



操作系统

# 第六章 设备管理

文艳军

计算机学院

# 实验讲授计划

- ① 设备I/O子系统和块设备
- ② 字符设备和存储设备

# Linux 0.11

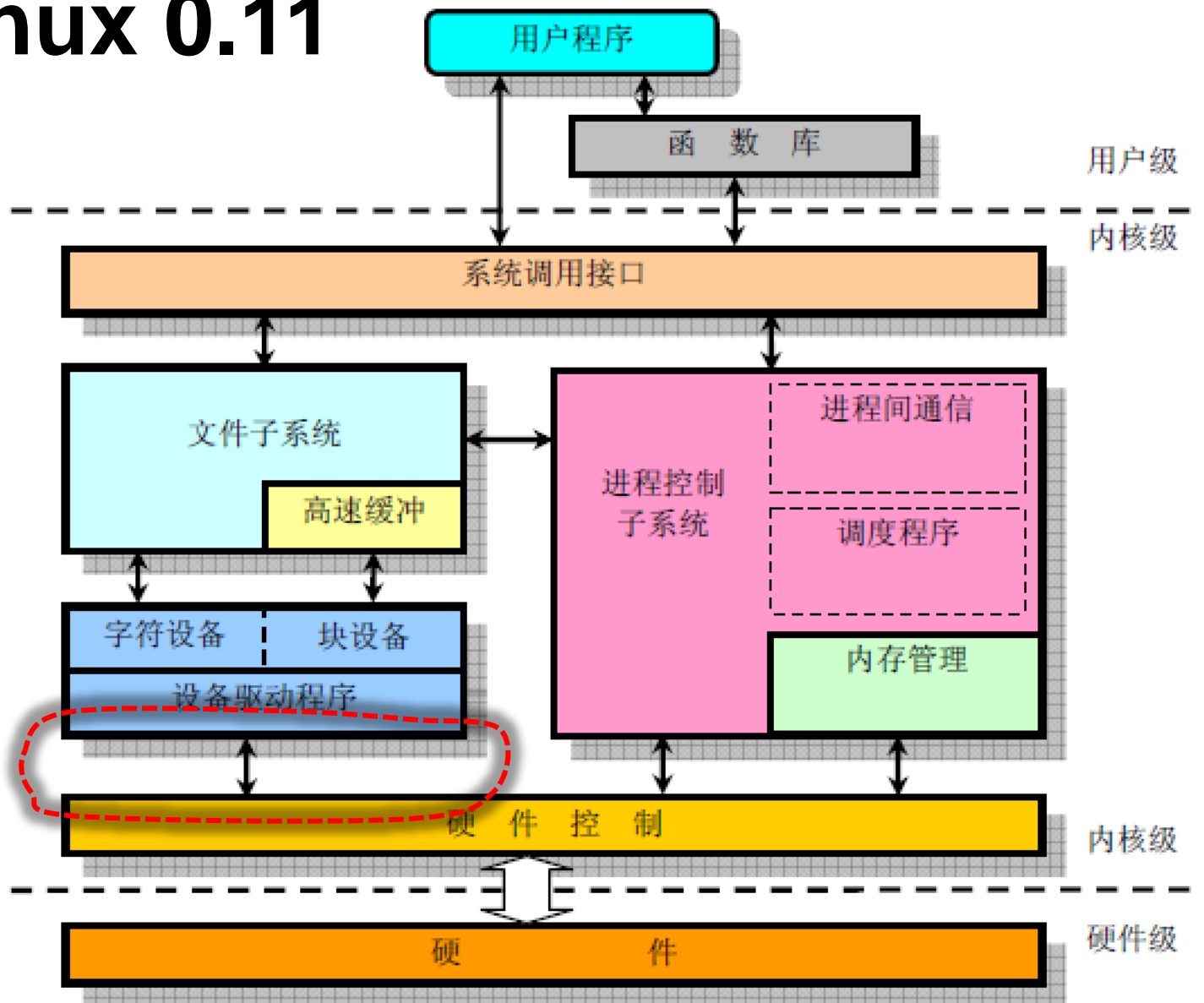
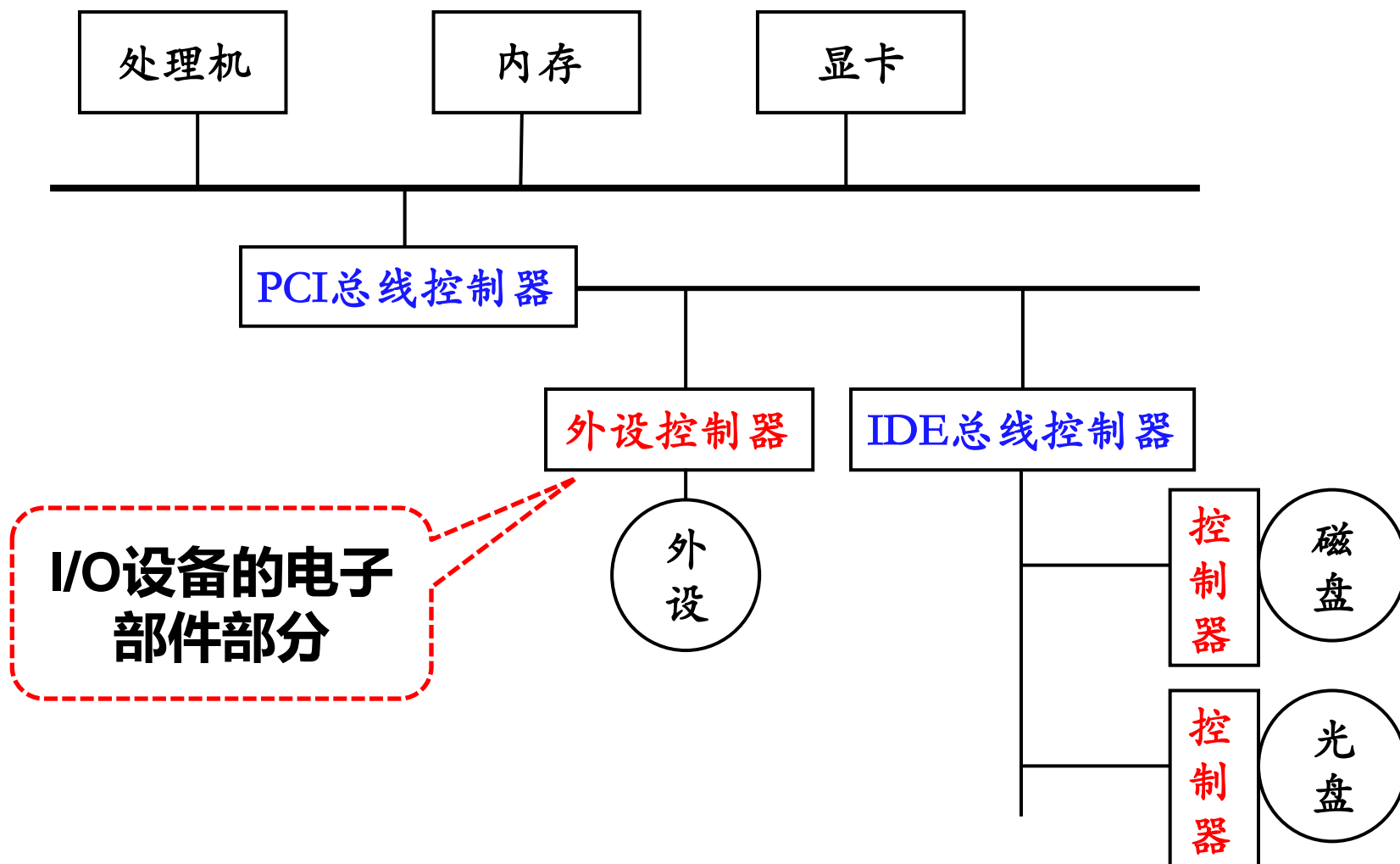


图 2-4 内核结构框图

# 目录

- 一. 设备控制器
- 二. 设备的使用方法
- 三. 输入输出的层次结构
- 四. 演示: 读硬盘文件

# 设备控制器



# 设备控制器

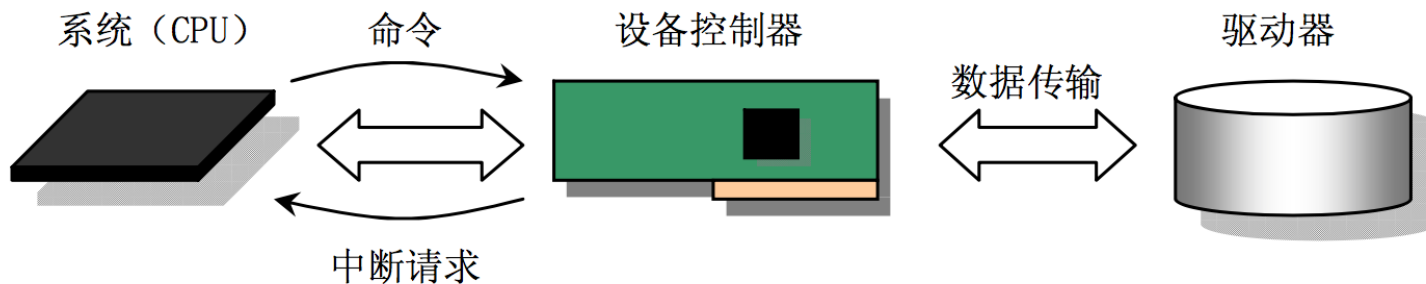


图 6-2 系统、块设备控制器和驱动器

# 设备控制器端口 (Port)

- 端口：用来与CPU通讯的**I/O寄存器**，可传输命令、状态、数据等。
- I/O端口地址
  - I/O映射方式(I/O-mapped)：独立地址空间，专门指令
  - 内存映射方式(Memory-mapped)：映射到内存的物理地址空间，普通访存指令

# 例：Linux 0.11的硬盘读写函数

```
static void hd_out(unsigned int drive, unsigned int nsect, unsigned int sect,  
    unsigned int head, unsigned int cyl, unsigned int cmd,  
    void (*intr_addr)(void))  
{  
    register int port asm("dx");  
  
    if (drive>1 || head>15)  
        panic("Trying to write bad sector");  
    if (!controller_ready())  
        panic("HD controller not ready");  
    do_hd = intr_addr;  
    outb_p(hd_info[drive].ctl, HD_CMD);  
    port=HD_DATA;  
    outb_p(hd_info[drive].wpcom>>2, ++port);  
    outb_p(nsect, ++port);  
    outb_p(sect, ++port);  
    outb_p(cyl, ++port);  
    outb_p(cyl>>8, ++port);  
    outb_p(0xA0 | (drive<<4) | head, ++port);  
    outb_p(cmd, ++port);  
} ? end hd_out ?
```

设置中断处理程序

写端口，发出命令



# Linux 0.11

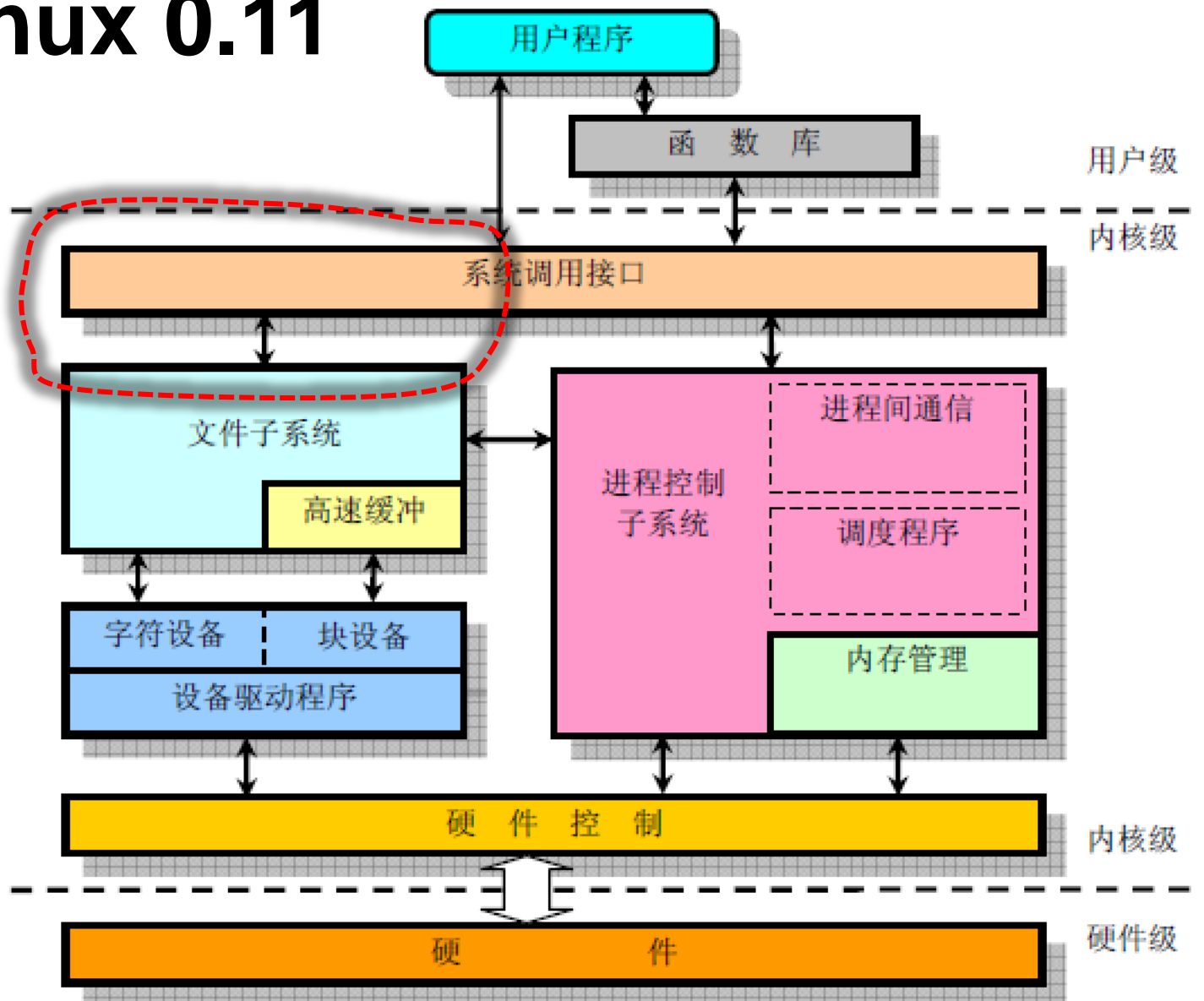


图 2-4 内核结构框图

# 目录

- 一. 设备控制器
- 二. 设备的使用方法
- 三. 输入输出的层次结构
- 四. 演示: 读硬盘文件

## 2.1 设备的使用方法

### ■ 三种使用管理方式

- 独占式：如摄像头
- 分时式：如硬盘
- SPOOLing虚拟设备方式：如打印机



## 2.1 设备的使用方法

- 人机交互类慢速外设：提供一套系统调用
- 存储类外设：通过文件管理系统调用
- 网络通讯外设：SOCKET通讯系统调用

## 2.1 设备的使用方法

### ■ 设备相关系统调用

- 申请设备 (open)
- 将数据写入设备 (write)
- 从设备读取数据 (read)
- 释放设备 (close)

## 2.1 设备的使用方法

### ■ 设备相关系统调用

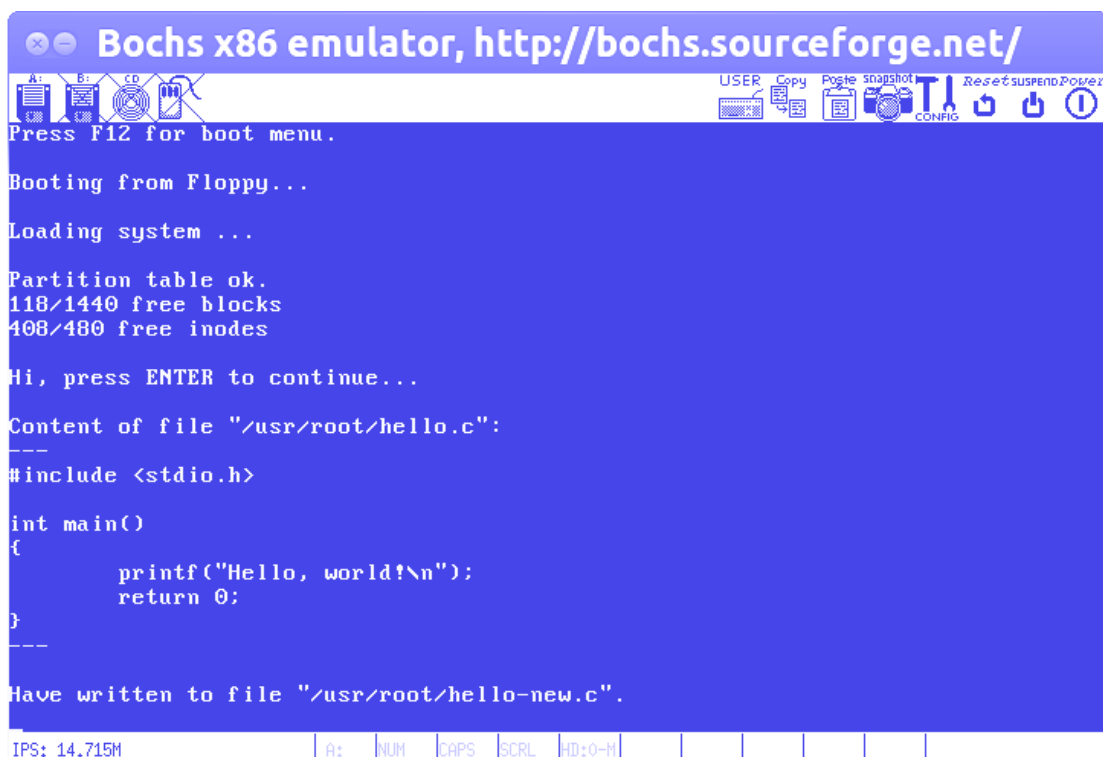
举例：在Linux中，直接写硬盘：

```
fd=open("/dev/sda", O__RDRW);  
lseek(fd, 1024, 0);           // 定位  
write(fd, buffer, 36);  
...  
close(fd) ;
```

注：该方式直接写硬盘，须谨慎使用。

# 实验内容(版本2内核)

## ■ 分析一个只具有两个进程的Linux 0.11内核映像的执行过程



```
Bochs x86 emulator, http://bochs.sourceforge.net/
Press F12 for boot menu.
Booting from Floppy...
Loading system ...
Partition table ok.
118/1440 free blocks
408/480 free inodes
Hi, press ENTER to continue...
Content of file "/usr/root/hello.c":
---
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
---
Have written to file "/usr/root/hello-new.c".

IPS: 14,715M | A: | NUM | CAPS | SCRL | HD:0-M |
```



Press F12 for boot menu.

Booting from Floppy...

Loading system ...

Partition table ok.

118/1440 free blocks

408/480 free inodes

Hi, press ENTER to continue...

Content of file "/usr/root/hello.c":

---

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

```
int main()
```

```
{  
    printf("Hello, world!\n");  
    return 0;
```

```
}
```

---

Have written to file "/usr/root/hello-new.c".



# main函数结构

```
void main(void)          /* This really IS void, no error here. */
{
    /* The startup routine assumes (well, ...) this */
    /*
    * Interrupts are still disabled. Do necessary setups, then
    * enable them
    */
```

```
    move_to_user_mode();
    if (!fork()) {        /* we count on this going ok */
        init();
    }
    /*
    * NOTE!! For any other task 'pause()' would mean we have to get a
    * signal to awaken, but task0 is the sole exception (see 'schedule()')
    * as task 0 gets activated at every idle moment (when no other tasks
    * can run). For task0 'pause()' just means we go check if some other
    * task can run, and if not we return here.
    */
    for(;;) pause();
} ? end main ?
```

# init函数结构

```
#define MSG_LEN 80
#define FILE_FROM "/usr/root/hello.c"
#define FILE_TO "/usr/root/hello-new.c"

void init(void)
{
    int pid, i, num, fd;
    char msg[MSG_LEN+1];

    setup((void *) &drive_info);
    (void) open("/dev/tty0", O_RDWR, 0);
    (void) dup(0);
    (void) dup(0);

    /* welcome */
    printf("\n\rHi, press ENTER to continue...\n\r");
    read(0, msg, 1);
}
```

```
/* open and read the file */
```

```
if ((fd=open(FILE_FROM,O_RDONLY,0)) < 0) {
```

```
    printf("Open file error.\n\r");
```

```
    _exit(1);
```

```
}
```

```
if ((num = read(fd, msg, MSG_LEN))<0) {
```

```
    printf("Read file error.\n\r");
```

```
    _exit(1);
```

```
}
```

```
close(fd);
```

```
/* display on screen */
```

```
msg[num] = '\0';
```

```
printf("Content of file \"%s\":\n\r---\n\r%s---\n\r",
```

```
    FILE_FROM, msg);
```

```
/* write in a file */
```

```
if ((fd=open(FILE_TO, O_WRONLY|O_CREAT, S_IWUSR)) < 0) {
```

```
    printf("Create file error.\n\r");
```

```
    _exit(1);
```

```
}
```

```
if (write(fd, msg, num) < 0) {
```

```
    printf("Write file error.\n\r");
```

```
    _exit(1);
```

```
}
```

```
printf("\n\rHave written to file \"%s\".\n\r", FILE_TO);
```

```
/* exit */
```

```
close(0);close(1);close(2);close(fd);
```

```
sync();
```

```
while(1)
```

```
;
```

# Linux 0.11

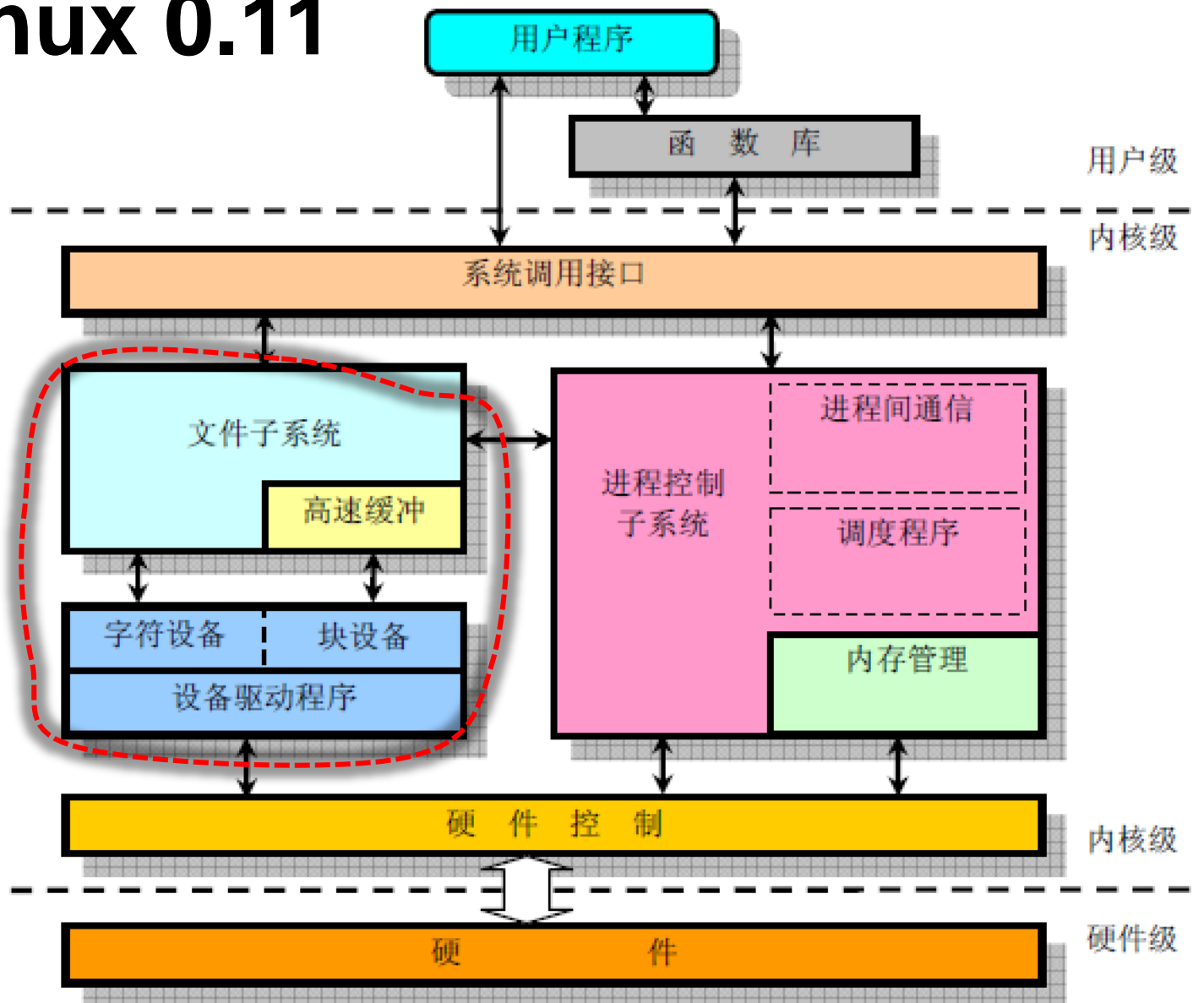


图 2-4 内核结构框图

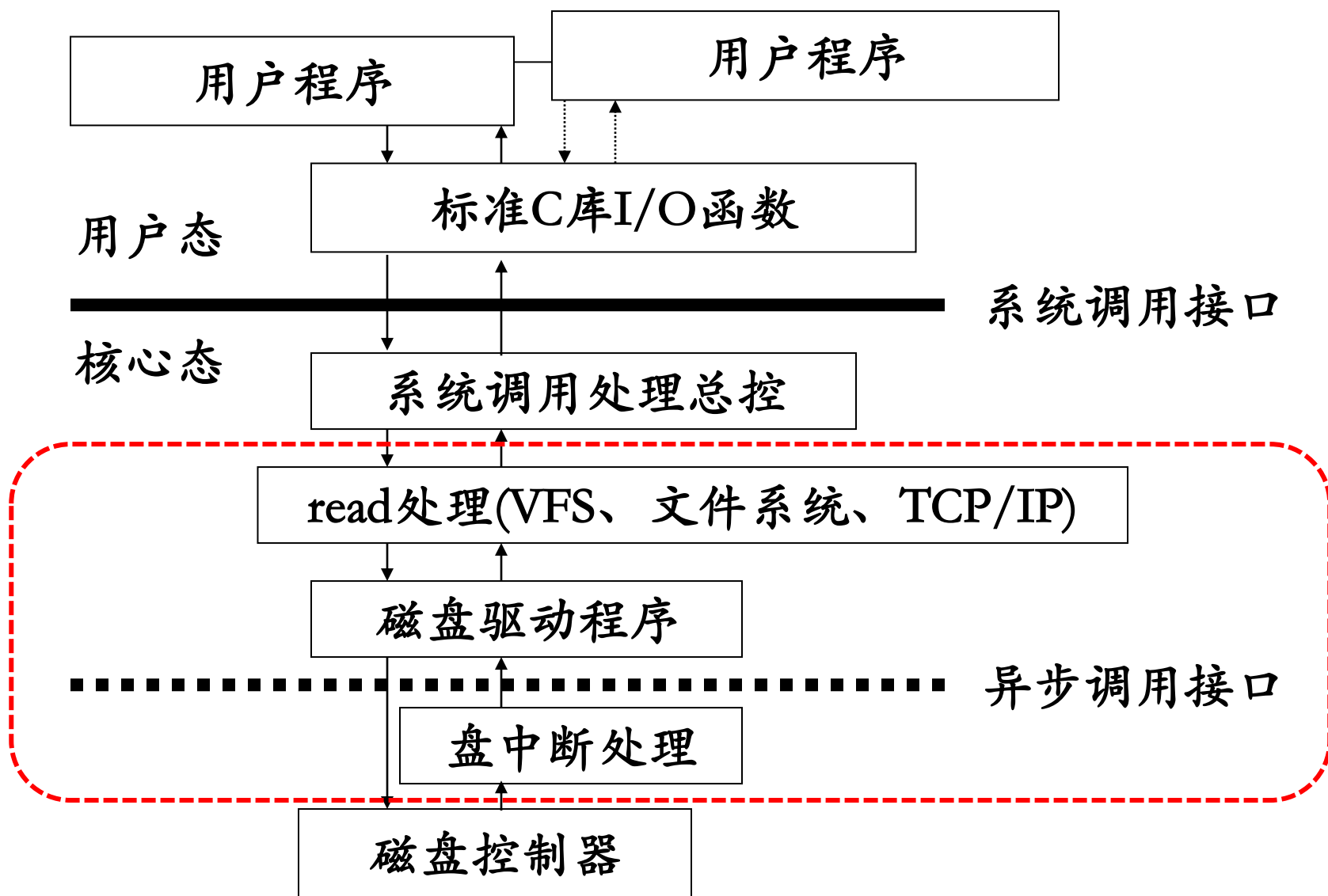
# 目录

- 一. 设备控制器
- 二. 设备的使用方法
- 三. 输入输出的层次结构
- 四. 演示: 读硬盘文件

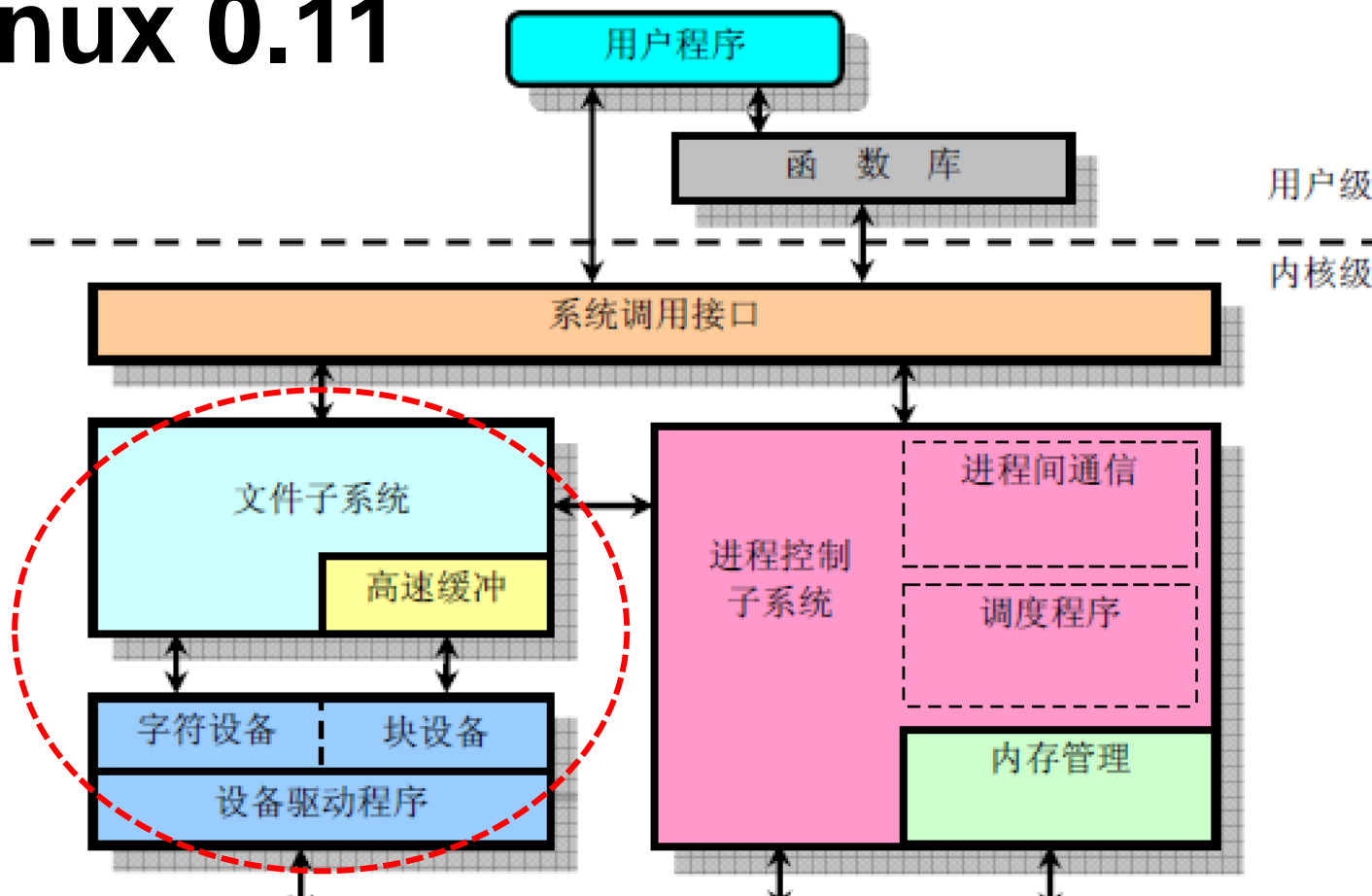
## 2.2 输入输出层次结构



# read系统调用处理各模块结构图



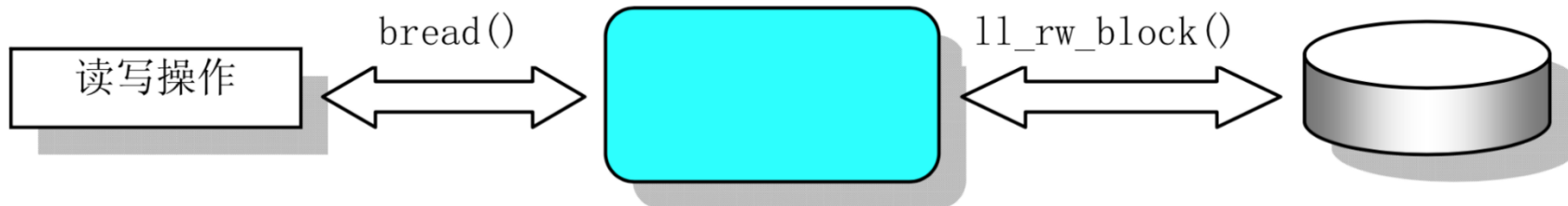
# Linux 0.11



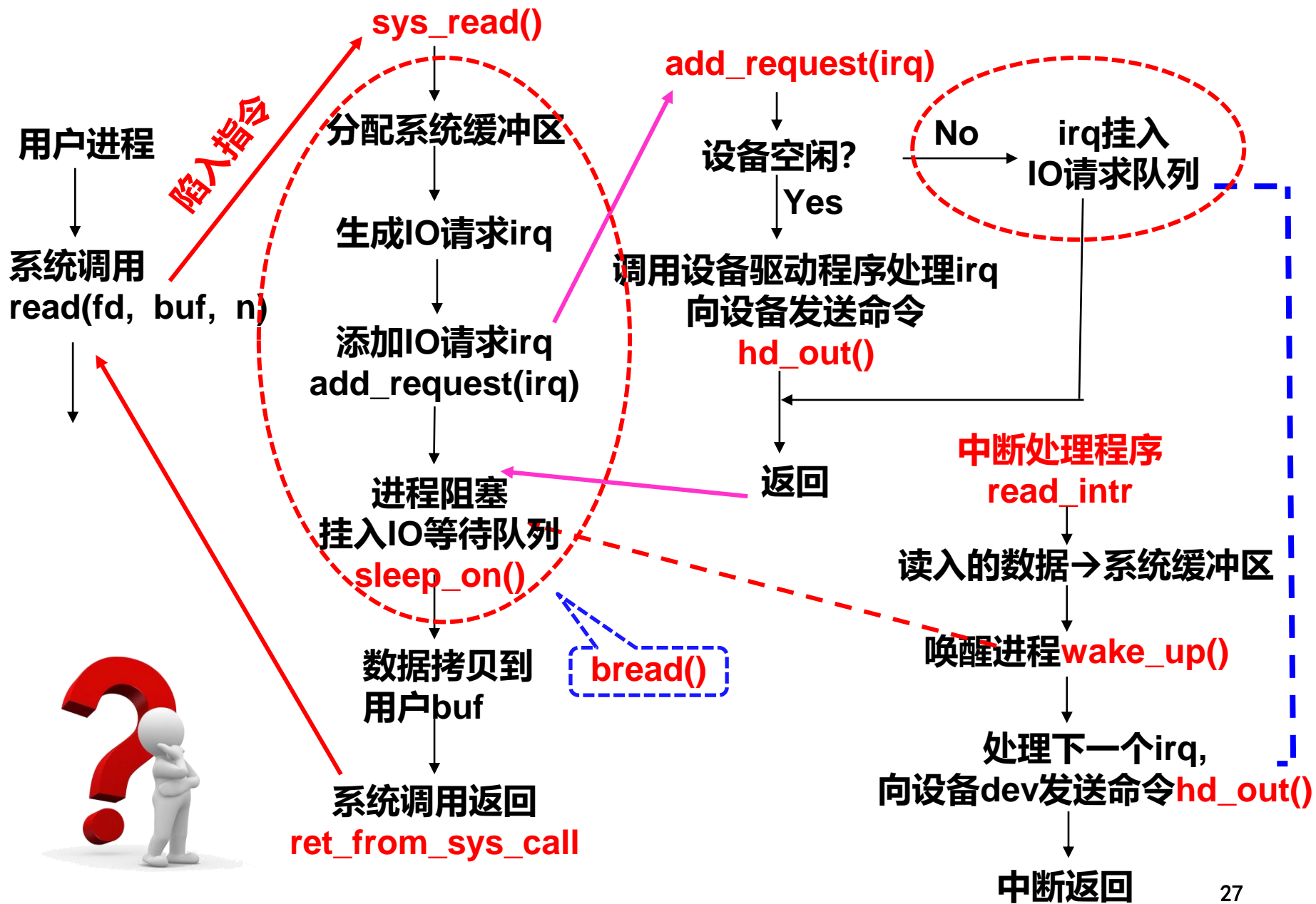
内核上层程序

高速缓冲区

块设备



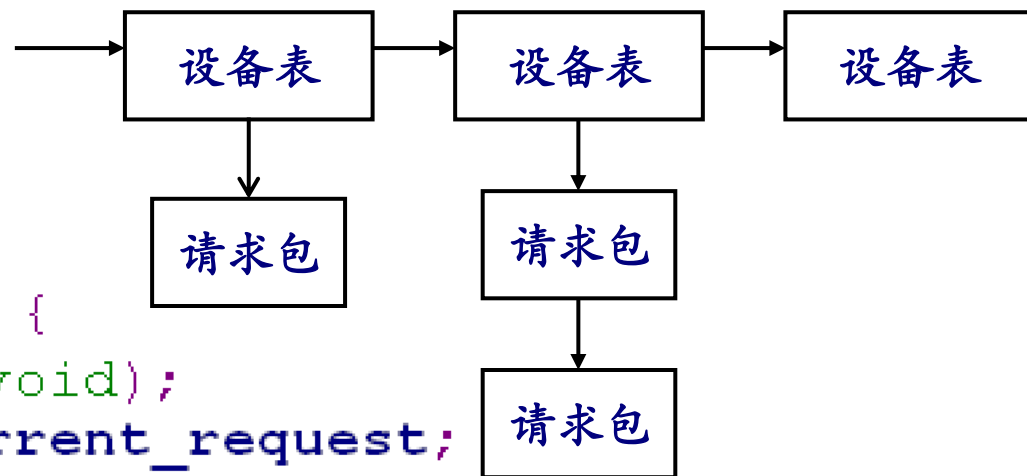




# 目录

- 一. 设备控制器
- 二. 设备的使用方法
- 三. 输入输出的层次结构
- 四. 演示: 读硬盘文件

# Linux 0.11例



```
struct blk_dev_struct {  
    void (*request_fn)(void);  
    struct request * current_request;  
};
```

```
extern struct blk_dev_struct blk_dev[NR_BLK_DEV];  
extern struct request request[NR_REQUEST];
```

```
struct request {  
    int dev;           /* -1 if no request */  
    int cmd;           /* READ or WRITE */  
    int errors;  
    unsigned long sector;  
    unsigned long nr_sectors;  
    char * buffer;  
    struct task_struct * waiting;  
    struct buffer_head * bh;  
    struct request * next;  
};
```

# 实例:Linux 0.11的块设备读操作

## ■ 演示1:

#1进程读文件hello.c

- 位置： 相关函数
- 数据： 请求队列、请求等

## 请求队列头指针

```
static void add_request(struct blk_request_queue *dev, struct request *req)
{
    struct request *tmp;

    req->next = NULL;
    cli();
    if (req->bh)
        req->bh->b_dirt = 0;
    if (!(tmp = dev->current_request)) {
        dev->current_request = req;
        sti();
        (dev->request_fn)();
        return;
    }
    for ( ; tmp->next ; tmp=tmp->next)
        if ((IN_ORDER(tmp, req) ||
             !IN_ORDER(tmp, tmp->next)) &&
            IN_ORDER(req, tmp->next))
            break;
    req->next=tmp->next;
    tmp->next=req;
    sti();
}
? end add_request?
```



### add\_request(irq)



## 设备的请求处理函数

# 小结

## 一. 设备控制器

端口

## 二. 设备的使用方法

文件、系统调用

## 三. 输入输出的层次结构

设备无关层、设备相关层

## 四. 演示: 读硬盘文件

# 小结

## 一. I/O硬件概念

设备控制器、I/O控制方式

## 二. 设备I/O子系统

设备的使用方法、输入输出层次  
结构、缓冲技术

## 三. 存储设备

# 作业

- 实训：6.2.3-观察从键盘输入的口  
令
- 大实验：命令解释器开发
- 原理作业



## 2.2 输入输出层次结构



### 设备驱动程序接口函数

- I/O操作函数
- 中断处理函数
- 申请设备函数
- 释放设备函数
- 驱动程序初始化函数
- 驱动程序卸载函数



# 操作系统

## 第六章：设备管理

### 第2讲：字符设备和存储设备

文艳军

计算机学院

# 回顾

## 一. I/O硬件概念

设备控制器、I/O控制方式

## 二. 设备I/O子系统

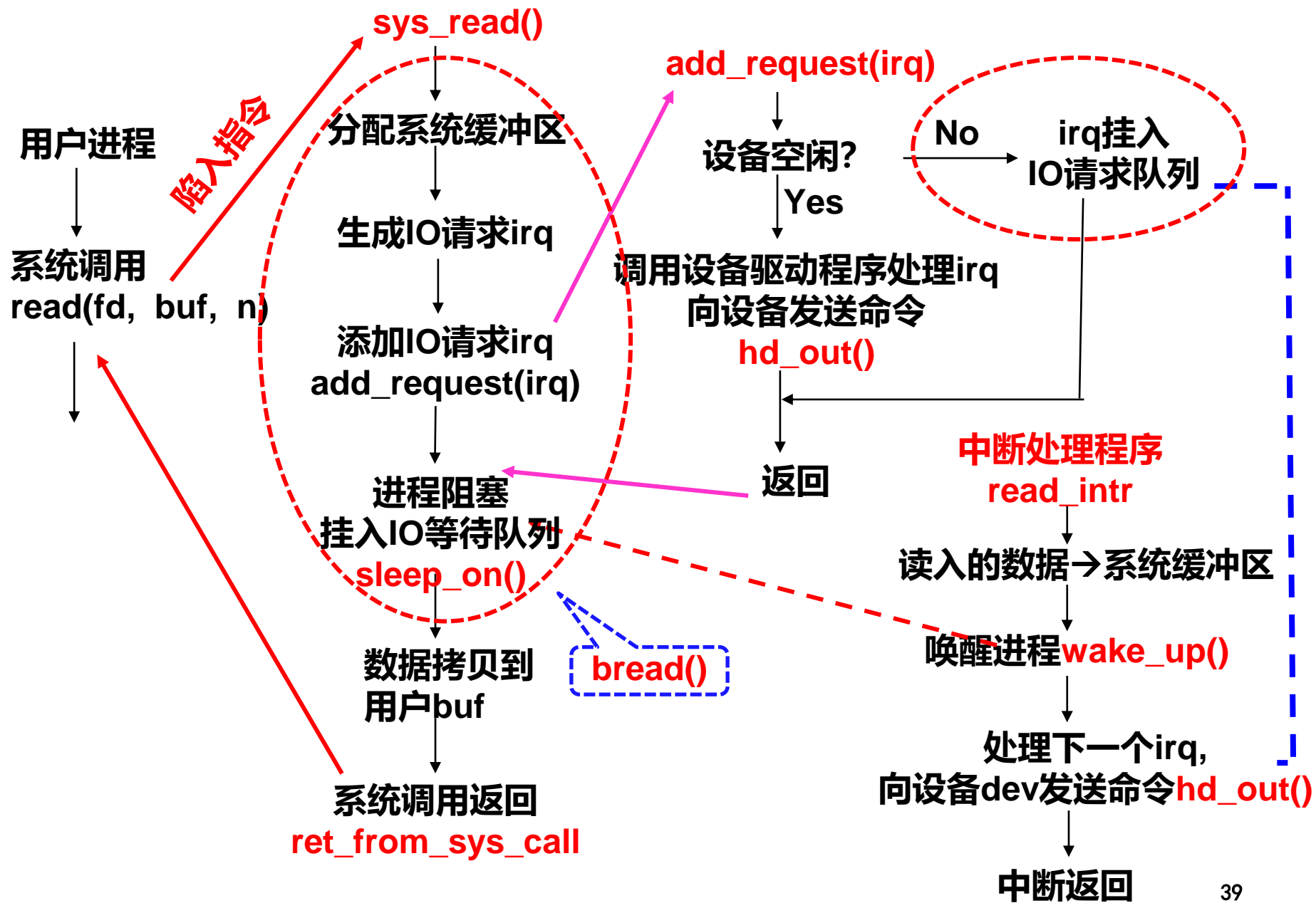
设备的使用方法、输入输出层次结构、缓冲技术

## 三. 存储设备

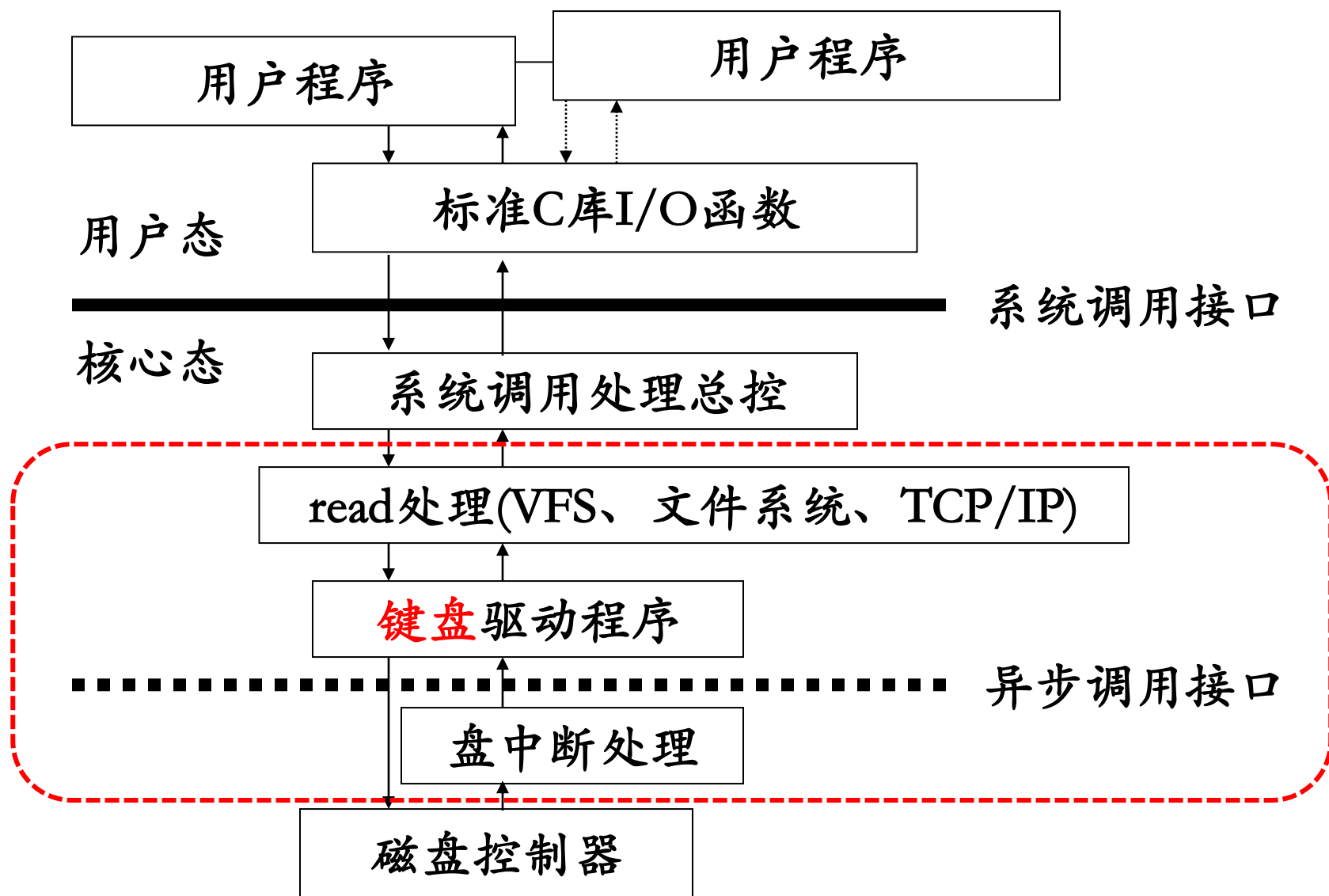
常见的存储外设、盘阵与逻辑卷

# 目录

- 一. 对字符设备的访问
- 二. I/O控制方式
- 三. 内存缓冲技术
- 四. 常见的存储外设
- 五. 盘阵与逻辑卷



# read系统调用处理各模块结构图



```

sys_read(fd, buf, count) {
    tty_read(...){
        while(...) {
            if(...){
                sleep_if_empty(&tty_secondary)
                continue; //tty_io.c:259
            }
            从tty_table[0].secondary读取数据到buf
        }
    } //tty_io.c:288
}

```

```

keyboard_interrupt() {
    将数据从键盘控制器拷贝到tty_table[0].read_q
    do_tty_interrupt(...){
        将tty_table[0].read_q中的未读数据复制到-
        tty_table[0].secondary
        wake_up(...);
    }
}

```

# 实例:Linux 0.11的字符设备读操作

## ■ 演示1:

### #1进程读键盘

- 位置： 相关函数
- 数据： 键盘的数据队列



# 目录

一. 对字符设备的访问

二. I/O控制方式

三. 内存缓冲技术

四. 常见的存储外设

五. 盘阵与逻辑卷

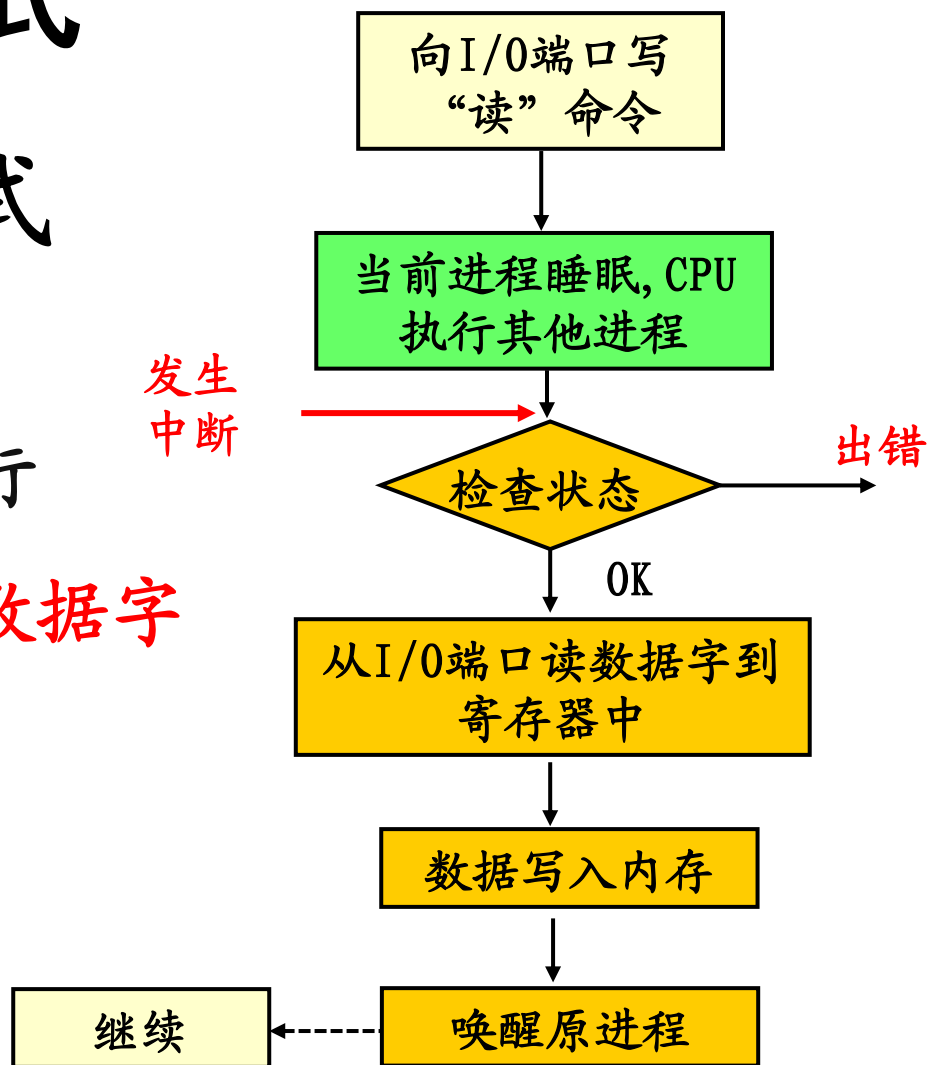
## 二. I/O控制方式

- ① 直接控制方式
- ② 中断控制方式
- ③ DMA控制方式

## 二. I/O控制方式

### ② 中断控制方式

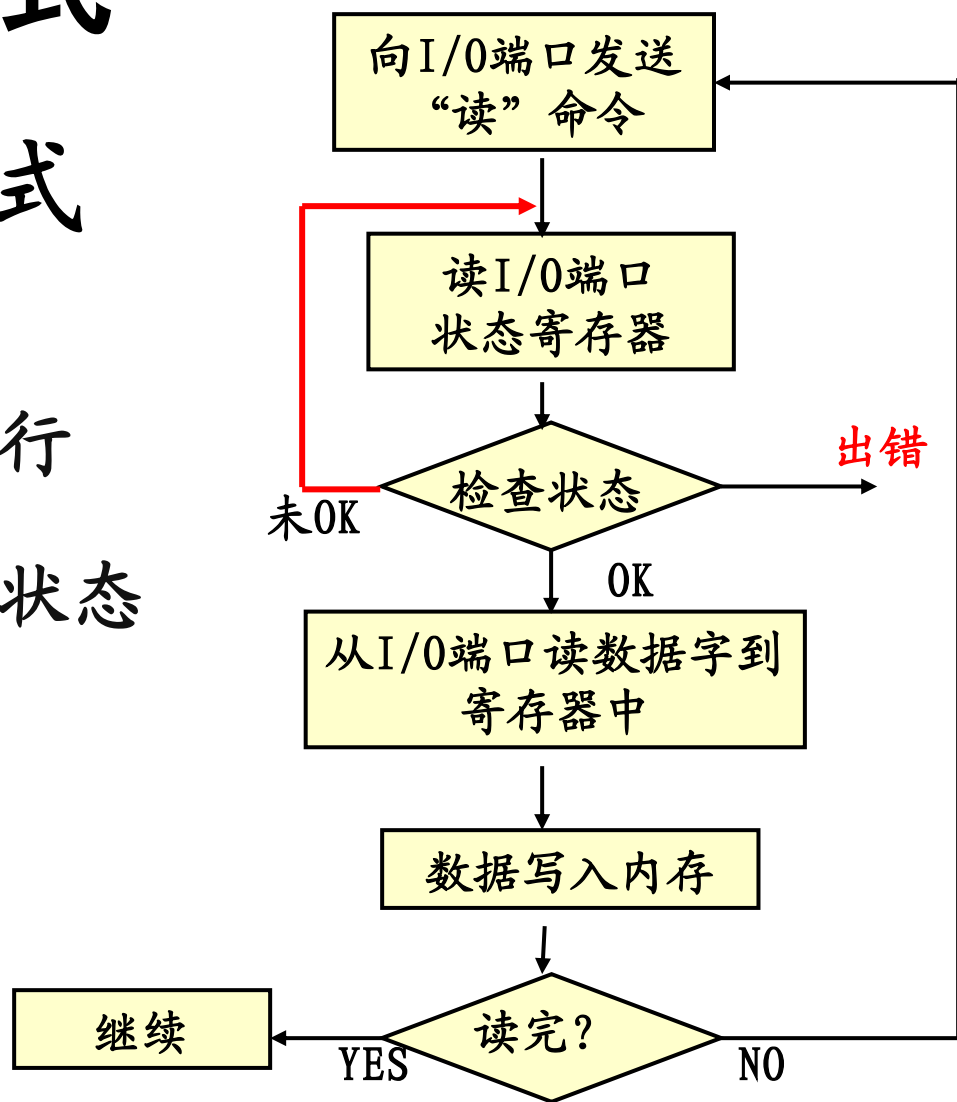
- ➡ CPU与外设可以并行
- ➡ CPU需要控制每个数据字的传输
- ➡ 中断次数多



## 二. I/O控制方式

### ① 直接控制方式

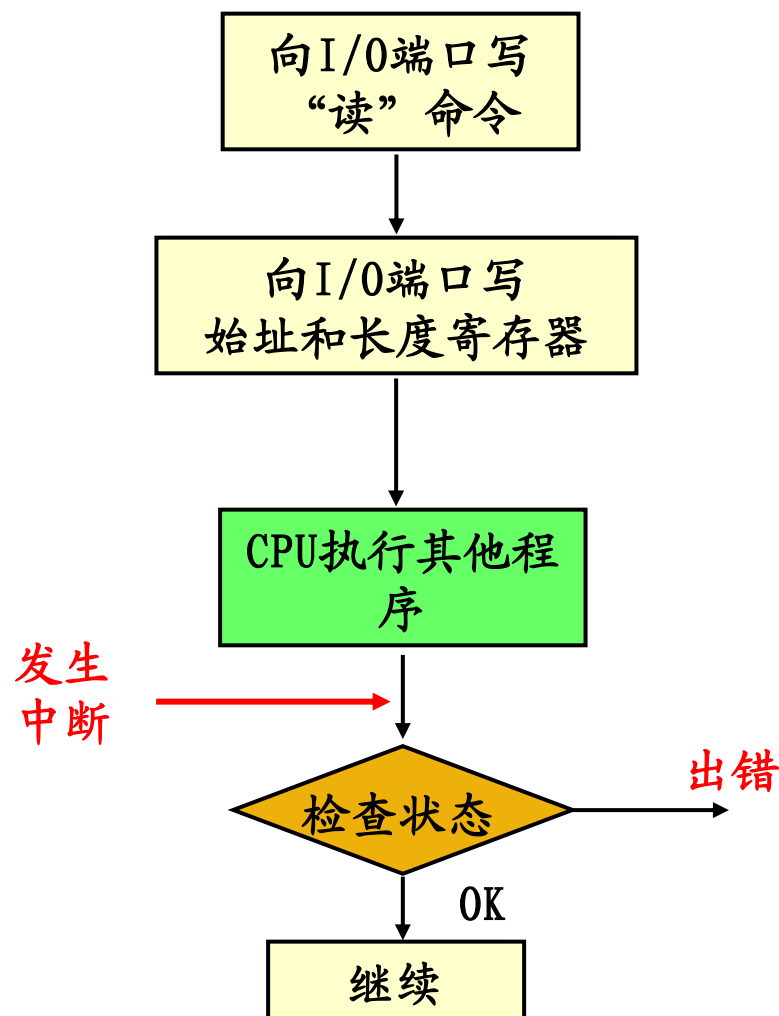
- ➡ CPU与外设不能并行
- ➡ CPU不断查询外设状态
- ➡ CPU效率极低



## 二. I/O控制方式

### ③ DMA控制方式

- ➡ 由DMA硬件直接访问内存，完成数据块的传输
- ➡ CPU与外设的并发粒度增大
- ➡ 中断次数少



## 二. I/O控制方式

设备自主性



直接控制方式  
中断控制方式  
DMA控制方式

CPU与设备的  
并发性



采用下列哪几种方式时，在将数据从设备控制器传输到内存的过程中，需要经过CPU寄存器的中转？

A

直接控制方式

B

中断控制方式

C

DMA控制方式

提交

# 目录

- 一. 对字符设备的访问
- 二. I/O控制方式
- 三. 内存缓冲技术
- 四. 常见的存储外设
- 五. 盘阵与逻辑卷

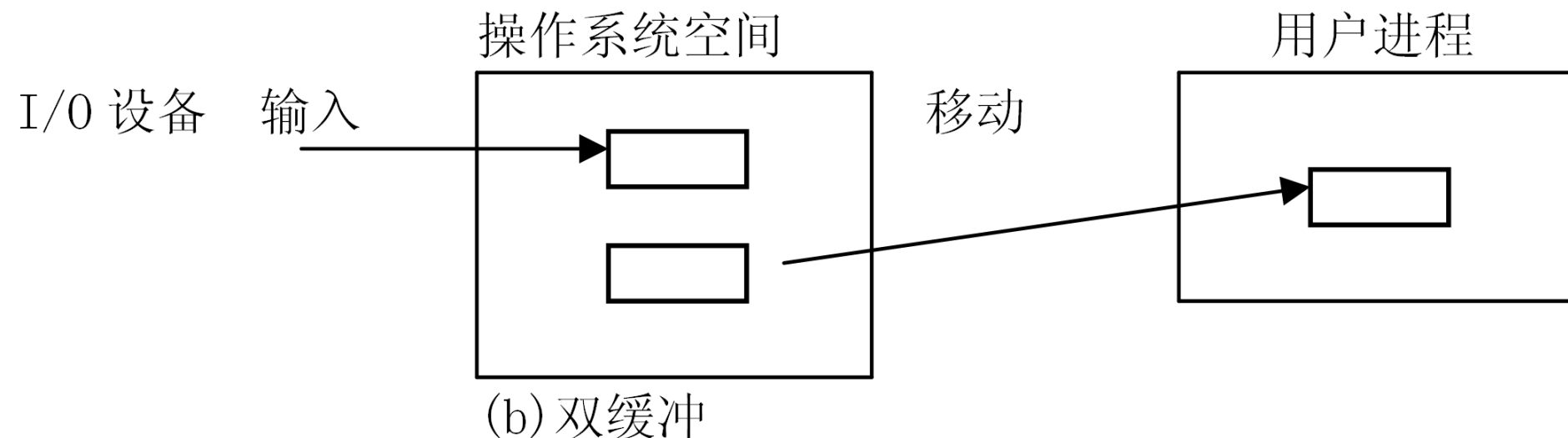


# 三. 缓冲技术

在进程工作区与外设之间：

一、单缓冲：预读和滞后写

二、双缓冲：可以实现用户数据区—缓冲区之间交换数据和缓冲区—外设之间交换数据并行



# 三. 缓冲技术

在进程工作区与外设之间：

一、单缓冲：预读和滞后写

二、双缓冲：可以实现用户数据区—缓冲区之间交换数据和缓冲区—外设之间交换数据并行

三、循环缓冲：有限缓冲区的生产者/消费者模型

➤ 缓冲技术对具有重复性及突发性的I/O操作性能提升很有帮助。

# Linux 0.11/0.12中的内存高速缓冲区

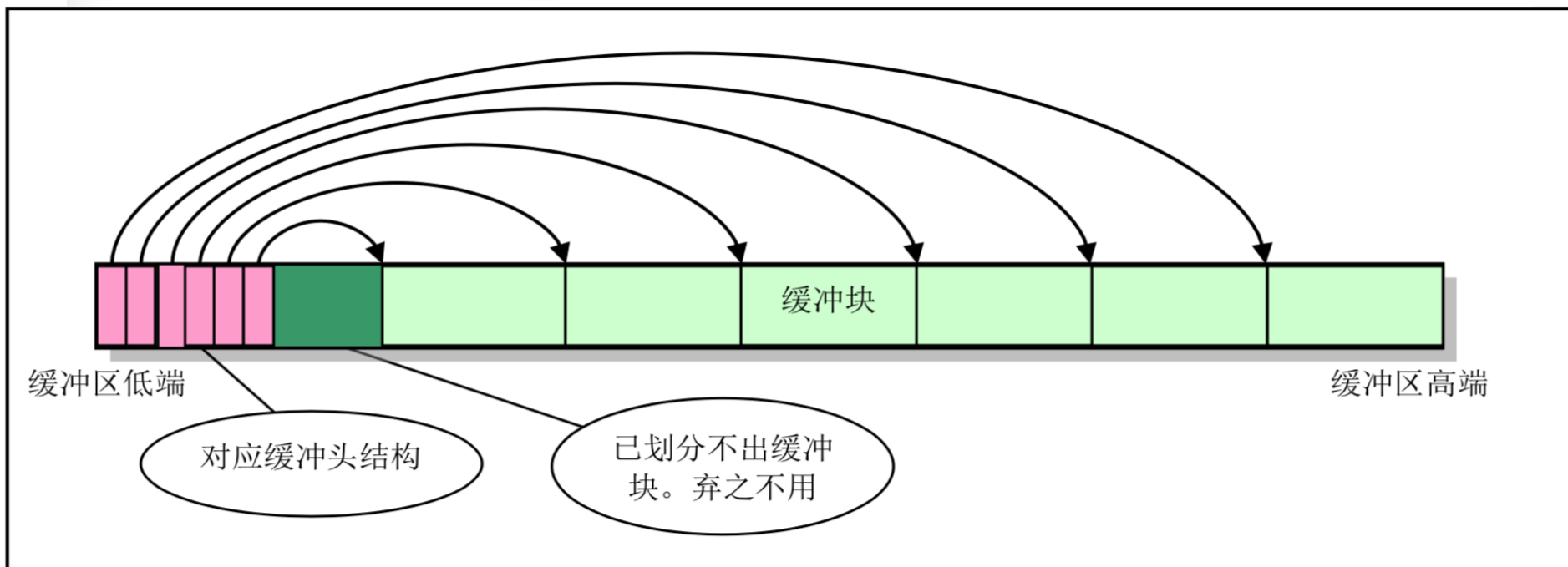
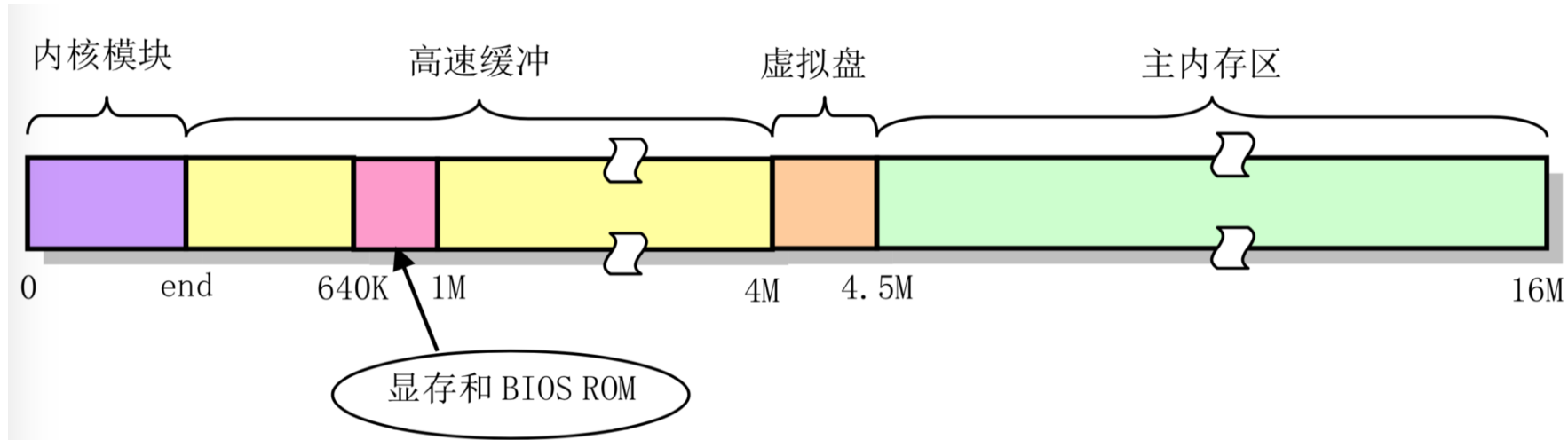


图 12-16 高速缓冲区的初始化

# 目录

- 一. 对字符设备的访问
- 二. I/O控制方式
- 三. 内存缓冲技术
- 四. 常见的存储外设
- 五. 盘阵与逻辑卷

## 四. 常见的存储外设

- 磁盘

- 光盘 (CD-ROM, CD-RW等)

- 闪存 (U盘, 固态硬盘)

  - 优点：低功耗、无噪声、抗震动

  - 缺点：成本较高、写入次数受限

# 四. 常见的存储外设

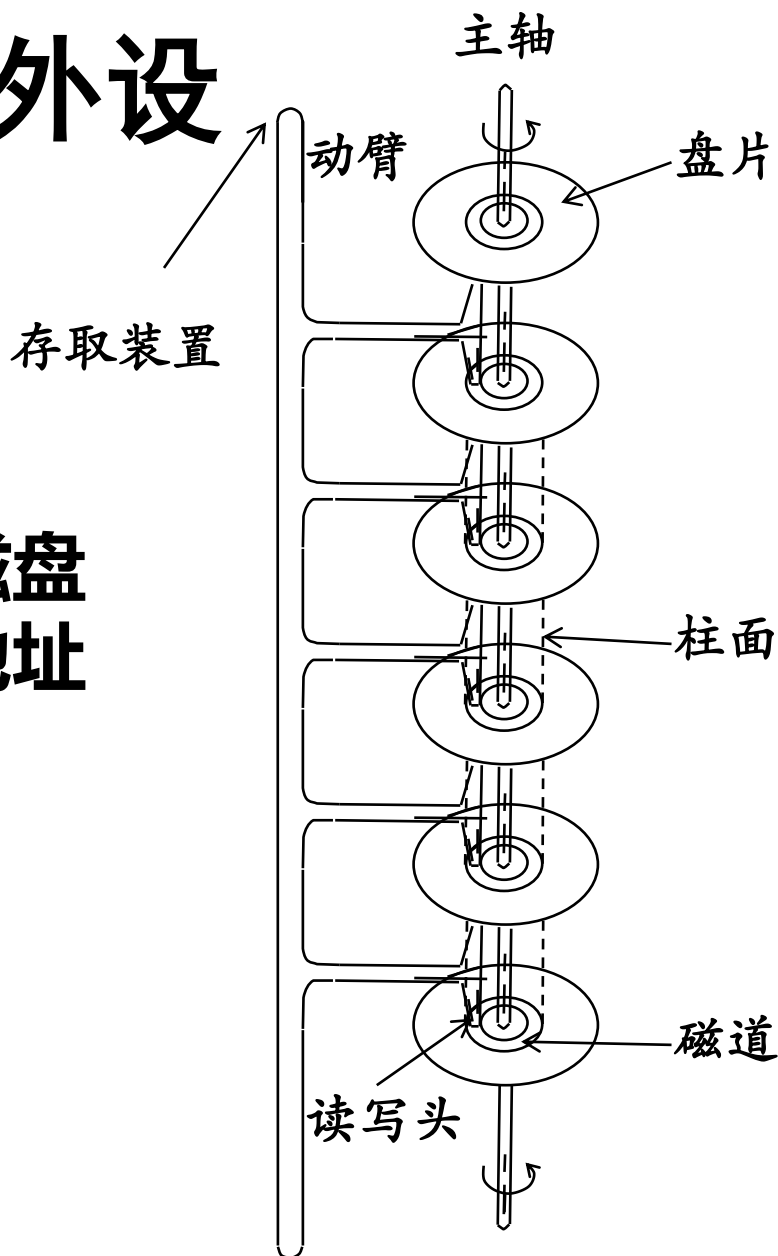
## ■ 磁盘

① 柱面号

② 磁头号

③ 扇区号

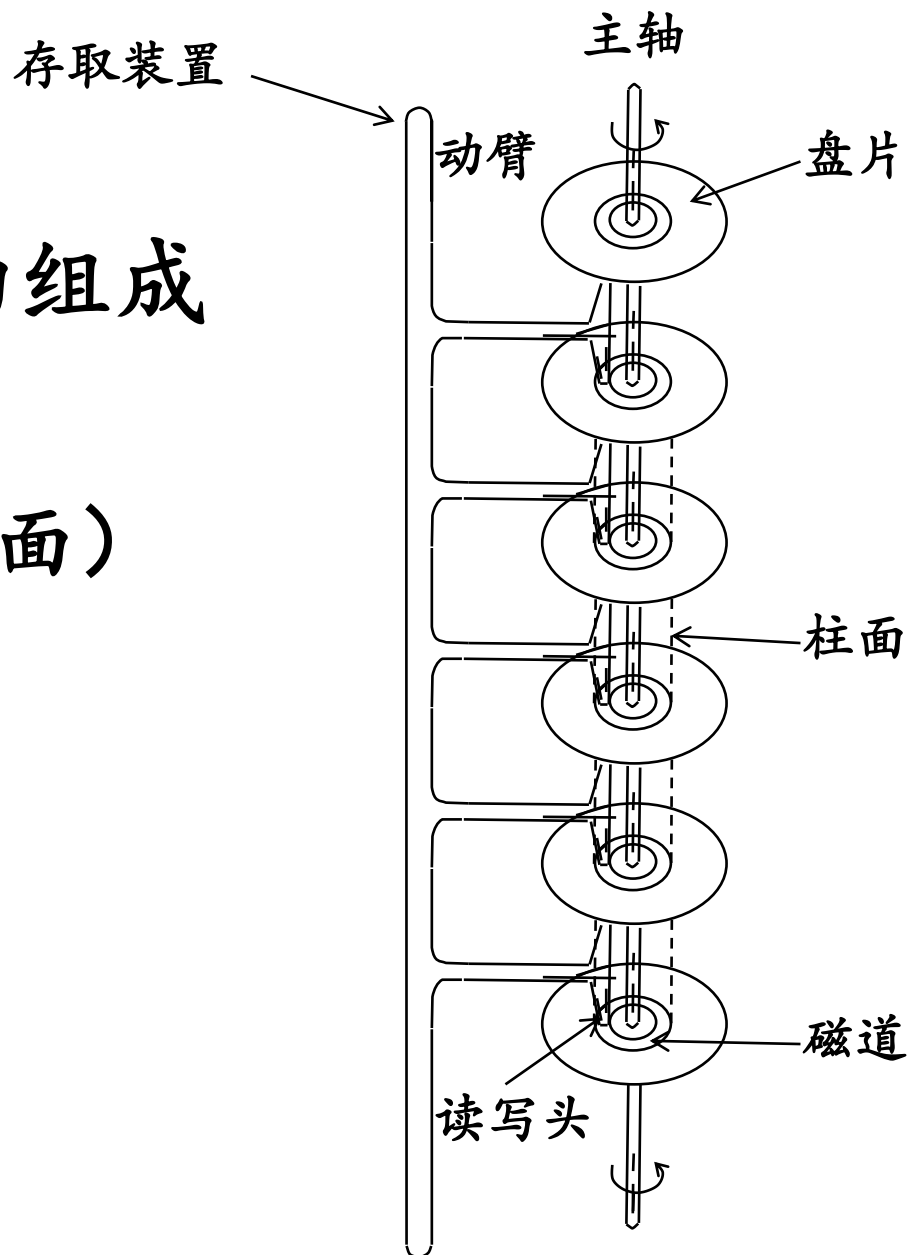
磁盘  
地址



# 磁盘

## ■ 磁盘访问时间的组成

- ① **寻道**时间：  
定位磁道（柱面）
- ② **延迟**时间：  
定位扇区
- ③ **传输**时间：  
扇区数据传输



# 磁盘

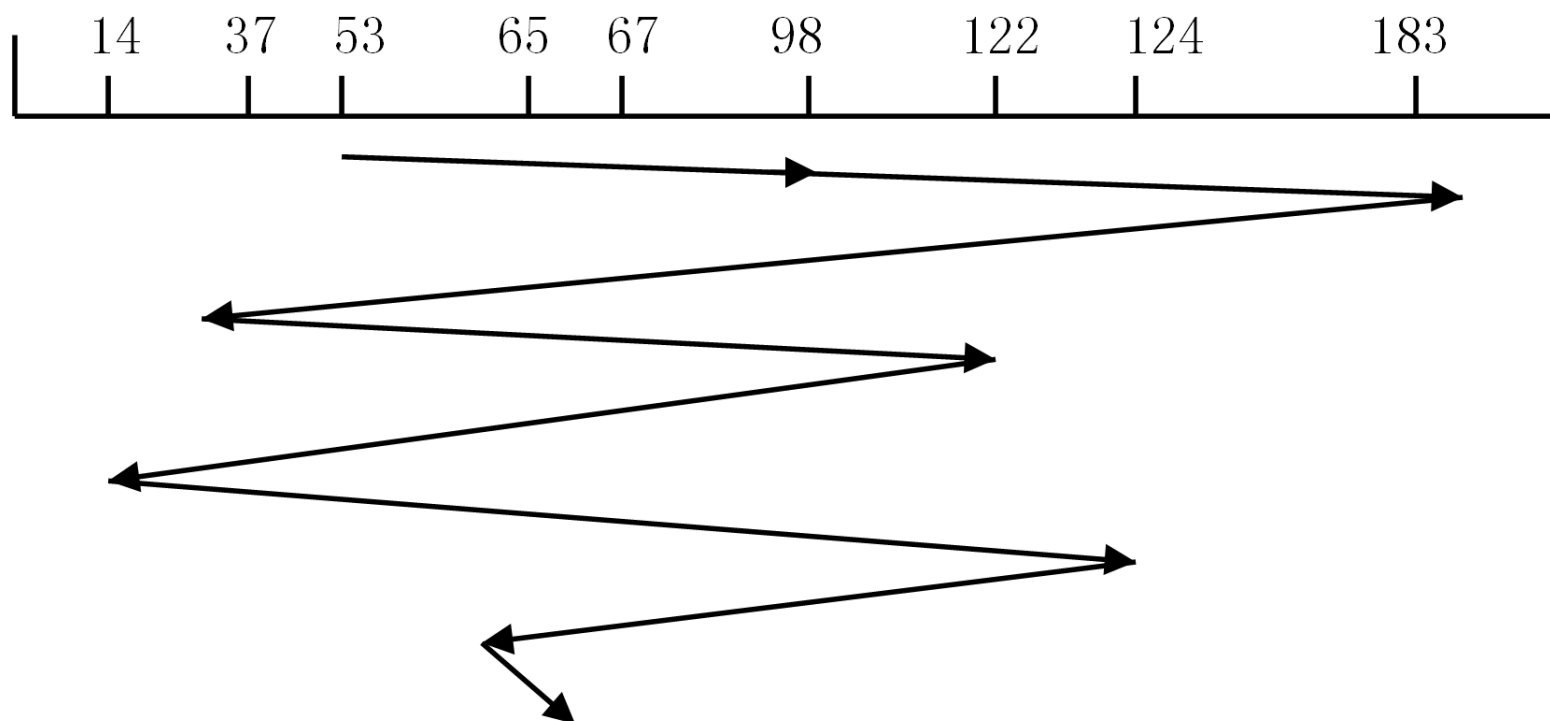
## ■ 减少寻道时间的方法

- 磁盘（磁头）调度算法：调度多进程对磁盘的访问请求的顺序，减少磁头移动



# 磁盘调度算法

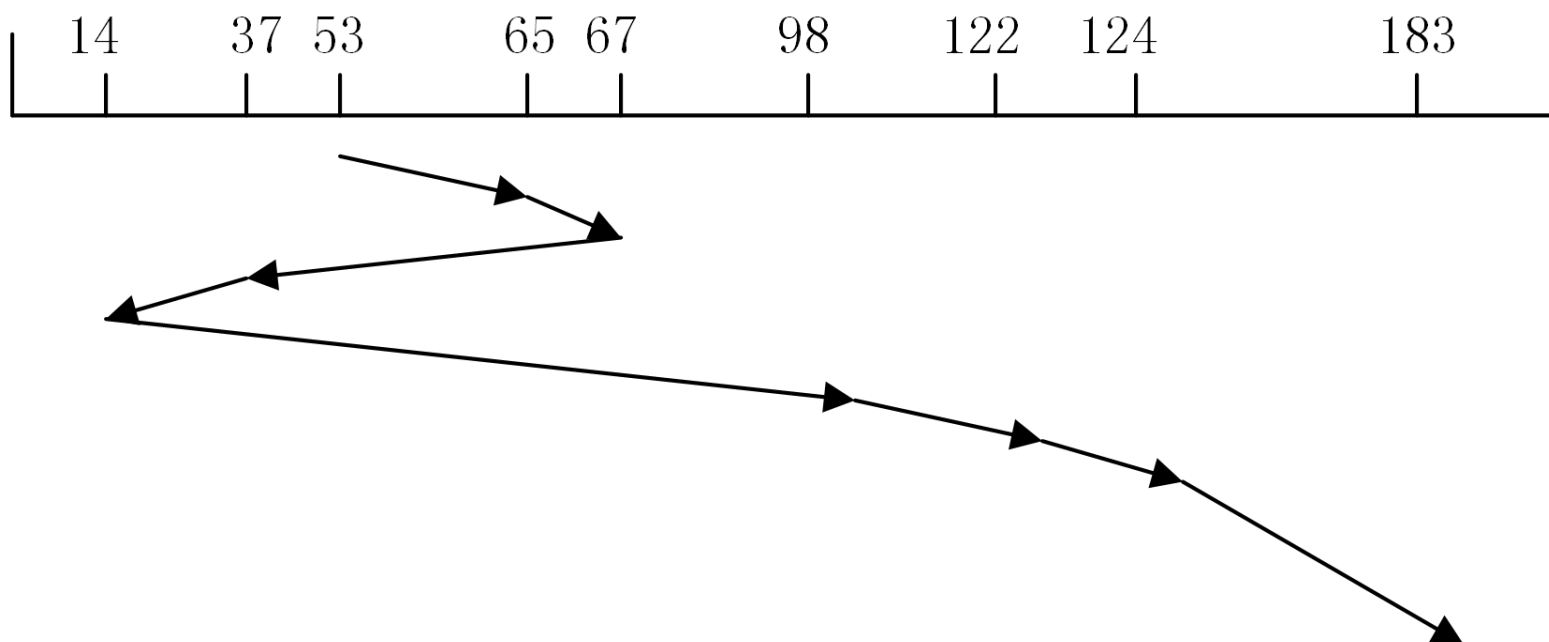
## ■ FCFS (先来先服务)



请求队列：98,183,37,122,14,124,65,67  
初始位置：53

# 磁盘调度算法

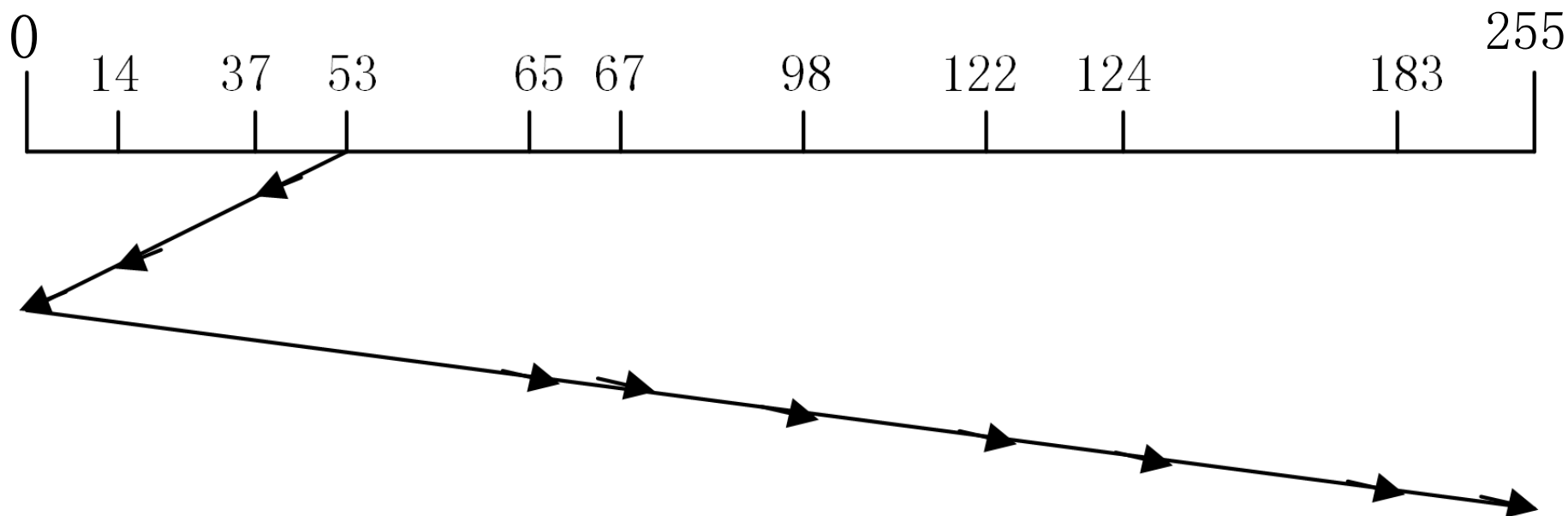
## ■ SSTF (最短寻道时间优先)



请求队列: 98,183,37,122,14,124,65,67  
初始位置: 53

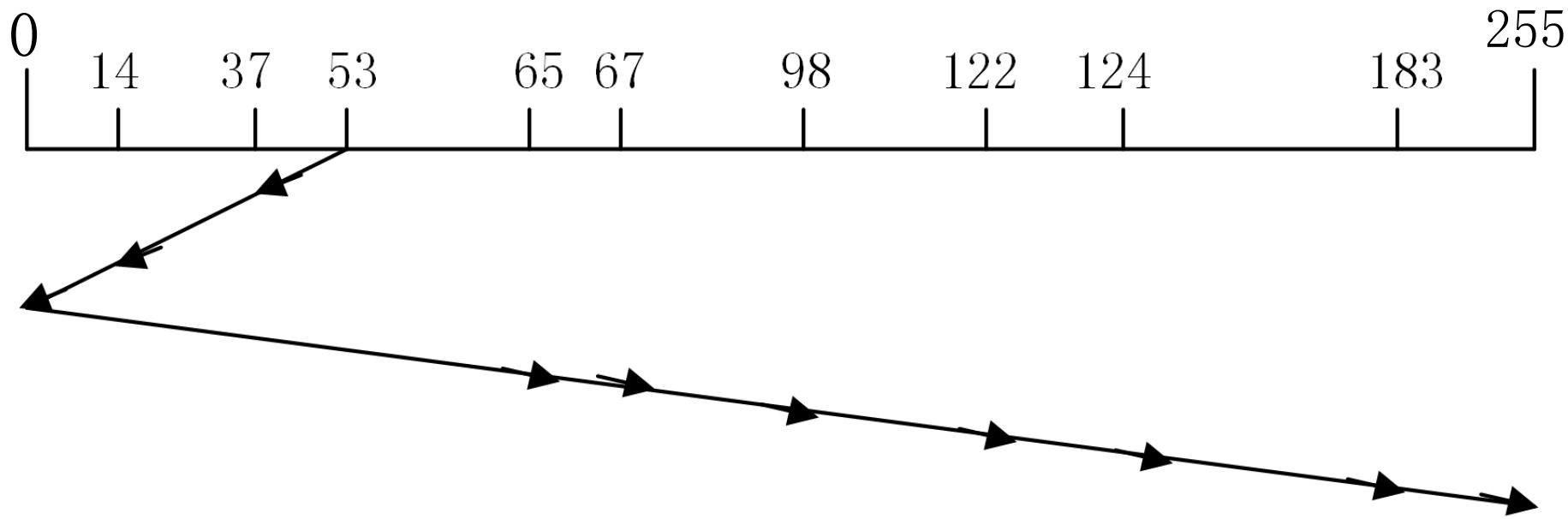
# 磁盘调度算法

## ■ SCAN (电梯算法)



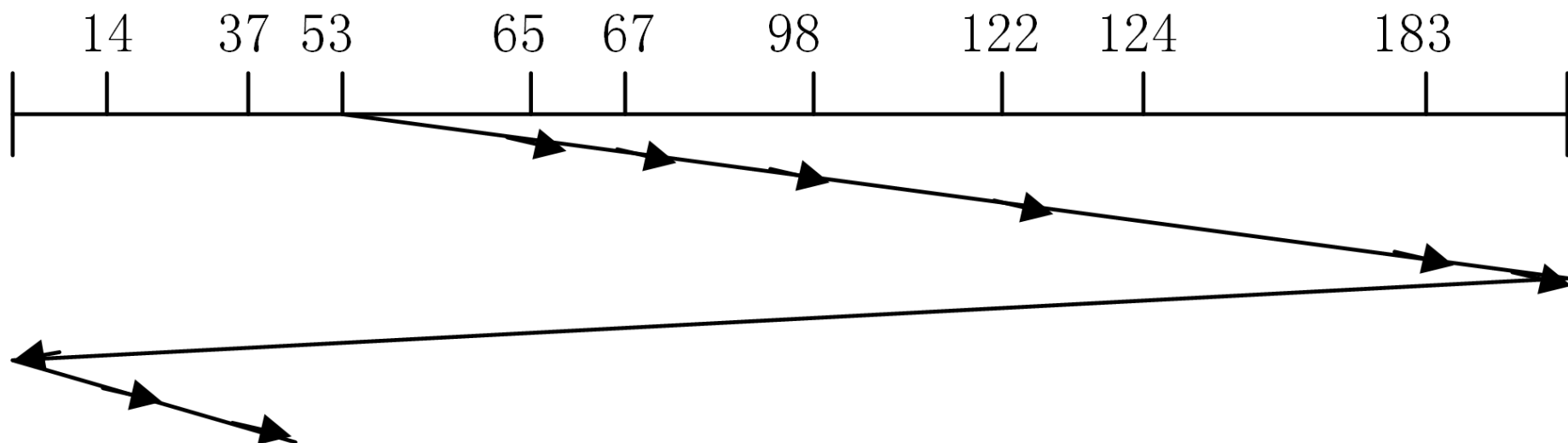
请求队列: 98,183,37,122,14,124,65,67  
初始位置: 53

在包含256个柱面的硬盘中，0号磁道的访问请求最坏情况下需要等待的移动距离是 [填空1] ； 98号磁道的访问请求最坏情况下需要等待的移动距离是 [填空2] 。



# 磁盘调度算法

## ■ C-SCAN (单向电梯算法)



请求队列: 98,183,37,122,14,124,65,67  
初始位置: 53

# 磁盘调度算法

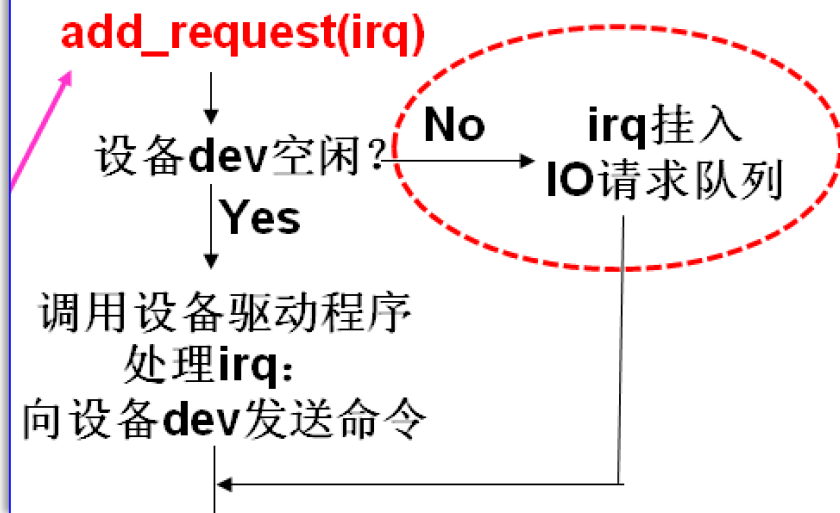
## ■ SCAN和C-SCAN:

- 1972年Teorey和Pinkerton的研究:  
这两种算法适合于磁盘负载较大的系统

# Linux 0.11的磁盘调度算法

```
static void add_request(struct blk_dev_struct * dev, struct request * req)
{
    struct request * tmp;

    req->next = NULL;
    cli();
    if (req->bh)
        req->bh->b_dirt = 0;
    if (!(tmp = dev->current_request)) {
        dev->current_request = req;
        sti();
        (dev->request_fn)();
        return;
    }
    for ( ; tmp->next ; tmp=tmp->next)
        if ((IN_ORDER(tmp, req) ||
             !IN_ORDER(tmp, tmp->next)) &&
            IN_ORDER(req, tmp->next))
            break;
    req->next=tmp->next;
    tmp->next=req;
    sti();
} ? end add_request ?
```



第一个请求是否小于第二个

假设请求按此顺序几乎同时到达：98,183,37,122,14,124,65,67

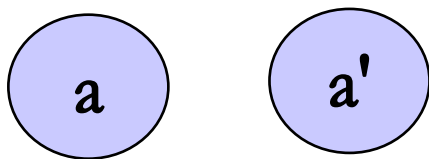
# 目录

- 一. 对字符设备的访问
- 二. I/O控制方式
- 三. 内存缓冲技术
- 四. 常见的存储外设
- 五. 盘阵与逻辑卷



## 五. 盘阵与逻辑卷

- 通过冗余提高可靠性：如建立镜像盘、增加奇偶校验等
- 通过并行性提高性能：如将原来在一个物理盘的连续数据分条分布到多盘，此过程称为条带化(striping)



a'为a的备份

abcdefghijklmnopqrst

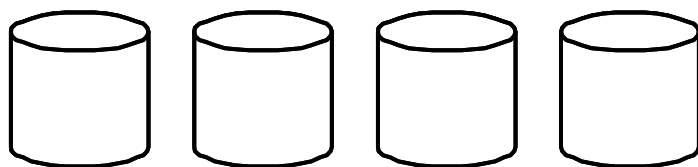
aeimq   bfjnr   cgkos   dhlpt

将数据分布到多个盘中

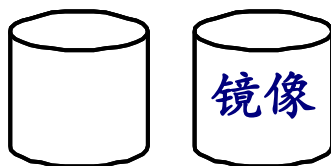
# 五. 盘阵与逻辑卷

## RAID级别

- RAID0: 指用到块级条带化的磁盘阵列
- RAID1: 磁盘镜像



(a) RAID 0: 无冗余条带化



(b) RAID1: 镜像磁盘

## 五. 盘阵与逻辑卷

- RAID 级别0、1、2、3、4、5、6可以由硬件提供，许多功能也可以由操作系统**逻辑卷**来提供。

### 逻辑卷

由驱动程序**软件**实现，如Linux的LVM。可以实现镜像卷、条带化卷、线性组合卷、快照卷等各种复杂逻辑卷。

# Linux 0.11

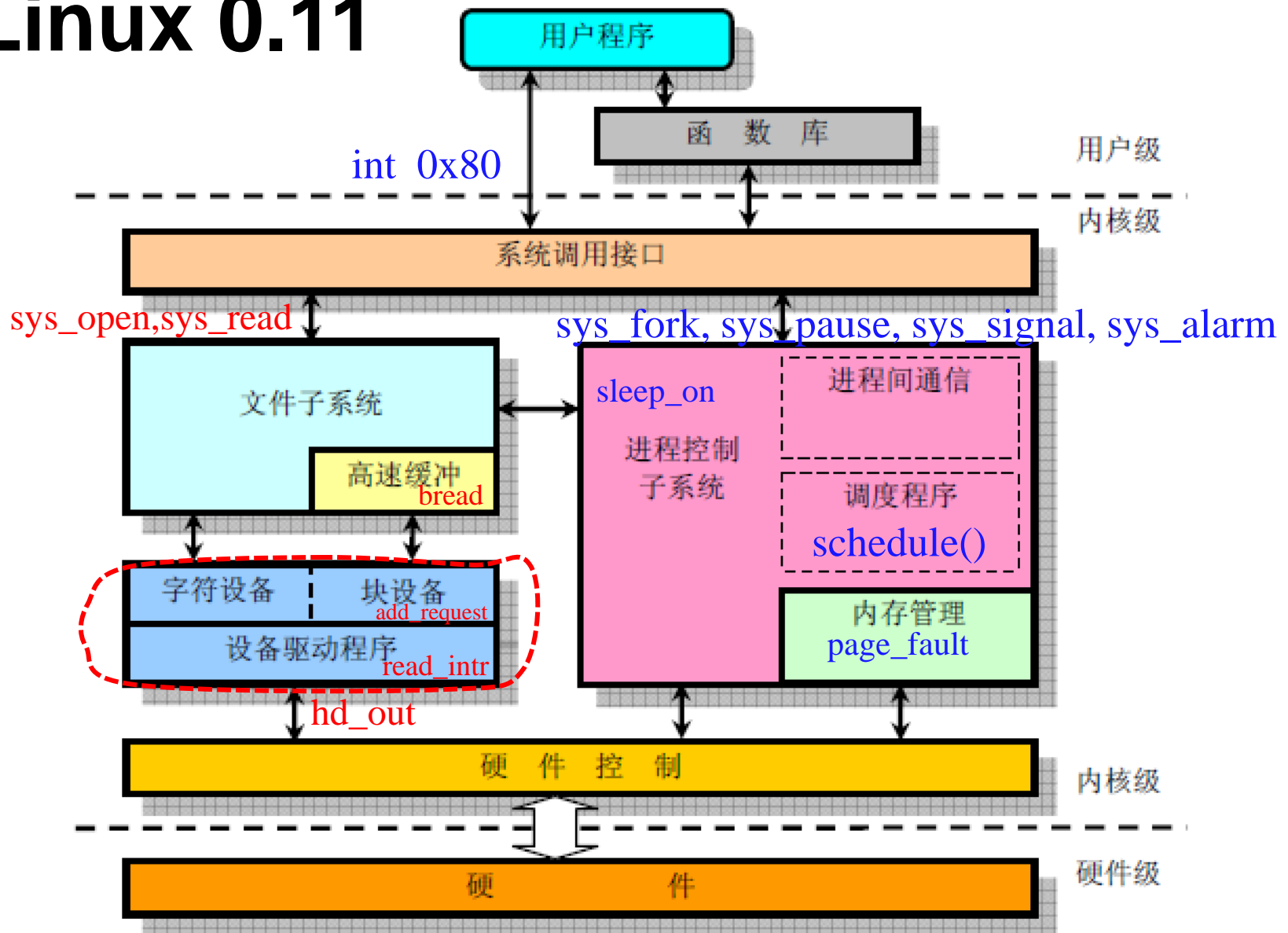


图 2-4 内核结构框图

# 小结

## 一. 对字符设备的访问

字符设备的数据队列

## 二. I/O控制方式

直接控制、中断控制、DMA控制

## 三. 内存缓冲技术

## 四. 常见的存储外设

磁盘、磁头调度算法

## 五. 盘阵与逻辑卷

# 小结

## 一. I/O硬件概念

设备控制器、I/O控制方式

## 二. 设备I/O子系统

设备的使用方法、输入输出层次结构、缓冲技术

## 三. 存储设备

常见的存储外设、盘阵与逻辑卷