Event Notation

Java Memory Model

Happens-Before

Visibility-based Data Races versus Non-Determinism

Event Notation

We can discuss a program trace as a series of program events (essentially lines of code but we’ll see why this other terminology is used).

We can limit ourselves to looking at some specific types of program events:

- \text{start}(T): \text{thread} \ T \ \text{starts}
- \text{end}(T): \text{thread} \ T \ \text{ends}
- \text{read}(T,x,v): \text{an instruction in thread} \ T \ \text{that reads a value} \ v \ \text{from variable} \ x
- \text{write}(T,x,v): \text{an instruction in thread} \ T \ \text{that writes a value} \ v \ \text{to variable} \ x
- \text{spawn}(T1,T2): \text{thread} \ T1 \ \text{spawns thread} \ T2
- \text{join}(T1,T2): \text{thread} \ T1 \ \text{waits for thread} \ T2 \ \text{to end}
- \text{lock}(T,x): \text{thread} \ T \ \text{acquires a lock on variable} \ x
- \text{unlock}(T,x): \text{thread} \ T \ \text{releases its lock on variable} \ x
The Java Memory Model

Key notions

• Event sequences
  – Programs are understood in terms of events they generate.
  – Events can be reads, writes, lock acquisition, etc.
  – A trace (or event sequences) record reads and writes to memory during an example execution of a program.

• “happens-before”
  – Some events are guaranteed to happen before others
  – Others can be reordered

Intuitively: if an event happens before another, the effect of the first event is visible to the second

Example: Program Sequence Order

Consider following sequential program

```java
public static void main(String[] args) {
    int a = 0;
    int b = 0;
    a = b+1;
}
```

Here is a potential event sequence, the actual program sequence order:

```plaintext
write(main, a, 0)
write(main, b, 0)
read (main, b, 0)
write(main, a, 1)
```
Example: Possible Alternative Order
Consider following sequential program
```java
public static void main(String[] args) {
    int a = 0;
    int b = 0;
    a = b+1;
}
```
Here is a potential event sequence that differs from the actual program sequence order:
```java
write(main, b, 0)
write(main, a, 0)
read (main, b, 0)
write(main, a, 1)
```
Notice though, the result of the code is the same...

---

Example: Possible Optimized Code
Consider following sequential program
```java
public static void main(String[] args) {
    int a = 0;
    int b = 0;
    a = b+1;
}
```
Here is a potential event sequence that differs from the actual program sequence order and has a line optimized out:
```java
write(main, b, 0)
read (main, b, 0)
write(main, a, 1)
```
Notice though, the result is the same and it runs faster due to the removal of a wasted write!
Event Sequences in the Sequential Case

So: what other event sequences are allowed, and what are not?

Java Memory Model specifies an \textit{as-if-serial semantics} for single threads

- Events can be reordered
- Results must remain consistent with program order
  - Precise definition of “consistent” is tricky!
  - For now:
    - Assume terminating threads
    - Then consistent means: same final write events observed for every variable

Concurrency and the JMM

Goal of the JMM: \textit{sequential consistency} for threaded programs that have no data races

- Suppose you have threads $t_1$, $t_2$, $t_3$
- In general, they may execute in parallel, or in an interleaved fashion, etc.
- An execution of $t_1$, $t_2$ and $t_3$ is \textit{sequentially consistent} if it corresponds to executing $t_1$, $t_2$, $t_3$ to completion, one right after the other, in some order (say, $t_2$ first, then $t_3$, then $t_1$)

Definition of JMM guarantees this property for every execution lacking a data race.

Adding lock/unlock adds the re-order rules:

- If a memory access (read or write) precedes an unlock event in the program-order sequence then it must precede the unlock event in any consistent sequence
- If a memory access follows a lock event in the program-order sequence then it must follow that lock event in any consistent sequence
### Code Example with Event Notation

**Initialization:**

\[ x = 0; \]

**Thread 1:**

\[ x = 1; \quad \text{write}(T1, x, 1); \]
\[ y = 2; \quad \text{write}(T1, y, 2); \]

**Thread 2:**

\[ y = x; \quad \text{read}(T2, x, _); \quad \text{write}(T2, y, _); \]

---

### All possible traces of code example!

**Initialization:** \( x = 0; \)

**Thread 1:** \( \text{write}(T1, x, 1); \quad \text{write}(T1, y, 2); \)

**Thread 2:** \( \text{read}(T2, x, _); \quad \text{write}(T2, y, _); \) //cannot “optimize away” this order

\[
\begin{align*}
\text{write}(T1, x, 1); & \quad \text{write}(T1, y, 2); \quad \text{read}(T2, x, _); \quad \text{write}(T2, y, _); \\
\text{write}(T1, x, 1); & \quad \text{read}(T2, x, _); \quad \text{write}(T1, y, 2); \quad \text{write}(T2, y, _); \\
\text{write}(T1, x, 1); & \quad \text{read}(T2, x, _); \quad \text{write}(T2, y, _); \quad \text{write}(T1, y, 2); \\
\text{write}(T1, x, 1); & \quad \text{read}(T2, x, _); \quad \text{write}(T2, y, _); \quad \text{write}(T1, x, 1); \\
\text{read}(T2, x, _); & \quad \text{read}(T1, x, 1); \quad \text{write}(T2, y, _); \quad \text{write}(T1, y, 2); \\
\text{read}(T2, x, _); & \quad \text{write}(T1, x, 1); \quad \text{write}(T2, y, _); \quad \text{write}(T1, y, 2); \\
\text{read}(T2, x, _); & \quad \text{read}(T1, x, 1); \quad \text{write}(T2, y, _); \quad \text{write}(T1, x, 1); \\
\text{read}(T2, x, _); & \quad \text{write}(T1, x, 1); \quad \text{write}(T1, y, 2); \quad \text{write}(T1, x, 1); \\
\text{read}(T2, x, _); & \quad \text{write}(T1, x, 1); \quad \text{write}(T1, y, 2); \quad \text{write}(T1, x, 1); \\
\end{align*}
\]
**All possible values for y by filling in blanks:**

Initialization: \(x=0\);

Thread 1: \(\text{write}(T1,x,1); \text{write}(T1,y,2)\);

Thread 2: \(\text{read}(T2,x,\_); \text{write}(T2,y,\_); \leftarrow\) if we assume \(x\) was volatile, then these blanks can now be filled

\[
\begin{align*}
\text{write}(T1,x,1); & \text{write}(T1,y,2); \text{read}(T2,x,1); \text{write}(T2,y,1); \\
\text{write}(T1,x,1); & \text{read}(T2,x,1); \text{write}(T1,y,1); \text{write}(T2,y,1); \\
\text{write}(T1,x,1); & \text{read}(T2,x,1); \text{write}(T2,y,1); \text{write}(T1,y,2); \\
\text{write}(T1,y,2); & \text{write}(T1,x,1); \text{read}(T2,x,1); \text{write}(T2,y,1); \\
\text{write}(T1,y,2); & \text{read}(T2,x,0); \text{write}(T1,x,1); \text{write}(T2,y,0); \\
\text{write}(T1,y,2); & \text{read}(T2,x,0); \text{write}(T2,y,0); \text{write}(T1,x,1); \\
\text{read}(T2,x,0); & \text{write}(T1,y,2); \text{write}(T1,x,1); \text{write}(T2,y,0); \\
\text{read}(T2,x,0); & \text{write}(T1,y,2); \text{write}(T2,y,0); \text{write}(T1,x,1); \\
\text{read}(T2,x,0); & \text{write}(T2,y,0); \text{write}(T1,x,1); \text{write}(T1,y,2); \\
\text{read}(T2,x,0); & \text{write}(T1,x,1); \text{write}(T2,y,0); \text{write}(T1,y,2); \\
\end{align*}
\]

Depending on order, \(y\) could still have one of three values.

This is an example of a **non-determinism-based race condition**.

---

**Locking Example with Event Notation**

Initialization:

\[x=0;\]

Thread 1:

\[
\begin{align*}
\text{lock}(x); & \quad \text{lock}(T1,x); \\
x = 1; & \quad \text{write}(T1,x,1); \\
\text{unlock}(x); & \quad \text{unlock}(T1,x); \\
\end{align*}
\]

Thread 2:

\[
\begin{align*}
\text{lock}(x); & \quad \text{lock}(T2,x); \\
x = x +1; & \quad \text{read}(T2,x,\_); \text{write}(T2,x,\_); \\
\text{unlock}(x); & \quad \text{unlock}(T2,x); \\
\end{align*}
\]
**Permutation traces of locking example**

Initialization: $x=0$

Thread 1: lock(T1,x); write(T1,x,1); unlock(T1,x);

Thread 2: lock(T2,x); read(T2,x,\_); write(T2,x,\_); unlock(T2,x);

The Java Memory Model prevents the re-ordering of Thread 2’s events.

The locks prevent interleaving of the inner events.

This leaves two valid permutations. **Due to the locks in use, even if $x$ were not marked as volatile, we would still be able to fill in the “\_” values due to the force cache flushes of $x$.**

lock(T1,x); write(T1,x,1); unlock(T1,x); lock(T2,x); read(T2,x,1); write(T2,x,2); unlock(T2,x);

lock(T2,x); read(T2,x,0); write(T2,x,1); unlock(T2,x); lock(T1,x); write(T1,x,1); unlock(T1,x);

There are still two possible values that can be in the last write to $x$, so we clearly have a **non-determinism race condition** in play even with the locks.

However, we do not have a **data race in terms of visibility** because of the locks.

---

**Event Notation Recap**

We can discuss a program trace as a series of program events (essentially lines of code but we’ll see why this other terminology is used).

We can limit ourselves to looking at some specific types of program events:

- **start(T):** thread $T$ starts
- **end(T):** thread $T$ ends
- **read(T,x,v):** an instruction in thread $T$ that reads a value $v$ from variable $x$
- **write(T,x,v):** an instruction in thread $T$ that writes a value $v$ to variable $x$
- **spawn(T1,T2):** thread $T_1$ spawns thread $T_2$
- **join(T1,T2):** thread $T_1$ waits for thread $T_2$ to end
- **lock(T,x):** thread $T$ acquires a lock on variable $x$
- **unlock(T,x):** thread $T$ releases its lock on variable $x$
**Happens-Before logic in terms of visibility**

Let E1 and E2 represent two program events.

Let E1 < E2 be the ordering of events as they appear in a specific trace (a possible valid permutation of the events).

We will define the happens-before ordering <: in a trace as follows:

E1 <: E2 iff E1 < E2 and one of the following holds:

a) thread(E1) = thread(E2)

b) E1 is spawn(T1,T2) and E2 is start(T2)

c) E2 is join(T1,T2) and E1 is end(T2)

d) E1 is unlock(T1,x) and E2 is lock(T2,x)

e) there exists E3 with E1 <: E3 and E3 <: E2

**Visibility and data races**

A visibility-based data race in a specific execution trace sequence exists if the following hold:

1. A sequence has two events (E1, E2) such that one is a write on a variable and the other is either a read or a write of the same variable.
2. There is nothing forcing E1 <: E2 nor is there anything forcing E2 <: E1.

This would mean that the results depend on the runtime decisions in terms of flushing local values back to the shared memory space.
**Our B=A+1; example**

Initialization: A=0; B=0; Then spawn T1 and T2…

Thread 1: A=B; B=A+1;
   `read(T1,B,_); write(T1,A,_); read(T1,A,_); write (T1,B,_);`

Thread 2: if (A>B) then ERR;
   `read(T2,A,_); read(T2,B,_);`

Let’s assume the following execution trace:
   `read(T1,B,_);
   write(T1,A,_);
   read(T1,A,_);
   write (T1,B,_);
   read(T2,A,_);
   read(T2,B,_);`

Do we have a **visibility-based race condition** here?

---

**Terminology**

Going forward, I will try to refer call these visibility-based races as “**data races**” and refer to a “race for a lock” scenario as “**non-determinism**” to avoid ambiguity.

We can examine a specific execution trace directly to look for **data races** in that specific ordering of events.

We would have to examine all valid execution trace sequences (ie: all valid permutations of events) to make sure we find any possible **non-determinism** in our program code.