Data Race Detection

Assaf Schuster
What is a Data Race?

- Two concurrent accesses to a shared location, at least one of them for writing.
  - Result non-deterministic
  - For some memory models result is undefined
  - Indicative of a bug

Thread 1
- X++
- Z=2

Thread 2
- T=Y
- X++
How Can Data Races be Prevented?

- Explicit synchronization between threads:
  - Locks
  - Critical Sections
  - Barriers
  - Mutexes
  - Semaphores
  - Monitors
  - Events
  - Etc.

Thread 1
1. Lock(m)
2. X++
3. Unlock(m)

Thread 2
1. Lock(m)
2. X++
3. Unlock(m)
Is This Sufficient?

- Yes!
- No!
  - Programmer dependent
    - Correctness – programmer may forget to synch
      - Need tools to detect data races
  - Expensive
    - Efficiency – to achieve correctness, programmer may overdo.
      - Need tools to remove excessive synch’s
#define N 100
Type g_stack = new Type[N];
int g_counter = 0;
Lock g_lock;

void push( Type& obj ){lock(g_lock);...unlock(g_lock);}  
void pop( Type& obj ) {lock(g_lock);...unlock(g_lock);}  
void popAll() {  
    lock(g_lock);  
    delete[] g_stack;  
    g_stack = new Type[N];  
    g_counter = 0;  
    unlock(g_lock);  
}

int find( Type& obj, int number ) {  
    lock(g_lock);  
    for (int i = 0; i < number; i++)  
        if (obj == g_stack[i]) break; // Found!!!  
    if (i == number) i = -1; // Not found... Return -1 to caller  
    unlock(g_lock);  
    return i;  
}

int find( Type& obj ) {  
    return find( obj, g_counter );  
}
#define N 100
Type g_stack = new Type[N];
int g_counter = 0;
Lock g_lock;

void push( Type& obj ){lock(g_lock);...unlock(g_lock);}
void pop( Type& obj ) {lock(g_lock);...unlock(g_lock);}
void popAll( ) {
  lock(g_lock);
  delete[] g_stack;
  g_stack = new Type[N];
  g_counter = 0;
  unlock(g_lock);
}

int find( Type& obj, int number ) { 
  lock(g_lock);
  for (int i = 0; i < number; i++)
    if (obj == g_stack[i]) break; // Found!!
  if (i == number) i = -1; // Not found... Return -1 to caller
  unlock(g_lock);
  return i;
}

int find( Type& obj ) {
  return find( obj, g_counter );
}
Detecting Data Races?

- NP-hard [Netzer & Miller 1990]
  - Input size = # instructions performed
  - Even for 3 threads only
  - Even with no loops/recursion
- Execution orders/scheduling \((#\text{threads})^{\text{thread\_length}}\)
- # inputs
- Detection-code side-effects
- Weak memory, instruction reorder, atomicity...
Apparent Data Races

- Based only on the behavior of the explicit synch
  - not on program semantics
- Easier to detect
- Less accurate (false negatives). Example:

Initially: grades = oldDatabase; updated = false;

Thread T.A.

grades = newDatabase;
updated = true;

while (updated == false);
X:=grades.gradeOf(lecturersSon);

Thread Lecturer
Lamport’s *happens-before* partial order (for a given execution)

- $a, b$ concurrent if neither $a \xrightarrow{hb} b$ nor $b \xrightarrow{hb} a$
  - $\rightarrow$ Apparent data race
  - Otherwise, they are “synchronized”

- **Djit basic idea:** while running (“on-the-fly”) check each access performed against all previously logged accesses
Local Time Frames (LTF)

- The execution of each thread is split into a sequence of *time frames*.
- A new time frame starts on each unlock.
- For every access there is a *timestamp* = a vector of LTFs known to the thread at the moment the access takes place.

<table>
<thead>
<tr>
<th>Thread</th>
<th>LTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>1</td>
</tr>
<tr>
<td>lock( m1 )</td>
<td>1</td>
</tr>
<tr>
<td>z = 2</td>
<td>1</td>
</tr>
<tr>
<td>lock( m2 )</td>
<td>1</td>
</tr>
<tr>
<td>y = 3</td>
<td>1</td>
</tr>
<tr>
<td>unlock( m2 )</td>
<td>1</td>
</tr>
<tr>
<td>z = 4</td>
<td>2</td>
</tr>
<tr>
<td>unlock( m1 )</td>
<td>2</td>
</tr>
<tr>
<td>x = 5</td>
<td>3</td>
</tr>
</tbody>
</table>
## Vector Time Frames

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>write X</td>
<td>(1 1 1)</td>
<td>(1 1 1)</td>
</tr>
<tr>
<td>unlock( m1 )</td>
<td>(2 1 1)</td>
<td>lock( m1 )</td>
</tr>
<tr>
<td>read Z</td>
<td>(2 1 1)</td>
<td>read Y</td>
</tr>
<tr>
<td>unlock( m2 )</td>
<td></td>
<td>unlock( m2 )</td>
</tr>
<tr>
<td>write X</td>
<td></td>
<td>write X</td>
</tr>
<tr>
<td>lock( m2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Time Stamps

From vector clocks lecture:

\[ a \xrightarrow{hb} b \text{ iff } a.\text{LTF}_a < b.\text{timestamp}[t_a] \]

In the example on the right,

\[ \text{timestamp}(b)[t_a] = T_{\text{sync}} \]
### Checking for a Race

**a and b are racing if:**

1. **a and b access same variable**
2. \( a . \text{type} = \text{write} \lor b . \text{type} = \text{write} \)
3. \( a . \text{ltf} \geq b . \text{timestamp}[a . \text{thread_id}] \)
4. \( b . \text{ltf} \geq a . \text{timestamp}[b . \text{thread_id}] \)

\[ \neg (a^{hb} \rightarrow b) \quad \text{not} \]

\[ \neg (b^{hb} \rightarrow a) \quad \text{not} \]

---

**Issues:** Too much storing necessary (long timestamps), too many checks per access (all other accesses).
Optimization 1: Using Sequential Consistency of log

- Suppose access $a$ was logged earlier than $b$
  - Then $\neg (b^{hb} \rightarrow a)$ ... no need to check 😊
  - Will not prove - implied by Sequential Consistency of log

It is left to check that

$$a.ltf \geq b.timestamp[a.thread_id] \quad \neg (a^{hb} \rightarrow b)$$

No need to store full vector timestamps 😊

- $b$'s full timestamp known at the time $b$ happens
- only $a.ltf$ logged and known at the time $b$ happens
Optimization 2:
Checking at most two accesses per variable per ltf

Let:
- Access $a$ by thread $t_1$
- $b$ and $c$ by $t_2$ in same ltf
- $b$ precedes $c$ by program order.

Then:
- If $a$ and $b$ synchronized then $a$ and $c$ are synchronized too.

It is sufficient to store and check only the first read access and the first write access to each variable in each ltf
Algorithm finds a single race
... but have to check write after read

- $a$ by thread $t_1$
- $b$ and $c$ by $t_2$ in same ltf
- $b$ precedes $c$ by program order.
- If $a$ and $b$ synchronized: $a$ and $c$ synchronized too.

- It is sufficient to store and check only the first read access and the first write access to each variable in each ltf.
- Algorithm finds a single race.

<table>
<thead>
<tr>
<th>Thread 2</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock( m )</td>
<td>lock( m )</td>
</tr>
<tr>
<td>write X</td>
<td>write X</td>
</tr>
<tr>
<td>read X</td>
<td>read X</td>
</tr>
<tr>
<td>unlock( m )</td>
<td>unlock( m )</td>
</tr>
</tbody>
</table>

No check no logging
Let:

- $a$ by $t_1$
- $b$ and $c$ by $t_2$, stored in log

Then:

- If $a$ is synch-ed with $c$ then it must also be synch-ed with $b$.

- It is sufficient to check a current access with the most recent accesses in each of the other threads 😊

- Algorithm finds a single race 😞
Access History

For every variable $v$ for each of the threads:

- The last ltf in which the thread read from $v$
- The last ltf in which the thread wrote to $v$

On each first read and first write to $v$ in a ltf every thread updates the access history of $v$

- If the access to $v$ is a read, the thread checks all recent writes by other threads to $v$
- If the access is a write, the thread checks all recent reads as well as all recent writes by other threads to $v$
Djit
Pros and Cons

😊 No false alarms
😊 No missed races (in a given execution)

😡 Very sensitive to differences in scheduling
😡 Requires enormous number of runs. Yet: cannot prove tested program is race free.

- Can be extended to support other synchronization primitives, like barriers, counting semaphores, massages, …
Lockset [Savage et.al. 1997]  
Locking Discipline

- A *locking discipline* is a programming policy that ensures the absence of data-races.

- A simple, yet common locking discipline is to require that every shared variable is protected by a mutual-exclusion lock.

- The *Lockset* algorithm detects violations of locking discipline.

- The main drawback is a possibly excessive number of false alarms.
Djit vs Lockset

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Y = Y + 1;</strong>[^1]</td>
<td><strong>Y = Y + 1;</strong>[^1]</td>
</tr>
<tr>
<td>Lock( m ); V = V + 1; Unlock( m );<strong>[^1]</strong></td>
<td>Lock( m ); V = V + 1; Unlock( m );<strong>[^2]</strong></td>
</tr>
<tr>
<td><strong>[^1] hb → [2]</strong>, yet there is a data-race under different schedule.**</td>
<td><strong>[^2]</strong></td>
</tr>
</tbody>
</table>

**[^1]** Djit will not find until schedule change, Lockset will find

No locking discipline on Y.
Yet [1] and [2] are ordered under all possible schedules.
Lockset will false alarm, Djit will not
The basic Lockset algorithm

- For each shared variable $v$ let $C(v)$ be as set of locks that protected $v$ for the computation so far.
- Let $locks\_held(t)$ at any moment be the set of locks held by the thread $t$ at that moment.
- The Lockset algorithm:
  - for each $v$, init $C(v)$ to the set of all possible locks
  - on each access to $v$ by thread $t$:
    - $C(v) \leftarrow C(v) \cap locks\_held(t)$
    - if $C(v) = \emptyset$, issue a warning
In other words...

- A lock \( m \) is in \( C(\nu) \) at a point if in the execution up to that point, every thread that accessed \( \nu \) held \( m \).
- The process, called *lockset refinement*, ensures that any lock that always protects \( \nu \) is contained in \( C(\nu) \).
- If some lock \( m \) always protects \( \nu \), it will remain in \( C(\nu) \) until execution termination.
Example

<table>
<thead>
<tr>
<th>Program</th>
<th>locks_held</th>
<th>C(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock( m1 ); v = v + 1; Unlock( m1 );</td>
<td>{ }</td>
<td>{m1, m2}</td>
</tr>
<tr>
<td>Lock( m2 ); v = v + 1; Unlock( m2 );</td>
<td>{m1}</td>
<td>{m1}</td>
</tr>
</tbody>
</table>

The locking discipline for v is violated since no lock always protects it.
Business as usual: less checks

\(a\) and \(b\) in same thread, same time frame, \(a\) precedes \(b\), then:

\[
\text{Locks}_a(v) \subseteq \text{Locks}_b(v)
\]

- \(\text{Locks}_u(v)\) is set of locks held during access \(u\) to \(v\).

\[\Rightarrow\text{Only first accesses need be checked in every time frame}\]

\[\Rightarrow\text{Lockset can use same access history as Djit}\]

<table>
<thead>
<tr>
<th>Thread</th>
<th>(\text{Locks}(v))</th>
</tr>
</thead>
<tbody>
<tr>
<td>unlock</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>lock(m1)</td>
<td>{m1}</td>
</tr>
<tr>
<td>(a): write (v)</td>
<td>{m1} = {m1}</td>
</tr>
<tr>
<td>write (v)</td>
<td></td>
</tr>
<tr>
<td>lock(m2)</td>
<td>{m1} \supseteq {m1}</td>
</tr>
<tr>
<td>(b): write (v)</td>
<td></td>
</tr>
<tr>
<td>unlock(m2)</td>
<td></td>
</tr>
<tr>
<td>unlock(m1)</td>
<td></td>
</tr>
</tbody>
</table>
Locking Discipline is too strict

There are three very common programming practices that violate it, yet are free from any data-races:

- **Initialization**: Shared variables are usually initialized without holding any locks.
- **Read-Only (or, Write Once)**: Some shared variables are written during initialization and are read-only thereafter.
- **Read-Write Locks**: when shared variables are protected with these locks, either multiple readers are allowed or a single writer (but not both).
Initialization

- When initializing newly allocated data there is no need to lock it, since other threads can not hold a reference to it yet.

- There is no easy way of knowing when initialization is complete.

- Solution: As long as a variable $\nu$ is accessed by a single thread, accesses don’t update $C(\nu)$.

- Therefore, LockSet is initialized for a variable when it is first accessed by a second thread.
Read-Only Data

- There is no need to protect a variable if it’s read-only.
- To support non-locked read-only accesses, races are reported only after an initialized variable is accessed by more than one thread and is written.
Initialization and Read-Only

- Newly allocated variables begin in the Virgin state. As various threads read and write the variable, its state changes according to the transition above.
- Races are reported only for variables in the Shared-Modified state.
- The algorithm becomes more dependent on the schedule.
Initialization and Read-Only

The states are:

- **Virgin** – Indicates that the data is new and have not been referenced by any other thread.
- **Exclusive** – Entered after the data is first accessed (by a single thread). Subsequent accesses don’t update $C(v)$ (handles initialization).
- **Shared** – Entered after a read access by a new thread. $C(v)$ is updated, but data-races are not reported. In such way, multiple threads can read the variable without causing a race to be reported (handles read-only).
- **Shared-Modified** – Entered when more than one thread access the variable and at least one is for writing. $C(v)$ is updated and races are reported as in original algorithm.
Read-Write Locks

- Many programs use Single Writer/Multiple Readers (SWMR) locks in addition to simple locks.
- The basic algorithm doesn’t support correctly such style of synchronization.
- **Definition:** For a variable $v$, some lock $m$ *protects* $v$ if $m$ is held in write mode for every write of $v$, and $m$ is held in some mode (read or write) for every read of $v$. 
Refinement for Read-Write Locks

- When the variable enters the *Shared-Modified* state:
  - Let $\text{locks\_held}(t)$ be the set of locks held in any mode by thread $t$.
  - Let $\text{write\_locks\_held}(t)$ be the set of locks held in write mode by thread $t$. 
Refinement for Read-Write Locks

The refined algorithm (for Shared-Modified):
- for each $v$, initialize $C(v)$ to the set of all locks
- on each read of $v$ by thread $t$:
  - $C(v) \leftarrow C(v) \cap \text{locks\_held}(t)$
  - if $C(v) = \emptyset$, issue a warning
- on each write of $v$ by thread $t$:
  - $C(v) \leftarrow C(v) \cap \text{write\_locks\_held}(t)$
  - if $C(v) = \emptyset$, issue a warning

Locks held purely in read mode do not protect against data-races between the writer and readers. Thus, they are removed from $C(V)$ when a write occurs.
The refined algorithm will still produce a false alarm in the following simple case:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>C(v)</th>
</tr>
</thead>
</table>
| Lock( m1 );  
\( v = v + 1; \)  
Unlock( m1 );  
Lock( m2 );  
\( v = v + 1; \)  
Unlock( m2 ); | Lock( m1 );  
Lock( m2 );  
\( v = v + 1; \)  
Unlock( m2 );  
Unlock( m1 ); | \{m1,m2\} |
|           |           | \{m1\} |
|           |           | \{}    |
Additional False Alarms

Additional possible false alarms are:

- Queue that implicitly protects its elements by accessing them through locked head and tail fields.
- Thread that passes arguments to a worker thread. Since the main thread and the worker thread never access the arguments concurrently, they do not use any locks to serialize their accesses.
- Privately implemented SWMR locks, which don’t communicate with Lockset.
- True data races that do not affect the correctness of the program (for example “benign” races).

```c
if (f == 0)
    lock(m);
if (f == 0)
    f = 1;
unlock(m);
```
Lockset
Pros and Cons

😊 Less sensitive to scheduling

😊 Detects a superset of all apparently raced locations in an execution of a program: races cannot be missed

😊 Lots (and lots) of false alarms

😊 Still dependent on scheduling: cannot prove tested program is race free
Combining Djit$^+$ and Lockset

- Lockset can detect suspected races in more execution orders.
- Djit can filter out the spurious warnings reported by Lockset.
- Lockset can help reduce number of checks performed by Djit.
  - If $C(v)$ is not empty yet, Djit$^+$ should not check $v$ for races.
- The implementation overhead comes mainly from the access logging mechanism.
  - Can be shared by the algorithms.
The End