Memory Model

COS 597C
10/5/2010

Example

```
                    a = Flag = 0

        Thread

a = 26;

Flag = 1;
```

Example

```
a = Flag = 0

Thread

a = 26;

Flag = 1;
```

Compiler Transformation

```
Flag = 1;

a = 26;
```

Example

```
a = Flag = 0
```

Thread 1

```
a = 26;

Flag = 1;
```

Thread 2

```
while (Flag != 1)
  {};

b = a;
```

What is the value of b after execution?

Example

```
a = Flag = 0
```

Thread 1

```
a = 26;

Flag = 1;
```

Thread 2

```
while (Flag != 1)
  {};

b = a;
```

26 ?

What is the value of b after execution?

Example

```
a = Flag = 0
```

Thread 1

```
a = 26;

Flag = 1;
```

Thread 2

```
while (Flag != 1)
  {};

b = a;
```

0 !!

What is the value of b after execution?

How could this happen?

- Compilers can reorder instructions

```
a = Flag = 0
```

Thread 1

```
a = 26;

Flag = 1;
```

Thread 2

```
while (Flag != 1)
  {};

b = a;
```

How could this happen?

- Compilers can reorder instructions

```
                a = Flag = 0

      Thread 1                    Thread 2

Flag = 1;        (1)      while (Flag != 1)  (2)
                            {};

a = 26;          (4)

                          b = a;   0          (3)
```

How could this happen?

- Lets disable compiler reordering. How about now?

```
                    a = Flag = 0

      Thread 1                          Thread 2

a = 26;                          while (Flag != 1)
                                   {};

Flag = 1;

                                 b = a;
```

How could this happen?

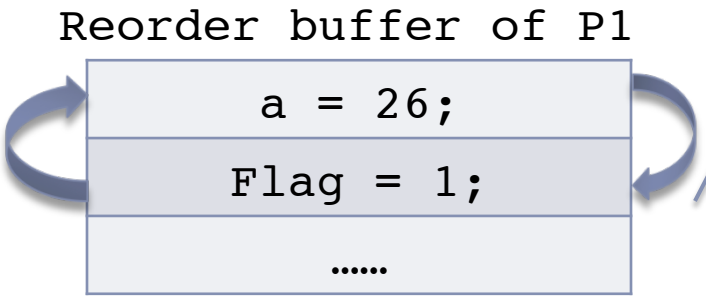
- Lets disable compiler reordering. How about now?

```
a = Flag = 0

Thread 1                    Thread 2

a = 26;                     while (Flag != 1)
                              {};
Flag = 1;
                            b = a;    0 !!
```

How could this happen?

- Hardware out-of-order execution

```
a = Flag = 0

Thread 1                    Thread 2

a = 26;                     while (Flag != 1)
                              {};

Flag = 1;                   b = a;    0 !!
```



How could this happen?

- Hardware out-of-order execution

```
a = Flag = 0

Thread 1                    Thread 2

a = 26;                     while (Flag != 1)
                              {};

Flag = 1;                   b = a;    0 !!
```

Reorder buffer of P1

Flag = 1;
a = 26;
……

Things could go crazy .....

If we don't define what is a valid optimization

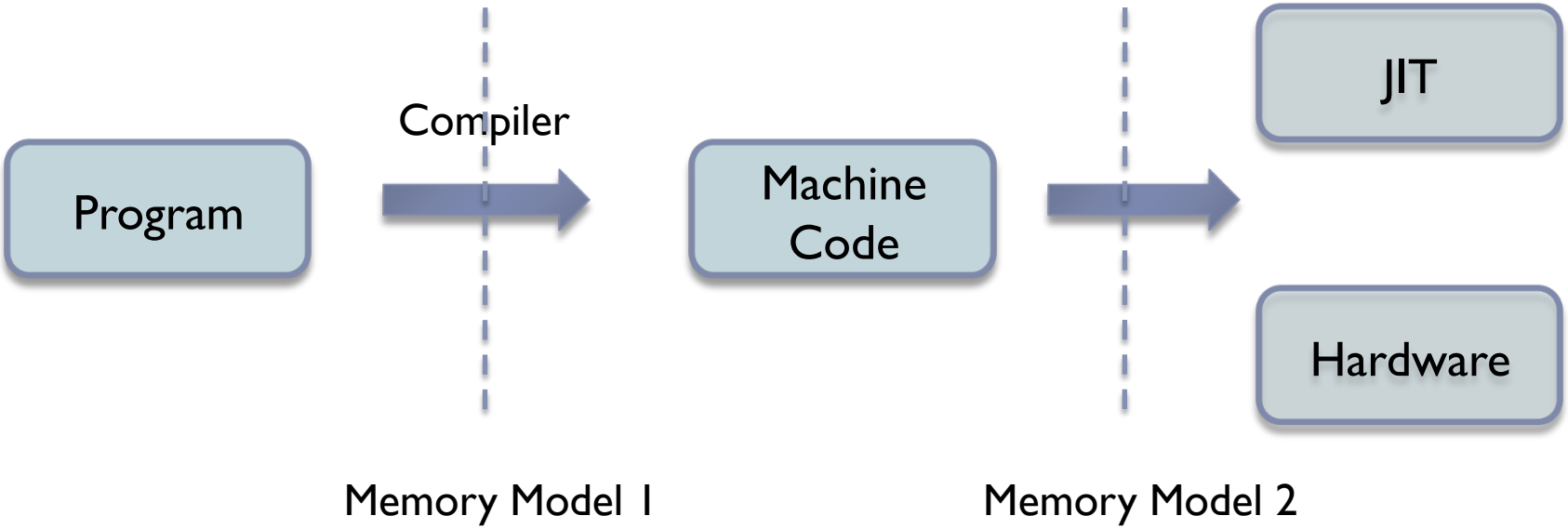
What is Memory (Consistency) Model?

- "A formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system." [Adve' 1995]

- Memory model specifies:
  - How threads interact through memory
  - What value a read can return
  - When does a value update become visible to other threads
  - What assumptions are allowed to make about memory when writing a program or applying some program optimization

Why do We Care?

- Memory model affects:
  - Programmability
  - Performance
  - Portability



The Single Thread Model

- Memory access executes one-at-a-time in program order
- Read returns value of last write
- For hardware & compiler reordering
  - Optimization must respect data/control dependences
  - Memory operations must follow the order the program is written
- Easy to program and optimize

Strict Consistency Model

- Any read to memory location X returns the value stored by the latest write to X

Timeline

Thread 1	Thread 2

`X = 1;`
……
	……
	`R1 = X;`
	`R2 = X;`

R1	1
R2	1
X	1

Strict Consistency Model

- Any read to memory location X returns the value stored by the latest write to X

Timeline

Thread 1	Thread 2
X = 1; ……	R1 = X; …… R2 = X;

R1	0
R2	1
X	1

Strict Consistency Model

- Any read to memory location X returns the value stored by the latest write to X

Timeline

Thread 1	Thread 2

`X = 1;`
……
	……
	`R1 = X;`
	`R2 = X;`

R1	0
R2	1
X	1

Sequential Consistency

- Definition: [Lamport' 1979]
  the result of any execution is the same as:
  - The operations of each thread appears in program order
  - Operations of all threads were executed in some sequential order atomically
- Atomicity
  - Isolation : no one sees partial memory update
  - Serialization : memory access appear to occur at the same time for everyone

Under Sequential Consistency Model

- The operations of each thread appears in program order
- Operations of all threads were executed in some sequential order atomically

Timeline

Thread 1	Thread 2
X = 1; ……	…… R1 = X; R2 = X;

R1	0
R2	1
X	1

Under Sequential Consistency Model

- The operations of each thread appears in program order
- Operations of all threads were executed in some sequential order atomically

Timeline

Thread 1	Thread 2
X = 1; ……	…… R1 = X; R2 = X;

R1	1
R2	0
X	1

Example

- Dekker's algorithm for critical sections

```
Flag1 = Flag2 = 0;
```

Thread 1	Thread 2

`Flag1 = 1;`
 	
`if (Flag2 == 0)`
`  critical`
	`Flag2 = 1;`
	
	`if (Flag1 == 0)`
	`  critical`

Example

- Dekker's algorithm for critical sections

```
Flag1 = Flag2 = 0;
```

Thread 1	Thread 2

`Flag1 = 1;`
 	
`if (Flag2 == 0)`
`  critical`
	`Flag2 = 1;`
	
	`if (Flag1 == 0)`
	`  critical`

Flags1	1
Flags2	0

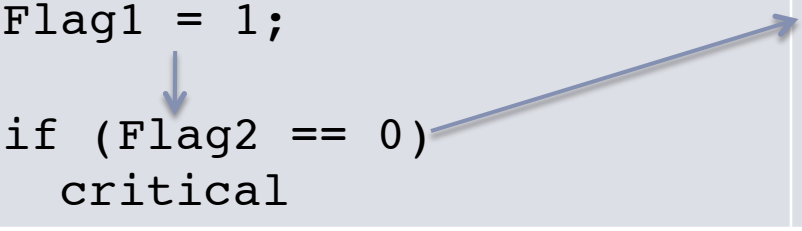
Example

- Dekker's algorithm for critical sections

```
Flag1 = Flag2 = 0;
```

Thread 1	Thread 2

`Flag1 = 1;`
`if (Flag2 == 0)`
`  critical`
	`Flag2 = 1;`
	`if (Flag1 == 0)`
	`  critical`



Flags1	1
Flags2	1

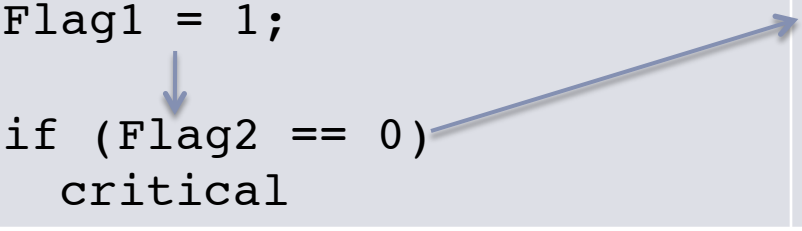
Example

- Dekker's algorithm for critical sections

```
Flag1 = Flag2 = 0;
```

Thread 1	Thread 2

`Flag1 = 1;`
`if (Flag2 == 0)`
`  critical`
	`Flag2 = 1;`
	`if (Flag1 == 0)`
	`  critical`



Flags1	0
Flags2	1

Violation !!!

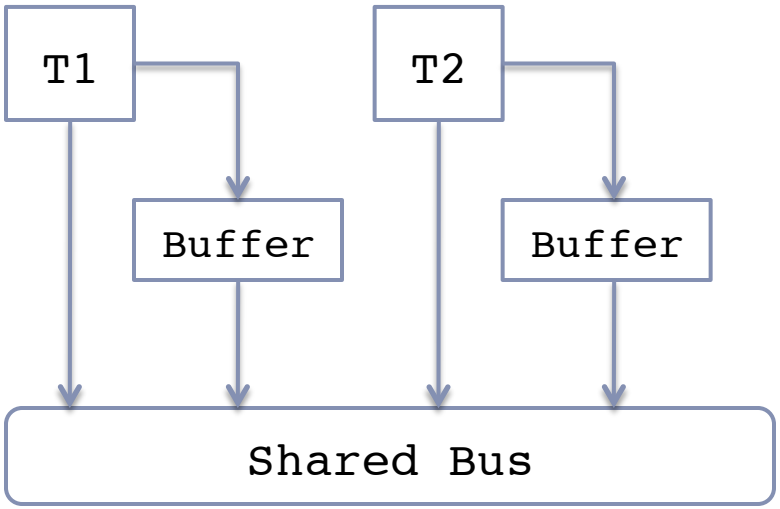
How do we violate sequential consistency?

Very EASY !

Lets take a look at several hardware/ compiler optimizations that are commonly used for uniprocessor

Violation of SC: Architecture without Caches

- Write buffers with bypassing

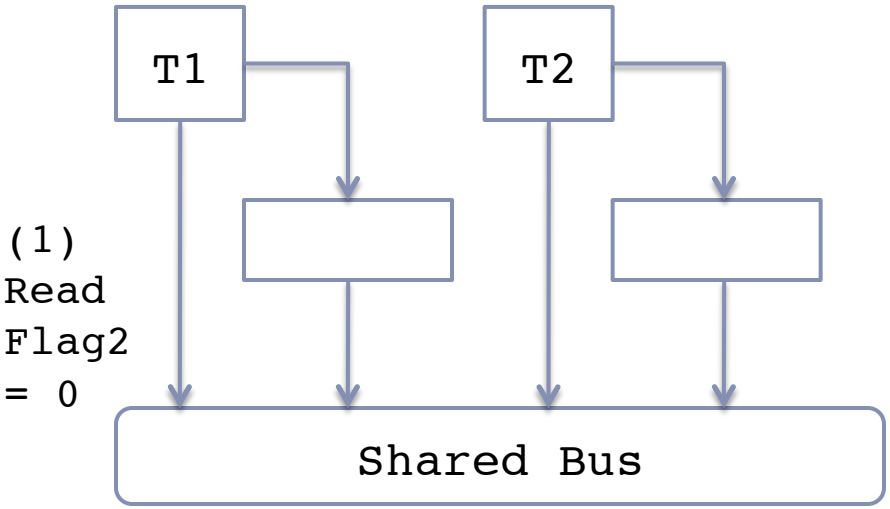



```
Thread 1                Thread 2

Flag1 = 1;              Flag2 = 1;
if (Flag2 ==0)          if (Flag1 ==0)
  critical                critical
```

Flag1	0
Flag2	0

Violation of SC: Architecture without Caches

- Write buffers with bypassing

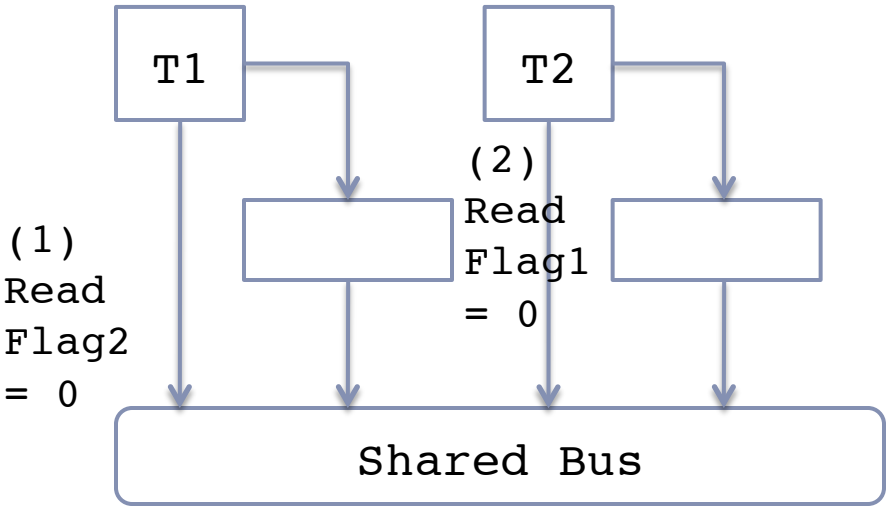



```
Thread 1                Thread 2

Flag1 = 1;              Flag2 = 1;
if (Flag2 ==0)          if (Flag1 ==0)
  critical                critical
```

Flag1	0
Flag2	0

Violation of SC: Architecture without Caches

- Write buffers with bypassing

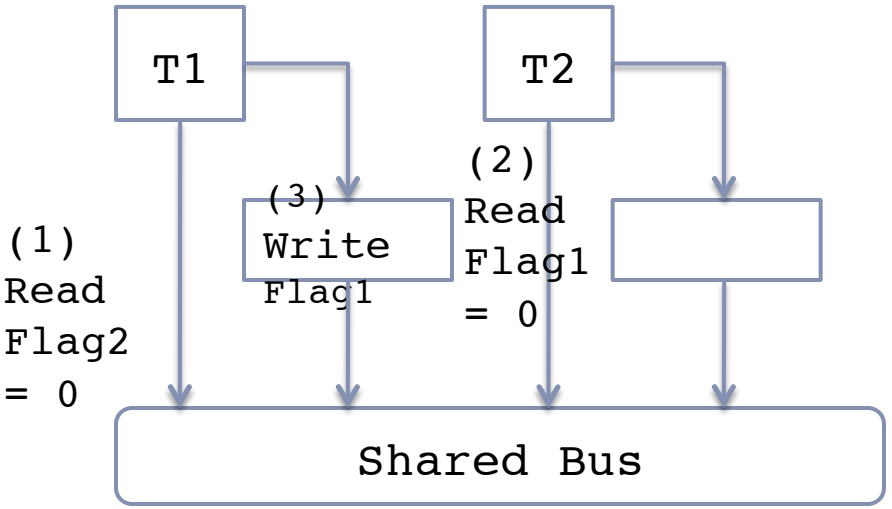



```
        Thread 1                Thread 2

Flag1 = 1;              Flag2 = 1;
if (Flag2 ==0)          if (Flag1 ==0)
  critical                critical
```

Flag1	0
Flag2	0

Violation of SC: Architecture without Caches

- Write buffers with bypassing

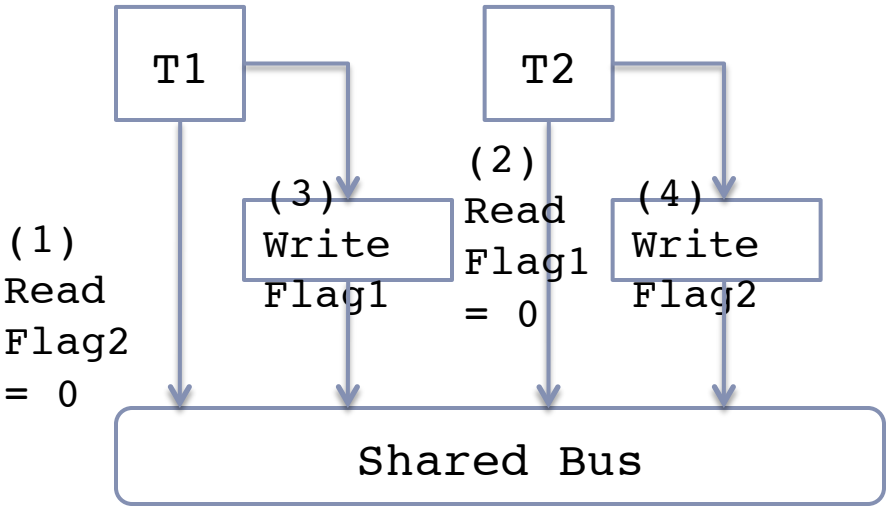



```
Thread 1                Thread 2

Flag1 = 1;              Flag2 = 1;
if (Flag2 ==0)          if (Flag1 ==0)
  critical                critical
```

Flag1	0
Flag2	0

Violation of SC: Architecture without Caches

- Write buffers with bypassing



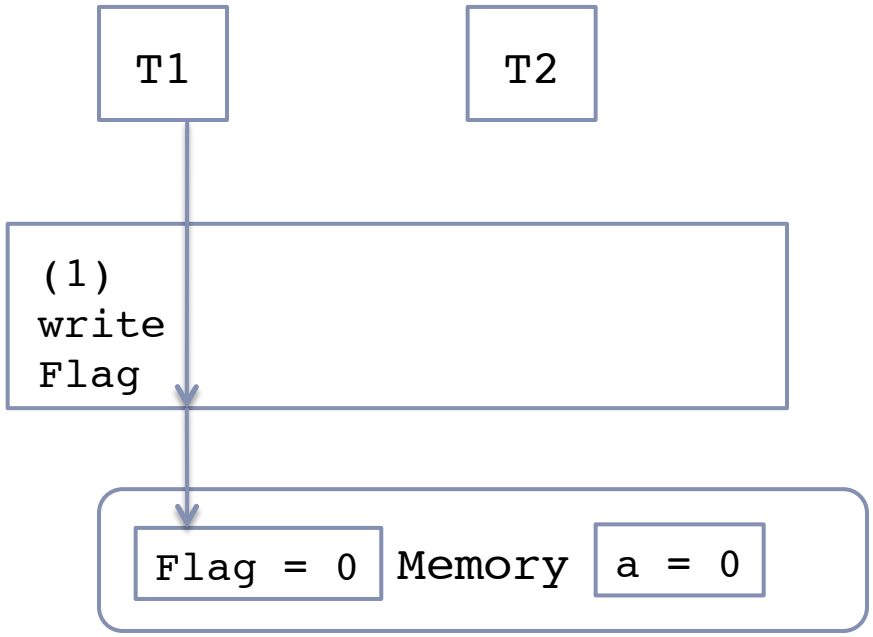

```
Thread 1                Thread 2

Flag1 = 1;              Flag2 = 1;
if (Flag2 ==0)          if (Flag1 ==0)
  critical                critical
```

Flag1	0
Flag2	0

Violation of SC: Architecture without Caches

- Overlapping writes

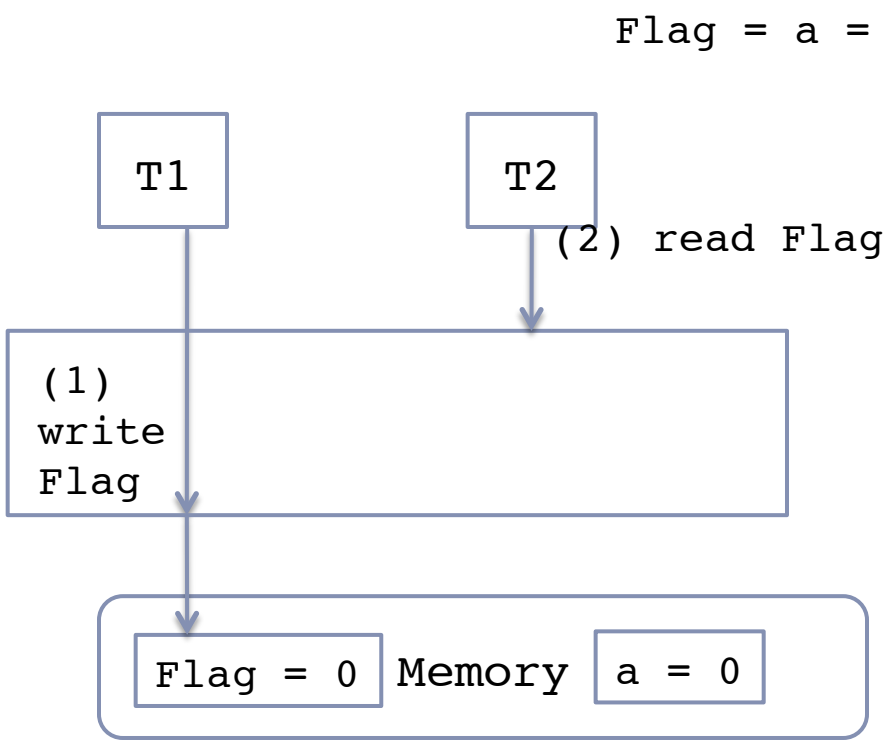
```
Flag = a = 0;
```




```
Thread 1            Thread 2

a = 26;             while (Flag == 0)
                        {};
Flag= 1;
                    b = a;
```

Violation of SC: Architecture without Caches

- Overlapping writes

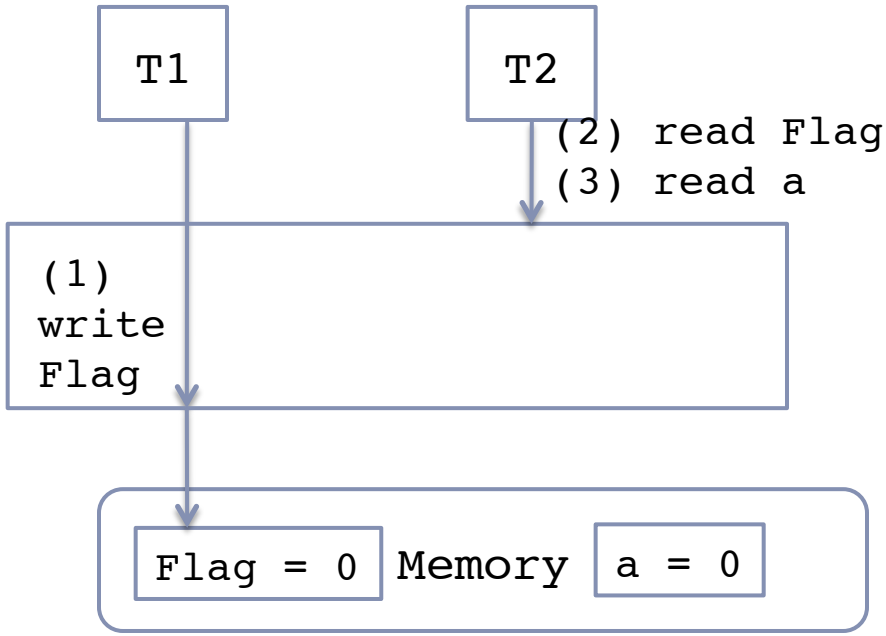



```
Thread 1          Thread 2

a = 26;           while (Flag == 0)
                      {};
Flag= 1;
                  b = a;
```

Violation of SC: Architecture without Caches

- Overlapping writes

```
Flag = a = 0;
```

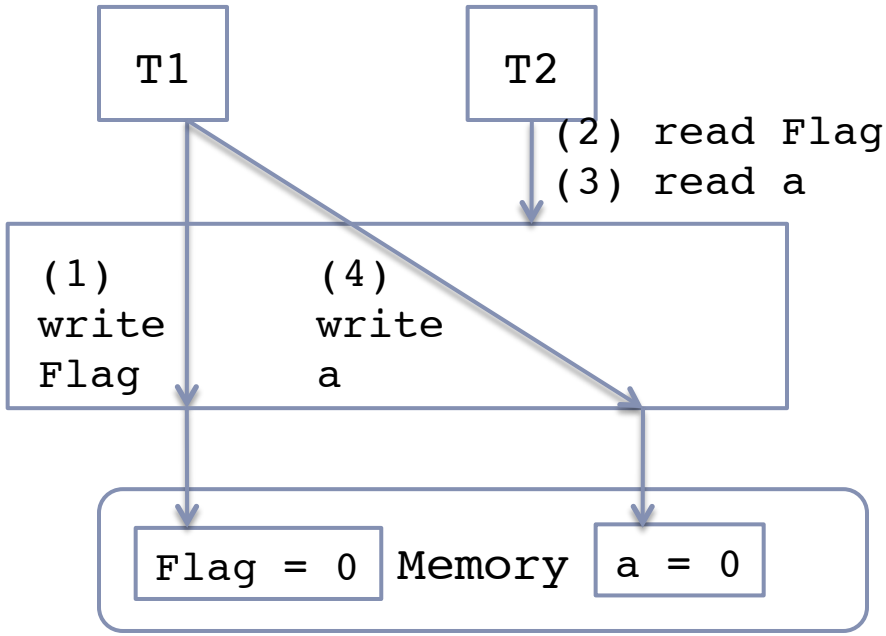



```
Thread 1                Thread 2

a = 26;                 while (Flag == 0)
                            {};
Flag= 1;
                        b = a;
```

Violation of SC: Architecture without Caches

- Overlapping writes

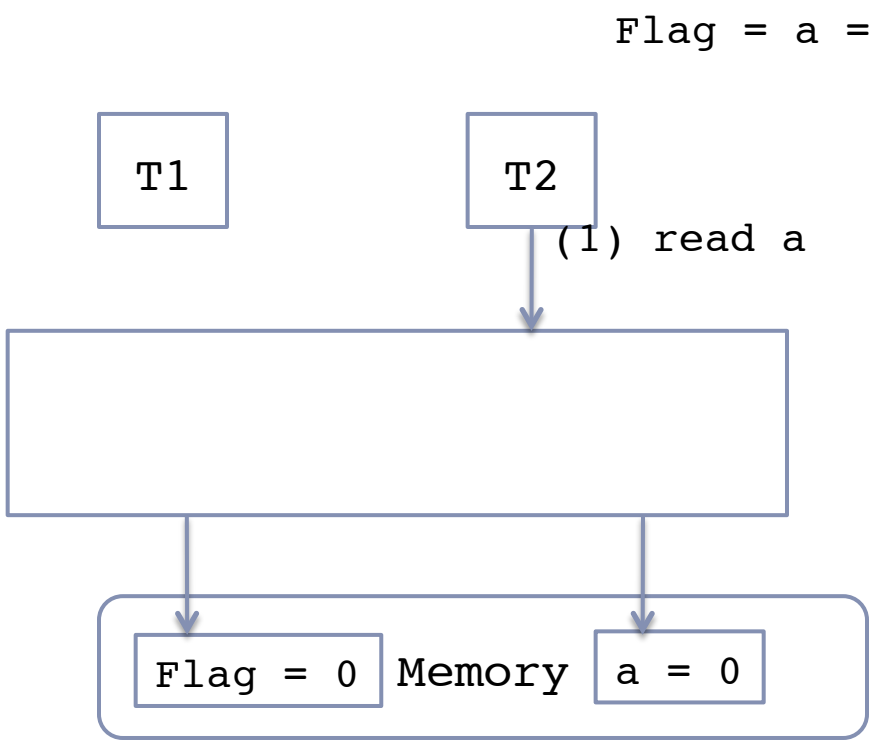
```
Flag = a = 0;
```




```
Thread 1                Thread 2

a = 26;                 while (Flag == 0)
                            {};
Flag= 1;
                        b = a;
```

Violation of SC: Architecture without Caches

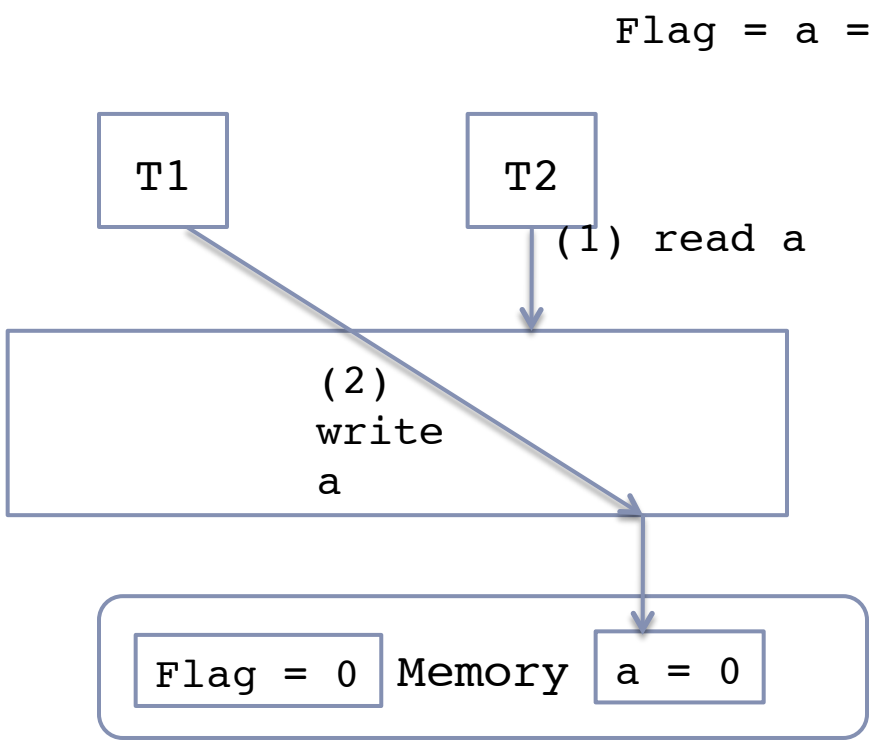
- Non-blocking reads




```
Thread 1          Thread 2

a = 26;           while (Flag == 0)
                      {};
Flag= 1;
                  b = a;
```

Violation of SC: Architecture without Caches

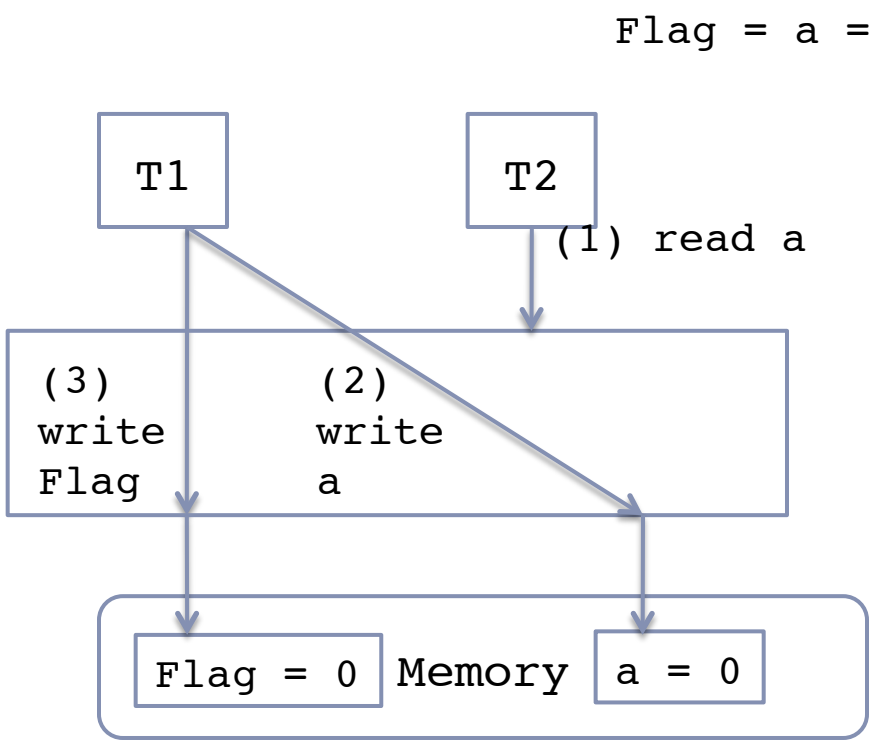
- Non-blocking reads




```
Thread 1          Thread 2

a = 26;           while (Flag == 0)
                      {};
Flag= 1;
                  b = a;
```

Violation of SC: Architecture without Caches

- Non-blocking reads

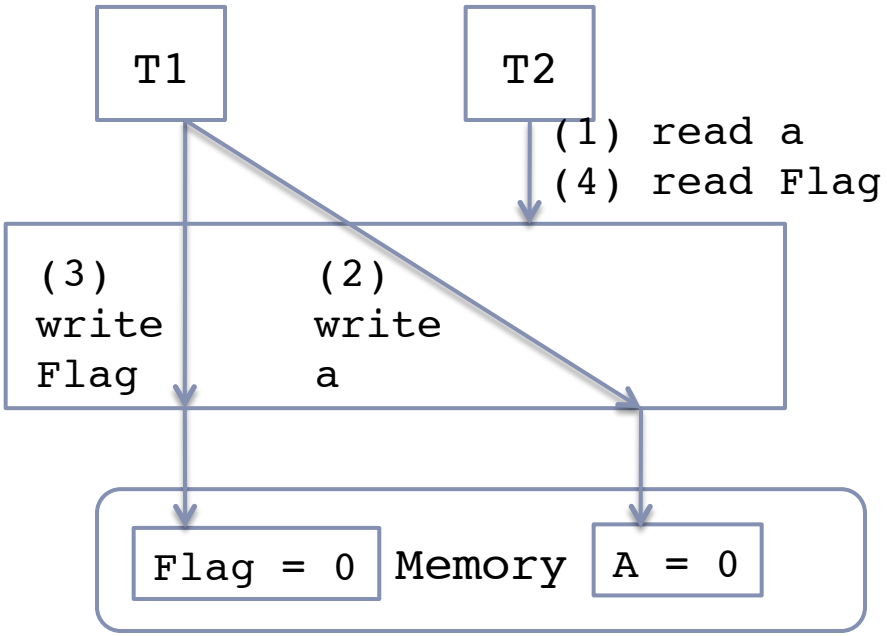



```
Thread 1                Thread 2

a = 26;                 while (Flag == 0)
                            {};
Flag= 1;
                        b = a;
```

Violation of SC: Architecture without Caches

- Non-blocking reads

```
Flag = a = 0;
```




```
Thread 1                Thread 2

a = 26;                 while (Flag == 0)
                            {};
Flag= 1;
                        b = a;
```

Architecture with Private Caches

To comply with Sequential Consistency, we need:

- Cache coherency protocol
  - A write is eventually made visible to all processors
  - Writes to the same location appear to be seen in the same order by all processors (***serialization***) [Gharachorloo'90]
- Ability to detect the completion of write operations
  - Acknowledgement messages
  - Invalid or update messages
- The illusion of atomic writes

Write atomicity

```
A = B = C = 0;
```

Thread 1	Thread 2	Thread 3	Thread 4

A = 1;
B = 1;
	A = 2;
	C = 1;
		while (B!=1) {};
		while (C!=1) {};
		R1 = A;
			while (B!=1) {};
			while (C!=1) {};
			R2 = A;

What is the value of R1 and R2 after execution?

Write Atomicity

```
A = B = C = 0;
```

Thread 1	Thread 2	Thread 3	Thread 4

`A = 1;`
`B = 1;`
	`A = 2;`
	`C = 1;`
		`while (B!=1) {};`
		`while (C!=1) {};`
		`R1 = A;`
			`while (B!=1) {};`
			`while (C!=1) {};`
			`R2 = A;`

R1 = 1 **R2 = 1**

Write Atomicity

```
A = B = C = 0;
```

Thread 1	Thread 2	Thread 3	Thread 4

`A = 1;`
`B = 1;`
	`A = 2;`
	`C = 1;`
		`while (B!=1) {};`
		`while (C!=1) {};`
		`R1 = A;`
			`while (B!=1) {};`
			`while (C!=1) {};`
			`R2 = A;`

R1 = 2 **R2 = 2**

Write Atomicity

```
A = B = C = 0;
```

Thread 1	Thread 2	Thread 3	Thread 4

A = 1;
B = 1;
	A = 2;
	C = 1;
		while (B!=1) {};
		while (C!=1) {};
		R1 = A;
			while (B!=1) {};
			while (C!=1) {};
			R2 = A;

R1 = 1 **R2 = 2**

Violation !!!

Sequential Consistency:
operation from all threads must appear
In some sequential order atomically

Compilers Optimization that Violates SC

- Compiler reordering must respect data and control dependencies
- Code motion

```
      Thread 1                Thread 2

for(i=0;i<10;i++)     while (true)
  *a = i;               b = *a;
```

Load from a cannot be moved out of the loop

Compilers Optimization that Violates SC

- Compiler reordering must respect data and control dependencies
- Code motion
- Common sub-expression elimination

```
    Thread 1                Thread 2

                    c = a – 1;
a = 6;
                    while (Flag == 0) {};
Flag = 1;
                    b = a - 1;
```

(a-1) cannot be eliminated for assignment of b

Compilers Optimization that Violates SC

- Compiler reordering must respect data and control dependencies
- Code motion
- Common sub-expression elimination
- Register allocation

```
Thread 1                Thread 2

a = 6;                  while (Flag == 0) {};

Flag = 1;               b = a;
```

Flag cannot be allocated to a register

Sequential Consistency: Summary

- Sequential consistency does not guarantee data race free

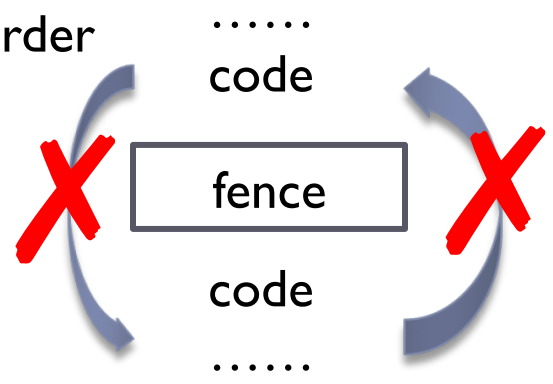
Thread 1	Thread 2
`A = 1;`	`A = 2;`
`B = 1;`	`C = 1;`

Data race:
- two memory access to the same location
- one is a write
- they can occur simultaneously

- Possible hardware/compiler optimizations allowed
  - Hardware/software prefetching
  - Speculating read values
- Determining which instructions are allowed to be reordered remain an open question

Relaxed Memory Models

- Key points
  - Program order for different memory addresses
  - Write atomicity
- Possible relaxations
  - Relaxation on program order (different memory locations)
    - Relax *write to read* program order
    - Relax *write to write* program order
    - Relax *read to read* and *read to write* program order
  - Relaxation on write atomicity
    - Read other's write early
    - Read own write early
- Safety nets, such as *fence*



Major Relaxed Hardware Models

Relax	W->R	W->W	R->RW	Read others' write early	Read own write early	Safety Net
SC					✔	
IBM 370	✔					Serial inst
TSO(x86)	✔				✔	RMW, fence
PC	✔			✔	✔	RMW
PSO	✔	✔			✔	RMW
WO	✔	✔	✔		✔	synch
RCsc	✔	✔	✔		✔	lock, nsync, RMW
RCpc	✔	✔	✔	✔	✔	
Alpha	✔	✔	✔		✔	MB, WMB
RMO	✔	✔	✔		✔	MEMBAR
PowerPC	✔	✔	✔	✔	✔	synch

Processor Consistency

- Writes done by a single processor are received by other processors in the same order as they are issued.
- Writes from different processors may be seen in different order by different processors.

```
A = B = C = 0;
```

Thread 1	Thread 2	Thread 3	Thread 4

A = 1;
B = 1;
	A = 2;
	C = 1;
		while (B!=1) {};
		while (C!=1) {};
		R1 = A;
			while (B!=1) {};
			while (C!=1) {};
			R2 = A;

R1 = 1 R2 = 2

Weak Ordering Model [Dubois' 86]

- Classification of memory operations
  - Data operations: load, store…
  - Synchronization operations: lock unlock etc
- How does it work ?
  - All pre-issued operations must complete on all processors before executing a synchronization operation
  - Execution of synchronization operations must follow program order
  - Memory operations between synchronization operations can be reordered

Data-Race-Free-0 Model

- A program is data-race-free on a particular input if no sequential consistent execution results in a data race
- A new definition of weak ordering [Adve'90 ISCA]
- Advantage:
  - Simple programmability of sequential consistency
  - Implementation flexibility of relaxed models

- Sequential consistency for DRF is widely used
  - C++ memory model

Relaxed Memory Model: Summary

- Relaxed memory model
  - Relaxes restrains on the order of some memory operations
  - Allows some hardware/compiler optimization
- Why do we use relaxed memory model : performance
- Why do we not use relaxed memory model : complexity

CASE STUDY: The C++ Memory Model

- Adaption of DRF0
  - Sequential consistency for race-free programs
  - Behavior of a program with data race is undefined (no benign data races in C++)
- Data operations:

  load, store
- Synchronization operations:

  lock, unlock, atomic load, atomic store, atomic read-modify-write
- Atomic operations must appear sequentially consistent

[Boehm's PLDI 2008: Foundations of the C++ Concurrency Memory Model]

CASE STUDY: The C++ Memory Model

- Compiler code reordering allowed when:

  For memory operations M1 and M2

  - M1 is a data operation and M2 is a read synchronization operation
  - M1 is write synchronization and M2 is data
  - M1 and M2 are both data with no synchronization sequence-ordered between them.
  - M1 is data and M2 is the write of a lock operation
  - M1 is unlock and M2 is either a read or write of a lock.
- Hardware optimization allowed for non-atomic writes

CASE STUDY: The C++ Memory Model

- Semantic of trylock

```
      Thread 1                        Thread 2

X = 42;                       while (trylock(l) == success)
                                  unlock(l);
lock(l);                      assert( X==42 );
```

Can the program assert?

CASE STUDY: The C++ Memory Model

- Semantic of trylock

```
Thread 1                    Thread 2

X = 42;                     while (trylock(l) == success)
                                unlock(l);
lock(l);                    assert( X==42 );
```

Can the program assert?

CASE STUDY: The C++ Memory Model

- Semantic of trylock

```
Thread 1                    Thread 2

lock(l);                    while (trylock(l) == success)
                                unlock(l);
X = 42;                     assert( X==42 );
```

Yes, if the compiler reorders code in T1

CASE STUDY: The C++ Memory Model

- Semantic of trylock

```
      Thread 1                        Thread 2

X = 42;                     while (trylock(l) == success)
                                unlock(l);
lock(l);                    assert( X==42 );
```

We can use a fence, but it is unfair for properly used trylock

CASE STUDY: The C++ Memory Model

- Semantic of trylock

```
      Thread 1                        Thread 2

X = 42;                    while (trylock(l) == success)
                               unlock(l);
lock(l);                   assert( X==42 );
```

```
Solution: in C++ memory model, trylock does not
guarantee to reveal anything about the state
of the lock
```

CASE STUDY: The JAVA Memory Model

- JAVA: the first language specification attempts to incorporate memory model
- What JAVA should do?
  - Define semantics of all programs
  - Support execution of untrusted "sandboxed" code
- Sequential consistency for DRF
- Synchronization implemented using monitors
  - Volatile
  - synchronized primitive
- JAVA memory model does not guarantee deadlock free

CASE STUDY: The JAVA Memory Model

- JAVE bugs found historically
  - Detached thread
  - Double-checked locking

```
Helper helper;

Helper getHelper() {
  if (helper==null) {
    synchronized(this) {
      if (help==null)
        helper=new Helper();
    }
  }
  return helper;
}
```

Lessons Learnt from C++/JAVA

- SC for DRF is the minimal baseline
- Specifying semantics for programs with data races is extremely HARD
- Simple optimization may introduce unintended consequences
- State-of-the-art is still broken
  - Abandon shared memory?
  - Hardware co-designed with high-level memory models?
  - Any volunteer for fixing the whole thing?

Conclusion

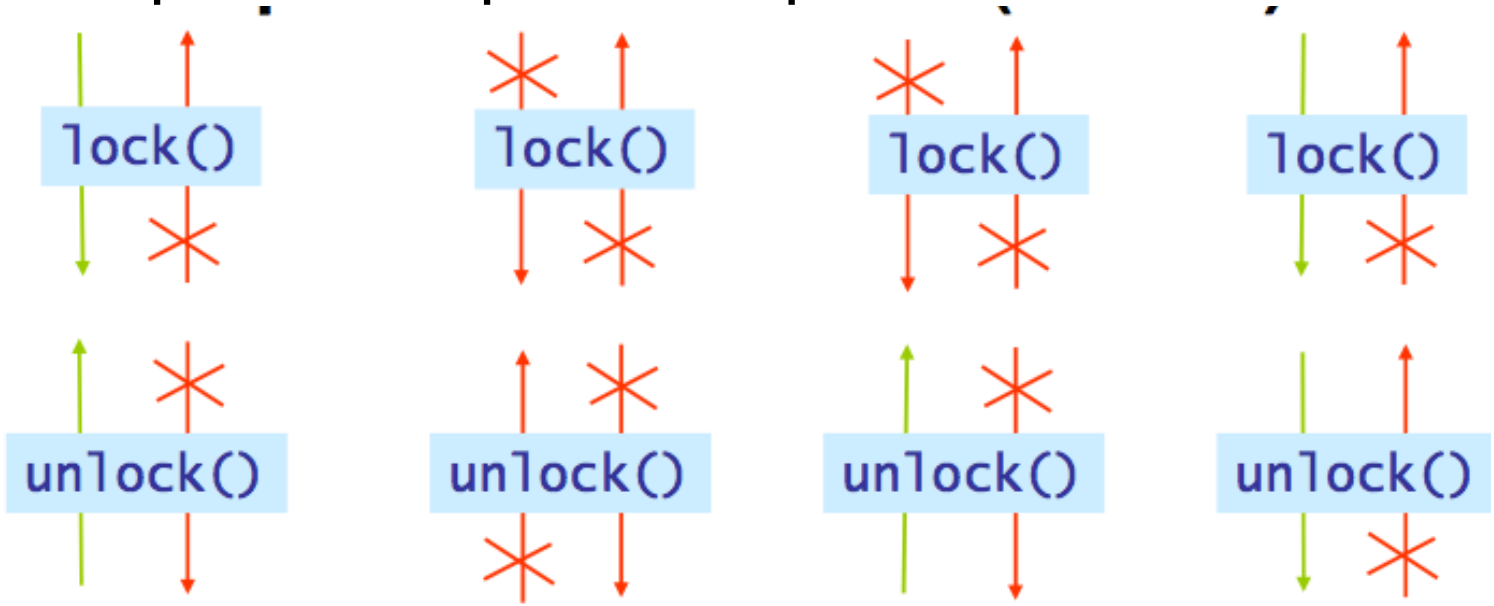
- Memory model is very important and confusing
- Memory model specifies what hardware/compiler can do and cannot do
- Sequential consistency is very intuitive yet prohibits performance
- Relaxed memory models allows some optimization but introduces programming complexity
- Don't try to be clever, unless you are clever enough

Advanced Topics

- Why threads cannot be implemented as a library
- Ongoing projects:
  - Deterministic Parallel JAVA (DPJ)
  - Functional languages
  - DeNoVo hardware project

Pthreads

- Some open source pthread implementation



[technically incorrect]	[Correct, slow]	[Correct]	[Incorrect]
NPTL	NPTL	NPTL	FreeBSD
{Alpha, PowerPC}	Itanium (&X86)	{ Itanium, X86 }	Itanium
{mutex, spin}	mutex	spin	spin

References

- Boehm's "Foundations of the C++ Concurrency Memory Model"
- Pugh's "Fixing the JAVA Memory Model"
- Adve's "Shared Memory Consistency Models: A Tutorial"
- Dubois' "Memory Access Buffering in Multiprocessors"
- Bohem's "Threads cannot be implemented as a library"