July 26, 2003, 11:48am

1 Definitions

Programs A program is a set of threads, each of which is a sequence of atomic statements. For the purposes of this model, a statement contains at most one thread or heap operation. For example, a statement may read one heap variable or write one heap variable; it may not increment a heap variable, as this is both a write and a read. *Compound statements* are those which perform multiple operations: they may be broken down into multiple individual statements.

Shared/Heap memory Shared or heap memory can be shared between threads. All instance fields, static fields and array elements are stored in heap memory. Variables local to a method are never shared between threads.

Actions An action is the dynamic counterpart of a program statement. Example actions include reads of and writes to heap locations and locks and unlocks of monitors. Actions may be purely thread local actions (e.g., updating a local variable). Compound actions (e.g., incrementing a heap location) must be broken down into actions consisting of a single atomic operation on the heap.

An action is annotated with information about the execution of that action: the monitor accessed, the value (and corresponding write) read or written, the iteration of the statement, and so on. Each action is further annotated with a globally unique identifier, or GUID, so that it may be uniquely named and referred to.

Variables Read and write actions are annotated with the *variable* read or written; this variable is a memory location into which data may be stored, and from which data may be retrieved. The variable associated with a given action is determined at run time.

Execution Trace An *execution trace* (which we sometimes simply call an *execution*) E of a program P consists of three parts:

- A set of actions.
- A partial order over the actions derived from the statements in P; the partial order is determined by the *happens-before* relationship defined below.
- A prefix of a causal order; the causal order is defined below.

This triple is written as $\langle S, \stackrel{hb}{\rightarrow}, co \rangle$. The behaviors of instructions other than reads are determined strictly by other actions within the thread. Reads of shared variables are more complicated, because the values observed by such a read can be affected by writes to that variable by other threads. Therefore, except for read actions of t in E, all of the actions of t in E must be consistent with a standard, intra-thread execution of t, with each action

occurring in the original program order. If a read action in t observes the value of a write by thread t, that write must be the most recent write by t to that memory location. A read action may observe a value written by another thread; in that case, the values that can be observed are determined by the memory model, as described in Section 2.

An execution trace E is a *valid execution trace* if the actions of each thread obey intra-thread semantics and the values observed by the reads in E are valid according to the memory model (as defined in Section 2, with exceptions as noted). A program's behavior is only legal if it is the result of some execution trace following this rule.

At the beginning of each execution trace, there is an initial write of the default value (i.e., zero or null) to each variable. This is ordered before the first action of each thread.

When we say that the same action occurs in two different execution traces, we mean that there is an action with the same annotations in each execution (e.g., GUID, variable read and value observed).

Happens-before edge If we have two actions x and y, we use $x \stackrel{hb}{\rightarrow} y$ to indicate that x *happens-before* y. Within an execution trace, there is a happens-before edge from each action in a thread t to each following action in t.

There is a total order between all lock and unlock actions on the same monitor. There is a happens-before edge from an unlock action on monitor m to all subsequent lock actions on m (where subsequent is defined according to the total order over the actions on m).

Similarly, there is a total order between writes to and reads from the same *volatile* variable. There is a happens-before edge from each write to a volatile variable v to all subsequent reads of v (where subsequent is defined according to the total order over the actions on v).

Happens-before path There is a happens-before path $x \stackrel{hb}{\rightarrow} y$ from an action x to a later action y if there is a path of happens-before edges from x to y.

2 Memory Model

2.1 Consistency

We first introduce a simple memory model called *consistency*.

The happens-before relationship defines a partial order over the actions in an execution trace; one action is ordered before another in the partial order if one action happens-before the other. We say that a read r of a variable v is *allowed* to observe a write w to v if, in the happens-before partial order of the execution trace:

- r is not ordered before w (i.e., it is not the case that $r \stackrel{hb}{\rightarrow} w$), and
- there is no intervening write w' to v (i.e., no write w' to v such that $w \stackrel{hb}{\rightarrow} w' \stackrel{hb}{\rightarrow} r$).

Informally, a read r is allowed to observe the result of a write w if there is no happens-before ordering to prevent that read. An execution trace is *consistent* if all of the reads in the execution trace are allowed.

Because consistent execution traces do not have causal orders, they are represented by a tuple $\langle S, \stackrel{hb}{\rightarrow} \rangle$.

2.2 Causal Consistency

Consistency is a necessary, but not sufficient, set of constraints. In other words, we need the requirements imposed by Consistency, but they allow for unacceptable behaviors. In particular, they allow for executions in which an action a causes itself to occur. This would have a number of very undesirable effects. To avoid problems such as this, we require that executions be *causal*: no event can cause itself to occur.

For any execution trace, we assume the existence of a *causal order*, which is a total order over some or all of the actions in that execution. The causal order does *not* have to be consistent with the program order or the happens-before order. Any total order over actions in an execution trace is potentially a valid causal order. The causal order could, for example, reflect the order in which the code would occur after compiler transformations have taken place.

The intuition behind causal orders is that for each prefix of that causal order, the next action in the order is *justified* by the actions in the prefix. Actions that do not involve cycles do not need to be explicitly justified. We only require the justification of *prescient* actions. An action x in a trace $\langle S, \stackrel{hb}{\rightarrow}, co \rangle$ is prescient iff either:

- there exists an action y that occurs after x in the causal order such that $y \stackrel{hb}{\rightarrow} x$, or
- x observes a write that occurs after it in the causal order.

All prescient actions must be justified. To justify a prescient action x in trace E, we need to show that the actions before x in the causal order guarantee that x will be allowed.

The formal definition of causality is somewhat involved. We will first define a number of terms; these are summarized in Figure 1.

2.2.1 Action Correspondence

To use causal orders, we must first define the notion of what it means for two actions to correspond to each other in two different executions. Two actions and the causal orders that justify them correspond to each other in separate executions if

- They are generated by the same statement in both,
- The causal orders that justify each are the same length, and
- All of the elements in each of the causal orders are the same, and
- If the i^{th} element of the causal order that justifies one happens before the action, then the i^{th} element of the causal order that justifies the other happens before the action.

Given $E = \langle S, \overset{hb}{\rightarrow}, \alpha x \beta \rangle$, $E' = \langle S', \overset{hb'}{\rightarrow}, \alpha' \beta' x' \gamma' \rangle$,

- $\alpha \preceq \alpha' \iff$
  - $\forall i, 0 \leq i < \text{length}(\alpha) : (\alpha_i, \alpha_{k<i}) \cong (\alpha'_i, \alpha'_{k<i})$
  - $\forall i, 0 \leq i < \text{length}(\alpha)$: all of the information with which α_i is annotated (including the monitor accessed, the iteration of the statement, the variable read or written and the value read or written) is the same as that for α'_i.
- $x \in \text{prescient}_E \iff$
  - $\exists y \in \beta$:
    - $y \overset{hb}{\rightarrow} x \vee$
    - (x is a read) $\wedge$ (y is a write) $\wedge$ (x observes the write performed by y)
- $x \mapsto x' \iff$
  - $(x, \alpha) \cong (x', \alpha')$
  - if x' is a read, it is allowed to observe the same value that a observes
- $(x, \alpha) \cong (x', \alpha') \iff$
  - they are generated by the same statement of P in both
  - $\text{length}(\alpha) = \text{length}(\alpha')$
  - $\forall i, 0 \leq i < \text{length}(\alpha) : (\alpha_i, \alpha_{k<i}) \cong (\alpha'_i, \alpha'_{k<i}) \wedge$
    $\alpha_i \overset{hb}{\rightarrow} x \iff \alpha'_i \overset{hb}{\rightarrow} x' \wedge$
    $x \overset{hb}{\rightarrow} \alpha_i \iff x' \overset{hb}{\rightarrow} \alpha'_i$

Figure 1: Definitions

For two actions a and b, with causal orders that justify them α and β, this is written $(a, \alpha) \cong (b, \beta)$. This is laid out more formally in Figure 1. Additionally, if $(a, \alpha) \cong (b, \beta)$, a and b are reads, and b is allowed to read the same value that a read, we say $a \mapsto b$.

2.2.2 Execution Containment

What it means for one execution to contain another must also be explained for causal orders to work.

The causal order co' of an execution $E = \langle S', \overset{hb'}{\rightarrow}, co' \rangle$ in $valid_k$ contains a causal order co (written $co \preceq co'$) if, for the i^{th} action co_i of co , $(co_i, co_{k<i}) \cong (co'_i, co'_{k<i})$ and all of the information with which co_i is annotated (including the monitor accessed, the variable read or written and the value read or written) is the same as that for co'_i.

2.2.3 Definition of Causal Executions

Consider an execution trace E of a program P, and an action x in E. Let α be the set of actions in E that occur strictly before x in the causal order of E (note that x is not contained in α). Remember that the actions in α do not have to happen before x in an execution. We simply want to say that the set of actions α cause x to occur. Again, the causal order does not have be consistent with the program order, but its results should be determined by the rules for consistent executions.

We have formulated two different ways of expressing this definition. Both definitions are closely related, and we can easily show that they have the same properties. So the choice between them should be determined by whichever approach is easier for more people to understand.

2.2.4 Recursive Approach

For this approach, we build a recursive definition of valid causal orders. To build the set valid_0, we consider each consistent execution E. For each E, there is an execution $\langle S, \overset{hb}{\rightarrow}, co \rangle$ in valid_0 that contains

- A set S containing all of the actions in E,
- The happens-before relationship reflected in E, and
- The empty causal order.

We create a set valid_{k+1} by adding actions to the causal orders of executions in valid_k. An execution $\langle S, \overset{hb}{\rightarrow}, co \rangle$ is legal if it is in some set valid_n; however, the only observable actions of E will be those in the causal order, **not** those in the set S.

We build new executions in valid_{k+1} by appending sets of actions to the causal orders co of executions in valid_k. A set of non-prescient actions can always be added. Alternatively, we may add a single prescient action that is followed in the causal order by a set of non-prescient actions. If the prescient action in this set is $x \in S$, the set can only be appended to co if, in

$$
\begin{aligned}
&\text{valid}_0 \stackrel{def}{\equiv} \{\langle S, \stackrel{hb}{\rightarrow}, \alpha\rangle \mid \langle S, \stackrel{hb}{\rightarrow}\rangle \in \text{consistent} \wedge \alpha = \emptyset\} \\
&\text{valid}_{k+1} \stackrel{def}{\equiv} \text{valid}_k - \text{prohibited}_k \cup \\
&\quad \{E = \langle S, \stackrel{hb}{\rightarrow}, \alpha x \beta\rangle \mid \langle S, \stackrel{hb}{\rightarrow}, \alpha\rangle \in \text{valid}_k - \text{prohibited}_k \wedge \\
&\qquad (b \in \beta) \Rightarrow b \notin \text{prescient}_E \\
&\qquad \wedge (x \in \text{prescient}_E) \Rightarrow \\
&\qquad\quad \textbf{let } (\text{justifiers} = \{E' = \langle S', \stackrel{hb'}{\rightarrow}, \alpha'\rangle \mid \\
&\qquad\qquad (E' \in \text{valid}_k - \text{prohibited}_k) \wedge \\
&\qquad\qquad \alpha \preceq \alpha'\}) \textbf{ in} \\
&\qquad\quad \text{justifiers} \neq \emptyset \wedge \\
&\qquad\quad \forall \langle S', \stackrel{hb'}{\rightarrow}, \alpha'\rangle \in \text{justifiers} : (\exists x' \in \alpha' : x' \mapsto x)\} \\
&\text{prohibited}_k \stackrel{def}{\equiv} \{\langle S, \stackrel{hb}{\rightarrow}, \alpha r \beta\rangle \mid \\
&\quad \langle \langle S, \stackrel{hb}{\rightarrow}, \alpha r \beta\rangle, \langle S', \stackrel{hb'}{\rightarrow}, \alpha' r' \beta'\rangle\rangle \in \text{AE} \wedge \\
&\quad \langle S', \stackrel{hb'}{\rightarrow}, \alpha' r' \beta'\rangle \in \text{valid}_k \wedge \alpha \preceq \alpha' \wedge r \mapsto r' \wedge \\
&\quad r \text{ observes a different write from } r'\}
\end{aligned}
$$

E is a valid execution trace for the memory model if and only if there $\exists k, \text{AE} \cdot \forall j \geq k \cdot E \in \text{valid}_j$. The results of the actions in an execution trace's causal order are the legal results of the execution.

Figure 2: Full Recursive Semantics

each execution that contains *co* in valid_k, *co* allows some x' to occur, where $(x', co) \cong (x, co)$ and $x' \mapsto x$. Here, *allows* is used in the sense of Section 2.1.

Informally, this builds a progressively longer causal order, with every read that observes a write that happens later in the causal order justified independently.

Prohibited Executions On first reading, prohibited executions may be skipped; however, the memory model is not complete without them. For the purposes of showing that a prescient action x is justified, a set of behaviors that are not possible on a particular implementation of a JVM may be specified. This, in turn, allows other actions to be guaranteed and performed presciently, allowing for new behaviors.

In order to justify an execution by prohibiting a certain read, an alternate execution in which the read sees a different write must be demonstrated.

Each application of the semantics is therefore associated with a set of prohibited executions. This list $[P_1, P_2, \ldots P_n]$ is a list of executions that could be disallowed in a particular instance because of a compiler transformation, the architecture of the machine the program is run on, or other characteristics of the JVM.

To formalize this, we associate an alternative execution with each execution we want to prohibit. This alternative execution must always be allowed, and must be the same as the prohibited one up until the prohibited behavior occurs. Formally, we have a pair $\langle \langle S, \stackrel{hb}{\rightarrow}, \alpha r \beta\rangle, \langle S', \stackrel{hb'}{\rightarrow}, \alpha' r' \beta'\rangle\rangle$, where $\langle S, \stackrel{hb}{\rightarrow}, \alpha r \beta\rangle$ is the prohibited execution and $\langle S', \stackrel{hb'}{\rightarrow}, \alpha' r' \beta'\rangle$ is the alternative execution used in its place.

In order to show a particular execution legal, any well formed list of prohibited executions $[P_1, \ldots, P_n]$ may be used.

$$
\begin{array}{l}
\text{prohibited}_0 = \emptyset \\
\text{prohibited}_k = \{P_i \mid 1 \leq i \leq k\} \\
\text{valid}_k = v(\text{prohibited}_k) \\
v(\text{prohibited}) \stackrel{def}{\equiv} \{\langle S, \stackrel{hb}{\rightarrow}, co\rangle \mid \{\langle S, \stackrel{hb}{\rightarrow}\rangle \in \text{consistent} \\
\quad \text{for each prescient action } x \text{ in } co \text{ such that } co = \alpha x \beta, \\
\quad\quad \textbf{let } J = \{E' = \langle S', \stackrel{hb'}{\rightarrow}, \alpha'\beta'\rangle \mid \langle S', \stackrel{hb'}{\rightarrow}\rangle \in \text{consistent} \wedge E' \notin \text{prohibited} \\
\quad\quad\quad\quad \wedge \text{length}(\alpha) = \text{length}(\alpha') \\
\quad\quad\quad\quad \wedge \beta' \text{ does not contain prescient actions} \\
\quad\quad\quad\quad \wedge \alpha \preceq \alpha'\beta'\} \\
\quad\quad\quad \textbf{in } \; J \neq \emptyset \\
\quad\quad\quad\quad \wedge \forall E' = \langle S' :\stackrel{hb'}{\rightarrow}, \alpha'\beta'\rangle \in J \\
\quad\quad\quad\quad\quad\quad \exists x' \in \beta' : x' \mapsto x\}
\end{array}
$$

For a list of prohibited executions $[P_1, \ldots, P_n]$ to be well formed, the following conditions must hold.

$$
\begin{array}{l}
\forall P_i = \langle S_i, \stackrel{hb_i}{\rightarrow}, \alpha_i r_i \beta_i\rangle, \\
\quad \exists E' = \langle S', \stackrel{hb'}{\rightarrow}, \alpha' r' \beta'\rangle \in \text{valid}_{k-1} \\
\quad\quad \text{such that } \alpha \preceq \alpha' \\
\quad\quad\quad \wedge\ r_i \mapsto r' \\
\quad\quad\quad \wedge\ r_i \text{ observes a different corresponding write from } r'\}
\end{array}
$$

Given this, any execution E in valid_n is legal. In addition, any execution of just the actions in a prefix of the causal order of E is also legal, reflecting a situation where due to deadlock or lack of fairness, execution stalls.

Figure 3: Full Iterative Semantics

There are four conditions that allow the model to use an alternative execution:

- It must be a valid execution ($\langle S', \stackrel{hb'}{\rightarrow}, \alpha' r' \beta'\rangle \in \text{valid}_k$)
- It must contain all of the actions that led up to the read ($\alpha \preceq \alpha'$)
- The read must be allowed in the alternative execution ($r \mapsto r'$)
- The read must not see the same write in the alternative execution that it sees in the prohibited one.

The results of the actions in any valid execution's causal order are the legal results of the execution. The full recursive approach can be seen in Figure 2.

2.2.5 Iterative Approach

For this approach, a causal order must contain all of the actions in a given execution from the start. All prescient actions in this order must be justified.

To justify a prescient action x in trace E, we need to show that the actions before x in the causal order guarantee that x will be allowed (in the sense of Section 2.1):

- Define J, the justification for x, to be

$$\begin{aligned} J = \{E' = \langle S', \overset{hb'}{\rightarrow}, \alpha'\beta' \rangle \quad | \quad & E' \text{ is consistent and not prohibited} \\ & \wedge \text{ length}(\alpha) = \text{length}(\alpha') \\ & \wedge \ \alpha \preceq \alpha' \\ & \wedge \ \beta' \text{does not contain prescient actions}\} \end{aligned}$$

- For x to be justified, J must be non-empty and for each $E' = \langle S', \overset{hb'}{\rightarrow}, \alpha'\beta' \rangle$ in J, there must exist an action x' in β' such that $x' \mapsto x$.

Here, all executions that contain fully justified causal orders are legal executions; an execution that can exhibit the behavior of a prefix of a fully justified causal order is also a legal execution.

Prohibited Executions For the purposes of showing that a prescient action x is justified, a set of behaviors that are not possible on a particular implementation of a JVM may be specified. This, in turn, allows other actions to be guaranteed and performed presciently, allowing for new behaviors.

In order to justify an execution by prohibiting a certain read, an alternate execution in which the read sees a different write must be demonstrated.

Each application of the semantics is therefore associated with a set of prohibited executions. This list $[P_1, P_2, \ldots P_n]$ is a list of executions that could be disallowed in a particular instance because of a compiler transformation, the architecture of the machine the program is run on, or other characteristics of the JVM.

Prohibited executions for the iterative method are quite similar (although not identical) to prohibited executions for the recursive method. Define valid_k as the set of execution traces shown to be valid by prohibiting $\{P_1, \ldots, P_k\}$. For a list of prohibited executions to be well formed, for each $P_i = \langle S_i, \overset{hb_i}{\rightarrow}, \alpha_i r_i \beta_i \rangle$, we must demonstrate an $E'_i = \langle S'_i, \overset{hb'_i}{\rightarrow}, \alpha'_i r'_i \beta'_i \rangle$ such that

- E'_i is be in valid_{i-1},
- $\alpha_i \preceq \alpha'_i$,
- $r_i \mapsto r'_i$, and
- r_i must observe a different corresponding write from r'_i.

The full iterative approach can be seen in Figure 3.

2.3 Full Model

The set of executions allowed under the memory model are the set of valid executions allowed by causal consistency, with every possible set of prohibited executions applied separately.