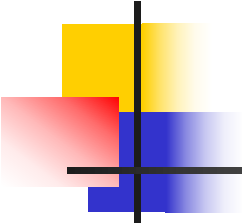


# 操作系统

# Operating Systems

综合版

Department of Computer Science & Technology  
University of Science & Technology Beijing  
2022



---

总学时：64学时

讲授：48学时，1-8周

实验：16学时，7、8、12、13周

分2组，每组2个班

### **最终成绩的构成：**

平时成绩占30%（实验20%，考勤（课堂练习）、作业等10%）

期末考试（非集中考试）成绩占70%



# 教材与参考书

---

- [1] 张尧学, 宋虹, 张高. 计算机操作系统教程 (第4版). 清华大学出版社, 2013 (**教材**)
- [2] Andrew S. Tanenbaum, Herbert Bos. 陈向群, 马洪兵等译. 现代操作系统 (原书第4版). 机械工业出版社, 2017
- [3] 汤子瀛, 哲凤屏, 汤小丹. 计算机操作系统. 西安电子科技大学出版社, 1996
- [4] William Stallings. 陈向群, 陈渝等译. 操作系统 - 精髓与设计原理 (第八版). 电子工业出版社, 2017
- [5] Randal E. Bryant, David R. O'Hallaron. 龚奕利, 贺莲译. 深入理解计算机系统 (原书第3版). 机械工业出版社, 2016
- [6] 邹恒明. 计算机的心智: 操作系统之哲学原理. 机械工业出版社, 2009
- [7] Abraham Silberschatz, Peter Galvin, Greg Gagne. Applied Operating System Concepts, John Wiley & Sons, Inc., 2000
- [8] 孟庆昌, 牛欣源, 张志华, 路旭强. Linux教程 (第5版). 电子工业出版社, 2019



# 讲授的主要内容

---

第1章 操作系统概述

第2章 进程管理

第3章 进程调度

第4章 内存管理

第5章 进程与内存管理实例

第6章 文件系统

第7章 I/O设备管理

第8章 Linux文件系统

第9章 死锁

**学时安排：**进程管理（多） 内存管理 文件系统 设备管理（少）



# 第1章 操作系统概述

---

**1.1 认识操作系统**

**1.2 操作系统发展过程中形成的一些概念**

**1.3 OS对运行环境的要求**

**1.4 典型OS实例**

**1.5 现代操作系统的基本特征**

**1.6 从不同角度认识操作系统**

**1.7 学习操作系统课程要达到的目标**

# 1.1 认识操作系统

## 1. 程序是如何执行的?

```
#include <stdio.h>

void swap();

int buf[2] = {1, 2};

int main()
{
    swap();
    printf("buf = %d, %d\n", buf[0], buf[1]);
}
```

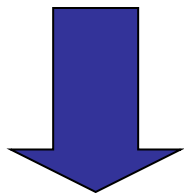
a) main.c文件

```
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
    int temp;

    *bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

b) swap.c文件



预处理  
编译  
链接

**可执行程序 (机器代码)**

# 1.1 认识操作系统

**可执行程序（示意）：** 机器指令

080483b4 <main>:

80483b4: 55  
80483b5: 89 e5  
80483b7: 83 e4 f0  
80483ba: e8 09 00 00 00

...

080483c8 <swap>:

80483c8: 55  
80483c9: 8b 15 5c 94 04 08

...

08049454 <buf>:

8049454: 01 00 00 00 02 00 00 00

0804945c <bufp0>

804945c: 54 94 04 08

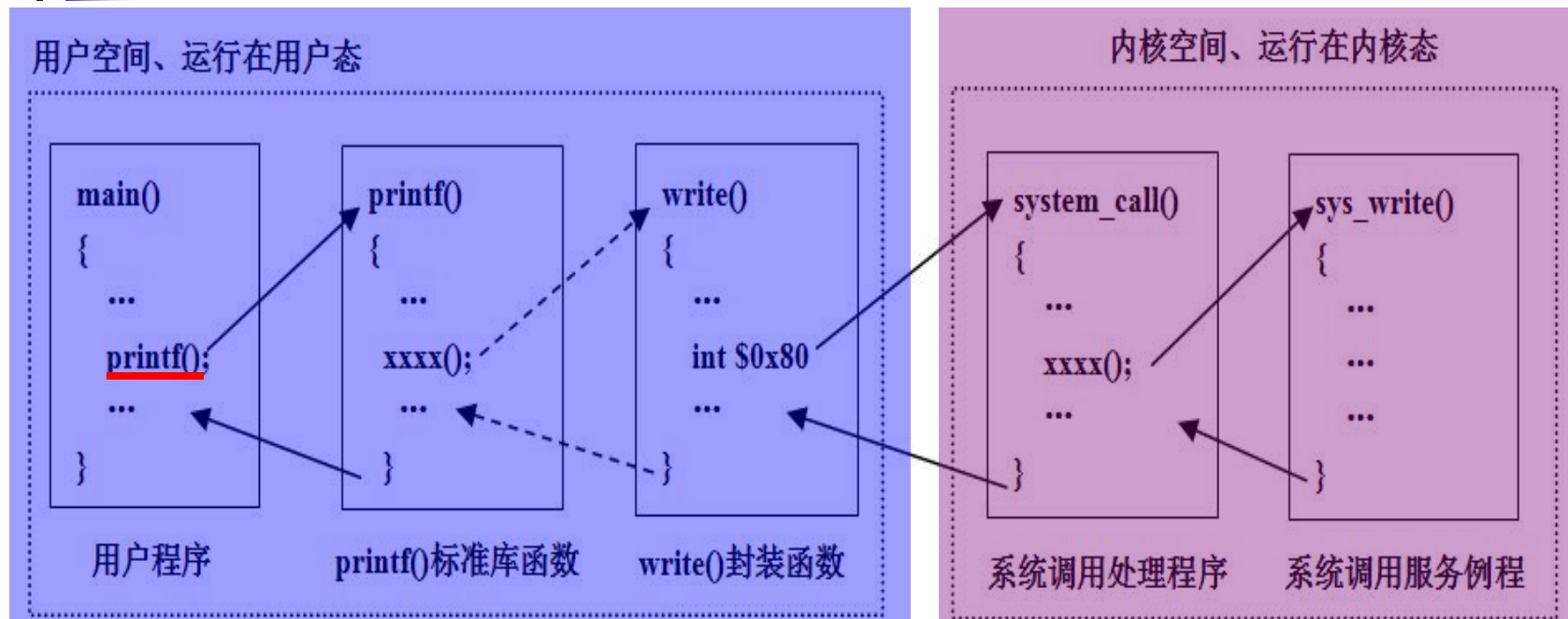
push %ebp  
mov %esp, %ebp  
and \$0xffffffff0, %esp  
call 80483c8<swap>

汇编语言指令

push %ebp  
mov 0x804945c, %edx

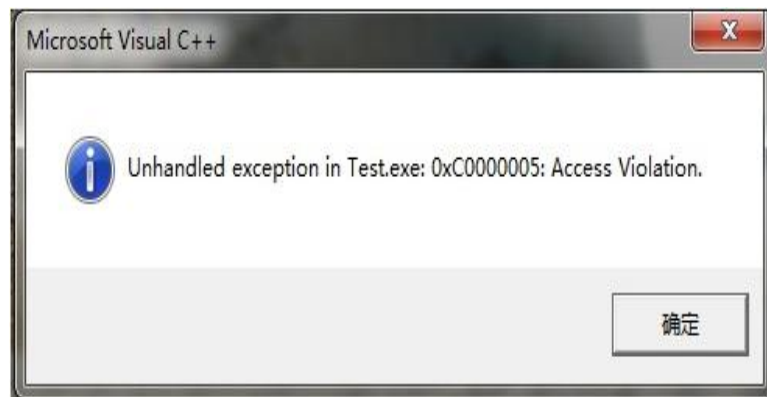
数据

# 1.1 认识操作系统



## 程序执行的整个生命周期都受操作系统控制:

- ✓ 建立程序的内存映像, CPU的分配
- ✓ 内存分配、释放
- ✓ 文件访问, 输入输出 (I/O)
- ✓ 中断 (异常) 处理

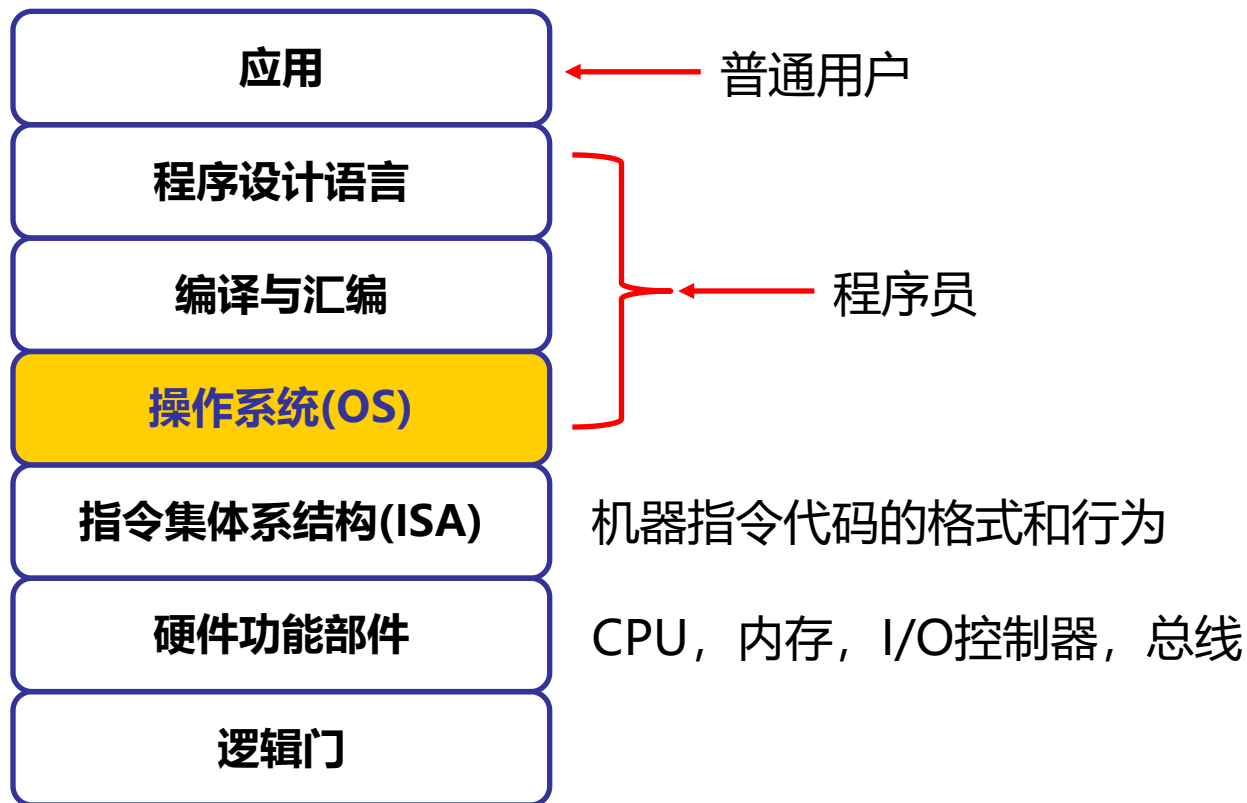




# 1.1 认识操作系统

## 2. 什么是操作系统(Operating System, OS)?

### 1) OS在计算机系统的位置





# 什么是操作系统(Operating System, OS)

---

## 2) OS以什么形式出现?

是一组程序。

OS与普通程序有何区别?



# 什么是操作系统(Operating System, OS)

---

## 3) OS的作用 (功能)

### (1) 一个虚拟机 (Virtual Machine) - 用户观点

让用户（程序员）在使用计算机时不涉及计算机硬件的细节，使硬件细节和用户（程序员）隔离开来，即建立一种简单的高度抽象。

用户与计算机之间的接口。

- ✓ 命令接口（面向普通用户）：命令行，GUI，命令脚本
- ✓ 编程接口（面向程序员）：系统调用，高级语言库函数

如果没有OS，计算机可以使用吗？



# OS的作用

---

## (2) 一个资源管理器：管理系统的软硬件资源 - 系统观点

硬件资源：构成计算机系统所必须配置的所有硬件：

CPU、内存、时钟、磁盘、. . . .

软件资源：程序和数据（文件）。

- ✓ 进程管理：程序的调度；处理机（CPU）的分配等；
- ✓ 内存管理：内存分配、释放与保护；内存扩充等；
- ✓ 文件管理：文件存储空间管理；文件存取；文件访问控制等；
- ✓ I/O设备管理：设备分配；缓冲区管理等。



# 什么是操作系统(Operating System, OS)

---

## 4) OS的定义

OS是硬件之上的**第1层软件**（系统软件）

是一组**程序**，

用来有效控制和**管理**计算机系统的各类**资源**（硬件和软件资源：设备、文件、存储器、CPU、程序（进程）），

以方便用户使用计算机（用户和计算机的**接口**）。



## 1.2 操作系统发展过程中形成的一些概念

---

### 1. 作业(Job)

从输入开始到输出结束，用户要求计算机所做的一次业务处理的全部工作。

作业由顺序的一组作业步组成。

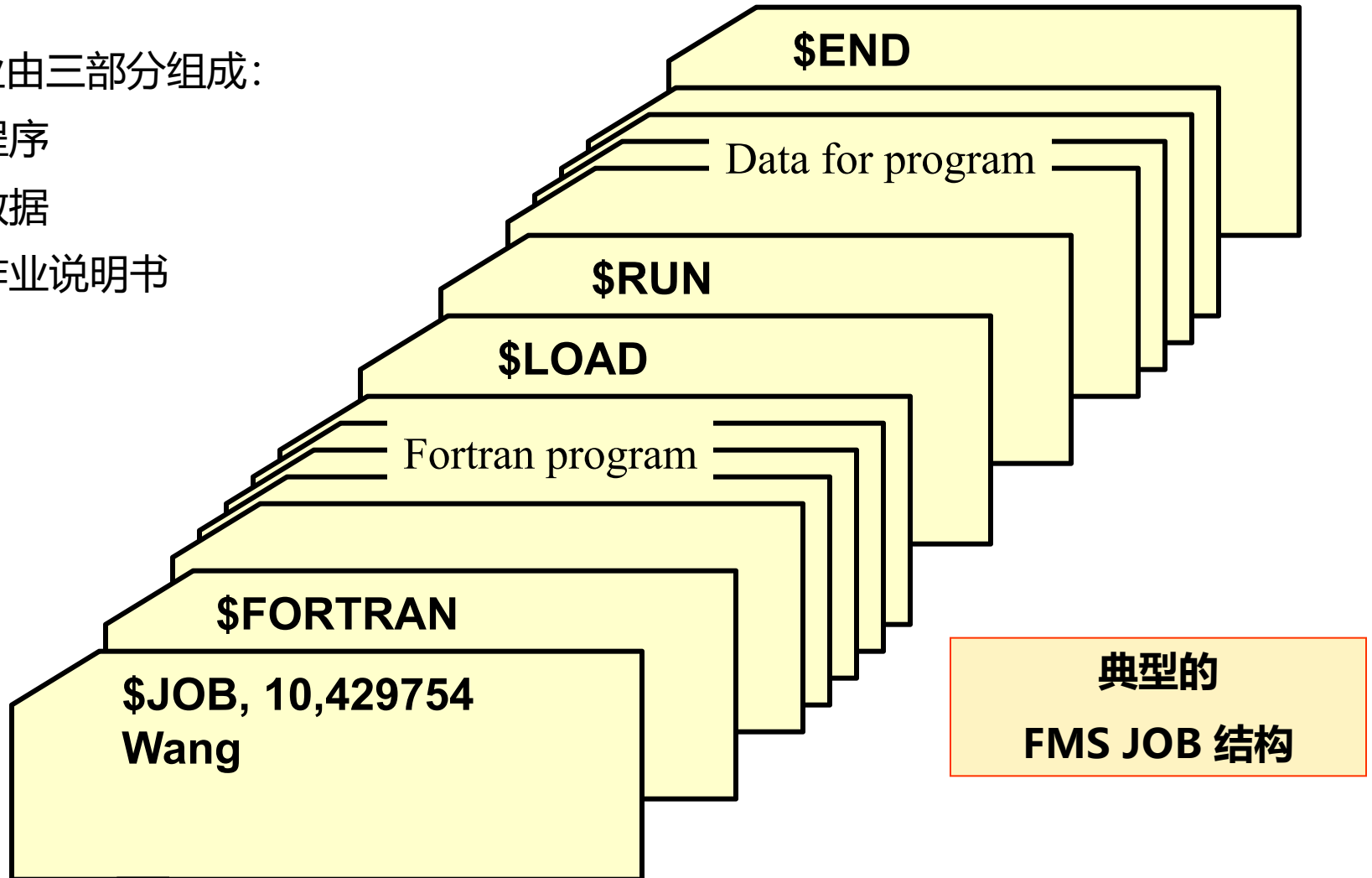
作业的概念来源于批处理系统。

分时系统中一般不存在传统作业的概念。

# 作业

作业由三部分组成：

- ✓ 程序
- ✓ 数据
- ✓ 作业说明书





# 作业

---

运行一个作业的步骤：

- 1) 将程序写在纸上（用高级语言或汇编语言）
- 2) 穿孔成卡片，再将卡片盒交给操作员
- 3) 计算结果从打印机上输出
- 4) 操作员到打印机上撕下运算结果送到输出室
- 5) 程序员稍后可从输出室取到结果
- 6) 操作员从输入室的卡片盒中读入另一个任务
- 7) 如果需要FORTRAN编译器，还要把它取来读入计算机

缺点：机时在走来走去浪费掉





## 1.2 操作系统发展过程中形成的一些概念

### 2. 批处理 (batch)

为改进内存和I/O设备之间的吞吐量

IBM 7094计算机引入了I/O 处理机概念

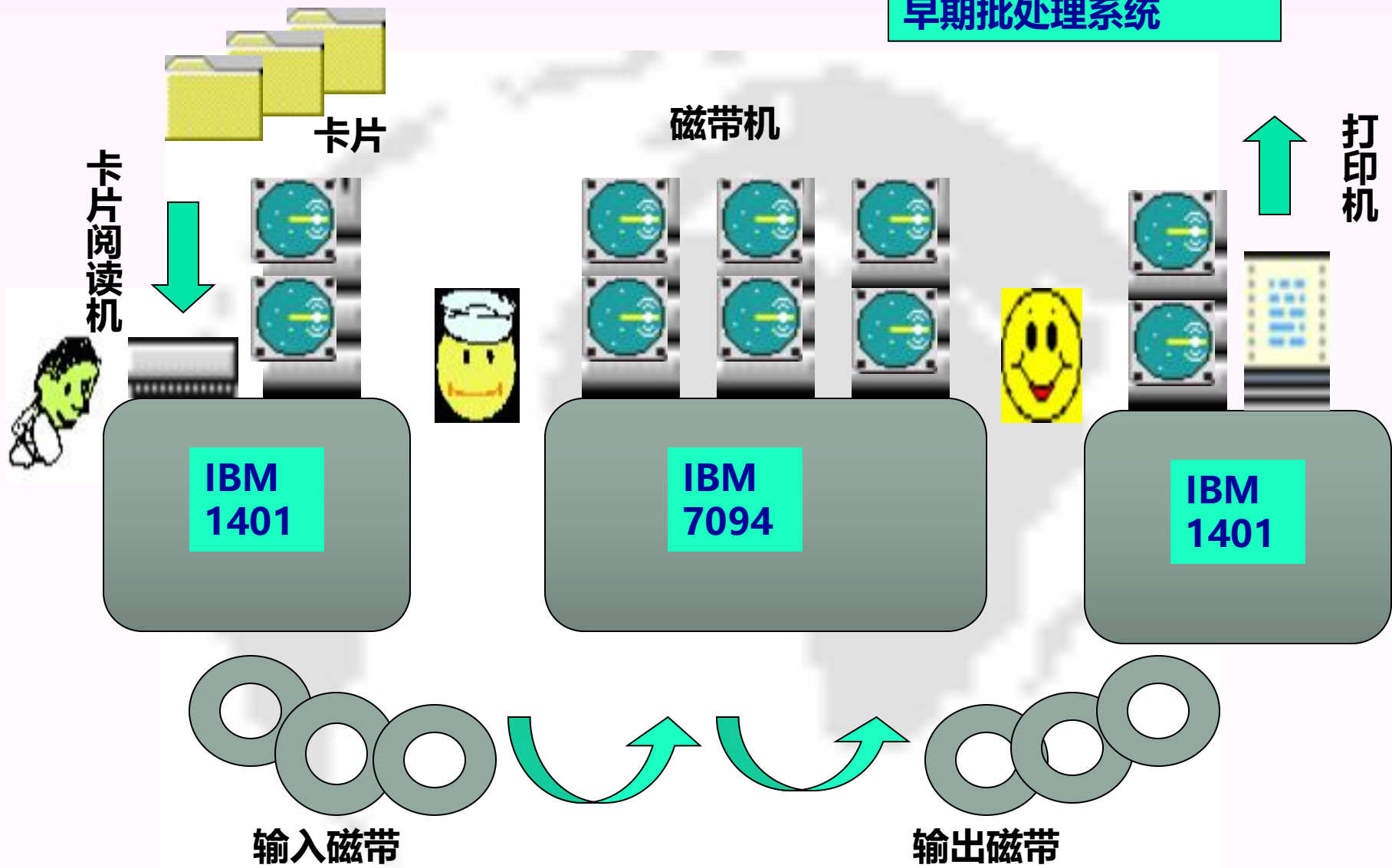
**思想：**

在输入室收集全部的作业，用一台相对便宜的计算机（IBM 1401计算机），  
将作业读到磁带上

再用较昂贵的计算机（IBM7094）完成真正的计算

**一批作业构成一个作业队列，依次处理。**

早期批处理系统





# 批处理

---

批处理的含义：

- ✓ 无交互能力：作业从提交到完成，用户不能与之交互；
- ✓ 从传统的作业 - > 命令文件的扩展

把一系列命令放在一个文件中，称之为命令文件

用文件名作为命令名来执行

批处理命令可以是专门的命令，也可是系统的基本命令；还有有关的控制结构，包括循环、分支、转移等，构成一套特殊的命令语言，可以接受参数，使用变量、宏替换等

缺点？



# 批 处 理

---

## 成批处理:

用户不能干预自己作业的运行

一旦发现作业错误不能及时改正

延长了软件开发时间

一般只适用于成熟的程序或大型的计算程序



## 1.2 操作系统发展过程中形成的一些概念

### 3. 单道程序与多道程序

单道程序：

在内存中只能有一个用户程序（从进入到结束）

若当前程序因等待I/O而暂停，则CPU空闲

对于CPU操作密集的科学计算问题，浪费时间少

对于商业数据处理，I/O等待时间常占80% - 90%。

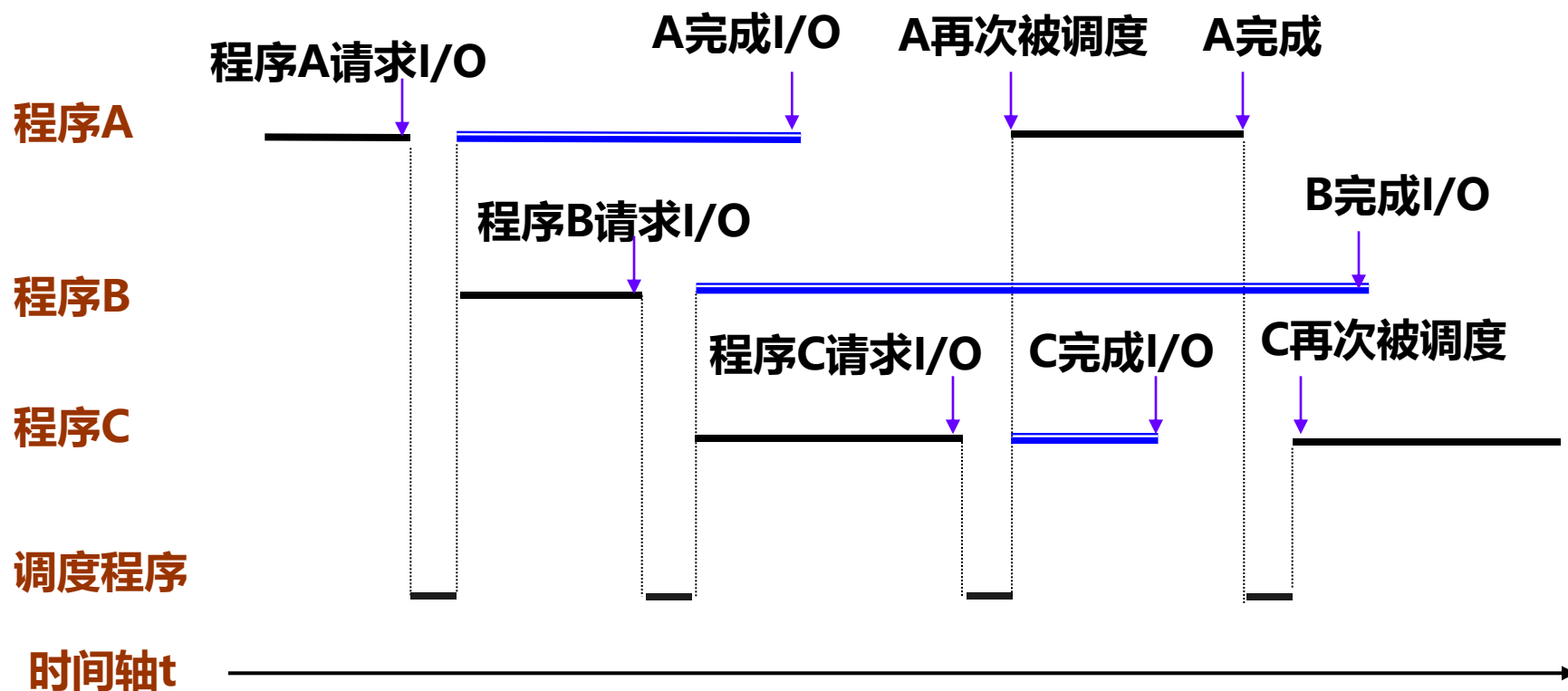
多道程序：

在内存中存放多个用户程序，同时处于**可运行状态**。

当一个程序等待I/O时，另一个程序可以使用CPU。

## 1.2 操作系统发展过程中形成的一些概念

多道程序示意图：



单线(黑色)表示程序占用CPU，双线(蓝色)表示外设在执行相应程序的I/O请求



## 1.2 操作系统发展过程中形成的一些概念

---

### 4. 多道批处理系统

批处理系统中引入多道程序技术

与单道批处理系统相比：

- ✓ 系统吞吐量（单位时间内完成的总工作量）大；
- ✓ 资源利用率高；
- ✓ 周转时间（作业从进入系统到完成所经历的时间）长。



## 1.2 操作系统发展过程中形成的一些概念

### 5. 分时系统 (Time-sharing System)

多个用户（程序）共享一台计算机，按时间片（time slice）轮流使用。

**时间片**：OS将CPU时间划分为若干个片段

#### 分时的特点：

- ✓ 多路性：同时有多个用户（程序）使用一台计算机  
宏观上：是多个人（程序）同时使用一个CPU  
微观上：多个人（程序）在不同时刻轮流使用CPU
- ✓ 交互性：用户根据系统响应结果进一步提出新请求(用户直接干预每一步)
- ✓ "独占"性：用户感觉不到计算机为其他人服务  
(OS提供虚机器，各个用户的虚机器互不干扰)
- ✓ 及时性：系统对用户提出的请求能及时响应





## 1.2 操作系统发展过程中形成的一些概念

---

### 6. 并发 (Concurrence) 与并行 (Parallel)

- ✓ 并行：两个或多个事件在同一时刻发生。
- ✓ 并发：两个或多个事件在同一时间间隔内发生。

在单处理机系统中，多个程序的并发执行是如何体现的？



## 1.2 操作系统发展过程中形成的一些概念

---

### 7. 多用户 (Multiuser) 与多任务 (Multitask)

- ✓ 多用户：允许多个用户通过各自的终端使用同一台主机，共享主机系统中的各类资源。
- ✓ 多任务：允许多个程序并发执行。



## 1.2 操作系统发展过程中形成的一些概念

---

### 8. 实时OS

- ✓ 是指系统能够实时响应外部事件的请求，在规定的短时间内完成对该事件的处理，并控制所有实时设备和实时任务协调运行。
- ✓ 及时响应：延迟时间短
- ✓ 高可靠性：容错，冗余
- ✓ 一般是专用的，如武器系统的实时控制，生产过程的实时控制等。



## 1.2 操作系统发展过程中形成的几个概念

---

### 9. 网络OS

**计算机网络：**物理上分散的**自主**计算机通过通信系统的线路**互连**而成。

自主：具有独立处理能力

互连：计算机之间的通信和相互合作。

✓ 通信、信息交换、资源共享

✓ 互操作、协作

**网络OS：**提供网络通信和网络服务功能的操作系统。

**网络OS的两种基本模式：**

✓ 客户/服务器（Client/Server）模式

✓ 对等（Peer-to-Peer）模式



## 1.2 操作系统发展过程中形成的一些概念

---

### 10. 分布式OS (Distributed OS)

- ✓ 基础：网络

- ✓ 分布处理的**透明性**

运行在**不具有共享内存**的多台机器上，但在用户眼里却象一台计算机。

- ✓ 一个**统一的操作系统**

- ✓ 逻辑上紧密耦合



## 1.2 操作系统发展过程中形成的一些概念

---

### 网络OS和分布式OS的比较:

#### ✓ 耦合程度

分布式OS是在各机器上统一建立的，统一进行全系统的管理；

网络OS通常容许异种OS互连，各机器上各种服务程序需按不同网络协议互操作。

#### ✓ 并行性

分布式OS可以将一个进程分散在各机器上并行执行，包括进程迁移；

网络OS则各机器上运行的程序是相互独立的。

#### ✓ 透明性

用户是否知道或指定资源在哪个机器上。

分布式OS的网络资源调度对用户透明，用户不了解所占有资源的位置；

网络OS中对网络资源的使用要由用户明确指定。



## 1.3 OS对运行环境的要求

---

### 1. CPU

#### 1) 特权指令 - 多道程序的需要

只能由OS使用。例如，启动外部设备，建立存储保护，清内存、关中断等。

如果没有特权指令的话，会有什么问题？

**CPU如何知道是OS还是用户程序在执行呢？**

依赖于CPU的状态标识。



# CPU

---

## 2) CPU的2种工作状态（执行模式）

- ✓ 核心态（Kernel Mode）或称管态
- ✓ 用户态（User Mode）或称目态

### 核心态和用户态的区别：

#### 处理器处于核心态时：

- ✓ 全部指令（包括特权指令）可以执行
- ✓ 可使用所有资源
- ✓ 并具有改变处理器状态的能力

#### 处理器处于用户态时：

- ✓ 只有非特权指令能执行

特权级别不同，可运行的指令集合也不同

特权级别越高，可以运行的指令集合越大

高特权级别对应的可运行指令集合包含低特权级的





# CPU

---

## 3) 程序状态字PSW ( Program Status Word) 和程序计数器PC (Program Counter)

✓ **PSW**: 指示程序执行的当前状态, 主要包括

**CPU的工作状态**——指明核心态还是用户态, 用来说明当前在CPU上执行的是操作系统还是应用程序, 从而决定其是否可以使用特权指令或拥有其他的特殊权力

**条件标志**——反映指令执行后的结果特征

**中断标志**——指出是否允许中断

✓ **PC**: 指示下一条要执行的指令



## 1.3 OS对运行环境的要求

---

### 2. 内存

是支持OS运行的硬件环境的一个重要方面。

- ✓ 程序必须存放在内存中才能运行。
- ✓ 在多任务系统中，操作系统要管理、保护各任务的程序和数据，使它们不至于受到破坏和相互干扰。
- ✓ 操作系统本身也要存放在内存中并运行，不能被破坏。

**内存空间：**由若干个存储单元（字节或字）组成的一维连续的地址空间

- ✓ 存储的最小单位：1个二进制位
- ✓ 最小编址单位：字节，一个字节包含8个二进制位



# 内存

---

## 1) 内存分块

块作为分配内存空间的基本单位，如4KB为1块。

为什么要按块来分配内存空间？

旨在简化对内存的分配和管理

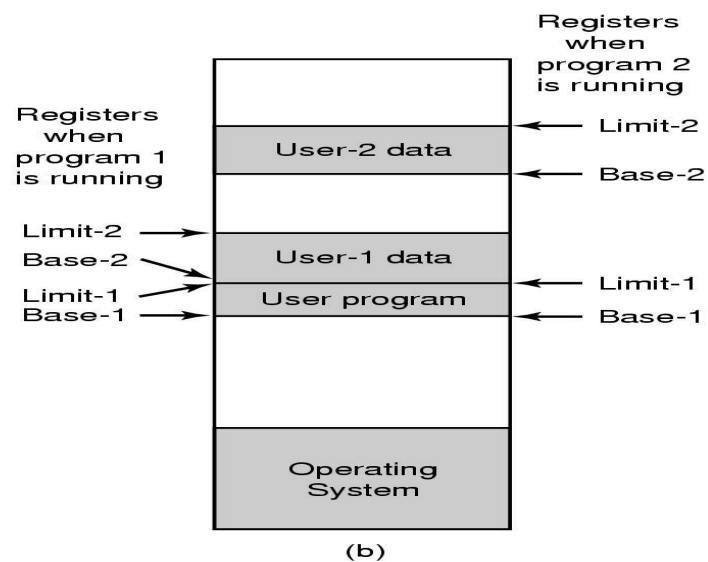
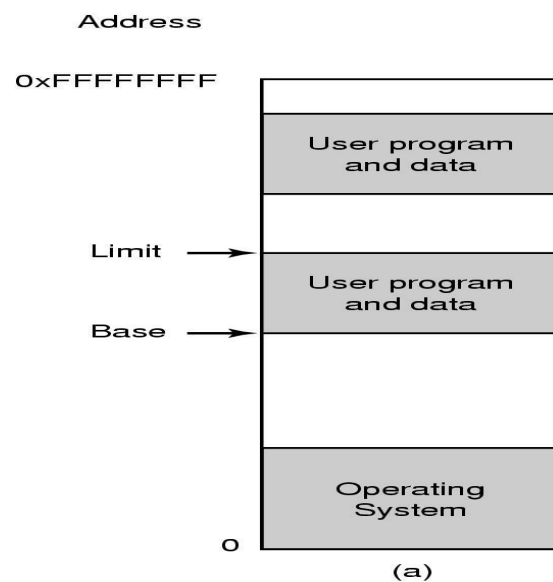
# 内存

## 2) 内存保护 - OS正常运行的基本条件

常用的保护机制:

### (1) 界限寄存器

存放某任务在内存的上界和下界地址(或者下界与长度)。





# 内存

---

## 界限寄存器实现存储保护的方法：

- ✓ 在CPU中设置一对下界寄存器和上界寄存器  
存放用户程序在内存中的下界和上界地址
- ✓ 也可将一个寄存器作为基址寄存器，另一寄存器作为限长寄存器（指示存储区长度）
- ✓ 每当CPU要访问内存，硬件自动将被访问的内存地址与界限寄存器的内容进行比较，以判断是否越界
- ✓ 如果未越界，则按此地址访问内存，否则将产生中断——越界中断（存储保护中断）



# 内存

---

## (2) 存储保护键 (Key)

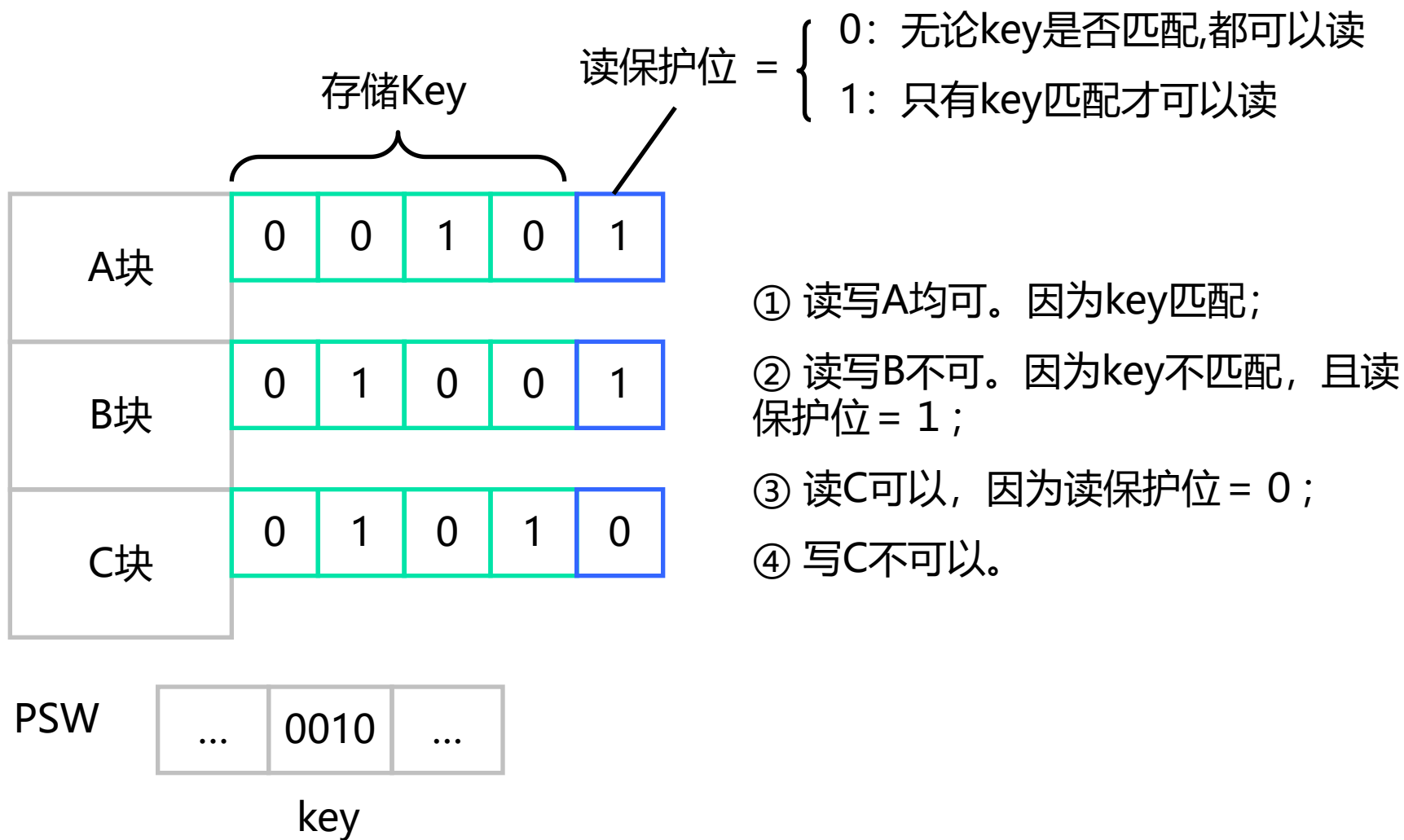
当一个程序进入内存时，OS为其分配一个唯一的Key。

同时将分配给它的每个存储块都设置成该Key。如IBM 370。

### 该方法的基本要点：

- ✓ 每个运行的程序及其存储块有1个Key；
- ✓ PSW中的存储Key字段存放当前运行程序的Key；
- ✓ 访问内存时，两个Key匹配；
- ✓ 通常将0（在PSW中）作为“万能键”；
- ✓ 存储块引入读保护位：0：Key不匹配时也可读，1：Key不匹配时不可读。

# 内存





## 1.3 OS对运行环境的要求

---

### 3. 中断 - 如果没有中断，OS将难以工作。

指CPU在收到外部中断信号后，停止原来工作，转去处理该中断事件，完毕后回到原来断点继续工作。

CPU对系统中发生的“异步（随机）”事件的处理

#### 中断的类型：

- ✓ 硬件中断
- ✓ 异常 (Exception)
- ✓ 陷入 (Trap) - 访管中断 (系统调用)

有人称 “OS是中断驱动的”。







## 1.3 OS对运行环境的要求

---

### 4. 时钟 - OS必不可少的硬件设施

(1) 硬件时钟：通过时钟寄存器实现。

✓ 绝对时钟：记录当前时间

✓ 相对时钟（间隔时钟）：**分时系统的基础。**

(2) 软件时钟：通过时钟队列实现。



## 1.3 OS对运行环境的要求

---

### 5. 重定位

将程序中的相对地址转换为绝对地址。

原因：运行前不可能知道程序将放在内存的什么位置。

- ✓静态重定位：程序装入内存时，由装入程序重定位；
- ✓动态重定位：CPU每次访问内存时，由动态地址转换机构（硬件）自动进行



## 1.4 典型OS实例

---

### 1. Unix

一群计算机迷在贝尔实验室开发出Unix

初衷：可以在一台无人使用的DEC PDP-7 小型计算机上玩星际探险游戏

Ken Thompson, Dennis Ritchie

1983年图灵奖获得者

1999年4月 美国国家技术金奖



## 1.4 典型OS实例

---

(1) UNICS(Uniplexed Information and Computing Service)

改名为Unix

(2) 结构：用C、汇编语言写成的

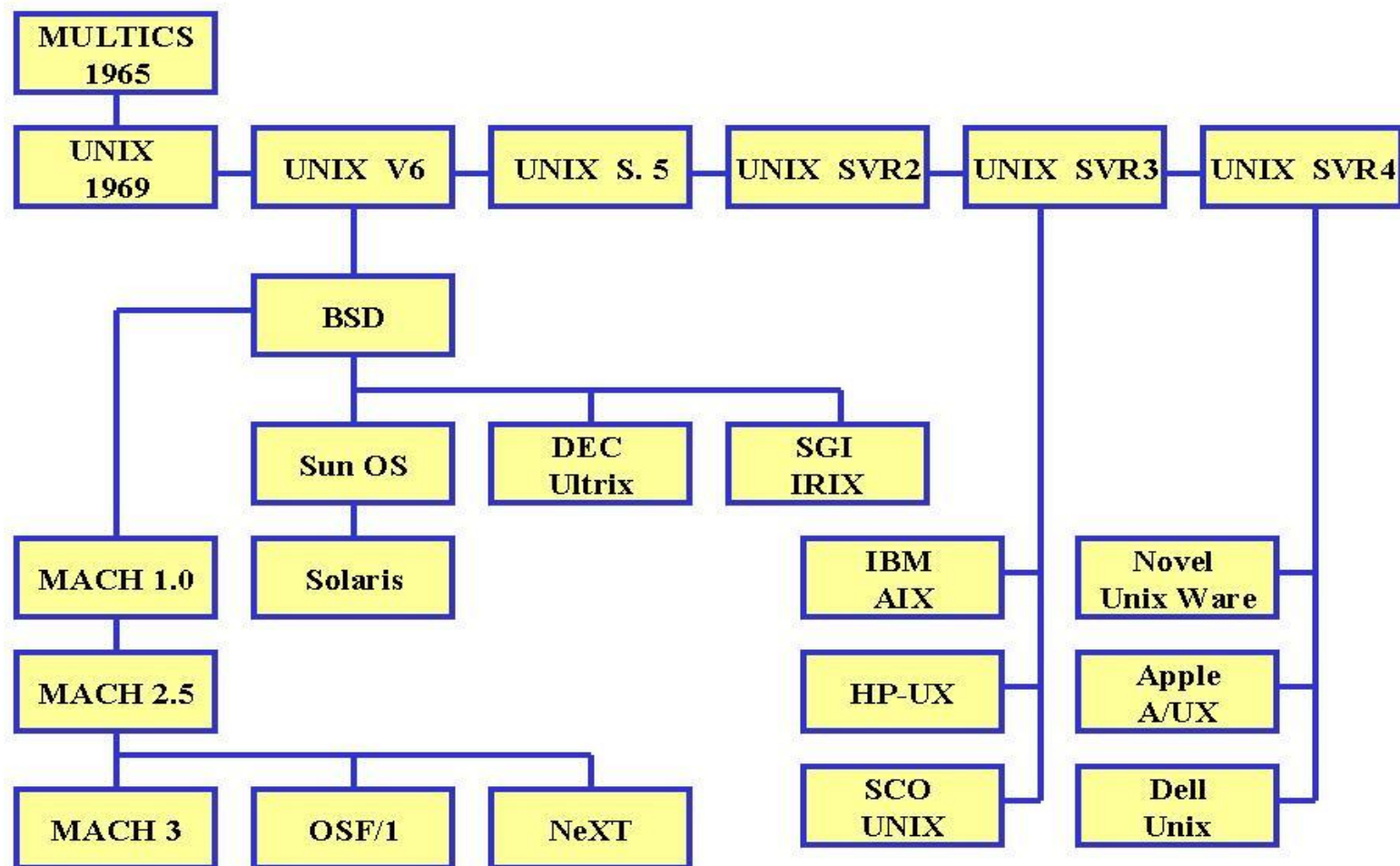
**良好的、通用的、多用户、多任务、分时操作系统**

(3) 多种变体

**两个版本系列**

- ✓ AT&T System V
- ✓ BSD (Berkeley Software Distribution)

## 1.4 典型OS实例





## 1.4 典型OS实例

---

### 2. Linux

1991年，芬兰赫尔辛基大学的一名大学生Linus Benedic Torvalds首先开发

- ✓ 9300行C语言代码，950行汇编语言代码（0.01版）
- ✓ 源代码公开
- ✓ 与Unix兼容
- ✓ 1994年，1.0版：约165000行代码

### 3. Windows

MS-DOS、Windows 3.1/95/98/Me、Windows NT、Windows 2000/XP、Windows CE、Windows Server 2003，Windows Vista，Windows Server 2008，Windows 7，Windows 8，Windows 10



## 1.4 典型OS实例

---

### 历史上一些重要的操作系统:

- ◆ FMS (FORTRAN Monitor System) 和IBSYS (IBM为7094配备的操作系统)
- ◆ OS/360 (IBM为系列机360配备的操作系统)
- ◆ CTSS (Compatible Time Sharing System)
- ◆ MULTICS (MULTiplexed Information and Computer Service)
- ◆ UNIX类、Linux
- ◆ CP/M
- ◆ MS-DOS, Windows 3.1/95/98/Me, Windows NT, Windows 2000/XP, Windows CE, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7
- ◆ Macintosh
- ◆ OS/390
- ◆ Mach
- ◆ VxWorks





## 1.5 现代操作系统的基本特征

---

- (1) 并发
- (2) 共享
- (3) 虚拟
- (4) 不确定



## 1.6 从不同角度认识操作系统

---

### (1) 软件

外在特性：使用方式（命令，系统调用）

内在特性：结构，功能

### (2) 资源管理器

底层资源共享，提高资源利用率

### (3) 虚拟机

方便用户使用计算机

### (4) 标准服务提供者

提供每个用户需要的标准工具

如标准库、窗口系统



## 1.6 从不同角度认识操作系统

---

### (5) 仲裁者 (协调者)

使多个应用程序 (用户) 高效、公平地一起工作

保护应用程序之间不互相干扰

### (6) 幻觉制造者 ( illusionist )

提供硬件的高层接口, 程序员 (用户) 感觉不到硬件限制

操作系统提供 “无限大” 的内存、 “无限多” 的CPU



## 1.7 学习操作系统课程要达到的目标

---

### 1. 为什么要学习操作系统?

- ✓ 操作系统涉及到计算机科学的很多领域：计算机体系结构，软件设计，数据结构，算法等
- ✓ 设计操作系统或者修改现有的系统  
如嵌入式系统(Embedded OS)
- ✓ 开发应用程序必须与操作系统打交道，特别是并发程序
- ✓ 开发应用程序时借鉴操作系统的设计思想和算法，特别是复杂软件系统设计的思维方法
- ✓ 操作系统的许多思想和方法可以应用到其他领域



# 1.7 学习操作系统课程要达到的目标

---

## 2. 学习操作系统要达到的目标

1) 理解OS的基本概念、工作原理、典型实现技术以及OS设计中考虑的各种因素并能合理运用

- 从设计OS的角度去思考、学习
- 理解（**非简单的记忆**）概念、原理的实质：是什么？为什么？有什么用？如何实现？适用于什么场合？如信号量。
- 领会其中的重要思想：如工作集，设备独立性，SPOOLing技术
- **建立OS的整体概念**
- 系统观与方法学



## 1.8 学习操作系统课程要达到的目标

---

### 2) 设计、实现具体的操作系统；或剪裁操作系统（如嵌入式OS）

- 能独立对小型操作系统的部分功能进行源代码分析、设计和实现
- 设计一个功能强大的实用的OS 是非常不容易的
- 需要的理论、技术涉及计算机体系结构、软件设计方法学、软件管理、数据结构与算法、网络等相关知识
- 需要考虑硬件、应用程序、将来可能的变化  
概念的抽象，接口设计



## 1.8 学习操作系统课程要达到的目标

---

### 3) 应用软件（特别是大型软件）开发

- OS作为一个复杂软件系统，涉及到分析解决复杂工程问题所需的许多重要思想和方法
- 并发软件开发：同步，死锁
- 借鉴OS 设计中的问题及其解决方法、思想
  - 如：权衡（tradeoff）：时间与空间；性能与方便使用；通用性与效率
  - 抽象、虚拟、实证、分层
- 思想、方法在不同领域往往是相通的



# 1.8 学习操作系统课程要达到的目标

---

## 3. 对OS的整体理解

从设计OS（OS设计者）的角度去学习、思考

OS是软件：

- 资源管理器：管理的对象包括硬件、软件
- 用户接口

功能：

- 管理CPU、内存、外存、I/O设备
- 管理程序、文件
- 提供用户使用OS的接口：程序级（系统调用）、命令级

**4大功能：进程管理、内存管理、文件管理、I/O 设备管理**

整个操作系统课程就是围绕这4大功能展开的





## 1.8 学习操作系统课程要达到的目标

---

### 性能:

- 效率：时间；空间
- 方便使用：屏蔽细节；统一接口
- 提供用户使用OS的接口：程序级（系统调用）、命令级
- 扩展性：如新设备的加入
- 可靠性：容错
- 安全性：访问控制



## 1.8 学习操作系统课程要达到的目标

---

### 应用环境:

- 是否多处理机
- 是否多任务
- 存储空间限制
- 响应时间要求

### 总体目标:

批处理/分时/实时, 是否多任务、多用户, 通用/专用



## 进一步学习的参考书:

### 1) 操作系统设计与实现

- [1] Andrew S. Tanenbaum. 陈渝等译. 操作系统设计与实现 (第三版). 电子工业出版社, 2007
- [2] 于渊. Orange S: 一个操作系统的实现. 电子工业出版社, 2009
- [3] Scott Maxwell. 冯锐等译. Linux内核源代码分析. 机械工业出版社, 2000

### 2) 系统程序设计

- [4] Kay A. Robbins, Steven Robbins. 陈涓等译. Unix系统编程. 机械工业出版社, 2005
- [5] W. Richard Stevens, Stephen A. Rago. 戚正伟, 张亚英, 尤晋元译. Unix环境高级编程 (第3版). 人民邮电出版社, 2014
- [6] Jeffrey Richter. 葛子昂等译. Windows核心编程 (第5版). 清华大学出版社, 2008

### 3) 分布式操作系统

- [7] Andrew S. Tanenbaum. 陆丽娜等译. 分布式操作系统. 电子工业出版社, 2008



# 第2章 进程管理

---

**2.1 进程 (Process)**

**2.2 进程控制**

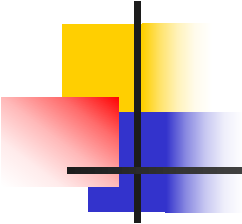
**2.3 进程同步**

**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。

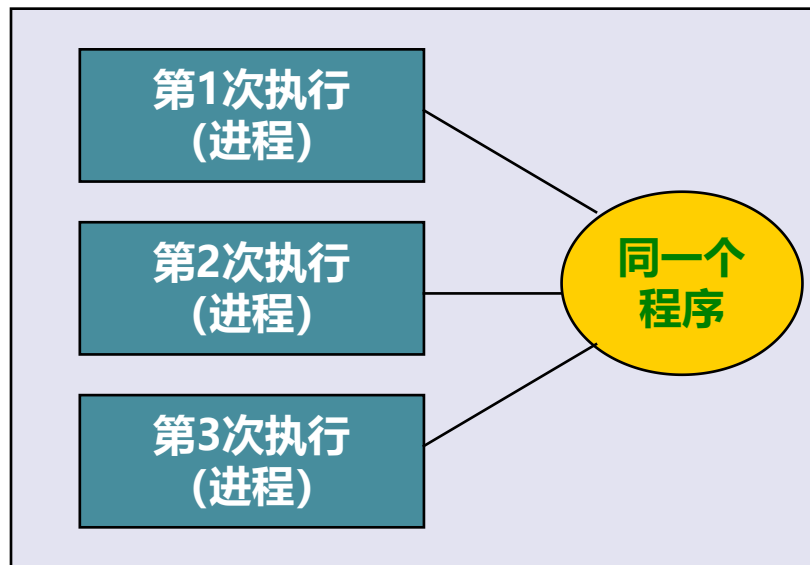
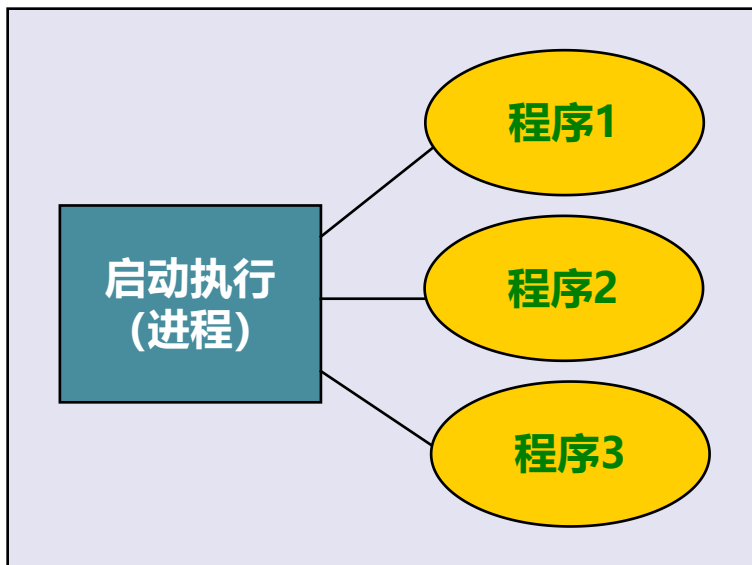
每个进程都有自己的地址空间。

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

---

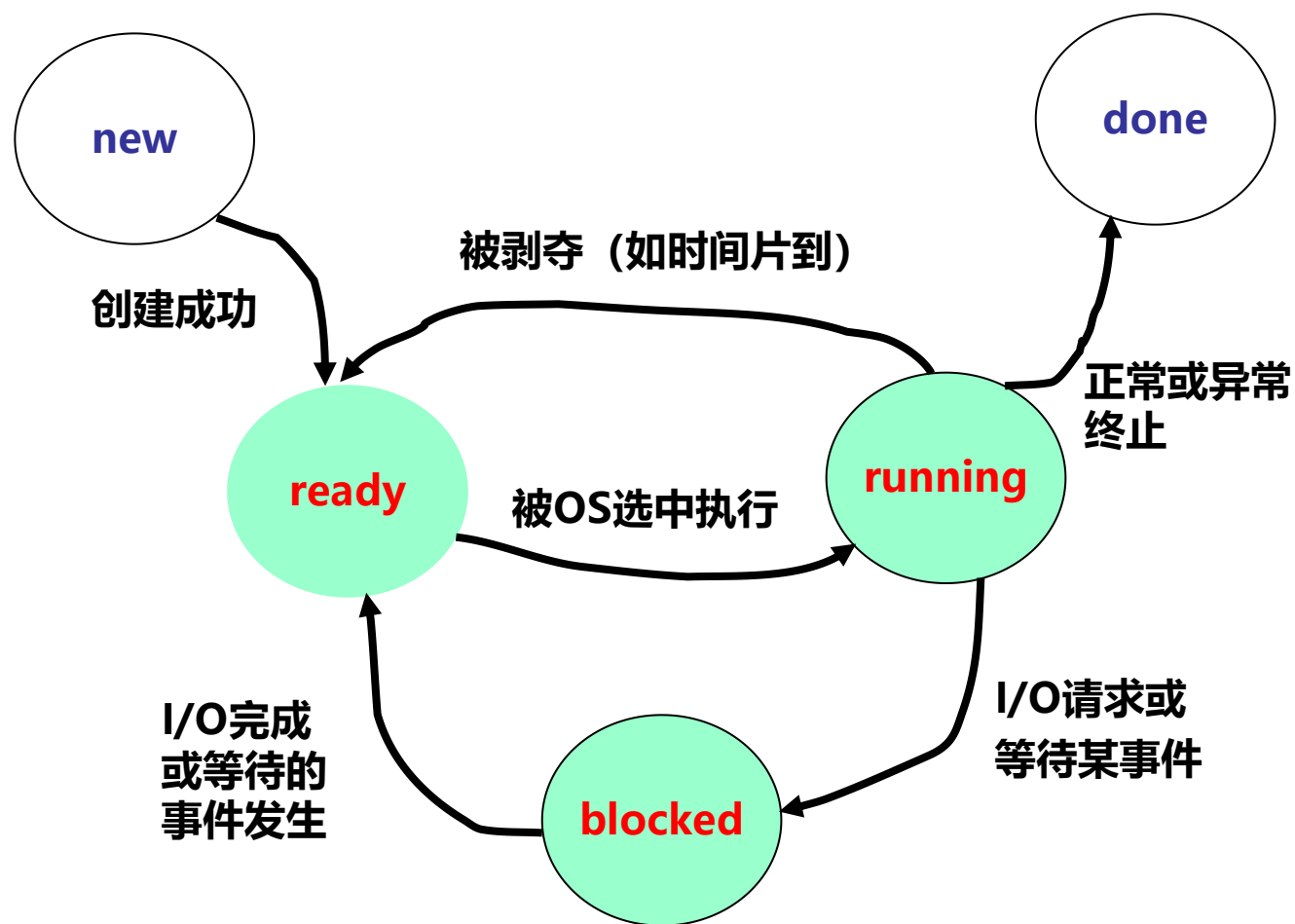
## 引入new和done状态的原因：

- ✓ 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，其退出系统的过程一般也分为两步：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

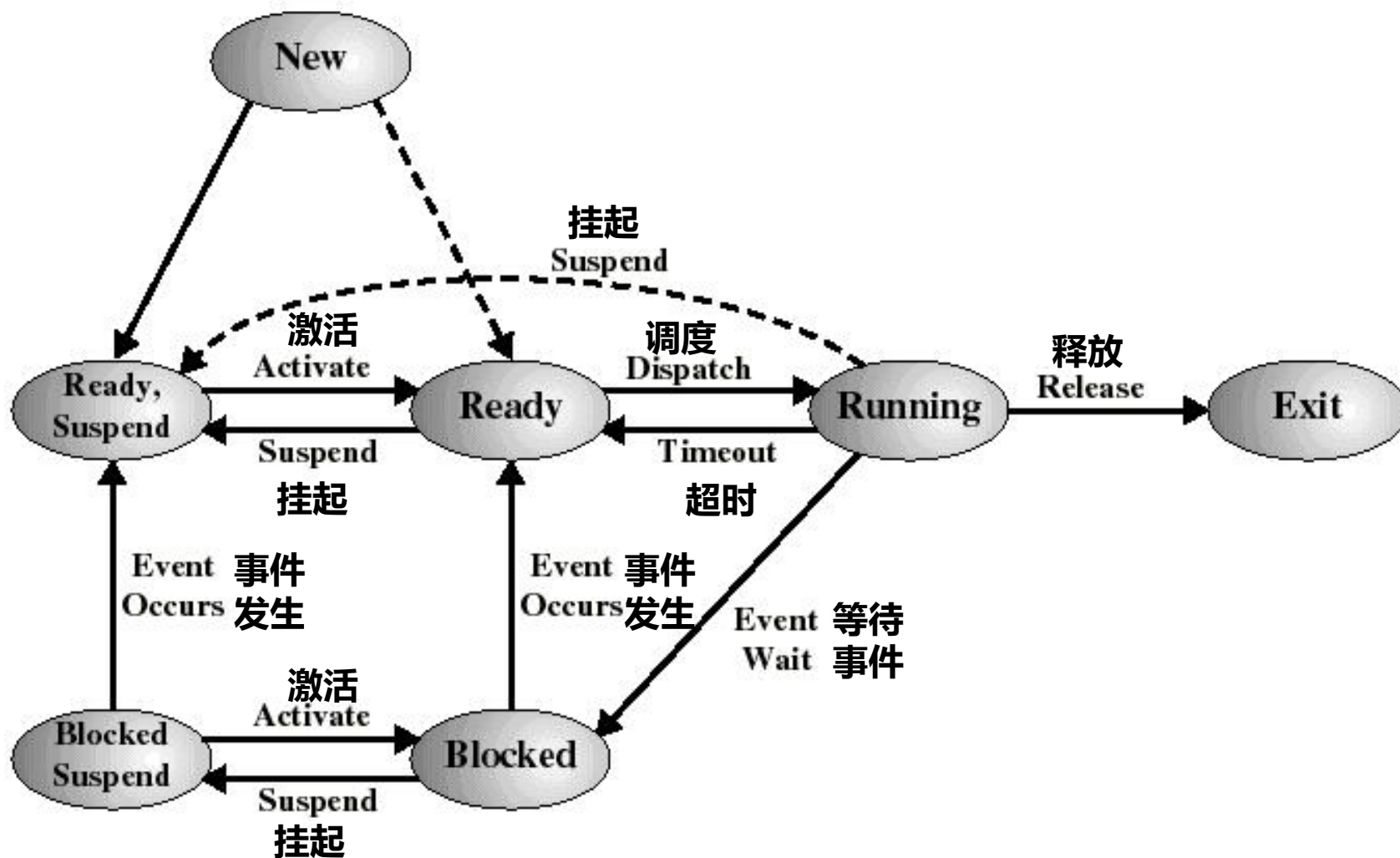
---

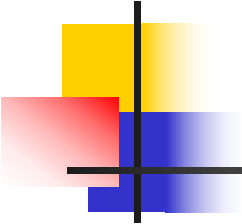
- **激活 (Activate)** : 把一个进程从外存转到内存。

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

# 七状态进程模型





## 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

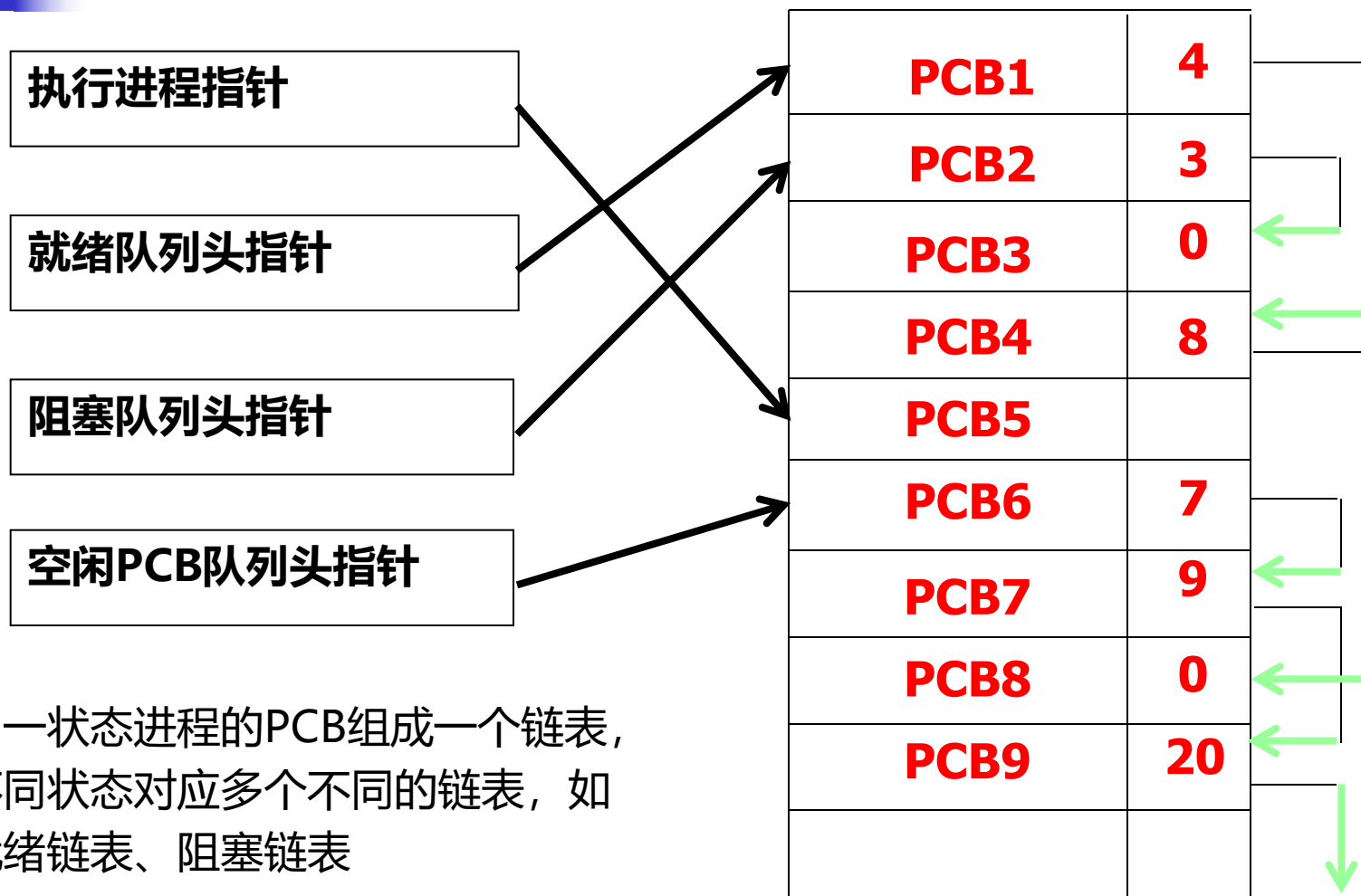
---

## 4. PCB的组织方式

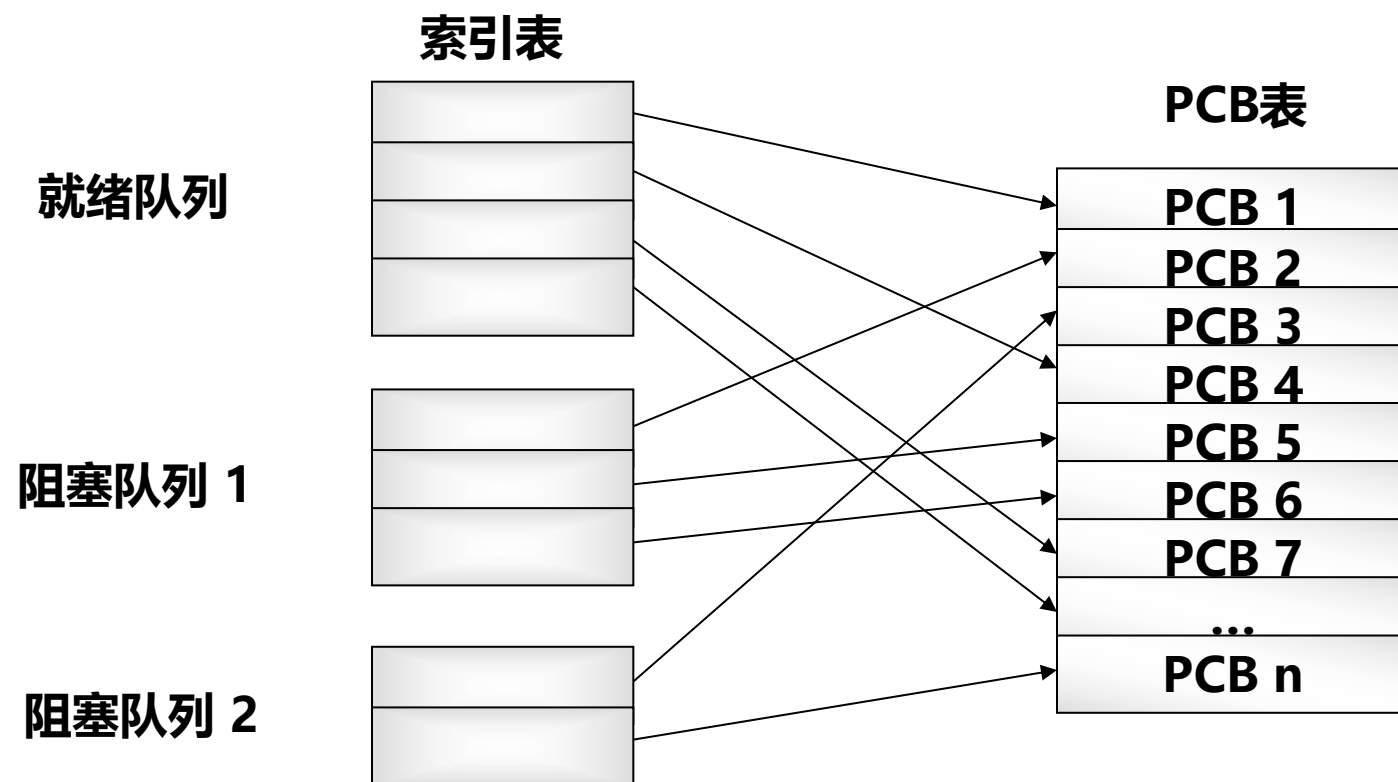
一般来说，系统把所有PCB组织在一起，并把它们放在内存的固定区域，构成**PCB表**。

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

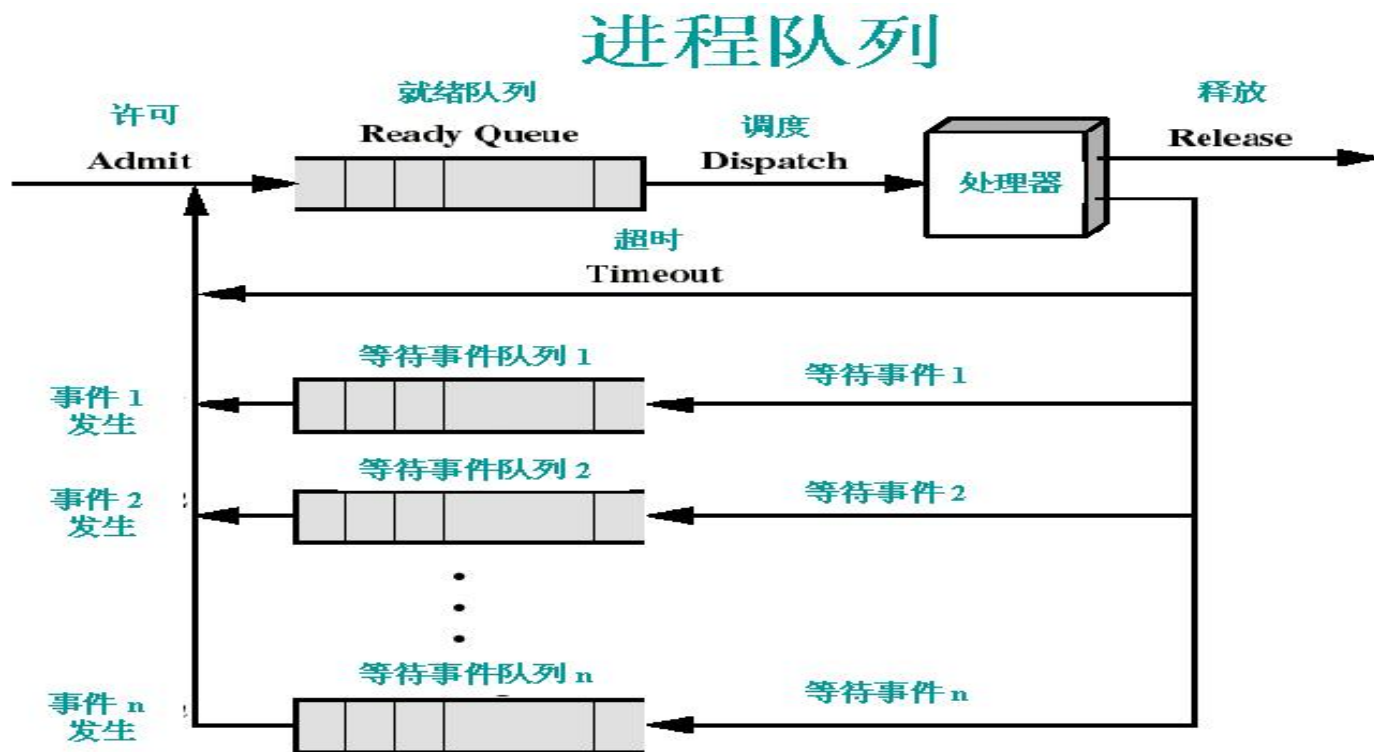


# PCB的组织方式



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：FIFO）
- 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 = { 0: 临界区内无进程, 初始值  
1: 临界区内有进程

```
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:  $\text{register} \leftarrow [\text{memory}]$  ; 将内存单元的值送寄存器

$[\text{memory}] \leftarrow 1$  ; 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操作在很多情况下是成对出现的。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。

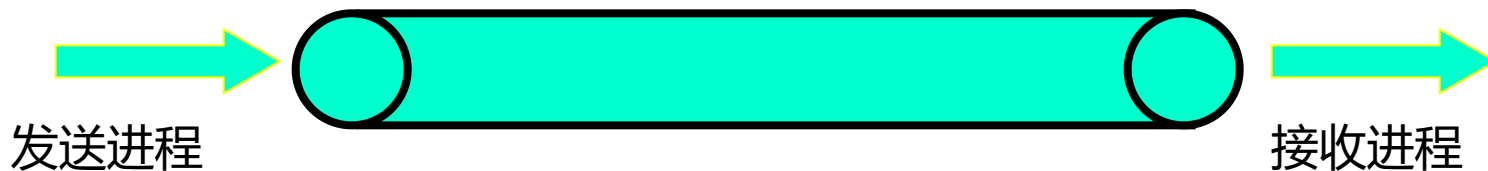
邮箱是一种中间实体，一般用于非实时通信。



# 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？

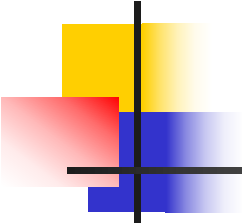
【例】一个简化的生产者-消费者问题。

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。

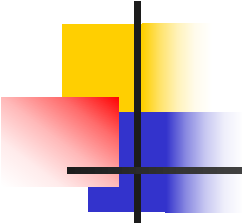




# 如何使用线程？

---

```
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

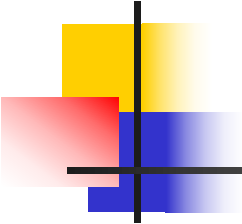
多个任务处在不同的地址空间，不会相互干扰；  
不能共享全局变量。

✓ 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。

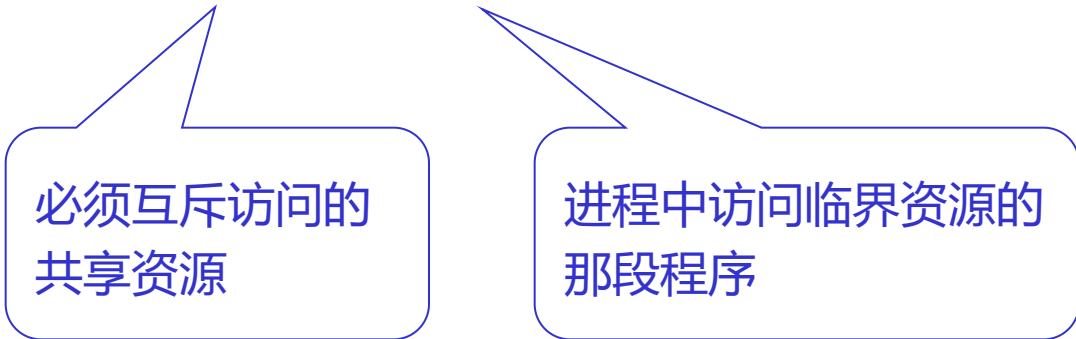


# 重点掌握的内容

---

## 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。**

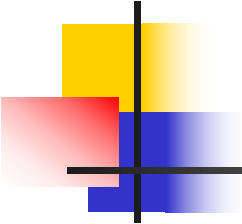


# 重点掌握的内容

---

## 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)

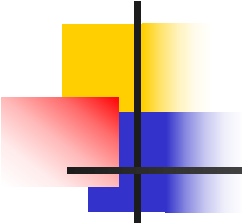
需要同步和保护

- ✓ 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

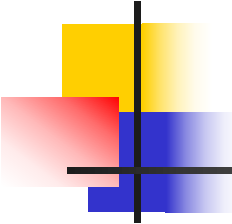
---

### 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。



# 第3章 进程调度 ( Process Scheduling )

---

**3.1 进程调度的功能与目标**

**3.2 进程调度的方式与时机**

**3.3 进程调度算法**



## 3.1 进程调度的功能与目标

---

### 一、进程调度的功能

#### 1. 进程调度要做什么？

当多个进程就绪时，OS的调度程序（Scheduler）决定先运行哪一个。

- ✓ 记录进程的状态 - 借助于PCB;
- ✓ 选择投入运行的进程 - 依赖于调度策略（算法）；
- ✓ 进行进程的上下文切换。



# 进程调度的功能

---

## 2. 进程调度要考虑的问题 - 处理机调度

- ✓ 按什么原则分配CPU：需要确定调度方式和算法
- ✓ 何时分配CPU：需要确定调度时机
- ✓ 如何分配CPU：进程的上下文切换（CPU调度过程）



## 3.1 进程调度的功能与目标

---

### 二、进程调度程序要达到的目标（选择调度算法的准则）

(1) 公平：确保每个进程获得合理的CPU份额。

(2) CPU利用率高：尽可能使CPU 100%忙碌。

(3) 响应时间短。

响应时间指的是交互式用户从输入到有结果输出之间的时间。

(4) 周转时间短。

周转时间 = 作业完成时间 - 作业提交时间

(5) 吞吐量大。

吞吐量指的是单位时间内完成的作业数。

(6) 调度算法不宜太复杂，以免占用太多的CPU时间。





## 3.2 进程调度的方式与时机

---

### 一、进程切换

暂停一个进程，运行另一个进程。

**进程切换的时机：OS靠中断获得CPU的控制权**

#### (1) 硬件中断

- ✓ 时钟中断：OS检查时间片是否到。若时间片到，则转入Ready，否则当前进程继续运行；
- ✓ I/O中断：若是某个进程等待的事件，则将其由阻塞->就绪。然后，决定是否继续运行当前进程还是让更高优先级的就绪进程剥夺之。

#### (2) 进程异常

由OS处理，决定是否终止之（若是致命错误），还是继续运行或切换。

#### (3) 请求OS服务（陷入）

如请求I/O，在启动I/O后通常置当前进程为阻塞。



## 3.2 进程调度的方式与时机

---

### 二、调度方式

(1) 非抢占方式 (Non-preemptive mode) :非剥夺

一旦某进程被调度, 直到其完成或等待某事件而阻塞, 才会切换到其他进程。

(2) 抢占方式 (Preemptive mode) :可剥夺

允许暂停正在运行的进程, 切换到其他进程。

**抢占的原则:**

✓时间片原则: 时间片到时抢占

✓优先级原则: 优先级高者到时抢占



## 3.2 进程调度的方式与时机

---

### 三、引起进程调度的时机

- (1) 当前进程终止：执行完毕或出现错误而结束
- (2) 当前进程阻塞：等待I/O完成，执行了阻塞原语
- (3) 当前进程执行的时间片到
- (4) 出现了一个更高优先级的就绪进程

**进程中断/异常/系统调用返回到用户态时**



## 3.3 进程调度算法

---

- ✓ 引入多道程序系统的直接目的就是想让处理机“忙”。所以当处理机空闲时，系统需要从多个就绪进程中挑选一个使其投入运行。选择哪一个呢？这需要一种算法。**调度算法的实质就是一种资源分配。**
- ✓ **从资源的角度来看**，该算法确定了处理机的分配策略，故称其为**处理机调度算法**；
- ✓ **而从资源使用者的角度看**，该算法确定了进程运行的次序，故也称**进程调度算法**。



## 3.3 进程调度算法

---

### 一、先来先服务 (First Come First Serve, FCFS)

或称先进先出进程调度算法 (FIFO)

按照进程就绪的先后次序来调度进程

优点：实现简单

缺点：

- (1) 未考虑进程的优先级
- (2) 有利于CPU繁忙型进程，不利于I/O繁忙型进程
- (3) 有利于长进程，不利于短进程



## 3.3 进程调度算法

---

### 二、时间片轮转 (Round Robin, RR)

实现方法：

将所有的就绪进程按FCFS原则排成一个队列，  
规定一个时间片为进程每次使用CPU的最长时间，  
每次选择队首进程运行，  
当时间片到时，剥夺该进程的运行，将其排在队尾。



# 时间片轮转

---

选择多大的时间片合适？大一些好还是小一些好？

- ✓ 如果太小，则导致频繁的进程切换，消耗CPU时间
- ✓ 如果太大，则响应时间长，大到一个进程足以完成其全部运行工作所需的时间时，就退化为FCFS模式。



# 时间片轮转

---

## 选择时间片大小要考虑的因素：

- (1) 对响应时间的要求
- (2) 就绪进程数
- (3) 系统的处理能力

## 基本要求：

保证用户键入的常用命令能在1个时间片内处理完。

响应时间 = 进程数 \* 时间片





## 3.3 进程调度算法

---

### 三、短进程（作业）优先

选择估计运行时间最短的进程运行。

缺点：

对长进程（作业）不利。

极端情况下，会使长进程（作业）得不到调度。

如何知道哪个是最短的？



## 3.3 进程调度算法

---

### 四、基于优先级（优先数）的调度

- ✓ 每个进程一个优先级；
- ✓ 总是选择就绪队列中优先级最高的进程投入运行；
- ✓ 可以是抢占式，或非抢占式。



# 基于优先级的调度

---

## 优先级的确定:

### (1) 静态优先级

在进程创建时确定，在进程的运行期间保持不变。

很可能出现优先级低者永远得不到调度的情况。

### (2) 动态优先级

在进程创建时指定一个基础优先级，

每隔一定时间（如一个时钟中断），重新计算优先级。例如：

等待时间越长，优先级越高；

已占用CPU时间越长，优先级越低。



## 3.3 进程调度算法

---

### 五、多级反馈队列

- (1) 按优先级设置 $n$  ( $n > 1$ ) 个就绪队列，每个队列赋予不同优先级。如第一级队列到最后一级队列，优先级依次降低。
- (2) 优先级越高队列，分配的时间片越小。为什么？
- (3) 每个队列按照先来先服务原则排队。
- (4) 一个新进程就绪后进入第一级（或相应优先级）队尾。
- (5) **调度方法**
  - ✓ 每次选择优先级最高队列的队首进程运行；
  - ✓ 若被调度进程的时间片到，则放入下一级队列的队尾；
  - ✓ 最后一级队列采用时间片轮转；
  - ✓ 当有一个优先级更高的进程就绪时，可以抢占CPU，被抢占进程回到原来队列的末尾。



# 多级反馈队列

---

该算法综合了前面几种算法的优点。

既考虑了先来先服务，又照顾了长进程；

既考虑了优先级，又讲求公平。

## 说明：

在实际系统中，可能会采取更复杂的动态优先级调度策略。

比如，当一个进程等待时间较长时，提升到上一级队列。



## 3.3 进程调度算法

---

### 六、实时OS的调度

对时间要求严格。

一般基于开始截止时间、完成截止时间。



# 本章作业

---

教材《计算机操作系统教程（第4版）》

p100:

4.6



# 第4章 内存管理(Memory Management)

---

## 4.1 概述

## 4.2 程序的连接与装入

## 4.3 实存储器管理

## 4.4 虚拟存储管理

## 4.5 小结





## 4.1 概述

---

### 一、存储器的层次结构

**存储系统的设计目标可归纳成3个问题：**

容量，速度，成本

- ✓ 容量：需求无止境
- ✓ 速度：能匹配处理器的速度
- ✓ 成本：成本和其他部件相比应在合适范围之内



# 存储器的层次结构

---

## 容量、速度和成本之间的矛盾：

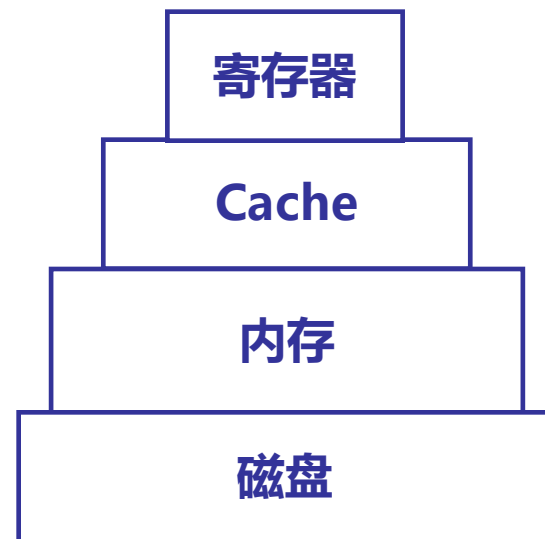
- ✓ 三个目标不可能同时达到最优，要作权衡
- ✓ 存取速度越快，每一个位（Bit）的价格越高
- ✓ 追求大容量，就要降低存取速度
- ✓ 追求高速度，就要降低容量

# 存储器的层次结构

**解决方案：**采用**层次化的存储体系结构**

当沿着层次向下时

- ✓ 每位的价格下降
- ✓ 容量增大
- ✓ 速度变慢



**策略：**

- ✓ 较小、较贵的存储器由较大、较便宜的慢速存储器作为后援

**虚拟存储器**

- ✓ 降低较大、较便宜的慢速存储器的访问频率

**高速缓存**



## 4.1 概述

---

### 二、内存管理的目的

- ✓ 有效利用内存空间

**帕金森 (parkinson) 定律：**内存有多大，程序就有多大

- ✓ 考虑管理的开销：时间，空间
- ✓ 应用程序不必特别考虑内存的大小



## 4.1 概述

---

### 三、内存管理的功能

- ✓ 记录内存的使用情况（是否空闲）
- ✓ 进程所占内存空间的分配与回收
- ✓ 当内存不足时，采取相应措施
- ✓ 内存空间的共享与保护



## 4.2 程序的连接与装入

---

### 一、程序连接的功能

多个目标文件及库文件连接成1个完整的可执行文件。

- ✓ 定位目标文件可能存在的一些外部符号
- ✓ 浮动地址的重定位

### 二、程序连接的时机

- ✓ 静态连接：静态的，只连接1次，多次运行
- ✓ 装入时连接：装入后是静态的
- ✓ 实际运行时连接：调用时动态连接



## 4.1 程序的连接与装入

---

### 三、程序的装入方式

程序能否直接在外存中执行？

不行！每个程序在运行前，必须装入内存。

不一定一次全部装入。

**装入方式：**

- (1) 完全静态装入
- (2) 静态重定位装入
- (3) 动态重定位装入



# 程序的装入方式

---

## 1. 完全静态装入

程序装入时不作任何修改。

即装入内存的每个字节与其可执行文件完全相同。

例如，早期DOS操作系统下的.com文件。





# 程序的装入方式

---

## 2. 静态重定位装入

程序**装入时进行**一次地址重定位，运行时不变。

**重定位：**

程序中的相对地址（从0开始）  绝对地址

**逻辑地址（相对地址）：**

用户的程序经过汇编或编译连接后形成可执行代码，代码通常采用相对地址的形式，其首地址为0，指令中的地址都采用相对于首地址的偏移量。

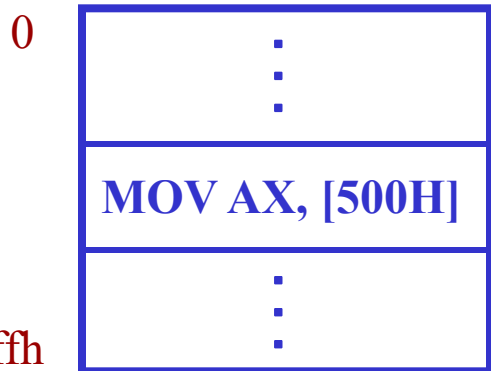
机器是不能用逻辑地址在内存中读取信息的。

**物理地址（绝对地址）：**

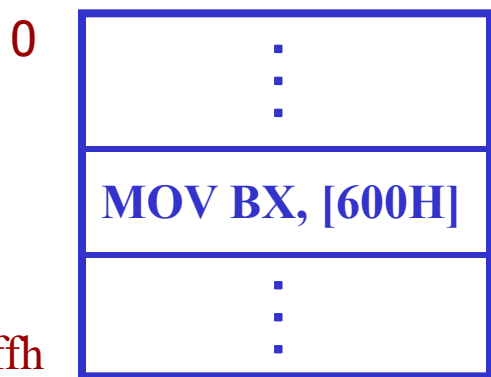
内存中存储单元的实际地址

# 静态重定位装入

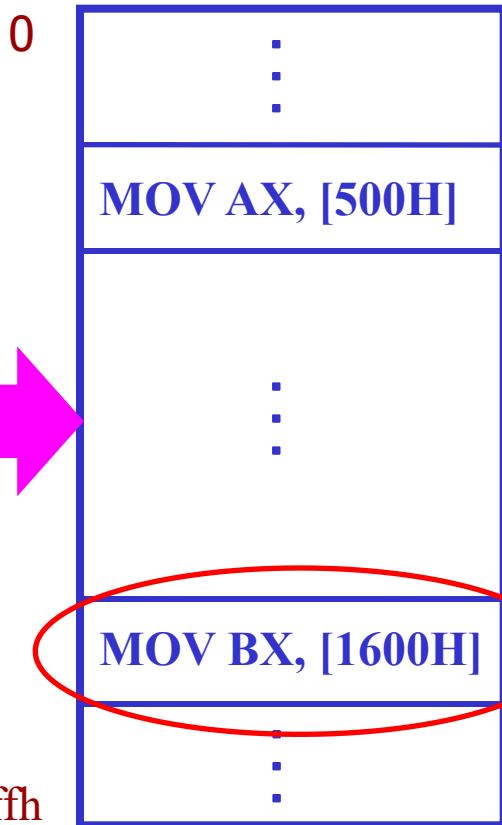
模块A



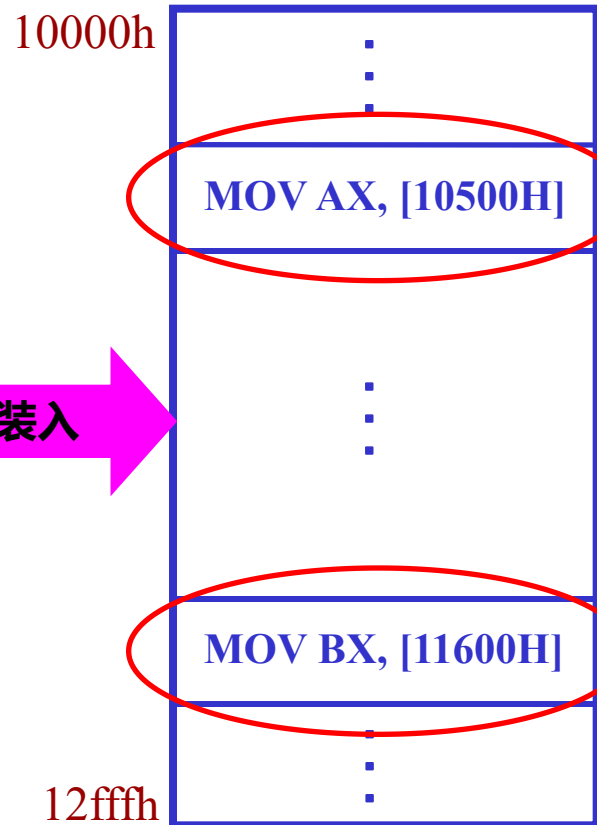
模块B



连接



装入





# 程序的装入方式

---

## 3. 动态重定位装入

真正**执行到一条指令要访问某个内存地址时**，才进行地址重定位。

好处：程序可以在内存中移动。

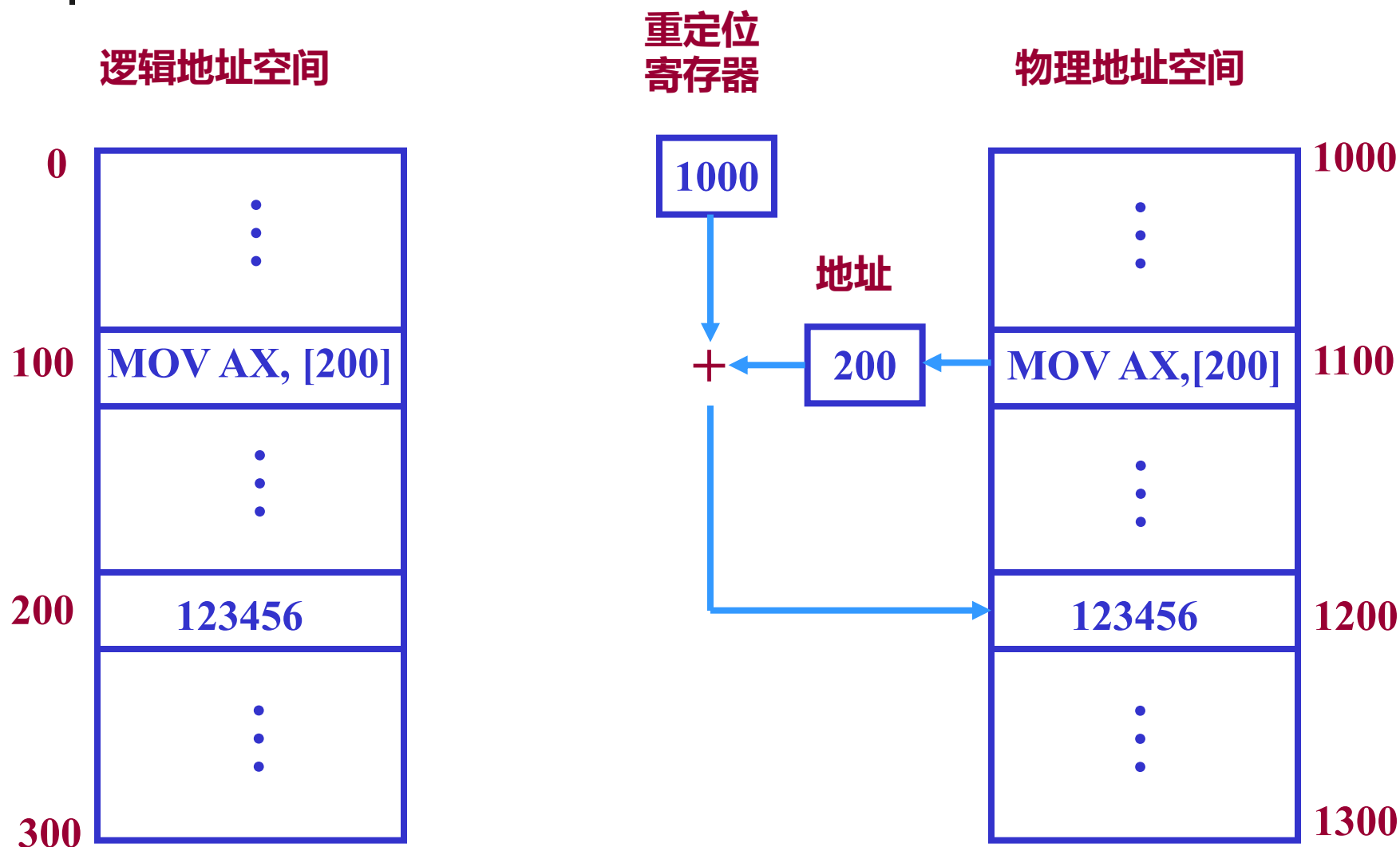
**如何实现动态重定位？**

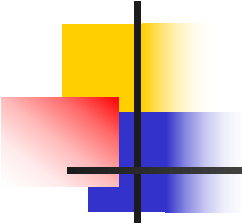
一般设置1个重定位寄存器，

存放当前进程在内存的起始地址

**绝对地址 = 相对地址 + 重定位寄存器的值**

# 动态重定位装入





## 4.3 实存储器管理

---

程序的大小不能超过可用内存空间的大小。

### 一、连续分配

为每个进程分配连续的内存空间。

#### 1. 单一连续区分配

内存中只存在1个用户程序。

整个用户区为该程序独占。

**一般将内存划分为2个区：**

- ✓ 系统区：存放OS程序和数据
- ✓ 用户区：存放用户程序和数据

只能用于单用户、单任务OS。



# 连续分配

---

## 2. 固定分区

将内存的用户区预先划分为若干区域（分区）

分区个数和每个分区的大小是固定的

每个分区存放1个进程

**管理所需的数据结构：**

1个分区使用表（内存分配表），记录分区状态。

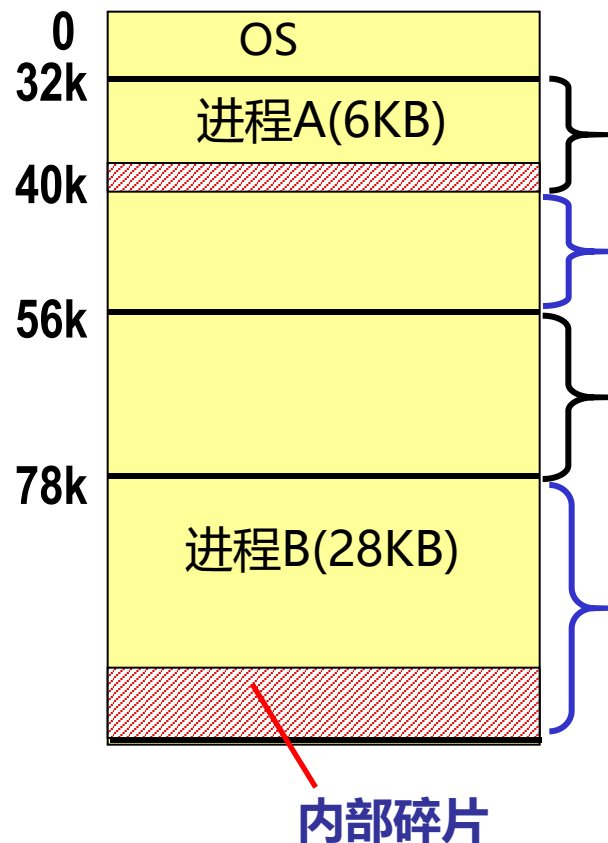
# 固定分区

分区使用表

| 区号 | 分区大小 | 起始地址 | 状态   |
|----|------|------|------|
| 1  | 8kB  | 32k  | Used |
| 2  | 16kB | 40k  | Free |
| 3  | 22kB | 56k  | Free |
| 4  | 34kB | 78k  | Used |

## 存在的问题:

- (1) 超过最大分区的程序无法装入;
- (2) 存在内部碎片(Internal fragmentation):  
由于程序长度小于分区, 使得分区内部有空间浪费。





# 连续分配

---

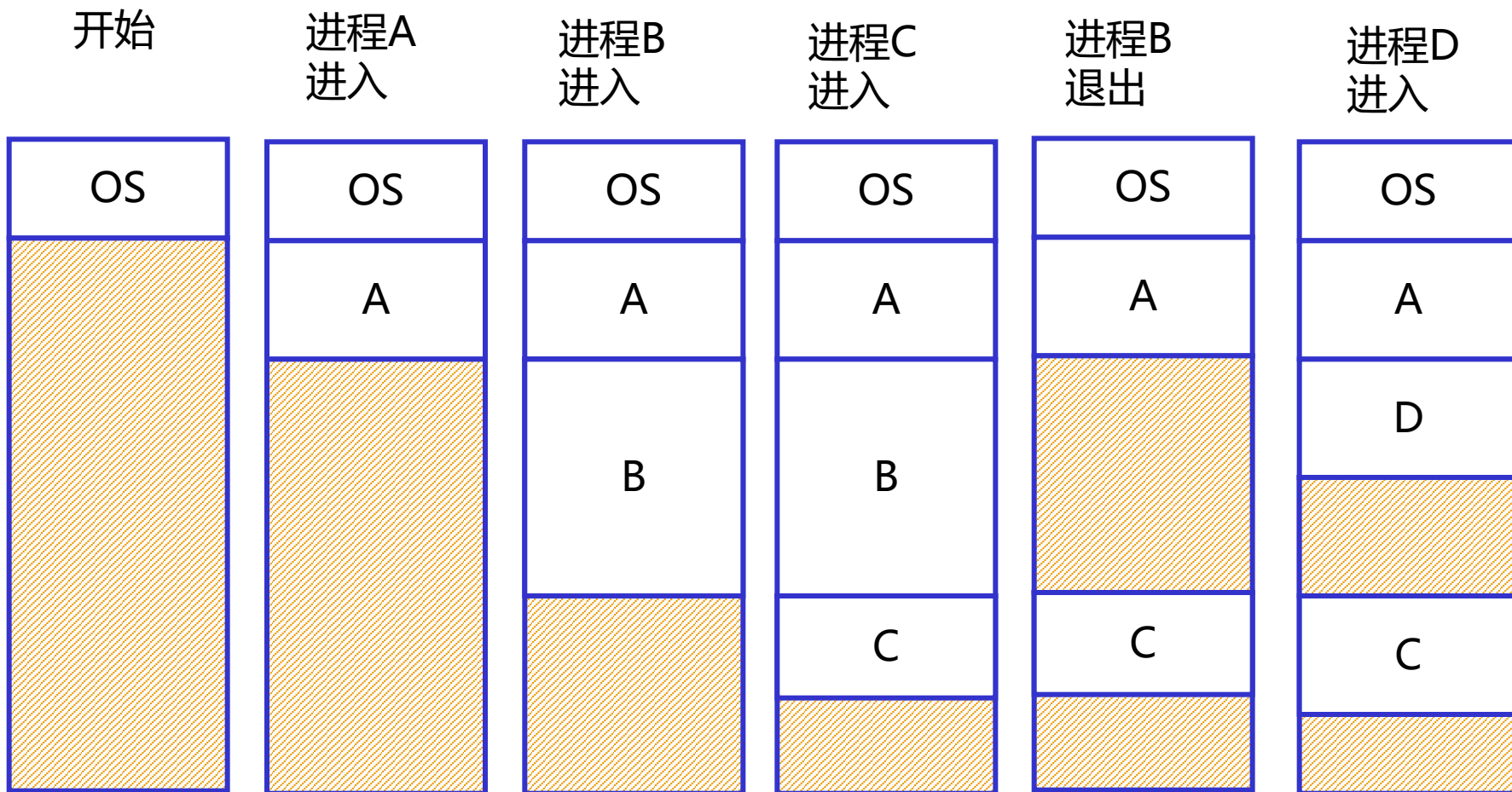
## 3. 可变分区 (Variable Partition)

或称动态分区。

开始时只有1个空闲分区，随着进程的装入和退出，分区的个数和每个分区的大小、位置会动态变化。



# 可变分区



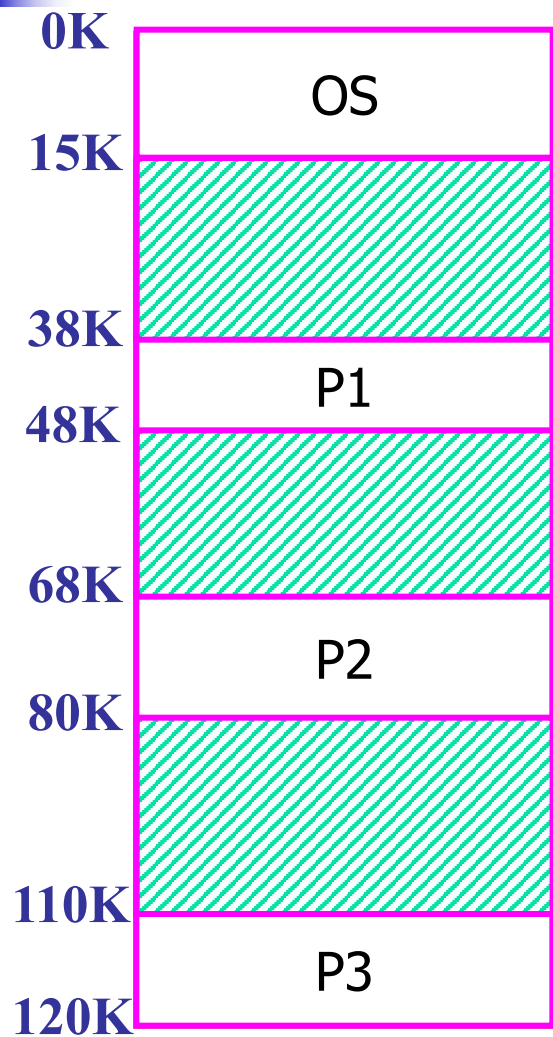
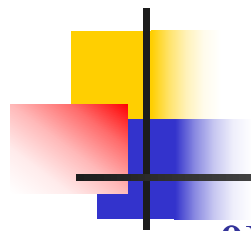


# 可变分区

---

## (1) 分区管理所需的数据结构

- ✓ 空闲分区表/空闲分区链  
记录空闲分区的起始地址和长度
- ✓ 已分配分区表

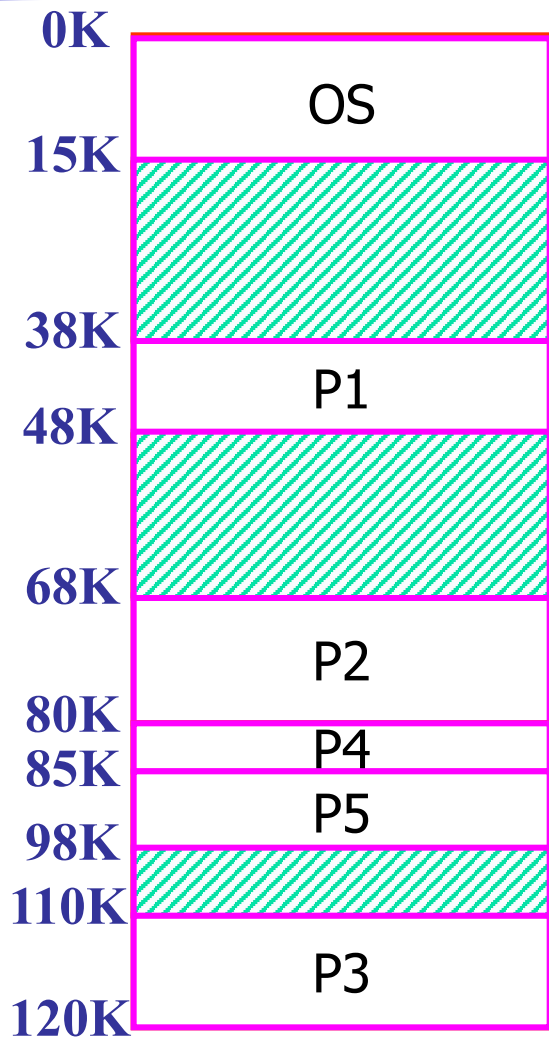
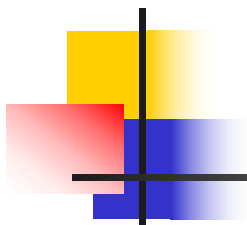


空闲分区表

| 始址  | 长度   | 状态  |
|-----|------|-----|
| 15K | 23KB | 未分配 |
| 48K | 20KB | 未分配 |
| 80K | 30KB | 未分配 |
|     |      | 空   |
|     |      | 空   |

已分配分区表

| 始址   | 长度   | 进程 |
|------|------|----|
| 38K  | 10KB | P1 |
| 68K  | 12KB | P2 |
| 110K | 10KB | P3 |
|      |      | 空  |
|      |      | 空  |



空闲分区表

| 始址  | 长度   | 状态  |
|-----|------|-----|
| 15K | 23KB | 未分配 |
| 48K | 20KB | 未分配 |
| 98K | 12KB | 未分配 |
|     |      | 空   |
|     |      | 空   |

已分配分区表

| 始址   | 长度   | 进程 |
|------|------|----|
| 38K  | 10KB | P1 |
| 68K  | 12KB | P2 |
| 110K | 10KB | P3 |
| 80K  | 5KB  | P4 |
| 85K  | 13KB | P5 |



# 可变分区

---

## (2) 分区分配算法

### ① 最先适配法 (first fit)

空闲分区链 (表) 按地址递增的次序排列

从头开始, 选择第1个大小足够的分区

### ② 下次适配法 (next fit)

从上次分配的分区的下一个开始, 选择第1个大小足够的分区

### ③ 最佳适配法 (best fit)

空闲分区链 (表) 按大小递增的次序排列

从头开始, 选择第1个大小足够的分区

### ④ 最差适配法 (worst fit)

空闲分区链 (表) 按大小递减的次序排列

从头开始, 选择第1个分区 (如果足够大)



# 可变分区

---

## (3) 分区的回收

- ✓ 将回收的分区插入到空闲分区链（表）的合适位置
- ✓ 合并相邻的多个空闲分区

**考虑：上邻、下邻、上下相邻、上下不相邻**

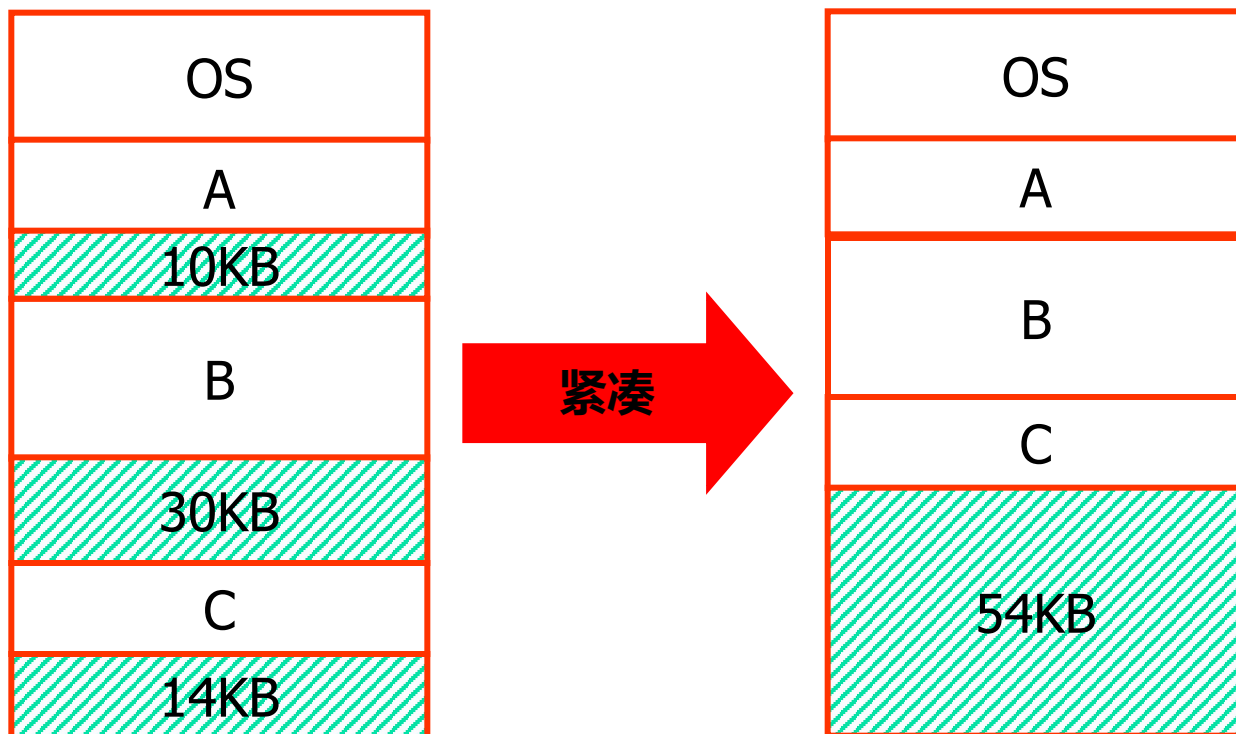
存在一个问题：**外部碎片**（External fragmentation）

经过一段时间的分配和回收后，内存中存在很多很小的空闲分区。它们每一个都很小，不足以满足分配要求，但其总和满足分配要求。这些空闲块被称为碎片，造成存储资源的浪费。

# 可变分区

## (4) 如何解决碎片问题?

**内存紧凑** (compaction) : 集中小碎片为大分区



涉及到程序在内存中的移动, 开销很大。



# 可变分区

---

为什么会产生大量碎片而难以利用呢？

**根本原因是：连续分配。**

如果把程序分成几部分装入不同分区呢？

**引入离散分配：**

- ✓ 分页
- ✓ 分段
- ✓ 段页式（分段 + 分页）





## 4.3 实存储器管理

---

### 二、分页 (Paging, 静态页式管理)

#### 1. 基本原理

(1) 等分内存为物理块 (或称页面, 页框, page frame)

物理块编号为0, 1, 2, . . . .

(2) 进程的逻辑地址空间分页 (page)

**页大小 = 物理块大小**

页编号为0, 1, 2, . . . .

(3) 内存分配原则:

**进程的1页可装入任一物理块**



# 分页

---

进程的最后1页可能装不满，产生“页内碎片”。

**分页类似于固定分区，但不同之处在于：**

- 1) 页比较小，且大小相等；
- 2) 一个进程可占据多页，且不要求连续。

进程的逻辑地址如何构成？

为实现分页管理，OS需要记录什么信息？

如何实现逻辑地址到物理地址的转换？



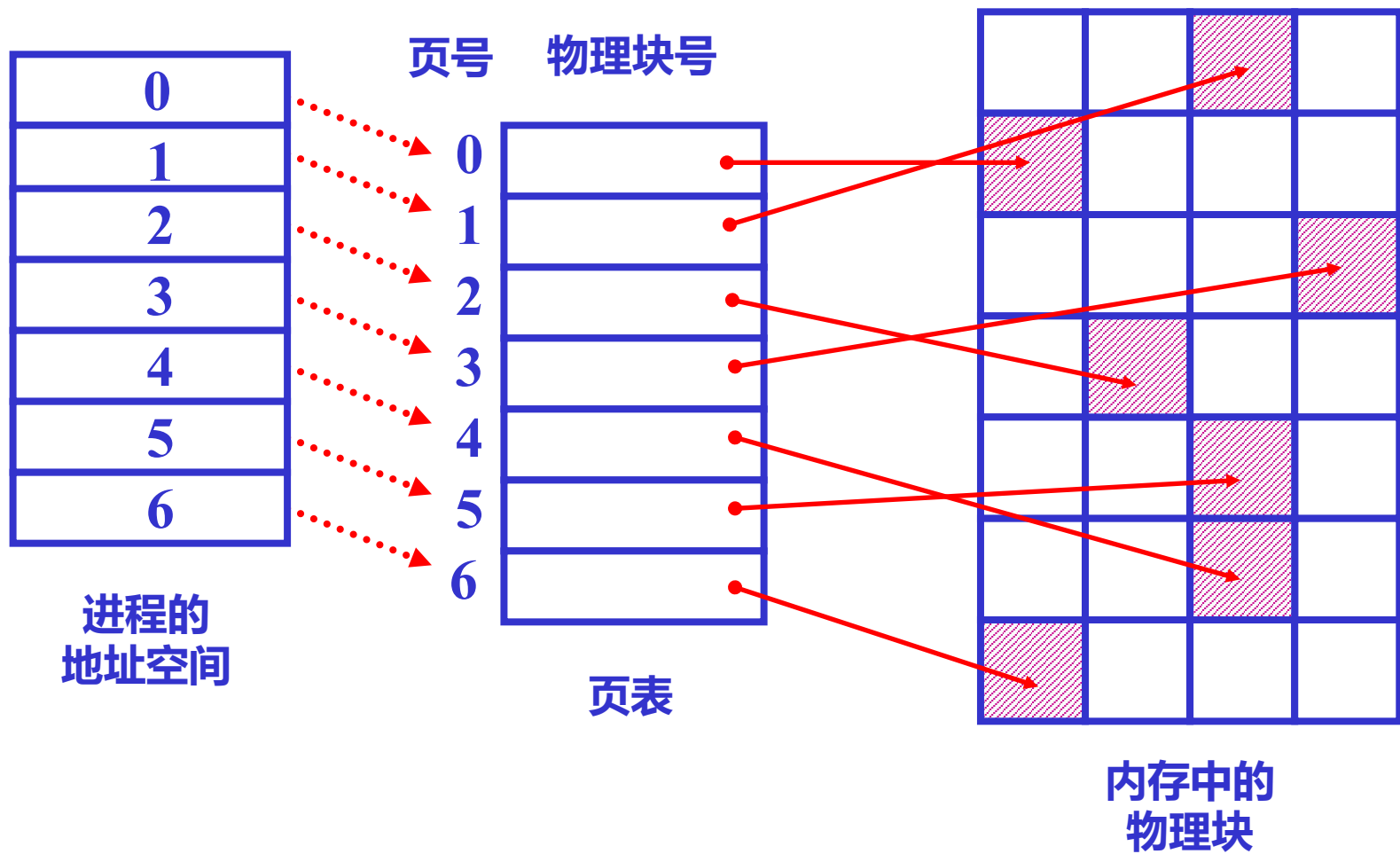
# 分页

---

## 2. 管理需要的数据结构

OS为每个进程建立1个**页表** (Page Table)

记录进程的页号和物理块号的对应关系



## 3. 地址转换

逻辑地址 → 物理地址

### (1) 页表寄存器 (Page Table Register, PTR)

系统设置1个页表寄存器，存放当前进程的页表起始地址和长度

每个进程的页表起始地址和长度平时放在其PCB中，调度时由OS放入PTR

# 分页

## (2) 页大小的选择

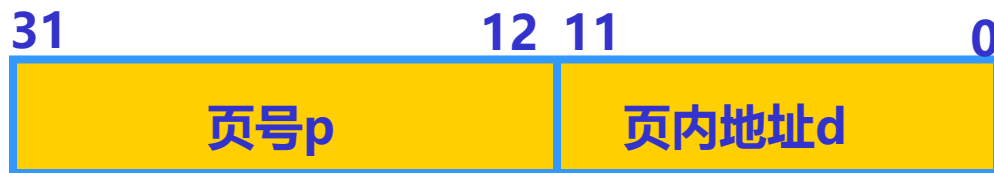
页小：碎片小，但页表占用空间大

页大：页表小，但页内碎片大

通常，页的大小为2的整数次幂

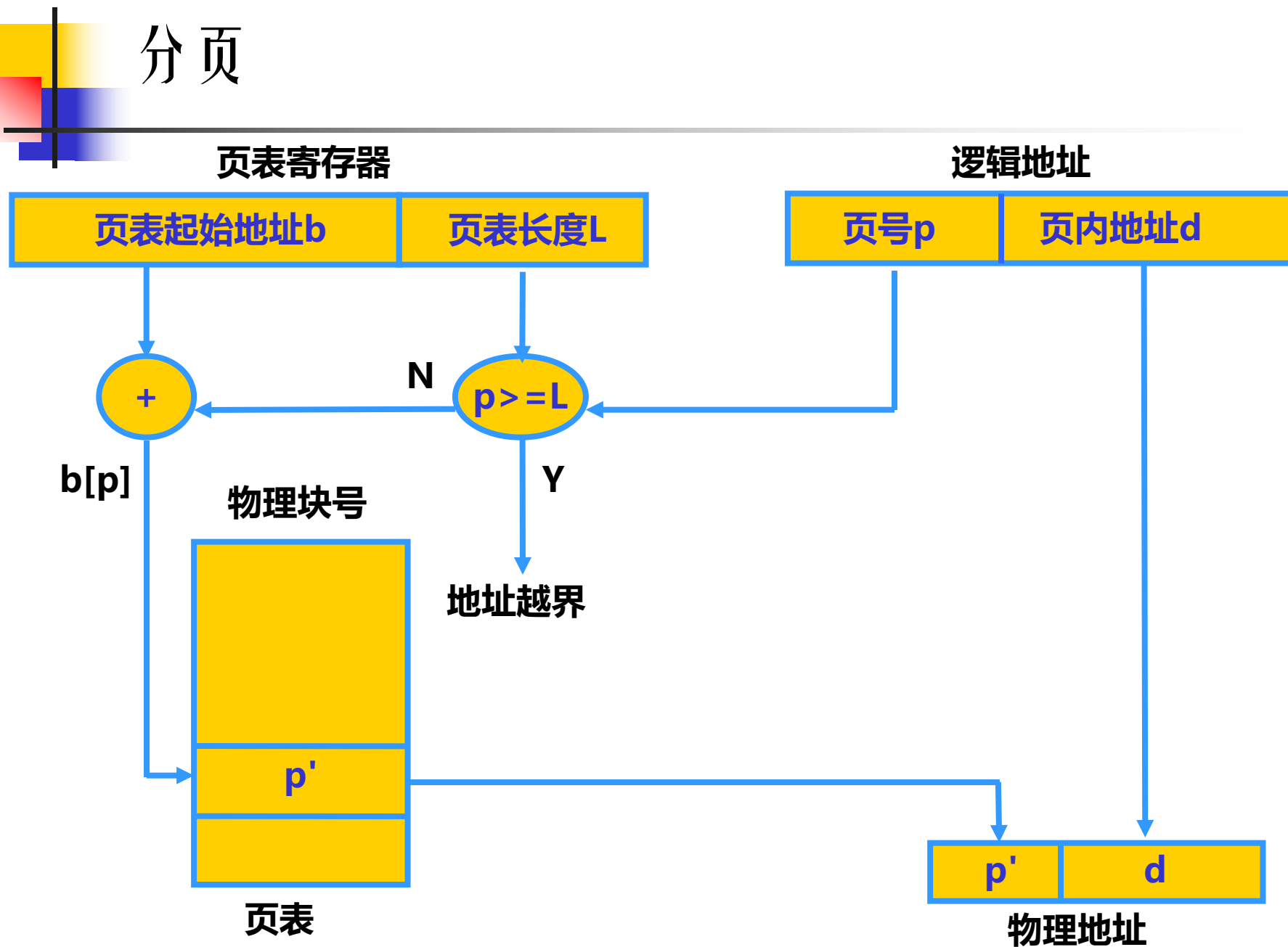
## (3) 进程的逻辑地址结构

地址的高位部分为页号，低位部分为页内地址



## (4) 基本的地址转换

# 分页



# 分页

【例】设逻辑地址是16位，  
页大小为 1KB = 1024B，  
即第0-9位是页内地址，第  
10-15位是页号。

逻辑地址空间

|       |
|-------|
| page0 |
| page1 |
| page2 |
| page3 |

页号 页表

|   |   |
|---|---|
| 0 | 1 |
| 1 | 4 |
| 2 | 3 |
| 3 | 7 |

块号 物理地址空间

|   |       |
|---|-------|
| 0 |       |
| 1 | page0 |
| 2 |       |
| 3 | page2 |
| 4 | page1 |
| 5 |       |
| 6 |       |
| 7 | page3 |
| 8 |       |

已知逻辑地址05DEH，即0000010111011110B，  
物理地址为：

11DEH，即0001000111011110B

因为页号 = 1，对应的物理块号 = 4

物理地址 = 块号 × 页大小 + 页内地址





# 基本的地址转换

---

CPU要存取一个数据时，需要访问内存几次？

2次。

- 第1次：访问页表，找到该页对应的物理块号，将此块号与页内地址拼接形成物理地址；
- 第2次：访问该物理地址，存取其中的指令或数据。



# 分页

---

## (5) 引入快表的地址转换

快表，又称联想存储器(Associative Memory)：

具有并行查找能力的特殊高速缓存（cache）。

在x86系统中叫做TLB（Translation lookaside buffers）

用途：保存当前进程的页表的子集（部分表项），比如最近访问过的页表项。

当切换到新进程时，快表要刷新。

目的：提高地址转换速度

# 分页

页表寄存器

逻辑地址

页表起始地址b

页表长度L

页号p

页内地址d

+

N

$p \geq L$

Y

快表

p

p'

$b[p]$

物理块号

地址越界

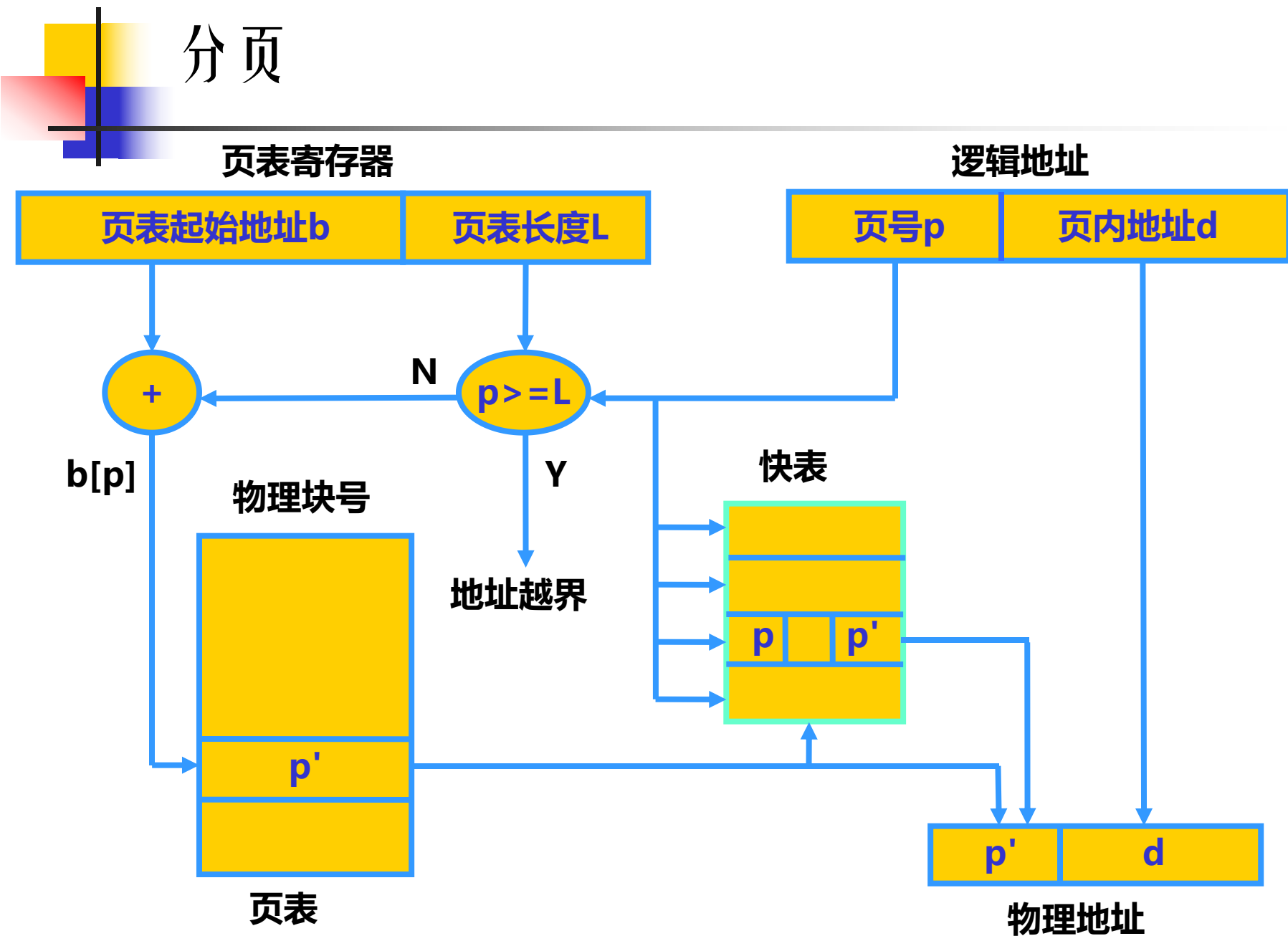
p'

页表

p'

d

物理地址





# 分页

---

## 快表表项:

- ✓ 页号：进程访问过的地址空间的页号
- ✓ 块号：该页所对应的物理块号
- ✓ 访问位：指示该页最近是否被访问过（0：没有被访问，1：访问过）
- ✓ 状态位：该快表项是否被占用（0：空闲，1：占用）

为了保证快表中的内容为现正运行进程的页表内容，**在每个进程被选中时，由恢复现场程序把快表的所有状态位清0，或恢复已保存的快表内容。**



# 分页

---

## 具有快表的地址转换过程：

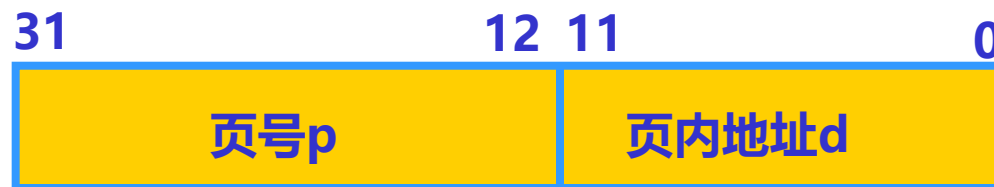
- ①当进程访问一页时，系统将页号与快表中的所有项进行并行比较。若访问的页在快表中，即可取得块号，生成物理地址。
- ②当被访问的页不在快表中时，就要根据页号查内存中的页表，找到对应的物理块号，生成物理地址。同时将页号与块号填入快表中，若快表已满，则置换其中访问位为0的一项。

说明：

- ✓ 由于成本的关系，快表不能太大；
- ✓ 引入快表的效果，取决于快表访问时的**命中率**(hit ratio)。

# 分页

**问题：**页表可能占用相当大的内存空间，而且是连续的。



设每个页表项占4B，页表最多可占用的内存空间是多少？

每个进程最多可达到 $2^{20}$  (1M) 个页

页表占用的内存空间 = 4MB

如何解决？



# 二级页表

## 4. 二级页表

### (1) 基本原理

将页表进行分页，页大小 = 物理块大小

设置1个一级页表，多个二级页表

一级页表：第*i*项记录第*i*号二级页表所在的物理块号

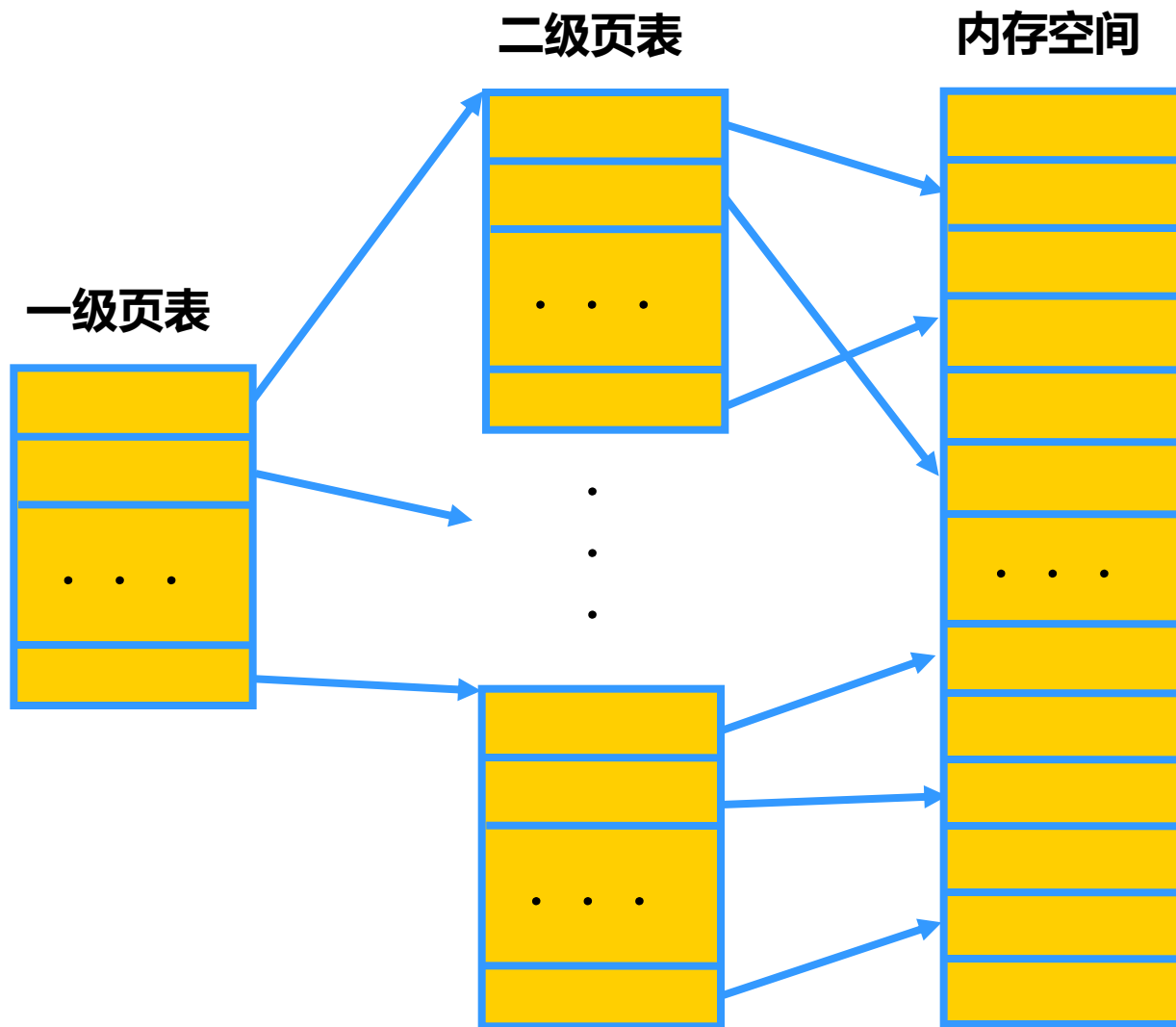
二级页表：第*i*项记录第*i*页对应的物理块号

系统设置 1 个页表寄存器，存放一级页表的起始地址和长度

### (2) 进程的逻辑地址结构：



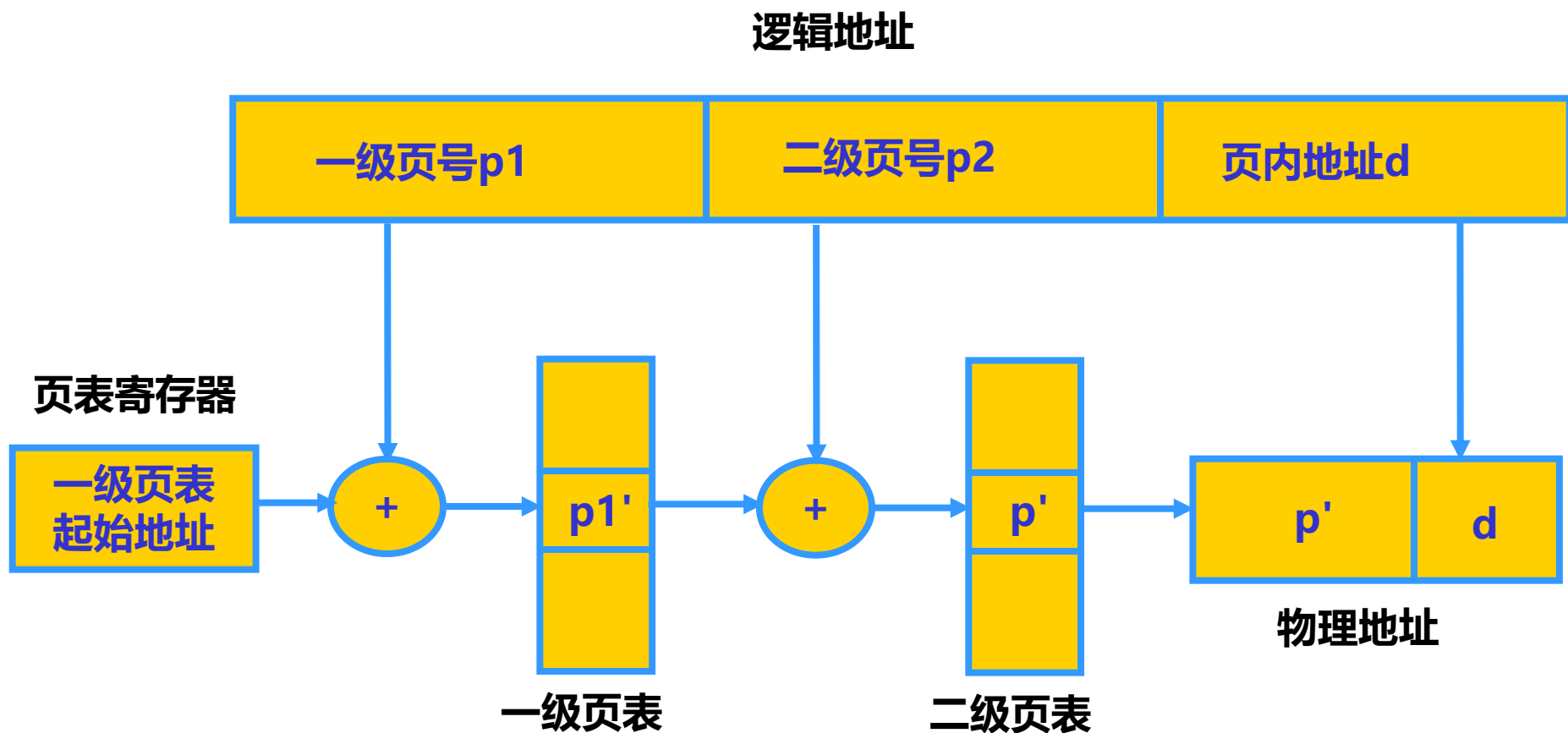
# 二级页表





# 二级页表

## (3) 地址转换





## 二级页表

---

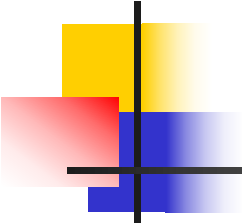
### 说明:

二级页表并未减少页表所占的内存空间，但解决了页表的离散分配问题。

对于32位地址，采用二级页表是可以的。

如果是64位地址呢？

需要三级甚至更多级的页表。



## 4.3 实存储器管理

---

### 三、分段 (Segmentation, 静态段式管理)

#### 1. 基本原理

(1) 进程的逻辑地址空间分段 (segment) :

按程序自身的逻辑关系划分为若干个程序段

每一个段是连续的

各个段的长度不要求相等

段号从0开始。

(2) 内存分配原则:

**以段为单位, 各段不要求相邻**



# 分段

---

**分段类似于可变分区，但不同之处在于：**

一个进程可占据多个分区，而且分区之间不要求连续。

分段无内部碎片，但有外部碎片。

进程的逻辑地址如何构成？

为实现分段管理，OS需要记录什么信息？

如何实现逻辑地址到物理地址的转换？



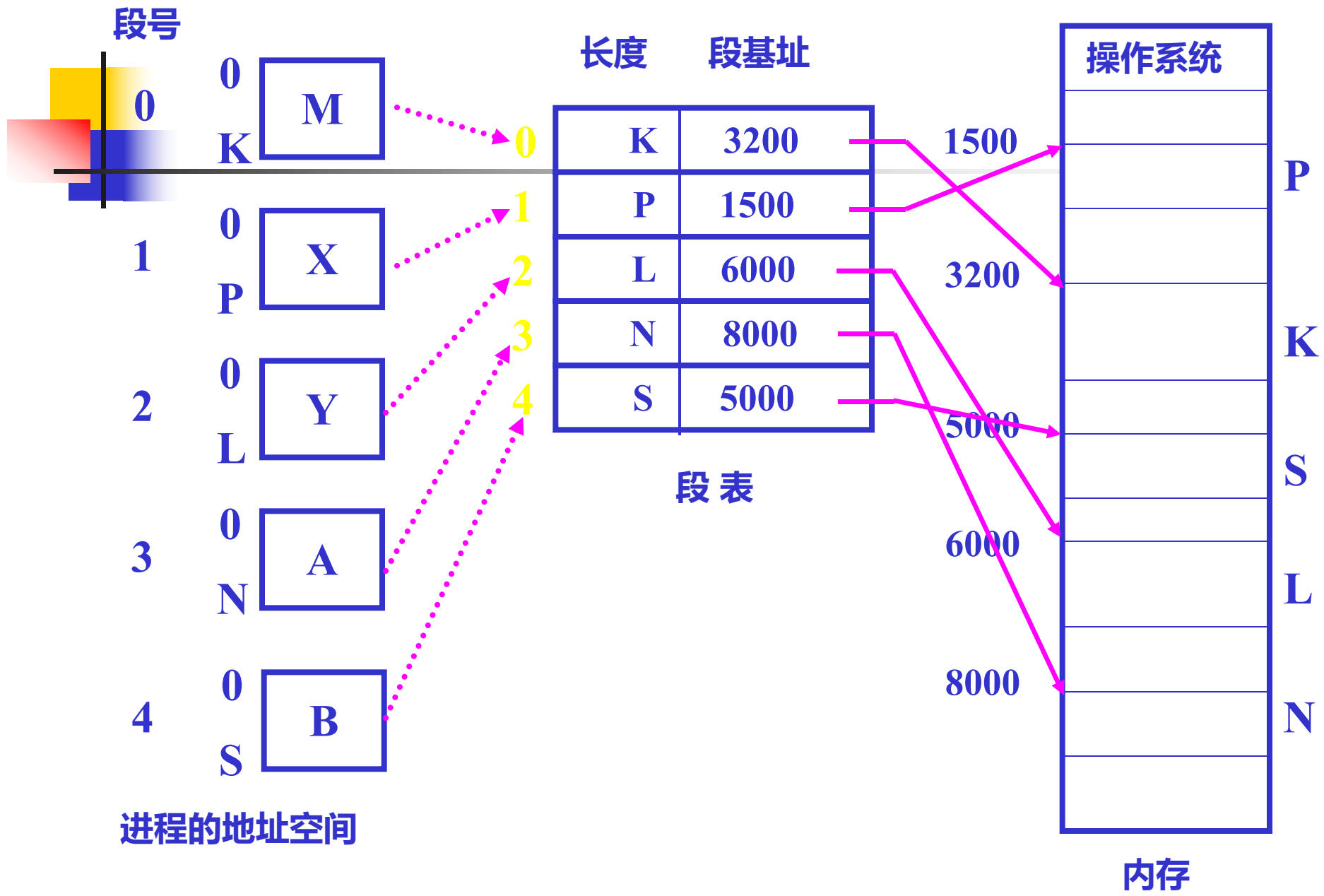
# 分段

## 2. 管理需要的数据结构

OS为每个进程建立1个**段表** (Segment Table)

每个段在段表中有1项，记录该段在内存中的基址和长度

| 段号 | 段基址     | 段长度  |
|----|---------|------|
| 0  | 20000H  | 20K  |
| 1  | 60000H  | 110K |
| 2  | 0A0000H | 140K |



# 分段

## 3. 地址转换

### (1) 段表寄存器

系统设置1个段表寄存器，存放**当前进程**的段表起始地址和长度  
每个进程的段表起始地址和长度平时放在其PCB中

### (2) 进程的逻辑地址结构

地址的高位部分为段号，低位部分为段内地址

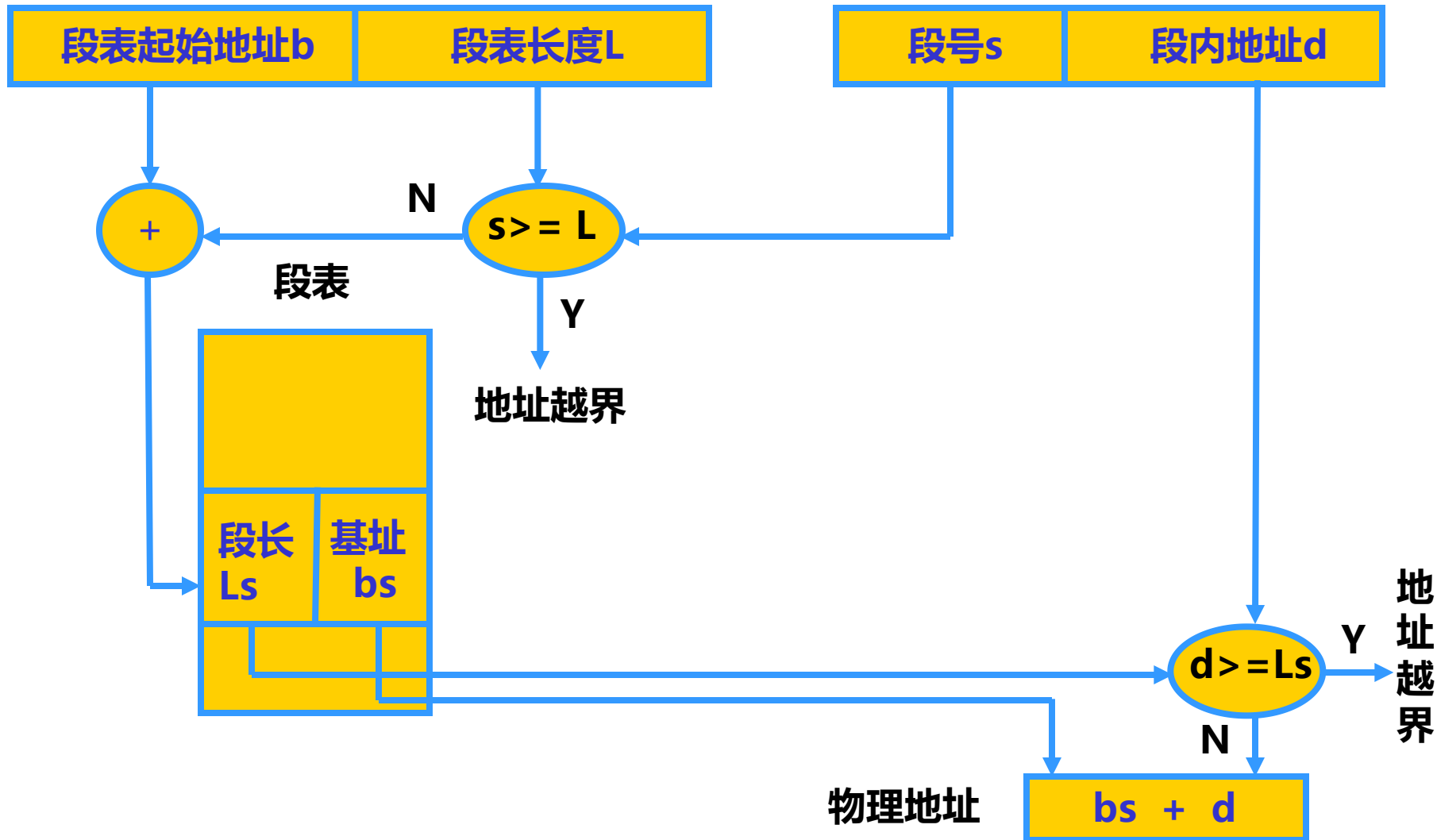


### (3) 基本的地址转换

# 分段

段表寄存器

逻辑地址







# 分段

【例】设逻辑地址是16位，段号占4位，段内地址占12位。

已知逻辑地址12f0h, 860h, 6000h

物理地址分别为：

12f0h: 物理地址=2310h

段号 = 1,

段基址 = 2020h, 段内地址 = 2f0h

860h, 6000h: 越界, 不能得到物理地址

| 段号 | 基址    | 长度    |
|----|-------|-------|
| 0  | 1000h | 400h  |
| 1  | 2020h | 0a00h |
| 2  | 4000h | 600h  |
| 3  | 6300h | 800h  |

段表



# 基本的地址转换

---

CPU要存取一个数据时，需要访问2次内存。

- 第1次：访问段表，找到该段的基址，将基址与段内地址相加形成物理地址；
- 第2次：访问该物理地址，存取其中的指令或数据。



# 分段

---

## (4) 引入快表的地址转换

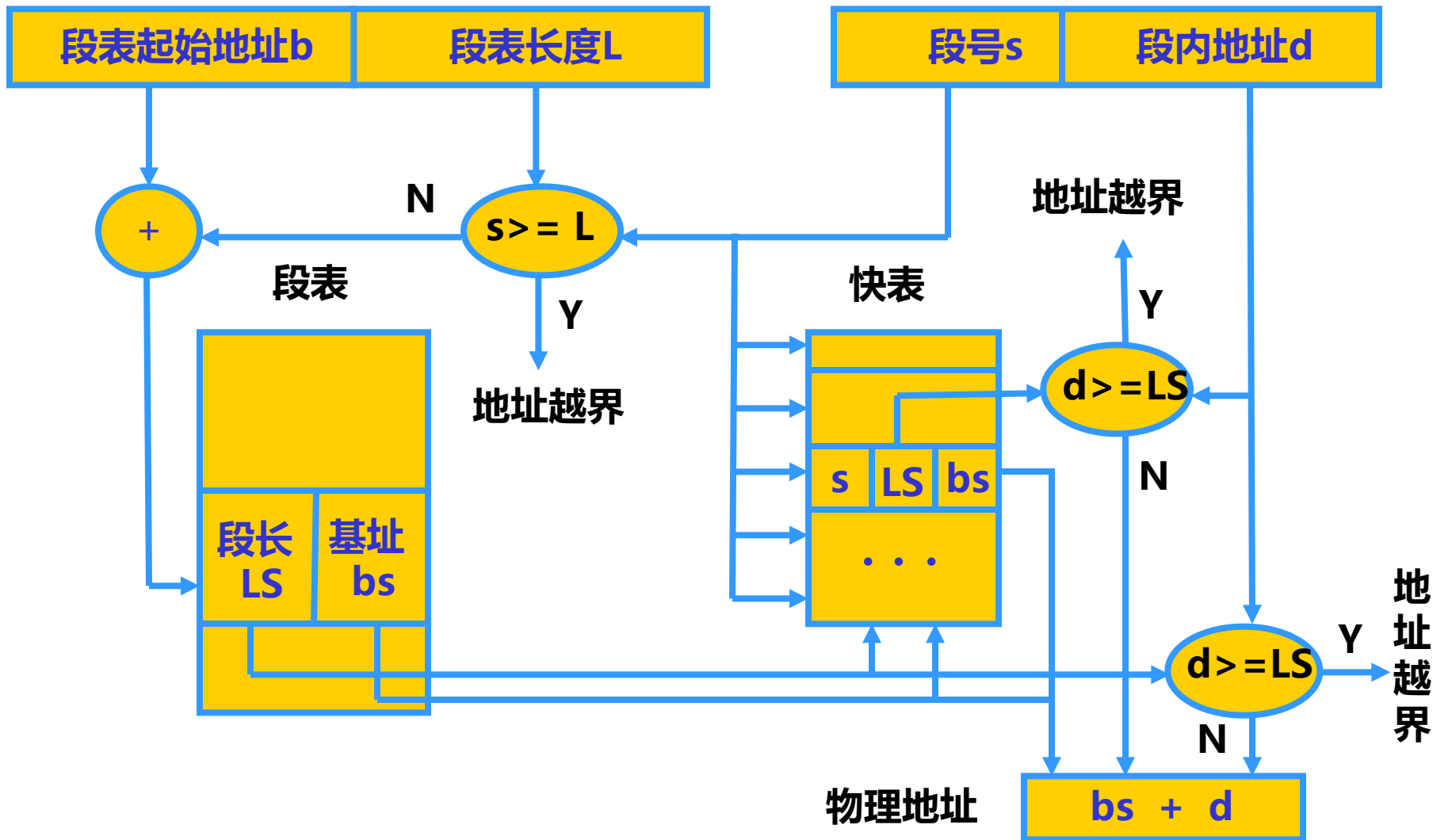
**快表表项:**

- ✓ 段号
- ✓ 段基址
- ✓ 段长度
- ✓ 访问位
- ✓ 状态位

# 分段

段表寄存器

逻辑地址





# 分段

---

## 4. 分段与分页的比较

- (1) **分页对程序员是不可见的；分段通常是可见的**，并作为组织程序和数据的手段提供给程序员；
- (2) **页的大小由系统确定，段的大小由用户程序确定**；
- (3) 分段更利于多个进程共享程序和数据；
- (4) 分段便于实现动态链接；
- (5) 分页可有效提高内存利用率，分段可更好地满足用户需要。

如果把分段和分页结合起来呢？



## 4.3 实存储器管理

---

### 四、段页式管理（分段 + 分页）

动机：结合分段和分页的优点，克服二者的缺点。

#### 1. 基本原理

(1) 进程分段：同段式管理

每段分页，内存分块，内存以块为单位分配：同页式管理

(2) 进程的逻辑地址结构

由段号、页号和页内地址构成。

|     |       |       |
|-----|-------|-------|
| 段号s | 段内页号p | 页内地址d |
|-----|-------|-------|



# 段页式管理

---

## 2. 管理需要的数据结构

- ✓ 每个进程1个段表

记录每个段对应的页表起始地址和长度

- ✓ 每个段有1个页表

记录该段所有页号与物理块号的对应关系



# 段页式管理

---

## 3. 地址转换

### (1) 段表寄存器

系统设置1个段表寄存器，存放[当前进程](#)的段表起始地址和长度  
每个进程的段表起始地址和长度平时放在其PCB中

### (2) 基本的地址转换

CPU要存取一个数据时，需要访问3次内存。

- 第1次：访问段表，获得该段的页表地址；
- 第2次：访问页表，取得物理块号，形成物理地址；
- 第3次：访问该物理地址，存取其中的指令或数据。





# 段页式管理

---

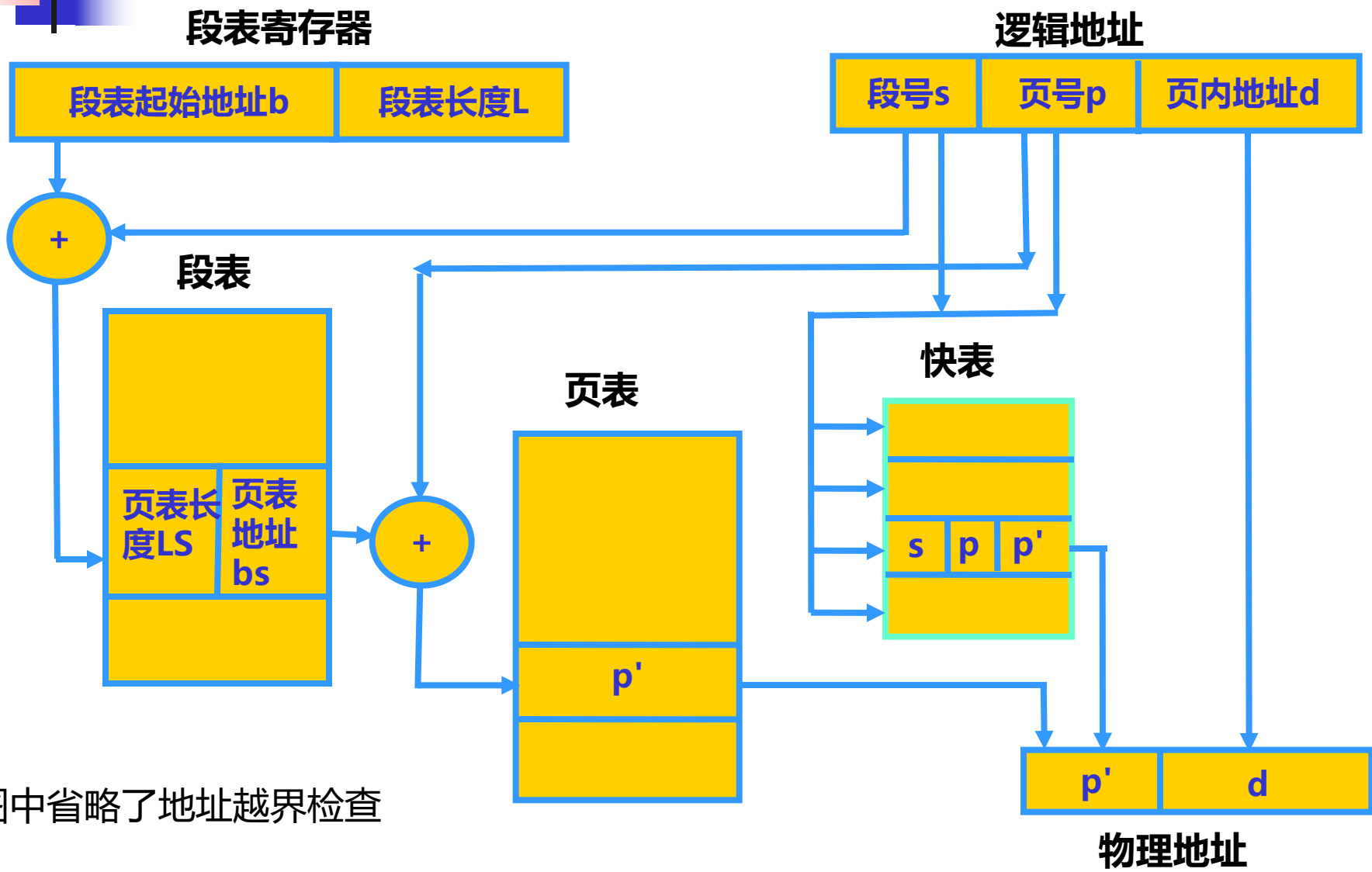
## (4) 引入快表的地址转换

**快表表项:**

- ✓ 段号
- ✓ 页号
- ✓ 物理块号
- ✓ 访问位
- ✓ 状态位

同时利用段号与页号查找相应的物理块号。

# 地址转换



图中省略了地址越界检查



## 4.3 实存储器管理

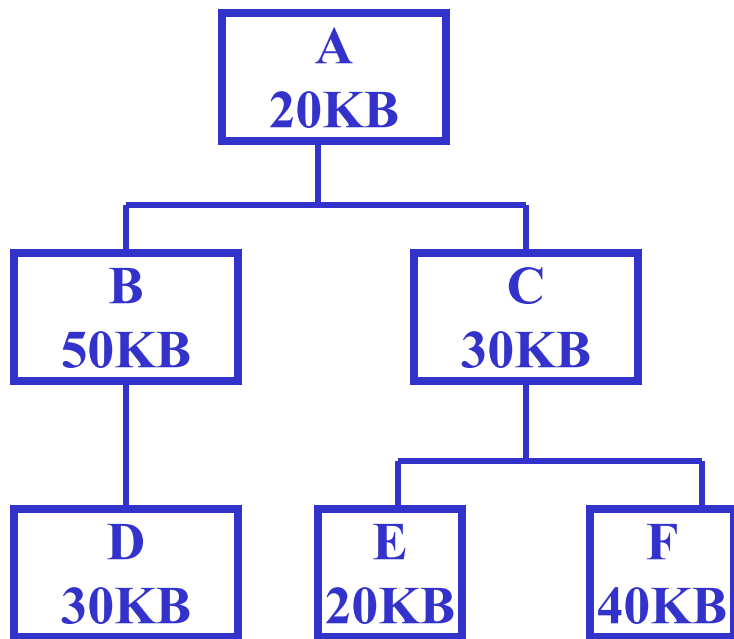
---

### 五、覆盖

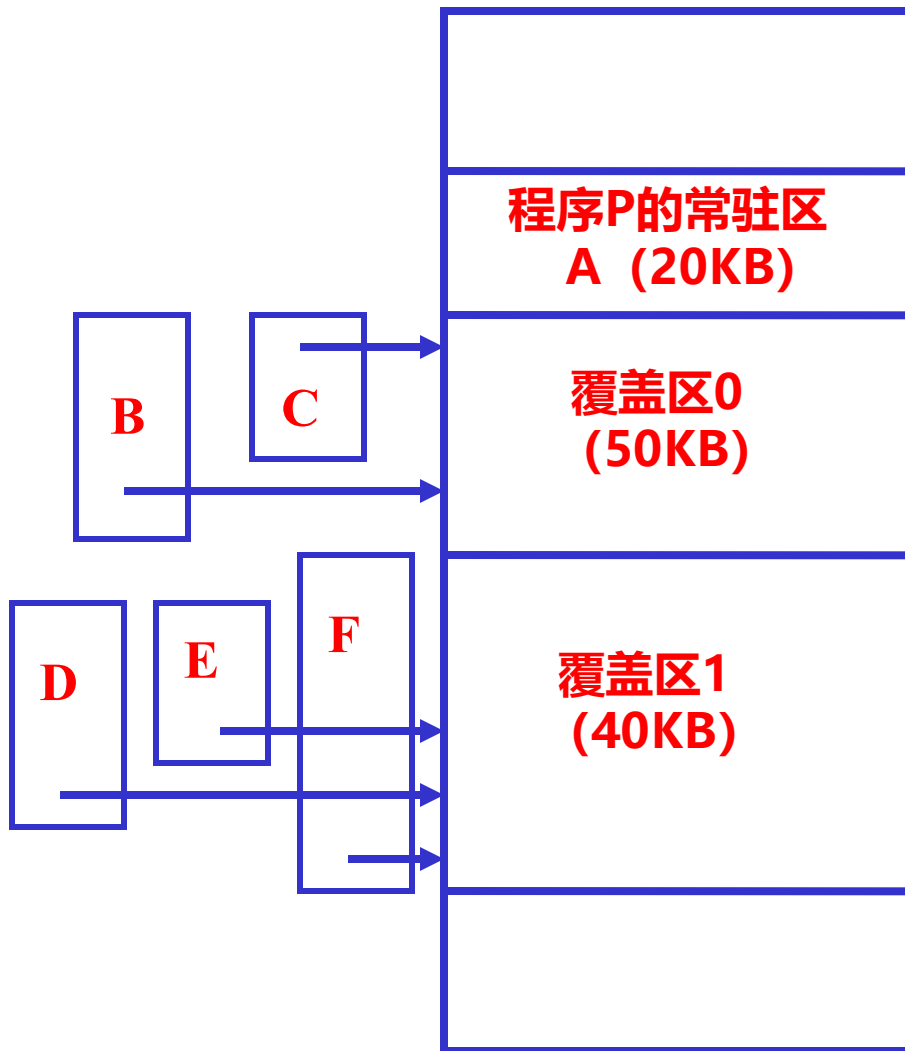
#### 基本原理:

- (1) 把程序划分为若干个功能上相对独立的程序段（称为覆盖块），按照其自身的逻辑结构使那些不会同时执行的程序段共享同一块内存区域；
- (2) 覆盖块存放在磁盘上，当一个程序段执行结束，把后续程序段调入内存，覆盖前面的程序段（内存“扩大”了）。

# 覆盖



程序P的调用结构





## 覆盖

---

### 缺点:

要求程序员划分程序，提供一个明确的覆盖结构；

对用户不透明，增加了用户负担。

这种技术曾用在早期的操作系统中。



## 4.4 虚拟存储管理

---

### 一、虚拟存储器 (Virtual Memory)

#### 1. 基本思想

进程的大小可以超过可用物理内存的大小，

由OS把当前用到的那部分留在内存，其余的放在外存中。

#### 2. 几个概念

- ✓ 虚拟地址：程序中使用的地址。进程的虚拟地址从0开始。
- ✓ 物理地址：可寻址的内存实际地址
- ✓ 虚拟地址空间：虚拟地址的集合
- ✓ 物理地址空间：实际的内存空间



# 虚拟存储器

---

## 3. 交换 (Swapping) 技术

借助于外存（磁盘），将当前要使用的那部分程序或数据装入内存，将暂时不需要的放在磁盘上，待需要时再装入。

**交换：**进程的整体或一部分的换入/换出。

**换入：**从磁盘移入内存

**换出：**从内存移出到磁盘

**交换是实现虚拟存储器的基础。**

最初的交换是针对整个进程的交换。



# 交换技术

---

## (1) 交换区（交换空间）的概念

磁盘上为虚拟内存保留的区域，称为交换区或交换空间。

## (2) 交换区的实现方式

- ✓ 交换分区，或称交换设备
- ✓ 交换文件，用于交换的有固定长度的文件

## (3) 交换区的分配方式

- ✓ 进程创建时分配：每次换出在同一个地方
- ✓ 换出时分配：首次换出时分配，以后换出在同一个地方；或者每次换出在不同地方。





# 虚拟存储器

---

## 4. 虚拟存储器的特征

逻辑上扩充了内存容量，对于用户程序来说，仿佛内存“无限大”。  
只不过有时慢一点而已。

## 5. 虚拟存储管理的实现方案

分页 + 虚拟存储技术

分段 + 虚拟存储技术

段页式 + 虚拟存储技术



## 4.4 虚拟存储管理

---

### 二、虚拟页式管理（动态页式管理）

#### 1. 基本思想

- ✓ 在进程开始运行之前，不是装入全部页，而是装入部分或0个页，之后根据进程运行的需要，动态装入其它页；
- ✓ 当内存空间已满，而又需要装入新的页时，则根据某种算法淘汰某个页，以便装入新的页。



# 虚拟页式管理

---

## 2. 页表项内容的扩充

除了页号和物理块号外，需要增加下列字段：

- ✓ 有效位（状态）：表示该页是否在内存中
- ✓ 访问位A（访问字段）：记录该页最近是否被访问过
- ✓ 修改位M：表示该页在装入内存后是否被修改过
- ✓ 外存地址：该页在磁盘上的地址



# 虚拟页式管理

---

## 3. 地址转换

虚拟地址 - > 物理地址

类似于静态页式管理。

由于访问的页p可能不在内存中，因而引出以下几个问题：

- ✓ 缺页中断处理
- ✓ 页的换入/换出



# 虚拟页式管理

---

## (1) 缺页中断 (Page Fault) 处理

在地址转换的过程中，当访问页表时，若根据状态位发现所访问的页不在内存，则产生缺页中断。

### 缺页中断处理：

- ① 保护当前进程现场；
- ② 根据页表中给出的外存地址，在外存中找到该页；
- ③ 若内存中无空闲物理块，则选择1页换出；
- ④ 分配一个空闲物理块，将新调入页装入内存；
- ⑤ 修改页表中相应表项的状态位及相应的物理块号，修改空闲物理块表（链）；
- ⑥ 恢复现场。



# 虚拟页式管理

---

## (2) 页的换入/换出

### 1) 页的分配策略：为每个进程分配多少个物理块

#### ✓ 固定分配

为每个进程分配的总物理块数固定，在整个运行期间不变。

#### ✓ 可变分配

先为每个进程分配一定数目的物理块，OS自身维持一个空闲物理块队列。

当发生缺页时，由系统分配一个空闲块，存入调入的页；

当无空闲块时，才会换出。



# 页的换入/换出

---

## 2) 页的置换策略：在什么范围内选择淘汰页

- ✓ 全局置换

从整个内存中选择淘汰页

- ✓ 局部置换

只从缺页进程自身选择淘汰页



# 页的换入/换出

---

## 3) 页的调入策略

### ① 何时调入

#### ✓ 请求调页 (demand paging)

只有访问的页不在内存中时，才会调入该页。

#### ✓ 预调页 (prepaging)

一次调入多个连续的页。为什么这样做？

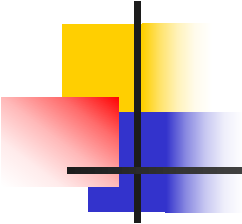
### ② 从何处调入：文件区（可执行文件）、交换区

#### ✓ 全部从交换区调入

进程创建时，全部从文件区拷贝到交换区。

#### ✓ 首次调入从文件区，以后从交换区





# 虚拟页式管理

---

## 4. 页的置换算法 (Page Replacement Algorithm)

作用：选择换出页

目的：减少缺页率

思路：以过去预测未来



# 页的置换算法

---

## (1) 最优置换算法 (Optimal, OPT)

**淘汰以后永不使用的，或者过最长的时间后才会被访问的页。**

这是Belady于1966年提出的一种理论上的算法。

显然，采用这种算法会保证最低的缺页率，但无法实现，因为它必须知道“将来”的访问情况。

因此，该算法只是作为衡量其他算法优劣的一个标准。



# 页的置换算法

---

## (2) 先进先出置换算法 (First In First Out, FIFO)

淘汰最早进入内存的页。

**实现方法：**

只需把进程中已调入内存的页，按先后次序链成一个队列即可。

优点：开销较小，实现简单。

缺点：

① 它与进程访问内存的动态特性不相适应；

② 会产生belady现象。即：当分配给进程的物理块数增加时，有时缺页次数反而增加。



# 先进先出置换算法

---

**【例】** 设系统为某进程在内存中固定分配 $m$ 个物理块，初始为空，进程访问页的走向为1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5。采用FIFO算法，当 $m=3$ ,  $m=4$ 时，缺页次数分别是多少？

$m=3$ 时，缺页次数：9次

$m=4$ 时，缺页次数：10次

产生Belady现象



# 页的置换算法

---

## (3) 最近最久未使用算法 (Least Recently Used, LRU)

**淘汰最近一次访问距当前时间最长的页。**

即淘汰未使用时间最长的页

关键：如何快速地判断出哪一页是最近最久未使用的。

算法较好，但实现代价高。



# LRU 算法

---

## 实现方法:

### 1) 计时器

对于每一页增设一个访问时间计时器

每当某页被访问时，当时的绝对时钟内容被拷贝到对应的访问时间计时器中，这样系统记录了内存中所有页最后一次被访问的时间。

淘汰时，选取访问时间计时器值最小的页。

### 2) 移位寄存器

为内存中的每一页配置一个移位寄存器

当访问某页时，将相应移位寄存器的最高位置1

每隔一定时间，寄存器右移1位

淘汰寄存器值最小的页。



# LRU 算法实现方法

---

## 3) 栈

每次访问某页时，将其页号移到栈顶。使得栈顶始终是最近被访问的页，栈底是最近最久未用的。



# LRU 算法实现方法

4, 7, 0, 7, 1, 0, 1, 2, 1, 2, 6

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   |   |   | 2 | 1 | 2 | 6 |
|   |   |   |   | 1 | 0 | 1 | 1 | 2 | 1 | 2 |
|   |   | 0 | 7 | 7 | 1 | 0 | 0 | 0 | 0 | 1 |
|   | 7 | 7 | 0 | 0 | 7 | 7 | 7 | 7 | 7 | 0 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 7 |

当访问页6时，发生缺页，淘汰处于栈底的页4。





# 页的置换算法

---

**【例】** 设系统为进程P固定分配3个物理块，初始为空，进程访问页的顺序为4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5。分别采用OPT、FIFO、LRU置换算法的情况下，计算缺页次数。



# 页的置换算法

---

|     |   |   |   |   |   |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|---|---|---|---|
| OPT | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页①  | 4 | 3 | 2 | 1 | 1 | 1 | 5 | 5 | 5 | 2 | 1 | 1 |
| 页②  |   | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 |
| 页③  |   |   | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
|     | x | x | x | x | √ | √ | x | √ | √ | x | x | √ |

缺页次数: 7次



# 页的置换算法

---

|      |   |   |   |   |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|---|---|---|---|
| FIFO | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页①   | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 5 | 5 | 2 | 1 | 1 |
| 页②   |   | 4 | 3 | 2 | 1 | 4 | 3 | 3 | 3 | 5 | 2 | 2 |
| 页③   |   |   | 4 | 3 | 2 | 1 | 4 | 4 | 4 | 3 | 5 | 5 |
|      | x | x | x | x | x | x | x | √ | √ | x | x | √ |

缺页次数: 9次



# 页的置换算法

---

|     |   |   |   |   |   |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|---|---|---|---|
| LRU | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页①  | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页②  |   | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 |
| 页③  |   |   | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 |
|     | x | x | x | x | x | x | x | √ | √ | x | x | x |

缺页次数: 10次



# 页的置换算法

---

## (4) 最近未使用算法 (Not Recently Used, NRU)

### 1) 简单的NRU算法

页表中设置一个访问位，当访问某页时，将访问位置1

将内存中的所有页链成一个循环队列

从上次换出页的下一个位置开始扫描

在扫描过程中，将访问位=1的页清0，直到遇到访问位=0的页，淘汰该页，并将指针指向下一页

NRU算法是用得较多的LRU的近似算法

由于该算法循环检查各页的访问位，故称Clock算法。

问题：如果考虑页是否已被修改，选择什么页换出好？



# 页的置换算法

---

## 2) 改进的NRU算法 (改进的Clock算法)

该算法的提出基于如下考虑:

- ① 对于已修改的页, 换出时必须重新写回到磁盘上。因此, 优先选择未访问过、未修改过的页换出;
- ② 淘汰一个最近未访问的已修改页要比淘汰一个被频繁访问的“干净”页好。



# 改进的NRU算法

---

## 基本方法:

设置1个访问位A, 1个修改位M

按照下列次序选择淘汰页:

第1类:  $A = 0, M = 0$ : 未访问, 未修改; 最佳淘汰页

第2类:  $A = 0, M = 1$ : 未访问, 已修改

第3类:  $A = 1, M = 0$ : 已访问, 未修改; 有可能再次访问

第4类:  $A = 1, M = 1$ : 已访问, 已修改



# 改进的NRU算法

---

## 实现算法：

- ① 找第1类页，将遇到的第1个页作为淘汰页；
- ② 若查找1周后未找到第1类页，则寻找第2类页，并将扫描经过的页的访问位清0。将遇到的第1个页作为淘汰页；
- ③ 否则，转①、②，一定能找到淘汰页。





# 页的置换算法

---

## (5) 最少使用算法 (Least Frequently Used, LFU)

选择最近访问次数最少的页淘汰。

实现方法：

通常不直接利用计数器来记录页的访问次数，而是采用移位寄存器R。

类似于LRU算法，每次访问某页时，将其移位寄存器的最高位置1。每隔一定时间将移位寄存器右移1位。

这样，在最近一段时间内使用次数最少的页就是移位寄存器各位之和最小的页。

当然，这并不能真正反映出页的访问次数。因为，在每一时间间隔内，只用1位来记录页的访问情况，访问1次和100次是等效的。



# 页的置换算法

---

## (6) 页缓冲算法 (Page Buffering Algorithm)

出发点：与上述算法配合使用，以提高效率。

### 基本方法：

设置2个链表：空闲页链表，已修改页链表。为什么？

- ① 若淘汰页未修改，则直接放入空闲页链表，否则放入已修改页链表；
- ② 当已修改页达到一定数量时，再将其一起写回磁盘，即**成簇写回**，以减少I/O操作的次数。



# 虚拟页式管理

---

## 5. 局部性原理

### (1) 程序的局部性特征

程序在执行过程中的一个较短时期，所执行的指令地址和操作数地址，分别局限于一定区域。表现在时间与空间两方面。

#### ✓ 时间局部性

一条指令被执行了，则在不久的将来它可能再被执行；数据也类似。例如循环。

#### ✓ 空间局部性

若某一存储单元被访问，则在一定时间内，与该存储单元相邻的单元可能被访问。如程序的顺序结构、数组的处理。



# 局部性原理

## (2) 程序结构对性能的影响

**【例】** 将二维整型数组a[256][256]的每个元素初始化为0的C语言程序:

程序1:

```
int a[256][256];  
for (i = 0; i < 256; i++)  
    for (j = 0; j < 256; j++)  
        a[i][j] = 0;
```

程序2:

```
int a[256][256];  
for (j = 0; j < 256; j++)  
    for (i = 0; i < 256; i++)  
        a[i][j] = 0;
```

假定在分页系统中, 页的大小为1KB, int类型占4B, 设分配给该程序1个物理块。这两个程序哪个好? 忽略该程序代码所占的内存空间。

**程序1好。** 因为C数组在内存中以行主次序存放, 程序1的缺页次数是256, 程序2的缺页次数为 $256 \times 256 = 65536$ 。



# 局部性原理

---

缺页率与程序的行为有关。

因此，在设计程序时，要力求提高程序访问的局部性。

**局部性特征正是虚拟存储技术能有效发挥作用的基础。**



# 虚拟页式管理

---

## 6. 抖动 (颠簸)

指页在内存与外存之间频繁换入/换出，以至于调度页所需时间比进程实际运行的时间还多，此时系统效率急剧下降，甚至导致系统崩溃。这种现象称为颠簸或抖动。

### 导致抖动的原因：

- ✓ 页置换算法不合理
- ✓ 分配给进程的物理块数太少



# 虚拟页式管理

---

## 7. 工作集(Working Set)

是由Denning提出并加以推广的，对于虚拟存储管理有着深远的影响。

**一个进程在时刻 $t$ 、参数为 $\Delta$ 的工作集 $W(t, \Delta)$ ，表示该进程在过去的 $\Delta$ 个时间单位中被访问到的页的集合。**

$\Delta$ 称为工作集的窗口大小。

**工作集的内容取决于三个因素：**

- ✓ 访页序列特性
- ✓ 时刻 $t$
- ✓ 观察该进程的时间窗口大小( $\Delta$ )



# 工作集

---

## 工作集的基本思想：

根据程序的局部性原理，一般情况下，进程在一段时间内总是集中访问一些页，这些页称为活动页。

如果分配给一个进程的物理块数太少了，使该进程所需的活动页不能全部装入内存，则进程在运行过程中将频繁发生缺页中断。

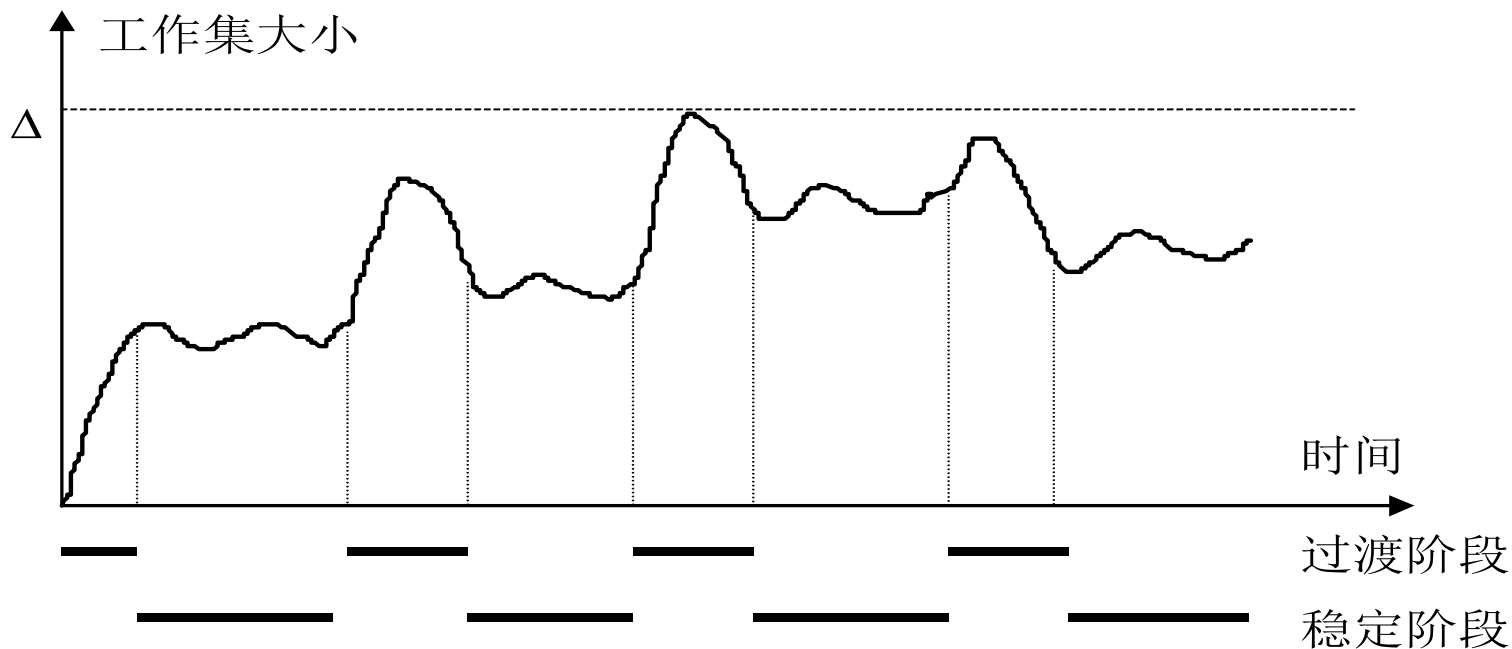
如果能为进程提供与活动页数相等的物理块数，则可减少缺页中断次数。



# 工作集

## 工作集大小的变化:

进程开始执行后，随着访问新页逐步建立较稳定的工作集。当内存访问的局部性区域的位置大致稳定时，工作集大小也大致稳定；局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值。





# 工作集

---

## 引入工作集概念的意义:

- ✓ 可以指导分配给每个进程多少个物理块，以及如何动态调整各进程的物理块数
- ✓ 通过监视每个进程的工作集，周期性地从一个进程驻留在内存的页的集合中移去那些不在其工作集中的页



# 虚拟页式管理

---

## 8. 影响缺页次数的因素

### (1) 分配给进程的物理块数

一般来说，进程的缺页中断率与进程所占的内存块数成反比。

分配给进程的内存块数太少是导致抖动现象发生的最主要原因。

### (2) 页的大小

缺页中断率与页的大小成反比，但页的大小也不能一味地求大，它一般在0.5KB~4KB之间，是个实验统计值。因为页大时，页表较小，占空间少，查表速度快，缺页中断次数少，但页内碎片较大。页小时，恰恰相反。

### (3) 程序的编写方法

进程的缺页中断率与程序的局部性（包括时间和空间局部性）程度成反比。

用户程序编写的方法不合适可能导致程序运行的时空复杂度高，缺页次数多。

### (4) 页置换算法

算法不合理会引起抖动。



## 4.4 虚拟存储管理

---

### 三、虚拟段式管理

#### 1. 段表项的扩充

- ✓ 段号
- ✓ 段基址、长度
- ✓ 访问位（访问字段）
- ✓ 修改位
- ✓ 有效位（状态）
- ✓ 外存地址
- ✓ 存取方式：读/写/执行
- ✓ 扩充位（增长位）：固定长/可扩充



# 虚拟段式管理

---

## 2. 地址转换

类似于静态段式管理，但可能产生缺段中断。

## 3. 缺段中断处理

检查内存中是否有合适的空闲区

①若有，则装入该段，修改有关数据结构，中断返回；

②若没有，则检查内存中空闲区的总和是否满足要求，是则应采用紧缩技术，转①；否则，淘汰一（些）段，转①。



# 虚拟段式管理

---

## 4. 段的置换

可用页式管理的置换算法。但一次调入时所需淘汰的段数与段的大小有关。

## 5. 分段的保护措施

### (1) 越界检查

由段表长度、每段长度控制，保证每个进程只访问自己的地址空间。

然而，进程在执行过程中，有时需要扩大分段，如数据段。由于要访问的地址超出原有的段长，所以触发越界中断。操作系统处理越界中断时，首先判断该段的“扩充位”，如可扩充，则增加段的长度；否则按出错处理。

### (2) 访问控制检查

根据段表项的“存取方式”字段进行控制。



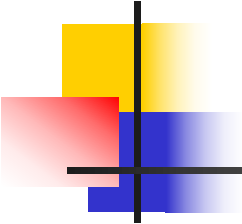
## 4.4 虚拟存储管理

---

### 四、虚拟段页式管理

涉及缺段中断、缺页中断。

注意的是，对于缺段中断处理，主要是在内存中建立该段的页表，而非调入完整的一段。



## 4.5 小结

---

### 一、存储器的层次结构及其思想

- ✓ 成本、容量、速度之间的权衡
- ✓ 较小、较贵的存储器由较大、较便宜的慢速存储器作为后援：虚拟存储器
- ✓ 降低较大、较便宜的慢速存储器的访问频率：高速缓存

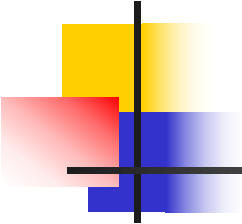
### 二、程序的连接与装入

- ✓ 程序编译、连接的功能以及与操作系统（硬件）的关系
- ✓ 程序装入的方式、局限性、对硬件的要求（包括在虚拟存储管理下的装入）

### 三、内存管理有关的重要概念及其意义

- ✓ 局部性原理、页缓冲、交换区
- ✓ 抖动、工作集
- ✓ . . .



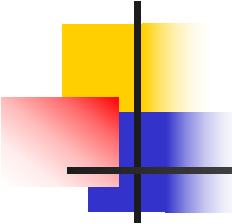


## 4.5 小结

---

### 四、典型内存管理机制（连续分配、离散分配；虚拟存储器）

- ✓ 基本原理、逻辑（虚拟）地址结构
- ✓ 数据结构
- ✓ 进程运行时的内存访问过程（包括快表、地址转换、缺页中断等）
- ✓ 对硬件的要求
- ✓ 局限性
- ✓ **注意整体理解**（连续分配能否实现虚拟存储器？能的话如何实现？不能的话为什么？）



## 4.5 小结

---

### 五、针对特定需求，能设计合理的内存管理方案

- ✓ 培养问题分析的意识
- ✓ 方案的基本思路
- ✓ 数据结构（逻辑结构，物理结构的表示）
- ✓ 内存分配与释放算法
- ✓ 进程运行时的内存访问过程
- ✓ 所设计方案的有效性分析

### 六、内存管理与进程管理（包括进程调度）的关系

- ✓ 内存管理是一种调度（资源分配）
- ✓ 与文件系统也有关系
- ✓ 进程从创建到结束需要OS完成的工作



# 本章作业

---

教材《计算机操作系统教程（第4版）》

p135:

5.9, 5.11



# 第5章 进程与内存管理实例

---

**5.1 Linux简介**

**5.2 Linux进程管理**

**5.3 Linux内存管理**

**5.4 Windows 2000/XP的进程与内存管理**

**5.5 小结**



## 5.1 Linux简介

---

1987年, MINIX操作系统:

Andrew S. Tanenbaum设计, 类Unix OS, 用于教学

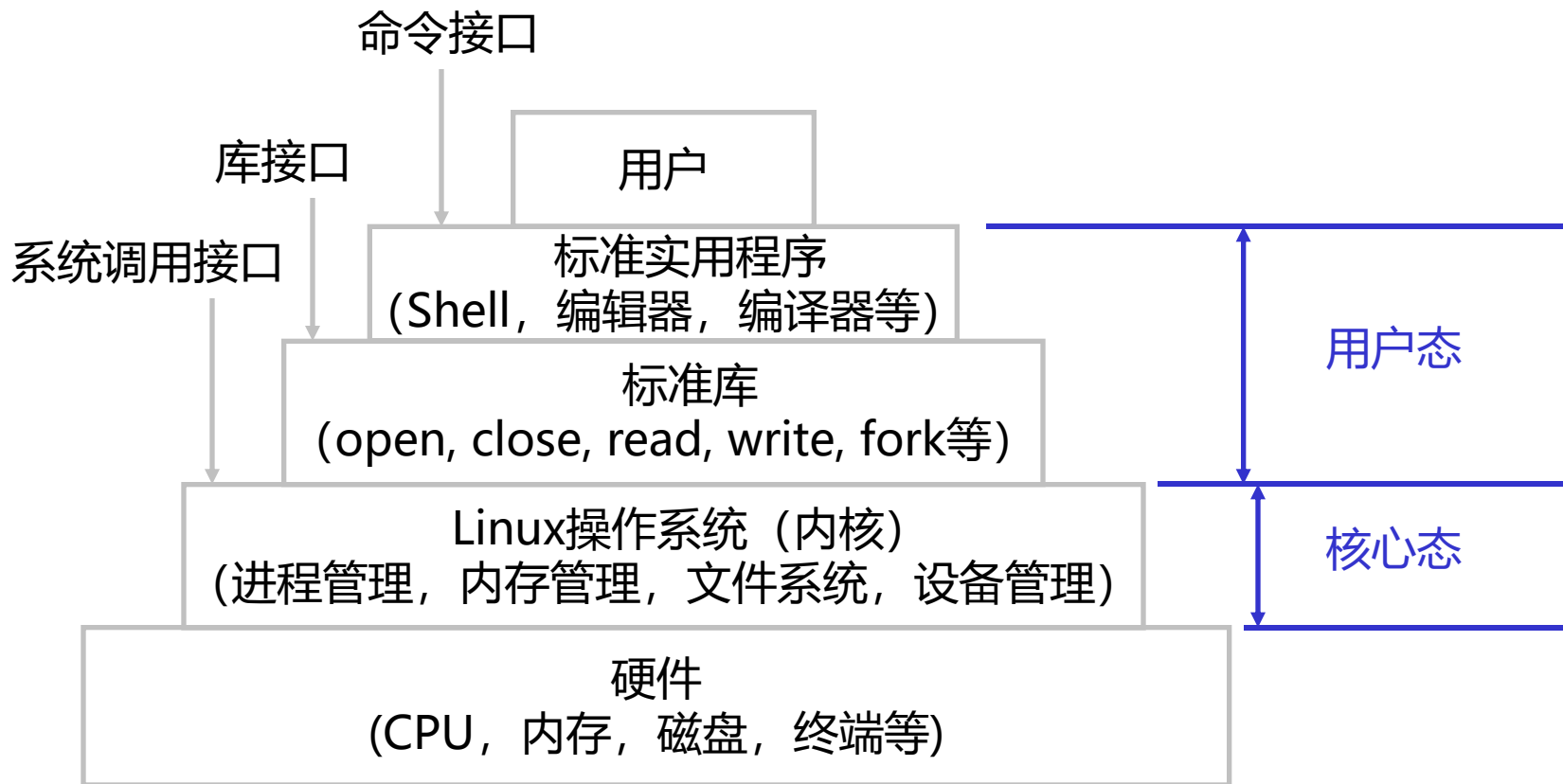
1991年, Linux操作系统

芬兰赫尔辛基大学生Linus Torvalds开发

- ✓ 与Unix兼容: 具有全部Unix特征, 遵从POSIX标准
- ✓ 自由软件, 源代码公开
- ✓ 多用户、多任务操作系统

# 5.1 Linux 简介

## 一、Linux的层次结构





# Linux的层次结构

---

## 1. 进程执行的2个级别(模式)

### (1) 用户态

用户态下的进程只能存取自己的指令和数据，不能执行特权指令

### (2) 核心态

核心态下的进程可以访问OS核心和用户进程的地址空间

### 说明:

- ✓ 虽然系统在执行时处于两种状态之一，但核心是为用户进程运行的；
- ✓ **核心不是与用户进程平行运行的孤立的进程集合，而是每个用户进程的一部分。**
- ✓ **用户进程通过系统调用由用户态切换到核心态。**



# Linux的层次结构

---

## 2. Linux的3个接口

### (1) 系统调用接口

用户在程序中（一般用汇编语言）调用OS提供的一些功能

### (2) 库接口

以高级语言（C语言）标准库函数的方式，为程序员调用OS提供的功能

### (3) 命令接口

在操作系统与计算机用户之间提供的双向通信机制

接口包括：一组联机命令，终端处理程序，命令解释程序等。

实现方式：

- ✓ 命令行方式：要求用户记忆命令格式。
- ✓ 图形用户接口（GUI）方式：用户可利用鼠标对屏幕上的图标进行操作，完成与操作系统的交互，从而减少记忆内容，方便用户使用。





# Linux的层次结构

---

## 3. 系统调用

### (1) 什么是系统调用?

用户在程序中调用操作系统所提供的一些功能;

系统调用是用户请求OS服务的途径。

- ✓ 系统调用本身也是由若干指令构成的过程（函数），但与一般过程的区别在于：系统调用运行在核心态，一般过程运行在用户态
- ✓ 系统调用发生在用户进程通过特殊函数（如open）请求OS内核提供服务的时候
- ✓ 系统调用负责对内核所管理资源的访问
- ✓ 每个操作系统都提供许多种系统调用



# 系统调用

---

## (2) 系统调用的方式

为了保证OS不被用户程序破坏，不允许用户程序访问OS的系统程序和数据。

那么，**如何得到系统服务呢？**

通过执行一条**陷入指令**（或称访管指令），**由用户态切换到核心态**，转入执行相应的处理程序（陷入处理程序）。

### 说明:

- ✓ 系统调用是一个低级过程，只能由汇编语言直接访问；
- ✓ 系统调用是操作系统提供给程序设计人员的唯一接口；
- ✓ C语言为每个系统调用提供对应的标准库函数，应用程序可以通过C语言库函数来使用系统调用。



# 系统调用

---

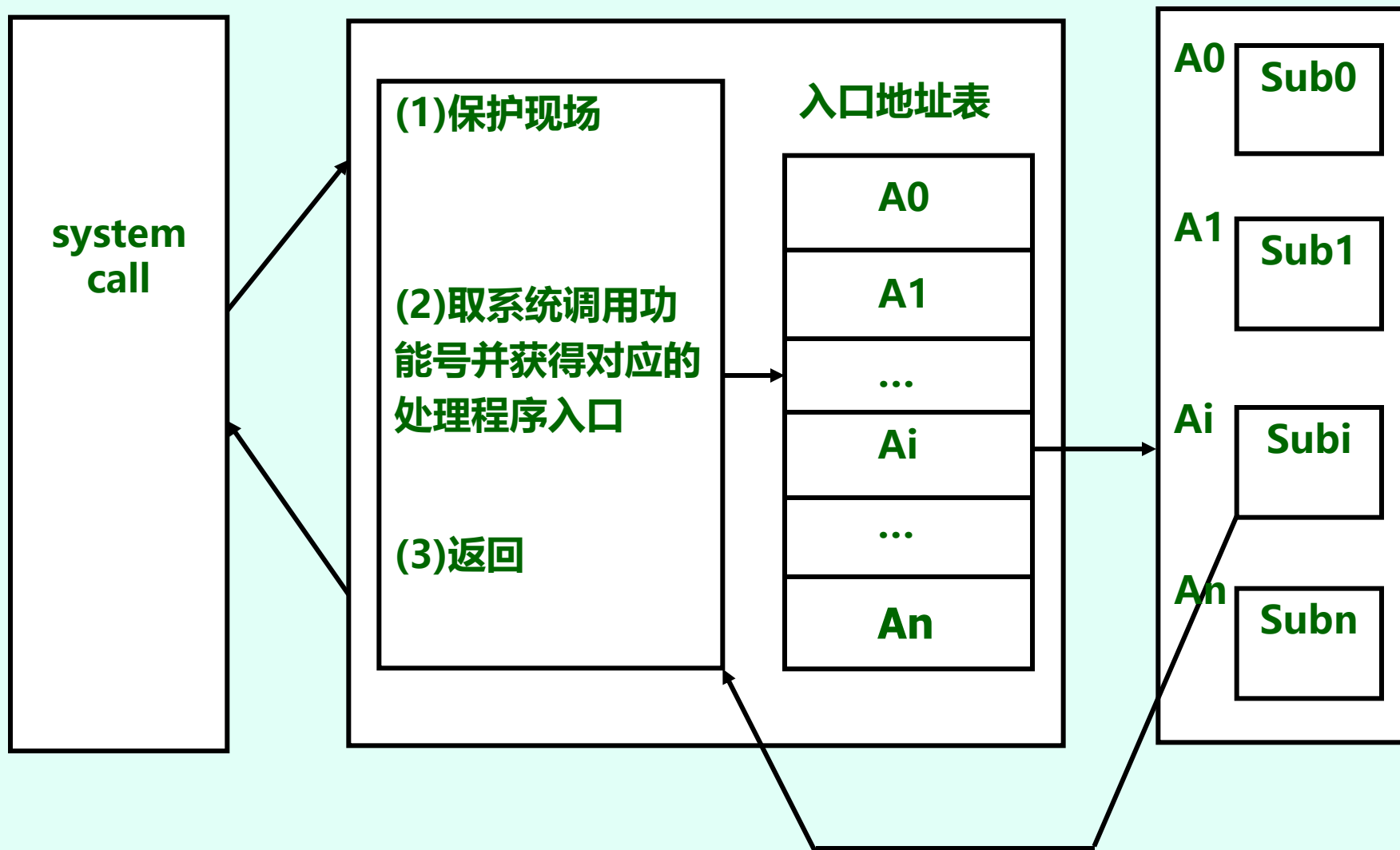
## (3) 系统调用的处理过程

- ✓ 每个系统调用对应一个事先给定的功能号，如0、1、2、3等
- ✓ 系统为实现系统调用功能的子程序（过程）设置入口地址表
- ✓ 每个入口地址与相应的陷入处理程序对应
- ✓ 进入系统调用处理前，保护处理机现场
- ✓ 在系统调用返回后，要恢复处理机现场
- ✓ 在系统调用处理结束后，用户程序可以利用系统调用的返回结果继续执行

## 用户程序

## 陷入处理机构

## 系统子程序



系统调用的处理过程



# 系统调用

---

怎样实现用户程序和系统程序间的参数传递？

**常用的3种实现方法：**

①由陷入指令自带参数

陷入指令的长度是有限的，且还要携带系统调用功能号，只能自带有限的参数

②通过有关通用寄存器来传递参数

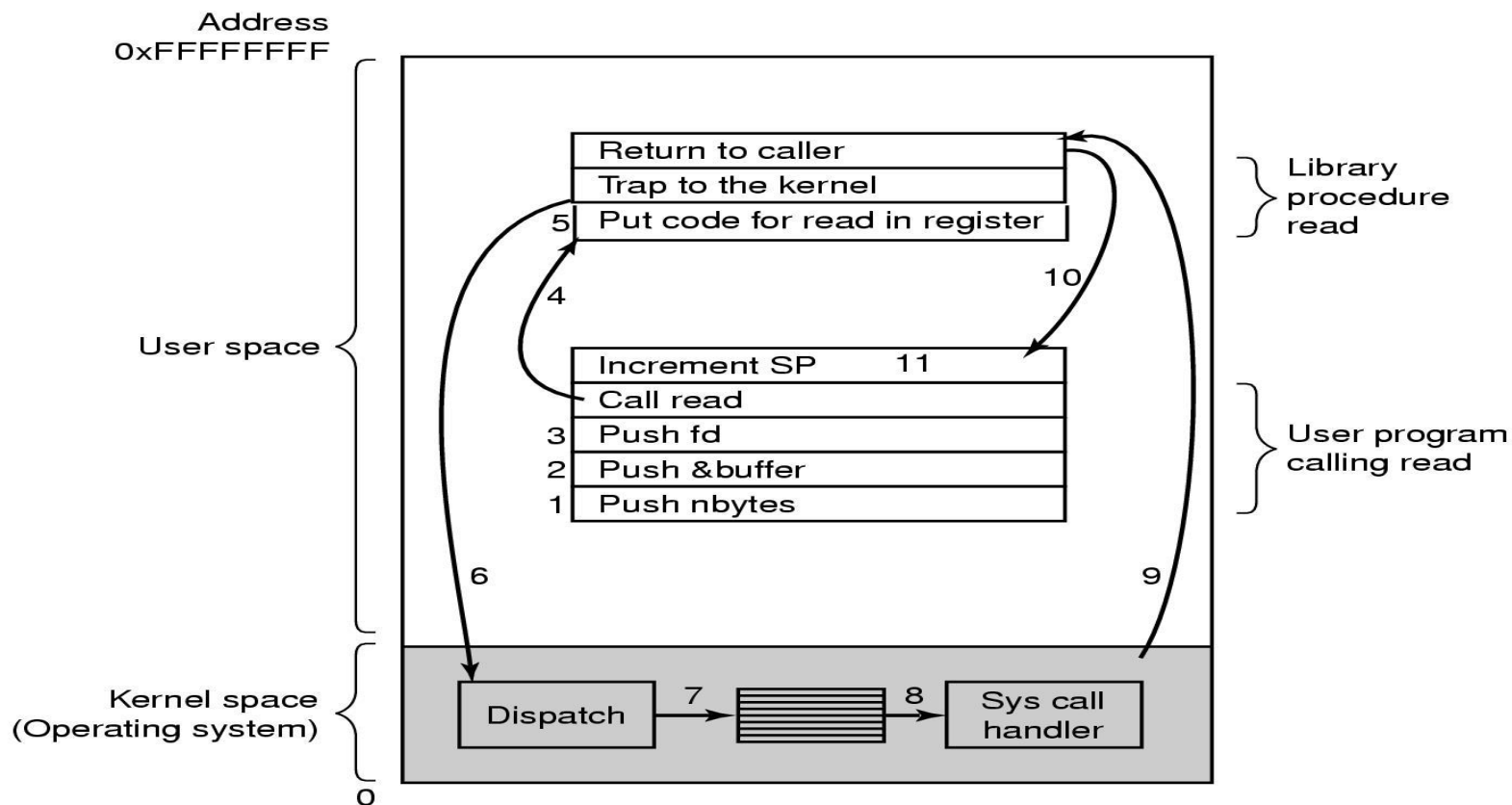
这些寄存器应是系统程序 and 用户程序都能访问的，

由于寄存器长度和个数有限，无法传递较多的数据

③大多在内存中开辟专用的区域（如堆栈区）来传递参数

# 系统调用

read (fd, buffer, nbytes)





# 系统调用

---

## (4) 系统调用的分类

系统调用分为两大类：系统自身所需要的；作为服务提供给用户的

- ✓ 进程管理系统调用
- ✓ 内存管理系统调用
- ✓ 文件和目录管理系统调用
- ✓ I/O设备管理系统调用



# 系统调用

---

## (5) 系统调用与一般过程调用的比较

### 相同点:

- ✓ 改变指令执行流程
- ✓ 是为实现某个特定功能而设计的，可多次调用
- ✓ 执行完成后返回





# 系统调用与一般过程调用的比较

---

## 系统调用与一般过程调用的不同：

### ① 运行在不同的系统状态

一般的用户过程运行在用户态

系统调用运行在核心态

### ② 返回问题

在抢占式调度的系统中，系统调用在返回时，OS将对所有就绪进程进行优先级分析。如果调用进程仍有最高优先级，则返回到调用进程执行，否则，引起重新调度，让优先级最高的进程执行。此时，系统把调用进程放入就绪队列。



# 系统调用与一般过程调用的不同

---

## ③ 嵌套或递归调用

对于系统调用，一般不允许在同一个进程中发生嵌套或递归（不同进程可以重入同一个系统调用）

## ④ 进入和返回方式不同

利用int或trap指令进行系统调用；

利用call指令进入普通的过程调用；

一般过程的返回用ret指令

系统调用的返回用iret指令

**核心态与用户态的转换由系统在int指令与iret指令内部自动完成，**

没有用一条单独的专门指令，好处在于：有效地防止在核心态下执行用户程序



# 系统调用与一般过程调用的不同

---

- CALL指令的内部实现过程
  - 返回地址压栈（即该CALL指令下一条指令的地址）
  - 将该CALL指令中所含的地址（即被调用代码所在地址）送入PC
- RET指令的内部实现过程
  - 从栈顶弹出返回地址送入程序计数器PC
- INT指令和IRET指令的执行过程中
  - 要处理程序状态字PSW
  - INT指令中要保存用户程序的老PSW
  - IRET指令中要在返回用户程序前恢复用户程序的老PSW



# 5.1 Linux 简介

---

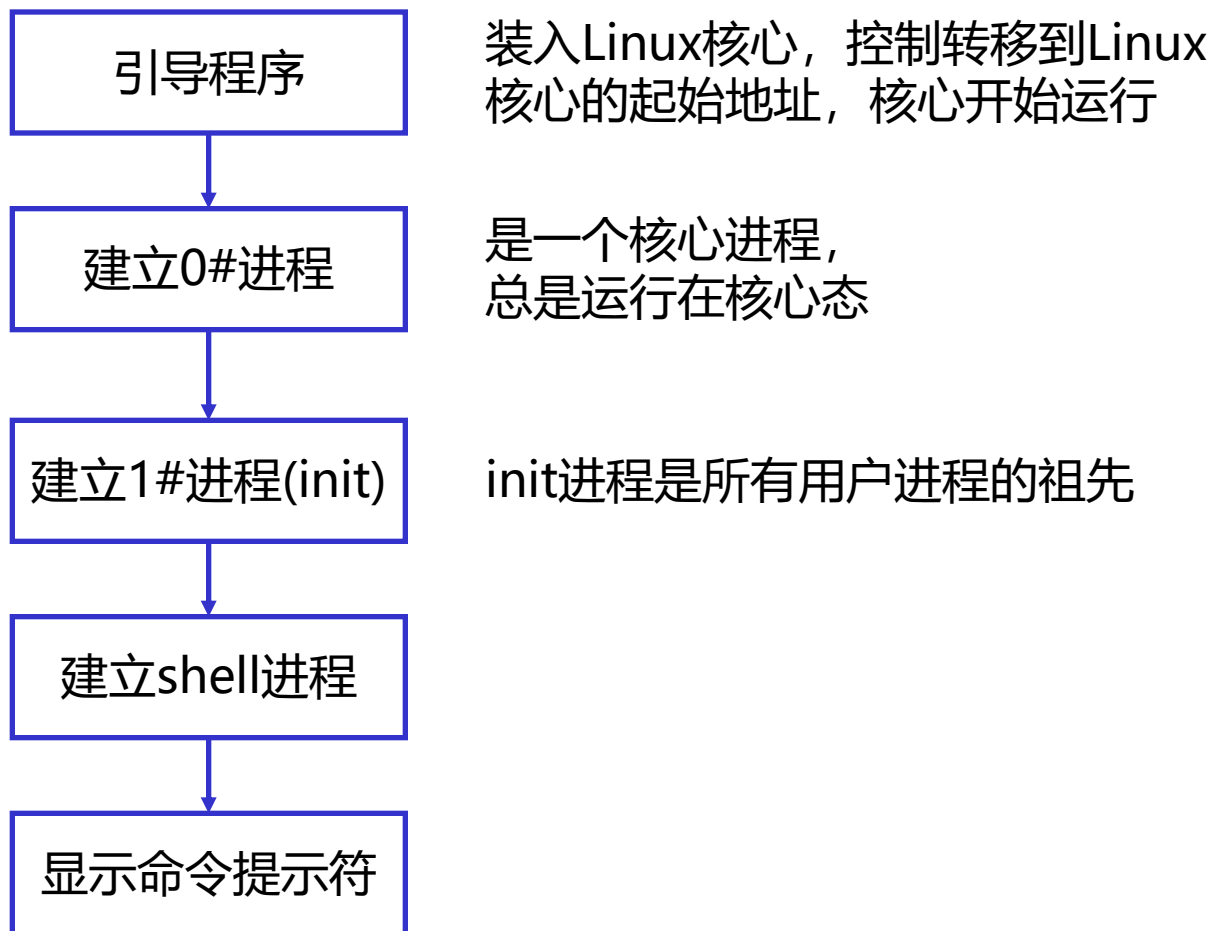
## 二、Shell

Linux的命令解释器称为Shell

Shell常常被普通用户看作是“Linux”，而实际上它与操作系统本身毫不相干，可以很容易换掉。

# Shell

## 1. Shell进程





# Shell

---

## 2. Shell解释命令的方式

当用户输入一行命令后，shell抽出第1个词，作为要运行程序的名字，将后面的参数以字符串的形式传递给被调用程序，同时建立一个执行该命令的用户进程。

shell等待直到用户进程执行结束后，shell恢复运行。

## 3. Shell脚本

用**shell语言**（看似C语言，具有if、for、while等结构），写一个shell脚本文件，可执行完成复杂任务。

一个简单的shell脚本由普通的shell命令组成，就像在shell命令行中交互地输入这些命令一样。用户可以在shell脚本中定义自己的命令。脚本中可以包括注释行。



## 5.2 Linux 进程管理

---

### 一、进程的结构task\_struct

每个进程对应1个task\_struct结构，称为**进程描述符**相当于PCB。

主要包含下列信息：

- ✓ 进程标识符pid
- ✓ 进程状态
- ✓ 调度信息：优先数，消耗的CPU时间
- ✓ 信号及其处理方式
- ✓ 文件描述符表：记录该进程打开的所有文件
- ✓ 进程在内存的位置（页表指针）



## 5.2 Linux 进程管理

---

### 二、进程的状态及其转换

5种状态：

(1) TASK\_RUNNING：运行状态

进程正在运行或就绪。表示进程具备运行的资格，正在运行或等待被调度执行。进程控制块中有一个run\_list成员，所有处于TASK\_RUNNING状态的进程都通过该成员链在一起，称之为可运行队列。

(2) TASK\_INTERRUPTIBLE：可中断睡眠（阻塞）

可被软中断信号唤醒

(3) TASK\_UNINTERRUPTIBLE：不可中断睡眠（阻塞）

不可以被软中断信号唤醒

以上两种状态均表示进程处于阻塞状态





# Linux进程状态

---

## (4) TASK\_STOPPED: 暂停

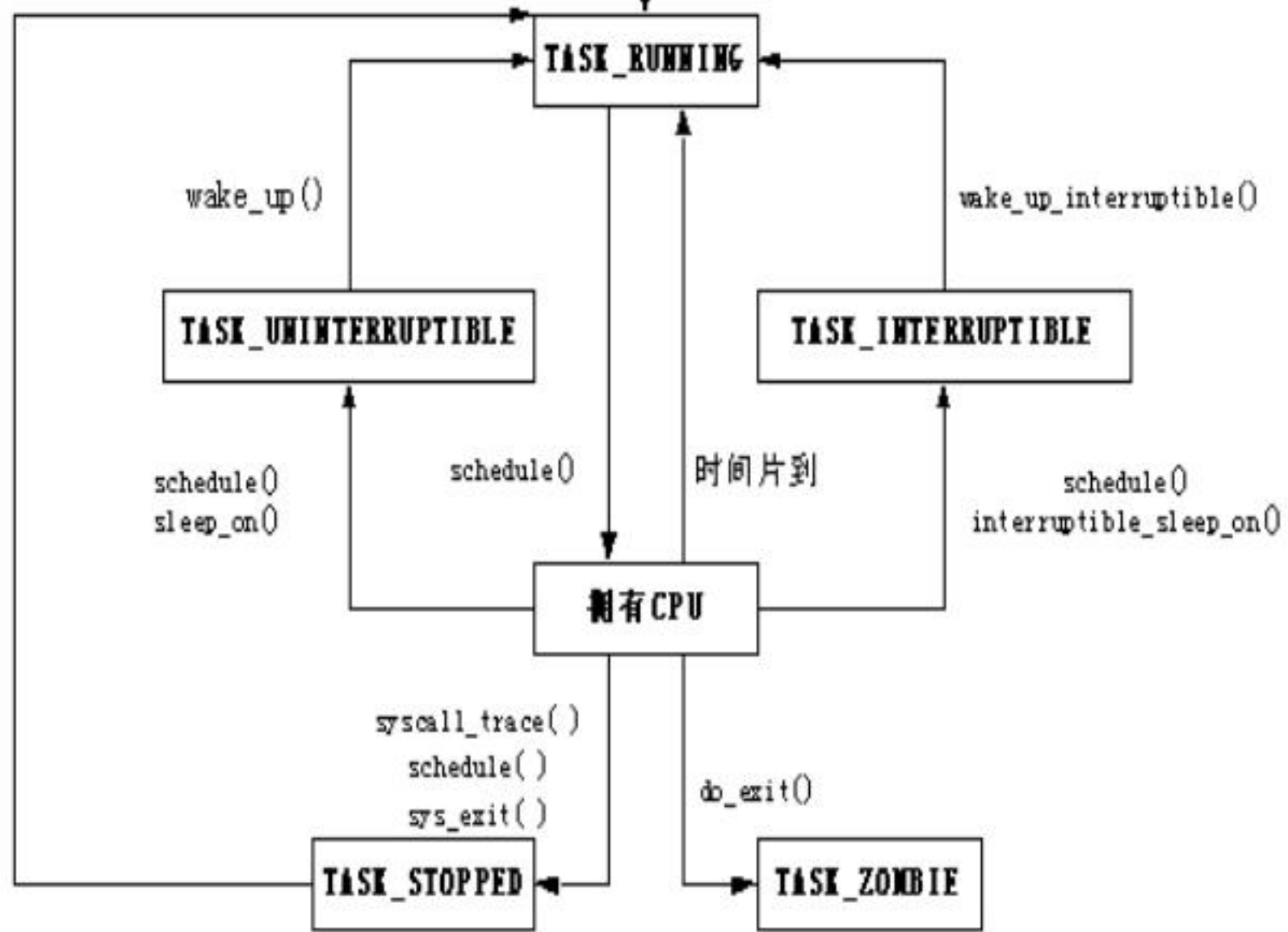
进程处于暂停状态，主要用于调试目的。如正在运行的进程收到信号SIGSTOP、SIGSTP、SIGTTIN、SIGTTOU后进入这个状态。

## (5) TASK\_ZOMBIE: 僵死

表示进程已经结束运行并释放了大部分占用的资源，但task\_struct结构还未被释放。

收到SIG\_KILL或  
SIG\_CONT后, 执行  
wake\_up()

do\_fork()





## 5.2 Linux进程管理

---

### 三、进程控制

#### 1. 进程创建fork

```
pid = fork()
```

创建子进程。子进程是父进程的一个逻辑副本。

**父进程返回子进程id，子进程返回0。**



# fork

---

algorithm fork()

in: 无

out: 父进程返回子进程的pid, 子进程返回0

```
{  
    为新进程分配一个task_struct结构和唯一的pid;  
    复制父进程task_struct的内容到子进程中, 并重新设置与父进程不同的数据成员;  
    父进程地址空间的逻辑副本复制到子进程;  
    if (执行的是父进程) {  
        将子进程的状态置为TASK_RUNNING;  
        return 子进程的pid;  
    }  
    else { //正在执行的子进程, 被调度后才会执行  
        return 0;  
    }  
}
```



# fork

---

说明:

- (1) 进程的pid是一个整数，在0到最大值之间。
- (2) 复制逻辑副本的意思是：Linux使用写时复制（copy on write）机制，子进程暂时共享父进程的有关数据结构，复制延迟到子进程作修改时才进行。



# 进 程 控 制

---

## 2. 执行一个文件的调用exec

由指定的可执行文件覆盖进程的地址空间。

具有多种形式：execve, execlp, execvp等。



# 进程控制

---

`execve(char *filename, char *argv[], char *envp[])`

执行文件filename, 覆盖进程的地址空间

C语言主函数的形式:

`main(int argc, char **argv, char **envp)`

argc: 参数个数, 包括程序名

argv: 或写作char \*argv[]

argv[0]~argv[argc-1]指向每个参数串

envp: 环境变量串, 以空串结束



# 进程控制

---

**【例】 一个简化的shell。**

```
while (1) {  
    type_prompt(); //在屏幕上显示提示符  
    read_command(command, params); //读输入的命令  
    pid = fork();  
    if (pid < 0) {  
        printf( "Unable to fork.\n" ); //出错  
        continue;  
    }  
    if (pid != 0) { //子Shell创建完毕  
        wait(0); //父Shell代码，等待子Shell结束  
    }  
    else {  
        execve(command, params, 0); //子Shell代码  
    }  
}
```





## 5.2 Linux 进 程 管 理

---

### 四、进程调度

#### 1. 调度原理

3种调度策略：

① 动态优先级：普通进程

② 先来先服务：实时进程

③ 时间片轮转：实时进程

✓ 进程可以设置调度策略（保存在进程描述符中）

✓ 优先数越大，优先级越低

✓ 类似于多级反馈队列



# 进程调度

---

## 2. 调度的时机

(1) 进程主动放弃CPU —— 直接调用`schedule()`

✓ 隐式主动放弃CPU

如执行系统调用`read`阻塞而转入`schedule()`

✓ 显式主动放弃CPU

如执行系统调用`sched_yield()`、`nanosleep()`等

(2) 进程从核心态返回到用户态（从中断处理、系统调用、异常处理）之前，核心检查当前进程的调度标志`need_resched`，若为1，则调用`schedule()`。



# 进程调度

---

## 3. 调度标志`need_resched`置1的时机

- ✓ 当前进程的时间片用完，时钟中断处理程序设置该进程的`need_resched`标志。
- ✓ 刚唤醒的进程优先级高于当前进程，将当前进程的`need_resched`标志置1。



## 5.2 Linux 进程管理

---

### 五、进程同步与通信

#### 1. 信号 (signal)

信号是软件层次上对中断的一种模拟，又称为软中断。

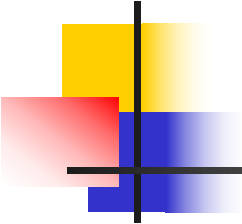
表示进程之间传送预先约定的信息类型，用于通知某进程发生了一个事件。

例如：

2 SIGINT 中断（用户键入ctrl-C）

9 SIGKILL 杀死（终止）进程

11 SIGSEGV 越界访问内存



# 信号

---

## (1) 信号的发送

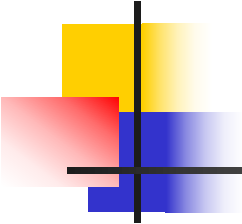
`kill(int pid, int sig)`

pid: 发给哪个进程

= 0: 信号给同组所有进程

> 0: 信号发给pid所指的进程

sig: 发送的信号



# 信号

---

## (2) 信号处理方式的预置

`signal(sig, func);`

sig: 信号

func: 信号处理函数的指针

0: SIG\_DFL, 收到信号后终止

1: SIG\_IGN, 忽略

其他: 处理函数的指针



# 信号处理方式预置例

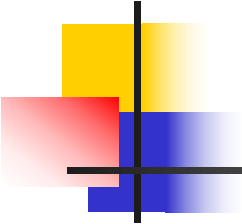
```
void SetupSignalHandlers(void)
{
    signal( SIGHUP, SIG_IGN );          /* 1 */
    signal( SIGINT, SIG_IGN );          /* 2 */

    /* . . . . . */

    signal( SIGSEGV, UnexpectedSignal ); /* 11 */
    signal( SIGSYS, UnexpectedSignal );  /* 12 */
    signal( SIGPIPE, SIG_IGN );          /* 13 broken pipe */
    signal( SIGALRM, SIG_IGN );          /* 14 */
    signal( SIGTERM, Terminate);         /* 15 */
}

void UnexpectedSignal(int sig)
{
    ErrorMessage("Unexpected signal %d caught. Process aborted.", sig);
    ClearUp();
    abort();
}

void Terminate( int sig )
{
    ClearUp();
    exit(0);
}
```



# 信号

---

## (3) 信号和中断的区别

- ✓ 中断有优先级；信号没有优先级（所有信号是平等的）
- ✓ 信号处理程序在用户态运行；而中断处理程序运行在核心态
- ✓ 中断响应是及时的；信号响应通常有较大的时间延迟，必须等到接收进程执行时才生效





# 进程同步与通信

---

## 2. 消息队列 (message queue)

每个进程可以有多个消息队列

(1) 每个消息队列都有1个key(整数), 由应用程序指定

(2) 在进程内, 创建/打开的消息队列都有1个消息队列描述符Qid, 其作用与文件描述符类似

(3) 使用方法

`Qid = msgget(key, msgflag);` //创建/打开消息队列

`msgsnd(Qid, buf, nsize, ...);` //发送消息

`msgrcv(Qid, buf, nsize, ...);` //接收消息

`msgctl(Qid, cmd, buf);` //读取消息队列的状态, 或删除消息队列



# 进程同步与通信

---

## 3. 共享内存 (shared memory)

与消息队列类似，以key唯一标识。

```
shmid = shmget(key, size, shmflag); //创建/打开共享内存
```

```
char *shmaddr = shmat(shmid, ...); //将共享内存附接到进程的虚地址空间
```

```
shmdt(shmaddr); //把共享内存与进程断开
```

```
shmctl(shmid, cmd, buf); //读取共享内存的状态，或删除共享内存
```



# 共享内存例

```
/*
** Attach System BB
*/
if ((SystemBBid = shmget(SystemBBKey, sizeof( BBOARD ), 0600 ))<0)
{
    ErrorMsg("%s:Cannot open BB: shmget():%m", ProgName);
    return -1;
}

if ((pBBoard = (BBOARD *)shmat( SystemBBid, (char *)0, 0))== (BBOARD *)0)
{
    ErrorMsg("%s:Cannot attach BB: shmat():%m",ProgName);
    return -1;
}

pServerDesc = &(pBBoard->bb_servertab[ServerIndex]);
```



# 进程同步与通信

---

## 4. 信号量 (semaphore)

以key唯一标识。

```
semid = semget(key, nsems, semflag); //创建/打开信号量集
```

nsem: 信号量个数

```
semop(semid, sops, nsops); //信号量的P、V操作
```

sops: 指向信号量集操作结构的数组指针, 操作可以是:

正数: 相当于V操作

负数: 相当于P操作

```
semctl(semid, ...); //可用来设置信号量的初始值
```



# 进程同步与通信

---

## 5. 管道 (pipe)

连接一个写进程和读进程的pipe文件

- ✓ 无名管道

- 一个临时文件，用于进程及其子孙进程之间的通信

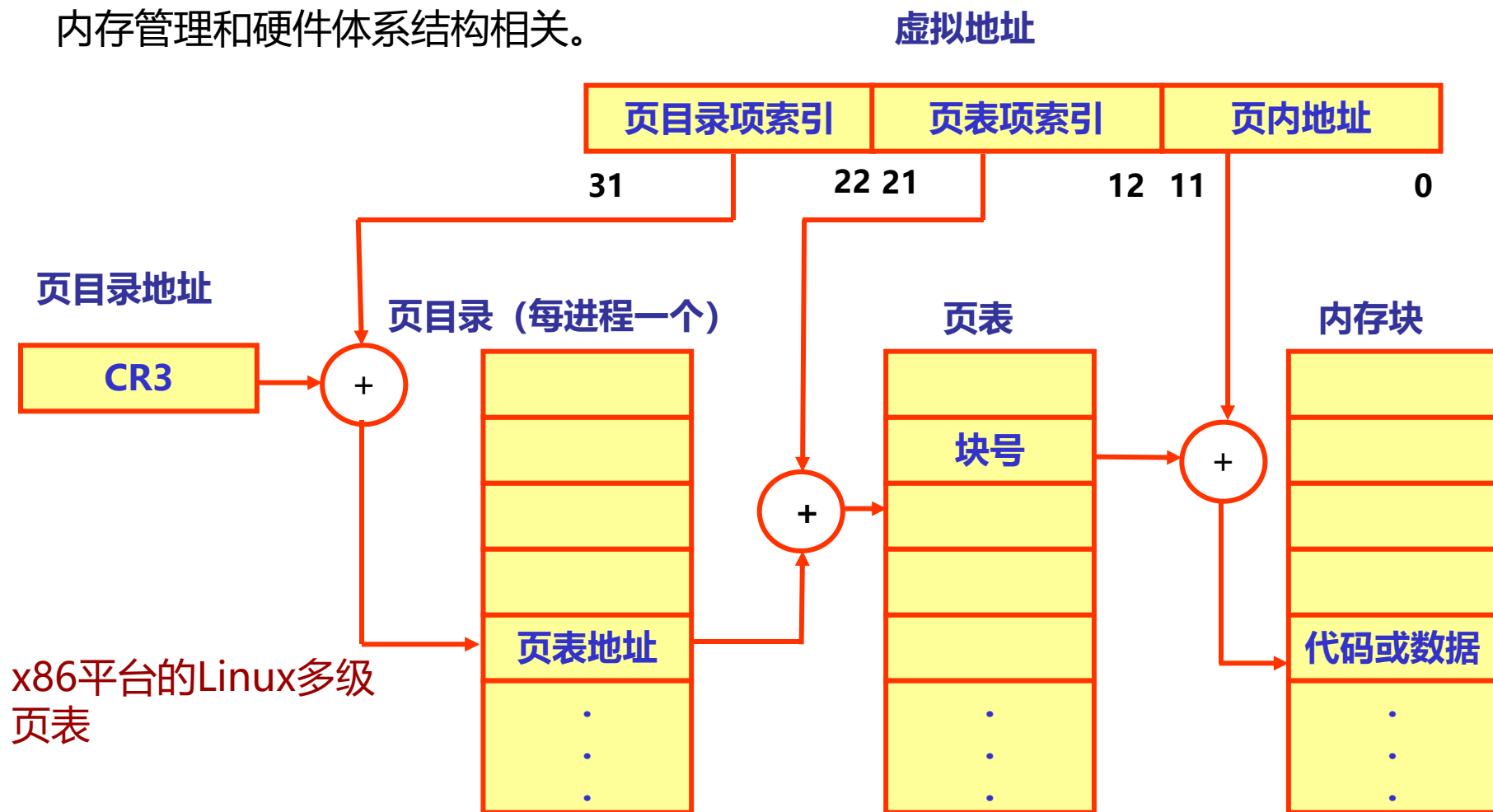
- ✓ 有名管道

- 具有路径名的文件，可用于任何进程之间的通信

## 5.3 Linux 内存管理

### 一、Linux的多级页表

内存管理和硬件体系结构相关。





# Linux的多级页表

---

## 页表项的主要内容:

- ✓ 有效位: 该页是否在内存中
- ✓ 访问位: 该页最近是否被访问过
- ✓ 修改位: 该页是否被修改过
- ✓ 读写位: 只读/可写



## 5.3 Linux 内存管理

---

### 二、Linux的请求调页技术

Linux采用请求调页策略进行内存管理。

当页表有效位 = 0时，产生缺页异常，进行缺页处理。

#### (1) 交换缓冲区

缺页若在内存的交换缓冲区中，则不必读交换区。





# Linux的请求调页技术

---

## (2) 页的换出

### 页的换出时机:

- ① 分配内存时发现空闲内存低于设定的极限值
- ② 使用核心线程kswapd**周期性地换出内存页**

Linux将未使用的物理块作为系统的缓冲区和块设备的缓冲区。如果内存中无足够的物理块存放调入页，则首先减少各种缓冲区的大小来满足进程的需要。如果仍然不够，则淘汰相应的物理块。

Linux的页置换算法是近似的LRU算法，是NRU算法。



## 5.4 Windows 2000/xp的进程与内存管理

---

Windows 2000/2003/xp/...:

用C/C++和少量汇编语言实现

基于客户/服务器模型、对象模型、对称多处理模型

网络功能是OS的一部分

可移植、运行于不同CPU和硬件平台OS



## 5.4 Windows 2000/xp的进程与内存管理

---

### 一、进程与线程

进程是作为对象实现的。

进程的组成部分：

- ✓ 唯一的Id
- ✓ 一个可执行程序（代码、数据）
- ✓ 私有的虚拟地址空间
- ✓ 拥有的系统资源
- ✓ 至少1个执行线程

资源分配的对象是进程



# Windows 2000/xp的进程与线程

---

多个线程共享其进程的虚拟地址空间

线程的组成部分：

- ✓ 唯一的Id
- ✓ 描述处理机状态的一组寄存器值
- ✓ 用户栈和核心栈

调度的基本单位是线程



# Windows 2000/xp的进程与线程

---

## 二、与进程和线程有关的API

Win32子系统的进程（线程）控制系统调用主要有：

CreateProcess()

ExitProcess()

TerminateProcess()

CreateThread()

ExitThread()

SuspendThread()

ResumeThread()

等等。



# 与进程和线程有关的API

---

CreateProcess(): 用于创建新进程及其主线程，以执行指定的程序。

ExitProcess()或TerminateProcess(): 进程包含的线程全部终止。

ExitProcess()终止一个进程和它的所有线程；它的终止操作是完整的，包括关闭所有对象句柄、它的所有线程等。

TerminateProcess()终止指定的进程和它的所有线程；它的终止操作是不完整的（如：不向相关DLL通报关闭情况），通常只用于异常情况下对进程的终止。



# 与进程和线程有关的API

---

CreateThread()函数在调用进程的地址空间上创建一个线程，以执行指定的函数；返回值为所创建线程的句柄。

ExitThread()函数用于结束本线程。

SuspendThread()函数用于挂起指定的线程。

ResumeThread()函数递减指定线程的挂起计数，挂起计数为0时，线程恢复执行。



## 5.4 Windows 2000/xp的进程与内存管理

---

### 三、线程调度

#### 1. 线程的7种状态

- (1) 就绪 (Ready) : 具备执行条件, 等待被挑选进入 “备用” 状态
- (2) 备用 (Standby) : 准备在特定处理机上执行。每个处理机上只能有一个线程处于备用状态
- (3) 运行 (Running) : 完成描述符表切换, 线程进入运行状态, 直到内核抢占、时间片用完、线程终止或进入等待状态
- (4) 等待 (Waiting) : 线程等待对象句柄, 以同步它的执行。等待结束时, 根据优先级进入运行或就绪状态。





# 线程状态

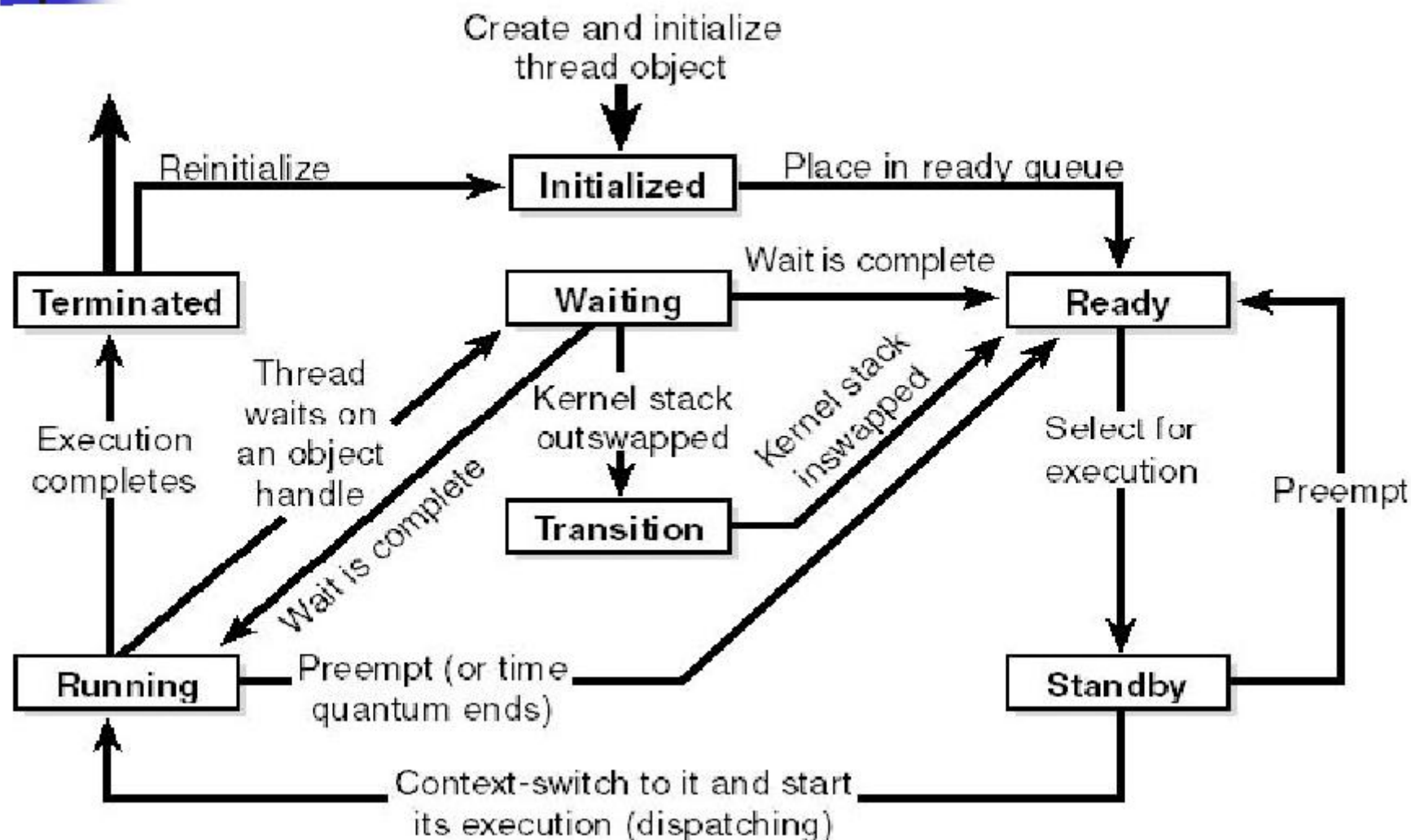
---

(5) 转换 (Transition) : 与就绪状态类似, 但线程的内核栈处于外存。当线程等待的事件出现而它的内核栈处于外存时, 进入转换状态。当其内核栈调入内存, 线程进入就绪状态

(6) 终止 (Terminated) : 线程执行完就进入终止状态。如果执行体有一指向线程对象的指针, 可将线程对象重新初始化, 并再次使用

(7) 初始化 (Initialized) : 线程正在创建过程中

# 线程状态





# 线程调度

## 2. 调度算法

调度单位是线程而不是进程，采用抢占式动态优先级多级队列，依据优先级和分配时间配额来调度。

### (1) 基本思想

- ✓ 每个优先级的就绪线程排成一个先进先出队列
- ✓ 线程的基本优先级与其进程的基本优先级有关
- ✓ 当一个线程变成就绪时，它可能立即运行或排到相应优先级队列的尾部
- ✓ 调度程序总是挑选优先级最高的就绪线程运行
- ✓ 在同一优先级的各线程按时间片轮转算法进行调度
- ✓ 线程时间配额用完而被抢占，优先级降低一级，进入下一级队列，直到其基本优先级
- ✓ 在多处理机系统中多个线程可并行运行



# 线程调度

---

## (2) 线程时间配额

- ✓ 时间配额是一个线程从进入运行状态到Windows 检查是否有其他优先级相同的线程需要开始运行之间的时间总和
- ✓ 一个线程用完了自己的时间配额时，如果没有其它相同优先级线程，Windows将重新给该线程分配一个新的时间配额，并继续运行
- ✓ 时间配额不是一个时间长度值，而是一个称为配额单位(quantum unit)的整数



# 线程调度

---

## (3) 线程优先级

Windows 2000/XP内部使用32个线程优先级，范围从0到31，它们被分成以下三类：

- ✓ 16个实时线程优先级（16 - 31）

Windows 并不是通常意义上的实时，并不提供实时操作系统服务

- ✓ 15个可变线程优先级（1 - 15）

- ✓ 1个系统线程优先级（0），仅用于对系统中空闲物理块进行清零的零页线程



# 线程调度

---

## (4) 线程优先级提升

- ✓ 在下列5种情况下，Windows 会提升线程的当前优先级：
  - I/O操作完成
  - 信号量或事件等待结束
  - 前台进程中的线程完成一个等待操作
  - 由于窗口活动而唤醒图形用户接口线程
  - 线程处于就绪状态超过一定时间，但没能进入运行状态(处理机饥饿)
- ✓ 线程优先级提升的目的是改进系统吞吐量、响应时间等整体特征，解决线程调度策略中潜在的不公正性。但它也不是完美的，它并不会使所有应用都受益
- ✓ Windows 永远不会提升实时优先级范围内(16至31)的线程优先级



# 线程优先级提升

---

## 1) I/O操作完成后的线程优先级提升

- ✓ 在完成I/O操作后，Windows 将临时提升等待该操作线程的优先级，以保证等待I/O操作的线程能有更多的机会立即开始处理得到的结果
- ✓ 为了避免I/O操作导致对某些线程的不公平，在I/O操作完成后唤醒等待线程时将把该线程的时间配额减1
- ✓ 线程优先级的实际提升值是由设备驱动程序决定的
- ✓ 设备驱动程序在完成I/O请求时通过内核函数IoCompleteRequest来指定优先级提升的幅度
- ✓ 线程优先级的提升幅度与I/O请求的响应时间要求是一致的，响应时间要求越高，优先级提升幅度越大



# 线程优先级提升

---

## 2) 等待事件和信号量后的线程优先级提升

- ✓ 当一个等待事件对象或信号量对象的线程完成等待后，将提升一个优先级
- ✓ 阻塞于事件或信号量的线程得到的处理机时间比处理机繁忙型线程要少，这种提升可减少这种不平衡带来的影响





# 线程优先级提升

---

## 3) 前台线程在等待结束后的优先级提升

- ✓ 对于前台线程，一个内核对象上的等待操作完成时，内核函数 KiUnwaitThread 会提升线程的当前优先级 (不是线程的基本优先级)
- ✓ 在前台应用完成它的等待操作时小幅提升其优先级，以使它更有可能马上进入运行状态，有效改善前台应用的响应时间特性



# 线程优先级提升

---

## 4) 图形用户接口线程被唤醒后的优先级提升

- ✓ 拥有窗口的线程在被窗口活动唤醒(如收到窗口消息)时将得到一个幅度为2的额外优先级提升
- ✓ 这种优先级提升的目的是为了改善交互应用的响应时间



# 线程优先级提升

---

## 5) 对处理机饥饿线程的优先级提升

- ✓ 系统线程“平衡集管理器(balance set manager)”会每秒钟检查一次就绪队列，看是否存在一直在就绪队列中排队超过300个时钟中断间隔的线程
- ✓ 如果找到这样的线程，平衡集管理器将把该线程的优先级提升到15，并分配给它一个长度为正常值两倍的时间配额
- ✓ 当被提升线程用完它的时间配额后，该线程的优先级立即衰减到它原来的基本优先级



# 线程调度

---

## (5) 空闲线程

- ✓ 如果在一个处理机上没有可运行的线程，Windows会调度相应处理机对应的空闲线程
- ✓ 由于在多处理机系统中可能存在多个处理机同时运行空闲线程，所以系统中的每个处理机都有一个对应的空闲线程
- ✓ Windows给空闲线程指定的线程优先级为0，该空闲线程只有在没有其他线程运行时才运行

空闲线程的功能就是在一个循环中检测是否有要做的工作，例如处理所有待处理的中断请求，检查是否有就绪线程可进入运行状态。如果有，调度相应线程进入运行状态，调用硬件抽象层的处理机空闲例程，执行相应的电源管理功能等



## 5.4 Windows 2000/xp的进程与内存管理

---

### 四、线程间同步与通信

#### 1. 内核的同步与互斥机制

(1) 提高临界区代码执行的中断优先级，暂时屏蔽那些有可能也要使用该临界资源的中断。

只能用于单处理机。因为一个处理机上提高中断优先级不能阻止其他处理机上的中断。

(2) 转锁 (spin-lock) 机制

用TestAndSet指令实现，可用于多处理机。



# 线程的同步与通信

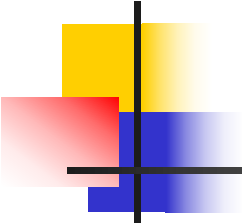
---

## 2. 线程间同步

### (1) 同步对象

Windows提供了以下几种同步对象的API，用于进程和线程同步：

- ✓ 事件 (Event)
- ✓ 互斥量 (Mutex)
- ✓ 信号量 (Semaphore)
- ✓ 临界区 (Critical Section)



# 同步对象

---

## 同步对象的2种状态：

- ✓ 可用，或称有信号（signaled）：允许等待函数返回
- ✓ 不可用，或称无信号（nonsignaled）：阻止等待函数返回

## 这些同步对象有2种使用方式：

- ✓ 无名对象：只能用于同一进程中的线程同步
- ✓ 有名对象：可以用于不同进程中的线程同步。临界区除外。

为同步对象指定一个名称（字符串），不同进程用同样名称打开或创建对象，从而获得该对象在本进程的句柄。



# 同步对象

---

## 1) 事件

相当于“触发器”，用于通知一个或多个等待它的线程“事件已发生”。

相关API有CreateEvent、OpenEvent、SetEvent、ResetEvent。

- ✓ CreateEvent: 创建一个事件对象，返回对象句柄。
- ✓ OpenEvent: 返回一个已存在的事件对象的句柄，用于后续访问。
- ✓ SetEvent: 设置指定事件对象为可用状态
- ✓ ResetEvent: 设置指定事件对象为不可用状态





# 同步对象

---

## 2) 互斥量 (Mutex)

就是互斥信号量，某一时刻只能被一个线程所拥有。

相关API有CreateMutex、OpenMutex、ReleaseMutex。

- ✓ CreateMutex：创建一个互斥量对象，返回对象句柄。
- ✓ OpenMutex：返回一个已存在的互斥量对象的句柄，用于后续访问。
- ✓ ReleaseMutex：释放对互斥量对象的占用，使之成为可用。



# 同步对象

---

## 3) 信号量 (Semaphore)

取值为0到指定最大值之间，可用来限制并发访问共享资源的线程个数。

相关API有CreateSemaphore、OpenSemaphore、ReleaseSemaphore等。

- ✓ CreateSemaphore: 创建一个信号量对象，在参数中指定最大值和初始值，返回对象句柄。
- ✓ OpenSemaphore: 返回一个已存在的信号量对象的句柄，用于后续访问。
- ✓ ReleaseSemaphore: 释放信号量对象，使之成为可用。



# 同步对象

---

## 4) 临界区 (Critical Section)

同时只能由一个线程所拥有，但只能用于同一个进程中的线程互斥。

相关API有InitializeCriticalSection、EnterCriticalSection、LeaveCriticalSection、DeleteCriticalSection等。

- ✓ InitializeCriticalSection：对临界区对象进行初始化。
- ✓ EnterCriticalSection：等待拥有临界区对象，得到使用权后返回。
- ✓ LeaveCriticalSection：释放临界区对象的使用权。
- ✓ DeleteCriticalSection：释放与临界区对象相关的所有资源。



# 线程间同步

---

## (2) 等待函数

对于同步对象，Win32 API提供了**2个等待函数**：

WaitForSingleObject与WaitForMultipleObjects。

等待函数执行时会阻塞，直到超时或者等待条件满足。

WaitForSingleObject：等待指定对象为可用状态

WaitForMultipleObjects：等待多个对象为可用状态



# 线程间同步

---

## (3) 使用Win32 API实现线程同步的一般方法

以信号量为例，其他同步对象类似。

① 给信号量对象约定一个名字；

② 一个线程以指定的名字为参数，调用CreateSemaphore()创建一个信号量，返回句柄；

任一线程访问共享资源前，以指定的名字调用OpenSemaphore()打开该信号量，返回句柄；

③ 以句柄为参数调用等待函数（P操作）；

④ 访问完后，调用ReleaseSemaphore释放该信号量（V操作）。



# 线程的同步与通信

---

## 3. 线程间通信 —— 共享存储区 (shared memory)

- ✓ 共享存储区可用于进程间的大数据量通信
- ✓ 进行通信的各进程可以任意读写共享存储区，也可在共享存储区上使用任意数据结构
- ✓ 在使用共享存储区时，需要进程互斥和同步机制的辅助来确保数据一致性
- ✓ Windows 2000/XP采用文件映射(file mapping)机制来实现共享存储区，用户进程可以将整个文件映射为进程虚拟地址空间的一部分来加以访问



# 共享存储区

---

## Windows 的文件映射（内存映象文件）：

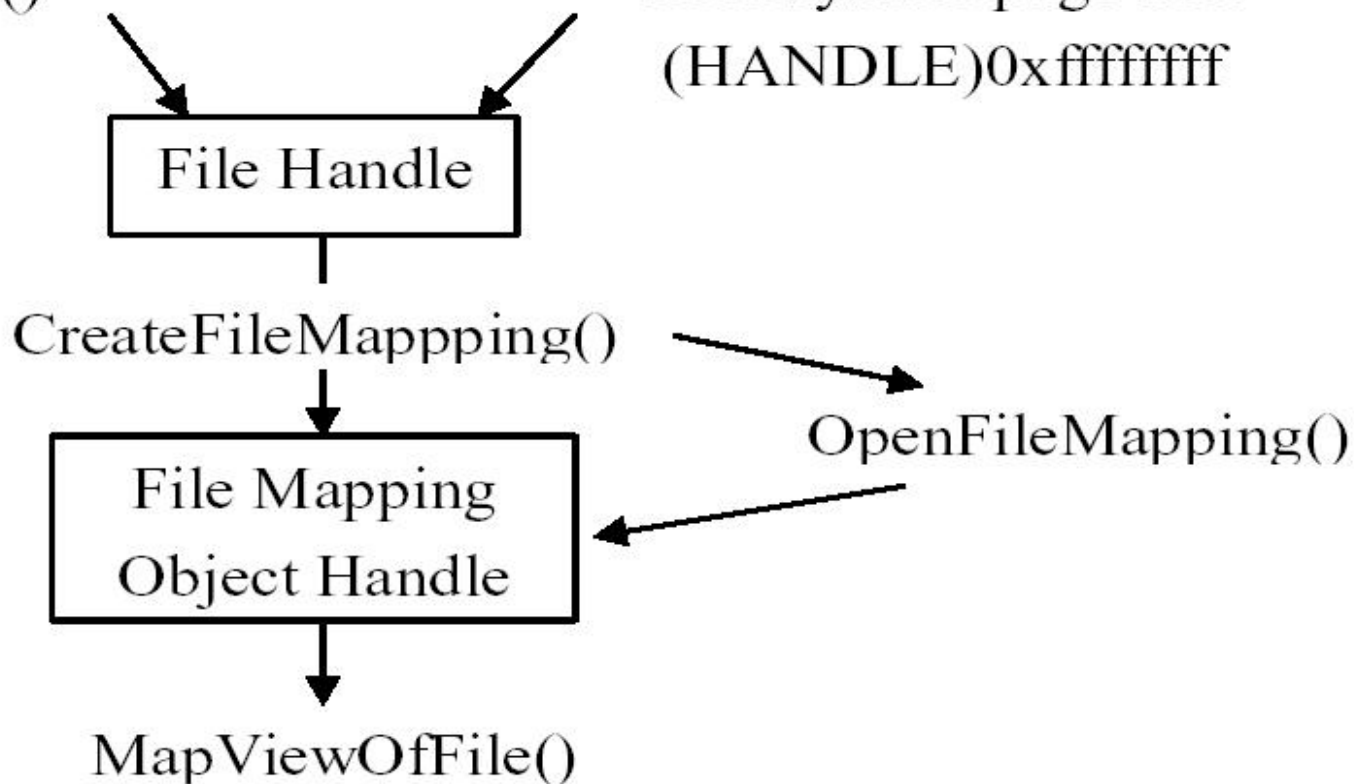
- ✓ CreateFileMapping为指定文件创建一个文件映射对象，返回对象句柄
- ✓ OpenFileMapping打开一个命名的文件映射对象，返回对象句柄
- ✓ MapViewOfFile把文件视图（文件的一部分或全部）映射到本进程的地址空间，返回映射地址空间的首地址。当完成文件到进程地址空间的映射后，就可利用首地址进行读写
- ✓ FlushViewOfFile可把映射地址空间的内容写到物理文件中
- ✓ UnmapViewOfFile拆除文件映射与本进程地址空间的映射关系
- ✓ CloseHandle可关闭文件映射对象

# 共享存储区

基于文件映射的共享存储区的用法:

CreateFile()

from system page file:  
(HANDLE)0xffffffff







# 线程的同步与通信

---

除了共享存储区外，Windows还提供以下进程通信机制：

- ✓ 管道：无名管道，有名管道
- ✓ 信号
- ✓ 邮件槽：不定长、不可靠的单向消息通信机制
- ✓ Socket：网络通信机制



## 5.4 Windows 2000/xp的进程与内存管理

---

### 五、内存管理

#### 1. 内存管理器的组成部分

6个关键组件：

- ✓ 工作集管理器(MmWorkingSetManager)：当空闲内存低于某一界限值时，便启动所有的内存管理策略，如：工作集的修整、老化和已修改页的写入等
- ✓ 进程/堆栈交换器(KeSwapProcessOrStack)：完成进程和内核线程堆栈的换入和换出操作
- ✓ 已修改页写入器(MiModifiedPageWriter)：将已修改页链表上的“脏”页写回到页文件



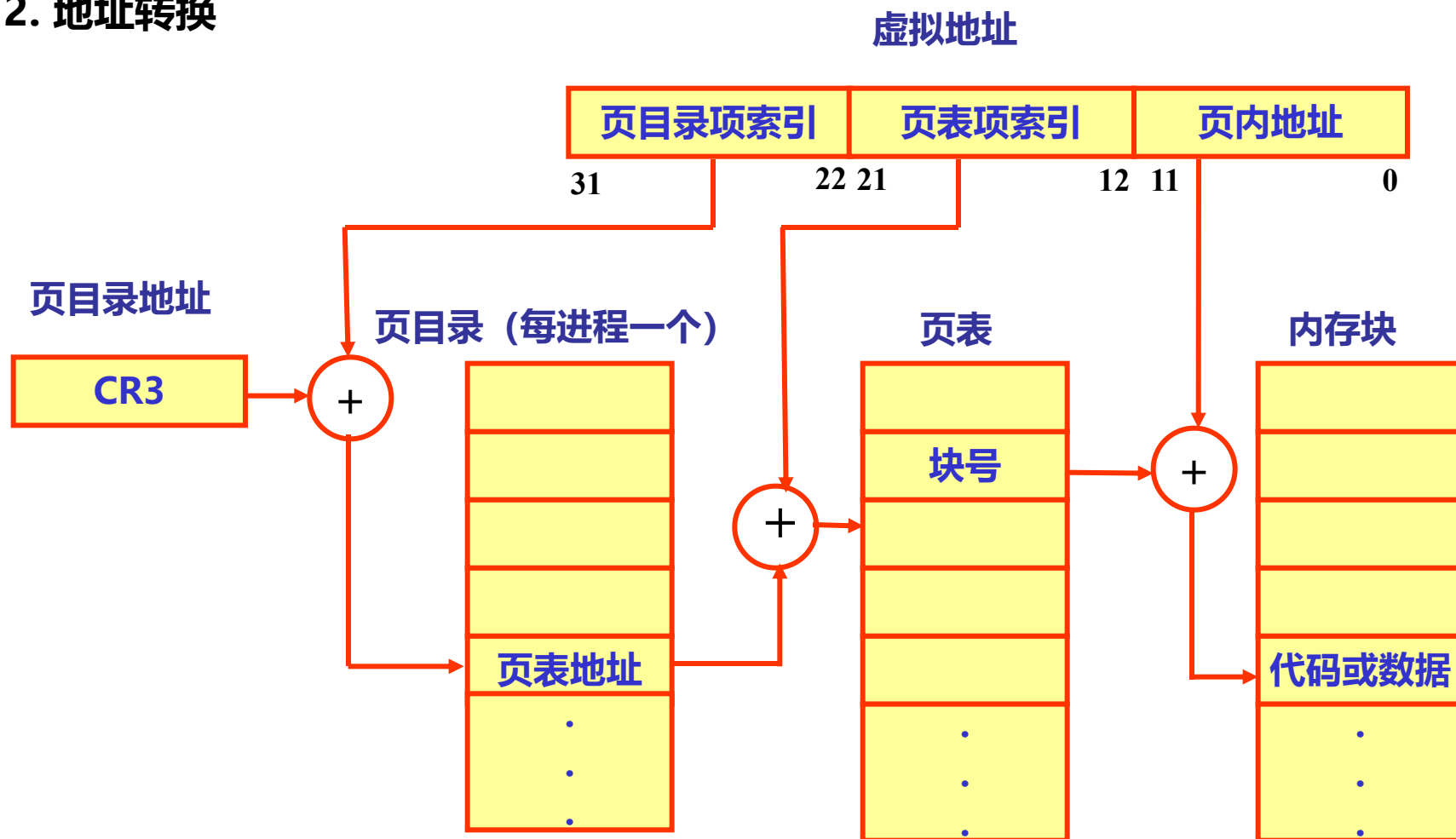
# 内存管理的组成

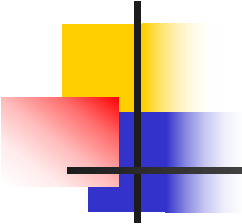
---

- ✓ 映射页写入器(MiMappedPageWriter): 将映射文件中脏页写回磁盘
- ✓ 废弃段线程(MiDereferenceSegmentThread): 负责系统高速缓存和页文件的扩大和缩小
- ✓ 零页线程(MmZeroPageThread): 将空闲链表中的页清零

# 内存管理

## 2. 地址转换



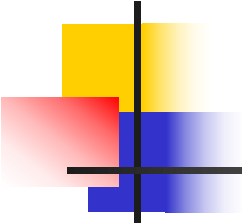


# 内存管理

---

## 说明:

- (1) 页内地址占12位, 页大小 = 4KB
- (2) 每个进程1个页目录, 每个页目录占4KB (1024项, 每项4B)
- (3) 页表大小是4KB (1024项, 每项4B)
- (4) 页表不常驻内存, 可换入/换出
- (5) 运行PAE (物理地址扩展) 内核的系统是三级页表



# 内存管理

---

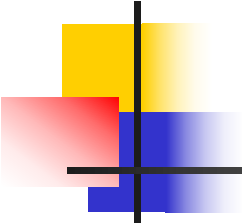
## 3. 页的调度策略

### (1) 取页策略

- ✓ 请求调页
- ✓ 提前取页：采用簇（cluster）方式将页装入内存

### (2) 置换策略

- ✓ 采用局部置换策略
- ✓ 在多处理器系统中，采用了FIFO置换算法。而在单处理器系统中，其实现接近于最近最少使用策略（LRU）算法（称为轮转算法）
- ✓ 使用“自动调整工作集”技术

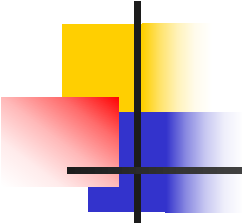


## 5.5 小结

---

### 一、Linux

1. Linux的3个接口
2. 系统调用
3. 用户态和核心态
4. Shell的功能
5. 进程创建与执行: `fork()`, `exec()`
6. 进程调度时机
7. 了解进程同步与通信机制及其适用场合  
信号, 消息队列, 共享内存, 信号量
8. 内存管理  
页式管理; 页的周期性换出; 交换缓冲区



## 5.5 小结

---

### 二、Windows 2000/XP

1. 线程优先级提升的思想

2. 进程/线程的创建

Createprocess(); CreateThread()

3. 了解线程同步与通信机制

同步对象，共享内存（内存映射文件）





# 第6章 文件系统

---

## 6.1 文件系统概述

## 6.2 文件的结构与存取方式

## 6.3 文件目录

## 6.4 空闲存储空间的管理

## 6.5 文件的使用、共享和保护

## 6.6 小结



## 6.1 文件系统概述

---

### 一、为什么要引入文件？为什么引入文件系统？

- ✓ 长期保存（大量的）数据
- ✓ 方便用户使用（包括共享）



## 6.1 文件系统概述

---

### 二、什么是文件？文件系统有何作用？

#### 1. 什么是文件？

文件是**有名字的**记录在**外存中**的一组有逻辑意义的数据项的序列

- ✓ 名字：文件名
- ✓ 数据项：构成文件内容的基本单位
- ✓ 长度：文件的字节数
- ✓ 文件内容的含义：由文件的建立者和使用者解释

# 什么是文件？

编号:    0                    1                    .....                    i                    .....                    n-1



**读写指针**

- 各数据项之间具有顺序关系



# 什么是文件？

---

## 文件命名：

每个操作系统都有自己的文件命名规则

- ✓ 长度
- ✓ 数字和特殊字符
- ✓ 是否大小写敏感
- ✓ 文件扩展名（一个或多个）

例子：.bak .c .exe .gif

.hlp .html .mpg .o

.doc .java .txt .zip



# 6.1 文件系统概述

---

## 2. 什么是文件系统

文件系统是OS中用来管理文件的那一部分软件

### 文件系统的功能：

- ✓ 统一管理文件的存储空间，实施存储空间的分配与回收
- ✓ 实现文件信息的共享，提供文件的保护和保密措施
- ✓ 实现文件的**按名访问**

**访问的透明性**：用户不关心文件的物理位置和存储结构

- ✓ 向用户提供一个方便使用的**接口**，提供对文件系统操作的命令
- ✓ 提供与I/O的**统一接口**



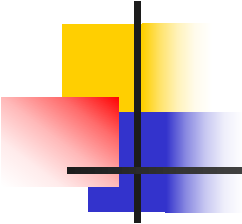
# 6.1 文件系统概述

---

## 3. 文件的分类

文件分类角度很多，比如：

- ✓ 按文件的性质和用途：系统文件；用户文件；库文件
- ✓ 文件中数据的形式：文本文件；二进制文件
- ✓ 文件的保护方式：只读文件；读写文件；可执行文件
- ✓ **文件的逻辑结构：流式文件；记录式文件**
- ✓ **文件的物理结构：顺序（连续）文件；链式文件；索引文件**



# 文件分类

---

Unix系统将文件分为3类:

- ✓ 普通文件(regular): ASCII或二进制文件
- ✓ 目录文件(directory)
- ✓ 特殊文件: 设备文件, 管道, 套接字 (socket) , 符号链接等





## 6.1 文件系统概述

---

### 三、看待文件系统的两种观点

#### ✓ 用户观点：

文件系统如何呈现在用户面前：一个文件由什么组成，如何命名，如何保护文件，可以进行何种操作等等

尽可能方便用户使用

#### ✓ 系统观点：

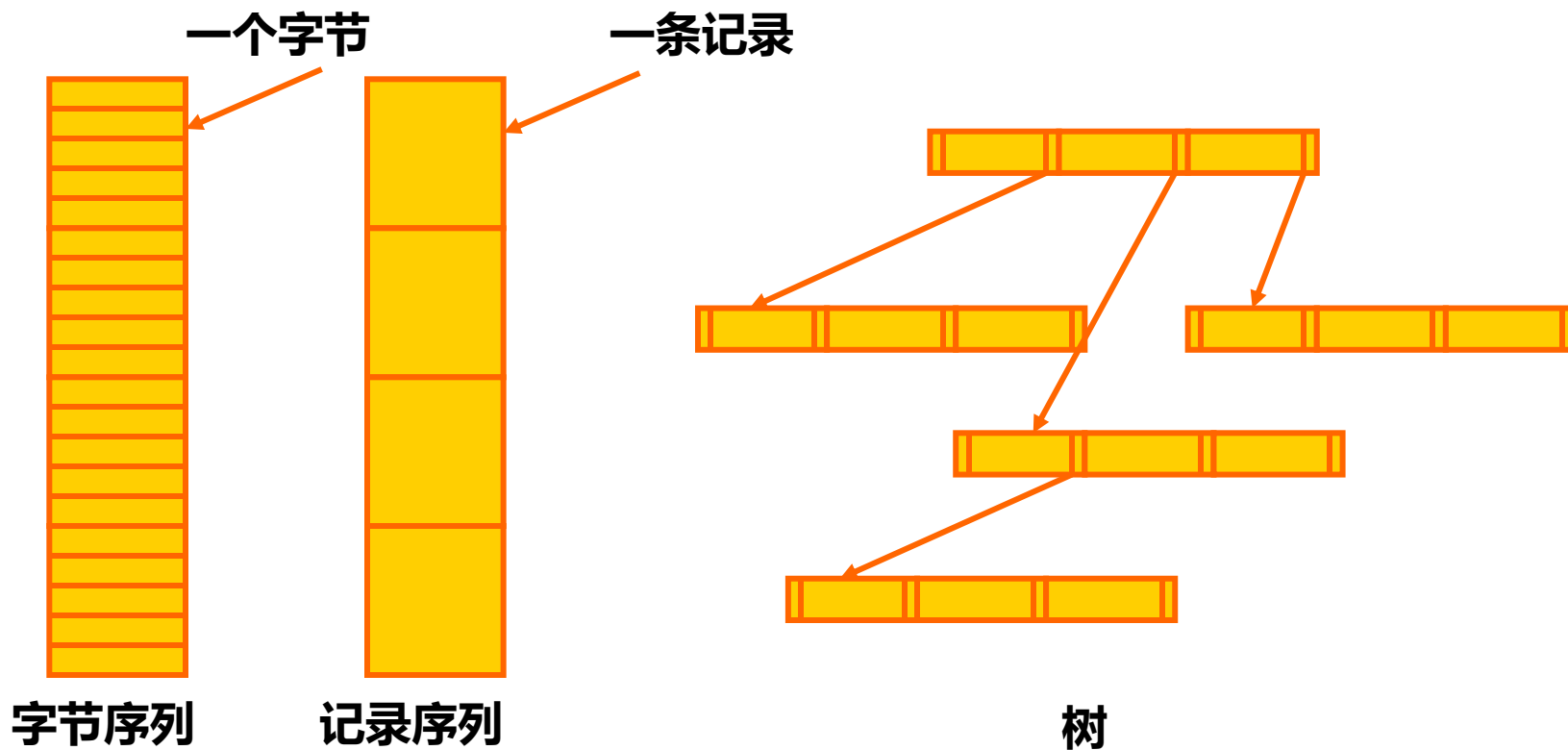
文件目录怎样实现，怎样管理文件存储空间，文件存储位置，与设备管理的接口，等等

尽可能提高效率（时间，空间）

## 6.2 文件的结构与存取方式

### 一、文件的逻辑结构

从用户的观点看文件的组织形式





# 文件的逻辑结构

---

一般分为2类：

(1) **字节流文件**（流式文件）：无结构文件

文件：**一个无结构的字节序列，其含义由使用者解释**

Unix的所有文件都看作字节流文件

好处：非常灵活

(2) **记录文件**：有结构文件

文件：记录的序列，每条记录有其内部结构

记录：定长，不定长



## 6.2 文件的结构与存取方式

---

### 二、文件的物理结构

文件的存储结构，指文件在外存上的存放方式

即从系统的角度看文件的组织方式

基本结构有3种：

- ✓ 连续结构（顺序结构）
- ✓ 链式结构（串联结构）
- ✓ 索引结构



# 文件的物理结构

---

## (1) 连续结构

文件的数据存放在若干连续的物理块中

### 优点:

- ✓ 简单，只要记住首块的地址和文件长度即可
- ✓ 支持顺序存取和随机存取
- ✓ 顺序存取速度快
- ✓ 所需的磁盘寻道时间最少

# 连续文件

磁盘空间

count

0 1 2 3

f

4 5 6 7

8 9 10 11

tr

12 13 14 15

mail

16 17 18 19

20 21 22 23

24 25 26 27

list

28 29 30 31

文件目录

| 文件名 | 始址 | 块数 |
|-----|----|----|
|-----|----|----|

|       |   |   |
|-------|---|---|
| count | 0 | 2 |
|-------|---|---|

|    |    |   |
|----|----|---|
| tr | 14 | 3 |
|----|----|---|

|      |    |   |
|------|----|---|
| mail | 19 | 6 |
|------|----|---|

|      |    |   |
|------|----|---|
| list | 28 | 4 |
|------|----|---|

|   |   |   |
|---|---|---|
| f | 6 | 2 |
|---|---|---|



# 连续文件

---

## 缺点:

- ✓ 不利于文件的动态增长  
若预留空间：浪费，而且预先不知道文件的最大长度  
否则需要重新分配和移动
- ✓ 不利于文件内容的插入和删除
- ✓ 存在外部碎片问题



# 文件的物理结构

---

## (2) 链式结构

一个文件的数据存放在若干不连续的物理块中，各块之间通过指针连接，每个物理块指向下一个物理块

### 优点：

- ✓ 提高了磁盘空间利用率
- ✓ 不存在外部碎片问题
- ✓ 有利于文件的插入和删除
- ✓ 有利于文件的动态扩充

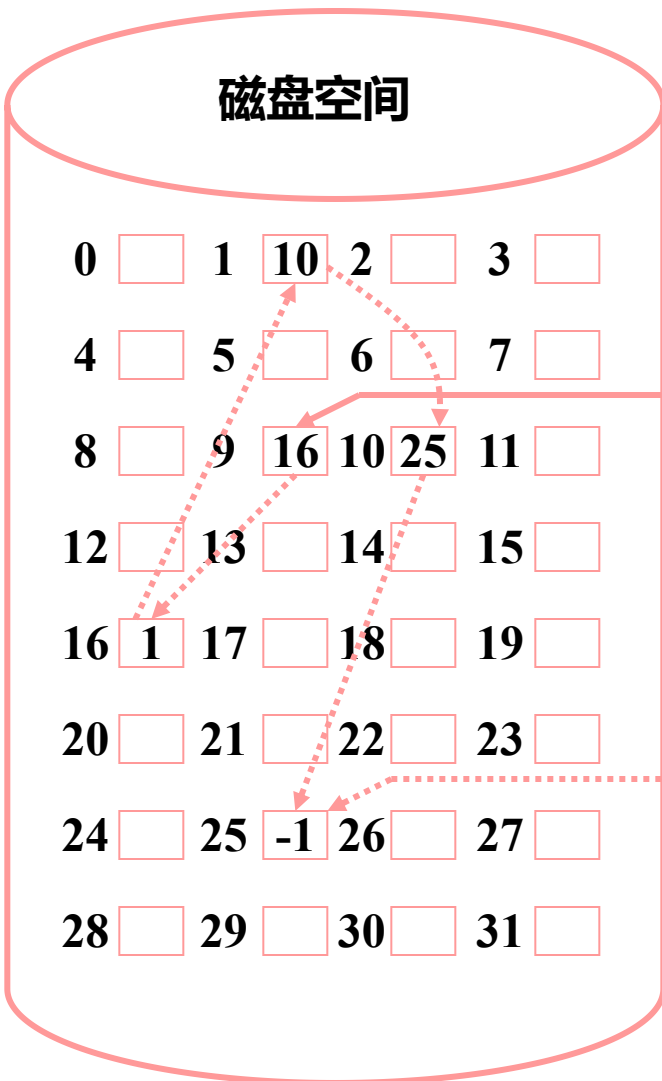


# 链式文件

文件目录

| 文件名  | 始址 | 末址 |
|------|----|----|
| jeep | 9  | 25 |

磁盘空间





# 链式文件

---

## 缺点:

- ✓ 随机存取相当缓慢
- ✓ 需要更多的寻道时间
- ✓ 链接指针占用一定的空间

## 链接结构的变形:

### **文件分配表FAT** (File Allocation Table)

一般以簇为单位分配空间，簇由若干连续的物理块组成



# 链式文件

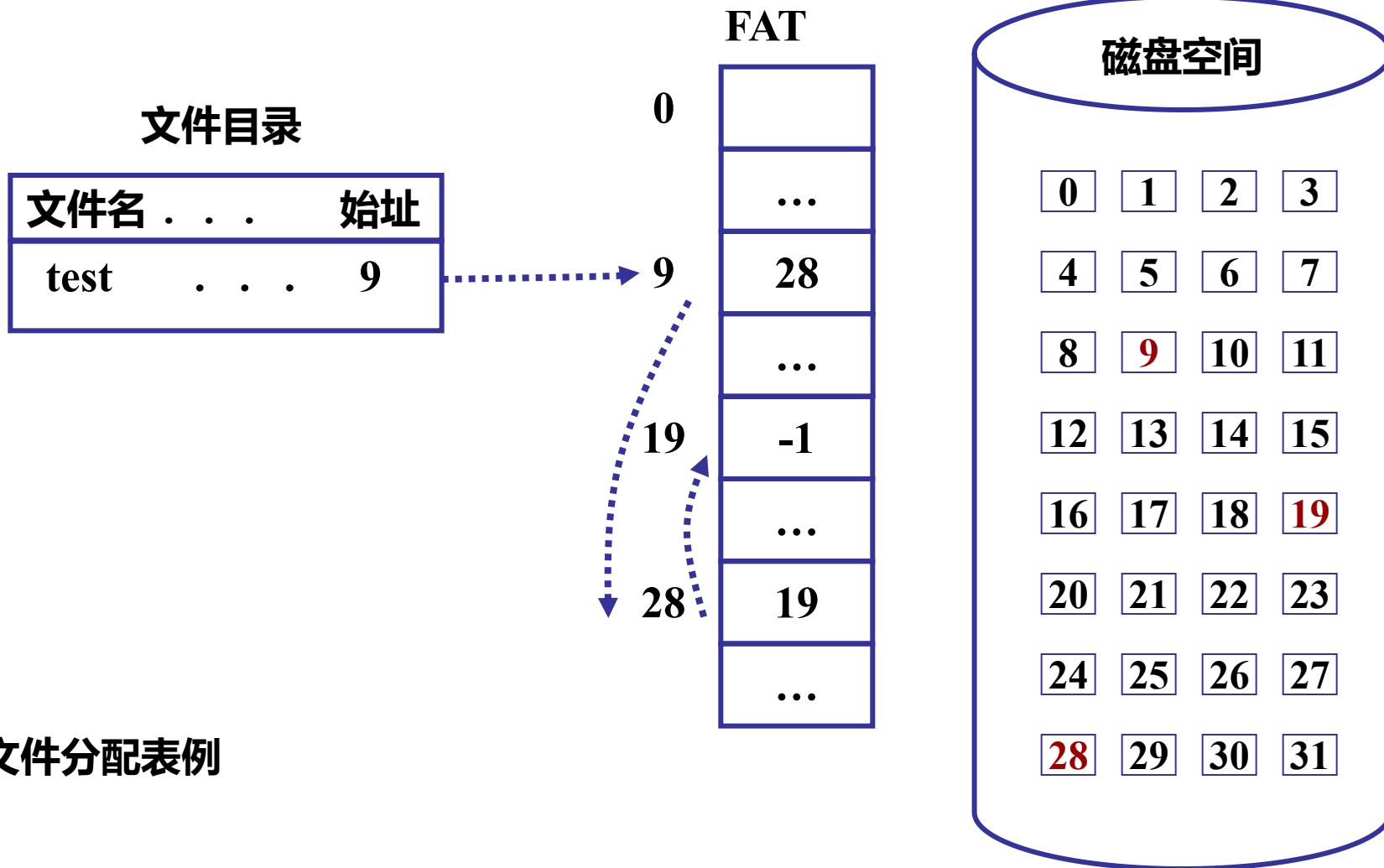
---

## **文件分配表FAT的一种实现：**

磁盘的每个分区包含一个FAT，分区中的每个盘块在其中占有1项（以块号为索引），指出文件中下一块的块号。

在目录项中包含文件首块的块号。

# 链式文件





# 文件的物理结构

---

## (3) 索引结构

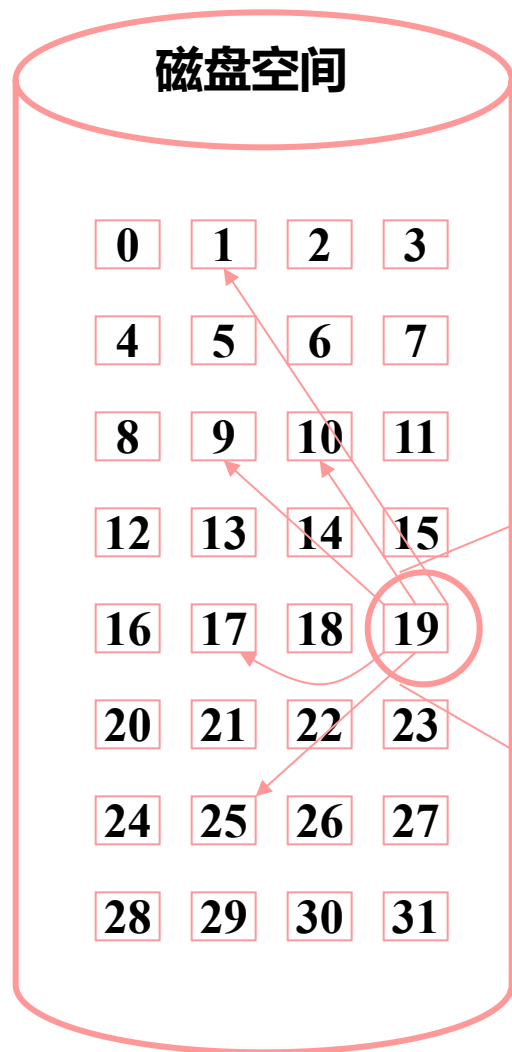
一个文件的数据存放在若干不连续的物理块中，系统为每个文件建立一个专用数据结构--索引表。

索引表存放逻辑块号与物理块号的对应关系

一个索引表就是磁盘块地址（块号）数组，其中第 $i$ 个条目存放的是逻辑块号 $i$ 对应的物理块号

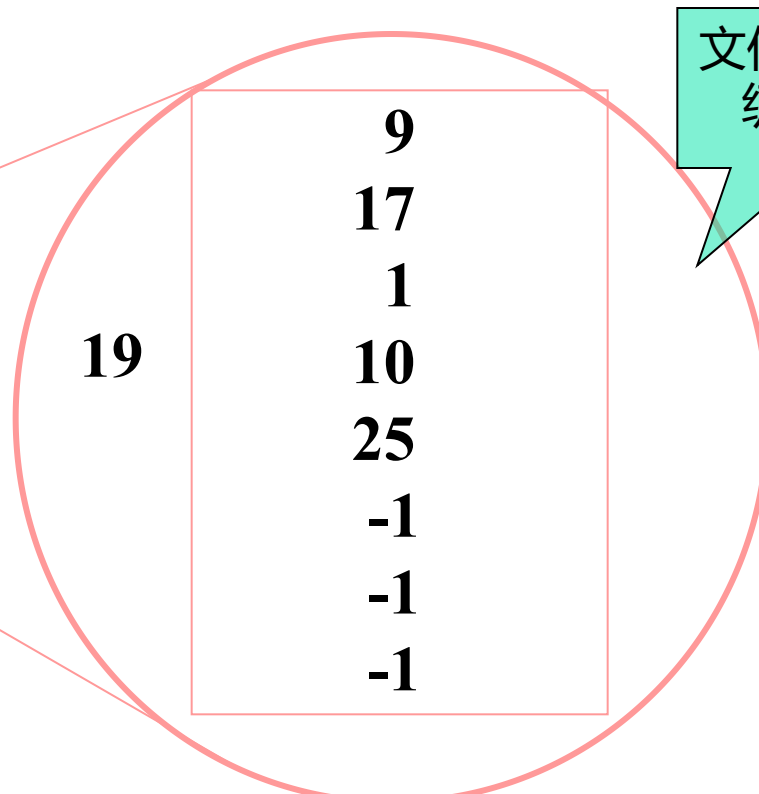
文件目录的目录项中指出索引表的物理地址

# 索引结构



## 文件目录

| 文件名  | 索引表地址 |
|------|-------|
| test | 19    |



文件test的单  
级索引表



# 索引结构

---

**优点：**保持了链接结构的优点，又避免了其缺点

- ✓ 既能顺序存取，又能随机存取
- ✓ 能满足文件动态增长、插入、删除的要求
- ✓ 能充分利用外存空间

**缺点：**

索引表本身带来了系统开销



# 索引结构

---

一般来说，小文件的索引表保存在一个单独的物理块中。

如果文件很大，索引表较大，超过了一个物理块，就必须考虑索引表的组织方式

## 索引表的组织:

### (1) 连续方式

索引表占用多个连续的盘块

### (2) 链接方式

索引表按照链式结构组织，占用多个不连续的盘块

### (3) **多级索引** (多重索引)

例如，二级索引：将一个大文件的所有索引表（二级索引）的地址放在另一个索引表（一级索引）中

此外，还有三级索引等。





# 索引结构

---

UNIX文件系统采用的是多级混合索引结构。

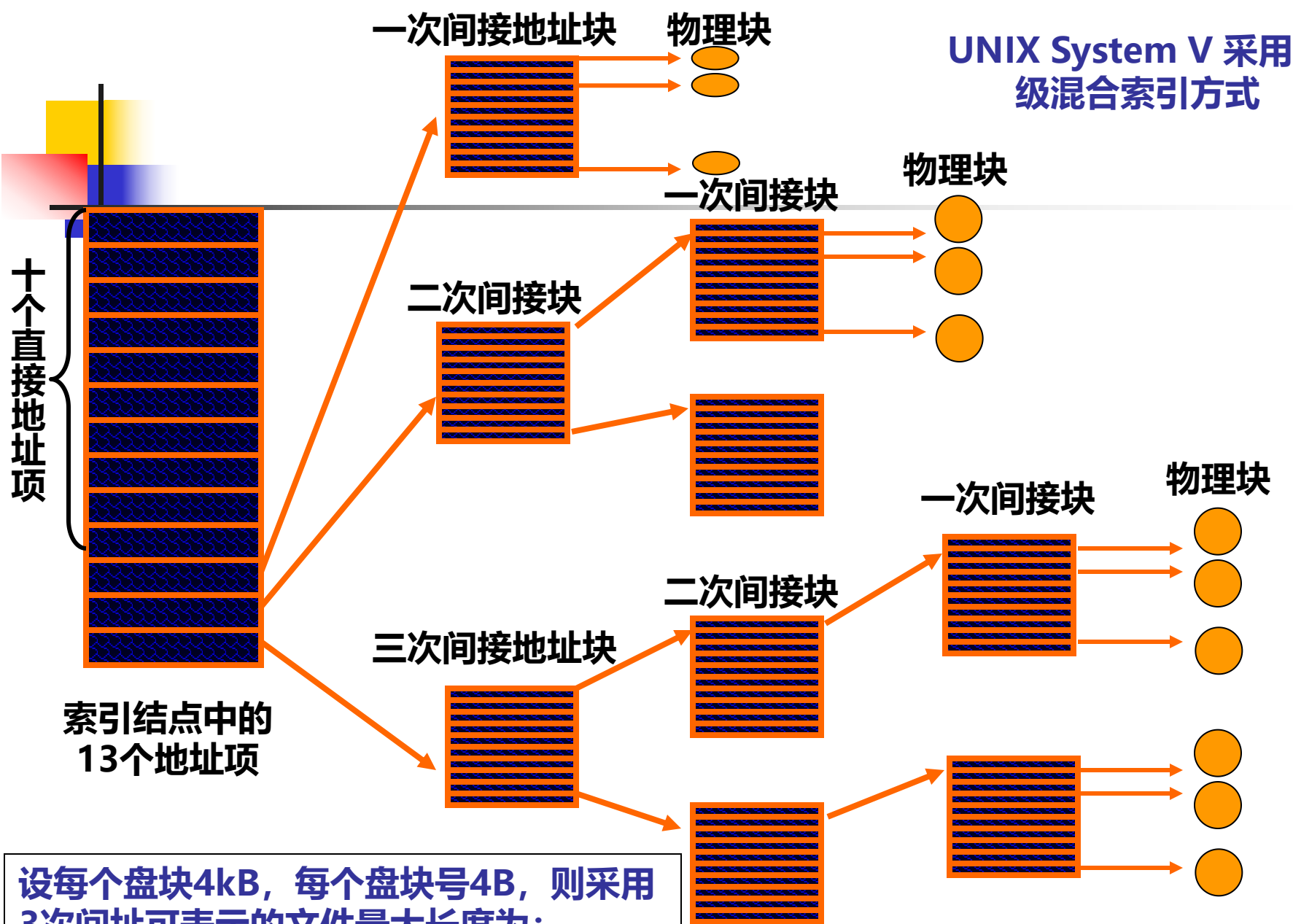
每个文件的索引表为13个索引项

最前面10项直接登记存放文件数据的物理块号（直接寻址）

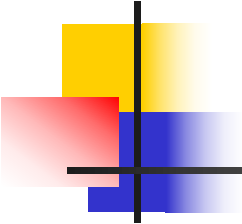
如果文件大于10块，则利用第11项指向一个物理块，该块中最多可放256个文件物理块的块号（一次间接寻址）。对于更大的文件还可利用第12和第13项作为二次和三次间接寻址

UNIX中采用了三级索引结构后，文件最大可达16M个物理块

# UNIX System V 采用多级混合索引方式



设每个盘块4kB，每个盘块号4B，则采用3次间址可表示的文件最大长度为：  
 $4T + 4G + 4M + 40K(B)$



## 6.2 文件的结构与存取方式

---

### 三、文件的存取方式

主要有顺序存取和随机存取两种。

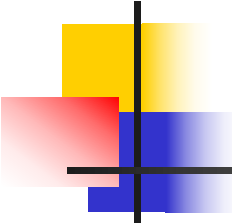
- ✓ 顺序存取

对文件中的数据按逻辑顺序进行读/写的存取方式

- ✓ 随机存取

对文件中的数据按任意顺序进行读/写的存取方式

- ✓ 按键 (key) 存取：如DBMS



## 6.2 文件的结构与存取方式

### 四、文件结构、存取方式与存储介质的关系

文件的存取方式不仅与文件的结构有关，还与文件所在存储介质的特性有关，如下表所示：

| 存储介质 | 磁带   | 磁盘 |    |    |
|------|------|----|----|----|
| 物理结构 | 连续结构 | 连续 | 链接 | 索引 |
| 存取方式 | 顺序存取 | 顺序 | 顺序 | 顺序 |
|      |      | 随机 |    | 随机 |



## 6.3 文件目录

---

### 一、几个基本概念

#### 1. 文件控制块 (File Control Block, FCB)

文件控制块是操作系统为管理文件而设置的数据结构，存放了为管理文件所需的所有相关信息（**文件属性**）

也称为**文件说明**，或**文件目录项**

**文件控制块是文件存在的标志**

#### 文件控制块的内容：

- ✓ 基本信息：**文件的名字、地址**（起始物理块号）、长度、结构（逻辑结构、物理结构）、类型
- ✓ 存取控制信息：文件属主（owner）、存取权限或口令
- ✓ 使用信息：共享计数，文件的建立、修改日期等



## 6.3 文件目录

---

### 2. 文件目录

把所有的FCB组织在一起，就构成了文件目录  
即文件控制块的有序集合

### 3. 目录项

构成文件目录的项目（目录项就是FCB）

### 4. 目录文件

为了实现对文件目录的管理，通常将文件目录以文件的形式保存在外存，  
这个文件就叫目录文件

目录主要是为了系统快速实现“按名访问”而引入的，

**查目录是文件系统最频繁的操作**，因此目录的合理组织很重要



## 6.3 文件目录

---

### 二、目录结构

#### 1. 单级目录结构

为所有文件建立一个目录文件（组成一线性表）

优点：简单，易实现

缺点：

- ✓ 限制了用户对文件的命名（容易出现“命名冲突”）
- ✓ 顺序检索文件时，平均检索时间长
- ✓ 不利于对文件的共享



# 目录结构

---

## 2. 二级目录结构

目录分为两级：

一级称为主文件目录，给出用户名，用户子目录所在的物理位置；

二级称为用户文件目录（又称用户子目录），给出该用户所有文件的FCB

优点：一定程度上解决了文件的重名和文件共享问题

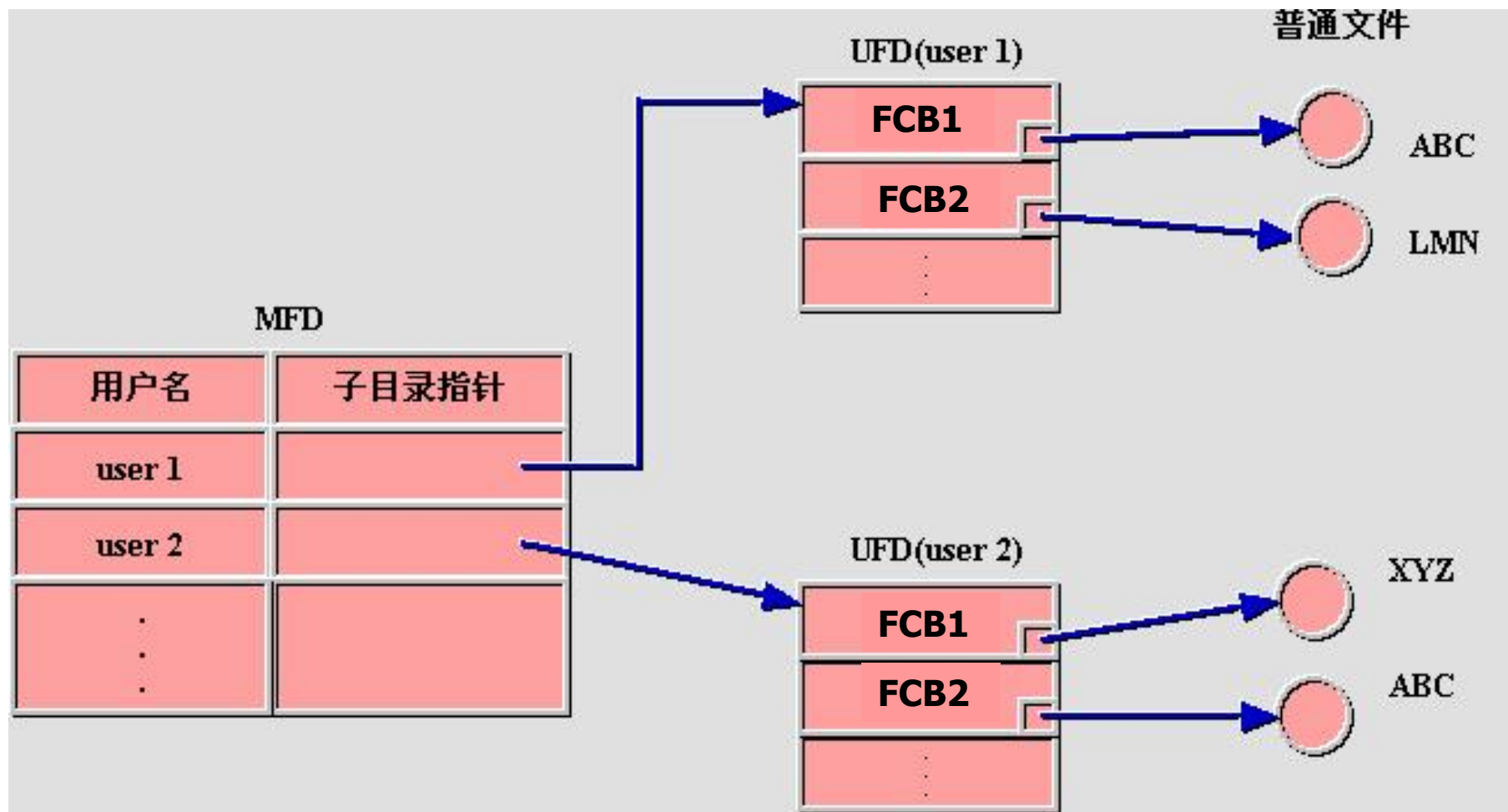
用户名|文件名

查找时间缩短

缺点：增加了系统开销



## 二级目录结构





# 目录结构

---

## 3. 多级目录结构（树型目录）

对二级目录简单扩充可得三级或三级以上的多级目录结构，即允许每一级目录中的FCB要么指向文件，要么指向下一级子目录。这是当今主流OS普遍采用的目录结构

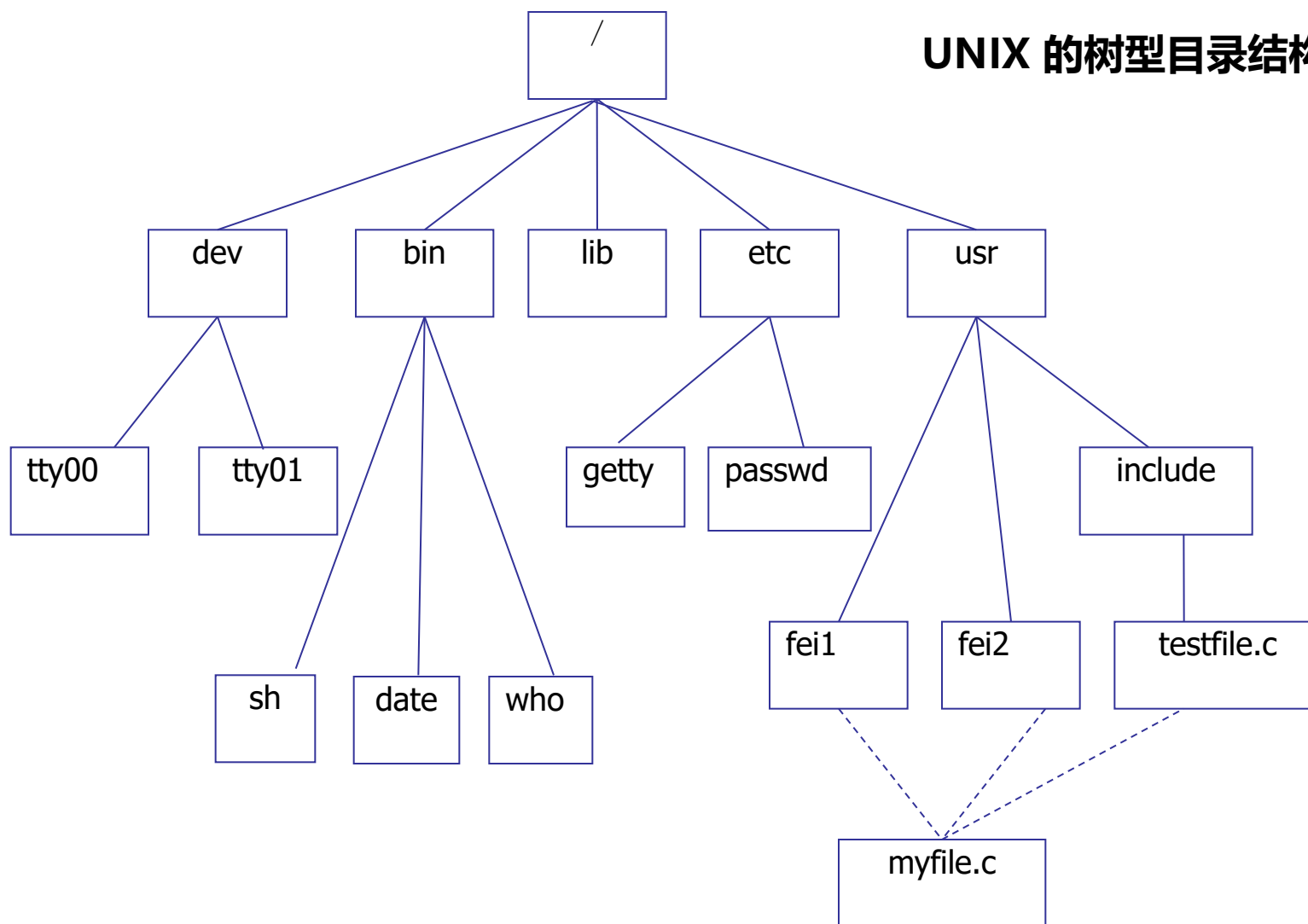
优点：

层次结构清晰，便于管理和保护；有利于文件分类；

能较好地避免重名问题；提高了文件检索速度；有利于访问权限的控制

# 树型目录结构

## UNIX 的树型目录结构





## 6.3 文件目录

---

### 三、文件目录检索

访问文件时，必须首先确定读写文件的地址，需要下列2步：

- (1) **目录检索**：根据文件名，查目录，确定文件的起始地址。
- (2) **文件寻址**：确定所要访问文件内容的起始位置（地址）。



## 6.3 文件目录

---

### 1. 目录检索

文件的“按名存取”是通过查目录实现的，系统按照文件的路径名检索

基本的目录检索技术主要有：

- ✓ 线性检索法
- ✓ Hash方法

为了加快目录检索，许多系统引入当前目录（工作目录）、相对路径名等。



## 6.3 文件目录

---

### 2. 文件寻址

根据目录项（FCB）中记录的文件物理地址等信息，  
求出文件的任意记录或字节在存取介质上的地址

文件寻址与文件的物理结构和逻辑结构以及设备的物理特性有关  
文件的内容是以块为单位存储的。

但存取文件时，对于记录式文件，是以逻辑记录为单位提出存取要求的，因此，  
存储介质上的物理块长度与逻辑记录的长度是否匹配直接影响到对文件的寻址



## 6.3 文件目录

---

### 四、文件目录的实现

#### 1. 把文件说明信息 (FCB) 都放在目录项中

当查找文件时，需要依次将存放目录的物理块装入内存，逐一比较文件名，直到找到为止。

例如，文件说明占128B，每块512B，则每块可存放4个目录项，100个文件就需要25块。

设目录文件占用的盘块数是N个，则要找到一个目录项，

平均需要读入多少个盘块？

$$(N+1)/2 \text{ 块}$$



# 文件目录的实现

---

**把文件说明信息（FCB）都放在目录项中的缺点：**

查找文件缓慢，因为目录项较大

文件目录平常放在外存中，当文件很多时，可能占用大量的外存物理块

检索目录时，需要文件说明中的什么信息？

文件名即可。





# 文件目录的实现

---

## 2. 将文件说明分成2部分（目录项分解法）

把FCB分成两部分：

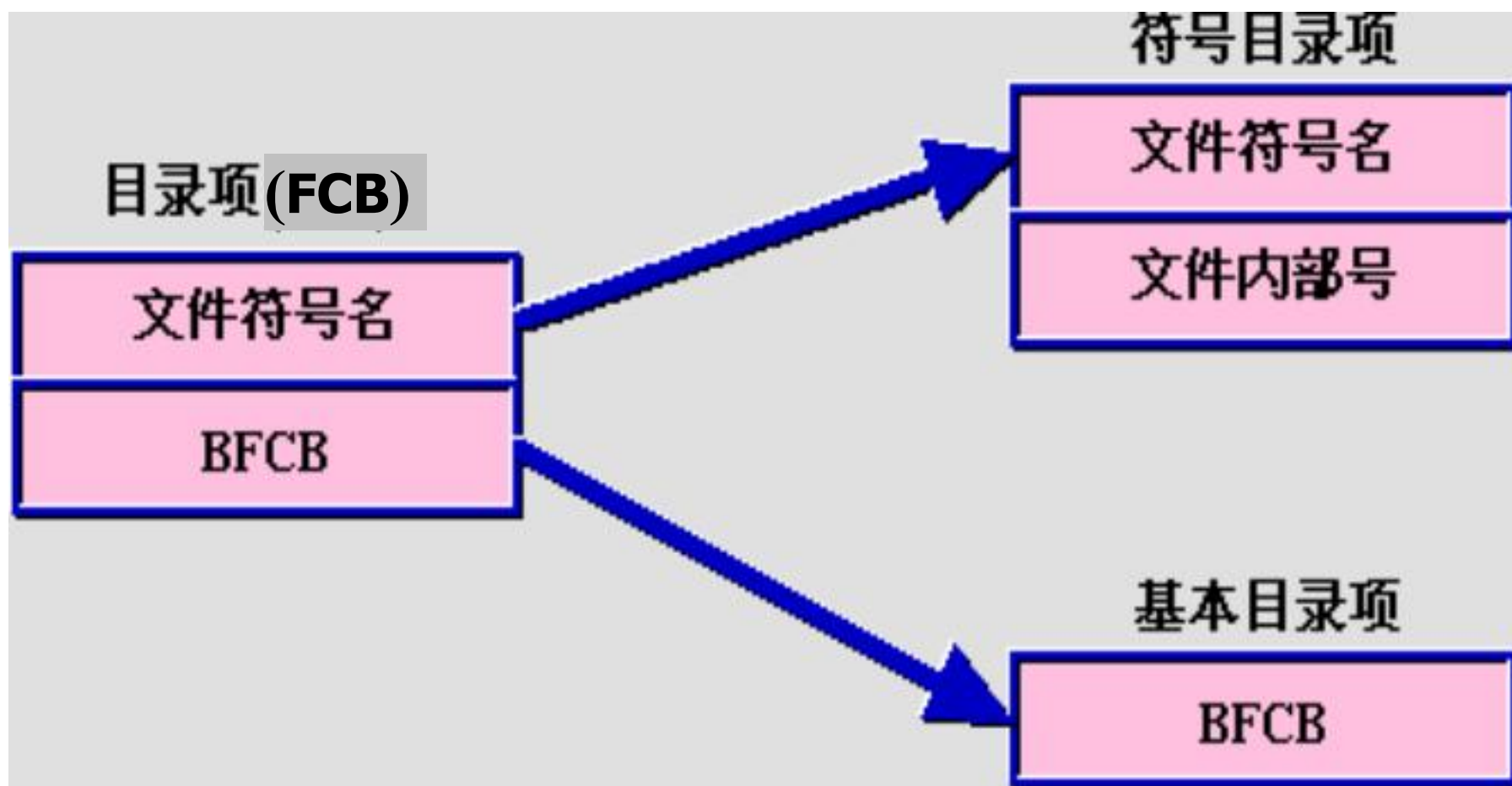
### ① 符号目录项

文件名，文件号（基本目录项编号）

### ② 基本目录项

除文件名外的所有文件说明

# 目录项分解法





# 目录项分解法

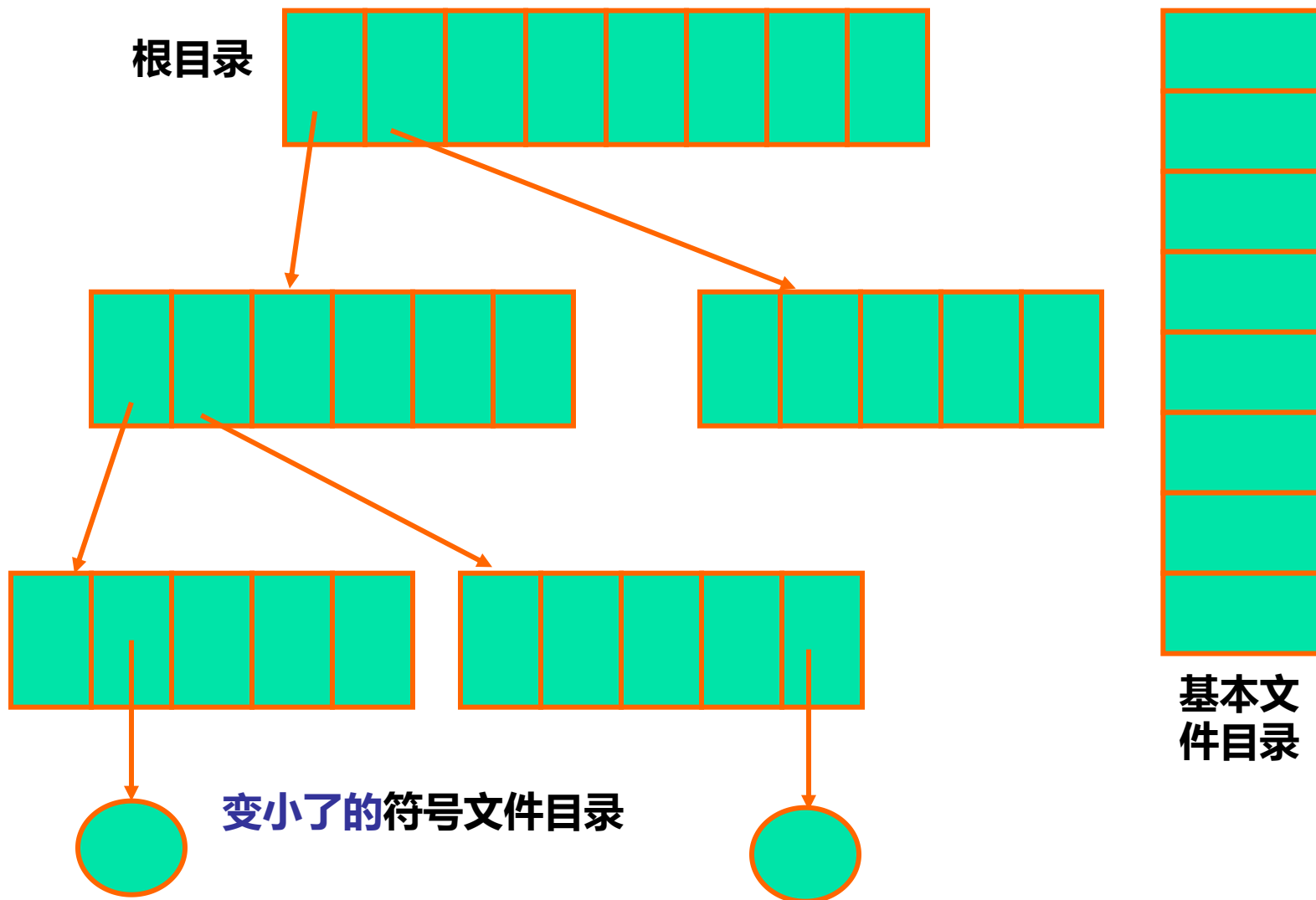
---

## 目录项分解法的典型实现:

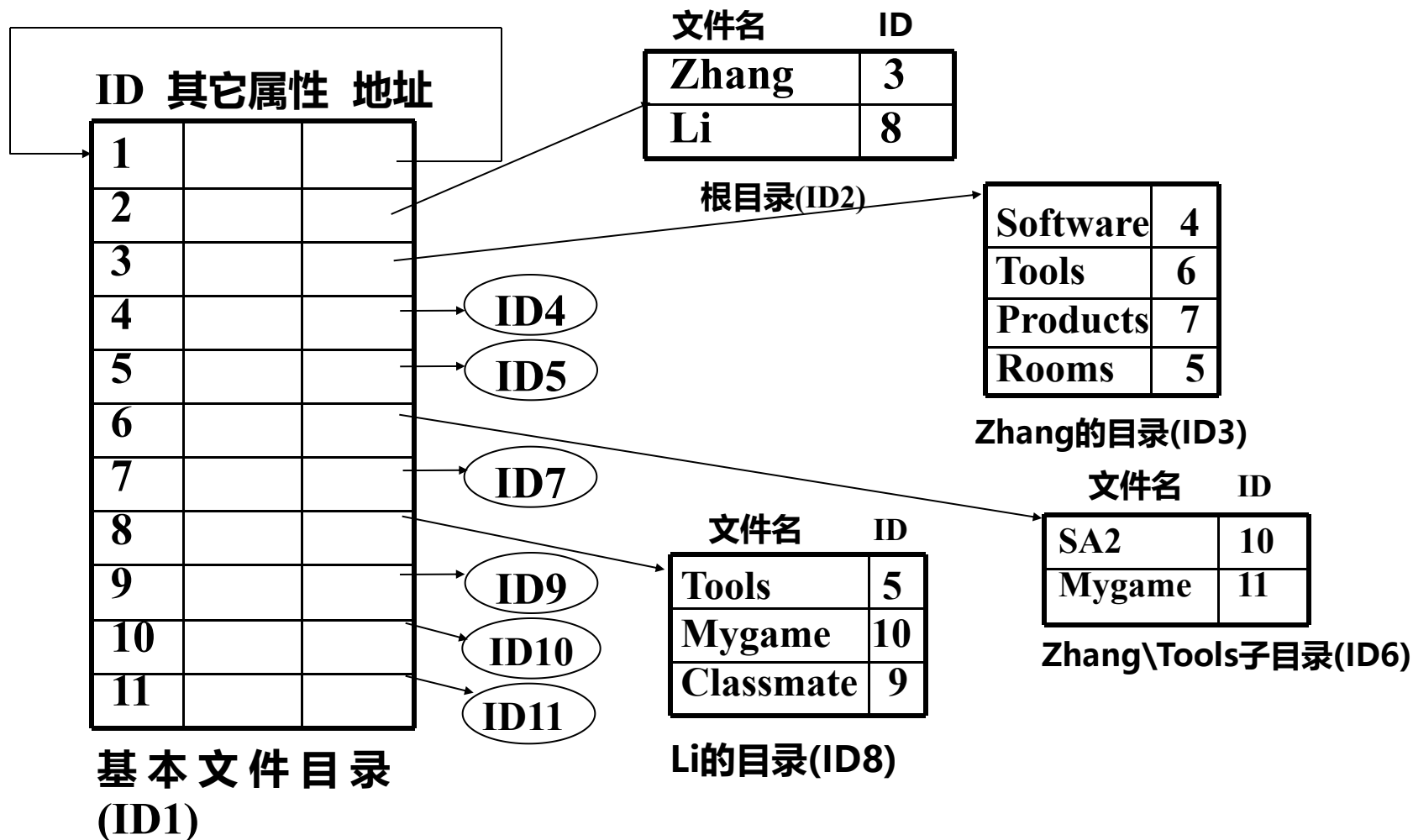
- (1) 基本文件目录 + 符号文件目录
- (2) 目录项 + I节点

Unix/Linux采用此方法，它把符号目录项称为目录项，而把基本目录项称为**I节点**（Index node，**索引节点**），这样，目录项中的文件号就是**I节点号**。

# 目录项分解法



# 目录项分解法





# 文件目录的实现

---

**【例】** 设磁盘物理块大小为512个字节，一个FCB有64个字节，符号目录项占8个字节，其中文件名占6个字节，文件号占2个字节，基本目录项占 $64 - 6 = 58$ 个字节。若把含有256个目录项的某目录文件改造成符号文件目录和基本文件目录的结构，试说明改造前后查找一个文件的平均访问磁盘块数。



# 文件目录的实现

---

**解：**分解前：1块含 $512/64=8$ 个FCB

分解后：1块含 $512/8=64$ 个符号目录项

该目录文件含有256个目录项

分解前占 $256/8 = 32$ 块

分解后其符号文件目录占 $256/64 = 4$ 块

故查找一个文件的平均访问磁盘块数：

分解前： $(1 + 32)/2 = 16.5$

分解后： $(1 + 4)/2 + 1 = 3.5$

由此可见：改造后减少了访问磁盘的次数，提高了检索速度。



# 文件目录的实现

---

目录的其他实现方法：

✓ Hash表

目录文件按目录项键（key）的Hash值进行顺序组织。

创建或搜索时，根据文件名计算Hash值，得到一个指向目录表中相应表目的指针

✓ 其他方法：

如B+树，这是一种将大的单级索引目录文件组织成有序的树型多级索引目录文件的方法，是索引顺序文件中实际采用的基本索引结构，支持随机访问和顺序访问，多见于DBMS中。

NTFS文件系统就采用了B+树





## 6.4 空闲存储空间的管理

---

### 一、空闲区表

将所有空闲区记录在一个表中。

适合连续文件的外存分配与回收。如今很少用

## 6.4 空闲存储空间的管理

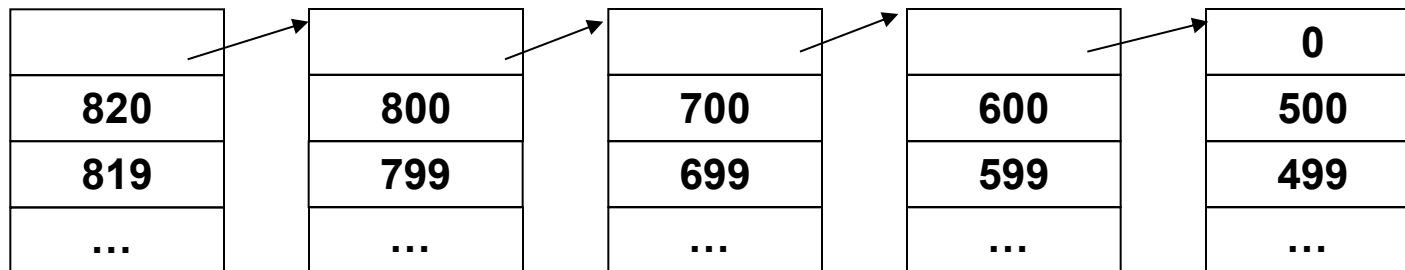
### 二、空闲块链

把所有空闲块链成一个链。适合离散分配

Windows、Unix使用类似方法

空闲块如何链接？

- (1) 每个空闲块中指出下一个空闲块的块号。
- (2) 采用多个空闲块构成的链表存放空闲块号。





# 空闲块链

---

扩展:

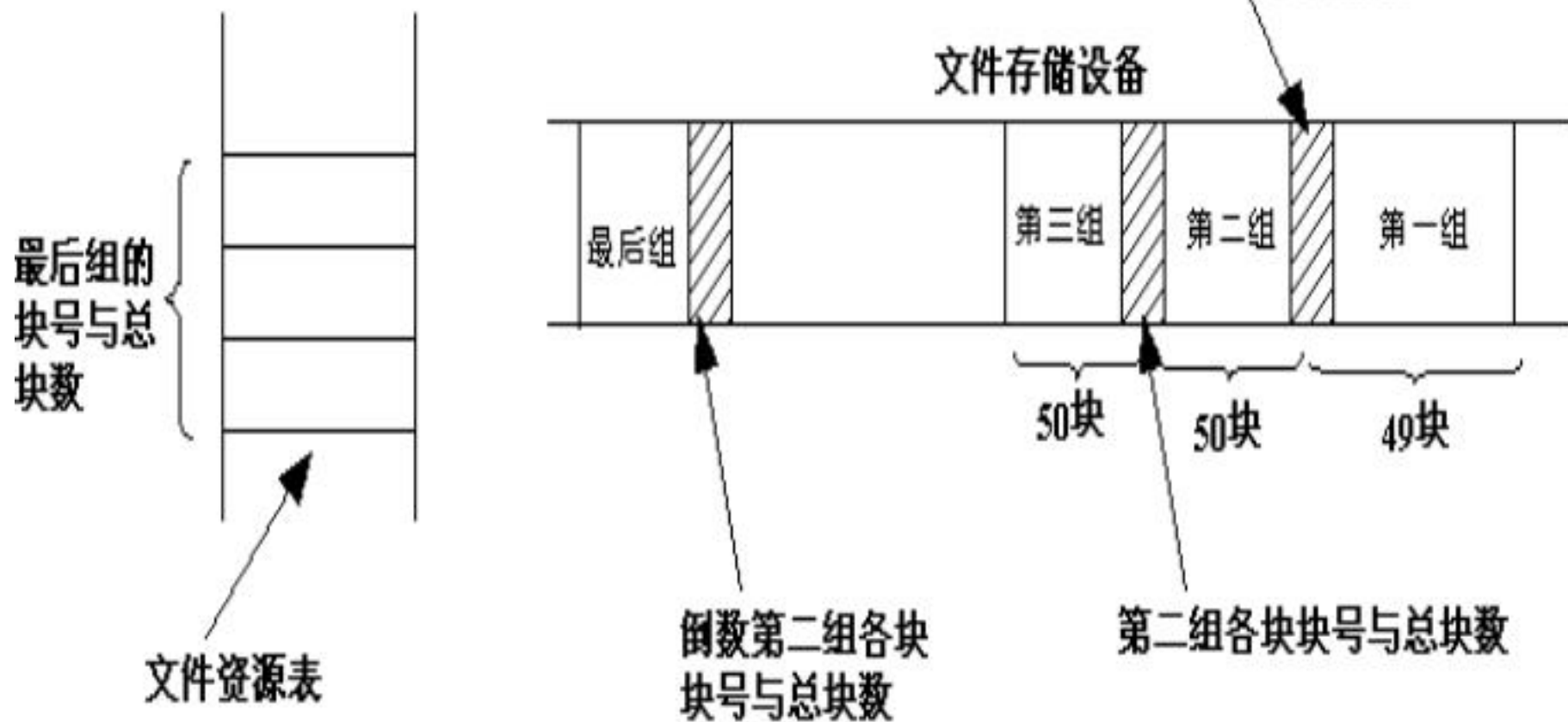
①不断地适度增加块大小

从最早的512B  $\Rightarrow$  1KB  $\Rightarrow$  2KB  $\Rightarrow$  4KB  $\Rightarrow$  8KB  $\Rightarrow$  16KB  $\Rightarrow$  32KB  $\Rightarrow$  64KB。

②成组链接法

链上每个节点记录1组空闲块。适合大型文件系统，分配、释放快，链占用空间少（除首组外均隐藏在空闲块中）。UNIX用之

# 成组链接法





## 6.4 空闲存储空间的管理

---

### 三、位图

- ✓ 用一串二进制位反映磁盘空间的分配情况，每个物理块对应1位，已分配的物理块为1，否则为0
- ✓ 申请物理块时，可以在位图中查找为0的位，返回对应的物理块号
- ✓ 归还时，将对应位设置为0
- ✓ 描述能力强，适合各种物理结构



## 6.5 文件的使用、共享和保护

---

### 一、文件的使用

为方便用户使用文件，文件系统提供对文件的各种操作，使用的形式包括系统调用或命令

- ① 提供设置和修改用户对文件访问权限的操作
- ② 提供建立、修改、删除目录的操作
- ③ 提供文件共享、设置访问路径的操作
- ④ 提供创建、**打开、读、写、关闭**、删除文件等操作

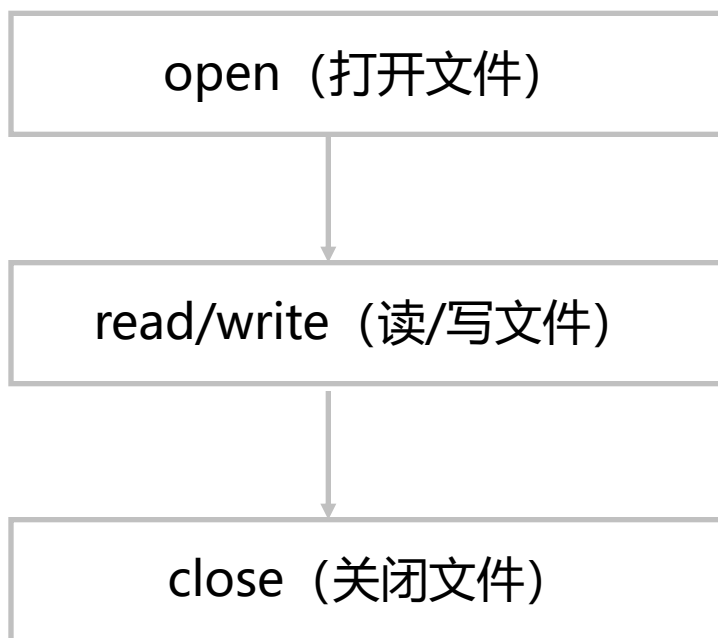
其中，最基本的操作是：打开、关闭、读、写文件等



# 文件的使用

---

## 1. 对文件的基本操作方式





# 文件的使用

---

## 2. 为什么要open/close文件?

**open:** 把文件说明信息 (FCB) 装入内存, 便于以后的快速访问。

- (1) 根据指定的文件路径名, 查目录, 找到相应文件的目录项, 检查权限;
- (2) 将文件说明信息装入内存;
- (3) 分配一个文件id (整数)。后面通过该id实施对该文件的操作。

**close:**

- (1) 释放文件说明信息所占的内存空间;
- (2) 把文件缓冲区中已修改的内容写回文件。

很多系统限制进程打开文件的个数, 用户尽可能要关闭不再使用的文件。





## 6.5 文件的使用、共享和保护

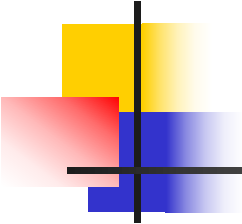
---

### 二、文件共享

文件共享：一个文件被多个用户或进程使用

共享的目的：

- ✓ 节省时间和存储空间，减少用户工作量
- ✓ 进程间通过文件交换信息



# 文件共享

---

## 1. 普通的文件共享方法

### (1) 按路径名访问共享文件

实现简单，不需要建立另外的目录项

但路径名可能长，检索较慢

### (2) 链接法

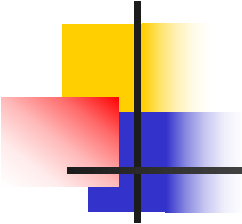
在相应目录项之间建立链接。即一个目录项中含有指向另一个目录项的指针。

#### 实现方法：

在目录项中设置一个“链接属性”，

表示目录项中的“物理地址”是指向另一目录项的指针。

同时，在共享文件的目录项中包含“用户计数”。



# 文件共享

---

## (3) 基本文件目录BFD

- ✓ 整个文件系统有1个基本文件目录BFD:

每个文件（及目录）有1个目录项，包含系统赋予的唯一标识符ID（整数）  
以及其他的文件说明信息

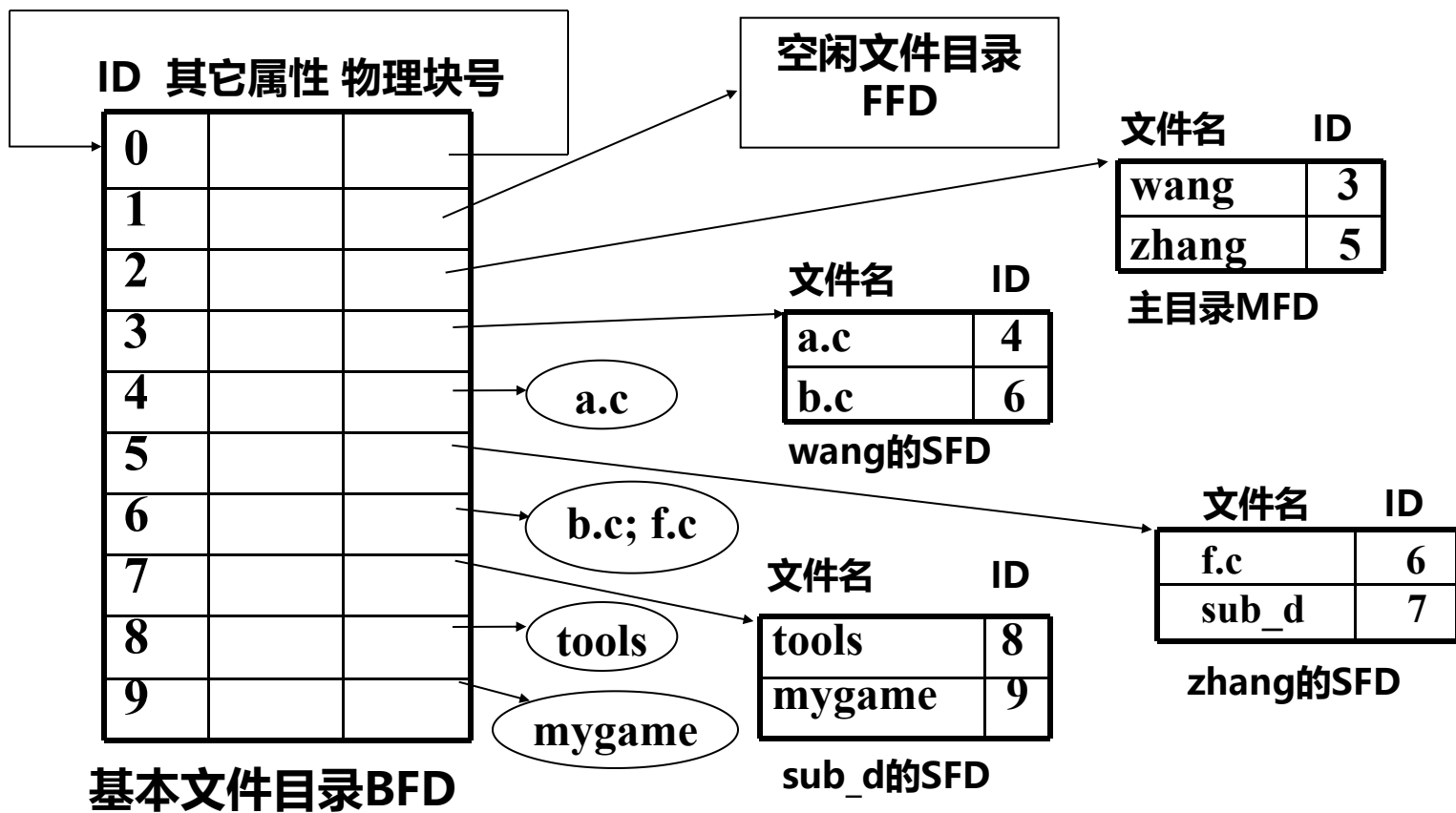
- ✓ 每个目录有1个符号文件目录SFD：除了ID = 0, 1, 2外，  
每个目录项仅包含文件名和ID

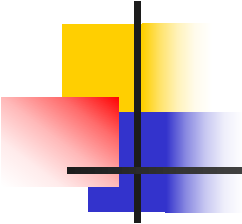
- ✓ 系统把ID = 0, 1, 2的目录项分别作为BFD、FFD、MFD的标识符

- ✓ **共享方法：**

若一个用户想共享另一用户的文件，只需在自己的目录文件中增加一个目录项，  
填上自己起的文件名和该共享文件的唯一ID即可。如ID = 6的文件。

# 文件共享





# 文件共享

---

## 2. 基于I节点的文件共享方法 (Unix采用)

### (1) 硬链接

多个目录项指向同一个I节点。

引入链接计数count: 表示链接到本I节点的文件数

# 硬链接

## 【例】

1) 用户A创建1个新文件name1

/dirA中的目录项

**文件名**    **I节点号**

|       |      |
|-------|------|
| name1 | 1234 |
|-------|------|

|           |
|-----------|
| ...       |
| owner = A |
| count = 1 |
| 5678      |

**I节点1234**

**文件内容**

"This is  
the text  
in the file"

**块5678**

# 硬链接

2) 用户B共享该文件，命名为name2

/dirB中的目录项

| 文件名   | I节点号 |
|-------|------|
| name2 | 1234 |

/dirA中的目录项

| 文件名   | I节点号 |
|-------|------|
| name1 | 1234 |

|           |
|-----------|
| ...       |
| owner = A |
| count = 2 |
| 5678      |

文件内容

"This is  
the text  
in the file"

I节点1234

块5678

将I节点1234中的count置为2。

# 硬链接

3) 用户A删除文件name1

/dirB中的目录项

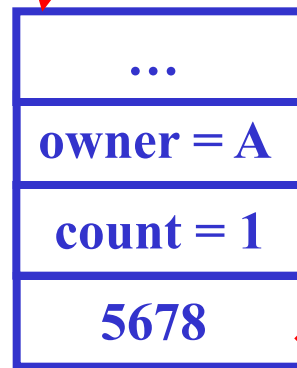
文件名 I节点号

|       |      |
|-------|------|
| name2 | 1234 |
|-------|------|

删除了文件name1后，并不会删除I节点1234，只是将count减1。

缺点：

由于文件主是A，若要记账收费，A即使把文件name1删除了，还要为B付费，直到B把name2删除为止。



I节点1234

文件内容

"This is  
the text  
in the file"

块5678





# 基于I节点的文件共享

---

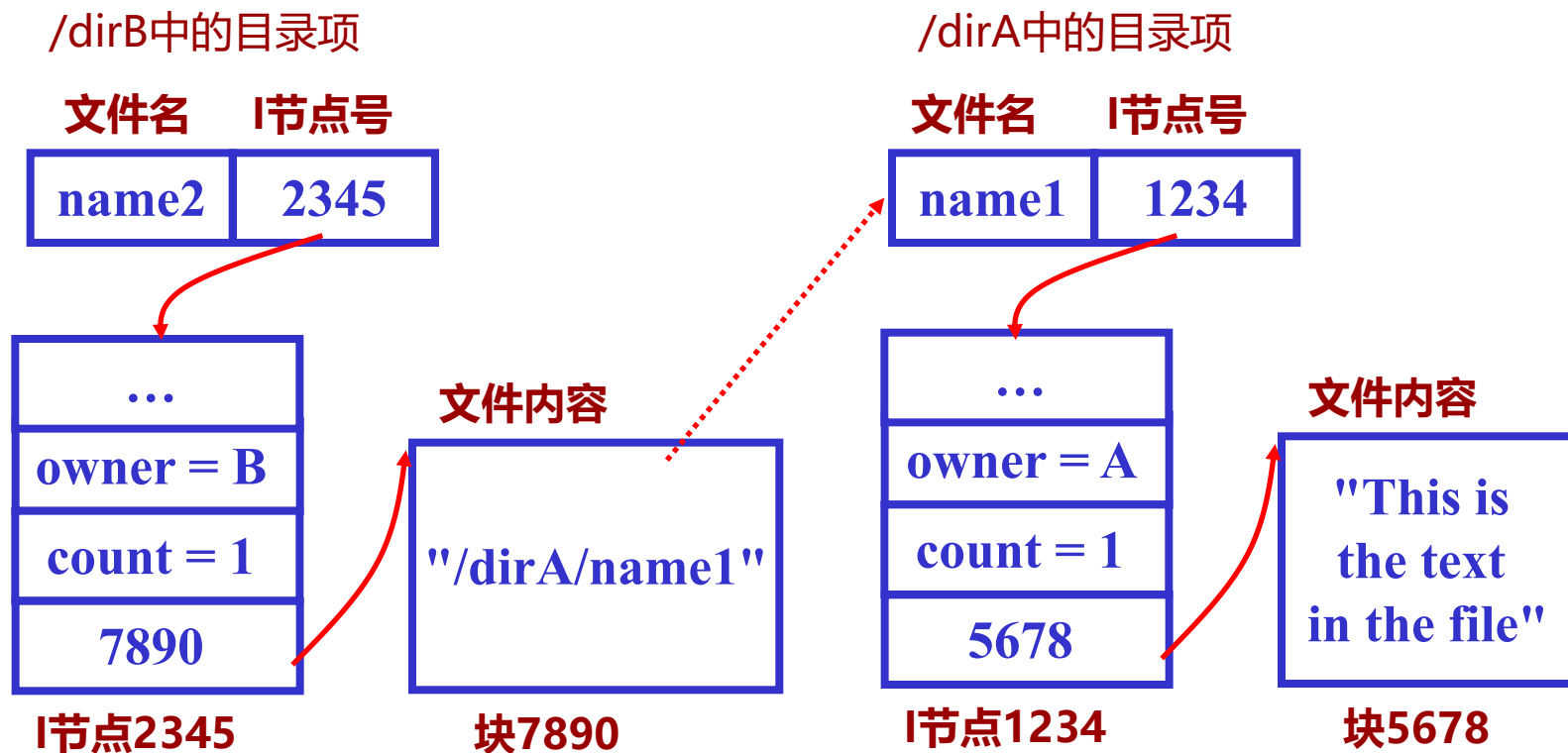
## (2) 符号链接

又称软链接

文件内容是所链接文件的路径名

# 符号链接

【例】文件name1和它的一个符号链接name2





# 符号链接

---

符号链接的缺点：

- ① 访问共享文件时，要根据路径名查目录，直到找到节点，开销大；
- ② 若执行

```
rm /dirA/name1
```

则符号链接依然存在，此时对/dirB/name2的引用将出错。

然而，若后来创建一新文件/dirA/name1，这个符号链接将指向该文件。

符号链接在Windows中叫快捷方式。



## 6.5 文件的使用、共享和保护

---

### 三、文件存取控制

#### 1. 存取控制矩阵

给出每个用户对每个文件的访问权限。

一维是所有用户，另一维是所有文件，  
对应的矩阵元素是用户对文件的访问权限。

例如，访问操作分为：

- ✓ 读操作 (r)
- ✓ 写操作 (w)
- ✓ 执行操作 (x)
- ✓ 不能执行任何操作 (-)

当用户和文件较多时，很庞大。



# 存取控制矩阵

| 用户    | 文件    |       |       |       |
|-------|-------|-------|-------|-------|
|       | File1 | File2 | File3 | ..... |
| User1 | rwX   | r     | rw    | ..... |
| User2 | x     | rw    | —     | ..... |
| User3 | r     | —     | r     | ..... |
| User4 | rwX   | r     | rw    | ..... |



# 文件存取控制

---

## 2. 存取控制表 (Access Control List, ACL)

每个文件一张ACL，将用户分类，规定每类用户的访问权限。

例如，Unix/Linux将用户分类为：

- ✓ 文件主 (owner)
- ✓ 文件主的同组用户 (group)
- ✓ 其他用户 (other)



# 存取控制表

| 用户           | 对File1的访问权限 |
|--------------|-------------|
| <b>owner</b> | <b>rwX</b>  |
| <b>group</b> | <b>rx</b>   |
| <b>other</b> | <b>-</b>    |

这样UNIX索引节点中可用9个二进制位

rwX rwX rwX

对各类用户设访问权限。

文件主可修改访问权限，例如

chmod 711 file1

chmod 755 file2



# 文件存取控制

---

## 3. 存取权限表 (Capability List, CL)

每个用户一张CL，规定对每个文件的访问权限。

| 文件    | 某用户的访问权限 |
|-------|----------|
| file1 | r        |
| file2 | rw       |
| file3 | rwX      |





# 文件存取控制

---

## 4. 口令

用户创建文件时，设置一个口令，放在文件目录中。

## 5. 密码

写入时加密，读出时解密。



## 6.6 小结

---

### 一、文件系统及其作用

- ✓ 什么是文件和文件系统？为什么引入文件和文件系统？
- ✓ 文件系统有何作用？用户观点？系统观点？

### 二、文件的逻辑结构和物理结构

掌握每种结构的特点，会一般的计算。例如读盘次数等。

### 三、文件目录

- ✓ 几个概念：文件控制块FCB；文件说明；目录项；文件目录
- ✓ 文件目录的实现方法：FCB都放在目录项中；目录项分解
- ✓ 目录检索与文件寻址
- ✓ 会一般的计算



## 6.6 小结

---

### 四、基于I节点的文件共享

- ✓ 硬链接
- ✓ 软链接

### 五、为什么要打开/关闭文件？ Open/close有何作用？



# 第7章 设备管理

---

## 7.1 设备管理概述

## 7.2 I/O控制方式

## 7.3 I/O缓冲

## 7.4 设备分配与设备处理

## 7.5 I/O管理中的几个重要思想

## 7.6 磁盘I/O

## 7.7 小结



# 7.1 设备管理概述

---

## 一、I/O设备的分类

### 1. 按从属关系分类

#### (1) 系统设备

指在操作系统生成时已经登记在系统中的标准设备。

如键盘、显示器、打印机等。

#### (2) 用户设备

指操作系统生成时未登记入系统的非标准设备。如绘图仪、扫描仪等。



# I/O 设备分类

---

## 2. 按传输速率分类

### (1) 低速设备

指传输速率为每秒钟几个字符至数百个字节的设备

如键盘、鼠标、语音输入等。

### (2) 中速设备

指传输速率为每秒钟数千个字节至数万个字节的设备

如针式打印机、激光打印机等。

### (3) 高速设备

指传输速率为数兆字节的设备，如磁带机、磁盘机、光盘机等。



# I/O 设备分类

---

## 3. 按使用特性分类

### (1) 存储设备

是计算机用来保存各种数据的设备，如磁盘、磁带等。

### (2) I/O设备

是向CPU传输数据或输出CPU加工处理数据的设备。 例如：键盘，CRT



# I/O 设备分类

---

## 4. 按共享属性分类

### (1) 独占设备

指在**一段时间内**只允许一个用户（进程）访问的设备。

也就是在某个用户（进程）对设备的一次使用过程（包含多次I/O操作）中，不允许其他用户（进程）使用该设备。一般是低速的I/O设备,如打印机等。

独占设备属于临界资源，多个并发进程必须互斥访问独占设备。





# 按设备共享属性分类

---

## (2) 共享设备

指在一段时间内允许多个进程同时访问的设备，

多个进程以交叉的方式来使用设备，其资源利用率高，如硬盘。

## (3) 虚拟设备

指通过虚拟技术将一台独占设备变换为若干台供多个用户（进程）共享的逻辑设备。一般可以利用假脱机（SPOOLing）技术实现虚拟设备。



# I/O 设备分类

---

## 5. 按数据传送的基本单位分类

### (1) 字符设备

以字节为单位传送数据，如键盘。

传输速度慢。

### (2) 块设备

以块（如512B）为单位传送数据，如磁盘。

传输速度快。

是**可寻址**的和**可随机访问**的设备



## 7.1 设备管理概述

---

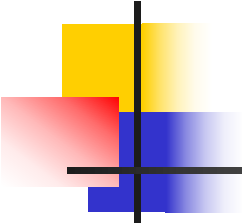
### 二、设备管理的2个重要目标：效率、通用性

#### (1) 效率

提高设备的利用率和I/O效率

充分利用各种技术（中断、DMA、通道、缓冲等）提高CPU与设备、设备与设备之间的并行工作能力，充分利用资源，提高资源利用率，提高I/O处理的效率

- ✓ 并行性
- ✓ 均衡性（使设备充分忙碌）



# 设备管理的2个重要目标

---

## (2) 通用性

为用户提供方便、统一的接口，希望能**用统一的方式处理所有设备**

屏蔽硬件细节（设备的物理细节，错误处理，不同I/O的差异性）

使用户摆脱繁琐的程序设计负担

- ✓ 方便性
- ✓ 接口的友好性
- ✓ **透明性**



## 7.1 设备管理概述

---

### 三、设备管理的功能

#### (1) 设备分配

按照设备类型和相应的分配算法决定将I/O设备分配给哪一个要求使用该设备的进程。凡未分配到所需设备的进程被放入一个等待队列。

#### (2) 设备处理

设备处理程序实现CPU和设备控制器之间的通信。即当CPU向设备控制器发出I/O指令时，设备处理程序应启动设备进行I/O操作，并能对设备发来的中断请求作出及时的响应和处理。

#### (3) 实现其他功能

包括对缓冲区的管理功能以及实现设备独立性。



## 7.2 I/O 控制方式

---

I/O控制方式：主机和I/O设备之间的数据传送方式

### **4种I/O控制方式:**

- (1) 程序直接控制方式（轮询）
- (2) 中断方式
- (3) DMA方式
- (4) 通道方式

发展的思路：解放CPU



## 7.2 I/O 控制方式

---

### 1. 程序直接控制方式（轮询）

由用户进程控制，不断测试设备状态

缺点：

- ✓ 忙等待
- ✓ CPU与I/O设备只能串行工作



## 7.2 I/O 控制方式

---

### 2. 中断方式

为了减少设备驱动程序不断询问设备控制器中状态寄存器的开销

当I/O操作结束后，由设备控制器主动通知设备驱动程序

不足：

- ✓ 数据传送是在中断处理时由CPU控制完成
- ✓ 每次传输的数据量小，比如1个字节或1个字。尽管，可以通过在I/O控制器中设置字符缓冲区而增大每次的数据传输量
- ✓ 可能造成数据由于CPU来不及取而丢失（当外设速度快时）





## 7.2 I/O 控制方式

---

### 3. DMA方式

- ✓ 在内存与I/O设备之间直接传送数据块
- ✓ CPU在开始时向设备发“传送一块”命令，数据传送由DMA控制器控制完成，每次1个数据块
- ✓ 传送结束时，由DMA控制器给CPU发送一个中断信号
- ✓ DMA的功能可以以独立的DMA部件在系统I/O总线上完成，或者整合到I/O部件中完成



# DMA 方式

---

## **DMA方式与中断的主要区别：**

- ✓ 中断方式是在数据缓冲寄存器满后，发中断请求，CPU进行中断处理  
DMA方式则是在所要求传送的数据块全部传送结束时要求CPU进行中断处理，大大减少了CPU进行中断处理的次数
- ✓ 中断方式的数据传送是由CPU控制完成的  
DMA方式则是在DMA控制器的控制下不经过CPU控制完成的



## 7.2 I/O 控制方式

---

### 4. 通道方式

通道：独立于CPU的专门负责数据输入/输出传输工作的处理机，对外部设备实现统一管理，代替CPU对输入/输出操作进行控制，从而使输入/输出可与CPU并行操作。

引入通道的目的：

为了使CPU从I/O操作中解脱出来，同时为了提高CPU与设备、设备与设备之间的并行工作能力

数据传送的方向、长度、内存地址等都由通道控制。



## 7.3 I/O 缓冲

---

### 一、什么是I/O缓冲？

**缓冲 (buffering) :**

为了缓解通信双方速度不匹配而引入的一个中间环节。

**I/O缓冲:**

在CPU和I/O设备之间设立缓冲区，用以暂存CPU与外设之间交换的数据，从而缓和CPU与外设速度不匹配所产生的矛盾。

其实，凡是数据到达和离去速度不匹配的地方均可采用缓冲技术。



## 7.3 I/O 缓冲

---

**引入I/O缓冲的目的:**

- (1) 改善CPU与I/O设备之间速度不匹配的矛盾
- (2) **减少对CPU的中断频率**，放宽对I/O中断响应时间的限制
- (3) **减少访问I/O设备（如磁盘）的次数**
- (4) 提高CPU与I/O设备之间的并行性



## 7.3 I/O 缓冲

---

### **I/O缓冲的实现方式:**

#### (1) 硬件缓冲

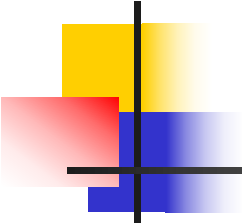
I/O设备或控制器内部设置的纯硬件缓冲区

#### (2) 软件缓冲

为I/O在内存开辟的缓冲区，由软件来管理

容量大，使用灵活

下面要讲的是OS采用的I/O缓冲技术。



## 7.3 I/O 缓冲

---

### 二、OS为什么要引入I/O缓冲?

**【例】**应用程序要从磁盘中读一块数据 (512B) 到自己的地址空间1000-1511(虚拟地址)。在执行I/O命令后阻塞, 等待数据变成可用的。

#### 存在的问题:

该进程的虚拟地址单元1000-1511必须锁定在内存中, 不能换出, 干扰了OS的交换决策;

否则有可能造成单进程死锁。即进程发出I/O命令后阻塞, 若在开始I/O操作之前被换出, 则I/O操作也会阻塞, 以等待该进程被换入。



## 7.3 I/O 缓冲

---

### 三、I/O缓冲的种类

#### 1. 单缓冲

当用户进程发出I/O请求时，操作系统在内存的系统空间为该操作分配一个缓冲区

输入：数据输入到系统缓冲区，传送完成时，用户进程将其复制到用户空间

输出：用户进程将数据复制到系统缓冲区，然后由OS输出。

如果连续输入/输出多个数据块呢？





# I/O 缓冲的种类

---

## 2. 双缓冲

由OS指定2个缓冲区

当用户进程从一个缓冲区取数据（或向一个缓冲区写数据）时，  
OS可以向另一个缓冲区输入数据（或输出另一个缓冲区的数据）。

## 3. 缓冲池

又称循环缓冲

多个缓冲区构成循环队列

类似于生产者/消费者问题



## 7.4 设备分配与设备处理

---

### 一、设备分配

当某进程向系统提出I/O请求时，设备分配程序按一定策略分配设备、控制器和通道，形成一条数据传输通路，以供主机和设备间信息交换。



# 设备分配

---

## 1. 与设备分配有关的数据结构

### ✓ 设备类表

系统中拥有1张设备类表

每类设备对应于表中一栏，包括内容有：

设备类、总台数、空闲台数、设备驱动程序入口和设备表起始地址等。

### ✓ 设备表

每一类设备都有各自的设备表，用来登记这类设备中每一台设备的状态，

包含的内容有：物理设备名、逻辑设备名、占有设备的进程号、已分配/未分配、好/坏等。



# 与设备分配有关的数据结构

---

采用通道结构的系统中设备分配采用的数据结构：

系统设备表、通道控制表、控制器控制表和设备控制表

- ✓ 系统建立1张系统设备表，记录配置在系统中的所有物理设备的情况。
- ✓ 每个通道、控制器、设备各设置一张表，记录各自的地址(标识符)、状态(忙/闲)、等待获得此部件的进程队列指针、及一次分配后相互链接的指针，以备分配和执行I/O时使用。

具体内容如下：



# 与设备分配有关的数据结构

---

- ✓ 设备控制块DCB (设备控制表DCT)

记录本设备的使用情况。主要内容：设备类型、设备标识符、设备状态、与此设备相连的COCT、重复执行的次数或时间、等待队列的队首和队尾指针

- ✓ 控制器控制块COCB (控制器控制表COCT)

- ✓ 通道控制块CHCB (通道控制表CHCT)

- ✓ 系统设备表SDT

整个系统一张表，记录系统中所有I/O设备的信息，表目包括：设备类型、设备标识符、设备驱动程序入口、DCT表指针等，是分配程序首先查找的数据结构。



# 设备分配

---

## 2. 设备分配策略

由于在多道程序系统中，进程数多于资源数，引起资源的竞争。  
因此，要有一套合理的分配策略。

### 考虑的因素：

- ✓ I/O设备的固有属性
- ✓ I/O设备使用场合的目标需求：
  - 设备分配的安全性
  - 与设备的无关性（即设备独立性）
  - 设备利用率
  - 公平性
  - 请求设备的进程优先级



# 设备分配策略

---

## (1) 独占设备的分配

所谓独占式共享使用设备是**以一次设备使用过程（包含多次I/O操作）为单位使用设备**

在申请设备时，如果设备空闲，就将其独占，不再允许其他进程申请使用，一直等到该设备被释放，才允许其他进程申请使用

考虑效率问题，并且避免由于不合理的分配策略造成死锁

**静态分配：**在进程运行前，完成设备分配；运行结束时，收回设备

缺点：设备利用率低

**动态分配：**在进程运行过程中，当用户提出设备请求时，进行分配，一旦停止使用立即收回

优点：设备利用率高；缺点：分配策略不好时，产生死锁



# 设备分配策略

---

## (2) 分时式共享设备的分配

所谓分时式共享就是以**一次I/O操作为单位分时使用设备，不同进程的I/O操作请求以排队方式分时地占用设备进行I/O**

由于同时有多个进程同时访问，且访问频繁，因此要考虑多个访问请求到达时服务的顺序，使平均服务时间越短越好

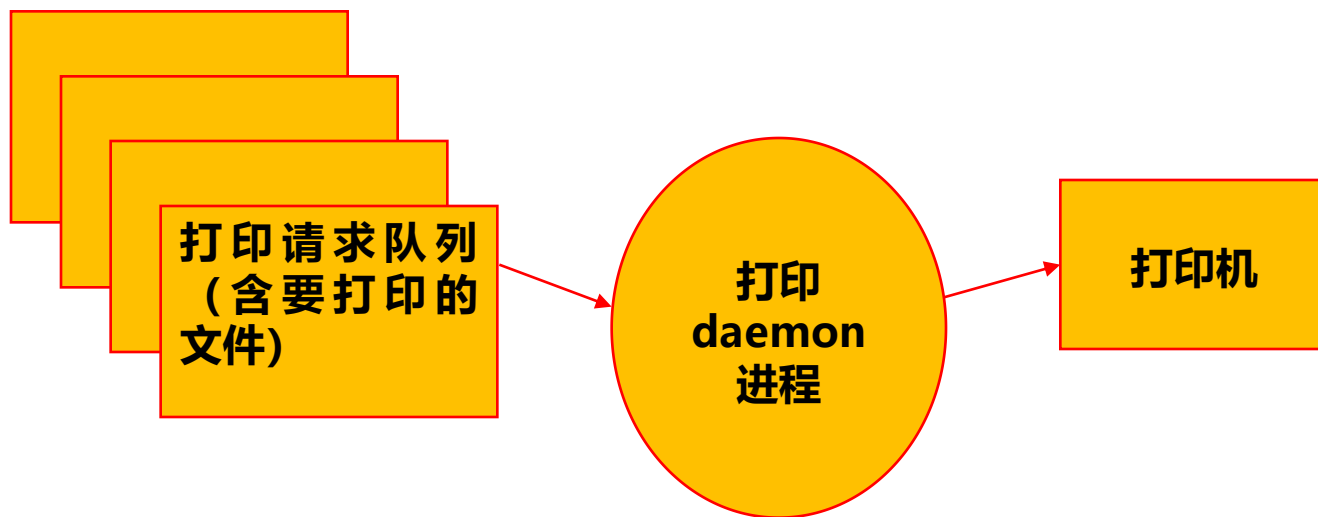


# 设备分配策略

## (3) 以SPOOLing方式使用外设

SPOOLing 技术是在批处理操作系统时代引入的，即假脱机输入/输出技术。

例如：所有输出数据已经写到文件中，并排到打印输出队列，打印进程申请占用打印机后，成批读出文件中的数据，并送打印机打印





# 设备分配策略

---

常用的I/O设备分配算法：

- ✓ 先来先服务
- ✓ 优先级高者优先

设备请求队列：

当多个进程对同一设备提出I/O请求时，系统响应后，为它们分别建立I/O请求包，按先来先服务或者优先级高者优先的原则组织成设备请求队列。

设备分配程序总是把设备首先分配给队首进程。具体分配是从设备类表或者系统设备表开始顺序查找相应的数据结构进行的。



## 7.4 设备分配与设备处理

---

### 二、设备处理

#### 1. 设备驱动程序

**每类设备对应1个设备驱动程序**，以控制I/O传输

任务：

主要负责接收和分析从设备分配转来的信息，把用户I/O请求转换为具体要求后，发送给设备控制器，启动设备执行。



# 设备驱动程序

---

## 设备驱动程序的处理过程:

- (1) 将抽象I/O请求转换为具体的操作参数
- (2) 检查I/O请求的合法性
- (3) 读出和检查设备的状态
- (4) 传送必要的参数, 预置设备的初始状态
- (5) 设置设备的工作方式 (在有通道的系统中, 构造通道程序)
- (6) 启动设备进行I/O操作
- (7) 响应来自设备的中断

## 2. I/O中断处理程序

处理来自设备或通道的中断

包括正常结束, 或异常结束



## 7.5 I/O 管理中的几个重要思想

---

### 一、设备独立性 (Device Independence)

设备独立性是I/O软件设计中的一个重要目标。

#### 设备独立性的含义:

用户在编写程序时，能独立于具体使用的物理设备，甚至不关心设备类型。

- ① 应用程序与给定设备类型的哪一台具体设备无关；
- ② 应用程序尽可能地与设备类型无关。



# 设备独立性

---

## 设备独立性的实现方法：

### (1) 引入逻辑设备

由OS管理一个逻辑设备映射表，记录逻辑设备对应的物理设备。

用户程序对I/O设备的请求不指定特定的设备，而采用逻辑设备名，程序执行时由OS完成逻辑设备到物理设备的映射

例如：用户申请使用设备时，只需要指定设备类型，而无须指定具体物理设备，系统根据当前的请求及设备分配情况，在相同类别设备中，选择一个空闲设备，并将其分配给一个申请进程。

### (2) 统一命名，统一接口

对不同的设备采取统一的操作方式

例如：把设备看作文件，所有设备和文件使用相同的方式（路径名定位）



## 7.5 I/O 管理中的几个重要思想

---

### 二、SPOOLing技术

#### **SPOOLing:**

Simultaneous Peripheral Operation On Line

即外围设备同时联机操作，又称假脱机操作

SPOOLing的概念最早出现在作业处理中，那时多道程序的概念还没有提出  
这个概念至今还有很大的意义



# SPooling 技术

---

## 1. 脱机输入/输出 (Offline I/O)

脱机I/O：I/O是脱离主机的。

目的：为了解决CPU和I/O设备的速度不匹配。

输入：由一台低档计算机（外围机）将作业输入到磁盘（磁带）；

CPU需要时，从磁盘读入内存。

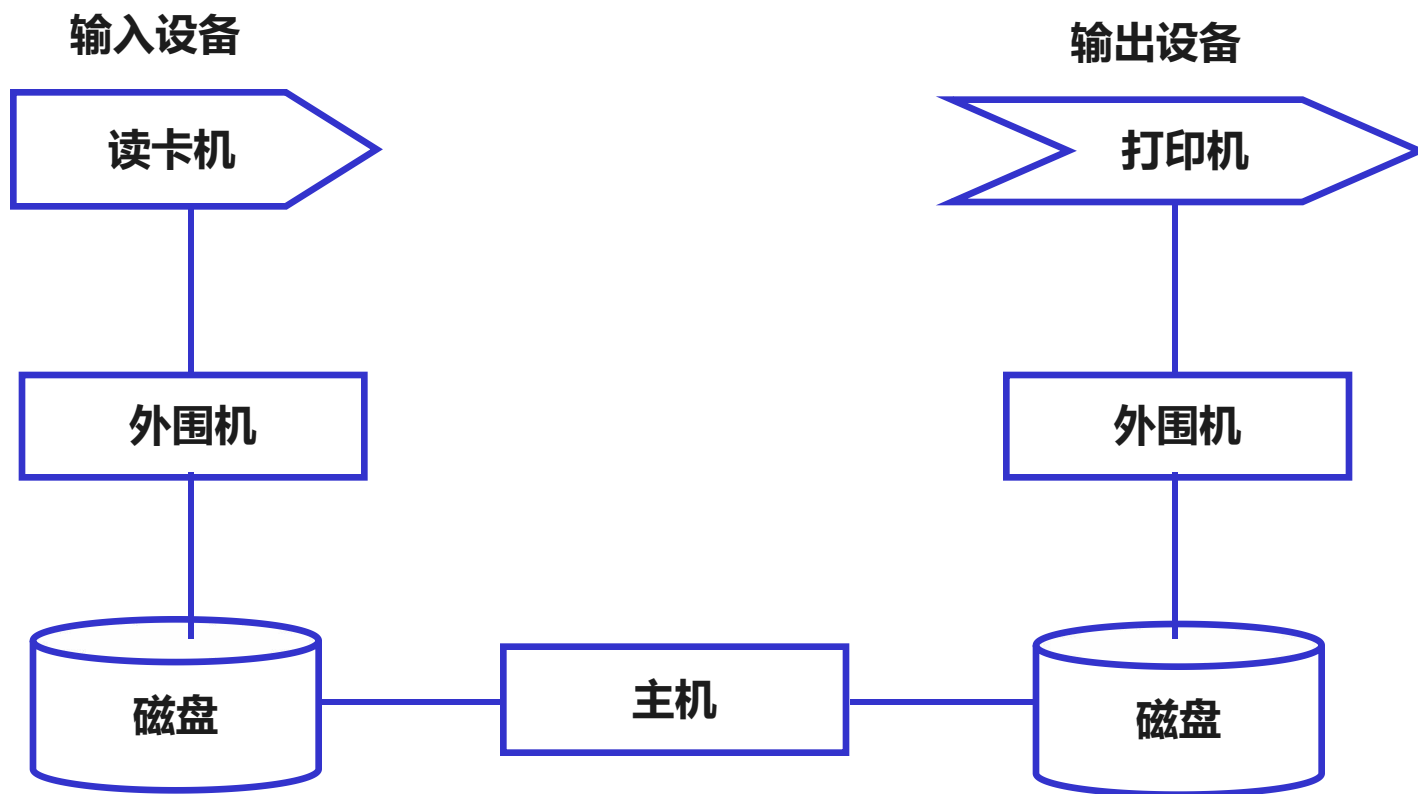
输出：CPU需要输出时，将数据从内存送到磁盘；

由一台外围机将磁盘中的数据输出。



# SPooling 技术

脱机I/O示意图:





# SPOOLing 技术

---

## 2. 什么是SPOOLing?

### 假脱机:

联机情况下, 即在主机的控制下, 模拟脱机I/O。

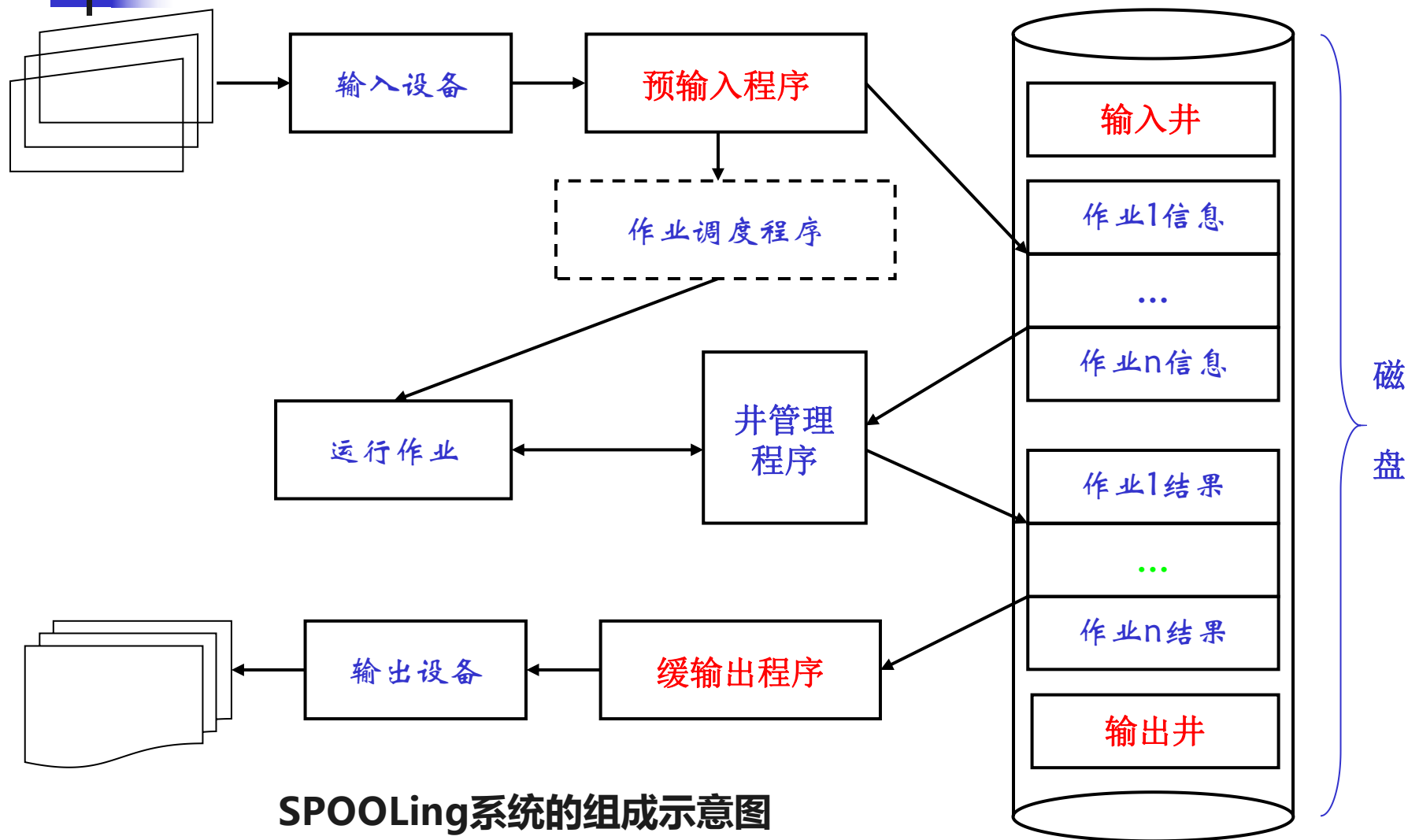
### 实现方法:

OS利用2个进程分别模拟脱机I/O时外围机的功能:

其中一个进程负责将输入设备的数据传送到磁盘;

另一个进程负责将数据从磁盘传送到输出设备。

# SPOOLing 技术





# SPOOLing 技术

---

## **SPOOLing系统的组成:**

### **(1) 输入井和输出井**

**在磁盘上开辟出来的两个专用的存储区域。**

“井”是用作缓冲的存储区域。输入井和输出井分别用于收容从输入设备输入的数据和用户程序的输出数据。

输入井和输出井可分别看作是对输入设备（如读卡机）和输出设备（如打印机）的虚拟或者模拟。



# SP00Ling系统的组成

---

## (2) 预输入进程和缓输出进程

**预输入进程模拟脱机输入时的外围控制机**，将用户要求输入的数据从输入设备通过输入缓冲区再送到输入井。当CPU需要输入数据时，直接从输入井读入内存。

**缓输出进程模拟脱机输出时的外围控制机**，把用户要求输出的数据，先从内存送到输出井，待输出设备空闲时，再将输出井中的数据经过输出缓冲区送到输出设备上。



# SPOOLing 技术

---

## 3. SPOOLing技术的基本思想

### 是一种虚拟技术

是OS协调并发I/O（主要用于输出）的一种技术

用来**把一台独占设备改造成为可共享的虚拟设备**，使得每个进程都以为是独占一台设备



# SPOOLing 技术

---

**SPOOLing技术的应用：**

**打印机的共享：**

实际上系统并没有很多打印机，**只不过是磁盘的一个存储区**

**实现方法：**

- 1) 创建一个守护进程（daemon）、一个打印目录spooling;
- 2) 某个进程要打印文件时，首先生成要打印的文件，将文件放入spooling目录下，形成一个请求打印队列；
- 3) 统一由daemon负责在打印机空闲时，按队列的先后次序打印spooling目录里的文件。



# SPOOLing 技术

---

SPOOLing技术现在仍被广泛使用。

- ✓ 网络文件传送

先把文件送到网络spooling目录，由网络守护进程把它取出并传递到目标地址

- ✓ Internet电子邮件系统

在Internet上发Email时，电子邮件发送程序send先将待发信件存入spooling  
电子邮件目录下，供以后传输。





## 7.5 I/O 管理中的几个重要思想

---

### 三、I/O 软件的多层模型

I/O 软件按分层的思想构成

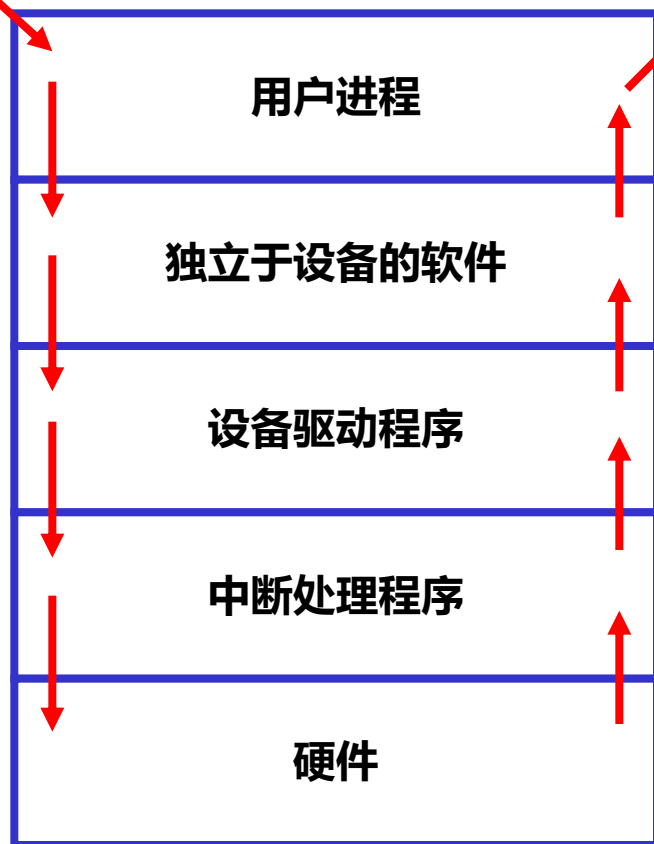
较低层软件要使较高层软件独立于硬件的特性

较高层软件则要向用户提供一个友好、清晰、简单、功能更强的接口

# I/O 软件的多层模型

I/O请求

I/O应答



进行I/O调用

提供与下层的统一接口：命名、  
分配、缓冲、保护

与设备有关的代码：向设备  
控制器发命令



# I/O 软件的多层模型

---

## 1. I/O软件各层的功能

### (1) 用户进程层

执行I/O系统调用，对I/O数据进行格式化

### (2) 独立于设备的软件

实现设备的命名、设备的分配、数据的缓冲、设备的保护，提供与下层的统一接口

### (3) 设备驱动程序

与设备有关的代码。向设备控制器发命令，检查设备的执行状态

### (4) 中断处理程序

负责I/O完成时，唤醒设备驱动程序进程，进行中断处理

### (5) 硬件层（设备控制器，设备）

实现物理I/O的操作



# I/O 软件的多层模型

---

## 2. 中断处理程序

- ✓ 每个进程在启动一个I/O操作后阻塞
- ✓ 直到I/O操作完成并产生一个中断
- ✓ 由操作系统接管CPU后唤醒该进程为止



# I/O 软件的多层模型

---

## 3. 设备驱动程序

- ✓ 与设备密切相关的代码放在设备驱动程序中，每个设备驱动程序处理一种设备类型
- ✓ 每个设备控制器都设有一个或多个设备寄存器，用来存放向设备发送的命令和参数。设备驱动程序负责发出这些命令，并监督它们正确执行
- ✓ 一般地，设备驱动程序的任务是接收来自与设备无关的上层软件的抽象请求，并执行这个请求
- ✓ 在设备驱动程序的进程发出一条或多条命令后，系统有两种处理方式，多数情况下，执行设备驱动程序的进程必须等待命令完成，这样，在命令开始执行后，它阻塞自己，直到中断处理时将它解除阻塞为止。而在其它情况下，命令执行不必延迟就很快完成



# I/O 软件的多层模型

---

## 4. 独立于设备的软件

- (1) 设备驱动程序与独立于设备的软件之间的确切界限是依赖于具体系统的
- (2) 独立于设备的软件的基本任务是实现所有设备都需要的功能，并且向用户级软件提供一个统一的接口
- (3) 如何给文件和设备这样的对象命名是操作系统要考虑的一个重要问题。  
独立于设备的软件负责把设备的符号名映射到正确的设备驱动程序上
- (4) 设备保护。防止无权存取设备的用户存取设备



# 设备独立软件

---

(5) 不同的磁盘可以采用不同的扇区尺寸。向较高层软件隐藏这一事实并提供大小统一的块尺寸，这正是独立于设备的软件的一个任务。它可将若干扇区合成一个逻辑块。这样，较高层的软件只与抽象设备打交道，独立于物理扇区的尺寸而使用等长的逻辑块

(6) 缓冲

(7) 设备分配

(8) 出错处理



# I/O 软件的多层模型

---

## 5. 用户空间的I/O软件

尽管大部分I / O软件都包含在操作系统中，但仍有一小部分是由与用户程序连接在一起的库过程、甚至完全由运行于核心外的程序构成。系统调用（包括I/O系统调用）通常由库过程进入

这些过程所做的工作只是将系统调用时所用的参数放在合适的位置，由其它的I/O过程实际实现真正的操作





# I/O 软件的多层模型

---

## 说明:

各层之间的接口并不是死的，分层并不一定是严格的

上层中的某些功能可能放在下层中完成，如中断时的驱动以及驱动层中的某些与设备无关的处理等。



# 一个典型的读I/O设备的过程

①发出系统调用Read

用户进程

⑩当该用户进程被调度运行时，完成系统调用，继续执行

②核心的系统调用代码检查参数的正确性。若数据在缓冲区可得到，则转⑨

③向相应的设备驱动程序发送请求

独立于设备的软件

⑨将数据传送到请求进程的地址空间，并将该进程从阻塞队列移入就绪队列

④设备驱动程序分配一个内核缓冲区，向设备控制器发送命令，阻塞

设备驱动程序

⑧设备驱动程序确定是哪个I/O完成，确定请求的状态，通知独立于设备的软件“I/O请求已完成”

中断处理程序

⑦中断服务程序将数据存入设备驱动程序的缓冲区，唤醒阻塞的设备驱动程序，中断返回

⑤控制器操作硬件设备执行数据传输

设备控制器

⑥当I/O完成时，产生中断。假定传输由DMA控制器管理。当然，也许是由设备驱动程序轮询



## 7.6 磁 盘 I/O

---

### 一、影响磁盘I/O操作性能的几个因素

#### 1. 磁盘的结构

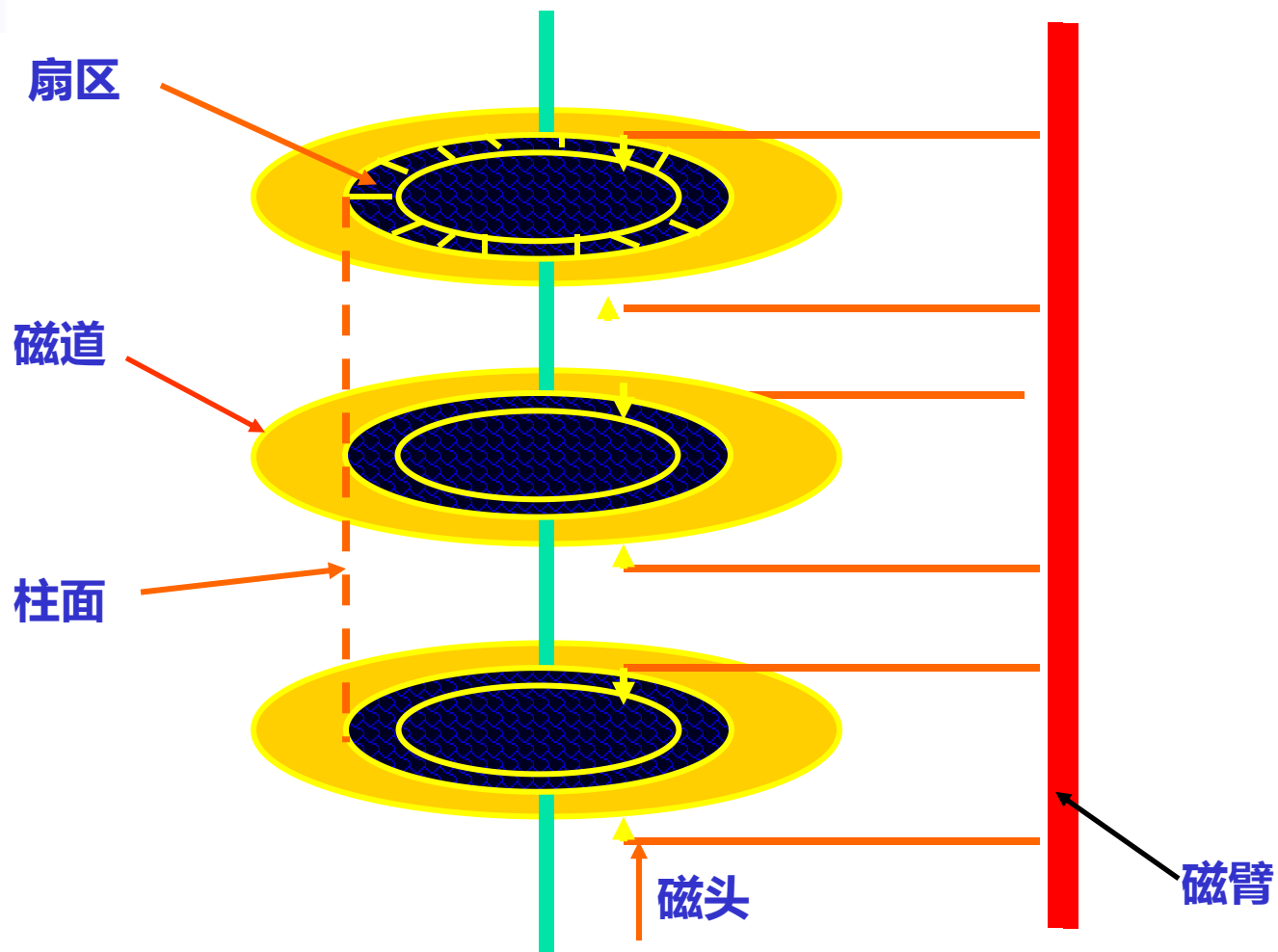
磁盘一般分为固定头磁盘和移动头磁盘两大类：

**固定头磁盘：**每个盘面的每条磁道都有一个读/写磁头。固定头磁盘各磁头可并行读写，但成本较高，主要用在大型机中。

**移动头磁盘：**每个盘面只有一个读/写磁头。每次读写须先移动磁头到目标磁道上，这称为**寻道 (seek) 操作**。磁头装在同一个架子（磁臂）上，在磁盘一致的方向上径向移动，不能单独移动。

个人计算机中的硬盘（Winchester盘）和软盘一般都是移动头磁盘。软盘由单盘片组成，硬盘则是个盘片组（因为单一盘片的容量越来越大，故硬盘所含的盘片数有逐渐减少的趋势）。盘片安装在一个高速旋转的枢轴上。读写头安装在移动臂上，移动臂可沿磁盘半径方向移动。

# 磁盘的结构



移动头硬盘驱动器结构示意图



# 磁盘的结构

---

## 磁盘的结构:

包含1个或多个盘片

每片分2面

每面分为若干磁道（同心环）

每个磁道分为若干扇区（Sector）

数据存储在磁道上

所有盘片的每面上相同位置的磁道称为一个柱面(cylinder)

**每个盘块的地址由柱面号、磁头号 and 扇区号表示**



## 7.6 磁 盘 I/O

---

### 2. 磁盘访问时间

#### (1) 寻道时间 (Seek Time) $T_s$

**把磁臂（磁头）径向移动到指定磁道或柱面上所经历的时间**，包含启动磁臂和磁头移动n条磁道或柱面所花费的时间。

#### (2) 旋转延迟 (Rotational Delay) 时间 $T_r$

**指定扇区旋转到达磁头下面所经历的时间**。与盘面的旋转速度有关。

一旦选择好磁道，磁盘控制器开始等待，直到相应扇区旋转 to 磁头处。  
然后，开始执行读/写操作。

#### (3) 传输 (Transfer) 时间 $T_t$

**把数据从磁盘读出或向磁盘写入数据所经历的时间**。

与旋转速度和一次读写的数据量有关。



# 磁盘访问时间

---

磁盘访问时间 $T_a$ 可表示为

$$T_a = T_s + T_r + T_t$$

$$\text{平均值} = T_s + 1/(2r) + b/(rN)$$

其中,  $r$ : 磁盘的旋转速度 (转/秒)

$N$ : 一个磁道中的 字节数

$b$ : 要读/写的字节数



## 7.6 磁 盘 I/O

---

### 二、磁盘调度算法

**目标：使磁盘的平均寻道时间最短。**

当磁臂为一个请求寻道时，其他进程会产生其他磁盘请求

磁盘驱动程序维护一个请求队列，按柱面号索引，每个柱面的待处理请求组成一个链表。

#### 1. 先来先服务 (First Come First Served, FCFS)

按请求访问磁盘的先后次序进行调度





# 磁盘调度算法

---

## 2. 最短寻道时间优先 (Shortest Seek Time First, SSTF)

选择处理与当前磁头距离最近的磁道请求，以减少寻道时间

但不能保证平均寻道时间最短

有可能出现“饿死” (Starvation)



# 磁盘调度算法

---

## 3. SCAN (扫描) 算法

要求磁头臂仅沿一个方向（假设向磁道号增加的方向）移动，并在途中满足所有未完成的请求，直到到达该方向的最后一个磁道或该方向上没有别的请求为止。

然后转向，沿相反方向扫描，同样按顺序完成所有请求。

如此不断反复。

又称**电梯调度算法**，其移动规律类似于电梯的运行。

克服了最短寻道时间优先的缺点，既考虑了距离，同时又考虑了方向



# 磁盘调度算法

---

## 4. C-SCAN (Circular SCAN, 循环扫描)

将扫描限定在一个方向

当沿某个方向访问到最后一个磁道时，磁头臂返回到磁盘的另一端，再次开始扫描。



# 磁盘调度算法

---

**【例】** 设请求访问的磁道顺序为

55, 58, 39, 18, 90, 160, 150, 38, 184

假定从磁道100处开始。

(1) FCFS

**下一个访问  
的磁道**

55 58 39 18 90 160 150 38 184

**移动距离  
(磁道数)**

45 3 19 21 72 70 10 112 146

**平均寻道  
长度**

55.3



# 磁盘调度算法

---

**【例】** 设请求访问的磁道顺序为

55, 58, 39, 18, 90, 160, 150, 38, 184

假定从磁道100处开始。

(2) SSTF

**下一个访问  
的磁道**

90 58 55 30 38 18 150 160 184

**移动距离  
(磁道数)**

10 32 3 16 1 20 132 10 24

**平均寻道  
长度**

27.5



# 磁盘调度算法

---

**【例】** 设请求访问的磁道顺序为

55, 58, 39, 18, 90, 160, 150, 38, 184

假定从磁道100处开始。

(3) SCAN (假定开始时向磁道号增加的方向)

**下一个访问  
的磁道**

150 160 184 90 58 55 39 38 18

**移动距离  
(磁道数)**

50 10 24 94 32 3 16 1 20

**平均寻道  
长度**

27.8



# 磁盘调度算法

---

**【例】** 设请求访问的磁道顺序为

55, 58, 39, 18, 90, 160, 150, 38, 184

假定从磁道100处开始。

(4) C-SCAN(向磁道号增加的方向)

**下一个访问  
的磁道**

150 160 184 18 38 39 55 58 90

**移动距离  
(磁道数)**

50 10 24 166 20 1 16 3 32

**平均寻道  
长度**

27.5



# 磁盘调度算法

---

## SSTF、SCAN、C-SCAN存在的问题:

**磁臂粘着 (Arm Stickiness) 现象:** 一个或多个进程反复请求某个磁道I/O, 从而垄断了整个磁盘, 导致“饥饿”。

## 5. N-Step-SCAN

将磁盘请求队列分成若干长度为N的子队列

每一次用SCAN处理一个子队列

在处理某个子队列时, 新请求必须加到其他队列中

队列间用FCFS

若N较大, 接近于SCAN;

当 $N = 1$ , 就是FCFS





# 磁盘调度算法

---

## 6. FSCAN

使用2个子队列

当扫描开始时，所有请求都在一个子队列中，另一个子队列为空

在扫描过程中，所有新到的请求加入另一个队列中

使得新请求的服务延迟到老请求处理完之后。



## 7.6 磁 盘 I/O

---

### 三、磁盘I/O性能的改善（提高磁盘I/O性能的方法）

#### 1. 磁盘高速缓存（Disk Cache）

逻辑上属于磁盘，物理上驻留内存

**在内存中为磁盘设置一个缓冲区**，包含磁盘块的副本

当请求磁盘读时，先查看所需的块是否在高速缓存中。如果在，则可直接进行读操作。否则，首先要将块读到高速缓存，再拷贝到所需的地方

如果高速缓存已满，则需要进行淘汰

置换算法：常用LRU、NRU等



# 磁盘I/O性能的改善

---

## 2. 合理分配磁盘空间

分配盘块时，把有可能顺序访问的块放在一起，最好在同一个柱面上，从而减少磁臂的移动次数

## 3. 提前读

读当前块时，提前将下一盘块读入高速缓存

## 4. 延迟写

数据不立即写回磁盘

周期性地成簇写回



## 7.6 磁 盘 I/O

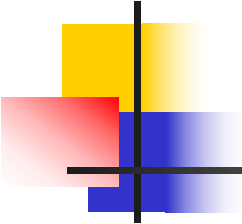
---

### 四、RAID

RAID:

Redundant Array of Independent Disks, 独立磁盘冗余阵列

- ✓ RAID是一组物理磁盘驱动器，OS将其看作单个逻辑驱动器
- ✓ 数据分布在物理驱动器阵列中
- ✓ 一种容错技术，通过增加冗余来提高可靠性
- ✓ 若访问的数据块分布在多个磁盘上，I/O请求可以并行



## 7.7 小结

---

### 一、设备分类

- ✓ 字符设备与块设备
- ✓ 独占设备与共享设备

### 二、什么是设备独立性？为什么引入？如何实现？

### 三、设备驱动程序

### 四、I/O软件的分层结构以及每层的功能

### 五、SPOOLing技术的思想、应用

### 六、磁盘I/O

磁盘调度算法；提高磁盘I/O性能的几种方法



# 第8章 Linux文件系统

---

**8.1 Linux文件系统的特点**

**8.2 ext2文件系统**

**8.3 Linux虚拟文件系统VFS**

**8.4 Linux的设备管理**



# 8.1 Linux文件系统的 特点

---

## 1. 字节流文件

Linux不关心文件的结构

ASCII文件与二进制文件无本质区别

## 2. 目录当作文件



## 8.1 Linux文件系统的 特点

---

### 3. 文件描述符 (file descriptor)

一个整数，用于标识文件

open/creat时由系统分配，close时释放

#### 3个固定的标准文件描述符：

0：标准输入

1：标准输出

2：标准错误输出

第1个打开的文件描述符是3





# 8.1 Linux文件系统的特 点

---

## 4. 文件的种类

(1) 正规文件 (regular file) : 普通文件

(2) 目录文件

(3) 特殊文件

设备文件: 字符特殊文件, 块特殊文件

(4) 套接字 (socket)

只是抽象成文件, 不是真正的文件

(5) 管道

(6) 符号链接和硬链接: 用ln命令可创建



## 8.1 Linux文件系统的特 点

---

### 5. 支持多种文件系统

- ✓ 支持ext, FAT, ext2, ext3, MINIX, MS DOS等

Linux的第1个文件系统是MINIX

1992, 第1个专门为Linux设计的文件系统ext

ext: Extended File System

ext → ext2 → ext3

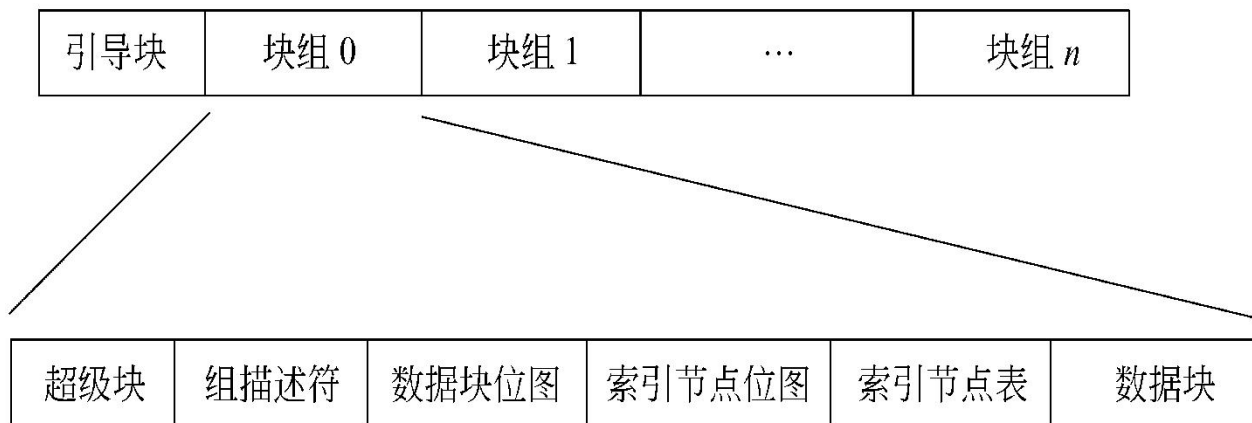
ext2是Linux的主要文件系统

- ✓ 引入虚拟文件系统VFS (Virtual File System)

VFS为用户程序提供了一个统一、抽象、虚拟的文件系统接口, 该接口主要由一组标准的、抽象的有关文件操作的系统调用构成

## 8.2 ext2 文件系统

### 一、ext2文件系统的物理结构



- ✓ **引导块 (boot block)** : 装有启动OS的引导代码。当有多个文件系统时, 只有1个有引导代码
- ✓ **块组 (block group)** : 由1个或多个连续的柱面组成



## 8.2 ext2 文件系统

---

✓ **超级块 (super block)** : 每个块组一个副本

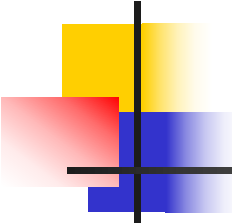
包含文件系统本身的基本信息:

数据块的大小

每组数据块的个数

首个I节点的号, 即根目录的入口

通常只有块组0的超级块才被读入内存, 其他块组的超级块用于备份。在系统运行期间, 超级块被复制到内存缓冲区中, 形成了一个 `ext2_super_block` 结构。



## 8.2 ext2 文件系统

---

### ✓ 块组描述符

每个块组都有一个相应的组描述符来描述它，所有的组描述符形成一个组描述符表，并在使用时被调入块高速缓存。

一个文件系统的所有数据块组描述符组成一个表，每一个块组在超级块后都包含一个数据块组描述符表的副本，以防遭到破坏。

### ✓ 位图

ext2中每个块组有两个位图块，一个用于表示数据块的使用情况，叫数据块位图；另一个用于表示索引节点的使用情况，叫索引节点位图。

位图中的每一位表示该组中一个数据块或一个索引块的使用情况，0表示空闲，1表示已分配。



## 8.2 ext2 文件系统

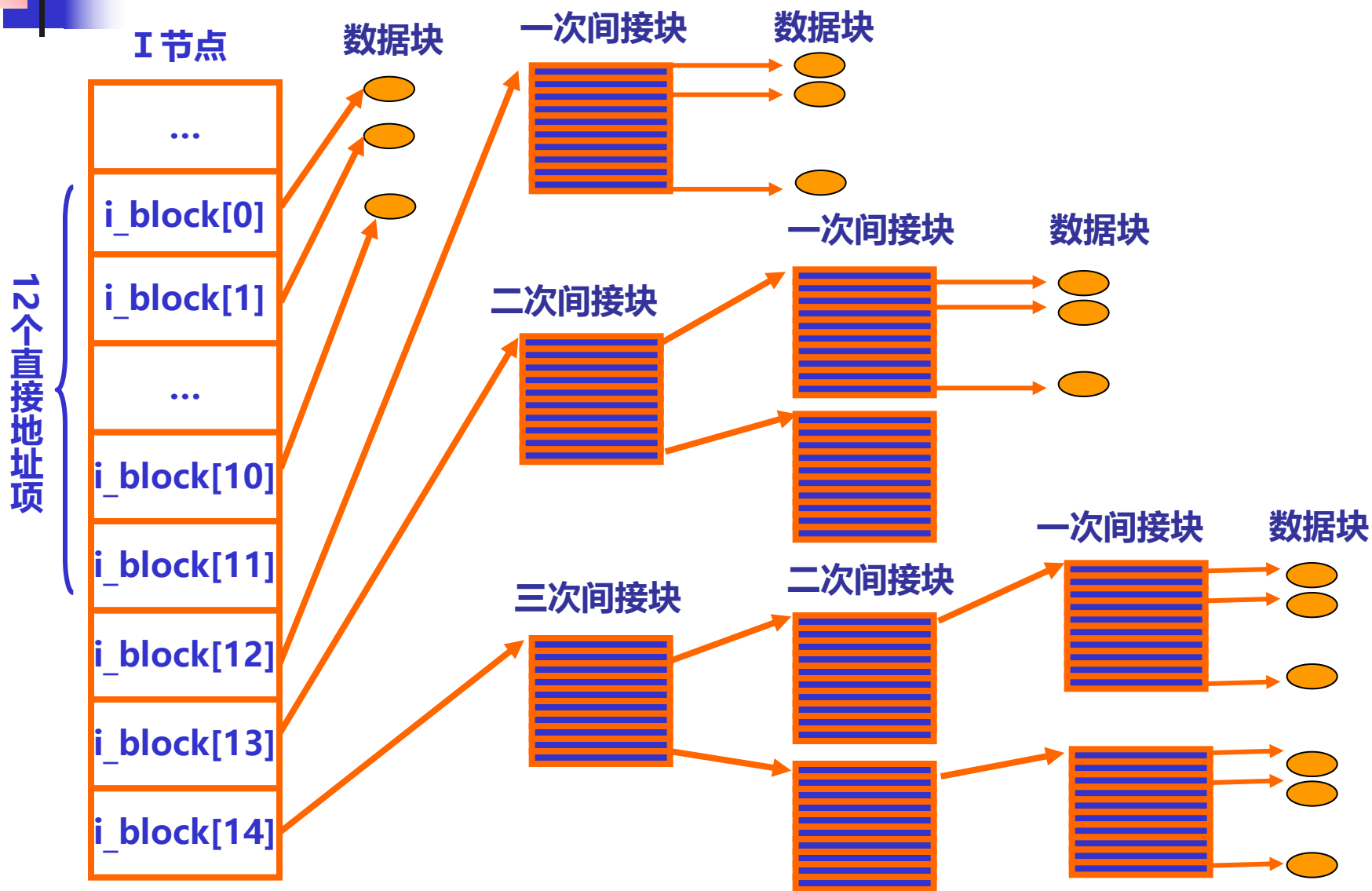
---

### 二、几个重要的数据结构

#### 1. 磁盘索引节点ext2\_inode

- ✓ 文件类型：普通文件、目录文件、字符特殊文件、块特殊文件
- ✓ 访问权限
- ✓ 文件长度（字节数）
- ✓ 用户id、组id
- ✓ 与该节点链接的文件数
- ✓ 文件的建立时间、最近访问/修改时间
- ✓ 文件的物理地址（数据块指针）i\_block[0]~i\_block[14]

# 磁盘I节点ext2\_inode





# 磁盘I节点ext2\_inode

---

## 说明:

设每个盘块1KB, 块号用4B表示

用i\_block[0]~i\_block[11]: 文件最大长度 = 12KB

用i\_block[12]:  $1K/4 = 256$ , 文件最大长度 = 256KB

用i\_block[13]: 文件最大长度 =  $256 * 256 = 64MB$

用i\_block[14]: 文件最大长度 =  $256 * 256 * 256 = 16GB$





# 磁盘I节点ext2\_inode

---

如何根据文件的逻辑地址偏移量得到物理地址？

字节偏移量/每块字节数

商：逻辑块号

余数：块内偏移量

if 逻辑块号 $i < 12$ ：物理块号为 $i\_block[i]$

$12 \leq i < 268$ ：需要查一次间接块

等等。



## 8.2 ext2 文件系统

---

### 2. 目录项

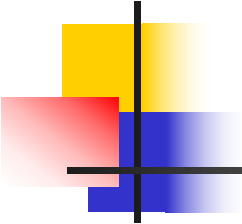
- ✓ 文件名
- ✓ 文件名长度
- ✓ I节点号

### 3. 内存索引节点inode

文件被打开后，在内存中建立相应的I节点，利于访问。

除了磁盘I节点的信息外，还包括：

引用计数：记录访问该I节点的进程数



## 8.2 ext2 文件系统

---

### 4. 用户打开文件表和系统打开文件表

#### (1) 用户打开文件表

又称为进程打开文件表，或**文件描述符表**

由task\_struct的字段files所指向

- ✓ 记录进程当前打开的文件
- ✓ 其中的fd字段是一个指向file对象（结构）的指针数组，数组的索引就是文件描述符。file对象构成的链表就是系统打开文件表



## 8.2 ext2 文件系统

---

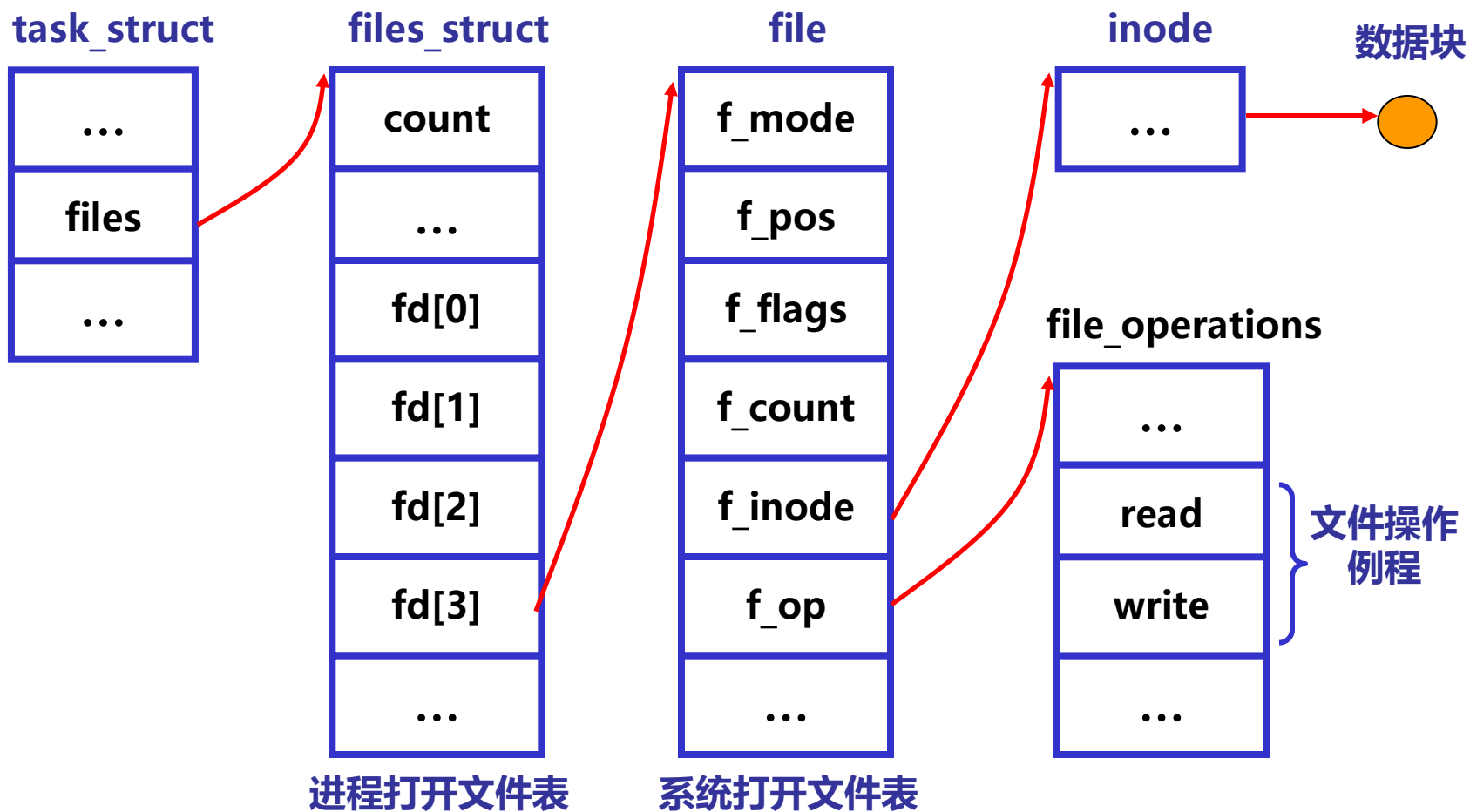
### (2) 系统打开文件表

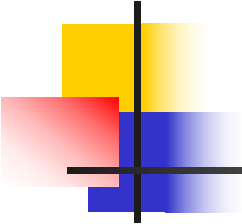
由若干file结构组成

每个file结构包含下列字段：

- ✓ f\_mode: 文件打开的方式
- ✓ f\_pos: 文件当前的读写位置
- ✓ f\_flags: 文件属性（有许多标志位）
- ✓ f\_count: 引用计数
- ✓ f\_inode: 指向VFS中该文件的内存I节点
- ✓ f\_op: 指向file\_operations结构，该结构包含文件操作的各种例程

## 8.2 ext2 文件系统





## 8.2 ext2 文件 系 统

---

### 说明:

- ✓ 当父子进程共享同一个文件时，它们使用同一个file结构；
- ✓ 以上介绍的内存I节点以及系统打开文件表、用户打开文件表，实际上是VFS提供的。



## 8.3 Linux 虚拟文件系统VFS

---

### 一、VFS的原理

**VFS建立在具体文件系统之上，为用户程序提供一个统一、抽象、虚拟的文件系统接口。**

虚拟文件系统也叫虚拟文件系统转换（Virtual File system Switch，VFS）。

- ✓ VFS的各种数据结构都是随时建立或删除的，在磁盘上并不永久保存，只能存放在内存之中。
- ✓ VFS是Linux核心的一部分，其他内核子系统与VFS打交道，VFS又管理其他具体的文件系统。所以VFS是文件系统和Linux内核的接口，VFS以统一数据结构管理各种具体的文件系统，接受用户层对文件系统的各种操作。



## 8.3 Linux 虚 拟 文 件 系 统 VFS

---

### 二、VFS的通用文件模型

VFS的文件模型由下列对象组成：

- 1) 超级块super\_block：存放已挂装文件系统的有关信息
- 2) 索引节点inode：一个具体文件的信息
- 3) 文件file：打开文件与进程之间进行交互的有关信息
- 4) 目录项dentry





## 8.3 Linux 虚拟文件系统 VFS

---

### **VFS与具体文件系统的接口：**

#### 1) super\_operations

超级块操作，实现文件系统的挂装、卸载等

用来将VFS对超级块的操作转化为具体文件系统处理这些操作的函数 内核为每一个挂装的文件系统分配一个超级块

VFS超级块super\_block包括的内容有：

- ✓ s\_fs\_info：指向具体文件系统的超级块信息
- ✓ s\_op：指向super\_operations



## 8.3 Linux 虚 拟 文 件 系 统 VFS

---

### 2) file\_operations

用来将VFS对file结构的操作转化为具体文件系统处理这些操作的函数  
file有一个字段f\_op, 指向file\_operations

### 3) inode\_operations

用来将VFS对索引节点的操作转化为具体文件系统处理相应操作的函数



## 8.3 Linux 虚 拟 文 件 系 统 VFS

---

### 三、对Linux文件系统的整体理解

从资源管理的角度来看，

每个文件要在磁盘上占用哪些资源？

当文件被打开后，在内存要占用哪些资源（数据结构）？



# 对Linux文件系统的整体理解

---

## **每个文件要在磁盘上占用3种资源：**

- 1) 1个目录项：记录文件名和I节点号
- 2) 1个磁盘I节点：记录除了文件名之外的文件属性（包括地址）
- 3) 若干盘块：存放文件内容

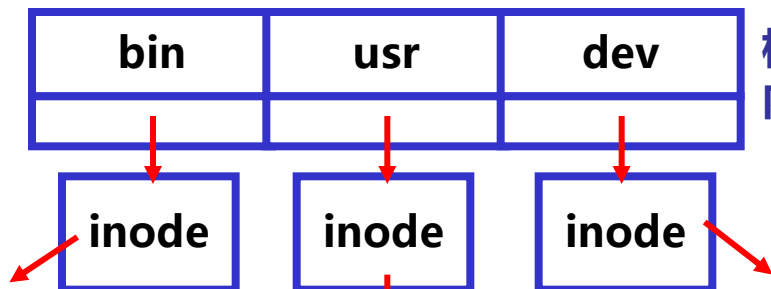
## **文件被打开后，在内存增加3种资源：**

- 1) 1个内存I节点：文件说明信息
- 2) 系统打开文件表的1项：1个file结构
- 3) 用户打开文件表中的1项

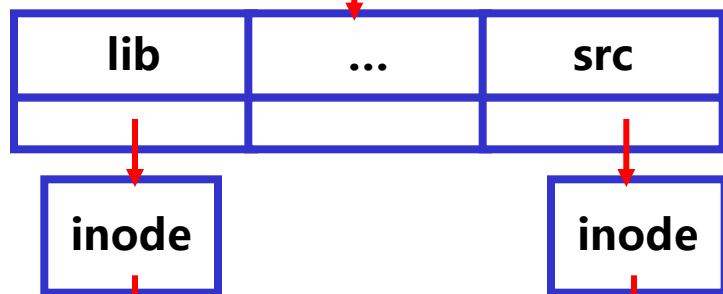
# 一个文件系统的结构例

## 根目录

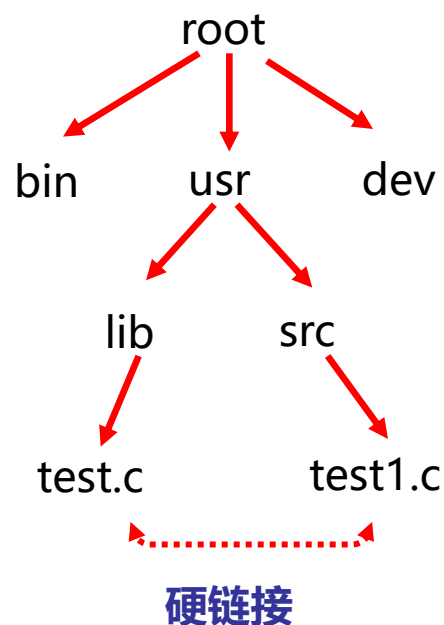
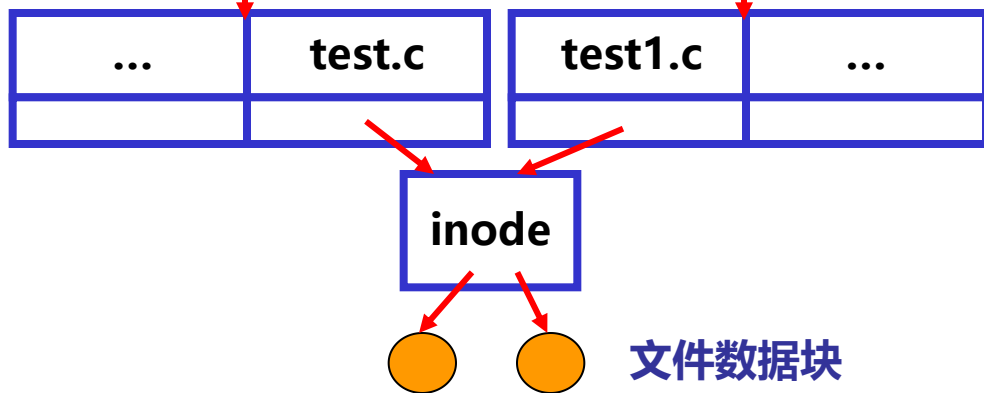
根目录文件的内容



目录文件/usr的内容



目录文件/usr/src的内容





## 8.3 Linux 虚拟文件系统VFS

---

### 四、查找文件

根据路径名得到磁盘I节点

【例】查找文件/usr/src/test1.c。文件系统结构见前例。

- 1) 按照超级块中根目录的I节点号找到根目录的数据块;
- 2) 在根目录的数据块中找到/usr对应的目录项, 然后  
找到/usr的I节点→/usr文件的数据块地址→/usr/src对应的目录项  
→/usr/src的I节点→/usr/src文件的数据块地址  
→/usr/src/test1.c对应的目录项→/usr/src/test1.c的I节点;
- 3) 得到I节点后, 就可以获取文件数据块地址, 从而访问文件。



## 8.3 Linux 虚拟文件系统VFS

---

### 五、文件系统调用

#### 1) open

```
fd = open(char *filename, int oflag [, int mode])
```

filename: 文件名

oflag: 打开方式

mode: 给出文件访问权限, 文件创建时使用

功能: 根据oflag的值打开文件, 也可以创建文件。

成功则返回文件描述符, 否则返回-1。

#### 2) close

```
close(int fd); //关闭文件
```



# 文件系统调用

---

## 3) read

```
n = read(int fd, char* buf, int nbytes);
```

功能：读fd所指文件，从当前位置读nbytes个字节放入buf，返回实际读出的字节数。

## 4) write

```
n = write(int fd, char* buf, int nbytes);
```

功能：写fd所指文件，将buf中的nbytes个字节写入文件，返回实际写入的字节数。

## 5) lseek

```
pos = lseek(int fd, long offset, int origin)
```

功能：文件指针定位。





## 8.4 Linux的设备管理

---

### 一、基本思想

#### 1) 设备作为一种特殊文件

字符特殊文件：用于字符设备，如键盘

块特殊文件：用于块设备，如磁盘

- 2) 用户通过文件系统访问设备，VFS为应用程序隐藏设备文件和普通文件的差异
- 3) 每类设备对应自己的驱动程序
- 4) 每个设备对应文件系统的一个I节点，都有一个文件名。

**设备的文件名一般由2部分构成：**

- ✓ 主设备号：设备类型，可唯一确定设备驱动程序和接口
- ✓ 次设备号：具体设备在同类设备中的序号



## 8.4 Linux的设备管理

---

### 二、设备驱动程序与文件系统的接口

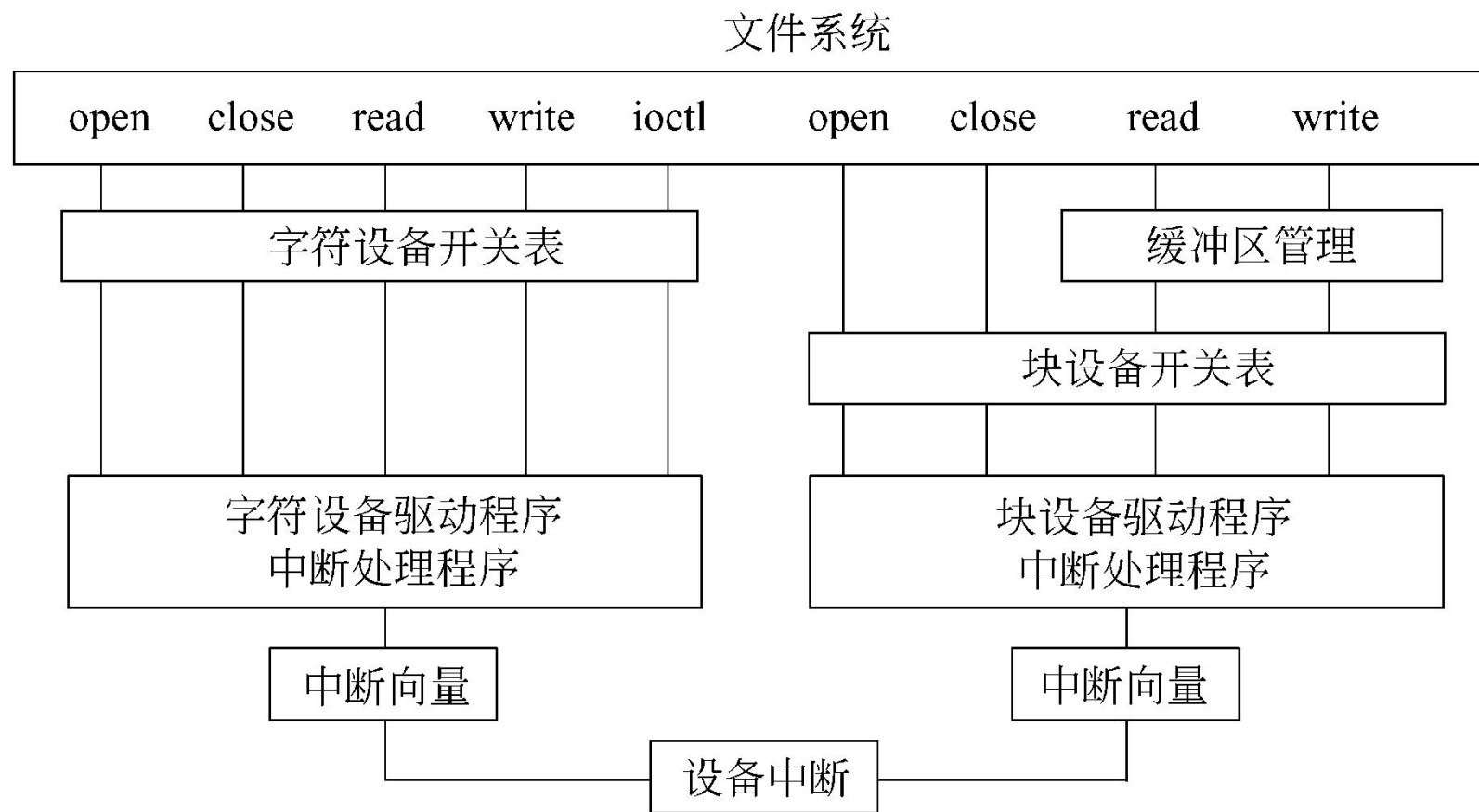
#### 设备开关表:

引导系统调用转向相应驱动程序中的函数，包括块设备开关表与字符设备开关表

指出缺省的文件操作（open, close, read, write等）与相应设备操作函数的对应关系

例如：进程请求分配一个设备，即打开一个设备，可以通过系统调用open()来完成。

## 8.4 Linux的设备管理



文件系统与设备驱动程序的关联



# 设备开关表例

表 10-2 用于块设备文件的函数与缺省文件的操作方法对应表

| 方 法     | 用于块设备文件的函数          | 方 法   | 用于块设备文件的函数           |
|---------|---------------------|-------|----------------------|
| open    | blkdev_open()       | write | generic_file_write() |
| release | blkdev_close()      | mmap  | generic_file_mmap()  |
| llseek  | block_llseek()      | fsync | block_fsync()        |
| read    | generic_file_read() | ioctl | blkdev_ioctl()       |



## 8.4 Linux的设备管理

---

### 三、块设备缓冲

系统在文件系统和设备驱动程序之间设置了缓冲池  
含有最近使用过的数据块



# 第9章 死锁

---

## 9.1 死锁的基本概念

## 9.2 死锁的检测与恢复

## 9.3 死锁避免

## 9.4 死锁预防



## 9.1 死锁的基本概念

---

### 一、什么是死锁 (Deadlock) ?

一个进程集合中的每个进程都在等待只能由该集合中的其它进程才能引发的事件，这种状态称作死锁。

**一组竞争系统资源的进程由于相互等待而出现“永久”阻塞。**



## 9.1 死锁的基本概念

---

为什么会出现死锁？

竞争资源，而资源有限。

例如，2个进程A、B，都需要资源R1、R2

若A：拥有R1，申请R2

若B：拥有R2，申请R1

如何？





# 9.1 死锁的基本概念

---

## 二、什么情况下会产生死锁？

**4个必要条件：**coffman(1971)年提出

### 1) 互斥条件

每个资源要么被分配给了1个进程，要么空闲

### 2) 占有及等待（部分分配）条件

已经得到了资源的进程要申请新的资源

### 3) 不可剥夺条件

已经分配给一个进程的资源不能被剥夺，只能由占有者显式释放

### 4) 环路等待条件

存在由2个或多个进程组成的一条环路，

该环路中的每个进程都在等待相邻进程占有的资源



## 9.1 死锁的基本概念

---

### 三、如何处理死锁？

#### 1、由OS处理

- ✓ 检测死锁并恢复
- ✓ 分配资源时避免死锁
- ✓ 假装没看见（鸵鸟策略）：多数OS对待死锁的策略

#### 2、由应用程序本身预防死锁



## 9.2 死锁的检测与恢复

---

### 一、死锁的检测方法

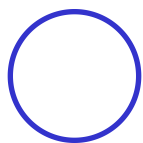
#### 1. 每种资源只有1个

借助于**资源分配图**

**方法：**构造资源分配图，若存在环，则环中进程死锁。

# 死锁的检测方法

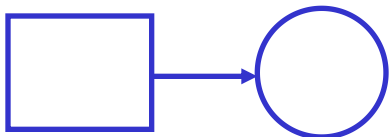
资源分配图：



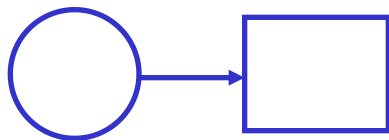
： 进程



： 资源



： 进程已占有资源



： 进程申请资源，处于等待状态



# 死锁的检测方法

**【例】** 设系统中有进程7个：A~G，资源6种：R~W，每种资源只有1个。  
资源占有及申请情况如下：

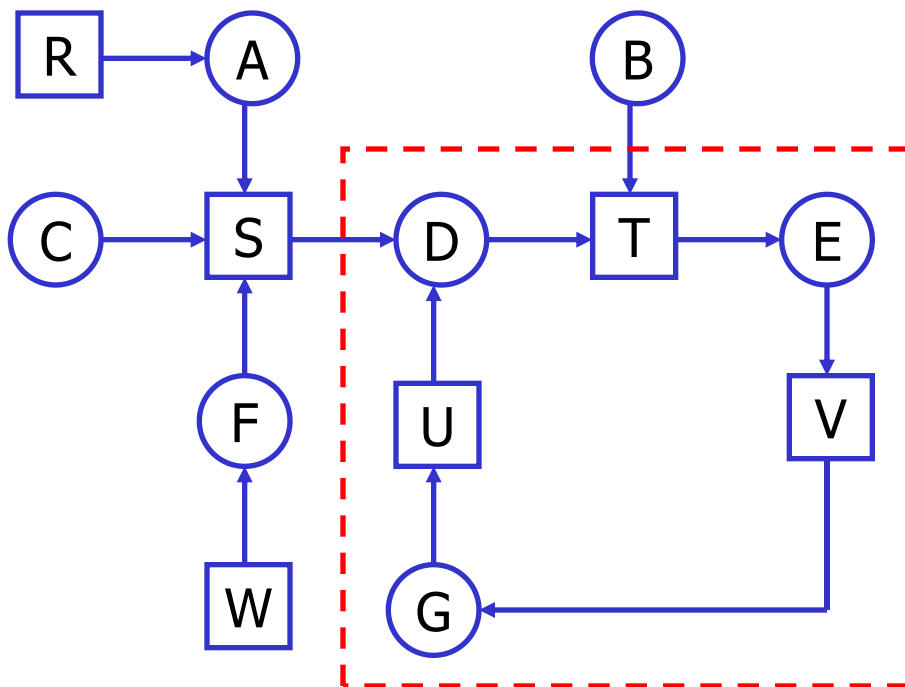
是否存在死锁？

若存在，涉及哪些进程？

| 进程 | 占有资源 | 申请资源 |
|----|------|------|
| A  | R    | S    |
| B  |      | T    |
| C  |      | S    |
| D  | U    | S, T |
| E  | T    | V    |
| F  | W    | S    |
| G  | V    | U    |

## 9.2 死锁的检测与恢复

构造资源分配图如下：



存在环。可以看出，  
D、E、G死锁。



# 死锁的检测方法

---

## 2. 每种资源有多个

设有M种资源，N个进程

### 1) 数据结构

$E[M]$ : 总资源数;  $E[i]$ : 资源i的个数

$A[M]$ : 当前可用资源数;  $A[i]$ : 资源i的可用数

$C[N][M]$ : 当前分配矩阵;  $C[i][j]$ : 进程i对资源j的占有数

第i行是进程i当前占有的资源数

$R[N][M]$ : 申请矩阵;  $R[i][j]$ : 进程i对资源j的申请数

第i行是进程i申请的资源数

$F[N]$ : 进程标记;  $F[i]$ 取0/1, 为1表示进程i能够执行

已分配资源数 + 可用资源数 = 总资源数



# 死锁的检测方法

---

## 2) 算法

① 置 $F[0] \sim F[N]$ 为0;

② 寻找一个满足下列条件的进程 $i$ :

$F[i] == 0$  且

$R[i] \leq A$ , 即 $R[i][j] \leq a[j]$ ,  $a \leq j \leq M$  //进程 $i$ 申请的资源可满足

③ 若找不到这样的进程, 则算法终止;

④  $A = A + C[i]$ ;

$F[i] = 1$ ;

转②继续;

算法结束后, 所有 $F[i] == 0$ 的进程( $i$ )都会死锁。

**该算法的思想: 查找一个进程, 使可用资源能满足该进程的请求; 设想让该进程执行直到完成, 再释放它占有的资源。**





# 死锁的检测方法

【例】3个进程，4种资源。已知

总资源数 $E = (4, 2, 3, 1)$ ；可用资源数 $A = (2, 1, 0, 0)$

$$\text{当前分配矩阵} C = \begin{bmatrix} 0, 0, 1, 0 \\ 2, 0, 0, 1 \\ 0, 1, 2, 0 \end{bmatrix} \quad \text{申请矩阵} R = \begin{bmatrix} 2, 0, 0, 1 \\ 1, 0, 1, 0 \\ 2, 1, 0, 0 \end{bmatrix}$$

检测步骤：

- 1) 进程3可执行，释放其拥有的资源后， $A = (2, 2, 2, 0)$
- 2) 进程2可执行，使  $A = (4, 2, 2, 1)$ 。

不存在死锁。



# 死锁的检测方法

---

如果分配矩阵C如下，其它不变，会死锁吗？

$$\text{当前分配矩阵} C = \begin{pmatrix} 0, 0, 2, 0 \\ 2, 0, 1, 1 \\ 0, 1, 0, 0 \end{pmatrix}$$



## 9.2 死锁的检测与恢复

---

### 二、何时检测死锁？

- 1) 每当有资源请求时；
- 2) 周期性检测；
- 3) 每当CPU的使用率降到某一阈值时。

死锁检测会占用大量的CPU时间



## 9.2 死锁的检测与恢复

---

### 三、如何从死锁中恢复？

#### 1) 剥夺法恢复

将某一资源从一个进程抢占过来给另一个进程使用

不能影响原进程的最终执行结果

取决于资源的特性

#### 2) 回退法恢复

从各进程最近的检查点 (check point) 逐次重新启动

#### 3) 杀死进程来恢复

最好杀死可重复执行、不会产生副作用的进程



## 9.3 死锁避免

---

在保证安全的条件下分配资源。

在给每个进程分配资源前，先进行检查：

如果满足该资源请求肯定不会导致死锁，则予以分配；否则，拒绝。



## 9.3 死锁避免

---

### 一、安全状态与不安全状态

#### 安全状态：

若在某一时刻，系统中存在进程序列 $\langle P_1, P_2, \dots, P_n \rangle$ ，如果按此序列依次为每个进程 $P_i$ 分配资源，使每个进程均可顺利完成。则称此时系统的状态为安全状态，称这样的一个进程序列 $\langle P_1, P_2, \dots, P_n \rangle$ 为**安全序列**。

**安全序列的实质是：对于每一个进程 $P_i (1 \leq i \leq n)$ ，它以后尚需要的资源量不超过系统当前剩余资源量与所有进程 $P_j (j < i)$ 当前占有资源量之和**

若在某一时刻，系统中不存在一个安全序列，则称系统处于**不安全状态**。



## 9.3 死锁避免

**【例】**假定系统有三个进程 $P_1$ 、 $P_2$ 、 $P_3$ ，共有12台磁带机。进程 $P_1$ 总共要求10台磁带机， $P_2$ 和 $P_3$ 分别要求4台和9台。设在 $T_0$ 时刻，进程 $P_1$ 、 $P_2$ 和 $P_3$ 已经获得5台、2台和2台，还有3台空闲没有分配。

| 进程    | 最大需求 | 已分配 |
|-------|------|-----|
| $P_1$ | 10   | 5   |
| $P_2$ | 4    | 2   |
| $P_3$ | 9    | 2   |

$T_0$ 时刻系统是安全的。这时存在一个安全序列 $\langle P_2, P_1, P_3 \rangle$



## 9.3 死锁避免

---

### 说明:

- (1) 系统在某一时刻的安全序列可能不唯一，但这不影响对系统安全状态的判断。
- (2) **处于安全状态一定可以避免死锁，而系统处于不安全状态则仅仅可能进入死锁状态。**





## 9.3 死锁避免

---

### 二、银行家算法

银行家算法是最有代表性的避免死锁算法  
是Dijkstra 1965年提出的。

**基本思想：**一个小城镇的银行家，他向一群客户分别承诺了一定金额的贷款，而他知道不可能所有客户同时都需要最大的贷款额。在这里，我们可将客户比作进程，银行家比作操作系统。银行家算法就是对每一个客户的请求进行检查，检查如果满足它是否会引起不安全状态。如果是，则不满足该请求；否则，便满足。



# 银行家算法

---

## 银行家算法的实质：

设法保证系统动态分配资源后不进入不安全状态，以避免可能产生的死锁。

每当进程提出资源请求且系统的资源能够满足该请求时，系统将判断如果满足此次资源请求，系统状态是否安全；

如果判断结果为安全，则给该进程分配资源，否则不分配资源，申请资源的进程将阻塞。



# 银行家算法

【例】已知4个进程：A、B、C、D，总资源数 = 10，进程对资源的最大需求量分别为6、5、4、7。

| 进程 | 已分配 | 最大需求 |
|----|-----|------|
|----|-----|------|

|   |   |   |
|---|---|---|
| A | 0 | 6 |
| B | 0 | 5 |
| C | 0 | 4 |
| D | 0 | 7 |

剩余：10

安全

安全序列：随意

| 进程 | 已分配 | 最大需求 |
|----|-----|------|
|----|-----|------|

|   |   |   |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

剩余：2

安全

安全序列：C, B, A, D

| 进程 | 已分配 | 最大需求 |
|----|-----|------|
|----|-----|------|

|   |   |   |
|---|---|---|
| A | 1 | 6 |
| B | 2 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

剩余：1

不安全



# 银行家算法

---

## 银行家算法中所用的主要数据结构:

### (1) 可用资源向量 Available

记录资源的当前可用数量。如果  $Available[j] = k$ , 表示现有  $R_j$  类资源  $k$  个。

### (2) 最大需求矩阵 Max

每个进程对各类资源的最大需求量

### (3) 分配矩阵 Allocation

已分配给每个进程的各类资源的数量

### (4) 需求矩阵 Need

进程对各类资源尚需要的数量

$$Need = Max - Allocation$$

### (5) 请求向量 Request

记录某个进程当前对各类资源的申请量



# 银行家算法

---

当 $P_i$ 发出资源请求后，系统按下列步骤进行检查：

- (1)如果 $Request > Need[i]$ ，则出错;
- (2)如果 $Request > Available$ ，则 $P_i$ 阻塞;
- (3)假设把要求的资源分配给进程 $P_i$ ，则有

$$Available = Available - Request;$$

$$Allocation[i] = Allocation[i] + Request;$$

$$Need[i] = Need[i] - Request;$$

检查此次资源分配后，系统是否处于安全状态。若安全，正式将资源分配给进程 $P_i$ ，以完成本次分配；否则，恢复原来的资源分配状态，让进程 $P_i$ 等待。



# 银行家算法

---

为进行安全性检查，定义数据结构：

## ① 工作向量Work

表示系统可提供给进程的各类资源的数量

开始时， $Work = Available$

## ② 向量Finish

标记进程是否可运行结束。开始时先令 $Finish[i] = false$ ；当有足够的资源分配给进程时，令 $Finish[i] = true$



# 银行家算法

---

## 安全性检查的步骤:

(1)  $Work = Available$ ;

对于每个  $i$ , 置  $Finish[i] = false$ ;

(2) 寻找满足条件的  $i$ :

$Finish[i] = false$ , 且  $Need[i] \leq Work$ ;

如果不存在, 则转(4);

(3) 假设把资源分配给进程  $P_i$ ,  $P_i$  可以运行到结束并释放资源

$Work = Work + Allocation[i]$ ;

$Finish[i] = true$ ;

转(2);

(4) 若对所有  $i$ ,  $Finish[i]$  都等于  $true$ , 则系统处于安全状态, 否则处于不安全状态。



# 银行家算法

---

银行家算法虽然可以避免死锁，但**缺乏实用价值**，

因为**需要事先知道每个进程对每种资源的最大需求**。

很少有进程能够在运行前就知道其所需资源的最大值，而且进程数也不是固定的，往往在不断地变化（如新用户登录或退出），况且原本可用的资源也可能突然间变成不可用（如磁带机可能坏掉）。因此，在实际中，如果有，也只有极少的系统使用银行家算法来避免死锁。





## 9.4 死 锁 预 防

---

在系统（用户程序）设计时考虑死锁问题

**基本做法：**

使产生死锁的四个必要条件至少有一个不成立



## 9.4 死锁预防

---

### 一、破坏互斥条件

互斥是由资源的固有特性决定的，是不能改变的。

可利用的思想：

对资源统一管理，如SPOOLing技术。



## 9.4 死锁预防

---

### 二、破坏不可剥夺条件

当进程申请的资源被其他进程占用时，可以通过操作系统抢占这一资源。

**当资源状态很容易保存并恢复时，这种方法是实用的。**

**举个剥夺的例子：**

换入/换出（剥夺内存）

进程切换（剥夺CPU）



## 9.4 死锁预防

---

### 三、破坏占有及等待条件

基本思想：

**原子性地获得所需全部资源。**

2种方法：

- (1) 每个进程开始执行前一次性地申请它所要求的所有资源，且仅当该进程所要资源均可满足时才给予一次性分配
- (2) 当进程申请资源时，先暂时释放其当前占有的所有资源，然后再试图一次申请获得所需的全部资源



## 9.4 死锁预防

---

### 四、破坏环路等待条件

#### 采用资源有序分配法：

把系统中所有资源编号

每个进程只能按编号的递增次序申请资源

存在的问题：有时不容易给出合适的资源编号，特别是资源数量庞大时



# 破坏环路等待条件

例如：1, 2, 3, ..., 10

**P1:**

**申请1**

**申请3**

**申请9**

...

**P2:**

**申请1**

**申请2**

**申请5**

...

**P3 ..... P10**