Program Analysis
Analyzing Concurrent Programs
(Part 1)
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes
4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008
Why Bother with Concurrency?

- The free lunch provided by Moore’s law is over
  - CPU clock speeds stopped to increase around 2005
  - Instead, multi-core processors became mainstream
  - Need concurrent programs to make full use of the hardware

- Many real-world problems are inherently concurrent, e.g.,
  - Servers must handle multiple concurrent requests
  - Computations done on huge data often are "embarrassingly parallel"
Concurrent Styles

- **Message-passing**
  - Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
  - Used in **actor concurrency model**, e.g., popular in Erlang and Scala
  - No shared memory (ideally), all communication via messages

- **Thread-based, shared memory**
  - Multiple concurrently executing threads
  - All threads access the same shared memory
  - Synchronize via **locks** and **barriers**
Concurrency Styles

- **Message-passing**
  - Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
  - Used in actor concurrency model, e.g., popular in Erlang and Scala
  - No shared memory (ideally), all communication via messages

**Focus of this lecture**

- **Thread-based, shared memory**
  - Multiple concurrently executing threads
  - All threads access the same shared memory
  - Synchronize via locks and barriers
Example

```java
int a = 0, b = 0;
boolean r = false, t = false;

a = 1;
r = true;
t = r;
b = a;
```

What does this program mean?
→ Behavior depends on thread interleaving
1. \(a = 1\)  
   \(r = \text{true}\)  
   \(t = r\)  
   \(b = a\)  
   \(r = \text{true}\)  
   \(t = \text{true}\)  
   \(b = 1\)  

2. \(a = 1\)  
   \(t = r\)  
   \(b = a\)  

3. \(a = 1\)  
   \(t = r\)  
   \(r = \text{true}\)  
   \(b = a\)  
   \(r = \text{true}\)  
   \(t = \text{false}\)  
   \(b = 1\)  

4. \(t = r\)  
   \(a = 1\)  
   \(b = a\)  
   \(r = \text{true}\)  
   \(b = a\)  
   \(t = \text{false}\)  
   \(b = 1\)  

5. \(t = r\)  
   \(a = 1\)  
   \(r = \text{true}\)  
   \(b = a\)  
   \(r = \text{true}\)  
   \(t = \text{false}\)  
   \(b = 1\)  

6. \(t = r\)  
   \(a = 1\)  
   \(b = a\)  
   \(r = \text{true}\)  
   \(b = 1\)  

\(t = \text{true}\) implies \(b = 1\)
Sequential Consistency

Assumption made here:
Programs execute under **sequential consistency**

- **Program order** is preserved: Each thread’s instructions execute in the specified order
- Shared memory behaves like a global array: **Reads and writes are done immediately**

- We assume sequential consistency for the rest of the lecture
- Many real-world platforms have more complex semantics (”memory models”)
What Can Go Wrong?

Common source of errors: **Data races**

- Two accesses to the same shared memory location
- At least one access is a write
- Ordering of accesses is non-deterministic
Example

```java
// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
```

Thread 1 Thread 2
Example

// bank account
int balance = 10;

Thread 1
// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

Read

Thread 2
// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Write

Shared memory location
Example

```java
// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
```

3 races
Example

// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Quiz: What values can balance have after executing this code?
Example

// bank account
int balance = 10;

Thread 1  Thread 2

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Possible outcomes:
balance may be 3, 8, and 15
But: Only 8 is correct
Avoiding Data Races

Use **locks** to ensure that **accesses to shared memory do not interfere**

```
int balance = 10;
Thread 1
acquire(L);
int tmp1 = balance;
balance = tmp1 + 5;
release(L);
Thread 2
acquire(L);
int tmp2 = balance;
balance = tmp2 - 7;
release(L);
```
Avoiding Data Races

Use **locks** to ensure that **accesses to shared memory do not interfere**

```
int balance = 10;

Thread 1
acquire(L);
int tmp1 = balance;
balance = tmp1 + 5;
release(L);
```

```
Thread 2
acquire(L);
int tmp2 = balance;
balance = tmp2 - 7;
release(L);
```

**Same lock ⇒ Mutually exclusive critical sections**
Avoiding Data Races

Use **locks** to ensure that accesses to shared memory **do not interfere**

```java
int balance = 10;

synchronized (L) {
    int tmp1 = balance;
    balance = tmp1 + 5;
}

synchronized (L) {
    int tmp2 = balance;
    balance = tmp2 - 7;
}
```

(Java syntax)