Relaxed Memory-Consistency Models

[§9.1] In Lecture 13, we saw a number of relaxed memory-consistency models. In this lecture, we will cover some of them in more detail.

Why isn't sequential consistency good enough?

-

- Compilers

Basically, what we want to do is develop models where one operation doesn't have to complete before another one is issued, but the system assures that operations don't become visible out of program order.

SC requires that all memory operations be ordered. Let's see how this prevents performance optimizations that are common in compilers.

Compilers allocate variables to registers for better performance, but this is not possible if sequential consistency is to be preserved.

```
Without register allocation           After register allocation

  P1           P2                       P1           P2
 B = 0        A = 0                    r1 = 0       r2 = 0
 A = 1        B = 1                     A = 1        B = 1
 u = B        v = A                     u = r1       v = r2
                                        B = r1       A = r2
```

Notice that the compiler has reordered some memory operations. Which ones?

After executing the code without register allocation, what final values are possible for **u** and **v**?

After executing the code without register allocation, what final values are possible for **u** and **v**?

So, mere usage of processor registers causes a result not possible with sequential consistency.

Clearly, we must circumvent sequential consistency if we want our programs to perform acceptably.

- We can preserve sequential consistency, but *overlap operations* as much as possible.

  E.g., we can prefetch data, or use more multithreading.

  But sequential consistency is preserved, so the compiler cannot reorder operations.

- We can *allow the compiler to reorder operations* as long as we are sure that sequential consistency will not be violated in the results.

  Memory operations can be executed out of order, as long as they become visible to other processors in program order.

  For example, we can execute instructions speculatively, and simply not commit the results if those instructions should not have been executed.

- We can *relax the consistency model*, abandoning the guarantee of sequential consistency, but still retain semantics that are intuitive.

The reason that relaxed consistency models are useful is that most of the sequential constraints in a program are not really necessary for it to give intuitively correct results.

P_1		P_2	
1a.	`A = 1`	2a.	`while (flag == 0);`
1b.	`B = 1`	2b.	`u = A`
1c.	`flag = 1`	2c.	`v = B`

If the code were to execute sequentially, statement 1a would have to precede 1b, which would have to precede 1c; ditto for 2a–2c.

What precedence relationships among the statements are really necessary?

CS&G discuss relaxed consistency models from two standpoints.

- The *system specification*, which tells how a consistency model works and what guarantees of ordering it provides.
- The *programmer's interface*, which tells what code needs to be written to invoke operations provided by the system specification.

The system specification

[§9.1.1] CS&G divide relaxed consistency models into three classes based on the kinds of reorderings they allow.

- *Write → read reordering*. These models only allow a write to bypass (complete before) an incomplete earlier read.

  Examples: Total store ordering, processor consistency

- *Write → write reordering*. These models also allow writes to bypass previous writes.

  Example: Partial store ordering.

- *All reorderings*. These models also allow reads and writes to bypass previous reads.

  Example: Weak ordering, release consistency.

Write → read reordering

Models that allow reads to bypass pending writes are helpful because they help hide the latency of write operations.

While a write-miss is still in the write buffer, and not yet visible to other processors, the processor can perform reads that hit in its cache, or a single read that misses in its cache.

These models require minimal changes to programs. For example, no change is needed to the code for spinning on a flag:

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

(In this and later code fragments, we assume all variables are initialized to 0.)

This is because the model does not permit writes to be reordered. In particular, the write to ______ will not be allowed to complete out of order with respect to the write to **`flag`**.

What would write $\rightarrow$ read reordering models guarantee about this fragment?

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

Recall from Lecture 13 that processor consistency does not require that all processors see writes in the same order if those writes come from different processors.

Under processor consistency (PC), what value will be printed by the following code?

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

However, this problem does not occur with total store ordering (TSO).

TSO provides—

- Store atomicity—there is a global order defined between all writes.
- A processor's own reads and writes are ordered with respect to itself.

  "`X = 3; print X`" will print **`3`**.
- The processor issuing the write may observe it sooner then the other processors.
- Special **membar** instructions are used to impose global ordering between write $\rightarrow$ read.

Now consider this code fragment.

```
          P1                          P2
1a.   A = 1;               2a.   B = 1;
1b.   print B;             2b.   print A;
```

With sequential consistency, what will be printed?

How do we know this?

Might something else happen under PC?

What about under TSO?

To guarantee SC semantics when desired, we need to use the **membar** (or "fence") instruction. This instruction prevents a following read from completing before previous writes have completed.

If an architecture doesn't have a **membar** instruction, an atomic read-modify-write operation can be substituted for a read. Why do we know this will work?

Such an instruction is provided on the Sun Sparc V9.

Write → write reordering

What is the advantage of allowing write → write reordering?

Well, the write buffer can merge or retire writes before previous writes in program order complete.

What would it mean to "merge" two writes?

Multiple write misses (to different locations) can thus become fully overlapped and visible out of program order.

What happens to our first code fragment under this model?

P_1		P_2	
1a.	`A = 1`	2a.	`while (flag == 0);`
1b.	`B = 1`	2b.	`u = A`
1c.	`flag = 1`	2c.	`v = B`

In order to make the model work in general, we need an instruction that enforces write-to-write ordering when necessary.

On the Sun Sparc V9, the **membar** instruction has a "write-to-write flavor" that can be turned on to provide this.

Where would we need to insert this instruction in the code fragment above?

Relaxing all program orders

Models in this class don't guarantee that anything becomes visible to other processors in program order.

This allows multiple read and write requests to be outstanding at the same time and complete out of order.

This allows read latency to be hidden as well as write latency. (In earlier models, writes couldn't bypass earlier reads, so everything in a process had to wait for a pending read to finish.)

These models are the only ones that allow optimizing compilers to use many key reorderings and eliminate "unnecessary" accesses.

Two models, and three implementations, fall into this category.

- Weak ordering
- Release consistency
- Digital Alpha implementation
- Sparc V9 Relaxed Memory Ordering (RMO)
- IBM PowerPC implementation

We have already seen the two models in Lecture 13.

To get a better appreciation for the differences between the two, lets consider the two examples from CS&G.

Here is a piece of code that links a new task onto the head of a doubly linked list. It may be executed in parallel by all processors, so some sort of concurrency control is needed.

```
…
lock(taskQ);
newTask→next = head;
if (head != NULL) head→prev = newTask;
head = newTask;
unlock(taskQ);
…
```

In this code, **`taskQ`** serves as a synchronization variable. If accesses to it are kept in order, and accesses by a single processor are kept in program order, then it will serve to prevent two processes from manipulating the queue simultaneously.

This example uses a **`flag`** variable to implement a producer/consumer interaction.

```
      P1                             P2
TOP: while (flag2 == 0);  TOP: while (flag1 == 0);
    A = 1;                    x = A;
    u = B;                    y = D;
    v = C;                    B = 3;
    D = B * C;                C = D / B;
    flag2 = 0;                flag1 = 0;
    flag1 = 1;                flag2 = 1;
    goto TOP;                 goto TOP;
```

Which shared variables are produced by P_1?

Which shared variables are produced by P_2?

What is the synchronization variable here?

Again, as long as one access to it completes throughout the system before the next one starts, other reads and writes can complete out of order, and the program will still work.

The diagram below (similar to the one at the end of Lecture 13) illustrates the difference between weak ordering and release consistency.

Each block with reads and writes represents a run of non-synchronization operations from a single processor.

With weak ordering, before a synchronization operation, the processor waits for all previous

Also, a synchronization operation has to complete before later reads and writes can be issued.

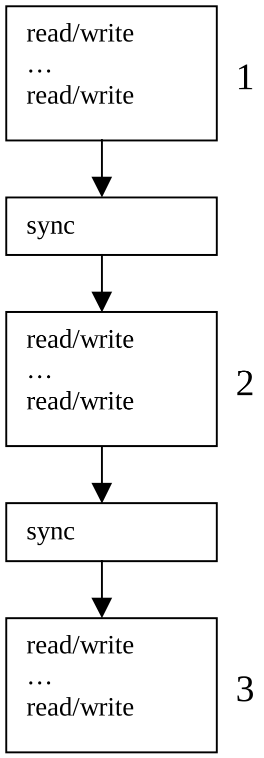
Release consistency distinguishes between

- an **acquire** performed to gain access to a critical section, and

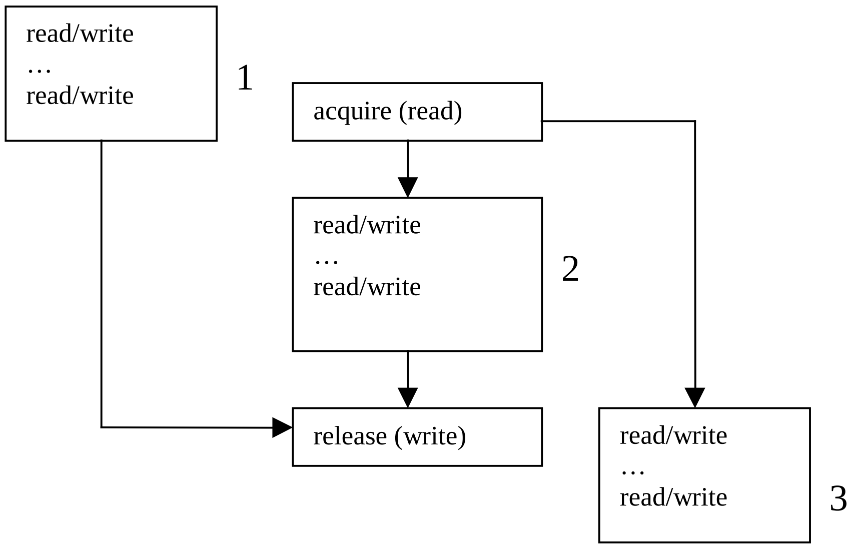
- a **release** performed to leave a critical section.

An **acquire** can be issued before previous reads (in block 1) complete.

After a **release** is issued, instructions in block 3 can be issued without waiting for it to complete.



Weak ordering



Release consistency

Modern processors provide instructions that can be used to implement these two models.

- The Alpha architecture supports two “fence” instructions.
  - ◻ The **memory barrier** (MB) operation is like a **synch** operation in RO.
  - ◻ The **write memory barrier** (WMB) operation imposes program order only between writes.

A read issued after a WMB can bypass (complete before) a write issued before a WMB.

- The Sparc V9 relaxed memory order (RMO) provides an **membar** (fence) with four “flavor bits.”

  Each flavor bit guarantees enforcement between a certain combination of previous & following memory operations.

  - □ read to read
  - □ read to write
  - □ write to read
  - □ write to write

- The PowerPC provides only a single **sync** (fence) instruction; it can only implement WO.

The programmer’s interface

When writing code for a system that uses a relaxed memory-consistency model, the programmer must label synchronization operations.

This can be done using system-specified programming primitives such as locks and barriers.

For example, how are lock operations implemented in release consistency?

- A lock operation translates to
- An unlock operation translates to

Arrival at a barrier indicates that previous operations have completed, so it is

Leaving a barrier indicates that new accesses may begin, so it is

What about accesses to ordinary variables, like the `flag` variables earlier in this lecture?

Sometimes the programmer will “just know” how they should be labeled. But if not, here’s how to determine it.

1. Decide which operations are *conflicting*.

   Two memory operations from different processes *conflict* if they access the same memory location and at least one of them is a write.

2. Decide which conflicting operations are *competing*.

   Two conflicting operations from different processes *compete* if they could appear next to each other in a sequentially consistent total order.

   This means that one could immediately follow another with no intervening memory operations on shared data.

3. *Label* all competing memory operations as synchronization operations of the appropriate type.

Notice that we can decide where synchronization operations are needed by assuming SC, even if we are not using an SC memory model.

To see how this is done, consider our first code fragment again.

	P_1		P_2
1a.	`A = 1`	2a.	`while (flag == 0);`
1b.	`B = 1`	2b.	`u = A`
1c.	`flag = 1`	2c.	`v = B`

Which operations are conflicting?

Which operations are competing?

Which operations need to be labeled as synchronization operations?

How should these operations be labeled in weak ordering?

How should these operations be labeled in release consistency?