Principles of Concurrency

Lecture 9
Data Races
The Problem

T1
R(x) //r1
W(x, r1+1)

T2
R(x) //r2
W(x, r2+1)

- If the value of x was initially 0, we would expect that the final value of x after both threads have completed would be 2
- But, the read of x in T1 (T2) can happen concurrently with the write of x in T2 (T1), leading to a final result of 1
- Not sequentially consistent!

An execution contains a data race if it allows two concurrent accesses to the same location, one of which is a write
  - this manifests in a trace by having the events corresponding to these accesses be adjacent to one another
  - equivalently, all conflicting actions (i.e., R-W or W-W actions) are ordered by a happens-before edge
A data race is a symptom of a potential race condition
Happens-Before Relation

A history \( H \) is a trace/interleaving of actions (or events) performed by different threads in a concurrent execution.

Event \( e_1 \) happens-before \( e_2 \) (\( \text{hb}(e_1, e_2) \)) in a history \( H \) if:
- \( e_1 \) and \( e_2 \) are events from the same thread and \( e_1 \) occurs before \( e_2 \) in \( H \)
- \( e_1 \) is a lock.rel() event and \( e_2 \) is lock.acq() event and \( e_1 \) occurs before \( e_2 \) in \( H \)
- if \( \text{hb}(e_1, e_2) \) and \( \text{hb}(e_2, e_3) \) then \( \text{hb}(e_1, e_3) \)

Events \( e_1 \) and \( e_2 \) are in a data race in history \( H \) if \( e_1 \) and \( e_2 \) are:
- not related by \( \text{hb} \)
- generated from different threads
- perform accesses to the same memory location, and one of the accesses is a write
Concurrency Problems

Unintended sharing:

```
int x;

void P () {
  x += ...
}
```

Two threads may call P() concurrently

Atomicity violation:

```
void deposit(int x) {
  int t = balance
  balance = x + t
}

void withdraw(int x) {
  int t = balance
  balance = t - x
}
```

Two threads may incorrectly interleave their actions

Ordering violation:

```
work = ...;
createThread(2);
work = new Work();
ConsumeWork(work)
```

Two threads may incorrectly (not) observe desired actions of the other
Preventing Data Races

Use locks
- concurrent accesses separated by lock/unlock pairs
- challenge: using locks consistently

A compiler views a lock/unlock instruction as potentially modifying any location
- no reorderings around them are permitted
- induce happens-before edge on conflicting accesses
- they enforce strong fences to prevent hardware/software reordering

Other approaches:
- 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 variables not visible outside monitor
  - But, lots of issues: dynamically allocated data, exceptions, signals, nesting, ...
A program is data race-free if none of its possible executions contains a data race.

```
T1
if (R(x) == 1) {
    W(y, 1)
}

T2
if (R(y) == 1) {
    W(x, 1)
}
```

Is this program DRF?

What should the semantics of a program that contains a data race be?

- In C11, Java, Posix, etc. only programs that are DRF are given a semantics. Racy programs have undefined behavior.
- DRF programs, on the other hand, always exhibit SC behavior.
Dynamically compute the happens-before relation
a partial order on events of all threads in a concurrent execution
Between threads, events are ordered according to the synchronization objects they access

Thread 1

```
lock(mu);
v := v+1;
unlock(mu);
```

Thread 2

```
lock(mu);
v := v+1;
unlock(mu);
```
Detecting Data Races

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

- But, dynamic detection of happens-before is difficult
  Requires per-thread information about concurrent accesses to shared memory
  Highly dependent upon interleavings induced by the scheduler
- To construct the happens-before relation, we need:
  program order: the total order of thread instructions
  synchronization order: the total order of accesses to the same synchronization variable
Example

T1

W(x, 1)  
W(y, 1)

T2

if (R(y) == 1) {
    W(x, 2)
}

Both \( x \) and \( y \) have unordered conflicting accesses

But, there is no data race on \( x \), although there is one on \( y \)
Rather than computing a precise happens-before relation, we can approximate its structure

- To avoid data races, every shared variable must be protected by some lock
- To infer what these locks are:
  For each shared variable, maintain a 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 $v$
    - signal a data race
Example

<table>
<thead>
<tr>
<th>Program</th>
<th>locks_held</th>
<th>( C(v) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(mu1);</td>
<td>{}</td>
<td>{mu1,mu2}</td>
</tr>
<tr>
<td>v := v+1;</td>
<td>{mu1}</td>
<td></td>
</tr>
<tr>
<td>unlock(mu1);</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>lock(mu2);</td>
<td>{mu2}</td>
<td></td>
</tr>
<tr>
<td>v := v+1;</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>unlock(mu2);</td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>
Eraser Algorithm

- Assume a database D storing a set of tuples \((m, L)\) where
  - \(m\) is a memory location
  - \(L\) is a set of locks that protect \(m\)
  - Initially, assume \(L\) is the set of all locks in the program
- For a memory event \((R, W)\), generate the tuple \((m, L(t))\) where \(L(t)\) is the set of locks held by \(t\) at the time the event is generated
- Let \((m, L')\) be in \(D\). Then
  - report a race if \(L(t) \cap L' = \phi\)
  - otherwise, replace \((m, L')\) with \((m, L(t) \cap L')\) in \(D\)
Improvements

Common programming practices often violate locking discipline, but are still race free:

- Initialization
- Reading shared data
- Read-write locks:
  - multiple readers, single writer
- multiple locks for an object:

```
T1:
  11.acq()
  ...
  12.acq()
  comp1()
  11.rel()
  ...
  12.rel()

T2:
  12.acq()
  ...
  13.acq()
  comp2()
  12.rel()
  ...
  13.rel()

T3:
  11.acq()
  ...
  13.acq()
  comp3()
  11.rel()
  ...
  13.rel()
```

Can refine lockset algorithm to handle these cases
A vector clock is an array of logical clocks maintained by each thread that records its view of the global state of an execution.
Tracking Happens-Before

Can be used to propagate a global ordering from local views
Application to Data Race Detection

- Each thread maintains a clock $C$ incremented at each lock release operation, as well as a vector clock $C_t(t')$ that records the clock for the last operation performed by $t'$ that happens before the current operation of $t$.

- A global vector clock $L$ is also maintained for every lock $m$:
  - When thread $t'$ releases lock $m$, $L(m) = C_t'$.
  - When $t$ subsequently acquires $m$, $C_t = \max(L(m), C_t)$.

- Two vector clocks ($R_x$ and $W_x$) are also maintained that records the clock of the last read and write to $x$ by a thread $t$:
  - A read from $x$ by thread $t'$ is race-free if $W_x \subseteq C_t'$.
  - A write to $x$ by thread $t'$ is race-free if the write happens after all previous access to the variable, $W_x \subseteq C_t'$ and $R_x \subseteq C_t'$.

$$\forall t. \ VC_1(t) \leq \ VC_2(t)$$
Application to Data Race Detection

$VC_A$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

$VC_B$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

$L_m$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

$W_x$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

$R_x$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

$x = 0$

Write-Write Check: $W_x \subseteq VC_A$?

3 0 ⊆ 4 1? Yes

Read-Write Check: $R_x \subseteq VC_A$?

0 1 ⊆ 4 1? Yes

O(n) time
Application to Data Race Detection

\[
\begin{array}{c|c|c|c|c|c}
VCA & VC_B & L_m & W_x & R_x \\
4 & 1 & 2 & 8 & 2 & 1 \\
4 & 1 & 2 & 8 & 4 & 0 \\
5 & 1 & 4 & 1 & 5 & 1 \\
\end{array}
\]
Application to Data Race Detection

Write-Read Check: $W_x \subseteq VCA$?

$\begin{array}{cccc}
4 & 8 & \subseteq & 5 & 1 \\
\text{No}
\end{array}$
Application to Data Race Detection

- Sound: if the algorithm does not raise a warning, the execution is DRF
- Complete: if the algorithm does raise a warning, then there is an actual data race

As described, the implementation is expensive

- in space overheads (vector clocks for every read and write operation for every location)
- performance - every vector clock operation requires $O(n)$ time where $n$ is the number of threads
- It’s possible to reduce these costs by having a lighter-weight representation (FastTrack algorithm)