Data Race Detection Technique

Prof. Moonzoo Kim

Computer Science, KAIST
Race Condition in Multithreaded Program

• A multithreaded program has a *race condition* if (1) execution order of certain operations are not fixed, and (2) their execution results are decided by their non-deterministic execution orders.

• Race conditions sometime cause serious errors in SW – e.g. Radiation therapy machine: Therac-25

Q: Is a race condition always problematic?
Harmful Race Condition

Ex. Parallel adder

// adding all numbers in arr[]
long cnt=0 ;
long arr[100] ;
long sum1=0, sum2=0;

main() {
    read(arr, 100) ;
    start(work, &sum1);
    start(work, &sum2);
    print(sum1 + sum2) ;
}

work(long * sum) {
    while (cnt < 100) {
        *sum += arr[cnt] ;
        cnt++ ;
    }
}

Has a race condition?
Is this race condition harmful?
Not Harmful Race Condition

Ex. Seminar room reservation system

```c
1  service() {
2      input(&room, &timeslot) ;
3      if(isAvailable(room, timeslot)){
4          print(“available. continue?”) ;
5          input(&continue) ;
6          if(continue)
7              if(reserve(room, timeslot))
8                  print(“reserved.”) ;
9              else
10                  print(“not available.”) ;
11 } }
```

Is this race condition harmful?

User#1: “Rm01”, “7PM Today”  
System: available. continue?

User#1: “Yes”  
System: not available.

User#2: “Rm01”, “7PM Today”  
System: available. continue?
User#2 “yes”  
System: reserved.
Race Bug

• A *race bug* is a multithreaded program fault that causes race conditions leading to *unintended* program behaviors (i.e. invalid states)

• Race bug detectors detect (predict) race conditions that may violate common concurrency requirements
Data Race

• A *data race* is a pair of *concurrent operations* that read and *write* (or both write) data on a same memory location *without synchronization* (i.e., concurrently without any coordination)

• A data race may violate *sequential consistency* of a target program execution

* L. Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, IEEE Transactions on Computers, 1978
Sequential Consistency

• Lamport’s definition
  “A multiprocessor is *sequentially consistent* if
  
  (1) the result of *any* execution is the same as if the operations of all the processors were executed in *some* sequential order, and
  
  (2) the operations of each individual processor occur in this sequence in the order specified by its program”

• Most intuitive consistency model for programmers
  – Processors see their own loads and stores in program order
  – Processors see others’ loads and stores in program order
  – All processors see same global load and store ordering

A SC preserved program is easy to understand but architects and compiler writers want to violate it for performance

Excerpts from the Prof. Huh’s lecture note on “Consistency”
Data Race Example

class Account {
1  long balance;
   //must be non-negative
2  void withdraw(long x) {
3      if (balance >= x) {
4          synchronized (this) {
5              balance = balance - x;
6          }
7      }
8  }
}
Data Race can Break Sequential Consistency

Initially: \( x = y = 0; \)

data race

data race

data race

thread1:
1: \( x = 1; \)
2: \( t_1 = y; \)

thread2:
3: \( y = 1; \)
4: \( t_2 = x; \)

assert \((t_1 == 1 || t_2 == 1);\)

Does the assertion hold with a SC memory model?
Does the assertion hold with a weak memory model?

Memory Model

• A **memory model** describes the interactions of threads through memory and their shared use of the data (necessary for compiler optimization)
  – A memory model allows a compiler to perform many important optimizations.

• Ex. The Java memory model (a weak memory model) stipulates that changes to the values of shared variables only need to be made visible to other threads when a synchronization barrier (e.g. lock/monitor) is reached.
  – the compiler needs to make sure only that the values of (potentially shared) variables at synchronization barriers are guaranteed to be the same in both the optimized and unoptimized code. In particular, reordering statements in a block of code that contains no synchronization barrier is assumed to be safe by the compiler.

• Most research in the area of memory models revolves around:
  – Designing a memory model that allows a **maximal degree of freedom** for compiler optimizations while still giving sufficient guarantees about race-free and (perhaps more importantly) race-containing programs.
  – Proving program optimizations that are correct with respect to such a memory model.

• [https://en.wikipedia.org/wiki/Memory_model_(programming)](https://en.wikipedia.org/wiki/Memory_model_(programming))
Why is Data Race Harmful? (1/2)

- Sometimes developers intentionally induce data races for efficient read on shared variables (*benign race or dirty read*)
  - e.g. test-test-and-set pattern (a.k.a., double-checked locking)

```java
if (balance>=x) {
    synchronized (this) {
        if (balance>=x) {
            balance=balance-x ;
        }
    }
}
```

* H. J. Boehm: Nondeterminism is unavoidable, but data races are pure evil, RACES Workshop, 2012
Why is Data Race Harmful? (2/2)

However, data races are harmful in most cases

- Execution results are (almost) unpredictable with weak memory models
- Compilers may reorder statements around data races*
- Performance benefit of benign race is really marginal*
- It is bad for maintenance

* H. J. Boehm: Nondeterminism is unavoidable, but data races are pure evil, RACES Workshop, 2012
Data Race Detection/Prediction

• Data races are notoriously difficult to detect
  – Unlike deadlock, the program behavior change by a data race may not be noticeable to users
  – Data races induce errors only under specific thread schedules
  – There are too many shared variables

• There have been two approaches:
  1. Happens-before based detection technique
  2. Lockset algorithm based detection technique
**Happens-Before Example**

```java
class Account {
    long balance;
    //must be non-negative

    void withdraw(long x) {
        if (balance >= x) {
            synchronized (this) {
                balance = balance - x;
            }
        }
    }

    void deposit(long x) {
        synchronized (this) {
            balance = balance + x;
        }
    }
}
```

Initially, `balance: 10`

- `t1: deposit(10)`
  - `11: lock(this)`
  - `12: t = balance` (value: 10)
  - `12: balance = t + 10` (value: 20)
  - `13: unlock(this)`

- `t2: withdraw(15)`
  - `1: if (balance >= 15)`
  - `2: lock(this)`
  - `3: t = balance` (value: 10)
  - `3: balance = t - 15` (value: 5)
  - `4: unlock(this)`

Which execution order is controlled by program/sync?
Which is by chance?
Happens-before Relation (1/2)

• The *happens-before relation* \(<\) is a smallest relation over operations in an execution that satisfies the following conditions:

(1) \(a < b\) when \(a\) and \(b\) are executed by the same thread, and \(a\) comes before \(b\)

(2) \(a < b\) when \(a\) and \(b\) are ordered by the same synchronization entity, and \(a\) comes before \(b\) (e.g. lock, wait/notify, join)

(3) If \(a < b\) and \(b < c\) then \(a < c\)

\(\Rightarrow a\) and \(b\) are *concurrent* if \(a \not< b\) and \(b \not< a\)
Happens-before Relation (2/2)

- Leslie Lamport (Microsoft research)
  - Winner of the 2013 Turing award for advances in reliability of distributed/concurrent systems
  - Happens-before relation, sequential consistency, Bakery algorithm, TLA (temporal logic of actions), and LaTeX

http://amturing.acm.org/
class Account {
    long balance;
    // must be non-negative

    void withdraw(long x) {
        if (balance >= x) {
            synchronized (this) {
                balance = balance - x;
            }
        }
    }

    void deposit(long x) {
        synchronized (this) {
            balance = balance + x;
        }
    }
}

Initially, balance: 10

\(1\): if (balance >= 15)
\(2\): lock (this)

\(3\): t = balance
\(4\): unlock (this)

\(5\): balance = t - 15

(1) \(a < b\) when \(a\) and \(b\) are executed by the same thread,
and \(a\) comes before \(b\)

E.g. \(p_1 < p_2, p_1 < p_3, p_1 < p_4\)

(2) \(a < b\) when \(a\) and \(b\) are ordered by the same lock,
and \(a\) comes before \(b\)

E.g. \(p_1 < q_2\)

(3) If \(a < b\) and \(b < c\) then \(a < c\)

E.g. \(p_1 < q_2 \land q_2 < q_3 \rightarrow p_1 < q_3\)

Data Race Detection Techniques, Prof. Moonzoo Kim
Happens-before Based Detection (1/2)

• The pair of operations $a$ and $b$ is data race if all of the following conditions hold:
  (1) $a$ and $b$ access the same variable, and
  (2) at least one operation is writing, and
  (3) $a \not\prec b$ and $b \not\prec a$

• Several tools such as MultiRace$^1$ and FastTrack$^2$ use this definition for data race detection

---

$^1$ E. Pozniansky et al.: MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs, PPoPP, 2003

Happens-before Based Detection (2/2)

- Happens-before relation provides **precise** reasoning of concurrency of operations (i.e., no false positives)
- However, these techniques may or may not detect data races depending on observed execution scenario (i.e., **false negatives**)
- In addition, tracking happens-before relation induces heavy runtime overhead
Happens-before Based Detection (2/2)

- Happens-before relation provides **precise** reasoning of concurrency of operations (i.e., no false positives)
- However, these techniques may or may not detect data races depending on observed execution scenario (i.e., **false negatives**)
- In addition, tracking happens-before relation induces heavy runtime overhead
Another Execution Scenario

class Account {
    long balance;
    //must be non-negative

    void withdraw(long x) {
        if (balance >= x) {
            synchronized (this) {
                balance = balance - x;
            }
        }
    }

    void deposit(long x) {
        synchronized (this) {
            balance = balance + x;
        }
    }
}

Initially, balance: 10

t1: deposit(10)

11: lock(this)

12: t = balance

12: balance = t + 10

13: unlock(this)

Data race is not detected!

t2: withdraw(5)

1: if (balance >= 5)

2: lock(this)

3: t = balance

3: balance = t - 5

4: unlock(this)
Lockset Based Data Race Detection

• Lock discipline
  – Every access to a shared variable MUST be guarded by at least one lock consistently

• Dynamic data race detector *Eraser* [Savage, SOSP 97]
  – Checks that every shared memory location follows the lock discipline
    • Consider memory locations for global variables, and heap memory locations as shared memory locations
Lockset Algorithm

• Eraser monitors every read/write operation and every lock/unlock operation in an execution

• For each variable $v$, Eraser maintains the lockset $C(v)$, candidate locks for the lock discipline
  – Let $L(t)$ be the set of locks held by thread $t$
  – For each $v$, initialize $C(v)$ to the set of all locks

• For each read/write on variable $v$ by thread $t$
  – $C(v) := C(v) \cap L(t)$
  – If $C(v) = \emptyset$, report that there is a data race for $v$
Lockset Algorithm Example

class Account {
    long balance;
    //must be non-negative

    void withdraw(long x) {
        if (balance >= x) {
            synchronized (this) {
                balance = balance - x;
            }
        }
    }

    void deposit(long x) {
        synchronized (this) {
            balance = balance + x;
        }
    }
}

Initially, balance: 10

L(t1) = ∅,
L(t2) = ∅

C(balance) = {this} ∩ L(t1) = {this}

L(t1) = {this}
C(balance) = {this} ∩ L(t1) = {this}

L(t2) = ∅
C(balance) = {this} ∩ L(t2) = {}
Revisiting False Negative Example

```java
class Account {
    long balance;
    // must be non-negative

    void withdraw(long x) {
        if (balance >= x) {
            synchronized (this) {
                balance = balance - x;
            }
        }
    }

    void deposit(long x) {
        synchronized (this) {
            balance = balance + x;
        }
    }
}
```

Initially, balance: 10

```
t1: deposit(10)
C(balance) = { * } \cap L(t2) = \emptyset

t2: withdraw(5)
```

1: if (balance >= 5)

2: lock (this)

3: t = balance

3: balance = t - 5

4: unlock (this)

11: lock (this)

12: t = balance

12: balance = t + 10

13: unlock (this)
Improving Lockset Algorithm

• The naïve lockset algorithm may generate many false positives and false negatives

• 2 cases that cause false positives
  – **Initialization**
    • A thread writes data on the variable without locking before it makes the variable accessible by other threads
  – **Read-shared variable**
    • After initialization, the variable is only read, and never updated.

• 1 case that causes false negatives
  – **Readers-writer lock (aka. shared-exclusive lock, or multi-reader lock)**
Memory Location State

• Eraser maintains the state for each memory location to check if it is in initialization, and if read-shared.

\[ C(v) = \{\ast\} \]

\[ C(v) \text{ is not updated} \]

\[ C(v) = C(v) \cap L(t) \]
(no bug report)

Initialization

\[ C(v) = C(v) \cap L(t), \text{report if } C(v) = \emptyset \]

Read-shared
Example

```c
int max, iter;
Lock m;

main()
{
    iter = 10;
    max = 0;
    start(f);
    start(f);
}

f()
{
    int i, t;
    for(i=0; i<iter; i++)
    {
        t = input();
        lock(m);
        if(t > max)
        {
            max = t;
        }
        unlock(m);
    }
}
```
### Example

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>L(t0)</th>
<th>L(t1)</th>
<th>L(t2)</th>
<th>C(iter)</th>
<th>S(iter)</th>
<th>C(max)</th>
<th>S(max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(initial state)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iter=10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>max = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start(f)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start(f)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(0&lt;iter)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(0&lt;iter)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock(m)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(t&gt;max)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>max = t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(m)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock(m)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if(t&gt;max)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock(m)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Race Detection Techniques, Prof. Moonzoo Kim 29
Reducing More False Positives

- Use happens-before relation induced by wait/notify and thread start/join to reduce false positives

- Check if one memory location is once used for a variable, and then re-used for another variable
  - For cases where malloc() reuses allocated memory

- Track all references to a memory location to precisely check if multiple threads can access the memory location
  - For cases where global variables become local (e.g., an element of a global list which is removed from the list)
Reducing False Negative

• Check for a set of memory locations assigned for a single variable rather than a single memory location
  – E.g. long, double, array, compound data (struct)
Considering Readers-Writer Locks

• A thread acquires a readers-writer lock either in read-mode or write-mode

• For each variable, Eraser additionally checks if there is a lock consistently held in write-mode for write accesses
  – In Shared-Modified state
    • For each read on variable \( v \) by thread \( t \)
      – \( C(v) := C(v) \cap L(t) \)
      – If \( C(v) = \emptyset \), report that there is a data race for \( v \)
    • For each write on variable \( v \) by thread \( t \)
      – \( C(v) := C(v) \cap LW(t) \)
        » \( LW(t) \) is a set of locks held in a write mode by \( t \)
      – If \( C(v) = \emptyset \), report that there is a data race for \( v \)
Performance Improvement (1/2)

• Dynamic data race detection tools are still too slow to be practical
  – Intel ThreadChecker incurs 100—200x slow down, Google ThreadSanitizer 30--40x, and FastTrack 8.5x in average*

• Approach
  – **Pre-processing**: use static analyses to filter out non-shared variables and read-only variables before runtime monitoring
  – **Hardware assisted monitoring**: use a customized hardware to monitor memory accesses and synchronization with low cost

Performance Improvement (2/2)

• Approach (cond.)
  – **Sampling**: monitor only a subset of operations, or a subset of memory locations
    • *LiteRace* [Marino, PLDI 09] assumes the cold region hypothesis “data races are likely to occur when a thread is executing **cold** (infrequently accessed) region in the program”
    • *Pacer* [Bond, PLDI 10] allows users to configure sampling ratio, and guarantees higher detection ratio for higher sampling ratio.
    • *RACEZ* [Sheng, ICSE 11] exploits performance monitoring unit to obtain partial information on memory accesses with low cost
Next Class: Race Bug Which Is Not a Data Race

class Account {
    long balance;
    //must be non-negative

    void getBalance() {
        synchronized (this) {
            return balance;
        }
    }

    void withdraw(long x) {
        if (getBalance() > x) {
            synchronized (this) {
                balance = balance - x;
            }
        }
    }
}

Initially, balance: 10

1: lock(this)
2: tmp = balance
3: unlock(this)

4: if(tmp>10)
5: lock(this)
6: tmp = balance
6: balance = tmp-10
7: unlock(this)

Data race free, but race bug

1: lock(this)
2: tmp = balance
3: unlock(this)

4: if(tmp>10)
5: lock(this)
6: tmp = balance
6: balance = tmp-10
7: unlock(this)