# Concurrency analysis for multithreaded programs

Soham Chakraborty

18.02.2022

# Outline

Analysis Techniques

Data race detection

Atomicity violation detection

# Analysis Techniques

Static analysis

Model checking

Dynamic analysis

Testing

Predictive analysis

# 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

# 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

- abstraction refinement

Reasons about one executions

Instruments program

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

On the fly analysis or trace analysis after execution

# Predictive Analysis

A variant of dynamic analysis

Instrument program to collect a trace

Reasons about related executions
e.g. Given a program with a property $P$, is there an alternative execution that satisfy property $\neg P$?

# Data Race

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

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

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

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

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

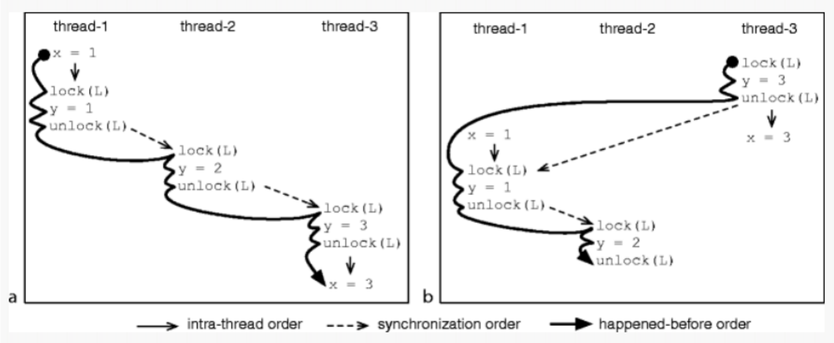
Concurrent: $(e_1, e_2)$, $(e_2, e_3)$

$e_3$ happens-before $e_1$

- $end(e_3) \rightarrow start(e_1)$

# Happens-Before

concurrent/conflict ⇒ Not in happens-before (HB) order




Execution 1: No data race
Execution 2: data race on $x$

# Example

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

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 $C(v) := C(v) \cap \textit{locks\_held}(t)$;
  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

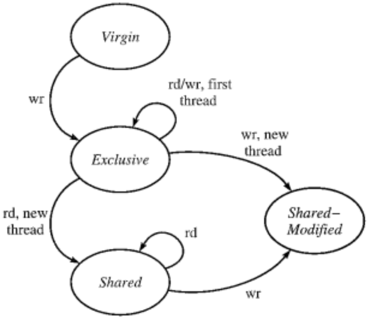
no need to protect a variable if it is read-only

It is possible to refine the algorithm

# State of All Locations

State of each shared variable

Race conditions are issued only in the Shared-Modified state



# Example

| | |
|---|---|
| $int\ v;$<br>$v = 0;$<br>$lock(mu2);$<br>$v = v + 1;$<br>$unlock(mu2);$ | $lock(mu1);$<br>$v = v + 1;$<br>$unlock(mu1);$ |

Execution Trace

$int\ v;$
$v = 0;$
$lock(mu1);$

$v = v + 1;$

$unlock(mu1);$

$lock(mu2);$

$v = v + 1;$

$unlock(mu2);$

# Example

| Program | locks_held | C(v) | State(v) |
|---|---|---|---|
| int v; | {} | {mu1, mu2} | Virgin |
| v = 1024; | | | |
| | | | Exclusive |
| lock(mu1); | | | |
| | {mu1} | | |
| | | | Shared |
| v = v+1; | | | |
| | | | Shared-Modified |
| unlock(mu1) | | {mu1} | |
| | {} | | |
| lock(mu2) | | | |
| | {mu2} | | |
| v = v+1; | | | |
| | | **{}** | |
| | | **Race detected** | |
| unlock(mu2) | | | |
| | {} | | |

# Improved Algorithm

Let *locks_held*$(t)$ be the set of locks held in any mode by thread $t$.
Let *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 $C(v) := C(v) \cap$ *locks_held*$(t)$;
  if $C(v) := \{ \}$, then issue a warning.
On each write of $v$ by thread $t$,
  set $C(v) := C(v) \cap$ *write_locks_held*$(t)$;
  if $C(v) = \{ \}$, then issue a warning.

Warnings are issued only in the Shared-Modified state

# Atomicity Checking

"a method is atomic if its execution is not affected by and does not interfere with concurrently executing threads."
– Atomizer

Dynamic analysis on an execution trace

Execution trace is a state transition system

# Data race vs Atomicity

Absence of data race $\not\Rightarrow$ atomicity

Example from java.lang.StringBuffer

```
public final class StringBuffer {

   public synchronized
         StringBuffer append(StringBuffer sb) {
     int len = sb.length();
     ...  // other threads may change sb.length(),
     ...  // so len does not reflect the length of sb
     sb.getChars(0, len, value, count);
     ...
   }

   public synchronized int length() { ... }
   public synchronized void getChars(...)  { ... }
   ...
}
```

# Multithreaded Program

$$
\begin{array}{rcl}
u, t \in & Tid & \\
x \in & Var & \\
v \in & Value & \\
m \in & Lock & \\
\sigma \in & GlobalStore & = (Var \to Value) \cup (Lock \to (Tid \cup \{\bot\})) \\
\pi \in & LocalStore & \\
\Pi \in & LocalStores & = Tid \to LocalStore \\
\Sigma \in & State & = GlobalStore \times LocalStores
\end{array}
$$

$$
\begin{array}{rcl}
a \in Operation ::= & & rd(x,v) \mid wr(x,v) \\
& \mid & acq(m) \mid rel(m) \\
& \mid & begin \mid end \mid \epsilon
\end{array}
$$

[ACT READ]
$$\frac{\sigma(x) = v}{\sigma \to_t^{rd(x,v)} \sigma}$$

[ACT WRITE]
$$\frac{}{\sigma \to_t^{wr(x,v)} \sigma[x := v]}$$

[ACT OTHER]
$$\frac{a \in \{begin, end, \epsilon\}}{\sigma \to_t^{a} \sigma}$$

[ACT ACQUIRE]
$$\frac{\sigma(m) = \bot}{\sigma \to_t^{acq(m)} \sigma[m := t]}$$

[ACT RELEASE]
$$\frac{\sigma(m) = t}{\sigma \to_t^{rel(m)} \sigma[m := \bot]}$$

State transition: $\Sigma_0 \xrightarrow{act_1} \Sigma_1 \xrightarrow{act_2} \ldots$

Each thread has serial execution

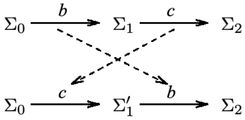
The actions from the serial executions interleave

# Reduction

Consider actions from concurrently running threads

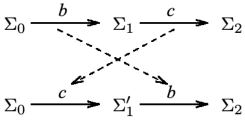
The actions can reorder without affecting the program state

Example:



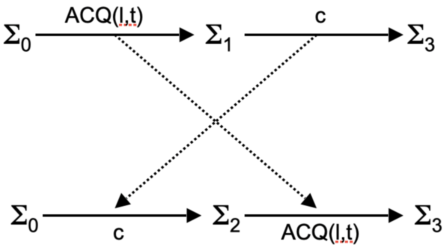
# Right and Left Movers

Example:



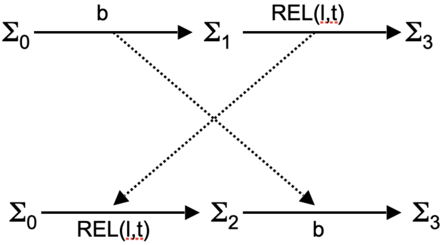

$b$ is a right-mover action (R) and $c$ is a left mover action (L)

# Mover Actions

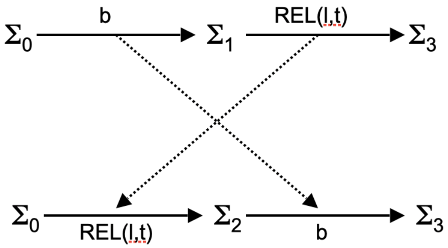



ACQ is right mover

# Mover Actions




REL is left mover

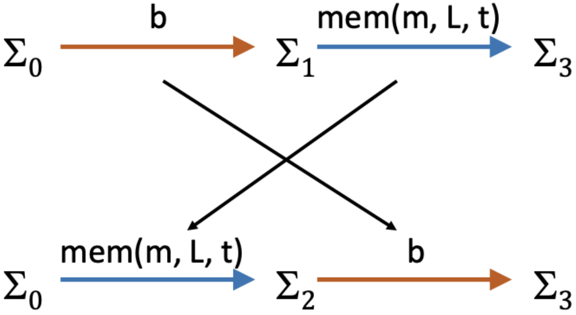



REL is left mover

# Both Mover Action

Both-mover (B): every access of a well-protected shared variable

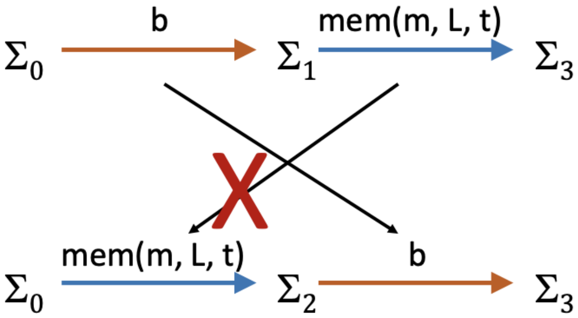
- Race free access




Shared variable m is always protected by lockset L

thread t holds at least one lock in L during the access to m

Non-mover (N): access of a variable for which all accesses are not well-protected



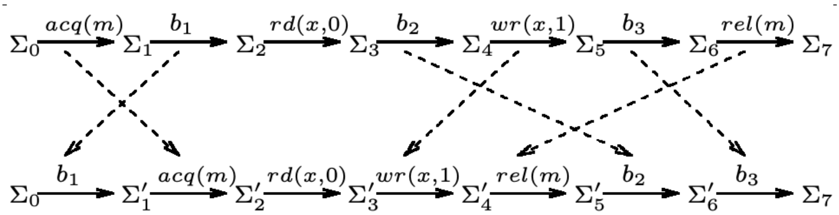

Shared variable m is always protected by lockset L

thread t holds no lock in L during the access to m

# Example: Atomicity Checking

1. acquires a lock *m*,
2. reads a variable *x* and then writes *x* (protected by *m*)
3. release *m*

Execution path is interleaved with actions from other threads




the thread has a serial execution which does not interleave with other threads

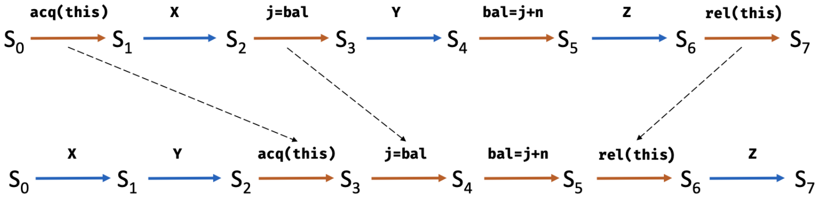
- Satisfies atomicity

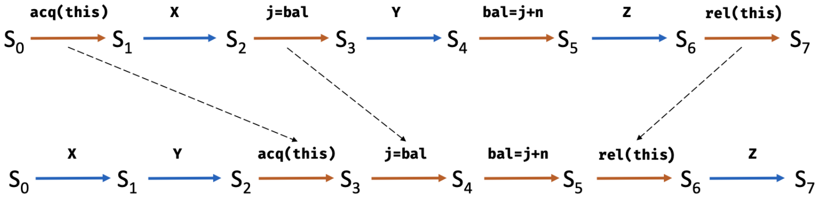
$$\Sigma_0 \xrightarrow{acq(lock)} \Sigma_1 \xrightarrow{j=bal} \Sigma_2 \xrightarrow{bal=j+n} \Sigma_3 \xrightarrow{rel(lock)}$$

$$\Downarrow$$

$$\Sigma_0 \xrightarrow{R} \Sigma_1 \xrightarrow{B} \Sigma_2 \xrightarrow{B} \Sigma_3 \xrightarrow{L}$$

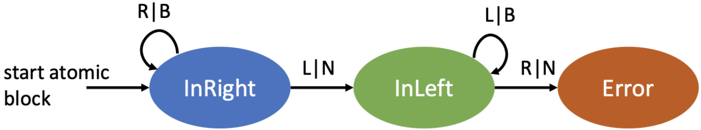
# Reduction Method






Atomicity checking:

Reducible methods: $(R \mid B)^*[N](L \mid B)^*$



# Atomizer Algorithm

Instrumented code calls Atomizer runtime

Lockset algorithm identifies races

- classify movers/non-movers

Atomizer checks reducibility of atomic blocks

- If not reducible: warns about atomicity violations

# Atomizer Algorithm

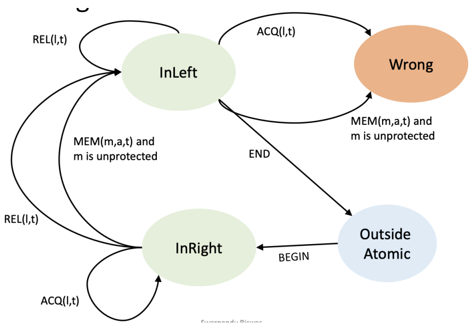
Instrumented code calls Atomizer runtime

Lockset algorithm identifies races

- classify movers/non-movers

Atomizer checks reducibility of atomic blocks

- If not reducible: warns about atomicity violations



# References

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

A Dynamic Atomicity Checker for Multithreaded Programs.
C Flanagan and S. Freund.
POPL 2004.

Analysis of Concurrent Programs
Swarnendu Biswas
CS 636, Semester 2020-2021-II, IIT Kanpur

Research on Atomicity
Cormac Flanagan
https://users.soe.ucsc.edu/ cormac/atom.html