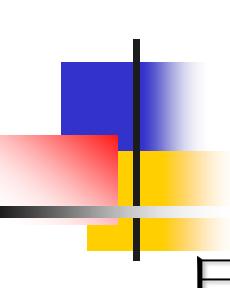


Chap2 进程管理

- 内容

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



Chap2 进程管理

■ 目的及要求

- 领会程序顺序执行和并发执行的特征；
- 理解和掌握进程的概念和特征、进程的基本状态及转换
- 理解进程控制块的作用、包含信息和组织方式；
- 领会引起进程创建、终止、阻塞、唤醒、挂起、激活的过程和各类进程控制原语的执行过程；
- 了解线程的基本概念，理解线程与进程的联系与区别。
- 理解临界资源和临界区的概念，领会进程同步机制应遵循的准则；
- 理解和掌握整型信号量和记录型信号量机制；
- 熟练掌握利用信号量机制解决进程同步问题；
- 了解进程通信的类型，理解消息传递系统中的发送和接收原语。



Chap2 进程管理

■ 重点

- 程序并发执行的特征；
- 进程的概念和特征；
- 进程的基本状态及其转换；
- 临界资源和临界区的概念；
- 记录型信号量机制；
- 利用信号量机制解决进程同步问题；
- 线程与进程的比较。

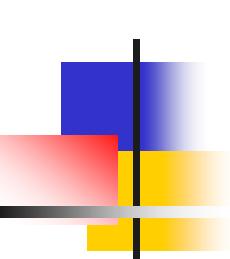
■ 难点

- 程序的顺序执行和并发执行比较；
- 进程和程序的联系与区别；
- 利用信号量机制解决经典进程同步与互斥问题；
- 线程与进程的比较。



Chap2 进程管理

- 内容
 - § 2.1 前趋图和程序执行
 - § 2.2 进程的描述
 - § 2.3 进程控制
 - § 2.4 进程同步
 - § 2.5 经典进程的同步问题
 - § 2.6 进程通信
 - § 2.7 线程的基本概念
 - § 2.8 线程的实现



2.1 前趋图和程序执行

操作系统的特性之一是并发与共享，即在系统中（内存）同时存在几个相互独立的程序，这些程序在系统中既交叉地运行，又要共享系统中的资源，这就会引起一系列的问题，包括：对资源的竞争、运行程序之间的通信、程序之间的合作与协同等。

要解决这些问题，用程序的概念已经不能描述程序在内存中运行的状态，必须引入新的概念——进程。

2.1 前趋图和程序执行

2.1.1 前趋图

前趋图是一个有向无循环图(DAG)，用于描述进程之间执行的前后关系

结点：描述一个程序段或进程，或一条语句

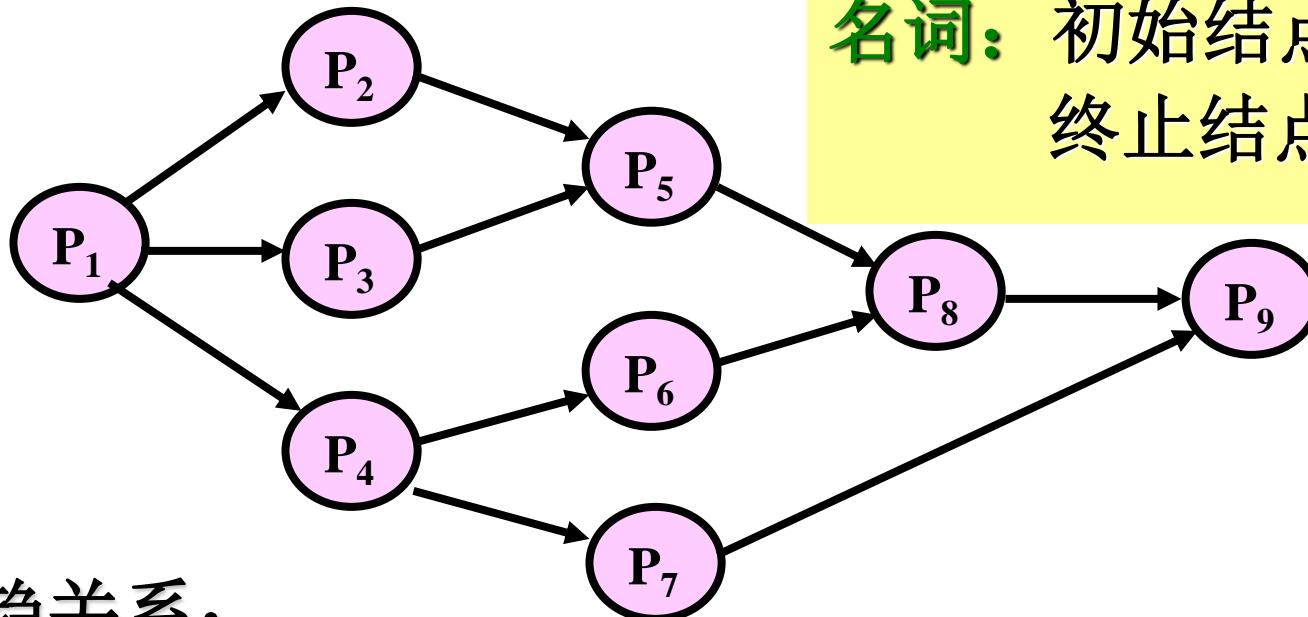
有向边：结点之间的偏序或前序关系“ \rightarrow ”

$\rightarrow = \{(P_i, P_j) | P_i \text{ must complete before } P_j \text{ may start}\}$

若 $(P_i, P_j) \in \rightarrow$, 记为 $P_i \rightarrow P_j$, 则

P_i 是 P_j 的直接前趋, P_j 是 P_i 的直接后继

2.1 前趋图和程序执行



名词: 初始结点;
终止结点; 重量

前趋关系:

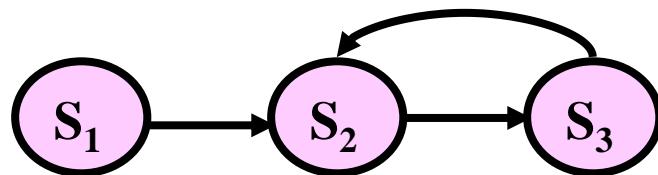
$P_1 \rightarrow P_2, P_1 \rightarrow P_3, P_1 \rightarrow P_4, P_2 \rightarrow P_5, P_3 \rightarrow P_5, P_4 \rightarrow P_6,$
 $P_4 \rightarrow P_7, P_5 \rightarrow P_8, P_6 \rightarrow P_8, P_7 \rightarrow P_9, P_8 \rightarrow P_9$

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

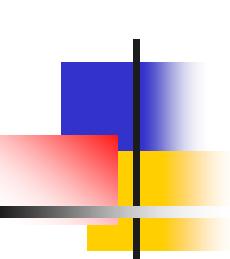
$\rightarrow = \{(P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3, P_5),$
 $(P_4, P_6), (P_4, P_7), (P_5, P_8), (P_6, P_8), (P_7, P_9), (P_8, P_9)\}$

2.1 前趋图和程序执行

注意：前趋图中必须不存在循环



前趋关系： $S_2 \rightarrow S_3$, $S_3 \rightarrow S_2$, 不可能满足



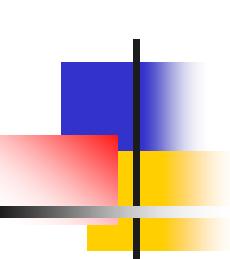
Chap2 进程管理

§ 2.1 前趋图和程序执行

2.1.1 前驱图

2.1.2 程序的顺序执行

2.1.3 程序的并发执行



2.1 前趋图和程序执行

首先描述程序的顺序和并发执行方式

2.1.2 程序的顺序执行

1. 程序的顺序执行

一个程序由若干个程序段组成，而这些程序段的执行必须是顺序的，这种程序执行的方式就称为程序的顺序执行。

2.1 前趋图和程序执行

例：讨论单道系统的工作情况

对用户作业的处理——

首先输入用户的程序和数据 **Input**

然后进行计算 **Caculate**

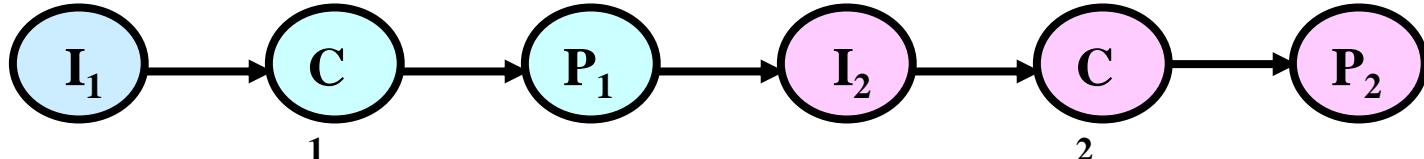
最后打印计算结果 **Print**

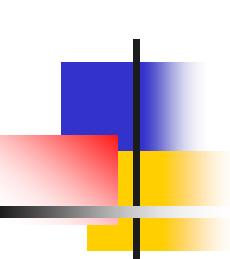
即有三个顺序执行的操作——

I：输入操作

C：计算操作

P：输出操作





2.1 前趋图和程序执行

2. 程序顺序执行时的特征

(1) 顺序性

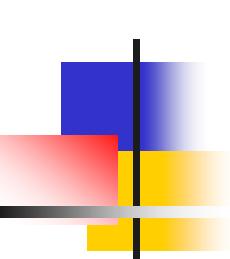
处理机的操作严格按照程序所规定的顺序执行。

(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。

(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。



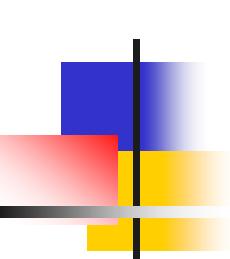
Chap2 进程管理

§ 2.1 前趋图和程序执行

2.1.1 前驱图

2.1.2 程序的顺序执行

2.1.3 程序的并发执行



2.1 前趋图和程序执行

2.1.3 程序的并发执行

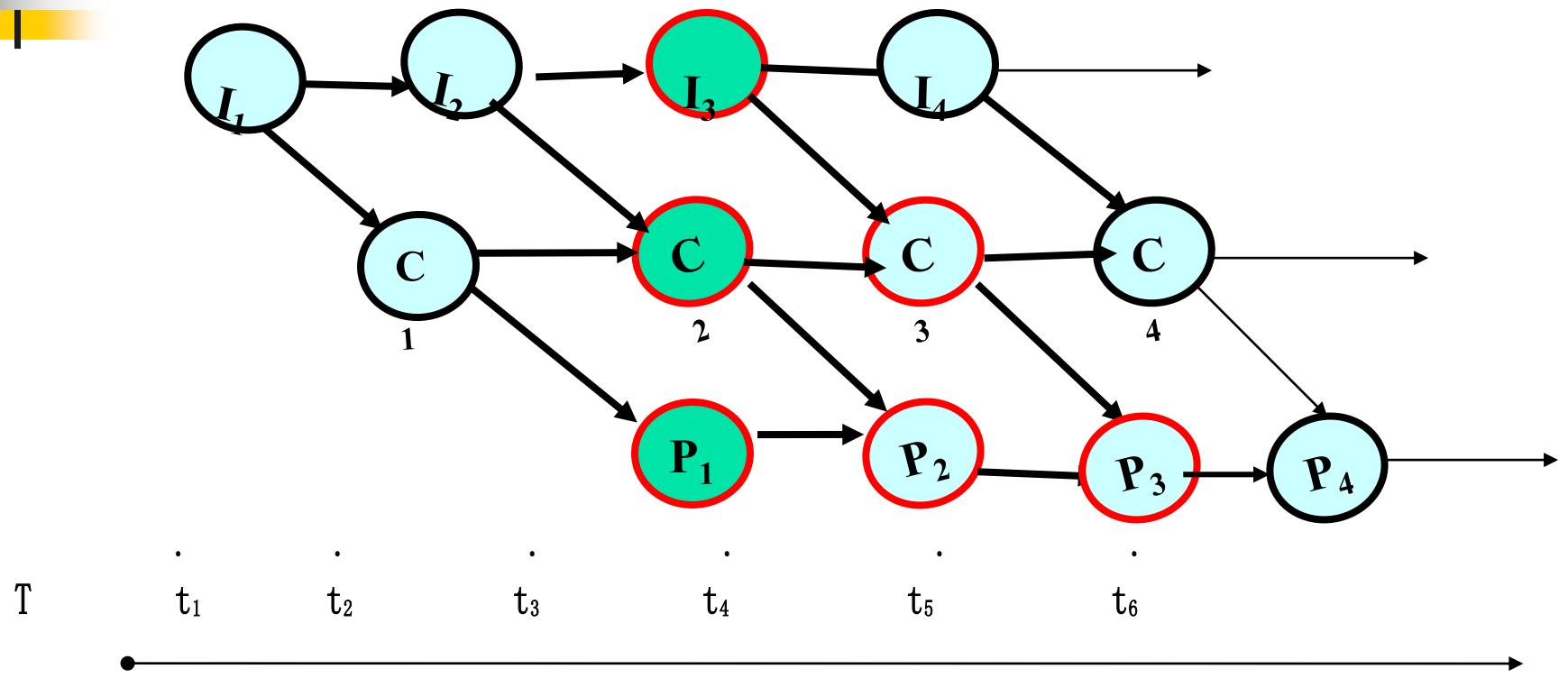
1. 程序的并发执行

例：

在系统中有n个作业，每个作业都有三个处理步骤，

输入数据、处理、输出，即 I_i, C_i, P_i
 $(i=1, 2, 3, \dots, n)$ 。

2.1.3 程序的并发执行



例如：

I_1 、 C_1 、 P_1 的执行必须严格按照 I_1 ， C_1 ， P_1 的顺序；

而 C_1 与 I_2 ， P_1 与 I_3 ， I_3 与 C_2 是可以同时执行的。

2.1.3 程序的并发执行

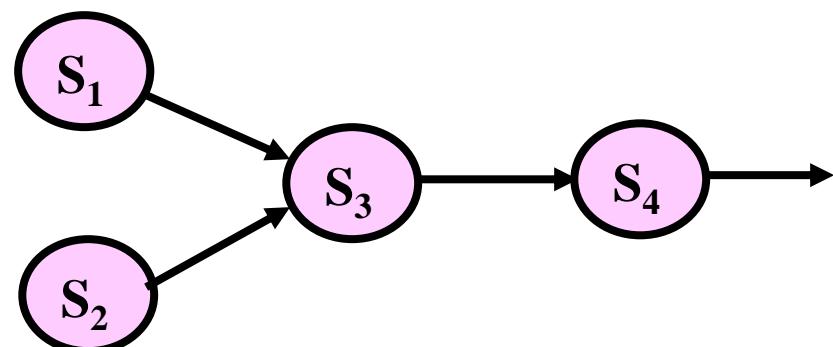
又如：四个程序段

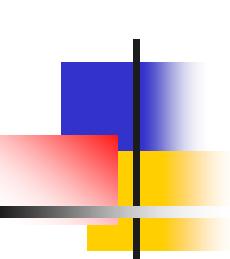
$S_1: a := x + 2$

$S_2: b := y + 4$

$S_3: c := a + b$

$S_4: d := c + b$





2.1.3 程序的并发执行

1. 程序的并发执行
 2. 程序并发执行时的特征
- (1) 间断性

在多道程序设计的环境下，程序的并发执行，以及为完成一项任务而相互合作，这些程序之间要共享系统的资源，形成了相互制约的关系。

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。

2.1.3 程序的并发执行

2. 程序并发执行时的特征

(1) 间断性

(2) 失去封闭性

程序在并发执行时，

改变，程序运行失去封闭性。一程序的运行受到其他程序的影响。

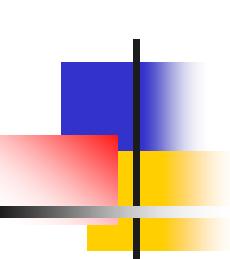
(3) 不可再现性

程序在并发执行时，多次运行初始条件相同的同一程序会得出不同的运行结果。

例：共享公共变量的两个程序，它们执行时可能产生不同结果。

程序顺序执行时的特征

- (1) 顺序性
- (2) 封闭性
- (3) 可再现性



并发程序失去可再现性的例子

例：讨论共享公共变量的两个程序，它们执行时可能产生的不同结果。

n:=0

程序A

...

n := n+1;

...

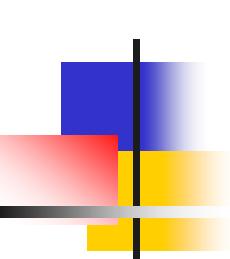
程序B

...

print(n);

n := 0;

...



Chap2 进程管理

§ 2.1 前趋图和程序执行

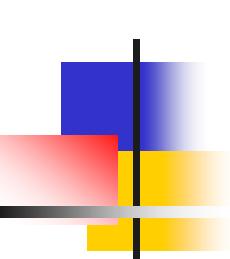
§ 2.2 进程的描述

2.2.1 进程的定义和特征

2.2.2 进程的基本状态及转换

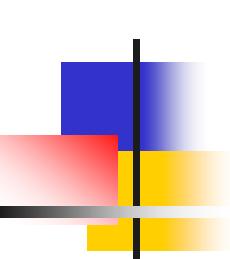
2.2.3 挂起操作和进程状态的转换

2.2.4 进程管理中的数据结构



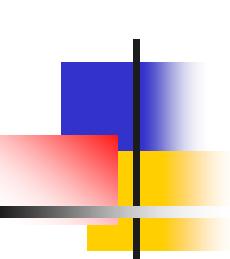
2.2 进程的描述

1. 进程的特征和定义
2. 进程的三种基本状态
3. 挂起状态
4. 进程控制块



2.2 进程的描述

1. 进程的特征和定义
2. 进程的三种基本状态
3. 挂起状态
4. 进程控制块



2.2 进程的描述

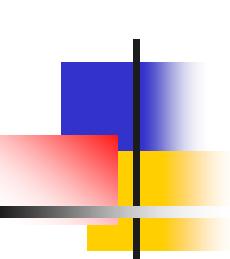
2.2.1 进程的定义和特征

1. 进程的特征和定义

在多道程序设计的环境下，为了描述程序在计算机系统内的执行情况，必须引入新的概念——进程。

1) 进程的定义

进程：程序关于某个数据集合的一次执行过程



2.1 进程的描述

进程的其它定义

- 行为的一个规则叫做程序，程序在处理机上执行时所发生的活动称为进程（Dijkstra）。
- 进程是这样的计算部分，它是可以和其它计算并行的一个计算。（Donovan）
- 进程（有时称为任务）是一个程序与其数据一道通过处理机的执行所发生的活动。（Alan.C. Shaw）
- 进程是执行中的程序。（Ken Thompson and Dennis Ritchie）

2.2 进程的描述

1. 进程的特征和定义

1) 进程的定义

2) 进程的特征（与程序比较）

(1) 结构特征

进程控制块(PCB) + 程序 + 数据 = 进程实体

(2) 动态性——最基本特征

进程：进程实体的一次执行过程，有生命周期。

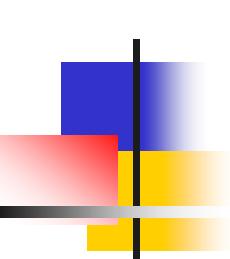
程序：程序是一组有序指令的集合，是静态的概念。

(3) 并发性

(4) 独立性

(5) 异步性

进程按各自独立的、不可预知的速度向前推进



2.2 进程的描述

1. 进程的特征和定义
2. 进程的三种基本状态
3. 挂起状态
4. 进程控制块

2.2 进程的描述

2. 进程的三种基本状态

(1) 就绪状态(Ready)

进程已获得除CPU之外的所有必需的资源，一旦得到CPU控制权，立即可以运行。

(2) 运行状态(Running)

进程已获得运行所必需的资源，它的程序正在处理机上执行。

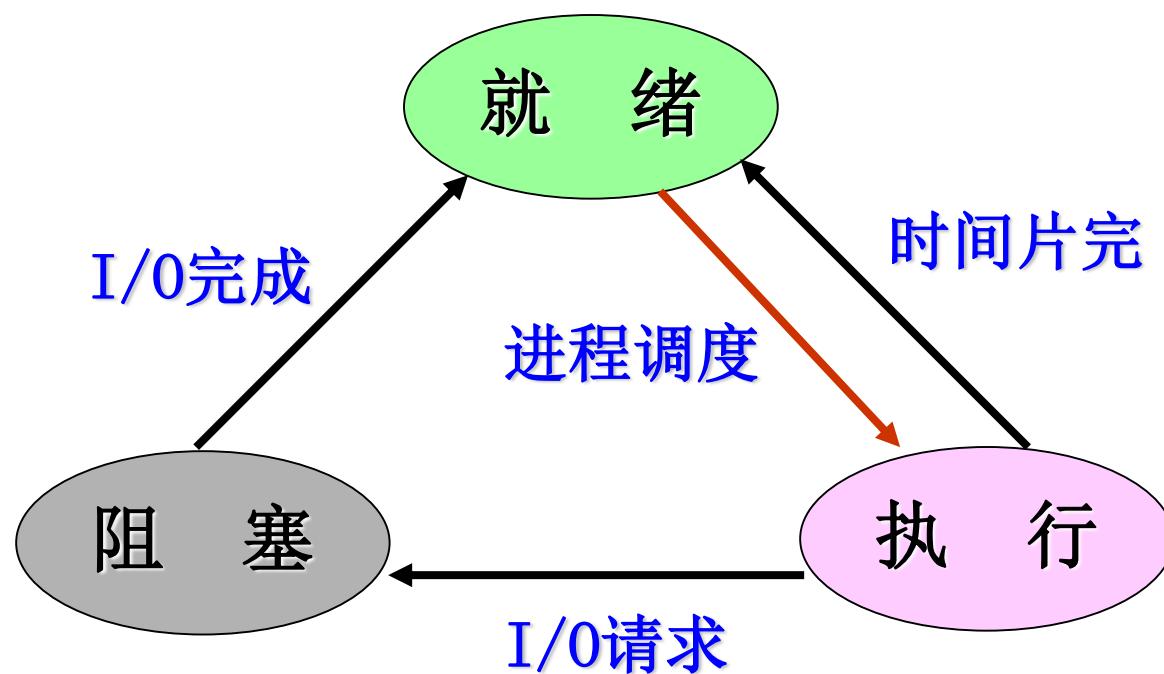
(3) 阻塞状态(Blocked)

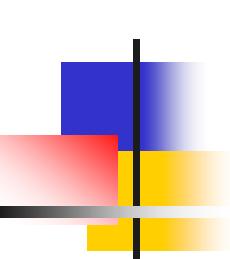
正在执行的进程由于发生某事件而暂时无法执行时，便放弃处理机而处于暂停状态，称该进程处于阻塞状态或等待状态。

就绪队列与阻塞队列

2.2 进程的描述

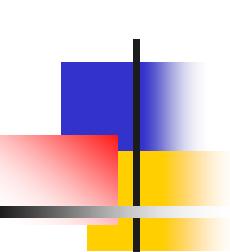
进程的三种基本状态以及各状态之间的转换关系





2.2 进程的描述

1. 进程的特征和定义
2. 进程的三种基本状态
3. 挂起状态
4. 进程控制块



2.2 进程的描述

3. 挂起状态

1) 引起挂起状态的原因:

- 终端用户的请求
- 父进程请求
- 负荷调节的需要
- 操作系统的需要

2) 进程状态的转换

引入挂起状态后，增加了挂起状态(静止状态)到非挂起状态(活动状态)的转换，或者相反。

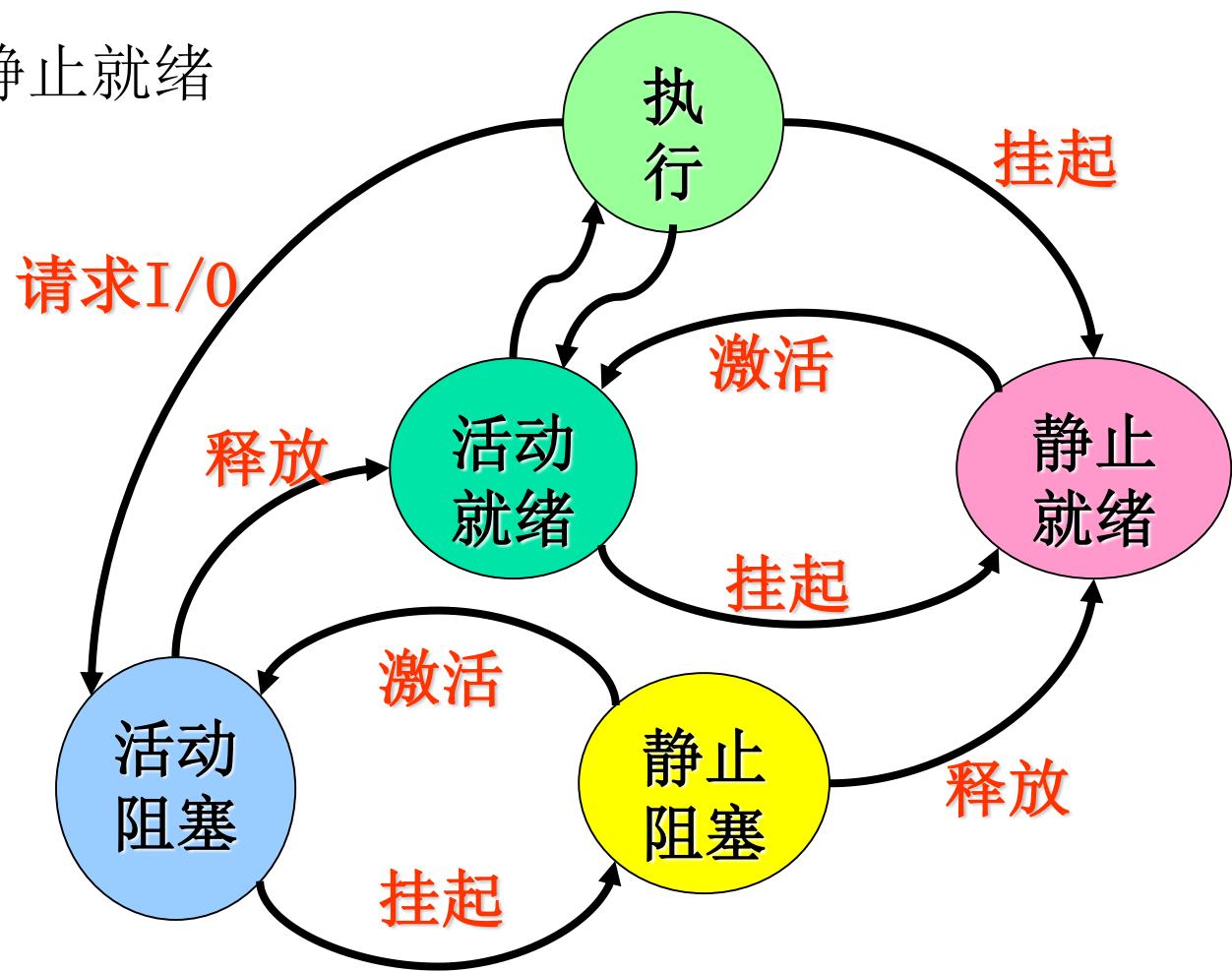
2.2 进程的描述

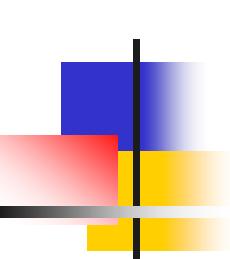
1. 活动就绪 → 静止就绪

2. 活动阻塞
→ 静止阻塞

3. 静止就绪 →
活动就绪

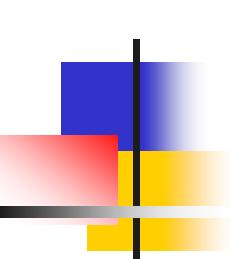
4. 静止阻塞 →
活动阻塞





2.2 进程的描述

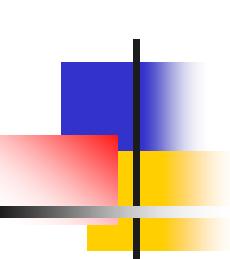
1. 进程的特征和定义
2. 进程的三种基本状态
3. 挂起状态
4. 进程控制块



2.2 进程的描述

2.2.4 进程管理中的数据结构

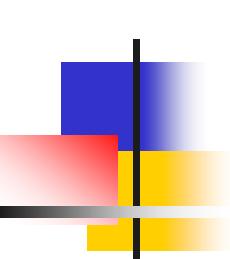
1. 进程控制块（PCB）的作用
2. 进程控制块中的信息
3. 进程控制块的组织方式



2.2 进程的描述

2.2.4 进程管理中的数据结构

1. 进程控制块（PCB）的作用
2. 进程控制块中的信息
3. 进程控制块的组织方式



2.2 进程描述

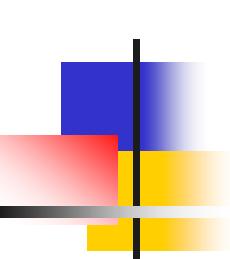
2.2.4 进程管理中的数据结构

1. 进程控制块的作用

存放进程管理和控制信息的数据结构称为进程控制块。它是进程管理和控制的最重要的数据结构，在创建时，建立PCB，并伴随进程运行的全过程，直到进程撤销而撤销。PCB就象我们的户口。

PCB是进程存在的唯一标志。

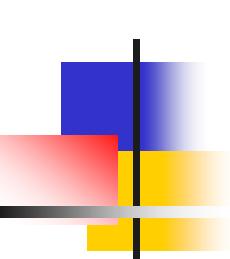
系统的所有PCB组织成链表或队列，常驻内存的PCB区。



2.2 进程的描述

2.2.4 进程管理中的数据结构

1. 进程控制块（PCB）的作用
2. 进程控制块中的信息
3. 进程控制块的组织方式



2.2.4 进程管理中的数据结构

2. 进程控制块中的信息

1) 进程标示符

每个进程都必须有一个唯一的标识符

- 内部标示符
- 外部标示符

2) 处理机状态

处理机状态信息主要由处理机的各种寄存器中的内容组成。处理机运行时的信息存放在寄存器中，当被中断时这些信息要存放在PCB中。

2.2.4 进程管理中的数据结构

2. 进程控制块中的信息

1) 进程标识符

2) 处理机状态

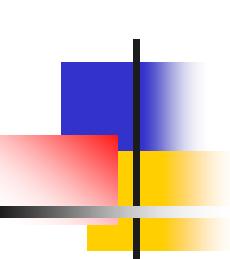
- 通用寄存器
- 指令计数器
- 程序状态字 PSW
- 用户栈指针

3) 进程调度信息

- 进程状态
- 进程优先级
- 进程调度所需的其他信息
- 事件

4) 进程控制信息

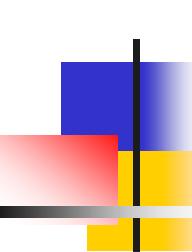
- 程序和数据的地址
- 进程通信和同步机制
- 资源清单
- 链接指针



2.2 进程的描述

2.2.4 进程管理中的数据结构

1. 进程控制块（PCB）的作用
2. 进程控制块中的信息
3. 进程控制块的组织方式



2.2.4 进程管理中的数据结构

3. 进程控制块的组织方式

1) 线性方式

2) 链接方式

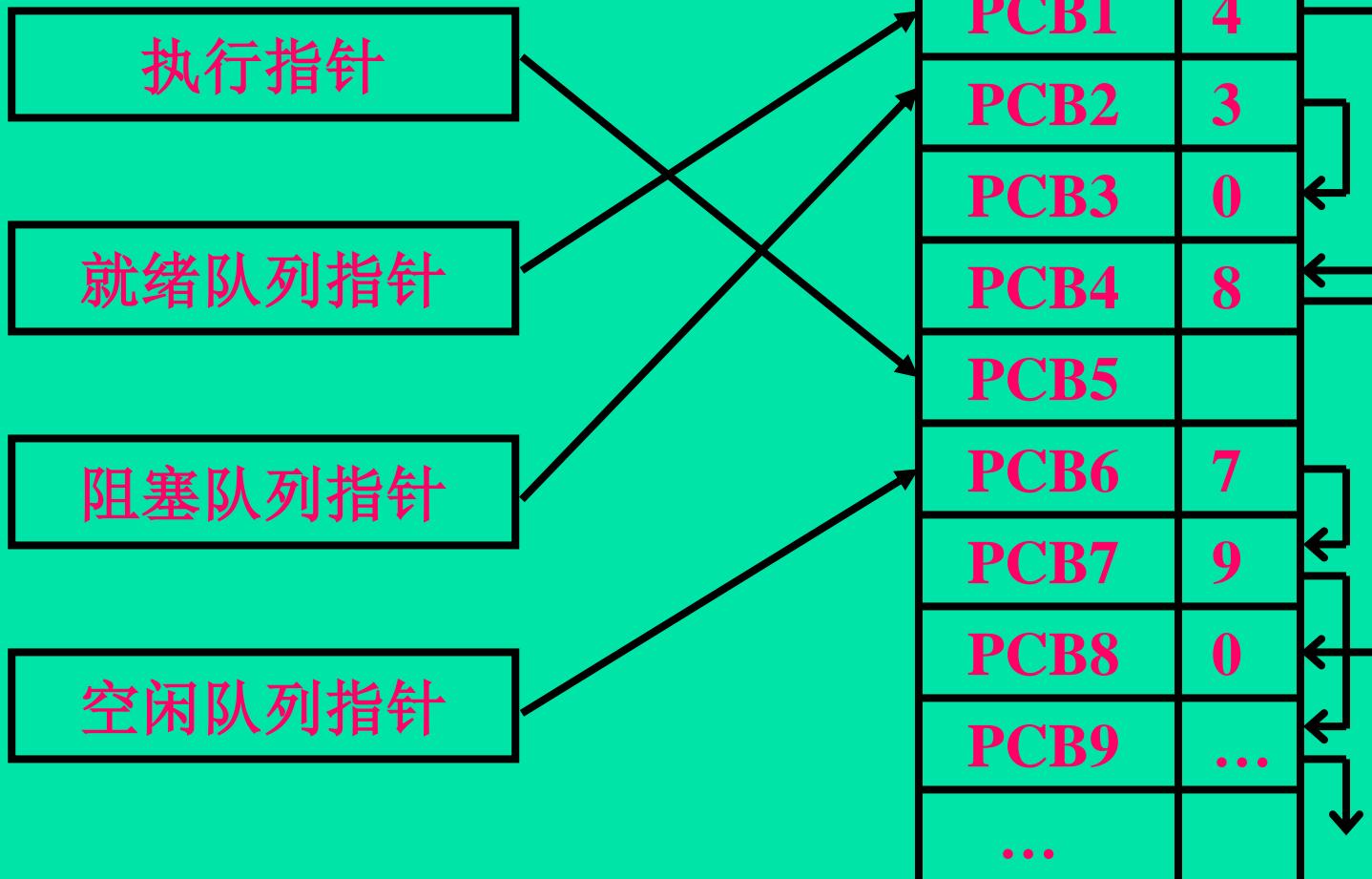
把具有同一状态的PCB用其中的链接字链接成一个队列。

就绪队列；若干个阻塞队列；

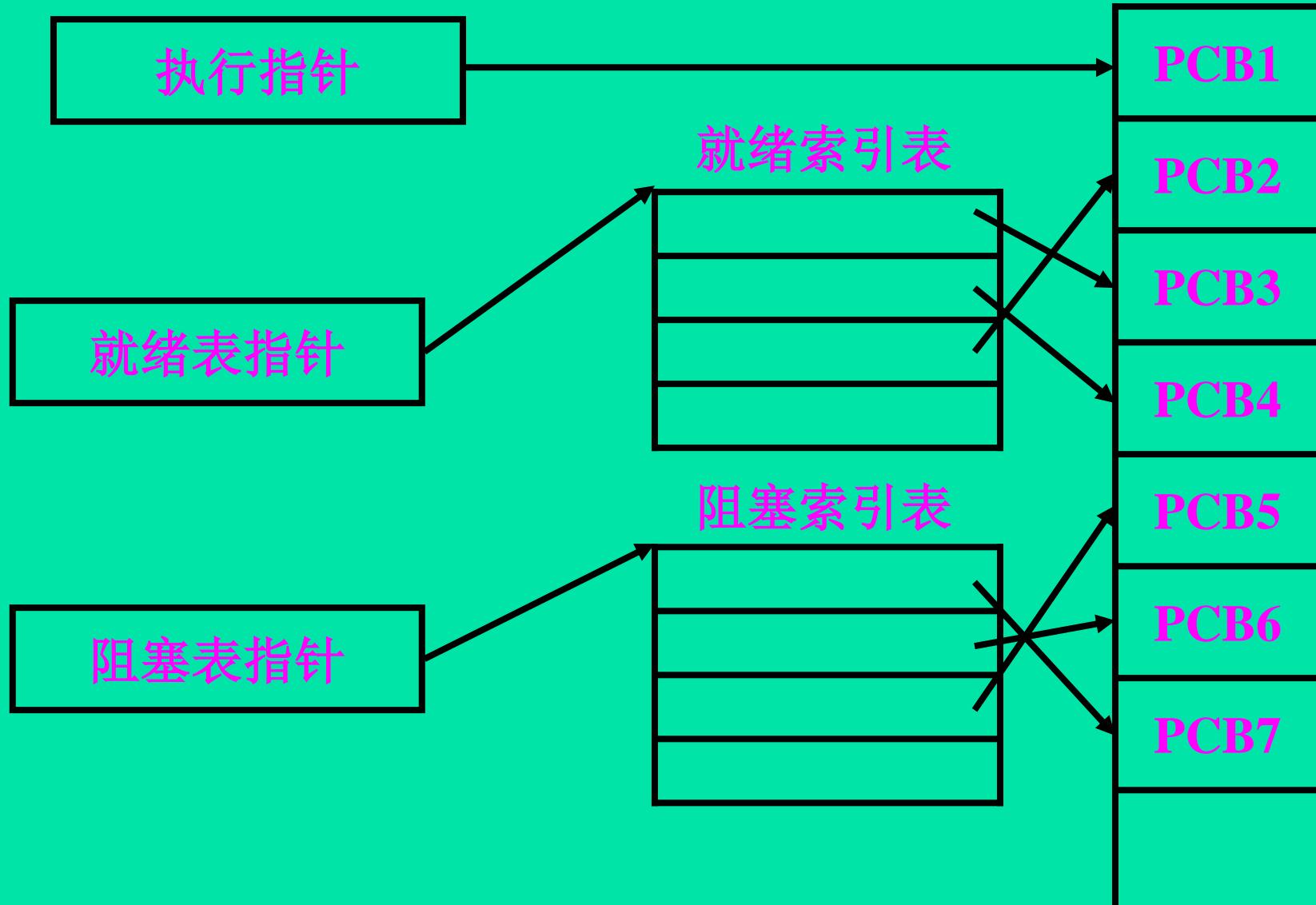
3) 索引方式

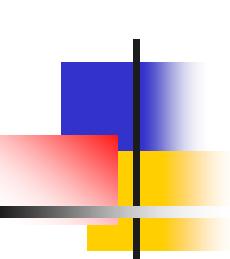
系统根据所有进程的状态建立几张索引表，把各表的内存首地址记录在内存的专用单元中。索引表的表目中记录了相应状态的某个PCB在PCB表中的地址。

PCB的链接组织方式:



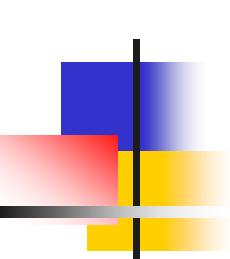
PCB的索引组织方式:





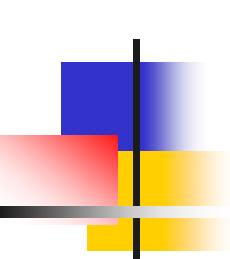
Chap2 进程管理

- 内容
 - § 2.1 前趋图和程序执行
 - § 2.2 进程的描述
 - § 2.3 进程控制
 - § 2.4 进程同步
 - § 2.5 经典进程的同步问题
 - § 2.6 进程通信
 - § 2.7 线程的基本概念
 - § 2.8 线程的实现



2.3 进程控制

- 2.3.1 操作系统内核
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活



2.3 进程控制

- 2.3.1 操作系统内核
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活

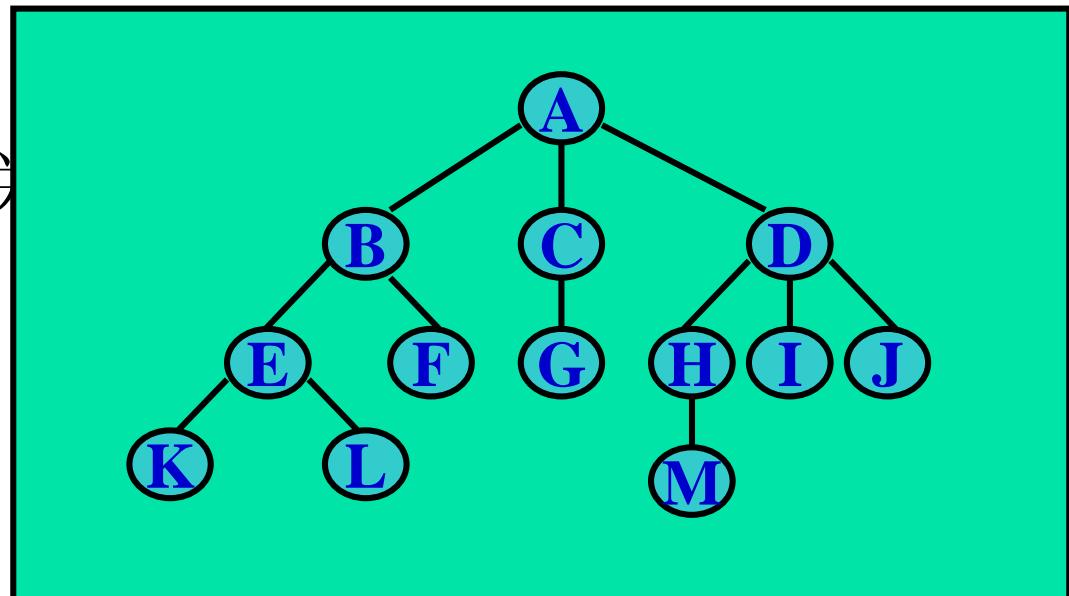
2.3 进程控制

对系统中的全部进程实施有效的管理，负责进程状态的改变。

2.3.2 进程的创建

1. 进程图

描述进程的家族关系



2.3.2 进程的创建

1. 进程图
2. 引起创建进程的事件

多道程序环境中，只有进程才能在系统中运行。

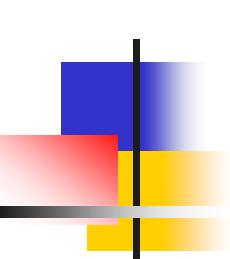
- 1) 用户登录;
- 2) 作业调度
- 3) 提供服务;
- 4) 应用请求

3. 进程的创建

操作系统发现要求创建新进程的事件后，调用进程创建原语Create()创建新进程。

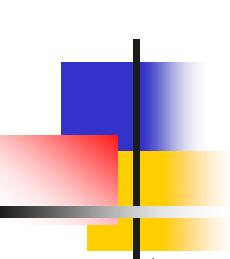
进程的创建过程：

申请空白PCB → 为新进程分配资源 →
初始化进程控制块 → 将新进程插入就绪队列



2.3 进程控制

- 2.3.1 操作系统内核
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活



2.3.3 进程的终止

1. 引起进程终止的事件
 - 1) 正常结束
 - 2) 异常结束

越界错误；非法指令 等
 - 3) 外界干预

操作员或操作系统干预；
父进程请求；
父进程终止

2.3.3 进程的终止

1. 引起进程终止的事件
2. 进程的终止过程

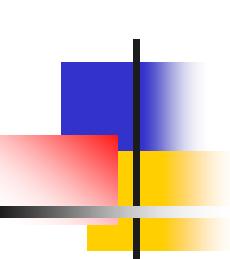
找出被终止进程的PCB →

若进程状态为运行态，置CPU调度标志为真 →

若其有子孙进程，终止其子孙进程并回收其资源 →

回收终止进程的资源 →

回收终止进程的PCB



2.3 进程控制

- 2.3.1 操作系统内核
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活

2.3.4 进程的阻塞与唤醒

1. 引起进程阻塞和唤醒的事件
 - 1) 请求系统服务
 - 2) 启动某种操作
 - 3) 新数据尚未到达
 - 4) 无新工作可做

2. 进程阻塞过程

调用阻塞原语**阻塞自己**；

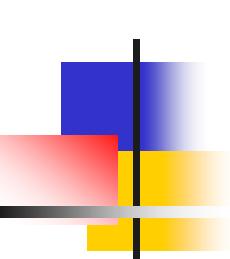
- 将PCB中的状态改为阻塞，并加入阻塞队列；
- 转进程调度。

3. 进程唤醒过程

阻塞进程等待的事件发生，**有关进程**调用唤醒原语唤醒等待该事件的进程

- 把阻塞进程从等待该事件的阻塞队列中移出；
- 置进程状态为就绪态，将PCB插入到就绪队列中。

阻塞原语与唤醒原语作用相反，成对使用



2.3 进程控制

- 2.3.1 操作系统内核
- 2.3.2 进程的创建
- 2.3.3 进程的终止
- 2.3.4 进程的阻塞与唤醒
- 2.3.5 进程的挂起与激活

2.3.5 进程的挂起与激活

1. 进程的挂起过程

检查被挂起进程的状态：

当出现引起进程挂起的事件时，系统利用**挂起原语**将指定进程或处于阻塞的进程挂起。

若处于活动就绪，则改为静止就绪；

若处于活动阻塞，则改为静止阻塞；

若挂起的进程正在执行，则重新进行进程调度。

2. 进程的激活过程

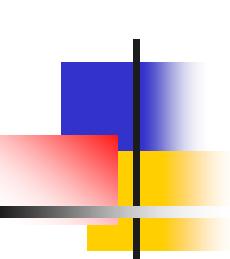
1) 激活原语先将进程从外存调入内存；

2) 检查该进程的状态：

若为静止就绪，则改为活动就绪；

若处于静止阻塞，则改为活动阻塞。

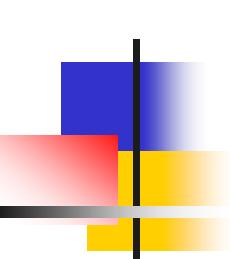
当发生激活进程的事件时，系统利用**激活原语**将指定进程激活。



Chap2 进程管理

■ 内容

- § 2.1 前趋图和程序执行
- § 2.2 进程的描述
- § 2.3 进程控制
- § 2.4 进程同步
- § 2.5 经典进程的同步问题
- § 2.6 进程通信
- § 2.7 线程的基本概念
- § 2.8 线程的实现

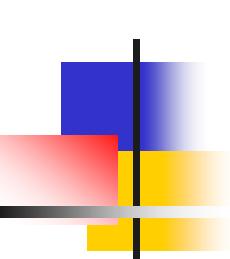


2.4 进程同步

- 2.4.1 进程的同步基本概念
- 2.4.3 信号量机制
- 2.4.4 信号量的应用

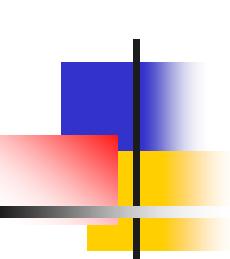
进程同步的主要任务：

使并发执行的诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。



2.4 进程同步

- 2.4.1 进程的同步基本概念
- 2.4.3 信号量机制
- 2.4.4 信号量的应用



2.4.1 进程同步的基本概念

1. 进程间两种形式的制约关系

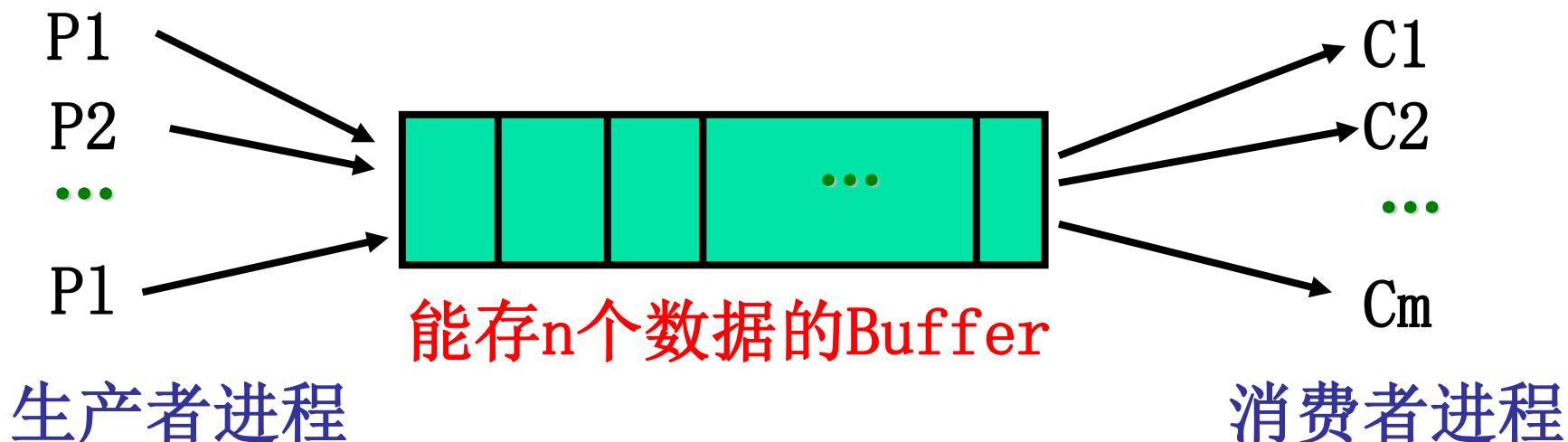
- (1) 间接相互制约关系 --- 源于资源共享
- (2) 直接相互制约关系 --- 源于进程合作

例如

2.4.1 进程同步的基本概念

1. 两种形式的制约关系
2. 临界资源 --- **互斥**访问

生产者—消费者问题：



生产者进程和消费者进程都以异步方式运行，
但它们之间必须保持同步。

注意：缓冲池组织为循环缓冲，输入指针加1表示为出产品，输出指针加1表示为入产品

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，初值均为**0**。

生产者逻辑

```
item buf[10];  
int in=0, out=0,  
counter=0;
```

出产品使其减1

2.4.1 进程同步的基本概念

生产者—消费者问题：

```
Void Producer(){  
while(1){  
    produce an item  
    in nextp;  
    ...  
    while (counter==n)  
        do no-op;  
    buffer[in] =nextp;  
    in=(in+1) % n;  
    counter++;  
}  
}
```

```
Void Consumer(){  
while(1){  
    while (counter== 0)  
        do no-op;  
    nextc=buffer[out];  
    out=(out+1) % n;  
    counter--;  
    consumer the item  
    in nextc;  
}  
}
```

重复的测试条件

空操作指令

并发操作时会出错

把一次仅允许一个进程访问的资源叫做临界资源

并发执行出错—**counter** (初值为5)

生产者对它加1， 消费者对它减1

机器语言实现：

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

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

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

(register1=5)
(register1=6)
(register2=5)
(register2=4)
(counter=6)
(counter=4)

counter应当为5

解决方法：把**counter**作为临界资源， 对其互斥访问

2.4.1 进程同步

1. 两种形式的制约关系
2. 临界资源 --- 互斥访问
3. 临界区

临界区：每个进程中访问临界资源

访问临界区的程序设计为：

对欲访问的临界资源进行检查，

若此刻未被访问，设正在访问的标志

.....进入区

访问临界资源

.....临界区

将正在访问的标志恢复为未被访问的标志

.....退出区

其余部分

.....剩余区

进程互斥：两进程不能同时进入访问同一临界资源的临界区

While(true)

{

entry section

critical section

exit section

remainder section

}

2.4.1 进程同步的基本概念

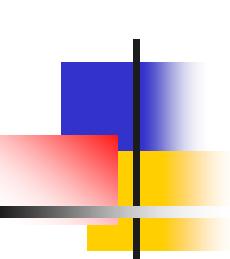
1. 两种形式的制约关系
2. 临界资源 --- **互斥**访问
3. 临界区

进程同步: 某进程未获得合作进程发来消息之前该进程等待，消息到来之后方可继续执行的合作关系

同步机构(同步机制): 系统用来实现进程间同步与互斥的机构

4. 同步机制应遵循的规则

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待



2.4 进程同步

- 2.4.1 进程同步的基本概念
- 2.4.3 信号量机制
- 2.4.4 信号量的应用

2.4.3 信号量机制

1965年荷兰Dijkstra提出的进程同步工具

1. 整型信号量

定义：整型量，除初始化外，仅能通过两个原子操作
来访问

- P操作 wait(S):

P(S)

While ($S \leq 0$) do no-op;

$S--;$

- V操作 signal(S):

V(S)

$S++;$

P、V操作是原子操作，不可中断。

```
semaphore mutex =1;
```

```
begin  
parbegin
```

```
process 1: begin
```

```
repeat
```

```
    wait(mutex);
```

```
    critical section
```

```
    signal(mutex);
```

```
    remainder section
```

```
until false;
```

```
end
```

```
process 2: begin
```

```
repeat
```

```
    wait(mutex);
```

```
    critical section
```

```
    signal(mutex);
```

```
    remainder section
```

```
until false;
```

```
end
```

```
parend
```

用信号量机制实现互斥的模式

wait(mutex)和
signal(mutex)必
须成对出现

2.4.3 信号量机制

1965年荷兰Dijkstra提出的进程同步工具

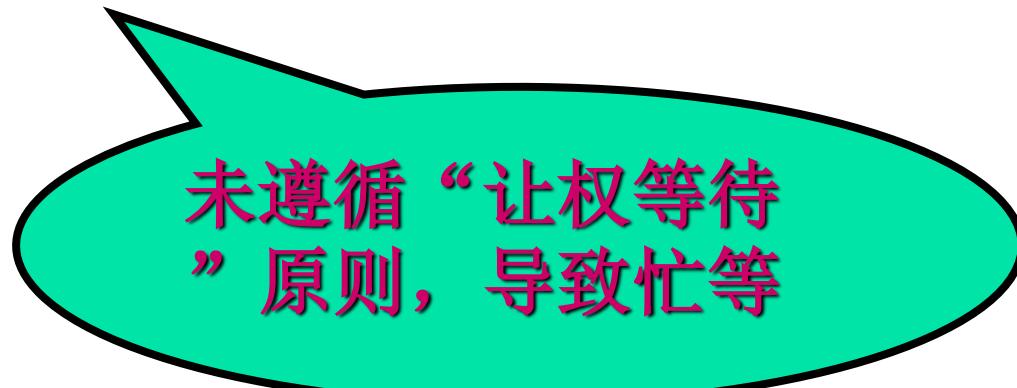
1. 整型信号量

定义：整型量，除初始化外，仅能通过两个原子操作
来访问

- P操作 wait(S):

```
While (S<=0) do no-op;
```

```
S--;
```



未遵循“让权等待”原则，导致忙等

- V操作 signal(S):

```
S++;
```

P、V操作是原子操作，不可中断。

2.4 进程同步

1. 整型信号量
2. 记录型信号量

引入整型变量value(代表资源数目)、进程链表List(链接所有等待进程)

记录型数据结构:

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

2.4 进程同步

1. 整型信号量
2. 记录型信号量

正确使用时能实现同步和互斥
含义：
信号量>0，代表可用资源的数量
信号量<0，代表由于申请信号量
代表的资源而阻塞的进程数量

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

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

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

```
semaphore mutex =1;
```

```
begin  
parbegin
```

```
process 1: begin
```

```
repeat
```

```
    wait(mutex);
```

```
    critical section
```

```
    signal(mutex);
```

```
    remainder section
```

```
until false;
```

```
end
```

```
process 2: begin
```

```
repeat
```

```
    wait(mutex);
```

```
    critical section
```

```
    signal(mutex);
```

```
    remainder section
```

```
until false;
```

```
end
```

```
parend
```

用信号量机制实现互斥的模式

wait(mutex)和
signal(mutex)必须
成对出现

2.4 进程同步

Process A: wait(Dmutex); 于是Dmutex=0

Process B: wait(Emutex); 于是Emutex=0

Process A: wait(Emutex); 于是Emutex=-1 A阻塞

Process B: wait(Dmutex); 于是Dmutex=-1 B阻塞

量Dmutex和Emutex，初值均为1。

Process A:

wait(Dmutex);

wait(Emutex);

使用D、E

Signal(Dmutex)

Signal(Emutex)

Process B:

wait(Emutex);

wait(Dmutex);

使用D、E

Signal(Dmutex)

Signal(Emutex)

设执行过程为

共享的资源越多，死锁的可能越大

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

```
{
```

```
  While(TRUE)
```

```
{
```

```
    if (S1 >= 1 and ... and Sn >= 1 ){
```

```
      for( i=1 ;i<=n; i++) Si --;
```

```
      break;
```

```
}
```

```
  else{
```

将进程放入第一个 $S_i < 1$ 的信号量对应的阻塞队列；且置程序计数器指向 Swait 的开始



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

```
}
```

```
}
```

```
}
```

2.3 进程同步

1. 整型信号量
2. 记录型信号量
3. AND型信号量

AND同步机制的基本思想：将进程在整个运行过程

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

将信号量S_i对应的阻塞队列中的所有进程置为就绪态

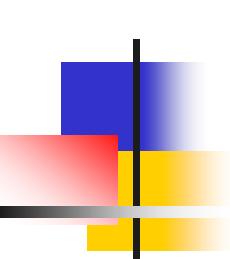
2.4 进程同步

```
Swait(S1, t1, d1; ...; Sn, tn, dn)
if S1>= t1 and ... and Sn>= tn then
    for i:=1 to n do
```

一般信号量集的几种特殊情况：

- **Swait(S, d, d)**, 只有一个信号量S, 允许每次申请d个资源, 若现有资源数少于d, 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)**, 当**S>=1时**, 允许多个进程进入某特定区, 当**S变为0后**, 阻止任何进程进入特定区, 相当于可控开关。

endfor



2.4 进程同步

- 2.4.1 进程同步的基本概念
- 2.4.3 信号量机制
- 2.4.4 信号量的应用

2.4.4 信号量的应用

1. 利用信号量实现进程互斥（模式）

为使多个进程互斥的访问某临界资源，须为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问资源的临界区CS置于wait(mutex)和signal(mutex)之间即可。

```
semaphore mutex =1;
```

```
begin  
parbegin
```

```
process 1: begin
```

```
repeat
```

```
    wait(mutex);
```

```
    critical section
```

```
    signal(mutex);
```

```
    remainder section
```

```
until false;
```

```
end
```

```
process 2: begin
```

```
repeat
```

```
    wait(mutex);
```

```
    critical section
```

```
    signal(mutex);
```

```
    remainder section
```

```
until false;
```

```
end
```

```
parend
```

用信号量机制实现互斥的模式

wait(mutex)和
signal(mutex)必须
成对出现

```
semaphore mutex =1;
```

//表示打印机

```
begin
```

```
parbegin
```

```
p1: begin
```

```
repeat
```

```
....
```

```
wait(mutex);
```

使用打印机

```
signal(mutex)
```

```
....
```

```
until false;
```

```
end
```

```
p2: begin
```

```
repeat
```

```
....
```

```
wait(mutex);
```

使用打印机

```
signal(mutex);
```

```
....
```

```
until false;
```

```
end
```

```
parend
```

例：用记录型信号量实现两个进程互斥使用一台打印机

练习：用记录型信号量实现三个进程互斥使用一台打印机

```
p3: begin
```

```
repeat
```

```
....
```

```
wait(mutex);
```

使用打印机

```
signal(mutex);
```

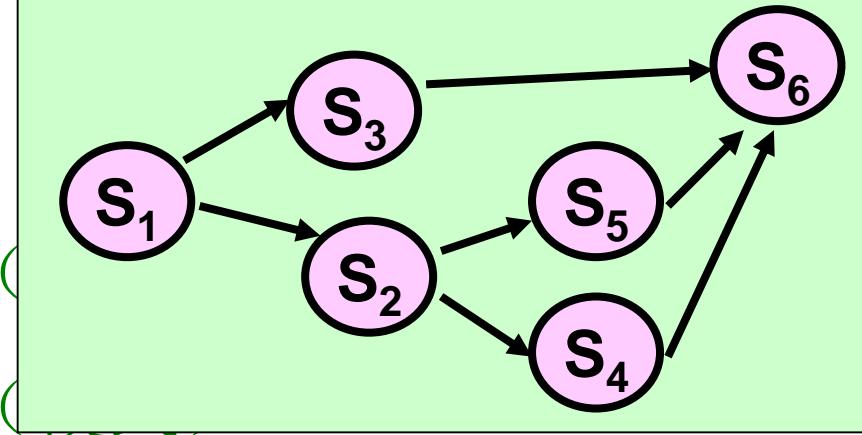
```
....
```

```
until false;
```

```
end
```

2.4 进程同步

- 利用信号量实现进程互斥
- 利用信号量实现前驱关系



```
semaphore a, b, c, d, e, f, g = 0, 0, 0, 0, 0, 0, 0;  
begin
```

```
parbegin
```

```
begin S1; signal(a); signal(b); end;  
begin wait(a); S2; signal(c); signal(d); end;  
begin wait(b); S3; signal(e); end;  
begin wait(c); S4; signal(f); end;  
begin wait(d); S5; signal(g); end;  
begin wait(e); wait(f); wait(g); S6; end;
```

```
parend
```

```
end
```

2.4 进程同步

1. 利用信号量实现进程互斥（模式）
2. 利用信号量实现前驱关系（模式）
3. 利用记录型信号量实现同步（模式）

p1, p2两进程因合作完成一项任务而共用一个变量x。

进程p2将处理结果送入x; 进程p1将x的结果打印。

即: p2: $x = \text{处理结果};$

p1: Print(x);

进程同步

如何实现该合作关系?

semaphore empty=1; //变量x可赋值使用，即P1
的print(x)已完成

semaphore full=0; //变量x已赋值，即P1可
print(x)

```
begin  
parbegin  
    p1: begin  
        repeat
```

请计算信号量
的取值范围。

```
        ... ...  
        wait(full);  
        print(x);  
        signal(empty);  
        ... ...
```

```
    until false;
```

```
end
```

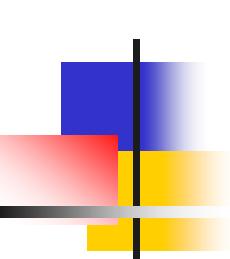
```
parend
```

p2: x=处理结果;
p1: Print(x);

用信号量机制实现同步的模式

```
    p2: begin  
        repeat  
            ... ...  
            wait(empty);  
            x:=处理结果;  
            signal(full);  
            ... ...  
        until false;  
    end
```

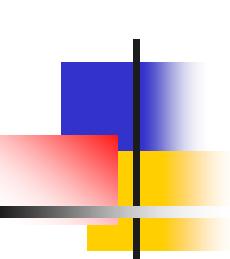
wait(mutex)和
signal(mutex)必须
成对出现



Chap2 进程管理

■ 内容

- § 2.1 前趋图和程序执行
- § 2.2 进程的描述
- § 2.3 进程控制
- § 2.4 进程同步
- § 2.5 经典进程的同步问题
- § 2.6 进程通信
- § 2.7 线程的基本概念
- § 2.8 线程的实现

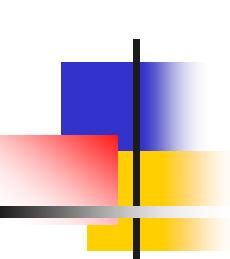


2.5 经典进程的同步问题

2.5.1 生产者——消费者问题

2.5.2 哲学家进餐问题

2.5.3 读者——写者问题



2.5 经典进程的同步问题

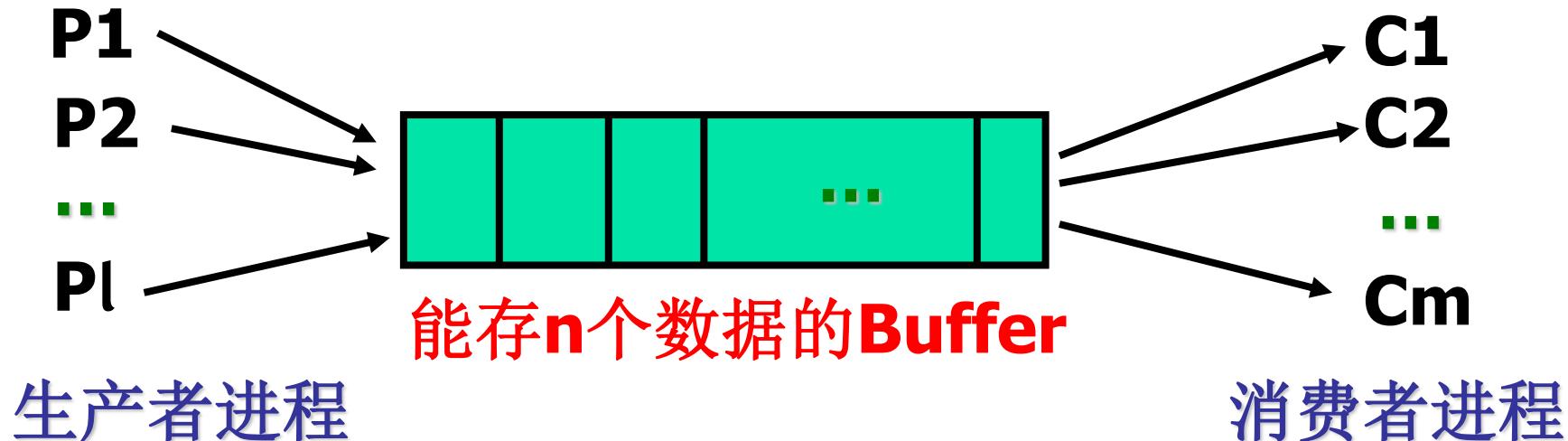
2.5.1 生产者——消费者问题

2.5.2 哲学家进餐问题

2.5.3 读者——写者问题

2.5.1 生产者——消费者问题

相互合作的进程关系的一种抽象。



生产者进程和消费者进程都以异步方式运行，
但它们之间必须保持同步。

1. 利用记录型信号量解决生产者——消费者问题

可利用互斥信号量mutex实现诸进程对缓冲池的互斥使用；利用信号量empty和full分别表示缓冲池中空缓冲池和满缓冲池的数量。假定这些生产者和消费者相互等效

```
Int i=0,out=0;  
Item buffer[n];  
Semaphore mutex=1,empty=n,full=0;  
Void producer(){  
    do{  
        生产一个产品放入nextp;
```

注意：

- 每个程序中用于实现互斥的**wait(mutex)**和**signal(mutex)**必须成对地出现。
- 对资源信号量**empty**和**full**的**wait**和**signal**操作，同样需要成对地出现，但处于不同的程序中。
- 在每个程序中的多个**wait**操作顺序不能颠倒。应先执行对资源信号量的**wait**操作，再执行对互斥信号量的**wait**操作，否则可能引起进程死锁。

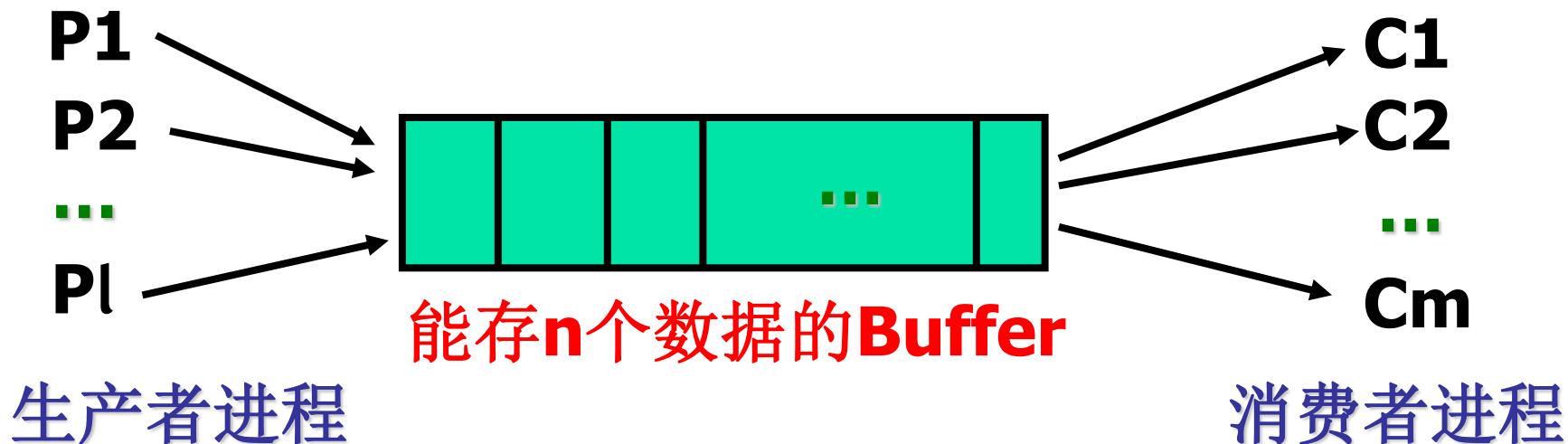
```
Void main(){  
    cobegin  
        proceducer();  
        consumer();  
    coend  
}
```

mutex: 生产者间，消费者间互斥使用缓冲区
empty: 缓冲区的空闲容量
full: 缓冲区的已占容量

消费 nextc 中的产品；
}while(TRUE)

2.5.1 生产者——消费者问题

- 利用记录型信号量解决生产者——消费者问题
- 利用AND信号量解决生产者——消费者问题



```

Int in=0, out= 0;
Item buffer[n];
semaphore mutex=1,empty=n,full=0;
Void producer(){
    do{
        生产一个产品放入nextp;
        ...
        Swait(empty,1,1; mutex,1,1);
        buffer[in]=nextp;
        in=(in+1) mod n;
        Ssignal(mutex,1; full,1);
    }while(TRUE)
}

Void main(){
    cobegin
        proceducer();
        consumer();
    coend
}

```

mutex: 生产者间，消费者
 间互斥使用缓冲区
empty: 缓冲区的空闲容量
full: 缓冲区的已占容量

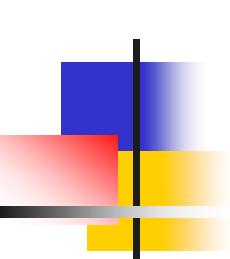
```

Void consumer(){
    do{
        Swait(full, mutex);
        nextc:=buffer[out];
        out=(out+1) mod n;
        Ssignal(mutex, empty);
        消费产品nextc;
    }while(TRUE)
}

```

用AND型信号量解决生产者—消费者问题

用信号量集解决生产者—消费者问题



2.5 经典进程的同步问题

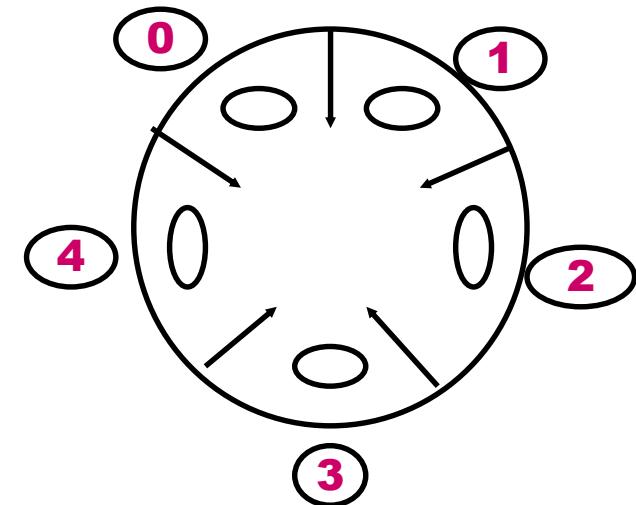
2.5.1 生产者——消费者问题

2.5.2 哲学家进餐问题

2.5.3 读者——写者问题

2.5.2 哲学家进餐问题

五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在桌子上有五只碗和五只筷子，他们的生活方式是交替地进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最近他的筷子，只有在他拿到两只筷子时才能进餐。进餐毕，放下筷子继续思考。



可见：相邻两位不能同时进餐；
最多只能有两人同时进餐。

2.5.2 哲学家进餐问题

1. 利用记录型信号量解决哲学家进餐问题

semaphore chopstick[5]={1,1,1,1,1};

第*i*位哲学家的活动可描述为:

表示chopstick[i]可用

do{

wait(chopstick[i]);

wait(chopstick[(i +1) mod 5]);

 ...

eat;

 ...

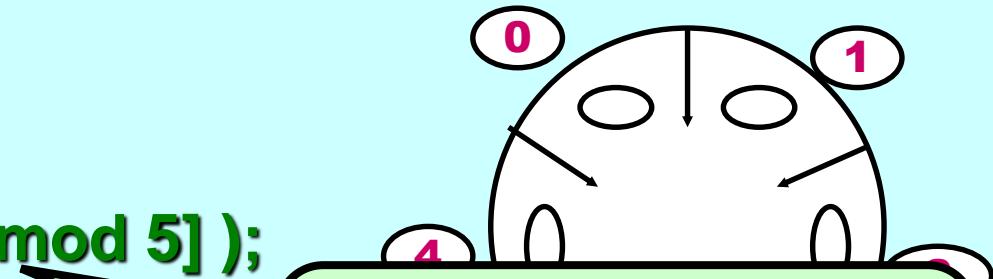
signal(chopstick[i]);

signal(chopstick[(i+1)mod 5]);

 ...

think;

}while(**TRUE**);



当哲学家饥饿时，总是先拿左边的筷子，再拿右边的筷子。

当哲学家进餐毕，先放下左边的筷子，再放下右边的筷子。

2.5.2 哲学家进餐问题

1. 利用记录型信号量解决哲学家进餐问题

semaphore chopstick[5]={1,1,1,1,1};

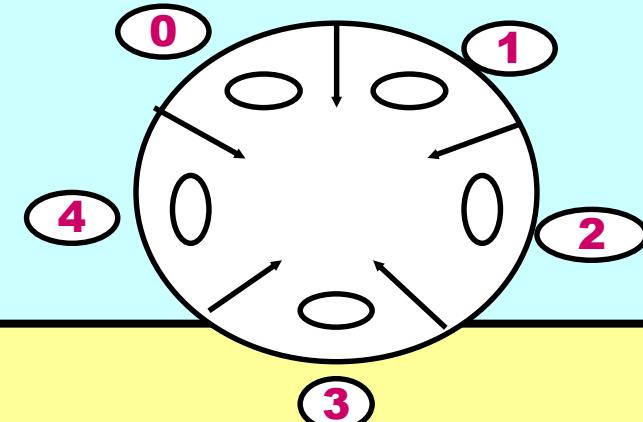
第*i*位哲学家的活动可描述为:

表示chopstick[i]可用

do{

wait(chopstick[i]);

wait(chopstick[(i +1) mod 5]);



解决方法:

- 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕后释放出他用过的两只筷子，从而使更多的哲学家能够进餐。
- 仅当哲学家的左右两只筷子均可用时，才允许他拿起筷子进餐。
- 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；偶数号哲学家则相反。

2.5.2 哲学家进餐问题

- 利用记录型信号量解决哲学家进餐问题
- 利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源（筷子）后方能进餐。本质上是

```
semaphore chopstick[5]={1,1,1,1,1};
```

```
Philosopher i
```

表示chopstick[i]可用

```
do{
```

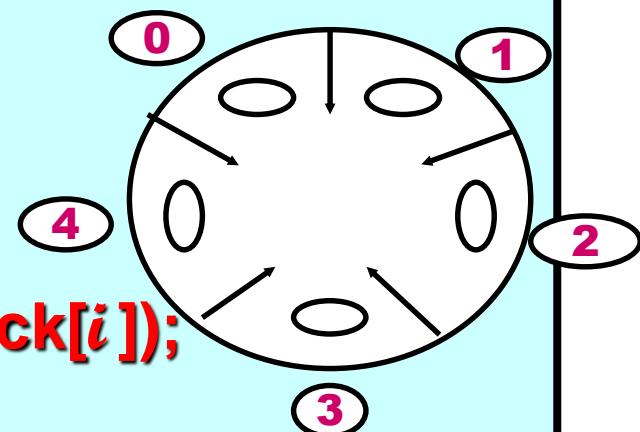
```
    think;
```

Swait(chopstick[(i+1)mod 5],chopstick[i]);

```
    eat;
```

Ssignal(chopstick[(i+1)mod 5], chopstick[i]);

```
}while(TRUE)
```

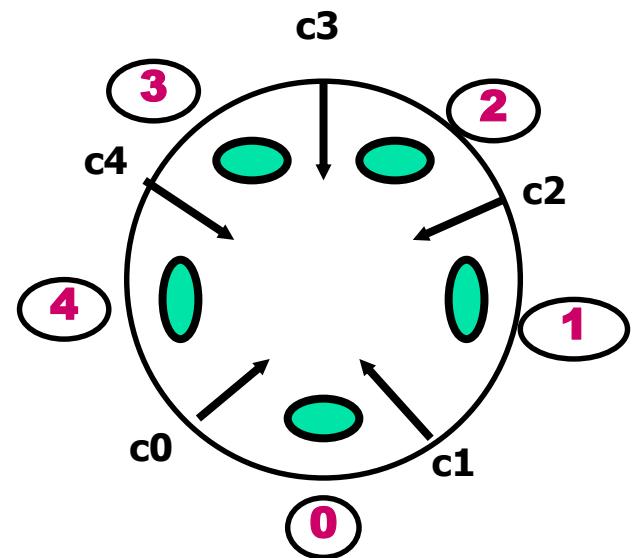


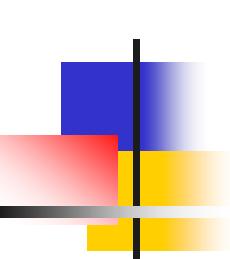
semaphore chopstick[5]={1,1,1,1,1};

第i位哲学家的活动可描述为：

do{

```
if i mod 2=1 {  
    wait(chopstick[ i ]);  
    wait(chopstick[ ( i +1) mod 5] )  
}  
else  
{  
    wait(chopstick[ ( i +1) mod 5] );  
    wait(chopstick[ i ])  
}  
eat;  
signal(chopstick[ i ]);  
signal(chopstick[(i +1)mod 5]);  
...  
think;  
}while(TRUE)
```





2.5 经典进程的同步问题

2.5.1 生产者——消费者问题

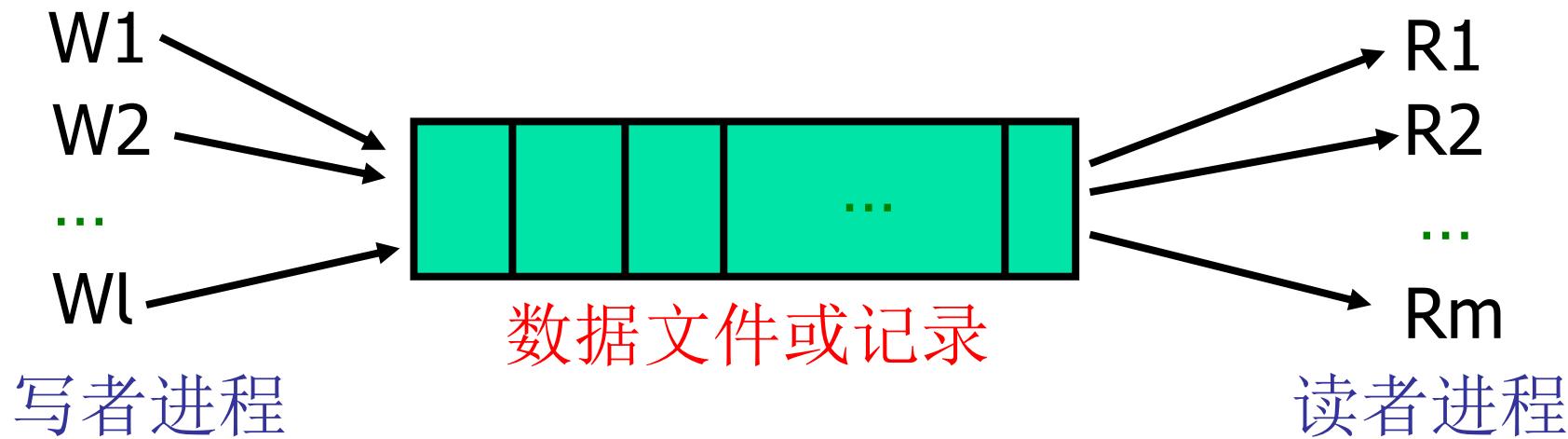
2.5.2 哲学家进餐问题

2.5.3 读者——写者问题

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。

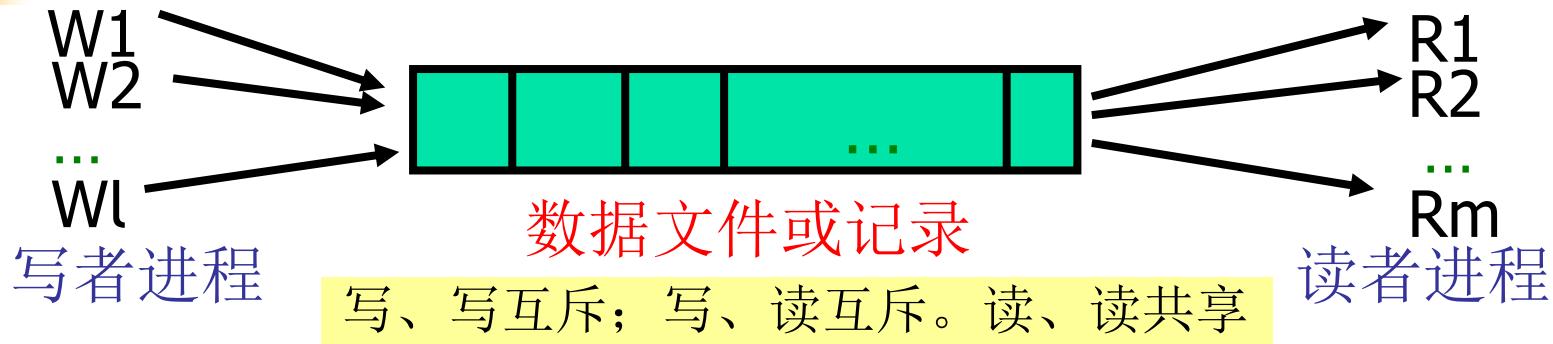
- 只要求读文件的进程称为“**Reader**进程”，其它进程则称为“**Writer**进程”。
- 允许多个进程同时读一个共享对象，但不允许一个**Writer**进程和其他**Reader**进程或**Writer**进程同时访问共享对象。“读者——写者问题”是保证一个**Writer**进程必须与其他进程互斥地访问共享对象的同步问题。



读、读共享。写、写互斥；写、读互斥

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



互斥信号量wmutex: 实现Reader与Writer进程间在读或写时的互斥
整型变量Readcount: 表示正在读的进程数目；

由于只要有一个Reader进程在读，便不允许Writer进程写。
∴仅当Readcount=0，即无Reader进程在读时，Reader才需要执行Wait(wmutex)操作。若Wait(wmutex)操作成功，Reader进程便可去读，相应地，做Readcount+1操作。

同理，仅当Reader进程在执行了Readcount减1操作后其值为0时，才需执行signal(wmutex)操作，以便让Write进程写。

互斥信号量rmutex: Reader进程间互斥访问Readcount

```
semaphore rmutex=1, wmutex =1;  
int readcount =0;  
Void Reader(){  
    do{  
        wait(rmutex);  
        if (Readcount==0)  
            wait(wmutex);  
        Readcount ++;  
        signal(rmutex);  
        ...  
        读;  
        ...  
        wait(rmutex);  
        Readcount --;  
        if (Readcount==0)  
            signal(wmutex);  
        signal(rmutex);  
    }while(TRUE);  
}
```

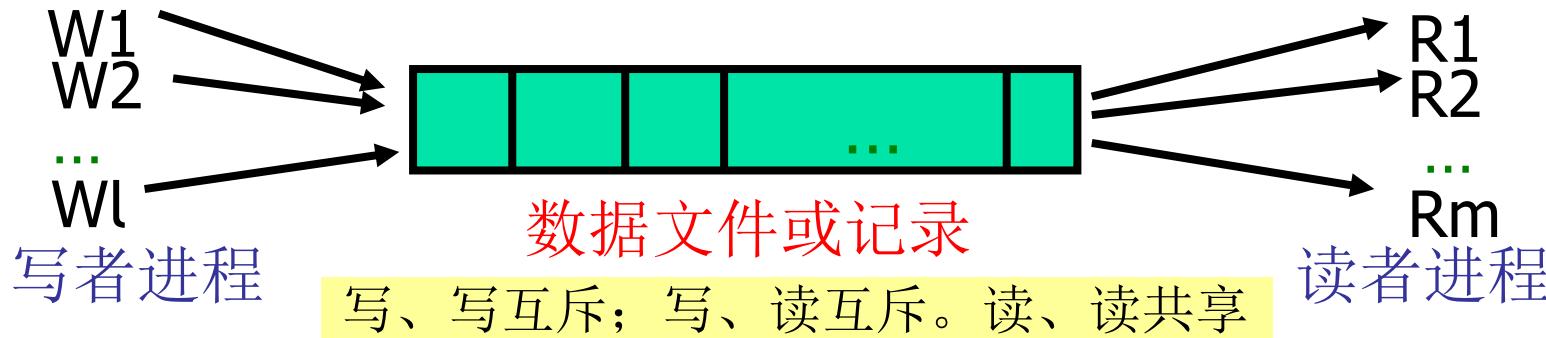
Wmutex: 读、写互斥；写、写互斥
Rmutex: 读间访问**Readcount**互斥
Readcount: 记录读者进程数

```
Void writer(){  
    do{  
        wait(wmutex);  
        写;  
        signal(wmutex);  
    }while(TRUE);  
}  
  
Void main(){  
    cobegin  
        reader(); writer();  
    Coend  
}
```

评价：能实现读者—写者问题
但读优先，对写者不公平

2.5.3 读者——写者问题

- 利用记录型信号量解决读者——写者问题
- 利用信号量集机制解决读者——写者问题



增加一个限制：最多只允许RN个读者同时读。

引入信号量L，并赋予其初值RN，通过执行Swait(L, 1, 1)操作，来控制读者的数目。

每当有一个读者进入时，就要先执行Swait(L, 1, 1)操作，使L的值减1。当有RN个读者进入读后，L便减为0，第RN + 1个读者要进入读时，必然会因Swait(L, 1, 1)操作失败而阻塞。

```
int RN;  
semaphore L=RN, mx= 1;  
Void reader() {  
    do{  
        Swait(L, 1, 1);  
        Swait(mx, 1, 0);  
        ...  
        读;  
        ...  
        Ssignal(L, 1);  
    }while(TRUE)  
}
```

```
Void main(){  
    cobegin  
        reader(); writer();  
    Coend  
}
```

L: 控制读进程的数目≤RN

Mx: 实现读、写互斥；写、写互斥

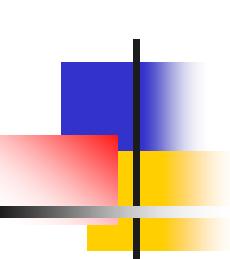
Void writer(){

do{

Swait(mx, 1, 1; L, RN, 0);
...
写;

Swait(mx, 1, 0)语句起
关的作用。只要无w
且无读进
入区，mx=1

Swait(mx, 1, 1; L, RN, 0)语句表示
仅当既无writer进程在写(mx=1),
又无reader进程在读(L=RN),
writer进程才能进入临界区写。



Chap2 进程管理

- 内容

- § 2.1 前趋图和程序执行
- § 2.2 进程的描述
- § 2.3 进程控制
- § 2.4 进程同步
- § 2.5 经典进程的同步问题
- § 2.6 进程通信
- § 2.7 线程的基本概念
- § 2.8 线程的实现

2.6 进程通信

进程通信：指进程之间的信息交换。

1. 低级通信：进程间仅交换一些状态和少量数据。

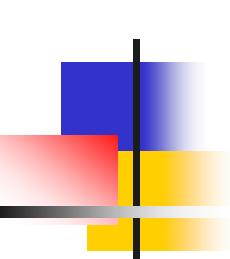
进程之间的互斥和同步——低级通信

信号量机制作为通信工具的缺点：

(1)效率低 (2)通信对用户不透明

2. 高级通信：进程间可交换大量数据。是指用户可直接利用操作系统所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。

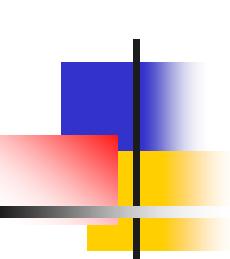


2.6 进程通信（高级通信）

2.6.1 进程通信的类型

2.6.2 消息传递通信的实现方法

2.6.3 直接消息传递的系统实例



2.6 进程通信（高级通信）

2.6.1 进程通信的类型

2.6.2 消息传递通信的实现方法

2.6.3 直接消息传递的系统实例

2.6.1 进程通信的类型

1. 共享存储器系统

相互通信的进程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。

- 基于共享数据结构的通信方式

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：程序员--公用数据结构的设置；对进程间同步的处理
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。

- 基于共享存储区的通信方式

在存储器中划出一块共享存储区，诸进程间实现通信

实现：进程在通信前，先向系统申请获得共享存储区中的一个分区，并指定该分区的关键字；→ 把获得的共享存储分区连接到需要进行通信的进程上；→ 此后，便可像读、写普通存储器一样地读、写该公用存储分区

2.6.1 进程通信的类型

1. 共享存储器系统 { 共享数据结构
 共享存储区

2. 消息传递系统

进程间的数据交换，以格式化的消息为单位。

程序员直接利用系统提供的一组通信命令(原语)进行通信。

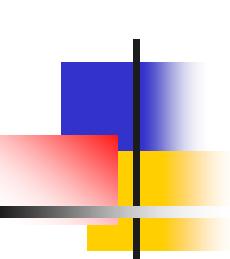
例：计算机网络。网络报文

3. 管道通信

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。



管道机制提供的协调能力：互斥；同步；确定对方是否存在



2.6 进程通信（高级通信）

2.6.1 进程通信的类型

2.6.2 消息传递通信的实现方法

2.6.3 直接消息传递的系统实例

2.6.2 消息传递通信的实现方法

进程间通信时，源进程可

1. 直接通信方式

发送进程利用OS提供的发
发送进程和接收进程都以

通信原语：

Send(Receiver, message)

Receive(Sender, message)

利用直接通信原语解决生

当生产者生产出一个
送给消费者进程；而消费
消息。若消息尚未产生出
将消息发送过来。

Producer:
repeat

...

produce an item in nextp;

...

send(consumer, nextp);
until false;

Consumer:

repeat

receive(producer, nextc);

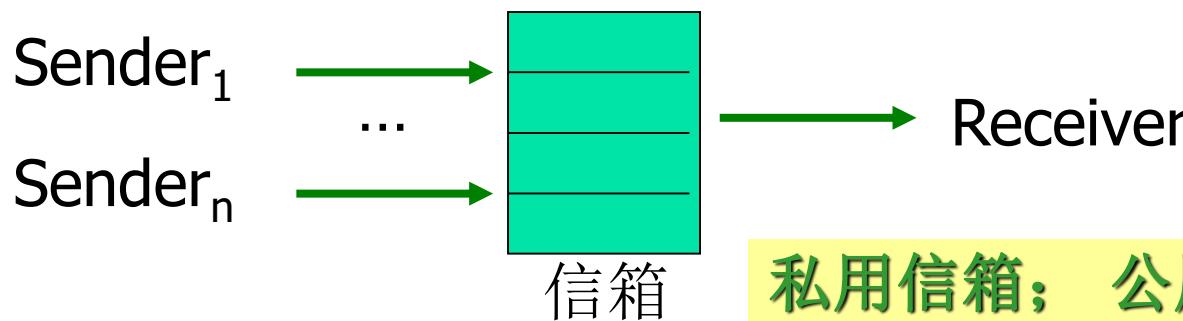
...

consume the item in nextc;
until false;

2.6.2 消息传递通信的实现方法

1. 直接通信方式
2. 间接通信方式 —— 通过信箱通信

消息在信箱中可安全保存，只允许核准的目标用户随时读取
利用信箱通信方式，既可实时通信，又可非实时通信。



私用信箱；公用信箱；共享信箱

系统为信箱通信提供若干原语：

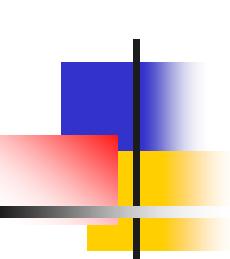
- 信箱的创建和撤销
- 消息的发送和接收

进程通信时,(发送进程, 接收进程)间关系:

一对一；多对一；一对多；多对多

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

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



2.6 进程通信（高级通信）

2.6.1 进程通信的类型

2.6.2 消息传递通信的实现方法

2.6.3 直接消息传递的系统实例

2.6.3 消息传递系统实现中的若干问题

1. 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

分类：

(1) 根据通信链路的建立方式：

- 显示连接：先用“建立连接”命令(原语)建立一条通信链路；通信；用显式方式拆除链路。——用于计算机网络
- 隐式连接：发送进程无须明确提出建立链路的要求，直接利用系统提供的发送命令(原语)，系统会自动地为之建立一条链路。——用于单机系统

(2) 根据通信链路的连接方法

{ 点一点连接通信链路
多点连接通信链路

(3) 根据通信方式的不同

{ 单向通信链路
双向链路

(4) 根据通信链路容量的不同

{ 无容量通信链路
有容量通信链路

2.6.3 消息传递系统实现中的若干问题

1. 通信链路

2. 消息的格式

- 单机系统环境：环境相同，消息格式简单
- 计算机网络环境：环境不同，消息的传输距离很远，
消息格式比较复杂

消息格式：
 { 消息头：消息在传输时所需的控制信息
 消息的正文：发送进程实际上所发送的数据

从消息长度分类：
 { 比较短的定长消息格式
 变长的消息格式

2.6.3 消息传递系统实现中的若干问题

1. 通信链路
2. 消息的格式
3. 进程同步方式

在进程之间进行通信时，辅以进程同步机制，使诸进程间能协调通信。

发送进程或接收进程在完成消息的发送或接收后，都存在两种可能性：进程或者继续发送(接收)；阻塞。

- 发送进程阻塞、接收进程阻塞
- 发送进程不阻塞、接收进程阻塞
- 发送进程和接收进程均不阻塞

2.6.3 消息缓冲队列通信机制

发送进程利用Send原语，将消息直接发送给接收进程；
接收进程利用Receive原语接收消息。

1. 消息缓冲队列通信机制中的数据结构

(1) 消息缓冲区

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

2.6.3 消息缓冲队列通信机制

1. 消息缓冲队列通信机制中的数据结构

(1)消息缓冲区

(2)PCB中有关通信的数据项

增加用于对消息队列进行操作和实现同步的信号量，并将它们置入进程的PCB中。

```
type processcontrol block = record
```

...

mq;

消息队列队首指针

mutex;

消息队列互斥信号量

sm;

消息队列资源信号量

...

```
end
```

2.6.3 消息缓冲队列通信机制

图示

2. 发送原语

```
procedure send(receiver, a) //将发送区a的内容发给receiver
begin
    getbuf(a.size, i);           //根据a.size申请缓冲区i
    i.sender := a.sender;        //将发送区a中的信息复制
    i.size := a.size;            //到消息缓冲区i中
    i.text := a.text;
    i.next := 0;

    getid(PCB set, receiver.j); //取接收进程内部标识符放于j
    wait(j.mutex);             //将消息缓冲区插入消息队列
    insert(j.mq, i);
    signal(j.mutex);
    signal(j.sm);

end
```

2.6.3 消息缓冲队列通信机制

3. 接收原语

图示

```
procedure receive(b) //从进程自己的消息接收队列中取  
begin //消息i放入消息接收区b中
```

j:=internal name; //j为接收进程内部标识符

```
wait(j.sm); //将消息队列中的第一个消息移出  
wait(j.mutex);  
remove(j.mq, i);  
signal(j.mutex);
```

```
b.sender :=i.sender; //将消息缓冲区i中的信息  
b.size :=i.size; //复制到接收区b  
b.text :=i.text;  
end
```

返回1

返回2

进程A

PCB(B)

进程B

send(B, a)

receive(b)

sender:A

size:5

text:Hello

mq

mutex

sm

第一消息缓冲区

sender:A

size:5

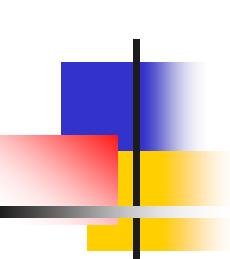
text:Hello

next:0

发送区
a

接收区
b

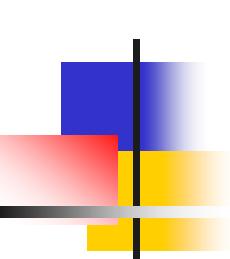




Chap2 进程管理

- 内容

- § 2.1 前趋图和程序执行
- § 2.2 进程的描述
- § 2.3 进程控制
- § 2.4 进程同步
- § 2.5 经典进程的同步问题
- § 2.6 进程通信
- § 2.7 线程的基本概念
- § 2.8 线程的实现

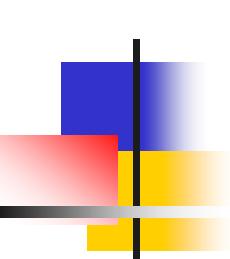


2.7 线程的基本概念

2.7.1 线程的引入

2.7.2 线程与进程的比较

2.7.3 线程的状态和线程控制块



2.7 线程的基本概念

2.7.1 线程的引入

2.7.2 线程与进程的比较

2.7.3 线程的状态和线程控制块

2.7 线程的基本概念

1. 线程的引入

- 引入进程的目的：

使多个程序能并发执行，提高资源利用率和系统吞吐量。

进程的基本属性：

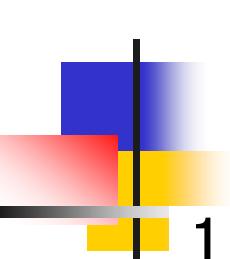
- 进程是一个可拥有资源的独立单位
- 进程同时又是一个可独立调度和分派的基本单位
- 系统进行的其它操作：创建进程；撤销进程；进程切换

将进程的两个基本属性分开：作为调度和分派的基本单位，不同时作为拥有资源的单位，以“轻装上阵”；

对于拥有资源的基本单位，又不对之进行频繁切换。

线程是进程的一条执行路径，它包含独立的堆栈和CPU寄存器状态，每个线程共享其所附属的进程的所有的资源，包括打开的文件、页表（因此也就共享整个用户态地址空间）、信号标识及动态分配的内存等等。

线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一物理内存空间，当进程退出时该进程所产生的线程都会被强制退出并清除。



2.7 线程的基本概念

1. 线程的引入

2. 线程的属性

- 轻型实体（为进程的一个实体）

只有一点必不可少的、能保证独立运行的资源

- 独立调度和分派的基本单位
- 可并发执行
- 共享进程资源

在多线程OS中，通常一个进程包括多个线程，每个线程是利用CPU的基本单位，是花费最小开销的实体。

■ 引入线程的目的：

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。

2.7 线程的基本概念

1. 线程的引入

2. 线程的属性

3. 线程的状态

□ **状态参数：** OS中可用线程标识符和一组状态参数进行描述

- 寄存器状态

- 堆栈

- 线程运行状态

- 优先级

- 线程专有寄存器

- 信号屏蔽

- 线程运行状态 {
 - 执行状态
 - 就绪状态
 - 阻塞状态

2.7 线程的基本概念

- 1. 线程的引入
- 2. 线程的属性
- 3. 线程的状态
- 4. 线程的创建和终止

在多线程OS中，应用程序启动时，通常只有一个线程（初始化线程）在执行，它根据需要可创建若干线程。

创建新线程时，需要利用线程创建函数（或系统调用），并提供相应参数。线程创建函数执行完后，返回一个线程标识符供以后使用。

线程的终止方式 {
 自愿退出
 被其他线程强行终止

线程被终止后并不立即释放资源，只有当进程中的其它线程执行分离函数后，资源才分离出来能被其他线程利用。

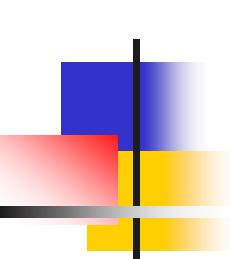
2.7 线程的基本概念

1. 线程的引入
2. 线程的属性
3. 线程的状态
4. 线程的创建和终止
5. 多线程OS中的进程

在多线程OS中，进程是作为拥有系统资源的基本单位，通常进程都包含多个线程并为它们提供资源，但进程不再作为一个执行的实体。

此时进程的属性：

- 作为系统资源分配的单位
- 可包括多个线程
- 进程不是一个执行的实体



2.7 线程的基本概念

线程间的同步和通信

1. 互斥锁

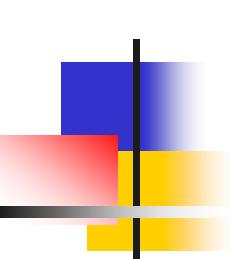
Lock(); Unlock()

2. 条件变量

3. 信号量机制

私有信号量：实现同一进程内的各线程间的同步

公用信号量：实现不同进程间或不同进程的线程间的同步



Chap2 进程管理

- 内容
 - § 2.1 前趋图和程序执行
 - § 2.2 进程的描述
 - § 2.3 进程控制
 - § 2.4 进程同步
 - § 2.5 经典进程的同步问题
 - § 2.6 进程通信
 - § 2.7 线程的基本概念
 - § 2.8 线程的实现

2.8 线程的实现

1. 内核支持线程(Kernel-Supported Threads)

依赖于OS核心，由内核的内部需求进行创建和撤销，用来执行一个指定的函数。Windows NT和OS/2支持内核线程；

- 内核维护进程和线程；
- 线程调度、切换由内核完成；
- 一个线程发起系统调用而阻塞，不会影响其他线程的运行；
- 时间片分配给线程，所以多线程的进程获得更多CPU时间。

2.8 线程的实现

1. 内核支持线程(Kernel-Supported Threads)
2. 用户级线程 (User-level Threads)

不依赖于OS核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。

如：数据库系统informix，图形处理PageMaker。

- 线程的维护由应用进程完成，内核不了解用户级线程的存在
- 调度、切换由应用软件内部进行，通常采用简单的规则，也无需用户态/核心态切换，所以速度特别快
- 一个线程发起系统调用而阻塞，则整个进程在等待
- 时间片分配给进程，线程多则每个线程就慢

本章小结

1. 理解程序的顺序执行及其特征，理解前趋图的概念，理解程序的并发执行及其特征，掌握进程的特征与状态，掌握进程控制块的主要内容。
2. 理解进程的创建、进程的终止、进程阻塞与唤醒，了解进程的挂起与激活。
3. 理解进程同步的基本概念，掌握信号量机制及信号量的应用。
4. 掌握生产者—消费者问题、哲学家进餐问题、读者—写者问题的解决方法。
5. 理解进程通信的类型和消息传递通信的实现方法，了解消息传递系统实现中的若干问题，了解消息缓冲队列通信机制。
6. 了解线程的基本概念