Lecture 6: Synchronization

- "Milk problem" solution
- Spin locks
- Semaphores
Safety: "bad things" shouldn't happen.

Fairness: threads should not have to artificially wait for other threads.

Liveness: "good things" eventually do happen.
\textbf{init. working}_1 = \text{false} \land \text{true if T1 is working}
\text{working}_2 = \text{false} \land \text{true if T2 is working}
\text{turn} = 0

\begin{align*}
\text{T1} & \quad 1: \text{working}_1 = \text{true} \\
& \quad 2: \text{turn} = 2 \quad \text{while turn \text{== 1} \land \text{working}_1} \\
& \quad \text{if either turn \text{== 1} or other thread not running} \\
& \quad \text{critical-section} \\
& \quad \text{working}_1 = \text{false} \\
\text{T2} & \quad 1: \text{working}_2 = \text{true} \\
& \quad 2: \text{turn} = 2 \quad \text{while turn \text{== 2} \land \text{working}_1} \\
& \quad \text{if either turn \text{== 2} or other thread not running} \\
& \quad \text{critical-section} \\
& \quad \text{working}_2 = \text{false}
\end{align*}

Know either turn = 0, there was a point in time, after line 2, at which T1 ran line 2, after T1 ran line 2, know working-1 was true when T2 ran line 2, so T2 cannot enter CS.

2. working-2 = false know there was a point in time, after T1 executed line 2, during which T2 hadn't run line 1. So for T2 to get to CS, it has to run line 2, after which, turn will never be 2, know working-1 true, T2 can't enter CS.

reasoning is complex.
(Hardware)

Instructions for synchronization:
- interruptions between load & compute & store are difficult to reason about
- atomic load, “think” store instruction that can't be interrupted would help
  test-and-set instruction
  - given an address
  - if contents are 0, set to 1, return 0
  - otherwise, keep contents same, return old contents
  (load & store at same time)

lock = 0

while test-and-set (lock):
  if returns 0:
    lock was = 0,
    nobody in c.s.
  lock = 0

spin lock

spinning
(polling)

or if other
thread is
running on
another pro,
otherwise: yield() to other threads.
Compare-and-swap instruction (CAS)

CAS location, old value, new value.

(Informally) if value in location is old value, replace it with new value, return true.
otherwise do nothing, return false.

in either case, return current value in location.
return true if actually swapped.

Typical use:
load "observe state of world"

Computation

CAS compare current state of world to what it was when you read, if it’s same, update based on computation

Increment i:

load i → r1
inc i → r2
CAS i, r1, r2 → replace it with new value.

loop until CAS indicates successful swap.
Optimistic Concurrency

- increment locally, assuming no conflict.
- when done, check for conflict, redo work if so.

$i = 0$

$T_1$
- success = false
- while not success:
  - $r_1 = i$
  - $r_2 = r_1 + 1$
  - success = CAS $i$, $r_1$, $r_2$

$T_2$
- success = false
- while not success:
  - $r_1 = i$
  - $r_2 = r_1 + 1$
  - success = CAS $i$, $r_1$, $r_2$

\[ i \quad | \quad r_1 \quad | \quad r_2 \quad | \quad \text{success} \]
\[ 2 \quad | \quad 1 \quad | \quad 2 \quad | \quad \checkmark \]
\[ i \quad | \quad r_1 \quad | \quad r_2 \quad | \quad \text{success} \]
\[ 0 \quad | \quad 0 \quad | \quad 1 \quad | \quad \checkmark \]
Higher-level primitives for processes to use to protect critical sections (manage resources & communicate)

Semaphore is an object that encapsulates a counter.

```python
class Semaphore:
    init(initial value)  # invariant: semaphore is always > 0
    V()  # incr()
    P()  # decr()  # block until value is > 1 before decrementing.
```

"verhooft" release.
"probe"
Critical Sections with sema

Init: lock = Semaphore(1)

T1:  
- P(lock)
- Critical_section
- V(lock)

T2:  
- P(lock)
- CS
- V(lock)

Semaphore
init: \( i = 0 \)

\[ T_1: \quad i++ \quad (i \leftarrow i+1) \]

Load \( i \rightarrow r1 \)
Incr \( r1 \)
Store \( r1 \rightarrow i \)

\[ T_2: \quad i++ \]

Load \( i \rightarrow r1 \)
Incr \( r1 \)
Store \( r1 \rightarrow i \)

\( i \) for \( T_1 \)
\( r1 \) for \( T_2 \)

- Lost update
- Reasoning about sync. not compositional