# Concurrency Testing with Data-Race Preemption Points

**Ben Blum**

**Garth Gibson**

PDL Retreat 2015

Parallel Data Laboratory
Carnegie Mellon University

# Outline

**What is Systematic Testing?**

- Motivation
- Role of preemption points
- Challenges

***Iterative Deepening***

- Data race detection
- State space estimation
- Implementation in Landslide

**Evaluation**

- Finding bugs in CMU OS student projects

**Conclusion**

# Background: What is Systematic Testing?

# Example

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

# Example

```
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

# Example

| Thread 1 | Thread 2 | |
|---|---|---|
| `spawn_new_thread` | | |
| `add_to_runqueue` | | (new thread) |
| `return child->tid` | | (yield to child) |
| | `exit` | |
| | (TCB gets freed) | |

| Thread 1 | Thread 2 | |
|---|---|---|
| `spawn_new_thread` | | |
| `add_to_runqueue` | | (new thread + preempted) |
| | `exit` | |
| | (TCB gets freed) | (yield to parent) |
| `return child->tid` | | (bad!) |

# Systematic Testing

**Stress testing**

- Common approach to finding concurrency bugs
- Relies on randomly-occurring context switches
- User-friendly but unpredictable/unbounded

# Systematic Testing

**Stress testing**

- Common approach to finding concurrency bugs
- Relies on randomly-occurring context switches
- User-friendly but unpredictable/unbounded

**Systematic testing** *[Godefroid '97]*

- Thread scheduling controlled by test framework
- Each iteration, test a different thread interleaving
- Goal: Exhaustively search “all possible” interleavings

# Systematic Testing

**Stress testing**

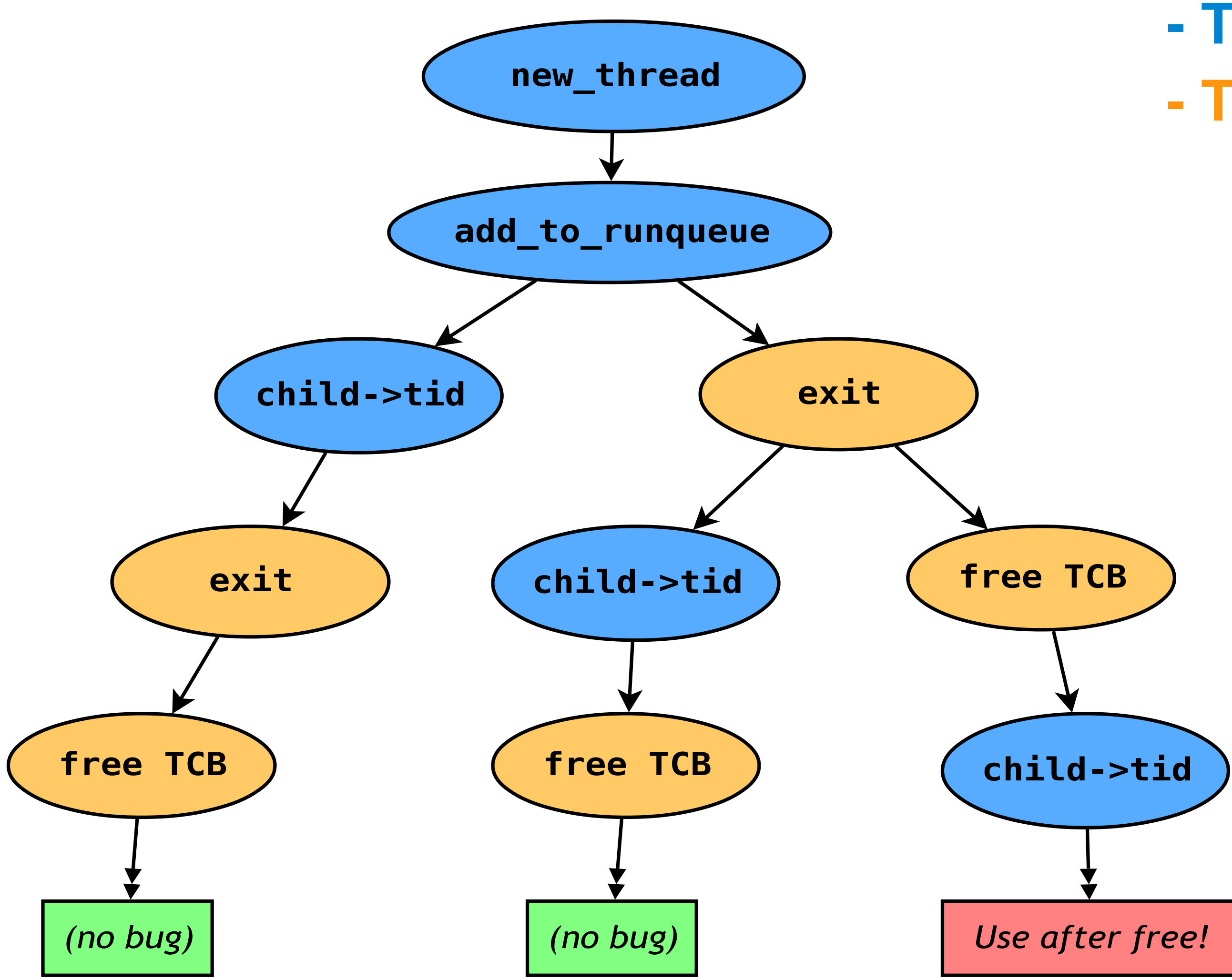
- Common approach to finding concurrency bugs
- Relies on randomly-occurring context switches
- User-friendly but unpredictable/unbounded

**Systematic testing** *[Godefroid '97]*

- Thread scheduling controlled by test framework
- Each iteration, test a different thread interleaving
- Goal: Exhaustively search “all possible” interleavings
  - Wait... what?
  - In practice, interleavings are of coarse granularity.

# Example - State Space

- Thread 1
- Thread 2



# Preemption Points

The burning question of systematic testing:
"Which **preemption points** (PPs) are important?"

State space of interleavings is *parameterized* by PPs.

- Preemption points everywhere: completion infeasible
- Too few preemption points: won't find bugs

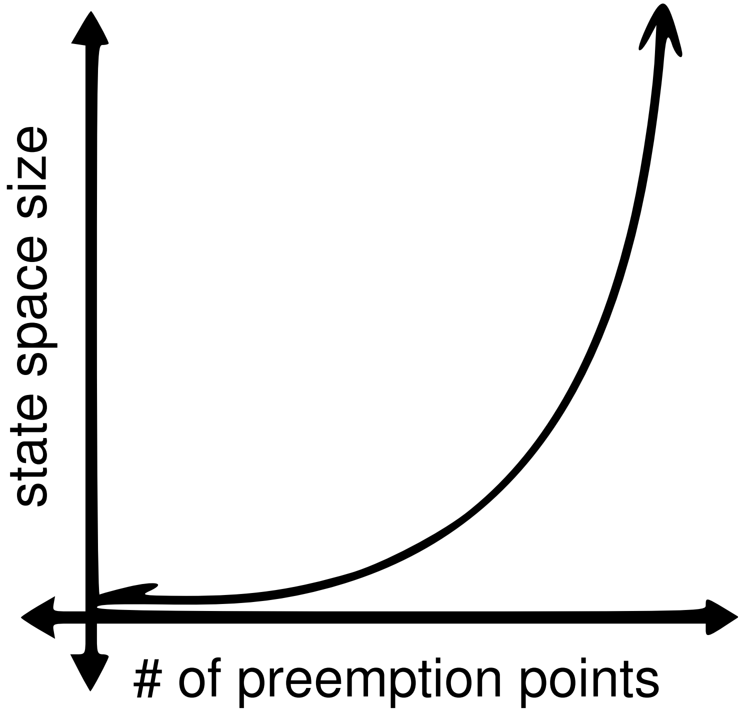
Prior work implementations hard-code a fixed set of PPs.

- dBug [Simsa '11]: pthread library calls
- Landslide [Blum '12]: kernel mutex locking/unlocking

# Challenges

Parameters of systematic tests must be kept small.

- Test program length / number of threads
- Number of preemption points used



# Coping with Exponential Explosion

Some prior work focuses on improving *coverage* with...

- Reduction techniques (identifying equivalences)
  - *[Flanagan '05, Guo '11, Huang '15]*
- Determinizing runtimes *[Cui, Simsa, Blum '13]*

Other prior systems give up on 100% completion...

- Heuristic exploration ordering (ICB) *[Musuvathi '08]*
- Random distribution of PPs *[Fonseca '14]*

# Coping with Exponential Explosion

**Current systematic testing model not user-friendly.**

- Test framework: “I want to use these PPs, but can't predict how long until completion.”
- User: “I have 16 CPUs and 24h to test my program.”

Stress testing offers this... can we offer it too?

# Our Technique: Iterative Deepening

# Iterative Deepening

Goal: Run the best tests for a given CPU budget.

- User studies with 15-410 (operating systems)
  - Students worked best with an iterative process
  - "Start small, then add more preemption points"

# Iterative Deepening

Goal: Run the best tests for a given CPU budget.

- User studies with 15-410 (operating systems)
  - Students worked best with an iterative process
  - "Start small, then add more preemption points"

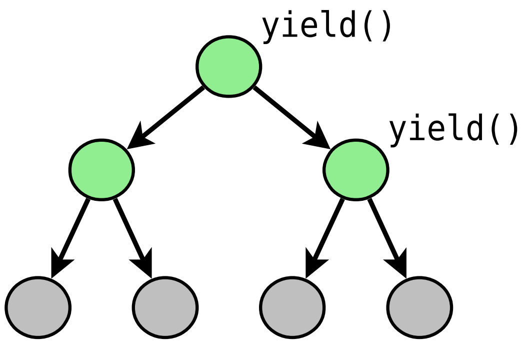
Named after analogous technique in chess AI.

- Chess search is DFS limited by max # of moves (ply).
- Chess AIs repeat DFS, increasing ply, until timeout.

# Iterative Deepening

"Minimal" state space == mandatory thread switches only

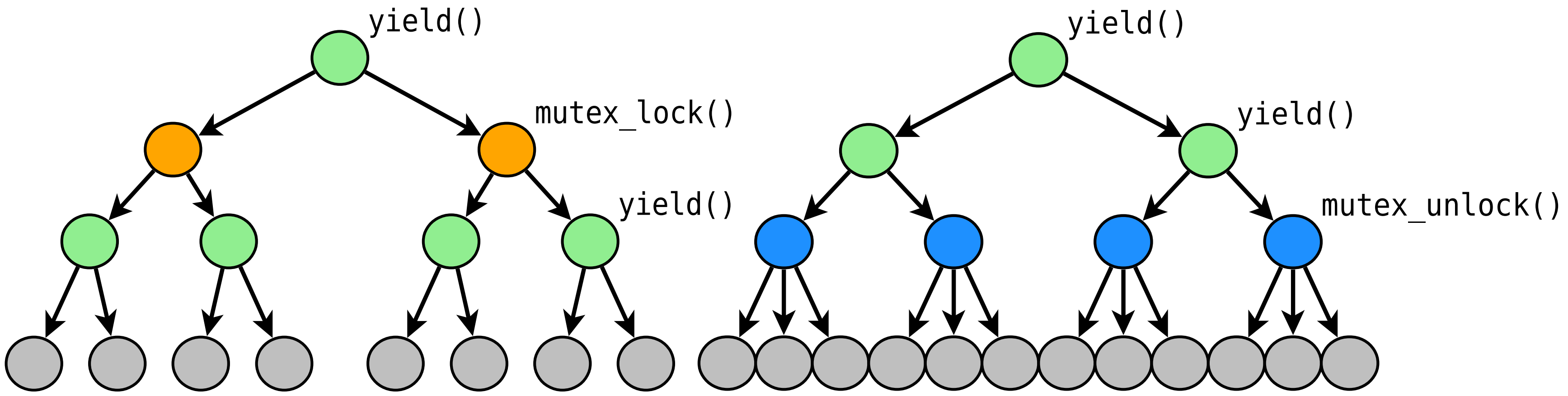
- `yield()`
- `cond_wait()`



# Iterative Deepening

Different PPs can produce state spaces of different sizes.

Testing them in parallel hedges our bets.

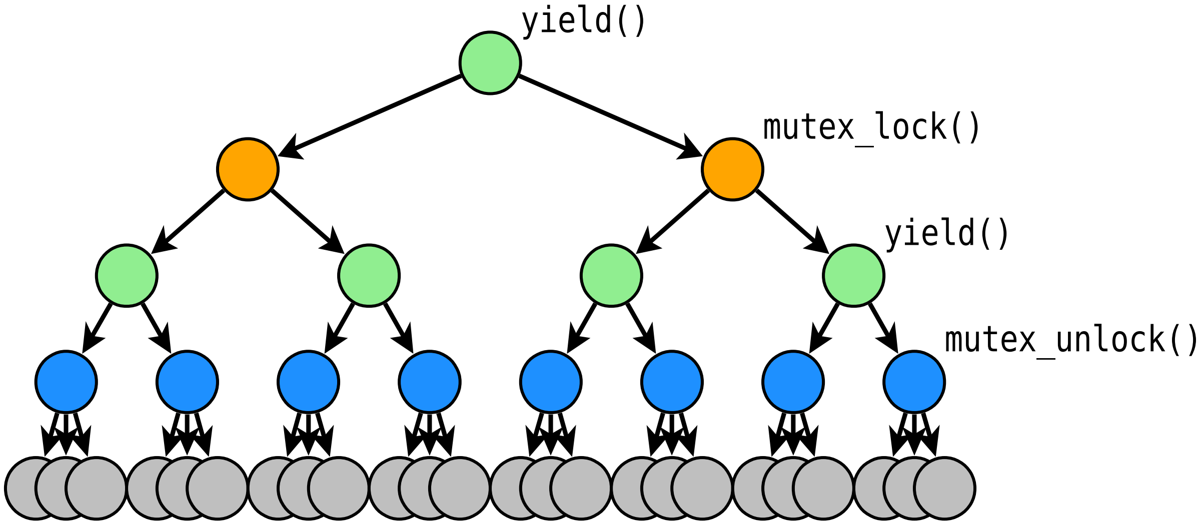


# Iterative Deepening

If time allows, we combine PPs to produce larger tests.

All PPs enabled = “maximal” state space

- Prior work tools explore this state space only.



# Choosing Preemption Points

How do we choose good preemption points?

- Start with “hardcoded” PPs as candidates
- Dynamic analysis during run can find new candidates:
  - Ad-hoc yield or busy loops
    - » Avoid cyclic state spaces
  - Data races *[Savage '97]*
    - » Including atomic operations enables testing locks, lock-free queues, etc.

# Data-Race Preemption Points

**Data race:** 2 threads access the same memory, and:

- At least one access is a write
- Sets of locks held by each thread do not overlap
- No sync-enforced ordering (e.g. `cond_signal()`)

Let's preempt threads during these accesses!

- Preempting only on API boundaries can miss bugs
- 1-pass data-race analysis has false positive problem
  - Using data races as PPs automatically verifies or refutes each data race candidate.

# Landslide

**Landslide** *[Blum '12]*: simulator-based tester

- Originally built to test 15-410 Pebbles kernels
  - (a *landslide* tests the stability of *pebbles*)
- Wind River Simics: Full-system x86 simulator
  - Supports instruction/memory tracing, backtracking
  - No additional overhead for data race analysis

Landslide's features

- DPOR *[Flanagan '05]* prunes equivalences
- State space estimation guesses total time *[Simsa '12]*
- Outputs tabular *preemption traces* to aid debugging

# A bug was found!

Current stack (TID 4):
0x1000e03 in **cond_broadcast** (cond_var.c:123)
0x100057b in **thr_exit** (threads.c:218)
0x1000059 in **main** (thr_exit_join.c:30)

**USE AFTER FREE - read from 0x10070a8 at eip 0x1000e03**

Heap block [0x10070a8 | 12] was allocated at:
0x1001253 in **malloc** (malloc.c:34)
0x1000ab1 in **cond_wait** (cond_var.c:68)
0x10003d3 in **thr_join** (threads.c:174)
0x1000018 in **waiter** (thr_exit_join.c:13)
0x10006ad in **new_child** (threads.c:0)

...and freed at:
0x1001353 in **free** (malloc.c:94)
0x1000e00 in **cond_broadcast** (cond_var.c:123)
0x100057b in **thr_exit** (threads.c:218)
0x1000059 in **main** (thr_exit_join.c:30)

| TID 4 | TID 5 |
| --- | --- |
| 0x100079b in **mutex_lock** (mutex.c:61)<br>0x100048c in **thr_exit** (threads.c:203)<br>0x1000059 in **main** (thr_exit_join.c:30) | |
| | 0x0105f0d in **context_switch** <unknown><br>0x100313b in **deschedule** (deschedule.S:16)<br>0x1000b4c in **cond_wait** (cond_var.c:77)<br>0x10003d3 in **thr_join** (threads.c:174)<br>0x1000018 in **waiter** (thr_exit_join.c:13)<br>0x10006ad in **new_child** (threads.c:0) |
| 0x1000e03 in **cond_broadcast** (cond_var.c:123)<br>0x100057b in **thr_exit** (threads.c:218)<br>0x1000059 in **main** (thr_exit_join.c:30) | |

**Distinct interleavings tested: 4**
**Estimated state space size: 6.000000**
**Estimated state space coverage: 66.666667%**

# Landslide & Quicksand

**Quicksand**: wrapper program for Iterative Deepening

- Manages work queue of jobs with different PP configs
- Each job is a new state space for Landslide to explore
- Prioritizes which jobs are important / likely to finish
  - Based on nature of PPs (data races? mutexes?)
  - Based on estimated completion time

*Only required argument is user's CPU budget*

# Evaluation

# PLDI Evaluation Plan

Comparing to single-state-space (SSS) testing

- Finding bugs faster, while SSS times out
- Finding new bugs with data-race PPs

Comparing to 1-pass data-race analysis

- Avoiding false-positives & false-negatives
- No details now, happy to explain at poster session

Testing OS projects from CMU, Berkeley, and U Chicago

- CMU: 79 “P2” thread libraries
- Berkeley & U Chicago: 79 “Pintos” kernels
- Under construction, feedback much appreciated!

# Test Suite

15-410 – Undergrad Operating Systems at CMU

- P2: pthread-like userspace thread library
  - Sync primitives: mutexes, cvars, sems, rwlocks
  - Thread API: thr_create, thr_join, thr_exit
- 6 test cases (50 LOC average):
  - *`thr_exit_join`*
    - » Test of thread lifecycle
  - *`broadcast_test, paraguay, rwlock_test, paradise_lost`*
    - » Test of cvars, sems, rwlocks
  - *`mutex_test`*
    - » Test of low-level lock implementation

# `mutex_test` is special...

**Normally,** we give a lock's internal accesses a “free pass”.

- This enables productive data-race analysis elsewhere.

lock()

```
while (xchg(&lock->held,1)==1)
    yield(lock->owner);
lock->owner = gettid();

// in critical section...
// access protected data...
```

unlock()

```
lock->owner = -1;
lock->held = 0;
```

All accesses in this range are considered protected by the lock.

# `mutex_test` is special...

**In mutex_test,** we wish to verify the implicit assumption that other tests make that the locks are correct.

lock()

```
while (xchg(&lock->held,1)==1)
    yield(lock->owner);
lock->owner = gettid();

// in critical section...
// access protected data...
```

All accesses in this range are considered protected by the lock.

unlock()

```
lock->owner = -1;
lock->held = 0;
```

# Experimental Setup

79 P2s × 6 test cases × 10 CPU-hours ≈ 200 CPU-days

- 18 12-core (HT) 3.2 GHz Xeon machines, 12 GB RAM

Compare to a "control" experiment, also given 10 CPU-hours

- 1 state space, all "fixed" PPs enabled
  - mutex_lock()
  - mutex_unlock()
  - yield()

# Results

Among 474 total tests (P2+unit test pairs)...

- 23 *deterministic* bugs (e.g. use-after-free); fixed by hand

Control experiment with fixed PPs

- 70 concurrency bugs
- 90 timeouts

Quicksand with iterative deepening

- 107 concurrency bugs
  - 37 requiring data-race PPs to expose
  - `mutex_test`: 13 bugs, 12 requiring data-race PPs
- 2 bugs found without data-race PPs (control timeout)
- 2 bugs missed that control found (*explain why*)

# Conclusion

# Future Work

Immediate future

- Ongoing user study with 15-410 students
  - Evaluate effectiveness for education/grading
- Finish PLDI evaluation plan, obviously
  - Have any experiment design ideas to share?

More uncertain future

- Smarter policies for handling too-large state spaces
- Incorporate parallel DPOR to saturate CPUs better
- Applying to production, not student, code
- *maybe think of more stuff to put here, idk*

# Conclusion

**Systematic Testing**

- Controls scheduling rather than rely on randomness
- More reliable bug-finding than stress testing
- State space depends on **preemption points** (PPs)
- Completion time is exponential and unpredictable

**Iterative deepening** with Landslide & Quicksand

- Automatically searches for optimal set of PPs to use
- User need only supply CPU time budget
- Data race analysis uncovers new bugs
- PLDI submission soon; want evaluation feedback

# References

**[Godefroid '97]**

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

**[Savage '97]**

- Stefan Savage et al. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM TOCS 1997.

**[Flanagan '05]**

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

**[Musuvathi '08]**

- Madanlal Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. (a.k.a. the CHESS paper.) OSDI 2008.

**[Simsa '11]**

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

# References

**[Guo '11]**

- Huayang Guo et al. Practical Software Model Checking via Dynamic Interface Reduction. SOSP 2011.

**[Blum '12]**

- Ben Blum. Landslide: Systematic Dynamic Race Detection in Kernel Space. CMU-CS-12-118. May 2012.

**[Simsa '12]**

- Jiri Simsa. Runtime Estimation and Resource Allocation for Concurrency Testing. CMU-PDL-12-113. December 2012.

**[Cui '13]**

- Heming Cui et al. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. SOSP 2013.

**[Fonseca '14]**

- Pedro Fonseca et al. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. OSDI 2014.

# Related Work

**Systematic testing**

- MaceMC (NSDI '07) – liveness, random walking
- CHESS (PLDI '07) – iterative context bounding (ICB)
- MoDist (NSDI '09) – network/disk model checking
- dBug (SSV '10/SPIN '11) – dynamic partial order reduction
- SimTester (VEE '12) – interrupt injection, drivers
- Parrot (SOSP '13) – dBug + deterministic userspace scheduler
- SKI (OSDI '14) – randomly-chosen PPs for testing Linux
- SAMC (OSDI '14) – annotated sync primitives, better reduction

**Data race detection**

- Eraser (TOCS '97) – lock-set tracking, annotations
- DataCollider (OSDI '10) – random sampling, kernel
- RacePro (SOSP '11) – inter-process races
- Portend (ASPLOS '12) – alternate interleavings, symbolic execution

# Bonus Slides

# Testing P2 Thread Libraries

15-410 – Undergrad Operating Systems at CMU

- P2: pthread-like userspace thread library
- P3: UNIX-like x86 kernel with VM, reentrant scheduler
- Wind River Simics: Full-system x86 simulator
  - Supports instruction/memory tracing, backtracking

Landslide for 15-410

- Tests P3 kernel code with user-supplied annotations
  - e.g., thread lifecycle, runnable/blocking, mutexes
- P2 more concretely specified; can test automatically