CS:3620 Operating Systems

Semaphores

What is a semaphore?

- Synchronization primitive like condition variables
- Semaphore is a variable with an underlying counter

- Two functions on a semaphore variable

  - Up/post increments the counter, and wakes up one of processes blocked on the semaphore

  - Down/wait decrements the counter and blocks the calling thread if the resulting value is negative
- A semaphore with init value 1 acts as a simple lock (binary semaphore = mutex)

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

Figure 31.3: **A Binary Semaphore (That Is, A Lock)**

Semaphores for ordering

- Can be used to set order of execution between threads like CV
- Example: parent waiting for child (init = 0)


```
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;
}
```

Example: Producer/Consumer (1)

- Need two semaphores for signaling
  - One to track empty slots, and make producer wait if no more empty slots
  - One to track full slots, and make consumer wait if no more full slots
- One semaphore to act as mutex for buffer


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

Example: Producer/Consumer (2)


```
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)

Incorrect solution with deadlock

- What if lock is acquired before signaling?
- Waiting thread sleeps with mutex and the signaling thread can never wake it up


```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // Line P0 (NEW LINE)
        sem_wait(&empty);           // Line P1
        put(i);                     // Line P2
        sem_post(&full);            // Line P3
        sem_post(&mutex);           // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // Line C0 (NEW LINE)
        sem_wait(&full);            // Line C1
        int tmp = get();            // Line C2
        sem_post(&empty);           // Line C3
        sem_post(&mutex);           // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Disclaimer

- *These lecture slides are based on a slide set by Youjip Won (Hanyang University) and Mythili Vutukuru (IIT Bombay)*