Systematic Dynamic Race Condition Detection

more clever than "`slaughter cho2`" *since 2011.*

Ben Blum (`bblum@andrew.cmu.edu`)

Carnegie Mellon University - 15-410

2013, April 14

Outline

Theory: Seeing race conditions in a new way

- Case study (example)
- The execution tree
- Decision points

Technique: Systematic testing

- Requirements
- Challenges and feasibility

Tool: Landslide

- How it works
- The user-tool relationship
- User study (that's you!)

Case Study

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

Decision Points ("good" case)

Thread 1	**Thread 2**	
`spawn_new_thread`		
`add_to_runqueue`		(new thread)
`return child->tid`		
	`vanish`	
	(TCB gets freed)	(voluntary reschedule)

Decision Points (race condition)

Thread 1	Thread 2	
`spawn_new_thread`		
`add_to_runqueue`		(new thread + preempted)
	`vanish`	
	(TCB gets freed)	(voluntary reschedule)
`return child->tid`		(bad!)

Testing Mechanisms

Stress testing: `slaughter cho2` and friends

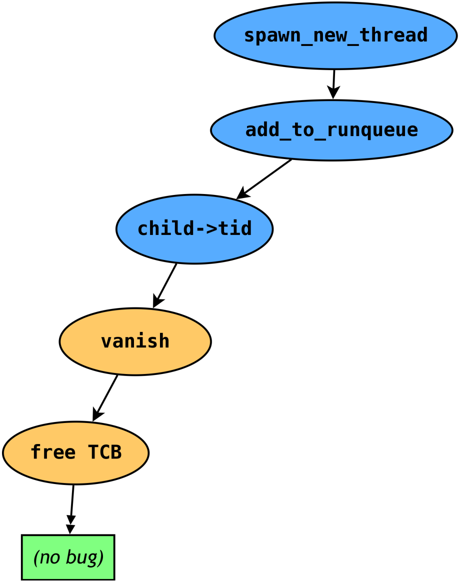
- Attempting to exercise as many interleavings as practical
- Exposes race conditions at random
  - "If a preemption occurs at just the right time..."
- Cryptic panic messages or machine reboots

What if...

- Make educated guesses about when to preempt
- Preempt enough times to run *every single* interleaving
- Tell the story of what *actually happened*.
- Overlook fewer bugs!

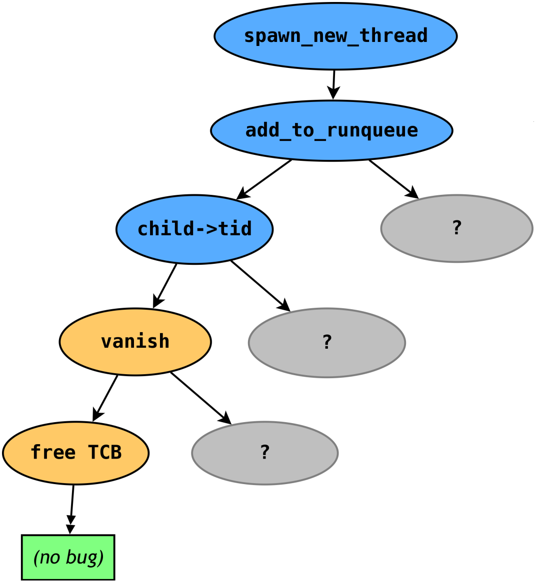
A different way of looking at race conditions. . .

Execution Tree



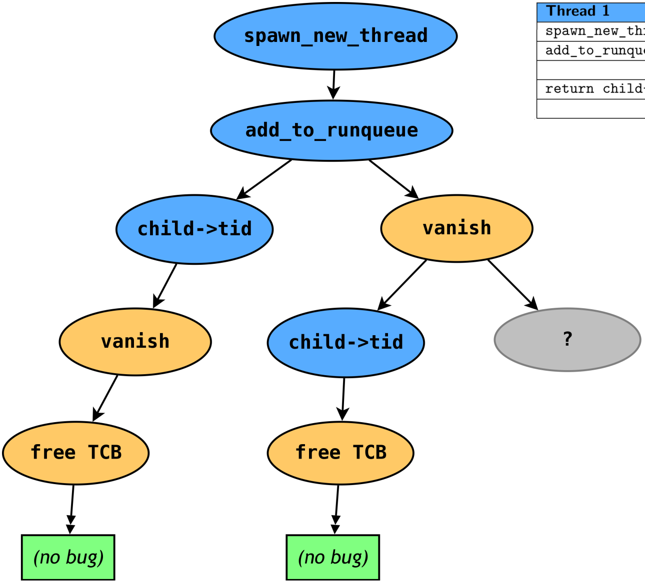
Thread 1	Thread 2
`spawn_new_thread`	
`add_to_runqueue`	
`return child->tid`	
	`vanish`
	(TCB gets freed)

Execution Tree



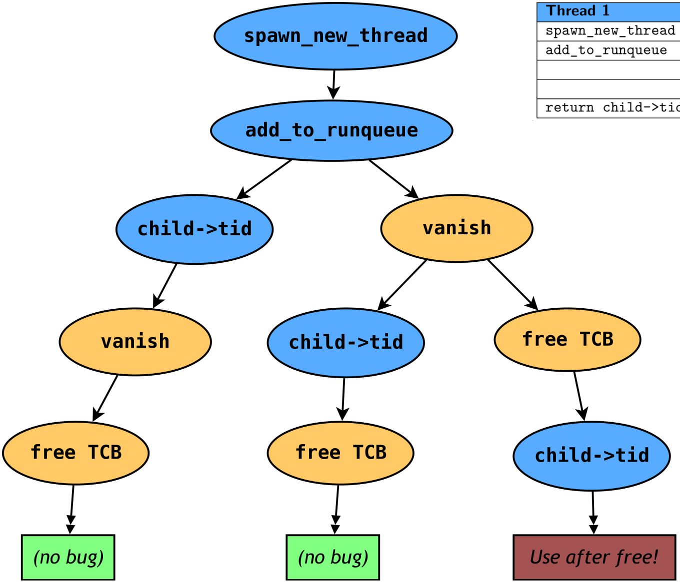
Thread 1	Thread 2
spawn_new_thread	
add_to_runqueue	
return child->tid	
	vanish
	(TCB gets freed)

Execution Tree



Thread 1	Thread 2
spawn_new_thread	
add_to_runqueue	
	vanish
return child->tid	
	(TCB gets freed)

Execution Tree



Thread 1	Thread 2
`spawn_new_thread`	
`add_to_runqueue`	
	`vanish`
	(TCB gets freed)
`return child->tid`	

Decision Points

A **decision point** is...

A code location where being preempted causes different behaviour.

- Intuitively: Somewhere that interesting interleavings can happen around.

Examples:

- A new thread becomes runnable
- Voluntary reschedule (e.g. `yield`, `cond_wait`)
- Synchronization primitives

Systematic Testing

Systematic Testing

Systematic testing is:

- Systematically enumerating different interleavings
  - Intuitively: Generate many "tabular execution traces"
- Exploring all branches in these trees
  - (by controlling scheduling decisions at decision points)
- In practice: Depth-first search of branches

Execution Tree Exploration

Important point: When does a branch end?

- All threads run to completion, or
- A bug is detected

Backtracking:

- Identify a decision point to choose differently
- Reset machine state and start over
- Replay test from the beginning, with different "decisions"

More on Decision Points

Important point: What does "all possible interleavings" mean?

One extreme: Decide at every instruction

- *Good news*: Will find every possible race condition.
- *Bad news*: Runtime of test will be impossibly large.

Other extreme: Nothing is a decision point

- *Good news*: Test will finish quickly.
- *Bad news*: Only one execution was checked for bugginess.
- *Bad news*: No alternative interleavings explored.
  - Makes "no race found" a weak claim.

More on Decision Points

Sweet spot: Insert a thread switch everywhere it “might matter”.

When do we fear being preempted?

- New threads becoming runnable (`fork`, `cond_signal`, etc)
  - Preemptions may cause it to run before we’re ready
- Synchronization primitives (`mutex_lock`, etc)
  - If used improperly…
- Unprotected shared memory accesses
  - May result in data structure corruption

Finding the sweet spot is a joint effort between programmer and tool.
(More on this later.)

Controlling Scheduling Decisions

Control over sources of nondeterminism

- Device interrupts/input
  - Disk drivers: when disk reads finish
  - Ethernet drivers: when packets arrive
- To control thread switches in a 410 kernel, vary when clock ticks happen.

Memory Interposition

In order to find use-after-free, need to know:

- When objects are `free()`d
- When threads access shared memory in the heap

Solution: Keep track of all memory events

- All calls to `malloc`/`free`
- All shared memory reads/writes

State Space Explosion

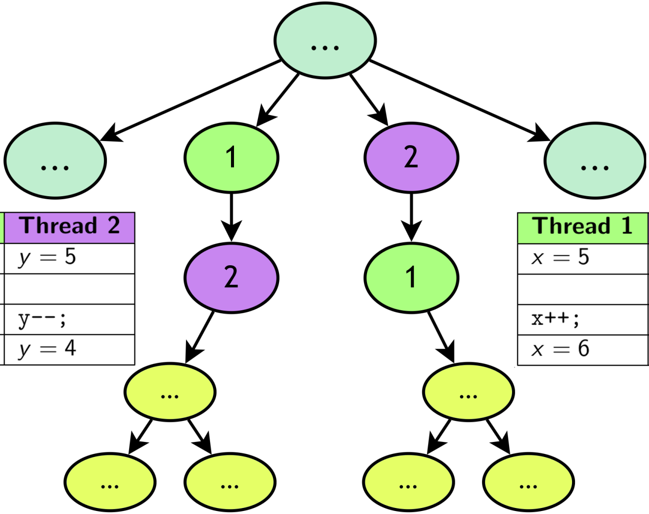
State spaces grow exponentially

- With d decision points, k runnable threads, size d^k.
- Threatens our ability to explore everything.
- Fortunately, some sequences result in identical states.

Partial Order Reduction can help.

- Complicated algorithm; ask me later for details.
- Intuitive explanation follows.

State Space Explosion

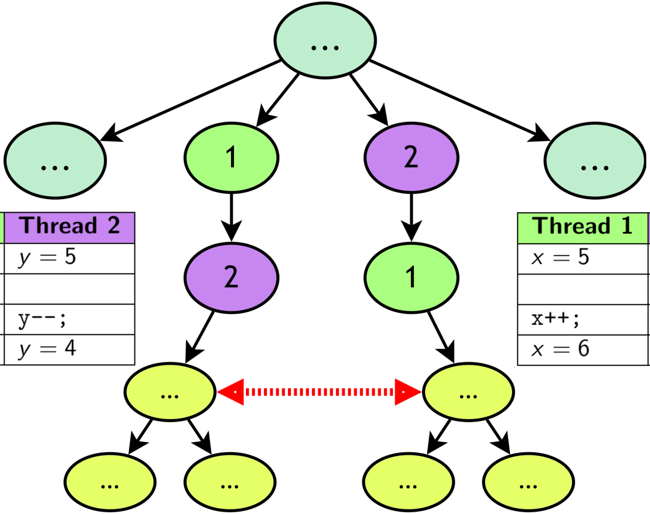


Thread 1	Thread 2
$x = 5$	$y = 5$
`x++;`	
	`y--;`
$x = 6$	$y = 4$

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

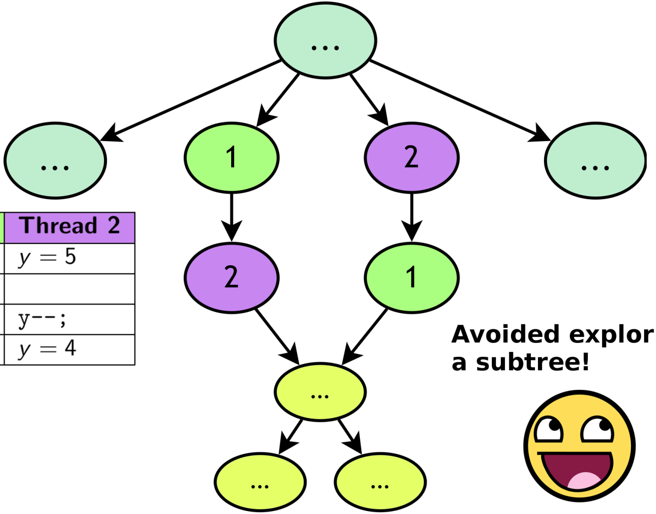
State Space Explosion

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



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

State Space Explosion



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

Avoided exploring a subtree!

Landslide

About The Project

5th year MS since June 2011 My MS thesis project (June 2011-2012)

Now I'm a Ph.D. student...

Working with Garth Gibson, Jiri Simsa

Landslide: Shows that your Pebbles are not as stable as you thought.

Landslide in Simics

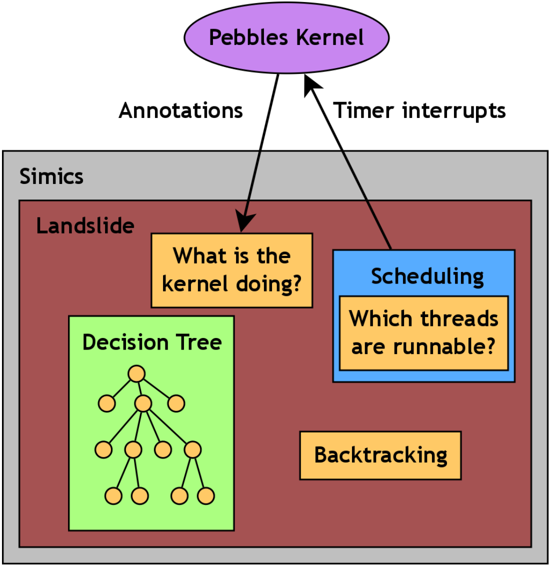
As a Simics module, Landslide knows:

- Every instruction the kernel executes
- Every memory address the kernel reads/writes

Artificially causes timer interrupts

Checkpointing/backtracking via Simics bookmarks (reverse execution)

Anatomy


Pebbles Kernel
Annotations
Timer interrupts
Simics
Landslide
What is the kernel doing?
Scheduling
Which threads are runnable?
Decision Tree
Backtracking

Identifying Bugs

Landslide can *definitely discover*:

- Kernel panics
- Deadlock
- Use-after-free / double-free

Landslide can *reasonably suspect*:

- Memory leak
- Infinite loop (halting problem)

Using Landslide

In Which Ben Offers Help

This is something you can try!

Mutual benefit

- Landslide may help you find bugs
- You help Ben evaluate his research

Keeping It Real

Finding race conditions is hard for humans.

It is hard for computer programs too.

Landslide is not an oracle.

Annotating Your Kernel

Step 1

```
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;
}
```

Your kernel needs to say when certain events happen:

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

Configuring Landslide

Step 2

```
# What test case to run?
TEST_CASE="double_thread_fork"

# The names of some important functions
TIMER_WRAPPER="timer_handler_wrapper"
CONTEXT_SWITCH="sched_switch"

# What functions to pay attention to?
within_function "thread_fork"
within_function "vanish"
```

Edit `config.landslide` with some details and tweaks

Fill in two implementation-dependent C functions in Landslide ($\leq$10 lines)

Configuring Decision Points

Landslide automatically identifies a minimal set of decision points.

- It might find bugs.
- It might overlook more fine-grained interleavings.

With help from you, it could find more.

- Optional annotation: `tell_landslide_decide()`
- Hints to where a context switch should be forced.
- Inside every call to `mutex_lock`...

And Hopefully...

```
[SCHEDULE]      thread 4 vanished
[SCHEDULE]      switched threads 4 -> 3
[MEMORY]        USE AFTER FREE - read from 0x0015a8f0 at eip 0x00104209
[MEMORY]        Heap contents: {...}
[MEMORY]        [0x15a8f0 | 4136] was allocated by TID3 at (...)
[MEMORY]                          and freed by TID4 at (...)
[BUG!]          ****     A bug was found!    ****
[BUG!]          **** Decision trace follows. ****
[BUG!]          1:   1347079 instructions, old 3 new 4, current 4
[BUG!]               TID3 at 0x00105a10 in context_switch,
[BUG!]                       0x001041f4 in thread_fork,
[BUG!]                       0x0010362b in thread_fork_wrapper
[BUG!]          2:   1350725 instructions, old 4 new 3, current 3
[BUG!]               TID3 at 0x00105a10 in context_switch,
[BUG!]                       0x00104681 in yield,
[BUG!]                       0x00104570 in vanish,
[BUG!]                       0x00103708 in vanish_wrapper
[BUG!]          Stack: TID3 at 0x00104209 in thread_fork,
[BUG!]                          0x0010362b in thread_fork_wrapper
[BUG!]          Total decision points 24, total backtracks 5
[BUG!]          Average instrs/decision 16155, average branch depth 5
```

Quick Demo

Last Year's Results

- Worked with 5 groups
  - …of which 4 put in enough time to make Landslide work
- **Investment**: about 2 to 3 hours
  - Average 112 minutes doing required instrumentation
  - Average 35 minutes configuring extra decision points & interpreting "found a bug" output

Last Year's Results

- Worked with 5 groups
  - . . . of which 4 put in enough time to make Landslide work
- **Investment**: about 2 to 3 hours
  - Average 112 minutes doing required instrumentation
  - Average 35 minutes configuring extra decision points & interpreting "found a bug" output
- **Return**: All 4 groups found bugs in their kernels.
  - All groups found deterministic bugs (e.g., use-after-free)
  - 2 groups found race conditions (i.e., Landslide needed to backtrack)

In Which Ben Offers Help - Warning

If you are already struggling, this will not “save” you.

- False-negatives: not guaranteed to find races at all
- Research-quality: possibly difficult to integrate with your kernel
- Finishing the kernel project is more important.

In Which Ben Offers Help - Warning

If you are already struggling, this will not "save" you.

- False-negatives: not guaranteed to find races at all
- Research-quality: possibly difficult to integrate with your kernel
- Finishing the kernel project is more important.

This is for you if:

- You have already submitted
- You are using late days, but willing to work over Carnival
- You are taking p3extra, but you already pass the hurdle
- You are looking for...
  - That "one pesky race"
  - A race that stress-testing missed
  - Or just familiarity with a new technique

In Which Ben Offers Help

Your kernel

- Must load the shell and run programs
- `fork`, `exec`, `vanish`, `wait`, `readline`
- Must never spin-wait (see hurdle form!)
- Should `assert()` important invariants
  - Think of `panic()` as `tell_landslide_bug()`

In Which Ben Offers Help

User study this week, by appointment (`bblum@cs.cmu.edu`)

Expect to spend:

- Up to 3 hours, just to try it out.
- 4-6 hours, if you find a bug and track it down.
- More, for multiple bugs or the truly dedicated...

Give feedback (intuitive? frustrating? found bugs?)

Watch `410.announce` for details!

Questions?