welcome back!

CONCURRENCY: SEMAPHORES

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Midterm: Solutions, Grades → Canvas → ~40 / 60

Mid-semester grades soon

P4 progress

Midterm 2

AGENDA / LEARNING OUTCOMES

Concurrency abstractions

- How can semaphores help with producer-consumer?
- How to implement 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 (with state)*

CONDITION VARIABLES

associated

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)

which thread, when thread woken up will run

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

JOIN IMPLEMENTATION

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

Rule of Thumb: Keep state in addition to CV's!

PRODUCER/CONSUMER: TWO CVS AND WHILE

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i);  // 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);
    }
}
```

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

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

initialize → set the state

SEMAPHORE OPERATIONS

Allocate and Initialize

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

← user can set the state at initialization !

User cannot read or write value directly after initialization

SEMAPHORE OPERATIONS

Wait or Test: sem_wait(sem_t*)

Decrements sem value by 1, Waits if value of sem is negative (< 0)

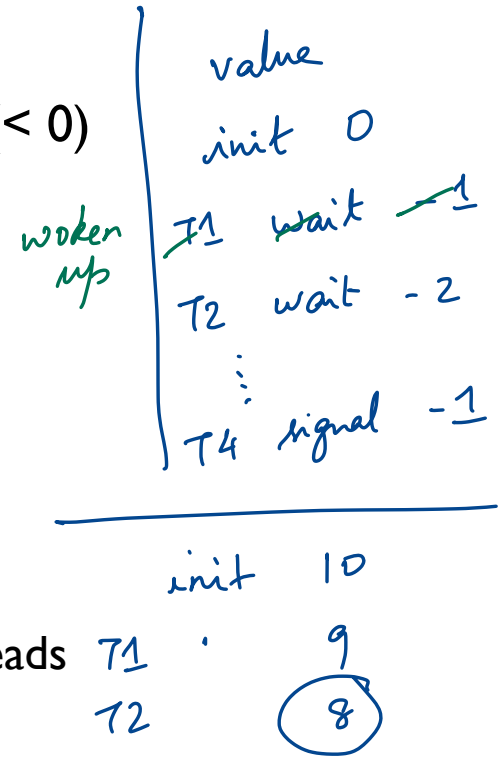
```
{
    s->value--;
    if (s->value < 0)  wait();
}
```

Signal or Post: sem_post(sem_t*)

Increment sem value by 1, then wake a single waiter if exists

```
{
    s->value++;
    wake up waiting thread
}
```

Value of the semaphore, when negative = the number of waiting threads



BINARY SEMAPHORE (LOCK)

```
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*): Decrement, wait if value < 0
sem_post(sem_t*): Increment value
then wake a single waiter

exactly 1 thread should be able to acquire

		value	
T1	acquire()	0	✓
T2	acquire()	-1	wait
⋮			
T1	release()	0	wake up T2

JOIN WITH CV VS SEMAPHORES

Parent → block
0
Child → wakes up parent

```
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
}
```

Child post → Parent
0
1
0 → not wait

```
sem_t s;
sem_init(&s, 0);
```

Parent wait

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

sem_wait(): Decrement, wait if value < 0
sem_post(): Increment value, then wake a single waiter

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

PRODUCER/CONSUMER: SEMAPHORES #1

Single producer thread, single consumer thread

Single shared buffer between producer and consumer

Use 2 semaphores

- emptyBuffer: Initialize to 1
- fullBuffer: Initialize to 0

Prod first → not block on empty Buffer

Cons first → block on full buffer

```
Producer

while (1) {

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

    sem_post(&fullBuffer);

}
```

```
Consumer

while (1) {

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

    sem_post(&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
- fullBuffer: Initialize to 0

still want consumer to block if it runs first

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

Producer can go N times

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

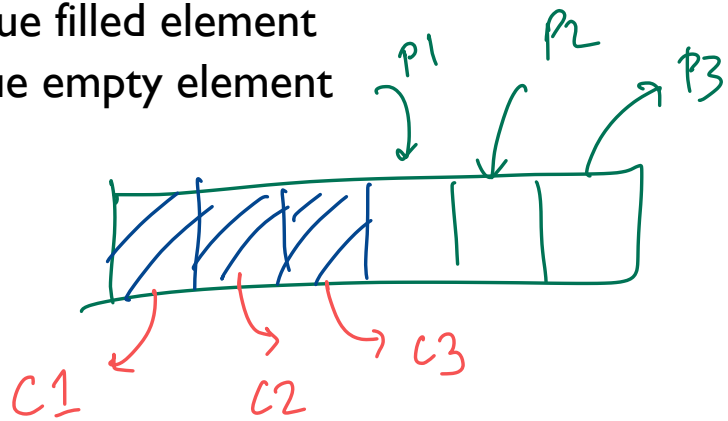
PRODUCER/CONSUMER: SEMAPHORE #3

Final case:

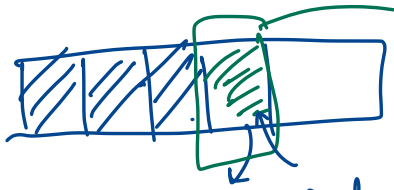
- Multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer

Requirements

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



PRODUCER/CONSUMER: MULTIPLE THREADS



each thread gets its own spot

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

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

don't want two threads to get same my_i

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

↳ locks!

PRODUCER/CONSUMER: MULTIPLE THREADS

binary locks

Consider three possible locations for mutual exclusion

Which work??? Which is best???

blocked

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

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

grabs mutex

sleep while holding the mutex

empty

→ Consumer goes first: Producer comes in

dead lock

PRODUCER/CONSUMER: MULTIPLE THREADS

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

```
Consumer #2
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    Use(&buffer[myj]);
    sem_post(&mutex);
    sem_post(&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_post(&mutex);
    Fill(&buffer[myi]);
    sem_post(&fullBuffer);
```

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

might be expensive outside

only search empty slot while holding mutex

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

QUIZ 12

```
int done = 0;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    done = 1;                    //e1
    Pthread_cond_signal(&c);     //e2
}

void thr_join() {
    if (done == 0)               //j1
        Pthread_cond_wait(&c);   //j2
}
```

Execution order

j1, e1, e2, j2

check done == 0

child thread

cond_wait

parent sleeps forever

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if(numfull == max) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}
```

10

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);  // c1
        if(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```

Producer runs for one iteration followed by the consumer?

Assume you start from i = 0, numfull = 0, max = 5

p1 p2 p4 p5 p6 c1 c2 c4 c5 c6 c7

If the consumer runs first?

c1 c2 c3 p1 p2 p4 p5 p6 c4 c5 c6 c7

The variable 'numfull' cannot be greater than the variable 'loops'. True

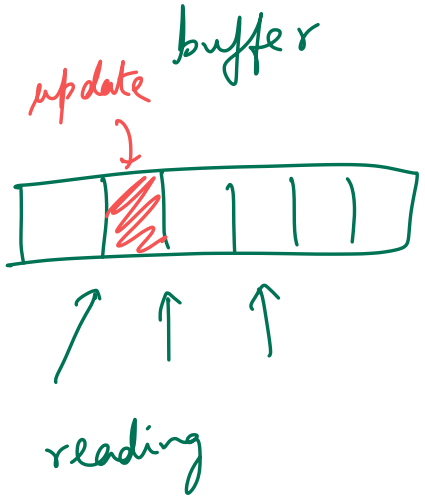
READER/WRITER LOCKS

Let multiple reader threads grab lock (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

Let us see if we can understand code…



already 1 reader → its ok to admit any other reader thread

READER/WRITER LOCKS

```
1 typedef struct _rwlock_t {
2     sem_t lock;
3     sem_t writelock;
4     int readers;
5 } rwlock_t;
6
7 void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 1);
10    sem_init(&rw->writelock, 1);
11 }
```

number of readers

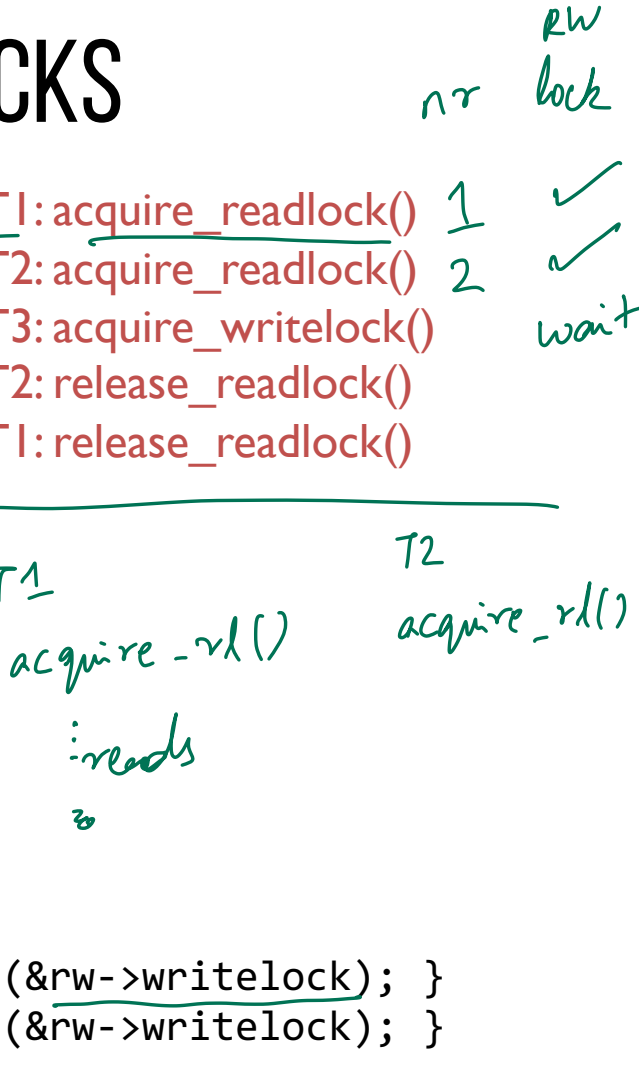
no readers

similar to binary lock

READER/WRITER LOCKS

```
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock);
18     sem_post(&rw->lock);
19 }
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock);
26     sem_post(&rw->lock);
27 }
29 rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
31 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()



READER/WRITER LOCKS

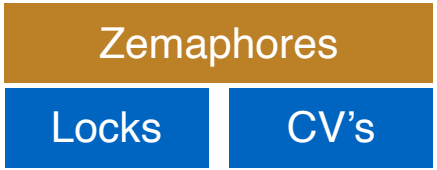
```
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14       sem_wait(&rw->lock);
15       rw->readers++;
16       if (rw->readers == 1)
17           sem_wait(&rw->writelock);
18       sem_post(&rw->lock);
19 }
21 void rwlock_release_readlock(rwlock_t *rw) {
22       sem_wait(&rw->lock);
23       rw->readers--;
24       if (rw->readers == 0)
25           sem_post(&rw->writelock);
26       sem_post(&rw->lock);
27 }
29 rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
31 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()
T4: acquire_readlock()
T5: acquire_readlock()
T3: release_writelock()
// what happens next?

BUILD ZEMAPHORE!

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

void zem_init(zem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```



zem_wait(): Waits while value <= 0, Decrement

zem_post(): Increment value, then wake a single waiter

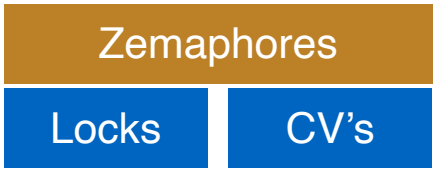
BUILD ZEMAPHORE FROM LOCKS AND CV

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

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

zem_wait(): Waits while value <= 0, Decrement

zem_post(): Increment value, then wake a single waiter



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(): Decrement and then wait if < 0 (atomic)

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

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

NEXT STEPS

Concurrency Bugs!