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.

init: working_1 = False # true if T1 is working
working_2 = false # " T2 "
turn = 0

T1

```
1: working_1 = true
2: turn = 2
   while turn ≠ 1 and working_2:
       pass
   # either turn = 1 or other thread not running
   critical_section
   working_1 = False
```

T2

```
   working_2 = true
2: turn = 1
   while turn ≠ 2 and working_1:
       pass
   # either turn = 2 or other thread not running.
   enter section.
   working_2 = false
```

know either

turn=1 case: ① there was a point in time, after line 2, at which turn=1, know T2 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 @ same time)

Spin lock

```
lock=0    # 1 if a thread in CS.

while test_and_set (lock):
    pass

CS,

lock=0
```

↑ if returns 0, lock was =0, nobody in C.S.

← spinning (polling)

OK if other thread is running on another proc,

Otherwise: yield() to other threads.

Compare_and_swap instruction (CAS)

CAS location, old_value (expected), new_value.

(atomically) 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.

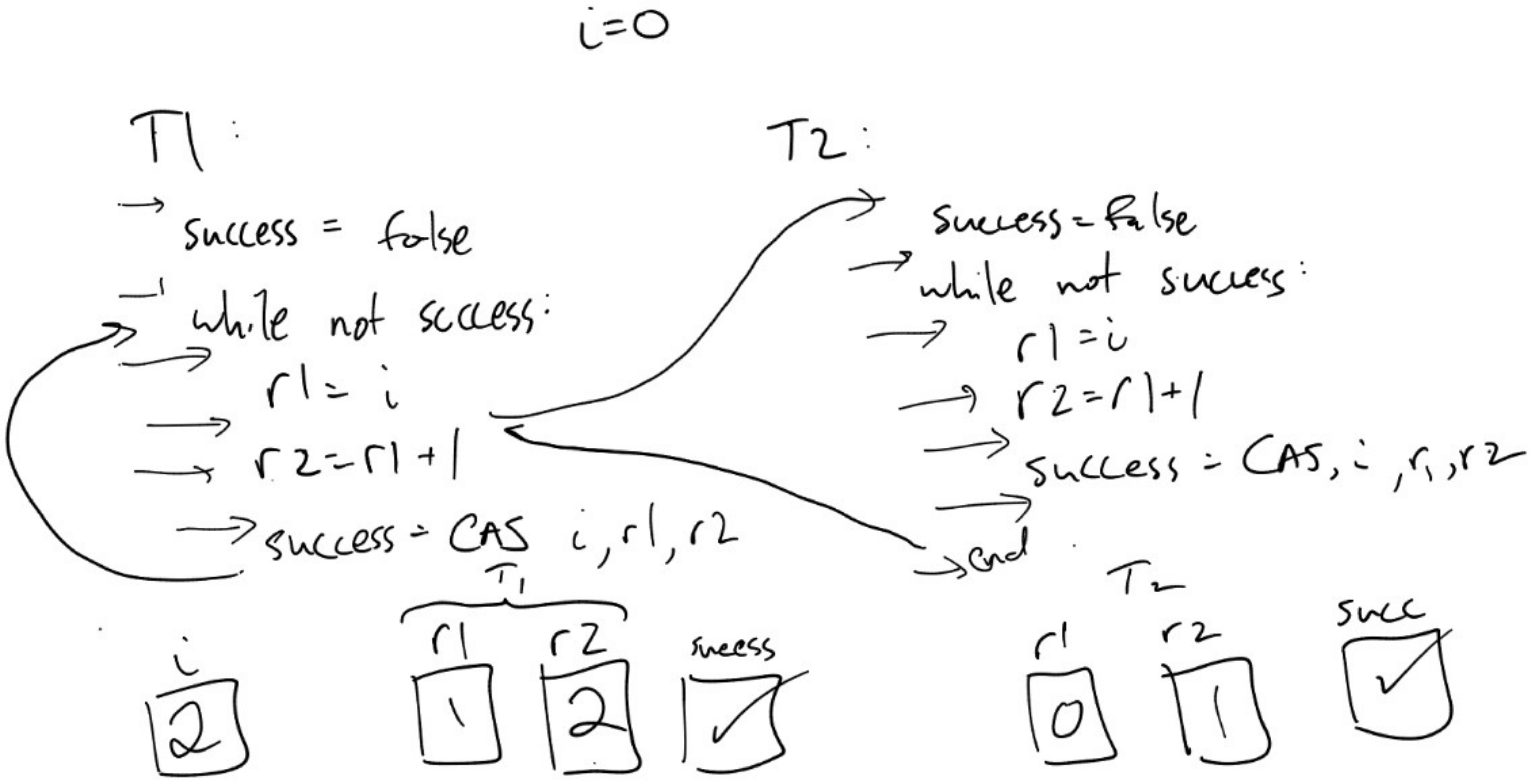
Typical use:

- 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
incr i → r2
CAS i, r1, r2   ← if i still has old value, 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.

Higher-level primitives for processes to use to protect crit. sections (manage resources & communicate)

Semaphore is an object that encapsulates a counter.

class Semaphore:

init(initial value)

invariant: semaphore is always ≥ 0

"verhook" release. V() incr()

"probe" P() decr() ← block until value is ≥ 1 before decrementing.

Critical sections with sema

init: lock = Semaphore(1)

T1:

```
P(lock)
Critical_section
→ V(lock)
→
```

T2:

```
P(lock)
→ CS.
→ V(lock)
→
```

Sema

1

init: i=0

T1: i++ (i=i+1)

- ✓ load i→r1
- ✓ incr r1
- ✓ store r1→i

T2: i++

- ✓ load i→r1
- ✓ incr r1
- ✓ store r1→i

- lost update
- reasoning about synch. not compositional.