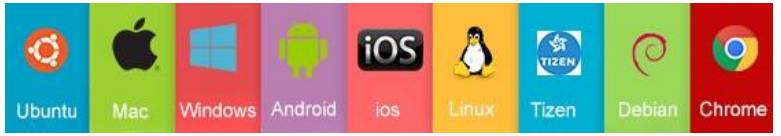
操作系统
Operating System

汤臣薇

tangchenwei@scu.edu.cn

四川大学计算机学院（软件学院）

数据智能与计算艺术实验室



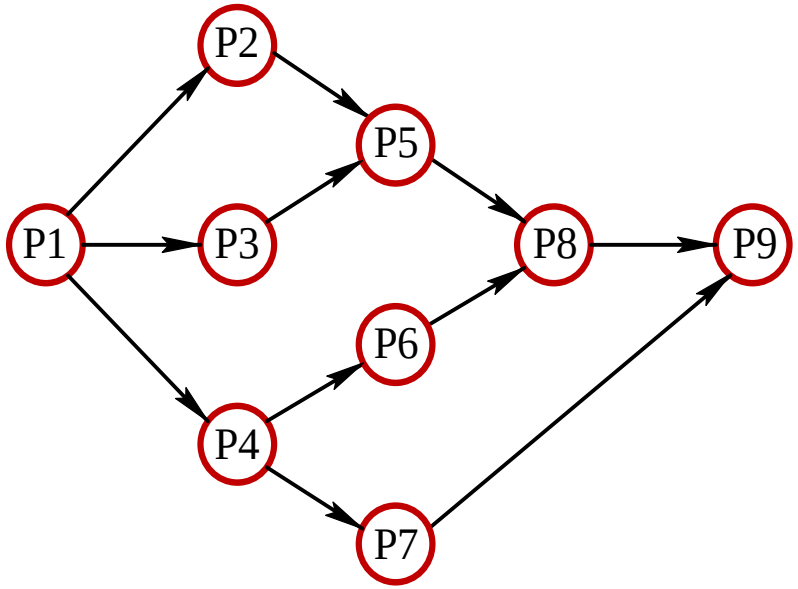
进程的描述和控制

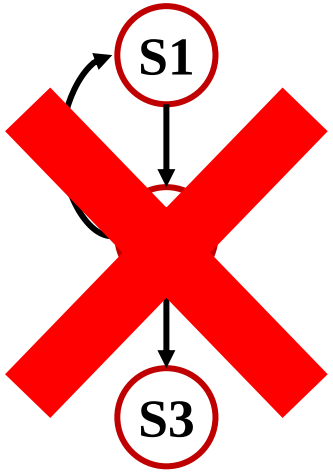
- 前趋图和程序执行
- 进程的描述
- 进程控制
- 进程同步 ★
- 经典进程的同步问题 ★
- 进程通信
- 线程的基本概念
- 线程的实现

回顾——前趋图

◆ **前趋图（Precedence Graph）：描述程序执行先后顺序**

- 描述多个进程之间的关系
- 有向无循环图（DAG）
- 结点表示一个进程或一段程序
- 结点之间用一个有方向的线段相连
- 方向表示所连接的结点之间的前趋和后继关系
- 被指向的结点为后继结点，离开箭头的结点是前趋结点

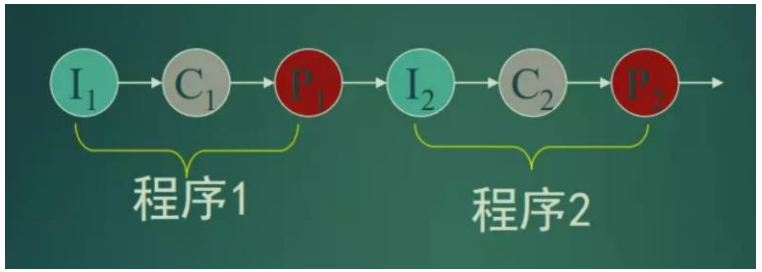




回顾——程序的顺序执行

➢ 顺序性

- 每个操作必须在上一个操作完成后才能开始；一个程序模块执行完成后另一个程序模块才能开始

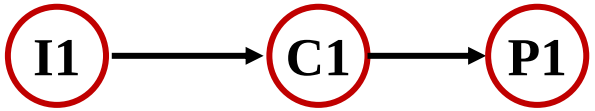


➢ 封闭性

- 程序运行独占系统全部资源，程序执行的结果除初始条件外，由程序本身决定，不会受到任何其它程序和外界因素的干扰

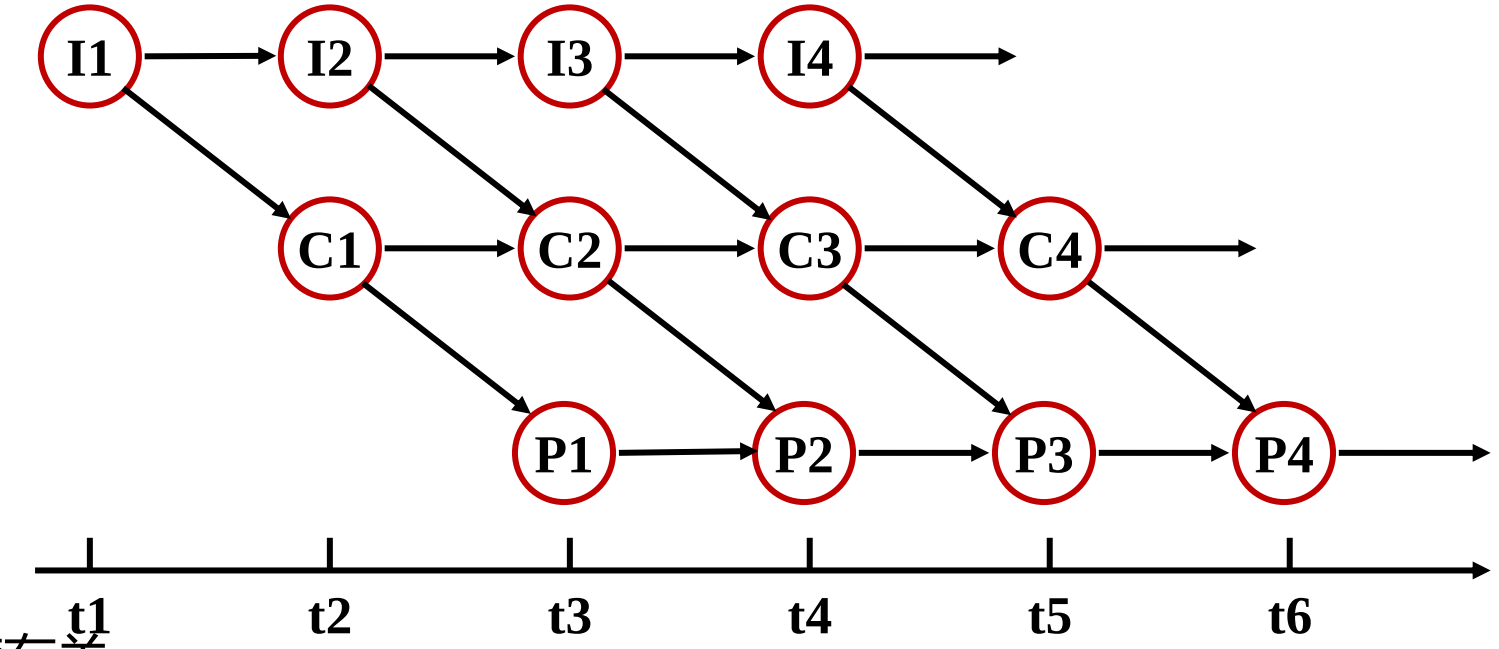
➢ 再现性

- 针对相同的数据集合，程序执行的结果总是相同的；中断对程序执行的最终结果没有影响；程序执行的结果是可再现的。



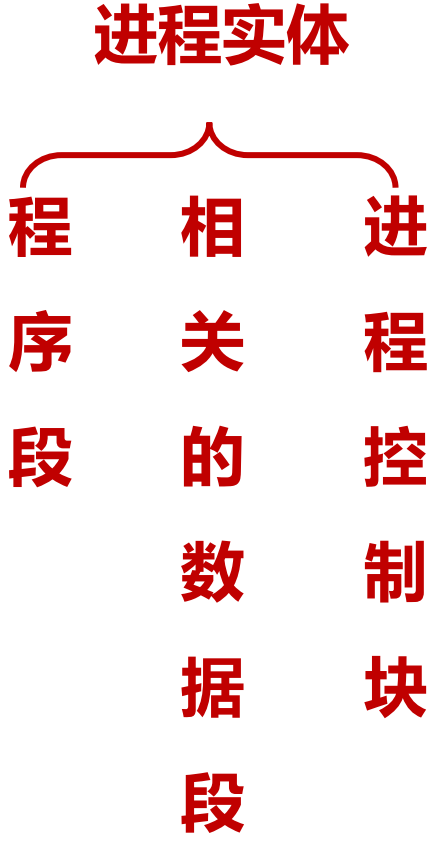
回顾——程序的并发执行

- **间断性制约**
  - 资源共享致使并发执行的程序形成相互制约的关系
  - 并发程序：执行→暂停→执行
- **失去封闭性**
  - 资源共享导致运行失去封闭性
  - 程序等待
- **不可再现性**
  - 计算结果与并发程序的执行速度有关



回顾——进程的定义

- 进程是**程序**的一次**执行**
- 进程是一个程序及其数据在处理机上顺序**执行**时所发生的**活动**
- 进程是具有独立功能的程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的**独立单位**
- **进程是进程实体的运行过程，是操作系统进行资源分配和调度的基本单位**



回顾——进程的特征

结构性：进程包含有描述进程信息的数据结构（包含进程控制块、程序块和代码块等）和运行在进程上的程序，OS用PCB描述和记录进程的动态变化过程。

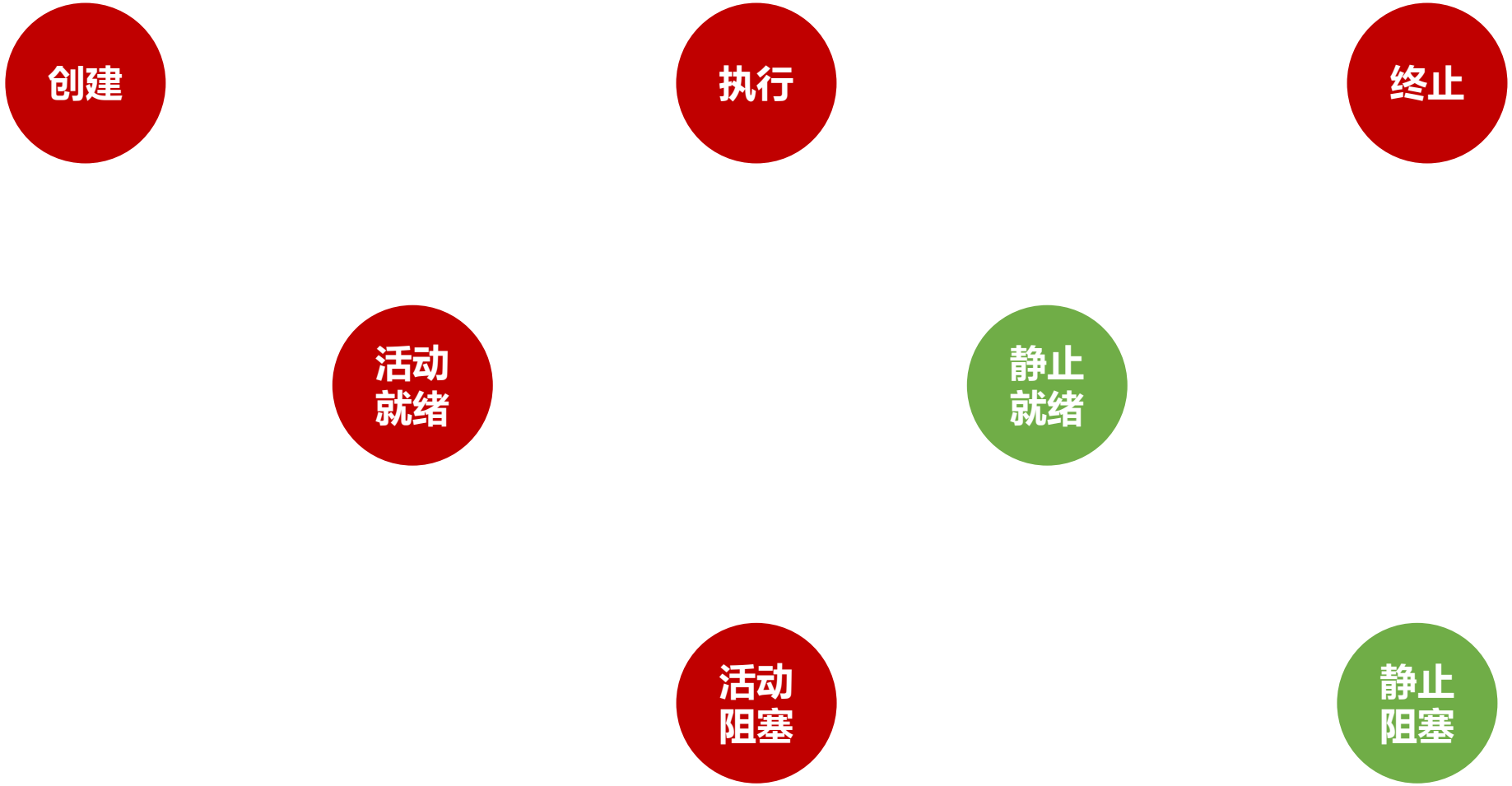
动态性：最基本特征，是程序执行过程，有一定的生命期：由创建而产生、由调度而执行，因得不到资源而暂停，由撤消而死亡。而程序是静态的，它是存放在介质上一组有序指令的集合，无运动的含义。

并发性：进程的重要特征，也是OS的重要特征。指多个进程实体同存于内存中，能在一段时间内同时运行。而程序（没有建立PCB）是不能并发执行。

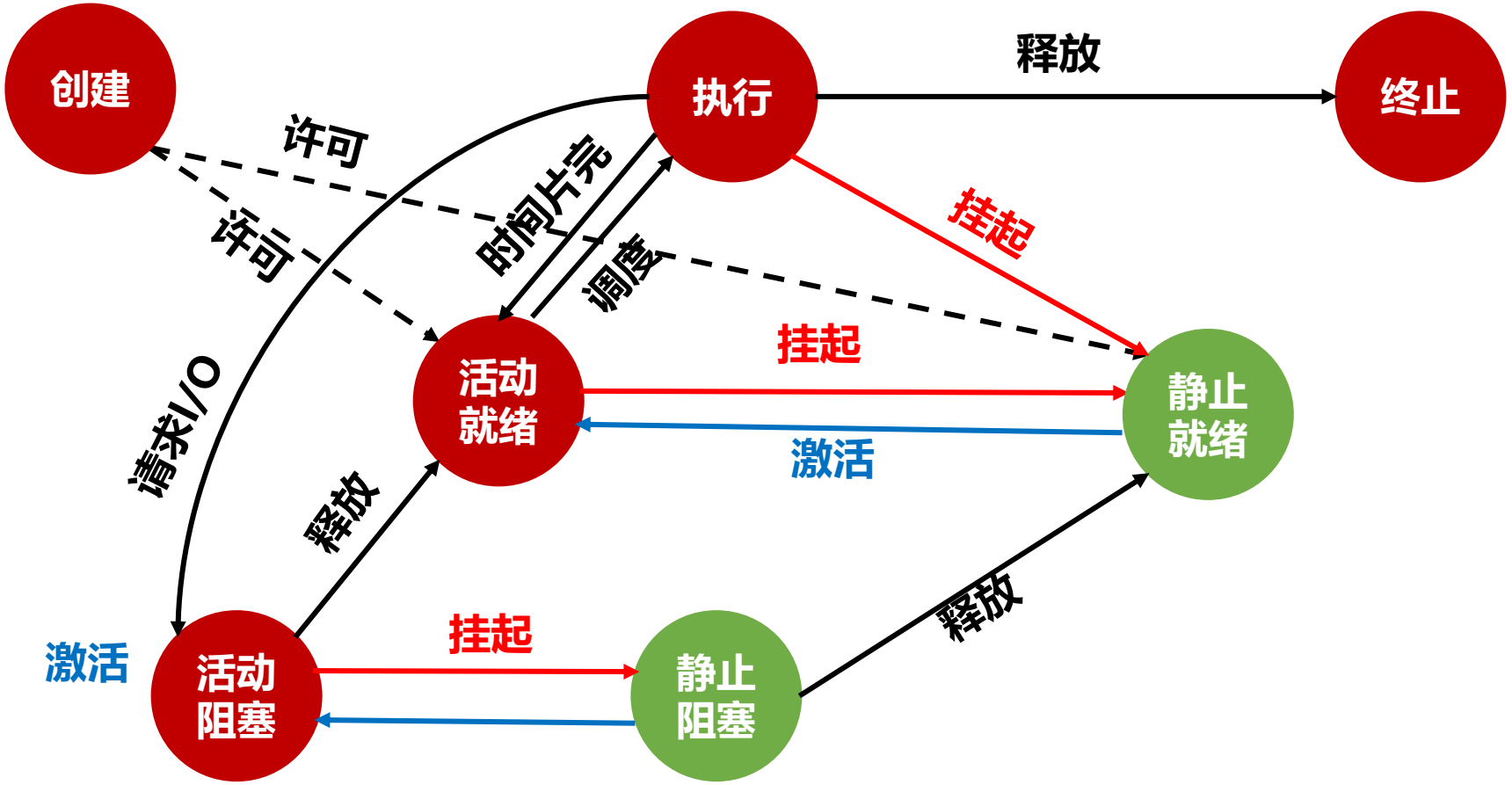
独立性：进程是一个能独立运行的基本单位，即是一个独立获得资源和独立调度的单位，而未简历PCB的程序不能作为独立单位参加运行、获取资源。

异步性：进程按各自独立的不可预知的速度向前推进（即按异步方式进行），正是这一特征导致程序执行的不可再现性，因此OS必须采用某种措施来限制各进程推进序列以保证各程序间正常协调运行。

回顾——进程的基本状态与转换



回顾——进程的基本状态与转换



回顾——进程管理中的数据结构

◆ 进程控制块PCB的作用

- 作为独立运行基本单位的标志
- 能实现间断性运行方式
- 提供进程管理所需要的信息
- 提供进程调度所需要的信息
- 实现与其他进程的同步与通信

◆ 进程控制块PCB中的信息

- 进程标识符（进程的唯一标识）
- 处理机状态（上下文）
- 进程调度信息
- 进程控制信息

进程标志符	进程名
进程调度信息	进程状态
	进程优先级
进程控制信息	现场保留区
	指示出于同一状态进程的链指针
	资源清单
	进程起始地址
	家族关系
	其他

回顾——进程的创建和终止

◆ 引起创建进程的事件

- 操作系统初始化：
- 提供用户服务
- 分时系统用户登录
- 用户请求系统创建新进程
- 执行创建新进程的系统调用
- 批处理作业的初始化和调度

◆ 引起进程终止的事件

- 进程正常结束
- 操作异常退出：保护错（写只读文件）、算术运算错、非法指令、特权指令错、I/O故障
- 时间指标超限引起进程异常结束：运行超时、等待超时
- 多个进程之间竞争资源
- 内存的使用出错：越界错
- 父进程结束或请求
- 操作系统终止：系统死锁

回顾——进程的创建和终止

◆ 创建步骤（creat原语）

命名进程：为进程设置进程标志符；

从PCB集合中为新进程申请一个空白PCB；

确定进程的优先级；

为进程的程序段、数据段和用户栈分配内存空间；如果进程中需要共享某个已在内存的程序段，则必须建立共享程序段的链接指针；

为进程分配除内存外的其他各种资源；

初始化PCB，将进程的初始化信息写入PCB；

如果就绪队列能够接纳新创建的进程，则将新进程插入到就绪队列；

通知OS的其他管理模块，如记账程序、性能监控程序等。

回顾——进程的创建和终止

◆ 进程的终止过程

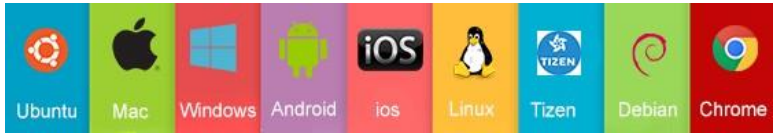
第一步：根据被终止进程的标识符，从PCB集合中查找对应进程控制块并读出该进程的状态；

第二步：若被终止进程正处于执行状态，则终止该进程的执行，并设置调度标志为真，用于指示该进程被终止后应重新进行调度，选择一新进程，把处理机分配给它。

第三步：若进程还有子孙进程，应将其所有子孙进程终止，以防它们成为不可控制的。

第四步：将进程所占有的全部资源释放（还给父进程或系统），释放进程控制块（若该进成为执行态，要进行进程调度）。

第五步：将被终止进程（它的PCB）从所在队列（或链表）中移出，等待其他程序来收集相关信息。



- 进程
- 多道程序并发执行
- 资源利用率
- 系统吞吐量
- 系统更加复杂

进程同步的基本概念

多道程序系统
进程同步机制

- 进程运行管理
- 对系统资源的无序争夺
- 系统混乱
- 结果不确定性
- 结果不可再现性

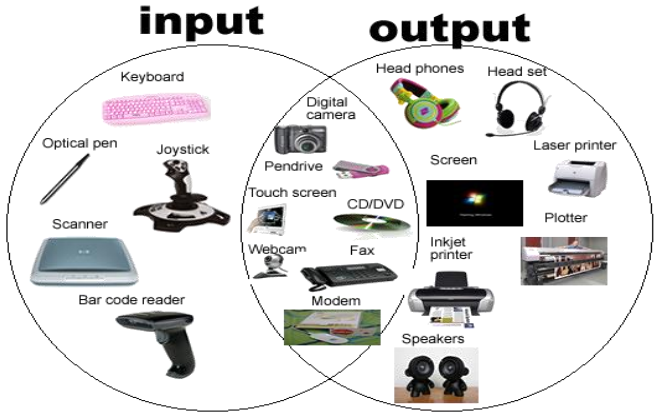
硬件同步机制

信号量机制

管程机制

任务：对多个相关进程在执行次序上进行协调，使并发执行的诸进程之间按照一定的规则(或时序)共享系统资源，并能很好地相互合作，从而使程序的执行具有可再现性

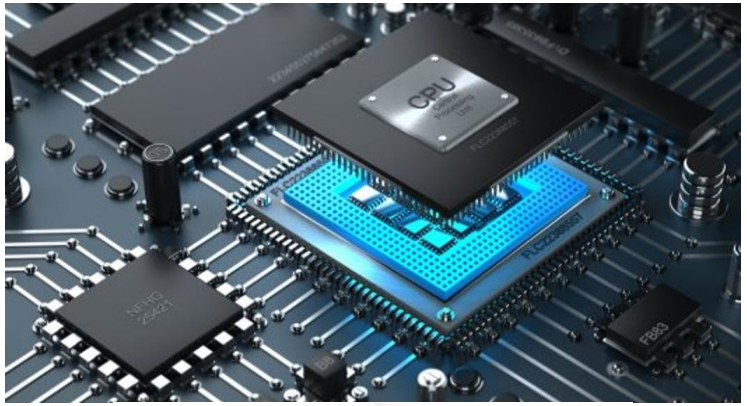
2.4.1 进程同步的基本概念



◆ 两种形式的制约关系

➢ 间接相互制约关系

- 多个程序在并发执行时，对**共享系统资源**及**临界资源**访问时形成间接相互制约关系。
- 对于**互斥共享资源**，必须由系统实施统一分配，使用前，应提出申请，不允许用户进程直接使用。



➢ 直接相互制约关系

- 为完成**同一任务而相互合作**的进程（如输入进程和计算进程共享缓冲区）

2.4.1 进程同步的基本概念

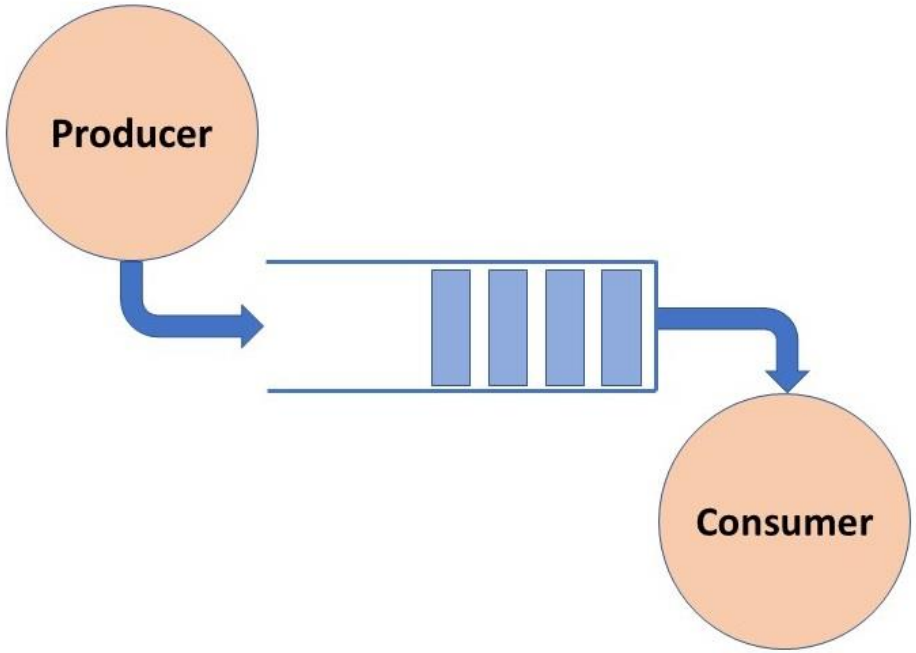
◆ 进程同步和互斥

- ➢ 进程间的相互关系主要有三种：同步、互斥、通信
- ➢ 进程的同步关系
  - 若干合作进程为了完成一个共同的任务，需要相互协调运行进度，一个进程开始某个操作之前，必须要求另一个进程已经完成某个操作，否则前面的进程只能等待。
- ➢ 进程的互斥关系：和资源共享相关
  - 多个进程由于共享了独占性资源，必须协调各进程对资源的存取顺序，确保没有任何两个或以上的进程同时进行存取操作
- ➢ 互斥关系也属于同步关系，是一种特殊的同步。

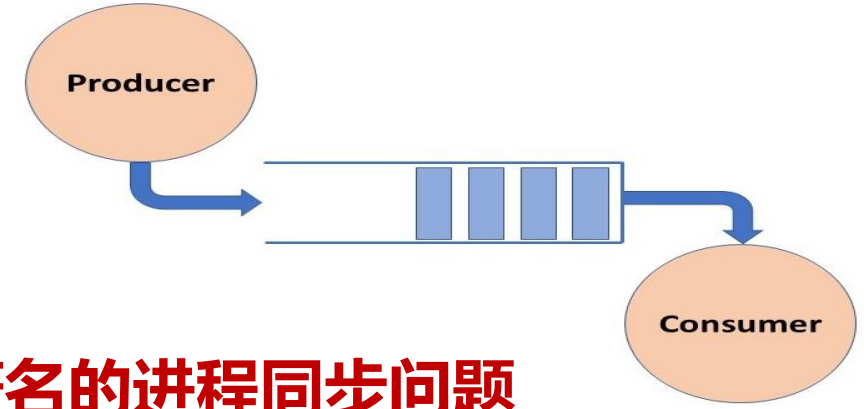
2.4.1 进程同步的基本概念

◆ 临界资源（Critical Resource）

➢ 进程采取互斥方式实现对资源的共享



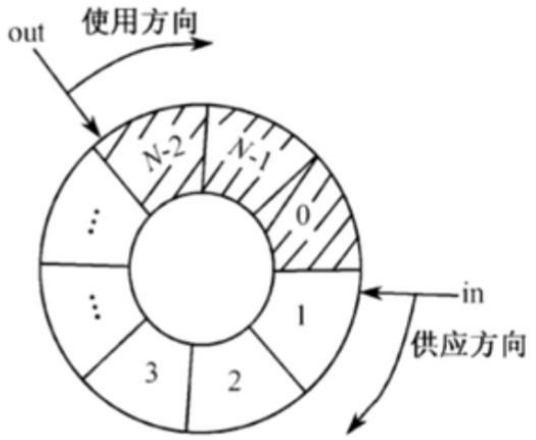
2.4.1 进程同步的基本概念

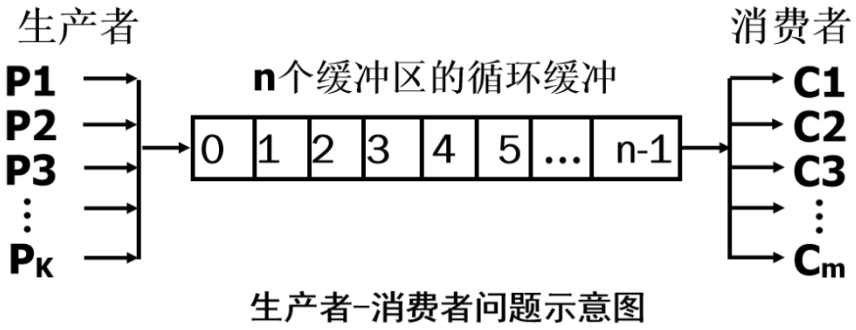


◆ 临界资源（Critical Resource）

➢ 生产者与消费者（producer-consumer）问题——著名的进程同步问题

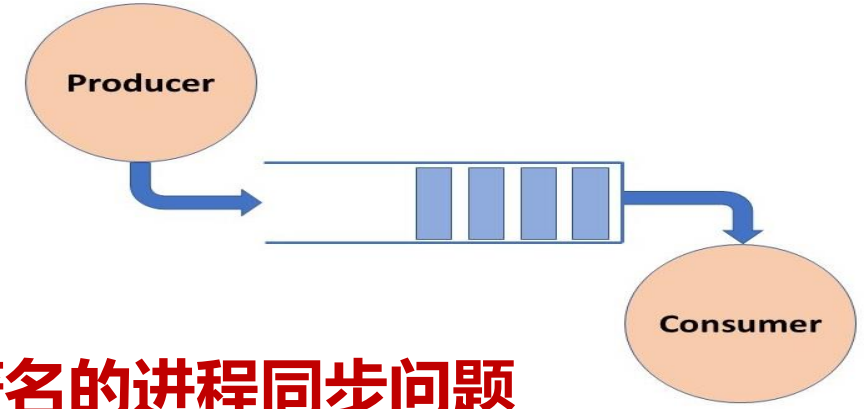
- ➢ 缓冲池：数组表示，具有n个(0，1，…，n-1)缓冲区
- ➢ 输入指针in：指示下一个可投放产品的缓冲区
- ➢ 输出指针out：指示下一个可从中获取产品的缓冲区
- ➢ 缓冲池采用循环组织，故：
  - 输入加1表示成 in:= (in+1) % n；
  - 输出加1表示成out:= (out+1) % n；
  - (in+1) % n=out时表示缓冲池满；
  - in=out则表示缓冲池空。
- ➢ 整型变量counter：
  - 生产者投放产品counter加1；
  - 消费者取走产品counter减1。





生产者-消费者问题示意图

2.4.1 进程同步的基本概念



◆ 临界资源（Critical Resource）

➢ 生产者与消费者（producer-consumer）问题——著名的进程同步问题

生产者进程

```
producer: repeat
    produce an item in nextp;
    while counter=n do no-op;
    buffer［in］:=nextp;
    in:=in+1 mod n;
    counter:=counter+1;
until false;
```

异步运行

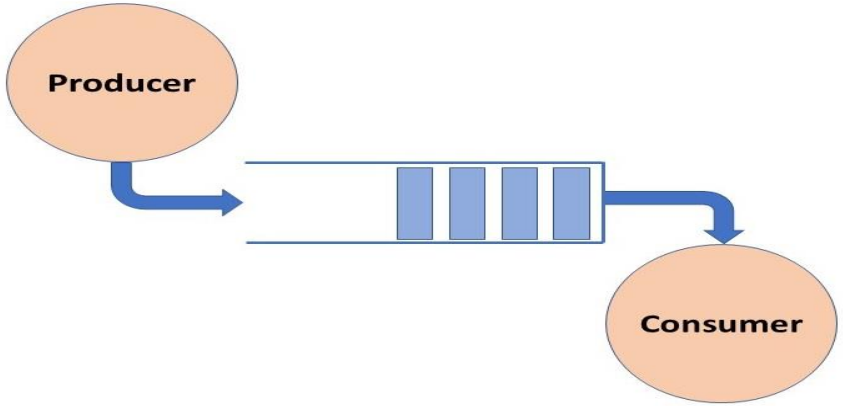
```
int in = 0;
int out = 0;
int count = 0;
item buffer[n];
```

保持同步

消费者进程

```
consumer: repeat
    while counter=0 do no-op;
    nextc:=buffer［out］;
    out:=(out+1) mod n;
    counter:=counter-1;
    consumer the item in nextc;
until false;
```

2.4.1 进程同步的基本概念



◆ 临界资源（Critical Resource）

➢ 问题的出现

```
register1:=counter;
register1:=register1+1;
counter:=register1;
```

```
register2:=counter;
register2:=register2-1;
counter:=register2;
```

程序的执行失去了再现性，为预防这种错误，应把变量counter作为临界资源处理，即令生产者进程和消费者进程互斥地访问变量counter

2.4.1 进程同步的基本概念

◆ 临界区Critical Section

- **临界区：**每个进程中访问临界资源的那段代码
- **进入区：**检查临界资源是否正在被访问的代码（**判别能否访问临界资源的关键**）
- **退出区：**将临界区正被访问的标志恢复为未被访问的标志
- **剩余区：**除进入区、临界区、退出区之外的代码

```
While ( Ture )
{
        entry_section;          //申请进入（进入区）
        critical_section;       //临界区
        exit_section;           //声明退出（退出区）
        remainder_section;      //剩余区
}
```

2.4.1 进程同步的基本概念

◆ 临界区Critical Section进入准则

- **单个入区：**一次仅允许一个进程进入。
- **独自占用：**处于临界区内的进程不可多于一个。如果已有一个进入临界区，其它试图进入的进程必须等待。
- **尽快退出：**访问完后尽快退出，以让出资源。
- **落败让权：**如果进程不能进入临界区，则应让出CPU，以免出现“忙等”现象。

2.4.1 进程同步的基本概念

◆ 同步机制应遵循的规则

➢ 空闲让进：

- 当无进程在互斥区时，任何有权使用互斥区的进程可进入

➢ 忙则等待：

- 不允许两个以上的进程同时进入互斥区

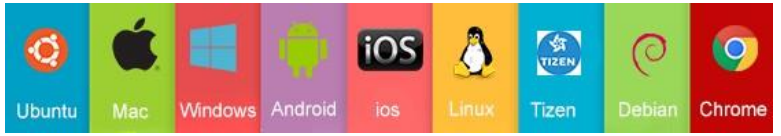
➢ 有限等待：

- 任何进入互斥区的要求应在有限的时间内得到满足

➢ 让权等待：

- 处于等待状态的进程应放弃占用CPU，以使其他进程有机会得到CPU的使用权


第二章
进程的描述和控制



- 进程
- 多道程序并发执行
- 资源利用率
- 系统吞吐量
- 系统更加复杂

进程同步的基本概念

多道程序系统
进程同步机制

- 进程运行管理
- 对系统资源的无序争夺
- 系统混乱
- 结果不确定性
- 结果不可再现性

硬件同步机制

信号量机制

管程机制

任务：对多个相关进程在执行次序上进行协调，使并发执行的诸进程之间按照一定的规则(或时序)共享系统资源，并能很好地相互合作，从而使程序的执行具有可再现性

2.4.2 硬件同步机制

◆ 硬件同步机制

- 特殊的硬件指令解决临界区问题，在对临界区进行管理时，可以将标志看做一个锁，“锁开”进入，“锁关”等待，初始时锁是打开的。每个要进入临界区的进程必须先对锁进行测试，锁未开时等待，锁打开时立即把其锁上，以阻止其它进程进入临界区。为防止多个进程同时测试到锁为打开的情况，测试和关锁操作必须是连续的，不允许分开进行。

- **关中断**
- **利用test-and-set指令实现互斥**
- **利用swap指令实现进程互斥**

2.4.2 硬件同步机制

◆ 关中断

- **方法：**实现互斥的最简单的方法之一。每个进程在进入临界区之后关闭所有中断,在离开临界区之前才重新打开中断。
- **原理**：由于禁止中断，时钟中断也被禁止，这样就不会把CPU切换到另外的进程。不必担心其它进程对它的干扰。
- **缺点**
  - 滥用关中断权力可能导致严重后果；
  - 关中断时间过长，会影响系统效率，限制了处理器交叉执行程序的能力；
  - 关中断方法也不适用于多CPU系统，因为在一个CPU上关中断并不能防止进程在其它CPU上执行相同的临界段代码。

2.4.2 硬件同步机制

◆Test-and-Set指令实现互斥（原语不可分割）

- 是**硬件**提供的一种**原子操作**，允许进程检查共享变量的值并将其原子地更新为另一个值
- 通过这种方式，Test-and-set能够有效地用于实现锁机制，确保在多个进程或线程竞争同一资源时，只有一个进程能够进入临界区
- 该原子操作通常由**硬件直接支持**，能够避免竞争条件(race condition)和确保互斥
- Test-and-set 操作通常有两个步骤，这**两个步骤在硬件层面被保证是原子**的，也就是说，其他任何进程或线程在执行 Test-and-set 操作时，不会干扰当前操作。
  - 测试(Test)：检查一个共享变量的当前值
  - 设置(Set)：将该共享变量的值设置为一个新值

2.4.2 硬件同步机制

register1:=counter;
register1:=register1+1;
counter:=register1;

register2:=counter;
register2:=register2-1;
counter:=register2;

◆Test-and-Set指令实现互斥（原语不可分割）

- 进程在进入临界区之前，首先用Test-and-Set指令测试lock，如果为FALSE则表示可进入，并将TRUE值赋值给lock；

```
function TestAndSet(boolean *lock) {
    boolean original = *lock;  // 读取当前锁状态
    *lock = true;              // 将lock设置为true，表示临界区已被占用
    return original;           // 返回原值
}
```

TS 的原子性避免了以下竞态条件：线程 A 读取 lock == false，准备设置 lock = true。线程 B 在线程 A 设置前也读取到 lock == false，导致两者同时进入临界区。

```
boolean lock=false;   //锁的状态，初始时没有进程占用临界区
void enter critical section(){
    while(TestAndset(&lock)==true){
        //如果 lock 已经是 true，表示临界区已经被占用，进程需要等待
    }
    //进入临界区，执行关键代码
}
void leave critical section(){
    lock = false;//离开临界区，释放锁
}
```

忙等待：未获取锁的线程会持续循环检查锁状态，直到锁被释放。

优点：响应速度快，适合短临界区和多核处理器。

锁释放的显式操作：退出临界区时必须显式将 lock 置为 false，否则其他线程将永久等待。

2.4.2 硬件同步机制

◆ 利用Swap指令实现互斥

➢ 可简单有效地实现互斥，为每一个临界资源设置一个**全局布尔变量lock**，初值为false

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

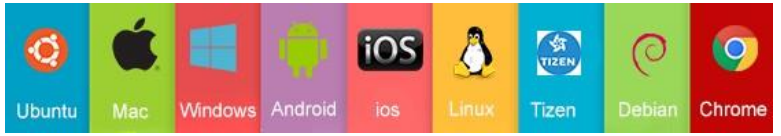
```
int lock=false;   //锁的状态，初始时没有进程占用临界区
void enter critical section(int *lock){
    int key = true;
    do{
        swap(&key, lock)  //使用swap指令尝试获取锁
    }while(key!=false)  //如果key不为false，锁被占用，继续等待
}
//进入临界区，执行关键代码
void leave critical section(){
    *lock = false;//离开临界区，释放锁
}
```

2.4.2 硬件同步机制

◆ TS指令和Swap指令的缺点

- 利用上述硬件指令虽能有效地实现进程互斥，但当临界资源忙碌时，其它访问进程必须不断地进行测试，处于一种“忙等”状态，不符合“让权等待”的原则，造成处理机浪费，同时也很难将它们用于解决复杂的进程同步问题。

  - ✓ 空闲让进
  - ✓ 有限等待
  - ✓ 忙则等待
  - × 让权等待



- 进程
- 多道程序并发执行
- 资源利用率
- 系统吞吐量
- 系统更加复杂

进程同步的基本概念

多道程序系统
进程同步机制

- 进程运行管理
- 对系统资源的无序争夺
- 系统混乱
- 结果不确定性
- 结果不可再现性

硬件同步机制

信号量机制

管程机制

任务：对多个相关进程在执行次序上进行协调，使并发执行的诸进程之间按照一定的规则(或时序)共享系统资源，并能很好地相互合作，从而使程序的执行具有可再现性

进程同步机制

- **硬件同步机制**
  - 关中断
  - 利用test-and-set指令实现互斥
  - 利用swap指令实现进程互斥
- **软件方式**
  - 锁机制 **基本思想**
  - 信号量机制（PV操作） **重中之重**

临界区访问的方式

◆ 锁机制

- **基本原理**：设置一个标志锁W，表明临界区“可用”（0）还是“不可用”（1）？
- “上锁”：进入临界区前，检查标志锁W是否“可用”
  - 若为“不可用”状态，进程在临界区外等待
  - 若为“可用”状态：
    - i) 将标志锁W设置为“不可用”；
    - ii) 访问临界资源；
- “开锁”：退出临界区前将标志锁W设置为“可用”状态

临界区访问的方式

- **用锁访问临界区举例**

```
w=0;  //可用
w=1; //不可用
上锁
lock(w):
    while(w==1);
    w=1;
开锁
unlock(w):
    w=0;
```

临界区访问的方式

- **用锁访问临界区举例**

```
w=0;  //可用
w=1; //不可用
上锁
lock(w):
    while(w==1);
    w=1;
开锁
unlock(w):
    w=0;
```

```
程序A:
…
i=100;
…
print(i)
…
```

```
程序B:
…
i=200;
…
print(i)
…
```

临界区访问的方式

- **用锁访问临界区举例（int w=0）**

```
w=0;  //可用
w=1; //不可用
上锁
lock(w):
    while(w==1);
    w=1;
开锁
unlock(w):
    w=0;
```

```
程序A:
…
lock(w);
i=100;
…
print(i)
unlock(w)
…
```

```
程序B:
…
lock(w);
i=200;
…
print(i)
unlock(w)
…
```

临界区访问的方式

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

◆ 用锁访问临界区举例（int w=0）

```
w=0;  //可用
w=1; //不可用
上锁
lock(w):
    while(w==1);
    w=1;   不可分割
开锁
unlock(w):
    w=0;
```

```
程序A:
…
lock(w);
i=100;
…
print(i)
unlock(w)
…
```

```
程序B:
…
lock(w);
i=200;
…
print(i)
unlock(w)
…
```

临界区访问的方式

- ✓ 空闲让进
- ✓ 忙则等待
- ✓ 有限等待
- × 让权等待

◆ 用锁访问临界区举例（int w=0）

```
w=0;  //可用
w=1; //不可用
上锁
lock(w):
    while(w==1);
    w=1;   不可分割
开锁
unlock(w):
    w=0;
```

```
程序A:
…
lock(w);
i=100;
…
print(i)
unlock(w)
…
```

```
程序B:
…
lock(w);
i=200;
…
print(i)
unlock(w)
…
```

2.4.3 信号量机制

◆ 原语操作（原子操作）

- 原语的实现是由底层所支持的
- 原语是机器指令的延伸，往往是为完成某些特定的功能而编制的一段系统程序。
- **原语操作**也称做“**原子操作**”(atomic action)，即一个操作中的所有动作要么全做要么全不做。
- 原语操作是**不允许并发**的，代码通常较短。
- PV操作、上锁等都是原子操作

2.4.2 硬件同步机制

◆ 硬件同步机制

- 特殊的硬件指令解决临界区问题，在对临界区进行管理时，可以将标志看做一个锁，“锁开”进入，“锁关”等待，初始时锁是打开的。每个要进入临界区的进程必须先对锁进行测试，锁未开时等待，锁打开时立即把其锁上，以阻止其它进程进入临界区。为防止多个进程同时测试到锁为打开的情况，测试和关锁操作必须是连续的，不允许分开进行。

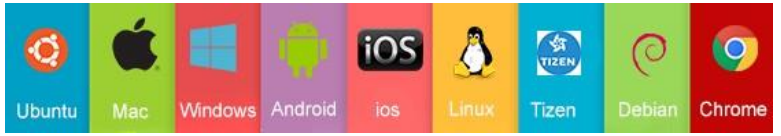
- **关中断**

- **利用test-and-set指令实现互斥**

- **利用swap指令实现进程互斥**

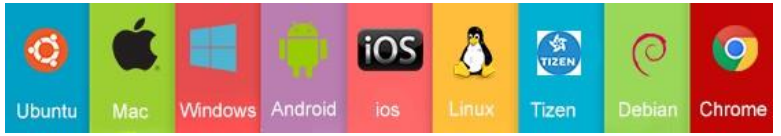
✓ 空闲让进　　✓ 忙则等待

✓ 有限等待　　× 让权等待



进程的描述和控制

- 前趋图和程序执行
- 进程的描述
- 进程控制
- 进程同步 ★
- 经典进程的同步问题 ★
- 进程通信
- 线程的基本概念
- 线程的实现



- 进程
- 多道程序并发执行
- 资源利用率
- 系统吞吐量
- 系统更加复杂

进程同步的基本概念

多道程序系统
进程同步机制

- 进程运行管理
- 对系统资源的无序争夺
- 系统混乱
- 结果不确定性
- 结果不可再现性

硬件同步机制

信号量机制

管程机制

任务：对多个相关进程在执行次序上进行协调，使并发执行的诸进程之间按照一定的规则(或时序)共享系统资源，并能很好地相互合作，从而使程序的执行具有可再现性

进程同步机制

- **硬件同步机制**
  - 关中断
  - 利用test-and-set指令实现互斥
  - 利用swap指令实现进程互斥
- **软件方式**
  - 锁机制 **基本思想**
  - 信号量机制（PV操作） **重中之重**

2.4.3 信号量机制

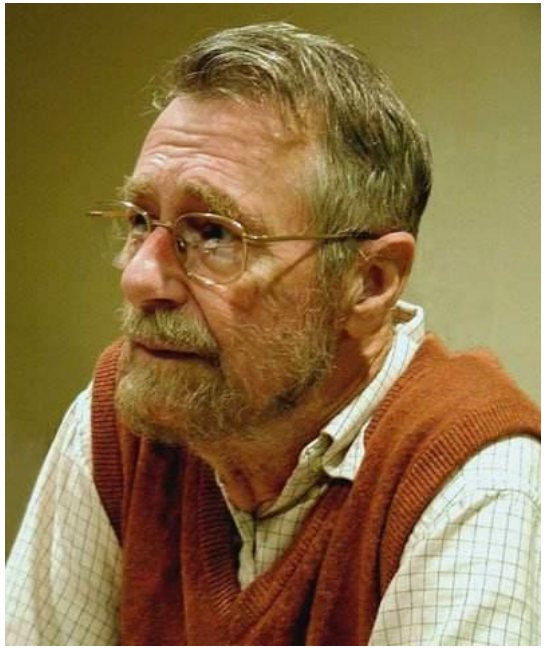
◆ 原语操作（原子操作）

- 原语的实现是由底层所支持的
- 原语是机器指令的延伸，往往是为完成某些特定的功能而编制的一段系统程序。
- **原语操作**也称做“**原子操作**”(atomic action)，即一个操作中的所有动作要么全做要么全不做。
- 原语操作是**不允许并发**的，代码通常较短。
- PV操作、上锁等都是原子操作

2.4.3 信号量机制

◆ 信号量机制

- **解决并发进程同步的工具**
- **整型信号量最初由Dijkstra把整型信号量定义为一个用于表示资源数目的整型量S，它与一般整型量不同，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) wait(S)和 signal(S)来访问**
- **很长时间以来，这两个操作一直被分别称为P、V操作。**
  - 荷兰语 “Proberen” 的意思为“检测”，用P操作表示；
  - 荷兰语 “Verhogen” 的意思为“增量”，用V操作表示。“增量”即增加一个可以使用的临界资源，也就是“归还”的意思



2.4.3 信号量机制

◆ 信号量机制

- **P操作：表示同步进程发出的检测信号量操作，检测是否能够使用临界资源**
  - 如果能使用，则检测通过，进程可以访问临界资源；
  - 如果不能使用，则等待，直到访问临界区的进程将临界资源归还后，等待的进程才能够访问临界资源。
- **V操作：表示访问完临界资源的进程通知等待进程已经完成了临界资源的访问，将临界资源释放**
- **信号量机制强调P操作和V操作都是原子操作，执行时不可中断**

2.4.3 信号量机制

- **信号量机制**
  - 整型信号量
  - **记录型信号量（结构型信号量）**
  - AND型信号量
  - 信号量集

2.4.3 信号量机制

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

◆ 整型信号量

- **P操作**：表示同步进程发出的检测信号量操作，检测是否能够使用临界资源
- **V操作**：表示访问完临界资源的进程通知等待进程已经完成了临界资源的访问，将临界资源释放
- **P操作**

```
wait(S){
    while S≤0；  // do no-op
    S: = S - 1;}
```

- **V操作**

```
signal(S){
     S: = S + 1; }
```

其中，S为一个需要初始化值的正整型量。对S的访问只能通过P、V操作。

- S>0，表示临界资源可以访问，P(S)中的检测语句通过，调用P(S)的进程可以访问临界资源。
- S≤0，表示有进程在访问临界资源，此时临界资源处于忙，调用P(S)的进程只能等待，直到S的值大于0，才可以访问临界资源。

2.4.3 信号量机制

✓ 空闲让进　✓ 忙则等待

✓ 有限等待　× 让权等待

◆ 整型信号量

- **P操作**：表示同步进程发出的检测信号量操作，检测是否能够使用临界资源
- **V操作**：表示访问完临界资源的进程通知等待进程已经完成了临界资源的访问，将临界资源释放
- **P操作**

```
wait(S){
    while S≤0；  // do no-op
    S: = S - 1;}
```

- **V操作**

```
signal(S){
     S: = S + 1; }
```

其中，S为一个需要初始化值的正整型量。对S的访问只能通过P、V操作。

- S>0，表示临界资源可以访问，P(S)中的检测语句通过，调用P(S)的进程可以访问临界资源。
- S≤0，表示有进程在访问临界资源，此时临界资源处于忙，调用P(S)的进程只能等待，直到S的值大于0，才可以访问临界资源。

2.4.3 信号量机制

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

◆ 记录型信号量（结构型信号量）

➢ 增加了一个进程链表指针list，用于链接所有等待的进程

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

➢ P操作

```
wait(semaphore *S)
{
    S.value = S.value - 1
    if S.value<0 then block(S,L)
}
```

➢ V操作

```
signal(S)
{
    S.value = S.value + 1
    if S.value≤0 then wakeup(S,L)
}
```

2.4.3 信号量机制

✓ 空闲让进　✓ 忙则等待
✓ 有限等待　✓ 让权等待

◆ 记录型信号量（结构型信号量）

➢记录型信号量在整型信号量的基础上进行了改进，让不能进入临界区的进程**“让权等待”**，即进程状态由运行转换为阻塞状态，进程进入阻塞队列中等待。

➢P操作

```
wait(S){
    S.value = S.value - 1
    if S.value<0 then block(S,L)
}
```

若信号量S的S.value的值为正数，该正数表示可对信号量可进行的P操作的次数，即实际可以使用的临界资源数

➢V操作

```
signal(S){
    S.value = S.value + 1
    if S.value≤0 then wakeup(S,L)
}
```

若信号量S的S.value的值为负值，表示有多个进程申请临界资源，而又不能得到，在阻塞队列等待。S.value的绝对值表示在阻塞队列等待临界资源的进程个数

2.4.3 信号量机制

✓ 空闲让进　✓ 忙则等待
✓ 有限等待　✓ 让权等待

◆ 记录型信号量（结构型信号量）

- 记录所有**等待**同一资源的进程，解决忙等问题，实现**让权等待**
- 在记录型信号量机制中，S.value的初值表示系统中某类**资源的数目**，因而又称为**资源信号量**。如果S.value的初值为1，表示只允许一个进程访问临界资源——互斥信号量，用于进程互斥。
- 对它的每次wait操作，意味着进程请求一个单位的该类资源，使系统中可供分配的该类资源数减少一个，因此描述为**S.value:=S.value-1**
- 当S.value<0时，表示该类资源已分配完毕，应调用**block原语**，进行自我阻塞，放弃处理机，并插入到信号量链表中，此时S.value的绝对值表示在该信号量链表中已阻塞进程的数目。
- 对信号量的每次signal操作，表示执行进程释放一个单位资源，使系统中可供分配的该类资源数增加一个，故S.value:=S.value+1操作表示资源数目加1。若加1后仍是S.value≤0，则表示在该信号量链表中，仍有等待该资源的进程被阻塞，故还应调用**wakeup原语**，将S.L链表中的第一个等待进程唤醒。

2.4.3 信号量机制

◆ 整型信号量与记录型信号量的比较

- 整型信号量机制中的wait操作，只要信号量S≤0，就会不断地测试
- 整型信号量机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态
- 记录型信号量机制则是一种不存在“忙等”现象的进程同步机制
- 但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况
- 为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表指针L，用于链接上述的所有等待进程
- 记录型信号量是因采用了记录型的数据结构而得名

2.4.3 信号量机制

◆ AND型信号量

- 指同时需要**多种资源**且每种占用一个时的信号量操作；
- AND型信号量集的基本思想：在一个原语中申请整段代码需要的**多个临界资源**，要么**全部分配给它**，要么一**个都不分配**；
- AND型信号量集**P原语**为**Swait：**

```
Swait(S1, S2, …, Sn){
    While(TRUE){
        if (S1 >=1 and … and Sn>=1 ){
            for( i=1;i<=n; i++)    Si--;
            break;
        }else{
            Place the process in the waiting queue associated  with the first Si found with Si < 1,and
            set the progress count of this process to the beginning of Swait operation
        }}}
```

2.4.3 信号量机制

◆ AND型信号量

- 指同时需要**多种资源**且每种占用一个时的信号量操作；
- AND型信号量集的基本思想：在一个原语中申请整段代码需要的**多个临界资源**，要么**全部分配给它**，要么一**个都不分配**；
- AND型信号量集**V原语**为**Ssignal：**

```
Ssignal(S1, S2, …, Sn){
        while(TRUE){
            for (i=1; i<=n; i++)
            {
                Si++ ;
                Remove all the process waiting in the queue associated
                with Si into the ready queue}}}
```

2.4.3 信号量机制

◆ AND型信号量

- AND同步机制的基本思想是：将进程在整个运行过程中需要的所有资源，**一次性全部地分配给进程**，待进程使用完后再**一起释放**。
- 只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源也不分配给它。
- 亦即，对若干个临界资源的分配，采取**原子操作方式**：要么把它所请求的资源全部分配到进程，要么一个也不分配。由死锁理论可知，这样就可避免上述死锁情况的发生。
- 为此，在**wait操作中**，增加了一个“**AND**”条件，故称为**AND同步**，或称为**同时wait操作**。

2.4.3 信号量机制

◆ 信号量集

- 当进程需要**申请的临界资源种类较多**，**每类临界资源个数较多**时，如果用记录型信号量，进程每次只能一次申请或释放一个临界资源，非常麻烦。**因此，引入信号量集。**
- 在每次分配之前，测试**资源数量是否大于可分配的下限值**

  Swait(S1, t1, d1; ...; Sn, tn, dn);　（t：分配下限；d：需求值）

  Ssignal(S1, d1; ...; Sn, dn);
- 一般“信号量集”可以用于各种情况的资源分配和释放。下面是几种特殊的情况：

  1) **Swait(S, d, d)**表示每次申请d个资源，当资源数量少于d个时，便不予分配。

  2) **Swait(S, 1, 1)**表示互斥信号量。

  3) **Swait(S, 1, 0)**可作为一个可控开关(当S≥1时，允许多个进程进入临界区；当S=0时禁止任何进程进入临界区)。
- 由于一般信号量在使用时的灵活性，因此通常并不成对使用Swait和Ssignal。为了避免死锁可一起申请所有需要的资源，但**不一起释放**

2.4.3 信号量机制

◆ 信号量机制的对比

- **整型信号量**
  - 基本信号量原理
- **记录型信号量**
  - 多个进程申请同一类资源
- **AND型信号量**
  - 同一进程申请多个不同资源
- **信号量集**
  - 同一进程申请多个同类资源
  - 多个进程申请多个不同资源

◆ 信号量机制应用

- **信号量按联系进程的关系分成二类：**
  - **公用信号量（互斥信号量）**
  - **私用信号量（同步信号量）**

2.4.4 信号量机制应用

◆ 信号量机制应用

- ➢ 信号量按联系进程的关系分成二类：
  - 公用信号量（互斥信号量）

它为一组需的并发进程而设置，代表共享的临界资源，每个进程均可对它施加P、V操作，即都可申请和释放**互斥共享临界资源**该临界资源，其初始值置为1。取值意义如下：

- ➢ 信号量s＝1 ；表示该资源空闲，可供使用
- ➢ 信号量s＝0 ；表示该资源已被占用，无其它进程等待
- ➢ 信号量s＝-n ；表示该资源已被占用，还有n个进程因等待该资源而阻塞。

2.4.4 信号量机制应用

◆ 信号量机制应用

- 信号量按联系进程的关系分成二类：
  - 公用信号量（互斥信号量）
  - 私用信号量（同步信号量）

它为一组需同步协作完成任务的并发进程而设置，只有拥有该资源的进程才能对它施加P操作（即可申请资源），而由其合作进程对它施加V操作（即释放资源）。

2.4.4 信号量机制应用

◆ 整型信号量实现进程互斥

➢ 进程A和进程B共用缓冲区，如图所示。进程A和进程B并发执行，**缓冲区是竞争访问的临界资源**。用整型信号量实现进程同步。

- 为缓冲区设置整型信号量/互斥信号量mutex;
- 设mutex的初值为1，取值范围（-1，0，1）
- 将各进程对临界区访问置于P(mutex)和V(mutex)之间

2.4.4 信号量机制应用

◆ 整型信号量实现进程互斥

➢ 进程A和进程B共用缓冲区

```
P(A):
   begin
        while(1)
        {
           P(mutex);
           临界区;
           V(mutex);
        }
    end;
```

➢ **P操作**

```
wait(S){
    while S≤0;
    S: = S - 1;
}
```

➢ **V操作**

```
signal(S){
     S: = S + 1;
}
```

```
P(B):
   begin
        while(1)
        {
           P(mutex);
           临界区;
           V(mutex);
        }
    end;
```

2.4.4 信号量机制应用

◆ 整型信号量实现进程互斥

- **对于并发进程A和进程B：**如果两个进程都能够访问缓冲区，哪个进程先访问缓冲区，关键在于哪个进程先执行P(mutex)。先执行的进程先进入缓冲区访问；后执行P(mutex)的进程等待，直到先进入缓冲区访问的进程用V(mutex)释放缓冲区为止。

- **P操作和V操作必须成对出现**，缺少P操作将导致系统混乱，不能保证临界资源的互斥访问；缺少V操作将会使临界资源永远不被释放，使等待该资源而阻塞的进程不能被唤醒。

2.4.4 信号量机制应用

◆ 整型信号量实现进程同步

- 输入进程I和计算进程P之间共用缓冲区，且缓存区只能装入一条数据时，用整型信号量实现输入进程I和计算进程P的同步。

设置两个整型信号量is（初值为1）和ps（初值为0）
用于输入进程I和计算进程P协调访问公共缓冲区

2.4.4 信号量机制应用

◆ 整型信号量实现进程同步

➢ 输入进程和计算进程共用的缓冲区中只能装入一条数据

```
P(I):       /* 输入进程 */
  begin
    while(1)
    {
      …
      P(is);   /* 初值为1 */
      将一批输入数据放入缓冲区;
      V(ps);
    }
  end;
```

➢ **P操作**

```
wait(S){
    while S≤0;
    S: = S - 1;
}
```

➢ **V操作**

```
signal(S){
    S: = S + 1;
}
```

2.4.4 信号量机制应用

◆ 整型信号量实现进程同步

➢ 输入进程和计算进程共用的缓冲区中只能装入一条数据

```
P(P):        /* 计算进程  */
  begin
    while(1)
    {
      P(ps);   /* 初值为0  */
      从缓冲区中拿出一批数据;
      V(is);
      计算;
      …
    }
  end;
coend;
```

➢ **P操作**

```
wait(S){
    while S≤0;
    S: = S - 1;
}
```

➢ **V操作**

```
signal(S){
    S: = S + 1;
}
```

2.4.4 信号量机制应用

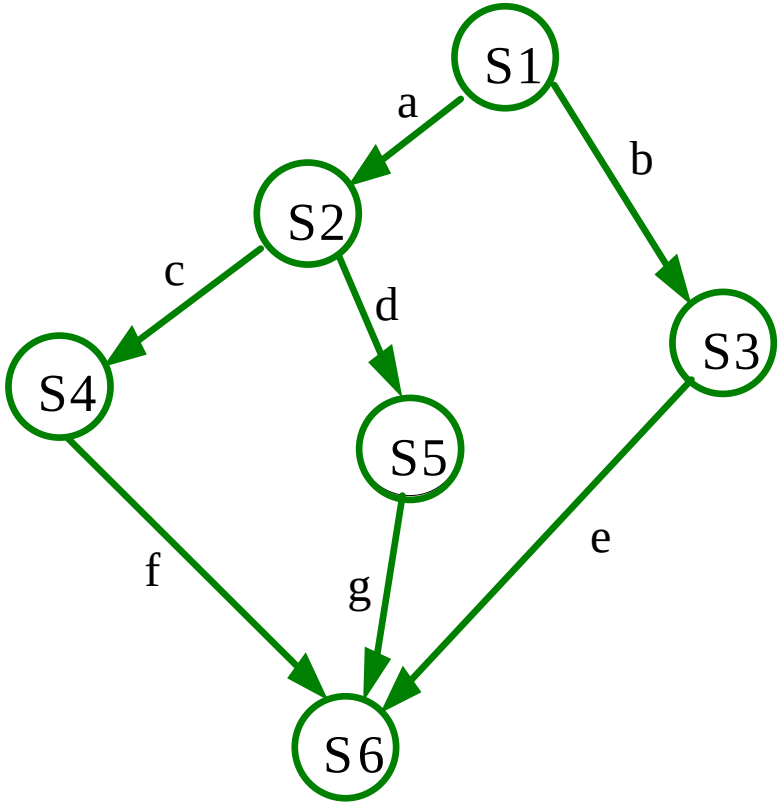
◆ 整型信号量实现进程同步

- **信号量的设置是关键：**将信号量is的初值设置为1，ps的初值设置为0。总是**输入进程先进入缓冲区放入数据，计算进程先等待**。输入进程将一条数据放入缓冲区后释放ps，计算进程才能进入缓冲区取走一条数据。
- **对于输入进程：**如果计算进程没有进入缓冲区取走一条数据，输入进程不能再进入缓冲区放数据，因为输入进程需要计算进程释放缓冲区is。
- **对于计算进程：**如果没有输入进程释放缓冲区ps，计算进程不能多次连续进入缓冲区取走数据。

2.4.4 信号量机制应用

◆ 整型信号量实现进程前趋图

- 通过共享一个**公用信号量S**，并赋予**初值为0**，实现进程之间的前驱关系
- 如：将signal(X)操作放在进程S1后面，wait(X)放在进程S2前面，就可以实现S1→S2的前趋关系

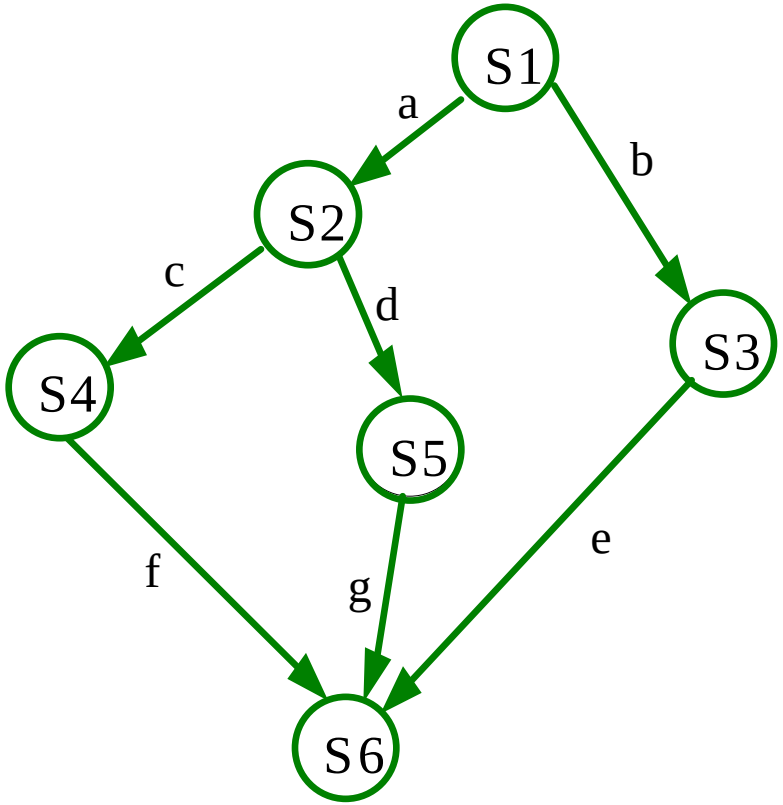


2.4.4 信号量机制应用

◆ 整型信号量实现进程前趋图

a,b,c,d,e,f,g:semaphore : = 0,…,0

begin S_1; V(a); V(b); end;



2.4.4 信号量机制应用

◆ 整型信号量实现进程前趋图

a,b,c,d,e,f,g:semaphore : = 0,…,0

begin S_1; V(a); V(b); end;

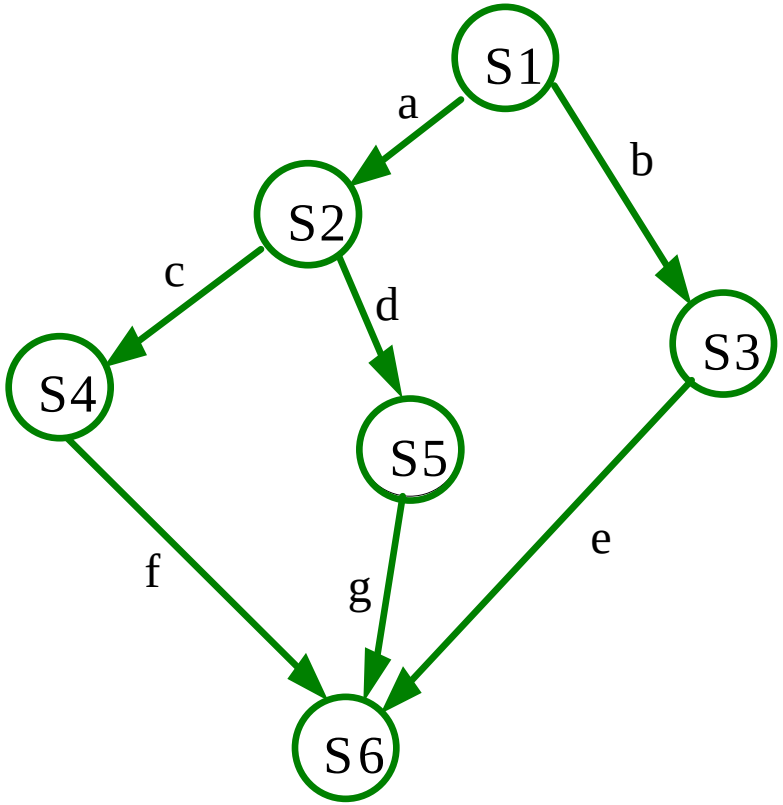
begin P(a); S_2; V(c); V(d); end;

begin P(b); S_3; V(e); end;

begin P(c); S_4; V(f); end;

begin P(d); S_5; V(g); end;

begin P(e); P(f); P(g); S_6 ;end;



2.4.4 信号量机制应用

◆ 整型信号量实现进程前趋图

a,b,c,d,e,f,g:semaphore : = 0,…,0

begin S_1; signal(a); signal(b); end;

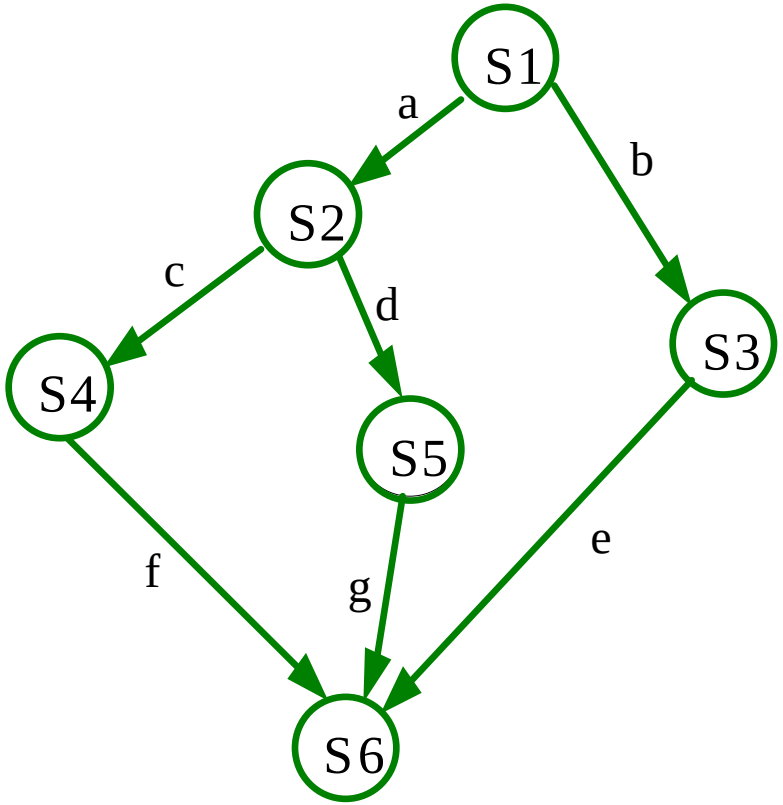
begin wait(a); S_2; signal(c); signal(d); end;

begin wait(b); S_3; signal(e); end;

begin wait(c); S_4; signal(f); end;

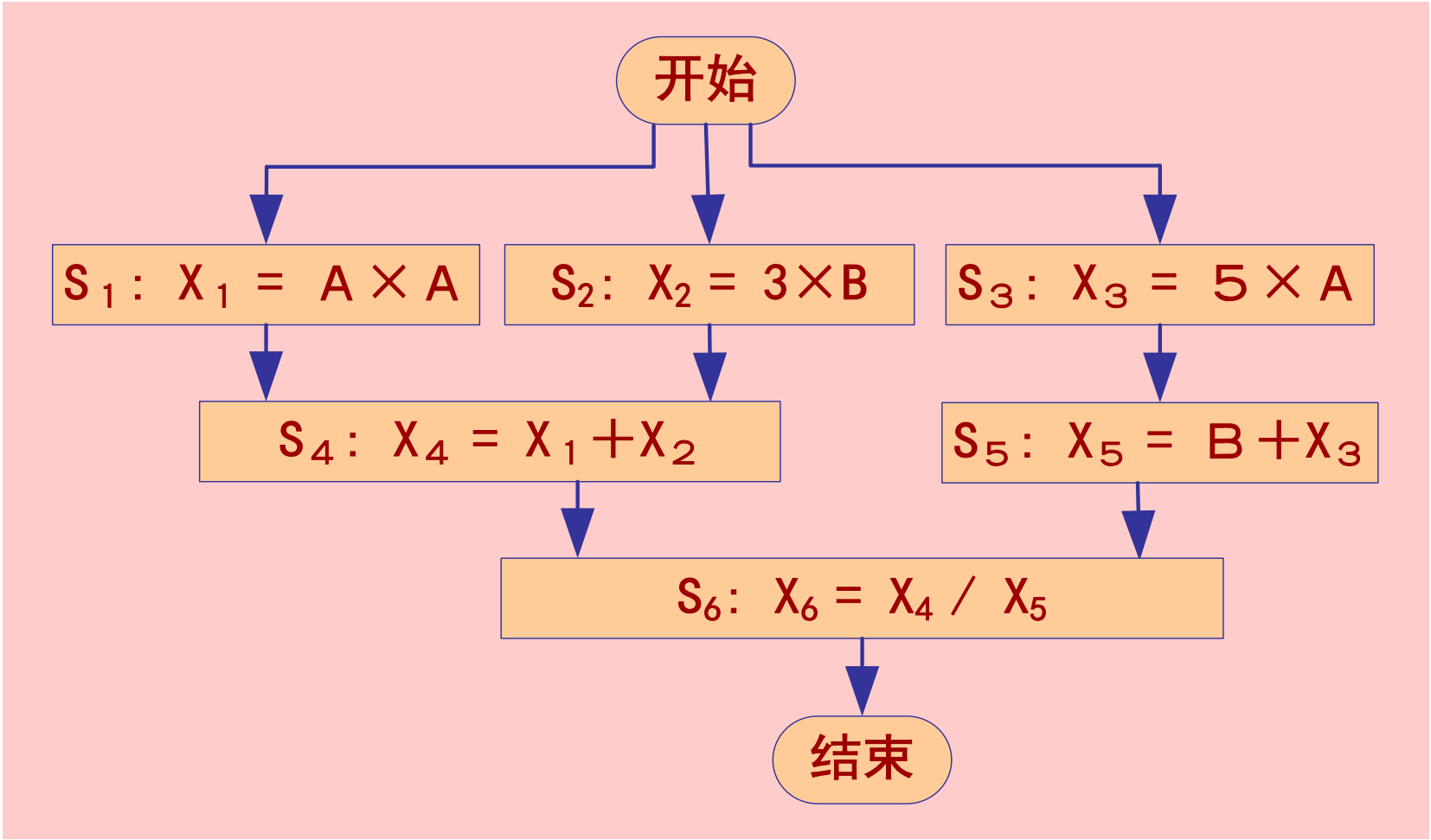
begin wait(d); S_5; signal(g); end;

begin wait(e); wait(f); wait(g); S_6 ;end;



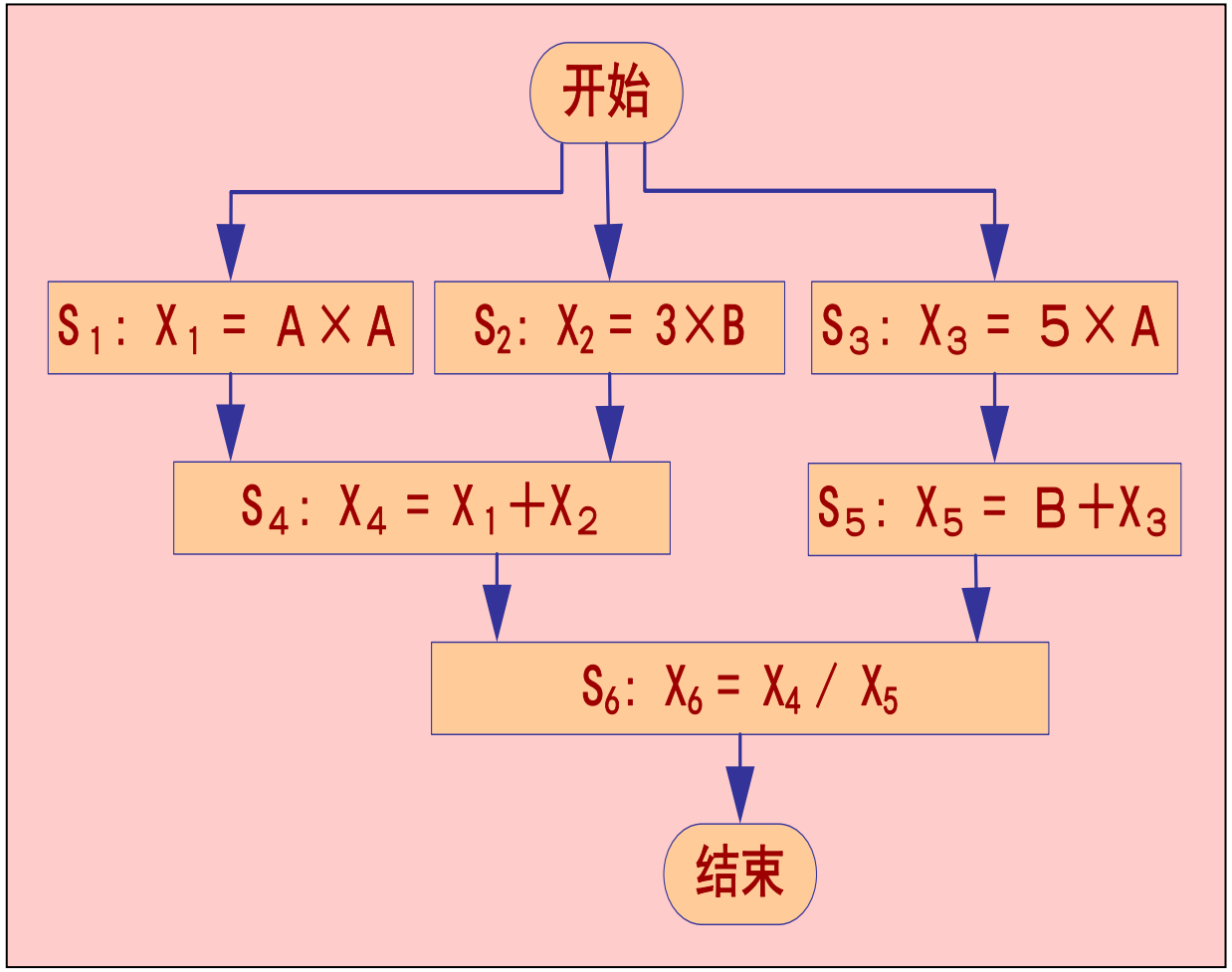
2.4.4 信号量机制应用

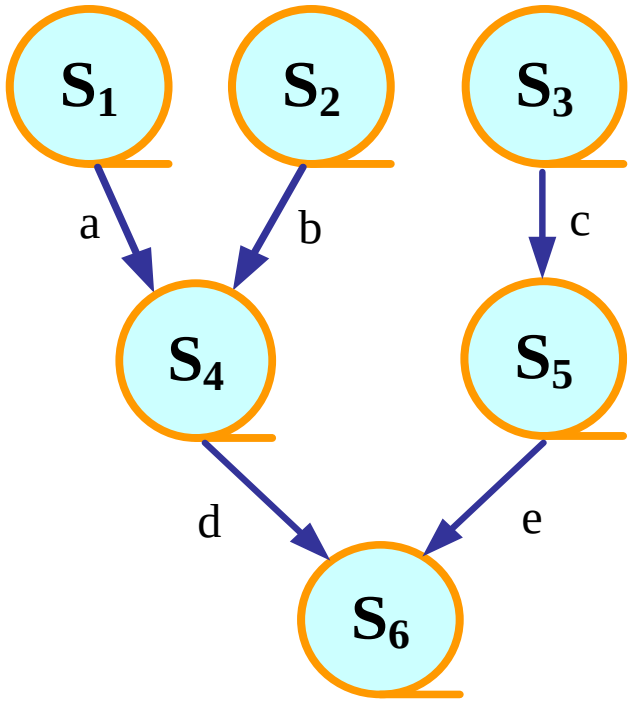
◆ 信号量应用：补充例题1



2.4.4 信号量机制应用

◆ 信号量应用：补充例题1





2.4.4 信号量机制应用

◆ 信号量应用：补充例题1

a,b,c,d,e:semaphore : = 0,0,0

S1: { ... V(a);}

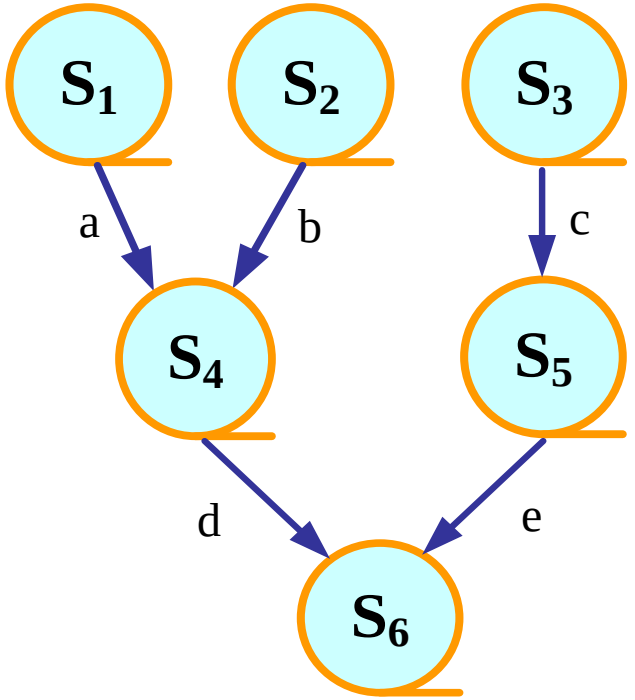
S2: { ... V(b);}

S3: { ... V(c);}

S4: { P(a); P(b); ... V(d);}

S5: { P(c); ... V(e);}

S6: { P(d); P(e); ... }



2.4.4 信号量机制应用

◆ 信号量应用：补充例题3

思路：

- **P1和P2具有相同的优先级，并发，具有顺序的不确定性；**
- **信号量M1和M2使得语句的执行顺序具有了制约关系。**
- **M1和M2初始值为0**

绘制对应的前趋图

进程P1

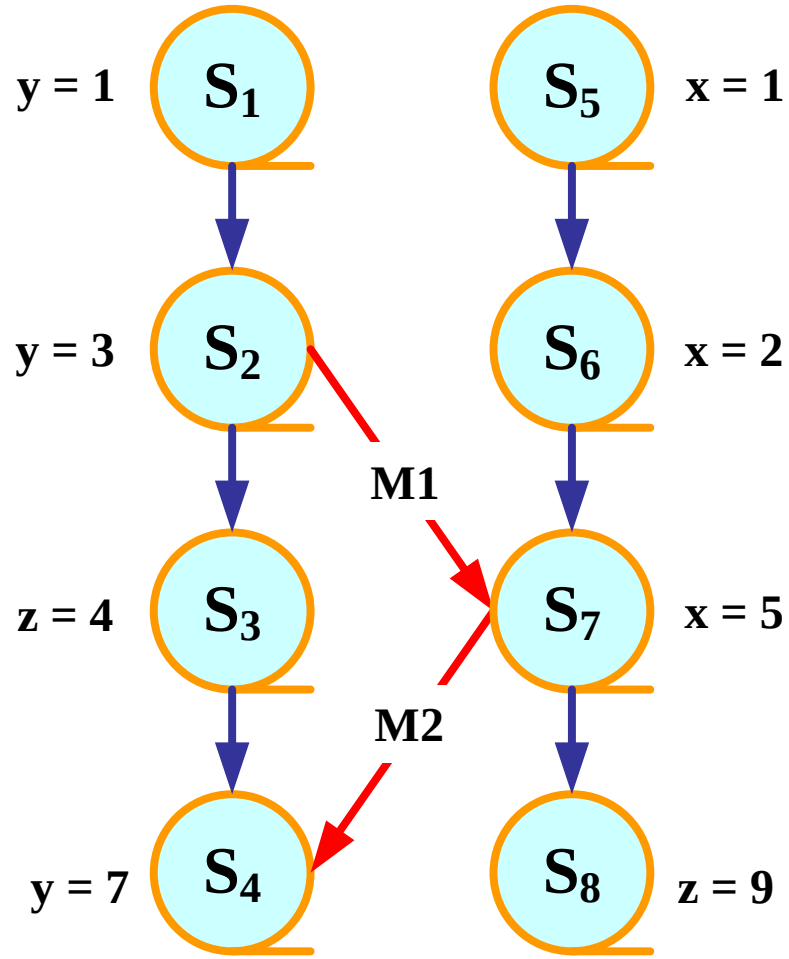
```
1: y = 1;
2: y = y + 2;
V(M1);
3: z = y + 1;
P(M2);
4: y = z + y;
```

进程P2

```
5: x = 1;
6: x = x + 1;
P(M1);
7: x = y + x;
V(M2);
8: z = x + z;
```

2.4.4 信号量机制应用

◆ 信号量应用：补充例题3



x = 5; y = 7; z = 9

进程P1

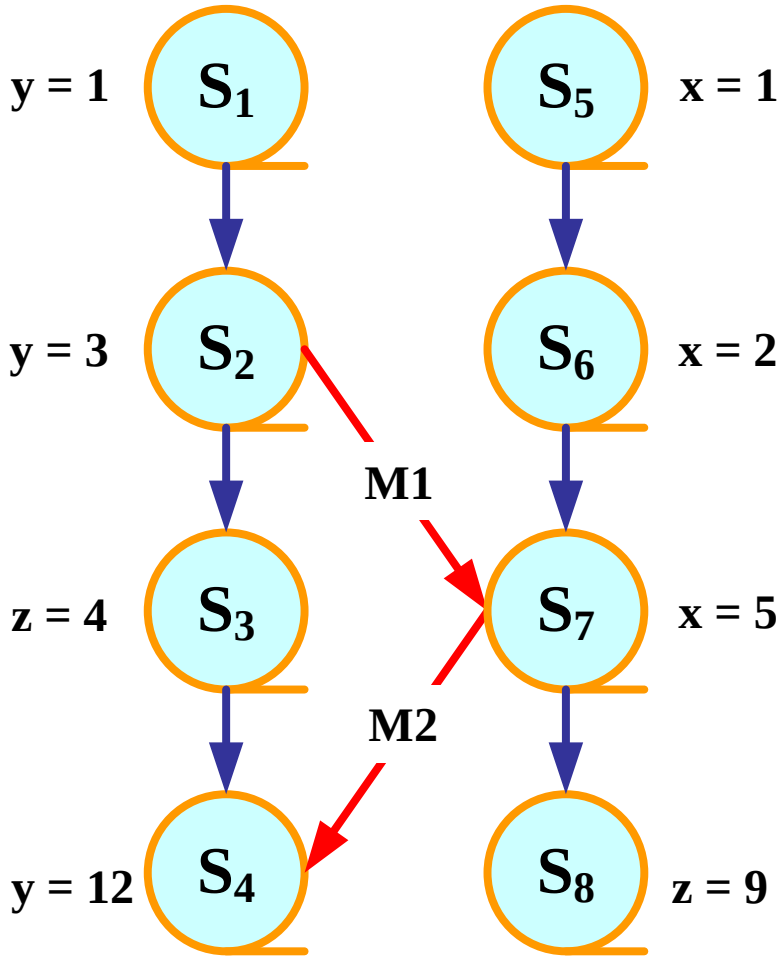
```
y = 1;
y = y + 2;
V(M1);
z = y + 1;
P(M2);
y = z + y;
```

进程P2

```
x = 1;
x = x + 1;
P(M1);
x = y + x;
V(M2);
z = x + z;
```

2.4.4 信号量机制应用

◆ 信号量应用：补充例题3



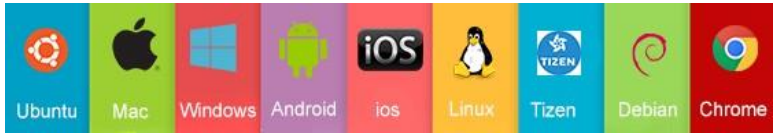
x = 5; y = 12; z = 9

进程P1

```
y = 1;
y = y + 2;
V(M1);
z = y + 1;
P(M2);
y = z + y;
```

进程P2

```
x = 1;
x = x + 1;
P(M1);
x = y + x;
V(M2);
z = x + z;
```



- 进程
- 多道程序并发执行
- 资源利用率
- 系统吞吐量
- 系统更加复杂

进程同步的基本概念

多道程序系统
进程同步机制

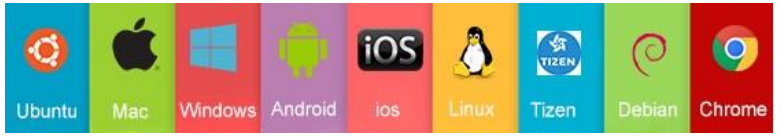
- 进程运行管理
- 对系统资源的无序争夺
- 系统混乱
- 结果不确定性
- 结果不可再现性

硬件同步机制

信号量机制

管程机制

任务：对多个相关进程在执行次序上进行协调，使并发执行的诸进程之间按照一定的规则(或时序)共享系统资源，并能很好地相互合作，从而使程序的执行具有可再现性



- 进程
- 多道程序并发执行
- 资源利用率
- 系统吞吐量
- 系统更加复杂

进程同步的基本概念

多道程序系统
进程同步机制

- 进程运行管理
- 对系统资源的无序争夺
- 系统混乱
- 结果不确定性
- 结果不可再现性

每个访问临界资源的进程都必备同步操作P、V，使得大量的同步操作分散在各个进程中，给**系统管理**带来麻烦，还可能导致系统**死锁**。因此，引入新的进程同步工具——**管程（Monitors，监视器）**

管程机制

任务：对多个相关进程在执行次序上进行协调，使并发执行的诸进程之间按照一定的规则(或时序)共享系统资源，并能很好地相互合作，从而使程序的执行具有可再现性

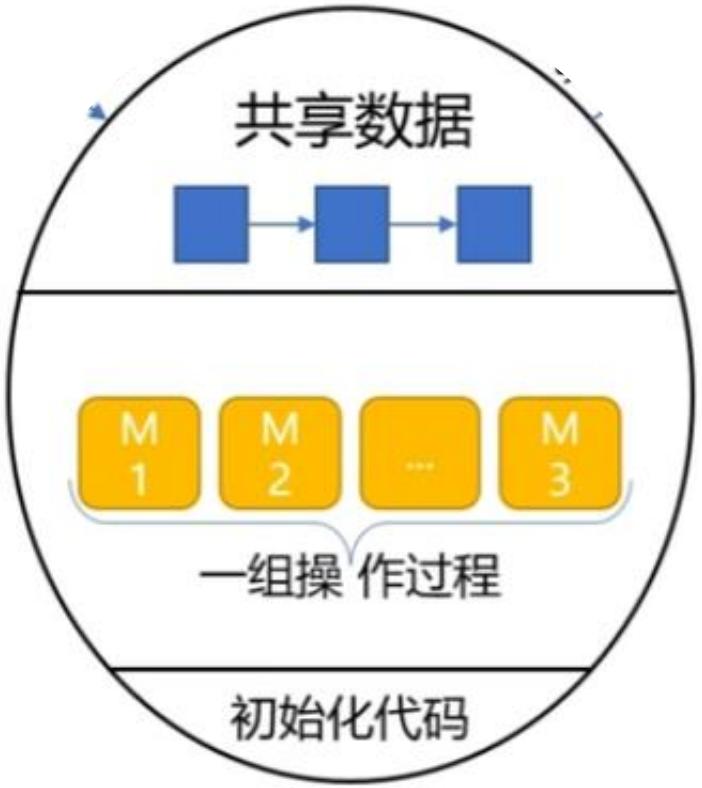
2.4.5 管程机制

◆ 管程的定义（进程同步的管理工具）

- 硬件资源和软件资源均可用数据结构（少量的信息、可表示对资源执行的操作）抽象地描述其资源特性
- 利用共享数据结构抽象地表示系统中的共享资源，并对其数据结构实时的特点操作定义为一组过程（进行PV操作的函数）
- 进程对共享资源的申请、释放和其他操作都需要通过这组过程，间接对共享数据结构实现操作
- 对于并发进程，可以根据资源的情况接受或阻塞，确保每次仅有一个进程进入管程，执行过程、使用共享资源，达到对共享资源所有访问的统一管理，实现进程互斥

2.4.5 管程机制

- **管程——一个操作系统的资源管理模块**
  - 代表共享资源的数据结构
  - 对该共享数据结构实施的一组过程所组成的资源管理程序
- **管程的定义（Hansan）**
  - 一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作所能同步进程和改变管程中的数据
- **管程的组成**
  - 管程的名称：
  - 局部于管程的共享数据结构说明：共享资源
  - 对该数据结构的进行操作的过程：PV操作
  - 对局部于管程的共享数据设置初始值的语句



2.4.5 管程机制

◆ 管程的语法描述

```
Monitor monitor_name    {    /*管程名*/
    share variable declarations;        /*共享变量说明*/
    cond declarations;                  /*条件变量说明*/
    public:                             /*能被进程调用的过程*/
    void P1(……)                         /*对数据结构操作的过程*/
    {……}
    void P2(……)
    {……}
……
    void (……)
    {……}
    ……
    {                                   /*管程主体*/
        initialization code;            /*初始化代码*/
        ……
    }
}
```

◆管程的组成

- 管程的名称
- 局部于管程的共享数据结构说明
- 对该数据结构的进行操作的过程
- 对局部于管程的共享数据设置初始值的语句

◆管程——面向对象的思想

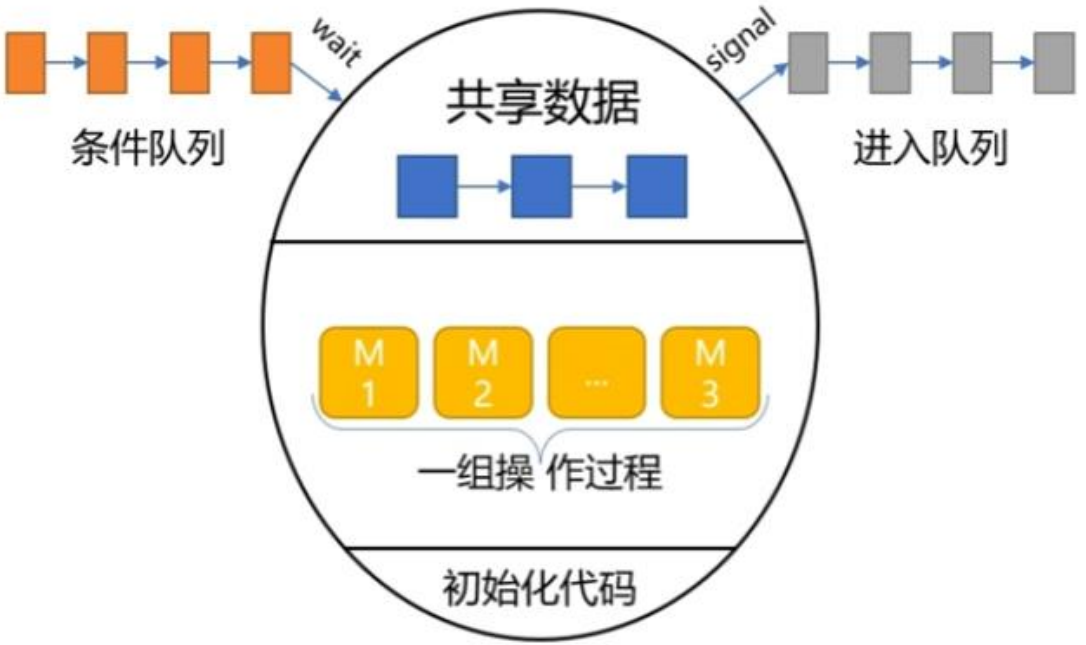
- 共享资源的数据结构、过程、同步机制集中封装在一个对象内部
- 信息屏蔽：封装在管程内部的数据结构仅能被封装于管程内的过程访问
- 所有进程对临界资源的访问都需要通过管程间接访问
- 管程每次只准许一个进程进入管程——进程互斥

2.4.5 管程机制

- **管程——一种程序设计语言的结构成分，和信号量有同等的表达能力**
- **管程特性（语言角度）**
  - **模块化：基本程序单位，可单独编译**
  - **抽象数据类型：数据+对数据的操作**
  - **信息掩蔽：数据结构仅供管程中的过程访问，数据结构及过程的具体实现外部不可见**
- **管程与进程的区别**
  - 数据结构不同：进程是私有数据结构PCB，管程是公共数据结构（如消息队列）
  - 操作不同：进程由顺序程序执行有关操作，管程主要进行同步操作和初始化操作
  - 目的不同：进程未来系统并发性，管程解决共享资源的互斥使用问题
  - 进程是主动工作方式，管程被进程调用以实现对共享数据结构的操作，属于被动工作方式
  - 进程可并发执行，管程不能与其调用者并发
  - 进程具有动态性，管程是一个资源管理模块，供进程调用

2.4.5 管程机制

- **条件变量——在利用管程实现进程同步时，必须设置同步工具**
  - 同步工具：wait和signal（操作原语）
  - 条件变量condition（一种抽象数据类型）
    - 解决因阻塞或挂起导致的资源浪费和其他进程的长时间等待
    - 根据阻塞或挂起的原因不同，可设置多个条件变量
    - 对条件变量的访问只能在管程中进行
    - 只能通过x.wait和x.signal对条件变量进行操作
    - 每个条件变量保存了一个链表，用于记录因该条件变量阻塞的所有进程

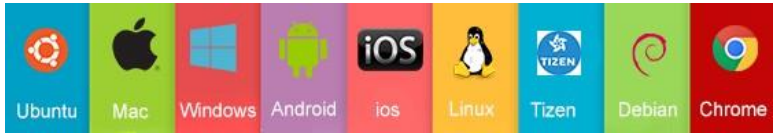


2.4.5 管程机制

➢ 管程条件变量的释放处理方式

- Hansen管程

- Hoare管程

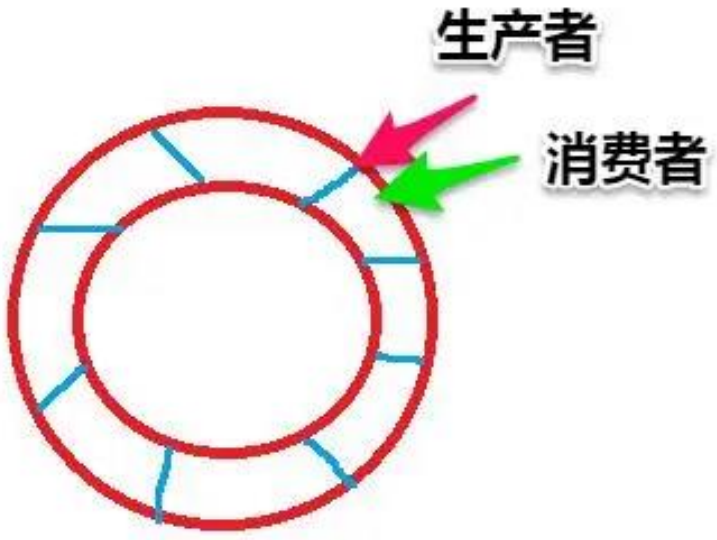


进程的描述和控制

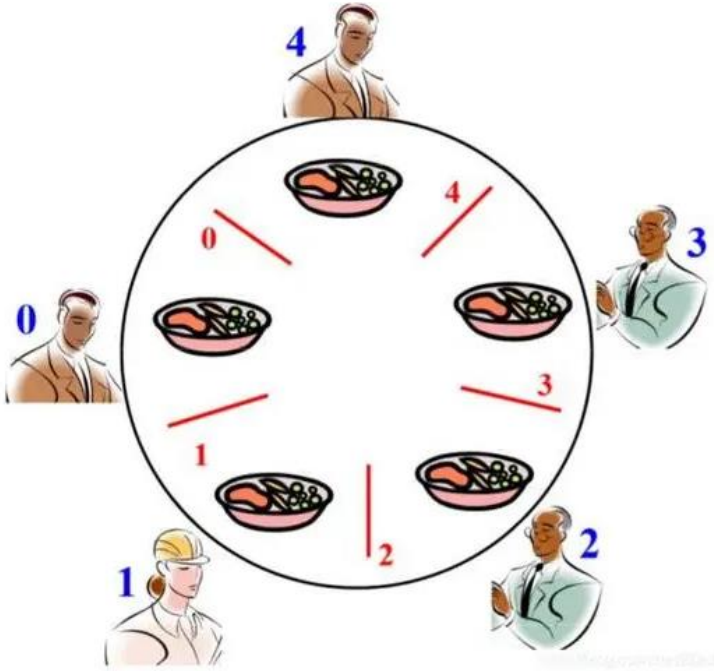
- 前趋图和程序执行
- 进程的描述
- 进程控制
- 进程同步 ★
- 经典进程的同步问题 ★
- 进程通信
- 线程的基本概念
- 线程的实现

经典进程的同步问题

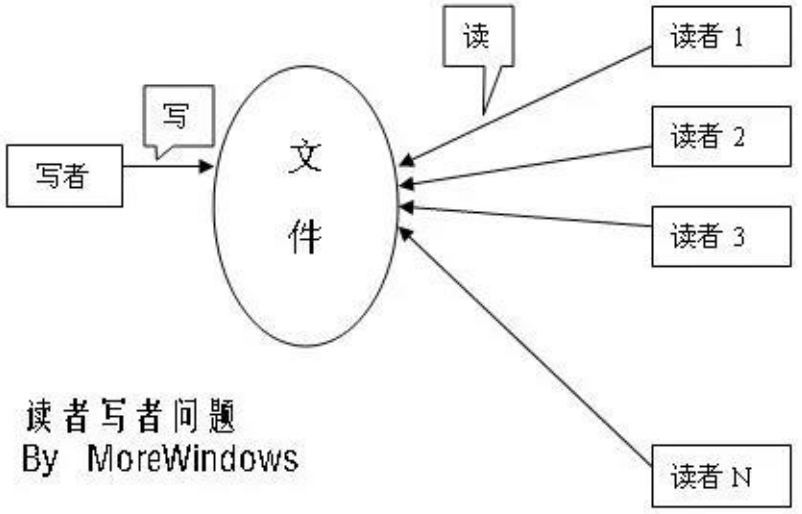
生产者和消费者问题



哲学家就餐问题

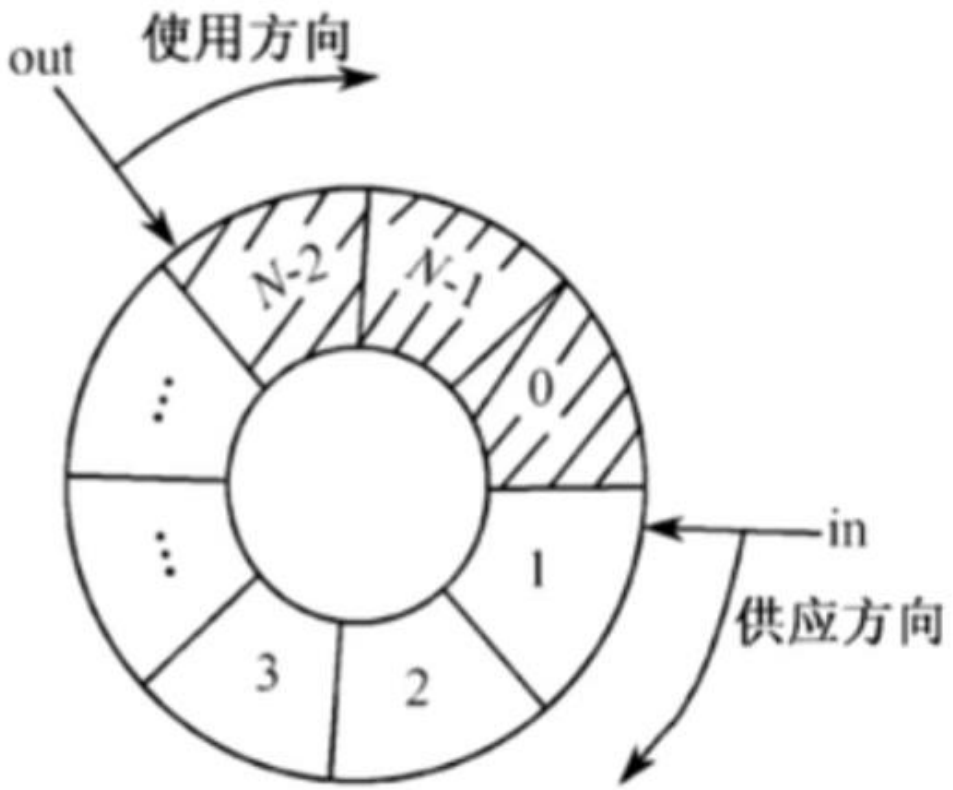


读者和写者问题



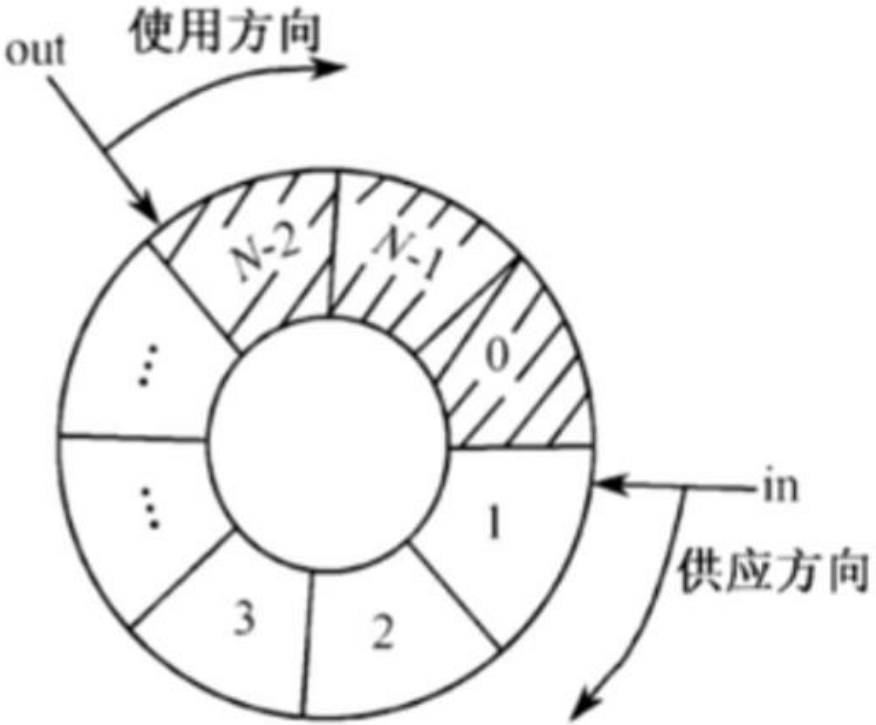
2.5.1 生产者和消费者问题

- 假设缓冲池中有**n**个缓冲区，每个缓冲区存放一个消息；
- 可利用互斥信号量**mutex**使诸进程对缓冲池实现互斥访问；
- 利用**empty**和**full**计数信号量分别表示空缓冲及满缓冲的数量。
- 又假定这些生产者和消费者互相等效，只要缓冲池未满，生产者可将消息送入缓冲池；只要缓冲池未空，消费者可从缓冲池取走一个消息。



2.5.1 生产者和消费者问题

- 互斥信号量
  - mutex：防止多个进程同时进入临界区
- 同步信号量
  - empty和full：保证事件发生的顺序
  - 缓冲区满时，Producer停止运行
  - 缓冲区空时，Consumer停止运行
- 概念差别——互斥与同步（并发的两个要素）
  - 互斥：保护临界区，防止多个进程同时进入
  - 同步：保证进程运行的顺序合理



同步问题：生产者P：P进程不能往“满”的缓冲区中放产品；消费者C：C进程不能从“空”的缓冲区中取产品。

2.5.1 生产者和消费者问题

```
mutex,full,empty:semaphore
mutex :=1;  full:=0;   empty:=n;
```

- **生产者P:**

```
While(True){
    ∘ ∘ ∘
    Buffer(in)=nextp;
    in:=(in+1) mod n;
    ∘ ∘ ∘
}
```

- **消费者C:**

```
While(Ture){
    ∘ ∘ ∘
    netxc=buffer(out);
    out:=(out+1) mod n;
    ∘ ∘ ∘
}
```

2.5.1 生产者和消费者问题

```
mutex,full,empty:semaphore
mutex :=1;  full:=0;   empty:=n;
```

- **生产者P:**

```
While(True){
    P(XXX);
    P(XXX);
    Buffer(in)=nextp;
    in:=(in+1) mod n;
    V(XXX);
    V(XXX);
}
```

- **消费者C:**

```
While(Ture){
    P(XXX)
    P(XXX);
    netxc=buffer(out);
    out:=(out+1) mod n;
    V(XXX);
    V(XXX);
}
```

2.5.1 生产者和消费者问题

```
mutex,full,empty:semaphore
mutex :=1;  full:=0;   empty:=n;
```

- **生产者P:**

```
While(True){
    P(empty);
    P(mutex);
    Buffer(in)=nextp;
    in:=(in+1) mod n;
    V(mutex);
    V(full);
}
```

- **消费者C:**

```
While(Ture){
    P(full)
    P(mutex);
    netxc=buffer(out);
    out:=(out+1) mod n;
    V(mutex);
    V(empty);
}
```

2.4.5 管程机制

- 管程的语法描述（生产者-消费者问题）

```
Monitor producerconsumer {  // 管程名称
    item buffer[N];    // 共享变量说明
    int in,out;
    condition notfull, notempty;    // 条件变量说明
    int count;
    public    // 能被进程调用的过程
    void put(item x) {    // 对数据结构操作的过程
        if (count≥N) then cwait(notfull);
        buffer[in] = x;
        in = (in+1) % N;
        count++;
        csignal(notempty);
    }
```

2.4.5 管程机制

◆ 管程的语法描述（生产者-消费者问题）

```
void get(item x) {    // 对数据结构操作的过程
    if (count ≤ 0) cwait(notempty);
     x = buffer[out];
     out = (out+1) % n;
     count--;
     csignal(notfull);
   }
   { in = 0; out  = 0; count = 0;}   // 初始化代码
}PC  // 管程的主体
```

2.4.5 管程机制

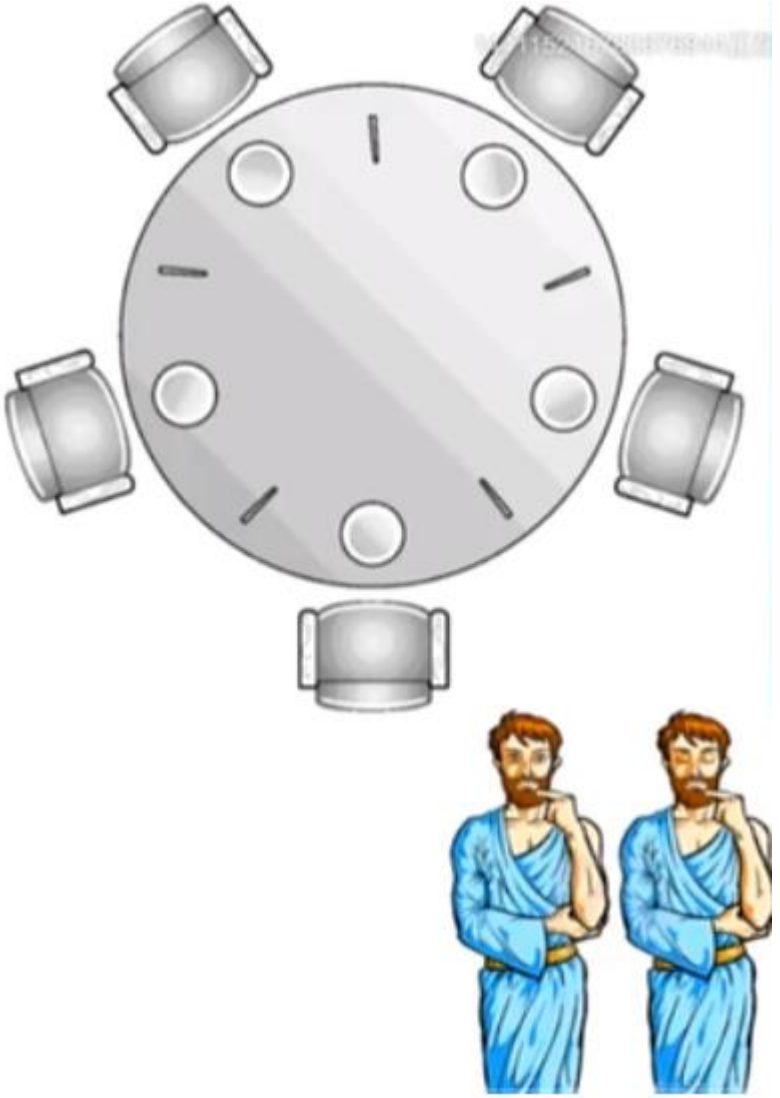
◆ 管程的语法描述（生产者-消费者问题）

```
void producer( ) {
        item x;
        while(TRUE) {
          ...
         produce an item in nextp;
        PC.put(x);
        }
    }
```

```
void consumer ( ) {
        item x;
        while(TRUE) {
         PC.get(x);
         consume the item in nextc;
         ...
        }
    }
void main( ) {
    cobegin
    producer(); consumer();
    coend
}
```

2.5.2 哲学家就餐问题

- **哲学家进餐问题由 Dijkstra提出并解决的哲学家进餐问题是典型的同步问题。**
- **问题描述:**
  - 有五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五**支**筷子，他们的生活方式是交替地进行思考和进餐。
  - 平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。
  - 进餐毕，放下筷子继续思考。



2.5.2 哲学家就餐问题

- 放在桌子上的筷子是临界资源,在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用可以用一个信号量表示一只筷子,由这五个信号量构成信号量数组semaphore chopstick[5}={1,1,1,1,1}; //各信号量初始化为1
- 互斥关系分析
  - 筷子：同一时刻只能有一个哲学家拿起筷子
- 同步关系分析
  - 就餐：只有获得两个筷子后才能进餐
- 特殊情况考虑
  - 死锁：如果每个哲学家都拿起一只筷子，都饿死
  - 并行程度：五只筷子允许两人同时进餐

2.5.2 哲学家就餐问题

- 为防止死锁发生可采取的措施：
  - 最多允许4个哲学家同时坐在桌子周围
  - 仅当一个哲学家左右两边的筷子都可用时才允许他拿筷子；
  - 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之；
  - 为了避免死锁，把哲学家分为三种状态，思考，饥饿，进食，并且一次拿到两只筷子，否则不拿。
  - 对所有筷子进行编码，规定先取编号晓得筷子
  - 如果哲学家拿起左边的筷子后,申请右边筷子得不到满足,则放下左边的筷子,隔一段时间（随机）后再申请左边的筷子。

2.5.2 哲学家就餐问题

- 方案1：最多允许4个哲学家同时坐在桌子周围

semaphore chopstick[5}={1,1,1,1,1};//各信号量初始化为1

semaphore count=4

哲学家活动描述为:

```
Do {
   wait(count);
   wait(chopstick[i]);
   wait(chopstick[(i+1)%5]);
   // 吃饭
  signal(chopstick(i);
  signal(chopstick[(i+1)%5]);
  signal(count);
  //思考
} while(TRUE);
```

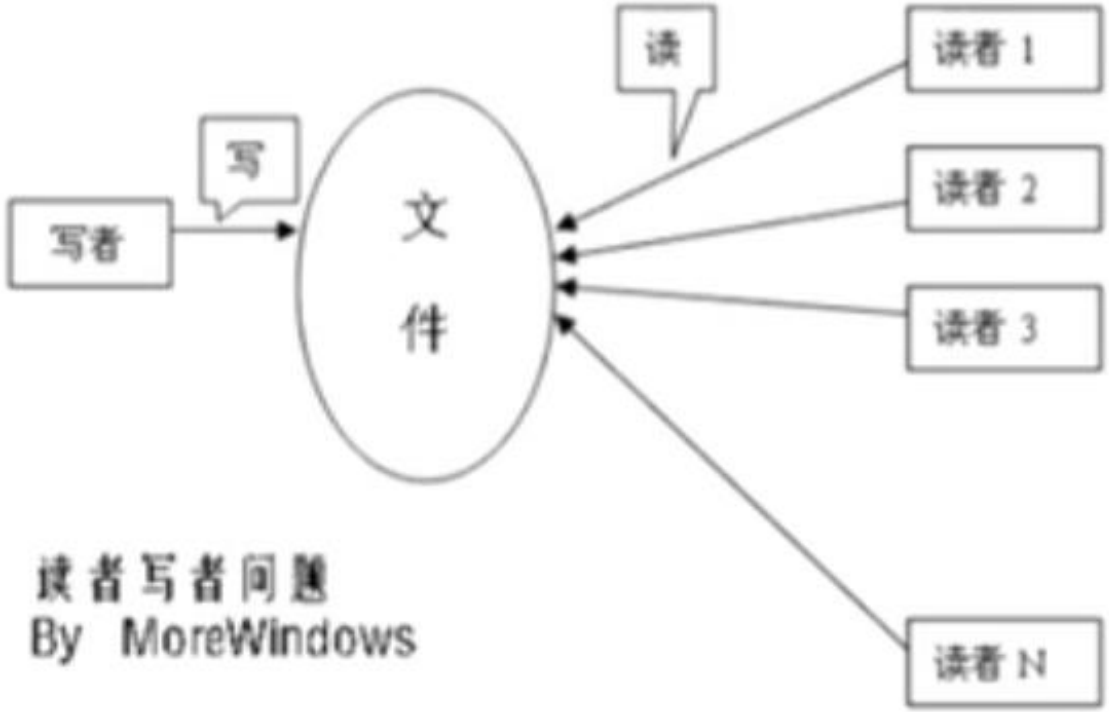
2.5.2 哲学家就餐问题

- 方案2：仅当一个哲学家左右两边的筷子都可用时才允许他拿筷子（AND信号量机制）

```
semaphore chopstick[5]={1,1,1,1,1};
do {
      ……
      //思考
      Swait(chopstick[(i+1)%5],chopstick[i]);
      ……
      //进餐
      Signal(chopstick[(i+1)%5],chopstick[i]);
}while[TURE];
```

2.5.3 读者和写者问题

- 有两组并发进程:
  - 读者（reader进程）和写者（writer进程），共享一组数据区
- 要求:
  - 允许多个读者同时执行读操作
  - 不允许读者、写者同时操作
  - 不允许多个写者同时操作



2.5.3 读者和写者问题

- 没有读者的时候，才可以写，读者比较优先

- 两个进程：
  - Reader、Writer
  - 读者与写者间的互斥信号量：Wmutex＝1
  - 多个读者间的互斥信号量：Rmutex＝1

- Readcount：正在读取的进程数目
  - Readcount＝0时允许写

2.5.3 读者和写者问题

读者Reader

```
While(True){
    ∘ ∘ ∘
    执行读操作
    ∘ ∘ ∘
}
```

写者Writers

```
While(True){
    ∘ ∘ ∘
    执行写操作
    ∘ ∘ ∘
}
```

2.5.3 读者和写者问题

读者Reader

```
While(True){

    ∘ ∘ ∘

   执行读操作

    ∘ ∘ ∘

}
```

写者Writers

```
While(True){



 P(wmutex);
  执行写操作
 V(wmutex);




}
```

2.5.3 读者和写者问题

读者Reader

```
While(True){
    P(rmutex)
    readcount++;
    if(readcount==1)
          P(wmutex);
    V(rmutex)
    执行读操作
    P(rmutex)
    readcount--;
    if(readcount==0)
        V(wmutex);
    V(rmutex)}
```

写者Writers

```
While(True){



 P(wmutex);
  执行写操作
 V(wmutex);




}
```

2.5.3 读者和写者问题

- 增加限制条件，即同时读取的读者数不能超过**RN**
- 信号量集:
  - Swait(S, t, d);　Ssignal(S,d)
  - S为信号量，t为下限值, d为需求量，
- **L, mx:=RN, 1**
  - 通过执行wait(L,1,1)操作来控制读者的数目,每当有一个读者进入时,就要先执行wait(L,1,1)操作,使L的值减1。
  - 当有RN个读者进入读后L便减为0，第RN+1个读者要进入读时，必然会因wait(L,1,1)操作失败而阻塞。
  - mx表示写资源。

2.5.3 读者和写者问题

```
int RN;
semaphore L=RN,mx=1;
void reader() {
   do {
      Swait(L,1,1;mx,1,0);
      …
      进行读操作;
      Ssignal(L,1);
   }while(TRUE);
}
```

```
void writer() {
   do {
      Swait(mx,1,1;L,RN,0);
      ……
      进行写操作;
      Ssignal(mx,1);
    }while(TRUE);
}

void main() {
   cobegin
      reader();
      writer();
    coend
```


THEORY
PRACTICAL