- 基本概念
- 同步与互斥
  - 软件方法
  - 硬件方法
- 信号量
- 经典进程同步问题
- 管程
- 小结

基本概念

4.1.1 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。

同步与互斥

- 同步与互斥基本概念
  - 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。

- 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

软件方法

- ppt中算法一直在解决的一个问题就是，如果两个进程同时想要访问临界区的资源如何解决，于是得出了**用turn表示现在轮到谁来访问临界区，flag数组表示访问临界区的意愿**
- Dekker算法

算法4的描述

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

```
P0:
1.    do{
2.       flag[0] = true;
3.       while (flag[1]){
4.          if (turn!=0){
5.             flag[0] = false;
6.             while (turn!=0);
7.             flag[0] = true;
8.          }
9.       }
10.      进程P0的临界区代码CS0;
11.      turn = 1;
12.      flag[0] = false;
13.      进程P0的其他代码;
14.   } while (true)
```

```
P1:
1.    do{
2.       flag[1] = true;
3.       while (flag[0]){
4.          if (turn!=1){
5.             flag[1] = false;
6.             while (turn!=1);
7.             flag[1] = true;
8.          }
9.       }
10.      进程P1的临界区代码CS1;
11.      turn = 0;
12.      flag[1] = false;
13.      进程P1的其他代码;
14.   } while (true)
```

- 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;
```

```
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)
}
```

硬件方法

- 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

- 关中断

- 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

```
    ¦
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。

- 原子指令

- 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。
  - 测试设置方法
  - 对换指令方法

- 测试设置方法

- TS指令的功能可描述如下：

```
boolean TS(boolean * lock)
{
    if (false == *lock){
          *lock = true;
          return false;
    }
    else
          return true;
}
```

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

- 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;
        ┆
```

- 对换指令方法

Swap指令（或Exchange指令）

- Swap指令的功能可描述如下：

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

- 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;
      ┆
```

- 互斥锁

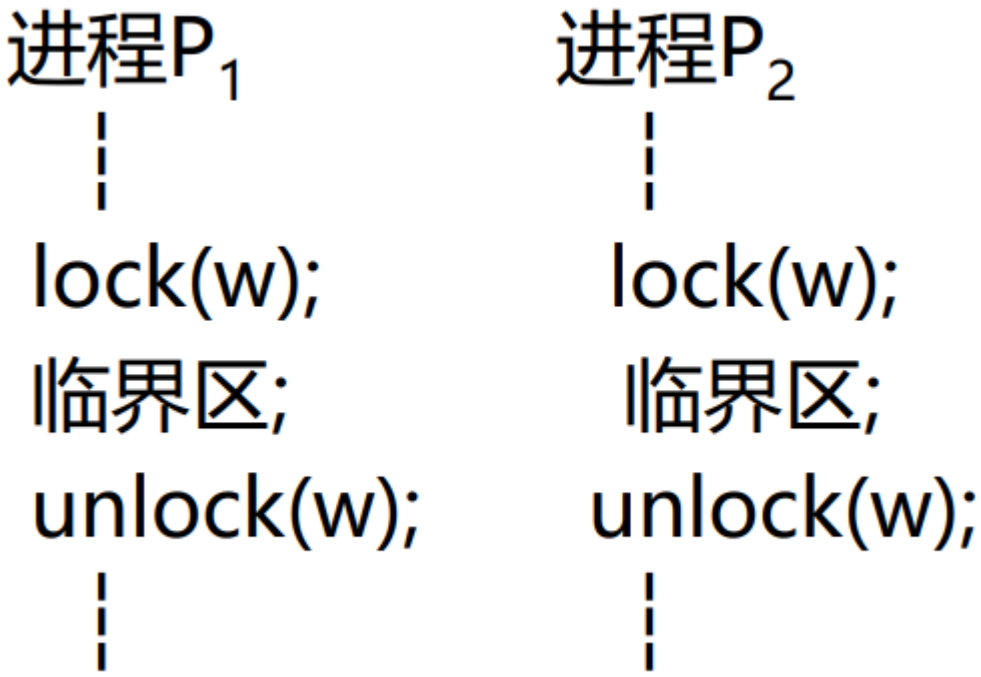
- 互斥锁是一个代表资源状态的变量，通常用0表示资源可用（开锁），用1表示资源已被占用（关锁）。
- 在使用临界资源前需先考察锁变量的值
  - 如果值为0则将锁设置为1（关锁）
  - 如果值为1则回到第一步重新考察锁变量的值
  - 当进程使用完资源后，应将锁设置为0（开锁）。

- 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
}
```


进程P_1
lock(w);
临界区;
unlock(w);
进程P_2
lock(w);
临界区;
unlock(w);

信号量

信号量（Semaphore）是一种用于控制对共享资源访问的同步机制，最初由荷兰计算机科学家艾兹赫·迪克斯特拉（Edsger W. Dijkstra）提出。信号量主要用于多线程或多进程环境中，用于管理对共享资源的访问，以防止竞争条件和数据不一致性。

信号量可以被看作是一个计数器，它可以有一个整数值，并且支持两种基本操作：`wait` 和 `signal`。

1. `wait` 操作（也称为 P 操作）：如果信号量的值大于 0，则将其减 1；如果值为 0，则调用线程（或进程）将被阻塞，直到信号量的值变为大于 0。
2. `signal` 操作（也称为 V 操作）：将信号量的值加 1。如果有线程（或进程）因为执行 `wait` 操作而被阻塞，则会有一个或多个线程（或进程）被唤醒。

信号量的一般性质使其可以用于多种并发场景，例如：

- 保护临界区：使用信号量可以确保同一时间只有一个线程或进程可以进入临界区，从而避免竞争条件。
- 控制资源访问：如果共享资源的数量有限，可以使用信号量来控制对其的访问，确保同时只有有限数量的线程或进程可以访问。
- 实现生产者-消费者模型：信号量可以用于实现生产者-消费者模型，其中生产者和消费者之间通过信号量来同步对共享缓冲区的访问。

虽然信号量是一种强大的同步原语，但它也存在一些问题，如死锁和活锁等。因此，在实际使用中，需要仔细设计和管理信号量的使用，以避免出现潜在的并发问题。

- 定义

- 信号量(semaphore)由两个成员（s，q）组成
  - 其中s是一个具有非负初值的整型变量
  - q是一个初始状态为空的队列。

- 除信号量的初值外，信号量的值仅能由P操作（又称为wait操作）和V操作（又称为signal操作）改变。

- P操作

P操作

- 设S=（s，q）为一个信号量，P（S）执行时主要完成下述动作：
  - s = s - 1;
  - If (s < 0) {

    设置进程状态为等待;

    将进程放入信号量等待队列q;

    转调度程序;

    }

- V操作

V操作

- V(S)执行时主要完成下述动作：
  - s = s + 1;
  - If(s≤0){
    将信号量等待队列q中的第一个进程移出;
    设置其状态为就绪状态并插入就绪队列;
    然后再返回原进程继续执行;
    }

- 几个重要含义
  - 信号量中的整型变量s表示系统中某类资源的数目。
  - 当s>0时，
    - 该值等于在封锁进程之前对信号量S可施行的P操作数，
    - 等于S所代表的实际还可以使用的资源数
  - 当s<0时，
    - 其绝对值等于登记排列在该信号量S队列之中等待的进程个数，
    - 亦即恰好等于对信号量s实施P操作而被封锁起来并进入信号量s队列的进程数
- 一些理解 P操作相当于wait，就是把当前进程加入到等待队列中，V操作相当于signal，就是从等待队列取出一个继续执行。同时要注意P，V操作具有原子性

- 通常，P操作意味着请求一个资源，V操作意味着释放一个资源。
  - 在一定条件下，P操作代表使进程阻塞的操作，而V操作代表唤醒阻塞进程的操作。

- P、V操作的原子性要求
  - 即，一个进程在信号量上操作时，不会有别的进程同时修改该信号量。
  - 对于单处理器，可简单地在**封锁中断**的情况下执行
  - 对于多处理器环境，需要**封锁所有处理器的中断**，困难且影响性能，往往用swap()或自旋锁等方式加锁

- 资源是有限的，wait()用来获取资源，如果资源数<0，说明当前资源已经被占用完了，但是这个进程又需要资源，那么就要等待，因此加入等待队列，signal()就是从等待队列中取出一个进程进行唤醒

信号量的另一种描述

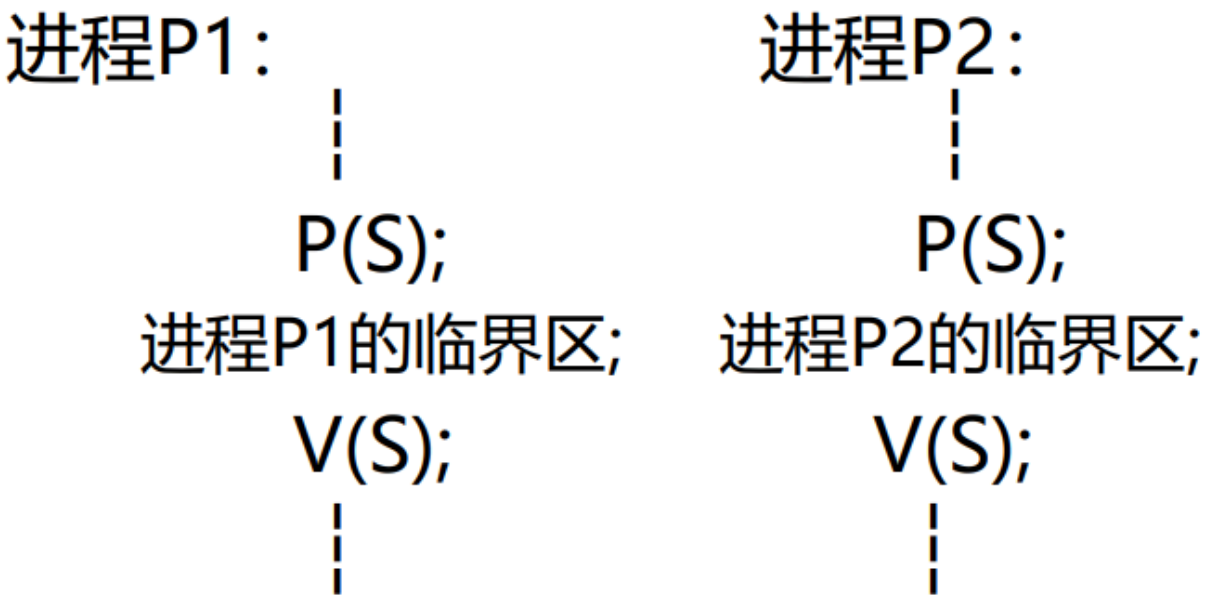
```
typedef struct{
   int value;
   struct process *list;
   } semaphore;

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

- 利用信号量实现互斥

- 设S=(s,q)为两个进程P1、P2实现互斥的信号量
  - s的初值应为1，即可用资源数目为1。
- 只需把临界区置于P（S）和V（S）之间，即可实现两进程的互斥。

-



- 互斥信号量的取值范围

- **P,V操作实际上就是阻塞和唤醒操作，用这种方式更好理解**

经典进程同步问题

利用信号量解决同步问题的思路

- 理清同步与互斥关系
  - 哪些资源及对象需要互斥访问
  - 哪些资源的访问顺序对进程调度有制约关系
  - 同步信号量要表示出资源的等待条件及数目
  - P操作内包含等待；V操作内包含唤醒
  - 依多个访问顺序约束，同类资源可设置多个信号量
    - 生产者消费者问题中的empty和full
  - 一定要注意互斥量与同步信号量的顺序
    - 同步P优先于互斥P
  - 信号量的操作只能为PV，切记不要直接取指和赋值
    - 可设置副本，如理发师问题中的count 和customers

1. 生产者—消费者问题
2. 读者—写者问题
3. 哲学家进餐问题
4. 睡眠的理发师问题

管程

- 一个通俗的理解

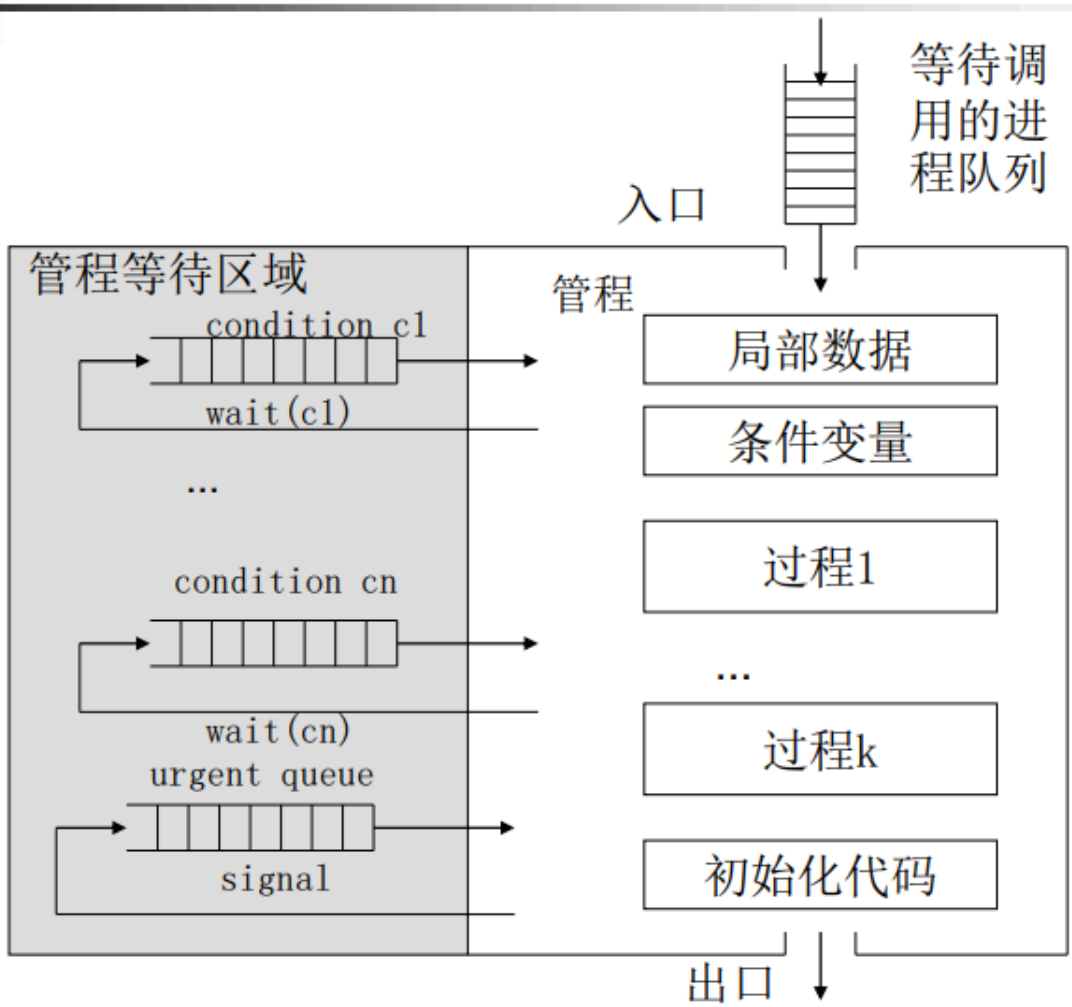
- 信号量的同步操作分散在各进程中不便于管理，还可能导致系统死锁。如：生产者消费者问题中将P颠倒可能死锁。
- 为此Dijkstra于1971年提出：把所有进程对某一种临界资源的同步操作都集中起来，构成一个所谓的秘书进程。凡要访问该临界资源的进程，都需先报告秘书，由秘书来实现诸进程对同一临界资源的互斥使用。

- 定义:管程定义了一个数据结构和能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据。
- 构成：
  - 局部于管程的共享数据结构
  - 对共享数据结构进行操作的一组函数
  - 对局部于管程的数据设置初始值的语句
- 语法：

```
type 管程名=monitor {
   局部变量说明;
   条件变量说明;
   初始化语句;
define 管程内定义的，管程外可调用的过程或函数名列表;
use 管程外定义的，管程内将调用的过程或函数名列表;
过程名/函数名(形式参数表) {
    <过程/函数体>;
   }
                    …
过程名/函数名(形式参数表) {
    <过程/函数体>;
   }
}
```

- 特性：
  1. 安全性
     - 管程内的局部数据**只能**被该管程内的函数所访问。
  2. 共享性
     - 进程对管程的访问**都可以通过**共享相同的管程函数来实现。
  3. 互斥性：
     - **每次仅允许一个**进程在管程内执行某个函数，共享资源的进程可以访问管程，但是**只有至多一个**调用者**真正**进入管程，其他调用者**必须**等待。
  4. 透明性
     - 管程是一个语言成分，所以管程的互斥访问完全由编译程序在编译时自动添加上，**无需**程序员关心，而且保证正确。
- 结构：

管程的结构



- 管程中涉及的同步等问题

小结

- 临界资源访问的原则
- 同步与互斥的定义
- 互斥的实现方法（软件、硬件、锁）
- 信号量的定义
- 信号量解决经典问题
- 管程的基本概念
- 同步技术的新进展