July 24, 2003, 3:27pm

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 read or written, 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 $\langle S, \xrightarrow{hb}, co\rangle$ The behavior 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, $x \xrightarrow{hb} y$ means 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 $x \xrightarrow{hb} y$ 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 $r \xrightarrow{hb} w$), and
- there is no intervening write w' to v (i.e., no write w' to v such that $w \xrightarrow{hb} w' \xrightarrow{hb} r$).

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 $\langle S, \overset{hb}{\rightarrow} \rangle$.

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.

For any execution trace, we assume the existence of a *causal order*, which is a total order over some 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.

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 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
- The i^{th} elements of their 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.

For two actions a and b, this is written $a \cong b$. Additionally, if $a \cong b$, 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 Causal Consistency

Consider an execution trace E of a program P, and an action a in E. Let α be the set of actions in E that occur strictly before a in the causal order of E (note that a is not contained in α). Remember that the actions in α do not have to happen before a in an execution. We simply want to say that the set of actions α cause a 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 build a recursive definition. To build the set valid$_0$, we consider each consistent execution E. For each E, there is an execution $\langle S, \xrightarrow{hb}, co\rangle$ 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 $\langle S, \xrightarrow{hb}, co\rangle$ 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.

An action x in a trace $\langle S, \xrightarrow{hb}, co\rangle$ is prescient iff either:

- there exists an action y that occurs after x in the causal order such that $y \xrightarrow{hb} x$, 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.

If an execution $E = \langle S, \xrightarrow{hb}, co\rangle$ is in valid$_k$, A prescient action $a \in S$ can be appended to co, resulting in a new trace in valid$_{k+1}$ if, in each execution that contains co in $valid_k$, co allows (in the sense of Section 2.1) some a' to occur, where $a' \cong a$ and $a' \mapsto a$.

This begs the question of what it means for one execution to contain the causal order of another. The causal order co' of an execution $E = \langle S', \xrightarrow{hb'}, co'\rangle$ in $valid_k$ contains a causal order co (written $co \preceq co'$) if, for the i^{th} action co_i of co , $co_i \cong co'_i$ 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.

The results of the actions in an execution's causal order are the legal results of the execution.

2.3 Prohibited Sets

The full semantics can prohibit certain executions based on whether they contain a given read. In order to justify an execution by prohibiting a certain read, an alternate execution containing the read must be demonstrated. Each application of the semantics is therefore associated with a set of alternate executions. This set, labeled alternativeExecutions, contains tuples $\langle E, r, E'\rangle$, where E is the execution it is desired to prohibit, r is the read that defines this prohibited execution, and E' is an alternate execution trace that contains r's causal execution trace in E and contains r; however, in E', r returns a different value. If an execution E is prohibited in alternativeExecutions with an alternate E' listed, E' cannot also be prohibited in alternativeExecutions.

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

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

Figure 1: Full Semantics

2.4 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. The formal semantics for a program P are detailed in Figure 1.

Given $E = \langle S, \xrightarrow{hb}, \alpha : x : \beta \rangle$, $E' = \langle S', \xrightarrow{hb'}, \alpha' : x' : \beta' \rangle$,

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

    $\alpha_i \xrightarrow{hb} x \iff \alpha'_i \xrightarrow{hb} x' \wedge$

    $x \xrightarrow{hb} \alpha_i \iff x' \xrightarrow{hb} \alpha'_i$

Figure 2: Definitions