Helgrind:
a constraint-based data race detector

Julian Seward, jseward@acm.org

The big picture

Helgrind is a debugging tool for POSIX pthreads based programs

Three checkers in one:

- Misuse of the POSIX pthreads API
  unlocking of not-locked mutexes, pth_con_signal cv/mx mismatches, etc
- Lock acquisition order inconsistencies – potential deadlocks

```
T1: ... lock(A) ... lock(B) ...
T2: ... lock(B) ... lock(A) ...
```

- Data races: uncoordinated accesses to the same location by more than one thread

Data races are bad

Cause unrepeatable data corruption and failures -- hard to debug

```
int x = 0
Thread1:  x++
Thread2:  x++
```

Value of x is unknown afterwards. 1 ? 2 ? Something else?

Gives the program “undefined behaviour” per C++11

- Compiler can transform your code into whatever it likes, if it can detect the program is racey
- Hardware ditto -- will reorder stores
- Not widely (enough) appreciated!

In practice, cause a long tail of in-the-field weird failures that you can't make sense of

How to detect data races?

Need to watch all memory accesses and all inter-thread synchronisation events

Two schemes:

Lockset inference (eg Eraser): check a specific locking discipline is used

- Too inflexible
- Too many false positives in race-free code

Pure “happens-before” detectors

- No false positives provided it can see all sync events
- Can miss some races that lockset inference will find
- Scheduling sensitive

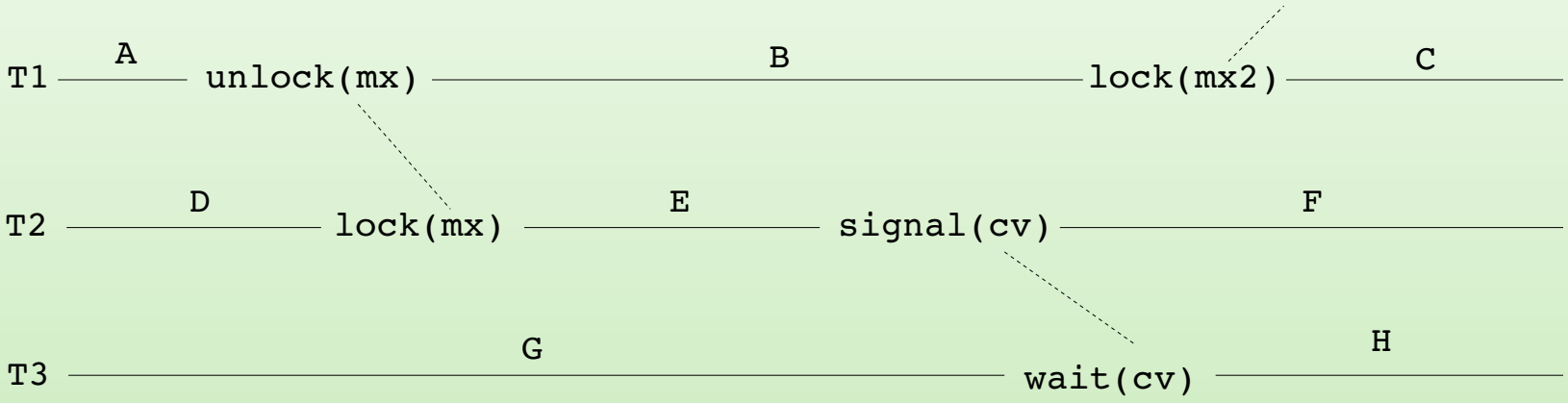
Two race detector tools in the Valgrind tree

- DRD and Helgrind
- Compute the same thing using different algorithms
- Both are the “happens-before” kind

Thread lifetime segments

Each thread's lifetime is divided up into segments

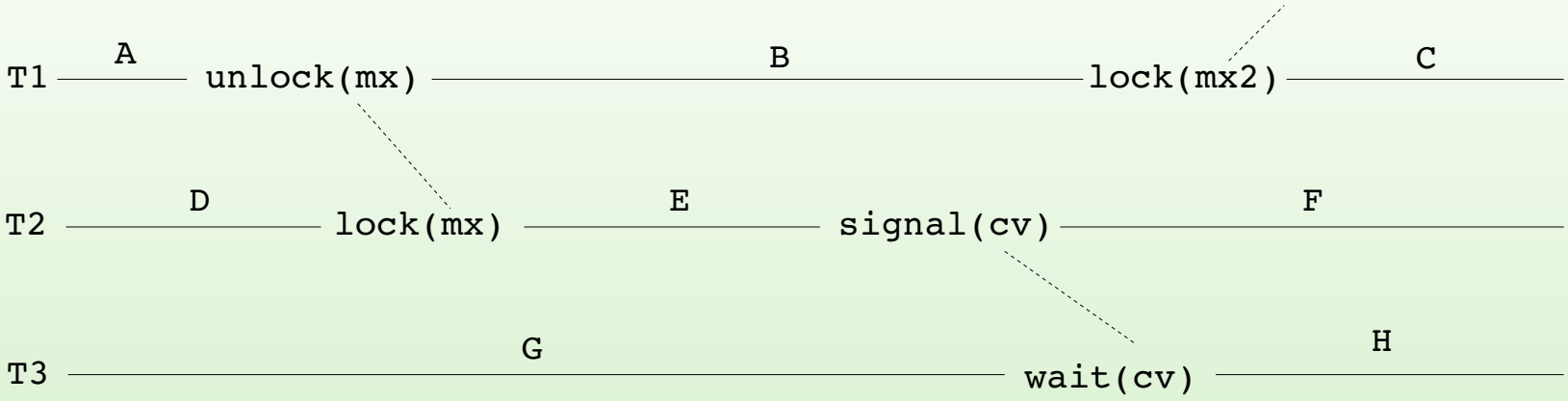
Each sync event (lock acquire, release, etc) starts a new segment



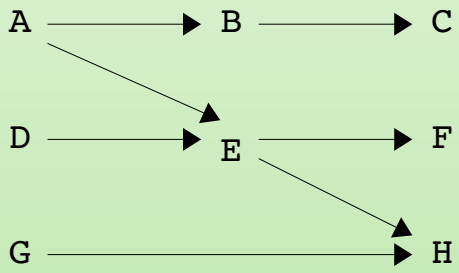
All sync events are viewed as inter-thread messages

Any kind of sync event is enough “evidence” for intent to synchronise

Thread lifetime segments



The thread segments and inter-thread edges form a partially ordered set:



Basic idea: if two different segments access the same location, they must be connected (transitively) via ⟶ in the partial ordering. If not, there's a race.

⟶ is the “happens-before” relationship, “⊏” in the partial ordering.

How to implement it?

One way to implement it

- For each segment, record each accessed memory address
- When program finished, compare all segments not related by ⊏ or ⊐
- If any pair contains the same address, there has been a race
- eg, if any accesses in B also occur in E, there has been a race
- Can be implemented using sparse bitmaps

Naive implementation:

- has unbounded storage requirements
- can't get stack trace for a race

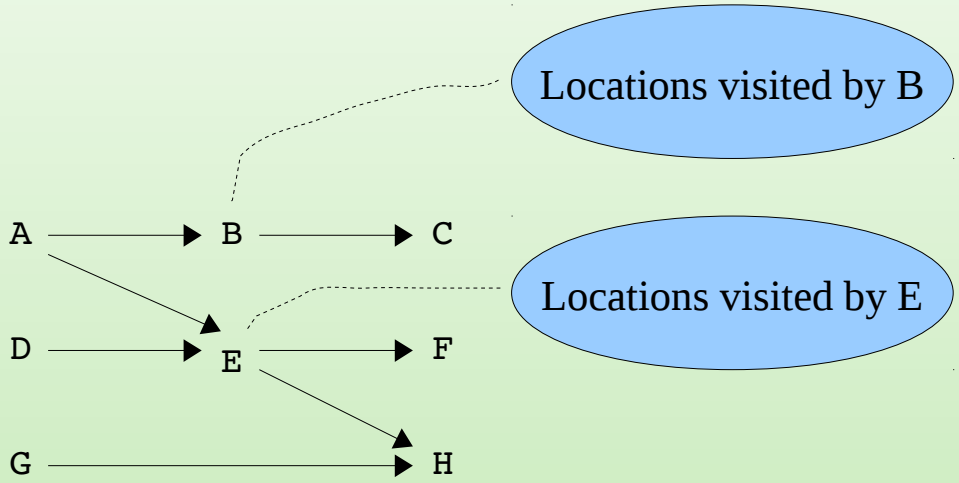
Real implementation (eg DRD):

- Does accessed-set comparisons incrementally
- Reduces storage requirements
- Can get stack trace for the later access in a race
- DRD has many other clever optimisations

A different representation

Problem: the bitmap-per-segment scheme can only get the later stack for a race

- Makes diagnosis of races difficult
- Is there another way to do this?



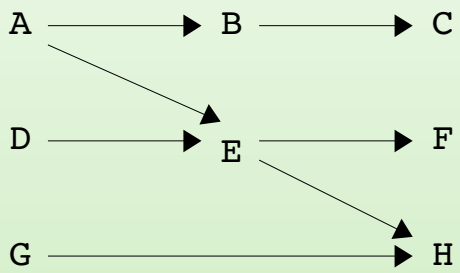
This is: a mapping from thread segments to set of memory locations

Can we use a mapping from memory locations to set of thread segments?

A different representation

Basic idea:

- Each location carries a “constraint set” of segments
- Specifies the “minimum safe condition” for access
- A safe access must be from a segment that is ⊒ all segments in the constraint set



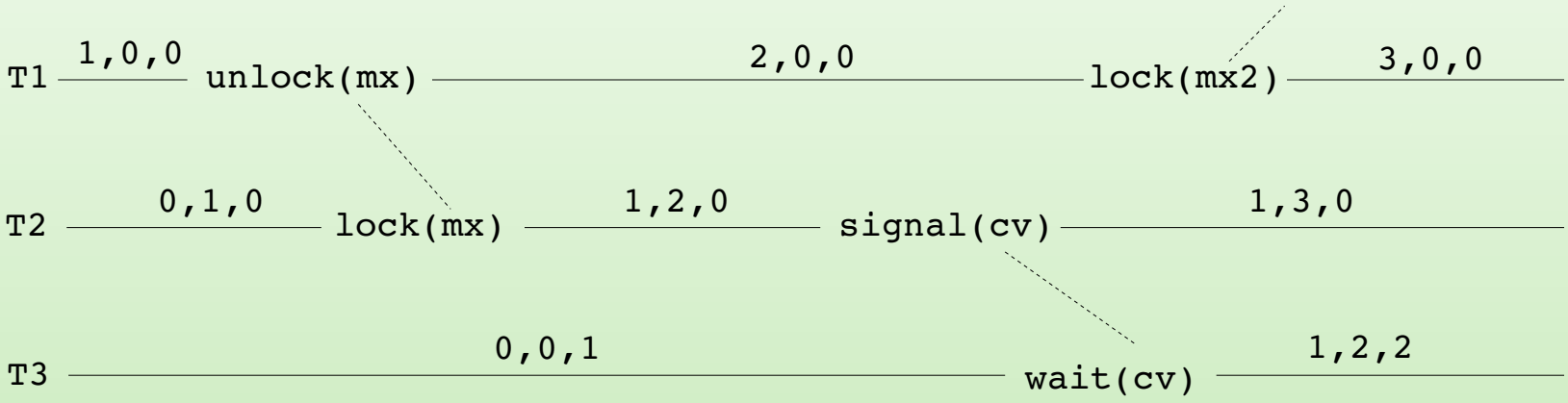
eg. Location accessed by A
Constraint set = { A }. All safe accesses must be in segments ⊒ A.
Later accessed by E
Constraint set = { E }
The only valid accessing segments now are F and H

Corresponds with our intuition about the locking

Representing constraints

Sets of segments are infeasible

Vector timestamps are an old idea, from mid 70s at least



Each thread knows its own “time” and keeps best approximation for times of other threads

So

- Each thread has a vector timestamp, **now**
- Each memory location has a vector timestamp, **safe-access-constraint**
- For each access, check **safe-access-constraint** $\sqsubseteq$ **now** and update **safe-access-constraint**

Vector timestamps

A vector of small ints, one per thread

- easy to compute ⊏ (“check constraint”) and ⊓ (“merge constraints”)
- **safe-access-constraint** and **now** are vector timestamps
- actually need two timestamps per location: one for reads, one for writes
- reader-writer locks add further complexity

This gives a huge storage management problem!

- eg, app with 50 threads, 32-bit int as timestamp element
- requires 2 * 50 * sizeof(int) = 400 bytes per timestamp
- for each race-checked byte of memory
- No chance!

VTS compression

Two levels of compression

Have a dictionary of timestamps, and use 32-bit dictionary-id-numbers (“VtsID”)

- reduces the cost to 64 bits per address-space-byte
- 32 bits for the read constraint VtsID, 32 bits for the write constraint VtsID

Most memory ranges are unshared

- so they have only one constraint value
- use a second level dictionary to exploit this
- implement a direct-mapped cache w/ 128 byte lines
- when ejecting a cache line, transform common case to a compressed representation
- contains 4 VtsID pairs and an array of 128 2-bit numbers
- when loading a cache line, decompress to the 64-bits-per-byte representation

Result: a complex implementation, with much refcounting, GCing, VTS pruning

- `helgrind/libhb_core.c` is about 6500 lines

Achieves about 9 bits per address-space-byte for complex threaded apps

Both stacks for a race

How to get both stacks for a race?

Whenever a thread accesses in violation of its constraint VTS

- Gives the second, “later” access

When a thread's constraint set changes, take a stack snapshot

- Gives the first, “earlier” access
- Requires very fast stack unwinder; V's achieves < 250 insns/CFI-recovered frame
- Referred to as a “dragover” in the sources

Works reliably

Expensive in space, time, complexity (again)

Stepping back from the details

DRD (mapping of segments to location-sets) is

- Faster when there are relatively few synchronisation events
- Simpler
- More correct in the presence of reader/writer locks

Helgrind (mapping of locations to segment-sets) is

- Faster when there are many synchronisation events
- Excessively complex
- Provides both stacks in a race

Is there a way to reconcile the two data structures?

- Get the best of both worlds
- We should be able to prove isomorphism results (or not)
- AFAIK, this is an open research problem

Questions?