第6章 进程同步

第6章 进程同步

6.1 同步与互斥的概念

6.2 互斥的实现方法

6.3 信号量

6.4 经典的进程同步问题

6.5 信号量集机制

6.6 管程

第6章 进程同步

- 在多道程序系统中，由于资源共享或进程合作，使并发进程间形成间接相互制约和直接相互制约关系，使得程序执行呈现不确定性。
  - 执行的相对执行速度不可预测
  - 与进程调度有关，错误是间歇发生的 （数据不一致/结果不可再现，）
- 进程同步的主要任务是协调这两种制约关系，使并发执行的进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。

与时间有关的错误例子1：结果不唯一

- 现有订票系统，x为存储某班次飞机剩余票数的存储区域

- A：

```
1. R1=x;
2. if (R1>=1) {
3.     R1--;
4.     x=R1;
5.     {出票}
6. }
```

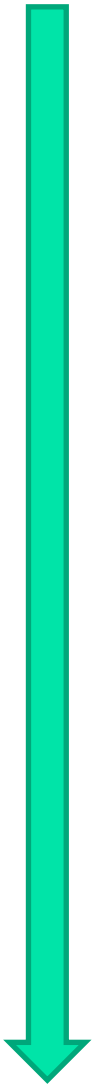
- B：

```
1. R2=x;
2. if (R2>=1){
3.     R2--;
4.     x=R2;
5.     {出票}
6. }
```

与时间有关的错误例子1：结果不唯一

进程A：

```
1.  R1=x;
2.  if (R1>=1) {
3.      R1--;
4.      x=R1;
5.      {出票}
6.  }
```



- 如果先执行A再执行B，则x值最终减少2，A、B输出的x不同。

进程B：

```
1.  R2=x;
2.  if (R2>=1){
3.      R2--;
4.      x=R2;
5.      {出票}
6.  }
```

与时间有关的错误例子1：结果不唯一

进程A：

1. R1=x;
2. if (R1>=1) {
3. R1--;
4. x=R1;
5. {出票}
6. }

进程B：

1. R2=x;
2. if (R2>=1){
3. R2--;
4. x=R2;
5. {出票}
6. }

- 若按顺序R1=x;R2=x;R1--;x=R1;R2=R2--;x=R2;则x值最终减少1，A、B输出的x是相同的。

与时间有关的错误例子1：结果不唯一

- A：

```
1. R1=x;
2. if (R1>=1) {
3.     R1--;
4.     x=R1;
5.     {出票}
6. }
```

- B：

```
1. R2=x;
2. if (R2>=1){
3.     R2--;
4.     x=R2;
5.     {出票}
6. }
```

- 这种错误称为“**与时间有关的错误**”，有些地方称为（race condition）
- 产生原因
  - 没有互斥使用共享变量，或者说没有在某进程进入时禁止另一个进程的进入。
  - 我们称以上例子中的错误为***结果不唯一***的错误

与时间有关的错误例子2：永远等待

- alloc和free为申请和归还资源的函数，进程在申请和释放时候调用它们。
- x为当前可用的总主存量，B为申请数量：

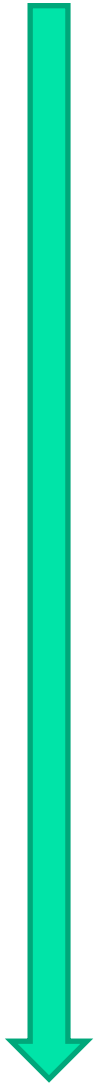
```
void alloc(int B)
{
    while(B>x)
    {将申请进程设置进入等待队列，等待主存资源}
    x=x-B;
    {修改主存分配表}
    {申请进程获得主存资源}
}
```

```
void free(int B)
{
    x=x+B;
    {修改主存分配表}
    {唤醒等待主存资源的进程}
}
```

与时间有关的错误例子2：永远等待

进程A：

```
void alloc(int B)
{
    while(B>x)
    {将申请进程设置进入等待队列，等待主存资源}
    x=x-B;
    {修改主存分配表}
    {申请进程获得主存资源}
}
```



进程B：

```
void free(int B)
{
    x=x+B;
    {修改主存分配表}
    {唤醒等待主存资源的进程}
}
```

与时间有关的错误例子2：永远等待

进程A：

```
void alloc(int B)
{
    while(B>x)

    {将申请进程设置进入等待
    队列，等待主存资源}
    x=x-B;
    {修改主存分配表}
    {申请进程获得主存资源}
}
```

进程B：

```
void free(int B)
{
    x=x+B;
    {修改主存分配表}
    {唤醒等待主存资源的进程}
}
```

- 如果某个时刻B>x，进程A调用alloc，在执行while之后，{申请...等待...}之前，进程B进入处理器，调用了free，则由于A还未来得及进入排队，则不会被唤醒，此后A进程进入等待队列，但不再有人唤醒A。

第6章 进程同步

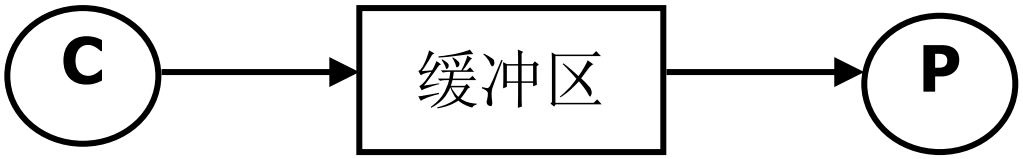
6.1 同步与互斥的概念

6.1 同步与互斥的概念

- 在多道程序系统中，进程之间的相互制约关系体现在如下两个方面：
  - 直接制约关系：合作进程之间产生的制约关系
    - 或称协作关系，需要同步（synchronization）
  - 间接制约关系：共享资源产生的制约关系。
    - 或称竞争关系，需要互斥（mutual exclusion）

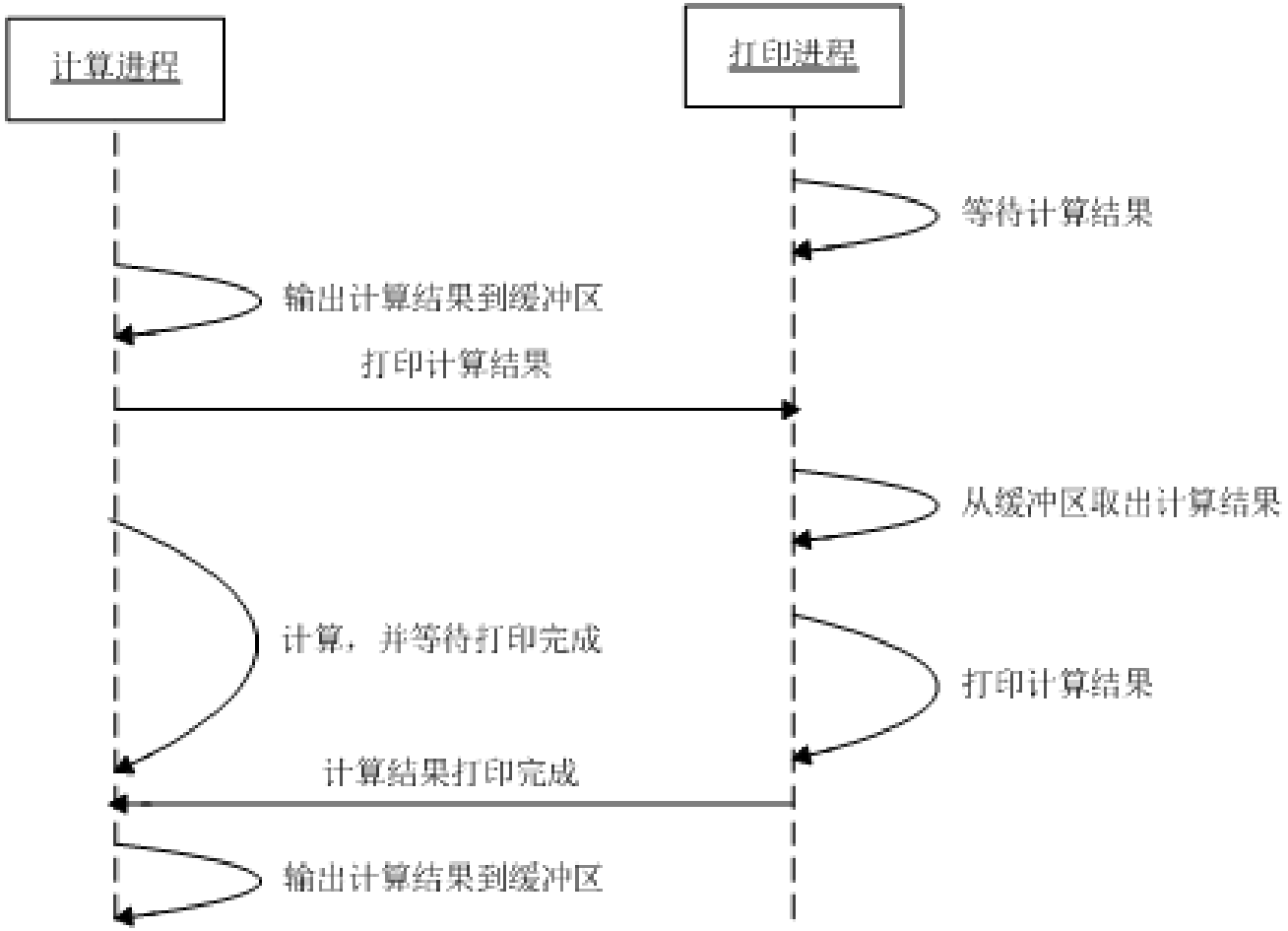
同步

- 同步（synchronization）：
  - 多个相互合作的进程在一些关键点上可能需要互相等待或互相交换信息，这种相互制约关系称为进程同步。
  - 不确定中蕴含了确定性！
- 同步例子：计算进程与打印进程共享一个单缓冲区。



同步

- 计算进程与打印进程的同步



互斥

- 互斥(mutual exclusion)：相互制约关系
  - 当一个进程正在使用某资源时，其他希望使用该资源的进程必须等待
  - 当该进程用完资源并释放后，才允许其他进程去访问此资源

- 即：各进程间竞争使用共享资源，而这些资源需要排他性使用。

临界资源与临界区

- 临界资源：
  - 一段时间内仅允许一个进程使用的资源称为临界资源。(Critical Resource)
  - 如：打印机、共享变量。
- 临界区：
  - 进程中访问临界资源的那段代码称为临界区，又称临界段。
- 同类临界区：
  - 所有与同一临界资源相关联的临界区。

临界资源访问过程

一般包含四个部分：

1.进入区

- 检查临界资源访问状态
- 若可访问，设置临界资源被访问状态

2.临界区

- 访问临界资源代码

3.退出区

- 清除临界资源被访问状态

4.剩余区

- 其他部分

访问临界资源应遵循的原则

1. 空闲让进
   - 若无进程处于临界区时，应允许一个进程进入临界区，且一次至多允许一个进程进入。
2. 忙则等待
   - 当已有进程进入临界区，其他试图进入的进程应该等待。
3. 有限等待
   - 应保证要求进入临界区的进程在有限时间内进入临界区。也蕴含着有限使用的意思！
4. 让权等待
   - 当进程不能进入自己的临界区时，应释放处理机，不至于造成饥饿甚至死锁。

访问临界资源应遵循的原则cont.

- 互斥原则，mutual exclusion
  - 如果进程Pi在其临界区内执行，那么其他进程都不能在其临界区执行
- 推进原则，Progress
  - 如果没有进程在其临界区内执行，并且有进程需要进入临界区，那么只有那些不在剩余区执行的进程可以参与选择，以便确定谁能下次进入临界区，而且这种选择不能无限推迟。
- 有限等待，bounded waiting
  - 从一个进程作出进入临界区的请求，直到这个请求允许为止，其他进程允许进入临界区的次数具有上限。

第6章 进程同步

6.2 互斥的实现方法

6.2 互斥的实现方法

- 互斥的实现
  - 软件方法
  - 硬件方法
  - 更高级的抽象方法

6.2.1 互斥的软件实现方法

- 假设：有两个进程P0和P1
  - P0、P1为并发进程
  - P0、P1互斥地共享某个临界资源
  - P0、P1不断工作，每次使用该资源一个有限的时间间隔。

算法1的思想

- 朴素想法：设置一个开关变量turn
  - turn为公用整型变量，用来指示允许进入临界区的进程标识。
  - 对于P0，若turn为0，则允许进程P0进入临界区;否则循环检查该变量，直到turn变为本进程标识0;
  - 在退出区，修改允许进入进程的标识为1。进程P1的算法与此类似。

算法1的描述

- int turn＝0;

```
P0(){
do{
    while (turn!=0);
    进程P0的临界区代码CS0;
    turn = 1;
    进程P0的其他代码;
    }while(true)
}
```

```
P1(){
do{
    while (turn!=1);
    进程P1的临界区代码CS1;
    turn = 0;
    进程P1的其他代码;
    }while(true)
}
```

- 算法1存在的问题
  - 此算法可以保证互斥访问临界资源，但两个进程在调度中必须以交替次序进入临界区。
  - P0执行完一次循环之后，P1若不被调度执行，P0也无法继续运行;
  - 此算法不能保证实现空闲让进准则。

算法2的思想

- 算法1的程序问题在于
  - 使用标识后设置了标识，导致标识与初始不一致，从而造成两程序的依赖关系
  - 注意：这种依赖关系在后续某些问题中是有价值的！。
- 改进思想：消除依赖关系
  - 设置彼此独立的标志数组flag[i]表示进程i是否在临界区中执行，初值均为假。
  - 在每个进程访问临界资源之前，先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，
  - 在退出临界区时，修改本进程临界区标志为假。

算法2的描述

- enum bool {false，true};
- bool flag[2]＝{false，false};

```
P0：{
      do {
            while (flag[1]);
            flag[0] = true;
            进程P0的临界区代码CS0；
            flag[0] = false ;
            进程P0的其他代码;
          }
      while(true)
    }
```

```
P1：{
      do {
            while (flag[0]);
            flag[1] = true;
            进程P1的临界区代码CS1；
            flag[1] = false ;
            进程P1的其他代码;
          }
      while(true)
    }
```

- 算法2的问题：
  - 此算法解决了空闲让进的问题，但有可能两个进程同时进入临界区。
  - 当两个进程都未进入临界区时，它们各自的访问标志值都为false，若此时刚好两个进程同时都想进入临界区，并且都发现对方标志值为false，两个进程可以同时进入了各自的临界区，这就违背了临界区的访问原则忙则等待。

算法3的思想

- 算法2的程序问题在于
  - 检测点过早，标志flag[0]与flag[1]之间没有形成互反，导致进入条件均可满足
- 算法3的改进：
  - 延迟检测点，本算法仍然设置标志数组flag[i]，标志用来表示进程i是否计划进入临界区，**即一种意愿**。
  - 在每个进程访问临界资源之前，先将自己的标志设置为true，表示进程希望进入临界区
  - 然后再检查另一个进程的标志，若另一个进程的标志为true，则进程等待，否则进入临界区。

算法3的描述

- enum bool {false，true};
- bool flag［2］={false，false};

```
P0：{
    do {
        flag[0] = true;
        while (flag[1]) ;
        进程P0的临界区代码CS0;
        flag[0] = false;
        进程P0的其他代码;
      }
     while(true)
     }
```

```
P1：{
    do {
        flag[1] = true;
        while (flag[0]) ;
        进程P1的临界区代码CS1;
        flag[1] = false;
        进程P1的其他代码;
      }
     while(true)
     }
```

- 算法3的问题：
  - 该算法可以有效地防止两个进程同时进入临界区，但存在两个进程都进不了临界区的问题。
  - 当两个进程同时想进入临界区时，它们分别将自己的标志位设置为true，并且同时去检查对方的状态，发现对方也要进入临界区，于是双方互相谦让，结果谁也进不了临界区。**违背了空闲让进和有限等待原则**

算法4的思想：Dekker算法

- 算法3的问题在于
  - 预期进入临界区标识Flag，无法识别当前已经进入的其他进程情况
- 算法4的改进：
  - 荷兰数学家T. Dekker提出了一种Dekker算法，解决此问题。
  - 本算法的基本思想是算法3和算法1的结合。是一个正确的算法。
  - 标志数组flag[ ]表示进程是否希望进入临界区或是否正在临界区中执行。
  - 还设置了一个turn变量，用于指示允许进入临界区的进程标识。

算法4的描述

- enum bool {false，true};
- bool flag［2］ ={false，false};
- int turn =0 或者 1;

```
P0:
    do{
        flag[0] = true;
        while (flag[1]){
            if (turn ! = 0){
                flag[0] = false;
                while (turn != 0);
                flag[0] = true;
            }
        }
        进程P0的临界区代码CS0;
        turn = 1;
        flag[0] = false;
        进程P0的其他代码;
    } while (true)
```

```
P1:
    do{
        flag[1] = true;
        while (flag[0]){
            if (turn ! = 1){
                flag[1] = false;
                while (turn != 1);
                flag[1] = true;
            }
        }
        进程P1的临界区代码CS1;
        turn = 0;
        flag[1] = false;
        进程P1的其他代码;
    } while (true)
```

讨论

1. 会不会出现无法推进呢，例如，如果等待在p0第一层while循环里如何？如果等待在p0第二层循环里如何?

2. 为什么需要if(turn！=0)呢？如果没有会怎样?

算法5的描述:Peterson算法

- 1981年，G. L. Peterson 给出了一种更为简单的实现算法
  - enum boolean {false，true};
  - boolean flag[2] ={false，false};
  - int turn=0;

```
P0:
{
   do
   {
      flag[0] = true;
      turn = 1;
      while (flag[1] &&turn = = 1)
         ;
      进程P0的临界区代码CS0;
      flag[0] = false;
      进程P0的其他代码;
   } while (true)
}
```

```
P1:
{
   do
   {
      flag[1] = true;
      turn = 0;
      while (flag[0] &&turn = = 0)
         ;
      进程P1的临界区代码CS1;
      flag[1] = false;
      进程P1的其他代码;
   } while (true)
}
```

6.2.2 硬件方法

- 软件算法实现互斥是较为困难的
  - 最大的问题是对临界区的**检测与设置**动作很难作为一个**整体**来实现
  - 软件一条语句可能会被拆分
- 用硬件方法实现互斥的主要思想是
  - 在单处理器情况下，并发进程是交替执行的，因此只需要保证检查操作与修改操作不被中断即可，因此可以对关键部分进行硬件实现
  - 关中断方法
  - 原子指令方法

1、关中断方法

- 当进程执行临界区代码时，要防止其他进程进入其临界区访问，最简单的方法是关中断。Why?
- 关中断
  - 能保证当前运行进程将临界区代码顺利执行完，从而保证了互斥的正确实现，然后再允许中断。
  - 现代计算机系统都提供了关中断指令。
  - 中断响应将延迟到中断启用之后

用关中断方法实现互斥

关中断;

临界区;

开中断;

关中断方法的不足

- 效率问题
  - 如果临界区执行工作很长，则无法预测中断响应的时间
  - 系统将处于暂停状态，无法响应事件
  - 限制了处理机交替执行程序的能力，执行的效率将会明显降低;
- 适用范围问题
  - 关中断不一定适用于多处理器计算机系统
  - 一个处理器关掉中断，并不意味着其他处理器也关闭中断，不能防止进程进入其他处理器执行临界代码。
- 安全性问题
  - 将关中断的权力交给用户进程则很不明智，若一个进程关中断之后不再开中断，则系统可能会因此终止甚至崩溃。
  - 因此，只适用于操作系统内核进程。

2、原子指令方法

- 许多计算机中提供了专门的硬件指令，实现对字节内容的检查和修改或交换两个字节内容的功能。
- 使用这样的硬件指令就可以解决临界区互斥的问题。
  - 比较并交换指令（Swap）
    - X86系统中是compare and exhange指令
  - 测试并设置指令（TS)
    - X86系统中是xchg（atomic exhange原子交换）指令

TS（Test-and-Set）指令

- TS指令的功能逻辑描述如下：

```
boolean  TS(boolean *lock)
{
        boolean old;
        old=*lock;
        *lock=true;
        return old;
  }
```

用TS指令实现进程互斥

- 为每个临界资源设置一个共享布尔变量lock表示资源的两种状态：true表示正被占用，false表示空闲。算法如下：

```
    ┆
while  TS(&lock);
进程的临界区代码CS；
lock=false；
进程的其他代码;
    ┆
```

Swap指令

- Swap指令的功能逻辑描述如下：

```
Swap(boolean *a，boolean *b)
{
   boolean  temp;
   temp=*a;
   *a=*b;
   *b=temp;
}
```

用Swap指令实现进程互斥

- 为每个临界资源设置一个共享布尔变量lock表示临界资源状态;再设置一个局部布尔变量key用于与lock交换信息。算法如下：

```
        ┆
key=true;
while(false !=key) Swap(&lock，&key);
进程的临界区代码CS；
lock=false ;// or Swap(&lock，&key);
进程的其他代码;
        ┆
```

原子指令方法的优缺点

- 优点：
  - 适用于多处理器环境
- 缺点：
  - 复杂、需要硬件支持
  - 不满足有限等待
  - 不满足让权等待，忙等浪费CPU

6.2.3 互斥锁机制

- 互斥锁是一个代表资源状态的变量，通常用0表示资源可用（开锁），用1表示资源已被占用（关锁）。
- 在使用临界资源前需先考察锁变量的值
  - 如果值为0则将锁设置为1（上锁）
  - 如果值为1则回到第一步重新考察锁变量的值
  - 当进程使用完资源后，应将锁设置为0（开锁）。

锁原语

- 操作系统提供了上锁和开锁两种原子操作：
- 上锁

```
lock（w）
{
   while（w==1）;//等待可锁状态
   w = 1;//加锁
}
```

- 开锁

```
unlock（w）
{
    w = 0;//解锁
}
```

用互斥锁机制实现互斥

进程P_1

```
lock(w);
临界区;
unlock(w);
```

进程P_2

```
lock(w);
临界区;
unlock(w);
```

自旋锁

- 互斥锁往往采用前面所介绍的硬件机制来实现；
- 上述实现的缺点是存在忙等，因此也叫做自旋锁。
- 对于单处理器，为了提高效率，忙等会改为休眠，释放时要唤醒。
  - 但是，锁的申请和释放，带来上下文切换。
- 对于多核处理器的busy waiting：一个线程在一个处理器中自旋，另一个线程可以在另一个处理器中运行，表现形式则是用户进程可以继续工作
  - 对于多核线程，预计线程等待锁的时间很短，短到持有自旋锁的时间小于两次上下文切换的时间，则使用自旋锁是划算的。

自旋锁

- 互斥锁
  - 往往采用前面所介绍的硬件机制来实现
  - 对于单处理器，为了提高效率，忙等会改为休眠，释放时要唤醒
  - 锁的申请和释放，带来上下文切换，但是，在多处理器情况下，会造成非常大的浪费
  - 自旋锁是专为防止多处理器并发而引入的一种锁。
- 自旋锁工作机理
  - 自旋锁最多只能被一个内核任务持有，如果一个内核任务试图请求一个已被争用(已经被持有)的自旋锁，那么这个任务就会一直进行忙循环——旋转——等待锁重新可用。要是锁未被争用，请求它的内核任务便能立刻得到它并且继续进行
  - 多核处理器中的busy waiting：一个线程在一个处理器中自旋，另一个线程可以在另一个处理器中运行，表现形式则是用户进程可以继续工作
  - 对于多核线程，预计线程等待锁的时间很短，短到持有自旋锁的时间小于两次上下文切换的时间，则使用自旋锁是划算的。

小结

- 软件解法：
  - 复杂、容易出现逻辑漏洞
- 关中断
  - 不适应多处理器环境、不适应于用户进程
- 原子指令方法
  - 需要硬件支持、忙等浪费CPU
- 互斥锁
  - 忙等浪费CPU；阻塞需要两次上下文切换