

- 条件竞争 (Race Condition)
- 临界区 (Critical Section)
  - 安全性 (Safety)、活性(Liveness)、以及其他两个要求
- 互斥 (Mutual Exclusion)
  - 忙等待 (busy waiting)，皮特森算法 (Peterson' s Solution)
- 条件变量 (Condition Variables)
  - Wait、Signal
- 信号量 (Semaphores)
  - P、V
- 管程 (Monitors)
  - Hoare、Brinch Hansen、Mesa

## 竞争条件

两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序

- 适当安排使两个进程不同时在临界区中，就可以避免竞争条件

## 临界区

对共享内存进行访问的程序片段

- 安全性**safety**：最多一个进程在临界区
- 活性**liveness**：（达到预期状态）任何进程要求进入临界区，最后一定会进入
- 不对CPU 的速度和数量做任何假设。
- 临界区外运行的进程不得阻塞其他进程。

## 互斥

- 忙等待：

定义：连续测试一个变量直到某个值出现为止

只有认为等待时间会比较短时，才使用忙等待

- 其缺点是等待进程循环探测竞争条件，浪费了时间片
- 用于忙等待的锁，叫自旋锁

• 皮特森算法

19、简要说明Peterson算法的原理，Peterson算法符合safety和liveness性质吗？说明理由。

原理

- 临界资源是能够被多个线程共享的资源，临界区是对临界资源进行操作的那一段代码。在进入其临界区之前，各个进程使用其进程号0或1作为参数来调用 `enter_region` 。
- 维持全局数组 `flag`，0 和 1 代表每个线程是否尝试进入临界区，初始化为0
- 维持全局变量 `turn`，如果两个线程都申请进入临界区，那么 `turn` 将决定最终能够进入临界区的线程编号
- 进程将等待直到能安全地进入临界区。安全即其他线程 `flag=0` 或 `turn` 决定选择此进程
- 在完成对共享变量的操作之后，进程将调用 `leave_region`，表示操作已完成，此时其他希望进入临界区的进程可以进入。

算法符合 safety 性质理由：

**safety**：程序运行中不会进入非预期的状态

- 竞争时只会有一个进程进入临界区，因为 `turn` 的值只能为0或1，不可能有两个值。
- 线程*i*在临界区时，始终有 `flag[i]==1` 和 `turn==i`，其他线程被阻止进入

算法符合 liveness 性质理由：

**liveness**：程序运行中预期状态一定会到达

- 线程*i*在离开临界区时，会设置 `flag[i]==0`，表示不再占有临界区，因此其他线程可以进入临界区

4) 算法四：Peterson’s Algorithm。为了防止两个进程为进入临界区而无限期等待，又设置了变量 `turn`，每个进程在先设置自己的标志后再设置 `turn` 标志。这时，再同时检测另一个进程状态标志和允许进入标志，以便保证两个进程同时要求进入临界区时，只允许一个进程进入临界区。

$P_i$ 进程:	$P_j$ 进程:	
<code>flag[i]=TRUE;turn=j;</code>	<code>flag[j]=TRUE;turn=i;</code>	//进入区
<code>while(flag[j]&amp;&amp;turn==j);</code>	<code>while(flag[i]&amp;&amp;turn==i);</code>	//进入区
<code>critical section;</code>	<code>critical section;</code>	//临界区
<code>flag[i]=FALSE;</code>	<code>flag[j]=FALSE;</code>	//退出区
<code>remainder section;</code>	<code>remainder section;</code>	//剩余区

具体如下：考虑进程  $P_i$ ，一旦设置 `flag[i] = true`，就表示它想要进入临界区，同时 `turn = j`，此时若进程  $P_j$  已在临界区中，符合进程  $P_i$  中的 `while` 循环条件，则  $P_i$  不能进入临界区。若  $P_j$  不想要进入临界区，即 `flag[j] = false`，循环条件不符合，则  $P_i$  可以顺利进入，反之亦然。本算法的基本思想是算法一和算法三的结合。利用 `flag` 解决临界资源的互斥访问，而利用 `turn` 解决“饥饿”现象。

条件变量

- 进程进入管程后被阻塞多种原因，每种设置一个条件变量，每个条件变量保存了一个等待队列，用于记录因该条件变量而阻塞的所有进程
- 只能wait和signal
- x.wait: x条件不满足时，调用管程的进程调用wait进入等待队列，释放管程
- x.signal: x条件有了，调用signal唤醒一个等待进程
- 如果向一个条件变量发送信号，但该条件变量上没有等待进程，信号将会丢失。也就是说，**wait 操作必须在signal之前执行，而且signal之前要判断等待队列是否为空**

## 信号量

使用一个整型变量来累计唤醒次数

- 可实现同步、互斥、前驱

## 管程

- 定义：资源管理程序：一个管程定义了一个数据结构和能被并发进程执行的一组操作，这组操作能同步进程并修改管程数据
- 作用&必要性：管程把共享资源的操作封装起来，每次只允许一个进程进入，互斥，避免对临界区的访问分散在各个进程中造成了混乱、

## 3种分歧

分歧原因：对条件执行 signal 后，系统调度只能选择其中一个进程恢复运行

Hansen 管程与 Hoare 管程	
<b>Hansen-style: Deposit()</b> <pre>lock-&gt;acquire(); while (count == n) {     notFull.wait(&amp;lock); } Add thing; count++; notEmpty.signal(); lock-&gt;release(); }</pre>	<b>Hoare-style: Deposit()</b> <pre>lock-&gt;acquire(); if (count == n) {     notFull.wait(&amp;lock); } Add thing; count++; notEmpty.signal(); lock-&gt;release(); }</pre>
<b>■ Hansen管程</b> <ul style="list-style-type: none"> <li>▣ 条件变量释放仅是一个提示</li> <li>▣ 需要重新检查条件</li> </ul>	<b>■ Hoare管程</b> <ul style="list-style-type: none"> <li>▣ 条件变量释放同时表示放弃管程访问</li> <li>▣ 释放后条件变量的状态可用</li> </ul>
<b>■ 特点</b> <b>高效</b>	<b>■ 特点</b> <b>低效</b>

[http://blog.csdn.net/Dylan\\_Fran](http://blog.csdn.net/Dylan_Fran)

## Hoare

signal后让新唤醒的进程运行，挂起另一个

## 让等待队列中的优先

What if the current thread has more things to do?

```
if (only one item)
    signal(empty);
something else
end;
```

After the signaled thread exits monitor or waits.

## Brinch Hansen

发信号者立刻退出管程，signal是管程过程的最后一句

signal把条件变量等待队列里的就绪

让管程中的线程优先

```
put item into buffer;
if (only one item)
    signal(empty);
end;
```

## Msea

让发信号者继续运行直到推出管程，再唤醒等待进程

signal把条件变量等待队列里的就绪

Continues its execution

```
if (only one item)
    signal(empty);
something else
end;
```

Can be preempted

This is easy to implement!

## RCU

- 支持一个写和多个读同时进行（死锁预防破坏互斥条件）

发布-订阅机制：读取时恰好读到新写入的东西，需要保证读到的是完整的

宽限期：从写更新开始到最后一个读者读完（再删）

- 生产者-消费者问题 (The Producer-Consumer Problem)
- 读者写者问题 (The Readers and Writers Problem)
  - 读者偏向、写者偏向、相对公平
- Read-Copy-Update (RCU)
  - 订阅-发布机制 (Publish-Subscribe Mechanism)
  - 宽限期 (RCU Grace period)
- 哲学家就餐问题 (The Dining Philosophers Problem)
  - 饿死(starvation)、死锁 (deadlock)

## 信号量代码

P是-, V是+

- PV不满足时是在一直等待的, P的变量要大于0才放行
- 简单的PV夹住操作过程: 互斥
- V表示某件事已完成(资源已提供), P检测是否完成: 同步(先后次序, 前驱)
- V恢复访问是唤醒的是队首进程
- 技巧1: 不同信号量代表不同事件
- 技巧2: 代码基本都是对称的
- 技巧3: 多个不互斥的读者要维持count变量, 只有第一个读才去互斥写防止重复, 这种全局变量(访问并修改)需要互斥上锁!
- 技巧4: 适当制定规则避免死锁

梳理转化问题中存在的关系:

- 数量上的限制条件转化为P(S): 新变量S大于0才放行
- 注意区分永久工作的和一次性工作的, 是否加while(1)

关系转化代码:

- 一个盘子的互斥需要一个信号量去PV夹, P代表盘子不让用
- 其他人如果也想让盘子进入不让用状态, 也调用对应的P

## 哲学家问题

## 法一管程：

记录哲学家状态：吃，想，想吃

```
#define N 5          /* number of philosophers */
#define LEFT (i-1+N)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0    /* philosopher is thinking */
#define HUNGRY 1      /* philosopher is trying to get forks */
#define EATING 2      /* philosopher is eating */

int state[N];        /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N];      /* one semaphore per philosopher */

void philosopher(int i) {
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

放下叉子时，检测左右两边的人是否可以开吃

```
void take_forks(i){ /* i: philosopher number, from 0 to N-1 */
    mutex.P();      /* enter critical region */
    state[i] = HUNGRY; /* record that philosopher i is hungry */
    test(i);        /* try to acquire 2 forks */
    mutex.V();      /* exit critical region */
    s[i].P();       /* block if forks were not acquired */
}

void put_forks(i){ /* i: philosopher number, from 0 to N-1 */
    mutex.P();      /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT);      /* see if left neighbor can now eat */
    test(RIGHT);      /* see if right neighbor can now eat */
    mutex.V();       /* exit critical region */
}
```

test作用

- 当此人想吃且左右人都不在吃的时候，才可以拿起两个筷子

```
void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[i] = EATING;
        s[i].V();
    }
}
```



## 法二信号量

```
semaphore chopstick[5]={1,1,1,1,1}; //初始化信号量
semaphore mutex=1; //设置取筷子的信号量
Pi(){ //i 号哲学家的进程
    do{
        P(mutex); //在取筷子前获得互斥量
        P(chopstick[i]); //取左边筷子
        P(chopstick[(i+1)%5]); //取右边筷子
        V(mutex); //释放取筷子的信号量
        eat; //进餐
        V(chopstick[i]); //放回左边筷子
        V(chopstick[(i+1)%5]); //放回右边筷子
        think; //思考
    } while(1);
}
```