



课程主要内容

👉 操作系统引论（第1章）

👉 进程管理（第2-3章）

👉 存储器管理（第4-5章）

👉 设备管理（第6章）

👉 文件管理（第7-8章）

👉 操作系统接口（第9章）

👉 Unix操作系统



从进程的观点研究操作系统

- 把OS看作是由若干个可独立运行的程序和一个可对这些程序进行协调控制的核心（内核）组成。
- 这些运行的程序称为进程，它是资源分配和独立运行的基本单位，每一个进程都完成某一特定任务。
- OS的内核则必须要控制和协调这些进程的运行，解决进程之间的通信，并从系统可并发工作为出发点，实现并发进程间通信，并解决由此带来的共享资源的竞争问题。



第2章 进程管理 Process Management

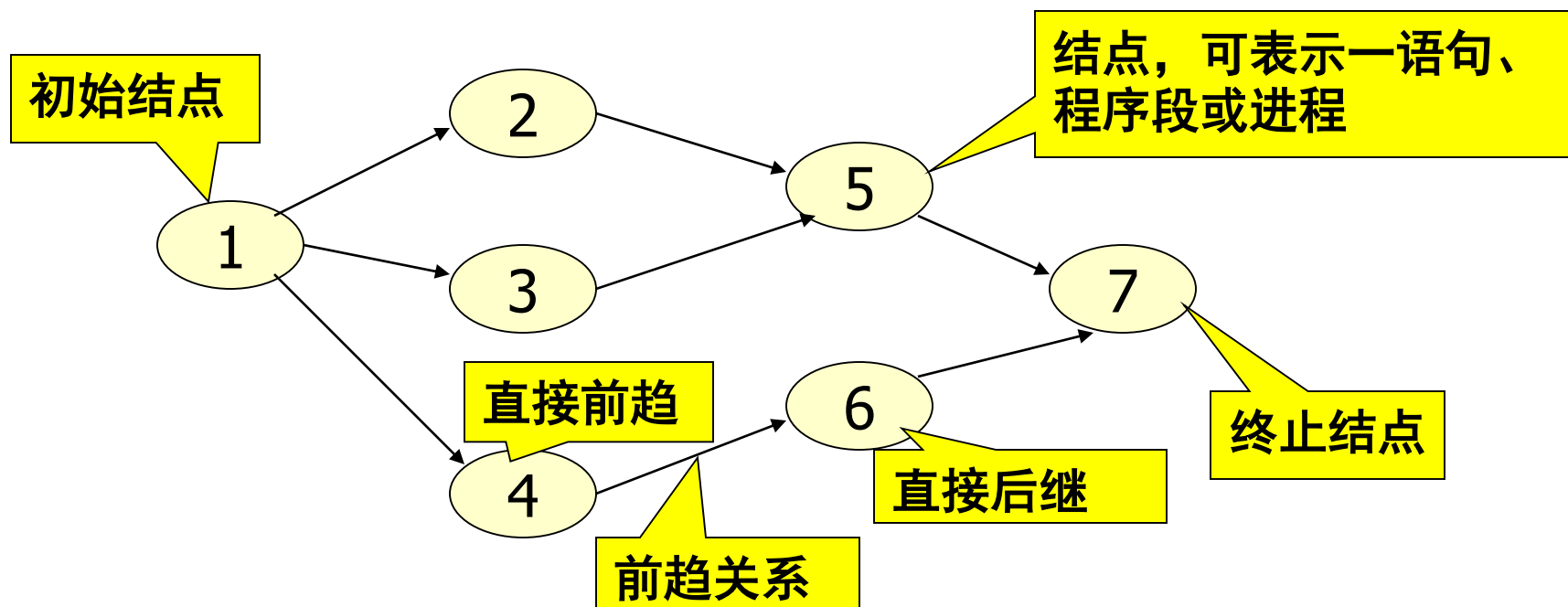
- 进程的基本概念与控制
 - 进程的基本概念
 - 进程控制
 - 线程的基本概念
 - UNIX中进程的描述与控制
- 进程同步与通信
 - 进程同步
 - 经典进程的同步问题
 - 管程机制
 - 进程通信
 - UNIX中进程的同步与通信
- 调度与死锁（第3章）



2.1 进程的基本概念

- 前趋图
- 程序顺序执行
- 程序并发执行
- **进程的描述**
 - 进程的定义、特征
 - 进程的状态（状态、状态转换 及挂起状态）
 - 进程控制块PCB

一、前趋图的定义



前趋关系: $P_1 \rightarrow P_2$, $P_2 \rightarrow P_5$, $P_5 \rightarrow P_7$
 $P_1 \rightarrow P_3$, $P_3 \rightarrow P_5$
 $P_1 \rightarrow P_4$, $P_6 \rightarrow P_7$

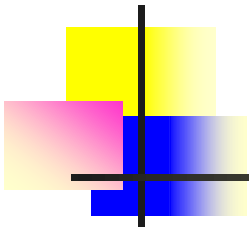
有向无循环图,
记作DAG



■ 前趋图：

定义：有向无循环图（DAG），是一个二元组，由结点的集合和有向边组成。其中：

- 结点：表示一条语句、一个程序段或一个进程
- 有向边：两个结点之间的前趋关系 “ \longrightarrow ”
($\longrightarrow = \{(P_i, P_j) | P_i \text{ 必须在 } P_j \text{ 开始执行之前完成}\}$)。
- 直接前趋、直接后继
- 初始结点、终止结点

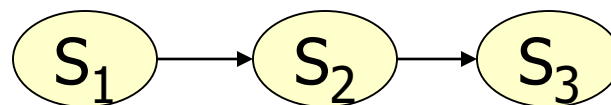


Eg1: 以下三条语句的前趋图为:

S_1 : $a := x + y$

S_2 : $b := a - 5$

S_3 : $c := b + 1$

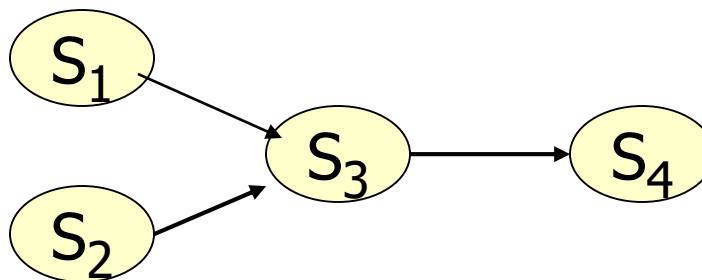


Eg2: **S_1 :** $a := x + 2$

S_2 : $b := y + 4$

S_3 : $c := a + b$

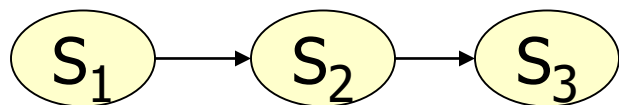
S_4 : $d := c + 6$



二、程序顺序执行(1)

- 通常一个程序可分成若干个程序段，它们必须按照某种先后次序执行，仅当前一操作执行后，才能执行后继操作。

- Eg1: $S_1: a := x + y$
 $S_2: b := a - 5$
 $S_3: c := b + 1$



- Eg2: 进行**计算**: I: 输入操作 C: 计算操作
P: 打印操作 在进行计算时，总是先输入用户的程序和数据，然后进行计算，最后将结果打印出来。



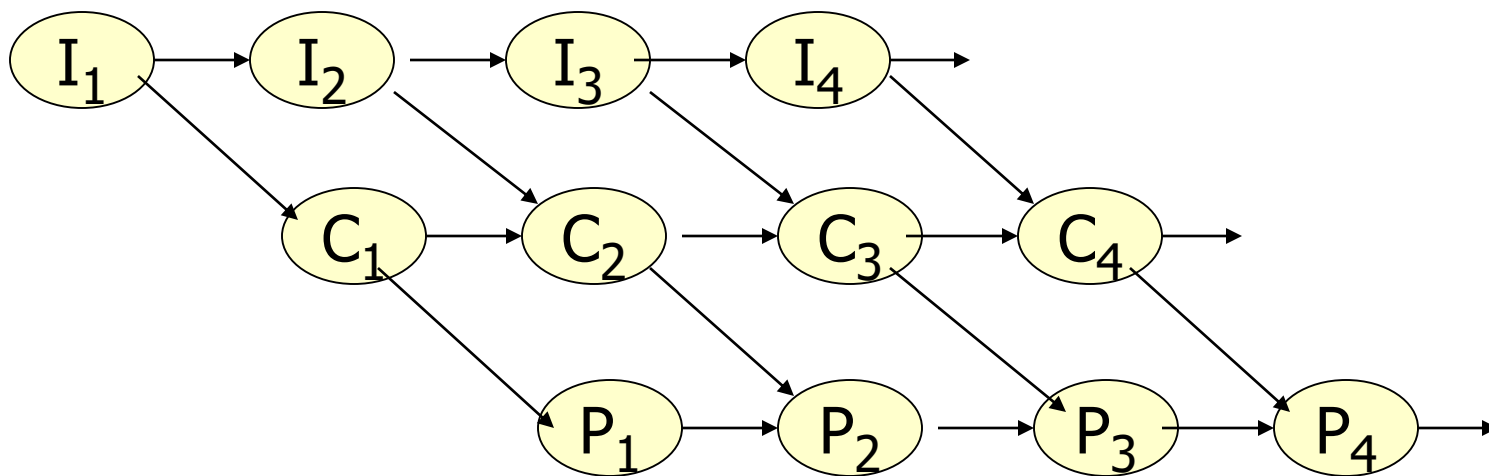


二、程序顺序执行(2)

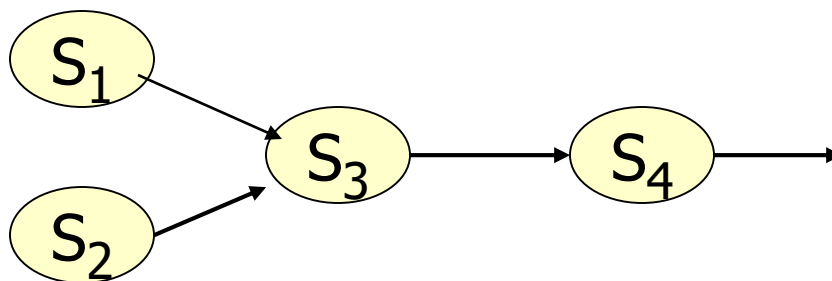
- 程序顺序执行时有如下特征：
 - 顺序性—处理机的操作严格按照程序所规定的顺序执行，每一操作必须在下一个操作开始之前结束。
 - 封闭性—在封闭环境下执行，独占全机资源，执行结果不受外界影响。
 - 可再现性—只要程序执行时的环境和初始条件相同，当程序重复执行时，不论它是从头到尾不停顿地执行，还是“走走停停”地执行，都将获得相同的结果。

三、程序并发执行 (1)

在处理一批作业时，有的程序可实现并发执行



■ S_1 : $a := x + 2$
 S_2 : $b := y + 4$
 S_3 : $c := a + b$
 S_4 : $d := c + 6$





三、程序并发执行（2）

- 程序并发执行时的特征
 - 间断性—相互制约导致并发程序具有“执行-暂停-执行”这种间断性的活动规律。
 - 失去封闭性—多个程序共享系统中的各种资源，资源状态由多个程序来改变。
 - 不可再现性—由于程序的并发执行，打破了由一个程序独占系统资源的封闭性，因而破坏了可再现性。
- （补充）程序并发执行的条件（Bernstein条件）（见下页）



三、程序并发执行（3）

- **（补充）** 程序并发执行的条件（Bernstein条件）（使并发执行的程序能保持可再现性）：
 - 读集 $R(p_i) = \{a_1, a_2, a_3, \dots, a_m\}$ ：表示程序 p_i 在执行期间所需参考的所有变量的集合。
 - 写集 $W(p_i) = \{b_1, b_2, b_3, \dots, b_m\}$ ：表示程序 p_i 在执行期间要改变的所有变量的集合。
 - Bernstein条件：

若两个程序 p_1 和 p_2 能满足下述条件，它们便能并发执行，且具有可再现性。

$$R(p_1) \cap W(p_2) \cup R(p_2) \cap W(p_1) \cup W(p_1) \cap W(p_2) = \{\}$$



程序并发执行条件例题

■ Eg $S_1: a:=x+2$ $S_2: b:=z+4$ $S_3: c:=a-b$ $S_4: w:=c+1$

试利用**Bernstein**条件证明：

(1) S_1 与 S_2 并发执行； (2) S_1 与 S_3 , S_2 与 S_3 , S_3 与 S_4 不能。

解：各语句的读、写集分别为：

$R(S_1)=\{x\}, \quad W(S_1)=\{a\},$

$R(S_2)=\{z\}, \quad W(S_2)=\{b\},$

$R(S_3)=\{a,b\}, \quad W(S_3)=\{c\},$

$R(S_4)=\{c\}, \quad W(S_4)=\{w\},$

因为 $R(S_1) \cap W(S_2)=\{\}, R(S_2) \cap W(S_1)=\{\}$

且 $W(S_1) \cap W(S_2)=\{\}$

所以由**Bernstein**条件， S_1 与 S_2 并发执行。

同理可证 S_1 与 S_3 , S_2 与 S_3 , S_3 与 S_4 不能并发执行（略）。



进程的定义、特征

1、进程(process)的定义

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

注：进程与程序的主要区别（见下页）



进程与程序的主要区别

- 程序是指令的有序集合，其本身没有任何运行的含义，它是一个**静态**的概念。而进程是程序在处理器上的一次执行过程，它是一个**动态**概念。
- 程序的**存在是永久**的。而**进程**则是**有生命期**的，它因创建而产生，因调度而执行，因得不到资源而暂停执行，因撤消而消亡。
- 程序仅是**指令的有序集合**。而进程则由**程序段、相关数据段、进程控制块（PCB）**组成。
- **进程与程序之间不是一一对应**。



进程与程序的主要区别

	程序	进程
概念	静态	动态
所在存储器	外存	内存
存在时间	永久	有生命期
组成	有序指令	程序段, 数据段, PCB
对应关系	一个程序可对应多个进程 一个进程可包含多个程序（至少包含一个程序）	



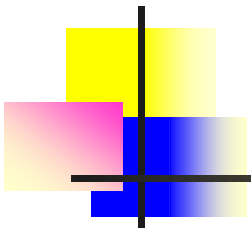
进程的特征

2、进程（process）的基本特征

（1）结构特征

为了描述和记录进程的运动变化过程，并使之能正确运行，每个进程都应配置了一个PCB。所以，从结构上看，每个进程（进程实体）都是由**程序段、相关数据段及进程控制块（PCB）**组成。

注：在早期UNIX版本中称进程的三个组成部分为“进程映像”



(2) 动态性

进程的实质是**程序在处理机上的一次执行过程**，因此是动态的。所以动态性是进程的最基本的特征。同时动态性还表现在**进程是有生命期**的，它因创建而产生，因调度而执行，因得不到资源而暂停执行，因撤消而消亡。



(3) 并发性

指多个进程实体同时存在于内存中，能在一段时间内同时运行。

引入进程的目的就是为了使进程能并发执行，以提高资源利用率，所以并发性是进程的重要特征，也是OS的重要特征。

(4) 独立性

指进程是一个能独立运行的基本单位，也是系统进行资源分配和调度的独立单位。

(5) 异步性

指进程以各自独立的、不可预知的速度向前推进。



进程的状态（1）

为了刻画了整个进程，可以将一个进程的生命周期划分为一组状态：

1、进程的五种状态（两种短暂的状态、三种基本状态）

- new新建/创建：进程正在创建中的状态
- terminated终止/撤消/退出：进程执行完毕，释放所占资源的状态。



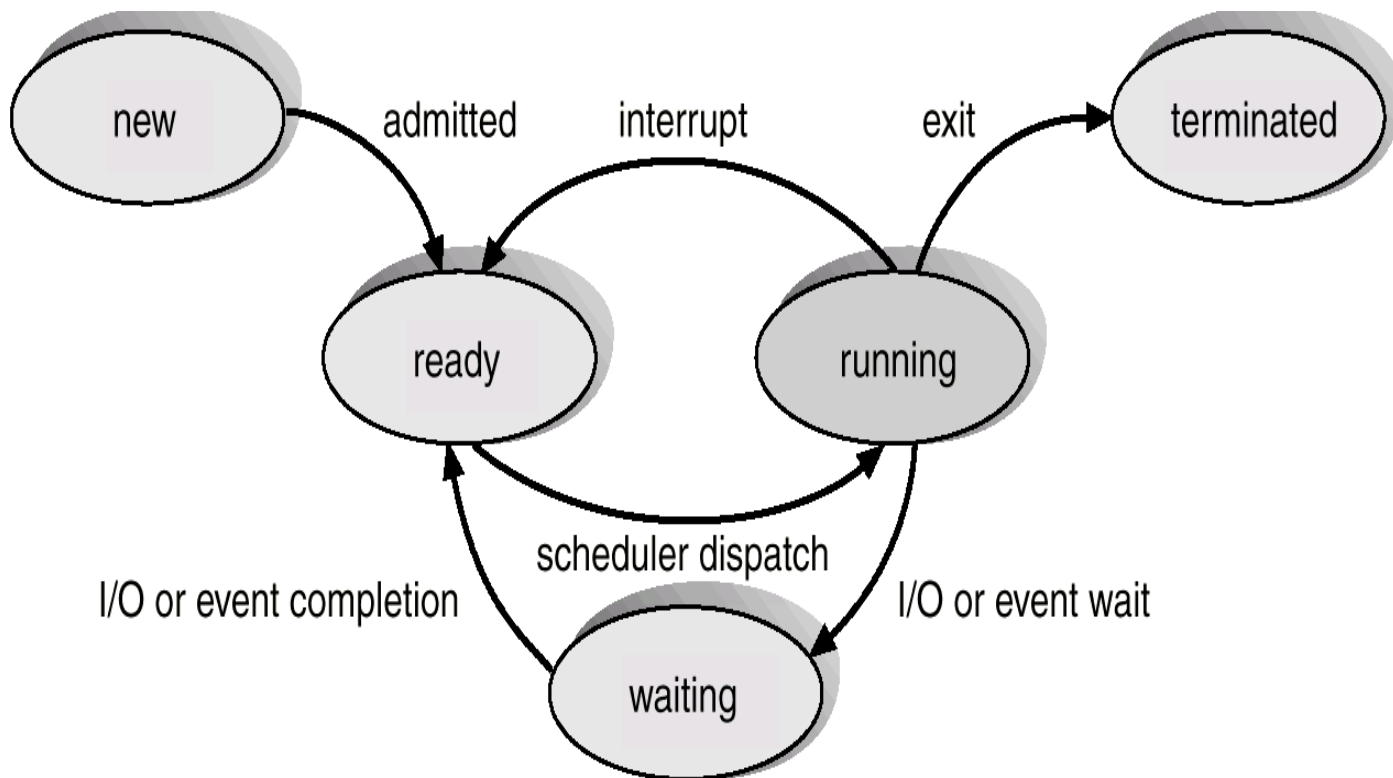
进程的状态（2）

- ready-就绪：进程已获得了除处理机以外的所有资源，等待分配处理机执行的状态。
- running-运行/执行：当一个进程获得必要的资源并正在处理机上执行的状态。
- waiting-等待/阻塞：正在执行的进程由于发生某事件而暂时无法执行下去，此时进程所处的状态。

进程的状态（3）

进程在运行期间并非固定处于某个状态，而是不断从一个状态转换到另一个状态。

2、进程状态转换





进程的状态（4）

3、进程的挂起状态

在某些系统中，为了更好地管理和调度进程，引入了挂起状态：

- 挂起状态/静止状态：

程序在运行期间，由于某种需要，往往要将进程暂停执行，使其静止下来，以满足某些需要。这种静止状态就称为进程的挂起状态。

- 挂起原语Suspend和激活原语Active

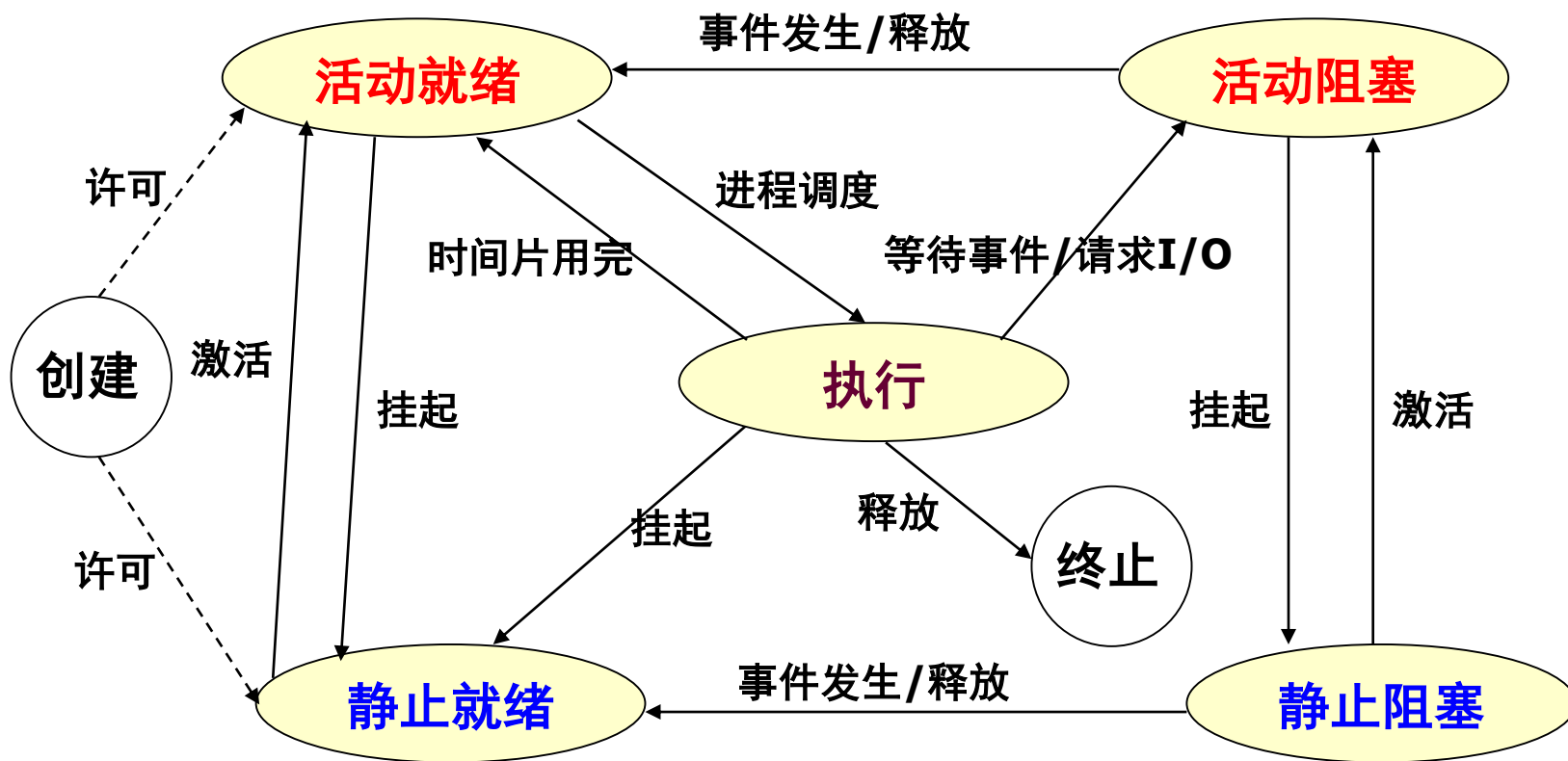


引入挂起状态的原因

- **终端用户的需要：** 终端用户在自己程序运行中发现问题要求使正在执行的进程暂停执行而使进程处于挂起状态。
- **父进程的需要：** 父进程为了考查和修改某个子进程，或者协调各子进程间的活动，需要将该子进程挂起。
- **操作系统的需要：** 操作系统为了检查运行中的资源使用情况或进行记帐，而将某些进程挂起。
- **对换的需要：** 为了提高内存的利用率，将内存中某些进程挂起，以调进其它程序运行。
- **负荷调节的需要：** 由于工作负荷较重，而将一些不重要的进程挂起，以保证系统能正常运行（实时操作系统）

具有挂起状态的进程状态转换图：

在引入挂起状态后，就增加了挂起状态（静止状态）与非挂起状态（活动状态）间的转换，如图所示：





进程控制块Process Control Block

进程控制块PCB

是操作系统为了管理和控制进程的运行而为每一个进程定义的一个记录型数据结构，它记录了系统管理进程所需的全部信息。系统根据PCB而感知进程的存在，PCB是进程存在的唯一标志。例：[Linux中的PCB](#)

```
grep -r "task_struct { " /usr/src/
```

1、进程控制块PCB的作用

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。具体作用如下：



进程控制块Process Control Block

(1) 作为独立运行基本单位的标志

系统是通过PCB感知进程的存在的，PCB是进程存在的唯一标志。

(2) 能实现间断性运行方式

当进程因阻塞暂停运行时，CPU的现场信息保存在该进程的PCB中

(3) 提供进程管理所需要的信息

(4) 提供进程调度所需要的信息

(5) 实现与其他进程的同步和通信



2、进程控制块PCB中的信息（1）

根据操作系统的要求不同，PCB所包含信息有些不同，但通常包含以下信息：

① 进程标识符：

内部标识符（由系统创建进程时分配给进程的唯一标识号，通常为一整数，称为进程号，用于区分不同的进程）

外部标识符（由字母、数字组成，由用户访问该进程时使用）。



2、进程控制块PCB中的信息（2）

② 处理机状态（断点信息）：

即处理机中各种寄存器（通用寄存器、PC、PSW等）的内容

③ 进程调度：

记录了进程调度的相关信息（状态、优先级、事件等）。

④ 进程控制：

记录了系统对进程控制的信息（程序和数据地址、同步机制、资源清单、链接指针）



3、进程控制块PCB的组织方式

在一个系统中，通常存在着许多进程，它们所处的状态不同，为了方便进程的调度和管理，需要将各进程的PCB用适当方法组织起来。目前常用的组织方式有：

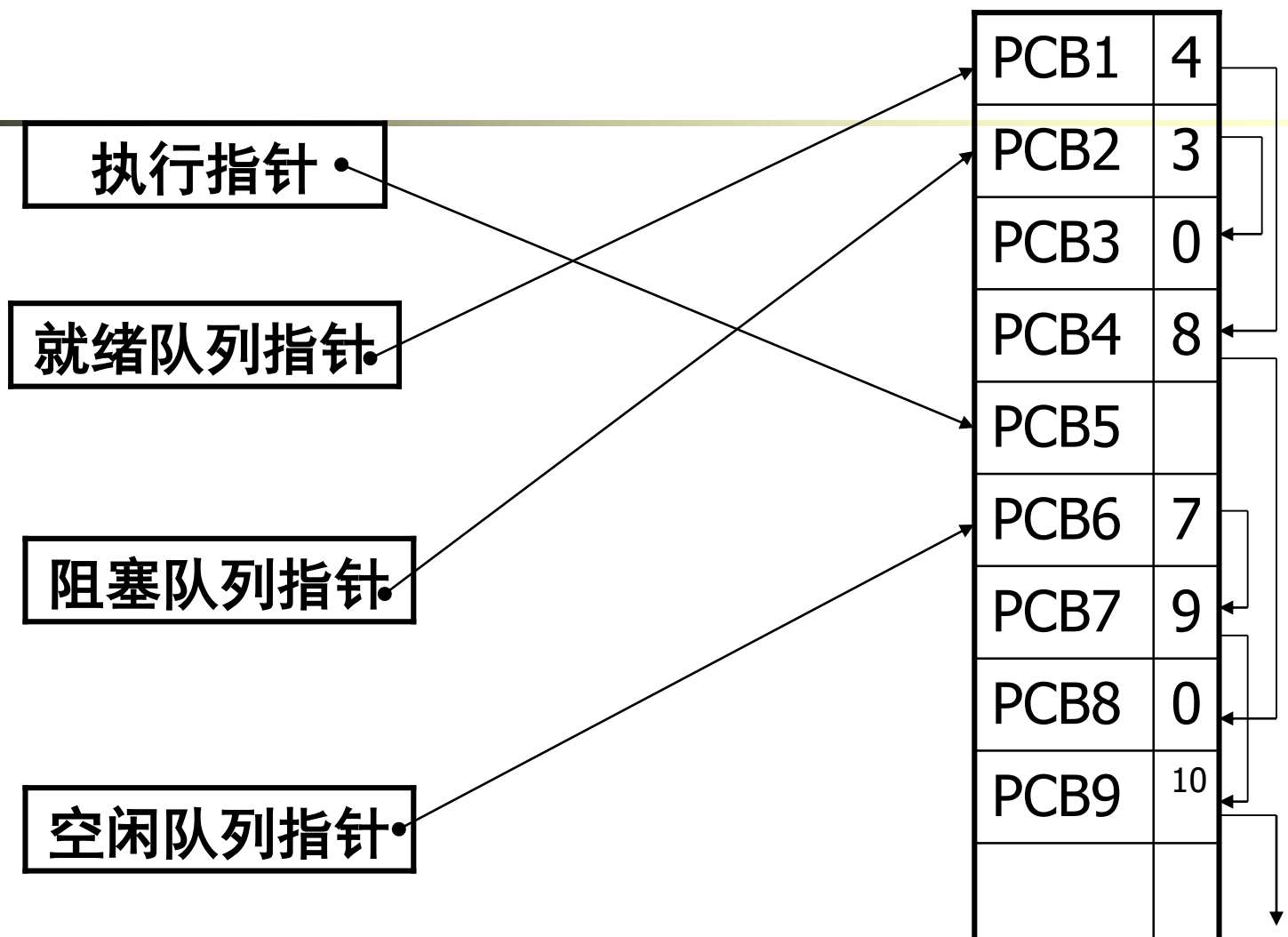
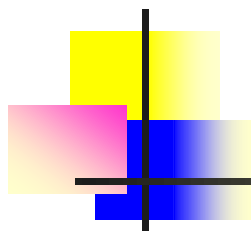
(1) 线性方式 线性表

(2) 链接方式 图示

把同一状态的PCB链接成一个队列，这样就形成了就绪队列、阻塞队列等。

(3) 索引方式 图示

将同一状态的进程组织在一个索引表中，索引表的表项指向相应的PCB，不同状态对应不同的索引表。



按链接方式组织PCB



执行指针 •

就绪表指针 •

阻塞表指针 •

就绪索引表



阻塞索引表



PCB1
PCB2
PCB3
PCB4
PCB5
PCB6
PCB7
PCB8
PCB9

按索引方式组织PCB



2.2 进程的控制

进程控制是进程管理中最基本的功能，即对系统中所有的进程实施有效的管理，其功能包括

进程的创建

进程的撤消

进程的阻塞与唤醒

进程的挂起与激活等，

这些功能一般是由操作系统的内核来完成。



OS 内核:

- 在现代OS中，常把一些功能模块（与硬件紧密相关的、常用设备的驱动程序及运行频率较高的）放在紧靠硬件的软件层次中，加以特殊保护，同时把它们常驻内存，以提高OS的运行效率，这部分功能模块就称OS的内核。
- 内核是基于硬件的第一层软件扩充，它为系统控制和管理进程提供了良好的环境。
- 最核心功能的集合



OS 内核的主要功能：

- 支撑功能：中断处理、时钟管理、原语操作
- 进程管理：创建和终止、调度和分派、进程切换、同步和通信、对PCB的管理
- 存储器管理：内存分配和回收、内存保护和交换、分页和分段管理
- 设备管理：缓冲区管理、I/O设备的分配、设备驱动、设备独立性功能模块



处理机的执行状态

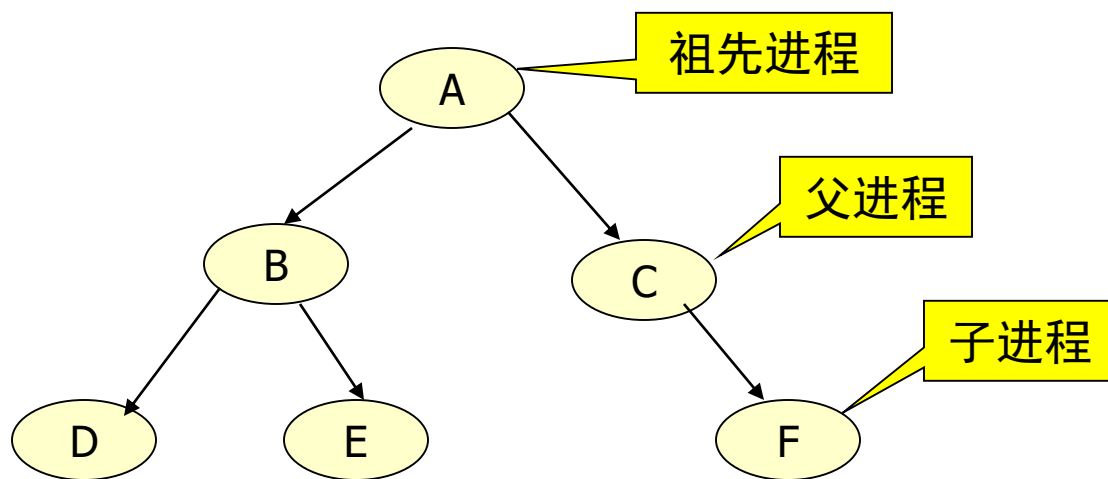
为了防止OS及其关键数据（如PCB等）不被用户有意或无意破坏, 通常将处理机的执行状态分为两种:

处理机状态	特权(执行指令, 访问)	程序
系统态(核心态、管态)	较高(一切指令, 所有寄存器及存储区)	OS内核
用户态(目态)	较低(规定指令, 指定寄存器及存储区)	用户程序或系统外层的应用程序

一、进程创建

一个进程可以创建若干个新进程，新创建的进程又可以创建子进程，为了描述进程之间的创建关系，引入了进程图（如下图所示：）

1、进程图：又称为进程树或进程家族树，是描述进程家族关系的一棵有向树。





2、引起进程创建的事件（1）

在多道程序环境中，只有进程才可以在系统中运行。为了使一个程序能运行，必须为它创建进程。导致进程创建的事件有：

① 系统初始化：

启动操作系统时，会创建若干进程，包括一些前台进程和后台进程（具有专门的功能，如接收发来的邮件、接收Web请求等）

② 用户登录：

在分时OS中，用户在终端键入登录命令后，如是合法用户，则系统为该终端创建一进程，并插入就绪队列。

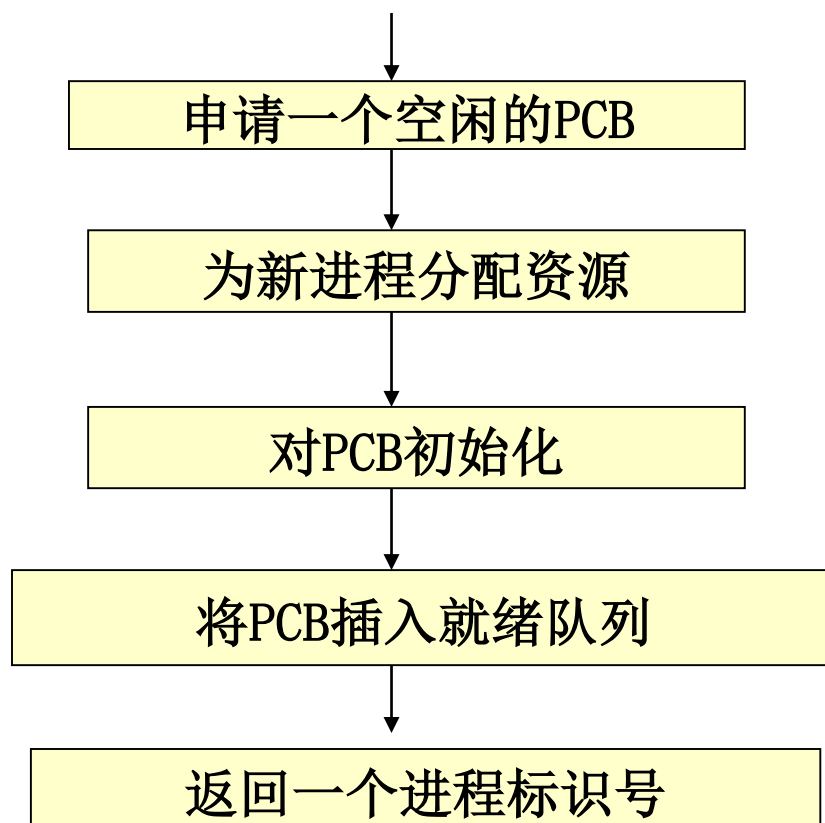


2、引起进程创建的事件（2）

- ③ **作业调度**：在批处理OS中，当按某算法调度一作业进内存，系统为之分配必要资源，同时为该作业创建一进程，并插入就绪队列。
- ④ **提供服务**：在程序运行中，若用户需某种服务，则系统创建一进程为用户提供服务，并插入就绪队列。
- ⑤ **应用请求**：在运行中，由于应用进程本身的需求，自己创建一进程，并插入就绪队列。

3、进程的创建

操作系统一旦发现了要求创建进程的事件后，便调用进程创建原语create()按以下过程创建一新进程：





二、进程的撤消

一个进程在完成其任务后，应加以撤消，以便及时释放其占有的各类资源。

1、导致进程撤消的事件

◆ 进程正常结束

执行一个OS调用表示进程运行完毕（如：批处理系统的Halt指令，UNIX中的exit，Windows中相关调用是ExitProcess等）

◆ 进程异常结束

如运行超时、内存不足、越界错误、保护错误、算术错误、等待超时、I/O故障、非法指令、特权指令错等。



二、进程的撤消

1、导致进程撤消的事件（续）

◆外界干预

操作员或操作系统干预；

父进程终止；

父进程请求（父进程拥有终止所有子孙进程的权限）

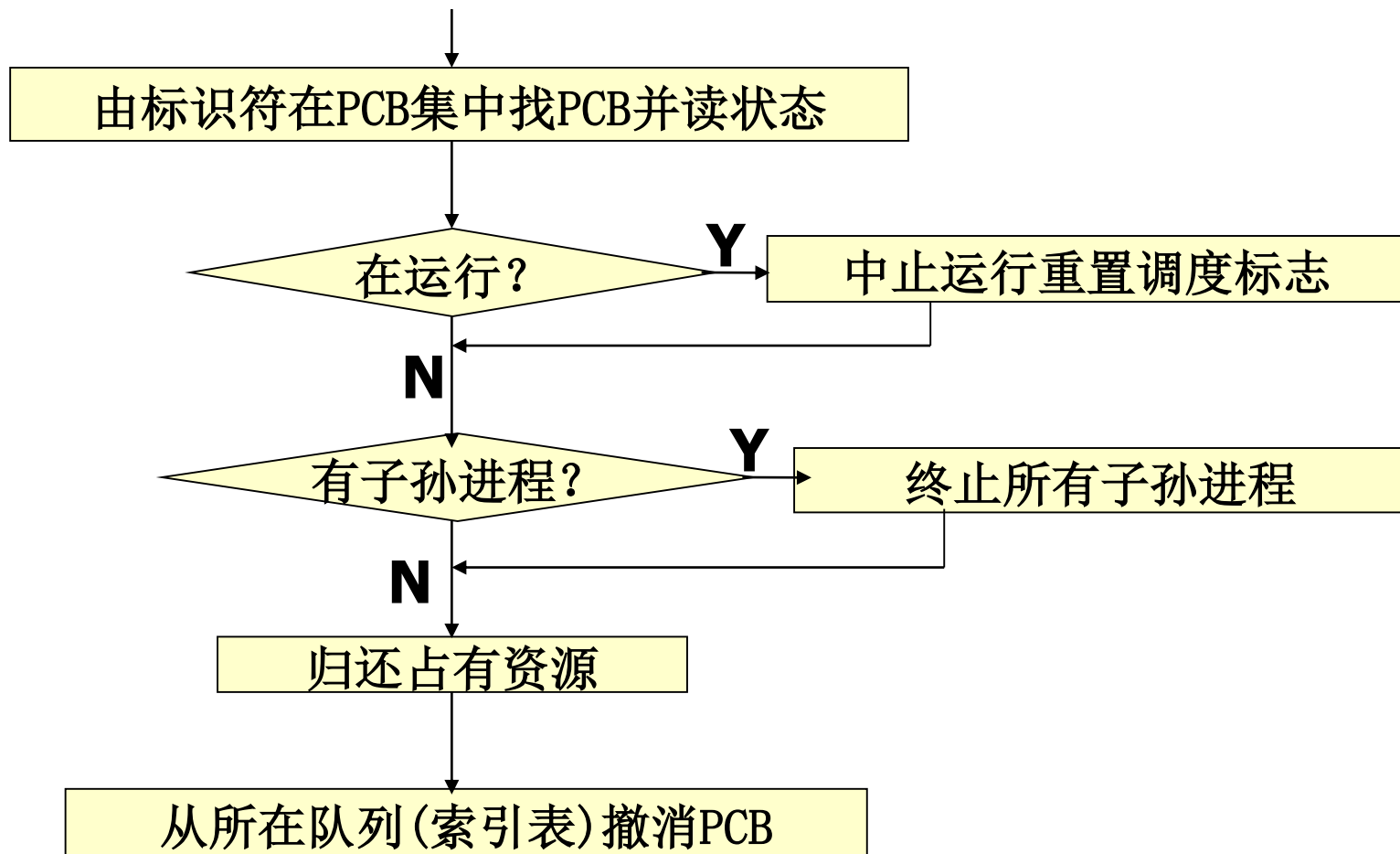
如果系统中发生了要求撤消进程的事件，OS便调用撤消原语去撤消进程。

2、撤消原语可采用2种撤消策略

◆只撤消指定的进程（指定进程撤消后，会将其子孙进程交给系统进程管理）

◆撤消指定进程及其所有的子孙进程

3、进程撤消的过程





三、进程的阻塞与唤醒

当一个进程期待的事件还没有出现时，该进程调用阻塞原语**block()**将自己阻塞起来；

block() 功能：将进程由执行状态转变为阻塞状态。

对于处于阻塞状态的进程，当该进程期待的事件出现时，由其它相关进程调用唤醒原语**wakeup()**将阻塞的进程唤醒，使其进入就绪状态；

wakeup() 功能：将进程由阻塞状态转变为就绪状态。



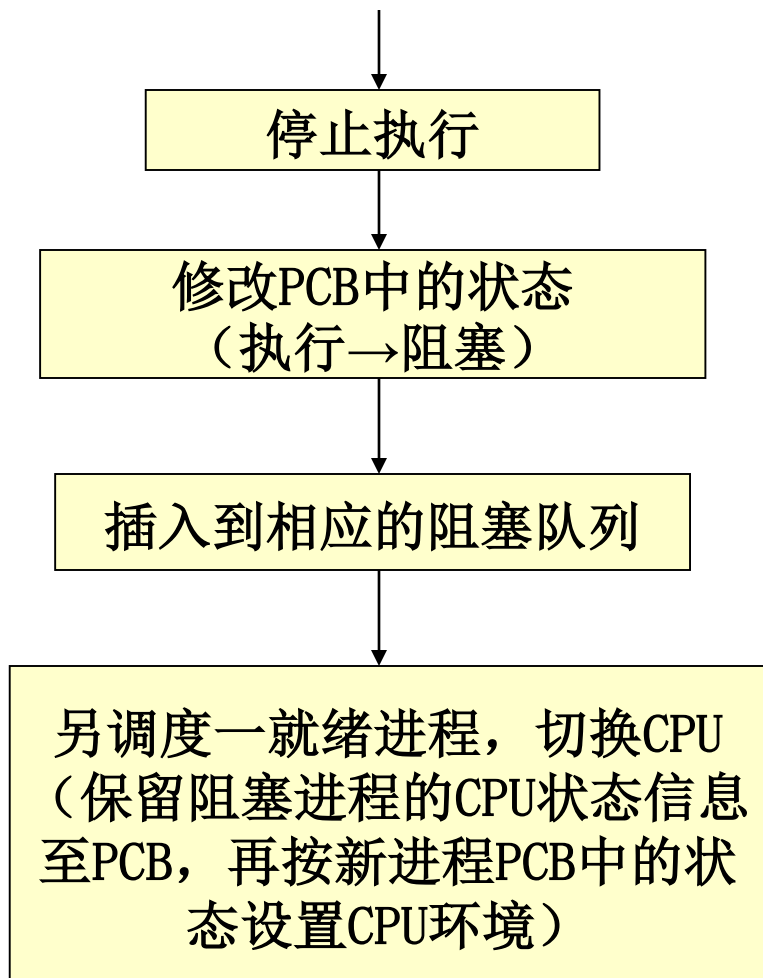
三、进程的阻塞与唤醒

1、引起进程阻塞和唤醒的事件

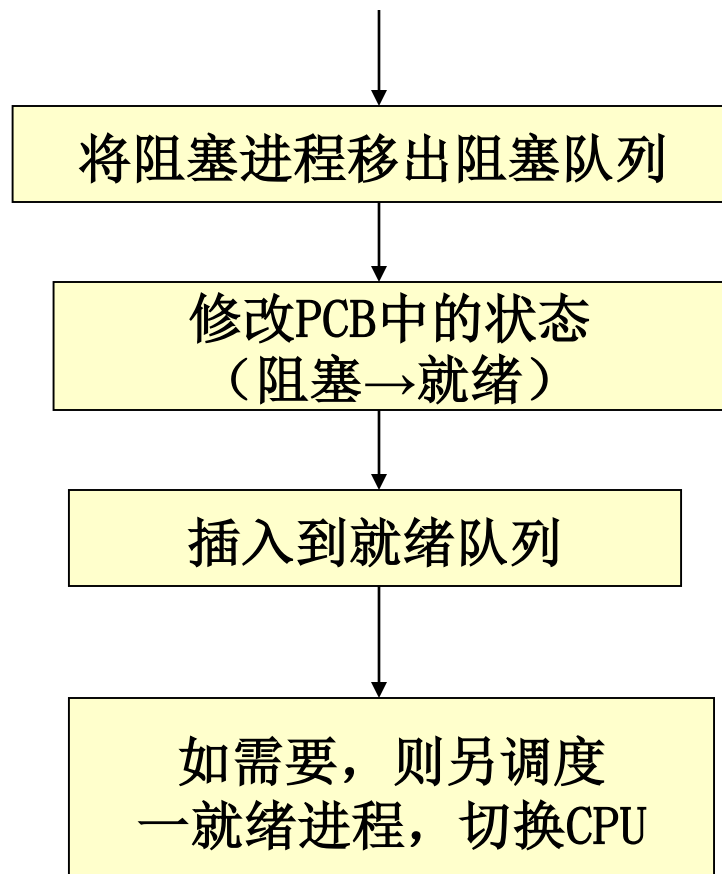
- 请求系统服务
- 启动某种操作
- 新数据尚未到达
- 无新工作可做



2、进程的阻塞过程



3、进程的唤醒过程



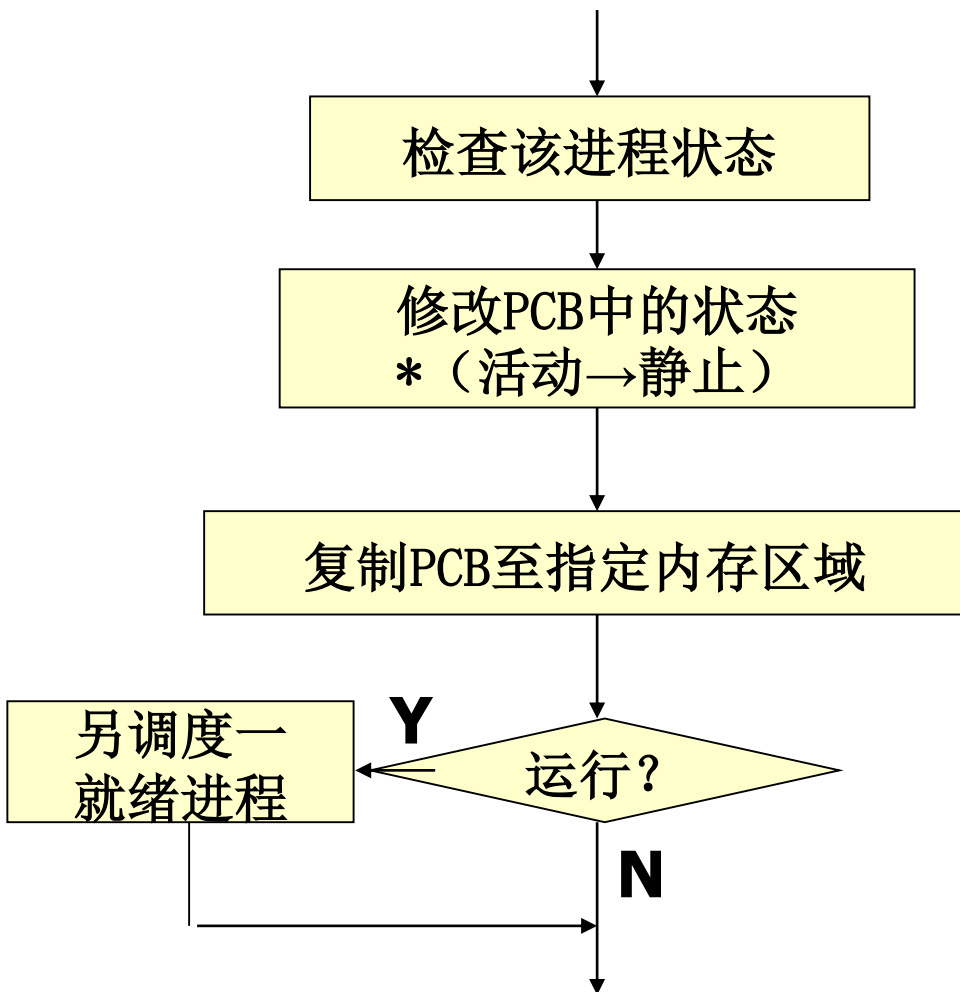


四、进程的挂起与激活

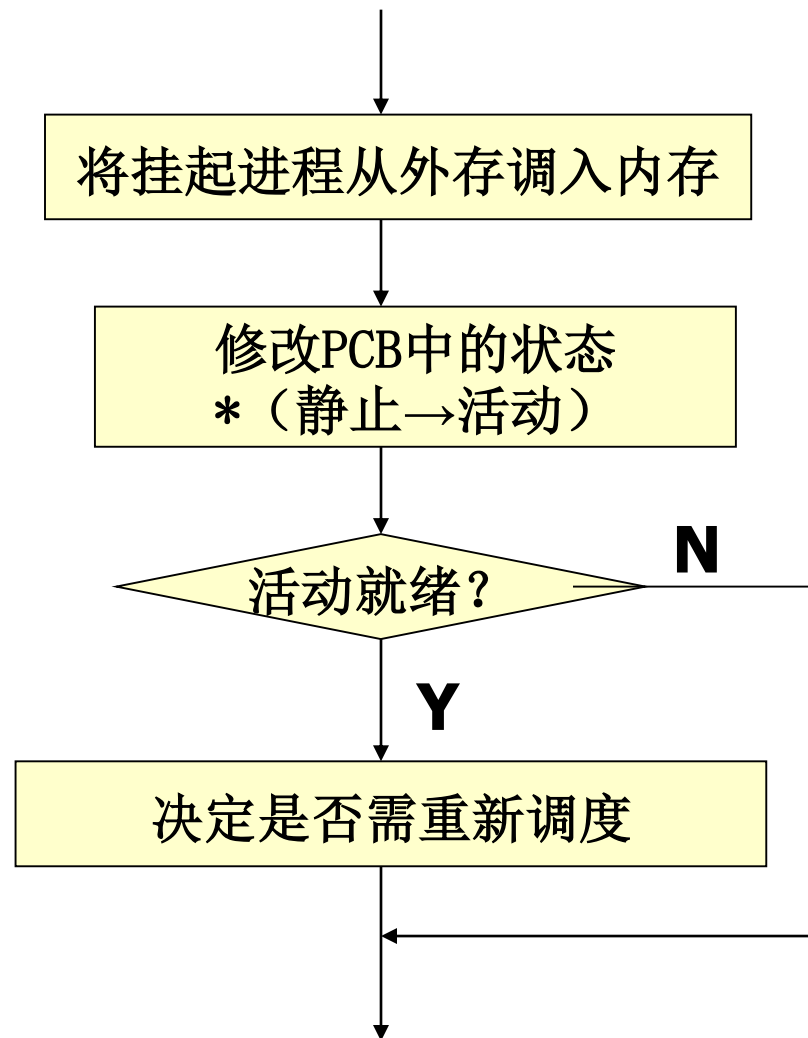
当引起进程挂起的事件发生时，系统就将利用挂起原语suspend()将指定进程或处于阻塞状态的进程挂起。当发生激活进程的事件时，系统就将利用激活原语active()将指定进程激活。

1、引起进程的挂起与激活的事件

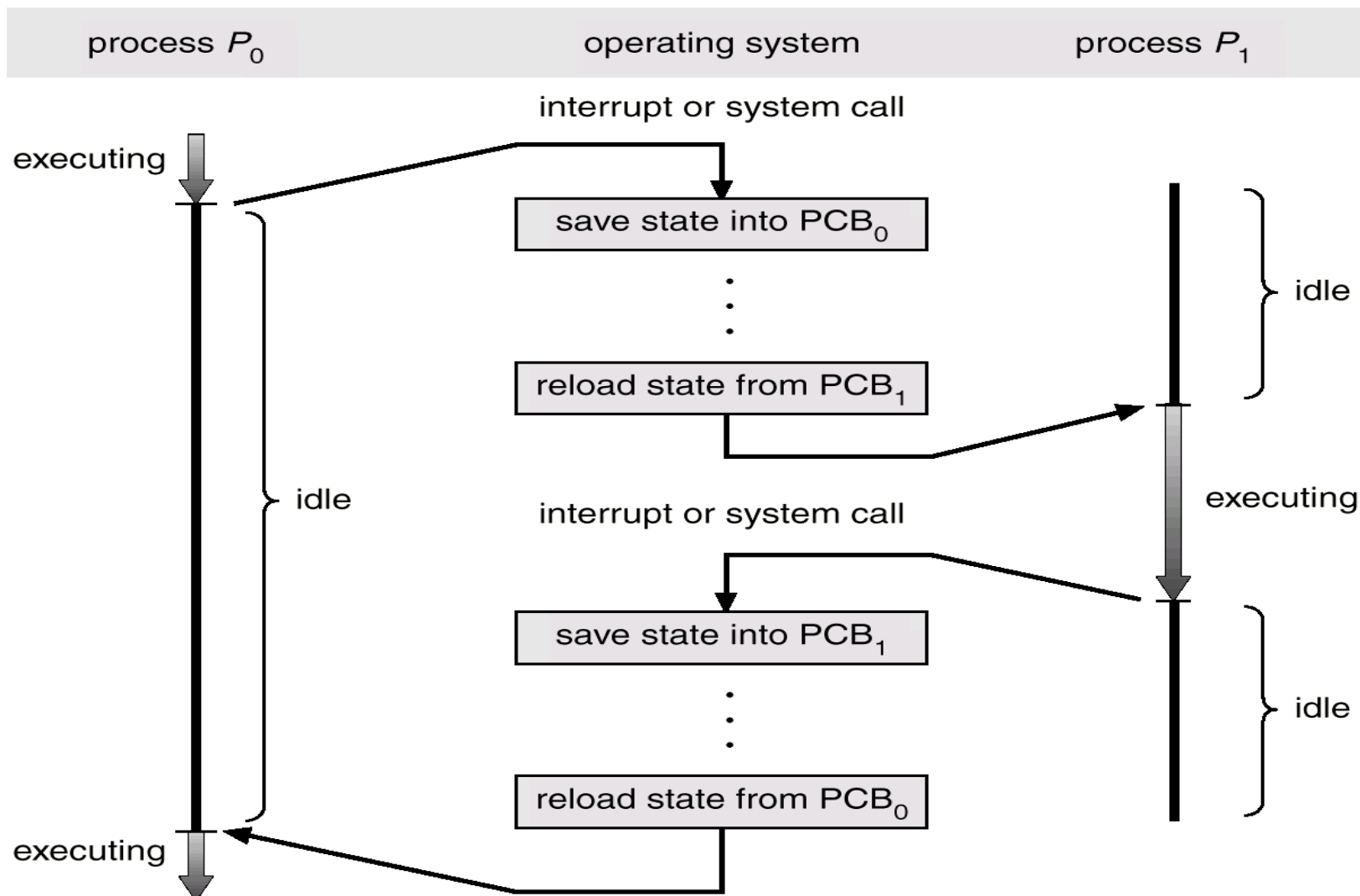
2、进程的挂起过程



3、进程的激活过程



进程间CPU的切换/上下文切换





2.3 进程同步 (1)

- OS设计的核心是进程管理， 主要涉及三个方面：
 - 单处理机中多个进程的管理
 - 多处理机中多个进程的管理
 - 分布式计算机系统中多个进程的管理
- 上述三个方面共同的基础是并发
 - 进程间资源共享、进程间竞争、进程间通信
 - 处理机时间的分配
- 为了使多道程序系统中， 多个进程能够正确地运行， 必须引入同步机制



2.3 进程同步 (2)

- **进程同步**：指对多个相关进程在执行次序上进行协调；
- 同步的**任务**：使系统中各进程之间能有效地**共享资源**和**相互合作**，从而使程序的执行具有可再现性；
- 系统中各进程之间在逻辑上的相互制约的关系：
 - 直接关系—同步
 - 间接关系—互斥



2.3 进程同步 (3)

- 用来实现同步的机制称为**同步机制**。如：
 硬件同步机制；
 信号量机制；
 管程机制



2.3 进程同步 (4)

■ 进程同步的基本概念

- 两种形式的制约关系
- 临界资源、临界区
- 同步机制应遵循的规则

■ 信号量机制

- 整型信号量
- 记录型信号量
- AND型信号量集、一般信号量集

■ 信号量的应用

- 信号量实现进程互斥
- 信号量描述进程间的前趋关系

■ 管程机制



一、进程同步的基本概念

1、进程之间两种形式的制约关系

系统中各进程之间在逻辑上存在着两种制约关系：

- **直接相互制约关系/相互合作关系（进程同步）：**

即为完成同一个任务的各进程之间，因需要协调它们的工作而相互等待、相互交换信息所产生的直接制约关系。

- **间接相互制约关系/资源共享关系（进程互斥）：**

是进程共享独占型资源而必须互斥执行的间接制约关系。



■ 同步与互斥比较

同 步（直接制约）	互 斥（间接制约）
进程-进程	进程-资源-进程
时间次序上受到某种限制	竞争不到某一物理资源时不允许进程工作
相互清楚对方的存在及作用，交换信息	不一定清楚其它进程的情况
往往指有几个进程共同完成一个任务	往往指多个任务多个进程间通讯制约
例：生产与消费之间，发送与接收之间，写者与读者之间	例：交通十字路口，单轨火车的拨道岔



2、临界资源、临界区（1）

- 一次只允许一个进程使用的共享资源称为临界资源，如打印机，绘图机，变量，数据等，各进程间采取互斥方式实现对这种临界资源的共享，从而实现并发进程的封闭性。

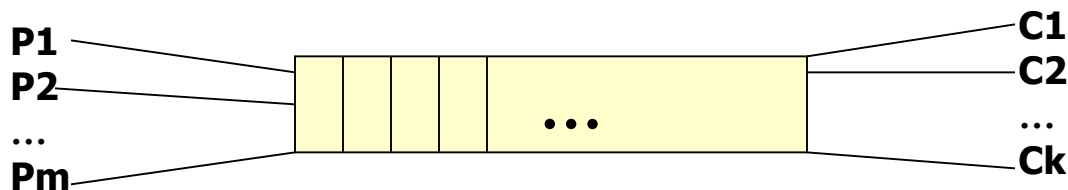
例：生产者-消费者问题

一组生产者向一组消费者提供产品，它们共享一个缓冲池，生产者向其中投放产品，消费者从中取得产品。它是许多相互合作进程的抽象，如输入进程与计算进程；计算进程与打印进程等。

2、临界资源、临界区（2）

设缓冲池的长度为 n ($n > 0$)，一群生产者进程 P_1, P_2, \dots, P_m ，一群消费者进程 C_1, C_2, \dots, C_k ，如图所示。假定生产者和消费者是相互等效，只要缓冲池未满，生产者就可以把产品送入缓冲区，类似地，只要缓冲池未空，消费者便可以从缓冲区取走产品并消耗它。

生产者和消费者进程是以异步方式运行的，但必须保持同步关系，即禁止生产者向满的缓冲池输送产品，也禁止消费者从空的缓冲池提取产品。





2、临界资源、临界区 (3)

- 生产者消费者进程共享如下变量：

TYPE item=...;

VAR n:integer;

buffer:array[0..n-1] of item;

in, out:0..n-1;

counter:0..n;

注：n为缓冲池中缓冲区的个数；

buffer为具有n个缓冲区的缓冲池；

in和out为指针，分别指向下一个可投放产品的缓冲区和下一个可获取产品的缓冲区



2、临界资源、临界区（4）

- 生产者消费者进程可分别描述为：

producer:

repeat

.....

produce an item in nextp;

.....

while counter=n do no-op;

buffer[in]:=nextp;

in:=(in+1) mod n;

counter:=counter+1;

until false;

consumer:

repeat

while counter=0 do no-op;

nextc:=buffer[out];

out:=(out+1) mod n;

counter:=counter-1;

consume the item in nextc;

until false;



2、临界资源、临界区（5）

- 单独看生产者进程和消费者进程，或者顺序执行生产者和消费者进程，都是正确的；但是并发执行时会出现差错。

问题出在两进程共享的变量counter上：

用机器语言实现变量的加1、减1的操作为：

变量counter加1：

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

变量counter减1：

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



2、临界资源、临界区（6）

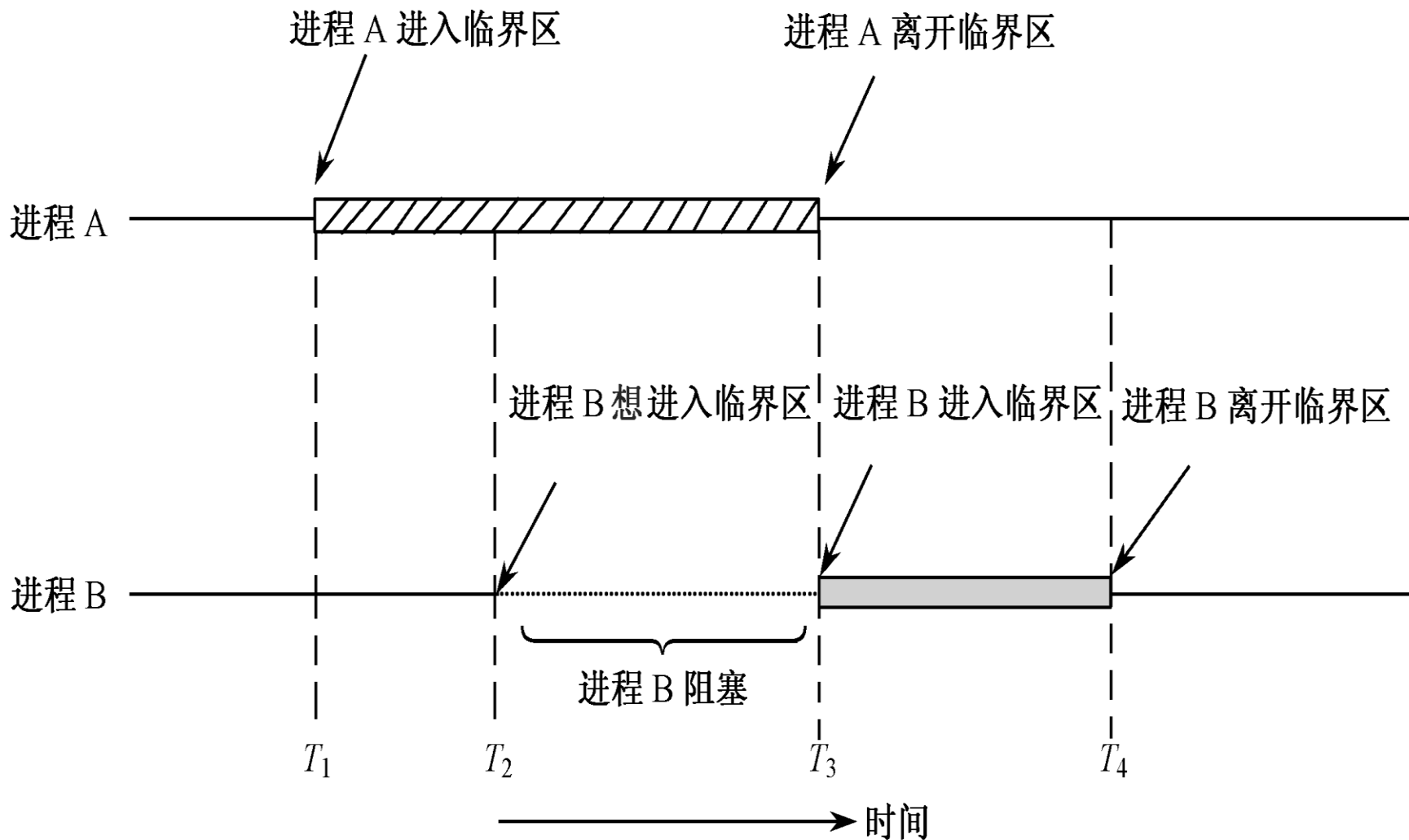
若按另一顺序对变量进行修改，会得到什么结果？

```
register1:=counter;  
register1:=register1+1;  
register2:=counter;  
register2:=register2-1;  
counter:=register1;  
counter:=register2;
```

（1）变量counter必须按临界资源处理。

（2）每个进程中访问临界资源的那段代码称为临界区

2、临界资源、临界区（7）





2、临界资源、临界区（8）

为了保证临界资源的正确使用，可以把一个访问临界资源的循环进程描述如下：

Repeat

Entry Section

Critical Section

Exit Section

Remainder Section

Until False

或

Do {

Entry Section;

Critical Section;

Exit Section;

Remainder Section;

} While(1);



2、临界资源、临界区（9）

❖ 进入区(Entry Section)

增加在临界区前面的一段代码，用于检查所要访问的临界资源此刻是否被访问。

❖ 退出区(Exit Section)

增加在临界区后面的一段代码，用于将临界资源的访问标志恢复为未被访问标志。

❖ 剩余区(Remainder Section)

进程中除了进入区、临界区及退出区之外的其余代码。

2、临界资源、临界区（10）

进入区

临界区

退出区

剩余区

要进入临界区的若干进程必须满足：

- (1) 一次只允许一个进程进入临界区
- (2) 任何时候，处于临界区的进程不得多于一个
- (3) 进入临界区的进程要在有限的时间内退出
- (4) 如果不能进入自己的临界区，则应让出处理机资源

解决临界区（互斥）问题的几类方法：

- (1) 硬件的方法
- (2) P-V操作
- (3) 管程机制

同步机制



3、同步机制应遵循的规则

- **空闲让进**: 当无进程处于临界区时, 表明临界资源处于空闲状态, 应允许一个请求进入临界区的进程立即进入自己的临界区, 以有效地利用临界资源。
- **忙则等待**: 当已有进程进入临界区时, 表明临界资源正在被访问, 因而其他试图进入临界区的进程必须等待, 以保证对临界资源的互斥访问。
- **有限等待**: 对要求访问临界资源的进程, 应保证在有限时间内能进入自己的临界区, 以免陷入“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”。



二、信号量机制

- 信号量机制是荷兰科学家E. W. Dijkstra在1965年提出的一种同步机制，也称为P、V操作。由最初的整型信号量发展为记录型信号量，进而发展为信号量集机制。
 - 整型信号量
 - 记录型信号量
 - AND型信号量
 - 一般信号量集



1、整型信号量

- **整型信号量**——非负整数，除了初始化（即赋初值：必须置一次且只能置一次初值，初值不能为负数）外，只能通过两个原子操作wait和signal（或者叫P、V操作）来访问（即信号量的值仅能由这两个原子操作来改变）。

- **wait和signal操作描述：**

wait(S) : while $S \leq 0$ do no-op; 测试有无可用资源
 $S := S - 1$; 可用资源数减一个单位

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

主要问题：只要 $S \leq 0$ ，wait操作就不断地测试（忙等），因而，未做到“**让权等待**”。



2、记录型信号量（1）

- 基本思想

- 1、设置一个代表资源数目的整型变量
value（资源信号量）

- 2、设置一链表L用于链接所有等待访问同一资源的等待进程

- 记录型信号量的数据结构

```
Type semaphore=record
```

```
    value: integer;
```

```
    L: list of process;
```

```
end
```

```
Var    S: semaphore;
```



2、记录型信号量（2）

- wait和signal操作描述:

wait(S) :

S.value:=S.value-1;

if S.value<0 then

block(S.L); //调用阻塞原语将该进程状态置为
//等待状态, 将该进程的PCB插入

signal(S) :

//相应的等待队列S.L末尾;

S.value:=S.value+1;

if S.value≤0 then

wakeup(S.L); //调用唤醒原语唤醒相应等待队列

做到“让权等待”。 //S.L中等待的一个进程, 改变其状
//态为就绪态, 并将其插入就绪队列



2、记录型信号量（3）

- 在记录型信号量机制中
 - S. value的初值表示系统中某类资源的数目。
 - S. value的初值为1，S为互斥信号量；
 - S. value的初值大于1，S为资源信号量；
 - S. value的初值为0：用来描述进程之间的前趋关系



2、记录型信号量（3）

- 在记录型信号量机制中
 - 每次wait(S)操作意味着进程请求一个单位的资源，资源数目减1。当 $S.value < 0$ 时表示资源已分配完毕，进程调用Block()原语进行自我阻塞，放弃处理机，并插入到信号量链表S.L中。此时， $|S.value|$ 表示在该信号量链表中已阻塞进程的数目。



2、记录型信号量（3）

- 在记录型信号量机制中
 - 每次signal(S)表示执行进程释放一个单位资源，资源数目加1。若加1后仍有 $S.value \leq 0$ ，则表示在该信号量链表中，还有等待该资源的进程，则调用Wakeup()原语，将链表中第一个进程唤醒。



3、AND型信号量（了解）（1）

■ AND型信号量的引入

- 有些进程是需要获得两类或两类以上的共享资源后，才可以执行任务
- “死锁”

Process A:
Wait (Dmutex);
Wait (Emutex);

Process B:
Wait (Emutex);
Wait (Dmutex);



3、AND型信号量（了解）（2）

■ AND型信号量的基本思想

将进程在整个运行过程中所需要的**所有临界资源**，**一次性全部分配**给进程，待进程使用完后再一起释放。只要有一个资源未能分配给进程，其它所有可能分配的资源也不分配给该进程。从而可避免死锁发生。在wait操作中，增加了一个“AND”条件，故称为AND同步。



3、AND型信号量（了解）（3）

- 因此在“wait”操作中，增加了一个“AND”条件，即
Swait

Swait(S1,S2,S3,.....Sn)

if $S1 \geq 1$ **and ...and** $S_n \geq 1$ then

for $i:=1$ to n do

$S_i := S_i - 1;$

endfor

else

place the process in the waiting queue associated with the first S_i found with $S_i < 1$,
and set the program count of this process to the beginning of Swait operation

endif

Ssignal (S1,S2,.....Sn)

for $i:=1$ to n do

$S_i := S_i + 1;$

Remove all the process waiting in the queue associated with S_i into the ready queue

endfor;



4、一般信号量集（了解）（1）

■ 一般信号量集的基本思想

一次可分配多个某类临界资源，而不需执行N次的P操作；每次分配前都测试该类资源数目是否大于测试值。

- 当一次需要N个某类临界资源时，不需要进行N次wait(S)操作
- 在有些情况下，当资源数量低于某一下限值，不予以分配
- $\text{Swait}(S_1, t_1, d_1; S_2, t_2, d_2; \dots; S_n, t_n, d_n)$, S_i 为信号量, d_i 为需求值, t_i 为下限值（测试值）
- $\text{Signal}(S_1, d_1; S_2, d_2; \dots; S_n, d_n)$



4、一般信号量集（了解）（2）

■ 一般信号量集的几种特殊情况

- **Swait(S, d, d)**：信号量集中只有一个信号量，但允许它每次申请d个资源，当资源数少于d时，不予分配。
- **Swait(S, 1, 1)**：蜕化为一般的记录型信号量（ $S > 1$ 时）或互斥信号量（ $S = 1$ 时）。
- **Swait(S, 1, 0)**：很有用的信号量操作，相当于一个可控开关，当 $S \geq 1$ 时，开（允许多个进程进入）；当 $S = 0$ 时，关（阻止任何进程进入）。



三、信号量的应用（1）

■ 利用信号量实现进程互斥

利用信号量可以方便地解决临界区问题（进程互斥）（见后面的图）。为临界资源设置一互斥信号量lock，初值为1，则实现进程A、B、C互斥的描述：

A进程：

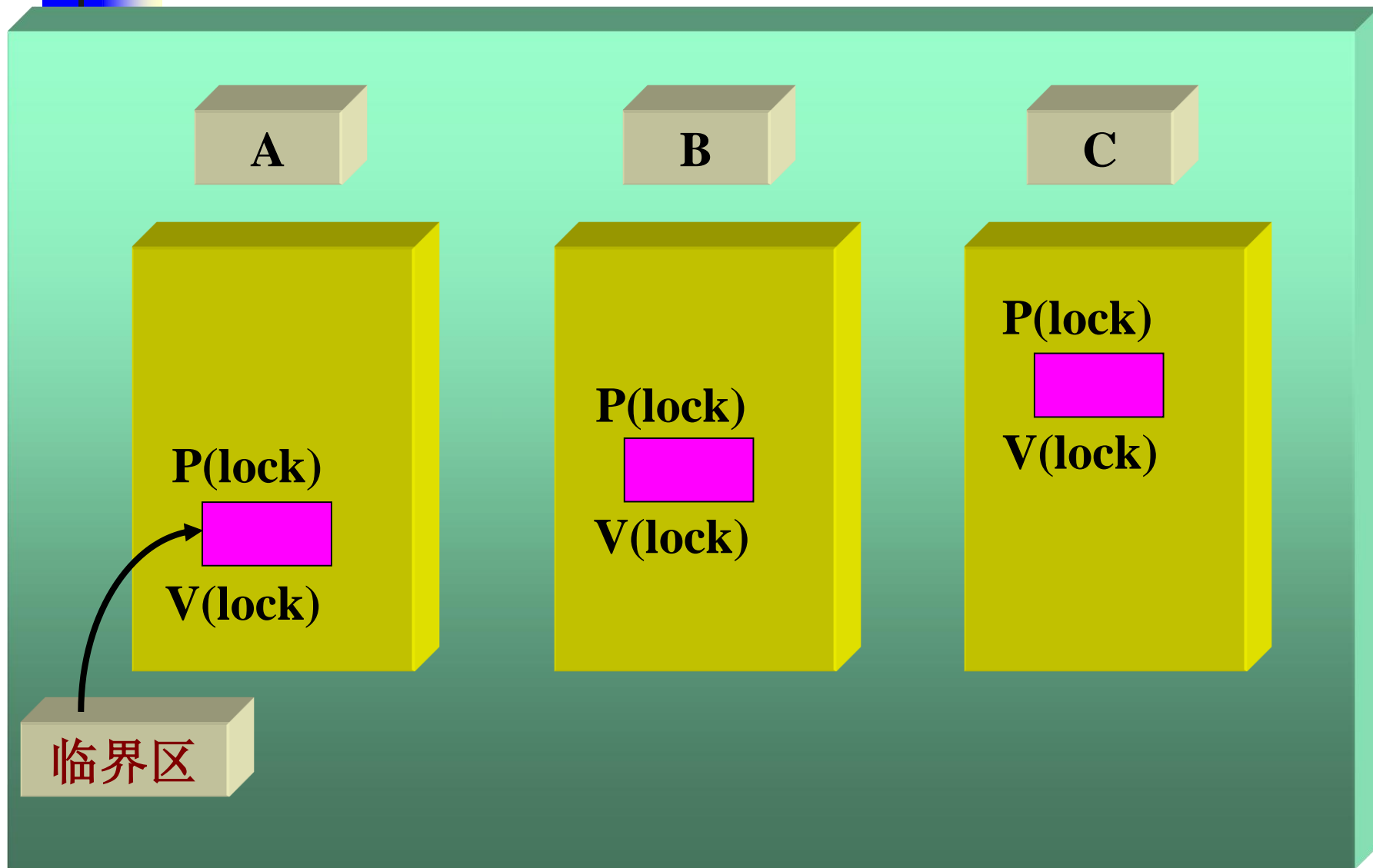
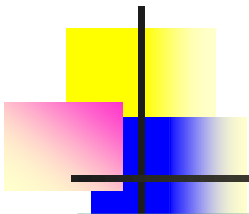
```
⋮  
wait(lock);  
critical section;  
signal(lock);  
⋮
```

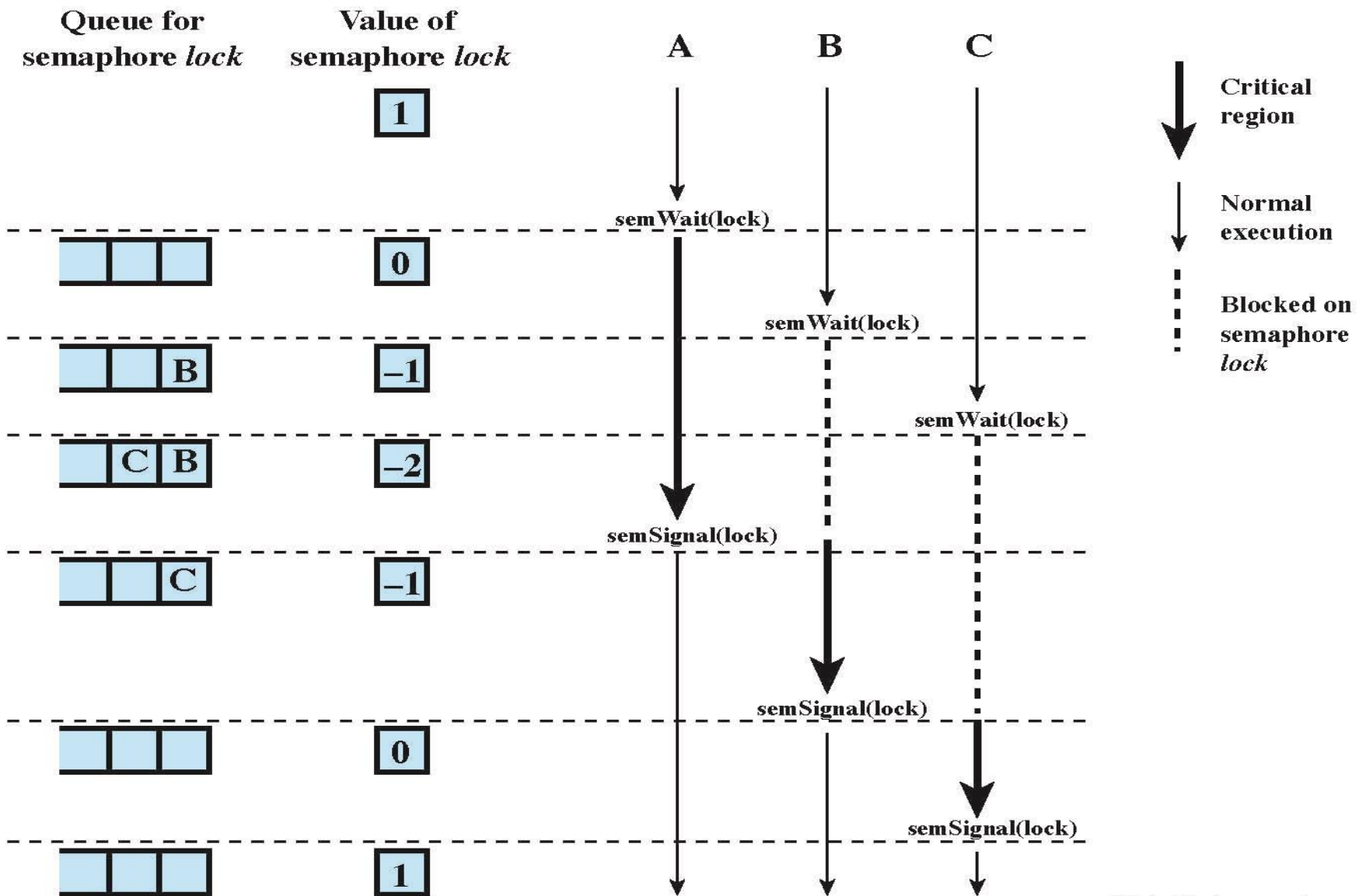
B进程：

```
⋮  
wait(lock);  
critical section;  
signal(lock);  
⋮
```

C进程：

```
⋮  
wait(lock);  
critical section;  
signal(lock);  
⋮
```





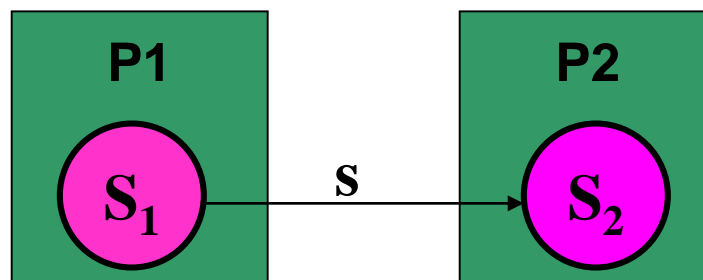
Note that normal execution can proceed in parallel but that critical regions are serialized.

三、信号量的应用（2）

- 利用信号量描述前趋关系（信号量可用来描述程序或语句间的前趋关系。）

方法：

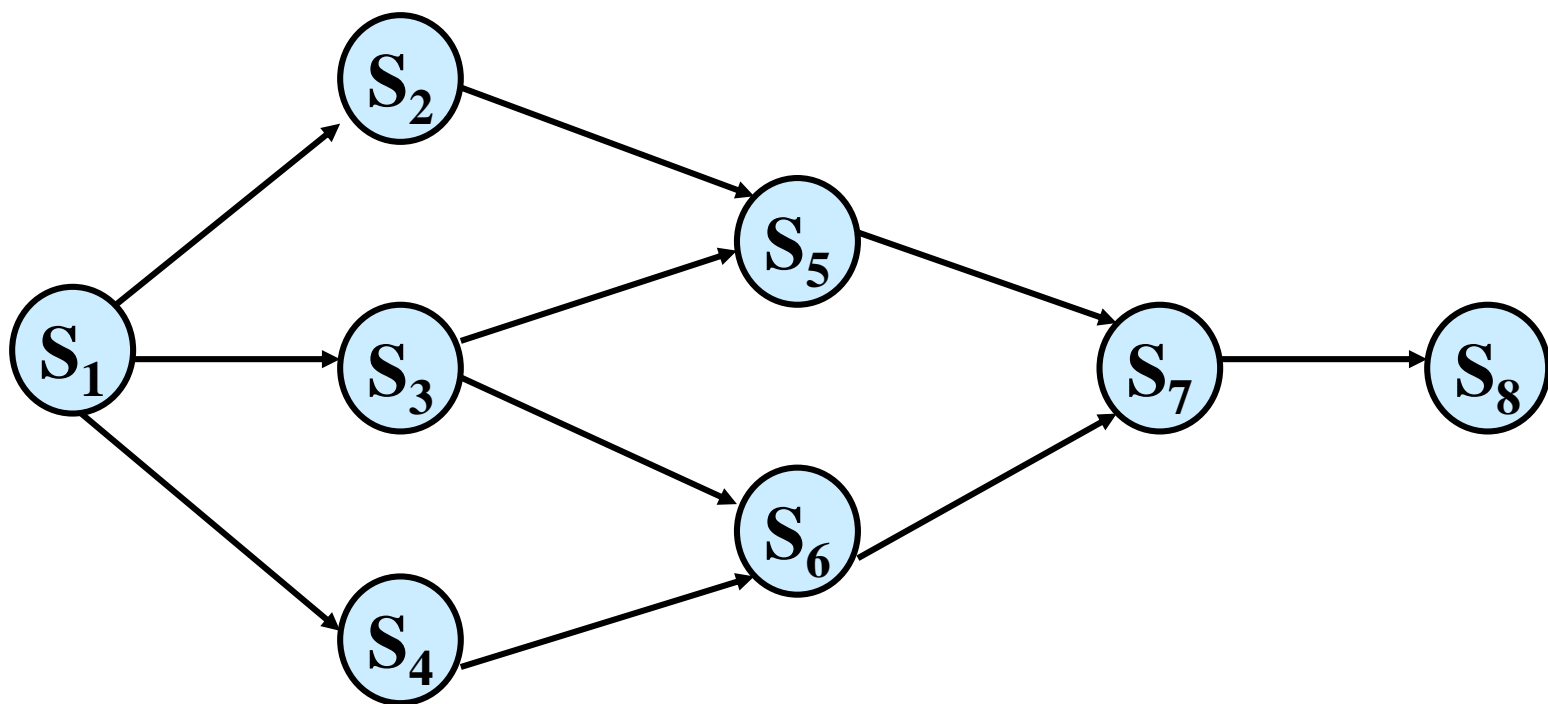
若图中存在结点 S_1 指向结点 S_2 的有向边，表示进程 P_1 中的程序段 S_1 应该先执行，而进程 P_2 中的程序段 S_2 后执行。设置一个信号量 s ，初值为0，将 $V(s)$ 放在 S_1 后面，而在 S_2 前面先执行 $P(s)$ 。



- 进程 P_1 的语句序列为： $S_1; V(s)$
- 进程 P_2 的语句序列为： $P(s); S_2$

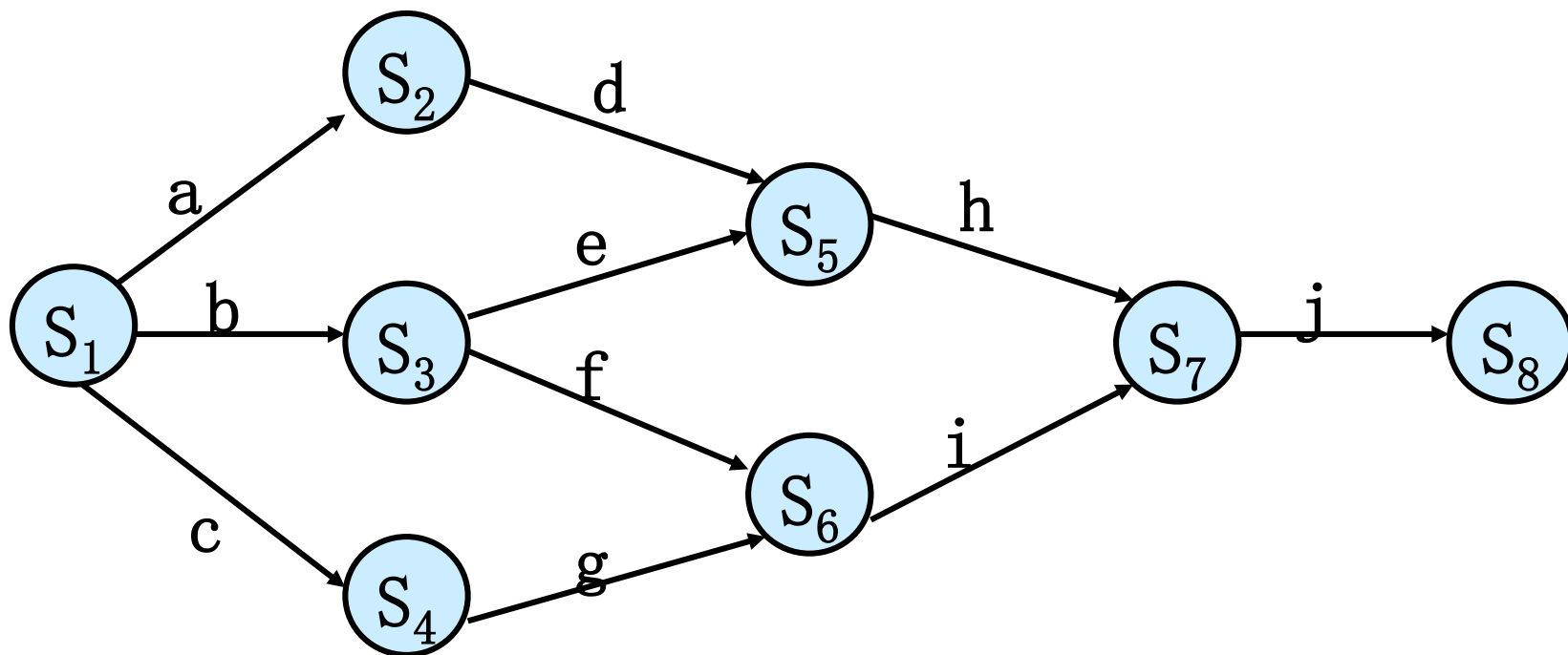
三、信号量的应用（3）

- 例 利用信号量来描述前趋图关系



三、信号量的应用（4）

- 具有8个结点的前趋图。前趋图中共有有向边10条，可设10个信号量，初值均为0；有8个结点，可设计成8个并发进程，具体描述如下：





三、信号量的应用 (5)

```
struct semaphore
    a, b, c, d, e, f, g, h, i, j=0, 0, 0, 0, 0, 0, 0, 0, 0, 0
cobegin
    {S1;V(a);V(b);V(c);}
    {P(a);S2;V(d);}
    {P(b);S3;V(e);V(f);}
    {P(c);S4;V(g);}
    {P(d);P(e);S5;V(h);}
    {P(f);P(g);S6;V(i)}
    {P(h);P(i);S7;V(j);}
    {P(j);S8;}
coend
```



四、管程机制（1）

■ 信号量机制实现进程同步的问题

用信号量机制实现进程间的同步和互斥，既方便又有效。但存在以下两个问题：

- ❖ 每个访问临界资源的进程都必须自备同步操作（P、V操作），**这使大量的同步操作分散在各个进程中**，给系统的管理带来麻烦。
- ❖ 会因同步操作使用不当而导致系统死锁。



四、管程机制（2）

■ 解决方法——管程 (Monitors)

Dijkstra于1971年提出：

为每个共享资源设立一个“秘书”来管理对它的访问。

一切来访者都要通过秘书，而秘书每次仅允许一个来访者（进程）来访问共享资源。

这样既便于系统管理共享资源，又能保证进程的互斥访问和同步。

1973年，Hanson和Hoare又把“秘书”概念发展为管程概念。



管程的基本概念（1）

※基本思想

系统中的各种软硬件资源，其特性都可用数据结构抽象地描述，把对该共享数据结构实施的操作定义为一组过程。

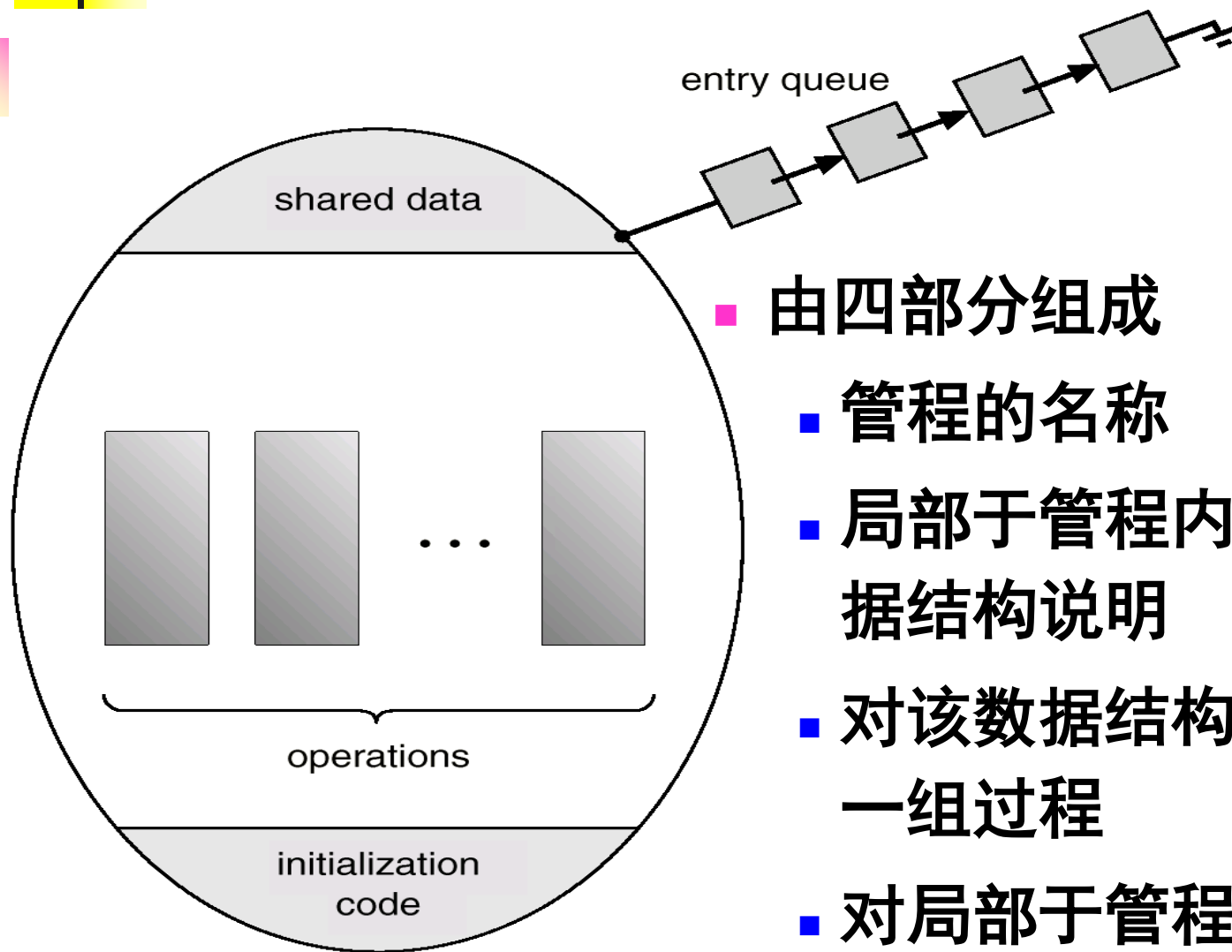
进程对共享资源的访问都是通过这组过程对共享数据结构的操作来实现的，这组过程可以根据资源的使用情况，接受或阻塞进程的访问，确保每次只有一个进程使用共享资源，实现进程的互斥。



管程的基本概念（2）

※管程的定义

一个**数据结构**和在该数据结构上能被并发进程所执行的**一组操作**，这组操作能同步进程和改变管程中的数据。如下图所示。



■ 由四部分组成

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

管程的基本概念 (3)

※管程的结构

TYPE <管程名> = MONITOR

<共享变量说明>;

procedure <过程名>(<形式参数表>;

begin

<过程体>;

end;

.....

procedure <过程名>(<形式参数表>;

begin

<过程体>;

end;

begin

<管程的局部数据初始化语句序列>;

end;

局部于管程内部的数据结构，只能被局部于管程内部的过程所访问；反之，局部于管程内部的过程也只能访问管程内的数据结构

管程相当于围墙，它把共享变量和对它进行操作的若干过程围了起来，所有进程要访问临界资源时，都必须经过管程才能进入，而管程每次只允许一个进程进入管程，从而实现进程互斥。



管程的基本概念（4）

- 管程的特性

- 模块化

- 是一个基本程序单位，可单独编译

- 抽象数据类型

- 管程中不仅有数据，而且有对数据的操作

- 信息隐蔽

- 数据结构和过程的具体实现外部不可见



管程的基本概念（5）

■ 管程与进程作比较

- 管程定义的是公用数据结构，而进程定义的是私有数据结构**PCB**；
- 管程把共享变量上的同步操作集中起来，而临界区却分散在每个进程中；
- 管程是为管理共享资源而建立的，进程主要是为占有系统资源和实现系统并发性而引入的；



管程的基本概念（6）

■ 管程与进程作比较

- 管程是被要使用共享资源的进程所调用的，管程和调用它的进程不能并发工作，而进程之间能并发工作，并发性是进程的固有特性；
- 管程是**OS**中一个资源管理模块，供进程调用；而进程有生命周期，由创建而产生、由撤销而消亡



管程的基本概念 (7)

※条件变量

在管程机制中，当某进程通过管程请求临界资源未能满足时，管程便调用wait原语使该进程等待，但等待的原因可能有多个，为了加以区别，在P，V操作前，引入条件变量加以说明。

条件变量也是一种抽象数据类型，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。

(1) 条件变量的定义格式

```
var x, y: condition;
```



管程的基本概念（8）

※条件变量

（2）对条件变量执行的两种操作

- wait操作（如x.wait）：用来阻塞正在调用管程的进程并将其插入到与条件变量x相应的等待队列上，并释放管程，直到x条件变化。此时其它进程可以使用该管程。
- signal操作（如x.signal）：用来唤醒与条件变量x相应的等待队列的一个进程（因x条件而阻塞）。如果没有这样的进程，则继续执行原进程，不产生任何结果。



2.4 经典进程的同步问题

在多道程序环境下，进程同步问题十分重要，出现一系列经典的进程同步问题，其中有代表性有：

- 生产者—消费者问题
- 哲学家进餐问题
- 读者—写者问题



一、“生产者—消费者”问题

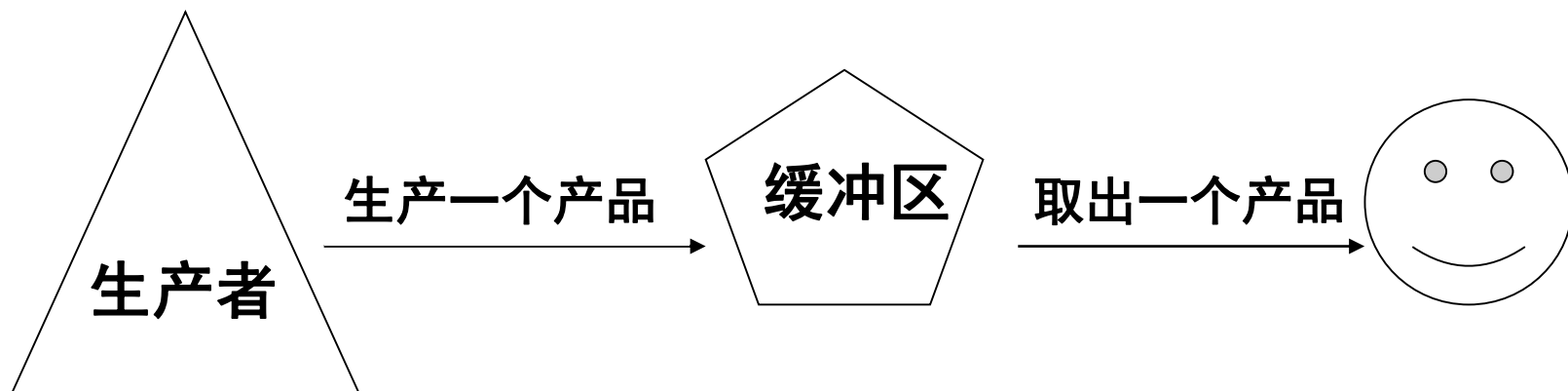
■ 问题描述：

“生产者—消费者”问题是最著名的进程同步问题。它描述了一组生产者向一组消费者提供产品，它们共享一个缓冲池（有 n 个缓冲区），生产者向其中投放产品，消费者从中取得产品。

它是许多相互合作进程的抽象，如输入进程与计算进程；计算进程与打印进程等。

一、“生产者—消费者”问题

- 一个生产者，一个消费者，公用一个缓冲区



定义两个同步信号量：

`empty`——表示缓冲区是否为空，初值为1。

`full`——表示缓冲区中是否为满，初值为0。



一、“生产者—消费者”问题

生产者进程:

```
while (TRUE) {
```

```
    生产一个产品;
```

```
    P(empty);
```

```
    把产品送往Buffer;
```

```
    V(full);
```

```
}
```

消费者进程:

```
while (TRUE) {
```

```
    P(full);
```

```
    从Buffer取出一个产品;
```

```
    V(empty);
```

```
    消费产品;
```

```
}
```



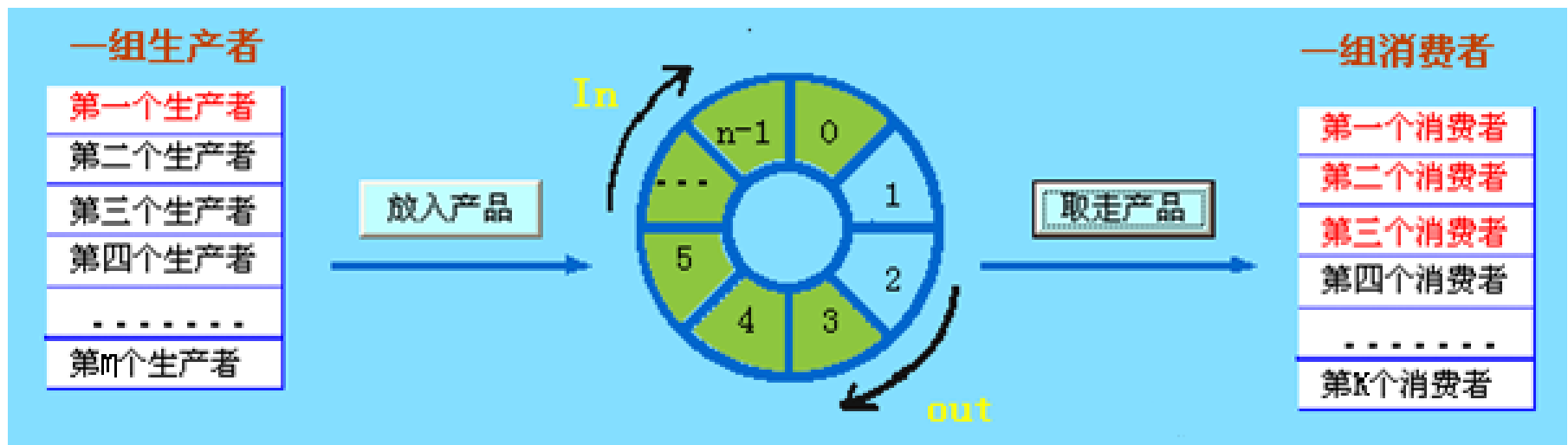
一、“生产者—消费者”问题

- M个生产者，K个消费者，公用有n个缓冲区的缓冲池

设缓冲池的长度为 n ($n > 0$)，一群生产者进程 P_1, P_2, \dots, P_m ，一群消费者进程 C_1, C_2, \dots, C_k ，如图所示。假定生产者和消费者是相互等效，只要缓冲池未满，生产者就可以把产品送入缓冲区，类似地，只要缓冲池未空，消费者便可以从缓冲区取走产品并消耗它。生产者和消费者的同步关系将禁止生产者向满的缓冲池输送产品，也禁止消费者从空的缓冲池提取产品。

一、“生产者—消费者”问题

- M个生产者，K个消费者，公用有n个缓冲区的缓冲池





“生产者—消费者”问题的解决

- 设置两个同步信号量及一个互斥信号量

empty: 说明空缓冲区的数目，其初值为缓冲池的大小 n ，表示消费者已把缓冲池中全部产品取走，有 n 个空缓冲区可用。

full: 说明满缓冲区的数目（即产品数目），其初值为 0 ，表示生产者尚未把产品放入缓冲池，有 0 个满缓冲区可用。

mutex: 说明该缓冲池是一临界资源，必须互斥使用，其初值为 1 。



“生产者—消费者”问题的解决

■ “生产者—消费者”问题的同步算法描述

```
semaphore full=0; /*表示满缓冲区的数目*/  
semaphore empty=n; /*表示空缓冲区的数目*/  
semaphore mutex=1; /*表示对缓冲池进行操作的互斥信  
    号量*/
```

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




producer()

```
{  
    while(true)  
    {  
        生产一个产品放入nextp;  
        P(empty);  
        P(mutex);  
        Buffer[in]=nextp;  
        in=(in+1) mod n;  
        V(mutex);  
        V(full);  
    }  
}
```

consumer()

```
{  
    while(true)  
    {  
        P(full);  
        P(mutex);  
        nextc =Buffer[out];  
        out=(out+1) mod n;  
        V(mutex);  
        V(empty);  
        消费nextc中的产品;  
    }  
}
```



“生产者-消费者”问题中应注意

1. 互斥信号量的P、V操作在每一进程中必须成对出现。
2. 对资源信号量(full, empty)的P、V操作也必须成对出现，但可分别处于不同的进程中。
3. 多个P操作顺序不能颠倒。
4. 先执行资源信号量的P操作，再执行互斥信号量的P操作，否则可能引起进程死锁。
5. 利用AND信号量解决生产者-消费者问题。



“生产者-消费者”问题中应注意

6. 它是一个同步问题：

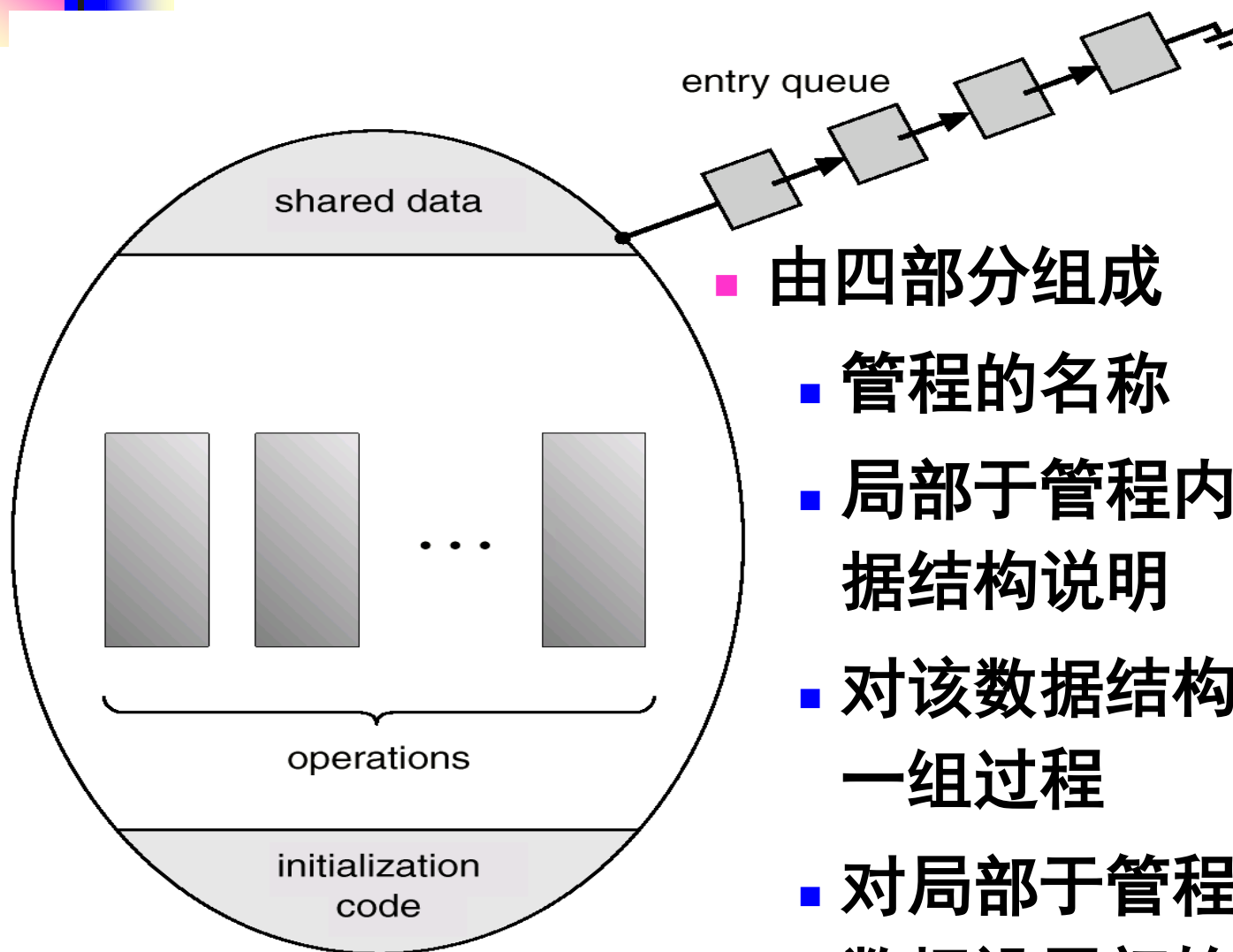
(1) 消费者想要取产品，缓冲池中至少有一个缓冲区是满的。

(2) 生产者想要放产品，缓冲池中至少有一个缓冲区是空的。

7. 它是一互斥问题

缓冲池是临界资源，因此，各生产者进程和各消费者进程必须互斥访问。

用管程机制解决生产者—消费者问题



■ 由四部分组成

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

用管程机制解决生产者—消费者问题

1、建立Producer-consumer (PC) 管程

Type PC=monitor

**var in,out,count:integer;
buffer:array[0,...,n-1] of item;
notfull,notempty:condition;**

put(item);

get(item);

begin

in:=out:=0;

count:=0;

end

/* 初始化代码*/

管程中的两个条件变量：

(1) notfull 表示等待未满缓冲区（空缓冲区）。

(2) notempty 表示等待未空缓冲区（满缓冲区）。



用管程机制解决生产者—消费者问题

1、建立Producer-consumer (PC) 管程

- **put(item) 过程**

生产者利用此过程将自己生产的产品放到缓冲池中，若发现缓冲池已满 ($\text{count} \geq n$)，则等待。

- **get(item) 过程**

消费者利用此过程将缓冲池中的产品取走，若发现缓冲池已空 ($\text{count} \leq 0$)，则等待。



put (item)

get (item)

Procedure entry put(item)

begin

if count \geq n then

notfull.wait;

buffer[in]:=nextp;

in:=(in+1) mod n;

count:=count+1;

if notempty.queue then

notempty.signal;

end

Procedure entry get(item)

begin

if count = 0 then

notempty.wait;

nextc:=buffer[out];

out:=(out+1) mod n;

count:=count-1;

if notfull.queue then

notfull.signal;

end



2、生产者—消费者问题的解决

Producer:begin

repeat

produce an item in nextp;

PC.put(item);

until false

end

Consumer:begin

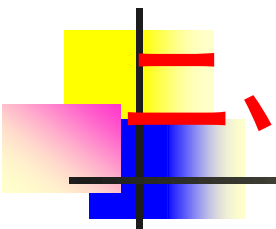
repeat

PC.get(item);

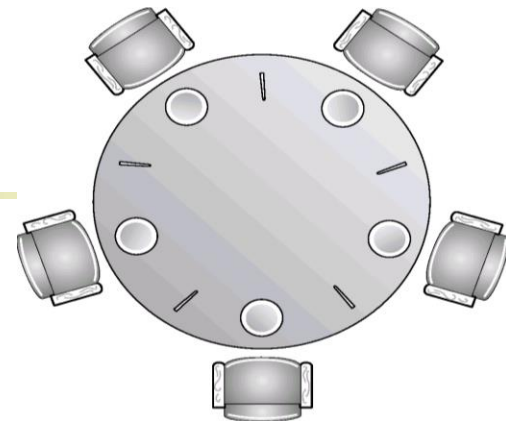
consume the item in nextc;

until false

end



二、“哲学家进餐”问题



■ 问题描述：

有五个哲学家，他们的生活方式是交替地进行思考和进餐。他们共用一张圆桌，分别坐在五张椅子上。在圆桌上有五个碗和五支筷子，平时一个哲学家进行思考，饥饿时便试图取用其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐。进餐毕，放下筷子又继续思考。

- 哲学家进餐问题可看作是并发进程并发执行时，处理共享资源的一个有代表性的问题。



哲学家进餐问题的解决

■ semaphore stick[5]={1,1,1,1,1}; /*分别表示5支筷子*/

main()

{

cobegin

philosopher(0);

philosopher(1);

philosopher(2);

philosopher(3);

philosopher(4);

coend

}



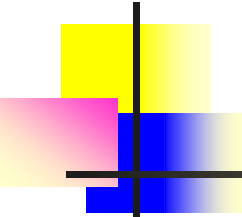
第i个哲学家的活动算法

philosopher(int i)

```
{  
    while(true)  
    {  
        思考;  
        P(stick[i]);  
        P(stick[(i+1) mod 5]);  
        进餐;  
        V(stick[i]);  
        V(stick[(i+1) mod 5]);  
    }  
}
```

说明:

- 1、此算法可以**保证不会有相邻的两位哲学家同时进餐。**
- 2、若五位哲学家同时饥饿而各自拿起了左边的筷子，这使五个信号量stick均为0，当他们试图去拿起右边的筷子时，都将因无筷子而无限期地等待下去，即**可能会引起死锁。**

- 
-
- 上述解法可能出现永远等待，有若干种办法可避免死锁：
 - 至多允许四个哲学家同时想到要进餐（解决方法一）；
 - 奇数号先取左手边的筷子，偶数号先取右手边的筷子（解决方法二）；
 - 每个哲学家取到手边的两只筷子才进餐，否则一只筷子也不取（解决方法三）。



解决的方法（一）

- 设置一个信号量 S_m , 其初值为4, 用于限制同时想到要进餐的哲学家数目至多为4, 这样, 第 i 个哲学家的活动可描述为:

```
while(true)
```

```
{
```

```
    wait( $S_m$ );
```

```
    wait(stick[i]);
```

```
    wait (stick[(i+1) mod 5]);
```

```
    .....
```

```
    eat;
```

```
    .....
```

```
    signal(stick[i]);
```

```
    signal(stick[(i+1) mod 5]);
```

```
    signal( $S_m$ );
```

```
    .....
```

```
    think;
```

```
}
```



解决的方法（二）

```
while(true)
{if odd(i)
    {wait(stick[i]);
      wait (stick[(i+1) mod 5]);
    }
  else
    {wait (stick[(i+1) mod 5]);
      wait(stick[i]);
    }
  .....
  eat;
  .....
```

```
signal(stick[i]);
signal (stick[(i+1) mod 5]);
.....
think;
.....
}
```

对5个哲学家，假设规定：单号者进餐时，先拿左手（ i ）的筷子，然后再拿右手（ $i+1$ ）的筷子。双号则先右后左。这样既可使5个人同时想到要进餐，又不致产生死锁。



解决的方法（三）

- 利用**AND信号量机制**解决哲学家进餐问题

```
semaphore  stick[5]={1, 1, 1, 1, 1};  
philosopher (int i)  
{  
    while (true)  
    {  
        think;  
        Swait(stick[(i+1) mod 5], stick[i]);  
        eat;  
        Ssignal(stick[(i+1) mod 5], stick[i]);  
    }  
}
```

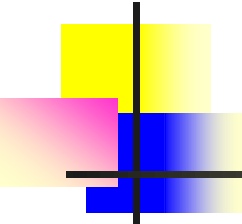


三、“读者—写者”问题

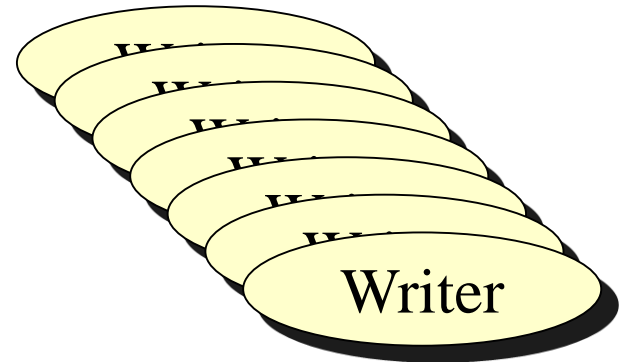
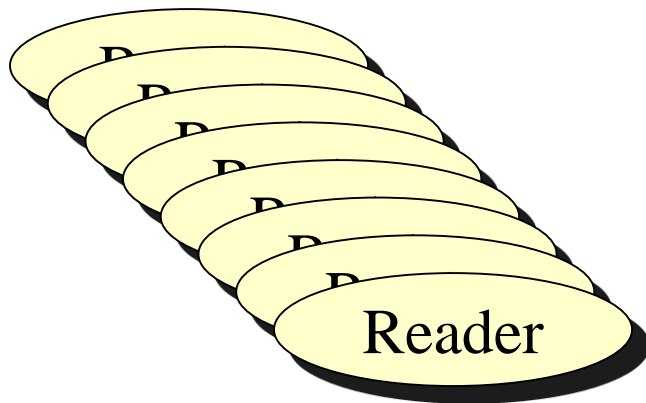
■ 问题描述：

一个数据对象（数据文件或记录）可被多个进程共享。其中，读者（reader）进程要求读，写者（writer）进程要求写或修改。

为了保证读写的正确性和数据对象的一致性，系统要求：当有读者进程读文件时，不允许任何写者进程写，但允许多个读者同时读；当有写者进程写时，不允许任何其它写者进程写，也不允许任何读者进程读。

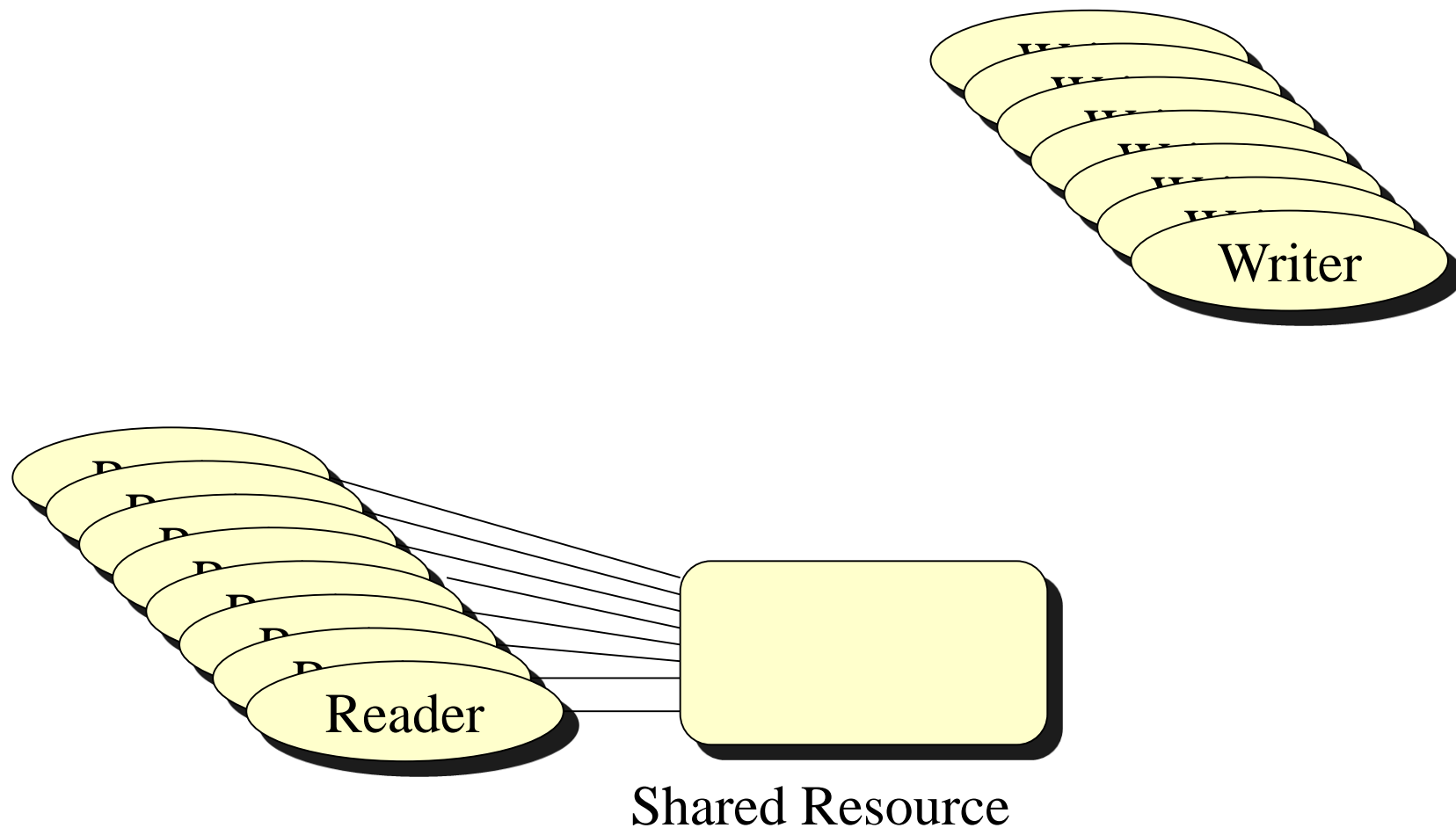


三、“读者—写者”问题



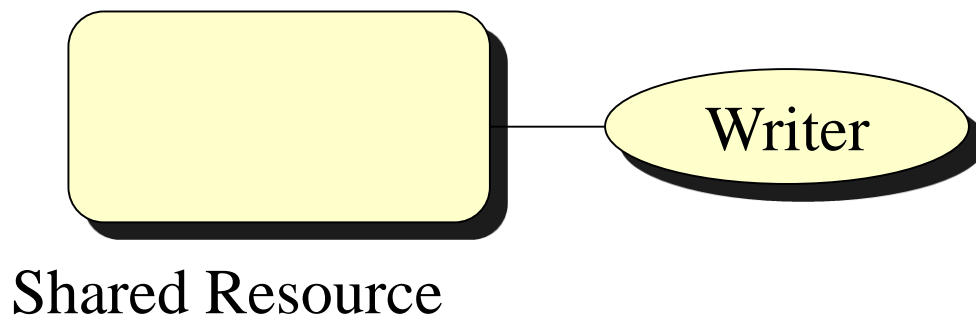
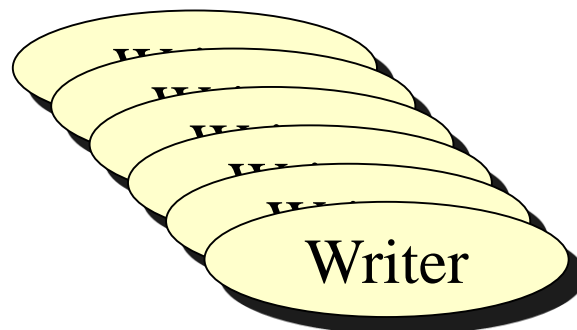
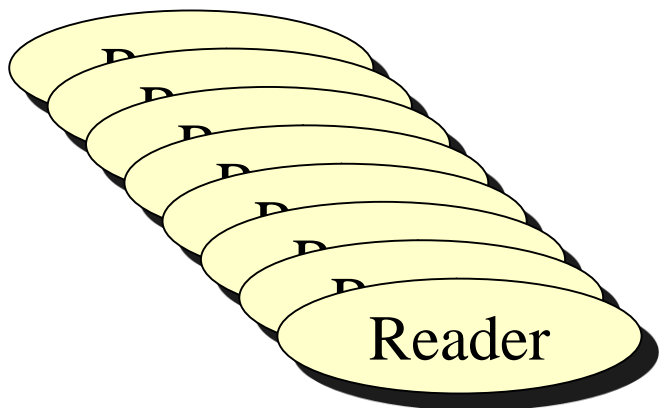
Shared Resource

三、“读者—写者”问题





三、“读者—写者”问题





“读者—写者”问题的解决

■ “读者—写者”问题的同步算法描述

设置一个共享变量和两个信号量：

共享变量**readcount**：记录当前正在读数据集的读进程数目，初值为0。

读互斥信号量**rmutex**：表示读进程互斥地访问共享变量**readcount**，初值为1。

写互斥信号量**wmutex**：表示写进程与其它进程（读、写）互斥地访问数据集，初值为1。



“读者—写者”问题的解决

■ “读者—写者”问题的同步算法描述

```
semaphore rmutex=1;  
semaphore wmutex=1;  
int readcount=0;  
main()  
{  
    cobegin  
        reader();  
        writer();  
    coend  
}
```



reader()

```
{  
    while(true)  
    {  
        P(rmutex);  
        {  
            if(readcount==0) P(wmutex);/*第一位读者阻止写者  
            */  
            readcount++;  
            V(rmutex);  
            读数据集;  
            P(rmutex);  
            {  
                readcount--;  
                if(readcount==0) V(wmutex);/*第末位读者允许写者  
                进*/  
                V(rmutex);  
            }  
        }  
    }  
}
```

修改
readcount

修改
readcount



writer()

```
{  
  while(true)  
  {  
    P(wmutex); /*阻止其它进程（读、写）； * /  
    写数据集;  
    V(wmutex); /*允许其它进程（读、写）； * /  
  }  
}
```

如果增加一个限制条件：最多只允许RN个读者同时读，该如何解决？（P72 利用信号量集机制解决）

利用信号量集解决“读者—写者”问题

■ 一般信号量集的几种特殊情况

- **$\text{Swait}(S, d, d)$** ：信号量集中只有一个信号量，但允许它每次申请 d 个资源，当资源数少于 d 时，不予分配。
- **$\text{Swait}(S, 1, 1)$** ：蜕化为一般的记录型信号量（ $S > 1$ 时）或互斥信号量（ $S = 1$ 时）。
- **$\text{Swait}(S, 1, 0)$** ：很有用的信号量操作，相当于一个可控开关，当 $S \geq 1$ 时，开（允许多个进程进入）；当 $S = 0$ 时，关（阻止任何进程进入）。



利用信号量集解决“读者—写者”问题

```
const RN=100;           //最多只允许RN个读者同时读
semaphore L=RN, mx=1;    //L是资源信号量, mx是互斥信号量
reader() {
    while(true)
    {
        Swait(L, 1, 1);   //蜕化为一般的记录型信号量wait(L)
        Swait(mx, 1, 0);  //相当于一个可控开关
        . . .
        perform read operation;
        . . .
        Ssignal(L, 1);    //相当于一一般的记录型信号量signal(L)
    }
}
```



利用信号量集解决“读者—写者”问题

```
writer() {  
    while(true)  
    { //既无writer在写(mx==1)又无reader在读(L==RN)  
        Swait(mx, 1, 1; L, RN, 0);  
        perform write operation;  
        Ssignal(mx, 1);  
    }  
}  
  
main() {  
    cobegin  
        reader(); writer();  
    coend  
}
```



2.5 进程通信—高级通信（1）

一、进程通信的类型

进程通信是指进程之间的信息交换。根据所交换的信息量的多少分为：

❖ **低级通信** 进程之间交换的信息量较少且效率低。如进程同步和互斥。

❖ **高级通信** 进程之间交换的信息量较多且效率高。又分为：



2.5 进程通信—高级通信（2）

➤ **共享存储器系统** 指进程之间通过对共享某些数据结构或共享存储区读写来交换数据。

➤ **消息传递系统** 指进程间的通信以消息为单位，程序员可通过通信原语实现通信，按其实现方式不同可分为：

- ✓ **直接通信方式** 发送进程直接把消息发送给接收进程。

- ✓ **间接通信方式** 发送进程把消息发送到某个中间实体（信箱），接收进程从中取得消息。



2.5 进程通信—高级通信（3）

➤管道通信系统

发送进程（写进程）以字符流形式将大量数据送入管道（管道：用于连接读进程和写进程以实现它们之间通信的一个共享文件，又称pipe文件），接收进程（读进程）从管道接收数据。

三方面的协调能力：互斥、同步、确定对方是否存在。



2.5 进程通信—高级通信（4）

➤ **客户机-服务器系统** 在网络环境的各种应用领域已成为当前主流的通信实现机制。主要的实现方式分为三类：

- ✓ **套接字** 一个套接字就是一个通信标识类型的数据结构。是进程通信和网络通信的基本构件。
- ✓ **远程过程调用** 是一个通信协议，用于通过网络连接的系统。
- ✓ **远程方法调用** 如果涉及的软件采用面向对象编程，则RPC可称作远程方法调用。



一、共享存储器系统

相互通信的进程通过**共享数据结构**和**存储区**进行通信，因而可进一步分为：

- **基于共享数据结构的通信方式**（低效，只适于传递少量数据，属于低级通信）
- **基于共享存储区的通信方式**。为了传送大量数据，在存储区中划出一块共享存储区，各个进程可通过对共享存储区进行读或写数据实现通信。属于高级通信。



二、消息传递系统

——直接通信方式（消息缓冲通信）（1）

发送进程利用OS所提供的发送命令，直接把消息发送给接收进程。

- 要求：发送进程和接收进程都以显式方式提供对方的标识符。（对称寻址方式）
- 系统提供的两条通信原语：
 - **Send(receiver,message);**发送一个消息给接收进程receiver;
 - **Receive(sender,message);**接收sender发来的消息;



二、消息传递系统

——直接通信方式（消息缓冲通信）（2）

- 生产者-消费者问题的解决

repeat

⋮

Produce an item in nextp;

⋮

Send(consumer,nextp);

until false;

repeat

⋮

Receive(producer,nextc);

⋮

**consume the item in
nextc;**

until false;



三、消息传递系统实现中的若干问题（1）

- 消息格式
 - 消息头和消息正文
- 进程同步方式
 - 发送进程阻塞，接收进程阻塞（用于进程之间紧密同步，无缓冲时）
 - 发送进程不阻塞，接收进程阻塞（应用最广）
 - 发送进程和接收进程均不阻塞（较常见，消息队列）



三、消息传递系统实现中的若干问题（2）

■ 通信链路

必须在发送进程和接收进程之间建立一条通信链路，有以下两种方式：

- 用于计算机网络：由发送进程在通信之前用显示的“建立连接”原语请求系统建立一条通信链路，用完后拆除。
- 用于单机系统：发送进程无须明确提出建立链路的请求，只须利用系统提供的发送原语，系统会自动建立一条链路（单向或双向链路）。



二、消息传递系统

——间接通信方式（信箱通信）（1）

发送进程把消息发送到某个中间实体（信箱），接收进程从中取得消息。

- **信箱的结构：**信箱头、信箱体
- **系统提供的若干条原语**
 - 信箱创建和撤消原语
 - 两条通信原语

`Send(mailbox, message)` 将一个消息发送给指定信箱；

`Receive(mailbox, message)` 从指定信箱中接收一个消息；

- **信箱的分类：**私用信箱（进程为自己创建）、公用信箱（操作系统创建）、共享信箱（由某进程创建，指明是共享的）



二、消息传递系统

——间接通信方式（信箱通信）（2）

- 在利用信箱通信时，在发送进程和接收进程之间，存在着四种关系：
 - **一对一关系**：即可以为发送进程和接收进程建立一条专用的通信链路；
 - **多对一关系**：允许提供服务的进程与多个用户进程进行交互，也称客户/服务器交互；
 - **一对多关系**：允许一个发送进程与多个接收进程交互，使发送进程用广播的形式，发送消息；
 - **多对多关系**：允许建立一个公用信箱，让多个进程都能向信箱投递消息，也可取走属于自己的消息。



四、消息缓冲队列通信机制实例（1）

■ 消息缓冲区

```
typedef struct message_buffer {  
    int sender;    发送消息的进程名或标识符  
    int size;      发送的消息长度  
    char *text;    发送的消息正文  
    struct message_buffer *next;  指向下一个消息缓冲  
    区的指针  
}
```

■ 在进程的PCB中涉及通信的数据结构

```
struct message_buffer *mq;  消息队列队首指针  
semaphore mutex;           消息队列互斥信号量，初值为1  
semaphore sm;              资源信号量，表示接收进程消息队列上  
    消息的个数，初值为0，是控制收发进程同步的信号量
```



四、消息缓冲队列通信机制实例（2）

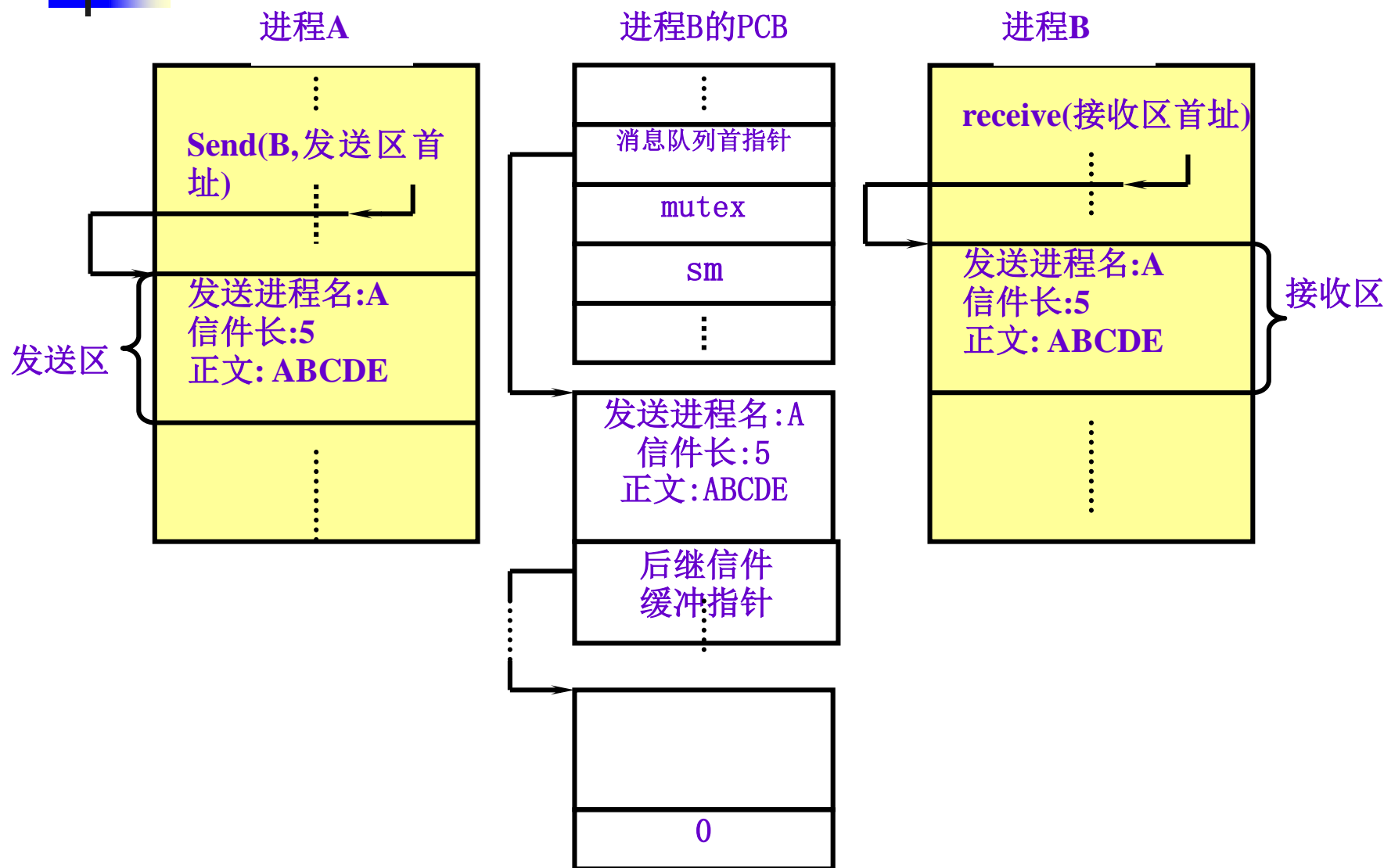
■ 发送原语 `send`

申请一个消息缓冲区，把发送区内容复制到这个缓冲区中；找到接收进程的PCB，执行互斥操作 $P(\text{mutex})$ ；把缓冲区挂到接收进程消息队列的尾部，执行 $V(\text{mutex})$ ，然后执行 $V(\text{sm})$ ，即消息数加1；

■ 接收原语 `receive`

执行 $P(\text{sm})$ ，查看是否有信件；执行互斥操作 $P(\text{mutex})$ ，从消息队列中摘下第一个消息，执行 $V(\text{mutex})$ ；把消息缓冲区内容复制到接收区，释放消息缓冲区

四、消息缓冲队列通信机制实例（3）





四、消息缓冲队列通信机制实例（4）

■ 发送原语send

```
void send(receiver, a) {  
    getbuf(a.size, i); //根据a.size申请缓冲区;  
    i.sender:= a.sender;  
    i.size:=a.size;  
    i.text:=a.text;  
    i.next:=0;  
    //将发送区a中的信息复制到消息缓冲区i中;  
    getid(PCB set, receiver.j); //获得接收进程内部标识符;  
    wait(j.mutex);  
    insert(j.mq, i); //将消息缓冲区插入消息队列;  
    signal(j.mutex);  
    signal(j.sm);  
}
```



四、消息缓冲队列通信机制实例（5）

■ 接收原语receive

```
void receive(b) {  
    j := internal name;    //j为接收进程内部的标识符;  
    wait(j.sm);  
    wait(j.mutex);  
    remove(j.mq, i);    //将消息队列中第一个消息移出, 移到缓冲区i;  
    signal(j.mutex);  
    b.sender := i.sender;  
    b.size := i.size;  
    b.text := i.text;  
    releasebuf(i);    //释放消息缓冲区  
}
```



2.6 线程（1）

20世纪80年代中期，出现了比进程更小的
能独立运行的基本单位——线程

线程的引入

- 引入进程的目的
 - 使多个程序并发执行
 - 改善资源利用率，提高系统吞吐量



2.6 线程（2）

- 引入线程的目的
 - 减少程序并发执行时所付出的时空开销
 - 使操作系统具有更好的并发性
- 从进程的两个基本属性（拥有资源的独立单位、可独立调度和分派的基本单位）来进一步说明引入线程的目的



一、线程的基本概念（线程的定义）

线程的定义

线程是进程中的一个实体，是被系统独立调度和分派的基本单位。线程基本上自己不拥有系统资源，只拥有少部分在运行中必不可少的资源，但它可与同属一个进程中的其它线程共享进程所拥有的全部资源。



一、线程的基本概念（线程的定义）

- 一个线程可以创建和撤消另一个线程
- 同一个进程中的多个线程可以并发地执行
- 当一个线程改变了存储器中的一个数据项时，在其它线程访问这一项时它们能够看到变化后的结果
- 如果一个线程为读操作打开一个文件时，同一个进程中的其它线程也能够从该文件中读。



一、线程的基本概念（线程的属性）

线程的属性

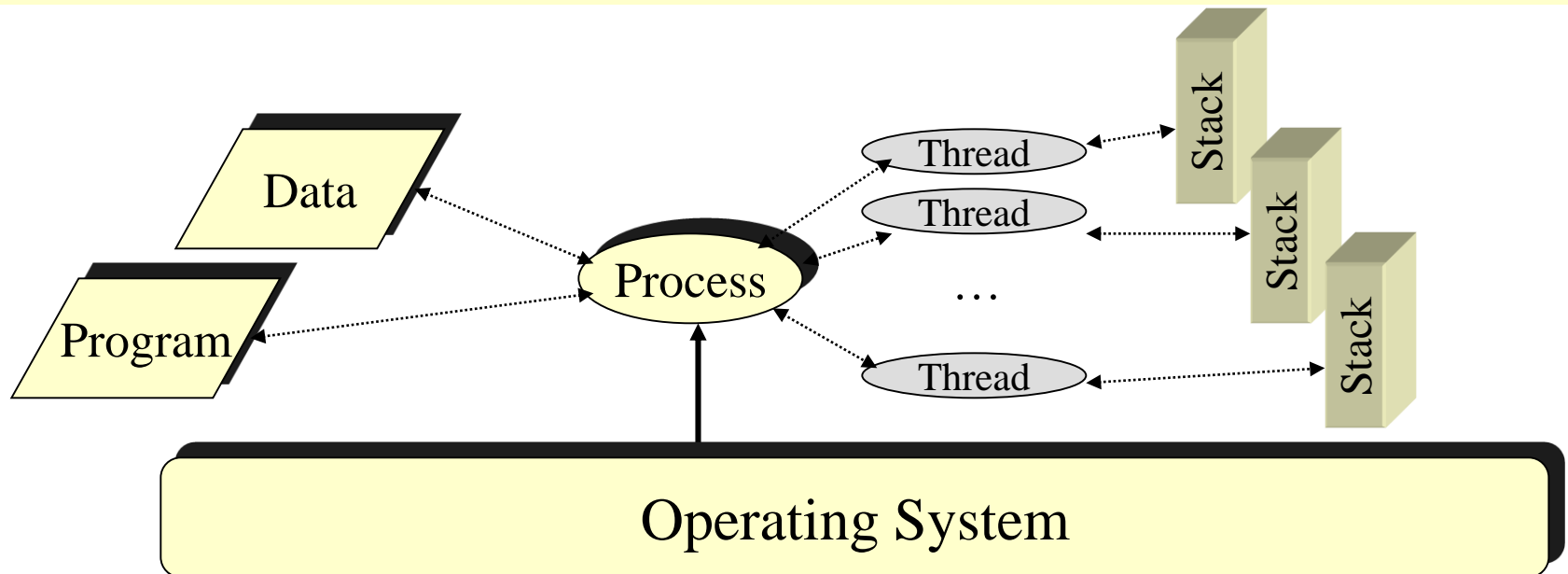
- 线程是轻型实体。基本上不拥有系统资源，只有一点必不可少的、能保证独立运行的资源。
- 线程是独立调度和分派的基本单位。
- 可并发执行。一个进程中的多个线程可并发执行；不同进程中的线程也能并发执行。
- 共享进程资源。同一进程中的各个线程，都可以共享该进程所拥有的资源。

一、线程的基本概念（线程和进程的比较）

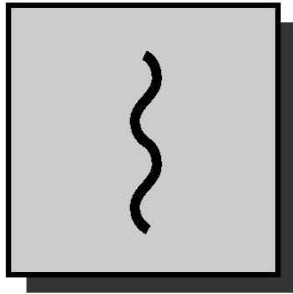
	进程	线程
引入目的	能并发执行, 提高资源的利用率和系统吞吐量.	提高并发执行的程度, 进一步提高资源的利用率和系统吞吐量.
并发性	较低	较高
基本属性 (调度)	资源拥有的基本单位—进程 独立调度/分派的基本单位—进程	资源拥有的基本单位—进程 独立调度/分派的基本单位—线程
基本状态	就绪, 执行, 等待	就绪, 执行, 等待
拥有资源	资源拥有的基本单位—进程	资源拥有的基本单位—进程
系统开销	创建/撤消/切换时空开销较大	创建/撤消/切换时空开销较小
系统操作	创建, 撤消, 切换	创建, 撤消, 切换
存在标志	进程控制块PCB	进程控制块PCB, 线程控制块TCB
支持多处理机系统	单线程进程, 只能运行在一个处理机上	多线程进程, 可以将多个线程分配到多个处理机上
关系	单进程单线程; 单进程多线程; 多进程单线程; 多进程多线程	

一、线程的基本概念（线程和进程的比较）

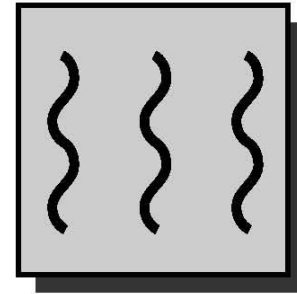
- **Process** is an infrastructure in which execution takes place → (address space + resources)
- **Thread** is a program in execution within a process context – each thread has its own stack



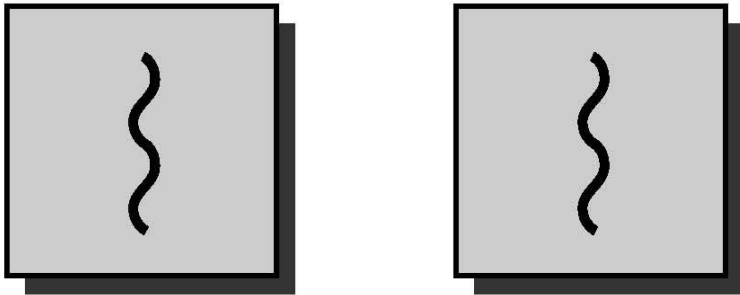
一、线程的基本概念（线程和进程的比较）



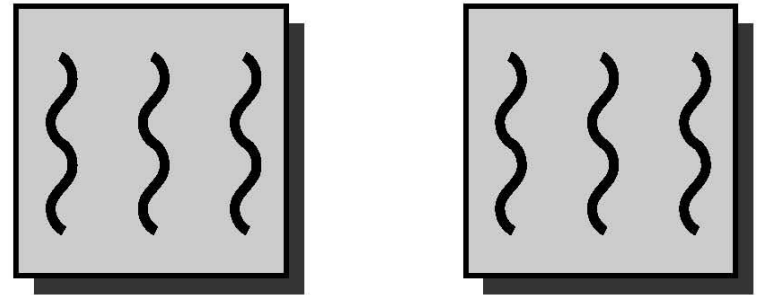
one process
one thread



one process
multiple threads



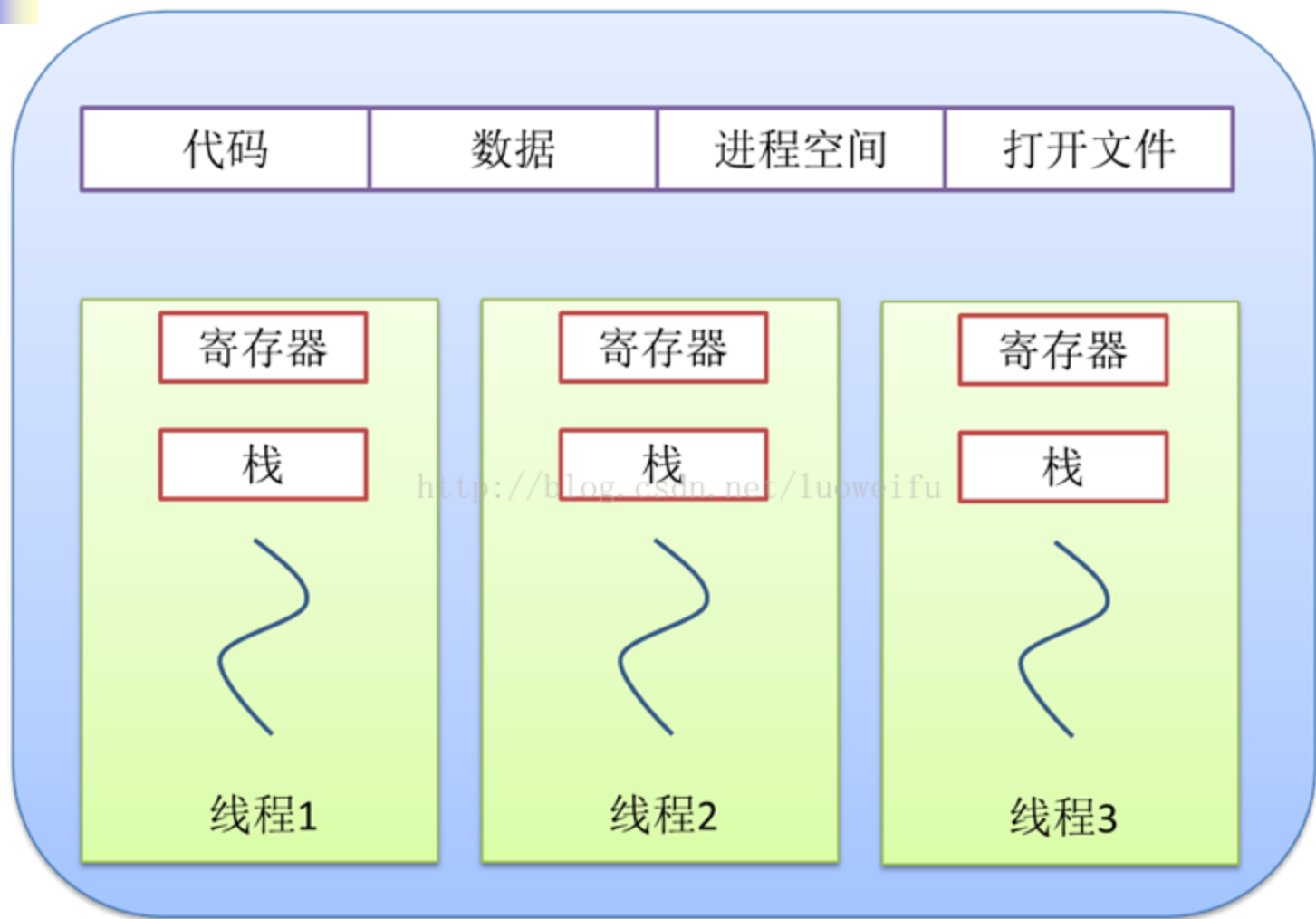
multiple processes
one thread per process



multiple processes
multiple threads per process

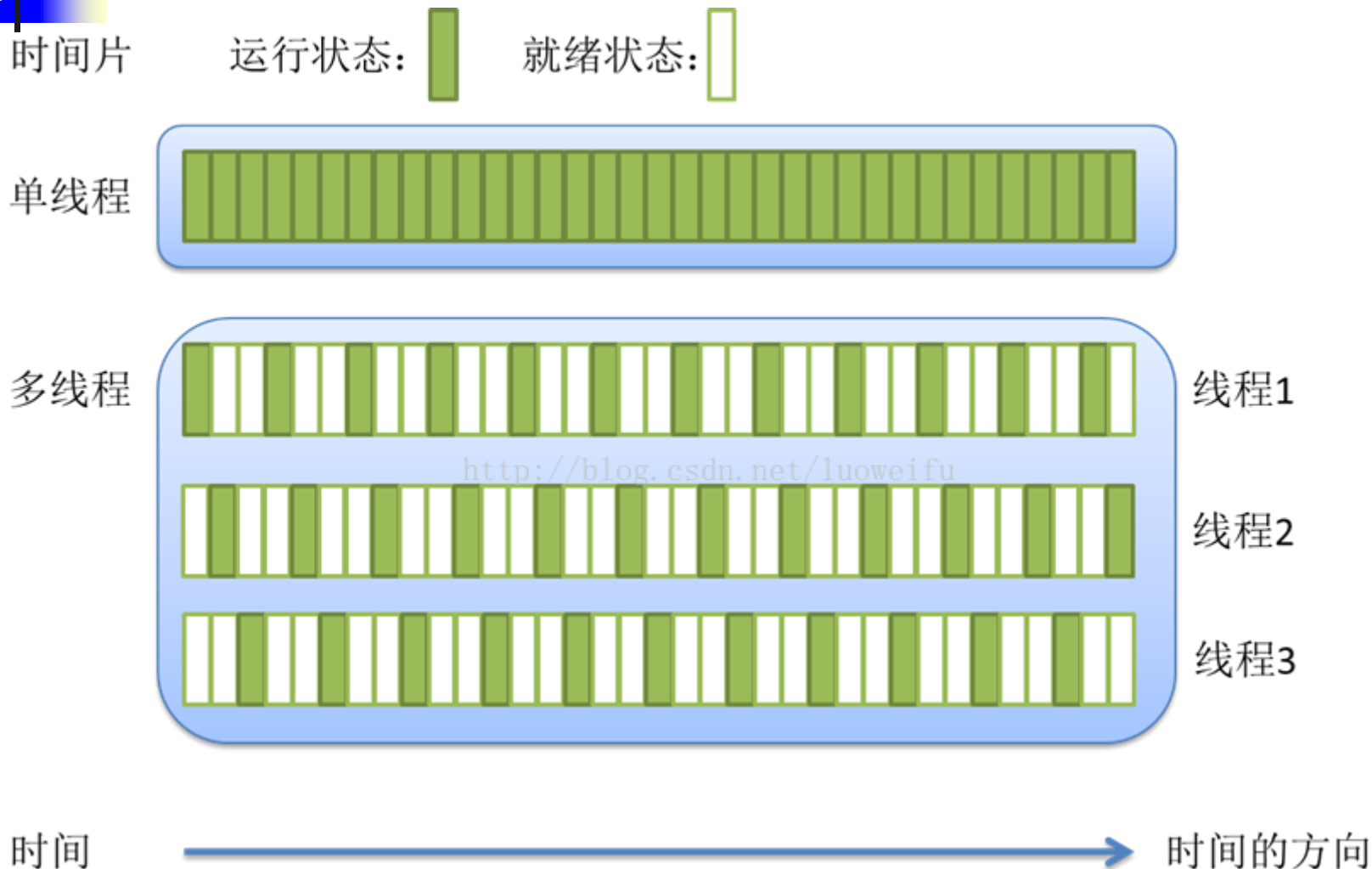
} = instruction trace

一、线程的基本概念（线程和进程的比较）



进程和线程的资源共享关系

一、线程的基本概念（线程和进程的比较）



单线程和多线程的关系



二、多线程（1）

- 指**OS**支持在一个进程中执行多个线程的能力
- 传统的每个进程中只有一个线程在执行
 - **MS-DOS**是支持单用户进程和单线程的**OS**
 - 传统的**UNIX**支持多用户进程，但只支持每个进程一个线程（即多进程单线程）
- 单进程多线程的例子：**Java**运行环境
- 多进程多线程的例子：**Windows 2000**、**Solaris**和很多现代版本的**UNIX**



二、多线程（2）

- **多线程OS中进程的属性**
 - 作为系统资源分配的单位
 - 可包括多个线程。至少包括一个线程，这些线程可并发执行。**OS**中的所有线程都只能属于某一个特定的进程
 - 进程不是一个可执行的实体，线程是作为独立运行的基本单位



三、线程的状态和线程控制块

■ 线程运行的三个状态

- 执行状态、就绪状态、阻塞状态

■ 线程控制块TCB

- 线程标示符
- 一组寄存器（包括PC、状态寄存器和通用寄存器的内容）
- 线程运行状态
- 优先级
- 线程专有存储区
- 信号屏蔽
- 堆栈指针



四、线程的实现方式（1）

- **内核级线程（KST-Kernel Supported Threads,内核支持线程）**
 - 线程的创建、调度和管理由**OS内核**在**内核空间**实现；
 - 内核为整个进程和进程中的所有线程维护现场信息；
 - 内核建立和维护进程的**PCB、TCB**，内核的调度是在线程的基础上进行的；
 - 如**Macintosh和OS/2操作系统**



四、线程的实现方式（2）

■ KST的优点

- 在**多处理器**上，内核能够同时调度同一进程中的多个**线程并行**执行；
- 进程中的一个线程被阻塞时，其他线程仍可运行；
- 具有很小的数据结构和堆栈，线程的切换比较快、切换开销小；
- 内核本身也可以采用多线程技术，提高系统效率。

■ KST的缺点

- 应用程序运行在用户态，线程调度程序运行在内核态，所以同一进程中的线程切换要有两次模式转换（用户态→内核态→用户态）。



四、线程的实现方式（3）

- **用户级线程（ULT—User Level Threads）**
 - **ULT由用户应用程序建立；**
 - **由用户应用程序负责对这些线程进行调度和管理；**
 - **OS内核不知道ULT的存在，即ULT与内核无关；**
 - **设置了ULT的系统，系统的调度仍是以进程为单位进行的；**
 - **如：MS-DOS,UNIX,一些数据库管理系统（如Informix）**



四、线程的实现方式（4）

■ **ULT的优点**

- 应用程序中的线程切换的时空开销比内核级线程的开销小得多，因为不需要内核特权方式；
- 线程的调度算法与**OS**的低级调度算法无关；
- **ULT**可适用于任何**OS**，因为**ULT**的实现与内核无关；

■ **ULT的缺点**

- 如果一个线程被阻塞，则该进程中所有其他线程都将被阻塞；
- 如果系统有多个处理器，无法实现一个进程中的多个线程并行执行。



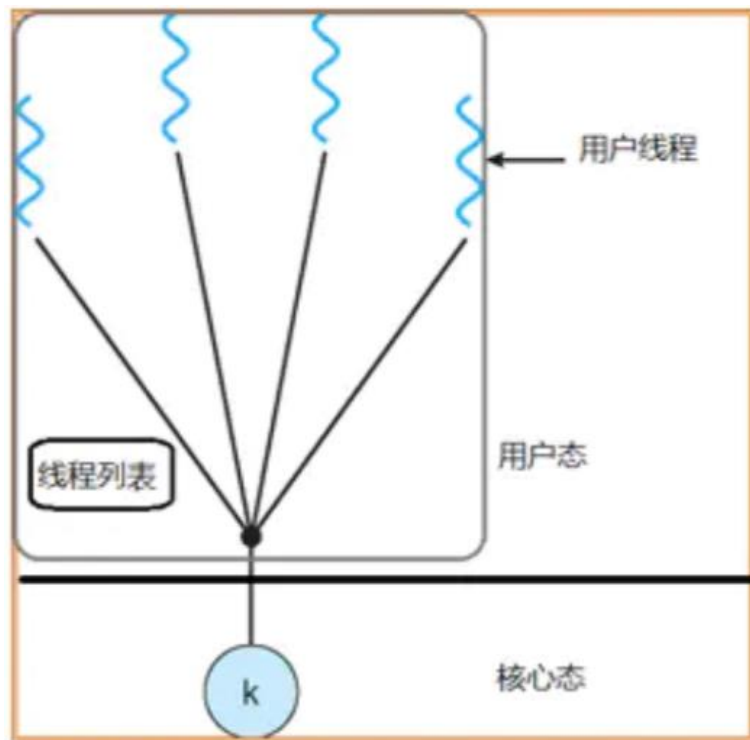
四、线程的实现方式（5）

- 组合方式—**ULT**和**KST**结合
 - 把**ULT**和**KST**两种方式进行组合，提供了组合方式**ULT/KST**线程；
 - 具备**ULT**和**KST**的全部优点；
 - 内核支持多线程的创建、调度和管理；
 - 同时系统提供了线程库，允许用户程序管理用户级的线程；
 - 如**Solaris**操作系统，就是将两种方式结合起来的系统
- 由于用户级线程和内核支持线程连接方式不同，形成了三种不同的模型：

四、线程的实现方式（6）

■ 多对一模型

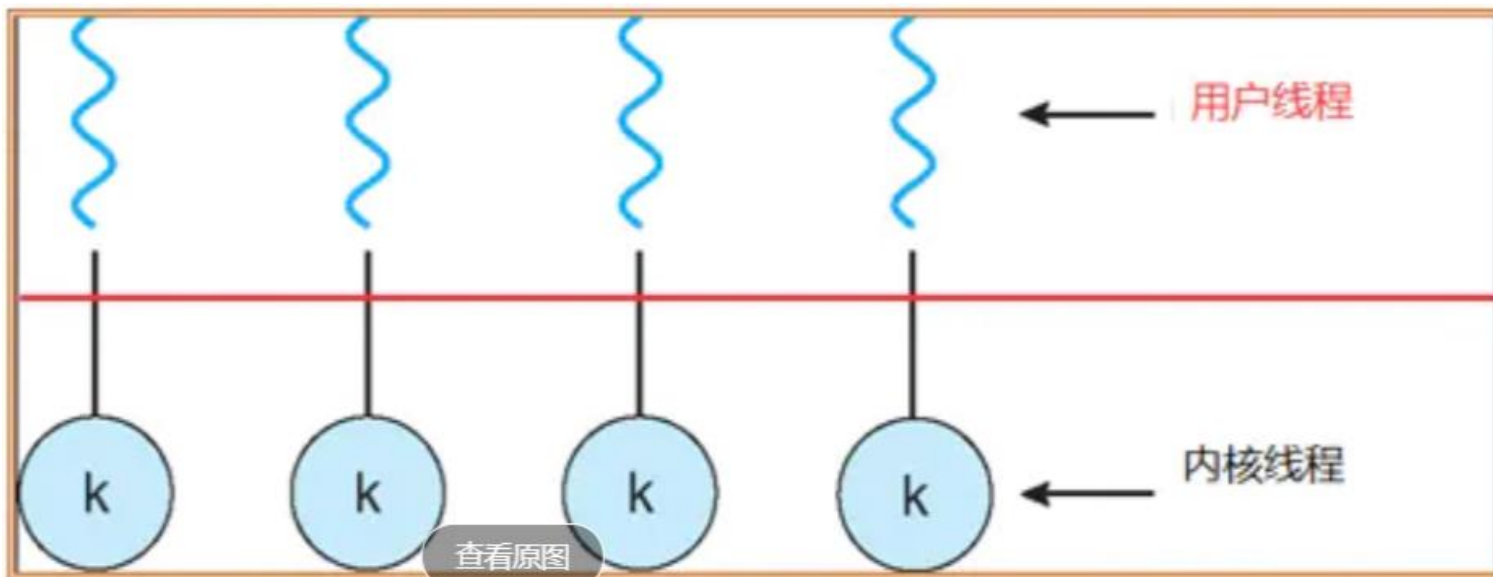
- 将用户线程映射到一个内核控制线程；
- 这些用户线程一般属于一个进程，运行在该进程的用户空间，对它们的调度和管理也是在该进程的用户空间中完成；
- 仅当用户线程需要访问内核时，才将它映射到一个内核控制线程上；



四、线程的实现方式（7）

■ 一对一模型

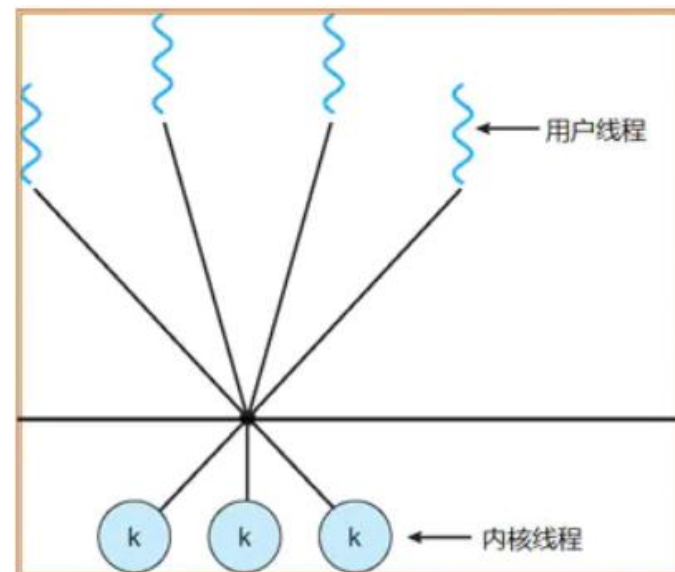
- 将每个用户级线程映射到一个内核支持线程；
- 为每一个用户线程都设置一个内核支持线程与之相连接；



四、线程的实现方式（8）

■ 多对多模型

- 将许多用户线程映射到同样数量或更少数量的内核线程上；
- 内核支持线程的数目可以根据应用进程和系统的不同而变化，可以等于或少于用户线程；
- 可以像一对一模型，使一个进程的多个线程并行地运行在多台处理机系统上；
- 可以像多对一模型，减少线程的管理开销和提高效率。

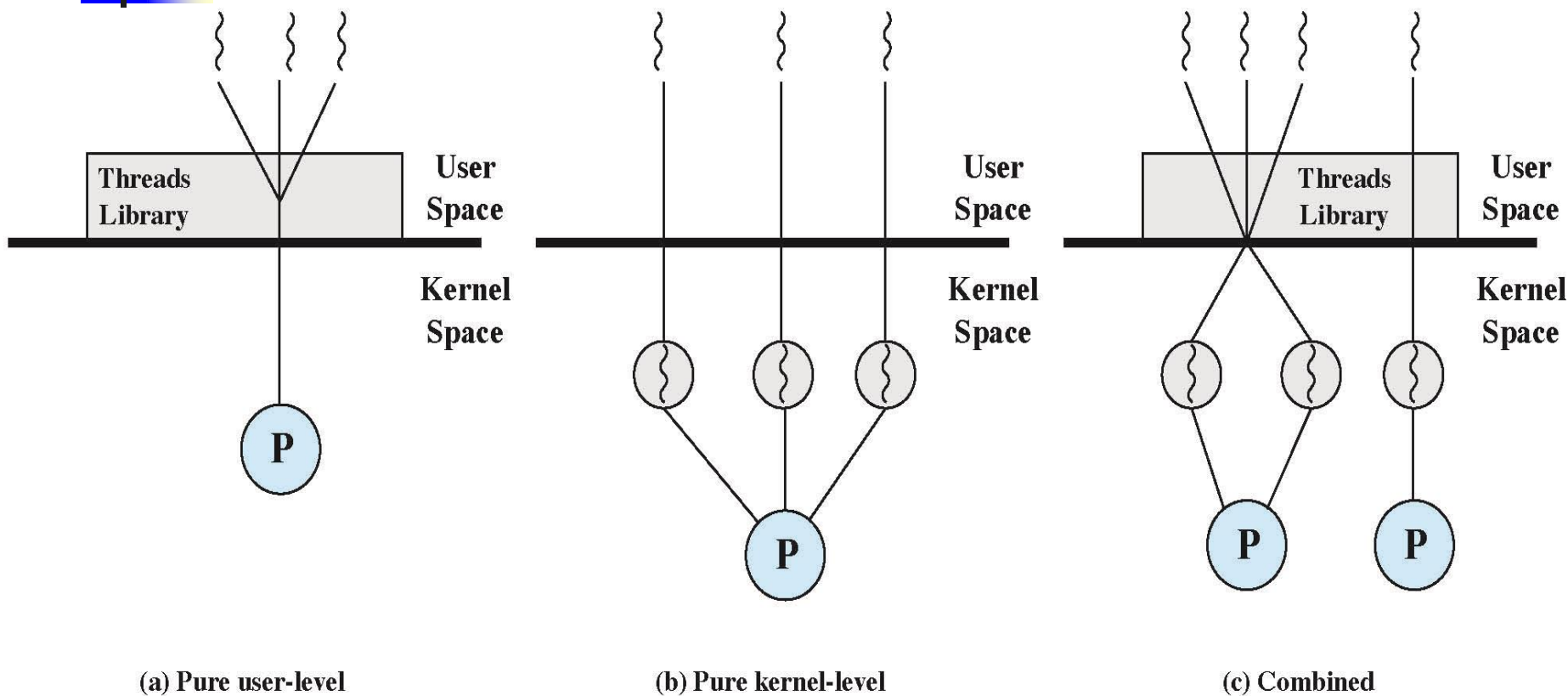




四、线程的实现方式（9）

	内核支持线程 KST	用户级线程 ULT
管理	由操作系统内核进行管理。	全部由用户程序完成，操作系统内核只对进程进行管理
调度单位	线程	进程
切换速度	由内核完成，速度较慢	同一进程各线程间切换无需内核支持，速度较快
系统调用行为	内核只看作该线程的行为	内核看作是整个用户进程的行为
阻塞	线程	用户进程
优点	对多处理器，可同时调度同一进程的多个线程，速度较快；阻塞是在线程一级	线程切换不调用内核，切换速度较快；调度算法可由应用程序定
缺点	同一进程内的线程切换速度慢	阻塞在用户进程一级

四、线程的实现方式（10）



} User-level thread (wavy circle) Kernel-level thread (P) Process



五、线程的创建和终止

■ 线程的创建

- 应用程序在启动时，通常仅有一个线程在执行（“初始化线程”），它的主要功能是用于创建新线程（利用线程创建函数或系统调用）

■ 线程的终止

- 由终止线程通过调用相应的函数或系统调用对运行完的线程执行终止操作；
- 有些线程（主要是系统线程）一旦被建立便一直运行下去而不被终止；
- 大多数**OS**中线程被终止后并不立即释放所占的资源。