A New Approach to the Semantics of Multithreaded Java

Jeremy Manson and William Pugh

Institute for Advanced Computer Science and Department of Computer Science
University of Maryland, College Park
{jmanson,pugh}@cs.umd.edu

July 22, 2002

1 Introduction

The following is a rough overview of the new version of the multithreaded semantics we will propose for Java. This version of the semantics is simplified by assuming that volatiles are fully sequentially consistent; we also don't include anything about final fields.

The surface level description of the semantics is very different from our previous description, but the underlying semantics are very similar. The core portion of the old semantics (excluding initWrites and guaranteedReads) are identical to the core portion of the new semantics. Rather that talking about allWrites, overwritten and previous sets, we just talk about happens-before edges and paths. But all allWrites, overwritten and previous ever did was to encode happens-before edges.

In our new semantics, we have something called *prescient unsynchronized reads* that address the same issues/reorderings addressed by *initWrites* in our previous semantics. Prescient unsynchronized reads do a much better job of solving those issues than initWrites.

We have moved away from an operational model for our semantics. Instead, the semantics allow you to decide whether an execution trace for a program is valid.

2 Overview of semantics

Programs Not much needs to be said. At the moment, we are going to ignore some issues associated with loops, and just say that a program is a set of threads, each of which is a sequence of statements, each of which is uniquely identified by a GUID.

Actions Each action corresponds to an activity performed by a statement of a program. Example actions include reads or writes of a heap location and locks or unlocks of a monitor. We don't have compound actions (e.g., incrementing a heap location) or purely thread local actions (e.g., updating a register). Each action has the GUID of the statement that generated it.

An action is annotated with information about the execution of that action: the variable/monitor accessed, the value read or written, etc.

A read action may be annotated as a *prescient unsynchronized read.* More on that later.

Execution Trace An execution trace E of a program P is a sequence of actions that is an interleaving of the actions generated by each thread of P, such that the actions of each thread t are consistent with and ordered according to the unithread semantics of t in P. In other words, the actions of a thread must occur in exactly the same order as the corresponding statements in the program. As it turns out, some reorderings will not result is any observable changes to the potential behavoir of a program, and thus compilers and other portions of the JVM will be allowed to perform those reorderings.

The values seen by reads are not determined by the unithread semantics. Instead, they are determined by the memory model. An execution trace is a *valid execution trace* if the values seen by the reads in E are valid according to the memory model.

At the beginning of the execution trace, there is an initial write of the default value to each variable.

When we say that the same action occurs in two different execution traces, we mean that the actions have the same annotations (e.g., GUID, variable read and value seen).

Unithread semantics The unithread semantics are the standard semantics for single threaded programs, and allow the complete prediction of the behavior of a thread based on the values seen by read actions within the thread.

Observable Behavior The observable behavior $B(E)$ of an execution trace E is the behavior of E that could be observed. While there are a couple of ways of defining observable behavior, an adequate definition for our purposes is that the observable behavior $B(E)$ of E is described by the values read by each read action in E. In particular, $B(E)$ does not directly describe the interleaving of the actions from different threads, although parts of that interleaving might be inferable from the behavior.

Happens-before edge When we say there is a happens-before edge from action x to action y, we mean that y semantically happens before x. Within an execution trace, there is a happens-before edge from each action in a thread t to each previous action in t. Furthermore, there is an edge from a lock action on monitor m to all previous unlock actions on m, and a path from each read of a volatile variable v to all previous writes to v. There is also an edge from every thread action to all initial write actions.

Happens-before path There is a happens-before path from an action y to an earlier action x if there is a path of happens-before edges from y to x.

```
              Initially:
           g0:  a = 0; g1:  b = 0

         Thread 1:     Thread 2:
         g2: r1 = a;   g4: r2 = b;
         g3: b = 1;    g5: a = 1;
```

Figure 1: Sample program

Memory Model The memory model describes what values a read is allowed to observe in a valid execution trace. A normal read r of a heap variable v may observe the value of any previous write w to v unless there is a happens-before path from r to w on which there is another write to v.

One observation is that the values that can been seen by a read r are determined only by the sequences of aliased writes on happens-before paths from r.

A read action may be marked as a *prescient unsynchronized read* (PUR). A PUR r of variable v may see the value of a write w with GUID g to v if there is no happens-before path between w and r (usually, the write occurs later in the execution trace but we allow it to occur earlier to simplify some proofs).

Each PUR must be *validated* by a valid execution trace. Let r be PUR of variable v in execution trace E that sees the value of a later write w with GUID g. A PUR r is validated by an execution trace E' in which r sees the same value from the same variable from the same write (with GUID g). However, in E', the write w must occur before r and there must be no happens-before path in E' from the read r to the write w (which means that the read is incorrectly synchronized).

We observe that the values that can been seen by a PUR r are determined only by the sequences of aliased writes on happens-before paths *either* from or to r.

If execution trace E' validates a PUR in E, then E' validates E. The graph of which execution trace validates which other execution traces must be acyclic (i.e., all validation must eventually originate in valid execution traces with no PURs).

3 Example

We now derive some valid behaviors of the program in Figure 1. Note that each statement is labeled with a GUID. Figure 2 shows four execution traces of 1 and the corresponding behavior. Trace E_2 contains a prescient unsynchronized read that is validated by E_1. Note that Figure 1 was one of the motivating examples for initWrites in the earlier version of the semantics. With this new version of the semantics, those issues are handled by prescient unsynchronized reads.

Trace E_1

GUID	action	notes
g0	a = 0	initial write
g1	b = 0	initial write
g2	r1 = a	sees 0 from g0
g3	b = 1	
g4	r2 = b	sees 1 from g3
g5	a = 1	

Behavior: r1 = 0 and r2 = 1

Trace E_2

GUID	action	notes
g0	a = 0	initial write
g1	b = 0	initial write
g4	r2 = b	PUR, sees 1 from g3, validated by E_1
g5	a = 1	
g2	r1 = a	sees 1 from g5
g3	b = 1	

Behavior: r1 = 1 and r2 = 1

Trace E_3

GUID	action	notes
g0	a = 0	initial write
g1	b = 0	initial write
g2	r1 = a	sees 0 from g0
g3	b = 1	
g4	r2 = b	sees 0 from g1
g5	a = 1	

Behavior: r1 = 0 and r2 = 0

Trace E_4

GUID	action	notes
g0	a = 0	initial write
g1	b = 0	initial write
g4	r2 = b	sees 0 from g1
g5	a = 1	
g2	r1 = a	sees 1 from g5
g3	b = 1	

Behavior: r1 = 1 and r2 = 0

Figure 2: Sample Execution Traces for Figure 1

4 Theorems

Theorem 4.1 *Correctly synchronized programs have sequentially consistent semantics.*

Proof Two memory accesses are conflicting if they access the same variable and one or both of them are writes. If a program is correctly synchronized, two conflicting accesses are always ordered by a happens-before path. First consider execution traces without any prescient unsynchronized reads. For each read r of a variable v, consider the most recent write w to v. There will be a happens-before path from r to w, and from w to any earlier writes to v. Therefore, the only value r can see is w, which is the same as sequentially consistent semantics.

Furthermore, since each read has a happens-before path to the write whose value it sees, no prescient unsynchronized reads can be validated.

Theorem 4.2 *Reordering a pair of independent statements as allowed by Figure 3 are legal in the memory model. (This is not an exhaustive list of the reorderings possible; just the reorderings proven valid by this theorem).*

Proof Consider a program P, and a modified program P' obtained by reordering two adjacent independent statements of a thread t of P as allowed by Figure 3. For example, we can reorder a read and a following lock, but not a read and a following unlock. In saying

2nd	Read	Write	Lock	Unlock
1st	b	b		
Read a	yes	$a \neq b$	yes	
Write a	$a \neq b$	$a \neq b$	yes	
Lock				
Unlock	yes	yes		

$a \neq b$: reordering allowed if non-aliased

Figure 3: Reorderings proven valid by Theorem 1

that statements are independent, we mean that there are no data or control dependences between these statements preventing their reordering.

For such a reordering to be legal, P' must have no behaviors that are not also behaviors of P. We provide a proof by contradiction for individual reorderings.

Let P' be a modification of program P obtained by a single reordering as described in Theorem 1. Assume E' is an execution of P' such that there is no execution E of P whose behavior $B(E') = B(E)$.

Let x and y be the actions of E' that correspond to the instructions reordered between P and P', with x occurring first in E'. Let t denote the thread executing x and y.

The actions in E' can be written as $[\alpha x \beta y \gamma]$, where α, β and γ are possibly empty sequences of actions, with β not containing any actions corresponding to instructions of thread t.

We give the proof by cases, depending of the kinds of actions that x and y correspond to. In each case, we derive an execution trace E with the following properties:

- $B(E) = B(E')$,
- E respects the unithread semantics of P, and
- E is valid according to the memory model.

x is a read and y is a read or a write Let $E = [\alpha \beta y x \gamma]$. Since each read action is labeled with the value read, $B(E) = B(E')$. We only have to prove that E can be generated by the unithread semantics of P and that the values seen by reads in E are valid according to the memory model.

The order yx corresponds to the order of the corresponding statements in P and there are no other intra-thread reorderings of actions from E' to E so E can be generated by the unithread semantics of P.

There are no happens-before edges between β and either x or y. So the only difference between the happens-before paths in E' and E are that x and y are reordered in paths containing both. If y is a write then y and x are not aliased, so the value seen by the read x is valid in E. For any other read r in E', the sequence of aliased writes on any happens-before path from or to r is unchanged and the value seen by r is valid in E. So the reads in E are valid according to the memory model.

x is a read or a write and y is a write Let $E = [\alpha y x \beta \gamma]$. The order yx corresponds to the order of the corresponding statements in P and there are no other intra-thread reorderings of actions so E can be generated by the unithread semantics of P.

There are no happens-before edges between β and either x or y. So the only difference between the happens-before paths in E' and E are that x and y are reordered in paths containing both. Since y and x are not aliased, for any read r in E', the sequence of aliased writes on any happens-before path from or to r is unchanged and the value seen by r is valid in E.

x is a write and y is a read There are no happens-before edges from β to x, so any reads in β that see the write x are incorrectly synchronized. Let β' be the same as β, except that all reads in β that see the write x are marked as prescient unsynchronized reads validated by E'.

Let $E = [\alpha \beta' y x \gamma]$. Since each read action is labeled with the value read and the read actions in β' are labeled with the same values as the corresponding read actions in β, we know $B(E) = B(E')$.

The order yx corresponds to the order of the corresponding statements in P and there are no other intra-thread reorderings of actions so E can be generated by the unithread semantics of P.

There are no happens-before edges between β and either x or y. So the only difference between the happens-before paths in E' and E are that x and y are reordered in paths containing both. Since y and x are not aliased, for any read r in E', the sequence of aliased writes on any happens-before path from or to r is unchanged and the value seen by r is valid in E.

x is a lock and y is a read or a write (Note: this case corresponds to transforming P to P' by reordering a read or write operation and a following lock operation. Since $E' = [\alpha x \beta y \gamma]$ is an execution of P', the lock action comes first in E').

Let $E = [\alpha \beta y x \gamma]$. The order yx corresponds to the order of the corresponding statements in P and there are no other intra-thread reorderings of actions so E can be generated by the unithread semantics of P.

β can't contain any operations on the same lock as x, since in E' thread t holds the lock during β. Thus there are no happens-before edges between β and either x or y.

The reordering changes the happens-before edge from y to x into a happens-before edge from x to y. This eliminates some happens-before paths from y to actions in α; no other happens-before paths are eliminated or introduced. If y is a read the value seen by y in E' is valid in E, and if y is a write, any PURS in α or β that see y have no happens-before path from y in E.

x is a read or a write and y is a unlock Let $E = [\alpha y x \beta \gamma]$. The order yx corresponds to the order of the corresponding statements in P and there are no other intra-thread reorderings of actions so E can be generated by the unithread semantics of P.

β can't contain any operations on the same lock as y, since in E' thread t holds the lock during β. Thus there are no happens-before edges between β and either x or y.

The reordering changes the happens-before edge from y to x into a happens-before edge from x to y. This eliminates some happens-before paths from actions in γ to x; no other happens-before paths are eliminated or introduced. If x is a read the value seen by x in E' is valid in E, and if x is a write, any reads in γ that see x in E' can also see x in E.

None of the programs P' that result from one of the above reorderings produce a valid execution E' with a behavior that cannot be produced by a valid execution E of the original program P. Therefore, our assumption that such an execution did exist was faulty. Since there is no reordered execution that produces results that cannot be produced by the original ordering, all reorderings examined above are legal.