CSC2/458 Parallel and Distributed Systems
Parallel Memory Systems – Consistency

Sreepathi Pai

February 8, 2018

URCS

Outline

Memory Consistency

Programming on Relaxed Consistency Machines

Memory Models for Languages

Special Relativity

Outline

Example

```
Initial: flag = 0, value = 0

   T0                  T1

value = 1          if(flag == 1)
flag = 1              assert(value == 1)
```

Will the assert ever fail?

I.e. will value $==$ 0?

Hardware Reorderings

- Out-of-order processors can reorder *independent instructions*
- Independent:
  - Not data-dependent
  - Not control-dependent

Compiler Reorderings

- Optimizations that change order:
  - Code hoisting
  - Code sinking
  - Moving values to registers

Problem

- Relative ordering of memory operations can be changed by hardware or compilers.
- Data/Control dependences are still preserved on a single processor.
- However, reorderings may cause other processors to see an order that was not intended by the programmer.

Sequential Consistency

Definition: [A multiprocessor system is *sequentially consistent* if] the result of any execution is the same as if the operations of all the 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.

- Leslie Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs" (1979)

Sequential Consistency

- There is a global interleaving of operations
  - Implies every read or write is atomic
- There is a per-processor ordering of operations
  - Defined by program order

Sequential Consistent Execution of the Example

```
   T0                T1
value = 1         if(flag == 1)
flag = 1             assert(value == 1)
```

Which of these are sequentially consistent orderings?

```
Order 1          Order 2          Order 3          Order 4

value = 1        if(flag == 1)    flag = 1         value = 1
if(flag == 1)    value = 1        value = 1        flag = 1
flag = 1         flag = 1         if(flag = 1)     if(flag == 1)
                                    assert(...)      assert(...)
```

Implementing Sequential Consistency

Why don't machines implement sequential consistency?

Relaxed Consistency

- Most processors relax orderings between operations:
  - Write to a Read (i.e. a later read can overtake an earlier write)
  - Write to a Write
  - Read to a Read/Write
- Many processors also relax atomicity:
  - A processor can read its writes before it becomes visible to all processors
  - A processor can read another processor's writes before it becomes visible to all processors

Relaxing Write to Read Orderings

```
T1:                          T2:
  flag1 = 1                    flag2 = 1
  if(flag2 == 0)               if(flag1 == 0)
     critical-section             critical-section
```

- Writes are slower than reads
- Processors allow reads to overtake writes in the queue
- This will result in both T1 and T2 entering the critical section!

Relaxing Write to Write Orderings

```
T0                      T1
push:                   pop:
  data = x                while(head == NULL);
  head = &data;           assert(*head == x);  // i.e. read data
```

- `data` may miss in T0 cache
- `head` may hit in T0 cache and overtake the write to `data` in T0
- `pop` may read non-null `head` and possibly inconsistent value of `data`
  - e.g. T1 had `data` cached but not yet invalidated

Relaxing Read to Read/Write Orderings

```
T0                  T1
push:               pop:
  data = x            while(head == NULL);
  head = &data;       assert(*head == x);  // i.e. read data
```

- Same example as for Write to Write ordering
- Here:
  - Read of `*head` (i.e. `data`) in T1 may overtake read of `head`
  - Speculative execution(?)

Atomicity Relaxation 1

- "Reading others writes before they become visible to all processors"
- Writes are communicated to each processor serially
- Some processors will receive values before others
- Should they wait until all processors have received values?
  - Implies two-phase-like propagation
  - Phase 1: propagate values
  - Phase 2: propagate approval to use values

Atomicity Relaxation 1 Example

```
Initial: A = B = 0

T0          T1              T2
A = 1       if(A == 1)      if(B == 1)
              B = 1           assert(A == 1)
```

- Here the `assert` in T2 involving `A` may fail!

Atomicity Relaxation – Reading own writes early

- Writes are stored in a queue
- A read to a location which has been recently written must come from the last write
    - For a processor, this is the write in the queue
- Should the read wait for the write to retire and complete?
- Or should it just read the value in the queue and proceed?

Takeaways

- Memory consistency is a subtle topic
  - Complex Interactions between hardware and software
- Bugs due to incorrect ordering may not be immediately visible
  - Usually show up during high loads
- Goals:
  - Recognize situations where ordering is important
  - Make required required ordering explicit in program

Outline

Memory Ordering Relaxations on Processors

Type	$R \rightarrow R$	$W \rightarrow R$	$W \rightarrow W$	$R \rightarrow W$
Alpha	Y	Y	Y	Y
ARMv7	Y	Y	Y	Y
PA-RISC	Y	Y	Y	Y
POWER	Y	Y	Y	Y
SPARC RMO	Y	Y	Y	Y
SPARC PSO			Y	Y
SPARC TSO				Y
x86				Y
x86 oostore	Y	Y	Y	Y
AMD64				Y

Paul E. McKenney, "Memory Ordering in Modern Multiprocessors"

Ensuring Orderings

- We're going to assume a machine that does not preserve *any* ordering
  - But x86 does preserve some orderings
- Two steps:
  - Identify *synchronizing* vs *data* reads/writes
  - Specify ordering between *data* and *synchronizing* read/write

Sync vs Data Read/Writes

- A *synchronizing* read/write is one that has a race in a sequentially consistent execution
  - Accesses to the same location
  - One of the access is a write
  - No other operation between the accesses
- All other accesses are *data* read/writes
- Note these classifications apply only to shared data reads/writes

Example

```
  T0        T1
D = x    while(H == NULL);
H = &D    assert(*H == x)
```

One possible sequentially consistent ordering:

```
D = x
H == NULL
H = &D
 ...
```

- With assumption of sequential consistency:
  - Accesses to `D` are always interleaved with `H`
  - Accesses to `H` can form a race as shown above
- `D` is data, `H` is synchronization
- Need to specify that it is `not` okay to reorder with respect to `H`

The Memory Ordering Toolbox

- Fences (also called memory barrier)
  - Operations can't cross a fence
  - May be too conservative
- Acquires
  - Ensures *acquire* $\rightarrow$ *all* ordering
  - Operations after acquire cannot execute before it
  - Analogous to lock
- Releases
  - Ensures *all* $\rightarrow$ *release* ordering
  - Operations before release cannot execute after it
  - Analogous to unlock
- "Consume" (C++11)
  - Only preserves ordering between barriers and synchronizations
  - Relaxes atomic visibility constraint

Using a fence

```
  T0                    T1
D = x                 while(H == NULL);
__mfence();           __mfence();
H = &D                assert(*H == x);
```

Using acquire and release – quiz

```
  T0                    T1
D = x                 while(H == NULL);
H = &D                assert(*H == x)
```

- Syntax is $x.acquire()$ and $x.release(value\_to\_write)$

Using acquire and release

```
  T0                    T1
D = x                 while(H.acquire() == NULL);
H.release(&D);        assert(*H == x);
```

- Syntax is made-up and varies from language-to-language

Summary

- Accesses to shared data needs to be ordered
- Ordering can distinguish between data and sync operations
- Can always uses fences to ensure ordering
- Or use acquires and releases for better performance

Further reading:

Sarita V. Adve, Khourosh Gharachorloo, "Shared Memory Consistency Models: A Tutorial", WRL Research Report 95/7

(this lecture draws heavily from this tutorial)

Outline

Remember Compilers can Reorder too!

- What reorderings are compilers permitted to perform?
- What is the behaviour of the abstract machine in the presence of concurrent reads/writes?
- How to write efficient and portable parallel code
  - Code on x86 requires fewer memory ordering instructions than (e.g.) Alpha

C++ memory model

- A thread of execution is a flow of control within a program that begins with the invocation of a top-level function by std::thread::thread, std::async, or other means.
- Memory order
  - When a thread reads a value from a memory location, it may see the initial value, the value written in the same thread, or the value written in another thread
- Forward progress guarantees

Text from `http://en.cppreference.com/w/cpp/language/memory_model`

C++ Data Races

When an evaluation of an expression writes to a memory location and another evaluation reads or modifies the same memory location, the expressions are said to conflict. A program that has two conflicting evaluations has a data race unless

- both evaluations execute on the same thread or in the same signal handler, or
- both conflicting evaluations are atomic operations (see std::atomic), or
- one of the conflicting evaluations *happens-before* another (see std::memory_order)

If a data race occurs, the behavior of the program is undefined.

Text from `http://en.cppreference.com/w/cpp/language/memory_model`

C++ Forward Progress Guarantees

In a valid C++ program, every thread eventually does one of the following:

- terminate
- makes a call to an I/O library function
- performs an access through a volatile glvalue
- performs an atomic operation or a synchronization operation

A thread is said to make progress if it performs one of the execution steps above (I/O, volatile, atomic, or synchronization), blocks in a standard library function, or calls an atomic lock-free function that does not complete because of a non-blocked concurrent thread.

Text from `http://en.cppreference.com/w/cpp/language/memory_model`

Outline

Simultaneity

- A light is switched on in the middle of train carriage
  - train is in an inertial frame
- When does light reach the back and front of the train carriage?
  - for an observer in the train?
  - for an observer standing outside the train?
  - for an observer in a car moving at uniform velocity w.r.t. the train?

For more: Wikipedia article on Relativity of Simultaneity