Understanding Memory Ordering using LKMM and Herd7

Author: Joel Fernandes <joel@joelfernandes.org> Last Updated: 06/19/2023

Introduction

The herd7 memory consistency tool is used to verify if certain (likely undesirable) outcomes of memory accesses made by concurrent programs exist, given a memory model.

As a matter of fact, herd7 on its own does not know anything about how programs execute.

For example, given a program executing on processor P0:

```
P0(int *x) {
  WRITE_ONCE(*x, 2);
  WRITE_ONCE(*x, 3);
}
```

Any sane computer architecture will conclude that the final value of `x` is 3.

However, herd7 without any memory model (or a memory model that allows everything) will consider the following 2 possible sequences as valid:

```
Candidate execution 1:
1. Value 2 is stored to x.
2. Value 3 overwrites the value 2 that was just stored to x.
```

and,

```
Candidate execution 2:
1. Value 3 is stored to x.
2. Value 2 overwrites the value 3 that was just stored to x.
```

These 2 possible executions are called *candidate executions*.

It is up to the memory model to instruct herd7 to *disallow* some of these candidate executions. A sane memory model should disallow the second candidate execution as it would otherwise be the model of a broken architecture.

The `LKMM` is a memory model for the Linux Kernel which can be fed to the herd7 tools to verify the memory ordering properties of a test program, also known as a Litmus Test. The model is written in the `linux-kernel.cat` and `linux-kernel.bell` files. This article will cover some parts of the model, focusing on how to use the herd7 tool to visually understand the more complex parts of the files. With herd7, it is possible to generate complex graphs showcasing the relationship between memory accesses in a litmus test. This aids in reverse engineering the equations and axioms in the `.bell` and `.cat` files. Armed with this knowledge, readers can explore more advanced nuances on their own using herd7.

Where possible, we will try to prioritize describing how to use herd7, and the syntax of `CAT` code, over actually describing too many details of the axioms, as we believe understanding the axioms can be achieved once the reader is empowered with the knowledge of how to use herd7 and read/write `CAT` code.

Note that, even though a herd7 memory model is abstract in some sense (it does not describe CPU implementation but just a set of properties and rules on memory ordering and program execution), it can still be considered to be a model of how a CPU should behave if it CPU wishes to run the Linux kernel code correctly. It does so by formally defining how memory ordering in a typical multiprocessor running kernel code should behave.

In the next section, we will describe how to eliminate the nonsensical candidate execution 2, which no sane CPU design should support, certainly not the `LKMM`.

A few basics first

Cache coherence and ordering

Cache coherence refers to the principle that in a multi-processor system, all CPUs must share a consistent view of the memory contents. This requires that for each location in shared memory, the stores to that location must form a single global ordering which all the CPUs agree on (the coherence order), and this ordering must be consistent with the program order for accesses to that location. Cache coherence protocols are used to

manage this ordering and ensure that all processors have the same view of the order in which the data to the shared memory location was written. It is important to understand that due to concurrency and timing, the order in which writes to the same memory location occur is unpredictable. However, once the writes occur, all CPUs see the same order in which those writes happened.

In the `LKMM`, the cache-coherent ordering between any two writes is connected by a `co` relationship. For example, two writes `W1` and `W2` to the same memory location, where `W1` occurred before `W2`, would be denoted by `W1 ->co W2.`

Program Order

Program order (`po`) is the order in which instructions are presented to a CPU's execution unit. `po-loc` is a sub-relation of `po` that links two memory accesses when the first comes before the second in program order and they access the same memory location.

For example, using `po-loc`, we can link the two memory accesses in the following program:

```
P0(int *x) {
  WRITE_ONCE(*x, 5);  // W1
  WRITE_ONCE(*x, 6);  // W2
}
```

Since `W2` comes after `W1` in program order and they access the same memory location (`x`), we can say `W1 ->po-loc W2`.

Ordering and cycles

Memory models are primarily concerned with ordering. One of the main types of ordering is temporal ordering, which specifies the order in which a set of events happened in time. Two different sequences of temporal ordering may also overlap in space, making them independent of each other and happening in parallel.

When a memory model requires certain accesses to be ordered, cycles are ruled out. If a certain outcome for final values of a piece of code can only happen if those accesses would form a cycle, then the memory model forbids the cycle and predicts that the outcome cannot occur.

However, a memory model on its own may not always be able forbid cycles, and needs help from the programmer by having them issue memory barriers. Such memory barriers serve to filter out certain execution candidates before the model can even be applied to them.

Axioms of the Linux Kernel Memory Model

Now, let's examine the various axioms (rules) of the `LKMM`. To permit a candidate execution of a concurrent program, it must not be prohibited by the axioms of the `LKMM`. As previously stated, models like the `LKMM` that are reasonable will prohibit executions that are impossible on modern hardware.

1. Sequential consistency per-variable (SCPV)

This property is fundamental in modern processors, and it basically means that reads and writes to a certain variable happen in a total-order. In other words, for a specific variable, it is not possible to observe a sequence of writes to that variable in an order different from the order in which its values were written.

This also applies to the writes happening on the same CPU. In a single CPU, the writes happening on the same variable happen in program order execution.

The way the memory model can enforce this is by defining a rule forbidding a certain property. Let us see if we can define the violations of `SCPV` as a cycle in a particular candidate execution, and then tell the model that such execution candidates are forbidden.

Consider a program doing the same program from earlier doing a pair of writes, this time with the events labeled:

```
P0(int *x) {
  WRITE_ONCE(*x, 2);            // event W1
  WRITE_ONCE(*x, 3);            // event W2
}
```

As described earlier, there are 2 candidate executions:

Candidate 1. The final value of x is 2. This happens because of the following candidate execution:

```
W1 ->co W2
```

Candidate 2. The final value of x is 3. This happens because of the following candidate execution:

```
W2 ->co W1
```

Visually this can be shown as 2 candidate graphs: [comment]: <> (Add 2 graphs here)

A quick note on `->co`. It describes the order of writes to the same variable. For example:

```
W1 ->co W2
```

means the writes (to the same variable) followed the order of first W1, and then W2 in the cache-coherent memory. In other words, the program execution resulted in W2 overwriting W1 with the final value of the variable decided by W2.

So, we wish to forbid the pattern in candidate #2. How do we do that?

First, lets learn a new relation. Following the instruction order in the instruction stream, there is a relation in LKMM called `po-loc`.

The `po-loc` relation links 2 program-ordered memory accesses happening on the same CPU, and on the same variable.

So we have a relation `W1 ->po-loc W2` in the program.

Lets learn a new notion of how heard builds a relation (a set of event-pairs).

In herd7's `CAT` language, using a keyword like `po-loc` or `co` gives you a set of all possible event-pairs (relations). This set is actually (confusingly) called a relation.

For example, `->co` is the following relation with 2 event pairs:

```
[ (W1, W2)  ,  (W2, W1)  ]
```

Similarly, `->po-loc` is the following relation:

```
[ (W1, W2) ]
```

We can combine program ordering (`->po-loc`) and cache coherent ordering (`->co`) to build a cycle.

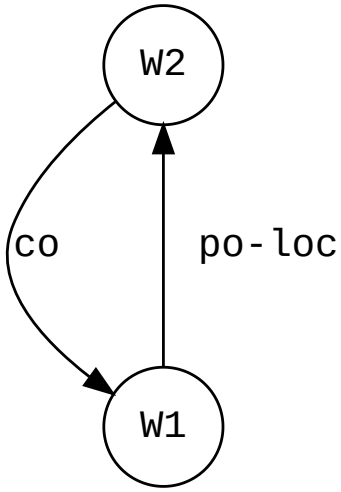
We can build a new relation by taking the union of the 2, using the union order (pipe). `po-loc | co`

This united relation is:

```
[ (W1, W2), (W2, W1) ]
```

Or it can be written as `W1 ->po-loc W2 ->co -> W1`

Visually this union results in the following graph:



Graph showing po-loc and co cycle

This is a cycle! So we can we can simply define a property (or axiom) in the CAT code as:

```
let scpv = acyclic po-loc | co
```

This makes herd7 forbid all candidate executions that have such a cycle, and thus don't satisfy the `scpv` property.

Note that to forbid candidate execute #2, we could have simply said:

```
let scvp = acyclic co
```

However, consider the following 2 CPU example, with writes happening on different CPUs:

```
P0(int *x) {
  WRITE_ONCE(*x, 2);            // event W1
  WRITE_ONCE(*x, 3);            // event W2
}

P1(int *x) {
  WRITE_ONCE(*x, 4);            // event W3
}
```

Here there are 6 possible candidate executions:

1. Final value is 4.

```
W1 ->co W2->co W3
```

1. Final value is 4.

```
W2 ->co W1 ->co W3
```

1. Final value is 2.

```
W2 ->co W3->co W1
```

1. Final value is 3.

```
W1 ->co W3 ->co W2
```

1. Final value is 3.

```
W3 ->co W1 ->co W2
```

1. Final value is 2.

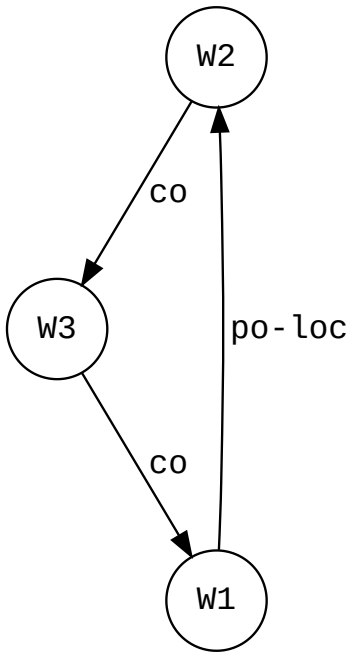
```
W3 ->co W2 ->co W1
```

Here cases #3 and #6 should be forbidden, as the only allowed final-value outcomes should be 3 or 4.

Candidate execution #3 has the following relations:

```
W2 ->co W3
W3 ->co W1
W1 ->po-loc W2
```

A cycle can be observed when uniting all of these relations using `po-loc | co`, which is equivalent to this graph:



Graph showing po-loc and co cycle

Thus `acyclic po-loc | co` can again be used to forbid the candidate executions #3, and similarly #6.

So far we have only considered stores, however we must order the loads with respect to these stores as well, and such reads cannot observe the stores to the same variable out of order. Let us next look at an example, where the above acyclic definition is incomplete.

Consider the following Litmus test involving read accesses:

```
C scpv-rf

{}

P0(int *x)
{
        WRITE_ONCE(*x, 2);
        WRITE_ONCE(*x, 3);
}

P1(int *x)
{
        int r1;
        int r2;

        r1 = READ_ONCE(*x);
        r2 = READ_ONCE(*x);
}

exists (1:r1=3 /\ 1:r2=2)
```

Here, we hope that the reads to variable `x` are observed by P1 in the program-order that were written in P0. So the forbidden exists clause should never occur.

However, if you were to build a CAT model as follows, using the previously determined acyclic property, then the forbidden case indeed happens.

Here is the CAT code:

```
include "cos.cat"

acyclic po-loc | co
```

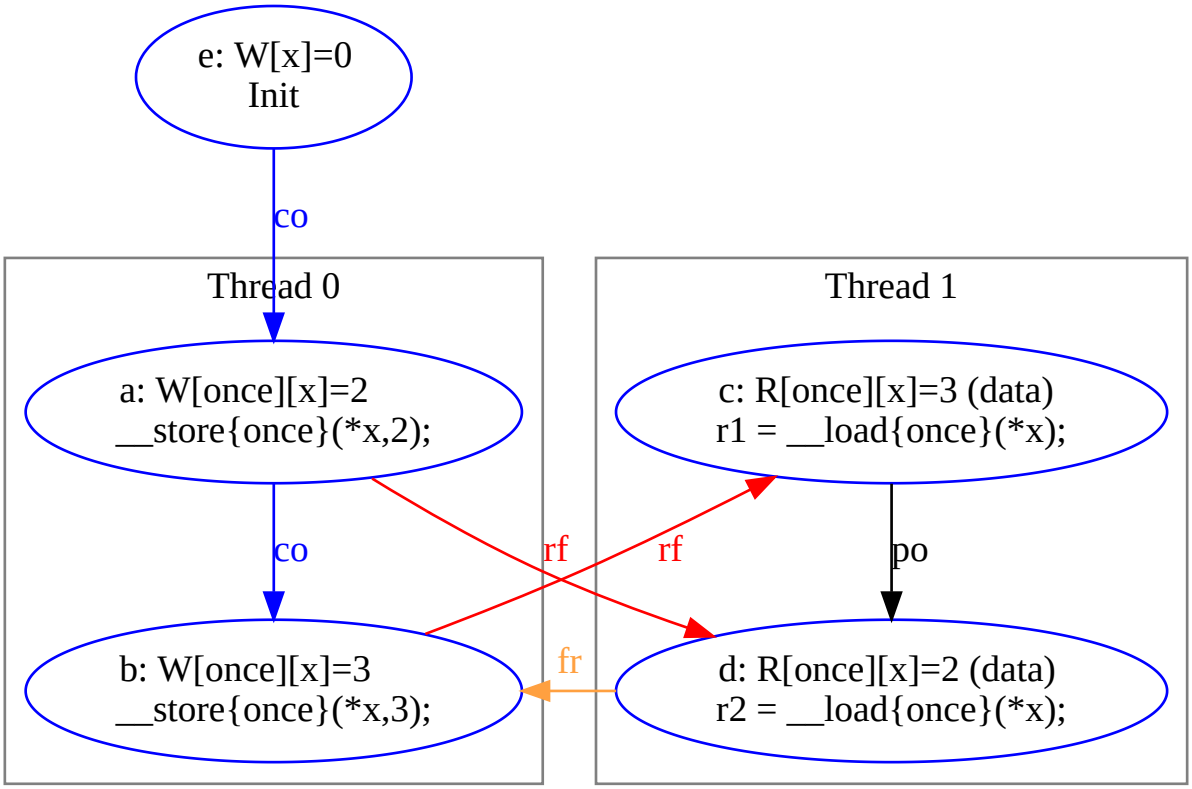
This can be run using herd7 as follows, with the `-show prop` options to generate a DOT graph file of the forbidden case:

```
herd7 -bell linux-kernel.bell -macros linux-kernel.def -cat test.cat scpvrf.litmus -show p
rop -o OUT/
```

Running this shows:

```
Test scpv-rf Allowed
States 9
1:r1=0; 1:r2=0;
1:r1=0; 1:r2=2;
1:r1=0; 1:r2=3;
1:r1=2; 1:r2=0;
1:r1=2; 1:r2=2;
1:r1=2; 1:r2=3;
1:r1=3; 1:r2=0;
1:r1=3; 1:r2=2;
1:r1=3; 1:r2=3;
Ok
Witnesses
Positive: 1 Negative: 8
Condition exists (1:r1=3 /\ 1:r2=2)
Observation scpv-rf Sometimes 1 8
Time scpv-rf 0.00
Hash=f2f1ffdc787b0e923ae8cf087fcd5b12
```

And the graph for the forbidden case generated by herd7 is as follows:



Test scpv-rf, Generic(Linux-kernel memory consistency model)

A graph showing failure of read sequential consistency

As you can see, there is a cycle between `->po-loc`, `->rf` and `->fr`.

This shows that both `->rf` and `->fr` should also included in the acyclic property as well. Hence to avoid the problematic candidate execution, the SCPV property should be `acyclic po-loc | co | rf | fr`. That is indeed the case in the Linux kernel's memory model.

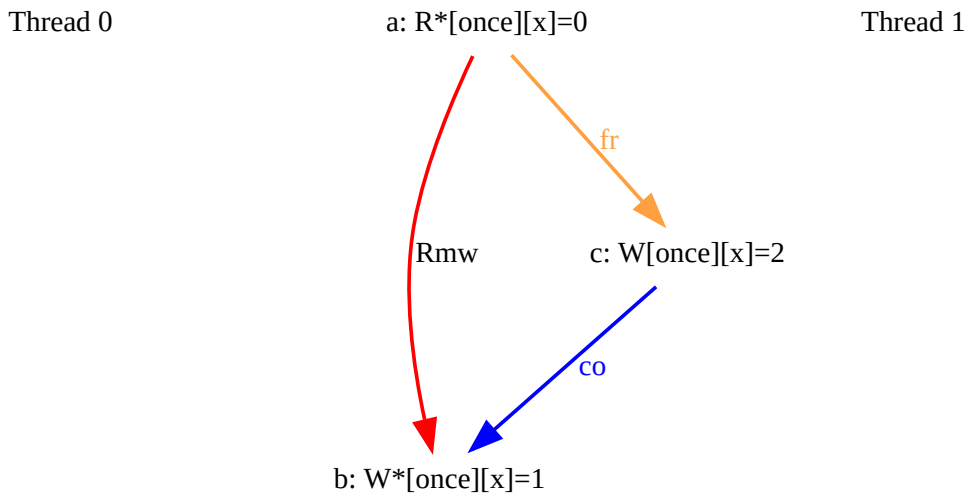
2. Atomicity

Atomicity can be defined as a read-modify-write (RMW) operation on a memory location which happens atomically, that is no write from another CPU can happen between the read and the write. In other words, the read and write operation in the RMW operation are one (atomic).

First let us see what happens if we have an RMW on CPU 0 being interleaved with a write from another CPU 1. Consider the litmus test we will use to generate a graph from:

```
C rmw-1

{}

P0(int *x)
{
        int r0;

        r0 = xchg(x, 1);
}

P1(int *x)
{
        WRITE_ONCE(*x, 2);
}


exists (0:r0=0 /\ x=1)
```

The below graph generated by herd7 shows the case that exists:



Test rmw-1, Generic(Linux-kernel memory consistency model)

The `Rmw` edge in the graph illustrates the data-dependent relation between the read and the write, with the additional implication that it is to be atomic. The `fr` edge shows that a write on another CPU happened after the read operation of the RMW. The `co` edge shows that another write overwrote that write.

This is precisely what we want our model to prevent – another write should not be allowed to interleave in such a fashion, and all modern CPU architectures have hardware support to prevent such interleaving. We expect APIs like `xchg()` in the Linux kernel, that does use RMW instructions to work correctly.

To prevent the above case, we first build a relation linking the `fr` and `co` edges using the sequence operator (semicolon) as follows:

```
(fr; co)
```

The only thing left to form an intersection between this relation and the relation consisting of the rmw operation, to form a new relation, and then forbid then forbid that such a relation exists in any candidate execution. These candidates will be rejected, as our model cannot possibly support them (just like the underlying hardware cannot).

```
empty rmw & (fr; co)
```

The next several sections will discuss difficult-to-understand topics using the herd7 modeling tool and using examples and mathematics where possible.

3. Propagation

One of the most confusing parts of memory ordering is that of delayed propagation. Using a formal modeling tool like LKMM, we can get a better understanding of this concept.

Consider the following litmus test with 2 concurrent threads running on 2 CPUs (P0 and P1):

```
P0(int *x, int *y)
{
    WRITE_ONCE(*x, 1);
    smp_wmb();
    WRITE_ONCE(*y, 1);
}

P1(int *x, int *y)
{
    int r0;
```

```
    WRITE_ONCE(*y, 2);
    smp_mb();
    r0 = READ_ONCE(*x);
}

exists (y=2 /\ 1:r0=0)
```

The exists clause tries to verify if any candidate execution in this concurrent program can result in `y` having a final value of 2, and P1's read into register r0 having a value of 0.

Intuitively speaking, if y has a final value of 2, then the store of 1 to y in some sense preceded the store of 2 to y. The `smp_mb()` guarantees that the `P1()`'s store to y precedes its load from x, and the `smp_wmb()` guarantees that `P0()`'s stores are seen in order.

Putting all of this together, one might hope that whenever the final value of y is 2, the final value of r0 would be guaranteed to be 1.

Unfortunately, there is real hardware that runs the Linux kernel on which the final value of y can be 2 and the final value of `r0` can be 0. LKMM must therefore allow this outcome, counter-intuitive though it might be.

The issue happens because we did not consider a subtle point related to propagation here. Even though we have the `->co` relation between the stores to `y`, the store to `x` can be propagated much later to thread `P1` as the `weak fence` delays the propagation.

Let us see what it takes to forbid this mathematically and why weak fences cannot forbid it. First lets define a `->prop` relation. A `->prop` relation guarantees the propagation of changes to different memory locations to happen in a certain order.

So for instance, if we have writes `W1` and `W2`, `W1 ->co W2` implies `W1 ->prop W2`. Weak fences also assist in propagation of previous `->co` links.

So, `W1 ->co W2 ->weak-fence W3` also implies `W1 ->prop W3`. This property of weak fences is called cumulativity.

Applying this to the previous example, for `x` to be read as 0 with the final value as `y`, we generate:

```
WRITE_ONCE(*y, 1) ->co WRITE_ONCE(*y, 2) ->strong-fence  READ_ONCE(*x)
```

and,

```
READ_ONCE(*x); ->fr WRITE_ONCE(*x, 1); ->weak-fence WRITE_ONCE(*y, 1);
```

This implies:

```
WRITE_ONCE(*y, 1) ->prop  READ_ONCE(*x)
```

and

```
READ_ONCE(*x); ->prop WRITE_ONCE(*y, 1);
```

That may appear like a cycle at first that we can should be intuitively forbidden, however it is important to realize that `A ->prop B` and `B ->prop C` does not imply `A ->prop C`. Because we have no way of chaining 2 `->prop` relations this way, we cannot define a chain of `->prop` relations to be acyclic because `->prop` relations may not happen temporally in a strict order.

In plain words, The action “A propagating before B” can happen after the action “B propagating before C”.

In order to enforce the order `A ->prop B ->prop C`, we need both `prop` relations to involve strong fences, not just one of them. This upgrades the `prop` relation to a `pb` relation (propagates before) in LKMM terminology.

Applying this to the previous example, we have:

```
WRITE_ONCE(*y, 1) ->co WRITE_ONCE(*y, 2) ->strong-fence  READ_ONCE(*x)
```

and

```
READ_ONCE(*x); ->fr WRITE_ONCE(*x, 1); ->strong-fence WRITE_ONCE(*y, 1);
```

This implies:

```
WRITE_ONCE(*y, 1) ->pb  READ_ONCE(*x)
```

and

```
READ_ONCE(*x); ->pb WRITE_ONCE(*y, 1);
```

Now we can forbid this undesirable cause of `x` being read as 0, simply saying that the LKMM forbids cycles in `->pb`.

4. Speculative execution and memory ordering

In this section, we will go over an example of control-flow speculation that causes memory accesses to happen in an unexpected order, and how herd7 formally models this.

Consider the following litmus test:

```
C rfitest

{
    int a = 0;
    int *x;
    int y = 0;
}

P0(int **x, int *y, int *a)
{
    WRITE_ONCE(*a, 1);
    smp_mb();
    WRITE_ONCE(*y, 1);

}

P1(int **x, int *y, int *a)
{
    int r0;
    int *r1;
    int r2;

    r0 = READ_ONCE(*y);
    if (r0 == 1) {
       WRITE_ONCE(*x, a);
       r1 = READ_ONCE(*x);
       r2 = READ_ONCE(*r1);
```

```
    }
}

exists (1:r0 = 1 /\ 1:r2 = 0)
```

In this example, to avoid the condition where register r0's value in P1 is 1 and r2's value is 0, we expect P0's store to `a` to propogate to P1's load of `a`. This is intuitively enforced by the fact that, between that "store to a" and "load from a" event, we have the following: 1. A strong fence: `smp_mb()`. 2. A `read-from` dependency between P0's store to `y` and P1's load of `y`. 3. And a control dependency that we will execute the body of the loop only if the value loaded in step 2 obtained a value of 1.

However, modern hardware is anything but intuitive! So the case where register r0's value in P1 is 1 and r2's value is 0 can very well happen on weakly ordered architectures like PowerPC.

The reason for this is control-flow speculation along with a feature in pipelined hardware called `store forwarding`. The body of the loop can be speculatively executed by the processor in advance of knowing the value `y` being loaded into `r0`. Further, the body of the loop is indendent of r0. So as long as the processor does not COMMIT the side effects of store to `x` in P1 until the processor knows the value of `y`, everything is fine.

This is where store-forwarding comes in. Even before the store of the address of a to `x` in P0 can be committed, the address of `x` can be forwarded directly to the load of `x`. The next statement then loads a stale value of `a`, before the value of `y` can even be loaded, thus resulting in the counter-intuitive outcome. The store-forwarding behavior is modeled by the `->rfi` relation in the LKMM (Read-from internal relation).

As can be seen in this example, both store-forwarding and speculation are required to result in the counter-inutive outcome.

This can be avoided by adding a full memory barrier as the first statement in the body of the `if` block. This will ensure that the process of speculation completes before executing the body of the loop.

Conclusion

These is a work in progress article. I tried to go deep and explain things that are not explained in other places. I may update this article in the future as I uncover the mysteries of herd7 and LKMM.