4. 同步与互斥

4.1. 进程同步的基本概念

1️⃣背景：多个进程对数据并发访问会导致数据不一致(如共享变量修改)，所以要保证并发进程按顺序执行

2️⃣进程类型：协作进程、独立进程

3️⃣进程间的制约关系

1. 间接互相制约关系(互斥)：进程-资源-进程。同种进程互斥共享某种系统资源，如打印机
2. 直接互相制约关系(同步)：进程-进程。一进程收到另一进程的必要信息，才能继续运行

PS: 一般同类进程互斥，不同进程同步

4️⃣进程间的交互关系

1. 互斥：多个进程不能同时使用同一个资源
2. 同步：异步执行过程中，相合作的进程在关键点互相等待/交换信息
3. 死锁：多个进程互不相让，都得不到足够的资源
4. 饥饿：资源被其他进程轮奸，该进程一直得不到它

4.2. 临界资源与临界区

1️⃣临界资源：只能同时给一个进程使用的资源，比如打印机

2️⃣临界区：进程中访问**临界资源**的一段代码，每进程都有一段临界区代码(可不同)，在该区中进程可修改共享变量等，一个进程在其临界区时，同类进程都不可以进入临界区

3️⃣访问临界资源的过程：

1. 进入区：检查**可否进入**临界区的一段代码，若可以则设置相应“正在访问临界区”标志
2. 临界区：
3. 退出区：属于临界区，清除“正在访问临界区”标志
4. 剩余区：代码其余部分

4️⃣进程对临界区互斥访问的实现

1. 空闲则入：临界区无进程时，进程请求加入临界区就进吧
2. 忙则等待：临界区有进程了，禁止其他请求进入临界区的请求
3. 有限等待：进程请求访问临界资源后，就应该在有限时间内加入临界区，不死等
4. 让权等待：一个进程不能进入自己的临界区时，释放处理器阻塞自己

4.3. 互斥的实现方式

4.3.1. 软件方法(困难复杂)

1️⃣算法1：两个进程P0，P1使用公共变量turn来实现交替进入临界区

```
int turn = 0;
void processP0()        //进程P0
```

```
{
 while(true)            //无限循环，表示进程的持续执行
 {
     while(turn != 0); //不为0就卡在这，直到turn为0，P0进入临界区
     /*进程P0的代码区*/
     turn = 1;          //退出区
     /*进程P0其它代码*/
 }
}
void processP1()        //进程P1
{
 while(true)            //无限循环，表示进程的持续执行
 {
     while(turn != 1); //不为1就卡在这，直到turn为1，P1进入临界区
     /*进程P1的代码区*/
     turn = 0;          //退出区
     /*进程P0其它代码*/
 }
}
```

1. 强制轮流进入临界区，没有考虑进程的实际需要
2. 不保证空闲则入：一个进程处于非临界区(即便临界区空闲)，另一个进程也进不去临界区

   例如：P0执行完后，置turn=1他自己就进不去了，而P1此时也没请求进入，临界区就空了

2️⃣算法2：设置标志数组flag[]表示进程是否在临界区中执行

```
/*每个进程访问临界资源前，检查另一个进程是否在临界区中
*若不在，则修改本进程的临界区标志为真并进入临界区
*退出时，在退出区修改本进程临界区标志为假*/
bool flag[2] = {0,0};      //初始均为假
void processP0()           //进程P0
{
    while(true)            //无限循环，表示进程的持续执行
    {
        while(flag[1]);    //不为0就卡这，直到falg[1]=0(P1退出临界区了)，P0进入临界区
        flag[0] = 1;       //声明我P0进程在临界区
        /*进程P0的代码区*/
        flag[0] = 0;       //退出区
        /*进程P0其它代码*/
    }
}
void processP0()           //进程P1
{
    while(true)            //无限循环，表示进程的持续执行
    {
        while(flag[0]);    //不为0就卡这，直到falg[0]=0(临界区没东西了)，P1进入临界区
        flag[1] = 1;
        /*进程P1的代码区*/
        flag[1] = 0;       //退出区
        /*进程P1其它代码*/
```

```
    }
}
```

1. 此算法保证空闲让进，不保证忙则等待
2. 会出现死锁：想象如下场景

P0置flag[0]=0退出→P0执行剩余代码(此时: P1进入临界区→快速执行完后置flag[1]=0)→此时flag数组中两项都为0→两个进程都要进入临界区→都进不了，死锁

3️⃣算法3：设标志组flag(进程是否希望进入临界区)

```
bool flag[2] = {0, 0};          // 初始化为假，表示两个进程初始时都不希望进入临界区
void processP0()                // 进程P0
{
    while(true)                 // 无限循环，表示进程的持续执行
    {
        flag[0] = 1;            // 声明进程P0希望进入临界区
        while(flag[1]);         // 如果进程P1也希望进入，则等待
        /* 进程P0的代码区*/
        flag[0] = 0;            // 进程P0不再希望进入临界区
        /* 进程P0其它代码*/
    }
}
void processP1()                // 进程P1
{
    while(true)                 // 无限循环，表示进程的持续执行
    {
        flag[1] = 1;            // 声明进程P1希望进入临界区
        while(flag[0]);         // 如果进程P0也希望进入，则等待
        /* 进程P1的代码区*/
        flag[1] = 0;            // 进程P1不再希望进入临界区
        /* 进程P1其它代码*/
    }
}
```

1. 不满足有限等待：一个进程一直执行，另一个就一直无法进入
2. 防止了两进程同时进入临界区，但可能两个进程都进不了临界区(都表示不希望进入)

4️⃣在算法3️⃣基础上加上一个turn变量，turn=0/1表示允许P0/P1进程访问临界区

```
bool flag[2] = {0, 0};                  // 初始化为假，表示两个进程初始时都不希望进入临界区
int turn = 0;                           // 初始时，让进程P0先进入
void processP0()                        // 进程P0
{
    while(true)                         // 无限循环，表示进程的持续执行
    {
        flag[0] = 1;                    // 声明进程P0希望进入临界区
        turn = 1;                       // 此时P0还没进去，让进程P1还有机会进入
        while(flag[1] && turn == 1); // 如果进程P1也希望进入且turn为P1，则等待
        /* 进程P0的代码区 */
```

```
        flag[0] = 0;                    // 进程P0退出临界区
        /* 进程P0其它代码 */
    }
}

void processP1()                        // 进程P1
{
    while(true)                         // 无限循环，表示进程的持续执行
    {
        flag[1] = 1;                    // 声明进程P1希望进入临界区
        turn = 0;                       // 此时P1还没进去，让进程P0还有机会进入
        while(flag[0] && turn == 0); // 如果进程P0也希望进入且turn为P0，则
等待
        /* 进程P1的代码区 */
        flag[1] = 0;                    // 进程P1不再希望进入临界区
        /* 进程P1其它代码 */
    }
}
```

4.3.2. 硬件方法(当前主流)

1️⃣主要思想：通过硬件指令/中断屏蔽，确保关键代码段在不被打断的情况下连续执行，从而保障进程间的互斥访问

2️⃣优势：适用广(进程数随意/处理器数随意)，简单(容易验证正确性)，支持多临界区

3️⃣缺点：不能让权等待(只能忙等耗费CPU时间)，看你饥饿(有的进程可能一直选不上到临界区)

PS—让全等待：进程抛弃CPU资源等待，区别于不放弃CPU的忙则等待

4.4. 信号量semaphore(Dijkstra提出的同步机构)

之前的互斥算法都是平等进程间的协商，信号量使得有一个更高地位的进程管理者来分配资源

4.4.1. 信号量及同步原语

1️⃣信号量是一个二元组 [s,q] ——且初值非负，q为初始为空的队列

1. s是信号量的值：初值非负表示可用资源数，其值只能被P(wait)操作/V(signal)操作改变
2. q是初始为空的队列：就是该信号量的进程**等待队列**

2️⃣P/V操作：申请/释放资源，二者成对出现，被视为不可分割原子操作

1. 原始版本：会忙则等待，s>0表示可用资源数，s不可为负数

```
P(S);
{
    if(S<=0);     //不做仍和操作
    if(S>0) S--; //一个资源被申请走了，所以信号量的值减少
}
V(S){S++;}        //一个资源被释放了，信号量增加
```

2. 改进版：不会忙则等待，s>0表示可用资源数，s<0表示请求该资源而阻塞的进程数(绝对值)

```
/*详见记录型信号量*/
```

4.4.2. 信号量的分类

1️⃣整型信号量：就是上面所提到的(int)s，只有初始化/p/v操作可以改变s。存在忙等，因为P操作后若无资源进程会持续测试s直到其有资源了

2️⃣记录型信号量：int s + 链表q(链接了等待该资源的进程)。P操作后无资源则进程自我阻塞放弃CPU，插入等待链表(让权等待)，V操作时唤醒链表中第一个程序

```
typedef struct
{
 int value;
 struct process *L;
}semaphore;

semaphore s;

P(s)
{
 s.value--;      //可用资源数-1，或者等待资源进程数+1
 if(s.value<0)  //如果已经没有可用资源了
 {/*将该进程加入到s.L中去，然后阻塞*/}
}
V(s)
{
 s.value++;      //可用资源数+1，或者等待资源进程数-1
 if(s.value<=0) //如果此刻正在有进程等待这个资源
 {/*将该进程从s.L中移除，然后唤醒*/}
}
```

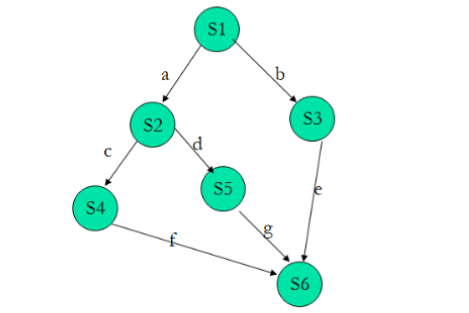
4.4.3. 信号量的应用

PV操作成对出现，同步时PV不在一个进程，互斥时PV在同一进程

1️⃣同步进程：P1(含S1),P2(含S2)两进程并发，S1要在S2之前执行

```
int N=0;                  //信号量，初值为0
P1(){...;S1;V(N);...}  //执行S1，后通过V操作增加信号量N的值，这表示S1已执行完毕
P2(){...;P(N);S2;...}  //P操作试图减少信号量N的值但被阻塞，S1执行完N增加后才执行S2
```

举例：S1生成S2/S3，S2继续生成S4/S5，最后S3/S4/S5一起生成S6。如下总结就是入为P出为V



定义信号量:a, b, c, d, e, f, g ,初值均为0

```
int a=b=c=d=e=f=g=0;
begin
  begin
     生成数据 S1;
     V(a);
     V(b);
  end
  begin
     P(a);
     生成数据 S2;
     V(c);
     V(d);
  end
  begin
      P(b);
      生成数据S3;
     V(e);
  end
```

```
begin
     P(c);
     生成数据 S4;
     V(f);
end
 begin
     P(d);
     生成数据 S5;
     V(g);
end
 begin
      P(e);
      P(f);
      P(g);
      生成数据S6;
end
end
```

2️⃣进程互斥：P1,P2只有一个进程可以进入自己的临界区

```
int N=1; //互斥信号量，初值为1，只有一个进程可以获得资源
P1(){...;P(N);P1的临界区代码;V(N);...} //临界区代码置于P/V原语之间
P2(){...;P(N);P2的临界区代码;V(N);...}
```

反过来想，如果P12同时进入临界区N就会变成负数，不可能的，所以只能进入一个

🔮相连两个P操作，同步P应该在互斥P之前(先检查是否满足同步再进入临界区)，但是二者的V操作顺序无关紧要

4.4.4. 信号量集：多个信号量的集合

1️⃣概述：用于处理复杂进程同步/互斥，允许进程在执行操作前同时检查多个信号量

2️⃣AND信号量集：

1. 功能：保证代码执行前获得多有临界资源(避免锁死)
2. 原子操作Swait：要么一次分配所有资源，要么一个都不分配，防止中间态而死锁
3. 操作Ssignal：释放所有资源，检查等待队列中是否有其他简称能因此获得全部资源

3️⃣一般信号量集：AND信号量集的扩展

1. 允许进程请求/释放不同数目的多种资源
2. `Swait(S1, t1, d1; ...; Sn, tn, dn)`：对每个信号量Si，都设置测试值ti+占用值di

```
Swait(S,m,n);                  //S每次申请m个资源，不够则阻塞，够则S减n
Swait(S1,m1,n1; S2,m2,n2)      //S1,S2每次申请m1,m2个资源，不够则阻塞，够则分别减n1,n2
Swait(S,1,1)                   //表示互斥信号量;
Swait(S,1,0)                   //作为一个可控开关
```

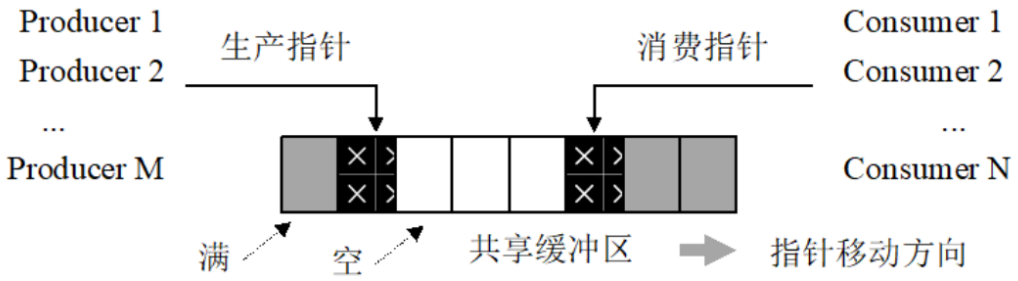
3. `Ssignal(S1, d1; ...; Sn, dn)`：对每个信号量，只设置占用值di

```
Ssignal(S,n)                   //释放n个资源S=S+n
Ssignal(S1,n1; S2,n2)          //S1=S1+n1，S2=S2+n2
```

4.5. 经典同步问题

4.5.1. 生产者-消费者问题

1️⃣问题描述：进程通过共享缓冲区交换数据，生产者写入/消费者读出，共享缓冲区共N个但一个时刻只有一个进程可操作缓冲区



2️⃣三种信号量激起初值

```
Semaphore full=0;          //表示当前已满的缓冲区数量
Semaphore empty=n;         //表示当前空的缓冲区数量
Semaphore mutex=1;         //互斥信号量，用于确保同一时刻只有一个进程对缓冲区进行操作
```

4️⃣P/V操作

```
P(empty) //检查是否有空余位来填充，若没有则阻塞直到有空位
V(empty) //填充完了一个空闲区
P(full)  //检查缓冲区是否有数据可以来取，若没有就阻塞直到有数据可读
V(full)  //
P(mutex) //取得互斥锁，告诉其他进程临界区我占了你们都别来
V(mutex) //释放互斥锁，高速
```

4️⃣实现：P操作顺序不可倒(先检查资源数目，再检查是否互斥)否则可能死锁

死锁情况：生产者先执行P(mutex)进入缓冲区→缓冲区满但执行P(empty)→没有空位然后阻塞

Producer	Consumer
P(empty); P(mutex); //进入区	P(full); P(mutex); //进入区
one unit --> buffer;	one unit <-- buffer;
V(mutex); V(full); //退出区	V(mutex); V(empty); //退出区

4.5.2. 读者-写者问题

多进程共享数据区，进程分为读者写者(读者只能读/写者只能写)，同一时刻可多读但最多一写

4.5.2.1. 读者优先：写者排队，读者插队，多读一写

1️⃣ 示例：(最左边表示最新到达的进程)

```
[W][W][W][W][W]-[数据区:R]--读者优先+R读完后-->[ ][W][W][W][W]-[数据区:W]
[R][W][R][R][W]-[数据区:W]--读者优先+W写完后-->[ ][ ][ ][W][W]-[数据区:RRR]
[R][W][R][R][W]-[数据区:R]--不可能出现这种情况
[R][W][W][W][W]-[数据区:R]--读者优先+R没读完-->[ ][W][W][W][W]-[数据区:RR]
```

2️⃣ 信号量

```
readcount=0//记录读者的数量
rmute=1    //保证读者进程对readcount的互斥访问(只有当前的唯一读者可修改readcount)
mutex=1    //标识允许写
```

3️⃣ 实现

```
reader()
{
while(1)
{
    P(remutex);                 //申请readcount的使用权
    if(readcount==0);P(mutex);//第一个读者，阻止写入
    readcount++;                //读者数量+1
    V(rmutex);                  //释放readcount的使用权，允许其他读者读
    /*读操作，完成读操作后：*/
    P(remutex);                 //申请readcount的使用权
    readcount--;                //读者数量-1
    if(readcount==0);V(mutex);//读者全部读完了，就允许写入
    V(rmutex);                  //释放readcount的使用权，允许其他读者读
}
}
write()
{
while(ture)
{
    P(mutex);                    //允许写
    /*写操作，完成写操作后：*/
    V(mutex);                    //释放写的许可
}
}
```

4.5.2.2. 平等策略：读者写者都要排队，不可插队，仍然多读一写

1️⃣ 示例：(最左边表示最新到达的进程)

```
[W][W][R][R][R]-[数据区:W]--平等策略+W写完后-->[ ][ ][ ][W][W]-[数据区:RRR]
[W][W][R][R][R]-[数据区:R]--这种情况不可能出现
[R][W][R][W][W]-[数据区:R]--平等策略+R读完后-->[ ][R][W][R][W]-[数据区:W]
```

2 信号量

```
readcount=0//记录读者的数量
rmute=1    //保证读者进程对readcount的互斥访问(只有当前的唯一读者可修改readcount)
mutex=1    //标识允许写
wmutex=1:  //是否存有在写/等着写的写者，存在的话就禁止新读者进入
```

3 实现

```
reader()
{
while(1)
{
    P(wmutex);                  //是否有写者存在(多读一写→不可能全是读者等),无则进
    P(remutex);                 //申请readcount的使用权
    if(readcount==0);P(mutex);//如果这是第一个读者，那么占据数据区阻止其他写着进入
    readcount++;                //读者数量+1
    V(rmutex);                  //释放readcount的使用权，允许其他读者用
    V(wmutex);                  //恢复wmutex
    /*读操作，完成读操作后：*/
    P(remutex);                 //申请readcount的使用权
    readcount--;                //读者数量-1
    if(readcount==0);V(mutex);//如果读者都没有了，就允许写者进入
    V(rmutex);                  //释放readcount的使用权，允许其他读者使用

}
}
write()
{
while(ture)
{
    P(wmutex);                  //检测是否有其他写者存在，无写者时进入
    P(mutex);                   //申请对数据区进行访问
    /*写操作，完成写操作后：*/
    V(mutex);                   //释放数据区，允许其他进程读写
    V(wmutex);                  //恢复wmutex
}
}
```

4.5.2.3. 写者优先：读者排队，写者插队，多读一写

1 示例：(最左边表示最新到达的进程)

```
[W][W][R][R][R]-[数据区:W]--写者优先+W写完后-->[ ][W][R][R][R]-[数据区:W]
[ ][W][R][R][R]-[数据区:W]--写者优先+W写完后-->[ ][ ][R][R][R]-[数据区:W]
[ ][ ][R][R][R]-[数据区:W]--写者优先+W写完后-->[ ][ ][ ][ ][ ]-[数据区:RRR]
```

队列中有读者写者时，先按顺序执行完所有写者，然后才开始执行读者

2 信号量

```
readcount=0 //记录读者的数量
writecount=0//记录写者的数量
rmutex=1    //保证读者进程对readcount的互斥访问(只有当前的唯一读者可修改readcount)
wmutex=1    //保证写者进程对writecount的互斥访问
mutex=1     //互斥访问数据区
readable=1  //表示当前是否有写者
```

3 实现

```
reader()
{
while(1)
{
   P(readable)                //检查是否存在写者，若没有则占用
   P(remutex);                //申请readcount的使用权
   if(readcount==0);P(mutex);//如果这是第一个读者，那么占据数据区阻止其他写着进入
   readcount++;               //读者数量+1
   V(rmutex);                 //释放readcount的使用权，允许其他读者使用
   V(readable);               //释放readable,允许其他读者或写者占用
   /*读操作，完成读操作后：*/
   P(remutex);                //申请readcount的使用权
   readcount--;               //读者数量-1
   if(readcount==0);V(mutex);//如果读者都没有了，就允许写者进入
   V(rmutex);                 //释放readcount的使用权，允许其他读者使用

}
}
write()
{
while(ture)
{
   P(wmutex);                 //检测是否有其他写者存在，无写者时进入
   if(writecount==0)
   P(readable);               //若为第一个写者，则阻止后续读者进入
   writecount++;              //写者数量加1
   V(wmutex);                 //释放wmutex,允许其他写者修改writecount
   P(mutex);                  //等当前正在操作的读者或写者完成后，占用数据区
```

```
    /*写操作，完成写操作后：*/
    V(mutex);                    //写完，释放数据区，允许其他进程读写
    P(wmutex);                   //占用wmutex,准备修改writecount
    writecount--;                //写者数量减1
    if(writecount==0)
    V(readable);                 //若为最后一个写者，则允许读者进入
    V(wmutex);                   //释放wmutex,允许其他写者修改writecount
}
}
```

4.5.3. 哲学家进餐问题

1. 问题描述：5人围桌而坐，两人间各一根筷子(临界资源)，每人有两个动作，进餐(先左后右拿起筷子)和思考(先左后右放回筷子)

2. 死锁：所有人都同时拿起左筷子，同时等待右筷子(但等不到)

3. 避免死锁：赶走一个人/同时拿起左右筷子/给人编号然后奇数先拿左边偶数先拿右边再奇偶交替

4.5.4. 理发师问题

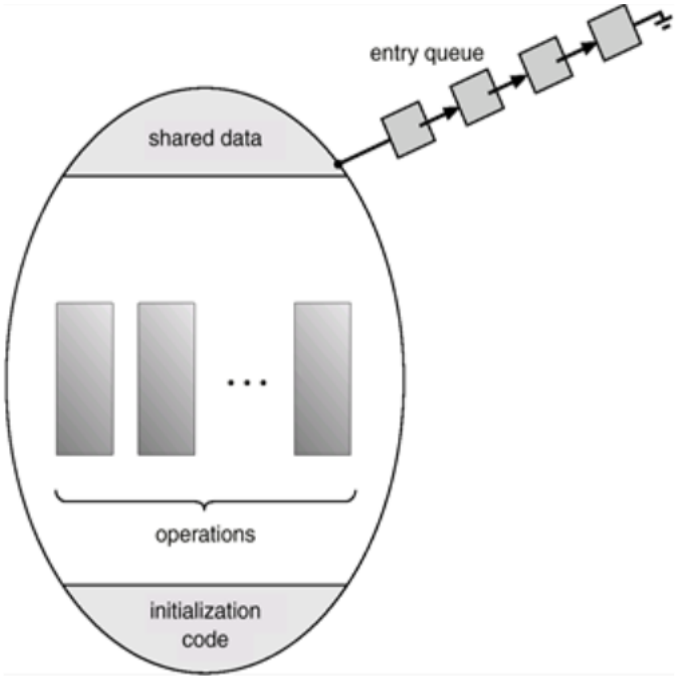
1. 描述：理发师+理发椅+等待椅，无顾客时理发师就会在理发椅上睡觉，顾客到达会唤醒理发师，理发时新顾客会在等待椅空闲/满时选择等待/离开

2. 问题核心：保证顾客对于理发师的互斥访问，确保等待队列满后顾客会走，服务完一个顾客后会服务下一个顾客

3. 解决方案：使用信号量来控制对临界资源的访问

    1. 5个信号量：记录等待顾客数，代表理发椅，代表等待凳子，两个记录理发师和顾客的同步
    2. 临界资源：凳子和理发椅

4.6. 管程：优于信号量的进程同步机构(了解即可)



1. 定义：关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块

2️⃣基本思想：把信号量+操作原语(共享变量+对共享变量的操作)封装在一个对象内部

3️⃣功能：集中管理进程中互斥访问的临界区，保证进程对于共享资源的互斥访问

4️⃣特点：局部于管程的数据只能被管程内部访问，进程只有通过调用进入管程才能访问共享数据，每次只允许一个进程调用管程

5️⃣管程的同步设施

1. 条件变量：仅能从管程内进行访问，用于表示进程不同的等待原因
2. wait和signal函数：进程调用wait后会被阻塞然后释放管程，调用signal唤醒在该条件变量上阻塞的进程