May 12th description of final fields

For each execution, the behavior of reads is influenced by two additional partial orders, dereference chain ($\overset{dc}{\rightarrow}$), and memory chain ($\overset{mc}{\rightarrow}$) which are considered to be part of the execution (and thus, fixed for any particular execution). These partial orders must satisfy the following constraints (which need not have a unique solution):

- **Dereference Chain** If an action a is a read or write of a field or element of an object o by a thread t that did not initialize o, then there must exist some read r by thread t that sees the address of o such that $r \overset{dc}{\rightarrow} a$.
- **Memory Chain** There are several constraints on the memory chain ordering:
  - a) If r is a read that sees a write w, then it must be the case that $w \overset{mc}{\rightarrow} r$.
  - b) If r and a are actions such that $r \overset{dc}{\rightarrow} a$, then it must be the case that $r \overset{mc}{\rightarrow} a$.
  - c) If w is a write of the address of an object o by a thread t that did not initialize o, then there must exist some read r by thread t that sees the address of o and $r \overset{mc}{\rightarrow} w$.

Given a write w, a freeze f, an action a (that is not a read of a final field), a read r_1 of the final field frozen by f and a read r_2 such that $w \overset{hb}{\rightarrow} f \overset{hb}{\rightarrow} a \overset{mc}{\rightarrow} r_1 \overset{dc}{\rightarrow} r_2$, then when determining which values can be seen by r_2, we consider $w \overset{hb}{\rightarrow} r_2$ (but these orderings do not transitively close with other $\overset{hb}{\rightarrow}$ orderings). Note that the $\overset{dc}{\rightarrow}$ order is reflexive, and r_1 can be the same as r_2.

For reads of final fields, the only writes that are deemed to come before the read of the final field are the ones derives through the final field semantics.

The intuition for the partial orders is that neither of then can be easily violated, by either compiler or processor optimizations, without doing some kind of aggressive speculative address prediction. The Alpha processor is the only known processor that can perform reads in an order contrary to the $\overset{dc}{\rightarrow}$ order.

From these rules, we can deduce that for any read r of a final field of an object o by a thread other than the thread that allocated o, there must be a write w of the address of o by the thread that allocated o such that $w \overset{mc}{\rightarrow} r$. Thus, if the only writes of the address of an object happen after all freezes of fields of that object, we can straightforwardly deduce that all reads of the final fields of that object by other threads are correct.

On most systems, excluding the Alpha processor, final field semantics can be conservatively implemented simply by treating a freeze operation as a memory barrier that ensures that earlier stores are ordered before subsequent stores both in the compiler and, if needed, within the processor. This is sometimes called a StoreStore barrier.

Because the semantics of final fields are more subtle than those of normal fields, they allow for additional compiler transformations. For example, if a final field contains a primitive value, then there can't be any dependence chains from the read r_1 of that field. Thus, only the write of the final field needs to be ordered before the freeze; any other write may be reordered with the freeze.

Similarly, any operation that could not be the start of a memory chain leading to a read of a final field can be freely reordered with a freeze operation. Thus, a write of a primitive value can be always be reordered with a preceding freeze operation.

`f` is a final field; its default value is 0

Thread 1	Thread 2	Thread 3
`r1.f = 42;`	`r2 = p;`	`r6 = q;`
`p = r1;`	`r3 = r2.f;`	`r7 = r6.f;`
`freeze r1.f;`	`r4 = q;`	
`q = r1;`	`if (r2 == r4)`	
	`    r5 = r4.f;`	

Assuming the reads of `p` and `q` see the values written by Thread 1, `r3` and `r5` can be either 0 or 42, and `r7` must be 42.

Figure 1: Final field example where reference to object is read twice

Example

In order to determine if a read of a final field is guaranteed to see the initialized value of that field, you must determine that there is no way to construct the partial orders $\overset{mc}{\rightarrow}$ and $\overset{dc}{\rightarrow}$ without providing the chain $w \overset{hb}{\rightarrow} f \overset{hb}{\rightarrow} a \overset{mc}{\rightarrow} r_1 \overset{dc}{\rightarrow} r_2$ from the write of the field to the read of that field.

In Figure 1, an object o is constructed in Thread 1 and read by Threads 2 and 3. We assume that the reads in Threads 2 and 3 of `p` and `q` see the values written by Thread 1. The actions of this execution, along with a set of memory chain orderings, is shown in Figure 2; the edges are labeled with the memory chain rule that produces them. Note that the edges labeled b are also dereference chain orderings.

The write to the final field happens-before the freeze, which happens before a write of the address the object to `q`. This write is seen by Thread 3; thus, there is a memory chain from that write to the read. The value seen by the read is dereferenced by the next action; thus, the write to the final field happens-before the read. The important point here is that if Thread 3 is able to read the final field, this chain **must** be in place. There is no other chain that can be formed using the rules for constructing dereference chains and memory chains.

Thread 2 is not so lucky. The write in Thread 1 and read in Thread 2 of `p` are involved in a memory chain. The write in Thread 1 and read in Thread 2 of `q` are also involved in a memory chain. Both reads of `f` see the same variable. There can be a dereference chain from the reads of `f` to either the read of `p` or the read of `q`, because those reads see the same address. If the dereference chain is from the read of `p`, then there is no guarantee that `r5` will see the value 42.

Notice that for Thread 2, the deference chain orders `r2 = p` $\overset{dc}{\rightarrow}$ `r5 = r4.f`, but *does not* order `r4 = q` $\overset{dc}{\rightarrow}$ `r5 = r4.f`. This reflects the fact that the compiler is allowed to move any read of a final field of an object o to immediately after the the very first read of the address of o within that thread.

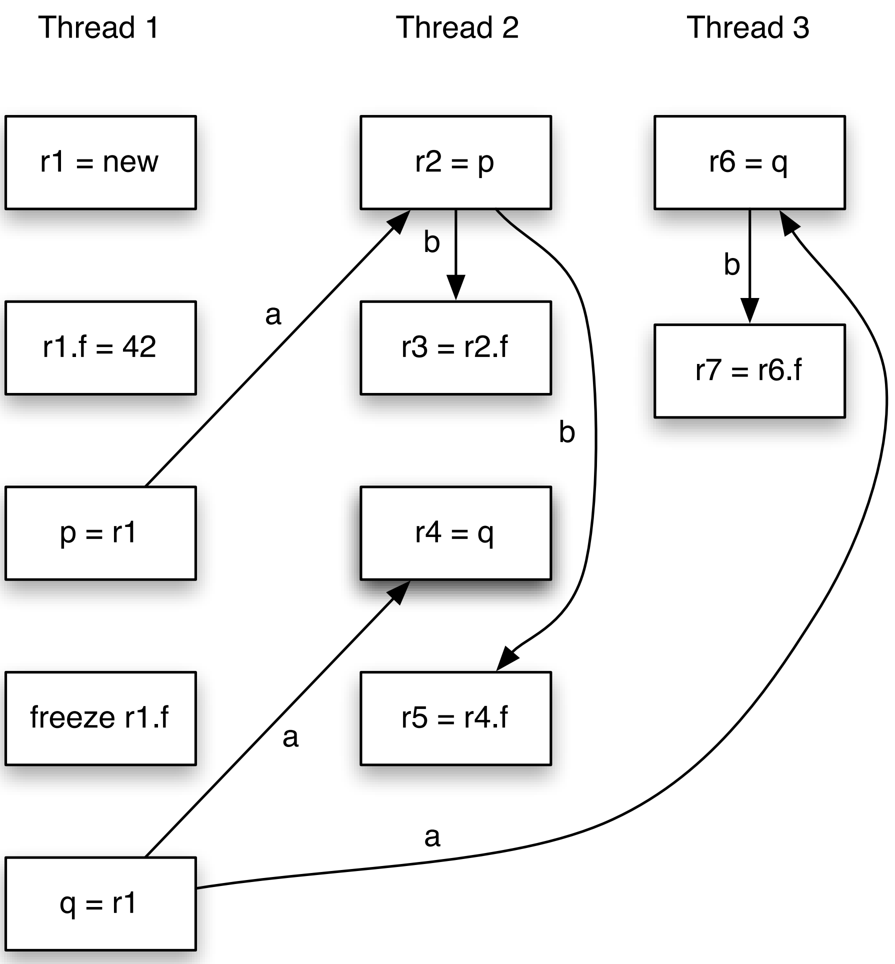

Figure 2: Partial Orders over an execution of Figure 1