10/26 Deadlock

Single Lock vs. Multiple Locks

→ Single lock: simple, coarse (protects the whole system), poor concurrency & performance

→ Multiple locks: more complex, fine-grained (protects specific data structure), more concurrency (more things can be scheduled), better perf.

Problems w/ Multiple Locks

Thread 1
(copy from directory A to B)
acquire(A_lock);
acquire(B_lock);

Thread 2
(copy from directory B to A)
acquire(B_lock);
acquire(A_lock);

* Deadlock: cycle of waiting amongst threads, where each thread waits for thread in the cycle to take some actions

Nested Waiting Deadlock Example

```
pipe_read() {
  acquire(global_ftable_lock);
  acquire(pipe_lock);
  while(...) {
    wait(pipe, pipe_lock);
  }
  release(pipe_lock);
  release(global_ftable_lock);
}
```

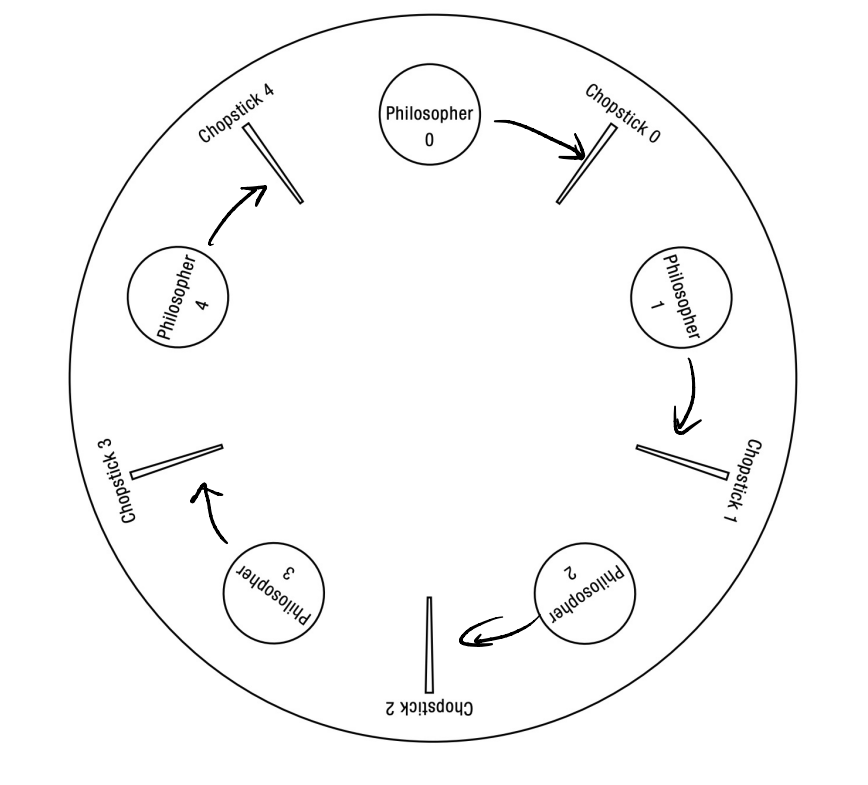
t1 has … releases

```
pipe_write() {
  acquire(global_ftable_lock);
  acquire(pipe_lock);
  signal(pipe);
  release(pipe_lock);
  release(global_ftable_lock);
}
```

t2 … can't acquire … can't make progress to signal reader

* Can end up with deadlocks with condition variables & even if you acquire locks in the same order.

Dining Philosopher



5 Philosophers (each needs 2 chopsticks to eat)

5 chopsticks

Necessary (but not sufficient) Conditions for deadlock

① Bounded Resources

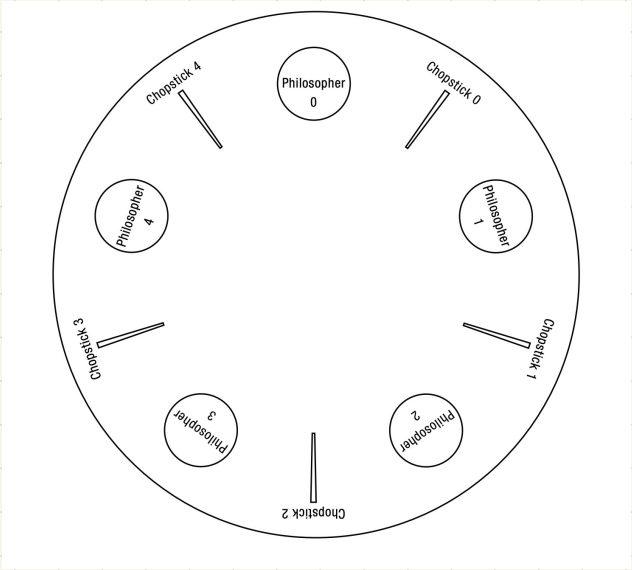
② No preemption

③ Hold & wait

④ Circular Wait

Deadlock Prevention

→ structure the program so that one of the necessary conditions won't be met



Breaking:

① Bounded Resources
→ add more chopsticks

② No preemption
→ take away a chopstick from a philosopher

③ Hold & Wait
→ release the chopstick while waiting for another

④ Circular wait
→ even # philosopher grabs right first then left, odd # does the opposite

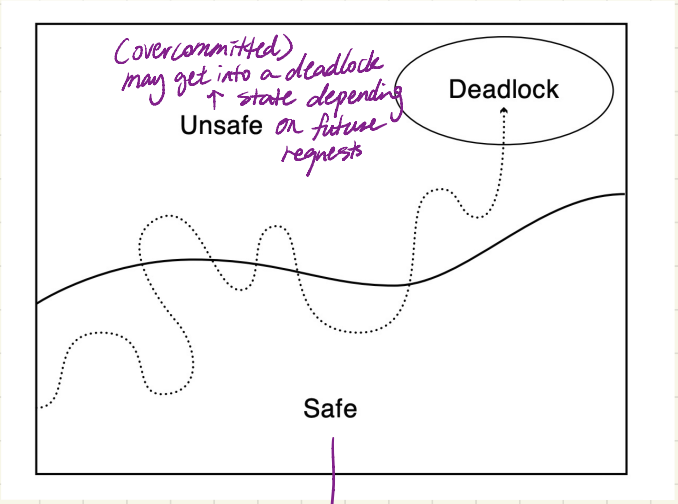
In the kernel context:

→ change global_htable lock to individual file info lock
→ try_lock() to see if you can grab it, release existing locks and try again if some locks aren't free.
→ lock ordering: all threads acquire locks in the same order

Deadlock Avoidance

→ execution/scheduling strategy to avoid getting deadlock

→ requirement = know what resources and maximum resources each thread needs



Banker's Algorithm
(details in the book)

↓

there is a way to serve
all requests safely (no chance of deadlock)

Dining Philosopher (with Deadlock Avoidance)

→ 5 philosophers

→ 1 kitchen facing
5 chopsticks