Parallel & Concurrent Programming:
Dynamic Race Detection

Emery Berger
CMPSCI 691W
Spring 2006
Outline

- Last time:
  - Performance + ease of programming
  - Capriccio, Flux

- Today:
  - Race detection
Problem with Races

- Many programs contain races
  - Inadvertent programming errors
  - Failure to observe locking discipline

- Race conditions – insidious bugs
  - Non-deterministic, timing dependent
  - Cause data corruption, crashes
  - Difficult to detect, reproduce, eliminate
A **data race** happens when two threads access a variable simultaneously, and one access is a **write**

```c
int t1;
t1 = hits;
hits = t1 + 1;
```

```c
int t2;
t2 = hits;
hits = t2 + 1;
```
**Data Races**

- A **data race** happens when two threads access a variable simultaneously, and one access is a **write**

```c
int t1;
t1 = hits;
hits = t1 + 1;
```

```c
int t2;
t2 = hits;
hits = t2 + 1;
```
Data Races

- A **data race** happens when two threads access a variable simultaneously, and one access is a **write**

```c
int t1;
t1 = hits;
hits = t1 + 1;

int t2;
t2 = hits;
hits = t2 + 1;
```
Data Races

- Problem with data races: non-determinism
  - Depends on interleaving of threads
- Usual way to avoid data races: mutual exclusion
  - Ensures serialized access
Data Races

Using mutual exclusion:

```
acquire
t1 = hits;
hits = t1 + 1;
release
```

```
acquire
t2 = hits;
hits = t2 + 1;
release
```
**Data Races**

- **Data race types:**
  - Read-write conflict
  - Write-write conflict

```
x = 2;
```

```
x = 3;
a = x;
```
Detecting Races

- Tools to detect data races:
  - **Static** (not today)
  - **Dynamic**
    - Happens-before [Lamport]
    - Locksets [Savage et al.]
Happens-Before

- happens-before (a,b):
  - a immediately precedes b in same thread
    - E.g.: a; b
  - a releases a lock, b acquires it
Using Happens-Before

- Two accesses to shared object without being ordered by happens-before: possible data race

```c
int t1;
t1 = hits;
hits = t1 + 1;

int t2;
t2 = hits;
hits = t2 + 1;
```
**Drawbacks**

- Happens-before – numerous drawbacks
  - Must track per-thread info about concurrent accesses to *every* shared location
  - Depends on scheduler interleaving: can miss races (false negative)
Drawback Example

- Missed race condition by luck

Thread 1

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

Thread 2

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

- Another approach: track locksets
  - Discover which locks are held for every shared object
  - If at any time no locks are held while accessing shared object: data race
- Finds more races than happens-before
Lockset Algorithm

- Each shared variable $\nu$
  - $C(\nu)$ – candidate locks – initially set of all locks
  - Every access to $\nu$
    - $C(\nu) = C(\nu) \cap \text{locks currently held}$
    - lock refinement
    - If $C(\nu) = \emptyset$, data race warning
Lockset Example

Program       locks_held      C(v)

lock(mu1);
{}            {mu1,mu2}

v := v+1;
{mu1}

unlock(mu1);
{}            {mu1}

lock(mu2);
{mu2}

v := v+1;
{}            {mu2}

unlock(mu2);
{}
Lockset Limitations

- Too strict for common synch operations
  - Initialization
    - Usually no lock held
  - Read-shared data
    - Some written during initialization, but only read from then on
    - Safe without locks
- Reader-writer locks
Refined Algorithm

- How do you know when data is completely initialized?
  - Assume initialized when accessed by other thread than creator
- Read-sharing
  - Assume safe until first written
Updated Algorithm

- Initially Virgin
- Exclusive
  - Initialization
- Shared
  - C(v) updated but no race reports
- Shared-Modified
  - As in original algorithm
R/W Locks

- Track locks held only when writing, separately from usual lock checking

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

- Eraser implemented using **ATOM**
  - Binary rewriting tool (Alpha only)
  - Now would be in **Pin**
- Locks represented by *lockset index* into table
  - Locksets = sorted vectors
- **Shadow word** (lockset index + state) for every word in DS & heap
- Instruments every direct memory access
  - 10-30x performance hit
class Account {
    private int balance = 0;

    public int read() {
        int r;
        synchronized(this) {
            r = balance;
        }
        return r;
    }
}

public void deposit(int n) {
    int r = read();
    other threads can update balance
    synchronized(this) {
        balance = r + n;
    }
}
class Account {
    private int balance = 0;

    public int read() {
        int r;
        synchronized(this) {
            r = balance;
        }
        return r;
    }
}

public void deposit(int n) {
    synchronized(this) {
        int r = balance;
        balance = r + n;
    }
}
Race-Freedom Needed?

```java
class Account {
    private int balance = 0;

    public void deposit(int n) {
        synchronized(this) {
            int r = balance;
            balance = r + n;
        }
    }

    public int read() {
        return balance;
    }
}
```

- Race-freedom neither sufficient nor necessary!
The End

Next time:
- Atomicity