identified from that page identifier. If we want some additional flexibility we can take that object identifier and then use it as an index into a mapping table. This mapping table is the one that will be used at every single node for the global map. And every entry in that mapping table will actually encode a manager node. So for a particular object or a particular page identifier we will first index into that mapping table. Using some bytes from the object id or some hash function. That's applied on top of this object id. And then the entry at the particular location at the mapping table will tell us what the manager node for that page is. What's nice about this approach is. If for any reason we want to change who is the manager node for a particular object or of a particular page. Whether the original manager field or whether we need to rebalance the system or any other reason. We just need to update the mapping table. There is no need to consider making any kind of changes for the object identifier. The object can remain identified in the exact same way as it was before we chose to make a change into the manager nodes.

18 - Implementing DSM

Now that we described the possible DSM architecture I'd like to comment on its possible implementation. One thing to consider when implementing a distributed shared memory system is that the distributed shared memory layer has to intercept every single access to that shared state. This is needed in order to detect whether the access is local or remote and then trigger

remote messages requesting access if that's necessary. Or to detect that somebody is performing an update operation to a locally controlled portion of the distributed shared memory and then to trigger any necessary coherence messages. These overheads should be avoided whenever we're trying to just access local non-shared pages or non-shared state. So what we would like to achieve is an implementation where it is possible to dynamically engage whether the distributed shared memory layer will be triggered and will be intercepting any accesses to memory in order to determine what to do about them. Or disengage the distributed shared memory layer if we are performing access to pages which are really not shared and are just local pages accessed on a particular node. To achieve this a DSM implementation can leverage the hardware support that's available at the memory management unit level. As we explained earlier if the hardware MMU doesn't find the valid mapping for a particular virtual address in the page table it will trap into the operating system. And similarly the hardware will also cause a trap if it's detected that there is an attempt to modify a page that has been protected for a write operation so a write protected page. We can leverage this mechanism to implement the DSM system. Whenever we need to perform an access to a remote memory there will not be a valid mapping from the local virtual address to the remote physical address. The hardware will generate a trap in that case. And at that point the operating system will detect what is the reason for that trap will pass that page information to the DSM layer and the DSM layer will send the message. Similarly whenever content is cached in a particular node the DSM layer will ensure that that content is write protected. And that will cause a trap if anybody tries to modify that content that will turn control over to the operating system. The operating system can pass relevant information to the DSM module. And then that one will trigger all of the necessary coherence operations. When implementing a DSM system it is also useful perhaps to leverage additional information that's also maintained by the existing memory management that the operating system and the underlying hardware provide. For instance we can track information like whether our page is dirty. We can track information whether our page has been accessed in the first place. And this can let us implement different coherence mechanisms and consistency policies. For an object-based distributed shared memory system that's implemented at the level of the programming language runtime the implementation can consider similar types of mechanisms that leverage the underlying operating systems services or as an alternative everything can be done completely in software with support from the compilers so tracking whatever particular object reference is remote or local. Or whether an object is going to be modified or not we can make sure that we generate code that will perform those checks on every single object reference.

Implementing DSMs

Solution: Use Hardware MMU support!

- trap in OS if mapping invalid or access not permitted
- remote address mapping \Rightarrow trap and pass to DSM to send msg
- cached content \Rightarrow trap and pass to DSM to perform necessary coherence ops.
- other MMU information useful (e.g., dirty page)



19 - What is a Consistency Model

What is a Consistency Model?

Consistency model == agreement between memory (state) and upper software layers

"mem behaves correctly if and only if software follows specific rules"

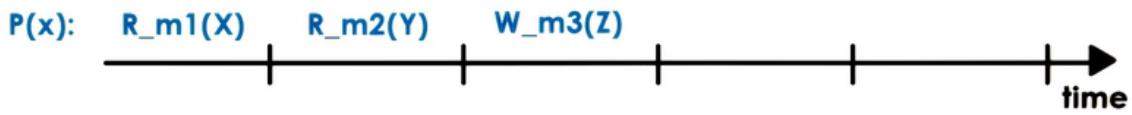
memory (state) guarantees to behave correctly ...

- access ordering
- propagation / visibility of updates

We said multiple times throughout this lecture that the exact details of how a distributed shared memory system should be designed or how the coherence mechanisms will be triggered depends on the exact Consistency Model. But before describing several Consistency Models let's first explain what is a Consistency Model. Consistency models exist in the context of the implementations of applications or services that manage distributed state. The consistency model is a guarantee that the state changes will behave in a certain way as long as the upper software layers follow a certain set of rules. For the memory to behave correctly what that means is that we're making some guarantees how our memory access is going to be ordered. For the memory to behave correctly what that means is that the way that the operations will access memory will somehow be representative to how those operations were issued in the first place. And that we will be able to make some guarantees that whenever somebody is trying to read the memory location that the value that they will see will be a value that's representative of what was written to that location recently. Now what it means is that a consistency model guarantees that the memory will behave correctly that the accesses will be correctly interleaved and the updates will be correctly propagated only if the software follows certain rules. That implies that the software needs to either use specific APIs when it performs an update or when it requests to perform an access or that the software needs to set certain expectations based on what this memory guarantee is. For instance that I'm just not going to enforce that updates to the particular memory location are written in the exact same order as they were issued. If the software knows this particular information then it is up to the programmer to use some additional operations such as locking constructs or similar in order to get the desired behavior. This is not something that's specific to a distributed system. Even in a single CPU when we have multiple threads that are trying to access a particular memory location we know that there's no guarantee how those thread accesses will be ordered. And which particular update will be returned when let's say thread end tries to read a particular shared variable. If we want to achieve some guarantees that are stronger than that the software will have to use locks would

have to use atomic operations would have to use some counters. So a consistency model it specifies how the memory will behave and the upper layers of the software must understand that and set their expectations accordingly. At the same time the memory layer may export certain API's certain operations like the locks that we mentioned. And then if the software uses those API's correctly then perhaps the memory system can make even stronger guarantees. In the discussion of consistency models we will use timeline diagrams which will look like this. Which will show when certain operations occur according to real time based on some neutral external observer that sees everything instantaneously the minute those operations occur. Our notation for this is as follows $R_{m1}(x)$ means that this value x was read from a particular memory location $m1$. What this means here is that at this particular point of time the value x was read from memory location $m1$. And then at this later point in time the value y was read from memory location $m2$. Similarly $W_{m1}(y)$ means that the value y was written to in this case memory location $m1$. So at this particular point in time the value z was written to a memory location $m3$. We will also assume that initially at the start of this timeline diagrams all of the memory set to zero

Our Notation:



Notation:

- $R_{m1}(x)$ == x was read from memory location $m1$
- $W_{m1}(y)$ == y was written to memory location $m1$
- initially all memory set to zero

20 - Strict Consistency

Theoretically for a perfect consistency model we would like to achieve absolute ordering and immediate visibility of any state update and axis and we also want this to be in the exact same manner as those updates were performed in some real time. With this model changes are instantaneous and immediately visible everywhere so even if we had some read operations that were immediately performed over here to the locations m and m they would have still returned these values x and y . So in P regardless how far away from P it is P would always be able to instantaneously see that P performed this write operation to the memory location m and that it wrote x there. Furthermore what's also really important about the strict consistency model is that it guarantees that every single node in the system will see all of the writes that have happened in the system in the exact same order. So if we have the situation where P is maybe closer to P than P and these time intervals are really really small in reality it's possible that it took longer for this message from P to get to P so that P can see this x value. That's not allowed with strict consistency. Strict consistency every single update has to be immediately visible and everywhere visible and the ordering of these updates needs to be preserved. In practice even in a single shared memory processor there are no guarantees regarding the ordering of memory access operations from different cores unless we use some additional locking and synchronization primitives. In distributed systems the additional latency any possibility for the messages to be lost or reordered make this not just harder but also even impossible to guarantee. For that reason strict consistency remains a nice theoretical model but in practice it is not something that's sustainable and other consistency models are used instead.

21 - Sequential Consistency

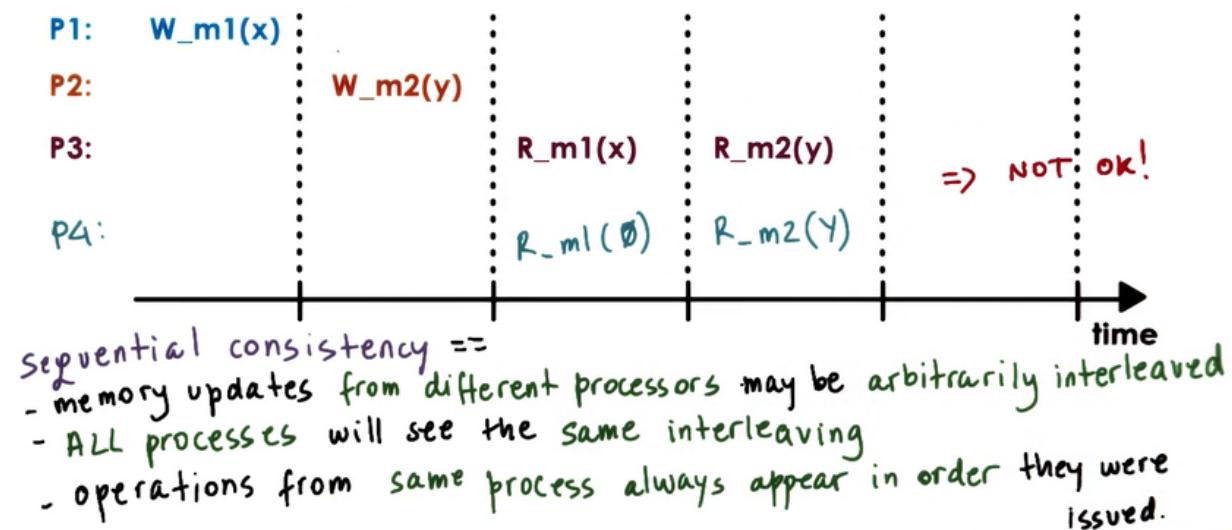
Sequential Consistency



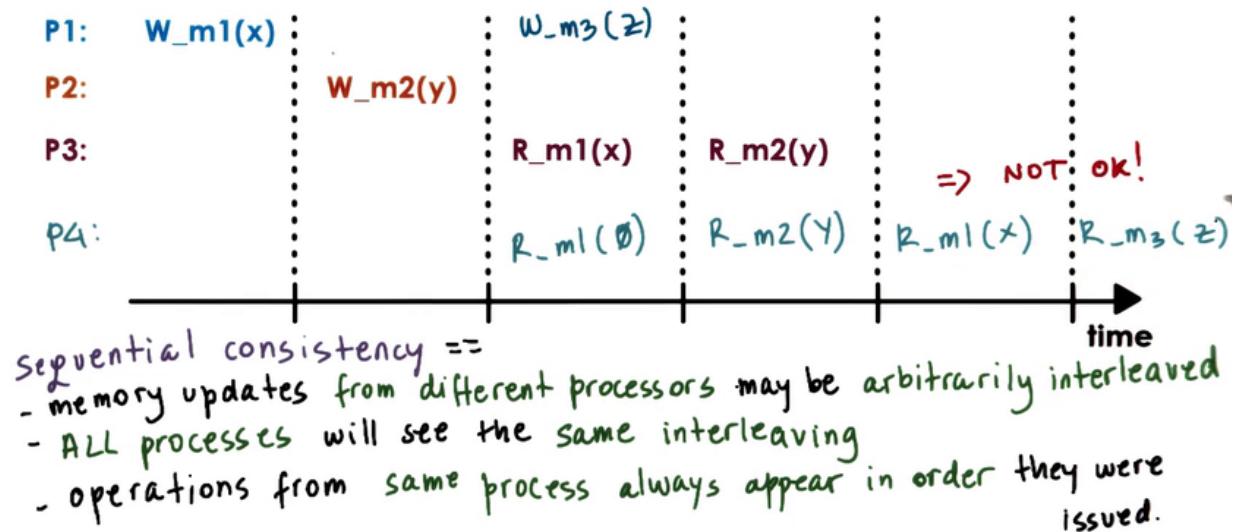
Given the strict consistency is next to impossible to achieve the next best option with reasonable cost is sequential consistency. If you look at the example from the previous idiom if x and y are completely unrelated,. And P computed the value of y independently without any reference to the memory location m1. Then these two operations don't really have to strictly appear in this particular order. For instance it is possible for P to observe just like what we did in the previous case that the value of m has become x. And then to see that the value of m has become y. But it is also perfectly reasonable for P at a particular point in time to access the value of m and to see that it's still zero. So it doesn't know that this right operation happened. However at a subsequent moment when it tries to read the memory location m it sees that it has already been updated to y. This is the equivalent of what would happen in a regular shared memory system if we have multiple threads. And if we don't use any additional controls and your walking sequentializations and anything like that. When these two operations are issued they will be applied in some way some order. But we don't necessarily have control which is the order that these operations will be applied in. So that's why we say that as long as the ordering that's observed is equivalent to some possible ordering of these operations. If they executed on a single shared memory system then that would be considered a legal operation. According to this sequential consistency model. So in summary according to sequential consistency the memory updates from different processors may be arbitrarily interleaved. However if we let one process see one ordering of these updates. We have to make sure that all other processes see the exact same ordering of those updates. For instance it is not okay to let process P see that the update to m with the value x. Appear before the update of m with the value y. And instead at the same time to allow process to think that the value of m was still zero. When the value of m was already updated to y. In this case process P thinks that m has been updated before the memory location m1. It would be too hard to reason about what exactly will happen in a program execution. If every processor can see completely different ordering of some updates,. And if we the software or the programmer doesn't have an understand how and when are those updates

reorder. What this means is that the sequential consistency every single process will see the exact same interleaving. This interleaving may not correspond to the real way in which these operations were ordered. However sequential consistency at least guarantees that every single process in the system. Will see the exact same sequential ordering of all of this. And one constraint of the interleaving is that the updates that are made by the same process will not be arbitrarily interleaved. For instance if P makes another update in the future. Then on any of the other processes it will not be possible to first observe that the value z was written to the memory location m3. And only then find out that the value x was written to memory location m1. So for instance what we see on P4. This will be correct under sequential consistency because the updates to m and m are observed here in the correct way. When it sees that the value of m was z it already knew that the value of m had become x.

Sequential Consistency



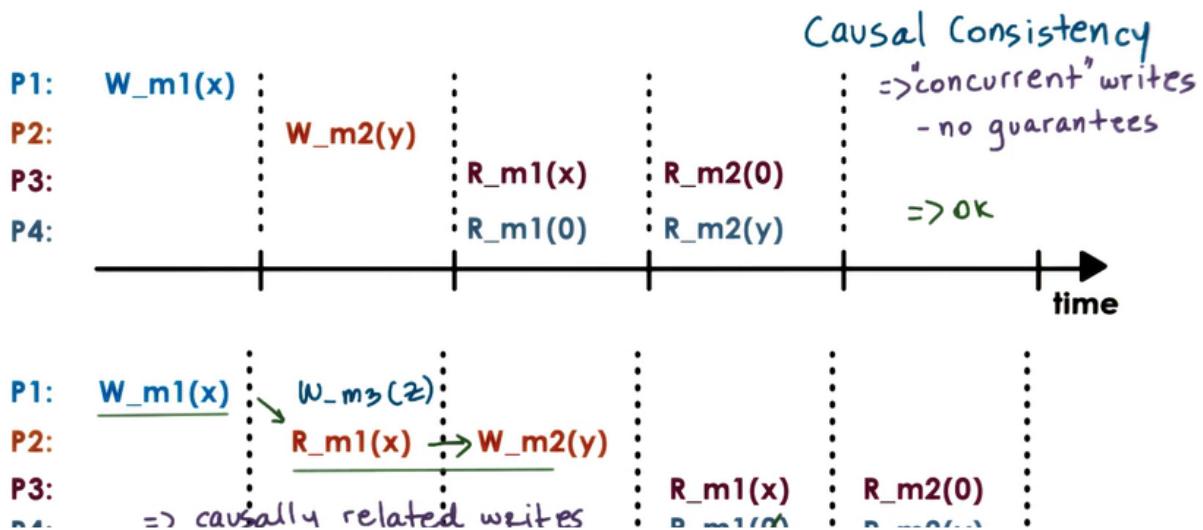
Sequential Consistency



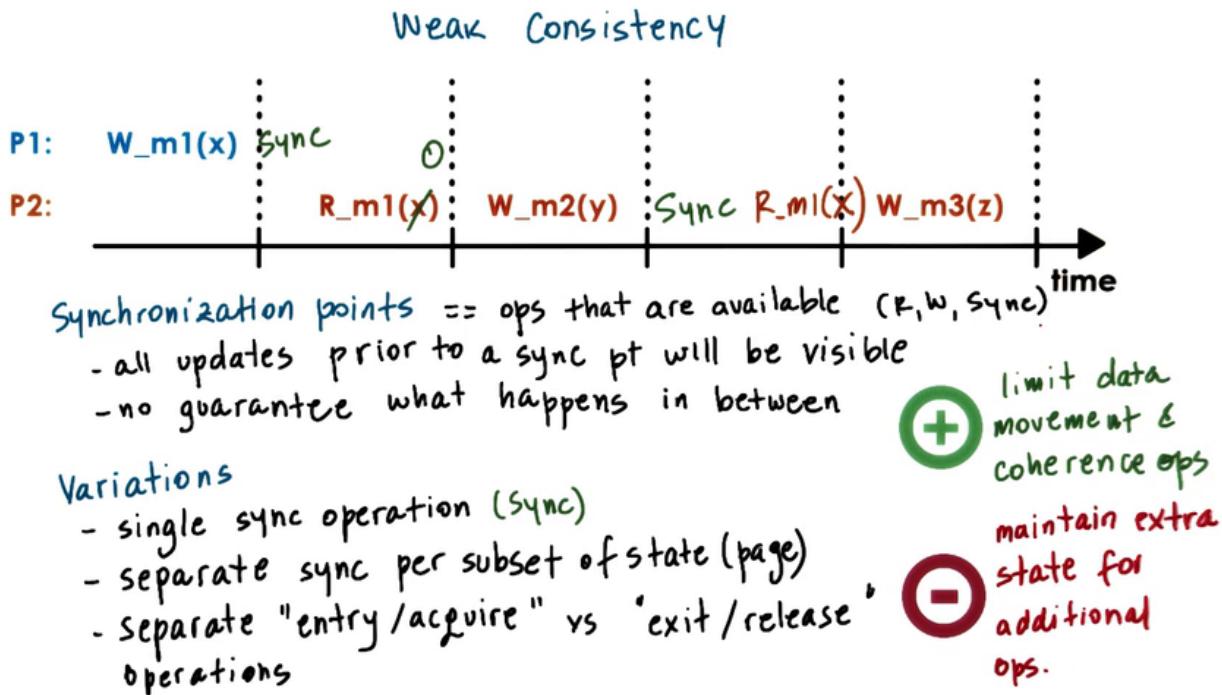
22 - Causal Consistency

Forcing all processes to see the exact same order on all updates may be an overkill. For instance P and P here may perform completely different in independent computations with the values m and m2. And furthermore the update to m had nothing to do with the value of m they were also completely independent. However if we take a look at the second example before the value m was written the value of the memory location m was read. This means that there is potentially some relationship between the memory location m this particular update to the memory location m and this update to the memory location m1. Given that this read operation that was performed at P just before this write. Return the value of this write operation that happened on processor one. There is clearly some flow of information some potential dependence between these two updates that happened on two different processors. Based on this observation clearly it is not okay for these two operations this m update to y and the update of memory location m to x to be perceived in this different order. It would be very incorrect for somebody like P in this case to observe that the value of m was updated to y before the value of m was updated to x. P here thinks that m is still zero and then it finds out that m became y. But there is dependence between the fact that y was written to m and the fact that x was already written to m1. So clearly this is not a reasonable execution. It will be very difficult for software to understand what is exactly happening with the memory if it cannot reason about dependencies like this. However in this case with such dependence did not exist. It will probably be okay for us to tolerate this kind of reordering. The consistency model that provides exactly this kind of behavior is called causal consistency. Causal consistency models guarantee that they will detect the possible causal relationship between updates. And if updates are causally related then the

memory will guarantee that those rights those update operations will be correctly ordered. What that would mean is that in this situation where the two updates are in fact correctly ordered we have to make sure that every processor observes that the update of x to the memory location m happened before the update of y to the memory location m_2 . So we have to make sure that on P this first read of the value of the memory location m returned an x . In this case this execution will be causally consistent given this particular relationship between the updates. For writes that are not causally related or they're referred to in causal consistency as concurrent writes there are no guarantees. Causal consistency doesn't guarantee anything about such updates and they may perfectly legally appear in arbitrary orders on different processors. Just like before causal consistency also makes sure that the writes that are performed on the same processor will be visible in the exact same order on other processors. Again it would be too much of a challenge to understand how to really reason about a system if the updates from a single processor can be arbitrarily reordered.



23 - Weak Consistency



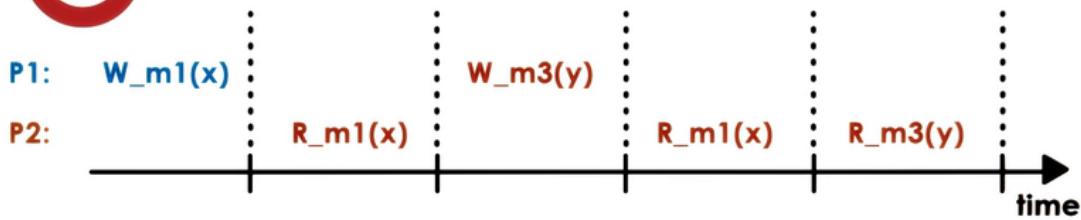
In the consistency models we discussed so far the memory was accessed via read and write operations only. In the weak consistency models it's possible to have something beyond just read and write operations when memory's accessed. In the models we've described so far the underlying memory managers or state managers in general had to determine or infer what are important updates and then how to order and propagate the information regarding those updates. For instance there's no way for the distributed memory managers to understand whether or not indeed the value of y was somehow computed based on this value x that was read from m_1 . So it's perfectly possible that p really just completely randomly came up with this value y that was written in m and that had nothing to do with this fact that it previously read the value of x from location m_1 . So it saw this particular update. However with the causal consistency model it will be forced that every process every processor in the system has to observe that the value of the memory location m has been updated to x before the value of the memory location was updated to y . However for instance in the same process p maybe at a later time there is another update that's happening to memory location m and the value z is written there. And perhaps this particular value z was directly computed based on the value that was read from memory location m_1 . So there maybe is a dependence between this particular write and the value that was read here but not between these two. How can a memory system distinguish these? To do this the memory system introduces synchronization points. The synchronization points are operations that the underlying memory system makes available to the upper layers of the software. So in addition to being able to read and write a memory location you'll also be able to tell the distributed shared memory sync. What a synchronization point does it makes sure that all of the updates that have happened prior to sync the synchronization point will become visible at other processors. And also synchronization point makes sure that all of the updates that have happened on other processors will become visible subsequently at this particular processor in

the future. If p performs a synchronization operation at this particular point after this write was performed that doesn't guarantee that p will see that particular update at this moment. Although this read operation is happening according to some real time some external observer at a later point in time than the synchronization point in the preceding write p has not made an attempt to synchronize with the distributed shared memory. So p has no guarantees that it will see the updates performed by others. So this synchronization point it has to be called both by the process that's performing the updates and also by the processes that want to see and want to see the guarantee that they will see updates. However once the synchronization operation is performed in p afterwards p will be guaranteed that it will see all of the previous updates that have happened to any memory location in the system. So if we perform a read operation of the memory location m at this point in time after the synchronization we're guaranteed that p will see this update. In this case in this model we use a single synchronization operation for all of the variables in the system for the entire shared memory. It is possible to have solutions where different synchronization operations are associated with some subset of the state. For instance with a granularity of individual pages. It is also possible to separate the steps when a particular process requires that all of the updates performed by other processors are visible to it. We call that the entry point or the point when we are acquiring the updates made by others. For instance at this particular synchronization point we can perform an entry operation and synchronize with all the updates that were performed on others in this case on p_1 . And then we can have a separate synchronization primitive that's the exit point or this is the point where we release to all other processes or processors the fact that we have performed certain updates. So for instance at this synchronization point only the updates made by p will be forced to other processes. The idea with these finer grained operations or these mechanisms to really directly control what kinds of messages will be sent by the underlying state management system by the DSM layer at a particular point of time is to make sure that the system controls the amount of overheads that are imposed by the DSM layer. The idea is to limit the required data movement the required messages and coherence separations that will be exchanged among the nodes in the system. But the downside of this is that the distributed shared memory layer will have to maintain some additional state to keep track of exactly what are all the different operations that it needs to support and how it needs to behave when it sees a particular type of request. In summary all of these models that in addition to just pure read and write operations introduce some types of synchronization primitives are referred to as weak consistency models and they allow us to control the overheads of the system from one aspect. However they introduce some additional overheads that are necessary in order to support that the added operations.

24 - Consistency Models Quiz 1



Consistency Model Quiz 1



Consider the following sequence of operations. Is this execution sequentially consistent?

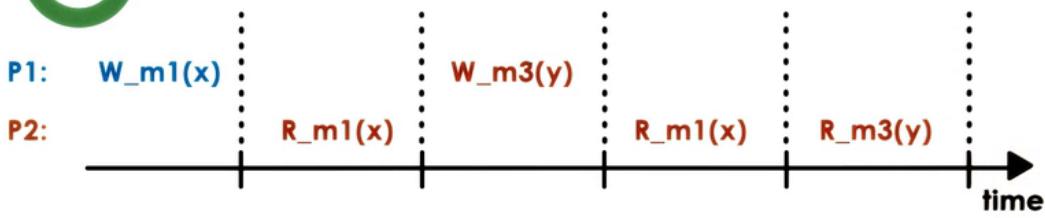
YES NO

For the next few quizzes I'll ask questions that are all related to the consistency models we just looked at. For each quiz I will show you a timeline and I will ask you whether if that particular execution is consistent according to some consistency model or not. Here is the first quiz. Consider the following sequence of operations. I'm showing two processes P and P and according to some real time they're performing certain operations. The question to you is is this particular execution of these two processes sequentially consistent? The possible answers are yes or no.

25 - Consistency Models Quiz 1



Consistency Model Quiz 1



Consider the following sequence of operations. Is this execution sequentially consistent?

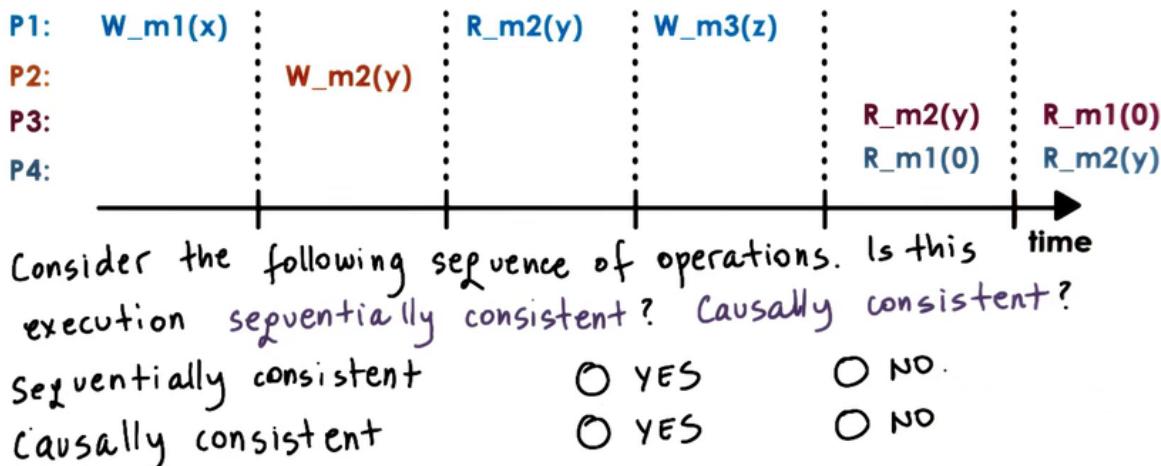
YES NO

This is a somewhat trivial example so the answer yes. This is a sequentially consistent execution. Any of the updates in the system are all performed from processor one. We know that sequential consistency requires that all of the updates from a single processor that they be visible in the same order. And if we take a look at P yes indeed. The fact that the memory location m was updated to value y only becomes visible after we have seen that on P that the value of m has become x. If we had a situation where these reads here to m were still returning zero. So this write was not visible however this read from m already returned y so this write became visible that would've been the problem but that's not the case in this scenario. So this is the correct answer.

26 - Consistency Models Quiz 2



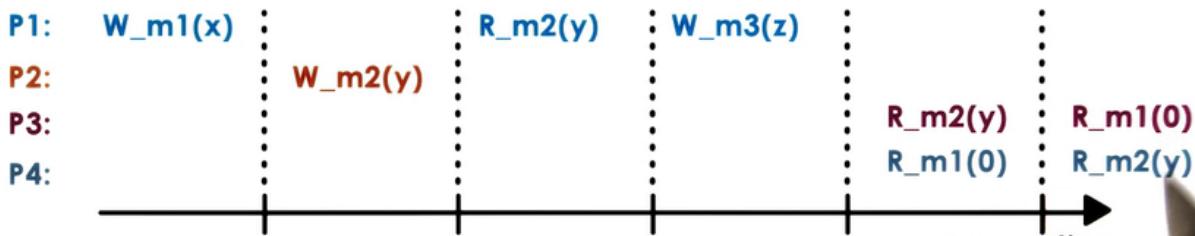
Consistency Model Quiz 2



Here is a second quiz. Again there are a number of processes P through P and their execution over time. And the question is considering the following sequence of operations is this execution sequentially consistent? And then what if the question was is it causally consistent? For each of the models sequentially consistent or causally consistent answer the question with yes or no.

27 - Consistency Models Quiz 2

Consistency Model Quiz 2



Consider the following sequence of operations. Is this execution sequentially consistent? Causally consistent?

Sequentially consistent
Causally consistent

- YES NO
 YES NO

ERRATA - This is sequentially consistent too, since $m1=0$ isn't a result of a particular write operation, so both P3 and P4 observe that m2 has changed, and m1 hasn't.

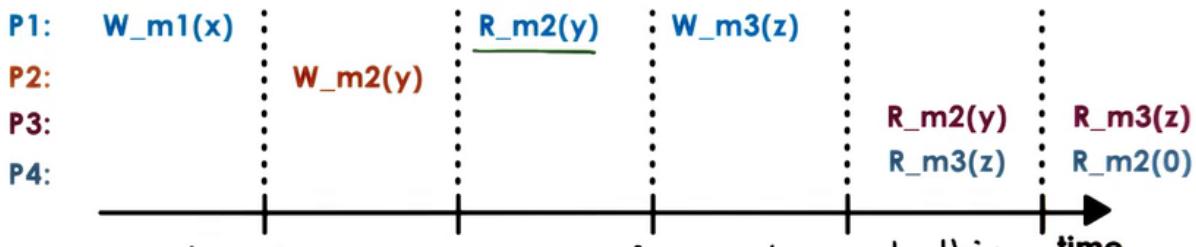
If we look at the execution here we see that P and P observe the updates to the memory allocations M and M in reverse way. We take a look at what's happening here we see that M and M are not causally related. So regarding causal consistency this is okay. This execution is correct with respect to causal consistency. However regarding sequential consistency we said that that all the processors in the system must observe the same order of the events that are occurring on some other processors. So in this case this would not be legal as far as sequential consistency is concerned. So the correct answers for these questions is this is not a valid execution for sequential consistency but it is with respect to causal consistency. Meaning that if we're running an application on top of a memory model that we know [INAUDIBLE] causal consistency if we observe this kind of behavior this is perfectly legal. We cannot complain to anyone. However if we observe this kind of behavior when running an application on top of a supposedly sequentially consistent system we have the rights to complain this is not correct. Somebody has made a mistake with their implementation of this system.

28 - Consistency Models Quiz 3

Now let's look at the quiz number three in this sequence. We have a very similar set of executions and processors P and P and slightly different focus on what's happening on P and P4. The question is the same. Considering the following sequence of operations is this execution sequentially consistent? You need to provide your answer yes or no. And also is this execution causally consistent? Again you need to provide your answer yes or no.

29 - Consistency Models Quiz 3

Consistency Model Quiz 3



Consider the following sequence of operations. Is this execution sequentially consistent? Causally consistent?

Sequentially consistent
Causally consistent

- YES NO
 YES NO

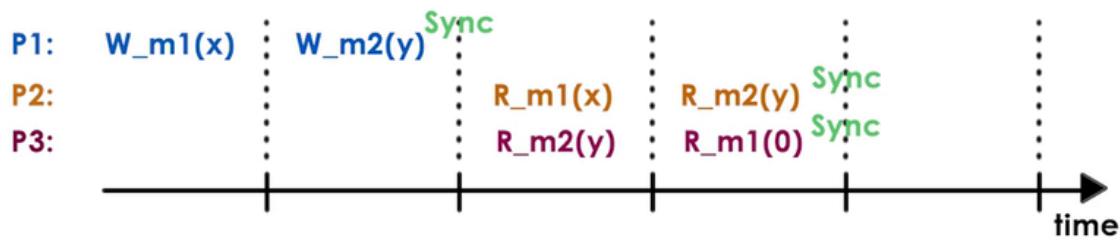


If you take a look at what's happening on P and P in this case we see that now P and P are observing reverse order of how they saw the updates to m and m2. As far as P is concerned P observed that this update to m happened and it also observed that this update to m happened. As far as P is concerned it observed this update of z to the memory location m3. But it still has not seen that P updated the memory location m with the value of y. It thinks that still the memory location m holds value zero. So P thinks that this write occurred before this write from P and then P thinks the other way around. This is clearly illegal as far as sequential consistency is concerned. Now if we take a look at these two updates we see that on P before the value of the memory location m was updated to z P read the value of the memory location m and it saw that that value was y. So it saw the effects of this particular update. What that means is that these operations are potentially causally related and therefore every processor in the system has to observe that this particular update happened after the update to m given the causal relationship between the two. Therefore the answer is no for both of these questions.

30 - Consistency Models Quiz 4



Consistency Model Quiz 4



Consider the following sequence of operations. Is this execution weakly consistent?

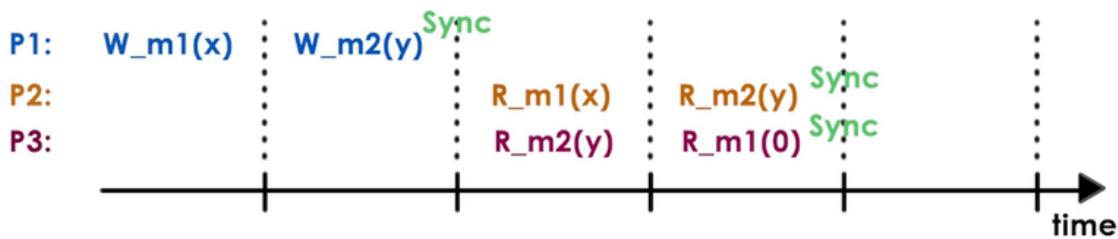
Yes No

Here's another quiz on the consistency model. Now we are looking at a weakly consistency model, so we see that in addition to writes and reads, the processor performs synchronization operations. And the question that I'm asking you is, given this sequence of operations, is this execution weakly consistent? Again, pick either yes or no.

31 - Consistency Models Quiz 4



Consistency Model Quiz 4



Consider the following sequence of operations. Is this execution weakly consistent?

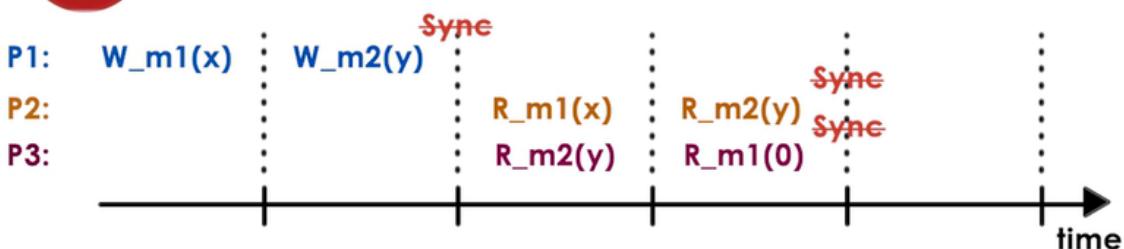
Yes No

The answer to this question is yes. Although P2 and P3 observe these operations in arbitrary way, neither one of them synchronized and forced the memory, the underlying distributed shared memory, to make any kind of guarantees regarding the updates that it observes. Weak consistency will not make any guarantees regarding the ordering unless explicit synchronization operations are used.

32 - Consistency Models Quiz 5



Consistency Model Quiz 5



If you ignore the Sync operations, is this execution causally consistent?

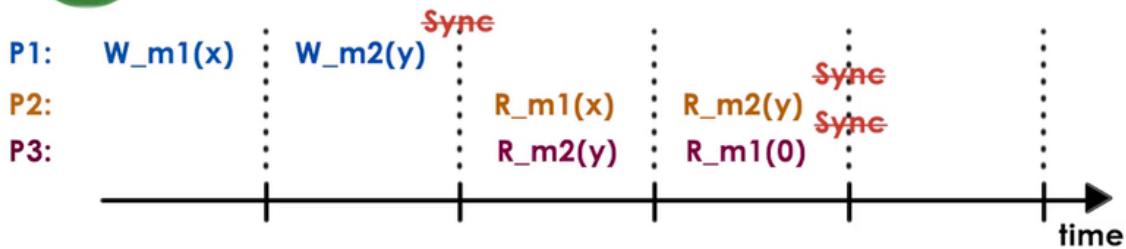
Yes No

And for the final quiz in this sequence, if you ignore this sync operations that we had in the fourth quiz, is the rest of the execution causally consistent? Yes or no?

33 - Consistency Models Quiz 5



Consistency Model Quiz 5



If you ignore the Sync operations, is this execution causally consistent?

Yes No

The answer to this question is no. It is not causally consistent because causal consistency does not permit that the writes from a single processor are arbitrarily reordered (writes from same processor must be in the same order).

34 - Lesson Summary



In this lesson we talked about distributed shared memory systems and about the mechanisms they use in order to manage that distributed state. We talked about coherence mechanisms we talked about consistency models that are meaningful in those environments,. As a final reminder note again that all of what we talked about was in the context of memory management. These types of methods apply to other applications that are responsible for managing distributed and shared state.