Landslide: Systematic Testing for Kernel Space Race Detection

Ben Blum

Jiří Šimša, Garth Gibson

Parallel Data Laboratory

Carnegie Mellon University

Outline

Motivation: Concurrency debugging

- Systematic testing versus stress testing
- State space explosion
- Challenges of kernel-space

Tool: Landslide

- Design and interface
- Addressing challenges

Evaluation: Finding Races

- Student user study
- Case study

Future Work and Conclusion

Motivation – Example

```
int thread_fork()
{
    thread_t *child = spawn_new_thread();
    add_to_runqueue(child);
    return child->tid;
}
```

Motivation – Example

```c
int thread_fork()
{
    thread_t *child = spawn_new_thread();
    add_to_runqueue(child);
    return child->tid;
}
```

← **“child” gets freed!**

- On exit, child's state is freed
- Forking thread does use-after-free
- Might return garbage instead of thread ID

Motivation – Testing Techniques

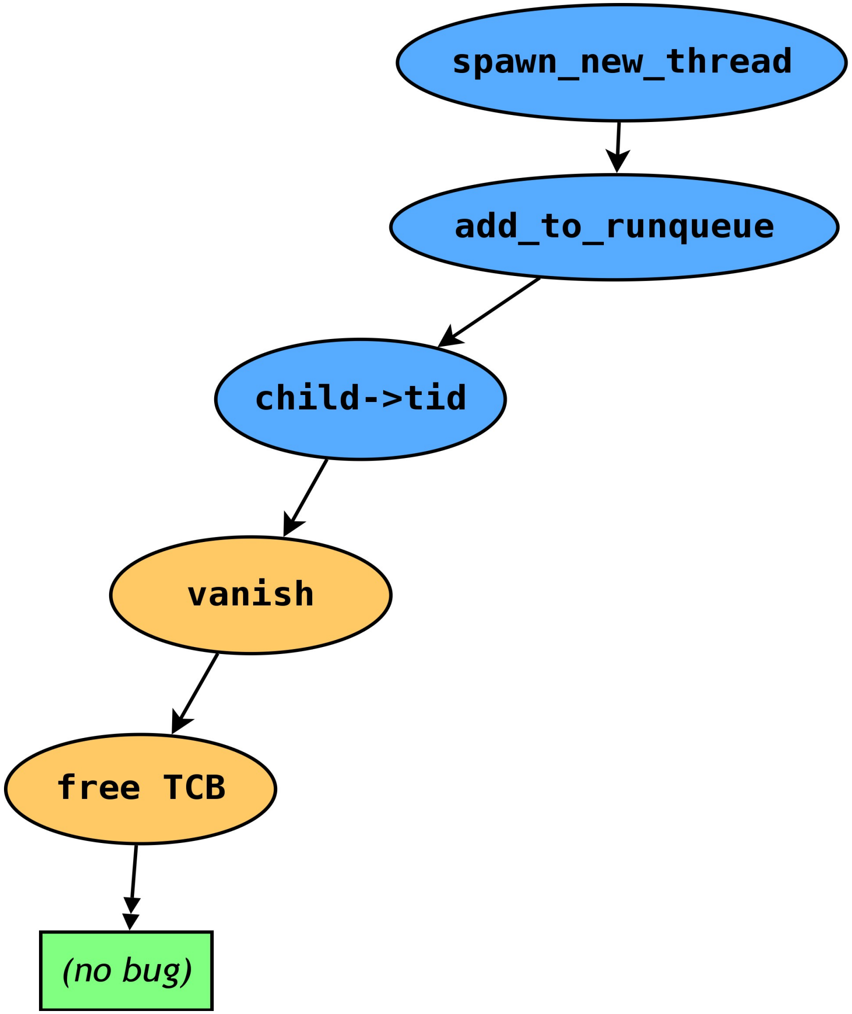
Stress Testing: Common testing approach

- Trying to exercise as many interleavings as possible
- Exposes race conditions at random
  - "If a preemption occurs at *just the right time*..."
- Cryptic panic messages or machine reboots

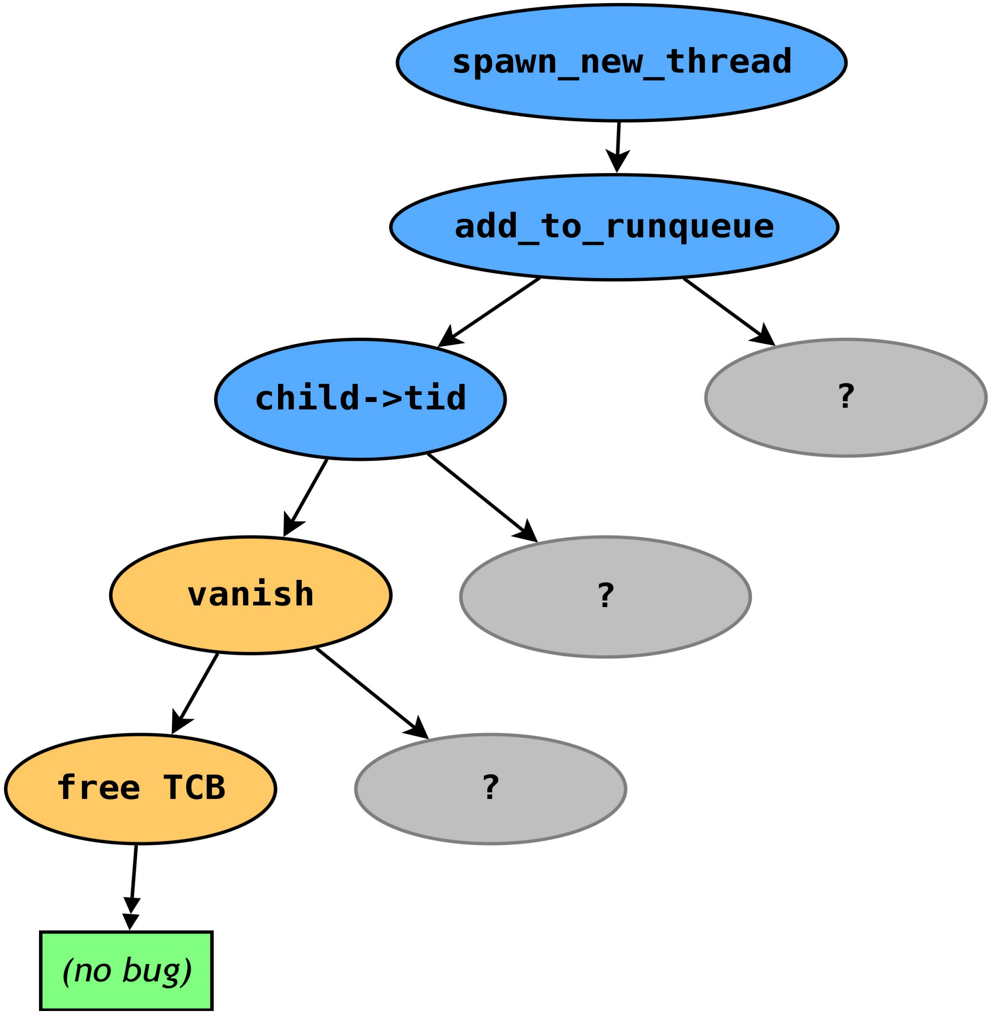
Systematic Testing *[Godefroid '97]*

- Make educated guesses about when to preempt
- Run *every single* interleaving
- Provide better debugging information, reproducibility

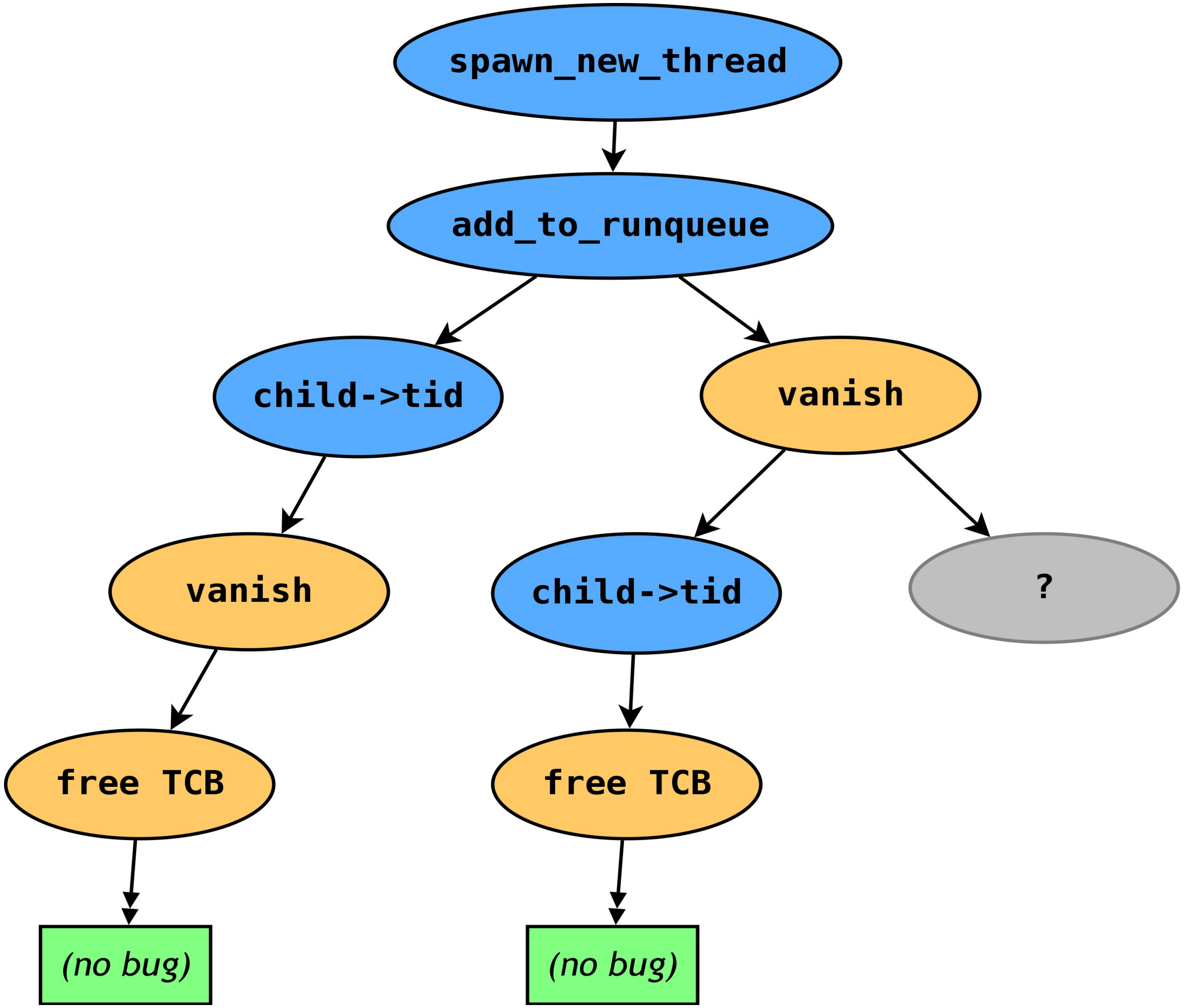
The Execution Tree



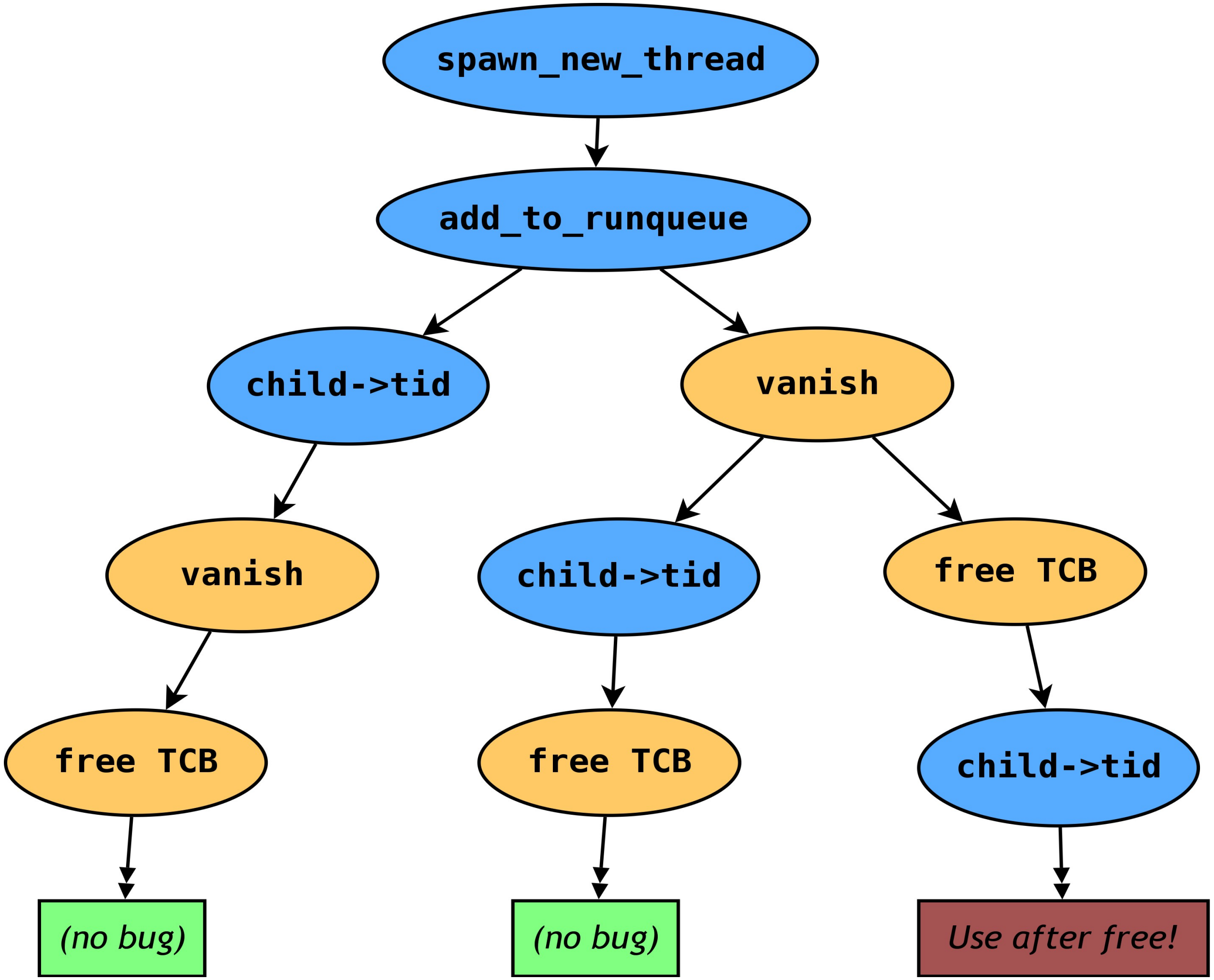
The Execution Tree



The Execution Tree



The Execution Tree



Carnegie Mellon
Parallel Data Laboratory

Decision Points

Decision points: where being preempted causes different behaviour.

What does “all possible interleavings” mean?

- One extreme: Decide at every instruction
  - Impossibly large test runtime
- Other extreme: No decision points
  - Makes “no race found” a weak claim
- Sweet spot: Intuit where preemptions “might matter”
  - Joint effort between programmer and tool

Decision Points

In userspace, pthread library calls. *[Simsa '11]*

In kernels, we need to tell the story from scratch.

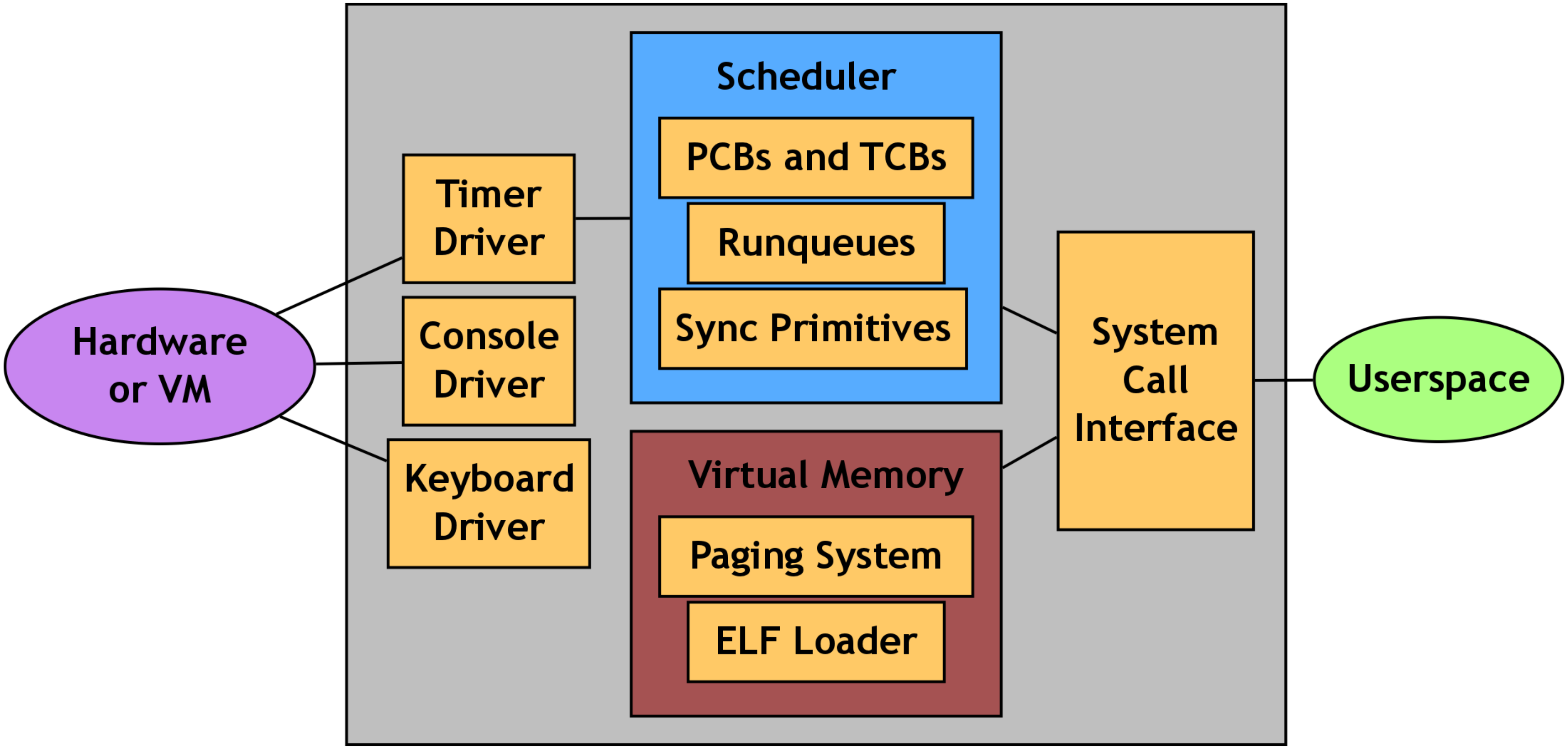
Challenge 1: Causes of Concurrency

Kernels contain their own concurrency implementation.

- Context switching
  - Nondeterministic timer-driven thread scheduling
- Runqueue tracking
  - Which threads are runnable?
- Thread lifecycle tracking
  - When are threads created/destroyed?

Challenge 2: The Kernel as One Program

“Everything interleaves with everything else”?



Carnegie Mellon
Parallel Data Laboratory

Contribution

Systematic testing is a *tool* for testing concurrent systems.

- How can it be applied in kernel-space?
- What simple things can the user provide to help?

Outline

Motivation: Concurrency debugging

- *Systematic testing versus stress testing*
- *State space explosion*
- *Challenges of kernel-space*

Tool: Landslide

- Design and interface
- Addressing challenges

Evaluation: Finding Races

- *Student user study*
- *Case study*

Future Work and Conclusion

Execution Environment

Simics: a full-system x86 simulator

- Landslide runs as a Simics module.
  - Can see kernel instructions, memory accesses

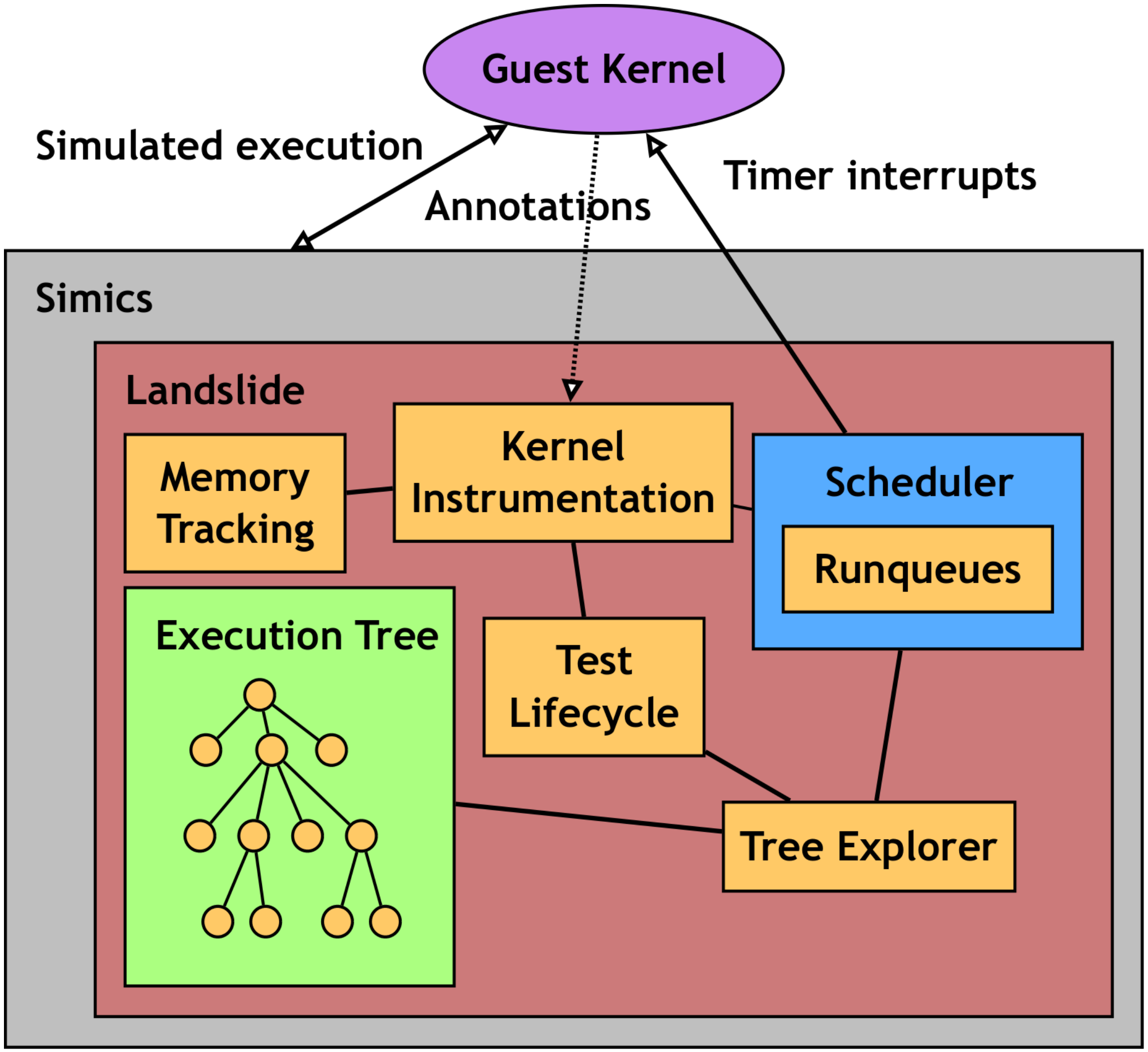
Controlling system nondeterminism

- Injecting timer interrupts triggers thread scheduling
- Future work may use device interrupts (disk, network)

Backtracking

- After each test execution, can rewind system state using Simics "bookmarks"

Anatomy of Landslide



Interface

```
int thread_fork()
{
    thread_t *child = spawn_new_thread();
    tell_landslide_forking();
    add_to_runqueue(child);
    tell_landslide_decide(); /* Interrupt me here! */
    return child->tid;
}
```

User must tell Landslide when certain events happen:

- When do threads become runnable / descheduled?
- When does the scheduler switch threads?

Can also configure optional additional decision points

Decision Trace

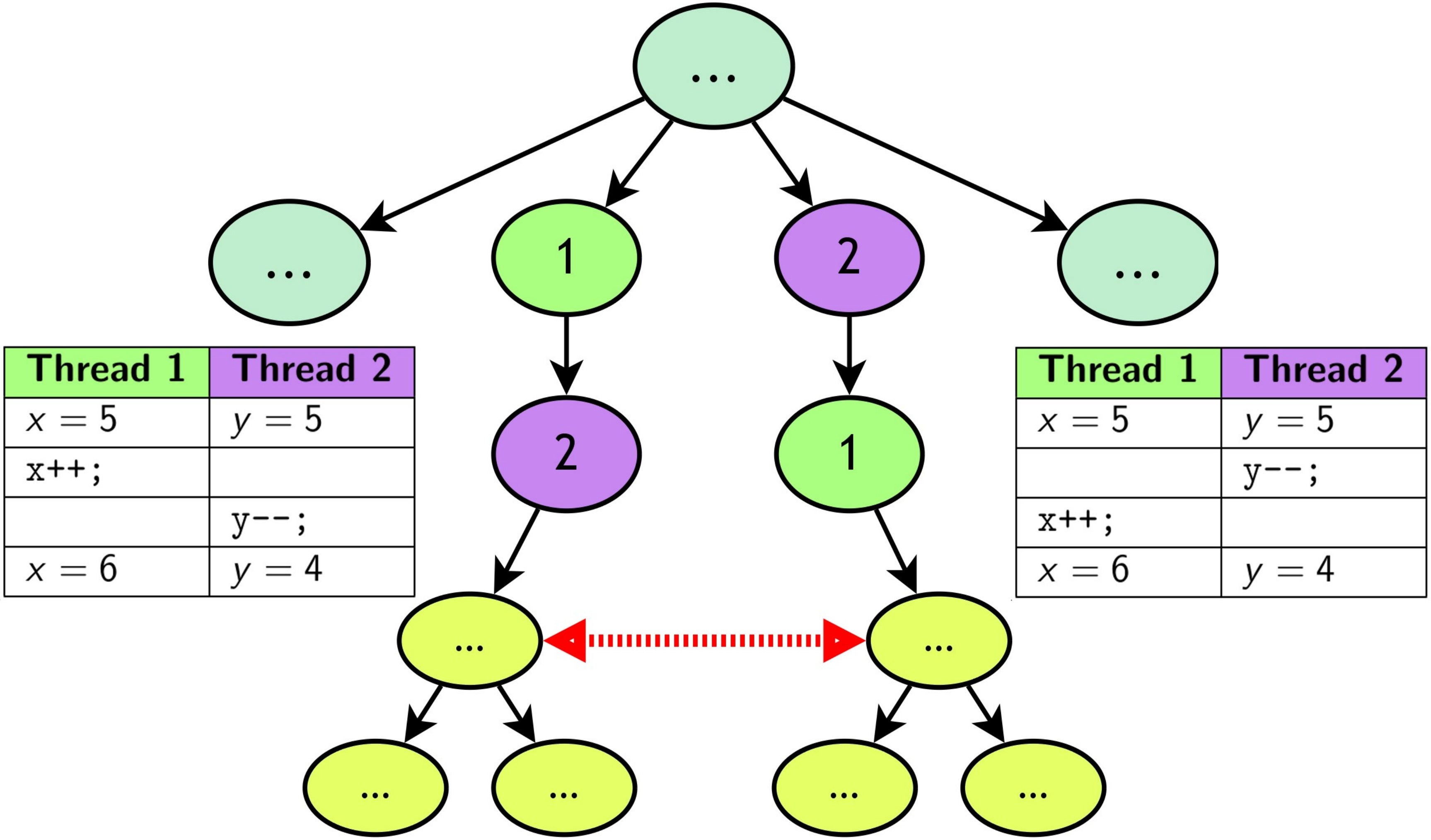
```
USE AFTER FREE - read from 0x0015a8f0 at PC 0x00104209
[0x15a8f0 | 4136] was allocated by TID3 at (...)
                      and freed by TID4 at (...)
****     A bug was found!    ****
**** Decision trace follows. ****
1:  1347079 instructions, old TID 3, new TID 4
    TID3 at 0x00105a10 in context_switch,
            0x001041f4 in thread_fork,
            0x0010362b in thread_fork_wrapper
2:  1350725 instructions, old TID 4, new TID 3
    TID4 at 0x00105a10 in context_switch,
            0x00104681 in yield,
            0x00104570 in exit,
            0x00103708 in exit_wrapper
Stack: TID3 at 0x00104209 in thread_fork,
               0x0010362b in thread_fork_wrapper
Total decision points 24, total backtracks 5
```

State Space Reduction

State spaces grow exponentially.

- Fortunately, some sequences cause identical states.
- Dynamic Partial Order Reduction *[Flanagan '05]*
  - Can even be parallelized *[Simsa '12]*
  - Requires “memory independence relation” between transitions

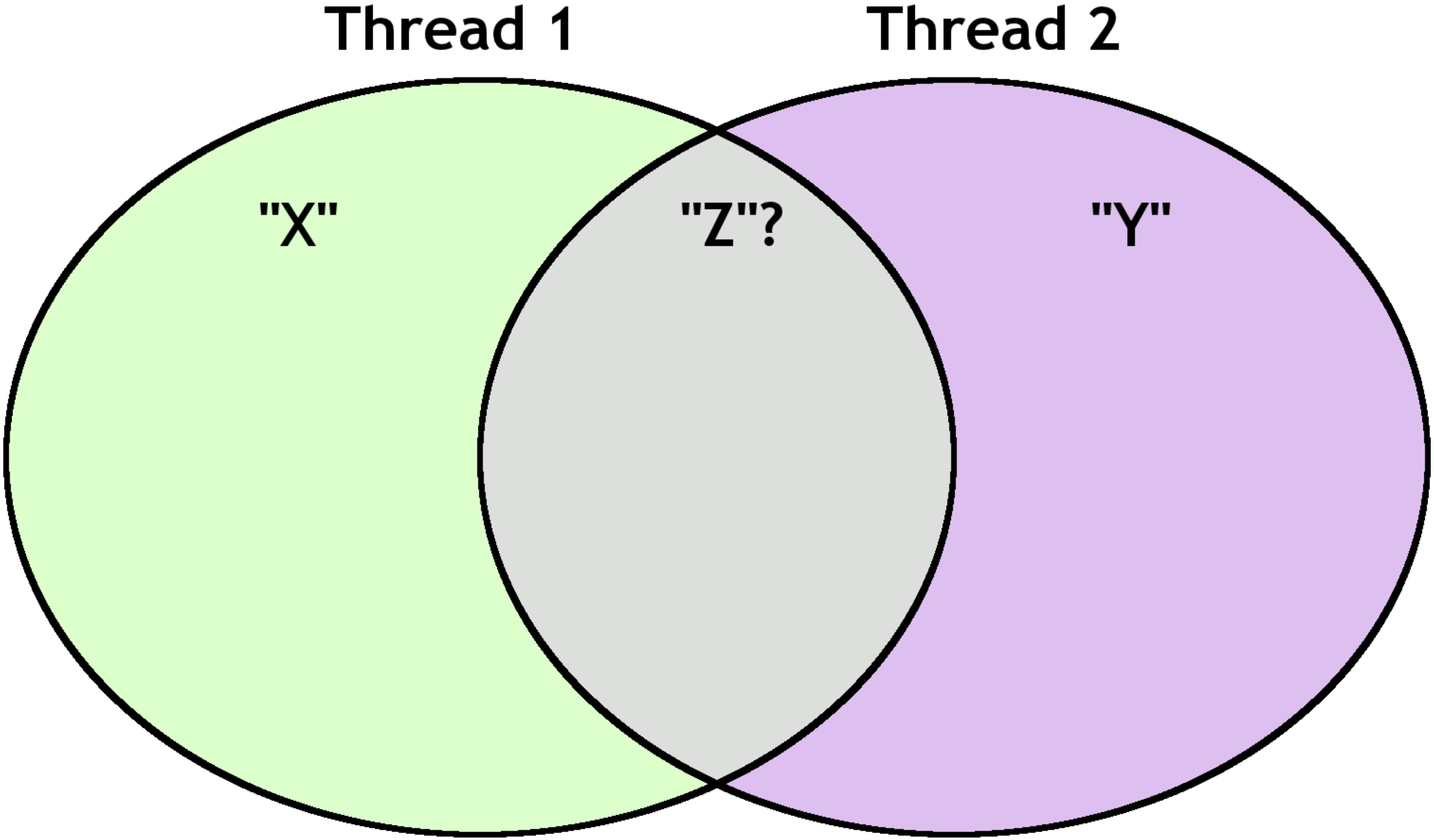
State Space Explosion



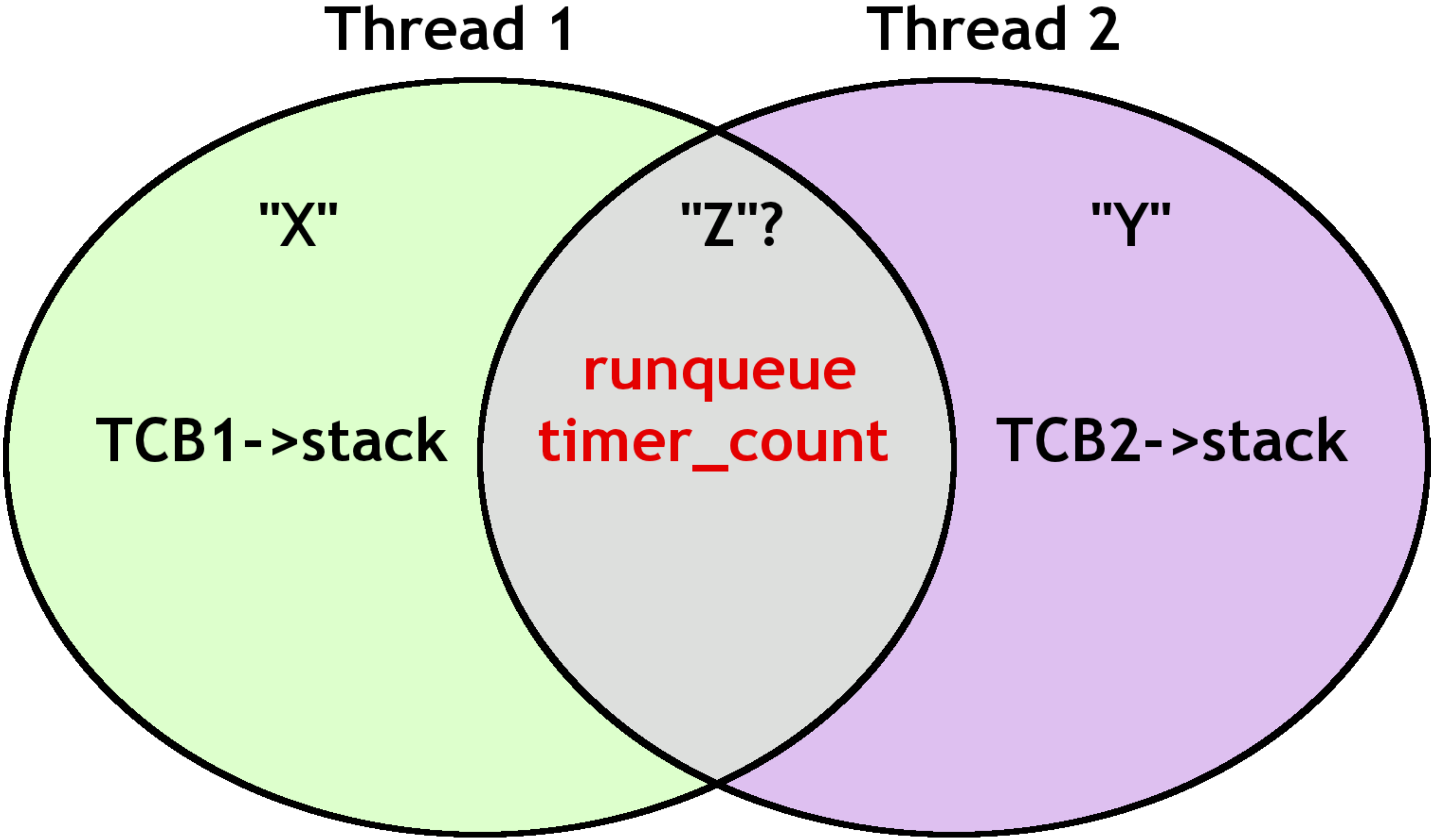
Thread 1	Thread 2
$x = 5$	$y = 5$
`x++;`	
	`y--;`
$x = 6$	$y = 4$

Thread 1	Thread 2
$x = 5$	$y = 5$
	`y--;`
`x++;`	
$x = 6$	$y = 4$

Shared Memory Conflicts

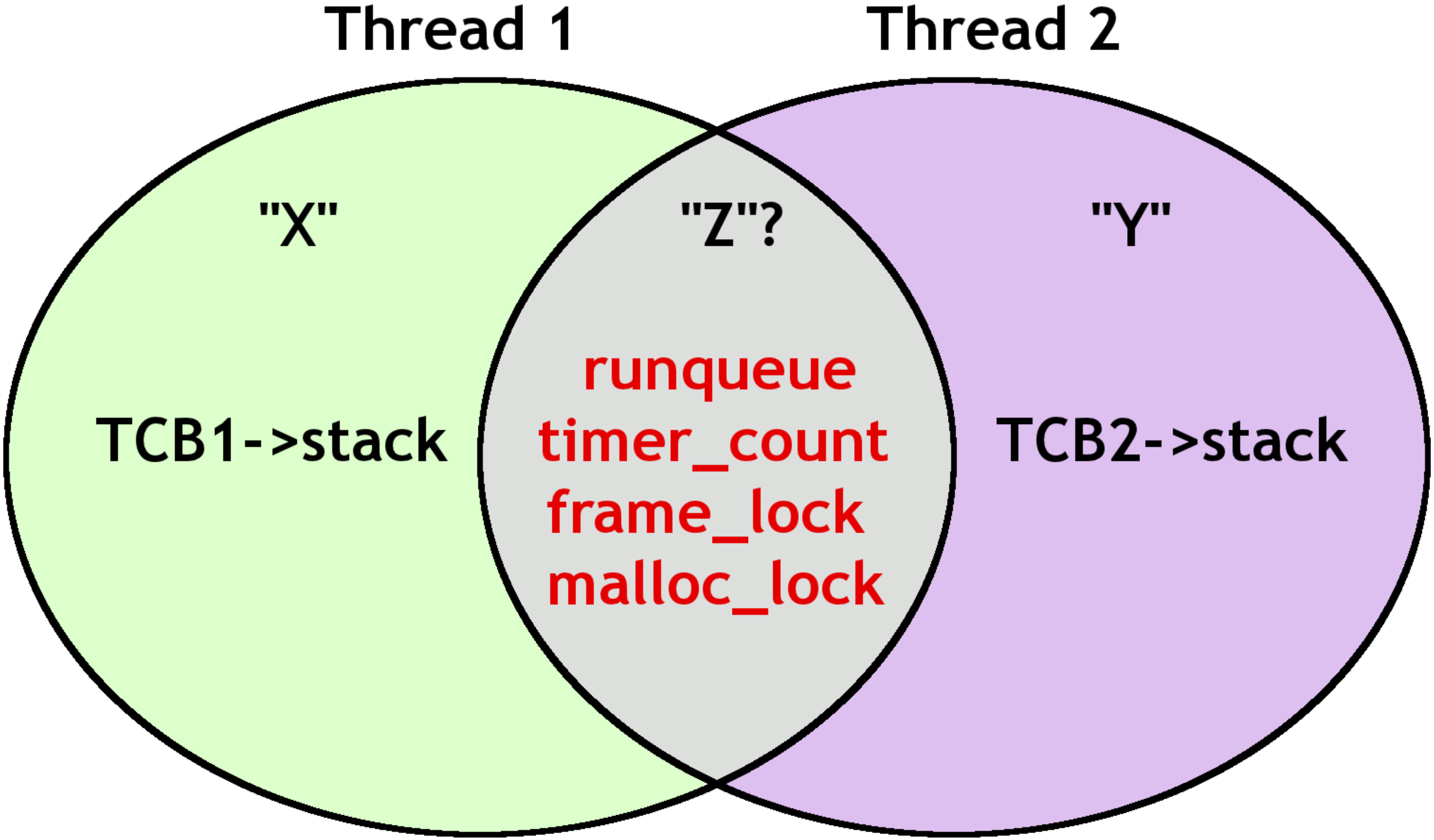


Shared Memory Conflicts



Carnegie Mellon
Parallel Data Laboratory

Shared Memory Conflicts



Shared Memory Conflicts

Solution: Only consider “relevant” memory conflicts.

- Ignore scheduler, global objects assumed to be correct.
  - Sacrifice ability to test these to efficiently test everything else.
- User must specify what to ignore.

Focusing the Search Space

Recommended decision points:

`mutex_lock(), mutex_unlock()`

Problem: Kernel uses mutexes *everywhere*.

- Good: `exit()` calls `mutex_lock()`
- Bad: `destroy_address_space()` calls it too

Solution: User specifies which modules to pay attention to.

- `within_function  exit`
- `without_function destroy_address_space`

Identifying Bugs

How do we know we've found a bug?

Definite bugs

- Kernel panics / assertion failures
- Use-after-free
- Deadlock

Probable bugs

- Infinite loop
  - Use structure of execution tree to judge progress.

Identifying Bugs

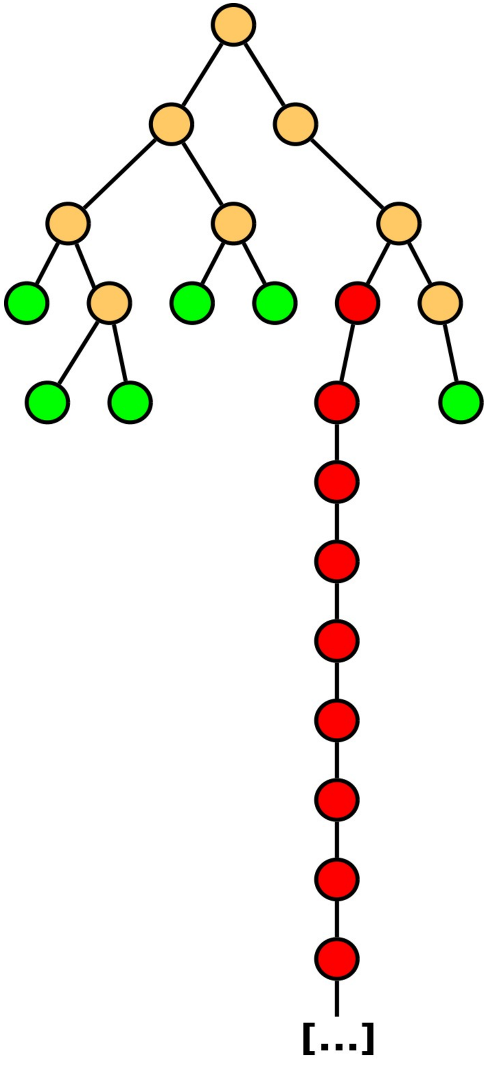
How do we know we've found a bug?

Definite bugs

- Kernel panics / assertion failures
- Use-after-free
- Deadlock

Probable bugs

- Infinite loop
  - Use structure of execution tree to judge progress.



Outline

Motivation: Concurrency debugging

- *Systematic testing versus stress testing*
- *State space explosion*
- *Challenges of kernel-space*

Tool: Landslide

- *Design and interface*
- *Addressing challenges*

Evaluation: Finding Races

- Student user study
- Case study

Future Work and Conclusion

Working with Students

15-410: Operating System Design and Implementation

- Students implement a small kernel in 6 weeks
- “Pebbles”, a UNIX-like system call specification
- On average 4000 lines of code

Solicited students to use Landslide on their kernels.

- Can Landslide find bugs “in the wild”?
- How much time does manual instrumentation take?

Five groups participated; four got it to work.

- Average instrumentation time 100 minutes (55-158)
- All four groups found bugs; two were nondeterministic

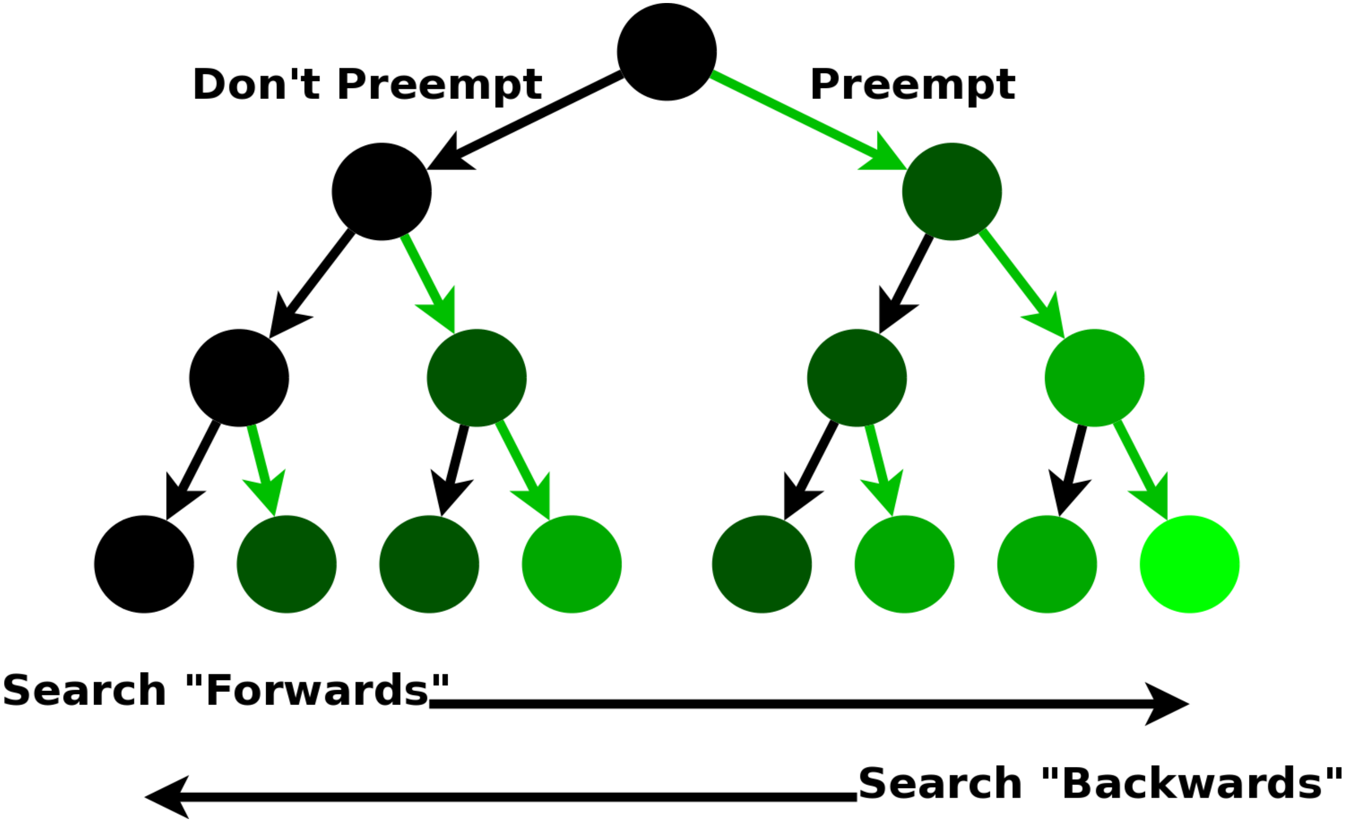
Studying Bugs In-Depth

Tested two Pebbles kernels: my own, and one I graded

- Confirmed several of the most subtle races I had found in the kernel I graded
  - Parent and child exit simultaneously
- Found a previously unknown bug in my kernel
  - Condition variable wait queue corruption
  - Missed by TA's manual inspection, stress tests

Studying Bugs In-Depth

Exploration ordering: Search branches with more or fewer preemptions first?

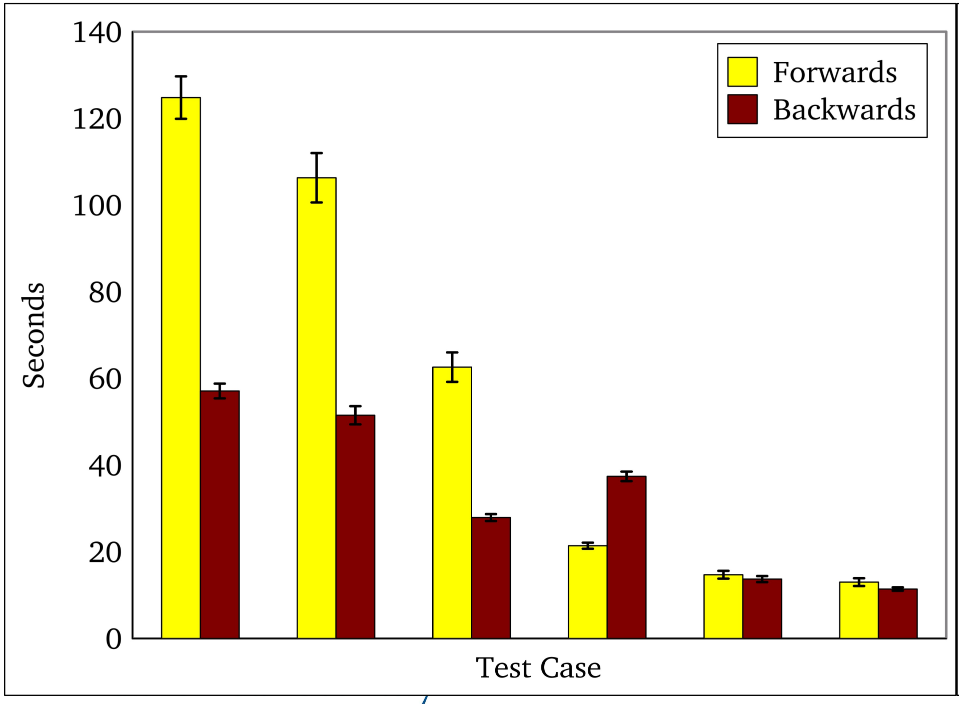


Studying Bugs In-Depth

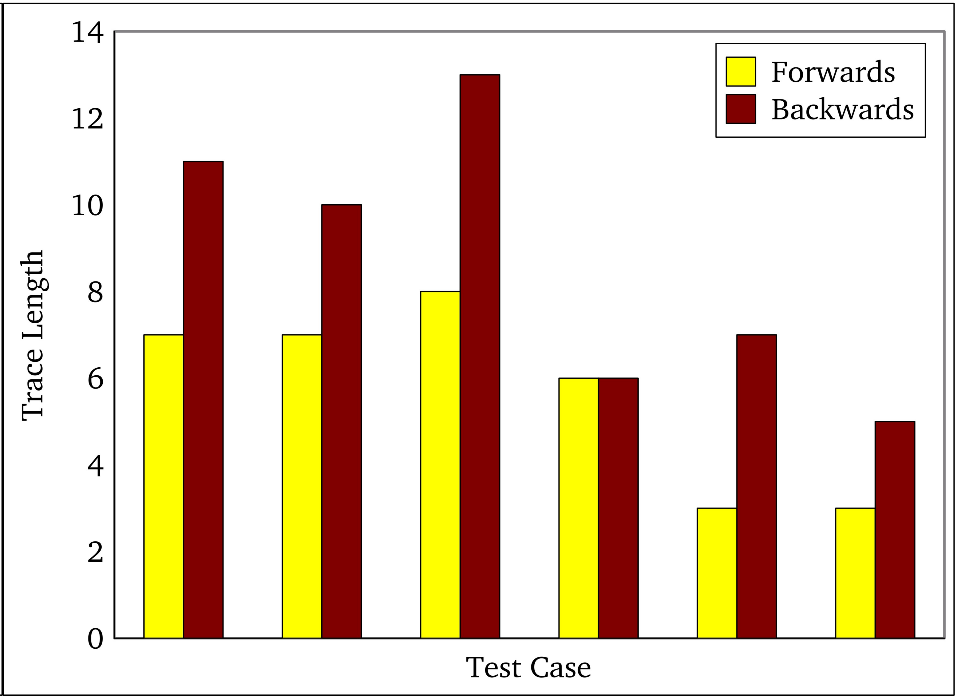
Exploration ordering

- “Backwards” exploration found bugs sooner
- “Forwards” exploration found shorter decision traces

Exploration Time



Decision Trace Length



Outline

Motivation: Concurrency debugging

- *Systematic testing versus stress testing*
- *State space explosion*
- *Challenges of kernel-space*

Tool: Landslide

- *Design and interface*
- *Addressing challenges*

Evaluation: Finding Races

- *Student user study*
- *Case study*

Future Work and Conclusion

Future Work

Performance

- Virtualization – less control, compared to Simics

Generalizing the concurrency model

- Device driver input as new source of nondeterminism
- Multicore execution
- Better model more complex kernels
  - e.g. educational (Pintos) or production (Linux)

Emphasis on "steering" test by changing parameters

- Start with coarse granularity, iteratively refine
- Perhaps we can automate this steering?

Conclusion

Systematic testing

- Make educated guesses about when to preempt.
- Find many types of races; provide debugging info.

Systematic testing in kernel-space

- Use internal kernel abstractions to understand concurrency behaviour.
- Relying on user's knowledge makes testing easier.

Landslide

- A first step towards sophisticated kernel debugging techniques.

Related Work

Systematic testing

- MaceMC (NSDI '07) – liveness, random walking
- CHESS (PLDI '07) – iterative context bounding
- MoDist (NSDI '09) – network/disk model checking
- dBug (SSV '10) – dynamic partial order reduction
- SimTester (VEE '12) – interrupt injection, drivers

Data race detection

- Eraser (TOCS '97) – lock-set tracking, annotations
- DataCollider (OSDI '10) – random sampling, kernel
- RacePro (SOSP '11) – inter-process races

References

[Godefroid '97]

- Patrice Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. CAV 1997.

[Flanagan '05]

- Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. POPL 2005.

[Simsa '11]

- Jirí Simsa, Randy Bryant, Garth A. Gibson: dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. SPIN 2011.

[Simsa '12]

- Jiri Simsa. Scalable Dynamic Partial Order Reduction. Talk at PDL Retreat 2012.