Welcome back!

CONCURRENCY: SEMAPHORES

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

Piazza / Canvas

Midterm: Solutions, Grades → Tomm.

Project 2 grades – Late, Spurious. Don't Panic!

– TODAY!

Next: Spring break!

→ P3, P4 grades

AGENDA / LEARNING OUTCOMES

Concurrency abstractions

- How can semaphores help with producer-consumer?
- How to implement semaphores?

Abstractions

how do you use them?

how do you implement them?

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

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

calling thread is put to sleep

signal(cond_t *cv)

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

JOIN IMPLEMENTATION

state variable

hold the lock for wait / signal

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:			waiting	a	b	c	d signal

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

Producer/Consumer with 1 buffer

num full

state

buffer

Thread 1 state:

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

Thread 2 state:

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

WHAT ABOUT 2 CONSUMERS?

Can you find a problematic timeline with 2 consumers (still 1 producer)?

1 producer, 1 shared buffer, 2 consumers

Starvation

$\boxed{\cancel{42}}$ buffer

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

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);   // c1
        if(numfull == 0) // c2
  C1 →      Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4 →
  C2 →  Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7 read empty buffer
    }
}
```

Prod. , C1

			wait()			wait()				signal()				wait()			signal()
Producer:							p1	p2	p4	p5	p6	p1	p2	p3			
Consumer1:	c1	c2	c3														
Consumer2:				c1	c2	c3				C2 woken up					c2	c4	c5

HOW TO WAKE THE RIGHT THREAD?

Use two condition variables!

PRODUCER/CONSUMER: TWO CVS

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

Producer will wait on "empty"

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

Consumer will wait on fill

PRODUCER/CONSUMER: TWO CVS

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

Producer

```
void *consumer(void *arg) {
    while (1) {
1       Mutex_lock(&m);
2       if (numfull == 0)
3  C1       Cond_wait(&fill, &m);
4       int tmp = do_get();
5       Cond_signal(&empty);
6       Mutex_unlock(&m);
    }
}
```

C1: 1, 2, 3

P: P1, P2, P4, P5

C2:

1,2,4..

time

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.

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);
    }
}
```

No concurrent access to shared state

Every time lock is acquired, assumptions are reevaluated

Using while instead of if

- A consumer will get to run after every do_fill()
- A producer will get to run after every do_get()

RULE OF THUMB

Whenever a lock is acquired, recheck assumptions about state!

Another thread could grab lock in between signal and wakeup from wait

Note that some libraries also have "spurious wakeups" (may wake multiple waiting threads at signal or at any time)

↳ signal will wake up only one thread

↳ signal will wake up one or more threads

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

QUIZ 17

https://tinyurl.com/cs537-sp23-quiz17

What is the sequence of execution when the producer runs for one iteration followed by the consumer?

P1 P2 P4 P5 P6 c1 c2 c4 C5 C6 C7

skip wait ... Skip wait

What is the sequence of execution if the consumer runs first?

→ sleep, releases lock

C1 C2 C3 P1 P2 P4 P5 P6 C4 C5 C6 C7

wait ... signal

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

False. numfull → buffer size (1 in the case)

loops can be bigger

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

SEMAPHORE OPERATIONS

Allocate and Initialize

0, 1 or any value

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

integer

User cannot read or write value directly after initialization

SEMAPHORE OPERATIONS

init (sem_t *s, int v)

s→val = v

Wait or Test: sem_wait(sem_t*)

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

initialize value = 0

Signal or Post: sem_post(sem_t*)

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

value

T1 -1, T1 waits

T2 -2, T2 waits

⋮

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

0

1

-1, T1 wait

0, T1 gets lock

-1, T2 wait

-2, T3 wait

T1 releases lock → -1, wake up T2

JOIN WITH CV VS SEMAPHORES

Parent first
-1, Parent waits

Child first
1, parent dec to 0

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
}
```

```
sem_t s;
sem_init(&s, _0_); 0
```

2   1

sem_wait(): Decrement, wait if value < 0

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

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

-1 waits

1 → not wait

0 → not wait

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

→ incr 0 & wake up Parent

PRODUCER/CONSUMER: SEMAPHORES #1

EB : 1
FB : 0

Prod first
→ Decrement 0
Fill item

Cons first
→ Wait after decrementing (F.B = -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

Producer

```
while (1) {
    sem_wait(&emptyBuffer);
  → Fill(&buffer);

  → sem_post(&fullBuffer);
}
```

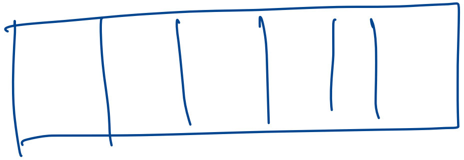
Consumer

```
while (1) {
  → sem_wait(&fullBuffer);
    Use(&buffer);

    sem_post(&emptyBuffer);
}
```

PRODUCER/CONSUMER: SEMAPHORES #2

buffer = N items



- 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

Prod can run N items

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

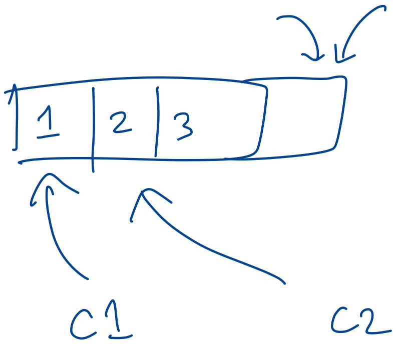
```
Consumer
j = 0;
While (1) {
   sem_wait(&fullBuffer);
   Use(&buffer[j]);
   j = (j+1)%N;
   sem_post(&emptyBuffer);
}
```

PRODUCER/CONSUMER: SEMAPHORE #3

Final case:

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

most general



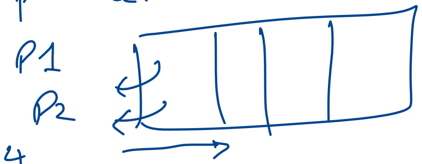
Requirements

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

C2 must not get item 1 as C1 has grabbed it

PRODUCER/CONSUMER: MULTIPLE THREADS

Two producer threads could grab the same empty slot



Producer

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

4

Consumer

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

lock

```
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);      → acquire
    sem_wait(&fullBuffer); → wait
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_post(&emptyBuffer);
    sem_post(&mutex);
```

Deadlock → where consumer grabs mutex but is blocked on full buffer

PRODUCER/CONSUMER: MULTIPLE THREADS

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

Safe

```
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]);   → expensive
    sem_post(&fullBuffer);
```

need to mark so another

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

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

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…

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()
T4: acquire_readlock()
T5: acquire_readlock()
T3: release_writelock()
// what happens next?

QUIZ 18

https://tinyurl.com/cs537-sp23-quiz18

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()

T4: acquire_writelock()
T5: acquire_writelock()
T6: acquire_readlock()

T8: acquire_writelock()
T7: acquire_readlock()
T9: acquire_readlock()

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

Spring break!