EECS-343 Operating Systems
Lecture 13:
Synchronization Bugs

Steve Tarzia

Spring 2019

Northwestern

Last Lecture – Concurrent Data Structures

- Simplest strategy is to use ***one big lock***, but this limits concurrency
  - It's ***thread-safe***, but not really concurrent
- Concurrent queue used two locks (head & tail)
- Concurrent hash table used one lock per bucket
- ***Condition Variables*** are used to order threads, using *signal()* & *wait()*.
  - ***Wait*** puts a thread to sleep, ***signal*** wakes a waiting thread.
  - Pthreads allows ***spurious wakeups***, so we still need to check a status variable.
  - ***broadcast()*** wakes all waiting threads
- ***Producer/consumer queue*** was implemented using two condition variables.

Semaphores

- A generalization of condition variables and locks
  - But they're more difficult to understand and use
  - More general is not always better
- Semaphore has an integer value
  - often indicates the number of resources available

Two functions *(with many alternative names!)*:

- up/V/signal/**post**:
  - Increase the value. If there is a waiting thread, wake one.
- down/P/**wait**:
  - Decrease the value. Wait if the value is negative.
- ***Counting semaphore*** is very useful in cases when a finite number of threads are allowed to use a resource (eg., bounded buffer)

Semaphores vs Condition Variables

Semaphores

- ***Up/Post***: increase value and wake one waiting thread
- ***Down/Wait***: decrease value and wait if it's negative

Condition Variables

- ***Signal***: wake one waiting thread
- ***Wait***: wait

- Compared to CVs, Semaphores add an integer value that controls when waiting is necessary
- It counts the quantity of a shared resource currently available
- *Up* makes a resource available, *down* reserves a resource
- Negative value ***-x*** means that ***x*** threads are waiting for the resource

Implementing a lock with a semaphore

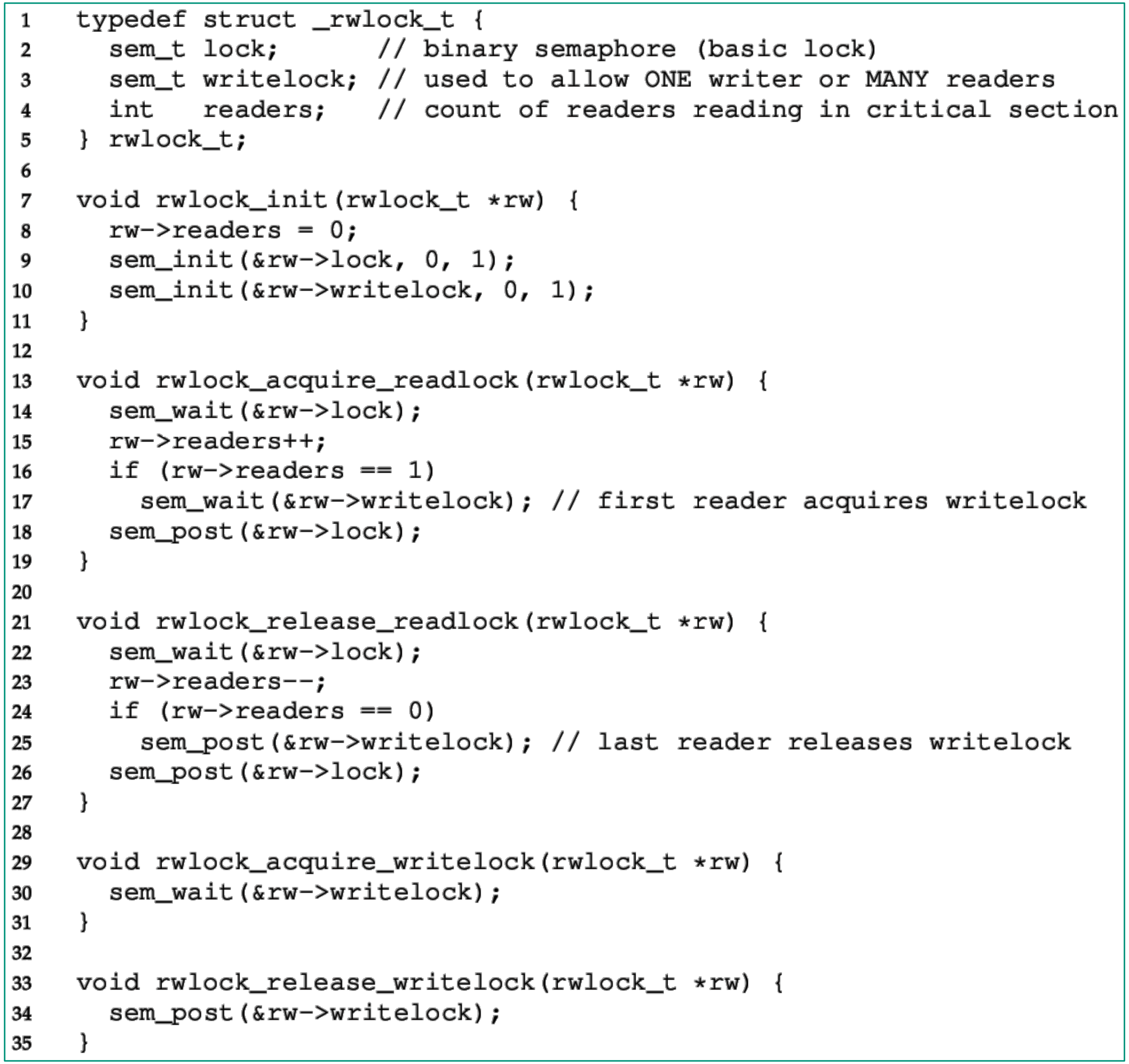
- Choose an appropriate initial value for the semaphore
- To implement a ***Lock***:
  - Initialize to 1 (access to the critical section is the one shared resource)
  - Lock → Down: (decreases the value and waits if negative)
    - Will decrease the value to 0 if it lock *is not* already taken
    - Will decrease the value to -1 and wait if the lock *is* taken (value was 0)
  - Unlock → Up: (increases the value and wakes one waiting thread)
    - If value was 0, then no thread was waiting, and no thread is woken
    - If value was -1, then one thread was waiting, and it is woken
    - If value was -x, then x threads are waiting, one is woken, value becomes x-1.
  - If value is already 1, *Up* should not be called. (Unlock before lock?!)

Reader-writer Lock

- Some resources don't need strict mutual exclusion, especially if they have many ***read-only*** accesses. (eg., a linked list)
- Any number of readers can be active simultaneously, but
- Writes must be mutually exclusive, and cannot happen during read
- API:
  - `acquire_read_lock(), release_read_lock()`
  - `acquire_write_lock(), release_write_lock()`

Reader-writer Lock

- Writelock must be held during read to block writes.
- Number of active readers is counted.
- First/last reader handles acquiring/releasing writelock.



```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // used to allow ONE writer or MANY readers
  int   readers;   // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock); // first reader acquires writelock
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock); // last reader releases writelock
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Common synchronization bugs

- **Atomicity violation**
  - Critical section is violated (due to missing lock).
- **Order violation**
  - Something happens sooner (or later) than we expect.
- **Deadlock**
  - Two threads wait indefinitely for each other.
- **Livelock** *(not common in practice)*
  - Two threads repeatedly block each other from proceeding and retry.

Atomicity violation

- It’s relatively easy to find and protect critical sections,
- But often we forget to add locks around other uses of the shared data.

- Obvious critical section is here:
    - Two threads should not enter this at once
- But, we also have to make sure that *file* is not modified elsewhere.
- Even if this one-line *close* is atomic we have to make sure it doesn’t run during the above critical section.

```
lock(lck);
if (file == NULL) {
   file = open("~/myfile.txt");
}
write(file, "hello file");
unlock(lck);


…


close(file);  // whoops!!
```

Order violation

- Code often requires a certain ordering of operations, especially:
  - Objects must be initialized before they’re used
  - Objects cannot be freed while they are still in use

Parent

```
file = open("file.dat");
thread_create(child_fcn);
// do some work
…
close(file);
```

Child Thread

```
child_fcn() {
  write(file, "hello");
}
```

Close must happen after *write*, but code does not enforce this ordering.

Why is this difficult?

- It seems like we can just add lots of locks and CVs to be safe, right?
  - Wrong! Too many locks can cause ***deadlock*** – indefinite waiting.
- How about just one big lock?
  - (+) Cannot deadlock with one lock.
  - (–) However, this would ***limit concurrency***
    - If every task requires the same lock, then unrelated tasks cannot proceed in parallel.
- Concurrent code is always difficult to write ☹
  - although somewhat easier with some higher-level languages

Intermission

"Which came first, Mom, the Chicken McNugget or the Egg McMuffin?"

Locking granularity

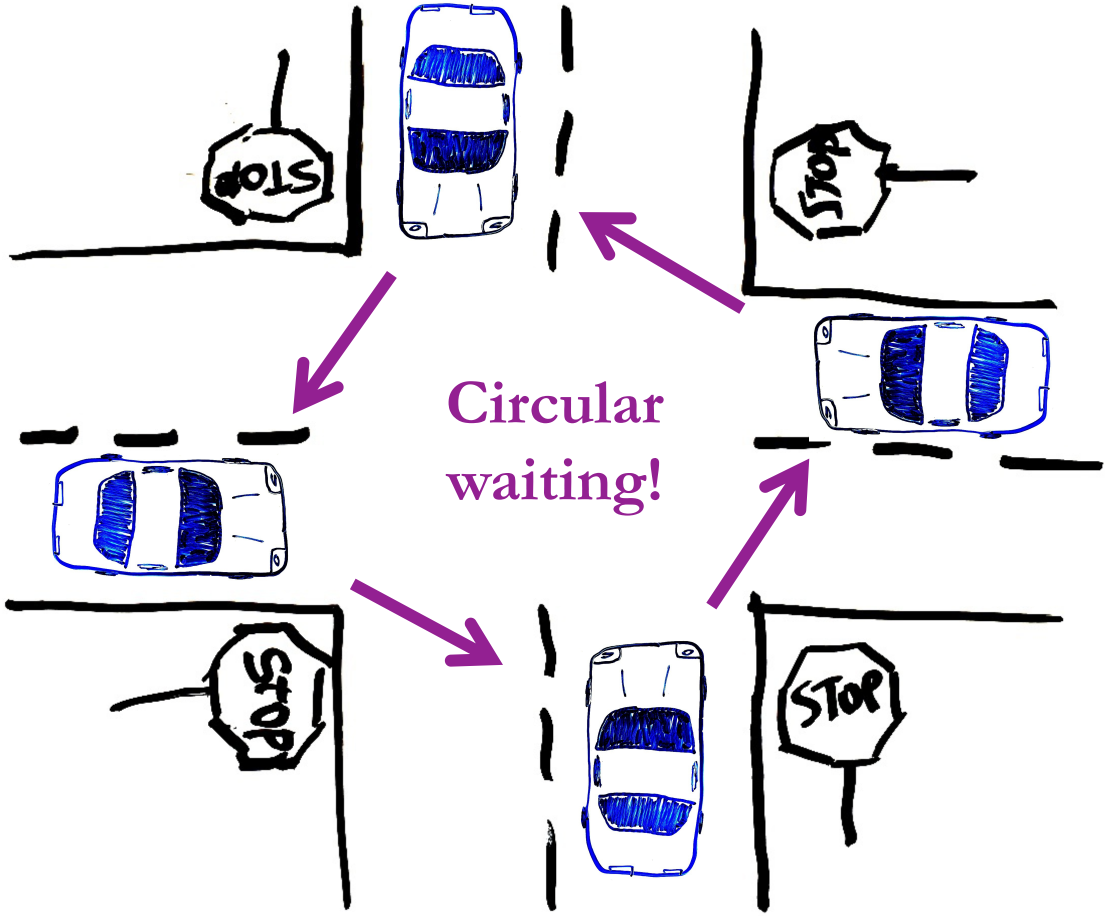
- ***Coarse grained lock*:**
  - Use one (or a few) locks to protect all (or large chunks of) shared state
  - Linux kernel < version 2.6.39 used one "Big Kernel Lock"
  - Essentially only one thread (CPU core) could run kernel code
  - It's simple but there is much contention for this lock, & concurrency is limited
- ***Fine grained locks*:**
  - Use many locks, each protecting small chunks of related shared state
  - Leads to more concurrency and better performance
  - However, there is greater risk of ***deadlock***

Deadlock

- A concurrency bug arising when:
  - Two threads are each waiting for the other to release a resource.
  - While waiting, the threads cannot possibly release the resource already held.
  - So the two threads ***wait forever***.
- Can arise when ***multiple*** shared resources are used.
  - For example, acquiring two or more locks.

Simple example: four-way stop

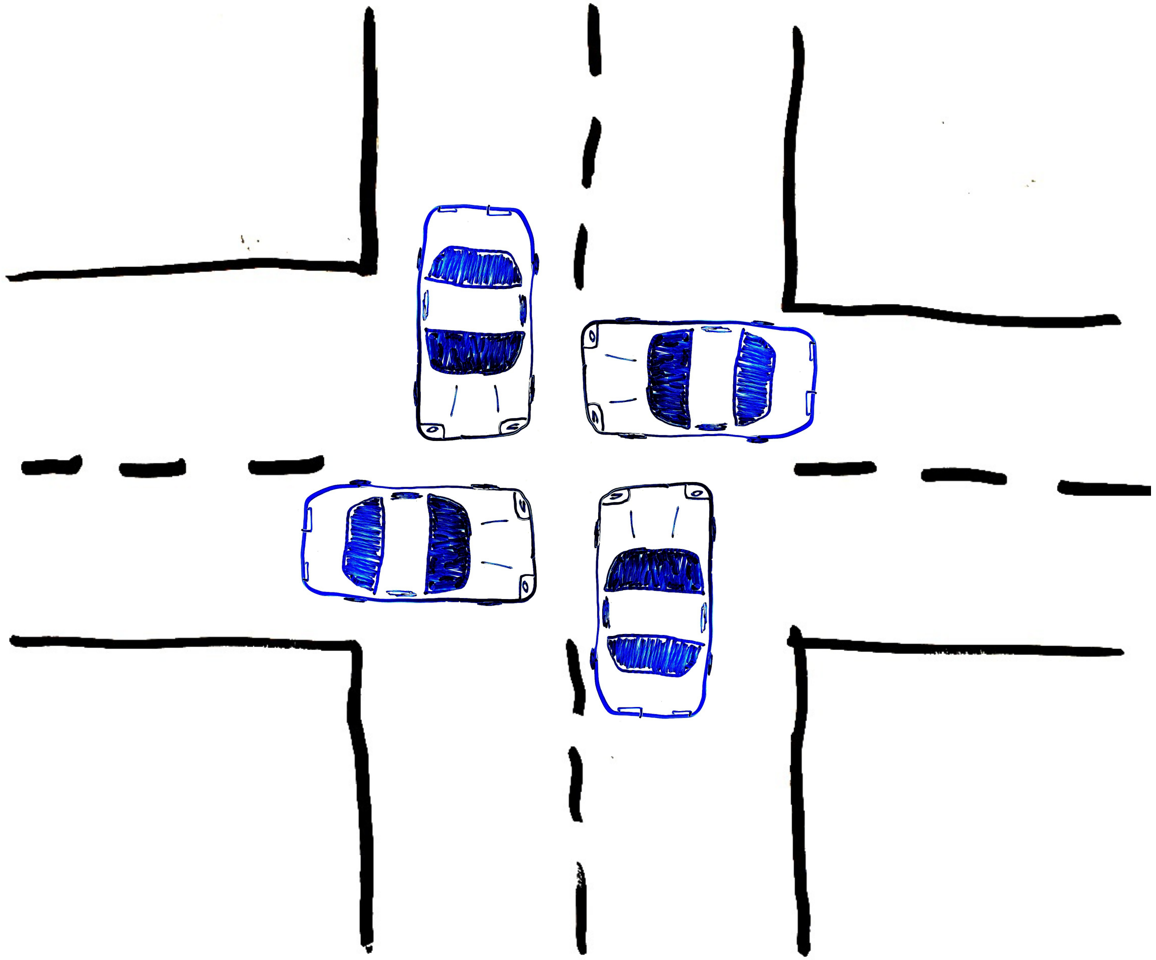
- Traffic rules state that you must **yield to the car on your right** if you reach the intersection simultaneously.
- This rule usually works well.
- But there's a problem if four cars arrive simultaneously.



Circular waiting!

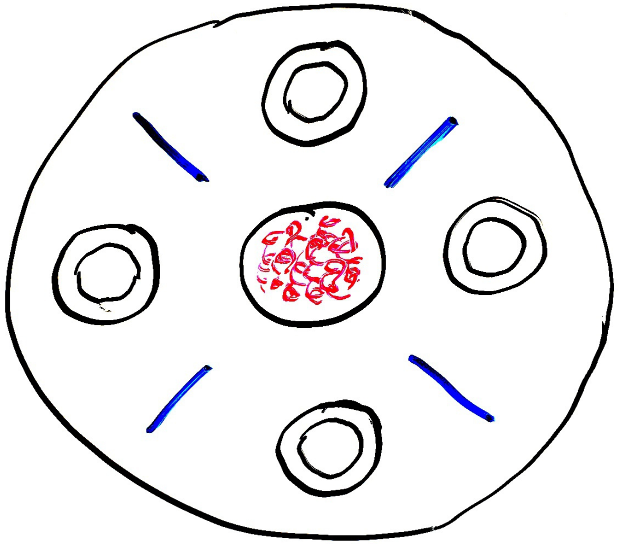
Another 4 way intersection, without stop signs

- There is a problem here if drivers are unwilling to reverse



Dining philosophers

- A theoretical example of deadlock
- There are N philosophers sitting in a circle and N chopsticks
  - left and right of each philosopher
- Philosophers repeatedly run this loop:
  1. Think for some time
  2. Grab chopstick to left
  3. Grab chopstick to right
  4. Eat
  5. Replace chopsticks



- If they all grab the left chopstick simultaneously (step 2), they will deadlock and starve!
- A solution: one philosopher must grab right before left

A more practical deadlock example

- *Thread 1*

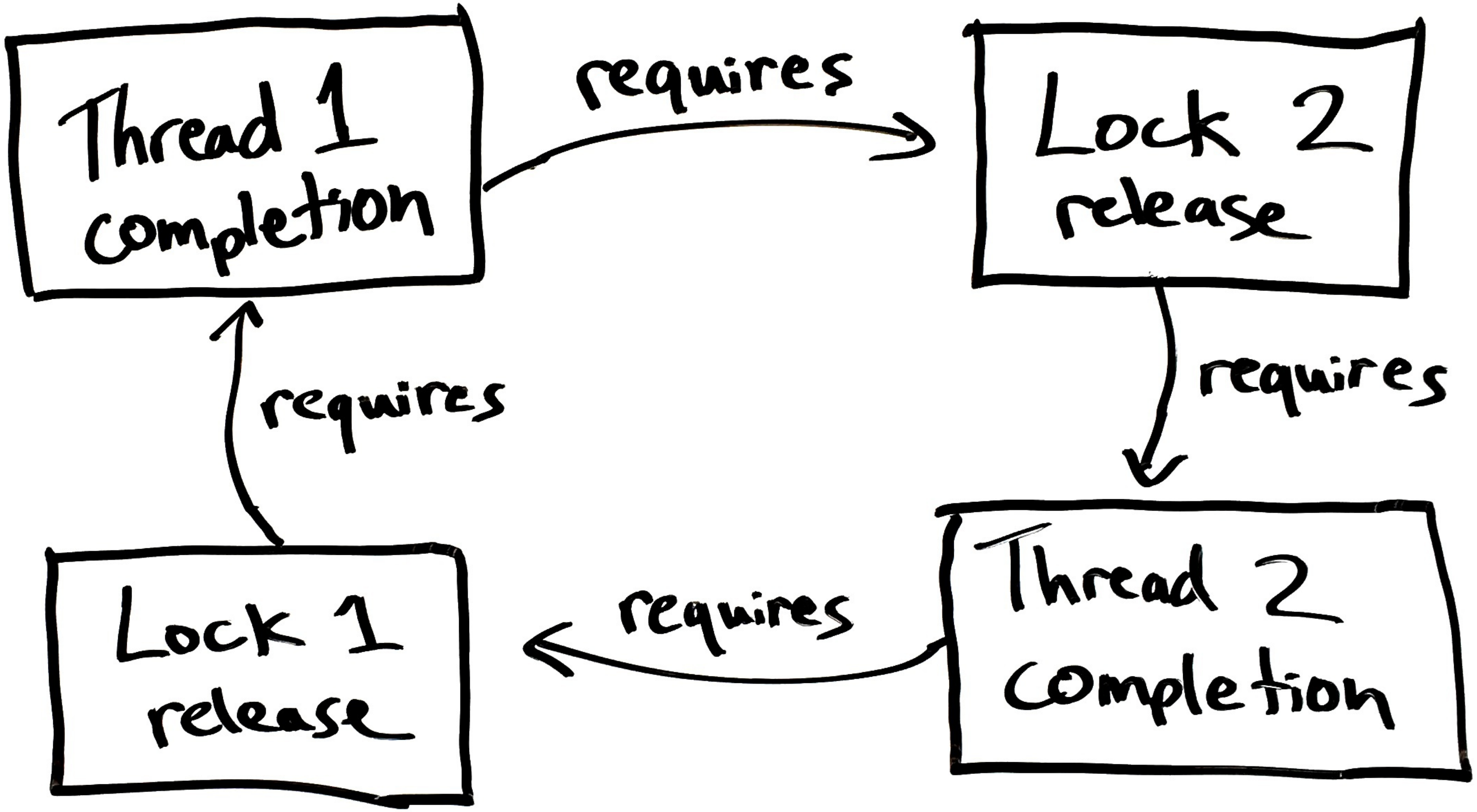
```
lock(L1);
lock(L2);
// do work
…
unlock(L2);
unlock(L1);
```

- *Thread 2*

```
lock(L2);
lock(L1);
// do work
…
unlock(L1);
unlock(L2);
```

- If we are unlucky and both of the first lines execute before the second lines, we will deadlock.
- T1 holds L2 while waiting for L1… T2 holds L1 while waiting for L2

Deadlocks involve *circular dependencies*



Deadlock requires four conditions

1. **Mutual exclusion**
    - Threads cannot access a critical section simultaneously
    - In other words, we're using locks so there is the potential for waiting.
2. **Hold-and-wait**
    - Threads do not release locks while waiting for additional locks
3. **No preemption**
    - Locks are always held until released by the thread. We cannot *cancel* a lock.
4. **Circular wait**
    - Thread is waiting on a thread that is waiting on the original thread
    - This can involve just two threads or a chain of many threads.

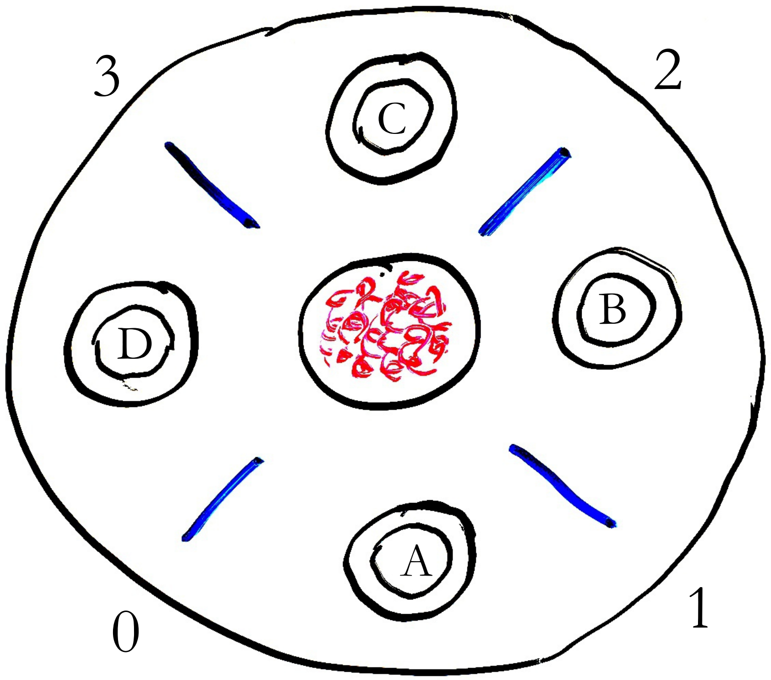
Avoid *any one of these* to avoid deadlock.

4. Avoiding Circular Wait

- This is the most practical way to avoid deadlock.
- The simplest solution is to always acquire locks in the same order.
  - If you hold lock L1 and are waiting for lock L2
  - The holder of L2 cannot be waiting on you,
    because they would have already acquired L1 before acquiring L2.

- However, in practice it can be difficult to know when locks will be acquired because they can be buried in subroutines.

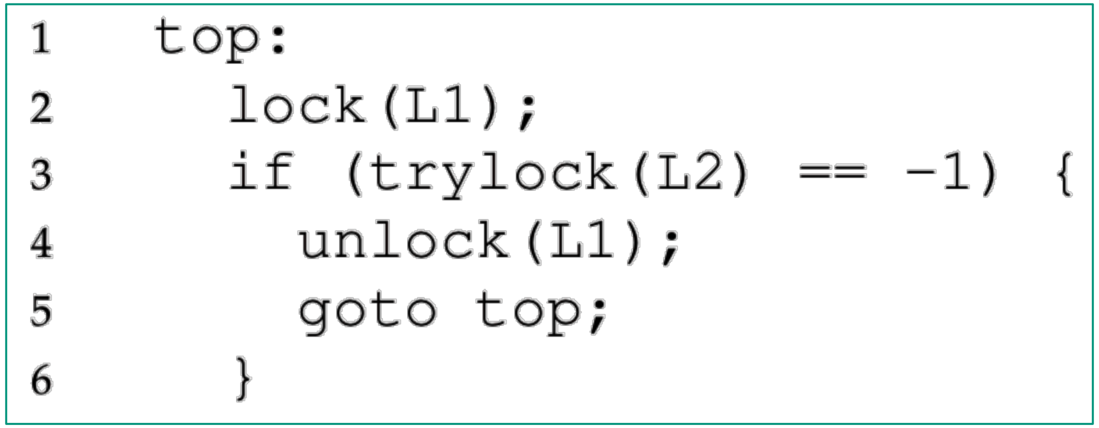
Ordered locking for dining philosophers



- The chopsticks are shared resources, like locks
- If we require the **lower-numbered chopstick to be grabbed first**, this eliminates circular waiting.
- Philosophers A, B, C grab *left then right*.
- However philosopher D will grab *right then left*.
- If everyone tries to start at once, A & D race to grab chopstick 0 first, and the winner eats first.
- While one is waiting to grab its first chopstick a neighbor will be able to grab two chopsticks.

2. Trylock to avoid hold and wait

- We can avoid deadlock if we release the first lock after noticing that the second lock is unavailable.
- ***Trylock*** tries to acquire a lock, but returns a failure code instead of waiting if the lock is taken:

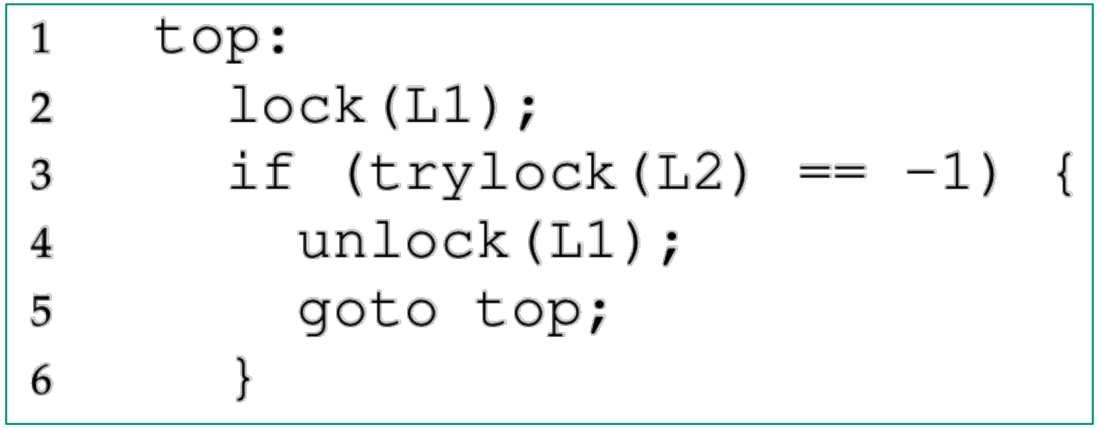


```
top:
    lock(L1);
    if (trylock(L2) == -1) {
        unlock(L1);
        goto top;
    }
```

- This code *cannot deadlock*, even if another thread does the same with L2 first, then L1.
- However it can ***livelock*** – two threads can get stuck in this loop forever

Livelock *vs* Deadlock

- Livelock is a condition where two threads repeatedly take action, but still don’t make progress.



```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

- Differs from deadlock because deadlock is always permanent.
- Livelock involves retries that ***may*** lead to progress, but there is ***no guarantee of progress.***
  - A malicious scheduler can always keep the livelock stuck
- Any randomness in the timing of retries will fix livelock.
- In practice, livelock is a much less serious concern than deadlock.

Other deadlock avoidance strategies

- Wait-free synchronization
    - Instead of using locks, build data structures that directly use atomic primitives like compare-and-swap or load-linked & store-conditional.
    - This is difficult!
- Don't simultaneuously schedule threads that use the same sets of locks.
    - Like the "one big lock" strategy, this reduces concurrency and performance.
- Detect and kill:
    - Periodically check which threads are holding locks and waiting for locks.
    - If there is a circular wait, then kill the process.
      It's not making progress anyway!
    - Yes, the crash can be harmful, but it's inevitable because we're stuck.
    - At least it frees up resources for other processes and makes the user aware of the deadlock bug.

Helgrind tool

- Helgrind (part of the Valgrind tool) detects many common errors when using the POSIX pthreads library in C & C++, such as:
  - Race conditions (missing locks)
  - Lock ordering problems (leading to deadlock)
  - Double-unlocking
  - Freeing a locked lock
  - … and *much*, *much* more
  - http://valgrind.org/docs/manual/hg-manual.html

Recap – Synchronization Bugs

- ***Semaphore*** (up/down) is an all-purpose synchronization primitive
- ***Reader-writer*** lock allows multiple readers, but one writer.
- Adding too many locks can lead to ***deadlock***, which requires:
    - Mutual exclusion (avoid locks to avoid deadlock)
    - Hold and wait (use ***trylock*** to release first lock to before deadlocking)
    - No preemption
    - Circular wait (always acquire locks in the same order to avoid deadlock)
- Dining philosophers was an example of deadlock
    - Circular wait can be avoided by making one philosopher grab right-hand side instead of left first.