Data Race Detection

Lecture 11

CS 390

3/20/08
Data Race

- A data race occurs when two concurrently executing threads access a shared variable and when:
  - at least one of the accesses is a write
  - there is no explicit mechanism used to prevent the accesses from being simultaneous

- Meaning of programs with data races depends upon interleaving of thread executions.
  - Sometimes this is ok (when?)
  - Usually, it is not
How can data races be detected and prevented?

- Enforce the use of high-level language mechanisms
  - monitors, synchronized, etc.
  - Monitors: (Hoare 1974)
    - a group of shared variables along with procedures to access them.
    - all accesses protected by the same (anonymous) lock acquired and released upon entry/exit of the monitor
    - shared variable not visible outside monitor
    - lots of issues
      - dynamically allocated data, waiting, exceptions, nesting, ...
Dynamic Approaches

• Happens-before relation
  - partial order on events of all threads in a concurrent execution
  - E.g., basis for Java memory model
  - Between threads, events are ordered according to the synchronization objects they access
Happens-Before

Thread 1
lock(mu);
\[\downarrow\]
v := v+1;
\[\downarrow\]
unlock(mu);

Thread 2
lock(mu);
\[\downarrow\]
v := v+1;
\[\downarrow\]
unlock(mu);
Data Race

- If two threads access a shared variable, and the accesses are not ordered under a happens-before relation, then there is a potential data race.

- Dynamic detection of happens-before violations is difficult:
  - Require per-thread information about concurrent accesses to shared memory
  - Highly dependent upon interleaving induced by scheduler
Happens-Before

Thread 1

\[ y := y+1; \]
\[ \downarrow \]
\[ \text{lock}(mu); \]
\[ \downarrow \]
\[ v := v+1; \]
\[ \downarrow \]
\[ \text{unlock}(mu); \]

Thread 2

\[ \text{lock}(mu); \]
\[ \downarrow \]
\[ v := v+1; \]
\[ \downarrow \]
\[ \text{unlock}(mu); \]
\[ \downarrow \]
\[ y := y+1; \]

Using happens-before, need a large number of test cases to catch the error.

Can we do better?

CS390C: Principles of Concurrency and Parallelism
Lockset Algorithm

- To avoid data races, every shared variable must be protected by some lock.
- A dynamic tool must infer what these locks are.
- For each shared variable v, maintain the set $C(v)$ of candidate locks for v.
  - This set contains those locks that have protected v for the computation so far.
  - Initially, the set holds all possible locks.
  - When v is accessed, compute the intersection of $C(v)$ with the current set of locks held by the thread.
  - If the set is empty, there is no lock that consistently protects v.
Example

\[
\begin{align*}
\text{Program} &: \quad locks\_held & \quad C(v) \\
\text{lock(mu1);} &: \quad \{\} & \quad \{\text{mu1, mu2}\} \\
\quad v := v+1; &: \quad \{\text{mu1}\} \\
\text{unlock(mu1);} &: \quad \{\} \\
\text{lock(mu2);} &: \quad \{\text{mu2}\} \\
\quad v := v+1; &: \quad \{\} \\
\text{unlock(mu2);} &: \quad \{\}
\end{align*}
\]
Improvements

- Common programming practices often violate locking discipline, but are still race free:
  - Initialization
  - Reading shared data
  - Read-write locks:
    - multiple readers, single (exclusive) writer
Initialization

- How can we tell when initialization is complete?
  - Assume initialization is complete if a variable is accessed by a second thread.
  - As long as a variable is only accessed by a single thread, reads and writes have no effect on the candidate lock set.
  - Similar conditions hold for read-only data
State Transition Diagram

Virgin

initial allocation

Exclusive

rd/wr, first thread

wr, new thread

Shared

rd

C(v) updated, but empty set not reported

Shared–Modified

wr

C(v) updated, and races reported when it becomes empty

CS390C: Principles of Concurrency and Parallelism
Read-Write Locks

• For each variable v, some lock m protects v
  - m is held in write mode for every write of v
  - m is held in some mode (read or write) for every read of v

Let locks\_held(t) be the set of locks held in any mode by thread t.
Let write\_locks\_held(t) be the set of locks held in write mode by thread t.
For each v, initialize C(v) to the set of all locks.
On each read of v by thread t,
  set C(v) := C(v) \cap locks\_held(t);
  if C(v) : { }, then issue a warning.
On each write of v by thread t,
  set C(v) := C(v) \cap write\_locks\_held(t);
  if C(v) = { }, then issue a warning.

- Locks held in read mode are removed from the candidate set when a write occurs
  - such locks held by a writer do not protect against a data race between the writer and some other reader thread
Implementation

- Binary instrumentation
  - Each load and store
    - Except loads/stores indirect off the stack
  - Each lock/unlock
  - Storage allocator

- Lockset representation
  - Each set of locks represented as index into a table of lock addresses

- Use a shadow memory to hold lockset index
Implementation

Program memory

&v + shadow offset

Shadow memory

Lockset index table

mu1

mu2

Lock vector

CS390C: Principles of Concurrency and Parallelism
Is Race Detection Enough?

Excerpt from `java.lang.StringBuffer`

```java
public final class StringBuffer {

    public synchronized StringBuffer append(StringBuffer sb) {
        int len = sb.length();
        // other threads may change sb.length(),
        // so len does not reflect the length of sb
        sb.getChars(0, len, value, count);
    }

    public synchronized int length() { ... }
    public synchronized void getChars(...) { ... }

}
```

CS390C: Principles of Concurrency and Parallelism
Atomicity

- A method (or code block) is atomic if for every interleaved execution, there is an equivalent execution with the same overall behavior where the code block is executed serially, without any interleaving.

- Atomicity provides a strong (maximal) guarantee of non-interference between threads.

- How can we check atomicity violations?
If a variable is not consistently protected by the same lock, then the variable access is a non-mover.

If a variable is protected by some lock that is left-leaved with arbitrary actions and then released, again without changing the resulting state, and we classify each lock release operation as a left-mover.

If b is a lock acquire, then c neither acquires or releases the lock. Hence b can be moved to the right of c. (b is a right mover)

If c is a lock release, then b can neither acquire or release the lock. Hence c can be moved to the left of c. (c is a left mover)

Suppose we have a read (or write) of a variable that is shared by multiple threads. If the variable is protected by a lock, then only one thread can access the variable at a time. Such accesses are both movers.

If a variable is not consistently protected by the same lock, then the variable access is a non-mover.
Example

Reduced execution sequence

\[
\begin{align*}
\Sigma_0 & \xrightarrow{acq(m)} \Sigma_1 & b_1 & \xrightarrow{rd(x,0)} \Sigma_2 & b_2 & \xrightarrow{wr(x,1)} \Sigma_3 & b_3 & \xrightarrow{rel(m)} \Sigma_4 \\
\Sigma_0 & \xrightarrow{b_1} \Sigma'_1 & acq(m) & \xrightarrow{rd(x,0)} \Sigma'_2 & wr(x,1) & \xrightarrow{rel(m)} \Sigma'_3 & \xrightarrow{b_2} \Sigma'_4 & \xrightarrow{b_3} \Sigma'_5 & \xrightarrow{b_3} \Sigma'_6 & \xrightarrow{b_3} \Sigma'_7
\end{align*}
\]

acquire operation by thread 1 is a right mover.
read and write operations are left movers
release is a left mover

In general:
a path through a code block that contains a series of right movers, followed by at most one non-mover action, followed by a sequence of left movers is reducible to a serial execution
Formalization

Domains

\[
\begin{align*}
    u, t & \in Tid \\
    x & \in Var \\
    v & \in Value \\
    m & \in Lock \\
    \sigma & \in \text{GlobalStore} = (\text{Var} \to \text{Value}) \cup (\text{Lock} \to (\text{Tid} \cup \{\bot\})) \\
    \pi & \in \text{LocalStore} \\
    \Pi & \in \text{LocalStores} = \text{Tid} \to \text{LocalStore} \\
    \Sigma & \in \text{State} = \text{GlobalStore} \times \text{LocalStores}
\end{align*}
\]

\[A : \text{LocalStore} \to \text{Nat}\]

\[P : \text{Var} \rightarrow \text{Lock}\]

\[\phi : \text{Tid} \to \{\text{InRight}, \text{InLeft}\}\]
Rules

**Instrumented operations:** $\Sigma \xrightarrow{a}_t \phi'$ and $\Sigma \xrightarrow{a}_t \text{wrong}$

**[INS ACCESS PROT]**

$ a \in \{rd(x, v), wr(x, v)\}$  
$P(x)$ defined  
$\sigma(P(x)) = t$

$(\sigma, \phi, \Pi) \xrightarrow{a}_t \phi$

**[INS ACCESS COMMIT]**

$ a \in \{rd(x, v), wr(x, v)\}$  
$P(x)$ undefined  
$A(\Pi(t)) \neq 0$  
$\phi(t) = \text{InRight}$

$(\sigma, \phi, \Pi) \xrightarrow{a}_t \phi[t := \text{InLeft}]$

**[INS ACCESS OUTSIDE]**

$ a \in \{rd(x, v), wr(x, v)\}$

$P(x)$ undefined  
$A(\Pi(t)) = 0$

$(\sigma, \phi, \Pi) \xrightarrow{a}_t \phi$

**[INS ACQUIRE]**

$\phi(t) = \text{InRight}$  
or  
$A(\Pi(t)) = 0$

$(\sigma, \phi, \Pi) \xrightarrow{acq(m)} \phi$

**[INS RELEASE]**

$(\sigma, \phi, \Pi) \xrightarrow{rel(m)} \phi[t := \text{InLeft}]$
Rules

**Instrumented operations:** $\Sigma \Rightarrow_{t}^{a} \phi'$ and $\Sigma \Rightarrow_{t}^{a} wrong$

**[INS ENTER]**

\[
(\sigma, \phi, \Pi) \Rightarrow_{t}^{\text{begin}} \phi[t := InRight]
\]

**[INS OTHER]**

\[
a \in \{\text{end,} \epsilon\} \quad (\sigma, \phi, \Pi) \Rightarrow_{t}^{a} \phi
\]

**[WRONG RACE]**

\[
a \in \{rd(x, v), wr(x, v)\} \\
P(x) \text{ defined} \\
\sigma(P(x)) \neq t
\]

\[
(\sigma, \phi, \Pi) \Rightarrow_{t}^{a} wrong
\]

**[WRONG UNPROTECT]**

\[
a \in \{rd(x, v), wr(x, v)\} \\
P(x) \text{ undefined} \\
A(\Pi(t)) \neq 0 \\
\phi(t) = InLeft
\]

\[
(\sigma, \phi, \Pi) \Rightarrow_{t}^{a} wrong
\]

**[WRONG ACQUIRE]**

\[
A(\Pi(t)) \neq 0 \\
\phi(t) = InLeft
\]

\[
(\sigma, \phi, \Pi) \Rightarrow_{t}^{acq(m)} wrong
\]
Atomizer

- Can use an extended version of the lockset algorithm to infer the locks that should protect variable access.
- Extensions to reduce false alarms:
  - re-entrant locks
  - thread-local locks
  - write-protected data