Concurrency: Semaphores

CS 537: Introduction to Operating Systems

Louis Oliphant & Tej Chajed

University of Wisconsin - Madison

Spring 2024

Administrivia

- Project 4 due Mar 12th @ 11:59pm

Review: Condition Variables

Condition variables are used to control the order that threads execute.

Can be used for thread to wait for another thread to finish

Can be used for the producer / consumer problem

Quiz: Condition Variables

https://tinyurl.com/cs537-sp24-q12

Semaphores Agenda

- Definition of Semaphore
- Using Semaphores instead of locks and condition variables
- Producer/Consumer Problem
- Reader-Writer locks
- Dining Philosophers
- Building Zemaphores

Semaphores

A **semaphore** is an object with an integer value that can be manipulated with two routines:

- sem_wait()

```
int sem_wait(sem_t *s) {
    // decrement the value of semaphore s by one
    // wait if value of semaphore s is negative
}
```

- sem_post()

```
int sem_post(sem_t *s) {
    // increment the value of semaphore s by one
    // if there are threads waiting, wake one
}
```

Semaphores (things to note)

- A semaphore must first be initialized before using `sem_wait()` or `sem_post()`

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); //initializes to 1 (3rd arg)
```

- Notice `sem_wait()` may block or may return immediately
- If the semaphore is negative, it is equal to the number of waiting threads

Binary Semaphores (Locks)

What should `X` be to make the semaphore a lock?

```
sem_t m;
sem_init(&m, 0, X);

sem_wait(&m);   //equivalent to acquire_lock()
///critical section
sem_post(&m);   //equivalent to release_lock()
```

Semaphore as Lock

- X should be set to 1
  - One thread gets through call to `sem_wait()` and enters critical section, other threads will block
  - After critical section, on call to `sem_post()`, one blocked thread will be awoken, leave `sem_wait()`, and enter critical section
- What are the range of values that the semaphore could be?
- Since a lock only has two states, a semaphore used as a lock is called a **binary semaphore**

Semaphores for Ordering

Parent Waiting For Child Using Locks

```
void thr_join() {
   Pthread_mutex_lock(&m);
   while (done == 0)
      Pthread_cond_wait(&c,&m);
   Pthread_mutex_unlock(&m);
}
```

```
void thr_exit() {
   Pthread_mutex_lock(&m);
   done = 1;
   Pthread_cond_signal(&c);
   Pthread_mutex_unlock(&m);
}
```

Using Semaphores

What should X be?

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

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

```
void thr_exit() {
  sem_post(&s);
}
```

Producer/Consumer (Using Locks and CV)

```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Failed Producer/Consumer (Using Semaphores)

```c
sem_t empty, full, mutex;

void *producer(void *arg) {
    int i;
    for(i=0;i<loops;i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Deadlock may occur.

What sequence of steps would cause deadlock?

```c
void *consumer(void *arg) {
    int i=0;
    for(i=0;i<loops;i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n",tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);
    // ...
}
```

Correct Producer/Consumer (Using Semaphores)

```
sem_t empty, full, mutex;

void *producer(void *arg) {
    int i;
    for(i=0;i<loops;i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}
```

reduce the scope of the mutex.

```
void *consumer(void *arg) {
    int i=0;
    for(i=0;i<loops;i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        int tmp = get();
        sem_post(&mutex);
        sem_post(&empty);
        printf("%d\n",tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);
    // ...
}
```

Reader-Writer Locks

New type of lock:

- allows one writer thread to hold lock at a time
- OR allows many reader threads to hold lock at a time (and no writers)
- when acquiring lock, acquire either read-lock or write-lock

Implementation (next slide):

- Keeps track of number of readers:
  - First reader acquiring also acquires write-lock
  - Last reader releasing also releases write-lock
- With lots of readers, write threads may starve
  - How could this be fixed?
  - Complicated locks add more overhead

Read-Writer Lock Implementation

```
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, 0, 1);
   sem_init(&rw->writelock, 0, 1);
}

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);
}
```

Read-Writer Lock Implementation (cont.)

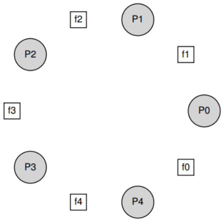
```
void rwlock_release_readlock(rwlock_t *rw) {
   sem_wait(&rw->lock);
   rw->readers--;
   if (rw->readers == 0)
      sem_post(&rw->writelock);
   sem_post(&rw->lock);
}

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

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

The Dining Philosophers Problem

- Five "philosophers" sitting around a table thinking and eating
- To eat philosopher requires fork on their left and right
- Write `getforks()` and `putforks()` so that:
  - no deadlock
  - no philosopher starves
  - concurrency is high




```
while(1) {
  think();
  getforks();
  eat();
  putforks();
}
```

Dining Philosophers

Helper Functions

```
int left(int p) { return p; }
int right(int p) { return (p+1) % 5; }
```

Broken Solution

```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}

void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

Working Solution – Why?

```
void getforks() {
   if (p==4) {
      sem_wait(forks[right(p)]);
      sem_wait(forks[left(p)]);
   } else {
      sem_wait(forks[left(p)]);
      sem_wait(forks[right(p)]);
   }
}
```

Dining Philosophers

- Broken solution suffers from deadlock if all philosophers acquire lock for left fork and all would wait forever to acquire lock for right fork.
- Solution is to change order of how locks are acquired for one philosopher – Cycle of waiting is broken.

Zemaphores

Use low-level synchronization primitives to build semaphores – **zemaphores**

Does not maintain the invariant that the value of the semaphore, when negative, reflects the number of waiting threads, however this behavior matches the current Linux implementation.

Zemaphore Implementation

```c
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}
```

```c
void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Summary

- Common Problems in Concurrent Programming:
  - **Mutual Exclusion**
  - Controlling **Ordering** of threads
- **Semaphores** are a powerful and flexible primitive
- Can be used exclusively
- Becoming a concurrency expert takes years of effort
  - https://greenteapress.com/wp/semaphores/