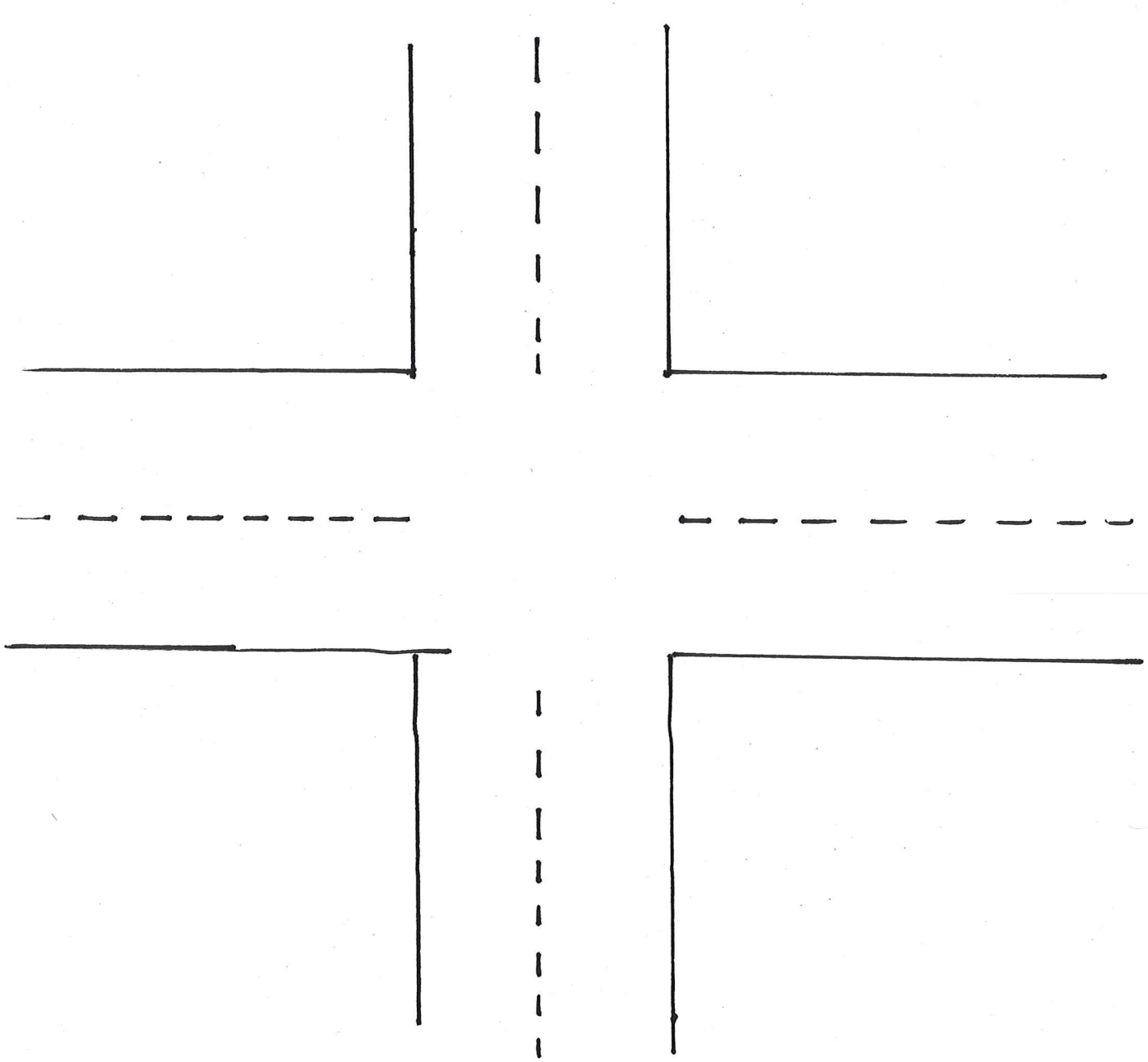
Lec 10

Deadlock

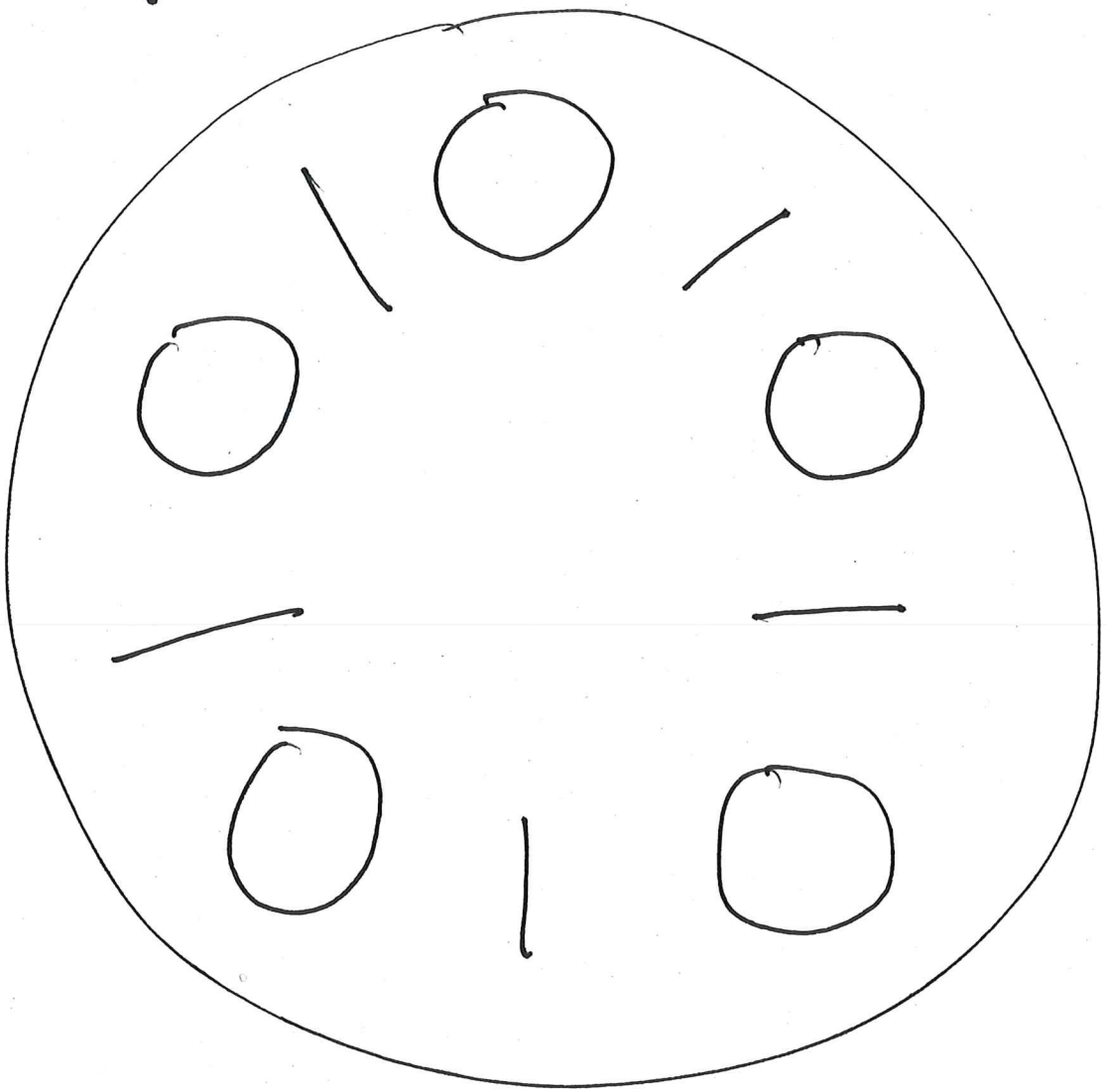
```
b1 = new BlockingQueue()
b2 = new BlockingQueue()
```

T1	T2
b2.put(foo)	b2.put(bar)
b2.get()	b1.get()

if both buffers are almost full,
both threads wait for the other forever



Dining Philosophers



Necessary Conditions for Deadlock

bounded resources

no resource preemption

wait while holding

circular waiting

Eliminating any one of these guarantees deadlock freedom

Preventing Deadlock

avoidance: write the program so that ≥ 1 necessary condition is false

prediction: wait until all future resources are available
then atomically acquire all of them

detection: undo changes/operations that caused deadlock

Avoiding the four conditions

- **bounded resources:** make sure there are enough resources
- **no preemption:** forcibly reclaim resources
- **wait while holding:** don't wait or don't hold
- **circular waiting:** acquire resources in some global order

Predicting the future

- simple version: declare max resources up front
  - delay requests until those resources are free

Banker's Algorithm:

grant requests as long as there is <u>some</u> sequence of thread executions that allows each granted request to receive its max resources

(assume each thread eventually requests resources + finishes)

in other words, grant requests as long as system is <u>safe</u>

Safe / Unsafe States

a state is safe if there is some sequence of granted requests that allows each to receive its requested resources and then finish

unsafe state: no such sequence

system has some set of granted requests which declare their max resource needs

each granted request eventually requests resources ($\leq$ max) and then finishes (releasing all resources)

Deadlock detection + recovery

detect: compute the waits-for graph + check for cycles

recover:

- proceed without the resource
- rollback + retry (transactions)

Lock-free data structures

safe, concurrent operations without locking

Building block: compare-and-swap (CAS)

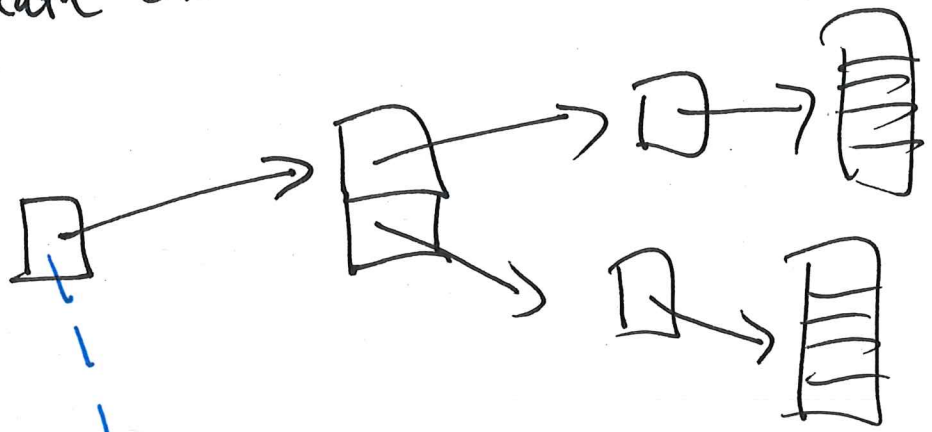
CAS(ptr, x, y) returns bool success

atomically:

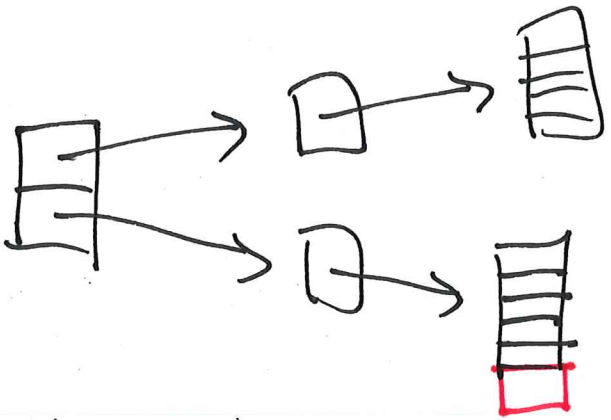
```
if *ptr == x:
    then *ptr = y; return true
else return false
```

Using CAS

old data structure:



to do an update, first make a copy with the update



then "swing" the ptr. to point to new copy

retry if error

GC after success

(can optimize to share common parts)

Performance on multicore

- ideally, a program with N threads would run N times faster (if N ≤ # CPU cores)

- may not achieve ideal:
  - only one thread can hold lock at a time
  - shared mutable data must move between cores
  - false sharing

Cache Coherence

T1	T2
acquire() x++ release()	acquire() print(x) release()

if T1 ~~[illegible]~~ runs on core 1, x will be in core 1's cache

if T2 runs on core 2,
how does core 2 access x?

Cache Coherence: behave as if one copy of data