CS6868 - Concurrent Programming

Memory Consistency Models

V. Krishna Nandivada

IIT Madras

Memory Consistency Models

- A memory consistency model is

Memory Consistency Models

- A memory consistency model is
  - a set of rules governing how the memory systems will process memory operations from multiple processors.

Memory Consistency Models

- A memory consistency model is
  - a set of rules governing how the memory systems will process memory operations from multiple processors.
    - Order in which memory operations will appear to execute - determines what value should a read return?

Memory Consistency Models

- A memory consistency model is
  - a set of rules governing how the memory systems will process memory operations from multiple processors.
    - Order in which memory operations will appear to execute - determines what value should a read return?
  - a contract between programmer and system.

Memory Consistency Models

- A memory consistency model is
  - a set of rules governing how the memory systems will process memory operations from multiple processors.
    - Order in which memory operations will appear to execute - determines what value should a read return?
  - a contract between programmer and system.
  - Determines what optimizations can be performed for correct programs.

Memory Consistency Models

- A memory consistency model is
  - a set of rules governing how the memory systems will process memory operations from multiple processors.
    - Order in which memory operations will appear to execute - determines what value should a read return?
  - a contract between programmer and system.
  - Determines what optimizations can be performed for correct programs.
- Affects : Ease of programming, and performance

Uniprocessor Memory model

- Memory value requirement: Memory operations occur in program order: read returns the value of the last write in program order.
- Simple to reason about.
- Compiler optimizations preserve these semantics.
- Independent operations can execute in parallel.

Strict consistency

- Strictest memory model.
- Requires that the ‘read’ should get the value written by the last ‘write’.

Strict consistency

- Strictest memory model.
- Requires that the ‘read’ should get the value written by the last ‘write’.
- Requires a Global clock

Strict consistency

- Strictest memory model.
- Requires that the ‘read’ should get the value written by the last ‘write’.
- Requires a Global clock $\equiv$ Halting problem.

Sequential consistency

[Lamport]: A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program.

Sequential consistency

Result of an execution appears as if:

- All operations executed in some sequential order.
- Memory operations of each process in program order.
- Nothing specified about caches, write buffers.

Understanding Program Order. Dekker's Algorithm

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Execution:

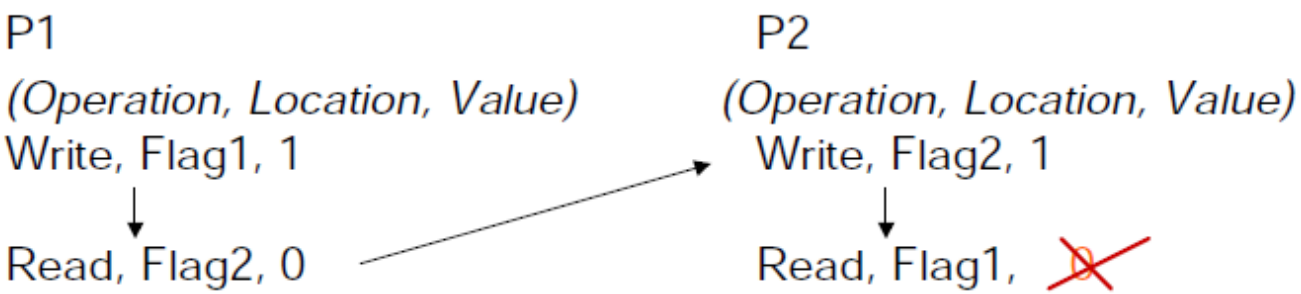
P1	P2
(Operation, Location, Value)	*(Operation, Location, Value)*
Write, Flag1, 1	Write, Flag2, 1
Read, Flag2, 0	Read, Flag1, ___

Understanding Program Order. Dekker's Algorithm

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Execution:



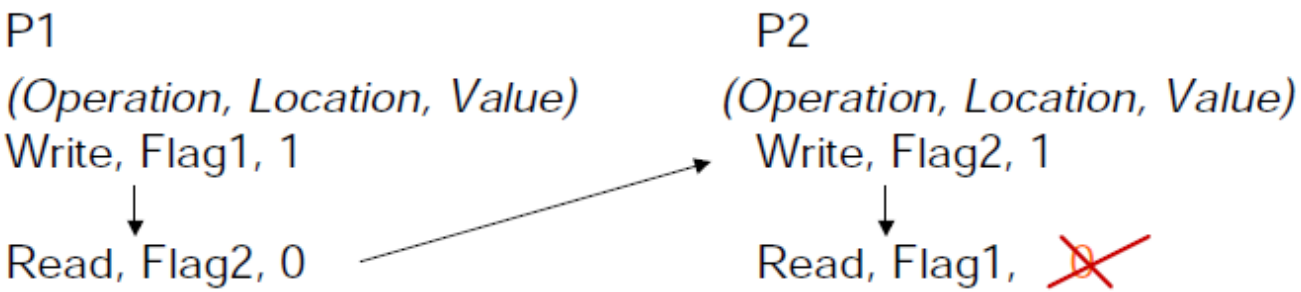
Reads of 1 by `Flag1` `Flag2` are valid.

Understanding Program Order. Dekker's Algorithm

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Execution:



Reads of 1 by `Flag1` `Flag2` are valid. Problematic situation

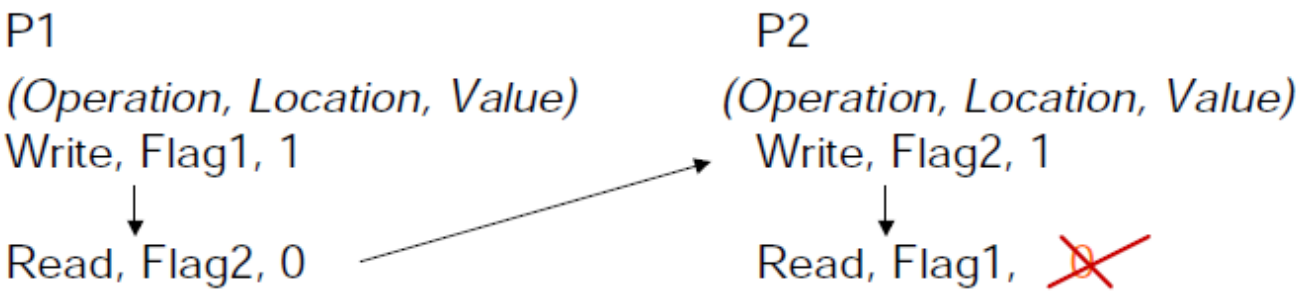
- Write buffers with read bypassing.

Understanding Program Order. Dekker's Algorithm

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Execution:



Reads of 1 by `Flag1` `Flag2` are valid. Problematic situation

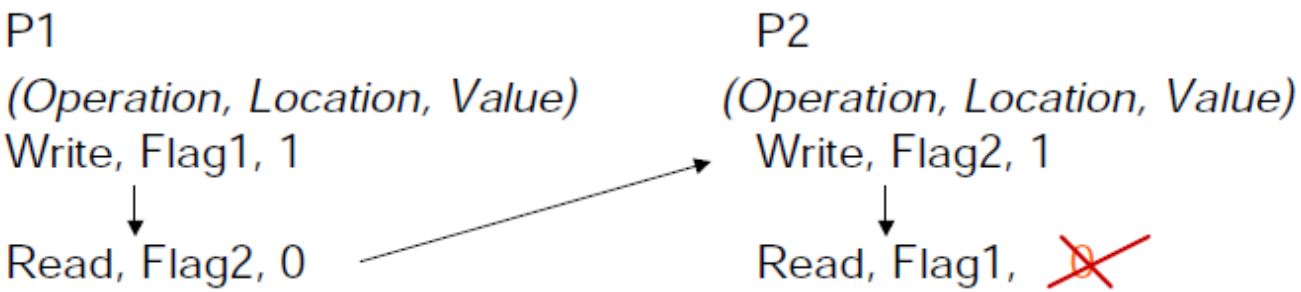
- Write buffers with read bypassing.
- Overlap or reorder writes/reads by compiler / hardware.

Understanding Program Order. Dekker's Algorithm

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Execution:



Reads of 1 by `Flag1 Flag2` are valid. Problematic situation

- Write buffers with read bypassing.
- Overlap or reorder writes/reads by compiler / hardware.
- Values in registers.

Understanding Program Order. Ex 2

Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

P1
Write, A, 23
Write, Flag, 1

P2
Read, Flag, 0

Read, Flag, 1
Read, A, _____

Understanding Program Order. Ex 2

Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

P1
Write, A, 23
Write, Flag, 1

P2
Read, Flag, 0

Read, Flag, 1
Read, A, ~~0~~

Problematic situation

- Overlap or reorder writes/reads by compiler / hardware.

Write Atomicity

Initially A = B = C = 0

P1	P2	P3	P4
A = 1;	A = 2;	while (B != 1) ;	while (B != 1) ;
B = 1;	C = 1;	while (C != 1) ;	while (C != 1) ;
		tmp1 = A;	tmp2 = A;

Q: What are the possible values of tmp1 and tmp2?

Write Atomicity

Initially A = B = C = 0

P1	P2	P3	P4
A = 1;	A = 2;	while (B != 1) ;	while (B != 1) ;
B = 1;	C = 1;	while (C != 1) ;	while (C != 1) ;
		tmp1 = A;	tmp2 = A;

Q: What are the possible values of tmp1 and tmp2?

Q: Can tmp1 = 1 and tmp2 = 2 be possible? How?

Write Atomicity

Initially A = B = C = 0

P1	P2	P3	P4
A = 1;	A = 2;	while (B != 1) ;	while (B != 1) ;
B = 1;	C = 1;	while (C != 1) ;	while (C != 1) ;
		tmp1 = A;	tmp2 = A;

Q: What are the possible values of tmp1 and tmp2?

Q: Can tmp1 = 1 and tmp2 = 2 be possible? How?

- Cache coherence protocol must serialize writes to same location.
- Writes to same location should be seen in same order by all.

Atomicity Ex 2

Initially A = B = 0

P1	P2	P3
A = 1	while (A != 1) ;	while (B != 1) ;
	B = 1;	tmp = A

P1	P2	P3
Write, A, 1		
	Read, A, 1	
	Write, B, 1	
		Read, B, 1
		Read, A, ~~0~~

Atomicity Ex 2

Initially A = B = 0

P1	P2	P3
A = 1	while (A != 1) ;	while (B != 1) ;
	B = 1;	tmp = A

P1	P2	P3
Write, A, 1		
	Read, A, 1	
	Write, B, 1	
		Read, B, 1
		Read, A, ~~0~~

- if ‘read’ returns a new value before all copies see it.

Atomicity Ex 2

Initially A = B = 0

P1	P2	P3
A = 1	while (A != 1) ;	while (B != 1) ;
	B = 1;	tmp = A

P1	P2	P3
Write, A, 1		
	Read, A, 1	
	Write, B, 1	
		Read, B, 1
		Read, A, ~~0~~

- if 'read' returns a new value before all copies see it.
- Read others'-write early optimization is unsafe.

Sequential Consistency implementation

Implementations of this model must satisfy the following:

- Program Order Requirement : The operations of same processor must be executed in program order

Sequential Consistency implementation

Implementations of this model must satisfy the following:

- Program Order Requirement : The operations of same processor must be executed in program order
- Write Atomicity : All writes appear to be instantaneous (no buffer).

Sequential Consistency implementation

Implementations of this model must satisfy the following:

- Program Order Requirement : The operations of same processor must be executed in program order
- Write Atomicity : All writes appear to be instantaneous (no buffer).
- All processors must see all write operations in the same order (cache coherence).

Sequential Consistency implementation

Implementations of this model must satisfy the following:

- Program Order Requirement : The operations of same processor must be executed in program order
- Write Atomicity : All writes appear to be instantaneous (no buffer).
- All processors must see all write operations in the same order (cache coherence).
- Easier to implement in architectures with no cache, no write buffers, blocking reads, .

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

  Implications (not allowed)
  - Read others' write early.
  - Read own write early.
  - Unserialized writes to the same location.

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

  Implications (not allowed)
  - Read others' write early.
  - Read own write early.
  - Unserialized writes to the same location.
- Simple model to reason about given parallel programs.

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

  Implications (not allowed)
  - Read others' write early.
  - Read own write early.
  - Unserialized writes to the same location.
- Simple model to reason about given parallel programs.
- Makes it very hard to modify a parallel program (automatic and manual)
  - Processor reordering for performance - write buffers, overlapped writes, non-blocking reads

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

  Implications (not allowed)
  - Read others' write early.
  - Read own write early.
  - Unserialized writes to the same location.
- Simple model to reason about given parallel programs.
- Makes it very hard to modify a parallel program (automatic and manual)
  - Processor reordering for performance - write buffers, overlapped writes, non-blocking reads
  - Compiler transformations - scalar replacement, register allocation, instruction scheduling.

Sequential Consistency - issues

- Sequential Consistency constraints
  - write $\rightarrow$ read
  - write $\rightarrow$ write
  - read $\rightarrow$ read, write

  Implications (not allowed)
  - Read others' write early.
  - Read own write early.
  - Unserialized writes to the same location.
- Simple model to reason about given parallel programs.
- Makes it very hard to modify a parallel program (automatic and manual)
  - Processor reordering for performance - write buffers, overlapped writes, non-blocking reads
  - Compiler transformations - scalar replacement, register allocation, instruction scheduling.
  - Programmer reordering code for aesthetics/SE requirements.

Sequential consistency - too strict

- Many architectures do not give SC.
- Compiler optimizations on SC are limited.
- Sofwtware engineering issues.

- Give up!
- Use weaker models - relax the program order requirement and write atomicity requirement.

Sequential consistency (English)

- Memory operations of each process happens in program order.
- any valid interleaving of read and write operations is OK.
- all processes must see the same interleaving.

Sequential consistency examples

P1	W(x)1				
P2		W(x)2			
P3			R(x)2		R(x)1
P4				R(x)2	R(x)1

Sequentially consistent - as both P3 and P4 see writes in the same sequential order.

Sequential consistency (counter) example

P1	W(x)1				
P2		W(x)2			
P3			R(x)2		R(x)1
P4				R(x)1	R(x)2

Sequentially inconsistent - as both P3 and P4 see writes in the two different sequential orders.

Sequential consistency (counter) example

P1	P2	P3
x = 1;	y = 1	z = 1
print(y,z)	print (x,z)	print (x,y)

Sequential consistency (counter) example

P1	P2	P3
x = 1;	y = 1	z = 1
print(y,z)	print (x,z)	print (x,y)

Inconsistent execution:

1. x = 1
2. print (y, z);
3. print (x, z);
4. y = 1;
5. z = 1;
6. print (x, y);

Causal Consistency

- Slightly weaker than Sequential Consistency Model.

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.
- Causal order is transitive.

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.
- Causal order is transitive.
  - memory operations that are causally related must have a total order and

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.
- Causal order is transitive.
  - memory operations that are causally related must have a total order and
  - program order for the ones issued by same processor.

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.
- Causal order is transitive.
  - memory operations that are causally related must have a total order and
  - program order for the ones issued by same processor.
- Hence such memory operations must be seen in same order by all processors.

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.
- Causal order is transitive.
  - memory operations that are causally related must have a total order and
  - program order for the ones issued by same processor.
- Hence such memory operations must be seen in same order by all processors.
- Here, write atomicity has been slightly weakened.

Causal Consistency

- Slightly weaker than Sequential Consistency Model.
- Causally related memory operations : issued by same processor and access same memory location - are seen by every node in causal order.
- Causal order is transitive.
  - memory operations that are causally related must have a total order and
  - program order for the ones issued by same processor.
- Hence such memory operations must be seen in same order by all processors.
- Here, write atomicity has been slightly weakened.
- weaker than sequential consistency, which requires that all nodes see all writes in the same order.

Causal consistency (example)

P1	W(x)1			W(x)3		
P2		R(x)1	W(x)2			
P3		R(x)1			R(x)3	R(x)2
P4		R(x)1		R(x)2	R(x)3	

Causally consistent, but not sequentially/strict consistent.

- Processors may see different order.
- All orders respect causal order (program order and read-write order).
- Has no global order, partial order for each processor.

Causal consistency (counter) Example

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

- Violates causal consistency.
- Removing the Read from the P2 – makes the execution causally consistent.

PRAM consistency

- All processes see memory writes from one process in the order they were issued from the process.

PRAM consistency

- All processes see memory writes from one process in the order they were issued from the process.
- Writes from different processes may be seen in a different order on different processes.

PRAM consistency

- All processes see memory writes from one process in the order they were issued from the process.
- Writes from different processes may be seen in a different order on different processes.
- no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order, as though they were in a pipeline.

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

PRAM consistency

- All processes see memory writes from one process in the order they were issued from the process.
- Writes from different processes may be seen in a different order on different processes.
- no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order, as though they were in a pipeline.

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

- PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict

PRAM consistency

- All processes see memory writes from one process in the order they were issued from the process.
- Writes from different processes may be seen in a different order on different processes.
- no guarantees about the order in which different processes see writes, except that two or more writes from a single source must arrive in order, as though they were in a pipeline.

P1	W(x)1				
P2		R(x)1	W(x)2		
P3				R(x)2	R(x)1
P4				R(x)1	R(x)2

- PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict
- (Also known as, FIFO consistency, or Processor consistency)

Weak Ordering

- Divide memory operations into data operations and synchronization operations

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.
  - Synchronizations are performed in program order.

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.
  - Synchronizations are performed in program order.
  - All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.
  - Synchronizations are performed in program order.
  - All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
  - All other accesses may be seen in different order on different processes

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.
  - Synchronizations are performed in program order.
  - All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
  - All other accesses may be seen in different order on different processes
- Illusion of write atomicy has to be maintained.

Weak Ordering

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence.
  - All data operations before synch in program order must complete before synch is executed.
  - All data operations after synch in program order must wait for synch to complete.
  - Synchronizations are performed in program order.
  - All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
  - All other accesses may be seen in different order on different processes
- Illusion of write atomicy has to be maintained.
- Hardware implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed.

Weak Ordering

- Example 1:

P1	W(x)1 W(x)2 `Sync`	
P2		R(x)1 R(x)2 `Sync`
P3		R(x)2 R(x)1 `Sync`

Weak Ordering

- Example 1:

P1	W(x)1 W(x)2 `Sync`	
P2		R(x)1 R(x)2 `Sync`
P3		R(x)2 R(x)1 `Sync`

Weak Ordering

- Example 1:

P1	W(x)1 W(x)2 `Sync`	
P2		R(x)1 R(x)2 `Sync`
P3		R(x)2 R(x)1 `Sync`

- Example 2:

P1	W(x)1 W(x)2 `Sync`	
P2		`Sync`R(x)2

Weak Ordering

- Example 1:

P1	W(x)1 W(x)2 `Sync`	
P2		R(x)1 R(x)2 `Sync`
P3		R(x)2 R(x)1 `Sync`

- Example 2:

P1	W(x)1 W(x)2 `Sync`	
P2		`Sync`R(x)2

Weak Ordering

- Example 1:

P1	W(x)1 W(x)2 `Sync`	
P2		R(x)1 R(x)2 `Sync`
P3		R(x)2 R(x)1 `Sync`

- Example 2:

P1	W(x)1 W(x)2 `Sync`	
P2		`Sync`R(x)2

- The programmer has to manage synchronization explicitly.

Weak Ordering

- Example 1:

P1	W(x)1 W(x)2 `Sync`	
P2		R(x)1 R(x)2 `Sync`
P3		R(x)2 R(x)1 `Sync`

- Example 2:

P1	W(x)1 W(x)2 `Sync`	
P2		`Sync`R(x)2

- The programmer has to manage synchronization explicitly.
- Weak $\leq$ PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict

Weak consistency (counter) example

P1 W(x)1 W(x)2 `Sync`

P2 `Sync`R(x)1

- P2 will observe the most recent write of the variable x, which has the value 2. Thus, it's not a valid sequence.

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.
  - But accesses after ‘release’ in program order do not have to wait for release (unless protected by another acquire).

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.
  - But accesses after ‘release’ in program order do not have to wait for release (unless protected by another acquire).
- do “acquite” = that writes on other processors to protected variables will be known

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.
  - But accesses after ‘release’ in program order do not have to wait for release (unless protected by another acquire).
- do “acquite” = that writes on other processors to protected variables will be known
- do “release” = that writes to protected variables are exported

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.
  - But accesses after ‘release’ in program order do not have to wait for release (unless protected by another acquire).
- do “acquite” = that writes on other processors to protected variables will be known
- do “release” = that writes to protected variables are exported
- and will be seen by other machines when they do a lock (lazy release consistency) or immediately (eager release consistency)

Release Consistency

- A problem with weak consistency: when a synchronization variable is accessed, we do not know whether it is done because the process is finished writing shared data or is about to start reading data.
- Synchronization instructions divided : Acquire (such as lock) and Release (such as unlock).
- Acquire: Any memory operation after acquire must be executed only after acquire is completed (and seen by all).
- Release :
  - Release must be executed only when all memory operations statements are complete.
  - But accesses after ‘release’ in program order do not have to wait for release (unless protected by another acquire).
- do “acquite” = that writes on other processors to protected variables will be known
- do “release” = that writes to protected variables are exported
- and will be seen by other machines when they do a lock (lazy release consistency) or immediately (eager release consistency)
- Total order among all synchronization instructions must be maintained.

Weak and Release comparison

- Weak: Shared data can be counted on to be consistent only after a synchronization is done.
- Release: Shared data are made consistent when a critical region is exited.

Release Consistency - example

- Example:

```
P1:   L W(x)1 W(x)2 U
P2:                     L R(x)2 U
P3:                                 R(x)1
```

- RC $\leq$ Weak $\leq$ PRAM $\leq$ Causal $\leq$ SC $\leq$ Strict

Delta and Eventual consistency models

- **Delta consistency**: The write operations will propagate through the shared memory system and all the replicas will be consistent after a fixed time period δ.

Delta and Eventual consistency models

- **Delta consistency**: The write operations will propagate through the shared memory system and all the replicas will be consistent after a fixed time period δ.
  - if an object is modified, during the short period of time following its modification, the read may not be consistent.

Delta and Eventual consistency models

- **Delta consistency**: The write operations will propagate through the shared memory system and all the replicas will be consistent after a fixed time period δ.
  - if an object is modified, during the short period of time following its modification, the read may not be consistent.
  - after a fixed time period, the modification is propagated and the read will be consistent.

Delta and Eventual consistency models

- **Delta consistency**: The write operations will propagate through the shared memory system and all the replicas will be consistent after a fixed time period δ.
  - if an object is modified, during the short period of time following its modification, the read may not be consistent.
  - after a fixed time period, the modification is propagated and the read will be consistent.
- **Eventual Consistency Model** : The writes propagates eventually (we cannot have a fixed bound on the delay)

Eventual Consistency

- Allow stale reads, but ensure that reads will eventually reflect previously written values (no guarantees on delays)
- Doesnt order concurrent writes as they are executed, which might create conflicts later: which write was first?
- More concurrency opportunities than strict, sequential, or causal consistency.
- Used a lot: Amazon: Dynamo, a key/value store, file synchronization

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.
- However, from a programmer's perspective, it is not clear how to use these effectively.

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.
- However, from a programmer's perspective, it is not clear how to use these effectively.
  - How to reason about programs for systems with relaxed memory models

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.
- However, from a programmer's perspective, it is not clear how to use these effectively.
  - How to reason about programs for systems with relaxed memory models
  - How to use the safety nets minimally, to get the desired semantics from program

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.
- However, from a programmer's perspective, it is not clear how to use these effectively.
  - How to reason about programs for systems with relaxed memory models
  - How to use the safety nets minimally, to get the desired semantics from program
- Even Sequential Consistency is not simple enough.

Programmer centric models

- Problem with relaxed models is that most of them are based on the performance optimization that can be performed.
- However, from a programmer's perspective, it is not clear how to use these effectively.
  - How to reason about programs for systems with relaxed memory models
  - How to use the safety nets minimally, to get the desired semantics from program
- Even Sequential Consistency is not simple enough.
- We need models which is simple for the programmer, but provides enough information about program to apply optimization and get efficiency.

Programmer centric models

Programmers understand their code:

- Different operations have different semantics

P1	P2
A = 23;	while (Flag != 1) ;
B = 37;	. . . = B;
Flag = 1;	. . . = A;

- Flag = Synchronization; A, B = Data
- Can reorder data operations
- Distinguish data and synchronization

Data Race Free 0 - DRF0

- Data-Race-Free-0 Program
  - All access distinguished as either synchronization or data.
  - All races distinguished as synchronization (in any SC execution).
- Data-Race-Free-0 Model
  - Guarantees SC to data-race-free-0 programs.
  - Others - reads return value of some write to the location.

A program is considered to be data-race-free-0 if and only if (i) for any sequentially consistent execution, all conflicting accesses are ordered by the happens-before relation, and (ii) all synchronization operation in the program are recognizable by the hardware and each accesses exactly a single memory location.

Programming with Data Race Free 0 - DRF0

- Needed information: for each operation: if it will race (in any SC execution).
- Procedure:
  - Write program assuming SC.
  - For each memory operation in the program:
    - Guaranteed to be no races? “Data access”: “synchronization”.

Problems with data race free model

- It does not define any semantics for programs with data races.
- A concern for safe languages like Java, which provide safety for any program and cannot let the behavior of a program to be ambiguous.
- Either define safe semantics for such programs or identify them and prevent their execution.
- Define higher abstractions for programmers which are inherently data race free
- Expensive for hardware to implement

Goals of Memory model

- Programmability? - Lost intuitive interface of SC
- Portability? - Many different models
- Performance? - Can we do better?

Future:

- Parallel programs today are inherently non deterministic
- We need deterministic outcomes from our parallel programs.
- Deterministic Outcomes from Inherent non determinism. Possible?

Advantages from relaxed models

- Gains both in the H/W and compiler.
- Gains in H/W (during execution):
  - latency hiding - can overlap many reads and writes.
- Gain by the compiler:
  - more operations can be reordered. (compare with SC).
  -

Sources

- Wikipedia
- fixstars.com
- Jernej Barbic slides.
- Loop Chunking in the presence of synchronization.
- Vivek Sarkar's slides.
- Sarita Adve's slides.
- Nimit's Singhania's presentation.
- http://regal.csep.umflint.edu/ swturner/Classes/csc577/Online/Chapter06/Chapter06.html
- Java Memory Model JSR-133: "Java Memory Model and Thread Specification Revision"