Data Race Detection

Assaf Schuster
Previous Lectures

• Concurrent and distributed programming limitations
  • Amdahl’s vs. Gustafson's law
  • Memory architecture: UMA, NUMA and distributed (latency vs. bandwidth)
  • Cache coherence and false sharing

• Problem partitioning/decomposition
  • SIMD vs. MIMD
  • Granularity and load balancing
  • Task-management/synchronization/communication overhead vs. strugglers.

• Concurrent and distributed libraries
  • OpenMP: shared memory
  • MPI: distributed memory (tutorials)

• Virtual time:
  • Order of distributed events and gathering distributed information
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++

LOAD X REG
ADD REG 1
STORE REG X
How Can Data Races be Prevented?

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

• Is this sufficient?

Thread 1
lock (m)
X++

Thread 2
unlock (m)

lock (m)
X++
unlock (m)
Is This Sufficient?

• 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
Where is Waldo?

```c
#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 );
}
```
Can You Find the Race?

```c
#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);}  // write
void pop( Type& obj ) {lock(g_lock); ...; unlock(g_lock);}      // read
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 );
}
```

Similar problem was found in `java.util.Vector`
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
- #user-inputs
- Weak memory, instruction reorder, atomicity...
- Detection-code have side-effects
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);
```
Apparent Data Races (2)

- All raced locations in program $P$
- All apparently raced locations in program $P$
- All shared locations in some program $P$
Detection Approaches

• Restricted pgming model
  • Usually fork-join
• Static
  • Emrath, Padua 88
  • Balasundaram, Kenedy 89
  • Mellor-Crummy 93
  • Flanagan, Freund 01
• Postmortem
  • Netzer, Miller 90, 91
  • Adve, Hill 91
• On-the-fly
  • Dinning, Schonberg 90, 91
  • Savage et.al. 97
  • Itskovitz et.al. 99
  • Perkovic, Keleher 00
  • Choi 02

Issues:
• pgming model
• synch’ method
• memory model
• accuracy
• overhead
• granularity
• coverage
Reminder: The Happened Before Relation

• We say that event $e$ happened before event $e'$ if one of the following properties holds:
  
  - **Processor Order**: $e$ precedes $e'$ in the same process
  - **Send-Receive**: $e$ is a send and $e'$ is the corresponding receive
  - **Transitivity**: exists $e''$ such that $e < e''$ and $e'' < e'$

• Denoted by $e \rightarrow e'$ or $e < e'$ or $e \rightarrow e'_{hp}$

• Two events $e$, $e'$ are said to be **independent** or **concurrent** if:
  
  - not $e \rightarrow e'$ and not $e' \rightarrow e$  
  
  • Denoted by $e \parallel e'$
**Reminder:** Independent/Concurrent Events

- Independent events may or may not happen concurrently, depending on the scheduling.
Djit [Itskovitz et.al. 1999]
Apparent Data Races with Locks

• Lamport’s **happened-before** partial order (for a given execution)

• For two events \( a \) and \( b \), if they have a **happened-before** relation
  • \( \Rightarrow \) They are “synchronized”
• If \( a \) and \( b \) are **concurrent**
  (not \( a \xrightarrow{hp} b \) nor \( b \xrightarrow{hp} a \))
  • \( \Rightarrow \) Apparent **data race**

• Djit basic idea:
  • While running (“on-the-fly”) check each access performed against all previously logged accesses

\[
\begin{array}{c|c}
\text{Thread 1} & \text{Thread 2} \\
\hline
\ldots & \ldots \\
\text{a} & \ldots \\
\ldots & \ldots \\
\text{unlock(L)} & \ldots \\
\ldots & \ldots \\
\ldots & \ldots \\
\end{array}
\]

\[
a \xrightarrow{hp} b
\]
Local Time Frames (LTF) & Time Stamps (TS)

- The execution of each thread is split into a sequence of time frames
- **local time frame (LTF)**
  - A new LTF starts on each **unlock**
- **timestamp (TS)**
  - For every access there is a vector of LTFs (timestamp) 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>2</td>
</tr>
<tr>
<td>z = 4</td>
<td>2</td>
</tr>
<tr>
<td>unlock( m1 )</td>
<td>3</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><strong>Start:</strong> (1 1 1)</td>
<td><strong>Start:</strong> (1 1 1)</td>
<td><strong>Start:</strong> (1 1 1)</td>
</tr>
<tr>
<td>write X</td>
<td>lock (m1)</td>
<td>lock (m2)</td>
</tr>
<tr>
<td>unlock (m1)</td>
<td>read Y (2 1 1)</td>
<td>write X (2 2 1)</td>
</tr>
<tr>
<td>read Z (2 1 1)</td>
<td>unlock (m2)</td>
<td></td>
</tr>
<tr>
<td>Time Stamp (TS)</td>
<td></td>
<td>Local Time Frame (LTF)</td>
</tr>
</tbody>
</table>
Vector Time Compare

- If event $a$ have happened in process $P_i$, we denote:
  \[ a.LTF \triangleq TS(a)[i] \]
- Then, for events $a \neq b$:
  \[ a \xrightarrow{hp} b \iff a.LTF < TS(b)[i] \]
- In the example on the right
  \[ TS(b)[1] = T_{sync} > T_a = a.LTF \]
  \[ \Rightarrow a \xrightarrow{hp} b \]
Checking for a Race

• Events $a$ and $b$ ($a \neq b$) are racing if:
  • $a$ and $b$ access same variable
  • $And (a.type = write \ or \ b.type = write)$
  • $And a.LTF \geq b.timestamp[a.thread_id]$ (i.e., $not (a \xrightarrow{hp} b)$)
  • $And b.LTF \geq a.timestamp[b.thread_id]$ (i.e., $not (b \xrightarrow{hp} a)$)

• 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 $\not \left( b \xrightarrow{hp} a \right)$ ... no need to check 😊
  • True if we assume that the log is Sequentially Consistent - will not prove

• It is left to check that:
  • $a.LTF \geq b.$timestamp$[a.$thread_id$]$ (i.e., $\not \left( a \xrightarrow{hp} b \right)$)

• 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 access to each variable in each LTF

<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
Optimization 2: cont. 1
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 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>write X</td>
<td>lock ( m )</td>
</tr>
<tr>
<td>unlock( m2 )</td>
<td>write X</td>
</tr>
<tr>
<td></td>
<td>lock( m2 )</td>
</tr>
<tr>
<td></td>
<td>write X</td>
</tr>
<tr>
<td></td>
<td>unlock( m )</td>
</tr>
<tr>
<td></td>
<td>lock ( m )</td>
</tr>
<tr>
<td></td>
<td>read X</td>
</tr>
<tr>
<td></td>
<td>unlock( m )</td>
</tr>
</tbody>
</table>
Optimization 2: cont. 2
... but have to check write after read

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

\[\text{It is sufficient to store and check only the first read access and the first write access to each variable in each LTF}\]

- Algorithm finds a single race
Optimization 3: Checking only latest access to same variable per thread

Let:
• Access a by thread $t_1$
• 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
Final Djit Algorithm

• **Access History:** stores 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 \( v \)

For every thread, on each **first read** and **first write** to \( v \) in a LTF:
• 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.
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 (1)

- \([1] \rightarrow [2]\), yet there is a data-race under different schedule change
- Djit will not find until schedule change
+ Lockset will find

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Y = Y + 1;) ([1])</td>
<td>()</td>
</tr>
<tr>
<td>(\text{lock}(m);)</td>
<td>(\text{lock}(m);)</td>
</tr>
<tr>
<td>(V = V + 1;)</td>
<td>(V = V + 1;)</td>
</tr>
<tr>
<td>(\text{unlock}(m);)</td>
<td>(\text{unlock}(m);)</td>
</tr>
<tr>
<td>(Y = Y + 1;) ([2])</td>
<td>(Y = Y + 1;) ([2])</td>
</tr>
</tbody>
</table>
Djit vs. Lockset (2)

- No locking discipline on Y.
- Yet [1] and [2] are ordered under all possible schedules

- Lockset will false alarm
+ Djit will not

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y = Y + 1;[1]</td>
<td>lock(m);</td>
</tr>
<tr>
<td>lock(m); Flag = true;</td>
<td>T = Flag;</td>
</tr>
<tr>
<td>unlock(m);</td>
<td>unlock(m);</td>
</tr>
<tr>
<td>if (T == true)</td>
<td>Y = Y + 1;[2]</td>
</tr>
</tbody>
</table>
The basic Lockset algorithm

• Let $locks\_held(t)$ at any moment be the set of locks held by the thread $t$ at that moment

• For each shared variable $v$ let $C(v)$ be as set of locks that protected $v$ for the computation so far.

• 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 $C(v)$ at a point if in the execution up to that point, every thread that accessed $v$ held $m$
- If some lock $m$ always protects $v$, it will remain in $C(v)$ until execution termination

- The process called lockset refinement
Lockset Example

- The locking discipline for $v$ is violated since no lock always protects it.

<table>
<thead>
<tr>
<th>Program</th>
<th>locks_held</th>
<th>$C(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{lock}(m_1);$</td>
<td>${ }$</td>
<td>${m_1, m_2}$</td>
</tr>
<tr>
<td>$v = v + 1;$</td>
<td>${m_1}$</td>
<td>${m_1}$</td>
</tr>
<tr>
<td>$\text{unlock}(m_1);$</td>
<td>${ }$</td>
<td></td>
</tr>
<tr>
<td>$\text{lock}(m_2);$</td>
<td>${ }$</td>
<td></td>
</tr>
<tr>
<td>$v = v + 1;$</td>
<td>${m_2}$</td>
<td></td>
</tr>
<tr>
<td>$\text{unlock}(m_2);$</td>
<td>${ }$</td>
<td>${ }$</td>
</tr>
</tbody>
</table>
Business as usual: less checks

- a and b in same thread
- same local time frame (LTF)
- a precedes b, then:
  - $Locks_a(v) \subseteq Locks_b(v)$
    - $Locks_u(v)$ is set of locks held during access $u$ to $v$.

- Only first accesses need be checked in every local time frame

<table>
<thead>
<tr>
<th>Thread</th>
<th>Locks(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$unlock$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$lock(m_1)$</td>
<td></td>
</tr>
<tr>
<td>$a$: write $v$</td>
<td></td>
</tr>
<tr>
<td>write $v$</td>
<td></td>
</tr>
<tr>
<td>$lock(m_2)$</td>
<td></td>
</tr>
<tr>
<td>$b$: write $v$</td>
<td></td>
</tr>
<tr>
<td>$unlock(m_2)$</td>
<td></td>
</tr>
<tr>
<td>$unlock(m_1)$</td>
<td></td>
</tr>
</tbody>
</table>

\[ m_1 \subseteq \{m_1\} \]

\[ m_1, m_2 \supseteq \{m_1\} \]
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**: Later...
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 $v$ is accessed by a single thread, accesses don’t update $C(v)$.
• 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, or write-once
• 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 States: **Virgin**

- Indicates that the data is new and have not been referenced by any other thread.
Initialization and Read-Only

States: **Exclusive**

- Entered after the data is first accessed (by a single thread).
- Subsequent accesses don’t update $C(v)$ (handles initialization).
Initialization and Read-Only

States: **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).
Initialization and Read-Only States: **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.
Initialization and Read-Only (summary)

- Races are reported only for variables in the **Shared-Modified** state.
- The algorithm becomes more dependent on the schedule.
Read-Write Locks

• **Read-Write Locks**: when shared variables are protected with these locks, either multiple readers are allowed or a single writer (but not both).
  - 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 \( \mathbf{v} \), some lock \( \mathbf{m} \) protects \( \mathbf{v} \) if \( \mathbf{m} \) is held in write mode for every write of \( \mathbf{v} \), and \( \mathbf{m} \) is held in some mode (read or write) for every read of \( \mathbf{v} \).
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.
Still False Alarms

- 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>
<tbody>
<tr>
<td>lock(m1); v = v + 1; unlock(m1); lock(m2); v = v + 1; unlock(m2);</td>
<td>lock(m1); lock(m2); v = v + 1; unlock(m2); unlock(m1);</td>
<td>{m1,m2}</td>
</tr>
</tbody>
</table>

{m1}
Additional False Alarms

• **Producer consumer**
  • 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.

• True data races that do not affect the correctness of the program (for example “benign” races).

```java
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 detects more suspected races in a single execution than Djit
  - However, Djit incurs less overhead
- Lockset can help reduce number of checks performed by Djit
  - If \( C(v) \) is not empty yet, Djit should not check \( v \) for races

\[
\begin{align*}
S & \quad \text{All shared locations in some program } P \\
L & \quad \text{Violations detected by Lockset in } P \\
D & \quad \text{All apparently raced locations in } P \\
F & \quad \text{Raced locations detected by Djit in } P
\end{align*}
\]