Semaphores

Chapter 31

Previously in CS212…

- We’ve looked at locks and condition variables

- We can combine these two concepts to create a new type of synchronization primitive

Semaphores

- Object with an integer value

- We can decrement the value of the integer by 1
  - `sem_wait()` (POSIX semaphores)
  - `P()` (Dijkstra - Dutch for “prolaag” or “try decrease”)

- We can increment the value of the integer by 1
  - `sem_post()` (POSIX semaphores)
  - `V()` (Dijkstra - Dutch for “verhoog” or “increase”)

Semaphore Usage

- We can initialize a semaphore's starting integer value to anything we want

- As we make calls to sem_wait(), we decrease the value and then check if the integer value in negative
  - If so, our thread waits until the value becomes non-negative

- When we make calls to sem_post(), sleeping threads from the sem_wait() operation using the semaphore can then wake up and try to complete their task

Semaphore Implementation

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

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

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);
}
```

Figure 31.17: **Implementing Zemaphores With Locks And CVs**

Semaphore Nuance

- Multiple threads can call sem_wait() causing the integer value and will queue to be woken up

- The sem_post() call does not wait for a condition; it just increments the integer value and wakes up a sleeping thread

- We can envision that our semaphore integer value can go negative if we have multiple calls to sem_wait(); how negative the value is, represents the number of threads waiting
  - Note that this ”invariant” is not always used in implementation (Linux semaphores do not go negative).

Application – Binary Semaphores

- Like a lock
- Integer value represents a 1/0 value to "lock and unlock" a critical section

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

Application - Ordering

- Like a condition variable
- If we choose the correct starting value, we can ensure some simple ordering scenarios
- X = ?

```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

Figure 31.6: **A Parent Waiting For Its Child**

Application - Ordering

```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

Figure 31.6: **A Parent Waiting For Its Child**

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)→sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)**

Application - Ordering

```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

Figure 31.6: **A Parent Waiting For Its Child**

Val	Parent	State	Child	State
0	create(Child)	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	child runs	Run
0		Ready	call sem_post()	Run
1		Ready	inc sem	Run
1		Ready	wake(nobody)	Run
1		Ready	sem_post() returns	Run
1	parent runs	Run	*Interrupt→Parent*	Ready
1	call sem_wait()	Run		Ready
0	decrement sem	Run		Ready
0	(sem≥0)→awake	Run		Ready
0	sem_wait() returns	Run		Ready

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)**

Application - Producer/Consumer

- As before with condition variables, we can create this pattern
- Note the need for locking around the critical section if the queue is > 1 in size

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // Line P1
        sem_wait(&mutex);       // Line P1.5 (MUTEX HERE)
        put(i);                 // Line P2
        sem_post(&mutex);       // Line P2.5 (AND HERE)
        sem_post(&full);        // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);        // Line C1
        sem_wait(&mutex);       // Line C1.5 (MUTEX HERE)
        int tmp = get();        // Line C2
        sem_post(&mutex);       // Line C2.5 (AND HERE)
        sem_post(&empty);       // Line C3
        printf("%d\n", tmp);
    }
}
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

Check out the GitHub class repo for the code example

Application - Reader-Writer Locks

- A unique pattern where we may have multiple threads interested in consuming some data
- If the threads aren’t making any changes, we can allow them to read the data
- However, if a thread wants to write to the data, we need to ensure:
  - There are no readers actively using the data
  - There are no other writers using the data
- Readers need to acquire a lock on the critical section (reader count update), and a lock on the writing capability (write lock)
- Writers must wait until they can acquire the write lock

Check out the GitHub class repo for the code example

Application – Thread Throttling

- Sometimes we may have more threads in use than other resources may be able to handle
- Imagine several hundred threads malloc-ing memory
- This burden on the system might be more than it can handle
- So instead, we can use our semaphore to limit the number of threads that may enter a section of code at once

Check out the GitHub class repo for the code example

Next Time

- We take a closer look at concurrency bugs.