EECS 570

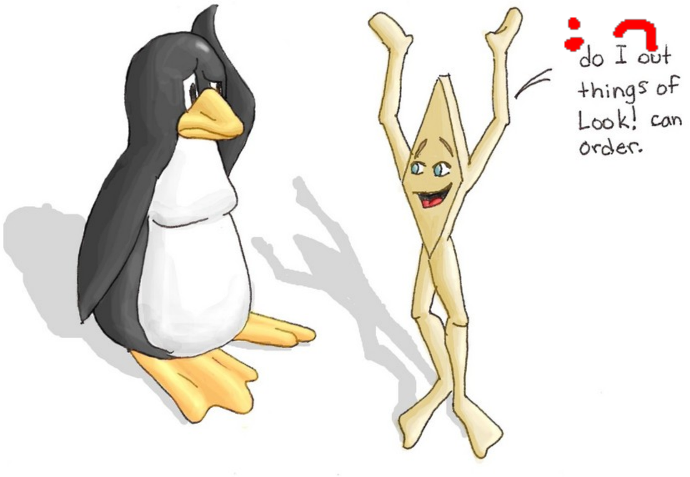
Lecture 12

Memory Consistency

Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/



Slides from Keshav Pingali, UT Austin

Outline

- Memory consistency problem
  - Reordering of instructions within a core may change semantics of parallel program
- Need for memory consistency models
  - architecture level
  - programming language level
- Sequential consistency model
- Relaxed memory models
  - weak consistency model
  - release consistency model
- Conclusions

Recall: uniprocessor execution

- Processors reorder operations to improve performance
- Constraint on reordering: must respect dependences
  - data dependences must be respected: in particular, loads/stores to a given memory address must be executed in program order
  - control dependences must be respected

Permitted memory-op reorderings

- Stores to different memory locations can be performed out of program order

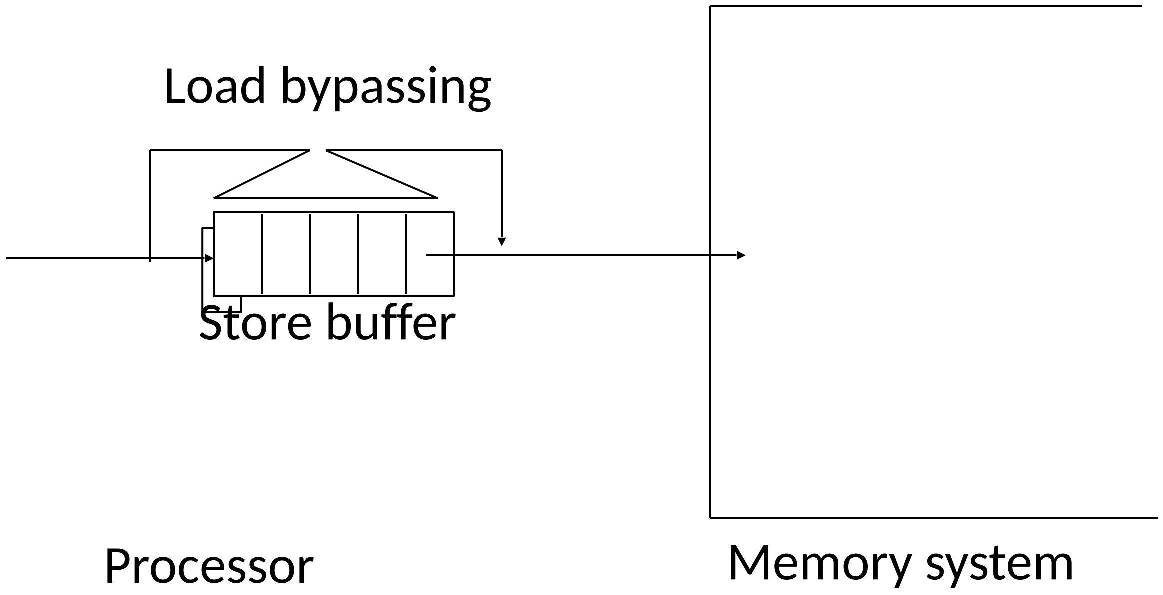
store v1, data   store b1, flag
store b1, flag   ←→   store v1, data

- Loads from different memory locations can be performed out of program order

load flag, r1   load data,r2
load data, r2   ←→   load flag,
r1

- Load and store to different memory locations can be performed out of program order by the hardware

Example of hardware reordering



- Store buffer holds store operations that need to be sent to memory
- Loads are higher priority operations than stores since their results are needed to keep processor busy, so they bypass the store buffer
  - load can bypass previous stores on its way to memory
- Load address is checked against addresses in store buffer, so store buffer satisfies load if there is an address match
  - load can return result before previous stores have completed

Key issue

- In single-threaded programs, reordering of instructions does not affect the output of the program
  - may improve performance but does not change the semantics
- In shared-memory programs, reordering of instructions executed by a thread may change the output of the program.
  - these usually occur in programs that use ordinary loads and stores instead of atomic operations to synchronize threads (data races)
- Two examples follow
  - Example (I): reordering of stores to different locations
  - Example (II): reordering of load and store to different locations

Example (I)

Code:

Initially A = Flag = 0

```
P1                         P2
A = 23;                    while (Flag != 1) ;
Flag = 1;                  … = A;
```

Idea:

- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.

Execution Sequence for (I)

Code:

Initially A = Flag = 0

```
P1                     P2
A = 23;                while (Flag != 1) {;}
Flag = 1;              ... = A;
```

Possible execution sequence on each processor:

```
P1                     P2
Write A 23                    Read Flag    //get 0
Write Flag 1                   ……
                              Read Flag    //get 1
                              Read A       //what do you get?
```

Problem: If the two writes on processor P1 can be reordered, it is possible for processor P2 to read 0 from variable A.

Can happen on most modern processors even with coherent caches

Intuitively, there is a connection between A and Flag that is not obvious if you just look at the program, which is all that compilers and processors can do.

Example II

Code: (like Dekker's algorithm)

Initially Flag1 = Flag2 = 0

```
P1                          P2
Flag1 = 1;                  Flag2 = 1;
If (Flag2 == 0)             If (Flag1 == 0)
  critical section            critical section
```

Possible execution sequence on each processor:

```
P1                          P2
Write Flag1, 1              Write Flag2, 1
Read Flag2  //get 0         Read Flag1  //what do you get?
```

Execution sequence for (II)

Code: (like Dekker's algorithm)

Initially Flag1 = Flag2 = 0

```
P1                               P2
Flag1 = 1;                  Flag2 = 1;
If (Flag2 == 0)              If (Flag1 == 0)
   critical section                 critical section
```

Possible execution sequence on each processor:

```
P1                               P2
Write Flag1, 1                   Write Flag2, 1
Read Flag2 //get 0               Read Flag1, ??
```

Most people would say that P2 will read 1 as the value of Flag1.

Since P1 reads 0 as the value of Flag2, P1's read of Flag2 must happen before P2 writes to Flag2. Intuitively, we would expect P1's write of Flag to happen before P2's read of Flag1.

However, this is true only if reads and writes on the same processor to different locations are not reordered by the compiler or the hardware.

Unfortunately, this is very common on most processors (store-buffers with load-bypassing).

Lessons

- Uniprocessors can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory context
  - parallel programs with data races may produce counter-intuitive results
- Question: what constraints must we put on uniprocessor instruction reordering so that
  - shared-memory programming is intuitive
  - but we do not lose uniprocessor performance?
- Many answers to this question
  - memory consistency model supported by the processor

Consistency models

- Consistency models are not about memory operations from different processors.
- Consistency models are not about dependent memory operations in a single processor’s instruction stream (these are respected even by processors that reorder instructions).
- Consistency models are all about ordering constraints on independent memory operations in a single processor’s instruction stream that have some high-level dependence (such as flags guarding data) that should be respected to obtain intuitively reasonable results.

Sequential Consistency

- Simple model for reasoning about parallel programs
- Meaning of parallel program: interleaving of instructions from threads
  - at each step, one thread is chosen for execution
  - one instruction is executed from that thread
- Note: different interleavings of instructions may produce different results but all are legal executions
- You can verify that the programs we considered before execute as expected under these semantics

Example:

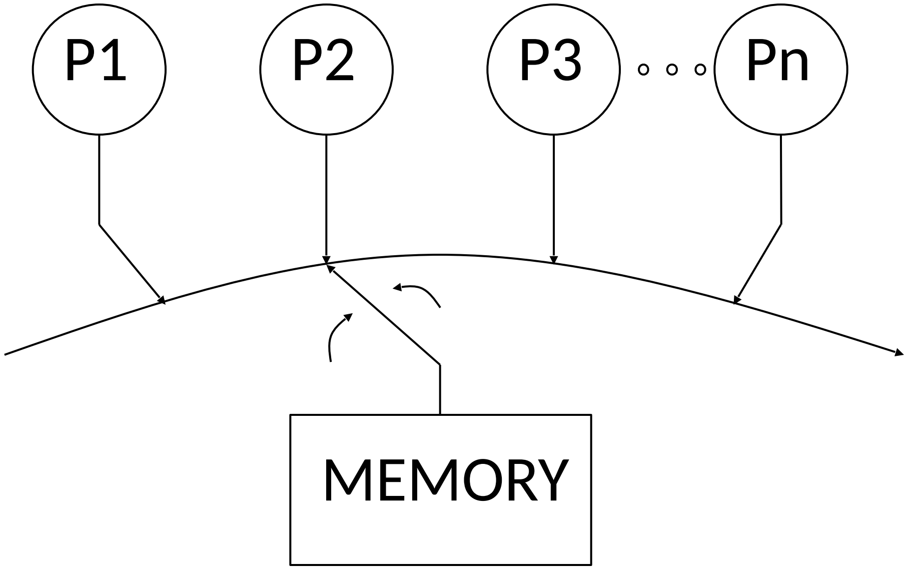
Initially A = Flag = 0

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

Sequential consistency

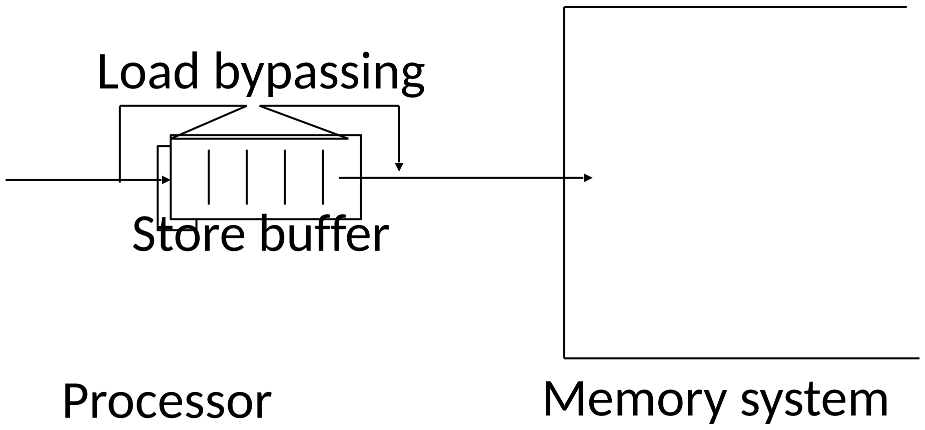
Equivalent to this model:

- processor does not reorder its own loads and stores to global memory
- loads and stores from different processors are sent to global memory in some interleaved order (but what about caching?)



Sequential Consistency

- Systems with coherent caches:
  - SC execution if processor does not reorder loads and stores to global memory
- Examples of forbidden behavior:
  - load by-passing with store buffers
  - load satisfied by store buffer before store has become visible globally (i.e., before line has been invalidated from other caches)



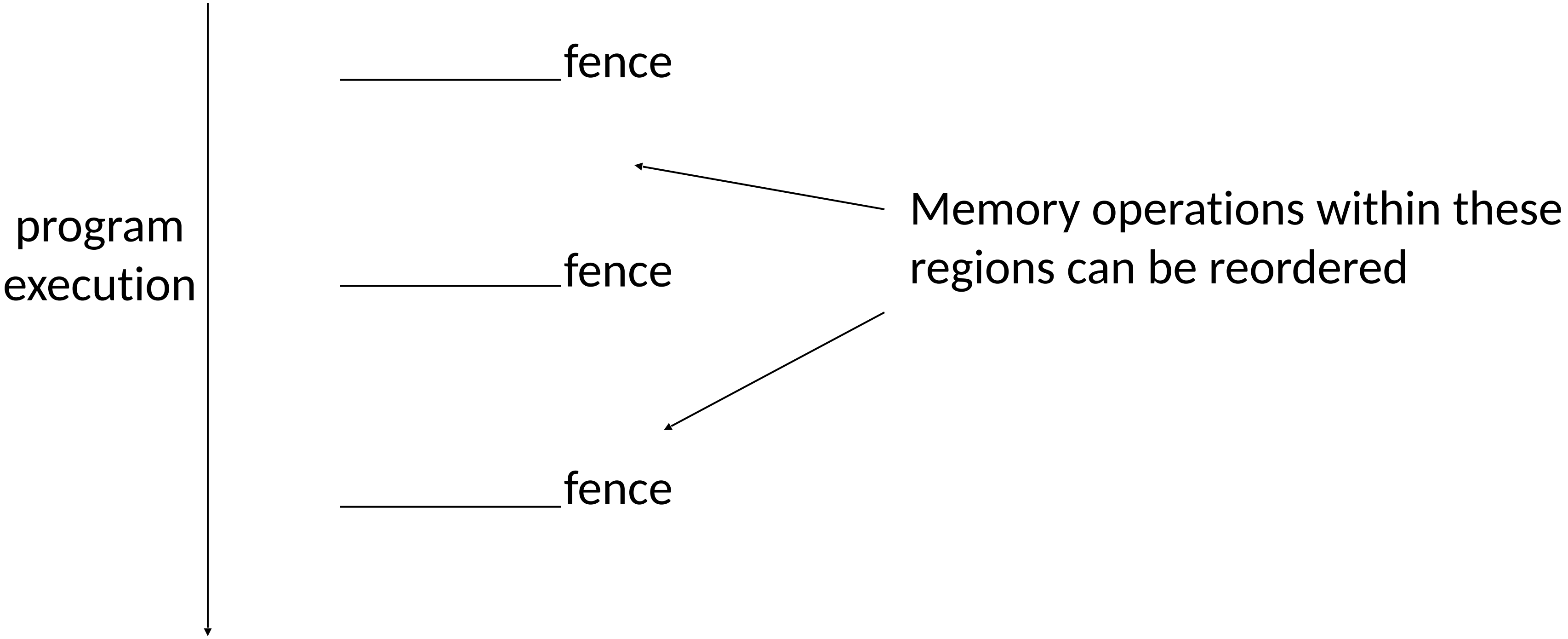
Problem

- Sequential consistency provides a simple model for reasoning about parallel programs
- However, it disallows use of features like store buffers that are used to speed up uniprocessor programs
- Key issue:
  - sequential consistency assumes every global memory operation might be involved in inter-thread synchronization
  - this is usually not the case
    - (e.g.) once you enter a critical section, you may do a lot of operations on global data structures
  - Unfortunately, all global memory operations are slowed down
- Solution: ask the programmer

Relaxed consistency model: Weak consistency

- Programmer specifies regions within which global memory operations can be reordered
- Processor has fence instruction:
  - all data operations before fence in program order must complete before fence is executed
  - all data operations after fence in program order must wait for fence to complete
  - fences are performed in program order
  - atomic instructions are treated as fences
- Implementation of fence:
  - processor has counter that is incremented when data op is issued, and decremented when data op is completed
- Example: PowerPC has SYNC instruction
- Language constructs:
  - OpenMP: flush
  - All synchronization operations like lock and unlock act like a fence

Weak ordering picture



program execution

fence

fence

fence

Memory operations within these regions can be reordered

Example (I) revisited

Code:

Initially A = Flag = 0

```
P1                          P2
A = 23;
fence;                      while (Flag != 1) ;
Flag = 1;                   fence;
                            … = A;
```

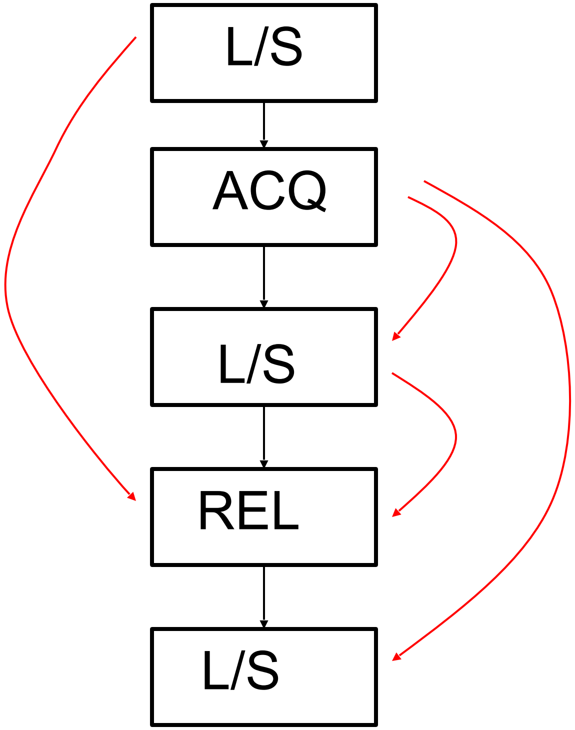
Execution:

- P1 writes data into A
- Fence waits till write to A is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of A even if memory operations in P1 before fence and memory operations after fence are reordered by the hardware or compiler.

Another relaxed model: release consistency

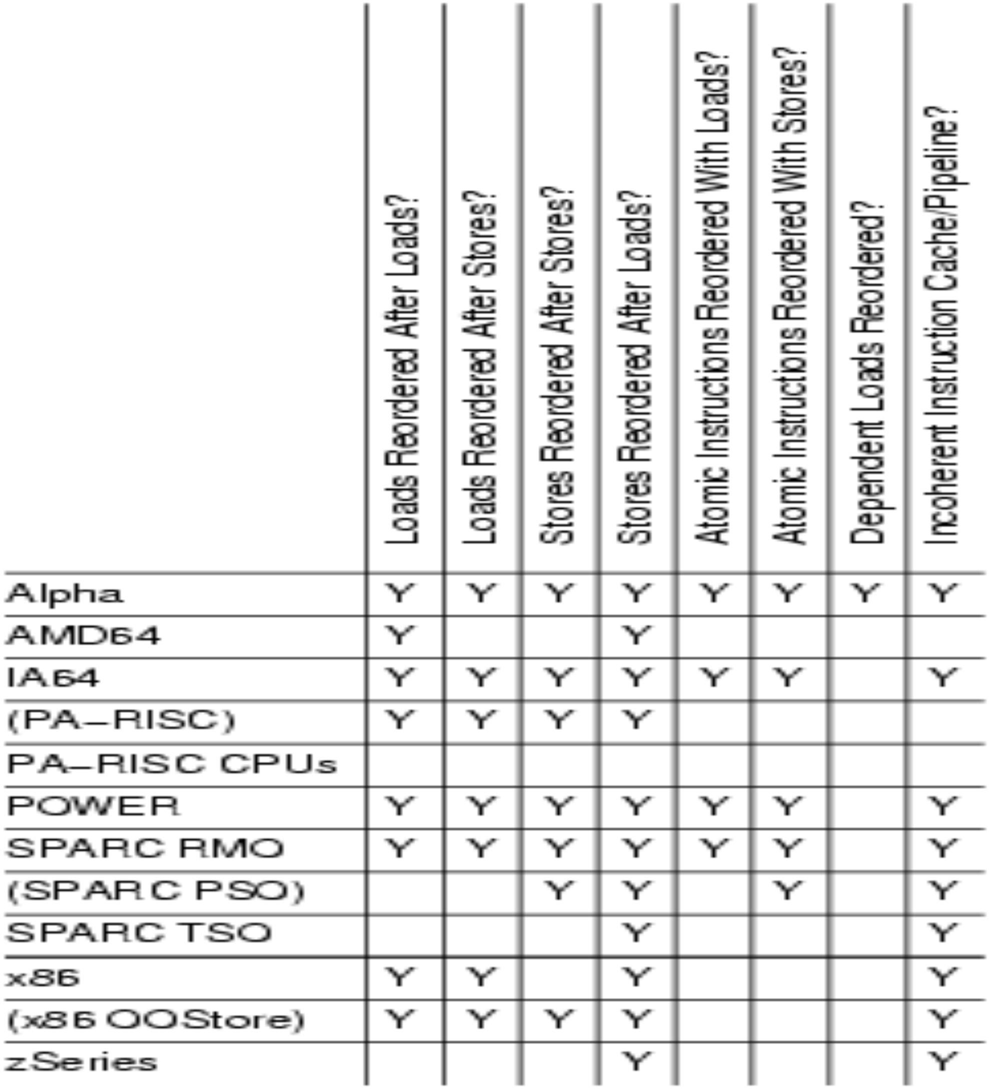
- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Release: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
- However,
  - acquire does not wait for accesses preceding it
  - accesses after release in program order do not have to wait for release
    - operations which follow release and which need to wait must be protected by an acquire

Example



Which operations can be overlapped?

Implementations on Current Processors



	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64	Y			Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA–RISC)	Y	Y	Y	Y				
PA–RISC CPUs								
POWER	Y	Y	Y	Y	Y	Y		Y
SPARC RMO	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86	Y	Y		Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries				Y				Y

Comments

- In the literature, there are a large number of other consistency models
  - processor consistency
  - total store order (TSO)
  - ….
- It is important to remember that these are concerned with reordering of independent memory operations within a processor.
- Easy to come up with shared-memory programs that behave differently for each consistency model.
- All processors today support some version of memory fences and those are exposed in programming language

Memory consistency: program level

- Shared-memory *programming languages* also need to have a memory consistency model
- Example:
  - *compiler* may reorder the two statements in P1 or the two statements in P2, leading to incorrect results
- This is similar to problem at instruction level but it affects compilation, not execution

Code:

Initially A = Flag = 0

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

C++ Memory Model

- Provides SC for data-race free programs

- Memory operations:
  - Data: load, store
  - Synchronization: mutex lock/unlock, atomic load/store, atomic read-modify-write

Write Correct C++ Code

- Mutually exclusive execution of critical code blocks

```
std::mutex mtx;
{
    mtx.lock();
     // access shared data here
    mtx.unlock();
}
```

- Mutex provides inter-thread synchronization
  - Unlock() synchronizes with calls to lock() on the same mutex object

Synchronize Using Locks

```
std::mutex mtx;
bool dataReady = false;
```

```
{
  mtx.lock();
  prepareData();
  dataReady = true;
  mtx.unlock();
}
```

```
{
  mtx.lock();
  if (dataReady) {
    consumeData();
  }
  mtx.unlock();
}
```

Synchronize Using Locks

```
std::mutex mtx;
bool dataReady = false;
```

```
prepareData();

{
  mtx.lock();
  dataReady = true;
  mtx.unlock();
}
```

```
bool b;
{
  mtx.lock();
  b = dataReady;
  mtx.unlock();
}
 if (b) {
    consumeData();
 }
```

Using Atomics

- "Data race free" variable by definition: `std::atomic<int>`
- A store synchronizes with operations that load the stored value
- Similar to `volatile` in Java
- C++ `volatile` is different!
  - Does not establish inter-thread synchronization, not atomic (can be part of a data race)

```
                  std::mutex mtx;
                  std::atomic<bool> dataReady(false);


prepareData();                        if (dataReady.load()) {
dataReady.store(true);                  consumeData();
                                      }
```

Summary

- Two problems: memory consistency and cache coherence
- Cache coherence
  - preserve the illusion that there is a single logical memory location corresponding to each program variable even though there may be many physical memory locations where the variable is stored
- Memory consistency model
  - what instructions is hardware allowed to reorder?
  - nothing really to do with memory operations from different processors/ threads
  - sequential consistency in systems with coherent caches: perform global memory operations in program order
  - relaxed consistency models: all of them rely on some notion of a fence operation that demarcates regions within which reordering is permissible