操作系统

Operating System

Gengdan Institute of Beijing University of Technology

授课教师：段雪丽

信息工程学院 计算机科学与技术教研室

第二章 进程的描述与控制

2.1 前趋图和程序的执行
2.2 进程的描述
2.3 进程控制
2.4 进程通信
2.5 进程同步基本概念
2.6 经典的同步问题
2.7 线程及其实现

本节课 主要内容

重点：

- 进程的并发执行
- 进程特征
- 进程状态转换
- 进程控制块

难点：

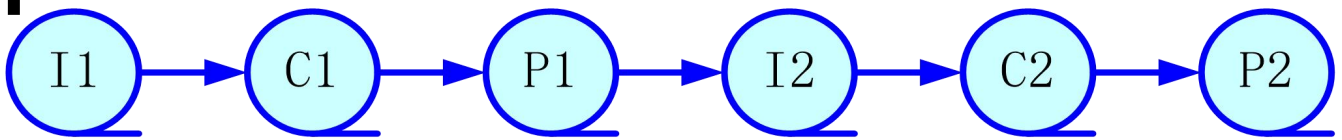
- 进程管理中的数据结构

2.1.1 程序的顺序执行及其特征

- **程序的顺序执行:一个具有独立功能的程序,独占处理机直至得到最终结果的过程,称为程序的顺序执行。**（一个较大的程序通常都由若干个程序段组成。程序在执行时，必须按照某种先后次序逐个执行，仅当前一操作执行完后，才能执行后继操作。）

1) 程序段按固定的流程执行下去

例1：输入I，计算C，打印P



2) 语句的顺序执行

例2：

```
S1：a=x+y
S2：b=a-5
S3：c=b+1
```

2.1.1 程序的顺序执行及其特征

- 程序顺序执行时的特征

(1) 顺序性　处理机的操作严格按照程序所规定的顺序执行。

(2) 封闭性　程序一旦开始执行，其计算结果不受外界因素的影响。即程序运行时独占全机资源，资源的状态（除初始）只有本程序才能改变它。

(3) 可再现性　程序执行的结果与它的执行速度无关(即与时间无关)，而只与初始条件有关。

2.1.2 前趋图

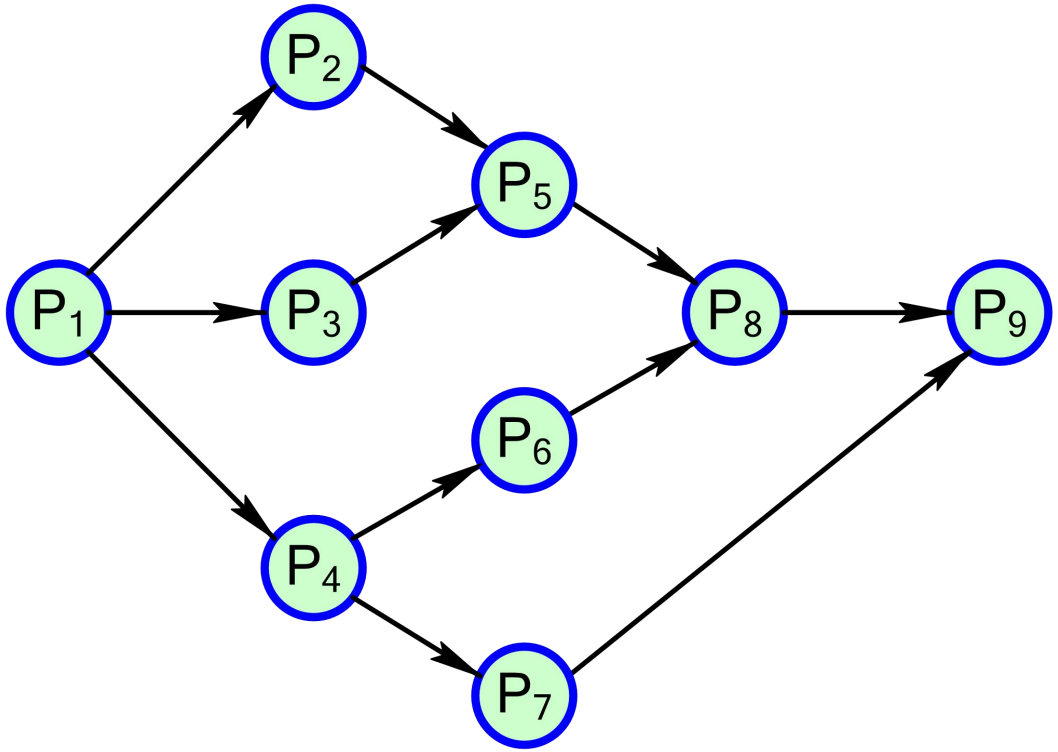
- 引入前趋图描述程序执行的先后顺序：为了描述一个程序的各部分（程序段、语句）间的依赖关系，或是一个大的计算的各子任务间的因果关系，采用前驱图方式。

2.1.2 前趋图

- 有向无循环图
- 节点用于表示一个进程或一段程序
- 节点之间用一个有方向的线段相连
- 方向表示所连接的节点之间的前趋和后继关系
- 节点间的有向边则表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)“→”
- 被指向的节点为后继节点，离开箭头的节点是前趋节点。

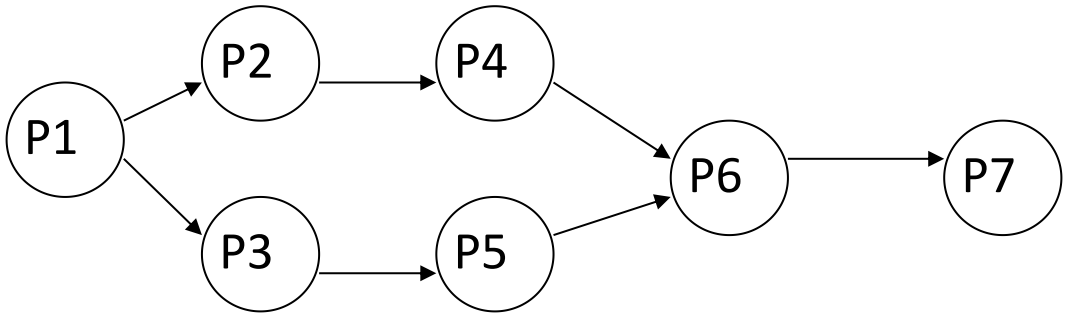
(Pi,Pj)∈→ 可写成Pi→Pj,表示Pi必须在Pj前完成

2.1.2 前趋图



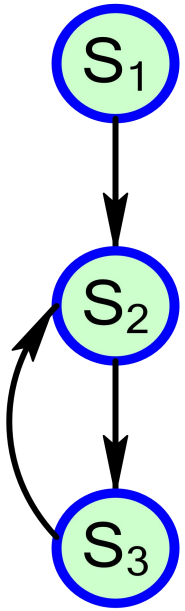
- G={P, → }
- P={P1, P2, P3, P4, P5, P6, P7, P8, P9}
- →={ (P1, P2), (P1, P3), (P1, P4), (P2, P5), (P3, P5), (P4, P6), (P4, P7), (P5, P8), (P6, P8), (P7, P9), (P8, P9)}

2.1.2 前趋图



请根据前趋图写出偏序关系节点的集合。

2.1.2 前趋图

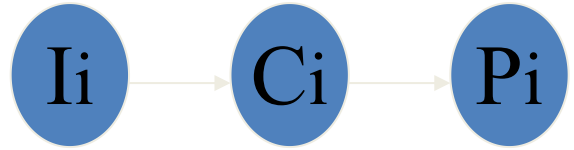


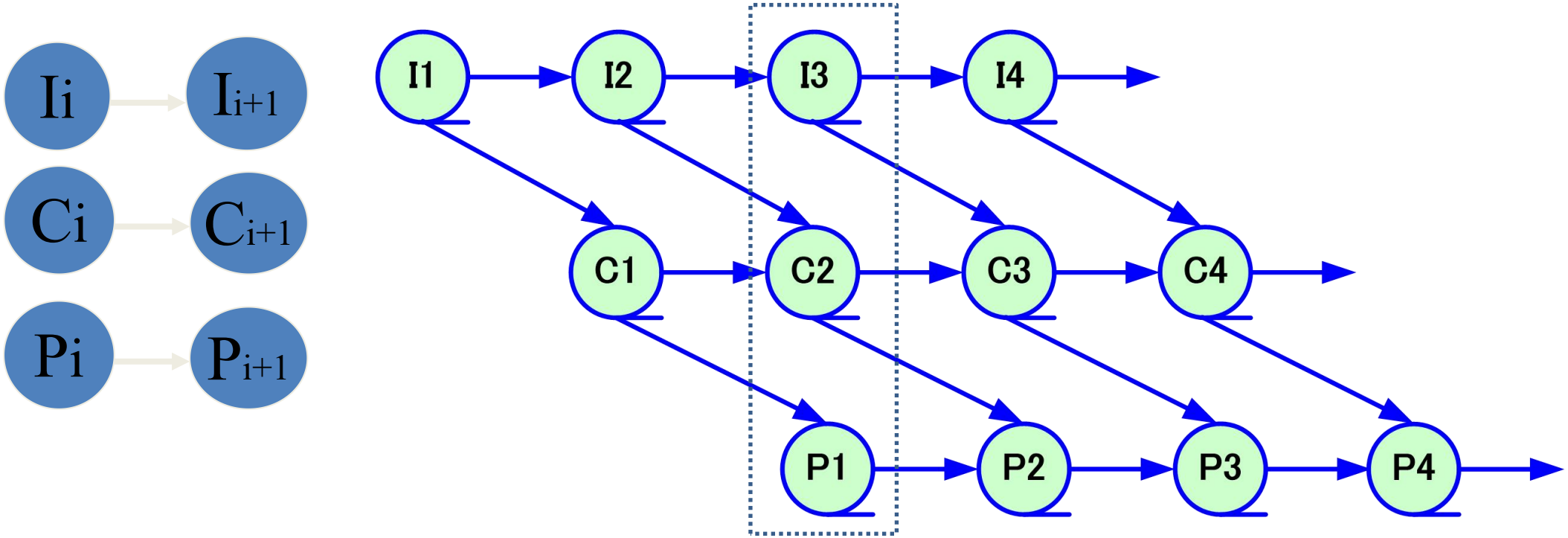
具有循环的前趋图

前趋图中是不允许有循环的，否则必然会产生不可能实现的前趋关系。
S2→S3，S3→S2

2.1.3 程序的并发执行及其特征

1）程序并发执行





前趋关系Ii→Ci，Ii→Ii+1，Ci→Pi，Ci→Ci+1，Pi→Pi+1，而Ii+1和Ci及Pi-1是重叠的，即在Pi-1和Ci以及Ii+1之间，不存在前趋关系，可以并发执行。

2.1.3 程序的并发执行及其特征

2）程序并发执行时的特征

(1) 间断性（制约性）

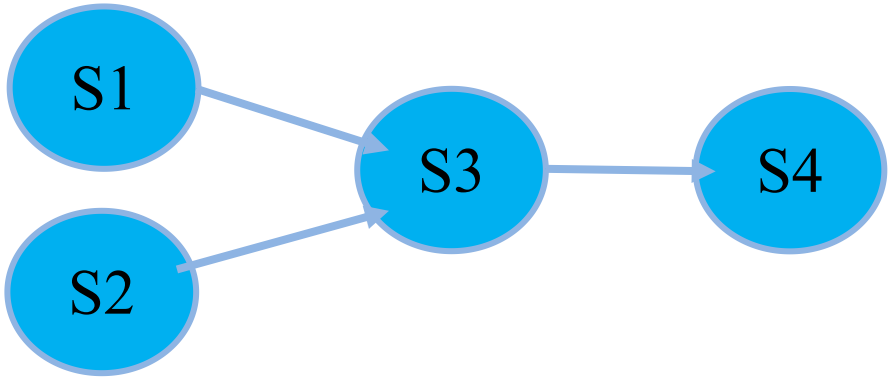
(2) 失去封闭性

(3) 不可再现性

2.1.3 程序的并发执行及其特征

2) 语句的并发

S1: A=X+2
S2: B=Y+1
S3: C=A+B
S4: D=C-6



请绘制前趋图；

这是顺序执行还是并发执行？

2.1.3 程序的并发执行及其特征

对于下述五条语句的程序段：

S_1： a:=x+2

S_2： b:=y+4

S_3： c:=a+b

S_4： d:=c+b

S5： e:=d+3

请绘制前趋图；

这是顺序执行还是并发执行？

顺序执行与并发执行的特征

完成前一操作才能进入下一操作

初始条件相同，结果唯一
(执行次数和执行速度与结果无关)

程序顺序执行的特征	1)顺序性	2)封闭性	3)可再现性

资源的状态只能由执行程序改变(运行时程序独占资源)

执行　暂停 ⇒ 原因：相互制约（直接/间接）

程序并发执行时的特征	1)间断性	2)失去封闭性	3)不可再现性

资源被共享，状态由多个共享者改变

程序并发执行带来的问题:

不可再现性

程序A

N=N+1

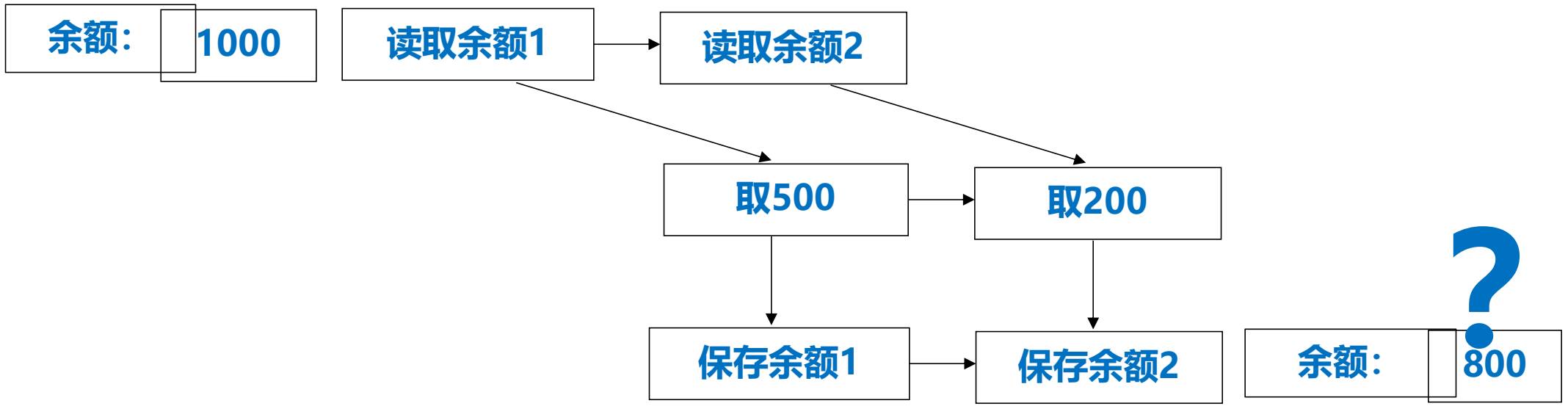
程序B

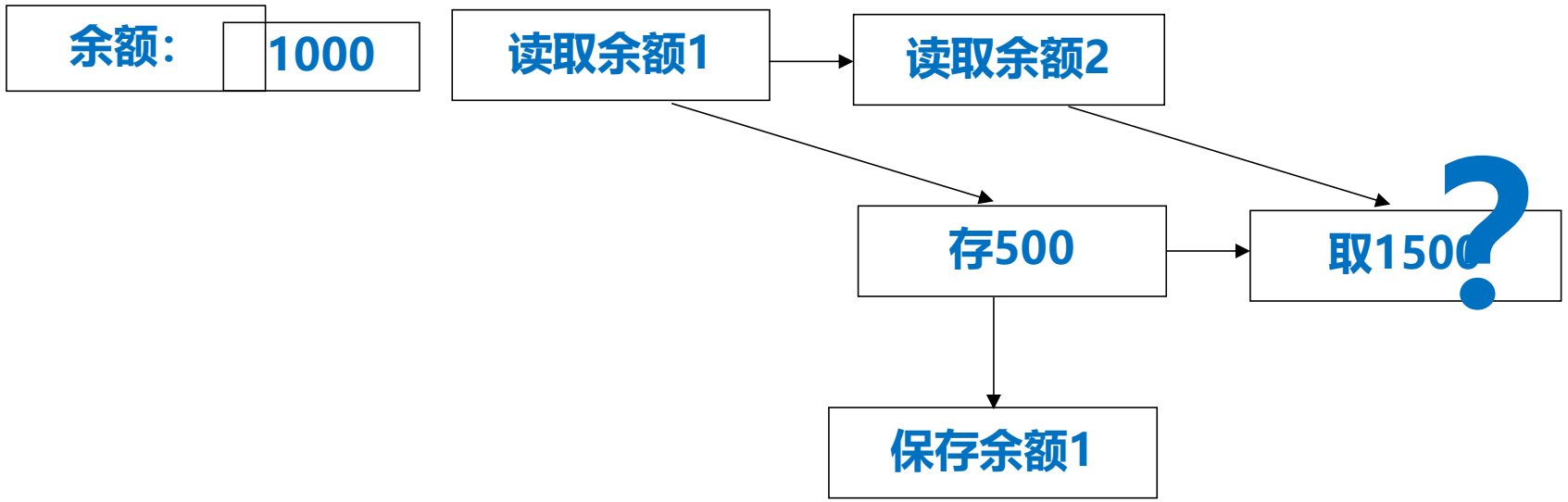
PRINT(N)

N=0

	情况A	情况B	情况C
	N=N+1	PRINT(N)	PRINT(N)
	PRINT(N)	N=0	N=N+1
	N=0	N=N+1	N=0
N值为:	0	1	0
打印结果:	N + 1	N	N

程序并发执行带来的问题:





程序并发执行带来的问题:

程序并发执行，虽然能有效地提高资源利用率和系统的吞吐量，但必须采取某种措施，以便并发程序能保持其“可再现性”。

程序并发性的分析		
1)如何使程序并发(行)	2)并发中出现的问题	3)如何处理

任务能否并发，不但取决于算法，还与程序的结构形式有很大关系，并行的方法:

- **在标准语言的基础上加以扩充**
- **重新设计专门的并行语言**

2.2.1 进程的定义和特征

? 为什么要引入进程?

为了使内存中的多道程序能够正确地并发执行，

? 为什么程序不能并发执行?

因为并发执行的程序是“停停走走”地执行，会失去封闭进而结果不可再现。因此，人们引入“进程”（Process）这一概念来描述程序动态执行过程的性质并加以记录，实现可再现性，即进程就是操作系统为进行处理机管理而引入的概念。

为了使参与并发执行的每个程序（含数据）都能独立地运行，在操作系统中必须为之配置一个专门的数据结构，称为进程控制块PCB，当程序停下时，能将其现场信息保存在它的PCB中，待下次被调用执行时，再从PCB中恢复CPU现场而继续执行。进而PCB、程序段和数据是进程的核心内容。

2.2.1 进程的定义和特征

进程的定义

异步性 ← 进程都是并发的

进程的定义：进程是可并发执行的、具有独立功能的程序在一定数据集合上的一次执行过程，是操作系统进行资源分配和调度的基本单位。

组成：程序+数据

独立性：资源的拥有者和CPU调度的分配者

一次性+动态性

创建/撤消

进程是一个程序及数据在处理机上顺序执行时所发生的活动（执行的过程）

书中的定义

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

进程的理解：

- 进程是程序运行过程
- 进程是以异步为

进程的异步特
程中的“走走停停”

操作系统为了管理
程都有一个数据结构
详细信息，该数据结
态过程不断更新。

操作系统通过进程的数据结构了解
进程
实现

相同的程序可以多次运行，每次运行的环境可能不相同。这种多

虽然进程是程序运行，但是进程与程序却不能完全等同。程序是静态的，是以文件形式存放在磁盘上的代码序列。

进程是动态的，是不断向前推进的过程，进程具有各种状态并可以在状态之间转换。

源和软件资源，特别是处理器资源。

2.2.1 进程的定义和特征

进程的特征：

- 结构性
- 动态性
- 并发性
- 独立性
- 异步性

进程包含有描述进程信息的数据结构和运行在进程上的程序。操作系统用进程控制块描述和记录进程的动态变化过程。进程的

进程是程序在数据集合上的一次执行过程，具有生命周期，由创建而产生，由调度

代

而这

过程

在同一段时间内，若干个进程可以共享一个处理器。进程的并发性能够改进系统的

在操作系统管理上，进程是一个独立的资源分配单位，进程可以在创建时获取资源，

也可

进程

间等

在计算机环境中，处理器的数量总是小于进程的数量，多个进程被强制分享同一个处理器，进程以交替方式被处理器执行。进程的这种执行方式为异步性。

小结

程序顺序执行时的特征

- 1)顺序性：前一操作完成才能进入下一操作
- 2)封闭性：运行时独占资源,结果不受外界影响；即资源的状态只能由本程序改变
- 3)可再现性：初始条件相同，结果唯一(结果与程序的执行次数和速度无关（停—走—停—走）

程序并发执行时的特征

- 1)间断性：出现执行——暂停——执行这种间断性的活动规律
原因：相互制约（直接 / 间接）
- 2)失去封闭性：资源被共享，状态由多个共享者改变
- 3)不可再现性：初始条件相同，结果不唯一
(结果与执行程序的速度相关)

小结

进程的定义：进程是可并发执行的、具有独立功能的程序在一定数据集合上的一次执行过程，是操作系统进行资源分配和调度的基本单位。

进程是程序的执行过程,创建而产生、调度而执行、撤消而消亡

按各自独立的、不可预知的速度向前推进，导致程序执行的不可再现性。

1)动态性	2)并发性	3)独立性	4)异步性	5)结构特征

进程可以并发执行,而程序却不能

进程实体由程序段、数据段和进程控制块组成,又称为“进程映像”。

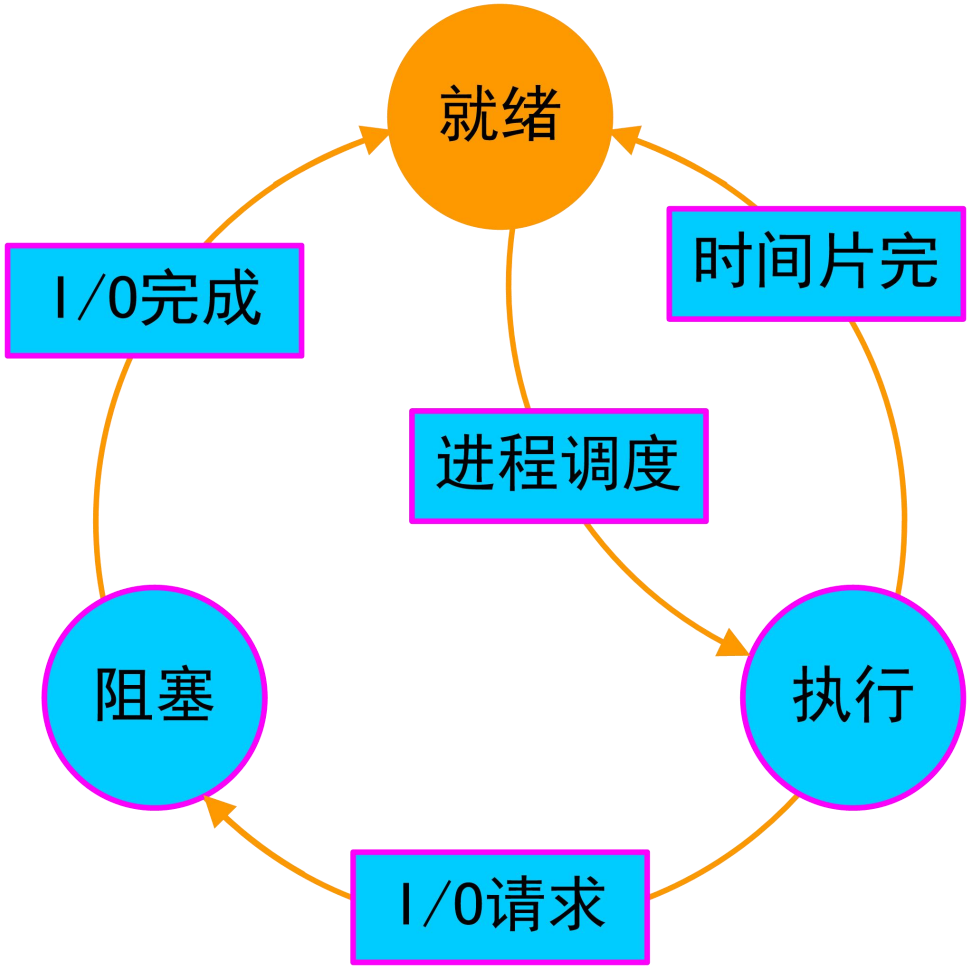
独立运行 / 资源分配 / 调度的基本单位

小结

进程与程序的区别	
动态	静态（定义）
暂时性（具有生命期）	永久性
并发性、独立性、结构特征	程序都没有
程序+数据+PCB块	程序+数据
一个进程可以涉及多个程序	一个程序可以对应多个进程
异步运行，会相互制约	程序不具备此特征

（不同的进程可以包含相同的程序）

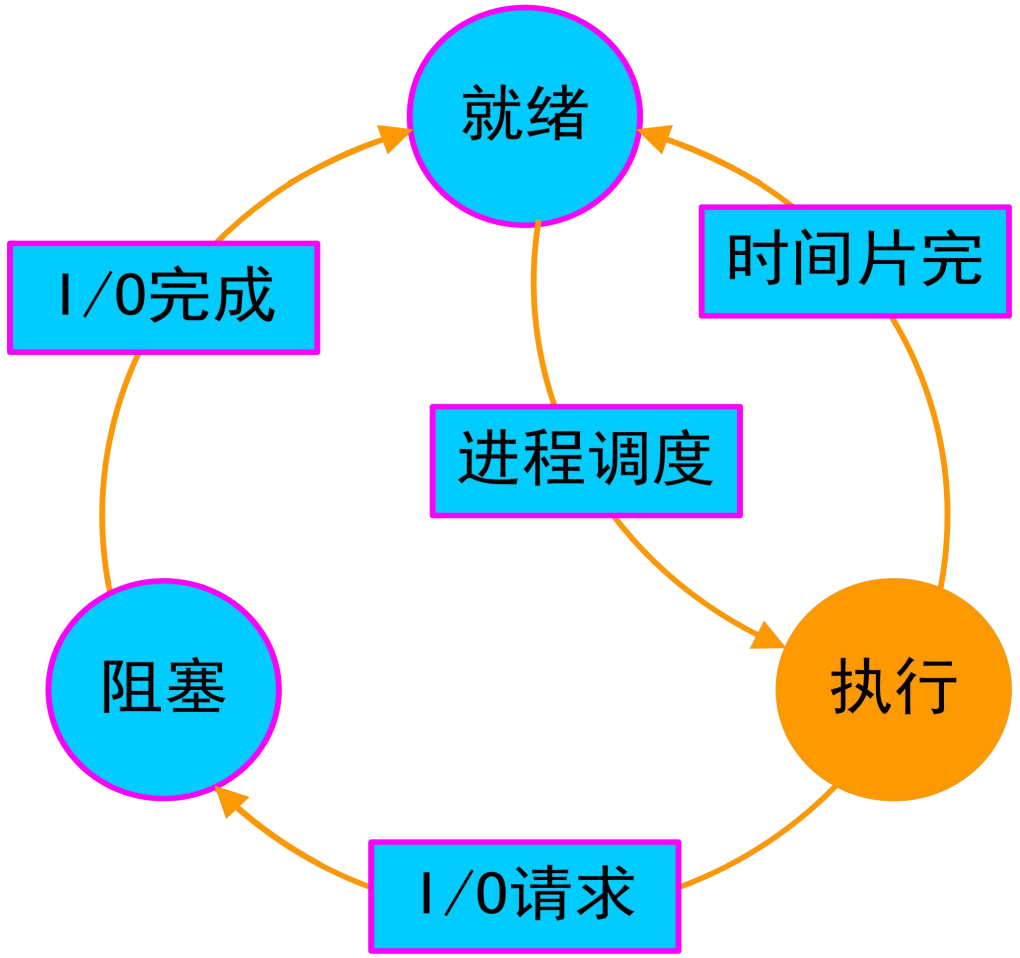
2.2.2 进程的状态及其转换



Ready：

该进程运行所需的一切**条件都得到满足**，但因处理机资源个数少于进程个数，所以该进程不能运行，而必须**等待分配处理机资源**，一旦获得处理机就立即投入运行。

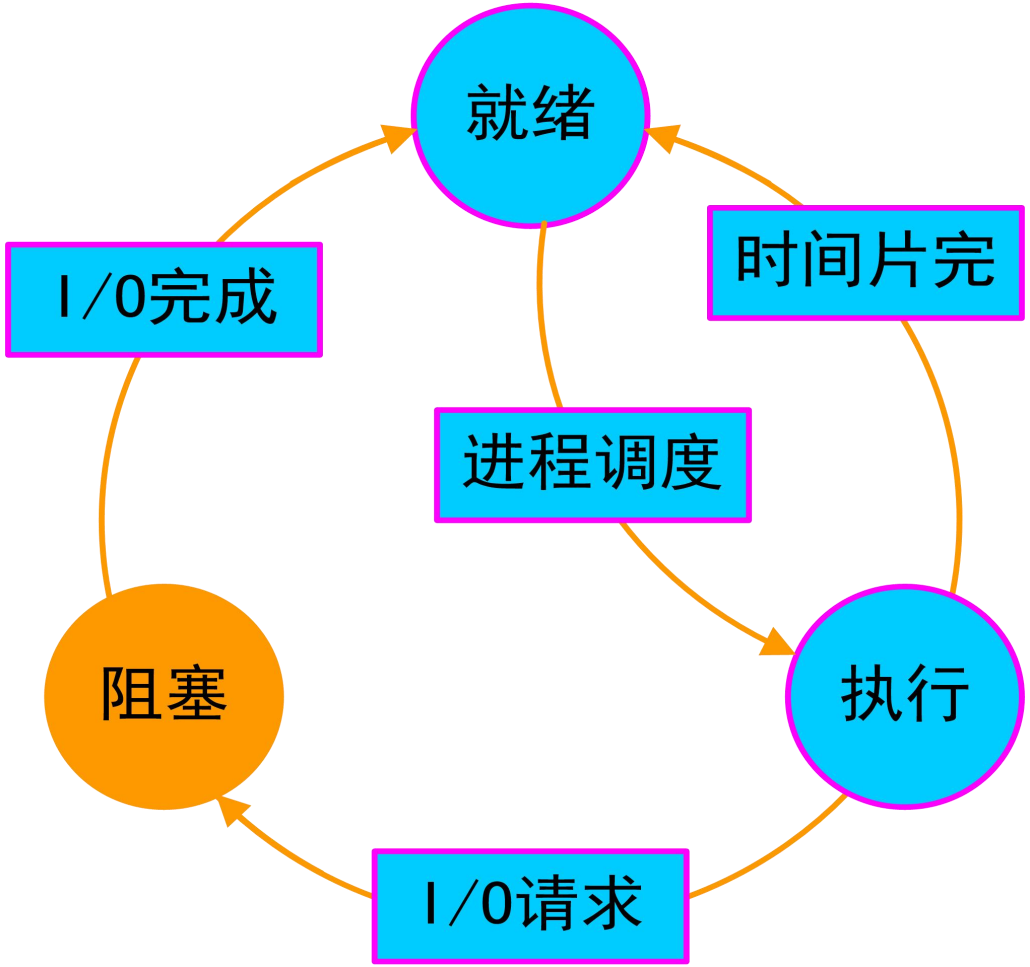
2.2.2 进程的状态及其转换



Running：

进程正在处理机上运行的状态，该进程已获得必要的资源，也**获得了处理机**，用户程序正在处理机上运行。

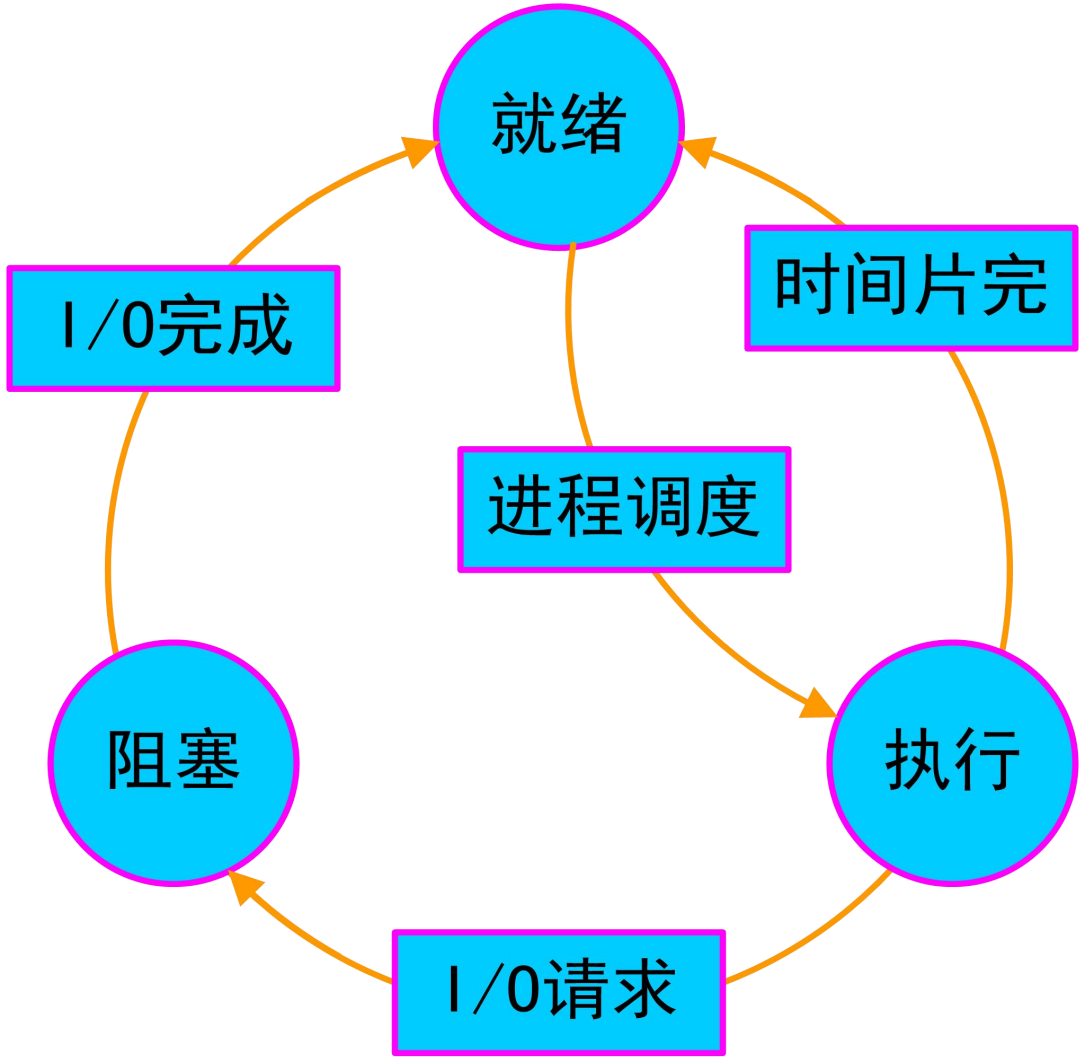
2.2.2 进程的状态及其转换



Blocked:

进程**等待某种事件完成**（例如，等待输入/输出操作的完成）而暂时不能运行的状态，处于该状态的进程**不能参加竞争处理机**，此时，即使分配给它处理机，它也不能运行。

2.2.2 进程的状态及其转换



状态变化：

就绪状态 → 执行状态

执行状态 → 就绪状态

执行状态 → 阻塞状态

阻塞状态 → 就绪状态

2.2.2 进程的状态及其转换

创建状态是指操作系统创建进程时，进程所处的状态。进程新建成功后即转入就绪状态，在就绪进程队列中排队。

操作系统创建进程需要为进程分配资源。因此，操作系统将根据系统的性能和内存容量的情况决定是否创建新的进程。如

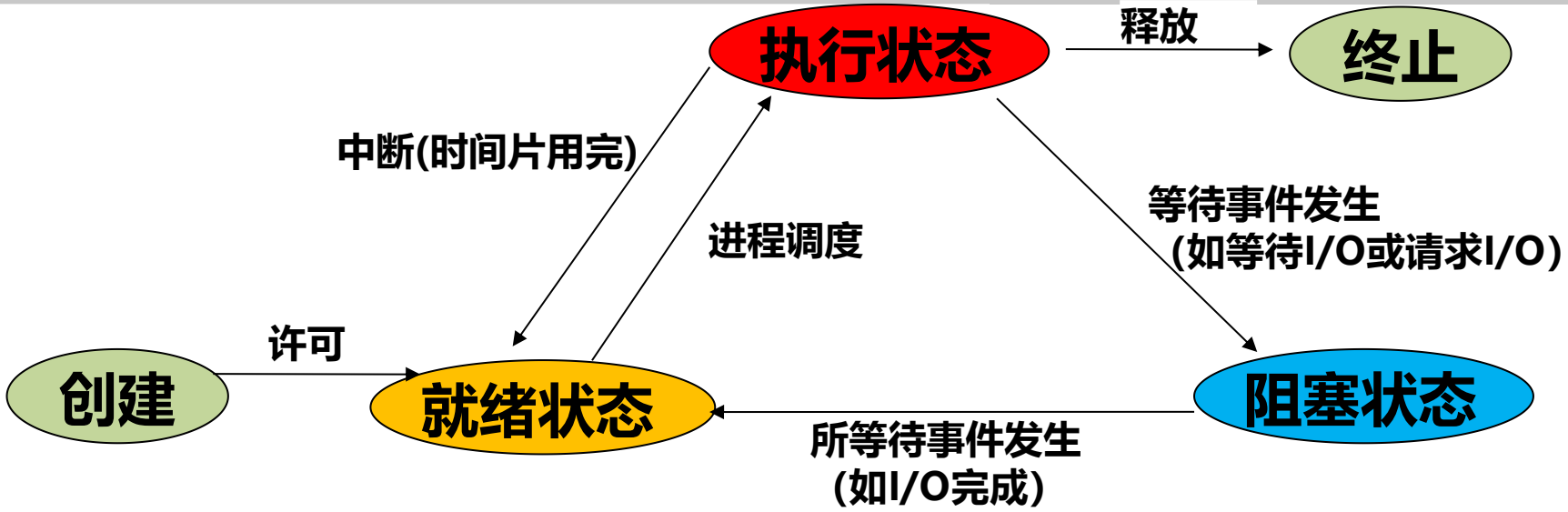
新进程分配

应或将创建

进程达到了结束点或进程出现了严重的错误时，会被操作系统终止或被其他有终止权的进程终止。

进入**终止状态**的进程不再被执行，等待操作系统完成进程终止处理。当操作系统完成进程终止处理后，操作系统会删除进程，收回进程所占用的资源。

小结



执行态：此时正用CPU；

就绪态：可运行,但未分到CPU；

阻塞态：不能运行,等待某个外部事件发生。

在一定条件下,进程状态才发生转换

2.2.3 挂起操作和进程状态的转换

- 挂起状态
  - 系统资源的需要
  - 调节竞争或消除故障
  - 终端用户的需要
  - 父进程的需要
  - 调节进程的需要

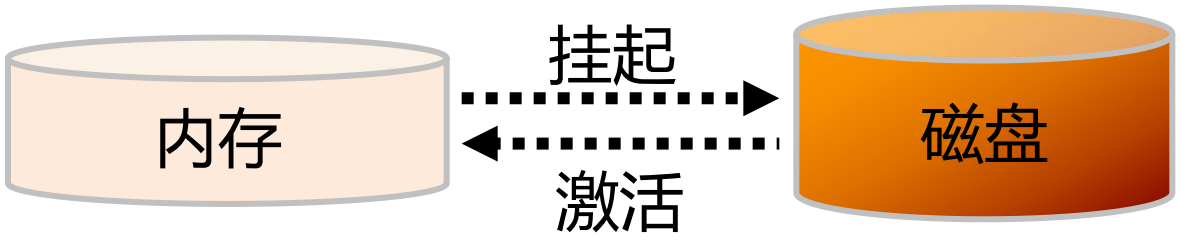
终端用户可以直接操作自己的程序。当程序员需要调试、检查和修改自己的程序时，可以要求挂起与程序相对应的进程

某些定期执行的进程，如系统监控进程、日志进程等，在执行时间未到而需要等待时，可以将进程挂起，对换到外存，从而减轻内存负担。当执行时间到时，再将这些进程激活换入到内存。

受到破坏时，为了解决进程的资源竞争或消除系统故障，操作系统需要挂起某些对资源竞争的进程或怀疑引起系统故障的进程，将进程对换到外存。

2.2.3 挂起操作和进程状态的转换

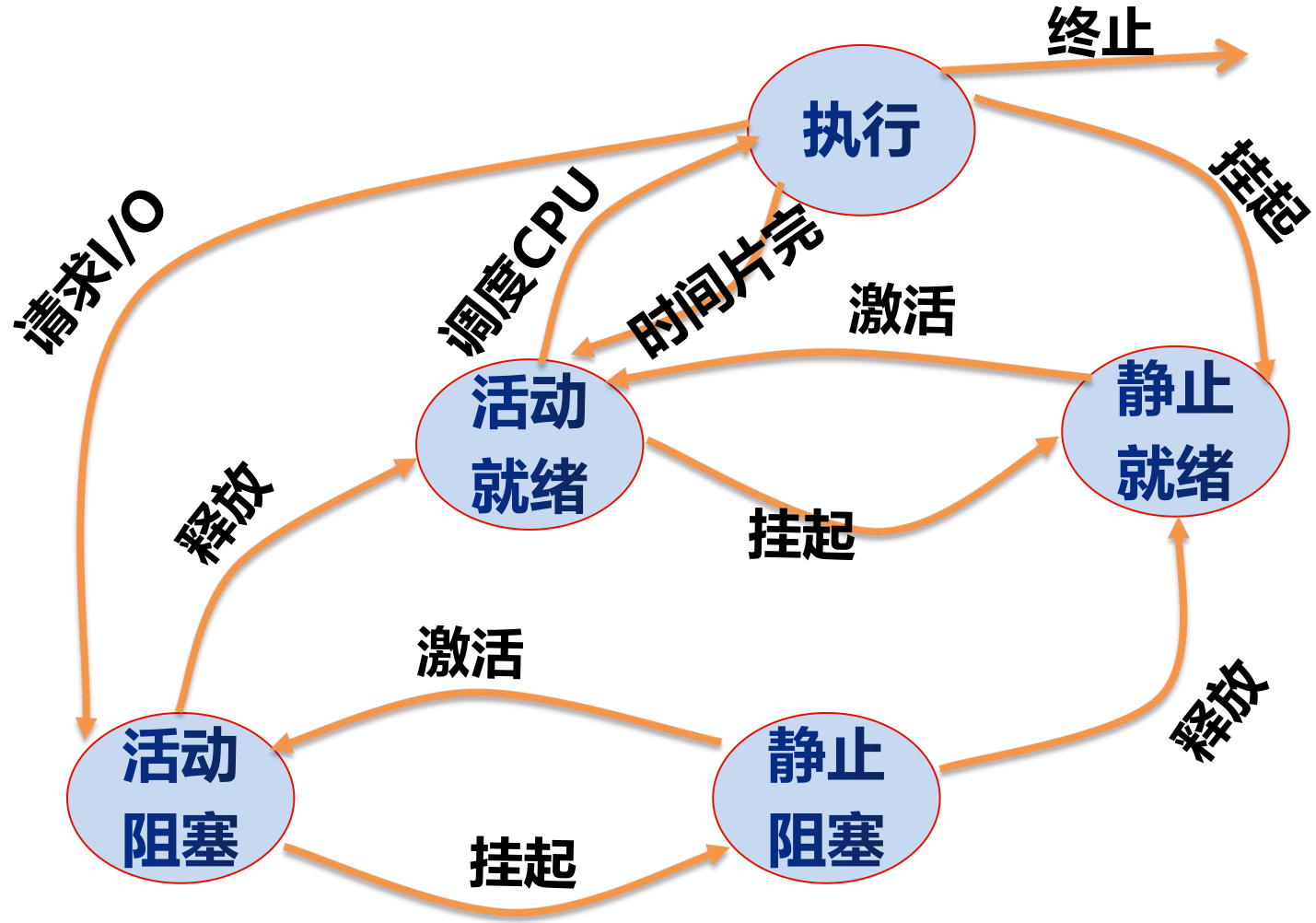
“挂起”的实质是使进程不能继续执行，处于**静止状态**，而没被挂起的进程称为活动状态。处于静止状态的进程，只有通过“激活”动作，才能转换成活动状态。



“挂起”常被用于进程对换中，挂起的进程**从内存对换到外存**，以便腾出更多内存空间接纳新创建的进程。常将内存中处于阻塞状态的进程挂起。

进程只能由操作系统，父进程，进程自身挂起

2.2.3 挂起操作和进程状态的转换



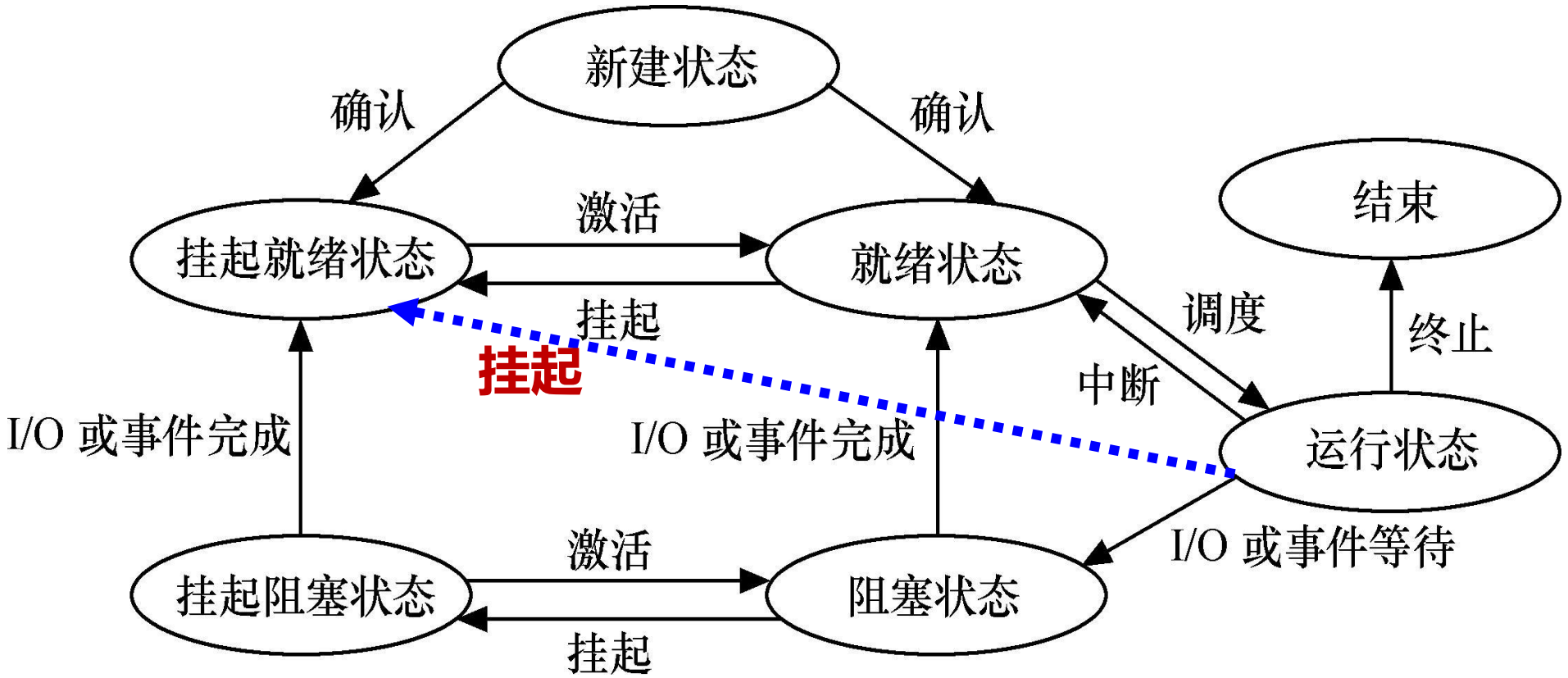
- 活动就绪 → 静止就绪
- 活动阻塞 → 静止阻塞
- 静止就绪 → 活动就绪
- 静止阻塞 → 活动阻塞

图2-7　具有挂起状态的进程状态图

小结

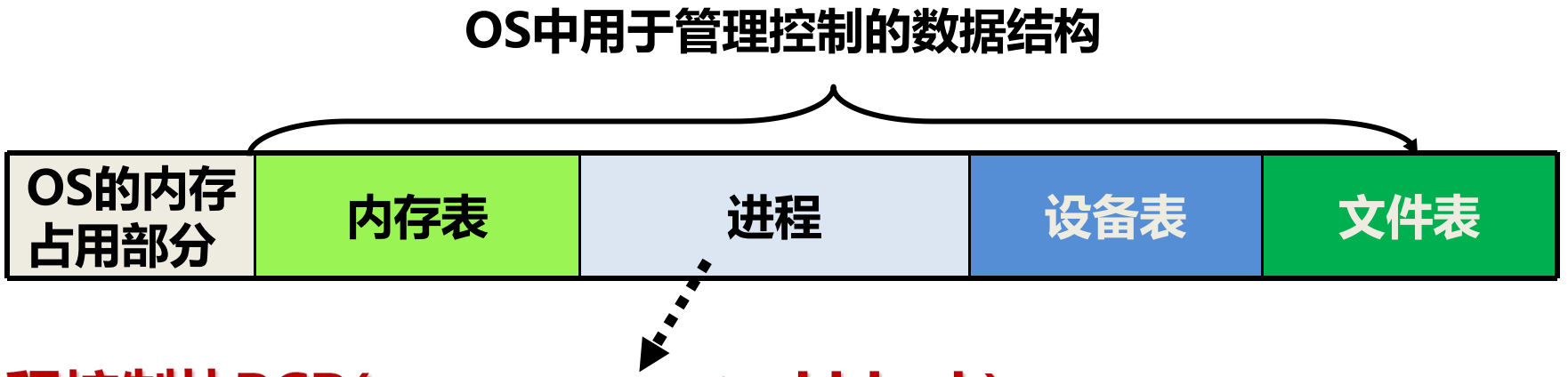
进程可以在：新建，就绪，阻塞，运行状态被挂起。

进程只能由操作系统，父进程，进程自身挂起



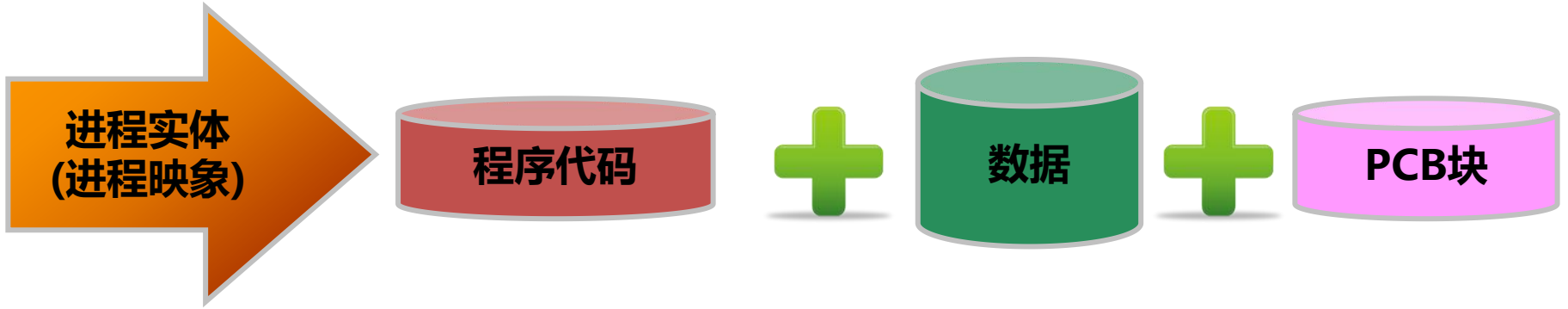
进程状态转换

2.2.4 进程管理中的数据结构



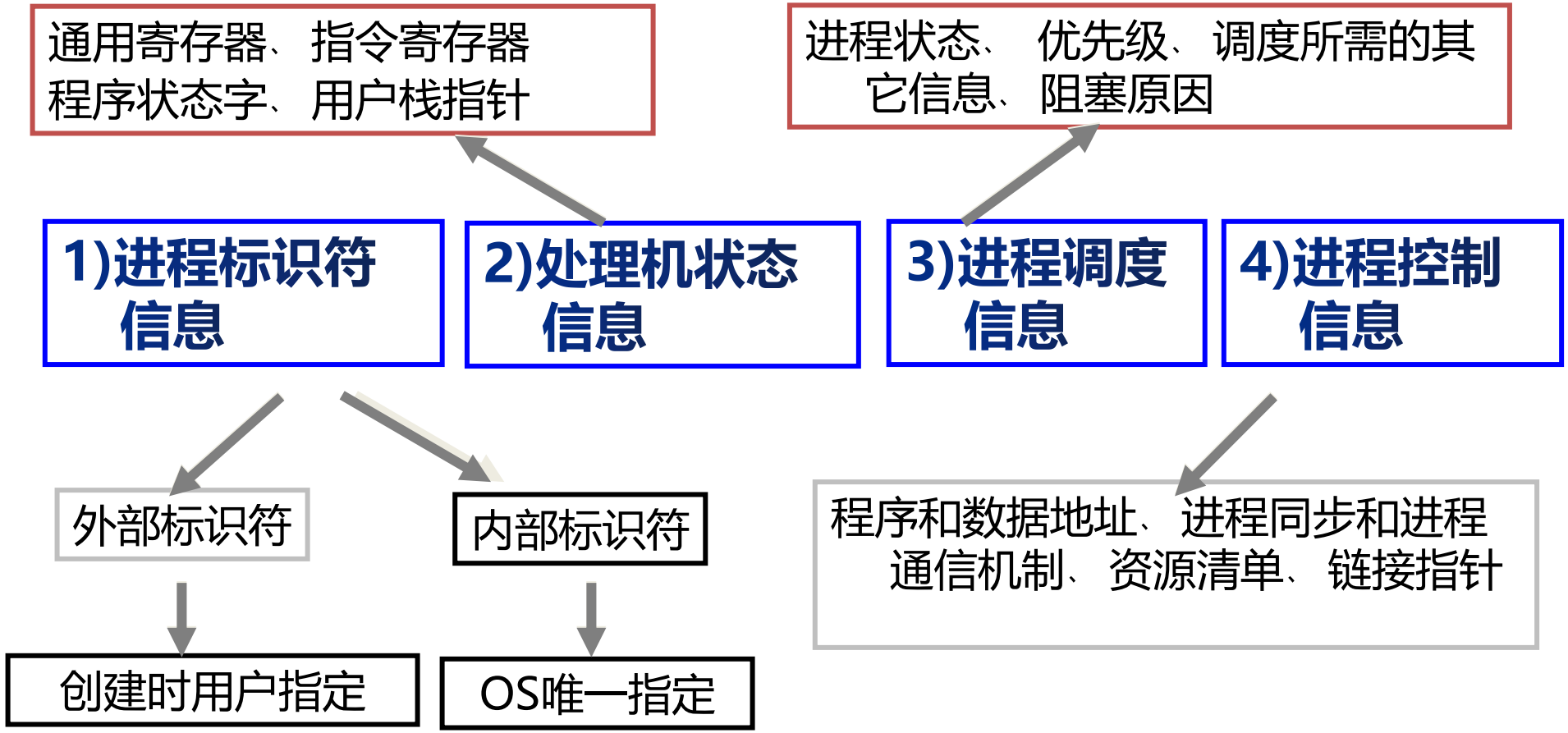
进程控制块PCB(process control block)

是进程的唯一实体,系统根据PCB块而感知进程的存在,即PCB块是进程存在的唯一标识。



2.2.4 进程管理中的数据结构

进程控制块中的信息



2.2.4 进程管理中的数据结构

PCB的作用

进程的创建　异步性　状态的转变　进程管理　同步和通信

1)进程标识符信息	2)处理机状态信息	3)进程调度信息	4)进程控制信息

OS不同，PCB的长度不一(时空开销)

OS的内存占用部分	内存表	进程	设备表	文件表

PCB的组织方式

1）线性表　2）链接方式　3）索引方式

2.2.4 进程管理中的数据结构

PCB的组织方式：

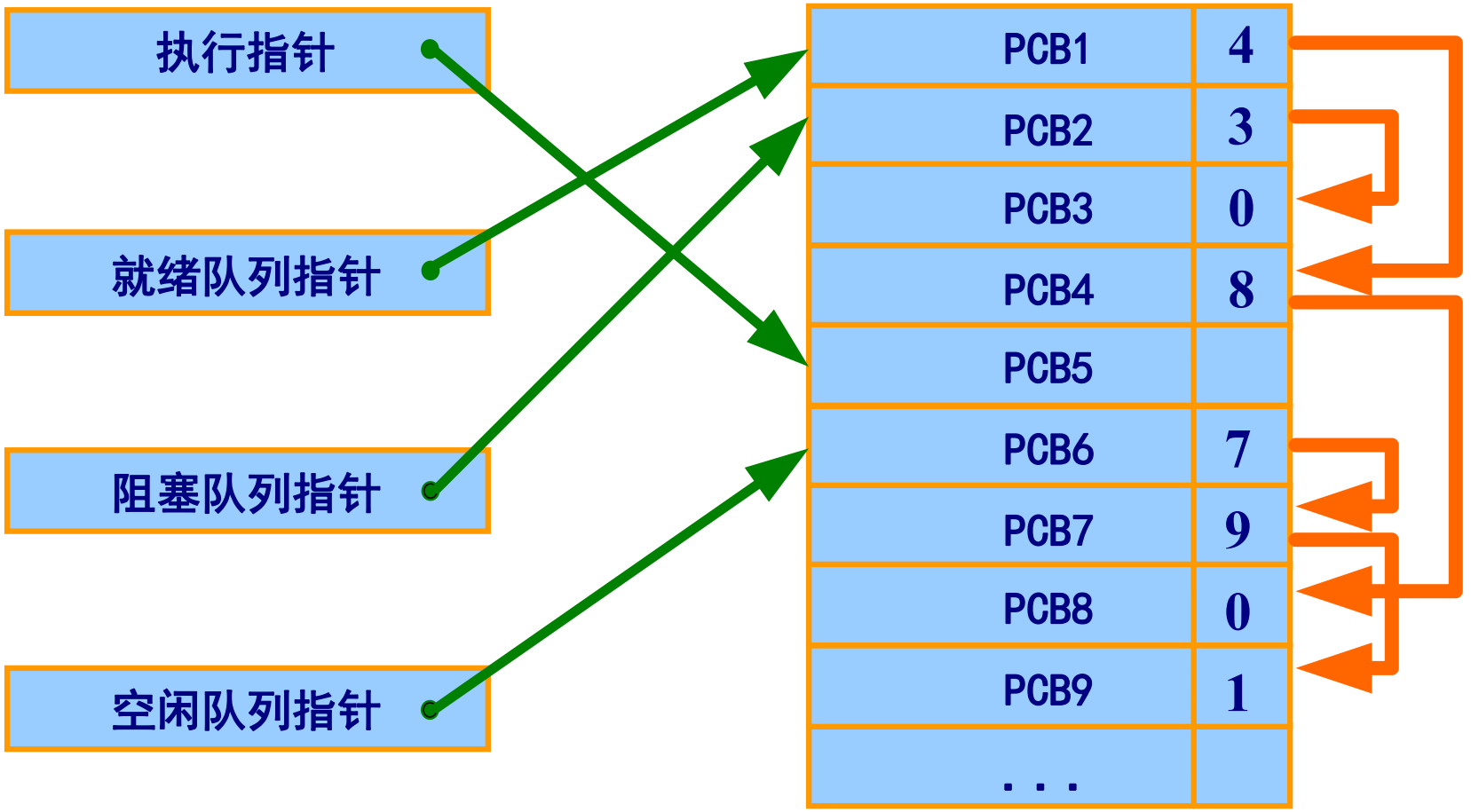
1）PCB线性队列示意图

PCB块指针

PCB块号	指针
PCB 1	4
PCB 2	3
PCB 3	5
PCB 4	8
PCB 5	6
PCB 6	7
PCB 7	9
PCB 8	2
PCB 9	X
⋮	

2.2.4 进程管理中的数据结构

2）PCB链接队列示意图



2.2.4 进程管理中的数据结构

链接方式

- 优点

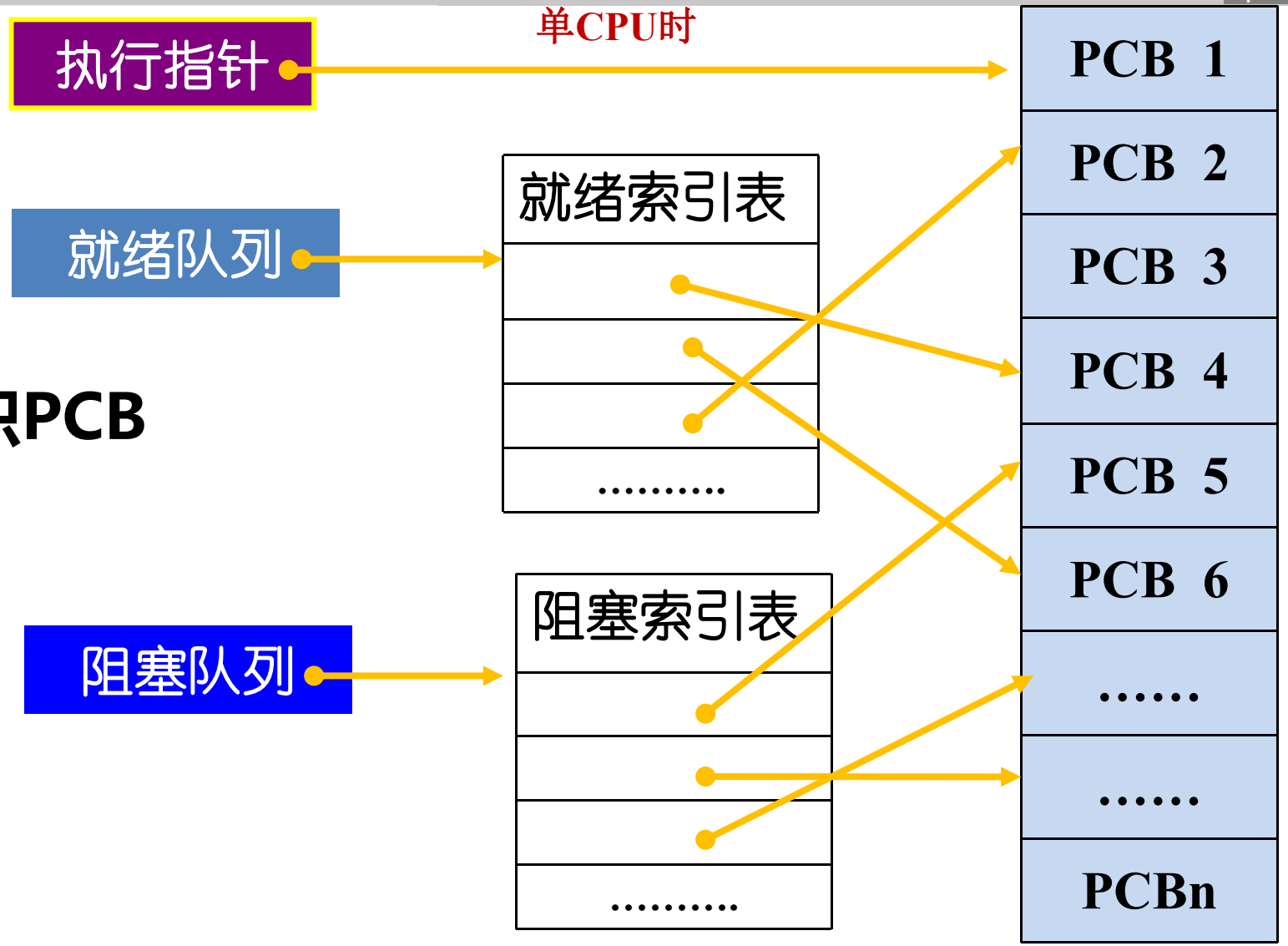
  直观，体现了进程的本身特性，如等待时间的长短、优先级的高低、需要处理时间的长短，为进程调度算法的实施提供了方便。

- 缺点

  以链接方式组织进程控制块的主要缺点是如果进程状态发生变化，则链接队列需要作相应的调整，进程控制块中的首部和尾部指针需要改变。

2.2.4 进程管理中的数据结构

3)按索引方式组织PCB



PCB的查找和修改都在索引表中进行

2.2.4 进程管理中的数据结构

索引方式

- 优点
  - 通过索引表可以快速得到进程控制块地址，不需要像链接方式一样，从链首到链尾查找；
  - 如果进程状态变化，不需要修改进程控制块的链接指针，只需要增加或删除索引表中的记录。
- 缺点
  - 索引表本身需占用内存空间；
  - 搜索索引表需要时间。

2.2.4 进程管理中的数据结构

内存映像

- 是进程在内存中的组成
- 包括如下内容：
  - 进程程序块
  - 进程数据块
  - 系统或用户堆栈
  - 进程控制块

进程程序块为执行的程序代码，规定了进程一次运行需要完成的功能。

进程运行时的全局变量、局部变量和常量等的存储区以及开辟的工作区。

每个进程捆绑的系统/用户堆栈，用来解决过程调用或系统调用时的信息存储和参数传递。

进程的标志信息、处理机状态信息、进程调度信息、进程控制信息。

2.2.4 进程管理中的数据结构

进程上下文

- 进程的物理实体和支持进程运行的环境合称为进程上下文（process context）。
  - 用户级上下文（user-level context）
  - 系统级上下文（system-level context）
  - 寄存器上下文（register context）。

由进程的正文区、数据区、用户
组成，在编译目标
地址空

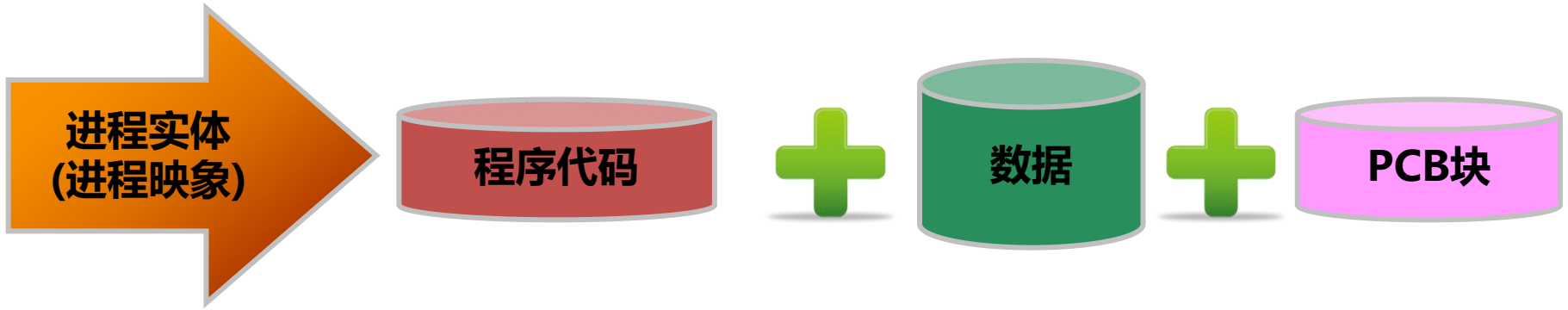
由进程控制块、内存管理信息、
进程环境块和系统堆栈等组成的进程
地址空间

由程序状态寄存器、各类控制寄存器、地址寄存器、通用寄存器和用户栈指针等组成。

小结

进程控制块PCB(process control block)

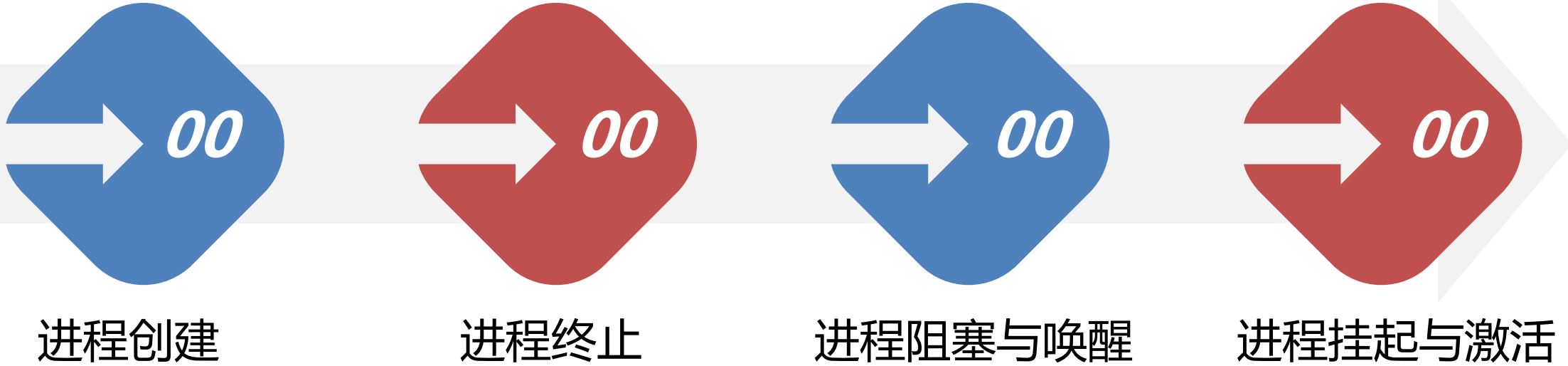
是进程的唯一实体,系统根据PCB块而感知进程的存在,即PCB块是进程存在的唯一标识。



PCB表的物理组织：常见的是线性方式、链接方式和索引方式。

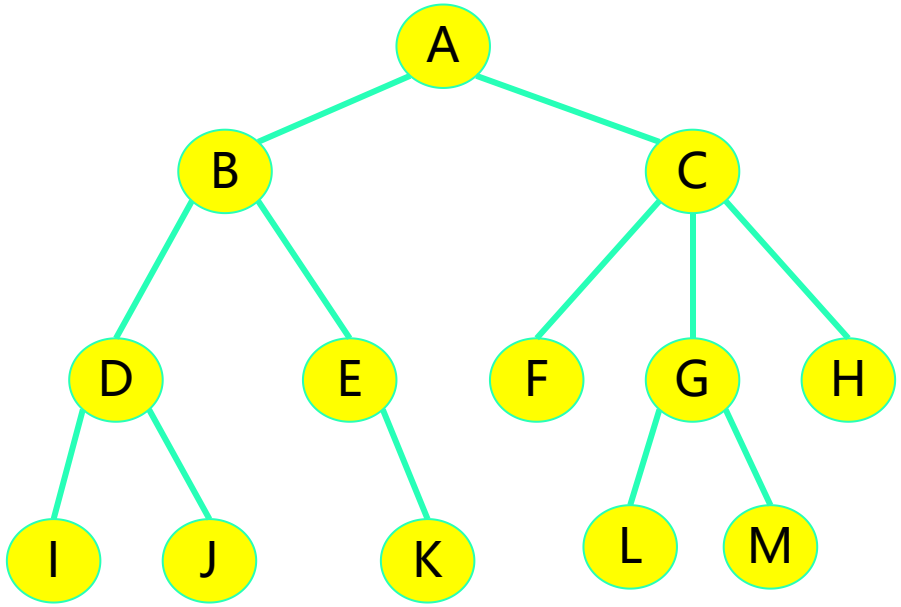
2.3 进程控制

- 进程管理最基本的功能
- 一般由OS内核中的原语实现
- 包括：



2.3.1 进程的创建

- 进程具有层次结构
- 创建新进程是通过创建原语完成的，被创建的进程称作子进程，而创建子进程的进程则称作父进程。子进程又可以创建自己的子进程，从而形成一棵有向的进程树，即进程图



- 关系
  - 子进程可以继承父进程的所有资源
  - 子进程撤销时，向父进程归还资源
  - 父进程撤销时，所有子进程也被撤销

2.3.1 进程的创建

- 进程创建过程：

① 申请空白PCB；

② 分配所需资源；

③ 初始化PCB；

④ 插入就绪队列。

2.3.1 进程的创建

- 引起进程创建的事件
  - 操作系统初始化
  - 提供用户服务(输入/输出)
  - 用户登录（分时系统）
  - 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

正在运行的进程由于程序自身需要可以用系统调用创建新进程。一个正在运行的进程，如果需要处理与其相关却又相互独立的事件，可以用系统调用来创建一个新的进程来完成这样的事件。如UNIX操作系统中父进程用系统调用创建子进程，子进程协同父进程完成同一任务。

用户提交批处理作业，当操作系统能够提供资源时，作业调度程序按照一定的算法，从批处理作业清单中调度某个作业并装入内存，为作业分配必要的资源，创建作业的进程。

2.3.1 进程的创建

- 创建步骤

进程创建原语的主要任务是创建进程控制块PCB。

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；

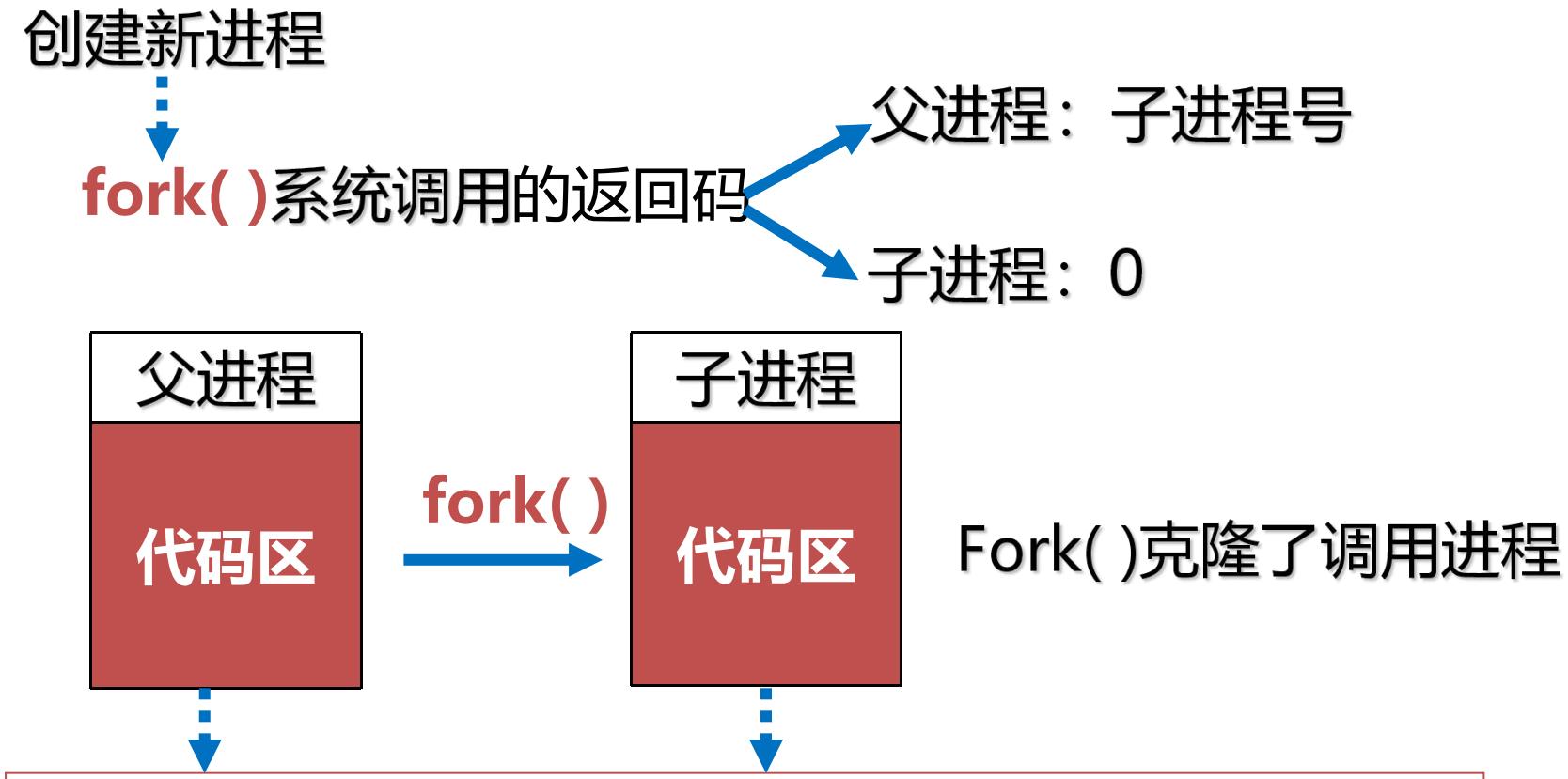
为进程分配除内存外的其他各种资源；

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。

2.3.1 进程的创建 Linux中进程的创建




```
if (fork()= = 0)
  {printf( “hello” );}/* 这是子进程代码 */
 else
  {printf( “ok” ) ;}  /* 这是父进程代码 */
```

2.3.2 进程的终止

- 终止原因
  - 进程正常结束: 批处理作业中的“halt”，多用户系统中的“log off”，用户程序中的“end”等。
  - 操作异常(
    运行超时、
  - 外界干预:操
    请求、父进程

第一步：根据被终止进程的标识符，从PCB集合中查

第二步：若被终止进程正处于执行状态，则终止该

第三步：若进程还有子孙进程，应将其所有子孙进

第四步：将进程所占有的全部资源释放（还给父进

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。

2.3.3 进程的阻塞与唤醒

引起进程阻塞和唤醒的事件

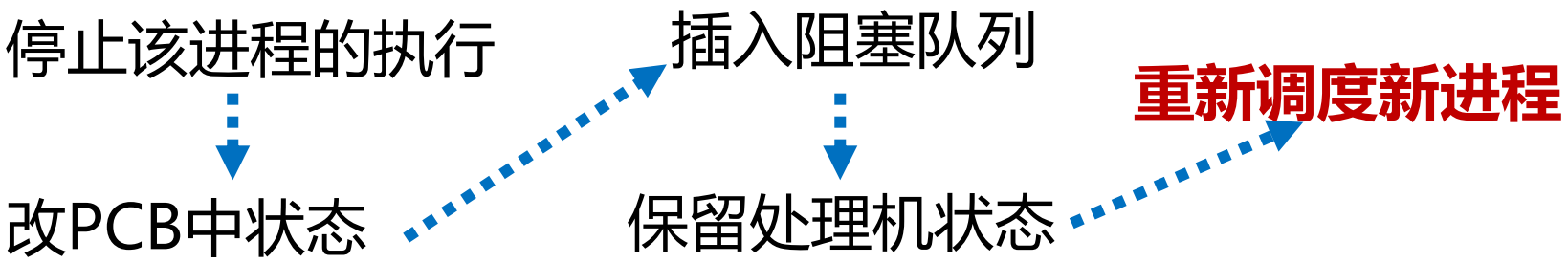
1)向系统请求资源失败：　请求打印机(间接制约)

2)等待某种操作的完成：　等待I/O的完成

3)新数据尚未到来：合作进程之间(直接制约)

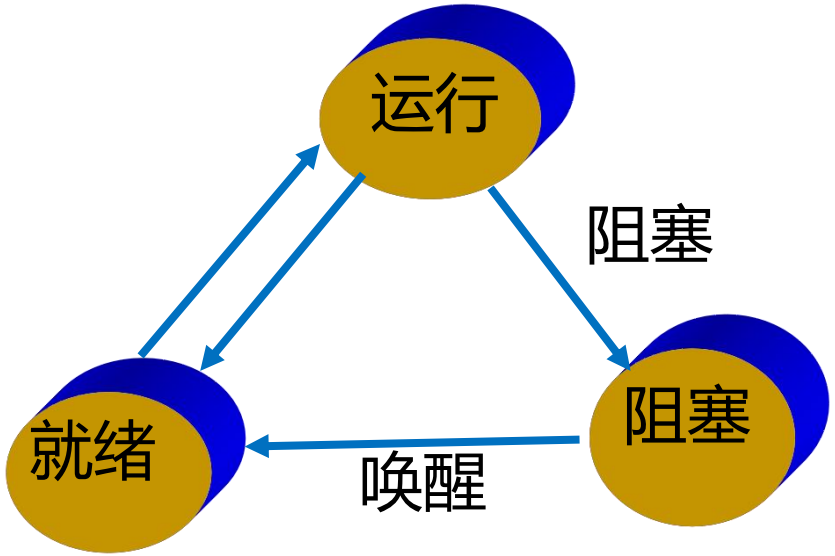
4)等待新任务的到达：　服务进程

进程阻塞过程（自身阻塞）



2.3.3 进程的阻塞与唤醒

进程阻塞是进程的一种自主行为，是进程为了等待某事件的发生，而自己调用阻塞原语使得自己放弃处理器，进入阻塞队列中等待



阻塞 ⟷ 唤醒

进程唤醒过程
（合作进程/系统进程唤醒）

从阻塞队列中移出

↓

改PCB的状态为就绪

↓

插入就绪队列

	主体	客体	
阻塞	A	A	自我行为
唤醒	B	非B	被动行为

2.3.4 进程的挂起与激活

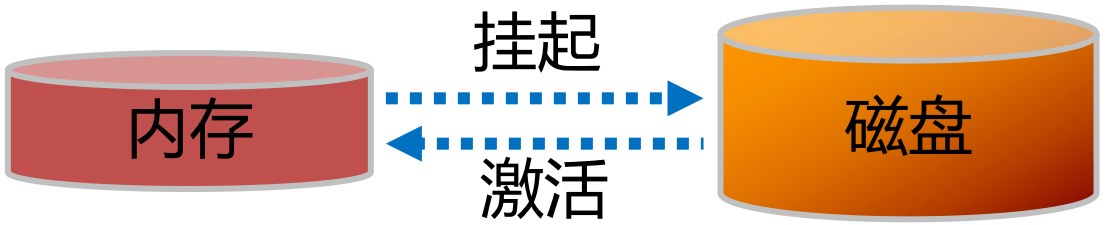
处于阻塞状态、就绪状态和运行状态的进程都可以被挂起，根据进程挂起前的状态决定挂起后的状态。

如果进程被挂起，进程的进程控制块中的信息要修改，进程上下文被放到外存。

如果进程挂起的时间到或者内存资源充足时，系统或有关进程，特别是进程同步中的原语操作，会激活被挂起的进程。

激活进程的主要工作是将进程上下文从外存调入内存，修改进程控制块信息并调入内存，修改进程控制块中的进程状态，并按照进程状态将进程控制块排入到相应的进程队列。

2.3.4 进程的挂起与激活



挂起进程

修改PCB块中的信息

进程调入外存 → 进程放入相应的状态队列中

激活进程

将调出的进程从外存调入内存

修改PCB块中的进程状态

并按照进程状态将进程控制块排入到相应的进程队列

进程只能由操作系统，父进程，进程自身挂起

2.3.4 进程的挂起与激活

1）三对进程**控制原语**

- 创建Creation
- 结束Termination
- 阻塞Block
- 唤醒Wakeup
- 挂起Suspend
- 激活Active

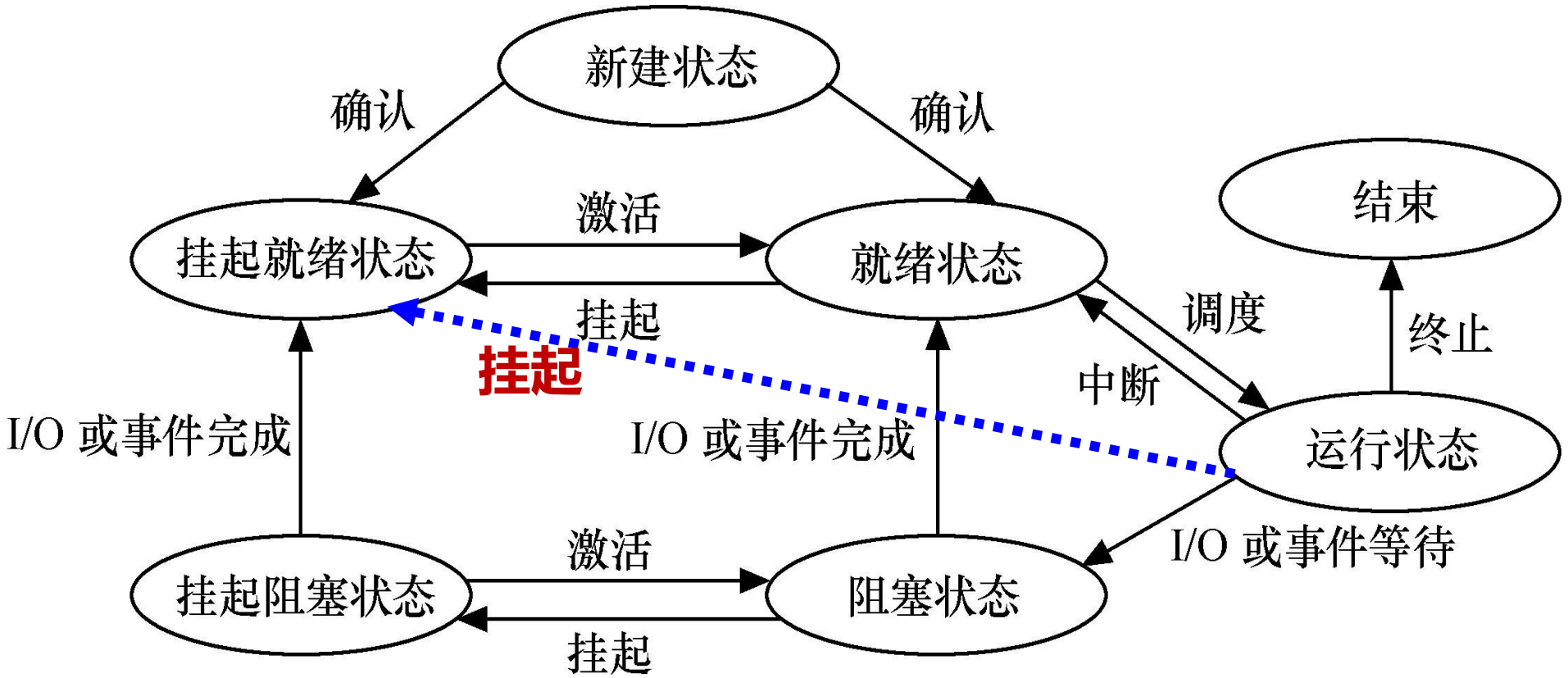
2）PCB块使用频繁，放入**内存的专用区**

3）每个系统的PCB块结构都不相同，进程的状态也都不一样。

小结

进程可以在：新建，就绪，阻塞，运行状态被挂起。

进程只能由操作系统，父进程，进程自身挂起



进程状态转换

2.4 进程通信概念

进程通信是指进程之间的信息交换

低级进程通信： 进程的同步和互斥

- 效率低
- 通信对用户不透明

高级进程通信

- 使用方便
- 高效地传送大量数据

2.4.1 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（效率低）
- 基于共享存储区的通信方式（高级）

管道通信

- 管道：用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。
- 管道机制的协调能力：互斥、同步、对方是否存在

2.4.1 进程通信的类型(2)

消息传递系统

- 直接通信方式
- 间接通信方式（通过邮箱）

客户机-服务器系统

- 套接字（Socket）
- 远程过程调用（RPC）和远程方法调用（RMI，Java）

2.4.2 消息传递通信的实现方式

直接通信方式

- 发送原语：send(P, message) 发送拼命搞信息到进程P
- 接收原语：receive(Q, message) 从进程Q接收消息

通信链路的属性

- 自动建立链路
- 一条链路恰好对应一堆通信进程
- 每对进程之间只有一个链路存在
- 链路可以是单向的，通常都是双向

2.4.2 消息传递通信的实现方式

间接通信方式：通过信箱来完成

- 定向接收消息
- 每个消息队列都有一个唯一的ID
  - 只有他们共享了一个消息队列进程才能通信
- 信箱类型
  - 私用邮箱，公共邮箱，共享邮箱

send(mailbox, message)
receiver(mailbox,message)

通信链路属性

- 只有进程共享一个共享的消息队列才建立链路
- 链接可以与许多进程相关联
- 每对进程可以共享多个通信链路
- 连接可以单向或双向

2.4.3 Linux进程通信方式

2.4.3 无名管道实例

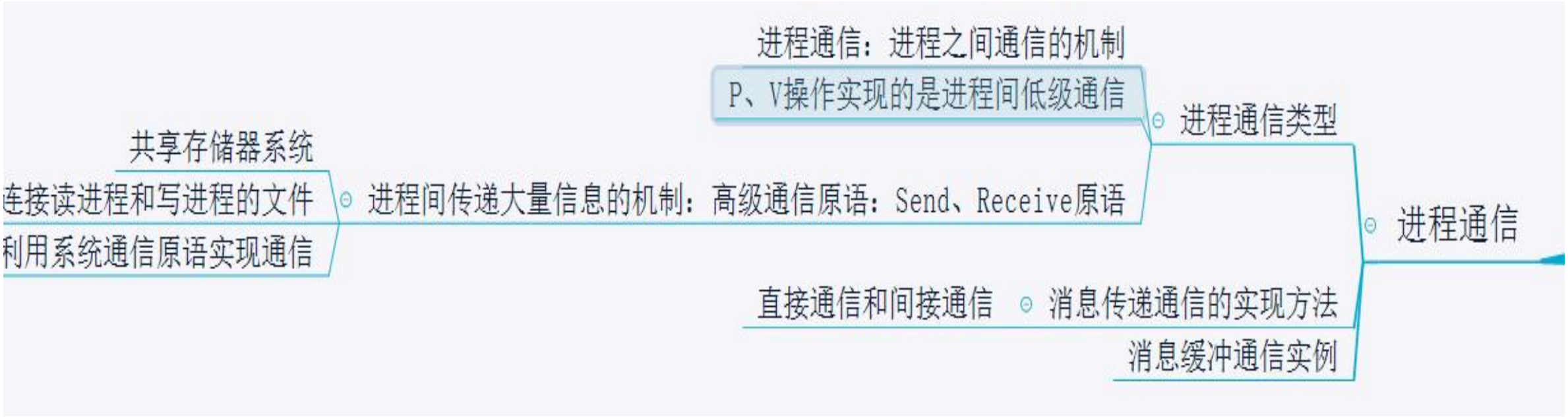

```
#define INPUT 0
#define OUTPUT 1
void main() {
        int file_descriptors[2];
        pid_t pid;                    /*定义子进程号*/
        char buf[256];
        int returned_count;
        pipe(file_descriptors);   /*创建无名管道*/
        if((pid = fork()) == -1) {   /*创建子进程*/
                printf("Error in fork/n");
                exit(1);
        }
        if(pid == 0) {                    /*执行子进程*/
                printf("in the spawned (child) process.../n");
        /*子进程向父进程写数据，关闭管道的读端*/
        close(file_descriptors[INPUT]);
        write(file_descriptors[OUTPUT], "test data",
                                        strlen("test data"));
        exit(0);
}
else {              /*执行父进程*/
        printf("in the spawning (parent) process.../n");
        /*父进程从管道读取子进程写的数据，关闭管道的写端*/
        close(file_descriptors[OUTPUT]);
        returned_count = read(file_descriptors[INPUT],
                                        buf, sizeof(buf));
        printf("%d bytes of data received from spawned process:
        %s/n",returned_count, buf); } }
```

2.4.3 有名管道实例


```
#include <stdio.h>
#include <unistd.h>
void main() {
FILE * in_file, *out_file;
int count = 1;
char buf[80];
in_file = fopen("mypipe", "r");    /*读有名管道*/
if (in_file == NULL) {
printf("Error in fdopen./n");
exit(1);
}
while ((count = fread(buf, 1, 80, in_file)) > 0)
printf("received from pipe: %s/n", buf);
fclose(in_file);
```

```
out_file = fopen("mypipe", "w");   /*写有名管道*/
if (out_file == NULL) {
printf("Error in fdopen./n");
exit(1);
}
sprintf(buf,"this is test data for the named pipe
example./n");
fwrite(buf, 1, 80, out_file);
fclose(out_file);
}
```

小结


进程通信
进程通信类型
进程通信：进程之间通信的机制
P、V操作实现的是进程间低级通信
进程间传递大量信息的机制：高级通信原语：Send、Receive原语
共享存储器系统
连接读进程和写进程的文件
利用系统通信原语实现通信
消息传递通信的实现方法
直接通信和间接通信
消息缓冲通信实例

- 具体看思维导图

小结

无论是在单机系统、多机系统还是计算机网络中，消息传递机制都是一种使用十分广泛的进程通信机制，我们必须了解以下几个问题。

（1）什么是消息传递通信机制。消息传递通信机制是指以格式化的消息为进程间数据交换单位的进程通信方式。

（2）消息传递通信机制有哪几种实现方式。消息传递通信机制有直接通信和间接通信两种实现方式，在学习时应注意比较它们在原语的提供、通信链路的建立、通信的实时性等方面的异同。

（3）如何协调发送进程和接收进程。为了使诸进程间能协调地进行通信，必须对进程通信的收、发双方进行进程同步

（4）消息队列通信机制是一种常用的直接通信方式。

长期“冷战”之后

Linux与Windows究竟谁会更胜一筹?

20世纪90年代初期，基于MINIX和UNIX思想而研发的开源Linux系统面市，其是一款支持多用户、多任务、多线程和多内核的操作系统，不仅能够运行UNIX工具软件、应用程序和网络协议，还具有稳定的系统性能。发展至今，Linux已有上百种不同的发行版本。

在Linux系统稳步发展过程中，Windows系统亦不分昼夜地进行着功能完善、界面美化以及版本更新等工作。进入21世纪后，微软公司的Windows系统在个人计算机领域基本占领了垄断地位。由垄断所导致的潜在安全问题是各国相关部门尤为关心的核心问题，而解决该问题（即去微软公司化）的主流途径便是采用开源Linux系统。

长期“冷战”之后

Linux与Windows究竟谁会更胜一筹?

但是，要想简单地通过采用Linux系统实现去微软公司化，实属不易。

典型例子：2004年，德国慕尼黑政府宣布将政府办公计算机中所采用的Windows系统换为Linux系统，希望此举可以降低运维信息化成本；然而10年之后，这场“吃螃蟹”的试验并未获得预期的效果；近年来，该政府又开始逐步在政府办公计算机上重新安装Windows系统。

从此类案例中可以看出，单纯地通过采用Linux系统+开源软件的模式来降低运维信息化成本，效果并不理想，大都会陷入所须开发的专业软件多和后期维护工作量大等风险中。此外，由于Linux系统代码开源，其产品化始终不足，因此，采用Linux系统实现去微软公司化缺乏相应的产业基础。

长期“冷战”之后

Linux与Windows究竟谁会更胜一筹?

形势即便如此，我们也绝对不应放弃反垄断！在此情形下，我们知识分子与科技人才更应深度思考：究竟应当如何解决操作系统垄断问题，以及如何在解决该问题的过程中，通过发挥我们自身的价值，助力研发自主可控的国产操作系统?

TRANSITION PAGE

第二章 进程的描述与控制

2.1 前趋图和程序的执行

2.2 进程的描述

2.3 进程控制

2.4 进程同步基本概念

2.5 管程机制

2.6 经典的同步问题

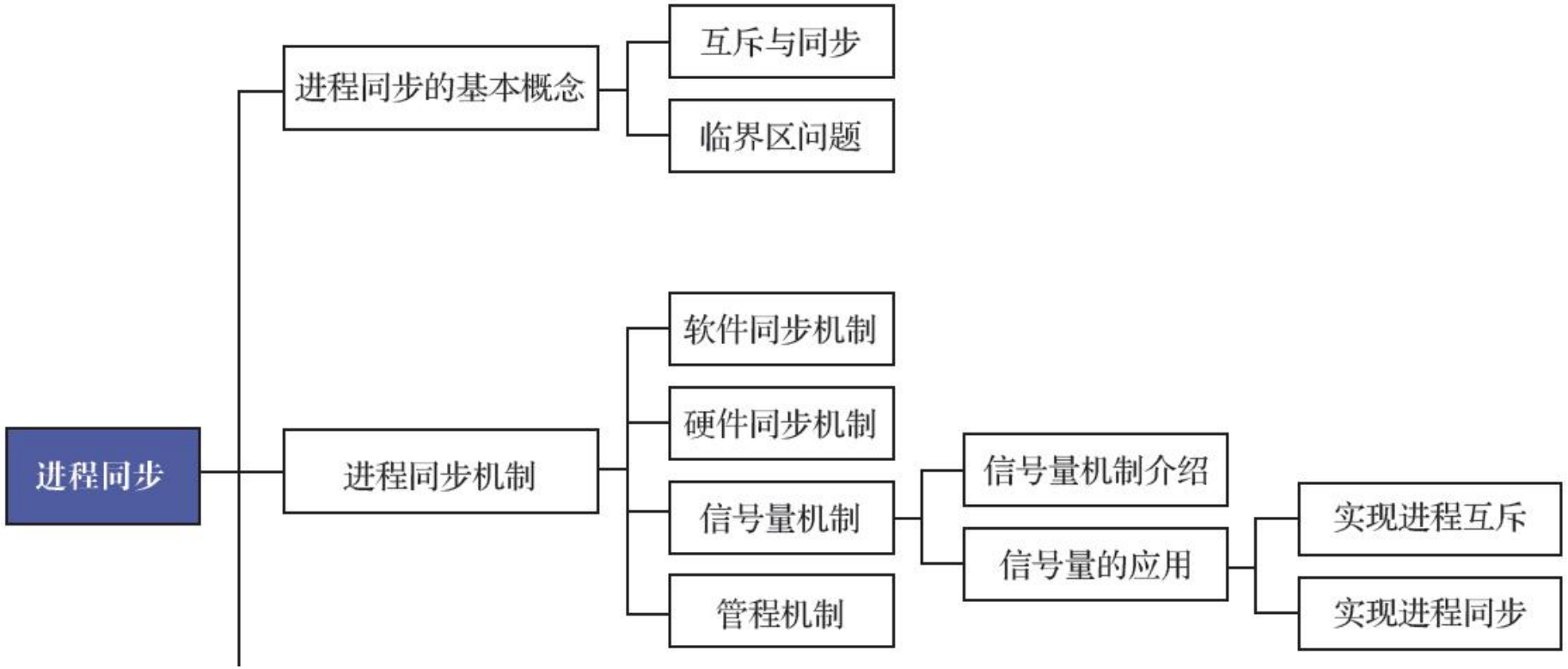
2.7 进程通信

2.7 线程及其实现

回顾

- 在OS中引入进程，可以使系统中的多道程序并发执行，提高了资源利用率及系统吞吐量。但也会使系统更加复杂，引起系统对资源的无序争夺。
- 为了保证多进程能有条不紊运行，必须引入进程同步机制。

本节课内容导图


进程同步
进程同步的基本概念
互斥与同步
临界区问题
进程同步机制
软件同步机制
硬件同步机制
信号量机制
信号量机制介绍
信号量的应用
实现进程互斥
实现进程同步
管程机制

2.4 进程同步与互斥

- 教学目的和要求
  - 知道同步互斥问题的原因
  - 掌握临界区和信号量概念
  - 掌握解决简单同步互斥问题的信号量方法
- 重点
  - 临界区和信号量概念
- 难点
  - 应用信号量解决同步互斥问题

2.4.1 进程同步概念的引入

- 把异步环境下的一组**并发进程**因直接**制约**而互相发送消息互相合作、互相等待，使得各进程**按一定的速度执行**的过程，称为**进程同步**。

- 具有同步关系的一组并发进程称为协作进程。

- 进程***同步***机制的主要**任务**——在执行次序上对多个协作进程进行协调，使并发执行的诸多协作进程之间能按照一定的规则（或时序）共享资源、相互合作，从而使程序的执行具有**可再现性。**

2.4.2 两种制约关系

- 在多道程序系统中，由于资源共享或进程合作，使进程间形成间接相互制约和直接相互制约关系。
- 间接制约关系（互斥关系）
  - 这是由于竞争相同资源而引起的，得到资源的程序段可以投入运行，而得不到资源的程序段就是暂时等待，直至获得可用资源时再继续运行。
- 直接制约关系（同步关系）
  - 这通常是在那些逻辑上相关的程序段之间发生的，一般是由于各种程序段要求共享信息引起的。
- 这种相互依赖、相互合作又相互竞争的关系，意味着进程间需要用某种形式的通信来实现，主要表现为进程互斥与同步两方面。

同步关系

- **例子1：**

有一输入进程 A 通过单缓冲向进程 B 提供数据。当该缓冲空时，计算进程因不能获得所需数据而阻塞，而当进程 A 把数据输入缓冲区后，便将进程 B 唤醒；反之，当缓冲区已满时，进程 A 因不能再向缓冲区投放数据而阻塞，当进程 B 将缓冲区数据取走后便可唤醒 A。

7

互斥关系

- **例子2：两个进程共享打印机**

进程A：

……

打印文件AAA；

……

进程B：

……

打印文件BBB；

……

打印结果：

AAA的文本；
BBB的文本；
AAA的文本；
AAA的文本；
BBB的文本；
AAA的文本；
BBB的文本；
BBB的文本；
BBB的文本；
AAA的文本；
AAA的文本；
……

进程间的关系（一）：同步 synchronism

- **直接制约**：“进程—进程”

  进行**协作**，等待来自其他进程的信息，否则就进行不下去，“**同步**”。
- **概念**：指系统中多个进程中发生的事件存在某种时序关系，必须协同动作，相互配合，从而共同完成一项任务。
- 一个进程运行到某一点时要求另一伙伴进程为它提供消息，在未获得消息之前，该进程处于等待（阻塞）状态，获得消息后被唤醒进入就绪状态。

进程间的关系（一）：同步 synchronism

例子3：“司机 - 售票员”问题（同步）

司机进程

```
While(True)
{
  启动公车；
  驾驶公车；
  停止公车；
}
```

售票员进程

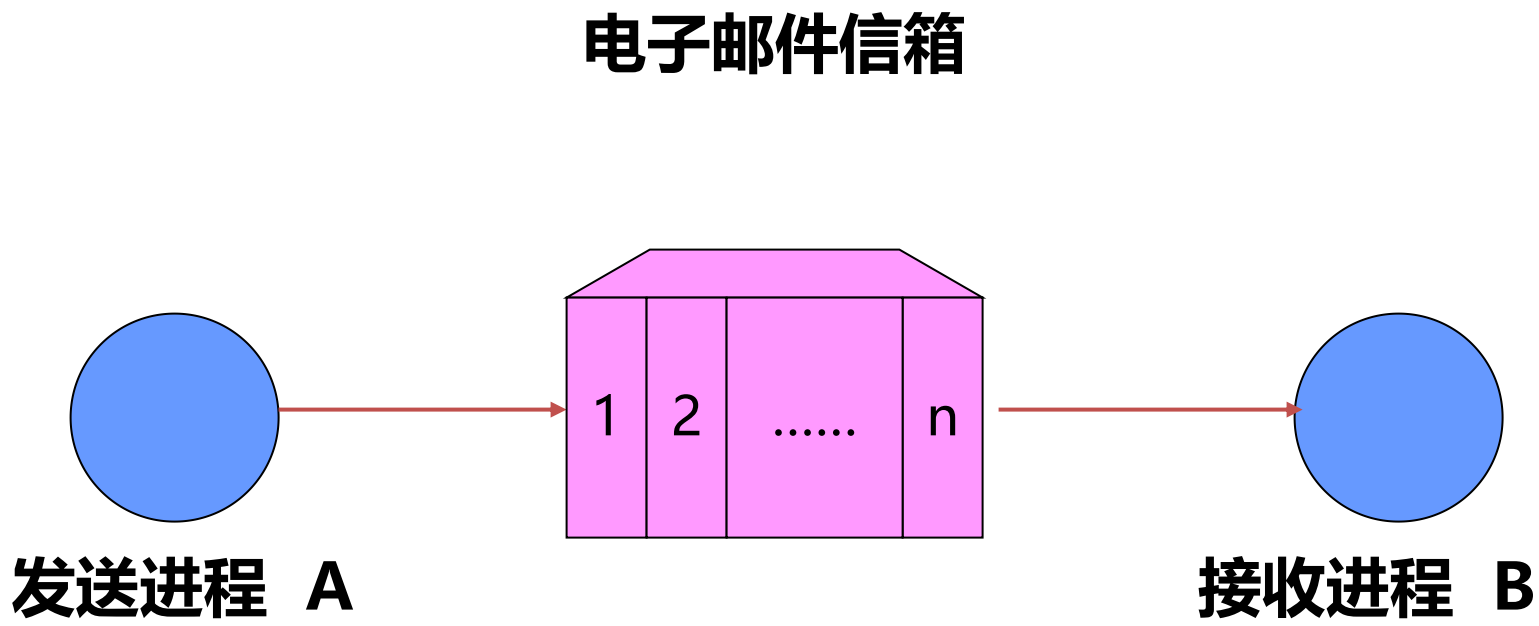
```
While(True)
{
  关车门；
  卖车票；
  开车门；
}
```

正确运行过程

```
While(True)
{
  （售票员）关车门
  （司机）启动公车；
  （司机）驾驶公车；
  （售票员）卖车票
  （司机）停止公车；
  （售票员）开车门；
}
```

进程间的关系（一）：同步 synchronism

例4:

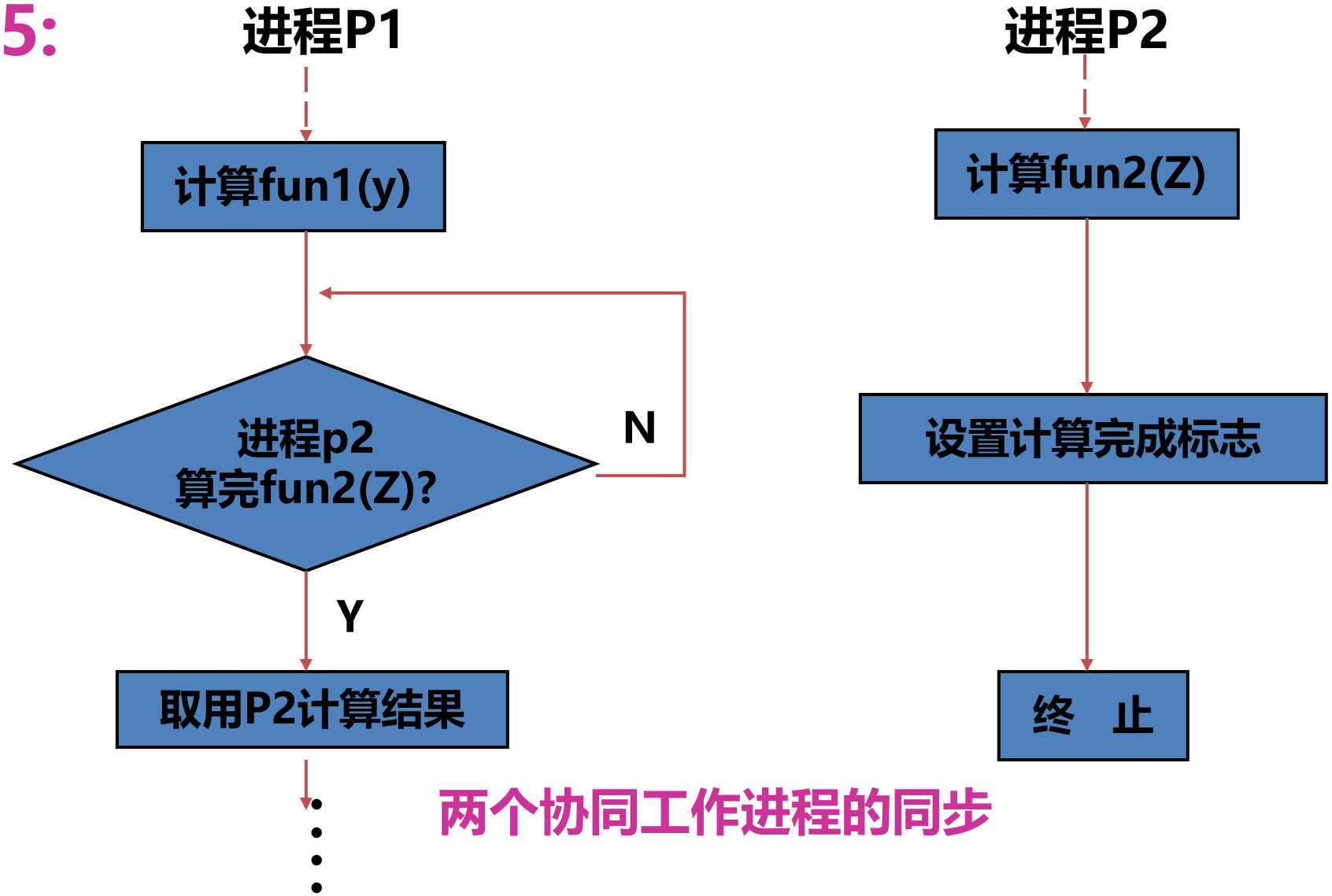


接收进程必须等待发送进程发送信件。

进程间的关系（一）：同步 synchronism

X = fun1(y)*fun2(Z)

例 5:



两个协同工作进程的同步

进程间的关系（二）：互斥 mutual exclusion

进程互斥：

- **间接制约**：“进程—资源—进程”

  进行**竞争**，独占分配到的部分或全部共享资源，等待释放后才能使用，“**互斥**”。

- **概念**：有些资源需要互斥使用，因此各进程间要相互竞争，以使用这些**互斥资源**，进程的这种关系为进程的互斥。

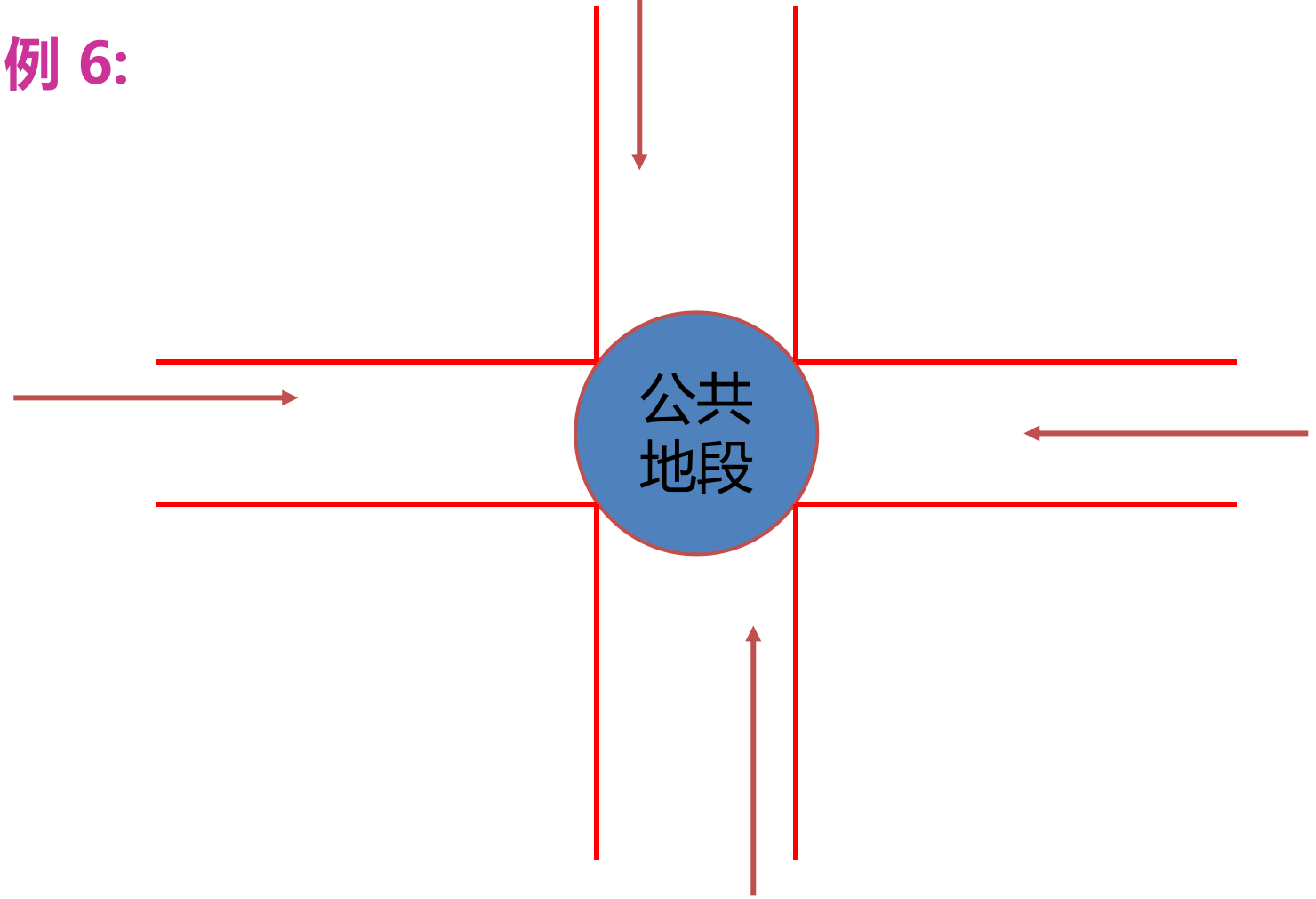
- **解决进程互斥**有两种办法：

  1.由竞争各方**平等协商**。

  2.**引入进程管理者**，由管理者来协调竞争各方对互斥资源的使用。

进程间的关系（二）：互斥 mutual exclusion

例 6:



交通十字路口的控制：公共地段互斥

2.4.3 临界资源 critical resource

系统中某些软件或者硬件资源一次只允许一个进程使用，称这样的资源为临界资源 或互斥资源 或共享变量。

如：外设（打印机、磁带机、绘图仪等）、共享代码段、共享数据结构等。

2.4.3 临界资源 critical resource

系统中共享的资源分为

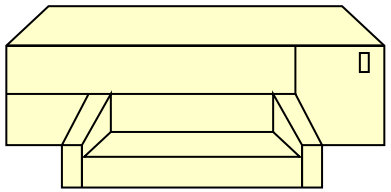
互斥共享资源(打印机)

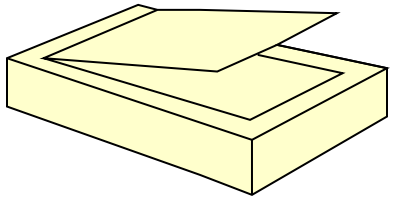
同时共享资源(磁盘)

临界资源(Critical Resouce):（互斥资源、逐次再使用资源）

一段时间内只允许一个进程使用的资源。

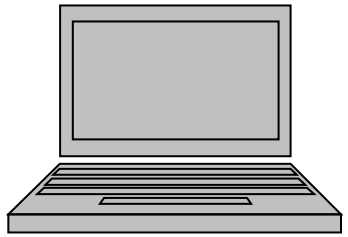
硬件

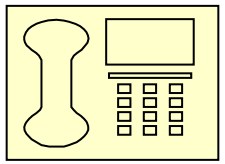


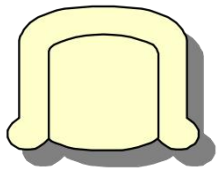


扫描仪

生活中

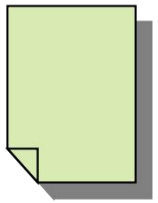






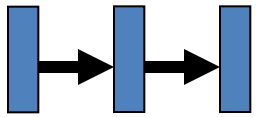
座位

软件



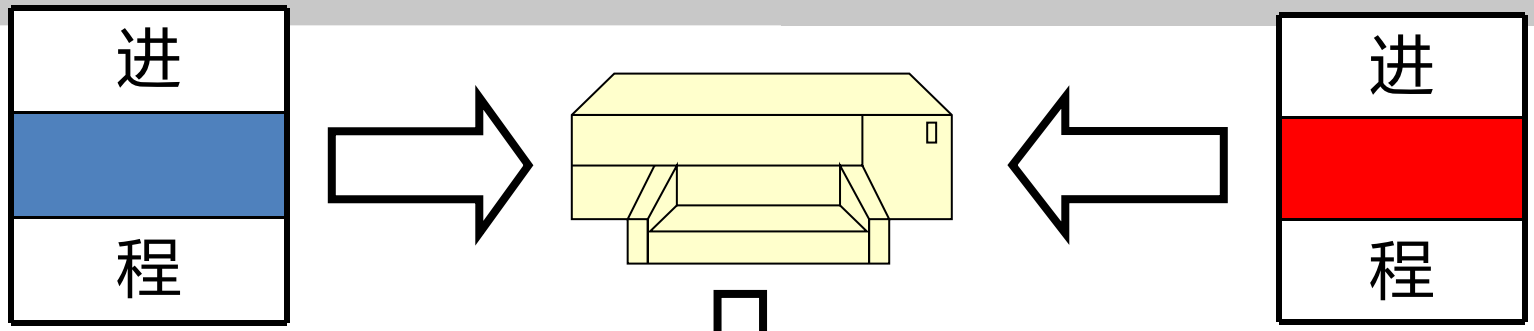
R/W文件

变量N



队列

2.4.3 临界资源 critical resource



互斥性/排它性/独占性

二个进程串行执行 → 降低并发性

关中断执行

实现的困难

- 时间片
- 实时系统及时性
- 高优先权的抢占
- 中断时间的太长
- 多CPU

系统中资源的共享程度

互斥：

指多个进程**不能同时使用同一个资源**。是正确使用资源的基本要求。

死锁：

指多个进程**互不相让**，都得不到足够的资源。

饥饿：

指一些进程**一直得不到资源**或得到资源的概率很小。

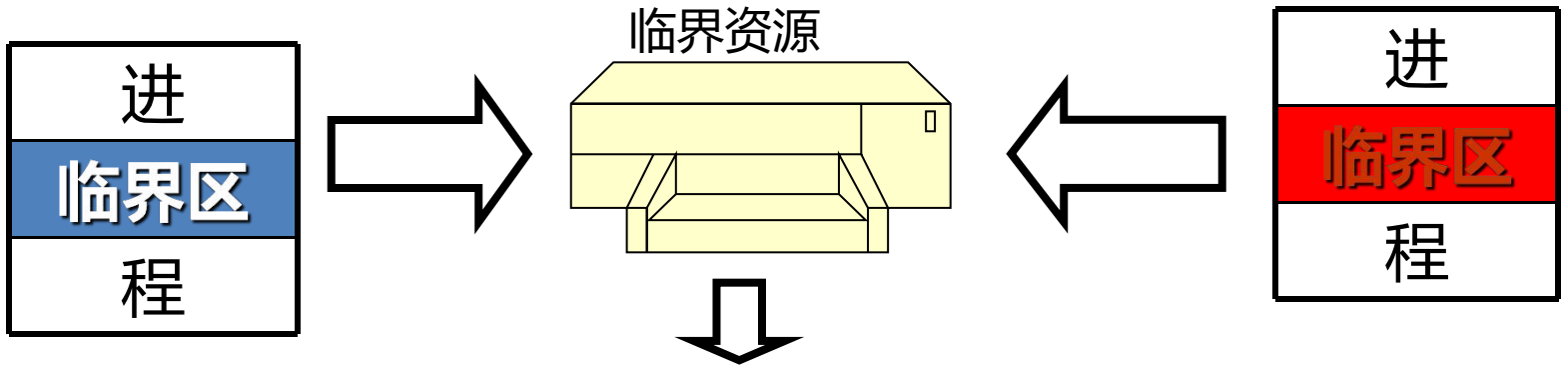
18

进程的制约关系

相互感知的程度	交互关系	对其他进程的影响	潜在的控制问题
相互不感知 (完全不了解其它进程的存在)	竞争 (competition)	对其他进程的结果无影响	互斥、死锁、饥饿
间接感知 (双方都与第三方交互，如共享资源)	通过共享进行协作	一个进程的结果间接依赖于从其他进程获得的信息	互斥、死锁、饥饿
直接感知 (双方直接交互，如通信)	通过通信进行协作	一个进程的结果直接依赖于从其他进程获得的信息	死锁、饥饿

2.4.4临界区

临界区: 每个进程中访问临界资源的代码段就是临界区



互斥性/排它性/独占性

访问相同临界资源的进程进入临界区时才互斥

二进程串行执行

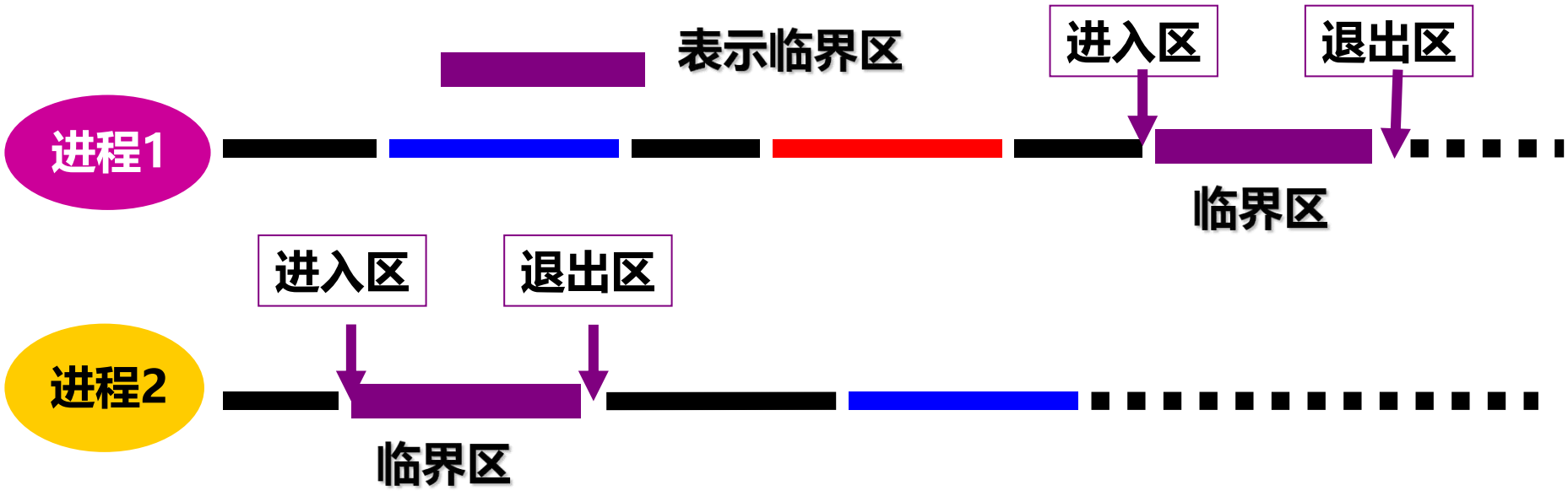
临界区串行执行或互斥进入

临界区执行中可能被中断

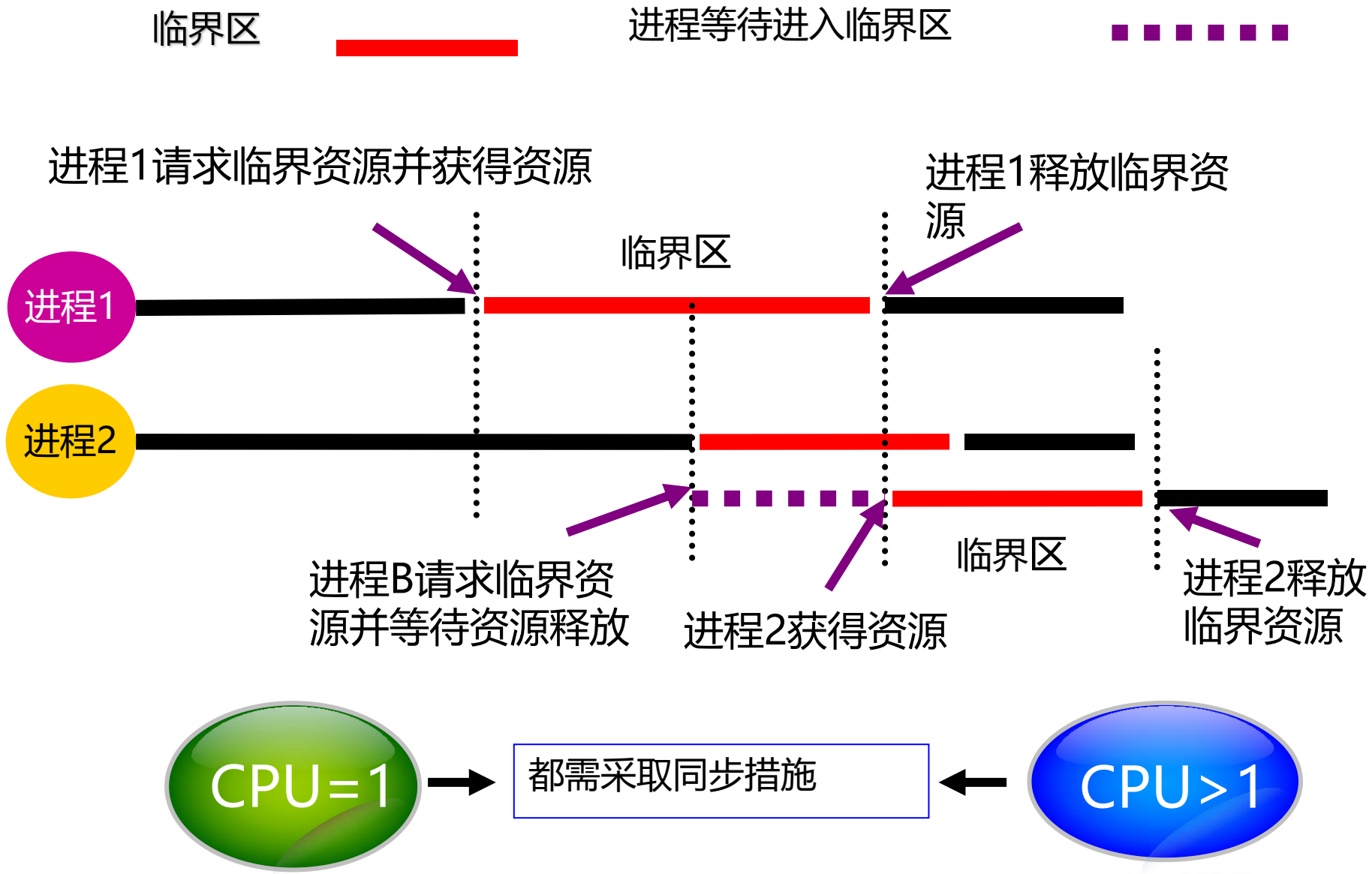
临界区的访问过程

- **进入区(entry section)**
  - 检查当前进程可否进入临界区的一段代码。
    如果当前进程可以进入临界区，通常设置相应“正在访问临界区”标志，防止其他进程同时进入临界区。
- **临界区(critical section)**
  - 进程中访问临界资源的一段代码。
- **退出区(exit section)**
  - 用于将“正在访问临界区”的进程的标志清除。
- **剩余区(remainder section)**
  - 代码中的其余部分。

临界区的访问过程



临界区的访问过程


临界区
进程等待进入临界区
进程1请求临界资源并获得资源
进程1释放临界资源
临界区
进程1
进程2
进程B请求临界资源并等待资源释放
进程2获得资源
临界区
进程2释放临界资源
CPU=1
都需采取同步措施
CPU>1

临界区与临界资源

一个访问临界资源的循环进程的描述

```
While(TRUE){
    进入区
    临界区
    退出区
    剩余区
}
```

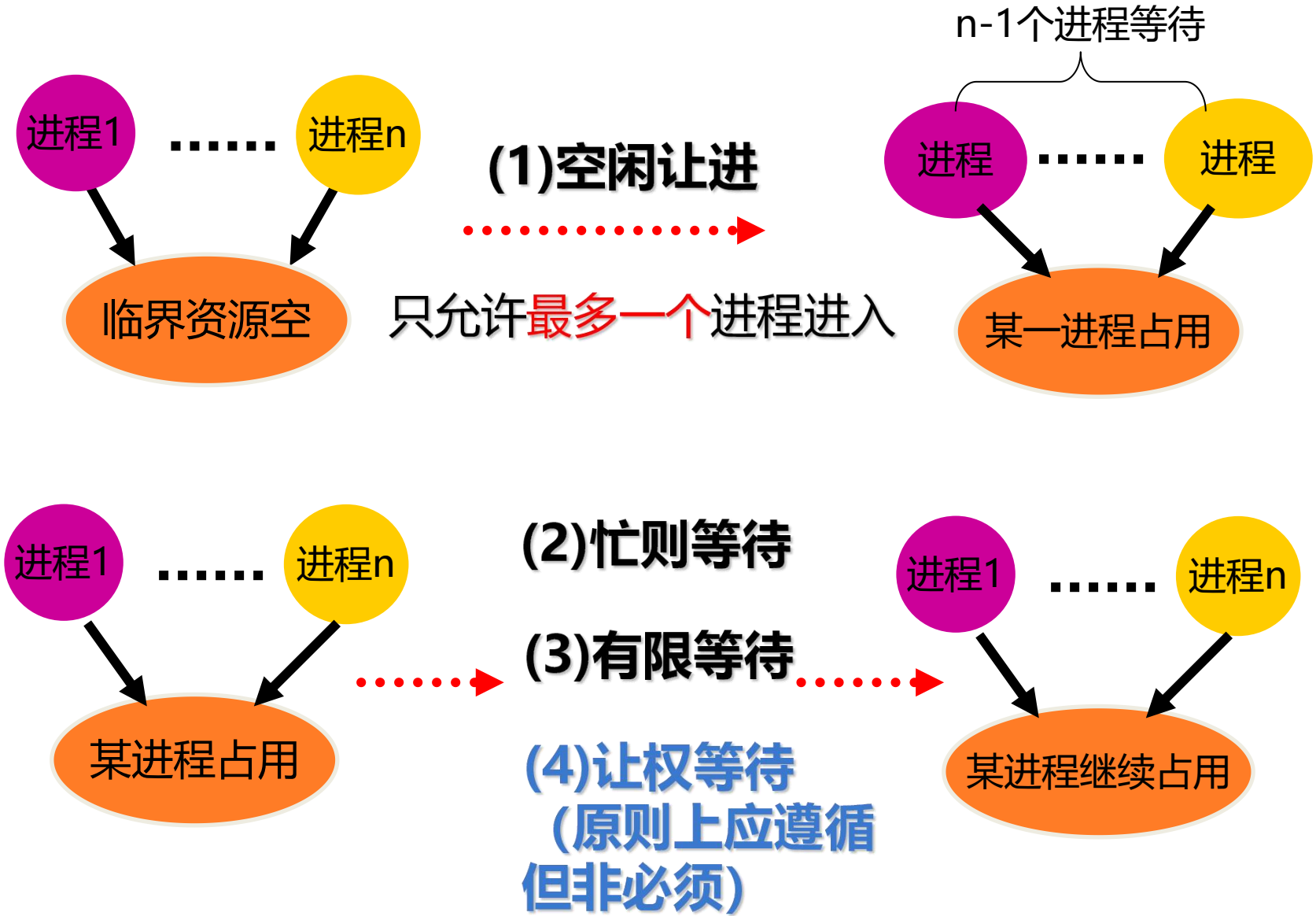
附加部分 → 进入区

原程序体 → 临界区

附加部分 → 退出区

进入区	通常是测试语句或判断语句(while,if等)
临界区	**critical section**
退出区	**临界资源访问结束后的标志,**标明临界资源可用,等待进程可进入
剩余区	**remainder section**

同步机制应遵循的四条准则



小结

- 同步、互斥
  - 进程同步 (直接制约)
    - 并发性带来了异步性，有时需要通过进程同步解决这种异步问题。
    - 有的进程之间需要相互配合地完成工作，各进程的工作推进需要遵循一定的先后顺序。
  - 进程互斥 (间接制约)
    - 对临界资源的访问，需要互斥的进行。即同一时间段内只能允许一个进程访问该资源
    - 四个部分
      - 进入区：检查是否可进入临界区，若可进入，需要"上锁"
      - 临界区：访问临界资源的那段代码
      - 退出区：负责"解锁"
      - 剩余区：其余代码部分
    - 需要遵循的原则
      - 空闲让进：临界区空闲时，应允许一个进程访问
      - 忙则等待：临界区正在被访问时，其他试图访问的进程需要等待
      - 有限等待：要在有限时间内进入临界区，保证不会饥饿
      - 让权等待：进不了临界区的进程，要释放处理机，防止忙等

小结

注意：

- 临界区是进程中访问临界资源的代码段。
- 进入区和退出区是负责实现互斥的代码段。
- 临界区也可称为"临界段"。

2.4.5进程同步和互斥的解决方法

软件同步机制

01

- 使用编程方法解决临界区问题
- 有难度、具有局限性，现在很少采用

硬件同步机制

02

- 使用特殊的硬件指令，可有效实现进程互斥

信号量机制

03

- 一种有效的进程同步机制，已被广泛应用

管程机制

04

- 新的进程同步机制

进程互斥的软件方法

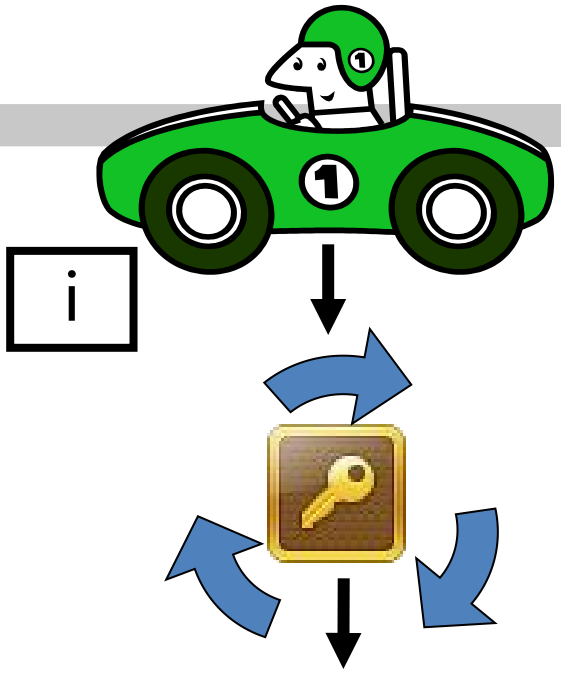
- 通过平等协商方式实现进程互斥的最初方法是软件方法。

- **软件方法基本思路**：

  在**进入区**检查和设置一些标志，如果已有进程在临界区，则在进入区通过循环检查进行等待；在**退出区**修改标志。

- 其中的**主要问题**：

  设置什么标志和**如何检查标志**。



进入区

退出区

```
While(1){
   while (turn!=i );
      critical section;    /*临界区（开车）*/
   turn=j;
      remainder section;  /* 剩余区*/
   }
```

进程互斥的软件方法

算法1：单标志

- 有两个进程Pi, Pj，

	Pi		Pj
1	while (turn !=i);	1	while(turn !=j);
2	临界区；	2	临界区；
3	turn = j;	3	turn = i;
4	剩余区；	4	剩余区；

- 设立一个公用整型变量 turn：

  描述允许进入临界区的进程标识。

进程互斥的软件方法

- 在进入区循环检查是否允许本进程进入：turn为i时，进程Pi可进入；
- 在退出区修改允许进入进程标识：

进程Pi退出时，改turn为进程Pj的标识j；

- 缺点：
  - 强制轮流进入临界区。
  - 容易造成资源利用不充分。

（在Pi出让临界区之后，Pj使用临界区之前，Pi不能再次使用临界区。）

问题？同一进程不能连续的进入临界区.违背空闲让进的原则

进程互斥的软件方法

算法2：双标志、先检查后修改（先人后己）

	Pi		Pj
1	while(flag[j]);	1	while(flag[i]);
2	flag[i] = true;	2	flag[j] = true;
3	临界区；	3	临界区；
4	flag[i] = false;	4	flag[j] = false;
5	剩余区；	5	剩余区；

- 设立一个标志数组flag[]： 描述是否有进程在临界区，

true: 进程正在临界区； false:进程未进入临界区，**初值均为FALSE**

进程互斥的软件方法

算法2：双标志、先检查后修改（先人后己）

- 先检查，后修改：

  在进入区检查另一个进程是否在临界区，不在时修改本进程在临界区的标志；

- 在退出区修改本进程在临界区的标志；

  - 优点：
    - 不用交替进入，可连续使用。

进程互斥的软件方法

- 缺点：

  Pi和Pj可能同时进入临界区。

  按下面序列执行时，会同时进入：

  “Pi<a> Pj<a> Pi<b> Pj<b>”。

即在检查对方flag之后和切换自己flag之前有一段时间，结果双方都检查通过。

问题出在检查和修改操作不能连续进行。

问题？ 二个进程都看到对方没进临界区而决定进去时则会发生冲突,互不相让.违背忙则等待的原则.

进程互斥的软件方法

算法3：双标志、先修改后检查（先斩后奏）

3	Pi		Pj
1	flag[i] = true;	1	flag[j] = true;
2	while(flag[j]);	2	while(flag[i]);
3	临界区；	3	临界区；
4	flag[i] = false;	4	flag[j] = false;
5	剩余区；	5	剩余区；

- 类似于算法2，与算法2的区别在于：先修改后检查；flag[]表是否想进入临界区。

进程互斥的软件方法

- 优点：
  - 可防止两个进程同时进入临界区。
- 缺点：Pi和Pj可能都进入不了临界区。

按下面序列执行时，会都进不了临界区：“Pi<a> Pj<a> Pi<b> Pj<b>”

即在切换自己flag之后和检查对方flag之前有一段时间，结果都切换flag，结果都检查不通过。

问题？ 二个进程首先表示自己要进入临界区,再去判断另一进程是否会进入临界区,结果是都认为对方会进入临界区而谦让,都不进去.违背空闲让进的原则.

进程互斥的软件方法

算法4:先修改、后检查、后修改者等待（Peterson算法）

设置一个数组flag[i/j]=true/flase=进入/未进入临界区

一个变量turn=i/j=表示允许谁进入;

4	Pi		Pj
1	flag[i] = true;	1	flag[j] = true;
2	turn = j;	2	turn = i;
3	while(flag[j]&&turn==j);	3	while(flag[i]&&turn==i);
4	临界区；	4	临界区；
5	flag[i] = false;	5	flag[j] = false;
6	剩余区；	6	剩余区；

- **结合算法1和算法3，是正确的算法。turn描述标志修改的先后，flag[]表示是否想进入临界区。**

进程互斥的软件方法

- 在进入区先修改后检查，并检查并发修改的先后。
- 实现了：空闲则入、忙则等待。

  - 检查对方flag:

    如果对方不想进入临界区，则自己进入 - “空闲则入”。
  - 否则再检查turn：

    turn中保存的是较晚的一次赋值。则较晚的进程等待，较早的进程进入 - 先到先入，后到等待。

满足三个需求；解决了两个进程的临界区问题。

进程互斥的软件方法

▲ 软件实现方法中：

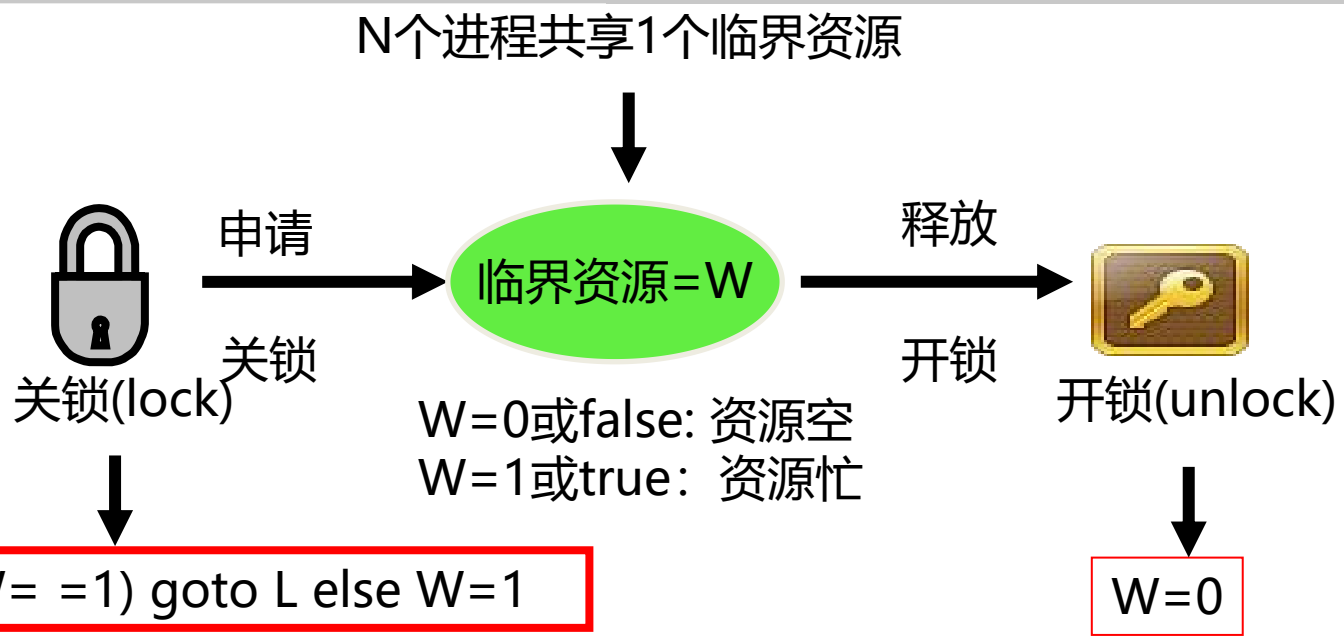
两个和三个以上进程间的互斥的进入区要区别对待。

不适合进程较多的进程间的互斥。

现在已很少单独采用软件方法。在平等协商时利用某些硬件指令来实现进程互斥。

最主要的问题：修改和检查标志不能作为一个整体被执行。

进程互斥的硬件方法




```
Lock(w);                    /* 进入区(关锁) */
    critical section;       /*临界区*/
unlock(w);                  /*退出区（开锁）*/
    remainder section;      /* 剩余区*/
```

- 将标志看作一个锁，"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，当锁未开时，则必须等待，直至锁被打开。当锁打开时，则应立即把其锁上，以阻止其他进程进入临界区。
- 测试和关锁操作必须是连续的，不允许分开进行。

进程互斥的硬件方法

▲ **硬件方法基本思路：**

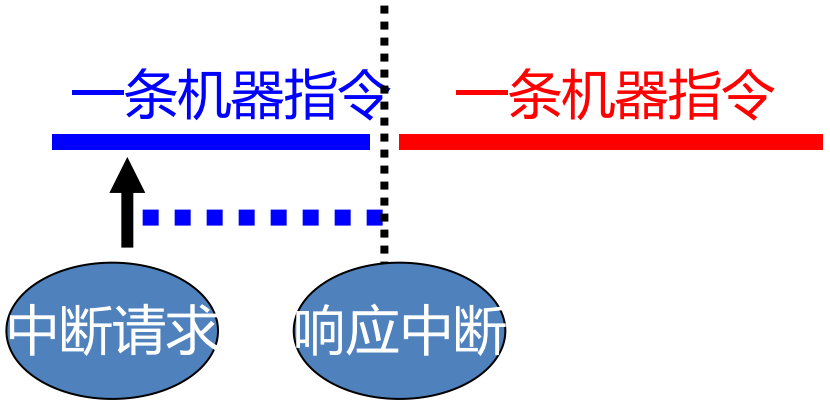
用一条指令完成读和写两种操作，因而保证读操作与写操作不被打断。

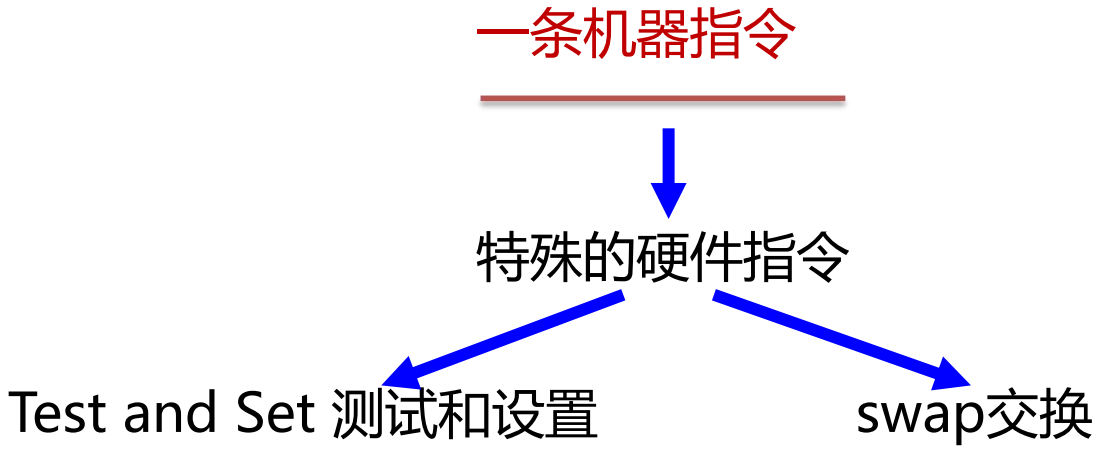
▲ **分类：**

依据所采用的指令不同，分为两种：

1.TS 指令

2.Swap 指令





进程互斥的硬件方法 中断屏蔽方法

利用"开/关中断指令"实现（与原语的实现思想相同，即在某进程开始访问临界区到结束访问为止都不允许被中断，也就不能发生进程切换，因此也不可能发生两个同时访问临界区的情况）

```
...
关中断;
临界区;
开中断;
...
```

关中断后即不允许当前进程被中断，也必然不会发生进程切换

直到当前进程访问完临界区，再执行开中断指令，才有可能有别的进程上处理机并访问临界区

优点：简单、高效

缺点：不适用于多处理机；只适用于操作系统内核进程，不适用于用户进程（因为开/关中断指令只能运行在内核态，这组指令如果能让用户随意使用会很危险）

进程互斥的硬件方法 TestAndSet指令

简称TS指令，也有地方称为 TestAndSetLock指令，或TSL指令

TSL指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用C语言描述的逻辑

```
//布尔型共享变量 lock 表示当前临界区是否被加锁
//true 表示已加锁, false 表示未加锁
bool TestAndSet (bool *lock){
  bool old;
  old = *lock;  //old用来存放lock 原来的值
  *lock = true; //无论之前是否已加锁, 都将lock设为true
  return old;   //返回lock原来的值
}
```

```
//以下是使用 TSL 指令实现互斥的算法逻辑
while (TestAndSet (&lock)); //"上锁"并"检查"
临界区代码段...
lock = false;       //"解锁"
剩余区代码段...
```

- 若刚开始 lock是 false，则 TSL返回的 old值为false，while 循环条件不满足，直接跳过循环，进入临界区。若刚开始 lock是 true，则执行TLS后 old 返回的值为 true，while循环条件满足，会一直循环，直到当前访问临界区的进程在退出区进行"解锁"。
- 相比软件实现方法，TSL指令把"上锁"和"检查"操作用硬件的方式变成了一气呵成的原子操作。
- 优点：实现简单，无需像软件实现方法那样严格检查是否会有逻辑漏洞；**适用于多处理机环境**。
- 缺点：**不满足"让权等待"原则**，暂时无法进入临界区的进程会占用CPU并循环执行TSL指令，从而导致**"忙等"**。

进程互斥的硬件方法 Swap指令

有的地方也叫 Exchange 指令，或简称 XCHG指令。

Swap指令是用硬件实现的，执行过程不允许被中断，只能一气呵成。以下是用C语言描述的逻辑

```
//Swap 指令的作用是交换两个变量的值
Swap (bool *a, bool *b) {
  bool temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

```
//以下是用 Swap 指令实现互斥的算法逻辑
//lock 表示当前临界区是否被加锁
bool old = true;
while (old == true)
  Swap (&lock, &old);
临界区代码段...
lock = false;
剩余区代码段...
```

- 逻辑上来看 Swap 和TSL并无太大区别，都是先记录下此时临界区是否已经被上锁（记录在 old变量上），再将上锁标记lock设置为 true，最后检查 old，如果 old 为false 则说明之前没有别的进程对临界区上锁，则可跳出循环，进入临界区。
- 优点：实现简单，无需像软件实现方法那样严格检查是否会有逻辑漏洞；**适用于多处理机环境**
- 缺点：**不满足"让权等待"原则**，暂时无法进入临界区的进程会占用CPU并循环执行TSL指令，从而导致"**忙等**"。

进程互斥的硬件方法

- **硬件方法的优点：**
  - **适用范围广：**

    适用于任意数目的进程，在单处理器或多处理器上完全相同。
  - **简单：**

    标志设置简单，含义明确，容易验证其正确性。
  - **支持多个临界区：**

    一个进程内 存在多个临界区时，只需为每个临界区设立一个布尔变量。

- **硬件方法的缺点：**
  - **不能实现“让权等待”。**

    进程在等待进入临界区时，要耗费处理机时间。
  - **可能“饥饿“。**

    进入临界区的进程是从等待进程中随机选择有的进程可能一直没有被选中。

测验

- 两进程互斥算法（双标志、后检查）存在什么问题?

A. 强制轮流进入临界区

B. 可能同时进入临界区

C. 可能都进入不了临界区

3	Pi		Pj
1	flag[i] = true;	1	flag[j] = true;
2	while(flag[j]);	2	while(flag[i]);
3	临界区；	3	临界区；
4	flag[i] = false;	4	flag[j] = false;
5	剩余区；	5	剩余区；

测验

- 用TS指令实现的临界区访问控制无法实现哪些控制准则?

A.空闲则入

B.忙则等待

C.有限等待

D.让权等待

思考？ 之前学习的这些进程互斥的解决方案分别存在哪些问题?

- 进程互斥的四种软件实现方式（单标志法、双标志先检查、双标志后检查、Peterson算法）
- 进程互斥的三种硬件实现方式（中断屏蔽方法、TS指令、Swap/CHG指令）

1. 在双标志先检查法中，进入区的"检查"、"上锁"操作无法一气呵成，从而导致了两个进程有可能同时进入临界区的问题;
2. 所有的解决方案都无法实现"让权等待"。

【1】单标志法：

- 算法思想：两个进程在访问完临界区后会把使用临界区的权限转交给另一个进程。也就是说每个进程进入临界区的权限只能被另一个进程赋予

```
int turn = 0; //turn 表示当前允许进入临界区的进程号

P0 进程:                              P1进程:
while (turn != 0);        ①           while (turn != 1);        ⑤ //进入区
critical section;         ②           critical section;         ⑥ //临界区
turn = 1;                 ③           turn = 0;                 ⑦ //退出区
remainder section;        ④           remainder section;        ⑧ //剩余区
```

- turn 的初值为0，即刚开始只允许0号进程进入临界区。
- 若P1先上处理机运行，则会一直卡在⑤。直到P1的时间片用完，发生调度，切换 P0上处理机运行。代码①不会卡住P0，P0可以正常访问临界区，在P0访问临界区期间即时切换回P1，P1依然会卡在⑤。只有P0在退出区将turn 改为1后，P1才能进入临界区。
- **因此，该算法可以实现"同一时刻最多只允许一个进程访问临界区"**

缺点：turn 表示当前允许进入临界区的进程号，而只有当前允许进入临界区的进程在访问了临界区之后，才会修改turn的值。也就是说，对于临界区的访问，

一定是P0->P1->P0->P1……这样轮流访问。这种必须“轮流访问”带来的问题是，如果此时允许进入临界区的进程是P0，而P0一直不访问临界区，那么虽然临界区空闲，但是并不允许P1访问。

因此，但标志法存在的主要问题是：违背了“空闲让进”原则。

【2】双标志先检查法：

- 算法思想：设置一个布尔型数组 flag[]，数组中各个元素用来**标记各进程想进入临界区的意愿**，比如"flag[0]=ture"意味着0号进程 P0现在想要进入临界区。每个进程在**进入临界区之前先检查**当前有没有别的进程想进入临界区，如果没有，则把自身对应的标志 flag[i] 设为true，之**后开始访问临界区**。

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区

P0 进程:                        P1 进程:
while (flag[1]);    ①           while (flag[0]);    ⑤   //如果此时 P0 想进入临界区，P1 就一直循环等待
flag[0] = true;     ②           flag[1] = true;     ⑥   //标记为 P1 进程想要进入临界区
critical section;   ③           critical section;   ⑦   //访问临界区
flag[0] = false;    ④           flag[1] = false;    ⑧   //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;              remainder section;
```

若按照 ①⑤②⑥③⑦...的顺序执行，P0和P1将会同时访问临界区。

因此，双标志先检查法的主要问题是：违反"忙则等待"原则。

【3】双标志后检查法：

- **算法思想:双标志先检查法的改版。前一个算法的问题是先"检查"后"上锁"，但是这两个操作又无法一气呵成，因此导致了两个进程同时进入临界区的问题。因此，人们又想到先"上锁"后"检查"的方法，来避免上述问题。**

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区

P0 进程:                        P1 进程:
flag[0] = true;     ①           flag[1] = true;     ⑤   //标记为 P1 进程想要进入临界区
while (flag[1]);    ②           while (flag[0]);    ⑥   //如果 P0 也想进入临界区，则 P1 循环等待
critical section;   ③           critical section;   ⑦   //访问临界区
flag[0] = false;    ④           flag[1] = false;    ⑧   //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;              remainder section;
```

若按照①⑤②⑥..的顺序执行，P0和 P1将都无法进入临界区

因此，双标志后检查法虽然解决了"忙则等待"的问题，

但是又违背了"空闲让进"和"有限等待"原则，

会因各进程都长期无法访问临界资源而产生"饥饿"现象。

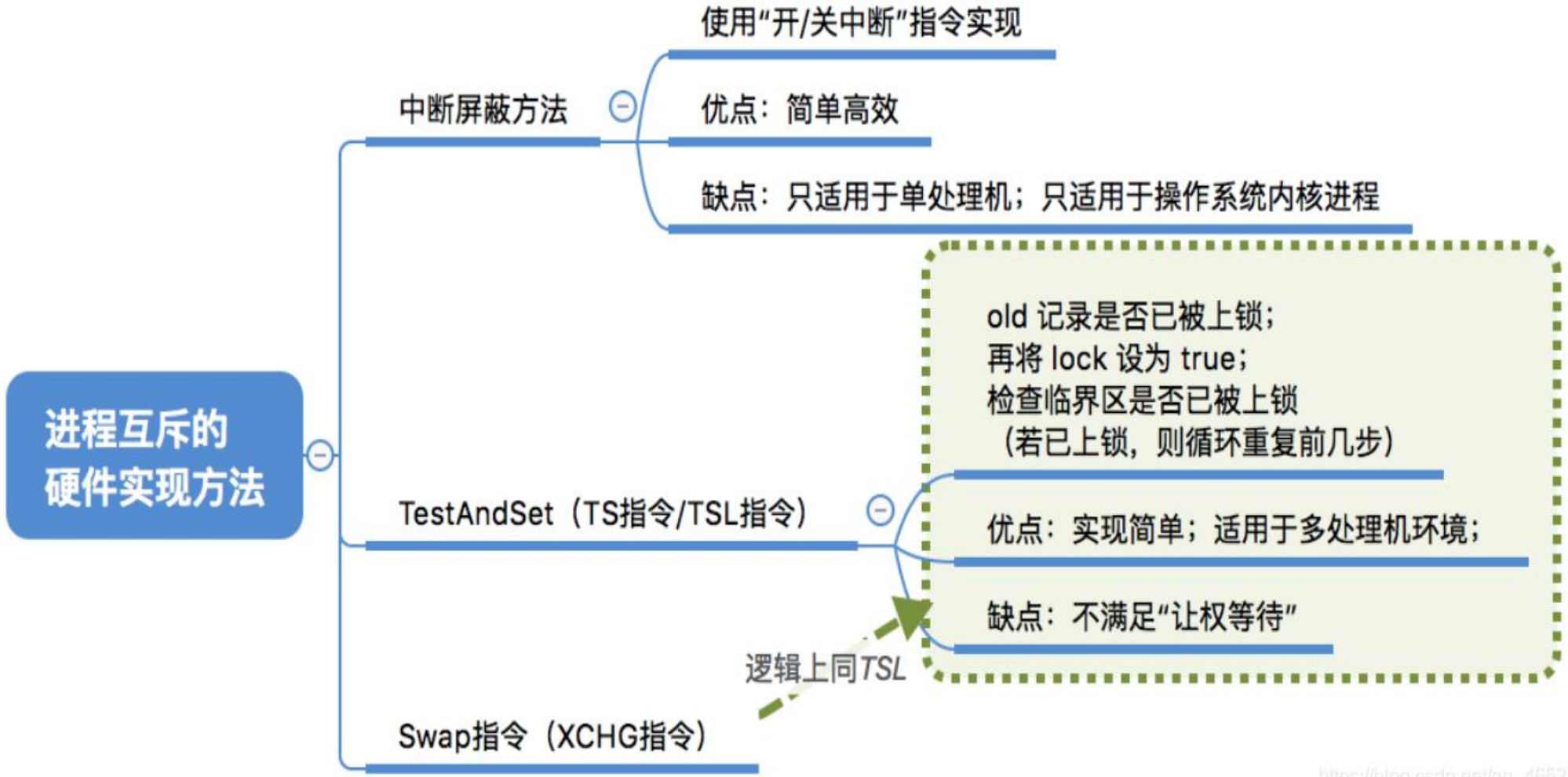
【4】Peterson算法

- **算法思想：双标志后检查法中，两个进程都争着想进入临界区，但是谁也不让谁，最后谁都无法进入临界区。GaryL.Peterson 想到了一种方法，如果双方都争着想进入临界区，那可以让进程尝试"孔融让梨"，主动让对方先使用临界区。**

```
bool flag[2];          //表示进入临界区意愿的数组，初始值都是false
int turn = 0;          //turn 表示优先让哪个进程进入临界区
P0 进程:
flag[0] = true;                ①
turn = 1;                      ②   进入区
while (flag[1] && turn==1);    ③
critical section;              ④
flag[0] = false;               ⑤
remainder section;
P1 进程:
flag[1] = true;                ⑥  //表示自己想进入临界区
turn = 0;                      ⑦  //可以优先让对方进入临界区
while (flag[0] && turn==0);    ⑧  //对方想进，且最后一次是自己“让梨”，那自己就循环等待
critical section;              ⑨
flag[1] = false;               ⑩  //访问完临界区，表示自己已经不想访问临界区了
remainder section;
```

两种双标志法的问题都是由于进入区的几个操作不能一气呵成导致的。我们可以推理验证在Peterson 算法中，两个进程进入区中的各个操作按不同的顺序穿插执行会发生什么情况：
①②③⑥⑦⑧...
①⑥②③...

【5】硬件实现方法



https://blog.csdn.net/qq_46527915

信号量（semaphore）机制

前面的互斥算法都存在问题，它们是平等进程间的一种协商机制。

有时需要一个地位高于进程的管理者来解决公有资源的使用问题。

信号量就是OS提供的管理公有资源的有效手段。

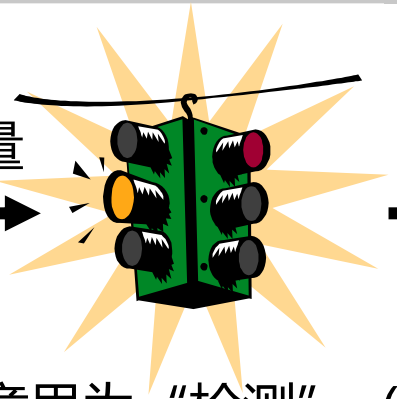
信号量代表可用资源实体的数量。

一个进程在某一特殊点上被迫停止执行直到接收到一个对应的特殊变量值，这种特殊变量就是信号量(semaphore)，复杂的进程合作需求都可以通过适当的信号结构得到满足。

迪科斯彻(1930-2002)

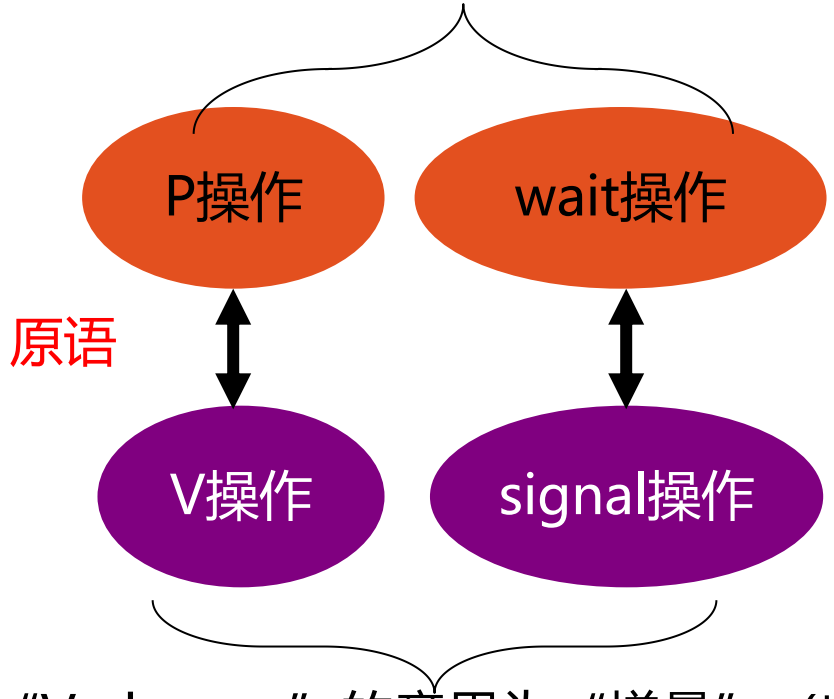
- 艾兹格·W·迪科斯彻(Edsger Wybe Dijkstra)
  - 信号量和PV原语发明者
  - 解决了“**哲学家就餐**”问题
  - 最短路径算法(SPF)和**银行家算法**的创造者
  - 结构程序设计之父
  - **THE操作系统**设计者和开发者
- 与D. E. Knuth并称为这个时代最伟大的计算机科学家

信号量（semaphore）机制

1965年 E.W.Dijkstra 信号量  不同形式

1)整型信号量
2)记录型信号量
3)AND型信号量
4)信号量集。

荷兰语“Proberen” 的意思为“检测” （判断）



“Verhogen”的意思为“增量” （归还）

- **信号量S**
- **提供两个不可分割的[原子操作]访问信号量**

1. 整型信号量

信号量机制——整型信号量

用一个整数型的变量作为信号量，用来表示系统中某种资源的数量。

Eg：某计算机系统中有一台打印机...

```
int S = 1;  //初始化整型信号量s，表示当前系统中可用的打印机资源数

void wait (int S) {    //wait 原语，相当于"进入区"
  while (S <= 0);      //如果资源数不够，就一直循环等待
  S=S-1;               //如果资源数够，则占用一个资源
}

void signal (int S) { //signal 原语，相当于"退出区"
  S=S+1;          //使用完资源后，在退出区释放资源
}
```

与普通整数变量的区别：
对信号量的操作只有三种，
即 初始化、P操作、V操作

"检查"和"上锁"一气呵成，
避免了并发、异步导致的问题

存在的问题：不满足"让权等待"
原则，会发生"忙等"

- Wait(s)又称为P(S)
- Signal(s)又称为V(S)
- 缺点：进程忙等

```
进程P0:
...
wait(S);          //进入区，申请资源
使用打印机资源...  //临界区，访问资源
signal(S);        //退出区，释放资源
...
```

```
进程P1:
...
wait(S);
使用打印机资源...
signal(S);
...
```

.....

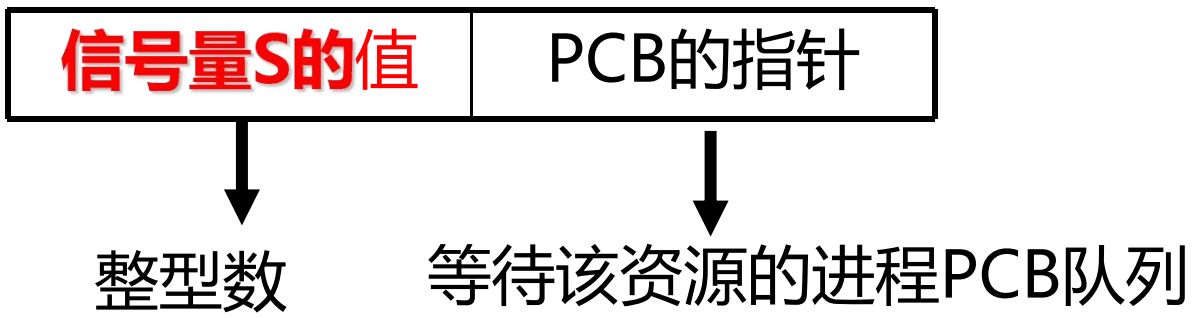
```
进程Pn:
...
wait(S);
使用打印机资源...
signal(S);
...
```

2. 记录型信号量： 去除忙等的信号量

每个信号量S除一个整数值S.value外，还有一个进程等待队列S.list，存放阻塞在该信号量的各个进程PCB

- 信号量只能通过初始化和两个标准的原语PV来访问，作为OS核心代码执行，不受进程调度的打断
- 初始化指定一个非负整数值，表示空闲资源总数（又称为"资源信号量"），若为非负值表示当前的空闲资源数，若为负值，其绝对值表示当前等待临界区的进程数

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



2. 记录型信号量

```
/*记录型信号量的定义*/
typedef struct {
  int value;              //剩余资源数
  struct process *L;      //等待队列
} semaphore;
```

注意两个原语中的value条件为什么不同？

```
/*某进程需要使用资源时，通过 wait 原语申请*/
void wait (semaphore S) {
  S.value--;
  if (S.value < 0 ) {
    block (S.L);
  }
}
```

如果剩余资源数不够，使用block原语使进程从运行态进入阻塞态，并把挂到信号量 S 的等待队列（即阻塞队列）中

```
/*进程使用完资源后，通过 signal 原语释放*/
void signal (semaphore S) {
  s.value++;
  if (S.value <= 0) {
    wakeup(S.L);
  }
}
```

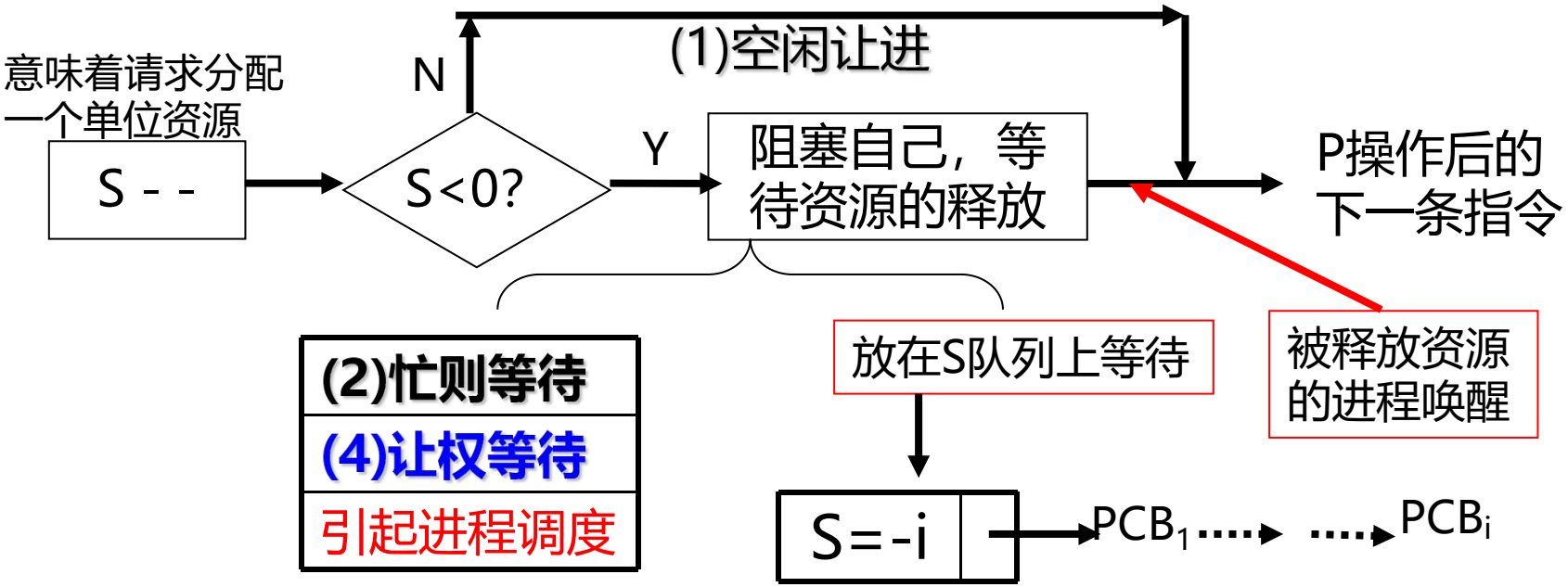
释放资源后，若还有别的进程在等待这种资源，则使用 wakeup 原语唤醒等待队列中的一个进程，该进程从阻塞态变为就绪态

2. 记录型信号量

P(S):申请一个资源		
S =S-1	>0	OK：能分配，还有资源
	= = 0	OK：能分配，刚好分完
	< 0	NO: 不能分配，没有资源

P(S)的定义

```
Void P(semaphore *S)
 { S->Value - -;
if (S->value<0 ) block(S->list);}
```

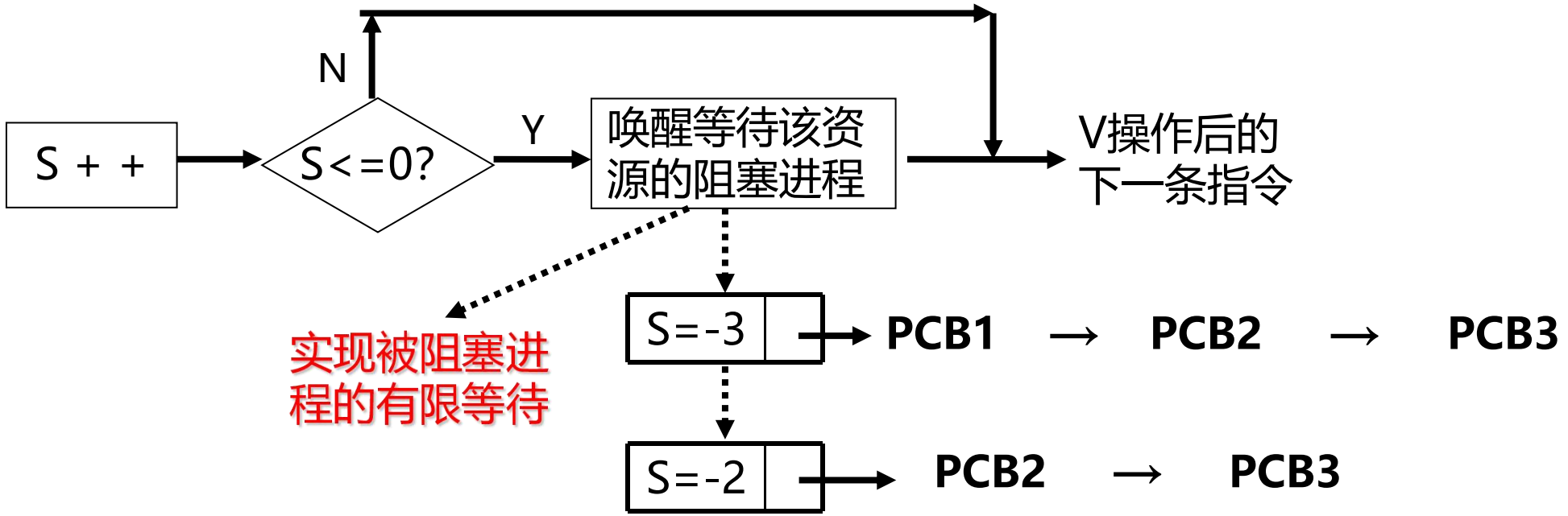


2. 记录型信号量

V(S):释放一个资源		
S =S+1	= =0	释放前有一个等待的进程
	< 0	释放前至少有二个等待的进程
	> 0	无进程等待该资源

V(S)的定义

```
V(semaphore *S)
{ S->value + +;
if (S->value<=0 ) wakeup(S->list); }
```



2. 记录型信号量

```
/*记录型信号量的定义*/
typedef struct {
   int value;              //剩余资源数
   Struct process *L;      //等待队列
} semaphore;

/*某进程需要使用资源时，通过 wait 原语申请*/
void wait (semaphore S) {
   S.value--;
   if (S.value < 0 ) {
      block (S.L);
   }
}

/*进程使用完资源后，通过 signal 原语释放*/
void signal (semaphore S) {
   s.value++;
   if (S.value <= 0) {
      wakeup(S.L);
   }
}
```

在考研题目中wait（S）、signal（S）也可以记为P（S）、V（S），这对原语可用于**实现系统资源的"申请"和"释放"**。

- ✓ **S.value的初值**表示系统中**某种资源的数目**。
- ✓ 对信号量S的**一次P操作**意味着进程**请求一个单位的该类资源**，因此需要执行S.value-，表示资源数减1，当S.value<0时表示该类资源已分配完毕，因此进程应**调用block 原语进行自我阻塞**（当前运行的进程从**运行态→阻塞态**），主动放弃处理机，并插入该类资源的等待队列S.L中。可见，该机制**遵循了"让权等待"原则**，不会出现"忙等"现象。
- ✓ 对信号量S的**一次V操作**意味着进程**释放一个单位的该类资源**，因此需要执行 S.value++，表示资源数加1，若加1后仍是S.value<=0，表示依然有进程在等待该类资源，因此应**调用 wakeup 原语唤醒等待队列中的第一个进程**（被唤醒进程从**阻塞态→就绪态**）。

信号量（semaphore）机制

定义结构体
```
type semaphore = record
value : integer;
*list : list of process
end;
var s: semaphore;
```

wait原子操作
```
wait(s):
s.value := s.value – 1;
if s.value < 0
    then begin
    block P;
    insert P into s.list;
end;
```

signal原子操作
```
signal(s):
s.value:= s.value + 1;
if s.value ≤ 0
then begin
wakeup the first P;
remove the P from s.list;
end;
```

```
program mutualexclusion;
const n=…; /* 进程数 */
var s: semaphore(:= 1); /* 定义信号量s，s.value初始化为1 */
procedure P(i:integer);
begin
   repeat
      wait(s);
      <临界区>;
      signal(s);
      <其余部分>
   forever
end;
```

```
begin /* 主程序 */
 parbegin
 P(1); P(2); … P(n)
 parend
end.
```

利用信号量实现互斥的通用模式

信号量（semaphore）机制

信号量S的初值			
>1	=1	=0	<0

信号量S上的正值，表示还有多少可用的资源，负值表示等待该资源上的进程数

如n个进程共享信号量S表示的资源	
初值S=1	**取值范围[1，1-n]**
初值S=m	**取值范围[m，m-n]**

3. AND型信号量

（1）多个进程共享两个或更多的资源：

A、B两个进程共享数据D和E，采用信号量的方法实现A、B进程的同步。

互斥使用数据D和E的信号量分别为Dmutex和Emutex。

Dmutex=1;　　Emutex=1;

进程A：

```
wait(Dmutex);
wait(Emutex);
访问数据D;
访问数据E;
signal(Dmutex);
signal(Emutex);
```

进程B：

```
wait(Emutex);
wait(Dmutex);
访问数据D;
访问数据E;
signal(Dmutex);
signal(Emutex);
```


66

3. AND型信号量

问题：A、B进程有可能进入死锁状态，导致A、B进程都无法向前推进。

（2）AND同步机制的基本思想：

将进程运行过程中所需的全部资源一次性全部分配给进程，待进程用完后再一起释放。只有有一个资源不能满足进程的要求，其他所有可能分配给它的资源也暂不分配。

主要思想：要么全分配，要么一个也不分配。

3. AND型信号量

Swait(S_1，S_2，…，S_n);
{
  if(S_1>=1 &&S_2>=1…S_n>=1)
        S_1--; S_2--; … S_n--;
  else
        将进程阻塞并加入到每个信号量链表中
}
Ssignal(S_1，S_2，…，S_n);
{
    S_1++; S_2++; … S_n++;
    将进程从信号量链表中删除并变为就绪状态
}

信号量（semaphore）机制

wait.signal操作讨论

1) 信号量的物理含义：

S>0，表示有S个资源可用。

S=0，表示无资源可用。

S<0，则|S|表S等待队列中的进程个数。

wait(S): 表示申请一个资源。

signal(S): 表示释放一个资源。

信号量的初值应该 〉= 0。

69

信号量的类型

信号量分为：互斥信号量和资源信号量。

- 互斥信号量用于申请或释放资源的使用权，常初始化为1，表示只允许一个进程访问临界资源，用于进程互斥。
- 资源信号量用于申请或归还资源，可以初始化为大于1的正整数，表示系统中某类资源的可用个数。

- wait操作用于申请资源（或使用权），进程执行wait原语时，可能会阻塞自己；
- signal操作用于释放资源（或归还资源使用权），进程执行signal原语时，有责任唤醒一个阻塞进程。

信号量的类型

信号量的意义

1. 互斥信号量：申请/释放使用权，常初始化为1

2. 资源信号量：申请/归还资源，资源信号量可以初始化为一个正整数（表示系统中某类资源的可用个数），

s.value的意义为：

- s.value≥0：表示还可执行wait(s)而不会阻塞的进程数（可用资源数）
- s.value<0：表示s.list队列中阻塞进程的个数（被阻塞进程数）

信号量的类型

当仅有两个并发进程共享临界资源时，互斥信号量仅能取值0、1、 - 1。其中，

- ✓ s.value=1,表示无进程进入临界区
- ✓ s.value=0，表示已有一个进程进入临界区
- ✓ s.value= - 1,则表示已有一进程正在等待进入临界区

当用s来实现n个进程的互斥时，s.value 的取值范围为1 ~ - （n-1）

小结

- 信号量机制
  - 整型信号量
    - 用一个整数型变量作为信号量，数值表示某种资源数
    - 整型信号量与普通整型变量的区别：对信号量只能执行 初始化、P、V 三种操作
    - 整型信号量存在的问题：不满足让权等待原则
  - 记录型信号量
    - S.value 表示某种资源数，S.L 指向等待该资源的队列
    - P 操作中，一定是先 S.value--，之后可能需要执行 block 原语
    - V 操作中，一定是先 S.value++，之后可能需要执行 wakeup 原语
    - 注意：要能够自己推断在什么条件下需要执行 block 或 wakeup
    - 可以用记录型信号量实现系统资源的“申请”和“释放”
    - 可以用记录型信号量实现进程互斥、进程同步

大题、小题超高频出题点

信号量的应用

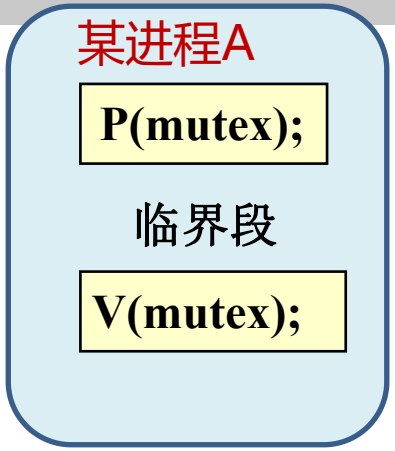
操作系统内核以系统调用形式提供wait和signal原语，

应用程序通过该**系统调用实现进程间的互斥**。

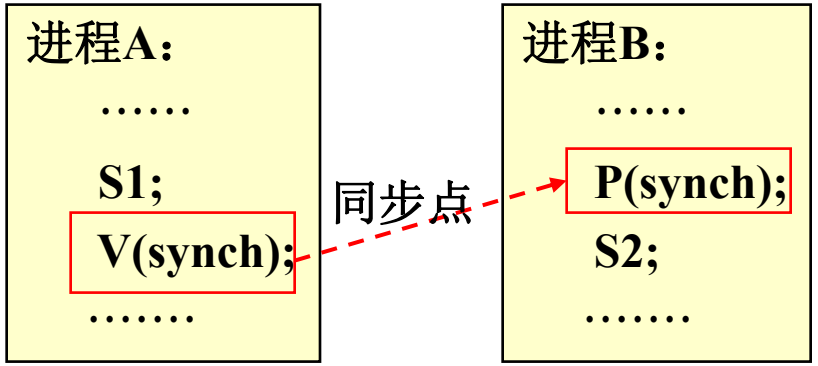
工程实践证明，利用信号量方法实现进程互斥是高效的，一直被广泛采用。

信号量的应用

- **实现进程互斥**
  - 使用信号量mutex（初值 = 1）



- **实现进程同步**
  - 进程A执行完S1语句后，进程B才能执行S2。
  - 使用信号量synch（初值 = 0）



- **实现进程的前驱关系（多层的同步关系）**

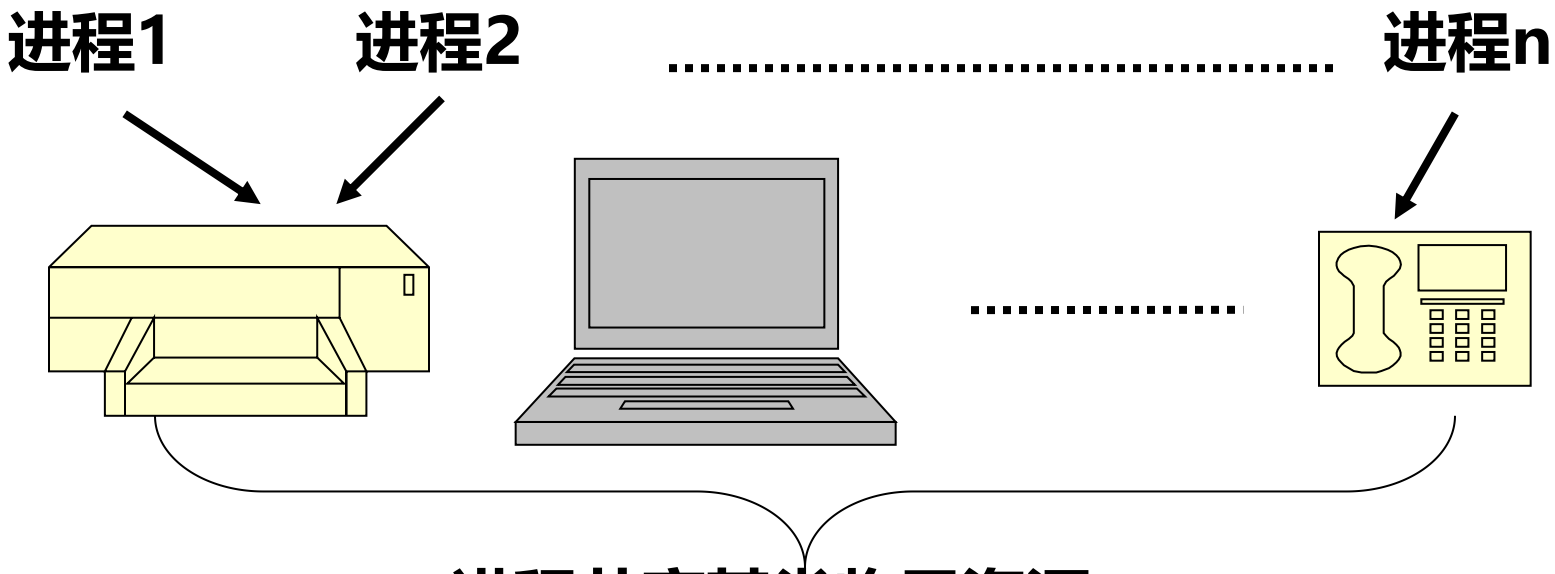
利用信号量实现进程互斥

- 为每个临界资源设置一个互斥信号量mutex(MUTual Exclusion)，其初值为1。
- 在每个进程中将临界区代码置于P(mutex)和V(mutex)原语之间。
- P、V原语不能次序错误、重复或遗漏。
- 必须成对使用P和V原语：

  遗漏P原语则不能保证互斥访问；

  遗漏V原语则不能在使用临界资源之后将其释放给其他等待的进程。

利用信号量实现进程互斥



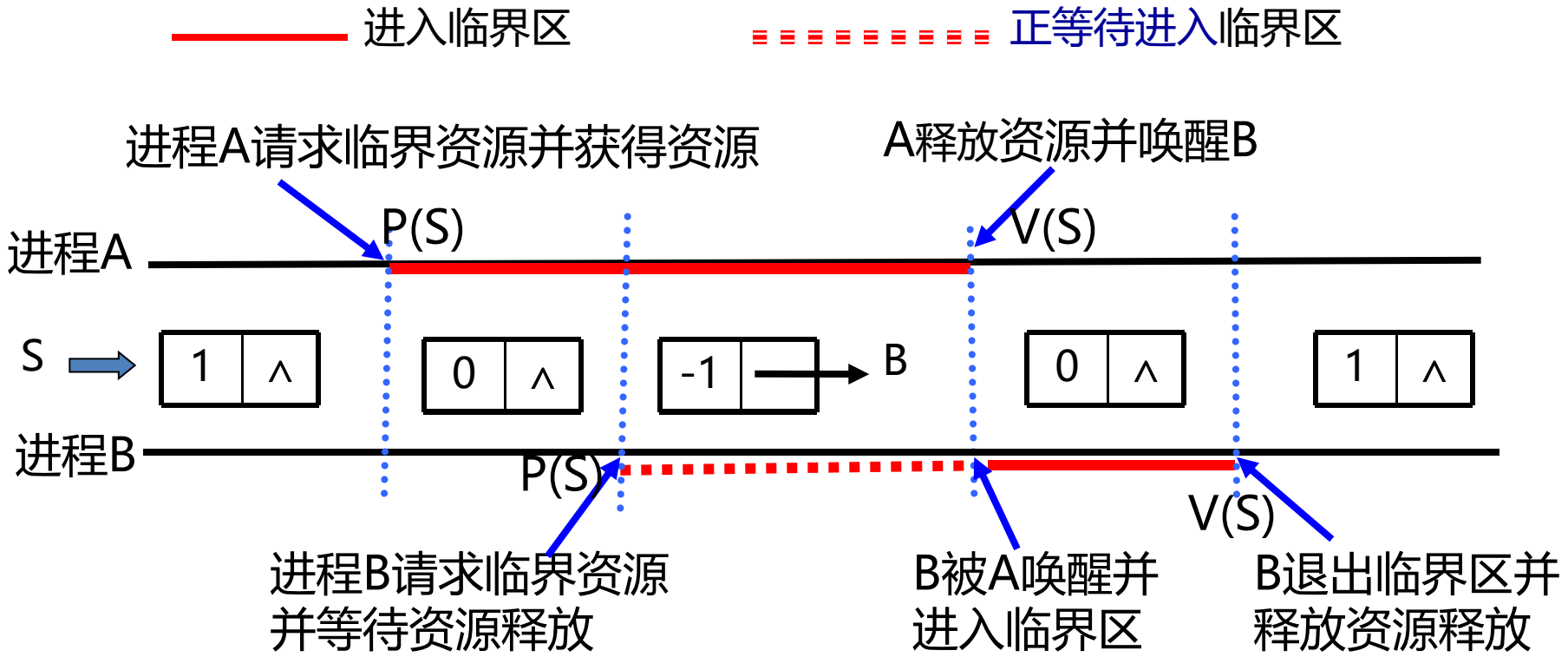
进程共享某类临界资源

信号量初值mutex为0,1,n?

互斥信号量P(mutex)和V(mutex)的成对

P(mutex)；cs_i(临界区)；V(mutex)

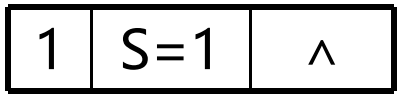
利用信号量实现进程互斥



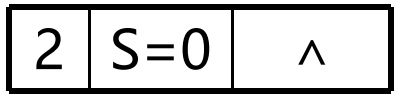
如果初值S=1，有二个进程，S的取值范围[1，-1]	
S==1	**表示无进程进入临界区**
S==0	**表示一进程进入临界区**
S==-1	**表示一进程在临界区，另一进程在等待进入临界区**

利用信号量实现进程互斥

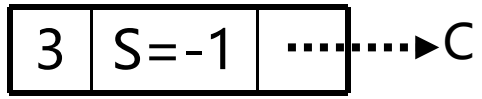
进程A	进程B	进程C
P(mutex);	P(mutex);	P(mutex);
CS1;	CS2;	CS3;
V(mutex);	V(mutex);	V(mutex);



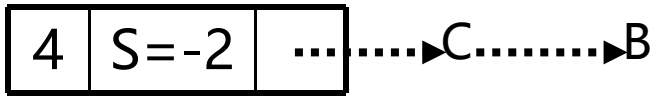
初始状态



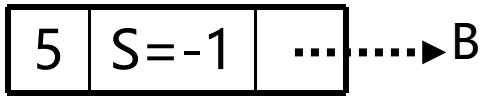
进程1已进入临界区



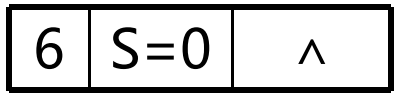
进程C申请进入临界区而阻塞



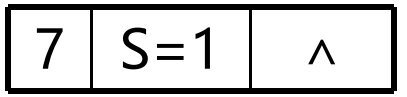
进程B申请进入临界区而阻塞



进程A退出临界区，并唤醒阻塞进程C



进程C退出临界区，并唤醒阻塞进程B



进程B退出临界区,资源空闲可用

利用信号量实现同步关系

```
P1(){
    代码1;
    代码2;
    代码3;
}

P2(){
    代码4;
    代码5;
    代码6;
}
```

P1、P2并发执行，由于存在异步性，因此二者交替推进的次序是不确定的。

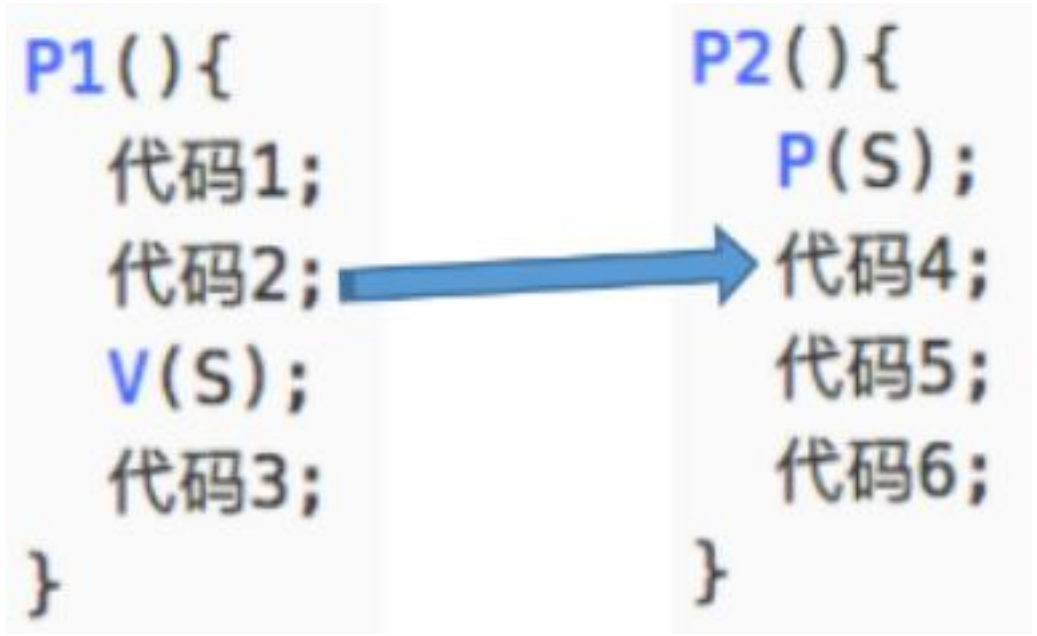
若P2的"代码4"要基于 P1的"代码1"和"代码2"的运行结果才能执行，那么就必须保证"代码4"一定是在"代码2"之后才会执行。

这就是**进程同步**问题，**让本来异步并发的进程互相配合，有序推进。**

利用信号量实现同步关系

/*信号量机制实现同步k/

semaphore S=0;//初始化同步信号量，初始值为0



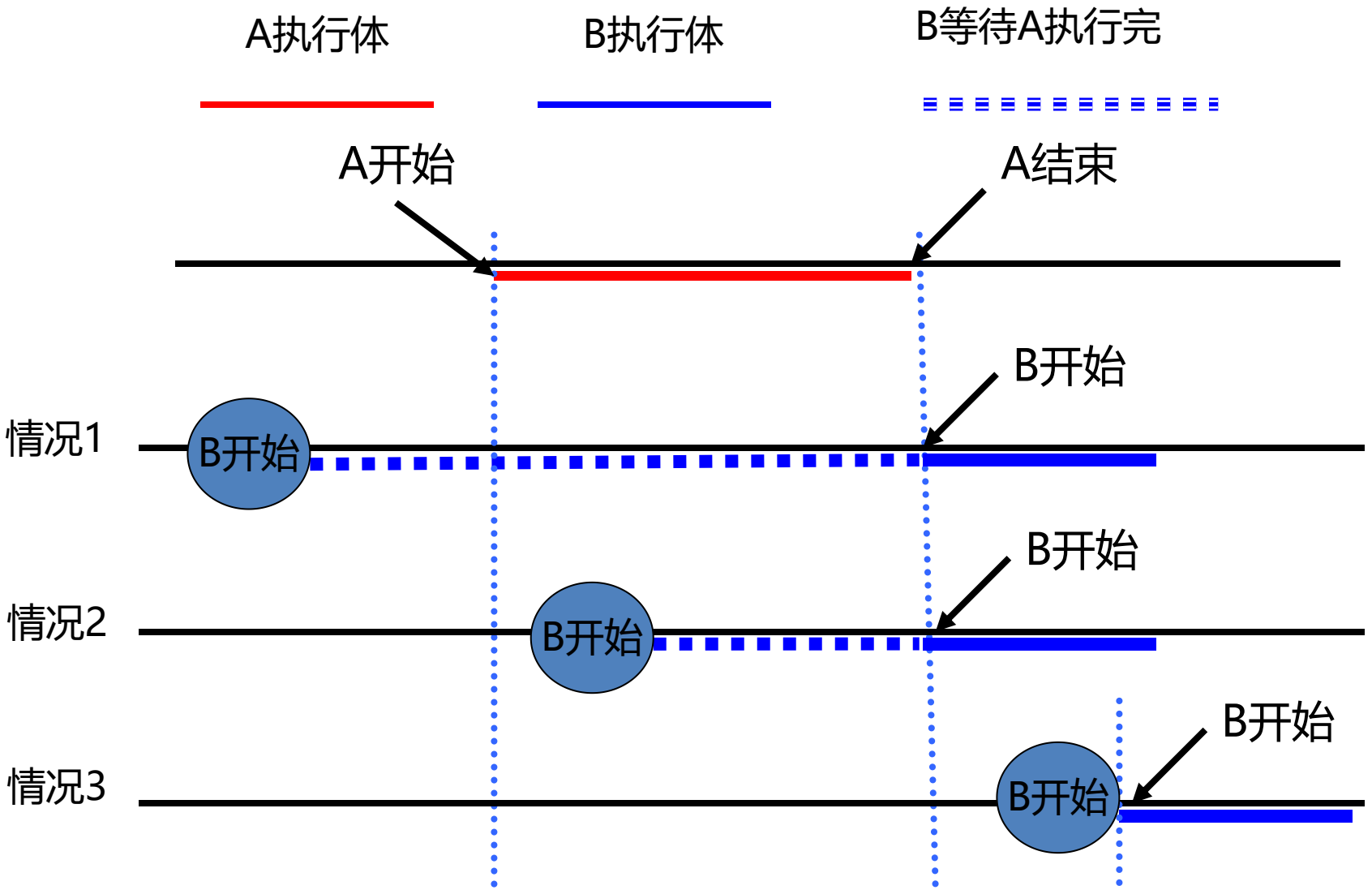
1.分析什么地方需要实现"同步关系"，即必须保证"一前一后"执行的两个操作（或两句代码）

2.设置同步信号量 S，初始为0

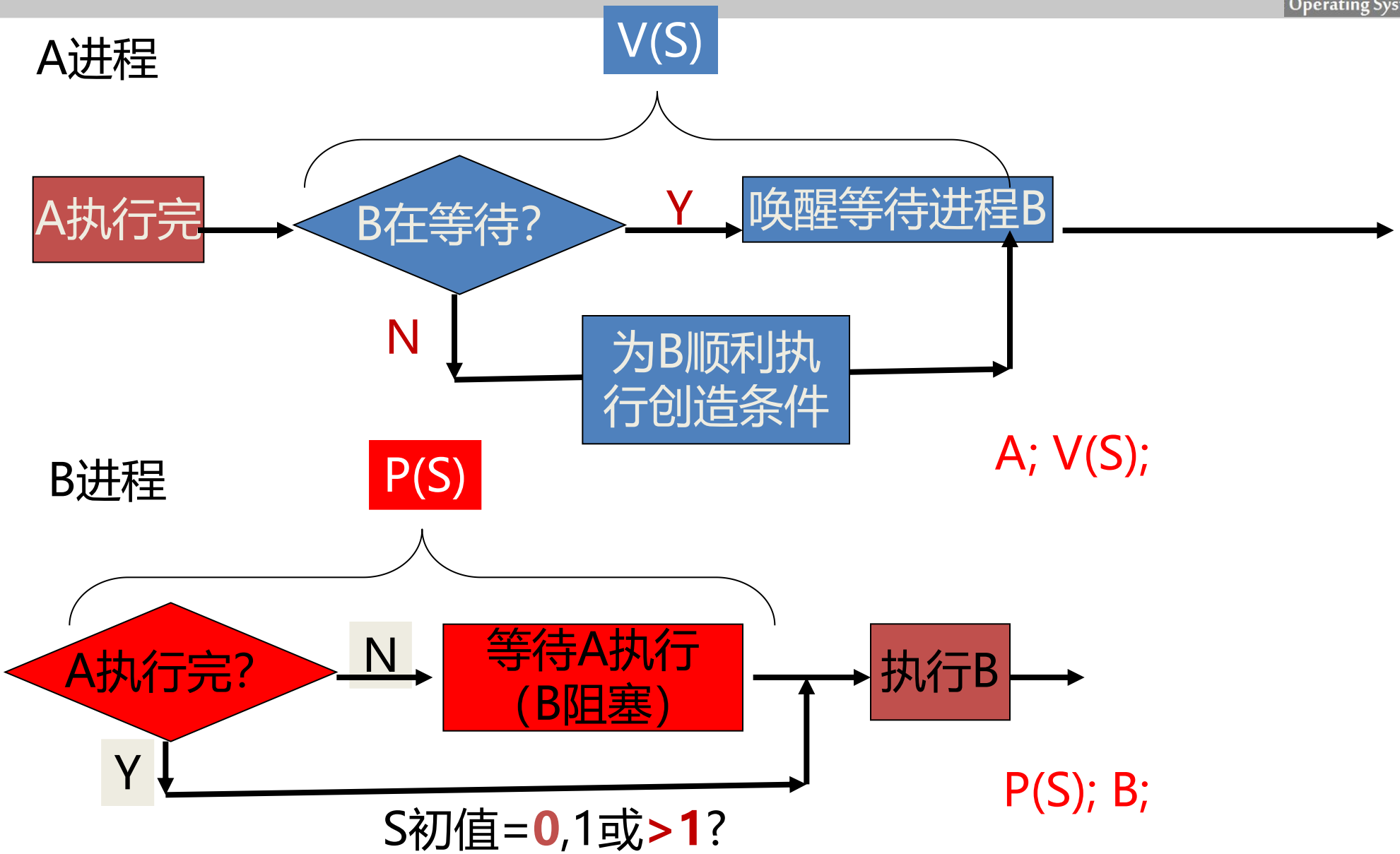
3.在"前操作"之后执行 V（S）

4.在"后操作"之前执行P（S）

利用信号量实现前驱关系：A---->B



利用信号量实现前驱关系：A---->B



利用信号量实现前驱关系

每一对前驱关系都是一个进程同步问题（需要保证一前一后的操作）

因此，

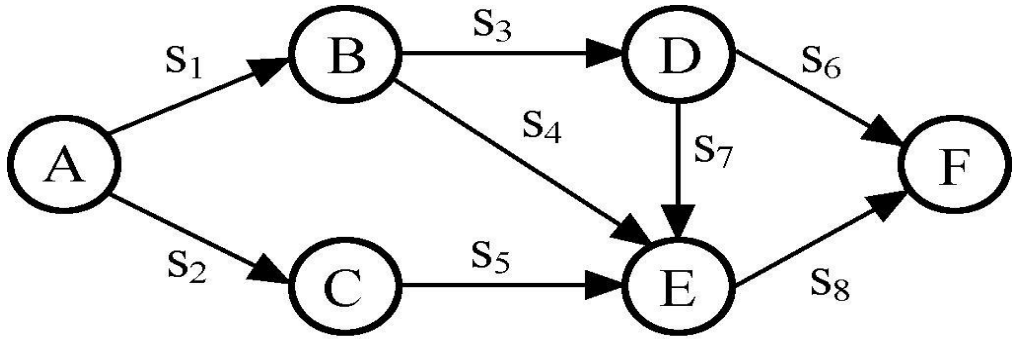
1.要为**每一对前驱关系各设置一个同步变量**

2.在**"前操作"之后**对相应的同步变量**执行 V操作**

3.在**"后操作"之前**对相应的同步变量**执行P操作**

利用信号量实现前驱关系

例 有6个进程A、B、C、D、E、F，它们并发执行的关系如下



/* 设置8个信号量,初值都为0 */

semaphore $s_1,s_2,s_3,s_4,s_5,s_6,s_7,s_8$= 0,0,0,0,0,0,0,0;

A: { A; V(s_1); V(s_2); }

B: { P(s_1); B; V(s_3); V(s_4); }

C: { P(s_2); C; V(s_5); }

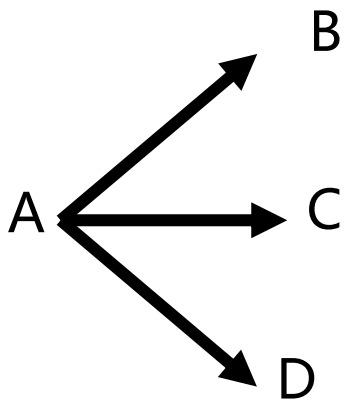
D: { P(s_3); D; V(s_6); V(s_7); }

E: { P(s_4); P(s_5); P(s_7); E; V(s_8);}

F: { P(s_6); P(s_8); F; }

（信号量个数能减少？）

利用信号量实现前驱关系

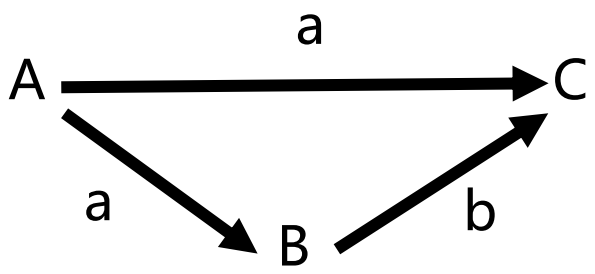


b=0,c=0,d=0

```
{   A;  V(b) ;V(c) ;V(d);}
{   P(b);   B;     }
{   P(c);   C ;    }
{   P(d) ;  D ;    }
```

a=0

```
{   A;  V(a) ;V(a) ;V(a);}
{   P(a);   B;             }
{   P(a);   C ;            }
{   P(a) ;  D  ;           }
```



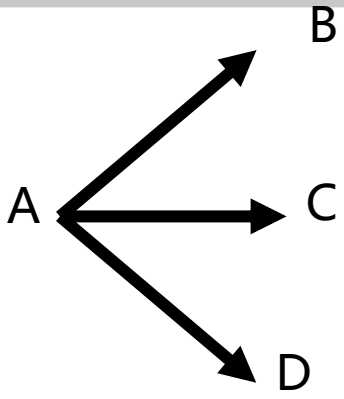
a=0,b=0

```
{ A;  V(a) ;V(a); }
{ P(a); B; V(b);  }
{ P(a); P(b);  C; }
```

a=0,b=0

```
{ A  ; V(a);       }
{ P(a); B; V(b); }
{ P(b); C;         }
```

利用信号量实现前驱关系

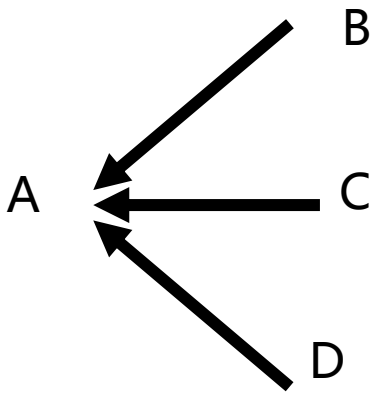


b=0,c=0,d=0

```
{   A;   V(b);V(c);V(d);}
{ P(b); B ;              }
{ P(c); C ;              }
{ P(d) ;D;               }
```

a=0

```
{  A;   V(a) ;V(a) ;V(a);}
{P(a);   B;              }
{P(a);   C;              }
{P(a) ;  D;              }
```

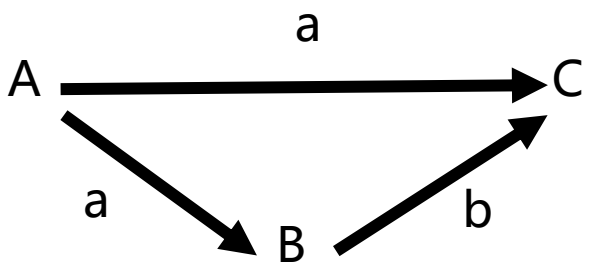


b=0,c=0,d=0

```
{ P(b) ;P(c) ;P(d) ;A;}
{ B;     V(b);          }
{ C;     V(c) ;         }
{ D;     V(d) ;         }
```

a=0

```
{ P(a) ;P(a) ;P(a) ;A;}
{   B;   V(a);         }
{   C;   V(a);         }
{   D;   V(a);         }
```



a=0,b=0

```
{ A;  V(a) ;V(a); }
{ P(a); B; V(b);  }
{ P(a); P(b);  C; }
```

a=0,b=0

```
{ A  ; V(a);       }
{ P(a); B; V(b); }
{ P(b); C;         }
```

利用信号量实现前驱关系

练习： 有并发进程P_1和P_2

进程P1	进程P2
S_1 $x= a + 2$	S_3 $y= x + 4$
S_2 $z= 3 \times y$	S_4 $w= z + 6$

进程并发需要按照S1→S3→S2→S4的顺序执行，才能得到*x*、*y*、z和w的合理值

$$S1 \xrightarrow{a} S3 \xrightarrow{b} S2 \xrightarrow{c} S4$$

semaphore a,b,c= 0,0,0; /* 3个信号量,初值都为0 */

进程P1

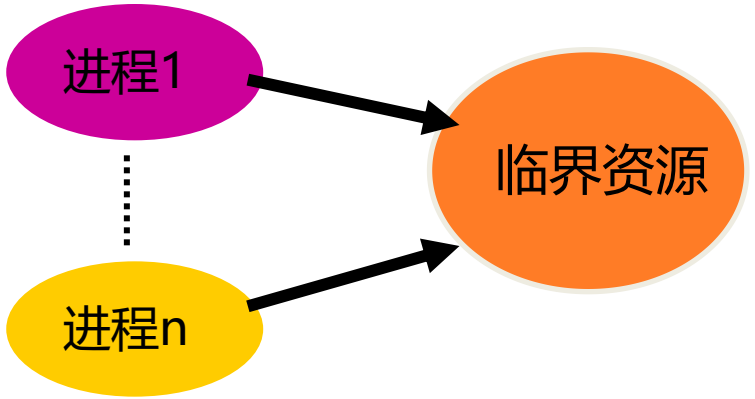
```
S1  x= a + 2;
    V(a);
    P(b);
S2  z= 3 × y;
    V(c);
```

进程P2

```
    P(a);
S3  y= x + 4;
    V(b);
    P(c);
S4  w= z + 6;
```

信号量（semaphore）机制

总结：N个进程共享一临界资源,一个互斥信号量mutex=1(初值)



互斥访问

mutex=1

进入区	P(mutex)；
	CSi;(临界区)
退出区	V(mutex)；

Mutex何时可以>1

利用信号量实现前趋关系A-->B

一个信号量S=0(初值)

进程A	A; V(S);
进程B	P(S); B;

信号量用于互斥与前趋关系比较

1)信号量的个数及初值

2)涉及的进程数量

3)P，V所处的位置

4)重复执行的次数

分析同步问题

分析进程同步和互斥问题的方法步骤

- **关系分析。找出问题中的进程数，并且分析它们之间的同步和互斥关系。同步、互斥、前驱关系直接按照上面例子中的经典范式改写。**

- **整理思路。找出解决问题的关键点，并且根据做过的题目找出解决的思路。根据进程的操作流程确定P操作、V操作的大致顺序。**

- **设置信号量。根据上面两步，设置需要的信号量，确定初值，完善整理。**

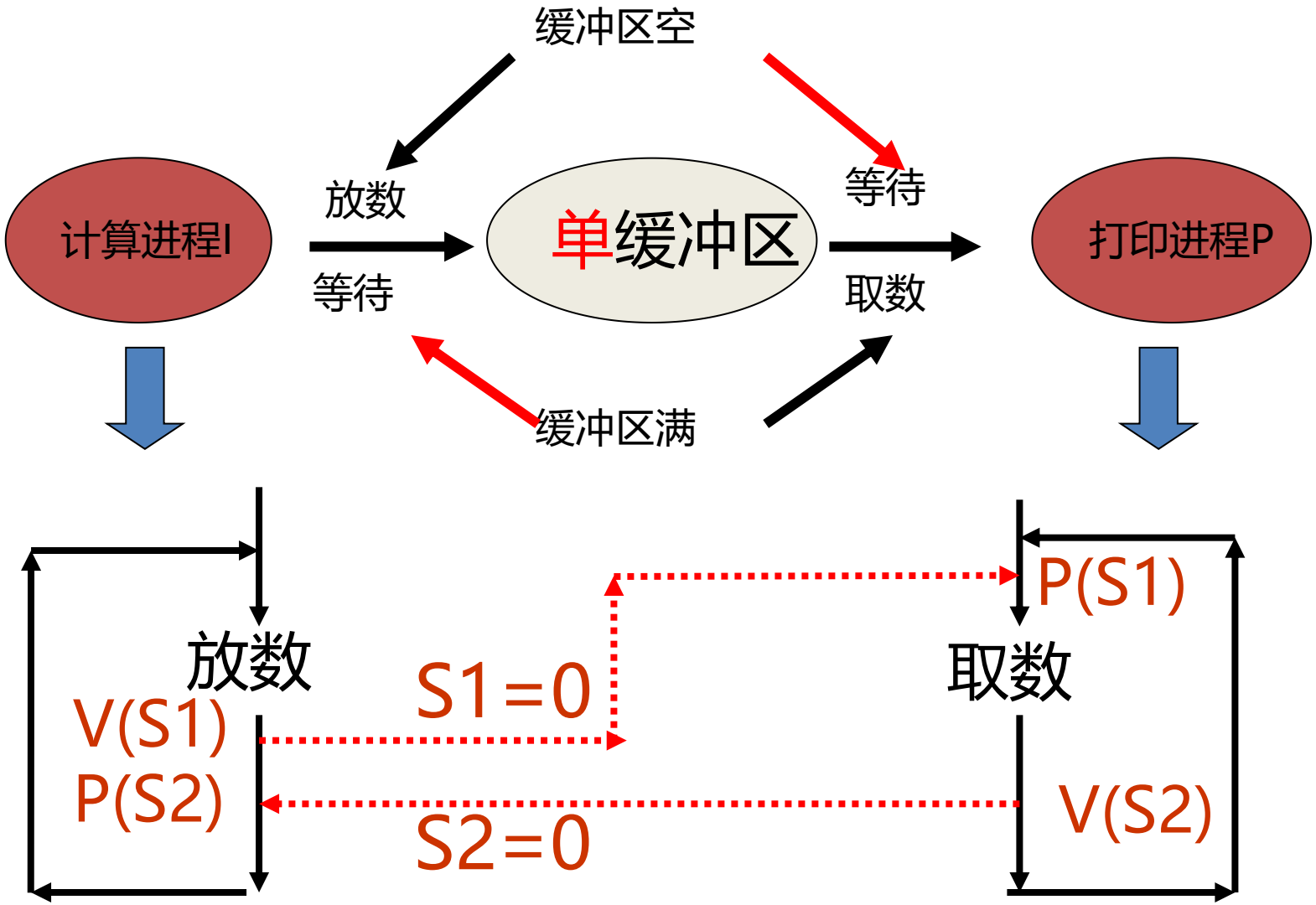
单缓冲的同步问题

【例】有一计算进程和打印进程，它们共享一个单缓冲区，计算进程不断地计算出一个整形结果并将它放入单缓冲区中，打印进程则负责从单缓冲区中取出每一个结果进行打印，请用信号量来实现它们的同步关系。

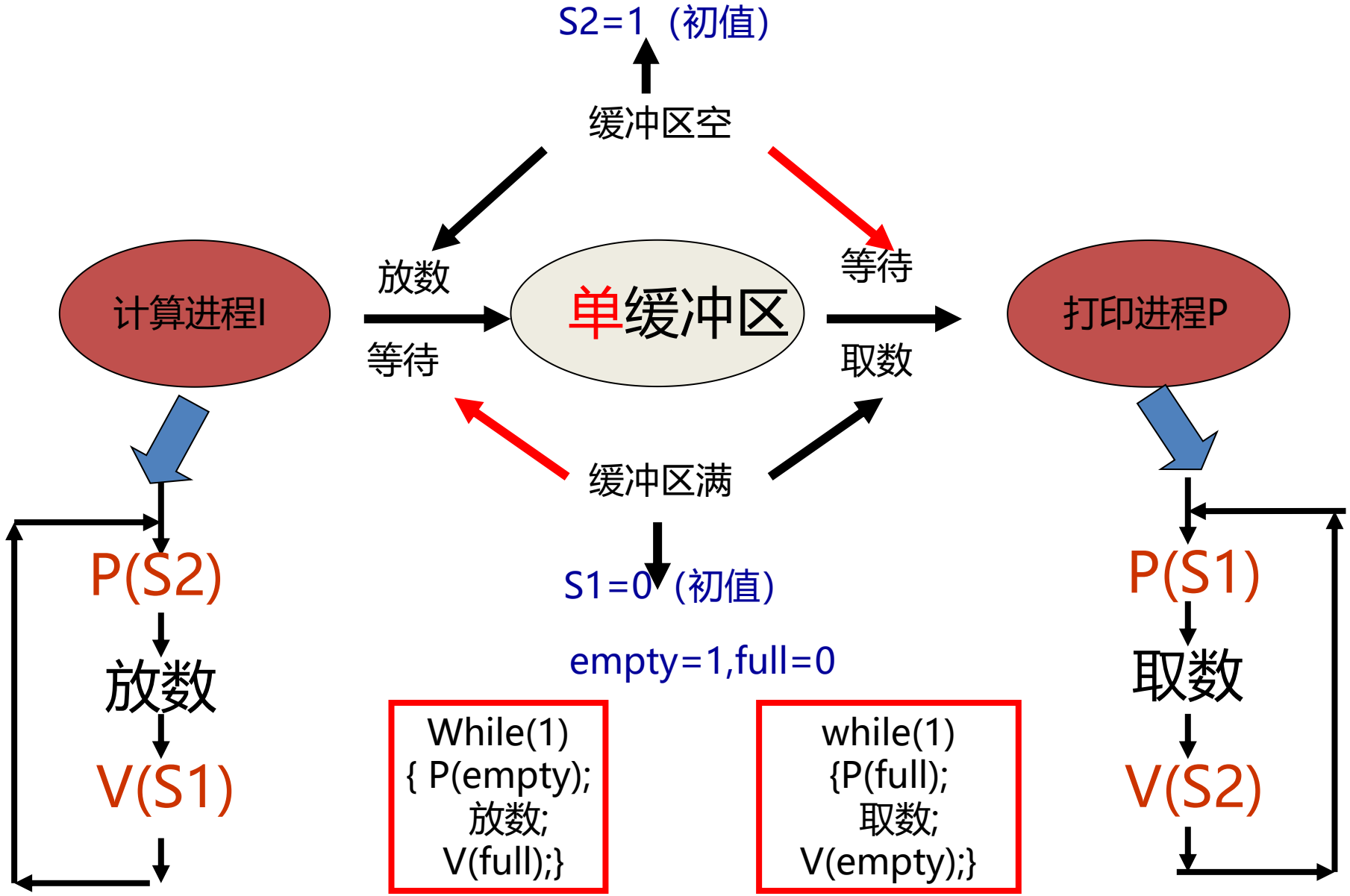
单缓冲的同步问题

分析1：可从临界资源的角度来思考，先找临界资源，并为每种临界资源设置信号量，在访问临界资源之前加wait操作来申请资源，访问完临界资源后加signal操作来释放临界资源。本题中有两类临界资源，第一类是计算进程争用的空闲缓冲区，初始状态下有一个空闲缓冲可供使用，故可为它设置初值为1的信号量empty;第二类是打印进程争用的已放入缓冲中的打印结果，初始状态下缓冲中无结果可供打印，故可为它设置初值为0的信号量full。

单缓冲的同步问题


缓冲区空
放数
等待
计算进程I
单缓冲区
等待
取数
打印进程P
缓冲区满
放数
V(S1)
P(S2)
S1=0
S2=0
P(S1)
取数
V(S2)

单缓冲的同步问题


S2=1 (初值)
缓冲区空
放数
等待
计算进程I
单缓冲区
等待
取数
打印进程P
缓冲区满
P(S2)
放数
V(S1)
S1=0 (初值)
empty=1,full=0
While(1)
{ P(empty);
放数;
V(full);}
while(1)
{P(full);
取数;
V(empty);}
P(S1)
取数
V(S2)

单缓冲的同步问题

答1：具体的同步算法可描述为：

```
semaphore    full=0, empty=1;
int buffer;
cp(){
        int nextc;
        while(1){
            compute the next number in nextc;;
            wait(empty);
            buffer=nextc;
            signal(full);
        }
}
```

```
pp(){
        int nextp;
        while(1){
            wait(full);
            nextp=buffer;
            signal(empty);
            print the number in nextp;
        }
}
```

```
main(){
        cobegin
                cp();   pp();
        coend
}
```

单缓冲的同步问题

分析2：还可从同步的角度来思考，对某种同步关系，如进程A在某处必须等待进程B完成某个动作D后才能继续执行，可为它设置一初值为0的信号量，并在A需要等待B 的位置插入wait操作，在B完成动作D之后插入signal操作。本题中存在两种同步关系：

（1）打印进程必须等待计算进程将计算结果放入缓冲之后，才能取结果打印，因此，可为它们设置初值为0的信号量S_A；

（2）除第一个计算结果可直接放入缓冲区外，计算进程必须等打印进程将缓冲中的前一个结果取走，缓冲区变空后，才能将下一个计算结果放入缓冲区，因此，可为它们设置初值为0的信号量S_B。

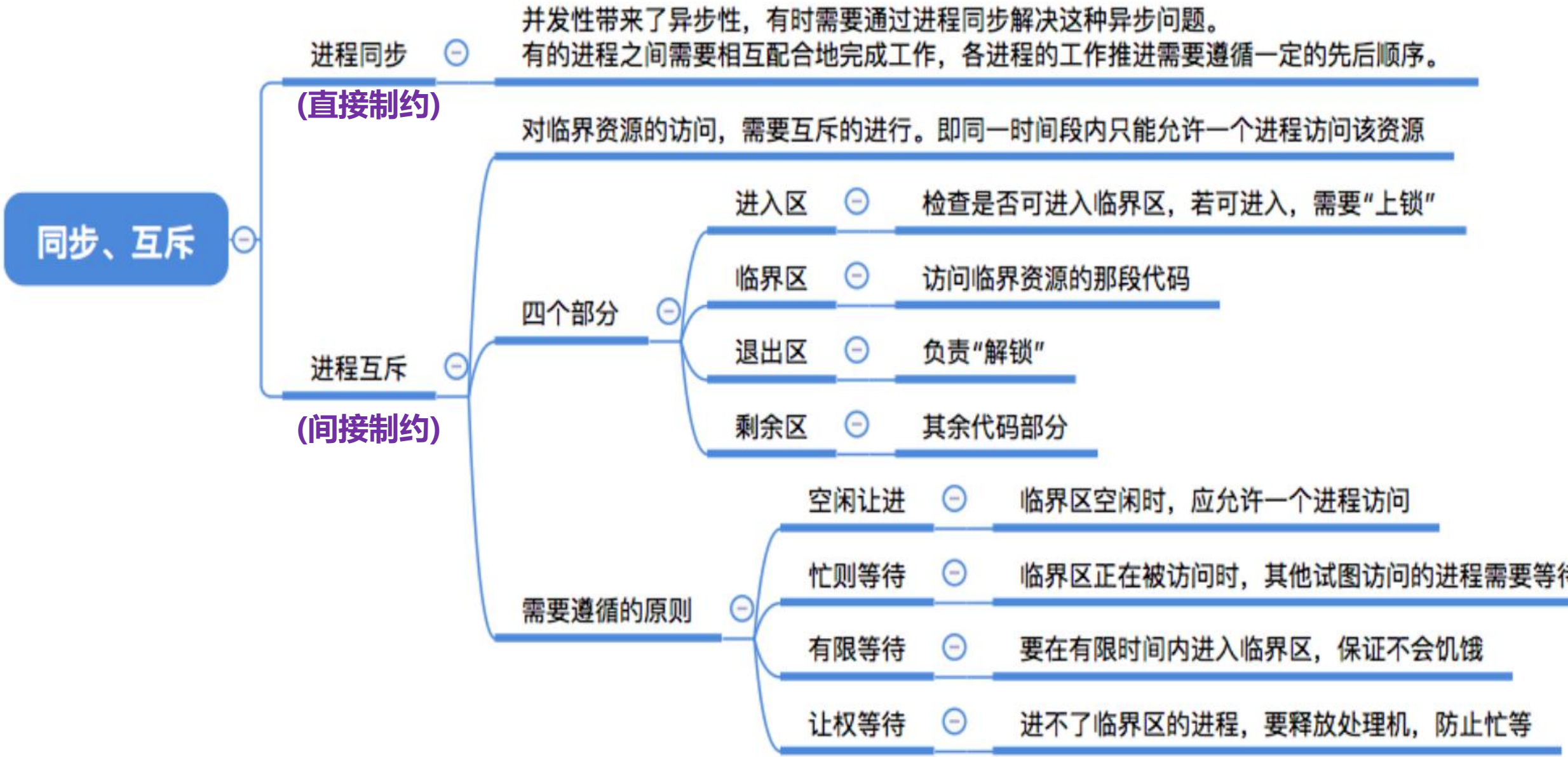
单缓冲的同步问题

答2：具体的同步算法可描述为：

```
semaphore SA=0, SB =0;
int buffer ;
cp(){
    int nextc;
    compute the first number in nextc;
    buffer=nextc;
    signal(SA);
    while(1){
        compute the next number in nextc;
        wait(SB);
        buffer=nextc;
        signal(SA);   }
}
```

```
pp(){
    int nextp;
    while(1){
        wait(SA);
        nextp=buffer;
        signal(SB);
        print the number in nextp;
    }
}
```

小结


同步、互斥
进程同步
(直接制约)
并发性带来了异步性，有时需要通过进程同步解决这种异步问题。
有的进程之间需要相互配合地完成工作，各进程的工作推进需要遵循一定的先后顺序。
进程互斥
(间接制约)
对临界资源的访问，需要互斥的进行。即同一时间段内只能允许一个进程访问该资源
四个部分
进入区
检查是否可进入临界区，若可进入，需要"上锁"
临界区
访问临界资源的那段代码
退出区
负责"解锁"
剩余区
其余代码部分
需要遵循的原则
空闲让进
临界区空闲时，应允许一个进程访问
忙则等待
临界区正在被访问时，其他试图访问的进程需要等待
有限等待
要在有限时间内进入临界区，保证不会饥饿
让权等待
进不了临界区的进程，要释放处理机，防止忙等

小结

注意：

- 临界区是进程中访问临界资源的代码段。
- 进入区和退出区是负责实现互斥的代码段。
- 临界区也可称为"临界段"。

小结

- 信号量机制
  - 整型信号量
    - 用一个整数型变量作为信号量，数值表示某种资源数
    - 整型信号量与普通整型变量的区别：对信号量只能执行 初始化、P、V 三种操作
    - 整型信号量存在的问题：不满足让权等待原则
  - 记录型信号量
    - S.value 表示某种资源数，S.L 指向等待该资源的队列
    - P 操作中，一定是先 S.value--，之后可能需要执行 block 原语
    - V 操作中，一定是先 S.value++，之后可能需要执行 wakeup 原语
    - 注意：要能够自己推断在什么条件下需要执行 block 或 wakeup
    - 可以用记录型信号量实现系统资源的“申请”和“释放”
    - 可以用记录型信号量实现进程互斥、进程同步

大题、小题超高频出题点

小结

- 信号量机制的基本原理：两个或多个进程可以利用彼此间收发的简单的信号来实现"正确的"并发执行，一个进程在收到一个指定信号前，会被迫在一个确定的或者需要的地方停下来，从而保持同步或互斥。
- 用户进程可以通过使用操作系统提供的一对原语来对信号量进行操作，从而很方便的实现了进程互斥、进程同步。
- 信号量其实就是一个变量（可以是一个整数，也可以是更复杂的记录型变量），可以用一个信号量来表示系统中某种资源的数量，比如：系统中只有一台打印机，就可以设置一个初值为1的信号量。

小结

- 原语是一种特殊的程序段，其执行只能一气呵成，不可被中断。原语是由关中断/开中断指令实现的。
- 一对原语：wait（S）原语和signal（S）原语，可以把原语理解为我们自己写的函数，函数名分别为 wait 和 signal，括号里的信号量S其实就是函数调用时传入的一个参数。
- wait、signal原语常简称为P、V操作（来自荷兰语proberen和verhogen）。因此，做题的时候常把wait(S)、signal(S)两个操作分别写为P(S)、V(S)

小结

除了互斥、同步问题外，还会考察有多个资源的问题，有多少资源就把信号量初值设为多少。申请资源时进行P操作，释放资源时进行 V 操作即可

信号量机制

- 实现进程互斥
  - 分析问题，确定临界区
  - 设置互斥信号量，初值为1
  - 临界区之前对信号量执行 P 操作
  - 临界区之后对信号量执行 V 操作

互斥问题，信号量初值为1

- 实现进程同步
  - 分析问题，找出哪里需要实现“一前一后”的同步关系
  - 设置同步信号量，初始值为0
  - 在“前操作”之后执行 V 操作
  - 在“后操作”之前执行 P 操作

同步问题，信号量初值为0

- 实现进程的前驱关系
  - 分析问题，画出前驱图，把每一对前驱关系都看成一个同步问题
  - 为每一对前驱关系设置同步信号量，初值为0
  - 在每个“前操作”之后执行 V 操作
  - 在每个“后操作”之前执行 P 操作

前驱关系问题，本质上就是更复杂的同步问题

回顾——实例分析

互斥量mutex_vat=1

某寺庙，有小和尚、老和尚若干。有一水缸，由小和尚提

两个流程　　empty=10

水入缸，老和尚从缸中取水饮用。水缸可容纳10桶水，水

互斥量mutex well=1

取自同一水井中，水井径窄，每次只能容一个水桶取水。

pail=3　　确认取水\入水时水缸互斥

水桶总数为3个，每次入、取缸水仅为1桶，且不可同时进

行。试给出取水、入水的算法描述。

小和尚**打水入缸**流程：
拿桶-->去水井取水（互斥，P、V操作）-->把水倒入水缸（互斥，P、V操作）-->放桶
老和尚**取水饮用**流程：
拿桶-->去水缸取水（互斥，P、V操作）--->放桶

回顾——实例分析

```
1   semaphore  mutex_well = 1, mutex_vat = 1;//互斥量
2  semaphore  pail = 3,empty = 10, full = 0;//定义自然量，empty表示水缸的总容量，full表示满的标志
3  project  small()//小和尚
4  {
5   while (true)
6   {
7    P(empty);//判断水缸是否还有容量，有则减一，程序向下执行
8    P(pail);//申请一个桶
9    P(mutex_well);//占用水井
10   从水井中打水;//活动
11   V(mutex_well);//用完水井，释放资源
12   P(mutex_vat);//占用水缸
13   将水倒入水缸中;//活动
14   V(mutex_vat);//释放资源
15   V(pail);//放桶
16   V(full);//full+1，注意成对出现问题！！！
17  }
18 }
```

```
19   project old()//老和尚
20  {
21   while (true)
22   {
23    P(full);//看是否有水，有则减一，程序向下执行
24    P(pail);//拿桶
25    P(mutex_vat);//占用水缸
26    从水缸中取水;//活动
27    V(mutex_vat);//用完水缸，释放资源
28    喝水;//此处虽可省，但有这一步更为具体形象
29    V(pail);//放桶
30    V(empty);//容量加一
31   }
32  }
```

3.5.1 生产者—消费者问题

①一个生产者、一个消费者共享一个缓冲区
②一个生产者、一个消费者共享多个缓冲区
③多个生产者、多个消费者共享多个缓冲区

一个生产者、一个消费者共享一个缓冲区的解


```
int B;
semaphore empty; //可以使用的空缓冲区数
semaphore full;  //缓冲区内可以使用的产品数
empty=1;         //缓冲区内允许放入一件产品
full=0;          //缓冲区内没有产品
cobegin
process producer(){
  while(true){
        produce( );
        P(empty);
        append( ) to B;
        V(full);
    }
  }
coend
```

```
process consumer(){
    while(true) {
        P(full);
        take( ) from B;
        V(empty);
        consume( );
      }
    }
```

3.5.1 经典进程同步问题之一：生产者—消费者问题

- 问题描述：在生产者和消费者之间有**共用缓冲池**，有n个缓冲区，生产者不断地向缓冲池中**生产**物品（相当于向缓冲区**添加**物品，即占有缓冲区），每个缓冲区可以放一个物品；消费者也不断**消费**物品（相当于向缓冲区**删除**物品，即腾空缓冲区）。

- 只要缓冲池中仍有空闲的缓冲区就可以**不断地生产**；
- 同样，只要有缓冲区仍有物品就可以**不断地消费**。

3.5.1 经典进程同步问题之一：生产者—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

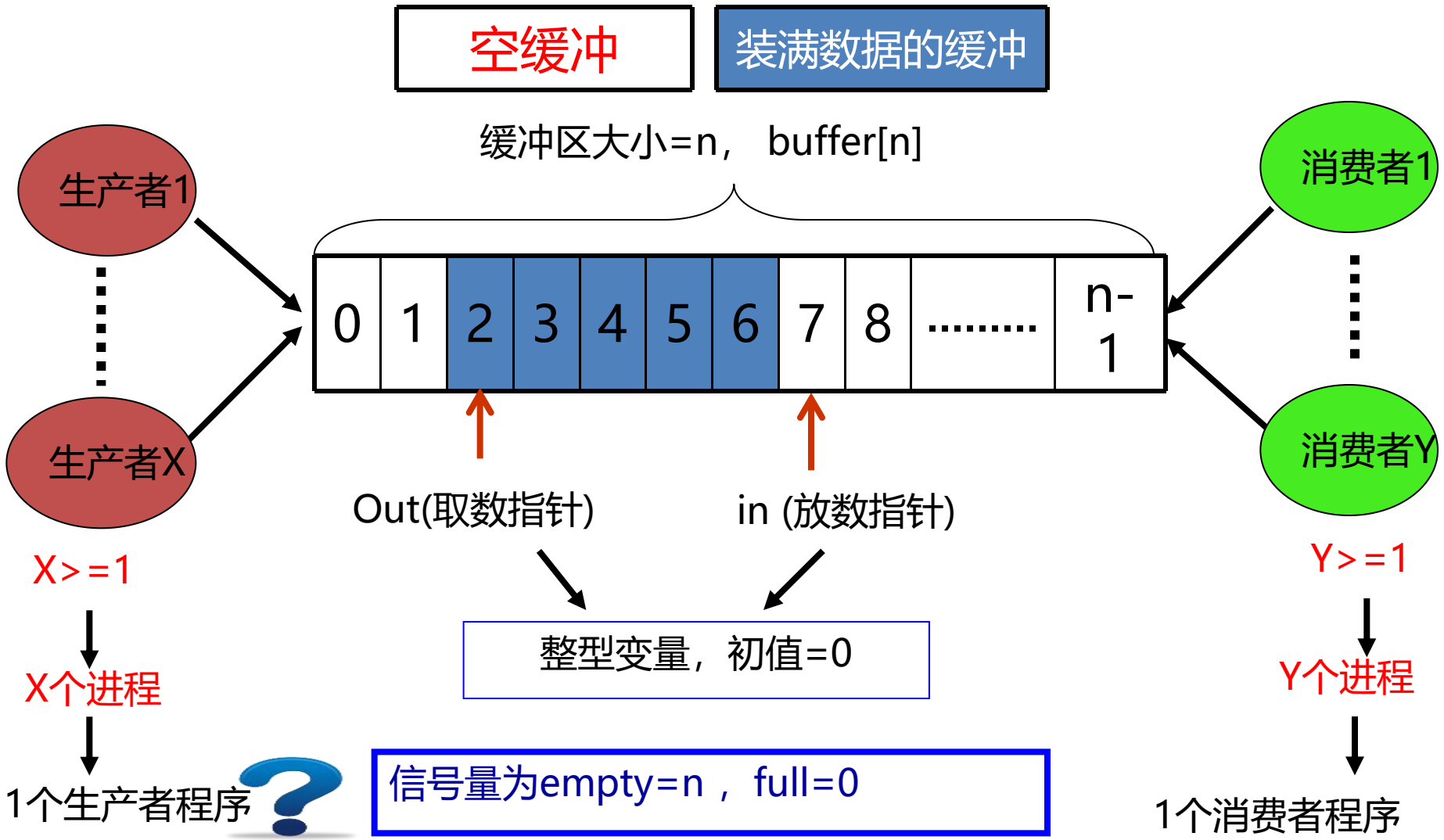
3.5.1 经典进程同步问题之一：生产者—消费者问题

- **互斥信号量**
  - 必定成对出现：
  - 进入临界区——临界区——退出临界区
- **同步信号量**
  - 未必成对出现，依赖于同步关系的性质
- **同步信号量和互斥信号量的操作顺序**
  - 基本原则：互斥信号量永远紧邻临界区

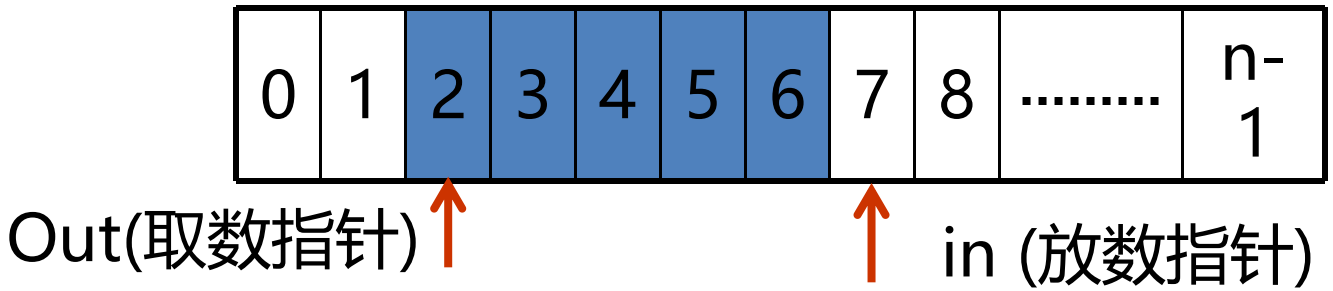
3.5.1 经典进程同步问题之一：生产者—消费者问题

- 缓冲池buffer[n]：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：
  - 输入指针加1表示成 in:= (in+1)mod n；
  - 输出指针加1表示成out:= (out+1) mod n。
  - 当 (in+1) mod n=out时表示缓冲池满；而in=out则表示缓冲池空。

3.5.1 经典进程同步问题之一：生产者—消费者问题



3.5.1 经典进程同步问题之一：生产者—消费者问题



信号量

1）empty表示生产者可放产品的缓冲区个数，初值=n
2）full表示消费者可取的产品数，初值=0

生产者

P(empty)

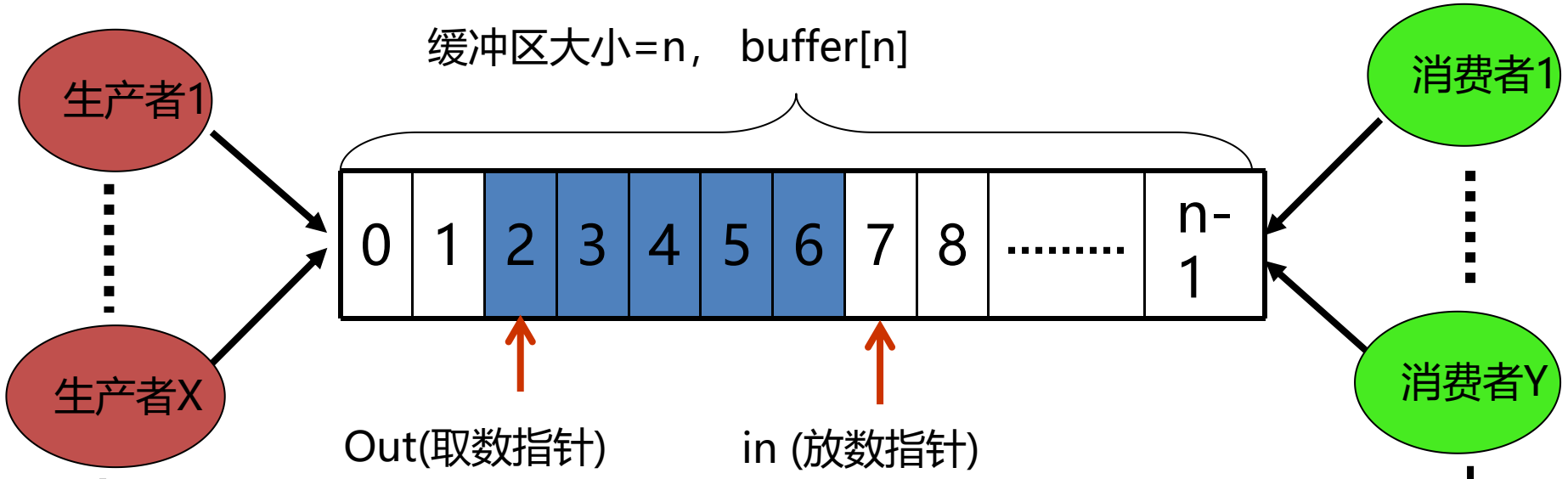
只要缓冲区未满，生产者可将产品放入；→V(full)
否则，等待缓冲区有空

消费者

P(full)

只要缓冲区未空，消费者可将产品取出，→ V(empty)
否则，等待有可取产品

3.5.1 经典进程同步问题之一：生产者—消费者问题



信号量为empty=n ， full=0

1个生产者程序

1个消费者程序

Producer(生产者)

P(empty);

buffer(in)=nextp;

in= (in+1) % n;

V(full);

Consumer(消费者)

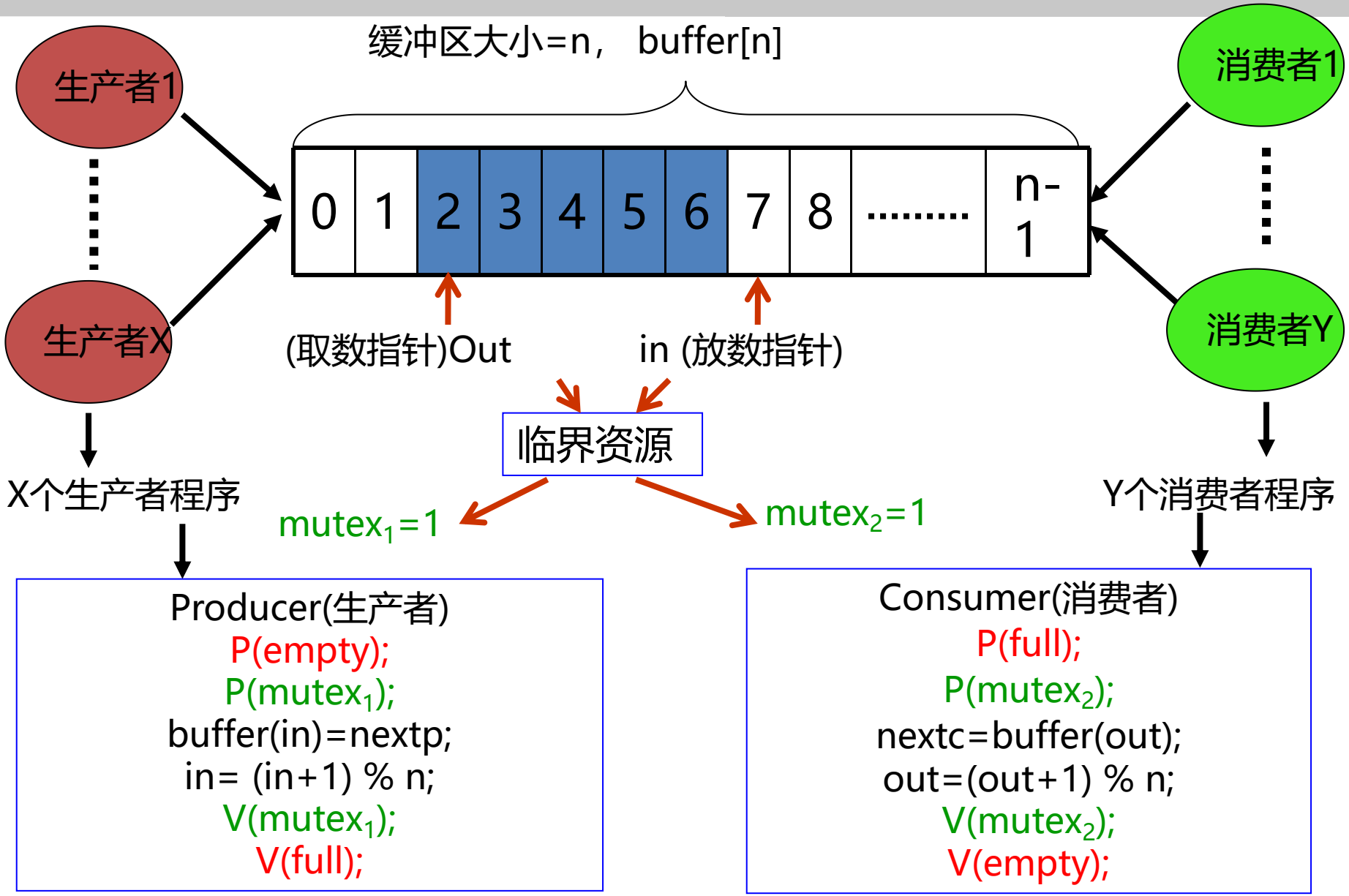
P(full);

nextc=buffer(out);

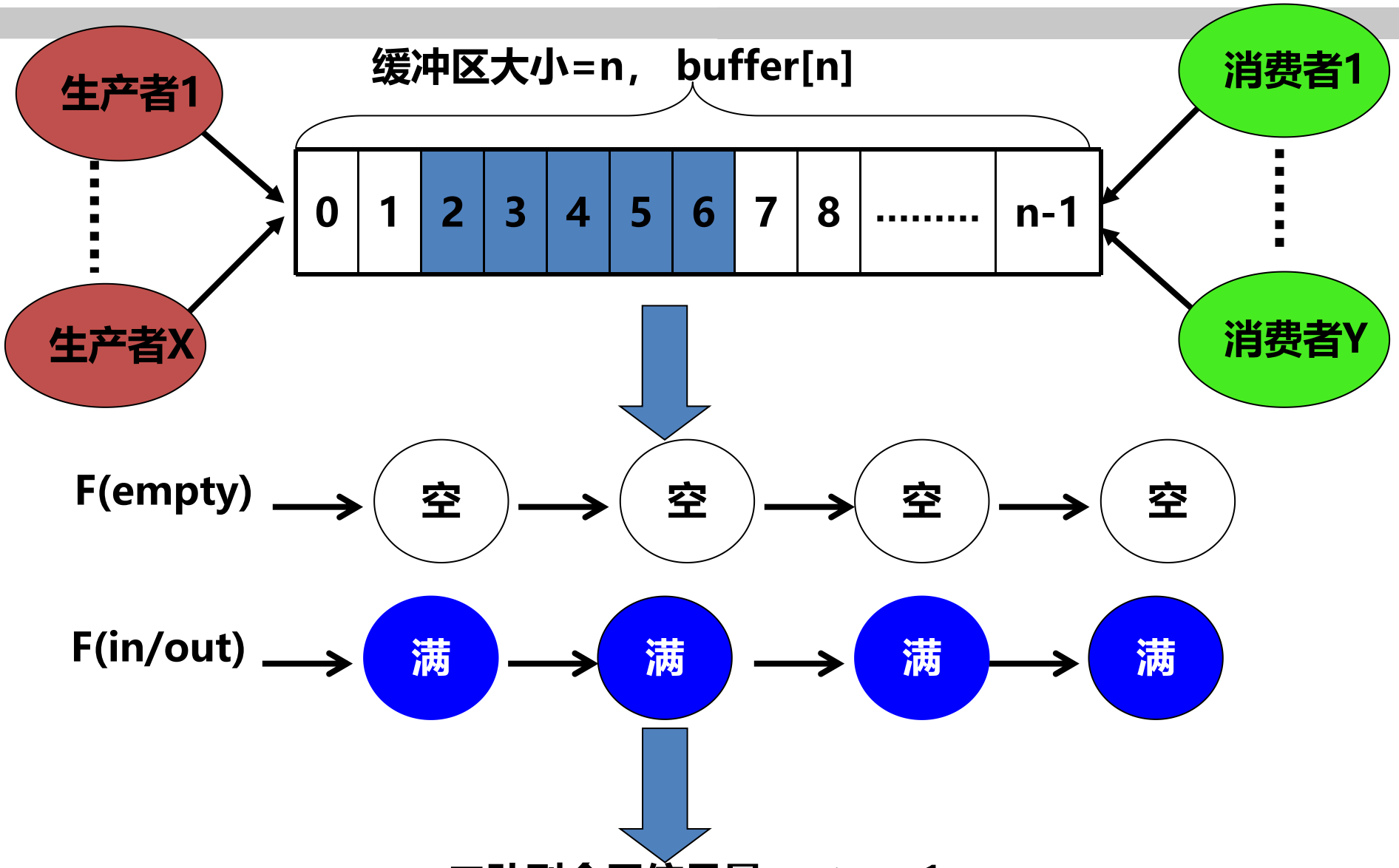
out=(out+1) % n;

V(empty);

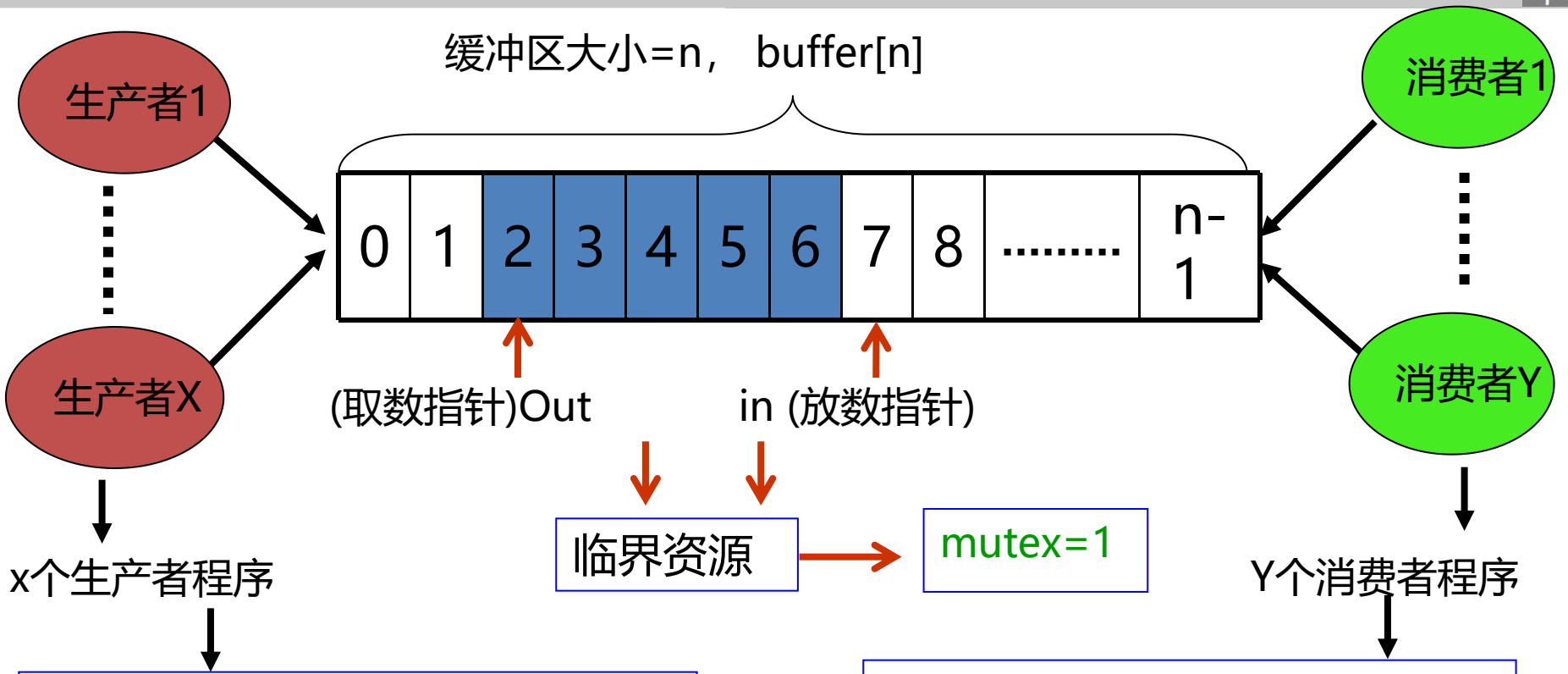
3.5.1 经典进程同步问题之一：生产者—消费者问题



3.5.1 经典进程同步问题之一：生产者—消费者问题



3.5.1 经典进程同步问题之一：生产者—消费者问题




```
Producer(生产者)
P(empty);
P(mutex);
buffer(in)=nextp;
in= (in+1) % n;
V(mutex);
V(full);
```

```
Consumer(消费者)
P(full);
P(mutex);
nextc=buffer(out);
out=(out+1) % n;
V(mutex);
V(empty);
```

3.5.1 经典进程同步问题之一：生产者—消费者问题

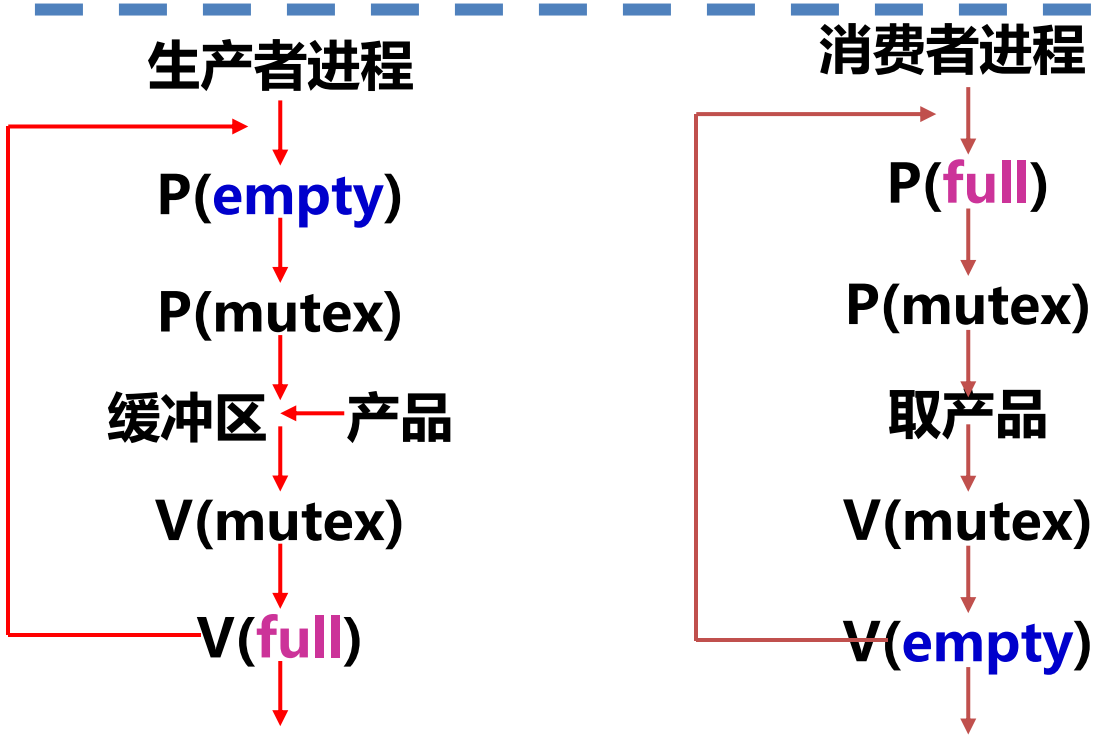
full是“满”缓冲数目，初值为0。

empty是“空”缓冲数目，初值为N。

实际上，full和empty是同一个含义。

full + empty == N

mutex用于访问缓冲区时的互斥，初值是1。



3.5.1 经典进程同步问题之一：生产者—消费者问题

生产者——消费者问题

整型变量in=0,out=0

资源信号量为empty=n ， full=0

互斥信号量为mutex=1

Producer(生产者)
P(empty);
P(mutex);
buffer(in)=nextp;
in= (in+1) % n;
V(mutex);
V(full);

Consumer(消费者)
P(full);
P(mutex); nextc=buffer(out);
out=(out+1) % n;
V(mutex);
V(empty);

Q)二个V的次序可交换? 先mutex后资源信号量有利?

3.5.1 经典进程同步问题之一：生产者—消费者问题


```
Producer(生产者)
P(empty);
P(mutex);
buffer(in)=nextp;
in= (in+1) % n;
V(mutex);
V(full);
```

```
Consumer(消费者)
P(full);
P(mutex);
nextc=buffer(out);
out=(out+1) % n;
V(mutex);
V(empty);
```

分析：如果将两个P操作，即 P(full)和P(mutex)互换位置，或者P(empty)和 P(mutex)互换位置，其后果如何?

如果将两个V操作互换位置，即V(full)和V(mutex)互换位置，或者V(empty)和V(mutex)互换位置，其后果又如何?

答：在生产者—消费者问题中，如果将两个 P操作，即P(full)和P(mutex)互换位置，或者P(empty)和P(mutex)互换位置，都**可能引起死锁**。考虑系统中缓冲区全满时，若一生产者进程先执行了P(mutex)操作并获得成功，当再执行P(empty)操作时，它将因失败而进入阻塞状态，它期待消费者执行V(empty)来唤醒自己，在此之前，它不可能执行 V(mutex)操作，从而使企图通过P(mutex)进入自己的临界区的其他生产者和所有的消费者进程全部进入阻塞状态，从而引起系统死锁。类似地，消费者进程若先执行 P(mutex)，后执行 P(full)同样可能造成死锁。

V(full)和V(mutex)互换位置，或者**V(empty)和V(mutex)互换**位置，则**不会引起死锁**，其影响只是**改变**临界资源的释放**次序**。

```
//生产者
Producer(){
    while(1){
        P(empty); //生产者申请一个可用缓冲区
        P(mutex);//互斥使用缓冲区
        buffer(in)=nextp;
        in= (in+1) % n;
        V(mutex);//释放缓冲区占用权
        V(full); //增加一个消费者可用数量
      }
   }
```

```
//消费者
Consumer(){
    while(1){
        P(full); //申请一个消费者可用数量
        P(mutex); //互斥使用缓冲区
        nextc=buffer(out);
        out=(out+1) % n;
        V(mutex); //释放缓冲区使用权
        V(empty); //增加一个缓冲区（生产者可用数量）
        }
    }
```

```
semaphore empty=n;//同步信号量
semaphore full=0;//同步信号量
semaphore mutex=1;//互斥信号量
int in,out=0;//放数取数指针
```

生产者和消费者之间的约束：临界区。
生产者在生产时，消费者不能消费。

控制：缓冲区空时不能继续消费，制约消费者进程C。(同步关系)

控制：缓冲区满时不能继续生产，制约生产者进程P。(同步关系)

生产者和消费者之间的约束：临界区。
生产者在生产时，消费者不能消费。

P和V操作成对出现；

对资源信号量的P操作在前

对互斥信号量的P操作在后

回顾思考：单缓冲区问题与生产者——消费者问题区别

单缓冲的情况下，缓冲区只需用简单变量来描述，而不必再用数组;另外，也不再需要in/out指针来指示产品放到（取自）哪个缓冲区，而且，由于此时生产者、消费者不可能同时访问缓冲区，所以原来的mutex信号量也不再需要。

Producer(生产者)
P(empty);
P(mutex);
buffer(in)=nextp;
in= (in+1) % n;
V(mutex);
V(full);

Consumer(消费者)
P(full);
P(mutex);
nextc=buffer(out);
out=(out+1) % n;
V(mutex);
V(empty);

```
semaphore   full=0, empty=1;
int buffer;
cp(){
    int nextc;
    while(1){
        compute the next number in nextc;;
        wait(empty);
        buffer=nextc;
        signal(full);
    }
}
```

```
pp(){
    int nextp;
    while(1){
        wait(full);
        nextp=buffer;
        signal(empty);
        print the number in nextp;
    }
}
```

读者——写者问题

- 问题描述：读者和写者共若干人，都可以向**同一个存储区**进行操作。
- 写者执行写操作时,每次**只允许一位写者**进行写操作；读者执行读操作，**可以多名读者**同时进行读操作，读者读的时候写者不能写。

20

读者——写者问题

- 问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个。
  - “读 - 写”互斥，
  - “写 - 写”互斥，
  - "读 - 读"允许

21

读者——写者问题

- 写者信号量mutex: 实现读者与写者进程间的互斥。
- 读者计数 rcount
- 读者信号量rmutex

注：其中写者信号量的权力最大，如果写者获得，读者不能读；如果读者获得，读者能读但不能写，写者肯定不能写。

22

读者——写者算法

Reader	Writer

P(rmutex) ;
if(rcount == 0)
P(mutex) ;
rcount++;
V(rmutex) ;
	P(mutex) ;

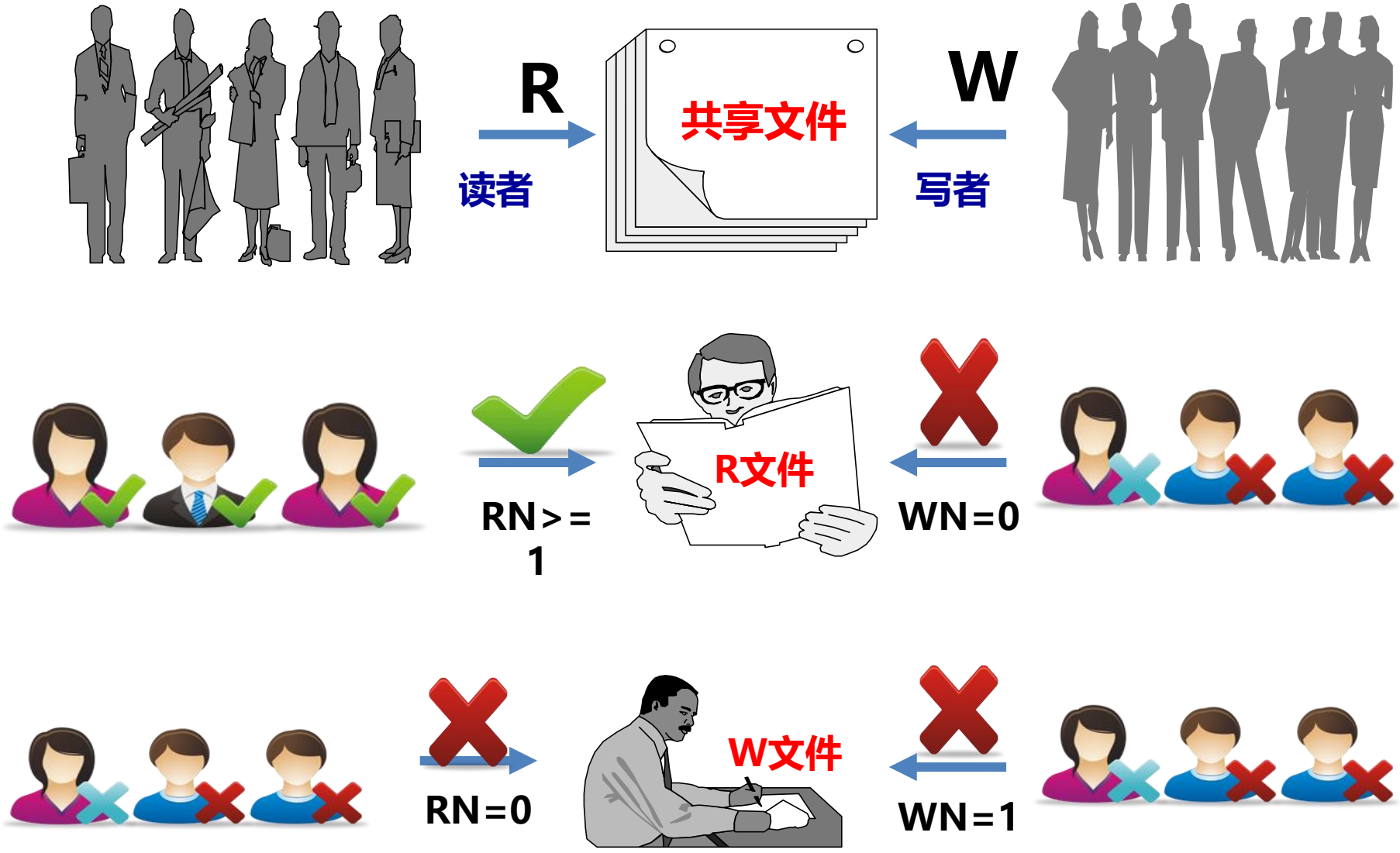
READ ;	WRITE ;
P(rmutex) ; rcount--; if(rcount == 0) V(mutex) ; V(rmutex) ;	V(mutex) ;

读者——写者算法说明

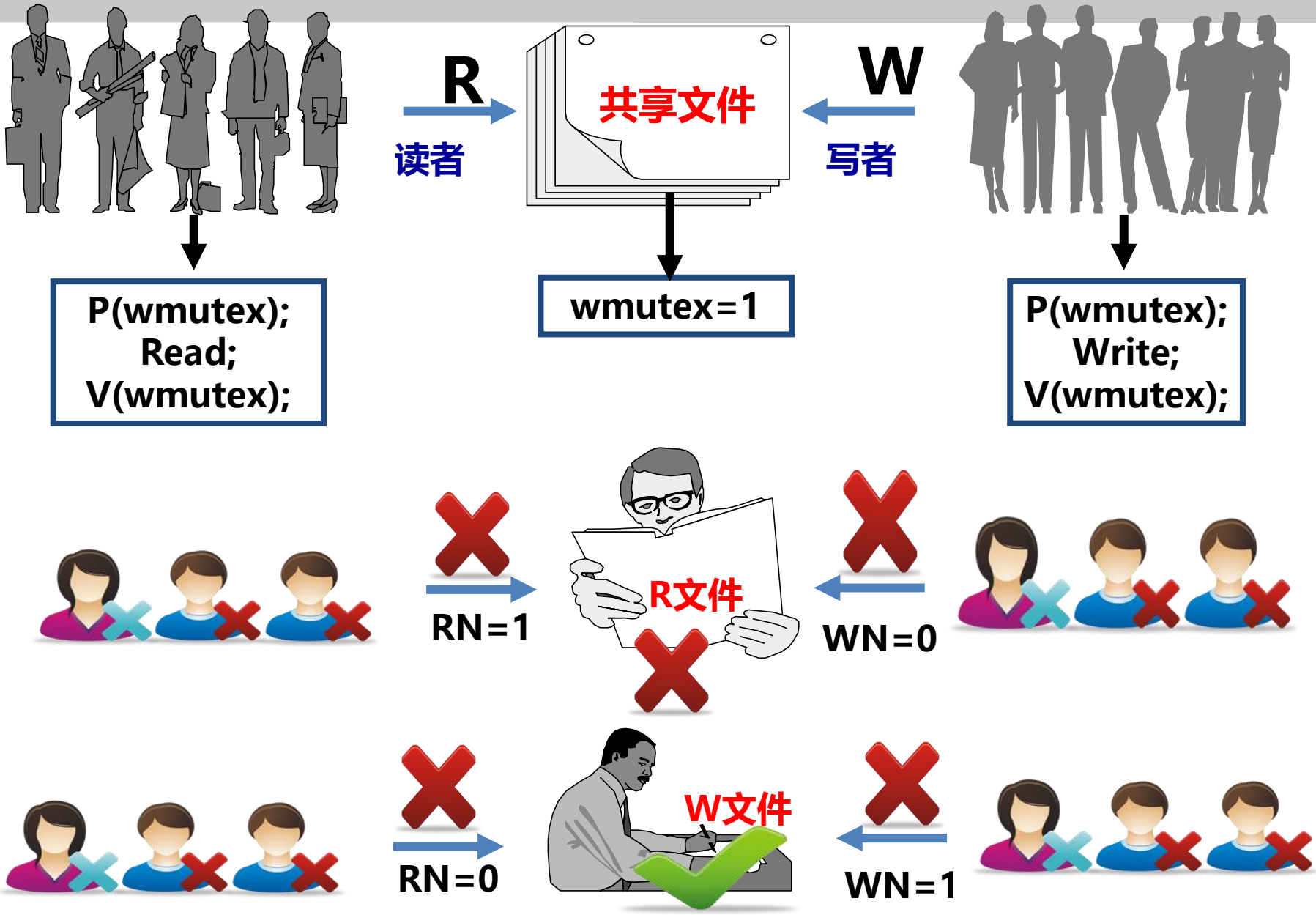
- 算法说明：
  - 由于只有一个读者进程在读时便不允许写者进程去写。因此，仅当rcount=0,表示尚无读者进程在读时，读者进程才需要执行P(mutex)操作。若P(mutex)操作成功，读者进程便可去读，相应地，做rcount+1操作。
  - 同理，仅当读者进程在执行了rcount减1操作后其值为0时，才须执行V(mutex)操作，以便让写者进程去写。

24

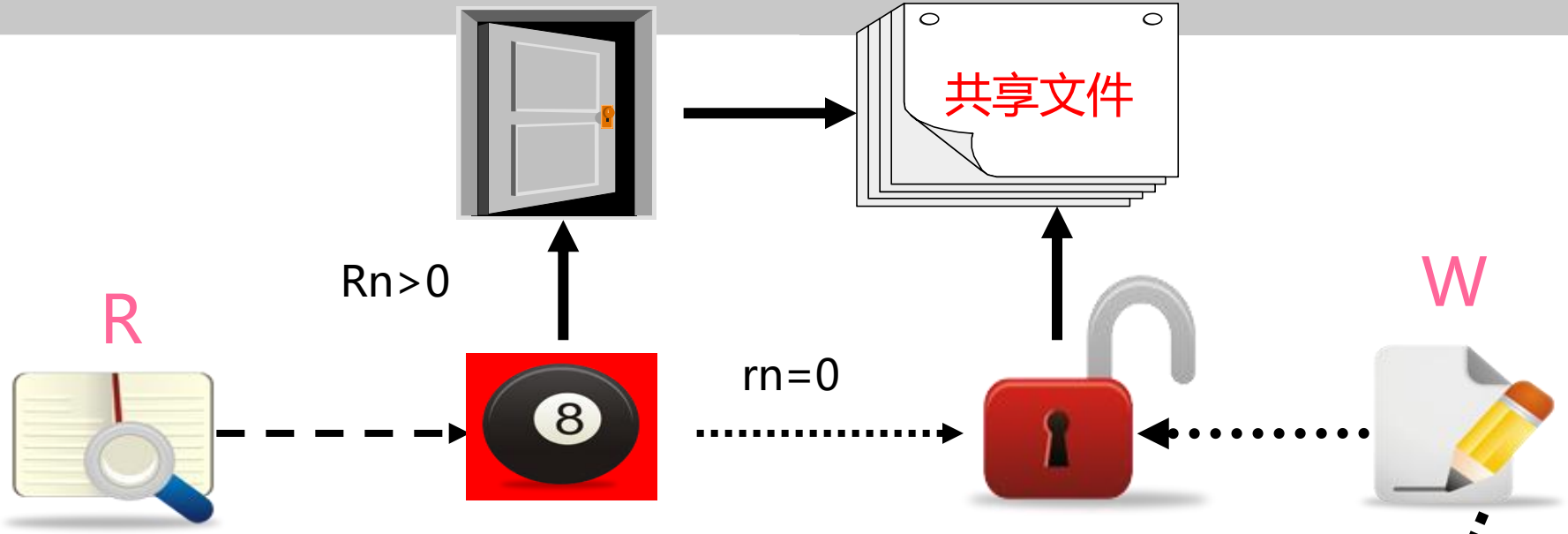
3.5.1 经典进程同步问题之二：读者—写者问题



3.5.1 经 利用记录型信号量解决读者---写者问题



3.5.1 经典进程同步问题之二：读者—写者问题



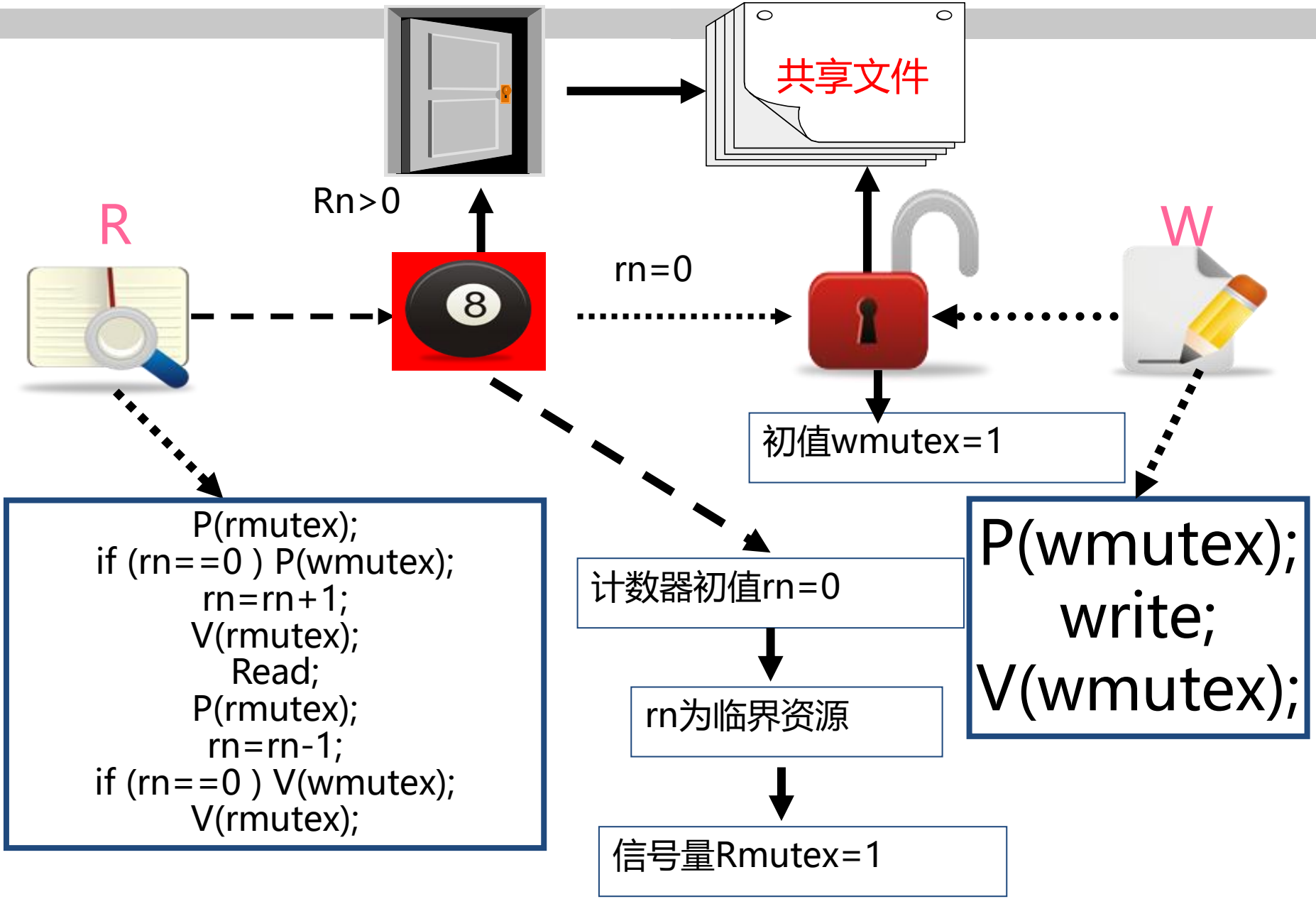
计数器初值rn=0

初值wmutex=1

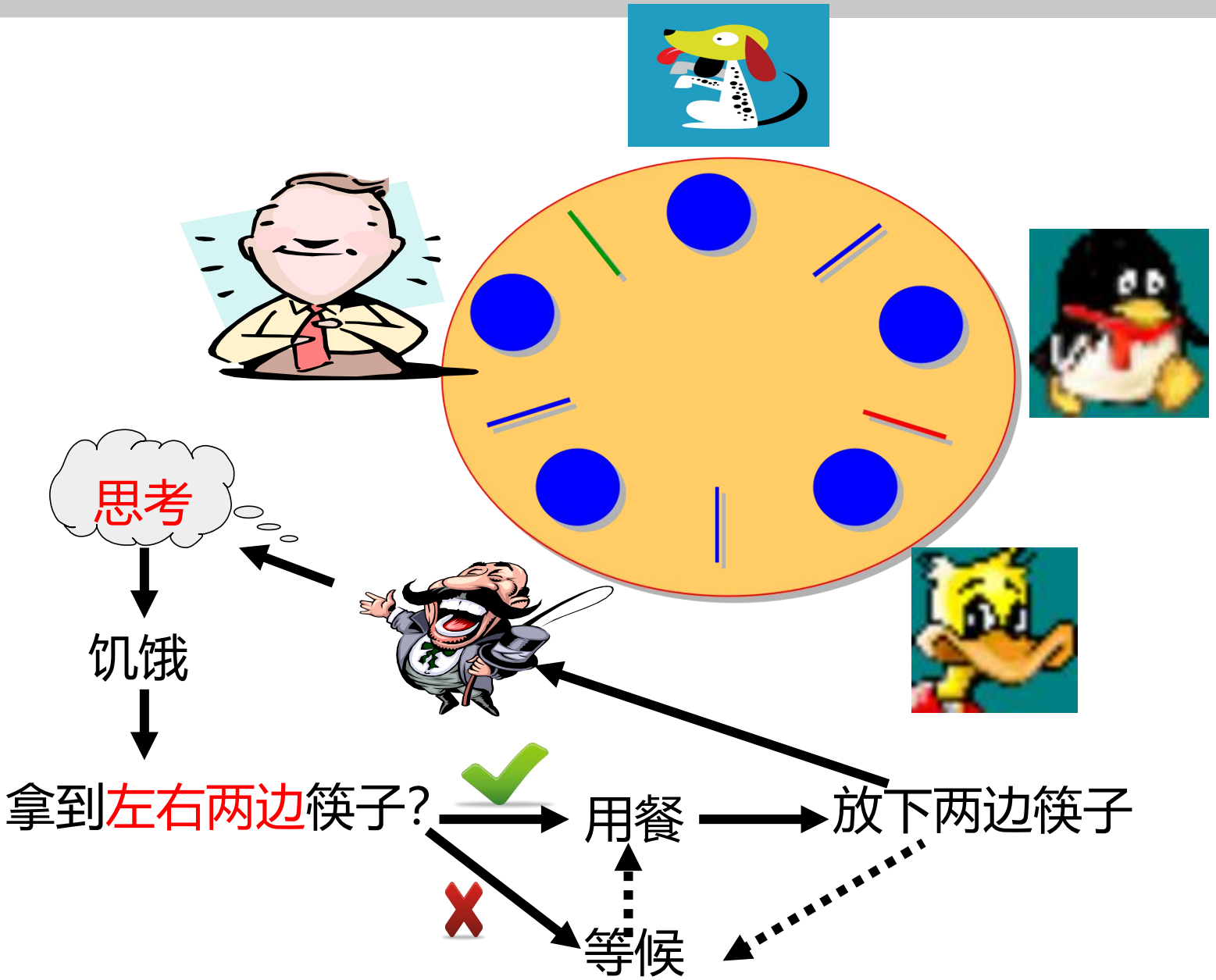
```
if (rn==0 ) P(wmutex);
    rn=rn+1;
    Read;
    rn=rn-1;
if (rn==0 ) V(wmutex);
```

```
P(wmutex);
write;
V(wmutex);
```

3.5.1 经典进程同步问题之二：读者—写者问题



3.5.1 经典进程同步问题之三：哲学家就餐问题



3.5.1 经典进程同步问题之三：哲学家就餐问题

哲学家就餐过程描述如下：
semaphore fork[5] = {1,1,1,1,1};

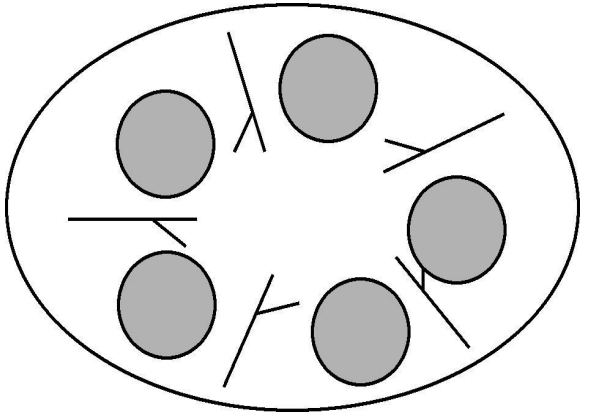
```
do
 {
    P(fork[i]);
    P(fork[(i+1)%5]);
       就餐;
    V(fork[i]);
    V(fork[(i+1)%5]);
      思考;

}while (TRUE);
```

思考1：这样的解法有何问题？

思考2：对左右的叉子是否可用进行验证，这样的修改有何优缺点？

思考3：需要引入几个信号量才能实现最优化的解法呢？



3.5.1 经典进程同步问题之三：哲学家就餐问题

哲学家就餐死锁问题的解决方法

1. 最多允许4个哲学家同时去拿左手边的叉子，在就餐完毕后放 下叉子，其他的哲学家能够就餐
2. 哲学家只有将左右两把叉子同时拿到后才能就餐，否则放弃叉子
3. 奇数者先左后右拿二边的叉子，偶数号哲学家则相反,先右后左拿叉子。
4. 调整哲学家就餐拿起叉子的时间，分间隔时间进行

3.5.1 经典进程同步问题之三：哲学家就餐问题

最多允许4个哲学家同时能够就餐

引入变量的基本方法

计数变量Count=0（初值）

互斥信号量mutex=1（初值）

Count= =5

阻塞信号量S=0 （初值）

进入部分
P(mutex);
Count=count+1;
if (count==5) P(S);
V(mutex);

退出部分
P(mutex);
Count=count-1;
if (count==4) V(S);
V(mutex);

If (count==5) {V(mutex);P(S);};
else V(mutex);

3.5.1 经典进程同步问题之三：哲学家就餐问题

最多允许4个哲学家同时能够就餐

计数变量Count=4（初值）

Count<0

互斥mutex=1（初值）

阻塞信号量S=0 （初值）

进入部分

```
P(mutex);
Count=count-1;
if(count<0){V(mutex);P(S);}el
se  V(mutex);
```

退出部分

```
P(mutex);
Count=count+1;
if(count==0 )V(S);
V(mutex);
```

3.5.1 经典进程同步问题之三：哲学家就餐问题

P(S);

```
P(mutex);
Count=count-1;
If (count<0 ){V(mutex);P(S)};
   else  V(mutex);
```

semaphore; S=4

```
P(fork[i]);
P(fork[(i+1)%5]);
  就餐;
V(fork[i]);
V(fork[(i+1)%5]);
```

V(S);

```
P(mutex);
Count=count+1;
if count=0 then V(S);
V(mutex);
```

思考；}

```
do
 {P(S);
   P(fork[i]);
   P(fork[(i+1)%5]);
     就餐;
   V(fork[i]);
   V(fork[(i+1)%5]);
 V(S);
  思考;
 }while (TRUE);
```

3.5.1 经典进程同步问题之三：哲学家就餐问题

计数变量Count=信号量S1的初值

互斥mutex=1 （初值）

阻塞信号量S=0 （初值）

```
P(mutex);
Count=count-1;
if (count<0){V(mutex);P(S)};
    else V(mutex);
```

↔ P(S1)

```
P(mutex);
Count=count+1;
if (count==0) V(S);
V(mutex);
```

↔ V(S1)

计数变量Count在程序中能参加运算

3.5.1 经典进程同步问题之三：哲学家就餐问题


```
procedure PB  (s)
  {
    if (S.value==1) S.value=0;
                else block(S,L)
   };
```

```
procedure VB  (s)
  {
      if (S.queue.is_empty())  S.value=1 ;
                else wakeup(S,L);
  };
```

按信号量的取值分 — 二元信号量(1,0) / 一般信号量(n)

按实施P，V操作分 — 公用信号量(= 1) / 私用信号量(>=0)

信号量其它应用问题

(例1)吃水果

父(apple)

儿(orange)

orange(full2)=0

plate (empty) =1

果盘

母(orange)

女(apple)

apple (full1)=0

信号量其它应用问题

二个不同的生产者和消费者
程序数与进程数的关系

```
P(plate);
Put apple;
V(apple);
```

```
P(orange);
Take orange;
V(plate);
```

apple

orange

果盘

```
P(plate);
Put orange;
V(orange);
```

orange

apple

```
P(apple);
Take apple;
V(plate);
```

单缓冲取放数问题

信号量其它应用问题

- **变量设置：**
  - dish：表示盘子是否为空，初值=1；
  - apple：表示盘中是否有苹果，初值=0；
  - banana：表示盘中是否有香蕉，初值=0。
- **进程描述：**
  - Father：父亲放置水果的进程；
  - Mother：母亲放置水果的进程；
  - Son：儿子吃水果的进程；
  - Daughter：女儿吃水果的进程。

信号量其它应用问题

dish：盘子是否为空，1
apple：盘中是否有苹果，0
banana：盘中是否有香蕉，0

Father:

P (dish);
将苹果放入盘中；
V (apple);

Mather:

P (dish);
将香蕉放入盘中；
V (banana);

Son:

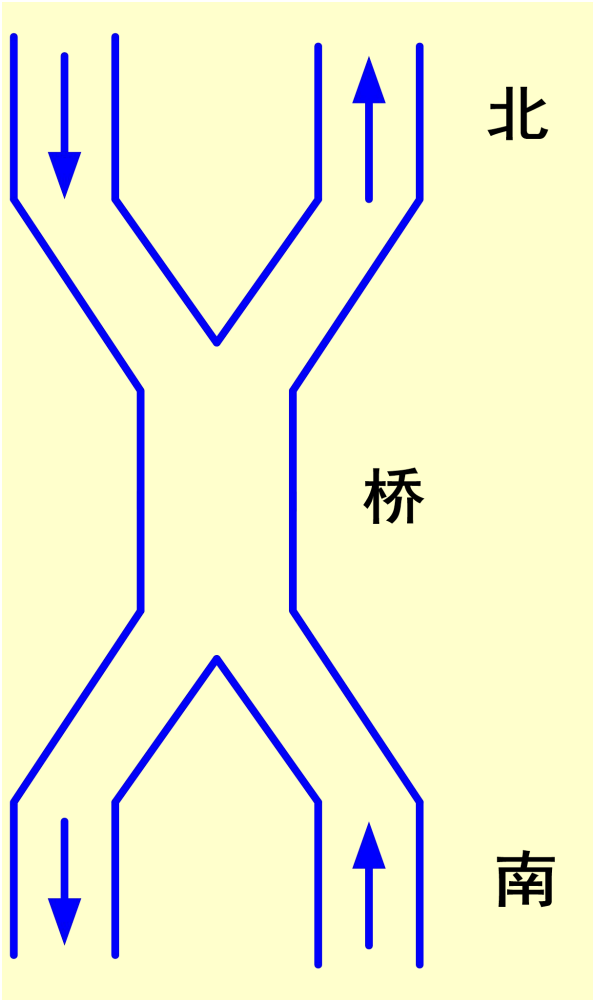
P (banana);
从盘中取出香蕉；
V (dish);
吃香蕉；

Daughter:

P (apple);
从盘中取出苹果；
V (dish);
吃苹果；

(例2)过桥问题

- **题目：**
  - 有一座桥如图所示，车流如图中箭头所示。桥上不允许两车交会，但允许同方向多辆车依次通行（即桥上可以有多个同方向的车）。
  - 用P、V操作实现交通管理，以防止桥上堵塞。



(例2)过桥问题

- **进程描述：**
  - North、South：分别代表北方、南方车辆过桥的进程
- **变量设置：**
  - mutexN：用于实现北方车辆互斥访问变量countN，初值=1；
  - mutexS：用于实现南方车辆互斥访问变量countS，初值=1；
  - wait：用于实现双方申请过桥车辆的排队，初值=1；
  - countN：用于记录当前北方正在过桥及已申请过桥的车辆数，初值=0；
  - countS：用于记录当前南方正在过桥及已申请过桥的车辆数，初值=0；

(例2)过桥问题

mutexN：北方车辆互斥访问变量countN，1

mutexS：南方车辆互斥访问变量countS，1

wait：双方申请过桥车辆的排队，1

countN：北方正在过桥及已申请过桥的车辆数，0

countS：南方正在过桥及已申请过桥的车辆数，0

```
North:
    P ( wait );
    P ( mutexN );
    if (countN==0) P( mutexS)//本方向第一辆车，阻塞对方车辆过桥
    countN ++;
    V ( mutexN );
    V ( wait );
    车辆过桥;
    P ( mutexN );
    countN --;
    if (countN==0) V( mutexS)//本方向最后一辆车允许对方车辆过桥
    V ( mutexN );
```

(例2)过桥问题

mutexN：北方车辆互斥访问变量countN，1
mutexS：南方车辆互斥访问变量countS，1
wait：双方申请过桥车辆的排队，1
countN：北方正在过桥及已申请过桥的车辆数，0
countS：南方正在过桥及已申请过桥的车辆数，0

```
South:
    P ( wait );
    P ( mutexS );
    if (countS==0) P( mutexNS)//本方向第一辆车，阻塞对方车辆过桥
    countS ++;
    V ( mutexS );
    V ( wait );
    车辆过桥;
    P ( mutexS );
    countS --;
    if (countS==0) V( mutexN)//本方向最后一辆车允许对方车辆过桥
    V ( mutexS );
```

（例3）信号量解决理发师问题(1)

- 理发店理有一位理发师、一把理发椅和n把供等候理发的顾客坐的椅子
- 如果没有顾客，理发师便在理发椅上睡觉
- 一个顾客到来时，它必须叫醒理发师
- 如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开

信号量解决理发师问题(2)

- int waiting=0;//等候理发顾客坐的椅子数
- int CHAIRS=N; //为顾客准备的椅子数
- semaphore customers,barbers,mutex;
- customers=0;barbers=0;mutex=1;

信号量解决理发师问题(3)


```
cobegin
process barber( ) {
        while(true) {
                P(customers);    //有顾客吗？若无顾客，理发师睡眠
                P(mutex);        //若有顾客时，进入临界区
                waiting--;       //等候顾客数少一个
            V(barbers);      //理发师准备为顾客理发
                V(mutex);        //退出临界区
                cut_hair();      //理发师正在理发(非临界区)
        }
}
process customer_i( ) {
        P(mutex);            //进入临界区
        if(waiting<CHAIRS) {    //有空椅子吗
                waiting++;       //等候顾客数加1
                V(customers);    //唤醒理发师
                V(mutex);        //退出临界区
                P(barbers);      //理发师忙，顾客坐下等待
                get_haircut();   //否则顾客坐下理发
        }
        else
                V(mutex);        //人满了，走吧!
}
Coend
```

信号量解决理发师问题(3)


```
cobegin
process barber( ) {
while(true) {
 P(customers);//有顾客吗？若无顾客，理发师睡眠
 P(mutex);          //若有顾客时，进入临界区
 waiting--;         //等候顾客数少一个
 V(barbers);        //理发师准备为顾客理发
 V(mutex);          //退出临界区
 cut_hair();        //理发师正在理发(非临界区)
     }
}
```

信号量解决理发师问题(3)


```
process customer_i( ) {
  P(mutex);                 //进入临界区
  if(waiting<CHAIRS) {   //有空椅子吗
  waiting++;             //等候顾客数加1
  V(customers);          //唤醒理发师
  V(mutex);              //退出临界区
  P(barbers);            //理发师忙，顾客坐下等待
  get_haircut();         //否则顾客坐下理发
  }
else
   V(mutex);             //人满了，走吧!
}
coend
```

利用AND信号量解决生产者---消费者问题

Producer(生产者)

```
P(empty);
P(mutex);
buffer(in)=nextp;
in= (in+1) % n;
V(mutex);
V(full);
```

↓

```
SP(empty,mutex);
buffer(in)=nextp;
in= (in+1) % n;
SV(mutex,full);
```

Consumer(消费者)

```
P(full);
P(mutex); nextc=buffer(out);
out=(out+1) % n;
V(mutex);
V(empty);
```

↓

```
SP(full,mutex);
nextc=buffer(out);
out=(out+1) % n;
SV(mutex,empty);
```

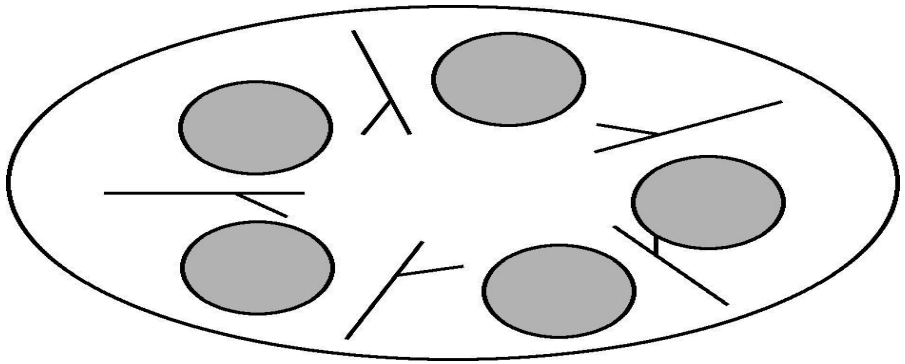
利用AND信号量机制解决哲学家进餐问题


```
do
 {
   P(fork[i]);
   P(fork[(i+1 )% 5]);
    就餐;
   V(fork[i]);
   V(fork[(i+1 )% 5]);
    思考;
 } while(TRUE)
```

```
semaphore fork[5] ={1,1,1,1,1}
```

```
SP(fork[(i+1) % 5],fork[i]);
```

```
SV(fork[(i+1) % 5],fork[i]);
```



小结

进程同步

方法:硬件和软件

信号量:整型,记录型(P,V操作)，AND型与一般型

解决：互斥,前趋,同步

经典进程同步问题

进程通信

- 共享存储区
- 管道通信
- 消息通信
- 客服机-服务器

- 生产者与消费者
- 读者与写者
- 哲学家进餐问题

基本概念

进程同步互斥、临界区、临界资源、同步机制应遵循四条的准则,信号量,管程

TRANSITION PAGE

第二章 进程的描述与控制

2.1 前趋图和程序的执行
2.2 进程的描述
2.3 进程控制
2.4 进程同步基本概念
2.5 管程机制
2.6 经典的同步问题
2.7 进程通信
2.7 线程及其实现

信号量机制的问题

问题

- 需要程序员实现，编程困难
- 维护困难
- 容易出错
  - wait/signal位置错
  - wait/signal不配对

解决方法

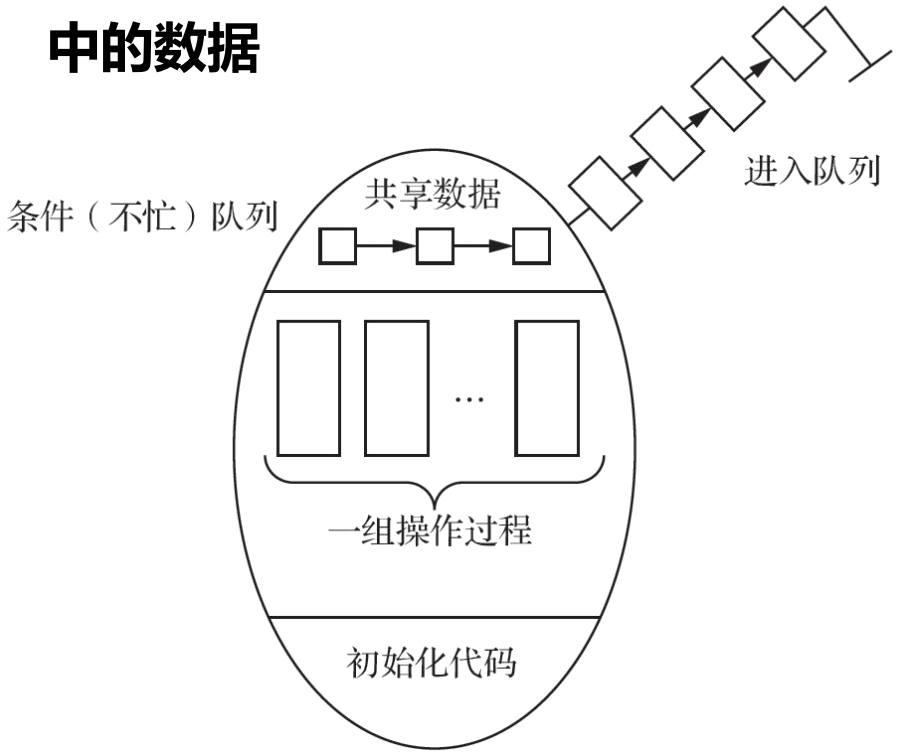
- 由编程语言解决同步互斥问题
- 管程（1970s, Hoare和Hansen）

信号量：**分散式**
管　　程：**集中式**

管程

管程定义

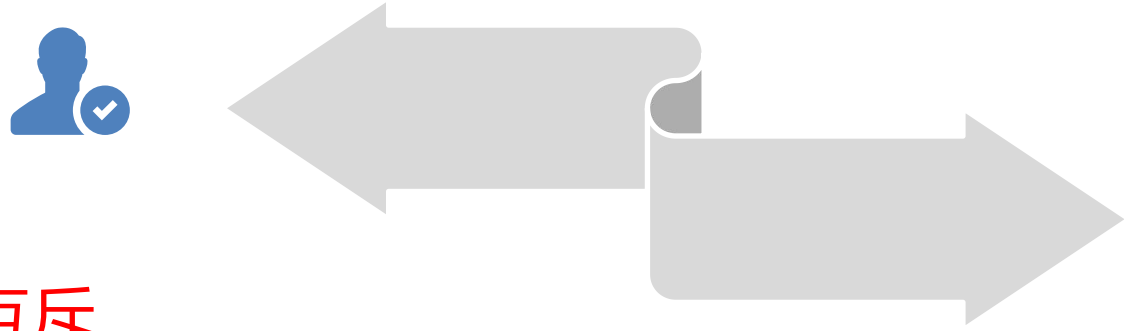
- 一个管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能**同步进程**和**改变管程中的数据**



语法描述如下：

```
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*/
    ……
    }
}
```

管程功能



互斥

- 管程中的变量只能被管程中的操作访问
- 任何时候只有一个进程在管程中操作
- 类似临界区
- 由编译器完成

同步

- 条件变量
- 唤醒和阻塞操作

条件变量

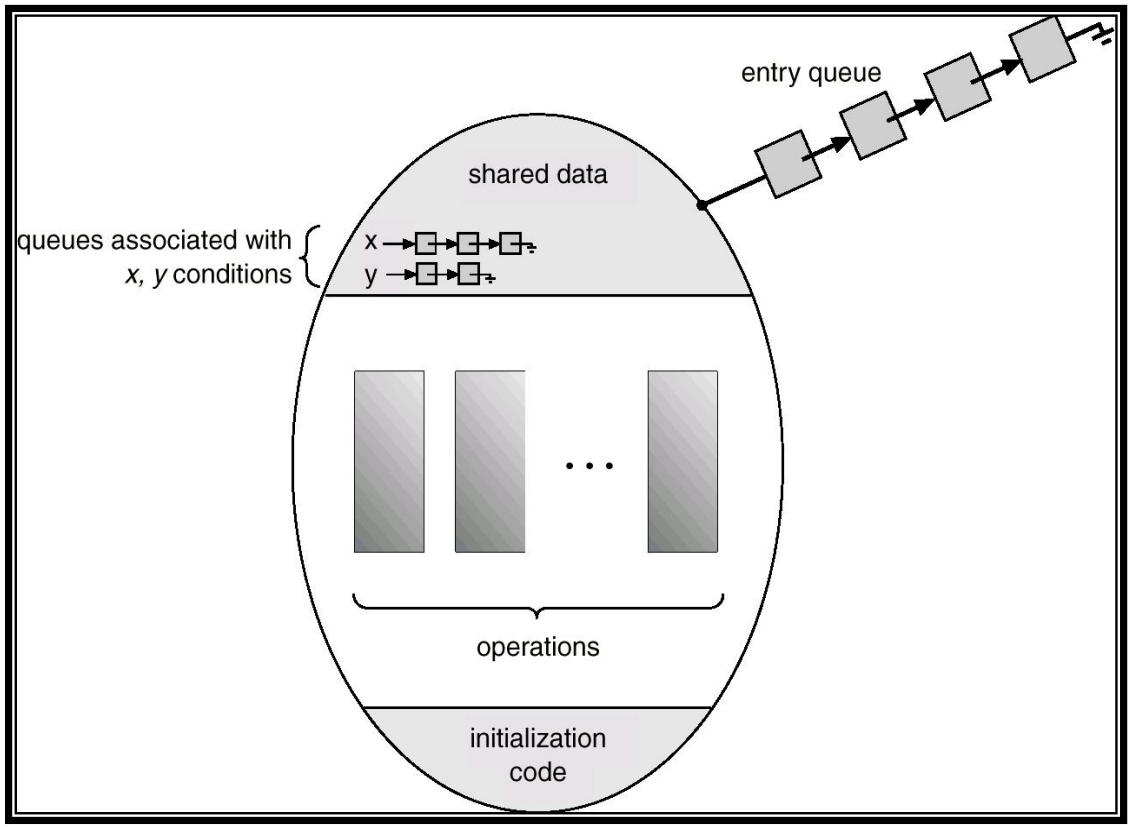
condition x, y;

条件变量的操作

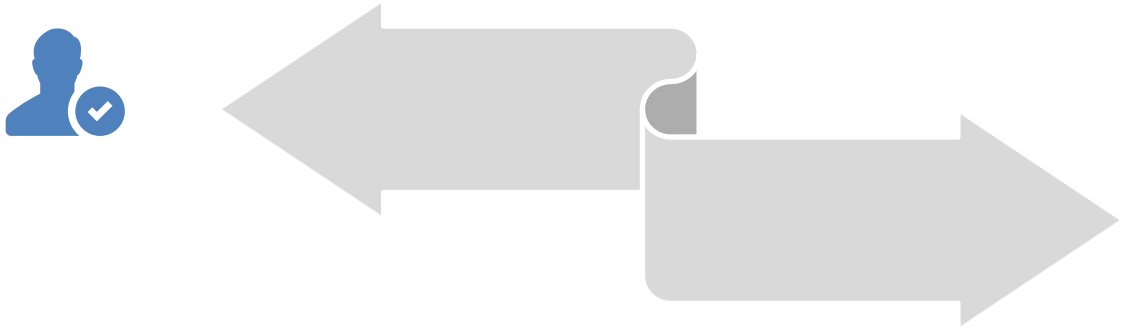
- 阻塞操作：wait
- 唤醒操作：signal

x.wait(): 进程阻塞直到另外一个进程调用x.signal()

x.signal()：唤醒另外一个进程



条件变量问题



管程内可能存在不止1个进程。

- 例如：进程P调用signal操作唤醒进程Q后。

存在的可能处理方式：

- P等待，直到Q离开管程或等待另一条件（Hoare）。
- Q等待，直到P离开管程或等待另一条件（Hansen）。

线程的概念

时间

- 60年代中期：提出进程概念
- 80年代中期：提出线程概念
- 90年代后：多处理机系统引入线程

引入进程的目的

- 使多个程序并发执行
- 提高资源利用率及系统吞吐量

进程的2个基本属性：

- 进程是一个可拥有资源的独立单位；
- 进程是一个可独立调度和分派的基本单位。

线程的概念

提出线程的目的

- 减少程序在并发执行时所付出的时空开销
- 使OS具有更好的并发性
- 适用于SMP结构的计算机系统

进程是拥有资源的基本单位（传统进程称为重型进程）

线程作为调度和分派的基本单位（又称为轻型进程）

线程和进程的比较(1)

01 调度的基本单位

- 在传统的OS中，拥有资源、独立调度和分派的基本单位都是进程；
- 在引入线程的OS中，线程作为调度和分派的基本单位，进程作为资源拥有的基本单位；
- 在同一进程中，线程的切换不会引起进程切换，在由一个进程中的线程切换到另一个进程中的线程时，将会引起进程切换。

02 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，也可并发执行。

线程和进程的比较(2)

03 拥有资源

- 进程是系统中拥有资源的一个基本单位，它可以拥有资源。
- 线程本身不拥有系统资源，仅有一点保证独立运行的资源。
- 允许多个线程共享其隶属进程所拥有的资源。

04 独立性

- 同一进程中的不同线程之间的独立性要比不同进程之间的独立性低得多。

线程和进程的比较(2)

05 系统开销

- 在创建或撤消进程时，OS所付出的开销将显著大于创建或撤消线程时的开销。
- 线程切换的代价远低于进程切换的代价。
- 同一进程中的多个线程之间的同步和通信也比进程的简单。

06 支持多处理机系统

线程的状态和线程控制块

线程状态

- 执行态、就绪态、阻塞态
- 线程状态转换与进程状态转换一样

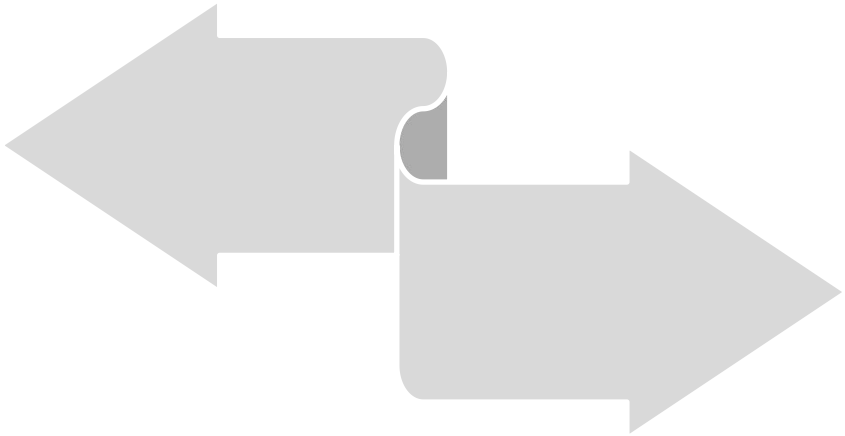
线程控制块（thread control block，TCB）

- 线程标识符、一组寄存器、线程运行状态、优先级、线程专有存储区、信号屏蔽、堆栈指针

线程的实现

实现方式：

- 内核支持线程KST
- 用户级线程ULT
- 组合方式



具体实现：

- 内核支持线程的实现（利用系统调用）
- 用户级线程的实现（借助中间系统）

内核支持线程KST

在内核空间实现

优点：

- 在多处理机系统中，内核可同时调度同一进程的多个线程
- 如一个线程阻塞了，内核可调度其他线程(同一或其他进程)。
- 线程的切换比较快，开销小。
- 内核本身可采用多线程技术，提高执行速度和效率。

缺点：

- 对用户线程切换，开销较大。

用户级线程ULT

在用户空间实现

优点：

- 线程切换不需要转换到内核空间。
- 调度算法可以是进程专用的。
- 线程的实现与OS平台无关。

缺点：

- 系统调用的阻塞问题。
- 多线程应用不能利用多处理机进行多重处理的优点。

ULT与KST组合方式

- 多对一模型

- 一对一模型

- 多对多模型

多对一模型

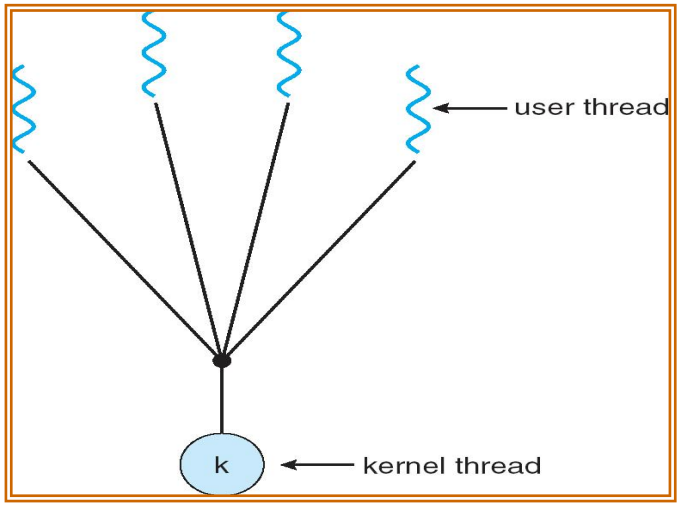
01 多个用户级线程映射到一个内核线程。

02 多个线程不能并行运行在多个处理器上。

03 线程管理在用户态执行，因此是高效的，但一个线程的阻塞系统调用会导致整个进程的阻塞 。

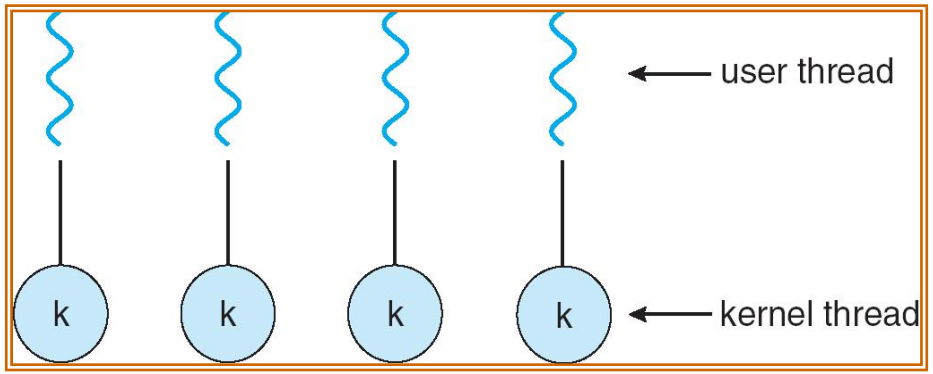
04 用于不支持内核线程的系统中。

05 例子:
- Solaris Green Threads
- GNU Portable Threads



一对一模型

01 每个用户级线程映射到一个内核线程。



02 比多对一模型有更好的并发性。

03 允许多个线程并行运行在多个处理器上。

04 创建一个ULT需要创建一个KLT，效率较差。

05 例子：

- Windows 95/98/NT/XP/2000
- Linux
- Solaris 9 and later
- OS/2

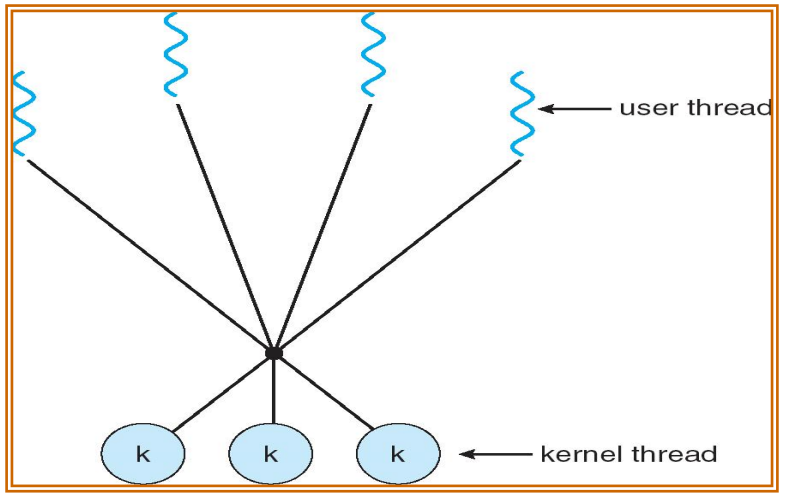
多对多模型

多个用户级线程映射为相等或小于数目的内核线程。

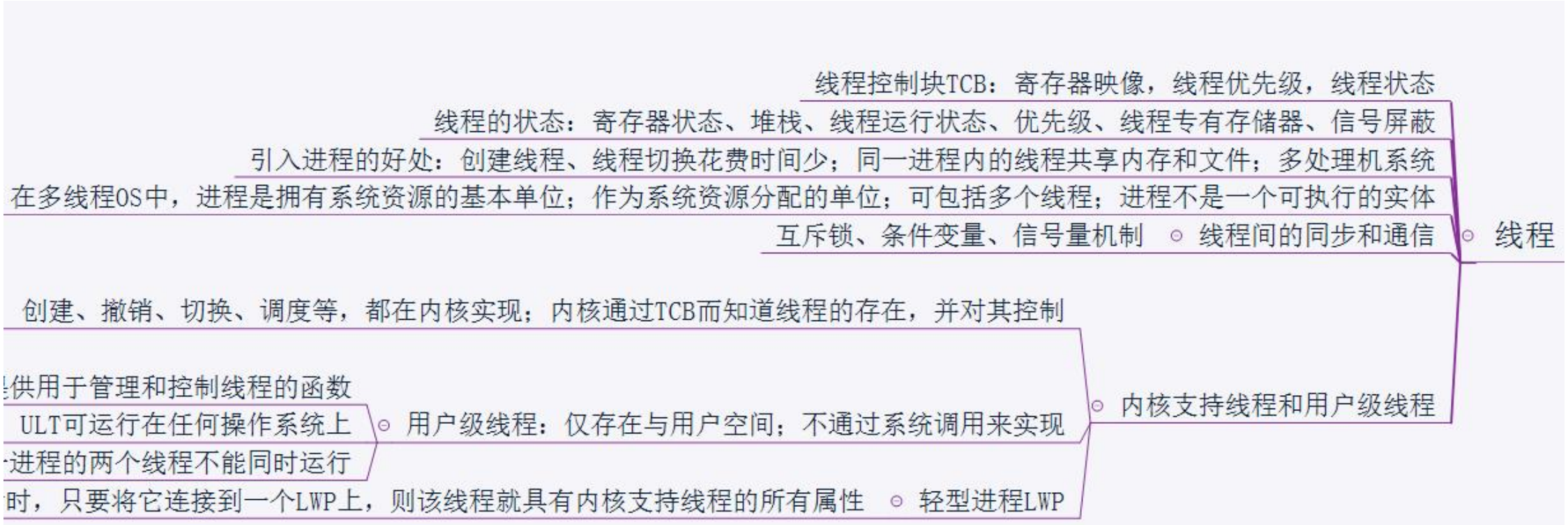
允许操作系统创建足够多的KLT。

例子：

- Solaris 9 以前的版本；
- 带有ThreadFiber开发包的Windows NT/2000 。



小结


线程
线程控制块TCB：寄存器映像，线程优先级，线程状态
线程的状态：寄存器状态、堆栈、线程运行状态、优先级、线程专有存储器、信号屏蔽
引入进程的好处：创建线程、线程切换花费时间少；同一进程内的线程共享内存和文件；多处理机系统
在多线程OS中，进程是拥有系统资源的基本单位；作为系统资源分配的单位；可包括多个线程；进程不是一个可执行的实体
线程间的同步和通信
互斥锁、条件变量、信号量机制
内核支持线程和用户级线程
创建、撤销、切换、调度等，都在内核实现；内核通过TCB而知道线程的存在，并对其控制
用户级线程：仅存在与用户空间；不通过系统调用来实现
供用于管理和控制线程的函数
ULT可运行在任何操作系统上
进程的两个线程不能同时运行
轻型进程LWP
时，只要将它连接到一个LWP上，则该线程就具有内核支持线程的所有属性

- 具体看思维导图

小结

程序顺序执行时的特征

- 1)顺序性：前一操作完成才能进入下一操作
- 2)封闭性：运行时独占资源,结果不受外界影响；即资源的状态只能由本程序改变
- 3)可再现性：初始条件相同，结果唯一(结果与程序的执行次数和速度无关（停—走—停—走）

程序并发执行时的特征

- 1)间断性：出现执行——暂停——执行这种间断性的活动规律
原因：相互制约（直接 / 间接）
- 2)失去封闭性：资源被共享，状态由多个共享者改变
- 3)不可再现性：初始条件相同，结果不唯一
(结果与执行程序的速度相关)

小结

进程的定义：进程是可并发执行的、具有独立功能的程序在一定数据集合上的一次执行过程，是操作系统进行资源分配和调度的基本单位。

进程是程序的执行过程,创建而产生、调度而执行、撤消而消亡

按各自独立的、不可预知的速度向前推进，导致程序执行的不可再现性。

1)动态性	2)并发性	3)独立性	4)异步性	5)结构特征

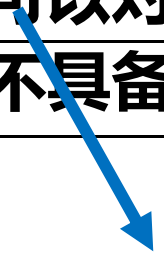
进程可以并发执行,而程序却不能

进程实体由程序段、数据段和进程控制块组成,又称为“进程映像”。

独立运行 / 资源分配 / 调度的基本单位

小结

进程与程序的区别	
动态	静态（定义）
暂时性（具有生命期）	永久性
并发性、独立性、结构特征	程序都没有
程序+数据+PCB块	程序+数据
一个进程可以涉及多个程序	一个程序可以对应多个进程
异步运行，会相互制约	程序不具备此特征

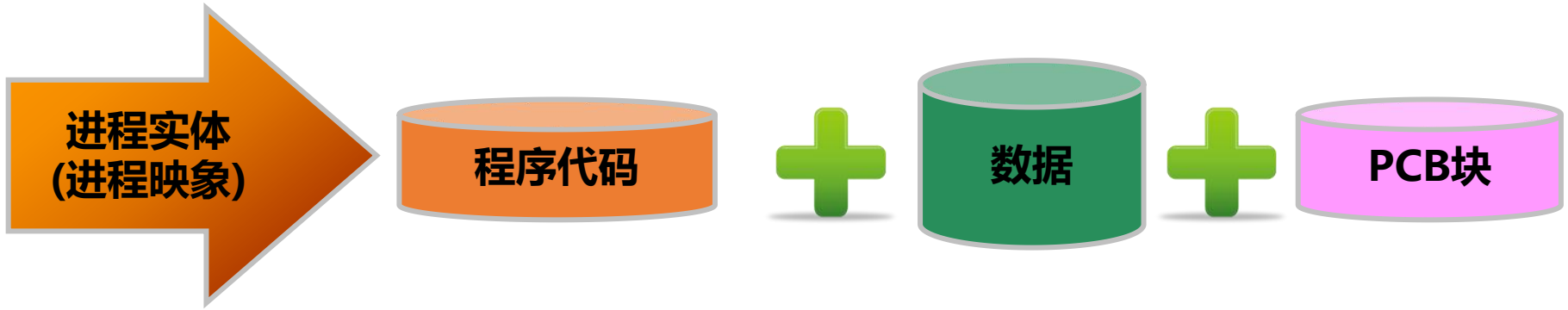


（不同的进程可以包含相同的程序）

小结

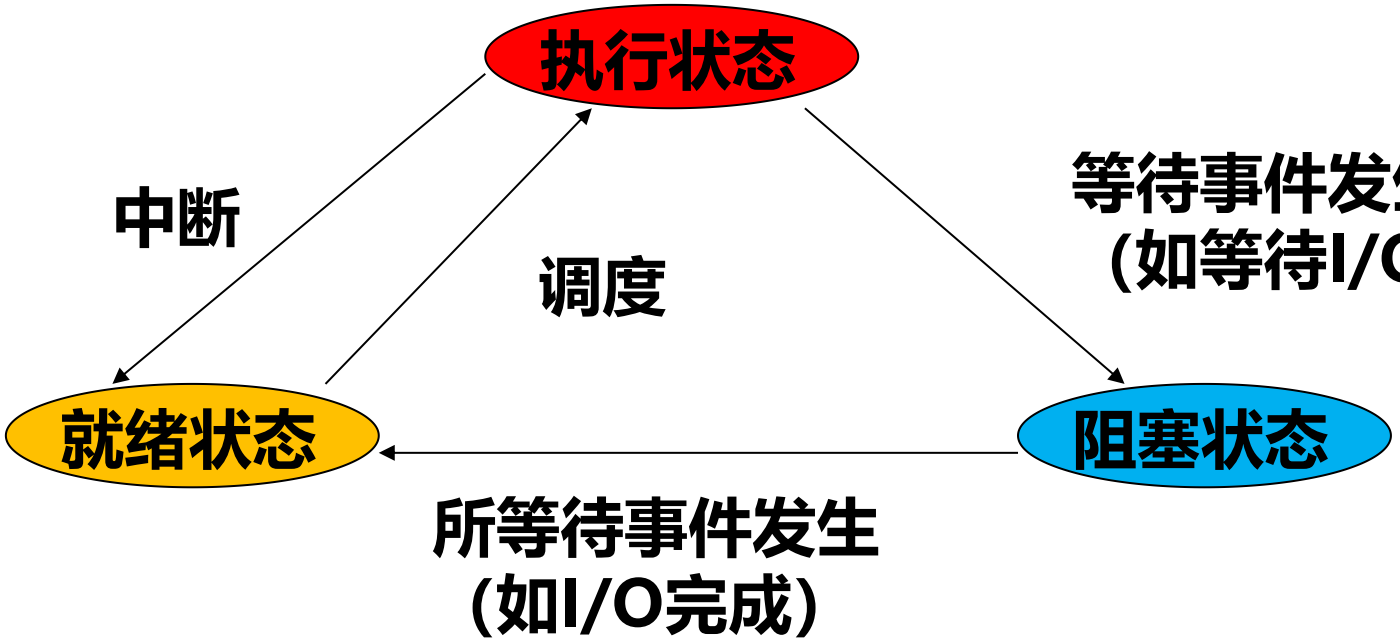
进程控制块PCB(process control block)

是进程的唯一实体,系统根据PCB块而感知进程的存在,即PCB块是进程存在的唯一标识。



PCB表的物理组织：常见的是线性方式、链接方式和索引方式。

小结



执行态：此时正用CPU；

就绪态：可运行,但未分到CPU；

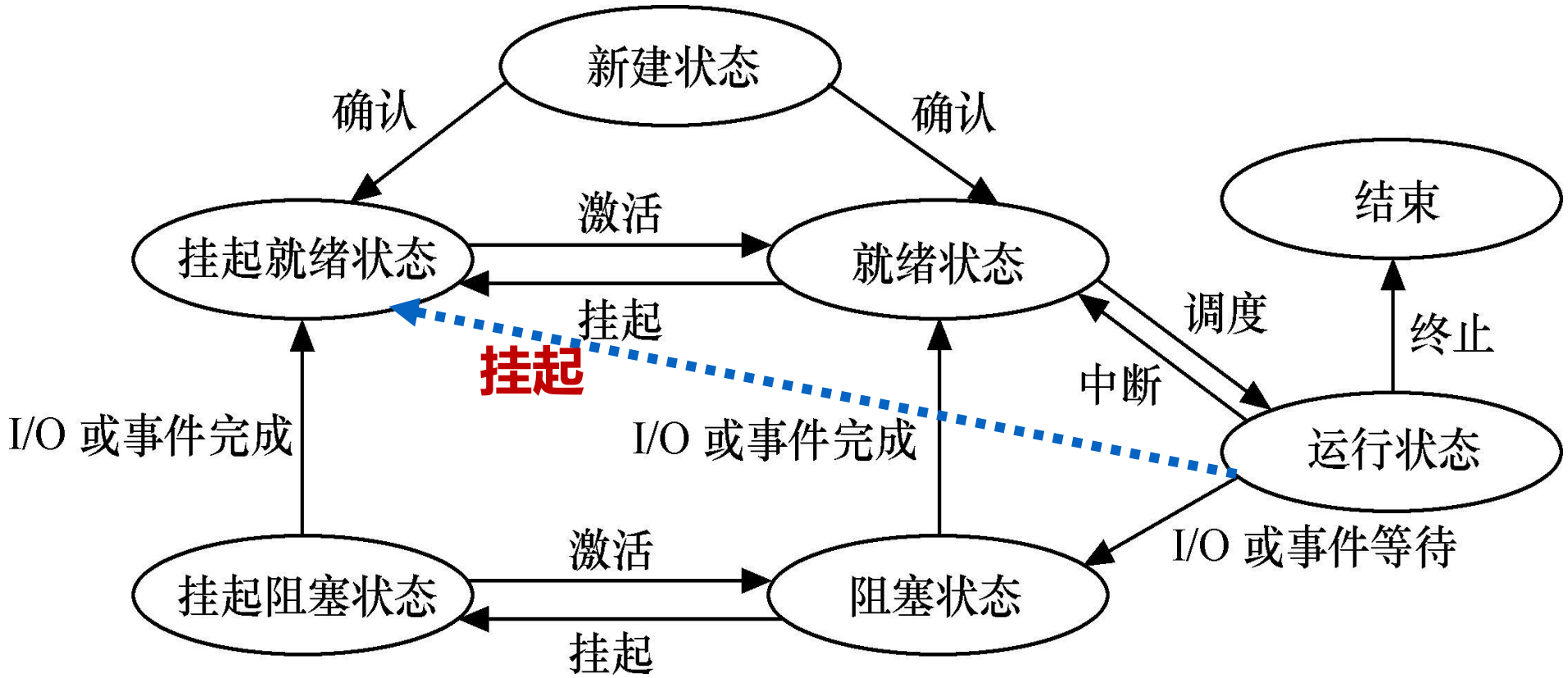
阻塞态：不能运行,等待某个外部事件发生。

在一定条件下,进程状态才发生转换

小结

进程可以在：新建，就绪，阻塞，运行状态被挂起。

进程只能由操作系统，父进程，进程自身挂起



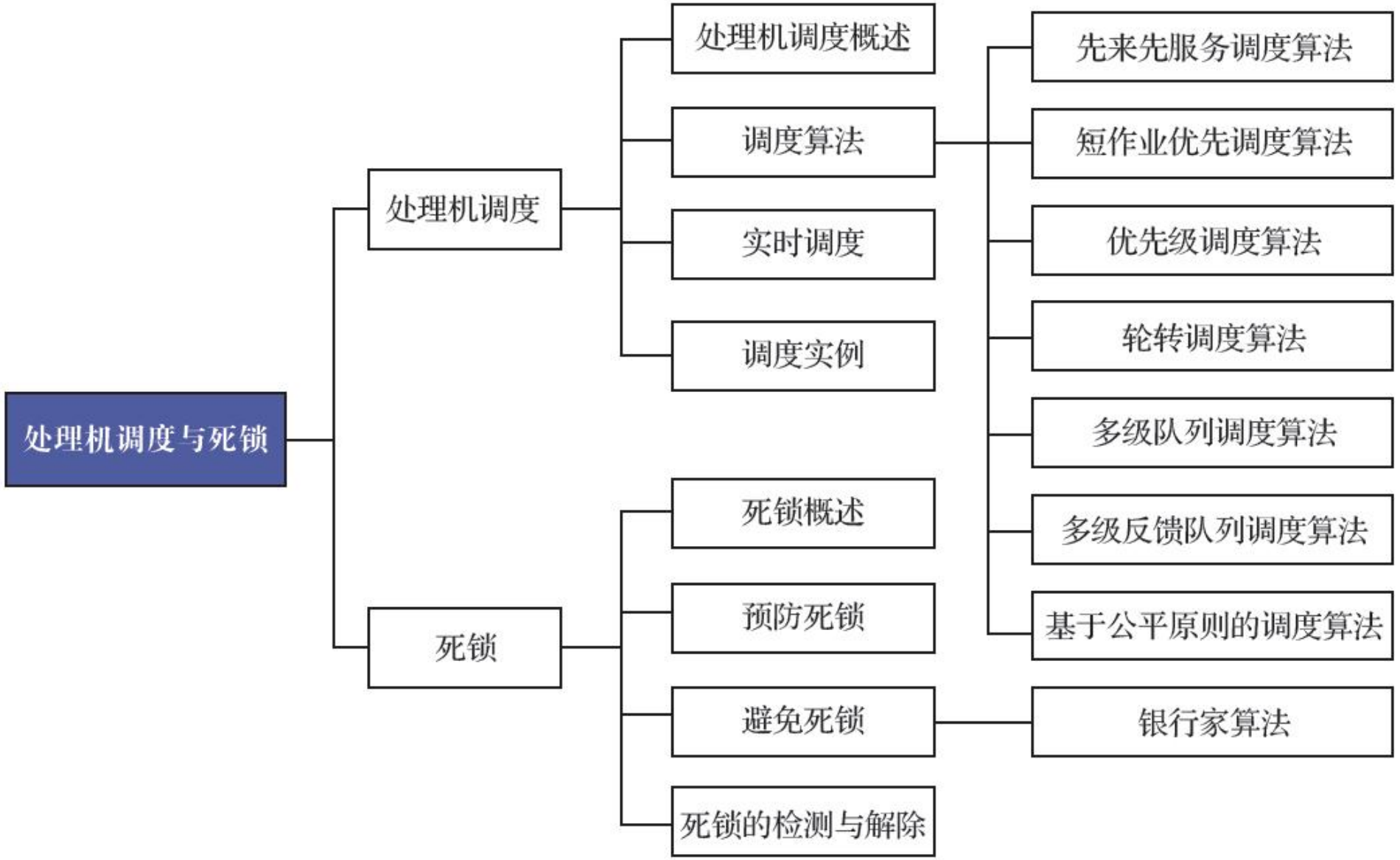
进程状态转换

TRANSITION PAGE

第三章 处理机调度与死锁

3.1 调度的层次算法选择的原则
3.2 作业和进程的关系
3.3 作业的管理与调度
3.4 进程调度的功能和类型
3.5 作业调度和进程调度算法

本章主要内容


处理机调度与死锁
处理机调度
处理机调度概述
调度算法
先来先服务调度算法
短作业优先调度算法
优先级调度算法
轮转调度算法
多级队列调度算法
多级反馈队列调度算法
基于公平原则的调度算法
实时调度
调度实例
死锁
死锁概述
预防死锁
避免死锁
银行家算法
死锁的检测与解除

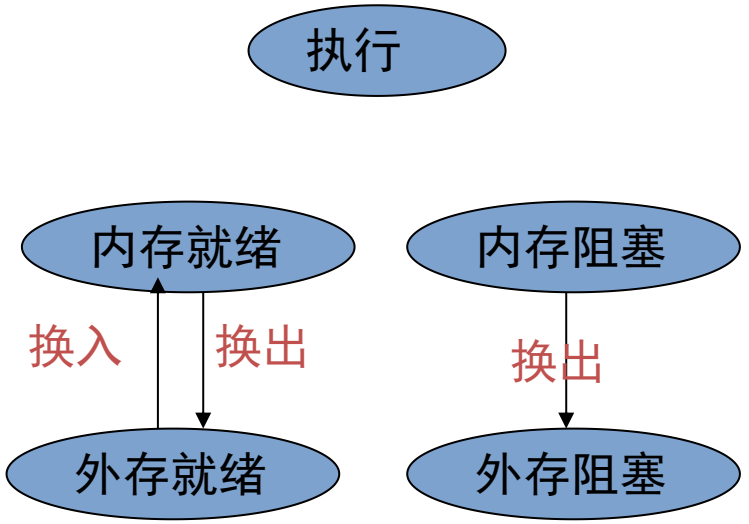
进程调度要解决的问题：

- What：按什么原则分配CPU—进程调度算法
- When：何时分配CPU—进程调度的时机
- How：如何分配CPU—CPU调度过程（进程的上下文切换）

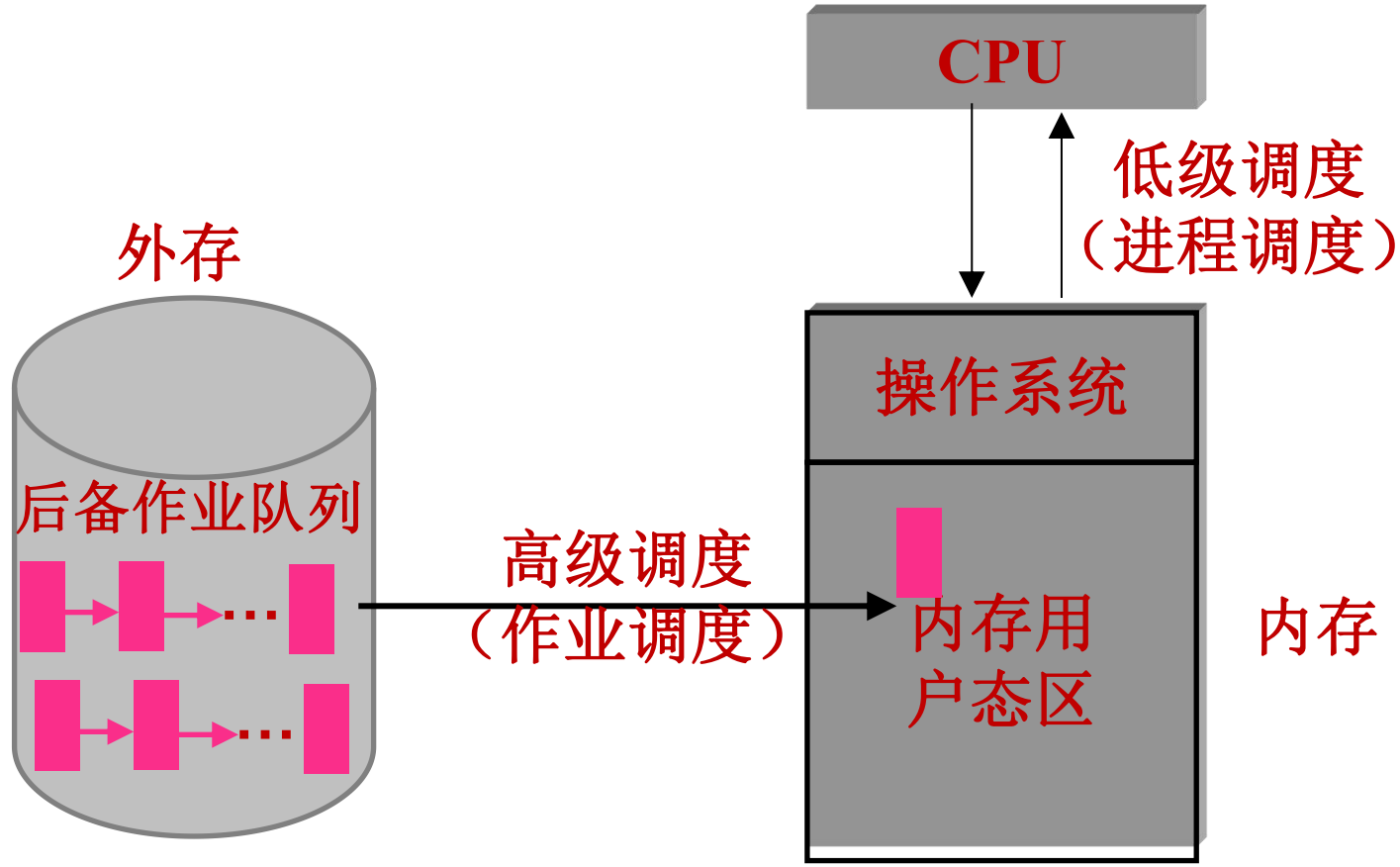
3.1.1 处理机调度的层次

作业从进入系统成为后备作业开始，直到运行结束退出系统为止，需经历不同级别的调度。

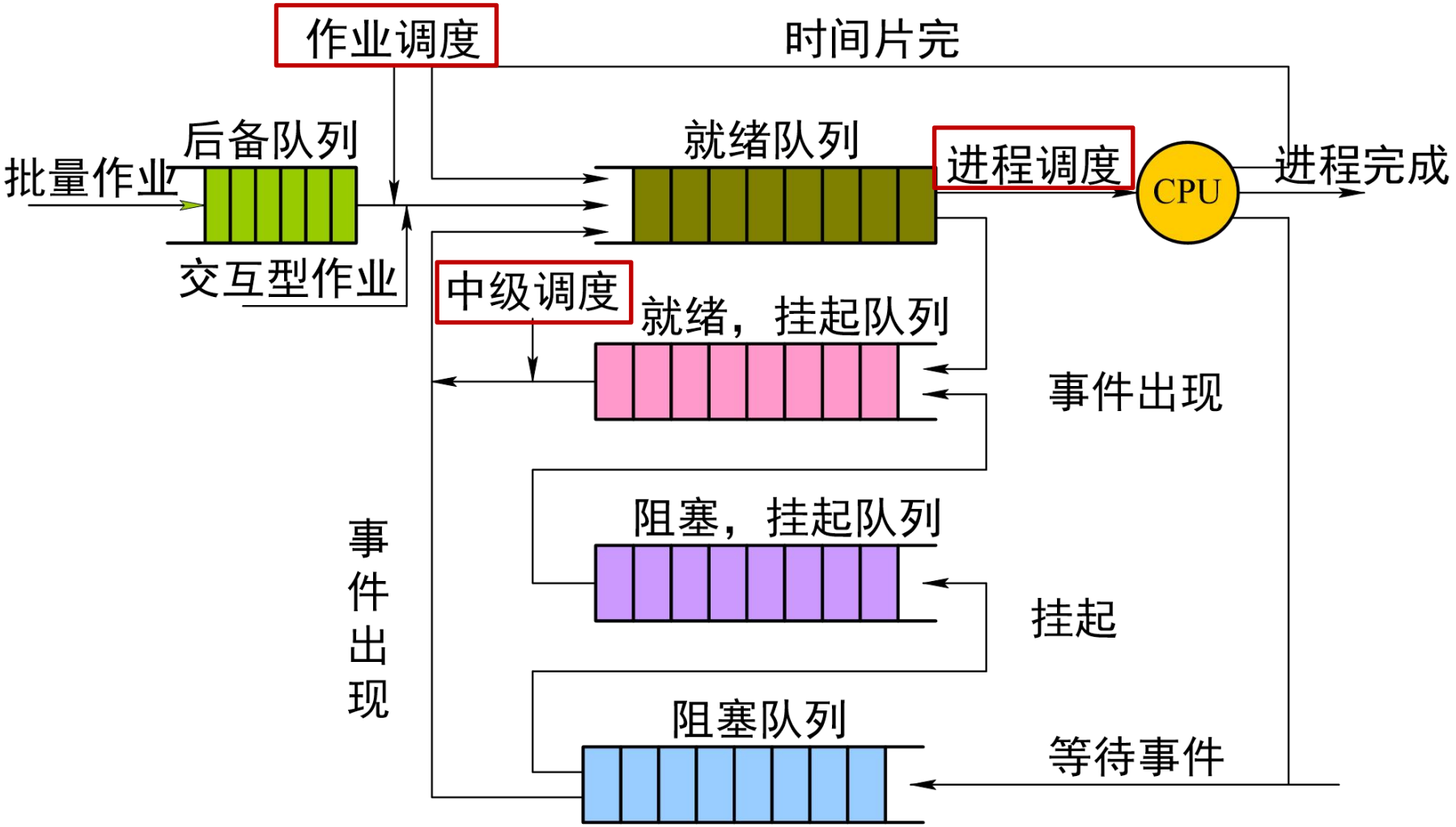
- 高级调度——作业调度
- 中级调度——换入换出
- 低级调度——进程调度



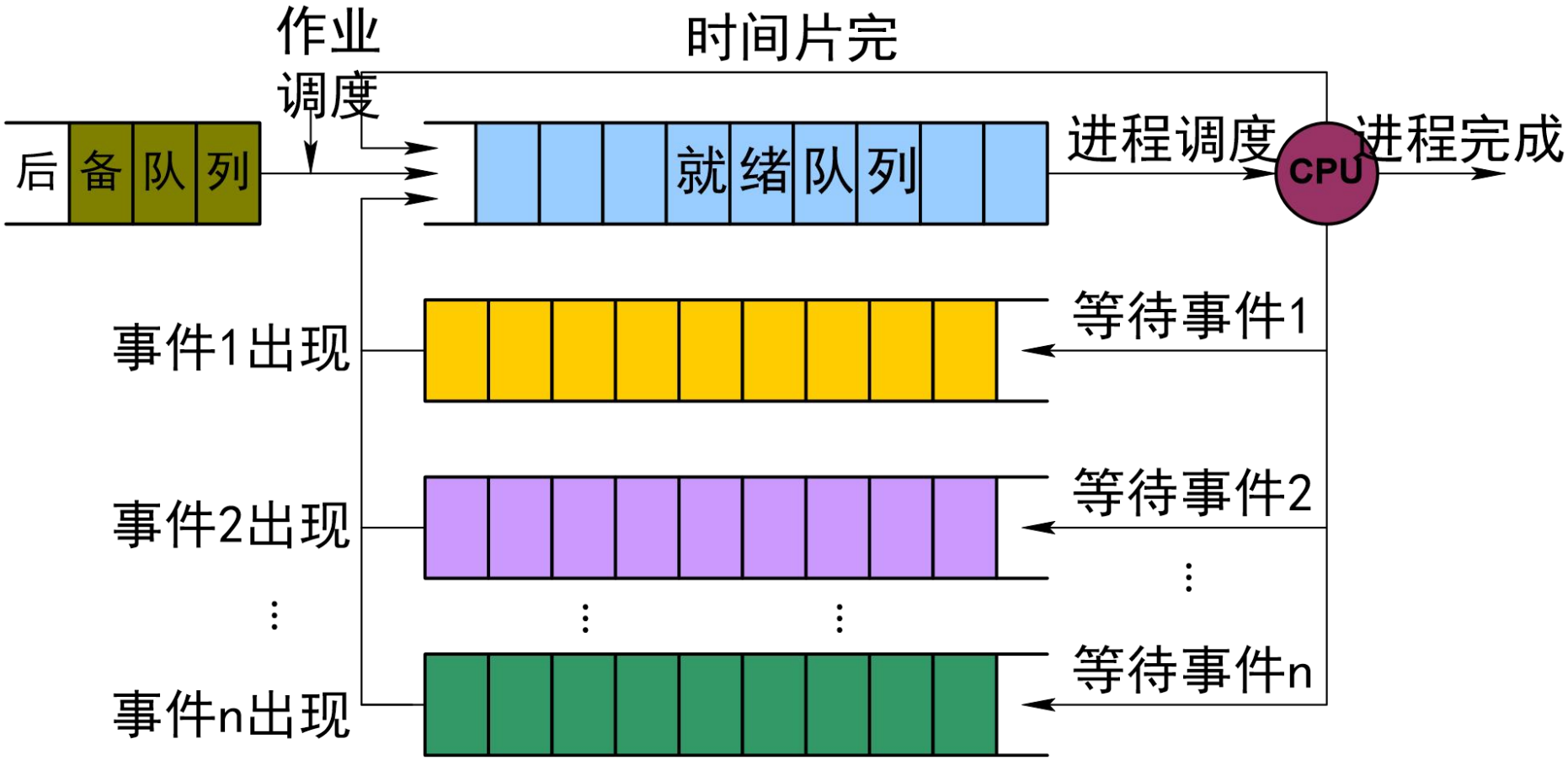
3.1.1 处理机调度的层次



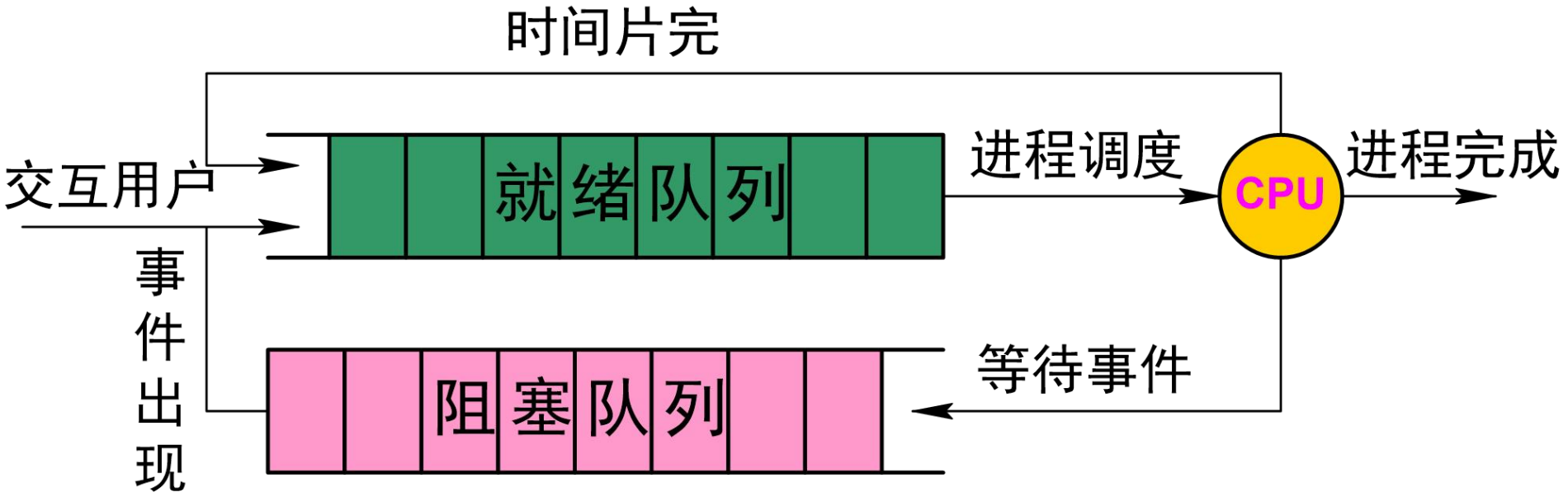
3.1.2 处理机三级调度模型



3.1.3 处理机两级调度模型



3.1.3 仅有进程调度的调度模型



3.1.4 选择调度算法的准则

- 用户
  - 周转时间短（批处理系统）
  - 响应时间快（分时系统）
  - 优先权
  - 截止时间的保证（实时系统）
- 系统
  - 系统吞吐量高（批处理系统）
  - 处理机效率好（批处理系统）
  - 资源利用的平衡（实时系统）

3.1.4 选择调度算法的准则——面向用户的准则

周转时间：从作业提交给系统开始，直到作业完成的时间间隔。

- 应使作业周转时间或平均作业周转时间尽可能短。
- 这是批处理系统衡量调度性能的一个重要指标。

3.2.1 选择调度算法的准则——面向用户的准则

周转时间： $T_i = T_f(\text{完成}) - T_t(\text{提交})$

平均周转时间： $T = 1/n \sum_{i=1}^{n} T_i$

带权周转时间： $W = T_i / T_s$

平均带权周转时间： $W = 1/n \sum_{i=1}^{n} \frac{T_i}{T_s}$

3.2.1 选择调度算法的准则——面向用户的准则

响应时间：用户从键盘提交一个请求到首次获得响应的时间。

- 输入时CPU处理请求的时间
- 处理后相应结果回送终端显示的时间等。
- 使交互式用户的响应时间尽可能短，或尽快处理实时任务。
- 这是分时系统和实时系统衡量调度性能的一个重要指标。

截止时间的保证：开始截止时间和完成截止时间

优先权准则：可以使关键任务达到更好的指标。

公平性：不因作业或进程本身的特性而使上述指标过分恶化。如长作业等待很长时间。

3.2.2 选择调度算法的准则——面向系统的准则

- **系统吞吐量高**：单位时间内所完成的作业数——*批处理系统*
- **处理机利用率好**：——*大中型主机*
  - CPU利用率=CPU有效工作时间/CPU总的运行时间
  - CPU总的运行时间=CPU有效工作时间+CPU空闲等待时间
- **各类资源的平衡利用**：如CPU繁忙的作业和I/O繁忙（指次数多，每次时间短）的作业搭配——*大中型主机*
  - 确保每个用户每个进程获得合理的CPU份额或其他资源份额，不会出现饿死情况。

3.3.1 作业和进程的关系

作业(JOB)
用户提交给操作系统计算的一个独立任务。

作业步(Job Step)
对作业的每个加工步骤。

作业组织
作业的提交、收容、执行和完成。

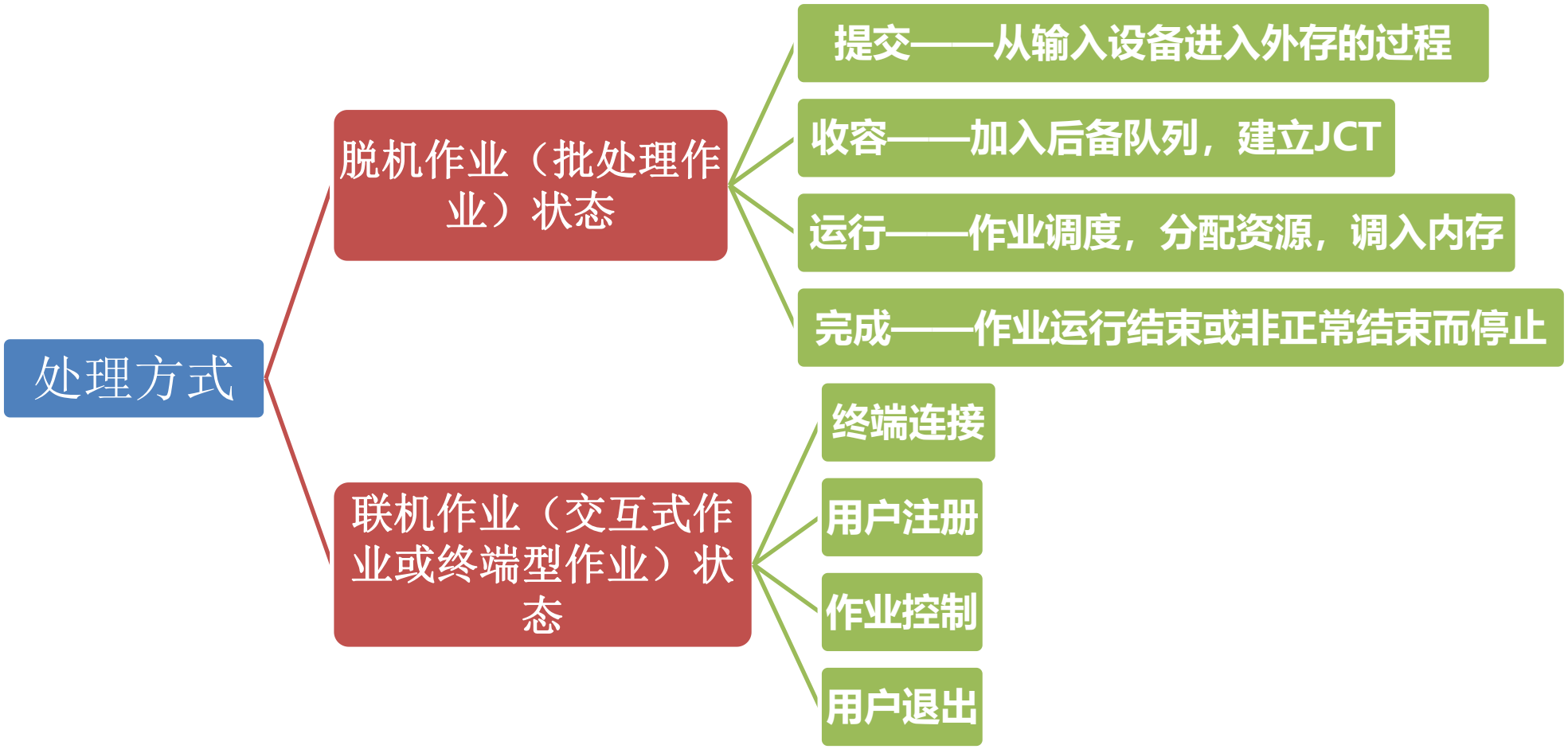
例如： user.c user.obj user.exe

编辑 → 编译 → 连接 → 运行

作业步之间的关系

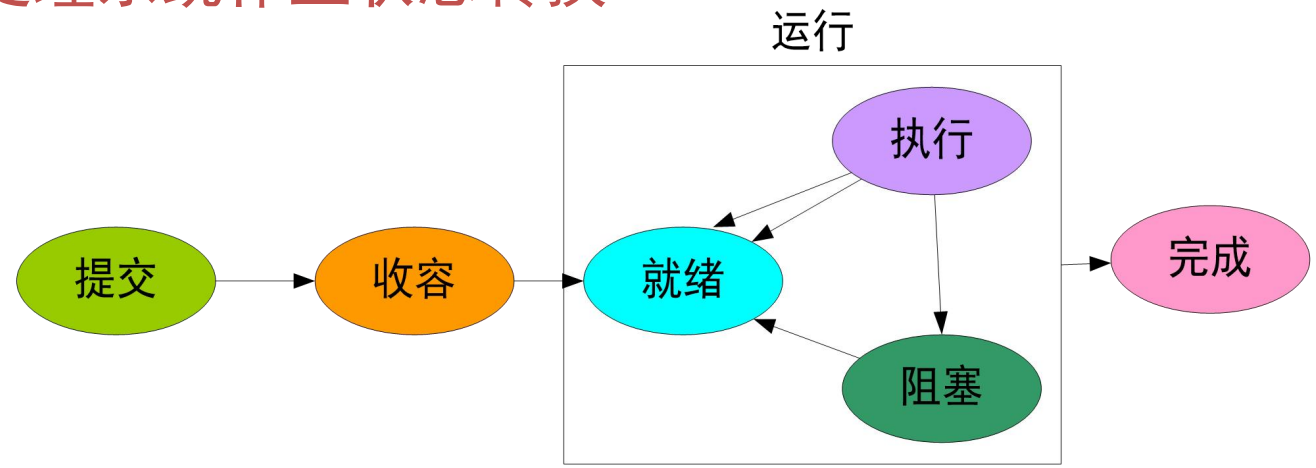
- 作业概念更多地用在批处理操作系统，而进程则可以用在各种多道程序设计系统。
- 作业是任务实体，进程是完成任务的执行实体;没有作业任务，进程无事可干，没有进程，作业任务没法完成。

3.4.1 作业分类及状态转换

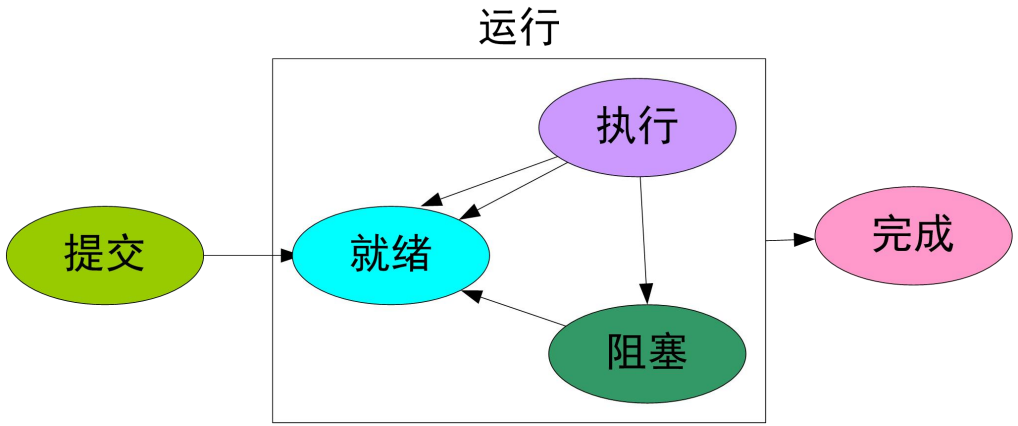


3.4.1 作业分类及状态转换

批处理系统作业状态转换



交互式作业状态转换



3.4.2 作业调度

❶ 作业调度是对批处理作业从收容到运行状态的转变

- 批处理作业需要作业调度
- 分时与实时作业是交互式的，直接进入内存，不需要作业调度

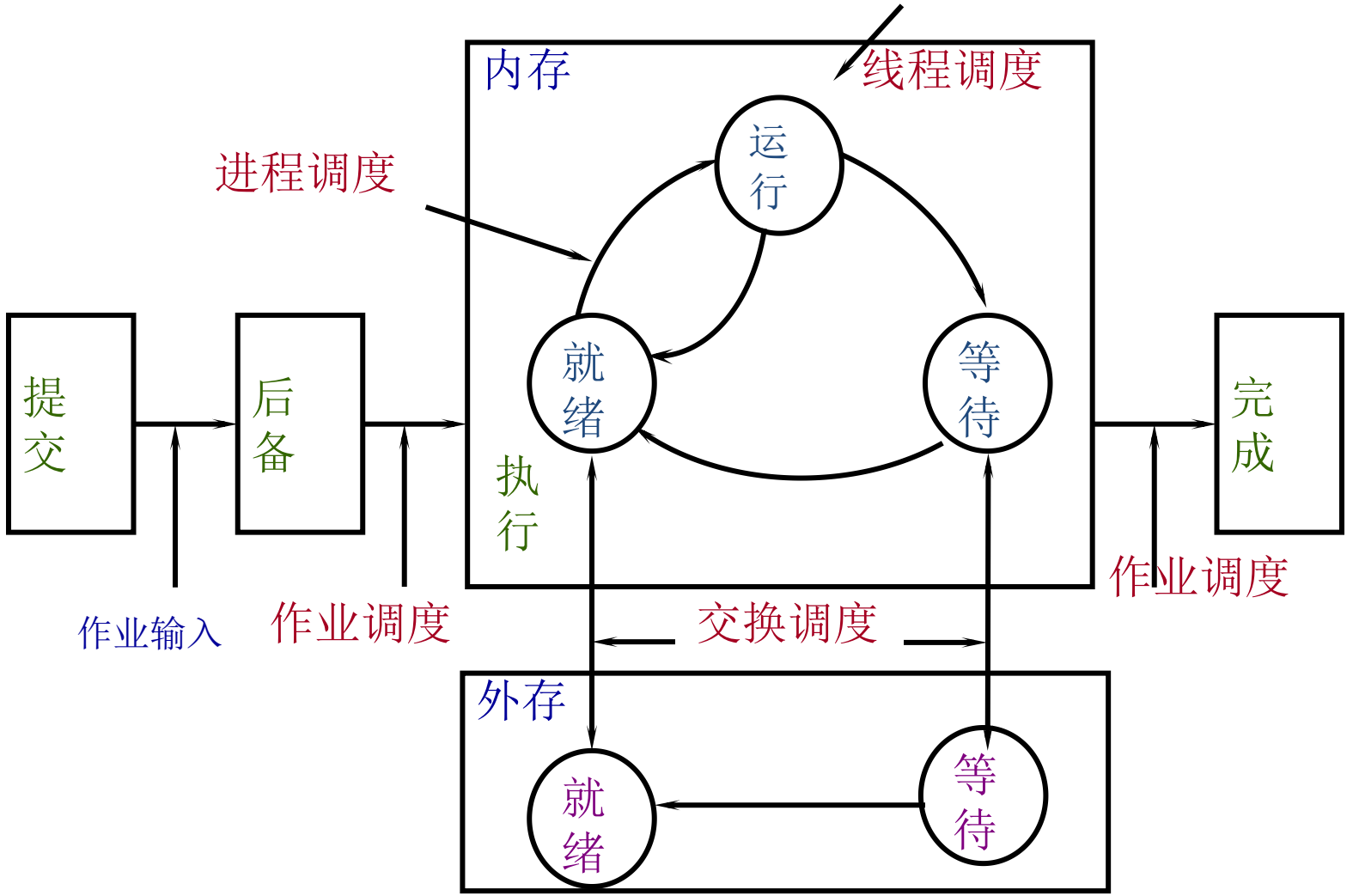
调度

决定

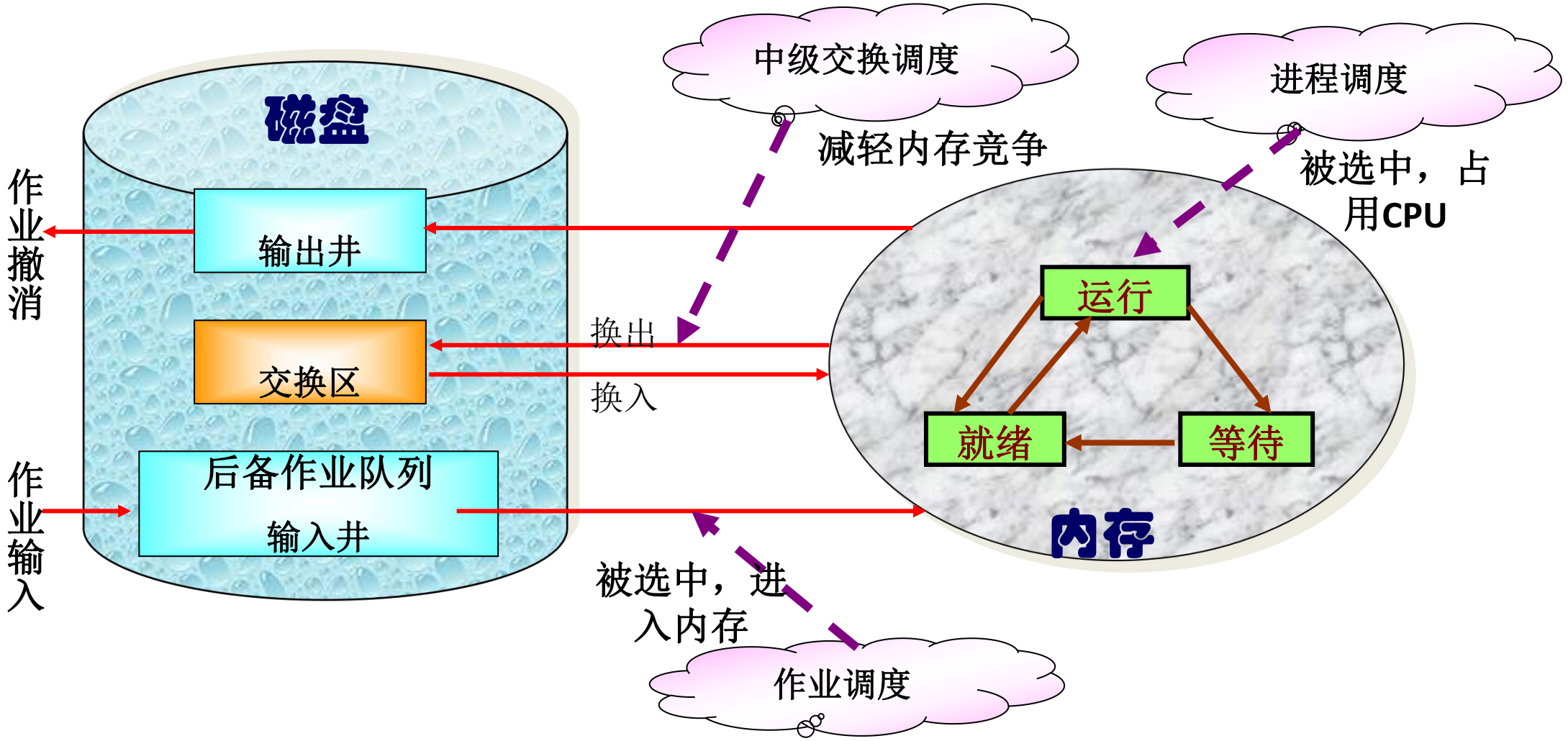
- *接纳多少作业*进入内存：取决于多道程序度
- *接纳哪些作业*：调度算法（如先来先服务短业优先等）

作业调度的主要任务是：根据JCB中的信息，检查系统中的资源能否满足作业对资源的需求，以及按照一定的调度算法，从外存的后备队列中选取某些作业调入内存，并为它们创建进程、分配必要的资源。然后再将新创建的进程排在就绪队列上等待调度。

3.3.2 作业调度和进程调度的关系



3.3.2 作业调度和进程调度的关系



3.5.1 进程调度的主要功能

进程调度 (低级调度或微观调度)

在进入内存并处于就绪队列的进程中决定那个进程真正获得CPU及其使用CPU的时间。

进程调度的任务主要有三：

(1) 保存处理机的现场信息。

(2) 按某种算法选取进程。

(3) 把处理器分配给进程。

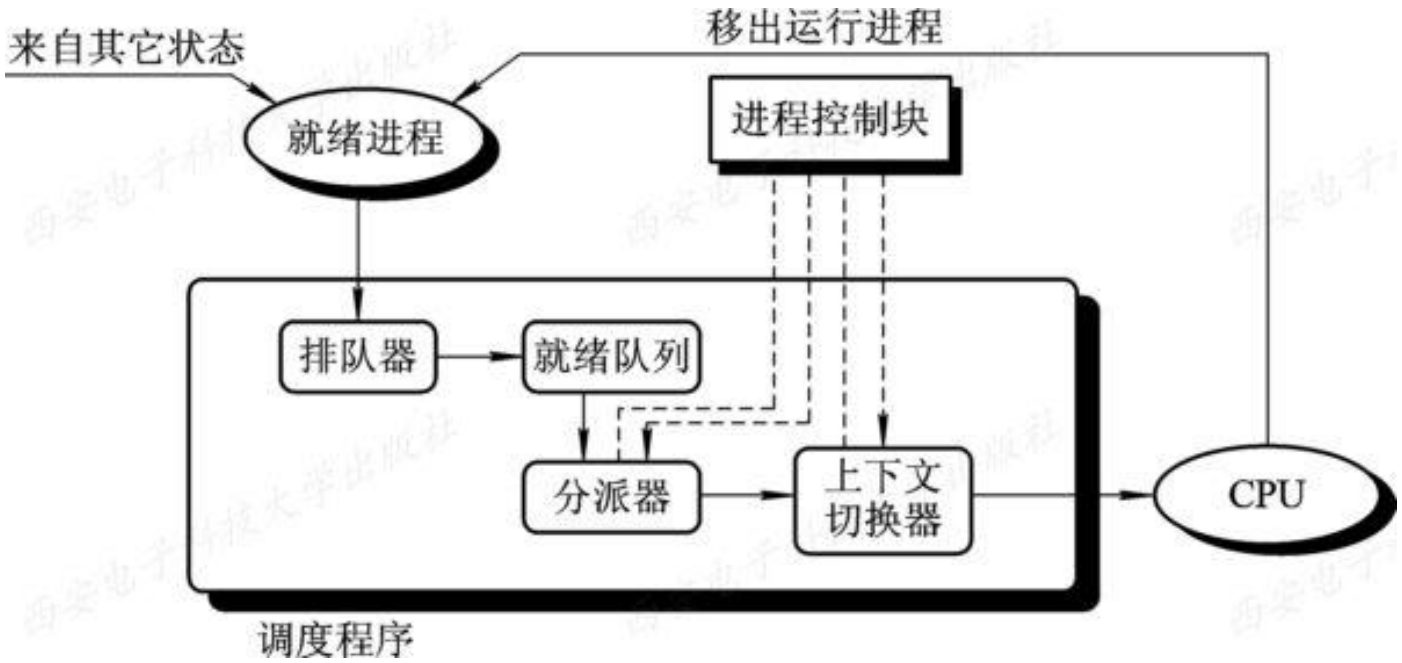
3.5.2 进程调度机制

为了实现进程调度，在进程调度机制中，应具有如下三个基本部分，如图所示。

(1) 排队器。

(2) 分派器。

(3) 上下文切换器。



3.5.3 进程调度方式

1) 非抢占方式（就绪进程不可从运行进程手中抢占CPU）

一旦把处理机分配给某进程后，就一直让它运行下去，决不会中断，直至该进程完成，或发生某事件而被阻塞时，才把处理机分配给其它进程。

2) 抢占方式（就绪进程可以从运行进程手中抢占CPU）

允许调度程序暂停正在执行的进程，将处理机分配给另一进程。

- *批处理机系统*，可以防止一个长进程长时间地占用处理机，以确保处理机能为所有进程提供更为公平的服务。
- *分时系统中*，只有采用抢占方式才有可能实现人—机交互。
- *实时系统中*，抢占方式能满足实时任务的需求。但抢占方式比较复杂，所需付出的系统开销也较大。

很多操作系统使用两种策略的组合，内核关键程序是非剥夺式的，用户进程是剥夺式的。

调度算法

调度的实质是一种资源分配，因而调度算法是指：根据系统的资源分配策略所规定的资源分配算法。

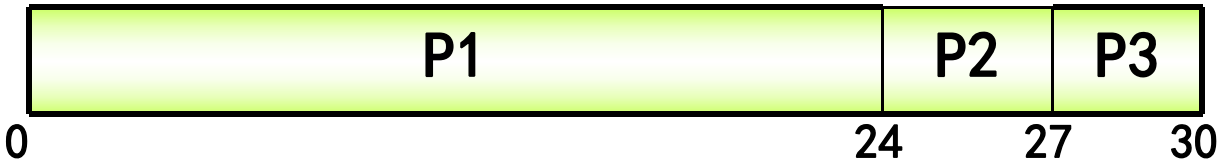
对于不同的系统和系统目标，通常采用不同的调度算法，例如：在批处理系统中为照顾为数众多的短作业，应采用短作业优先的调度算法；又如在分时系统中，为了保证系统具有合理的响应时间，应采用轮转法进行调度。目前存在的多种调度算法中，有的算法适用于作业调度，有的算法适用于进程调度；但有些算法作业调度和进程调度都可以采用。

3.6.1 先到先服务调度（first-come,first-served,FCFS）

先请求CPU的进程被首先分配到CPU，可用FIFO队列来实现

- 平均周转时间通常相当长，与作业的提交和调度顺序有关
- 非抢占调度
- 例

进程	区间时间
P1	24
P2	3
P3	3



平均周转时间(24+27+30)/3=27

平均周转时间(3+6+30)/3=13

FCFS算法比较有利于长作业（进程）即CPU繁忙型，而不利于短作业（进程）即I/O繁忙型。

3.6.1 先到先服务调度（first-come,first-served,FCFS）

例1：先来先服务调度算法（单位：小时，并以十进制计）

作业（进程）名	提交时间	运行时间	开始时间	完成时间	周转时间	带权周转时间
1	8.00	2.00	8.00	10.00	2.00	1
2	8.50	0.50	10.00	10.50	2.00	4
3	9.00	0.10	10.50	10.60	1.60	16
4	9.50	0.20	10.60	10.80	1.30	6.5

平均周转时间T＝1.725

平均带权周转时间W＝6.875

3.6.1 先到先服务调度（first-come,first-served,FCFS）

练习1：1→2→3→4

作业（进程）名	到达时间	服务时间	开始执行时间	完成时间	周转时间	带权周转时间
1	10.00	2	10	12		
2	10.10	1	12	13		
3	10.25	0.25	13	13.25		
4	11.90	0.1	13.25	13.35		

平均周转时间T=2.34时间单位，

平均带权周转时间W＝7.6

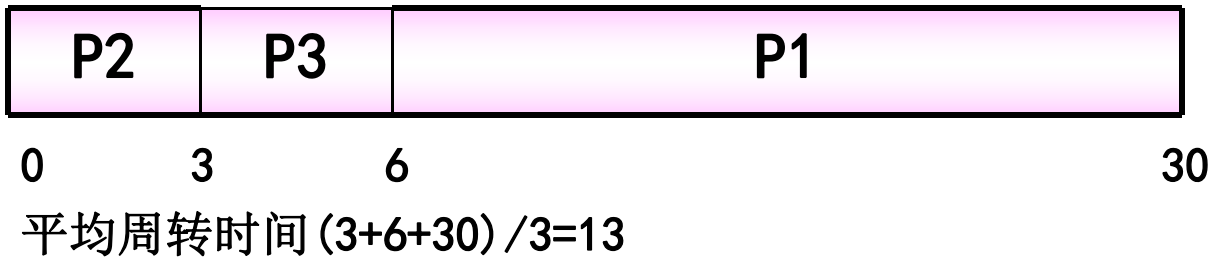
3.6.2 短作业（进程）优先调度算法 (SJF, Shortest Job First)

短作业优先（SJF）的调度算法，是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。

而短进程优先（SPF）调度算法则是从就绪队列中选出一估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时，再重新调度。

- 例

进程	区间时间
P1	24
P2	3
P3	3



平均周转时间(3+6+30)/3=13

SJF调度算法能有效的降低作业的平均等待时间，提高系统吞吐量。
缺点：对长作业不利；不能保证紧迫性作业（进程）的及时处理。

3.6.2 短作业（进程）优先调度算法 (SJF, Shortest Job First)

例2：短作业优先调度算法（单位：小时，并以十进制计）

作业	提交时间	运行时间	开始时间	完成时间	周转时间	带权周转时间
1	8.00	2.00	8.00	10.00	2.00	1
2	8.50	0.50	10.30	10.80	2.30	4.6
3	9.00	0.10	10.00	10.10	1.10	11
4	9.50	0.20	10.10	10.30	0.80	4

平均周转时间T＝1.55

平均带权周转时间W＝5.15

3.6.2 短作业（进程）优先调度算法 (SJF, Shortest Job First)

练习2：1→4→2→3

进程名	到达时间	服务时间	开始执行时间	完成时间	周转时间	带权周转时间
1	10.00	2	10.00	12.00	2.00	1
2	10.10	1	12.35	13.35	3.25	3.25
3	10.25	0.25	12.10	12.35	2.10	8.4
4	11.90	0.1	12.00	12.10	12.10	2

平均周转时间T=1.89时间单位，平均带权周转时间W＝3.66

3.6.3 响应比高者优先调度算法

响应比是指作业的响应时间与作业估计运行时间的比值。

$$\text{响应比} = \frac{\text{响应时间}}{\text{执行时间}}$$

响应比＝1＋作业等待时间/执行时间

选择原则是优先选取响应比值最大的作业。即兼顾等待时间长和运行时间短的作业，它是FCFS和SJF算法的结合。

3.6.3 响应比高者优先调度算法

例3：相应比高者优先调度算法（单位：小时，并以十进制计）

作业	提交时间	运行时间	开始时间	完成时间	周转时间	带权周转时间
1	8.00	2.00	8.00	10.00	2.00	1
2	8.50	0.50	10.10	10.60	2.10	4.2
3	9.00	0.10	10.00	10.10	1.10	11
4	9.50	0.20	10.60	10.80	1.30	6.5

响应比＝1＋作业等待时间/执行时间

例如：当作业3结束时，

Rp2= 1＋作业等待时间/可执行时间=1+(10.10-8.50)/0.5=1+3.2

Rp4= 1＋作业等待时间/可执行时间=1+(10.10-9.50)/0.2=1+3

平均周转时间T＝1.625，　平均带权周转时间W＝5.675

3.6.4 最高优先权优先调度算法（FPF）——作业调度

这种算法是根据确定的优先数来选取作业，每次总是选择优先级最高的作业。

规定用户作业优先数的方法：

1）由用户自己提出作业的优先数。有的用户为了自己的作业尽快被系统选中就设法提高自己作业的优先数，这时系统可以规定优先数越高则需付出的计算机使用费就越多，以作限制。

2）由系统综合有关因素来确定用户作业的优先数。

例如，根据作业的缓急程度、作业计算时间的长短、等待时间的多少、资源申请情况等来确定优先数。确定优先数时，各因素的比例应根据系统设计目标来分析这些因素在系统中的地位而决定。

3.6.4 最高优先权优先调度算法（FPF）——进程调度

采用最高优先级优先调度算法时，系统对每个进程确定一个优先数，进程的优先数用于表示进程的重要性及运行的优先性。调度时，系统把处理机分配给优先级最高的就绪进程。如果就绪进程具有相同的优先数，则再按先来先服务的次序分配处理机。

先来先服务调度算法是按照进程进入就绪队列的先后次序来选择进程分配处理机。

3.6.4 最高优先权优先调度算法（FPF）——进程调度

系统确定优先数的方法：

1. 静态优先数法：静态优先数是在创建进程时系统为其确定的，并且在进程的生命期内不再改变。

确定静态优先数的原则：

1）按进程类型。系统进程的优先级高于用户进程的优先级。

2）按进程使用的资源。进程所使用的资源越多，进程的优先级越低；反之，则进程的优先级越高。

3）按进程的估计运行时间。进程的估计运行时间越长，进程的优先级越低；反之，则进程的优先级越高。

4）由用户指定。有些系统可以按收费标准不同，设置不同的优先级别，可以由用户指定。

静态优先级法实现起来比较简单，但不能反映系统以及进程在运行过程中的动态变化情况，系统管理效果显然不佳。

3.6.4 最高优先权优先调度算法（FPF）——进程调度

2. 动态优先数法。动态优先数是指在系统创建进程时，根据系统资源的使用情况和进程的当前特点确定一个优先数，然后，在进程运行过程中再根据情况的变化动态调整进程的优先数。

调整进程优先数的原则：

1）进程占有处理机的时间越长，则进程的优先级越低；进程的等待时间越长，则进程的优先级越高。

2）根据进程所执行的程序的轻重缓急程度，调整进程的优先数。

3.6.4 最高优先权优先调度算法（FPF）——进程调度

例4：有一个具有两道作业的批处理系统，作业调度采用短作业优先的调度算法，进程调度采用以优先数为基础的抢占式调度算法，假设优先数小的优先级别高。如下表所示的作业序列。

（1）列出所有作业进入内存的时间及作业结束时间。

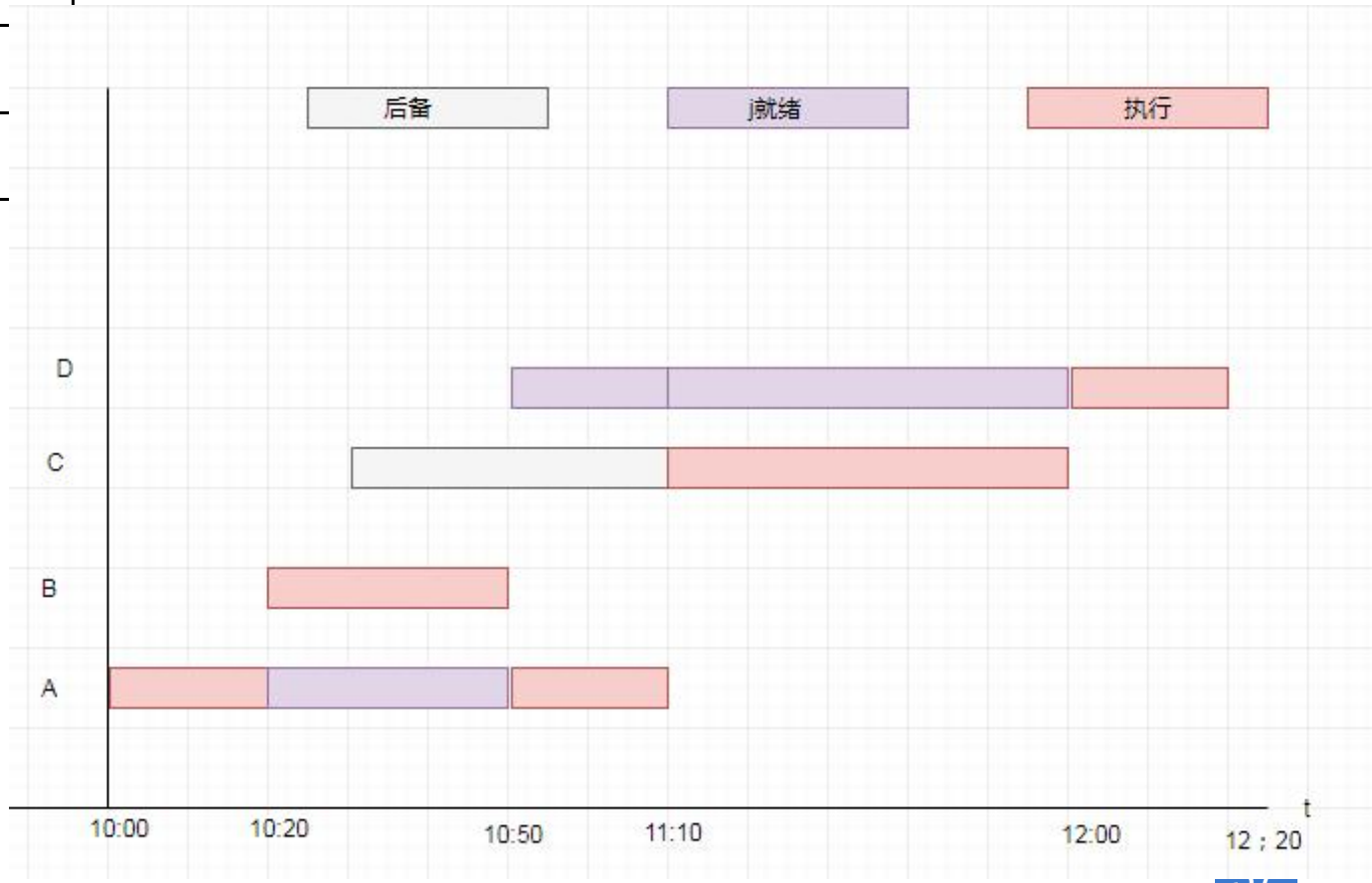
（2）计算平均周转时间。

作业名	到达时间	估计运行时间	优先数
A	10：00	40分钟	5
B	10：20	30分钟	3
C	10：30	50分钟	4
D	10：50	20分钟	6

3.6.4 最高优先权优先调度算法（FPF）——进程调度

作业名	到达时间	估计运行时间	优先数
A	10：00	40分钟	5
B	10：20	30分钟	3
C	10：30	50分钟	4
D	10：50	20分钟	6

分析：具有两道作业的批处理系统，意味着，在内存中同时最多只能有两道作业存在，如果有作业要进入内存，只能在后备队列中等待。



3.6.4 最高优先权优先调度算法（FPF）——进程调度

解：（1）各作业进入时间和结束时间如下：

作业名	进入时间	结束时间
A	10：00	11：10
B	10：20	10：50
C	11：10	12：00
D	10：50	12：20

（2）根据周转时间=完成时间—提交时间，可得各作业的周转时间为：

TA=11：10—10：00=70（分钟）

TB=10：50—10：20=30（分钟）

TC=12：00—10：30=90（分钟）

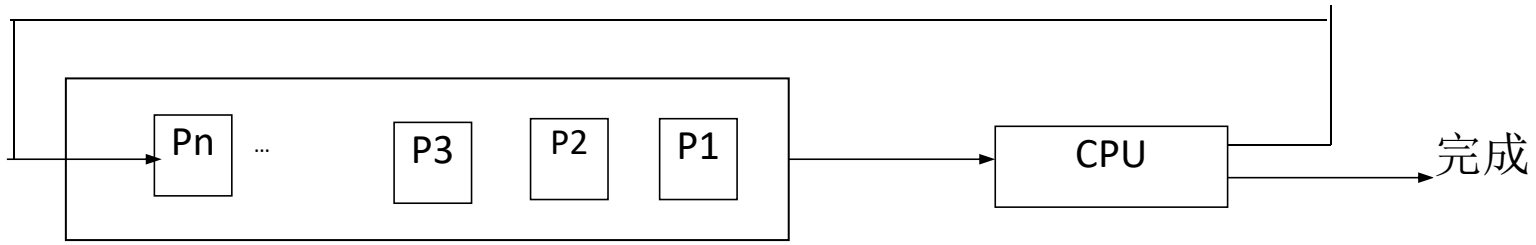
TD=12：20—10：50=90（分钟）

平均周转时间=（TA + TB + TC + TD ）/4=70（分钟）

3.6.5 基于时间片的轮转调度算法 (Round Robin)

1．时间片轮转法

时间片轮转法（RR算法）主要用于分时系统中的进程调度。系统把所有就绪进程按先入先出的原则排成一个队列。新来的进程加到就绪队列末尾。每当执行进程调度时，就绪队列的队首进程总是先被调度程序选中，让它在CPU上运行一个时间片的时间。



时间片轮转法原理图

3.6.5 基于时间片的轮转调度算法 (Round Robin)

例5：考虑下述四个进程A、B、C和D的执行情况。设它们依次进入就绪队列，但前后时间忽略不计。四个进程分别需要运行12，5，3和6个时间单位。如下表所示。计算当时间片分别为q=1、q=4时各进程的开始运行时间、完成时间、周转时间及带权周转时间。

进程名	到达时间	运行时间
A	0	12
B	0	5
C	0	3
D	0	6

3.6.5 基于时间片的轮转调度算法 (Round Robin)

解：通过分析及计算，我们可以得到下表：

进程名 到达时间		到达 时间	运行 时间	开始 时间	完成 时间	周转 时间	带权周 转时间
时间片 q=1	A	0	12	0	26	26	2.17
	B	0	5	1	17	17	3.4
	C	0	3	2	11	11	3.67
	D	0	6	3	20	20	3.33
	平均周转时间T=18. 5　　平均带权周转时间=3. 14						
时间片 q=4	A	0	12	0	26	26	2.17
	B	0	5	4	20	20	4
	C	0	3	8	11	11	3.67
	D	0	6	11	22	22	3.67
	平均周转时间T=19.75　　平均带权周转时间W=3.38						

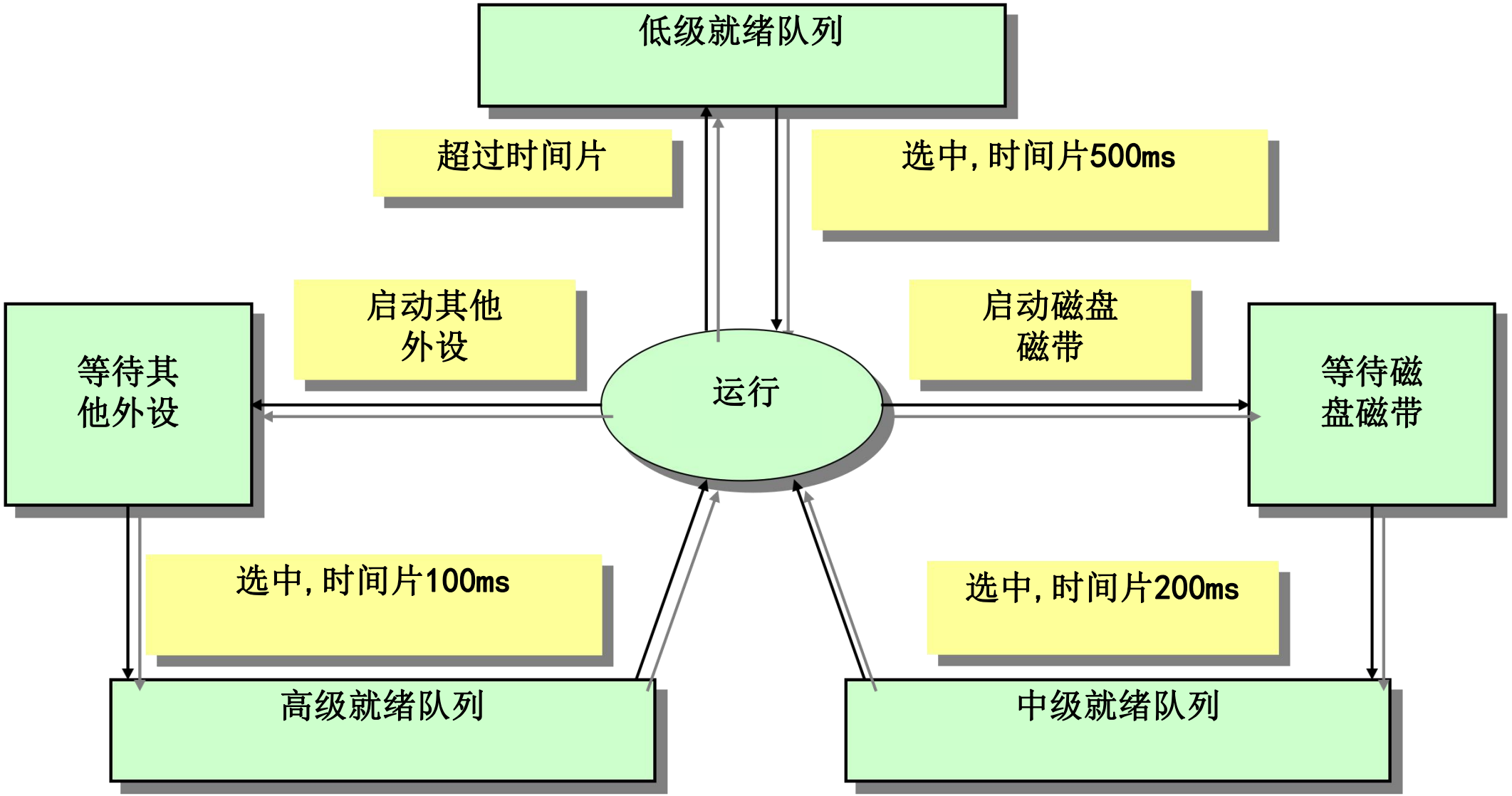
当q=1、q=4时，RR调度算法的比较

7. 多级反馈队列调度

- 又称反馈循环队列或多队列策略。主要思想是将就绪进程分为两级或多级，系统相应建立两个或多个就绪进程队列，较高优先级的队列一般分配给较短的时间片。
- 处理器调度先从高级就绪进程队列中选取可占有处理器的进程，只有在选不到时，才从较低级的就绪进程队列中选取。

42

一个三级反馈队列调度策略



总结：

处理机调度可以分为4级：

1）作业调度（高级调度）

2）交换调度（中级调度）

3）进程调度（低级调度）

4）线程调度

作业的4个状态：提交、后备（收容）、执行、完成。

作业控制块——JCB

总结：

作业调度功能：

1）记录各作业的状况。

2）按调度算法，从后备作业中挑选作业进入主存运行。

3）为被选中的作业做好执行前的准备工作。

4）在作业执行结束时做善后处理工作。

周转时间Ti = 完成时刻T_f— 提交时刻T_t

= 等待时间Tw + 运行时间Ts

带权周转时间Wi = Ti / Ts

总结：

作业调度算法：

1. 先来先服务调度算法（FCFS）

2. 短作业优先调度算法（SJF）

3. 响应比高者优先调度算法(HRN)

4. 最高优先级优先调度算法

调度程序：将进程插入到就绪队列，按一定原则保持队列结构；

分派程序：将进程从就绪队列中移出，建立它执行的机器状态。

总结：

进程调度功能：

1. 记录系统中所有进程的执行情况。
2. 按照一定调度策略选择一个占有处理机的就绪进程。
3. 实施处理机的分配和回收。

进程调度方式：

1. 非抢先调度方式
2. 可抢先调度方式

进程调度时机：

1. 进程运行结束；
2. 执行中的进程发生某个等待事件；
3. 分时系统时间片到；
4. 在采用可抢占调度方式的系统中，当具有更高优先级的进程要求使用处理机。

总结：

进程调度算法：

1. 最高优先级优先调度算法；
2. 时间片轮转调度算法。

系统确定进程优先数的方法：

1. 静态优先数法；
2. 动态优先数法。

时间片轮转调度算法：

1. 简单轮转法(固定时间片轮转法）；
2. 可变时间片轮转法；
3. 多级反馈轮转法。

小结

FCFS

算法思想	主要从“公平”的角度考虑（类似于我们生活中排队买东西的例子）
算法规则	按照作业/进程到达的先后顺序进行服务
用于作业/进程调度	用于作业调度时，考虑的是哪个作业先到达后备队列；用于进程调度时，考虑的是哪个进程先到达就绪队列
是否可抢占？	非抢占式的算法
优缺点	优点：公平、算法实现简单 缺点：排在长作业（进程）后面的短作业需要等待很长时间，带权周转时间很大，对短作业来说用户体验不好。即，FCFS算法对长作业有利，对短作业不利（Eg：排队买奶茶...）
是否会导致饥饿 （某进程/作业长期得不到服务）	不会

小结

SJF

- **算法思想**：追求最少的平均等待时间，最少的平均周转时间、最少的平均平均带权周转时间
- **算法规则**：最短的作业/进程优先得到服务（所谓“最短”，是指要求服务时间最短）
- **用于作业/进程调度**：即可用于作业调度，也可用于进程调度。用于进程调度时称为“短进程优先（SPF, Shortest Process First）算法”
- **是否可抢占？**：SJF和SPF是非抢占式的算法。但是也有抢占式的版本——最短剩余时间优先算法（SRTN, Shortest Remaining Time Next）
- **优缺点**：优点：“最短的”平均等待时间、平均周转时间
  缺点：不公平。对短作业有利，对长作业不利。可能产生饥饿现象。另外，作业/进程的运行时间是由用户提供的，并不一定真实，不一定能做到真正的短作业优先
- **是否会导致饥饿**：会。如果源源不断地有短作业/进程到来，可能使长作业/进程长时间得不到服务，产生“饥饿”现象。如果一直得不到服务，则称为“饿死”

小结

HRRN

- **算法思想**：要综合考虑作业/进程的等待时间和要求服务的时间
- **算法规则**：在每次调度时先计算各个作业/进程的响应比，选择响应比最高的作业/进程为其服务

$$\text{响应比} = \frac{\text{等待时间}+\text{要求服务时间}}{\text{要求服务时间}}$$

响应比 ≥ 1

- **用于作业/进程调度**：即可用于作业调度，也可用于进程调度
- **是否可抢占？**：非抢占式的算法。因此只有当前运行的作业/进程主动放弃处理机时，才需要调度，才需要计算响应比
- **优缺点**：综合考虑了等待时间和运行时间（要求服务时间）
  等待时间相同时，要求服务时间短的优先（SJF 的优点）
  要求服务时间相同时，等待时间长的优先（FCFS 的优点）
  对于长作业来说，随着等待时间越来越久，其响应比也会越来越大，从而避免了长作业饥饿的问题
- **是否会导致饥饿**：不会

小结

算法	思想&规则	可抢占？	优点	缺点	考虑到等待时间&运行时间？	会导致饥饿？
FCFS	自己回忆	非抢占式	公平；实现简单	对短作业不利	等待时间√ 运行时间×	不会
SJF/SPF	自己回忆	默认为非抢占式，也有SJF的抢占式版本最短剩余时间优先算法（SRTN）	“最短的”平均等待/周转时间；	对长作业不利，可能导致饥饿；难以做到真正的短作业优先	等待时间× 运行时间√	会
HRRN	自己回忆	非抢占式	上述两种算法的权衡折中，综合考虑的等待时间和运行时间		等待时间√ 运行时间√	不会

小结

优先级调度

算法思想	公平地、轮流地为各个进程服务，让每个进程在一定时间间隔内都可以得到响应
算法规则	按照各进程到达就绪队列的顺序，轮流让各个进程执行一个时间片（如 100ms）。若进程未在一个时间片内执行完，则剥夺处理机，将进程重新放到就绪队列队尾重新排队。
用于作业/进程调度	用于进程调度（只有作业放入内存建立了相应的进程后，才能被分配处理机时间片）
是否可抢占？	若进程未能在时间片内运行完，将被强行剥夺处理机使用权，因此时间片轮转调度算法属于抢占式的算法。由时钟装置发出时钟中断来通知CPU时间片已到
优缺点	优点：公平；响应快，适用于分时操作系统； 缺点：由于高频率的进程切换，因此有一定开销；不区分任务的紧急程度。
是否会导致饥饿	不会
补充	时间片太大或太小分别有什么影响？

复习题训练

5个进程P1、P2、P3、P4、P5几乎同时到达，预期运行时间分别为10、6、2、4、8个时间单位。各进程的优先级分别为3、5、2、1、4（数值越大，优先级越高）。请按下列调度算法计算任务的平均周转时间（进程切换开销可忽略不计）。

（1）先来先服务（按P1、P2、P3、P4、P5顺序）算法。

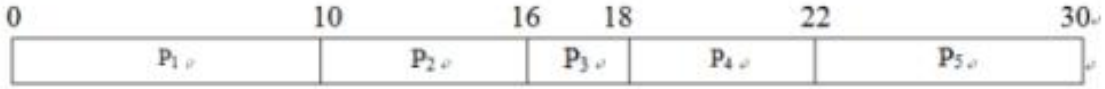
（2）时间片轮转算法，假定时间片大小为2个时间单位。

（3）优先权调度算法。

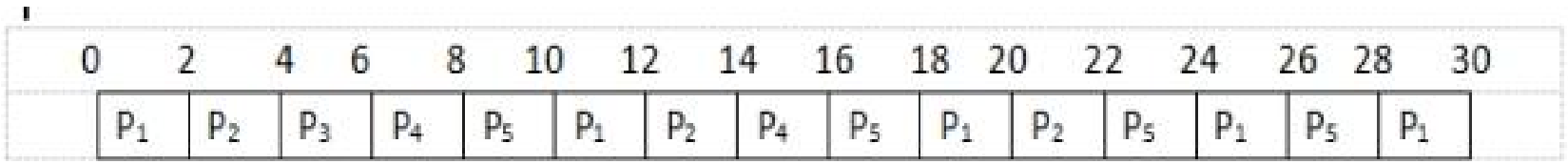
复习题训练

根据算法思想，确定调度先后顺序。

（1）FCFS调度顺序如图所示。



（2）时间片轮转调度顺序如图所示。



（3）优先权调度算法的调度顺序如图所示。

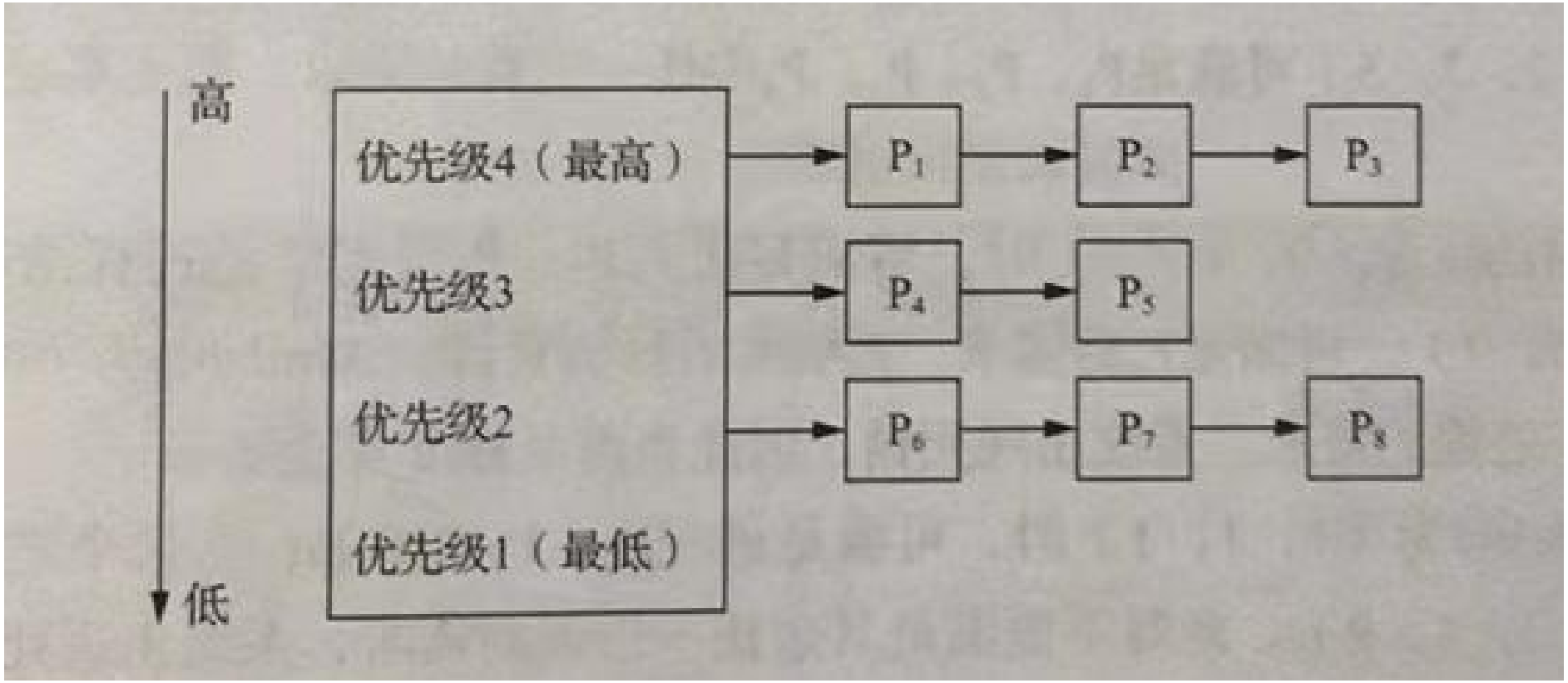
复习题训练

于是，可以得到如表所示的结果。

算法	时间类型	P_1	P_2	P_3	P_4	P_5	平均
	运行时间	10	6	2	4	8	
FCFS	周转时间	10	16	18	22	30	19.2
	带权周转时间	1	2.67	9	5.5	3.75	4.384
RR	周转时间	30	22	6	16	28	20.4
	带权周转时间	3	3.67	3	4	3.5	3.434
优先权	周转时间	24	6	26	30	14	20
	带权周转时间	2.4	1	13	7.5	1.75	5.13

复习题训练

（考研真题）将一组进程分为 4类，如图所示。各类进程之间采用优先级调度算法，而各类进程的内部采用 RR调度算法。请简述P1，P2，P3，P4，P5，P6，P7，P8进程的调度过程。



复习题训练

由于不同类进程间采用优先级调度算法，同类进程间采用RR调度算法，因此，系统首先对优先级为4的进程P1、P2、P3采用RR调度算法进行运行;当P1、P2、P3运行结束或阻塞时，再对优先级为3的进程P4、P5采用RR调度算法进行运行。在此期间，若P1、P2、P3队列中有转为就绪状态的进程，则优先级3队列的当前时间片用完后回到优先级4队列进行进程调度。类似地，当进程P1~P5运行结束或阻塞时，对优先级为2的进程P6、P7、P8采用RR调度算法进行运行，一旦进程P1~P5中有一个转为就绪状态，当前时间片用完后就立即回到相应的优先级队列进行RR调度。

第三章 处理机调度与死锁（二）

3.6 死锁的概述
3.7 预防死锁
3.8 避免死锁
3.9 死锁的检测与恢复

死锁概念

一个死锁的例子：如下图所示，系统中的两个进程都因等待对方的临界资源而不能继续运行。这种现象称为死锁。该例中的临界资源R1和R2可能是打印机、磁带机、缓冲区等。

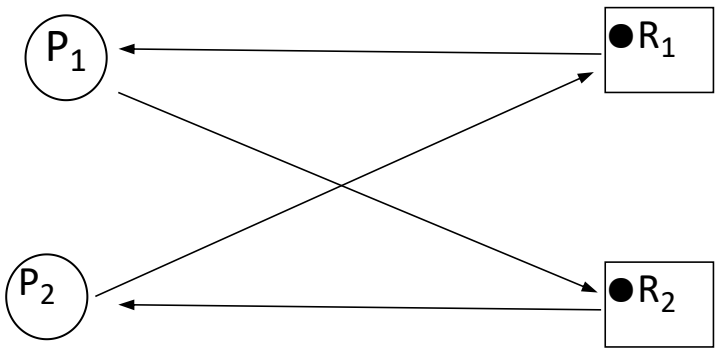


图3-6　进程P1、P2陷入死锁状态

资源问题

可重用性资源和可消耗性资源

- 可重用性资源：一次只能分配给一个进程，不允许多个进程共享，遵循：请求资源　使用资源　释放资源 （大部分资源）。
- 可消耗性资源：由进程动态创建和消耗 （进程间通信的消息）。

可抢占性和不可抢占性资源

- 可抢占性资源：某进程在获得这类资源后，该资源可以再被其他进程或系统抢占，**CPU（处理机）和主存区。**
- 不可抢占资源：当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，**打印机、磁带机。**

3.6.1 死锁的产生

例 1： 进程推进顺序不当产生死锁

设系统有打印机、读卡机各一台，被进程 P 和Q共享。两个进程并发执行，按下列次序请求和释放资源：

进程 P	进程Q
请求读卡机	请求打印机
请求打印机	请求读卡机
释放读卡机	释放读卡机
释放打印机	释放打印机

3.6.1 死锁的产生

例2：PV操作使用不当产生死锁

进程Q1	进程Q2
……	……
P(S1);	P(s2);
P(s2);	P(s1);
使用r1和r2;	使用r1和r2
V(S1);	V(s2);
V(S2);	V(S1);

3.6.1 死锁的产生

例 3： 资源分配不当引起死锁

若系统中有m个资源被n个进程共享，每个进程都要求K个资源，而m < n·K时，即资源数小于进程所要求的总数时，如果分配不得当就可能引起死锁。

3.6.1 死锁的产生

例 4：对临时性资源使用不加限制引起死锁

进程通信使用的信件是一种临时性资源，如果对信件的发送和接收不加限制，可能引起死锁。

进程P1等待进程P3的信件S3来到后再向进程P2发送信件S1；P2又要等待P1的信件S1来到后再向P3发送信件S2；而P3也要等待P2的信件S2来到后才能发出信件S3。这种情况下形成了循环等待，产生死锁。

死锁定义

死锁：一组等待的进程，其中每一个进程都持有资源，并且等待着由这个组中其他进程所持有的资源。

产生死锁的必要条件

互斥

01

- 一段时间内某资源只能被一个进程占用。

请求和保持

02

- 一个至少持有一个资源的进程等待获得额外的由其他进程所持有的资源。

不可抢占

03

- 一个资源只有当持有它的进程完成任务后，自由的释放。

循环等待

04

- 等待资源的进程之间存在环 {P0, P1, ..., Pn} 。
- P0 等待P1占有的资源, P1等待P2占有的资源, ..., Pn–1等待Pn占有的资源, P0等待Pn占有的资源

处理死锁的方法

- 确保系统永远不会进入死锁状态
  - 死锁预防
  - 死锁避免
- 允许系统进入死锁状态，然后恢复系统
  - 死锁检测
  - 死锁恢复
- 忽略这个问题，假装系统中从未出现过死锁。这个方法被大部分的操作系统采用，包括UNIX

处理死锁的方法

预防死锁

01

- 破坏死锁的四个必要条件中一个或几个。

避免死锁

02

- 在资源动态分配时，防止系统进入不安全状态。

检测死锁

03

- 事先不采取任何措施，允许死锁发生，但及时检测出死锁的发生。

解除死锁

04

- 检测到死锁发生时，采取相应措施，将进程从死锁状态中解脱出来。

3.7 预防死锁

破坏死锁的四个必要条件中的一个或几个

- **互斥**：互斥条件是共享资源必须的，不仅不能改变，还应加以保证
- **请求和保持**：必须保证进程申请资源的时候没有占有其他资源
  - 要求进程在执行前一次性申请全部的资源，只有没有占有资源时才可以分配资源
  - 资源利用率低，可能出现饥饿
  - 改进：进程只获得运行初期所需的资源后，便开始运行；其后在运行过程中逐步释放已分配的且用毕的全部资源，然后再请求新资源

3.7 预防死锁

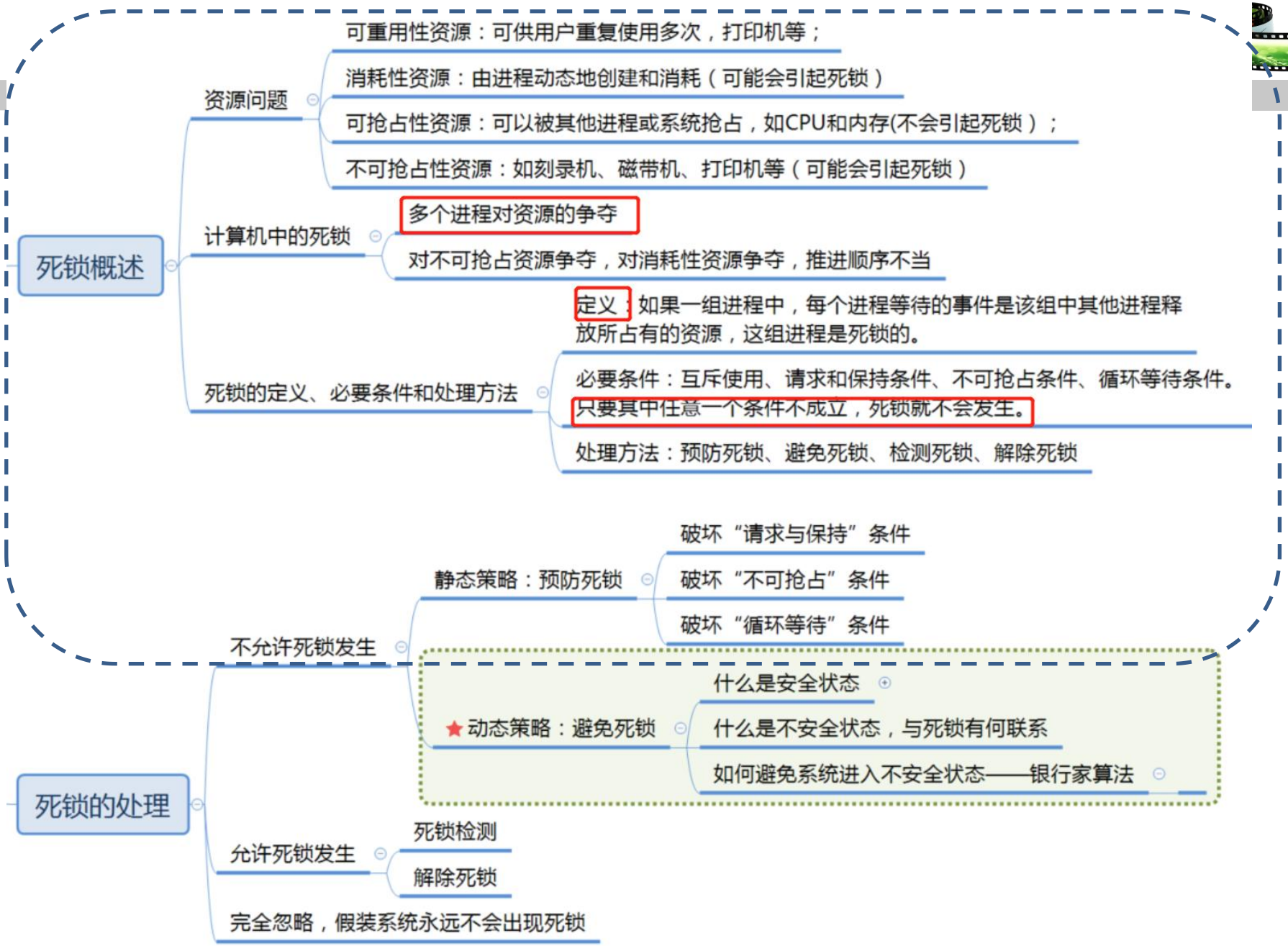
非抢占：

- 如果一个进程的申请没有实现，它要释放所有占有的资源；
- 先占的资源放入进程等待资源列表中；
- 进程在重新得到旧的资源的时候可以重新开始。

循环等待：对所有的资源类型排序进行线性排序，并赋予不同的序号，要求进程按照递增顺序申请资源。

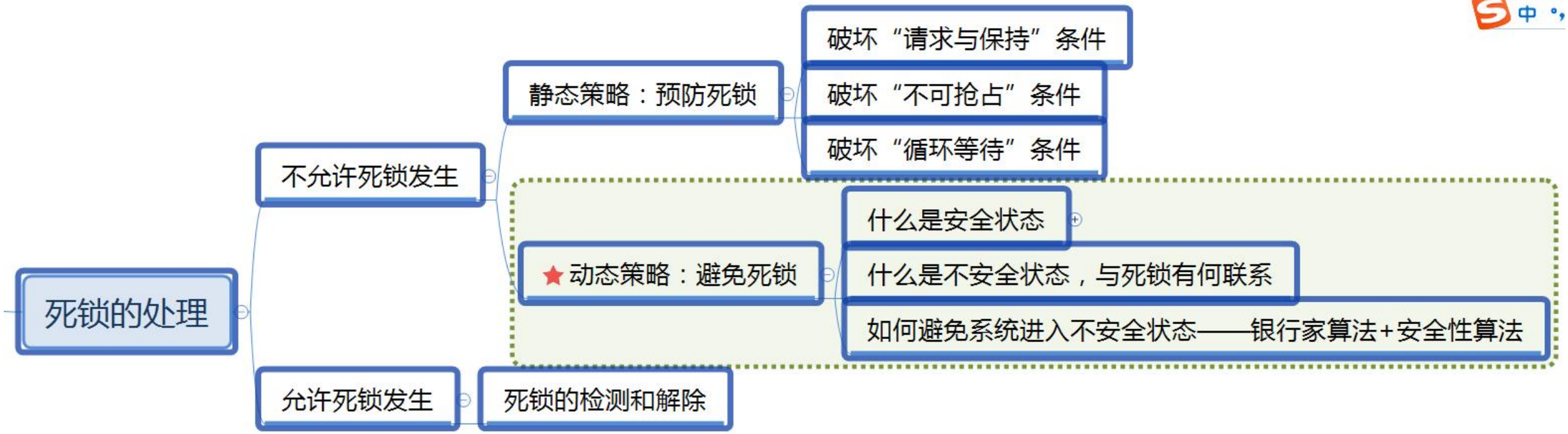
- 如何规定每种资源的序号是十分重要的；
- 限制新类型设备的增加；
- 作业使用资源的顺序与系统规定的顺序不同；
- 限制用户简单、自主的编程。

知识回顾


死锁概述
资源问题
可重用性资源：可供用户重复使用多次，打印机等；
消耗性资源：由进程动态地创建和消耗（可能会引起死锁）
可抢占性资源：可以被其他进程或系统抢占，如CPU和内存(不会引起死锁）；
不可抢占性资源：如刻录机、磁带机、打印机等（可能会引起死锁）
计算机中的死锁
多个进程对资源的争夺
对不可抢占资源争夺，对消耗性资源争夺，推进顺序不当
死锁的定义、必要条件和处理方法
定义：如果一组进程中，每个进程等待的事件是该组中其他进程释放所占有的资源，这组进程是死锁的。
必要条件：互斥使用、请求和保持条件、不可抢占条件、循环等待条件。
只要其中任意一个条件不成立，死锁就不会发生。
处理方法：预防死锁、避免死锁、检测死锁、解除死锁
死锁的处理
不允许死锁发生
静态策略：预防死锁
破坏“请求与保持”条件
破坏“不可抢占”条件
破坏“循环等待”条件
★动态策略：避免死锁
什么是安全状态
什么是不安全状态，与死锁有何联系
如何避免系统进入不安全状态——银行家算法
允许死锁发生
死锁检测
解除死锁
完全忽略，假装系统永远不会出现死锁

3.8 死锁的处理策略——避免死锁

本节课内容



3.8.1 什么是安全序列

你是一位成功的银行家，手里掌握着10个亿的资金...

有四个企业想找你贷款，分别是企业A、B、C、D，简称ABCD。

每个企业家提出的最大贷款数量分别为**6、5、4、7个亿**，并不是所有企业家都马上需要其全部贷款（6+5+4+7=22）。

然而...江湖中有个不成文的规矩：如果你借给企业的钱总数达不到企业提出的最大要求，那么不管你之前给企业借了多少钱，那些钱都拿不回来了...

- 客户要求分期贷款，一旦得到每期贷款，就能够归还贷款
- 银行家应谨慎的贷款，防止出现坏帐

刚开始，ABCD四个企业分别从你这儿借了 **1、1、2、4** 个亿...

3.8.1 什么是安全序列

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1	4
C	4	2	2
D	7	4	3

银行家当前剩余量: 2

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1+1=2	4-1=3
C	4	2	2
D	7	4	3

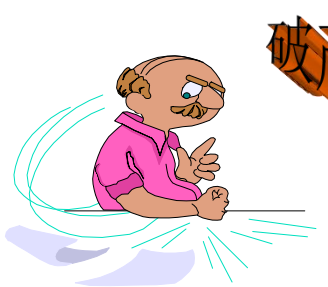
银行家当前剩余量:1

手里还有：2亿

此时，B 还想借1 亿，你敢借吗？假如答应了B的请求……

手里还有：1亿

只剩下1亿，如果ABCD都提出再借2亿的需求，都得不到满足…



现在银行家要破产了，剩余的资金贷给谁都不够，因此项目不能完成，银行家不能收回贷款。

错误发生在最后贷款给B的1个亿上

3.8.1 什么是安全序列

不敢吱声

所以：经过三百六十度无死角检查，
借给B的1亿是不安全的…

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1	4
C	4	2	2
D	7	4	3

银行家当前剩余量: 2

B请求1亿：不贷款

手里还有：2亿

此时，C还想借2 亿，你敢借吗？
假如答应了C的请求……

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1	4
C	4	2+2	2-2=0
D	7	4	3

银行家当前剩余量:0

C完成项目后，还回来4亿，银行家下次可贷给B，也可贷给D其中的3，不管如何处理，B或D都能完成归还5或归还7；最后贷给A所需资金5

3.8.1 什么是安全序列

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1	4
C	4	2+2	2-2=0
D	7	4	3

因此：经过三百六十度无死角检查，
借给C的2亿是安全的…
因为按照C->B->D->A或C->D->B->A
或C.....的顺序是可行的

最终，A、B、C、D都完成了项目，银行家得到了贷款利润。

存在的安全序列是：C、B、D、A或C、D、B、A

3.8.2 安全序列、不安全状态、死锁的联系

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1+1=2	4-1=3
C	4	2	2
D	7	4	3

给B借1亿是不安全的，之后手里只剩下1亿，如果其他的企业都提出借款，那么任何一个企业的需求都得不到满足…

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1	4
C	4	2+2	2-2=0
D	7	4	3

给C借2亿是安全的，因为存在C->B->D->A这样的*安全序列*

所谓**安全序列**，就是指如果系统按照这种序列分配资源，则每个进程都能顺利完成。只要能找出一个安全序列，系统就是**安全状态**。当然，**安全序列可能有多个**。

3.8.2 安全序列、不安全状态、死锁的联系

- 如果分配了资源之后，系统中找不出任何一个安全序列，系统就进入了**不安全状态**。这就意味着之后**可能**所有进程都**无法**顺利的**执行**下去。当然，如果有进程*提前归还*了一些资源，那系统**也有可能重新回到安全状态**，不过我们在分配资源之前总是要考虑到最坏的情况。

比如D 先归还了1亿，那么就有安全序列B->C->D->A

- 如果系统处于**安全状态**，就**一定不会**发生**死锁**。如果系统进入**不安全状态**，就**可能**发生**死锁**（处于不安全状态未必就是发生了死锁，但发生死锁时一定是在不安全状态）
- 因此可以在**资源分配之前预先判断这次分配是否会导致系统进入不安全状态**，以此决定是否答应资源分配请求。这也是**“银行家算法”**的核心思想。

3.8.3 银行家算法

银行家算法是荷兰学者 Dijkstra 为银行系统设计的，以确保银行在发放现金贷款时，不会发生不能满足所有客户需要的情况。后来该算法被用在操作系统中，用于**避免死锁**。

核心思想：在进程提出资源申请时，先预判此次分配是否会导致系统进入不安全状态。如果会进入不安全状态，就暂时不答应这次请求，让该进程先阻塞等待。

- 操作系统（银行家）
- 操作系统管理的资源(周转资金)
- 进程（要求贷款的客户）

3.8.3 银行家算法

- 思考：银行贷款的例子中，只有一种类型的资源——钱，但是在计算机系统中会有多种多样的资源，应该怎么把算法拓展为多种资源的情况呢?
- 可以把单维的数字拓展为多维的向量。比如：系统中有5个进程 P0~P4，3种资源 R0~R2，初始数量为 (10, 5, 7)，则某一时刻的情况可表示如下：

企业	最大需求	已借走	最多还会借
A	6	1	5
B	5	1	4
C	4	2	2
D	7	4	3

进程	最大需求 (R0,R1,R2)		已分配 (R0,R1,R2)
P0	(7, 5, 3)		(0, 1, 0)
P1	(3, 2, 2)	减	(2, 0, 0)
P2	(9, 0 ,2)		(3, 0, 2)
P3	(2, 2, 2)		(2, 1, 1)
P4	(4, 3, 3)		(0, 0, 2)

此时总共已分配 (7, 2, 5)，还剩余 (10,5,7)- (7, 2, 5)=(3, 3, 2),可把最大需求、已分配的数据看作矩阵，两矩阵相减，就可算出各进程最多还需要多少资源了

3.8.3 银行家算法

判断系统此时是否处于安全状态？（即寻找系统中是否存在安全调度序列）

进程	最大需求 (R0,R1,R2)	已分配 (R0,R1,R2)	最多还需要 (R0,R1,R2)
P0	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)
P1			
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P3			
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

≥(3, 3, 2)

<(3, 3, 2)

>(5, 3, 2)

<(5, 3, 2)

说明如果优先把资源分配给P1，那P1一定是可以顺利执行结束的。等P1结束了就会归还资源

说明如果优先把资源分配给P3，那P3一定是可以顺利执行结束的。等P3结束了就会归还资源，于是，资源数就可以增加到：(2, 1, 1)+ (5, 3, 2)=(7, 4, 3)

思路：尝试***找出一个安全序列***...{...} {P1,P3,P0,P2,P4}

依次检查剩余可用资源 (3, 3, 2) 是否能满足各进程的需求

(1)可满足P1需求，将 P1 加入安全序列，并更新剩余可用资源值为 (5, 3, 2)

(2)依次检查剩余可用资源 (5, 3, 2) 是否能满足剩余进程（不包括已加入安全序列的进程）的需求

(3)可满足P3需求，将 P3 加入安全序列，并更新剩余可用资源值为 (7, 4, 3)

(4)依次检查剩余可用资源 (7, 4, 3) 是否能满足剩余进程（不包括已加入安全序列的进程）的需求.....

以此类推，共五次循环检查即可将5个进程都加入安全序列中，最终可得一个安全序列。该算法称为***安全性算法***。

3.8.3 银行家算法

进程	最大需求 (R0,R1,R2)	已分配 (R0,R1,R2)	最多还需要 (R0,R1,R2)
P0	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)
P1			
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P3			
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

{P1,P3}P0,P2,P4}

实际做题（手算）时可用更快速的方法***找到一个安全序列***:

经对比发现，（3, 3, 2）可满足 P1、P3，说明无论如何，这两个进程的资源需求一定是可以依次被满足的，因此P1、P3 一定可以顺利的执行完，并归还资源。 可把 P1、P3 先加入安全序列。

(2, 0, 0) + (2, 1, 1) + (3, 3, 2) = (7, 4, 3)

剩下的 P0、P2、P4 都可被满足。同理，这些进程都可以加入安全序列。

于是，5个进程全部加入安全序列，说明此时系统处于安全状态，暂不可能发生死锁。

3.8.3 银行家算法

进程	最大需求 (R0,R1,R2)	已分配 (R0,R1,R2)	最多还需要 (R0,R1,R2)
P0	(8, 5, 3)	(0, 1, 0)	(8, 4, 3)
P1			
P2	(9, 5 ,2)	(3, 0, 2)	(6, 5, 0)
P3			
P4	(4, 3, 6)	(0, 0, 2)	(4, 3, 4)

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

{P1,P3}

再看一个***找不到安全序列的例子***:

经对比发现，（3, 3, 2）可满足 P1、P3，说明无论如何，这两个进程的资源需求一定是可以依次被满足的，因此P1、P3 一定可以顺利的执行完，并归还资源。 可把 P1、P3 先加入安全序列。

(2, 0, 0) + (2, 1, 1) + (3, 3, 2) = (7, 4, 3)

剩下的 P0 需要 (8, 4, 3)，P2 需要 (6, 5, 0)，P4 需要 (4, 3, 4)。任何一个进程都不能被完全满足

于是，无法找到任何一个安全序列，说明此时系统处于**不安全状态，有可能发生死锁。**

可以很方便地用代码实现以上流程，每一轮检查都从编号较小的进程开始检查

3.8.3 银行家算法——数据结构

假设系统中有 n 个进程，m 种资源；

在系统中必须设置这样**四个数据结构**，分别用来描述系统中的量：

（1）**最大需求矩阵Max**：每个进程在运行前先声明对各种资源的最大需求数，可用一个n*m的矩阵，如果Max[i,j] = K，则表示进程i需要Rj类资源的最大数目为K。

（2）**分配矩阵 Allocation**: 这也是一个n*m的矩阵，定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation[i,j] = K,则表示进程Pi当前已分得Rj类资源的数目为K。

（3）**需求矩阵Need:** n*m的矩阵，表示每一个进程尚需的各类资源数。如果Need[i,j] = K，则表示进程Pi还需要Rj类资源K个,方能完成其任务。**Need[i, j]=Max[i, j]-Allocation[i, j]**

Max 矩阵　Allocation 矩阵　Need 矩阵

进程	最大需求 (R0,R1,R2)	已分配 (R0,R1,R2)	最多还需要 (R0,R1,R2)
P0	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)
P1	(3, 2, 2)	(2, 0, 0)	(1, 2, 2)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P3	(2, 2, 2)	(2, 1, 1)	(0, 1, 1)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

Available=(3,3,2)

（4）**可利用资源向量 Available**: 长度为m的一维数组，表示当前系统中还有多少可用资源。其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果Available[j] = K，则表示系统中现有Rj类资源K个

3.8.3 银行家算法——实现过程

某进程Pi向系统申请Rj类资源，可用 Request[i, j]请求向量表示，如果 Request[i, j] = K，表示进程Pi需要Rj类资源K个。当Pi发出资源请求后，可用银行家算法预判本次分配是否会导致系统进入不安全状态：

Max 矩阵　Allocation 矩阵　Need 矩阵

进程	最大需求	已分配	最多还需要
	(R0,R1,R2)	(R0,R1,R2)	(R0,R1,R2)
P0	(7, 5, 3)	(2, 2, 1)	(5, 3, 2)
P1	(3, 2, 2)	(2, 0, 0)	(1, 2, 2)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P3	(2, 2, 2)	(2, 1, 1)	(0, 1, 1)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

Available=(1,2,1)　Request0=(2,1,1)

（1）如果 Request[i, j] ≤Need[i, j] 便转向步骤(2);否则认为出错，因为它所需要的资源数已超过它所宣布(请求)的最大值。

（2）如果 Request[i, j] ≤ Available[j]，便转向步骤(3); 否则，表示尚无足够资源，Pi须等待。

（3）系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值(并非真分配，修改数值只是为了做预判)：

Available[j] = Available[j] - Request [i, j];

Allocation[i,j] = Allocation[i,j] + Request[i, j];

Need[i, j] = Need[i, j] - request[i, j];

3.8.3 银行家算法——实现过程

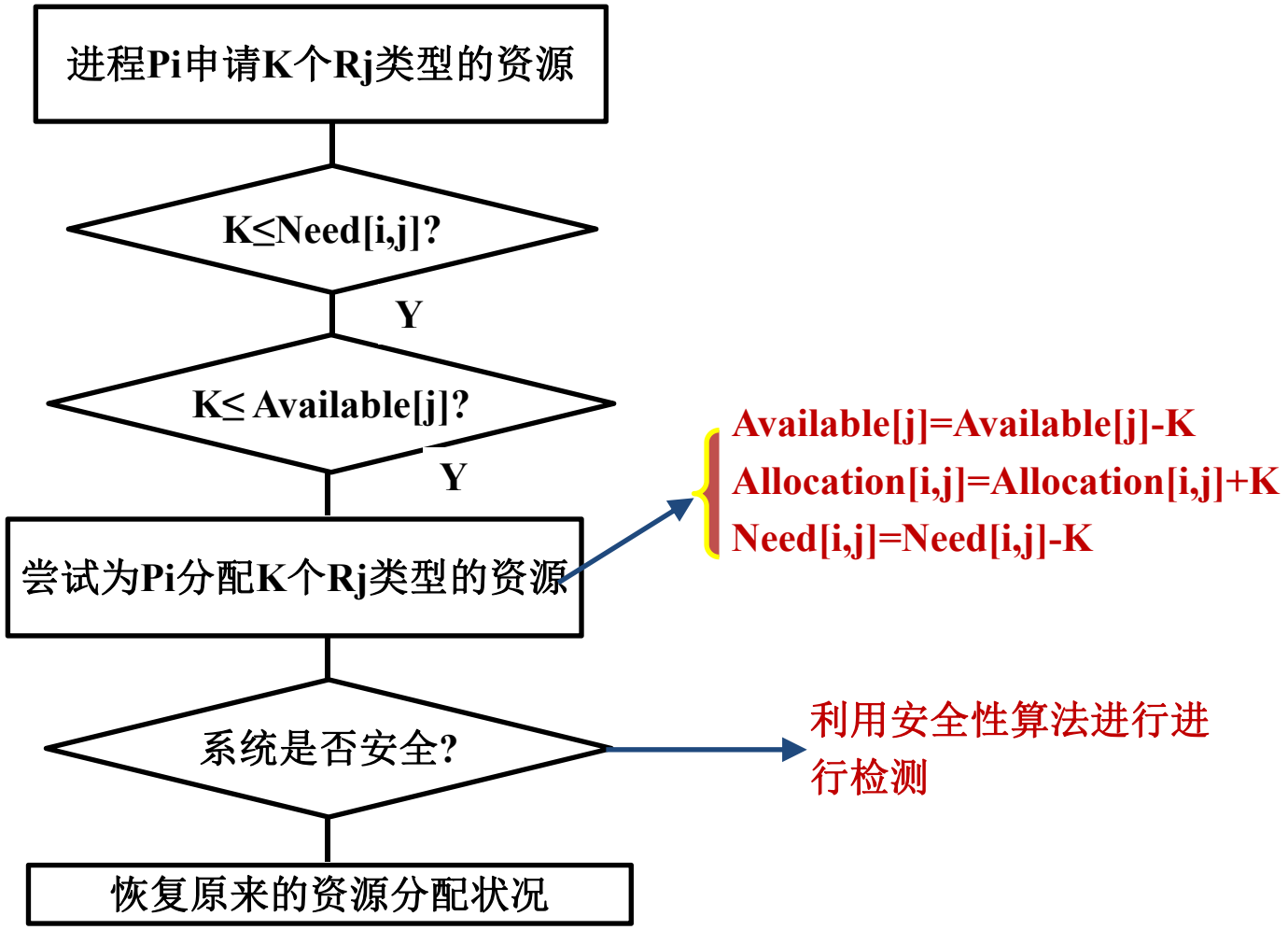
（4）操作系统执行**安全性算法**，检查此次资源分配后系统是否处于安全状态。若安全，正式将资源分配给进程Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来资源分配状态，让进程Pi阻塞等待。

Max 矩阵　Allocation 矩阵　Need 矩阵

进程	最大需求	已分配	最多还需要
	(R0,R1,R2)	(R0,R1,R2)	(R0,R1,R2)
P0	(7, 5, 3)	(2, 2, 1)	(5, 3, 2)
P1	(3, 2, 2)	(2, 0, 0)	(1, 2, 2)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P3	(2, 2, 2)	(2, 1, 1)	(0, 1, 1)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

Available=(1,2,1)　Request0=(2,1,1)

3.8.3 银行家算法——实现过程



3.8.4 安全性算法——实现过程

系统所执行的安全性算法可描述如下：

检查当前的剩余可用资源是否能满足某个进程的最大需求，

如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。

不断重复上述过程，看最终是否能让所有进程都加入安全序列。

安全性算法步骤：

（1）设置两个向量：

①**工作向量work**，它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，work = Available;

② **Finish**: 表示系统是否有足够的资源分配给进程，使之运行完成。

开始时先设 Finish[i] = false；当有足够资源分配给进程时，再令 Finish[i] = tur

3.8.4 安全性算法——实现过程

（2）从进程集合中找到一个能满足下述条件的进程：

① Finish[i] = false

② Need[i,j]≤ Work[j]

若找到，执行步骤（3），否则，执行步骤（4）。

（3）当进程P获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work[j]= Work [j]+ Allocation[i, j];

Finish[i]= true;

go to step 2:

（4）如果所有进程的 Finish[i] = true都满足，则表示系统处于安全状态：

否则，系统处于不安全状态

Work=Available;

Fininsh[i]=False;

对Finish[i]为False的进程：

```
if(Need[i,j] ≤Work[i])
{
    Work[i]=Work[i]+Allocation[i,j];
    Finish[i]=True;
}
```

如果所有的Finish[i]都为True，表示系统处于安全状态。

3.8.5 银行家算法举例：

系统中有5个进程{P0，P1，P2，P3，P4}和三类资源{A，B，C}，各资源的数量分别为10、5、7，**T0时刻**资源的分配情况如下所示。

进程 \ 资源情况	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
P1	3 2 2	2 0 0	1 2 2	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

(1) 判断系统此时是否处于安全状态？（即寻找系统中是否存在安全调度序列）

资源情况 / 进程	Work	Need	Allocation	Work +Alloction
	A B C	A B C	A B C	A B C
P1	3 3 2	1 2 2	2 0 0	5 3 2
P3	5 3 2	0 1 1	2 1 1	7 4 3
P4	7 4 3	4 3 1	0 0 2	7 4 5
P0	7 4 5	7 4 3	0 1 0	7 5 5
P2	7 5 5	6 0 0	3 0 2	10 5 7

存在安全调度序列P1，P3，P4，P0，P2，因此系统是安全的。

(2) 若P1发出请求向量Request1(1，0，2)，能否为其分配资源?

① Request1(1，0，2)≤Need1(1，2，2)

② Request1(1，0，2)≤Available(3，3，2)

③ 系统尝试为P1分配资源，修改相关的数据结构。

资源情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	2	3	0
P1	3	2	2	3	0	2	0	2	0			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

④ 判断此时系统是否安全：

资源情况 / 进程	Work	Need	Allocation	Work +Alloction
	A B C	A B C	A B C	A B C
P1	2 3 0	0 2 0	3 0 2	5 3 2
P3	5 3 2	0 1 1	2 1 1	7 4 3
P4	7 4 3	4 3 1	0 0 2	7 4 5
P0	7 4 5	7 4 3	0 1 0	7 5 5
P2	7 5 5	6 0 0	3 0 2	10 5 7

存在安全调度序列P1，P3，P4，P0，P2，因此系统是安全的。

（3）若P4发出请求向量Request4(3，3，0)，能否为其分配资源?

① Request4(3，3，0)≤Need4(4，3，1)

② Request4(3，3，0)≥Available(2，3，0)

此时可用资源无法满足请求资源的要求，因此P4进程等待。

(4) 若P0发出请求向量Request0(0，2，0)，能否为其分配资源?

① Request0(0，2，0)≤Need1(7，4，3)

② Request0(0，2，0)≤Available(2，3，0)

③ 系统尝试为P0分配资源，修改相关的数据结构。

进程 \ 资源情况	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	3	0	7	2	3	2	1	0
P1	3	2	2	3	0	2	0	2	0			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

④ 安全性检查： 此时可用资源无法满足任何进程的需求，因此不能为进程P0分配资源。

基本事实

- 如果一个系统在安全状态，就没有死锁
- 如果一个系统不是处于安全状态，就有可能死锁
- 死锁避免 $\Rightarrow$ 确保系统永远不会进入不安全状态

知识回顾与总结

银行家算法数据结构：

长度为 m 的一维数组 Available 表示还有多少可用资源

n*m 矩阵 **Max** 表示各进程对资源的最大需求数

n*m 矩阵 **Allocation** 表示已经给各进程分配了多少资源

Max – Allocation = Need 矩阵表示各进程最多还需要多少资源

n*m 矩阵 **Request** 表示进程此次申请的各种资源数

银行家算法步骤：

①检查此次申请是否超过了之前声明的最大需求数

②检查此时系统剩余的可用资源是否还能满足这次请求

③试探着分配，更改各数据结构

④用安全性算法检查此次分配是否会导致系统进入不安全状态

知识回顾与总结

安全性算法步骤：

①设置两个向量：初始**Work=Available；Fininsh[i]=False；**

②从进程集合中找到一个能满足下述条件的进程： ① Finish[i] = false ② Need[i,j]≤ Work[j]

若找到，执行步骤（3），否则，执行步骤（4）

③当进程获得资源顺利执行后，释放资源，执行：

Work[j]= Work [j]+ Allocation[i, j]; Finish[i]= true; go to step 2:

④如果所有进程的 Finish[i] = true都满足，则表示系统处于安全状态：

否则，系统处于不安全状态

系统处于**不安全状态未必死锁**，但**死锁时一定处于不安全状态**。系统**处于安全状态一定不会死锁**。

练习题

假定系统中有5个进程P0、P1、P2、P3、P4和4种资源A、B、C、D，若出现如表所示资源分配情况。

进程	已分配到资源	尚需资源需求	当前可用资源数
P_0	（1，1，1，0）	（0，3，3，1）	（0，3，2，2）
P_1	（0，2，3，1）	（0，3，4，2）	
P_2	（0，2，1，2）	（1，0，3，4）	
P_3	（0，3，1，0）	（0，3，2，0）	
P_4	（1，0，2，1）	（0，4，2，3）	

问：（1）该状态是否安全？为什么？

（2）如果进程P0提出资源请求（0，0，0，1），系统能否将资源分配给它？为什么？

练习题

严格按照银行家算法及安全检查子算法进行。

（1）初始状态如表所示。

进程	Work	Need	Allocation	Work +Allocation	Finish
P_3	0，3，2，2	0，3，2，0	0，3，1，0	0，6，3，2	True
P_0	0，6，3，2	0，3，3，1	1，1，1，0	1，7，4，2	True
P_1	1，7，4，2	0，3，4，2	0，2，3，1	1，9，7，3	True
P_4	1，9，7，3	0，4，2，3	1，0，2，1	2，9，9，4	True
P_2	2，9，9，4	1，0，3，4	0，2，1，2	2，11，10，6	True

存在一个安全序列P3，P0，P1，P4，P2，所以，该状态是安全的。

练习题

（2）Request0(0，0，0，1)<Need0(0，3，3，1)
Request0(0，0，0，1)<Available(0，3，2，2)
故尝试将资源分配给P0，修改P0对应资源，P0对应的Need（0，3，3，0），Allocation（1，1，1，1），系统的Available为（0，3，2，1）。
进程资源分配情况如表所示。

进程	Work	Need	Allocation	Work +Allocation	Finish
P_3	0，3，2，1	0，3，2，0	0，3，1，0	0，6，3，1	True
P_0	0，6，3，1	0，3，3，0	1，1，1，1	1，7，4，2	True
P_1	1，7，4，2	0，3，4，2	0，2，3，1	1，9，7，3	True
P_4	1，9，7，3	0，4，2，3	1，0，2，1	2，9，9，4	True
P_2	2，9，9，4	1，0，3，4	0，2，1，2	2，11，10，6	True

存在一个安全序列P3、P0、P1、P4、P2，该状态是安全的，所以，可以实施分配。

3.9.1 死锁的检测

为了能对系统中是否已发生了死锁进行检测，在系统中必须：

① 保存有关资源的请求和分配信息；

② 提供一种算法，它利用这些信息来检测系统是否已进入死锁状态。

1. **资源分配图**(Resource Allocation Graph)

系统死锁，可利用资源分配图来描述。

3.9.1 死锁的检测

该图是由一组结点N和一组边E所组成的一个对偶G = (N, E)，它具有下述形式的定义和限制：

(1) 把N分为两个互斥的子集，即一组进程结点P={P1, P2, ..., Pn}和一组资源结点R={R1, R2, ..., Rn}，N = P∪R。在下图所示的例子中，P = {P1, P2}，R = {R1, R2}，N = {R1, R2}∪{P1, P2}。

(2) 凡属于E中的一个边e∈E，都连接着P中的一个结点和R中的一个结点，e = {Pi, Rj}是资源请求边，由进程Pi指向资源Rj，它表示进程Pi请求一个单位的Rj资源。E = {Rj, Pi}是资源分配边，由资源Rj指向进程Pi，它表示把一个单位的资源Rj分配给进程Pi。

3.9.1 死锁的检测

图中示出了两个请求边和两个分配边，

即E = {(P1, R2), (R2, P2), (P2, R1), (R1, P1)}。

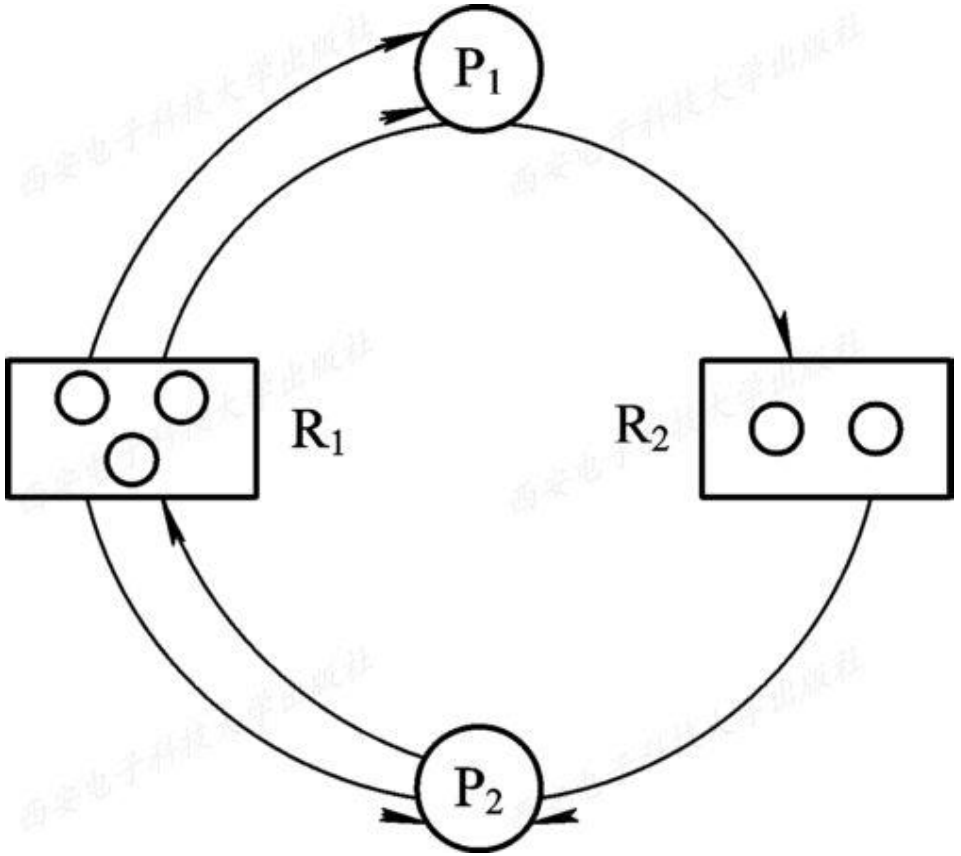


图3-19 每类资源有多个时的情况

3.9.1 死锁的检测

2．死锁定理

我们可以利用把资源分配图加以简化的方法(图3-19)，来检测当系统处于S状态时，是否为死锁状态。简化方法如下：

① 在资源分配图中，找出一个既不阻塞又非独立的进程结点p_i。在顺利的情况下，p_i可获得所需资源而继续运行，直至运行完毕，再释放其所占有得全部资源，这相当于消去p_i所有的请求边和分配边，使之成为孤立的结点；

② p_1释放资源后，便可使p_2获得资源而继续运行，直到p_2完成后又释放出它所占有的全部资源；

③ 在进行一系列的简化后，若能消去图中所有的边，使所有进程都成为孤立结点，则称该图是可完全简化的；若不能通过任何过程使该图完全简化，则称该图是不可完全简化的。

3.9.1 死锁的检测

在图3-20(a)中，将P1的两个分配边和一个请求边消去，便形成图(b)所示的情况。 P1释放资源后，便可使P2获得资源而继续运行，直至P2完成后又释放出它所占有的全部资源，形成图(c)所示的情况，即将P2的两条请求边和一条分配边消去。

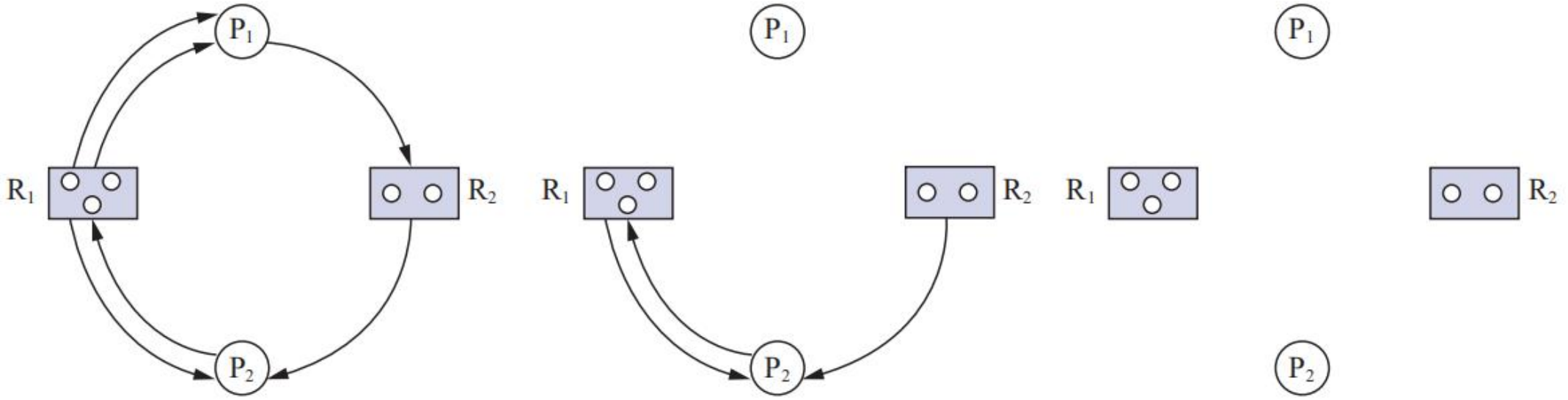


图3-20 资源分配图的简化

3.9.1 死锁的检测

S为死锁状态的充分条件是：当且仅当S状态的资源分配图是不可完全简化的。该充分条件称为死锁定理。

练习题

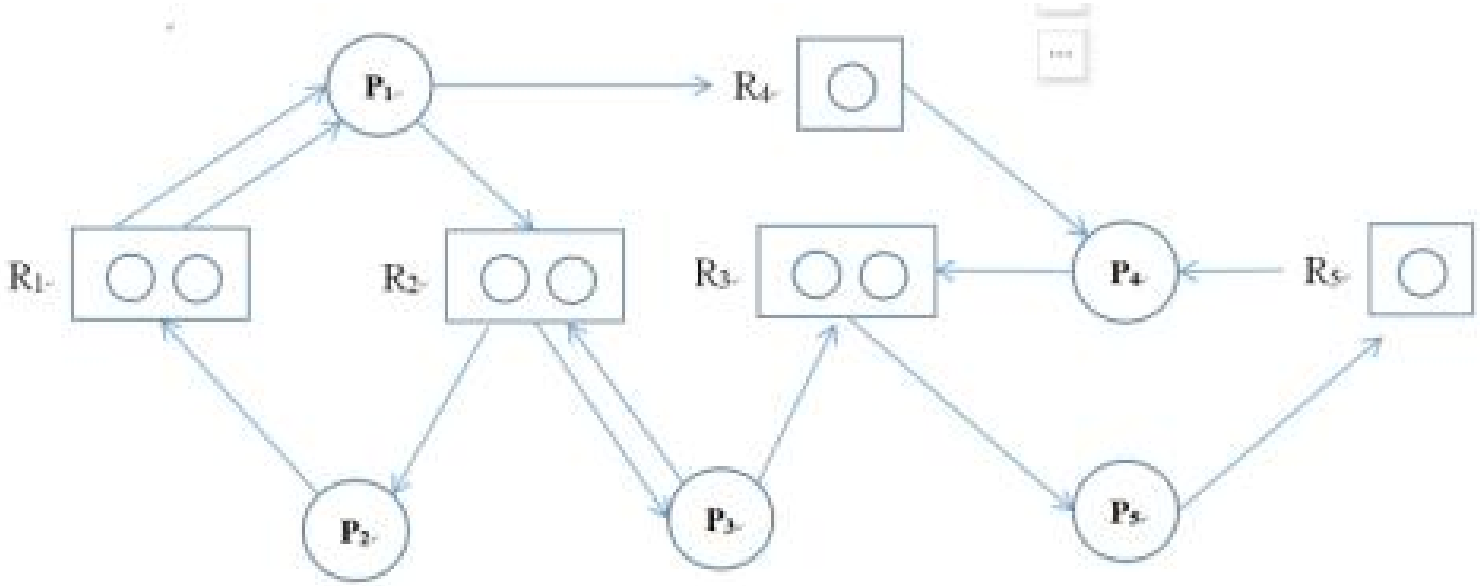
假设系统有5类独占资源：RI、R2、R3、R4、R5。各类资源分别有2、2、2、1、1个。系统有5个进程：P1、P2、P3、P4、P5。其中P1已占有2个R1，且申请1个R2和1个R4；P2已占有1个R2，且申请1个RI；P3已占有1个R2，且申请1个R2和1个R3；P4已占有1个R4和1个R5，且申请1个R3；P5已占有1个R3，且申请1个R5。

（1）试画出该时刻的资源分配图。

（2）什么是死锁定理？如何判断（1）中给出的资源分配图有无死锁？给出判断过程和结果。

解答

（1）该时刻的资源分配图如图所示。



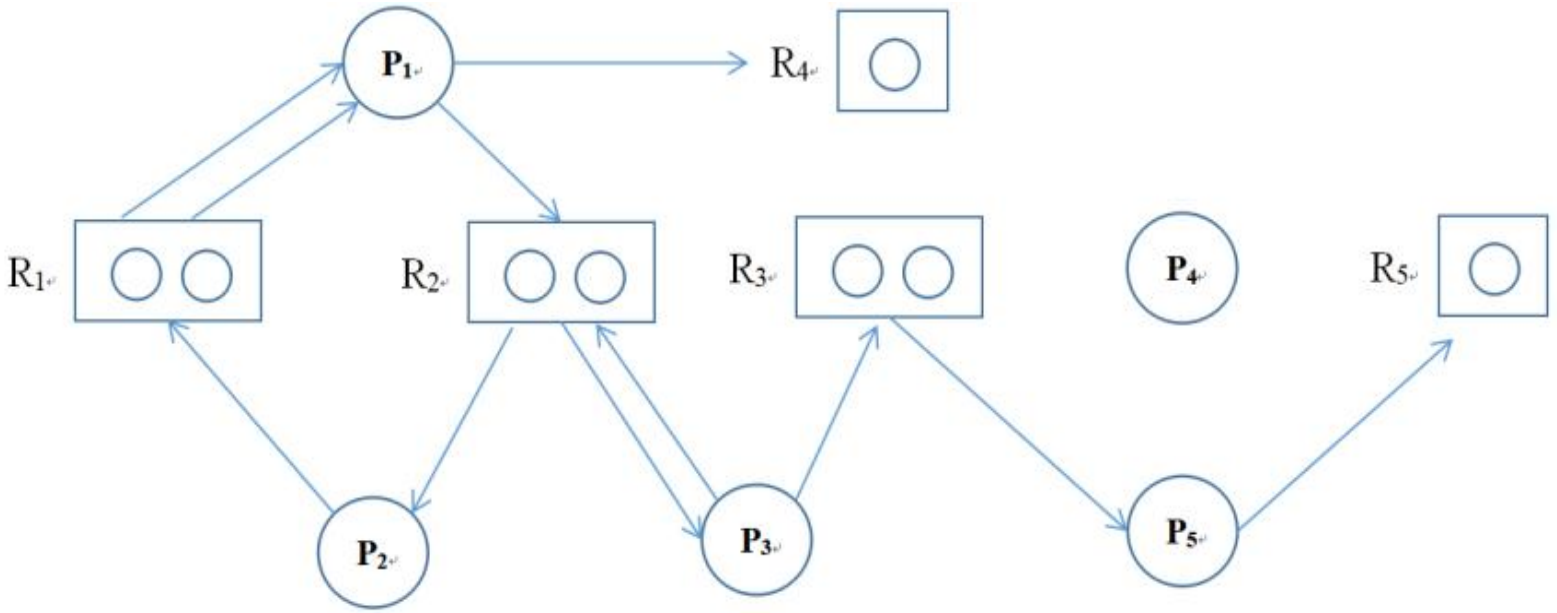
（2）系统状态S为死锁状态的充分条件：当且仅当S状态的资源分配图是不可完全简化的。该充分条件被称为死锁定理。

对于本题的情况，当前状态下系统可用资源数为（0，0，1，0，0），可以满足P4的申请需求，可将P4申请的资源进行分配，P4执行完毕后，系统的状态如图所示。

解答

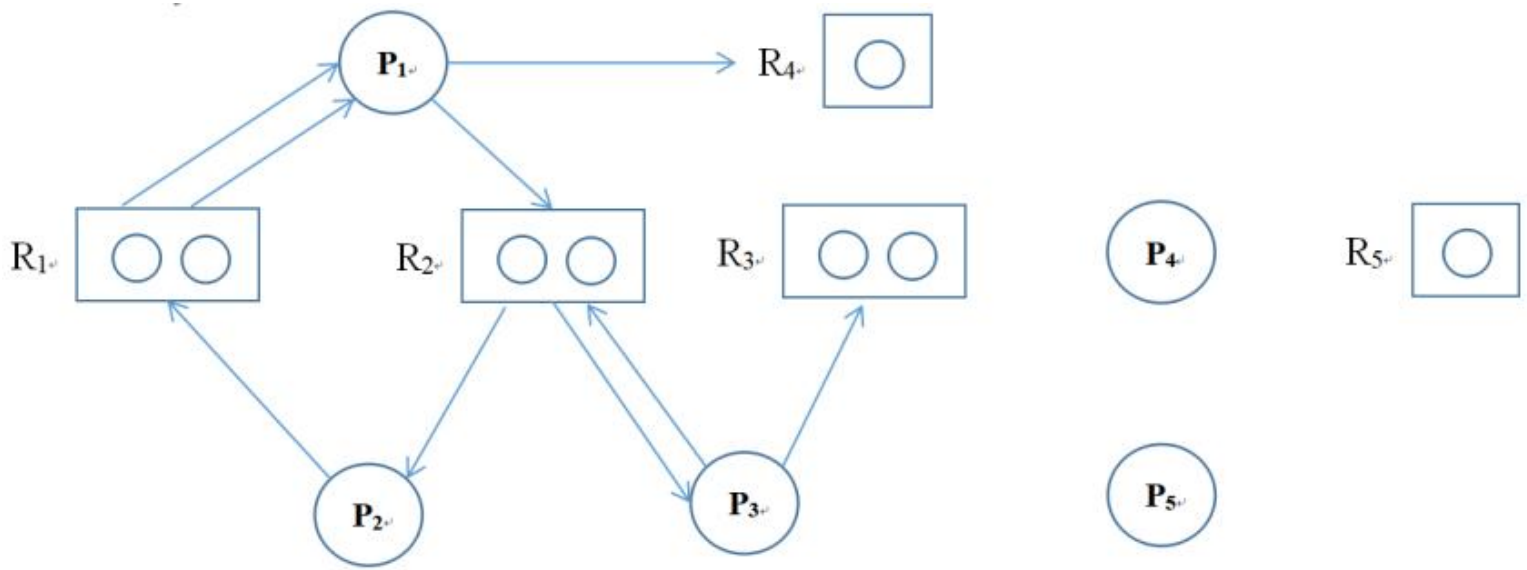
（2）系统状态S为死锁状态的充分条件：当且仅当S状态的资源分配图是不可完全简化的。该充分条件被称为死锁定理。

对于本题的情况，当前状态下系统可用资源数为（0，0，1，0，0），可以满足P4的申请需求，可将P4申请的资源进行分配，P4执行完毕后，系统的状态如图所示



解答

P4释放资源后，系统可用资源数变为（0，0，1，1，1），可以满足P5的申请需求，可将P5申请的资源进行分配，P5执行完毕后，系统的状态如图所示。



P5释放资源后，系统可用资源数变为（0，0，2，1，1），已不能满足任何进程的申请需求，系统当前资源分配图已经不能再简化，故系统处于死锁状态。

3.9.1 死锁的检测

3．死锁检测中的数据结构

死锁检测中的数据结构类似于银行家算法中的数据结构：

(1) 可利用资源向量Available，它表示了m类资源中每一类资源的可用数目。

(2) 把不占用资源的进程(向量Allocation=0)记入L表中，即Li∪L。

(3) 从进程集合中找到一个Requesti≤Work的进程，做如下处理：① 将其资源分配图简化，释放出资源，增加工作向量Work =Work + Allocation i。② 将它记入L表中。

(4) 若不能把所有进程都记入L表中，便表明系统状态S的资源分配图是不可完全简化的。因此，该系统状态将发生死锁。

3.9.1 死锁的检测

数据结构类似于银行家算法（基于资源分配图简化）

Work：=Available；

L：={L_i | $Allocation_i$=0∩$Request_i$=0}

for all L_i ∉ L do

begin

for all $Request_i$≤Work do

begin

Work：= Work + $Allocation_i$；

L_i∩L；

end

end

deadlock：=┐（L={P_1，P_2，…，P_n}）；

3.9.1 死锁的检测

01 让Work和Finish作为长度为m和n的向量初始化

(a) Work := Available

(b) For i = 1,2, …, n, if $Allocation_i \neq 0$, then

Finish[i] := false;otherwise, Finish[i] := true.

02 找到满足下列条件的下标i

(a) Finish[i] = false

(b) $Request_i \leq Work$

如果没有这样的i存在，转4

03 Work := Work + $Allocation_i$

Finish[i] := true

转 2.

04 如果有一些i, $1 \leq i \leq n$, Finish[i] = false, 则系统处在死锁状态。而且， 如果 Finish[i] = false, 则进程 P_i 是死锁的。

3.9.1 死锁的检测

检测算法的例子

五个进程P_n到P_4,三个资源类型A（7个实例），B（2个实例），C（6个实例）。
时刻T_0的状态：

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

对所有i，序列 $<P_0, P_2, P_3, P_1, P_4>$ 将导致$Finish[i]$ = true，因此死锁不存在。

3.9.1 死锁的检测

例子（续）

P2请求一个额外的C实例

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

系统的状态?

- 可以归还P_0所有的资源，但是资源不够完成其他进程的请求。
- 死锁存在，包括进程P1、P2、P3和P4。

3.9.2 死锁的解除

常用解除死锁的两种方法：

01 抢占资源。从一个或多个进程中抢占足够数量的资源给死锁进程，以解除死锁状态

02 终止或撤消进程。终止系统中一个或多个死锁进程，直到打破循环环路，使死锁状态消除为止。

- 终止所有死锁进程（最简单方法）
- 逐个终止进程（稍温和方法）

3.9.2 死锁的解除

中断所有的死锁进程。

一次中断一个进程，直到死锁环消失。

应该选择怎样的中断顺序，使“代价最小”？

- 进程的优先级；
- 进程需要计算多长时间，以及需要多长时间结束；
- 进程使用的资源，进程完成还需要多少资源；
- 进程是交互的还是批处理的。

练习题

假设系统中有下述解决死锁的办法：

（1）银行家算法；

（2）检测死锁，终止处于死锁状态的进程，释放该进程占有的资源；

（3）资源预分配。

简述哪种办法允许最大的并发性？请按“并发性”从大到小对上述3种办法排序

题中给出的3种办法中，检测死锁能允许更多的进程无等待地向前推进，并发性最大。因为该方法允许进程最大限度地申请并分配资源，直至出现死锁，再由系统解决。银行家算法允许进程自由申请资源，只是在某个进程申请时检查系统是否处于安全状态，若是，则可立即分配，若不是，才拒绝。并发性大小次于检测死锁的方法。最后是资源预分配，因为此方法要求进程在运行之前申请所需的全部资源才可以，这会使得许多进程因申请不到资源而无法开始，得到部分资源的进程因得不到全部资源也不释放已占用的资源，因此导致资源的浪费。因此，3种方法的并发性按从大到小排序为：检测死锁、银行家算法、资源预分配。

中科红旗之后，谁将再次勇战操作系统世界霸主？

提及北京中科红旗软件技术有限公司（简称中科红旗），读者可能有所不知，从事计算机信息技术领域具备国产自主知识产权的“红旗Linux系统”研发的中科红旗，曾长期占据国产操作系统市场老大位置。因此对于操作系统领域的相关人员而言，中科红旗绝对是一个“难以忘怀”的名字。

20世纪90年代初期，在微软公司大举进入中国市场并展示傲慢的信息技术（information technology，IT）巨头形象时，中科红旗被它的过分张扬与霸道所激怒，随即燃起了“起来，抵抗微软公司”的民族情绪，并联合北京金山办公软件股份有限公司（简称金山）等民族软件厂商抵抗微软公司的市场垄断行为。之后，红旗Linux系统在国内外渐渐具备一定的影响力，广泛

中科红旗之后，谁将再次勇战操作系统世界霸主？

应用于国家信息平台正版化、中国科学院、国家外汇管理局等全国性关键应用领域的国产信息化建设之中。联想、戴尔、惠普等公司的计算机也都曾预装过红旗Linux系统。

不料在种种因素的影响下，2014年2月10日，中科红旗突然宣布解散，引发业界轩然大波，令人感到万分遗憾！实际上，中科红旗的对手——微软公司是一家称霸全球、长期占据操作系统第一位置的企业，其庞大的生态系统，就像一艘巨型游轮，与之较量，实属不易。此外，令中科红旗十分尴尬的事实是，国人似乎已形成一种消费心理定势，宁可将预装的正版Linux系统卸载，也要安装上盗版的Windows系统！

中科红旗之后，谁将再次勇战操作系统世界霸主？

但是，这些都不是理由！当下，我们必须重整旗鼓、自主创新，要以蚕食桑叶之精神，与垄断型的操作系统争完善、比实用，待势均力敌之时再与它们抢市场、争地位。金山的WPS办公软件不就如此一路走来，现已可以取代微软公司的Office办公软件了吗？！

因此，无须畏惧现在的微软公司、苹果、谷歌等公司，只要我们坚持创新，机会的大门就会永远为我们敞开。卧薪尝胆，刻苦自励，坚信下一个勇于挑战Windows系统霸主地位的国产操作系统企业，定会浩然而至！

实验案例

利用银行家算法避免死锁

- 最有代表性的避免死锁的算法是 Dijkstra的银行家算法。起这样的名字是由于该算法原本是为银行系统设计的，以确保银行在发放现金贷款时，不会发生不能满足所有客户需要的情况。在OS中也可用它来实现避免死锁要每种资源类型的最大单元数目，其数目不应超过系统所拥有的资源总量。当进程请求一组资源时，系统必须首先确定是否有足够的资源分配给该进程。若有，再进一步计算在将这些资源分配给进程后，是否会使系统处于不安全状态。如果不会，才将资源分配给它，否则让进程等待。

1．银行家算法中的数据结构利用的资源、所有进程对资源的最大需求、系统中的资源分配，以及所有进程还需要多少

为了实现银行家算法，在系统中必须设置这样四个数据结构，分别用来描述系统中可

（1）可利用资源向量 Available。这是一个含有m个元素的数组，其中的每一个元素代资源的情况。表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 Available[j] = K，则表示系统中现有Rj类资源K个

（2）最大需求矩阵Max。这是一个nxm的矩阵，它定义了系统中n个进程中的每个进程对m类资源的最大需求。如果Max[i，j]＝K，则表示进程i需要Rj类资源的最大数目为K。

（3）分配矩阵 Allocation。这也是一个nxm的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 Allocation[i，j]＝K，则表示进程i当前已分得R1类资源的数目为K。

（4）需求矩阵Necd这也是一个nxm的矩阵，用以表示每一个进程尚需的各类资源数。如果Need[I,j]＝K，则表示进程i还需要Rj，类资源K个方能完成其任务上述三个矩阵间存在下述关系：Need[i, j]=Max[i, j]-allocation[i, j]

2．银行家算法

设 Request；是进程P的请求向量，如果 Request = K，表示进程P需要K个类型的资源。当P发出资源请求后，系统按下述步骤进行检查：

（1）如果 Request[i，j]≤Need[i，j]便转向步骤（2）：否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

（2）如果 Request[i， j] ≤ Available[i]，便转向步骤（3）;否则，表示尚无足够资源，Pi须等待。

（3）系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：Available[j]= Available[j]-Request[i，J];

Allocation [i, j] =Allocation[i, j]+Request[i，j]:

Need[i, j]= Need[i, j]-request[i，j];

（4）系统执行安全性算法，检查此次资源分配后系统是否处于安全状态。若安全，正式将资源分配给进程Pi，以完成本次分配：否则，将本次的试探分配作废，恢复原来资源分配状态，让进程Pi等待。

3．安全性算法

系统所执行的安全性算法可描述如下：

（1）设置两个向量：①工作向量work，它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，work = Available;(2) Finish:它表统示系统是否有足够的资源分配给进程，使之运行完成。开始时先做 Finishi[i] = false；当有足够资源分配给进程时，再令 Finish[i] = ture（2）从进程集合中找到一个能满足下述条件的进程：① Imishli = false② Need[i,j]≤ Work[j]若找到，执行步骤（3），否则，执行步骤（4）。

(3)当进程P获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：Work[j]= Work [j]+ Allocation[i, j];

Finish[i]= true;

go to step 2:

（4）如果所有进程的 Finishi = true都满足，则表示系统处于安全状态：否则，系统处于不安全状态

银行家算法之例

假设五个进程{P0，P1，P2，P3，P4}，三个资源{A,B,C}，各种资源的数量分别为10、5、7。T0时刻的资源分配表如下。

进程\资源情况	最大需求	分配	需求	可利用
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2 (2 3 0)
P1	3 2 2	2 0 0 (3 0 2)	1 2 2 (0 2 0)	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

T0时刻存的安全性：利用安全性算法对T0时刻的资源分配情况进行分析可知，T0时刻存在一个安全序列{P1,P3,P4,P2,P0}故系统是安全的

进程\资源情况	Work	Need	Allocation	Workl+Allocation	Final
	A B C	A B C	A B C	A B C	
P1	3 3 2	1 2 2	2 0 0	5 3 2	TRUE
P3	5 3 2	0 1 1	2 1 1	7 4 3	TRUE
P4	7 4 3	4 3 1	0 0 2	7 4 5	TRUE
P2	7 4 5	6 0 0	3 0 2	10 4 7	TRUE
P0	10 4 7	7 4 3	0 1 0	10 5 7	TRUE

P1请求资源，p1发出请求向量Request1（1，0，2），系统按银行家算法进行检查：

1.Request1(1,0,2)≤Need1(1,2,2);

2.Request1(1,0,2)≤Available1(3,3,2);

3.系统先假定可为p1分配资源,并修改Available1,Allocation1,和Need1向量,由此形成的资源变化情况在T0时刻为

P1			3	2	2	3	0	2	0	2	0		2	3	0

4.再利用安全性算法检查此时系统是否安全，如图。

进程\资源情况	Work	Need	Allocation	Workl+Allocation	Final
	A B C	A B C	A B C	A B C	
P1	2 3 0	0 2 0	3 0 2	5 3 2	TRUE
P3	5 3 2	0 1 1	2 1 1	7 4 3	TRUE
P4	7 4 3	4 3 1	0 0 2	7 4 5	TRUE
P0	7 4 5	7 4 3	0 1 0	7 5 5	TRUE
P2	7 5 5	6 0 0	3 0 2	10 5 7	TRUE

由所进行的安全性检查可知，可以找到一个安全性序列{P1,P3,P4,P2,P0}，因此系统是安全的，可以将p1所申请的资源分配给他。

（3）P_4 请求资源：P_4 发出请求向量$Request_4$ （3，3，0），系统按银行家算法进行检查：

1.$Request_4$ （3，3，0）≤$Need_4$ （4，3，1）；

2.$Request_4$ （3，3，0）＞Available（2，3，0），让P_4 等待。

（4）P_0 请求资源：P_0 发出请求向量$Request_0$ （0，2，0），系统按银行家算法进行检查：

1.$Request_0$ （0，2，0）≤$Need_0$ （7，4，3）；

2.$Request_0$ （0，2，0）≤Available（2，3，0）；

3.系统暂时先假定可为P_0 分配资源，并修改有关数据，如图

进程\资源情况	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	3	3	7	2	3	2	1	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

(5)进行安全性检查，可用资源Availabale（2，1，0）已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。