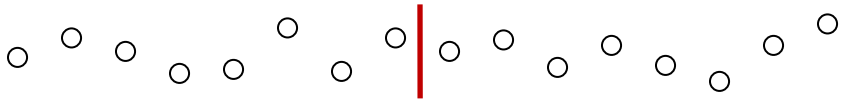
Relaxed Memory-Consistency Models

Review. Why are relaxed memory-consistency models needed?

How do relaxed MC models require programs to be changed?

The "safety net" between operations whose order needs to be guaranteed is often a *fence* instruction.



- The fence ensures that memory operations that are "younger" are not issued until the older mem ops have globally performed. The newer instruction must
  - wait until all older writes have been posted on the bus (or received InvAck);
  - wait until all older reads have completed;
  - flush the pipeline to avoid issuing younger mem ops early

- Programmers must insert fences.

What if amateur programmers perform their own synchronization, and forget fences?

A continuum of consistency models

Sequential consistency is one view of what a programming model should guarantee.

Let us introduce a way of diagramming consistency models. Suppose that—

- The value of a particular memory word in processor 2's local memory is 0.
- Then processor 1 writes the value 1 to that word of memory. Note that this is a remote write.
- Processor 2 then reads the word. But, being local, the read occurs quickly, and the value 0 is returned.

What's wrong with this?

This situation can be diagrammed like this (the horizontal axis represents time):

$P_{1:}$ $W(x)1$

$P_{2:}$ $R(x)0$

Depending upon how the program is written, it may or may not be able to tolerate a situation like this.

But, in any case, the programmer must understand what can happen when memory is accessed in a DSM system.

Sequential consistency

Sequential consistency: *The result of any execution is the same as if*

- *the memory operations of all processors were executed in some sequential order, and*
- *the operations of each individual processor appear in this sequence in the order specified by its program.*

Sequential consistency does *not* mean that writes are instantly visible throughout the system (it would be impossible to implement that anyway).

The example below illustrates two sequentially consistent executions.

Note that a read from P_2 is allowed to return an out-of-date value (because it has not yet "seen" the previous write).

$P_{1:}$ $W(x)1$

$P_{2:}$ $R(x)0$ $R(x)1$

$P_{1:}$ $W(x)1$

$P_{2:}$ $R(x)1$ $R(x)1$

From this we can see that running the same program twice in a row in a system with sequential consistency may not give the same results.

Causal consistency

The first step in weakening the consistency constraints is to distinguish between events that are potentially *causally* connected and those that are not.

Two events are causally related if one can influence the other.

P_1: $W(x)1$

P_2: $R(x)1$ $W(y)2$

Here, the write to x could influence the write to y, because

On the other hand, without the intervening read, the two writes would not have been causally connected:

P_1: $W(x)1$

P_2: $W(y)2$

The following pairs of operations are potentially causally related:

- A read followed by a later write by the same processor.
- A write followed by a later read to the same location.
- The transitive closure of the above two types of pairs of operations.

Operations that are not causally related are said to be *concurrent*.

Causal consistency: *Writes that are potentially causally related must be seen in the same order by all processors.*

Concurrent writes may be seen in a different order by different processors.

Here is a sequence of events that is allowed with a causally consistent memory, but disallowed by a sequentially consistent memory:

P_1:	$W(x)1$		$W(x)3$			
P_2:		$R(x)1$	$W(x)2$			
P_3:		$R(x)1$			$R(x)3$	$R(x)2$
P_4:		$R(x)1$			$R(x)2$	$R(x)3$

Why is this not allowed by sequential consistency?

Why is this allowed by causal consistency?

What is the violation of causal consistency in the sequence below?

P_1:	$W(x)1$				
P_2:		$R(x)1$	$W(x)2$		
P_3:				$R(x)2$	$R(x)1$
P_4:				$R(x)1$	$R(x)2$

Without the $R(x)1$ by P_2, this sequence would've been causally consistent.

Implementing causal consistency requires the construction of a dependency graph, showing which operations depend on which other operations.

Processor consistency

Causal consistency requires that all processes see causally related writes from *all* processors in the same order.

The next step is to relax this requirement, to require only that writes from the *same* processor be seen in order. This gives processor consistency.

Processor consistency: *Writes performed by a single processor are received by all other processors in the order in which they were issued.*

Writes from different processors may be seen in a different order by different processors.

Processor consistency would permit this sequence that we saw violated causal consistency:

P_1:	$W(x)1$				
P_2:		$R(x)1$	$W(x)2$		
P_3:				$R(x)2$	$R(x)1$
P_4:				$R(x)1$	$R(x)2$

Another way of looking at this model is that all writes generated by different processors are considered to be concurrent.

Note: Some definitions of processor consistency require cache coherence too. Processor consistency *without* cache coherence is called PRAM consistency.

Exercise: What is the strongest consistency model that each of the following satisfy?

P_1:	$W(x)1$				
P_2:		$R(x)1$	$W(x)2$		
P_3:				$R(x)1$	$R(x)2$
P_4:				$R(x)2$	$R(x)1$

P_1:	$W(y)1$				
P_2:		$R(x)1$	$W(y)2$		
P_3:				$R(y)1$	$R(y)2$
P_4:				$R(y)2$	$R(y)1$

$P_{1:}$	$W(x)1$			
$P_{2:}$		$R(x)1$	$W(y)2$	
$P_{3:}$				$R(x)1$ $R(y)2$
$P_{4:}$				$R(y)2$ $R(x)1$

Sometimes processor consistency can lead to counterintuitive results. Assume that *a* and *b* are initialized to 0.

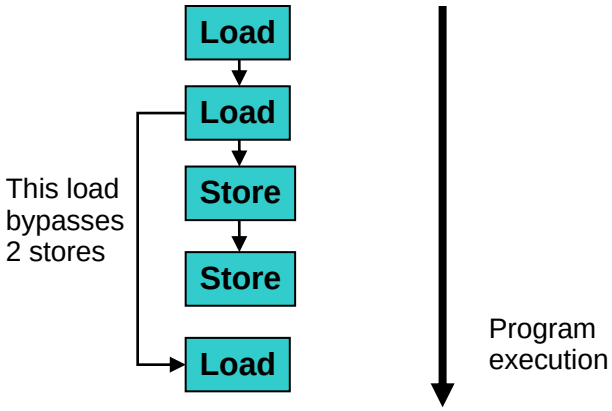
P_1:	P_2:
$a = 1;$	$b = 1;$
if $(b == 0)$	**if** $(a == 0)$
$kill(p_2);$	$kill(p_1);$

At first glance, it seems that no more than one process should be killed.

With processor consistency, however, it is possible for both to be killed. Explain how.

What processor consistency guarantees

- SC ensures ordering of
  - LD → LD
  - LD → ST
  - ST → LD
  - ST → ST
- PC removes the ST→LD constraint, with significant implications for ILP:
  - Values can be loaded into other caches, even if there's a store to the same location in some write buffer.
  - Loads do not wait for stores to complete ("perform"), they access the cache right away (without being speculative!).
  - A load dependent on an older store (in the same processor) can "bypass" (directly obtain the store value before it is stored).
- PC also removes write atomicity.



P1:
data = 2000;
flag = 1;

P2:
while (*flag* == 0) {};
print *data*;

PC produces SC results, because ordering between 2 stores is preserved.

P1:
flag1 = 1;
if (*flag2* == 0)
…

P2:
flag2 = 1;
if (*flag1* == 0)
…

PC fails to produce SC results, because PC does not guarantee ordering betw. store & younger load

- How close is PC to programmers' expectation?
  - Most of the time, very close (e.g., post-wait synchronization works correctly)
  - Major OSes are ported to PC with relative ease
- Cases that cause errors in PC usually are due to races that also happen in SC.
  - However, debugging races in PC is more difficult.

Weak ordering

Processor consistency is still stronger than necessary for many programs, because it requires that writes originating in a single processor be seen in order everywhere.

But it is not always necessary for other processors to see writes in order—or even to see all writes, for that matter.

Suppose a processor is in a tight loop in a critical section, reading and writing variables.

Other processes aren't supposed to touch these variables until the process exits its critical section.

Under processor consistency, the memory has no way of knowing that other processes don't care about these writes, so it has to propagate all writes to all other processors in the normal way.

To relax our consistency model further, we have to divide memory operations into two classes and treat them differently.

- Accesses to *synchronization variables* are sequentially consistent.
- Accesses to other memory locations can be treated as concurrent.

This strategy is known as *weak ordering*.

With weak ordering, we don't need to propagate accesses that occur during a critical section.

We can just wait until the process exits its critical section, and then—

- make sure that the results are propagated throughout the system, and
- stop other actions from taking place until this has happened.

Similarly, when we want to enter a critical section, we need to make sure that all previous writes have finished.

These constraints yield the following definition:

Weak ordering: *A memory system exhibits weak ordering iff—*

1. *Accesses to synchronization variables are sequentially consistent.*
2. *No access to a synchronization variable can be performed until all previous writes have completed everywhere.*
3. *No data access (read or write) can be performed until all previous accesses to synchronization variables have been performed.*

Thus, by doing a synchronization before reading shared data, a process can be assured of getting the most recent values written by other processes before their immediately preceding *S*s.

Note that this model does not allow more than one critical section to execute at a time, even if the critical sections involve disjoint sets of variables.

This model puts a greater burden on the programmer, who must decide which variables are synchronization variables.

Weak ordering says that memory does not have to be kept up to date between synchronization operations.

This is similar to how a compiler can put variables in registers for efficiency's sake. Memory is only up to date when these variables are written back.

If there were any possibility that another process would want to read these variables, they couldn't be kept in registers.

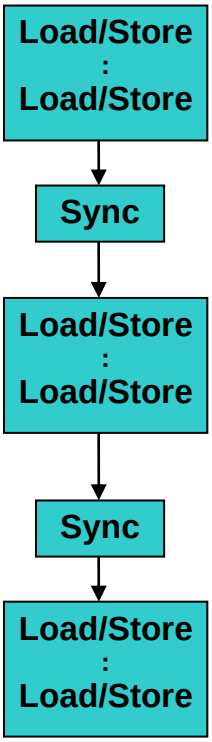
This shows that processes can live with out-of-date values, provided that they know when to access them and when not to.

The following is a legal sequence under weak ordering. Can you explain why?

P_1:	$W(x)1$	$W(x)2$	S		
P_2:			$R(x)2$	$R(x)1$	S
P_3:			$R(x)1$	$R(x)2$	S

Here's a sequence that's illegal under weak ordering. Why?

P_1:	$W(x)1$	$W(x)2$	S	
P_2:			S	$R(x)1$



Synch may be implemented as a lock acquire/release

Before a synch, all previous ops must finish.
Before any ld/st, all previous synch must finish.

Why safe? Typically within a critical section, we have made sure that only one process is inside, thus safe to reorder anything in the critical section.

Outside a critical section, we usually do not care about the order of mem ops (we would have used synchronization if we had cared).

How to know whether a particular ld/st serves as a synchronization point?

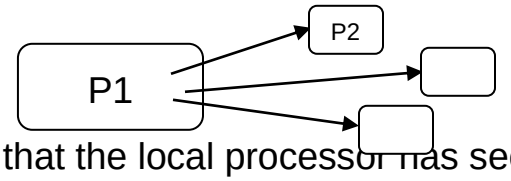
- Assume all atomic instructions are synchronization points
  - fetch-and-op, test-and-set
- Assume all load linked (LL) and store conditional (SC) are synchronization points

Release consistency

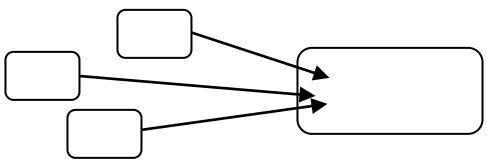
Weak ordering does not distinguish between entry to critical section and exit from it.

Thus, on both occasions, it has to take the actions appropriate to both:

- making sure that all locally initiated writes have been propagated to all other memories, and



- making sure that the local processor has seen all previous writes anywhere in the system.



If the memory could tell the difference between entry and exit of a critical section, it would only need to satisfy one of these conditions.

Release consistency provides two operations:

- *acquire* operations tell the memory system that a critical section is about to be entered.
- *release* operations say a c. s. has just been exited.

It is possible to acquire or release a single synchronization variable, so more than one critical section can be in progress at a time.

When an acquire occurs, the memory will make sure that all the local copies of shared variables are brought up to date.

When a release is done, the shared variables that have been changed are propagated out to the other processors.

But—

- doing an acquire does not guarantee that locally made changes will be propagated out immediately.
- doing a release does not necessarily import changes from other processors.

Here is an example of a valid event sequence for release consistency (*A* stands for "acquire," and *Q* for "release" or "quit"):

P_1:	$A(L)$ $W(x)1$ $W(x)2$ $Q(L)$		
P_2:		$A(L) R(x)2$ $Q(L)$	
P_3:			$R(x)1$

Note that since P_3 has not done a synchronize, it does not necessarily get the new value of x.

Release consistency: A system is release consistent if it obeys these rules:

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed.*

2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.*
3. *The acquire and release accesses must be processor consistent.*

If these conditions are met, and processes use *acquire* and *release* properly, the results of an execution will be the same as on a sequentially consistent memory.

Summary: Sequential consistency is possible, but costly. The model can be relaxed in various ways.
Consistency models not using synchronization operations:

Type of consistency	Description
Sequential	All processes see all shared accesses in same order.
Causal	All processes see all causally related shared accesses in the same order.
Processor	All processes see writes from each processor in the order they were initiated. Writes from different processors may not be seen in the same order, except that writes to the same location will be seen in the same order everywhere.

Consistency models using synchronization operations:

Type of consistency	Description
Weak	Shared data can only be counted on to be consistent after a synchronization is done.
Release	Shared data are made consistent when a critical region is exited.

The following diagram contrasts various forms of consistency.