



# 操作系统

## 第一章

- 操作系统的作用
- 操作系统的基本特征
- 分配的资源
- 中断
- 直接内存访问 (DMA)
- 多CPU、多核区别
- 多道程序设计系统
  - 工作图
  - 分时系统
- 操作系统双重模式
  - 用户模式
  - 内核模式
  - 模式之间相互切换

## 第二章

- 系统调用
- 库函数
- 系统调用类型5种
- 微内核vs宏内核vs模块化设计
  - 微内核
  - 宏内核
  - 模块化设计

## 第三章

- 进程
- 进程与线程的比较
- 进程控制块PCB
- 进程的切换
- 长、中、短期调度
  - 长期调度
  - 中期调度
  - 短期调度
- 进程的创建
  - fork函数创建子进程
  - exec函数作用
- 进程通信
  - 消息传递
  - 共享内存
- 并发和并行

## 第四章 线程

- 线程的优势
- 线程模型

多对一

一对一

多对多

## 第五章 CPU调度

先到先服务 FCFS

短作业优先 SJF

优先级调度 Priority

轮转法 RR

## 进程同步

竞争条件

临界区 critical section

解决临界区问题

Peterson

禁止中断

原子指令

PV信号量

互斥操作

同步操作

金典问题

生产者消费者

读者-写者

哲学家进餐

## 第七章 死锁

死锁特征

实例解释1

实例解释2

资源分配图

死锁预防

死锁避免

银行家算法

## 第八章 内存

重定位bind

MMU

动态加载、动态链接

动态加载

动态链接

内存分配

连续地址分配（动态内存分配）

多重分区分配

首次适应

最佳适应

最差适应

碎片

不连续分页

页表很大的时候

采用层次页面/多级页表

TLB有效访问时间

哈希页表

反向页表

分段 和段页结合

段页结合

第九章 虚拟内存

优点

请求调页过程

页面置换（淘汰）

FIFO

OPT(MIN)

LRU

换页产生的现象

# 操作系统

## 第一章

### 操作系统的作用

1. 资源分配器
2. 控制程序

### 操作系统的基本特征

并发、资源共享

### 分配的资源

CPU、内存、IO

# 中断

在计算机执行期间，系统内发生任何非寻常的或非预期的急需处理事件使得CPU暂时中断当前正在执行的程序而，转去执行相应的时间处理程序。待处理完毕后又返回原来被中断处继续执行或调度新的进程执行的过程。

分类：

外部中断、内部中断trap 、 exception

## 直接内存访问（DMA）

总线盗用

Direct Memory Access (DMA)

允许外围设备和主内存之间直接传输它们的I/O数据，而不需要系统处理器的参与。

## 多CPU、多核区别

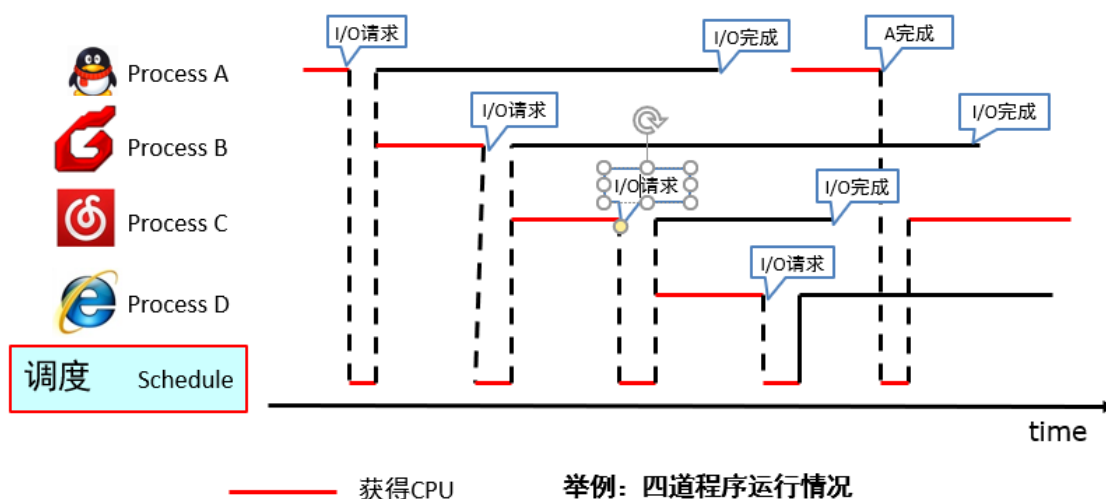
多核心共享相同的Cache

多CPU每个CPU都拥有独立的Cache

## 多道程序设计系统

提高CPU的利用率

### 工作图



## 分时系统

在分时系统中，虽然CPU还是通过在作业之间的切换来执行多个作业，但是由于切换频率很高，用户可以在程序运行期间与之进行交互。

分时操作系统要求响应时间通常小于1秒。

## 操作系统双重模式

硬件提供模式位：内核模式（0）和用户模式（1）

区分操作系统所执行的任务和用户执行的任务。

### 用户模式

当计算机系统表示用户应用程序正在执行，系统处于用户模式。

### 内核模式

将能引起损害的机器指令作为特权指令。

只有在内核模式下才可以执行特权指令。

### 模式之间相互切换

当用户应用程序需要操作系统的服务（通过系统调用），它必须从用户模式转换过来执行请求。

当执行完成之后再次切换回用户模式

操作系统启动的时候使用内核模式，启动完成之后切换用户模式，发生系统调用之后内核模式，系统调用结束之后切换回内核模式。

## 第二章

命令、GUI、Apps 都是应用OS提供的函数接口编程序

### 系统调用

操作系统提供的、与用户程序之间的接口，也就是操作系统提供给程序员的接口，这些接口提供了对系统硬件功能的操作。

### 库函数

库函数是对系统调用的又一次封装：把函数放到库里，给用户使用。

一个库函数有可能含有一个系统调用，有可能有好几个系统调用，当然也有可能没有系统调用，比如有些操作就不需要涉及内核的功能。

系统命令：是一个可执行程序，内部引用了系统函数来实现相应的功能。

## 系统调用类型5种

- Process control 进程控制
- File manipulation 文件管理
- Device manipulation 设备管理
- Information maintenance 信息维护
- Communications 通信

## 微内核vs宏内核vs模块化设计

### 微内核

代表：WindowNT,Minix,Mac

微内核技术：将所有非基本部分从内核中移走，并将它们实现为系统程序或用户程序。

微内核的主要功能是使客户程序和运行在用户空间的各种服务之间进行通信。通信以消息传递形式提供。

模块化程度高，一个服务失效不会影响另外一个服务。

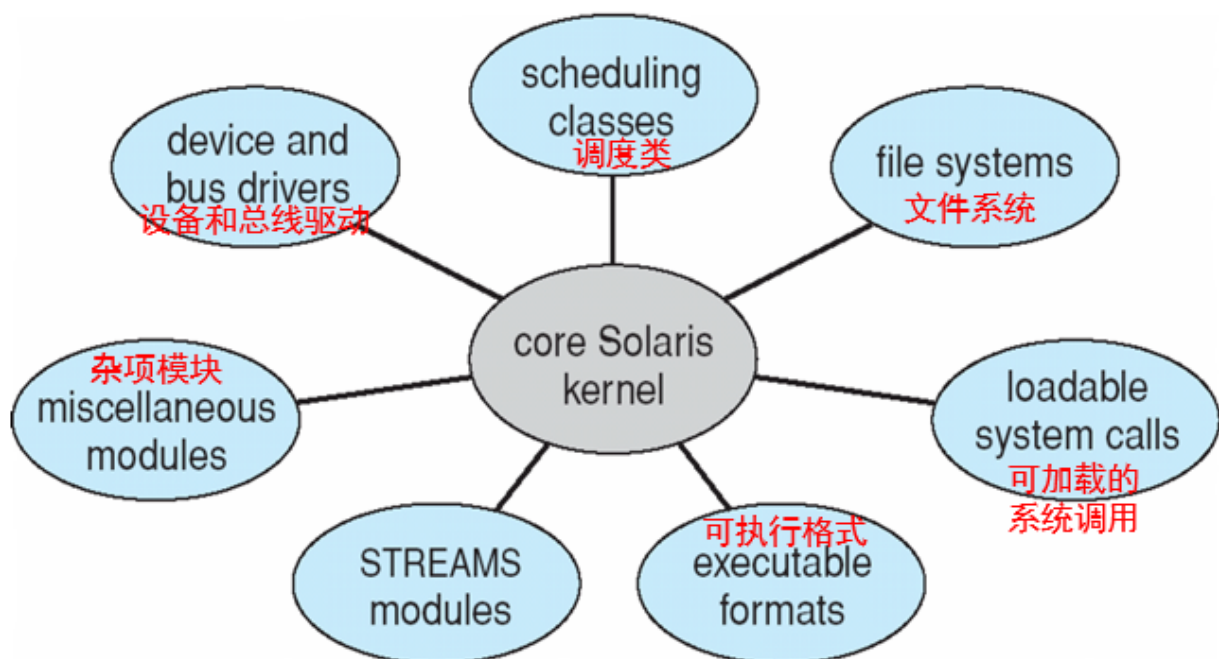
### 宏内核

代表系统：Unix, Linux

将内核从整体上作为一个大过程来实现，所有的内核服务都在一个地址空间运行，相互之间直接调用函数，简单高效。

### 模块化设计

很多现代的操作系统实际上都是混合结构。



内核有一组核心部件，以及在启动或运行时对附加服务的动态链接。

## 第三章

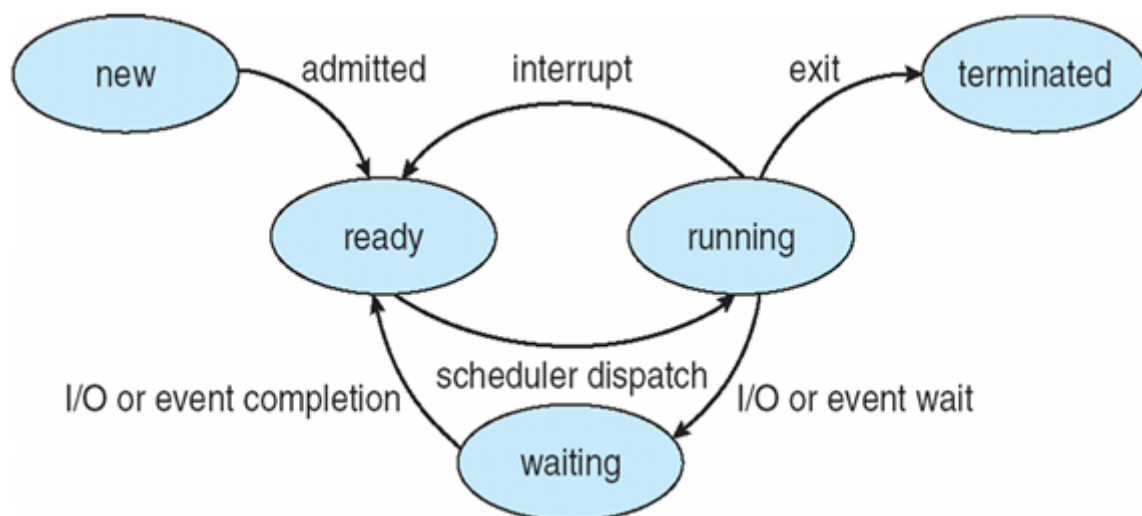
计算机解决问题的本质是靠进程。

## 进程

程序是被动执行的，进程是主动的。

运行中的程序，资源分配的最小单位，CPU调度的一个单位。

进程状态图



进程的基本特征：动态性（最基本特征）、并发性、独立性、异步性、结构性。



# 进程与线程的比较

**进程**是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位。

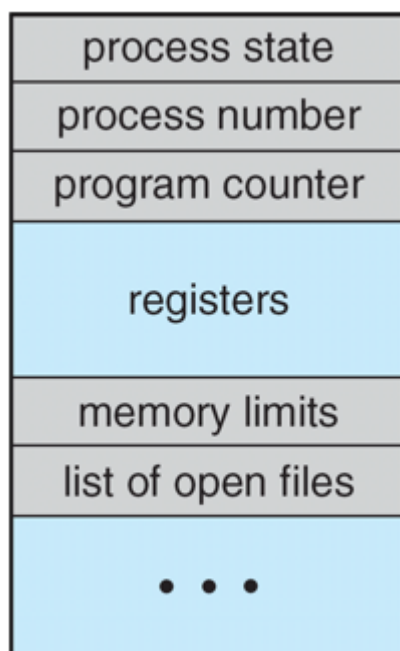
**线程**有时称轻量级进程，进程中的一个运行实体，是一个CPU调度单位。

不同比较：

1. **调度**，线程作为调度的基本单位，同进程中线程切换不引起进程，当不同进程的线程切换才引起进程切换；进程作为拥有资源的基本单位。
2. **并发性**，一个进程间的多个线程可并发。
3. **拥有资源**，线程仅拥有属于进程的资源；进程是拥有资源的独立单位。
4. **系统开销**，进程开销较大，线程小。

线程从进程中继承了什么？有自己的什么

## 进程控制块PCB



- 进程状态
- 程序计数器
- CPU寄存器
- CPU调度信息
- 内存管理信息
- 记账信息
- I/O状态信息

进程代码、页表不在PCB中，但PCB中有它们的指针。

## 进程的切换

将CPU切换到另一个进程需要保存当前进程的状态并恢复另一个进程的状态（上下文切换），发生上下文切换时，内核会将旧进程的状态保存在其PCB中，然后装入经调度要执行的并已保存的新进程的上下文。

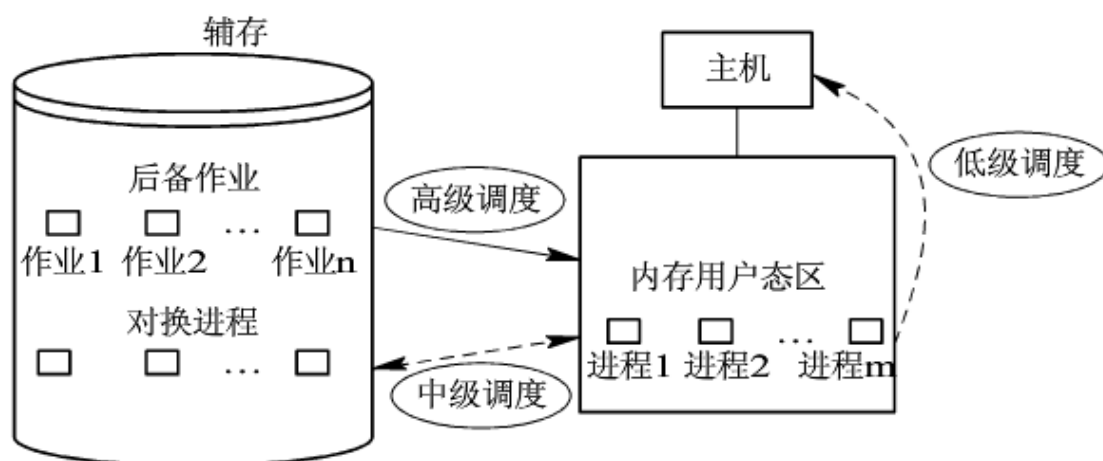
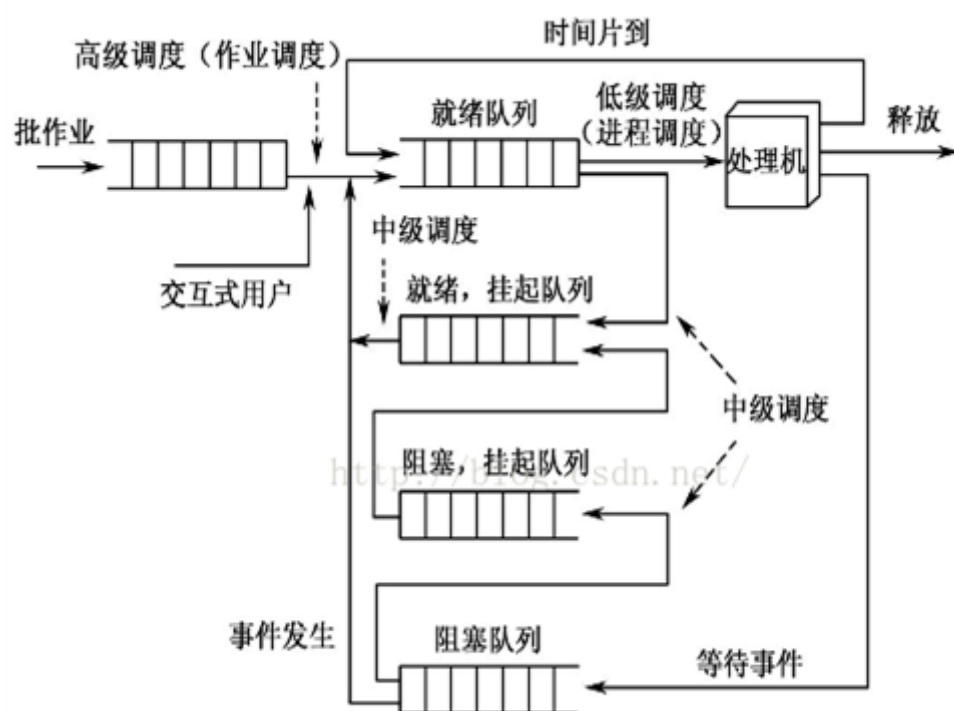
进程对切换对CPU带来的影响：

上下文切换过程中系统并不能做什么有用的工作，切换的时间越长，CPU利用率越低

## 长、中、短期调度

中期调度：换入换出

许多进程或线程都准备使用CPU进程任务处理时，就会存在资源竞争和分配问题，一般都会将进程或线程放到一个缓冲池钟函中，等待合适的时机调度程序从中选择一个进程或线程交给CPU处理。



## 长期调度

又称作作业调度或高级调度，这种调度将**已进入系统并处于后备状态的作业按算法选择一个或一批，为其建立进程，并进入主机，当该作业执行完毕时，还负责回收系统资源**，在批处理任务中，需要有作业调度的过程，以便将它们批量的装入内存，在分时系统和实时系统中，通常不需要长期调度。它的频率比较低，主要用来控制内存进程的数量。

## 中期调度

交换调度。它的核心思想是能将进程从内存或CPU竞争中移出，从而降低多道程序设计的程度，之后进程能被重新调入内存，并从中断处继续执行，这种交换的操作可以调整进程在内存中存在的数量和时机。其主要任务是按照给定的原则和策略，**将处于外存交换区中的就绪状态或等待状态的进程调入内存，或把处于内存就绪转态或等待状态的进程交换到外存区。**

## 短期调度

又称进程调度，微观调度或低级调度。这也是通常说的调度，一般情况下使用最多的就是短期调度。他的主要任务是**按照某种策略和算法将处理机分配给一个处于就绪状态的进程**，分为抢占式和非抢占式。

它们3者的区别在于频率不同，长期调度频率较低，短期调度频率较高。

# 进程的创建

## fork函数创建子进程

子进程，系统调用fork（）的返回值为0；而对于父进程，返回值为子进程的进程标识符（非零）。  
（实验）

## exec函数作用

在系统调用fork（）之后，一个进程会使用系统调用exec（），以用新程序来取代进程的内存空间。

系统调用exec（）将二进制文件装入内存）并开始执行。采用这种方式，两个进程能互相通信，并能按各自的方法执行。父进程能创建更多的子进程，或者如果在子进程运行时没有什么可做，那么它采用系统调用wait（）把自己移出就绪队列来等待子进程的终止。

子进程从父进程中继承到了些什么，没继承什么

# 进程通信

两种模型：消息传递、共享内存

## 消息传递

通过在协作进程间交换消息来实现通信。

## 共享内存

建立起一块供协作进程共享的内存区域，进程通过向此共享区域读或写入数据来交换信息。

## 并发和并行

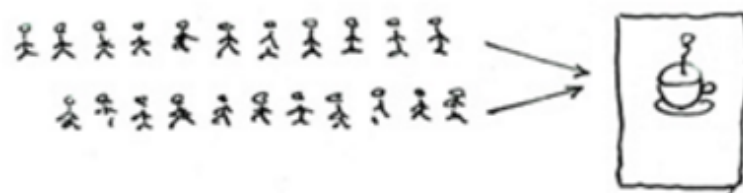
顺序执行：

- ✓ 资源封闭
- ✓ 结果封闭
- ✓ 可再现

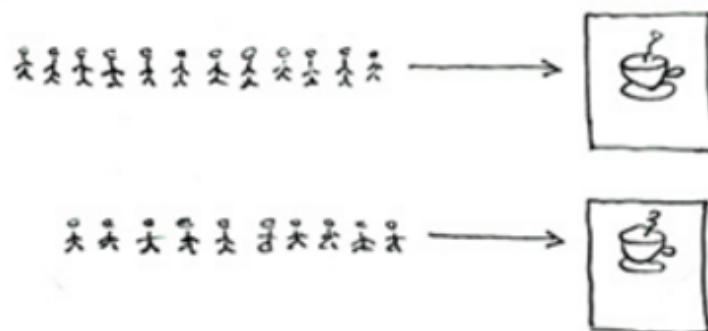
并发执行：

- ✓ 间断性
- ✓ 失去封闭性
- ✓ 不可再现性

Concurrent = Two Queues One Coffee Machine



Parallel = Two Queues Two Coffee Machines



© Joe Armstrong 2013

**Concurrent** = Two queues and one coffee machine.

**Parallel** = Two queues and two coffee machines.

## 第四章 线程

进程 = 地址空间 + 指令执行序列

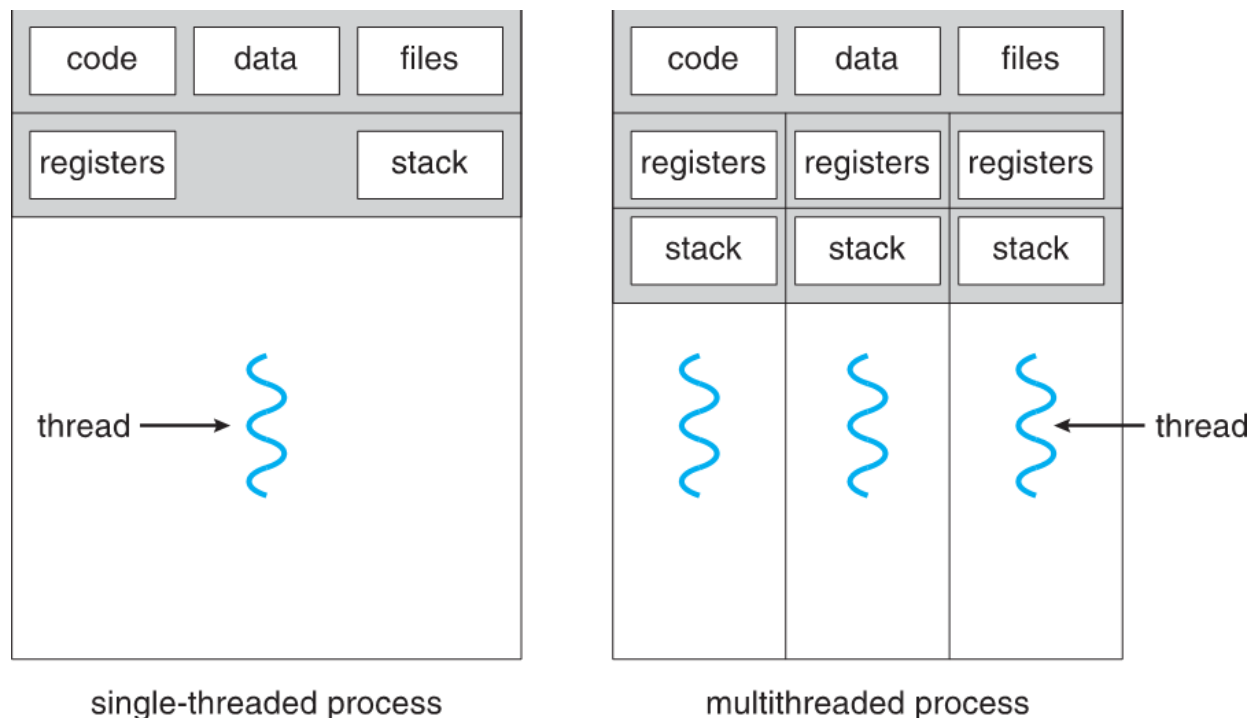
线程 = 一个地址空间 + 多个指令执行序列

线程具有并发的优点，却比进程代价低得多。

线程在同一个地址空间中，线程库可以用户级别实现。

操作系统可以不支持线程。

线程实现的细节，上下文切换是核心。



线程拥有：线程ID、程序计数器、寄存器集合和栈

它与属于同一进程的其他线程共享：代码段、数据、文件

## 线程的优势

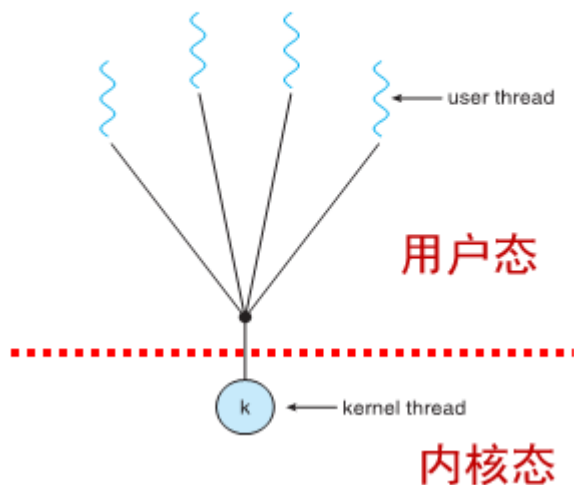
- 响应度高
- 资源共享
- 经济：进程创建所需要的内存和资源的分配比较昂贵。
- 多处理器体系结构的利用

## 线程模型

### 多对一

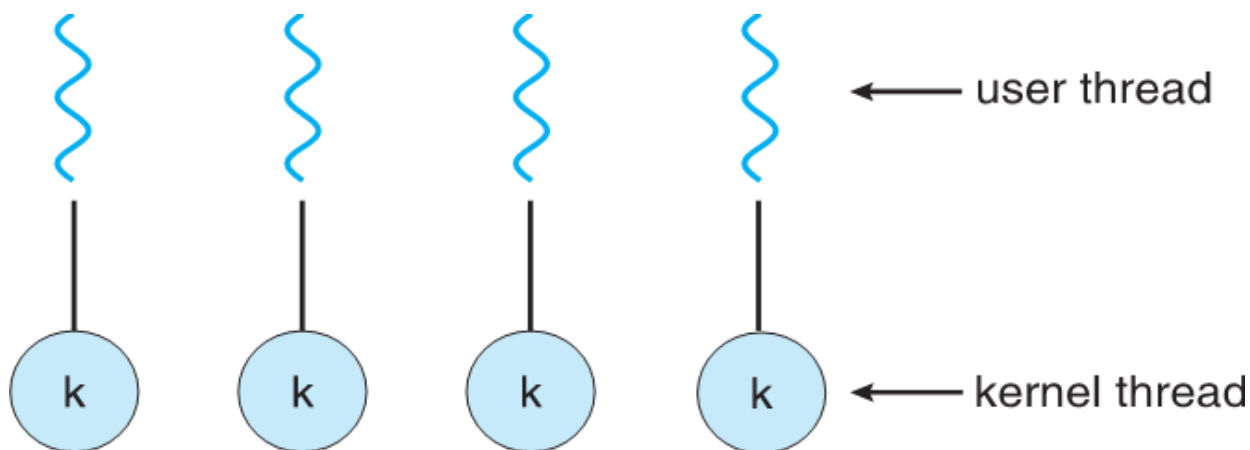
将许多用户级线程映射到一个内核线程。线程的管理是在用户空间进行的，但是如果一个线程执行了阻塞系统调用，那么整个进程会阻塞，因此只允许一个时刻内有一个进程访问内核。

线程管理是由线程库在用户空间进行的，因而效率较高



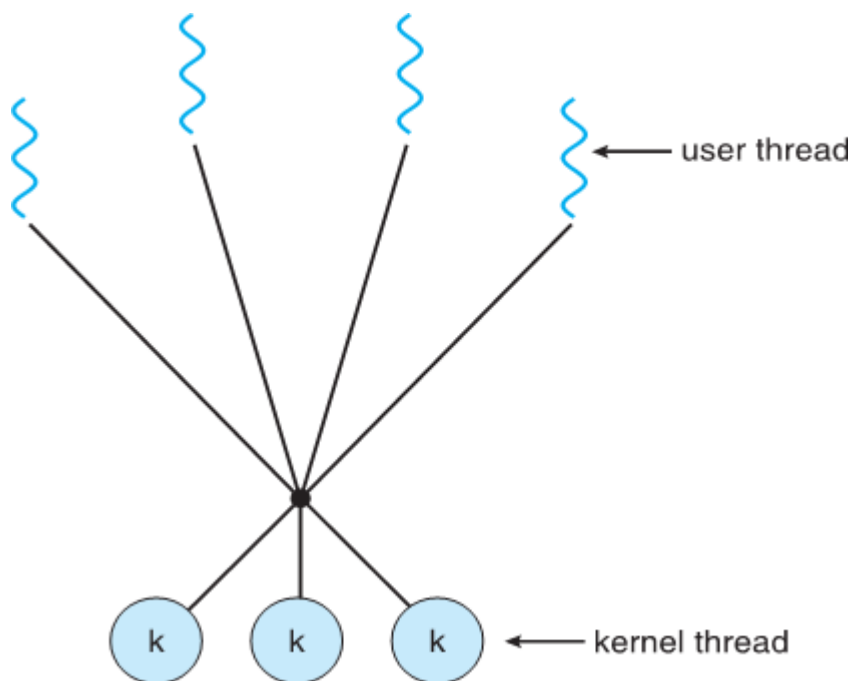
## 一对一

一个用户线程映射到一个内核线程，但是创建内核线程会影响应用程序的性能，限制了系统所支持的线程数量。



## 多对多

复用了许多用户线程到同样数量或更小数量的内核线程上。当一个线程造成了阻塞，内核能调度另一个线程来执行。



## 第五章 CPU调度

并发能提高效率，并发的核心是进程能让出CPU，就是调度。

线程、进程都能被调度。

调度任务分类：交互式、批处理

I/O限制的程序要比CPU限制的程序先执行：

因为I/O在执行的时候不占用CPU，CPU可以继续做其它事，这样CPU的利用率更高。

调度时机分类：抢占式、非抢占式

CPU 调度算法 FCFS、SJF、SRTF

周转时间：从进程提交到进程完成的时间段称为周转时间。也就是

等待时间：在就绪队列中等待所花费时间之和。

响应比：  $\text{周转时间} / \text{运行时间}$

### 先到先服务 FCFS

## 先到先服务调度

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

非抢占式

护航效应：其他进程都等待一个进程释放CPU，这个效果会导致CPU使用率变低。

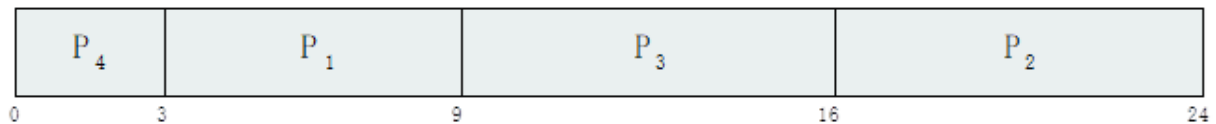
## 短作业优先 SJF

当CPU空闲的时候他会赋给具有最短CPU区间的进程，如果两个进程具有相同长度，那么使用FCFS调度处理。

非抢占式

Process	Burst Time
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$
- SJF是最佳的：对于给定的一组进程，SJF算法的平均等待时间最小。

SRTF

抢占式：

如果一个新的进程具有更短的CPU区间，则挂起当前运行任务，优先执行最短时间任务。

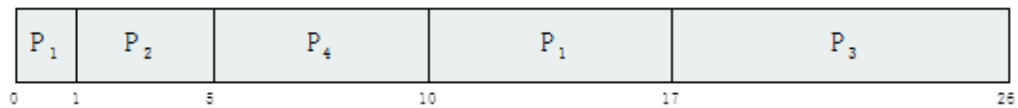


## 最短剩余时间优先调度

Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

Preemptive SJF Gantt Chart



Average waiting time =  $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$  msec

饿死现象:

如果一长作业进入系统的就绪队列，由于调度程序总是优先调度哪些后进来的短作业，导致长作业长期不被调度。

## 优先级调度 Priority

非抢占

批处理

优先级高的进程先获得执行机会，优先级相同的任务采用FCFS调度。

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

Priority scheduling Gantt Chart



Average waiting time =  $(0+1+6+16+18) / 5 = 8.2$  msec

优先级调度会长生饥饿现象。

老化现象:

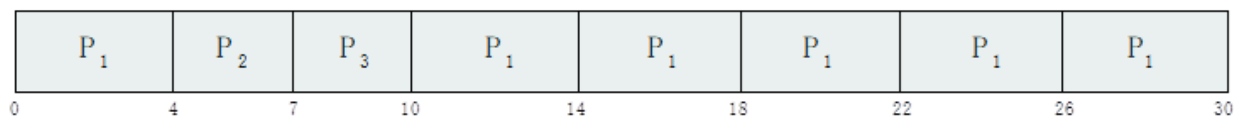
随着时间的推进，进程的优先级逐渐提高。

# 轮转法 RR

适用于交互式系统

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:



响应度高。

## 进程同步

并发，多个进程同时存在，相互影响。

非原子操作共享变量，出现语义错误

竞争的条件：临界区，互斥

peterson算法、强制的关中断（多CPU无效）、TestAndSet不适合用户实现，所以封装成锁PV操作。

一般的锁忙等，引入睡眠，最终化为信号量。

并发进程的关系：同步、互斥、进程通信

互斥：两个或以上的进程要同时访问某个共享变量，不能同时访问，否则会导致数据的不一致，产生时间有关的错误。

同步：合作进程间有意识的行为，进程之间有一定的依赖关系，有一定的同步机制保证他们的执行次序。

## 竞争条件

和时间有关的共享数据语义错误。

多个进程并发访问和操作同一数据且执行结果与访问发生的特定顺序有关。

## 临界区 critical section

临界资源：一次仅允许一个进程使用的共享资源。

临界区：无论是硬件临界资源，还是软件临界资源，多个进程必须相互地对它进行访问。临界区是每个进程中访问临界资源的那段程序。

临界区访问原则：每次只允许一个进程进入临界区，进入后不允许其他进程进入。

## 解决临界区问题

互斥：如果进程 $P_i$ 在其临界区内执行，那么其他进程都不能在某临界区内执行。

前进：如果没有进程在其临界区内执行且有进程需进入临界区，那么只有那些不在剩余区内执行的进程可参加选择，以确定谁能进入临界区，且这种选择不能无限推迟。

有限等待：从一个进程做出进入临界区的请求，直到该请求允许为止，其他进程允许进入临界区的次数上限。

### Peterson

适用于两个进程临界区与剩余区交互执行

变量turn 表示哪个进程可进入临界区

flag 表示哪个进程想进入临界区

谦让的算法，但是无法解决同步问题

<pre>do {     flag[0] = true;     turn = 1;     while (flag[1] &amp;&amp; turn == 1);         critical section     flag[0] = false;         remainder section } while (true);</pre>	<pre>do {     flag[1] = true;     turn = 0;     while (flag[0] &amp;&amp; turn == 0);         critical section     flag[1] = false;         remainder section } while (true);</pre>
---	---

Peterson算法解释

#### 条件1: Mutual Exclusion (互斥)

- ✓ 如果两个进程都进入了临界区，则：
- ✓ 对Process 0 来说：flag[0]=true, turn=1
- ✓ 对Process 1 来说：flag[1]=true, turn=0
- ✓ 上述条件无法同时满足

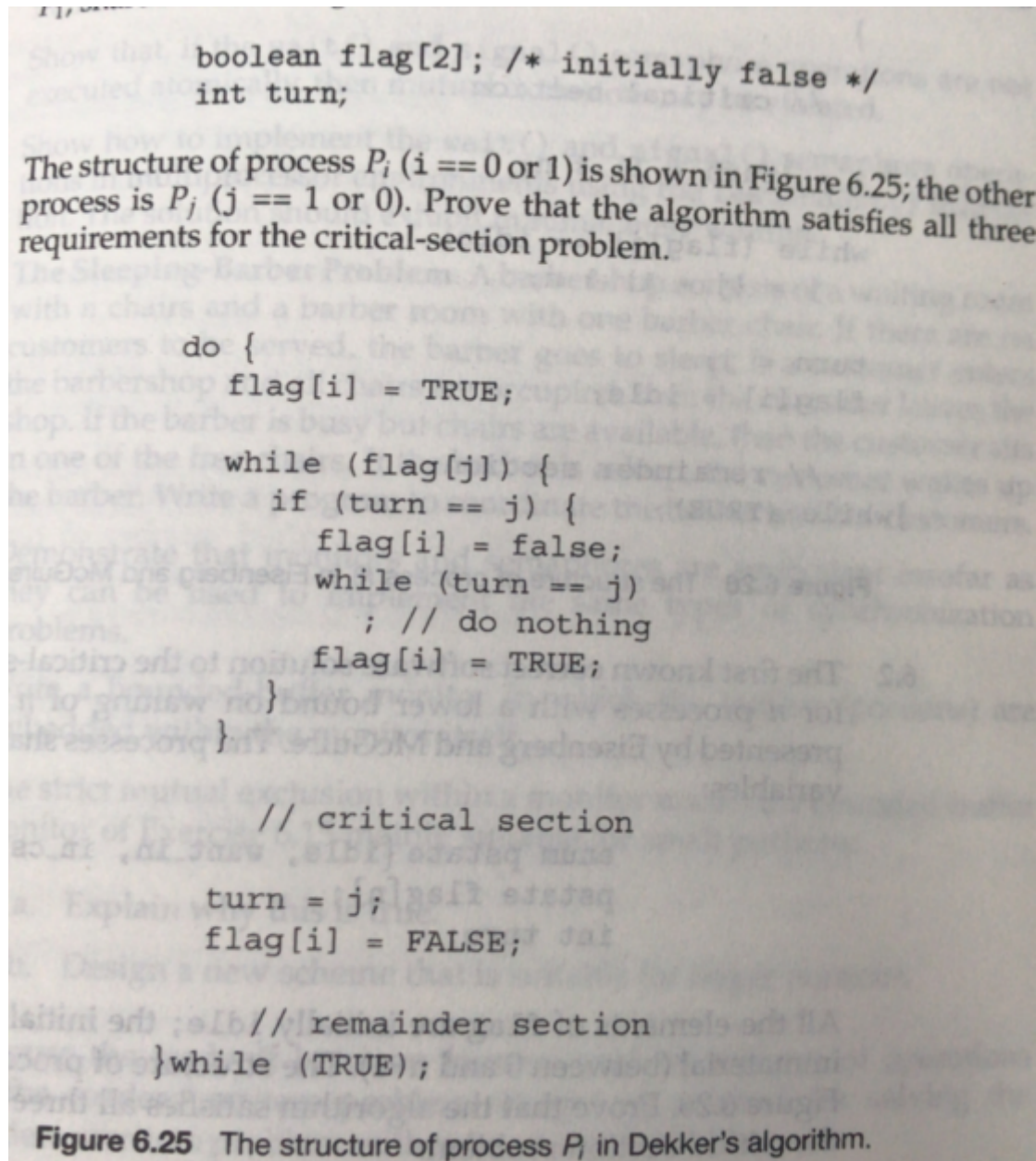
#### 条件2: Progress (前进)

- ✓ 如果进程 $P_1$ 不在临界区，则，有以下可能：
- ✓  $P_1$ 在进入区：turn=0 →  $P_0$ 能进入临界区
- ✓  $P_1$ 在退出区：flag[1]=false →  $P_0$ 能进入临界区
- ✓  $P_1$ 在剩余区：flag[1]=false →  $P_0$ 能进入临界区

### 条件3. Bounded Waiting (有限等待)

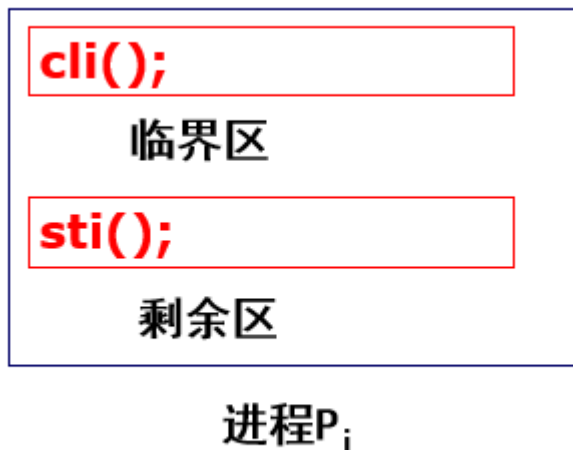
- ✓  $P_0$ 要求进入,  $P_0$ 必然先来到进入区, 则  $flag[0]=true$ ;
- ✓  $P_1$ 在临界区  $\rightarrow turn=0$ ,  $P_1$ 尽管会修改  $flag[1]$  的值, 但是不再改变  $turn$  值, 导致后面的  $P_1$  会进入循环,  $P_0$ 进入。  $P_0$ 最多在  $P_1$  进入临界区一次后就能进入, 反之亦然。

与皮特森算法类似的Dekker 算法



### 禁止中断

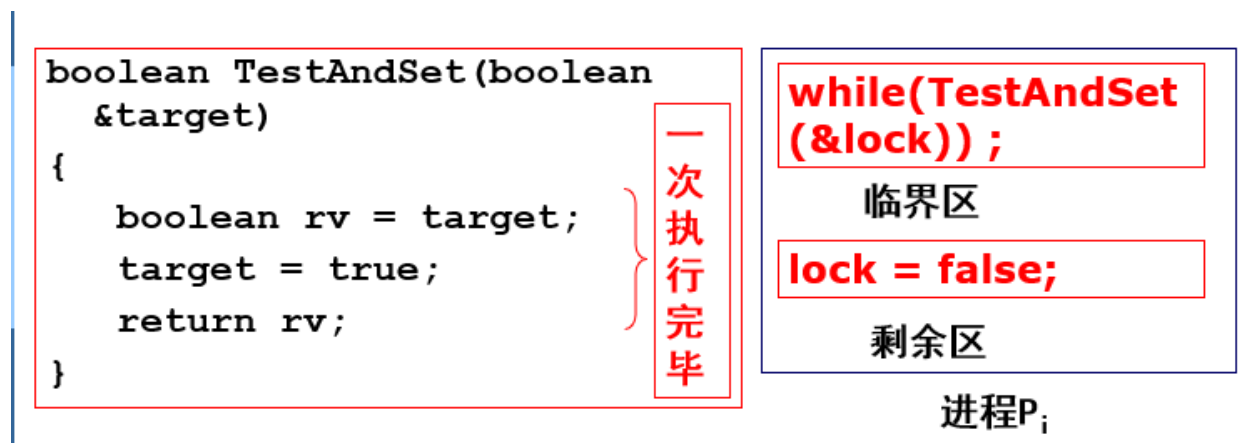
使用中断不允许过程中的调度



虽然简单，但是代价高，会死机

## 原子指令

test\_and\_set



用户需要查看硬件手册，不方便。

Swap()

```
do {
    key = TRUE
    while (key == TRUE)
        Swap(&lock, &key)
    /* critical section */
    lock = FALSE;
    /* remainder section */
} while (TRUE);
```

声明一个全局布尔变量lock，初始化为false。  
每个进程也有一个局部Boolean变量key。  
Swap后key一直都是true。

## PV信号量

```

struct semaphore
{
    int value; //记录唤醒操作个数
    PCB *queue; //等待在该信号量上的进程
}

P(semaphore s);    //消费唤醒操作 wait()
V(semaphore s);    //产生唤醒操作 signal()

```

P 会使资源数目减1 (Wait)

V 会使资源数加1 (Signal)

PV操作必须成对出现

### 互斥操作

当为互斥操作时候，它们同处于同一进程

```

Semaphore mutex = 1;
count = 5;
P1:  P(mutex);
      R1 = count;
      R1++;
      count = R1;
      V(mutex);

P2 :  P(mutex);
      R2 = count;
      R2++;
      count = R2;
      V(mutex);

```

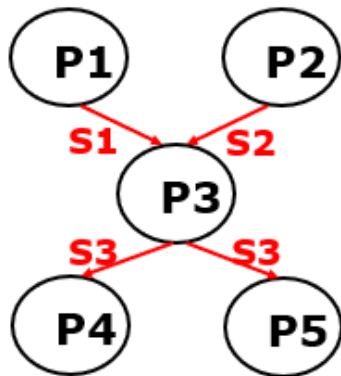
### 同步操作

当为同步操作时，不在同一个进程中出现。

Create a semaphore “**synch**” initialized to 0

```
P1:
    S1;
    V(synch); //signal(synch);
P2:
    P(synch); // wait(synch);
    S2;
```

但是 同步使用的变量从0开始，先进行的释放V，后进行的任务先申请。



```
semaphore S1, S2, S3;
S1=0; S2=0; S3=0;
```

```
P1: ...
    V(S1);
```

```
P2: ...
    V(S2);
```

```
P3: P(S1); P(S2);
    ...V(S3);
```

```
P4:
    P(S3); ...
```

```
P5:
    P(S3); ...
```

## 金典问题

生产者消费者



$n$  buffers, each can hold one item

Semaphore **mutex** initialized to the value 1

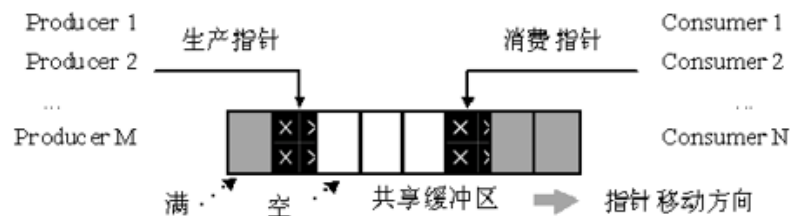
- either producer write or consumer read

Semaphore **full** initialized to the value 0

Semaphore **empty** initialized to the value  $n$

要求:

- ✓ 生产者和消费者不能同时对buffer进行操作, 也就是说某时刻要么生产者在写buffer, 要么消费者在读buffer
- ✓ 当buffer满的时候, 生产者不能继续写
- ✓ 当buffer空的时候, 消费者不能继续读



Producer

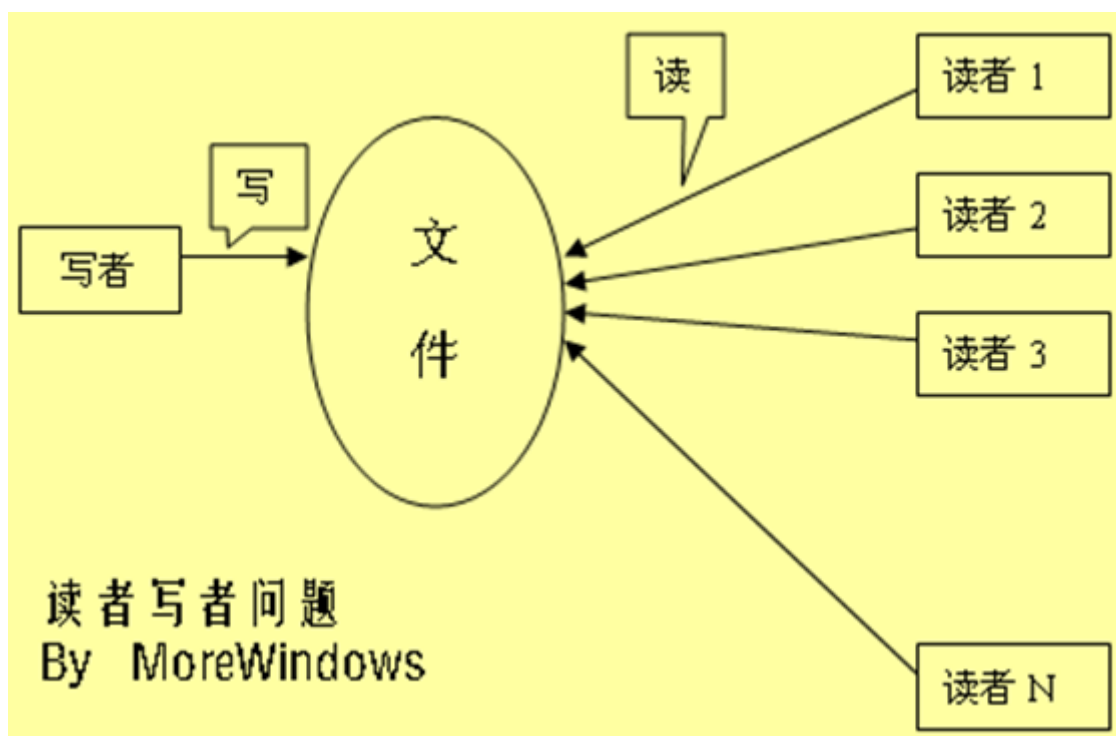
P(empty);
P(mutex); //进入区
one unit $\rightarrow$ buffer;
V(mutex);
V(full); //退出区

Consumer

P(full);
P(mutex); //进入区
one unit $\leftarrow$ buffer;
V(mutex);
V(empty); //退出区

## 读者-写者

1. 允许多个读者同时执行读操作。
2. 不允许读者、写着同时操作。
3. 不允许多个写者同时操作。



写者互斥



```

do {
    wait(rw_mutex);

    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);

```

读者，读的过程不能写

```

do {
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);

    ...
    /* reading is performed */
    ...

    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
} while (true);

```

为了防止多个写者对count的读写造成问题

```

do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);

    signal(mutex);

    ...
    /* reading is performed */
    ...

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);


    signal(mutex);
} while (true);

```



## 哲学家进餐

```
do {  
    wait (chopstick[i] );  
    wait (chopstick[ (i + 1) % 5] );  
    // eat  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
    // think  
} while (TRUE);
```



## 第七章 死锁

进程竞争资源，有可能形成循环竞争，造成死锁。

多个进程因循环等待资源而造成无法执行的现象。

死锁监测可能已经处于死锁。

监测加恢复

### 死锁特征

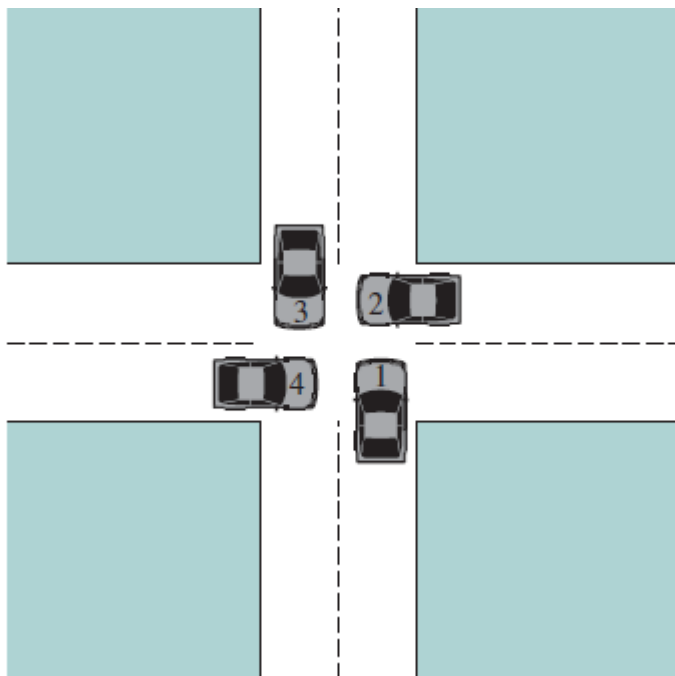
**占有并等待**，一个进程必须占有至少一个资源，并等待另一个资源，而该资源为其他程序占有。

**互斥**，至少有一个资源必须处于非共享模式，即一次只有一个进程使用。如果另一个进程申请该资源，那么申请进程必须等到该资源被释放为止。

**非抢占**，资源不能被抢占，即资源只能在进程完成任务后自动释放。

**循环等待**，有一组等待进程  $\{P_0, P_1, \dots, P_n\}$ ， $P_0$ 等待的资源为 $P_1$ 所占有， $P_1$ 等待的资源为 $P_2$ 所占有， $\dots$  $P_{n-1}$ 等待的资源为 $P_n$ 所占有， $P_n$ 等待的资源为 $P_0$ 所占有。

### 实例解释1



解释为什么发生了死锁？

1. 道路是互斥使用的，一辆车使用了另一辆就不能使用。
2. 十字路口上的车占有了车道，并且等待另一条路面的车道。
3. 这个情况下车辆无法抢占其他车道。
4. 如图所示，几辆车都在相互等待对方让出车道。

如何避免死锁：  
十字路只能右转

## 实例解释2

- 某系统中有11台打印机，N个进程共享打印机资源，每个进程要求3台。当N的取值不超过( )时，系统不会发生死锁。  
A. 4； B. 5； C. 6； D. 7

不发生死锁的资源数目  $\geq (\text{每个进程需要的资源数目} - 1) * \text{进程数目} + 1$

正好多出的哪一个资源能满足其中一个进程的需求

## 资源分配图

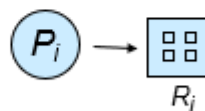
- Process



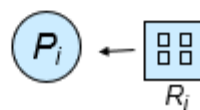
- Resource Type with 4 instances (实例)



- $P_i$  requests instance of  $R_j$



- $P_i$  is holding an instance of  $R_j$



分配图中没有环，那么系统就没有死锁。

虽然有环但是资源有多个，也不一定造成死锁。

## 死锁预防

破坏死锁的4个条件之一

这种破坏是不合理的

## 死锁避免

保证系统处于安全状态。

安全状态定义：如果系统中的所有进程存在一个可完成的执行序列，则称系统处于安全状态。  
使用银行家算法找安全序列。

## 银行家算法

		<u>Allocation</u>			<u>Max</u>			<u>Need</u>			<u>Available</u>		
		A	B	C	A	B	C	A	B	C	A	B	C
	$P_0$	0	1	0	7	5	3	7	4	3	3	3	2
	$P_1$	2	0	0	3	2	2	1	2	2			
	$P_2$	3	0	2	9	0	2	6	0	0			
	$P_3$	2	1	1	2	2	2	0	1	1			
	$P_4$	0	0	2	4	3	3	4	3	1			

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>			<u>Work</u>		
	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3	3	3	2
P1	2	0	0	3	2	2				1	2	2	5	3	2
P2	3	0	2	9	0	2				6	0	0	7	4	3
P3	2	1	1	2	2	2				0	1	1	7	4	5
P4	0	0	2	4	3	3				4	3	1	10	4	7

在进程执行过程中发生了资源请求

1. request 小于 need
2. request 小于 Available

效率:  $n * m^2$

Check that  $\text{Request} \leq \text{Need}$ ,  $\text{Request} \leq \text{Available}$ , that is,  $P1 (1,0,2) \leq (3,3,2) \Rightarrow \text{true}$

Pretend to allocate the resources the P1

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	4	3	2	3	0
$P_1$	3	0	2	0	2	0	0	2	2
$P_2$	3	0	2	6	0	0			
$P_3$	2	1	1	0	1	1			
$P_4$	0	0	2	4	3	1			

- Available = Available - Request
- Allocation = Allocation + Request
- Need = Need - Request

死锁监测和恢复，同样采用银行家算法。

## 第八章 内存

### 重定位bind

地址绑定（地址重定位），使用基地址和界限地址得到真正的物理地址。

源程序中地址通常是符号形式，经过编译之后，“名字空间符号”被映射为一个“可重定位空间”的地址。Linker或loader将可重定位地址映射为绝对地址。

制定与数据绑定到内存地址情况：

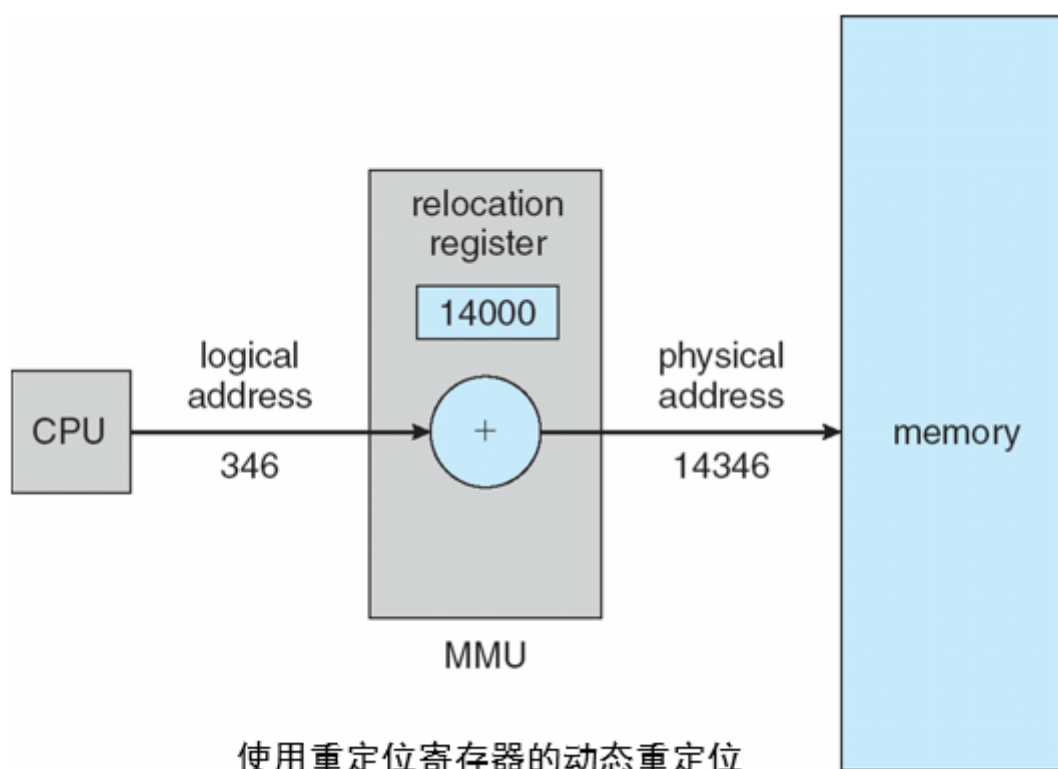
编译时：如果编译时就知道进程在内存中的驻留地址，那么就可以生成绝对代码。

加载时：如果编译时不知道进程将驻留在内存的什么地方，那么编译器就必须生成可重定位代码。

执行时：如果进程执行时可以从一个内存段移动到另一个内存段，那么绑定必须延迟到执行时才进行。

重定位最合适的时机：运行时重定位（基地址加偏移量）

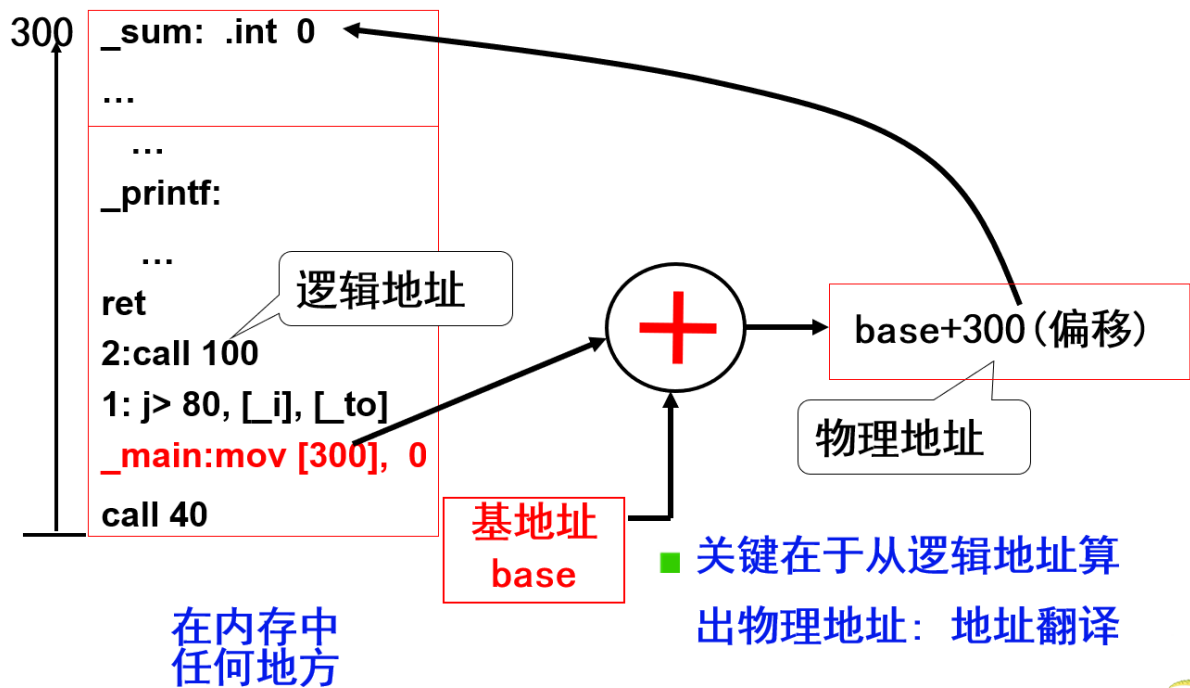
## MMU



内存管理单元

作用：将虚拟地址送往总线之前转换成物理地址

地址绑定在运行时重定位，每次都需要根据基地址和偏移地址计算物理地址，这一个过程采用硬件的方式速度更快。



## 动态加载、动态链接

### 动态加载

一个子程序只有在调用时才被加载。所有子程序都可以以重定位的方式保存在磁盘上。

优点：不需要的子程序不会被加载。动态加载不需要操作系统提供特别支持。

### 动态链接

进程即将调用到的代码段，不被预先连接到程序，只有到真正被调用的时刻才连接需要系统提供“动态链接库”的支持。

(将链接延迟到运行时)

静态链接就是在链接时候就完成链接。

## 内存分配

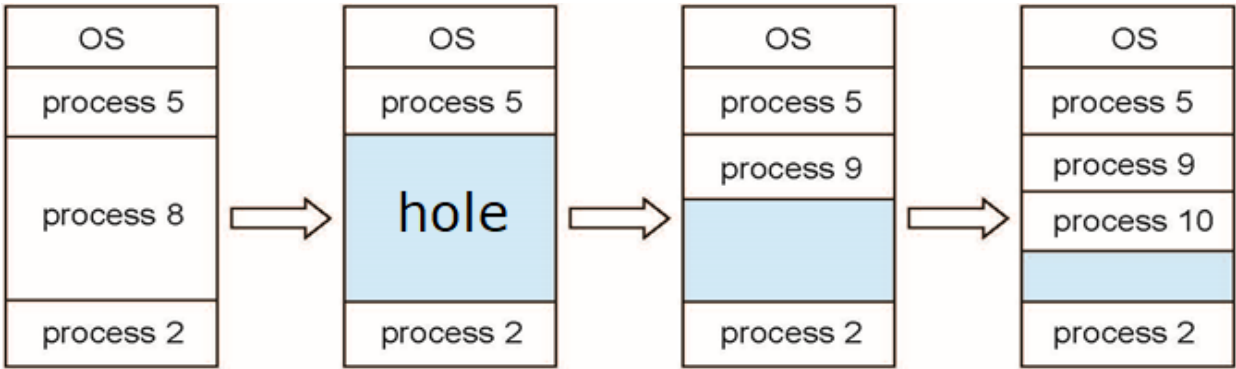
### 连续地址分配（动态内存分配）

通常需要将多个进程同时放在内存中，因此需要考虑如何为输入队列中需要调入内存的进程分配内存空间。采用连续内存分配时，每个进程位于一个连续的内存区域。

### 多重分区分配

将内存分为多个固定大小的分区，每个分区只能容纳一个进程。

在**可变分区**方案里，操作系统有一个表，用于记录哪些内存可用，哪些已经被占用。



### 首次适应

分配第一个足够大的孔。查找可以从头开始，也可以从上次首次适应结束时开始。一旦找到足够大的空闲孔，就可以停止。

### 最佳适应

分配最小的足够大的孔。必须查找整个列表，除非列表按大小排序。这种方法可以产生最小剩余孔。

### 最差适应

分配最大的孔。同样，必须查找整个列表，除非列表按大小排序。这种方法可以产生最大剩余孔，该孔可能比最佳适应方法产生的较小剩余孔更为有用。

### 碎片

#### 外部碎片

内存块加起来能满足一个请求，但是由于不连续（中间有断层），不能用来连续分配

#### 内部碎片

分出去的分区略大于请求的内存长度。这个余下的小内存块属于该分区，但是无法利用

解决碎片问题，紧缩。

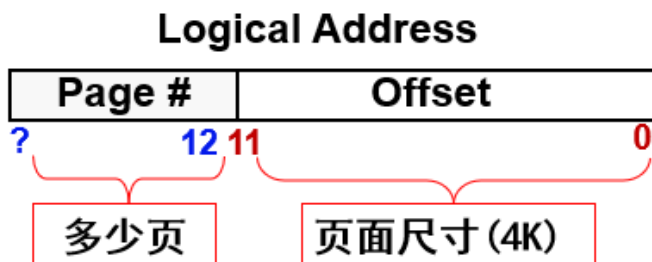
### 不连续分页

实现分页的基本方法涉及将物理内存分为固定大小的块，称为帧（frame）

将逻辑内存也分为同样大小的块，称为页（page）



## ■ 分页依靠页表结构



**进程页表**

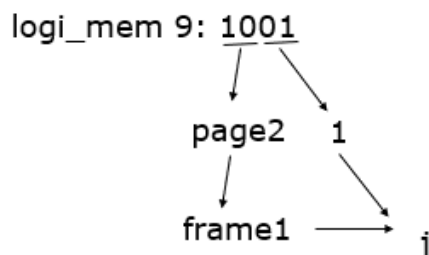
页号	页框(帧)号	保护
0	5	R
1	1	R/W
2	3	R
3	7	R

## Physical Address



$n=2$  and  $m=4$

页大小为4B, 物理内存为32B (8帧)



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

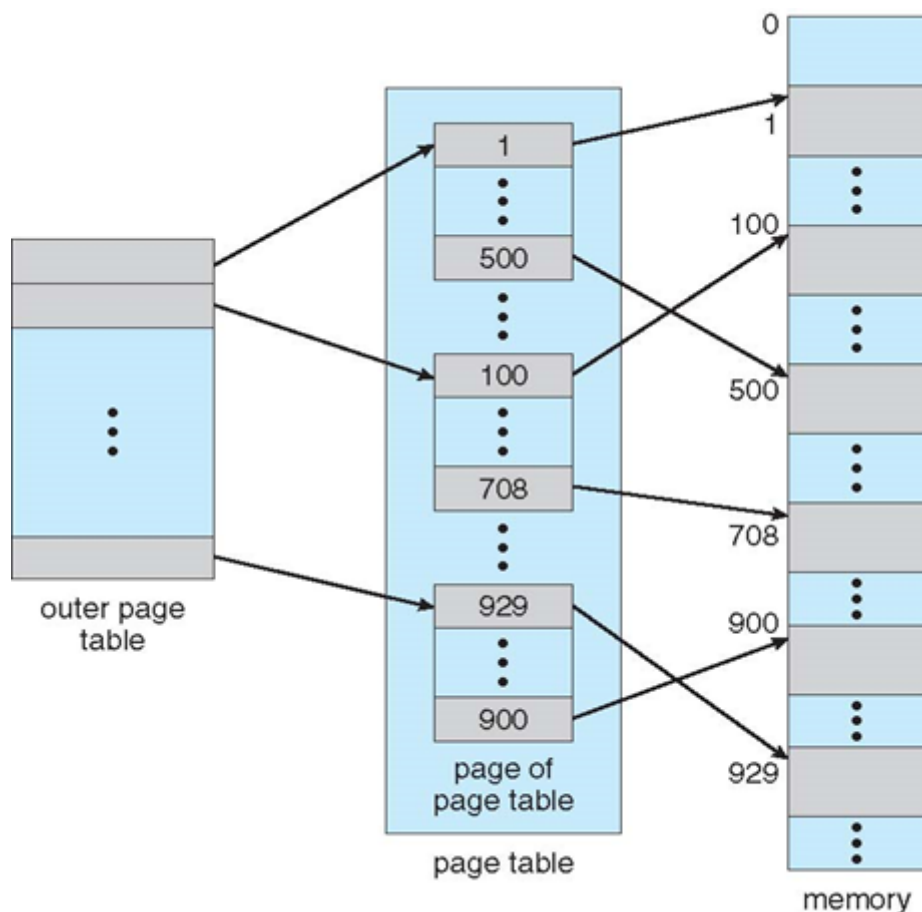
page number	page offset
p	d
$m - n$	$n$

0	
4	i
8	m
12	n
16	
20	a
24	b
28	c
	d
	e
	f
	g
	h

页表存在内部碎片

## 页表很大的时候

采用层次页面/多级页表



但是地址翻译效率很低，要提高效率。

■1级页表访存2次，速度下降50%

■2级页表访存3次，速度下降到33%

■3级页表访存4次，速度下降到25%

二级页表

页目录号 (10 位)	页表索引 (10 位)	页内偏移量 (12 位)
-------------	-------------	--------------

## TLB有效访问时间

寄存器制作，可以并行查询。

TLB是一组相联快速内存：快表

有效访问时间

$$\text{有效访问时间} = \text{HitR} \times (\text{TLB} + \text{MA}) + (1 - \text{HitR}) \times (\text{TLB} + 2\text{MA})$$

命中率!

内存访问时间!

TLB时间!

$$\text{有效访问时间} = 80\% \times (20\text{ns} + 100\text{ns}) + 20\% \times (20\text{ns} + 200\text{ns}) = 144\text{ns}$$

$$\text{有效访问时间} = 98\% \times (20\text{ns} + 100\text{ns}) + 2\% \times (20\text{ns} + 200\text{ns}) = 122\text{ns}$$

慢了22%!

■ TLB要想发挥作用，命中率应尽量高

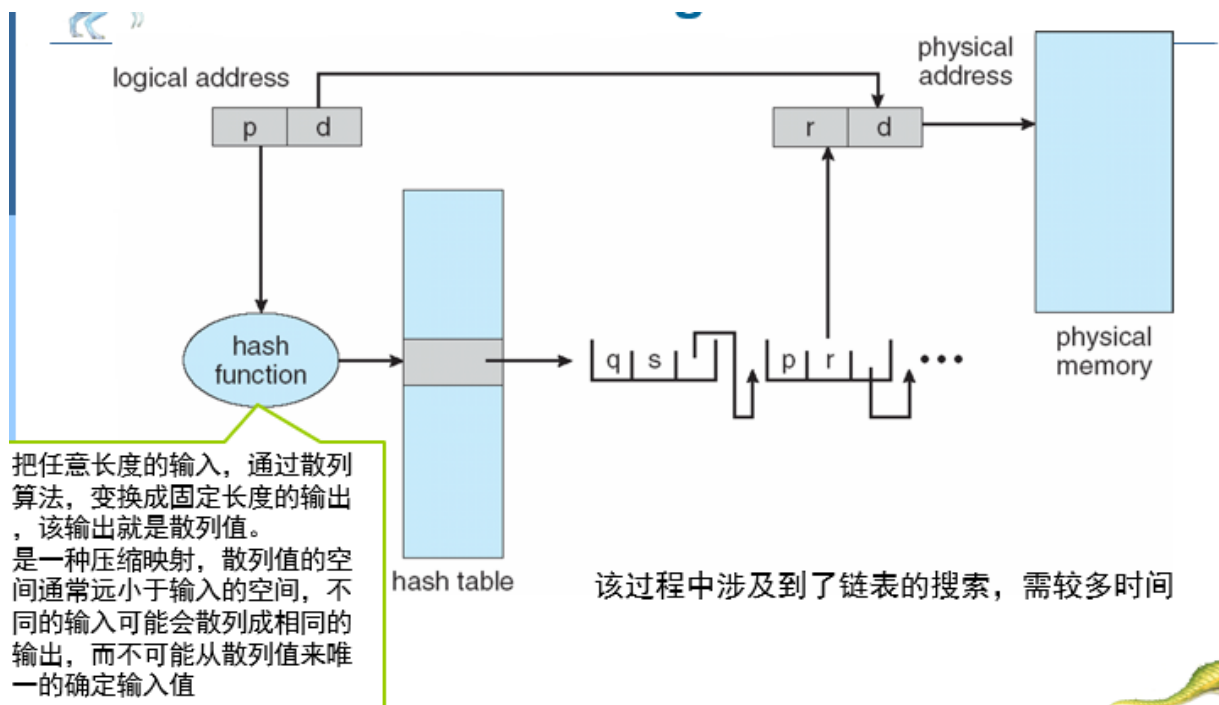
## 哈希页表

虚拟地址的页号通过哈希函数转换后，指向页表中某个页表项。

哈希函数值相同的虚拟页号，指向同一个页表项，它们在那个页表项下组成一个链表。

地址翻译时，由虚拟页号哈希后锁定对应链表，搜索虚拟页号的匹配项

如果找到匹配，则找到了虚拟页号对应的物理页帧。

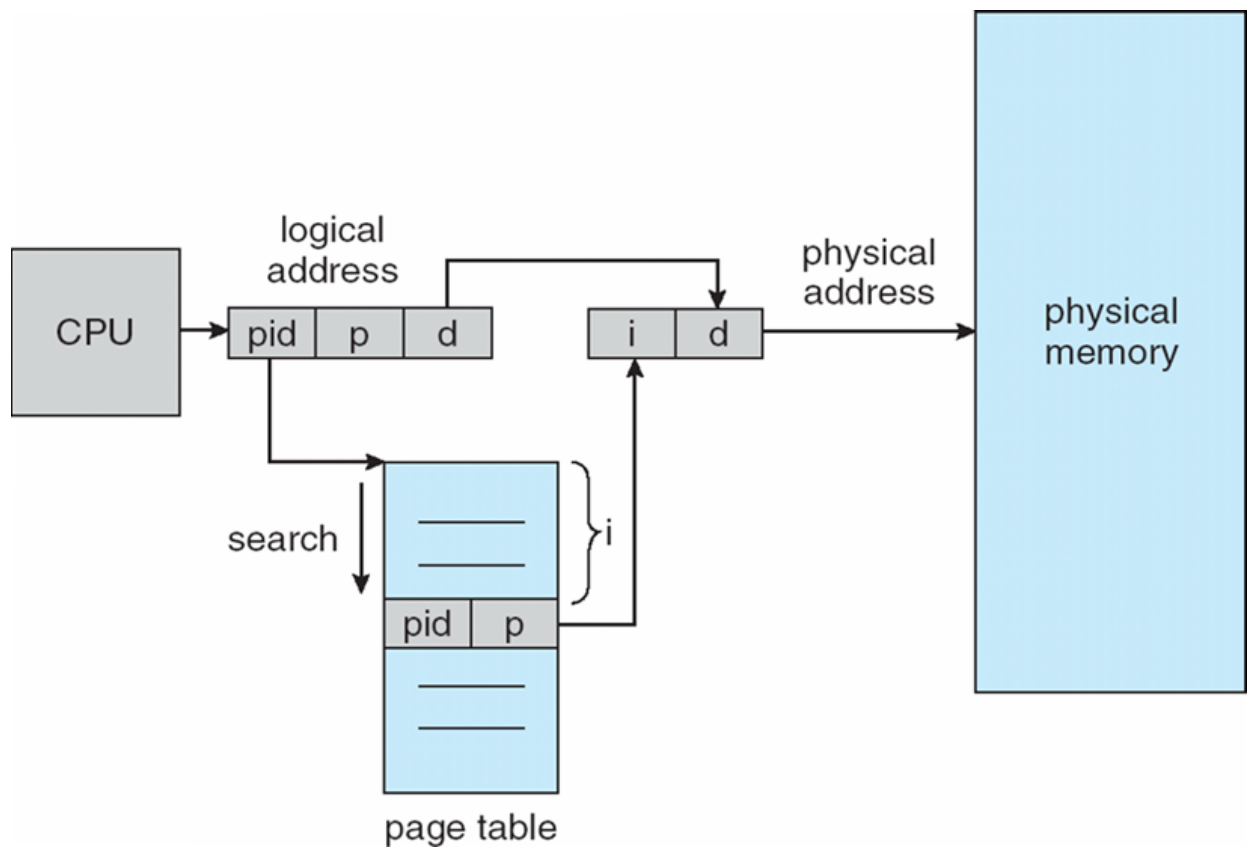


## 反向页表

正向页表，一个进程一个页表，PCB有指针指向他。

为每一个物理块设置一个也表项并按物理块号排序，其中的内容则是页号及属于的进程的标志符。

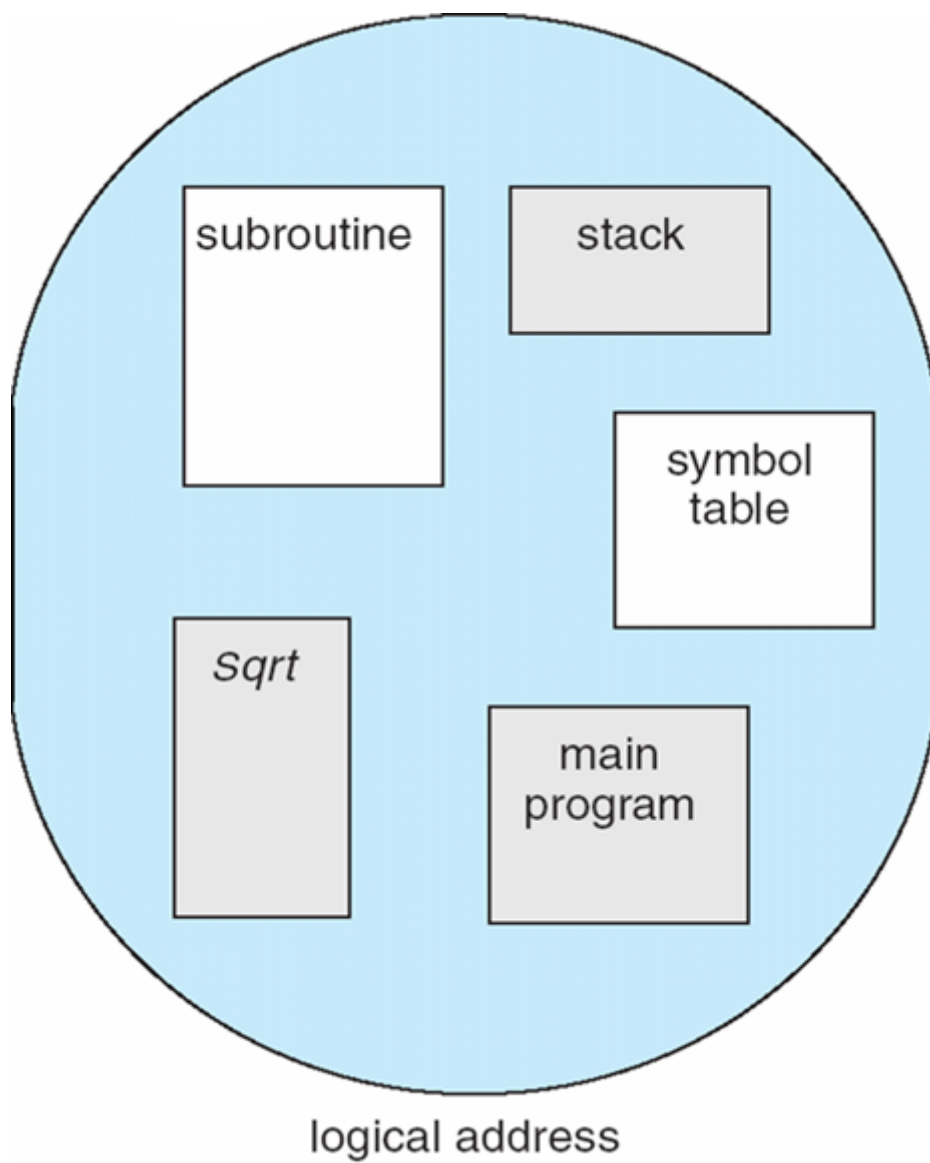
整个系统只有一个页表，对每个物理内存的页只有一条相应的条目。



空间小，但是查找时间

## 分段 和段页结合

用户视角的内存管理



把每一段代码和数据看做是“段”的单元

逻辑地址空间是由一组段组成。每个段都有名称和长度。地址指定了段名称和段内偏移。因此用户通过两个量来指定地址：段名称和偏移。为了实现简单，段是编号的，是通过段号而不是段名来引用的。

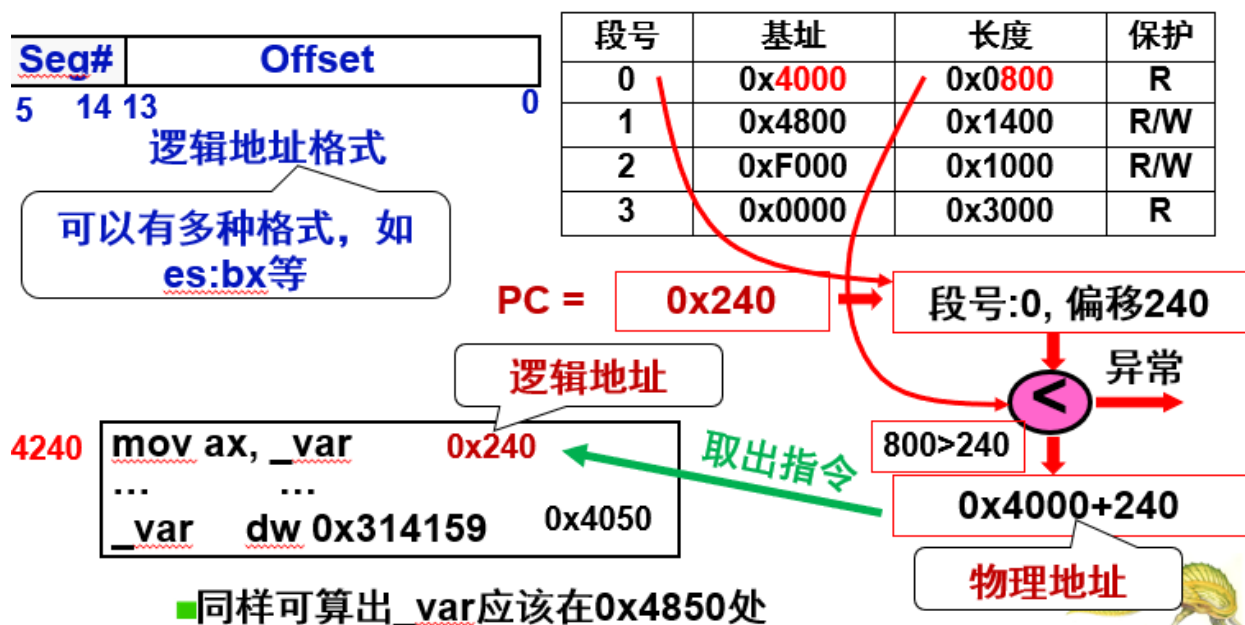
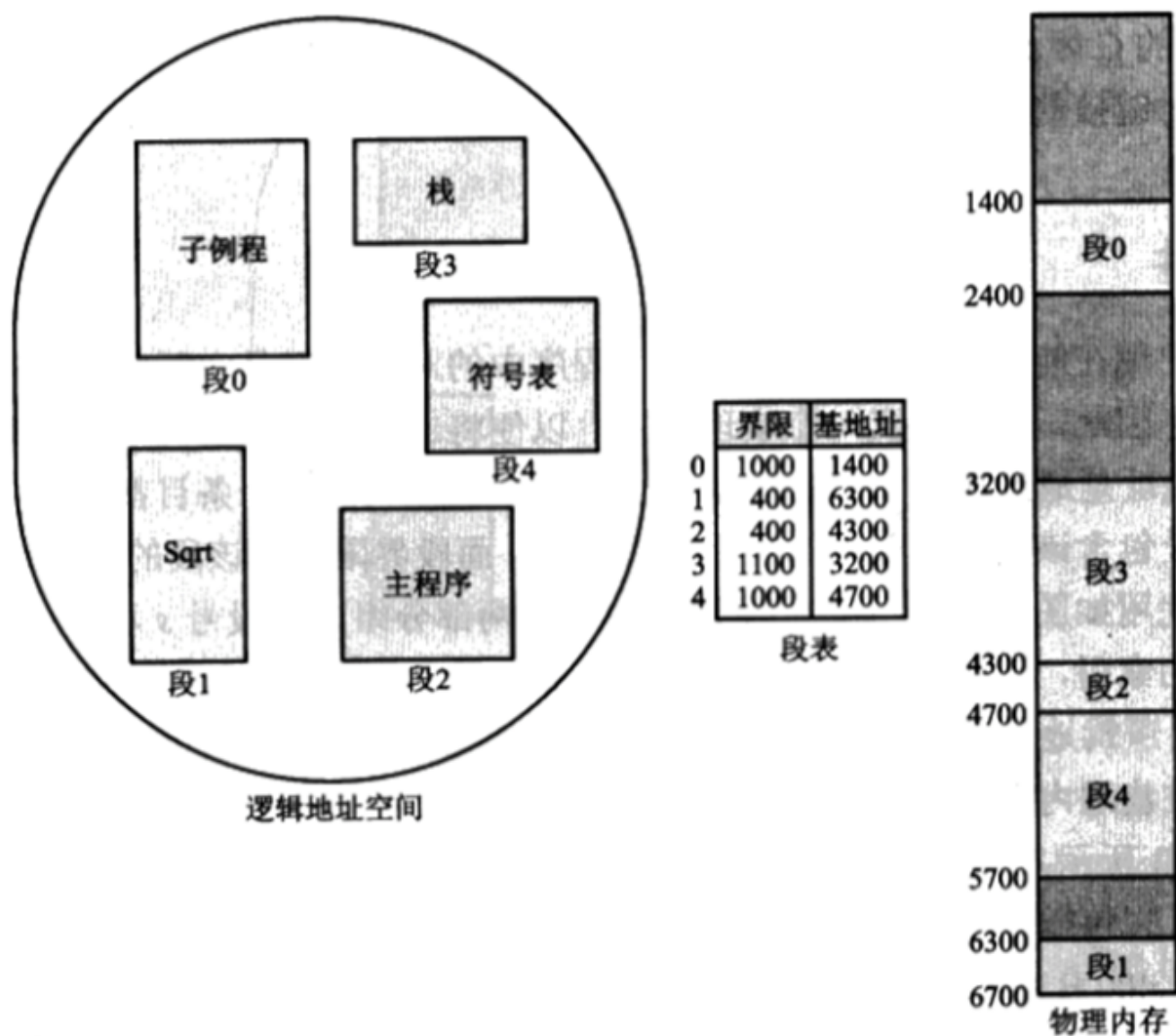
## 段页结合

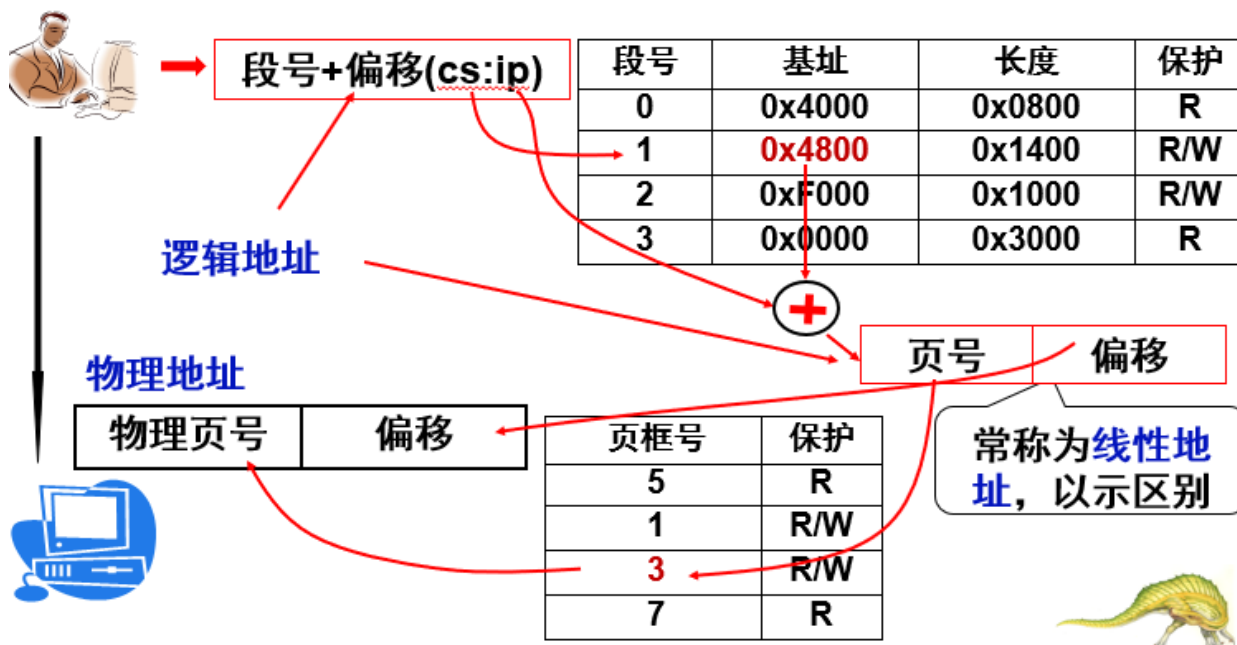
结构：

段号：段内偏移量

段表：以段号为索引映射到物理内存地址

段号：界限：基地址





首先通过段号找到基地址，基地址 + 偏移 获得 虚拟地址

从 虚拟地址中取出 页号 查页表或得到帧号，再从页地址中取出偏移量 + 帧号

优点：符合程序员习惯，并可高效利用内存

## 第九章 虚拟内存

会产生缺页现象，缺页触发中断，中断处理完成页面调入。

调入页面需要一个空页框，如果没有需要置换。

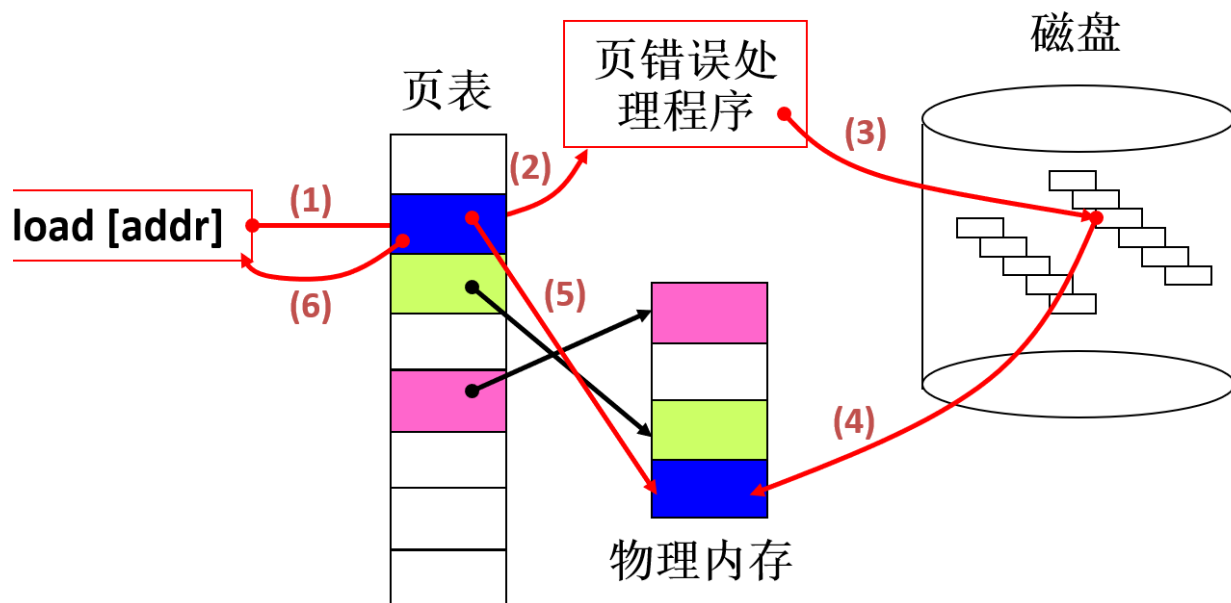
置换方法 FIFO、OPT、LRU

### 优点

可编写比内存大的程序，使用一个大地址空间（虚拟内存）

把部分程序放入内存中，部分放磁盘，需要的时候调入内存，内存利用率高。

### 请求调页过程



发起请求一个地址，MMU查表发现这个地址在内存中是不存在的，触发页错误处理程序（中断），中断处理程序到磁盘中找到相应的页，找到之后找到一个空闲的物理页框，如果没有空闲物理页框，那么使用页面置换，把页面换入内存，修改页表，建立连接，CPU继续从中断之后执行。

## 页面置换（淘汰）

选择一个空闲页框

中断率 = 缺页次数 / 内存调页次数

### FIFO

■ 一实例：给进程分配了3个页框(frame)，页面引用序列为

**A B C A B D A D B C B**

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

FIFO会导致Belady异常，分配的页框变多了但是缺页次数却增多了。

### OPT(MIN)



将最远将使用的淘汰换出。

**MIN**算法: 选**最远将使用的**页淘汰。是一种最优的方案，可以证明缺页数最小!

■ 继续上面的实例: (3frame)**A B C A B D A D B C B**

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A									C	
2		B									
3			C			D					

## LRU

选最近最长一段时间没有使用的页面淘汰换出。

用过去的历史预测将来。**LRU**算法: 选**最近最长一段时间没有使用的**页淘汰(最近最少使用)。

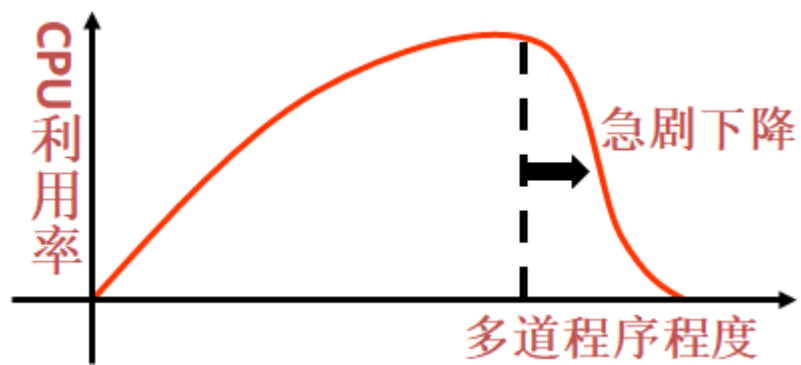
■ 继续上面的实例: (3frame)**A B C A B D A D B C B**

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A									C	
2		B									
3			C			D					

精准实现代价太大。

## 换页产生的现象

颠簸



抖动

页面刚换入然后马上又被换出，这就称作抖动。