# Concurrency Analysis for Multithreaded Programs

Soham Chakraborty

20.02.2023

# Recap

Lock-based and lock free programming

Concurrency bugs

Concurrency Primitives

# Outline: Today's Lecture

Data race

Concurrency analysis Techniques

- Data race detection

# Data Race

Event $a$ and $b$ is in data race if:

- $a$ and $b$ are concurrent/in conflict
- $a$ and $b$ access same location
- At least one of $a$ and $b$ is a write

# Data Race

Event $a$ and $b$ is in data race if:

- $a$ and $b$ are concurrent/in conflict
- $a$ and $b$ access same location
- At least one of $a$ and $b$ is a write

**Example Program**

$$\begin{array}{l} X = 1; \\ lock(m); \\ Y = 1; \\ unlock(m); \end{array} \;\Bigg\|\; \begin{array}{l} lock(m); \\ Y = 2; \\ unlock(m); \end{array} \;\Bigg\|\; \begin{array}{l} lock(m); \\ Y = 3; \\ unlock(m); \\ X = 3; \end{array}$$

# Concurrent Accesses

```
                    start      end
event e1 -------------|----------|---->

           start        end
event e2 ----|-----------|------------>

             start  end
event e3 ------|-----|---------------->
```

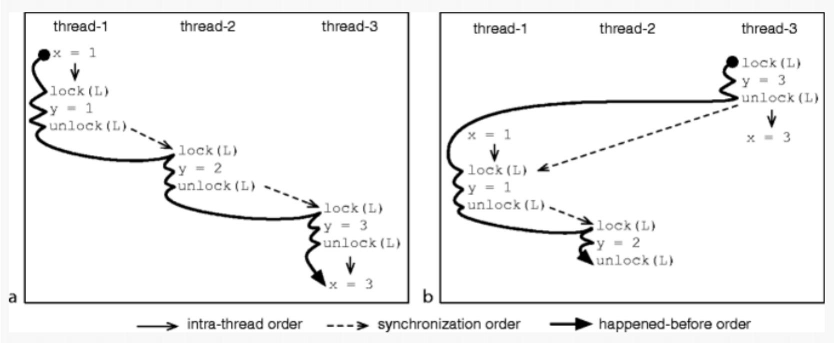
Concurrent: $(e_1, e_2)$, $(e_2, e_3)$

$e_3$ happens-before $e_1$

- $\mathit{end}(e_3) \rightarrow \mathit{start}(e_1)$

# Happens-Before

concurrent/conflict $\Rightarrow$ Not in happens-before (HB) order




Execution 1: No data race

Execution 2: data race on $x$

# Effect of Data Race

- violates programmer's intuition
- May result in arbitrary values
- Program behavior is undefined in many programming languages

# Example

$$class\ \ T\{int\ \ f;\ \cdots\}$$
$$X = 0, y = NULL;$$

$$\begin{array}{l|l}
r = X; & \\
\text{if}(r == 0)\{ & \\
\quad y = new\ \ T(); & \\
\} & X = 1; \\
\text{if}(r == 0)\{ & \\
\quad t = y.f; & \\
\} &
\end{array}$$

# Example

$$
\begin{array}{c}
class \;\; T\{int \;\; f; \;\; \cdots\} \\
X = 0, y = NULL; \\
\begin{array}{l|l}
r = X; & \\
\text{if}(r == 0)\{ & \\
\quad y = new \;\; T(); & \\
\} & X = 1; \\
\text{if}(r == 0)\{ & \\
\quad t = y.f; & \\
\} &
\end{array}
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{c}
class \;\; T\{int \;\; f; \;\; \cdots\} \\
X = 0, y = NULL; \\
\begin{array}{l|l}
r = X; & \\
\text{if}(r == 0)\{ & \\
\quad y = new \;\; T(); & \\
\} & \\
: \;\; // \; spills \;\; r & X = 1; \\
: \;\; // \; re\text{-}read \;\; X & \\
\text{if}(r == 0)\{ & \\
\quad t = y.f; & \\
\} &
\end{array}
\end{array}
$$

# Analysis Techniques

Unsound: bug finding

- Testing – output of an execution
- Dynamic analysis – analysis of an execution
- Predictive analysis – analysis of related executions

Sound: checking correctness

- Model checking – analysis of all possible states/executions
- Static analysis – abstract analysis of all executions

# Dynamic Analysis & Testing

Reasons about one executions

Instruments program

- Should not affect program behavior e.g. thread scheduling

On the fly analysis or trace analysis after execution

# Example

$$
\begin{array}{l||l}
lock(mu1); & lock(mu2); \\
v = v + 1; & v = v + 1; \\
unlock(mu1); & unlock(mu2);
\end{array}
$$

Execution Trace

$lock(mu1);$

$v = v + 1;$

$unlock(mu1);$

$lock(mu2);$

$v = v + 1;$

$unlock(mu2);$

## Lockset algorithm

Let *locks_held*$(t)$ be the set of locks held by thread $t$.
For each $v$, initialize $C(v)$ to the set of all locks.
On each access to $v$ by thread $t$,
  set $C(v) := C(v) \cap$ *locks_held*$(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

## Lockset algorithm

Let $locks\_held(t)$ be the set of locks held by thread $t$.
For each $v$, initialize $C(v)$ to the set of all locks.
On each access to $v$ by thread $t$,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

## Example:

| *Program* | *locks_held* | *C(v)* |
|---|---|---|
| | `{}` | `{mu1,mu2}` |
| `lock(mu1);` | | |
| | `{mu1}` | |
| `v := v+1;` | | |
| | | `{mu1}` |
| `unlock(mu1);` | | |
| | `{}` | |
| `lock(mu2);` | | |
| | `{mu2}` | |
| `v := v+1;` | | |
| | | `{}` |
| `unlock(mu2);` | | |
| | `{}` | |

# Common False Positives

**Initialization:** Shared variables are initialized without holding a lock.

**Read-Sharing:** read-only shared variable (written only during initialization). Read-only variables can be safely accessed without locks.

**Read-Write Locks:** Allows multiple readers but a single writer.

# Observations

If a variable is accessed by a single thread, no effect on analysis

no need to protect a variable if it is read-only

It is possible to refine the algorithm

# Predictive Analysis

Given an execution trace, predictive analysis derive alternative traces.

Predict errors which did not happen in the observed run, but can happen in an alternative execution of the same program.

analysis is on an abstract model extracted from the observed execution

| | T1 | T2 |
|---|---|---|
| 1 | lock(m) | |
| 2 | W(x) | |
| 3 | W(y) | |
| 4 | unlock(m) | |
| 5 | | R(x) |
| 6 | | W(y) |

**Reordering**

- move T2:R(x) before T1:W(x) ✗
- move T2:R(x) after T1:W(x) ✓
- move T2:W(y) before T1:W(y) ✗

A reordering $\sigma'$ of an execution $\sigma$ is allowed if any program $P$ that generates $\sigma$ can also generate $\sigma'$.

$$
\begin{array}{l|l}
lock(m); & if(x == 4)\{ \\
X = 4; & y = 2; \\
y = 1; & \} \\
unlock(m); &
\end{array}
$$

| | T1 | T2 |
|---|---|---|
| 1 | lock(m) | |
| 2 | W(x) | |
| 3 | W(y) | |
| 4 | unlock(m) | |
| 5 | | R(x) |
| 6 | | W(y) |

**Reordering**

- move T2:R(x) before T1:W(x) ✗
- move T2:R(x) after T1:W(x) ✓
- move T2:W(y) before T1:W(y) ✗

A reordering $\sigma'$ of an execution $\sigma$ is allowed if any program $P$ that generates $\sigma$ can also generate $\sigma'$.

## HB-based Race Prediction

Event $a$ and $b$ is in data race if:

- $a$ and $b$ access same location
- At least one of $a$ and $b$ is a write
- $a$ and $b$ are concurrent: $a \not\leq_{\mathsf{hb}} b$ and $b \not\leq_{\mathsf{hb}} a$

Soundness: if a trace has an HB-race, then the race is predictable

Efficient race detection algorithms

# HB-based Race Prediction

| | T1 | T2 |
|---|---|---|
| 1 | W(x) | |
| 2 | lock(m) | |
| 3 | unlock(m) | |
| 4 | | lock(m) |
| 5 | | unlock(x) |
| 6 | | W(x) |

# HB-based Race Prediction

| | T1 | T2 |
|---|---|---|
| 1 | W(x) | |
| 2 | lock(m) | |
| 3 | unlock(m) | |
| 4 | | lock(m) |
| 5 | | unlock(x) |
| 6 | | W(x) |

Predictive race missed by HB

| | T1 | T2 |
|---|---|---|
| 1 | | lock(m) |
| 2 | | unlock(x) |
| 3 | | W(x) |
| 4 | W(x) | |
| 5 | lock(m) | |
| 6 | unlock(m) | |

# Verification

Model checking

Reasons about all executions

Explores state space (enumerative, symbolic)

Static approach, no overhead in runtime

Main challenge: scalability
- Over-approximation & False positives

# Model Checking

Program $P = (X, L, \ell_0, T)$ where

- X: variables
- L: control locations
- $\ell_0 \in L$: initial control location
- $T$: state transition

Transition $(\ell, \rho, \ell')$

- $\rho$ is a constraint with free variables $X \cup X'$

Assignment $x = e$:

$$x' = e \wedge \bigwedge_{y \in X \setminus \{x\}} y' = y$$

Conditional $p$:

$$p \wedge \bigwedge_{x \in X} x' = x$$

# Example

Initially $m = 0$ and $x = 0$

$$
\begin{array}{ll|l}
L_0: & lock(m); & L'_0: \; lock(m); \\
L_1: & x = 1; & L'_1: \; x = 2; \\
L_2: & unlock(m); & L'_2: \; unlock(m); \\
L_3: & & L'_3:
\end{array}
$$

**Transitions in Thread 1**

$\langle L_0, m = 0 \wedge m' = 1 \wedge x' = x, L_1 \rangle$
$\langle L_1, m' = m \wedge x' = 1, L_2 \rangle$
$\langle L_2, m = 1 \wedge m' = 0 \wedge x' = x, L_3 \rangle$

**Transitions in Thread 2**

$\langle L'_0, m = 0 \wedge m' = 1 \wedge x' = x, L'_1 \rangle$
$\langle L'_1, m' = m \wedge x' = 1, L'_2 \rangle$
$\langle L'_2, m = 1 \wedge m' = 0 \wedge x' = x, L'_3 \rangle$

# Example

Initially $m = 0$ and $x = 0$

$$
\begin{array}{ll|ll}
L_0: & lock(m); & L'_0: & lock(m); \\
L_1: & x = 1; & L'_1: & x = 2; \\
L_2: & unlock(m); & L'_2: & unlock(m); \\
L_3: & & L'_3: &
\end{array}
$$

**Transitions in Thread 1**

$\langle L_0, m = 0 \wedge m' = 1 \wedge x' = x, L_1 \rangle$

$\langle L_1, m' = m \wedge x' = 1, L_2 \rangle$

$\langle L_2, m = 1 \wedge m' = 0 \wedge x' = x, L_3 \rangle$

**Transitions in Thread 2**

$\langle L'_0, m = 0 \wedge m' = 1 \wedge x' = x, L'_1 \rangle$

$\langle L'_1, m' = m \wedge x' = 1, L'_2 \rangle$

$\langle L'_2, m = 1 \wedge m' = 0 \wedge x' = x, L'_3 \rangle$

**States**

$\langle L_0, L'_0, m = 0, x = 0 \rangle$

$\langle L_1, L'_0, m = 1, x = 0 \rangle$

$\langle L_2, L'_0, m = 1, x = 1 \rangle$

$\langle L_3, L'_0, m = 0, x = 1 \rangle$

$\langle L_3, L'_1, m = 1, x = 1 \rangle$

$\langle L_3, L'_2, m = 1, x = 2 \rangle$

$\langle L_3, L'_3, m = 0, x = 2 \rangle$

$\langle L_0, L'_1, m = 1, x = 0 \rangle$

$\langle L_0, L'_2, m = 1, x = 2 \rangle$

$\langle L_0, L'_3, m = 0, x = 2 \rangle$

$\langle L_1, L'_3, m = 1, x = 2 \rangle$

$\langle L_2, L'_3, m = 1, x = 1 \rangle$

$\langle L_3, L'_3, m = 0, x = 1 \rangle$

# Example

Initially $m = 0$ and $x = 0$

$$\begin{array}{ll} L_0: & lock(m); \\ L_1: & x = 1; \\ L_2: & unlock(m); \\ L_3: & \end{array} \quad \Bigg\| \quad \begin{array}{ll} L'_0: & lock(m); \\ L'_1: & x = 2; \\ L'_2: & unlock(m); \\ L'_3: & \end{array}$$

**Transitions in Thread 1**

$\langle L_0, m = 0 \wedge m' = 1 \wedge x' = x, L_1 \rangle$
$\langle L_1, m' = m \wedge x' = 1, L_2 \rangle$
$\langle L_2, m = 1 \wedge m' = 0 \wedge x' = x, L_3 \rangle$

**Transitions in Thread 2**

$\langle L'_0, m = 0 \wedge m' = 1 \wedge x' = x, L'_1 \rangle$
$\langle L'_1, m' = m \wedge x' = 1, L'_2 \rangle$
$\langle L'_2, m = 1 \wedge m' = 0 \wedge x' = x, L'_3 \rangle$

**States**

$\langle L_0, L'_0, m = 0, x = 0 \rangle$

$\langle L_1, L'_0, m = 1, x = 0 \rangle$
$\langle L_2, L'_0, m = 1, x = 1 \rangle$
$\langle L_3, L'_0, m = 0, x = 1 \rangle$
$\langle L_3, L'_1, m = 1, x = 1 \rangle$
$\langle L_3, L'_2, m = 1, x = 2 \rangle$
$\langle L_3, L'_3, m = 0, x = 2 \rangle$

$\langle L_0, L'_1, m = 1, x = 0 \rangle$
$\langle L_0, L'_2, m = 1, x = 2 \rangle$
$\langle L_0, L'_3, m = 0, x = 2 \rangle$
$\langle L_1, L'_3, m = 1, x = 2 \rangle$
$\langle L_2, L'_3, m = 1, x = 1 \rangle$
$\langle L_3, L'_3, m = 0, x = 1 \rangle$

Unreachable state:
$\langle L_1, L'_1, \_, \_ \rangle$

# Enumerative Reachability

Enumerate the state-space to check if error state is reachable e.g. SPIN.

**Forward search**

- starts from the initial states
- checks if any error state is reachable

**Backward search**

- starts from the error states
- check if the initial state is reachable

**Challenge**

- state-space explosion
- machine state may have redundant information

**Optimizations**

- Reduction based techniques
- Compositional techniques
- Systematic exploration

**Algorithm: Enumerative Reachability**
**Input** simple program $P = (X, L, T, \ell_0)$, error location $\mathcal{E} \in L$
**Output** SAFE if $P$ is safe w.r.t. $\mathcal{E}$, UNSAFE otherwise

**def** EnumerativeReachability$(P, \mathcal{E})$:
    $reach = \emptyset$
    $worklist = \{(\ell_0, s) \mid s \in v.X\}$
    **while** $worklist \neq \emptyset$ **do:**
        choose$(\ell, s)$ from $worklist$, $worklist = worklist \setminus \{(\ell, s)\}$
        **if** $(\ell, s) \notin reach$:
            $reach = reach \cup \{(\ell, s)\}$
            **foreach** $(\ell, \rho, \ell')$ **in** $T$ **do:**
                add $\{(\ell', s') \mid s' \in Post(s, \rho)\}$ to $worklist$
    **if** exists $(\mathcal{E}, s) \in reach$:
        **return** UNSAFE
    **else:**
        **return** SAFE

$$Post(s, \rho) = \{s' \mid (s, s') \models \rho\}$$

# Other Model Checking Techniques

**Symbolic model checking**

- Algorithms process set of states
- Example: $1 \leq x \leq 10 \wedge 1 \leq y \leq 20$ implies 200 $\{x, y\}$ states

**Bounded model checking**

given program $P$, error location $L$, and $k \in \mathbb{N}$,
construct a constraint which is satisfiable iff
the error location E is reachable within k steps

# Static analysis

Program analysis techniques

+ Interested in reasoning about all executions

+ No overhead in runtime

Scales better

- Main challenges: Dynamic features
  - Dynamic class loading
  - Dynamic dispatch, indirect function call, reflection

- Conservative analysis and over-approximation
  - False positives

# Example

```
public class A{
  private A a;

  public void foo(A fa)
   synchronized (this) {
      System.out.println(fa.a);
   }


  public void bar(A ba) {
   ba.a = new A();
  }

}
```

Assume the references in the public methods alias.

Derive a racy execution

# References


Eraser: A Dynamic Data Race Detector for Multithreaded Programs
Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson. ACM TOCS 1997.

Hans-J. Boehm, "Position Paper: Nondeterminims is Unavoidable, but Data Races are Pure Evil", RACES 2012

Hans-J. Boehm, “How to miscompile programs with "benign" data races”, HotPar 11.

Maximal Causal Models for Sequentially Consistent Systems.
Traian Florin Serbanuta, Feng Chen, Grigore Rosu. In RV’12.

Software Model Checking
Ranjit Jhala, Rupak Majumdar. In ACM Computing Surveys 2009.

A True Positives Theorem for a Static Race Detector.
Nikos Gorogiannis, Peter W. O’Hearn, Ilya Sergey. In POPL’21.