

# 第二章 进程管理

任立勇

2017年3月

# 目录

- 2.1 前驱图和程序执行
- 2.2 进程的基本概念
- 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

## 2.1 前驱图和程序执行

### 2.1.1 程序的顺序执行及其特征

#### 1. 程序的顺序执行

仅当前一操作(程序段)执行完后，才能执行后继操作。例如，在进行计算时，总须先输入用户的程序和数据，然后进行计算，最后才能打印计算结果。

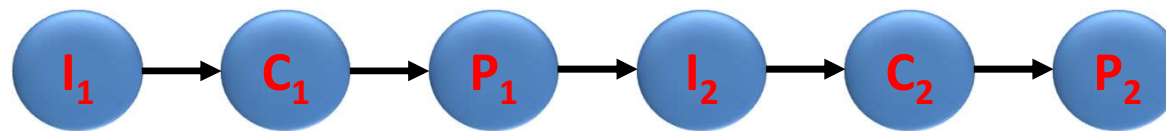


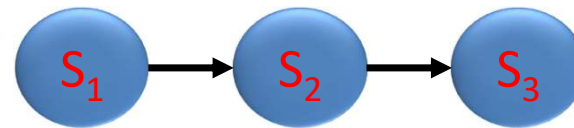
图 2-1 程序的顺序执行

例2:

$S_1: a := x+y;$

$S_2: b := a-5;$

$S_3: c := b+1;$



(b) 三条语句顺序执行



## 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**：程序运行时独占全机资源，程序一旦开始执行，其执行结果不受外界因素影响。
- (3) **可再现性**：只要程序执行时的环境和初始条件相同，都将获得相同的结果。  
(不论它是从头到尾不停顿地执行，还是“停停走走”地执行)



## 2.1.2 前趋图

**前趋图**(Precedence Graph)是一个有向无循环图, 记为DAG(Directed Acyclic Graph), 用于描述进程之间执行的前后关系。

- ◆ 图中的每个结点可用于描述一个程序段或进程, 乃至一条语句;
- ◆ 结点间的有向边则用于表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)“ $\rightarrow$ ”。

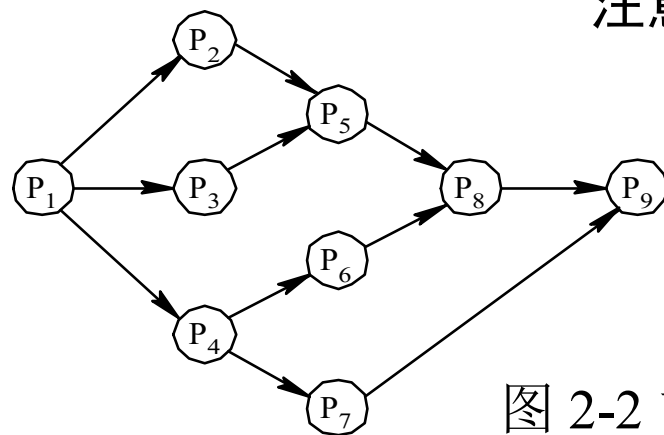
$\rightarrow = \{(P_i, P_j) | P_i \text{ must complete before } P_j \text{ may start}\}$ , 如果  $(P_i, P_j) \in \rightarrow$ , 可写成  $P_i \rightarrow P_j$ , 称  $P_i$  是  $P_j$  的**直接前趋**, 而称  $P_j$  是  $P_i$  的**直接后继**。

把没有前趋的结点称为**初始结点**(Initial Node)

把没有后继的结点称为**终止结点**(Final Node)。

**重量(Weight)**表示该结点所含有的程序量或结点的执行时间。

注意，**前趋图中必须不存在循环**



(a) 具有九个结点的前趋图



(b) 具有循环的前趋图



## 2.1.3 程序的并发执行及其特征

### 1. 程序的并发执行

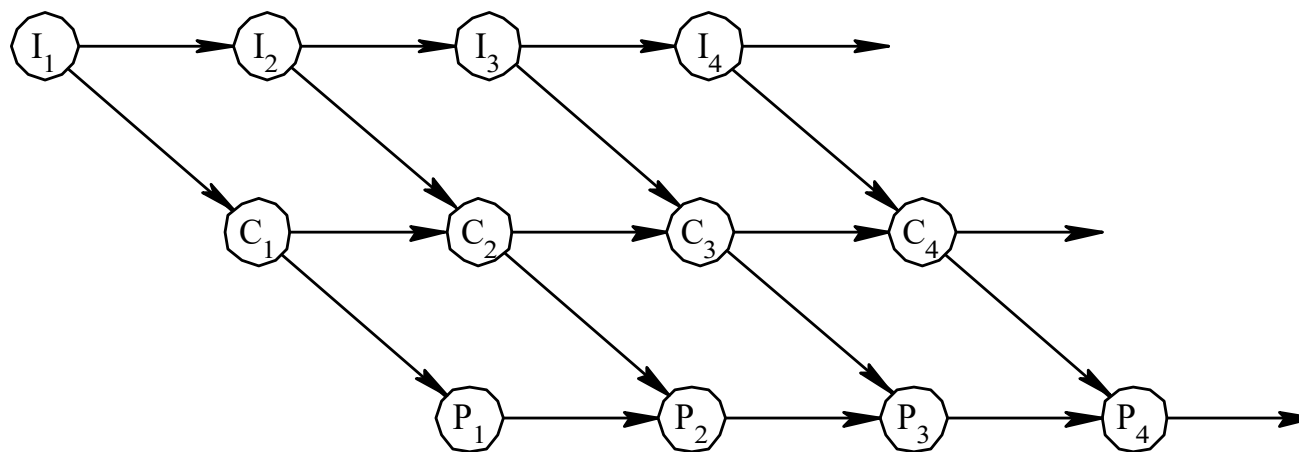


图 2-3 并发执行时的前趋图



对于具有下述四条语句的程序段：

$S_1: a := x + 2$

$S_2: b := y + 4$

$S_3: c := a + b$

$S_4: d := c + b$

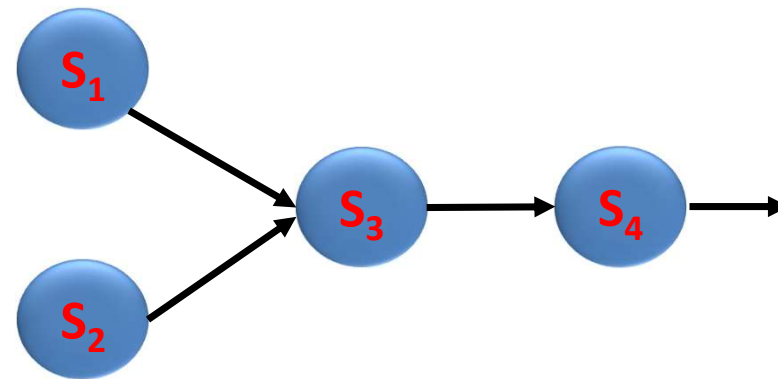


图 2-4 四条语句的前趋关系

## 2. 程序并发执行时的特征

- 1) **间断性**: 由于它们共享系统资源, 以及为完成同一项任务而相互合作, 致使在这些并发执行的程序之间, 形成了相互制约的关系。相互制约将导致并发程序具有“**执行—暂停—执行**”这种间断性的活动规律。
- 2) **失去封闭性**: 是多个程序共享系统中的各种资源, 因而这些资源的状态将由多个程序来改变, 致使程序的运行已失去了封闭性。
- 3) **不可再现性**: 程序在并发执行时, 由于失去了封闭性, 导致不可再现性。

## 举例：（P35）

- 有两个循环程序A和B它们共享一个变量N。
  - 程序A每执行一次时，都要做 $N = N + 1$ 操作；
  - 程序B每执行一次时，都要执行Print (N) 操作，然后再将N置成“0”。
- 程序A和B以不同的速度运行。这样，可能出现其计算结果不可再现性
  - 亦即，程序经过多次执行后，虽然它们执行时的环境和初始条件相同，但得到的结果却各不相同。

# 程序A和B以不同的速度运行出现的情况：

1、 $N=N+1$ ，在Print (N) 和 $N=0$ 之前执行，

即执行次序：	$N=N+1$	$n+1$
	Print (N)	$n+1$
	$N=0$	0

2、 $N=N+1$ ，在Print和 $N=0$ 之后执行，

即执行次序：	Print (N)	$n$
	$N=0$	0
	$N=N+1$	1

3、 $N=N+1$ ，在Print和 $N=0$ 之间执行，

即执行次序：	Print (N)	$n$
	$N=N+1$	$n+1$
	$N=0$	0


彩色的为执行结果，各不相同

## 2.2 进程的描述

- 1. 进程的特征和定义

- 1) 进程的定义

- 为使程序（含数据）能独立运行，应为之配置一进程控制块，即PCB；
    - 而由程序段、相关的数据段和PCB三部分便构成了进程实体。
    - 所谓创建进程，实质上是创建进程实体中的PCB；
    - 而撤消进程，实质上是撤消进程的PCB。



较典型的进程定义有：

(1)进程是程序的一次执行。

(2)进程是一个程序及其数据在处理机上顺序执行时所发生的活动。

(3)进程是程序在一个数据集合上运行的过程，它是系统进行**资源分配**和**调度**的一个独立单位。

在引入了进程实体的概念后，我们可以把传统OS中的**进程定义**为：“**进程是程序实体的运行过程，是系统进行资源分配和调度的一个独立单位**”。



## 1) 动态性

- 进程的实质是程序实体的一次执行过程，因此，动态性是进程的最基本的特征。
- 动态性表现：“它由创建而产生，由调度而执行，由撤消而消亡”。
  - 进程实体有一定的生命期。
- 对比：程序是一组有序指令的集合，其本身并不具有运动的含义，因而是静态的。
  - 程序为剧本，而进程是剧本的一次演绎。





## 2) 并发性


这是指多个进程实体同存于内存中，且能在一段时间内同时运行。

## 3) 独立性

指进程实体是一个能独立运行、独立分配资源和独立接受调度的基本单位；

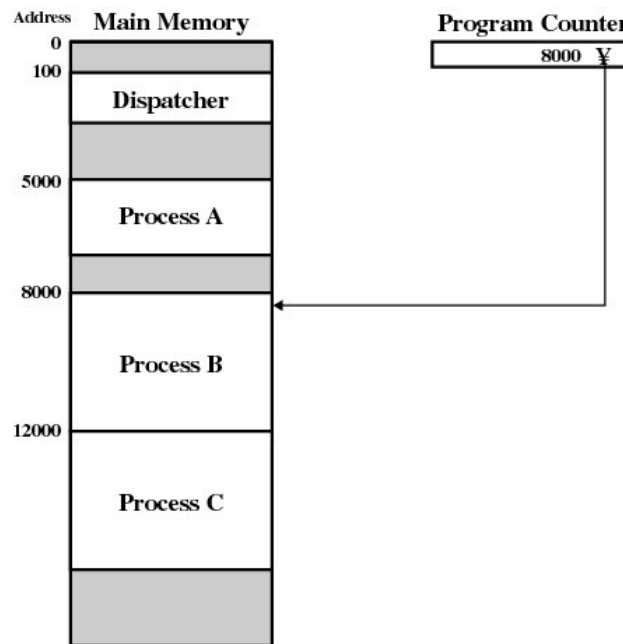
## 4) 异步性

指进程按各自独立的、不可预知的速度向前推进，或说进程实体按异步方式运行。



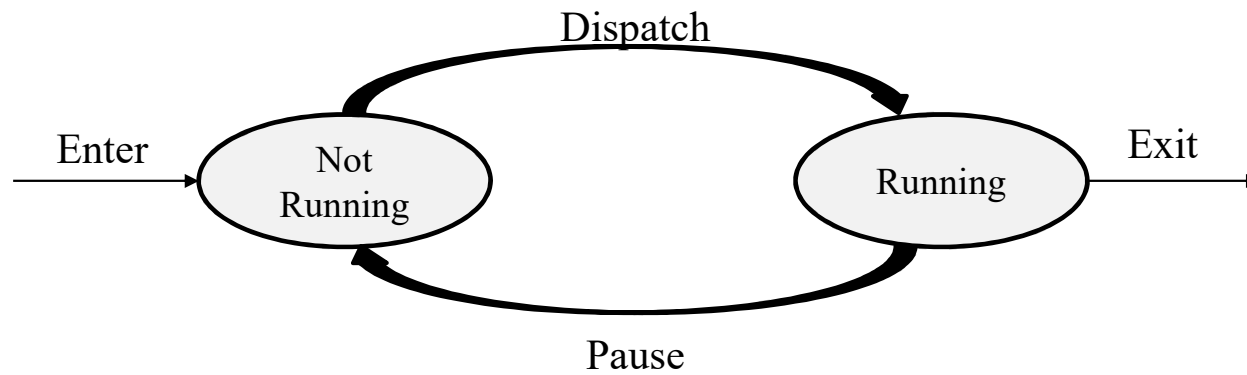
## 2.2.2 进程的状态转换

- 进程执行要由调度程序调度才能占用CPU执行。

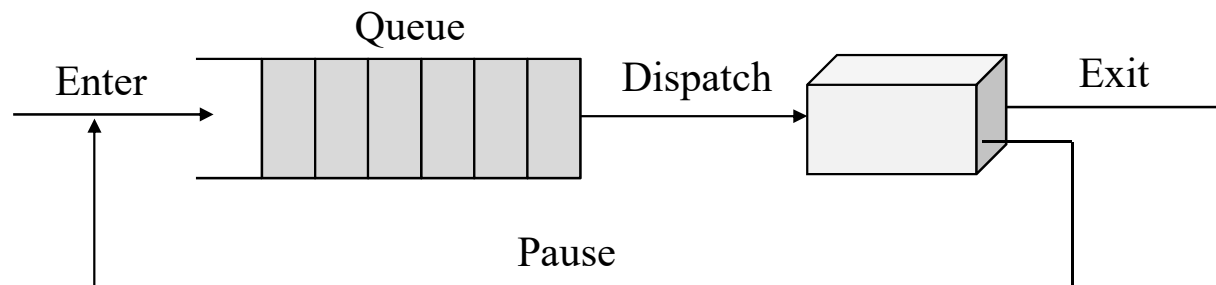


三个进程同时驻留内存

# 两状态进程模型



(a) State transition diagram



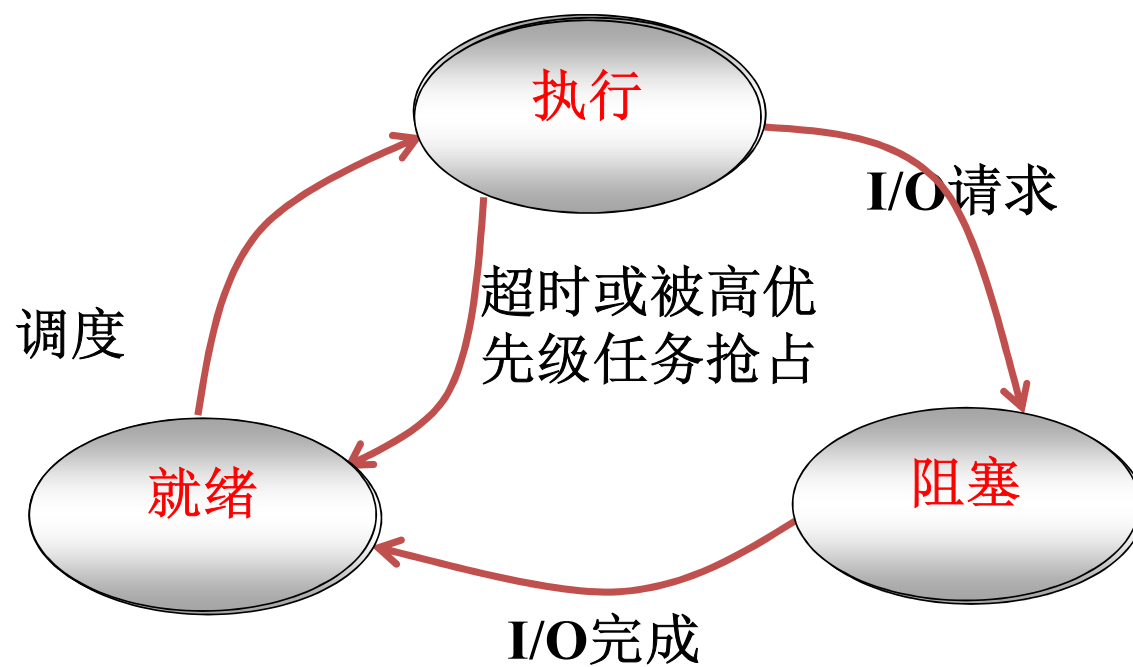
(b) Queuing Diagram

# 1. 进程的三种基本状态

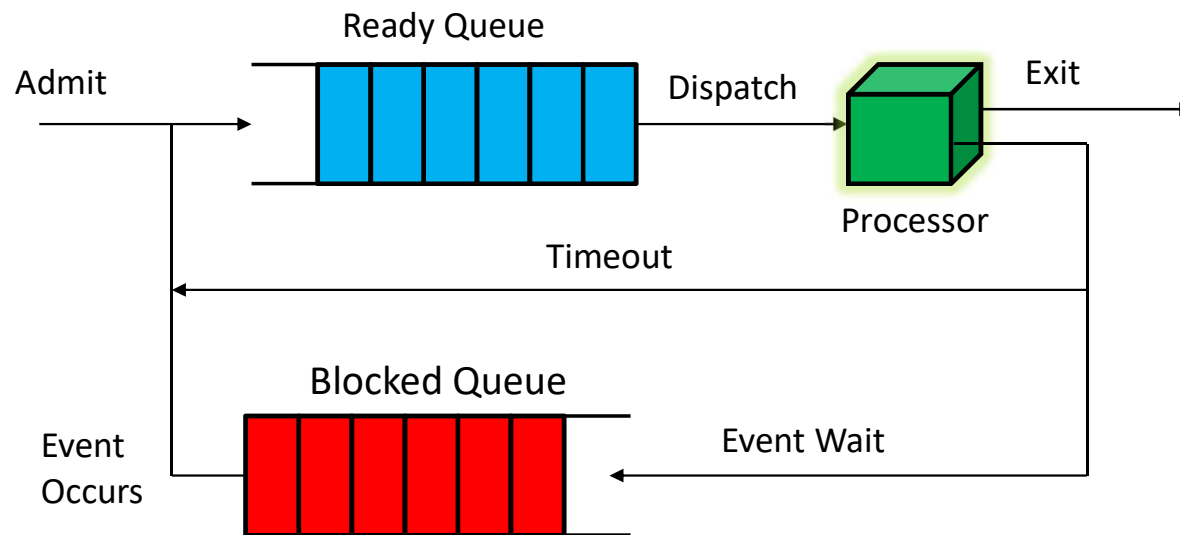
- 进程的三种基本状态：

- 1) **就绪（Ready）状态**：当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行。
- 2) **执行状态**：进程已获得CPU，其程序正在执行。
- 3) **阻塞状态**：正在执行的进程由于发生某事件而暂时无法继续执行时，便放弃处理机而处于暂停状态，把这种暂停状态称为阻塞状态，有时也称为等待状态。

# 进程的三种基本状态及其转换

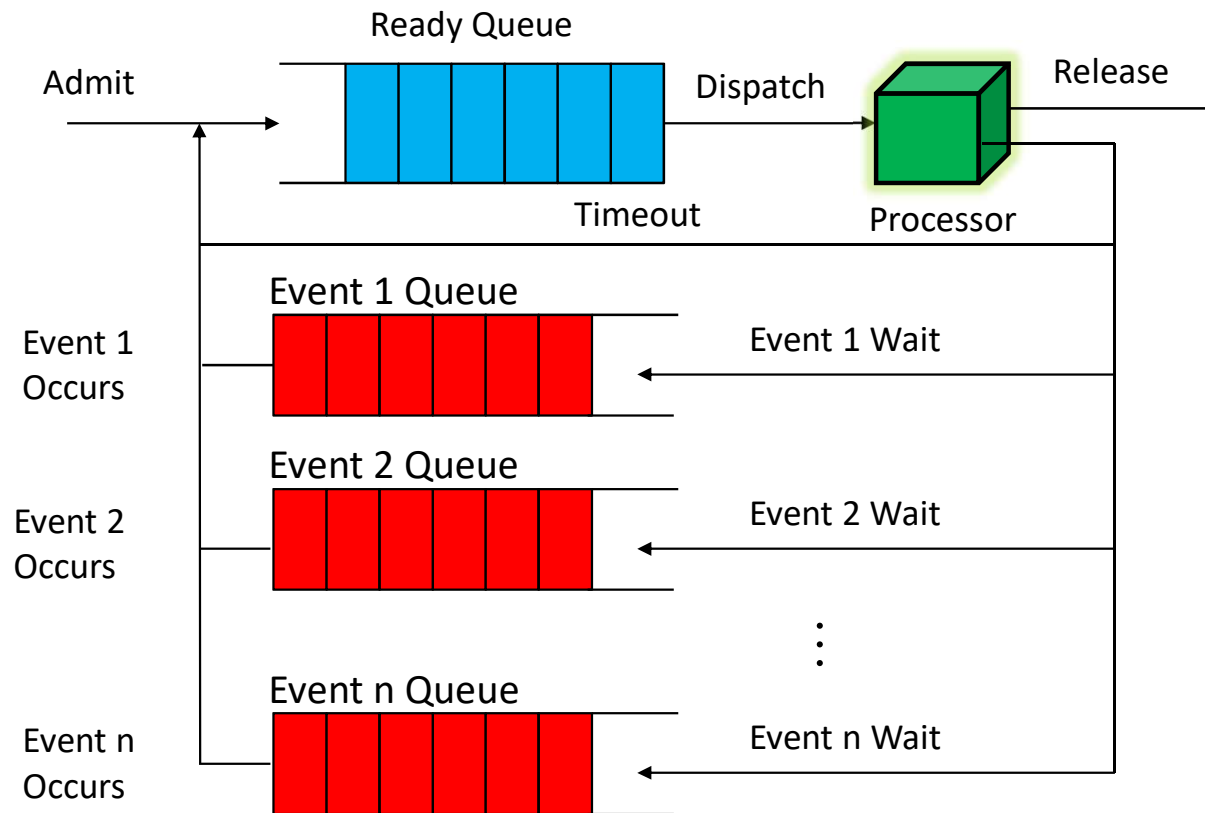


# 单阻塞队列



(a) Single Blocked queue

# 多阻塞队列



(b) Multiple Blocked queue





## • 三种基本状态的思考

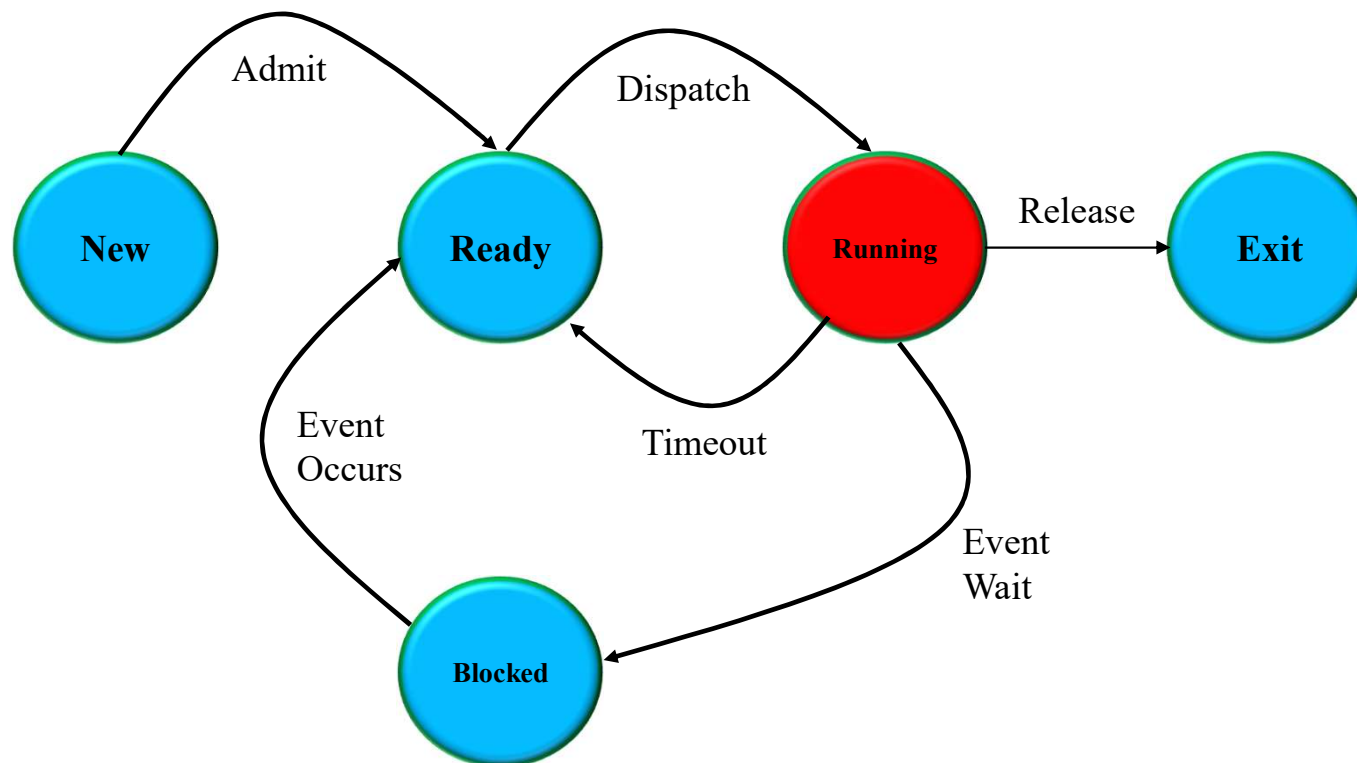
- 进程主动完成状态转换，还是被动完成？
- 进程状态是否唯一？
- 时间片用完是否是进程由执行变为就绪的唯一原因？
- 在单处理机系统中，是否可以有多个进程处于执行状态？
- 在多处理机系统中，是否可以有多个进程处于执行状态？



### 3. 创建状态和终止状态

- **执行状态：** 占用处理机（单处理机环境中，某一时刻仅一个进程占用处理机）。
- **就绪状态：** 准备执行。
- **阻塞状态：** 等待某事件发生才能执行，如等待I/O完成等。
- **创建状态：** 进程已经创建，但未被OS接纳为可执行进程，并且进程资源尚未分配，程序还在辅存，PCB在内存。
- **终止状态：** 因停止或取消，被OS从执行状态释放，将释放空间并保留PCB。

# 进程五状态转换图



## 2. 挂起状态

### 挂起状态

使执行的进程暂停执行, 静止下来, 使就绪状态的进程暂不接受调度, 我们把这种静止状态称为挂起状态。

#### 1) 引入挂起状态的原因

- (1) 终端用户的请求。
- (2) 父进程请求。
- (3) 负荷调节的需要。当实时系统中的工作负荷较重, 把一些不重要的进程挂起, 以保证系统能正常运行。
- (4) 操作系统的需要。操作系统有时希望挂起某些进程, 以便检查运行中的资源使用情况或进行记账。

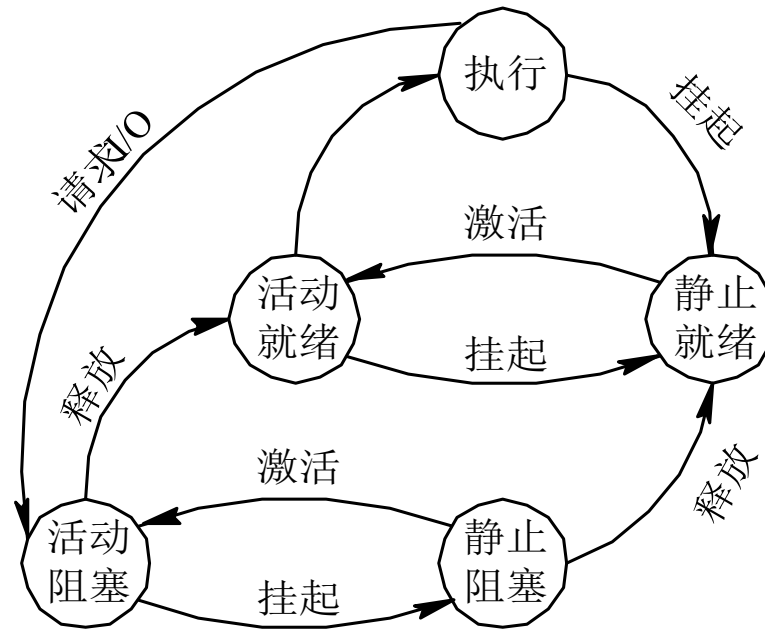
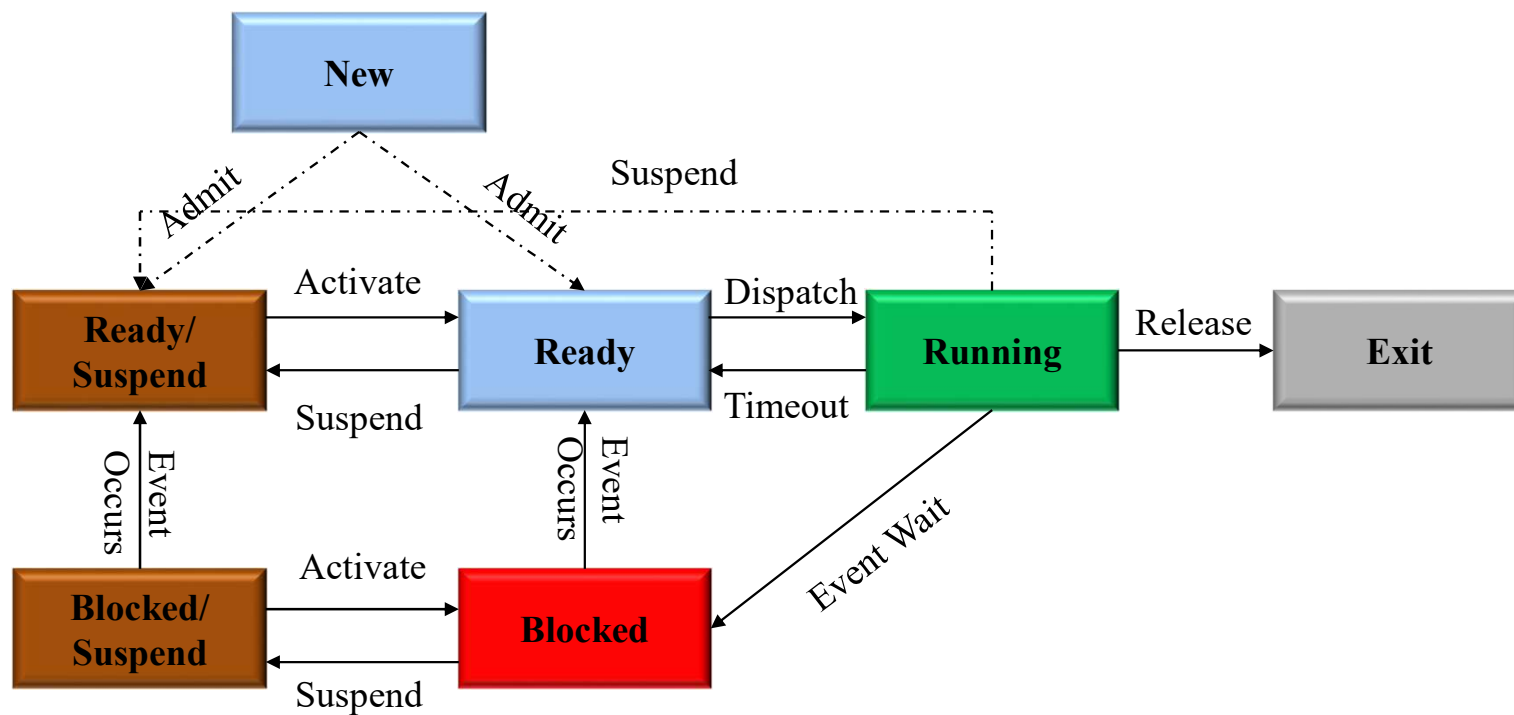
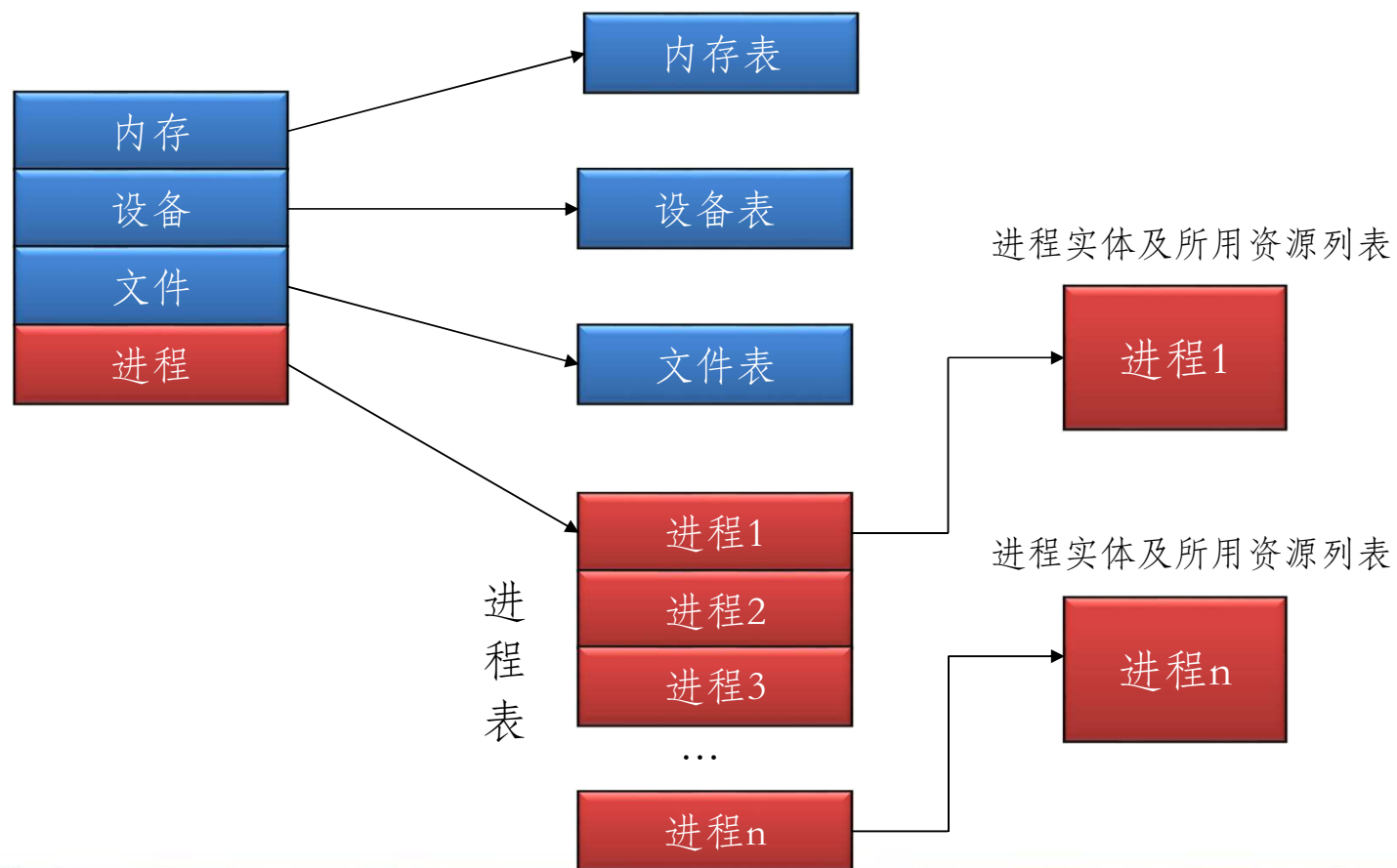


图 2-6 具有挂起状态的进程状态图

# 有挂起状态的进程转换



## 2.2.4 进程管理中的数据结构





## 2.2.4 进程管理中的数据结构

### 2. 进程控制块的作用

- ❑ 进程控制块的作用是使一个在多道程序环境下不能独立运行的程序(含数据)，成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程。
- ❑ PCB作为进程实体的一部分，记录了操作系统所需的，用于描述进程的当前情况及管理进程运行的全部信息。
- ❑ OS是根据PCB来对并发执行的进程进行控制和管理。PCB常驻内存
- ❑ PCB是进程存在的惟一标志。
  - 进程创建：分配进程控制块
  - 进程调度：保存和读取进程控制块
  - 进程撤销：回收进程控制块

## 2.2.4 进程管理中的数据结构

### PCB的具体作用：

- 作为进程独立运行基本单位的标志
- 能实现间断性运行方式
- 提供进程管理所需要的信息
- 提供进程调度所需要的信息
- 实现与其他进程的同步与通信

### 3. 进程控制块中的信息

- 进程控制块的内容
  - 进程标识符信息
  - 处理器状态信息
  - 进程调度信息
  - 进程控制信息

## 2. 进程控制块中的信息

### 1) 进程标识符

进程标识符用于唯一地标识一个进程。一个进程通常有两种标识符：

(1) **内部标识符**。在所有的操作系统中，都为每一个进程赋予一个唯一的数字标识符，它通常是一个进程的序号。设置内部标识符主要是为了方便系统使用。

(2) **外部标识符**。它由创建者提供，通常是由字母、数字组成，往往是由用户(进程)在访问该进程时使用。为了描述进程的家族关系，还应设置父进程标识及子进程标识。此外，还可设置用户标识，以指示拥有该进程的用户。

## 2) 处理机状态

- 处理机状态信息主要是由处理机的各种寄存器中的内容组成的。
  - **通用寄存器**，又称为用户可视寄存器, 暂存信息。
  - **指令计数器**，其中存放了要访问的下一条指令的地址。
  - **程序状态字PSW**，其中含有状态信息，如条件码、执行方式、中断屏蔽标志等。
  - **用户栈指针**，用于存放系统调用参数及调用地址。栈指针指向该栈的栈顶。

这些都是中断和进程切换时需要保护的内容！



### 3) 进程调度信息

□ 在PCB中还存放一些与进程调度有关的信息，包括：

- ① **进程状态**，指明进程的当前状态，作为进程调度和对换时的依据；
- ② **进程优先级**，用于描述进程使用处理机的优先级别的一个整数，优先级高的进程应优先获得处理机；
- ③ **进程调度所需的其它信息**，如，进程已等待CPU的时间总和、进程已执行的时间总和等；
- ④ **事件**，是指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。



## 4) 进程控制信息

### □ 进程控制信息包括：

- ① **程序和数据地址**，是指进程的程序和数据所在的内存或外存地(首)址；
- ② **进程同步和通信机制**，指实现进程同步和进程通信时必需的机制，如消息队列指针、信号量等；
- ③ **资源清单**，列出了除CPU以外的、进程所需的全部资源及已经分配到该进程的资源的清单；
- ④ **链接指针**，它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。



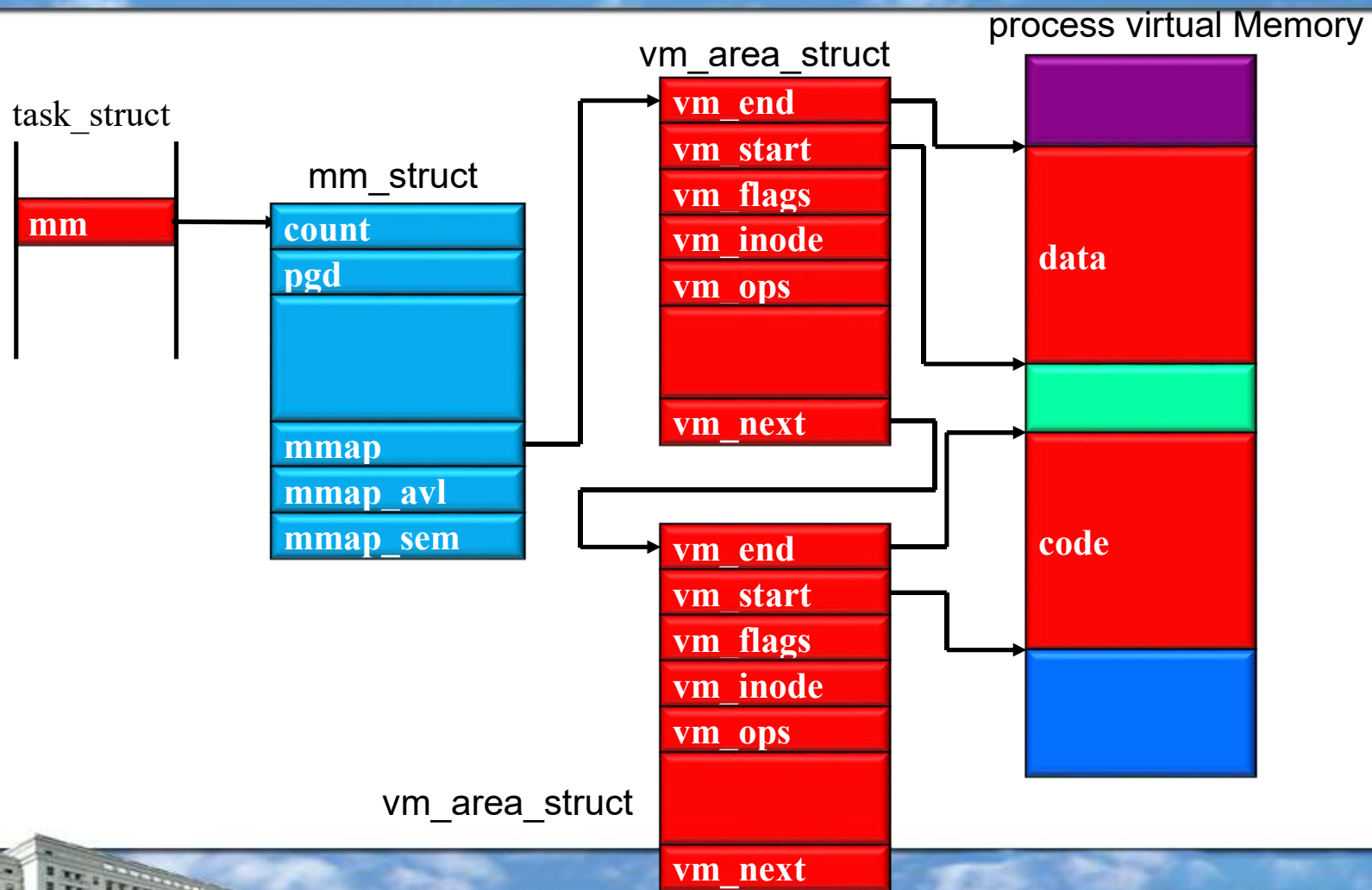
# 进程的实现

- 操作系统维护一张表格（进程表），每个进程占用一个表项（PCB）

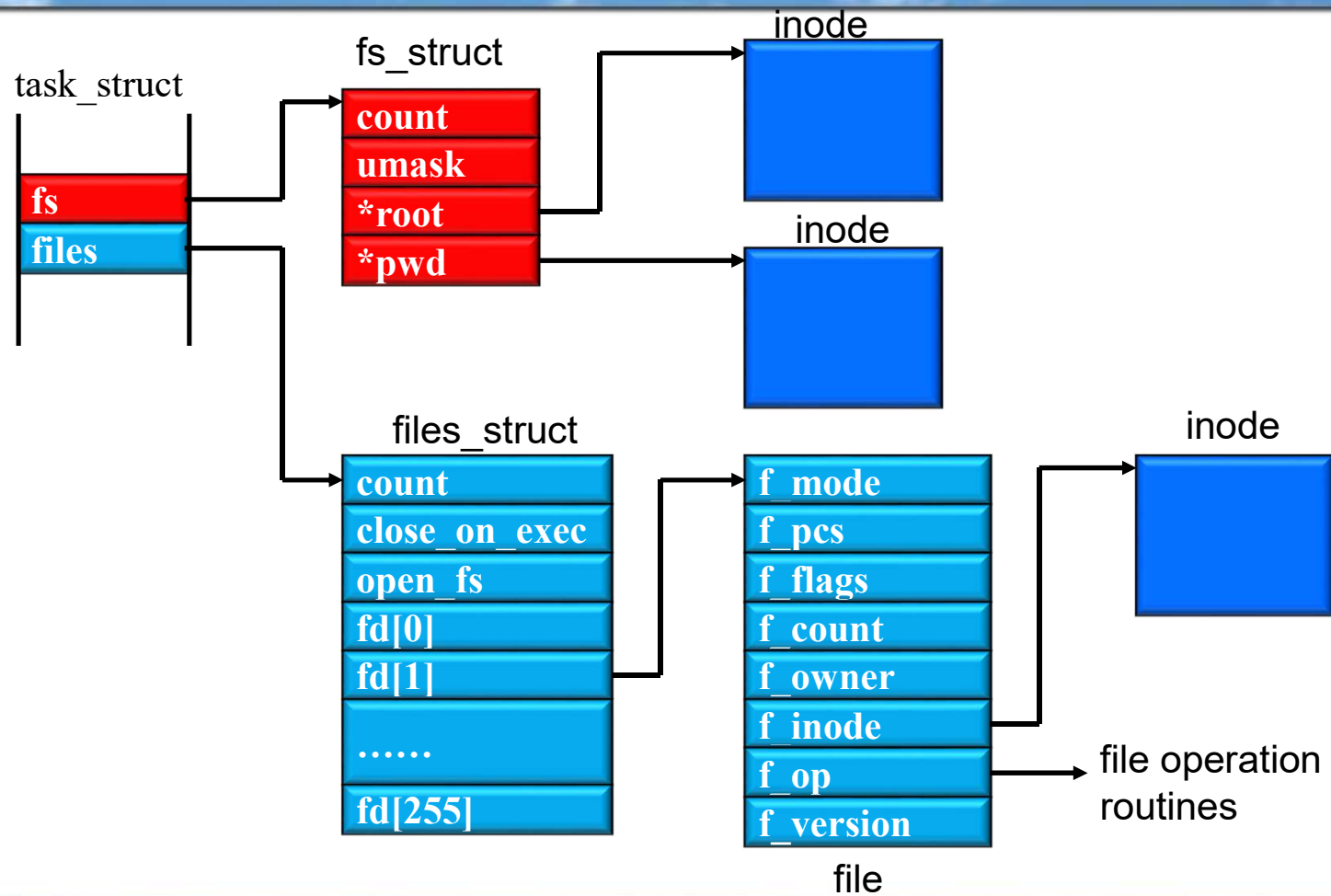
进程管理	存储管理	文件管理
寄存器 程序计数器 程序状态字 堆栈指针 进程状态 优先级 调度参数 进程ID 父进程ID 信号 进程开始时间 使用的CPU	正文段指针 数据段指针 堆栈段指针	根目录 工作目录 文件描述符 用户ID 组ID

图2-4 典型的进程表表项中的一些字段

# 进程的实现(Linux)



# 进程的实现(Linux)



## 4. 进程控制块的组织方式

1) 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。

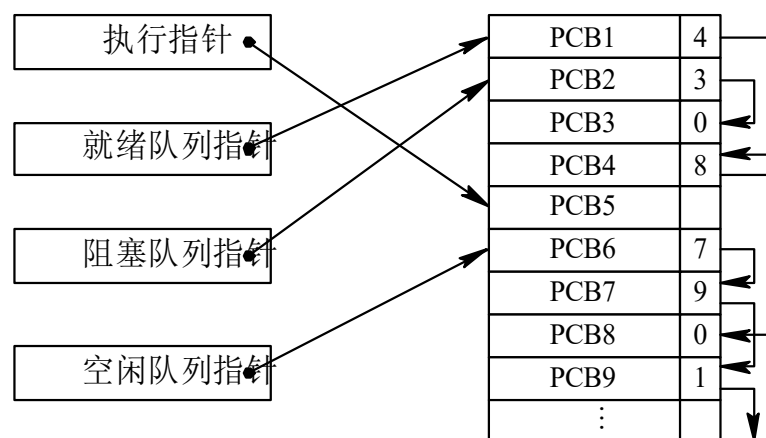


图 2-11 PCB链接队列示意图

2) 索引方式：相同状态进程的PCB组织在一张表格中，系统根据所有进程的状态建立几张索引表，系统分别记载各PCB表格的起始地址。

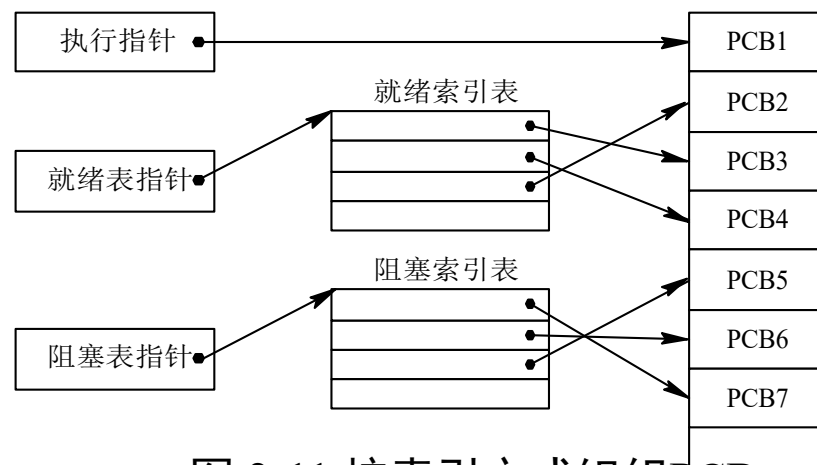


图 2-11 按索引方式组织PCB



# PCB实例解析

- Linux操作系统中采用task\_struct来表示进程控制块。
- Linux操作系统最多允许1024个进程，采用结构指针数组来表示。

– 定义格式：

```
struct task_struct *task[NR_TASKS];
```

其中， NR\_TASKS=512

– 例如，初始化进程在结构数组中的表示：

```
struct task_struct *task[1]=&init_task;
```

[Linux进程管理之task\\_struct结构体](#)



以下是对task\_struct的定义及注释：

```
1  struct task_struct {
2  volatile long state; //说明了该进程是否可以执行,还是可中断等信息
3  unsigned long flags; //Flage 是进程号,在调用fork()时给出
4  int sigpending; //进程上是否有待处理的信号
5  mm_segment_t addr_limit; //进程地址空间,区分内核进程与普通进程在内存存放的位置不同
6  //0-0xBFFFFFFF for user-thread
7  //0-0xFFFFFFFF for kernel-thread
8  //调度标志,表示该进程是否需要重新调度,若非0,则当从内核态返回到用户态,会发生调度
9  volatile long need_resched;
10 int lock_depth; //锁深度
11 long nice; //进程的基本时间片
12 //进程的调度策略,有三种,实时进程:SCHED_FIFO,SCHED_RR, 分时进程:SCHED_OTHER
13 unsigned long policy;
14 struct mm_struct *mm; //进程内存管理信息
15 int processor;
16 //若进程不在任何CPU上运行, cpus_runnable 的值是0, 否则是1 这个值在运行队列被锁时更新
17 unsigned long cpus_runnable, cpus_allowed;
18 struct list_head run_list; //指向运行队列的指针
19 unsigned long sleep_time; //进程的睡眠时间
20 //用于将系统中所有的进程连成一个双向循环链表,其根是init_task
21 struct task_struct *next_task, *prev_task;
22 struct mm_struct *active_mm;
23 struct list_head local_pages; //指向本地页面
24 unsigned int allocation_order, nr_local_pages;
25 struct linux_binfmt *binfmt; //进程所运行的可执行文件的格式
```

```
26 int exit_code, exit_signal;
27 int pdeath_signal; //父进程终止时向子进程发送的信号
28 unsigned long personality;
29 //Linux可以运行由其他UNIX操作系统生成的符合iBCS2标准的程序
30 int did_exec:1;
31 pid_t pid; //进程标识符,用来代表一个进程
32 pid_t pgrp; //进程组标识,表示进程所属的进程组
33 pid_t tty_old_pgrp; //进程控制终端所在的组标识
34 pid_t session; //进程的会话标识
35 pid_t tgid;
36 int leader; //表示进程是否为会话主管
37 struct task_struct *p_opptr,*p_pptr,*p_cpitr,*p_ysptr,*p_osptr;
38 struct list_head thread_group; //线程链表
39 struct task_struct *pidhash_next; //用于将进程链入HASH表
40 struct task_struct **pidhash_pprev;
41 wait_queue_head_t wait_chldexit; //供wait4()使用
42 struct completion *vfork_done; //供vfork()使用
43 unsigned long rt_priority; //实时优先级,用它计算实时进程调度时的weight值
44
45 //it_real_value, it_real_incr用于REAL定时器,单位为jiffies,系统根据it_real_value
46 //设置定时器的第一个终止时间. 在定时器到期时,向进程发送SIGALRM信号,同时根据
47 //it_real_incr重置终止时间, it_prof_value, it_prof_incr用于Profile定时器,单位为jiffies。
48 //当进程运行时,不管在何种状态下,每个tick都使it_prof_value值减一,当减到0时,向进程发送
49 //信号SIGPROF,并根据it_prof_incr重置时间。
50 //it_virt_value, it_virt_value用于Virtual定时器,单位为jiffies。当进程运行时,不管在何种
```



```
51 //状态下, 每个tick都使it_virt_value值减一当减到0时, 向进程发送信号SIGVTALRM, 根据
52 //it_virt_incr重置初值。
53 unsigned long it_real_value, it_prof_value, it_virt_value;
54 unsigned long it_real_incr, it_prof_incr, it_virt_value;
55 struct timer_list real_timer; //指向实时定时器的指针
56 struct tms times; //记录进程消耗的时间
57 unsigned long start_time; //进程创建的时间
58 //记录进程在每个CPU上所消耗的用户态时间和核心态时间
59 long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
60 //内存缺页和交换信息:
61 //min_flt, maj_flt累计进程的次缺页数(Copy on Write页和匿名页)和主缺页数(从映射文件
62 //设备读入的页面数); nswap记录进程累计换出的页面数, 即写到交换设备上的页面数。
63 //cmin_flt, cmaj_flt, cnswap记录本进程为祖先的所有子孙进程的累计次缺页数, 主缺页数和换
64 //在父进程回收终止的子进程时, 父进程会将子进程的这些信息累计到自己结构的这些域中
65 unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
66 int swappable:1; //表示进程的虚拟地址空间是否允许换出
67 //进程认证信息
68 //uid,gid为运行该进程的用户的用户标识符和组标识符, 通常是进程创建者的uid, gid
69 //euid, egid为有效uid,gid
70 //fsuid, fsgid为文件系统uid,gid, 这两个ID号通常与有效uid,gid相等, 在检查对于文件
71 //系统的访问权限时使用他们。
72 //suid, sgid为备份uid,gid
73 uid_t uid,euid,suid,fsuid;
74 gid_t gid,egid,sgid,fsgid;
75 int ngroups; //记录进程在多少个用户组中
```

```
76 gid_t groups[NGROUPS]; //记录进程所在的组
77 //进程的权能, 分别是有效位集合, 继承位集合, 允许位集合
78 kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
79 int keep_capabilities:1;
80 struct user_struct *user;
81 struct rlimit rlim[RLIM_NLIMITS]; //与进程相关的资源限制信息
82 unsigned short used_math; //是否使用FPU
83 char comm[16]; //进程正在运行的可执行文件名
84 //文件系统信息
85 int link_count, total_link_count;
86 //NULL if no tty 进程所在的控制终端, 如果不需要控制终端, 则该指针为空
87 struct tty_struct *tty;
88 unsigned int locks;
89 //进程间通信信息
90 struct sem_undo *semundo; //进程在信号灯上的所有undo操作
91 struct sem_queue *semsleeping; //当进程因为信号灯操作而挂起时, 他在该队列中记录等待的操作
92 //进程的CPU状态, 切换时, 要保存到停止进程的task_struct中
93 struct thread_struct thread;
94 //文件系统信息
95 struct fs_struct *fs;
96 //打开文件信息
97 struct files_struct *files;
98 //信号处理函数
99 spinlock_t sigmask_lock;
100 struct signal_struct *sig; //信号处理函数
```

```
101 sigset_t blocked; //进程当前要阻塞的信号，每个信号对应一位
102 struct sigpending pending; //进程上是否有待处理的信号
103 unsigned long sas_ss_sp;
104 size_t sas_ss_size;
105 int (*notifier)(void *priv);
106 void *notifier_data;
107 sigset_t *notifier_mask;
108 u32 parent_exec_id;
109 u32 self_exec_id;
110
111 spinlock_t alloc_lock;
112 void *journal_info;
113 };
```

## 2.3 进程控制

### 2.3.2 进程的创建

#### 1. 进程图(Process Graph) ■

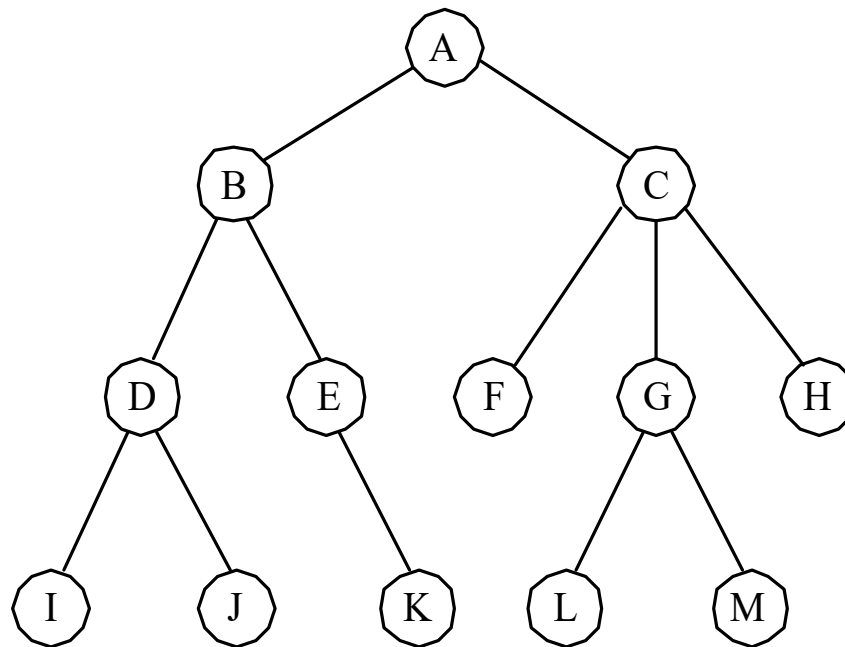


图 2-13 进程树

## 2.3.2 进程的创建

### 1. 进程图 (Process Graph)

- 进程图是用于描述一个进程的家族关系的有向树。
- 子进程可以继承父进程所拥有的资源。
- 当子进程被撤消时，应将其从父进程那里获得的资源归还给父进程。
- 在撤消父进程时，也必须同时撤消其所有的子进程？



## 2.引起创建进程的事件

- 导致一个进程去创建另一个进程的典型事件，可有以下四类：
  - (1) 用户登录。
  - (2) 作业调度。
  - (3) 提供服务。例如：I/O请求
  - (4) 应用请求。基于应用进程的需求，由它自己创建一个新进程，以便使新进程以并发运行方式完成特定任务。

### 3. 进程的创建(Creation of Progress)

调用进程创建原语Creat（）步骤：

(1)申请空白PCB：申请标识符和空白PCB结构

(2) 为新进程分配资源。

(3) 初始化进程控制块。包括：标识信息、处理机状态信息、处理机控制信息

(4) 如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。启动调度。

Linux用fork()函数创建进程

# 进程创建举例

- `fork()`:复制父进程全部资源。
- `clone()`:有选择的复制父进程的资源。(带参数)  
`clone(int(*fn)(void *arg),void *child stack,int flags,void *arg)`
- `vfork()`: 创建一个线程。复制除`tast_struct`和系统空间堆栈外的所有资源。
- 这三个系统调用都是通过调用`do_fork()`实现, 只是参数不同。



# Linux进程空间获取讲解

- 申请空白PCB详细阐述：

Do\_fork()  $\longrightarrow$  alloc\_task\_struct(void)

$\longrightarrow$  \_get\_free\_pages()申请两个页面，其中控制块占用1.5K，其余空间为进程的内核栈空间。

```
#include <stdio.h>
```

```
int main( )  
{
```

```
    int child;
```

```
    char *args[ ] = {"/bin/echo", "Hello", "World!", NULL};
```

```
    if ( !(child = fork( )) )  
    {
```

```
        /* child */
```

```
        printf("pid %d: %d is my father\n", getpid( ), getppid( ));
```

```
        execve("/bin/echo", args, NULL);
```

```
        printf("pid %d: I am back, something is wrong!\n", getpid( ));
```

```
    }
```

```
    else
```

```
    {
```

```
        int myself = getpid( );
```

```
        printf("pid %d: %d is my son\n", myself, child);
```

```
        wait4(child, NULL, 0, NULL);
```

```
        printf("pid %d: done\n", myself);
```

```
    }
```

```
    return 0;
```

```
}
```

子进程从**fork**返回时，返回值为**0**；  
父进程从**fork()**返回时，返回值为子  
进程的**id**时，非**0**。

通过系统调用使子进程  
执行某段可执行程序。

等待子进程去世。

## 2.3.3 进程的终止

- 引起进程终止的三类事件

- 1) 正常结束。

- 批处理系统: Holt指令或终止的系统调用
    - 分时系统: Logs off

- 2) 异常结束:

- ①越界错误。
    - ②保护错。
    - ③非法指令。
    - ④特权指令错。
    - ⑤运行超时。
    - ⑥等待超时。
    - ⑦算术运算错、被0除:
    - ⑧I/O故障。

## 2.2.2 进程的终止

### 3) 外界干预

外界干预并非指在本进程运行中出现了异常事件，而是指进程应外界的请求而终止运行。

①操作员或操作系统干预。

由于某种原因，例如，发生了死锁，由操作员或操作系统终止该进程；

②父进程请求终止该进程；

③当父进程终止时，OS也将他的所有子孙进程终止。？

## 2. 进程的终止过程

(1) 根据被终止进程的PID找到它的PCB，从中读出该进程的状态。

(2) 若被终止进程正处于执行状态，应立即终止该进程的执行，重新进行调度。

(3) 若该进程还有子孙进程，立即将其所有子孙进程终止。？

(4) 将被终止进程所拥有的全部资源，归还给其父进程，或者归还给系统。

(5) 将被终止进程的PCB从所在队列中移出。

Linux中，系统调用exit（）结束进程。

# Linux进程终止

- 1>正常退出
  - a. 在main()函数中执行return 。
  - b.调用exit()函数
  - c.调用\_exit()函数
- 2>异常退出
  - a.调用abort函数
  - b.进程收到某个信号，而该信号使程序终止。

课后学习： exit()、\_exit()、 return的区别及各自的使用方法。

## 2.2.3 进程的阻塞与唤醒

### 1. 引起进程阻塞的事件

- 1) 向系统请求共享资源失败
- 2) 等待某种操作完成，如I/O操作。
- 3) 新数据尚未到达
- 4) 无新工作可做



## 2. 进程阻塞过程

① 正在执行的进程，当发现上述某事件时，由于无法继续执行，于是进程便通过调用阻塞原语**block**把自己阻塞；（阻塞是主动行为）



② 把进程控制块中的现行状态由“执行”改为阻塞，并将**PCB**插入阻塞队列；



③ 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。

# 进程唤醒过程

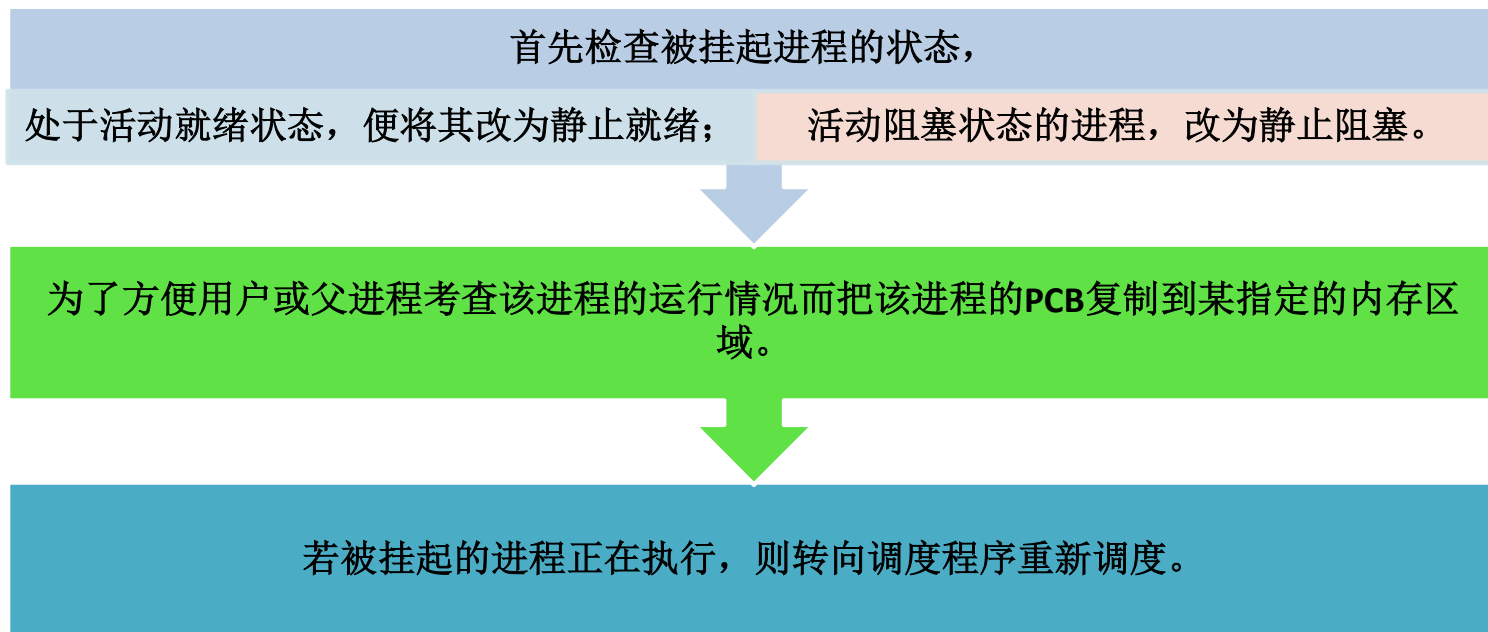
- ① 当被阻塞进程所期待的事件出现时，则由有关进程（比如，用完并释放了该I/O设备的进程）调用唤醒原语wakeup( )，将等待该事件的进程唤醒。
- ② 唤醒原语执行的过程是：
  - 首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态由阻塞改为就绪，
  - 然后再将该PCB插入到就绪队列中。

## 2.2.4 进程的挂起与激活

- 1. 进程的挂起

当出现了引起进程挂起的事件时，系统将利用挂起原语suspend( )将指定进程挂起。

- 挂起原语的执行过程是：



## 2. 进程的激活过程

- ①当发生激活进程的事件时，例如，父进程或用户进程请求激活指定进程，若该进程驻留在外存而内存中已有足够的空间时，则可将在外存上处于静止就绪状态的进程换入内存。这时，系统将利用激活原语active( )将指定进程激活。
- ②激活原语先将进程从外存调入内存，检查该进程的现行状态
  - 若是静止就绪，便将之改为活动就绪；
  - 若为静止阻塞，便将之改为活动阻塞。

# Linux的挂起和激活

Linux中挂起原语：

`int pause(void)`使调用进程挂起直到捕捉到一个信号，则被自动解挂；

`int sigsuspend(const sigset_t *sigmask)` 函数接受一个信号集指针，将信号屏蔽字设置为信号集中的值，在进程接受到一个信号之前，进程会挂起。

\*练习Linux的进程创建、终止、挂起、解挂等原子操作的使用方法。

# 进程控制原语可能引起的调度

时机：假如采用的是抢占调度策略，则每当有新进程进入就绪队列时，都应检查是否要进行重新调度。

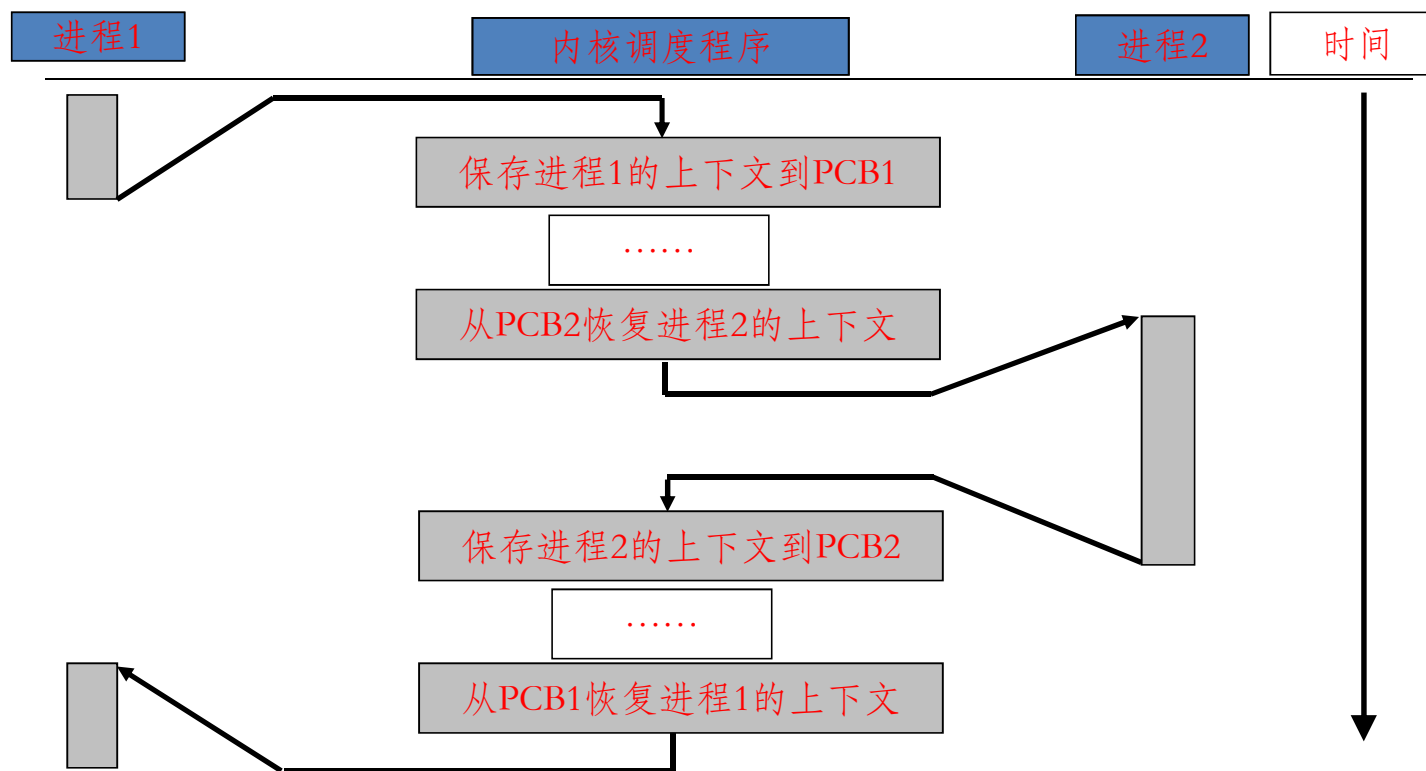
创建、终止（自己）、挂起（自己）、激活、阻塞、唤醒都可能会产生新的调度。

# 进程切换

- 当一个进程占处理机执行完（或不能继续执行），则换另一个进程占处理机执行，称为进程切换。
  - 把处理机分配给不同的进程占用执行，称为**进程调度**。
  - 实现分配处理机的程序称为**调度程序**。
  - 在进程切换时，要保护执行现场，称为**进程的上下文**。



# 进程切换过程



# 进程切换基本步骤

- 1 保存进程上下文环境
- 2 更新当前运行进程的控制块内容，将其状态改为就绪或阻塞状态
- 3 将进程控制块移到相应队列（就绪队列或阻塞队列）
- 4 改变需投入运行进程的控制块内容，将其状态变为运行状态
- 5 恢复需投入运行进程的上下文环境

# Linux 的进程切换过程

- Linux进程切换（context\_switch()）本质上两步：
- 1) 进程页表PGD切换；
- 2) 内核态堆栈和硬件上下文切换（包括CPU寄存器）；
- context\_switch()通过调用switch\_mm()切换进程空间，调用switch\_to切换内核上下文环境。

<http://blog.csdn.net/xiaoxiaomuyu2010/article/details/11935393>



## 2.4 进程同步



# ? (问题引入)

- 采用多道程序设计技术的操作系统，允许多个进程同时驻留内存并发执行。
  - ? 如何协调多个进程对系统资源，如内存空间、外部设备等的竞争和共享？
  - ? 如何解决多个进程因为竞争资源而出现执行结果异常，甚至导致系统不稳定、失效等问题？
  - ? 例如，多个进程同时申请文件打印，如何有效分配打印机？
- 1. 两种形式的制约关系
  - (1) 间接相互制约关系。共享系统资源
  - (2) 直接相互制约关系。进程间合作

# 例1

- 银行的联网储蓄业务允许储户同时用储蓄卡和存折对同一帐户进行存取款操作，如果某储户同时（在ATM机和营业柜台）办理两笔存款业务（假设分别为1000和2000元）
- 从系统的角度看，有两个进程将同时对储户余额等数据进行修改。如果两个进程同时读出原余额（假设为5000元），两个进程分别将最新余额修改为6000（ $5000+1000$ ）和7000（ $5000+2000$ ）。



# 分析及措施

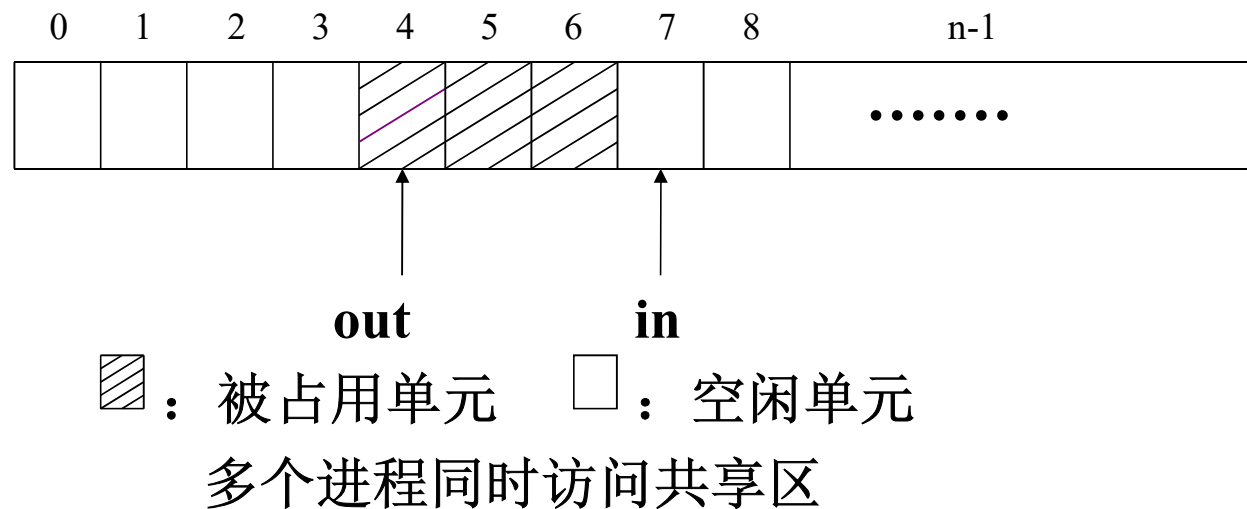
- 最后，储户余额可能是6000，或者7000，显然都不正确。
- **原因：两个进程同时修改同一数据，而没有进行有效控制。**
- 正确的方法：如果有多个进程需要同时修改某一数据，系统必须控制，一次仅允许一个进程完成读数据、修改数据两件事以后，才允许别的进程对同一数据的读和修改操作。

## 例2

- 假设系统中有3个进程P1、P2、P3，其中P1和P2是计算进程，P3是打印进程，每当P1或P2计算出一个结果以后，将计算结果送到缓存区中，以便P3进程从中取出数据打印。
- 假设缓冲区被划分为0、1、2...n-1共n个单元。
- 设两个指针：
  - in指针用于计算进程存放数据，指向缓冲区中下一个空闲的单元，
  - out指针用于打印进程取数据，指向缓冲区中下一个将取走数据的单元。

## 例2

- 假设某时刻，0到3号单元空闲，4到6号单元被占用，这时候P1、P2进程都准备将数据送入缓冲区，如下图所示。



# 分析

- 该例中，由于进程P1和P2共享缓冲区和位置指针，而未对这种共享进行有效控制，导致打印数据的丢失。
- 如果控制进程P1、P2互斥地访问缓冲区和修改位置指针，将避免这种因为并发执行而导致的程序执行结果的不确定性。

# 结论

通过上述两个例子可见，采用多道程序并发设计技术的操作系统对诸进程的并发控制是非常重要的和必需的。

# 并发控制 - 竞争资源

- 当并发进程竞争使用同一资源时，它们之间就会发生冲突。
- 如果操作系统将资源分配给其中的某一个进程使用，另一个进程就必须等待，直到申请的资源可用时，由操作系统分配给它。
- 如果竞争某资源的进程太多，这些进程还必须等待在一个队列中，如就绪队列，阻塞队列等。
- 极端情况下，被阻塞进程永久得不到申请的资源，而**死锁**。





## 2.临界资源

- 进程竞争资源首先必须解决“互斥”问题。某些资源必须互斥使用，如打印机、共享变量、表格、文件等。
- 这类资源又称为临界资源，
- 访问临界资源的那段代码称为临界区。
- 任何时刻，只允许一个进程进入临界区，以此实现进程对临界资源的互斥访问。

### 3. 临界区(critical section)

可把一个访问临界资源的循环进程描述如下：

repeat

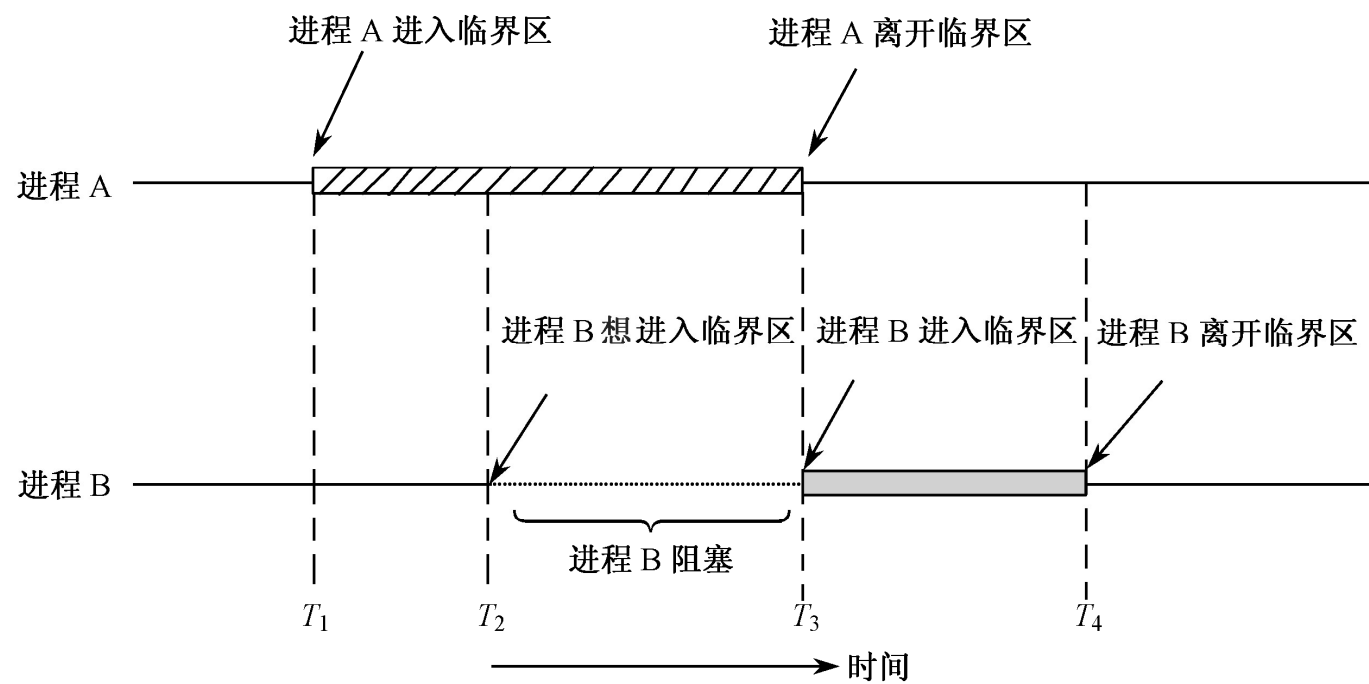
entry section

critical section;

exit section

remainder section;

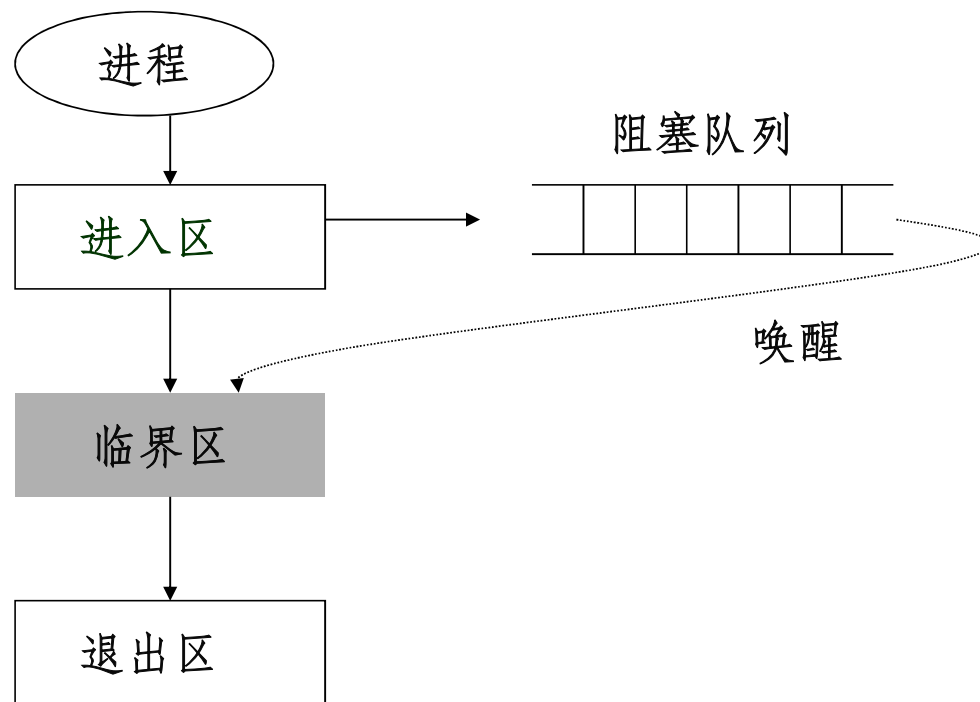
until false;



进程A和进程B互斥使用临界区的过程

# 互斥使用临界资源

- 首先，在“进入区”判断是否可以进入临界区，
  - 如果可以进入，则必须设置临界区使用标志，阻止其它后来的进程进入临界区。
  - 后来的进程通过查看临界区使用标志，知道自己不能进入临界区，就进入阻塞队列，将自己阻塞。
- 当临界区内的进程使用完毕，退出临界区时，即在“退出区”修改临界区使用标志，并负责唤醒阻塞队列中的一个进程，让其进入临界区。



进程互斥进入临界区

## 4. 临界区使用原则（也称为互斥条件）

- (1) **空闲让进**。如果临界区空闲，则只要有进程申请就立即让其进入，以有效利用资源。
- (2) **忙则等待**。每次仅允许一个进程处于临界区，保证对临界资源的“互斥”访问。
- (3) **有限等待**。进程只能在临界区内逗留有限时间，不得使其他进程在临界区外陷入“死等”。
- (4) **让权等待**。进程不能进入临界区时，应立即释放处理机，以免陷入“忙等”状态。





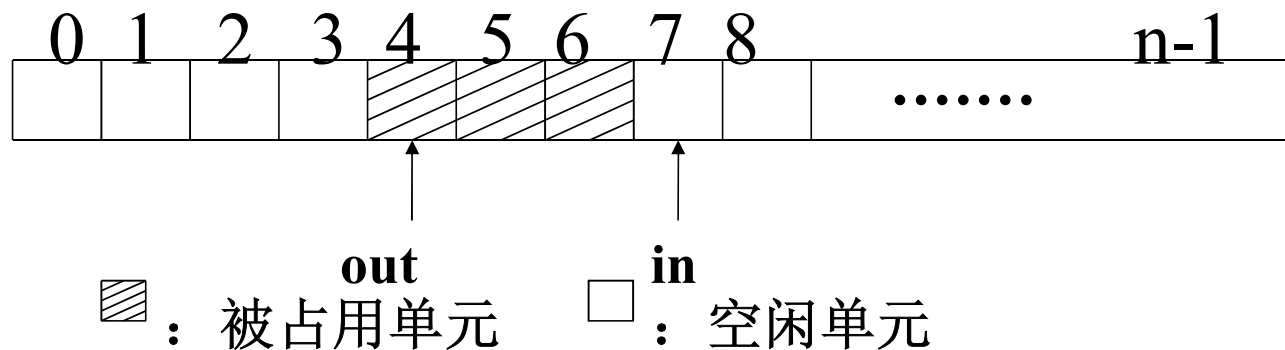
## 例2 生产者-消费者（P49）

- 有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有 $n$ 个缓冲区的缓冲池：
- 生产者进程将它所生产的产品放入一个缓冲区中；
- 消费者进程可从一个缓冲区中取走产品去消费。
- 它们之间必须保持同步原则：不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

- 用一个数组来表示上述的具有 $n$ 个 $(0, 1, \dots, n-1)$ 缓冲区的缓冲池。
- 设置两个指针：
  - 指针 $in$ ：指示下一个可投放产品的缓冲区，每当生产者进程生产并投放一个产品后， $in$ 指针加1，即 $in := (in+1) \bmod n$ ；
  - 指针 $out$ ：指示下一个可从中获取产品的缓冲区，每当消费者进程取走一个产品后， $out$ 指针加1，即 $out := (out+1) \bmod n$ 。
- 当 $(in+1) \bmod n = out$ 时表示缓冲池满；而 $in = out$ 则表示缓冲池空。

# 生产者和消费者两进程使用的变量

```
var n integer;  
type item=...;  
var buffer:array [0, 1, ..., n-1] of item;  
in, out: 0, 1, ..., n-1;  
counter: 0, 1, ..., n; // 缓冲区中产品的个数, 初始值为0
```



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;

- 若并发执行时，就会出现差错，问题就在于这两个进程共享变量counter。生产者对它做加1操作，消费者对它做减1操作，这两个操作在用机器语言实现时，常可用下面的形式描述：

### 生产者

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

### 消费者

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

假设：counter的当前值是5。

无论生产者先执行，还是消费者先执行，结果都是5.

- 但是，如果按下述顺序执行：

register <sub>1</sub> :	= counter;	(register <sub>1</sub> = 5)
register <sub>1</sub> :	= register <sub>1</sub> + 1;	(register <sub>1</sub> = 6)
register <sub>2</sub> :	= counter;	(register <sub>2</sub> = 5)
register <sub>2</sub> :	= register <sub>2</sub> - 1;	(register <sub>2</sub> = 4)
counter :	= register <sub>1</sub> ;	(counter = 6)
counter :	= register <sub>2</sub> ;	( <del>counter</del> = 4)

- Counter的值最终为4。
- 问题的原因是什么？该怎么办？
- 将counter作为临界资源处理，使生产者和消费者互斥的访问该变量。



## 2.4.2 硬件同步机制

### 1、屏蔽中断

- 每个进程刚进入临界区立即屏蔽所有中断（包括时钟中断），离开之前打开中断
  - ① 由用户进程屏蔽中断不理智
  - ② 屏蔽中断仅对执行disable指令的CPU有效

### 2. 锁变量

- 软件解决方案：假设有一个共享（锁）变量，其初始值为0，任何希望进入临界区的进程，首先测试该锁，若为0，则设置其为1，然后进入，否则等待。（该方案与前述假脱机目录一样缺陷）

## 2.4.2 硬件同步机制

### 3. 严格轮换法

- 用于忙等待的锁，称为**自旋锁**
- 这种解决方案违反了前述条件2：**忙则等待。**

<pre>while (TRUE) {     while (turn != 0)    /* loop */ ;     critical_region();     turn = 1;     noncritical_region(); }</pre>	<pre>while (TRUE) {     while (turn != 1)    /* loop */ ;     critical_region();     turn = 0;     noncritical_region(); }</pre>
(a)	(b)

图 a) 进程0    b) 进程1，请注意分号终止了while语句

## 2.4.2 硬件同步机制

### 4、TSL指令

- TSL RX, lock
  - 测试并加锁，方法是锁住内存线，与屏蔽中断不同

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered
```

需要忙等待

```
leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

图2-25 用TSL指令进入和离开临界区

# 睡眠与唤醒

- 前述方案共同缺点：**忙等待**。即：若某进程希望进入临界区，先测试，允许则进入，否则原地等待；
  - 这种方法浪费CPU，而且可能出现预想不到的问题，如“**优先级反转问题**”
  - 解决忙等待的方法：当无法进入临界区时，**阻塞**进程，直至另一进程将其**唤醒**，相应的通信原语为sleep\wakeup

# 睡眠与唤醒

- 生产者-消费者问题，也称为有界缓冲区问题
- 两个进程共享一个固定大小的缓冲区，其中一个是生产者，将信息放入缓冲区，另一个是消费者，从缓冲区中取出信息
- 问题：当缓冲区满，而此时生产者希望向其中放入一个新的数据项，或者当消费者试图从缓冲区取数据而发现缓冲区为空。

## 2.3.4 睡眠与唤醒

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                           /* repeat forever */
        item = produce_item();               /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(item);                   /* put item in buffer */
        count = count + 1;                   /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                           /* repeat forever */
        if (count == 0) sleep();              /* if buffer is empty, got to sleep */
        item = remove_item();                /* take item out of buffer */
        count = count - 1;                   /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                  /* print item */
    }
}
```

图 2.27 含有严重竞争条件的生产者、消费者问题



# 睡眠与唤醒

- 上述解法含有含有严重的**竞争条件**：
- 当缓冲区为空时，消费者读取count为0，此时**进程切换**至生产者，加入一个数据项，count加1，并调用wakeup唤醒消费者。由于此时消费者并未睡眠，因此该**唤醒信号丢失**。当消费者重新被调度运行，执行sleep睡眠。生产者继续运行，迟早会填满缓冲区。
- 一种快速的弥补方法是，设立一个“**唤醒等待位**”

## 2.4.3 信号量机制

- 软件方法和硬件方法都存在“忙等”问题，浪费了处理机时间。
- 信号量方法能实现进程互斥与同步，而不必“忙等”

# 信号量实现互斥的基本原理

- 两个或多个进程可以通过传递信号进行合作，可以迫使进程在某个位置暂时停止执行（阻塞等待），直到它收到一个可以“向前推进”的信号（被唤醒）。
- 相应地，将实现信号灯作用的变量称为信号量。

## 2.4.3 信号量机制

### 1. 整型信号量

最初由Dijkstra把整型信号量定义为一个整型量，除初始化外，仅能通过两个标准的原子操作(Atomic Operation)  $\text{wait}(S)$ 和 $\text{signal}(S)$ 来访问。这两个操作一直被分别称为P、V操作。 $\text{wait}$ 和 $\text{signal}$ 操作可描述为：

$\text{wait}(S)$ : while  $S \leq 0$  do no-op/\*忙等\*/

$S := S - 1$ ;

$\text{signal}(S)$ :  $S := S + 1$ ;

# 定义对信号量的两个原子操作

## wait(s) 和signal(s)

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

## 2. 记录型信号量

- **定义：**记录型信号量，其中一个域为整型，另一个域为队列，其元素为等待该信号量的阻塞进程（FIFO）。
- 记录型信号量机制，是一种不存在“忙等”现象的进程同步机制。
- 记录型信号量的数据结构：

```
type semaphore=record
    value : integer; /*信号量的值*/
    L: list of process; /*等待该信号量的阻塞进程*/
end
```



## 2. 记录型信号量

- 记录型信号量的wait(s)操作

```
procedure wait(s)
```

```
  var s: semaphore;
```

```
  begin
```

```
    s.value:=s.value-1;
```

```
    if s.value<0 then block (s.L) ;
```

```
  end
```

- 进程进入临界区之前，首先执行wait(s)原语，若 $s.value < 0$ ,则进程调用阻塞原语，将自己阻塞，并插入到 s.L队列排队，体现了“让权等待”准则。
- 直到某个从临界区退出的进程执行signal(s)原语，唤醒它。

## 2. 记录型信号量

- 记录型信号量的signal(s)操作

```
procedure signal(s)
```

```
var s:semaphore;
```

```
begin
```

```
  s.value:=s.value+1;
```

```
  if s.value≤0 then wakeup(s.L); /*绝对值表示阻塞队列中的进程数量*/
```

```
end
```

- 如果s.value的初始值为1，此时转化为互斥信号量，进行进程互斥；

# 信号量的类型

- 信号量分为：**互斥信号量**和**资源信号量**。
- 互斥信号量：用于申请或释放资源的使用权，常初始化为1。
- 资源信号量：用于申请或归还资源，可以初始化为大于1的正整数，表示系统中某类资源的可用个数。
- 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
  - 然后将各进程访问该资源的临界区CS置于wait（mutex）和signal（mutex）操作之间即可。
  - 利用信号量实现进程互斥的进程可描述如下：

# 利用信号量实现互斥的通用模式

```
program mutualexclusion;
const n=...;    /* 进程数 */
var s: semaphore(:= 1); /* 定义信号量s, S.value初始化为1 */
procedure P(i:integer);
begin
  repeat
    wait(s);
    <临界区>;
    signal(s);
    <其余部分>
  forever
end;
begin /* 主程序 */
  parbegin
    P(1); P(2); ... P(n)
  parend
end.
```


# 信号量的物理意义

- $s.value \geq 0$  ，表示还可执行wait(s)而不会阻塞的进程数（可用资源数）。
  - 每执行一次wait(s)操作，就意味着请求分配一个单位的资源
- $s.value < 0$  ，表示已无资源可用，因此请求该资源的进程被阻塞。
  - 此时，  $s.value$ 的绝对值等于该信号量阻塞队列中的等待进程数。执行一次signal操作，就意味着释放一个单位的资源。

## S.value的取值范围

- 当仅有两个并发进程共享临界资源时，互斥信号量仅能取值0、1、-1。其中，
  - $s.value = 1$ , 表示无进程进入临界区
  - $s.value = 0$ , 表示已有一个进程进入临界区
  - $s.value = -1$ , 则表示已有一进程正在等待进入临界区
- 当用s来实现n个进程的互斥时， $s.value$ 的取值范围为 $1 \sim -(n-1)$



- 
- 操作系统内核以系统调用形式提供wait和signal原语，应用程序通过该系统调用实现进程间的互斥。
  - 工程实践证明，利用信号量方法实现进程互斥是高效的，一直被广泛采用。



# Linux中的信号量

Linux中使用信号量实现进程间对共享资源的互斥访问，Down()和Up()函数对应于P、V操作。

```
struct Semaphore {  
    atomic_t count;  
    int sleepers;  
    wait_queue_head_t wait;  
#if WAITQUEUE_DEBUG  
    long __magic;  
#endif  
};
```

上述进程互斥只针对一个临界资源而言，如果在两个进程中都要包含两个对Dmutex和Emutex的操作，即

```
process A:      process B:  
wait(Dmutex);   wait(Emutex);  
wait(Emutex);   wait(Dmutex);
```

若进程A和B按下述次序交替执行wait操作：

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

### 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中需要的所有资源，一次性全都地分配给进程，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，其它所有可能为之分配的资源，也不分配给他。
- **原子操作**：要么全部分配到进程，要么一个也不分配。
- 在wait操作中，增加了一个“AND”条件，故称为AND同步，或称为**同时wait操作**。

### 3. AND型信号量

Swait ( $S_1, S_2, \dots, S_n$ )

if  $S_1 \geq 1$  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

### 3. AND型信号量

Ssignal ( $S_1, S_2, \dots, S_n$ )

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. 信号量集

- 在记录型信号量机制中，wait(s)或signal(s)操作仅能对信号量施以加1 或减1 操作，意味着每次只能获得或释放一个单位的临界资源。
- 在每次分配时，采用信号量集来控制，可以分配多个单位的资源。

## 4. 信号量集

- 在每次分配时，采用信号量集来控制，可以分配多个资源。

Swait ( $S_1, t_1, d_1, \dots, S_n, t_n, d_n$ ) (满足  $t_i \geq d_i$ )

if  $S_1 \geq t_1$  and...and  $S_n \geq t_n$  then

for  $i: =1$  to  $n$  do

$S_i := S_i - d_i$ ;

endfor

else

Place the executing process in the waiting queue of the first  $S_i$  with  $S_i < t_i$  and set its program counter to the beginning of the Swait operation。

endif

需求量

资源下限值

# 资源释放操作（ Ssignal ）：


Ssignal (  $S_1, d_1, \dots, S_n, d_n$  )

for i: =1 to n do

$S_i := S_i + d_i$ ;

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

endfor;




一般“信号量集”的几种特殊情况：

(1)  $\text{Swait}(S, d, d)$ 。此时在信号量集中只有一个信号量 $S$ ，但允许它每次申请 $d$ 个资源，当现有资源数少于 $d$ 时，不予分配。

(2)  $\text{Swait}(S, 1, 1)$ 。此时的信号量集已蜕化为一般的记录型信号量( $S > 1$ 时)或互斥信号量( $S = 1$ 时)。

(3)  $\text{Swait}(S, 1, 0)$ 。这是一种很特殊且很有用的信号量操作。当 $S \geq 1$ 时，允许多个进程进入某特定区；当 $S$ 变为0后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。



## 2.4.4 信号量的应用

1. 利用信号量实现进程互斥
2. 利用信号量实现前趋关系

## 2.4.4 信号量的应用

### 1. 利用信号量实现进程互斥

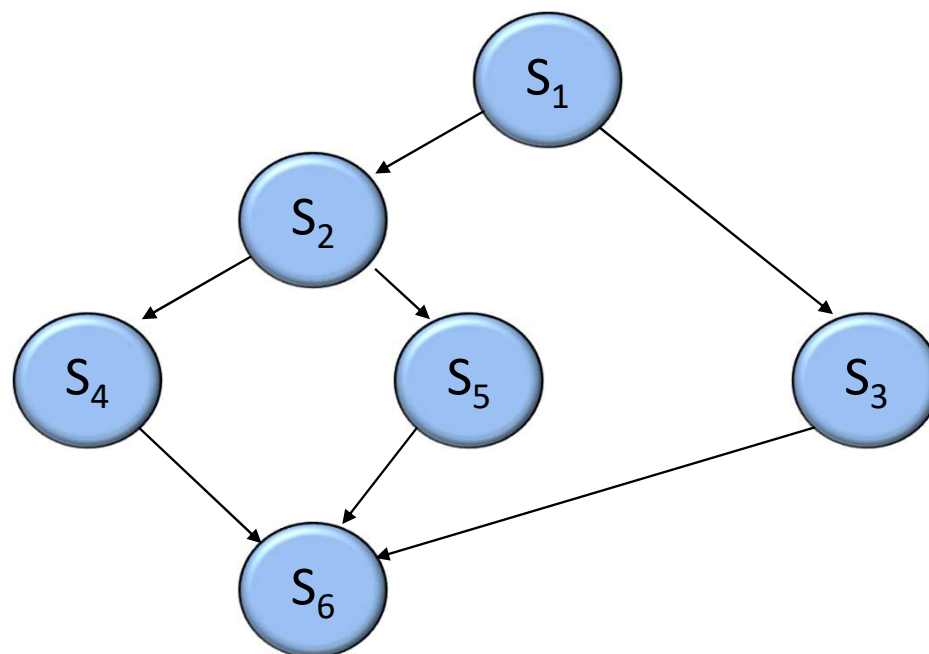
为使多个进程能互斥地访问某临界资源，只须为该资源设置一互斥信号量mutex，并设其初始值为1，然后将各进程访问该资源的临界区CS置于wait（mutex）和signal（mutex）操作之间即可。



```
var mutex: semaphore:=1;  
begin  
  parbegin  
    process1: begin  
      repeat  
        wait (mutex) ;  
        critical section  
        signal (mutex) ;  
        remainder section  
      until false;  
    end  
    process2: begin  
      repeat  
        wait (mutex) ;  
        critical section  
        signal (mutex) ;  
        remainder section  
      until false;  
    end  
  parend  
end
```

## 2. 利用信号量实现前趋关系

- 利用信号量写出图2\_12的并发执行的程序



```
Var a,b,c,d,e,f,g: semaphore: =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
```

## 2.4.5 管程

- 需要知道管程也是一种进程同步的机制。
- 其他管程相关的内容为自学内容。

## 2.5 经典进程的同步问题

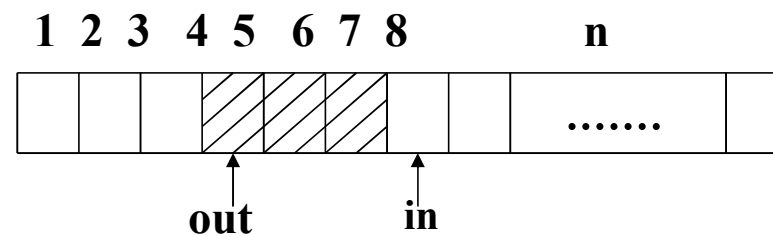
### 2.5.1 生产者—消费者问题(The producer-consumer problem)

- ❖ 生产者与消费者是一个广义的概念，可以代表一类具有相同属性的进程。
- ❖ 生产者和消费者进程共享一个大小固定的缓冲区，一个或多个生产者生产数据，并将生产的数据存入缓冲区，并有一个消费者从缓冲区中取数据。

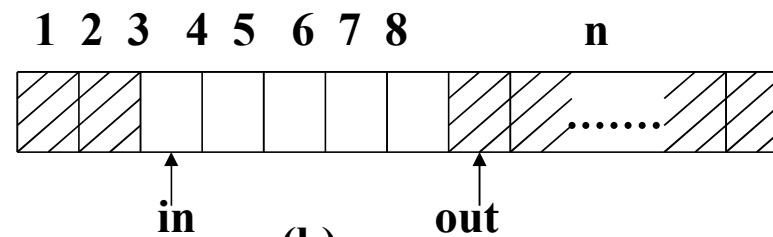
例如，在输入时，输入进程是生产者，计算进程是消费者；而在输出时，则计算进程是生产者，而打印进程是消费者。

- 假设缓冲区的大小为 $n$ （存储单元的个数），它可以被生产者和消费者循环使用。
- 分别设置两个指针 $in$ 和 $out$ ：
  - $in$ 指向生产者将存放数据的存储单元
  - $out$ 指向消费者将取数据的存储单元





(a)



(b)

其中，**in**表示存数据位置，**out**表示取数据位置



 : 被占用单元,  : 空存储单元

图 生产者/消费者循环使用缓冲区

# 不控制生产者/消费者？

- 试想，如果不控制生产者与消费者，将会产生什么结果？
- 生产者和消费者可能同时进入缓冲区，甚至可能同时读/写一个存储单元，将导致执行结果不确定。
- 这显然是不允许的。

# 生产者/消费者必须同步/互斥

- 必须使生产者和消费者 互斥进入缓冲区。即，某时刻只允许一个实体（生产者或消费者）访问缓冲区，生产者互斥消费者和其它任何生产者。
- 生产者不能向满缓冲区写数据，消费者也不能在空缓冲区中取数据，即生产者与消费者必须 同步。



- 互斥信号量mutex：实现诸进程对缓冲池的互斥使用；
- 资源信号量empty：表示缓冲池中空缓冲区的数量；
- 资源信号量full：表示满缓冲区的数量；
- 只要缓冲池未滿，生产者便可将消息送入缓冲池；
- 只要缓冲池未空，消费者便可从缓冲池中取走一个消息。

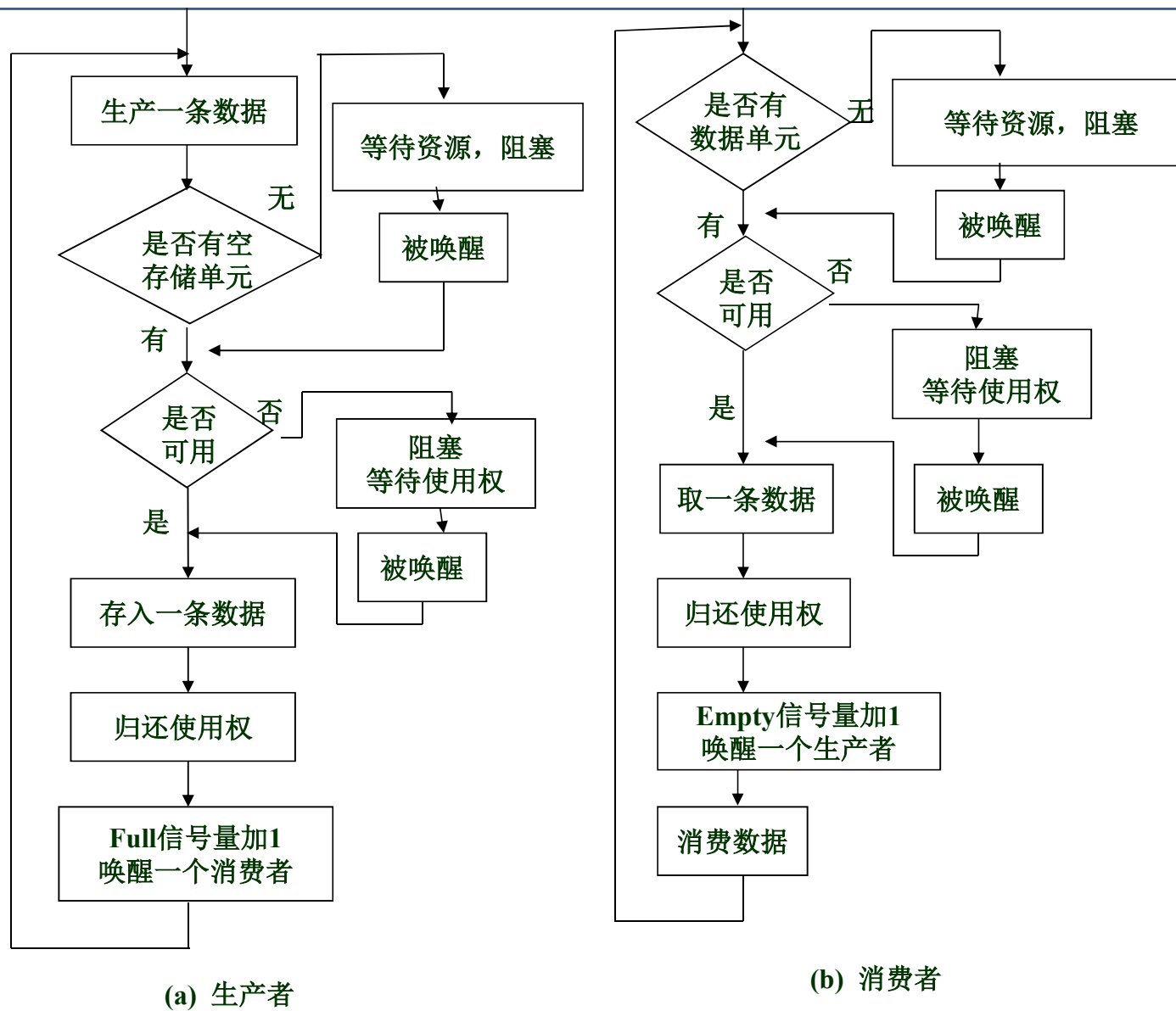


图 生产者/消费者执行流程图

# 生产者

```
var mutex, empty, full: semaphore: =1, n, 0;
    buffer: array[0, ..., n-1] of item;
    in, out: integer: =0, 0;
begin
    parbegin
        proceducer: begin
            repeat
                ⋮
            producer an item nextp;
                ⋮
            wait (empty) ;
            wait (mutex) ;
            buffer (in) : =nextp;
            in: = (in+1) mod n;
            signal (mutex) ;
            signal (full) ;
            until false;
        end
```



# 消费者

```
consumer : begin
    repeat
        wait (full);
        wait (mutex) ;
        nextc : =buffer(out);
        out: = (out+1) mod n;
        signal (mutex) ;
        signal (empty) ;
        consumer the item in nextc;
    until false;
end
parend
end
```

# 注意

1. 进程应该先申请资源信号量，再申请互斥信号量，顺序不能颠倒。
2. 对任何信号量的wait与signal操作必须配对。同一进程中的多对wait与signal语句只能嵌套，不能交叉。
3. 对同一个资源信号量的wait与signal可以不在同一个进程中。
  - 例如，wait(empty)在计算进程中，而signal(empty)则在打印进程中，计算进程若因执行wait(empty)而阻塞，则以后将由打印进程将它唤醒
4. wait与signal语句不能颠倒顺序，wait语句一定先于signal语句，否则可能引起进程死锁。

## 2. 利用AND信号量解决生产者—消费者问题

```
var mutex, empty, full:semaphore : =1, n, 0;
    buffer:array [0, ..., n-1] of item;
in out:integer : =0, 0;
begin
    parbegin
        producer:begin
            repeat
                ...
                produce an item in nextp;
                ...
                swait(empty, mutex);
                buffer(in) : =nextp;
                in : =(in+1)mod n;
                ssignal(mutex, full);
            until false;
        end
```

```
consumer:begin
  repeat
    Swait(full, mutex);
    nextc : =buffer(out);
    out : =(out+1) mod n;
    Ssignal(mutex, empty);
    consumer the item in nextc;
  until false;
end
parend
end
```

## 2.5.3 读者/写者问题

### 问题描述

- 该问题为多个进程访问一个共享数据区，如数据库、文件、内存区及一组寄存器等的数据库问题建立了一个通用模型
- 存在若干读进程只能读数据，若干写进程只能写数据。

# 问题描述

- 例如，一个联网售票系统，数据的查询和更新非常频繁，不可避免会出现多个进程试图查询或修改（读/写）其中某一条数据的情形。
- 多个进程同时读一条记录是可以的。
- 如果一个进程正在更新数据库中的某条记录，则所有其他进程都不能访问（读或写）该记录，否则可能会将同一个座位销售多次。

# 读者/写者进程满足的条件

- 1) 允许多个读者进程可以同时读数据;
- 2) 不允许多个写者进程同时写数据, 即只能互斥写数据;
- 3) 若有写者进程正在写数据, 则不允许读者进程读数据。



# 如何控制读者和写者

- 如果采用生产者/消费者问题解决方法，严格互斥任何读者和写者进程，可以保证数据更新操作的正确性。
  - 如果一个写者进程正在修改数据，别的写者以及任何读者都不能访问该数据。
  - 但是，对于若干读者进程，只不过希望查询售票情况，却被严格互斥，严重影响了系统效率。显然应该让多个读者同时读数据。

- 读者优先：一旦有读者正在读数据，允许多个读者同时进入读数据，*只有当全部读者退出，才允许写者进入写数据*。
- 现在假设一个写者到来，由于写操作是排他的，所以它不能访问数据，需要阻塞等待。如果一直都有新的读者陆续到来，写者的写操作将被严重推迟。
- 用信号量实现的具体过程见图

读者-写者问题可描述如下:

Var rmutex, wmutex:semaphore : =1,1; /\* 互斥信号量, 初始化为1 \*/

Readcount:integer : =0;; /\* 统计读者个数 \*/

begin

parbegin

Reader:begin

repeat

wait(rmutex);

if readcount=0 then wait(wmutex);

Readcount : =Readcount+1;

signal(rmutex);

...

perform read operation;

...

wait(rmutex);

readcount : =readcount-1;

if readcount=0 then signal(wmutex);

signal(rmutex);

until false;

end

```
writer:begin  
  repeat  
    wait(wmutex);  
    perform write operation;  
    signal(wmutex);  
  until false;  
end  
parend  
end
```

## 2. 利用信号量集机制解决读者-写者问题

```
var RN integer; /* 控制读者上限RN*/  
    L, mx: semaphore : =RN, 1;  
begin  
    parbegin  
        reader: begin  
            repeat  
                swait(L, 1, 1); /* 每进入一个读者, L减一, 控制总数RN个*/  
                swait(mx, 1, 0); /* 开关, mx=1标识无writer进程写*/  
                ...  
                perform read operation;  
                ...
```

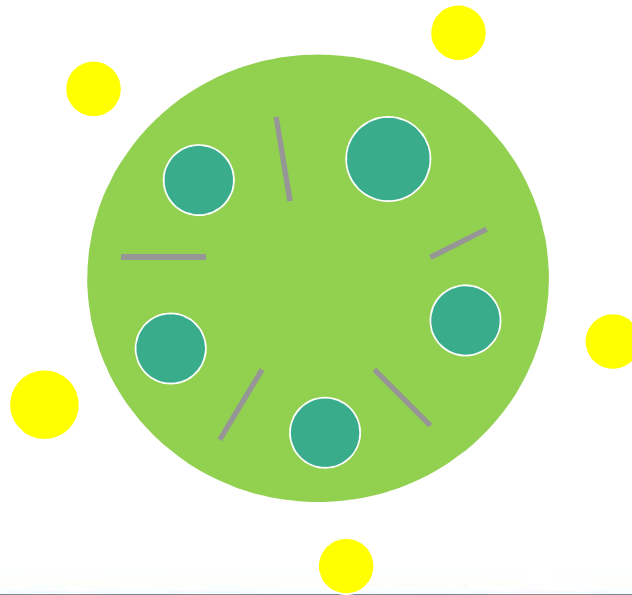
```
Ssignal(L,1);  
  until false;  
end  
writer:begin  
  repeat  
    Swait(mx,1,1; L,RN,0);  
    perform write operation;  
    Ssignal(mx,1);  
  until false;  
end  
parend
```

读写开关

互斥信号量

## 2.5.2 哲学家就餐问题

- 五个哲学家五只筷子，哲学家循环做着思考和吃饭的动作，吃饭程序是：先取左边筷子，再取右边筷子，再吃饭，再放筷子。





# 利用记录型信号量解决哲学家进餐问题

- 经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：
- Var chopstick: array [0, ..., 4] of semaphore;



实现：

- 为每个筷子设一把锁（信号量，初值为1）
- 每个哲学家是一个进程。

```
Var Chopstick;array [0,4]of semaphore;
```

第i个进程描述为( $i=0, \dots, 4$ )

```
repeat
```

```
wait (chopstick[i]) ;取左筷子;
```

```
wait(chopstick[(i+1)mod 5]); 取右筷子;
```

```
eat;
```

```
signal(chopstick[i]);放左筷子
```

```
signal (Chopstick[(i+1)mod 5]; 放右筷子;
```

```
think;
```

```
until false;
```

(这可能导致死锁)

## 2. 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，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。

```
Var chopstick array [0, ..., 4] of semaphore : =(1,1,1,1,1);  
processi  
    repeat  
        think;  
        Sswait(chopstick [(i+1) mod 5] , chopstick [i] );  
        eat;  
        Ssignal(chopstick [(i+1) mod 5] , chopstick [i] );  
    until false;
```

## 2.5.4 互斥与同步解决方法之四:管程

- 虽然信号量机制是一种既方便、又有效的进程同步机制，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，而且还会因同步操作的使用不当而导致系统死锁。这样，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。

- 当共享资源用共享数据结构表示时，资源管理程序可用对该数据结构进行操作的一组过程来表示(例如，资源的请求和释放过程request和release)，我们把这样一组相关的数据结构和过程一并称为管程。
- Hansan为管程所下的定义是：“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”。

- 管程由四部分组成：
  - ① 局部于管程的共享变量说明；
  - ② 对该数据结构进行操作的一组过程；
  - ③ 对局部于管程的数据设置初始值的语句。
  - ④ 管程的名称
- 前两个特点让人联想到面向对象软件中对象的特点。的确，面向对象操作系统或程序设计语言可以很容易地把管程作为一种具有特殊特征的对象来实现。
- Monitor（管程）——面向对象方法

# 管程的主要特点

- 局部数据变量只能被管程的过程访问，任何外部过程都不能访问。
- 一个进程通过调用管程的一个过程进入管程。
- 在任何时候，只能有一个进程在管程中执行，从而实现**进程互斥**。



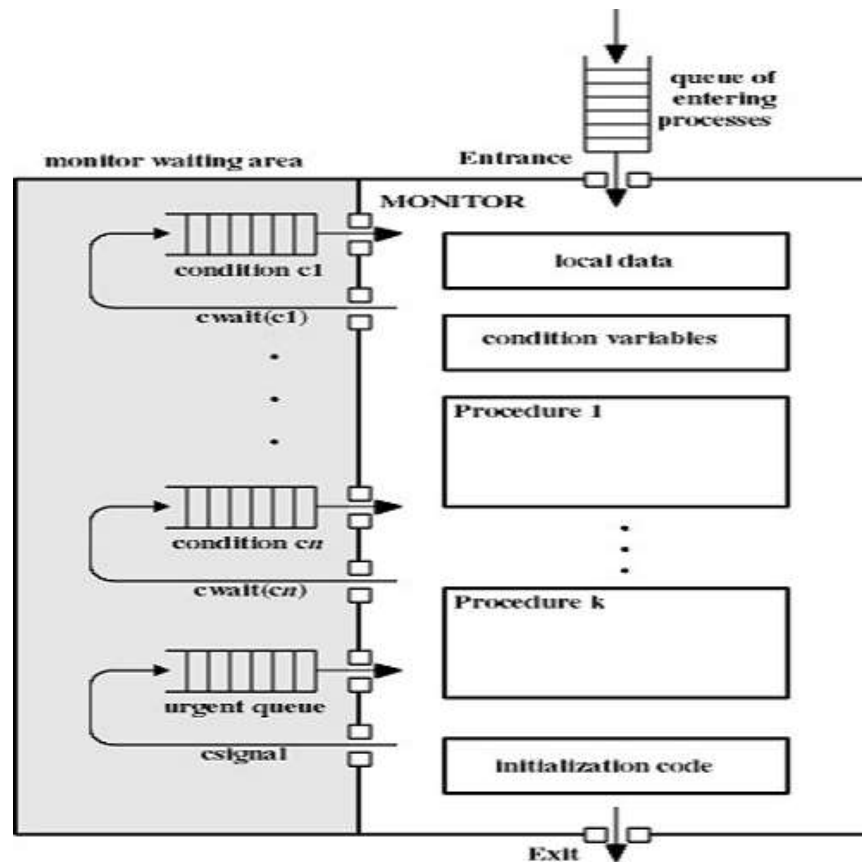


图2-54 管程结构示意图

### 3. 利用管程解决生产者-消费者问题

在利用管程方法来解决生产者-消费者问题时，首先便是为它们建立一个管程，并命名为Proclucer-Consumer, 或简称为PC。其中包括两个过程：

(1)put(item)过程。生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量count来表示在缓冲池中已有的产品数目，当 $\text{count} \geq n$ 时，表示缓冲池已满，生产者须等待。

(2) get(item)过程。消费者利用该过程从缓冲池中取出一个产品，当 $\text{count} \leq 0$ 时，表示缓冲池中已无可取用的产品，消费者应等待。



```
type producer-consumer=monitor
  Var in,out,count:integer;
  buffer:array [ 0,...,n-1 ] of item;
  notfull, notempty:condition;
  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.queene then notfull.signal;
end
begin in : =out : =0; count : =0 end
```

在利用管程解决生产者-消费者问题时，  
其中的生产者和消费者可描述为：

```
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
```

## 2.6 进程通信

- -通信协作
- 当进程进行通信合作时，各个进程之间需要建立连接，进程通信需要同步和协调。
- 进程通信的方式很多，包括消息传递、管道、共享存储区等。

# 进程通信的方式

- 进程之间的通信内容包含两种类型：控制信息与大批量数据。
- 低级通信：进程之间交换控制信息的过程
- 高级通信：进程之间交换批量数据的过程
- 进程之间同步与互斥是一种低级通信, 用来控制进程执行速度。
  - 效率低
  - 通信对用户不透明



# 常用的进程通信机制

- 基于共享存储区方式
- 管道通信
- 消息传递方式

# 1. 基于共享存储区方式

- 生产者/消费者问题：生产者与消费者通过共享缓冲区，实现数据传递。属于基于共享存储区通信。
- 共享数据区属于每个互相通信进程的组成部份。这种通信方式不要求数据移动，一般用于本地通信。
- 对于远程通信来说，每台计算机拥有各自的私有内存区，不易实现共享存储区的访问。



# 如何通过共享存储区通信？

1. 通过程序设计来实现。程序员设计程序时，利用程序指令设置共享数据结构，并处理通信进程之间的同步等问题，操作系统只需提供共享存储区。
2. 由操作系统在内存中划分出一块区域作为共享存储区。
  - 进程在通信前向系统申请共享存储区中的一个分区。
  - 然后，申请进程把获得的共享存储分区连接到本进程上，此后便可象读/写普通存储器一样地读/写共享存储分区。
  - 该方式下，通信进程之间的同步与互斥访问共享存储区可以由操作系统实现。

举例：

```
segment_id = shmget(IPC_PRIVATE, shared_segment_size,  
IPC_CREAT|IPC_EXCL|S_IRUSR|S_IWUSR );
```

```
shared_memory = (char*)shmat(segment_id, 0, 0); /* 绑定到共享内存块 */
```

```
shmctl(segment_id, IPC_STAT, &shmbuffer);
```

```
shmdt(shared_memory); /* 脱离该共享内存块 */
```

## 2. 管道(Pipe)通信

- 所谓“管道”，是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共享文件，又名pipe文件。
- 向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；
- 而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。
- 这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

- Linux命名管道非常适合同一机器上两个进程之间传递数据，其形式也是一个文件，但是读取与写入时遵循FIFO的原则。
- ```
#define FIFO_NAME "/tmp/my_fifo"  
#define BUFFER_SIZE PIPE_BUF  
mkfifo(FIFO_NAME,0777)
```

### 3. 消息传递方式

- ❑ 使用最广泛的一种进程间通信的机制。
- ❑ 进程间的数据交换，是以格式化的消息(message)为单位的（计算机网络称为报文）
- ❑ 程序员直接利用系统提供的一组通信命令(原语)进行通信。操作系统隐藏了通信的实现细节，大大减化了通信程序编制的复杂性。
- ❑ 消息通信实现方法：
  - 1.直接通信
    - Send( ), receive( )
  - 2.间接通信
    - 1) 信箱
    - 2) 消息缓冲队列通信



## 2.5.2 消息传递通信的实现方法

- 1. 直接通信方式
- 这是指发送进程利用OS所提供的发送命令，直接把消息发送给目标进程。
- 通常，系统提供下述两条通信命令(原语):
  - Send(Receiver, message); 发送一个消息给接收进程;
  - Receive(Sender, message); 接收Sender发来的消息;
- 例如，原语Send(P2, m1)表示将消息m1发送给接收进程P2; 而原语Receive(P1, m1)则表示接收由P1发来的消息m1。

## 利用直接通信原语，来解决生产者-消费者问题：

当生产者生产出一个产品(消息)后，便用Send原语将消息发送给消费者进程；而消费者进程则利用Receive原语来得到一个消息。如果消息尚未生产出来，消费者必须等待，直至生产者进程将消息发送过来。生产者-消费者的通信过程可分别描述如下：

```
repeat
    ...
    produce an item in nextp;
    ...
    send(consumer, nextp);
until false;

repeat
    receive(producer, nextc);
    ...
    consume the item in nextc;
until false;
```

## 2. 间接通信方式

进程之间需要通过共享数据结构的实体进行通信，通常把这种中间实体称为**信箱**


(1) **信箱的创建和撤消**。进程可利用信箱创建原语来建立一个新信箱。

创建者进程应给出信箱名字、信箱属性(公用、私用或共享)；对于共享信箱，还应给出共享者的名字。当进程不再需要读信箱时，可用信箱撤消原语将之撤消。

(2) **消息的发送和接收**。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信。

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；



信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类：

### 1) 私用信箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。当拥有该信箱的进程结束时，信箱也随之消失。




## 2) 公用信箱

它由操作系统创建，并提供给系统中的所有核准进程使用。进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。

## 3) 共享信箱

它由某进程创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息。




在利用信箱通信时，在发送进程和接收进程之间，存在以下四种关系：

(1) 一对一关系。这时可为发送进程和接收进程建立一条两者专用的通信链路，使两者之间的交互不受其他进程的干扰。

(2) 多对一关系。允许*提供服务的进程与多个用户进程之间进行交互*，也称为客户/服务器交互(client/server interaction)。

(3) 一对多关系。允许一个*发送进程与多个接收进程进行交互*，使发送进程可用广播方式，向接收者(多个)发送消息。

(4) 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中投递消息；也可从信箱中取走属于自己的消息。





## 2.5.4 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。在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区。它可描述如下：

```
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) PCB中有关通信的数据项。

在PCB中增加用于对消息队列进行操作和实现同步的信号量。在PCB中应增加的数据项可描述如下：

```
type processcontrol block=record
```

```
...
```

```
    mq; 消息队列队首指针
```

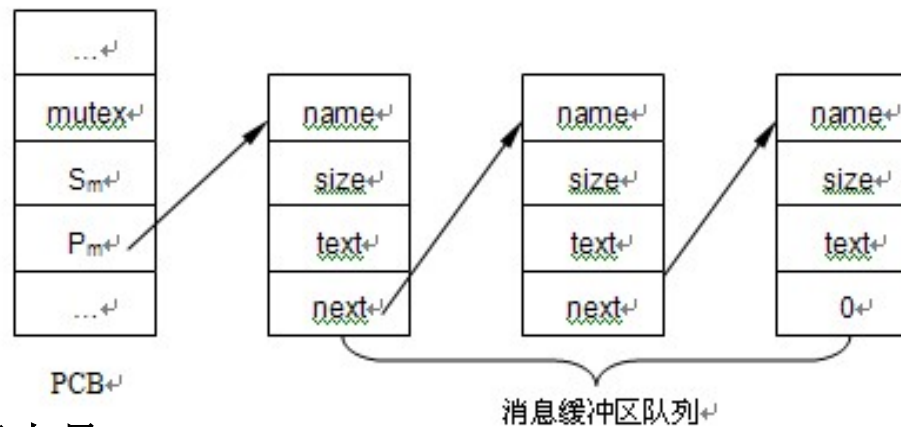
```
    mutex; 消息队列互斥信号量
```

```
    sm; 消息队列资源信号量
```

```
...
```

```
end
```

## ▲ 接收消息进程的PCB和它的消息缓冲链的关系



name: 发送消息的进程名或标志号

size: 消息长度

text: 消息正文

next: 下个缓冲区的地址

**mutex:** 消息队列操作互斥信号灯

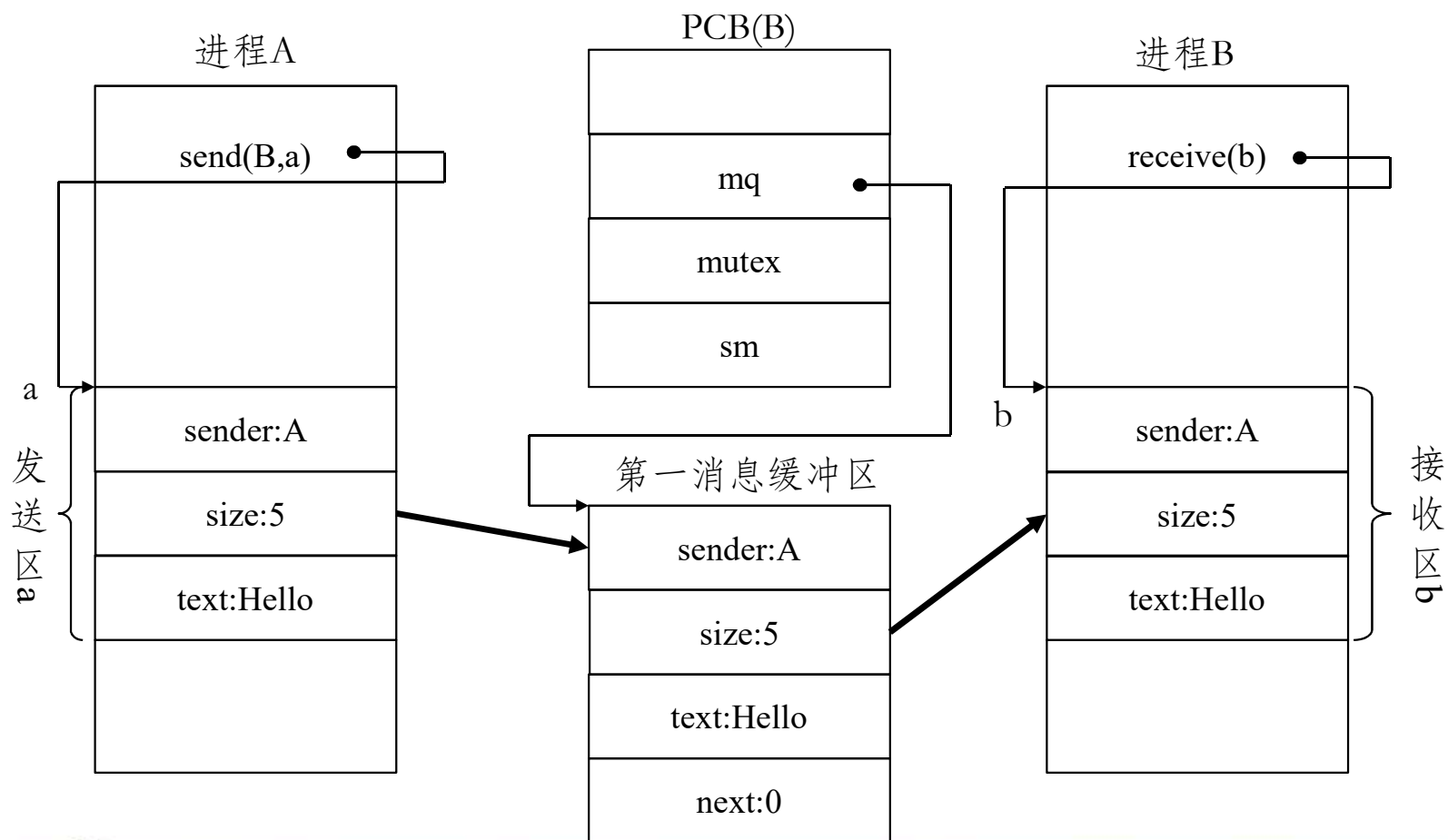
**S<sub>m</sub>:** 表示接收消息计数和同步的信号灯

**P<sub>m</sub>:** 指向该进程的消息队列中第一个缓冲区的指针

## 2. 发送原语

- 发送进程在利用发送原语发送消息之前，应先在自己的内存空间，设置一发送区a，把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。
- 发送原语首先根据发送区a中所设置的消息长度a.size来申请一缓冲区i，接着，把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的标识符j，然后将i挂在j.mq上。
- 由于该队列属于临界资源，故在执行insert操作的前后，都要执行wait和signal操作。

# 消息缓冲通信



```
procedure send(receiver, a)
begin
  getbuf(a.size,i);           根据a.size申请缓冲区;
  i.sender := a.sender;
  将发送区a中的信息复制到消息缓冲区之中;
  i.size := a.size;
  i.text := a.text;
  i.next := 0;
  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);
end
```

### 3. 接收原语

```
procedure receive(b)
begin
  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;
  signal(j.mutex);
  b.sender : =i.sender;
  将消息缓冲区i中的信息复制到接收区b;
  b.size : =i.size;
  b.text : =i.text;
end
```



## 2.5.3 消息传递系统实现中的若干问题

### A. 通信链路(communication link)

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路。有两种方式建立通信链路。

➤ **第一种方式**：由发送进程在通信之前，用显式的“建立连接”命令(原语)请求系统为之建立一条通信链路；在链路使用完后，也用显式方式拆除链路。这种方式**主要用于计算机网络**中。

➤ **第二种方式**：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命令(原语)，系统会自动地为之建立一条链路。这种方式**主要用于单机系统**中。





- 根据通信链路的**连接方法**，又可把通信链路分为两类：
  - ① 点—点连接通信链路，这时的一条链路只连接两个结点(进程)；
  - ② 多点连接链路，指用一条链路连接多个( $n > 2$ )结点(进程)。
- 而根据**通信方式**的不同，则又可把链路分成两种：
  - ① 单向通信链路，只允许发送进程向接收进程发送消息；
  - ② 双向链路，既允许由进程A向进程B发送消息，也允许进程B同时向进程A发送消息。

## B. 消息的格式

### 定长消息：

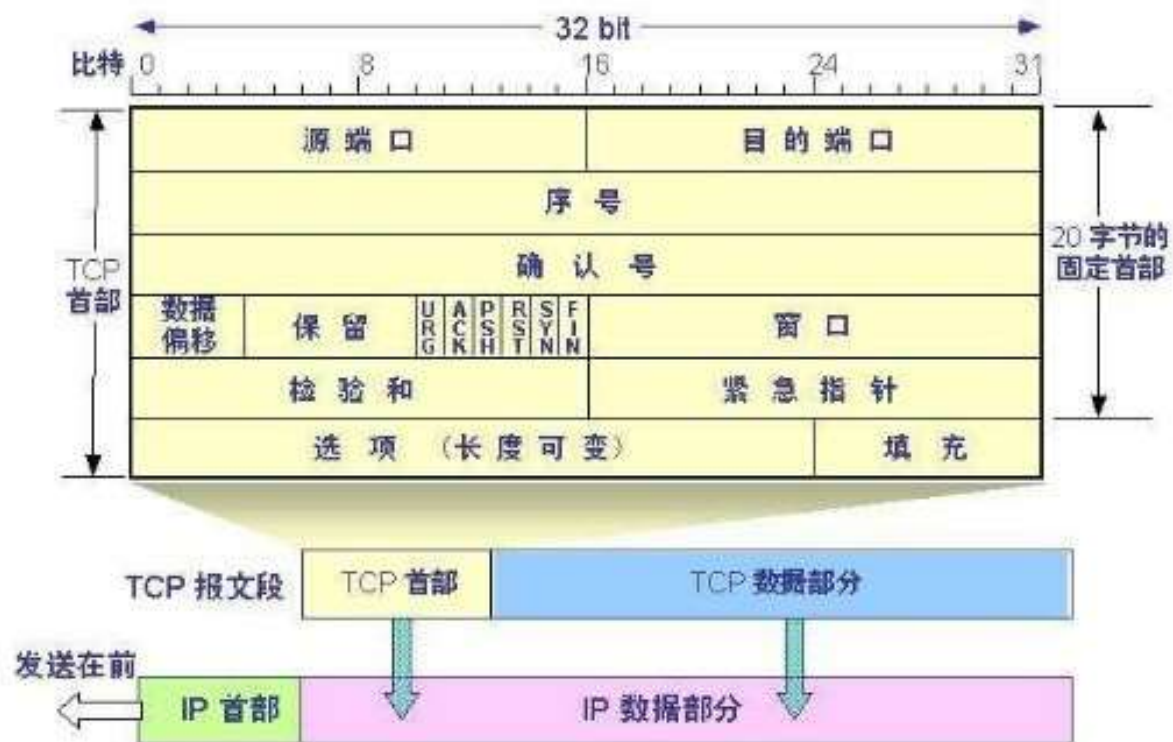
在某些OS中，消息是采用比较短的定长消息格式，这减少了对消息的处理和存储开销。

### 变长消息：

在有的OS中，采用另一种变长的消息格式，即进程所发送消息的长度是可变的。系统在处理 and 存储变长消息时，须付出更多的开销，但方便了用户。

注意：这两种消息格式各有其优缺点，故在很多系统(包括计算机网络)中，是同时都用的。

# TCP报文格式



## C. 进程同步方式

- (1) 发送进程阻塞、 接收进程阻塞。（汇合）
- (2) 发送进程不阻塞、 接收进程阻塞。
- (3) 发送进程和接收进程均不阻塞。

# 补充1：有关进程操作的命令

## 1. ps命令

查看进程状态

ps命令的一般格式是： ps [选项]

\$ ps

| PID  | TTY   | TIME     | CMD  |
|------|-------|----------|------|
| 1788 | pts/1 | 00:00:00 | bash |
| 1822 | pts/1 | 00:00:00 | ps   |

## 2. kill命令

终止一个进程的运行

kill命令的一般格式是：

- kill [-s 信号|-p] [-a] 进程号...
- kill -l [信号]

\$ kill 1651

### 3. sleep命令

使进程暂停执行一段时间。

一般使用格式是：**sleep** 时间值

```
$ sleep 100; who | grep 'mengqc'
```



## 补充2： 有关进程管理的系统调用

### ■ 系统调用的使用方式

在UNIX/Linux系统中，系统调用和库函数都是以C函数的形式提供给用户的，它有类型、名称、参数，并且要标明相应的文件包含。

`open`系统调用可以打开一个指定文件，其函数原型说明如下：

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int open(const char *path, int oflags);
```

例如：

```
int fd;
```

```
...
```

```
fd=open("/home/mengqc/myfile1",O_RDWR);
```

```
...
```



## ■ 有关系统调用的格式和功能

常用的有关进程管理的系统调用有：fork, exec, wait, exit, getpid, sleep, nice等

## ■ 应用示例

```
/*proc1.c演示有关进程操作*/
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
int main(int argc,char **argv)
{
    pid_t pid,old_ppid,new_ppid;
    pid_t child,parent;
    parent=getpid();          /*获得本进程的PID*/
    if((child=fork())<0){
        fprintf(stderr,"%s:fork of child failed:%s\n",argv[0],strerror(errno));
        exit(1);
    }
    else if(child==0){        /*此时是子进程被调度运行*/
        old_ppid=getppid();
        sleep(2);
        new_ppid=getppid();
    }
    else {
        sleep(1);
        exit(0);              /*父进程退出*/
    }
    /*下面仅子进程运行*/
    printf("Original parent:%d\n",parent);
    printf("Child:%d\n",getpid());
    printf("Child's old ppid:%d\n",old_ppid);
    printf("Child's new ppid:%d\n",new_ppid);
    exit(0);
}
```

程序运行的结果如下：

```
$ ./proc1
Original parent:2009
Child:2010
Child's old ppid:2009
Child's new ppid:1
```

## 2.7 线程

- 20世纪80年代中期，人们又提出了比进程更小的能独立运行的基本单位——线程（Threads）；以提高程序并发执行的程度，进一步提高系统的吞吐量。
- 线程能比进程更好地提高程序的并行执行程度，充分地发挥多处理机的优越性，在多处理机OS中也都引入了线程，以改善OS的性能。

## 2.7.1 线程的引入

- 进程的概念体现出两个特点：
  - ①**资源所有权**：一个进程包括一个保存进程映像的虚地址空间，并且随时分配对资源的控制或所有权，包括内存、I/O通道、I/O设备、文件等。
  - ②**调度 / 执行**：进程是被操作系统调度的实体。
- 为区分这两个特点，调度并分派的部分通常称为**线程或轻便进程**（lightweight process），而**资源所有权**的部分通常称为进程。

## 2.7.1 线程的引入

- 进程并发执行的时空开销

1. 创建进程，必须为其分配必需的、除处理机以外的所有资源，如内存、I/O设备，及PCB等
2. 撤销进程，必须对其占有的资源执行回收操作
3. 进程切换，需要保留当前进程的CPU环境，设置新选进程的CPU环境，替换页表等

# 线程的属性

- (1) 线程是一个被调度和分派的基本单位，并可独立运行的实体。大多数与执行相关的信息可以保存在线程级的数据结构中；
- (2) 线程是可以并发执行；
- (3) 共享进程资源。
- (4) 线程是轻型实体，在切换时只需保存少量寄存器的内容，不涉及存储器的管理等，因此系统开销小。
- (5) 进程中的所有线程共享同一个地址空间，挂起进程则会挂起进程中的所有线程。类似地，进程的终止会导致进程中所有线程的终止。

# 采用线程的优点

1. 在一个已有进程中创建一个新线程比创建一个全新进程所需的时间少。
2. 终止一个线程比终止一个进程花费的时间少。
3. 线程间切换比进程间切换花费的时间少。
4. 线程提高了不同的执行程序间通信的效率。同一个进程中的线程共享存储空间和文件，它们无需调用内核就可以互相通信。

## 2.7.2 进程和线程的比较

- 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



## 2.7.3 线程状态和线程控制块

(1) 线程的状态有：运行状态、就绪状态、阻塞状态等。

(2) 在线程切换时保存的线程信息：

①一个执行栈。

②每个线程静态存储局部变量。

③对存储器和其进程资源的访问，并与该进程中的其他线程共享这些资源。

如：寄存器状态、堆栈、线程运行状态、 优先级、线程专有存储器、信号屏蔽等

# 线程状态变化的4种基本操作

- ①**派生 (Spawn)**: 当产生一个新进程时, 同时也为该进程派生了一个“初始化线程”, 随后, 可以在同一个进程中派生另一个线程, 新线程被放置在就绪队列中。
- ②**阻塞 (Block)**: 当线程需要等待一个事件时, 它将阻塞, 此时处理器转而执行另一个就绪线程。
- ③**解除阻塞 (Unblock)**: 当阻塞一个线程的事件发生时, 该线程被转移到就绪队列中。
- ④**结束 (Finish)**: 当一个线程完成时, 其寄存器的信息和栈都被释放。



## 注意（一）

- 1、在多线程环境中，仍然有一个与进程相关联的进程控制块和用户地址空间，但是每个线程都有一个独立的栈和独立的线程控制块TCB，包含寄存器值、优先级和其他与线程相关的状态信息。
- 2、进程中的所有线程共享该进程的状态和资源，它们驻留在同一块地址空间中，并且可以访问到相同的数据。

## 注意（二）

3、线程阻塞不一定引起进程阻塞

— （系统调用时进入核心态时，若线程阻塞则其所在进程也会被阻塞。）。

4. 在同一进程中的线程切换不会引起进程切换。

5. 在不同一进程中的线程切换会引起进程切换。

## 2.6.2 线程间的同步和通信

### 1. 互斥锁(mutex)

互斥锁是一种比较简单的、用于实现线程间对资源互斥访问的机制。


- 较适合于高频度使用的关键共享数据和程序段。
- 互斥锁可以有两种状态，即开锁(unlock)和关锁(lock)状态。
- 相应地，可用两条命令(函数)对互斥锁进行操作，关上或打开mutex锁。



## 2. 条件变量

- 每一个条件变量通常都与一个互斥锁一起使用，亦即，在创建一个互斥锁时便联系着一个条件变量。
- 单纯的互斥锁用于短期锁定，主要是用来保证对临界区的互斥进入。
- 而条件变量则用于线程的长期等待，直至所等待的资源成为可用的。





下面给出了对上述资源的申请(左半部分)和释放(右半部分)操作的描述。

Lock mutex

check data structures;

while(resource busy);

wait(condition variable);

mark resource as busy;

unlock mutex;

Lock mutex

mark resource as free;

unlock mutex;

wakeup(condition variable);





### 3. 信号量机制

#### 1. 私用信号量(private semaphore)。

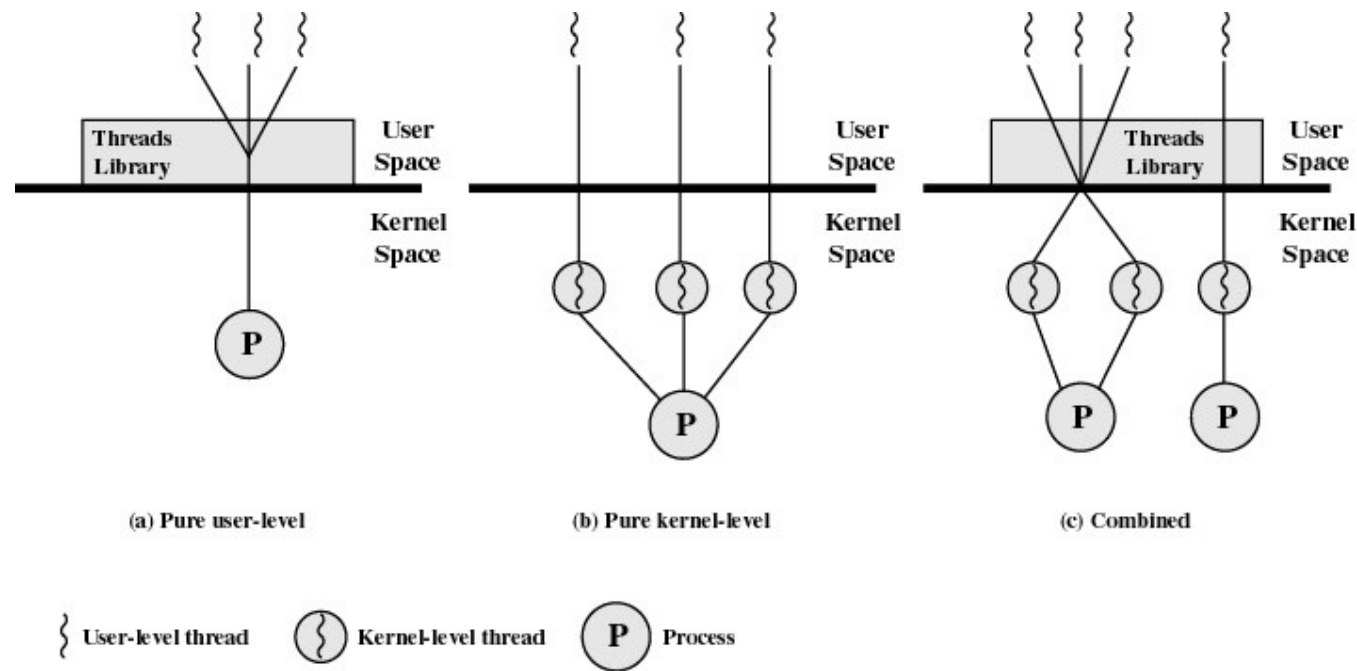
- 线程调用创建信号量的命令来创建，其数据结构是存放在应用程序的地址空间中。
- 私用信号量属于特定的进程所有，OS并不知道私用信号量的存在。

#### 2. 公用信号量(public semaphore)。

- 实现不同进程间或不同进程中各线程之间的同步而设，有公开的名字供所有的进程使用
- 其数据结构是存放在受保护的系统存储区中，由OS为它分配空间并进行管理，故也称为系统信号量。
- 如果信号量的占有者在结束时未释放该公用信号量，则OS会自动将该信号量空间回收。

## 2.6.3 线程的分类

- 线程分为：
  - 1.内核级线程
  - 2.用户级线程。



**Figure 4.6 User-Level and Kernel-Level Threads**

# 1. 内核支持线程

- 所谓的内核级线程，是在内核的支持下运行的，即无论是用户进程中的线程，还是系统进程中的线程，他们的创建、撤消和切换等，也是依靠内核实现的。
- 在内核空间为每一个内核线程设置了一个线程控制块，内核是根据该控制块而感知某线程的存在的，并对其加以控制。

- **KST方法的主要优点：**

1. 内核可以同时把同一个进程中的多个线程调度到多个处理器中并行执行；
2. 如果进程中的一个线程被阻塞，内核可以调度同一个进程中的另一个线程，也可以运行其它进程的线程。
3. 内核例程自身也是可以使用多线程的。
4. 结构简单，高效

- **KST方法的主要缺点**是在同一个进程中把控制从一个线程传送到另一个线程，需要从用户态转到内核态进行，线程调度和管理是在内核实现的，系统开销较大。

## 2. 用户级线程

- 用户级线程的创建、撤消、线程之间的同步与通信等功能，都无须利用系统调用来实现，用户级线程的切换也无须内核的支持。
- 优点：
  - 线程控制块设置在用户空间，内核完全不知道用户级线程的存在，这样可以节省模式切换系统开销。
  - 各进程可以独立选择线程调度算法
  - 用户级线程与操作系统平台无关，甚至可以在不支持线程机制的操作系统平台上实现。



# 用户级线程

- 缺点
  - 当线程执行系统调用引起进程阻塞时，进程中所有的线程都会被阻塞，会削弱进程中的线程的并发性。而内核支持线程不存在这个问题。
  - 由于内核每次给一个进程分配一个CPU，用户级线程不能有效利用多处理机进行进程内的多线程并行操作。



## 2.6.4 线程的实现

### 1. 内核支持线程的实现

- 系统为每个进程创建任务数据区PTDA

PTDA 进程资源

TCB # 1

TCB # 2

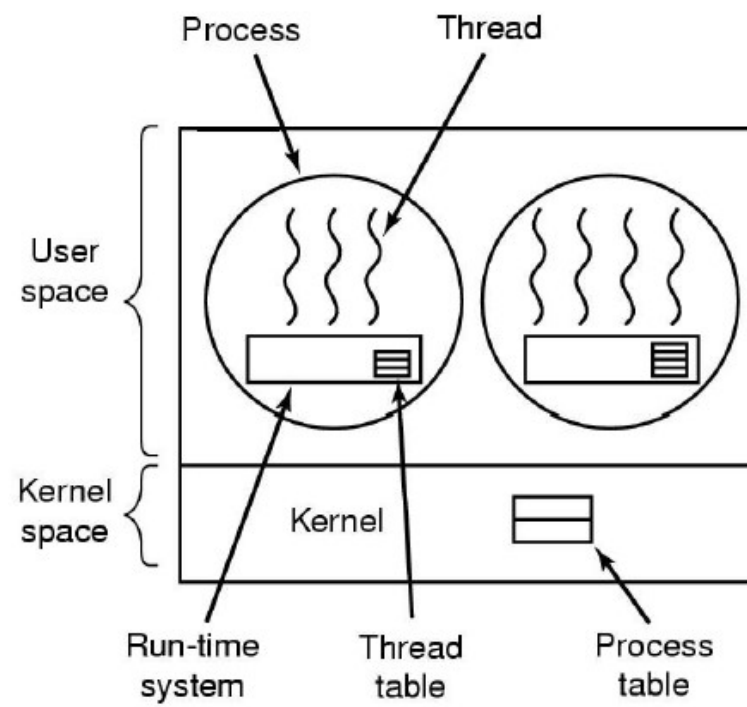
TCB # 3

图 2 - 13 任务数据区空间

## 2. 用户级线程的实现

### 1) 运行时系统(Runtime System)

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



## 2) 内核控制线程

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

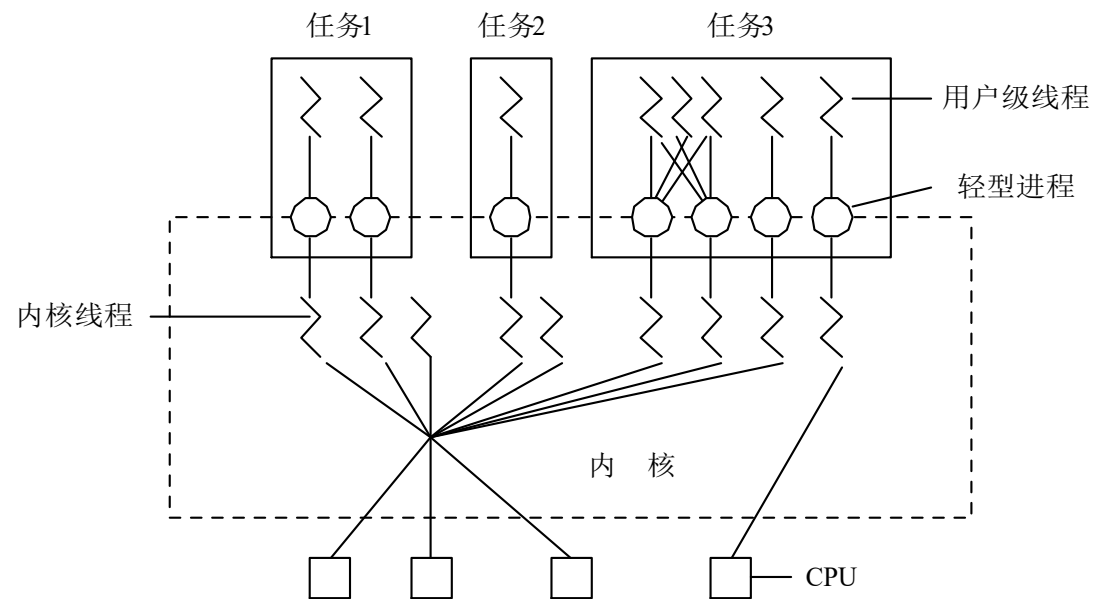


图 2 - 14 利用轻型进程作为中间系统

# 总结

- 为什么引入进程？
- 前趋图
- 进程的概念、结构、状态及其转换
- 进程的控制，控制什么？（执行模式、操作系统内核、原语）
- 进程状态转换操作如何实现？

# 进程并发控制

- 进程同步
- 进程互斥：临界资源、临界区
- 进程通信



# 进程互斥与同步

- 软件方法
- 硬件方法
- 信号量方法：信号量定义、类型、原语、应用
- 管程方法
- 消息通信方法

# 经典进程互斥与同步问题

- 生产者/消费者问题
- 读者/写者问题
- 哲学家进餐问题

# 线程

- 线程与进程的关系
- 线程状态转换
- 线程间的同步和通信
- 线程的实现方式（KST、ULT）

## 第二章作业

1. 试画出下面四条语句的前驱图：

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

2. 说明进程在基本状态之间转换的典型原因

3. 在进行进程切

4. 换时，所要保存的处理机状态信息有哪些

5. 在创建一个进程是所要完成的主要工作是什么

6. 同步机构应遵循哪些基本准则，为什么？

7. 在生产者-消费者问题中，如果将两个wait操作缓缓，或者将两个signal互换，结果会如何？

8. 修改第26题中的错误

9. 尝试解释引入线程的意义