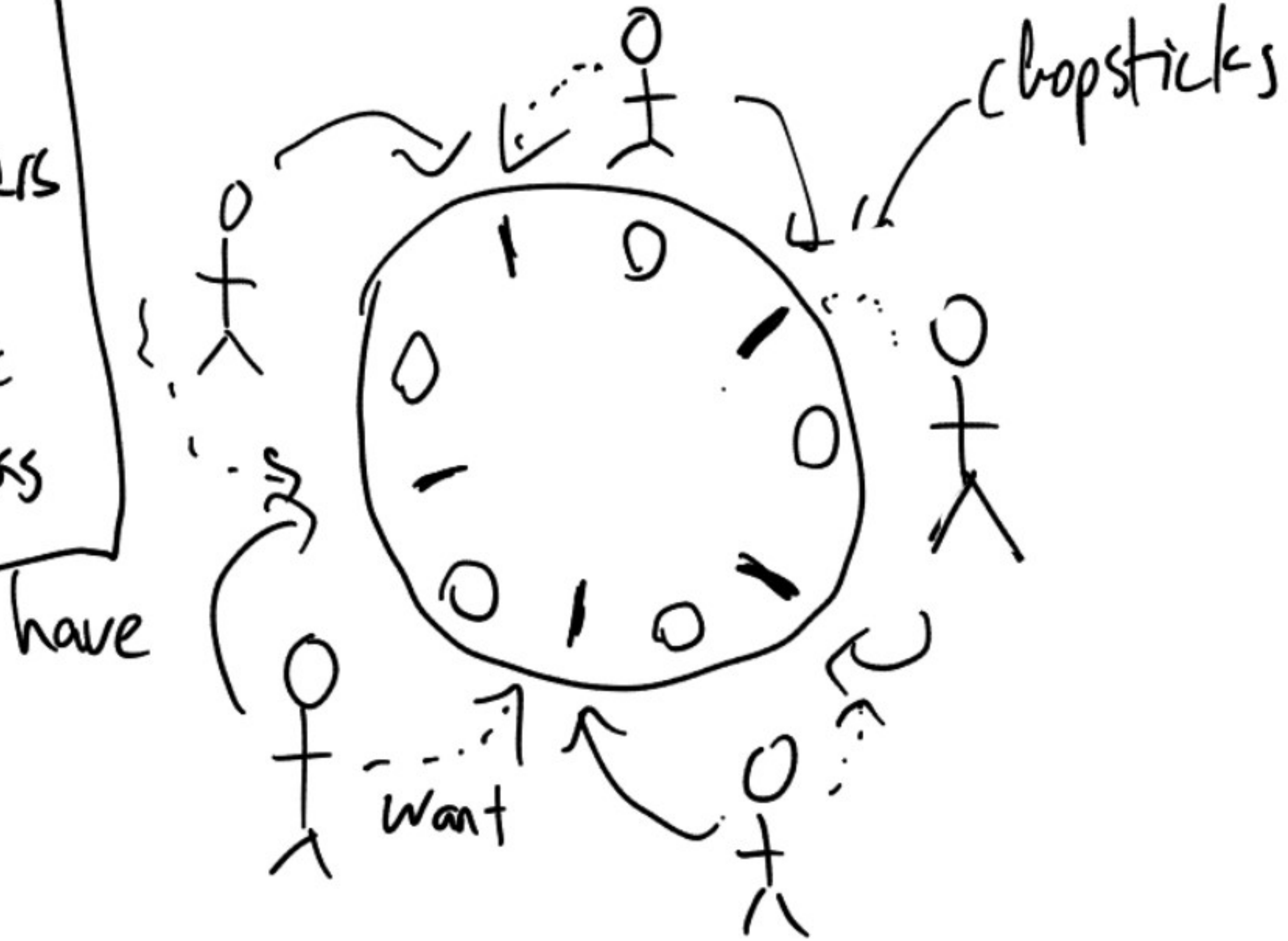
4410 Lecture 10: Deadlock

- Conditions for deadlock
- Deadlock prevention
- Deadlock detection
- Deadlock avoidance

"Dining philosophers"

Deadlock:
no philosophers can make progress b/c all chopsticks taken

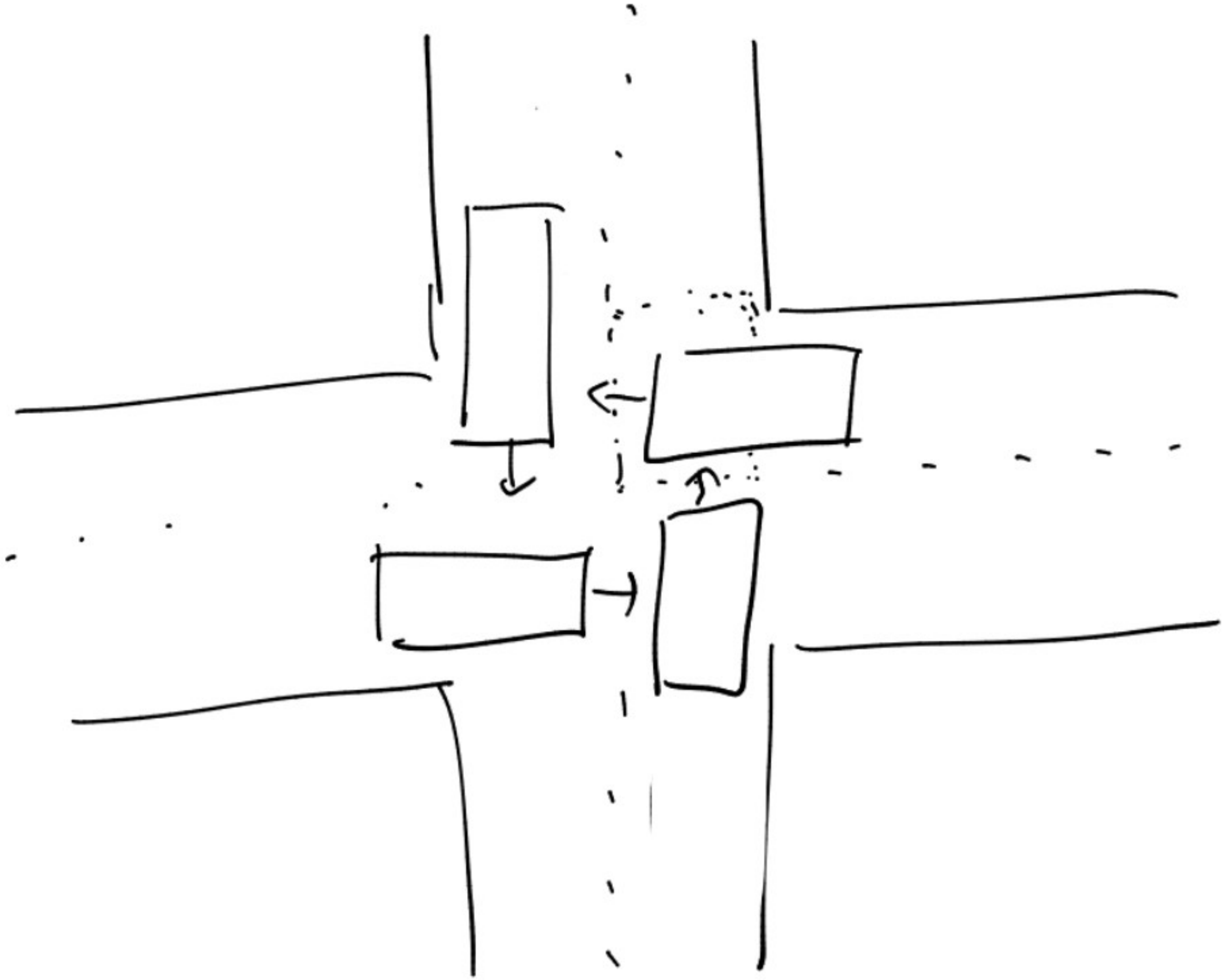


philosopher

while true:
- grab left chopstick

context switch →

- grab right chopstick
- eat()
- put sticks down
- think()



4 necessary & sufficient conditions for deadlock

Preventing any 1 of these makes deadlock impossible

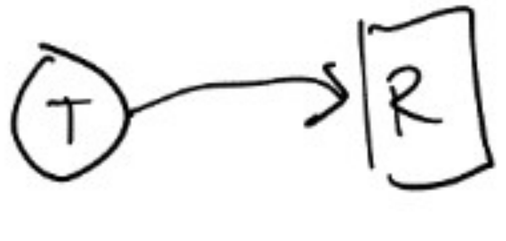
1. Mutual exclusion: can't share resource (e.g. bunch of readers can share R/W lock, can't deadlock)
2. No preemption: threads can't involuntarily give up resources (e.g. CPU can be taken away)
3. Hold and wait: thread is able to hold one resource & wait for another
4. Circular wait: in graph of processes & resources, with edges representing who holds what resource, who is waiting, need a cycle.

↑ Resource alloc. graph

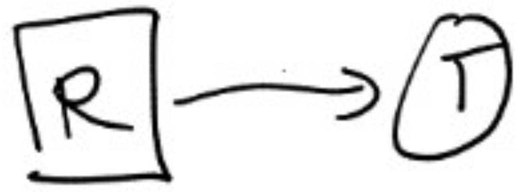
RAG


threads

resources

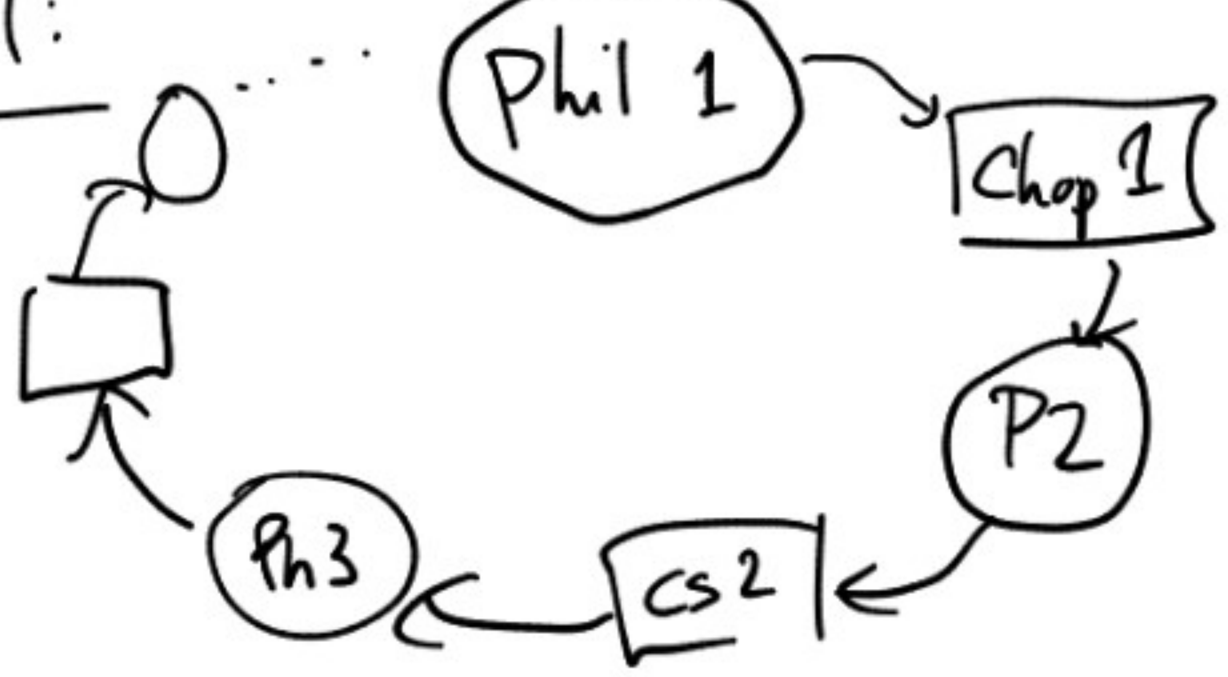


: T is trying to acquire R



: R is held by T.

Dining phil:



Phil. 2 has Chopstick 1, trying to get CS 2.

1. Preventing mutual exclusion
   - make resources shareable: virtualization.
     e.g. CPU is virtualizable

2. No preemption
   - add the ability for one thread to take resources from another.
     - safely! taking away a lock leads to lost updates.
     - "Roll back" the preempted to state before resource was acquired (by it)

       Example: thread wants to increment x
       1. acquire lock
       2. read x
       3. locally increment value
       4. write x
       5. release lock

       To pre-empt a thread here, reset all state to
       - return thread's IP to before line 1.
       - reset x to its value before lock was acquired.

     - can't roll back after doing I/O.

   - Rolling back can lead to "livelock":
     - threads seem to make progress, have to continually roll back, never actually finish
   - Example of optimistic concurrency: proceed with hope of no conflict, fix if it occurs.

3. Fixing hold and wait

- Require threads to acquire all resources at once.
- Doesn't work if first resource is needed to determine what next resource is

holds 1
waits for 2

ex: **T1**

lock file 1
read filename 2 from file 1
lock file 2
read file 2
release locks.

broken:
lock file 1
read filename 2 from f1
release file1 lock
lock file 2
read file 2
release file 2

broken reads stale file

"optimistic rollback version"

lock f1
read fn2 from file1
check whether f2 is locked
  no: read f2 & release
  yes: release f1
    acquire f1 & f2
    reread f1 to ensure fn2 unchanged
      changed
      unchanged, read f2, release.

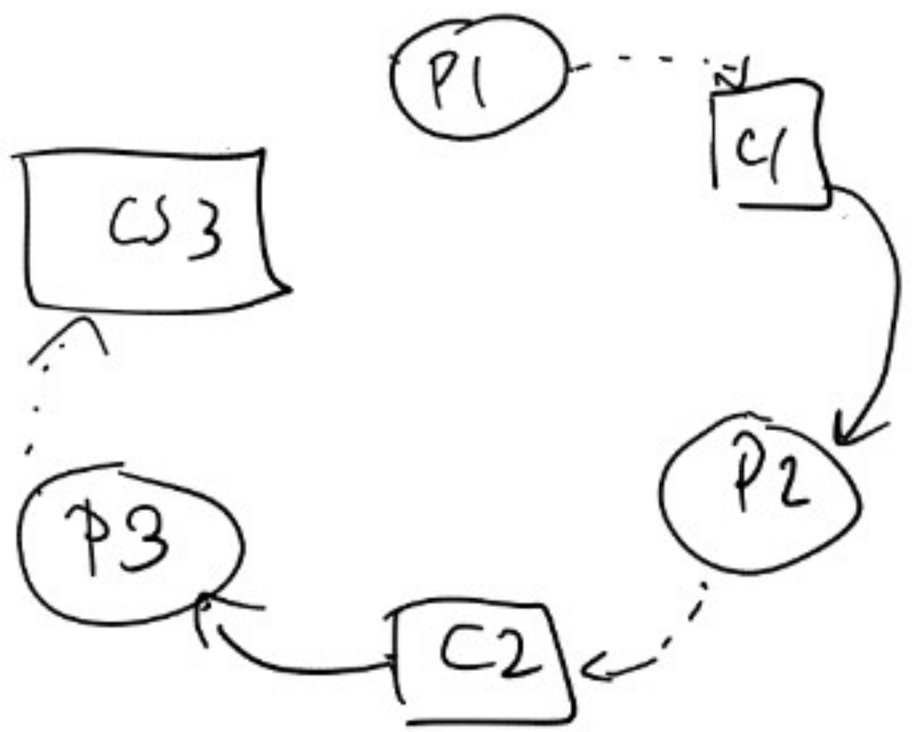
Not holding & waiting at same time

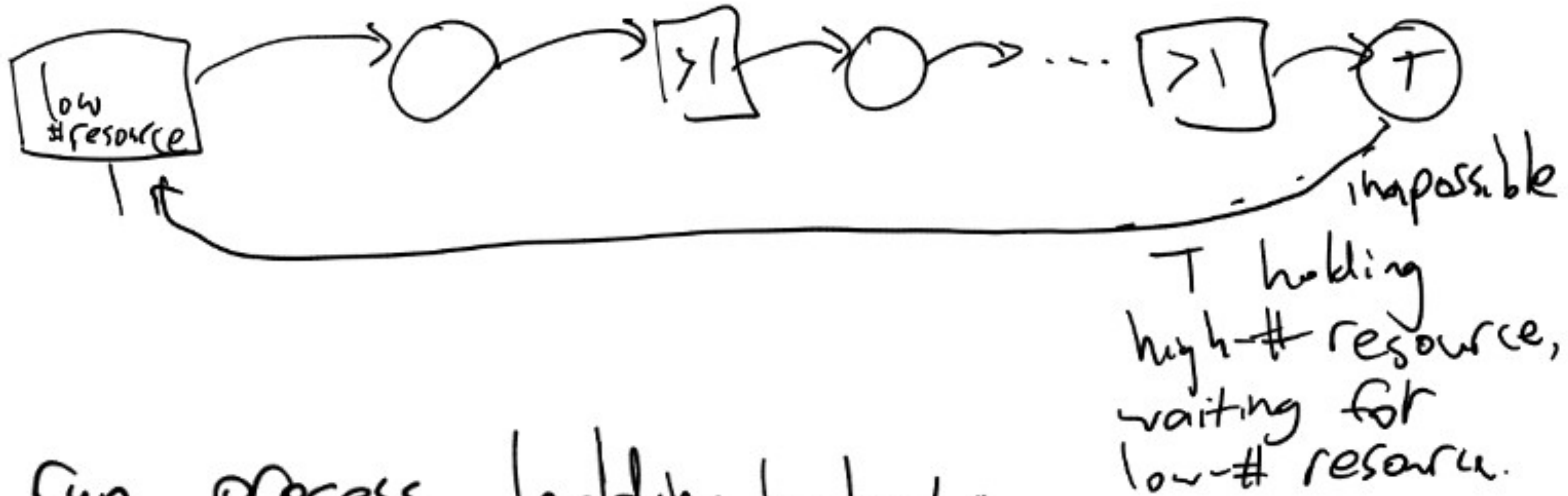
4. Breaking circular wait

- if locks are ordered, have to acquire them in increasing order, can't have a cycle.

philosopher:

- grab lower-numbered chopstick
- grab higher-numbered chopstick
- eat()
- release
- sleep()





Can always run process holding highest resource to completion.

Deadlock detection

- Maintain representation of RAG.
- periodically (or when resources requested/acquired) check for cycles.

  if found: break it by picking a process to kill / roll back.

  (roll back: safe, but requires app-specific logic)

- efficient if no deadlock
- easy to implement.

Banker's algorithm / deadlock avoidance

- Processes pre-declare what resources they might acquire. (credit limit)
- When resource is requested, Banker blocks until enough resources are available to guarantee completion w/o deadlock. (stress test)

	current A	current B	current C	current D	max A	max B	max C	max D
Free	5	3	1	2	5	3	1	2
P1					5	0	0	1
P2					1	1	1	1
P3					2	3	1	1

← Never changes (max)

have 5 A's, 3 B's, 1 C, 2 D

Def^n A current allocation is <u>safe</u> if there is some order to run procs in, guarantees all complete, even if they all acquire max resources.

Ex:

	A	B	C	D
~~P1~~	5	0	0	1
~~P2~~		1		
P3		2		1
Free	5	1	1	1

(current)

- P1 can complete because it has all its resources.
- P2 can then run, because its resources are available
- P3 has enough resources available to finish

	A	B	C	D
P1	5			1
P2		1		
P3		2		1
Free	0	0	1	0

⇓ (Run P1)

	A	B	C	D
~~P1~~				
P2		1		
P3		2		1
Free	5	0	1	1

⇓ (run P2)

	A	B	C	D
~~P1~~				
~~P2~~				
P3		2		1
Free	5	1	1	1

⇓ (run P3)

all processes done