

## 目录

第 1 章	绪论 .....	13
1.1	操作系统概念 .....	13
1.2	操作系统的历史.....	13
1.2.1	手工操作阶段.....	13
1.2.2	早期批处理 .....	13
1.2.3	多道程序系统.....	14
1.2.4	分时操作系统.....	15
1.2.5	实时操作系统.....	15
1.2.6	通用操作系统.....	15
1.2.7	操作系统的进一步发展 .....	15
1.3	操作系统的基本类型.....	15
1.3.1	批处理操作系统.....	15
1.3.2	分时系统 .....	16
1.3.3	实时系统 .....	16
1.3.4	通用操作系统.....	16
1.3.5	个人计算机上的操作系统.....	16
1.3.6	网络操作系统.....	16
1.3.7	分布式操作系统.....	16
1.4	操作系统功能 .....	16
1.4.1	处理机管理 .....	16
1.4.2	存储管理 .....	16
1.4.3	设备管理 .....	16
1.4.4	信息管理(文件系统管理).....	16
1.4.5	用户接口 .....	16

1.5 计算机硬件简介.....	16
1.5.1 计算机的基本硬件元素 .....	16
1.5.2 与操作系统相关的几种主要寄存器 .....	17
1.5.3 存储器的访问速度 .....	17
1.5.4 指令的执行与中断 .....	18
1.5.5 操作系统的启动.....	19
1.6 算法的描述 .....	19
1.7 研究操作系统的几种观点 .....	19
1.7.1 操作系统是计算机资源的管理者 .....	19
1.7.2 用户界面的观点.....	19
1.7.3 进程管理的观点.....	19
习题 .....	19
第2章 操作系统用户界面.....	20
2.1 简介.....	20
2.2 一般用户的输入输出界面 .....	20
2.2.1 作业的定义 .....	20
2.2.2 作业组织 .....	20
2.2.3 一般用户的输入输出方式.....	21
2.3 命令控制界面 .....	22
2.4Linux 与 Windows 的命令控制界面 .....	22
2.4.1Linux 的命令控制界面.....	22
2.4.2Windows 的命令控制界面 .....	23
2.5 系统调用 .....	24
2.6Linux 和 Windows 的系统调用.....	24
2.6.1Linux 系统调用 .....	24
2.6.2Windows 系统调用.....	25

本章小结.....	26
习题 .....	26
第3章 进程管理 .....	27
3.1 进程的概念 .....	27
3.1.1 程序的并发执行.....	27
3.1.2 进程的定义 .....	28
3.2 进程的描述 .....	28
3.2.1 进程控制块 PCB .....	28
3.2.2 进程上下文 .....	28
3.2.3 进程上下文切换.....	29
3.2.4 进程空间与大小.....	29
3.3 进程状态及其转换.....	29
3.3.1 进程状态 .....	29
3.3.2 进程状态转换.....	29
3.4 进程控制 .....	30
3.4.1 进程创建与撤销.....	30
3.4.2 进程的阻塞与唤醒.....	31
3.5 进程互斥 .....	31
3.5.1 资源共享所引起的制约 .....	31
3.5.2 互斥的加锁实现.....	32
3.5.3 信号量和 P,V 原语.....	32
3.5.4 用 P,V 原语实现进程互斥 .....	34
3.6 进程同步 .....	34
3.6.1 同步的概念 .....	34
3.6.2 私用信号量 .....	35
3.6.3 用 P, V 原语操作实现同步 .....	35

---

3.6.4 生产者-消费者问题.....	36
3.7 进程通信 .....	37
3.7.1 进程的通信方式.....	37
3.7.2 消息缓冲机制.....	38
3.7.3 邮箱通信 .....	38
3.7.4 进程通信的实例——和控制台的通信 .....	39
3.7.5 进程通信的实例——管道.....	40
3.8 死锁问题 .....	44
3.8.1 死锁的概念 .....	44
3.8.2 死锁的排除方法.....	45
3.9 线程的概念 .....	45
3.9.1 为什么要引入线程.....	45
3.9.2 线程的基本概念.....	45
3.9.3 线程与进程的区别 .....	46
3.9.4 线程的适用范围.....	46
3.10 线程分类与执行.....	47
3.10.1 线程的分类 .....	47
3.10.2 线程的执行特性.....	47
本章小结.....	48
习题 .....	48
第4章 处理机调度 .....	50
4.1 分级调度 .....	50
4.1.1 作业的状态及其转换 .....	50
4.1.2 调度的层次 .....	51
4.1.3 作业与进程的关系.....	51
4.2 作业调度 .....	51



4.2.1 作业调度功能.....	51
4.2.2 作业调度目标与性能衡量.....	53
4.3 进程调度 .....	53
4.3.1 进程调度的功能.....	53
4.3.2 进程调度的时机.....	53
4.3.3 进程调度性能评价 .....	53
4.4 调度算法 .....	53
4.5 算法评价 .....	54
4.5.1FCFS 方式的调度性能分析.....	54
4.5.2 轮转法调度性能评价 .....	55
4.5.3 线性优先级法的调度性能.....	55
4.6 实时系统调度方法.....	56
4.6.1 实时系统的特点.....	56
4.6.2 实时调度算法的分类 .....	56
4.6.3 时限调度算法与频率单调调度算法 .....	56
本章小结.....	58
习题 .....	58
第5章 存储管理 .....	60
5.1 存储管理的功能.....	60
5.1.1 虚拟存储器 .....	60
5.1.2 地址变换 .....	60
5.1.3 内外存数据传输的控制 .....	61
5.1.4 内存的分配与回收 .....	61
5.1.5 内存信息的共享与保护 .....	61
5.2 分区存储管理 .....	62
5.2.1 分区管理基本原理.....	62

---

5.2.2 分区的分配与回收 .....	64
5.2.3 有关分区管理其他问题的讨论 .....	66
5.3 覆盖与交换技术 .....	66
5.3.1 覆盖技术 .....	66
5.3.2 交换技术 .....	66
5.4 页式管理 .....	66
5.4.1 页式管理的基本原理 .....	66
5.4.2 静态页面管理 .....	67
5.4.3 动态页式管理 .....	69
5.4.4 请求页式管理中的置换算法 .....	70
5.4.5 存储保护 .....	72
5.4.6 页式管理的优缺点 .....	72
5.5 段式与段页式管理 .....	72
5.5.1 段式管理的基本思想 .....	72
5.5.2 段式管理的实现原理 .....	72
5.5.3 段式管理的优缺点 .....	75
5.5.4 段页式管理的基本思想 .....	75
5.5.5 段页式管理的实现原理 .....	75
5.6 局部性原理和抖动问题 .....	76
本章小结 .....	77
习题 .....	78
第6章 进程与存储管理示例 .....	79
6.1Linux 进程和存储管理简介 .....	79
6.2Linux 进程结构 .....	80
6.2.1 进程的概念 .....	80
6.2.2 进程的虚拟地址结构 .....	80

6.2.3 进程上下文 .....	80
6.2.4 进程的状态和状态转换 .....	81
6.2.5 小结 .....	82
6.3 进程控制 .....	82
6.3.1Linux 启动及进程树的形成 .....	82
6.3.2 进程控制 .....	83
6.4Linux 进程调度 .....	85
6.5 进程通信 .....	85
6.5.1Linux 的低级通信 .....	85
6.5.2 进程间通信 IPC.....	86
6.6 Linux 存储管理 .....	89
6.6.1 虚存空间和管理 .....	89
6.6.2 请求调页技术 .....	90
本章小结 .....	91
习题 .....	91
第7章 Windows 的进程与内存管理 .....	92
7.1Windows NT 的特点及相关的概念 .....	92
7.1.1Windows NT 体系结构的特点 .....	92
7.1.2Windows 的管理机制 .....	92
7.2Windows 进程和线程 .....	92
7.2.1Windows 的进程和线程的定义 .....	92
7.2.2 进程和线程的关联 .....	92
7.2.3Windows 进程的结构 .....	93
7.2.4Windows 线程的结构 .....	94
7.2.5Windows 进程和线程的创建 .....	94
7.3Windows 处理器调度机制 .....	95

---

7.3.1 调度优先级 .....	95
7.3.2 线程状态 .....	95
7.3.3 线程调度机制.....	96
7.4 Windows 的内存管理.....	97
7.4.1 内存管理器 .....	97
7.4.2 内存管理的机制.....	97
7.5 虚拟地址空间 .....	98
7.5.1 虚拟地址空间布局.....	98
7.5.2 虚拟地址转换.....	99
7.6 页面调度 .....	100
7.6.1 缺页处理 .....	100
7.6.2 工作集及页面调度策略 .....	101
7.6.3 页框号和物理内存管理 .....	101
本章小结.....	102
习题 .....	102
第8章 文件系统 .....	103
8.1 文件系统的概念.....	103
8.2 文件的逻辑结构与存取方法 .....	103
8.2.1 逻辑结构 .....	103
8.2.2 存取方法 .....	105
8.3 文件的物理结构与存储设备 .....	106
8.3.1 文件的物理结构.....	106
8.3.2 文件存储设备.....	107
8.4 文件存储空间管理.....	108
8.5 文件目录管理 .....	108
8.5.1 文件的组成 .....	108

8.5.2 文件目录 .....	108
8.5.3 便于共享的文件目录 .....	109
8.5.4 目录管理 .....	110
8.6 文件存取控制 .....	110
8.7 文件的使用 .....	112
8.8 文件系统的层次模型 .....	112
本章小结 .....	113
习题 .....	113
第9章 设备管理 .....	115
9.1 引言 .....	115
9.1.1 设备的类别 .....	115
9.1.2 设备管理的功能和任务 .....	116
9.2 数据传送控制方式 .....	116
9.2.1 程序直接控制方式 .....	116
9.2.2 中断方式 .....	116
9.2.3 DMA 方式 .....	117
9.2.4 通道控制方式 .....	118
9.3 中断技术 .....	119
9.3.1 中断的基本概念 .....	119
9.3.2 中断的分类与优先级 .....	119
9.3.3 软中断 .....	119
9.3.4 中断处理过程 .....	119
9.4 缓冲技术 .....	121
9.4.1 缓冲的引入 .....	121
9.4.2 缓冲的种类 .....	121
9.4.3 缓冲池的管理 .....	121

---

9.5 设备分配 .....	123
9.5.1 设备分配用数据结构 .....	123
9.5.2 设备分配的原则 .....	124
9.5.3 设备分配算法 .....	125
9.6 I/O 进程控制 .....	125
9.6.1 I/O 控制的引入 .....	125
9.6.2 I/O 控制的功能 .....	125
9.6.3 I/O 控制的实现 .....	126
9.7 设备驱动程序 .....	126
本章小结 .....	126
习题 .....	126
第 10 章 Linux 文件系统 .....	128
10.1 Linux 文件系统的特点与文件类别 .....	128
10.1.1 特点 .....	128
10.1.2 文件类型 .....	128
10.2 Linux 的虚拟文件系统 .....	128
10.2.1 虚拟文件系统 VFS 框架 .....	128
10.2.2 Linux 虚拟文件系统的数据结构 .....	128
10.2.3 VFS 的系统调用 .....	134
10.3 文件系统的注册和挂装 .....	135
10.3.1 文件系统注册 .....	135
10.3.2 已挂装文件系统描述符链表 .....	135
10.3.3 挂装根文件系统 .....	136
10.3.4 挂装一般文件系统 .....	136
10.3.5 卸载文件系统 .....	136
10.4 进程与文件系统的联系 .....	136

10.4.1 系统打开文件表.....	136
10.4.2 用户打开文件表.....	136
10.4.3 进程的当前目录和根目录.....	137
10.5ext2 文件系统 .....	137
10.5.1ext2 文件系统的存储结构 .....	137
10.5.2ext2 文件系统主要的磁盘数据结构 .....	138
10.5.3ext2 文件系统的内存数据结构.....	140
10.5.4 数据块寻址 .....	140
10.6 块设备驱动 .....	141
10.6.1 设备配置 .....	141
10.6.2 设备驱动程序的接口 .....	141
10.7 字符设备驱动 .....	142
习题 .....	142
第 11 章 Windows 的设备管理和文件系统.....	144
11.1Windows I/O 系统的结构.....	144
11.1.1 设计目标 .....	144
11.1.2 设备管理服务.....	144
11.2 设备驱动程序和 I/O 处理 .....	145
11.2.1 设备驱动类型和结构 .....	145
11.2.2Windows 的 I/O 处理.....	145
11.3Windows 的文件系统.....	146
11.3.1Windows 磁盘管理.....	146
11.3.2 Windows 文件系统格式 .....	147
11.3.3Windows 文件系统驱动 .....	147
11.4NTFS 文件系统 .....	147
11.4.1NTFS 的特点 .....	147

---

11.4.2NTFS 的磁盘结构.....	147
11.4.3NTFS 的文件系统恢复.....	148
本章小结.....	148
习题 .....	148
参考文献.....	149



# 第1章 绪论

## 1.1 操作系统概念

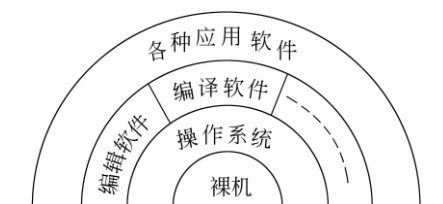


图 1.1 操作系统与硬件软件的关系

## 1.2 操作系统的历史

### 1.2.1 手工操作阶段

### 1.2.2 早期批处理

#### 1. 联机批处理

#### 2. 脱机批处理

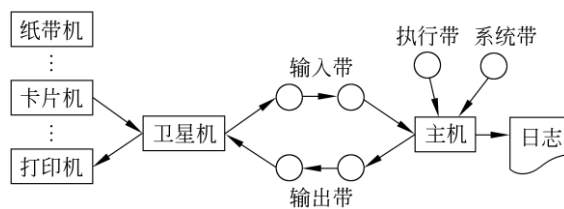


图 1.2 早期脱机批处理模型

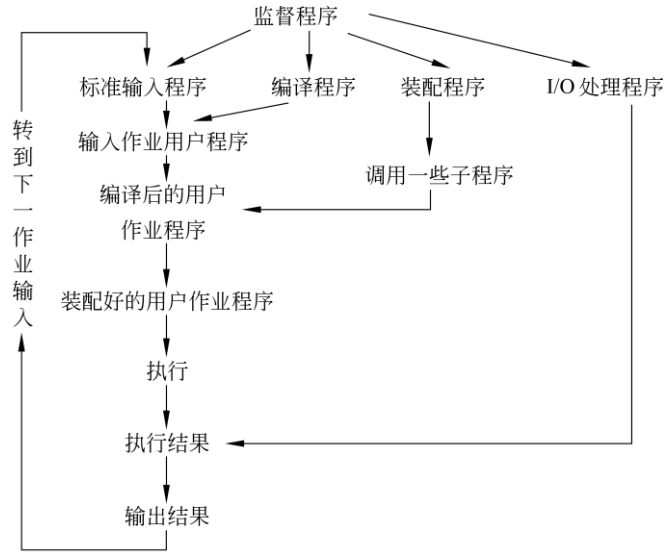


图 1.3 监督程序管理下的解题过程

### 1.2.3 多道程序系统

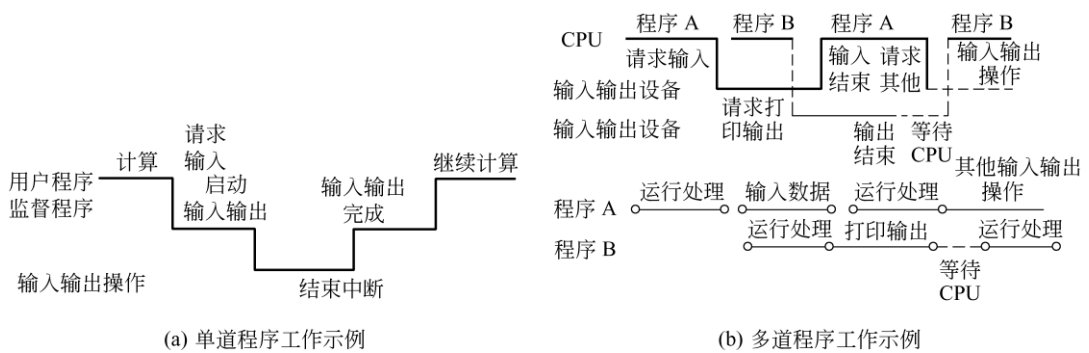


图 1.4 程序工作示例

## 1.2.4 分时操作系统

## 1.2.5 实时操作系统

## 1.2.6 通用操作系统

## 1.2.7 操作系统的进一步发展

# 1.3 操作系统的基本类型

## 1.3.1 批处理操作系统

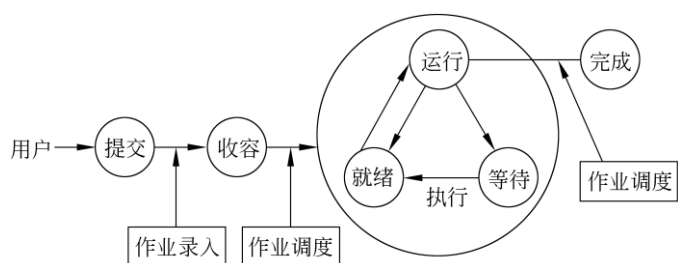


图 1.5 批处理系统中作业处理及状态

### 1.3.2 分时系统

### 1.3.3 实时系统

### 1.3.4 通用操作系统

### 1.3.5 个人计算机上的操作系统

### 1.3.6 网络操作系统

### 1.3.7 分布式操作系统

## 1.4 操作系统功能

### 1.4.1 处理机管理

### 1.4.2 存储管理

### 1.4.3 设备管理

### 1.4.4 信息管理(文件系统管理)

### 1.4.5 用户接口

## 1.5 计算机硬件简介

### 1.5.1 计算机的基本硬件元素

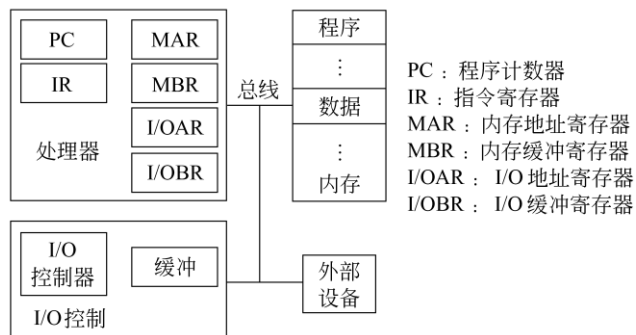


图 1.6 计算机的基本硬件元素

## 1.5.2 与操作系统相关的几种主要寄存器

1. 数据寄存器
2. 地址寄存器
3. 条件码寄存器
4. 程序计数器 PC
5. 指令寄存器 IR
6. 程序状态字 PSW
7. 中断现场保护寄存器
8. 过程调用用堆栈

## 1.5.3 存储器的访问速度

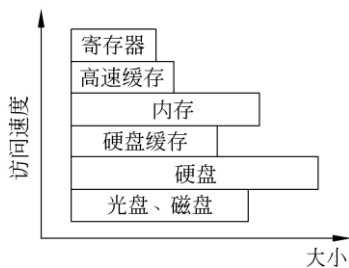


图 1.7 存储介质的访问速度

### 1.5.4 指令的执行与中断

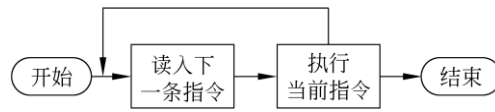


图 1.8 指令的执行周期

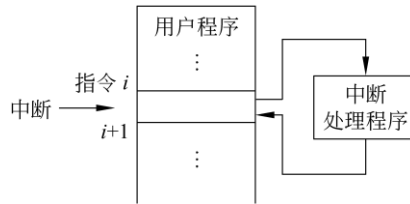


图 1.9 中断执行过程

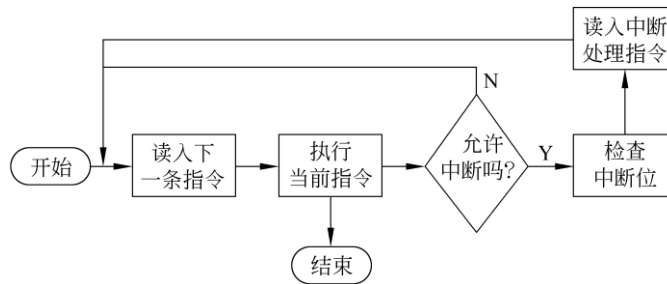


图 1.10 中断处理时的指令执行周期

### 1.5.5 操作系统的启动

## 1.6 算法的描述

## 1.7 研究操作系统的几种观点

### 1.7.1 操作系统是计算机资源的管理者

### 1.7.2 用户界面的观点

### 1.7.3 进程管理的观点

## 习题

1.1 什么是操作系统的基本功能？

1.2 什么是批处理、分时和实时系统？各有什么特征？

1.3 多道程序设计(multiprogramming)和多重处理(multiprocessing)有何区别？

1.4 讨论操作系统可以从哪些角度出发，如何把它们统一起来？

1.5 写出 1.6 节中巡回置换算法的执行结果。

1.6 设计计算机操作系统时与哪些硬件器件有关？

# 第 2 章 操作系统用户界面

## 2.1 简介

## 2.2 一般用户的输入输出界面

### 2.2.1 作业的定义

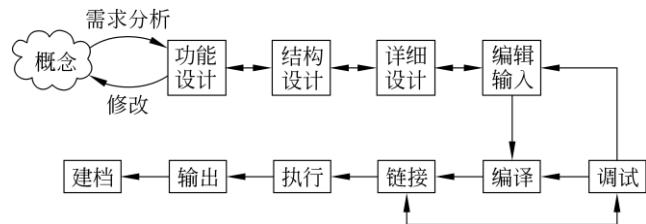


图 2.1 一般编程过程

### 2.2.2 作业组织

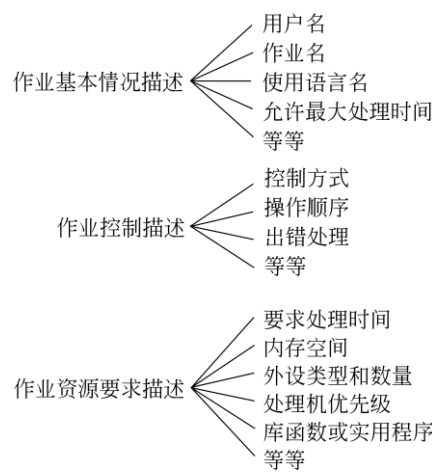


图 2.2 作业说明书的主要内容



## 2.2.3 一般用户的输入输出方式

1. 联机输入输出方式
2. 脱机输入输出方式
3. 直接耦合方式

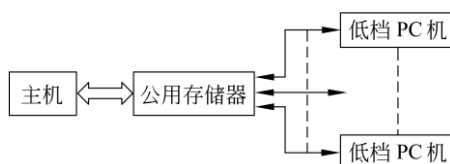


图 2.3 直接耦合方式

## 4. SPOOLING 系统

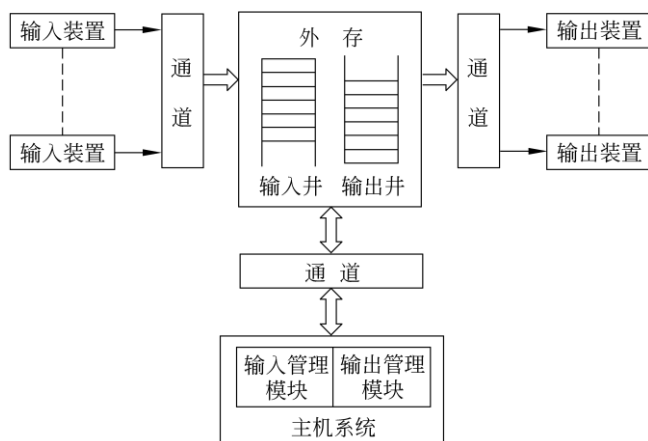


图 2.4 SPOOLING 系统

## 5. 网络联机方式

## 2.3 命令控制界面

## 2.4 Linux 与 Windows 的命令控制界面

### 2.4.1 Linux 的命令控制界面

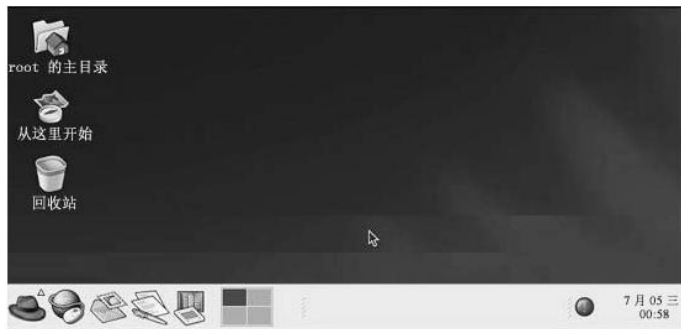


图 2.5 Redhat Linux 9.0 的窗口界面示例

```
mkdir backup
for filein 'ls'
do
    cp $ file backup/ $ file
    if [ $ > -ne 0 ] then
        echo "copying $ file error"
    fi
done
```

## 2.4.2 Windows 的命令控制界面

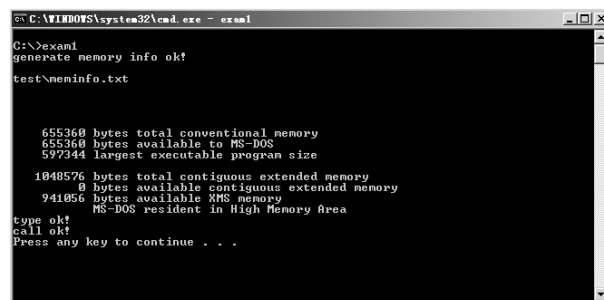
exam2. bat:

```
@ echo off
mem > %1/meminfo. txt
echo generate memoryinfo ok!
```

exam3. bat:

```
@ echo off
type %1\* . txt
echo type ok!
```

```
@ echo off
mdir test
call exam2. bat test
call exam3. bat test
echo call ok!
pause
```



```
C:\WINDOWS\system32\cmd.exe - exam1
C:\>exam1
generate memory info ok!
test\meminfo.txt
655360 bytes total conventional memory
655360 bytes available to MS-DOS
597344 largest executable program size
1048576 bytes total contiguous extended memory
0 bytes available contiguous extended memory
741856 bytes available XMS memory
MS-DOS resident in High Memory Area
type ok!
call ok!
Press any key to continue . . .
```

图 2.6 相互调用批处理示例

2.5 系统调用

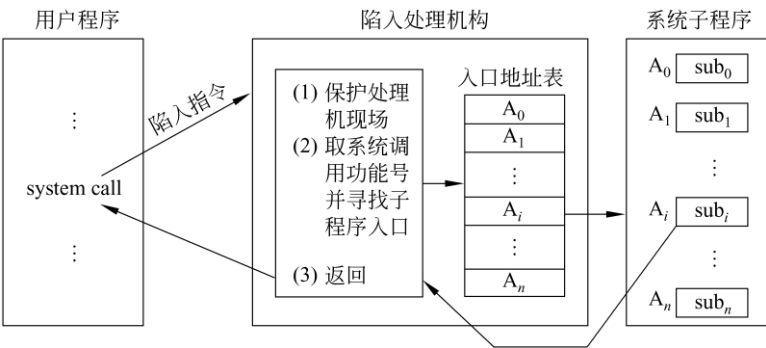


图 2.7 系统调用的处理过程

2.6Linux 和 Windows 的系统调用

2.6.1Linux 系统调用

```
#include <fcntl.h>
#include <sys/stat.h>
#define SIZE 1
void filecopy(char * infile, char * Outfile)
{
    char Buffer[SIZE];
    int in_fh, Out_fh, Count;
    if((In_fh = open(Infile, O_RDONLY)) == -1) /* 以只读模式打开输入文件 */
        printf("Opening infile");
    if((Out_fh = open(Outfile, (O_WRONLY|O_CREAT|O_TRUNC),
(S_IRUSR|S_IWUSR)) == -1) /* 以读写模式新建一个文件 */
        printf("Opening Outfile");
    while((Count = read(In_fh, Buffer, sizeof(Buffer))) > 0) /* 循环地向缓冲区读入输入文件内容 */
        if(write(Out_fh, Buffer, Count) != Count)
            /* 将缓冲区读入的内容写到输出文件中去 */
            printf("Writing date");
    if(count == -1)
        printf("Reading date");
    close(In_fh); /* 关闭输入文件 */
    close(Out_fh); /* 关闭输出文件 */
}
```

## 2.6.2 Windows 系统调用

```
#include < windows.h >
//入口函数
int WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, intiCmdshow)
{
    HANDLE hFile;
    LPTSTR lpBuffer = "Hello World !";
    //创建文件
    hFile = CreateFile("C:\\File.txt", GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    CloseHandle(hFile);
    TCHAR szBuf[ 128 ];
    DWORD dwRead;
    DWORD dwWritten;
    //打开文件
    hFile = CreateFile("C:\\File.txt", GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    //向文件中写入一个字符串
    WriteFile( hFile, lpBuffer, strlen( lpBuffer ) + 1, &dwRead, NULL);
    SetFilePointer( hFile, 0, NULL, FILE_BEGIN);
    //从文件中读出一个字符串并将它显示在对话框中
    if( ReadFile( hFile, szBuf, strlen( lpBuffer ) + 1, &dwWritten, NULL) );
    {
        messagebox( NULL, szBuf, "EXAM", MB_OK);
    }
    CloseHandle(hFile);
}
```

## 本章小结

### 习题

2.1 什么是作业、作业步？

2.2 作业由哪几部分组成？各有什么功能？

2.3 作业的输入方式有哪几种？各有何特点？

2.4 试述 SPOOLING 系统的工作原理。

2.5 操作系统为用户提供哪些接口？它们的区别是什么？

2.6 作业控制方式有哪几种？调查你周围的计算机的作业控制方式。

2.7 什么是系统调用？系统调用与一般用户程序有什么区别？与库函数和实用程序又有什么区别？

2.8 简述系统调用的实现过程。

2.9 为什么说分时系统没有作业的概念？

2.10 Linux 操作系统为用户提供哪些接口？试举例说明。

2.11 在你周围装有 Linux 系统的计算机上，查看有关 Shell 的基本命令，并编写一个简单的 Shell 程序，完成一个已有数据文件的复制和打印。

2.12 用 Linux 文件读写的相关系统调用，编写一个 copy 程序。

2.13 用 Windows 的 DLL 接口编写 copy 程序。

## 第3章 进程管理

### 3.1 进程的概念

#### 3.1.1 程序的并发执行

1. 程序(program)
2. 程序的顺序执行
3. 多道程序系统中程序执行环境的变化
4. 程序的并发(concurrent)执行

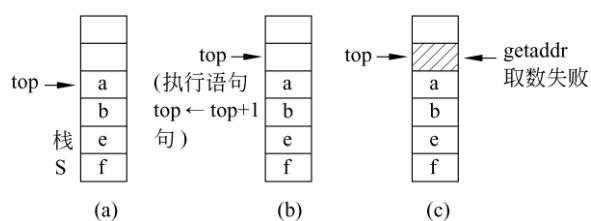


图 3.1 堆栈的取数和存数过程

3.1.2 进程的定义

3.2 进程的描述

3.2.1 进程控制块 PCB

- 1. 描述信息
- 2. 控制信息
- 3. 资源管理信息
- 4. CPU 现场保护结构

3.2.2 进程上下文

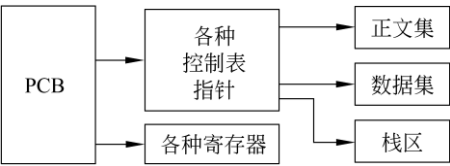


图 3.2 进程上下文结构

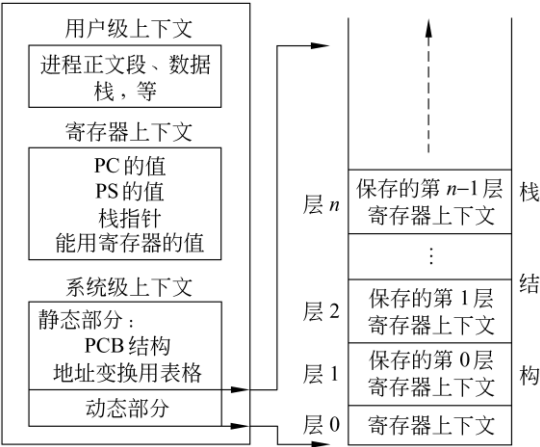


图 3.3UNIX System V 进程上下文组成



### 3.2.3 进程上下文切换

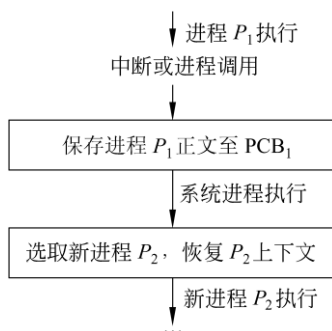


图 3.4 进程上下文的切换过程

### 3.2.4 进程空间与大小

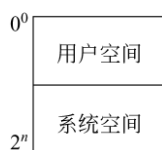


图 3.5 进程空间示例

## 3.3 进程状态及其转换

### 3.3.1 进程状态

### 3.3.2 进程状态转换

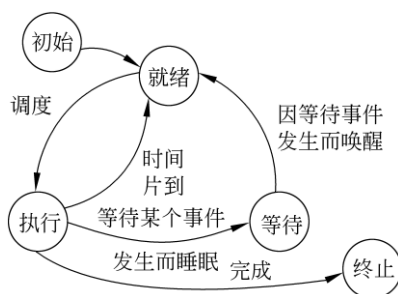


图 3.6 进程状态转换

## 3.4 进程控制

### 3.4.1 进程创建与撤销

#### 1. 进程创建

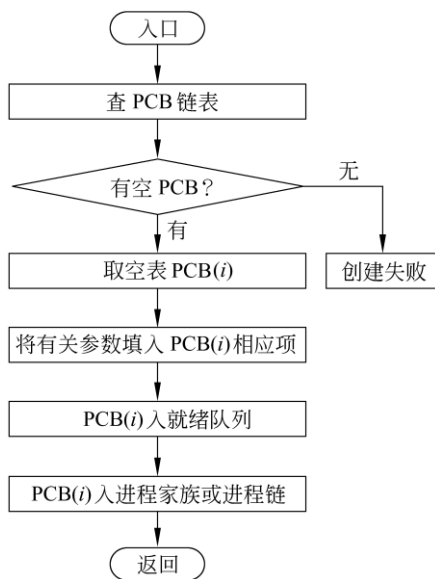


图 3.7 创建原语流程图

#### 2. 进程撤销

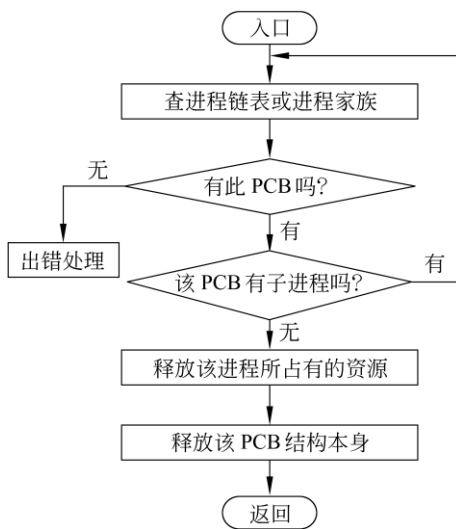


图 3.8 撤销原语流程图

3.4.2 进程的阻塞与唤醒

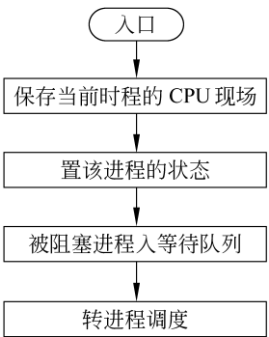


图 3.9 阻塞原语图

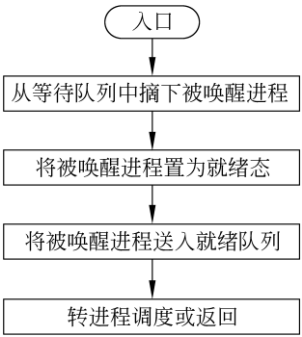


图 3.10 唤醒原语

3.5 进程互斥

3.5.1 资源共享所引起的制约

1. 临界区

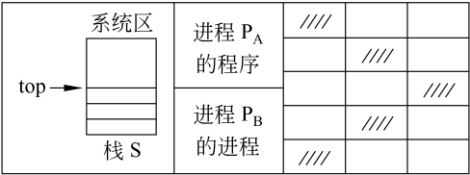


图 3.11 多进程共享内存栈区示例

2. 间接制约
3. 什么是互斥

### 3.5.2 互斥的加锁实现

### 3.5.3 信号量和 P,V 原语

1. 信号量(semaphore)
2. P,V 原语

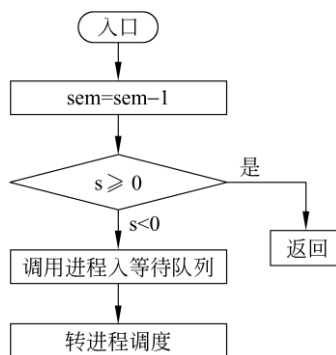


图 3.12P 原语操作功能

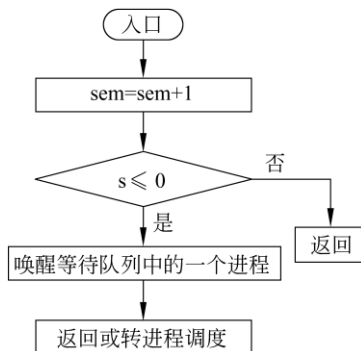


图 3.13V 原语操作功能

```

P(sem):
    begin
        封锁中断;
        lock(lockbit)
        val[sem] = val[sem] - 1
        if val[sem] < 0
            保护当前进程 CPU 现场
            当前进程状态置为"等待"
            将当前进程插入信号 sem 等待队列
            转进程调度
        fi
        unlock(lockbit);开放中断
    end
V(sem):
    begin
        封锁中断;
        lock(lockbit)
        val[sem] = val[sem] + 1
        if val[sem] ≤ 0
            localk

            从 sem 等待队列中选取一等待进程,将其指针置入 k 中
            将 k 插入就绪队列
            进程状态置"就绪"
        fi
        unlock(lockbit);开放中断
    end
end
    
```

### 3.5.4 用 P,V 原语实现进程互斥

## 3.6 进程同步

### 3.6.1 同步的概念

```
Pc  :  
    :  
    A: local Buf  
        Repeat  
            Buf ← Buf  
        Until Buf = 空  
        计算  
        得到计算结果  
        Buf ← 计算结果  
        Goto A  
  
Pp  :  
    :  
    B: local Pri  
        Repeat  
            Pri ← Buf  
        Until Pri ≠ 空  
        打印 Buf 中的数据  
        清除 Buf 中的数据  
        Goto B
```

```

Pc :
  A: wait( Bufempty)
    计算
    Buf ← 计算结果
    Bufempty ← false
    signal( Buffull)
    Goto A

Pp :
  B: wait( Buffull)
    打印 Buf 中的数据
    清除 Buf 中的数据
    Buffull ← false
    signal( Bufempty)
    Goto B
    
```

### 3.6.2 私用信号量

### 3.6.3 用 P , V 原语操作实现同步

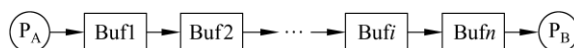


图 3.14 缓冲区队列

```

PA: deposit(data):
    begin local x
        P( Bufempty );
        按 FIFO 方式选择一个空缓冲区 Buf(x);
        Buf(x) ← dat
        Buf(x) 置满标记
        V( Buffull)
    end

```

```

PB: remove(data):
    Begin local x
        P ( Buffull );
        按 FIFO 方式选择一个装满数据的缓冲区 Buf(x)
        data ← Buf(x)
        Buf(x)置空标记
        V ( Bufempty)
    End

```

### 3.6.4 生产者-消费者问题

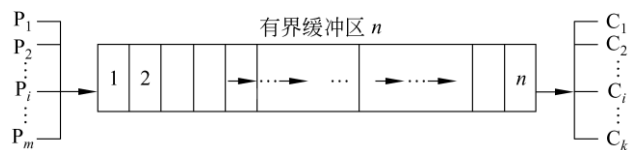


图 3.15 生产者-消费者问题



```

deposit( data ) :
begin
    P( avail )
    P( mutex )
    送数据入缓冲区某单元
    V( full )
    V( mutex )
end
remove( data ) :
begin
    P( full )
    P( mutex )
    取缓冲区中某单元数据
    V( avail )
    V( mutex )
End
    
```

## 3.7 进程通信

### 3.7.1 进程的通信方式

发送进程	接收进程	操作	数据
------	------	----	----

图 3.16 消息的组成

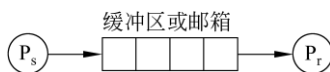


图 3.17 缓冲区或邮箱通信结构

```

send(m):
    begin
        向系统申请一个消息缓冲区
        P(mutex)
        将发送区消息 m 送入新申请的消息缓冲区
        把消息缓冲区挂入接收进程的消息队列
        V(mutex)
        V(SM)
    end
receive(n):
    begin
        P(SM)
        P(mutex)
        摘下消息队列中的消息 n
        将消息 n 从缓冲区复制到接收区
        释放缓冲区
        V(mutex)
    end
end

```

### 3.7.2 消息缓冲机制

### 3.7.3 邮箱通信

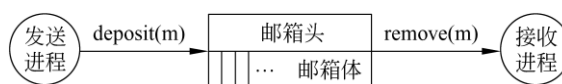


图 3.18 邮箱通信结构

```

deposit(m):
    begin local x
        P(fromnum)
        选择空格 x
        将消息 m 放入空格 x 中
        置格 x 的标志为满
        V(mesnum)
    end
remove(m):
    begin local x
        P(mesnum)
        选择满格 x
        把满格 x 中的消息取出放 m 中
        置格 x 标志为空
        V(fromnum)
    end
    
```

### 3.7.4 进程通信的实例——和控制台的通信

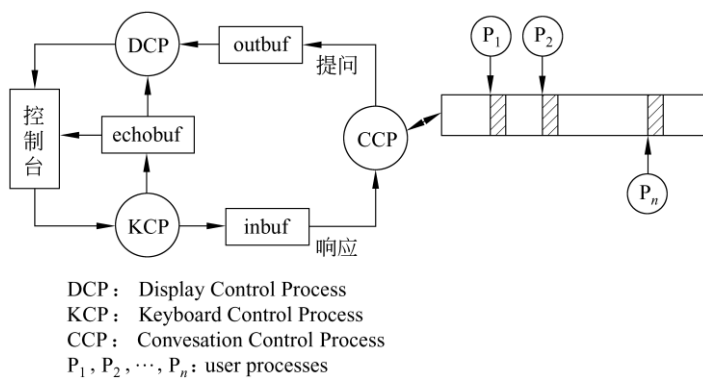


图 3.19 和控制台通信示例

- 1. KCP 和 DCP 的动作
- 2. CCP 和 KCP 及 DCP 的接口
- 3. CCP 与用户进程的接口

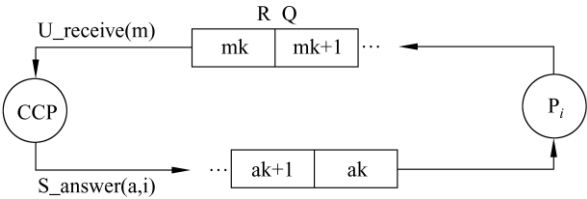


图 3.20CCP 和用户进程接口

- 4. CCP 的动作

3.7.5 进程通信的实例——管道

- 1. 管道 pipe

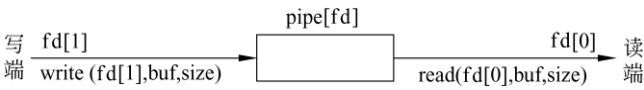


图 3.21 管道通信

## 2. 示例

```
#include <stdio.h>
main()
{
    int x,fd[2];
    char buf[30],s[30];
    pipe(fd);                                /* 创建管道 */
    while( ( x=fork()) == -1);                /* 创建子进程失败时,循环 */
    if(x == 0)
    {
        sprintf(buf,"This is an example\n");
        write( fd[1],buf,30);                /* 把 buf 中字符写入管道 */
        exit(0);
    }
    else                                      /* 父进程返回 */
    {
        wait(0);
        read( fd[0],s,30);                  /* 父进程读管道中字符 */
        printf("%s",s);
    }
}
```

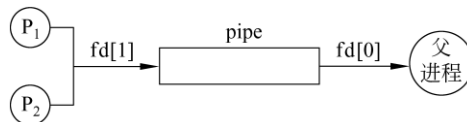


图 3.22 父进程和子进程 P1, P2 通信例子

```

#include <stdio.h>
main()
{
    int i,r,P1,P2,fd[2];
    char buf[50],s[50];
    pipe(fd);                                /* 父进程建立管道 */
    while((P1=fork()) == -1);                /* 创建子进程 P1,失败时循环 */
    if(p1 == 0)                              /* 由子进程 P1 返回,执行子进程 P1 */
    {
        lockf(fd[1],1,0);                   /* 加锁锁定写入端 */
        sprintf(buf,"child process P1 is sending messages! \n");
        printf("child process P1! \n");
        write(fd[1],buf,50);                /* 把 buf 中的 50 个字符写入管道 */
        sleep(5);                           /* 睡眠 5 秒,让父进程读 */
        lockf(fd[1],0,0);                   /* 释放管道写入端 */
        exit(0);                            /* 关闭 P1 */
    }
    else                                     /* 从父进程返回,执行父进程 */
    {
        while((P2=fork()) == -1);           /* 创建子进程 P2,失败时循环 */
        if(P2 == 0)                         /* 从子进程 P2 返回,执行 P2 */
        {
            lockf(fd[1],1,0);               /* 锁定写入端 */
            sprintf(buf,"child process P2 is sending messages\n");
            printf("child process P2 ! \n");
            write(fd[1],buf,50);             /* 把 buf 中字符写入管道 */
            sleep(5);                       /* 睡眠等待 */
            lockf(fd[1],0,0);               /* 释放管道写入端 */
            exit(0);                        /* 关闭 P2 */
        }
        wait(0);
        if(r=read(fd[0],s,50) == -1)
            printf("can't read pipe\n");
        else printf("%s\n",s);
    }
}

```

```

wait(0);
if(r = read(fd[0],s,50) == -1)
    printf("can't read pipe\n");
else printf("%s\n",s);
exit(0);
}
}
    
```

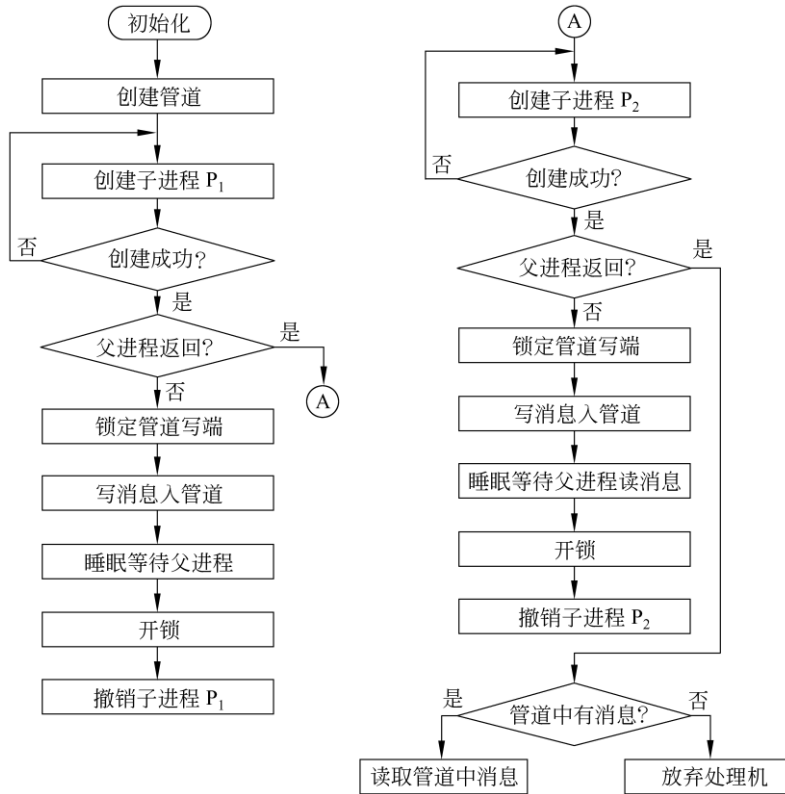


图 3.23 例 2 程序流程图

## 3.8 死锁问题

### 3.8.1 死锁的概念

#### 1. 死锁的定义

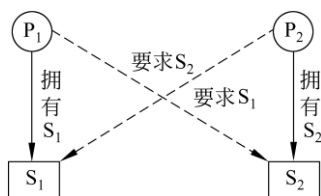


图 3.24 死锁的概念



- 2. 死锁的起因
- 3. 产生死锁的必要条件

3.8.2 死锁的排除方法

- 1. 死锁预防
- 2. 死锁避免
- 3. 死锁的检测和恢复

3.9 线程的概念

3.9.1 为什么要引入线程

3.9.2 线程的基本概念

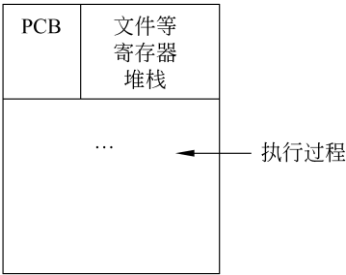


图 3.25 等效于单线程的进程执行示意图

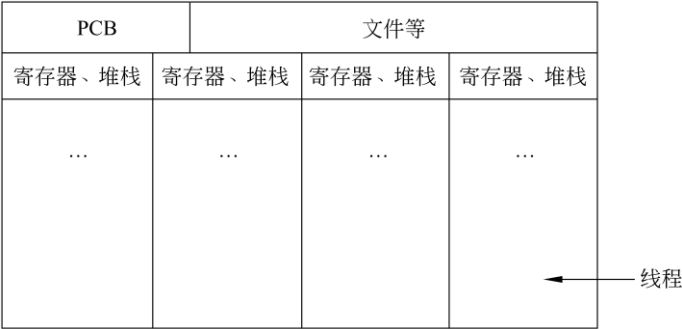


图 3.26 多线程情况下进程执行示意图

3.9.3 线程与进程的区别

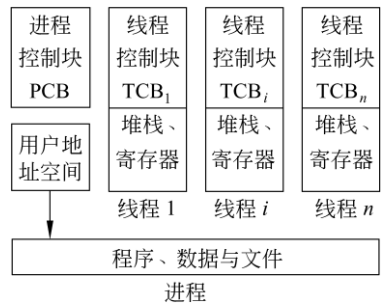


图 3.27 多线程与进程的关系

3.9.4 线程的适用范围

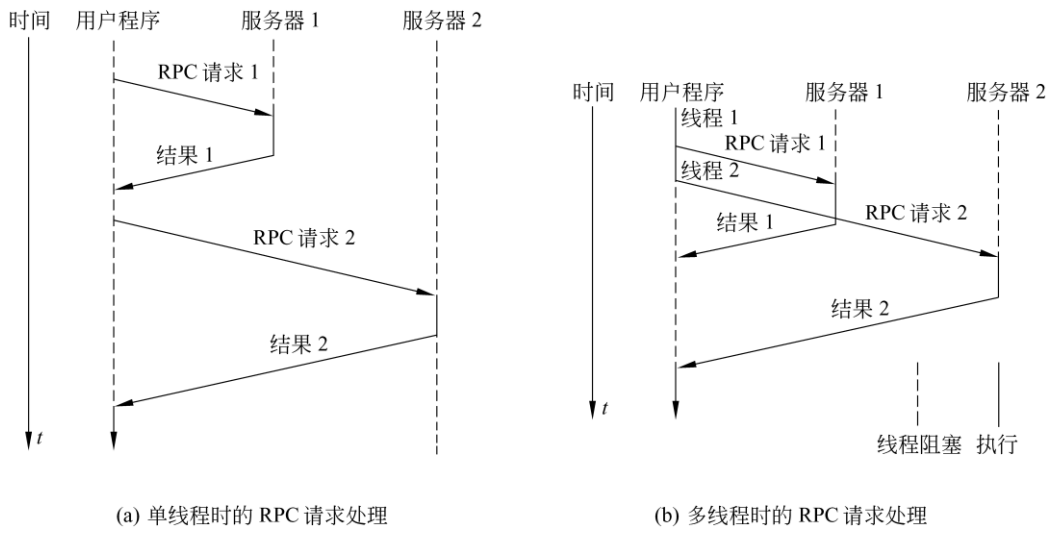


图 3.28RPC 请求处理

### 3.10 线程分类与执行

#### 3.10.1 线程的分类

操 作	用户级线程	系统级线程	进 程
Null Fork	34	948	11 300
信号等待	37	441	1840

图 3.29 线程、进程等的上下文切换开销

#### 3.10.2 线程的执行特性

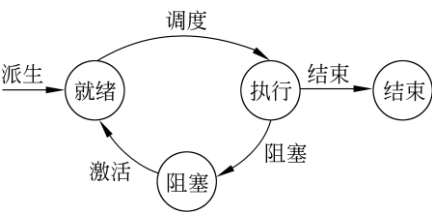


图 3.30 线程的状态与操作

## 本章小结

## 习题

3.1 有人说，一个进程是由伪处理机执行的一个程序，这话对吗？为什么？

3.2 试比较进程和程序的区别。

3.3 我们说程序的并发执行将导致最终结果失去封闭性。这话对所有的程序都成立吗？试举例说明。

3.4 试比较作业和进程的区别。

3.5 UNIX System V 中，系统程序所对应的正文段未被考虑成进程上下文的一部分，为什么？

3.6 什么是临界区？试举一临界区的例子。

3.7 并发进程间的制约有哪 2 种？引起制约的原因是什么？

3.8 什么是进程间的互斥？什么是进程间的同步？

3.9 试比较 P,V 原语法和加锁法实现进程间互斥的区别。

3.10 设在书 3.6 节中所描述的生产者-消费者问题中，其缓冲部分为  $m$  个长度相等的有界缓冲区组成，且每次传输数据长度等于有界缓冲区长度以及生产者和消费者可对缓冲区同时操作。重新描述发送过程  $\text{deposit}(\text{data})$  和接收过程  $\text{remove}(\text{data})$ 。

3.11 两进程  $P_A$ ,  $P_B$  通过两 FIFO 缓冲区队列连接(如图 3.31 所示)，每个缓冲区长度等于传送消息长度。进程  $P_A$ ,  $P_B$  之间的通信满足如下条件：

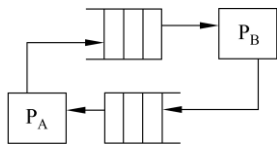


图 3.31 习题 3.11 图

(1) 至少有一个空缓冲区存在时，相应的发送进程才能发送一个消息。

(2) 当缓冲队列中至少存在一个非空缓冲区时，相应的接收进程才能接收一个消息。

试描述发送过程  $\text{send}(i, m)$  和接收过程  $\text{receive}(i, m)$ 。这里  $i$  代表缓冲队列。

3.12 在和控制台通信的例中，设操作员不仅仅回答用户进程所提出的问题，而且还能

独立地向各用户进程发出指示。对于这些指示，操作员不要求用户进程回答，但它们享有比其他消息优先传送的优先度。即如果 inbuf 中有指示存在，系统不能进行下一次通信会话。试按上述要求重新描述 CCP 和 KCP，DCP。

3.13 编写一个程序使用系统调用 fork 生成 3 个子进程，并使用系统调用 pipe 创建一管道，使得这 3 个子进程和父进程公用同一管道进行信息通信。

3.14 设有 5 个哲学家，共享一张放有五把椅子的桌子，每人分得一把椅子。但是，桌子上总共只有 5 支筷子，在每人两边分开各放一支。哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。

条件：

- (1) 只有拿到两支筷子时，哲学家才能吃饭。
- (2) 如果筷子已在他人手上，则该哲学家必须等待到他人吃完之后才能拿到筷子。
- (3) 任一哲学家在自己未拿到两支筷子吃饭之前，决不放下自己手中的筷子。

试：

- (1) 描述一个保证不会出现两个邻座同时要求吃饭的通信算法。
- (2) 描述一个既没有两邻座同时吃饭，又没有人饿死(永远拿不到筷子)的算法。
- (3) 在什么情况下，5 个哲学家全部吃不上饭？

3.15 什么是线程？试述线程与进程的区别。

3.16 使用库函数 clone() 与 creat thread() 在 Linux 环境下创建两种不同执行模式的线程程序。

# 第 4 章处理机调度

## 4.1 分级调度

### 4.1.1 作业的状态及其转换

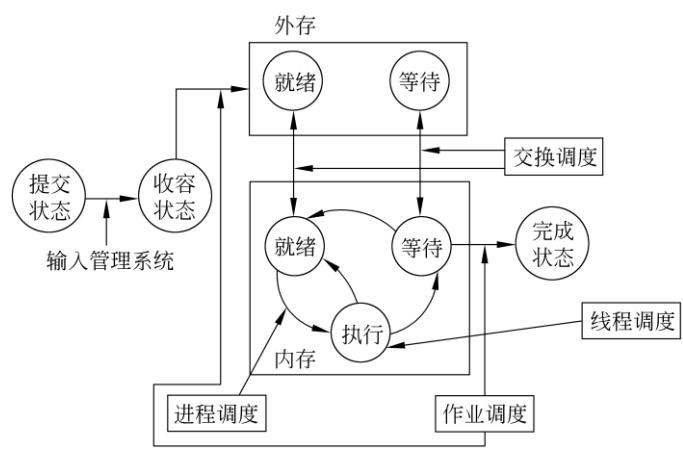


图 4.1 作业的状态及其转换

### 4.1.2 调度的层次

### 4.1.3 作业与进程的关系

## 4.2 作业调度

### 4.2.1 作业调度功能

作业名
作业类型
资源要求
资源使用情况
优先级(数)
当前状态
其他

图 4.2 作业控制块 JCB

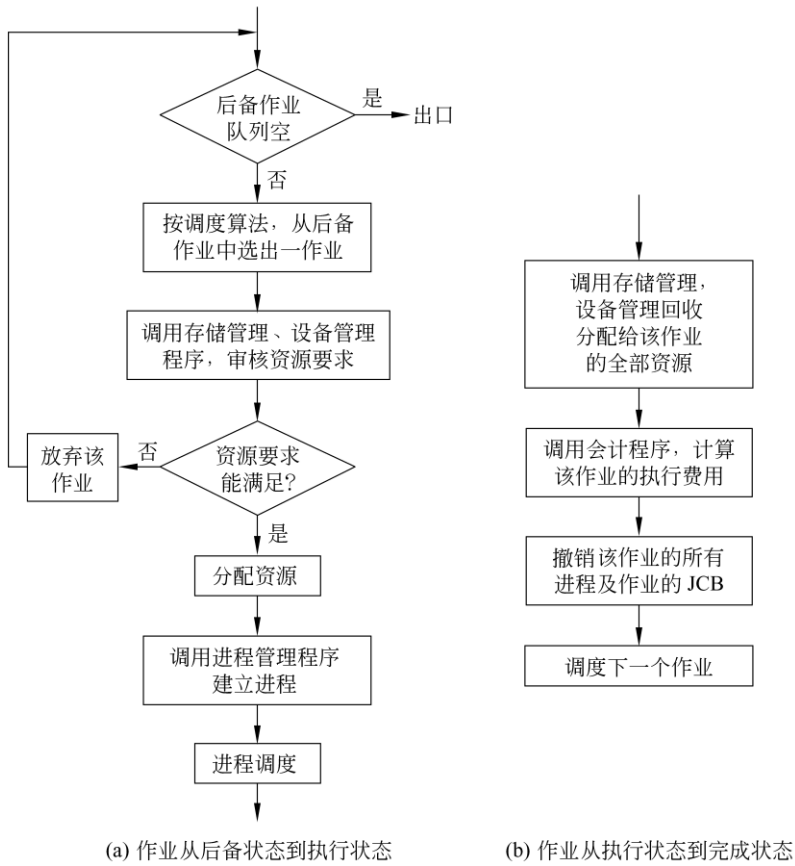


图 4.3 作业调度中状态的转换过程



## 4.2.2 作业调度目标与性能衡量

1. 周转时间
2. 带权周转时间

## 4.3 进程调度

### 4.3.1 进程调度的功能

1. 记录系统中所有进程的执行情况
2. 选择占有处理机的进程
3. 进行进程上下文切换

### 4.3.2 进程调度的时机

### 4.3.3 进程调度性能评价

## 4.4 调度算法

1. 先来先服务(FCFS)调度算法
2. 轮转法

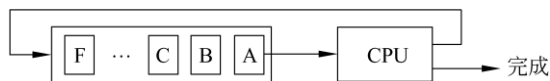


图 4.4 轮转法调度

3. 多级反馈轮转法
4. 优先级法

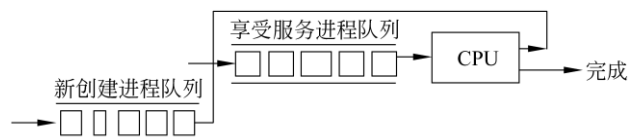


图 4.5 线性优先级调度

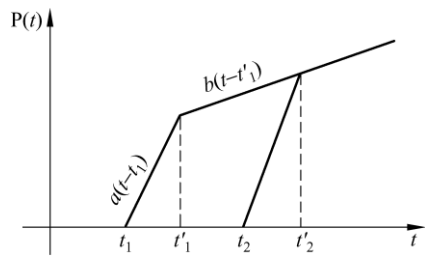


图 4.6 优先级变化曲线

- 5. 最短作业优先法
- 6. 最高响应比优先法

4.5 算法评价

4.5.1FCFS 方式的调度性能分析

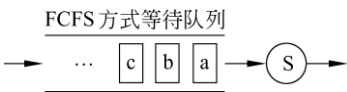


图 4.7FCFS 方式的评价模型

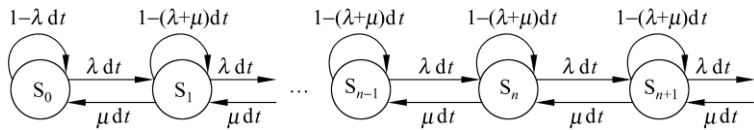


图 4.8 状态转换图

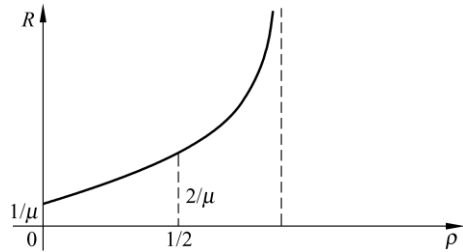


图 4.9 平均响应时间 R 和  $\rho$  的关系

## 4.5.2 轮转法调度性能评价

## 4.5.3 线性优先级法的调度性能

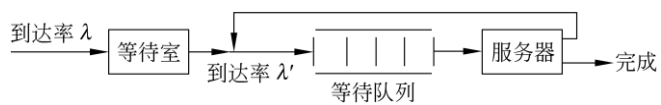


图 4.10 线性优先级调度的评价模型

## 4.6 实时系统调度方法

### 4.6.1 实时系统的特点

1. 很快的进程或线程切换速度
2. 快速的外部中断响应能力
3. 基于优先级的随时抢先式调度策略

### 4.6.2 实时调度算法的分类

1. 静态表格驱动类
2. 静态优先级驱动抢先式调度算法类
3. 动态计划调度算法类
4. 尽力而为调度算法类

### 4.6.3 时限调度算法与频率单调调度算法

1. 任务就绪时间或事件到达时间
2. 开始时限
3. 完成时限
4. 处理时间
5. 资源需求
6. 优先级

进 程	事件发生时限	执行时限	结束时限
DA(1)	0	15	30
DA(2)	30	15	60
DA(3)	60	15	90
⋮	⋮	⋮	⋮
DB(1)	0	38	75
DB(2)	75	38	150
DB(3)	150	38	225
⋮	⋮	⋮	⋮

图 4.11 周期性任务的预计发生、执行与结束时限

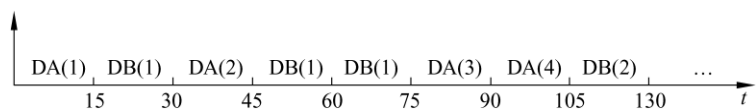


图 4.12 时限调度算法给出的调度顺序

## 本章小结

## 习题

- 4.1 什么是分级调度？分时系统中有作业调度的概念吗？如果没有，为什么？
- 4.2 试述作业调度的主要功能。
- 4.3 作业调度的性能评价标准有哪些？这些性能评价标准在任何情况下都能反映调度策略的优劣吗？
- 4.4 进程调度的功能有哪些？
- 4.5 进程调度的时机有哪几种？
- 4.6 假设有4道作业，它们的提交时刻及执行时间由下表给出：

作业号	提交时刻/小时	执行时间/小时
1	10.00	2
2	10.20	1
3	10.40	0.5
4	10.50	0.3

计算在单道程序环境下，采用先来先服务调度算法和最短作业优先调度算法时的平均周转时间和平均带权周转时间，并指出它们的调度顺序。

- 4.7 设某进程所需要的服务时间  $t = k * q$ ，其中， $k$  为时间片的个数， $q$  为时间片长度且为常数。当  $t$  为一定值时，令  $q$  趋于 0，则有  $k$  趋于无穷，从而服务时间为  $t$  的进程响应时间  $T$  为  $t$  的连续函数。对应于时间片调度方式 RR，先来先服务方式 FCFS 和线性优先级调度方式 SRR，其响应时间函数分别为：

$$T_{rr}(t) = t * \mu / (\mu - \lambda)$$

$$T_{fc}(t) = 1 / (\mu - \lambda)$$

$$T_{sr}(t) = 1 / (\mu - \lambda) - (1 - t * \mu) / (\mu - \lambda')$$

其中  $\lambda' = ((1 - b)/a) * \lambda = r * \lambda$

取  $(\lambda, \mu) = (50, 100)$  和  $(\lambda, \mu) = (80, 100)$ ，分别改变  $r$  的值，画出  $T_{rr}(t)$ 、 $T_{fc}(t)$  和  $T_{sr}(t)$  的时间变化图。

- 4.8 什么是多处理机系统？并行处理系统、计算机网络、分布式系统和多处理机系统的操作系统之间有何区别？
- 4.9 什么是实时调度？它与非实时调度相比，有何区别？
- 4.10 写出图 4.11 所示周期性任务调度用的时限调度算法。
- 4.11 设周期性任务  $P_1, P_2, P_3$  的周期  $T_1, T_2, T_3$  分别为 100, 150, 350，执行时间分别为 20, 40, 100，问：是否可用频率单调调度算法进行调度？

# 第 5 章 存储管理

## 5.1 存储管理的功能

### 5.1.1 虚拟存储器

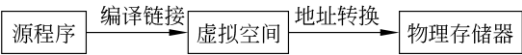


图 5.1 地址变换与物理存储器

### 5.1.2 地址变换

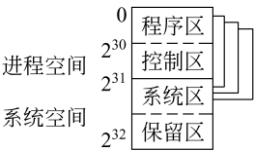


图 5.2 虚拟空间的划分

- 1. 静态地址重定位
- 2. 动态地址重定位

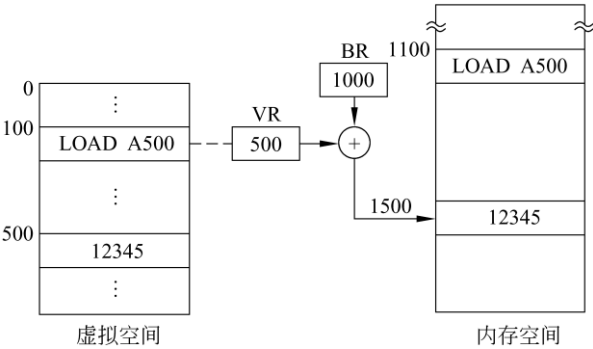


图 5.3 动态地址重定位



### 5.1.3 内外存数据传输的控制

### 5.1.4 内存的分配与回收

### 5.1.5 内存信息的共享与保护

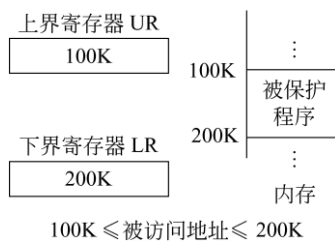


图 5.4 上、下界寄存器保护法

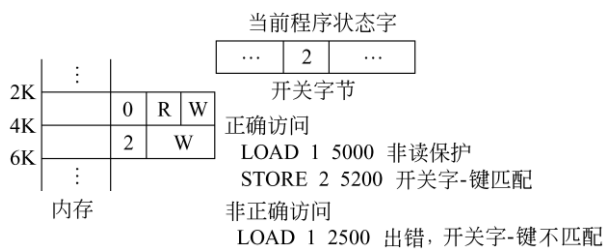


图 5.5 保护键保护法

# 5.2 分区存储管理

## 5.2.1 分区管理基本原理

### 1. 固定分区法

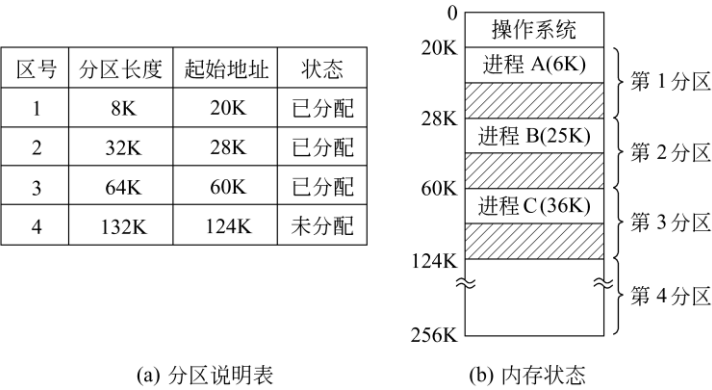


图 5.6 固定分区法

### 2. 动态分区法

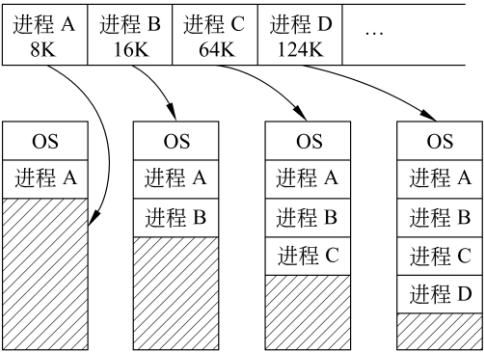


图 5.7 内存初始分配情况

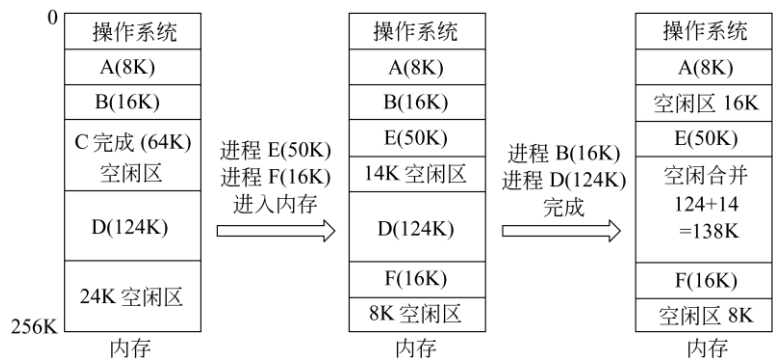


图 5.8 内存分配变化过程

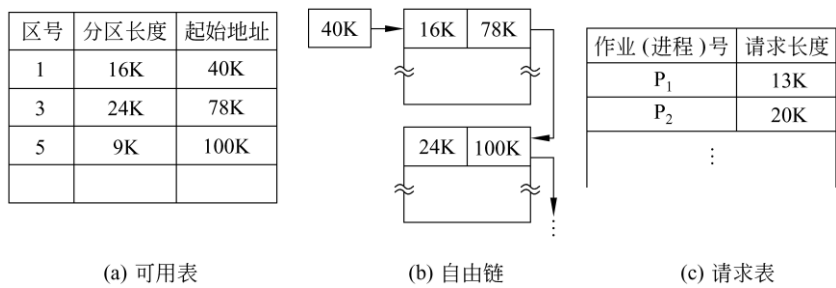


图 5.9 可用表、自由链及请求表

## 5.2.2 分区的分配与回收

### 1. 固定分区时的分配与回收

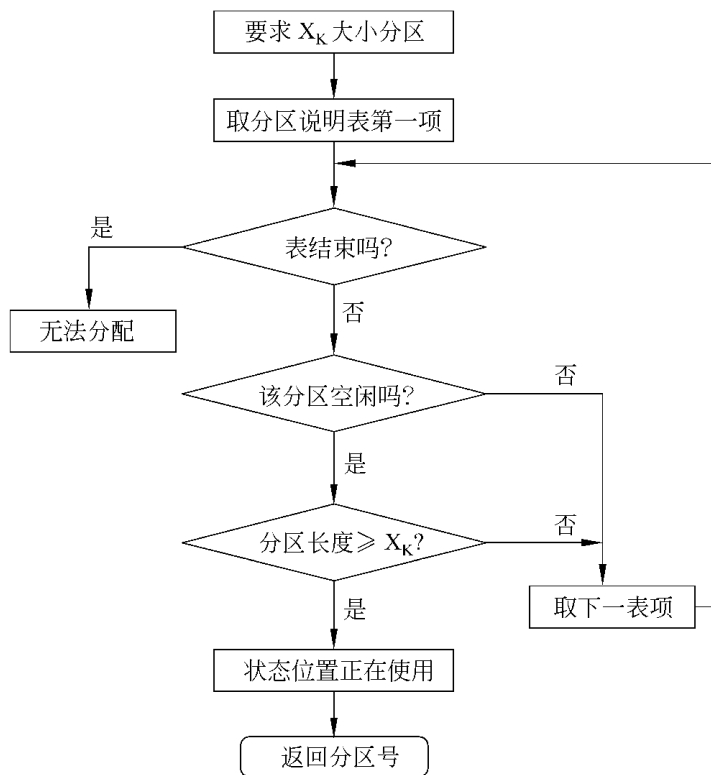


图 5.10 固定分区分配算法

## 2. 动态分区时的分配与回收

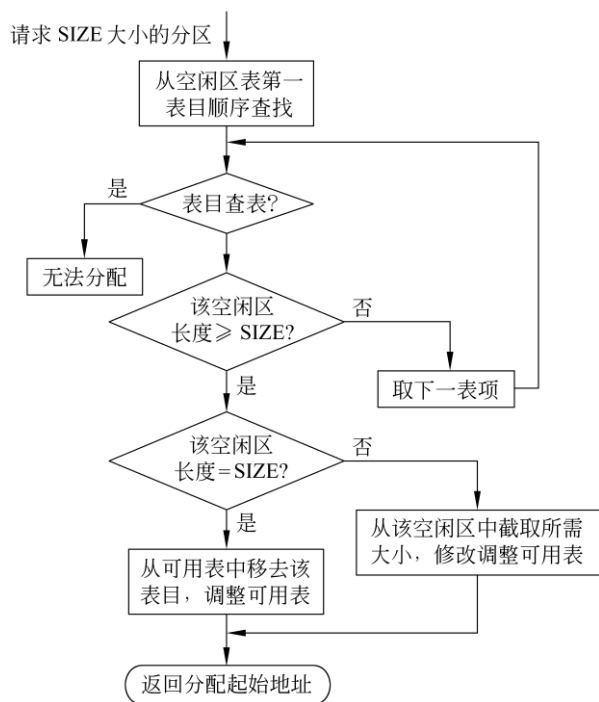


图 5.11 最先适应算法

## 3. 动态分区时的回收与拼接

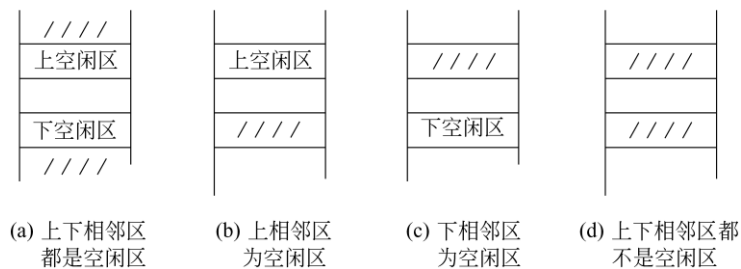


图 5.12 空闲区的合并

4. 几种分配算法的比较

5.2.3 有关分区管理其他问题的讨论

- 1. 关于虚存的实现
- 2. 关于内存扩充
- 3. 关于地址变换和内存保护
- 4. 分区存储管理的主要优缺点

5.3 覆盖与交换技术

5.3.1 覆盖技术

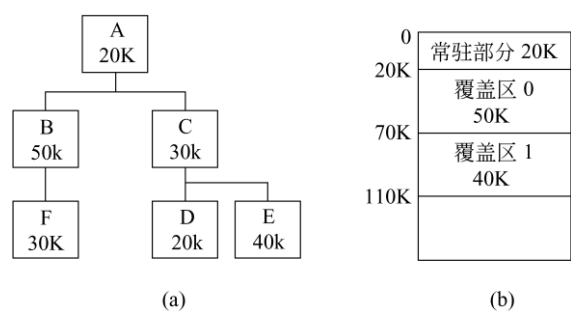


图 5.13 覆盖示例

5.3.2 交换技术

5.4 页式管理

5.4.1 页式管理的基本原理



图 5.14 页的划分

页号	页面号

图 5.15 基本页表示例

5.4.2 静态页面管理

1. 内存页面分配与回收

进程号	请求页面数	页表始址	页表长度	状态
1	20	1024	20	已分配
2	34	1044	34	已分配
3	18	1078	18	已分配
4	21	⋮	⋮	未分配
⋮	⋮			⋮

图 5.16 请求表示例

	19	18	17	16	15	⋯	4	3	3	2	0
0	1	1	1	1	1	⋯	1	1	0	1	1
0	0	0	1	1	1	⋯	0	0	1	1	0
0	0	1	1	1	1	⋯	0	0	0	0	0

图 5.17 位示图

2. 分配算法

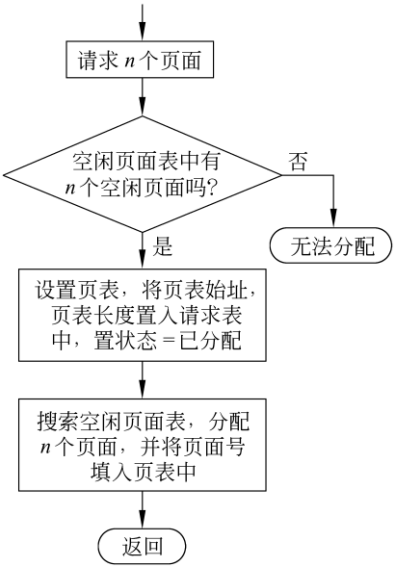


图 5.18 页面分配算法流程图

3. 地址变换

页号	页面号
0	2
1	3
2	8

图 5.19 页号与页面号

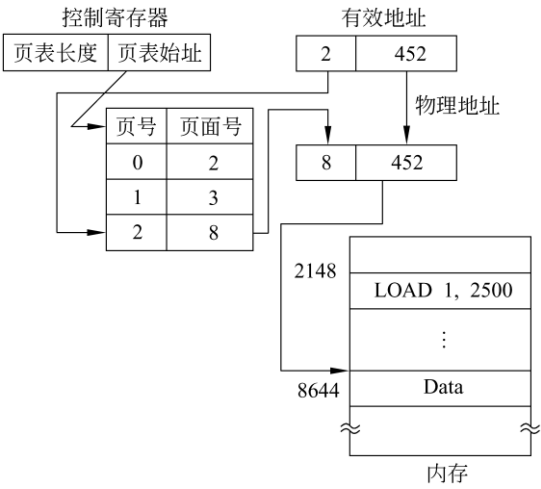




图 5.20 地址变换

5.4.3 动态页式管理

页号	页面号	中断位	外存始址
0			
1			
2			
3			

图 5.21 加入中断处理后的页表

页号	页面号	中断位	外存始址	改变位

图 5.22 加入改变位后的页表

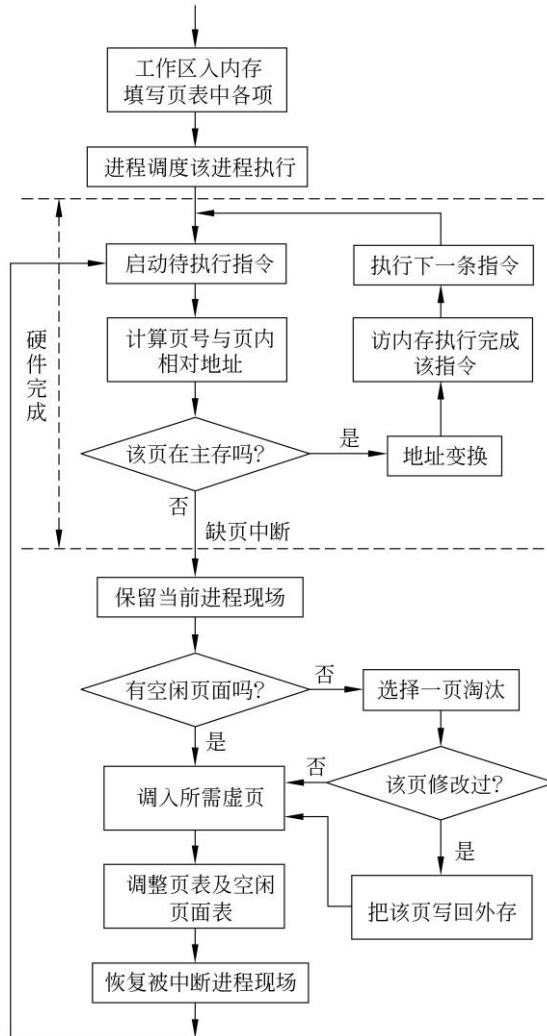


图 5.23 动态页式管理流程图

#### 5.4.4 请求页式管理中的置换算法

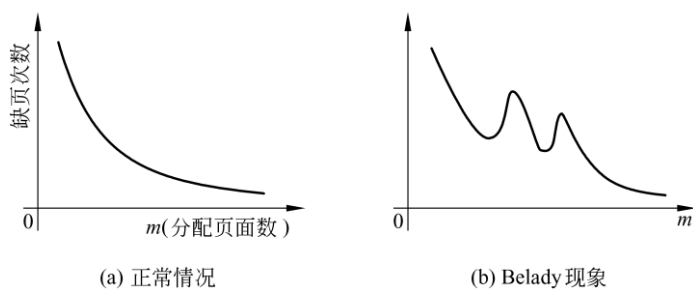


图 5.24 FIFO 算法的 Belady 现象

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0
	0	0	0	0	3	3	3	2	2	2	2	2	i	i	i	i
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2

图 5.25 Belady 现象示例(1)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	7	7	3	3	3	3	3	3	3	3	3	2	2	2
	0	0	0	0	0	0	4	4	4	4	4	4	4	4	4	4
		1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
			2	2	2	2	2	2	2	2	2	2	1	1	1	1

图 5.26 Belady 现象示例(2)

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

图 5.27 Belady 现象示例(3)

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

图 5.28 Belady 现象示例(4)

### 5.4.5 存储保护

### 5.4.6 页式管理的优缺点

## 5.5 段式与段页式管理

### 5.5.1 段式管理的基本思想

### 5.5.2 段式管理的实现原理

1. 段式虚存空间
2. 段式管理的内存分配与释放

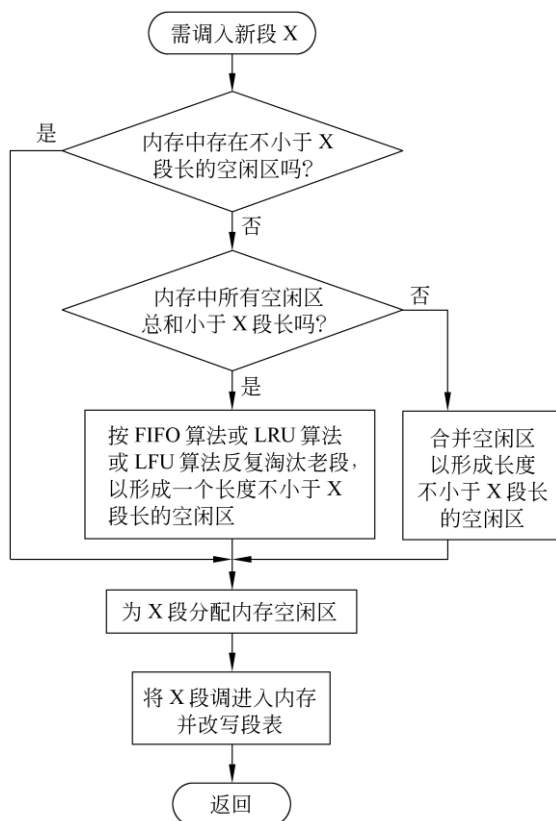


图 5.29 缺段中断处理过程

### 3. 段式管理的地址变换

段号	始址	长度	存取方式	内外	访问位

图 5.30 段表

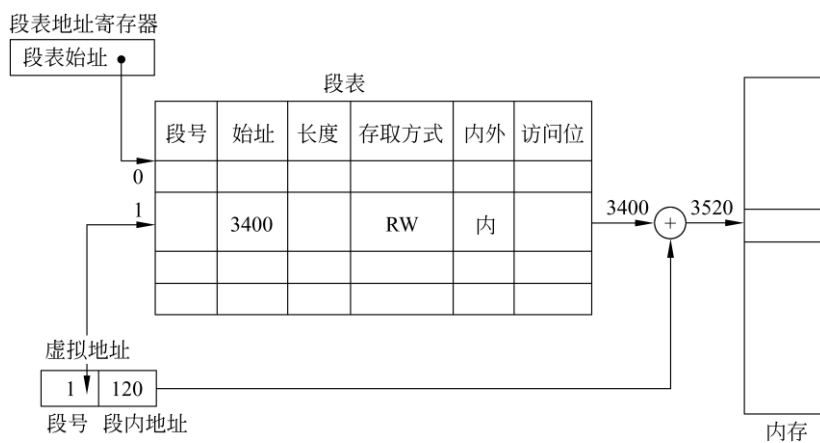


图 5.31 段式地址变换过程

4. 段的共享与保护

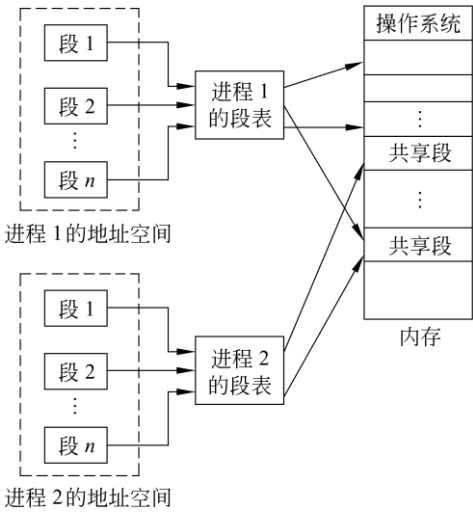


图 5.32 段式系统中共享内存副本

### 5.5.3 段式管理的优缺点

### 5.5.4 段页式管理的基本思想

### 5.5.5 段页式管理的实现原理

#### 1. 虚地址的构成

#### 2. 段表和页表

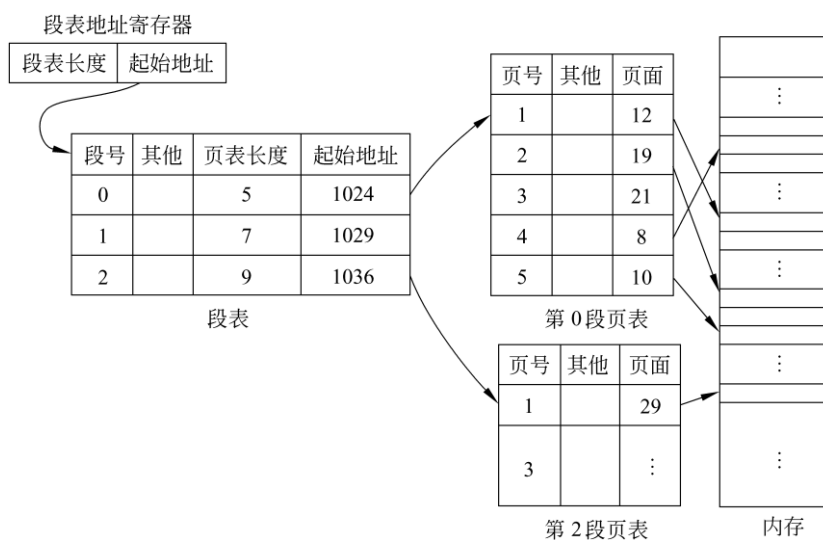


图 5.33 段页式管理中段表、页表与内存的关系

#### 3. 动态地址变换过程

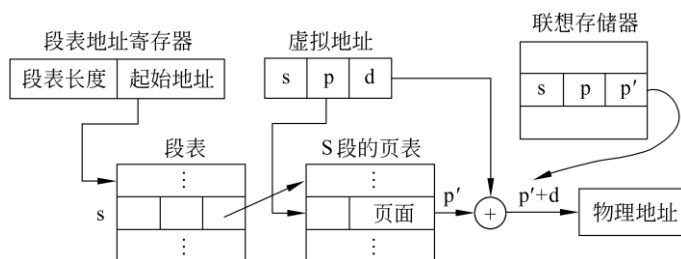


图 5.34 段页式地址变换

## 5.6 局部性原理和抖动问题

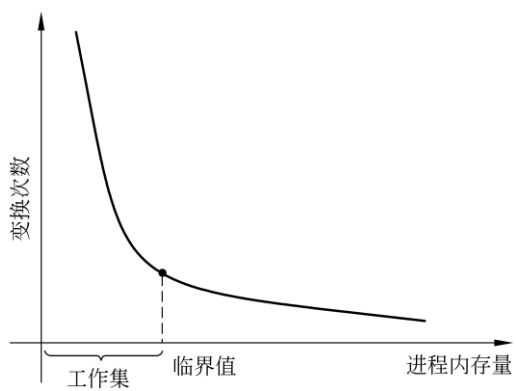


图 5.35 内存与交换次数的关系



## 本章小结

方法 功 能	单一 连续区	分 区 式		页 式		段 式	段页式
		固定分区	可变分区	静态	动态		
适用环境	单道	多道		多道		多道	多道
虚拟空间	一维	一维		一维		二维	二维
重定位方式	静态	静态 动态		动态		动态	动态
分配方式	静态分配连续区	静态 动态 分配 连续区		静态或动态页为单位非连续		动态分配段为单位非连续	动态分配页为单位非连续
释放	执行完成后全部释放	执行完成后全部释放	分区释放	执行完成后释放	淘汰与执行完后释放	淘汰与执行完成后释放	淘汰与执行完成后释放
保护	越界保护或没有	越界保护与保护键		越界保护与控制权保护		同左	同左
内存扩充	覆盖与交换技术	同左		同左	外存、内存统一管理的虚存	同左	同左
共享	不能	不能		较难		方便	方便
硬件支持	保护用寄存器	保护用同左 寄存器加重定位机构		地址变换机构 中断机构 保护机构		段式地址变换机，保护与中断，动态连接机构	同左

图 5.36 各种存储方法比较

## 习题

5.1 存储管理的主要功能是什么？

5.2 什么是虚拟存储器，其特点是什么？

5.3 实现地址重定位的方法有哪几类？形式化地描述动态重定位过程。

5.4 常用的内存信息保护方法有哪几种？它们各自的特点是什么？

5.5 如果把 DOS 的执行模式改为保护模式，起码应作怎样的修改？

5.6 动态分区式管理的常用内存分配算法有哪几种？比较它们各自的优缺点。

5.75.2 节讨论的分区式管理可以实现虚存吗？如果不能，需怎样修改？试设计一个分区式管理实现虚存的程序流程图。如果能，试说明理由。

5.8 简述什么是覆盖？什么是交换？覆盖和交换的区别是什么？

5.9 什么是页式管理？静态页式管理可以实现虚存吗？

5.10 什么是请求页式管理？试设计和描述一个请求页式管理时的内存页面分配和回收算法(包括缺页处理部分)。

5.11 请求页式管理中有哪几种常用的页面置换算法？试比较它们的优缺点。

5.12 什么是 Belady 现象？试找出一个 Belady 现象的例子。

5.13 描述一个包括页面分配与回收、页面置换和存储保护的请求页式存储管理系统。

5.14 什么是段式管理？它与页式管理有何区别？

5.15 段式管理可以实现虚存吗？如果可以，简述实现方法。

5.16 为什么要提出段页式管理？它与段式管理及页式管理有何区别？

5.17 为什么说段页式管理时的虚地址仍是二维的？

5.18 段页式管理的主要缺点是什么？有什么改进办法？

5.19 什么是局部性原理？什么是抖动？你有什么办法减少系统的抖动现象？

## 第6章 进程与存储管理示例

### 6.1 Linux 进程和存储管理简介

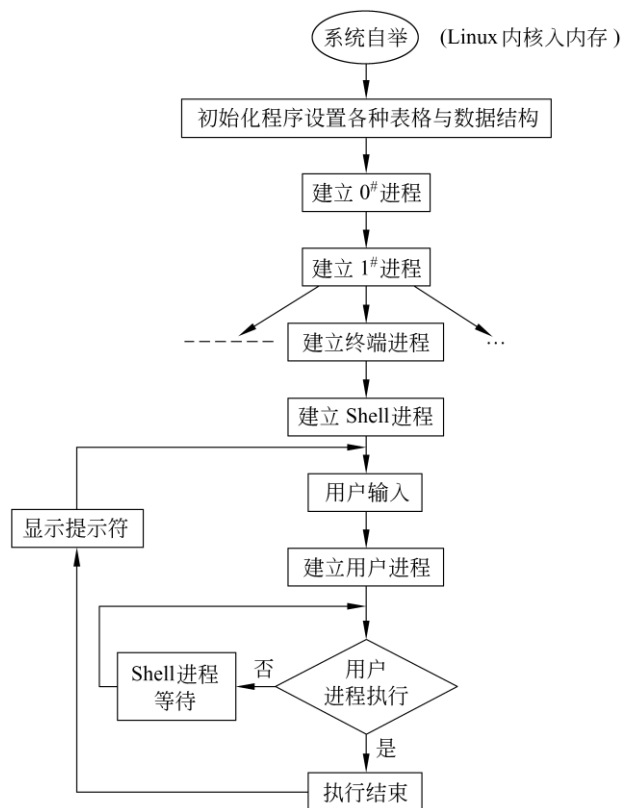


图 6.1 Linux 系统中各进程的关系

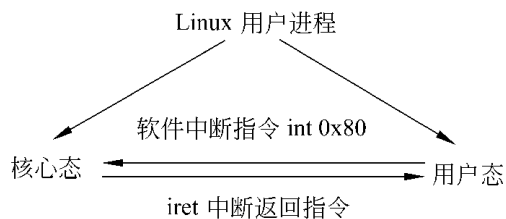


图 6.2 Linux 进程的核心态与用户态之间的转换

# 6.2Linux 进程结构

## 6.2.1 进程的概念

## 6.2.2 进程的虚拟地址结构

## 6.2.3 进程上下文

### 1. 进程上下文的基本结构

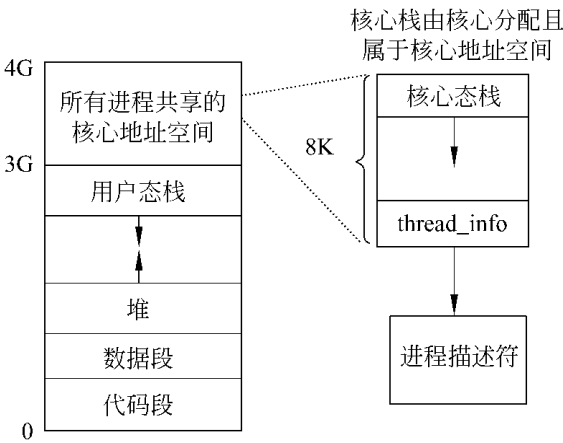


图 6.3 进程空间结构

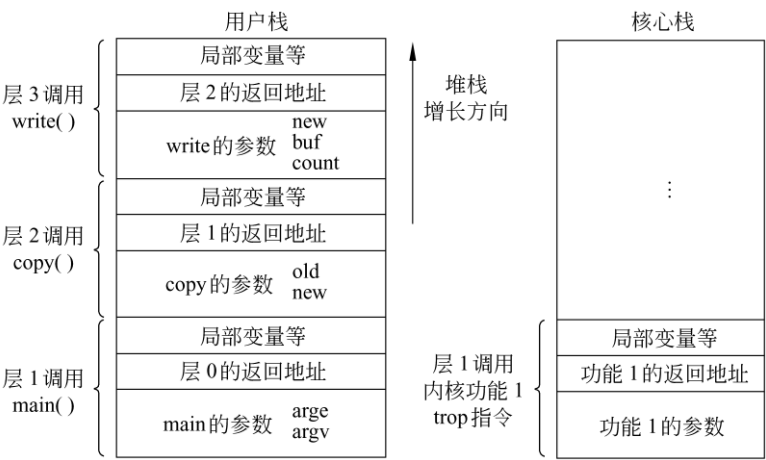


图 6.4 执行 copy 程序时用户栈和核心栈的变化

## 2. 进程上下文的组成部分

### 6.2.4 进程的状态和状态转换

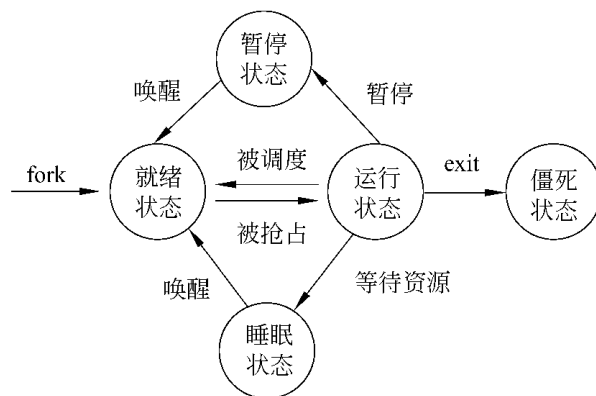


图 6.5 进程状态转换图

## 6.2.5 小结

## 6.3 进程控制

### 6.3.1 Linux 启动及进程树的形成

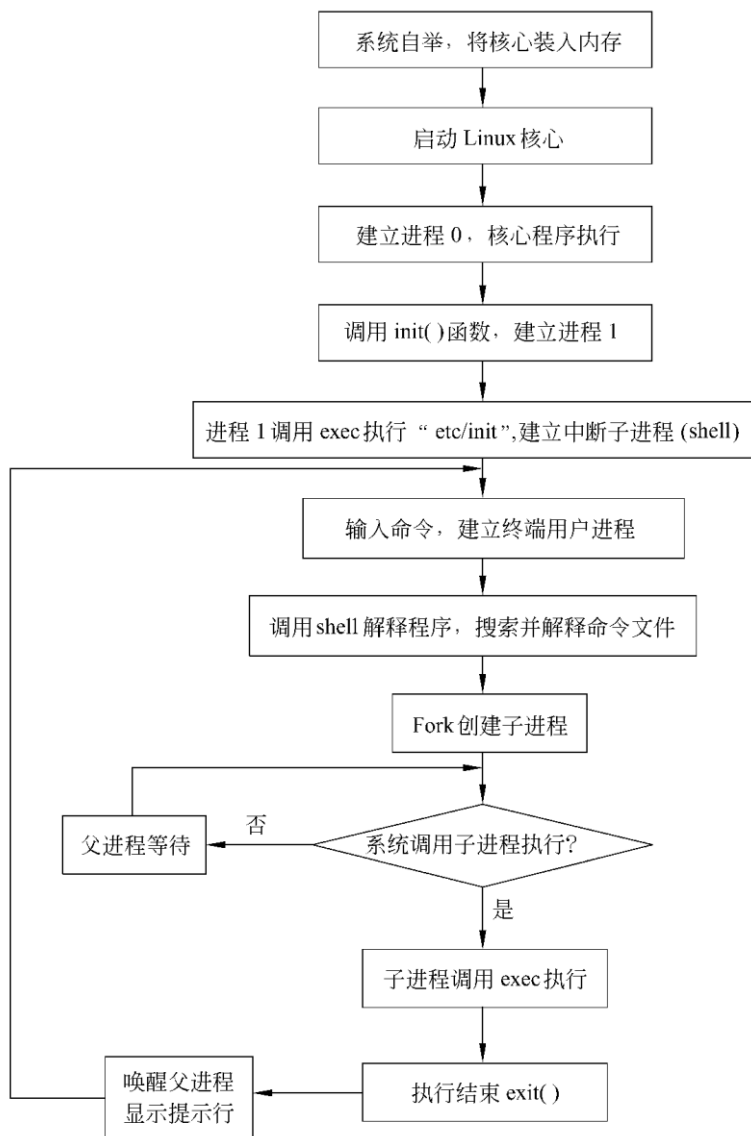


图 6.6 进程树的形成

## 6.3.2 进程控制

### 1. 进程的创建

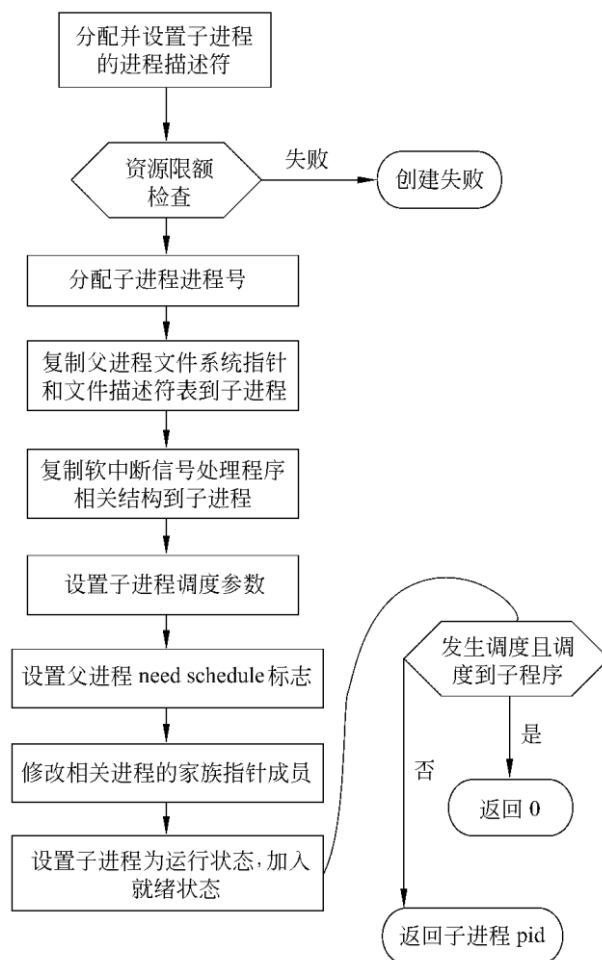


图 6.7fork 流程图

### 2. 执行一个文件的调用

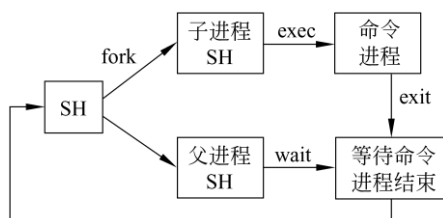


图 6.8shell 执行过程

```
#include <stdio.h>
main ()
{
    char  command[32];
    char  * prompt = " $ ";
    while ( printf ("%s",prompt) ,gets (command) != NULL)
    {
        if ( fork () == 0)
            execlp (command ,command ,(char *)0);
        else
            wait (0);
    }
}
```

图 6.9shell 程序的实现



### 3. 进程的终止

## 6.4Linux 进程调度

1. 调度原理
2. 调度的时机
3. 调度标识的设置
4. 调度策略与优先数的计算
5. 调度的实现

## 6.5 进程通信

### 6.5.1Linux 的低级通信

软中断号	符号名	功    能	软中断号	符号名	功    能
1	SIGHUP	用户终端连接结束	17	SIGCHLD	子进程消亡
2	SIGINT	键盘打入 DELETE 键	18	SIGCONT	继续进程的执行
3	SIGQUIT	键盘打入 QUIT 键	19	SIGSTOP	停止进程的执行
4	SIGILL	非法指令	20	SIGTSTP	键盘打入 SUSP 键
5	SIGTRAP	断点或跟踪指令	21	SIGTTIN	后台进程读控制终端
6	SIGABRT	程序 ABORT	22	SIGTTOU	后台进程写控制终端
7	SIGBUS	非法地址	23	SIGURG	socket 收到紧急数据
8	SIGFPE	浮点溢出	24	SIGXCPU	超过 CPU 资源限制
9	SIGKILL	要求终止该进程	25	SIGXFSZ	超过文件资源限制
10	SIGUSR1	用户定义	26	SIGVTALRM	虚拟时钟定时信号
11	SIGSEGV	段违例	27	SIGPROF	虚拟时钟定时信号 2
12	SIGUSR2	用户定义	28	SIGWINCH	窗口大小改变
13	SIGPIPE	PIPE 只有写者无读者	29	SIGIO	IO 就绪
14	SIGALRM	时钟定时信号	30	SIGPWR	电源失效
15	SIGTERM	软件终止信号	31	SIGSYS	系统调用错
16					

图 6.10Linux 软中断信号

## 6.5.2 进程间通信 IPC

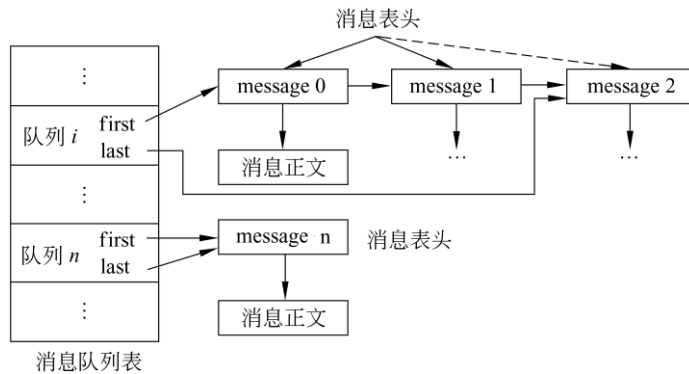


图 6.11 索引表与实例表的关系

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSGKEY 75
struct msgform
{
    long mtype;
    char mtext[256];
};
main()
{
    struct msgform msg;
    int msgqid, pid, *pint;
    msgqid = msgget(MSGKEY, 0777);          /* 建立消息队列 */
    pid = getpid();
    pint = (int *) msg.mtext;
    *pint = pid;
    msg.mtype = 1;                          /* 指定消息类型 */
    msgsnd(msgqid, &msg, sizeof(int), 0);    /* 往 msgqid 发送消息 msg */
    msgrcv(msgqid, &msg, 256, pid, 0);       /* 接收来自服务进程的消息 */
    printf("client : receive from pid%d\n", *pint);
}
```

图 6.12 顾客进程的程序段

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSGKEY 75
struct msgform
{
    long mtype;
    char mtext [256];
} msg;
int msgqid;
main ()
{
    int i,pid,*pint;
    extern cleanup ();
    for (i=0; i<20; i++)
        signal (i,cleanup);
    msgqid = msgget (MSGKEY,0777 | IPC_CREAT);
    for (;;)
    {
        msgrcv (msgqid,&msg,256,1,0);
        pint = (int *) msg.mtext;
        pid = *pint;
        printf("server: receive from pid %d\n",pid);
        msg.mtype = pid;
        *pint = getpid ();
        msgsnd (msgqid,&msg,sizeof(int),0);
    }
}
cleanup ()
{
    msgctl (msgqid ,IPC_RMID,0);
    exit ();
}

```

/\* 软中断处理 \*/

/\* 建立与顾客进程相同的消息队列 \*/

/\* 接收来自顾客进程的消息 \*/

/\* 发送应答消息 \*/

图 6.13 服务者进程的程序段

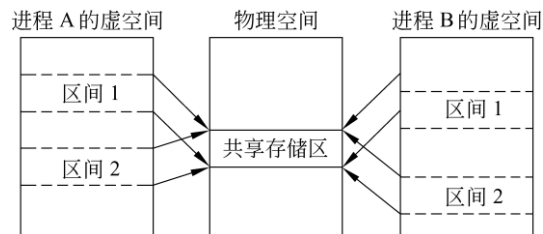


图 6.14 共享存储区示意图

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 75
#define K      1024
int shmid;
main ()
{
    int i , * pint;
    char * addr;
    extern char * shmat();
    extern cleanup();
    for (i=0; i<20; i++)                /* 软中断处理 */
        signal (i,cleanup);
    shmid = shmget (SHMKEY,16 * K,0777|IPC_CREAT); /* 建立 16KB 共享区
                                                    SHMKEY */
    addr = shmat (shmid,0,0);             /* 共享区首地址 */
    printf ("addr 0x%x \n",addr);
    pint = (int *) addr;
    for (i=0; i<256; i++)
        * pint++ = i;
    pint = (int *) addr;                 /* 共享区第一个字中写入长
                                          度 256,以便接收进程读 */
    * pint = 256;
    pause ();                           /* 等待接收进程读 */
}
cleanup ()
{
    shmctl (shmid ,IPC_RMID ,0);
    exit ();
}

```

图 6.15 共享存储区程序实例

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 75
#define K 1024
int shmid;
main ()
{
    int i, *pint;
    char *addr;
    extern char *shmat ();
    shmid = shmget(SHMKEY, 8 * K, 0777); /* 取共享区 SHMKEY 的 id */
    addr = shmat (shmid, 0, 0);          /* 连接共享区 */
    pint = (int *)addr;
    while (*pint == 0);                  /* 共享区的第一个字节为零时,等待 */
    for (i=0; i<256; *pint++)           /* 打印共享区中内容 */
        printf ("%d\n", *pint++);
}
```

图 6.16 共享存储区程序实例

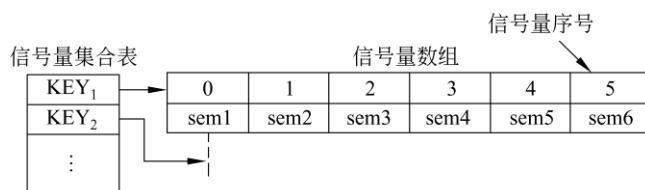


图 6.17 信号量数组

## 6.6 Linux 存储管理

### 6.6.1 虚存空间和管理

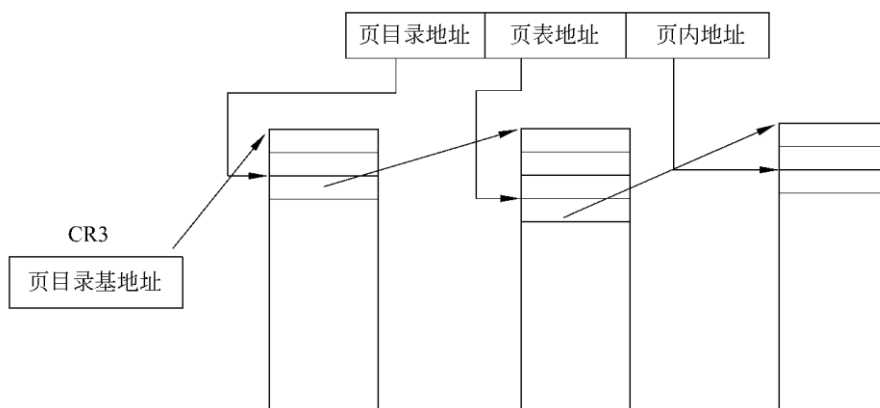


图 6.18 虚拟地址的内容

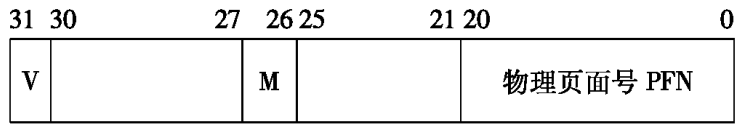


图 6.19 页表项内容

## 6.6.2 请求调页技术

### 1. 交换缓冲

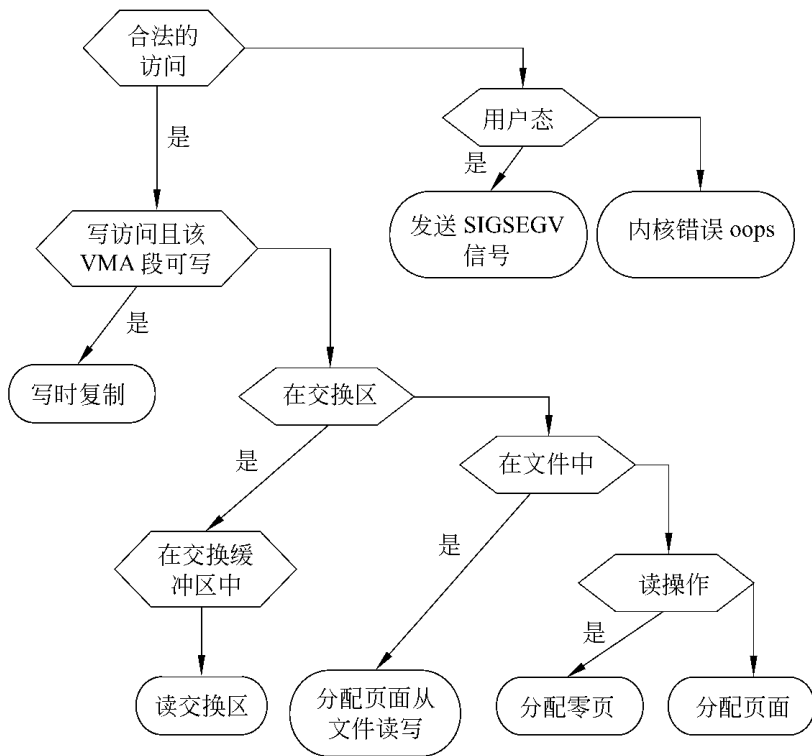


图 6.20 请求调页的调入基本处理过程

## 2. 页面换出过程

# 本章小结

## 习题

6.1 简述 Linux 系统进程的概念。

6.2 Linux 进程上下文由哪几部分组成？为什么说核心程序不是进程上下文一部分？进程页表也在核心区，它们也不是进程上下文的一部分吗？

6.3 假定在用户态下执行的某个进程用完了它的时间片，由于时钟中断的原因，核心调度一个新进程去执行。请形式化地描述出新、旧进程的上下文切换过程。

6.4 Linux 的调度策略是什么？调度时应该封锁中断吗？如果不封锁，会发生什么问题？

6.5 试述进程 0 的作用。

6.6 Linux 在哪几种情况下发生调度？

6.7 编写一个程序，利用 fork 调用创建一个子进程，并让该子进程执行一个可执行文件。

6.8 什么是软中断？

6.9 进程在什么时候处理它接收到的软中断信号？进程接收到软中断信号后放在什么地方？

6.10 Shell 符号 `>>` 将输出追加到一个指定的文件中，如果指定文件不存在，则该命令创建该文件并将输出写入其中。否则，它打开该文件并在该文件中数据尾部接着写入。编写实现“`>>`”的 C 语言代码。

6.11 编写一程序，比较使用共享存储区和消息机制进行数据传输的速度。

6.12 形式化地描述 Linux 中消息机制的通信原理。

6.13 Linux 存储管理策略中交换和请求调页方式有何区别？

6.14 在图 6.20 所示请求调页的调入处理过程中，有可能出现空闲页面链表中的页面内容不同于外存设备上的页面内容的情况，此时应取哪一个页面调入内存，为什么？

6.15 小结 Linux 进程管理与存储管理部分的联系。

## 第7章 Windows 的进程与内存管理

### 7.1 Windows NT 的特点及相关的概念

#### 7.1.1 Windows NT 体系结构的特点

#### 7.1.2 Windows 的管理机制

1. 核心态(kernel mode)和用户态(user mode)
2. Windows 操作系统的体系结构
3. 系统调用、中断和陷阱
4. 利用对象来共享系统资源
5. 本地过程调用

### 7.2 Windows 进程和线程

#### 7.2.1 Windows 的进程和线程的定义

#### 7.2.2 进程和线程的关联

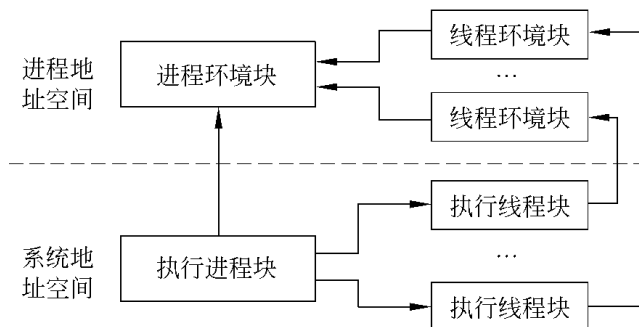




图 7.1Windows 进程和线程的关联

7.2.3Windows 进程的结构

表 7.1 Windows 进程的数据结构

执行进程块的数据项	功 能
进程标志	唯一的进程标志号、父进程标志号、运行的映像文件( Image) 名称
系统资源配额	对该进程所使用系统内存池以及分页文件的配额限制
虚拟内存管理	用来描述进程的哪些虚拟地址空间已被占用、哪些空间可以使用, 以及进程虚拟内存管理的状态信息
工作集信息	该进程的虚拟地址空间中驻留在物理内存中的页面的集合
意外本地过程调用端口	当进程所属的线程出现意外时, 进程管理器会通过本地过程调用发送意外信息到该进程, 通过该端口接受意外信息进行相应的处理
调试本地过程调用端口	当进程所属的线程触发调试事件时, 进程管理器会通过本地过程调用发送调试消息到该进程, 通过该端口接受调试消息进行相应的处理

续表

执行进程块的数据项	功 能
访问安全描述	对访问该进程的安全设置
对象句柄表	指向该进程所有对象句柄
Windows 子系统进程信息	Windows 子系统调用该进程所需要的进程信息
核心进程块	它包含了 Windows 内核调度该进程的所属线程所需要的基本信息, 如: 分配给该进程的处理器时间、时间片大小、核心栈信息、进程基准优先级以及进程状态等
进程环境块	它驻留在进程地址空间中, 提供映像调入器、堆管理器和其他运行在用户态的动态链接库所需要的进程信息, 如程序映像的基地址、用户栈信息和线程的局部存储空间

7.2.4Windows 线程的结构

表 7.2 Windows 线程的数据结构

内 容	功 能
线程时间	线程创建和结束时间
所属进程标志	所属进程的标志
起始地址	线程起始例程的地址
访问安全控制	线程级别的访问安全控制信息
局部过程调用信息	线程处理局部过程调用的消息标志以及处理消息的地址
I/O 信息	等待处理的 I/O 请求包的列表
核心线程块	存储系统进行线程调度和同步的线程信息,如:该线程的可用执行时间、核心栈的地址、指向系统服务列表的指针、线程环境块的指针以及与处理器调度相关的信息;如调度优先级、时间配额、空闲处理例程
线程环境块	驻留在进程地址空间,存储用于映像调入器和 Windows 动态链接库所使用的线程上下文信息,如:线程的唯一标志、用户栈的地址以及指向所属进程的进程环境块

7.2.5Windows 进程和线程的创建

1. 进程的创建过程

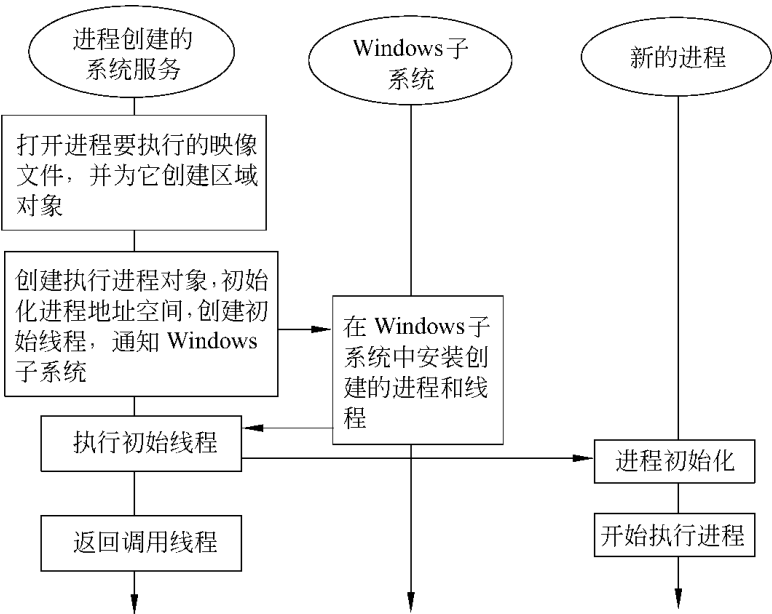


图 7.2 Windows 创建进程的过程

## 2. Windows 线程的创建过程

## 7.3 Windows 处理器调度机制

### 7.3.1 调度优先级

36	15	12	10	8	6
35	14	11	9	7	5
24	13	10	8	6	4
23	12	9	7	5	3
22	11	8	6	4	2
实时	高	高于一般	一般	低于一般	空闲

图 7.3 应用优先级别和系统的优先级别的对应关系

### 7.3.2 线程状态

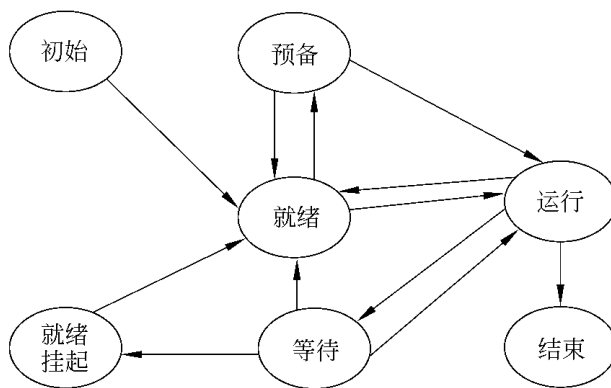


图 7.4 Windows 线程状态转换图

表 7.3 线程状态的说明

状 态	含 义
就绪	表示一个线程已经准备就绪,等待运行。调度器查找线程库中处于就绪状态的线程,来决定下一个运行的线程
预备	表示一个线程已经被选择作为下一个运行的线程,如果条件满足,调度器会将上下文环境切换到该线程。但处于预备状态的线程也可能会被转换到就绪状态继续等待

续表

状 态	含 义
运行	当调度器将上下文环境切换到一个进程,该线程就处于运行状态。当分配给线程的时间配额用完,或有更高优先级的线程抢占 CPU,它会让出处理器
等待	当一个线程需要等待必要的系统资源时,它会转入等待状态,直到系统资源就绪
就绪挂起	如果一个线程已经准备就绪,但运行它所需要的核心都在外存,它就进入就绪挂起状态,等待核心栈调入内存
终止	当一个线程完成执行后,它就进入了终止状态,对象管理器释放相应的执行进程块资源
已初始化	这是一个线程刚被创建时的内部状态

7.3.3 线程调度机制

1. 调度数据库

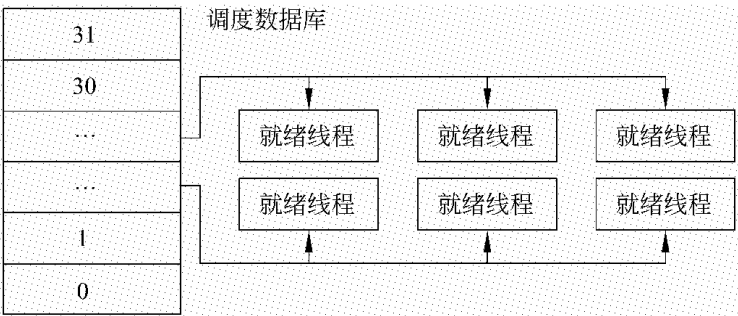


图 7.5 调度数据库的不同优先级就绪线程队列

2. 时间配额
3. 调度算法
4. 上下文切换

## 7.4 Windows 的内存管理

### 7.4.1 内存管理器

### 7.4.2 内存管理的机制

1. 页
2. 共享内存

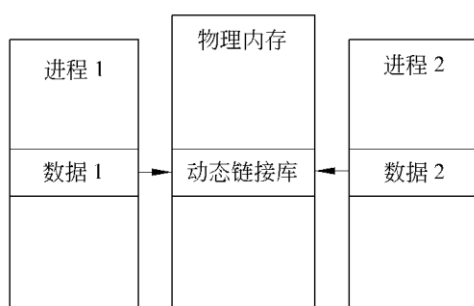


图 7.6 共享内存机制

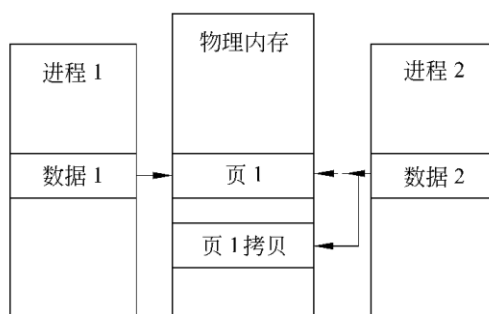


图 7.7 可写入共享内存机制

- 3. 堆管理
- 4. 系统内存池

7.5 虚拟地址空间

7.5.1 虚拟地址空间布局

0x0000,0000	进程私有地址空间 (应用代码, 全局变量, 线程堆)
0x8000,0000	内核及执行体、硬件抽象层、引导驱动
0xC000,0000	进程页表和工作集
0xC080,0000	系统缓存、系统内存池
0xFFFF,FFFF	

图 7.8 32 位 Windows 的虚拟地址空间布局

表 7.4 基于 32 位 x86 体系结构的 Windows 的虚拟地址布局

地 址 范 围	大 小	功 能
0x0000,0000 ~ 0x7FFF,FFFF	2GB	进程的私有地址空间(程序代码、全局变量、线程栈等)
0x8000,0000 ~ 0x9FF,FFFF	512MB	系统内核(NTLDR, HAL)和引导驱动
0xA000,0000 ~ 0xA2FF,FFFF	48MB	系统映射视图或会话空间
0xA300,0000 ~ 0xA3FF,FFFF	16MB	终端服务的系统映射视图
0xA400,0000 ~ 0xBFFF,FFFF	448MB	附加系统页表入口或附加系统高速缓存
0xC000,0000 ~ 0xC03F,FFFF	4MB	进程页表
0xC040,0000 ~ 0xC07F,FFFF	4MB	工作集链表
0xC080,0000 ~ 0xC0BF,FFFF	4MB	未使用
0xC0C0,0000 ~ 0xC0FF,FFFF	4MB	系统工作集链表
0xC100,0000 ~ 0xE0FF,FFFF	512MB	系统高速缓存
0xE100,0000 ~ 0xEAFF,FFFF	160MB	分页缓冲池
0xEB00,0000 ~ 0xFFBD,FFFF	331MB	系统页表入口和非分页缓冲池
0xFFBE,0000 ~ 0xFFFF,FFFF	4MB	故障处理和硬件抽象层(HAL)结构

7.5.2 虚拟地址转换

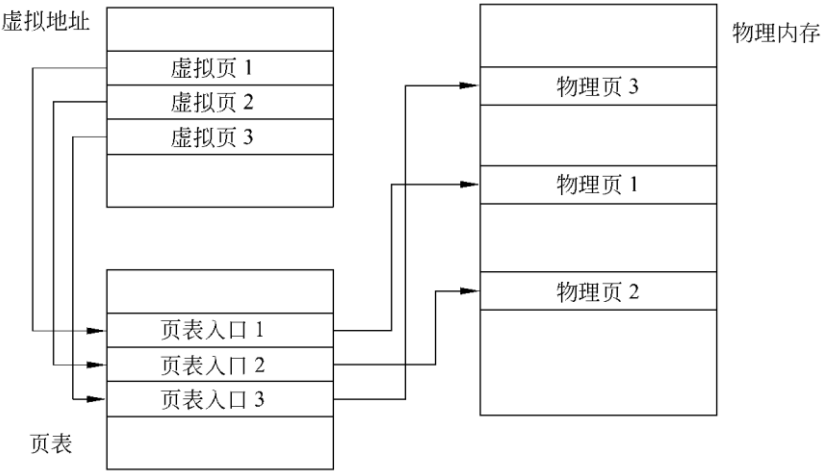


图 7.9 虚拟地址转换

31	22	21	12	11	0
页目录索引		页表索引		字节索引	

图 7.10 虚拟地址的页索引结构

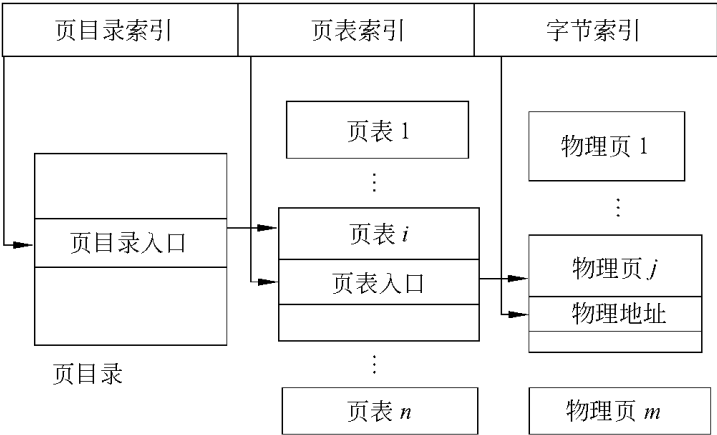


图 7.11 虚拟地址转换过程

31	12	11	0
物理页框号		页的状态位	

图 7.12 页表入口的结构

表 7.5 页表中的状态位及其含义

状 态 位	含 义	状 态 位	含 义
Accessed	该页是否已被读过	Owner	标明该页是否可以在用户态下访问
Cashed Disabled	该页是否不能被缓存	Valid	标明该页是否驻留在物理内存中
Dirty	该页是否已被写过	Write Through	是否跳过些缓存将该页面实时写入磁盘
Global	该转换是否适用所有的进程	Write	该页面是否可写
Large Page	该页是否为大页(4MB)		

7.6 页面调度

7.6.1 缺页处理

表 7.6 缺页处理的各种情况

缺 页 原 因	处 理 方 式
被访问的页不在物理内存中,而在磁盘上的页文件或映射文件中	分配一个物理页从磁盘上读入该页,并将它记入工作集
该页被挂起或在修改页列表中	将该页转移到进程或系统的工作集
访问预留页或超出了分配的地址空间	访问错误
在用户态访问只能在核心态访问的页	访问错误
写入只读的页	访问错误
写入“先拷贝后写入”页	拷贝该页并将新的页作为进程的私有页,同时更新相应的映射和工作集
执行页中标明不能被执行的代码	访问错误



## 7.6.2 工作集及页面调度策略

## 7.6.3 页框号和物理内存管理

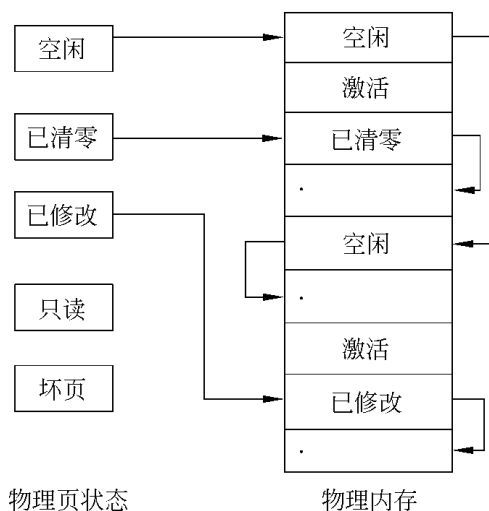


图 7.13 页框号数据库及状态链表

表 7.7 物理页的状态及其说明

状 态	说 明
激活	该页属于某一工作集,或者属于系统的常驻内存页
过渡	这是一个临时状态标志,表明它既不属于一个工作集也不在分页列表上,而表明改页正在被处理的过程中
挂起	该页被从一个工作集中移出了,对它的修改已经写入到了磁盘,页表入口还是指向了该物理页,但对访问设置了挂起标志
已修改	该页被从一个工作集中移出了,但对它的修改还没有被写入到了磁盘,页表入口还是指向了该物理页,完成写入磁盘后才可以投入使用
空闲	该页没有任何被修改的数据,但还没有被清零,出于安全性的考虑还不能被用户进程使用

续表

状 态	说 明
已清零	该页已经被清零,能被用户进程使用
只读	该页位于只读内存
坏页	该页物理损坏,不能被使用

## 本章小结

### 习题

7.1 简述 Windows 核心态和用户态的区别。

7.2 Windows 操作系统有哪些系统服务构成，简述它们的功能。

7.3 描述 Windows 的进程和线程的概念，解释它们的区别和联系。

7.4 在 Windows 进程结构中，执行进程块和进程环境块分别起到什么作用？

7.5 简述 Windows 的线程结构，以及它在内存中的驻留机制。

7.6 通过 Windows 任务管理器观察和分析系统中的进程。

7.7 用 C 语言编写程序利用 `CreateProcess` 和 `CreateThread` 函数创建一个 Windows 进程和两个线程。

7.8 简述哪些系统服务参与了 Windows 创建进程的过程，它们分别起到什么作用？

7.9 在 Windows 处理器调度的过程，线程的哪些状态可以转换到就绪状态，它们在什么条件下转换到该状态？

7.10 简述 32 位的 Windows 操作系统的虚拟地址空间的布局。

7.11 简述虚拟地址到物理内存地址的转换过程，其中页表起到什么作用？

7.12 简述工作集在 Windows 内存管理中的作用和工作过程。

# 第 8 章 文件系统

## 8.1 文件系统的概念

### 1. 文件系统的引入

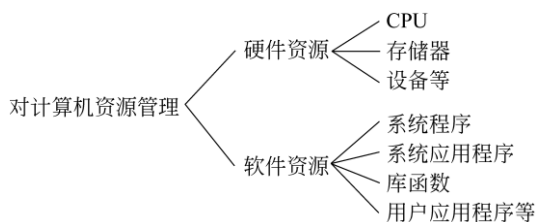


图 8.1 操作系统的软硬件管理

### 2. 文件与文件系统的概念

### 3. 文件的分类

## 8.2 文件的逻辑结构与存取方法

### 8.2.1 逻辑结构

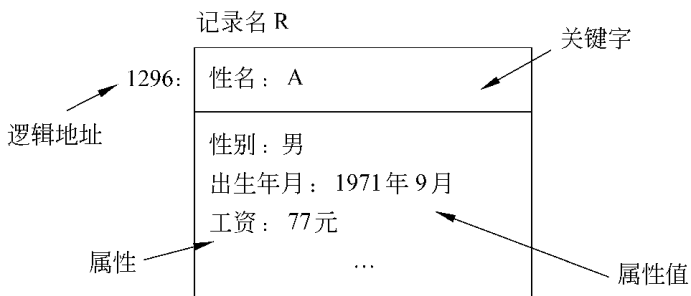


图 8.2 记录组成例

- 1. 连续结构
- 2. 多重结构

$$\begin{matrix} & R_1 & R_2 & \cdots & R_n \\ \begin{matrix} K_1 \\ K_2 \\ \vdots \\ K_m \end{matrix} & \begin{bmatrix} 1 & 0 & \cdots & 1 \\ 0 & \cdots & & \\ \cdots & \cdots & & \\ 1 & 1 & \cdots & 0 \end{bmatrix} \end{matrix}$$

图 8.3 文件的记录名和关键字构成的行列式

- 3. 转置结构

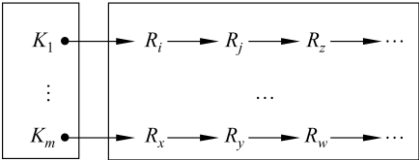


图 8.4 文件的多重结构

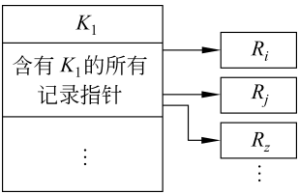


图 8.5 文件的转置结构

#### 4. 顺序结构

### 8.2.2 存取方法

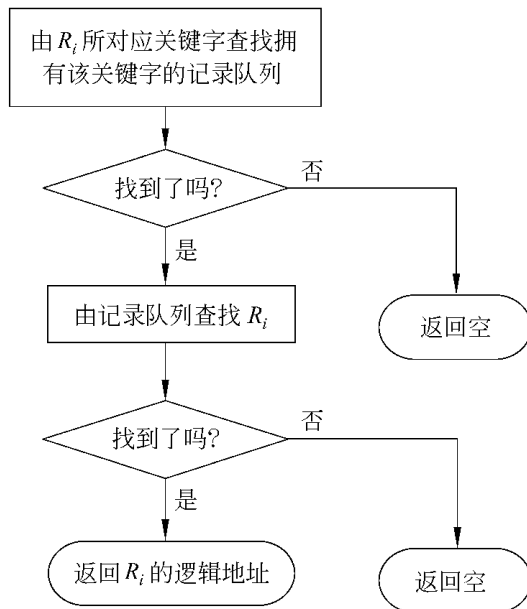


图 8.6 记录 Ri 的搜索过程

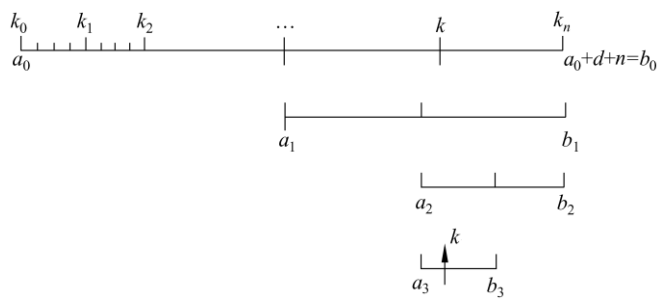


图 8.7 二分搜索法的搜索过程

# 8.3 文件的物理结构与存储设备

## 8.3.1 文件的物理结构

### 1. 连续文件

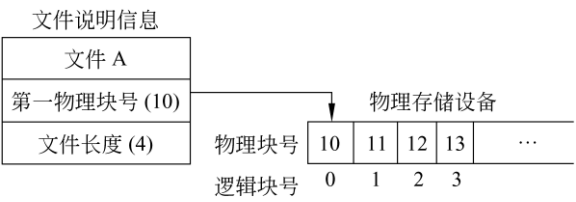


图 8.8 连续文件结构

### 2. 串联文件

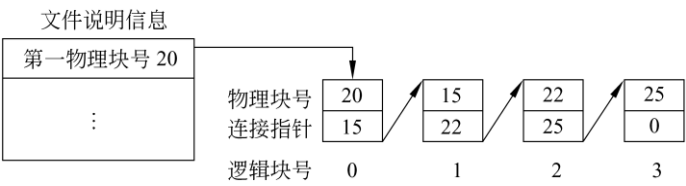


图 8.9 串联文件的物理结构

### 3. 索引文件

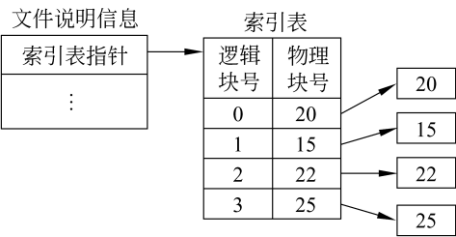


图 8.10 索引文件示意图

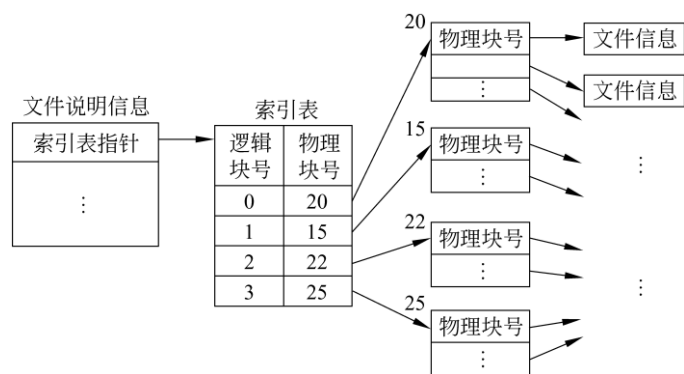


图 8.11 多重索引结构

8.3.2 文件存储设备

1. 顺序存取设备

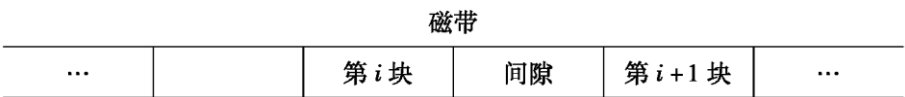


图 8.12 磁带的结构

2. 直接存取设备

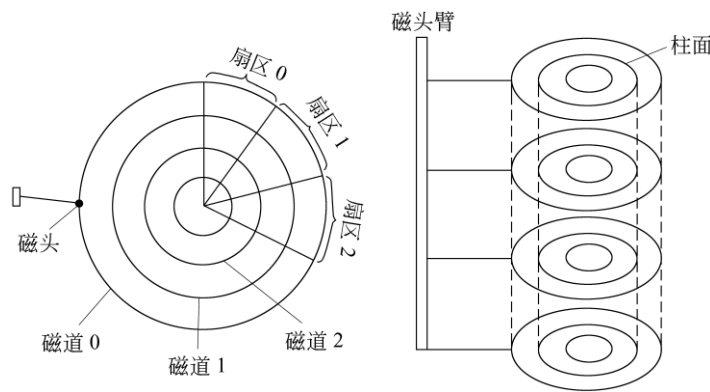


图 8.13 磁盘的结构

## 8.4 文件存储空间管理

### 1. 空闲文件目录

### 2. 空闲块链

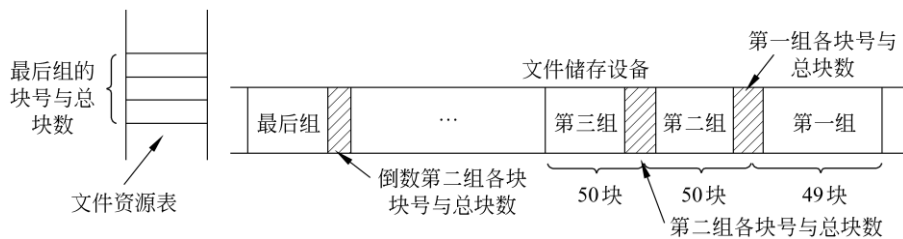


图 8.14 成组链法的组织

### 3. 位示图

## 8.5 文件目录管理

### 8.5.1 文件的组成

### 8.5.2 文件目录

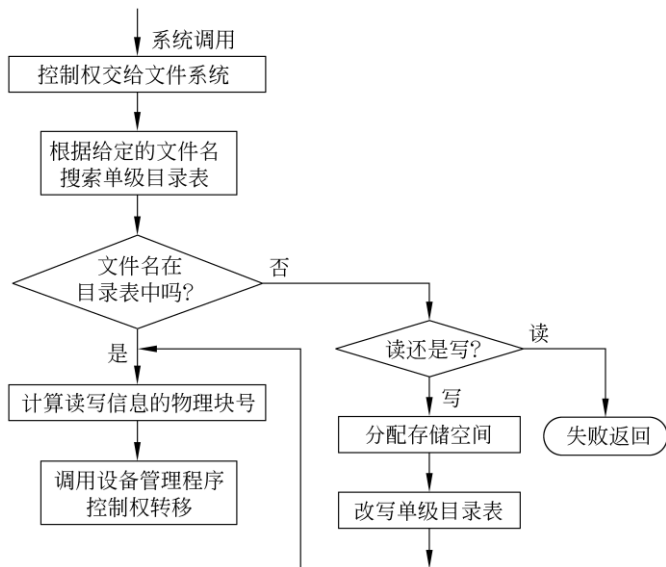




图 8.15 单级目录的读写处理过程

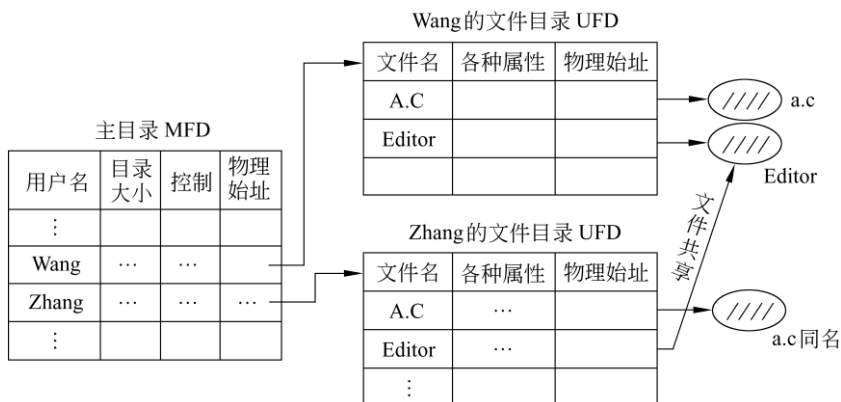


图 8.16 二级目录结构

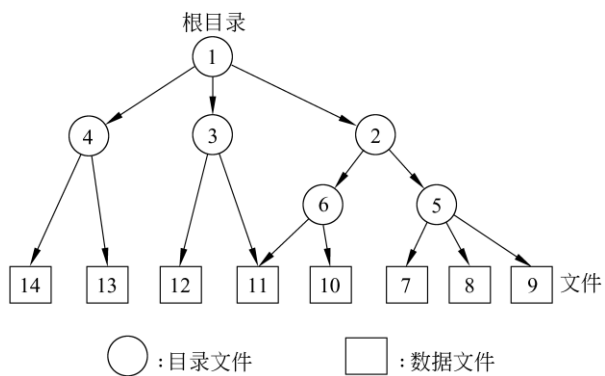


图 8.17 文件系统的树形结构

### 8.5.3 便于共享的文件目录

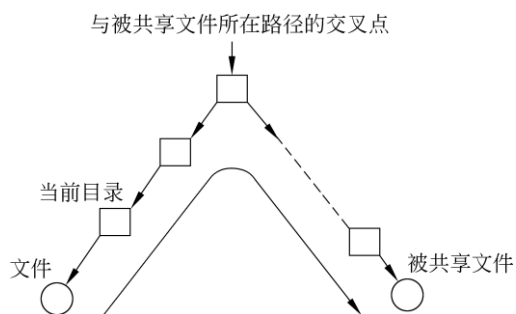


图 8.18 绕道法

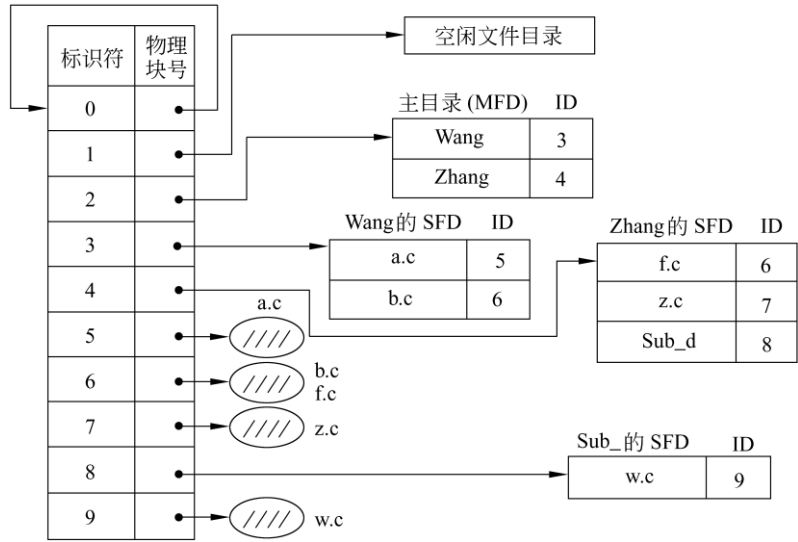


图 8.19 采用基本文件目录的多级目录结构

8.5.4 目录管理

8.6 文件存取控制

1. 存取控制矩阵

用 户	Wang	lee	Zhang	...	...
存 取 数					
文 件 名					
A. C.	RWE	E	RWE		
B. C	RW	R	RWE		
D. C	R	W	WE		
E. C	R	W	RW		

图 8.20 存取控制矩阵

## 2. 存取控制表

文件名 用户	A. C
Zhang	RWE
A 组	RE
B 组	E
Wang	RWE
其他	None

图 8.21 存取控制表

## 3. 口令方式

## 4. 密码方式

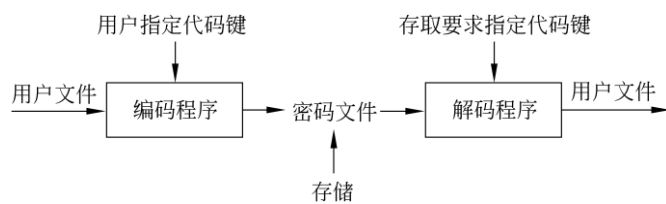


图 8.22 加密解密过程

## 8.7 文件的使用

## 8.8 文件系统的层次模型

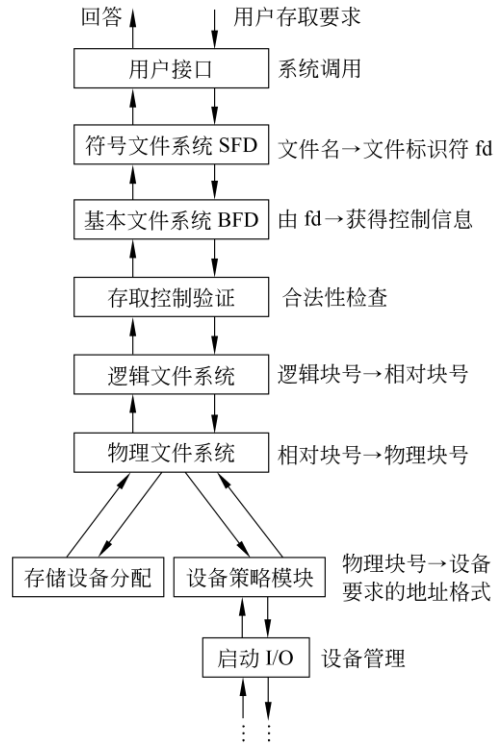


图 8.23 文件系统的层次模型

## 本章小结

## 习题

- 8.1 什么是文件、文件系统？文件系统有哪些功能？
- 8.2 文件一般根据什么分类？可以分为哪几类？
- 8.3 什么是文件的逻辑结构？什么是记录？
- 8.4 设文件的结构为多重结构和转置结构的组合，且定义函数  $\text{decode}(K, x)$  和  $\text{retrive}(K, x)$  如下：

函数  $\text{decode}(K, x)$  为关键字  $K$  的搜索函数。其中  $K$  为待搜索关键字， $x$  为顺序指针。当被搜索文件为多重结构时，指向关键字  $K$  的指针只有一个，即  $e(K)$ ，此时  $x = \text{nil}$ ，且  $\text{decode}(K, x)$  返回指针  $e(K)$ 。当被搜索文件为转置结构时，一般指向关键字  $K$  的指针有  $n$  个，即  $e_1(K), \dots, e_n(K)$ ， $n$  为包含关键字  $K$  的记录个数。此时，若  $x = \text{nil}$ ， $\text{decode}(K, x)$  返回  $e_1(K)$ ；若  $x = e_n(K)$ ，则  $\text{decode}(K, x)$  返回  $\text{nil}$ ；否则，若  $x = e_i(K)$ ， $(1 \leq i < n)$ ；则  $\text{decode}(K, x)$  返回  $e_{i+1}(K)$ 。

函数  $\text{retrive}(K, x)$  是记录搜索函数。其中  $K$  为指向关键字的指针， $x$  为记录顺序

指针。如果  $x = \text{nil}$  则  $\text{retrive}(K, x)$  返回被搜索记录队列的第一个记录的逻辑地址;若  $x$  等于该记录队列的最后一个记录的话,则  $\text{retrive}(K, x)$  返回  $\text{nil}$ ;否则,  $\text{retrive}(K, x)$  返回  $x$  的下一个记录的逻辑地址。试问:

- ① 如果  $\text{decode}(K, x)$  采用线性搜索法,怎样描述  $\text{decode}(K, x)$ ?
  - ② 如果  $\text{retrive}(K, x)$  采用二分搜索法,应怎样排列记录队列,和怎样描述函数  $\text{retrive}(K, x)$ ?
  - ③ 设函数  $\text{search}(k)$  给出含有关键字  $k$  的所有记录的逻辑地址,试描述  $\text{search}(k)$ 。
- 8.5 设散列函数  $h(n) = (676 \times l_1 + 26 \times l_2 + l_3) \pmod{t}$ ,  $t = 11$ ,  $l_i$  为关键字  $n$  的第  $i$  个英文字母序号。关键字表长为 11,关键字为英文字母。先按线性散列法,再按平方散列法计算将任意 7 个关键字放入链表中所用的计算次数。这里令  $a = 2$ ,  $c = 1$ 。
- 8.6 设关键字表按英文字母顺序排列,且两关键字项之间的距离为  $d$ ,写出  $n = 3$  的三分搜索算法。
- 8.7 文件的物理结构有哪几种?为什么说串联文件结构不适于随机存取?
- 8.8 设索引表长度为 13,其中  $0 \sim 9$  项为直接寻址方式,后 3 项为间接寻址方式,试描述出给定文件长度  $n$ (块数)后的索引方式寻址算法。
- 8.9 常用的文件存储设备的管理方法有哪些?试述主要优缺点。
- 8.10 试述成组链法的基本原理,并描述成组链法的分配与释放过程。
- 8.11 什么是文件目录?文件目录中包含哪些信息?
- 8.12 二级目录和多级目录的好处是什么?符号文件目录表和基本文件目录表是二级目录吗?
- 8.13 文件存取控制方式有哪几种?试比较它们各自的优缺点。
- 8.14 设文件 SQRT 由连续结构的定长记录组成,每个记录的长度为 500 字节,每个物理块长 1000 字节,且物理结构也为连续结构和采用直接存取方式;试按照图 8.23 所示文件系统模型,写出系统调用  $\text{Read}(\text{SQRT}, 5, 15000)$  的各层执行结果。其中, SQRT 为文件名,5 为记录号,15000 为内存地址。

## 第9章 设备管理

### 9.1 引言

#### 9.1.1 设备的类别

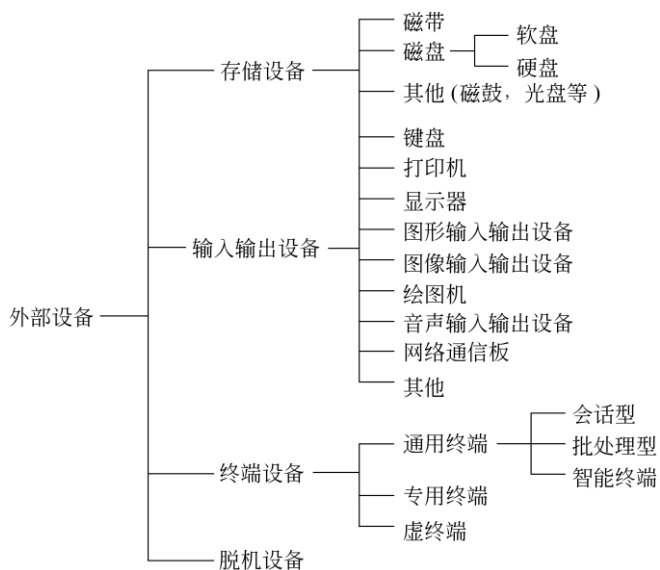


图 9.1 按使用特性对外部设备的分类

## 9.1.2 设备管理的功能和任务

## 9.2 数据传送控制方式

### 9.2.1 程序直接控制方式

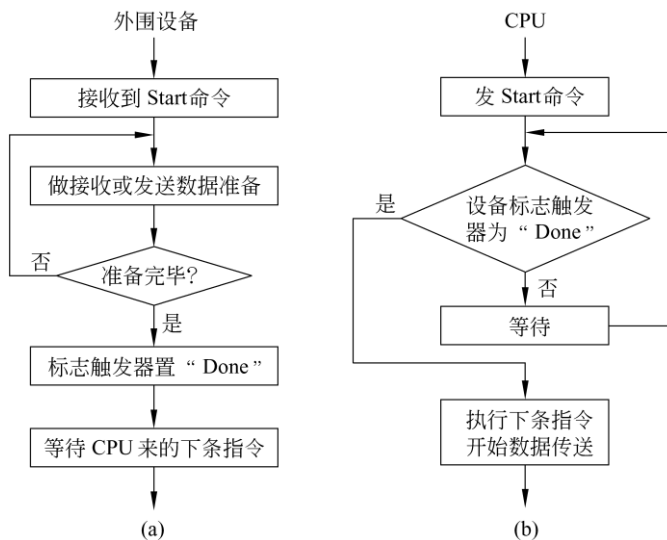


图 9.2 程序直接控制方式

### 9.2.2 中断方式

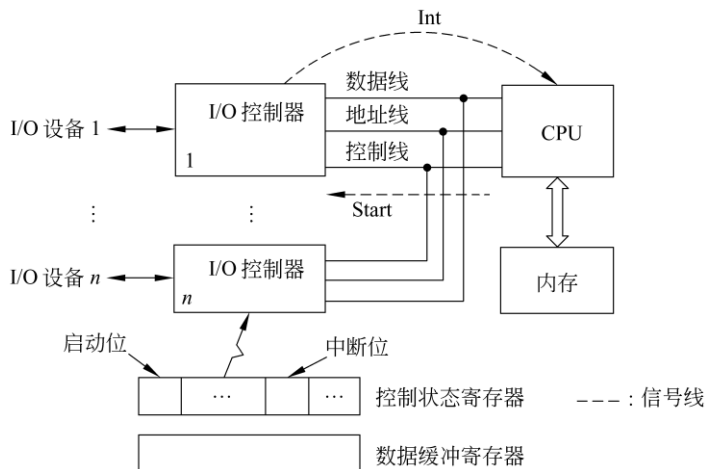




图 9.3 中断控制方式的传送结构

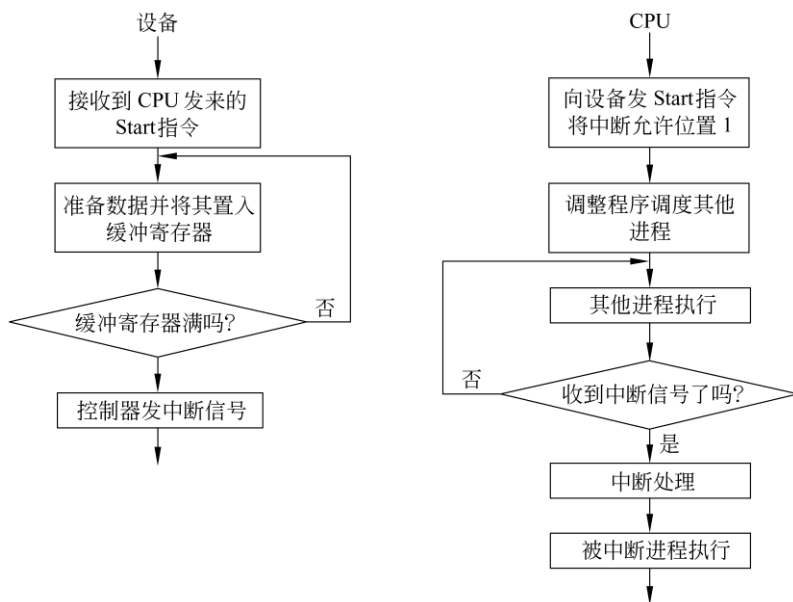


图 9.4 中断控制方式的处理过程

### 9.2.3 DMA 方式

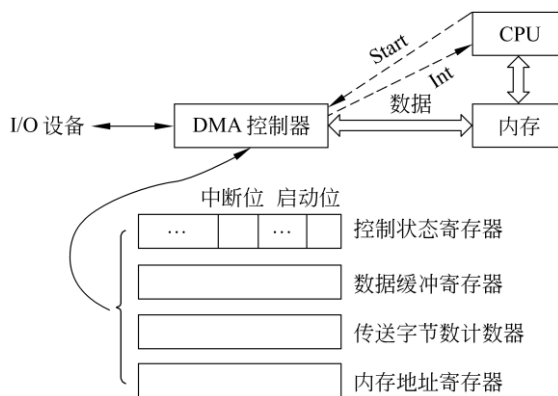


图 9.5 DMA 方式的传送结构

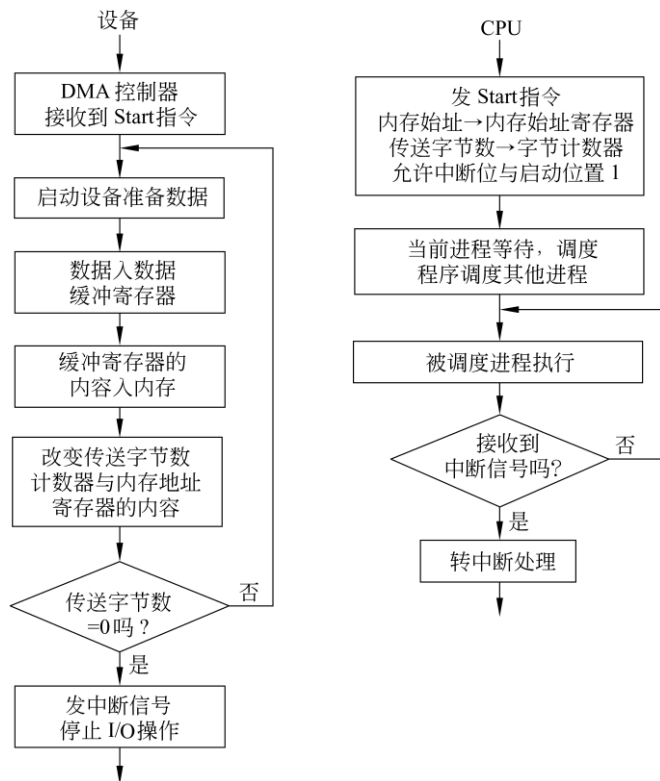


图 9.6 DMA 方式的数据传送处理过程

## 9.2.4 通道控制方式

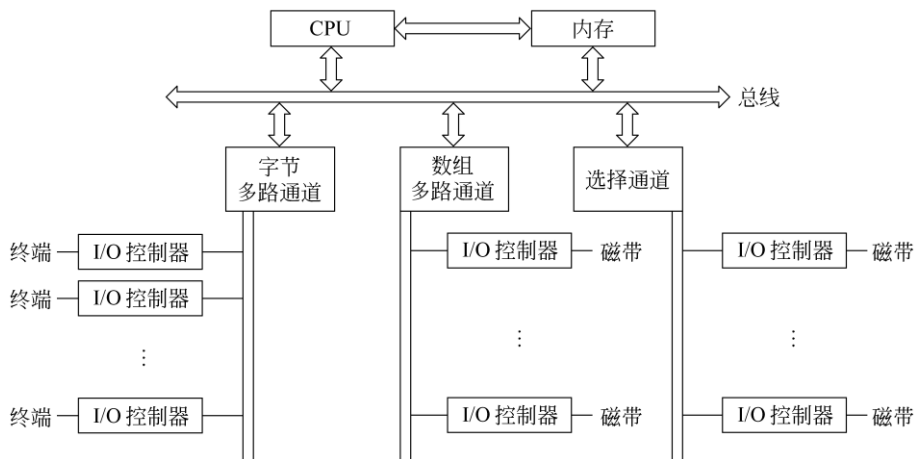


图 9.7 通道方式的数据传送结构

Channel control procedure:

repeat

IR ← M[ pc ]

pc ← pc + 1

execute( IR )

if require accessing with I/O Device

then Command( I/O operation, Address of I/O device, channel ) fi

if I/O Do ne Interrupt

then Call Interrupt processing control fi

until machine halt

Interrupt processing control procedure

...

## 9.3 中断技术

### 9.3.1 中断的基本概念

### 9.3.2 中断的分类与优先级

### 9.3.3 软中断

### 9.3.4 中断处理过程

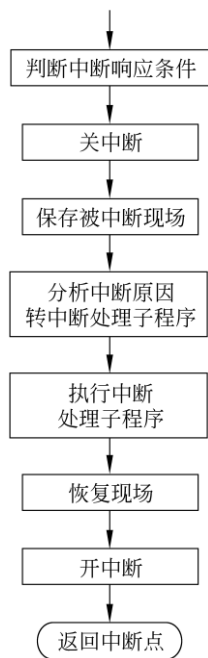


图 9.8 中断处理过程

```

I/O Interrupt processing control:
begin
    unusable I/O Interrupt flag
    save status of interrupt program
    If Input Device i Ready
    then Call input Device i Control fi
    if Output Device i Ready
    then Call Output Device i Control fi
    if Data Deliver Done
    then Call Data Deliver Done Control fi
    restore CPU status
    reset I/O Interrupt flag
end
Input Device i Control: ...
Output Device i Control: ...
Data Deliver Done Control: ...
    
```

## 9.4 缓冲技术

### 9.4.1 缓冲的引入

### 9.4.2 缓冲的种类

### 9.4.3 缓冲池的管理

#### 1. 缓冲池的结构

设备号
数据块号
缓冲器号
互斥标识位
连接指针

图 9.9 缓冲首部

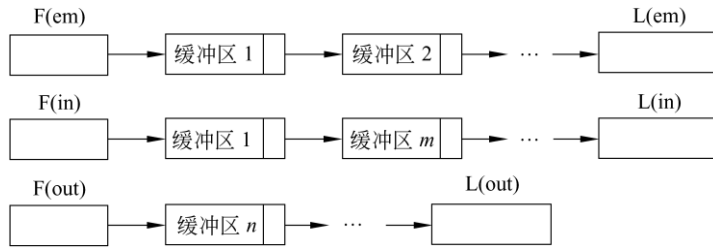


图 9.10 缓冲区队列

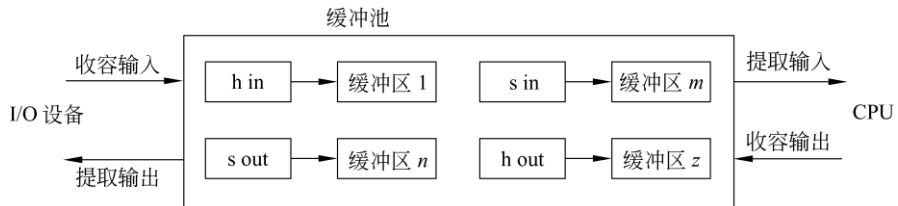


图 9.11 缓冲池的工作缓冲区

## 2. 缓冲池管理

```

get_buf( type, number ) :
begin
    P( RS( type ) )
    P( S( type ) )
    Pointer of buffer( number ) = take_buf( type, number )
    V( S( type ) )
end

put_buf( type, number ) :
begin
    P( S( type ) )
    add_buf( type, number )
    V( S( type ) )
    V( RS( type ) )
end
  
```

## 9.5 设备分配

### 9.5.1 设备分配用数据结构

1. 设备控制表
2. 系统设备表
3. 控制器表
4. 通道控制表

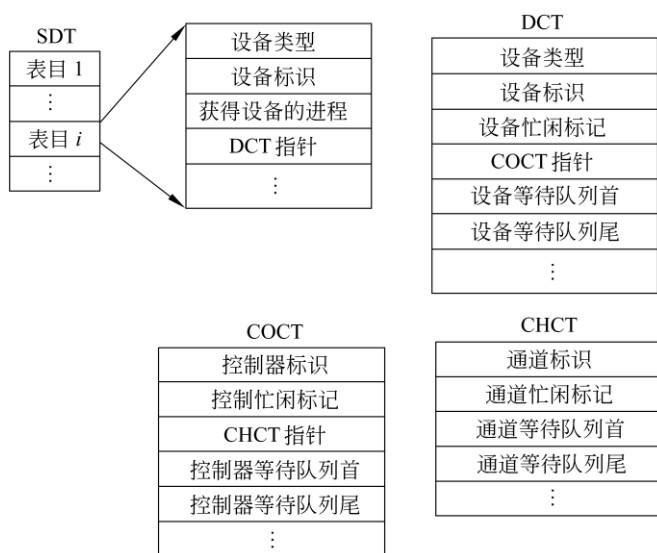


图 9.12 数据结构表

## 9.5.2 设备分配的原则

### 1. 设备分配原则

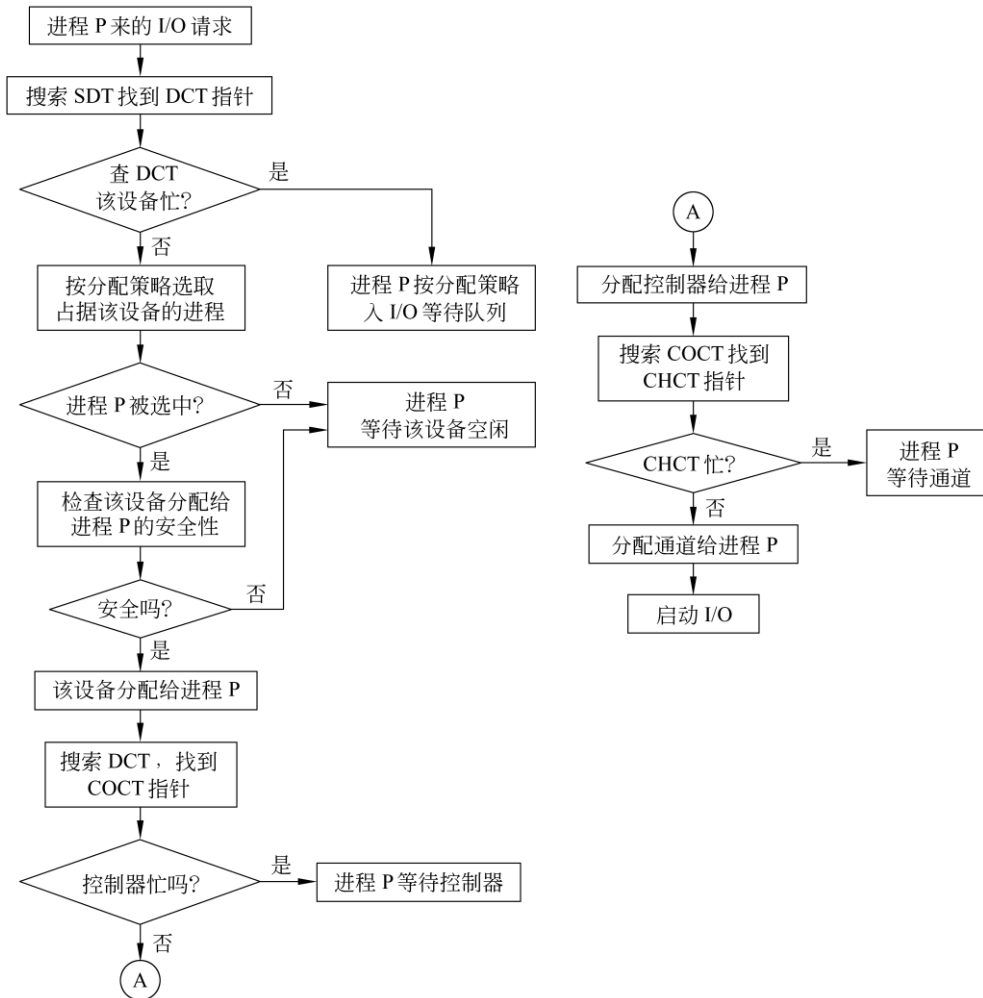


图 9.13 设备分配流程图



## 2. 设备分配策略

### 9.5.3 设备分配算法

## 9.6 I/O 进程控制

### 9.6.1 I/O 控制的引入

### 9.6.2 I/O 控制的功能

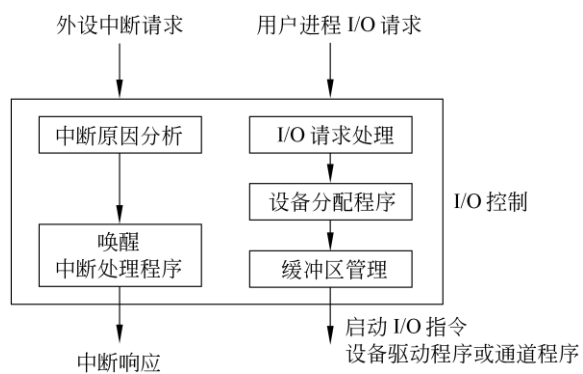


图 9.14 I/O 控制的功能

### 9.6.3 I/O 控制的实现

## 9.7 设备驱动程序

## 本章小结

## 习题

9.1 设备管理的目标和功能是什么？

9.2 数据传送控制方式有哪几种？试比较它们各自的优缺点。

9.3 什么是通道？试画出通道控制方式时的 CPU、通道和设备的工作流程图。

9.4 什么叫中断？什么叫中断处理？什么叫中断响应？

9.5 什么叫关中断？什么叫开中断？什么叫中断屏蔽？

9.6 什么是陷阱？什么是软中断？试述中断，陷阱和软中断之间异同。

9.7 描述中断控制方式时的 CPU 动作过程。

9.8 什么是缓冲？为什么要引入缓冲？

9.9 设在对缓冲队列 em, in 和 out 进行管理时，采用最近最少使用算法存取缓冲区，即在一个缓冲区分配给进程之后，只要不是所有其他的缓冲区都在更近的时间内被使用过了，则该缓冲区不再分配出去。试描述过程 `take_buf(type, number)` 和 `add_buf(type, number)`。

9.10 试述对缓冲队列 em, in 和 out 采用最近最少使用算法对改善 I/O 操作性能有什么好处？

9.11 用于设备分配的数据结构有哪些？它们之间的关系是什么？

9.12 设计一个设备分配的安全检查程序，以保证把某台设备分配给某进程时不会出现死锁。

9.13 什么是 I/O 控制？它的主要任务是什么？

9.14 I/O 控制可用哪几种方式实现？各有什么优缺点？

9.15 设备驱动程序是什么？为什么要有设备驱动程序？用户进程怎样使用驱动程序？



## 第 10 章 Linux 文件系统

### 10.1 Linux 文件系统的特点与文件类别

#### 10.1.1 特点

#### 10.1.2 文件类型

### 10.2 Linux 的虚拟文件系统

#### 10.2.1 虚拟文件系统 VFS 框架

#### 10.2.2 Linux 虚拟文件系统的数据结构

##### 1. VFS 的超级块 `super_block`

```

struct super_block {
    struct list_head    s_list;        /* 指向超级块链表的指针 */
    dev_t               s_dev;         /* 设备标识符 */
    unsigned long       s_blocksize;   /* 以字节为单位的块大小 */
    unsigned char       s_blocksize_bits; /* 以位为单位的块大小 */
    unsigned char       s_dirt;        /* 修改(脏)标志 */
    unsigned long long  s_maxbytes;    /* 文件最大尺寸 */
    struct file_system_type * s_type;   /* 文件系统类型 */
    struct super_operations * s_op;     /* 超级块方法 */
    struct dquot_operations * dq_op;    /* 磁盘限额方法 */
    struct quotactl_ops * s_qcop;       /* 磁盘限额管理方法 */
    struct export_operations * s_export_op; /* NFS 的输出方法 */
    unsigned long       s_flags;       /* 挂装标志 */
    unsigned long       s_magic;       /* 文件系统的魔数 */
    struct dentry       * s_root;      /* 文件系统挂装目录的目录项结构 */
    struct rw_semaphore s_umount;      /* umount 使用的信号 */
    struct mutex        s_lock;        /* 锁标志 */
    int                 s_count;       /* 参考计数器 */
    ...
    struct list_head    s_inodes;      /* 所有索引节点的链表 */
    struct list_head    s_dirty;       /* 所有已修改(脏)的索引节点的链表 */
    struct list_head    s_io;         /* 所有等待写回磁盘的索引节点链表 */
    struct hlist_head   s_anon;        /* 用于处理 NFS 输出的匿名目录项链表 */
    struct list_head    s_files;       /* 所有文件对象链表 */
    struct block_device * s_bdev;      /* 指向块设备驱动描述符的指针 */
    ...

```

```

...
    char                s_id[32];        /* 包含这个超级块的块设备名 */
    void                * s_fs_info;      /* 指向特定文件系统超级块信息数据结构的
                                           指针 */
...
...
}

struct super_operations {
    struct inode * ( * alloc_inode)(struct super_block * sb);
    void ( * destroy_inode)(struct inode * );
    void ( * read_inode)(struct inode * );
    void ( * dirty_inode)(struct inode * );
    int ( * write_inode)(struct inode * , int);
    void ( * put_inode)(struct inode * );
    void ( * drop_inode)(struct inode * );
    void ( * delete_inode)(struct inode * );
    void ( * put_super)(struct super_block * );
    void ( * write_super)(struct super_block * );
    int ( * sync_fs)(struct super_block * sb, int wait);
    void ( * write_super_lockfs)(struct super_block * );
    void ( * unlockfs)(struct super_block * );
    int ( * statfs)(struct super_block * , struct kstatfs * );
    int ( * remount_fs)(struct super_block * , int * , char * );
    void ( * clear_inode)(struct inode * );
    void ( * umount_begin)(struct super_block * );
    int ( * show_options)(struct seq_file * , struct vfsmount * );
    ssize_t ( * quota_read)(struct super_block * , int, char * , size_t, loff_t);
    ssize_t ( * quota_write)(struct super_block * , int, const char * , size_t, loff_t);
};

```

## 2. 索引节点 inode

```

struct inode {
    struct hlist_node    i_hash;        /* 指向散列链表的指针 */
    struct list_head     i_list;        /* 指向描述索引节点当前状态的链表的指针 */
    struct list_head     i_sb_list;    /* 指向超级块的索引节点链表的指针 */
    struct list_head     i_dentry;     /* 指向目录项链表的指针 */
    unsigned long        i_ino;        /* 索引节点号 */
    atomic_t             i_count;      /* 引用计数器 */
    umode_t              i_mode;       /* 文件类型与访问权限 */
    unsigned int         i_nlink;      /* 硬链接数目 */
    uid_t                i_uid;        /* 所有者标识符 */
    gid_t                i_gid;        /* 组标识符 */
    dev_t                i_rdev;       /* 真实设备标识符 */
    loff_t               i_size;       /* 文件的字节数 */
    struct timespec      i_atime;      /* 上次访问文件的时间 */
    struct timespec      i_mtime;      /* 上次修改文件的时间 */
    struct timespec      i_ctime;      /* 上次修改索引节点的时间 */
    unsigned int         i_blkbits;    /* 每块的位数 */
    unsigned long        i_blksize;    /* 每块的字节数 */
    unsigned long        i_version;    /* 版本号 */
    unsigned long        i_blocks;     /* 文件的块数 */
    unsigned short       i_bytes;      /* 文件占用的最后一块的字节数 */
    spinlock_t           i_lock;       /* 保护索引节点的某些域的自旋锁 */
    struct mutex          i_mutex;     /* 互斥量 */
    struct rw_semaphore  i_alloc_sem;  /* 读写保护信号量,用于在直接 I/O 访问文件时避免竞争关系 */

    struct inode_operations *i_op;     /* 索引节点方法 */
    struct file_operations *i_fop;     /* 默认的文件操作 */
    struct super_block    *i_sb;       /* 指向超级块对象的指针 */
    struct file_lock      *i_flock;    /* 指向文件锁链表的指针 */
    struct address_space  *i_mapping;   /* 指向地址空间对象的指针 */
    struct address_space  i_data;      /* 文件的地址空间对象 */

#ifdef CONFIG_QUOTA
    struct dqquot         *i_dquot[ MAXQUOTAS ]; /* 索引节点的磁盘限额 */
#endif

    ...

    ...

    unsigned long        i_state;      /* 索引节点状态标志 */
    unsigned long        dirtied_when; /* 索引节点修改时间 */
    unsigned int         i_flags;      /* 文件系统挂装标志 */

```

```

        atomic_t          i_writecount;    /* 写进程的使用计数器 */
        void              * i_security;    /* 指向索引节点安全结构的指针 */
        union {
            void           * generic_ip;    /* 指向文件系统具体数据的指针 */
        } u;
    ...
    ...
};

struct inode_operations {
    int ( * create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * ( * lookup) (struct inode *,struct dentry *, struct nameidata *);

    int ( * link) (struct dentry *,struct inode *,struct dentry *);
    int ( * unlink) (struct inode *,struct dentry *);
    int ( * symlink) (struct inode *,struct dentry *,const char *);
    int ( * mkdir) (struct inode *,struct dentry *,int);
    int ( * rmdir) (struct inode *,struct dentry *);
    int ( * mknod) (struct inode *,struct dentry *,int,dev_t);
    int ( * rename) (struct inode *, struct dentry *,struct inode *, struct dentry *);
    int ( * readlink) (struct dentry *, char _user *,int);
    void * ( * follow_link) (struct dentry *, struct nameidata *);
    void ( * put_link) (struct dentry *, struct nameidata *, void *);
    void ( * truncate) (struct inode *);
    int ( * permission) (struct inode *, int, struct nameidata *);
    int ( * setattr) (struct dentry *, struct iattr *);
    int ( * getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int ( * setxattr) (struct dentry *, const char *,const void *,size_t,int);
    ssize_t ( * getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t ( * listxattr) (struct dentry *, char *, size_t);
    int ( * removexattr) (struct dentry *, const char *);
    void ( * truncate_range)(struct inode *, loff_t, loff_t);
};

```



### 3. 文件 file

```
struct file {
    ...
    struct list_head    f_list;           /* 指向文件对象链表的指针 */
    struct dentry        *f_dentry;       /* 文件对应的目录项对象的指针 */
    struct vfsmount      *f_vfsmnt;       /* 包含这个文件的已挂装的文件系统 */
    struct file_operations *f_op;         /* 指向文件操作表的指针 */
    atomic_t             f_count;         /* 引用计数器 */
    unsigned int         f_flags;         /* 打开文件时指定的标志 */
    mode_t               f_mode;         /* 进程的访问模式 */
    loff_t               f_pos;          /* 当前位移量(文件指针) */
    struct fown_struct    f_owner;        /* 通过信号进行 I/O 事件通知的数据 */
    unsigned int         f_uid, f_gid;    /* 用户的 UID 和 GID */
    ...
};
```

### 4. 目录项 dentry

```
struct dentry {
    atomic_t             d_count;         /* 目录项对象引用计数器 */
    unsigned int         d_flags;         /* 目录项标志 */
    spinlock_t          d_lock;          /* 用于保护 dentry 对象的自旋锁 */
    struct inode         *d_inode;        /* 与文件名关联的索引节点 */
    struct dentry        *d_parent;       /* 父目录的目录项对象 */
    struct               qstr d_name;     /* 文件名 */
    struct               list_head d_lru; /* 用于未使用链表的指针 */
    struct               list_head d_child; /* 用于父目录的目录项对象的链表的指针 */

    struct               list_head d_subdirs; /* 所有子目录的目录项链表 */
    ...
    struct               dentry_operations d_op; /* 目录项方法 */
    struct               super_block *d_sb; /* 文件的超级块对象 */
    void                 *d_fsdata;       /* 依赖于文件系统的数据 */
    ...
    struct               hlist_node d_hash; /* 用于散列表表项的指针 */
    int                  d_mounted;       /* 挂装在这个目录项对象上的文件系统数目 */
    ...
};
```

### 10.2.3 VFS 的系统调用

```
inf = open("/mnt/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test", O_WRONLY|O_CREATE|O_TRUNC, 0600);
do {
    l = read(inf, buf, 4096);
    write(outf, buf, l);
} while(l);
close(outf);
close(inf);
```

mount() umount()	挂装/卸载文件系统
sysfs()	获取文件系统信息
statfs() fstatfs() ustat()	获取文件系统统计信息
chroot()	更改根目录
chdir() fchdir() getcwd()	操纵当前目录
mkdir() rmdir()	创建/删除目录
stat() fstat() lstat() access()	读取文件状态
open() close() creat() umask()	打开/关闭文件
dup() dup2() fcntl()	对文件描述符进行操作
select() poll()	异步 I/O 通告
read() write() readv() writev() sendfile()	进行文件 I/O 操作
readlink() symlink()	对软链接进行操作
chown() fchown() lchown()	更改文件所有者
chmod() fchmod() utime()	更改文件属性
pipe()	进行通信操作

## 10.3 文件系统的注册和挂装

### 10.3.1 文件系统注册

### 10.3.2 已挂装文件系统描述符链表

```
struct vfsmount
{
    struct list_head mnt_hash;        /* 用于散列链表的指针 */
    struct vfsmount *mnt_parent;      /* 父文件系统描述符。本文件系统挂装在其上 */
    struct dentry *mnt_mountpoint;    /* 指向这个文件系统挂装点的目录项 */
    struct dentry *mnt_root;          /* 指向这个文件系统根目录的目录项 */
    struct super_block *mnt_sb;       /* 指向这个文件系统的超级块 */
    struct list_head mnt_mounts;      /* 挂装在这个文件系统上的子文件系统的链表的头 */
    struct list_head mnt_child;       /* 父文件系统的 mnt_mount 链表所使用的指针 */
    atomic_t mnt_count;               /* 引用计数器 */
    int mnt_flags;                    /* 标志 */
    char *mnt_devname;                /* 设备文件名,如/dev/hda1 */
    struct list_head mnt_list;        /* 指向描述符全局链表的指针 */
    ...
};
```

### 10.3.3 挂装根文件系统

### 10.3.4 挂装一般文件系统

### 10.3.5 卸载文件系统

## 10.4 进程与文件系统的联系

### 10.4.1 系统打开文件表

### 10.4.2 用户打开文件表

```
struct files_struct {
    atomic_t count;           /* 共享该表的进程数目 */
    spinlock_t file_lock;    /* 用于表中字段的读写自旋锁 */
    int max_fds;              /* 文件对象的当前最大数目 */
    int max_fdset;            /* 文件描述符的当前最大数目 */
    int next_fd;              /* 已经分配的最大文件描述符号加1 */
    struct file ** fd;        /* 指向文件对象指针数组的指针 */
    ...
    ...
    struct file * fd_array[NR_OPEN_DEFAULT]; /* 文件对象指针的初始化数组 */
};
```

### 10.4.3 进程的当前目录和根目录

## 10.5 ext2 文件系统

### 10.5.1 ext2 文件系统的存储结构

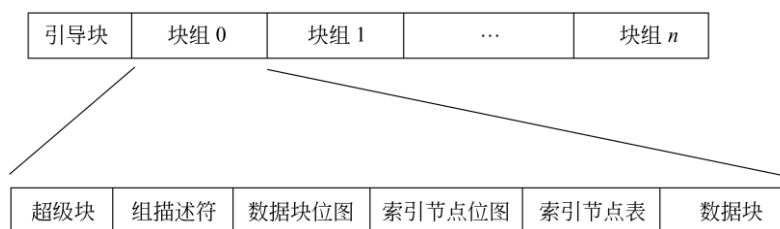


图 10.1Ext2 逻辑磁盘结构

## 10.5.2 ext2 文件系统主要的磁盘数据结构

### 1. ext2 文件系统的磁盘超级块 ext2\_super\_block

```

struct ext2_super_block {
    __le32  s_inodes_count;           /* 索引节点总数 */
    __le32  s_blocks_count;          /* 文件系统总块数 */
    __le32  s_r_blocks_count;        /* 保留的块数 */
    __le32  s_free_blocks_count;     /* 空闲块计数器 */
    __le32  s_free_inodes_count;     /* 空闲索引节点计数器 */
    __le32  s_first_data_block;      /* 第一个数据块号(总是1) */
    __le32  s_log_block_size;        /* 以2的幂次方表示的块大小 */
    __le32  s_log_frag_size;        /* 以2的幂次方表示的片大小 */
    __le32  s_blocks_per_group;      /* 一个块组中的块数 */
    __le32  s_frags_per_group;       /* 一个块组中的片数 */
    __le32  s_inodes_per_group;      /* 一个块组中的索引节点数 */
    __le32  s_mtime;                 /* 最后一次挂装操作的时间 */
    __le32  s_wtime;                 /* 最后一次写操作的时间 */
    __le16  s_mnt_count;             /* 挂装操作计数器 */
    __le16  s_max_mnt_count;         /* 最大挂装次数 */
    __le16  s_magic;                 /* 魔数 */
    __le16  s_state;                 /* 文件系统状态 */
    __le16  s_errors;                /* 错误检查时的动作 */
    __le16  s_minor_rev_level;       /* 次版本号 */
    __le32  s_lastcheck;             /* 最后一次文件系统检查时间 */
    __le32  s_checkinterval;         /* 两次检查之间的间隔 */
    __le32  s_creator_os;            /* 创建文件系统的操作函数 */
    __le32  s_rev_level;             /* 版本号 */
    __le16  s_def_resuid;            /* 保留块的默认 UID */
    __le16  s_def_resgid;            /* 保留块的默认 GID */
    __le32  s_first_ino;             /* 第一个保留的索引节点号 */
    __le16  s_inode_size;            /* 磁盘索引节点的大小 */
    __le16  s_block_group_nr;       /* 这个超级块的块组号 */
    .....
    __u8    s_uuid[16];              /* 128 位文件系统标识符 */
    char    s_volume_name[16];       /* 文件系统卷名 */
    .....
    __u32   s_reserved[190];         /* 用 NULL 填充到块末尾 */
};

```

## 2. ext2 的块组描述符

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;    /* 块位图的块号 */
    __le32 bg_inode_bitmap;    /* 索引节点位图的块号 */
    __le32 bg_inode_table;     /* 第一个索引节点表块的块号 */
    __le16 bg_free_blocks_count; /* 组中空闲块的个数 */
    __le16 bg_free_inodes_count; /* 组中索引节点的个数 */
    __le16 bg_used_dirs_count;  /* 组中目录的个数 */
    __le16 bg_pad;              /* 按字对齐 */
    __le32 bg_reserved[3];      /* 用 null 填充 */
};
```

## 3. 块位图和索引节点位图

### 4. ext2 文件系统的磁盘索引节点 ext2\_inode

```
struct ext2_inode {
    __le16 i_mode;    /* 文件类型及访问权限 */
    __le16 i_uid;     /* 文件属主的 uid */
    __le32 i_size;     /* 以字节为单位的文件长度 */
    __le32 i_atime;    /* 最后一次访问时间 */
    __le32 i_ctime;    /* 文件创建时间 */
    __le32 i_mtime;    /* 最后一次修改时间 */
    __le32 i_dtime;    /* 删除时间 */
    __le16 i_gid;      /* 文件属主的 gid */
    __le16 i_links_count; /* 链接数 */
    __le32 i_blocks;   /* 文件的数据块数 */
    __le32 i_flags;     /* 文件标志 */
    union {
        struct {
            __le32 l_i_reserved1;
        } linux1;
        struct {
            __le32 h_i_translator;
        } h_i_translator;
    };
};
```

```

    } hurd1;
    struct {
        __le32 m_i_reserved1;
    } masix1;
} osd1;          /* 特定的操作系统的信息 */
__le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
/* 数据块指针 */

...
};

```

### 10.5.3 ext2 文件系统的内存数据结构

1. ext2 的内存超级块
2. ext2 的内存索引节点

### 10.5.4 数据块寻址

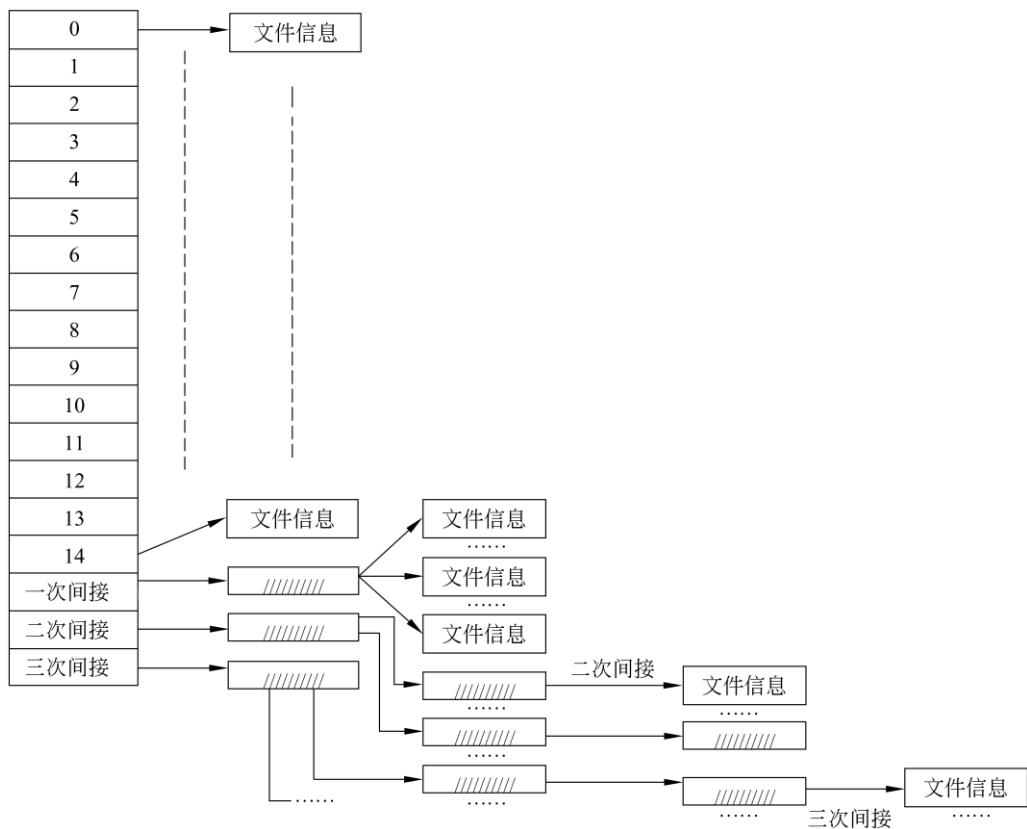


图 10.2 文件映射关系



表 10-1 块大小与其寻址方式所存放文件大小的上限对应表

块大小	直 接	一次间接	二次间接	三次间接
1024	12KB	268KB	64.26MB	16.06GB
2048	24KB	1.02MB	513.02MB	256.5GB
4096	48KB	4.04MB	4GB	~2TB

## 10.6 块设备驱动

### 10.6.1 设备配置

### 10.6.2 设备驱动程序接口

表 10-2 用于块设备文件的函数与缺省文件的操作方法对应表

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

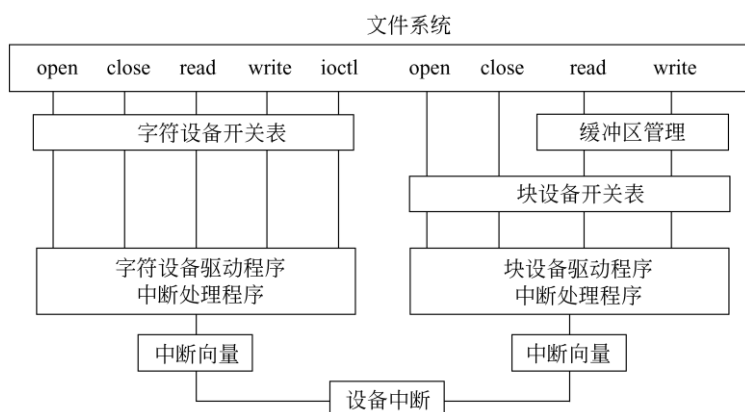


图 10.3 驱动程序及其接口

## 10.7 字符设备驱动

### 习题

10.1 Linux 文件系统的特点是什么？简单描述一个典型的 Linux 系统目录结构。

10.2 Linux 的文件类型有几种，分别是什么？

10.3 什么是 VFS？它有什么作用？其通用数据模型是什么？

10.4 VFS 包括哪些系统调用，并分别简述其功能。

10.5 简述文件系统的注册、挂装以及卸载过程。

10.6 ext2 文件系统的数据块寻址是如何实现的？

10.7 Linux 系统中的设备可分为几种？分别是什么？



## 第 11 章 Windows 的设备管理和文件系统

### 11.1 Windows I/O 系统的结构

#### 11.1.1 设计目标

#### 11.1.2 设备管理服务

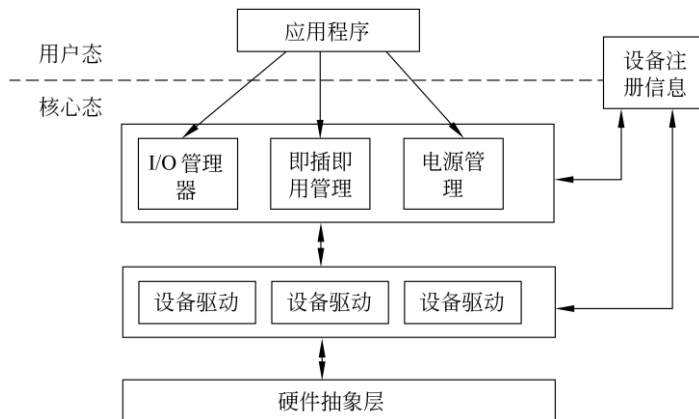


图 11.1 Windows 设备管理体系结构

## 11.2 设备驱动程序和 I/O 处理

### 11.2.1 设备驱动类型和结构

### 11.2.2 Windows 的 I/O 处理

#### 1. I/O 处理的类型

#### 2. 单层驱动的 I/O 处理

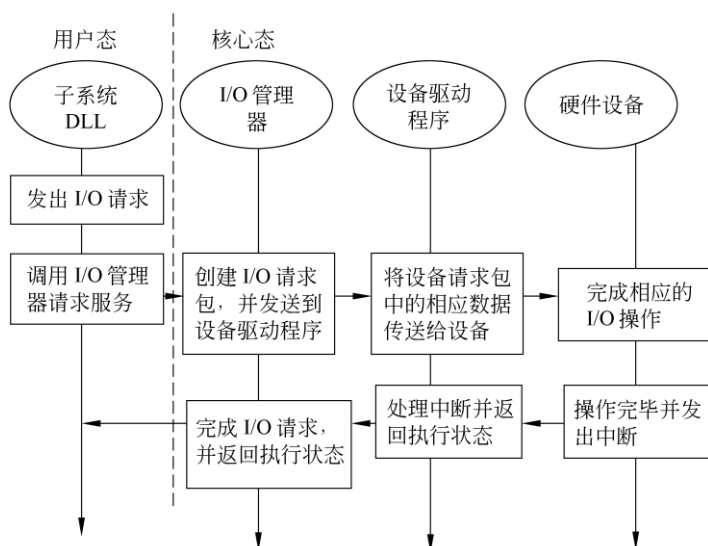


图 11.2 单层驱动处理同步 I/O 请求的过程

### 3. 多层驱动的 I/O 处理示例

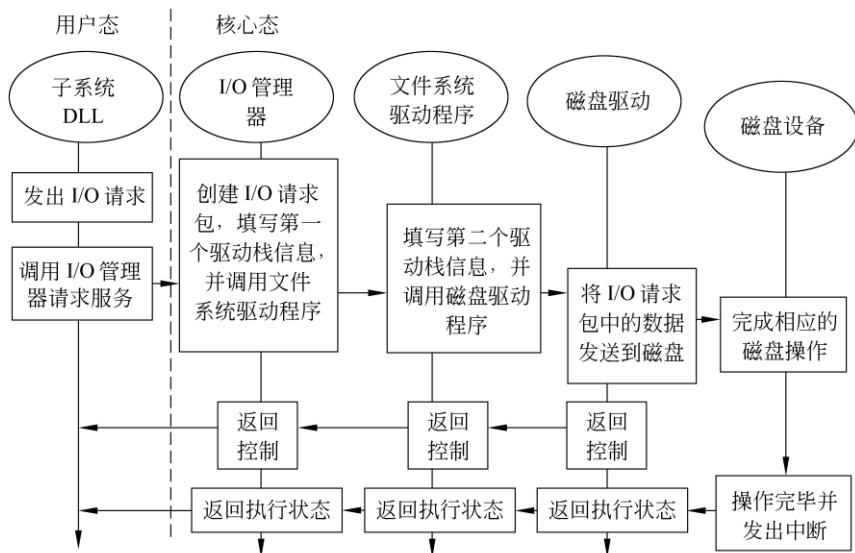


图 11.3 多层驱动处理非同步 I/O 请求的过程

## 11.3 Windows 的文件系统

### 11.3.1 Windows 磁盘管理

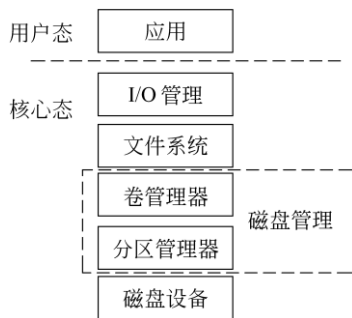


图 11.4 Windows 磁盘管理

## 11.3.2 Windows 文件系统格式

1. CDFS 和 UDF
2. FAT
3. NTFS

## 11.3.3 Windows 文件系统驱动

## 11.4 NTFS 文件系统

### 11.4.1 NTFS 的特点

### 11.4.2 NTFS 的磁盘结构

1. 卷
2. 簇
3. 文件

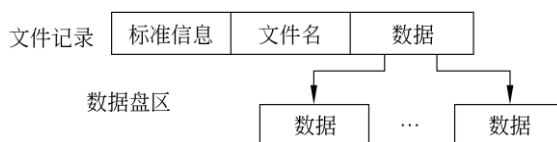


图 11.5 大于 1KB 的文件存储结构

4. 目录

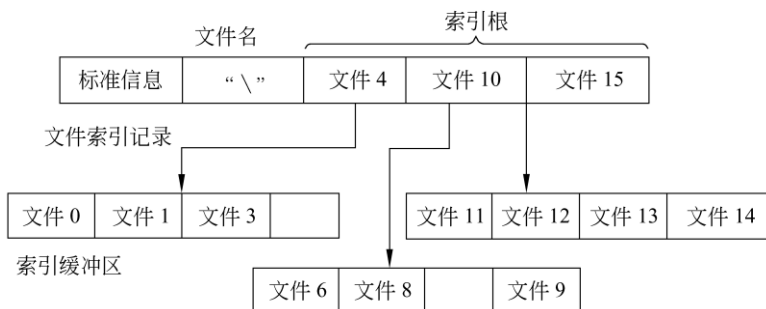


图 11.6 Windows 的根目录存储结构示例

### 11.4.3 NTFS 的文件系统恢复

## 本章小结

## 习题

- 11.1 Windows 的设备管理系统由哪几部分构成，它们之间的关系如何？
- 11.2 试述 I/O 管理器的功能，它如何将 I/O 请求传递到设备驱动程序？
- 11.3 表述 Windows 设备驱动程序的构成，以及其中各个例程的功能。
- 11.4 同步 I/O 处理过程和异步 I/O 处理过程的主要区别是什么？
- 11.5 Windows 的磁盘管理由哪几部分构成，它们如何管理磁盘？
- 11.6 在一个安装 Windows 操作系统的计算机上，分析它的文件目录结构。
- 11.7 为什么 NTFS 用簇，而不是扇区来操作磁盘，它的优点是什么？
- 11.8 描述一个 NTFS 文件的主控表文件记录的格式，并画出它的结构图。
- 11.9 描述一个大目录结构的主控文件表文件索引记录格式，并画出它的结构图。
- 11.10 NTFS 的日志中记录了哪几种操作，它们的功能是什么？



## 参考文献

- [1] Mark E. Russinovich, David A. Solomon. Microsoft Windows Internals. Microsoft Press, 2005
- [2] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne [美] 著. 操作系统概念 (第六版). 郑扣根译. 北京: 高等教育出版社, 2004
- [3] 陈向群、向勇、王雷、马洪兵、郑扣根、(美) Dave Probert 著. Windows 操作系统原理(第2版). 北京: 机械工业出版社, 2004
- [4] Johnson M. Hart. Windows System Programming, 3rd Edition. Addison Wesley Professional. 2004
- [5] J.E. Smith, Ravi Nair. The architecture of virtual machines. IEEE Computer, Volume 38, Issue 5, May 2005, Page(s):32~38
- [6] A.S. Tanenbaum, A.S. Woodhull. Operating Systems: Design and Implementation (Second Edition). Prentice Hall, January 15, 1997
- [7] D. Milojevic, F. Douglass, R. Wheeler (eds): Mobility: Processes, Computers, and Agents. ACM Press, Addison Wesley. 1999
- [8] J.L. Hennessy, D. Goldberg. Computer Architecture: A Quantitative Approach, Third Edition. Morgan Kaufmann, San Francisco, CA, 2003
- [9] H. Huang, P. Pillai, and K.G. Shin. Design and implementation of power aware virtual memory. In proc. USENIX 2003 Annual Technical Conference, pages 57~70, San Antonio, Tx, June 2003
- [10] Tanenbaum A.S. Modern Operating Systems. Englewood Cliffs, N.J.: Prentice Hall, 1992
- [11] Tanenbaum A.S., Woodhull A.S. Operating Systems Design and Implementation. Englewood Cliffs, N.J.: Prentice Hall. 1996
- [12] William Stallings. Operating Systems, Internals and Design principles. Englewood Cliffs, N.J.: Prentice Hall. 1998