

Lecture 4: 知识 - 同步与互斥

进程同步与异步基础概念

进程同步与异步

概念：同步

- 同步亦称为**直接制约关系**，它指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的工作次序而产生的制约关系；
- 进程间的直接制约关系就是源于它们之间的**相互合作**；

概念：异步性

- 指各**并发执行的进程以各自独立的，不可预估的速度**向前推进。

进程互斥

概念：竞争条件

- 概念：多个进程并发访问和操作同一数据，并且执行结果与特定访问顺序有关，称为竞争条件；

概念：资源共享方式

- 1. 互斥共享方式
 - 系统中的某些资源，虽然可以提供给多个进程使用，但**一个时间段内只允许一个进程**访问该资源 (**临界资源**)；
- 2. 同时共享方式
 - 系统中的某些资源，允许**一个时间段内**由多个进程**同时**对它们进行访问；

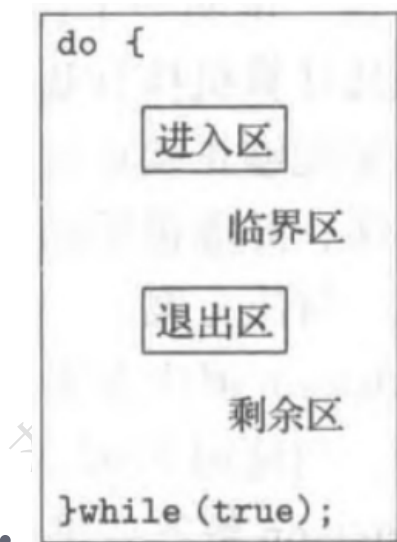
概念：临界资源

- 概念：
 - 指一个时间段内只允许一个进程使用的资源称为临界资源。
 - 许多物理设备 (比如摄像头，打印机)都属于临界资源。
 - 此外还有许多变量，数据，内存缓冲区等都属于临界资源。
- 特点：
 - 临界资源必须**互斥地**进行访问，亦称为间接制约关系；
 - 当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待；
- 注意：临界资源与共享资源区别
 - 临界资源：在同一个时间内**不能**被多个进程访问；
 - 共享资源：在同一个时间内**能够**被多个进程访问；

概念：临界区以及四类分区

- 目的：

- 实现对对临界资源的互斥访问；
- 概念：
 - 每个进程有一段代码，称为临界区，
 - 临界资源的互斥访问可以在逻辑上分为 4 个部分；
- 作用：
 - 负责检查是否可进入临界区：
 - 1. 若可进入，则应设置正在访问临界资源的标志（进入区后上锁），以阻止其他进程同时进入临界区；
 - 2. 若不可进入，则等待资源；
- 四类分区
 - 进入区
 - 在进入临界区前，每个进程应请求许可；
 - 负责检查是否可进入临界区，若可进入，则应设置正在访问临界资源的标志
 - 临界区
 - 每个进程中访问临界资源的那段代码。当一个进程在临界区内执行时，其他进程不允许在它们的临界区内执行；
 - 退出区
 - 负责解除正在访问临界资源的标志（解锁）
 - 剩余区
 - 其他处理
- 图示：



- 注意
 - 临界区是进程中访问临界资源的代码段；
 - 进入区和退出区是负责实现互斥的代码段；
 - 临界区也可称为临界段；
- 代码参考：

```

//伪代码
do{
    entry section;    //进入区 `->` 上锁
  
```

```
critical section; //临界区
exit section;      //退出区 `->` 解锁
remainder section; //剩余区
}while(true)
```

临界区访问原则

概念：临界区的四大访问原则

- 概念：
 - 1. 空闲让进：
 - 临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区；
 - 2. 忙则等待：
 - 当已有进程进入临界区时，其他试图进入临界区的进程必须等待；
 - 3. 有限等待：
 - 对请求访问的进程，应保证能在**有限时间内**进入临界区（保证不会饥饿）；
 - 4. 让权等待：
 - 进不了临界区的进程，要释放处理机，**防止忙等**（无法使用，但一直在等待）；

• 举例

• 题目

【2020 统考真题】下列准则中，实现临界区互斥机制必须遵循的是（ ）。

I. 两个进程不能同时进入临界区

II. 允许进程访问空闲的临界资源

III. 进程等待进入临界区的时间是有限的

IV. 不能进入临界区的执行态进程立即放弃 CPU

A. 仅 I、IV

B. 仅 II、III

• C. 仅 I、II、III

D. 仅 I、III、IV

• 解析

- **I**：当已有进程进入其临界区时，其他试图进入临界区的进程必须等待，是“忙则等待”准则，正确；
- **II**：访问临界资源的代码称为临界区，相应临界资源空闲时，应立即允许该请求进程进入临界区，是“空闲让进”准则，**II** 正确；
- **III**：对要求访问临界资源的进程，应保证能在有限时间内进入自己的临界区，是“有限等待”准则，**III** 正确；
- **IV**：当进程不能进入自己的临界区时，应释放 CPU，是“让权等待”准则。**I** **II** 和 **III** 是互斥机制必须遵循的原则。**IV** 是“让权等待”准则，不一定非得实现，如皮特森算法，**IV** 错误，故本题选 **C**；

进程互斥的软件实现方法

单标志法

概念：单标志法

- 思想
 - 两个进程在访问完临界区后会把使用临界区的权限转交给另一个进程；
 - 也就是说：每个进程进入临界区的权限**只能被另一个进程赋予**；
- 缺点：
 - 违背了**空闲让进**原则
- 举例：
 - 进程 A 使用完之后，把权限给了进程 B。但进程 B 当前不使用，而进程 A 此时如果过了一会还想要使用，因此 B 一直没开始使用，但一直有着 A 给的权限，导致当前想要使用的 A 无法从 B 上获得使用权，进而导致临界资源空闲；

代码示例

```
/*
1. turn的初值为0，即刚开始只允许0号进程进入临界区。
2. 若P1先上处理机运行，则会一直卡在5。直到 P1 的时间片用完，发生调度，切换 P0上处理机运行。
3. 代码1不会卡住 P0，P0 可以正常访问临界区，在 P0访问临界区期间即时切换回 P1，P1依然会卡。
*/
int turn = 0; //turn 表示当前允许进入临界区的进程号

//P0 进程
while (turn !=0); //1
critical section; //2
turn = 1; //3
remainder section; //4

//P1 进程
while (turn != 1); //5 进入区
critical section; //6 临界区
turn = 0; //7 退出区
remainder section; //8 剩余区
```

双标志先检查

概念：双标志先检查法

- 思想
 - 设置一个布尔型数组 `flag[]`，数组中各个元素用来标记**进程想进入临界区的意愿**；
 - 每个进程在进入临界区之前先**检查当前有没有别的进程想进入临界区**，如果没有，则把自身对应的标志 `flag[i]` 设为 `true`，之后开始访问临界区；
- 问题：
 - 当进程 AB 同时并发运行时 `-> while(flag[1]); -> while(flag[0]); -> flag[0] = true; -> flag[1] = true;`
 - 这种情况下可能两个进程都进入运行；
 - 问题的核心：**检查&上锁两个操作并没有一起完成**，而是分开成两个指令；
 - 违背了**忙则等待**原则；

代码示例

```

//flag[0] = true 表示0号进程P0想要进入临界区
bool flag[2];      //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;   //刚开始设置为两个进程都不想进入临界区

//P0 进程
while(flag[1]);    //1
flag[0] = true;    //2
critical = false;  //3
flag[0] = false;   //4
remainder section;

//P1 进程
while(flag[0]);    //5 如果此时P0想进入临界区，P1就一直循环等待
flag[1] = true;    //6 标记为P1进程想要进入临界区
critical section;  //7 访问临界区
flag[1] = false;   //8 访问完临界区，修改标记为P1不想使用临界区
remainder section;

```

双标志后检查法

概念：双标志后检查法

- 思想
 - 双标志先检查的改版；
 - 前一个算法的问题是**先检查后上锁**，但是这两个操作又无法一气呵成，因此导致了两个进程同时进入临界区的问题；
 - 因此，就有了**先上锁，后检查**的方法；
- 问题
 - 违背了**空闲让进**和**有限等待**原则；
 - 会因各进程都无法访问临界资源而产生**饥饿**现象；
 - 两个进程都争着想进入临界区，但是谁也不让谁，最后谁都无法进入临界区；

代码实现

```

bool flag[2];      //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;   //刚开始设置为两个进程都不想进入临界区

//P0 进程
flag[0] = true;    //1
while(flag[1]);    //2
critical = false;  //3
flag[0] = false;   //4
remainder section;

//P1 进程
flag[1] = true;    //5 如果此时P1想进入临界区，P1就一直循环等待
while(flag[0]);    //6 标记为P0进程想要进入临界区

```

```
critical section;//7 访问临界区
flag[1] = false; //8 访问完临界区，修改标记为P1不想使用临界区
remainder section;
```

Peterson 算法

概念：Peterson 算法

- 思想
 - 如果双方都争着想进入临界区，那可以让进程尝试互让，主动让对方先使用临界区；
 - 核心：
 - 对 `turn` 变量的更改。表示当前的意愿；
 - 进入区动作：
 - 1. 主动争取；
 - 2. 主动谦让；
 - 3. 检查对方是否也想使用，且最后一次是不是自己说了“客气话”；
 - 谁最后说了“客气话”，谁就失去了行动的优先权；
- 问题
 - 解决了互斥问题，遵循了空闲让进，忙则等待，有限等待三个原则，但是依然未遵循让权等待的原则 -> 当前访问不了临界资源的进程，依然需要在 `while` 循环中执行，占用 CPU 资源；
 - 它是上面三种最好的，但是仍然能改进

实现：Peterson 算法

- 真题：2010年
- 以下描述的代码是 Peterson 算法的实例，既设置 `flag` 标志表明进程是否请求进入临界区，又设置共享变量 `turn` 规定进程进入临界区的顺序；
- 在进入临界区前，进程 P 检查另一个进程 P 的标志 `flag[1]`，又检查共享变量 `turn`，可以保证两个进程同时希望进入临界区时，但只有一个进程能进入临界区，能确保互斥；
- 因为阻塞的条件是 `flag[1]&&turn==1`（以进程 P0 为例），因此如果此时 `turn==1`，则进程 1 顺利进入，进程 0 阻塞。因为进程 1 执行完成之后、会将其 `flag[1]=false`，因此运行进程 0 继续执行，避免了饥饿现象；
 - 若把条件改为 `flag[1] || turn==1`，则会发生饥饿；

```
bool flag[2]; // 表示进入临界区意愿的数组，初始值都是false
int turn = 0; // turn 表示优先让哪个进程进入临界区

//P0 进程
flag[0] = true; //1
turn = 1; //2 表示优先让对方进入临界区
while (flag[1] && turn == 1); // 3 当turn=1时，表示当前应当进程1进，所以0在此while循环
// 着阻塞等待。所以违背了让权等待原则
critical section;//4
flag[0] = false; //5
remainder section;
```

```

//P1 进程
flag[1] = true; //6 表示自己想进入临界区
turn = 0; //7 可以优先让对方进入临界区
while(flag[0] && turn == 0); //8 对方想进，且最后一次是自己让它进，那自己就循环等待
critical section; //9
flag[1] = false; //10 访问完临界区，表示自己已经不想访问临界区
remainder section;

```

分析：互斥代码实现

分析：真题：2021年 - 分析互斥代码实现

• 代码：

功能描述	方法1	方法2
<pre> Semaphore S; wait(S){ while(S<=0); S=S-1; } signal(S){ S=S+1; } </pre>	<pre> Semaphore S; wait(S){ 关中断; while(S<=0); S=S-1; 开中断; } signal(S){ 关中断; S=S+1; 开中断; } </pre>	<pre> Semaphore S; wait(S){ 关中断; while(S<=0){ 开中断; 关中断; } S=S-1; 开中断; } signal(S){ 关中断; S=S+1; 开中断; } </pre>

• 问题：

- (1) 方法一和方法二是否能够正确实现互斥？
- (2) 用户程序能否使用开/关中断指令实现临界区互斥？

• 解答：

• 问题一：

- 方法一不能，方法二能；
- 此问题当中的 `signal(S)` 不会产生问题，因为 `S=S+1` 在关中断后可以连续的完成，因此有问题的部分只会出现在 `wait(S)` 方法中；
- 而 `wait(S)` 方法中主要有两个操作：（1）判断信号量的大小；（2）执行信号量减1；需要对两个方法分别分析这两个操作的实现是否产生死锁；
- 方法一：在方法一当中，操作一和操作二都在一次关中断中执行。此时分为两种情况：
 - 情况一：`S>0`，此时程序不会进入 `while` 循环，直接进入到了 `S=S-1` 当中执行，因此不会产生死锁；
 - 情况二：`S<=0`，此时程序会进入到 `while` 循环，等待 `S>0` 后再跳出循环。但因为此时程序处于关中断阶段，其他进程中的 `signal` 当尝试访问 `S` 信号量

时，会因为中断而产生阻塞，进而导致此时进程产生死锁，`wait` 和 `signal` 都被阻塞，因此方法二不行；

- 方法二：在方法二中，由于在执行完操作 1 后程序就开中断，因此此时其他的进程中的 `signal` 方法可以尝试 `S=S+1`，进而打破当前 `wait` 中的阻塞，因此方法二可以；
- 问题二：
 - 用户程序不能使用开/关中断指令实现临界区互斥；
 - 因为开中断和关中断指令都是**特权指令**，不能在用户态下执行，只能在内核态下执行，所以无法使用开关中断指令实现临界区互斥；

进程互斥的硬件实现方法

引入：为什么需要进程互斥的硬件实现方法

- 基于软件的解决方案 (如 `Peterson`) 并不保证在现代计算机体系结构上正确工作。因此可以考虑硬件方法

中断屏蔽方法

概念：中断屏蔽方法

- 思想
 - 利用开关中断机制实现 `->` 在修改共享变量时只要禁止中断出现，即可实现单处理器环境下的临界区问题；
 - 与原语的实现思想相同：即在某进程开始访问临界区到结束访问为止都不允许被中断，也就不能发生进程切换
- 特点
 - 优点
 - 简单，高效
 - 往往为非抢占式内核所采用；
 - 缺点
 - 不适合用于**多处理机**；
 - 只适用于操作系统内核进程，**不适用于用户进程**：因为开/关中断指令只能允许在内核态；

```
关中断； //关中断后不允许当前进程被中断，也必然不会发生进程切换
临界区；
开中断； //直到当前进程访问完临界区，再执行开中断指令
```

TestAndSet

概念：`TestAndSet` 指令

- 思想
 - 简称 `TS` 指令，也有地方称为 `TestAndSetLock` 指令，或 `TSL` 指令；
 - `TSL` 指令是用硬件实现的，执行的过程**不允许被中断**，只能**一气呵成**；

- 特点
 - 优点
 - 1. 实现简单;
 - 2. 无需像软件实现方法那样, 不需要严格检查是否会有逻辑漏洞
 - 3. 适用于多处理机环境
 - 缺点
 - 不满足**让权等待原则** -> 在当前临界区被 **TS** 上锁的情况下, 其他的访问进程需要在 **TestAndSet** 中一直被阻塞;
 - 真题: 2016年
 - 暂时无法进入临界区(等待进入临界区的)的进程会占用 **CPU** 并循环执行 **TSL** 指令, 从而导致**忙等** -> **CPU** 占用, 表示等待进入临界区的进程不会主动放弃 **CPU**;

实现

- TSL 实现

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

```
//布尔型共享变量lock表示当前临界区是否被加锁
//true 表示已加锁, false表示未加锁
bool TestAndSet(bool *lock){
    bool old;
    old = *lock;    //old用来存放lock原来的值
    *lock = true;   //无论之前是否已加锁, 都将lock设为true
    return old;     //返回lock原来的值
}
```

- 使用

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

```
//一下是使用TSL指令实现互斥的算法逻辑
while(TestAndSet(&lock)); //上锁并检查
//临界区代码段
lock = false; //解锁
//剩余区代码段
```

Swap 指令

概念: Swap 指令

- 思想
 - 也叫 Exchange 指令, 或简称 XCHG 指令
 - Swap 指令是用硬件实现的, 执行的过程不允许被中断, 只能一气呵成。
- 特点
 - 优点
 - 1. 实现简单;
 - 2. 无需像软件实现方法那样严格检查是否会有逻辑漏洞;
 - 3. 适用于多处理机环境;
 - 缺点
 - 1. 不满足让权等待原则;
 - 2. 暂时无法进入临界区的进程会占用 CPU 并循环执行 TSL 指令, 从而导致忙等

代码实现

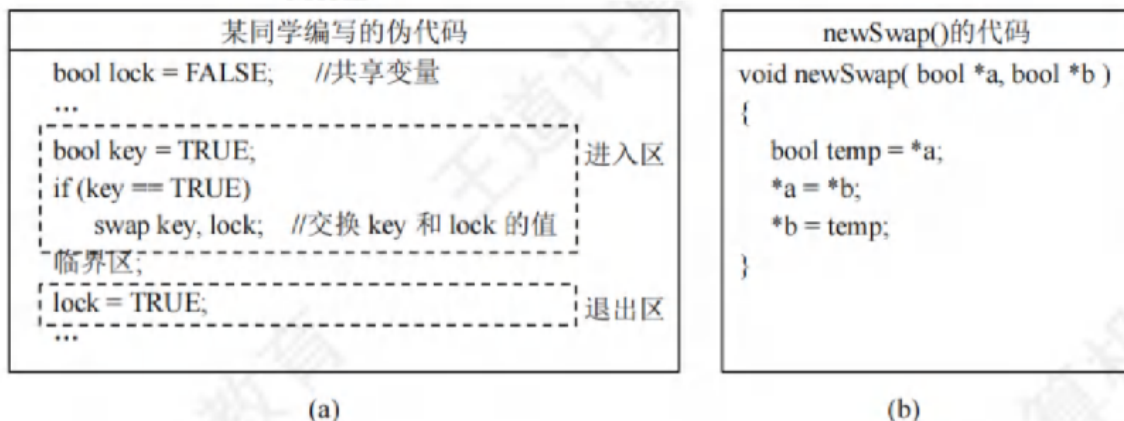
```
//Swap 指令的作用是交换了两个变量的值
Swap(bool *a, bool *b){
    bool temp;
    temp = *a;
    *a = *b;
    *a = temp;
}
```

```
//以下是用Swap指令实现互斥的算法逻辑
//lock表示当前临界区是否被加锁
bool old = true;
while(old == true)
    Swap(&lock, &old);
//临界区代码段
lock = false;
//剩余区代码段
```

分析: Swap 与临界区实现

- 题目:

【2023 统考真题】现要求学生使用 swap 指令和布尔型变量 lock 实现临界区互斥。lock 为线程间共享的变量，当 lock 的值为 TRUE 时线程不能进入临界区，为 FALSE 时线程能够进入临界区。某同学编写的实现临界区互斥的伪代码如下图所示。



请回答下列问题。

- 1) 图(a)的伪代码中哪些语句存在错误？将其改为正确的语句（不增加语句的条数）。
- 2) 图(b)给出了交换两个变量值的函数 newSwap()的代码，是否可以用函数调用语句“newSwap(&key, &lock)”代替指令“swap key, lock”以实现临界区互斥？为什么？

• 解析：

(1)

正确代码如下：

```
bool lock = FALSE; // 用于实现互斥的共享变量
...
bool key = TRUE; // 表示当前线程想进入临界区；
// 注意！这里必须得是while而不能是if，不然起不到阻塞线程的作用
while(lock==TRUE){}
swap key,lock; // 把lock设为真，表示自己进入临界区，阻塞其他进程

临界区；

lock = FALSE;
```

(2)

不能使用函数调用实现临界区互斥。因为函数调用不具有原子性，其执行过程当中为开中断，可能会出现同时多个线程一起进入newSwap中，或者线程在执行newSwap中、可能会进行线程的切换。

而swap为硬件实现的指令，执行过程当中关中断，可以实现临界区；

进程互斥：锁

概念：互斥锁（自旋锁）

• 思想：

- 解决临界区最简单的工具就是互斥锁（mutex lock）；

- 其内部实现机制通常也依赖于硬件指令；
- 过程：
 1. 一个进程在进入临界区时应获得锁；
 2. 在退出临界区时释放锁；
 3. 函数 `acquire()` 获得锁，而函数 `release()` 释放锁；
- 优点：
 - 等待期间不用切换进程上下文，多处理器系统中，**若上锁的时间短，则等待代价很低**；
- 缺点：
 - 互斥锁的主要缺点是：**忙等待**；
 - 需忙等，进程时间片用完才下处理机，违反让权等待；
 - 当有一个进程在临界区中，任何其他进程在进入临界区时必须连续循环调用 `acquire()`；
 - 当多个进程共享同一 `CPU` 时，就浪费了 `CPU` 周期；
- 使用环境：
 - 互斥锁通常用于多处理器系统；
 - 一个线程可以在一个处理器上等待，不影响其他线程的执行；一个核忙等，其他核照常工作，并快速释放临界区；

代码实现

```
acquire()
{
    //忙等待...
    while(!available)
    {
        //...
    }
    available = false; // 获得锁
}

release()
{
    available = true; // 释放锁
}
```

信号量机制

引入

- 上述的实现方式：
 - 四种软件实现方式（单标志法、双标志先检查、双标志后检查、**Peterson 算法**） + 三种硬件实现方式（中断屏蔽方法、**TS/TSL 指令**、**Swap/XCHG 指令**）
- 无法解决的问题：无法实现**让权等待**；

信号量机制

概念：信号量基础概念

- 信号量
 - 1. 其实就是一个变量(可以是整数，也可以是更复杂的记录型变量)；
 - 2. 可以用一个信号量来表示系统中某种资源的数量；
 - 3. 使用 `PV` 操作对信号量进行操作；
- 原语
 - 是一种特殊的程序段，其执行只能一气呵成，不可被中断；
 - 原语是由**关中断/开中断**指令实现的；
- 信号量与忙等：
 - 1. 操作 `wait()` 和 `signal()` 并没有完全取消忙等待；
 - 2. 将忙等待（让权等待）从**进入区移到临界区**，并将忙等待的区域限制在操作 `wait()` 和 `signal()` 的临界区内，而这些区比较短；
- 结论：临界区几乎不被占用，忙等待很少发生，而且所需时间很短；

概念：PV 原语

- 性质：
 - `PV` 操作是一种低级进程通信原语；
 - `PV` 操作是**两个不可被中断**的原语；
- 概念：
 - `wait(S)` 原语和 `signal(S)` 原语：可以把原语理解为我们自己写的函数，函数名分别为 `wait` 和 `signal`；
 - `wait` $\rightarrow$ `proberen` $\rightarrow$ P 操作；
 - `signal` $\rightarrow$ `verhogen` $\rightarrow$ V 操作；
 - 信号量 `S`：函数调用时传入的一个参数；
 - 常把 `wait(S)`，`signal(S)` 两个操作分别写为 `P(S)`，`V(S)`；
- 原因：
 - 软件解决方案主要问题是由进入区的各种操作无法一气呵成；
 - 因此如果能把进入区、退出区的操作都用**原语**实现，使这些操作能一气呵成就能避免问题；
- P 操作：
 - 概念：
 - 1. 当一个进程执行操作 `wait()` 并且发现信号量值不为正时，它必须等待；
 - 2. 该进程不是忙等待而是**阻塞**自己：阻塞操作将一个进程放到与信号量相关的**等待队列**中，并且将该进程状态切换成**等待状态**；
 - 3. 控制转到 CPU 调度程序，以便选择执行另一个进程；
 - 图示：

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- V 操作:

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

整型信号量

概念：整型信号量

- 原理
 - 用一个整数型的变量作为信号量，用来表示系统中某种资源的数量。

- 操作

- 对信号量的操作只有 3 种
- 初始化
- P 操作

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- V 操作

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

- 问题
 - 不满足让权等待原则，会发生**忙等**；

示例

- `int S`：表示信号量 -> 表示系统中一种有限的资源的数量；

```
//举例某计算机系统中有一台打印机
int S = 1; //初始化整型信号量s，表示当前系统中可用的打印机资源数

void wait (int S){ //wait 原语,相当于进入区
    while(S<=0);    //如果资源不够,就一直循环等待
    S = S-1;
}

void signal(int S){ //signal原语,相当于退出区
    S = S+1;        //使用完资源后，在退出区释放资源
}
```

```
//P0 进程
wait(S);      //进程区, 申请资源
使用打印机资源 //临界区, 访问资源
signal(S);    //退出区, 释放资源

//P1 进程
wait(S);
使用资源
signal(S);

//P2 进程
//同上略
```

记录型信号量

概念：记录型信号量

- 原理
 - 用记录型数据结构表示的信号量；
 - 用于解决整型信号量的忙等问题
- 组成：信号量组成：
 - 1. `S.value` 信号量值：其初值表示系统中某种资源的数目；
 - 2. `L` 进程链表：
 - (1) 当一个进程必须**等待信号量**时，就被添加到进程链表；
 - (2) 操作 `signal()` 从等待进程链表上取走一个进程，并加以唤醒；
- 实现：进程链表
 - 可以通过实现**每个进程控制块 PCB 的一个链接字段**，实现进程链表；
 - 向链表中增加、删除进程，以便确保有限等待的一种方法是采用 `FIFO` 队列；
- 注意：
 - 信号量的正确使用不依赖于信号量的进程链表的特定排队策略；

概念：信号量操作实现

- 单处理器：
 - 可以直接使用**屏蔽中断的方式**，来保证调用 `PV` 操作时可以原子的执行；
- 多处理器环境：
 - 分析：
 - 对于多处理器环境，**每个处理器**的中断都应被禁止；
 - 否则在不同处理器上不同的运行进程，可能会以任意不同方式一起交织执行；
 - 每个处理器中断的禁止会很困难，也会严重影响性能。
 - 方法：
 - `SMP` 系统应提供其他加锁技术如：`compare_and_swap()` 或自旋锁，以确保 `wait()` 与 `signal()` 原子执行；

信号量代码定义

```
//记录型信号量的定义
typedef struct{
    int value;           //剩余资源数
    struct process *L;   //等待队列 '->' 指向等待它的进程，比如进程C因为value值为-1、无法访问时，此指针指向当前等待的进程；
}semaphore;
```

wait 操作

- wait 操作 -> 也可以记为 P();

```
//某种进程需要使用资源时,通过wait原语申请
void wait(semaphore S){
    S->value--;
    if(S->value < 0){
        add this process to S->list ;
        block(S.L); //进入阻塞态，并把挂导信号量S的等待队列
    }
}
```

signal 操作

- signal 操作 -> 也可以记为 V();
- 若加 1 后仍是 $S.value \leq 0$ ，表示：依然有进程在等待该类资源，因此应调用 wakeup 原语唤醒等待队列中的第 1 个进程
- 被唤醒进程：从阻塞态 -> 就绪态；

```
//进程使用完资源后,通过signal 原语释放
void signal(semaphore *S){
    S->value++;
    if(S->value <= 0){
        remove a process P from S->list ;
        wakeup(S.L); //释放资源后，若还有别的进程在等待这种资源，则使用wakeup原语唤醒等待队列中的一个进程，该进程从阻塞态变为就绪态
    }
}
```

- 注意：若考试中出现 P(S)，V(S)的操作,除非特别说明，否则默认 S 为记录型信号量；

信号量机制实现

进程互斥

改变：信号量机制实现进程互斥

1. 分析并发进程的关键活动，划定临界区所放代码；
2. 设置互斥信号量 mutex，初值为 1；
3. 在临界区之前执行 P (mutex)

4. 在临界区之后执行V (mutex)

注意

- 1. 对不同的临界资源需要设置不同的互斥信号量
- 2. P, V 操作必须成对出现
 - 缺少 P (mutex) 就不能保证临界资源的互斥访问;
 - 缺少 V (mutex) 会导致资源永不被唤醒;

补充: 优先级继承协议

- 有正在访问资源的进程获得需要访问它的更高优先级进程的优先级, 直到它们用完了有关资源为止。当它们用完时, 它们的优先级恢复到原始值

代码示例

```
//信号量机制实现互斥
semaphore mutex = 1; //初始化信号量

P1(){
    ...
    P(mutex);        //使用临界资源前需要加锁
    临界区代码段...
    V(mutex);        //使用临界资源后需要解锁
    ...
}

P2(){
    ...
    P(mutex);
    临界区代码段...
    V(mutex);
}
```

进程同步

用信号量实现进程同步

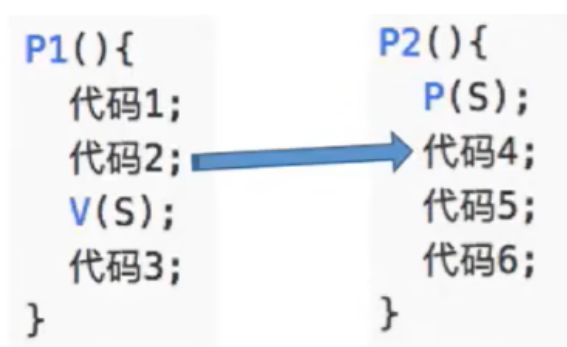
1. 分析什么地方需要实现**同步关系**, 即必须保证**一前一后**执行的两个操作;
2. 设置同步信号量 S, 初始为 0 ;
3. 在**前操作**之后执行 V (S);
4. 在**后操作**之前执行 P (S);

代码定义与实现

```
//信号量机制实现同步
semaphore S=0; //初始化同步信号量, 初始值为0
```

- 情况一: P 1 先执行 V(S)
 - 若先执行到 V (S)操作, 则 S++后 S=1;

- 之后当执行到 P(S)操作时，由于 S=1，表示有可用资源，会执行 S--，s 的值变回 0，P 2 进程不会执行 block 原语，而是继续往下执行代码 4；
- 情况二：P 2 先执行 P(S)
 - 若先执行到 P(S)操作，由于 S=0，s--后 S=-1，表示此时没有可用资源，因此 P 操作中会执行 block 原语，主动请求阻塞；
 - 之后当执行完代码 2，继而执行 V(S)操作，S++，使 S 变回 0，由于此时有进程在该信号量对应的阻塞队列中，因此会在 V 操作中执行 wakeup 原语，唤醒 P 2 进程。这样 P 2 就可以继续执行代码 4 了；
- 结论：
 - 通过设置 P、V 代码，P1 使用 V、P2 使用 P，保证了 P1 必须先使用、P2 必须后使用；
 - 实现了同步操作；
- 图示：



分析：信号量机制实现前驱关系

- 方法：
 1. 要为每一对前驱关系各设置一个同步信号量；
 2. 在“前操作”**之后**对相应的同步信号量执行 V 操作；
 3. 在“后操作”**之前**对相应的同步信号量执行 P 操作；
- 分析：同一代码、多个进程
 - 题目

【2020 统考真题】现有 5 个操作 A、B、C、D 和 E，操作 C 必须在 A 和 B 完成后执行，操作 E 必须在 C 和 D 完成后执行，请使用信号量的 wait()、signal()操作（P、V 操作）描述上述操作之间的同步关系，并说明所用信号量及其初值。
 - 解析：

```

进程()
{
    A;
    V(AC);
    B;
    V(BC);

    P(AC);
    P(BC);
    C;
    V(CE);

    D;
    V(DE);

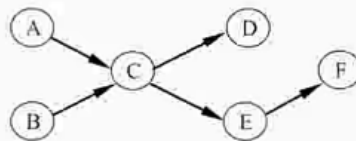
    P(CE);
    P(DE);
    E;
}

```

- 分析：不同代码、多个进程

- 题目：

【2022 统考真题】某进程的两个线程 T1 和 T2 并发执行 A、B、C、D、E 和 F 共 6 个操作，其中 T1 执行 A、E 和 F，T2 执行 B、C 和 D。下图表示上述 6 个操作的执行顺序所必须满足的约束：C 在 A 和 B 完成后执行，D 和 E 在 C 完成后执行，F 在 E 完成后执行。请使用信号量的 wait()、signal() 操作描述 T1 和 T2 之间的同步关系，并说明所用信号量的作用及其初值。



- 解析：

```

T1()
{
    //A动作无前驱，直接执行
    A
    V(AC)

    //E动作需要再C动作发生之后才可以进行，所以需要信号量

    P(CE)
    E

    //F进程因为就在T1内部，E执行完之后就直接进行F，因此不需要信号量
    F
}

T2()
{
    B
    P(AC)
    C
    V(CE)
    D
}

AC = 0
CE = 0

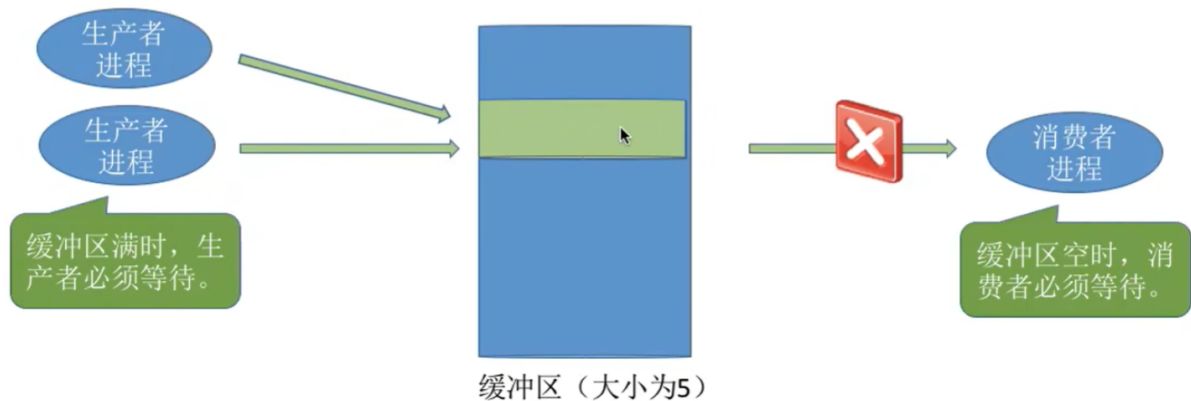
```

生产者-消费者问题

问题描述

概念：生产者-消费者问题

- 基本概念：
 - 1. 系统中有一组生产者进程和一组消费者进程，生产者进程每次生产一个产品放入缓冲区，消费者进程每次从缓冲区中取出一个产品并使用（注这里的“产品”理解为某种数据）；
 - 2. 生产者、消费者共享一个初始为空、大小为 n 的缓冲区
- 运作方式：
 - 1. 缓冲区**没满**时，生产者才能**把产品放入缓冲区**，否则必须等待；
 - 2. 缓冲区**不空**时，消费者才能从中**取出产品**，否则必须等待；
 - 3. 缓冲区是临界资源，各进程必须**互斥地**访问（比如如果两个生产者进程同时进行生产，那有可能出现部分区域被覆盖）；
- 图示：



步骤

- 1. 关系分析：找出题目中描述的各个进程，分析它们之间的**同步、互斥关系**；
 - 需要两种信号量：
 - 1. 互斥信号量；
 - `semaphore mutex=1; //互斥信号量，实现对缓冲区的互斥访问`
 - 2. 同步信号量；
 - `semaphore full=0; //同步信号量，表示产品的数量，也即非空缓冲区的数量`
 - `semaphore empty=n; //同步信号量，表示空闲缓冲区的数量，初始情况有n个空闲的缓冲区区域`
- 2. 整理思路：根据各进程的操作流程，确定 P、V 操作的大致顺序；
 - 有产品（V操作） `-(full)->` 消费者消费（P操作）；
 - 生产者生产（P操作） `<-(empty)-` 缓冲区未满（V操作）；
- 3. 设置信号量：根据题目条件确定信号量初值，互斥信号量初值一般为 1，同步信号量的初值要看对应资源的初始值是多少；

注意

- 需要注意实现互斥和实现同步的两个 **P 操作的先后顺序**；
- 实现互斥的 P 操作需要在实现同步的 P 操作之后，不然可能会**产生死锁**问题；
- 不过 V 操作的顺序可以调换；

代码实现

```
semaphore mutex = 1;    //互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n;    //同步信号量，表示空闲缓冲区的数量
semaphore full = 0;     //同步信号量，表示产品的数量，也即非空缓冲区的数量

//生产者
producer(){
    while(1){
        //生产一个产品;
        P(empty);        //消耗一个空闲缓冲区

        //上锁
        P(mutex);        //互斥信号量P操作
```

```

//把产品放入缓冲区;
V(mutex);          //互斥信号量V操作
//下锁

V(full);            //增加一个产品
}
}

```

```

//消费者
consumer(){
    while(1){
        P(full);          //消耗一个产品(非空缓冲区)
        P(mutex);
        //从缓冲区取出一个产品;
        V(mutex);
        V(empty);          //增加一个空闲缓冲区
        //使用产品;
    }
}

```

补充：改变相邻 P、V 操作的顺序 → 互斥的 P 操作再同步的 P 操作之前 → 会发生死锁

- 分析：
 - 若此时缓冲区内已经放满产品，则 $empty=0$ ， $full=n$ 。则生产者进程执行①使 $mutex$ 变为 0，再执行②，由于已没有空闲缓冲区，因此生产者被阻塞。由于生产者阻塞，因此切换回消费者进程。消费者进程执行③，由于 $mutex$ 为 0，即生产者还没释放对临界资源的“锁”，因此消费者也被阻塞；
- 结论：
 - 1. 实现互斥的 P 操作一定要在实现同步的 P 操作之后，**P 操作顺序不可交换**；
 - 2. V 操作不会导致进程阻塞，因此两个 **V 操作顺序可以交换**；

分析

真题：2009 年

- 题目：

【2009 统考真题】三个进程 P_1 、 P_2 、 P_3 互斥使用一个包含 N ($N > 0$) 个单元的缓冲区。 P_1 每次用 `produce()` 生成一个正整数并用 `put()` 送入缓冲区某一空单元； P_2 每次用 `getodd()` 从该缓冲区中取出一个奇数并用 `countodd()` 统计奇数个数； P_3 每次用 `geteven()` 从该缓冲区中取出一个偶数并用 `counteven()` 统计偶数个数。请用信号量机制实现这三个进程的同步与互斥活动，并说明所定义的信号量的含义（要求用伪代码描述）。
- 解析：

```

Semaphore mutex = 1;
Semaphore empty = N; // 空缓冲区单元个数信号量
Semaphore odd = 0; // 奇进程的同步信号量
Semaphore even = 0; // 偶进程的同步信号量

```

```
Process1()
```

```
{
```

```
// 生产资料不需要锁，可以无限制的进行
```

```
x = produce();
```

```
// 缓冲区中空位置的个数本身也是资源，需要同步信号量来同步其个数
```

```
// 若当前有空的缓冲区，消耗一个缓冲区资源
```

```
P(empty);
```

顺序
不可更改

```
// 将资源放入缓冲区是互斥操作，需要锁
```

```
P(mutex);
```

```
put();
```

```
V(mutex);
```

对正整数资源，进程2和进程3所需要的同步信号量不同，因此需要将其拆分，拆成两个信号量、使用if逻辑判断当前产生的资源属于哪一种，然后再分成生成信号量

```
if(x%2==1)
```

```
{
```

```
//生成奇数同步信号量
```

```
V(odd);
```

```
}
```

```
else
```

```
{
```

```
//生成偶数同步信号量
```

```
V(even);
```

```
}
```

```
}
```

两者顺序不可调转。若先P(mutex)、再P(odd)，则有可能出现以下情况：

- 进程1和进程2同时尝试进入缓冲区，此时若进程2（当前进程）先进入缓冲区，与此同时奇数资源odd又不够，则进程2会阻塞在这里。/但又因为进程2先获得了mutex，进程1因此无法获得mutex，进而无法生成odd信号量，进而导致这两个进程永久阻塞
- 同理进程3

```
Process2()
```

```
{
```

```
P(odd);
```

```
//缓冲区是互斥资源，需要互斥访问
```

```
P(mutex);
```

```
getodd();
```

```
V(mutex);
```

```
// 获得了一个正整数后，表示释放了一个缓冲区资源，因此需要V
```

```
V(empty);
```

```
countodd();
```

```
}
```

```

Process3()
{
    V(even);
    //缓冲区是互斥资源，需要互斥访问
    P(mutex);
    geteven();
    V(mutex);

    // 获得了一个正整数后，表示释放了一个缓冲区资源，因此需要V
    V(empty);

    counteven();
}

```

真题：2015 年

• 题目：

【2015 统考真题】有 A, B 两人通过信箱进行辩论，每个人都从自己的信箱中取得对方的问题。将答案和向对方提出的新问题组成一个邮件放入对方的邮箱中。假设 A 的信箱最多放 M 个邮件，B 的信箱最多放 N 个邮件。初始时 A 的信箱中有 x 个邮件 ($0 < x < M$)，B 的信箱中有 y 个邮件 ($0 < y < N$)。辩论者每取出一个邮件，邮件数减 1。A 和 B 两人的操作过程描述如下：

CoBegin

<pre> A{ while(TRUE){ 从 A 的信箱中取出一个邮件； 回答问题并提出一个新问题； 将新邮件放入 B 的信箱； } } </pre>	<pre> B{ while(TRUE){ 从 B 的信箱中取出一个邮件； 回答问题并提出一个新问题； 将新邮件放入 A 的信箱； } } </pre>
--	--

CoEnd

当信箱不为空时，辩论者才能从信箱中取邮件，否则等待。当信箱不满时，辩论者才能将新邮件放入信箱，否则等待。请添加必要的信号量和 P, V [或 wait(), signal()] 操作，

• 以实现上述过程的同步。要求写出完整的过程，并说明信号量的含义和初值。

• 解析：

```

Semaphore A_mutex = 1;
Semaphore B_mutex = 1;
Semaphore A_mail = x;
Semaphore B_mail = y;
Semaphore A_empty = M - x;
Semaphore B_empty = N - y;

A()                                B()
{                                  {
    while(TRUE)                    while(TRUE)
    {
        P(A_mail); // 消费一个A的邮件
        P(A_mutex);
        从A信箱取出一个邮件;
        V(A_mutex);
        P(A_empty); // 为A信箱生产一个空位

        回答问题并提出新问题;

        P(B_empty); // 消耗B信箱中的一个空位
        P(B_mutex);
        将新邮件放入B的信箱;
        V(B_mutex);
        V(B_mail); // 生产一个B邮件
    }
}

    {
        P(B_mail); // 消耗B信箱中的一个产品
        P(B_mutex);
        从B信箱取出一个邮件;
        V(B_mutex);
        V(B_mutex); // 为B信箱生产一个空位

        回答问题并提出新问题;

        P(A_empty); // 消耗A信箱中的一个空位
        P(A_mutex);
        将新邮件放入A的信箱;
        V(A_mutex);
        V(A_mail); // 生产一个A邮箱的邮件
    }
}

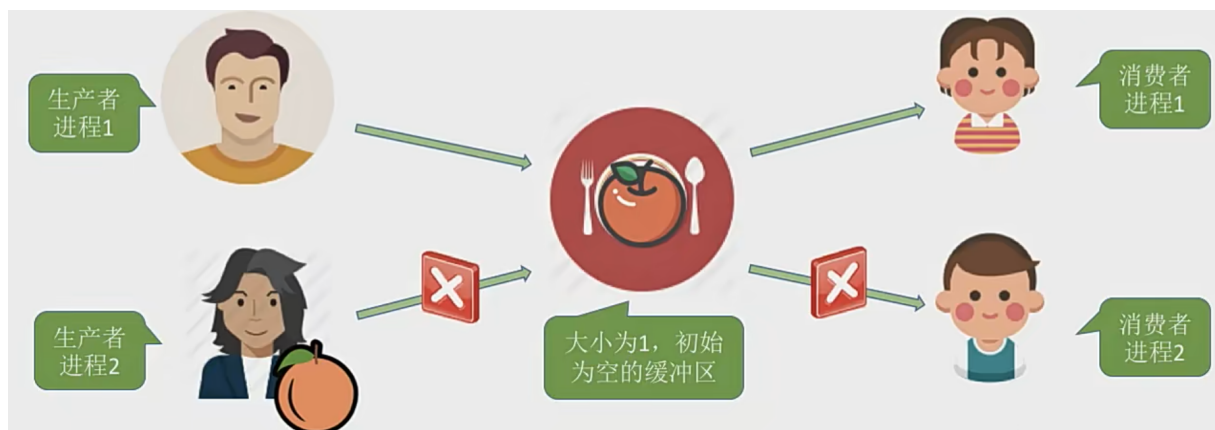
```

多生产者-多消费者问题

问题描述

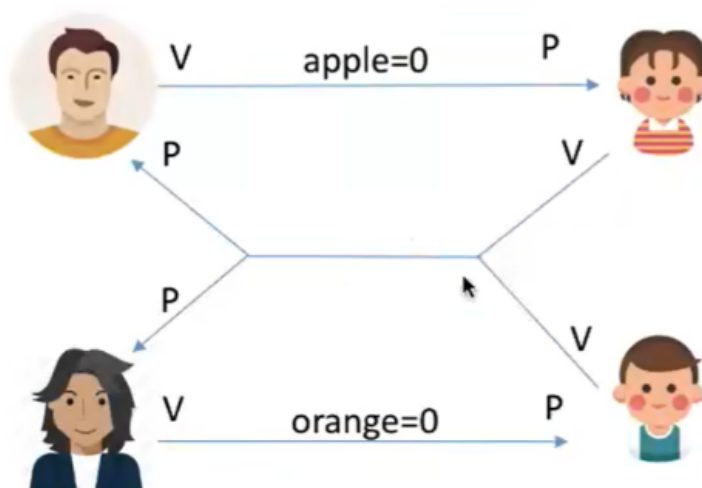
概念：问题描述

- 桌子上有一只盘子，每次只能向其中放入一个水果。爸爸专向盘子中放苹果，妈妈专向盘子中放橘子，儿子专等着吃盘子中的橘子，女儿专等着吃盘子中的苹果。只有盘子空时，爸爸或妈妈才可向盘子中放一个水果。仅当盘子中有自己需要的水果时，儿子或女儿可以从盘子中取出水果。用 PV 操作实现上述过程。
- 问题抽象：
 - 大小为 1，初始为空的缓冲区；
 - 分别有两类生产者（生产者 1 和生产者 2）以及对应的两个消费者；
- 图示：



分析

- 1. 关系分析。找出题目中描述的各个进程，分析它们之间的同步、互斥关系；
 - 互斥关系：对缓冲区 (盘子) 的访问要互斥地进行
 - 同步关系：(一前一后)
 - 1. 父亲将苹果放入盘子后，女儿才能取苹果；
 - 2. 母亲将橘子放入盘子后，儿子才能取橘子；
 - 3. 只有盘子为空时，父亲或母亲才能放入水果 (盘子为空这个事件可以由儿子或女儿触发，事件发生后才允许父亲或母亲放水果)；
- 2. 整理思路。根据各进程的操作流程确定 **P、V 操作的大致顺序**；
 - 互斥关系：`mutex=1`，对缓冲区 (盘子) 的访问要互斥地进行；
 - 同步关系：
 - 盘子：`plate = 1;`
 - 父亲生产苹果：`apple = 0;`
 - 母亲生产橘子：`orange = 0;`
- 补充：关于互斥信号量
 - 结论：在当前问题当中，即使不设置专门的互斥变量 `mutex`，也不会出现多个进程同时访问盘子的现象；
 - 原因：本题中的缓冲区大小为 1，在任何时刻，`apple`、`orange`、`plate` 三个同步信号量中最多只有一个是 1。因此在任何时刻，最多只有一个进程的 P 操作不会被阻塞，并顺利地进入临界区；
- 图示：



代码描述

信号量初始化

```
//信号量
semaphore mutex = 1;      //实现互斥访问盘子(缓冲区)
semaphore apple = 0;      //盘子中有几个苹果
semaphore orange = 0;     //盘子中有几个橘子
semaphore plate = 1;      //盘子中还可以放多少个水果
```

生产、消费

```
dad(){
    while(1){
        //准备一个苹果
        P(plate);
        P(mutex);          //加锁
        //把苹果放入盘子
        V(mutex);          //解锁
        V(apple);
    }
}

mom(){
    while(1){
        //准备一个橘子
        P(plate);
        P(mutex);
        //把橘子放入盘子
        V(mutex);
        V(orange);
    }
}

daughter(){
    while(1){
        //从盘子中取出苹果
        P(apple);
        P(mutex);
        //从盘中取出苹果
        V(mutex);
        V(plate);
        //吃掉苹果
    }
}

son(){
    while(1){
        P(orange);
        P(mutex);
        //从盘中取出橘子
        V(mutex);
        V(plate);
    }
}
```

```
}  
}
```

分析

真题：2014 年

- 题目：

【2014 统考真题】系统中有多个生产者进程和多个消费者进程，共享一个能存放 1000 件产品的环形缓冲区（初始为空）。缓冲区未空时，生产者进程可以放入其生产的一件产品，否则等待；缓冲区未空时，消费者进程可从缓冲区取走一件产品，否则等待。要求一个消费者进程从缓冲区连续取出 10 件产品后，其他消费者进程才可以取产品。请使用信号量 P、V（wait()、signal()）操作实现进程间的互斥与同步，要求写出完整的过程，并说明所用信号量的含义和初值。

- 分析：

- 总结：只有消费者进程间才具有写-写的互斥关系；
- 因此需要为此互斥关系建立互斥锁；
- 产品对于消费者和生产者仅有同步关系，因此只需要 P、V 产品数量即可；
- 缓冲区对于所有生产者、消费者都有互斥关系，因此上锁；

- 解析：

```
Semaphore mutex = 1; // 缓冲区的互斥锁  
Semaphore tenProductMutex; // 取十个产品的互斥锁  
Semaphore empty = 1000; // 缓冲区有1000个空位  
Semaphore product = 0; // 缓冲区的产品
```

```
生产者()
{
    // 题目中未明确说出是否只执行一次，因此按照进程一直运行来处理，即执行多次
    while(TRUE)
    {
        // 生产产品不需要等待
        生产一件产品；

        P(empty); // 消耗一个缓冲区的空位
        P(mutex);
        放入环形缓冲区；
        V(mutex);
        V(product); // 生产一个产品
    }
}
```

获取笔记持续更新可加

★ 注意：这里不能使用if(product>10)来进行访问限制

• 因为此题目是多生产者-多消费者模型，同一时刻可能会有多个消费者同时想进行消费。因此若使用if，则可能会出现多个消费者同时进入if语句，并执行for循环的内容。因此需要使用互斥锁把取十个产品的行为进行互斥

消费者()

{

- P(tenProductMutex); // 一次取出十个产品是一个互斥行为，
需要互斥的访问，因此需要给“取十个产品”这个动作上互斥锁

for(int i =0;i<10;i++)

{

P(product); // 消费一个产品

P(mutex);

从缓冲区取一个产品;

V(mutex);

V(empty); // 释放一个空位

}

-> V(tenProductMutex);

}

总结

• 总结

- 在生产者-消费者问题中，如果缓冲区大小为 1，那么有可能不需要设置互斥信号量就可以实现互斥访问缓冲区的功能；
- 当然这不是绝对的，要具体问题具体分析

• 建议

- 在考试中如果来不及仔细分析，可以加上互斥信号量，保证各进程一定会互斥访问缓冲区。
- 但是需要注意的是，实现互斥的 P 操作一定要在实现同步的 P 操作之后，否则可能引起死锁

PV 操作题目的解题思路

• 第一步：

- 分析有几类进程；
- 每个进程对应一个函数；

• 第二步：

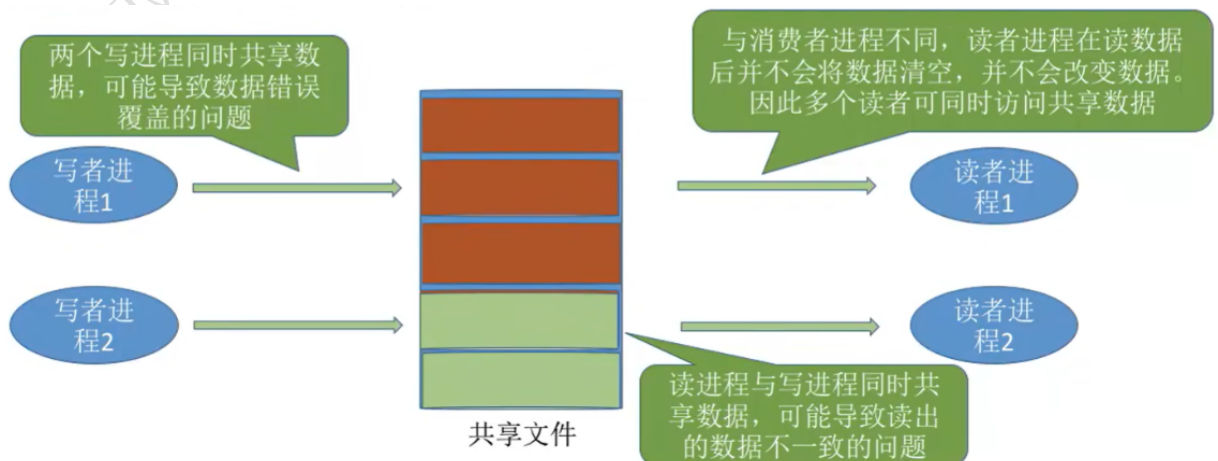
- 在每个进程的內部，用中文描述动作；
- 只做一次：不加 `while`
- 做很多次：加 `while`
- 第三步：
 - 分析每一次动作之前，需要 P 什么；
 - 然后根据有 P 必有 V 的原则，每写出一个 P、就写一个对应的 V；
- 第四步：
 - 所有 PV 操作写好后，定义信号量；
- 第五步：
 - 注意连续 P 的地方是否有死锁；
 - 通常，同步的 P 操作需要在互斥的 P 操作之外；
 - 产生死锁时：可尝试调整 P 顺序；
 - 若某信号量 PV 操作总连续出现，中间没有夹其他 P，则不可能因此产生死锁；

读者 - 写者问题

问题描述

概念：读者 - 写者问题

- 信息：
 - 有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误；
- 要求：
 1. 允许多个读者可以同时同时对文件执行读操作
 2. 只允许一个写者往文件中写信息
 3. 任一写者在完成写操作之前不允许其他读者或写者工作
 4. 写者执行写操作前，应让已有的读者和写者全部退出。
- 图示：



代码实现

信号量

```
semaphore rw = 1;    //用于实现对文件的互斥访问。表示当前是否有进程在访问共享文件
int count = 0;       //记录当前有几个读进程在访问文件
semaphore mutex = 1; //用于保证对count变量的互斥访问
semaphore w = 1;     //用于实现写优先
```

1 生产者、n 消费者

```
writer(){
    while(1){
        P(w);
        P(rw);    //写之前加锁
        写文件...
        V(rw);    //写之后解锁
        V(w);
    }
}
```

```
reader(){
    while(1){
        P(w);
        P(mutex);    //各进程互斥访问count
        if(count==0)
            P(rw);    //第一个读进程负责加锁
        count++;    //访问文件的读进程数+1
        V(mutex);
        V(w);
        读文件...
        P(mutex);    //各读进程互斥访问count
        count--;    //访问文件的读进程数-1
        if(count==0)
            V(rw);    //最后一个读进程负责解锁
        V(mutex);
    }
}
```

分析

分析：真题 - 2016年

- 题目：

<pre>// 进程P1 int x=0; Thread1 () { int a; a=1; x += 1; } Thread2 () { int a; a=2; x += 2; }</pre>	<pre>// 进程P2 int x=0; Thread3 () { int a; a=x; x += 3; } Thread4 () { int b; b=x; x += 4; }</pre>
---	---

下列选项中，需要互斥执行的操作是 ()。

- A. $a=1$ 与 $a=2$
- B. $a=x$ 与 $b=x$
- C. $x+=1$ 与 $x+=2$
- D. $x+=1$ 与 $x+=3$

• 分析：

- A：资源 a 分别属于线程 1 和线程 2，它们是各自的局部变量，不涉及互斥问题；
- B：线程 3 和 4 中的 $a=x$ 和 $b=x$ 都使用了临界资源，但对同一个临界资源的 读 - 读 进程之间不产生互斥关系；对同一个临界资源的 读 - 写 以及 写 - 写 才会产生互斥关系；因此 B 中不会产生互斥；
- C：在进程 P1 中， x 属于临界资源，进程 1 和 2 产生了 写 - 写 关系，会导致互斥关系，因此需要互斥操作；
- D：进程资源不共享；

分析：真题：2017年

• 题目：

【2017 统考真题】某进程中有 3 个并发执行的线程 thread1, thread2 和 thread3, 其伪代码如下所示。

<pre>// 复数的结构类型定义 typedef struct { float a; float b; } cnum; cnum x, y, z; // 全局变量 // 计算两个复数之和 cnum add(cnum p, cnum q) { cnum s; s.a = p.a+q.a; s.b = p.b+q.b; return s; }</pre>	<pre>thread1 { cnum w; w = add(x, y); ... } thread2 { cnum w; w = add(y, z); ... }</pre>	<pre>thread3 { cnum w; w.a = 1; w.b = 1; z = add(z, w); y = add(y, w); ... }</pre>
--	---	--

请添加必要的信号量和 P, V [或 wait(), signal()] 操作, 要求确保线程互斥访问临界资源,

- 并且最大限度地并发执行。

- 分析:

分析其互斥关系:

- 已知当前存在三个全局变量, 即x、y、z为临界资源, 需要分别为其分析互斥关系;
- x: 尽管x是临界资源, 但其只存在于thread1当中, 并且thread1只有一个, 因此其不存在互斥关系;
- z: z是互斥资源, 其同时存在在thread2和thread3中, 因此需要为z关于thread2和3的互斥关系建立一个锁;
 - thread2中对z进行读操作;
 - thread3中对z进行写操作;
 - 因此2、3产生读写互斥关系;
- y:
 - y是互斥资源, 并且thread1中会读y, thread2中会读y, thread3中会读写y
 - 因为在线程1和2中, y被传入add函数中, 而add函数值读取了y的值, 并没有改变y的值, 因此是读;
 - 因为在线程3中, 除了将y传入add中对其进行访问外, 还对其进行了修改, 所以是读写y;
 - 两个读操作不会发生互斥关系, 因此 线程1和线程2 无需定义互斥关系;
 - 读写操作 和 写写操作 之间会产生互斥关系, 且线程1和线程3为读、写关系, 线程2和线程3为读、写关系, 因此分别为其定义一个互斥锁;

- 解析:

```
Semaphore z_23_mutex = 1;  
Semaphore y_13_mutex = 1;  
Semaphore y_23_mutex = 1;
```

thread1	thread2	thread3
{	{	{
cnum w;	cnum w;	cnum w;
		w.a = 1;
		w.b = 1;
P(y_13_mutex);	P(y_23_mutex);	
w = add(x,y);	P(z_23_mutex);	
V(y_13_mutex);	w = add(y,z);	P(z_23_mutex);
}	V(z_23_mutex);	z = add(z,w);
	V(y_23_mutex);	V(z_23_mutex);
	}	
		P(y_13_mutex);
		P(y_23_mutex);
		y = add(y,w);
		V(y_23_mutex);
		V(y_13_mutex);
		}

- 总结:

★ 总结：互斥锁是建立在互斥关系上的，而非简单的建立在临界资源上。当同一个临界资源对应多个互斥关系时，需要为其定义多个互斥锁，用于提升并发度；

★ 总结：

- 对同一个临界资源的 读 - 读 进程之间不产生互斥关系；
- 对同一个临界资源的 读 - 写 以及 写 - 写 产生互斥关系；

★ 总结：

- 互斥关系分析的关键，在于分析临界资源的访问限制关系：
 - 缓冲区：只能一个读，或者一个写
 - 变量：只能一个读，或者一个写，一个一个 读-读，不允许 读-写 或 写-写
- 同步关系分析的关键，在于分析资源的前后依赖关系
 - 前驱后继问题：被依赖项要V，依赖者要P；
 - 理发师问题：分析服务-客户，客户数量-理发 之间的不同同步关系；

哲学家进餐问题

真题：2019

• 题目：

- 有 n ($n > 3$) 名哲学家围坐在一张圆桌边，每名哲学家交替地就餐和思考。在圆桌中心有 m ($m > 1$) 个碗，每两名哲学家之间有一根筷子。每名哲学家必须取到一个碗和两侧的筷子后，才能就餐，进餐完毕，将碗和筷子放回原位，并继续思考。为使尽可能多的哲学家同时就餐，且防止出现死锁现象，请使用信号量的 PV 操作描述上述过程中的互斥与同步，并说明所用信号量及初值的含义；

• 解析：

```
// 第一步：定义大锁
// 大锁是互斥信号量，用于保证取资源过程的互斥进行

Semaphore Lock = 1;

// 第二步：定义资源
// 当前问题中存在两类资源：碗和筷子
// 碗资源的数量即为 m
// 筷子资源比较特殊，因为每个哲学家只能拿其左右手两边的筷子，因此需要把其定义为数组
int Wan = m;
int Kuaizi[n];

for(int i = 0; i < n; i++)
```

```

{
    Kuaizi[i] = 1;
}

CoBegin
{
    // i为当前的进程序号
    Fun(int i)
    {
        while(TRUE)
        {
            // 第三步：定义大锁
            P(Lock);

            // 在一把大的互斥锁里面，判断碗筷数量是否充足；
            // 因为是在大锁里面，所以可以不用使用信号量来同步资源的同步关系，直接使用int变
            量来同步数量关系
            if( Wan>=1 && Kuaizi[i%n] == 1 && Kuaizi[(i+1)%n] == 1 )
            {
                Wan--;
                Kuaizi[i%n] = 0;
                Kuaizi[(i+1)%n] = 0;
                取走左右手两根筷子和一个碗；

                // 当前资源充足，释放锁，跳出循环
                V(Lock);
                break;
            }

            // 注意大锁的释放位置是循环内的最后一行；
            // 因此执行一次循环的检查后应当运行其他进程也可以检查当前资源数量是否充足
            // 若当前没有取到资源，则会释放锁，并且不进行break，继续执行循环
            V(Lock);
        }

        // 第四步：干饭
        哲学家干饭；

        // 第五步：释放资源
        // 注意释放资源的时候也要上锁
        P(Lock);
        Wan++;
        Kuaizi[i%n] = 1;
        Kuaizi[(i+1)%n] = 1;
        V(Lock);
    }
}
}

```

管程

基础概念

引入：为什么需要管程

- 信号量机制存在的问题：编写程序困难、易出错；
- 1973 年，Brinch Hansen 首次在程序设计语言 Pascal 中引入了“管程”成分，作为一种高级的同步机制；

概念：管程

- 基本概念：
 - 管程提供一组由程序员定义的、在管程内互斥的操作；
 - 管程类型也包括一组变量，用于定义这一类型的实例状态，也包括操作这些变量的函数实现；
 - 管程结构确保每次只有一个进程在管程内处于活动状态。因此，程序员不需要明确编写同步约束；
- 组成：
 - 管程是一种特殊的软件模块，由这些部分组成：
 1. 局部与管程的共享数据结构说明；
 2. 对该数据结构进行操作的一组过程(函数)；
 3. 对局部于管程的共享数据设置初始值的语句；
 4. 管程有一个名字；
- 特点：
 - 1. 管程由编程语言支持：
 - 既可以实现进程的互斥，也可以实现进程的同步；
 - 2. 管程具有封装性：
 - 管程中的过程只能使用管程中的数据结构，同时管程中的数据结构也只能在管程中使用。因此一个进程只有通过调用管程内的过程才能进入管程访问共享数据；
 - 3. 管程具有互斥性：
 - 在任何时候，一个管程中只能有一个活动进程，因此每次仅允许一个进程在管程内执行某个内部过程；
- 注意：
 - 管程不等于管道

分析：管道中的条件变量

- 问题：若 x 是管程内的条件变量，则当进程执行 `x.wait()` 时所做的工作是？
 - A. 实现对变量 x 的互斥访问
 - B. 唤醒一个在 x 上阻塞的进程
 - C. 根据 x 的值判断该进程是否进入阻塞态
 - D. 阻塞该进程，并将之插入 x 的阻塞队列中
- 解析：
 - A：管程由代表共享资源的数据结构以及对该共享数据结构实施操作的一组过程所组成，条件变量是管程内包含的一种全局数据结构，其作用类似于信号量机制中的“信号量”，都

用于实现进程同步，而 `x.wait()` 操作与变量 `x` 的互斥访问无关，A 错误；

- B：在同一时刻，管程中只能有一个进程在执行，`x.signal()` 用于唤醒一个在 `x` 上阻塞的进程 B 错误；
- C、D：当进程执行 `x.wait()` 时，该进程会阻塞，并挂到条件变量 `x` 对应的阻塞队列上。这样，管程的使用权被释放，接下来可以有另一个进程进入管程。直到另一个进程向条件变量执行了 `x.signal()` 操作，此时会唤醒 `x` 对应的阻塞队列的队首进程，此时此进程才会离开阻塞队列。C 错误，D 正确；

用管道解决生产者-消费者问题

```
monitor ProduceConsumer
condition full, empty;    //条件变量用来实现同步(排队)
int count = 0;    //缓冲区中的产品数
void insert(Item item){    //把产品item放入缓冲区
    if(count == N)
        wait(full);
    count++;
    insert_item(item);
    if(count == 1)
        signal(empty);
}
Item remove(){    //从缓冲区中取出一个产品
    if(count==0)
        wait(empty);
    count--;
    if(count==N-1)
        signal(full);
    return remove_item();
}
end monitor;
```

```
//生产者进程
producer(){
    while(1){
        item = 生产一个产品;
        ProdecerConsumer.insert(item);
    }
}

//消费者进程
consumer(){
    while(1){
        item = ProdecerConsumer.remove();
        消费产品item;
    }
}
```

总结

目的：更方便地实现进程互斥和同步

1. 需要在管程中定义共享数据
2. 需要在管程中定义用于访问这些共享数据的**入口**(函数)
3. 只有通过这些特定的**入口**才能访问共享数据
4. 管程中有很多**入口**，但是每次只能开放其中一个**入口**，并且只能让一个进程或线程进入 (互斥特性由编译器负责实现，程序员不用关心)
5. 可在管程中设置条件变量及其等待/唤醒操作以解决同步问题。
 - 可以让一个进程或线程在条件变量上等待 (此时，该进程应先释放管程的使用权，也就是让出**入口**)
 - 可以通过唤醒操作将等待在条件变量上的进程或线程唤醒

注意

- 程序员可以用某种特殊的语法定义一个管程
- 之后其他程序员就可以使用这个管程提供的特定**入口**很方便地使用实现进程同步/互斥 (封装思想)

获取笔记持续更新可加微信: Wandu2277