6 Process Synchronization

临界区问题 (Critical Section Problem)：假设有 $p_0, p_1, \ldots, p_{n-1}$ 一共 n 个进程，将这些进程种对于共享数据进行操作的代码称为临界区。这些代码可能引起冲突，称为临界区问题。

- 将每一个进程 p_i 分解成：

  ```
  while(true) {
      [entry section]
          critical section
      [exit section]
          remainder section
  }
  ```

  其中，`remainder section` 为不对共享数据进行操作的部分代码，`critical section` 为对共享数据进行操作的部分代码。添加 `entry section` 和 `exit section`，表示临界区的入口段和出口段。

- 解决临界区问题的条件：
  - **互斥性 (Mutual Exclusion)**：当一个进程执行临界区时，其他进程不能执行与该临界区相关的代码。
  - **实时进展 (Progress)**：一个在临界区的进程应该时刻报告运行进展，包括结束后的通知；
  - **有限等待 (Bounded Waiting)**：保证每个进程的等待（等待执行临界区）时间是有限的（避免饿死）。
- 解决临界区问题也有两种方式：抢占式 (Preemptive) 和非抢占式 (Non-preemptive)。

Peterson's Solution（经典解决方案）：针对于有且仅有两个进程的解决方案。

- 前提：`load` 和 `store` 指令是原子性的，也就是说**操作不能被打断**；
- 两个进程在内核态中共享两个变量：

  ```
  int turn;
  bool flag[2];
  ```

  `flag[i]` 表示进程 `i` 是否准备好进入临界区，`turn` 表示现在是轮到哪个进程执行临界区。

  ```
  // For process p_i
  while(true) {
      /* entry section */
      flag[i] = true;
      turn = j;
      while(flag[j] && turn == j);
      /* critical section */
      ...
      /* exit section */
      flag[i] = false;
      /* remainder section */
      ...
  }
  ```

当 `i` 进程准备执行临界区时，先让另一个进程 `j` 尝试执行，如果另外一个进程 `j` 准备好了就先执行，否则进程 `i` 尝试执行。`turn` 的作用是防止如果 P_i, P_j 同时到达则都无法进入临界区的情况。

- 通过验证，可以满足解决临界区问题的三个条件，

硬件系统处理共享的方法：非常消耗资源（锁的处理和硬件速度相比很慢，灵活性差，代码复杂），纯硬件处理使得软件开发更加复杂。

- 内存屏障：某段内存仅供特殊进程使用，其他进程无法访问
- 硬件指令（支持加锁操作）：
  - `test_and_set` 命令（原子操作，对于锁 `lock` 操作）；
  - `compare_and_swap` 命令（原子操作，对于锁 `lock` 和钥匙 `key` 操作）。
- 原子变量（局限性大，仅仅针对于单个变量）。

信号量, *Semaphore*：

- 信号量 S 是整型变量，操作系统底层自带两个原子操作的函数 `wait()`（加锁，`P()`）和 `signal()`（解锁，`V()`）。

```
def wait(S) {
    while(S <= 0);   // 信号量, busy waiting
    S --;
}

def signal(S) {
    S ++;
}
```

- 分类：计数型信号量 (*Counting semaphore*)，二进制信号量（*Binary semaphore*）（其实就是**互斥锁**，*Mutex Lock*）；
- 没有忙等待的信号量实现（直接 block 进程 和 wakeup 进程）。

```
def wait(semaphore *S) {
    if(S -> value <= 0) {
        add this process to S -> list;
        block();
    }
    S -> value --;
}
def signal(semaphore *S) {
    S -> value ++;
    if(S -> value <= 0) {
        remove a process P from S -> list;
        wakeup(P);
    }
}
```

- **死锁** (Deadlock)

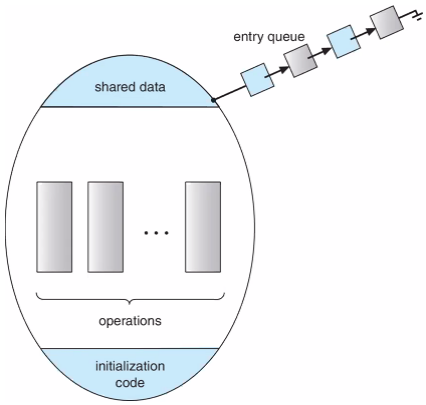
```
// P0
wait(S); wait(Q);
// critical section
signal(S); signal(Q);

// P1
wait(Q); wait(S);
// critical section
signal(Q); signal(S)
```

P_0 得到了锁 S，P_1 得到了锁 Q；双方都无法继续执行下去，这种现象被称为死锁。

解决方法：规定相对优先级，高优先级的进程可以“抢占”低优先级的锁（局限性：除非增加其他规定，否则不能避免饿死）。

管程 (Monitor)：为管理进程间数据同步所设置的模块，由一个锁 (lock)，一个等待访问共享数据的进程队列 (entry queue) 以及若干个条件变量 (condition variables) 构成。

```
monitor monitor_name {
    void func1(...) {
        /*  critical section  */
    }
    void func2(...) {
        /*  critical section  */
    }
    initialization code(...) {...}
};
```



- 实现方式：“锁 + 进程队列” 恰好用**信号量**实现。

```
semaphore mutex;   // initial value = 1;

void func(...) {
    wait(mutex);      // lock
    /*  critical section  */
    signal(mutex);    // unlock
}
```

- **管程**不会有两个进程同时在管程内执行。
- **问题**：如果一个进程在管程内部卡住了，所有进程都卡住了？
- **解答**：需要一个能够支持管程内挂起/休眠的机制——条件变量！

条件变量 (condition variables)：用于实现“管程内休眠”。用 `condition` 定义条件变量，并在需要休眠时候采用 `wait()`，需要唤醒的时候采用 `signal()`。**条件变量的实质是一个存储休眠进程的队列。**进程需要休眠时，调用 `wait()`，将当前进程休眠并加入队列中；当可以继续执行进程时，调用 `signal()`，让队列中的一个进程（可以随机选择）继续执行。

```
condition x;            // condition variable x
```

如何实现条件变量呢？我们需要修改管程的代码加以支持。

```
/* implement monitor */
semaphore mutex;        // initial value = 1;
semaphore next;         // initial value = 0;
int next_count = 0;     // initial value = 0;

void func(...) {
    wait(mutex);

    if (next_count > 0)
        signal(next);
    else
        signal(mutex);
}
```

然后我们实现条件变量的 `wait()` 和 `signal()`。

```
// we can use the semaphores and variables of the monitors
// such as mutex, next and next_count.
struct condition {
    semaphore sleeping;     // initial value = 0
    int sleeping_count = 0; // initial value = 0

    void wait() {
        sleeping_count ++;
        if (next_count > 0)
            signal(next);
        else
            signal(mutex);
        wait(sleeping);
        sleeping_count --;
    }

    void signal() {
        if(sleeping_count > 0) {
            next_count ++;
            signal(pending);
            wait(next);
            next_count --;
        }
    }
};
```

直接理解这段代码可能比较困难，我们先来解释一些基本的变量意义。

- `mutex` 和管程里的意义相同，表示进入队列（entry queue）以及一个互斥锁（mutex lock）；

- `next` 表示因为被暂时“挂起”的进程队列（信号量的本质是个队列）。注意：其并不是因为被“卡住”而无法执行，被强制休眠的进程；而是“挂起”，具体含义后文会提到！
- `next_count` 表示在 `next` 里的进程个数。
- 对于每个条件变量，`sleeping` 表示因为这个条件卡住而进入休眠的进程队列。
- `sleeping_count` 表示 `sleeping` 里进程的个数。

接着，我们来解释代码都在做什么。

- 我们首先修改了管程的函数实现代码，增加了对于 `next` 的判定，意味着如果当前“挂起”队列中有进程，那么就先执行“挂起”队列中的进程。否则，如果该队列为空，则向 mutex （即 entry queue） 中拿新的进程。
- 然后我们看 `wait()`：如果进程卡住了调用 `wait()`，然后该进程需要休眠，于是 `sleeping_count` 自然加一；然后我们可以选择下一个要执行什么进程，同样的也是优先选择“挂起”队列中的进程；接着我们将当前进程休眠 `wait(sleeping)`（由于信号量 `sleeping` 初始值为 0，因此每次 `wait(sleeping)` 都将导致休眠。（我们先跳过 `wait()` 的最后一行）。
- 接着来看 `signal()`：如果当前卡住的问题得到了解决，那么至少一个休眠队列中的进程可以执行，于是如果休眠队列中有进程，那么就 `signal(pending)` 将其执行。**注意：休眠进程的执行会从上次 `wait` 的地方继续往下执行**，也就是 `condition.wait()` 函数的最后一行，将 `sleeping_count` 自减，表示减少了一个卡住的进程，然后继续回到调用 `condition.wait()` 的地方继续往下执行。
- 我们终于可以解释“挂起”队列了！我们要执行一个休眠队列中的进程，然而管程中不允许两个进程同时执行，那么当前进程怎么办呢？挂起！**挂起队列和休眠队列的共同点是：都是暂时停止执行！**挂起队列中的进程是因为唤醒某个休眠进程而被“挂起”暂停执行；而休眠队列中的进程是被卡住了而暂停执行！换句话说，任何时候唤醒“挂起”队列中的进程都能继续执行，而唤起休眠队列中的进程不一定能顺利执行，很有可能被卡住。
- 于是我们每次唤醒前，将当前进程挂起，`next_count` 自增，然后执行 `wait(next)` 放入 `next` 的挂起进程队列中。当然，由于挂起进程任何时候都能继续执行，所以每次可以执行新进程的时候，我们优先考虑的也是挂起进程！这也就解释了 `condition.wait()` 和 `monitor.func(...)` 两处下述 `if-else` 代码的含义：

  ```
  if (next_count > 0)
      signal(next);
  else
      signal(mutex);
  ```

- 当“挂起”的进程被唤醒时，其从上次被挂起的地方继续往下执行，也就是 `condition.signal()` 的最后一句话，将 `next_count` 自减，表示唤醒了一个挂起进程，然后继续往下执行！

理解这个过程的一个很重要的思想是 信号量都是队列！

-