CONCURRENCY: SEMAPHORES

Andrea Arpaci-Dusseau

CS 537, Fall 019

Administrivia

- Project 4 due today at 5:00 pm (or 7:00 am…)
- Project 5 available Wed morning (xv6 Memory)
  - Request new project partner if desired
- Midterm 2: Nov 11/6 (Wed) from 7:30-9:30pm
  - Notify by tomorrow if conflict
  - Two "quizzes" on race conditions in Canvas
- Office Hours Changed Today: 4-5 pm

LEARNING OUTCOMES: SEMAPHORES

Semaphores (vs. condition variables?)

How to implement a **lock** with semaphores?

How to implement semaphores with locks and condition variables?

How to implement **join** and **producer/consumer** with semaphores?

How to synchronize **dining philosophers**?

How to implement **reader/writer locks** with semaphores?

RECAP

CONCURRENCY OBJECTIVES

Mutual exclusion (e.g., A and B don't run at same time)

solved with *locks*

Ordering (e.g., B runs after A does something)

solved with *condition variables* and *semaphores*

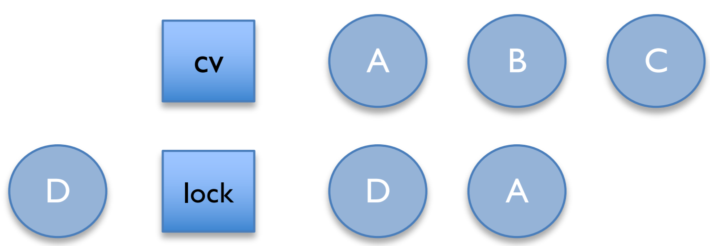
CONDITION VARIABLES

wait(cond_t *cv, mutex_t *lock)

- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing



signal(cv) - what happens?

release(lock) - what happens?

ORDERING EXAMPLE: JOIN

```
pthread_t p1, p2;
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish (call exit())
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done\n [balance: %d]\n [should: %d]\n",
    balance, max*2);
return 0;
```

how to implement join()?

JOIN IMPLEMENTATION: CORRECT

Parent:

```
void thread_join() {
        Mutex_lock(&m);        // w
        if (done == 0)         // x
            Cond_wait(&c, &m); // y
        Mutex_unlock(&m);       // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);      // a
        done = 1;            // b
        Cond_signal(&c);     // c
        Mutex_unlock(&m);    // d
}
```

Parent:	w	x	y				z
Child:				a	b	c	

Use mutex to ensure no race between interacting with state and wait/signal

PRODUCER/CONSUMER: TWO CVS AND WHILE

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill();  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

SUMMARY: RULES OF THUMB FOR CVS

1. Keep state in addition to CV's
2. Always do wait/signal with lock held
3. Whenever thread wakes from waiting, recheck state (while loop)

INTRODUCING SEMAPHORES

Condition variables have no state (other than waiting queue)

- Programmer must track additional state

Semaphores have state: track integer value

- State cannot be directly accessed by user program, but state determines behavior of semaphore operations

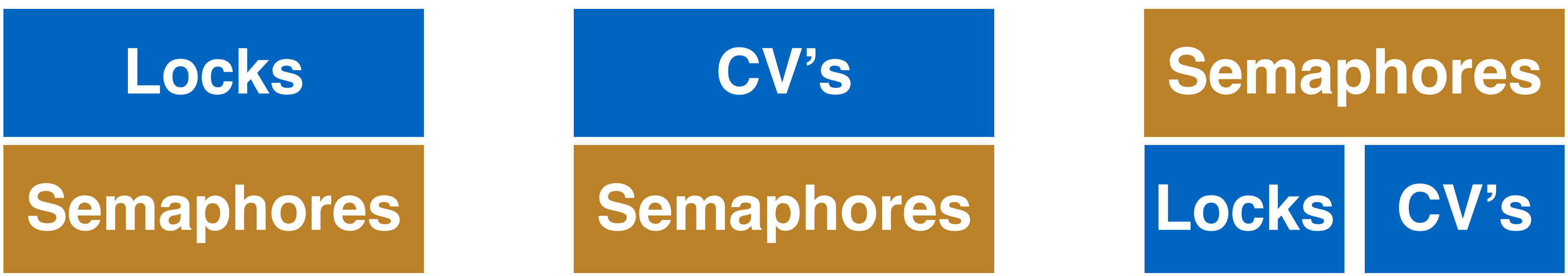
EQUIVALENCE CLAIM

Semaphores are equally powerful to Locks+CVs

- what does this mean?

One might be more convenient, but that's not relevant

Equivalence means each can be built from the other



SEMAPHORE OPERATIONS

Allocate and Initialize

```
sem_t sem;
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

User cannot read or write value directly after initialization

wait and post are atomic

Wait or Test (sometime P() for Dutch) sem_wait(sem_t*)

Waits until value of sem is > 0; Decrements sem value,

Signal or Post (sometime V() for Dutch) sem_post(sem_t*)

Increment sem value, if value > 0, wake a single waiter

BUILD LOCK FROM SEMAPHORE

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {

}

void acquire(lock_t *lock) {

}

void release(lock_t *lock) {

}
```

sem_init(sem_t*, int initial)

sem_wait(sem_t*): Wait until value > 0; dec

sem_post(sem_t*): Increment; if > 0, wake single waiter

Locks

Semaphores

BUILD LOCK FROM SEMAPHORE

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(&lock->sem, 1);
}

void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}

void release(lock_t *lock) {
    sem_post(&lock->sem);
}
```

sem_init(sem_t*, int initial)

sem_wait(sem_t*): Wait until value > 0; dec

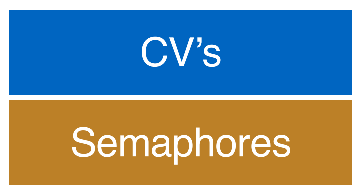
sem_post(sem_t*): Increment; if > 0, wake single waiter

Locks

Semaphores

BUILDING CV'S OVER SEMAPHORES

Possible, but really hard to do right



Read about Microsoft Research's attempts:

http://research.microsoft.com/pubs/64242/ImplementingCVs.pdf

JOIN WITH CV VS SEMAPHORES

```
void thread_join() {
        Mutex_lock(&m);        // w
        if (done == 0)         // x
          Cond_wait(&c, &m);   // y
        Mutex_unlock(&m);      // z
}
```

```
void thread_exit() {
        Mutex_lock(&m);        // a
        done = 1;              // b
        Cond_signal(&c);       // c
        Mutex_unlock(&m);      // d
}
```

```
sem_t s;
sem_init(&s, 
```

sem_wait(sem_t*): Wait until value > 0; dec

sem_post(sem_t*): Increment; if > 0, wake single waiter

```
void thread_join() {
        sem_wait(&s);
}
```

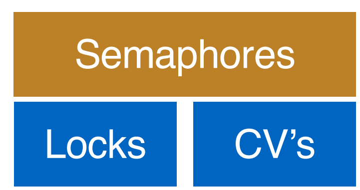
```
void thread_exit() {
        sem_post(&s)
}
```

BUILD SEMAPHORE FROM LOCK AND CV

```
Typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} sem_t;

Void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

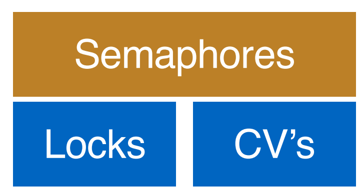


BUILD SEMAPHORE FROM LOCK AND CV

```
Sem_wait{sem_t *s) {
   lock_acquire(&s->lock);
   while (s->value <= 0)
      cond_wait(&s->cond);
   s->value--;
   lock_release(&s->lock);
}
```

```
Sem_post{sem_t *s) {
   lock_acquire(&s->lock);
   s->value++;
   cond_signal(&s->cond);
   lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter



PRODUCER/CONSUMER: SEMAPHORES #1

Single producer thread, single consumer thread
Single shared buffer between producer and consumer

Use 2 semaphores

- emptyBuffer: Initialize to: 1 → 1 empty buffer; producer can run 1 time first
- fullBuffer: Initialize to: 0 → 0 full buffers; consumer can run 0 times first

```
roducer

hile (1) {

    sem_wait(&emptyBuffer);
    Fill(&buffer);

    sem_signal(&fullBuffer);
```

```
Consumer

while (1) {

    sem_wait(&fullBuffer);
    Use(&buffer);

    sem_signal(&emptyBuffer);

}
```

PRODUCER/CONSUMER: SEMAPHORES #2

Single producer thread, single consumer thread

Shared buffer with **N elements** between producer and consumer

Use 2 semaphores

- emptyBuffer: Initialize to: N → N empty buffers; producer can run N times first
- fullBuffer: Initialize to: 0 → 0 full buffers; consumer can run 0 times first

```
Producer
i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer[i]);
    i = (i+1)%N;
    sem_signal(&fullBuffer);
}
```

```
Consumer
j = 0;
While (1) {
    sem_wait(&fullBuffer);
    Use(&buffer[j]);
    j = (j+1)%N;
    sem_signal(&emptyBuffer);
}
```

PRODUCER/CONSUMER: SEMAPHORE #3

Final case:

- Multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element

PRODUCER/CONSUMER: MULTIPLE THREADS

Does this code work correctly?

Producer

```
while (1) {
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_signal(&fullBuffer);
}
```

Consumer

```
while (1) {
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_signal(&emptyBuffer);
}
```

Are `my_i` and `my_j` private or shared? Where is mutual exclusion needed???

PRODUCER/CONSUMER: MULTIPLE THREADS

Consider three possible locations for mutual exclusion

Which work??? Which is best???

```
Producer #1
    sem_wait(&mutex);
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_signal(&fullBuffer);
    sem_signal(&mutex);
```

```
Consumer #1
    sem_wait(&mutex);
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_signal(&emptyBuffer);
    sem_signal(&mutex);
```

PRODUCER/CONSUMER: MULTIPLE THREADS

```
Producer #2
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    Fill(&buffer[myi]);
    sem_signal(&mutex);
    sem_signal(&fullBuffer);
```

```
Consumer #2
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    Use(&buffer[myj]);
    sem_signal(&mutex);
    sem_signal(&emptyBuffer);
```

Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers

PRODUCER/CONSUMER: MULTIPLE THREADS

```
Producer #3
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    sem_signal(&mutex);
    Fill(&buffer[myi]);
    sem_signal(&fullBuffer);
```

```
Consumer #3
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    sem_signal(&mutex);
    Use(&buffer[myj]);
    sem_signal(&emptyBuffer);
```

Works and increases concurrency; only finding a buffer is protected by mutex;
Filling or Using different buffers can proceed concurrently

DINING PHILOSOPHERS

Problem Statement

- **N** Philosophers sitting at a round table
- Each philosopher shares a chopstick (or fork) with neighbor
- Each philosopher must have both chopsticks to eat
- Neighbors can't eat simultaneously
- Philosophers alternate between thinking and eating

Each philosopher/thread **i** runs :

```
while (1) {
    think();
    take_chopsticks(i);
    eat();
    put_chopsticks(i);
  }
```

DINING PHILOSOPHERS: ATTEMPT #1

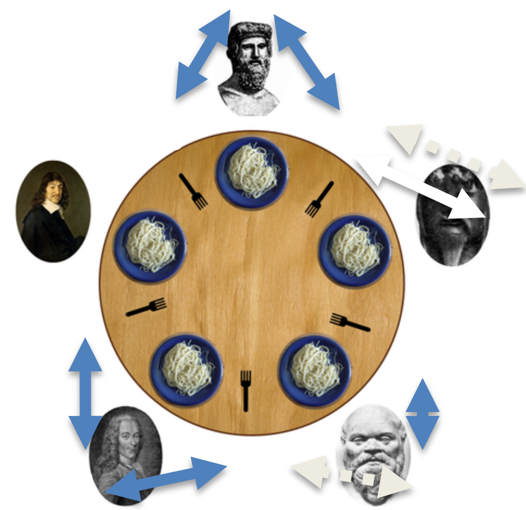
Two neighbors can't use chopstick at same time

Must test if chopstick is there and grab it atomically

- Represent each chopstick with a semaphore
- Grab right chopstick then left chopstick

Code for 5 philosophers:

```
sem_t chopstick[5]; // Initialize each to 1
take_chopsticks(int i) {
    wait(&chopstick[i]);
    wait(&chopstick[(i+1)%5]);
}
put_chopsticks(int i) {
    signal(&chopstick[i]);
    signal(&chopstick[(i+1)%5]);
}
```



DINING PHILOSOPHERS: ATTEMPT #1

Two neighbors can't use chopstick at same time

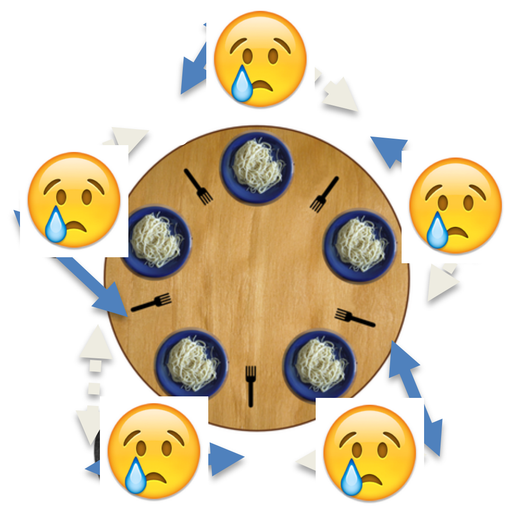
Must test if chopstick is there and grab it atomically

- Represent each chopstick with a semaphore
- Grab right chopstick then left chopstick

Deadlocked!

Code for 5 philosophers:

```
sem_t chopstick[5]; // Initialize each to 1
take_chopsticks(int i) {
   wait(&chopstick[i]);
   wait(&chopstick[(i+1)%5]);
}
put_chopsticks(int i) {
   signal(&chopstick[i]);
   signal(&chopstick[(i+1)%5]);
}
```



DINING PHILOSOPHERS: ATTEMPT #2

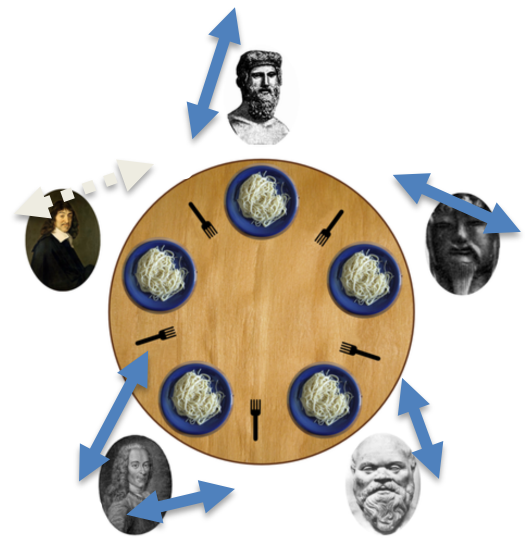
Grab lower-numbered chopstick first, then higher-numbered

```
sem_t chopstick[5]; // Initialize to 1
take_chopsticks(int i) {
  if (i < 4) {
      wait(&chopstick[i]);
      wait(&chopstick[i+1]);
  } else {
      wait(&chopstick[0]);
      wait(&chopstick[4]);
}
```

Philosopher 3 finishes take_chopsticks() and eventually calls put_chopsticks();

Who can run then?

What is wrong with this solution???



DINING PHILOSOPHERS: HOW TO APPROACH

Guarantee two goals

- **Safety:** Ensure nothing bad happens (don't violate constraints of problem)
- **Liveness:** Ensure something good happens when it can (make as much progress as possible)

Introduce state variable for each philosopher i

```
state[i] = THINKING, HUNGRY, or EATING
```

Safety:

No two adjacent philosophers eat simultaneously

```
for all i: !(state[i]==EATING && state[i+1%5]==EATING)
```

Liveness:

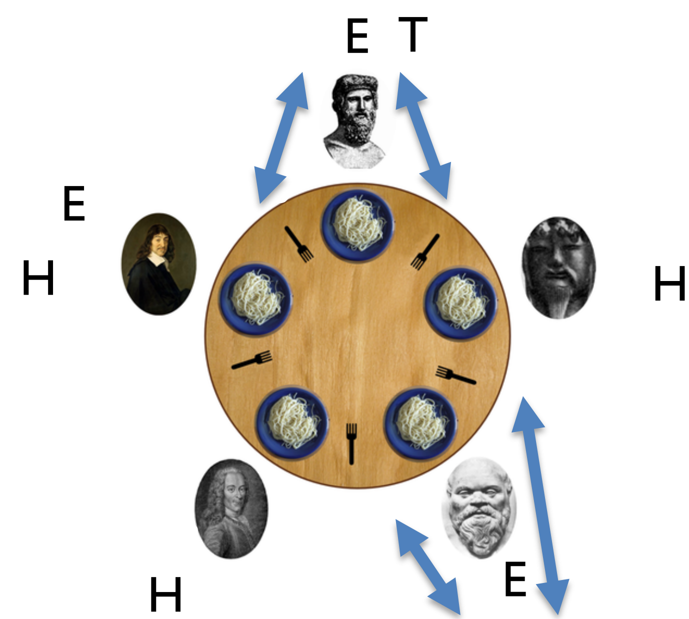
Not the case that a philosopher is hungry and his neighbors are not eating

```
for all i: !(state[i]==HUNGRY &&
    (state[i+4%5]!=EATING && state[i+1%5]!=EATING))
```

```
sem_t mayEat[5]; // how to initialize?
sem_t mutex;     // how to init?
int state[5] = {THINKING};
take_chopsticks(int i) {
  wait(&mutex); // enter critical section
  state[i] = HUNGRY;
  testSafetyAndLiveness(i); // check if I can run
  signal(&mutex); // exit critical section
  wait(&mayEat[i]);
}
put_chopsticks(int i) {
  wait(&mutex); // enter critical section
  state[i] = THINKING;
  test(i+1 %5); // check if neighbor can run now
  test(i+4 %5);
  signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
  if(state[i]==HUNGRY&&state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
      state[i] = EATING;
      signal(&mayEat[i]);
  } }
```

DINING PHILOSOPHERS: EXAMPLE EXECUTION

Take_chopsticks(0)
Take_chopsticks(1)
Take_chopsticks(2)
Take_chopsticks(3)
Take_chopsticks(4)
Put_chopsticks(0)
Put_chopsticks(2)



```
sem_t mayEat[5]; // how to initialize?
sem_t mutex;     // how to init?
int state[5] = {THINKING};
take_chopsticks(int i) {
  wait(&mutex); // enter critical section
  state[i] = HUNGRY;
  testSafetyAndLiveness(i); // check if I can run
  signal(&mutex); // exit critical section
  wait(&mayEat[i]);
}
put_chopsticks(int i) {
  wait(&mutex); // enter critical section
  state[i] = THINKING;
  test(i+1 %5); // check if neighbor can run now
  test(i+4 %5);
  signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
  if(state[i]==HUNGRY&&state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
      state[i] = EATING;
      signal(&mayEat[i]);
  } }
```

Take_chopsticks(0)
Take_chopsticks(1)
Take_chopsticks(2)
Take_chopsticks(3)
Take_chopsticks(4)
Put_chopsticks(0)
Put_chopsticks(2)

READER/WRITER LOCKS

Protect shared data structure; Goal:

Let multiple reader threads grab lock with other readers (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

Two possibilities for priorities – different implementations

1) No reader waits unless writer in critical section
    - How can writers starve?

2) No writer waits longer than absolute minimum
    - How can readers starve?

Let us see if we can understand code…

VERSION 1

Readers have priority

READER/WRITER LOCKS

```
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

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

READER/WRITER LOCKS

```
void rwlock_acquire_readlock(rwlock_t *rw) {
      sem_wait(&rw->lock);
      rw->readers++;
      if (rw->readers == 1)
          sem_wait(&rw->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);
      sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) {  sem_wait(&rw->writelock);  }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
    // who runs?
T4: acquire_readlock()
    // what happens?
T5: acquire_readlock()
    // where blocked?
T3: release_writelock()
    // what happens next?

READER/WRITER LOCKS

```
void rwlock_acquire_readlock(rwlock_t *rw) {
     sem_wait(&rw->lock);
     rw->readers++;
     if (rw->readers == 1)
         sem_wait(&rw->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);
     sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) {  sem_wait(&rw->writelock);  }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T4: acquire_readlock()
// what happens?

VERSION 2

Writers have priority

Three semaphores

- Mutex
- OKToRead (siimilar to myEat[] in Dining Philosphers, but 1 for all readers)
- OKToWrite

How to initialize?

Shared variables

Waiting Readers,

ActiveReaders

WaitingWriters

ActiveWriters

```
Acquire_readlock() {
   Sem_wait(&mutex);
   If (ActiveWriters +
      WaitingWriters==0) {
         sem_post(OKToRead);
         ActiveReaders++;
   } else WaitingReaders++;
   Sem_post(&mutex);
   Sem_wait(OKToRead);
}
Release_readlock() {
   Sem_wait(&mutex);
   ActiveReaders--;
   If (ActiveReaders==0 &&
     WaitingWriters > 0) {
         Sem_post(OKToWrite);
         ActiveWriters++;
         WaitingWriters--;
   }
   Sem_post(&mutex);
}
```

```
Acquire_writelock() {
   Sem_wait(&mutex);
   If (ActiveWriters + ActiveReaders + WaitingWriters==0) {
      ActiveWriters++;
      sem_post(OKToWrite);
   } else WaitingWriters++;
   Sem_post(&mutex);
   Sem_wait(OKToWrite);
}
```

```
Release_writelock() {
   Sem_wait(&mutex);
   ActiveWriters--;
   If (WaitingWriters > 0) {
      ActiveWriters++;
      WaitingWriters--;
      Sem_post(OKToWrite);
   } else while(WaitingReaders>0) {
      ActiveReaders++;
      WaitingReaders--;
      sem_post(OKToRead);
   }
   Sem_post(&mutex);
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T4: acquire_readlock()
// what happens?

SEMAPHORES

Semaphores are equivalent to locks + condition variables

- Can be used for both mutual exclusion and ordering

Semaphores contain **state**

- How they are initialized depends on how they will be used
- Init to 0: Join (1 thread must arrive first, then other)
- Init to N: Number of available resources

Sem_wait(): Waits until value > 0, then decrement (atomic)

Sem_post(): Increment value, then wake a single waiter (atomic)

Can use semaphores in producer/consumer and for reader/writer locks

REVIEW: PROCESSES VS THREADS

```
int a = 0;
int main() {
  fork();
  a++;
  fork();
  a++;
  if (fork() == 0) {
    printf("Hello!\n");
  } else {
    printf("Goodbye!\n");
  }
  a++;
  printf("a is %d\n", a);
}
```

How many times will "Hello!\n" be displayed?

4

What will be the **final** value of "a" as displayed by the final line of the program?

3