Memory consistency: In a distributed memory system, references to memory in remote processors do not take place immediately.

This raises a potential problem. 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):

\[
P_1: \quad W(x)1 \\
\frac{}{P_2: \quad R(x)0}
\]

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.

Consistency models: A consistency model is essentially a contract between the software and the memory.

It says that iff the software agrees to obey certain rules, the memory promises to store and retrieve the expected values.
Strict consistency: The most obvious consistency model is strict consistency.

**Strict consistency:** Any read to a memory location $x$ returns the value stored by the most recent write operation to $x$.

This definition implicitly assumes the existence of a global clock, so that the determination of “most recent” is unambiguous.

Strict consistency is supported by uniprocessors, but it is generally impossible to implement in DSM systems.

In a DSM system, a read from a nonlocal memory location, or a write to a nonlocal memory location, requires sending a message.

This message requires a finite time to travel to and from the node where the memory is located. During that time, some other processor might change the value in the memory location.

Regardless of how efficient a message system is, it cannot cause messages to travel faster than the speed of light.

If the difference in the two access times is, say, one ns., and the nodes are 3 m. apart, the signal would have to travel at 10 times the speed of light to return the most recent value.

Fortunately, strict consistency is rarely necessary.

To summarize, with strict consistency,

- all writes are instantaneously visible to all processes;
all subsequent reads see the new value of the write, no matter how soon after the write the read is done, and no matter where the process executing the read is located.

**Sequential consistency:** Strict consistency isn’t really necessary to write parallel programs.

Earlier in this course, we learned that parallel programs shouldn’t make any assumptions about the relative speeds of the processes, or how their actions would interleave in time.

Counting on two events within one process to happen so quickly that another process won’t have time to do something in between is asking for trouble.

Let us weaken the requirements we impose on the memory so that the resulting model is realizable.

**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.

What’s the difference between strict consistency and this?

- In sequential consistency, the temporal ordering of events does not matter.
- All that is required is that the processors see the *same ordering* of memory operations (regardless of whether this is the order that the operations actually occurred).
So, with sequential consistency we don’t have to worry about providing “up to the nanosecond” results to all of the processors.

The example below shows the difference between strict and sequential consistency. The two sequences of operations are equally valid.

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).

\[
\begin{align*}
P_1: & \quad W(x) \ 1 \\ P_2: & \quad R(x) \ 0 \quad R(x) \ 1 \\
\end{align*}
\]

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.

While it is possible to implement, sequential consistency has very poor performance. Can you guess why?

\[
\begin{align*}
P_1: & \quad W(x) \ 1 \\ P_2: & \quad R(x) \ 1 \quad R(x) \ 1 \\
\end{align*}
\]

Causal consistency: The next 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.

\[
\begin{align*}
P_1: & \quad W(x) \ 1 \\
\end{align*}
\]
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:

\[
P_1: \quad W(x)1 \\
\quad W(y)2
\]

The following pairs of operations are potentially causally related:

- A read followed by a later write.
- A write followed by a later read.
- 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:

\[
\begin{array}{ccc}
P_1: & W(x)1 & W(x)3 \\
P_2: & R(x)1 & W(x)2 \\
P_3: & R(x)1 & R(x)3 & R(x)2 \\
P_4: & R(x)1 & R(x)3 & R(x)2 & R(x)3
\end{array}
\]
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?

\[
\begin{array}{c}
P_1: & W(x)1 \\
P_2: & R(x)1 \ W(x)2 \\
P_3: & R(x)2 \ R(x)1 \\
P_4: & R(x)1 \ R(x)2 \\
\end{array}
\]

Without the \( R(x)1 \) by \( P_2 \), this sequence would've been legal.

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

**PRAM 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 pipelined-RAM (PRAM) consistency.
**PRAM consistency**: Writes performed by a single process 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.

PRAM consistency is so named because writes can be pipelined; that is, a processor does not have to stall waiting for a write to be completed before starting the next one.

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

\[
\begin{align*}
P_1: & \quad W(x)1 \\
P_2: & \quad R(x)1 \ W(x)2 \\
P_3: & \quad R(x)2 \ R(x)1 \\
P_4: & \quad R(x)1 \ R(x)2
\end{align*}
\]

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

Sometimes PRAM consistency can lead to counterintuitive results.

\[
\begin{align*}
P_1: & \quad a := 0; \\
& \quad \vdots \\
& \quad a := 1; \\
& \quad \textbf{if} \ b = 0 \ \textbf{then} \quad \text{kill}(p_2); \\

P_2: & \quad b := 0; \\
& \quad \vdots \\
& \quad b := 1; \\
& \quad \textbf{if} \ a = 0 \ \textbf{then} \quad \text{kill}(p_1);
\end{align*}
\]

At first glance, it seems that no more than one process should be killed. With PRAM consistency, however, it is possible for both to be killed.
Processor consistency: Processor consistency is very similar to PRAM consistency, but it has one additional condition: memory coherence.

Memory coherence requires that writes to the same location be viewed in the same order by all the processors.

Writes to different locations need not be seen in the same order by different processors.

Processor consistency = \( PRAM \) consistency + memory coherence.

Weak consistency: PRAM and processor consistency are still stronger than necessary for many programs, because they require 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 PRAM 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 consistency*.

With weak consistency, 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 consistency:** A memory system exhibits weak consistency 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.

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 consistency 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 consistency. Can you explain why?

\[
\begin{array}{c|c|c|c}
P_1: & W(x)_1 & W(x)_2 & S \\
\hline
P_2: & R(x)_2 & R(x)_1 & S \\
\hline
P_3: & R(x)_1 & R(x)_2 & S \\
\end{array}
\]
Here’s a sequence that’s illegal under weak consistency. Why?

\[ P_1: W(x)1 \quad W(x)2 \quad S \]
\[ P_2: \quad S \quad R(x)1 \]

**Release consistency**: Weak consistency 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 c.s. 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”):

\[
\begin{align*}
P_1: & \quad A(L) \ W(x)1 \ W(x)2 \ Q(L) \\
\hline
P_2: & \quad A(L) \ R(x)2 \ Q(L) \\
\hline
P_3: & \quad R(x)1
\end{align*}
\]

Note that since \( P_3 \) has not

**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:** Strict consistency is impossible. Sequential consistency is possible, but costly. The model can be relaxed in various ways.

Consistency models not using synchronization operations:

<table>
<thead>
<tr>
<th>Type of consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>All processes see absolute time ordering of all shared accesses.</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in same order.</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see all causally related shared accesses in the same order.</td>
</tr>
<tr>
<td>Processor PRAM</td>
<td>PRAM consistency + memory coherence</td>
</tr>
<tr>
<td>Processor RAM</td>
<td>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.</td>
</tr>
</tbody>
</table>

Consistency models using synchronization operations:

<table>
<thead>
<tr>
<th>Type of consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Shared data can only be counted on to be consistent after a synchronization is done.</td>
</tr>
<tr>
<td>Release</td>
<td>Shared data are made consistent when a critical region is exited.</td>
</tr>
</tbody>
</table>

The following diagram contrasts various forms of consistency.
In summary,

- In sequential consistency, LD and ST follow program order.

- In PRAM OR processor consistency, LD may precede buffered ST; other processors may see different order of \{ST, LD\} access.

- In weak consistency, SYNCH operations are sequentially consistent; other memory operations can occur in any order.

- In release consistency, SYNCH operations are split into ACQUIRE (lock) and RELEASE (unlock), and these operations are processor consistent.