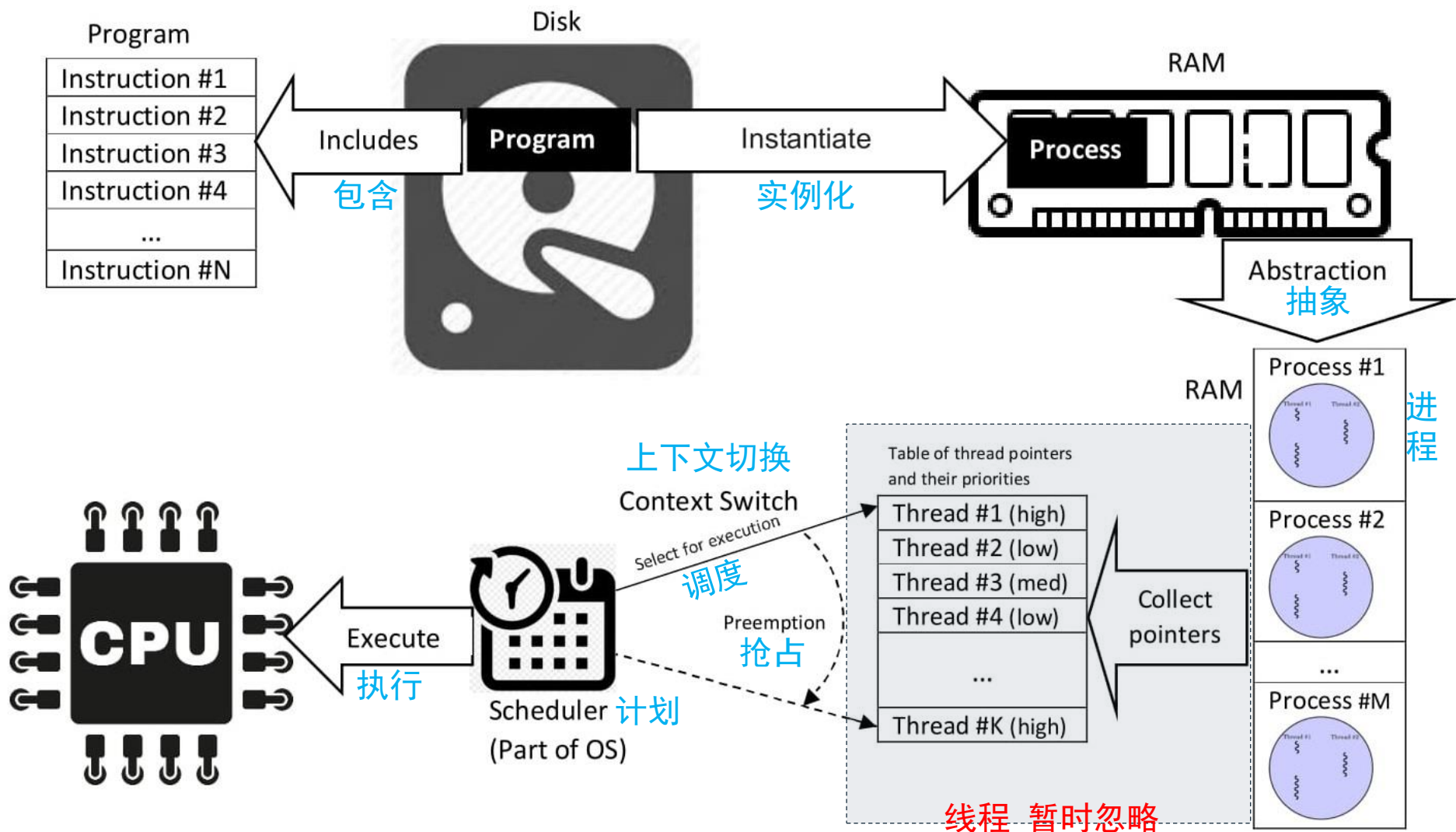


操作系统A

## 第二章

Email: [gaiazhao@ysu.edu.cn](mailto:gaiazhao@ysu.edu.cn)

# 程序的执行



这一系列的过程都是建立在【进程】上来实现的。

# 目录

CONTENTS

- 1 前趋图和程序执行
- 2 进程的描述
- 3 进程控制
- 4 进程同步
- 5 经典进程的同步问题（难点）
- 6 进程通信
- 7 线程及线程实现

## 第二章 进程的描述与控制

### 2.1 前趋图和程序执行

**问题：**多道批处理运行机制下，程序并发执行会存在什么问题？

**目标：**

- 掌握前趋图
- 了解程序的执行过程
- 重点掌握程序并发执行的过程和特征

## 2.1 前趋图和程序执行

### 顺序执行

- 内存中仅装入一道程序
- 独占系统所有资源
- 一个程序运行完后才能装入另一个程序执行

### 浪费资源

系统运行效率低

未配置OS 或 单道批处理系统

### 多道程序系统

### 并发执行

- 内存同时装入多个程序
- 共享系统资源
- 多个程序并发执行

提高资源利用率

提高系统运行效率

如何解决并发执行问题？

需要先了解程序并发执行的特点

### 前趋图

**前趋图** (Precedence Graph)，是指一个**有向无环图**，可记为 DAG(Directed Acyclic Graph)，它用于**描述进程之间执行的先后顺序**。

- 结 点 —— 进程、程序段、一条语句；
- 有向边 —— 两个结点间的偏序或前趋关系；

前趋关系用“ $\rightarrow$ ”来表示，若 $P_i$ 和 $P_j$ 存在前趋关系：

- $(P_i, P_j) \in \rightarrow$
- $P_i \rightarrow P_j$

则有：

- $P_i$ 是 $P_j$ 的直接前趋
- $P_j$ 是 $P_i$ 的直接后继

## 2.1.1 前趋图

每个结点具有一个重量或权重(Weight) —— 表示该结点所含有的程序量或程序的执行时间。

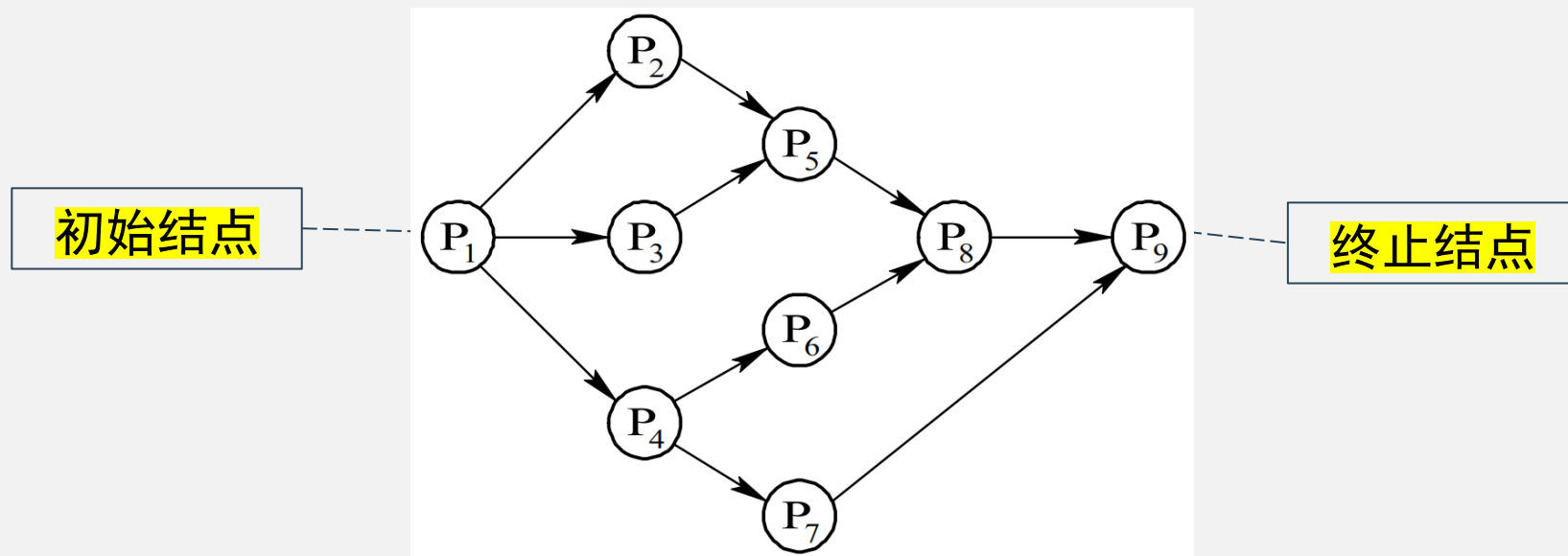


图2-1(a) 具有九个结点的前趋图



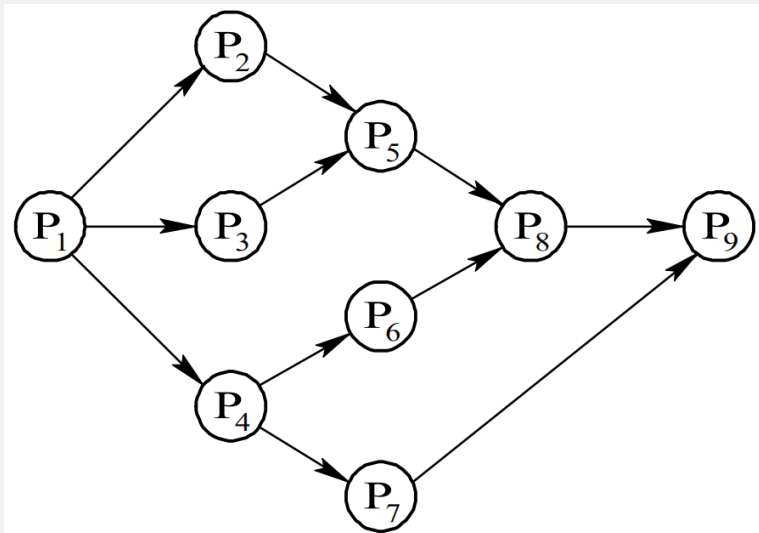


图2-1(a) 具有九个结点的前趋图

在图2-1(a)所示的前趋图中，存在着如下前趋关系：

$P_1 \rightarrow P_2$ ,  $P_1 \rightarrow P_3$ ,  $P_1 \rightarrow P_4$ ,  $P_2 \rightarrow P_5$ ,  
 $P_3 \rightarrow P_5$ ,  $P_4 \rightarrow P_6$ ,  $P_4 \rightarrow P_7$ ,  $P_5 \rightarrow P_8$ ,  
 $P_6 \rightarrow P_8$ ,  $P_7 \rightarrow P_8$ ,  $P_8 \rightarrow P_9$

或表示为：

$$P = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

$$= \{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3, P_5), (P_4, P_6), (P_4, P_7), (P_5, P_8), \\ (P_6, P_8), (P_7, P_8), (P_8, P_9)\}$$

- 前趋图中不允许有循环
- 否则必然会产生不可能实现的前趋关系

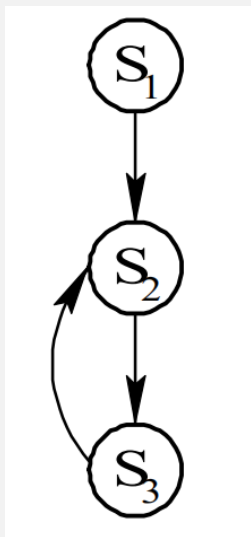


图2-1(b) 具有循环的非前趋图

如图2-1(b)所示的前趋关系中就存在着循环。

- 一方面要求在S3开始执行之前，S2必须完成；
- 另一方面又要求在S2开始执行之前，S3必须完成。

显然，这种关系是不可能实现的。

$$S_2 \rightarrow S_3, S_3 \rightarrow S_2$$

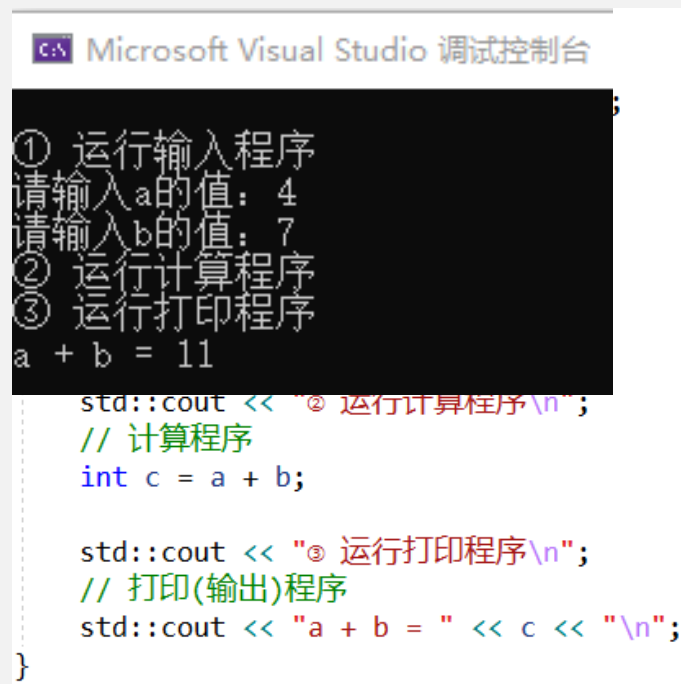
## 2.1.2 程序顺序执行

### 1. 程序的顺序执行

通常，一个应用程序由若干个程序段组成，每一个程序段完成特定的功能，它们在执行时，都需要按照某种先后次序顺序执行，仅当前一程序段执行完后，才运行后一程序段。

例：一个计算程序执行过程

- ① 运行输入程序：输入用户的程序和数据
- ② 运行计算程序：对输入的数据进行计算
- ③ 运行打印程序：打印计算结果



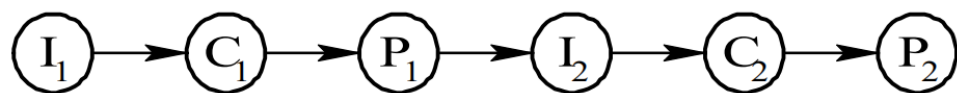
```
Microsoft Visual Studio 调试控制台  
① 运行输入程序  
请输入a的值: 4  
请输入b的值: 7  
② 运行计算程序  
③ 运行打印程序  
a + b = 11  
std::cout << "◎ 运行计算程序\n";  
// 计算程序  
int c = a + b;  
  
std::cout << "◎ 运行打印程序\n";  
// 打印(输出)程序  
std::cout << "a + b = " << c << "\n";  
}
```

## 2.1.2 程序顺序执行

### 1. 程序的顺序执行

例：一个计算程序执行过程

- ① 运行输入程序 —— **I**
- ② 运行计算程序 —— **C**
- ③ 运行打印程序 —— **P**



(a) 程序的顺序执行

上述的三个程序段间就存在着这样的前趋关系： $I_i \rightarrow C_i \rightarrow P_i$ ，其执行的顺序可用前趋图2-2(a)描述。

例：一个程序段的执行

S1:  $a = x + y$ ;

S2:  $b = a - 5$ ;

S3:  $c = b + 1$ ;



(b) 三条语句的顺序执行

S2必须在S1后(即a被赋值)执行，语句S3也只能在b被赋值后才能执行。

因此，三条语句存在着这样的前趋关系： $S1 \rightarrow S2 \rightarrow S3$ ，

## 2.1.2 程序顺序执行

### 2. 程序顺序执行时的特征



## 2.1.3 程序并发执行

### 1. 程序的并发执行

**背景：** 引入多道程序技术，使程序或程序段能并发执行。

所有程序都能并发执行吗？

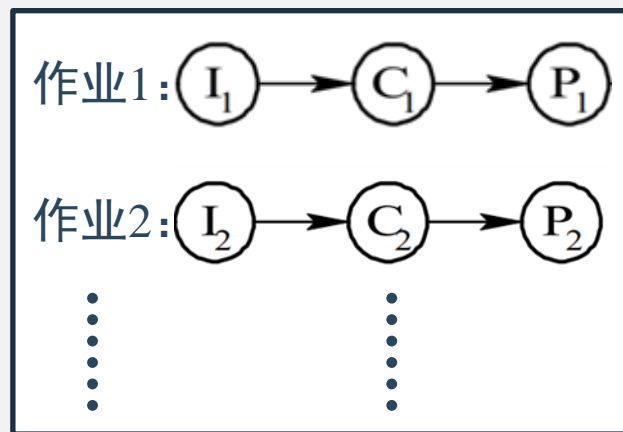
**条件：** 只有不存在前趋关系的程序之间才有可能并发执行。

例：一个计算程序执行，输入→计算→打印

- 存在 $I_i \rightarrow C_i \rightarrow P_i$ 的前趋关系
- 每个作业的输入、计算、打印  
三个程序段必须顺序执行

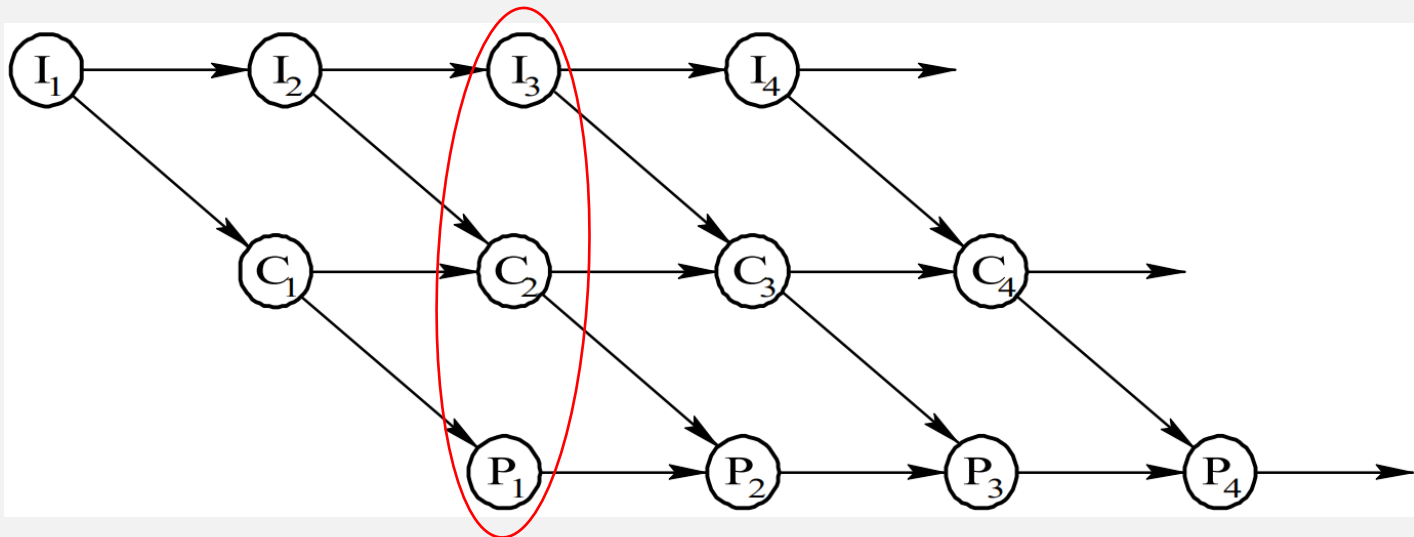
对一批这样的作业是否可以并发执行？

该怎么并发执行？



### 1. 程序的并发执行

一批 $I_i \rightarrow C_i \rightarrow P_i$ 作业的并发执行过程



由上图可以看出，存在前趋关系：

$$I_i \rightarrow C_i, I_i \rightarrow I_{i+1}, C_i \rightarrow P_i, C_i \rightarrow C_{i+1}, P_i \rightarrow P_{i+1},$$

而 $I_{i+1}$ 和 $C_i$ 及 $P_{i-1}$ 是重叠的，即在 $P_{i-1}$ 和 $C_i$ 以及 $I_{i+1}$ 之间，不存在前趋关系，可以并发执行。

## 2.1.3 程序并发执行

### 1. 程序的并发执行

例：具有下述四条语句的程序段

$S_1: a := x + 2$

$S_2: b := y + 4$

$S_3: c := a + b$

$S_4: d := c + b$

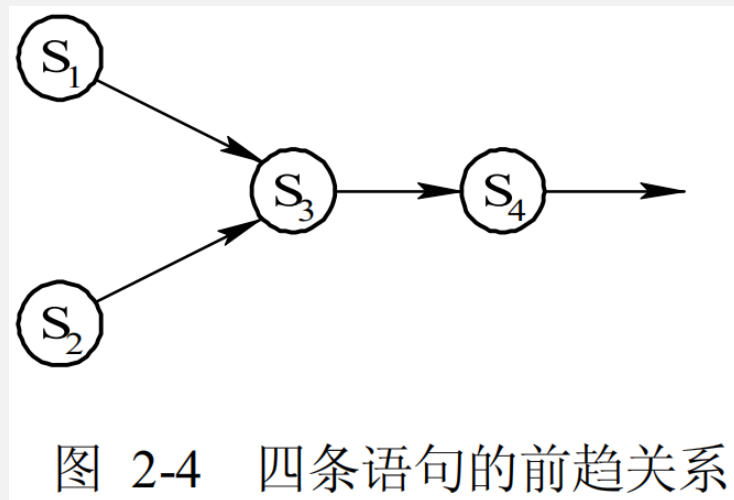


图 2-4 四条语句的前趋关系

可画出右图所示的前趋关系。

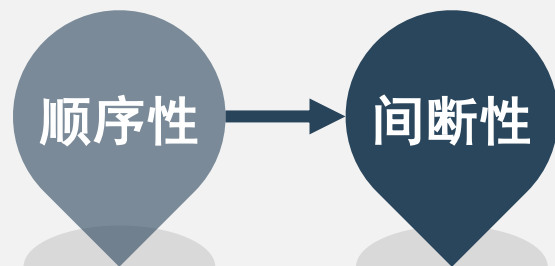
- $S_3$  必须在  $a$  和  $b$  被赋值后方能执行；
- $S_4$  必须在  $S_3$  之后执行；
- 但  $S_1$  和  $S_2$  则可以并发执行，因为它们彼此互不依赖。



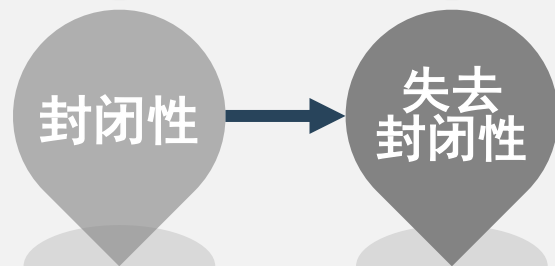
## 2.1.3 程序并发执行

### 2. 程序并发执行时的特征

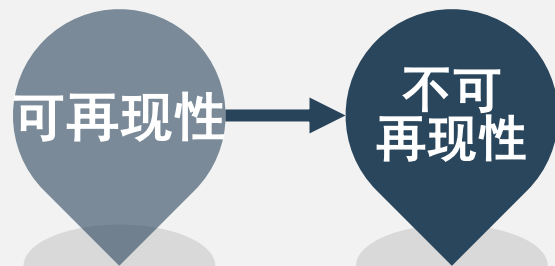
顺序执行      并发执行



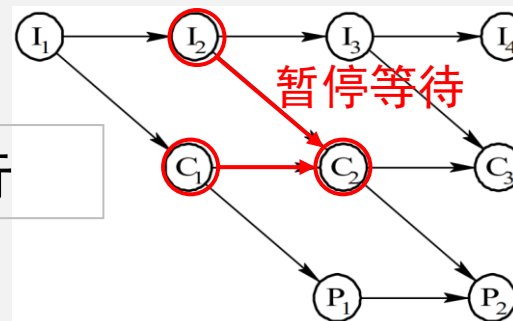
执行—暂停—执行



资源状态由多个程序改变  
运行环境受其他程序影响

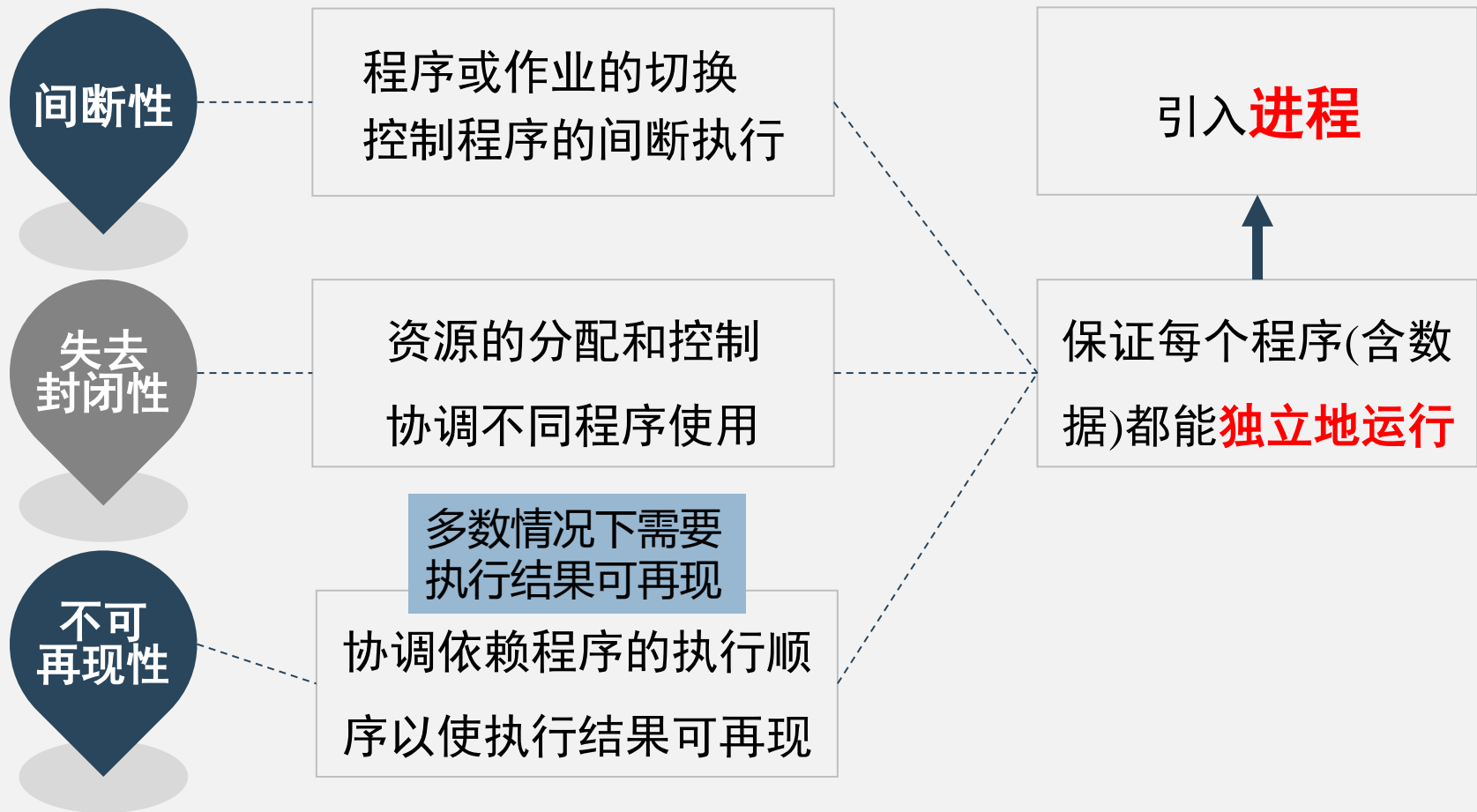


由于失去封闭性  
导致失去可再现性



参考课本P38例子

## 并发执行



## 第二章 进程的描述与控制

### 2.2 进程的描述

**问题：**多道程序环境下，通过引入进程，来保证每个程序(含数据)的独立运行——让程序能够并发执行。

**目标：**

- 掌握进程的定义和特征
- 掌握进程的基本状态和转换
- 掌握进程管理中的数据结构及作用

## 2.2.1 进程的定义和特征

### 1. 进程的定义

在多道程序环境下，程序的执行属于并发执行，此时它们将失去其封闭性，并具有间断性，以及其运行结果不可再现性的特征。

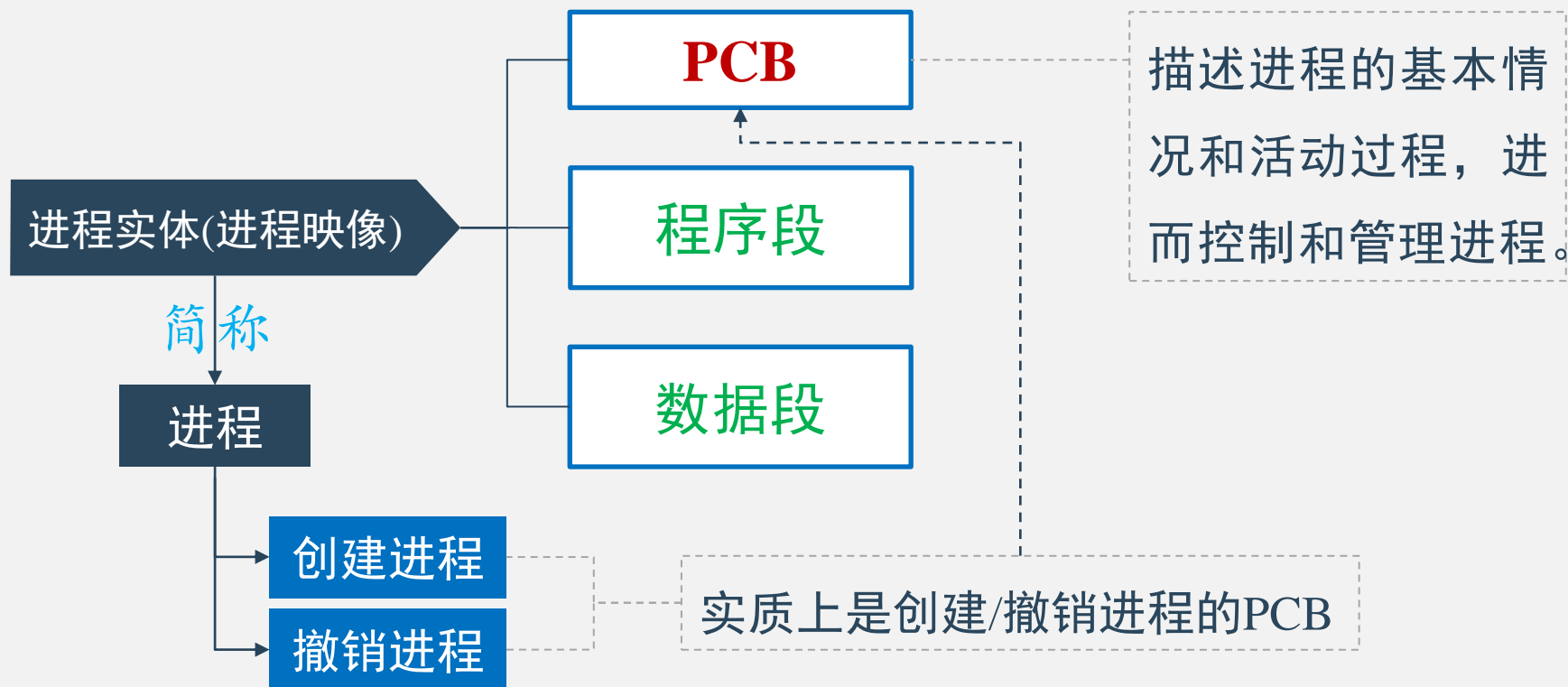


由此，决定了通常的程序是不能参与并发执行的，否则，程序的运行也就失去了意义。为了使程序并发执行，并且可以对并发执行的程序加以描述和控制，引入了“进程”的概念。

## 2.2.1 进程的定义和特征

### 进程控制块PCB

为了使参与并发的每个程序(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(Process Control Block, PCB)**。



## 2.2.1 进程的定义和特征

### 进程的定义

从不同的角度可以有不同的定义，其中较典型的定义有：

- (1) 进程是程序的一次执行。
- (2) 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。
- (3) 进程是具有独立功能的程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。

在引入进程实体的概念后，把传统 OS 中的进程定义为：“进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位”。

## 2.2.1 进程的定义和特征

### 2. 进程的特征 参考P39页来理解

#### 动态性

进程的实质是进程实体的执行过程。

由创建而产生  
由调度而执行  
由撤消而消亡  
程序是静态的  
进程是动态的

#### 并发性

多个进程实体同时存在内存，且能在一段时间内同时运行。

引入进程目的  
即使进程实体能并发执行

#### 独立性

进程实体是能独立运行、独立获得资源和独立接收调度的基本单位。

未建立PCB的程序不能作为独立单位运行。

#### 异步性

进程是按异步方式运行，按各自独立、不可预知的速度推进。

进程运行虽具异步性，仍能保证并发执行结果可再现。



## 2.2.2 进程的基本状态及转换

### 1. 进程的三种基本状态

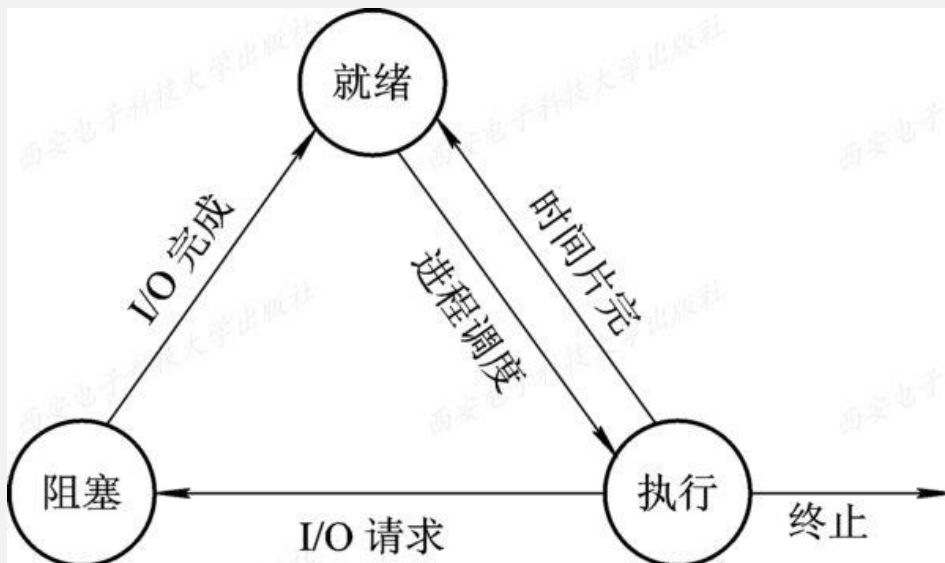
由于多个进程在并发执行时共享系统资源，致使它们在运行过程中呈现**间断性**的运行规律，所以进程在其**生命周期**内可能具有多种状态。一般分为以下三种基本状态：

- **就绪(Ready)状态**：已准备好运行，进程已分配到除CPU以外所有必要资源，再获得CPU即可运行；就绪进程按照一定策略排成就绪队列。
- **执行(Running)状态**：进程已经获得CPU，其程序正在执行；任一时刻，单处理机系统中只有一个进程处于执行状态；
- **阻塞(Block)状态**：正在执行的进程由于发生某事件(如I/O请求、申请缓冲区失败)等暂时无法继续执行的状态。

引起进程调度，OS把CPU分配给另一个就绪进程，让受阻进程处于**暂定状态**——即**阻塞状态**。阻塞进程通常排成一个或多个阻塞队列。

## 2.2.2 进程的基本状态及转换

### 2. 三种基本状态的转换



- 就绪 → 执行：就绪进程获得CPU之后便可执行；
- 执行 → 就绪：分配的时间片执行完毕被剥夺CPU而暂停执行；
- 执行 → 阻塞：发生某个事件(如访问临界资源而得不到)执行受阻；
- 阻塞 → 就绪：满足了阻塞进程执行的条件；
- 执行完全部程序则终止。

## 2.2.2 进程的基本状态及转换

### 3. 创建状态和终止状态 操作完整性、管理灵活性

正在创建过程中的进程处于创建状态

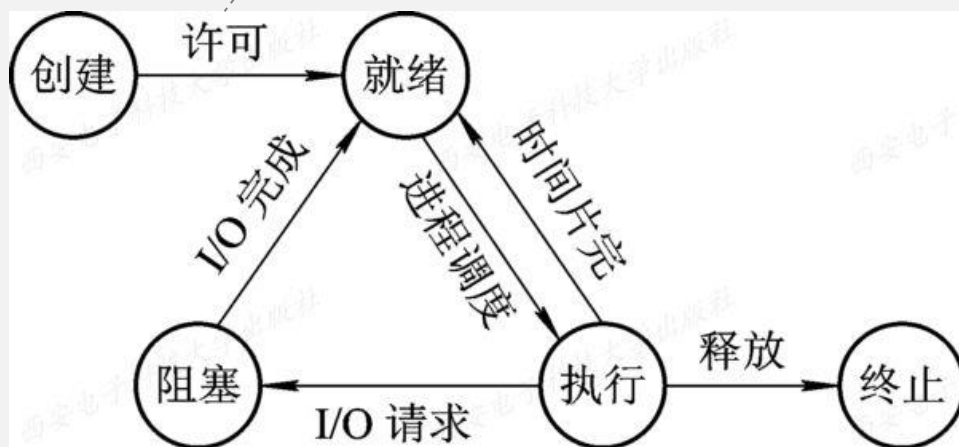
#### 进程创建过程

- (1) 由进程申请一个空白PCB，向PCB写入用于控制和管理进程的信息；
- (2) 为进程分配运行时所必须的资源；
- (3) 把该进程转入就绪状态并插入就绪队列中。

#### 创建状态

进程所需资源尚不能得到满足，比如系统无足够内存装入程序，创建工作未完成，不能被调度执行；

获得所需资源及完成PCB初始化工作



## 2.2.2 进程的基本状态及转换

### 3. 创建状态和终止状态

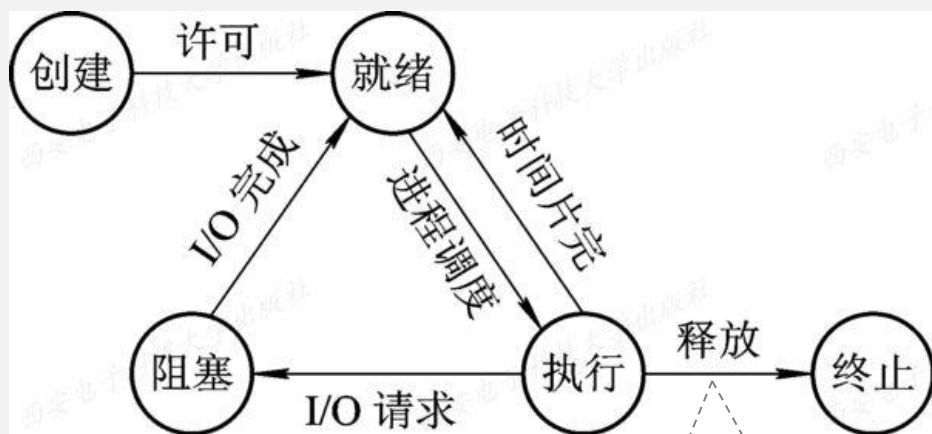
#### 进程终止过程

- (1) 等待OS进行善后处理；
- (2) 将PCB清零，将PCB空间归还系统。

删除进程

#### 终止状态

进程不能再执行，但在OS中依然保留一个记录，其中保存状态码和一些计时统计数据，以供其他进程收集。



- 进程到达自然结束点
- 出现无法克服的错误

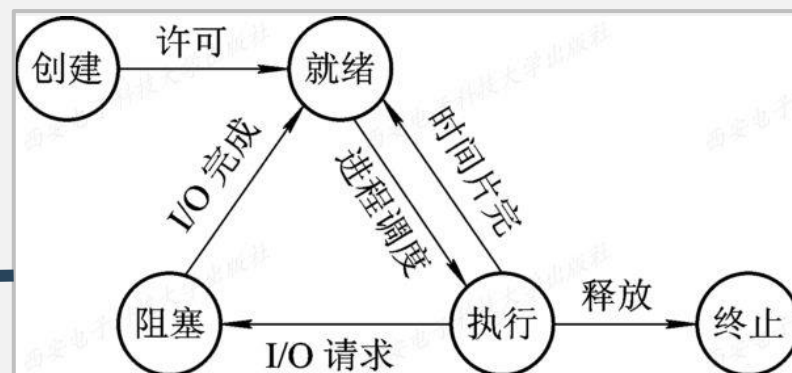
- 被操作系统所终结(如强制关机)
- 被其他有终止权的进程所终结(如任务管理器)

## 2.2.3 挂起操作和进程状态的转换

### 1. 挂起操作的引入

#### 挂起操作与激活操作

执行**挂起操作**后，进程进入静止状态。**把进程从内存移到外存的操作**。如果进程正在执行，则暂停执行；如果进程处于就绪状态，那么**暂不接受调度**。**激活操作**作用相反。



状态都是OS自动控制下的状态，无任何人为/其他干预。

#### 引入挂起的原因：

- 终端用户需要：发现程序运行问题，需要暂定调试修改，如Debug；
- 父进程请求：父进程需要考查和修改子进程，或协调多子进程间活动；
- 负荷调节的需要：系统运行负荷大，需要暂时挂起一些不重要进程；
- 操作系统的需要：挂起某些进程，以检查运行中资源使用情况或记账。

## 2.2.3 挂起操作和进程状态的转换

### 2. 引入挂起原语操作后三个进程状态的转换

挂起原语：Suspend，激活原语：Active。

- (1) **Readya**  $\xrightarrow{\text{Suspend}}$  **Readys**  
活动就绪  $\rightarrow$  静止就绪
- (2) **Blocka**  $\xrightarrow{\text{Suspend}}$  **Blocks**  
活动阻塞  $\rightarrow$  静止阻塞
- (3) **Readys**  $\xrightarrow{\text{Active}}$  **Readya**  
静止就绪  $\rightarrow$  活动就绪
- (4) **Blocks**  $\xrightarrow{\text{Active}}$  **Blocka**  
静止阻塞  $\rightarrow$  活动阻塞

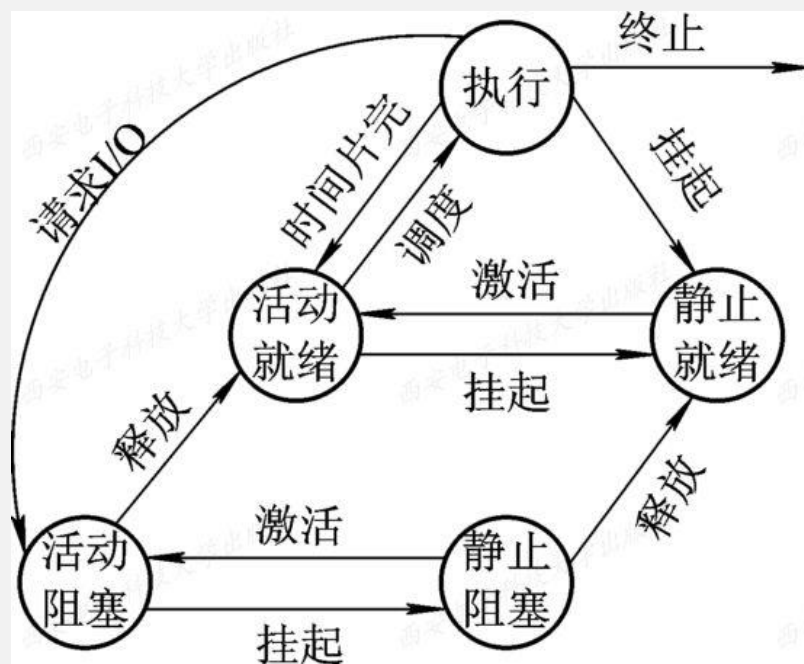


图2-7 具有挂起状态的进程状态图

挂起 — 静止\*\* — 暂不接受调度；  
激活 — 活动\*\* — 继续接收调度。

## 2.2.3 挂起操作和进程状态的转换

### 3. 引入挂起操作后五个进程状态的转换

与之前相比，增加下面几种状态的转换：

- (1) NULL → 创建
- (2) 创建 → Readya 活动就绪
- (3) 创建 → Readys 静止就绪
- (4) 执行 → 终止

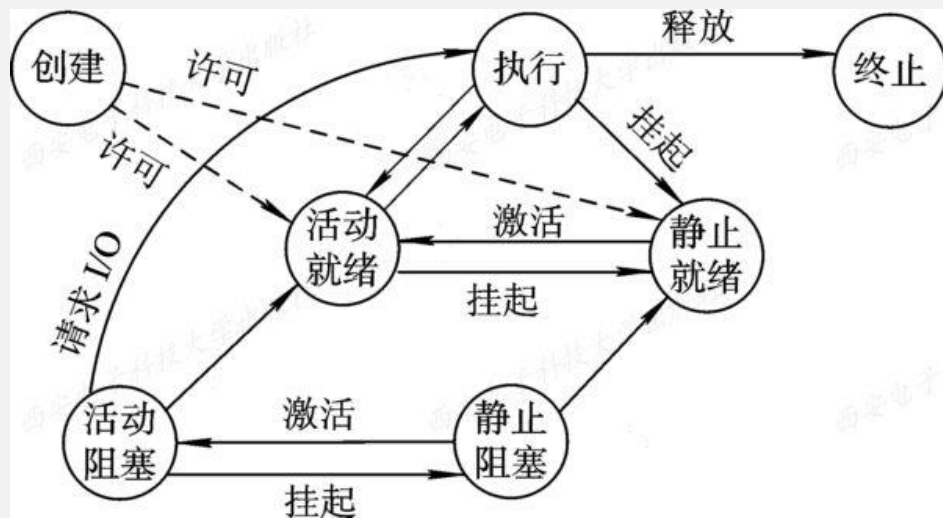


图2-8 具有创建、终止和挂起状态的进程状态图

创建 → 活动就绪：系统分配完所需资源，完成进程创建；

创建 → 静止就绪：不分配新建进程所需资源，主要是内存，未完成创建；

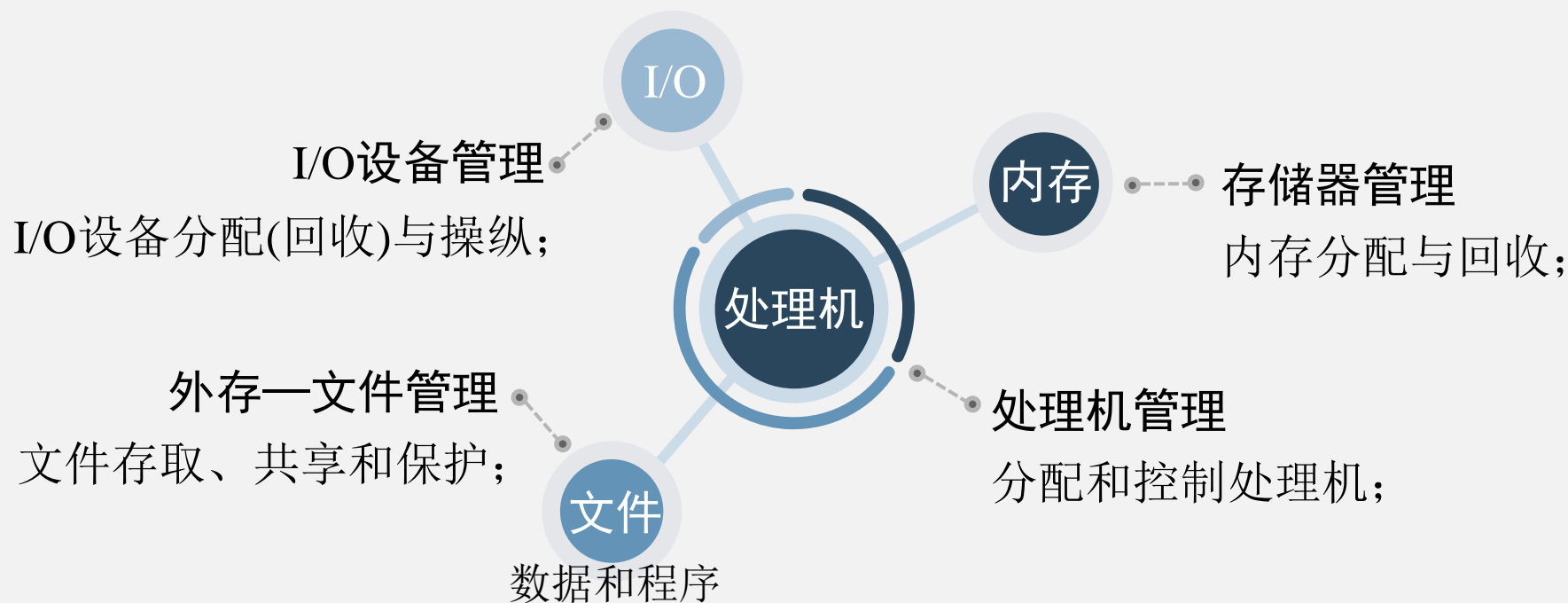
执行 → 终止：已完成执行、出现无法克服错误、被OS或其他进程终结。



## 2.2.4 进程管理中的数据结构

- 一方面, 为便于使用和管理, OS将各类资源抽象为各种数据结构;
- 另一方面, 为管理和协调计算机资源, 必须记录和查询:
  - 各种资源使用情况;
  - 各类进程运行情况。

OS对这些信息的组织和维护通过建立和维护相应的数据结构来实现。





## 2.2.4 进程管理中的数据结构

### 1. 操作系统中用于管理控制的数据结构

计算机系统中，对每个资源和每个进程都设置了一个数据结构，用于表征其实体，我们称之为资源信息表或进程信息表，其中包含了资源或进程的标识、描述、状态等信息以及一批指针。通过这些指针，可以将同类资源或进程的信息表，或者同一进程所占用的资源信息表分类链接成不同的队列，便于操作系统进行查找。

## 2.2.4 进程管理中的数据结构

### 1. 操作系统中用于管理控制的数据结构

OS管理的数据结构一般分为四类：

- 内存表
- 设备表
- 文件表
- 进程表

存储进程控制  
块PCB

如何管理进程呢？

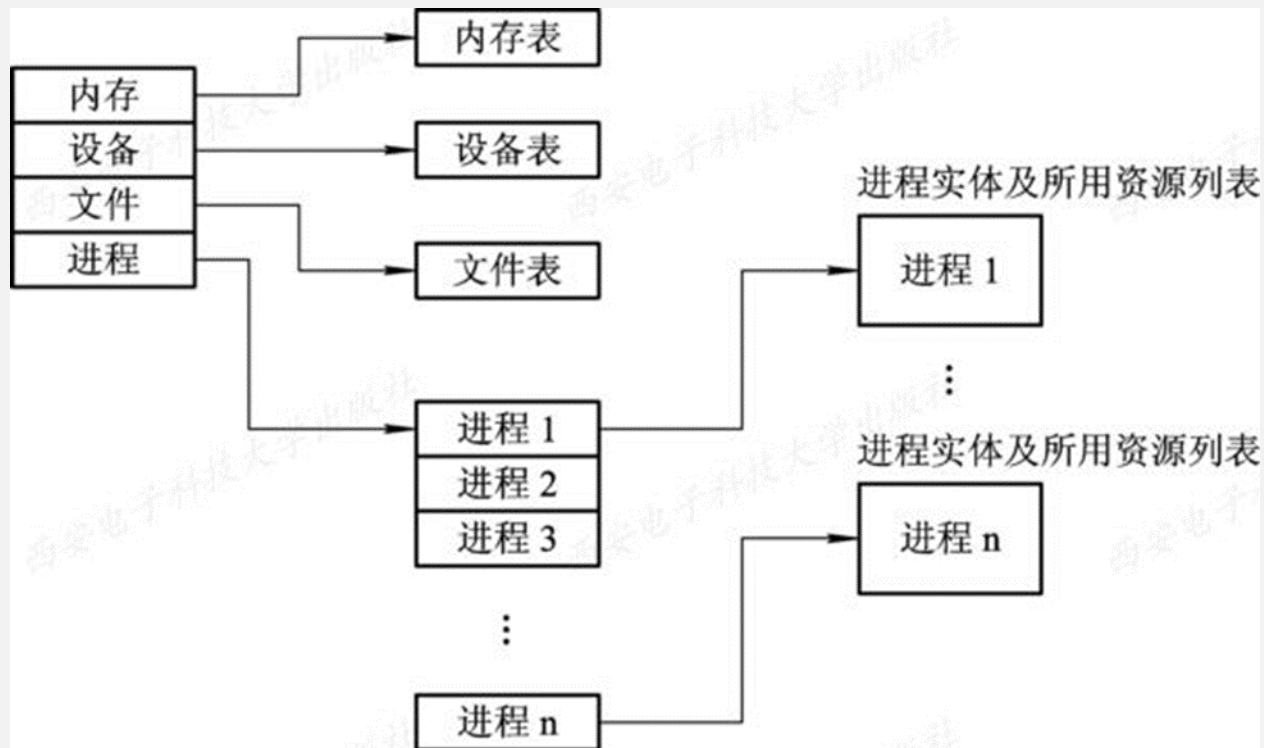


图2-9 操作系统控制表的一般结构

## 2.2.4 进程管理中的数据结构

### 2. 进程控制块PCB的作用

PCB是进程实体的一部分，记录了操作系统所需的、用于描述进程的当前情况以及管理进程运行的全部信息。（记录型数据结构）

#### PCB的作用

总体来说：使一个在多道程序环境下不能独立运行的程序(含数据)成为一个能独立运行的基本单位，一个能与其他进程并发执行的进程。

具体可从5个方面来分析！

进程实体

PCB

程序段

数据段



## 2.2.4 进程管理中的数据结构

### 2. 进程控制块PCB的作用

详细内容参考课本P44页

01 作为独立运行  
单位的标志

02 能实现间断运  
行方式

03 提供进程管理  
所需要的信息

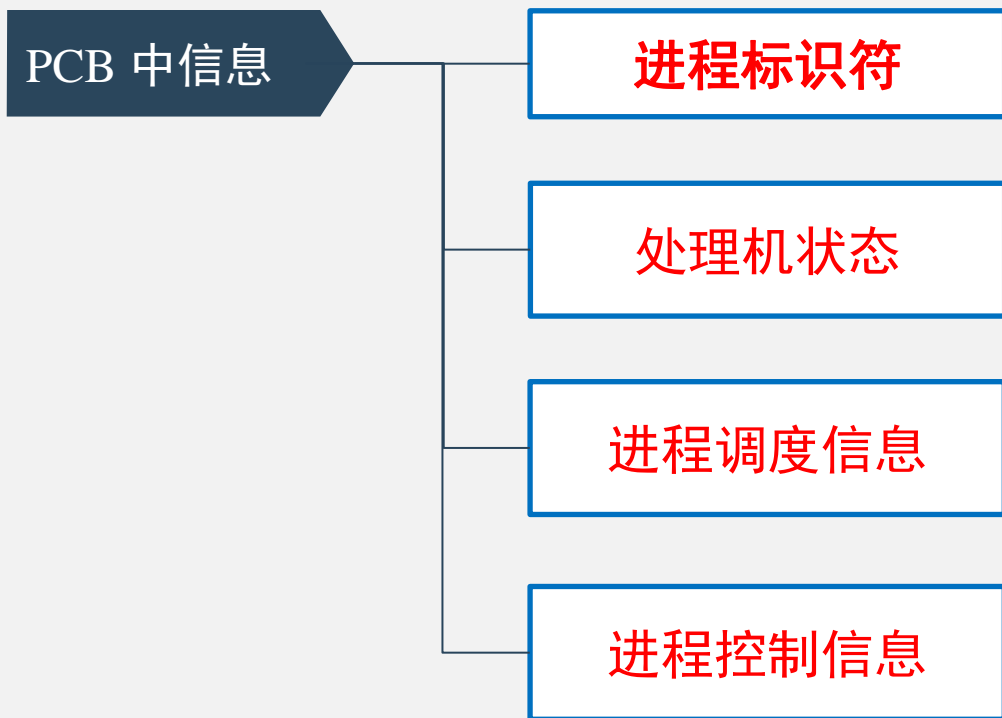
04 提供进程调度  
所需要的信息

05 实现与其他进  
程的同步与通信



## 2.2.4 进程管理中的数据结构

### 3. 进程控制块中的信息



### 3. 进程控制块中的信息



## 2.2.4 进程管理中的数据结构

### 3. 进程控制块中的信息

#### 处理机状态

处理机上下文，主要由处理机各种寄存器中的内容组成。

#### 通用寄存器

又称用户可视寄存器；  
是用户程序可以访问的，用于暂存信息；  
大多由8-32个，RISC结构计算机可超过100个。

#### 指令计数器

存放要访问的下一条指令的地址。

#### 程序状态字

程序状态字PSW；  
含有状态信息，如条件码、执行方式、中断屏蔽标志等。

#### 用户栈指针

每个用户进程都有一个或多个与之相关的系统栈，用于存放过程和系统调用参数及调用地址。栈指针指向栈顶。

当进程被切换时，处理机状态信息都必须保存在相应PCB中，以便重新执行时能从断点继续执行。

### 3. 进程控制块中的信息





### 3. 进程控制块中的信息



## 2.2.4 进程管理中的数据结构

### 3. 进程控制块中的信息

PCB是进程实体的一部分，记录了操作系统所需的、用于描述进程的当前情况以及管理进程运行的全部信息。



## 2.2.4 进程管理中的数据结构

### 4. 进程控制块的组织方式

在一个系统中，通常可拥有数十个、数百个乃至数千个PCB。为了能对它们加以有效的管理，应该用适当的方式将这些PCB组织起来。目前常用的组织方式有以下三种。

- 线性方式
- 链式方式
- 索引方式

## 2.2.4 进程管理中的数据结构

### 4. 进程控制块的组织方式

#### 线性方式

即将系统中所有的PCB都组织在一张线性表中，将该表的首址存放在内存的一个专用区域中。

该方式实现简单、开销小，但每次查找时都需要扫描整张表，因此适合进程数目不多的系统。

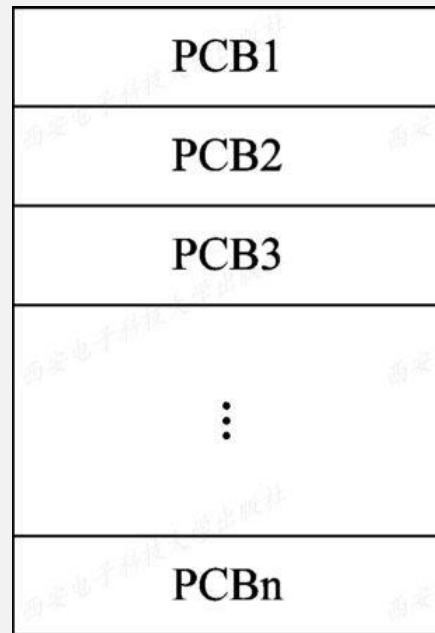


图2-10 PCB线性表示意图

## 2.2.4 进程管理中的数据结构

### 4. 进程控制块的组织方式

#### 链接方式

即把具有相同状态进程的PCB分别通过PCB中的链接字链接成一个队列。

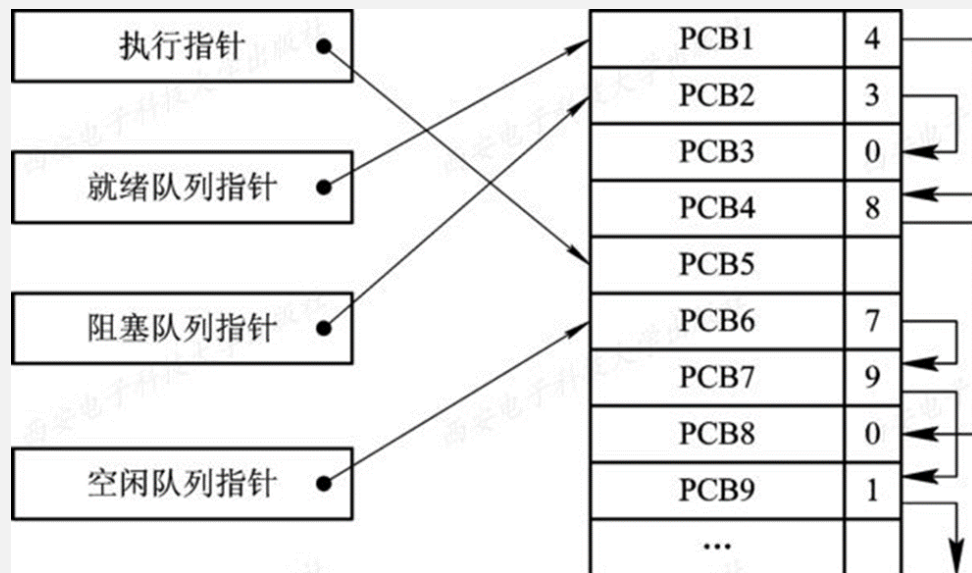


图2-11 PCB链接队列示意图

- 就绪队列：往往按进程的优先级将PCB从高到低进行排列，将优先级高的进程PCB排在队列的前面；
- 阻塞队列：根据其阻塞原因的不同，排成多个阻塞队列，如等待I/O操作完成的队列和等待分配内存的队列等。

### 4. 进程控制块的组织方式

#### 索引方式

即根据所有进程状态的不同，建立几张索引表，例如：就绪索引表、阻塞索引表等，并把各索引表在**内存的首地址**记录在内存的一些专用单元中。在每个索引表的表目中，记录具有相应状态的某个PCB在PCB表中的地址。

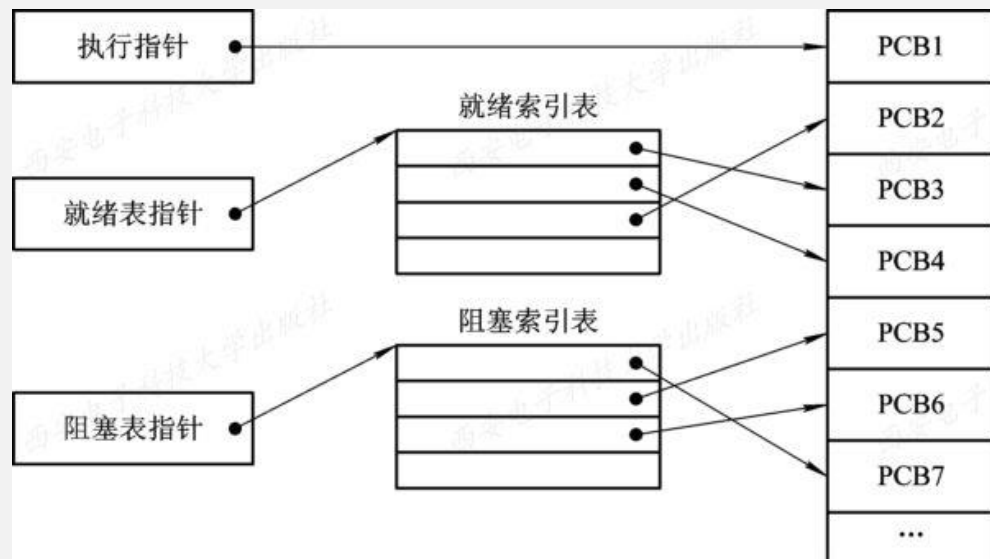
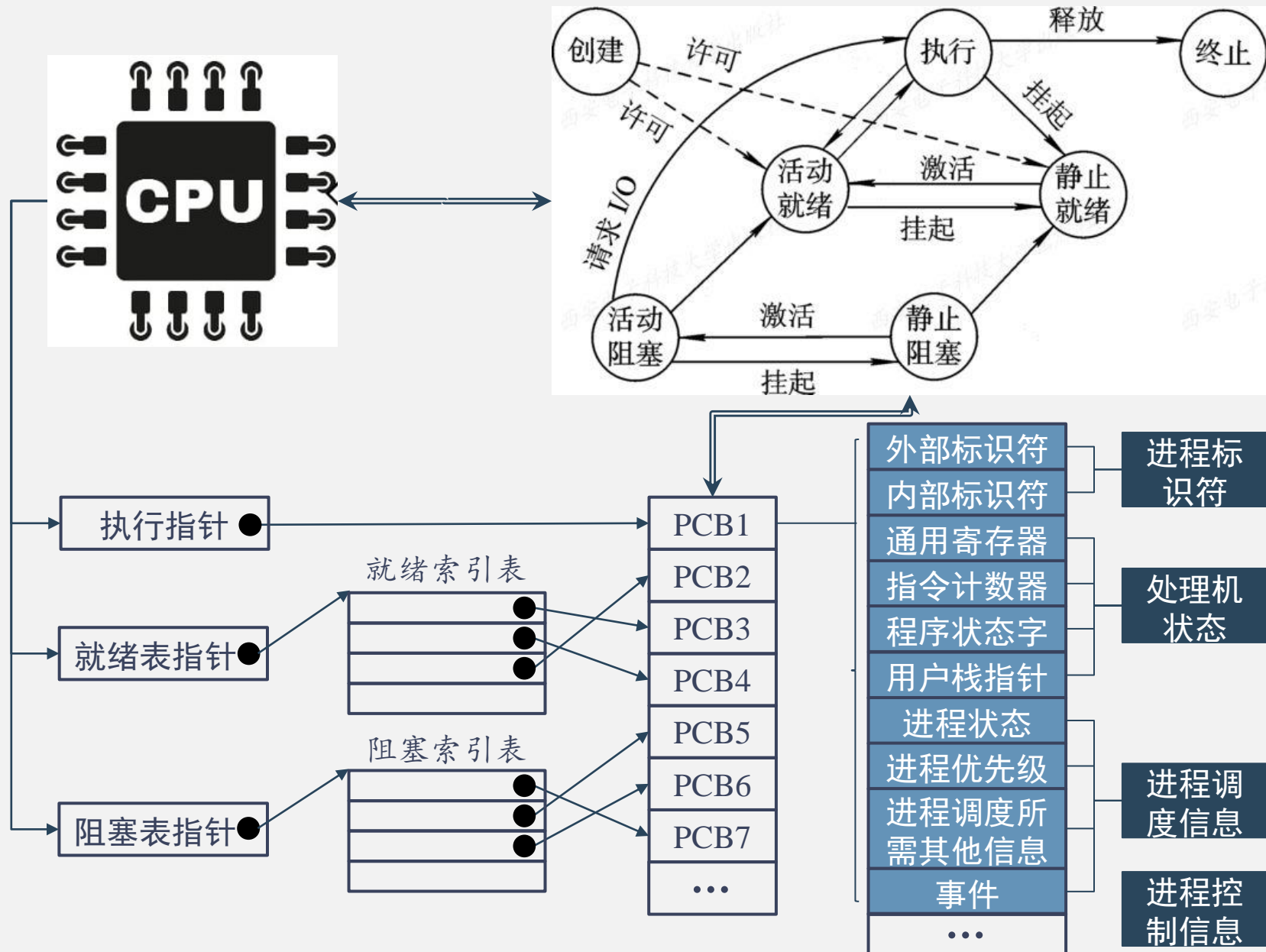


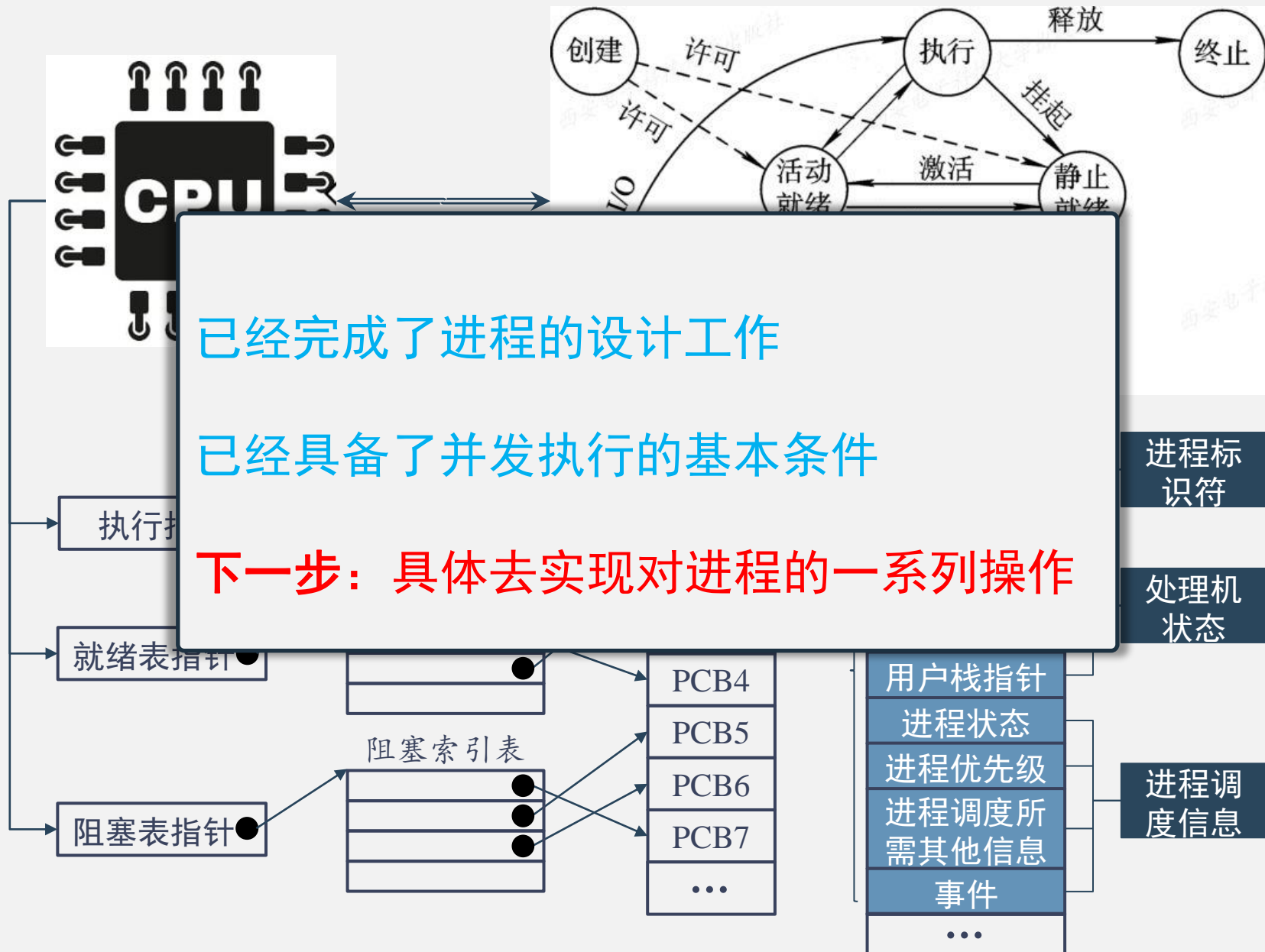
图2-12 按索引方式组织PCB

汲取了线性方式和链式方式的优点。

## 2.2 进程的描述——总结



## 2.2 进程的描述——总结





## 第二章 进程的描述与控制

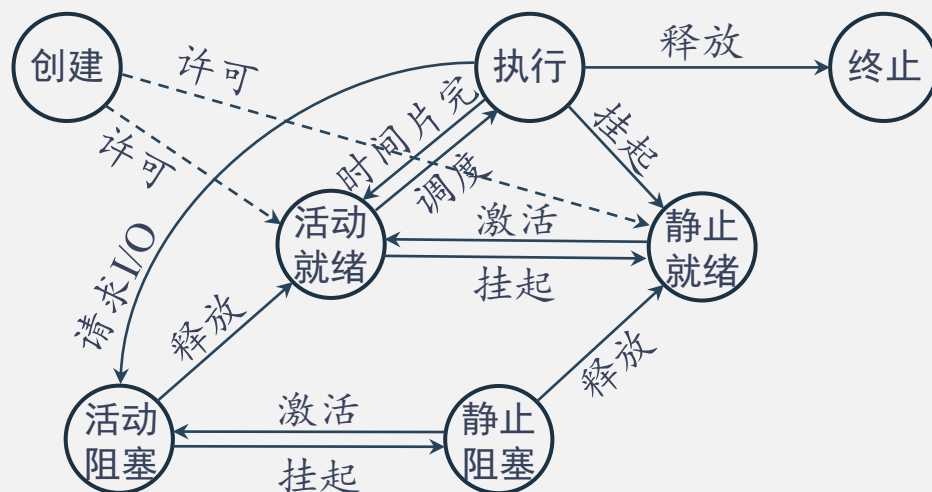
### 2.3 进程控制

## 本节问题和目标

**问题：**多道程序环境下，  
OS怎样控制进程？

**目标：**

- 了解操作系统内核及功能
- 掌握进程的创建、终止、阻塞与唤醒、挂起与激活的进程控制操作



## 2.3 进程控制

**背景** 多道程序运行环境 & 引入进程控制块PCB、建立进程实体

**目标** 在OS系统中实现进程控制，以建立/实现多道程序并发执行。

进程控制是进程管理中**最基本的功能**，主要包括：

- 创建新进程；
- 终止已完成的进程；
- 将因发生异常情况而无法继续运行的进程置于阻塞状态；
- 负责进程运行中的状态转换等功能。

如当一个正在执行的进程因等待某事件而暂时不能继续执行时，将其转变为阻塞状态，而在该进程所期待的事件出现后，又将该进程转换为就绪状态等。**进程控制一般是由OS的内核中的原语来实现的。**

进程控制是OS的核心功能之一，位于OS的内核中。

### OS内核

通常将一些与硬件紧密相关的模块(如中断处理程序)、各种常用设备的驱动程序以及运行频率较高的模块(如时钟管理、进程调度和许多模块所公用的一些基本操作)，都安排在紧靠硬件的软件层次中，将**它们常驻内存，即通常被称为的OS的内核。**

便于对这些软件进行保护，防止遭受其他应用程序的破坏；

提高OS的运行效率。

## 2.3.1 操作系统内核

### 处理机执行状态

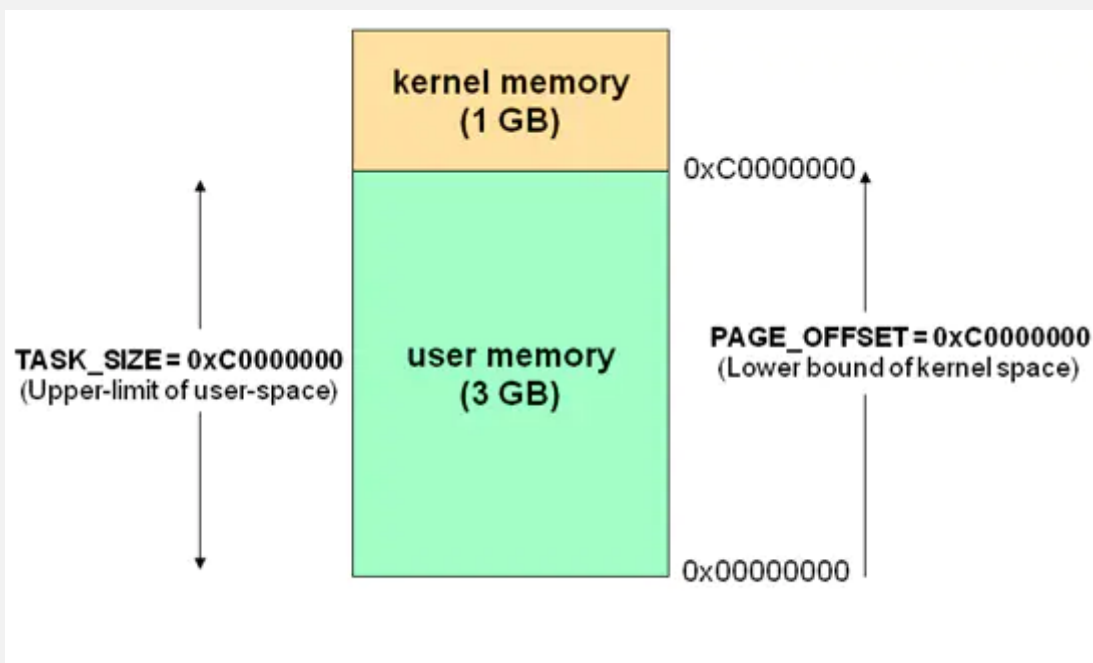
#### 系统/内核态

较高的特权  
能执行一切指令  
访问所有寄存器和存储区

#### 用户态

较低的特权  
执行规定的指令  
访问指定寄存器和存储区

### 内存划分情况



## 2.3.1 操作系统内核

两个例子：  
超级管理员/普通用户  
执法人员/普通民众

### 处理机执行状态

#### 系统/内核态

较高的特权

能执行一切指令

访问所有寄存器和存储区

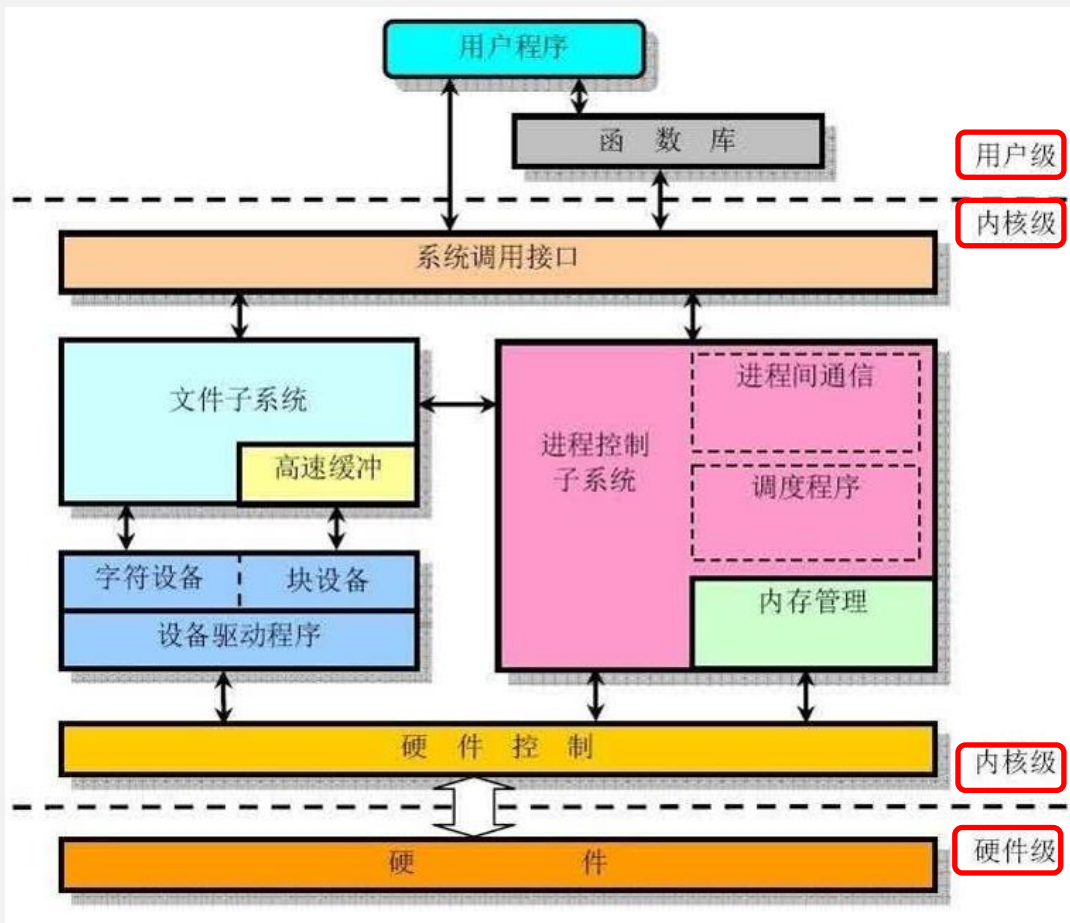
#### 用户态

较低的特权

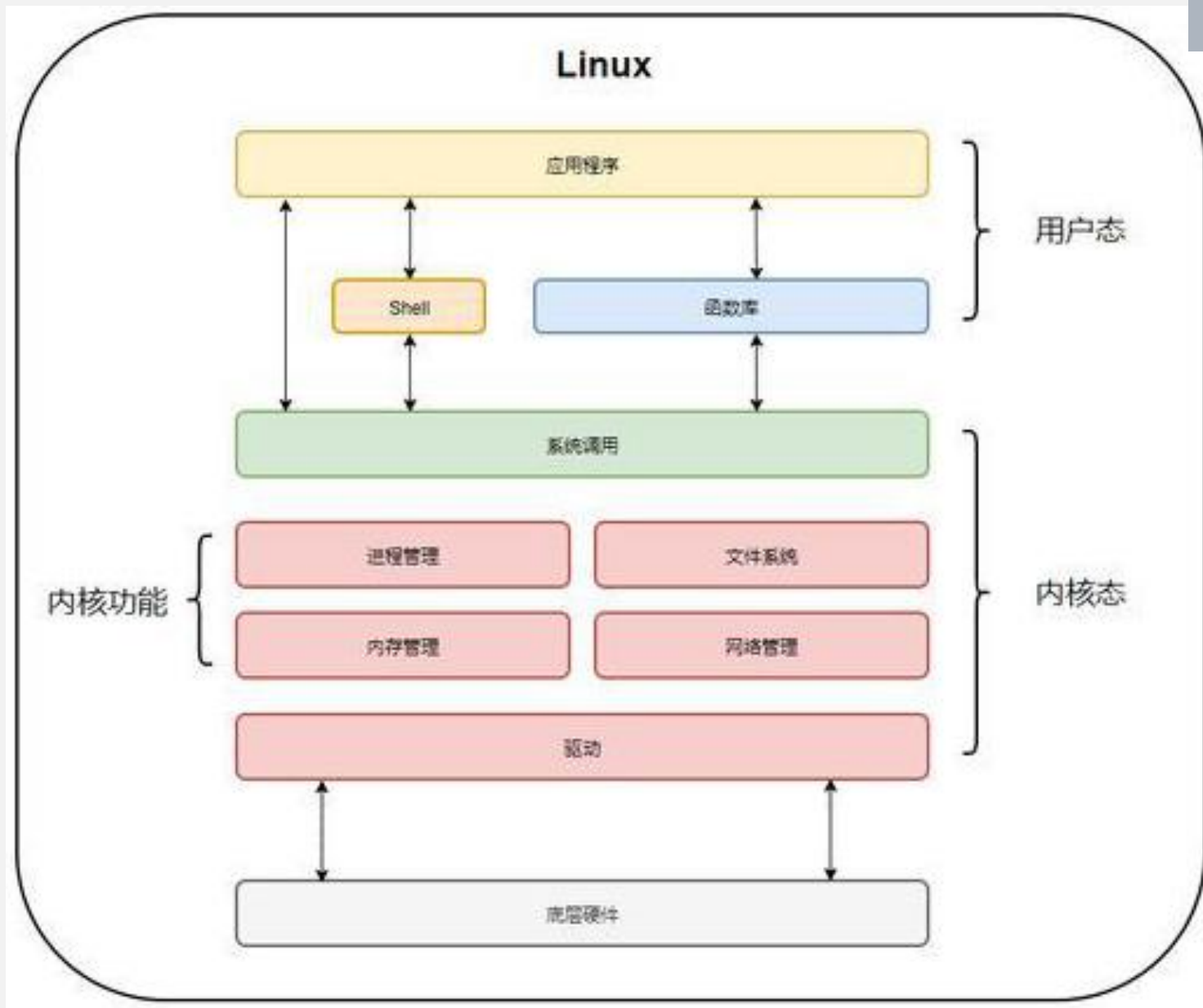
执行规定的指令

访问指定寄存器和存储区

### 系统划分/架构



## 2.3.1 操作系统内核



### 1. 支撑功能 提供给OS其他众多模块所需的基本功能

中断  
处理

内核最基本功能，是整个OS赖以活动的基础；  
如系统调用、键盘命令输入、进程调度、设备驱动等，均依赖中断处理。

时钟  
管理

内核的一项基本功能，OS中许多活动都需要它的支撑；  
如在时间片轮转调度中，每当时间片用完时，便由时钟管理产生一个中断信号，促使调度程序重新进行调度。

原语  
操作

原语(Primitive)：若干条指令组成、用于完成一定功能的一个过程，是一种**原子操作**，一个不可分割的基本单位；  
原语在执行过程不允许被中断，在内核态执行，常驻内存。



## 2.3.1 操作系统内核

### 2. 资源管理功能

#### 进程管理

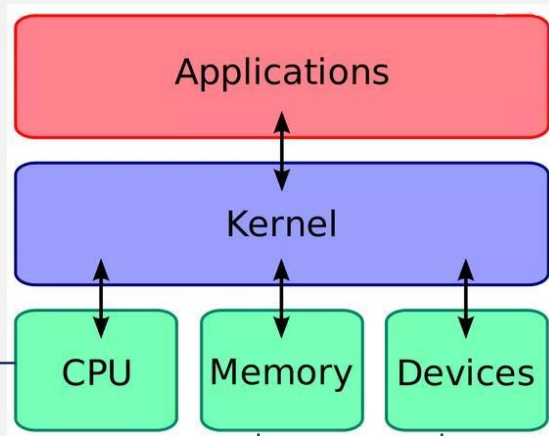
进程控制功能操作频繁，如进程调度与分派、进程创建与撤消、进程同步等；  
放在内核中，以提高OS性能。

#### 存储器管理

内存操作频繁，如逻辑地址变换、内存分配与回收、内存保护等；  
放在内核中，以提高内存操作速度。

#### 设备管理

设备管理与硬件密切相关，包括驱动程序、缓冲管理、设备分配等。



## 2.3.2 进程的创建

### 1. 进程的层次结构

在OS中，**允许一个进程创建另一个进程**，通常把创建进程的进程称为**父进程**，而把被创建的进程称为**子进程**。子进程可继续创建更多的孙进程，由此便形成了一个进程的**层次结构**。

了解这种关系十分重要

- 子进程可以继承父进程所拥有的资源；
- 子进程撤消时，归还父进程的资源；
- 父进程撤消时，同时撤销所有子进程；
- 在PCB中设置家族关系表项；
- 进程不能拒绝其子进程的继承权。

### 2. 进程图

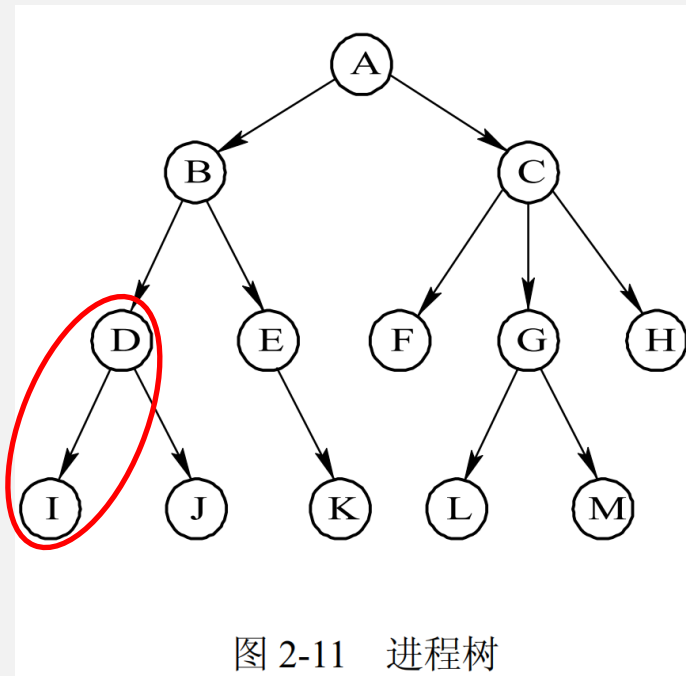
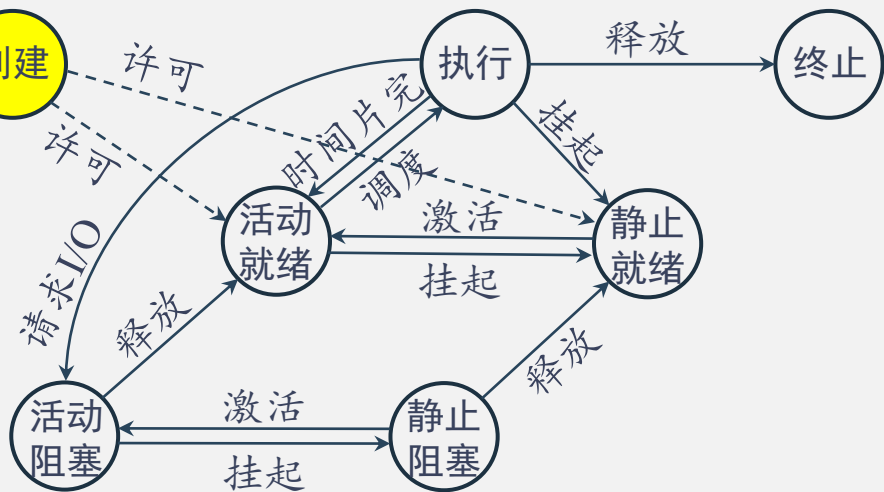


图 2-11 进程树

进程图—形象描述进程家族关系；例如：D创建进程I，D是I的父进程，I是D的子进程；A是根节点，是整个进程家族的祖先。

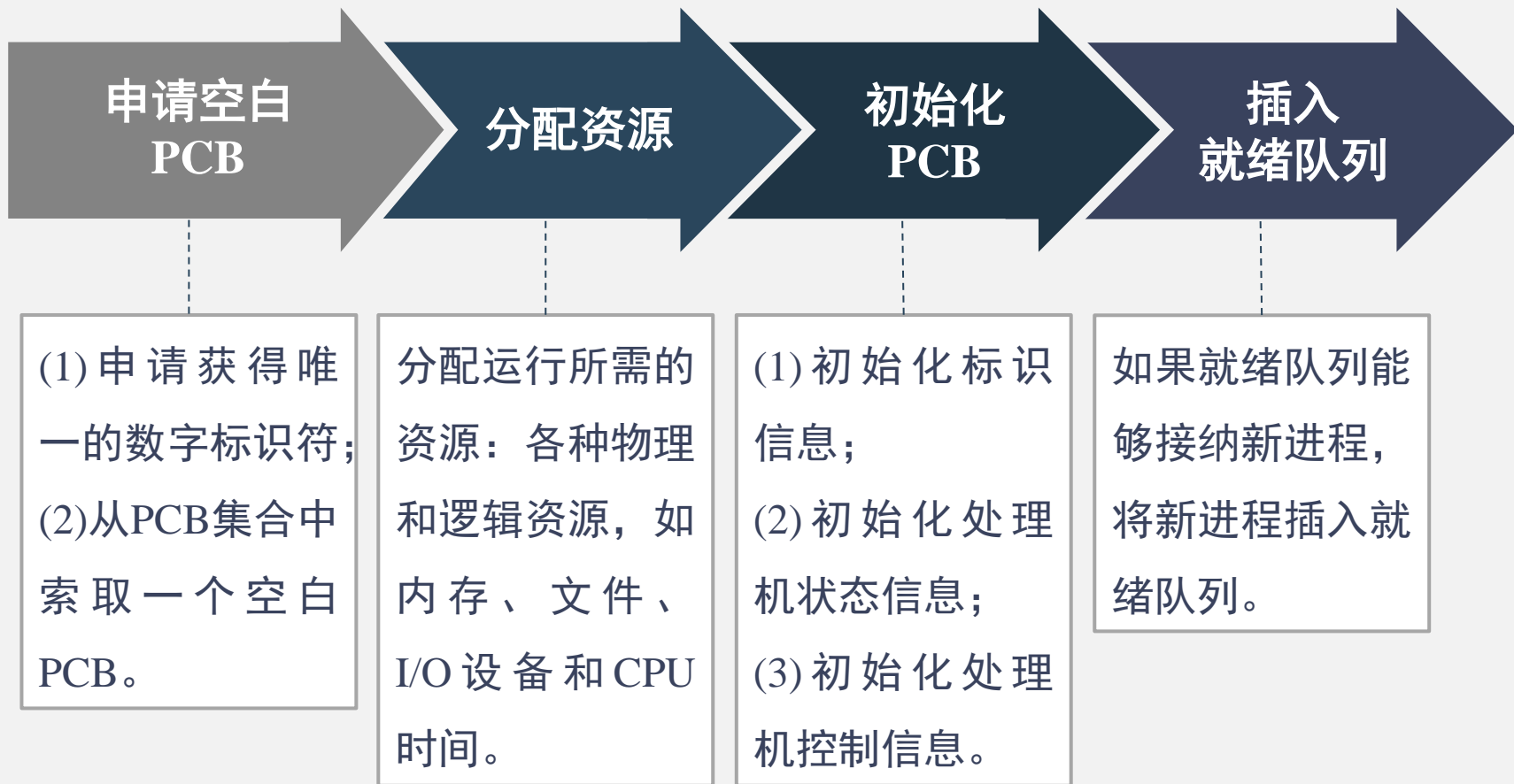
### 3. 引起创建进程的事件



## 2.3.2 进程的创建

### 4. 进程的创建

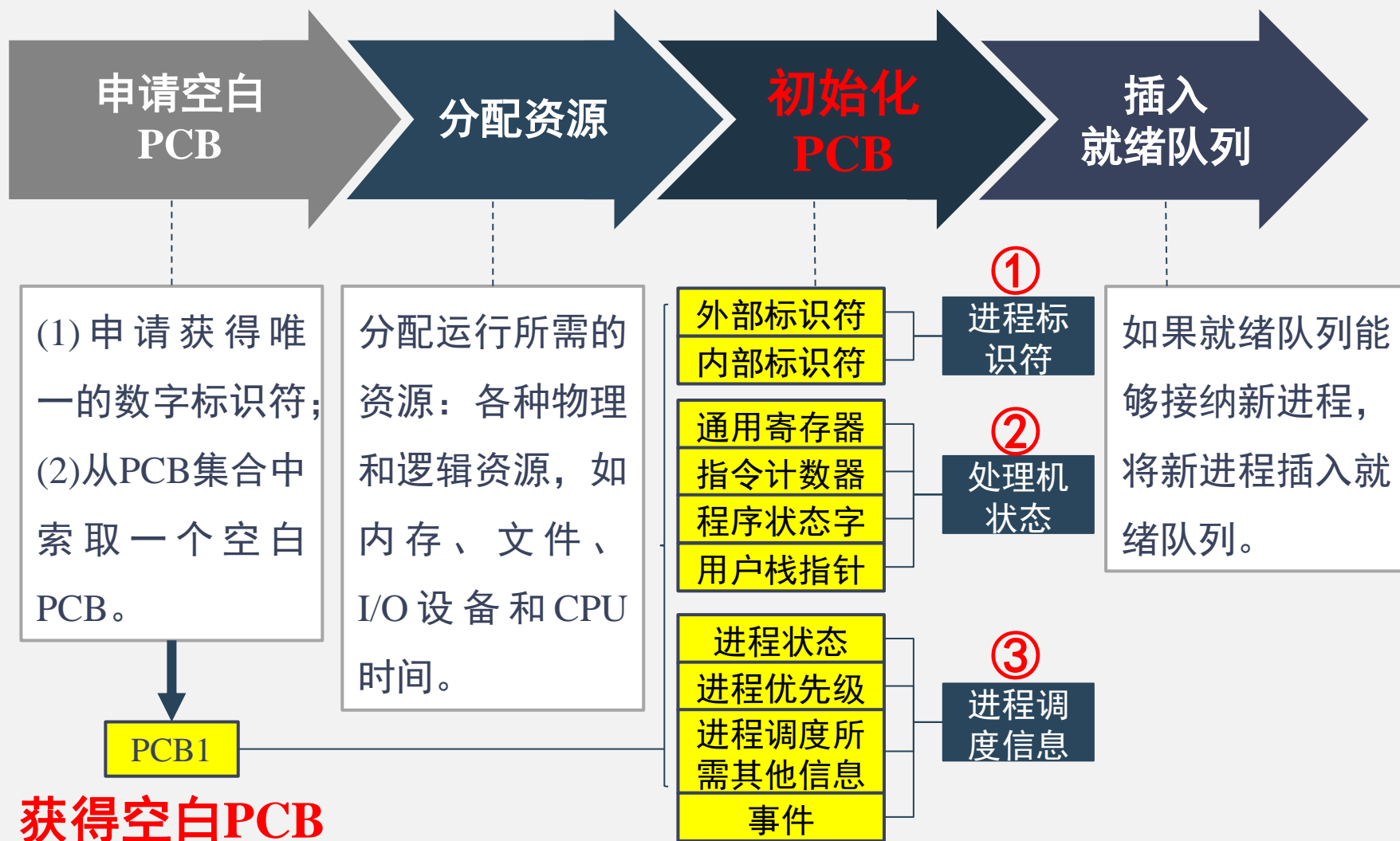
调用创建原语Create



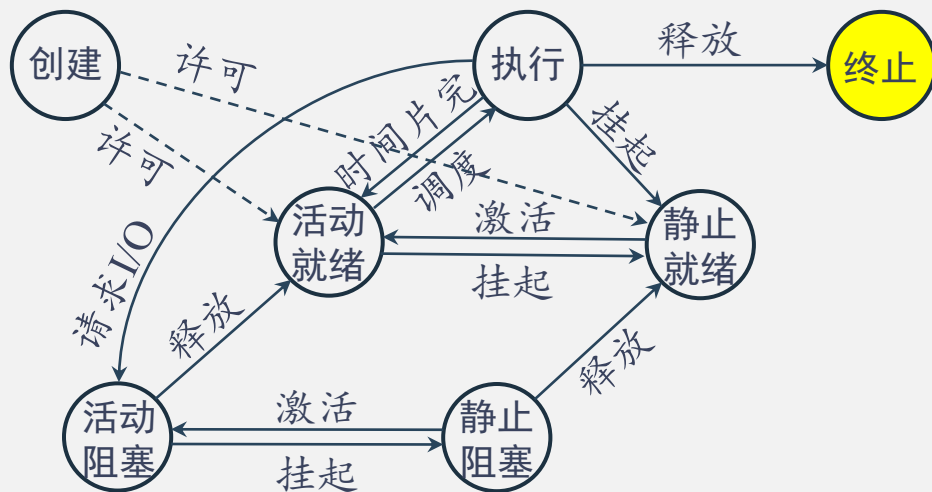
## 2.3.2 进程的创建

### 4. 进程的创建

调用创建原语Create



### 1. 引起进程终止的事件



#### 正常结束

OS

程序任务已完成，准备退出，如Halt或Log off → 中断 → OS；

#### 异常结束

OS

进程运行发生异常事件，无法运行，了解课本中8种事件；

#### 外界干预

外界

进程应外界请求而终止运行，了解课本中3种事件。

### 2. 进程的终止过程

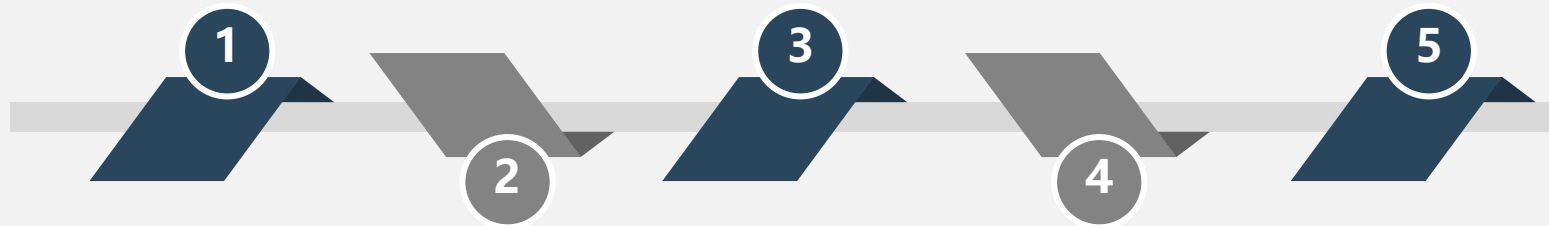
### 调用终止原语

终止该进程的执行；  
设置调度标志为真；

将终止进程的全部  
资源归还给父进程  
或系统；

终止执行

归还资源



找到该进程

根据进程标识符，从PCB集合中找到该进程的PCB，读取该进程的状态；

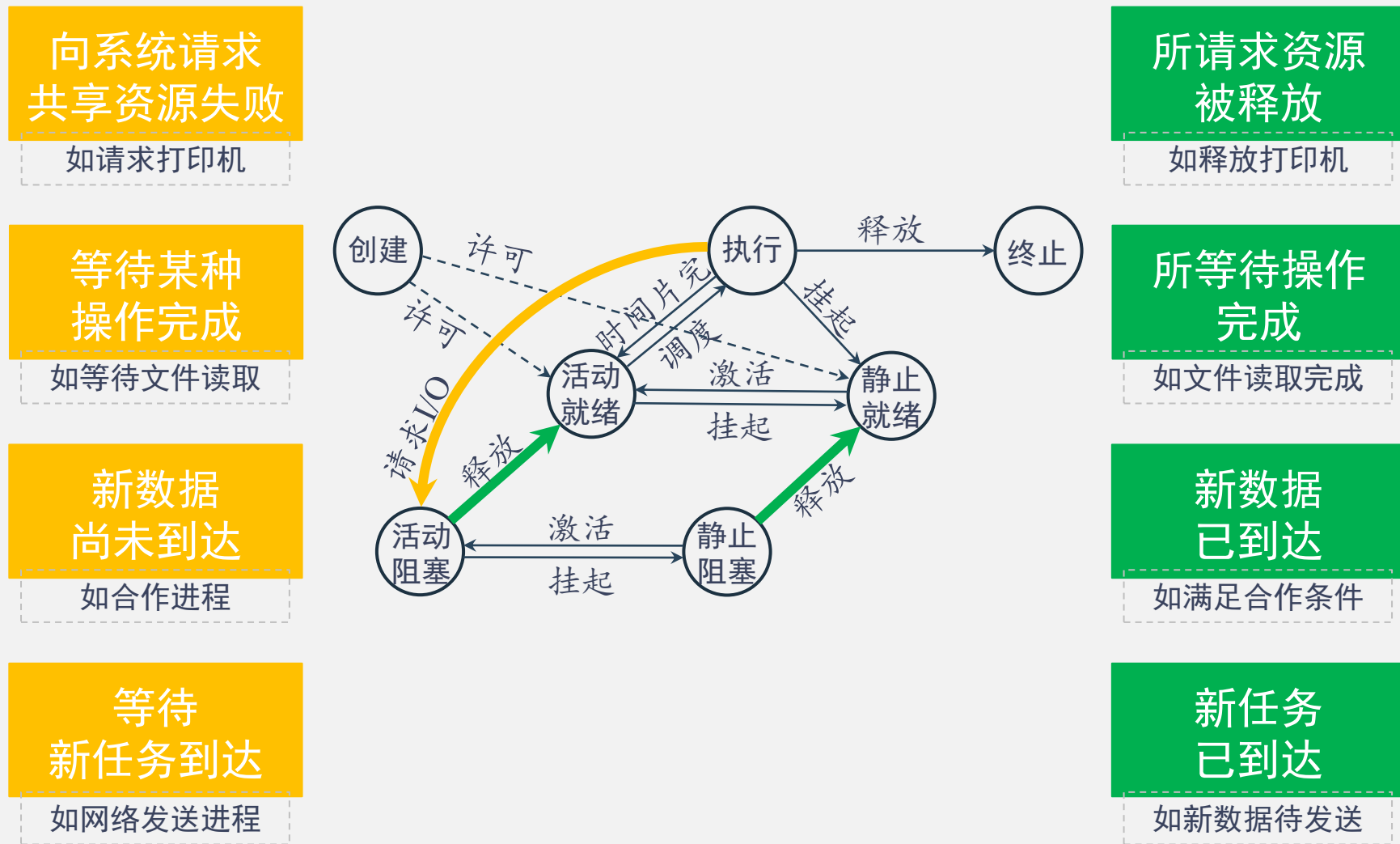
终止子进程

若其存在子进程，则终止其所有子进程的执行；

移出PCB

将被终止进程的PCB从所在队列或链表中移出，等待其他程序来搜集信息。

## 1. 引起进程阻塞和唤醒的事件





## 2.3.2 进程的阻塞与唤醒

### 2. 进行阻塞过程

正在执行的进程，如果发生了上述某事件：

- 进程便通过调用**阻塞原语block**将自己阻塞；

- 可见，阻塞是进程自身的一种**主动行为**

- 进入block过程后，执行状态 → 立即停止执行 → 阻塞状态

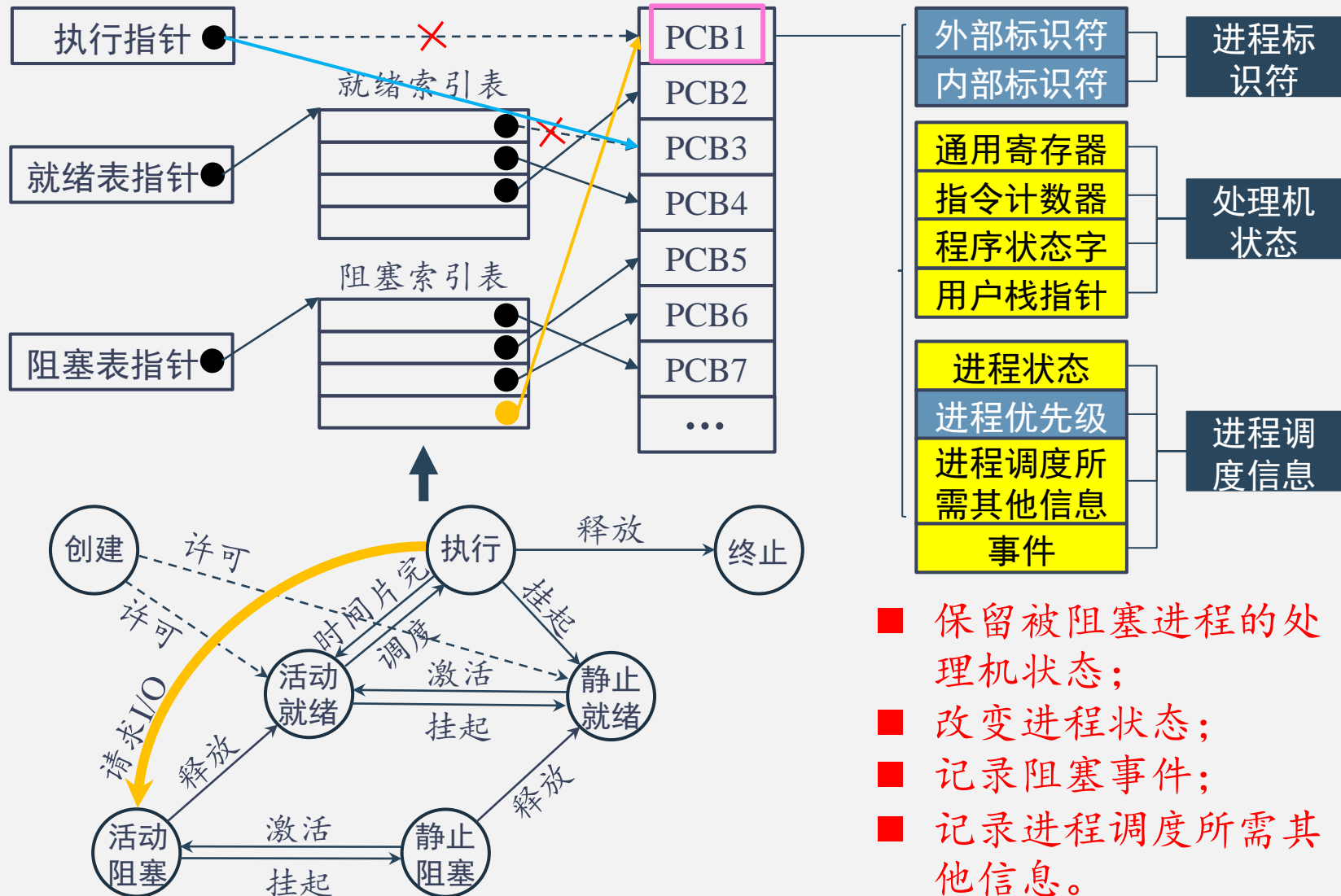
- 并将**PCB插入阻塞队列**，如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞队列

- 转调度程序进行重新调度

- 将处理机分配给另一就绪进程，并进行切换，亦即，**保留被阻塞进程的处理器状态**，按新进程的PCB中的处理器状态设置CPU的环境

## 2.3.2 进程的阻塞与唤醒

### 2. 进行阻塞过程



## 2.3.2 进程的阻塞与唤醒

### 3. 进行唤醒过程

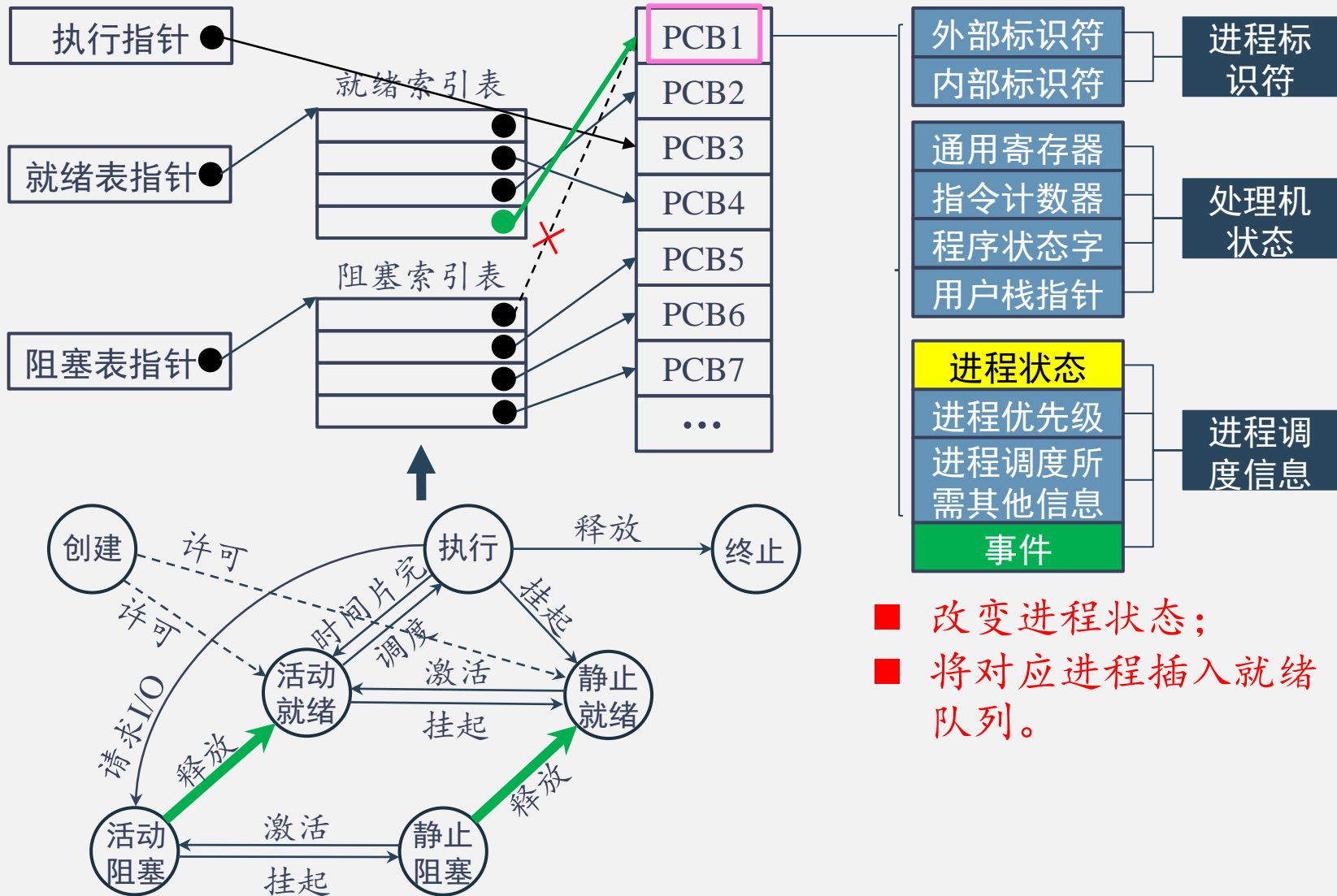
当被阻塞进程所期待的事件发生时：由有关进程(比如提供数据的进程)调用**唤醒原语wakeup**，将等待该事件的进程唤醒。

wakeup执行的过程是：

- 首先把被阻塞的进程**从等待该事件的阻塞队列中移出**，将其PCB中的现行**状态由阻塞改为就绪**；
- 然后再将该PCB**插入到就绪队列中**。

## 2.3.2 进程的阻塞与唤醒

### 3. 进行唤醒过程



## 2.3.2 进程的挂起与激活

### 1. 进程的挂起

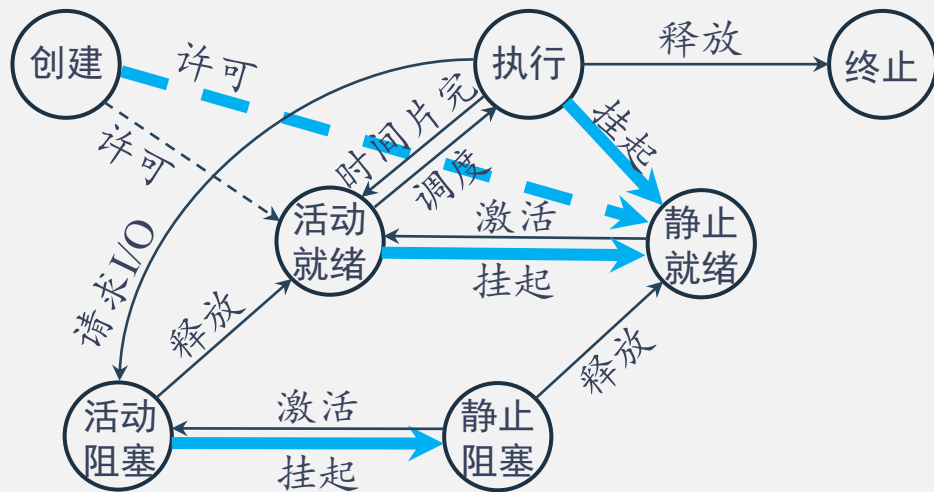
OS将利用**挂起原语suspend**将指定进程或处于阻塞状态的进程挂起。

**引起事件:**系统和用户需求。P42

suspend执行过程:

(1) 首先检查挂起进程的状态:

- 活动就绪 → 静止就绪;
  - 活动阻塞 { → 静止阻塞;
  - 执 行 { → 静止就绪;
- 重新调度。



### 2. 进程的激活过程

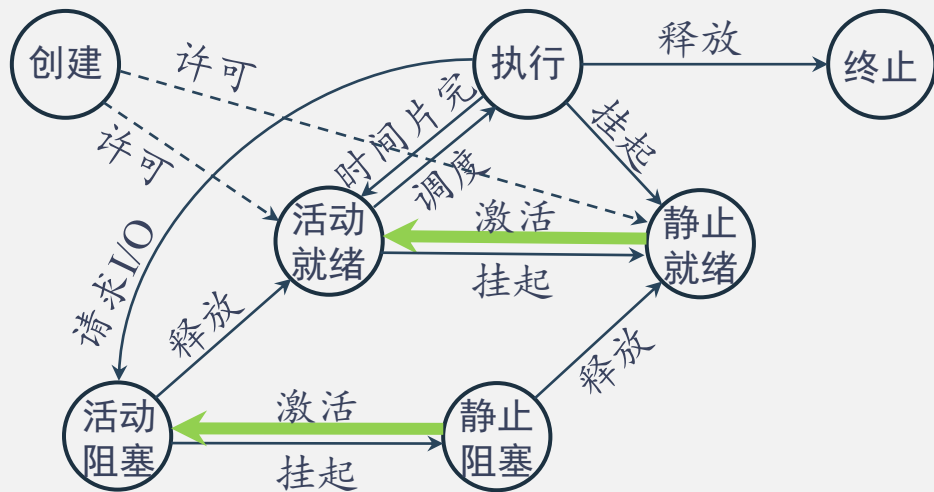
OS将利用**激活原语active**将指定进程激活。

active执行过程：

(1) 首先将进程从外存调入内存，  
检查进程的现行状态：

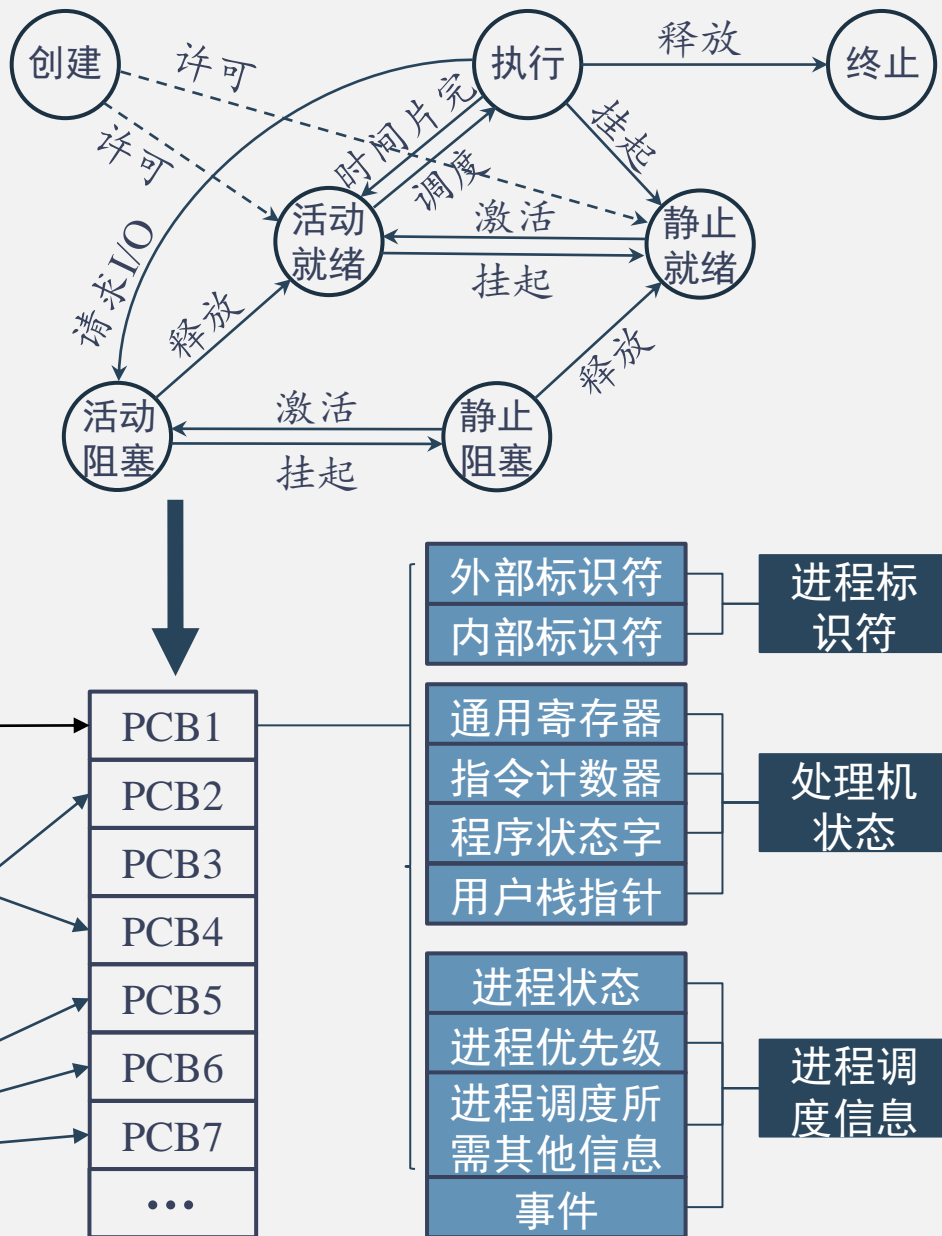
- 静止就绪 → 活动就绪；
- 静止阻塞 → 活动阻塞；

如果系统采用的是**抢占调度策略(第三章)**，每当发生 **静止就绪 → 活动就绪** 状态转变时，则**检查是否需要进行调度，根据优先级进行比较**。



## 2.3 进程控制——总结

- 进程控制实现进程并发执行的全过程管理，主要体现在进程状态的转换；
- 进程控制依赖于PCB来实现，PCB对于进程控制至关重要；
- 进程管理模块由系统内核完成，常驻内存。





人人词典

1

前趋图和程序执行

2

进程的描述

3

进程控制

4

进程同步

5

经典进程的同步问题

6

进程通信

7

线程及线程实现

Good morning, and in case I don't see you, good afternoon, good evening, and good night!

早上好！假如再也见不到你，就再祝你下午好，晚上好，晚安！

——《楚门的世界》

IMDb评

23

国际奥林匹克日

Jun | Wednesday

农历五月十四



## 第二章 进程的描述与控制

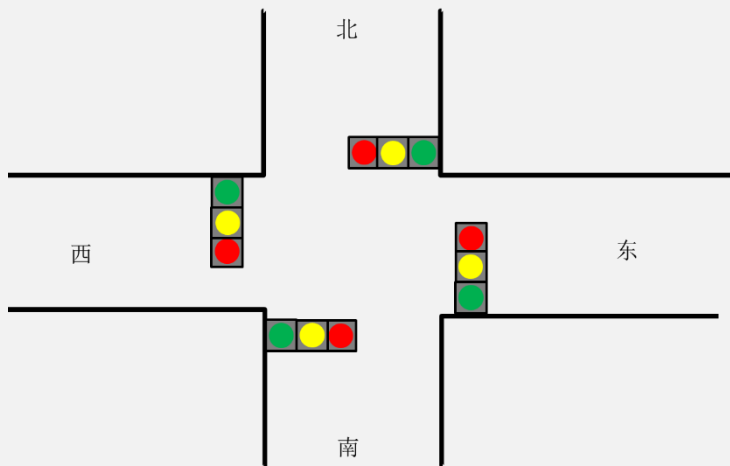
### 2.4 进程同步

**问题：**如何解决并发程序的不可再现性问题？

**目标：**

- 理解进程同步的概念
- 掌握OS实现进程同步的机制
- 重点掌握信号量机制及应用

## 2.4 进程同步

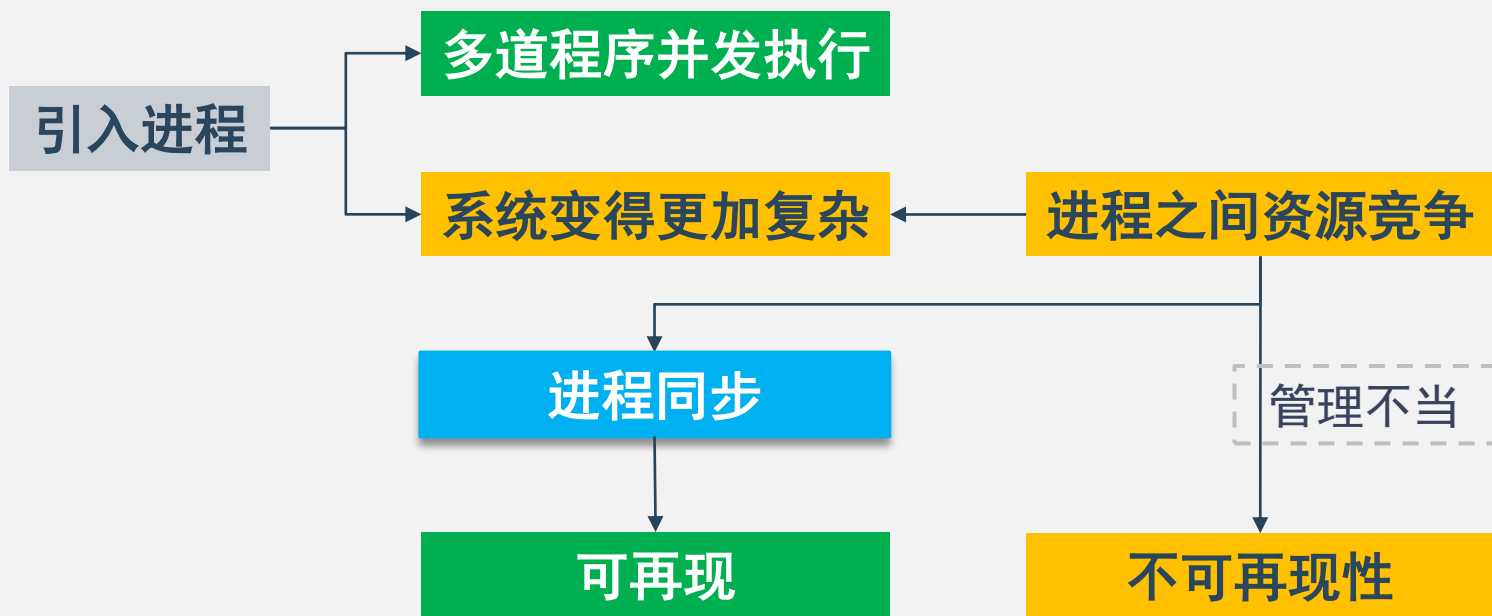


十字路口，单行车道，  
需要用红绿灯指示车辆行进顺序



共享会议室  
需要用指示牌指示会议室是否被  
占用

## 2.4 进程同步



通过引入【**进程同步**】机制，对多个并发进程的资源竞争和分配进行妥善的管理，以保证程序并发执行的可再现性。

## 2.4.1 进程同步的基本概念

问题对象

着手点

进程同步机制的**主要任务**：对多个**相关进程**在**执行次序**上进行协调，

使并发执行的诸进程之间能按照**一定的规则**(或时序)**共享系统资源**，并

能很好地合作，从而使程序的执行具有**可再现性**。

进程同步要制定的

最终目的

核心问题

## 2.4.1 进程同步的基本概念

### 1. 两种形式的制约关系

#### 间接相互制约



进程之间的间接制约关系源于对共享资源的互斥访问。

例如：进程A和进程B都需要使用打印机进行打印输出；

这类资源由OS统一分配，先申请，再使用，不允许直接使用。

#### 直接相互制约



进程之间的直接制约关系源于它们之间的相互合作。

例如：输入进程A向缓冲区N存数据，计算进程B从缓冲区N读数据计算；

- 缓冲区为空时，进程B阻塞，等待数据；
- 进程A输入数据进缓冲区后，唤醒进程B；
- 缓冲区满时，进程A不能再存入数据，被阻塞；
- 进程B取出数据后，唤醒进程A。

## 2.4.1 进程同步的基本概念

### 2. 临界资源(Critical Resource)

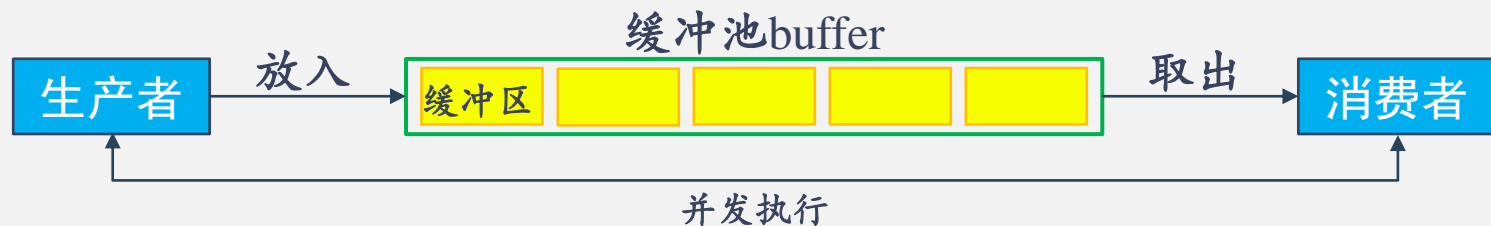
多道程序系统中存在许多进程，它们共享各种资源，然而有很多资源一次只能供一个进程使用。**一次仅允许一个进程使用的资源称为临界资源。**许多物理设备都属于临界资源，如输入机、打印机、磁带机等。

对临界资源的访问，诸进程间应采取**互斥方式**，实现对这种资源的共享。

## 2.4.1 进程同步的基本概念

### 2. 临界资源

### 生产者-消费者问题



- 生产者生产(进程)和消费者消费(进程)是异步运行的（异步性）；
- 它们必须保持**同步**：
  - 消费者不能去空缓冲区中取出商品；
  - 生产者不能往满缓冲区中放入商品；

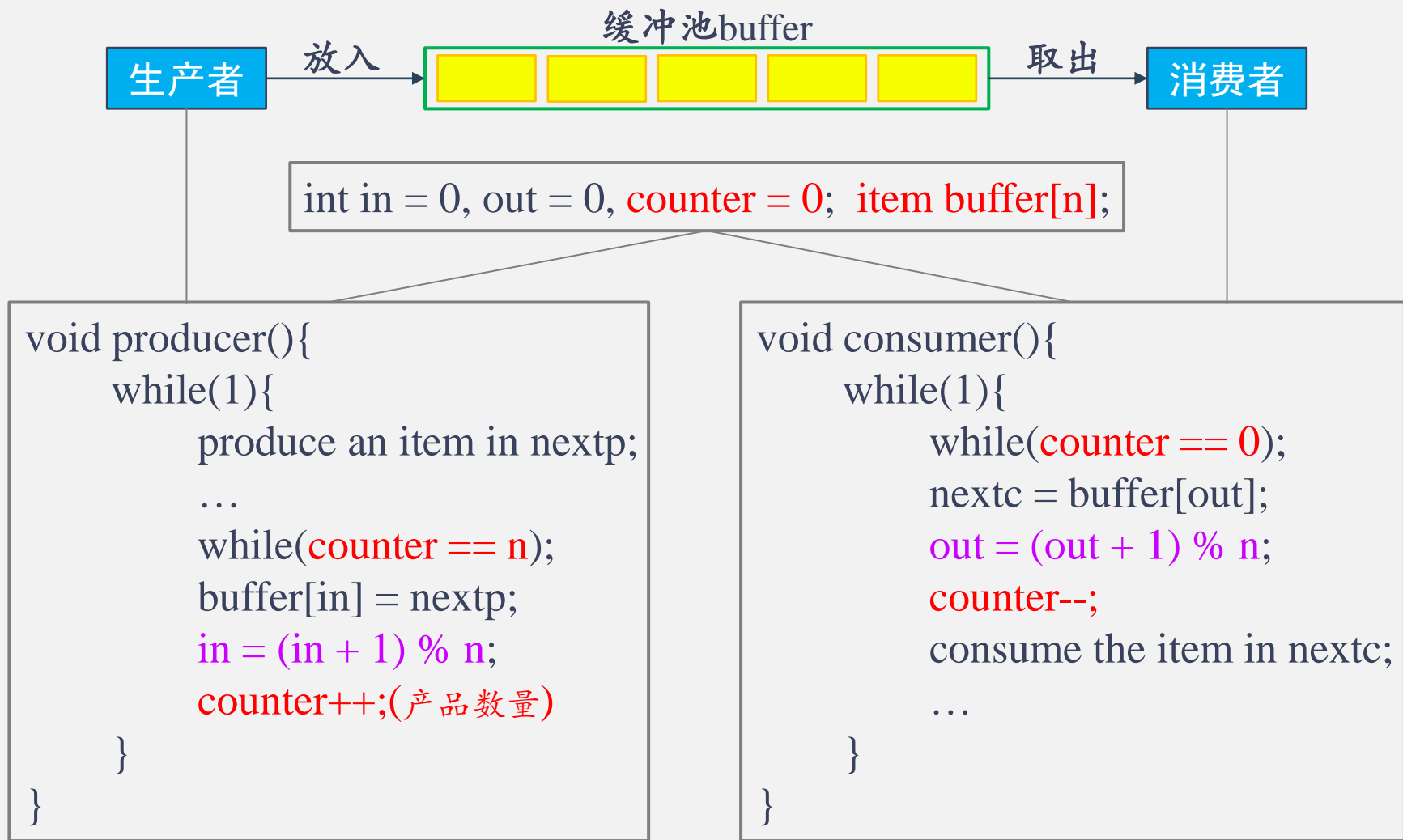
根据以上条件设计程序来解决该问题。



## 2.4.1 进程同步的基本概念

### 2. 临界资源

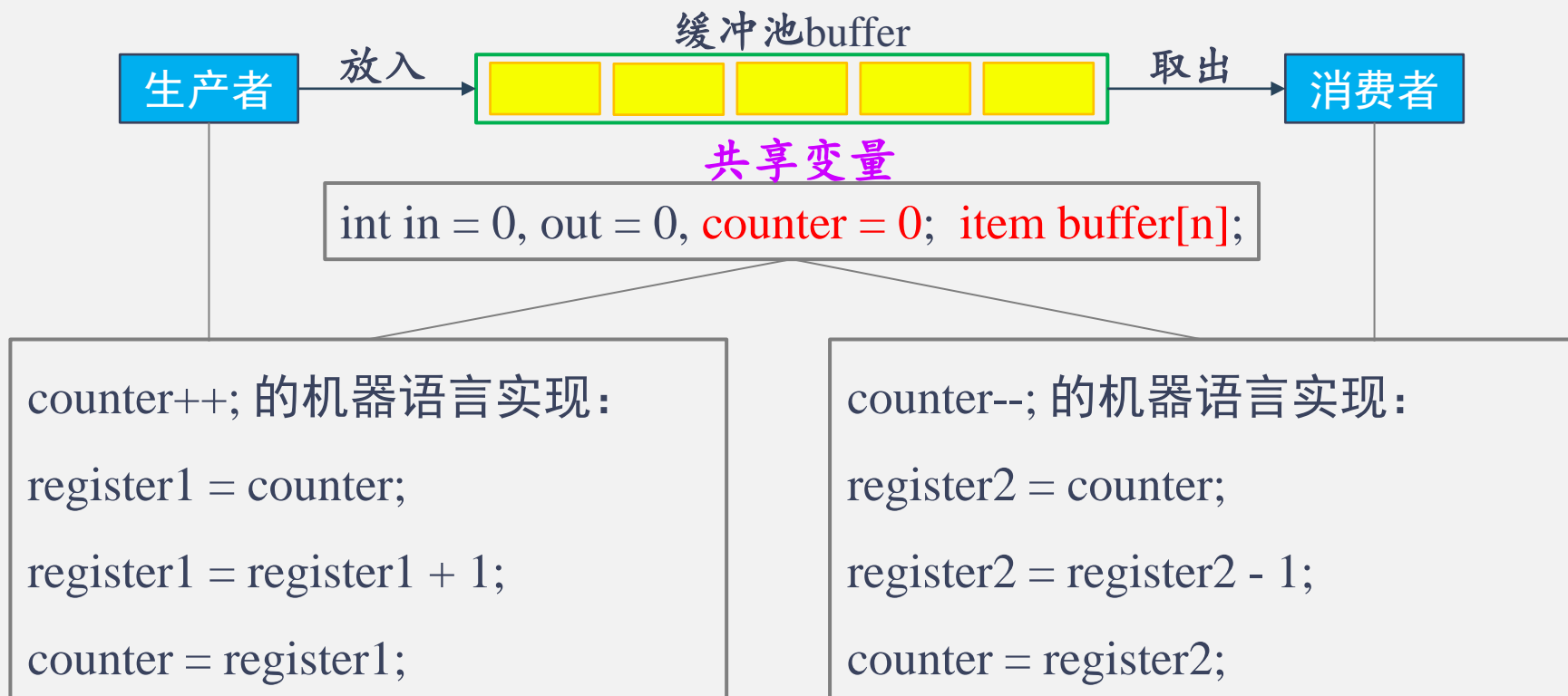
### 生产者-消费者问题



## 2.4.1 进程同步的基本概念

### 2. 临界资源

### 生产者-消费者问题



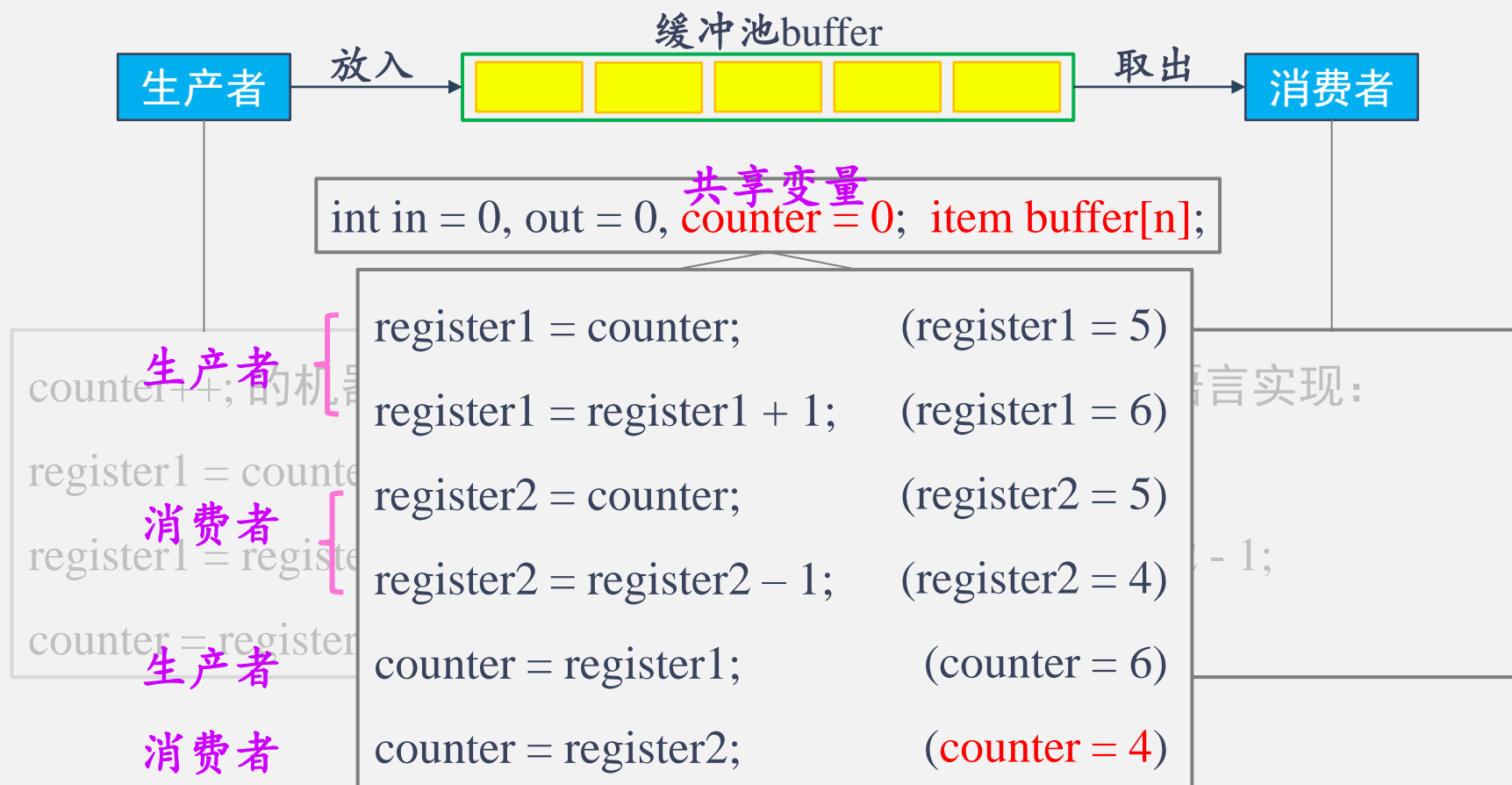
假设：counter当前值是5；正常情况下：

- 先执行生产者的机器语句，再执行消费者的机器语句，最后counter = 5；
- 先执行消费者的机器语句，再执行生产者的机器语句，最后counter = 5；

## 2.4.1 进程同步的基本概念

### 2. 临界资源

### 生产者-消费者问题



程序的执行已经失去了再现性，解决此问题的关键在于：把`counter`作为临界资源处理，即让生产者进程和消费者进程互斥地访问变量`counter`。

## 2.4.1 进程同步的基本概念

### 3. 临界区

### 解决进程同步问题的策略

不论硬件临界资源还是软件临界资源，多个进程必须互斥地进行访问。

进入区

检查临界资源是否正被访问。

临界区

把在每个进程中访问临界资源的那段代码称为临界区。

退出区

释放临界资源：  
被访问标志→未被访问状态。

剩余区

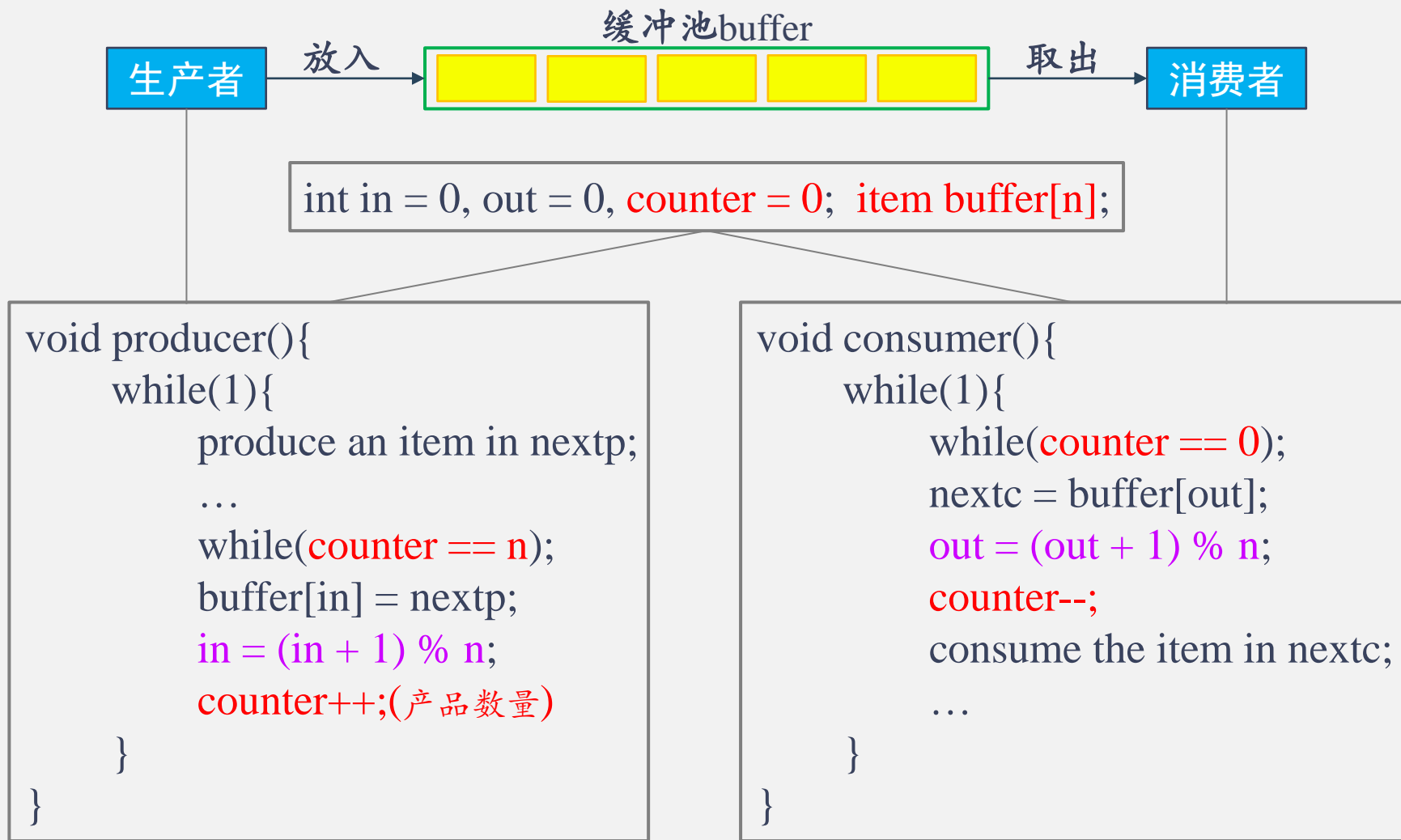
与临界资源访问无关。

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

## 2.4.1 进程同步的基本概念

### 2. 临界资源

### 生产者-消费者问题



## 2.4.1 进程同步的基本概念

### 4. 同步机制应遵循的准则

为实现进程互斥地进入自己的临界区，可用软件方法，更多的是在系统设置专门的同步机构来协调各进程间的运行。所有同步机制都应该遵循下面四条准则：

#### 空闲让进

无进程处于临界区，表明临界资源空闲，允许一个请求进入临界区的进程进入自己的临界区。

#### 忙则等待

已有进程进入临界区，表明临界资源正在被访问，其他试图进入临界区的进程必须等待。

#### 有限等待

对要求访问临界区的资源，应保证在有限时间内能进入自己的临界区，避免陷入“死等”状态。

#### 让权等待

当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”状态。

目前许多计算机已提供一些特殊硬件指令，允许对一个字的内容进行检查和修正，或者对两个字的内容进行交换。因此，可采用硬件指令来解决临界区问题解决诸进程互斥进入临界区的问题。

实际上，在对临界区进行管理时，可以将标志看做一个锁，“锁开”进入，“锁关”等待，初始时锁是打开的。每个要进入临界区的进程必须先对锁进行测试，当锁未开时，则必须等待，直至锁被打开。反之，当锁是打开的，则应立即把其锁上，以阻止其他进程进入临界区。

为防止多个进程同时测试到锁为打开的情况，测试和关锁操作必须是连续的，不允许分开进行。

## 2.4.2 硬件同步机制

### 1. 关中断

关中断是实现互斥的最简单的方法之一。

- 在进入锁测试之前关闭中断，直到完成锁测试并上锁之后才能打开中断。

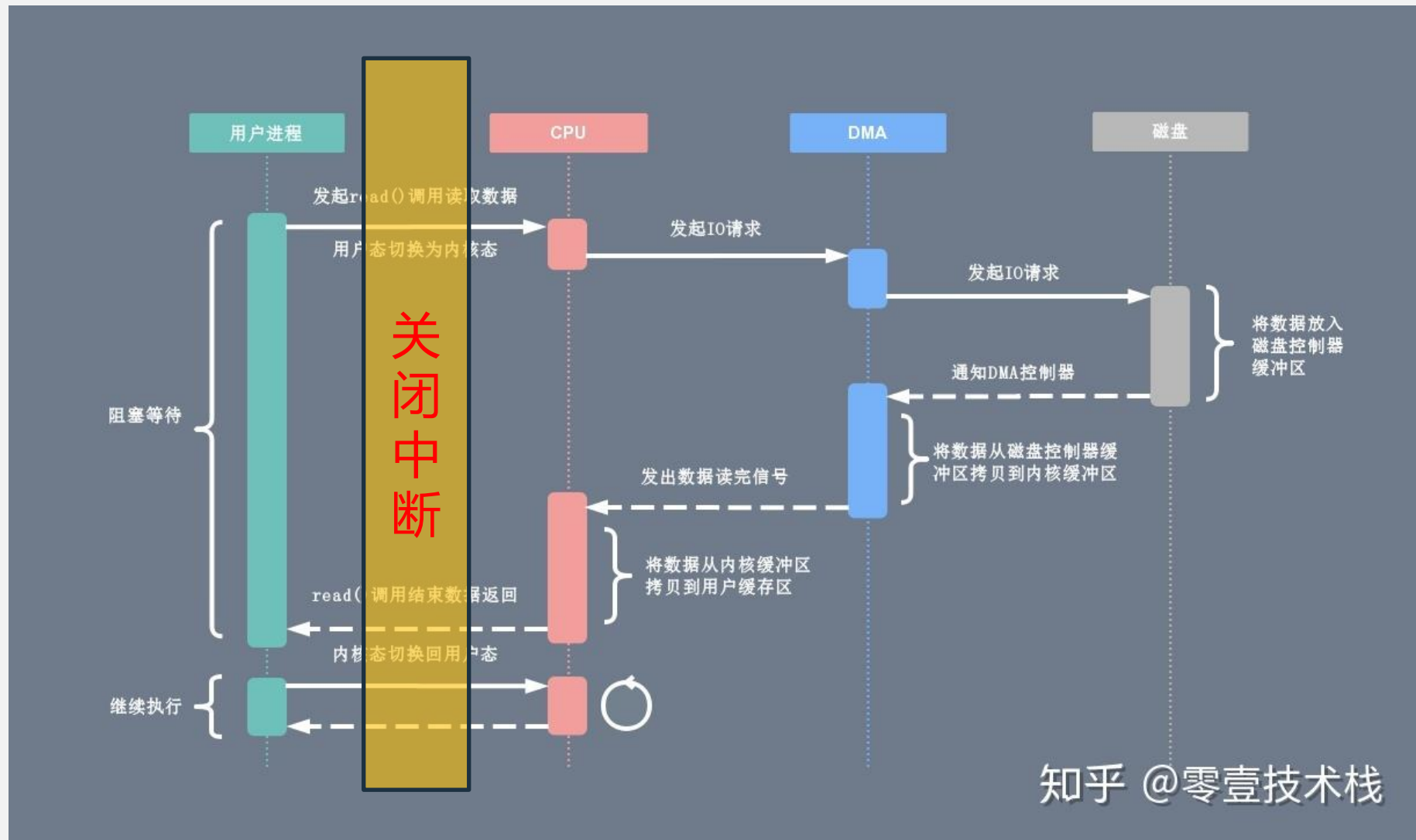
这样，进程在临界区执行期间，计算机系统不响应中断，从而不会引发调度，也就不会发生进程或线程切换。由此，保证了对锁的测试和关锁操作的连续性和完整性，有效地保证了互斥。



## 2.4.2 硬件同步机制

### 1. 关中断

### 案例



### 1. 关中断

关中断的方法存在许多缺点：

- ① 滥用关中断权力可能导致严重后果；
- ② 关中断时间过长，会影响系统效率，限制了处理器交叉执行程序的能力；
- ③ 关中断方法也不适用于多CPU 系统，因为在一个处理器上关中断并不能防止进程在其它处理器上执行相同的临界段代码。

## 2.4.2 硬件同步机制

### 2. 利用Test-and-Set指令实现互斥

这是一种**借助一条硬件指令**——“测试并建立”指令TS(Test-and-Set)以实现互斥的方法。在许多计算机中都提供了这种指令。 **(原语)**

- 为**每个临界资源**设置一个**布尔变量lock**，表示资源状态，可看成一把锁，**初值为FALSE**，表资源空闲。

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

**使用简单快速的硬件指令来实现状态控制**

```
do{  
    ...  
    while TS(&lock);  
    critical section;  
    lock = FALSE;  
    remainder section;  
}while(TRUE);
```

## 2.4.2 硬件同步机制

### 3. 利用Swap指令实现进程互斥

该指令称为**对换指令**，用于交换两个字的内容，可以简单有效地实现互斥，方法是**为每个临界资源设置一个全局布尔变量lock**，初值为FALSE，在**每个进程中再利用一个局部布尔变量key**。

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

使用简单快速的硬件指令来实现状态控制

```
do{  
    key = TRUE;  
    do{  
        swap(&lock, &key);  
    }while(key != FALSE);  
    critical section;  
    lock = FALSE;  
    ...  
}while(TRUE);
```

### 1. 关中断

```
do{  
    ...  
}while(TRUE);
```

### 2. 利用Test-and-Set指令实现互斥

### 3. 利用Swap指令实现进程互斥

#### 缺点：

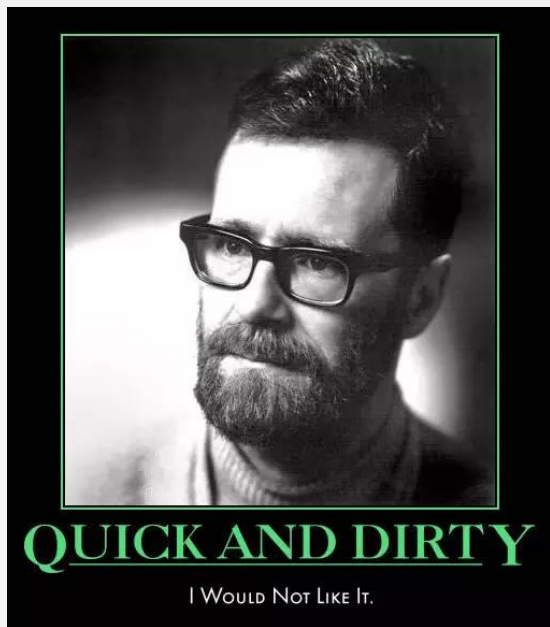
当临界资源忙碌时，其他进程必须不断地进行测试，处于一种“忙等”状态，不符合“让权等待”的原则，造成处理机时间的浪费，同时也很难将它们用于解决复杂的进程同步问题。

1965年，荷兰学者Dijkstra提出信号量(Semaphores)机制，是一种卓有成效的进程同步工具。



信号量机制已经被广泛应用于单处理机和多处理机系统以及计算机网络中。

# 信号量之父—Dijkstra



艾兹格·迪克斯特拉  
(Edsger Wybe  
Dijkstra)

1930.5.11~2002.8.6

一生致力于把程序设计发展成一门科学

- 提出“GOTO”有害论和结构化程序设计；
- 提出信号量和PV原语；
- 解决了“哲学家聚餐”问题；
- Dijkstra最短路径算法和银行家算法的创造者
- 第一个ALGOL 60编译器的设计者和实现者；

专注之痴：  
“艺痴者技必良”、“术业有专攻”  
专注并做到极致，往往能取得更大成就。

# 专注造就行业“领头羊”

## 大疆专注无人机



2006年创立  
2012年推出第一台无人机  
16年专注无人机  
世界无人机领域的王者

## 华为专注通信技术



1987年成立  
1989年开始自主研发交换机  
35年专注ICT领域  
2018年世界首次突破5G技术  
成为世界第一的通讯公司

## 乔布斯专注手机



2007年iPhone横空出世  
2011年发布iPhone 4  
专注“高贵而尖端”设计  
引领了现代智能手机发展



## 2.4.3 信号量机制

### 1. 整型信号量

P/V操作 荷兰文

Passeren—通过

Vrijgeven—释放

整型信号量

定义为一个用于表示资源数目的整型量 $S$ ，与一般整型量不同，除初始化外，仅能通过两个原子操作 $\text{wait}(S)$ 和 $\text{signal}(S)$ 来访问。

```
wait(S){
```

```
    while(S <= 0);
```

```
    S--;
```

```
}
```

“忙等”，不符合“让权等待”

原子操作

执行时是不可中断的

```
signal(S){
```

```
    S++;
```

```
}
```

当一个进程在修改某信号量时，没有其他进程可同时对该信号量进行修改。此外，在 $\text{wait}$ 操作中，对 $S$ 值测试和做 $S=S-1$ 操作时都不可中断。

## 2.4.3 信号量机制

### 2. 记录型信号量

背景

考虑采取“让权等待”策略，同时会出现多个进程等待访问同一临界资源的情况

#### 记录型信号量

采用记录型的数据结构而得名，除了一个用于代表资源数目的整型变量 **value**，还增加一个进程链表指针 **list**，链接上述所有等待进程。

#### P操作

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

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

**S->value**的初值表示系统中某类资源的数目，因而又称为资源信号量。

#### V操作

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

### 3. AND型信号量

**背景** 前面所述机制针对的是多个并发进程仅共享一个临界资源的情况。

↓  
**案例** 进程A和进程B，都要求访问共享数据D和E；因此，设置了两个互斥信号量Dmutex和Emutex对两个数据进行控制。

扫描仪 打印机

```
processA:
wait(Dmutex);
wait(Emutex);
```

```
processB:
wait(Emutex);
wait(Dmutex);
```

```
processA: wait(Dmutex); // 于是Dmutex = 0
processB: wait(Emutex); // 于是Emutex = 0
processA: wait(Emutex); // 于是Emutex = -1, A阻塞
processB: wait(Dmutex); // 于是Dmutex = -1, B阻塞
阻塞即暂停，并不释放已有资源。
```

最后，A和B进入僵持状态，两者都无法从僵持状态中解决，即进程A和B陷入死锁状态(第3章)。当进程同时要求的共享资源越多，进程发生死锁的可能性也越大。

## 2.4.3 信号量机制 3. AND型信号量

### AND同步机制基本思想

将进程在整个运行过程中所需的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。只要尚有一个资源未能分配给进程，其他所有可能为之分配的资源也不分配给它。(原子操作)

```
Swait(S1, S2, ..., Sn){  
    while(TRUE){  
        if(Si>=1 && ... && Sn>=1){  
            for(i=1;i<=n;i++) Si--;  
            break;  
        }else{  
            将进程放入Si(第一个Si<1)阻塞  
            队列，并释放所有资源  
        }  
    }  
}
```

```
Ssignal(S1, S2, ..., Sn){  
    while(TRUE){  
        for(i=1;i<=n;i++) {  
            Si++;  
            将所有等待Si的进程加入就绪队列  
        }  
    }  
}
```

## 2.4.3 信号量机制

### 4. 信号量集

#### 背景

前面所述信号量机制中，wait(S)或signal(S)操作仅能对信号量加1或减1操作，也就是每次只能对某类临界资源的一个单位进行申请或释放。

#### 问题

当一次需要N个相同资源时，就需要进行N次wait(S)操作，效率较低，甚至会增加死锁的概率。

#### “信号量集”机制

当进程申请某类临界资源时，在每次分配之前，都必须测试资源的数量，判断是否大于可分配的下限值，决定是否予以分配。

通过对AND信号量机制加以扩充，对进程所申请的所有资源以及每类资源不同的需求量，在一次P、V原语操作中完成申请或释放。

## 2.4.3 信号量机制

### 4. 信号量集

- 进程对信号量 $S_i$ 的测试不再是1，而是该资源的分配下限值 $t_i$ ，即要求 $S_i \geq t_i$ ，否则不予分配。
- 一旦允许分配，进程对该资源的需求值为 $d_i$ ，则表示资源占用量，进行 $S_i = S_i - d_i$ 操作，而不是 $S_i = S_i - 1$ 。

$\text{Swait}(S1, t1, d1, \dots, Sn, tn, dn);$

$\text{Ssignal}(S1, d1, \dots, Sn, dn);$

一般“信号量集”还有以下几种特殊情况：

- (1)  $\text{Swait}(S, d, d)$ ：只有一个信号量 $S$ ，允许每次申请 $d$ 个，现有资源 $< d$ 不予分配；
- (2)  $\text{Swait}(S, 1, 1)$ ：蜕化为一般的记录信号量( $S > 1$ )或互斥信号量( $S = 1$ )；
- (3)  $\text{Swait}(S, 1, 0)$ ：特殊且很有用的信号操作， $S \geq 1$ 时，允许多个进程进入某特定区； $S$ 为0后，将阻止任何进程进入特定区。换言之，它相当于一个开关。

## 2.4.4 信号量的作用

### 1. 利用信号量实现进程互斥

为使多个进程能互斥地访问某临界资源，只需为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问该资源的临界区CS置于wait(mutex)和signal(mutex)操作之间即可。

设mutex为互斥信号量，其初值为1，取值范围为(-1, 0, 1)：

- mutex = 1时，两个进程都未进入需要互斥的临界区；
- mutex = 0时，有一个进程进入临界区运行，另一个必须等待，挂入阻塞队列；
- mutex = -1时，表示由一个进程正在临界区运行，另外一个进程因等待而阻塞在信号量队列中，需要被当前已在临界区运行的进程退出时唤醒。

## 2.4.4 信号量的作用

### 1. 利用信号量实现进程互斥

```
semaphore mutex = 1;
```

<pre>P<sub>A</sub>{     while(1){         wait(mutex);         临界区;         signal(mutex);         剩余区;     } }</pre>	<pre>P<sub>B</sub>{     while(1){         wait(mutex);         临界区;         signal(mutex);         剩余区;     } }</pre>
---	---

**wait(mutex)和signal(mutex)必须成对地出现：**

- 缺少wait(mutex)会导致系统混乱，不能保证对临界资源的互斥访问；
- 缺少signal(mutex)将会使临界资源永不被释放，阻塞进程无法唤醒。



## 2.4.4 信号量的作用

### 2. 利用信号量实现前趋关系

两个并发进程P1和P2，P1中有语句S1，P2中有语句S2,。希望在S1执行后S2执行。为实现这种前趋关系，只需使进程P1和P2共享一个公用信号量S，其初值为0。

将signal(S)操作放在语句S1后面，而在S2语句前面插入wait(S)操作：

- 在进程P1中，用S1； signal(S)；
- 在进程P2中，用wait(S)； S2；

由于S被初始化为0，这样，若P2先执行必定阻塞，只有在进程P1执行完S1； signal(S)； 操作后使S增加为1时，P2进程方能成功执行语句S2。

### 2. 利用信号量实现前趋关系

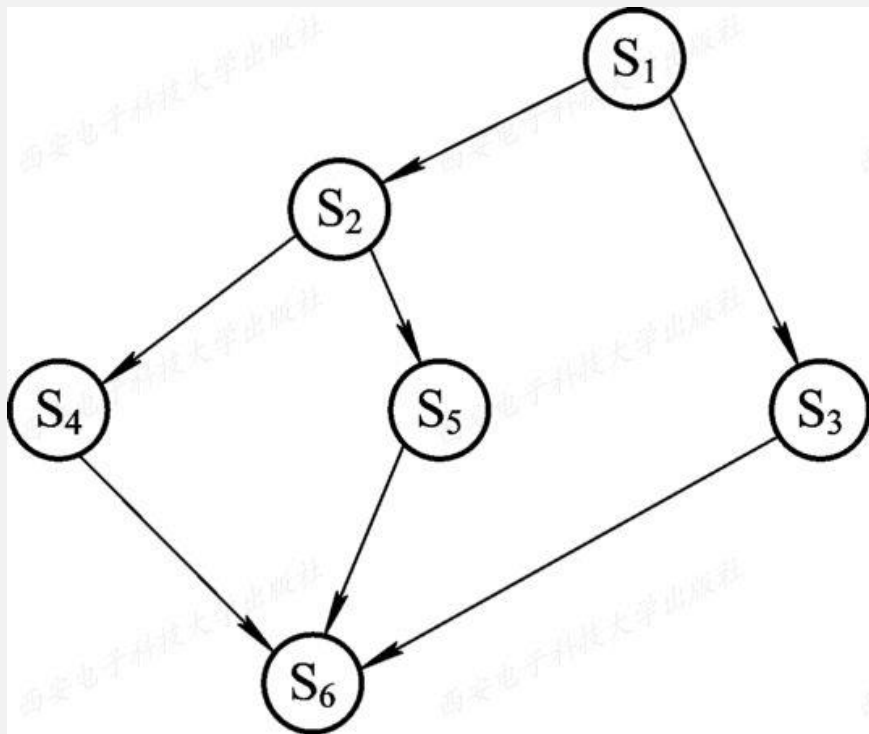


图2-14 前趋图举例

```
p1(){S1; signal(a); signal(b);}
p2(){wait(a); S2; signal(c); signal(d);}
p3(){wait(b); S3; signal(e);}
p4(){wait(c); S4; signal(f);}
p5(){wait(d); S5; signal(g);}
p6(){wait(e); wait(f); wait(g); S6;}

main(){
    semaphore a, b, c, d, e, f, g;
    a.value = b.value = c.value = 0;
    d.value = e.value = 0;
    f.value = g.value = 0;
    cobegin
        p1(); p2(); p3(); p4(); p5(); p6();
    coend
}
```

虽然信号量机制是一种既方便、又有效的同步机制，但每个要访问临界资源的进程都必须**自备同步操作wait(S)和signal(S)**。这就使**大量的同步操作分散在各个进程中**。这不仅给系统的管理带来了麻烦，而且还会因为同步操作的使用不当而导致系统死锁。

在解决上述问题的过程中，便产生了一种新的进程同步工具——**管程(Monitors)**。由**代表共享资源的数据结构以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序共同构成了一个操作系统的资源管理模块，称为管程。**

### 1. 管程的定义

```
semaphore mutex = 1;
```

```
PA{  
    while(1){  
        wait(mutex);  
        临界区;  
        signal(mutex);  
        剩余区;  
    }  
}  
  
PB{  
    while(1){  
        wait(mutex);  
        临界区;  
        signal(mutex);  
        剩余区;  
    }  
}
```

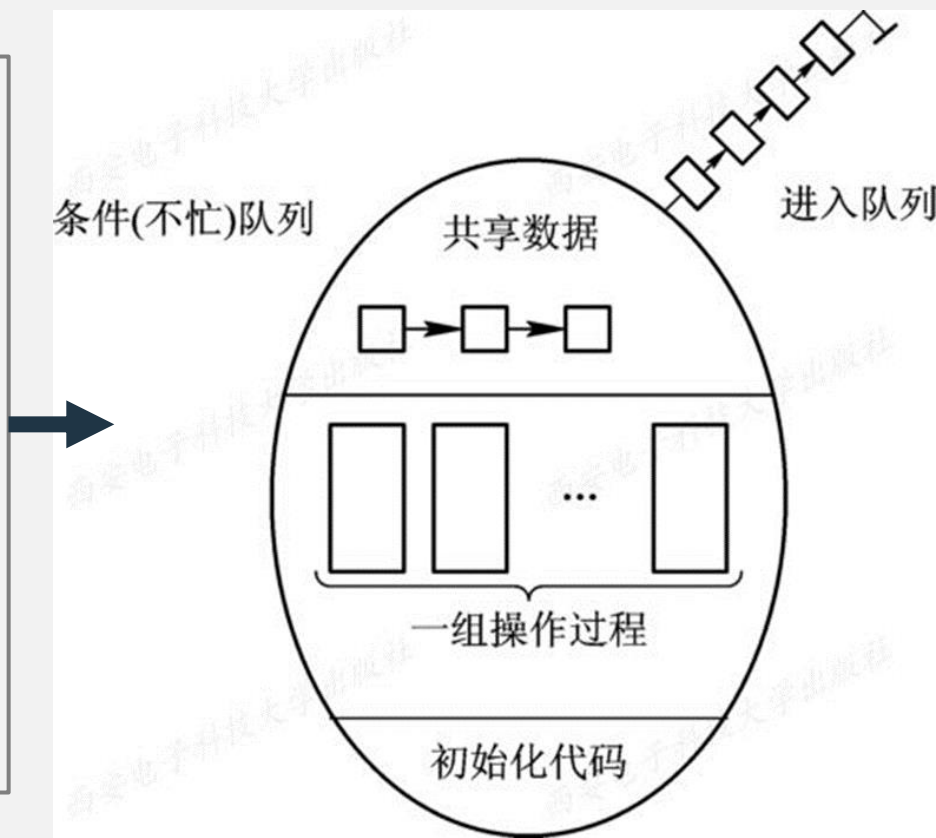


图2-15 管程的示意图

## 2.4.5 管程机制

### 1. 管程的定义

#### 管程

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。

- 管程的名称
- 局部于管程的共享数据结构说明
- 对该数据结构进行操作的一组过程
- 对局部于管程的共享数据设置初始值得语句

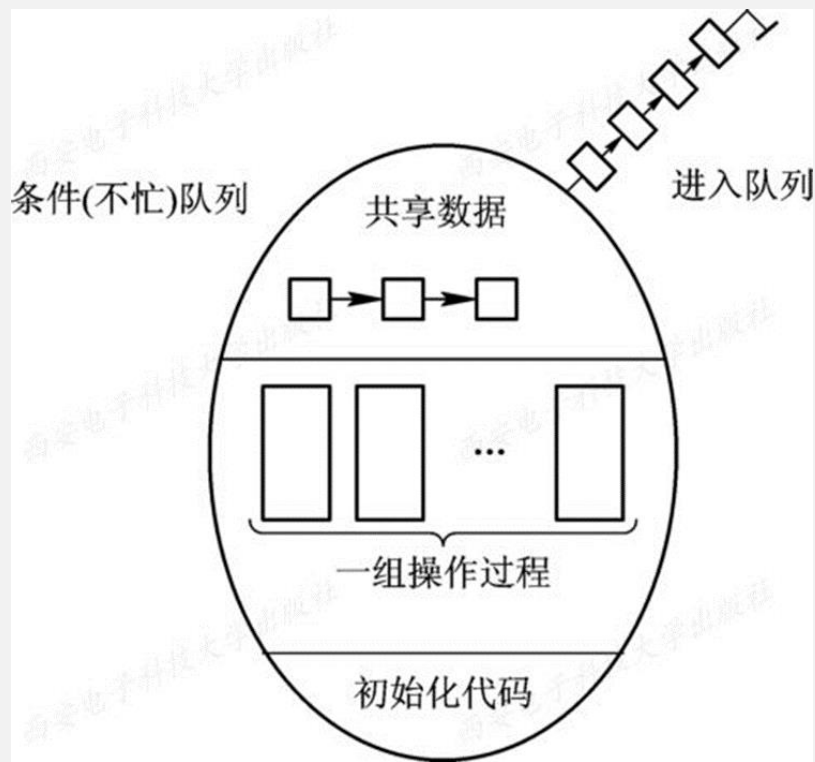


图2-15 管程的示意图

### 1. 管程的定义

管程的语法描述如下：

```
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  
        void P2(.....){.....} 的过程 */  
        .....  
        void(.....){.....}  
        .....  
        {                               /* 管程主体 */  
            initialization code; /* 初始化代码 */  
            .....  
        }  
}
```

所有进程要访问临界资源时，都只能通过管程**间接访问**，而管程每次只允许一个进程进入管程，执行管程内的过程，从而实现了互斥。

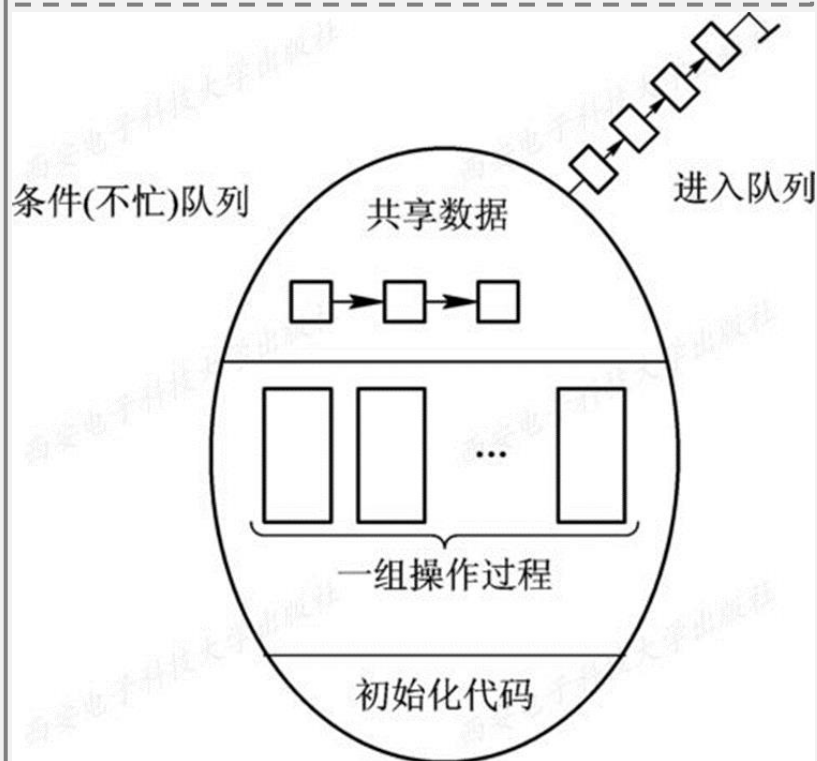


图2-15 管程的示意图

## 2.4.5 管程机制

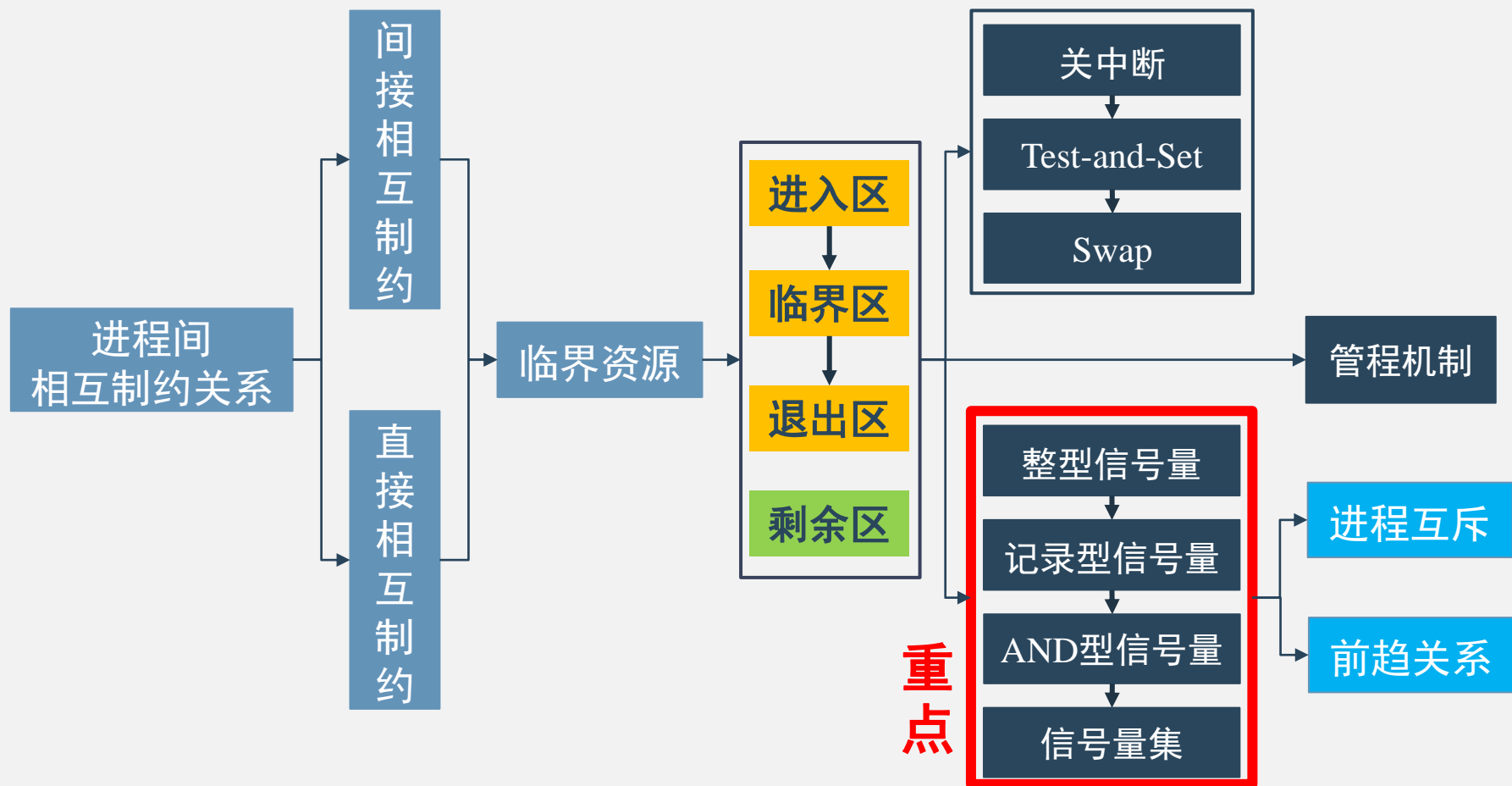
### 2. 条件变量

管程中的条件变量是一种**抽象数据类型**，管程对每个条件变量都必须予以说明，形式为：condition x, y；对条件变量的操作仅仅是wait和signal操作，每个条件变量保存了一个链表，用于记录因该条件变量而阻塞的所有进程。

- x.wait：正在调用管程的进程因x条件需要被阻塞或挂起，则调用x.wait将自己插入到x条件的等待队列上，并释放管程，直到x条件变化；此时其他进程可以使用该管程。
- x.signal：正在调用管程的进程发现x条件发生了变化，则调用x.signal，重新启动一个因x条件而阻塞或挂起的进程，如果存在多个这样的进程，则选择其中一个。

## 2.4 进程同步总结

进程同步机制的主要任务：对多个相关进程在执行次序上进行协调，使并发执行的诸进程之间能按照一定的规则(或时序)共享系统资源，并能很好地合作，从而使程序的执行具有可再现性。





## 第二章 进程的描述与控制

### 2.5 经典进程的同步问题

**问题：利用进程同步方法和机制，解决进程同步的问题**

**目标：**

- 掌握生产者—消费者问题的解决方法
- 掌握读者—写者问题的解决方法
- 加深对信号量机制的理解并掌握使用方法

## 2.5.1 生产者—消费者问题



- 生产者生产(进程)和 消费者消费(进程) 是异步运行的；
- 它们必须保持**同步**：
  - 消费者不能去空缓冲区中取出商品；
  - 生产者不能往满缓冲区中放入商品；

根据以上条件设计程序来解决该问题。

## 2.5.1 生产者—消费者问题

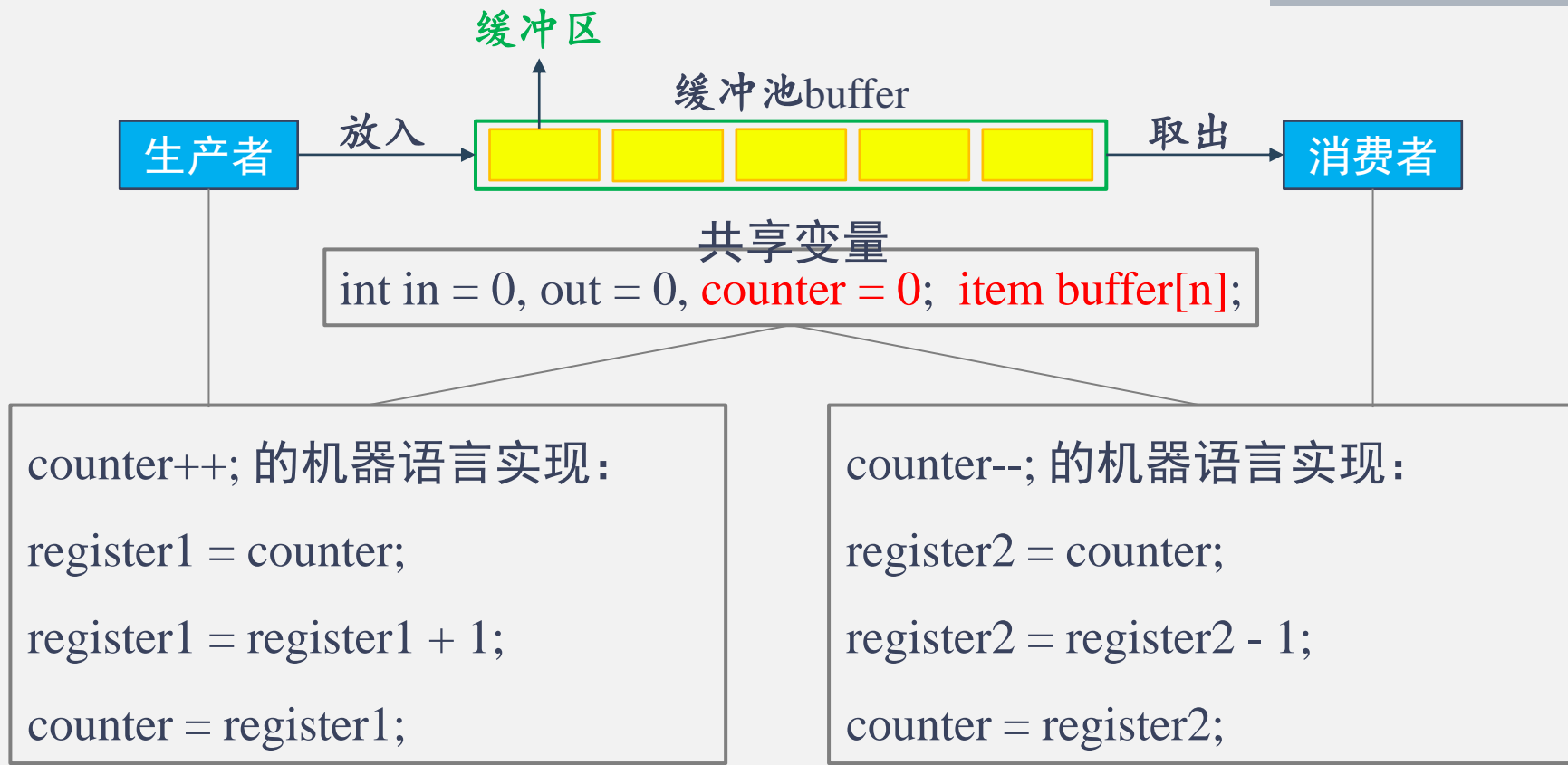


```
int in = 0, out = 0, counter = 0; item buffer[n];
```

```
void producer(){
    while(1){
        produce an item in nextp;
        ...
        while(counter == n);
        buffer[in] = nextp;
        in = (in + 1) % n;
        counter++;
    }
}
```

```
void consumer(){
    while(1){
        while(counter == 0);
        nextc = buffer[out];
        out = (out + 1) % n;
        counter--;
        consume the item in nextc;
        ...
    }
}
```

## 2.5.1 生产者—消费者问题



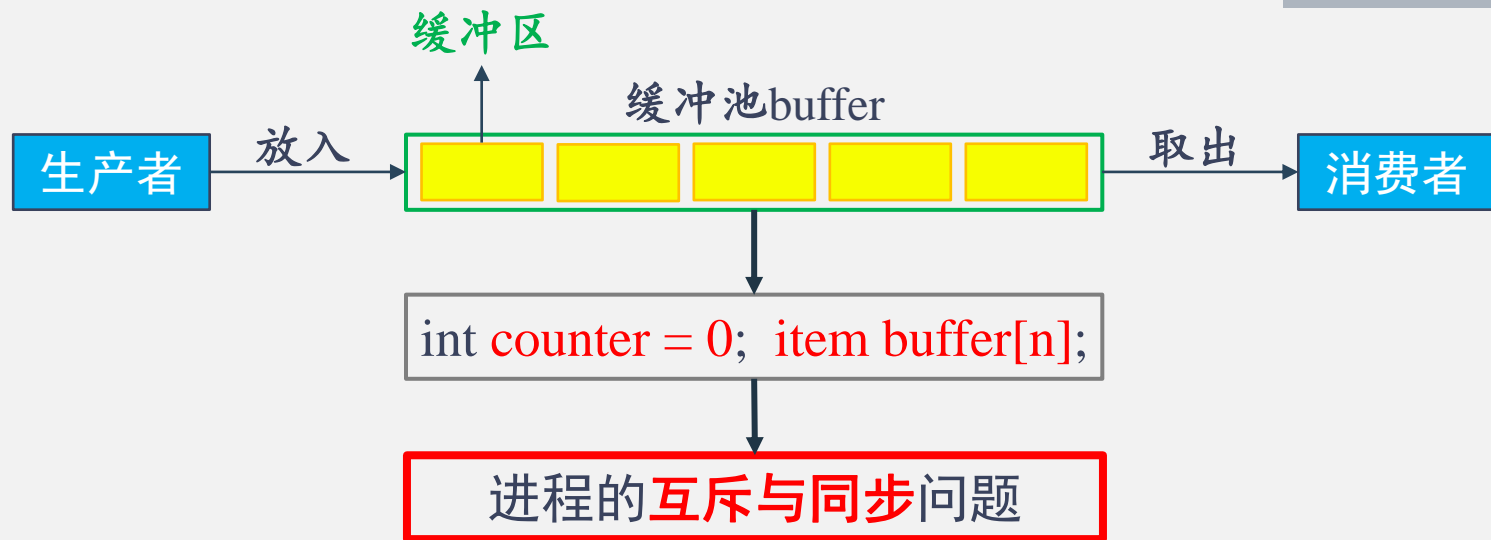
程序的执行已经失去了再现性，解决此问题的关键在于：把counter作为临界资源处理，即让生产者进程和消费者进程互斥地访问变量counter。

## 2.5.1 生产者—消费者问题



程序的执行已经失去了再现性，解决此问题的关键在于：把counter作为临界资源处理，即让生产者进程和消费者进程互斥地访问变量counter。

## 2.5.1 生产者—消费者问题



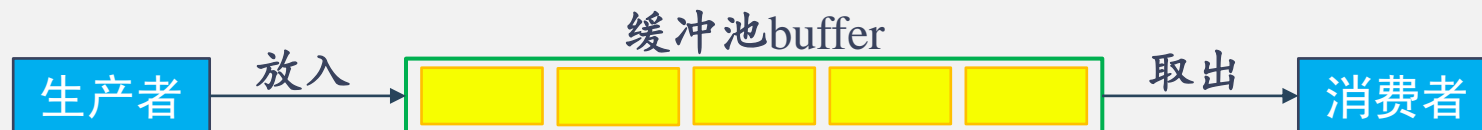
生产者—消费者问题是相互合作的进程关系的一种抽象，如：

- 在输入时，输入进程是生产者，计算进程是消费者
- 在输出时，计算进程是生产者，打印进程是消费者

本小节，利用信号量机制来解决生产者—消费者问题。

## 2.5.1 生产者—消费者问题

### 1. 利用记录型信号量解决生产者—消费者问题



```
int in, out = 0; item buffer[n];  
semaphore mutex = 1, empty = n, full = 0;
```

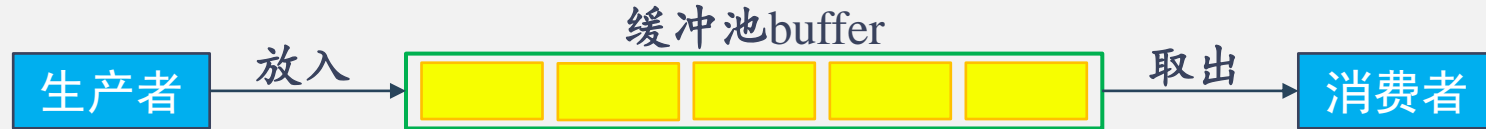
```
void producer(){  
    do{  
        produce an item in nextp;  
        ...  
        wait(empty);  
        wait(mutex);  
        buffer[in] = nextp;  
        in = (in + 1) % n;  
        signal(mutex);  
        signal(full);  
    }while(TRUE);  
}
```

```
void consumer(){  
    do{  
        wait(full);  
        wait(mutex);  
        nextc = buffer[out];  
        out = (out + 1) % n;  
        signal(mutex);  
        signal(empty);  
        consume the item in nextc;  
        ...  
    }while(TRUE);  
}
```



## 2.5.1 生产者—消费者问题

### 1. 利用记录型信号量解决生产者—消费者问题



```
int in, out = 0; item buffer[n];  
semaphore mutex = 1, empty = n, full = 0;
```

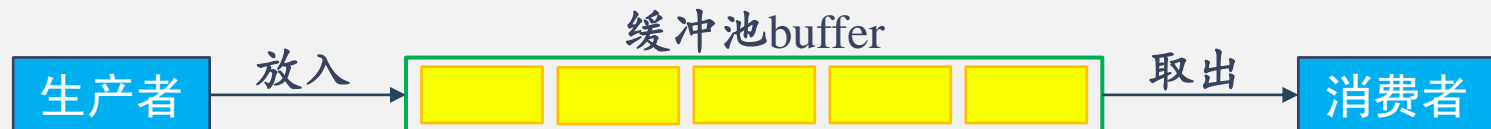
```
void producer(){  
    do{  
        produce an item  
        ...  
        wait(empty);  
        wait(mutex);  
        buffer[in] = next;  
        in = (in + 1) % n;  
        signal(mutex);  
        signal(full);  
    }while(TRUE);  
}
```

```
void main(){  
    cobegin  
        producer();  
        consumer();  
    coend  
}
```

```
void consumer(){  
    ...  
    wait(full);  
    wait(mutex);  
    nextc = buffer[out];  
    out = (out + 1) % n;  
    signal(mutex);  
    signal(empty);  
    consume the item in nextc;  
    ...  
}while(TRUE);  
}
```

## 2.5.1 生产者—消费者问题

### 1. 利用记录型信号量解决生产者—消费者问题

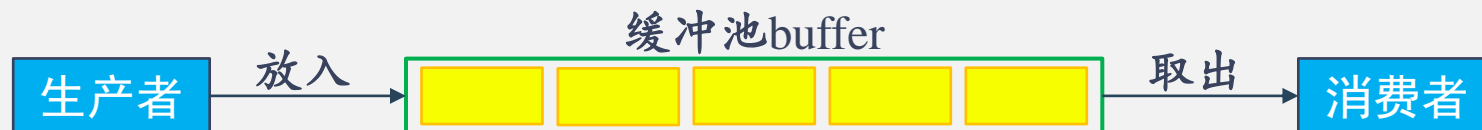


#### 生产者—消费者问题注意：

- 在每个程序中**互斥信号量**必须成对出现：wait(mutex); signal(mutex);
- 对**资源信号量**empty和full的P、V操作，成对出现在不同程序中。
- 每个程序中的多个wait操作顺序不能颠倒，否则会引起死锁。

## 2.5.1 生产者—消费者问题

### 2. 利用AND型信号量解决生产者—消费者问题



```
int in, out = 0; item buffer[n];  
semaphore mutex = 1, empty = n, full = 0;
```

```
void producer(){  
    do{  
        produce an item in nextp;  
        ...  
        Swait(empty, mutex);  
        buffer[in] = nextp;  
        in = (in + 1) % n;  
        Ssignal(mutex, full);  
    }while(TRUE);  
}
```

```
void consumer(){  
    do{  
        Swait(full, mutex);  
        nextc = buffer[out];  
        out = (out + 1) % n;  
        Ssignal(mutex, empty);  
        consume the item in nextc;  
        ...  
    }while(TRUE);  
}
```

## 2.5.1 生产者—消费者问题

### 3. 利用管程解决生产者—消费者问题

```
Monitor PC{
    item buffer[N];
    int in, out;
    condition notfull, notempty;
    int count;
    public:
        void put(item x){
            if(count >= N) cwait(notfull);
            buffer[in] = x;
            in = (in + 1) % N;
            count++;
            csignal(notempty);
        }
        void get(item x){
            if(count <= 0) cwait(notempty);
            x = buffer[out];
            out = (out + 1) % N;
            count--;
            csignal(notfull);
        }
        {in = 0; out = 0; count = 0;}
}PC;
```

管程的语法描述如下：

```
Monitor monitor_name{
    share variable declarations;
    cond declarations;
    public:
        void P1(.....){.....}
        void P2(.....){.....}
        .....
        void(.....){.....}
        .....
        {
            initialization code;
            .....
        }
}
```

## 2.5.1 生产者—消费者问题

### 3. 利用管程解决生产者—消费者问题

```
Monitor PC{
    item buffer[N];
    int in, out;
    condition notfull, notempty;
    int count;
    public:
        void put(item x){
            if(count >= N) cwait(notfull);
            buffer[in] = x;
            in = (in + 1) % N;
            count++;
            csignal(notempty);
        }
        void get(item x){
            if(count <= 0) cwait(notempty);
            x = buffer[out];
            out = (out + 1) % N;
            count--;
            csignal(notfull);
        }
        { in = 0; out = 0; count = 0; }
}PC;
```

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

```
void consumer(){
    item x;
    while(TRUE){
        PC.get(x);
        consume the item in x;
        ...
    }
}
```

## 2.5.3 读者—写者问题



一个数据文件或记录可被多个进程共享：

- **Reader:** 读文件的进程，多个进程可同时读一个共享对象；
- **Writer:** 写文件的进程，不允许一个Writer进程和其他Reader进程或Writer进程同时访问共享对象；

**Readcount=0**

读者—写者问题：指保证一个Writer进程必须与其他进程互斥地访问共享对象的同步问题。

## 2.5.3 读者—写者问题

### 1. 利用记录型信号量解决读者—写者问题

```
semaphore rmutex = 1, wmutex = 1;  
int readcount = 0;
```

用于控制写进程与其他进程互斥访问文件

```
void Reader{  
    do{  
        wait(rmutex);  
        if(readcount==0) wait(wmutex);  
        readcount++;  
        signal(rmutex);  
        ...  
        perform read operation;  
        ...  
        wait(rmutex);  
        readcount--;  
        if(readcount==0) signal(wmutex);  
        signal(rmutex);  
    }while(TRUE);  
}
```



```
void Writer{  
    do{  
        wait(wmutex);  
        perform write operation;  
        signal(wmutex);  
    }while(TRUE);  
}
```

```
void main(){  
    cobegin  
        Reader(); Writer();  
    coend  
}
```

## 2.5.3 读者—写者问题

### 2. 利用信号量集解决读者—写者问题

$\text{Swait}(S_1, t_1, d_1, \dots, S_n, t_n, d_n); \text{Signal}(S_1, t_1, d_1, \dots, S_n, t_n, d_n);$



增加限制：最多RN个读者同时读；

控制读文件

控制写文件

```
void Reader{
    do{
        Swait(L, 1, 1);
        Swait(mx, 1, 0);
        ...
        perform read operation;
        ...
        Ssignal(L, 1);
    }while(TRUE);
}
```

```
int RN;
semaphore L = RN, mx = 1;
```

```
void Writer{
    do{
        Swait(mx, 1, 1; L, RN, 0);
        perform write operation;
        Ssignal(mx, 1);
    }while(TRUE);
}
```



## 本节总结

1. 判断同步问题类型；
2. 判断同步资源和同步条件，利用互斥信号量表示互斥共享资源，利用资源信号量表示其余共享资源，分配信号量初始值；
3. 判断进入区、临界区、退出区，将申请资源操作放在进入区，使用资源的操作放在临界区，释放资源的操作放在退出区，申请、释放资源利用信号量P\V操作实现。

## 第二章 进程的描述与控制

### 2.6 进程通信

进程通信：指进程之间的信息交换。

- **低级通信：**进程间仅交换一些状态和少量数据；如：进程之间的互斥和同步；
  - (1) 效率低；(2) 通信对用户不透明；
- **高级通信：**进程之间可交换大量数据；
  - 用户可直接利用操作系统提供的一组通信命令，高效地传送大量数据的一种通信方式；
  - 操作系统隐藏了进程通信的细节，对用户透明，减少了通信程序编制上的复杂性。

- I. 共享存储器系统(Shared-Memory System)
- II. 管道(Pipe)通信系统
- III. 消息传递系统(Message Passing System)
- IV. 客户机-服务器系统(Client-Server Syestem)

## 2.6.1 进程通信的类型

### 1. 共享存储器系统(Shared-Memory System)

相互通信的进程之间共享某些数据结构或共享存储区，通过这些空间进行通信。

#### (1) 基于共享数据结构的通信方式

低级通信

进程公用某些数据结构，借以实现诸进程间的信息交换。

实现：公用数据结构的设置以及对进程间同步的处理，都是程序员的职责；

操作系统——提供共享存储器；

特点：低效，只适合传递相对少量的数据；

## 2.6.1 进程通信的类型

### 1. 共享存储器系统(Shared-Memory System)

相互通信的进程之间共享某些数据结构或共享存储区，通过这些空间进行通信。

#### (2) 基于共享存储区的通信方式 **高级通信**

在存储器（内存）中划出一块共享存储区，诸进程可以通过对共享存储区中数据的读或写来实现通信。

实现：数据的形式和位置甚至访问控制都是由进程负责；

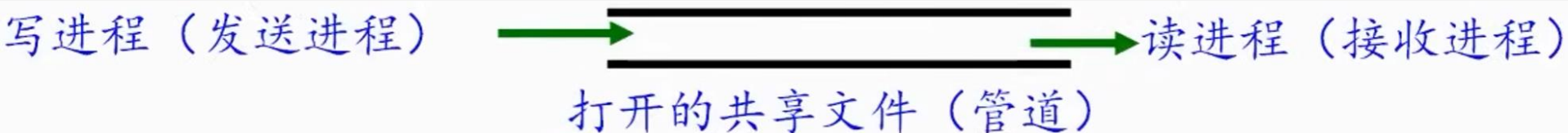
操作系统——提供共享存储区中的一个分区；

特点：可传输大量数据；

## 2.6.1 进程通信的类型

### 2. 管道(Pipe)通信系统

管道：指用于连接一个读进程和一个写进程以实现他们之间通信的一个打开的共享文件，又名pipe文件。



管道机制要提供的协调能力：**互斥、同步，确定对方是否存在；**

## 2.6.1 进程通信的类型

### 3. 消息传递系统(Message Passing System)

进程间的数据交换，以格式化的消息为单位。（最广泛）

程序员直接利用系统提供的一组通信命令(原语)进行通信。

例：计算机网络——网络报文  
微内核——消息传递机制

高级通信

- **直接通信方式：**指发送进程利用OS所提供的发送原语，直接把消息发送给目标进程；
- **间接通信方式：**指发送和接收进程，都通过共享中间实体(称为邮箱)的方式进行消息的发送和接收，完成进程间的通信；



## 2.6.2 消息传递通信的实现方式

### 1. 直接消息传递系统

#### 1) 直接通信原语:

##### 1. 对称寻址方式:

`send(receiver, message); receive(sender, message)`

##### 2. 非对称寻址方式

`send(P, message); receive(id, message)`

#### 2) 消息的格式 (定长、变长)

#### 3) 进程的同步方式

#### 4) 通信链路(显式、隐式)

## 2.6.2 消息传递通信的实现方式

### 2. 信箱通信

1) 信箱结构 P78

2) 信箱通信原语

1. 邮箱的创建和撤销

名字、属性、共享者

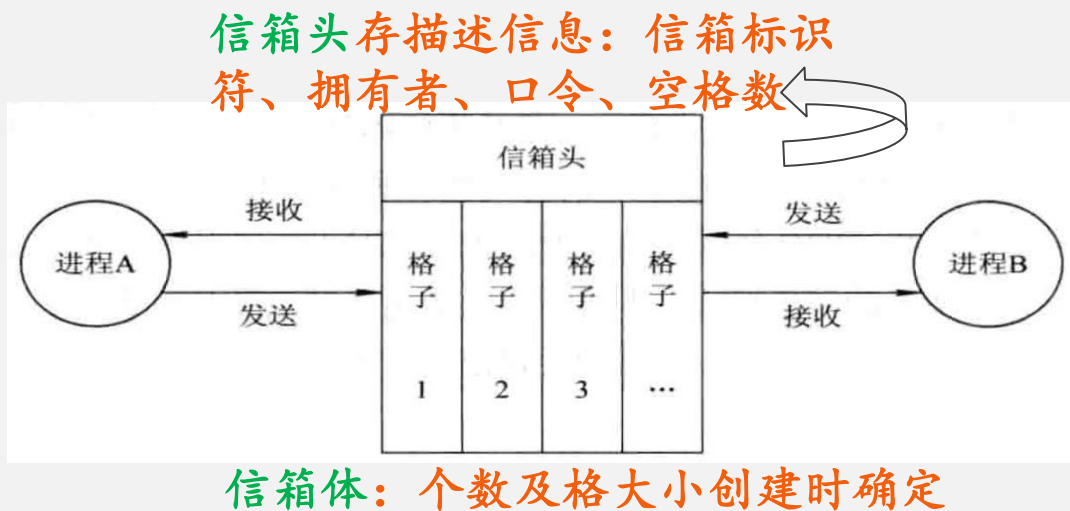
2. 消息的发送和接收

Send(mailbox, message); 将一个消息发送到指定邮箱

Receive(mailbox, message); 从指定邮箱中接收一个消息

3) 信箱的类型:

私用邮箱(用户进程); 公用邮箱(操作系统); 共享邮箱



## 第二章 进程的描述与控制

### 2.7 线程的基本概念

**问题：**如何提高程序并发执行的程度，进一步改善系统的服务质量

**目标：**

- 掌握线程的基本概念
- 了解线程和进程的区别
- 掌握线程的状态和线程控制块

**背景** 多道程序OS中以**进程**作为拥有资源和独立调度(运行)的基本单位。

↓  
**目标** 进一步提高程序并发执行的程度，以改善系统的服务质量。

### 1. 进程的两个基本属性

- **进程是一个可拥有资源的独立单位。**一个进程要能独立运行，必须拥有一定的资源，包括存放程序正文、数据的磁盘和内存地址空间，以及它在运行时所需要的I/O设备、已打开的文件、信号量等；
- **进程是一个可独立调度和分派的基本单位。**一个进程要能独立运行，还必须是一个可独立调度和分派的基本单位，每个进程在系统中有唯一的PCB，系统可根据其PCB感知进程的存在，根据PCB对进程进行调度，还可将断点信息保存在PCB或从PCB恢复进程运行现场。

**背景** 多道程序OS中以进程作为拥有资源和独立调度(运行)的基本单位。

↓  
**目标** 进一步提高程序并发执行的程度，以改善系统的服务质量。

**正是由于进程的这两个基本属性，才使进程成为一个能独立运行的基本单位，从而也构成了进程并发执行的基础。**

拥有一定的资源，包括存放程序正文、数据的磁盘和内存地址空间，以及它在运行时所需要的I/O设备、已打开的文件、信号量等；

- **进程是一个可独立调度和分派的基本单位。**一个进程要能独立运行，还必须是一个可独立调度和分派的基本单位，每个进程在系统中有唯一的PCB，系统可根据其PCB感知进程的存在，根据PCB对进程进行调度，还可将断点信息保存在PCB或从PCB恢复进程运行现场。

## 2.7.1 线程的引入

### 2. 程序并发执行所需付出的时空开销

为使程序能并发执行，系统必须进行以下的一系列操作：

- (1) **创建进程**。系统在创建一个进程时，必须为它分配其所必需的、除处理机以外的所有资源，如内存空间、I/O设备，以及建立相应的PCB；
- (2) **撤消进程**。系统在撤消进程时，又必须先对其所占有的资源执行回收操作，然后再撤消PCB；
- (3) **进程切换**。对进程进行上下文切换时，需要保留当前进程的CPU环境，设置新选中进程的CPU环境，因而须花费不少的处理机时间。

## 2.7.1 线程的引入

### 2. 程序并发执行所需付出的时空开销

进程是一个资源的拥有者，因而在创建、撤销和切换中，系统必须为之付出较大的时空开销。这就限制了系统中所设置进程的数目，而且进程切换也不宜过于频繁，从而限制了并发程度的进一步提高。



## 2.7.1 线程的引入

### 3. 线程——作为调度和分派的基本单位

如何能使多个程序更好地并发执行，同时又尽量减少系统的开销，已成为近年来设计操作系统时所追求的重要目标。有不少研究操作系统的学者们想到，要设法将进程的上述两个属性分开，由OS分开处理，亦即并不把作为调度和分派的基本单位也同时作为拥有资源的单位，以做到“轻装上阵”；而对于拥有资源的基本单位，又不对之施以频繁的切换。**将线程作为调度和分派的基本单位。**

## 2.7.2 线程与进程的比较

轻型进程

线程

VS

进程

重型进程

1. 调度的基本单位
2. 并发性
3. 拥有资源
4. 独立性
5. 系统开销
6. 支持多处理机系统

## 2.7.2 线程与进程的比较

### 1. 调度的基本单位

#### 线程

- 在引入线程OS中，已把线程作为调度和分派的基本单位，因而线程是能独立运行的基本单位；
- 线程切换时，仅需保存和设置少量寄存器内容，切换代价远低于进程。

VS

#### 进程

- 在传统OS中，进程作为独立调度和分派的基本单位，是能独立运行的基本单位；
- 每次被调度时，都需要进行上下文切换，开销较大。

同一进程中，线程的切换不会引起进程调度；但从一个进程中的线程切换到另一个进程中的线程时，必然就会引起进程的切换。

## 2.7.2 线程与进程的比较

### 2. 并行性

#### 线程

- 在引入线程OS中，不仅进程之间可并发执行，同一进程中的多个线程之间也可并发执行，甚至允许一个进程中的所有线程都能并发执行；
- 不同进程中的线程也可并发执行。

VS

#### 进程

- 在传统OS中，仅多个进程之间可以并发执行。

- 文字处理器设置三个线程：显示文字和图形、读入数据、拼写和检查；
- 应用程序中执行多个相似任务：网络服务器中每个线程监听一个用户。

## 2.7.2 线程与进程的比较

### 3. 拥有资源

#### 线程

- 线程本身不拥有系统资源，仅有一点必不可少的、能保证独立运行的资源。如：线程控制块TCB、程序计数器、保留局部变量、少数状态参数和返回地址的一组寄存器和堆栈。

VS

#### 进程

- 进程可以拥有资源，并作为系统中拥有资源的一个基本单位。

线程除拥有自己的少量资源外，还允许多个线程共享该进程所拥有的资源：①属于同一进程的所有线程具有相同地址空间，线程可访问该地址空间中每一个虚地址；②线程可以访问进程所拥有的资源。

### 4. 独立性

#### 线程

- 同一进程中的不同线程往往是为了提高并发性以及进行相互合作而创建的，共享进程的内存地址空间和资源。

VS

#### 进程

- 为防止进程之间彼此干扰和破坏，每个进程都拥有一个独立的地址空间和其他资源，除共享局部变量外，不允许其他进程访问。

同一进程中不同线程之间的独立性比不同进程之间的独立性低的多。如：每个线程都可访问它们所属进程地址空间中的所有地址等。

### 5. 系统开销

#### 线程

- 线程创建或撤消的资源分配或回收很小，线程切换代价也远低于进程；  
如在Solaris 2 OS中：
- 线程创建比进程创建快30倍；
- 线程上下文切换比进程上下文切换快5倍。

VS

#### 进程

- 创建或撤消进程时，系统要分配或回收进程控制块、分配或回收其他资源，如内存和I/O设备等；
- 进程切换时，涉及到进程上下文的切换。

此外，由于一个进程中的多个线程具有相同的地址空间，线程之间的同步和通信也比进程的简单。

## 2.7.2 线程与进程的比较

### 6. 支持多处理机系统

#### 线程

- 对于多线程进程，就可以将一个进程中的多个线程分配到多个处理机上，使它们并行执行，加速进程的完成。

VS

#### 进程

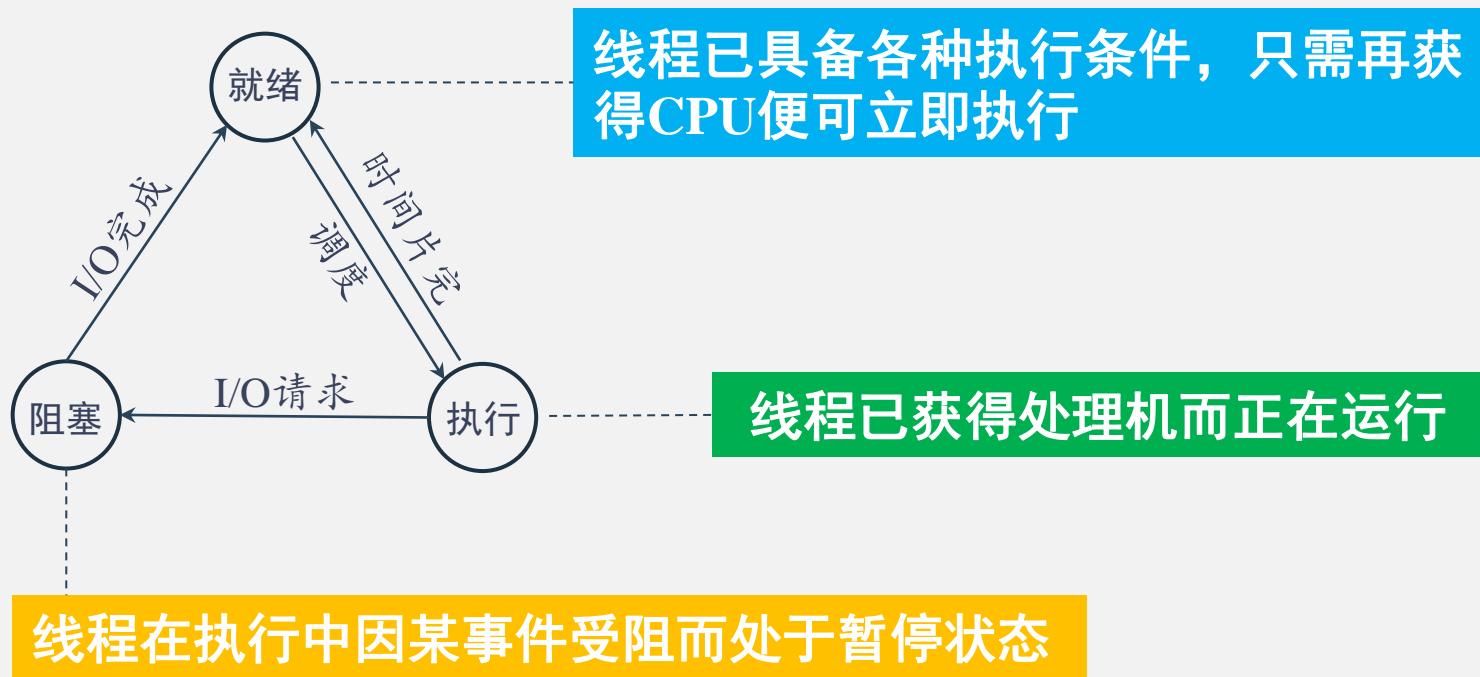
- 对传统的进程，即单线程进程，不管有多少处理机，该进程只能运行在一个处理机上。

现代多处理机OS都无一例外地引入了多线程。



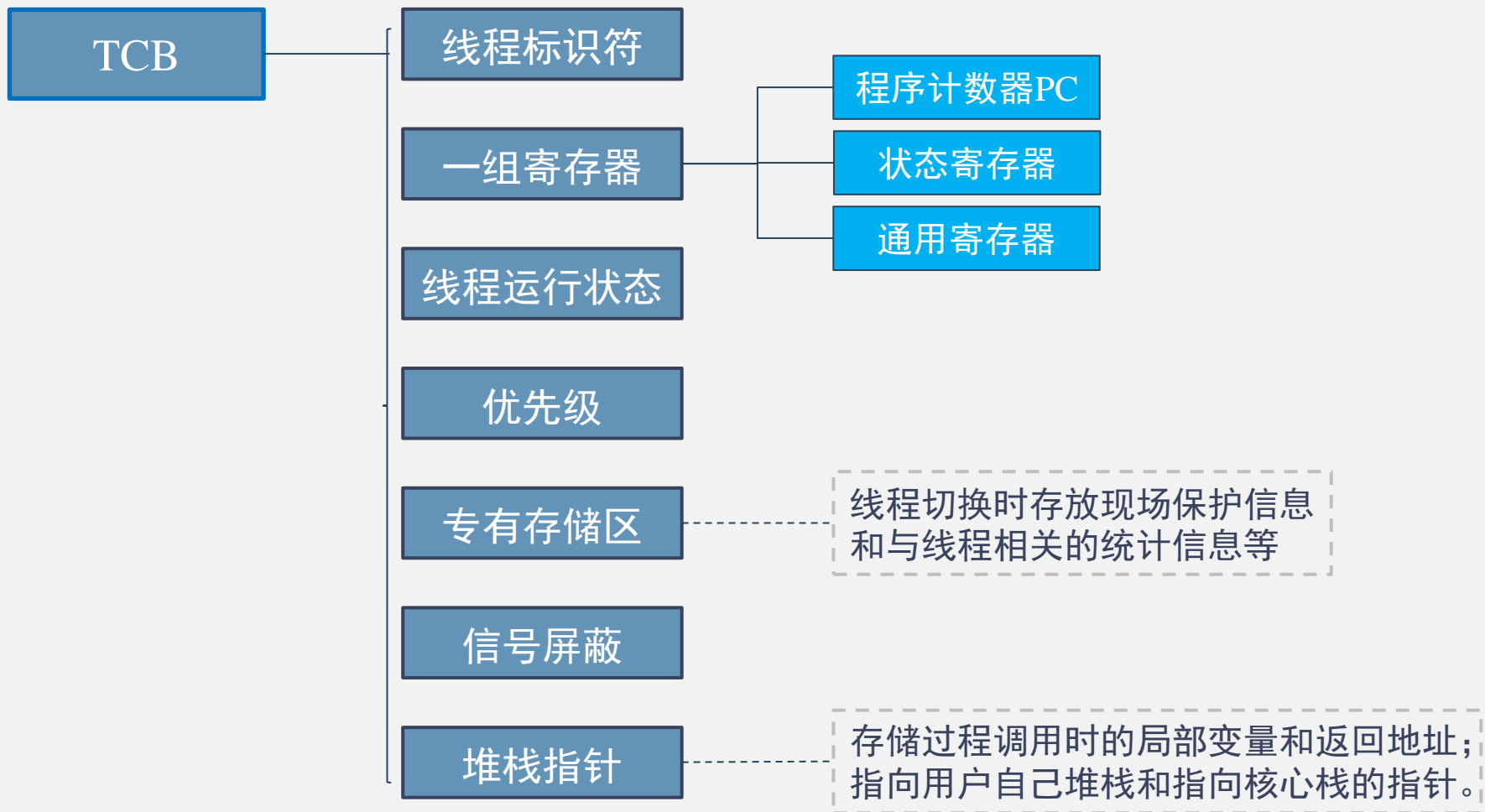
## 2.7.3 线程的状态和线程控制块

### 1. 线程运行的三个状态



## 2.7.3 线程的状态和线程控制块

### 2. 线程控制块TCB



## 2.7.3 线程的状态和线程控制块

### 3. 多线程OS中的进程属性

通常在线程OS中的进程都包含了多个线程，并为它们提供资源。OS支持在一个进程中的多个线程能并发执行，此时进程不再作为一个执行的实体。

进程是一个可拥有资源的基本单位。

在线程OS中，进程仍作为系统资源分配的基本单位，任一进程所拥有的资源包括：

- 用户的地址空间；
- 实现进程(线程)间同步和通信的机制；
- 已打开的文件；
- 已申请到的I/O设备；
- 一张由核心进程维护的地址映射表。

## 2.7.3 线程的状态和线程控制块

### 3. 多线程OS中的进程属性

多个线程可并发执行。

- 通常一个进程都含有若干个相对独立的线程，其数目可多可少，但至少有一个线程。此时，进程为这些线程提供资源及运行环境，使它们能并发执行。
- 在OS中的所有线程都只能属于某一个特定进程。
- 实际上，现在把传统进程的执行方法称为单线程方法，将每个进程支持多个线程执行的方法称为多线程方法。

如Java的运行环境是单进程多线程的，Windows 2000、Solaris、Mach等采用的则是多进程多线程的方法。

## 2.7.3 线程的状态和线程控制块

### 3. 多线程OS中的进程属性

进程不再是可执行的实体。

- 在多线程OS中，是把线程作为独立运行(或调度)的基本单位。此时的进程已不再是一个基本的可执行实体。
- 进程仍具有与执行相关的状态。例如：所谓进程处于“执行”状态，实际上是指该进程中的某些线程正在执行。
- 对进程所施加的与进程状态有关的操作对其线程也起作用。例如，在把某个进程挂起时，该进程中的所有线程也都将被挂起；把某进程激活时，属于该进程的所有线程也都将被激活。

## 2.8.1 线程的实现方式

- **内核支持线程KST**：在内核的支持下运行，其创建、阻塞、撤销和切换等，也是在内核空间实现。
- **用户级线程ULT**：在用户空间中实现，对线程的创建、撤销、同步与通信等功能，都无需内核支持。
- **组合方式**：把用户级线程和内核支持线程两种方式进行结合。

谢谢大家! Q & A