CONCURRENCY: SEMAPHORES

Shivaram Venkataraman
CS 537, Spring 2020
ADMINISTRIVIA

- Project 3 is due Today 10pm!

- Midterm is next Thursday 3/12, details on Piazza
- Discussion today: Practice, review for the midterm

- AEFIS midsemester feedback
Concurrency abstractions
  How can semaphores help with producer-consumer?
  How to implement semaphores?
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)*
**SUMMARY: 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 $\geq 1$ thread is waiting)
- if there is no waiting thread, just return, doing nothing
PRODUCER/CONSUMER: TWO CVS AND WHILE

```c
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) // p2
            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
A consumer will get to run after every do_fill()
A producer will get to run after every do_get()
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

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

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

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

https://tinyurl.com/cs537-sp20-quiz17
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
Allocate and Initialize

```c
sem_t sem;
sem_init(sem_t *s, int initval) {
    s->value = initval;
}
User cannot read or write value directly after initialization
```
SEMAPHORE OPERATIONS

**Wait or Test:** `sem_wait(sem_t*)`
Decrement sem value by 1, Waits if value of sem is negative (< 0)

**Signal or Post:** `sem_post(sem_t*)`
Increment sem value by 1, then wake a single waiter if exists

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) {
}

void acquire(lock_t *lock) {
}

void release(lock_t *lock) {

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
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, ___-);

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

void thread_exit() {
    sem_post(&s)
}
Single producer thread, single consumer thread
Single shared buffer between producer and consumer

Use 2 semaphores
- emptyBuffer: Initialize to ________
- fullBuffer: Initialize to __________

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 ____________
  – fullBuffer: Initialize to ____________

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);
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

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);
}

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

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

Consumer #1

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

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

Consumer #3

```c
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
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…
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**

```c
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()
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()
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 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

Project 3: Due Today!

Midterm details posted on Piazza
Discussion today: Practice for midterm

Next class: Deadlocks