CS4223
Memory Consistency Models

Trevor E. Carlson
National University of Singapore
tcarlson@comp.nus.edu.sg
(Slides from Tulika Mitra)

Learning Outcomes

- What is memory consistency Model and why is it important?
- Sufficient conditions for memory consistency
- Sequential Consistency
- Relaxed consistency models:
  - Total Store Ordering (TSO)
  - Partial Store Ordering (PSO)
  - Processor Consistency (PC)
  - Weak Ordering (WO)
  - Release Consistency (RC)

Memory Consistency

- How to establish order between a write and a read to the same location by different processors?
  - Synchronization

P1	P2
A = 1;	while (flag == 0);
flag = 1;	B = A;

- Programmer's intention: B = 1
- Is intention guaranteed by coherence ?

Formal Definition of Coherence

- A memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of execution and in which:
  - Operations issued by any particular process occur in program order
  - The value returned by a read is the value returned by the last write to that location in serial order

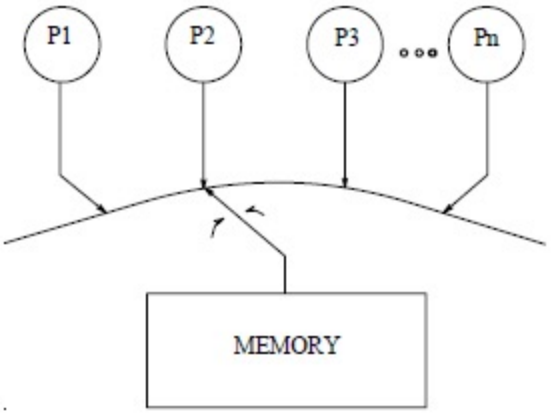
What is the problem?

- Programmer expects memory to respect order between accesses to different locations issued by a given process to preserve orders among accesses to the same location by different processes
- Coherence doesn't help as it pertains to a single location
- Need an ordering model across different locations

Memory Consistency Model

- Contract between system and programmer
- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
- Programmers use to reason about correctness
- System designers use to decide the reordering possible by hardware and compiler

Sequential Consistency (SC)



- A multiprocessor in 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 this sequence in program order.

More Formally

- Each process's program order imposes partial order on set of all operations
- Interleaving of these partial orders define a total order on all operations
- Many total orders may be SC (SC does not define particular interleaving)

More Formally (Contd.)

- SC Execution: An execution of a program is SC if the results it produces are the same as those produced by some possible total order (interleaving)
- SC System: A system is SC if any possible execution on that system is an SC execution

Example 1

P1	P2
A=1	print B
B=1	print A

What are the possible values of B and A under SC?

Example 2

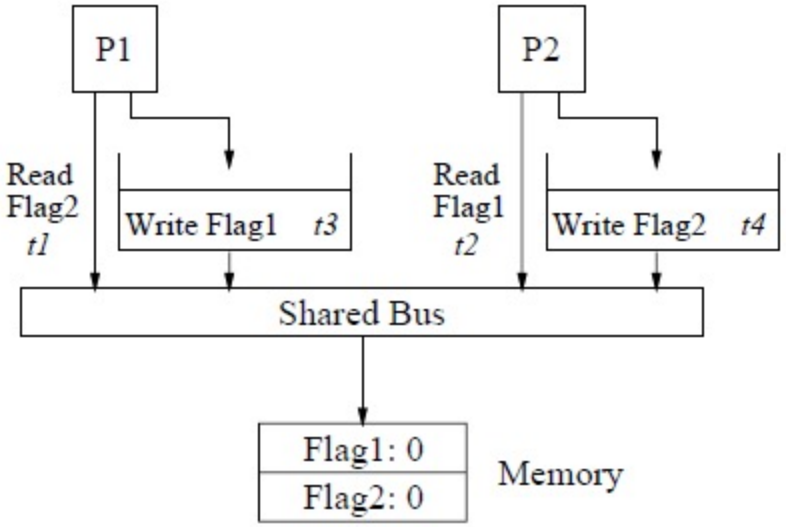
P1	P2
A=1	B = 1
print B	print A

What are the possible values of B and A under SC?

Sufficient Conditions for SC

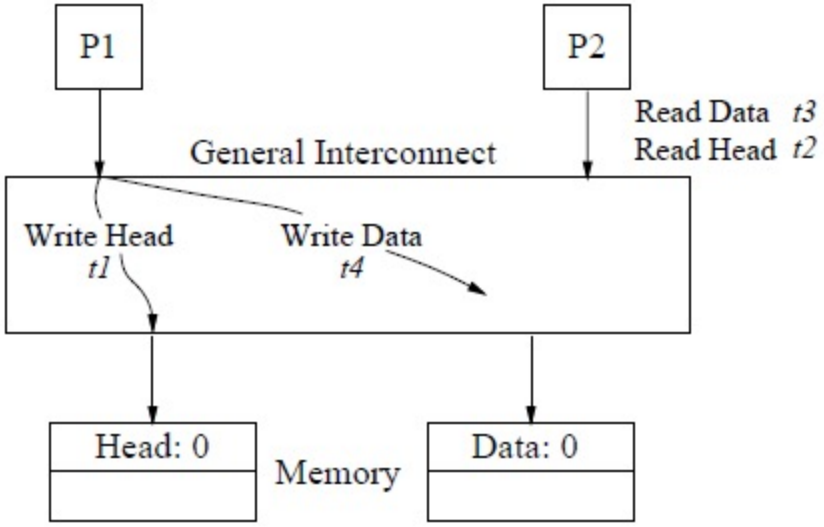
- Program Order Requirement
  - A processor must ensure that its previous memory operation is complete before proceeding with its next memory operation in program order
- Write atomicity
  - Writes to the same location be serialized, i.e., writes to the same location be made visible in the same order to all processors
  - Value of a write not returned by a read until all invalidates or updates generated by the write are acknowledged, i.e., until the write becomes visible to other processors

Program Order Violation: Write Buffer



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

Program Order Violation: Overlapped Writes



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Write atomicity Condition 1

P1	P2	P3	P4
A = 1	A = 2	while (B == 0)	while (B==0)
B = 1	C = 1	while (C == 0)	while (C==0)
		print A	print A

- Write serialization

  If updates of A by P1 and P2 reach processors P3 and P4 in different orders then SC will be violated

Write Atomicity Condition 2

P1	P2	P3
A = 1;	while (A == 0);	while (B == 0);
	B = 1;	print A;

Violates SC if

1) P2 reads A before update of A reaches P3
2) Update of B reached P3 before update of A
3) P3 reads the new value of B and then proceeds to read the value of A before it gets P1's update of A

Is it possible in a snooping cache coherence system?

SC Performance Problems

```
LD R1, A
LD R2, B
ST C, R3
ST D, R4
LD R5, D
```

- Bypassing through write buffer not allowed
- Two loads cannot be issued to cache
  - Non-blocking caches not allowed
- No compiler optimizations

Relaxed Consistency Models

- Relax memory access ordering restrictions of SC
- Impose additional burden on programmer to ensure that the programs conform to the consistency model that the hardware provides
- Provides safety net for enforcing strict ordering between memory accesses in the form of fence instructions

Write-to-Read Program Order

- Hides latency of write operations
- When the write is in write buffer and not yet visible to other processor, the processor can issue and complete read hits and even single read miss
- Total Store Ordering (TSO) : relax W->R order
- Processor Consistency (PC) : relax W->R order

no write atomicity

Example 1

P1	P2
A = 1;	while (flag == 0);
Flag = 1;	print A

SC: A = 0 ×

TSO/PC: A = 0 ×

Example 2

P1	P2
A = 1;	print B;
B = 1;	print A;

SC: A = 0, B = 1 ×

TSO/PC: A = 0, B = 1 ×

Example 3

P1	P2	P3
A = 1;	while (A == 0);	while (B == 0);
	B = 1;	print A;

SC: A = 0 ×

TSO: A = 0 ×

PC: A = 0 √

Example 4

P1	P2
A = 1;	B = 1;
Print B;	print A;

SC: A = 0 B = 0 ×

TSO: A = 0 B = 0 √

PC: A = 0 B = 0 √

Satisfying SC under TSO/PC

- Ensure read does not complete before an earlier write in program order
  - MEMBAR or fence instruction before read
  - Prevents a successor read from issuing before all preceding writes have completed
- Ensure write atomicity of read operation (PC)
  - Replacing a read with read-modify-write

Write-to-Write Program Order

- Multiple write misses can be fully overlapped
- Writes can bypass earlier writes (to different locations) in write buffer
- Partial Store Ordering (PSO)
  - Relax W-> R order
  - Relax W-> W order
  - Guarantees write atomicity

Example 1

P1	P2
A = 1;	while (flag == 0);
Flag = 1;	print A

SC: A = 0 ×

TSO/PC: A = 0 ×

PSO: A = 0 √

Example 2

P1	P2
A = 1;	print B;
B = 1;	print A;

SC: A = 0, B = 1 ×

TSO/PC: A = 0, B = 1 ×

PSO: A = 0, B = 1 √

Example 3

P1	P2	P3
A = 1;	while (A == 0);	while (B == 0);
	B = 1;	print A;

SC: A = 0 ×

TSO: A = 0 ×

PC: A = 0 √

PSO: A = 0 ×

Weak Ordering (WO)

- Memory operations can be classified as
  - Data operations
  - Synchronization operations
- Reordering operations in data regions between synchronization operations does not affect the correctness of a program
- WO relaxes all program orders for non-synchronization memory operations and guarantees the synchronization of memory operations

Example

P1	P2
lock(flag);	lock(flag);
A = 1;	print B;
B = 1;	print A;
unlock(flag);	unlock(flag);

P1	P2
while (flag2 == 0);	while (flag1 == 0);
A = 1;	print B;
B = 1;	print A;
flag2 = 0;	flag1 = 0;
flag1 = 1;	flag2 = 1;

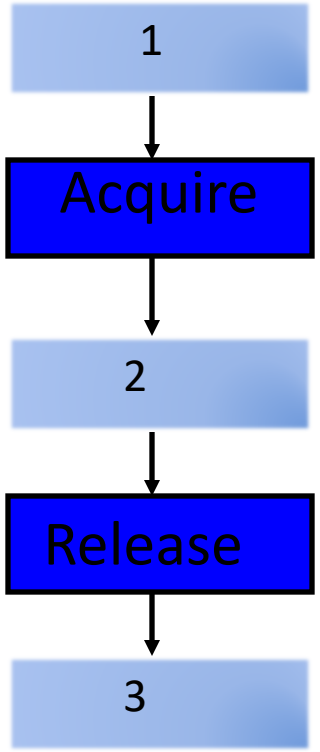
Sufficient Conditions for WO

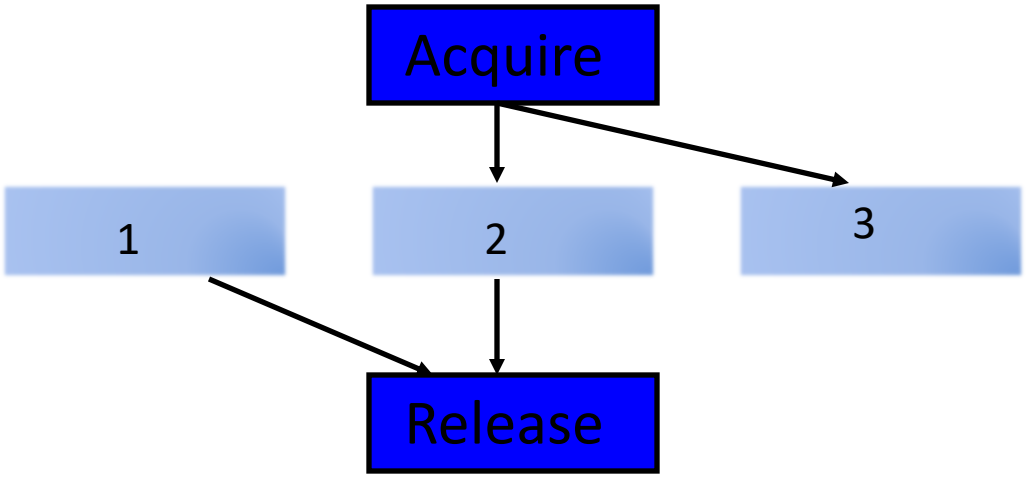
- Before a synchronization operation is issued, the processor waits for all previous memory operations in program order to complete
- Memory operations that follow the synchronization operation are not issued until the synchronization operation completes

Release Consistency (RC)

- Divides synchronization operation into
  - Acquire (e.g. lock)
  - Release (e.g. unlock)
- Acquire can be reordered with respect to previous operations
- Release can be reordered with respect to successor operations

WO versus RC





Java Memory Model (JMM)

- Consistency model at Multithreaded Java Programmer's interface
- Why JMM ?
  - Java bytecode should be portable
  - Unfortunately each multiprocessor has its own memory consistency model
  - Result of Java program will depend on underlying memory consistency model
  - JMM makes Java code portable across platforms

JMM Implications

- Programmer:
  - Should understand JMM to provide constraints in the program (via synchronization)
- Compiler:
  - Should only allow reordering allowed by JMM
- JVM:
  - Should provide JMM at the interface by taking into consideration the underlying memory model
  - JMM more relaxed than H/W memory model
    - JVM easy to implement
  - JMM less relaxed than H/W memory model
    - JVM should have lots of fences to implement JMM