CONCURRENCY: SEMAPHORES

Shivaram Venkataraman
CS 537, Spring 2020
- Project 3 is due Today 10pm!

- Midterm is next Thursday 3/12, details on Piazza

- Discussion today: Practice, review for the midterm
  TA review Friday & Monday

- AEFIS midsemester feedback → closes on Friday
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)
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 >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing
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
  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 CV'S

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

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

\[
P_1 P_2 P_4 P_5 P_6 C_1 C_2 C_4 C_5 C_6 C_7
\]

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

\[
C_1 C_2 C_3 P_1 P_2 P_4 P_5 P_6 C_4 C_5 C_6 C_7
\]

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

False
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

Initialized
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*)`
Decrements 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) {
    sem_init(&sem, 1)
}

void acquire(lock_t *lock) {
    sem_wait()
}

void release(lock_t *lock) {
    sem_post()
}

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

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Lock 1</th>
<th>Lock 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Lock</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>T1</td>
<td>Sleep</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>Lock</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>Sleep</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>Unlock</td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-2</td>
<td></td>
</tr>
</tbody>
</table>
```
JOIN WITH CV VS SEMAPHORES

Parent

blocks

Child

block

exit

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, 0);

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

void thread_exit() {
    sem_post(&s)
}

sem_wait(): Decrement, wait if value < 0
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
  - 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

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**

Producer
\[ \text{i} = 0; \]
\[ \text{while (1) {} } \]
\[ \text{sem\_wait}(&\text{emptyBuffer}); \]
\[ \text{Fill}(&\text{buffer}[\text{i}]); \]
\[ \text{i} = (\text{i}+1)\%\text{N}; \]
\[ \text{sem\_post}(&\text{fullBuffer}); \]
\[ } \]

Consumer
\[ \text{j} = 0; \]
\[ \text{While (1) {} } \]
\[ \text{sem\_wait}(&\text{fullBuffer}); \]
\[ \text{Use}(&\text{buffer}[\text{j}]); \]
\[ \text{j} = (\text{j}+1)\%\text{N}; \]
\[ \text{sem\_post}(&\text{emptyBuffer}); \]
\[ } \]
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/CONSUMER: MULTIPLE THREADS

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

```
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);
```

Dead lock
PRODUCER/CONSUMER: MULTIPLE THREADS

Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers
PRODUCER/CONSUMER: MULTIPLE THREADS

Works and increases concurrency; only finding a buffer is protected by mutex; Filling or Using different buffers can proceed concurrently
READER/WRITE 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…
typedef struct _rwlock_t {
  sem_t lock;  /* Two semaphores */
  sem_t writelock;
  int readers;  /* Num of readers */
} rwlock_t;

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

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

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

QUIZ 18

https://tinyurl.com/cs537-sp20-quiz18

T1: acquire_readlock()  →  T1 gets read lock  RUN
T2: acquire_readlock()  →  T2 gets read lock  RUN
T3: acquire_writelock() →  T3 blocked
T4: acquire_writelock()  →  T4 gets write lock  RUN
T5: acquire_writelock()  →  T5 blocked  →  waiting for write lock
T6: acquire_readlock()  →  T6 blocked

T7: acquire_readlock()  →  T7 wait for read lock
T8: acquire_writelock()  →  T8 gets write lock  RUN
T9: acquire_readlock()  →  T9 wait for read lock
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 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
Project 3: Due Today!

Midterm details posted on Piazza
Discussion today: Practice for midterm

Next class: Deadlocks