15-410
"...*fairness disabled*..."

Proving Dekker with SPIN and PROMELA

Joshua Wise
with help from Greg Hartman
(who defended in August -- congratulations!)

April 22, 2011

Synchronization

- Project 4 due Wednesday; P3x due today
  - Everyone having fun?
- Kernel interviews
  - Expect e-mail *soon*
  - (and when you get it, respond as soon as you can!)

Show of hands

Show of hands

- Who...

Show of hands

- Who...
  - Had concurrency bugs during P2?

Show of hands

- Who...
  - Had concurrency bugs during P2?
  - Had concurrency bugs during P3?

Show of hands

- Who...
  - Had concurrency bugs during P2?
  - Had concurrency bugs during P3?
  - Had the same concurrency bugs between P2 and P3?

Show of hands

- Who...
  - Had concurrency bugs during P2?
  - Had concurrency bugs during P3?
  - Had the same concurrency bugs between P2 and P3?
  - Stayed up all weekend debugging the same concurrency bugs from P2 to P3?

Show of hands

- Who...
  - Had concurrency bugs during P2?
  - Had concurrency bugs during P3?
  - Had the same concurrency bugs between P2 and P3?
  - Stayed up all weekend debugging the same concurrency bugs from P2 to P3?
  - Really stayed up all weekend debugging the same concurrency bugs from P2 to P3, but won't admit it?

Concurrency bugs suck

- It's true.
- Race conditions make us sad.
- Wouldn't it be nice if we could prove our code free of race conditions?
  - ...or that if they had race conditions, that at least they don't produce bad behavior?
  - *...in every possible case?*

What's a race condition?

- Well, better: what's a *program*?
  - A program can be represented by *states* and *transitions*.
  - ...some of them are not so good.
- Straight-line code has just one transition per state
- Sometimes states have multiple transitions
  - If chosen *nondeterministically*, and the output changes, then we have a race condition.
  - Interrupts, anyone?
- Not all races are bad.
  - Program may well "converge" on correct state.

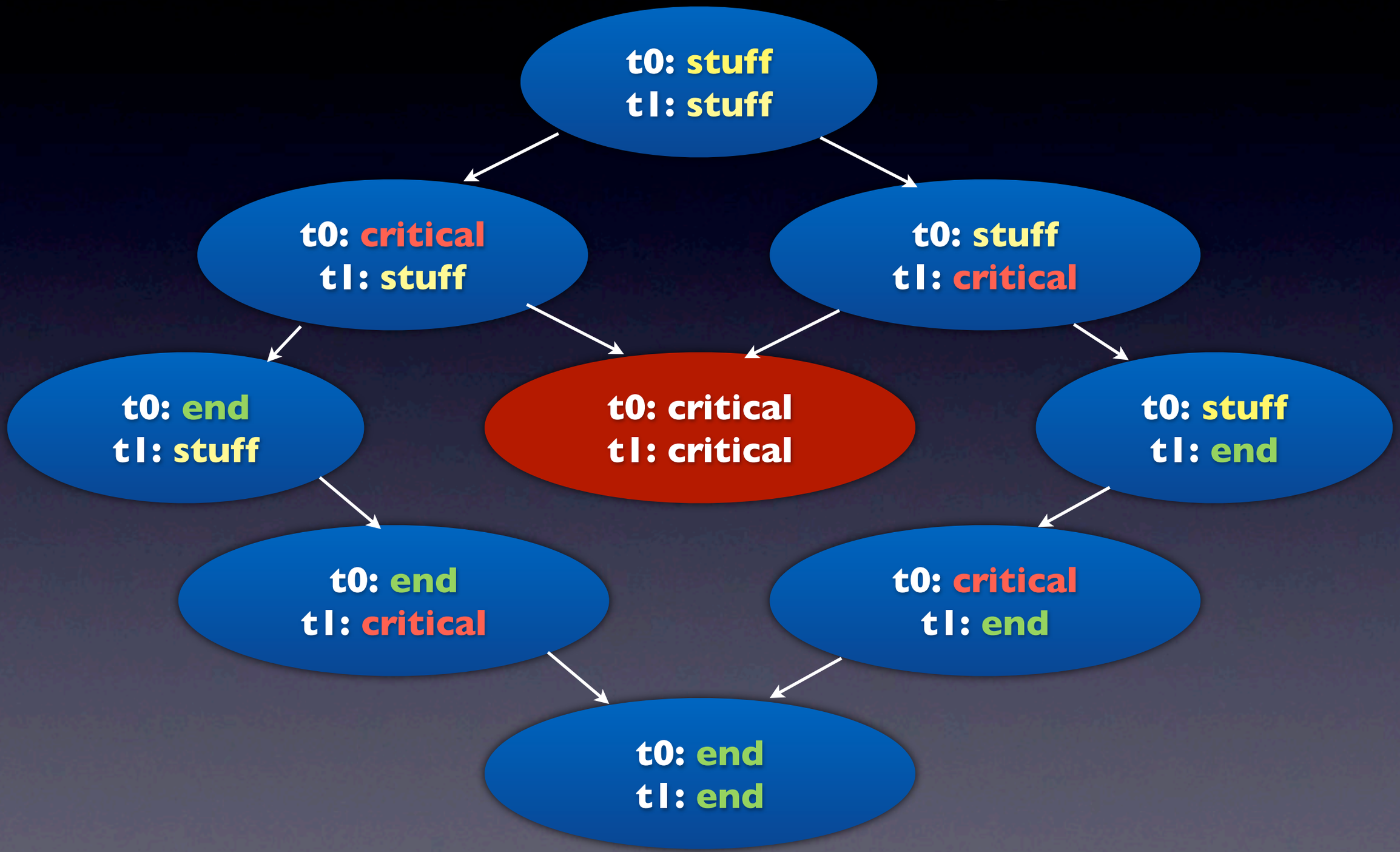
Multiple threads

- Threads change the game a bit
- Things are executed simultaneously, and *always* nondeterministically...
  - ...which is one *big* race condition!
- So which races will make our program sad?

Don’t try this at home

- Let’s take an example: *the saddest mutexes*.
- **`mutex_lock`** is a no-op, and **`mutex_unlock`** is, too.
  - (ouch!)
- We’ll have two threads, each of which looking like:
  - ```
    stuff
    mutex_lock();
    critical section
    mutex_unlock();
    end
    ```
- Let’s try to show that this won’t work.
    ```

The saddest diagram



See?

- That was no good!
- Path existed to something we called "bad"
  - (i.e., system crashed!)
- Key idea: after *every* instruction, an interrupt could occur.
  - Could cause a thread switch to *N steps* of another thd.!
- Why not do this kind of graphing on real code?
  - Maybe we'll get a few sheets of graph paper to take care of it

A bit of a problem

```
joshua@escape:~/p3$ find kern/ -iname '*.[chS]' |
                    xargs wc -l | grep total
  8096 total
```

Well, a... big... problem...

- For real code, *too many nodes to draw.*
  - (Especially with my poor Keynote skills!)
- Try doing this on mutexes that *do* work!
  - *Many* more states than simple spinlocks
  - Adding just *one* `int` to a mutex adds *four billion* states to our graph!

Introducing SPIN

- We are computer scientists
  - Well, except for me, I'm an ECE

Introducing SPIN

- We are computer scientists
  - Let's insert a layer of indirection!
- SPIN compiles things that look like programs into things that look like states
  - ...and exhaustively executes them
  - Developed at Bell Labs in the original UNIX group, starting in *1980*
- Some questions raised
  - What do we look for?

Introducing PROMELA, too

- Programs specified in a C-ish language called PROMELA
  - "PROcess MEta LAnguage"
- Originally designed for verifying hardware
- We'll convince it to verify some locking algorithms for us
- *Disclaimer: I am not a PROMELA guru!*

Don't try this at home, part 2

```
byte t0_incrit = 0;
byte t1_incrit = 0;

inline mutex_lock()
{
        skip;
}

inline mutex_unlock()
{
        skip;
}

proctype A()
{
        t0_incrit = 0;  /* Some stuff */
        mutex_lock();
        t0_incrit = 1;  /* Hi! */
        t0_incrit = 0;  /* Ok, done */
        mutex_unlock();
}
```

```
proctype B()
{
        t1_incrit = 0;  /* Some stuff */
        mutex_lock();
        t1_incrit = 1;  /* Hi! */
        t1_incrit = 0;  /* Ok, done */
        mutex_unlock();
}

init {
        run A();
        run B();
}
```

What about the tiberium?

- Let's run it:
  - ```
    $ spin -a saddest.promela
    $ gcc -o pan pan.c
    $ ./pan
    (Spin version 4.3.0 -- 22 June 2007)
                + Partial Order Reduction
    $
    ```
- Nothing happened?
  - Not quite.
- SPIN checked all states...
  - ...but not for anything useful.
    ```

Verifying mutual exclusion

- Some changes:
  - ```
    proctype monitor()
    {
        assert(!(t0_incrit && t1_incrit));
    }

    init {
        run A();
        run B();
        run monitor();
    }
    ```
- Key point of **`monitor`**
  - *it could happen anywhere!*
  - Effectively happens *everywhere*, then.
    ```

Run it again

- Another build cycle:
  - ```
    $ spin -a saddest.promela
    $ gcc -o pan pan.c
    $ ./pan
    pan: assertion violated !((t0_incrit&&t1_incrit))
                              (at depth 9)
    pan: wrote saddest.promela.trail
    [...]
    ```
- Aha!
- The assertion was violated, and it wrote out a trail telling us where.
    ```

What went wrong?

- XSPIN can tell us, or **`pan`** can tell us
- **`pan -C`**: read in the trail, and give annotation
  - ```
    $ ./pan -C
    1:  :init:(0):[(run A())]
    2:  :init:(0):[(run B())]
    3:  :init:(0):[(run monitor())]
    4:                                  B(2):[t1_incrit = 0]
    5:                                  B(2):[(1)]
    6:                                  B(2):[t1_incrit = 1]
    7:                  A(1):[t0_incrit = 0]
    8:                  A(1):[(1)]
    9:                  A(1):[t0_incrit = 1]
    pan: assertion violated !((t0_incrit&&t1_incrit))
                                                    (at depth 10)
    spin: trail ends after 10 steps
    ```
- *Concrete path*: what went wrong
    ```

Slightly happier mutexes

- One more example
- Spinlocks: this time, with more mutual exclusion.
- `mutex_lock` will spin-wait on a single 'locked' variable
- For the example we gave, guaranteed to terminate -- why?

New PROMELA Code

don't you wish C had that keyword?

```
byte locked = 0;

inline mutex_lock()
{
        do
        :: 1 ->
              atomic {
                if
                :: locked == 0 -> locked = 1; break;
                :: else -> skip;
                fi
              }
        od
}
inline mutex_unlock()
{
        assert (locked == 1);
        locked = 0;
}
```

Any better?

- Well, let's find out:
  - ```
    $ spin -a happier.promela
    $ gcc -o pan pan.c
    $ ./pan
    [...]
    State-vector 28 byte, depth reached 27, errors: 0
    [...]
    ```
- Sweet.
- But what about other properties of mutexes?
  - Progress
  - Bounded Waiting
    ```

Unbounded waiting

- PROMELA feature: "progress cycles"
  - *Don't get hosed!*
  - In an *infinite cycle*, a progress cycle must be hit *infinitely often*.
- Who can tell me an execution path that makes spinlocks sad?
- Let's see if we can get SPIN to make this for us
- Game plan: insert 'progress' cycle in a loop after both people get to run.

New monitor process

```
proctype fairness() {
        do
        :: 1 ->
                t0_incrit -> skip;
                t1_incrit -> skip;
progress:       skip
        od
}

[...]
        run fairness();
[...]
```

```
proctype A()
{
        do
        :: 1 ->
                mutex_lock();
                t0_incrit = 1;
                t0_incrit = 0;
                mutex_unlock();
        od
}
```

Is it fair?

```
$ spin -a happier-waiting.promela
$ gcc -o pan pan.c -DNP
$ ./pan -l
pan: non-progress cycle (at depth 20)
pan: wrote happier-waiting.promela.trail
[...]
State-vector 36 byte, depth reached 35, errors: 1
[...]
```

- As we guessed, something went wrong
- SPIN made a state tree for us
  - ... and found a set of transitions that break bounded waiting!

What went wrong? 2

```
$ ./pan -l -C
[...]
<<<<<START OF CYCLE>>>>>
22:                    B(3):[((locked==0))]
24:                    B(3):[break]
26:                    B(3):[t1_incrit = 1]
28:                    B(3):[t1_incrit = 0]
30:                    B(3):[assert((locked==1))]
32:                    B(3):[locked = 0]
34:                    B(3):[(1)]
36:                    B(3):[(1)]
```

Progress

- Third property of mutexes
- We'll define it as:
  - Given code running infinitely long, infinitely often *somebody* will obtain release the mutex.
    - Not quite correct definition, but serves as OK example
- Use progress cycles again to define this

New PROMELA Code

- Remove `fairness` monitor, and put progress cycles in procedures

```
proctype B()
{
        do
        :: 1 ->
                mutex_lock();
progress:       t1_incrit = 1;
                t1_incrit = 0;
                mutex_unlock();
        od
}
```

Verifying progress

```
$ spin -a happier-progress.promela
$ gcc -o pan pan.c -DNP
$ ./pan -l
pan: non-progress cycle (at depth 20)
pan: wrote happier-progress.promela.trail
```

- WTF???
  - We know these mutexes provide progress; SPIN said they didn't
- What happened?

```
$ ./pan -l -C
[...]
20:                                   B(3):[((locked==0))]
<<<<<START OF CYCLE>>>>>
22:                  A(2):[else]
23:                  A(2):[(1)]
25:                  A(2):[(1)]
```

Even when it's going wrong

```
$ ./pan -l
[...]
Full statespace search for:
        non-progress cycles     + (fairness disabled)
[...]
```

- ...oooh.
- Of course we can't have progress...
  - if the scheduler won't schedule us!
  - (What if a meteor hits the stadium?)

Fair progress

- ```
  $ pan -l -f
  [...]
  State-vector 52 byte, depth reached 80, errors: 0
  ```
- (-f is for “weak fairness”)
- Once the scheduler schedules us...
  - ...the world is much happier!
- We have shown that...
  - No state that the world grinds to a halt
  - If we keep going back and forth, repeatedly *somebody* will win
  ```

On to Dekker

- Dekker's Algorithm
  - T. J. Dekker, 1964; you know it as "taking turns when necessary"
  - 2 threads only
  - Guarantees the Big Three
- Let's try to prove it
  - Mutual exclusion
  - Bounded waiting
  - Progress
- SPIN style warning!
  - Namely, mine is bad. (I am not a professional!)

Mutual exclusion

- Let's write some PROMELA!

```
proctype A()
{
  f0 = 1;
  do
  :: f1 ->
    if
    :: turn != 0 ->
      f0 = 0;
      turn == 0 -> skip;
      f0 = 1;
    :: else -> skip;
    fi
  :: else -> break;
  od;

  t0_incrit = 1;
  t0_incrit = 0;

  turn = 1;
  f0 = 0;
}
```

```
proctype B()
{
  f1 = 1;
  do
  :: f0 ->
    if
    :: turn != 1 ->
      f1 = 0;
      turn == 1 -> skip;
      f1 = 1;
    :: else -> skip;
    fi
  :: else -> break;
  od;

  t1_incrit = 1;
  t1_incrit = 0;

  turn = 0;
  f1 = 0;
}
```

And, as we expect...

```
$ spin -a dekker.promela
$ gcc -o pan pan.c
$ ./pan
[...]
State-vector 32 byte, depth reached 29, errors: 0
[...]
```

- If both threads run "acquire lock; release lock" once...
  - we've shown mutual exclusion for *all reachable states*
- Can we show that they'll always have bounded waiting?

Finite waiting

- Let's try a different tactic
  - "Every time you are waiting for the lock, you eventually get the lock."

```
proctype fairness0() {
        f0 -> t0_incrit;
}

proctype fairness1() {
        f1 -> t1_incrit;
}
```

- Procedures go in loops, like before
- Slightly *weaker* form of bounded waiting
  - ...but much easier to show in a few slides.

As we expect 2

```
$ spin -a dekker-waiting.promela
$ gcc -o pan pan.c
$ ./pan
[...]
State-vector 40 byte, depth reached 106, errors: 0
[...]
```

- *Yes!*
- Everything reached a valid end state, so there are no errors
  - *Always!*
- Dekker's algorithm provides *finite* waiting

Dekker provides progress

- Similar modifications -- drop a progress cycle in the critical section

```
$ spin -a dekker-progress.promela
$ gcc -o pan pan.c -DNP
$ pan -l
pan: non-progress cycle (at depth 16)
pan: wrote dekker-progress.promela.trail
$ pan -l -f
State-vector 52 byte, depth reached 113, errors: 0
```

- Isn't that depth a bunch deeper?
  - Only 80 before for the simple ones
- *Every possible path* will make progress

Limitations

- State explosion problem
  - Machines are better than people...
  - ...but only by six or seven orders of magnitude.
  - Running RCU verification on Linux
    - 1 updater, 1 reader: 2.6MByte
    - 1u, 2r: 2.9 MByte
    - 2u, 2r: 75.4 MByte
    - 2u, 3r: 2,715.2 MByte...
    - 3u, 2r: **14,979.9 MByte!**
- Small-world theory
  - Extending proof to more
  - Non-trivial? Trivial?

Summary

- Race conditions suck
- It is possible to think of programs in a more deterministic fashion
- Programs exist to prove other programs
  - ...but those programs are mostly science fair toys.
- These programs prove using...
  - *simplified* models
  - *state traversals*
  - *exhaustive search*

References

- SPIN homepage: http://www.spinroot.com/
- Book: *The Spin Model Checker*
- LWN article on RCU: http://lwn.net/Articles/243851
- PROMELA files available to play with tonight
- Questions?