

UNIX/Linux

# 操作系统内核结构

刘玏 教授

电子科技大学信息与软件工程学院

# 课程概述

## 一. 课程内容简介

### 1、讲授范围

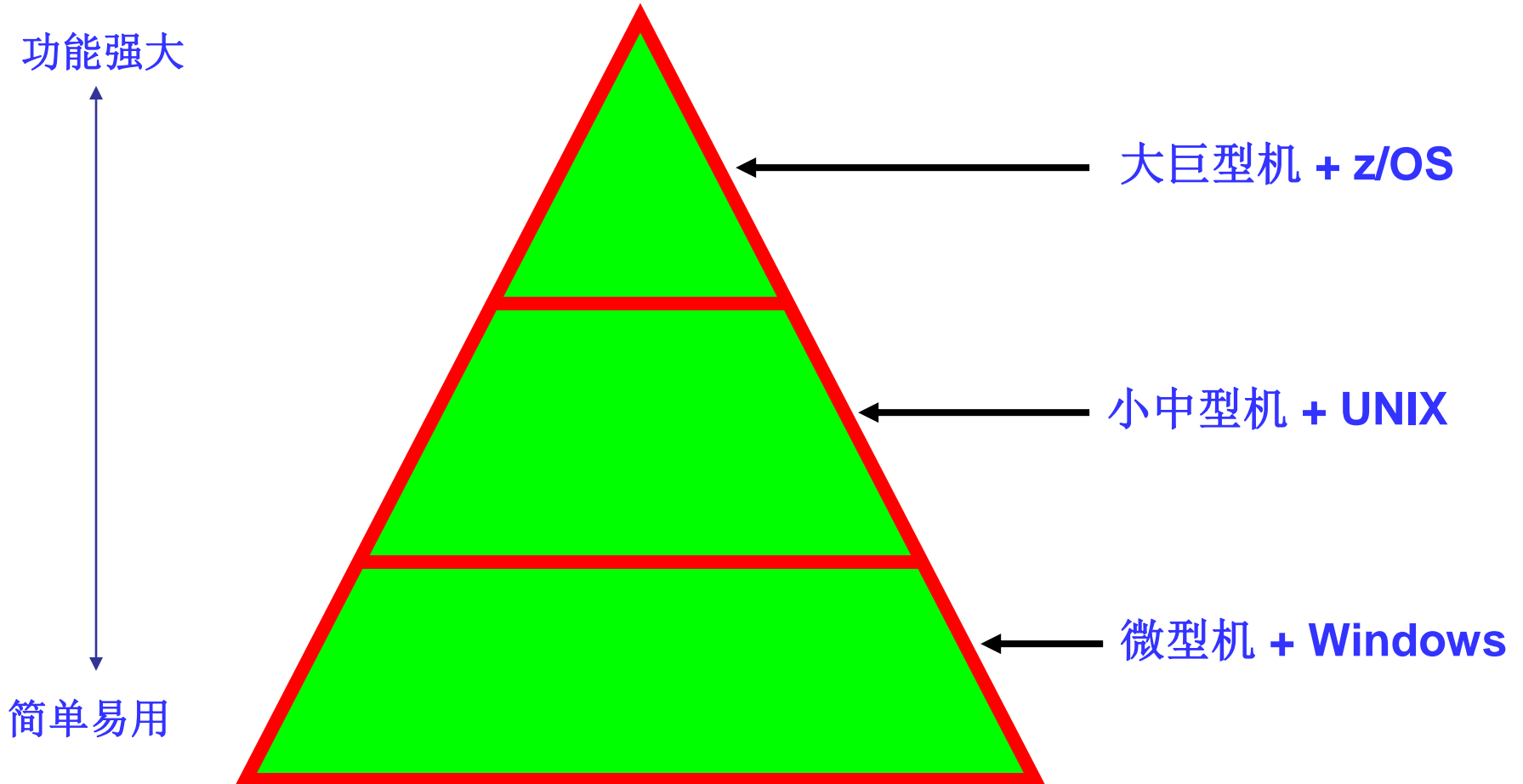
具体的技术系统及其算法和实现流程，而不是操作系统基本概念；

### 2、通用操作系统的现状和分类

**MS\_Windows类** ---- 结构简单、使用方便、效率低、安全性低

**UNIX类** ---- 运行高效、结构通用、安全可靠、适应能力强、系统较复杂

**z/OS类** ---- 功能强大、处理能力巨大、系统复杂、较封闭



### 3、UNIX操作系统的根本特点

#### 分时多用户、开放性

分时多用户：

多个用户多个进程同时在一个系统中运行

系统资源高度共享、有效协调——并发

开放性：

标准化——结构上的一致性

可移植性——应用程序的编码及系统应用接口

可互操作性——可保持用户原来的使用习惯

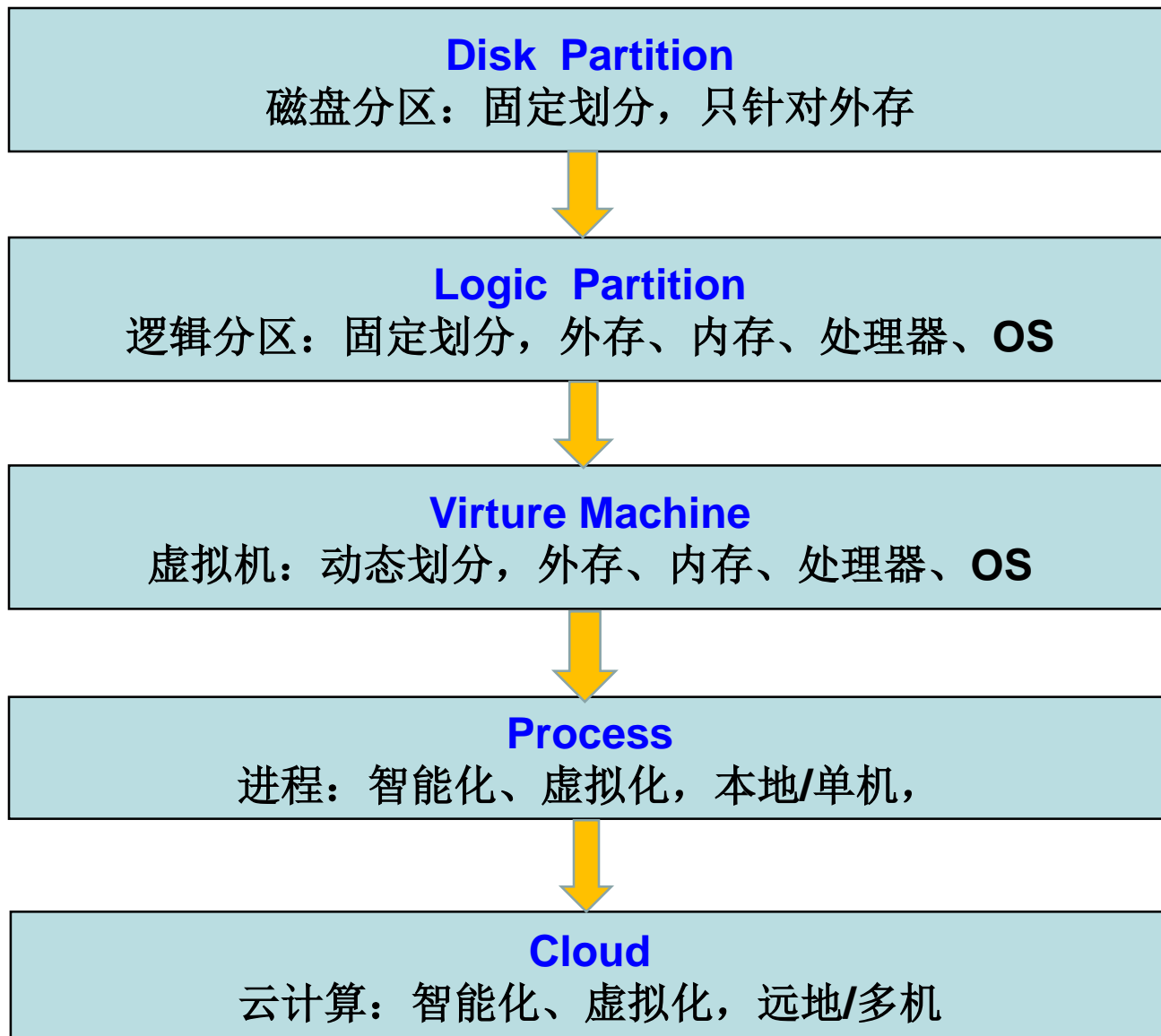
异种机之间的互操作

### 4、教学难点

多用户多进程——同步/互斥、数据一致性、访问安全性

开放性——硬件依赖性、结构伸缩性、广泛适应性

# UNIX/Linux操作系统的精髓是——进程



## 二、教学目的

### 1、了解主流操作系统的发展方向

低端操作系统 VS 高端操作系统

### 2、掌握UNIX类操作系统的内部结构和主要算法

文件、文件系统、进程、时钟、输入输出

### 3、学习大型程序设计的方法和理念

系统结构、功能流程、数据安全、思维模式

### 4、奠定系统开发和应用开发的基础

功能选择、层次划分、应用系统模式的确定

### 三、教材

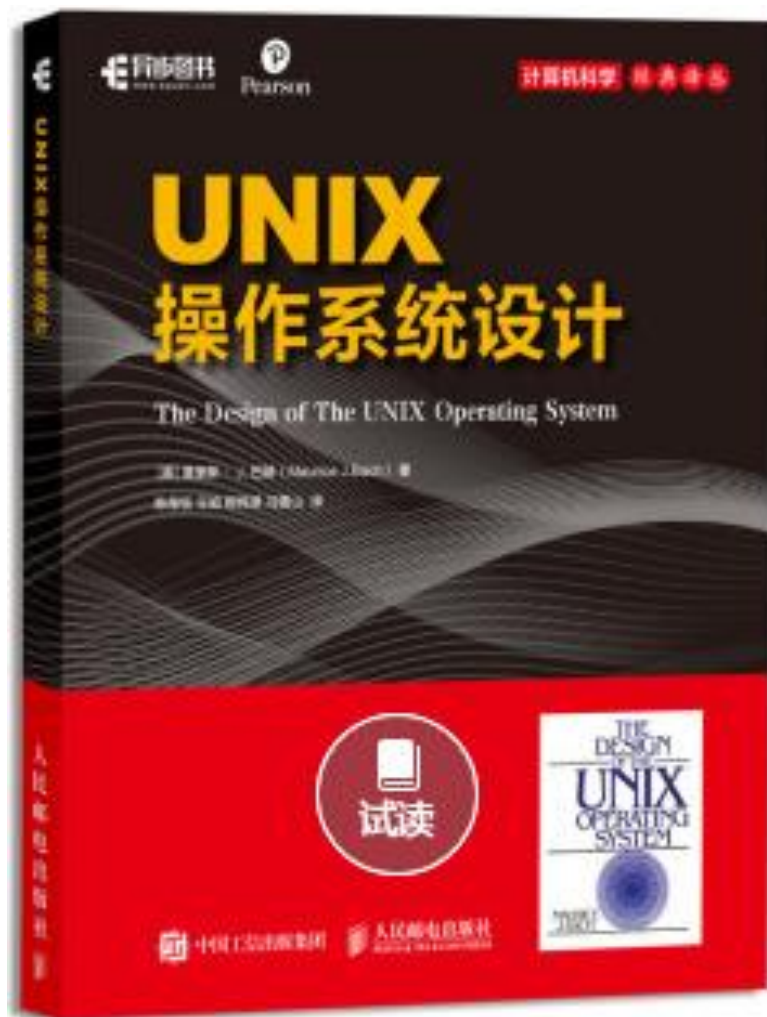
#### 《UNIX操作系统设计》

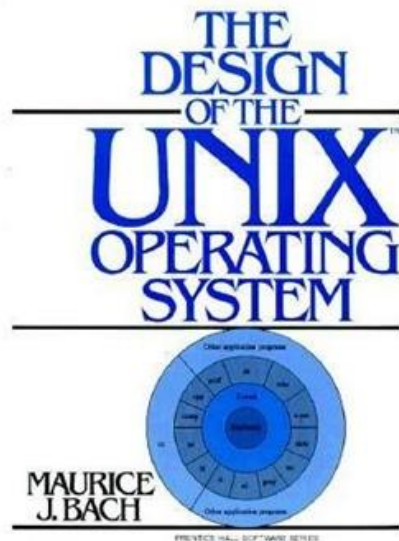
(The Design of The UNIX Operating System)

(美) Maurice J.Bach 著

陈葆珏 王旭 柳纯录 冯雪山 译

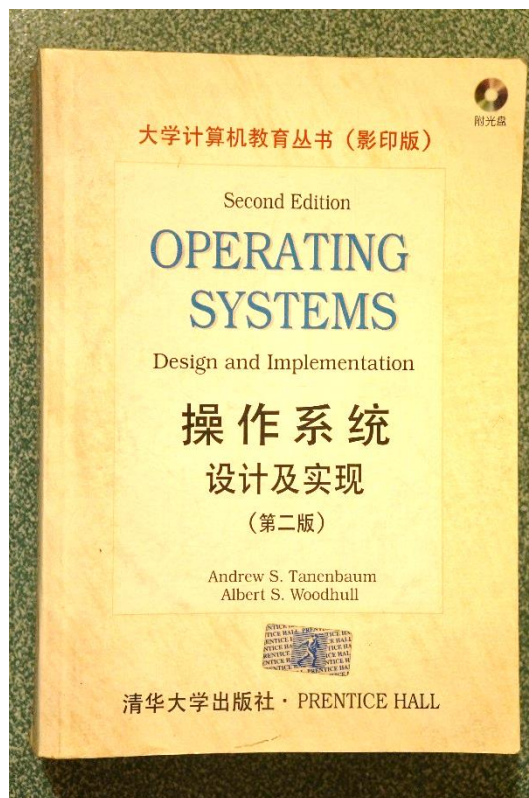
人民邮电出版社 2019年6月  
出版





## 参考资料:

1. The Design of the UNIX Operating System（影印）. Maurice J. Bach，人民邮电出版社。



2. Operating Systems: Design and Implementation（第二版）（影印）. Andrew S. Tanenbaum 等，清华大学出版社。



## 四、考核说明

1、考核方式：考查 / 考试

2、成绩构成：平时成绩 **20%**（考勤、交流讨论）

报告成绩 **30%**（研究报告）

期末成绩 **50%**（期末报告或试卷）

# 第一章 系统概貌

- 1.1 发展状况

- 1、发展历史及版本

- v.0 1970年

- Ken Thompson 和 Dennis Ritchie

- PDP-7 汇编语言

- UNICS

- v.1 1971年

- PDP-11 汇编语言

- UNIX

- v.2 1972年 增加管道功能

**v.5 1973年**  
**Dennis Ritchie**  
**B language ---- C language**

**重写UNIX**

**第一个高级语言OS**

**v.6 1975年**

**对外发表UNIX**

**大学和科研单位应用**

**v.7 1978年**

**第一个商业版本**

**我国开始深入研究应用的最早版本**

**System III**

**1981年**

**完全转向为社会提供的商品软件**

## System V

**1983年**

**系统功能稳定完善**

**公布号: 1.0、2.0、2.3、3.5、4.0、4.2、4.3**

**现在最后版本为 System V Release 4 (SVR4)**

## 2、主要分支和兼容版本

- **BSD** 加州大学伯克利分校
- **XENIX/OpenServer** Microsoft、SCO公司
- **HP-UX** HP公司
- **AIX** IBM
- **Solaris** SUN公司
- **IRIX** SGI公司
- **Ultrix** DEC公司
- **Linux** 开放源代码
- **Android/iOS** 嵌入式**UNIX/Linux**

### 3、基本功能特征

#### ① 交互式分时多用户

- 人机间实时交互数据
- 多个用户可同时使用一台机器
- 每个用户可同时执行多个任务

#### ② 软件复用

- 每个程序模块完成单一的功能
- 程序模块可按需任意组合
- 较高的系统和应用开发效率

#### ③ 可移植性强

- 数千行汇编码，数十万行**C**语言代码

- ④ 配置灵活, 适应性强
  - 小内核, 参数灵活可调
  - 核外应用系统, 任意裁减
  - 限制规则很少
- ⑤ 界面方便高效
  - 内部: 系统调用丰富高效
  - 外部: **shell**命令灵活方便可编程
  - 应用: **GUI** 清晰直观功能强大
- ⑥ 安全机制完善
  - 口令、权限、加密等措施完善
  - 抗病毒结构
  - 误操作的局限和自动恢复功能

## ⑦ 多国语言支持

- 支持全世界现有的几十种主要语言

## ⑧ 网络和资源共亨

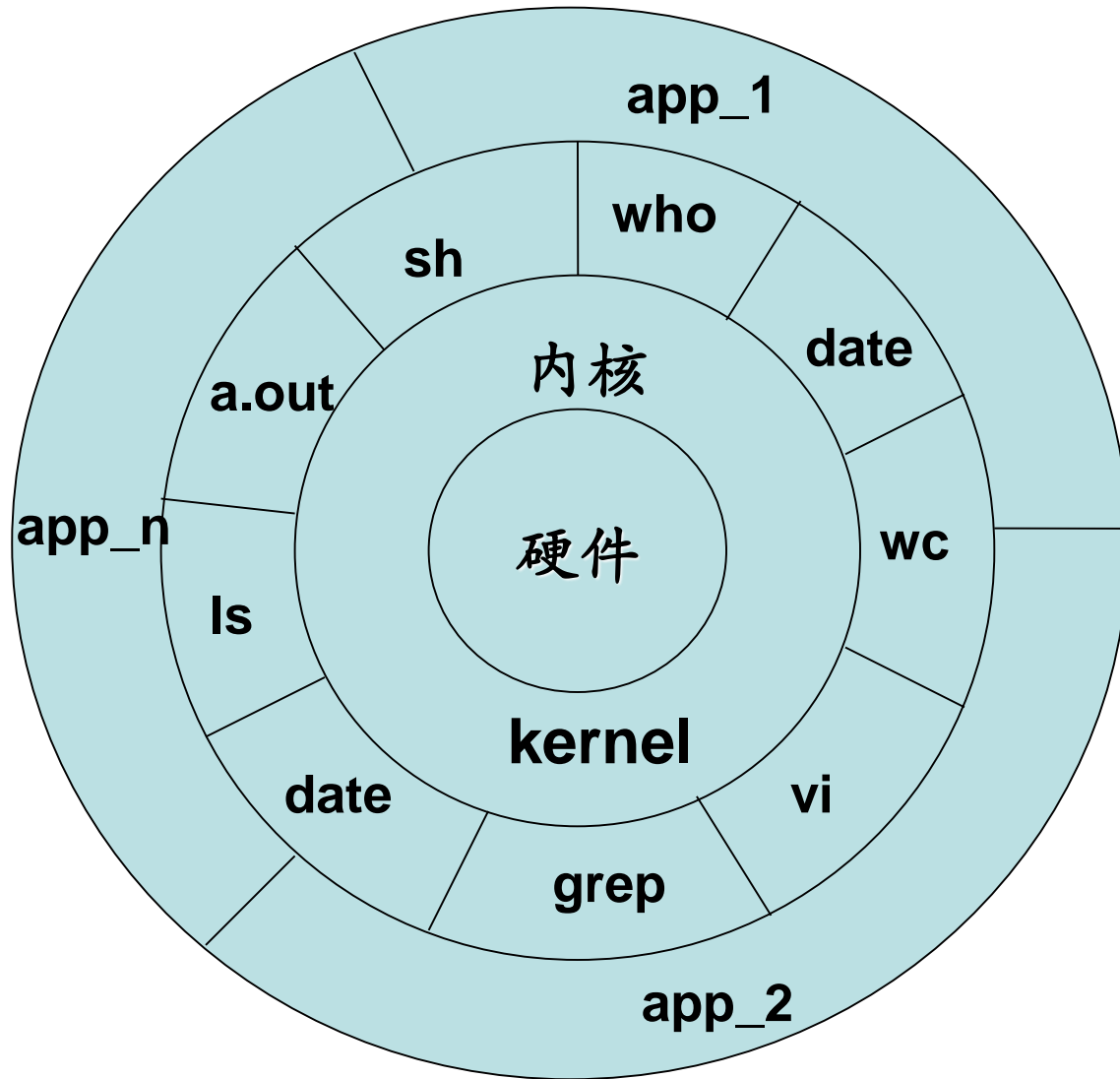
- 内部：多进程结构易于资源共享
- 外部：支持多种网络协议

说明：

- 1、其它操作系统可能包含部分上述**UNIX**的特征，但非全部（如**NT**就有部分多用户系统特征）
- 2、这些特征有些是核心直接实现的，有些是由核心提供实现这种特征的方便性和可能性，而由使用者来实现的。



- 1.2 系统结构



UNIX操作系统的整体结构

## 系统调用（**system call**）

以函数形式提供给核外的命令和上层应用系统使用的一组程序，涵盖操作系统的所有功能。是应用程序请求操作系统服务的唯一通道。

## 内核（**kernel**）

系统调用的集合及实现系统调用的内部算法就形成操作系统核心

## • 1.3 用户看法

进程和文件是**UNIX**操作系统中最基本的两个概念（抽象）

### 进程：

所有处在运行期间的程序实例都是进程

一个进程就是处在运行期间的一个程序实例

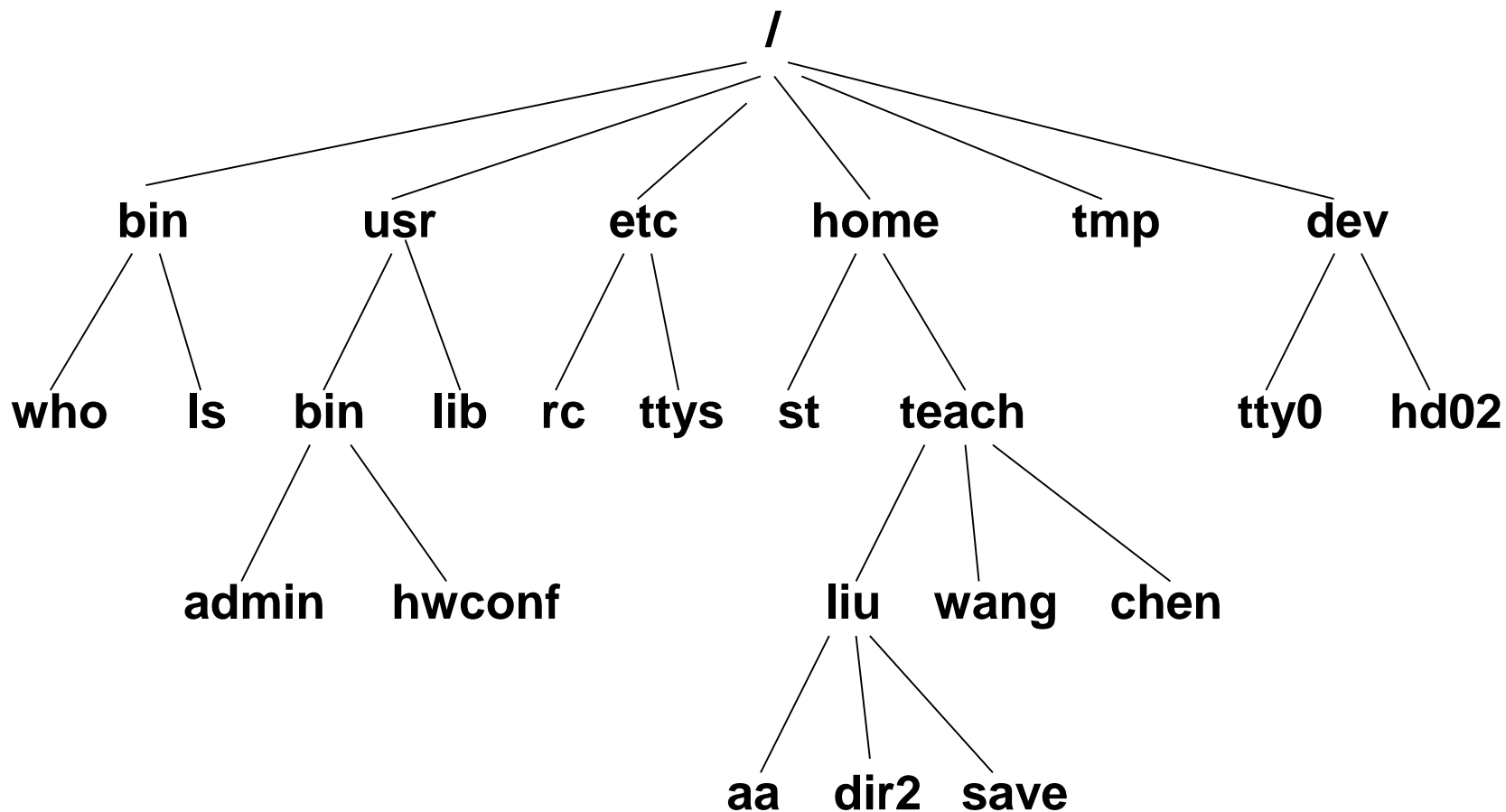
涵盖所有的动态概念

### 文件：

所有静态的无形数据和有形硬件设备

源程序、命令、图片、邮件、打印机、内存、磁盘等

## 1.3.1 文件系统



UNIX文件系统树示例

## UNIX文件系统的特征:

### 1、树状层次结构

树根、树枝、树叶、路径

### 2、对文件数据的一致对待

文件为有序无格式的字节流，逻辑意义由使用者解释

### 3、文件管理

建立、删除、修改、备份、移动、替换 —— 上层操作

存储空间的分配和释放 —— 下层操作

### 4、文件的访问和保护

索引节点 (**inode**)、文件描述符(**fd**)

用户分组、权限划分

### 5、设备文件管理

统一各外部设备的访问模式

```

char buffer[2048];
main(int argc, char *argv[])
{
    int fdold, fdnew;
    if(argc != 3)
    {
        printf("Need 2 arguments for copy program\n");
        exit(1);
    }
    fdold = open(argv[1], O_RDONLY);
    if (fdold == -1)
    {
        printf("cannot open file %s\n", argv[1]);
        exit(1);
    }
    fdnew = creat(argv[2], 0666);
    if(fdnew == -1)
    {
        printf("cannot create file %s\n", argv[2]);
        exit(1);
    }
    copy(fdold, fdnew);
    exit(0);
}

copy(int old, int new)
{
    int count;
    while((count = read(old, buffer, sizeof(buffer))) > 0)
        write(new, buffer, count);
}

```

## 1.3.2 处理环境

### 程序：可执行的文件



文件头包括：

- 文件的幻数（**magic number**）
- 编译器的版本号
- 机器类型
- 数据段、正文段、工作变量的段大小
- 程序入口点

进程:

程序的一次执行实例

一个程序可同时有多个实例; 系统中可同时有多个进程

父进程:

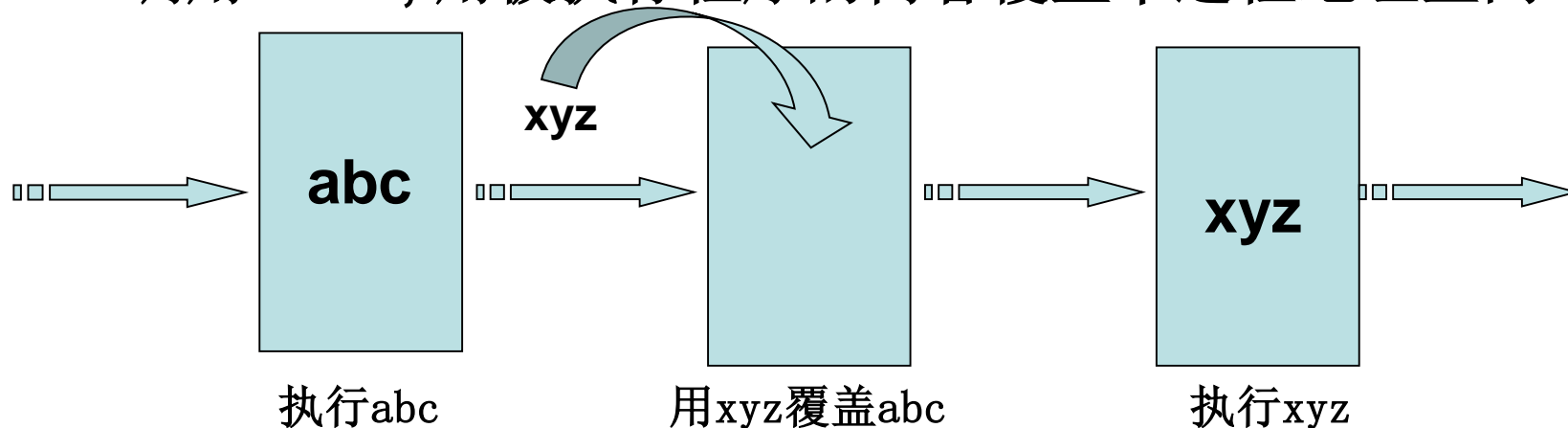
调用系统调用**fork**的进程

子进程:

由系统调用**fork**产生的新进程

执行程序:

调用**exec1**, 用被执行程序的内容覆盖本进程地址空间





例子:

执行可运行文件**copy**，其功能是拷贝文件，其运行格式为：  
**copy oldfile newfile**

另一个名为**cpfile**的程序具体调用**copy**，**cpfile**的源程序如下：

```
main(int argc, char *argv[ ])
{
    if (fork() == 0)
        execl("copy", "copy", argv[1], argv[2]], 0);
    wait((int *)0);
    printf("copy done\n");
}
```

在用户环境下，程序的执行通常由命令解释器**shell**来完成，标准的命令格式为：

**cmd [-options] [arguments]**

**shell**可识别的命令类型有：

1、简单命令

**cat file1**

2、多条命令

**who; date; ps**

3、复合命令

**ps -e | grep student2**

**(ls ; cat file3 ; pwd) > run\_log**

4、后台命令

**ls -lR /home/teacher > tlist &**

### 1.3.3 构件原语

源于“软件复用”和“模块组装”理念

程序内部：

简单功能划分；纯代码设计

程序外部：

使用构件原语进行功能重叠和组装

**UNIX**包含两种构件原语：

- ① 输入/输出重定向
- ② 管道

## I/O重定向 (I/O redirect) :

一个进程通常(**default**)打开三个文件:

标准输入文件 (**fd=0**)

标准输出文件 (**fd=1**)

标准错误输出文件 (**fd=2**)

例如:

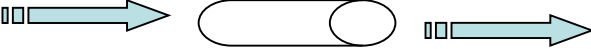
```
grep abc
```

```
grep abc < file1
```

```
grep abc < file1 > file2
```

```
grep abc < file1 > file2 2> file3
```

## 管道 (pipe) :

A进程的输出  B进程的输入

A进程将标准输出重新定向到管道中去;

B进程将标准输入重新定向从管道中来。

例如:

```
ps -e | grep student3
```

查看当前系统中与用户**student3**相关的进程

- **1.4 操作系统服务**

**UNIX**操作系统提供五种主要的服务（也是**UNIX**核心的五个重要组成部分）：

- 1. 进程管理**

建立、终止、挂起、通信等

- 2. 时钟管理**

分时共享**cpu**，时间片，调度

- 3. 存储管理**

二级存贮器（内存和对换区），分配主存

- 4. 文件系统管理**

文件操作：读、写、更名、拷贝 .....

二级存贮管理：分配和收回存贮区和索引节点

- 5. 设备管理**

对**I/O**设备进行有控制的存取（多进程系统的特征）

内核提供的服务的特点：

## 服务是透明的

### ①文件类型透明：

用户可不关心是普通文件还是外部设备，但O.S自己要关心文件类型！

### ②文件系统的透明：

文件系统类型、存放的物理位置。

### ③存贮方式透明：

文件的存放位置、存放方式、存放格式

### ④各用户进程能得到核心相同服务：

无论系统程序还是用户程序，平等对待，分时运行

## • 1.5 硬件假设

(假设机器硬件只支持的运行状态)

UNIX系统上进程的执行分成两种状态:

用户态、核心态

用户态:

进程正在执行用户代码时的状态

核心态:

进程正在执行系统代码（系统调用）时的状态

用户态和核心态的区别:

①用户态: 进程只能存取自己的地址空间

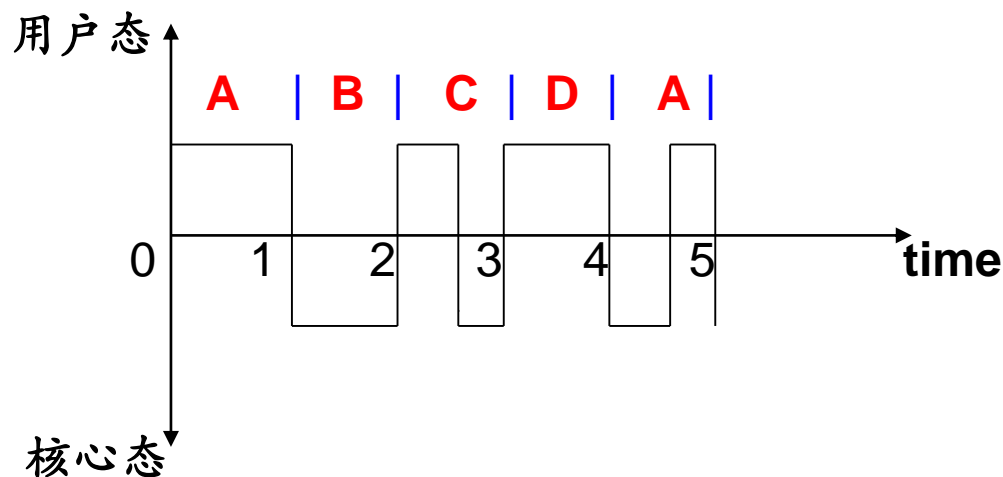
核心态: 进程可存取核心和用户地址空间

②用户态: 不能存取特权指令，只能存取自己的指令和数据

核心态: 除了能存取自己的指令和数据外，还可存取特权指令



一个进程在运行时必须处在，而且只能处在或者核心态或者用户态下：

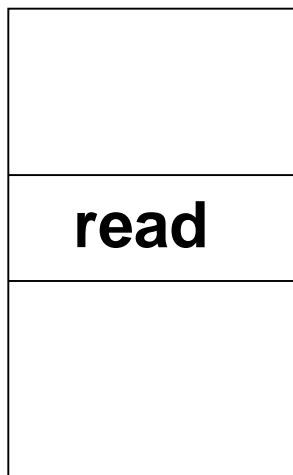


核心态的进程不是与用户进程平行运行的孤立的进程集合，而是每个用户进程的一部分。

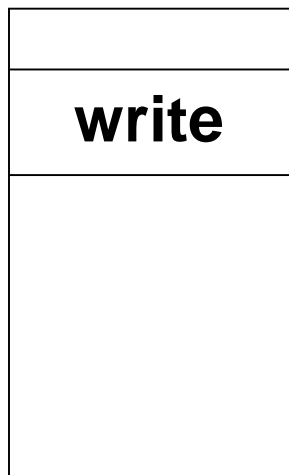
**“核心分配资源”** —— 一个在核心状态下执行的进程分配资源。

一个进程某时在“用户态”下运行，另一时刻又在“核心态”下运行，在其生命周期内可能在这两种状态间切换多次

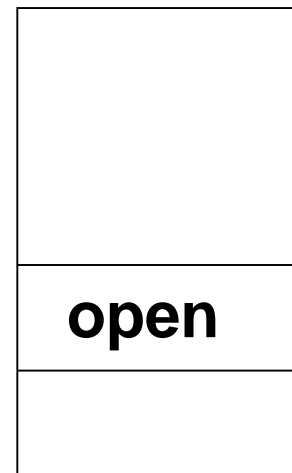
## 核心——处在核心态下的进程的相应部分的集合



**A** 进程



**B** 进程



**C** 进程

硬件是按核心态和用户态来执行操作的，但对这两种状态下正在执行程序的用户是相同对待的。

## 1.5.1 中断与例外


- 中断（要保存上下文）：

来自进程之外的事件（外设、时钟等）引起的，发生在两条指令执行之间，中断服务完毕后从下一条指令继续执行。  
(中断服务是由核心中特殊的函数，而不是特殊的进程来执行的)

- 例外（不保存上下文）：

来自进程内部的非期望事件（地址越界，除数为0等），发生在一条指令执行过程中，例外事件处理完后重新执行该指令。

## 1.5.2 处理机执行级

中断事件	中断级别
硬件故障	 高
时钟	
硬盘	
网络	
终端	
软件中断	
	低

用一组特权指令给处理机设置一个执行级，以屏蔽同级和低级的中断，最大限度地减少其它事件的干扰，使当前任务顺利执行并尽快完成；但开放更高级的中断，以响应更紧迫的请求。

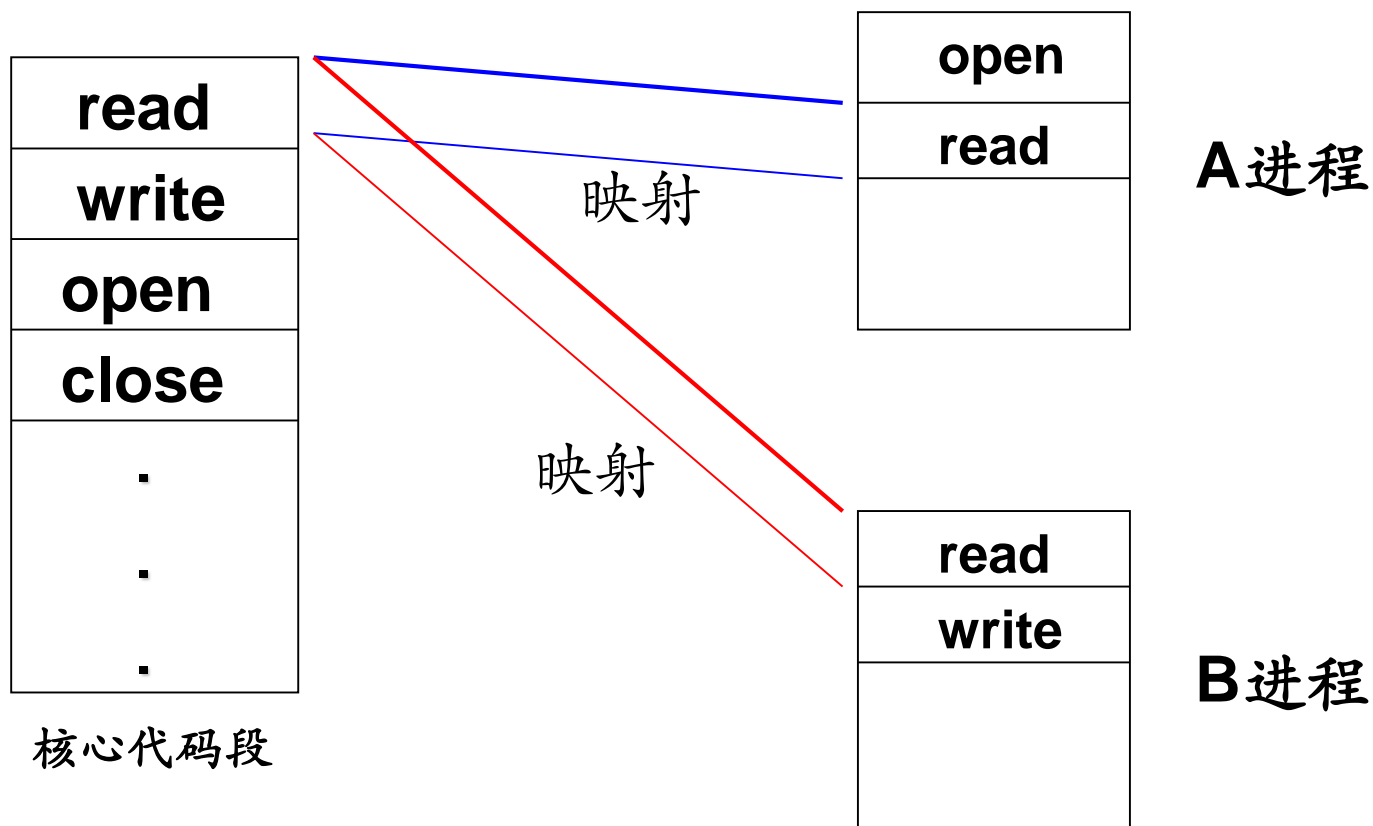
## 1.5.3 存储管理

### UNIX系统中的存储管理原则(或特点):

1. 当前正在执行的进程（全部或部分）驻留在主存中；
2. 核心是永远驻留在主存中的（是永远活动的！）；
3. 编译程序产生的指令地址是虚地址（逻辑地址）；
4. 程序运行时核心与硬件（存储管理部件**MMU**）一起建立虚地址到物理地址的映射。

核心永远是活跃的

普通进程具有特定的生命周期（除非人为设定为无限循环）



只是用户进程中的核心态下运行的代码段常驻内存，而非整个用户进程常驻内存。这些代码段是“可再入段”（或纯代码段、可共享代码段），被各用户进程段共享，为提高运行速度，避免频频访问磁盘，故常驻内存，这些代码段的集合就是**OS**的内核。

# 第二章 核心导言

- 2.1 UNIX操作系统的体系结构

“文件”和“进程”是UNIX系统的两个最基本实体和中心概念，UNIX系统的所有操作都是以这两者为基础的。整个系统核心由以下五个部分组成：

- ① 文件系统：

文件管理和存储空间管理（节点和空间管理）

- ② I/O设备管理：

核心→缓冲→块设备（随机存取设备）

核心→原始设备（raw设备，字符设备，裸设备）

- ③ 进程控制：

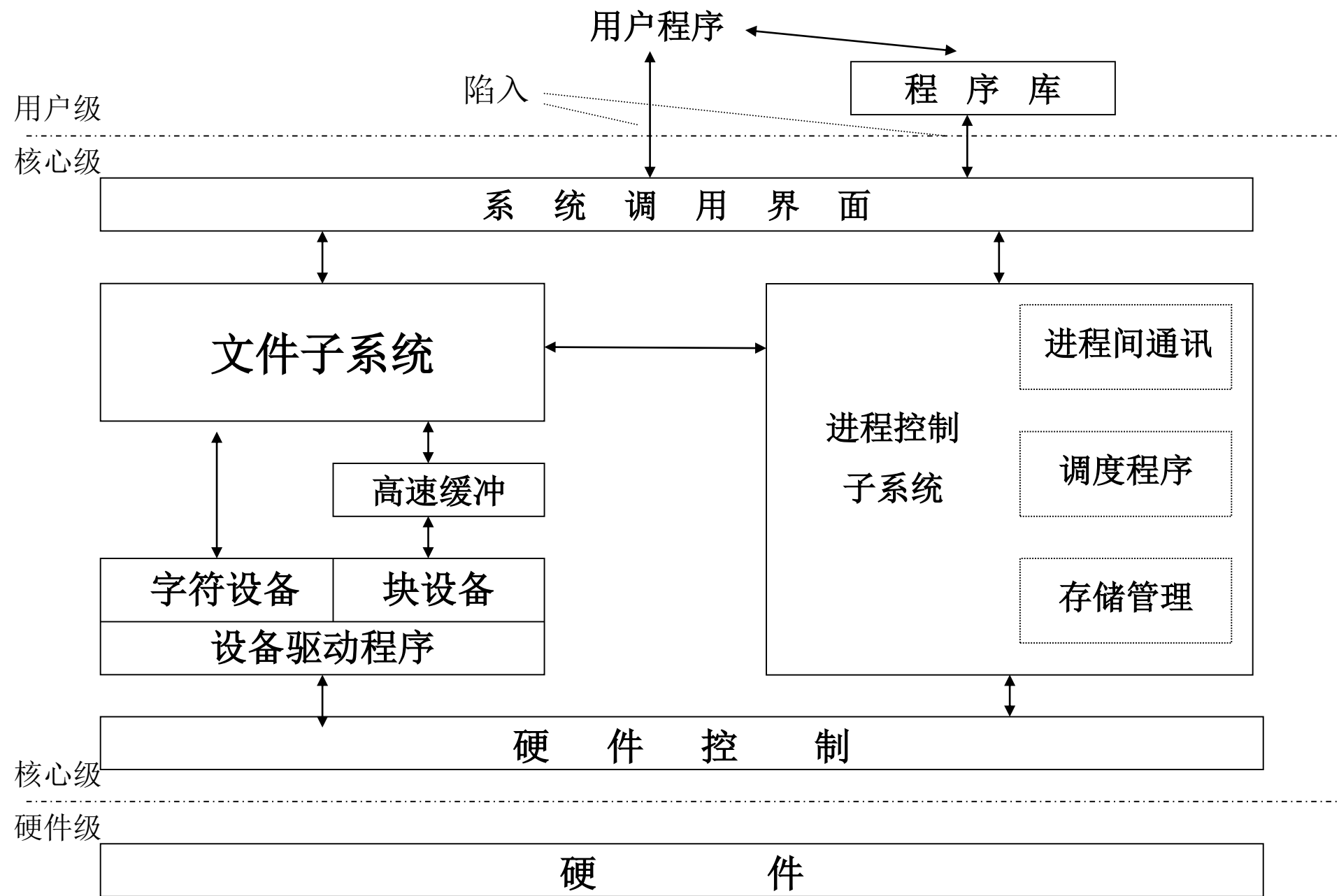
进程的调度、同步和通讯

- ④ 存贮管理：

在主存与二级存储之间对程序进行搬迁

- ⑤ 时钟管理：

把cpu的时间分配给当前最高优先权的进程。





## • 2.2 系统概念

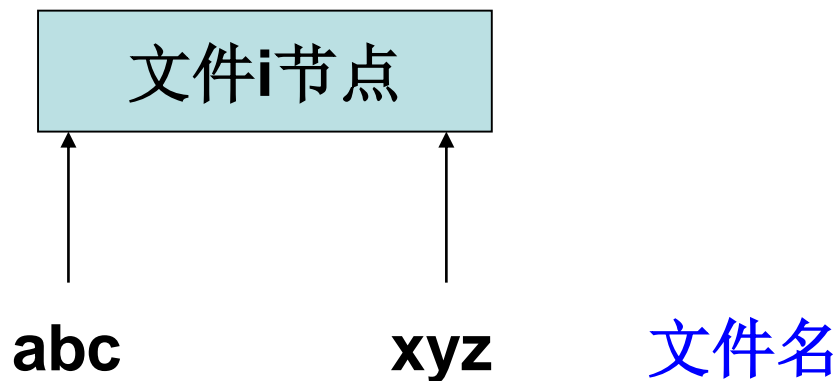
### 2.2.1 文件系统概貌

#### 1. 索引节点 (**index node——inode**)

##### **inode**特征:

- 文件的内部名称（或代号），方便机器操作；
- 每个文件都有一个且只有一个**inode**与之对应；
- **inode**存放文件的静态参数：存放地点、所有者、文件类型、存取权限、文件大小等；
- 每个文件都可以有多个名字，但都映射到同一个**inode**上；
- 各**inode**之间以**inode**号相区别；

## 2. 链结 (link) ——对应命令名 ln



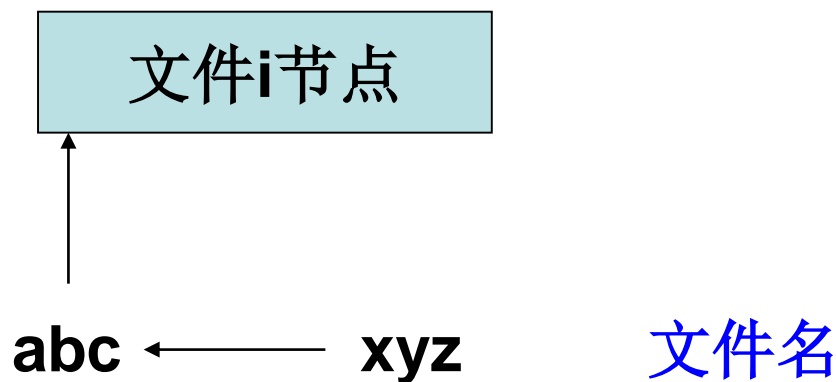
- 一个文件可有多多个名字，多个名字都对应同一个文件i节点，每个名字就是该文件节点的一个链结；
- 一个普通文件的名字个数，就是该文件的链结数；
- 每个链接名可以放在不同的目录下（同一个文件系统下）；
- 删除一个链接名时，文件链接数减一。如链接数不为零，则文件（节点）仍然存在。

## 使用文件链结的目的：

- ①方便用户的使用习惯，如“列目录”，可用**ls**、**dir**、**list**、**lc**等；
- ②误删文件时可补救，又不多占空间。**abc**和**xyz**具有相同的i结点号；
- ③减少移植应用程序时，因使用指定位置的文件，而拷贝该文件到指定位置去的麻烦。

### 3. 符号链结 (symbol link)

——对应命令名 **ln -s**



- 给文件的名称再取一个名字，而不是给文件节点再取一个名字。
- 链接的是“符号”而不是文件，因此“符号”可以是不存在的文件，即无意义的字符串。
- **abc**和**xyz**具有不同的**inode**号，**xyz**的内容是它所指向的文件的字符串，大小是字符串长度为3字节。
- “普通链结”中各名字必须在同一文件系统中，“符号链结”可在不同的文件系统中。

## 4. 活动i节点表（索引节点表）—— **inode**表

在内存中存放当前要使用的文件**inode**的表（或称为活动i节点表），表中的每一个表项对应一个当前正被使用的文件的状态信息。这样要使用（打开）同一个文件的进程不必再到盘上去寻找了，（共享！）

<b>inode</b>
<b>inode</b>
<b>inode</b>
<b>inode</b>
<b>inode</b>
.....

## 5. 用户打开文件表（或称用户文件描述符表）

在系统中每一个进程都有一个描述该进程的数据结构**user**（类似于描述文件的i节点），在**user**中有一个数组，存放一组指针指向系统打开文件表中该进程打开的文件所对应的表项。

**struct file \*u\_ofile[NOFILE]**

**NOFILE** 为每个进程最多可同时打开的文件数，这与系统中的进程数和内存大小以及交换区大小等有关，一般为**20~100**。

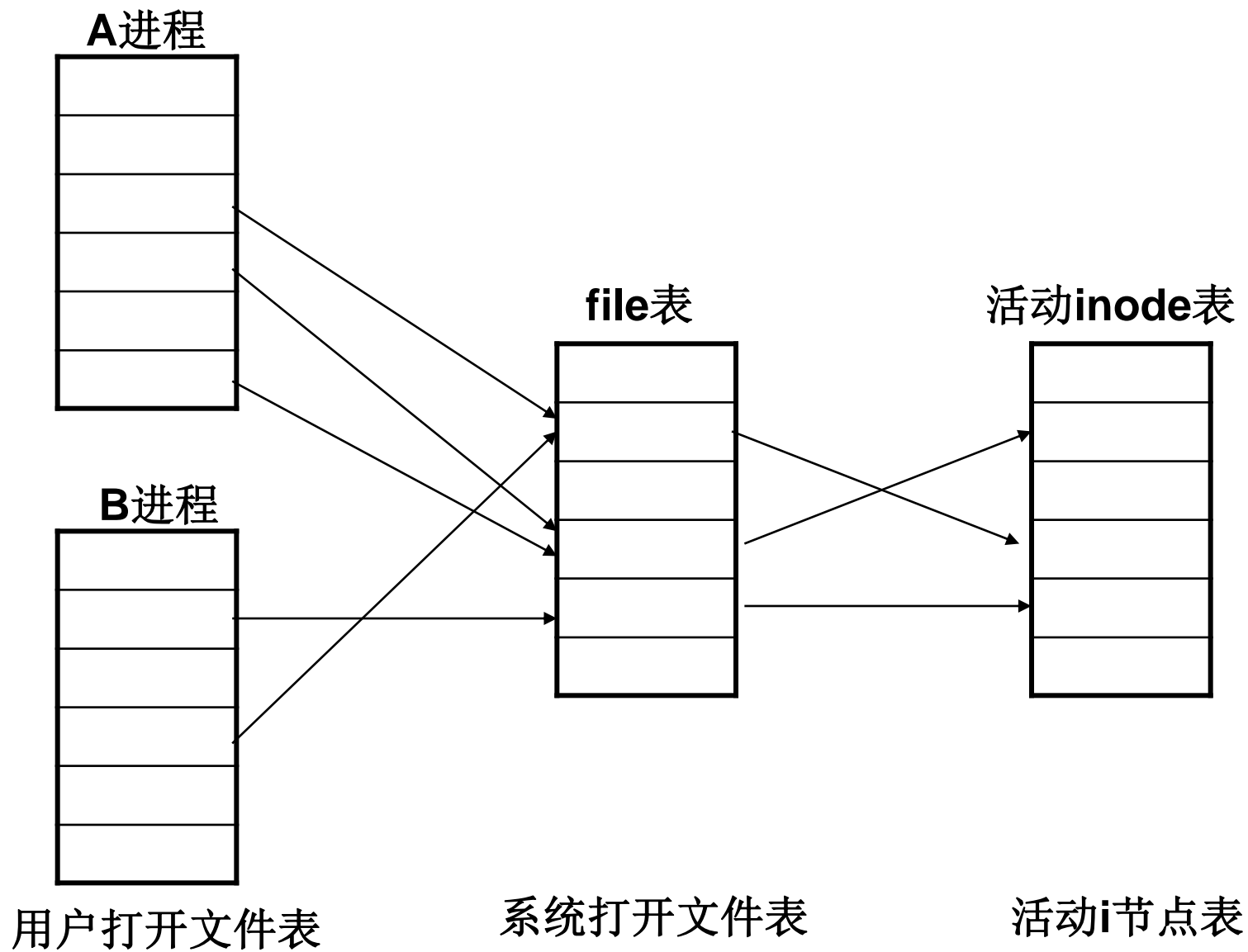
这个**u\_ofile**数组就是该进程的**用户打开文件表**。

## 6. 系统打开文件表（**file**表）

系统打开文件表主要存放被打开文件的读写指针。

因为一个进程在一个时间片内可能读写不完所需内容，需要在下一个时间片继续从上一个时间片结束时的读写位置开始读写，故在进程生存期间应保持一个读写指针。

此外**file**表中还存放被打开文件的动态信息：如文件状态、引用计数（当前使用该文件的进程数）等。





## 为什么要单独设立一个**file**表来存放读写指针呢？

由于可能有多个进程要共享一个被打开文件的**inode**，而每个进程的读写指针都不相同，故不能放在**inode**表中。

另一方面，要使不同进程的打开文件指针（文件描述符）或同一进程的不同打开文件指针能够共享一个打开文件指针（协同操作），就不能把读写指针放进某一个进程的用户打开文件表中。

因此只能在用户打开文件表和活动**inode**表之外再建立一个系统打开文件表（**file**表）来存放读写指针。

# UNIX操作系统共享活动文件的方法:

在内存中某个活动文件的副本只有一个，不同的进程采用不同的指针指向这文件的副本。由于任一时刻只有一个进程在运行（[微观上看](#)），故该文件也只要求内存中有一个副本即可，只是各个进程有自己的读写指针而已。

这是在UNIX系统中共享文件（包括用户文件和系统文件）的主要方法。对其它资源的共享采用的是与之相似的另外几种方法。

## 2.2.2 进程

### 相关概念：

#### 映像——

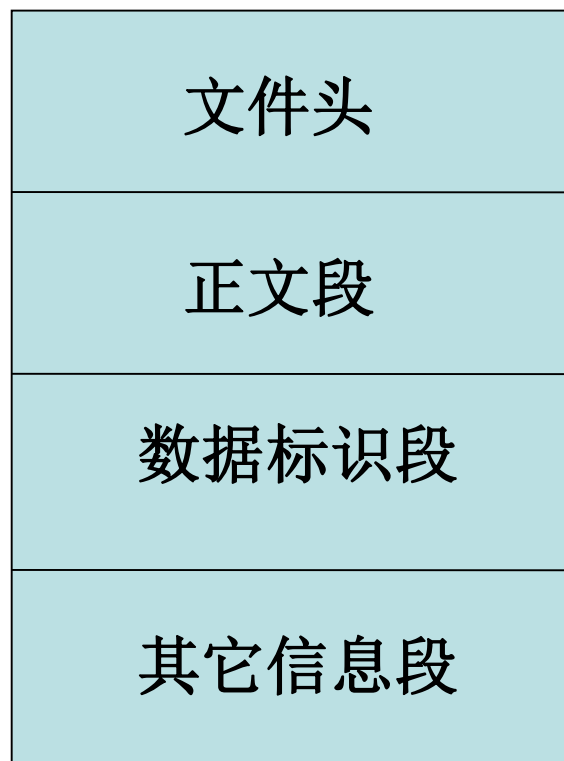
程序以及与动态执行该程序有关的各种信息的集合(类似于历史档案)。它包括存储器映象、通用寄存器映象，地址映射空间、打开文件状态等。

#### 进程——

对映像的执行。对映像的执行也就是一个程序在虚拟机上动态执行的过程。

## 1. 可执行文件的构成：

进程是可执行文件的一次执行实例，高级语言程序经过编译或汇编语言程序经过汇编后所产生的、缺省名为**a.out**的可执行文件的结构包括图示四个部分：



## 文件头——

- 文件的幻数（**magic number**）
- 编译器的版本号
- 机器类型
- 正文段、数据标识段、其它信息段的大小
- 程序入口点

## 正文段——

程序的功能代码

## 数据标识段——

标识未初始化的数据要占用的空间大小

## 其它信息段——

主要用于存放符号表

## 2. 程序的执行

一个进程在执行系统调用**exec**时，把可执行文件装入本进程的三个区域中：

**正文区**：对应可执行文件的正文段

**数据区**：对应可执行文件的数据标识段

**堆栈区**：新建立的进程工作区

堆栈主要用于传递参数，保护现场，存放返回地址以及为局部动态变量提供存储区。

进程在核心态下运行时的工作区为核心栈，在用户态下运行时的工作区为用户栈。核心栈和用户栈不能交叉使用。

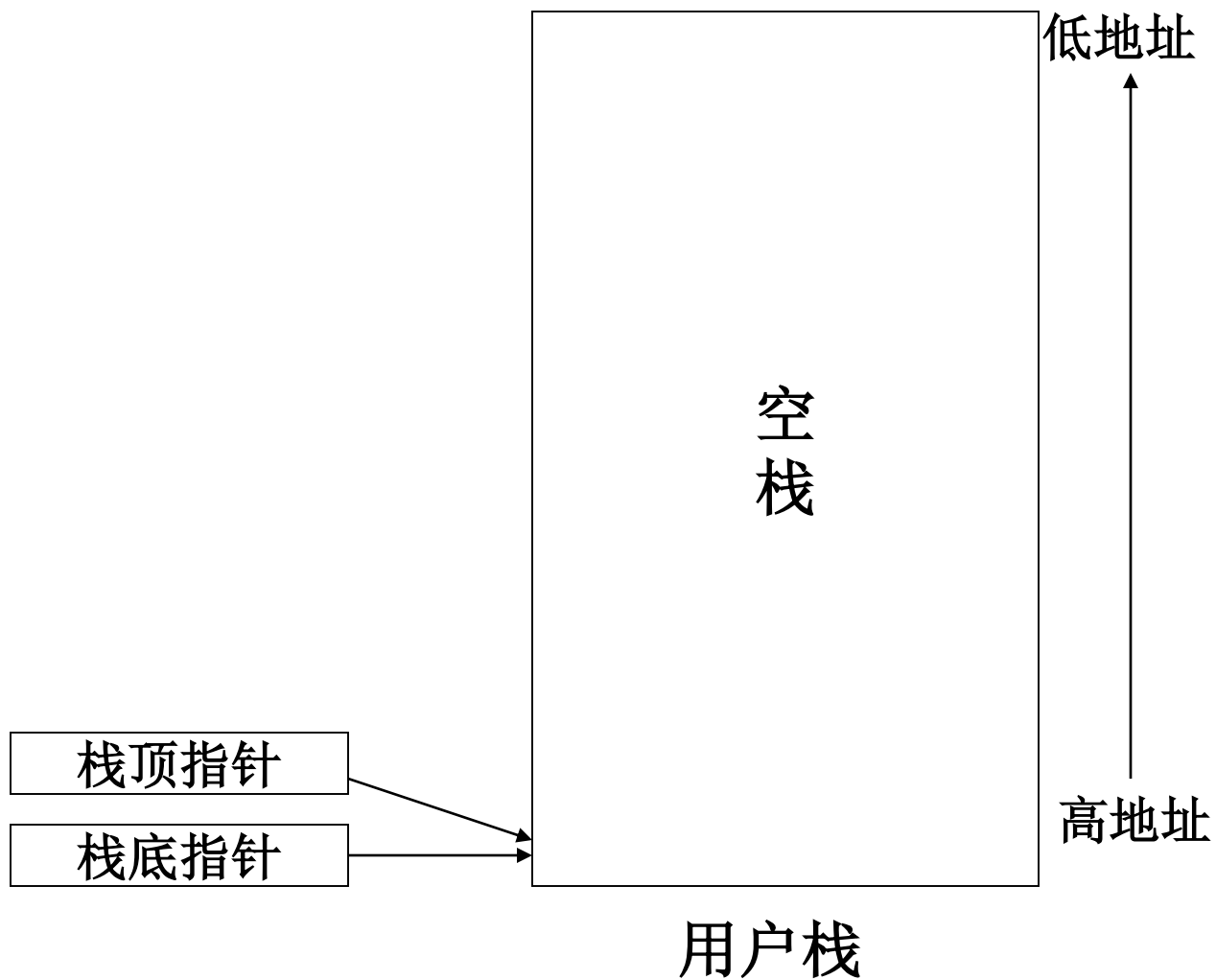
堆栈使用举例：如下程序在主程序中调用函数，并进行参数传递：

```
main (int argc, char *argv[ ])
```

```
{  
    char buf[1024];  
    int number;  
    ...  
    readfile (buf, number);  
    ...  
}
```

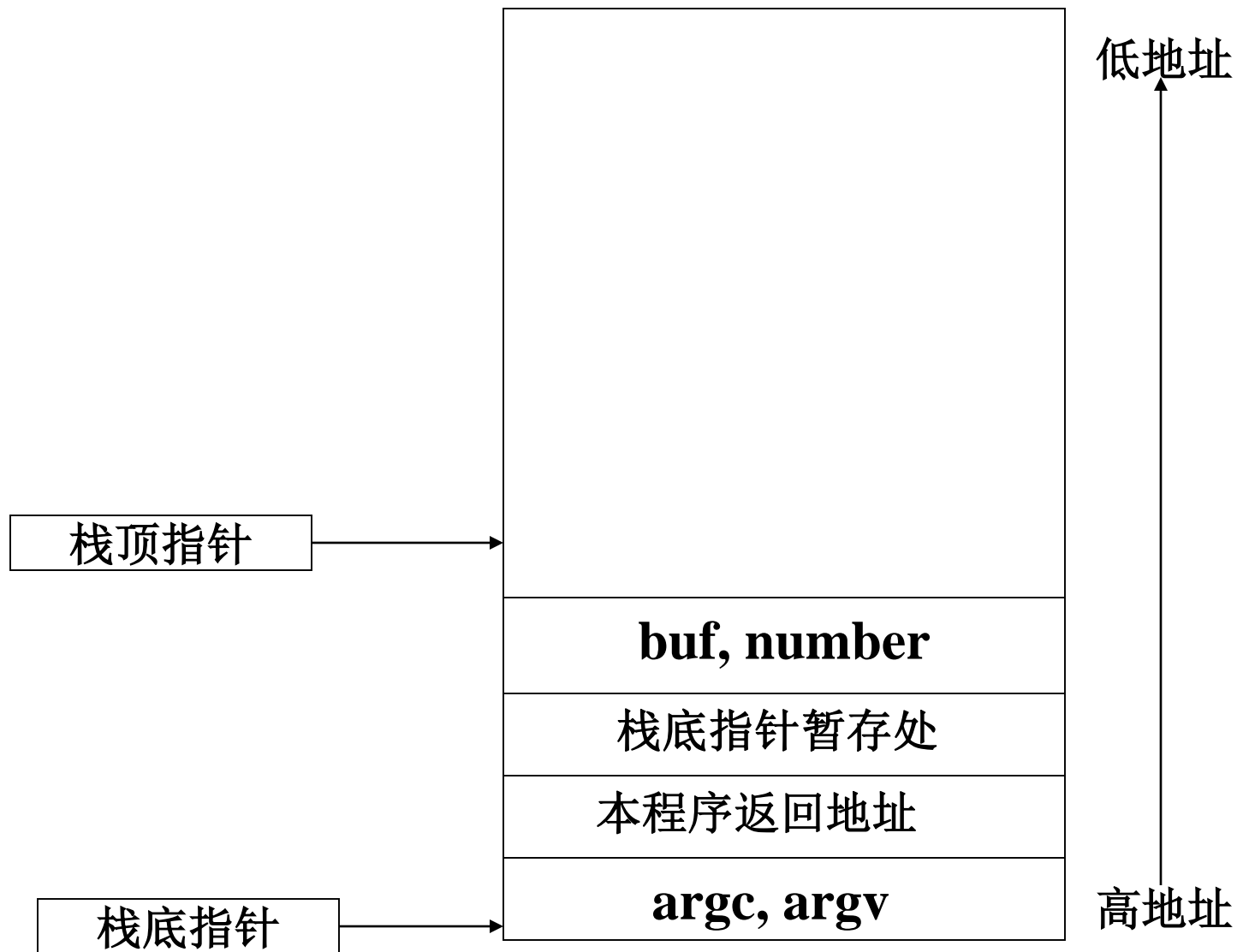
```
readfile (char buffer[ ], int line)
```

```
{  
    char *pointer;  
    int temp;  
    ...  
}
```

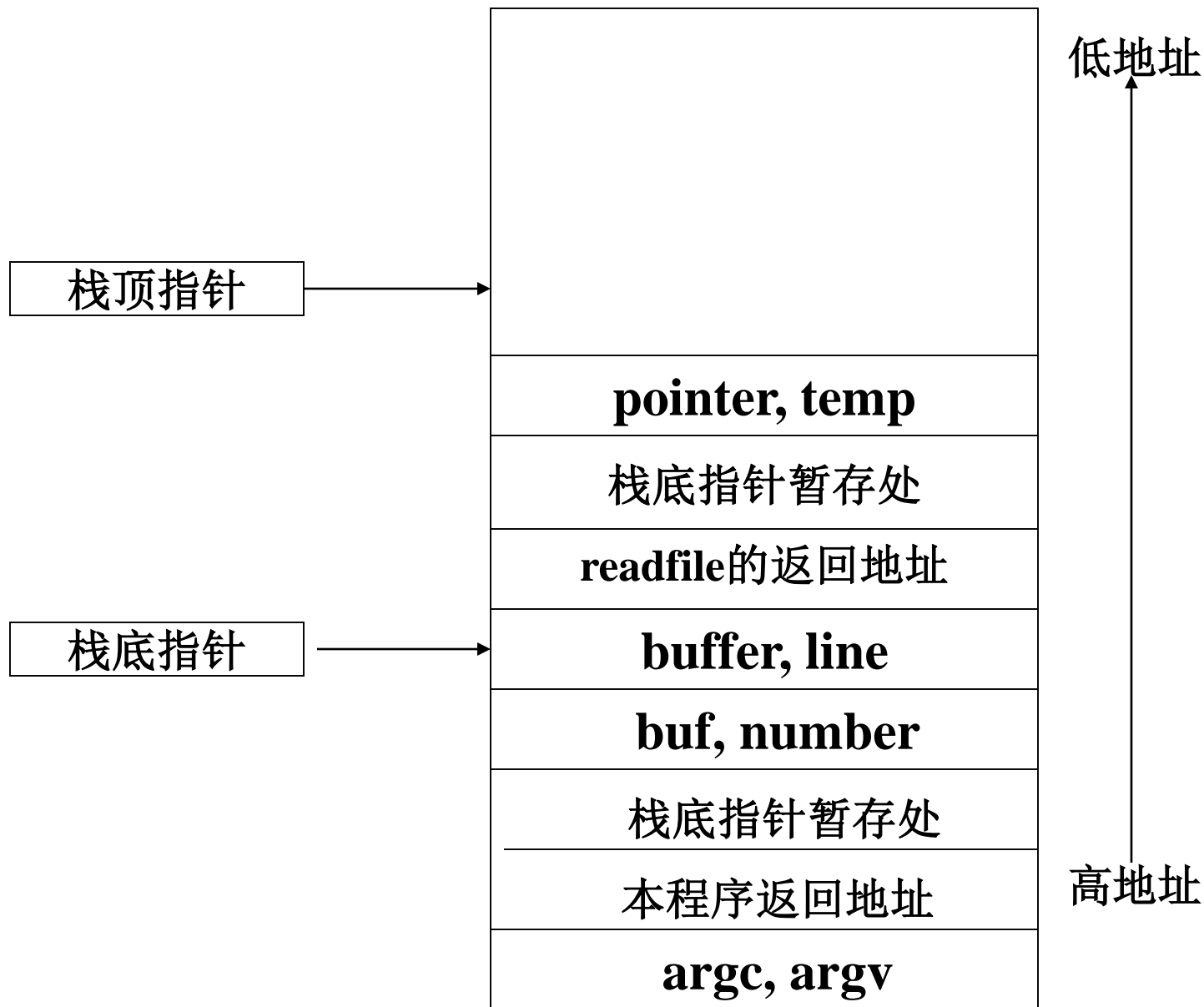


进入主程序时的堆栈状况





调用main()时



调用readfile时

### 3. 进程的标识

进程由其进程标识号**PID**来识别。

#### 0#进程

是由机器上电时“**手工**”创建的，调用**fork**创建了1#进程后，成为对换进程（**swap**）。

#### 1#进程

**init**进程，由它来创建系统初始化过程中所需的其它所有的进程。

#### 父进程

调用**fork**系统调用的进程

#### 子进程

由系统调用**fork**产生的进程

除0#进程外，其它所有进程都是另一个进程调用**fork**后产生的。

## 4. 进程状态及状态转换

### ①运行状态

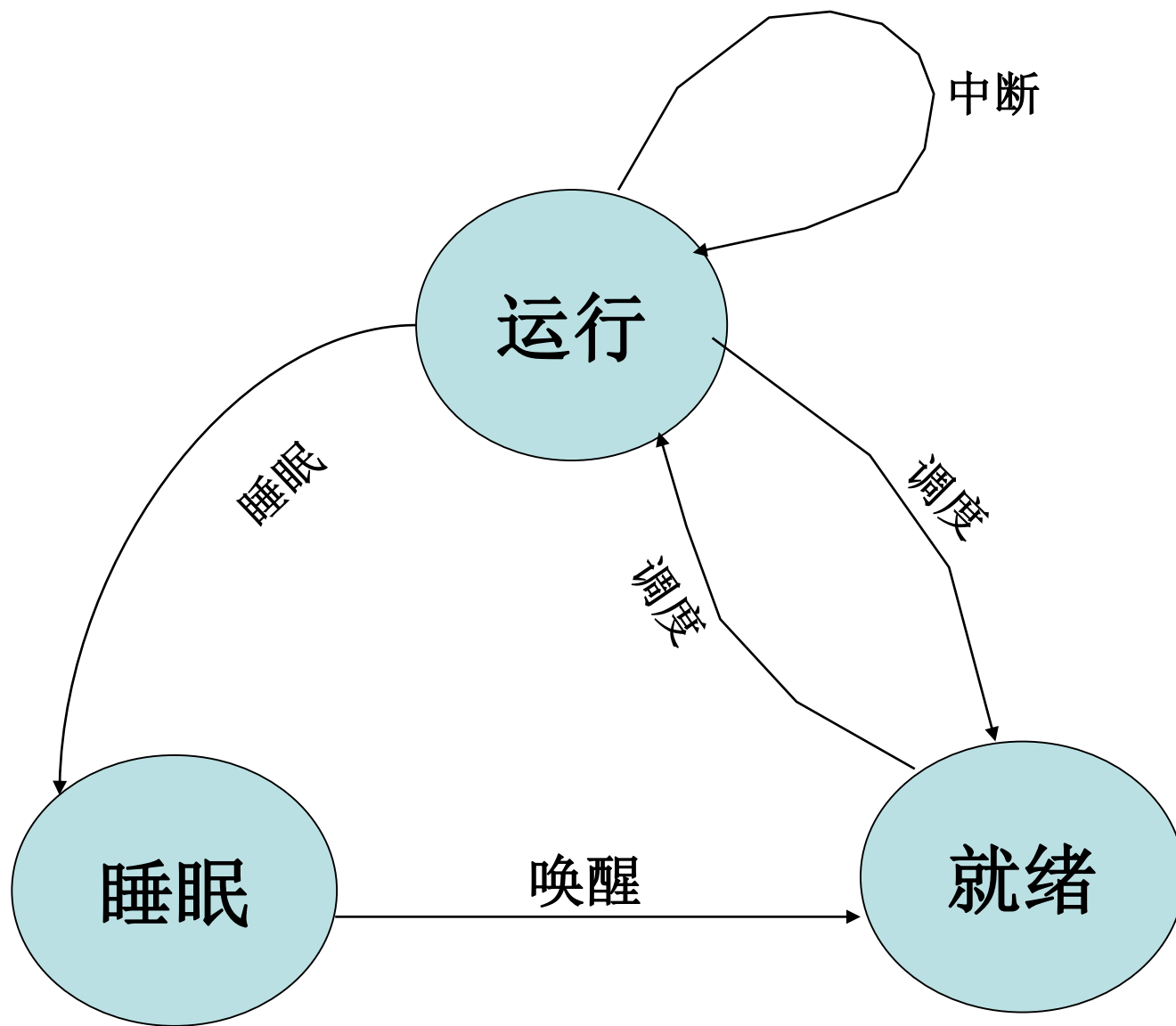
此时进程正在占用处理机，进程的全部映像驻在内存中。

### ②就绪状态

此时进程基本具备了运行条件，正在等待使用处理机。

### ③睡眠状态

进程不具备运行条件，需等待某种事件的发生，无法继续执行下去。



## 5. 在UNIX环境下，进程有如下特征：

- ① 每个进程在核心进程表（**proc**数组）都占有一项，在其中保留相应的状态信息。
- ② 每个进程都有一个“每进程数据区（**per process data area---ppda**）”保留相应进程更多的信息和核心栈；
- ③ 处理机的全部工作就是在某个时候执行某个进程
- ④ 一个进程可生成或消灭另一进程
- ⑤ 一个进程中可申请并占有资源
- ⑥ 一个进程只沿着一个特定的指令序列运行，不会跳转到另一个进程的指令序列中去，也不能访问别的进程的数据和堆栈。（抗病毒传播的重要原因之一）

# 第三章 数据缓冲区高速缓冲

## 硬件缓存（**cache**）

由一种高速寄存器（**register**）组成，主要解决**CPU**与**RAM**之间的速度差问题。

## 数据缓冲区高速缓冲（**buffer**）

由软件实现的解决文件系统和物理硬盘之间的数据同步的一种方法。

数据缓冲区高速缓冲是**UNIX**特有的对数据并发访问的一种控制机制。

## 问题的提出：

- 1、磁盘机械运行速度大大低于处理机的运行速度；
- 2、多进程并发运行，少量的磁盘（通道）I/O成为瓶颈；
- 3、数据访问的随机性，磁盘忙闲不均



## 解决办法:

1、建立一个被称为数据缓冲区高速缓冲（简称高速缓冲）的内部数据缓冲区池（**buffer pool**）来存放要用的数据；

### 2、写数据时

把数据尽量多地尽量长时间地保存在缓冲池中

延迟写出到磁盘上

以备后续进程使用

### 3、读数据时

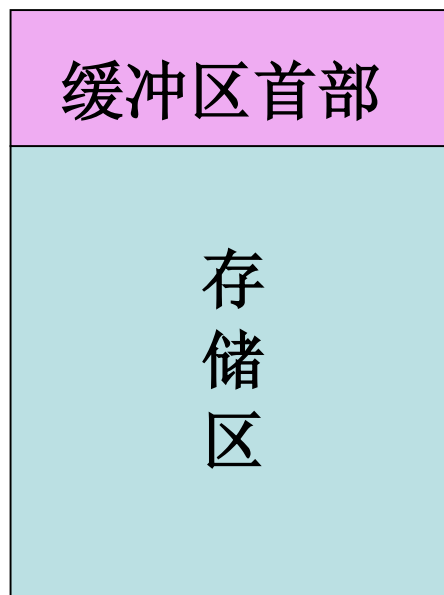
先在缓冲池中查找已有的数据

如没有，再从磁盘读取，并保存在缓冲池中

事先预读数据到缓冲池中

## 3.1 缓冲区及缓冲区首部

缓冲区池由若干个缓冲区组成，每一个缓冲区又由两部分组成：一个实际存放数据的存储区和一个标识该缓冲区的缓冲区首部。



因为缓冲区首部与数据存储区之间有一一对应的关系，所以通常把两者统称为缓冲区。

缓冲区是缓冲区池中数据存储的基本单位。

## 缓冲区首部的定义:

```
struct buf {
```

缓冲区标志

缓冲区链接指针

空闲缓冲区链表指针

设备号

块号

```
union{
```

数据块

超级块

柱面块

i节点块

```
}b_un
```

其它控制信息

```
}
```

标识缓冲区状态

向前向后串成链表

联结空闲缓冲区

标识缓冲区

缓冲区中的数据类型

## 3.2 缓冲池的结构

### 1、最近最少使用（LRU）算法：

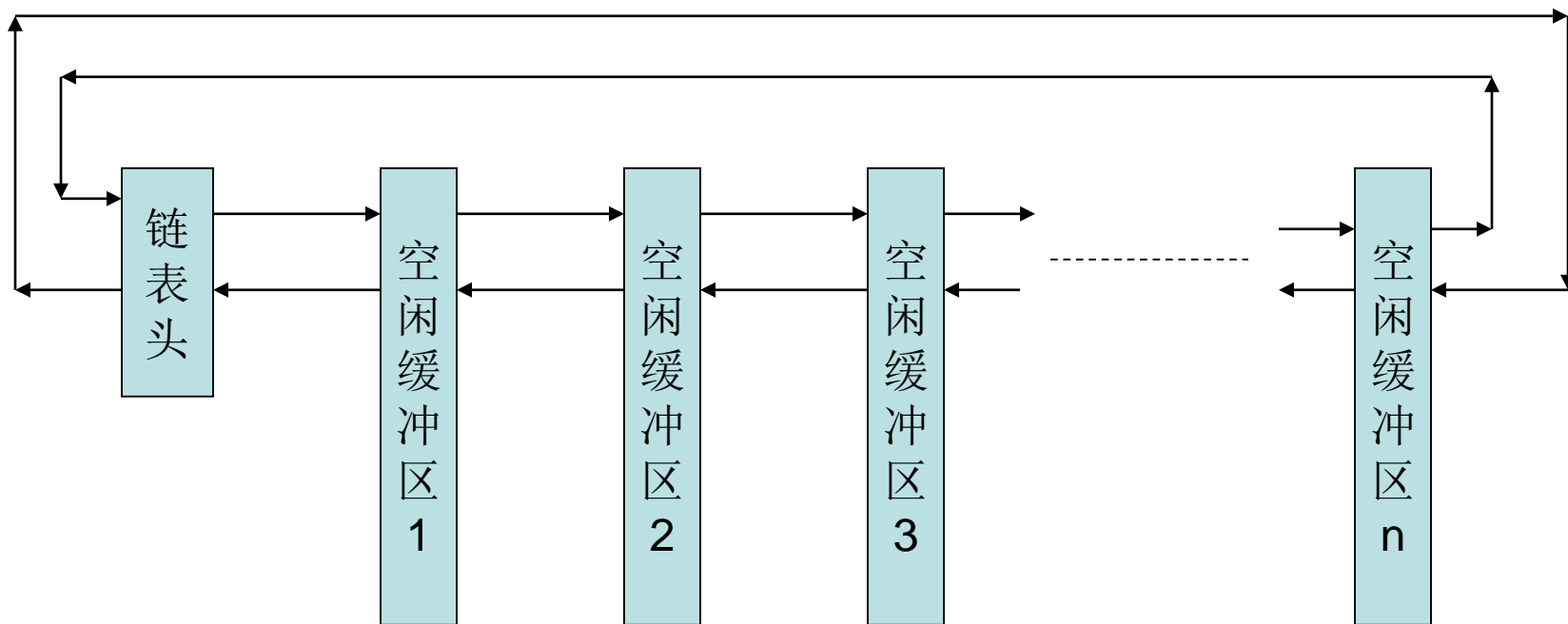
- ① 程序设计采用模块化和层次化结构，尽量避免使用**goto**语句，程序跳转少，适应“流水线（**pipeline**）”体系结构的系统；
- ② 特定时间段内，程序在一个相对集中空间（代码段）内运行，涉及的数据（广义的：文件名、变量、指针和数组等）的个数相对较少；
- ③ 当前使用过的数据，马上还要使用的可能性最大，较长时间未用过的数据，即将使用的可能性最小。

## 2、缓冲池设计基本原则：

- ① 存放有刚使用过的数据尽量长时间地保留在内存中，以便马上还要使用时能在内存中找到；
- ② 需要腾出内存空间时，把很久都未使用过（即最近最少使用）的数据交换到磁盘上去。这些数据马上还要使用的可能性最小。

### 3、空闲缓冲区链表

核心维护了一个空闲缓冲区链表，它按照最近被使用的先后次序排列。空闲链表是一个以空闲缓冲区链表头开始的“**双向循环链表**”。链表的开始和结束都以链表头为标志。



## 4、空闲缓冲区链表操作

### ① 取用任意空闲缓冲区

从空闲缓冲区链表的表头位置取下一个空闲缓冲区，后面的空闲缓冲区依次向前移动。

### ② 释放一个空闲缓冲区

把这个装有数据的空闲缓冲区附加到空闲链表的链尾。只有当该空闲缓冲区所装数据出错时才挂到链头。

### ③ 取用装有指定内容的空闲缓冲区

从链表头开始查找，找到后取下使用，用完后放到链尾。

当系统不断从链头取用空闲缓冲区，又把使用过的（装有数据的）缓冲区挂到链尾，一个装有有效数据的缓冲区就会逐渐向链表头移动。在链表头位置的就是“最近最少使用”的空闲缓冲区。

## 5、空闲缓冲区分类

系统中共设置了四个空闲缓冲区链表，根据缓冲区的不同用途而把它的放入不同的空闲缓冲区链表中。避免在取用空闲缓冲区时，逐个判断缓冲区中的内容。这四个空闲链表是：

**0#空闲缓冲区链表**——存放文件系统超级块

**1#空闲缓冲区链表**——存放通常使用的数据块

**2#空闲缓冲区链表**——存放延迟写、无效数据或错误内容

**3#空闲缓冲区链表**——存放没有对应存储空间的缓冲区首部

如果某种类型的空闲缓冲区不够用时，核心也从其它空闲缓冲区链表中取用空闲缓冲区。



## 6、缓冲区设置

当核心需有一个空闲缓冲区时，它根据要装入的数据类型，从相应的空闲缓冲区链表的表头位置取下一个空闲缓冲区，装入一个磁盘数据块；

根据该数据块所对应的设备号和块号数据对计算其 **hashno**（散列、杂凑）值，根据其 **hashno** 的值放入到相应 **hash** 链表的链头。

$$\text{hashno} = ((\text{diskno} + \text{blkno}) / \text{RND}) \% \text{BUFHSZ}$$

**diskno:** 设备号

**blkno:** 块号

**BUFHSZ:** 最大**hash**值，通常为**63**。

**RND:** 随机数，其值为： **$\text{RND} = \text{MAXBSIZE} / \text{DEV\_BSIZE}$**

**MAXBSIZE:** 最大文件系统块的大小

**DEV\_BSIZE:** 物理设备块大小

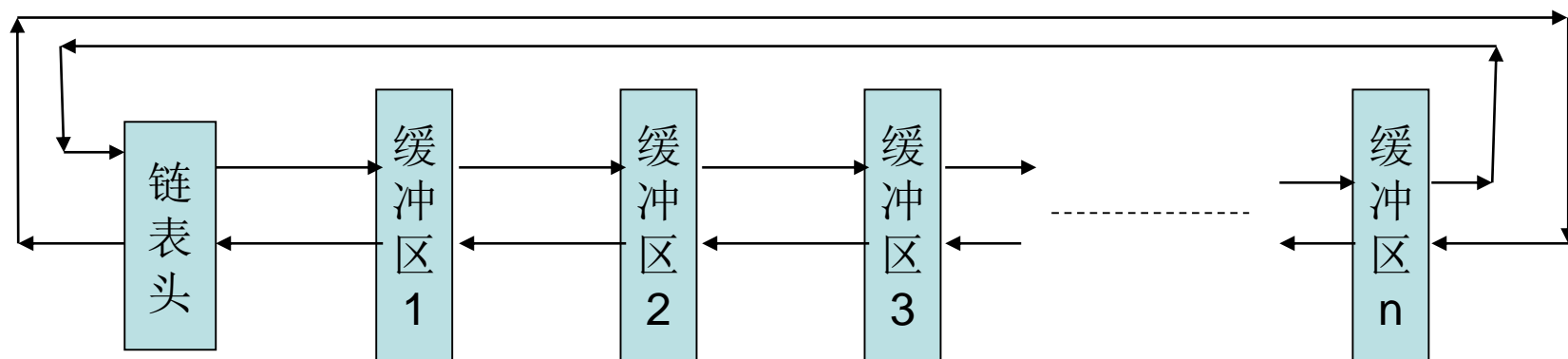
**hashno** 的取值范围： **0 ~ 62**

## 7、缓冲池的结构

具有相同 **hashno** 的缓冲区链接在同一个**hash**链表中，因此系统中共有 **63** 个**hash** 链表，分别链接 **hashno** 为 **0 ~ 62** 的缓冲区。

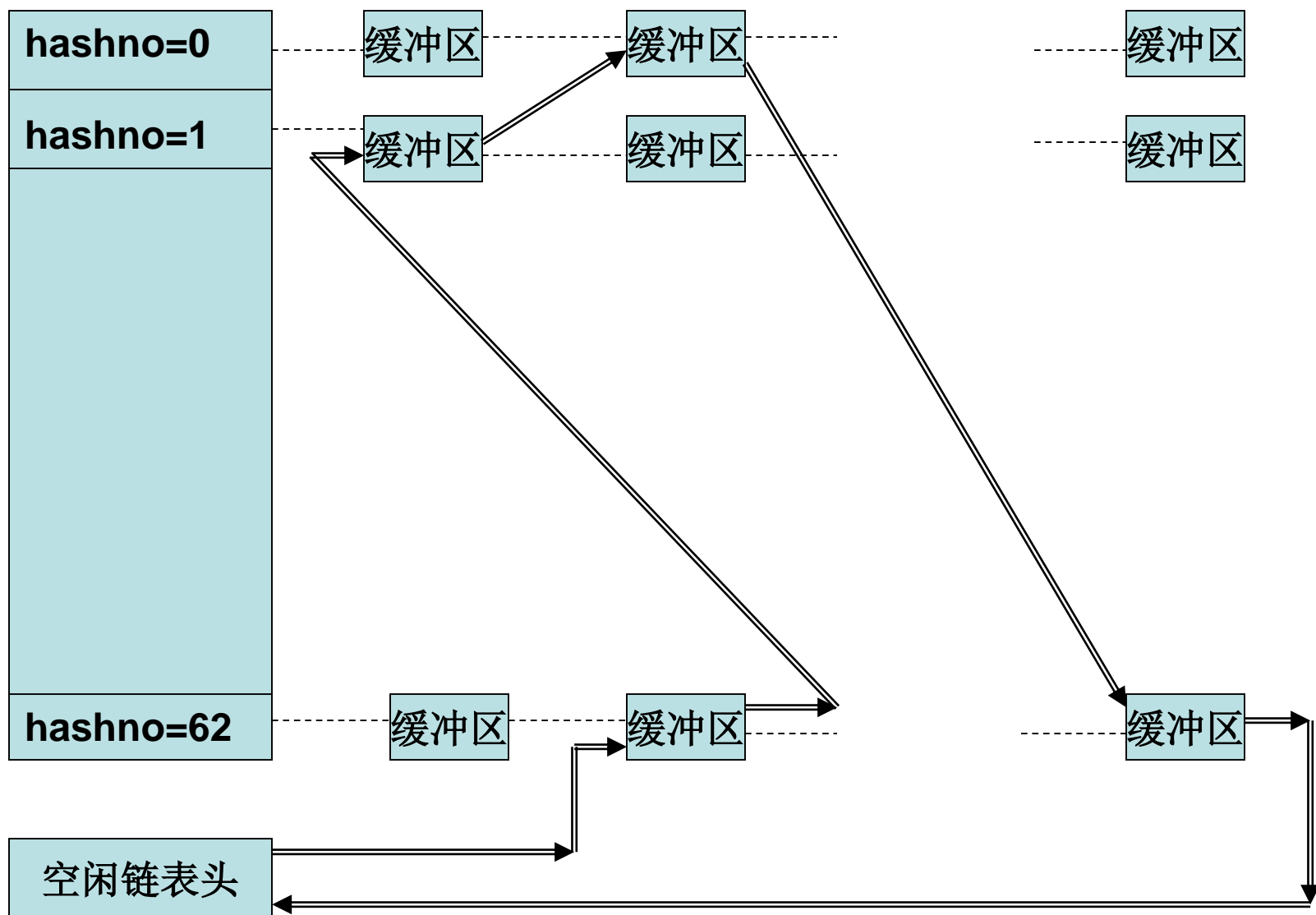
每一个 **hash** 链表都是一个由链表头指向的**双向循环链表**，查找某一个指定 **hashno** 值的缓冲区时，也是从相应的**hash**链表的表头位置开始向表尾方向进行查找。

这 **63** 个 **hash** 链表就构成了数据缓冲区高速缓冲的缓冲池，所有的缓冲区都存放在缓冲池中的某一个链表中。



hash 链表的结构

## Hash链表头



缓冲池的结构

## 8、缓冲区的使用

如果要找特定缓冲区，根据**hashno**从相应的**hash**链表的表头处开始逐个向后查找；

如果找到，则直接取用，并将其移动到**hash**链的链头；

如果未找到，则从相应的空闲缓冲区链表的表头处取下一个空闲缓冲区，填入相应数据，重新计算其**hashno**，并放到新的**hash**链表的表头；

释放缓冲区时，将该缓冲区仍保留在原**hash**队列中，同时挂接到空闲缓冲区链表的表尾。（同时在两个队列中）

申请缓冲区的两个途径：

要指定缓冲区 —— 在**hash**链表中查找

要空闲缓冲区 —— 在空闲链表中查找

## 9、进一步说明

一个缓冲区只有当它是空闲状态时，它才同时处在**hash**链表和空闲链表中。如果不空闲，则它只能处在某一个**hash**链表中。

在空闲缓冲区链表中的缓冲区一定在某个**hash**链表中；在**hash**链表中的缓冲区不一定在空闲链中。不存在脱离**hash**链表的另一个空闲的缓冲区链表。

缓冲池中的缓冲区**个数**是固定不变的，每个缓冲区在不同时刻存放着不同的磁盘数据块，具有不同的**hash**值，因此处在不同的**hash**链表中。

缓冲区中的数据与某个磁盘数据块一一对应，这种对应有两个特点：

- ① 一个磁盘数据块在缓冲池中最多只能有一个副本；
- ② 缓冲区与数据块的对应是动态的，**LRU**数据块将被释放。

## 3.3 缓冲区的检索算法

在**UNIX**文件系统中的下层，即直接与逻辑存储设备联系的部分，包含如下基本算法：

<b>getblk</b>	申请一个缓冲区
<b>brelease</b>	释放一个缓冲区
<b>bread</b>	读一个磁盘块
<b>breada</b>	读一个磁盘块，预读另一个磁盘块
<b>bwrite</b>	写磁盘块

# 1、申请一个缓冲区算法 `getblk`

根据缓冲池的结构，核心申请一个缓冲区分配个磁盘块时，可能出现的**五种典型状况**：

- ① 该块已在**hash**队列中，并且缓冲区是空闲的；
- ② **hash**队列中找不到该块，需从空闲链表中分配一个缓冲区；
- ③ **hash**队列中找不到该块，在从空闲链表中分配一个缓冲区时，发现该空闲缓冲区标记有“延迟写”，核心必须写出缓冲区内内容到磁盘上，再重新分配一个空闲缓冲区；
- ④ **hash**队列中找不到该块，并且空闲链表已空；
- ⑤ 该块已在**hash**队列中，但该缓冲区目前状态为“忙”。

算法 **getblk**

输入：文件系统号  
块号

输出：现在能被磁盘块使用的上了锁的缓冲区

```
{
    while (没有找到缓冲区)
    {
        if (块在散列队列中)
        {
            if (块忙)                                /* 第五种情况 */
            {
                sleep (等候“缓冲区变为空闲”事件);
                continue;                            /* 回到while循环 */
            }
            为缓冲区标记上“忙”；                    /* 第一种情况 */
            从空闲链表上摘下缓冲区；
            return(缓冲区);
        }
        else /* 块不在散列队列中 */
        {
            if (空闲链表上无缓冲区)                  /* 第四种情况 */
            {
                sleep (等待“任何缓冲区变为空闲”事件);
                continue;                            /* 回到while循环 */
            }
            从空闲链表上取下缓冲区；
            if (缓冲区标记着延迟写)                  /* 第三种情况 */
            {
                把缓冲区异步写到磁盘上；
                continue;                            /* 回到while循环 */
            }
            /* 第二种情况 —— 找到了一个空闲缓冲区 */
            从旧散列队列中摘下缓冲区；
            把缓冲区投入到新的散列队列中去；
            return (缓冲区);
        }
    }
}
```



## 2、 释放一个缓冲区算法 brelse

- 唤醒等待缓冲区的所有进程
- 提高处理机的执行级别以封锁同级或低级的中断
- 将该缓冲区放到空闲队列的尾部（缓冲区有效）或头部（缓冲区无效）
- 降低处理机的执行级别以开放中断

算法    brelse

输入： 上锁状态的缓冲区

输出： 无

{

    唤醒正在等待“无论哪个缓冲区变为空闲”这一事件的所有进程；

    唤醒正在等待“这个缓冲区变为空闲”这一事件的所有进程；

    提高处理机执行级别以封锁中断；

    if （缓冲区内容有效且缓冲区非“旧”）

        将缓冲区送入空闲链表尾部；

    else

        将缓冲区送入空闲链表头部；

    降低处理机执行级别以允许中断；

    给缓冲区解锁；

}

### 3、读一个磁盘块 **bread**

- 由 **getblk** 算法申请一个可用的缓冲区
- 如果缓冲区中的内容有效，则直接返回该缓冲区
- 如果缓冲区中的内容无效，则启动磁盘去读所需的数据块
- 等待磁盘操作完成后返回

## 算法 **bread**

输入：文件系统号

输出：含有数据的缓冲区

{

    得到该块的缓冲区（算法**getblk**）；

**if**（缓冲区数据有效）

**return**（缓冲区）；

    启动磁盘读；

**sleep**（等待“读盘完成”事件）；

**return**（缓冲区）；

}

## 4、 读一个磁盘块并预读另一个磁盘块 breada

### 预读的前提：

程序是在一个有限的空间内运行，程序对数据的访问是可预见的。

### 预读的命中率：

不一定达到**100%**，但良好的系统结构和算法可使命中率达到较高的水平。

### 预读的结果：

放在缓冲池内，以免需要的时候再去启动磁盘读数据块。

算法 **breada**

输入：（1）立即读的文件系统块号

（2）异步读的文件系统块号

输出：含有立即读的数据的缓冲区

```
{  
    if（第一块不在高速缓冲中）  
    {  
        为第一块获得缓冲区（getblk）；  
        if（缓冲区内容无效）  
            启动磁盘读；  
    }  
    if（第二块不在高速缓冲中）  
    {  
        为第二块获得缓冲区（getblk）；  
        if（缓冲区数据有效）  
            释放缓冲区（brelse）；  
        else  
            启动磁盘读；  
    }  
    if（第一块本来就在高速缓冲中）  
    {  
        读第一块（bread）；  
        return（缓冲区）；  
    }  
    sleep（第一个缓冲区包含有效数据的事件）；  
    return（缓冲区）；  
}
```

## 5、写磁盘块 `bwrite`

启动磁盘驱动程序的写操作

如果是“同步写”，则本进程睡眠等待磁盘写操作的完成，磁盘写操作完成后，中断唤醒本进程，本进程释放该缓冲区并返回；

如果是“异步写”，则无需等待磁盘写操作的完成，将缓冲区放到空闲链表的表头，以便随后某个进程申请空闲缓冲区时，将其写到磁盘上去。本进程不再关心该缓冲区实际被写出的时间和结果，而直接返回去作其它事情。

事实上无论是同步写还是异步写，其根本区别在于本进程是否等待磁盘驱动程序完成操作后所发出的中断信号。

算法 **bwrite**

输入：缓冲区

输出：无

```
{  
    启动磁盘写;  
    if (I/O同步)  
    {  
        sleep (等待 “I/O完成” 事件);  
        释放缓冲区 (brelse);  
    }  
    else if (缓冲区标记着延迟写)  
        为缓冲区做标记以放到空闲缓冲区链表头部;  
}
```



## 3.3 数据缓冲区高速缓冲的优缺点

### 优点：

- ✓ 提供了对磁盘块的统一的存取方法
- ✓ 消除了用户对用户缓冲区中数据的特殊对齐需要
- ✓ 减少了磁盘访问的次数，提高了系统的整体I/O效率
- ✓ 有助于保持文件系统的完整性

### 缺点：

- ✓ 数据未及时写盘而带来的风险
- ✓ 额外的数据拷贝过程，大量数据传输时影响性能

# 第四章 文件和文件系统的内部结构

现代**UNIX**的文件系统通常可由三大模块组成：

①本地文件系统（**UFS**）——**User File System**

②网络文件系统（**NFS**）——**Network File System**

③虚拟文件系统（**VFS**）——**Virtual File System**

# 本地文件系统（UFS）

是UNIX系统中的基本文件系统，它通常固定存放在本地机器的存贮设备上，任何一种结构形式的文件系统都必然会直接或间接地与某个本地文件系统相联系。

## 本地文件系统的构成

一个根文件系统 + 若干子文件系统所组成

### 根文件系统

存放本操作系统的最主要和最基本的部分  
可独立启动运行

系统起动后，根文件系统就不能卸下来

### 子文件系统

主要存放应用程序和用户文件

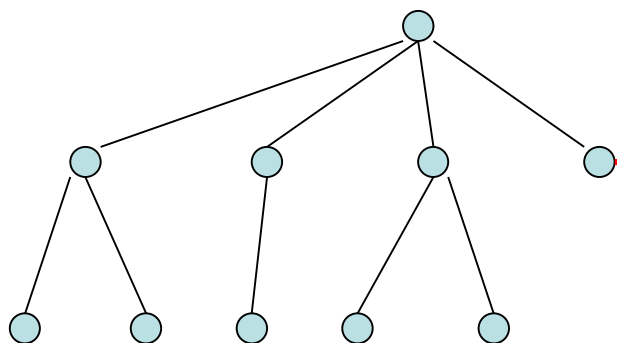
一般不能独立启动

系统运行过程中可随时安装和卸下

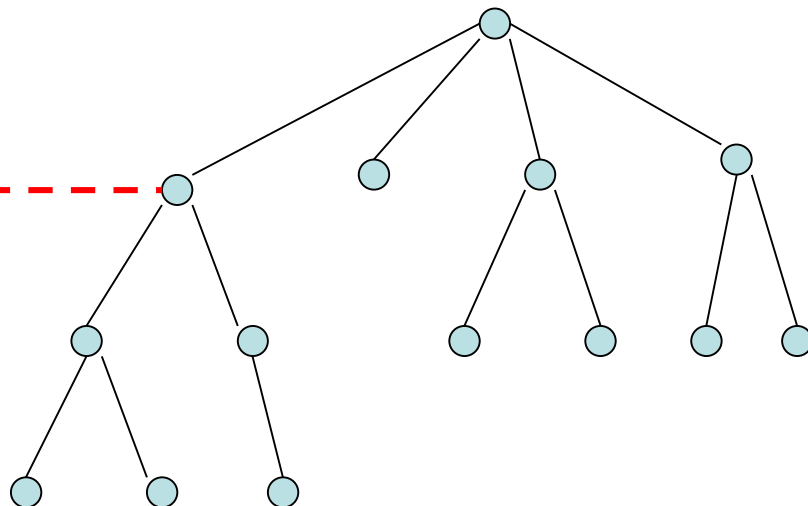
# 网络文件系统（NFS）

是本地机器上的文件系统和远地机器上的文件系统之间的介质，它管理和控制所有有关对远地文件的各种操作，给本地用户提供一个访问远地文件的使用方便的高层接口，避免用户直接涉及网络通讯方面的具体细节。

**A节点文件系统**



**B节点文件系统**

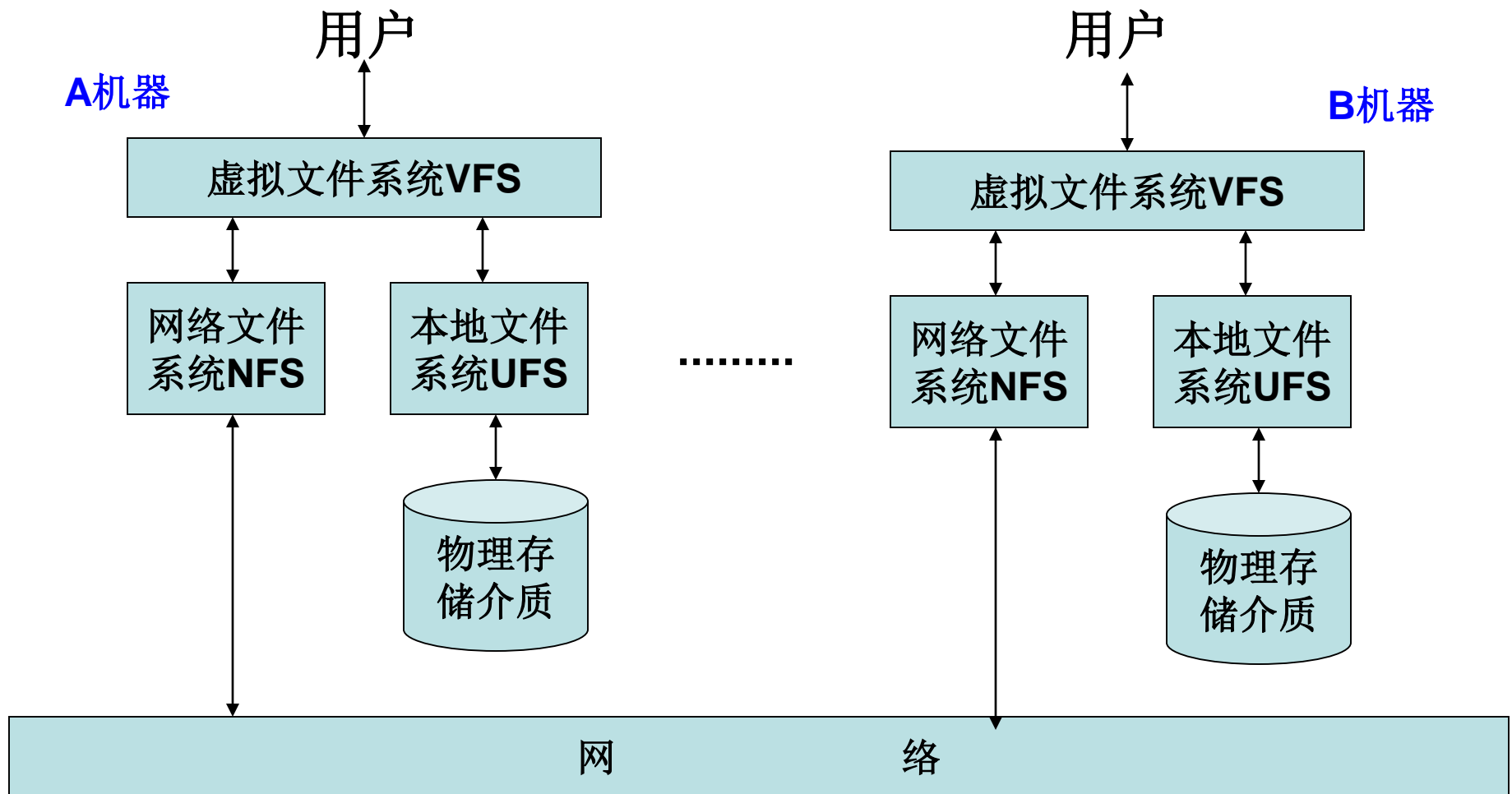


# 虚拟文件系统（VFS）

**VFS**是整个操作系统的用户界面，它给用户提供一个统一的文件系统使用接口，避免用户涉及各个子文件系统的特征部分。

用户感觉使用的是一个整体的，比本地机器上实际硬盘空间大得多的文件系统。

虚构文件系统接受来自用户的操作请求，根据该操作所访问的文件是存放在本地机器上，还是存放在远地机器上而分别把操作交给本地文件系统或网络文件系统；本地文件系统或网络文件系统（实际上再传给远地机器上的本地文件系统）进行相应的操作后，将结果返回到虚拟文件系统中再传回给用户。



基于虚拟文件系统的体系结构

# 4.1 文件系统结构

## 4.1.1 本地文件系统

### 1. 文件系统的存储结构

在**UNIX**系统中，一个物理磁盘通常被划分成一个或多个逻辑文件系统（简称文件系统或子文件系统），每个逻辑文件系统都被当作一个由逻辑设备号标识的逻辑设备。

**UNIX**的普通文件和目录文件就保存在这样的文件系统中。逻辑文件系统的存储结构可分为两类型：

**一级存储结构型**：常用于运行环境较小的文件系统中

**二级存储结构型**：常用于运行环境较大（特别是硬盘空间较大）的文件系统中

## ①、一级存储结构型

这种类型的逻辑文件系统由超级块、索引节点表块和数据区组成，（如果是根文件系统，就还包括引导块）。整个存储结构是一维的。

引导块	超级块	i节点表块	数据区
-----	-----	-------	-----

引导块： **boot**程序

超级块： **fs**结构，存放文件系统的静态参数

i 节点表块：磁盘**icommon**表

数据区： 各数据块

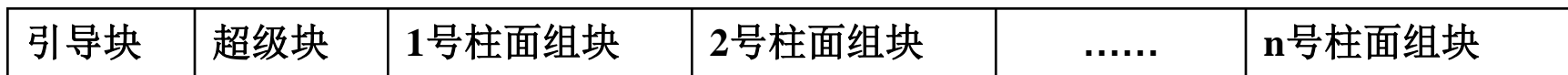


## ②、两级存储结构型

这种存储结构的文件系统由两级组成：第一级由超级块和若干个柱面组块（**cylinder group block**）所组成（如果是根文件系统则还包括引导块）。第二级（即柱面组块）又是由超级块拷贝块、柱面组信息块，*i*节点表块和数据区所组成。文件系统的存储结构是二维的。

目前大多数在**大存储环境**下运行的**UNIX**版本都采用这种存贮结构，其优点是能快速定位数据块。

### 第一级存储结构



### 第二级存储结构



## 超级块

是由**fs**定义的数据结构，用于存放文件系统的静态参数：

```
struct fs {  
    内存超级块链接指针  
    超级块的磁盘地址  
    柱面组块的位移量  
    最近修改时间  
    文件系统大小  
    文件系统块大小  
    柱面组数  
    柱面组大小  
    片大小  
    文件系统标识数  
    文件系统标志区  
    最近访问的柱面组号  
    确定分配算法的参数  
}
```

空气过滤片

主轴（马达  
电机与轴承  
在其下方）

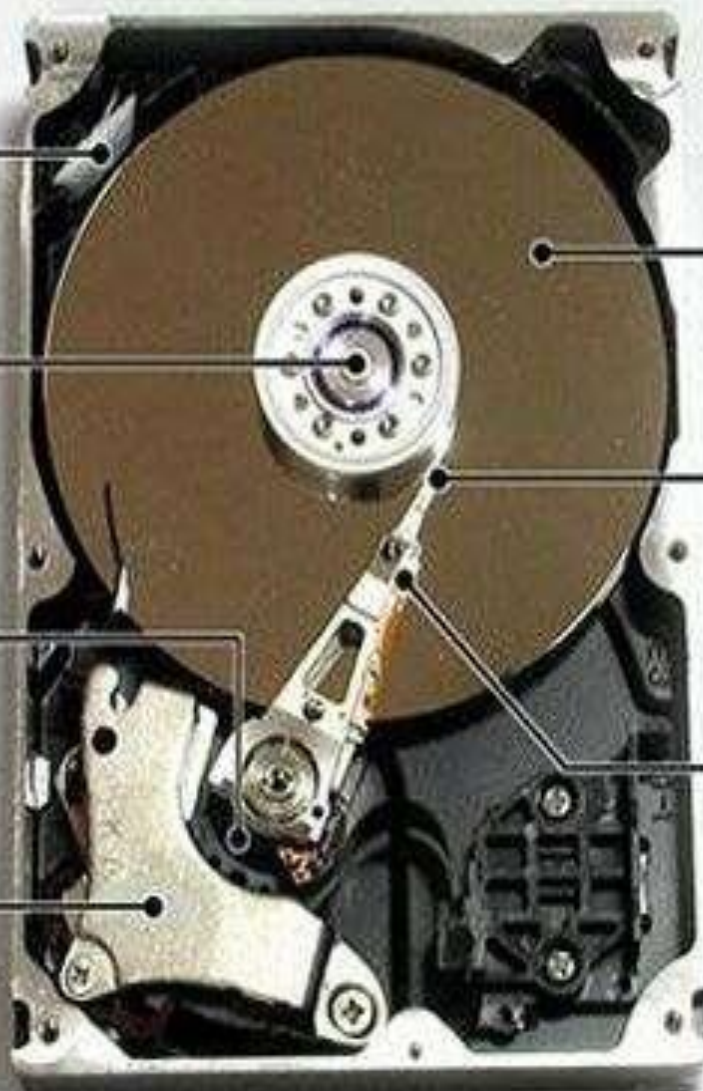
音圈马达

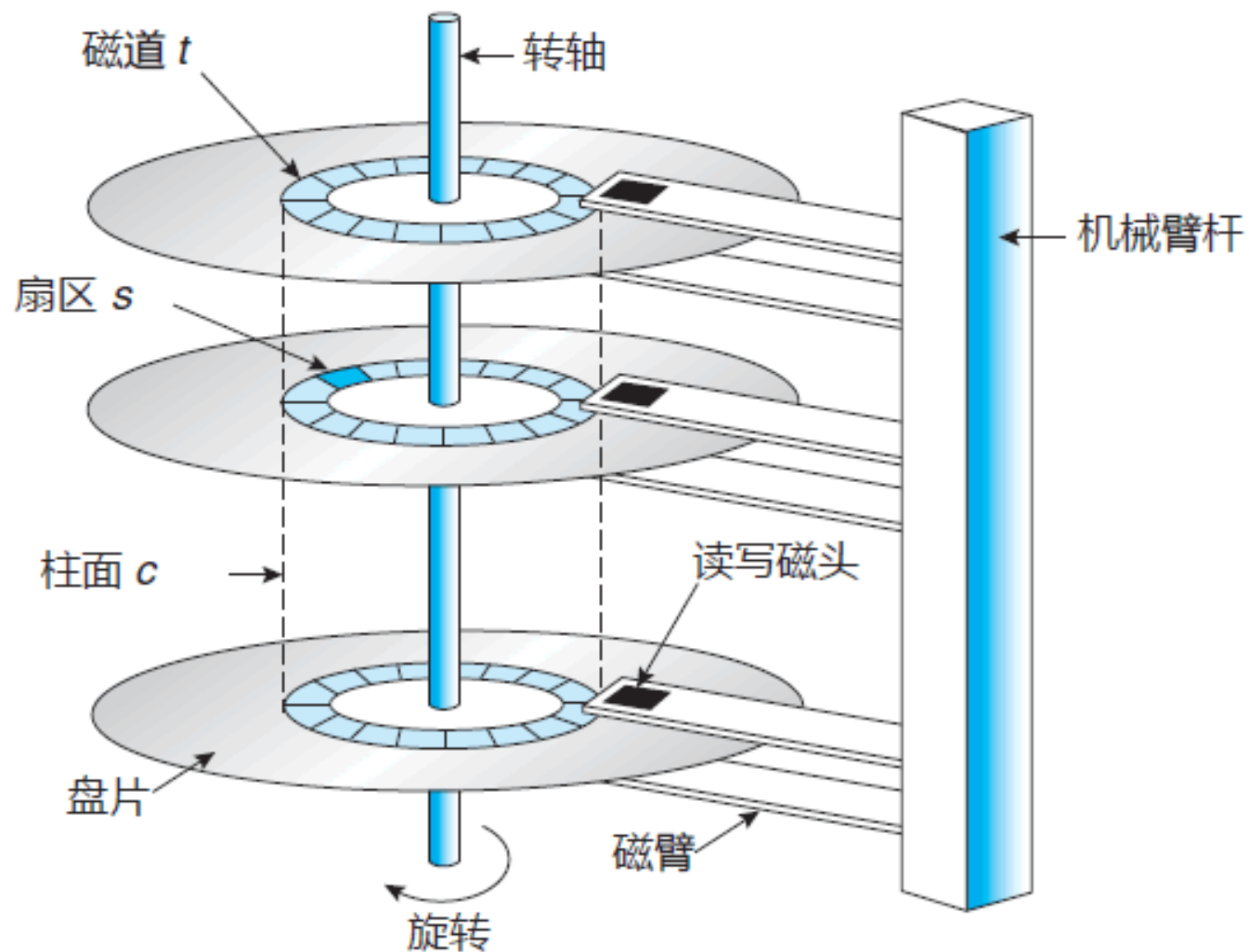
永磁铁

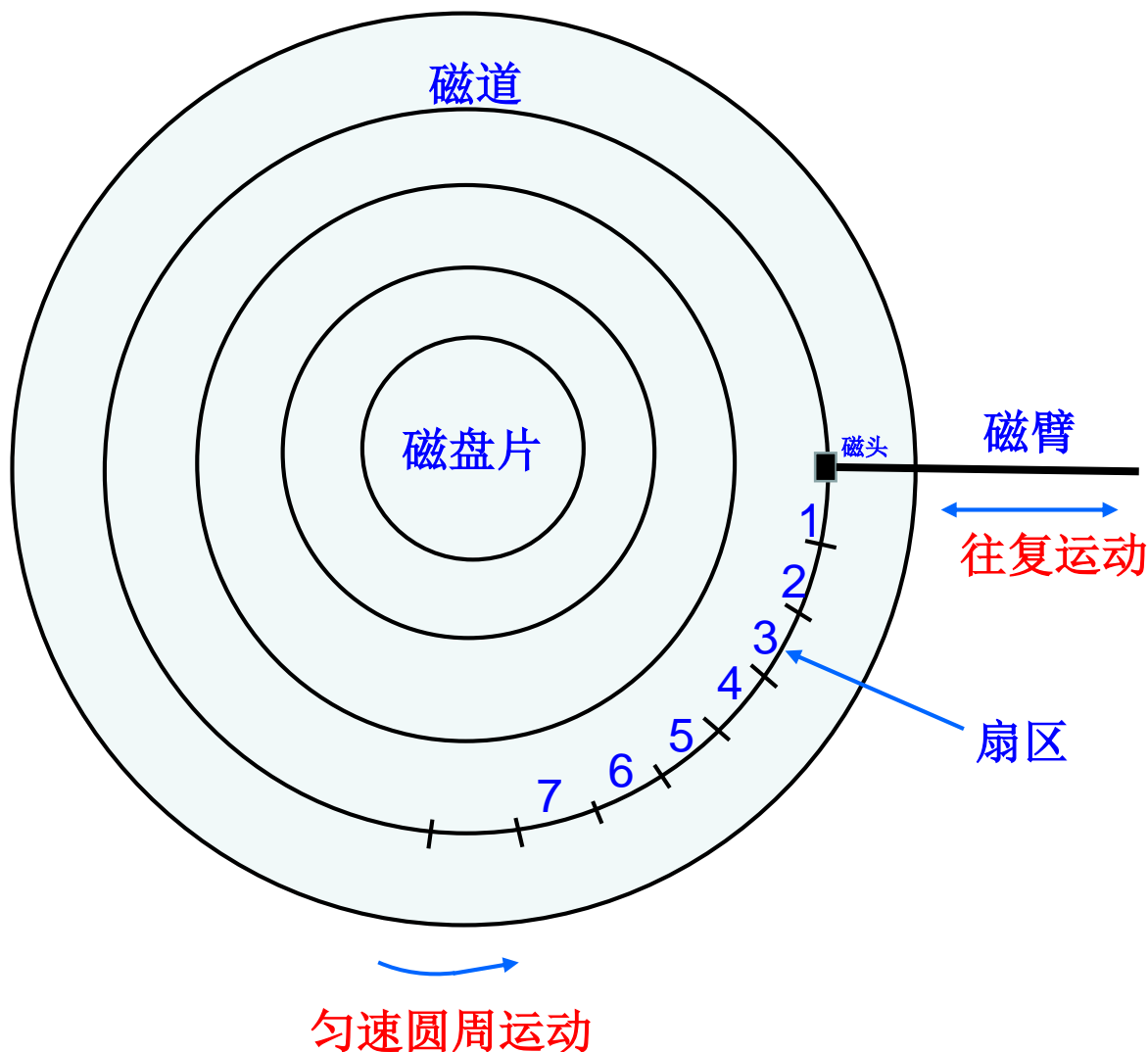
磁盘

磁头

磁头摆臂







1、扇区 (sector) 的大小？

512字节

2、磁盘读取数据的流程？

磁头 (读取数据)



磁盘控制板 (组装、  
校验、发中断信号)



内存 (传输、处理)



磁盘控制板 (初始化)

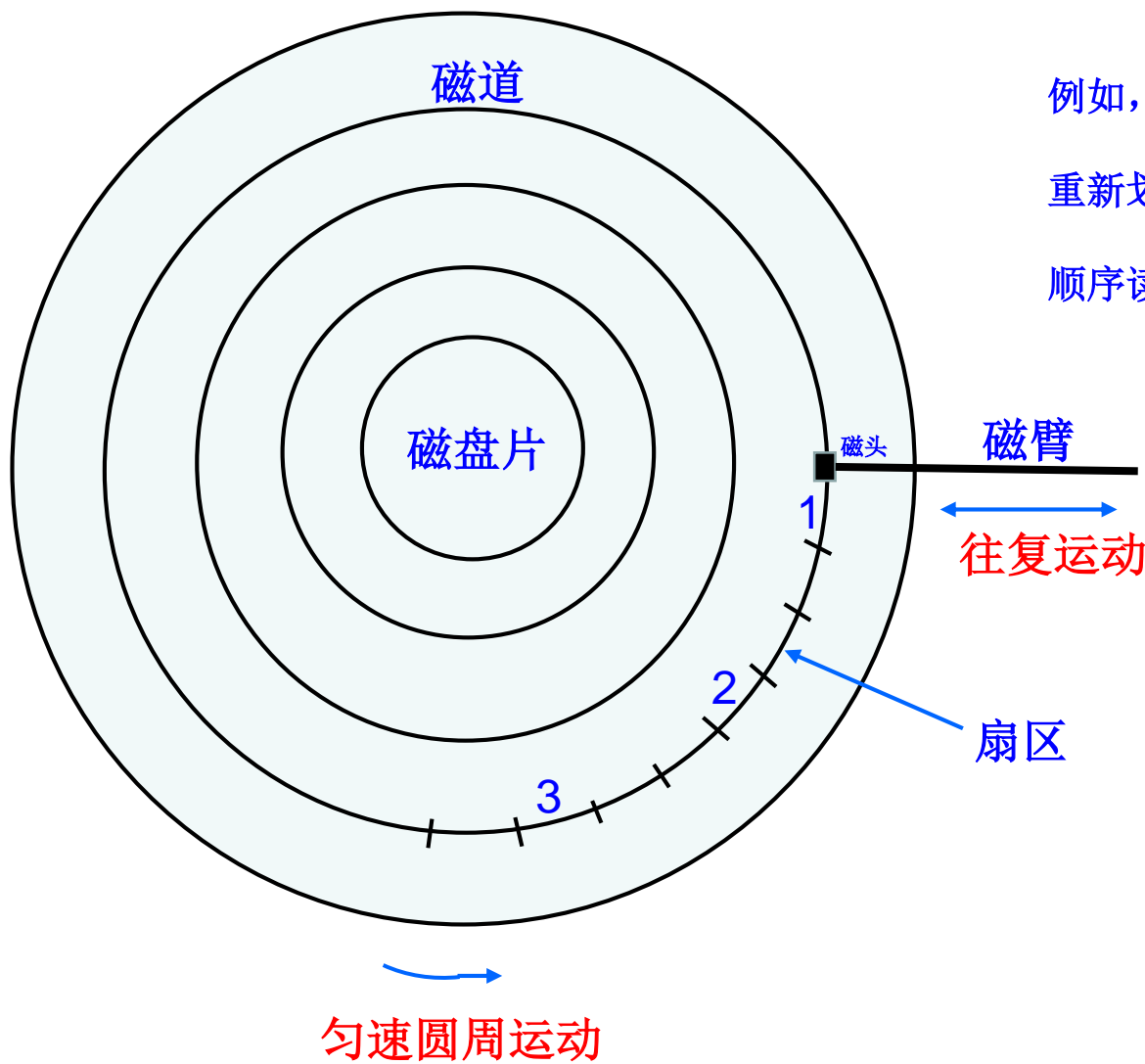


磁头 (再次读取数据)

3、系统开销？

gap 参数

4、顺序读取相邻扇区  
时有等待！



例如,  $\text{gap} = 2$

重新划分扇区编号

顺序读取相邻扇区时无等待!

## 超级块拷贝块：

在每个柱面组块中存放有一个超级块拷贝块，其目的是使系统在超级块被意外破坏时，能从任何一个柱面组中进行恢复而不致使整个文件系统陷入瘫痪。

每个柱面组中的超级块拷贝块的存放位置为安全起见不一定都装在柱面组中的最前面，而是可浮动地装在该柱面组中的任何位置。

一般性的方法是：如果第 $n$ 号柱面组中的超级块拷贝块开始于该柱面组中的第 $i$ 磁道，则第 $n+1$ 柱面组中的超级块拷贝块开始于该柱面组中的第 $i+1$ 磁道。文件系统一旦建立后，它们的位置就是固定不变的。

## 柱面组信息块（**cg**块）

柱面组信息块中存放的是有关该柱面组的静态参数，它由数据结构**cg**来定义：

```
struct cg {  
    内存中柱面组块的链接指针  
    本柱面组块中i节点表大小  
    本柱面组块中数据区大小  
    最近一次所用块的位置  
    最近一次所用片的位置  
    最近一次所用i节点的位置  
    本柱面组空闲数据块总数  
    i节点位示图  
    空闲块位示图  
}
```



## 位示图:

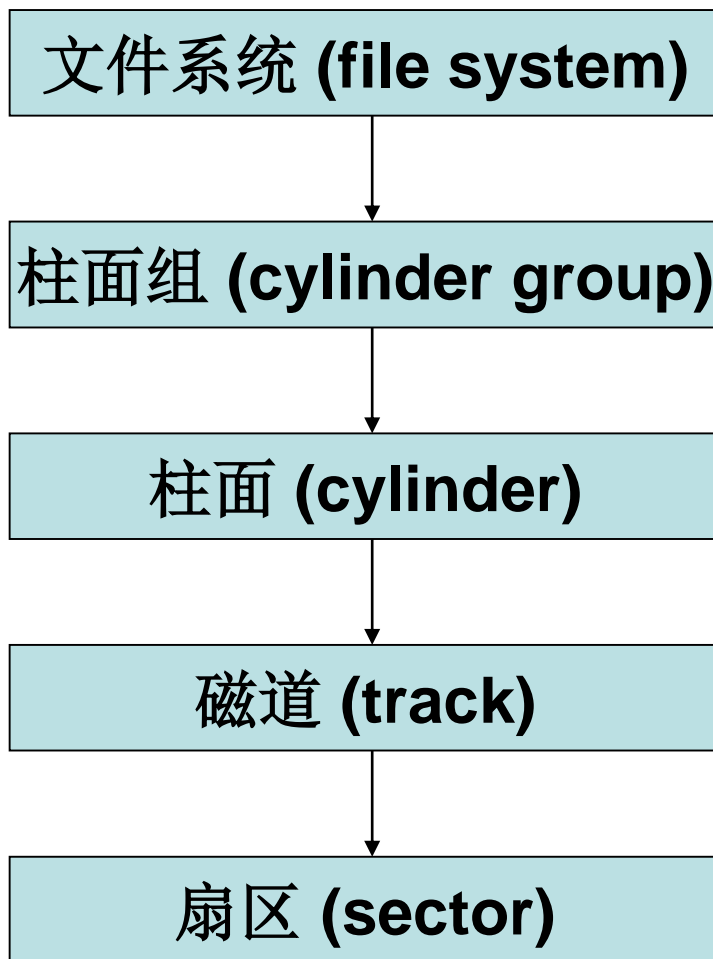
位示图为一张表，其中的每一个二进制位（**bit**）的值来表示某一个资源（例如数据块或i节点）的状态，这样每检测一个字节的值就可以知道八个资源的状态；每检测一个四字节的整数的值就可以知道**32**个资源的状态。

系统只需要维护一张较小的表（位示图）就可以快速地检测指定资源的忙闲状态，或快速查找可用的空闲资源。

1	0	0	1	1	0	1	0	9A
0	0	1	1	0	1	1	0	36
1	1	1	1	1	1	1	1	FF
1	0	0	1	0	1	0	0	94
0	1	1	0	0	0	1	1	63
0	0	0	0	0	0	0	0	00
1	1	0	1	1	0	0	1	DA
0	1	0	0	1	1	1	0	4E

## 2、文件系统的数据块

在文件系统中，按存储单位来划分，由大到小可有下列层次：



**DEV\_BSIZE    512字节**

## 文件系统的逻辑块大小:

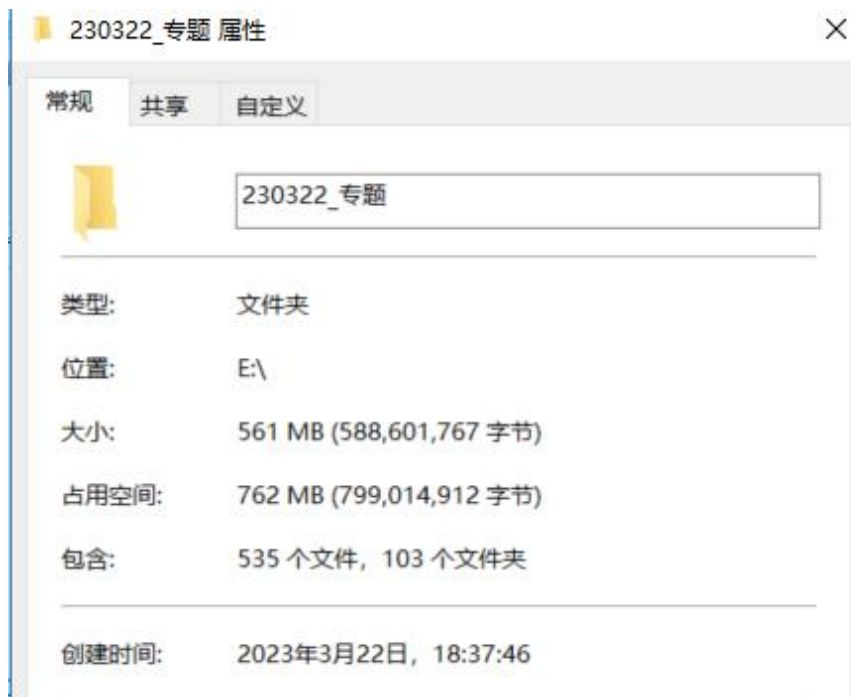
**DEV\_BSIZE \* 2<sup>n</sup>** 即1k、2k、4k、8k、16k ...

目的: 提高传输速度, 减少overhead

## 文件系统的逻辑片大小:

**DEV\_BSIZE \* 2<sup>n</sup>** 即1k、2k、4k、8k、16k ...

目的: 减少文件尾的碎片浪费。



左图: MS-windows中某文件夹的属性显示, 占用空间大于文件夹实际大小, 每个文件的末尾都可能有碎片浪费。

### 3、i节点与磁盘i节点表

超级块	磁盘i节点表	数据存储区
-----	--------	-------

icommon
icommon
icommon
icommon
icommon

磁盘icommon表

- 文件所有者标识 (**UID**)
- 用户组标识 (**GID**)
- 文件类型 (**FIFO**、**DIR**、**CHR**、**BLK**、**REG**、**LNK**等)
- 文件保护模式 (存取许可权) **mode**
- 文件存取时间 (**atime**, **mtime**, **ctime**)
- 链接数目 **link**
- 文件大小 **size**
- 文件数据块索引表 **index table**

## 4. 文件的存贮结构

**UNIX**的普通文件的逻辑结构是无格式的有序字节流，而它们的物理存贮结构是以索引方式来组织的。

每个文件都是由一个索引节点*i*节点来表示的，每个*i*节点由其*i*节点号来标识。

*i*节点通常以静态的形式存放在磁盘的*i*节点表中。每个磁盘*i*节点表项是由数据结构**icommon**定义的，描述对应文件的静态参数。

## icommon 与 inode 的关系

进程要读写一个文件时，先在内存的活动i节点表（即**inode**表）中申请一个空闲的活动i节点，并把磁盘上i节点（**icommon**）中的各项参数读入其中，当核心操作完成后，如果必要，就把在内存中的活动i节点写回到磁盘上去。

内存活动i节点由数据结构**inode**来定义，它除了包含磁盘上对应的**icommon**中的各项参数外，还包含有其它的参数，如该活动i节点的状态、文件所在的逻辑设备号、i节点号、活动i节点链接指针，最近使用的i节点在目录中的位置等动态信息。

```

struct inode {
    活动i节点链接指针
    状态标志
    设备号
    i节点号
    最近访问的i节点在目录中的位置
    空闲i节点链接指针
    struct icommon {
        文件模式和类型 (FIFO、DIR、CHR、BLK、REG、LNK等)
        文件链接数
        文件所有者标识数 (UID)
        文件所属用户组标识数 (GID)
        文件大小
        文件最近存取时间 (atime, mtime, ctime)
        数据块索引表
        其它信息
    }
}

```

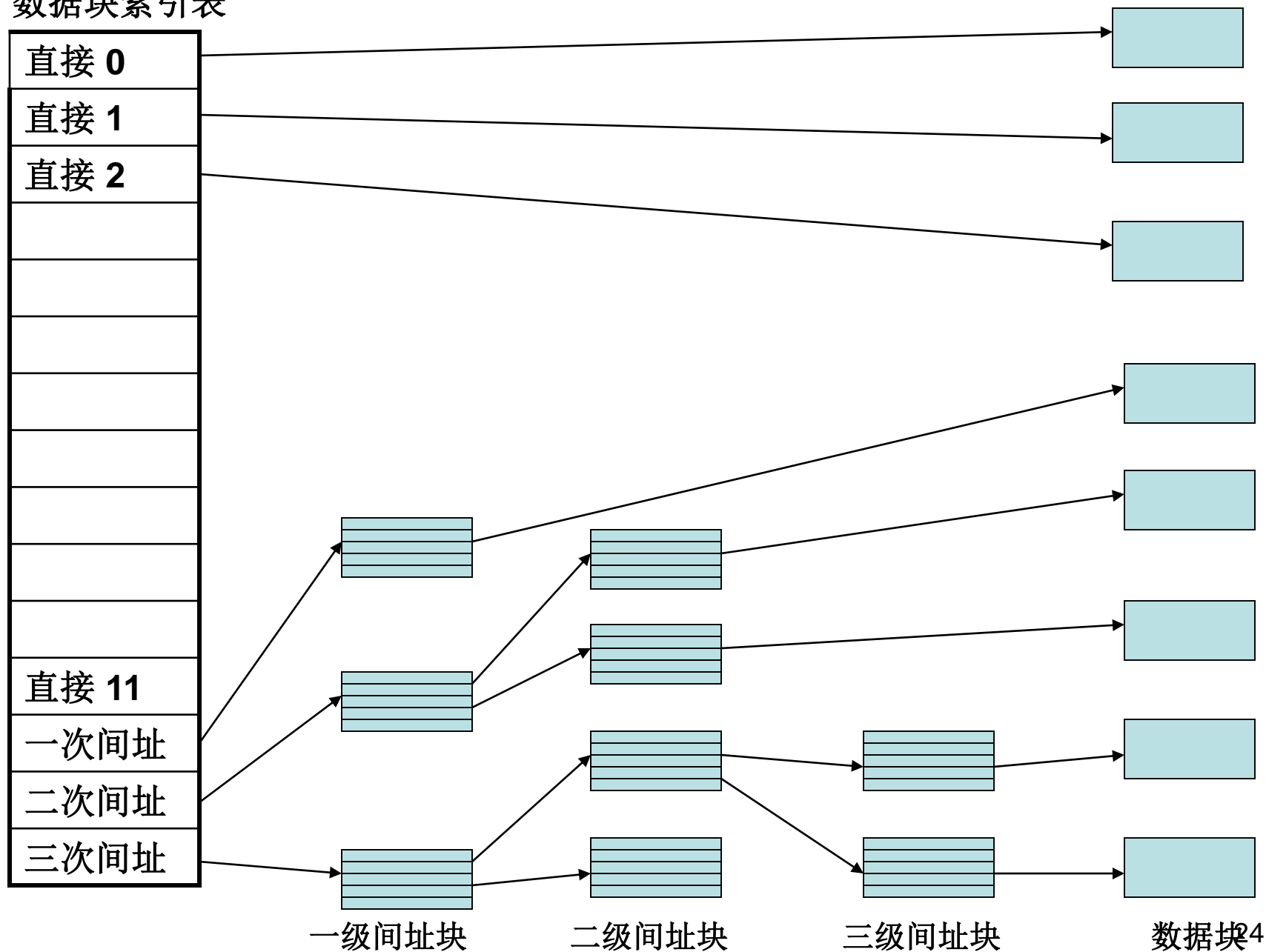
## 5、数据块索引表

数据块索引表用于检索本文件占用的数据块。它包含**12**项直接索引表目和**3**项间接索引表目。根据要读写的数据在文件中的位置可计算出该数据所在的逻辑块号，查索引表就可找到逻辑块所在的文件系统块号。

系统根据计算出来的逻辑块号判断是否包含在直接索引表中，如果是，则取出直接索引表中的文件系统块号；如不是，则看是否包含在一次间接索引块中，否则再寻找二次和三次间接索引块。最长要存取三次间址索引块才能找到相应数据的文件系统块号（要取出数据则要读**4**次磁盘）。



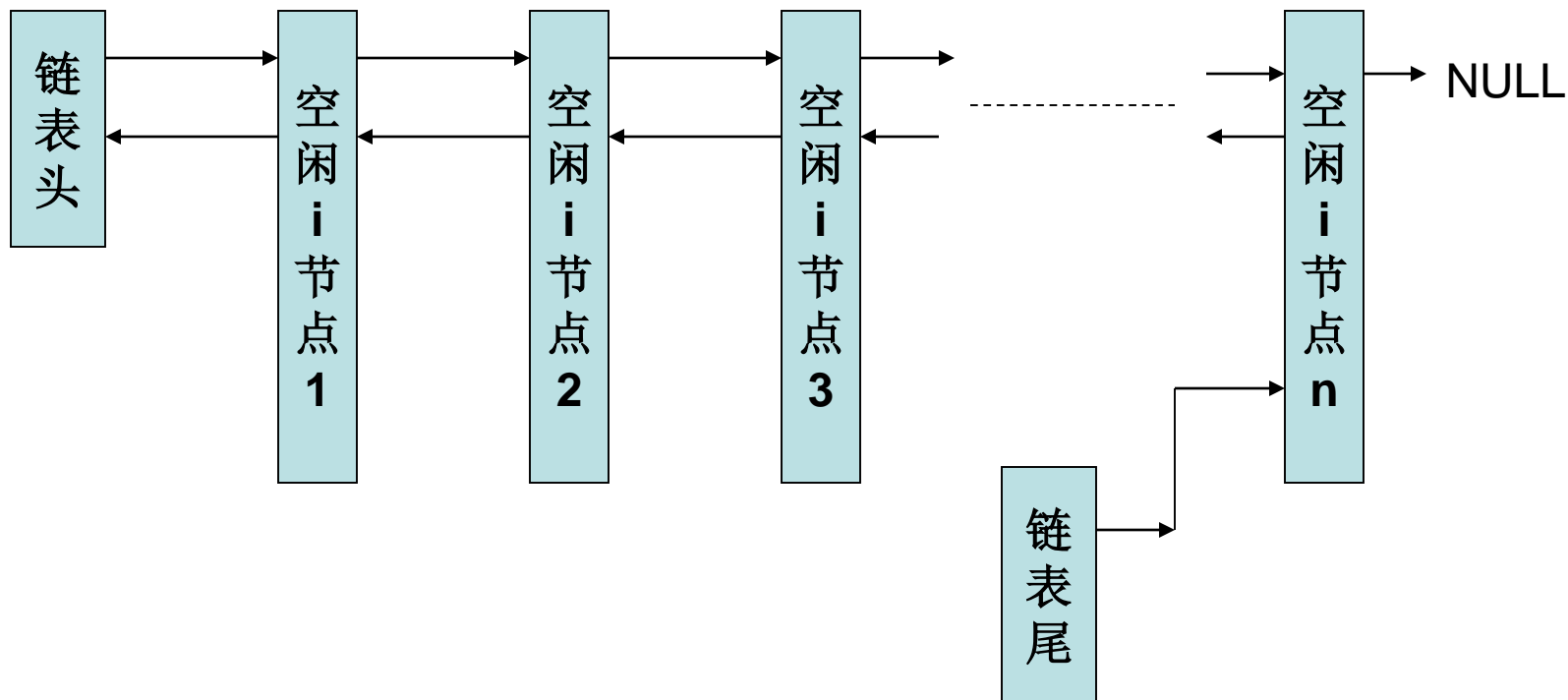
数据块索引表



## 6、inode表的结构

在内存中，活动i节点表类似于数据缓冲区高速缓冲中的缓冲池结构。活动i节点表中的每一项就是一个活动i节点缓冲区，用来存放一个被打开文件的**inode**。（以下把活动i节点缓冲区简称为“活动i节点”）。

空闲的活动i节点相互链接在一起构成“空闲活动i节点链表”，这是一个双向（非循环）链表，分别由链头指针和链尾指针指向空闲活动i节点链表的开始和结束。如下图所示：



空闲活动*i*节点链表为双向（非循环）链表，分别由链表头指针和链表尾指针指向空闲链表的首尾。

## 活动i节点hash链表

当某个文件（即某个磁盘i节点）被打开时，根据该i节点所对应的设备号和i节点号计算其hash值：

$$hn=(devno+inumber)\%64$$

可得到0~63共64个hash值。具有相同hash值的活动i节点链接在同一个hash链表中，这样内存中就有64个hash链表，每个hash链表都是由hash链头开始的双向链表（与数据缓冲区链表不同的是此处的空闲和非空闲链表都是非循环的）。内存活动i节点表就是由这64个hash链表组成。如下图所示：

The diagram illustrates a linked list structure. On the left, a vertical light blue rectangle represents a header table with three sections: 'hn=0' at the top, 'hn=1' in the middle, and 'hn=63' at the bottom. To the right of this table are several light blue boxes, each labeled 'inode'. Dashed lines connect the 'hn=0' section to the first 'inode' box, the 'hn=1' section to the second 'inode' box, and the 'hn=63' section to the sixth 'inode' box. Solid double-lined arrows show the following connections: from the first 'inode' to the second, from the second to the sixth, from the sixth to the seventh, and from the seventh to a final 'inode' box labeled 'NULL'. At the bottom, two light blue boxes are labeled '空闲链表头' (Free List Head) and '空闲链表尾' (Free List Tail). A solid double-lined arrow points from '空闲链表头' to the sixth 'inode' box, and another solid double-lined arrow points from '空闲链表尾' to the seventh 'inode' box.

## 活动 inode 表的结构

## 7. 目录和目录项

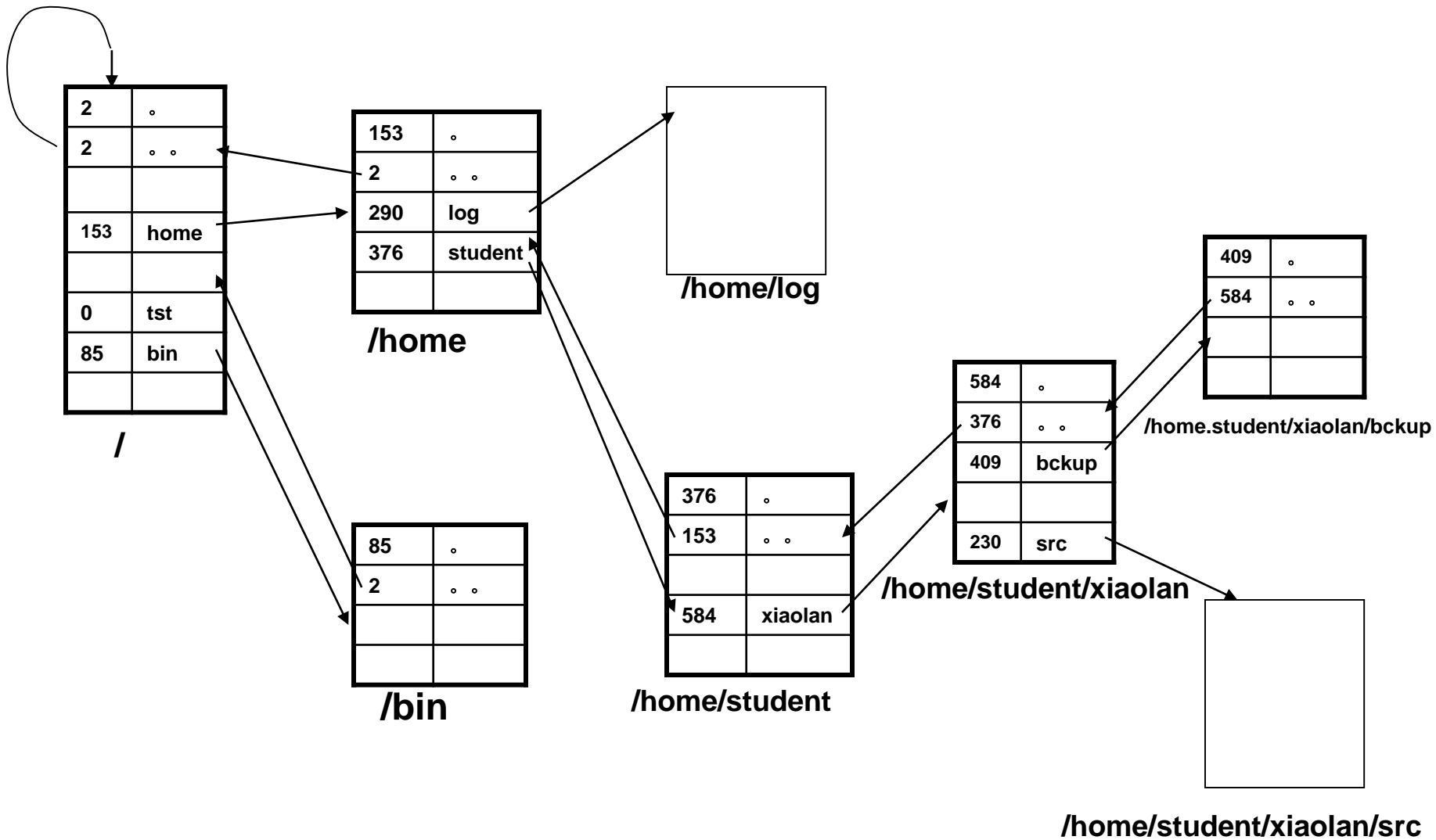
在 **UNIX** 文件系统中，目录的组织形式采用的是树形结构，一个逻辑文件系统就是一棵目录树。

目录也被当作文件进行处理，一个目录文件的结构为表状结构，其中通常包含有若干表项，称为目录项，这些目录项既可以是普通文件的入口，也可以是子目录的入口。

每一个目录项中通常包含两部分内容：

文件的 **i** 节点号

文件名



目录 `/home/student/xiaolan` 的路径和目录结构

每个目录项由数据结构**direct**来定义：

```
#define MAXNAMLEN 14
struct direct {
short d_ino;           /*目录项i节点号*/
char d_name [MAXNAMELEN]; /*目录项名字字符串*/
}
```

每个目录项的长度通常是确定的，为**16**个字节，其中前两个字节存放文件的i节点号**d\_ino**，后面**14**个字节存放文件名**d\_name**。

这种定长目录项在算法实现方面比较简单，在使用灵活方面都有所不便，并且可能因许多目录项名字长度不足**14**字符面有空间浪费。



在**UNIX**的每个文件系统中，有三个 **i** 节点号是有固定用途的：

### 0号**i**节点：

表示空目录项，当某个目录项被删除时，该目录项的 **i** 节点号被置为**0**。

### 1号**i**节点：

表示坏块文件，所有的磁盘坏块都划归到该节点上；

### 2号**i**节点：

固定表示该逻辑文件系统的根（**root**）目录；

### 3号**i**节点：

表示该文件系统中的 **lost+found** 目录。

## 8、变长目录项的目录结构

```
#define MAXNAMLEN 255
struct direct {
    long d_ino;           /* 目录项i节点号 */
    short d_reclen;       /* 目录项入口长度（占用长度） */
    short d_namelen;      /* 目录项名字长度 */
    char d_name [MAXNAMLEN+1] /* 名字字符串，+1为串结束符\0 */
}
```

由于目录中各目录项的长度是变化的，因此必须在目录项中标明本目录项的长度。前一个目录项释放时，把该目录项的空间全部合并到前一个目录项中，形成前面一个目录项占用空间大于实际使用的空间。

在增加新目录时，先查看目录中各目录项是否占用多余空间，如有，则进行压缩，把释放的空间分配给新目录项。

变长目录结构增加了算法复杂性和工作量，通常用在硬件性能较高的大型系统中。

## 9、文件名和文件名缓冲区

核心在内存中建立了一个高速的名字缓冲区，用来存放最近使用过的文件名，核心认为“最近使用过的文件名马上还要使用的可能性最大”。

名字缓冲区是由**ncache**定义的数据结构，只包含文件名和索引节点指针等重要信息：

```
struct ncache {
```

```
    hash链表指针
```

```
    LRU链表指针
```

```
    文件i节点指针
```

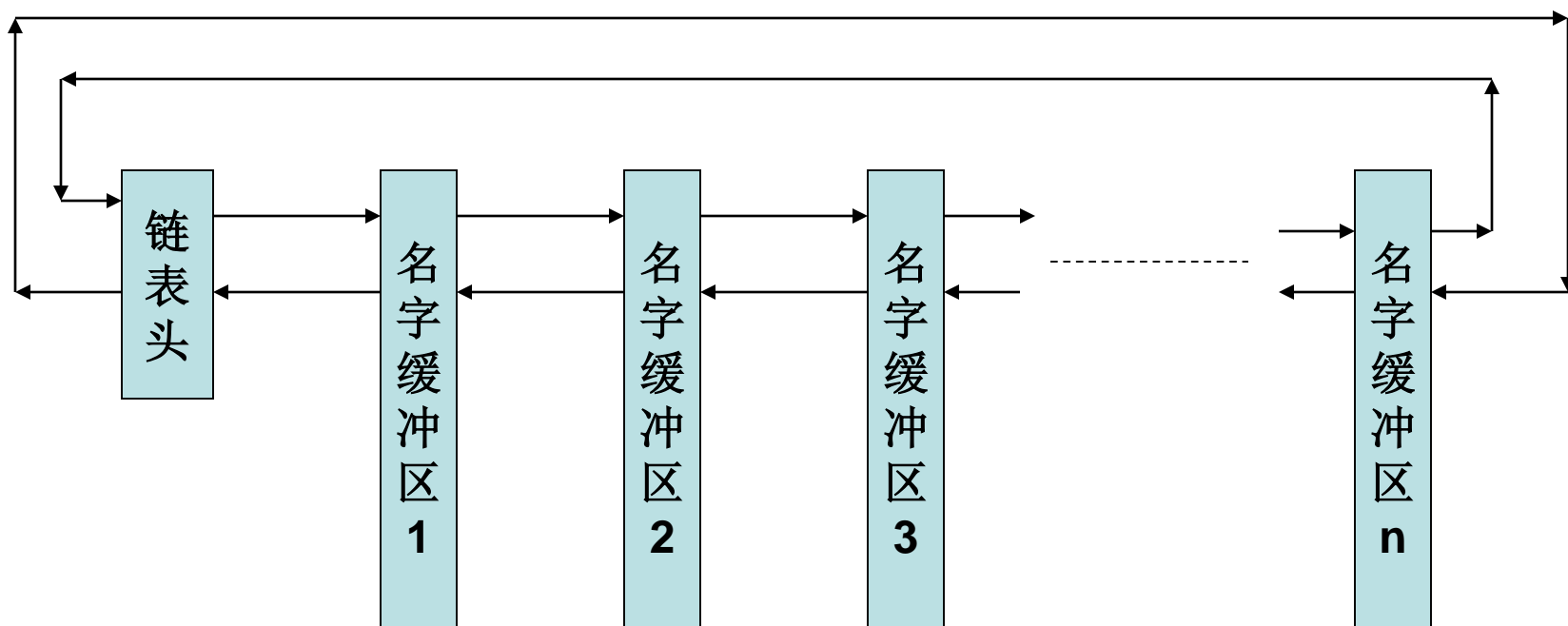
```
    /* 这里只需要节点指针，因内存  
       中已有活动i节点表 */
```

```
    文件父目录节点指针
```

```
    文件名
```

```
    确认信息
```

```
}
```



名字缓冲区（ncache）链表为定长双向循环链表（LRU链）

为提高查找文件名的速度，还根据每个ncache的hash值，将其挂接到一个hash链表中。ncache的hash值用下列公式计算：

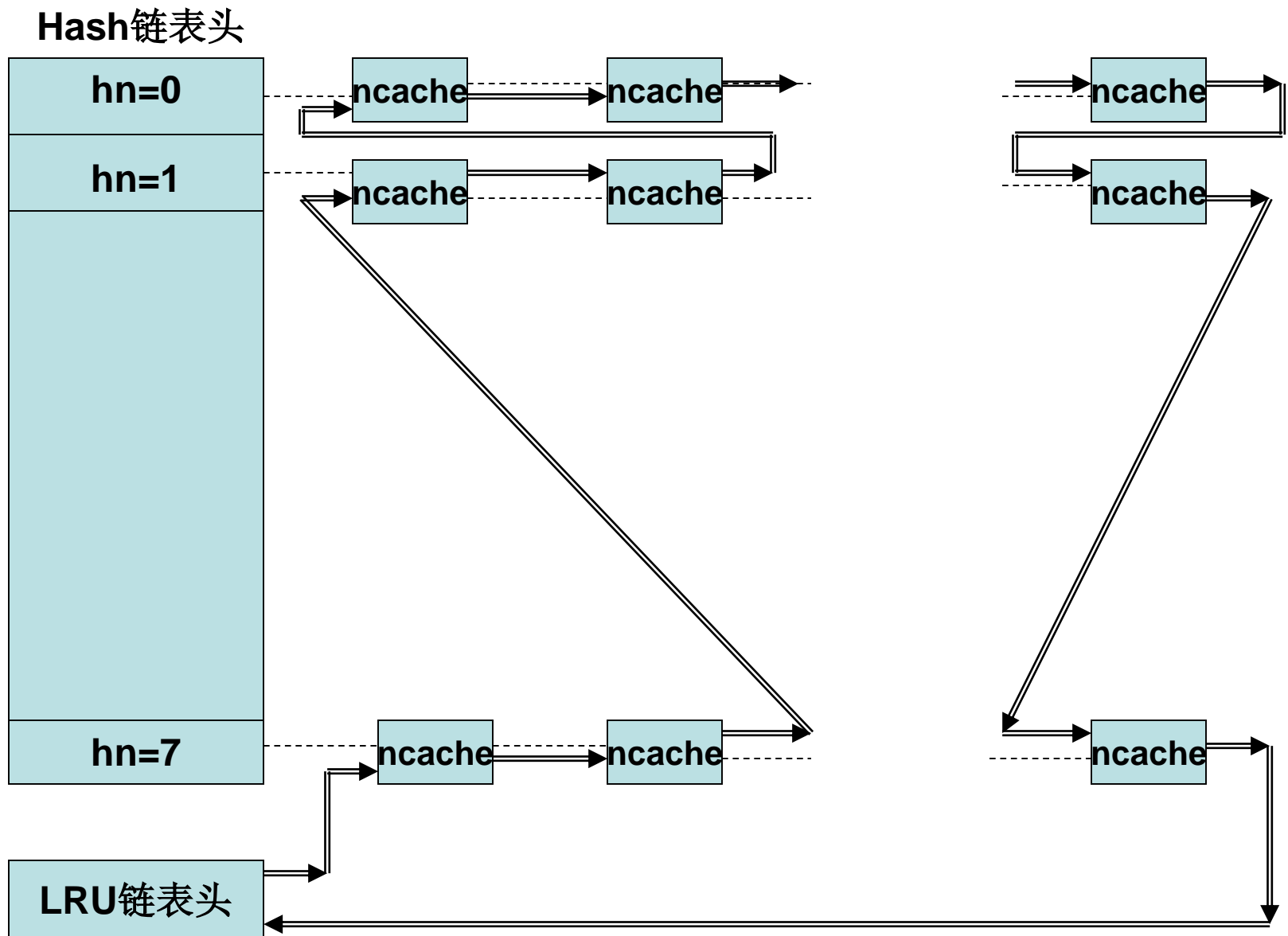
$$hn=7\&(*namep+*(namep-1+slen)+slen+(int)VP)$$

**namep:** 为指向名字字符串的指针

**slen:** 为名字长度

**VP:** 为父目录节点指针

计算出相应的hash值（0~7）。共建立8个hash链，每个hash链是一个以链表头开始的双向循环链表，每个ncache任何时候都同时既在hash链上，又在LRU链上。



名字缓冲区(ncache)池结构

## 10、资源保护

**UNIX**文件系统中的资源保护系统主要是对系统中的两类资源进行保护：

### ① 静态硬资源

包括存贮空间和索引节点，对这类资源的保护主要是通过 **quota** 系统提供的限量机制实现的。

### ② 动态资源

主要是指临界区资源或独享资源，对这类资源的保护主要是通过上锁机制来实现的。

## 1) quota系统

**quota**系统的主要功能就是检测和限制用户对文件系统资源的使用。

在每个逻辑文件系统（包括根文件系统和各子文件系统）中都**可以**建立一个磁盘**quota**文件来限制用户对本文件系统资源的使用。

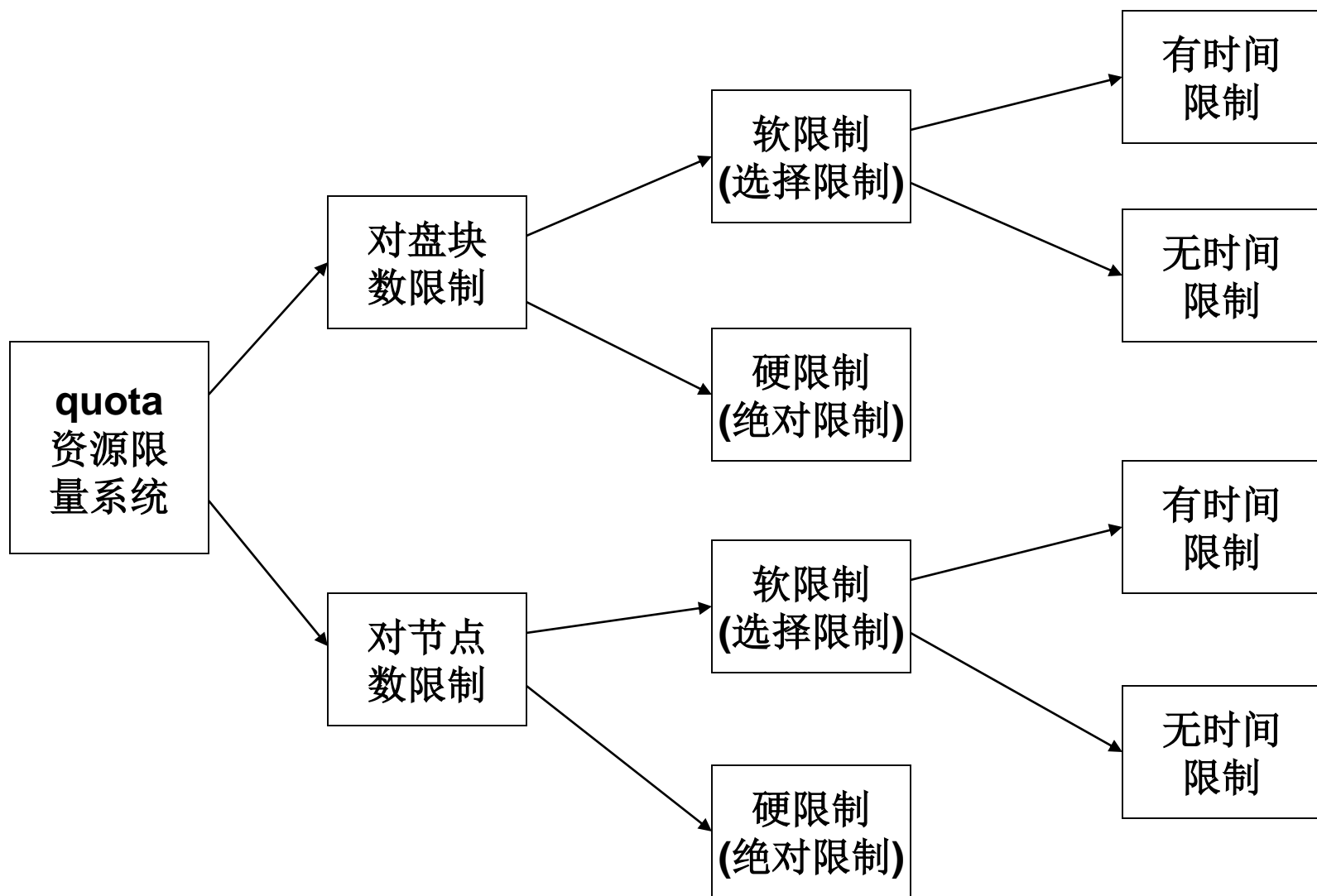
每个**quota**文件是由多个**dqblk**数据结构所组成的表格，每个**dqblk**都针对一个用户，用于存放该文件系统对该用户的限制参数。

如果一个用户的使用空间包含多个文件系统，则需在每个文件系统的硬盘**quota**文件中给该用户分配一个**dqblk**项。



```
struct dqblk {  
    盘块数硬限制  
    盘块数软限制  
    已用盘块数  
    节点数硬限制  
    节点数软限制  
    已用节点数  
    盘块数时间限制  
    节点数时间限制  
}
```

系统在运行时，在内存中建立了一个活动**dqblk**链表（类似于活动**i**节点表），每个活动**dqblk**由其**hash**值而分别处在**64**链中的某个链表上，如果某个活动**dqblk**是空闲的，它同时又处在空闲活动**dqblk**链表中（与活动**i**节点表的结构完全一样）。



quota限量系统构成

## 2) 上锁机构

为保持数据的一致性，能正常使用临界区资源，**UNIX**文件系统提供了一种对数据（文件或记录）的上锁机制，即劝告锁（**advisory lock**），用户既可给整个文件上锁，也可给部分记录上锁。

劝告锁包含两种类型：

共享锁（读操作锁）

互斥锁（写操作锁）

在同一个文件或同一个记录上同时只能有一个互斥锁（或写操作锁），但可以同时有多个共享锁（读操作锁）。共享锁的级别低于互斥锁，互斥锁可以覆盖共享锁。

对一个文件上锁只是一种“自觉性”的保护措施——“劝告锁”，进程在读写该文件之前都必须先检查该文件是否已上锁，然后再进行相应的锁操作后才能对该文件进行读写。否则将使该文件上原有的所有劝告锁失效。

## 4.1.2 虚拟文件系统

### 1. 文件结点

在本地文件系统中，每个文件都由一个索引节点**inode**来代表，对一个文件名的操作都被转换成对该文件的**inode**进行相应的操作。

类似地，在虚拟文件系统中，每个文件都是由节点**vnode**来代表，对文件名的操作都被转换成对该文件的**vnode**进行相应的操作。**v**节点中包含了它所代表的活动文件的有关信息，如文件状态、上锁情况、引用计数、文件类型等，它由数据结构**vnode**来定义：

```

struct vnode {
    节点状态标志      /*是否为所在OS的根，是否上锁，
                        有否等待上锁进程，是否已被修改*/
    访问计数值      /*多少进程在共享该节点*/
    上锁情况      /*上了多少共享锁，互斥锁**/
    该v节点操作指针 /*指向操作函数组*/
    节点类型      /*VREG, VDIR, VBLK, VCHR,
                  VLNK, VFIFO*/
    设备号      /*所在设备*/
}

```

**VFS**接受用户的操作请求，把用户对文件名的操作转换成对相应的**v**节点操作，如果该文件实体在本地机器上，则虚拟文件系统把**v**节点转换成本地文件系统中对应的**i**节点，并把用户操作转给本地文件系统去完成；

如果文件实体在远地机器上，虚拟文件系统把操作转给网络文件系统，由网络文件系统起动网络驱动程序，执行对网络上其它机器上某个文件的操作。事实上对远地文件的操作最终<sub>4</sub>落实到远地机器的本地文件系统（对它的**i**节点的）操作上。

**v节点与i节点有许多相似之处，都是对文件的属性进行描述，但又有较大的差异。**

## **两者的关系是：**

- ① **i节点是v节点的基础。任何v节点都直接或间接地与某个i节点相联系，而不论该i节点代表的文件实体存放在本地机器上还是存放在远地机器上。**
- ② **v节点是i节点的高层节点。它是活动i节点在虚拟文件系统中的代表。**

## 两者的区别：

- ① **v**节点是一个动态的概念。在系统中只有当前活动的文件（即正被使用的文件）才有对应的**v**节点；而磁盘上某个当前未被使用的文件仍然有磁盘**i**节点与之对应。
- ② **v**节点在磁盘上没有对应的数据区，当某个**v**节点所代表的文件使用完毕被关闭时，该**v**节点也随之完全消失；而不像**i**节点被释放时，要把被修改过的信息拷贝到磁盘上。

## 2. 子文件系统

一个虚拟文件系统通常可由多个子文件系统组成，这些子文件系统既可能是本地的，也可能是远地的。当这些子文件系统被安装到虚拟文件系统中时，为了方便地识别和找到它们，给它们各建立了一个控制块 **vfs** 作为代表，（类似于文件在虚拟文件系统中用**vnode**作代表）。**vfs**中包含的是该子文件系统的一些基本信息，它由数据结构**vfs**定义。

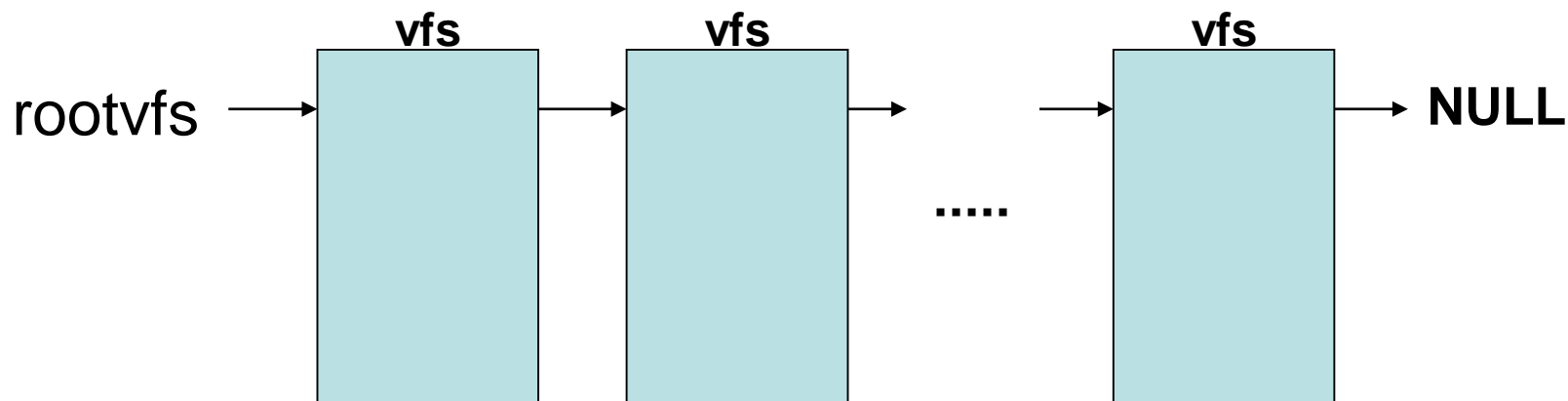


```

struct   vfs   {
    该子文件系统的操作指针
    安装点的v节点
    状态标志                      /* 只读、上锁、有等待上锁进程等 */
    基本文件系统块大小
    文件系统标识数                /* 指明是本地还是远地的文件系统 */
    安装表项指针
}

```

各个**vfs**链接成一个单向链表：**VFS链**



子文件系统被安装时，在系统安装表中给该子文件系统建立了一个安装表项，使得每个子文件系统的 **vfs** 都与一个安装表项一一对应，每个安装表项由数据结构**mount**定义：

```
struct mount {  
    该文件系统的vfs指针  
    设备号  
    超级块缓冲区指针  
    各项quota值  
    该文件系统根节点指针  
    安装点节点指针  
}
```

要对某个子文件系统进行操作，能够方便地由**vfs**链表找到该子文件系统的超级块：

由**vfs**→**mount**→**buffer**→超级块（**fs**）

### 3、虚拟文件系统的实现

通常用户与虚拟文件系统之间的交互界面是**32**个基本的系统调用，其中**26**个系统调用是实现对虚拟文件系统中的各个文件进行建立，删除、打开、关闭、查询、链接、修改、换名等操作。

用户通过文件名对文件的操作在虚拟文件系统中都被转换为相应的**vnode**的操作。在虚拟文件系统中的任何一个文件，无论它是属于本地文件系统还是属于远地文件系统，其**vnode**都具有**完全相同的形式**。

但是如果两个文件分别存放在不同的机器上，则在**某一台**机器上即使是对它们进行同一类型的操作（如读文件），也显然会是执行两个完全不同的程序（一个是起动本地磁盘，而另一个则是设置网络准备进行通讯）。

在虚拟文件系统中，给每个**vnode**都设置了一个操作指针，指向由**26**个功能函数（完成某一特定任务的函数）指针所组成的操作函数组，这**26**个指针分别指向对**vnode**进行各种基本操作的函数，如建立、删除、打开、关闭、查询、修改、链接、读写等。操作函数组由**vnodeops**定义：

```
struct vnodeops {
```

int (*vn_open)();	打开文件
int (*vn_close)();	关闭文件
int (*vn_rdwr)();	读写文件
int (*vn_ioctl)();	输入输出控制
int (*vn_select)();	同步操作
int (*vn_getattr)();	读取属性参数
int (*vn_setattr)();	设置属性参数
int (*vn_access)();	检查访问权限
int (*vn_lookup)();	根据文件名寻找V节点
int (*vn_create)();	建立文件
int (*vn_remove)();	删除文件
int (*vn_link)();	链接文件
int (*vn_rename)();	文件更名

<b>int (*vn_mkdir());</b>	<b>创建目录</b>
<b>int (*vn_rmdir());</b>	<b>删除目录</b>
<b>int (*vn_readdir());</b>	<b>读取目录</b>
<b>int (*vn_symlink());</b>	<b>符号链接</b>
<b>int (*vn_readlink());</b>	<b>读取符号链接文件名</b>
<b>int (*vn_fsync());</b>	<b>内存信息复制到磁盘</b>
<b>int (*vn_inactive());</b>	<b>释放V节点对应的i节点</b>
<b>int (*vn_bmap());</b>	<b>逻辑块到物理块的映射</b>
<b>int (*vn_strategy());</b>	<b>启动设备</b>
<b>int (*vn_bread());</b>	<b>读入数据块</b>
<b>int (*vn_brelse());</b>	<b>释放数据缓冲区</b>
<b>int (*vn_lockctl());</b>	<b>上锁控制</b>
<b>int (*vn_fid());</b>	<b>建立文件标识结构</b>
<b>}</b>	

以上在**vnodeops**定义的操作，是**UNIX**中对文件的最基本操作，对文件的其它任何操作都是通过对这些基本操作的组合来实现的。

如果根据虚拟文件系统中文件种类的不同，而分别设置这**26**个功能函数指针，使这些指针指向完成特定功能的函数，从而形成不同类别的操作函数组，就可适合对不同种类文件的打开、读写、链接、更名等基本操作。

这就是虚拟文件系统的基本操作模式。

在虚拟文件系统中定义了两个类型为**vnodeops**的操作函数组**ufs\_vnodeops**和**nfs\_vnodeops**。

### **ufs\_vnodeops:**

其中的**26**个功能函数指针按**vnodeops**中的功能次序被初始化为指向本地文件系统中，执行对文件进行建立、删除、打开、关闭、修改、链接和读写等操作的函数名；

### **nfs-vnodeops:**

其中的**26**个功能函数指针也按**vnodeops**中的功能次序被初始化为指向网络文件系统中，执行对远地文件进行建立、删除、打开、关闭、修改、链接和读写等操作的函数名。

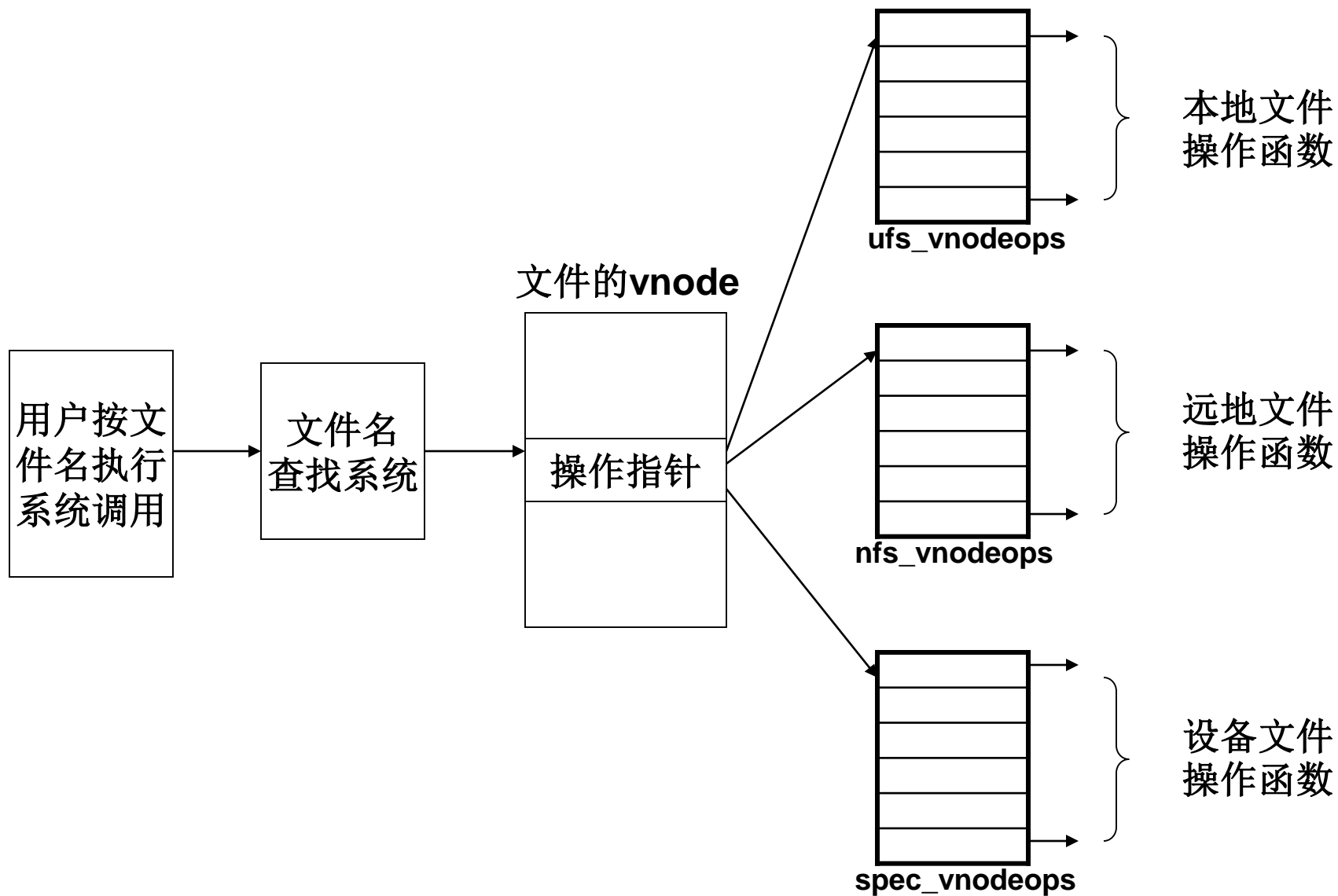


在**vnode**被建立时，根据它所代表的文件是本地的还是远地的，而将**vnode**中的操作指针分别指向**ufs\_vnodeops**或**nfs\_vnodeops**。

当用户对文件名的操作被转换为对相应**vnode**的操作后，再将该操作转换为调用**vnode**中操作指针所指向的操作函数指针组中相应功能的函数，即可完成用户指定的适合于该**vnode**的操作。

由于有这种执行流程的自动转换功能，因此无论对代表本地文件还是远地文件的**vnode**，都采用同一操作过程，从而实现整个文件系统对用户的透明性，这使用户可以完全不必知道要操作文件的具体存放地址和对不同类型文件的操作差异。

这就是虚拟文件系统基本原理。

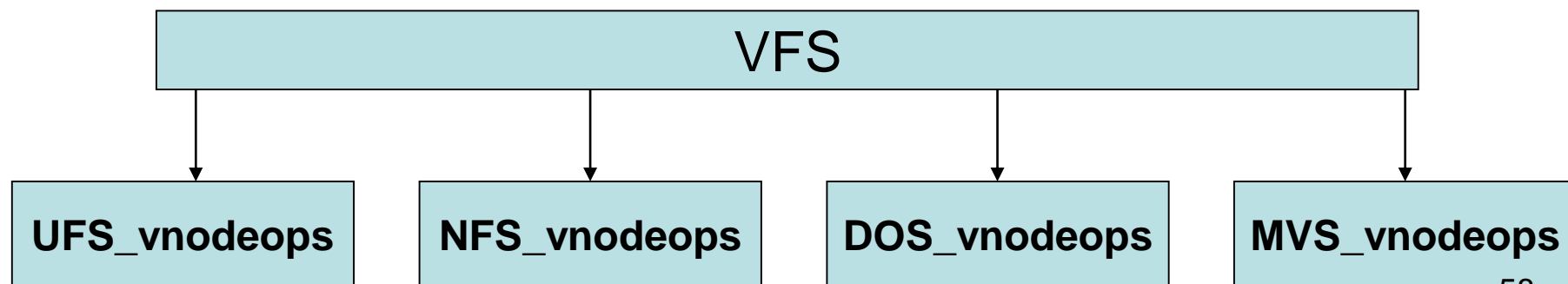


虚拟文件系统中对文件的操作流程

在虚拟文件系统中，除了**ufs\_vnodeops**和**nfs\_vnodeops**以外，还初始化了另外三个**vnode**的操作函数组，即**bdev\_vnodeops**、**fifo\_vnodeops**和**spec\_vnodeops**，这三个操作函数组是分别为块设备类型、**fifo**类型和特殊设备类型文件的**vnode**准备的基本操作函数。

核心对这些特殊文件的**vnode**进行操作时，也采用与普通文件的**vnode**相同的操作过程，从而增加了整个文件系统的模块化和规范化。

**UNIX**的开放性也由此突出地体现出来。要增加什么样的功能或文件类型（如**DOS**）只需增加一个相应的操作函数组（如**DOS\_vnodeops**）和相应的操作函数即可。



与虚拟文件系统中文件的表示方式类似，每一个子文件系统被安装时，在内存中都由一个**vfs结构**来代表，核心对各子文件系统的操作都是首先通过对它们所对应的**vfs**结构进行操作来完成的。

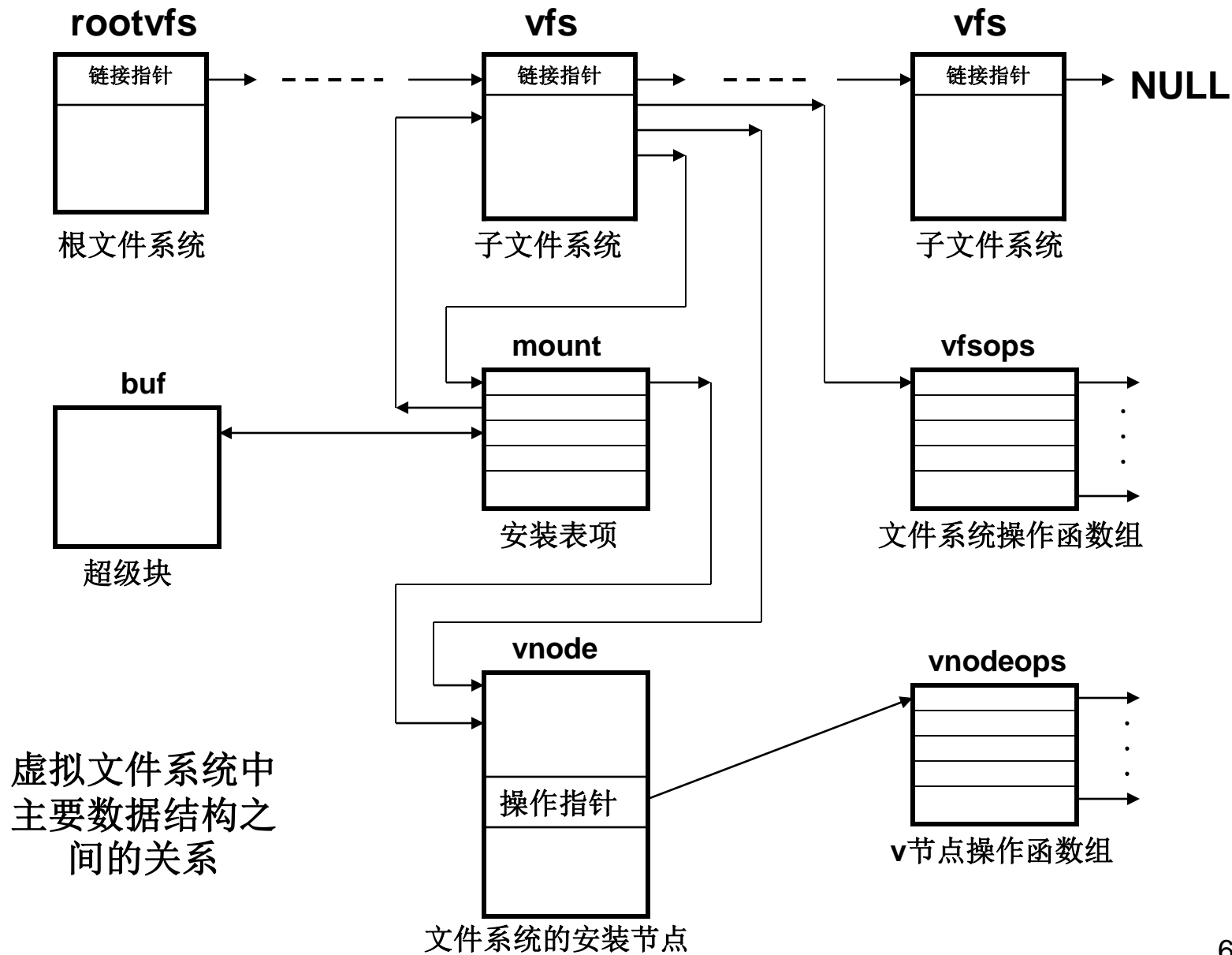
由于这些子文件系统既有本地的，也有远地的，对它们的具体操作必然不相同。

采用与**vnode**类似方法，在每个**vfs**结构中也设置一个操作指针，指向一个由**vfsops**定义的对文件系统进行操作的函数指针组，**vfsops**中共有六个功能函数指针，指向对文件子系统的安装、卸下、读取文件系统设计信息和刷新超级块缓冲区等操作的函数名。

**vfsops**定义如下:

<b>struct</b>	<b>vfsops</b>	{
int	(*vfs_mount)();	安装文件系统
int	(*vfs_umount)();	卸下文件系统
int	(*vfs_root)();	找到根目录的 <b>vnode</b>
int	(*vfs_statfs)();	读取文件系统统计信息
int	(*vfs_sync)();	将内存缓冲区内容复制到外存
int	(*vfs_vget)();	由文件标识数找到相应的 <b>vnode</b>
		}

在系统中初始化了两个这样的操作函数组，即**ufs\_vfsops**和**nfs\_vfsops**，分别适合对本地子文件系统和远地子文件系统的操作。在某个子文件系统被安装时，根据它是本地的还是远地的，而将它的**vfs**结构中的操作指针指向**ufs\_vfsops**或**nfs\_vfsops**。这样在对某个子文件系统进行操作时，不论它是本地的还是远地的都执行同一操作过程，即调用操作函数组中相应指针所指的函数即可完成希望的操作。



## 虚拟文件系统小结

在虚拟文件系统中，无论是代表各种类型文件的**vnode**之间，还是代表不同地点的各子文件系统的**vfs**之间，操作过程都是统一的，只是在进行虚拟文件系统之下的功能操作时，才根据各自的操作指针自动转到不同的函数中去。

这种方式使得整个文件系统结构统一，模块性强，增加功能非常方便；对用户来说，整个文件系统的透明性好，使用简便，避免了用户在对不同类型的文件或不同地点的文件系统进行操作时，分别来设置参数和安排操作过程。

## 4.2 文件系统算法

文件系统的低层算法是建立在缓冲区算法的基础之上的，主要包括以下几项：

分配活动索引节点

释放活动索引节点

逻辑位移量到文件系统块的映射

把路径名转换为索引节点

给新文件分配磁盘i节点

释放磁盘i节点



## 1、分配活动索引节点 **iget**

核心根据文件系统号和*i* 节点号，在内存活动*i* 节点表（**inode**表）中申请一个空闲的活动*i*节点，再把磁盘上对应的*i*节点（**icommon**）拷贝到这个空闲的活动*i*节点中。

如果要找的*i*节点已经在内存的活动*i*节点表中，则增加该*i*节点的**引用计数**，并将其上锁返回；

如果要找的*i*节点不在内存的活动*i*节点表中，则在空闲活动*i*节点表中申请一个活动*i* 节点，将磁盘**icommon**读入其中，重新计算其**hash**值后放到新的**hash**链表中。

## 算法 iget

输入：文件系统索引节点号

输出：上锁状态的索引节点

```
{
    while (未完成)
    {
        if (是索引节点高速缓冲中的索引节点)
        {
            if (索引节点为上锁状态)
            {
                sleep (索引节点变成开锁状态事件);
                continue;
            }
            /* 对安装节点进行特殊处理 */
            if (是空闲链表上的索引节点)
                从空闲链表上移去该节点;
            索引节点引用计数值加一;
            return (索引节点);
        }
    }
    /* 接下页 */
}
```

**/\* 不是索引节点高速缓冲中的索引节点 \*/**

**if**（空闲链表上没有索引节点）

**return**（错误）；

从空闲链表上移出一个新的索引节点；

重置文件系统号和索引节点号；

从老的**hash**队列中撤掉索引节点，把索引节点放到新的**hash**队列中；

从磁盘上读索引节点（**bread**）；

索引节点初始化（如访问计数值置为**1**）；

**return**（索引节点）；

**}**

## 2、释放活动索引节点 **iput**

当核心使用完一个文件要释放其i节点时，首先将该索引节点的引用计数（**reference count**）减一。

如果引用计数不为零，则表明还有其它进程在使用该文件（节点），本进程直接返回。

如果引用计数为零，则表明没有其它进程使用该索引节点了。此时如果该节点的链接数为零，则表明要删除该文件，释放磁盘**icommon**和数据块；如果链接数不为零，则把修改过的**icommon**从**inode**中写回到磁盘上。

最后再把该**inode**释放到空闲活动**inode**表中。

算法 **input**

输入：指向内存索引节点的指针

输出：无

```
{  
    如果索引节点未上锁，则将其上锁；  
    将索引节点的引用计数值减一；  
    if（引用计数值为0）  
    {  
        if（索引节点的链接数为0）  
        {  
            释放文件所占的磁盘块（free）；  
            将文件类型置为NULL；  
            释放磁盘索引节点（ifree）；  
        }  
        if（文件被存取或索引节点被改变或文件内容被改变）  
            修改磁盘索引节点；  
        把该索引节点放到空闲链表中；  
    }  
    为索引节点解锁；  
}
```

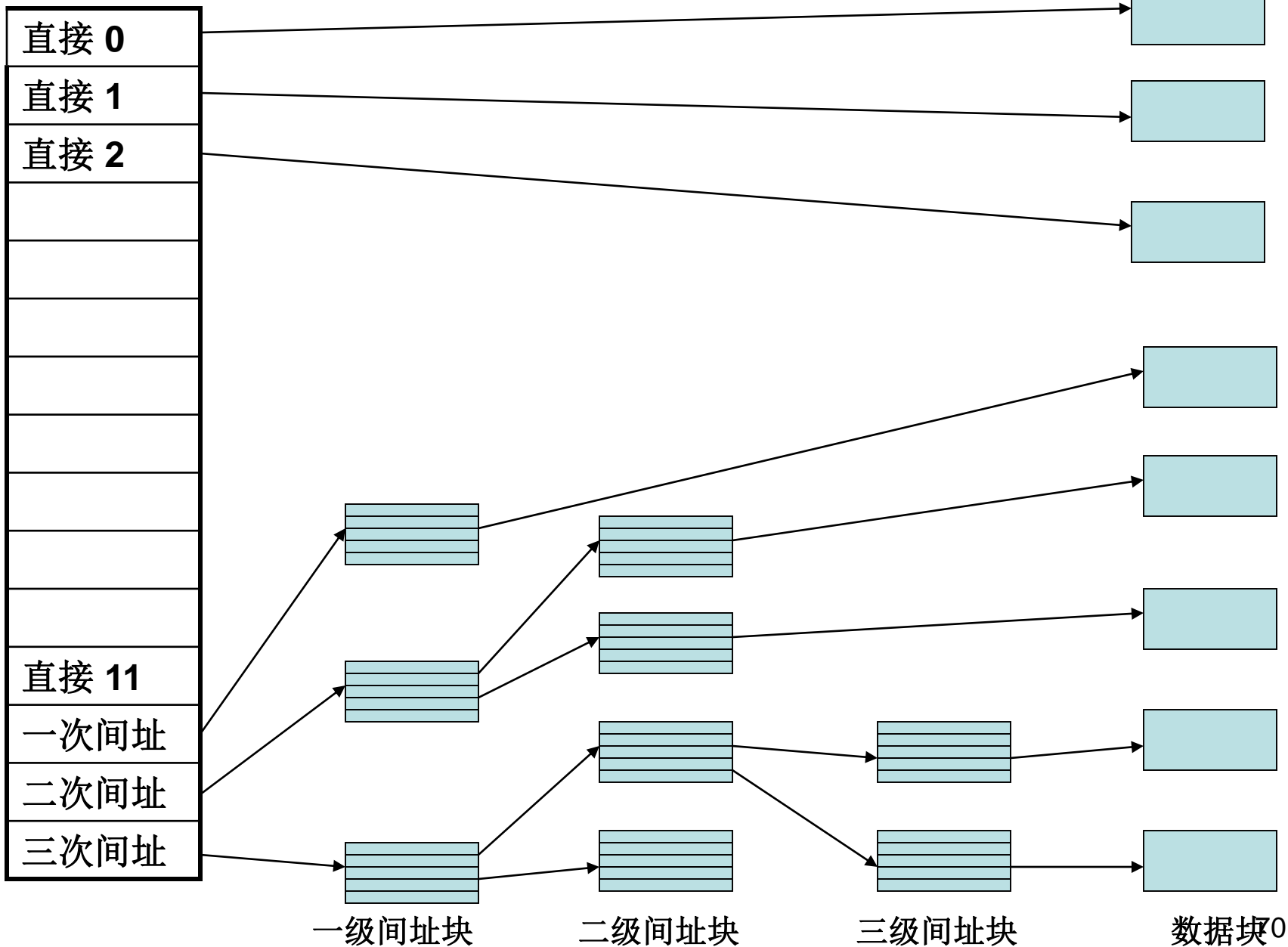
### 3、逻辑位移量到文件系统块的映射 **bmap**

进程用读写指针读写文件时，是以字节为单位来标识的；文件系统下层算法是以数据块来读写数据的，因此必须要把字符偏移量转换成块位移量。

文件占用的数据块在**icommon**的数据块索引表中标识。

由字节偏移量除以块大小得到块偏移量，判断块偏移量在直接索引表中还是间接表中，如果在直接索引表中，则直接得到块号；如果在间接索引表中，则需先读入间接块，并计算在间接块中的下标，再得到数据块的块号。

# 数据块索引表



## 算法 bmap

输入： (1) 索引节点号

(2) 字节偏移量

输出： (1) 文件系统中的块号

(2) 块中的字节偏移量

(3) 块中的I/O字节数

(4) 提前读块号

{

由字节偏移量计算出在文件中的逻辑块号；

为I/O计算出块中的起始字节； /\* 输出2 \*/

计算出拷贝给用户的字节数； /\* 输出3 \*/

检查是否可用提前读并标记索引节点； /\* 输出4 \*/

决定间接级；

/\* 接下页 \*/



**while**（没有在必须的间接级上）

{

从文件中的逻辑块号计算索引节点中或间接块中的下标；

从索引节点或间接块上得到磁盘块号；

如果需要，应释放先前读的磁盘块缓冲区（**brelease**）；

**if**（再也没有间接级了）

**return**（块号）；

读间接磁盘块（**bread**）；

按照间接级调节文件中的逻辑块号；

}

}

## 4、把路径名转换为索引节点 **namei**

用户对文件的操作通常是采用带路径的文件名的形式来进行。需要把带路径的文件名转换为文件的索引节点。

在**namei**中，从第一个路径名分量开始，逐个路径名分量进行查找，在指定的目录中从头至尾在各目录项中顺序匹配要查找的路径名分量。

找到相应的目录项后，取出其中的i节点号，进入下一级目录查找再下一个分量，直到最后一个路径名分量。

算法 **namei**

输入：路径名

输出：上了锁的索引节点

```
{  
    if（路径名从根开始）  
        工作索引节点=根索引节点（iget）；  
    else  
        工作索引节点=当前目录索引节点（iget）；  
    while（还有路径名）  
    {  
        从输入读下一个路径名分量；  
        验证工作索引节点确是目录，存取权限OK；  
        if（工作索引节点为根且分量为“..”）  
            continue； /* 循环回到while */  
        通过重复使用算法bmap、bread和brelse来读目录（工作索引节点）；  
    }  
    /* 接下页 */
```

```
if (分量与工作索引节点中的一个登记项匹配)
{
    得到匹配分量的索引节点号;
    释放工作索引节点 (input);
    工作索引节点=匹配分量的索引节点 (iget);
}
else /* 目录中无此分量 */
    return (无此索引节点);
}
return (工作索引节点);
}
```

## 5、给新文件分配磁盘i节点 ialloc

①、根据预分配的i节点号，实际分配一个i节点(**icommon**)。预分配的i节点是上一次分配的i节点后面的一个i节点(与预读数据块具有相似的原理)。

②、如果分配不成功，则采用下面的算法快速扫描一遍部分柱面组，寻找空闲i节点。具体算法是：

从预分配i节点所在柱面组的下一个柱面组开始寻找，如果没有找到，就加上一个偏移量后到另外一个柱面组中去寻找；如果还不成功，又再加上一个偏移量到另外一个柱面组中去寻找，如此进行下去。

第n次位移的位移量为 $2^n$ ，单位为柱面组，终止条件是 $2^n$ 大于等于文件系统的最大柱面组数，即：

$$E_{\max} = \log_2(NCG)$$

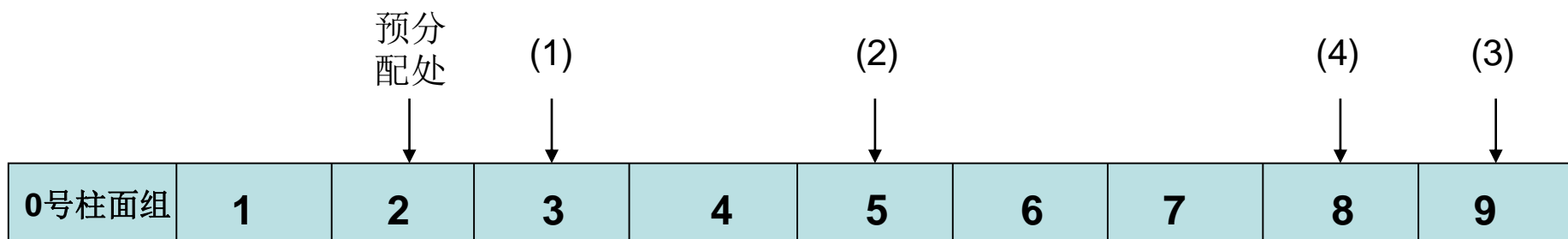
$E_{\max}$ 取最大整数。

如果第 $i$ 次位移时当前柱面组号加上 $2^n$ 后大于最大柱面组号，则把第 $i$ 次位移点取为当前柱面组号加上 $2^n$ 后与最大柱面组号的差值。

③、如果上述快速扫描完成后仍然没有找到空闲的 $i$ 节点，则从预分配点所在柱面组后面的第二个柱面组开始，依次逐个地在各个柱面组中寻找空闲 $i$ 节点。

④、如果到达最大柱面组处时还未找到，就再从0号柱面组开始到预分配点所在柱面组为止寻找空闲 $i$ 节点。

⑤、如果还未找到，表明该文件系统中已没有空闲的 $i$ 节点可供分配了。



预分配处： 2号柱面组

(1) 开始（第0次位移）： $2+2^0=3$

(2)  $3+2^1=5$

(3)  $5+2^2=9$

(4)  $9+2^3=17$ ,  $17-9=8$

(5)  $8+2^4=24$ ,  $24-9=13$ ,  $13-9=4$ , 但是  $2^4>9$ , 故终止;

再从4号柱面组开始向后顺序查找。如果到9号柱面组还未找到，则查找0号和1号柱面组。

## 说明：

- ①、这种分配也适合分配磁盘块，实际上**ialloc**和**alloc**合用同一个程序；
- ②、本算法在一级存贮结构的文件系统中也适用，只是将上述的柱面组换成存放硬盘*i*节点表的各项盘块即可；
- ③、具体在某一个柱面组中寻找空闲*i*节点时，是通过查看位示图来进行的；
- ④、这种算法使得各柱面组中的空闲*i*节点数分布比较均匀，因为起始点（即预分配的*i*节点所在的柱面组）是随机任意的（或相邻磁道）。



## 6、释放磁盘i节点 ifree

当删除一个文件时，需要把该文件对应的**icommon**释放掉。

核心首先释放该**icommon**中数据块索引表中标明的数据块，修改超级块中的空闲数据块数，以及数据块位示图；

再修改超级块中的空闲i节点数和i节点位示图。

# 第五章 文件系统的系统调用

本章主要介绍针对上层用户使用的系统调用。用户通过使用本章介绍的系统调用来申请操作系统中有关文件和文件系统操作的各项功能。

本章介绍的算法是基于第四章所介绍的底层文件系统算法之上的，主要包括七大类操作：

返回文件描述符类操作

路径名转换类操作

分配索引节点类操作

文件属性类操作

文件输入输出类操作

文件系统装卸类操作

文件系统目录树操作

系 统 调 用						
返回文件描述符	使用namei	分配索引节点	文件属性	文件I/O	文件系统结构	目录树操作
open    dup	open    stat    creat	creat	chown	read	mount	chdir
creat   pipe	link   chdir   unlink	mknod	chmod	write	umount	chroot
close	chroot   mknod   chown	link	stat	stat		
	mount   chmod   umount	unlink				
	底层文件系统算法					
	namei	ialloc    ifree		alloc free bmap		
	iget            iput					
	缓冲区分配算法					
	getblk            brelse            bread            breada            bwrite					

## 1、算法 **open**

输入：文件名

打开文件类型

文件许可权方式（对以创建方式打开而言）

输出：文件描述符

{

将文件名转换为索引节点（算法**namei**）；

**if** (文件不存在或不允许存取)

**return** (错)；

为索引节点分配系统打开文件表项，设置引用计数和偏移量；

分配用户文件描述符表项，将指针指向系统打开文件表项；

**if** （打开的类型规定清除文件）

释放占用的所有文件系统块（算法**free**）；

解锁（索引节点）； /\* 在上面的**namei**算法中上了锁 \*/

**return** （用户文件描述符）；

}

假定一个进程执行下列代码：

```
fd1=open("/etc/passwd",O_RDONLY);
```

```
fd2=open("local",O_WRONLY);
```

```
fd3=open("/etc/passwd",O_RDWR);
```

下图为打开文件后的数据结构：

用户文件描述符表

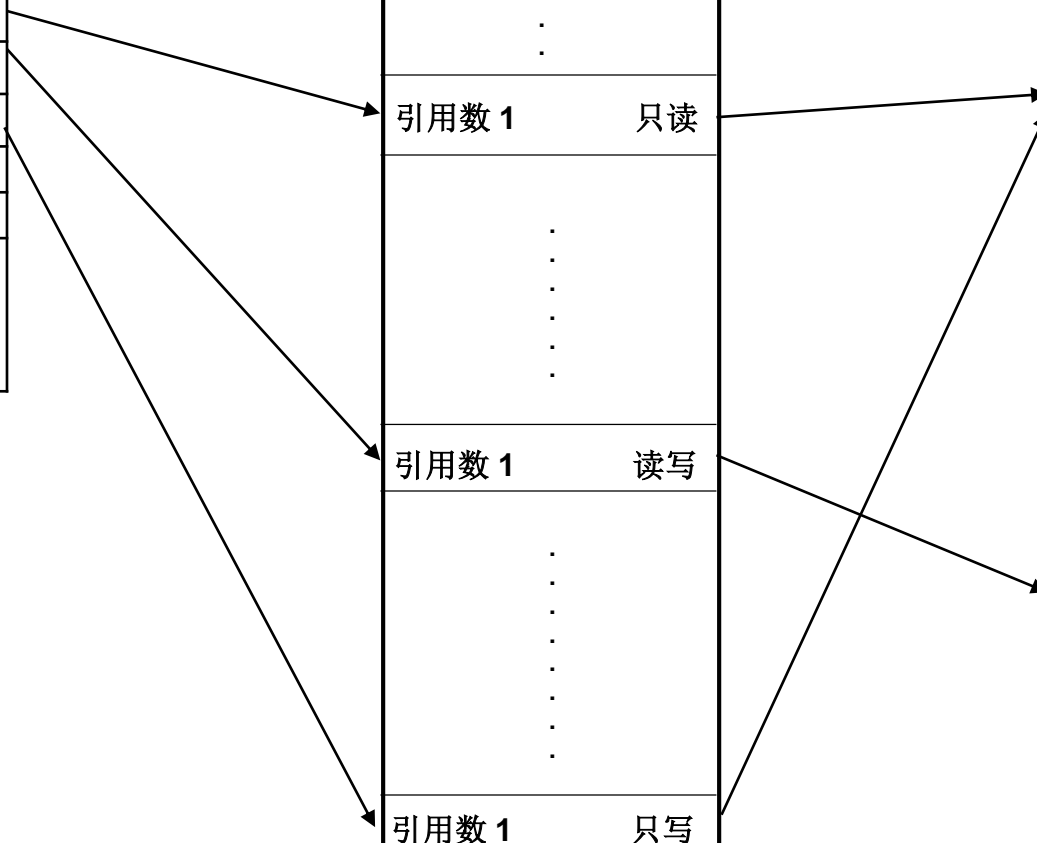
0	
1	
2	
3	
4	
5	
6	
7	
	⋮

系统打开文件表

	⋮
引用数 1	只读
	⋮
引用数 1	读写
	⋮
引用数 1	只写
	⋮

活动 inode 表

																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																										</
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----



假定第二个进程执行下列代码：

```
fd1 = open( “/etc/passwd” , 0_RDONLY) ;  
fd2 = open( “private” , 0_RDONLY) ;
```

则相关的数据结构如下：

用户文件描述符表

A进程

0	
1	
2	
3	
4	
5	
	⋮

B进程

0	
1	
2	
3	
4	
5	
	⋮

系统打开文件表

⋮	
引用数 1	只读
⋮	
引用数 1	读写
⋮	
引用数 1	只读
⋮	
引用数 1	只写
⋮	
引用数 1	只读

活动 inode表

⋮
引用数3 /etc/passwd
⋮
引用数1 local
⋮
引用数1 private
⋮



## 2、算法 read

系统调用read的语法格式如下：

```
number = read (fd, buffer, count);
```

fd 是由open返回的文件描述符

buffer 是用户进程中的用于保存数据的缓冲区地址

count 是用户要读的字节数

number 是实际读出的字节数

在u区中保存的I/O参数为

方式	指示读或写
计数	读或写的字节数
偏移量	文件中的字节偏移量
地址	拷贝数据的目的地址，在用户或核心存储器中
标志	指出地址是在用户空间还是核心空间

算法 read

输入：用户文件描述符

用户进程中的缓冲区地址

要读的字节数

输出：拷贝到用户区的字节数

{

由用户文件描述符得到系统打开文件表项（file表项）；

检查文件的可存取性；

在u区中设置用户地址、字节计数、输入/输出到用户的参数；

从file表项找到索引节点；

索引节点上锁；

用file表项中的偏移量设置u区中的字节偏移量；

（接下页）

**while** (要读的字节数还不满足)

 $\{$ 

将文件偏移量转换为文件系统块号（算法bmap）；

计算块中的偏移量和要读的字节数;

```
if (要读的字节数为0)          /* 企图读文件尾 */
```

```
break; /* 退出循环 */
```

读文件块(如果要预读, 用**breada**, 否则用**bread**);

将数据从系统缓冲区拷贝到用户地址:

修改u区中的字节偏移量、读计数、再写的用户空间地址;

释放缓冲区;                /\* 在bread中上了锁 \*/

}

## 解锁索引节点;

修改file表中的偏移量，用作下次读操作；

```
return(已读的总字节数);
```

}

# 读文件程序实例1

```
#include <fcntl.h>
main( )
{
    int  fd;
    char lilbuf[20], bigbuf[1024];

    fd = open( “/etc/passwd” , O_RDONLY);
    read(fd, lilbuf, 20);
    read(fd, bigbuf, 1024);
    read(fd, lilbuf, 20);
}
```

```
#include <fcntl.h>
main( )    /* 进程A */
{
    int fd;
    char buf[512];
    fd = open("/etc/passwd", O_RDONLY);
    read(fd, buf, sizeof(buf));          /* read1 */
    read(fd, buf, sizeof(buf));          /* read2 */
}
```

```
main( )    /* 进程B */
{
    int fd, i;
    char buf[512];
    for(i=0; i<sizeof(buf); i++)
        buf[i] = 'a';
    fd = open("/etc/passwd", O_WRONLY);
    write(fd, buf, sizeof(buf));          /* write1 */
    write(fd, buf, sizeof(buf));          /* write2 */
}
```

### 3、文件I/O位置调整 lseek

`position = lseek (fd, offset, reference);`

其中：fd —— 文件描述符

offset —— 字节偏移量

reference —— 偏移参照点：

- 0：从文件头开始
- 1：从当前位置开始
- 2：从文件尾开始

下图为lseek应用实例：

```

#include <fcntl.h>
main(int argc, char *argv[])
{
    int fd, skval;
    char c;
    if(argc != 2)
        exit();
    fd = open(argv[1], O_RDONLY);
    if(fd == -1)
        exit();
    while((skval = read((fd, &c, 1)) == 1)
    {
        printf( "char %c\n" , c);
        skval = lseek(fd, 1023L, 1);
        printf( "new seek val %d\n" , skval);
    }
}

```

## 4、读取索引节点状态参数 stat/fstat

系统调用stat和fstat永续进程查询文件的状态，它们返回诸如文件类型、文件所有者、存取权限、文件大小、链接数目、索引节点号、文件的访问时间等信息：

```
stat(pathname, statbuffer);  
fstat(fd, statbuffer);
```

其中pathname是文件名；fd是文件描述符；statbuffer是用户进程中的一个类别为stat的数据结构，在系统调用完成时用于存放返回的文件状态信息。



数据结构 **stat** 的定义:

```
struct  stat  {  
    dev_t    st_dev;  
    ino_t     st_ino;  
    ushort   st_mode;  
    short     st_nlink;  
    short     st_uid;  
    short     st_gid;  
    dev_t     st_rdev;  
    off_t     st_size;  
    time_t    st_atime;  
    time_t    st_mtime;  
    time_t    st_ctime;  
    long      st_blksize;  
    long      st_blocks;  
}
```

文件所在的设备号

文件的I节点号

读写保护模式

文件的链接数

用户标识

组标识

文件的起始位置

文件大小

最近访问时间

最近修改时间

最近（状态）改变时间

文件块大小

文件所用块数

实例：

打印一个文件的i节点号、文件名和文件大小（类似于ls命令的功能）：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
main(int argc, char *argv[ ])
{
    struct stat mystat;
    stat(argv[1], &mystat);
    printf(“%d  %s  %d\n”, mystat.st_ino, *argv[1],
           mystat.st_size);
}
```

## 5、建立无名管道 `pipe`

`pipe(fdptr)`

`fdptr[0]`      读管道指针

`fdptr[1]`      写管道指针

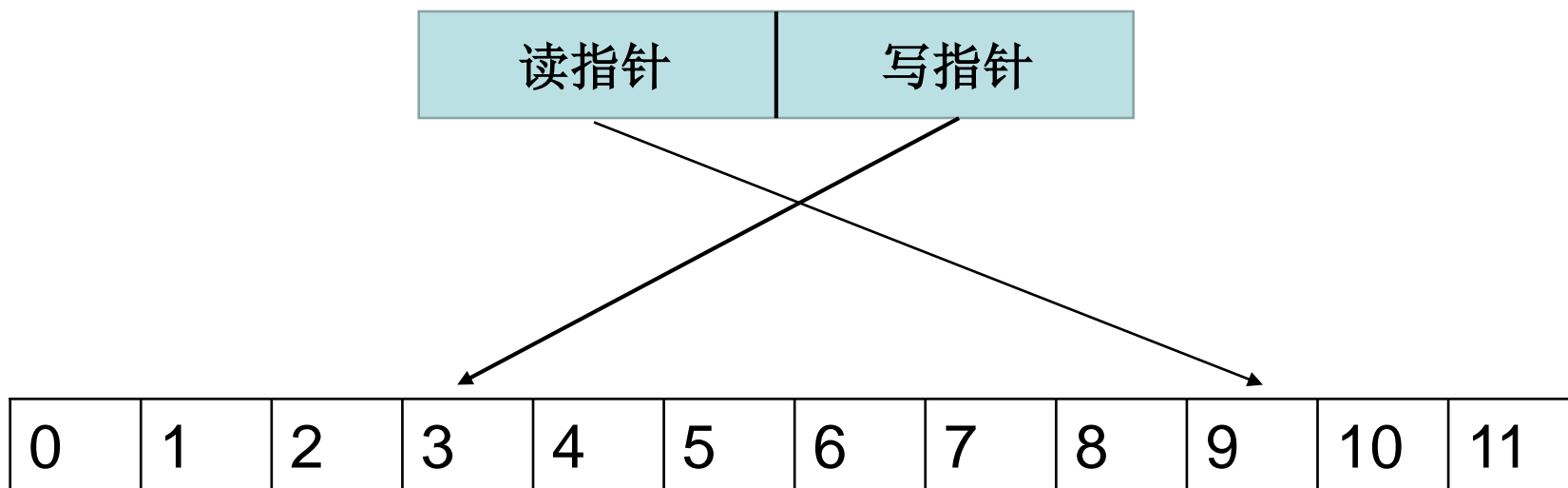


无名管道只能是建立管道的进程的子进程，才能共享无名管道。

`mknod("pipe_name", rw_mode)`      建立有名管道

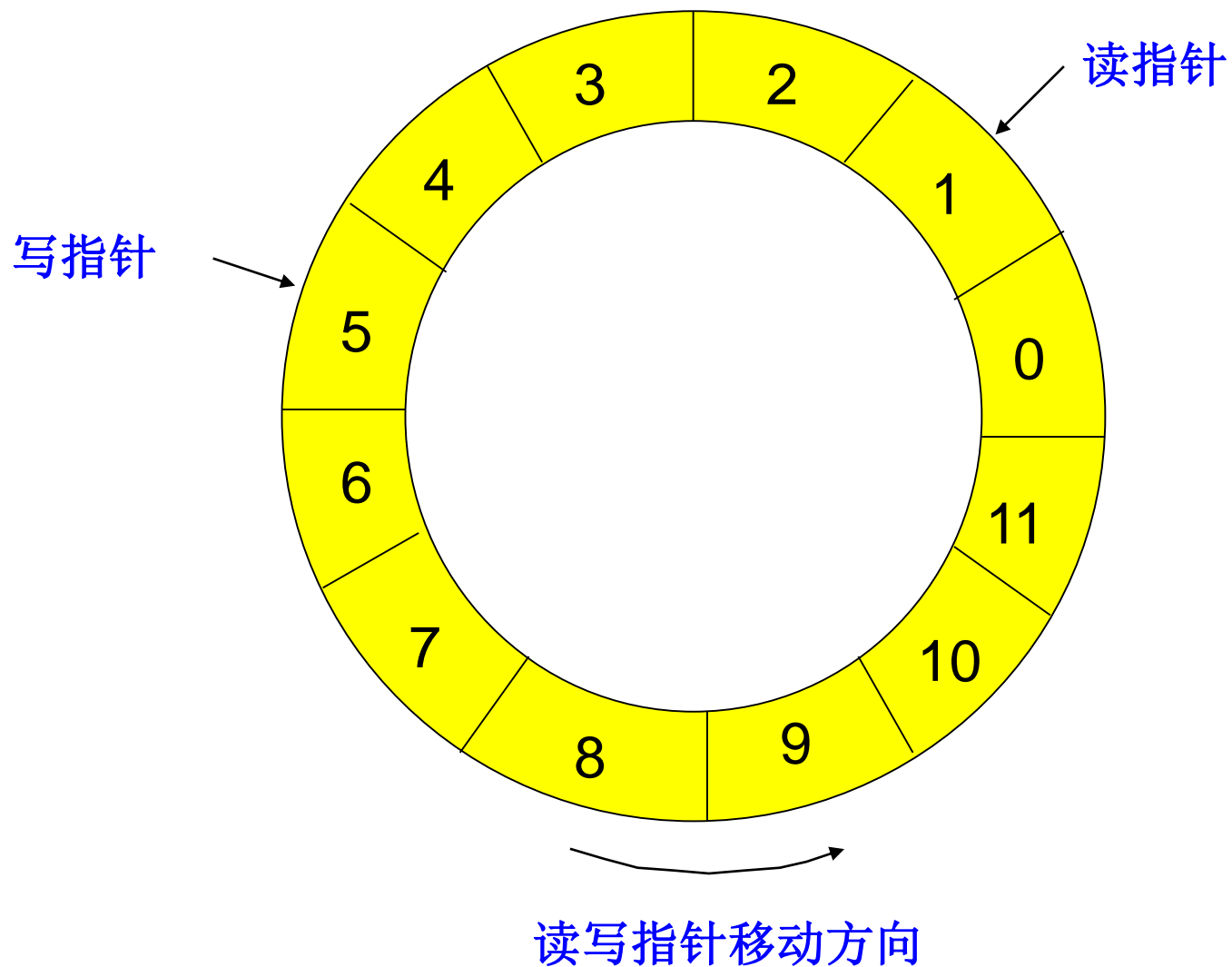
使用有名管道的进程间可以没有任何父子关系。

## 管道的结构和读写指针（1）



索引节点的直接索引块

## 管道的结构和读写指针（2）



## 管道读写的四种典型状况：

- (1)、写管道，管道中有空闲的存储空间满足写操作；
- (2)、读管道，管道中有足够的数据满足读操作；
- (3)、读管道，管道中没有足够的数据满足读操作；
- (4)、写管道，管道中没有足够的空闲空间存放数据。

## (1)、写管道，管道中有空闲的存储空间满足写操作

进程向管道中写数据。当写指针指向到一个间接块时（即超出数据索引表中的直接块索引表的大小），则写指针被重新置为0。因为核心能够确定，只要写指针没有超过读指针，则数据就不会超出管道的容量。

写操作完成后，核心唤醒所有等待从该管道中读数据的进程。

## (2)、读管道，管道中有足够的数据满足读操作

进程从管道中读取数据。读操作完成后，核心要唤醒所有等待着向管道中写数据的进程。同时，把读指针放到活动**inode**表中。为什么不像普通进程那样把读写指针放到系统打开文件表**file**表中？

因为写进程必须要知道读进程读到管道的哪里了  
读进程必须要知道写进程写到管道的哪里了



### (3)、读管道，管道中没有足够的数据满足读操作

进程读取管道中的全部数据，并成功返回，虽未完全满足读取数量。

如管道为空，通常进程进入睡眠状态，等待“管道中有数据可读”这个时间发生时被唤醒，并与可能存在的其他读进程竞争。

#### (4)、写管道，管道中没有足够的空间满足写操作

进程写数据到管道中，直到管道被写满为止，并进入睡眠状态，等待其他的读进程读取数据后，使管道中腾出空间。

当另一个读进程从管道中读取数据后，会唤醒包括本进程在内的所有写进程，本进程再次有机会把未写完的数据写入管道。

## 6、复制用户文件描述符 `dup`

`dup`把一个文件描述符复制到本用户的打开文件表中从头开始的第一个可用的空表项中。新的表项（文件描述符）由 `dup` 返回。

由于文件描述符被复制，因此**两个**文件描述符指向**同一个**系统打开文件表项，即指向同一个文件读写指针，该读写指针的引用计数加一。

## dup的应用举例：标准输出重定向

command > abc

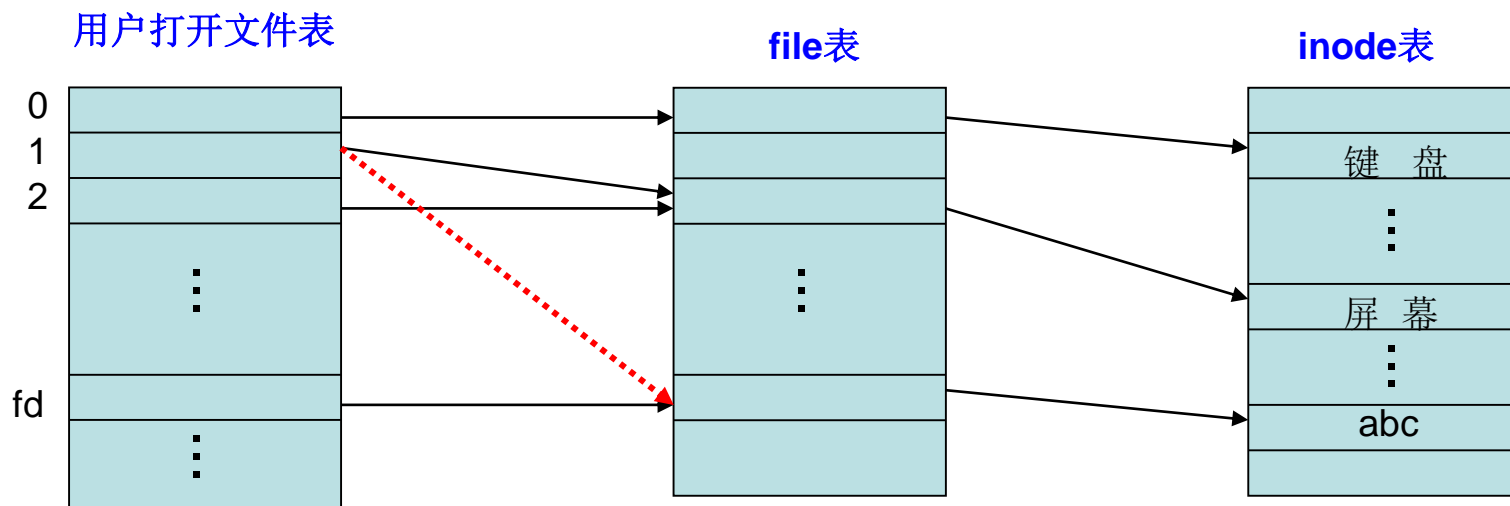
①、fd = open("abc", "w")

②、close(1)

③、dup(fd)

④、close(fd)

⑤、command



## 7、安装文件系统 **mount**

只有超级用户可以安装和拆卸文件系统

安装点目录**通常**为空目录

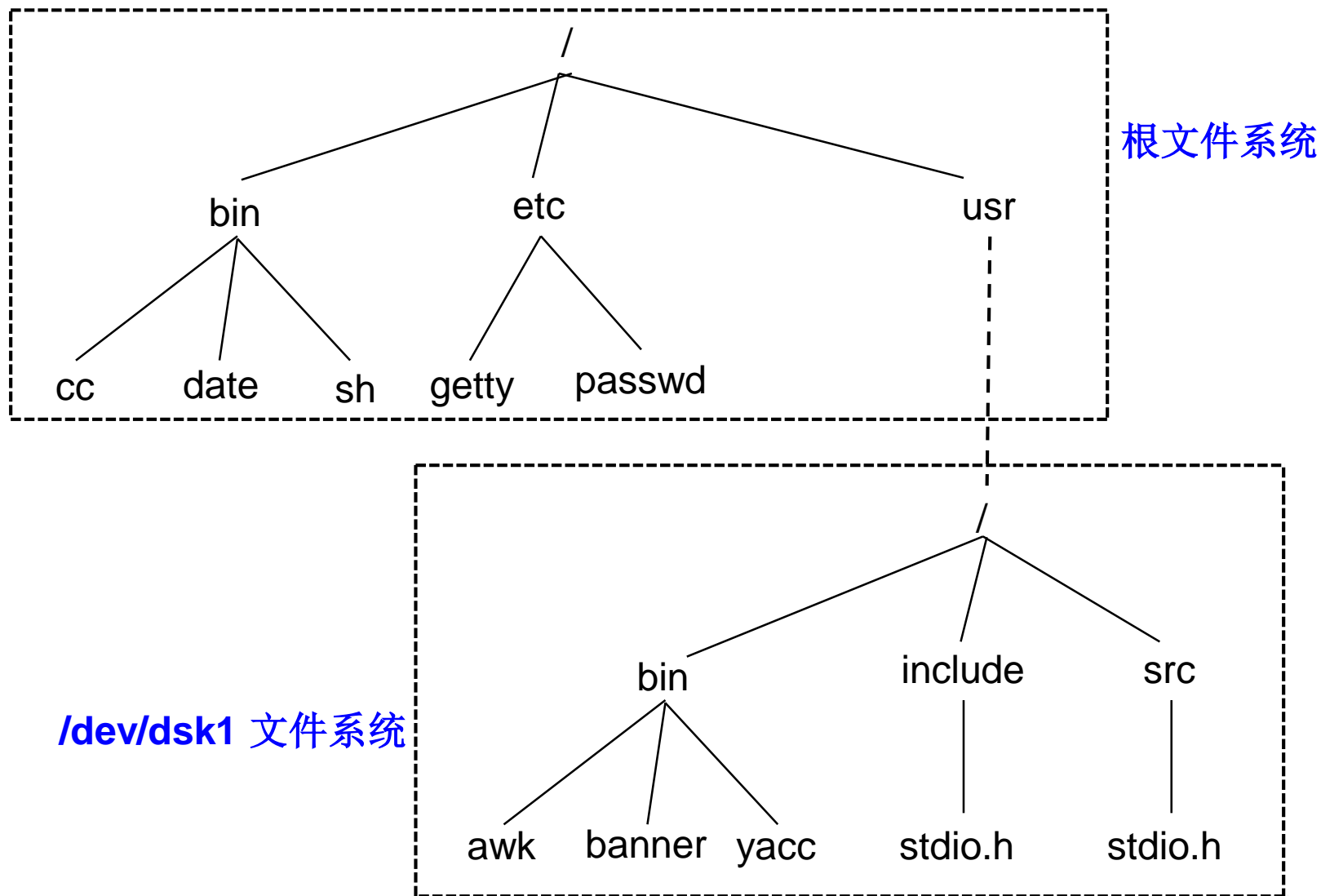
安装点的引用计数**只能为一**

在**mount**表中占用一个表项，保持了安装点和被安装文件系统根节点之间的对应关系

读入被安装文件系统的超级块，并保持指向超级块的指针

例如： **mount("/dev/dsk1", "/usr", 0)**

把逻辑设备（子文件系统）**/dev/dsk1**以可读可写的方式安装到目录**/usr**下面。



把子文件系统 **`/dev/dsk1`** 安装到 **`/usr`** 目录下

## 活动i 节点表

安装到的索引节点 标志为安装点 引用数为1
设备索引节点 空闲 引用数为0
被安装子文件系统的 根索引节点 引用数为1

## mount表

超级块 安装点索引节点 根索引节点

缓冲区

mount执行后的相关数据结构

算法 mount

输入：块特殊文件的文件名

安装点的目录名

选择项（只读）

输出：无

{

if（非超级用户）

return（错）；

取块特殊文件的索引节点（算法namei）；

合法性检查；

取安装点目录名的索引节点（算法namei）；

if（不是目录，或引用数大于1）

{

释放索引节点（算法input）；

return（错）；

}

（接下页）



（ 接上页 ）

查找安装表中的空项；

调用块设备驱动程序的open子程序；

从高速缓冲中取空闲缓冲区；

将超级块读入空闲缓冲区；

初始化超级块中的各个域；

取被安装设备的根索引节点，并保存在安装表中；

标记安装点的目录索引节点为安装点；

释放特殊文件的索引节点（算法input）；

解锁安装点目录的索引节点；

}

进程在存取一个目录节点时，可能遇到该目录是一个安装点，这时就要从安装点所在文件系统跨越到被安装子文件系统中，这种跨越是在检查到该目录节点上有“安装点”标志时进行。

包含这种跨越动作的常用系统调用包含：

分配内存活动i节点算法 **iget**

把路径名转换为索引节点算法 **namei**

## 8、删除一个目录项 `unlink`

`unlink(pathname);`

该系统调用删除一个名为pathname的目录项。如果该目录项是该文件的最后一个链接，则核心删除该文件的索引节点，并释放文件的数据块。如果该文件有多个链接，则其他文件名仍能正常存取该文件。

下面为unlink的算法：

```

算法  unlink
输入： 文件名
输出： 无
{
    取要删除的文件的父目录的索引节点（算法namei）；
    if （文件名的最后分量是“.”）
        父目录索引节点的引用计数加1；
    else
        取要被删除的文件的索引节点（算法iget）；
    if （要删除的是目录的链接项，但用户不是超级用户）
    {
        释放索引节点（算法iput）；
        return（错）；
    }
    if （要删除的是共享正文文件，且链接数为1）
        从区表中清除；
    写父目录：将被删除文件的索引节点号置为0；
    释放父目录的索引节点（算法iput）；
    文件链接数减1；
    释放文件的索引节点（算法iput）；
    /* iput检查链接数是否为0；如果是，
     * 则释放文件的数据块（算法free）
     * 并释放磁盘索引节点 icommon
     */
}

```

## 系统调用 `unlink` 的特殊应用

当一个进程使一个文件处于打开状态时，另一个进程可能在该文件打开期间删除该文件（甚至做这件事的进程本身就是发出系统调用`open`的进程）。

第一个进程除非关闭这个文件，否则就可一直读写该文件。而其他进程因为该文件已被删除而无法访问该文件。

——安全无痕地使用文件的方法！

## 9、文件系统管理 **fsck**

**fsck** 命令通常由具有超级用户权限的系统管理员执行, 用于检测和修复文件系统的错误. 运行时显示如下过程信息:

### **\*\* Phase 1 – Check Blocks and Sizes**

检查索引节点表中文件大小和所用块数

### **\*\* Phase 2 – Check Pathnames**

检查目录和文件路径的正确性

### **\*\* Phase 3 – Check Connectivity**

检查各目录之间的联结关系

### **\*\* Phase 4 – Check Reference Counts**

检查各文件的引用计数

### **\*\* Phase 5 – Check Free List**

检查文件系统的空闲块表

# 第六章 进程结构

## 1、进程的状态和状态的转换

进程的基本状态可分为：

运行态    就绪态    睡眠态

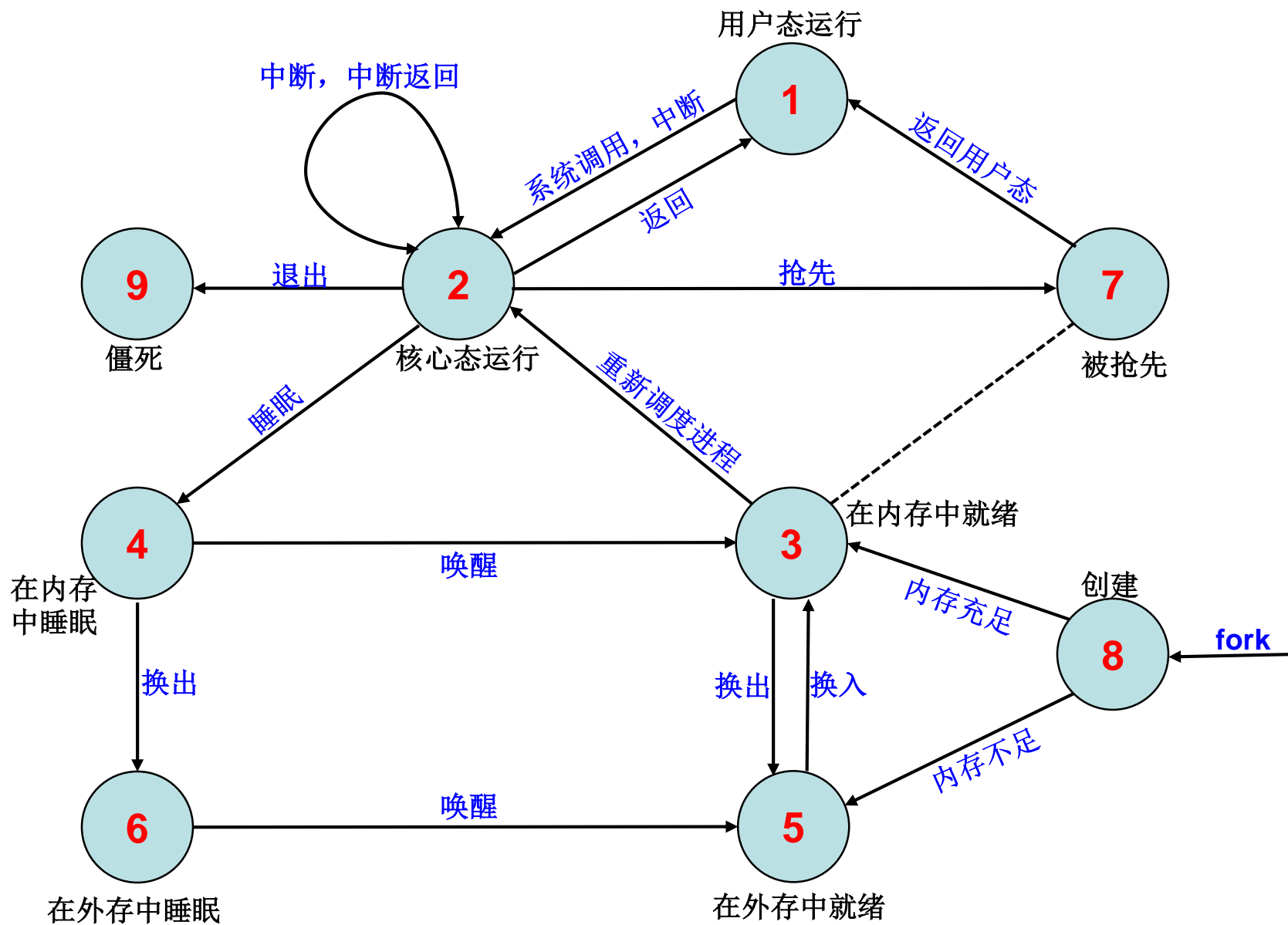
进一步细分，又可分为九种状态：

- ①、进程在用户态下执行；
- ②、进程在核心态下执行；
- ③、进程已经准备好运行，在内存中就绪；
- ④、进程等待资源，在内存中睡眠；
- ⑤、进程处于就绪状态，因内存不足，被放在交换区上等待；

- ⑥、进程睡眠等待资源，因内存不足，被换到交换区上等待；
- ⑦、进程正从核心态返回用户态，但核心抢先于它做了上下文切换，以调度另外一个进程；
- ⑧、进程处于刚被创建的状态，此时进程既没有处于就绪状态，也没有进入睡眠状态；
- ⑨、进程执行了系统调用`exit`，处于僵死状态。此时进程刚消亡，并向父进程发送退出状态信息和计时统计信息。

进程在其生命周期中必然处在这九种状态之一，并且根据运行时间和条件的变化，在这九种状态之间进行转换，这种有方向规定的状态转换路径，就构成了进程的状态转换图——有向图。





## 有关进程抢先:

——任何进程都不能抢先另一个在核心中运行的进程

(1)、对于一个在核心态下运行的进程，如果没有因等待资源而睡眠，或者被中断，或者执行完毕准备退出，则它永远占用处理机而不会发生上下文切换（调度其他进程）。

为什么？

## 有关进程抢先:

——任何进程都不能抢先另一个在核心中运行的进程

(2)、对于一个在用户态下运行的进程，也不会直接发生抢先。

如果该用户态下的进程运行时间较长，则在时钟中断处理程序（核心态）运行完毕、准备返回到用户态时发生抢先。

## 进程状态3与状态7的区别：

### 状态3 —— 在内存中就绪

通常（除创建状态外）进程是因执行系统调用（申请资源）进入睡眠后，又被唤醒后进入就绪状态。因此进程被重新调度后就进入状态2（核心态下）继续运行。

### 状态7 —— 被抢先

正在运行的进程从核心态正要返回到用户态时，因运行时间足够长了，发生进程切换，从而进入就绪状态。当重新被调度后，进入用户态继续执行。

### 直接效果 ——

一个是进入核心态运行；一个是进入用户态运行

## 2、进程的特征：

- ①、每个进程在核心进程表（**proc**数组）都占有一项，在其中记录了进程的状态信息；
- ②、每个进程都有一个“每进程数据区（**per process data area —— ppda**）”，保留相应进程更多的信息和核心栈；
- ③、处理机的全部工作就是在某个时候执行某个进程；
- ④、一个进程可生成或消灭另一进程；
- ⑤、一个进程中可申请并占有资源；
- ⑥、一个进程只能沿着一个特定的指令序列运行，不会跳转到另一个进程的指令序列中去，也不能访问别的进程的数据和堆栈。（抗病毒传播的重要原因之一）

### 3、进程的解释

在**UNIX**系统中进程的概念包含什么意义？

#### 在较高级的方面

进程是一个重要的组织概念。可以把计算机系统看作是若干进程组合的活动。进程是系统中活动的实体，它可以生成和消灭，申请和释放资源，可以相互合作和竞争，而真正活动的部件如处理机和外部设备则是看不见的。

#### 在较低级方面

进程是不活动的实体，而处理机则是活动的，处理机的任务就是对进程进行操作，处理机在各个进程映像之间转换。

## 4、进程映像

进程映像虽然包括很多方面，但关键问题是存储器映像。当一个进程暂时退出处理机时，它的处理机映像（也就是各个寄存器的值）就成为存储器映像中的一部分。当该进程再次被调度使用处理机时，又从存储器映像中恢复处理机映像。

一般说来，进程的结构，即进程映像是指其存储器映像，由下列几部分组成：

- ① 进程控制块（PCB）
- ② 进程执行的程序，即共享正文段（TEXT）
- ③ 进程执行时所需要使用的数据，即数据段（DATA）
- ④ 进程执行时使用的工作区，即栈段

## 1)、进程控制块 **PCB**

**PCB**包括两部分信息：

一部分是不论进程当前是否在处理机上运行，系统都要查询和修改的一些控制信息，它们构成一个数据结构**proc**（进程基本控制块）。所有进程的**proc**结构就构成了核心“进程表”——“进程表”中每一项都是一个存放某进程控制信息的**proc**结构。

另一部分信息当进程不在处理机上运行时，系统不会对它们进行查询和处理，这些信息构成另一个数据结构**user**，它是对**proc**的扩充，称为“进程扩充控制块**user**结构”。



## 进程基本控制块 **PROC**

**struct proc {**

进程状态标志域

指向**U**区的指针

进程属主 **UID**

进程标识数 **PID**

睡眠地址

进程优先级等调度参数

软中断信号域

进程运行计时域，用于计算进程的优先级

**}**

## 进程扩充控制块 **USER**

**struct user {**

指向**proc**的指针

真正用户标识号和有效用户标识号

计时器域，用于统计汇总

指向软中断信号处理函数的指针

控制终端

错误标识域

系统调用返回值

用于读写的**I/O**参数

进程的当前目录和当前根

用户文件描述符表（用户打开文件表）

其它标识域

**}**

## 2)、共享正文段:

进程执行的程序用可再入码编写并可使若干个进程共享执行的部分构成共享正文段（纯正文段，不可修改段），它包括再入的程序和常数。

## 3)、数据段:

进程执行时用到的数据构成数据段。如果进程执行的某些程序为非共享程序，则它们也构成数据段的一部分。

## 4)、栈段:

进程在核心态下运行时的工作区为核心栈，在用户态下运行时的工作区为用户栈。在进行函数调用时，栈用于传递参数，保护现场，存放返回地址以及为局部动态变量提供存储区。

核心栈的结构和操作系统与用户栈相同，只是存放进程在核心态下调用函数时的有关信息。

## 5、进程映像存贮器中的分布：

存放进程映像的地方有两个：

①、内存

②、磁盘交换区——内存的扩充部分

进程的映像分为常驻内存部分和非常驻内存部分  
常驻内存部分：

进程基本控制块**proc**

进程共享正文段的控制信息**text**

非常驻内存部分：

进程扩充控制块**user**

进程在核心态下的工作区（核心栈）

数据段

进程在用户态下的工作区（用户栈）

共享正文段

处理机正在执行本进程时：

该进程映像全部（或所需部分）在内存中

处理机执行其它进程时：

该进程映像所占用的内存可能被分配给其它进程，如果是这样，则当前进程的映像要被调出到对换区中去，等到重新满足运行条件并获得处理机时，再次被调入内存运行。

**UNIX**系统中所采用的进程映像调入调出措施，可以大大提高内存的利用率，增加系统的吞吐量，以便同时为尽可能多的分时用户提供服务。

在进程映像占用的内存被分配给其他进程之前，不但该进程的程序和数据需要调出内存，该进程的控制信息也被调出内存。但为了该进程能够再次被调入内存，内存中需要保留一部分必要的信息，这就把**进程控制信息**也分成了常驻内存和非常驻内存两部分：

### 常驻内存控制信息块

是系统需要经常查询以及恢复整个进程映像时所不可缺少的信息。

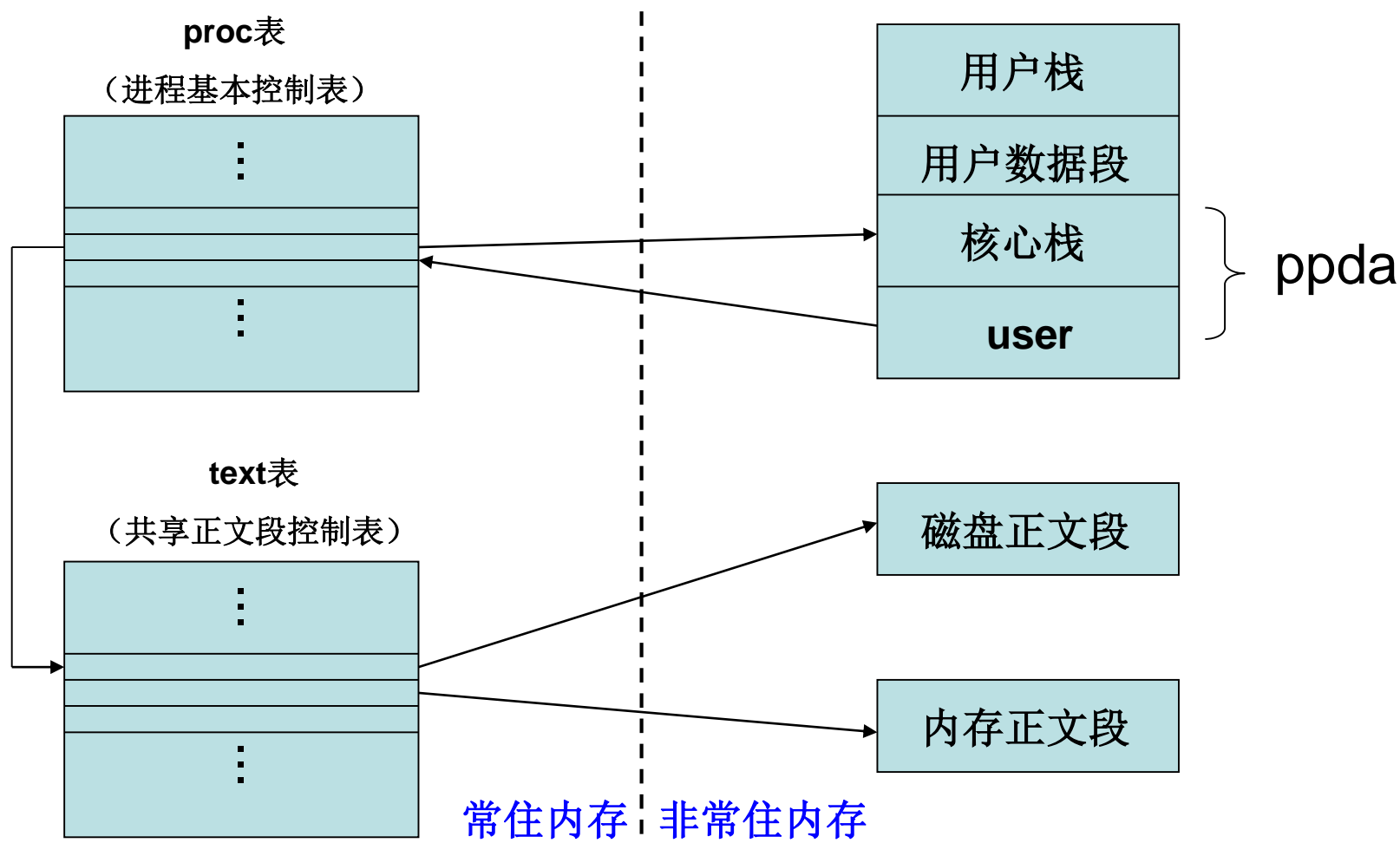
### 非常驻内存控制信息块

可以随进程状态的变化而在内外存之间交换的进程控制信息中的其余部分。

为了方便进程映像在内外之间交换，**UNIX**系统中把进程非常驻内存部分作为一个整体，占用连续的存贮区，其顺序是：首先是**user**结构（进程扩充控制块）和核心栈，然后是数据段和用户栈。

进程**user**结构和核心栈合并构成进程的“本进程数据区——**ppda**区（**per process data area**）。

**ppda**区以及用户数据段和用户栈，要调入内存则一起调入，要调出则一起调出。图示为进程映像的组织方式。



## 6、进程上下文（context）

一个进程的上下文包括五个方面：

- ①、被进程正文所定义的进程状态
- ②、进程所使用的全局变量和数据结构的值
- ③、机器寄存器的值
- ④、进程表项**proc**结构和**user**结构中的值
- ⑤、用户堆栈和核心堆栈中的值

“执行一个进程”——指系统在该进程的上下文中执行，也就是进程的上下文确定和限制了进程的运行环境和空间。

核心从一个进程转到另一个进程执行时，叫做“上下文切换”，也就是系统从一个进程上下文确定的环境换到另一个进程上下文确定的环境中去执行。并且保留前一个进程的必要信息，能够在以后需要时又再切换回前一个进程，并恢复它的执行。



## 相关说明：

①、用户态和核心态之间的转换是执行状态的转换而不是上下文的切换。

②、在遇到中断信号时，核心在**当前进程的上下文**中对中断服务，而不论该中断是否是本进程引起的。系统在核心态下对中断进行服务，而不是产生或调度一个特殊进程来处理中断。被中断的进程无论是在用户态下执行还是在核心态下执行，核心都保留必要的信息以便处理中断完后能恢复本进程的执行。

③、进程映像和进程上下文的区别

进程映像——主要是指进程的结构和内部组织形式

进程上下文——构成进程映像的各部分的各种取值的集合

## 7、进程状态及映像的对应关系

### ①、运行状态

此时进程正在占用处理机，进程的全部映像（或正在运行的部分）驻留在内存中，存储管理器（**MMU**）中装配的是本进程映象的地址映射参数，系统变量**U**（即**U**区）与本进程的**user**结构对应。

处理机可能是在执行用户进程本身，也可能是在执行系统程序，前一种情况称为进程在用户态下运行，后一种情况称为进程在核心态下运行。

在核心态下的进程虽然执行的是操作系统内部的程序，但由于所用的系统栈和进程控制块仍然是本进程的，所以此时进程可能是在调用系统调用，或是在进行中断处理，信号处理等。

## ②、就绪状态

处于就绪状态下的进程基本具备了运行条件，正在等待使用处理机。这时进程的映像可能全部在内存，也可能只有需要运行的部分在内存。**MMU**装配的不是本进程映像的地址映射参数，系统变量**U**也不与本进程的**user**结构对应。

系统中通常有若干的不同优先级的就绪队列，每个就绪队列中有若干的就绪进程在排队等待运行。

### ③、睡眠状态

进程不具备运行条件，需等待某种事件的发生，无法继续执行下去。此时进程的映像根据内存的空闲程度，可能全部在内存中、可能部分在内存中、也可能全部都在交换区上。

在程序实现上，造成进程睡眠的唯一直接原因是调用了**sleep()**函数，调用**sleep()**的原因有多种：

- a、进程在运行过程中要使用某种资源，如缓冲区，但由于该资源已被占用，不能立即得到满足，不得不进入睡眠状态，等待其它进程释放该资源。
- b、进程实施同步或互斥，如父进程对子进程进行跟踪时，子进程等待父进程发控制命令，或父进程等待子进程完成某一次操作等。
- c、等待输入/输出操作的完成
- d、进程完成某一项任务后，暂时停止自身的运行，等待新任务的来临，如0号进程或**shell**进程等。

## 8、睡眠与唤醒

一般而言，一个进程除非自己调用了**sleep**函数，否则是在**执行一个系统调用期间**进入**睡眠**的（**why?**）：

该进程执行一个操作系统陷入（**trap**），进入核心，然后可能进入睡眠等待某个资源（或软资源或硬资源）。

进程在一个事件上睡眠是它们处于睡眠状态，直到该事件发生。

当进程等待的事件发生时，等待该资源的进程就被**唤醒**并进入“就绪”状态，而不是直接进入“运行”状态。

**sleep**函数运行时完成下面三项主要工作：

- ①、将进程的状态标志设置为睡眠
- ②、记录下睡眠原因
- ③、修改进程优先级。这个优先级是进程下次被唤醒时所具有的竞争处理机的能力，它因不同的睡眠原因而异，因为不同的事件要求响应的紧急程度是不同的。

# 系统中各种睡眠原因相应的优先权

睡眠原因	睡眠优先数
进程对换	PSWP
<b>inode</b> 操作	PINOD
块设备操作	PRIBIO
	PRIUBA
	PIERO
管道操作	PPIPE
虚拟文件操作	PVFS
等待操作	PWAIT
上锁操作	PLOCK
资源等待	PSLEP
用户状态	PUSER
初始值	NZERO

优先数小，优先级高



优先数大，优先级低

## 睡眠事件及其地址：

“进程在一个事件上睡眠”或“进程睡眠等待一个资源”，就是等待处理该事件（或操作该资源）的程序代码段映射到本进程核心（因为是系统调用）地址空间中。

例如读写磁盘操作，当**A**进程正占用磁盘——执行磁盘驱动程序，设置磁道、扇区、读写数量和数据传输模式等参数，初始化磁盘缓冲区等操作时——**B**进程就不能再映射（执行）磁盘驱动程序的代码段，否则将破坏**A**进程设置的参数。

**A**进程设置完参数后，将睡眠等待（同步）或不等待（异步）物理磁盘的操作，磁盘操作完毕时，发出中断信号，由当前正在运行的程序执行磁盘中断处理程序，通知**A**进程接收数据，唤醒包括**B**进程在内的所有等待磁盘操作的进程。

进程

事件

地址

进程 a

进程 b

进程 c

进程 d

进程 e

进程 f

进程 g

进程 h

等待 I/O 完成

等待缓冲区

等待索引节点

等待终端输入

地址 A

地址 B

地址 C

睡眠在事件上的进程及映射到地址上的事件



## 9、中断处理

### 中断的分类：

- 硬件中断——来自时钟和各种外部设备
- 可编程中断——来自“软件中断”指令
- 例外中断——中断的特例，来自页面错误

### 核心处理中断的操作顺序：

- ① 保护现场 —— 保存当前进程的上下文
- ② 确定中断源 —— 根据中断向量查找中断处理程序
- ③ 调用中断处理程序 —— 完成处理任务
- ④ 中断返回 —— 恢复被中断程序的上下文

## 中断处理程序的算法

算法 **inthand**

输入：无

输出：无

{

保存当前上下文，并创建一个新的上下文层；

确定中断源，得到中断信号号；

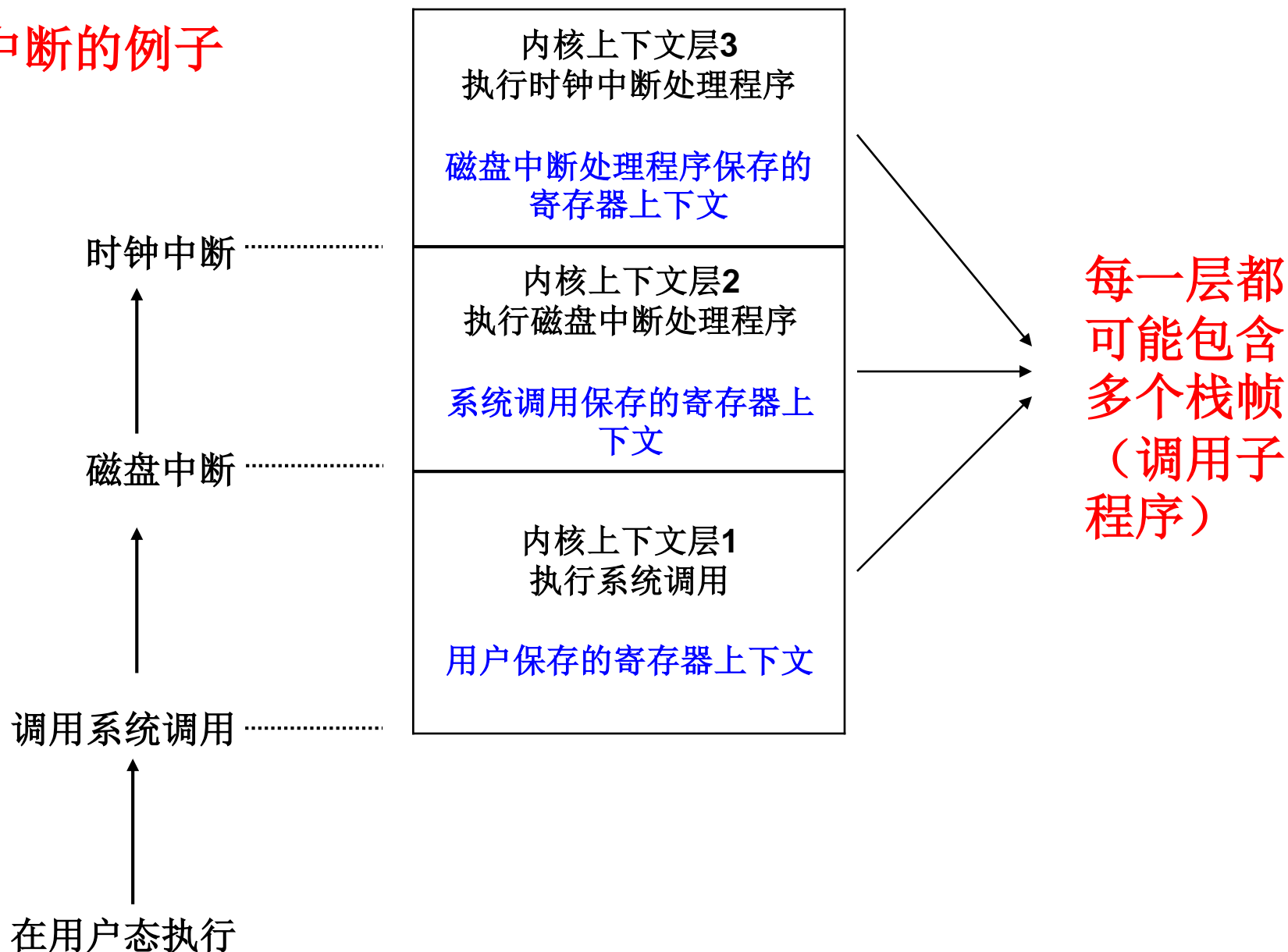
查找中断向量，找到中断处理程序入口；

调用中断处理程序，完成具体任务；

恢复前一个上下文，继续执行被中断程序；

}

## 中断的例子



## 10、系统调用接口

在为**UNIX**系统编写的**C**语言编译程序中，定义了一个与系统调用名字一一对应的函数库，使得应用程序能够执行系统调用。

库函数中包含一条操作系统陷入（**trap**）指令，把程序的运行模式由用户态转变为核心态。

每一个库函数在执行**trap**指令时，使用与机器有关的方式向核心传送一个操作系统陷入号，即**系统调用号**。

**C**编译程序中的库函数是在用户态下运行的，当库函数执行**trap**指令时，产生一个中断信号，调用中断处理程序处理中断请求（进入核心态，执行系统调用），中断向量就是系统调用号。

**系统调用接口是中断处理程序的特例！**

算法 **syscall** /\* 系统调用的算法 \*/

输入：系统调用号

输出：系统调用的结果

{

在系统调用表中找出对应于系统调用号的表项；

确定系统调用的参数数目；

将参数从用户地址空间拷贝到U区；

为废弃返回而保存当前上下文；

调用内核中的系统调用代码；

if (在系统调用的执行中有错)

{

将用户保存的寄存器上下文中的寄存器0设置为错误号；

将用户保存的寄存器上下文中的PS寄存器的进位位打开；

}

else

将用户保存的寄存器上下文中的寄存器0、1设置为系统调用的返回值；

}

数据寄存器

状态寄存器

### 系统调用实例分析:

```
char name[ ] = "file";
main()
{
    int fd;
    fd = creat(name, 0666);
}
```

#### # code for main

```
58: mov    &0x1b6, (%sp)
5e: mov    &0x204, -(%sp)
64: jsr     0x7a
```

#### # library code for creat

```
7a: movq    &0x8,%d0
7c: trap    &0x0
7e: bcc     &0x6 <86>
80: jmp     0x13c
86: rts
```

#### # library code for errors in system call

```
13c: mov     %d0,&0x20e
142: movq    &-0x1,%d0
144: mova    %d0,%a0
146: rts
```

### 更一般的用法:

```
char name[ ] = "file";
main()
{
    int fd;
    if ((fd = creat(name,0666)) < 0)
        printf(错误信息);
}
```

# 将0666压到堆栈上  
# 把变量name压到堆栈上  
# 调用creat的C语言库程序

# 将creat的陷入号8移到数据寄存器0  
# 操作系统陷入，调用8号中断处理程序  
# 如果进位位为零（正确），则转到地址86  
# 否则（错误），转到地址13c  
# 从子程序creat返回到main函数

# 把寄存器0中的错误号移到20e单元（变量errno）  
# 将常数-1放到数据寄存器0中，置错误标志

# 从子程序creat返回到main函数

## 算法 **fork**

输入：无

输出：对父进程是子进程的**PID**

对子进程是**0**

{

检查可用的核心资源（如内外存空间、页表等）；

取一个空闲的进程表项和唯一的**PID**号；

检查用户没有过多的运行进程；

将子进程的状态设为“创建”状态；

将父进程的进程表项中的数据拷贝到子进程的进程表项中；

当前目录的索引节点和改变的根目录的引用计数加一；

系统打开文件表中相关表项的引用计数加一；

在内存中作父进程上下文的拷贝（包括**u**区、正文、数据、堆栈）；

## 算法 fork （续）

在子进程的系统级上下文中压入虚设的系统级上下文层

/\* 虚设的上下文层中含有使子进程能够识别自己的数据，

\* 并使子进程被调度时从这里开始

\*/

if (正在执行的进程是父进程)

{

    将子进程的状态设置为“就绪”状态；

    return（子进程的PID）；      /\* 从系统态到用户态 \*/

}

else /\* 当前正在执行的进程是子进程 \*/

{

    初始化u区的计时域；

    return（0）；      /\* 从系统态返回到用户态 \*/

}

}



## 实例1：双进程文件拷贝

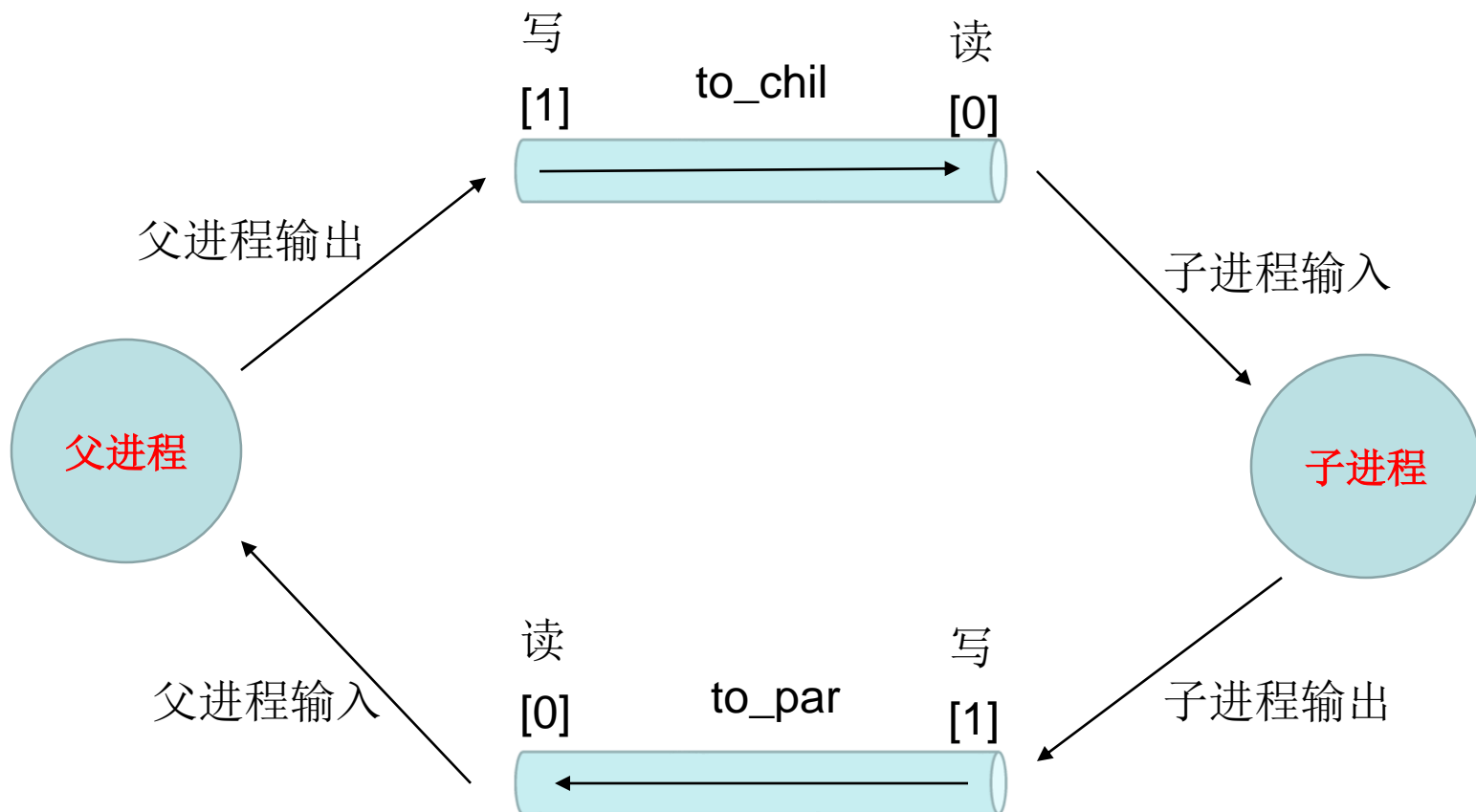
两个进程共享读指针和写指针，但运行的结果并不能保证新文件中的内容与老文件中的完全相同，这取决于父子两个进程的调度顺序。

—— 多进程环境下的数据一致性问题。

```
main(int argc, *argv[ ])
{
    fdrd = open(argv[1], O_RDONLY);
    fdwt = create(argv[2], 0666);
    fork( );
    /* 父子进程执行下面相同的代码 */
    rdwrt( );
    exit(0);
}

rdwrt( )
{
    for(;;)
    {
        if(read(fdrd, &c, 1) != 1)
            return;
        write(fdwt, &c, 1);
    }
}
```

## 实例2：确保数据一致性的双进程数据传输



在本例中，父子进程分别把自己的标准输入和标准输出重新定向到**to\_chil**和**to\_par**管道中了，无论这两个进程执行相应的系统调用的顺序如何，两个管道中的数据顺序不会发生变化，因此两个进程最终的读写结果不会改变。

```

#include <string.h>
char string[ ] = "hello,world";
main( )
{
    int count, i;
    int to_par[2], to_chil[2];          /* 到父、子进程的管道 */
    char buf[256];
    pipe(to_par);
    pipe(to_chil);
    if (fork( ) == 0)
    {
        /* 子进程从此处开始运行 */
        close(0);                      /* 关闭老的标准输入 */
        dup(to_chil[0]);                /* 把管道to_chil的读指针复制到标准输入 */
        close(1);                      /* 关闭老的标准输出 */
        dup(to_par[1]);                /* 把管道to_par的写指针复制到标准输出 */
        close(to_par[1]);              /* 关闭不再需要的管道描述符 */
        close(to_chil[0]);
        close(to_par[0]);
        close(to_chil[1]);
        for (; ;)
        {
            if((count=read(0, buf, sizeof(buf)) == 0)
                exit();
            write(1, buf, count);
        }
    }
}

```

```
/* 父进程从此处开始运行 */  
close(1);          /* 重新设置父进程的标准输入和输出 */  
dup(to_chil[1]);  
close(0);  
dup(to_par[0]);  
close(to_chil[1]);  
close(to_par[0]);  
close(to_chil[0]);  
close(to_par[1]);  
for (i=0; i<15; i++)  
{  
    write(1, string, strlen(string)); /* 每次向to_chil管道写11个字符 */  
    read(0, buf, sizeof(buf));       /* 每次从to_par管道读256个字符 */  
}  
}
```

## 7.2 软中断信号

### 1、软中断信号的作用：

通知进程发生了异步事件需要处理。

### 2、软中断信号的发送：

进程之间相互发送；

进程之内给自己发送。

### 3、发送软中断信号的方法：

**kill** 系统调用

## 4、软中断信号的分类：

### (1)、与进程终止相关的软中断信号

例如进程退出时

### (2)、与进程例外事件相关的软中断信号

例如地址越界、写只读内存区

### (3)、在系统调用期间遇到不可恢复的条件相关的软中断信号

例如执行**exec**而系统资源已用完

### (4)、在系统调用时遇到的非预测错误条件产生的软中断信号

例如调用不存在的系统调用、向无读进程的管道写数据

### (5)、由在用户态下运行的进程发出的软中断信号

例如用**kill**向自己或其他进程发出的软中断信号

### (6)、与终端交互有关的软中断信号

终端上按的**break**键或**delete**键等

### (7)、跟踪进程执行的软中断信号

# UNIX系统中的软中断信号定义

信号号	信号含义
0	正常的程序终止
1	挂断（拆线）
2	中断（ <b>break</b> 、 <b>delete</b> 、 <b>^C</b> ）
3	退出（ <b>FS</b> 字符，类似于 <b>break</b> ）
4	合法结构
5	跟踪陷进（被跟踪程序试图执行 <b>exec</b> ）
6	<b>IOT</b> 结构
7	<b>EMT</b> 结构
8	除浮点外
9	不可捕俘的 <b>kill</b> 信号
10	总线错误
11	溢出（地址越界）
12	系统调用时自变量非法
13	输出给管道，但管道没有接受者
14	警告
15	<b>kill</b> 信号
16	用户定义的信号1
17	用户定义的信号2
18	子进程的僵死不应该被捕俘，否则引起管道问题
19	掉电

其中 2、3、15 号信号是常由人工发出的软中断信号

## 5、软中断信号的标识

给一个进程发软中断信号时，核心在接收进程的核心进程控制表proc中，按所要接收的信号类型设置软中断信号域中的某一位。

当该接收进程睡眠在一个可被中断的优先级上时，核心就唤醒该进程。



## 6、检查软中断信号的时间

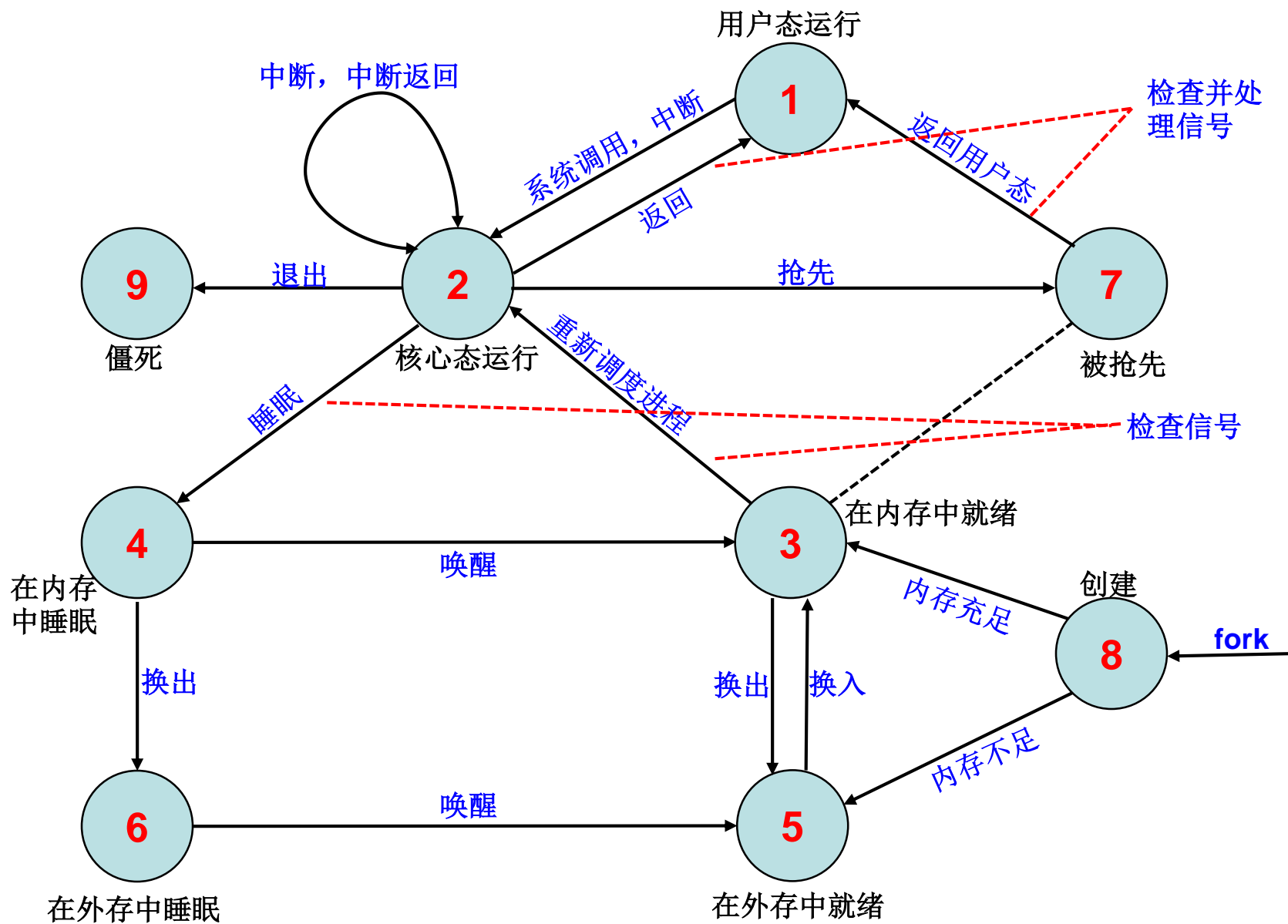
- (1) 当一个进程即将从核心态返回到用户态时；
- (2) 当一个进程即将进入或退出一个较低调度优先级的睡眠状态时。

## 7、处理软中断信号的时刻

仅当进程从核心态返回用户态时才处理软中断信号。

由于软中断信号主要由应用程序运行时发出，不是影响系统正常运行的特别紧急事件，故不在核心态下处理——当进程在核心态下运行时，软中断信号并不立即起作用。

并且，核心也保证了进程在核心态下运行的时间不会太长，能够保证进程能很快在从核心态返回用户态时，对可能的软中断信号进行处理，从而确保对应用程序的响应时间要求。



算法 **issg** /\* 检查是否收到软中断信号 \*/

输入：无

输出：如果进程收到不忽略的软中断信号，则为真；否则为假

```
{
    while (proc表项中收到的信号的域不为0)
    {
        找出发送给进程的一个软中断信号号——是第几号中断;
        if (该信号是“子进程死”)
        {
            if (忽略“子进程死”信号)
                释放僵死的子进程的proc表项;
            else if (捕俘“子进程死”信号)
                return (真);
        }
        else if (不忽略信号)
            return (真);
        关掉proc表项中该信号域的（要忽略的）信号位
    }
    return (假)
}
```

## 8、软中断信号的处理

进程在确认收到软中断信号后，有三种处理方式：

- (1) 进程退出 (**exit**) —— 缺省动作；
- (2) 进程忽略信号，就像没有收到该信号一样；
- (3) 进程收到信号后执行一个特殊的用户函数。通过系统调用**signal**来定义进程的动作。

**oldfunction = signal ( signum, function)**

**signum** —— 软中断信号号

**function** —— 函数地址。

**function=0** 进程在核心态下退出 (**exit**)

**function=1** 进程忽略以后出现的该信号

**oldfunction** —— 最近一次为信号**signum**所定义的函数的值

算法 **psig**       /\* 识别出软中断信号后处理信号 \*/

输入：无

输出：无

{

    取进程表项中设置的信号号；

    清理进程表项中的该信号号；       /\* 表明该信号已经被处理过了 \*/

    if (用户已调用**signal**来忽略该信号)     /\* 即**function=1** \*/

**return**;     /\* 完成 \*/

    if (用户指定了处理该信号的函数)

    {

        取u区中信号捕俘程序的虚地址；

        清除u区中存放信号捕俘程序地址的项；

        修改用户级上下文：创建用户栈来模拟对信号捕俘程序的调用；

        修改系统级上下文：将信号捕俘程序的地址写入用户保存的寄存器上下文中程序计数

器域；

**return**;

    }

    if (信号类型要求转储进程的内存映像)

        在当前目录中创建**core**文件，将用户级上下文中的内容写入**core**文件中；

    立即调用算法 **exit**;

{

## 捕获一个软中断信号的例子

```
#include <signal.h>
main()
{
    extern catcher();
    signal(SIGINT, catcher);
    .....
    kill(0, SIGINT);
}
```

进程向自己及其所有子进程发送软中断信号

```
catcher()
{
}
```

本例中程序捕获`interrupt`软中断信号（**SIGINT**），并向自己发一个`interrupt`软中断信号——即系统调用`kill`的结果。

**注意：**当进程处理软了中断信号且在其返回用户态**之前**，核心要清除u区中含有用户软中断信号处理函数的地址的域。如果进程要再次处理该信号，它必须再次调用系统调用 **signal**。

下面的例子演示了捕俘软中断信号时的竞争条件：

```

#include <signal.h>
sigcatcher()
{
    printf("PID %d caught one\n", getpid());    /* 打印进程标识号 */
    signal(SIGINT, sigcatcher);
}

main()
{
    int ppid;
    signal(SIGINT, sigcatcher);
    if (fork() == 0)
    {
        sleep(5);                                /* 为父子进程留足够的准备时间 */
        ppid = getppid();                        /* 获取父进程的标识号 */
        for (; ;)
        {
            if (kill(ppid, SIGINT) == -1)
                exit();                          /* 父进程不存在了，kill出错返回 */
        }
        nice(10);                                /* 降低优先权，增大出现竞争的机会 */
        for (; ;)
            ;
    }
}

```



## 7.3 进程组

**1、进程组标识号**用于标识一组相关的进程，这组进程对于某些事件将收到共同的信号。

**2、系统调用setpgrp**初始化一个进程的进程组号，将组号设置为与该进程的进程标识号相同的值。

**grp = setpgrp( )**

其中**grp**为新的进程组号，在系统调用**fork**期间，子进程**继承**其父进程的进程组号。

## 7.4 从进程发送软中断信号

进程使用系统调用kill来发送软中断信号：

**kill ( pid, signum)**

其中**signum**是要发送的软中断信号号，**pid**为软中断信号接收进程，**pid**的值与对应进程的关系如下：

- 1、**pid**为正值，信号发送给进程号为**pid**的进程；
- 2、**pid**为0，信号发送给与发送者同组的所有进程；
- 3、**pid**为-1，信号发送给真正用户标识号等于发送者的有效用户标识号的所有进程。
- 4、**pid**为负数但非-1，信号发送给**组号**为**pid**绝对值所有进程。

## 使用系统调用**setpgrp**的例子:

```
#include <signal.h>
```

```
main( )
```

```
{
```

```
    register int i;
```

```
    setpgrp( );
```

```
    for ( i=0; i< 10; i++ )
```

```
    {
```

```
        if ( fork( ) == 0 )
```

```
        {
```

```
            /* 子进程 */
```

```
            if ( i & 1 )
```

```
                setpgrp( );
```

```
                printf("pid = %d  pgrp = %d\n", getpid( ), getpgrp( ));
```

```
                pause( ); /* 挂起进程执行的系统调用 */
```

```
        }
```

```
    }
```

```
    kill (0, SIGINT);
```

```
}
```

## 命令行上的信号捕俘和处理命令 **trap**

### 1、捕俘信号，并执行指定命令：

```
trap “command” signal1 signal2 signal3 ...
```

例如：

```
trap “echo Receive a signal” 2 3 15
```

### 2、复位，恢复信号原有功能：

```
trap signal1 signal2 signal3 ...
```

例如：

```
trap 2 3 15
```

## trap命令应用实例：

- 1、建立具有root权限的turnoff用户，主目录：/home/turnoff，在.profile文件中设置一条sysoff命令
- 2、建立/bin/sysoff 命令，owner为root，sysoff程序如下：

```
trap ' ' 1 2 3 15
clear
echo "Would you want to shutdown the system? (y/n)"
read answer
if [ "$answer" = "y" ]
then
    sync
    shutdown
else
    clear
    kill -9 0
fi
```

普通用户登录无口令的turnoff账号，即可完成关机任务。

## 7.5 进程的终止

UNIX系统中的进程都是执行系统调用**exit**来终止运行。进程进入僵死状态后，释放占用的资源，拆除进程的上下文，但保留进程的**proc**表项。

**exit ( status )**

其中**status**是僵死进程发送给父进程的状态值。进程可以显式地调用**exit**，也可以在程序的结尾隐含地调用。

## 算法 **exit**

输入：给父进程的返回码

输出：无

{

忽略所有软中断信号；

if （是与控制终端关联的进程组组长）

{

向该进程组中的所有组员发送挂起信号；

将所有组员的进程组号设置为0；

}

关闭所有打开的文件；

释放当前工作目录；

释放改变的根目录（如果设置有“当前根”的话）；

释放进程占用的内存；

写记账记录；

设进程状态为僵死状态；

将所有子进程的父进程设置为1号进程（init）；

若有任何本进程的子进程僵死，则向init发送“子进程死”信号；

向父进程发送“子进程死”信号；

上下文切换；

}

不再接收任何中断信号了

告诉子进程准备结束运行

避免后续某新进程获得本进程释放的进程号，又成为新的进程组组长，从而与本进程组原来的组员混淆

将本进程及所有子进程的运行时间、内存、I/O累计到proc表项和全局记账文件中

使本进程从原进程树中断开，不再管理子进程了。

```
main( )
{
    int child;
    if (( child = fork( ) == 0)
    {
        printf("child PID %d\n", getpid( ));
        pause( );
    }
    printf("child PID %d\n", child);
    exit(child);
}
```

### **exit**的例子:

一个进程创建一个子进程。子进程打印自己的进程号后，执行系统调用**pause**挂起自己，直到收到一个软中断信号。

父进程打印子进程的进程号后退出，并返回子进程的**PID**作为状态码。

对子进程而言，尽管父进程已死，子进程仍可继续运行下去，直到收到软中断信号为止。

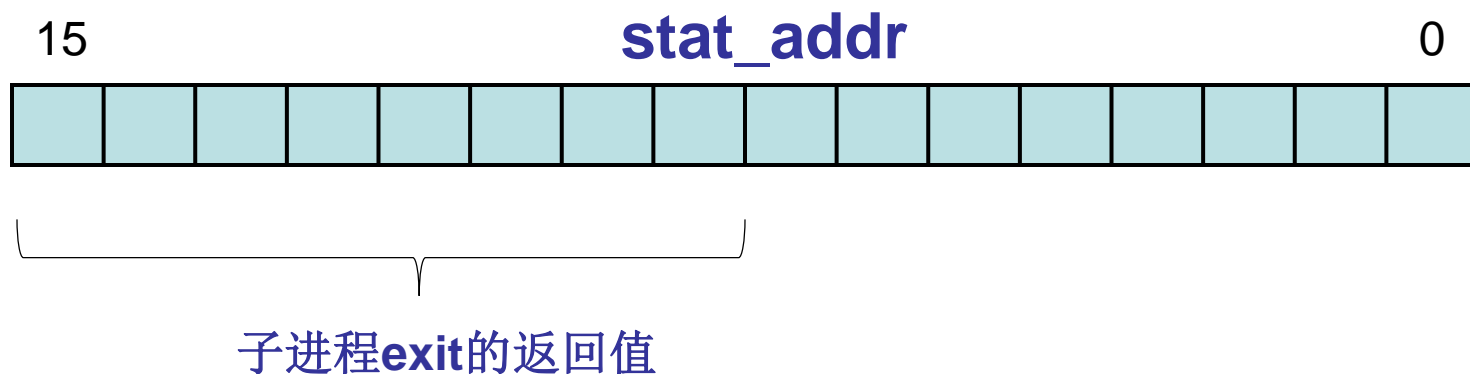


## 7.6 等待进程的终止

进程通过系统调用**wait**使自己的执行与子进程的终止同步：

**pid = wait(stat\_addr)**

其中**pid**是僵死子进程的进程号，**stat\_addr**是一个整数在用户空间的地址，它**含有**子进程的退出状态码，即**exit**的返回值。



## 算法 wait

输入：存放退出进程的状态的变量地址

输出：子进程标识号，子进程退出码

```
{
```

```
    if (本进程没有子进程)
```

```
        return (错误);
```

```
    for (; ; )
```

```
    {
```

```
        if (本进程有僵死子进程)
```

```
        {
```

```
            取任意一个僵死子进程;
```

```
            将子进程的CPU使用量累加到父进程;
```

```
            释放子进程的进程表项proc;
```

```
            return (子进程的进程标识号, 子进程的退出码);
```

```
        }
```

```
        if (本进程没有子进程)
```

```
            return (错误);
```

```
            睡眠在可中断的优先级上;
```

```
    }
```

```
}
```

并且不忽略“子进程死”软中断信号

僵死子进程不会自己释放proc表项，见exit()

虽进入for前检测到有子进程，但要忽略“子进程死”信号

并无特定的睡眠等待事件，而是被“子进程死”软中断信号唤醒

等待并忽略子进程死软中断信号的例子

```
#include <signal.h>
main(int argc, *argv[ ])
{
    int i, ret_pid, ret_code;
    if ( argc > 1)
        signal(SIGCLD, SIG_IGN); /* 忽略”子进程死”软中断信号*/
    for ( i=0; i<15; i++)
        if ( fork( ) == 0)
        {
            printf(“child proc %x\n”, getpid( ));
            exit (i); /* 把当前的 i值(子进程顺序号)返回给父进程 */
        }
    ret_pid = wait (&ret_code);
    printf(“wait ret_pid %x ret_code %x\n”, ret_pid, ret_code);
}
```

通过不同的运行参数的设置——确定进程对软中断信号的处理方式——影响进程间的同步关系和运行时序。

## 7.7 shell程序

```
/* 读命令行直到文件尾EOF */
```

```
while ( read(stdin, buffer, numcnars))
```

```
{
```

```
    /* 分析命令行 */
```

```
    if (命令行中含有 &)
```

```
        amper = 1;
```

```
    else
```

```
        amper = 0;
```

```
    /* 对于非shell的内部命令，即是操作系统的命令 */
```

```
    if (fork( ) == 0)    /* 由子进程来执行命令 */
```

```
{
```

```
    /* 是否为标准输入输出重定向? */
```

```
    if(标准输出重定向)
```

```
{
```

```
        fd = creat(newfile, fmask);
```

```
        close(stdout);
```

```
        dup(fd);
```

```
        close(fd);
```

```
}
```

```
    if(建立管道)
```

```
{
```

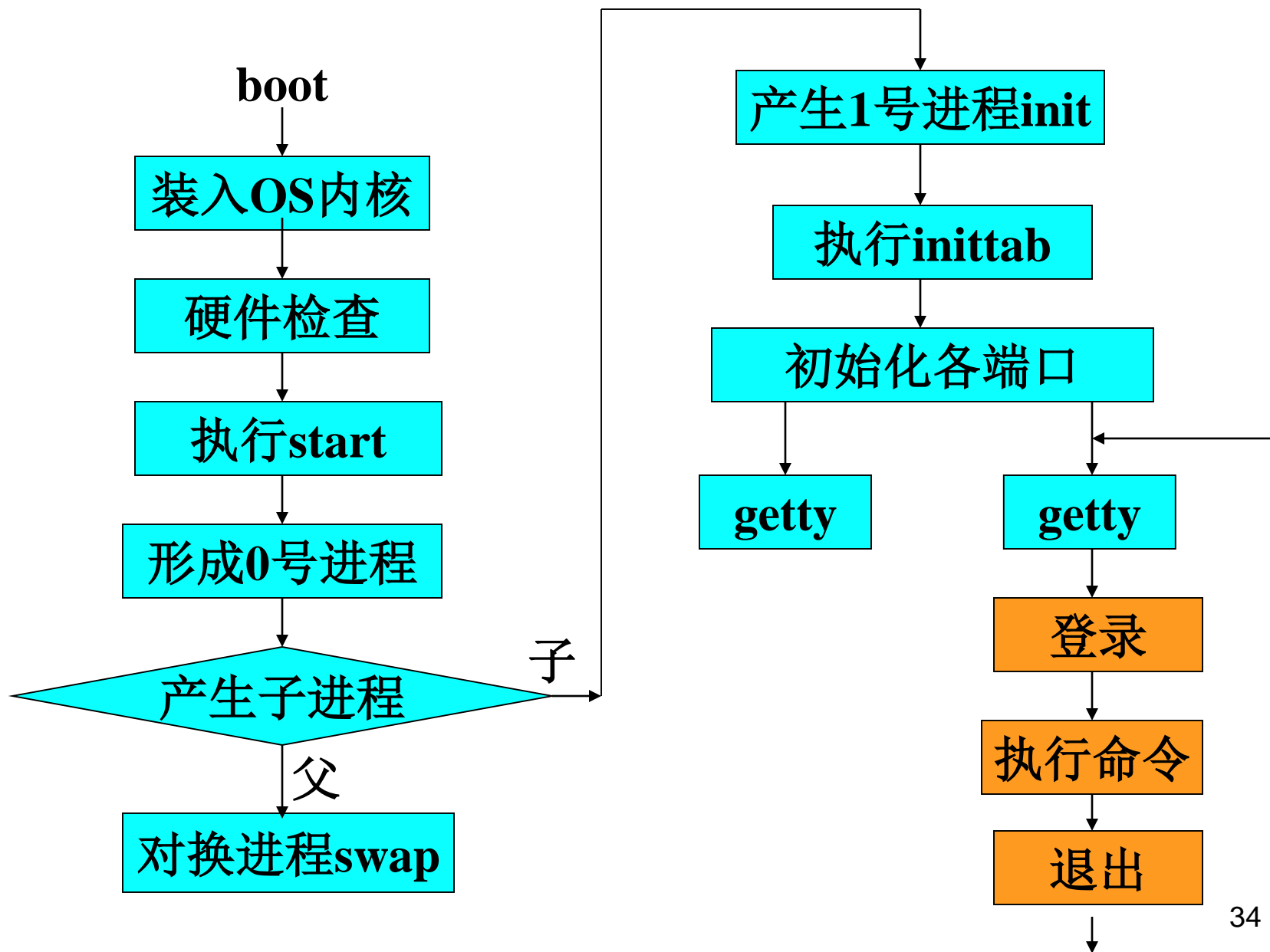
```
        pipe(fildes);
```

```

        if(fork( ) == 0)
        {
            /* 管道前面的命令，重定向标准输出到管道去 */
            close(stdout);
            dup(fildes[1]);          /* 复制管道写指针 */
            close(fildes[1]);
            close(fildes[0]);
            /* 子进程执行命令 */
            execlp(command1, command1, 0);
        }
        /* 管道后面的命令，重定向标准输入从管道来 */
        close(stdin);
        dup(fildes[0]);              /* 复制管道读指针 */
        close(fildes[0]);
        close(fildes[1]);
    }
    execve(command2, command2, 0); /* 管道后的命令 */
}
/* 父进程shell从此处继续运行.....
 * 如果需要则等待子进程退出
 */
if( amper == 0)
    retid = wait(&status);
} /* 回到while循环，重新读入下一个命令行 */

```

## 7.8 系统自举和init进程



```
算法  start    /* 系统初始化进程 */
输入： 无
输出： 无
{
    初始化全部核心数据结构，包括缓冲区、活动inode表、file表等；
    安装根文件系统；
    设置进程0的环境，初始化proc表和u区等；
    产生（fork）进程1；
    {
        /* 对于进程1 */
        为进程1分配地址空间；
        将用来执行init的代码（exec）从核心空间拷贝到用户空间，形成用户级上下文；
        改变状态：从核心态返回用户态；
        进程1调用exec执行/etc/init(模拟正常的系统调用);
    }
    /* 对于进程0 */
    产生(fork)核心进程，执行系统服务；
    如果需要，调用对换程序(swapper)进行内外存交换；
}
```

init根据/etc/inittab文件来进行系统初始化，并产生其他进程。下面是inittab的样本：

```
202.115.7.186 - PuTTY
shdaemon:2:off:/usr/sbin/shdaemon >/dev/console 2>&1 # High availability daemon
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
l7:7:wait:/etc/rc.d/rc 7
l8:8:wait:/etc/rc.d/rc 8
l9:9:wait:/etc/rc.d/rc 9
naudio2::boot:/usr/sbin/naudio2 > /dev/null
naudio::boot:/usr/sbin/naudio > /dev/null
ntbl_reset:2:once:/usr/bin/ntbl_reset_datafiles
rcml:2:once:/usr/sni/aix53/rc.ml > /dev/console 2>&1
logsymp:2:once:/usr/lib/ras/logsymptom # for system dumps
perfstat:2:once:/usr/lib/perf/libperfstat_updt_dictionary >/dev/console 2>&1
diagd:2:once:/usr/lpp/diagnostics/bin/diagd >/dev/console 2>&1
xmddaily:2:once:/usr/bin/xmddlm -L 2>&1 >/dev/null # Start xmddlm daily recording
ctrmc:2:once:/usr/bin/startsrc -s ctrmc > /dev/console 2>&1
dt:2:wait:/etc/rc.dt
cons:0123456789:respawn:/usr/sbin/getty /dev/console
ha_star:h2:once:/etc/rc.ha_star >/dev/console 2>&1
tty0:2:off:/usr/sbin/getty /dev/tty0
conserver:2:once:/opt/conserver/bin/conserver -d -i -m 64
inittab (95%)
```



**算法 init**      **/\* 系统1号进程 \*/**

输入：无

输出：无

```
{
    fd = open("/etc/inittab", O_RDONLY);
    while (从文件中读入一行到缓冲区)
    {
        /* 读inittab的每一行 */
        if(调用状态 != 缓冲区中标定的状态)
            continue;
        /* 状态匹配 */
        if(fork( ) == 0)
        {
            调用exec执行缓冲区中规定的程序;
            exit( );
        }
        /* init进程不等待，继续while循环 */
    }
    while((id = wait((int *)0)) != -1);
    {
        检查如果是本进程派生的子进程死，则考虑是否需要重新派生该子进程;
        否则，继续while循环;
    }
}
```

# 第八章 进程调度和时间

**UNIX**是分时分多进程操作系统，进程间的运行调度是整个系统运行控制的最根本内容。

## 进程调度的基本方式：

把每一次硬件时钟中断称为一个时钟“滴答”，由若干个时钟滴答构成一个时间片。

核心给每一个**用户**进程分配一个时间片，当该进程的时间片用完后，核心抢先该进程并调度另外一个进程运行。一段时间以后，核心又会重新调度该进程继续运行下去。以此方式，核心让各个进程轮流运行。

**核心**进程的运行：或者运行在不可被抢先的状态下；或者睡眠在某个中断级别上。

## 8.1 多级反馈循环调度算法

### 1、算法思想

核心给进程分配一个**CPU**时间片，抢先一个超过其时间片的进程，并把它反馈到若干优先级队列中的某一个队列上。

当进程的上下文切换结束时，核心执行**schedule\_process**算法来调度一个进程，即从处于“在内存中就绪（状态3）”和“被抢先（状态7）”状态的进程中，选取优先权最高的就绪进程。

如果若干个进程都具有相同的最高优先权，则核心选择在“就绪”状态时间最长的进程。

如果没有可运行的合格进程，核心则休闲等待，直到下次中断，下次中断最迟发生在下一个时钟滴答时。

## 算法 `schedule_process`

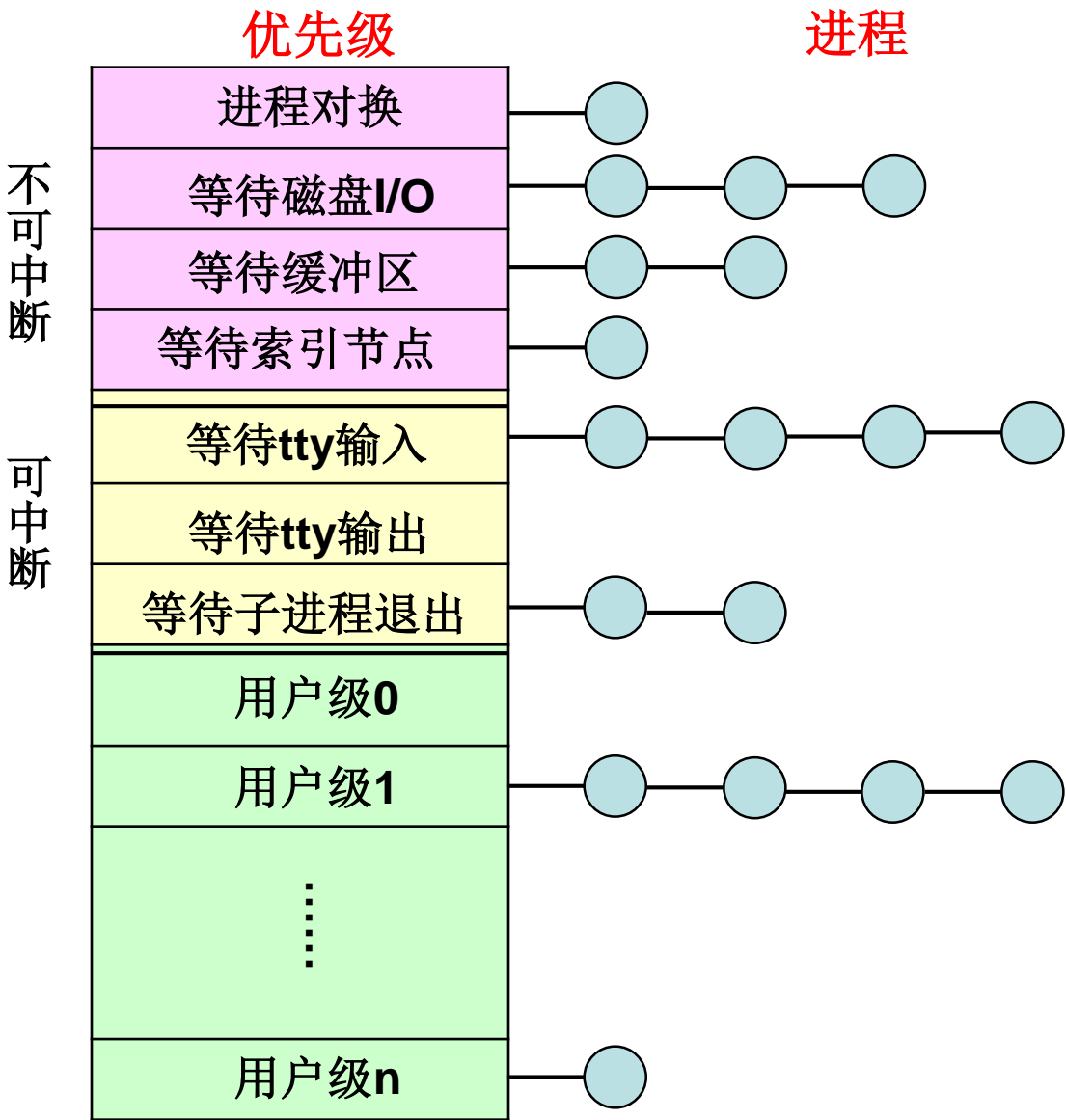
输入：无

输出：无

```
{  
    while (没有能被选取运行的进程)  
    {  
        for (就绪队列中的每个进程)  
            取已装入内存的、优先级最高的进程;  
        if (没有合格运行的进程)  
            机器休闲;    /* 下次时钟中断使机器脱离休闲状态, 重新开始循环 */  
    }  
    将选取的进程从就绪队列中移出;  
    切换到被选取进程的上下文, 恢复其执行;  
}
```

# 2、调度参数

核心态优先级



优先级阈值

用户态优先级

## 核心在三种情况下计算进程的优先权：

①、核心给一个即将进入睡眠态的进程赋予一个特定的优先权。

——越是等待系统**紧俏**资源的进程，获得的优先权越高。

②、核心调整从核心态返回到用户态的进程的优先权。

——进程在核心态下运行时拥有的较高优先权，必须在返回用户态时**降低**为用户级优先权。此外，该进程刚占用了宝贵的**CPU**时间，为公平起见也需降低本进程的优先权。

③、时钟中断处理程序每隔一个“时钟滴答”**调整所有**用户态进程的优先权。

——核心运行调度算法，防止某个进程垄断**CPU**的使用。

在一个进程的时间片中，时钟可能要使它中断若干次——遇到多次“时钟滴答”，每次中断时，时钟中断处理程序都要重新计算所有进程（包括运行进程和等待进程）的**CPU**使用量，并由此调整各就绪进程的优先权值。

**CPU使用量**=**decay(CPU) = CPU/2**

其中的**CPU**是进程占用处理器的时间（时钟滴答数）。

**进程优先数(priority)=(CPU使用量/2) + (初始用户优先数)**

其中的“初始用户优先数”就是**优先权初始值**。

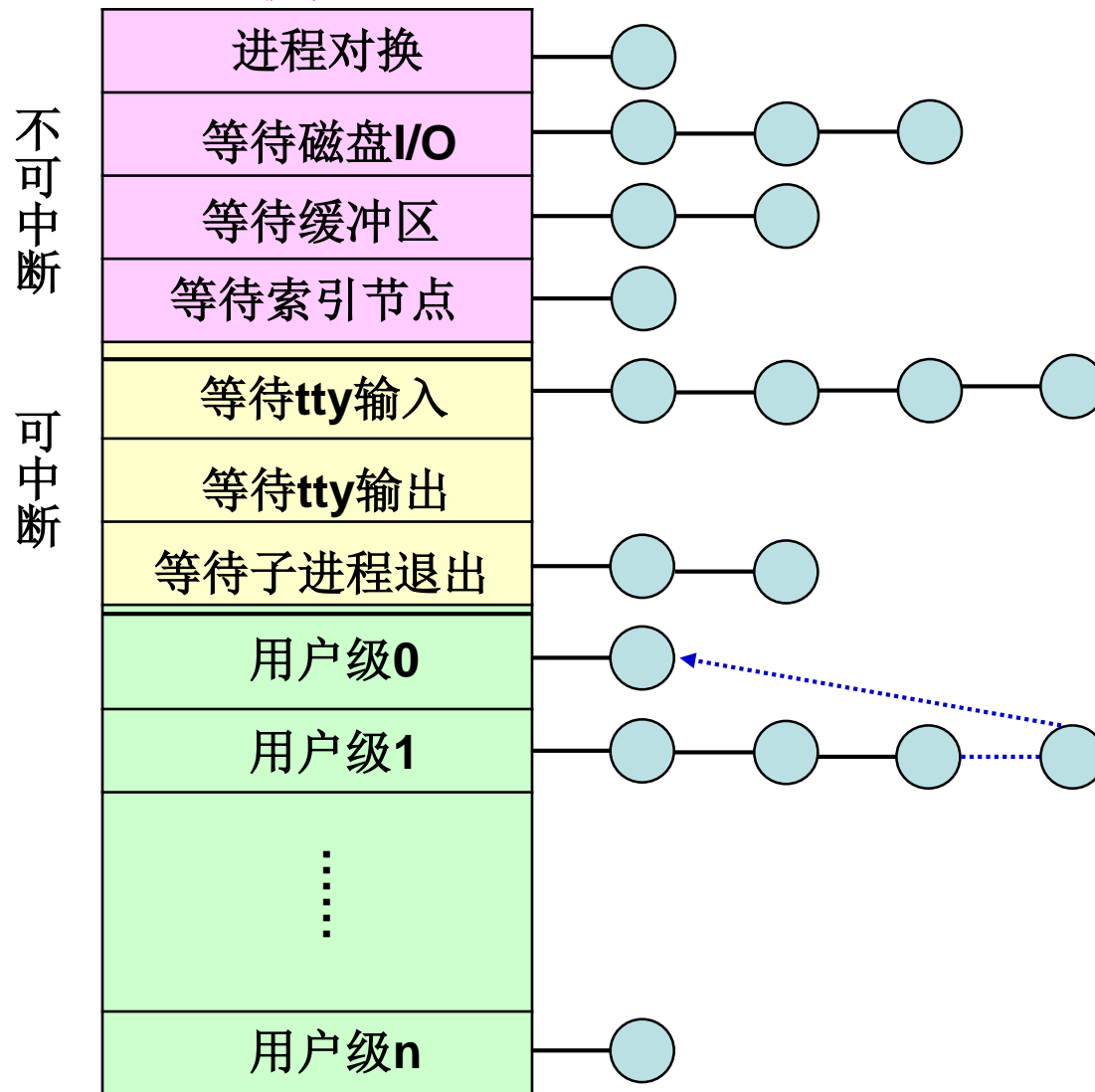
优先数越大，则优先级越低；优先数越小，则优先级越高——运行进程的运行时间越长，优先数越高，优先级降低；就绪进程等待的时间越长，优先数越低，优先级升高。

**用户级优先权进程不能跨越阈值获得核心级优先权。**

核心态优先权

优先级

进程



进程在优先级队列中移动



## 说明：

1、如果进程正在执行一段临界区代码时收到时钟中断，它先“**记住**”此事，并继续运行，直到当前处理机执行级别降低之后的下一次时钟中断时，再重新计算进程优先权。

2、提高系统**实时性**（响应速度）的方法：

缩短时间片

提高衰减函数的衰减速度

——使进程轮转的频率更快，但系统开销更高。

### 3、进程调度实例：

假设系统中有**A、B、C**三个同时建立的进程，具有相同的初始优先数**60**。系统中没有其它进程，这三个进程也没有做任何系统调用。时间片大小为一秒钟，每一秒钟产生**60**个时钟“滴答”，每个时间片到时，调用衰减函数重新计算每个进程的**CPU**使用量，必要的话就做上下文切换：

$$\text{CPU使用量} = \text{decay}(\text{时钟滴答数}) = \text{时钟滴答数}/2$$

则进程的优先数为：

$$\text{priority} = (\text{CPU使用量}/2) + 60$$

**60**为初始优先数。假设**A**进程先运行，下图为调度的顺序：

时间

进程A

进程B

进程C

优先数

CPU计数

优先数

CPU计数

优先数

CPU计数

0

60

0

1

2

...

60

60

0

60

0

时间片到，重新  
计算优先数

1

75

30

60

0

1

2

...

60

60

0

$75 = 30/2 + 60$

$30 = 60/2$

$60 = 0/2 + 60$

$0 = 0/2$

2

67

15

75

60

30

60

0

1

2

...

60

3

63

7

8

9

...

67

67

15

75

60

30

4

76

33

63

7

8

9

...

67

67

15

5

68

76

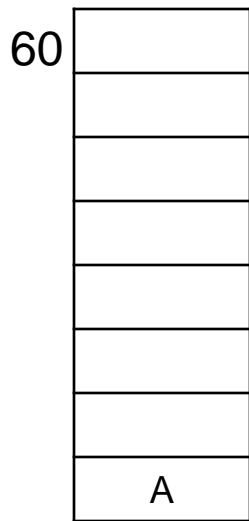
67

33

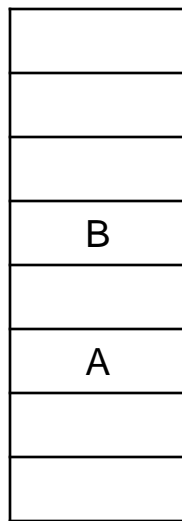
63

7

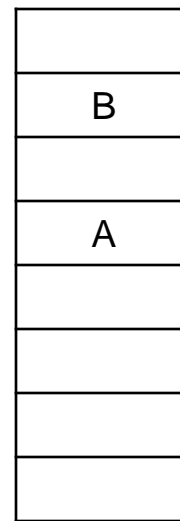
高  
↑  
优先级  
↓  
低



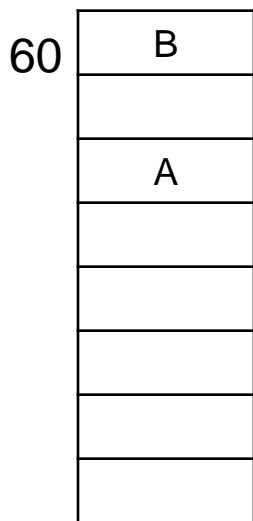
(a)



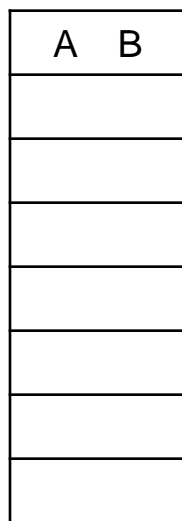
(b)



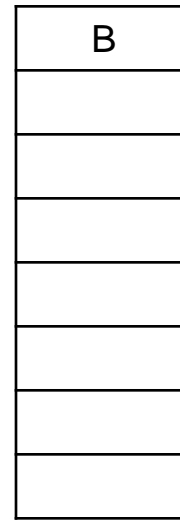
(c)



(d)



(e)



(f)

(A先运行)

## 4、进程上下文切换的时机

①、当进程进入睡眠或退出（exit）时，它**必须**做上下文切换；

②、当进程从核心态返回到用户态时，它**有机会**做上下文切换。即是，当另一个具有较高优先权的进程就绪的话，核心就会抢先一个从核心态返回用户态的进程。出现这种抢先进程的原因有**两点**：

**a、**核心唤醒了一个比当前运行的进程具有更高优先权的进程。既然可以得到一个具有较高优先权的核心进程，那么当前运行的进程就不应该在用户态下继续运行。

**b、**当时钟中断处理程序修改了所有就绪进程的优先权时，发现当前运行进程已经用尽了时间片，而且许多进程的优先权已被修改到较高值，于是，核心重新调度一个进程。

## 5、调整进程优先数

系统调用 **nice** 用于调整一个进程的调度优先数：

**nice(value);**

**nice**的值**value**被加到计算进程优先数的公式中：

**priority=CPU使用量/2 + 基级用户优先权值 + value**

对于普通用户来讲，**value**必须是正整数，因此使用**nice**后必然使优先数**priority**的值加大，导致进程的优先级降低 —— 对其它进程来讲是有益无害的，因此非常“**nice**”。

在一些**UNIX**版本中**root**用户可以把**value**的值设置为**负值**，也就是超级用户可以提高一个进程的执行速度！

## 6、公平共享调度策略

**问题：** 前述的调度算法不对用户做区分，（用户态）进程间按“绝对平均”的原则分配时间片，无法满足现实应用中不同用户或不同进程需要不同响应级别的要求。

### 基本解决方案：

将用户分成若干个组 —— 公平共享组，系统将**CPU**时间平均分给各组，而不管各个组内有多少成员，组内成员再平均分配本组的**CPU**时间。

第一组  
25%CPU时间

A

A进程  
25%CPU时间

第二组  
25%CPU时间

C D

每个进程  
12.5%CPU时间

第三组  
25%CPU时间

E F G

每个进程  
8%CPU时间

第四组  
25%CPU时间

H I J K

每个进程  
6%CPU时间

## 6、公平共享调度策略

### 具体解决算法：

在前述的多级反馈循环调度算法的优先数计算公式中加入“**公平共享组优先数**”项，共享组中的任何进程使用**CPU**，则组中的所有进程的“**组优先数**”都要同时增加**CPU**计数值。

$$\text{priority} = \text{CPU使用量}/2 + \text{基级用户优先权值} + \text{组优先数}$$

在下图的实例中，**A**进程单独一组，**B**和**C**进程另成一组，两个组平等分享**CPU**的时间 —— 各组**50%**的**CPU**时间。



时间	进程A				进程B				进程C			
	优先数	CPU	组		优先数	CPU	组		优先数	CPU	组	
0	60	0 1 2 ⋮ 60	0 1 2 ⋮ 60		60	0	0		60	0	0	
1	90	30	30		60	0 1 2 ⋮ 60	0 1 2 ⋮ 60		60	0	0 1 2 ⋮ 60	
2	74	15 16 17 ⋮ 75	15 16 17 ⋮ 75		90	30	30		75	0	30	
3	96	37	37		74	15	15 16 17 ⋮ 75		67	0 1 2 ⋮ 60	15 16 17 ⋮ 75	
4	78	18 19 20 ⋮ 78	18 19 20 ⋮ 78		81	7	37		93	30	37	
5	98	39	39		70	3	18		76	15	18	

## 8.2 有关时间的系统调用

### 1、 **stime(pvalue);**

设置系统当前时间。超级用户调用**stime**来设定系统的当前时间，**pvalue**是以秒为单位的整数值，从**1970年1月1日零时零分**开始计时。系统命令**settime**主要就是调用**stime**来实现的。

### 2、 **time(tloc);**

获取系统当前时间。**tloc**用于存放返回给用户的时间值的单元，这个时间值同样是自**1970年1月1日零时零分**以来的秒数，应用程序需要将其转换为具体的年月日时分秒。命令**gettime**就主要是调用**time**。

## 8.2 有关时间的系统调用

### 3、 **times(tbuffer);** **struct tms \*tbuffer;**

读取本进程及其子进程的运行时间。其中**tbuffer**存放查询到的时间，由如下数据结构**tms**定义：

```
struct tms
{
    /* time_t是用于时间的数据结构 */
    time_t tms_utime;      /* 进程的用户态时间 */
    time_t tms_stime;      /* 进程的核心态时间 */
    time_t tms_cutime;     /* 子进程的用户态时间 */
    time_t tms_cstime;     /* 子进程的核心态时间 */
}
```

系统调用**times**的**返回值**是“从过去的任意一个时刻”开始所消逝的时间，通常是从系统初始的时间开始。

```

#include <sys/types.h>
#include <sys/times.h>
extern long times();
main()    /* 使用系统调用times的程序实例 */
{
    int t;
    struct tms pb1, pb2;    /* tms是有4个时间元素的数据结构 */
    long pt1, pt2;

    pt1 = times(&pb1);    /* 获取启动时间 */

    for ( i=0; i<10; i++)
        if (fork() == 0)
            child(i);

    for ( i=0; i<10; i++)
        wait((int *)0);    /* 等待10个子进程全部结束 */

    pt2 = times(&pb2);    /* 获取结束时间 */

    printf("parent real %u user %u sys %u cuser %u csys %u\n",
        pt2 - pt1, pb2.tms_utime - pb1.tms_utime, pb2.tms_stime - pb1.tms_stime,
        pb2.tms_cutime - pb1.tms_cutime, pb2.tms_cstime - pb1.tms_cstime);
}

```

```
child(int n)
{
    int i;
    struct tms cb1, cb2;
    long t1, t2;

    t1 = times(&cb1);

    for ( i=0; i<10000; i++)
        ;

    t2 = times(&cb2);

    printf("child %d: real %u user %u sys %u\n", n, t2 - t1,
        cb2.tms_utime - cb1.tms_utime, cb2.tms_stime - cb1.tms_stime);

    exit();
}
```

### 结论:

父进程的用户时间不等于子进程的用户时间之和  
父进程的系统时间不等于子进程的系统时间之和

## 8.2 有关时间的系统调用

### 4、 **alarm(seconds);**

系统调用**alarm**用来设置闹钟软中断信号，其中**seconds**为秒数，用于设定从现在开始指定的时间后发出闹钟中断信号。

下图实例为一个无限循环，每分钟检查一次指定文件的存取时间，如果文件被访问过，则打印一个信息。

```

main(int argc, char *argv[ ])
{
    extern unsigned alarm( );
    extern wakeup( );
    struct stat statbuf;
    time_t axtime;
    axtime = (time_t)0;
    for (;;)
    {
        /* 检查文件的存取时间 */
        if (stat(argv[1], &statbuf) == -1)
        {
            printf("file %s not there\n", argv[1]);
            exit( );
        }
        if (axtime != statbuf.st_atime)
        {
            printf("file %s accessed\n", argv[1]);
            axtime = statbuf.st_atime;
        }
        signal(SIGALRM, wakeup); /* 设置收到闹钟信号后如何处理 */
        alarm(60);
        pause( );      /* 睡眠等待软中断信号 */
    }
}

wakeup( )
{
}

```

## 8.3 时钟

### 时钟中断处理程序的功能：

- ① 重新启动时钟；
- ② 按内部定时器有计划地调用内部的核心函数；
- ③ 对核心进程和用户进程提供运行直方图分析的能力；
- ④ 收集系统和进程记账及统计信息；
- ⑤ 计时；
- ⑥ 在有请求时，向进程发送闹钟软中断信号；
- ⑦ 定时唤醒对换进程；
- ⑧ 控制进程调度；



```

算法 clock      /* 时钟中断处理程序的算法 */
输入：无
输出：无
{
    重新启动时钟； /* 为了获得下一次时钟中断信号 */
    if ( callout表非空) /* 定时调用函数表 */
    {
        修改callout时间；
        如果时间已消逝，安排调度callout函数；
    }
    if (核心直方图分析已开)
        记下中断时刻的程序计数器；
    if (用户直方图已开)
        记下中断时刻的程序计数器；
    收集系统统计信息；
    收集本进程统计信息；
    if (自上次执行此语句以来已经过了1秒钟或更多时间，且中断不是发生在临界区代码区)
    {
        for (系统中的所有进程)
        {
            如果进程活动的话，调整闹钟时间；
            修改CPU的使用量；
            if (进程在用户态执行)
                修改进程优先数；
        }
        必要的话，唤醒对换进程
    }
}

```

过了一个时间片

## 8.3 时钟

### 1、重新启动时钟

系统时钟通常就是一个硬件计数器，对石英晶体的振动进行计数，当计数器达到指定值（如最大值，或进位位置一）时，产生一个硬件中断——时钟中断。操作系统常依次来同步和协调软件系统中各个程序（进程）的运行。

由于各个不同品牌的机器中，硬件计数器的差异巨大，以及对时钟系统准确性的要求，时钟中断处理程序通常使用汇编语言编写的。

每次时钟中断来临时，操作系统马上又立即启动时钟，以便获得下一次的时钟中断。

由于强调计时的及时和准确性，在**UNIX**中把时钟中断处理的优先级设置得最高（除硬件故障外）。

## 8.3 时钟

### 2、系统的内部定时


在分时系统中的许多操作需要在实时的基础上调用核心函数来完成，如设备驱动和网络协议。

核心设置了一个callout表，其中含有当定时时间到时所要调用的函数名、函数参数、以时钟滴答为单位的定时时间。

用户不能直接控制callout表中的表项，这些表项是由核心在需要时用相关算法创建的。对callout表中的表项，核心不是按它们被放入表中的先后次序排序，而是按它们各自的“启动时间”进行排序。

在callout表中，各个表项的时间域记录的是前一表项启动后，到该表项被启动时的时间量。


例：在callout表中加入新表项f函数 —— 5个时钟滴答后调用f函数

当前时间

函数名	启动时间
a()	-2
b()	3
c()	10

加入f函数之前的callout表

3个时钟滴答后  
13个时钟滴答后

当前时间

函数名	启动时间
a()	-2
b()	3
f()	2
c()	8

加入f函数之后的callout表

3个时钟滴答后  
5个时钟滴答后  
13个时钟滴答后

创建一个新表项时，核心找出新表项的位置，并适当调整紧接新表项之后的那一项的时间域，而不需要改动其他表项的时间域！

## callout表项的调度流程:

- ① 时钟中断处理程序在每次时钟中断时，只把表中的第一项的时间域减1，后续的也就相应地自动减1了。
- ② 如果表中第一项的时间域小于或等于0，则应该调用该函数了。
- ③ 时钟中断处理程序并不直接调用该函数，而是产生一个较低级别的“软中断”——可编程中断来“记住”要调用该函数，开放级别较高的中断（如时钟中断等）。
- ④ 当所有较高级别的中断都处理完毕后，再运行相应的“软中断处理程序”。
- ⑤ 在callout表中应该调用某个函数的时刻到实际发生软中断之间，可能发生过包括时钟中断在内的多个中断，因此callout的第一个表项的时间域可能已被减为负值，软中断处理程序将清除已过时的callout表项，并调用相应的函数。
- ⑥ 由于callout表中前面几项可能为零或小于零，时钟中断处理程序必须找出第一个时间域为正值的表项，并使其减1。

## 8.3 时钟

### 3、直方图分析

核心直方图驱动程序在时钟中断时，对系统的活动（地址）进行采样，以便监视系统在核心态和用户态下的执行时间的相对比例，对系统性能进行评估。

直方图分析程序有一个用于采样的核心地址表，表中包含有核心函数的地址。

允许核心进行直方图分析时，时钟中断处理程序就调用直方图驱动程序对应的中断处理程序，对当前程序计数器的取值进行记录，并与核心地址表相比较，以确定当前正在运行哪个程序。

## 示例：核心算法的采样地址表

算法	地址	计数
bread	100	5
breada	150	0
bwrite	200	0
brelse	300	2
getblk	400	1
user	—	2

**5次在地址100~149之间**

**0次在地址150~199之间**

**0次在地址200~299之间**

**2次在地址300~399之间**

**1次在地址400以上**

**2次在用户地址空间**

# 第九章 输入/输出子系统

输入/输出子系统（简称I/O子系统）的功能就是使进程能够与外部设备进行通讯，**设备驱动程序**是输入/输出系统中的核心模块，并且与设备类型一一对应。

每一种类型的设备都有特定的设备驱动程序；而每一种设备驱动程序控制这种类型的所有设备。



## 9.1 设备驱动程序接口

UNIX系统中把设备分为两大类：

- **块设备** —— 以块为单位进行数据的输入输出，如硬盘、软盘、磁带、光盘等设备。
- **字符设备**（原始设备，**raw**设备） —— 以字节为单位进行数据的输入输出，如终端、打印机、绘图仪、调制解调器、网卡等。

每个设备都有一个文件名（i节点）相对应，用于标识该设备的属性：

```
br--r--r--    1 root    system    14,    0 Mar 27 2009    cd0
crw-rw-rw-    1 root    system    15,    0 Mar 27 2009    clone
crw--w--w-    1 root    system      4,    0 Mar 27 2009    console
crw-rw-rwT    1 root    system    49,    0 Mar 28 2009    dlc8023
crw-rw-rwT    1 root    system    48,    0 Mar 28 2009    dlcether
crw-rw-rwT    1 root    system    47,    0 Mar 28 2009    dlcfdi
crw-rw-rwT    1 root    system    46,    0 Mar 28 2009    dlcqlc
crw-rw-rwT    1 root    system    45,    0 Mar 28 2009    dlcsdlc
crw-rw-rwT    1 root    system    44,    0 Mar 28 2009    dlctoken
crw-rw-rw-    1 root    system    15, 34 Mar 27 2009    echo
crw--w--w-    1 root    system      6,    0 Aug 28 10:05    error
crw-----    1 root    system      6,    1 Mar 27 2009    errorctl
brw-rw-rw-    1 root    system    19,    0 Mar 27 2009    fd0
```

# 1、系统配置

**系统配置**就是告诉核心，当前系统中包含哪些设备，以及这些设备的“地址”——建立设备文件、联接设备驱动程序，例如：

```
mknod /dev/tty15 c 2 15
```

核心与驱动程序的接口是由**块设备开关表**和**字符设备开关表**来描述的。

每一种设备类型在开关表中都有若干表项，这些表项在系统调用时引导核心转向适当的驱动程序接口。

硬件与驱动程序的接口，是由与机器相关的控制寄存器或操作设备的**I/O**指令以及中断向量组成：当一个设备发出中断时，系统识别发出中断的设备，并调用适当的中断处理程序。

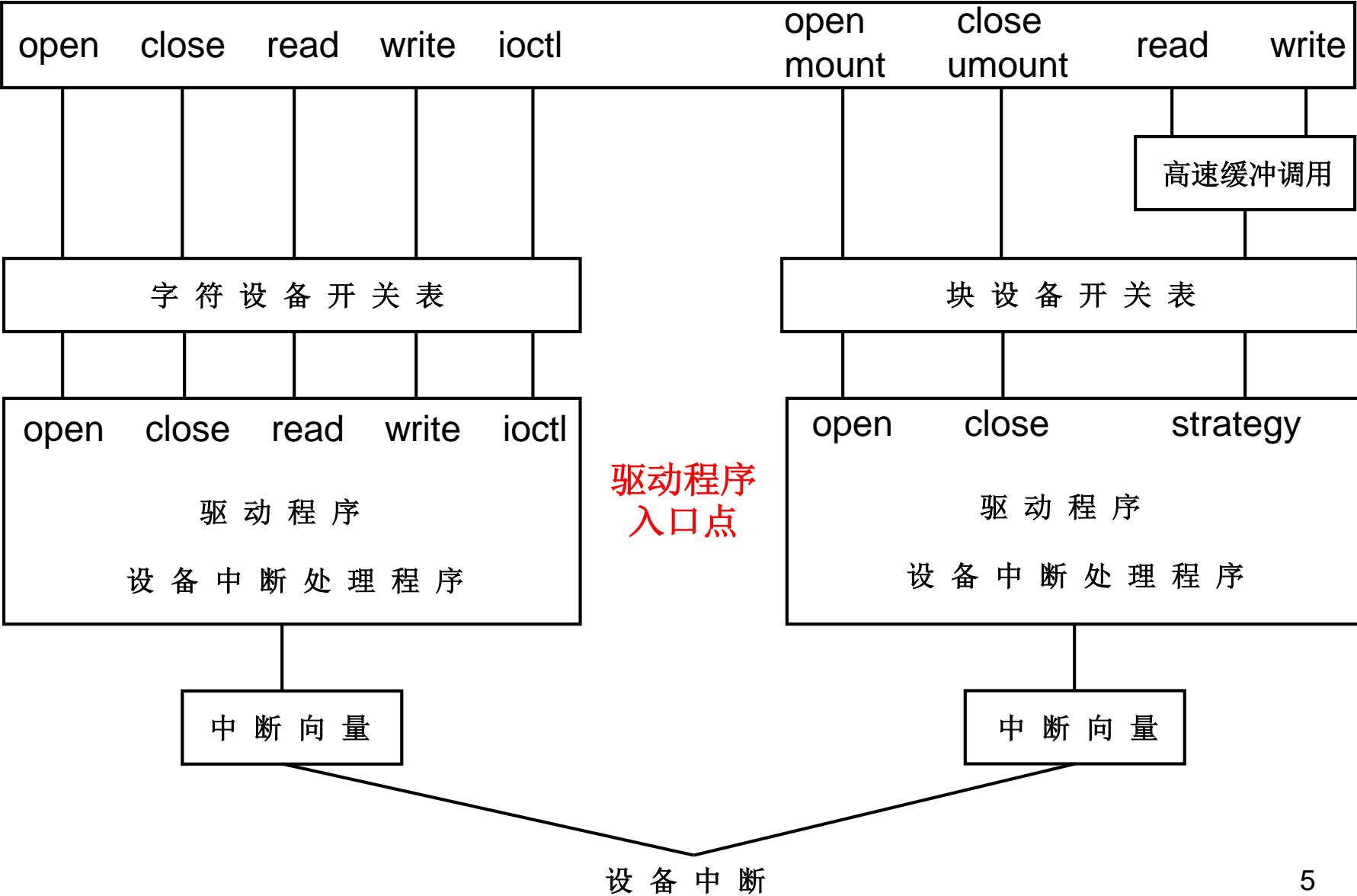
# 设备开关表示例

块 设 备 开 关 表			
表项	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

字 符 设 备 开 关 表					
表项	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzboopen	dzbclose	dzbread	dzbwrite	dzbiocli
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodev

# 驱动程序入口点

## 文件子系统



## 2、系统调用与驱动程序接口

对于使用文件描述符的系统调用的**基本操作流程**:

- 核心从用户文件描述符的指针找到系统打开文件表项和文件的索引节点。
- 检查文件的类型，根据需要存取块设备或字符设备开关表。
- 从索引节点中抽取主设备号和次设备号。
- 使用主设备号为索引值进入相应的开关表。
- 根据用户所发的系统调用来调用开关表中的对应函数。

## 2、系统调用与驱动程序接口（续）

对设备文件的系统调用与对正规文件的系统调用之间的一个重要区别是：当核心执行驱动程序时，对设备文件的索引节点是不上锁的。

因为驱动程序会频繁地睡眠，以等待硬连接或数据的到来，因此核心不能确定一个进程要睡眠多长时间。

如果对设备文件的索引节点上锁，则其它存取此节点的进程（如通过系统调用**stat**）会因为本进程在驱动程序中睡眠，而无限期地睡眠（等待）下去。

① 算法: **open** /\* 用于设备驱动程序的**open** \*/

输入: 路径名, 打开方式

输出: 文件描述符

{

    将路径名转换为索引节点, 增加索引节点的引用计数;

    与正规文件一样, 分配系统打开文件表项和用户文件描述符;

    从索引节点取主设备号和次设备号;

**if** (块设备)

    {

        使用主设备号作为查块设备开关表的索引值;

        调用该索引值对应的驱动程序打开过程——传递参数为次设备号和打开方式;

    }

**else** /\* 是字符设备 \*/

    {

        使用主设备号作为查字符设备开关表的索引值;

        调用该索引值对应的驱动程序打开过程——传递参数为次设备号和打开方式;

    }

**if** (**open**在驱动程序中失败)

        减少系统打开文件表项和索引节点的计数值;

}

## ②、算法：close 断开一个进程与设备的连接

**a**、搜索系统打开文件表（**file**表）以确认没有其他进程仍然打开着这个设备。

**既不能**仅仅看**file**表中的计数值来确认这是该设备的最后一次关闭操作，因为几个进程可能通过不同的**file**表项(读写指针)来存取该设备；

**也不能**依靠活动索引节点表(**inode**表)的计数值来确定这是否为最后一次关闭操作，因为可能有几个不同的文件代表同一设备，例如：

crw-rw-rw-	1	root	sys	9, 1	Aug 6 2014	/dev/tty01
crw-rw-rw-	1	root	unix	9, 1	Sep 5 2015	/dev/tty02

虽然上述两个设备的名字不同，但它们的主设备号和此设备号相同，因此是同一设备。有可能多个进程独立地打开这两个文件，这些进程存取不同的**inode**，但却是对同一设备进行操作。



## ②、算法：close 断开一个进程与设备的连接（续）

**b**、对于一个**字符设备**，核心调用该设备对应的**close**过程并返回用户态。

对于一个**块设备**，首先判断该设备上是否为一个已安装的文件系统。

如果是已安装的文件系统，则释放索引节点后返回。

如果不是已安装的文件系统，核心要做两件事：

- ① 搜索数据缓冲区高速缓冲，看是否有先前安装本文件系统时遗留下来的数据块以“延迟写”的形式还在内存中，还没有写回设备。如果有，则先写回这些数据块。
- ② 在关闭本设备后，核心再扫描一次数据缓冲区高速缓冲，使装有本设备数据块的缓冲区无效（释放到空闲链表的表头），以使装有有效数据的缓冲区能在内存中停留更长时间。

## ②、算法： **close** 断开一个进程与设备的连接（续）

**c**、核心释放设备文件的索引节点。设备关闭程序断开与设备的连接，并重新初始化驱动程序的数据结构和设备硬件，使核心以后又能再次打开该设备。

## ②、算法：close 断开一个进程与设备的连接

输入：文件描述符

输出：无

```
{
    执行正规的算法close;
    if ( 系统打开文件表项引用计数值非0 )
        goto finish;
    if ( 存在另一个打开文件，其主次设备号与将关闭文件的相同 )
        goto finish;
    if ( 字符设备 )
    {
        以主设备号为索引值查找字符设备开关表；
        调用驱动程序关闭子程序：参数为次设备号；
    }
    if ( 块设备 )
    {
        if ( 设备已作为子文件系统安装 )
            goto finish;
        将数据缓冲区高速缓冲中该设备的数据块写回设备；
        用主设备号为索引值查找块设备开关表；
        调用驱动程序关闭子程序：参数为次设备号；
        使数据缓冲区高速缓冲中该设备的数据块无效；
    }
finish:
    释放设备文件索引节点；
}
```

### ③ 算法 read 和 write

- 针对设备的读写操作的算法与对正规文件的算法相似;
- 字符设备的驱动程序通常不使用系统中的数据缓冲区, 而使用内部的数据缓冲机制;
- 由于外部设备的速度相对较慢, 驱动程序的写操作通常包含流量控制;
- 驱动程序与设备间的数据通讯依赖于具体的硬件 —— 通常有“直接存储器存取 (**DMA**)”模式和“可编程I/O”模式。

## ④、系统调用 **ioctl**

在较早版本的**UNIX**系统中提供有专门用于终端的系统调用**stty**(预置终端参数)和**gtty**(读取终端预置参数)。

**ioctl**融合了**stty**和**gtty**两者的功能，以提供一个更通用、更规范的设备控制入口。它允许一个进程去设置与一个设备相关联的硬件选项和与一个驱动程序相关联的软件选项。

**ioctl**可对多种设备进行操作，包括终端（如波特率、数据位宽度、奇偶校验）、磁带（如正绕或反绕的方式）、网络（如ip地址、虚电路数量）等。

由于**ioctl**规定的专门的动作对每个设备都是不同的，并由设备驱动程序来定义，因此，使用系统调用**ioctl**的程序必须知道他们正在与什么类型的文件打交道，因为这些文件都是设备专用的。

**ioctl ( fd, command, arg)**

**fd:** 系统调用返回的文件描述符；

**command:** 使驱动程序完成指定动作的请求命令；

**arg:** 对应该命令的一个参数。

## stty命令的运行时的屏幕截图

显示的是当前终端的相关参数，包括波特率、中断键、删除键的设置，字节位数、奇偶校验、字符回显的定义等。

```
$  
$  
$ stty -a  
speed 9600 baud; 24 rows; 80 columns;  
eucw 1:1:0:0, scrw 1:1:0:0:  
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = ^@  
eol2 = ^@; start = ^Q; stop = ^S; susp = ^Z; dsusp = ^Y; reprint = ^R  
discard = ^O; werase = ^W; lnext = ^V  
-parenb -parodd cs8 -cstopb hupcl cread -clocal -parext  
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl -iuc1c  
-ixon -ixany -ixoff imaxbel  
isig icanon -xcase echo echoe echok -echonl -noflsh  
-tostop echoctl -echoprt echoke -flusho -pending iexten  
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tab3  
$  
$  
$
```

### 3、设备中断处理程序

一个设备的中断将引起核心执行一个中断处理程序。而执行什么样的中断处理程序是根据发出中断的设备和中断向量表中的偏移量来决定的。

核心调用**设备专用**的中断处理程序，并将设备号和其它参数传递给它，以便识别引起中断的特定的设备单元。

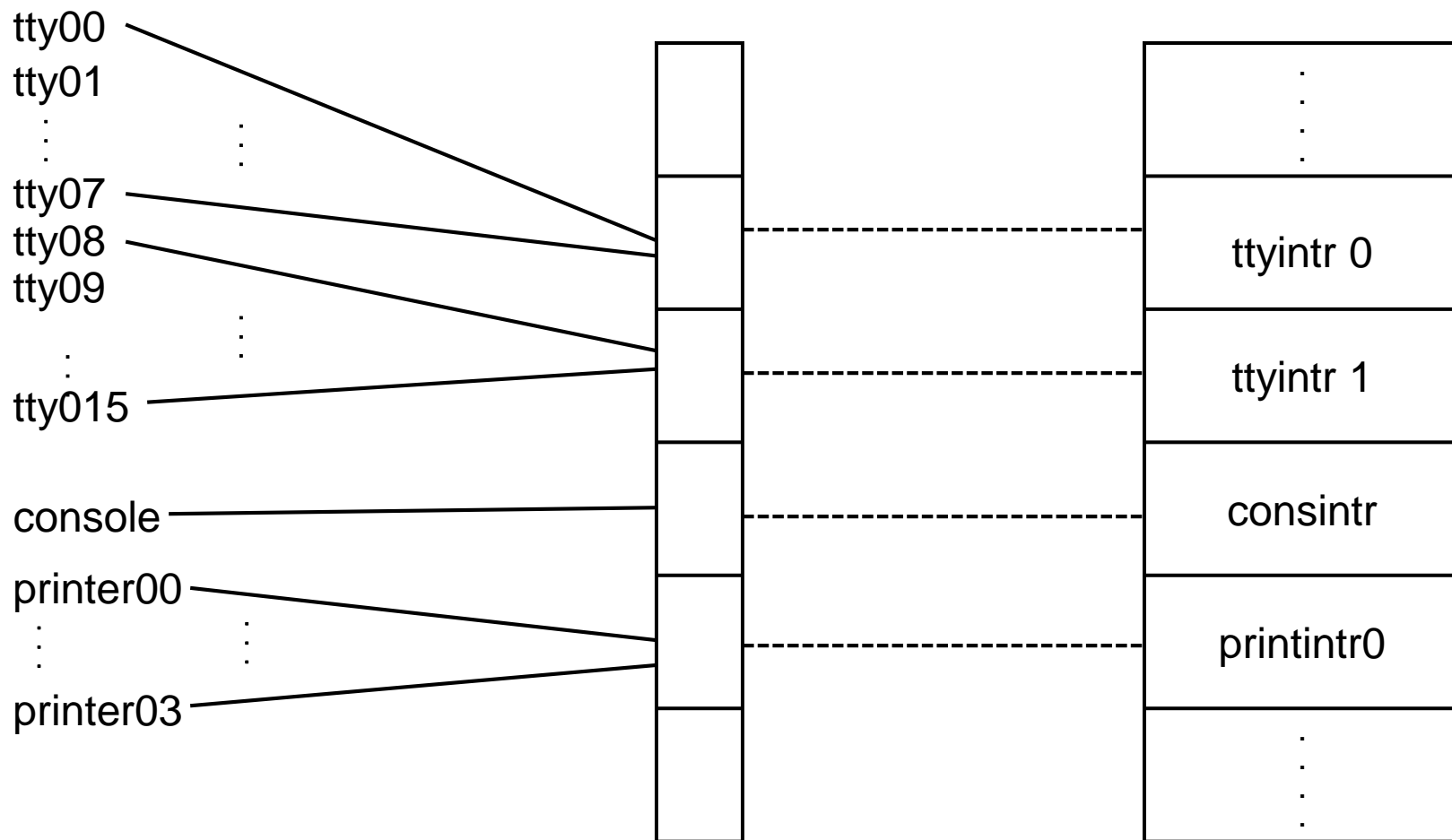
由于一个中断向量入口点可以与许多物理设备相联系，所以驱动程序必须能够判定是哪一个具体的设备引起的中断。

中断处理程序使用主设备号来识别硬件的类别，次设备号用来识别具体的设备。

外部设备

硬件底板

中断向量



设备中断



## 9.2 磁盘驱动程序

**UNIX**通常把一个物理的磁盘划分成若干个磁盘区域——称为**逻辑设备**，在每个磁盘区域上面都建立一个文件系统。

其主要目的是为了把不同类别的文件分别放在不同的文件系统中互不干扰，便于在进行文件系统的维护、备份和修复等工作时，分别对它们进行操作，减少相互影响。

在使用过程中，（除了根文件系统外）各个子文件系统可以根据需要选择“安装”、“不安装”、“只读”、“只写”和“可读可写”等工作方式，虽然这些子文件系统都在同一个物理设备上。

磁盘驱动程序把一个由逻辑设备号和块号构成的文件地址翻译成磁盘上特定的扇区号。

文件地址可由以下两种方法之一来获得：

- 1、使用数据缓冲区高速缓冲池中的一个缓冲区，该缓冲区的首部中就含有逻辑设备号和块号。
- 2、将逻辑设备号（次设备号）作为参数传递给读写程序，它们再把保留在u区中的字节偏移量转换为适当的块地址。

磁盘驱动程序用设备号来识别物理盘以及所使用的特定的磁盘分区，通过查找**磁盘分区表**就能找到特定磁盘分区在物理磁盘上的开始扇区，再把文件系统的块号加到起始扇区号上去，即可定位I/O传送所用的扇区号。

## 预分区的磁盘分区表示例

磁盘分区号	起始块号	块数
0	0	64000
1	64000	944000
2	168000	840000
3	336000	672000
4	504000	504000
5	672000	336000
6	840000	168000
7	0	1008000

如果设备文件“/dev/dsk0”、“/dev/dsk1”、“/dev/dsk2”和“/dev/dsk3”对应于磁盘分区0到分区3，则相应于次设备号0到3。

假设文件系统块与磁盘块大小相同，则文件系统“/dev/dsk3”中的第940块，就对应于物理磁盘上的第336940块。

```
#include "fcntl.h"
```

```
main()
```

```
{
```

```
    char buf1[4096], buf2[4096];
```

```
    int fd1, fd2, i;
```

```
    if ((( fd1 = open("/dev/dsk5", O_RDONLY)) == -1 ||  
        fd2 = open("/dev/rdisk5", O_RDONLY)) == -1 ))
```

```
    {
```

```
        printf("failure on open\n");
```

```
        exit();
```

```
    }
```

```
    if ((read(fd1, buf1, sizeof(buf1)) == -1 || (read(fd2, buf2, sizeof(buf2)) == -1))
```

```
    {
```

```
        printf("failure on read\n");
```

```
        exit();
```

```
    }
```

```
    for ( i=0; i<sizeof(buf1); i++)
```

```
        if (buf1[i] != buf2[i])
```

```
        {
```

```
            printf("different at offset %d\n", i);
```

```
            exit();
```

```
        }
```

```
    printf("reads match\n");
```

```
}
```

## 9.3 终端驱动程序

终端是用户与系统的交互界面。终端驱动程序用于控制从终端来和到终端去的数据。

终端驱动程序包含一个与**行规则**（**line discipline**）模块的内部接口，行规则程序的功能就是对输入和输出数据进行解释。

**终端有两种工作方式：**

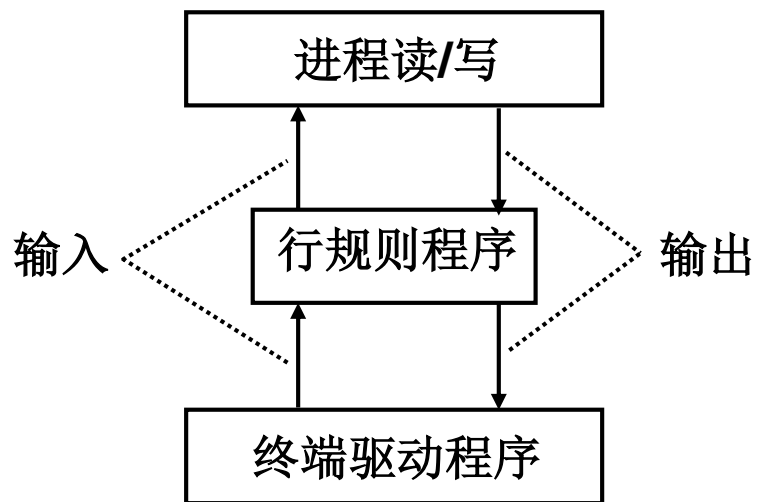
- ① **标准方式**（**canonical**）—— 行规则程序把键盘输入的原始数据变换成标准形式；或把进程输出的原始数据变换成用户期望的形式。
- ② **原始方式**（**raw**）—— 行规则程序只完成进程和终端之间的数据传输，而不做变换。

## 行规则程序的功能：

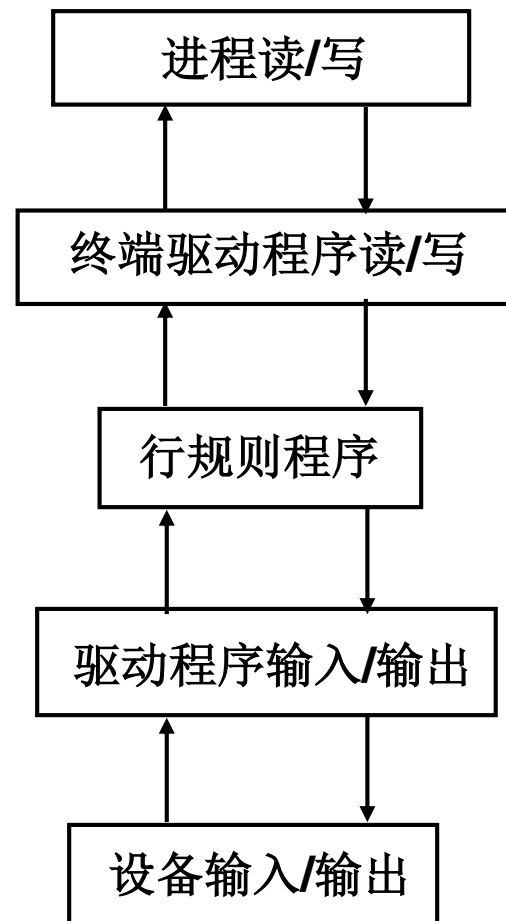
- ① 通过分析将输入字符串变成行；
- ② 处理擦除键；
- ③ 处理“抹行”字符，它使当前行上已敲入的所有字符无效；
- ④ 把接受的字符回显（写）到终端上；
- ⑤ 扩展输出，如把制表符变成一系列空格；
- ⑥ 为终端挂起（**hangup**）、断线或响应用户敲入的**delete**键，向进程产生软中断信号；
- ⑦ 允许不对特殊字符如擦除、抹行和回车进行解释的原始方式。

支持原始方式意味着可以使用一个异步的终端，这样进程可以在字符被敲入时就读它们，而不是等待用户敲入一个回车或“**enter**”键时才读——如**cbreak**功能。

## 数据流



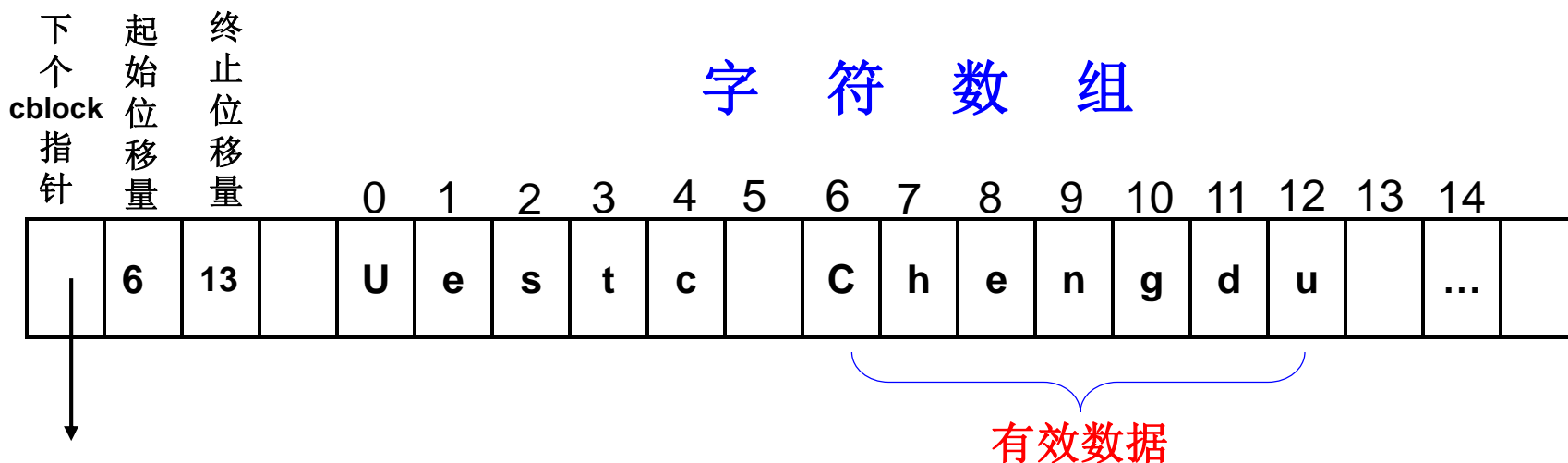
## 控制流



终端输入输出时的数据流和控制流

# 1、字符表clist

行规则程序在字符表上操作数据。字符表**clist**是字符缓冲区**cblock**的变长链表，并携带有表中的字符计数。



**cblock**的结构

起始位移量 —— 指向第一个有效数据的位置

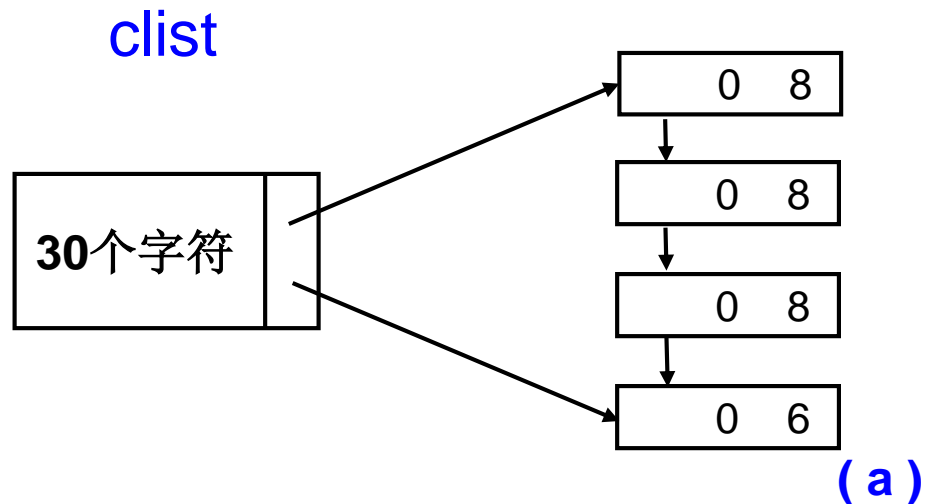
终止位移量 —— 指向第一个无效数据的位置



核心维护一个空闲**cblock**的链表，并规定在**clist**和**cblock**上有六种操作：

- ① 从空闲链表中分配一个**cblock**给驱动程序；
- ② 把一个**cblock**归还给空闲链表；
- ③ 从一个**clist**中取出一个字符；
- ④ 把一个字符放入**clist**表的末尾；
- ⑤ 从一个**clist**表的开头移去一组字符，每次一个**cblock**；
- ⑥ 把一个**cblock**的字符放到一个**clist**表的末尾。

# 图例1：从clist中移去字符 —— **getchar**（读数据）



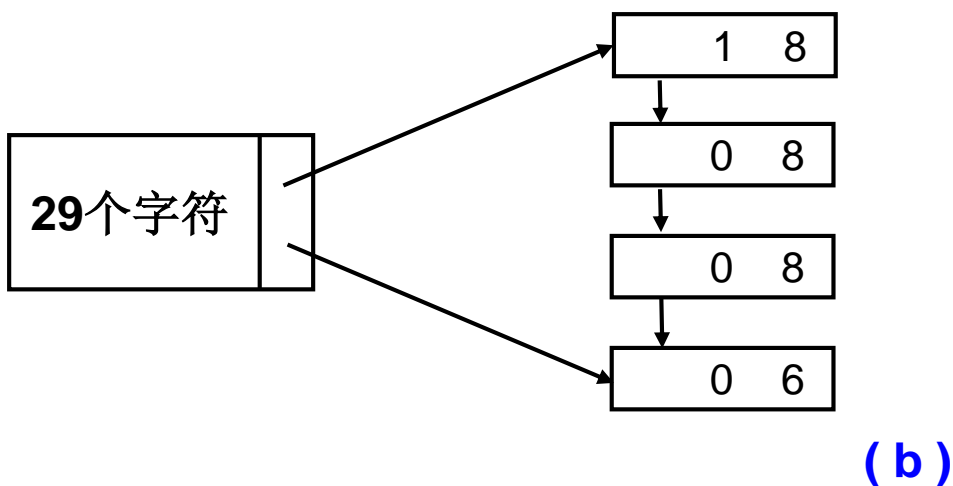
**cblocks**

k	e	r	n	e	l		o
---	---	---	---	---	---	--	---

f		U	N	I	X		o
---	--	---	---	---	---	--	---

p	e	r	a	t	i	n	g
---	---	---	---	---	---	---	---

s	y	s	t	e	m		
---	---	---	---	---	---	--	--



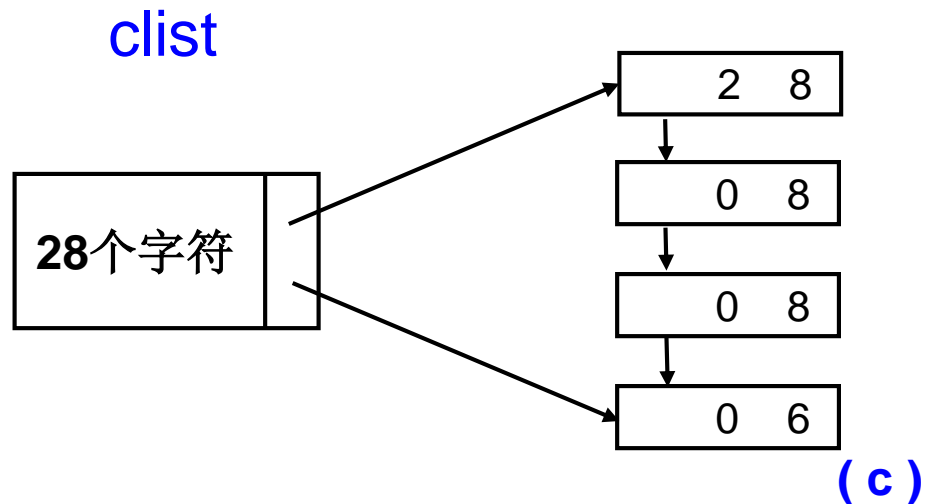
	e	r	n	e	l		o
--	---	---	---	---	---	--	---

f		U	N	I	X		o
---	--	---	---	---	---	--	---

p	e	r	a	t	i	n	g
---	---	---	---	---	---	---	---

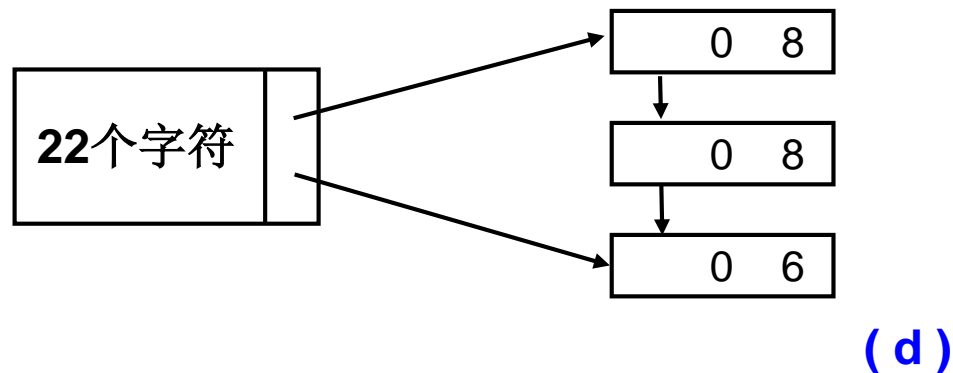
s	y	s	t	e	m		
---	---	---	---	---	---	--	--

# 图例1：从clist中移去字符 —— **getchar**（读数据）



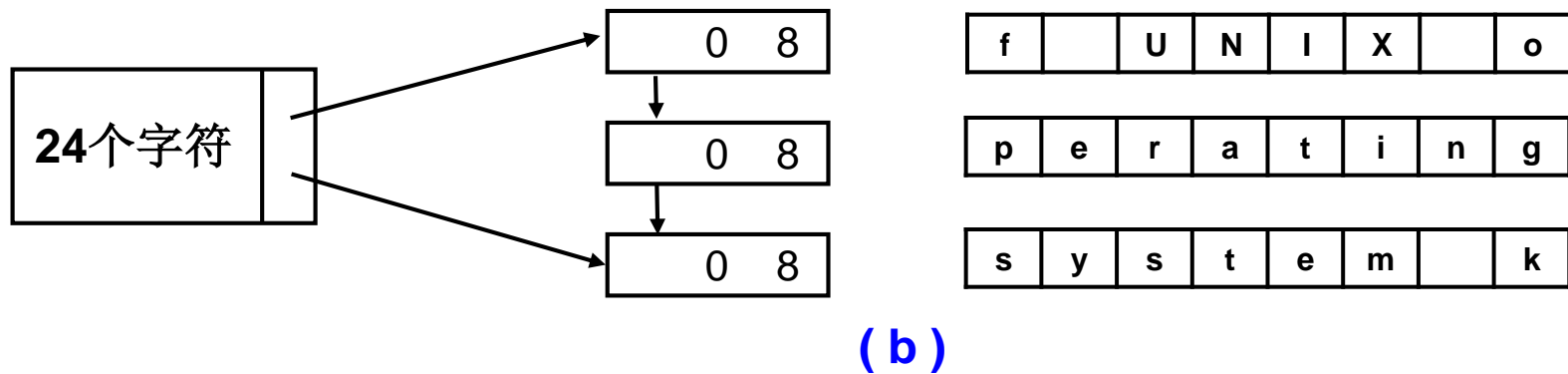
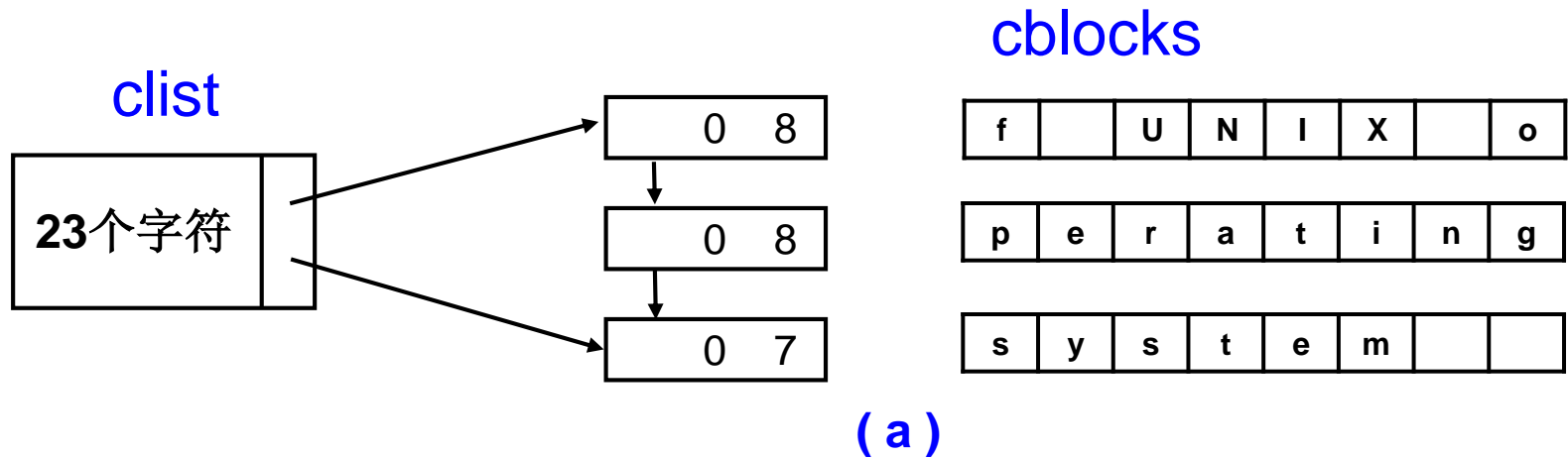
**cblocks**

		r	n	e	l		o
f		U	N	I	X		o
p	e	r	a	t	i	n	g
s	y	s	t	e	m		

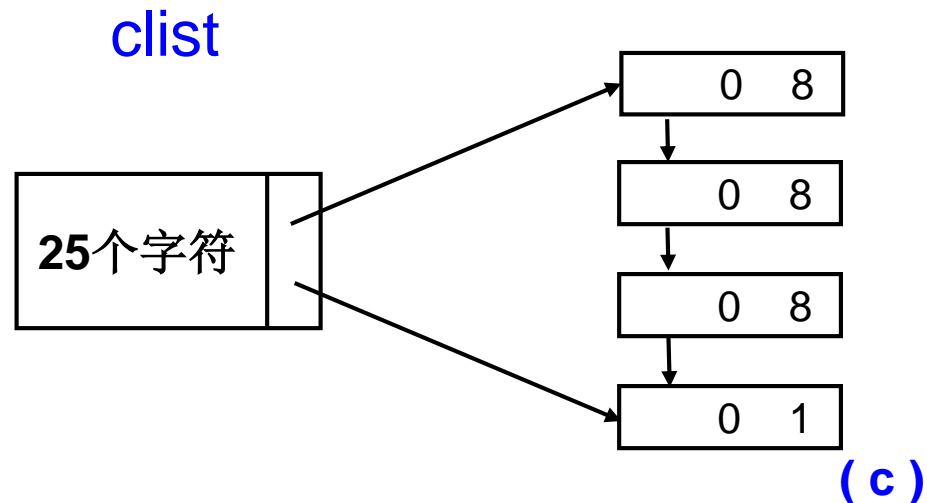


f		U	N	I	X		o
p	e	r	a	t	i	n	g
s	y	s	t	e	m		

## 图例2：向clist中放入字符 —— putchar（写数据）

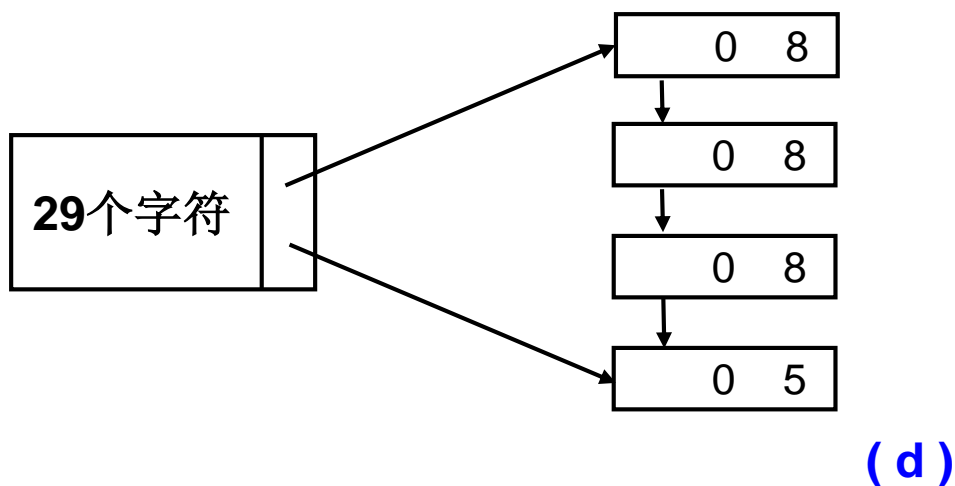


## 图例2：向clist中放入字符 —— putchar（写数据）



cblocks

f		U	N	I	X		o
p	e	r	a	t	i	n	g
s	y	s	t	e	m		k
e							



f		U	N	I	X		o
p	e	r	a	t	i	n	g
s	y	s	t	e	m		k
e	r	n	e	l			

## 2、标准方式下的终端驱动程序

在终端驱动程序的数据结构中包含三个与之关联的**clist**:

一个**clist**用来存放输出到终端去数据;

一个**clist**用来存放从终端输入的“原始”数据，这是当用户敲入字符时，终端中断处理程序放入的;

一个**clist**用来存放“已加工”的输入数据，称之为标准**clist**。

当一个进程要**向一个终端写数据**时，终端驱动程序调用行规则程序从用户地址空间读取要输出的字符，将其放入输出**clist**中。在此过程中，行规则程序对输出字符加以处理，例如将制表符扩展为一组空格。

当输出**clist**上的字符数达到最大值时，行规则程序调用终端驱动程序把**clist**中的数据发送到终端，并使写进程进入睡眠。

当输出**clist**中的字符数小于最小值时，终端驱动程序唤醒所有睡眠等待“该终端能够接受更多数据”事件的进程。

## 算法 `terminal_write`

```
{  
    while (有数据等待从用户空间输出)  
    {  
        if (tty结构填满待输出的数据)  
        {  
            启动硬件写操作以便送出“输出clist”中的数据;  
            sleep(tty能接受更多数据的事件);  
            continue;  
        }  
        从用户空间拷贝cblock大小的数据到“输出clist”中;  
        行规则程序变换制表符等;  
    }  
    启动硬件写操作以便送出“输出clist”中的数据;  
}
```

```

char form[ ] = "this is a sample output string from child";
main ( )
{
    char output [128];
    int i;
    for ( i=0; i<18; i++)
    {
        switch(fork( ))
        {
            case -1:                /* 错误——已到达最大进程数 */
                exit ( );
            default:                /* 父进程 */
                break;
            case 0:                /* 子进程 */
                /* 在变量output中形成输出字符串 */
                sprintf(output,"%s%d\n%s%d\n", form, i, form, i);
                for( ; ; )
                    write(1, output, sizeof(output));
        }
    }
}

```

多进程同时在标准输出上输出数据时的竞争



在标准方式下，当一个**进程从终端读数据**时，由终端中断处理程序把用户敲入的字符放到“原始**clist**”中，以便输入给读进程，同时还要把字符放到“输出**clist**”中，以便回显（**echo**）到终端屏幕上。

**如果输入字符中含有一个回车符，则中断处理程序要唤醒全部读进程。**

当一个读进程运行时，驱动程序从“原始**clist**”中移出字符，对擦除和抹行符进行处理，并把字符放入“标准**clist**”中去，然后再把字符拷贝到用户地址空间中去，直到接收到回车符或满足了系统调用**read**中的计数值时为止。

一个进程可能会发现，它被唤醒时的那些数据已不存在了，这是因为这个进程被调度之前，其它进程可能也从终端读，并从“原始**clist**”中移走了数据。

```

算法 terminal_read
{
    if (标准clist中无数据)
    {
        while (原始clist中无数据)
        {
            if (tty是以“不延迟”选项打开的)
                return;
            if (tty是基于定时的原始方式且定时器未激活)
                设置定时器唤醒(callout表);
            sleep(事件: 数据从终端到来);
        }
        /* 原始clist上有数据 */
        if (tty是原始方式)
            将全部数据直接从原始clist拷贝到标准clist;
        else /* tty是标准方式 */
        {
            while (原始clist中有字符)
            {
                每次从原始clist中拷贝一个字符到标准clist中, 并进行擦除、抹行处理;
                if (字符是回车符或文件尾)
                    break; /* 退出循环 */
            }
        }
    }
    while (标准clist中有字符且欲读的字节数未满足)
        从标准clist的cblock中拷贝字符到用户地址空间;
}

```

```

char input[256];
main( )
{
    register int i;
    for (i = 0; i < 18; i++)
    {
        switch(fork( ))
        {
            case -1:    /* 错误 */
                printf("error cannot fork\n");
                exit();
            default:    /* 父进程 */
                break;
            case 0:    /* 子进程 */
                for (;;)
                {
                    read(0, input, 256); /* 读入一行 */
                    printf("%d read %s\n", i, input);
                }
            }
        }
    }
}

```

示例：多进程对终端输入数据的竞争

### 3、原始方式下的终端驱动程序

进程可以通过系统调用*ioctl*来设置终端参数，如擦除符、抹行符、波特率、回显符等。行规则程序在收到系统调用*ioctl*的参数时，预置终端数据结构中有关的域。当一个进程预置终端参数时，其它所有使用该终端的进程都将使用这些参数。当预置参数的进程退出时，并不自动地复位对终端的预置。

进程也可以使终端在原始方向下工作，此时行规则程序对用户敲入的字符不作任何处理。由于“回车符”已被作为普通的输入字符了，所以核心必须要用以下**两种方法**之一来确定何时应该完成用户的系统调用*read*:

- ①、终端读入了**指定数量**的字符后；
- ②、从终端读入了任意字符后等待了**指定的时间**。

当上述条件成立时，行规则中断处理程序唤醒所有睡眠的进程，终端驱动程序把原始*clist*中的全部字符移到标准*clist*中，完成用户的读请求。

## 以原始方式从终端读入5个字符

```
#include <signal.h>
#include <termio.h>
struct termio savetty;
main()
{
    extern sigcatch();
    struct termio newtty;
    int nrd;
    char buf[32];
    signal(SIGINT, sigcatch);
    if ( ioctl(0, TCGETA, &savetty) == -1) /* 捕俘软中断信号，执行sigcatch */
    {                                       /* 保存原来的tty设置 */
        printf("ioctl failed: not a tty\n");
        exit();
    }
    newtty = savetty;
    newtty.c_lflag &= ~ICANON;           /* 停止标准方式 */
    newtty.c_lflag &= ~ECHO;             /* 停止字符回显 */
    newtty.c_cc[VMIN] = 5;               /* 最小值为5个字符 */
    newtty.c_cc[VTIME] = 10;             /* 10秒间隔 */
}
```

(接下页)

(接上页)

```
if ( ioctl(0, TCSETAF, &newtty) == -1)    /* 把终端设置为新的工作方式 */
{
    printf("cannot put tty info raw mode\n");
    exit();
}
for ( ; ; )
{
    nrd = read(0, buf, sizeof(buf));      /* 从终端读数据 */
    buf[nrd] = '\0';                      /* 设置字符串结束符 */
    printf("read %d chars '%s'\n", nrd, buf); /* 显示实际读入的字符串 */
}
}
sigcatch()    /* 在等待字符输入时，如果遇到软中断信号，则恢复原来的tty设置 */
{
    ioctl(0, TCSETAF, &savetty);
    exit();
}
```

## 4、终端探测

应用程序可能需要经常探测（等待）一个终端设备上是否有数据输入，这通常可以用一些简单的方法来实现：有数据出现就读数据；没有数据输入时，就继续正常的处理或者执行空操作等待。

下面的程序中，由于选定了以“不延迟”方式打开终端，无限循环探测终端数据。

```

#include <fcntl.h>
main()
{
    register int i, n;
    int fd;
    char buf[256];
    if ((fd = open("dev/tty", O_RDONLY | O_NDELAY)) == -1)
        exit();
    n = 10000;
    for ( ; ; )
    {
        for ( i=0; i<n; i++)
            ;
        if ( read (fd, buf, sizeof(buf)) > 0)
        {
            printf("read at n %d\n", n);
            n--;
        }
        else /* 无数据可读时，因“不延迟”而返回 */
            n++;
    }
}

```



在UNIX系统中有一个可探询多个设备的系统调用**select**，其语法格式是：

**select(nfds, rfd, wfds, efds, timeout)**

其中：

**nfds** —— 是要选择的文件描述符的数量；

**rfd** —— 指向“所选择的**读**文件描述符”的**位图**指针。如果用户要选择文件描述符**fd**，则“1”左移文件描述符的值那么多位的位置被置位；

**wfds** —— 指向“所选择的**写**文件描述符”的位图指针，置位方法与**rfd**相同；

**efds** —— 指向被**例外**条件监控的文件描述符位图；

**timeout** —— 指出为等待数据的到来**select**要睡眠多长时间，当指定时间到，无论是否有设备准备好，都返回。

## 5、建立控制终端

控制终端通常是用户注册到系统中时所用的终端，它控制用户在该终端上创建的那些进程。

当一个进程打开一个终端时，终端驱动程序就将打开行规则程序。

如果该进程是一个进程组组长，且该进程还没有一个与之相关联的终端，则行规则程序就将打开的这个终端成为该进程的控制终端。

行规则程序会把终端的主设备号和次设备号存放在该进程的u区中，并把该进程的进程组号存放在终端驱动程序的数据结构中。

控制终端在处理软中断信号时起着重要的作用。当用户按下 **delete**、**break**、**quit**、**^C**等键时，中断处理程序调用行规则程序，行规则程序向该控制进程所在组中的所有进程发出软中断信号。

当用户挂起，终端中断处理程序会从硬件收到一个挂起指示，而行规则程序向该控制进程所在组的所有进程发出一个 **SIGHUP** (**hangup**) 软中断信号，用这种方法使一个特定终端上创建的所有进程都收到**SIGHUP**信号。

收到这一信号的大多数进程的缺省动作是退出，这就是当一个用户突然关掉终端电源时，这些杂散的进程被杀掉的方法。

在发出**SIGHUP**信号后，终端中断处理程序断开该终端与该进程组的联系，使该进程组中的其它进程再也不能接收该进程发出的软中断信号了。

## 6、注册到系统

系统进入到多用户状态后，1号进程——**init**的一个主要工作就是允许用户通过终端注册进入系统。

**init**创建若干个**getty**进程，每个**getty**进程重置进程组号，打开一个特定的终端线路后，睡眠在系统调用**open**中，直到检测到与终端的硬件连接建立为止。

当**open**返回时，**getty**通过系统调用**exec**执行注册程序**login**，等待用户的登录。

如果用户注册成功，**login**通过系统调用**exec**执行**shell**，用户就可以使用系统了。该**shell**进程就是注册**shell**，与原来的**getty**进程具有相同的进程标识号，因此该**shell**进程就是一个进程组组长。

**init**进程进入睡眠状态，直到它收到一个子进程死软中断信号。当它被唤醒后，它要查明该僵死子进程是否为一个注册**shell**进程，如果是，则**init**又创建出另一个**getty**进程以便可以再次使用该终端。

```

算法  login  /* 注册的过程 */
{
    getty进程执行;
    设置进程组（系统调用setpgrp）;
    打开tty线路;          /* 睡眠直到被打开 */
    if ( 打开成功)
    {
        执行login程序;
        显示输入用户名要求;
        停止终端回显(echo)字符，显示输入口令要求;
        if ( 成功)      /* 与/etc/passwd中的口令相同 */
        {
            将tty设置为标准方式（用ioctl）;
            执行shell程序;
        }
        else
            计算企图登录的次数，小于设定值时再试;
    }
}

```

# 课程结束

# 谢谢！

信息与软件工程学院

刘 玒 教授

# 课程复习思考题

1、UNIX 操作系统最根本的功能特征是什么？包括哪些最基本的概念（动态/静态）？

2、操作系统中包括了哪些构建原语？他们的基本功能是什么？他们的基本实现流程是什么？如何用他们来构建更大的功能模块？

3、什么是纯代码编程？纯代码编程的好处是什么？

4、操作系统核心是什么？核心通过什么方式和什么原则向上层应用程序提供了哪些服务？

5、UNIX 系统假设底层硬件的工作方式是什么？

6、系统调用的基本实现方式是什么？

7、标准输入输出重定向是如何实现的？

8、数据缓冲区高速缓冲建立的基础和原则是什么？要解决的根本问题是什么？这样设置高速缓冲有什么优缺点？每个缓冲区的结构是什么？

9、改变缓冲区的大小对进程读写数据有什么影响？改变缓冲池的大小对系统性能又有什么影响？

10、UNIX 的文件系统包括了哪些大的功能模块？什么是本地文件系统？什么是虚拟文件系统？设置虚拟文件系统的优缺点是什么？

11、文件系统中目录的逻辑结构是什么样的？存储结构又是什么样的？为什么目录要采用树状结构？定长目录项的目录结构与变长目录项的目录结构有什么区别？目录结构与文件系统的容量之间有什么

什么关系？

12、资源保护系统以什么方式保护了哪些类型的资源？上锁机制的流程和特点是什么？在不同的应用场合如何选择或设定不同的上锁机制？

13、文件的 i 节点有什么样的特点和功能？在对文件进行打开、读写、关闭操作时对 i 节点进行了哪些操作？

14、用户打开文件表、系统打开文件表、活动 i 节点表分别的作用是什么？

15、文件的存储结构采用的是何种方式？进程是如何读取数据块的？这种方式有什么优缺点？还可能采用哪些其他的存储结构？

16、上层应用对普通文件和设备文件都是按相同的方式来访问的，但操作系统内部是如何分别按不同方式来执行对普通文件和设备文件的操作？

17、进程的生命周期中划分了哪几种状态？这几种状态之间是怎么转换的？状态转换的时机和条件是什么？

18、进程的核心态和用户态是怎么定义的？进程在核心态和用户态之间进行转换时的权限有哪些变化？

19、进程的控制块 PCB 包括哪些部分？他们分别包含了进程的哪些信息？

20、UNIX 系统中的中断包括了哪些类别？处理各类中断的基本流程是什么？

21、在操作系统中设置不同的处理机执行级别的目的是什么？通



常系统中设置了哪些中断级别？

22、软中断信号主要应用在哪些场合？如何使用软中断信号来实现进程间的同步和互斥？

23、进程是如何通过调用 signal 来捕俘并处理软中断信号的？  
signal 作用的范围和时间是怎样的？

23、进程调度的基本原则和方式是什么？对时间片长短的设定对系统效率的影响是什么？进程的优先级是如何划分的？

24、UNIX 系统中的进程是如何调度的？进程的调度优先数是如何计算的？

25、进程结束进入僵死状态的执行过程，释放了哪些资源？

26、块设备与字符设备在进行读写时的操作流程有哪些差异，从而在操作效果/结果上可能带来哪些不同？

### **注意：**

**1、考试时间/地点：以研管科/学生科的具体通知为准。**

**2、考试方式：开卷考试。可以携带纸质的资料，但不能携带和使用任何电子设备（包括电脑、平板、手机等）。**

## 1. UNIX 操作系统最根本的功能特征

- **根本特征：**

1. **多用户：**多个用户可以同时登录和使用资源，使用权限控制区分用户。
2. **多任务：**通过时间分片技术，实现多个进程同时运行（逻辑上的并行）。
3. **文件抽象：**一切皆文件，包括设备、管道、套接字等。
4. **分层设计：**内核（Kernel）提供底层硬件抽象，上层应用通过系统调用与内核交互。
5. **设备独立性：**应用程序通过统一接口访问设备，无需关心底层实现。

- **动态概念：**

- 进程管理：进程的创建、调度、退出。
- 内存管理：动态分配和回收内存。
- I/O 操作：设备的读写和缓冲管理。

- **静态概念：**

- 文件系统：目录结构、文件元数据。
- 权限控制：用户和组的访问权限。

---

## 2. 操作系统的构建原语

- **进程控制：**
  - fork：创建新进程，复制父进程的执行环境。
  - exec：用新程序替换当前进程的地址空间。
  - wait：父进程等待子进程完成并回收资源。
- **同步原语：**
  - 信号量：用于进程间同步。
  - 锁机制：保护临界区，避免数据竞争。
- **中断处理：**
  - 保存上下文，响应中断，恢复运行。

**实现流程：** 例如 fork 的基本流程：

1. 内核为新进程分配 PCB。
2. 复制父进程的页表和内存。
3. 将新进程添加到调度队列。

**构建更大模块：**

通过组合原语构建复杂的操作，如：

- 用信号量实现生产者-消费者模型。
  - 用文件 I/O 原语实现数据库系统。
-

### 3. 什么是纯代码编程？

- 定义：编写无状态的、可重入的代码模块，不依赖全局变量或硬件状态。
  - 好处：
    1. 提高代码的可移植性和复用性。
    2. 消除并发编程中的数据竞争问题。
    3. 模块化设计，便于测试和维护。
- 

### 4. 操作系统核心

- 定义：操作系统的内核，管理硬件资源，提供基本服务。
- 向应用程序提供服务：
  - 方式：通过系统调用接口，屏蔽底层硬件细节。
  - 原则：简单性、一致性和抽象。

服务示例：

- 文件管理：open、read、write。
  - 进程管理：fork、exec。
  - 内存管理：mmap、brk。
- 

### 5. UNIX 系统假设底层硬件的工作方式

1. 硬件提供基本资源 ( 内存、CPU、存储、I/O 设备 )。
  2. 支持中断机制，用于设备通信。
  3. 提供特权模式 ( 核心态/用户态 )。
- 

## 6. 系统调用的实现方式

- 实现方式：
    - 用户进程通过软件中断 ( 如 `int 0x80` ) 进入内核态。
    - 内核根据系统调用号定位具体服务例程。
    - 服务例程完成操作后，返回用户态。
- 

## 7. 标准输入输出重定向

- 实现方法：修改文件描述符表 ( `stdin=0` , `stdout=1` , `stderr=2` ), 将其指向新文件或设备。

示例：

`bash`

复制代码

`ls > output.txt # stdout 重定向到文件`

---

## 8. 数据缓冲区高速缓冲的原则

- **基础：**局部性原理。
  - **要解决的问题：**减少直接磁盘 I/O，提升访问速度。
  - **优缺点：**
    - 优点：减少设备瓶颈。
    - 缺点：占用更多内存，复杂性增加。
- 

## 9. 缓冲区大小对性能的影响

- **缓冲区大小对进程：**
    - 小缓冲区导致频繁 I/O。
    - 大缓冲区提高吞吐量，但内存占用增加。
  - **缓冲池大小对系统：**
    - 小缓冲池限制系统吞吐能力。
    - 大缓冲池提高性能，但可能导致内存不足。
- 

## 10. UNIX 的文件系统

- **模块：**
  1. **本地文件系统：**直接管理存储设备。
  2. **虚拟文件系统 (VFS)：**抽象多种文件系统类型。

- **虚拟文件系统优缺点：**
    - 优点：提高文件系统的可扩展性。
    - 缺点：增加实现复杂性。
- 

## 11. 文件系统中的目录结构

- **逻辑结构：**树状结构。
  - **存储结构：**
    - 定长目录项：快速访问。
    - 变长目录项：节省存储空间。
  - **树状结构优点：** 层次化管理，提高查找效率。
- 

## 12. 资源保护系统

- **保护方式：**
    - 文件权限 ( rwx )。
    - 进程间访问控制。
  - **锁机制：**
    - 流程：请求锁 -> 检查资源 -> 执行操作。
    - 特点：防止死锁和资源冲突。
-

### 13. 文件的 i 节点

- **特点：**存储文件元信息，如大小、权限、指针。
  - **操作：**
    - 打开文件时，加载 i 节点到内存。
    - 关闭文件时，更新 i 节点信息。
- 

### 14. 文件表的作用

- **用户打开文件表：**记录用户级文件句柄。
  - **系统打开文件表：**跟踪全局文件状态。
  - **活动 i 节点表：**缓存活跃的文件元数据。
- 

### 15. 文件存储结构

- **索引节点方式：**快速定位文件块。
- **优缺点：**
  - 优点：随机访问效率高。
  - 缺点：实现复杂。
- **其他方式：**
  - 链接法。



- 多级索引。
- 

## 16. 普通文件与设备文件的操作

内部通过设备类型区分，普通文件通过磁盘 I/O，设备文件通过驱动直接操作硬件。

---

## 17. 进程状态

- 状态：
    - 就绪：等待运行。
    - 运行：占用 CPU。
    - 等待：等待资源。
  - 状态转换条件：
    - 就绪 -> 运行：被调度。
    - 运行 -> 等待：I/O 请求。
    - 等待 -> 就绪：I/O 完成。
- 

## 18. 核心态和用户态

- 定义：
  - 用户态：受限操作，访问内核需通过系统调用。
  - 核心态：有特权访问硬件。

- **权限变化：** 用户态调用内核态功能需通过特定接口（系统调用）。
- 

## 19. PCB ( 进程控制块 )

- **包含：**
    - 进程标识。
    - 寄存器上下文。
    - 打开文件表。
- 

## 20. 中断类别

- **时钟中断：** 调度和计时。
- **I/O 中断：** 处理设备事件。
- **异常：** 内存访问错误等。
- **处理流程：**
  1. 保存现场。
  2. 调用中断处理程序。
  3. 恢复现场。