



第2章 进程管理

2.1 进程 (Process)

2.2 进程控制

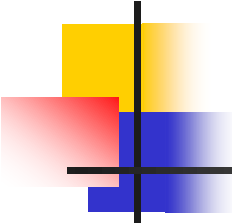
2.3 进程同步

2.4 经典进程同步问题

2.5 进程间通信

2.6 线程 (Thread)

2.7 小结



2.1 进 程 (Process)

一、什么是进程？

是程序的1次执行。

或者说，进程是程序执行的1个实例。

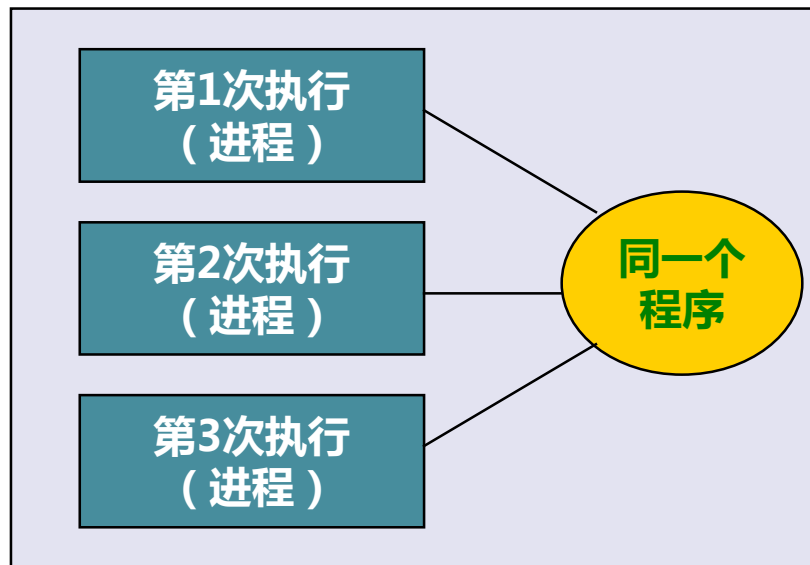
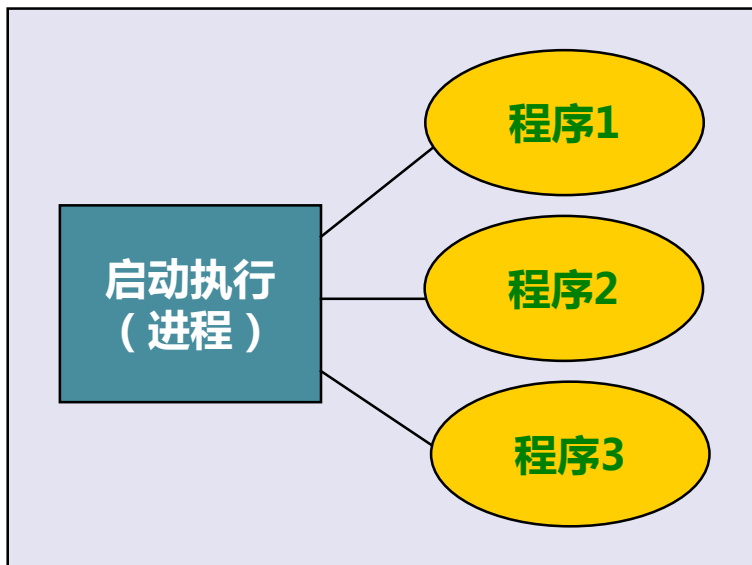
每个进程都有自己的地址空间。

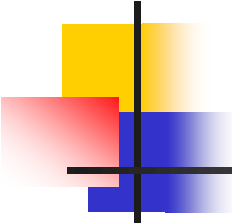
为什么要引入进程？直接用程序的概念不行吗？

2.1 进程(Process)

二、进程和程序的关系与差异

- ✓ 程序是进程的静态实体（即执行代码）。
- ✓ 程序是静态的，进程是动态的。
- ✓ 同一个程序可以对应多个进程，每启动1次产生1个进程。





2.1 进 程 (Process)

三、进程的状态

1. 进程的5种基本状态

- (1) 新建 (new) : 进程正在被创建。
- (2) 就绪 (ready) : 进程可运行 , 正等待获得处理机。
- (3) 运行 (running) : 进程的指令正在被执行。
- (4) 阻塞 (blocked) 或等待 : 进程因等待某事件 (如请求I/O) 而暂停执行。
- (5) 完成 (done) : 进程结束。



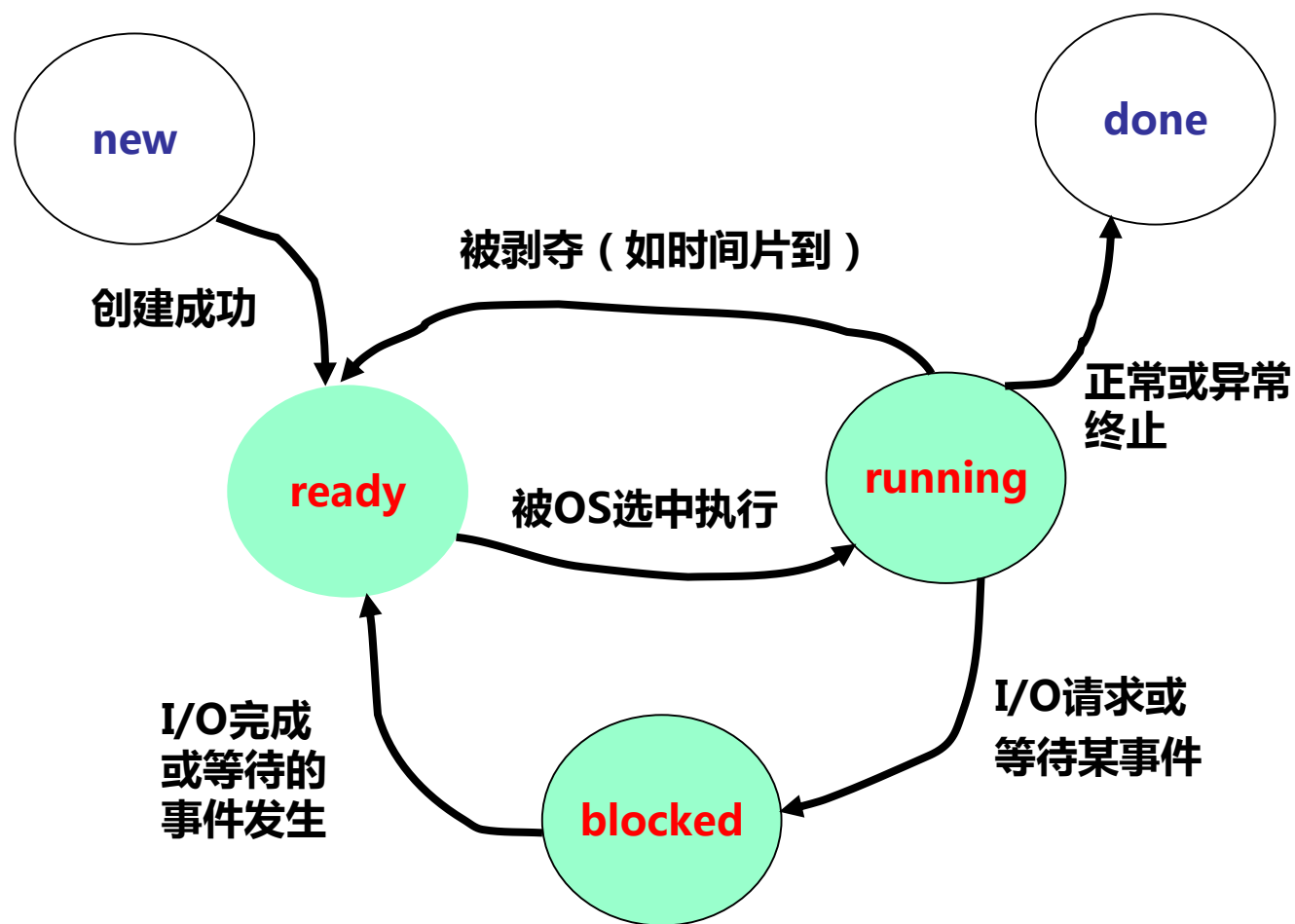
进程的状态

引入new和done状态的原因：

- ✓ 由于OS在建立一个新进程时，通常分为2步：第一步是创建进程，并为其分配资源，此时进程即处于new状态。第二步是把新创建的进程送入就绪队列，状态变为就绪状态。
- ✓ 一个结束了的进程，其退出系统的过程也分为两步：第一步是使该进程成为一个不可能再运行的进程，处于done状态。此时系统并不立即撤销它，而是将它暂时留在系统中，以便其它进程去收集该进程的有关信息，也有利于提高系统性能。

进程的状态

2. 状态之间的转换





进程的状态

3. 七状态进程模型

在有的系统中，为了暂时缓和内存的紧张状态，或为了调节系统负荷，又引入了**挂起 (suspend)** 功能：

暂时挂起一部分进程，把它们**从内存临时换出到外存**。

就绪状态分为2种：**活动就绪状态**（未被挂起的就绪进程）和**静止就绪状态**（被挂起的就绪进程）

阻塞状态分为2种：**活动阻塞状态**（未被挂起的阻塞进程）和**静止阻塞状态**（被挂起的阻塞进程）



七状态进程模型

- ✓ 就绪(Ready) : 进程在内存且可立即进入运行状态
- ✓ 阻塞(Blocked) : 进程在内存并等待某事件的出现
- ✓ 阻塞挂起(Blocked, suspend) : 进程在外存并等待某事件的出现
- ✓ 就绪挂起(Ready, suspend) : 进程在外存, 但只要进入内存, 即可运行
- ✓ 运行
- ✓ 新建
- ✓ 完成



七状态进程模型

- **挂起 (Suspend)** : 把一个进程从内存转到外存。

可能有以下几种情况：

- ✓ **阻塞→阻塞挂起**：没有进程处于就绪状态或就绪进程要求更多内存资源时，可能发生这种转换，以提交新进程或运行就绪进程
- ✓ **就绪→就绪挂起**：当有**高优先级阻塞**（系统认为会很快就绪的）进程和低优先级就绪进程时，系统可能会选择挂起低优先级就绪进程
- ✓ **运行→就绪挂起**：对**抢占式**系统，当有**高优先级**阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态



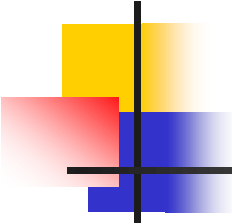
七状态进程模型

- **激活 (Activate)** : 把一个进程从外存转到内存。

可能有以下几种情况：

- ✓ **就绪挂起→就绪**：没有就绪进程或挂起就绪进程优先级高于就绪进程时，可能发生这种转换
- ✓ **阻塞挂起→阻塞**：当一个进程释放足够内存时，系统可能会把一个高优先级阻塞挂起（系统认为会很快出现所等待的事件）进程从外存转到内存





2.1 进 程 (Process)

四、进程的描述

进程控制块 (Process Control Block, PCB)

1. PCB是什么？

- ✓ 是OS管理和控制进程的数据结构。
- ✓ PCB记录着进程的描述信息。
- ✓ 每个进程对应1个PCB。



进程的描述

2. PCB的作用

- ✓ PCB是进程的一部分

进程由3部分组成：程序、数据、PCB。

- ✓ PCB伴随着进程的整个生命周期。

进程创建时，由OS创建PCB；

进程终止时，由OS撤销PCB；

进程运行时，以PCB作为调度依据。



进程的描述

3. PCB中的主要信息

(1) 进程本身的标识信息

- ✓ 进程标识符pid(process ID)：整数，由OS分配，唯一
- ✓ 用户标识符uid(user ID)：创建该进程的用户
- ✓ 对应程序的地址：内存、外存

(2) CPU现场 - 为进程正确切换所需

- ✓ 所有寄存器的值
或称进程上下文(context)



PCB中的主要信息

(3) 进程调度信息

- ✓ 进程的状态
- ✓ 优先级
- ✓ 使进程阻塞的条件
- ✓ 占用CPU、等待CPU的时间 (用于动态调整优先级)

(4) 进程占用资源的信息

- ✓ 进程间同步和通信机制，如信号量、消息队列指针
- ✓ 打开文件的信息，如文件描述符表



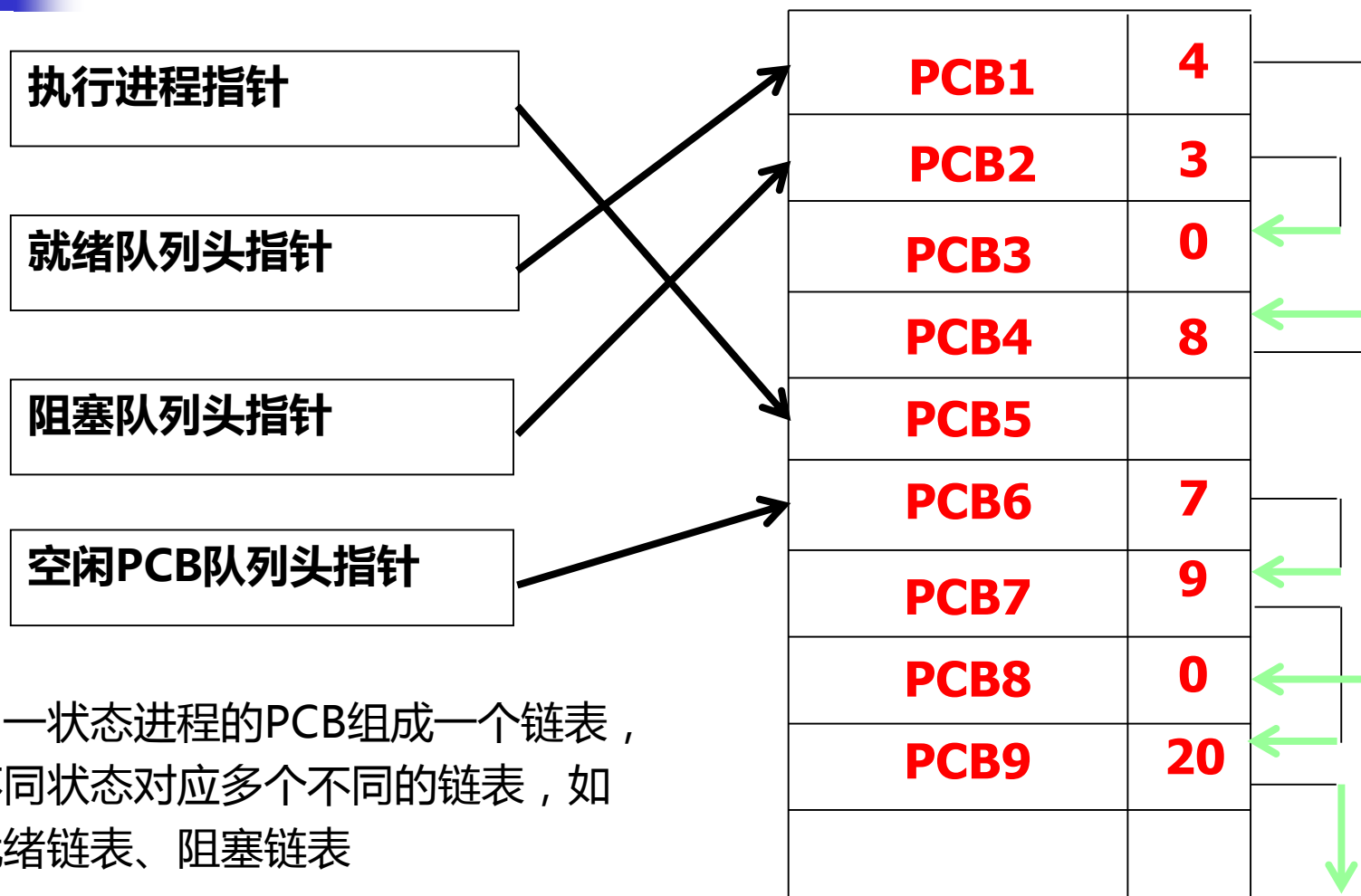
进程的描述

4. PCB的组织方式

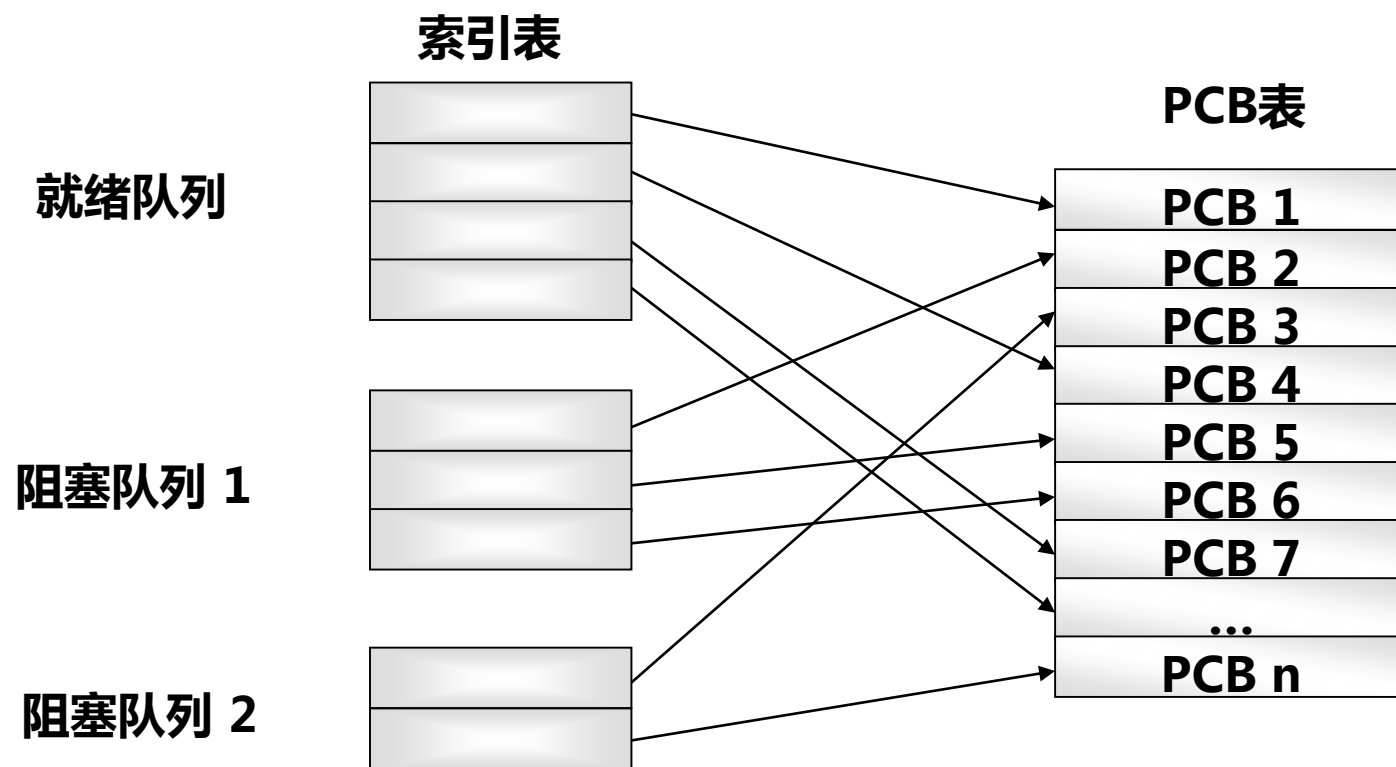
一般来说，系统把所有PCB组织在一起，并把它们放在内存的固定区域，构成**PCB表**。

PCB表的大小决定了系统中最多可同时存在的进程个数。

PCB的组织方式

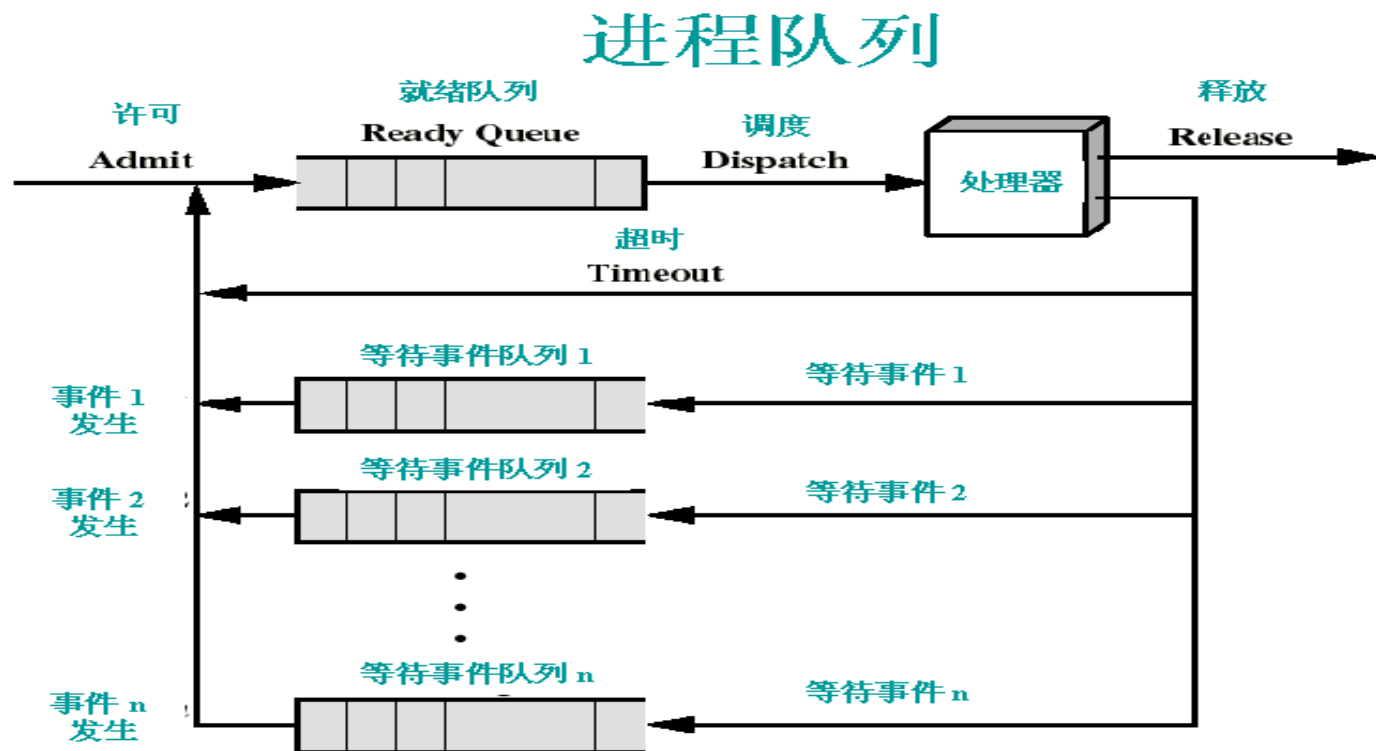


PCB的组织方式



对具有相同状态的进程，分别设置各自的索引表，表明其PCB在PCB表中的地址

PCB的组织方式



- 就绪队列无优先级 (例: **FI FO**)
- 当事件n发生, 对应队列移进就绪队列



2.2 进 程 控 制

创建、撤销进程以及完成进程各状态之间的转换，由具有特定功能的原语完成

- ✓ 进程创建原语
- ✓ 进程撤销原语
- ✓ 阻塞原语
- ✓ 唤醒原语
- ✓ 挂起原语
- ✓ 激活（解挂）原语
- ✓ 改变进程优先级



进程创建

- ✓ 创建一个PCB
- ✓ 赋予一个唯一的进程标识符pid
- ✓ 初始化PCB
- ✓ 设置相应的链接

如: 把新进程加到就绪队列中



进 程 撤 销

- ✓ 收回进程所占有的资源
- ✓ 撤销该进程的PCB



进程阻塞

运行->阻塞：

处于运行状态的进程，在其运行过程中期待某一事件发生，如等待键盘输入、等待磁盘数据传输完成、等待其它进程发送消息，当被等待的事件未发生时，**由进程自己**执行阻塞原语，使自己由运行态变为阻塞态。

- ✓ 保护CPU现场到PCB
- ✓ PCB插入到阻塞队列



进程唤醒

阻塞->就绪：

处于阻塞状态的进程，当等待的事件发生时，执行唤醒原语，使之由阻塞态变为就绪态。

✓ PCB插入到就绪队列



2.3 进 程 同 步

一、多个进程的并发执行

在执行时间上互相重叠（或交替），一个进程的执行尚未结束，另一个进程的执行已经开始的执行方式。



多个进程的并发执行

【例】一种文件打印的实现方案。

- ✓ 当一个进程需要打印文件时，将文件名放入一个特殊的目录spooler（即等待队列）下。
- ✓ 由一个后台进程负责打印：周期性地检查spooler，看是否有文件需要打印。如果有，则打印之，并将其从spooler目录中删除。



一种文件打印的实现方案

具体的实现方法：

设spooler目录有许多（潜在的无限多个）槽，编号为0，1，2，...。

每个槽放1个文件名。

设置2个共享变量（比如，保存在一个所有进程都能访问的文件中）：

out：指向下一个要打印的文件

in：指向下一个空闲的槽

进程要打印文件时的处理方式：

读in \rightarrow free_slot;

写文件名 \rightarrow spooler[free_slot];

free_slot + 1 \rightarrow in;

能正确实现文件打印吗？



一种文件打印的实现方案

出现的问题：

结果的不确定性。即结果取决于进程运行的时序。

原因：由资源共享引起。

为此，引入同步（synchronization）和互斥（mutual exclusion）。

同步：进程之间需要一种严格的时序关系。

如输入、计算、输出进程之间。

互斥：不能同时访问共享资源。可以看作是一种特殊的同步。

实现互斥是OS设计的基本内容。



2.3 进程同步

二、临界资源 (Critical Resource) 与临界区 (Critical Section)

临界资源：必须互斥访问的共享资源

临界区：进程中访问临界资源的那段程序

实现互斥的关键：

两个（多个）进程不同时处于临界区。



2.3 进程同步

三、实现互斥的方案

一个好的互斥方案应满足以下条件：

- (1) 任何两个进程不能同时处于临界区。
- (2) 临界区外的进程不应阻止其他进程进入临界区。
- (3) 不应使进程在临界区外无休止地等待。就是说，临界区代码执行时间要短。
- (4) 不应针对CPU的个数和进程之间的相对运行速度作任何假设。



实现互斥的方案

1. 设置锁变量lock

设置共享变量lock = $\begin{cases} 0: \text{临界区内无进程, 初始值} \\ 1: \text{临界区内有进程} \end{cases}$

```
while (lock)
    ;
lock = 1;
<Critical Section>
lock = 0;
<NonCritical Section >
```

该方案是错误的!



实现互斥的方案

2. 严格轮转法

设置共享变量turn，以指示进入临界区的进程号

以2个进程为例。turn = { 0: 允许进程0进入临界区，初始值
1: 允许进程1进入临界区

```
进程0 :  
while (turn != 0)  
    ;  
<Critical Section>  
turn = 1;  
<NonCritical Section >
```

```
进程1 :  
while (turn != 1)  
    ;  
<Critical Section>  
turn = 0;  
<NonCritical Section >
```

不是一个可取的方案!



实现互斥的方案

3. Peterson解决方案

```
enter_region(process); //process是 进入/离开临界区的进程号  
<Critical Region>  
leave_region(process);  
<Noncritical Region>
```

当一个进程想进入临界区时，先调用enter_region函数，判断是否能安全进入，不能的话等待；当进程从临界区退出后，需调用leave_region函数，允许其它进程进入临界区。

两个函数的参数均为进程号



Peterson 解决方案

```
#define FALSE 0
#define TRUE 1
#define N      2 // 进程的个数
int turn;          // 轮到谁？
int interested[N]; // 兴趣数组，所有元素初始值均为FALSE
void enter_region (int process) // process为进程号 0 或 1
{
    int other; // 另外一个进程的进程号
    other = 1 - process;
    interested[process] = TRUE; // 表明本进程希望进入临界区
    turn = process;           // 设置标志位
    while ( turn == process && interested[other] == TRUE);
}
void leave_region (int process)
{
    interested[process] = FALSE; // 本进程将离开临界区
}
```

turn、
interested是
共享变量

turn:表示要进
入临界区的进
程号

interested[i]
== TRUE表
示进程i要求进
入或正在临界
区执行



实现互斥的方案

4. 关中断

关中断;

<Critical Section>

开中断;

<NonCritical Section >

进程要进入临界区前先关中断，离开临界区前开中断。

因为CPU只有发生中断时才进行进程切换。

缺点：

(1) 对多处理机系统无效。在多处理机系统中，有可能存在一个以上的进程在不同处理机上同时执行，关中断是不能保证互斥的；

(2) 将关中断的权力交给用户不合适。



实现互斥的方案

5. 机器指令

(1) **TestAndSet指令** - TSL(Test and Set Lock)指令

为多处理机设计的计算机通常有类似的TSL指令。

格式：TSL register, memory

功能：register \leftarrow [memory]；将内存单元的值送寄存器

[memory] \leftarrow 1；置内存单元的值为非0

执行TSL指令的CPU将锁住总线，以禁止其他CPU在本指令结束之前访问内存。

TSL指令的功能用C语言描述：

```
int TSL (int* pLock)
{
    int retval;
    retval = *pLock; *pLock = 1;
    return retval;
}
```



TSL 指令实现互斥

实现互斥的方法（用C语言描述）：

设置一个共享变量lock = $\begin{cases} 0: \text{未上锁, 可进入临界区。初始值} \\ 1: \text{已上锁, 不可进入临界区} \end{cases}$

```
void enter_region ()
{
    while (TSL(&lock))
        ;
}

void leave_region()
{
    lock = 0;
}
```

```
enter_region ();

<Critical Region>

leave_region ();

<Noncritical Region>
```



TSL 指令实现互斥

实现互斥的方法（用汇编语言描述）：

设置一个共享变量lock = $\begin{cases} 0: \text{未上锁, 可进入临界区。初始值} \\ 1: \text{已上锁, 不可进入临界区} \end{cases}$

```
enter_region: tsl    register, lock
               cmp   register, 0
               jnz   enter_region
               ret
```

```
leave_region: mov   lock, 0
               ret
```

```
call enter_region
<Critical Region>
call leave_region
<Noncritical Region>
```



TSL 指令实现互斥

x86 CPU没有专门的TSL指令，但提供了可以达到类似目的的指令，如位测试指令bts。

BTS (Bit Test and Set) ; 位测试并置位

一般形式：

BTS dest, index ; CF = dest的第index位, dest的第index位 = 1

语法格式：

BTS reg16/mem16, reg16/imm8

BTS reg32/mem32, reg32/imm8

对标志位的影响：影响CF；其余标志无定义。

【例】位测试。

bts eax, 12 ; CF = eax的第12位, eax的第12位 = 1



TSL 指令实现互斥

LOCK前缀

功能：用于多处理器系统，作为某些指令的前缀，使CPU通过锁住总线等方式，以禁止其他CPU在本指令结束前访问内存。

当使用下列指令、且目标操作数为内存操作数时，可使用LOCK前缀，以保证原子性地执行对内存的“读—修改—写”操作：

- (1) 加法：ADD、ADC、INC 和XADD；
- (2) 减法：SUB、SBB、DEC和NEG；
- (3) 交换：XCHG、CMPXCHG和CMPXCHG8B；
- (4) 逻辑：AND、NOT、OR和XOR；
- (5) 位测试：BTS、BTC与BTR。

其它类型操作数或指令不能使用LOCK前缀。



TSL 指令实现互斥

使用位测试指令bts实现enter_region过程的代码如下:

```
enter_region proc  
inUse:  lock bts flag, 0; flag就是lock变量  
        jc  inUse  
        ret  
enter_region endp
```

在单处理机系统中不需要lock前缀。



机器指令实现互斥

(2) swap指令

例如，Intel x86的XCHG指令，不需要lock前缀。

swap指令的功能描述：

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



swap 指令实现互斥

实现互斥的方法（用C语言描述）：

设置一个共享变量lock = $\begin{cases} 0: \text{未上锁, 可进入临界区。初始值} \\ 1: \text{已上锁, 不可进入临界区} \end{cases}$

```
void enter_region ()
{
    int key = 1;
    while (key == 1)
        swap(&lock, &key);
}

void leave_region()
{
    lock = 0;
}
```

```
enter_region ();
<Critical Region>
leave_region ();
<Noncritical Region>
```



swap 指令实现互斥

实现互斥的方法（用汇编语言描述）：

设置一个共享变量lock = $\begin{cases} 0: \text{未上锁, 可进入临界区。初始值} \\ 1: \text{已上锁, 不可进入临界区} \end{cases}$

enter_region	proc
	mov eax, 1
inUse:	xchg eax, lock
	cmp eax, 0
	jnz inUse
	ret
enter_region	endp
leave_region	proc
	mov lock, 0
	ret
leave_region	endp



实现互斥的方案

前面给出的机器指令方法可以实现互斥，但有一个特点：

忙等待(busy waiting): 当1个进程想进入临界区时，先检测。若不允许进入，则进程不断检测，直到许可为止。

忙等待有什么问题？

(1) 浪费CPU时间。

(2) 可能引起优先级反转问题 (priority inversion problem)

设有2个进程，H优先级较高，L优先级较低。调度规则规定：只要H就绪就可以运行。在某一时刻，L处于临界区中，此时H处于就绪状态，调度程序切换到H。现在H开始忙等待，但由于L不会被调度，也就无法离开临界区，因此，H永远忙等待下去。



2.3 进程同步

四、信号量 (Semaphore)

1965年，由荷兰计算机科学家Dijkstra提出。

目的是用一个地位高于应用进程的管理者 (OS) 来解决共享资源的使用问题。



信号量

1. 什么是信号量？

信号量是OS引入的实现同步和互斥的机制。

取值：整数

访问信号量s的2个**原子操作：P、V操作**

P、V分别是荷兰语的test(proberen)和increment(verhogen)

(1) P(s) //等待，wait/down/lock

当 $s \leq 0$ 时等待，直到 $s > 0$ ；然后 $s--$;

(2) V(s) //唤醒，signal/up/unlock

$s++$;



信号量

P、V操作是原子操作（atomic operation），或称原语（primitive）

原语：用来完成特定功能的具有原子性的一段程序。

原子性：程序中的一组动作是不可分割的，要么全做，要么全不做。

原语的两方面含义：

（1）机器指令级。原语的程序段在执行过程中不允许中断。

（2）功能级。原语的程序段不允许并发执行。

信号量的使用：

必须置一次且只能置一次初值

只能执行P、V操作

除此之外，不能用其他方式访问信号量。



信号量

2. 信号量的含义

信号量 s 的值表示可用资源的数量。

$P(s)$ ：意味着请求分配一个资源，因而 s 要减1。当 $s \leq 0$ 时表示无可用资源，请求者必须等待别的进程释放了该资源后才能运行。

$V(s)$ ：即释放一个资源。



信号量

3. 信号量的实现原理

2种实现方式：

(1) 忙等待方式

(2) 阻塞方式



信号量的忙等待实现方式

// 信号量类型定义

```
typedef struct {  
    int value; // 信号量的值  
    int lock; // 锁，初始值为0  
} Semaphore_t;
```

// V操作

```
void V(Semaphore_t *ps)  
{  
    // 对ps操作的互斥  
    while (TSL(&ps->lock)) ;  
    ps->value++;  
    ps->lock = 0;  
}
```

// P操作

```
void P(Semaphore_t *ps)  
{  
    for (; ) {  
        // 对ps操作的互斥  
        while (TSL(&ps->lock)) ;  
        if (ps->value > 0) {  
            ps->value--;  
            break;  
        }  
        ps->lock = 0;  
    }  
    ps->lock = 0;  
}
```



信号量的阻塞实现方式

引入等待队列，当需要等待时，将进程链入等待该信号量的队列中。

//信号量类型定义

```
typedef struct {  
    int value; // 信号量的值  
    SemaQueue *list; // 等待该信号量的进程队列  
    int lock; // 锁，初始值为0  
} Semaphore_t;
```



信号量的阻塞实现方式

// P操作

```
void P(Semaphore_t *ps)
{
    while (TSL(&ps->lock));
    if (ps->value > 0) {
        ps->value--;
        ps->lock = 0;
    } else {
        将该进程加入ps->list;
        阻塞该进程并设置ps->lock=0;
    }
}
```

// V操作

```
void V(Semaphore_t *ps)
{
    while (TSL(&ps->lock));
    if (ps->list == NULL) {
        ps->value++;
    } else {
        从ps->list中移出一个进程P;
        将进程P放入就绪队列中;
    }
    ps->lock = 0;
}
```



信号量

4. 信号量应用实例

生产者-消费者问题 (Producer-Consumer Problem) ,
或称有界缓冲区问题。

2类进程共享1个公共的固定大小的缓冲区，缓冲区包含N个槽。

一类是生产者进程，负责将信息放入缓冲区；

另一类是消费者进程，从缓冲区中取信息。



生产者-消费者问题

使用3个信号量：

full：记录缓冲区中非空的槽数，初始值=0

empty：记录缓冲区中空的槽数，初始值=N

mutex：确保进程不同时访问缓冲区，初始值=1

```
#define N      100 //缓冲区的槽数
#define TRUE  1

//信号量定义
Semaphore_t mutex = 1,
             empty = N,
             full = 0;
```



生产者-消费者问题

```
void producer(void)
{
    while (TRUE) {
        produce(); //生产1项
        P(&empty); //申请1个空槽
        P(&mutex); //请求进入临界区
        append(); //加入缓冲区
        V(&mutex); //离开临界区
        V(&full); //递增非空槽
    }
}
```

```
void consumer(void)
{
    while (TRUE) {
        P(&full); //申请1个非空槽
        P(&mutex); //申请进入临界区
        remove(); //从缓冲区移出1项
        V(&mutex); //离开临界区
        V(&empty); //递增空槽数
        consume(); //消费数据
    }
}
```




信号量的应用

使用信号量应注意的几个问题：

- ✓ V操作是释放资源的，每个进程中连续多个V操作的出现次序关系不是很大。
- ✓ 每个进程中的多个P操作出现顺序不能颠倒。通常应先执行对资源信号量的P操作，再执行对互斥信号量的P操作，否则可能会引起死锁。
- ✓ 互斥信号量的P操作尽可能靠近临界区。
- ✓ P操作和V操作在很多时候是成对出现的。当为互斥操作时，它们同处于同一进程；当为同步操作时，则不在同一进程中出现。
- ✓ 用信号量可以解决任何同步互斥问题。



2.4 经典进程同步问题

一、哲学家进餐问题 (The Dining-Philosophers Problem)

问题描述：

有5个哲学家围坐在一张圆桌周围，每个哲学家面前有1碗饭，左右各1把叉子。

哲学家有两种活动：思考和吃饭。

只有拿到左右两把叉子才能吃饭。

吃饭后，放下叉子，继续思考。



哲学家进餐问题

哲学家活动的描述：

```
#define TRUE 1
#define N 5 //哲学家数
void philosopher (int i) //i是哲学家编号：0~N-1
{
    while (TRUE) {
        think(); //思考
        take_fork(i); //取左边的叉子fork[i]
        take_fork((i+1) % N); //取右叉fork[(i+1) % N] ;
        eat(); //吃饭
        put_fork(i); //放左叉fork[i]
        put_fork((i+1) % N); //放右叉fork[(i+1) % N] ;
    }
}
```



哲学家进餐问题解法(1)

//5个信号量，分别用于对5个叉子互斥

```
#define TRUE 1
```

```
#define N 5 //哲学家数
```

```
Semaphore_t fork[] = {1, 1, 1, 1, 1};
```

```
void philosopher (int i) //i是哲学家编号：0~N-1
```

```
{
```

```
    while (TRUE) {
```

```
        think();
```

```
        P(&fork[i]);
```

```
        P(&fork[(i+1) % N]);
```

```
        eat();
```

```
        V(&fork[i]);
```

```
        V(&fork[(i+1) % N]);
```

```
    }
```

```
}
```

当5位同时拿起左叉，如何？



哲学家进餐问题解法(2)

用1个信号量，互斥哲学家的活动

```
#define TRUE 1
#define N 5 //哲学家数
Semaphore_t fork = 1;
void philosopher (int i) //i是哲学家编号：0~N-1
{
    while (TRUE) {
        think();
        P(&fork);
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork( (i+1) % N);
        V(&fork);
    }
}
```

同一时刻只能有1位哲学家吃饭



哲学家进餐问题解法(3)

除了互斥叉子的5个信号量外，再引入用1个信号量，互斥拿左右2个叉子的动作。

```
#define TRUE 1
#define N 5 //哲学家数
Semaphore_t mutex = 1, fork[] = {1, 1, 1, 1, 1};
void philosopher (int i) //i是哲学家编号：0~N-1
{
    while (TRUE) {
        think();
        P(&mutex);
        P(&fork[i]);
        P(&fork[(i+1) % N]);
        V(&mutex);
        eat();
        V(&fork[i]);
        V(&fork[(i+1) % N]);
    }
}
```



哲学家进餐问题解法(4)

除了互斥叉子的5个信号量外，再引入用1个信号量 $e=4$ ，最多同时允许4位吃饭，保证至少有1位能拿到左右2个叉子。

```
#define TRUE 1
#define N 5 //哲学家数
Semaphore_t e = N - 1, fork[] = {1, 1, 1, 1, 1};
void philosopher (int i) //i是哲学家编号：0~N-1
{
    while (TRUE) {
        think();
        P(&e);
        P(&fork[i]);
        P(&fork[(i+1) % N]);
        eat();
        V(&fork[i]);
        V(&fork[(i+1) % N]);
        V(&e);
    }
}
```



哲学家进餐问题解法(5)

将叉子编号，哲学家拿叉子时，先拿编号小的，再拿编号大的。不会出现5位哲学家同时拿起左边叉子的情况。

```
#define TRUE 1
#define N 5 //哲学家数
Semaphore_t fork[] = {1, 1, 1, 1, 1};
void philosopher (int i) //i是哲学家编号：0~N-1
{
    while (TRUE) {
        think();
        if (i == N - 1) {
            P(&fork[0]); P(&fork[N-1]);
        } else {
            P(&fork[i]); P(&fork[i+1]);
        }
        eat(); V(&fork[i]); V(&fork[(i+1) % N]);
    }
}
```

类似的方法：给所有哲学家编号，奇数号的哲学家必须首先拿左边的叉子，偶数号的哲学家则反之。



哲学家进餐问题解法(6)

每个哲学家对应1个信号量，表示是否可以得到叉子吃饭

```
#define TRUE 1
#define N 5           //哲学家数
#define LEFT (i-1+N) % N //哲学家i的左邻居号
#define RIGHT (i + 1) % N //哲学家i的右邻居号
#define THINKING 0     //哲学家正在思考
#define HUNGRY 1       //哲学家想取得叉子
#define EATING 2       //哲学家正在吃饭
int state[] = {THINKING, THINKING, THINKING, THINKING, THINKING}; //哲学家状态
Semaphore_t mutex = 1, //临界区互斥
             s[] = {0, 0, 0, 0, 0}; //表示哲学家是否具备得到叉子吃饭的条件
void philosopher (int i) //i是哲学家编号：0~N-1
{
    while (TRUE) {
        think();
        take_forks(i); //取左右2把叉子
        eat();
        put_forks(i); //放左右2把叉子
    }
}
```



哲学家进餐问题解法(6)续

```
void take_forks(int i)
{
    P(&mutex); //进入临界区
    state[i] = HUNGRY;
    test(i); //看是否能进餐
    V(&mutex); //离开临界区
    P(&s[i]); //取得叉子进餐
}
```

```
void test(int i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        state[i] = EATING;
        V(&s[i]);
    }
}
```

```
void put_forks(int i)
{
    P(&mutex); //进入临界区
    state[i] = THINKING;
    test(LEFT); //唤醒满足条件的左邻居进餐
    test(RIGHT); //唤醒满足条件的右邻居进餐
    V(&mutex); //离开临界区
}
```

哲学家何时可以得到叉子吃饭?

饥饿且其左右邻居都不在吃饭



2.4 经典进程同步问题

二、读者-写者问题 (The Readers-Writers Problem)

问题描述：

多个Reader进程，多个Writer进程，共享文件F

要求：

允许多个Reader进程同时读文件

不允许任何一个Writer进程与其他进程同时访问（读或写）文件



读者-写者问题解法

```
int rc = 0; //reader的个数
Semaphore_t mutex = 1, //互斥对rc的访问
            f = 1; //互斥对文件F的访问
```

```
void reader()
{
    P(&mutex); //互斥对rc的访问
    rc++;
    if (rc == 1) P(&f); //第1个读者
    V(&mutex);
    read_file(); //读文件F
    P(&mutex);
    rc--;
    if (rc == 0) V(&f); //最后1个读者
    V(&mutex);
    use_data(); //使用数据，非临界区操作
}
```

```
void writer()
{
    form_data(); //准备数据
    P(&f);
    write_file(); //写文件F
    V(&f);
}
```

如果Reader不断，会出现
什么情况？

如何改进？



2.4 经典进程同步问题

三、睡眠的理发师问题 (The Sleeping-Barber Problem)

问题描述(一)：

理发店有1位理发师和1把理发椅。

没有顾客时，理发师便在理发椅上睡觉；

顾客到来时，如果理发师在睡觉，则叫醒理发师，否则等候理发；理完发后离开。



睡眠的理发师问题(一)解法

```
Semaphore_t customers = 0, //等候理发的顾客数
            barbers = 0; //等候顾客的理发师数
void barber() //理发师进程
{
    while (TRUE) {
        P(&customers); //无顾客时睡觉
        V(&barbers); //1个理发师可以理发了
        cut_hair(); //为1位顾客理发
    }
}
void customer() //顾客进程
{
    V(&customers); //来了1位顾客，必要时唤醒理发师
    P(&barbers); //等候理发
    get_haircut(); //坐到理发椅上接受理发，理完后离开
}
```



睡眠的理发师问题

问题描述(二)：

理发店里有1位理发师、1把理发椅和3把供等候理发的顾客坐的椅子。

没有顾客时，理发师便在理发椅上睡觉；

顾客到来时，**如果没有空椅子坐，则离开**；如果理发师忙，但有空椅子坐，则坐下等候；如果理发师在睡觉，则叫醒理发师；理完发后离开。



睡眠的理发师问题(二)解法

```
Semaphore_t customers = 0, //等候理发的顾客数
            barbers = 0, //等候顾客的理发师数
            chairs = 3; //空椅子数
```

```
void barber() //理发师进程
```

```
{
    while (TRUE) {
        P(&customers);
```

```
        V(&barbers);
        cut_hair();
```

```
    }
}
```

```
void customer() //顾客进程
```

```
{
    if (chairs == 0) return;
```

```
    V(&customers);
```

```
    P(&barbers);
    get_haircut();
```

```
}
```




睡眠的理发师问题(二)解法

```
# define CHAIRS 3
```

```
Semaphore_t customers = 0, //等候理发的顾客数
```

```
barbers = 0, //等候顾客的理发师数
```

```
int waiting = 0; //等候理发的顾客数(不包括正在理发的, 等于customers的值)
```

```
void barber() //理发师进程
```

```
{
```

```
    while (TRUE) {
```

```
        P(&customers);
```

```
        waiting--;
```

```
        V(&barbers);
```

```
        cut_hair();
```

```
    }
```

```
}
```

```
void customer() //顾客进程
```

```
{
```

```
    if (waiting >= CHAIRS) { //没有空椅子就离开  
        return;
```

```
    }
```

```
    waiting++;
```

```
    V(&customers);
```

```
    P(&barbers);
```

```
    get_haircut();
```

```
}
```



睡眠的理发师问题(二)解法

```
# define CHAIRS 3
```

```
Semaphore_t customers = 0, //等候理发的顾客数
```

```
    barbers = 0, //等候顾客的理发师数
```

```
    mutex = 1; //互斥
```

```
int waiting = 0; //等候理发的顾客数(不包括正在理发的，等于customers的值)
```

```
void barber() //理发师进程
```

```
{
    while (TRUE) {
        P(&customers);
        P(&mutex);
        waiting--;
        V(&mutex);
        V(&barbers);
        cut_hair();
    }
}
```

```
void customer() //顾客进程
```

```
{
    P(&mutex);
    if (waiting >= CHAIRS) { //没有空椅子就离开
        V(&mutex); return;
    }
    waiting++;
    V(&customers);
    V(&mutex);
    P(&barbers);
    get_haircut();
}
```



2.5 进程间通信 (InterProcess Communication, IPC)

P、V操作实现的是进程之间的低级通讯，所以P、V操作是低级通讯原语。

它只能传递简单的信息，不能传递交换大量信息。

因此，引入进程间的高级通讯方式。



2.5 进程间通信 (InterProcess Communication, IPC)

一、进程通信的类型

1. 共享存储区 (Shared Memory)

相互通信的进程间设有公共的内存区，每个进程既可向该公共内存中写，也可从公共内存中读，通过这种方式实现进程间的信息交换。



进程通信的类型

2. 消息传递 (Message passing)

源进程发送消息，目的进程接受消息。所谓消息，就是一组数据。

(1) 消息队列 (message Queue) 或消息缓冲

发送者发消息到一个消息队列中；

接收者从相应的消息队列中取消息。

消息队列所占的空间从系统的公用缓冲区中申请得到。

(2) 邮箱 (mailbox)

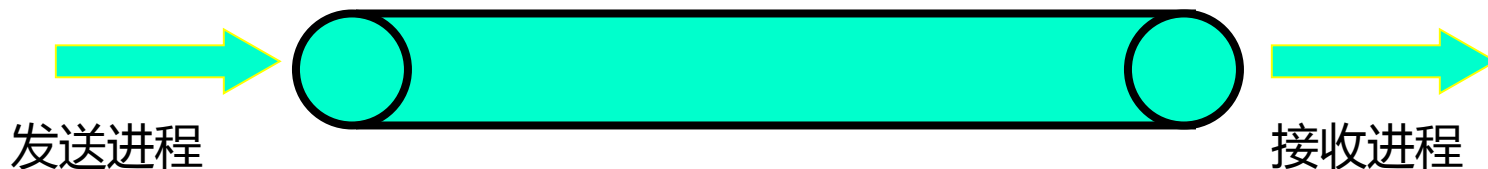
发送者发消息到邮箱，接收者从邮箱取消息。

邮箱是一种中间实体，一般用于非实时通信。

进程通信的类型

3. 管道 (pipe)

首创于Unix。用于连接一个读进程、一个写进程，以实现它们之间通信的共享文件，称为pipe文件。



管道分为下列2种：

- ✓ 有名管道
- ✓ 无名管道



2.5 进程间通信 (InterProcess Communication, IPC)

二、消息队列

1. 消息的发送 / 接收原语Send/Receive

(1) Send(QID qid, MSG msg) //qid是队列ID , msg是消息

- ✓ 阻塞：若消息队列满，等待。
- ✓ 非阻塞：若消息队列满，立即返回。

(2) Receive(QID qid, MSG msg)

- ✓ 阻塞：若消息队列空，等待。
- ✓ 非阻塞：若消息队列空，立即返回。

通常，Send采用非阻塞，Receive可采用两种方式。



消息队列

2. Send/Receive的实现

设置下列信号量：

mutex：互斥对消息队列的访问，初始值 = 1

sm：消息队列中的消息个数，初始值 = 0



Send/Receive的实现

//非阻塞的Send

```
void Send(QID qid, MSG &pMsg)
```

```
{
```

```
    向系统申请一个缓冲区b ;
```

```
    将pMsg中的消息复制到b中 ;
```

```
    p(&mutex);
```

```
    将消息缓冲区b挂到qid对应的消息队列中 ;
```

```
    V(&mutex);
```

```
    V(&sm); //消息个数加1
```

```
}
```



Send/Receive的实现

```
//阻塞的Receive  
void Receive(QID qid, MSG &pMsg)  
{  
    P(&sm);  
    p(&mutex);  
    从qid对应的消息队列中摘下第1个消息 ;  
    V(&mutex);  
    将消息复制到pMsg中 ;  
    释放消息缓冲区 ;  
}
```



2.6 线程 (Thread)

一、为什么要引入线程？

什么是进程？

是程序的1次执行（程序执行的1个实例）

每个进程有自己的地址空间。

为什么引入进程？

多任务的需要。在内存中同时有多个可执行的进程，以提高效率（特别是CPU的利用率）。

因此，需要对进程进行管理，以避免冲突：

借助于PCB，记录进程的描述和控制信息、上下文状态。



为什么要引入线程？

以多进程方式解决多任务，有什么问题？

(1) 进程的上下文切换复杂、耗时。

(2) 多个进程之间如何共享变量？

【例】一个简化的生产者-消费者问题。

x是共享变量，int型。

producer：修改x的值。

consumer：输出x的值。

✓ 文件

✓ 借助OS，如共享存储器。



2.6 线程 (Thread)

二、什么是线程？

线程是进程的 1 条执行路径。

1个进程可以有多个线程，其中至少有1个主线程（ primary thread ）。

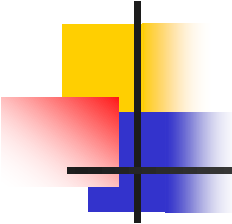
1个进程内的多个线程在同一个地址空间内（ 共享该进程的地址空间 ）。

每个线程有自己的线程控制块TCB（ Thread Control Block ），包含自己的堆栈和状态信息。TCB比PCB小得多。



如何使用线程？

```
int x = 0;
main()
{
    CreateThread(producer, ...); //创建生产者线程
    CreateThread(consumer, ...); //创建消费者线程
    处理其他任务; //如GUI , 与用户交互
}
void producer()
{
    修改x的值 ;
}
void consumer()
{
    输出x的值 ;
}
```



2.6 线程 (Thread)

三、线程的实现机制

2种方式：

- (1) 用户级线程 (User Level Thread)
- (2) 核心级线程 (Kernel Level Thread)



线程的实现机制

(1) 用户级线程

由在用户空间执行的[线程库](#)来实现，OS对此一无所知。

线程库提供线程创建、撤消、上下文切换、通信、调度等功能。

- ✓ 资源分配的实体是进程；
- ✓ OS分配CPU时间（调度）的基本单位也是进程。
- ✓ 线程的调度只进行线程上下文切换，而且在用户态下。



线程的实现机制

(2) 核心级线程

OS内核提供对线程的支持：

系统调用API（线程创建、撤销等）。

- ✓ 资源分配的实体是进程；
- ✓ OS分配CPU时间（调度）的基本单位是线程。
- ✓ 线程的调度在核心态下。



线程的实现机制

核心级线程与用户级线程这2种实现机制的比较：

(1) 同一进程内的多个线程是否可以在多个处理机上并行执行

用户级线程：不能

核心级线程：可以

(2) 同一个进程内的线程切换性能

用户级线程：性能高，无需陷入内核

核心级线程：性能低，需要陷入内核

(3) 用户级线程只要有线程库的支持，即可运行在任何OS上。



2.6 线程 (Thread)

一个多任务问题的实现方式：

✓ 单进程、单线程

实现复杂。需要自己实现多个任务的调度。

✓ 多进程，每个进程单线程

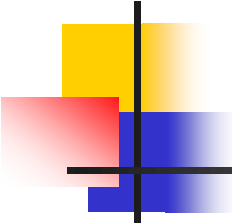
多个任务处在不同的地址空间，不会相互干扰；
不能共享全局变量。

✓ 单进程、多线程

多个任务处在同一个地址空间内，可以共享全局变量；
但容易相互干扰。

✓ 多进程、多线程

结合多进程和多线程的优点；
关系密切的任务作为同一进程的多个线程，否则作为不同的进程。



2.7 小结

一、重点掌握的内容

1. 进程、线程、程序的关系与差异

进程的1条执行路线；
共享进程的地址空间。

程序的1次执行；
有独立的地址空间。

静态的可执行文件

重点掌握的内容


2. 进程的基本状态及其转换

新建；就绪；运行；阻塞；完成

运行  就绪？ 时间片到，或有优先级更高的进程出现而被剥夺

运行  阻塞？ 等待事件（或请求I/O）

就绪  阻塞？ ×

就绪  运行？ 被OS调度程序选中

阻塞  就绪？ 等待的事件到（或I/O完成）

阻塞  运行？ ×?



重点掌握的内容

进程的描述和控制信息：
pid，状态，优先级等。

3. PCB是什么？其内容、作用？

PCB是OS管理进程的数据结构。
TCB也类似。

伴随着进程的整个生命周期：
进程创建时，OS创建PCB；
进程完成时，OS撤销PCB；
进程运行时，PCB作为调度依据。



重点掌握的内容

多任务的需要。

4. 为什么要引入进程？为什么要引入线程？

(1)性能。将一个多任务应用实现为单进程、多线程可能比实现为多进程效率更高。因为创建进程比创建线程慢。

(2)程序设计。资源共享更容易处理。因为一个进程内的多个线程可以共享进程的资源（全局变量、打开的文件等），便于任务之间的通信。



重点掌握的内容

5. 多进程或多线程的应用场合

(1) 前台和后台操作

前台：一个线程（主线程）显示界面（菜单），读取用户输入的命令。

后台：一个甚至多个线程执行用户命令，如计算、写磁盘文件等。

好处：用户可以在一个命令完成前输入下一个命令。

(2) 异步处理

例如字处理软件：专门有1个线程（或进程）周期性地执行写盘操作，以避免突然掉电带来的损失。

(3) 模块化的程序结构

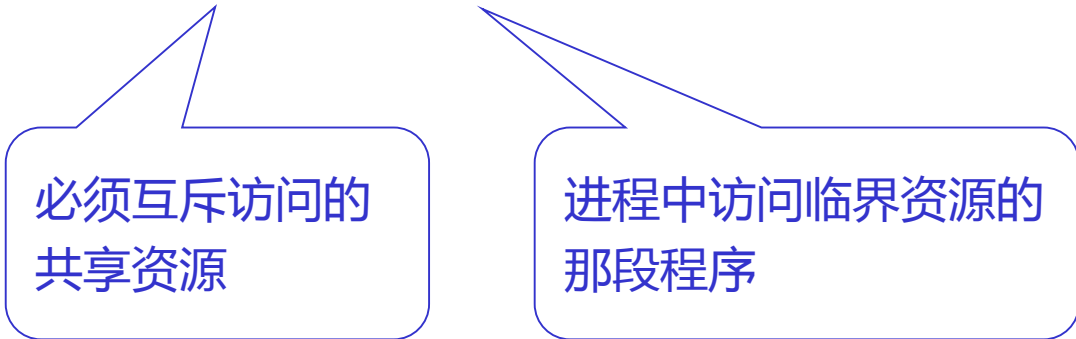
涉及到多种事件、多种资源和输入/输出任务。



重点掌握的内容

6. 进程（线程）同步

（1）临界资源、临界区



必须互斥访问的
共享资源

进程中访问临界资源的那段程序



重点掌握的内容

(2) 原子性、原语

不可分割的一组动作，要么全做，要么全不做

具有原子性的一段程序

(3) 实现互斥的方法

- ✓ 关中断：只能用于单处理机系统。
- ✓ 机器指令：TSL指令，Swap指令。



重点掌握的内容

(4) 信号量：含义、基本操作

表示可用资源的数量;
用于互斥时, 初值取1

2个基本操作: P、V

P(s):请求分配1个资源。因此, $s-1$ 。当
 $s \leq 0$ 时, 说明无可用资源, 必须等待。

V(s):释放1个资源, 故 $s+1$ 。

(5) 利用信号量设计同步算法

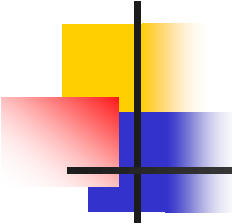
非常重要, 也是难点。



重点掌握的内容

7. 进程间通信 (IPC) 的几种方式

- ✓ 共享存储区
- ✓ 消息队列
- ✓ 管道



2.7 小结

二、现代操作系统基本特征的具体体现

并发，共享，虚拟，不确定

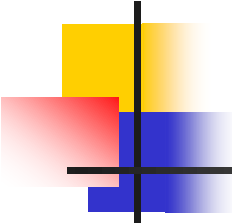
1. 并发(Concurrency)

需要同步和保护

- ✓ 多进程（线程）之间的交替执行（体现在分时）
- ✓ CPU和I/O之间的并行
- ✓ 多CPU的并行

2. 共享(Sharing)

- ✓ OS和用户进程之间、用户进程之间都存在共享
- ✓ 互斥共享：资源的排他使用
- ✓ 交替使用：如CPU



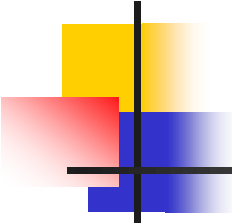
2.7 小结

3. 虚拟(Virtual)

把一个物理实体映射为一个或多个逻辑实体。

体现在：

- ✓ 虚拟处理机。单CPU，多进程分时使用，仿佛每个进程都有1个CPU
- ✓ 虚拟存储器
- ✓ 虚拟设备



2.7 小结

4. 不确定

- ✓ 由并发和共享引起。
- ✓ 系统中可同时运行多个用户进程（或线程），而每个进程的运行时间、要使用哪些资源、进程的推进顺序和速度等，事先是不知道的。
- ✓ 这种不确定性会引起难以重现的错误。



**OS必须为解决并发和共享问题提供支持；
应用程序在设计时也必须考虑。**



本章作业

教材《计算机操作系统教程（第4版）》

p78 :

3.3, 3.11



课 外 练 习 题

1. 写出Reader-Writer问题的算法，避免由于不断有Reader出现而使得Writer无限期等待。
2. 设计C程序（可以嵌入汇编语言），以忙等待方式实现信号量及其P、V操作。
3. 设计C程序，实现生产者-消费者问题。