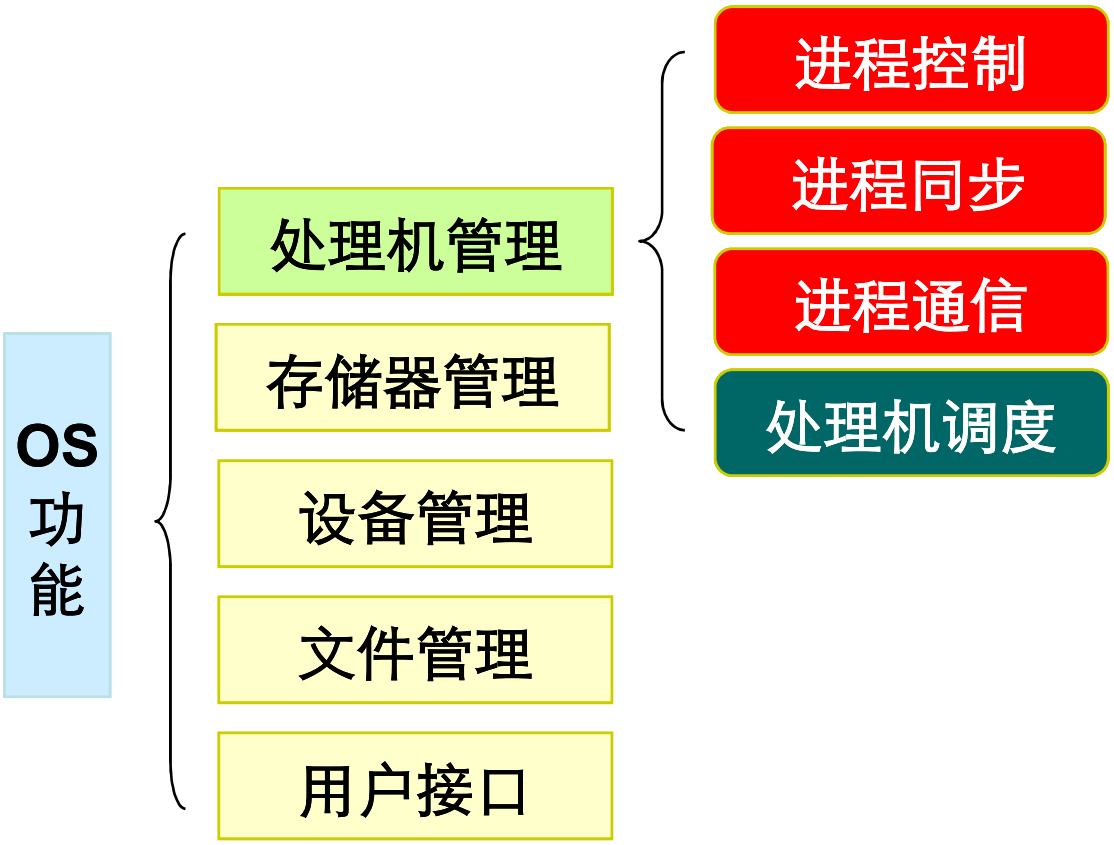
第2章 进程管理 (10学时)

主讲教师：张春元
联系电话：13876004640
课程邮箱：haidaos@126.com
邮箱密码：zhangchunyuan

本章内容所处位置

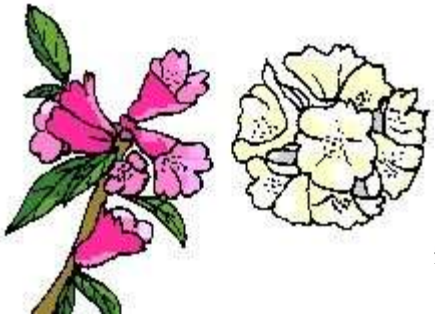


本章主要内容

- 2.1 前趋图和程序执行
- 2.2 进程的描述
- 2.3 进程控制
- 2.4 进程同步
- 2.5 进程同步的经典问题
- 2.6 进程通信
- 2.7 线程

2.1 前趋图和程序执行

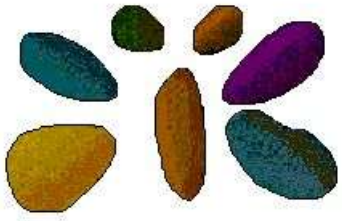
- 2.1.1 前趋图
- 2.1.2 程序的顺序执行及其特征
- 2.1.3 程序的并发执行及其特征



2.1.1 前趋图

❖ 1、概念

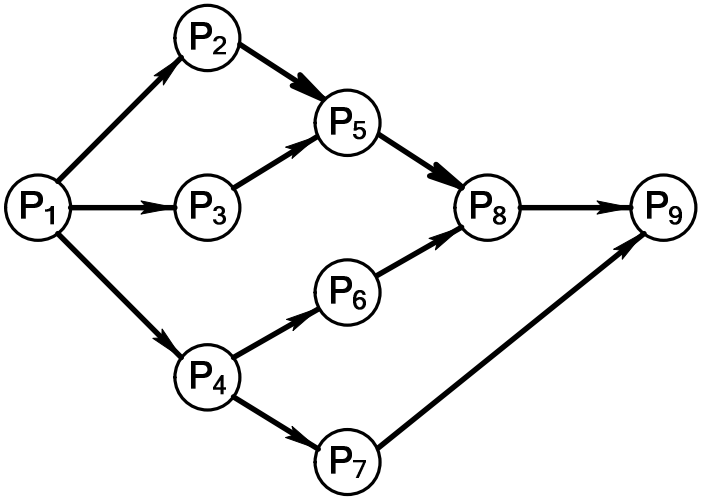
* 前趋图是一个有向无循环图，记为DAG(Directed Acyclic Graph)，其结点间的有向边则用于表示两个结点之间存在的偏序或前趋关系→={(Pi, Pj)|Pi必须在Pj开始前完成}。
* 如果(Pi, Pj)∈→，可写成Pi→Pj，称Pi是Pj的直接前趋，而称Pj是Pi的直接后继。
* 在前趋图中，把没有前趋的结点称为初始结点，把没有后继的结点称为终止结点。



2.1.1 前趋图

❖ 2、前趋图的应用

* 前趋图可用来描述进程之间执行的前后关系，图中的每个结点可用于描述一个程序段或进程，乃至一条语句；每个结点还具有一个重量(Weight，权值)，用于表示该结点所含有的程序量或结点的执行时间。

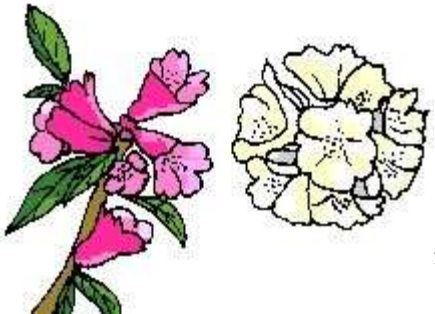


(a) 具有九个结点的前趋图

(b) 具有循环的图

2.1 前趋图和程序执行

- 2.1.1 前趋图
- 2.1.2 程序的顺序执行及其特征
- 2.1.3 程序的并发执行及其特征



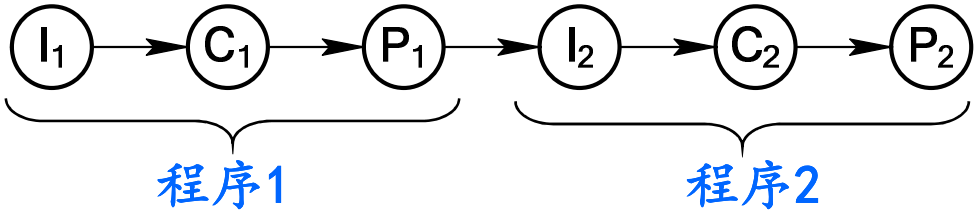
2.1.2 程序的顺序执行及其特征

❖ 1、概念

* 程序的顺序执行是指若干个程序或程序段之间必须严格按照某种先后次序来执行，仅当前一个程序或程序段执行完后，才能执行后面的程序或程序段。

❖ 2、程序顺序执行示例

* 1> 两个程序的顺序执行



* 2> 三条语句顺序执行

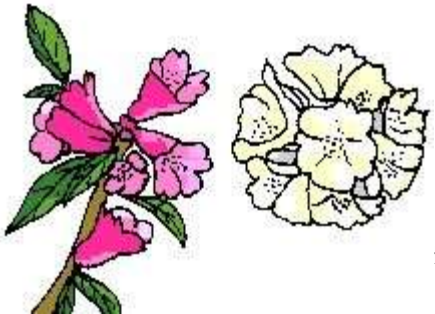
2.1.2 程序的顺序执行及其特征

❖ 3、程序顺序执行的特征

* 1> 顺序性：处理机的操作严格按照程序所规定的顺序执行，只有当上一操作完成后，下一操作才能执行。
* 2> 封闭性：程序运行在一个封闭的环境中，即程序运行时独占全机资源，资源的状态（初始状态除外）只有本程序才能改变，结果不受任何外界因素的影响。
* 3> 可再现性：由于程序顺序执行的封闭性，程序重复执行时只要程序顺序执行的初始条件和环境相同，则不论何时执行，也不论执行期间是否存在停顿，程序所得的结果相同。

2.1 前趋图和程序执行

- 2.1.1 前趋图
- 2.1.2 程序的顺序执行及其特征
- 2.1.3 程序的并发执行及其特征



2.1.3 程序的并发执行及其特征

❖ 1、概念

* 程序的并发执行是指两个或两个以上的程序或程序段可在同一时间间隔内同时执行。

❖ 2、程序并发执行示例

* 1> 四个程序并发执行

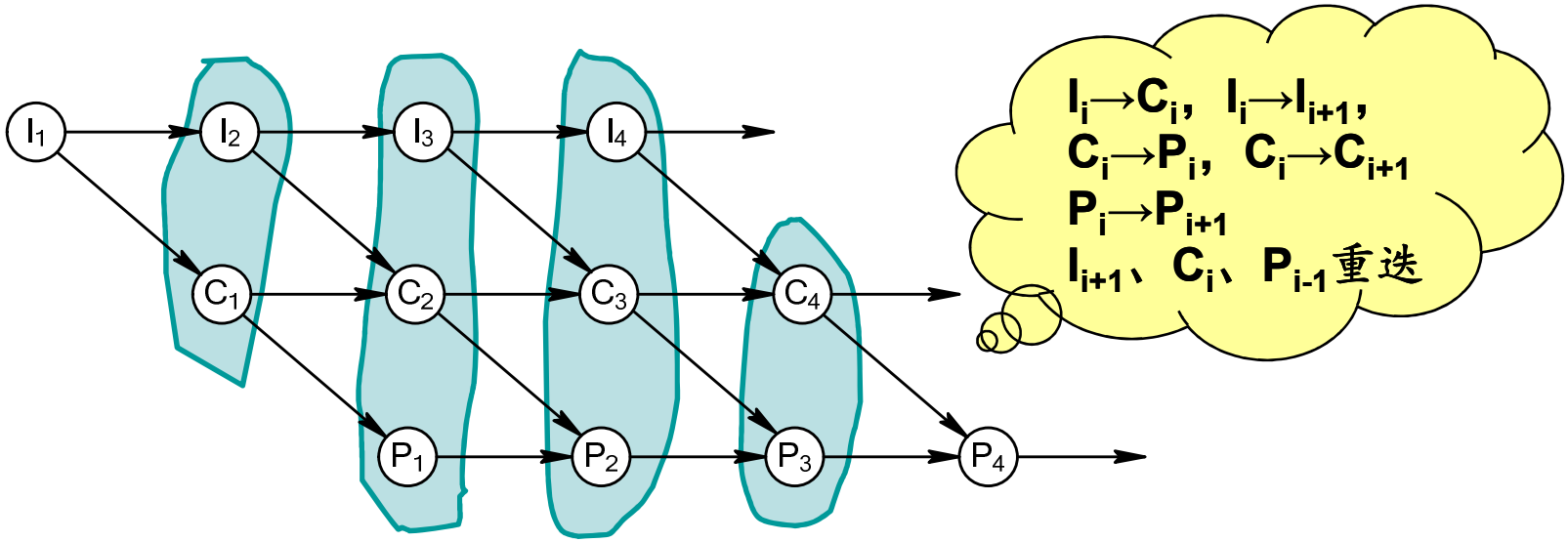


图2-3 四个程序并发执行前趋图

2.1.3 程序的并发执行及其特征

* 2> 一个程序段的两条语句并发执行

程序段：

S_1： a=x+2;

S_2： b=y+5;

S_3： c=a+b;

S_4： d=c+6;

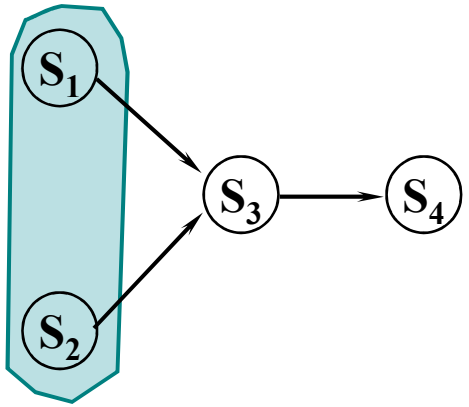
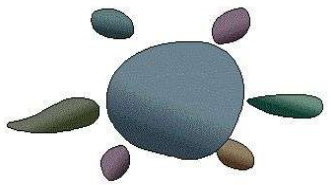


图2-4 四条语句的前趋图

2.1.3 程序的并发执行及其特征

❖ 3、程序并发执行的特征

* 1> 间断性：走走停停，一个程序可能走到中途停下来，失去原有的时序关系。
* 2> 失去封闭性：程序并发执行时，系统中多道程序共享资源，资源的状态不是唯一地取决于某一个程序，因此，必然失去了程序的封闭性，而程序的执行结果因依赖于外部环境也失去了可再现性。
* 3> 不可再现性： 失去封闭性导致失去可再现性；外界环境可能在程序的两次执行期间发生变化，失去原有的可重复特征。



2.1.3 程序的并发执行及其特征

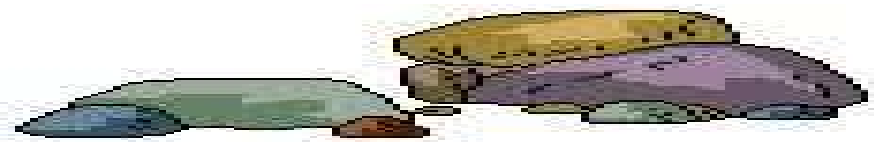
❖ 4、程序并发执行不可再现性示例

[例] 程序A： *N=N+1；* 程序B： *Print(N)；* *N=0；* 它们共享一个变量*N=n*，若程序A和B并发执行，以不可预知的速度运行，则可出现以下三种情况：

1> 程序A在程序B之前运行，得到的*N*值分别为*n+1* *n+1* *0*

2> 程序A在程序B之后运行，得到的*N*值分别为*n* *0* *1*

3> 程序A在程序B之间运行，得到的*N*值分别为*n* *n+1* *0*



2.1.3 程序的并发执行及其特征

❖ 附：程序并发执行的Bernstein条件

并发执行失去封闭性的原因是共享资源的影响，去掉这种影响就行了。1966年，Bernstein给出了并发执行的条件：若两个程序P_1和P_2满足下述条件，便能并发执行且有可再现性：

$$(R(P_1)\cap W(P_2))\cup(R(P_2)\cap W(P_1))\cup(W(P_1)\cap W(P_2))=\{\}$$

其中：运算的读集$R(P_i)$是指在运算执行期间需参考的所有变量的集合；运算的写集$W(P_i)$是指在运算执行期间要改变的所有变量的集合。

2.1.3 程序的并发执行及其特征

❖ 附：Bernstein条件应用示例

[例]已知四条语句分别如下：

S_1: a=x+y　　S_2:b=z+1　　S_3: c=a-b　　S_4:d=c+1

则相应的读集与写集分别为：

$R(S_1)=\{x, y\}$　　$R(S_2)=\{z\}$　　$R(S_3)=\{a, b\}$　　$R(S_4)=\{c\}$

$W(S_1)=\{a\}$　　$W(S_2)=\{b\}$　　$W(S_3)=\{c\}$　　$W(S_4)=\{d\}$

语句S_1和S_2可以并发执行吗？　可以

语句S_1和S_3可以并发执行吗？　不可以

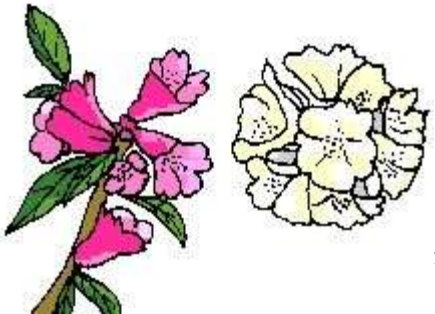
语句S_2和S_3可以并发执行吗？　不可以

语句S_3和S_4可以并发执行吗？　不可以

语句S_2和S_4可以并发执行吗？　可以

2.2 进程的描述

- 2.2.1 进程的定义和特征
- 2.2.2 进程的基本状态及转换
- 2.2.3 进程控制块



2.2.1 进程的定义和特征

❖ 1、进程的定义

进程的概念是20世纪60年代初首先由麻省理工学院的MULTICS系统和IBM公司的OS/360系统引入的。进程有很多各式各样的定义，较典型的定义有：

* 1> 进程是程序的一次执行。
* 2> 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。
* 3> 进程是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一个独立单位。
* 4> 在引入了进程实体的概念后，传统OS中的进程还可定义为：进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。

2.2.1 进程的定义和特征

注：进程实体（在早期的UNIX版本中，把这三部分总称为“进程映像”）=程序段+数据段+进程控制块(PCB，详参P41)。许多情况下所说的进程，实际上是指进程实体。进程的创建与撤消就是PCB的创建与撤消。

2.2.1 进程的定义和特征

❖ 2、进程的特征

* 1> 动态性：进程是一个动态的概念，实质上是程序的一次执行过程。进程具有生命期：它因创建而产生，因调度而执行，执行时还走走停停，因撤消而灭亡。动态性是进程最基本的特征。

* 2> 并发性：多个进程实体同存于内存中，且能在一段时间内同时运行，共享系统资源；引入进程实体的目的就是为了实现多道程序的并发执行。

* 3> 独立性：在传统OS中，进程是一个能独立运行、独立分配资源、独立接受调度的基本单位。

* 4> 异步性：各进程按各自独立的、不可预知的速度向前推进。

2.2.1 进程的定义和特征

重要！

❖ 3、进程与程序的区别

* 进程是动态的，程序是静态的：程序是有序代码的集合，它可以复制；进程是程序在数据集上的一次执行。

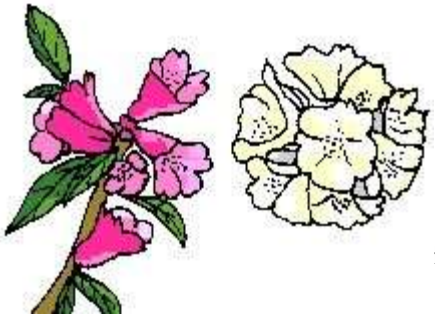
* 进程是暂时的，程序是永久的：进程是一个状态变化的过程，具有一定的生命期；程序可长久保存。

* 进程具有结构特征，由程序段、数据段和进程控制块（PCB）三者组成，而程序仅是指令的有序集合，是进程的组成部分之一。

* 进程与程序的对应关系：并非一一对应关系；通过多次执行，一个程序可对应多个进程。有时一个程序运行时，也可能创建多个进程。

2.2 进程的描述

- 2.2.1 进程的定义和特征
- 2.2.2 进程的基本状态及转换
- 2.2.3 进程控制块



2.2.2 进程的基本状态及转换

❖ 1、进程的状态

* 1.1 三种基本状态
* 1.2 创建状态和终止状态
* 1.3 挂起状态
* 注：OS类型不同，进程的状态类型有所不同，有的OS中是三状态，有的是五状态，有的是七状态， UNIX有九种状态。

2.2.2 进程的基本状态及转换

❖ 1.1 三种基本状态

* 1> 就绪(Ready)状态

- 进程已获得除处理机外的所需资源，等待分配处理机资源；此时只要分配到CPU就可执行。
- 一个系统中多个处于就绪状态的进程排成就绪队列。

* 2> 执行(Running)状态

- 处于就绪状态的进程一旦获得了处理机，就可以运行，此时进程状态也就处于执行状态。
- 处于此状态的进程的数目小于等于CPU的数目。

2.2.2 进程的基本状态及转换

* 3> 阻塞(Blocked)状态(等待状态、睡眠状态、封锁状态)
  - 进程处于等待某种事件(如I/O操作或进程同步)发生的状态，在等待的事件发生之前无法继续执行。该事件发生前即使把处理机分配给该进程，也无法运行。如：请求I/O操作、申请缓冲空间等。
  - 通常阻塞进程也排成一个或若干个队列（如可按阻塞原因类别排列）。

2.2.2 进程的基本状态及转换

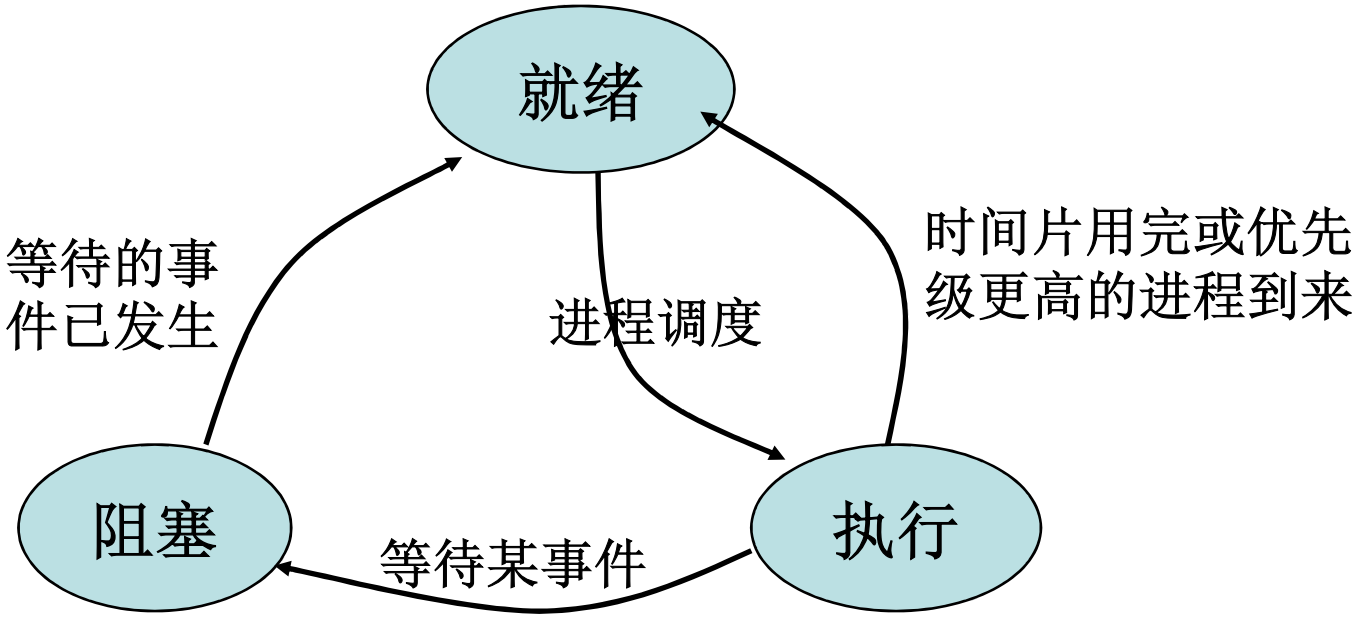


图2－5 进程的三种基本状态及其转换

2.2.2 进程的基本状态及转换

❖ 1.2 创建状态和终止状态

* 创建（New）状态

- 当一个新进程刚刚建立，还未将其放入就绪队列时的状态，称为创建状态（新状态）。
- 一般而言，此时进程已经拥有了自己的PCB，但尚未分配内存资源，进程还不能被调度运行。

* 终止（Terminated）状态

- 当一个进程已经正常结束或异常结束或受外界干预结束，即进入终止状态。
- 此时进程不再具有执行资格，OS将释放终止进程所拥有的全部资源，将其从系统队列中移出并收回其PCB。

2.2.2 进程的基本状态及转换

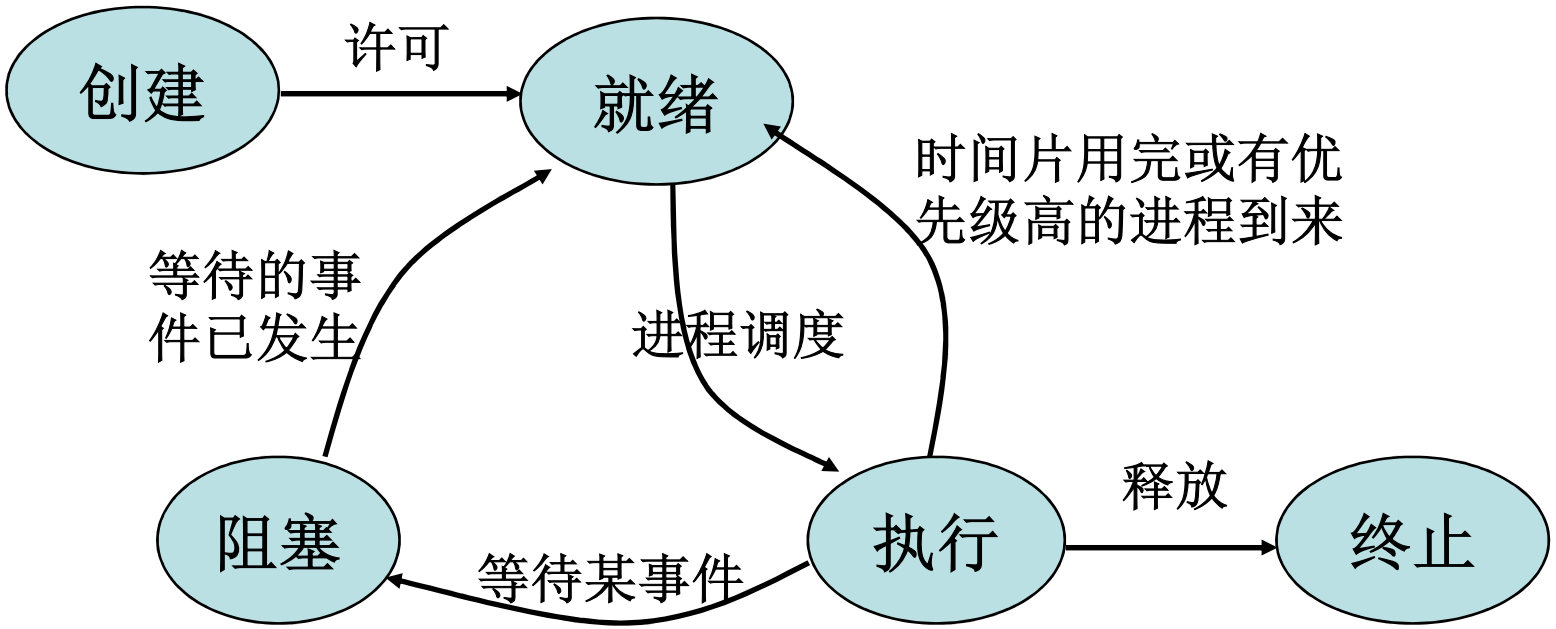


图2－6 进程的五种状态及其转换

2.2.2 进程的基本状态及转换

❖ 1.3 挂起（Suspend）状态（被换出内存的状态）

* 1> 引入挂起状态的原因

- (1) 终端用户的请求
  - 当终端用户在自己的程序运行期间发现有可疑问题时，希望暂时将自己的程序静止下来。
- (2) 父进程请求
  - 父进程需要考查和修改子进程，或者协调各子进程。
- (3) 负荷调节的需要
  - 在实时系统中为了调整工作负荷将不重要的进程挂起。
- (4) 操作系统的需要
  - OS检查运行中的资源使用情况或进行记帐。

2.2.2 进程的基本状态及转换

* 2> 具有挂起状态的进程状态转换（中程调度）

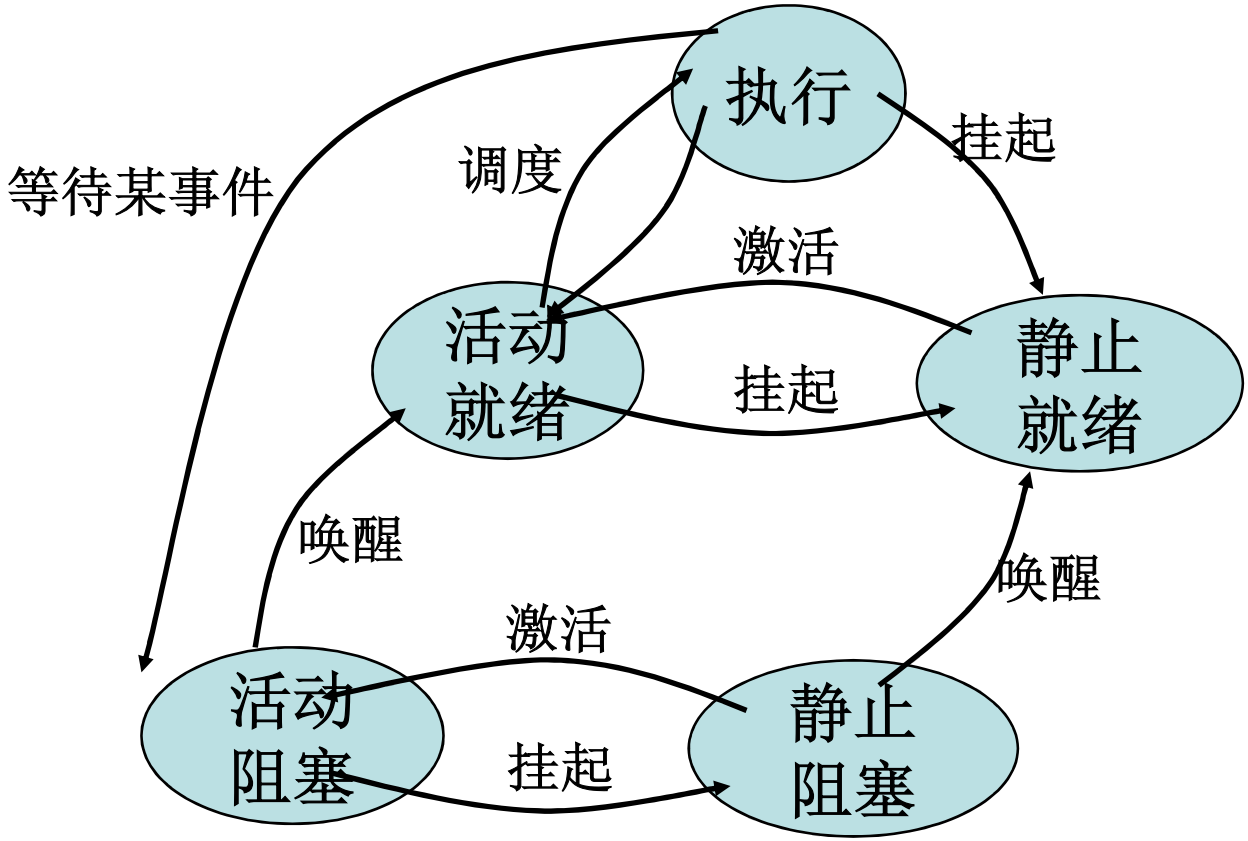


图2-7 具有挂起状态的基本进程状态转换

2.2.2 进程的基本状态及转换

* 3> 七种进程状态转换

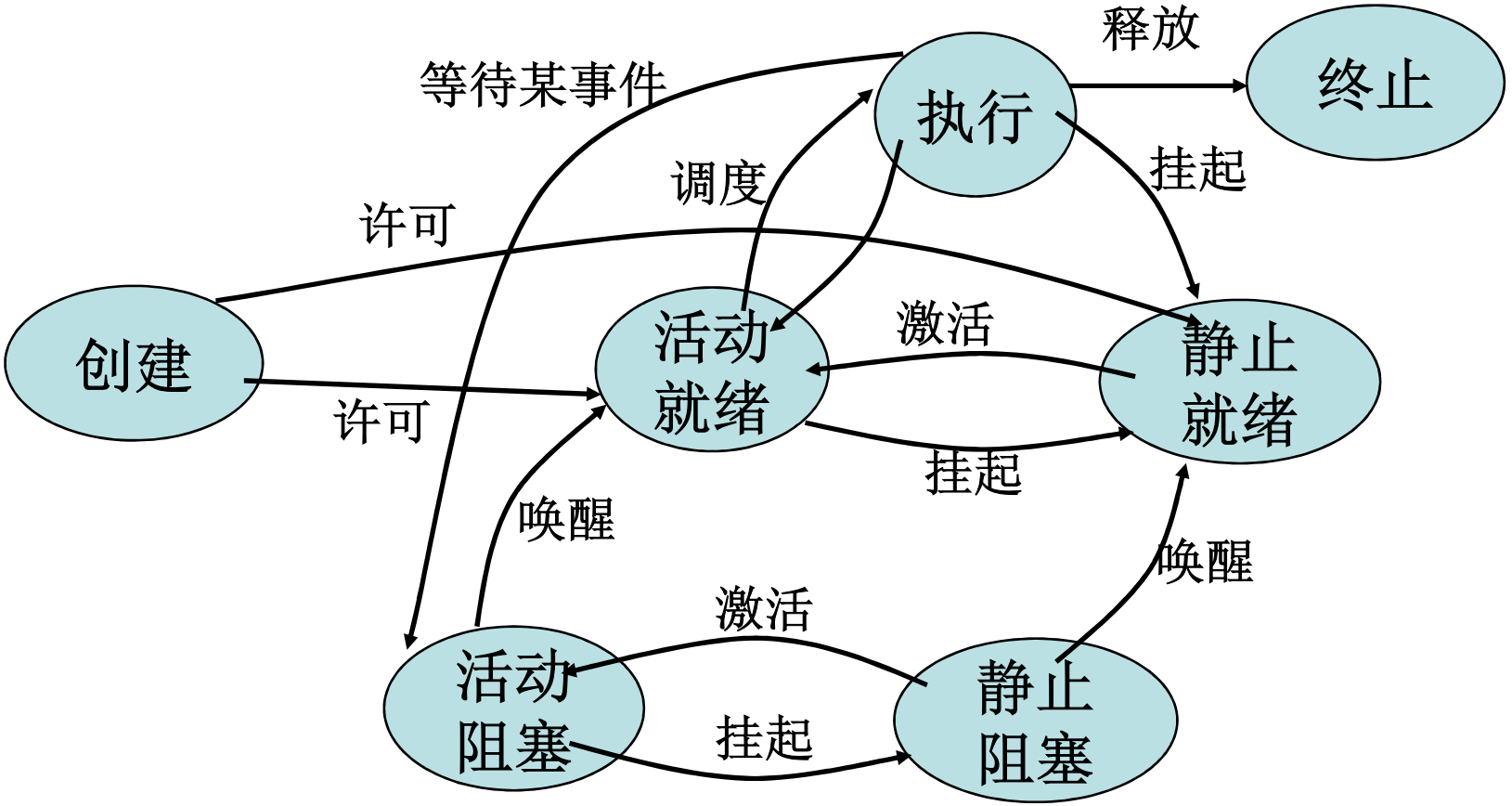


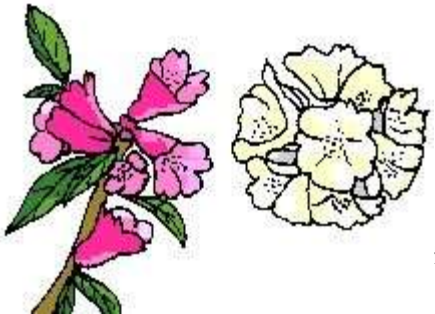
图2-8　七种进程状态转换

2.2.2 进程的基本状态及转换

* 2> 具有挂起状态的进程状态转换
  - 活动就绪→静止就绪（挂起）
    - 这种情况比较少见，一般是出于性能上的考虑，必须释放一些内存时
  - 活动阻塞→静止阻塞（挂起）
    - 等待某事件较长，为了节省内存
  - 静止就绪→活动就绪（激活）
    - 当内存中没有活动就绪进程时，OS会安排空间让一个静止就绪进程进入内存
  - 活动阻塞→活动就绪　静止阻塞→静止就绪
    - 当等待的事件发生时（唤醒）

2.2 进程的描述

- 2.2.1 进程的定义和特征
- 2.2.2 进程的基本状态及转换
- 2.2.3 进程控制块



2.2.3 进程控制块(PCB)

❖ 1、进程控制块中的信息

PCB是一种数据结构，记录了操作系统所需的、用于描述进程情况及控制进程运行所需的全部信息。

* 1> 进程标识符
* 2> 处理机状态
* 3> 进程调度信息
* 4> 进程控制信息

- PCB是OS感知进程存在的唯一标志。
- 进程与PCB是一一对应的。
- PCB应常驻内存。

2.2.3 进程控制块(PCB)

❖ 2、进程控制块的作用

PCB的作用是使一个在多道程序环境下不能独立运行的程序(含数据) 成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程。或者说，OS是根据PCB来对并发执行的进程进行控制和管理的。具体作用：

* 1> 作为程序独立运行的标志
* 2> 能实现间断性运行方式
* 3> 提供进程管理所需要的信息
* 4> 提供进程调度所需要的信息
* 5> 实现与其它进程的同步与通信

2.2.3 进程控制块(PCB)

❖ 3、进程控制块的组织方式

* 1> 线性方式 将系统中所有PCB 组织存放在一张线性表中。

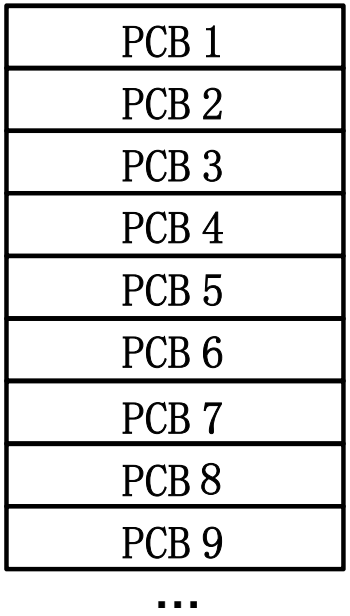


图2-9 PCB线性表示意图

2.2.3 进程控制块(PCB)

* 2> 链接方式 把具有同一状态的PCB用其中的链接字链接成一个队列，可以形成就绪队列、若干个阻塞队列和空白队列(下图中0表示该队列结束)。

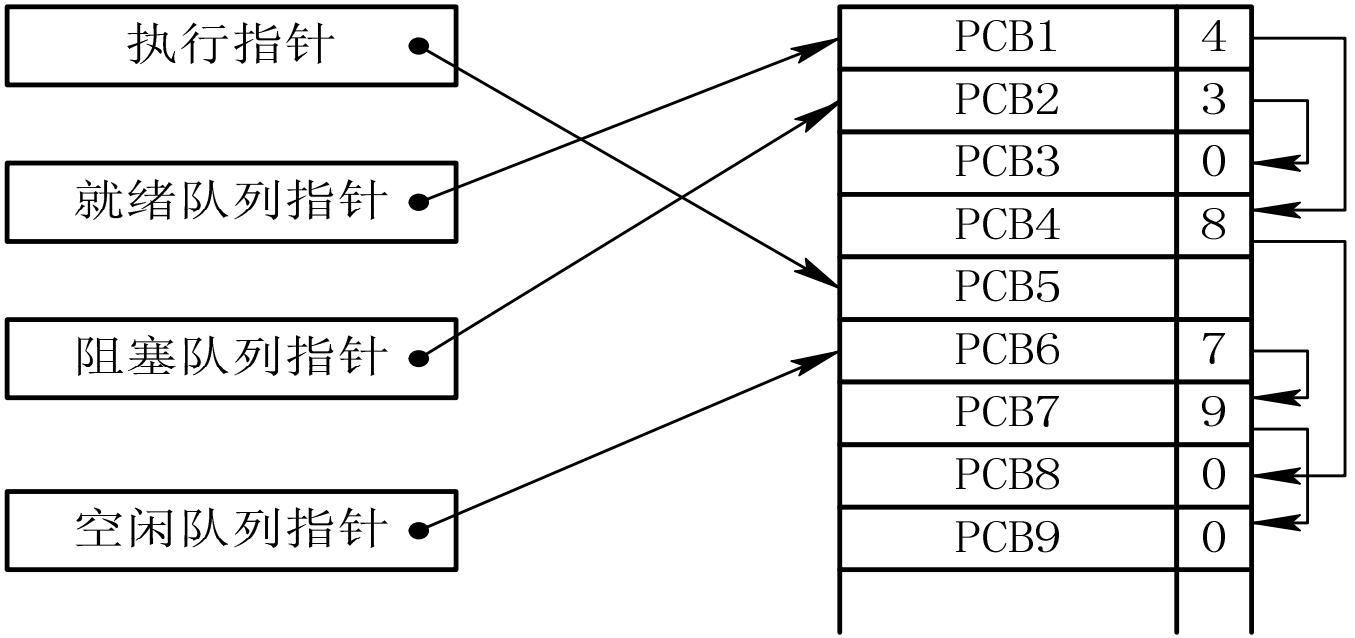


图2-10 PCB链接队列示意图

2.2.3 进程控制块(PCB)

* 3> 索引方式 系统根据所有进程的状态建立多张索引表，如就绪索引表、阻塞索引表等，并把各索引表在内存的首地址记录在专用单元中，索引表中记录的是PCB在PCB表中的地址。

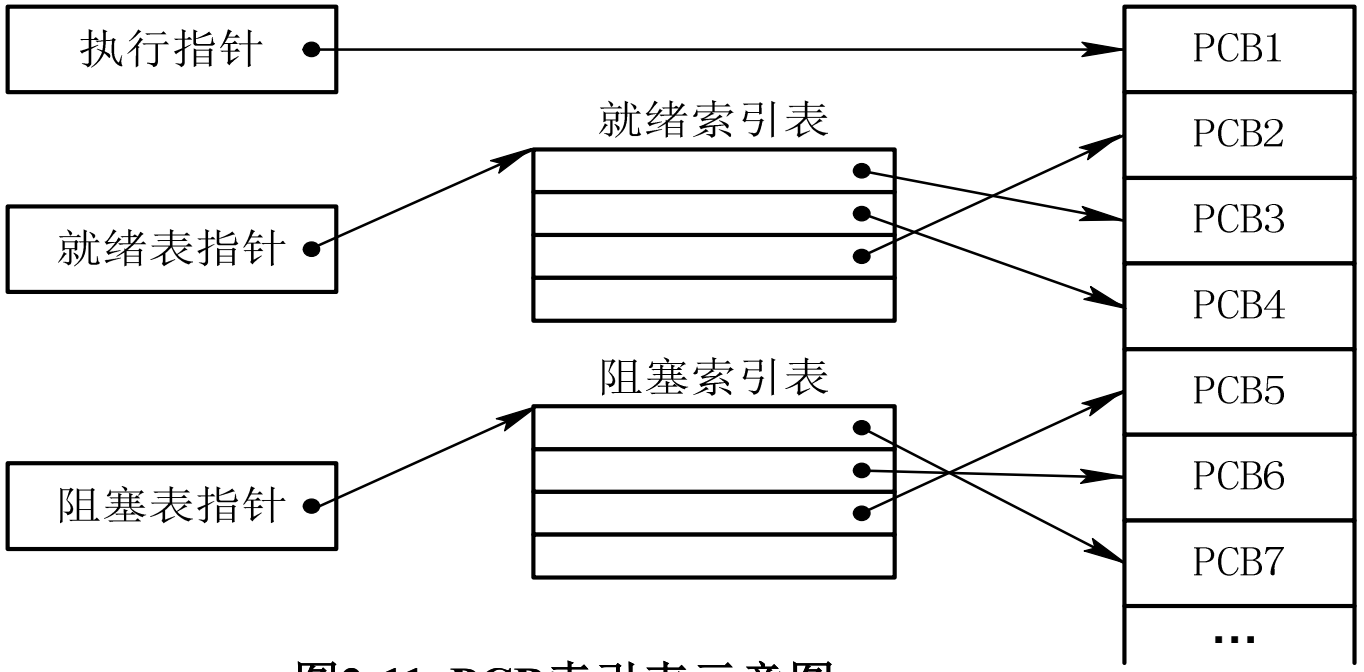
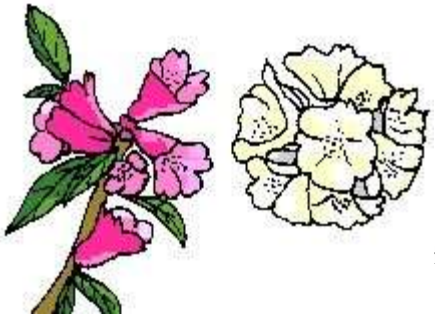


图2-11 PCB索引表示意图

2.3 进程控制

由OS内核中
的原语来实现

- 2.3.1 相关概念
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活



2.3.1 相关概念

❖1、系统态和用户态

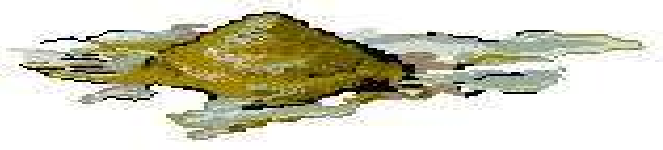
在计算机系统中，为了保证OS程序不被应用程序（用户程序）有意或无意破坏，将处理机的指令集的运行状态分为系统态和用户态两种：

* 系统态(管态、核心态)：有较高特权，能执行一切指令，访问所有寄存器和存储区。
  - OS微内核运行在系统态。
* 用户态(目态)：有较低特权，只能执行规定指令，访问指定寄存器和存储区。
  - 用户程序只能运行在用户态。
  - 系统调用会引起CPU从用户态转入核心态。

2.3.1 相关概念

❖2、原语与原子操作

* 原语（Primitive）：由若干条指令构成的用于完成一定功能的一个原子操作(atomic operation)过程。
* 原子操作：是指一个操作中的所有动作要么全做，要么全不做。换言之，它是一个不可分割的基本单位，在执行过程中不可中断，以保证其正确性。
* 许多系统调用（但并非所有）就是原语。
* 在OS中，原子操作在管态下执行，常驻内存。



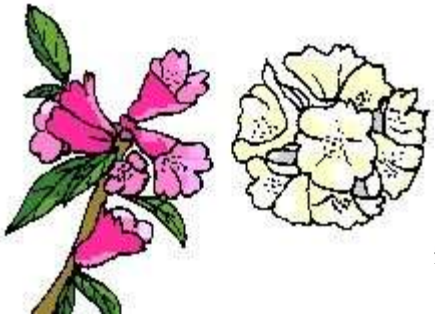
2.3.1 相关概念

❖ 3、进程控制的功能

进程控制是进程管理中最基本的功能，进程控制一般由OS内核中的原语（why？）来实现，其主要用于进程的创建、终止以及进程运行过程中的状态转换。

2.3 进程控制

- 2.3.1 相关概念
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活



2.3.2 进程的创建

❖ 1、进程图(Process Graph)

* 进程图是用于描述一个进程的家族关系的有向树，树中的结点表示进程。
* 在进程A创建了进程B之后称A是B的父进程，B是A的子进程。创建父进程的进程称为祖先进程，把树的根结点称为进程家族的祖先进程。
* 子进程可以继承父进程的资源。
* 撤消父进程时必须同时撤消子进程。

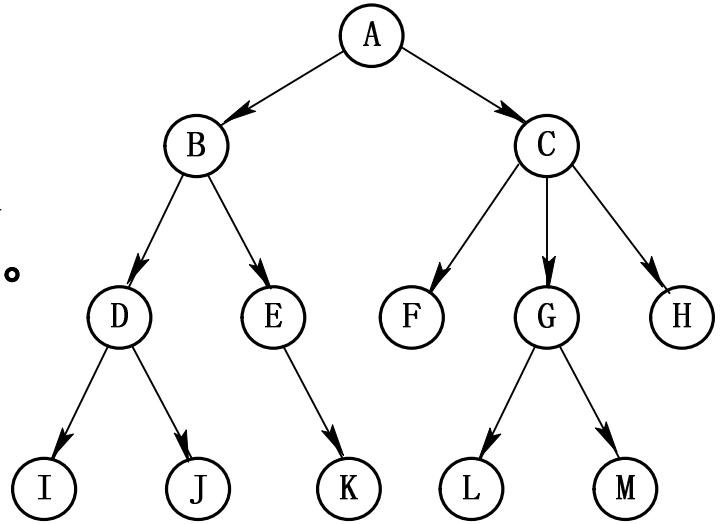


图2-11 进程树

2.3.2 进程的创建

❖ 2、引起创建进程的事件

* 1> 用户登录
  - 为终端用户建立一进程
* 2> 作业调度（不是进程调度）
  - 为被调度的作业建立进程、分配资源
* 3> 提供服务
  - 例如要打印时建立打印进程
* 4> 应用请求
  - 由应用进程（用户进程）为自己创建进程，以便能并发执行，如输入、计算、输出程序。

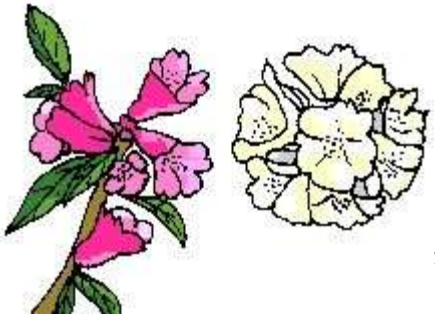
2.3.2 进程的创建

❖ 3、进程的创建过程（creat原语）

* 1> 申请空白PCB
  - 入口信息：进程标识符、优先级、进程开始地址、初始CPU状态、资源清单等。
* 2> 为新进程分配资源(?)
* 3> 初始化PCB
* 4> 将新进程插入就绪队列
  - 设置相应的链接（把新进程加到就绪队列的链表中），进程由创建状态转入就绪状态。

2.3 进程控制

- 2.3.1 相关概念
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活



2.3.3 进程的终止

❖ 1、引起进程终止的事件

* 1> 正常结束

- 在任何计算机系统中，都应有一个用于表示进程已经运行完成的指示。例如批处理系统的Holt指令、分时系统的Logs off指令。

* 2> 异常结束

- 在进程运行期间，由于出现某些错误和故障而迫使进程终止。这类异常事件很多，常见的有：①越界错误、②保护错误、③非法指令、④特权指令错误、⑤运行超时、⑥等待超时、⑦算术运算错误、⑧I/O故障。

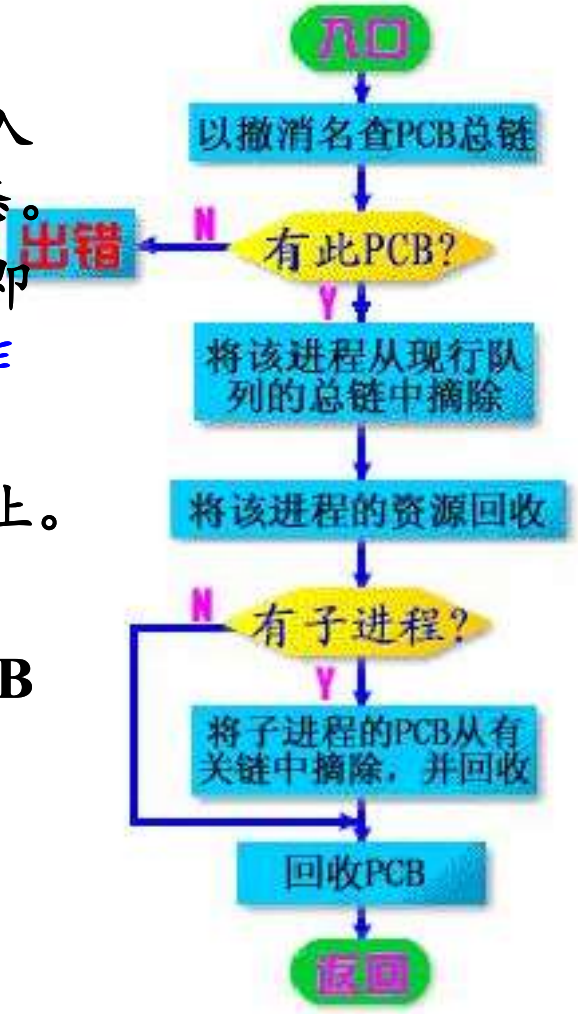
2.3.3 进程的终止

* 3> 外界干预
  - 外界干预并非指在本进程运行中出现了异常事件，而是指进程应外界的请求而终止运行。这些干预有：
    - ①操作员或操作系统干预。由于某种原因，例如，发生了死锁，由操作员或操作系统终止该进程。
    - ②父进程请求。由于父进程具有终止自己的任何子孙进程的权利，因而当父进程提出请求时，系统将终止该进程。
    - ③父进程终止。当父进程终止时，OS也将他的所有子孙进程终止。

2.3.3 进程的终止

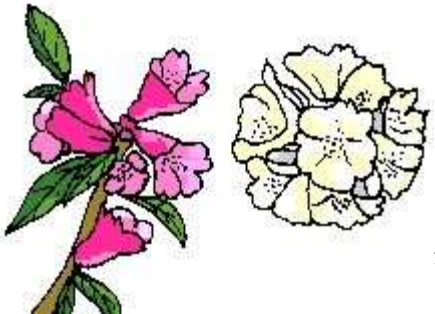
❖ 2、进程的终止过程（destroy原语）

* 1> 以被终止进程的PCB标识符为入口信息，检索、读出该进程的状态。
* 2> 若该进程处于执行状态，应立即终止其执行，且置重新调度（指作业调度）标志为真。
* 3> 查看该进程有无子孙进程需终止。
* 4> 归还资源给其父进程或系统。
* 5> 将被终止进程的PCB从所在PCB队列（链表）中移出。



2.3 进程控制

- 2.3.1 相关概念
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活



2.3.4 进程的阻塞与唤醒

❖ 1、引起进程阻塞与唤醒的事件

* 1> 请求系统服务失败

- 当正在执行的进程请求OS提供服务时，当OS不能立即满足该进程的要求时，该进程只能转为阻塞状态。
  - 如请求的打印机被其他进程正在占用。

* 2> 等待某种操作完成

- 当进程启动某种操作后，如果该进程必须在该操作完成后才能继续执行，则必须先使进程阻塞，以等待该操作完成。
  - 如启动I/O设备以完成I/O操作。

2.3.4 进程的阻塞与唤醒

* 3> 新数据尚未到达
  - 对于相互合作的进程，如果一个进程需要另一合作进程提供的数据，则在数据到达之前只能阻塞。
* 4> 无新工作可做
  - OS中的一些特殊功能的进程，在完成了任务之后将自己阻塞起来等待新任务的到来。
    - 如系统中的发送进程应要求给某进程发完数据。

2.3.4 进程的阻塞与唤醒

❖ 2、进程的阻塞过程（block原语）

进程的阻塞是进程自身的一种主动行为，具体过程如下：

* 1> 收集CPU现场信息存至PCB中。
* 2> 停止执行该进程，修改PCB中进程的状态信息。
* 3> 将该进程插入相应的阻塞队列。
* 4> 转调度程序重新调度别的就绪进程进入执行状态。

2.3.4 进程的阻塞与唤醒

❖ 3、进程的唤醒过程（wakeup原语）

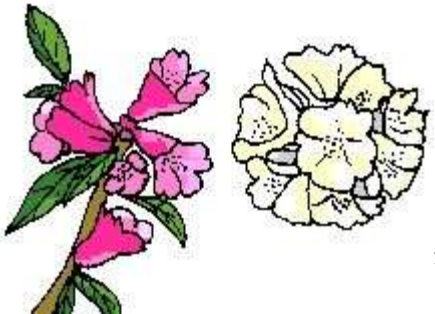
当被阻塞进程所期待的事件出现时，则调用相应wakeup原语，将等待该事件的进程唤醒，具体过程如下：

* 1> 把被阻塞的进程从等待该事件的阻塞队列中移出。
* 2> 将其PCB中的进程状态由阻塞改为就绪。
* 3> 将该PCB插入到就绪队列中。

注：有block原语，就应有相应的wakeup原语。

2.3 进程控制

- 2.3.1 相关概念
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活



2.3.5 进程的挂起与激活

❖ 1、引起进程挂起的事件（PPT P28，前面已讲过了）

* 1> 终端用户的请求

- 当终端用户在自己的程序运行期间发现有可疑问题时，希望暂时将自己的程序静止下来。

* 2> 父进程请求

- 父进程需要考查和修改子进程，或者协调各子进程。

* 3> 负荷调节的需要

- 在实时系统中为了调整工作负荷将不重要的进程挂起。

* 4> 操作系统的需要

- OS检查运行中的资源使用情况或进行记帐。

2.3.5 进程的挂起与激活

❖ 2、进程的挂起过程（suspend原语）

由进程自己或其父进程调suspend原语完成挂起：

* 1>首先检查、修改被挂起进程的状态：活动就绪→静止就绪、活动阻塞→静止阻塞、执行→静止就绪。
* 2> 为了方便用户或父进程考查该进程的运行情况，把该进程的PCB复制到某指定的内存区域。
* 3> 若被挂起的进程为执行状态，则转向调度程序重新调度，将CPU分配给另一活动就绪进程。

2.3.5 进程的挂起与激活

❖ 3、引起进程激活的事件

* 例如，父进程或用户进程请求激活指定进程，若该进程驻留在外存而内存中已有足够的空间时，则可将在外存上处于静止就绪状态的进程换入内存。这时，系统将利用激活原语将指定进程激活。

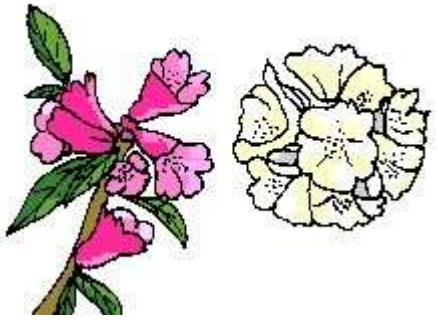
2.3.5 进程的挂起与激活

❖ 4、进程的激活过程（active原语）

* 激活原语先将进程从外存调入内存，检查、修改该进程的状态：静止就绪→活动就绪；静止阻塞→活动阻塞。
* 如采用的是抢占调度策略，则每当有被激活进程进入活动就绪队列时，应由调度程序检查被激活进程与当前进程的优先级，确定是否将被激活进程进一步调度为执行状态。
* 区别：阻塞、唤醒一般由OS实现，而挂起与激活可由用户干预实现。

2.4 进程同步

- 2.4.1 进程同步的基本概念
- 2.4.2 硬件同步机制
- 2.4.3 信号量机制
- 2.4.4 信号量的应用
- 2.4.5 管程机制



2.4.1 进程同步的基本概念

❖ 1、定义

* 进程同步：对多个相关的进程在执行次序上进行协调，以使并发执行的诸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。

❖ 2、进程间两种形式的制约关系

* 间接制约关系（互斥关系）
  - 需互斥地访问临界资源
* 直接制约关系（同步关系、合作关系）
  - 例：进程A通过缓冲区向进程B提供数据，相互唤醒。

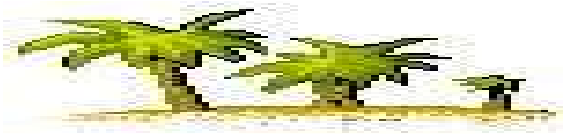
2.4.1 进程同步的基本概念

❖ 3、临界资源

* 临界资源：一段时间内仅允许一个进程访问的资源，也称独占资源、互斥资源、互斥变量。
  - 硬件临界资源：如打印机、磁带机
  - 软件临界资源：如不可重入码
* 临界资源共享冲突：程序并发执行的不可再现性通常是因为对临界资源没有互斥访问造成的。

2.4.1 进程同步的基本概念

* [例] 设一民航航班售票系统有n个售票处。每个售票处通过终端访问民航系统中的公用数据区（由一系列数据单元组成，每个单元Xk (k=1,2, …)保存着×月×日×次航班现存机票数量），设Pi (i=1,2,…,n)表示各售票处的处理进程， Ri (i=1,2,…,n)表示各进程Pi执行时所用的工作数据单元。售票终端程序实现如下，请问存在什么问题？？



2.4.1 进程同步的基本概念

```
// 售票终端程序
Process  Pi(d)(i=1,2,…,n)  //d为旅客×日×次航班机票订购数量
begin
    按旅客订票要求查询现存×日×次航班机票数量Xk;
    Ri=Xk;
    if  Ri≥d
          begin
                Ri=Ri-d;
                Xk=Ri;
                输出d张机票;
          end
    else
         begin
                输出“票已售完或没有d张票”;
          end
end
```

多个进程必须互斥地访问临界资源

2.4.1 进程同步的基本概念

❖ 4、临界区(Critical section)

* 定义：进程中访问临界资源的那段代码称为临界区。
* 临界资源互斥访问的实现

```
// 设置进入区（检查是否正在访问、设置访问标志）
entry section
```

critical section （临界区）

```
// 设置退出区（访问标志复位）
exit section
```

remainder section （剩余区、非临界区）

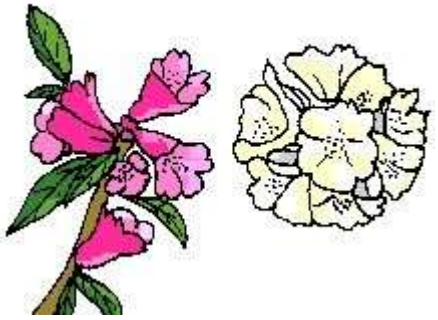
2.4.1 进程同步的基本概念

❖ 5、同步机制应遵循的规则

* 1> 空闲让进
  - 当无进程处于临界区时，应允许一个进程进入临界区，以有效利用临界资源。
* 2> 忙则等待
  - 当有进程进入临界区时，其他进程必须等待。
* 3> 有限等待
  - 对要求访问临界资源的进程，应保证在有限时间内进入自己的临界区，防止“死等”。
* 4> 让权等待
  - 当进程不能进入其临界区时，应立即让出处理机，防止“忙等”，不能一直用语句判断能不能进、占用处理机。

2.4 进程同步

- 2.4.1 进程同步的基本概念
- 2.4.2 硬件同步机制
- 2.4.3 信号量机制
- 2.4.4 信号量的应用
- 2.4.5 管程机制



2.4.2 硬件同步机制

进程的同步可采用软件方法实现，也可采用硬件方法实现。后者在对临界区管理时，设置一个标志（锁）来表示临界区有无进程进入。一个进程如要访问某临界区，首先测试锁是否打开，如开则进入并立即将锁锁上。为防止多个进程在试锁期间同时进入，测试锁与关锁必须是连续的，不允许分开进行。

* 1、关中断
* 2、利用 Test-and-Set指令实现互斥
* 3、利用Swap指令实现进程互斥

2.4.2 硬件同步机制

❖ 1、关中断

* 原理：使计算机不响应中断，从而不进行调度。
* 关开中断的时机：试锁前关中断，上锁后再开中断
* 缺点：
  - 1> 滥用关中断权力可能导致严重后果。
  - 2> 关中断时间过长，会影响系统效率。
  - 3> 不适合多CPU系统。

2.4.2 硬件同步机制

临界资源忙碌时，访问进程将处于忙等状态

❖ 2、利用 Test-and-Set指令实现互斥

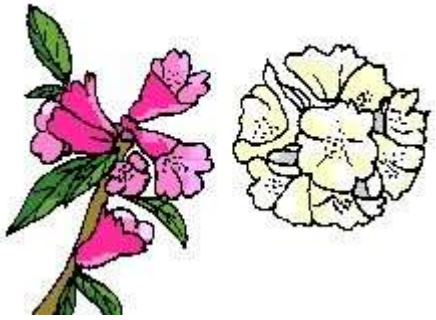
* 原理：为每个临界资源设置一个布尔变量lock表示该资源的状态（闲为false,忙为true），访问时利用TS原语指令检测lock。如果为false，则进入临界区；否则跳出当前检测。

❖ 3、利用Swap指令实现进程互斥

* 原理：为每个临界资源设置一个全局布尔变量lock，再在每个进程中设置一个局部变量key, 通过swap指令实现key与lock变量的内容互换，当key为false时进入临界区，否则继续运用swap指令互换

2.4 进程同步

- 2.4.1 进程同步的基本概念
- 2.4.2 硬件同步机制
- 2.4.3 信号量机制
- 2.4.4 信号量的应用
- 2.4.5 管程机制



2.4.3 信号量机制

1965年荷兰学者，Dijkstra提出的信号量机制是一种有效的进程同步工具，一个信号量(Semaphores) 代表着一类共享资源的可用数量。信号量机制共有四种类型：

* 1、整型信号量机制
* 2、记录型信号量机制
* 3、AND型信号量机制
* 4、信号量集机制

2.4.3 信号量机制

❖ 1、整型信号量机制

* 特点：信号量S是一个整型量，用于表示某类共享资源的数量，除初始化外，仅能通过两个原子操作wait(S)和signal(S)来访问：

```
wait(S):    while S<= 0;
            S=S-1;
```

忙等

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

* 缺点：当S <= 0 时，一直做空操作，并不交出CPU，该机制并未遵循“让权等待”的准则，而是使进程处于“忙等”的状态。

2.4.3 信号量机制

* wait(S)和signal(S)通常分别称为P操作、V操作(P、V分别来自荷兰语的proberen、verboten)，简记为P(S)、V(S)，其中P操作用来请求资源、V操作用来释放资源（回收资源）。

2.4.3 信号量机制

[例] 设S的初值为1，信号量用于售票终端进程互斥

```
Process  Pi(d)(i=1,2,…,n)   //d为旅客×日×次航班机票订购数量
begin
    P(S);  //执行P原子操作
    按旅客订票要求查询现存×日×次航班机票数量Xk;
    Ri=Xk;
    if  Ri≥d
        begin
            Ri=Ri-d;
            Xk=Ri;
            V(S);  //执行V原子操作
            输出d张机票;
        end
    else
        begin
            V(S);
            输出“票已售完或没有d张票”;
        end
end
```

2.4.3 信号量机制

❖ 2、记录型信号量机制

* 特点：记录型信号量机制采取了“让权等待”的策略，消除了整型信号量机制中存在的“忙等”现象，其信号量S是记录型数据结构：

```
type semaphore=record
        // value为某类资源的可用数量
        value: integer;
        // L为等待访问该类资源的进程链表
        L: list of process;
end
```

2.4.3 信号量机制

* 原子操作wait(S)和signal(S)

```
procedure wait(S)
    begin
        S.value=S.value-1;
        if S.value＜0 then block(S.L);
    end
```

当s.value<0时，其绝对值表示被阻塞的进程数

```
procedure signal(S)
    begin
        S.value=S.value+1;
        if S.value≤0 then wakeup(S.L);
    end
```

2.4.3 信号量机制

* 缺点：记录型信号量实现多类资源共享易造成死锁。

process A	process B
P(Dmutex);	P(Emutex);
P(Emutex);	P(Dmutex);

设信号量Dmutex、Emutex初始值均为1，若进程A和B按下述次序交替执行P操作：

process A: P(Dmutex)； 则Dmutex=0

process B: P(Emutex)； 则Emutex=0

process A: P(Emutex)； 此时Emutex=-1 A阻塞

process B: P(Dmutex)； 此时Dmutex=-1 B阻塞

2.4.3 信号量机制

* 记录型信号量机制讨论

  - 1> 若信号量S.value为正值，其表示某类资源还可以使用的数量。
  - 2> 若信号量S.value为负值，则其绝对值表示在信号量链表中已阻塞进程的数目。
  - 3> 在一定条件下，P操作代表进程的阻塞操作，而V操作代表被阻塞进程的唤醒操作。
  - 4> 如果S.value的初值为1，表示该类资源只允许一个进程访问临界资源，此时的信号量转化为互斥信号量。

2.4.3 信号量机制

- 5> P、V操作必须成对出现，有一个P操作就一定有一个V操作;当为互斥操作时，它们处于同一进程;当为同步操作（使两个进程合作）时，则不在同一进程中出现。
- 6> 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要，否则容易引起死锁；一个同步P操作与一个互斥P操作在一起时，同步P操作应在互斥P操作前（why??）；而两个V操作的顺序则无关紧要。

2.4.3 信号量机制

❖ 3、AND型信号量机制——实现多类资源共享

* AND型信号量也称AND型信号量集，由若干个需做“ AND”条件操作的信号量组成。
* 基本思想：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。在对若干类资源分配和释放过程中，采取原子操作方式，即要么一次性全部分配到进程，要么一个也不分配；释放亦如此。这样就可避免上述记录型信号机制中死锁情况的发生。为此，在P、V操作中，增加了一个“AND”条件，故称为AND同步，或称为同时P操作、同时V操作。

2.4.3 信号量机制

* 原子操作Swait(S1,S2, …,Sn)

```
Swait(S1,S2, …,Sn)
 if  S1≥1 and … and Sn≥1  then //各类资源是否还可分配
       for i=1 to n do
            Si=Si-1;  //每类资源都要分配1个
       endfor
else //否则，将进程放到等待资源Si的队列中
       调进程到第一个小于1的信号量的等待队列Si.queue;
       阻塞该进程;
endif
```

2.4.3 信号量机制

* 原子操作Ssignal(S1,S2 …,Sn)

```
Ssignal(S1,S2, …,Sn)
for i=1 to n do
    Si=Si+1;  //释放进程占用的各类资源
    从等待队列Si.queue中取出第一个进程P，唤醒进
    程P，移到就绪队列中;
endfor;
```

2.4.3 信号量机制

❖ 4、信号量集机制

* 信号量集是指由多个信号量组成的集合，其每一信号量代表着一类分配时存在一定下限要求、且一次需分配一定数量的资源。
* 信号量集机制的基本思路就是在AND型信号量机制的基础上进行扩充，在一次原语操作中完成所有的资源申请。
* 由于AND型信号量有时也称AND型信号量集，为此通常将信号量集称为一般信号量集加以区别。

2.4.3 信号量机制

di、ti--第i类资源的需求量、可分配下限值

* 原子操作Swait(S1,t1,d1; …; Sn,tn,dn)

```
Swait(S1,t1, d1; … ; Sn,tn,dn)
 if   S1≥t1 & S1 ≥d1  and … and Sn≥tn &Sn ≥dn then
        for i=1 to n do
             Si=Si-di;   //按要求数量di分配第i类资源
        endfor
else //否则，将进程放到等待资源Si的队列中
        调进程到第一个小于ti信号量的等待队列Si.queue;
        阻塞调用进程;
endif
```

2.4.3 信号量机制

* 原子操作Ssignal(S1,t1,d1; …; Sn,tn,dn)

```
Ssignal(S1,t1,d1; … ; Sn,tn, dn)
for i=1 to n do
    Si=Si+di;  //释放进程占用的全部资源
    从等待队列Si.queue中取出第一个进程P;
    if 判断进程P是否通过Swait中的测试 then
        唤醒进程P，移到就绪队列中;
    else
        调进程P到第一个小于tj的信号量的等待队列Sj.queue;
    endif;
endfor;
```

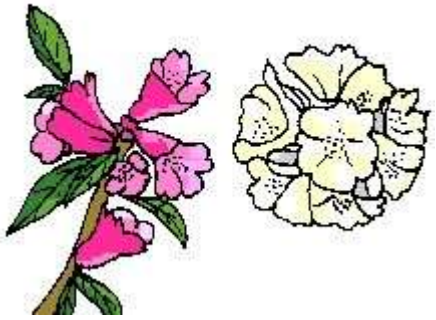
2.4.3 信号量机制

* 信号量集可以用于各种情况的资源分配和释放，下面是几种特殊情况：

  - **Swait(S, d, d)** 表示每次申请d个资源，当总资源少于d个时，便不分配。
  - **Swait(S, 1, 1)** 退化成一般记录型信号量，当S=1时表示互斥信号量。
  - **Swait(S, 1, 0)** 可作为一个可控开关（di=0表示用户即使进入临界区也不用分配资源，因此，当S≥1时，将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）。

2.4 进程同步

- 2.4.1 进程同步的基本概念
- 2.4.2 硬件同步机制
- 2.4.3 信号量机制
- 2.4.4 信号量的应用
- 2.4.5 管程机制



2.4.4 信号量的应用

❖ 1、利用信号量实现互斥

* 设S的初值为1，则P、V操作控制临界资源互斥访问描述如下：

```
repeat
        P(S);
        critical section;
        V(S);
        remainder section;
until false;
```

2.4.4 信号量的应用

❖ 2、利用信号量来描述前趋关系

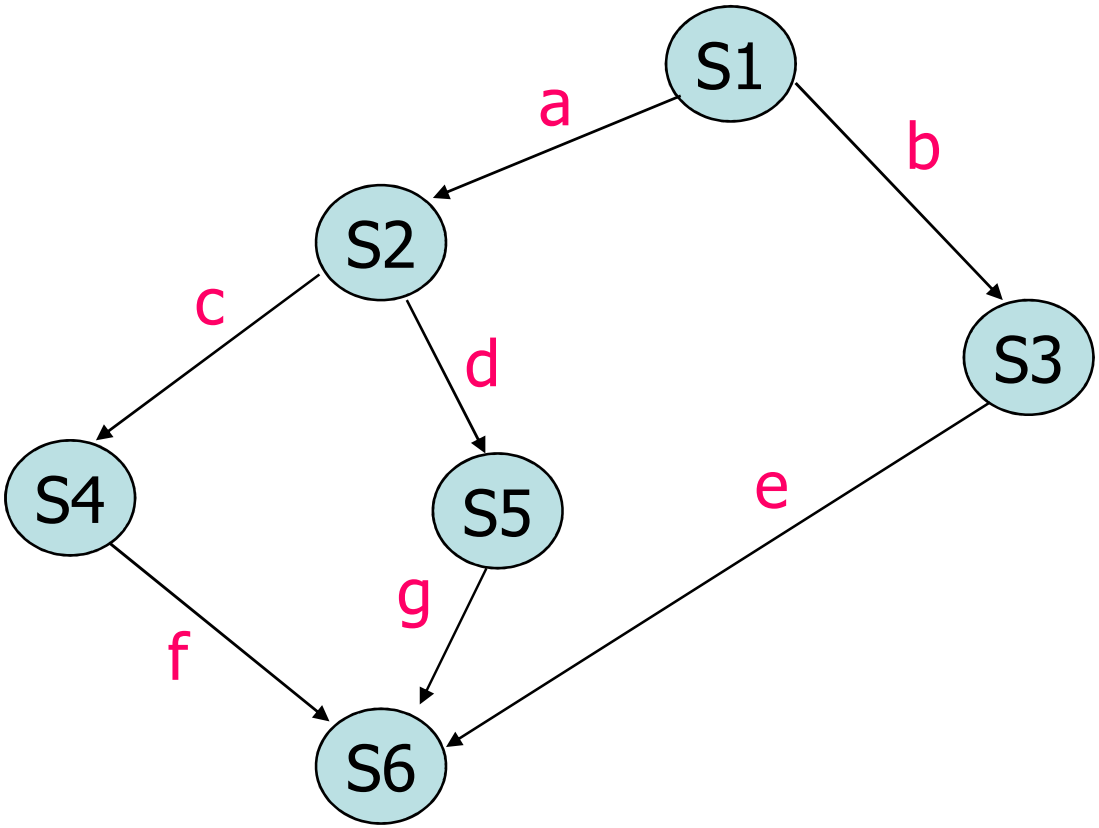
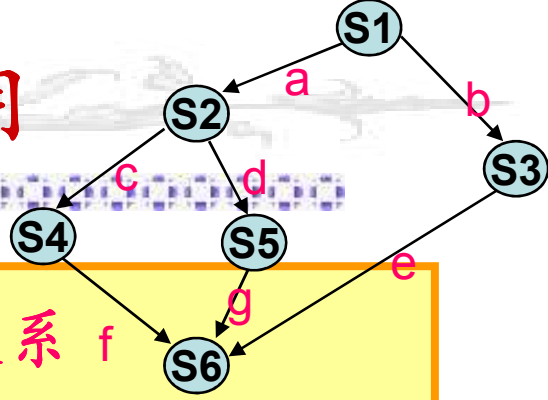


图2－10 前趋图举例

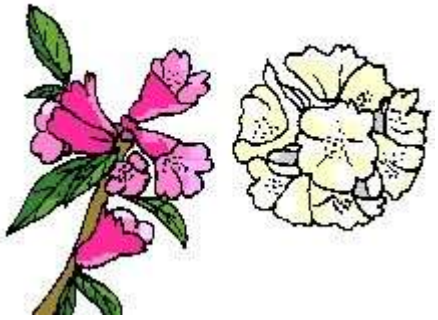
2.4.4 信号量的应用




```
//例题：利用信号量来描述图2-10前趋关系
Semaphore a=0，b=0，c=0，d=0，e=0，f=0，g=0；
Begin
   parbegin
     begin  S1；  signal(a)；  signal(b)；  end；
     begin  wait(a)；S2；  signal(c)；  signal(d)；  end；
     begin  wait(b)；S3；  signal(e)；  end；
     begin  wait(c)；S4；  signal(f)；  end；
     begin  wait(d)；S5；  signal(g)；  end；
     begin  wait(e)；  wait(f)；wait(g)；S6；  end；
   parend
end
```

2.4 进程同步

- 2.4.1 进程同步的基本概念
- 2.4.2 硬件同步机制
- 2.4.3 信号量机制
- 2.4.4 信号量的应用
- 2.4.5 管程机制



2.4.5 管程机制

❖ 1、信号量机制的缺点

信号量机制处理同步问题时，共享变量及信号量的操作分散在各个进程中并遍布整个程序，不仅给程序的管理、维护、修改带来了麻烦，而且还会因同步操作的不当造成死锁。

* 1>可读性差：要了解对于一组共享变量及信号量的操作是否正确，则必须通读整个系统或者并发程序。
* 2>不利于修改和维护：程序的局部性很差，所以任一组变量或一段代码的修改都可能影响全局。
* 3>正确性难以保证：操作系统或并发程序通常很大，要保证在一个复杂的系统中没有逻辑错误是很难的。

2.4.5 管程机制

❖ 2、管程（Monitor）的引入

* 基于信号量机制存在的问题，Dijkstra于1971年提出，把分散在各进程中的临界区集中起来进行管理，并把系统中的共享资源用数据结构抽象地表示出来。由于临界区是访问临界资源的代码段，建立一个“秘书”程序管理到来的访问。“秘书”每次仅让一个进程来访，这样既便于对临界资源的管理，又实现了互斥访问。
* 1973年Hansan和Hoare又把”秘书”进程思想发展为管程概念。采用这种方法，对共享资源的管理就可借助数据结构及在其上实施操作的若干过程来进行。而代表共享资源的数据结构及对这些数据结构进行操作的一组过程就构成了管程。并发进程通过调用管程来实现共享资源的请求和释放。

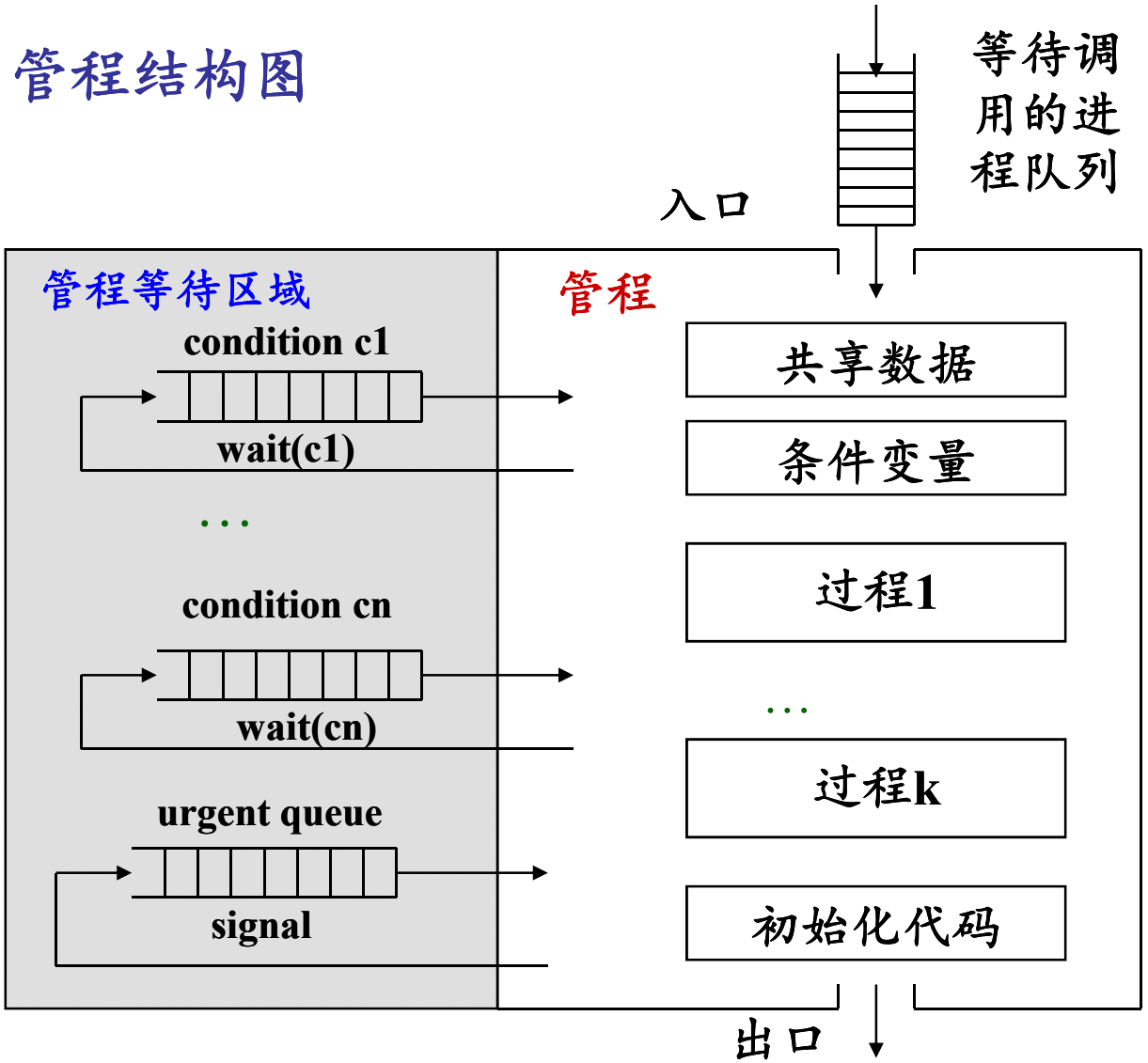
2.4.5 管程机制

❖ 3、管程（Monitor）的定义

* 书中定义及Hansan的定义（详见P58）
* 管程的组成：
  - 1> 管程的名称；
  - 2> 局部于管程内部的共享数据结构的说明；
  - 3> 对局部于管程的共享数据设置初始值的语句；
  - 4> 对该数据结构进行操作的一组过程。
* 我的定义：管程是由 一组代表着共享资源的局部变量、对局部变量进行初始化的语句序列和操作的过程以及为了实现进程同步而引入的一组条件变量 共同构成的一种资源管理模块。

2.4.5 管程机制

* 管程结构图



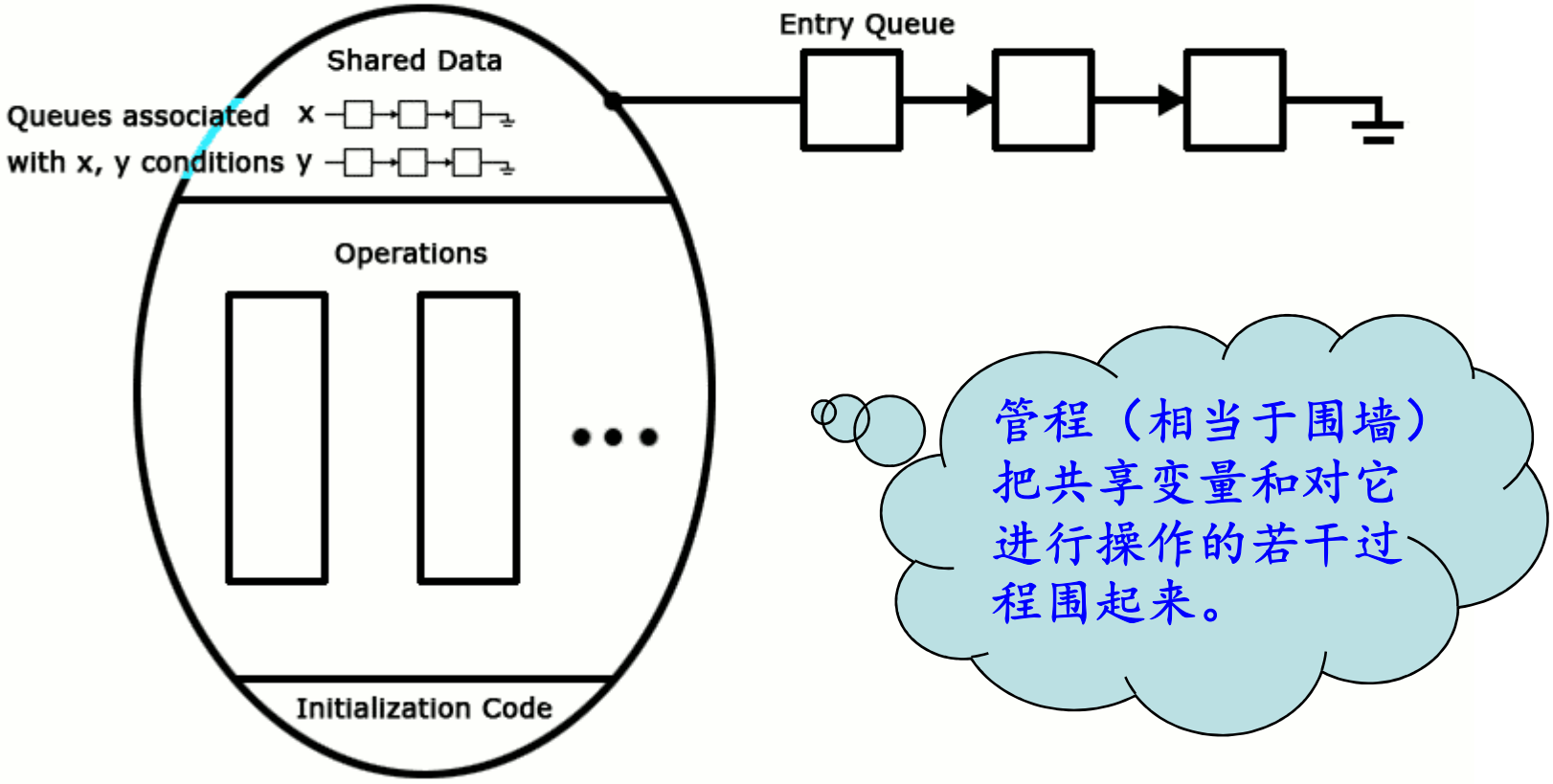
2.4.5 管程机制

* 管程的语法描述

```
type monitor-name=monitor;
share variable declarations; // 声明共享变量
cond declarations;  // 声明条件变量
define 本管程内可供其它外部模块调用的过程名列表;
Use   本管程内需要调用的其它外部过程名列表;
procedure 过程名(…)
      begin … end;
      …
procedure 过程名(…)
      begin … end;
begin
      initialization code;
end
```

2.4.5 管程机制

* 管程示意图



2.4.5 管程机制

* 管程的特性
  - 1> 模块化
    - 管程将进程的同步操作机制和临界资源结合到一起。
    - 一个管程是一个基本程序单位，可以单独编译。
  - 2>抽象性（抽象数据类型）
    - 管程内的共享数据结构抽象地表示系统中的共享资源。
  - 3>安全性（信息掩蔽）
    - 管程中的数据结构只能被管程内部的过程访问；虽然有部分管程内的过程可供其它外部模块调用，但管程中的数据结构以及过程的实现对外部模块来说是不可见的。

2.4.5 管程机制

- 4> 互斥性
  - 在任一时刻，最多只能有一个共享资源的进程能真正地进入管程，而任何其他调用者必须等待，直到访问者退出。
- 5> 共享性
  - 管程中的过程可被所有需调用管程的进程所共享，进程通过调用管程中的过程而进入管程。
- 6> 集中性
  - 管程把分散在各个进程中那些需互斥访问的临界区集中了起来。

2.4.5 管程机制

* 管程与进程比较

- 1> 设置目的不同：设置进程是为了实现系统的并发性，设置管程则是解决共享资源的互斥与同步使用；
- 2> 数据结构不同：管程定义的是公用数据结构，而进程定义的是私有数据结构；
- 3> 数据结构上的操作不同：进程是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；
- 4> 工作方式不同：管程被进程调用，为被动工作方式，进程则为主动工作方式；
- 5> 并发不同：管程和调用它的进程不能并发工作，而进程之间能并发工作，并发性是其固有特性；
- 6> 生命周期不同：管程是操作系统的固有成分，无创建和撤消；而进程有生命周期，由创建而产生、由撤销而消亡。

2.4.5 管程机制

❖ 4、管程实现同步—— 条件变量

* 管程在同一时刻只允许有一个进程进入，调用管程的进程如果因请求的共享资源而未获满足时，将在管程中被阻塞，此时必须将该进程调用的管程加以释放，否则将会导致别的进程也将无法进入管程，为此在管程中引入了条件变量。

* 条件变量是一种局部于管程内部的一种数据结构，每个条件变量设有一个链表，用来记录因不满足相应条件而被阻塞的所有进程。

2.4.5 管程机制

* 条件变量只能通过wait和 signal两个原语操作访问：
  - **wait（条件变量）：** 阻塞当前调用管程的进程并将其插入条件变量的阻塞队列中，释放管程。
  - **signal（条件变量）：** 从条件变量的阻塞队列中唤醒一个进程；如果条件变量的阻塞队列为空，则相当于做空操作，执行此操作的进程继续。
* 条件变量与信号量机制中的信号量的区别
  - 条件变量也是一种信号量，但它并不代表共享资源的数量，实际上只是一个链表结构，对其作wait操作时用来记录不满足相应条件而阻塞的所有进程；而信号量机制中的信号量通常代表着一类资源的可用数量，作wait或signal操作时通常需修改信号量的大小。

2.4.5 管程机制

* 条件变量signal操作问题

使用signal唤醒被阻塞进程时，可能出现两个进程同时停留在管程内，这是绝对不允许的，解决方法：

- 方法1：执行signal的进程P等待，直到被唤醒进程Q退出管程或因等待另一个条件再次被阻塞。
- 方法2：被唤醒进程Q等待，直到执行signal的进程P退出管程或因等待另一个条件被阻塞。

Hoare采用第一种办法；Hansan选择两者的折衷，规定管程中的过程所执行的signal操作是该过程体的最后一个操作。

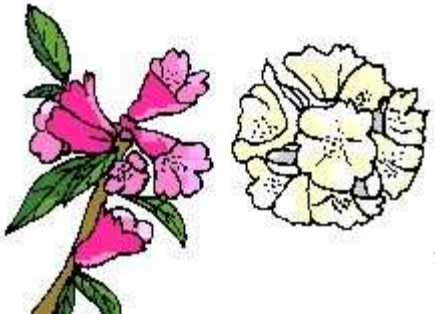
2.4.5 管程机制

* 管程的实现示例

```
TYPE SSU = MONITOR
 var busy : boolean;          /* busy 表示共享资源的忙闲状态*/
     nobusy : condition;      /* nobusy为条件变量*/
 define require, release;  /*本管程内可供其它外部模块调用的过程名*/
 use wait, signal;  /*本管程内需要调用的其它外部过程名*/
 procedure require
   begin
     if busy then wait(nobusy);  /*将当前进程加入条件等待队列*/
     else busy := true;
   end;
 procedure release
   begin
     busy := false;
     signal(nobusy);                /*从条件等待队列中释放进程*/
   end;
 begin                              /*管程变量初始化*/
   busy := false;
 end;
```

2.5 进程同步经典问题

- 2.5.1 生产者--消费者问题
- 2.5.2 哲学家进餐问题
- 2.5.3 读者--写者问题
- 补充例题：独木桥问题

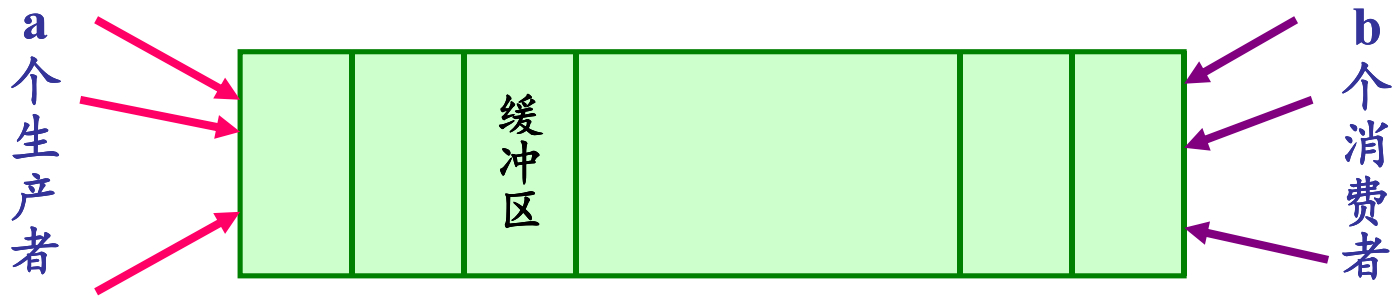


2.5.1 生产者-消费者问题

❖ 1、问题描述

a个生产者生产产品供b个消费者消费，在两者之间设置一个具有n个缓冲区的缓冲池用来存储生产者生产的产品（每个生产者每次生产1个产品、占一个缓冲区），消费者从缓冲池取走产品去消费（每个消费者每次取1个产品）。

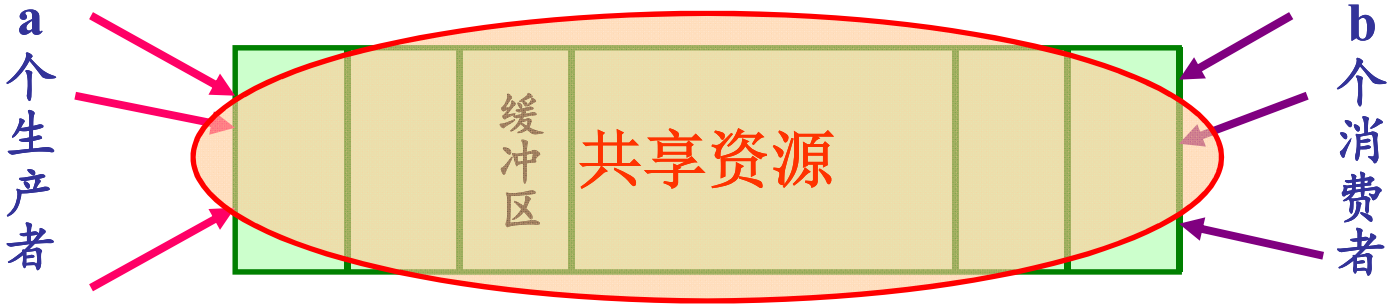
现要求生产者的生产和消费者的消费保持同步：即不允许生产者向一个已装满产品且尚未被取走的缓冲区中继续投放产品，也不允许消费者到一个空缓冲区中去取产品。



拥有n个缓冲区的缓冲池

2.5.1 生产者-消费者问题

❖ 2、利用记录型信号量解决生产者-消费者问题



拥有n个缓冲区的缓冲池

* 信号量设置

- mutex：互斥信号量控制缓冲池访问　初值为1
- empty：记录缓冲池中空缓冲区数目　初值为n
- full　：记录缓冲池中满缓冲区数目　初值为0

* 缓冲池表示

- buffer[0,1,…,n-1]：用一个长度为n的数组表示
- in:　生产者在缓冲池中拟插入产品的位置
- out:　消费者在缓冲池中拟取走产品的位置

2.5.1 生产者-消费者问题

* 利用记录型信号量描述生产者-消费者问题

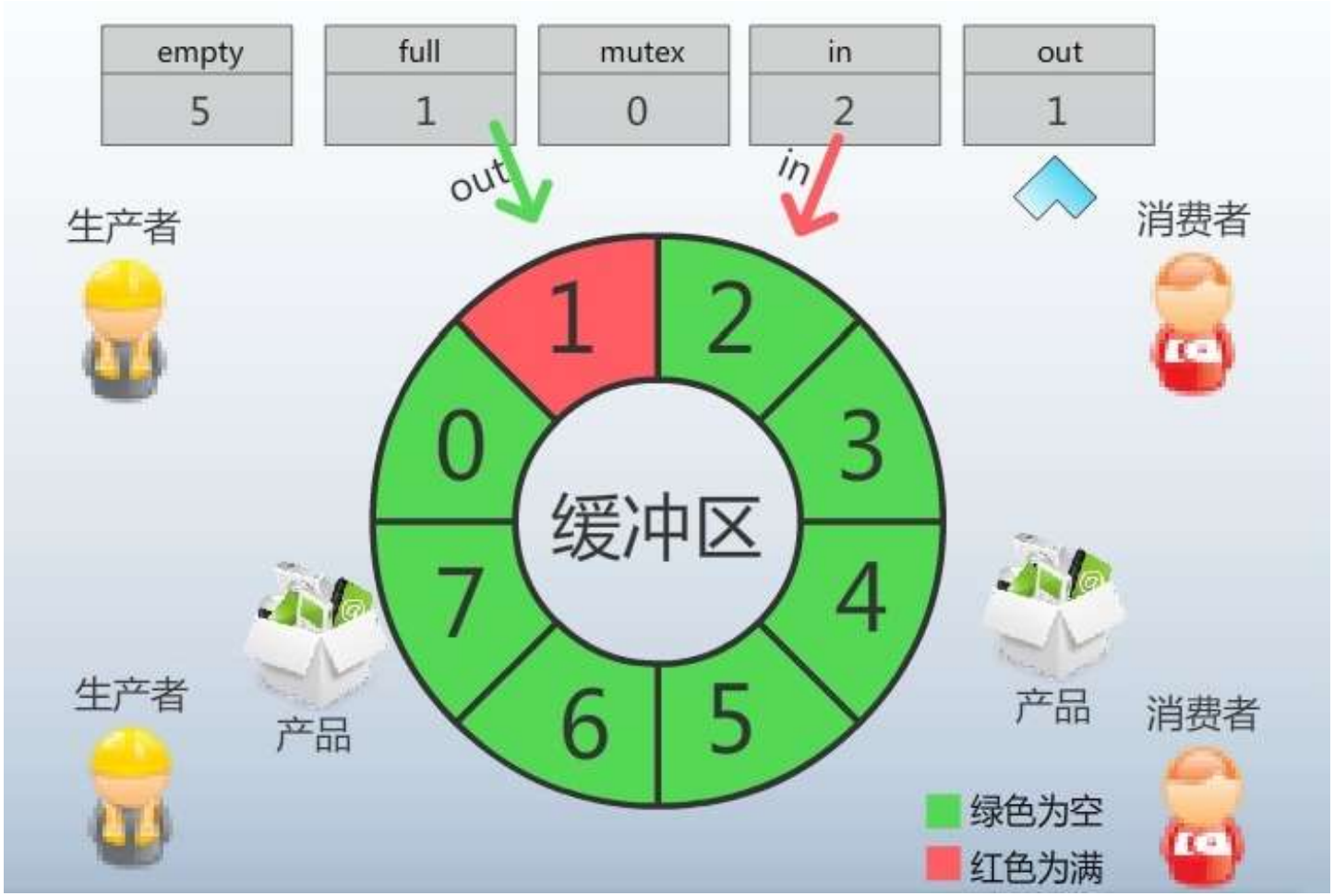
```
var mutex，empty，full：semaphore：=1，n，0；
buffer：array[0，1，…，n-1] of item；
in，out：integer：=0，0；
```

```
producer (i)
    repeat
        生产一个产品=>nextp;
        wait(empty);
        wait(mutex);
        buffer(in):=nextp;
        in:=(in+1)mod n;
        signal(mutex);
        signal(full);
    until false;
```

```
consumer (j)
    repeat
        wait(full);
        wait(mutex);
        nextc: =buffer(out);
        out:=(out+1) mod n;
        signal(mutex);
        signal(empty);
        消费掉产品nextc;
    until false;
```

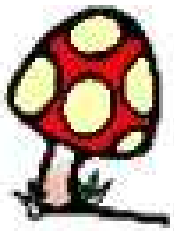
2.5.1 生产者-消费者问题

* 动画演示记录型信号量解决生产者-消费者问题



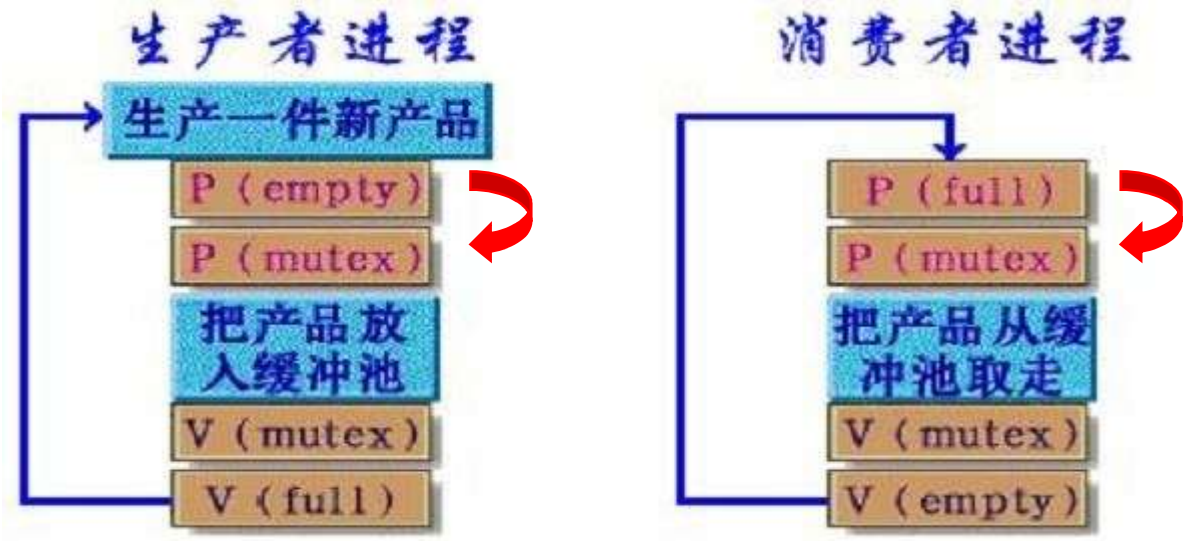
2.5.1 生产者-消费者问题

* 小结（在PPT80页我已讲过）
  - 1> P、V操作必须成对出现，有一个P操作就一定有一个V操作;当为互斥操作时，它们同处于同一进程;当为同步操作时，则不在同一进程中出现。
  - 2> 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要，否则容易引起死锁；一个同步P操作与一个互斥P操作在一起时，同步P操作应在互斥P操作前；而两个V操作其顺序则无关紧要。



2.5.1 生产者-消费者问题（作业）

* 问题：生产者、消费者的P操作的顺序可换吗？



- 提示1：如果生产者的P操作互换，假设此时缓冲池已满，如果生产者又生产了一件产品，会出现什么情况？
- 提示2：如果消费者的P操作互换，假设此时缓冲池已空，如果消费者又打算消费一件产品，会出现什么情况？

2.5.1 生产者-消费者问题

❖ 3、利用AND型信号量解决生产者-消费者问题

* 基本思路：在前面我们已经讲过记录型信号量机制在有多个信号量时容易出现死锁情况，因此我们可用AND型信号量替代多个记录型信号量来解决生产者-消费者问题：

生产者		消费者	
wait(empty) wait(mutex)	Swait(empty, mutex)	wait(full) wait(mutex)	Swait(full, mutex)
signal(mutex) signal(full)	Ssignal(mutex, full)	signal(mutex) signal(empty)	Ssignal(mutex, empty)

2.5.1 生产者-消费者问题

* 利用AND型信号量描述生产者-消费者问题

```
var mutex，empty，full：semaphore：=1，n，0；
buffer：array[0，1，…，n-1] of item；
in，out：integer：=0，0；
```

```
producer (i)
    repeat
        生产一个产品=>nextp;
        Swait(empty, mutex) ;
        buffer(in):=nextp;
        in:=(in+1)mod n;
        Ssignal(mutex，full);
    until false;
```

```
consumer (j)
    repeat
        Swait(full,mutex);
        nextc: =buffer(out);
        out:=(out+1) mod n;
        Ssignal(mutex, empty) ;
        消费掉产品nextc;
    until false;
```

2.5.1 生产者-消费者问题

❖ 4、利用管程解决生产者-消费者问题

* 管程名称
  - producer-consumer，在后面的伪代码中将其记为PC
* 共享局部变量
  - buffer[0,1, …,n-1]：用一个长度为n的数组表示
  - in: 生产者在缓冲池中拟插入产品的位置
  - out: 消费者在缓冲池中拟取走产品的位置
  - count：缓冲池中现有产品数目
* 条件变量
  - notfull: 如果缓冲池中已满，则将进入管程的生产者进程存在此条件变量的链表中。
  - notempty：如果缓冲池中已空，则将进入管程的消费者进程存在此条件变量的链表中。

2.5.1 生产者-消费者问题

* 过程
  - 1> put(nextp)过程：生产者利用该过程将自己生产的产品nextp投放到缓冲池中，当count≥n时，表示缓冲池已满，生产者须等待。
  - 2> get( )过程：消费者利用该过程从缓冲池中取出一个产品，当count≤0时，表示缓冲池中已无可取用的产品，消费者应等待。
* 初始化语句序列
  - in=0;
  - out=0;
  - count=0;

2.5.1 生产者-消费者问题

* 生产者-消费者管程描述

```
type PC=monitor
var in,out,count: integer;
buffer: array[0, …, n-1] of item;
notfull，notempty:condition;
procedure put(nextp)      //nextp表示生产出的产品
    begin
      if count>=n then   //缓冲池已满
         notfull.wait;    //阻塞当前生产进程
      buffer(in):=nextp;
      in:=(in+1) mod n;
      count:=count+1;
      if notempty.queue.length>0 then
         notempty.signal;  //唤醒被一个被阻塞消费者进程
    end
```

2.5.1 生产者-消费者问题

```
procedure get( )
    begin
      if count<=0 then      //缓冲池已空
         notempty.wait;    //阻塞当前消费者进程
      nextc:=buffer(out);  //nextc用来保存取出的产品
      out:=(out+1) mod n;
      count:=count-1;
      if notfull.quene.length>0 then
         notfull.signal;   //唤醒被一个被阻塞生产者进程
      return nextc;
    end
begin
   in:=0;
   out:=0;
   count:=0
end
```

2.5.1 生产者-消费者问题

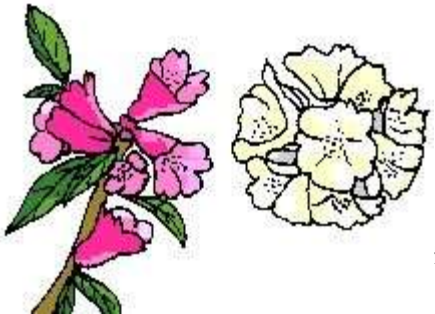
* 调用已定义管程描述生产者、消费者

```
producer(i)
   begin
      repeat
         生产一个产品item;
         PC.put(item);
      until false;
   end
```

```
consumer(j)
   begin
    repeat
       nextc=PC.get( );
       消费掉产品nextc;
    until false;
   end
```

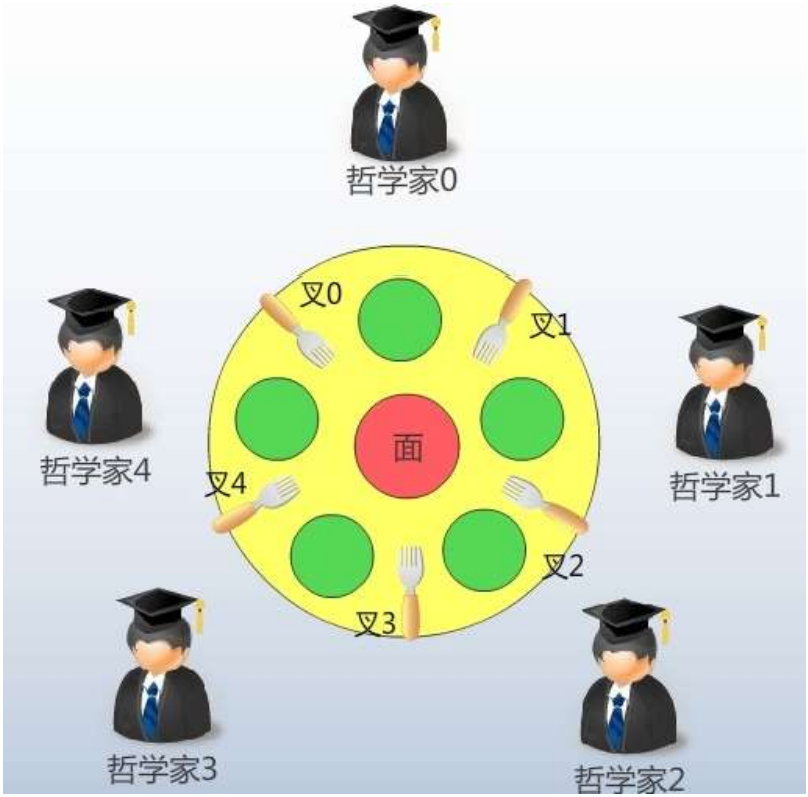
2.5 进程同步经典问题

- 2.5.1 生产者--消费者问题
- 2.5.2 哲学家进餐问题
- 2.5.3 读者--写者问题
- 补充例题：独木桥问题



2.5.2 哲学家进餐问题

❖ 1、问题描述



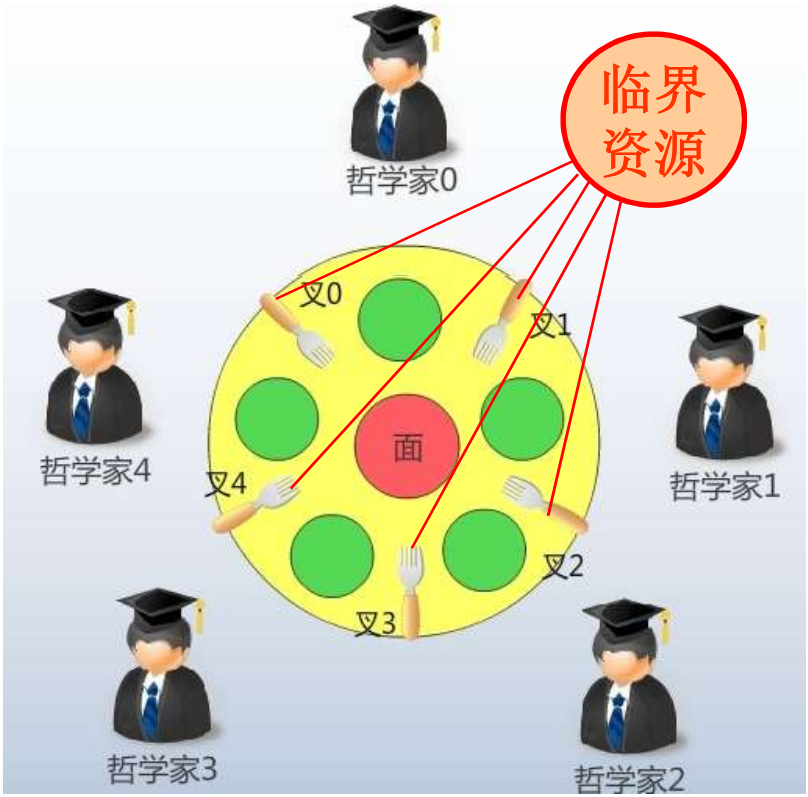
有五个哲学家围坐在一圆桌旁，桌中央有一盘通心粉，每人面前有一只空盘子，每两人之间放一只筷子。

设哲学家的生活方式为交替地进行思考和进餐，饥饿时便拿起两边的筷子进餐，但只有当拿到两支后才能进餐。用餐毕，放下筷子。

要求每个哲学家都能进餐？

2.5.2 哲学家进餐问题

❖ 2、利用记录型信号量解决哲学家进餐问题



* 信号量设置

- **chopstick[0,…,4]:** 数组中每一个元素表示一只筷子，初值均为1，用来控制每一只筷子的互斥访问

2.5.2 哲学家进餐问题

* 利用记录型信号量描述生产者-消费者问题

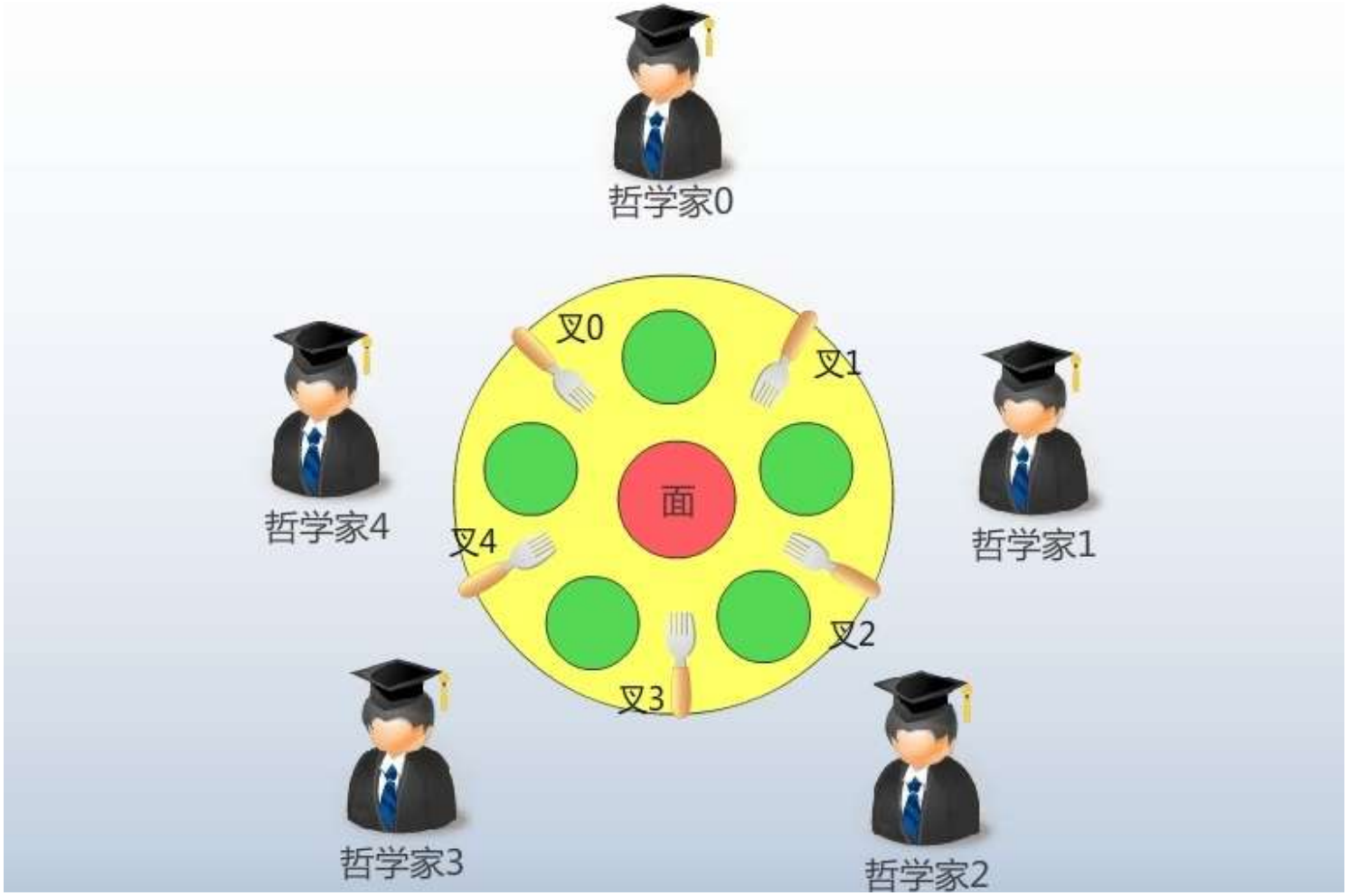
```
var chopstick: array［0, …, 4］of semaphore:=(1,1,1,1,1);
```

```
philosophers(i)
    begin
        repeat
            think;
            wait(chopstick[i]);
            wait(chopstick[(i+1) mod 5]);
            eat;
            signal(chopstick[i]);
            signal(chopstick[(i+1) mod 5]);
        until false;
    end
```

死锁

2.5.2 哲学家进餐问题

* 动画演示上述哲学家进餐过程中存在的死锁问题



2.5.2 哲学家进餐问题

* 如何解决上述哲学家进餐死锁问题？
  - 方法1：至多只允许有四位哲学家同时先去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。（代码详见演示动画）
  - 方法2：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。（代码见下页）
  - 方法3：仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。（用AND型信号量实现）

2.5.2 哲学家进餐问题

* 利用方法2解决哲学家进餐问题

```
var chopstick: array［0, …, 4］of semaphore:=(1,1,1,1,1);
```

```
philosophers(i)
 begin
  repeat
     think;
    if(i%2==1){wait(chopstick[i]); wait(chopstick[(i+1) mod 5]);}
    else{wait(chopstick[(i+1) mod 5]); wait(chopstick[i]);}
     eat;
    if(i%2==1){signal(chopstick[i]);signal(chopstick[(i+1) mod 5]);}
    else{signal(chopstick[(i+1) mod 5]); signal(chopstick[i]);}
  until false;
 end
```

2.5.2 哲学家进餐问题

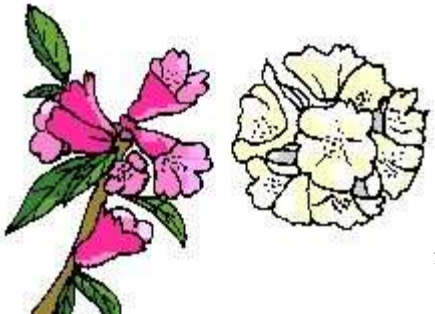
❖ 3、利用AND型信号量解决哲学家进餐问题

```
var chopstick: array［0, …, 4］of semaphore:=(1,1,1,1,1);
```

```
philosophers(i)
    begin
        repeat
            think;
            Swait(chopstick[i], chopstick[(i+1) mod 5]);
            eat;
            Ssignal(chopstick[i]), signal(chopstick[(i+1) mod 5]);
           until false;
    end
```

2.5 进程同步经典问题

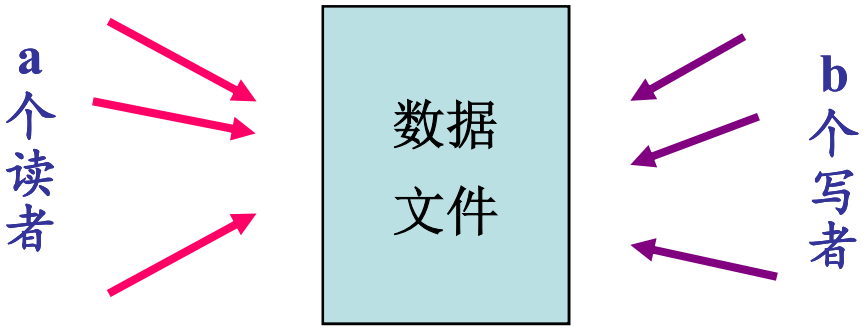
- 2.5.1 生产者--消费者问题
- 2.5.2 哲学家进餐问题
- 2.5.3 读者--写者问题
- 补充例题：独木桥问题



2.5.3 读者-写者问题

❖ 1、问题描述

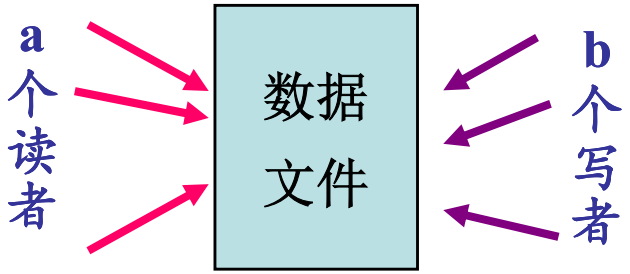
有两组并发进程：读者和写者，共享一个数据文件或记录，要求：允许多个读者进程同时执行读操作（因为读操作不会使数据文件混乱），但不允许一个写者进程和其它读者或写者进程同时访问共享对象（这种访问会使数据文件混乱）。



2.5.3 读者-写者问题

❖ 2、利用记录型信号量解决读者-写者问题

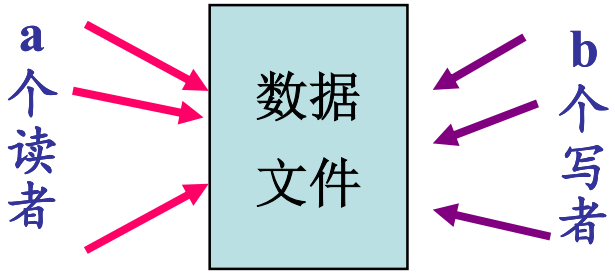
* 情景分析



- 如果有新读者想读：
  - 1> 有写者写，则新读者等
  - 2> 无写者写，则新读者可以读
  - 3> 有其它读者正在读，则新读者也可以读
- 如果有新写者想写：
  - 1> 有读者读，新写者等待
  - 2> 有其它写者写，新写者等待
  - 3> 无读、写者，新写者可以写

2.5.3 读者-写者问题

* 信号量与共享变量设置



- 互斥信号量**wmutex**：初值为1，控制写操作互斥访问
- 共享变量**readcount**：初值为0，统计数据文件同时被多少个读者在读，因为写者仅当**readcount==0**才可以进行写操作
- 互斥信号量**rmutex**：初值为1，因为变量readcount值为多个读者共享，但一段时间内只允许有一个读者可以修改它，因此readcount对各读者来说必须互斥访问

2.5.3 读者-写者问题

* 问题：为什么我没有将readcount设置成信号量，而在生产者-消费者问题中将同样起着统计作用的empty 和full却设置成了信号量？
* 原因：
  - 因为在这里我们并没有对共享数据文件的最大并发读者数量进行限定，也就是说多个读者在并发访问数据文件时并不存在竞争问题，当然更不存在合作问题，因此没有必要将其设为信号量。
  - 考虑到多个读者都将对readcount进行修改，当然你也可以将其设为信号量，用一个非常大的值做为上限，每有一个读者则readcount减1，此时：读者数目=上限值-readcount。（详参本课件PPT135页的信号量L的用法）

2.5.3 读者-写者问题

* 利用记录型信号量描述读者-写者问题

```
var wmutex，rmutex：semaphore：=1，1；
readcount： integer： =0；
```

```
reader (i)
    repeat
        P(rmutex);
        if (readcount==0)
            P(wmutex);
        readcount++;
        V(rmutex);
        读操作;
        P(rmutex);
        readcount --;
        if (readcount==0)
            V(wmutex);
        V(rmutex);
    until false;
```

```
writer (j)
    repeat
        P(wmutex);
        写操作;
        V(wmutex);
    until false;
```

2.5.3 读者-写者问题

❖ 3、利用信号量集机制解决读者-写者问题

* 假设
  - 最多只允许RN个读者同时读
* 信号量设置
  - **L：记录并控制读者数目，初值为RN。如有一个读者在读，则L减1；如果没有读者在读，则为RN**

    注意：此处L也起到了记录读者数目的作用，但与前面记录型信号量中的readcount有一些不同，此处：读者数目=RN-L，而前面的读者数目＝readcount
  - **wmutex：初值为1，控制写操作互斥访问**

2.5.3 读者-写者问题

* 利用信号量集描述读者-写者问题

```
var wmutex，L：semaphore：=1，RN；
```

```
reader (i)
    repeat
        //读者数量是否到上限
        P(L,1,1);
        //是否有写者在写
        P(wmutex,1,0);
        读操作;
        V (L,1)
    until false;
```

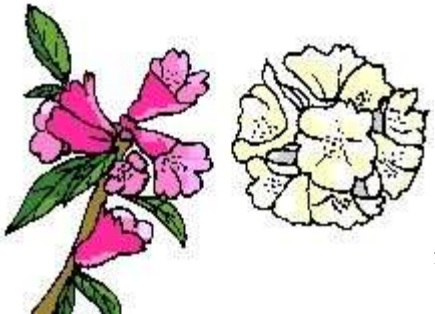
可控开关

```
writer (j)
    repeat
        //是否有写者或读者在访问
        P(wmutex,1,1; L,RN,0);
        写操作;
        V(wmutex,1);
    until false;
```

可控开关

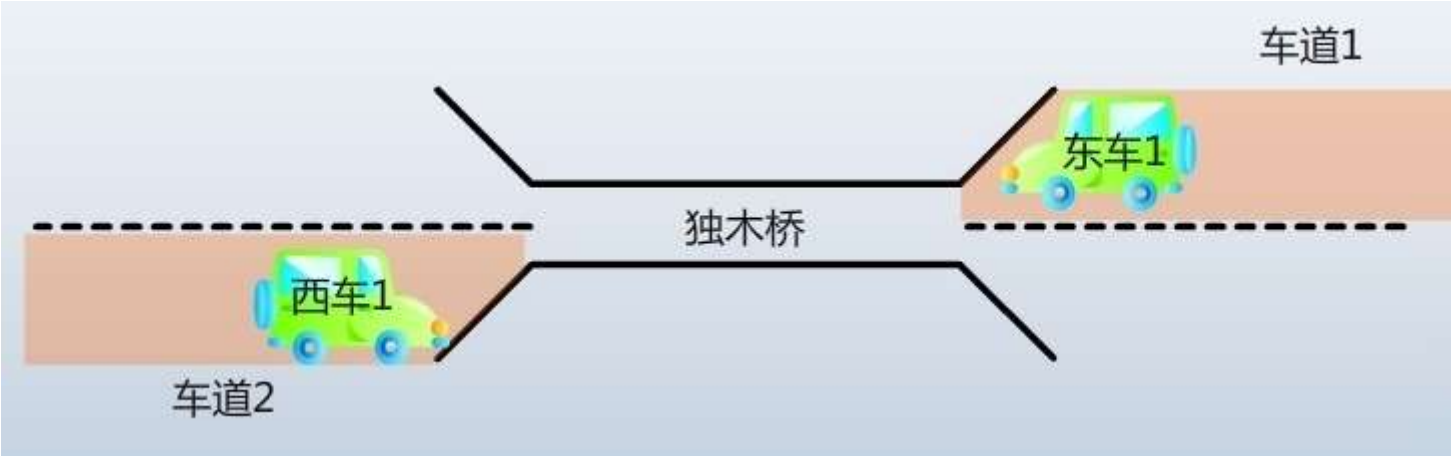
2.5 进程同步经典问题

- 2.5.1 生产者--消费者问题
- 2.5.2 哲学家进餐问题
- 2.5.3 读者--写者问题
- 补充例题：独木桥问题



补充例题：独木桥问题（自学）

- **问题描述：**现有一东西向的独木桥，现要设计一个自动管理系统，管理规则如下:1>当独木桥上有车辆在行驶时同方向的车也可以跟着驶入独木桥，但另一方向的车必须在独木桥外等待；2>当独木桥上无车时，到达两个桥头的车辆都可以进入，但不能从同时驶入；3>当在独木桥上朝某方向行驶的车辆驶出了独木桥且无后续车辆进入驶入，应让另一方向等待的车辆驶入独木桥。请用记录型信号量对独木桥通行实现正确管理。



补充例题：独木桥问题

- 问题分析：
  - 1> 应写两段程序分别代表桥的东、西两个方向上的车辆，设为east(i)、west(j)；
  - 2> 对east(i)、west(j)进程来说独木桥是临界资源，因此需要设置一个互斥信号量，记为bmutex，初值为1；
  - 3> 对每一车辆来说，如果桥上有同方向的车存在，则可跟着驶上桥，因此需要分别设置两个共享变量统计东、向西方向行驶的车辆数量供后续车辆参考是否可以过桥，记为ecount、wcount，初值均为0；
  - 4> 由于同方向可能同时有多个车辆过桥，这些车辆都可能修改ecount或者 wcount，因此ecount、wcount对东、西方向的车辆来说均是临界资源，故需设置两个互斥信号量，分别记为emutex、wmutex，初值均为1。

补充例题：独木桥问题

* 利用记录型信号量描述独木桥问题

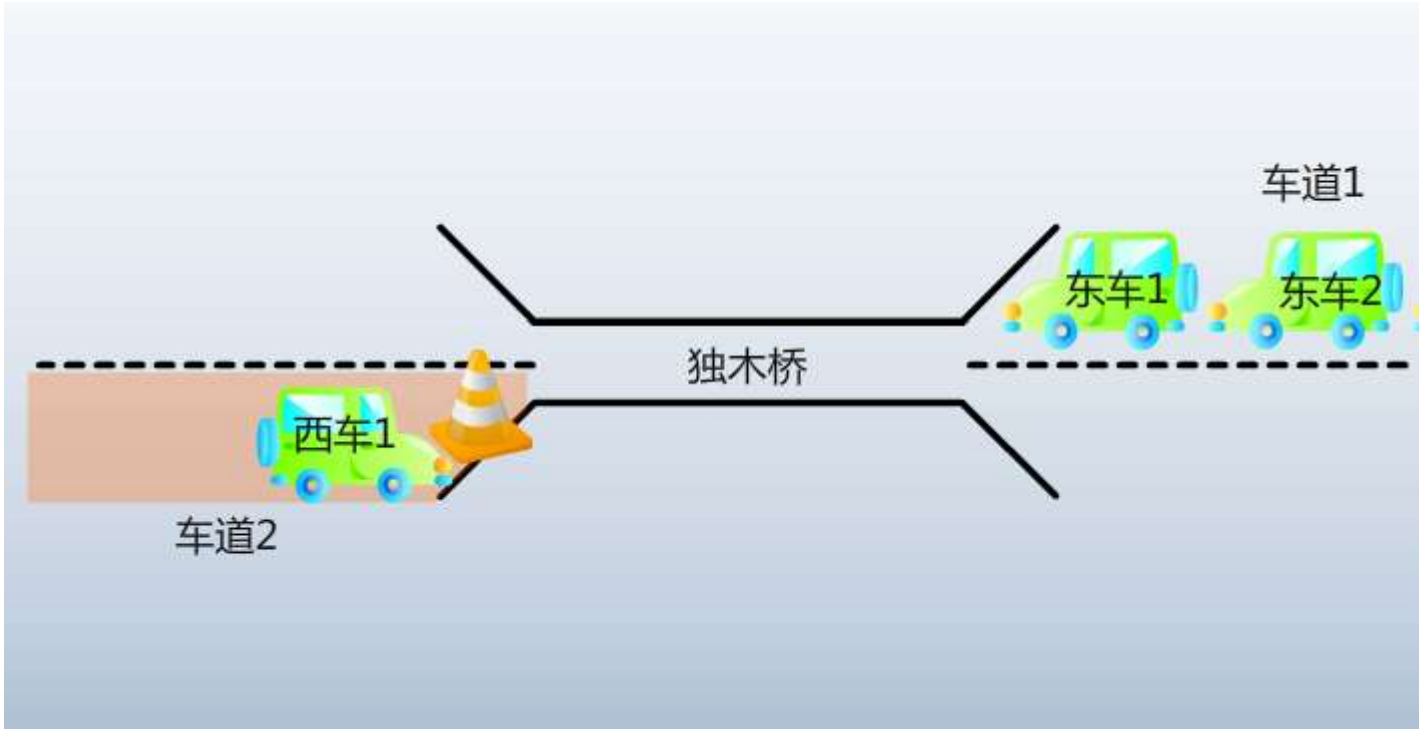
```
var bmutex，emutex，wmutex：semaphore：=1，1，1；
ecount，wcount： integer： =0，0；
```

```
east (i)
    P(emutex);
    ecount++;
    if(ecount==1)  P(bmutex);
    V(emutex);
    过桥;
    P(emutex);
    ecount--;
    if(ecount==0)  V(bmutex);
    V(emutex);
```

```
west (j)
    P(wmutex);
    wcount++;
    if(wcount==1)  P(bmutex);
    V(wmutex);
    过桥;
    P(wmutex);
    wcount--;
    if(wcount==0)  V(bmutex);
    V(wmutex);
```

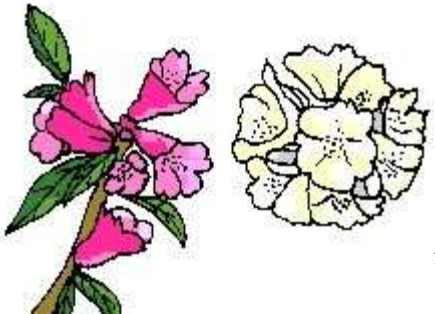
补充例题：独木桥问题

* 动画演示记录型信号量解决独木桥问题



2.6 进程通信

- 2.6.1 进程通信的类型
- 2.6.2 消息传递通信的实现方式
- 2.6.3 直接消息传递系统实例



2.6.1 进程通信的类型

❖ 1、基本概念

进程通信是指进程之间的信息交换，根据效率高低分为两大类：

* 低级进程通信：一次仅能交换少量信息，通常基于共享数据结构进行通信。主要用于传递一些控制信息。
  - 缺点：效率低，通信对用户不透明，使用不便。
  - 例如信号量机制中的P、V操作就是两条低级通信原语。
* 高级进程通信：是指用户直接利用OS提供的通信命令（原语）高效地传送大量数据的一种通信方式。
  - 优点：效率高，通信实现细节对用户透明，使用便利。

2.6.1 进程通信的类型

❖ 2、高级进程通信的类型

* 1> 共享存储器系统
  - ~~基于共享数据结构的通信方式（属低级通信方式）~~
  - 基于共享存储区的通信方式
* 2> 管道通信系统
* 3> 消息传递系统
  - 直接通信方式
  - 间接通信方式
* 4> 客户机-服务器系统
  - 套接字、远程过程（方法）调用

2.6.1 进程通信的类型

❖ 2.1 共享存储器系统

* 1> 基于共享数据结构的通信方式（属低级通信方式）
  - 实现：诸进程共享某些数据结构实现进程间的信息交换，共享数据结构的设置、数据的传送、进程的互斥与同步等，都必须由程序员实现，OS只提供共享存储器。
  - 特点：效率低、程序员负担重，只适合传递少量数据
* 2> 基于共享存储区的通信方式
  - 实现：OS在存储器中划出一块共享存储区，诸进程通过对共享存储区中的数据读写来实现通信，数据的形式和位置甚至访问控制都由进程负责，而不是OS。
  - 特点：高效、速度快，适合传送大量数据。

2.6.1 进程通信的类型

❖2.2 管道通信系统

* 历史：首创于Unix系统。
* 管道：连接一个发送进程（写进程）和一个接收进程（读进程）之间通信的共享文件，又名pipe文件。
* 实现：进程通过管道收发大量数据。
* 特点：效率高，能有效传送大量数据。
* 管道机制必须提供三方面的协调能力：
  - (1) 互斥：管道是临界资源。
  - (2) 同步：读进程与写进程必须同步。
  - (3) 确定双方是否存在：只有双方均存在，才能通信。

2.6.1 进程通信的类型

❖2.3 消息传递系统

* 实现：程序员直接利用OS提供的一组通信命令(原语)实现大量数据的传递，进程间的数据交换以格式化的消息(message)为单位。
* 特点：效率高、具有透明性、程序员负担轻，当前应用最为广泛
  - 单机系统、多机系统、计算机网络均广泛采用
* 类型：直接通信方式和间接通信方式

2.6.1 进程通信的类型

❖2.4 客户机-服务器系统

* 1> 套接字

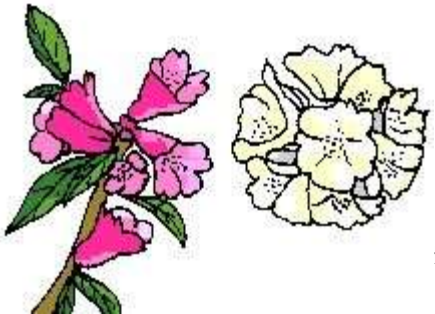
- 历史：始于1970s BSD UNIX
- 两种类型：
  - 基于文件类型：类似于Pipe
  - 基于网络类型：流套接字(TCP)、数据报套接字(UDP)

* 2> 远程过程（方法）调用

- RFC：是一个通信协议，允许运用一台主机上的进程调用另一台（远程）主机上的进程，对程序员而言，就像本地过程调用一样，具有透明性。
- 原理：通过客户存根与服务器存根关联实现。

2.6 进程通信

- 2.6.1 进程通信的类型
- 2.6.2 消息传递通信的实现方式
- 2.6.3 直接消息传递系统实例



2.6.2 消息传递通信的实现方式

❖ 1、直接通信方式（直接消息传递系统）

* 概念：直接通信是指发送进程利用OS所提供的发送命令，直接把消息发送给目标进程。

❖ 1.1 直接通信原语

* 直接通信原语（OS提供）：

要求发送进程和接收进程都以显式方式提供对方的标识符。

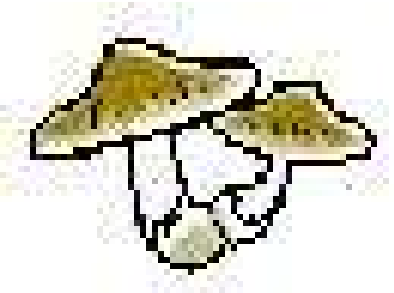
- 1> 对称寻址方式
  - Send(Receiver, message); //发送一个消息给接收进程
  - Receive(Sender, message); //接收Sender进程发来的消息
- 2> 非对称寻址方式
  - Send(P, message); //发送一个消息给接收进程P
  - Receive(id, message); //接收任何进程发来的消息,id为发送进程的参数，即完成通信后的返回值

2.6.2 消息传递通信的实现方式

* 用直接通信方式解决生产—消费问题：

```
producer(i)
    repeat
      produce an item in nextp;
      send(consumer(j)，nextp);
     until false;

consumer(j)
    repeat
       receive(producer(i)，nextc);
       consumer the item in nextc;
    until false;
```



2.6.2 消息传递通信的实现方式

❖ 1.2 消息的格式

* 消息的组成

- 消息头
  - 含传输时所需的控制信息，如：源进程名、目标进程名、消息长度、消息类型、消息编号、发送时间
- 消息正文

* 消息格式类型

- 定长消息
  - OS开销小，但用户不便（特别是传长消息用户）
- 变长消息
  - OS开销大，但用户方便

2.6.2 消息传递通信的实现方式

❖ 1.3 进程同步方式（此处实际上分析进程通信前、后的状态）

* 1> 发送进程阻塞、接收进程阻塞（汇合rendezrous）

- 直到有消息要传递时发、收进程才解除阻塞。
- 主要用于进程之间紧密同步，发送和接收进程之间无缓冲时。

* 2> 发送进程不阻塞、接收进程阻塞

- 相当于接收进程（可能是多个）一直等待发送进程，直到有消息发送来为止。
- 是应用最多的一种进程同步方式，如服务器上的打印服务。

* 3> 发送进程和接收进程均不阻塞

- 发送进程与接收进程平时都在忙自己的事情，仅当发生某事使它无法运行时才把自己阻塞起来等待，一般在发、收进程间有多个缓冲区时使用。

2.6.2 消息传递通信的实现方式

❖ 1.4 通信链路

* 根据建立方式划分
  - 显式通信链路（由发送进程建立）、隐式通信链路（由OS建立）
* 根据连接方式划分
  - 点-点通信链路、多点通信链路
* 根据通信方向划分
  - 单向通信链路、双向通信链路
* 根据链路容量划分
  - 无容量通信链路（无缓冲区）、有容量通信链路（有缓冲区）

2.6.2 消息传递通信的实现方式

❖2、间接通信方式（信箱通信方式）

* 概念：间接通信是指进程之间的通信需通过信箱（中间实体）作为媒介来进行通信，即发送进程将消息发往信箱暂存，接收进程则从信箱中取出对方发给自己的消息。

* 优点：
  - 既可实现实时通信，又可实现非实时通信

2.6.2 消息传递通信的实现方式

* 信箱的结构
  - 信箱头
    - 包含有关信箱的描述信息：信箱标识符、信箱拥有者、信箱口令、信箱的空格数等
  - 信箱体
    - 由若干个可以存放消息的信箱格组成

2.6.2 消息传递通信的实现方式

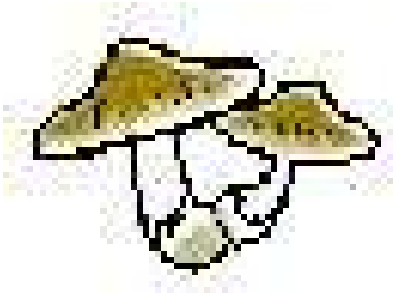
* 信箱通信原语
  - 1> 信箱的创建与撤消
    - 创建：信箱名 属性（公用、私用、共享）
    - 信箱可由OS或用户进程创建
  - 2> 消息的发送和接收
    - **Send (mailbox，message)** //将消息发往信箱
    - **Receive (mailbox，message)** //从信箱接收消息

2.6.2 消息传递通信的实现方式

* 信箱类型
  - 1> 私用信箱
    - 由用户进程自己创建，并作为进程一部分；信箱随着其创建进程的终止而消失
    - 信箱拥有者拥有对信箱的读/写权，其它用户只有写权（单向通信链路）
  - 2> 公用信箱
    - 由OS创建，在OS运行期间始终存在
    - 所有核准进程均可对信箱读/写（双向通信链路）
  - 3> 共享信箱
    - 由某用户进程创建，并指明其共享者
    - 信箱拥有者、共享者均可对信箱读/写（双向通信链路）

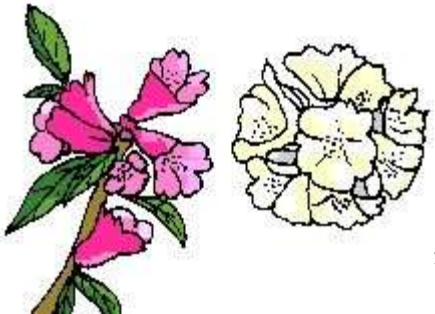
2.6.2 消息传递通信的实现方式

* 信箱通信中发送进程与接收进程的对应关系
  - 1> 一对一关系
    - 点对点通信
  - 2> 多对一关系
    - 客户机/服务器交互
  - 3> 一对多关系
    - 广播方式
  - 4> 多对多关系
    - 设置公用信箱实现



2.6 进程通信

- 2.6.1 进程通信的类型
- 2.6.2 消息传递通信的实现方式
- 2.6.3 直接消息传递系统实例
  - 消息缓冲队列通信机制



2.6.3 消息缓冲队列通信机制

❖ 1、消息缓冲队列通信机制所采用数据结构

* 1> 消息缓冲区数据结构

```
type message_buffer=record
        sender;  //发送者进程标识符
        size;    //消息长度
        text;    //消息正文
        next;    //指向下一个消息缓冲区的指针
end
```

2.6.3 消息缓冲队列通信机制

* 2> PCB中增加的通信数据项

在设置消息缓冲队列的同时，还应增加用于对消息队列进行操作和实现同步的信号量，并将它们置入进程的PCB中。在PCB中应增加的数据项可描述如下：

```
type PCB=record
        …
        mq;          //消息队列队首指针
        mutex;       //消息队列互斥信号量
        sm;          //消息队列资源信号量
        …
end
```

2.6.3 消息缓冲队列通信机制

❖ 2、消息缓冲队列通信流程

* 1> 发送进程先在自己的内存空间设置一发送区a，将待发送消息正文、发送进程标识符、消息长度等信息填入其中，然后再调用发送原语把消息发给接收进程。
* 2> 发送原语根据发送区a中所设置的消息长度来申请一个缓冲区i，然后 把发送区a中的信息复制到i中；再从PCB总链获得接收进程的PCB内部标识符j，将i挂在接收进程的消息队列j.mq上。
* 3> 接收进程调用接收原语从其消息队列mq中摘下第一个消息缓冲区i，将其中的信息复制到指定消息接收区b内。

2.6.3 消息缓冲队列通信机制

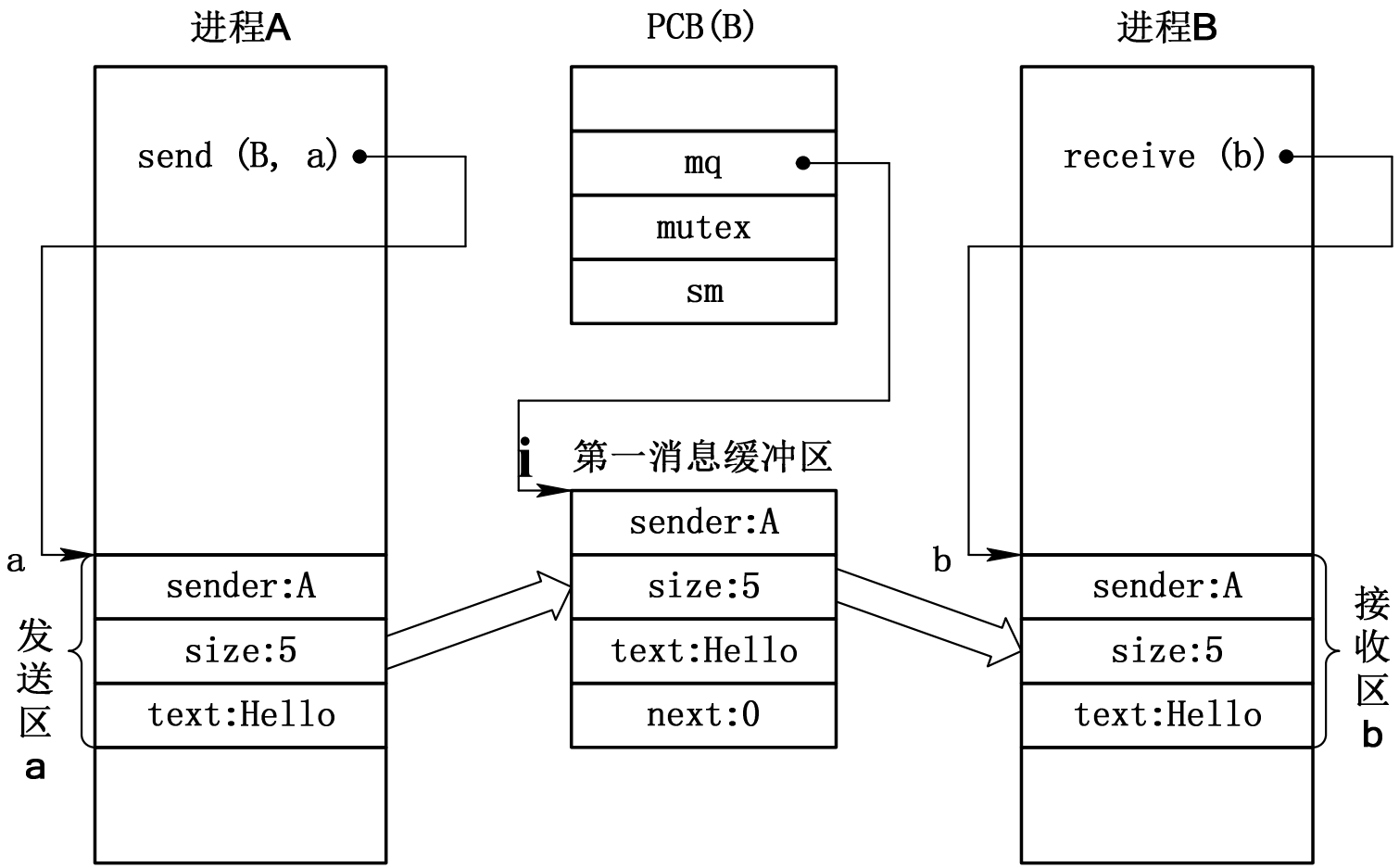


图2-14 消息缓冲通信

2.6.3 消息缓冲队列通信机制

❖3、发送原语

```
procedure send(receiver, a)
    begin
      getbuf(a.size,i);        //根据a.size申请缓冲区i
      i.sender:=a.sender;  /*
     i.size:=a.size;           将a中的信息复制到缓冲区i中
      i.text:=a.text;                                    */
      i.next:=0;
      getid(PCB set, receiver.j);  //获得接收进程内部标识符j
      P(j.mutex);
      insert(j.mq, i);    //将缓冲区插入接收进程消息队列j.mq
      V(j.mutex);
      V(j.sm); //表示已发送一个消息到j.mq中,用于实现同步
   end
```

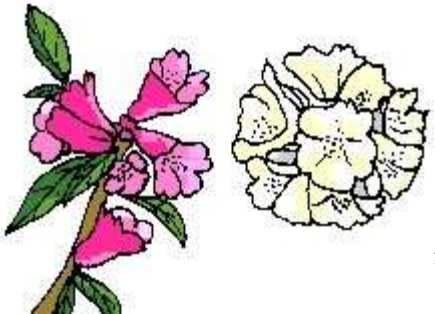
2.6.3 消息缓冲队列通信机制

❖ 4、接收原语

```
procedure receive(b)
  begin
    j:=internal name; // j为接收进程内部的标识符
    P(j.sm); //为了实现同步，表示将从j.mq消息队列移出一个消息
    P(j.mutex);
    remove(j.mq, i); //将消息队列中第一个消息移出
    V(j.mutex);
    b.sender:=i.sender;   /*
    b.size:=i.size;              将缓冲区i中的信息复制到b中
    b.text:=i.text;                                    */
  end
```

2.7 线程

- 2.7.1 线程的基本概念
- 2.7.2 线程间的同步和通信
- 2.7.3 线程的实现方式
- 2.7.4 线程的实现



2.7.1 线程的基本概念

❖ 1、线程的引入

* 引入缘由

- 传统OS中进程是一个可拥有资源的独立单位，同时又是一个可独立调度和分派的基本单位。由于进程是一个资源的拥有者，因而在创建、撤消和切换中，系统必须为之付出较大的时空开销。

* 解决办法

- 若能将进程的两个属性分开，由操作系统分开处理：作为调度和分派的基本单位时，不同时作为拥有资源的单位，以做到“轻装上阵”；而对于拥有资源的基本单位，又不对之进行频繁的切换。正是在这种思想的指导下，形成了线程的概念。

2.7.1 线程的基本概念

❖ 2、线程的属性

* 1> 轻型实体
  - 线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证其独立运行的资源（线程控制块TCB、程序计数器、寄存器和堆栈）
* 2> 独立调度和分派的基本单位
  - 在多线程OS中，线程是能独立运行的基本单位。
* 3> 可并发执行
* 4> 共享所属进程的资源
* 线程定义：线程是OS中可并发执行的轻型实体，其基本上不拥有系统资源，但共享其所属进程所拥有的全部资源，是OS独立调度和分派的基本单位。

2.7.1 线程的基本概念

❖ 3、线程控制块(TCB)

* ① 线程标识符；②寄存器状态；③ 线程运行状态；④ 优先级；⑤ 线程专有存储器；⑥ 信号屏蔽；⑦ 堆栈。

❖ 4、线程的状态

* 线程的三种基本状态
  - ① 执行状态：表示线程正获得处理机而运行
  - ② 就绪状态：指线程已具备了各种执行条件，一旦获得CPU便可执行的状态
  - ③ 阻塞状态：指线程在执行中因某事件而受阻，处于暂停执行时的状态

2.7.1 线程的基本概念

❖ 5、线程的创建和终止

* 线程的创建

- 在多线程OS环境下，应用程序启动时通常仅有一个线程在执行，该线程被称为“初始化线程”或“主线程”，它可根据需要再去创建若干个线程。

* 线程的终止

- 线程的终止方式有两种：自愿退出、异常终止。
- 有些线程(主要是系统线程) 一旦被建立便不再被终止。
- 在大多数的OS中，线程被中止后并不立即释放它所占有的资源，只有当进程中的其它线程执行了分离函数后，被终止的线程才与资源分离，此时的资源才能被其它线程利用。
- 虽被终止但尚未释放资源的线程，仍可以被需要它的线程调用而重新恢复运行。

2.7.1 线程的基本概念

❖ 6、多线程OS的进程属性

* 1> 作为系统资源分配的单位
* 2> 多个线程可并行执行
  - 一个进程可含有多个相对独立的线程，其数目可多可少，但至少也要有一个线程
* 3> 进程不再是一个可执行的实体
  - 虽然如此，进程仍具有与执行相关的状态。例如，所谓进程处于“执行”状态，实际上是指该进程中的某线程正在执行。此外，OS对进程所施加的与进程状态有关的操作，也对其所包含的线程起作用。

2.7.1 线程的基本概念

重要！

❖ 7、线程与进程的比较

* 1> 调度性

- 在传统OS中，作为拥有资源的基本单位和独立调度、分派的基本单位都是进程。而在引入线程的操作系统中，则把线程作为调度和分派的基本单位，而进程作为资源拥有的基本单位。

* 2> 并发性

- 在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，因此多线程OS具有更好的并发性。

2.7.1 线程的基本概念

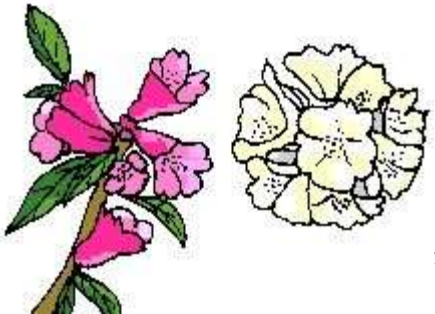
* 3> 拥有资源
  - 进程不论是在传统OS还是在多线程OS中都是一个拥有资源的基本单位；线程基本上自己不拥有系统资源(只有一点必不可少的资源)，但它可以访问其所属进程的资源。
* 4> 独立性
  - 不同进程之间具有较高的独立性，除了共享的全局变量之外，每个进程内的地址空间和其它资源不允许别的进程访问。
  - 同一进程内的线程独立性要比不同进程之间的独立性低很多，可共享所属进程的资源，一个线程的堆栈也可被其它线程读、写。

2.7.1 线程的基本概念

* 5> 系统开销
  - 在创建或撤消进程时，系统都要为之创建和回收进程控制块，分配或回收资源，OS所付出的开销明显大于线程创建或撤消时的开销。
  - 由于一个进程中的多个线程具有相同的地址空间，在同步和通信的实现方面线程也比进程容易。在一些操作系统中，用户级线程的切换、同步和通信都无须操作系统内核的干预。
* 6> 支持多处理机系统
  - 传统进程只能运行在一个处理机上。
  - 多线程进程中的多个线程可分配到多个处理机上并发运行。

2.7 线程

- 2.7.1 线程的基本概念
- 2.7.2 线程间的同步和通信
- 2.7.3 线程的实现方式
- 2.7.4 线程的实现



2.7.2 线程的同步和通信

❖ 1、互斥锁(mutex)

* 互斥锁是一种用于实现线程间对临界资源互斥访问的机制，操作简单、开销低，较适合于高频度使用的关键共享数据和程序段。

* 阻塞方式：

```
lock mutex;//上锁
…
unlock mutex; // 开锁
```

非阻塞方式:

```
if ( Trylock mutex )
….
else
….
```

2.7.2 线程的同步和通信

❖ 互斥锁使用可能死锁示例

Thread 1

lock mutex1

临界区C;

lock mutex2

临界资源R;

unlock mutex2

unlock mutex1

Thread 2

lock mutex2

临界资源R;

lock mutex1

进入临界区C;

unlock mutex1

unlock mutex2

2.7.2 线程的同步和通信

❖ 2、条件变量(condition)

* 条件变量通常与互斥锁一起配合使用，以解决只利用mutex实现互斥访问时可能引起的死锁问题。

Thread A:

```
Lock mutex
    while(resource 不足)
        wait(condition variable)
    使用resource并修改R
 unlock mutex
```

Thread B:

```
已生产一定数量的resource
Lock mutex
    修改R
 unlock mutex
wakeup(condition variable)
```

wait将当前线程A阻塞在条件变量里，同时将mutex解锁；wakeup在将条件变量里阻塞的线程A唤醒，还要将线程A被阻塞前的锁mutex给加上

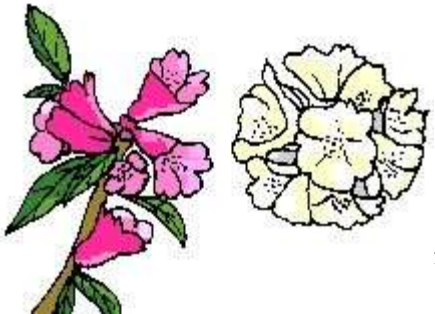
2.7.2 线程的同步和通信

❖ 3、信号量机制

* 1> 私用信号量（private semaphore)
  - 私用信号量作用域仅在一个进程当中，常用于一进程内各线程之间的同步
  - OS不知道其存在
  - 使用中如出了问题，OS无法将其复位
* 2> 公用信号量（public semaphore)
  - 公用信号量作用域在多个进程之间
  - 由OS为其分配空间并进行管理，又称系统信号量
  - OS可自动回收信号量空间，比较安全

2.7 线程

- 2.7.1 线程的基本概念
- 2.7.2 线程间的同步和通信
- 2.7.3 线程的实现方式（线程的类型）
- 2.7.4 线程的实现



2.7.3 线程的实现方式

❖ 1、内核级线程

* 内核级线程（内核支持线程 KST, Kernel Supported Threads）：是在内核的支持下运行的，它们的创建、撤消和切换等也是在内核空间实现的。
* 内核级线程TCB：在内核空间还为每一个内核级线程设置了一个线程控制块（TCB），内核根据该控制块而感知某线程的存在，并对其加以控制。
* 内核级线程优点
  - (1) 在多处理器系统中，内核能够同时调度同一进程中多个线程并行执行。

2.7.3 线程的实现方式

- (2) 如果某进程中的一个内核级线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程。
- (3) 内核级线程具有很小的数据结构和堆栈，线程的切换比较快，~~切换开销小~~。
- (4) 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率。

* 内核级线程缺点

- 对于用户进程的内核级线程切换而言，其模式切换开销较大（原因：因为用户的线程一般运行在用户态，其调度和管理由内核负责，需从用户态到内核态切换）

2.7.3 线程的实现方式

❖ 2、用户级线程

* 用户级线程(ULT, User Level Threads)：仅存在于用户空间中，其创建、撤消、切换、线程之间的同步与通信等功能都无须内核的支持。
* 用户级线程TCB：设置在用户空间，内核完全不知道用户级线程的存在，也不参与线程的调度。
* 用户级线程优点
  - (1) 线程切换不需要转换到内核空间，从而节省了模式切换的开销，也节省了内核的宝贵资源。
  - (2) 线程调度由所属进程完成。进程可以根据自身需要，选择相应调度算法对自己的线程进行管理和调度，与OS无关。

2.7.3 线程的实现方式

- (3) 用户级线程的实现与操作系统平台无关，其运行、管理、调度均在用户态。因此，用户级线程甚至可以在不支持线程机制的操作系统平台上实现。

* 用户级线程缺点

- (1) 存在系统调用阻塞问题。当用户级线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程都会被阻塞。（为什么？）
- (2) 在单纯的用户级线程实现方式中，多线程应用不能利用多处理机进行多重处理的优点。内核每次分配给一个进程仅有一个CPU，因此进程中仅有一个线程能执行，在该线程放弃CPU之前，其它线程只能等待。
- (3) 用户级线程以进程为单位平均分配CPU时间片，对线程间并发执行并不有利，而内核级线程以线程为单位平均分配CPU时间片。

2.7.3 线程的实现方式

❖ 3、组合式线程（ULT/KST 线程）

* 组合式线程是将用户级和内核级线程两种方式进行组合。
* 在组合式线程系统中：内核支持内核级线程的建立、调度和管理，用户应用程序建立、调度和管理用户级线程。一些内核级线程对应多个用户级线程，程序员可按应用需要和机器配置对内核支持线程数目进行调整，以达到较好的效果。
* 优点：组合式线程机制能够结合KST和ULT两者的优点，并克服了其各自的不足，即组合方式线程中，同一个进程内的多个线程可以同时在多处理器上并行执行，而且在阻塞一个线程时，并不一定需要将整个进程阻塞。

2.7.3 线程的实现方式

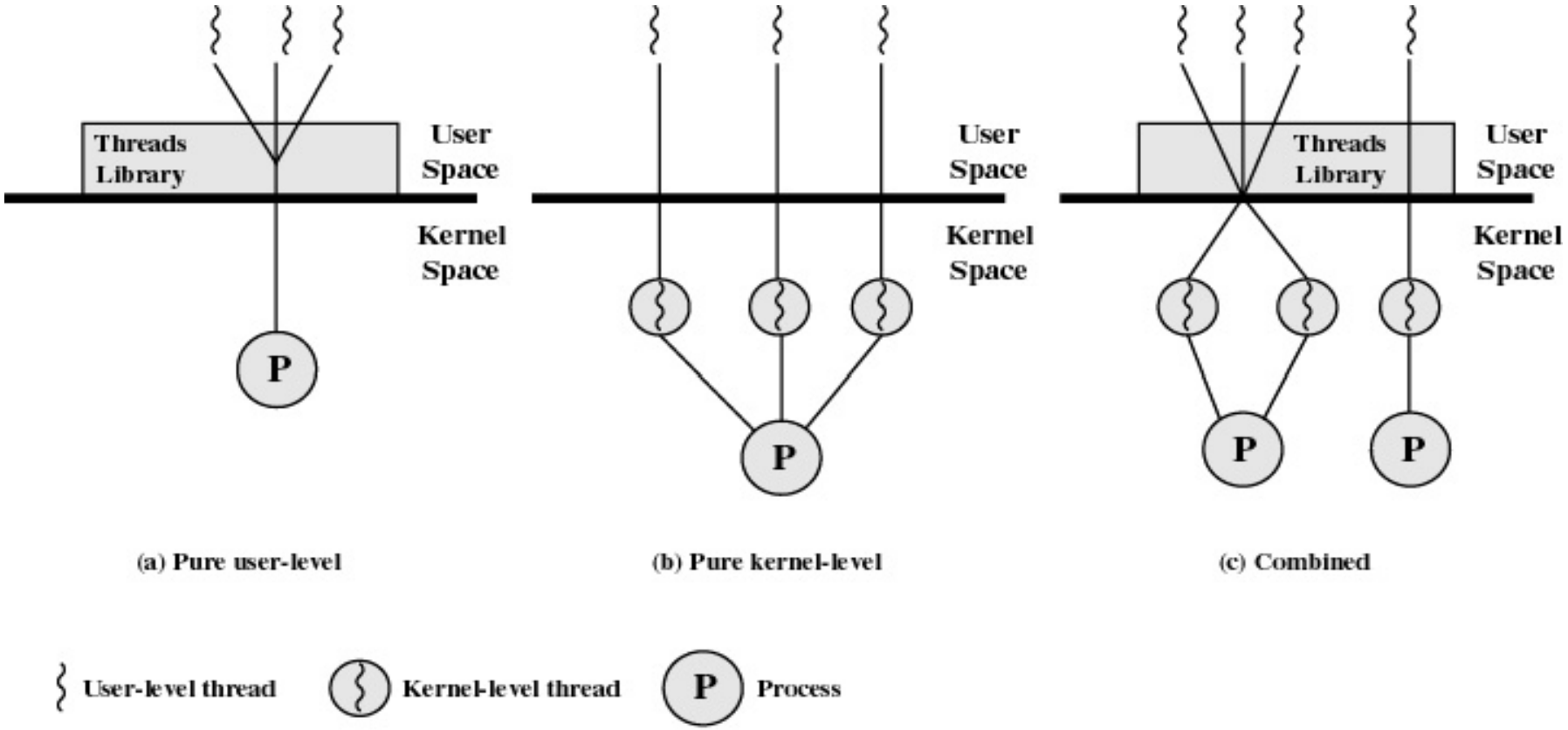
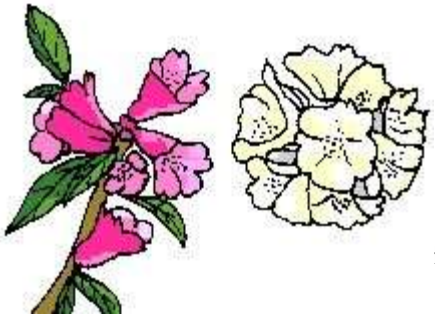


图　线程实现的三种方式

2.7 线程

- 2.7.1 线程的基本概念
- 2.7.2 线程间的同步和通信
- 2.7.3 线程的实现方式
- 2.7.4 线程的实现



2.7.4 线程的实现

❖ 1、内核级线程的实现

* 1> 系统在创建一个新进程时，便为它在内核中分配一个任务数据区PTDA，其中包括若干个空白线程控制块TCB空间（用来保存线程标识符、优先级、线程运行的CPU状态等信息）。
* 2> 每当进程要创建一个线程时，内核便为新线程分配一个TCB，将有关信息填入该TCB中，并为之分配资源。
* 3> 内核支持线程的调度和切换与进程的调度和切换十分相似，也分抢占式方式和非抢占方式两种。在线程的调度算法上，同样可采用时间片轮转法、优先权算法等。

2.7.4 线程的实现

❖ 2、用户级线程及组合式线程的实现

用户级线程在用户空间实现，所有的用户级线程具有相同的结构，都运行在一个中间系统的上面。用户级线程在切换和进行系统资源调用均由中间系统来完成，从而无需进入内核态。当前有两种方式实现的中间系统：

* 1> 运行时系统(Runtime System)

- 运行时系统实质上是用于管理和控制线程的函数(过程)的集合，其中包括用于创建和撤消线程的函数、 线程同步和通信的函数以及实现线程调度的函数等。
- 运行时系统中的所有函数都驻留在用户空间，并作为用户级线程与内核之间的接口。

2.7.4 线程的实现

* 2> 内核控制线程

- 内核控制线程又称为轻型进程LWP(Light Weight Process)。每一个进程都可拥有多个LWP，同用户级线程一样，每个LWP都有自己的数据结构(如TCB)，其中包括线程标识符、优先级、状态，另外还有栈和局部存储区等。
- LWP可通过系统调用来获得内核提供的服务，也可以共享进程所拥有的资源，这样，当一个用户级线程运行时，只要将它连接到一个LWP上，此时它便具有了内核级线程的所有属性。
- 这种实现方式实际上就是组合式线程的实现方式。

2.7.4 线程的实现

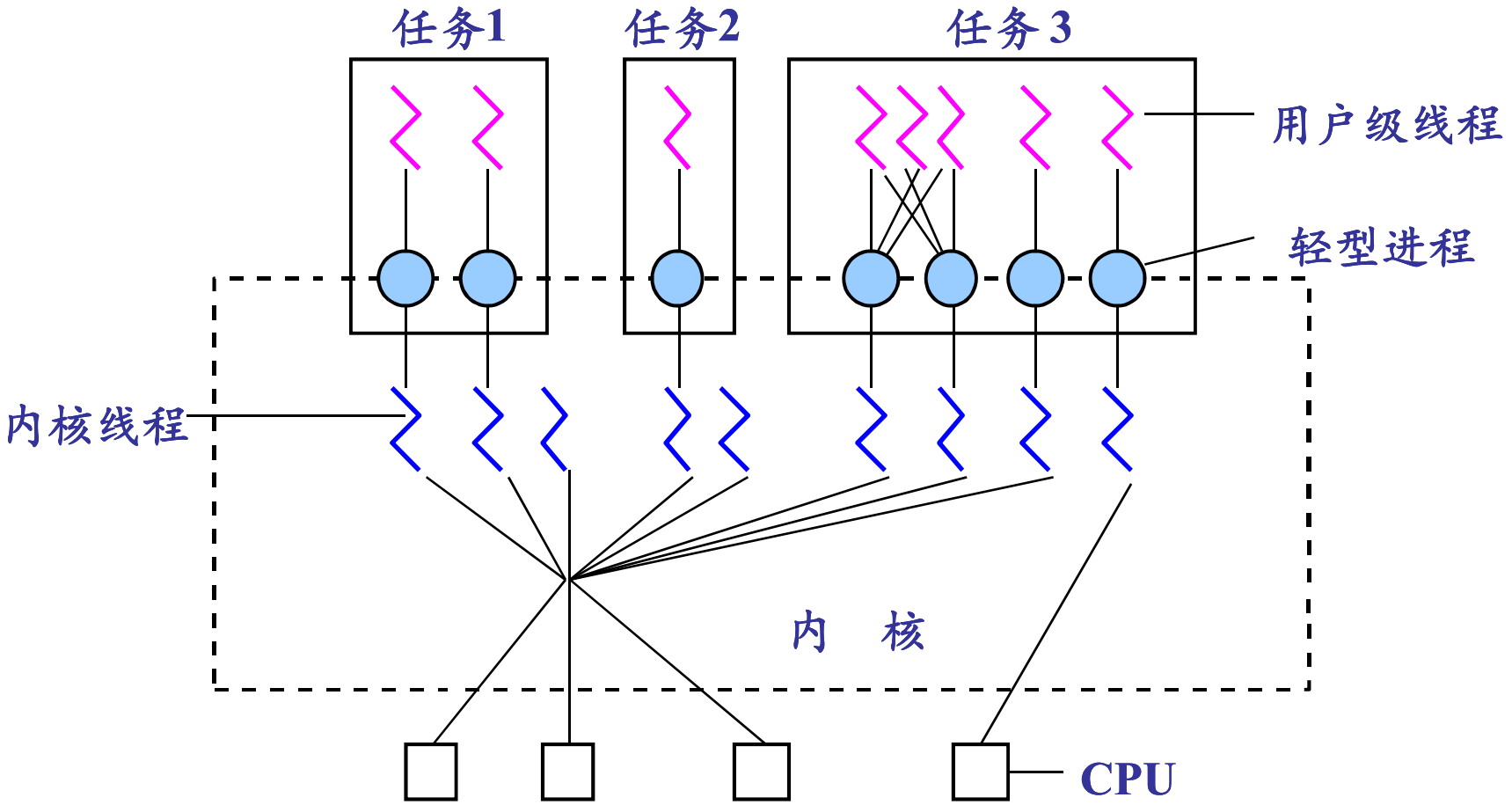


图 2-16　利用轻型进程作为中间系统

2.7.4 线程的实现

- 用户级线程与LWP的连接模型
  - 1> 一对一模型
    - » 该模型并行能力较强，但每创建一个用户线程相应地就需要创建一个内核线程，开销较大，因此需要限制整个系统的线程数。Windows 2000、Windows NT、OS/2等系统上都实现了该模型。
  - 2> 多对一模型
    - » 用户级线程的调度和管理在用户空间中完成，当用户线程需要访问内核时，才将其映射到一个内核控制线程上，但每次只允许一个线程进行映射。
    - » 该模型的主要优点是线程管理的开销小，效率高，但当一个线程在访问内核时发生阻塞，则整个进程都会被阻塞，而且在多处理机系统中，一个进程中的多个线程无法实现并行。

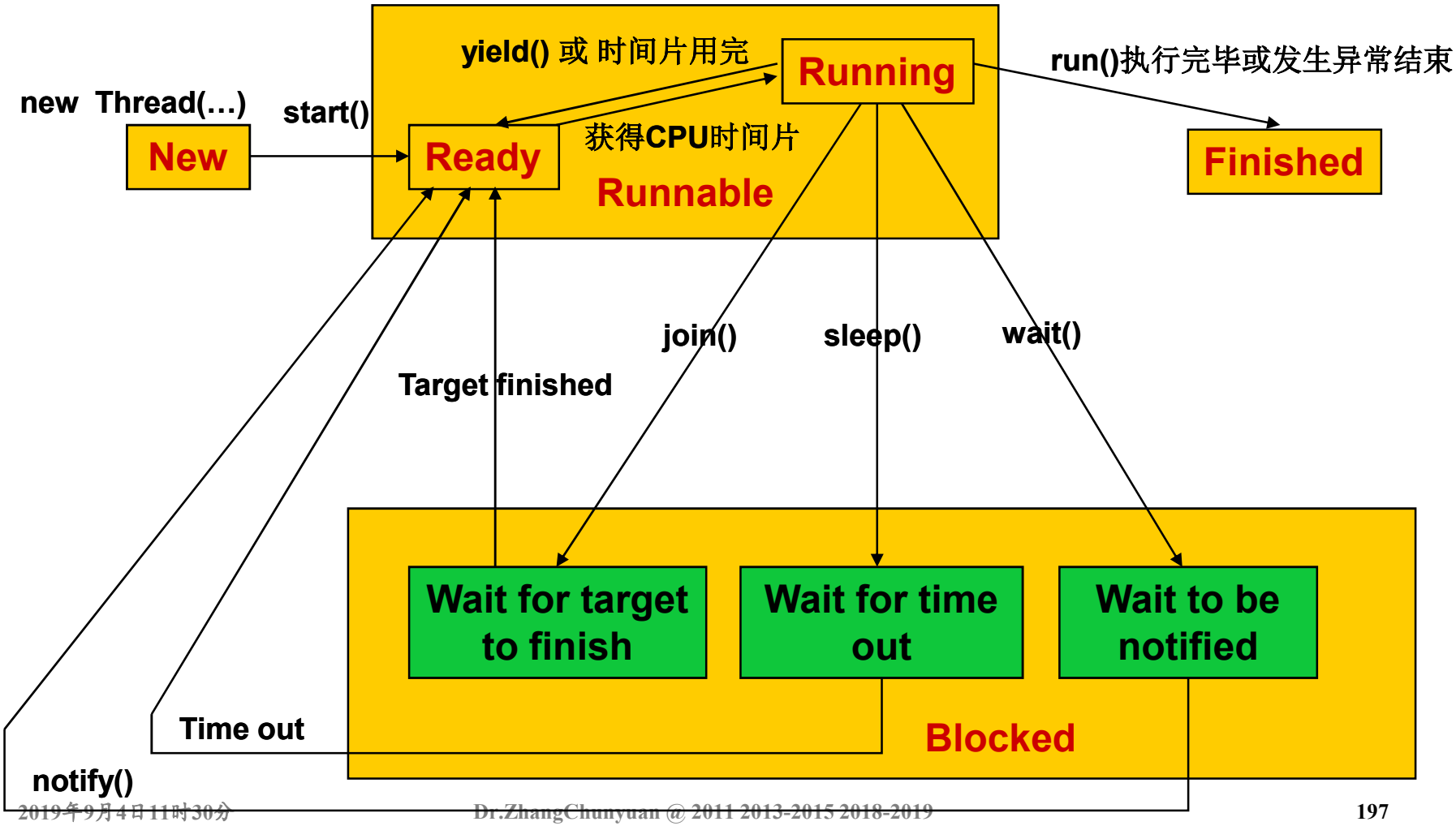
2.7.4 线程的实现

– 3> 多对多模型

» 该模型结合上述两种模型的优点，将多个用户线程映射到多个内核控制线程，内核控制线程的数目可以根据应用进程和系统的不同而变化，可以比用户线程少，也可以与之相同。

附：Java多线程状态及应用示例

❖ Java多线程状态及切换



2.2.2 进程的基本状态及转换

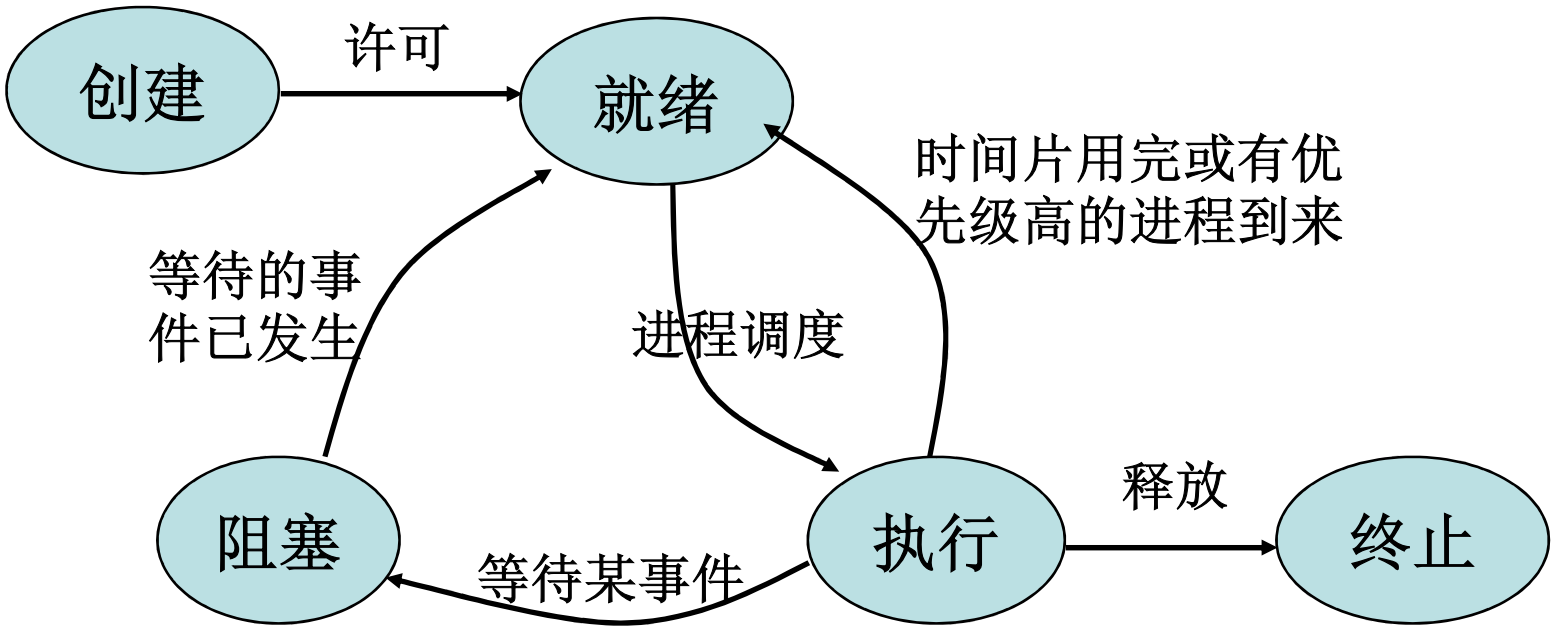


图2－6 进程的五种状态及其转换

附：Java多线程状态及应用示例

❖ Java多线程应用示例

* 程序说明

- 创建六个线程模拟六个人到银行的三个窗口同时从同一个银行帐户上存钱或取钱，六个线程的管理与调度由线程池来完成，线程池的数量代表开放的窗口数量。银行帐户对六个线程来说是临界资源，需互斥访问，本程序采用互斥锁+条件变量配合实现。

* 程序代码（参见代码SysncThread.java）

- 测试类SyncThread
- 取钱线程类DrawThread
- 存钱线程类SaveThread
- 银行帐户类MyCount

附：Java多线程状态及应用示例

* 运行结果
  - SysncThread.java每次输出结果可能都不一样，因为不同线程并发运行时存在异步性，但银行帐户的余额经过多次存取款后仍是对的，并没有因为多个线程并发操作而导致混乱。
  - 另在文件ASysncThread.java中我将互斥锁+条件变量注释掉了，运行该程序时发现银行帐户的余额经过多次存取款后有时是错的，即线程并发运行失去了再现性，原因是我们没有对临界资源进行互斥控制。

本章小结

- 程序顺序执行三大特征
- 程序并发执行三大特征**
- 进程的五大特征**
- 进程的七种状态及切换（三种基本状态**）
- 进程六种控制原语
- 进程的两种制约关系**
- 进程与程序的区别**
- 进程同步机制应遵循的四大规则*
- 四种信号量机制**
- 前趋图及信号量实现前趋关系*
- 管程机制与条件变量*

本章小结

- 进程和管程的区别**
- 进程同步三大经典问题**
- 进程通信的四大类型
- 消息传递通信的两种实现方式
- 进程和线程的区别**
- 线程的四大特性
- 线程的三种类型及实现方式
- 重要概念
  - 进程、线程 、管程、原语、进程实体、临界区、PCB、轻型进程、互斥、同步、条件变量、信号量、前趋图、进程图

本章作业

- 要求：
  - 一定要做在作业本上，不接受大本子，也不接受信笺、草稿纸
- 交作业日期：
  - 第六周周一课上交
- 作业内容：
  - 第2章网络测试题
  - 补充题共9道

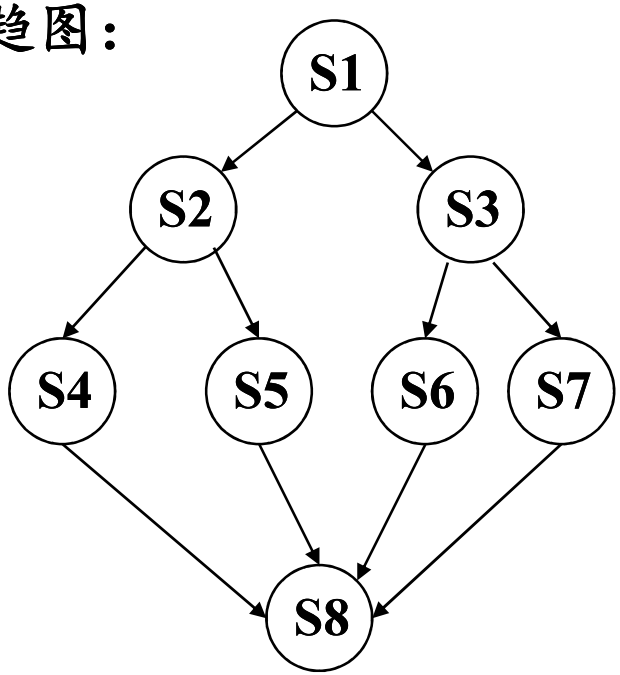
本章作业

❖ 补充作业：

1、试画出下面四条语句的前趋图：

S1：a = x+y；S2：b = z+1；S3：c = a+b；S4：w = c+1；

2、试写出相应的程序描述以下前趋图：



本章作业

3、在生产者和消费者问题中，如果缺少了signal(full)或signal(empty)，对执行结果将有何影响？

4、在生产者和消费者问题中，如果将两个wait操作即wait(full)和wait(mutex)互换位置，或将signal(mutex)和signal(full)互换位置，结果会如何？要求写出分析过程

5、我们为某种临界资源设置一把锁w, 当w=1时表示关锁，当w=0时表示开锁，试写出开锁和前锁原语，并利用它们去实现互斥。

6、试利用记录型信号量写出一个不会出现死锁的哲学家进餐问题的算法。

本章作业

7、试修改下面生产者-消费者问题解法中的错误：

```
producer
 begin
     repeat
         produce an item in nextp;
         wait(mutex);
         wait(full);
         buffer(in) = nextp;
         signal(mutex);
     until false;
 end
```

```
consumer
 begin
    repeat
         wait(mutex);
         wait(empty);
         nextc =buffer(out);
         out = out+1 ;
         signal(mutex);
         consume item in nextc;
    until false;
end
```

本章作业

8、在测量控制系统中：数据采集子系统将所采集的数据送往一单缓冲区；计算子系统则从该单缓冲区中取出数据进行计算。试写出利用信号量机制实现两子系统共享单缓冲区的同步算法。

9、分析说明 程序、进程、管程、线程四者的区别与联系。

本章课程结束！谢谢大家！