


# Eraser

## A Dynamic Data Race Detector for Multithreaded Programs

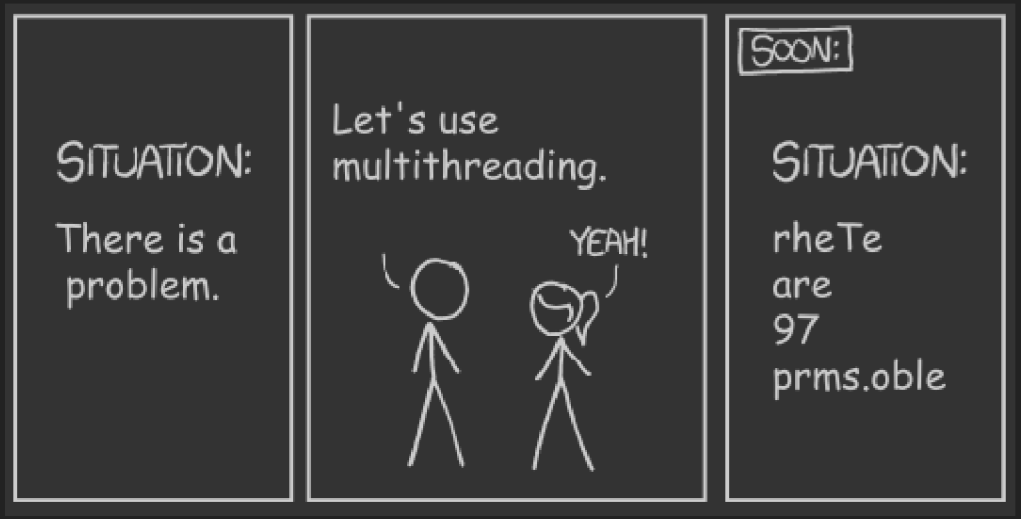
Fabien Savy - 11/12/2021

# Outline

- Motivation
- Background
- Design & Implementation
- Experiments
- Discussion

# Multithreading is hard

- time-dependent data races
- hard to debug and time consuming




See also:

- Lottery scheduling
- Scheduler activations

Comic from r/ProgrammerHumor

# Data race recipe

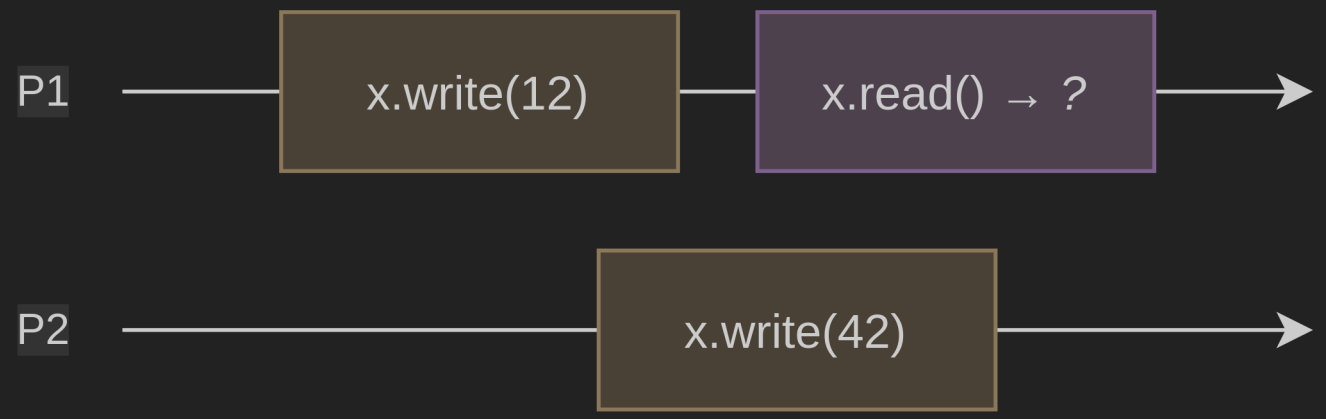
## Ingredients

- At least two concurrent threads
- A shared variable $v$

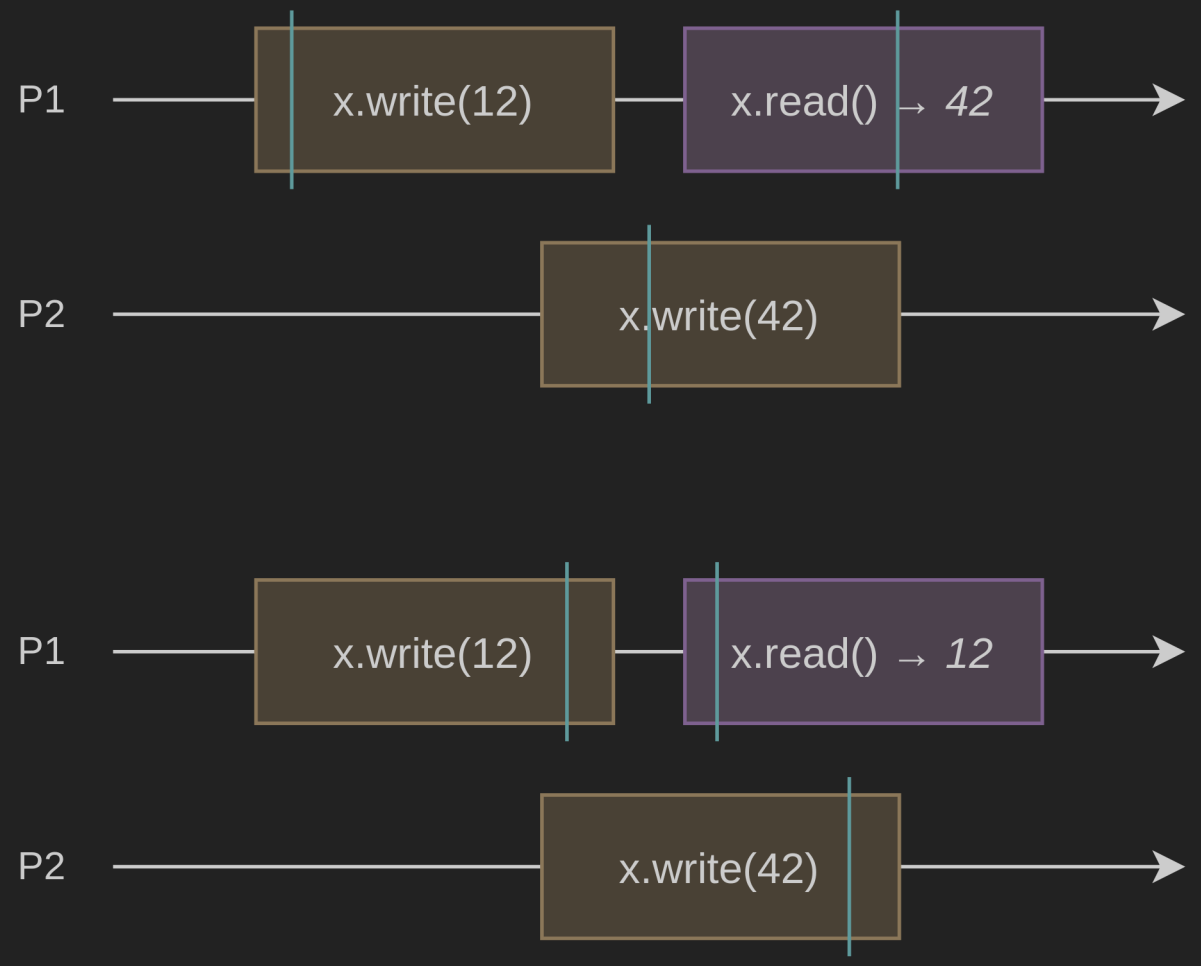
## Steps

- Don't use synchronization mechanisms
- Access $v$ concurrently while a thread is **writing**

# Data race example

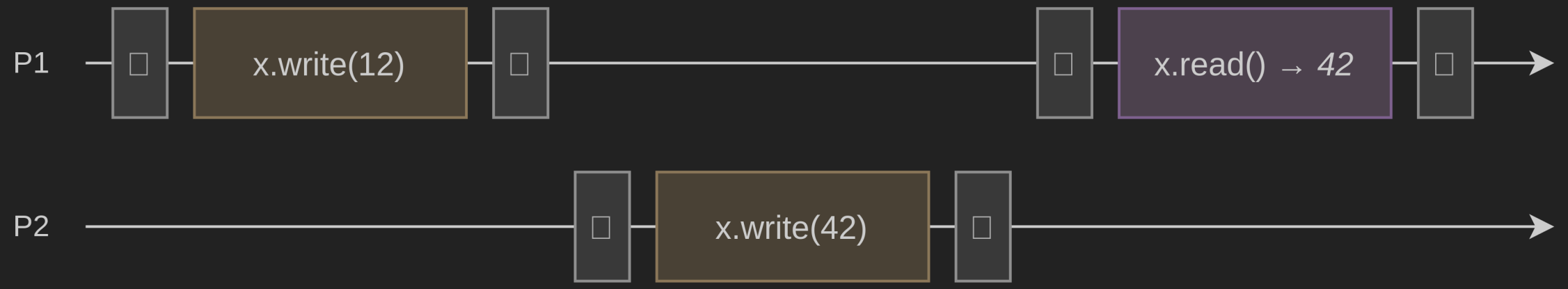


# Data race example: possible outcomes



# Synchronization primitives

## Locks



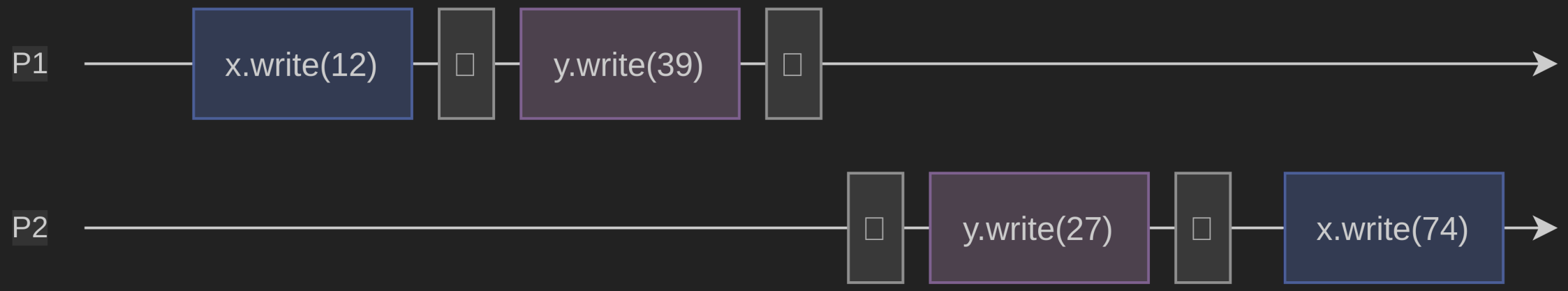

🤔 semaphores / events / condition variables / signals

# Previous work

- monitors (Hoare, 1974)
  - group shared variables with related procedures
  - protect the procedures with a lock
  - 🤔 does not support dynamic allocation
- LockLint (SunSoft, 1994)
  - purely static detection
- Lamport's *happens-before* relation (1978)
  - inside a thread (execution order)
  - between threads (synchronization accesses)

# *Happens-before* problems 🤔

- inefficient
- dependent on the execution interleaving
  - more runs can mitigate this issue



# Design

> Enforce a simple locking discipline that
> every shared variable is protected by some lock

- maintain a set of locks (*lockset*) held when accessing each variable
- refine each set after each access
- ⚠️ emit a warning if a set becomes empty

# Lockset refinement

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

Figure 3 from the Eraser paper

# Edge cases

- unprotected initializations ( 🚩 )
- read-shared variables
  - write once then always read
- read-write locks
  - multiple readers, single writer

# Edge cases handling

- stateful variables
- differentiate read and write lock sets

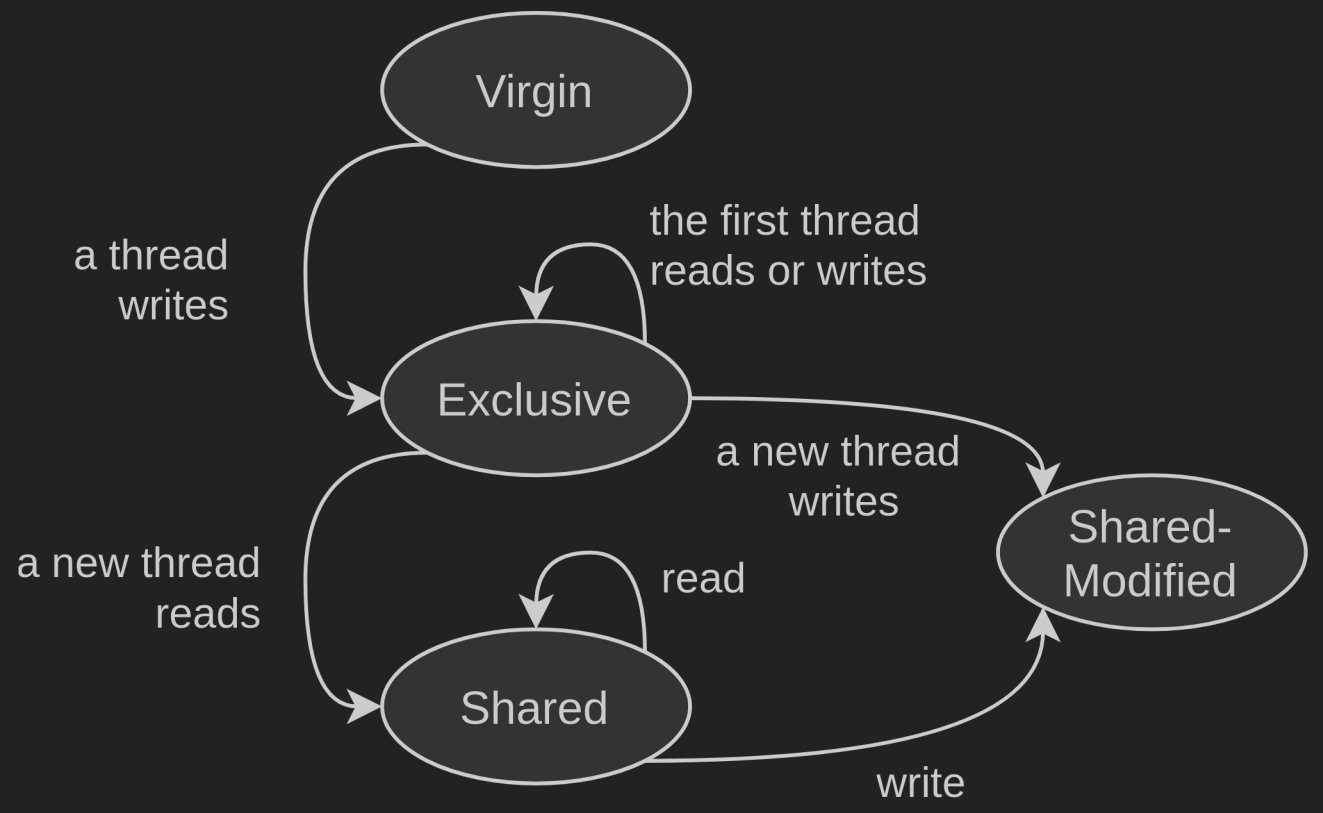



Figure 4 from the Eraser paper

# Implementation

A testing utility that instruments a binary to call the Eraser runtime.
(only for the heap and global data)

- loads & stores
- thread initialization & finalization
- memory allocation

# Representing Lock Sets

| index | lock set | hash |
|---|---|---|
| 1 | {mu1, mu2} | `0xBAADF00D` |
| 2 | {mu1} | `0xE5CA1ADE` |
| 3 | {mu2} | `0xB0BACAFE` |

- cache set intersections
- associate a *shadow word* to each variable
  - 30 bits for the lock set index
  - 2 bits for the variable state

# False alarm mitigation

- memory reuse (free lists, private allocators)
- private locks (non-`pthread`)
- benign races 🚩

→ developers can add annotations

- `EraserReuse(address, size)`
- `EraserReadLock(lock)`

# Performance

- a 10x to 30x slowdown
- probably due to the numerous procedure calls
- 🚩 probably impacts scheduler behavior

# Experiments

Tested against several industry programs:

- AltaVista `mhttpd` & `Ni2` (net indexer)
  > 30 minutes to identify and fix month-old races
- Vesta cache server
- Petal distributed storage system

→ numerous false alarm

→ but also several real race conditions

# Undergraduate coursework evaluation

- 10% had data races
- could have provided Eraser to students 😅

# Bonus

- experiment to detect races in the SPIN OS kernel
  - which leverages interrupt levels as informal locks
  - 🤔 proof that the system is not generic enough
- multiple lock handling

  > It's possible but it might break things

- deadlock detection
  - ordered locking & unlocking

# Thoughts

- only lock-based programs is quite restrictive
- several issues swept under the carpet 🧹
  - scheduler dependency (variable initialization)
  - slowdown impact
- strange experimentation

# Discussion

- The authors chose to work with **lock-based programs** only. Would it be possible to work with *other synchronization primitives*?
- What do you think about using a **dynamic testing method**?
- What are **current techniques** to ensure thread safety? Is it possible to *statically* ensure thread safety? (type-safe languages?)