7 Synchronization Examples

Bounded-Buffer Problem（有限缓冲区问题，也被称为 *producer-consumer problem*）大小为 n 的缓冲区，一开始为空。每次每个 producer 只能往缓冲区里增加一个数据，每次每个 consumer 只能往缓冲区里消耗一个数据。需要设计一个框架使得缓冲区不会溢出，也不存在 consumer 读不到数据。

设计信号量（*semaphore*）：

- 保证互斥：信号量 `mutex`，保证一个时间只有一个进程操作临界区（即操作 buffer），否则 buffer 的 count 会紊乱。
- 对于 consumer 进程保证 buffer 非满：信号量 `full` 表示当前有的数据的总量，当 `full = 0` 表示要阻塞 consumer 进程，因为当前没有数据。
- 对于 producer 进程保证 buffer 非满：信号量 `empty` 表示当前空的数据的数量，当 `empty = 0` 表示要阻塞 producer 进程，因为当前没有空位。

信号量的初始值：`mutex = 1`, `full = 0`, `empty = 1`.

解决方案：

```
// Producer
while(true) {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* critical section: add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}

// Consumer
while(true) {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next_consumed */
    ...
}
```

注意，这里的 `wait` 和 `signal` 是原子操作，同一个时刻只能操作一次。

Readers and Writers Problem（读者写者锁问题）：有一段数据。有一些读者，只能读数据而不能写数据；有一些写者，可以写数据和读数据。由于读者不改变数据，可以允许多个读者同时读数据。在任何时候，只能允许一个写者写数据。

设计信号量（*semaphore*）和变量：

- 保证读写互斥锁 `rw_mutex`；
- 一个变量：读者的数量 `read_count`。
- 关于读者数量 `read_count` 的互斥锁 `mutex`（由于 `read_count` 也是临界区，需要判断是否是第一个/最后一个读者）。

信号量和变量的初始值: `rw_mutex = 1`, `mutex = 1`, `read_count = 0`.

解决方案

```
// Writer
while(true) {
    wait(rw_mutex);
    ...
    /* perform writing */
    ...
    signal(rw_mutex);
}

// Reader
while(true) {
    wait(mutex);
    read_count ++;
    if (read_count == 1) wait(rw_mutex);
    signal(mutex);
    ...
    /* perform reading */
    ...
    wait(mutex);
    read_count --;
    if(read_count == 0) signal(rw_mutex);
    signal(mutex);
}
```

Dining-Philosophers Problem（哲学家就餐问题）：有 n 个哲学家围成一圈吃饭。每个哲学家面前有一碗饭，每两个哲学家之间都有一只筷子，哲学家需要同时拿起左右两边的筷子才能吃饭。设计一个锁协议使得哲学家们可以正常的吃饭，不会导致饿死或者死锁。

解决方案：利用管程和信号量。对于每一个哲学家开一个条件变量，每次都判断一下当前哲学家是否可以吃饭，如果可以就吃否则进入休眠队列。每次一个哲学家吃完饭，判断其两边的哲学家是否可以吃饭，如果可以吃饭则尝试唤醒。

```
monitor dining_philosopher_solution {
    enum State {THINKING, EATING, HUNGRY};
    condition philosopher[5];
    State state[5];
    // philosopher i pick up chopsticks and eating.
    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);        // check whether i can eat!
        if (state[i] != EATING) philosopher[i].wait();
    }
    // philosopher i put down chopsticks and thinking.
    void putdown(int i) {
        state[i] = THINKING;
        // check whether i+1 and i-1 can eat!
        test((i + 4) % 5);
```

```
        test((i + 1) % 5);
    }

    // test whether i can eat.
    void test(int i) {
        if (state[(i + 4) % 5] != EATING && state[(i + 1) % 5] != EATING &&
            state[i] == HUNGRY) {
            state[i] = EATING;
            // if he is waiting, then he can start eating!
            philosopher[i].signal();
        }
    }
    initialization_code () {
        for (int i = 0; i < 5; ++i) state[i] = THINKING;
    }
};
// philosopher i having meal
void having_meal(int i) {
    pickup(i);
    /* Critical Section: EATING */
    putdown(i);
}
```

我们尝试着从筷子的角度来看问题：

```
monitor dining_philosopher_my_solution {
    condition chopstick[5];
    bool state[5];
    // philosopher i pick up chopsticks and eating.
    void having_meal(int i) {
        while(state[i] || state[(i + 1) % 5]) {
            if (state[i]) chopstick[i].wait();
            if (state[(i + 1) % 5]) chopstick[(i + 1) % 5].wait();
        }
        state[i] = true;
        state[(i + 1) % 5] = true;
    }
    // philosopher put down chopsticks and thinking.
    void putdown(int i) {
        state[i] = false;
        state[(i + 1) % 5] = false;
        chopstick[i].signal();
        chopstick[(i + 1) % 5].signal();
    }
    initialization_code () {
        for (int i = 0; i < 5; ++i) state[i] = false;
    }
};
// philosopher i having meal
void having_meal(int i) {
    pickup(i);
    /* Critical Section: EATING */
    putdown(i);
}
```

效率不如从哲学家角度考虑优