5118020-03 Operating System

Conditional Variable & Semaphore

OSTEP Chapters 30 and 31

Shin Hong

Semaphore

- A counter-like synchronization primitive
  - represents the number of available shared resources
  - provides increase and decrease operations
    - decrease the counter when a thread is checking out a resource
      - block the caller thread if the counter became negative
    - increase the counter when a thread is putting back a held resource
      - wake up a blocked thread if exists

- POSIX semaphore
  - sem_init ()
  - sem_wait (), sem_post ()

Semaphore As Mutex

```
sem_t m;
sem_init(&m, 0,1 );

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

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

Semaphore as Ordering Primitive


```c
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;
}
```

Producer Consumer Problem

- There are one or more **producer threads** each of which place a task with data item at a buffer, and one or more **consumer threads** that takes a placed task and process it one by one

- Requirements
  - a producer must be blocked if the buffer is full
  - a consumer must be blocked if there is no task element found in buffer
  - accesses to a shared buffer must be synchronized

Buffer


```c
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use];    // Line G1
    use = (use + 1) % MAX;    // Line G2
    return tmp;
}
```

Producer and Consumer with Semaphore

- Version 1

```
sem_t empty;
sem_t full;

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

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);           // Line C1
        tmp = get();               // Line C2
        sem_post(&empty);          // Line C3
        printf("%d\n", tmp);
    }
}

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

```
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use];    // Line G1
    use = (use + 1) % MAX;    // Line G2
    return tmp;
}
```

Producer and Consumer with Semaphore

- Version 2

```
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);
    }
}
```

Producer and Consumer with Semaphore

- Version 3

```
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);
    }
}
```

Conditional Variable - Motivation

- There are many cases to program a thread to wait until shared variables satisfy a certain condition

- Example: thread join

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Conditional Variable

- A conditional variable is an explicit queue that threads can put themselves on when some state of execution is not desired
  - when it said the condition might be changed, a waiting thread woke up to continue its execution
  - operation
    - wait () `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);`
    - signal () `pthread_cond_signal(pthread_cond_t *c);`
- a conditional variable is used together with a mutex for ensuring mutual exclusion of the condition checking

Example

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

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

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

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

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Producer Consumer with Conditional Variable


```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

```
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 == MAX)
            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);
    }
}
```

Implementing Semaphore with Mutext and Conditional Variables


```
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);
}
```