

操作系统

期末复习提纲

1 引论

1.1 概念

1.1.1 地位

紧贴系统硬件之上，所有其他软件之下。

1.1.2 目标

方便性、有效性、可扩充性、开放性。

1.1.3 作用

1.1.3.1 作为用户与计算机硬件系统之间的接口

命令、系统调用、图形窗口方式。

1.1.3.2 作为计算机系统资源的管理者

管理对象	管理内容
处理器	处理机的分配和控制
存储器	内存的分配和回收
I/O 设备	I/O 设备的分配和操纵
信息（数据和程序）	即文件管理，负责文件的存取、共享和保护

1.1.3.3 实现了对计算机资源的抽象

1.1.4 ★非形式化定义

计算机系统中的一个系统软件，它是这样一些程序模块的集合——它们管理和控制计算机系统中的软硬件资源，合理地组织计算机的工作流程，以便有效地利用这些资源为用户提供一个功能强大、使用方便和可扩展的工作环境，从而在计算机与其用户之间起到接口的作用。

1.2 ★类型

1.2.1 批处理 OS

1.2.1.1 单道

OS 前身。

特征：**自动性、顺序性、单道性**。

1.2.1.2 多道

后备队列、作业调度：现代意义的 OS。

特征：**多道性、无序性、调度性**。

1.2.1.2.1 优点

资源利用率高、作业吞吐量大。

1.2.1.2.2 缺点

用户交互性差、作业平均周转时间长。

1.2.2 分时 OS

及时接收、及时处理；时间片。

特征：**多路性、独立性、及时性、交互性**。

1.2.3 实时 OS

及时响应、控制所有任务协调一致。

1.2.4 通用 OS

兼有分时、实时、批处理功能。

1.3 ★基本特性

1.3.1 并发

并发：同一时间间隔内发生。（单处理机）

并行：同一时刻发生。（多处理机可实现）

引入**进程、线程**。

1.3.2 共享

资源可供并发执行的进程（线程）共同使用。

1.3.2.1 共享方式

互斥共享（打印机、扫描仪、音频设备）

同时访问（处理机、内存、可重入代码、磁盘）

1.3.3 虚拟

物理实体→逻辑对应物。

1.3.3.1 时分复用

虚拟处理机（多道程序设计技术）

虚拟外部设备（SPOOLING 技术）

1.3.3.2 空分服用

虚拟磁盘、虚拟存储器

1.3.4 异步

进程执行顺序不确定。

1.3.5 特征关系

并发	最重要、最基本
并发和共享	互为存在的条件
虚拟	以并发和共享为前提
异步	并发和共享的结果

1.4 ★功能

1.4.1 处理机管理

进程控制	创建、撤销、改变进程状态 创建、撤销线程
进程同步	协调并发进程推进次序
进程通信	进程间数据传输
进程调度	在多个就绪进程（线程）分配 处理机

1.4.2 存储器管理

内存分配	为每道程序分配内存空间
内存保护	进程间互不干扰、相互保密
地址映射	进程逻辑地址→内存物理地址
内存扩充	覆盖、交换、虚拟存储

1.4.3 设备管理

缓冲区管理	匹配 CPU 和外设速度
设备分配	为用户分配所需设备
设备处理	设备驱动程序 CPU 与设备控制器间的通信

1.4.4 文件管理

文件存储空间管理	如何存放信息
目录管理	目录组织
读写、保护管理	读写、存取控制

1.4.5 操作系统与用户之间的接口

命令接口、图形接口、编程接口。

1.5 结构

1.5.1 设计原则

可维护性、可靠性、可理解性、性能。

1.5.2 类型

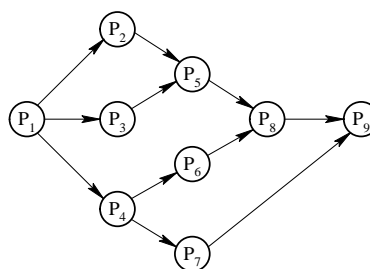
无结构（整体式）、模块化、分层、微内核

2 进程管理

2.1 基本概念

2.1.1 前趋图

有向无环图（DAG）。



(a) 具有九个结点的前趋图



(b) 具有循环的前趋图

上图中 b 是无法满足的前趋关系。

2.1.2 特征

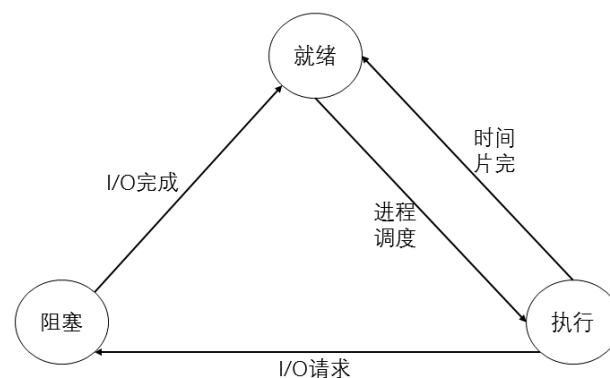
2.1.2.1 ★结构

程序段、数据段和 PCB（Process Control Block）。

进程是一个可拥有资源的独立实体，同时又是一个可以独立调度的基本单位。

动态性、并发性、独立性、异步性。

2.1.2.2 ★基本状态转换



2.1.3 ★与程序的关系

进程是程序的一次执行过程。

2.1.3.1 动态性

动态性是进程的基本特性。

创建产生、调度执行、等待阻塞、撤销消亡。

进程具有一定的生命期，程序只是有序指令集合，是静态概念。

2.1.3.2 并发性

并发性是进程的重要特征，也是 OS 的重要特征。

程序本身不能并发执行。

2.1.3.3 独立性

进程实体是一个能独立运行的基本单位，同时也是系统中独立获得资源和独立调度的基本单位。

未建立进程的程序，不能作为独立单位运行。

2.1.4 ★与线程的比较

2.1.4.1 调度性

线程作为调度和分配的基本单位，进程作为资源拥有的基本单位。

2.1.4.2 并发性

进程间可以并发执行，进程的多个线程也可以并发执行。

2.1.4.3 拥有资源

基本单位是进程。

线程除了一点运行中必不可少的资源（线程控制块、程序计数器、一组寄存器和堆栈）外，不拥有系统资源。

但线程可访问隶属进程的资源。

2.1.4.4 开销

进程创建和撤销，系统需要分配和回收资源；进程切换需要保存和设置的现场信息多于线程。

因此进程创建、撤销和切换的开销，明显大于线程。

隶属同一进程的多个线程，共享同一地址空间等，使其同步和通信实现更方便。

2.2 同步与互斥

2.2.1 ★概念

2.2.1.1 同步

一组并发进程因直接制约而进行**相互合作、相互等待**，使得**各进程按一定速度执行的过程**称为进程间的同步。

2.2.1.2 互斥

一组并发进程中的一个或多个程序段，因为共享公有资源而导致它们必须以一个不允许交叉执行的单位执行。

也可以说，不允许两个以上共享资源的并发进程同时进入临界区称为互斥。

2.2.1.3 临界资源

一次只允许一个进程使用的资源。

2.2.1.4 临界区

每个进程中，访问临界资源的那段代码。

2.2.2 ★同步规则

空闲让进	无进程处于临界区，允许一个进入
忙则等待	已有进程进入，其他等待
有限等待	保证有限时间内能进入临界区
让权等待	进程不能进入临界区时，立即释放

2.2.3 信号量

2.2.3.1 ★类型

2.2.3.1.1 整型信号量

```
wait(S):      while S <= 0 do no-op
               S := S - 1;
signal(S):    S := S + 1;
```

未遵循让权等待原则。

2.2.3.1.2 记录型信号量

```
type semaphore = record
    value: integer      资源数目
    L: list of process  等待队列
end
```

block：阻塞并插入到队列中

wakeup：唤醒并移除队首进程

```
procedure wait(S)
    var S: semaphore;
```

```

begin
    S.value := S.value - 1;
    if S.value < 0 then block(S.L);
end

procedure Signal(S)
var S: semaphore;
begin
    S.value := S.value + 1;
    if S.value <= 0 then wakeup(S.L);
end

```

若 value = 1, 则为互斥信号量。

2.2.3.1.3 AND 型信号量

同时分配所有所需资源。

Swait(S_1, S_2, \dots, S_n) Ssignal(S_1, S_2, \dots, S_n);

2.2.3.1.4 信号量集

可一次分配或释放多个单位的临界资源, 且数量若在指定下限之下, 则不予分配。

Swait($S_1, t_1, d_1, \dots, S_n, t_n, d_n$); t 下限, d 需求

Ssignal($S_1, d_1, \dots, S_n, d_n$);

2.2.3.2 ★应用

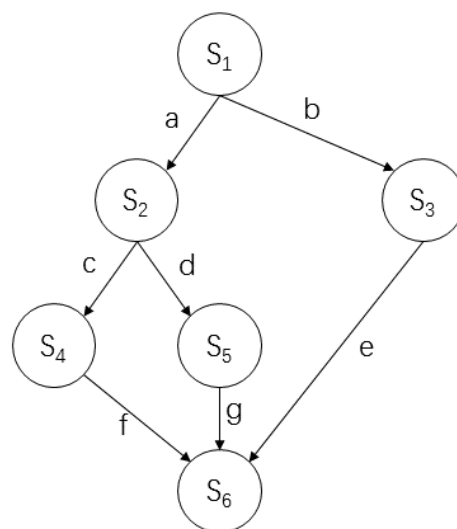
2.2.3.2.1 进程互斥

```

var mutex: semaphore := 1;
begin
    parbegin
        process 1: begin
            repeat
                wait(mutex);
                临界区
                signal(mutex);
                剩余代码
            until false;
        end
        process 2: begin
            repeat
                wait(mutex);
                临界区
                signal(mutex);
                剩余代码
            until false;
        end
    parend
end

```

2.2.3.2.2 实现前趋关系



```

var a, b, c, d, e, f, g: semaphore :=
    0,0,0,0,0,0,0;
begin
    parbegin
        begin
            S1; signal(a); signal(b); end;
        begin
            wait(a); S2; signal(c);
            signal(d); end;
        begin
            wait(b); S3; signal(e); end;
        begin
            wait(c); S4; signal(f); end;
        begin
            wait(d); S5; signal(g); end;
        begin
            wait(e); wait(f); wait(g);
            S6; end;
    parend
end

```

2.2.3.2.3 生产者与消费者

2.2.3.2.3.1 记录型信号

```

var mutex, empty, full: semaphore := 1,n,0;
buffer: array[0..n-1] of item;
i,j: integer := 0,0;
begin
    parbegin
        Producer: begin
            repeat
                生产一个产品
                wait(empty);
                wait(mutex);
                buffer(i) := 产品;
                i := (i+1) mod n;
                signal(mutex);
                signal(full);

```

```

        until false;
    end
    Consumer: begin
        repeat
            wait(full);
            wait(mutex);
            产品 := buffer(j);
            j := (j+1) mod n;
            signal(mutex);
            signal(empty);
            消费 产品;
        until false;
    end
end
parend
end

```

2.2.3.2.3.2 AND 型信号

用 Swait(empty, mutex)替代 wait(empty)和 wait(mutex)。

用 Ssignal(mutex, full)替代 signal(mutex)和 signal(full)。

用 Swait(full, mutex)替代 wait(full)和 wait(mutex)。

用 Ssignal(mutex, empty)替代 signal(mutex)和 signal(empty)。

2.2.3.2.4 读者-写者问题 (读者优先)

2.2.3.2.4.1 记录型信号

```

var rmutex, wmutex: semaphore := 1,1;
    readcount: integer := 0;
begin
    parbegin
        Reader: begin
            repeat
                wait(rmutex);
                if readcount=0 then
                    wait(wmutex);
                readcount := readcount + 1;
                signal(rmutex);
                读
                wait(rmutex);
                readcount := readcount - 1;
                if readcount = 0 then
                    signal(wmutex);
                signal(rmutex);
            until false;
        end
        Writer: begin
            repeat
                wait(wmutex);
                写
                signal(wmutex);
            until false;
        end
    end
end

```

```

        end
    parend
end

```

2.2.3.2.4.2 信号量集

```

var RN: integer;
    L, mx: semaphore := RN, 1;
begin
    parbegin
        Reader: begin
            repeat
                Swait(L, 1, 1);    // 读一个
                Swait(mx, 1, 0);    // 没人写
                读
                Ssignal(L, 1);      // 读完一个
            until false;
        end
        Write: begin
            repeat
                wait(mx, 1, 1, L, RN, 0);
                // 写一个, 没人读
                写
                Ssignal(mx, 1);     // 写完一个
            until false;
        end
    end
end
parend
end

```

2.2.3.2.5 哲学家进餐问题

使用 AND 型信号量以避免死锁。

2.3 例题

2.3.1 PCB 的作用；为什么说 PCB 是进程存在的唯一标志？

PCB 是**进程实体的一部分**，是操作系统中最重要的记录型数据结构。PCB 中记录了操作系统所需的用于**描述进程情况和控制进程运行**所需的全部信息。因而他的作用是使一个在多道程序环境下不能独立运行的**程序**（含数据），称为一个**能独立运行的基本单位**，一个能和其他进程并发执行的进程。

在进程的整个生命周期中，系统总是**通过其 PCB 对进程进行控制**，系统是**根据进程的 PCB** 而不是任何别的什么而**感知到该进程的存在的**，所以说，PCB 是进程存在的唯一标识。

2.3.2 ★读者-写者问题 (写者优先)

```

var rmutex, wmutex, S, wcMutex: semaphore :=
1,1,1,1;

```

```

    readcount, writecount: integer := 0,0;
begin
    parbegin
        Reader: begin
            repeat
                wait(S);

                wait(rmutex);
                if readcount=0 then
                    wait(wmutex);
                readcount := readcount + 1;
                signal(rmutex);

                读

                wait(rmutex);
                readcount := readcount - 1;
                if readcount = 0 then
                    signal(wmutex);
                signal(rmutex);

                signal(S);
            until false;
        end
        Writer: begin
            repeat
                wait(wmutex);
                if writecount = 0 then
                    wait(S);
                writecount := writecount + 1;
                signal(wmutex);

                wait(wmutex)
                写
                signal(wmutex);

                wait(wmutex);
                writecount := writecount - 1;
                if writecount = 0 then
                    signal(S);
                signal(wmutex);
            until false;
        end
    parend
end

```

3 处理机调度

3.1 ★调度层次

3.1.1 高级调度（作业调度）

决定把哪些外存上处于后备队列中的作业调入内存，分配资源，创建进程。

3.1.2 中极调度（交换调度）

按一定算法，将外存中处于静止就绪状态或静止阻塞状态的进程换入内存，而将内存中处于活动就绪状态或活动阻塞状态的某些进程换出到外存。

3.1.3 低级调度（进程调度）

按一定策略，选取就绪队列中的一个进程获得处理机。

最基本的调度。

3.2 ★调度算法

完成时间 - 开始执行时间 = 服务时间

周转时间 = 完成时间 - 到达时间

带权周转时间 = 周转时间 / 服务时间

3.2.1 先来先服务（FCFS）

有利于长作业，不利于短作业。

3.2.2 短作业优先

从后备队列选取估计时间最短的作业。

缺点：对长作业不利，未考虑作业的紧迫程度，作业长短是用户估计的。

3.2.3 优先数法

3.2.3.1 类型

非抢占式优先权算法

抢占式优先权算法

3.2.3.2 优先数

静态优先权：创建进程时确定，运行时保持不变。

动态优先权：创建进程时赋予，运行时可随等待时间增加而改变。

3.2.3.3 轮转调度

时间片轮转。

3.2.3.4 多级反馈队列

- 多个就绪队列，赋予不同优先级；
- 新进程进入内存，放入第一队列队尾，按 FCFS 调度；
- 轮到该进程，若该时间片未完成，则转入下一队列（更低优先级）队尾；
- 仅前几队列为空时，才调度该队列的进程；
- 高优先级队列进程抢占处理机。

3.2.4 ★死锁

3.2.4.1 概念

指多个并发进程彼此等待对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源，从而使得各进程不能继续向前推进。

3.2.4.2 原因

- 系统资源不足
- 进程推进顺序不合适

3.2.4.3 必要条件

- **互斥条件**
排他性使用，其他只能等待
- **请求和保持条件**
保持了至少一个资源，又请求新资源
- **不剥夺条件**
未使用完已获得资源前，不能被剥夺
- **环路等待条件**
死锁时，必然存在一个进程-资源的环形链

3.2.4.4 解决方法

3.2.4.4.1 预防死锁

3.2.4.4.1.1 摒弃“请求和保持”条件

所有进程必须一次性申请所需全部资源。

缺点：资源浪费，进程延迟运行。

3.2.4.4.1.2 摒弃“不剥夺”条件

已经保持了某些资源的进程，在提出新的资源请求而无法及时满足时，必须释放已获得的所有资源。

缺点：前段工作失效，延长周转时间，增加系统开销，降低吞吐量。

3.2.4.4.1.3 摒弃“环路等待”条件

将系统中的资源按类型赋予不同序号，规定进程严格按照资源序号递增顺序申请资源。

缺点：限制新设备增加，使用顺序与规定顺序不同导致资源浪费，限制条件对用户不友好。

3.2.4.4.2 避免死锁

允许进程动态申请资源，但分配前，计算分配的安全性。

若分配不会导致系统进入不安全状态，则分配；否则，等待。

3.2.4.4.2.1 安全状态

系统能按某种进程顺序，为每个进程分配所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利完成。

如果找不到这样一个安全序列，则系统处于不安全状态。

3.2.4.4.2.2 ★银行家算法

3.2.4.4.2.2.1 数据结构

n 进程数，m 资源数。

Available[m]	可利用资源向量
Max[n][m]	最大需求矩阵
Allocation[n][m]	分配矩阵
Need[n][m]	当前需求矩阵

3.2.4.4.2.2.2 处理请求

Request[i]是进程 P[i]的请求向量。

Request[i][j] = K，即进程 P[i]需要 K 个 R[j]资源。

1. 若 Request[i][j] > Need[i][j]，**报错，请求>需要**
2. 若 Request[i][j] > Available[i][j]，**资源不足等待**
3. 试探分配，修改数据

Available[j] -= Request[i][j] **减少可用**

Allocation[i][j] += Request[i][j] **增加分配**
 Need[i][j] -= Request[i][j] **减少需求**

4. 执行安全性算法，**检查安全状态**。
 若安全，正式分配；
 否则，试探分配作废，恢复状态，等待

3.2.4.4.2.3 安全性算法

Work = Available 临时可用资源向量
 Finish[n] = false 完成标识向量

1. 在进程集合中，**找到一个进程**满足
 Finish[i] = false
 Need[i][j] <= Work[j]，对所有 j=1..m
2. 若**找到**，则说明进程 i 获得资源后，可顺利执行到结束，**释放资源**
 Work[j] += Allocation[i][j] **增加临时可用**
 Finish[i] = true **已完成**
 Goto 第 1 步 **继续找**
3. 若**没找到**，则**判断 Finish**
 若 Finish 都为 true，则处于安全状态
 否则，处于不安全状态

3.2.4.4.3 检测死锁
 资源分配图不可简化。

3.2.4.4.4 解除死锁
 剥夺资源、撤销进程。

4 存储器管理

4.1 ★概念

4.1.1 逻辑地址（相对地址、虚地址）
 汇编或编译后的目标代码采用相对地址。

首地址为 0，其余指令相对首地址编址。

4.1.2 物理地址（绝对地址、实地址）
 内存中存储单元的地址，可直接寻址。

4.1.3 地址映射（重定位）
 将用户程序中的逻辑地址转换为运行时及其可直接寻址的物理地址。

4.2 ★程序的装入和链接

4.2.1 装入

4.2.1.1 绝对装入方式

事先知道程序驻留内存位置，编译程序时直接转换为绝对地址，装入时无需修改。

简单；不适于多道程序系统。

4.2.1.2 可重定位装入方式（静态重定位）

由装入程序一次性完成逻辑地址到物理地址的转换。

不需硬件支持；需占用连续的内存空间，程序装入内存后不能移动。

4.2.1.3 动态运行时装入方式（动态重定位）

重定位推迟到程序执行时。

增设重定位寄存器，存放程序（数据）起始地址，以避免影响执行速度。

允许进程在内存中移动；需要需要硬件支持。

4.2.2 链接

链接是将编译后得到的各个目标模块以及所需的库函数链接在一起，形成完整的装入模块。

链接程序需要将各目标模块的相对地址和外部调用符号转换成装入模块的相对地址。

4.2.2.1 静态链接

程序运行之前，将各目标模块及其所需库函数，链接成完整的装入模块。

4.2.2.2 装入时动态链接

链接在装入时进行。

被链接的共享代码称为动态链接库或共享库。

4.2.2.3 运行时动态链接

链接在运行时进行。

4.2.2.3.1 优点

共享：多进程公用一个 DLL

便于局部代码修改：代码升级和重用，无需对可执行文件重新编译或链接

便于运行环境适应：调用不同的 DLL 就可以适应不同环境

部分装入：一个进程可以将多种操作分散在不同 DLL，只装入需要的 DLL

4.2.2.3.2 缺点

链接开销：增加了程序执行时的链接开销

管理开销：程序由多个文件组成，增加管理复杂度

4.3 分区

4.3.1 分区原理

把内存分为一些大小相等或不等的分区，每个应用进程占用一个或几个分区。操作系统占用一个分区。

4.3.2 特点

适于多道程序系统和分时系统。

支持多个程序并发执行，难以进行内存分区的共享。

4.3.3 问题

可能存在内碎片和外碎片。

- **内碎片**：占用分区之内未被利用的空间
- **外碎片**：占用空间之间难以利用的空闲分区（通常是小空闲分区）

4.3.4 ★连续分配方式

4.3.4.1 单一连续分配

只用于单用户、单任务的操作系统中。

内存分为系统区和用户区。

4.3.4.2 固定分区分配

划分固定大小的区域，每个分区只装入一道作业。

4.3.4.3 动态分区分配

在装入程序时，按其初始要求分配；

或在其执行过程中，通过系统调用进行分配或改变分区大小。

4.3.4.3.1 算法

4.3.4.3.1.1 首次适应 (First Fit)

按序查找，分配最先找到的满足需求的空闲分区。

减少查找时间。

4.3.4.3.1.2 循环首次适应 (Next Fit)

将空闲区链成环形链，每次分配从上次分配的位置查找。

4.3.4.3.1.3 最佳适应 (Best Fit)

从全部空闲区，分配能满足作业需求的容量最小的空闲区。

使碎片化尽量小。

4.3.4.3.1.4 最坏适应 (Worst Fit)

从全部空闲区，分配能满足作业需求的容量最大的空闲区。

使剩余空区最大，减少碎片机会。

4.3.4.3.1.5 快速适应算法 (Quick Fit)

空闲分区按容量分类。对每类相同容量的空闲分区，单独设立空闲分区链表。

查找效率高；分区归还主存时算法复杂。

4.3.4.3.1.6 伙伴系统

4.3.4.3.1.7 哈希算法

4.3.4.4 可重定位分区分配

紧凑：拼接合并分散的小分区。

4.3.5 覆盖和对换

覆盖：一个程序的几个代码段或数据段，按照时间先后来占用公共的内存空间。

对换：多程序并发执行，可将暂时不能执行的程序送到外存中，获得空闲内存装入新程序。

4.3.6 ★分区的保护

4.3.6.1 界限寄存器

一对上、下限寄存器或一对基址、限长寄存器。

每次访问内存，硬件自动比较内存地址和界限寄存器的值。若地址越界，则产生越界中断。

4.3.6.2 保护键

为每个分区分配一个独立的**保护键**，为每个**进程**分配对应**钥匙**。

访问内存时，检查是否匹配。不匹配，产生保护性中断。

4.3.7 离散分配方式

4.3.7.1 实存管理

4.3.7.1.1 ★基本分页式

将程序**逻辑地址空间**和**物理内存**划分为**固定大小**的**页**或**页面**。程序加载时，分配所需的所有页（不必连续），需要 CPU 支持。

4.3.7.1.1.1 页的大小

几 KB – 几十 KB

- 小：内碎片小
- 大：页表短、管理开销小、交换时对外存 I/O 效率高

4.3.7.1.1.2 优点

- 无外碎片、内碎片不超过页大小。
- 一个程序不必连续存放。

4.3.7.1.1.3 缺点

程序全部装入内存。

4.3.7.1.1.4 地址结构

页号	位移量
----	-----

$$\text{页号} = \left\lfloor \frac{\text{逻辑地址}}{\text{页大小}} \right\rfloor$$

$$\text{页内地址} = \text{逻辑地址} \bmod \text{页大小}$$

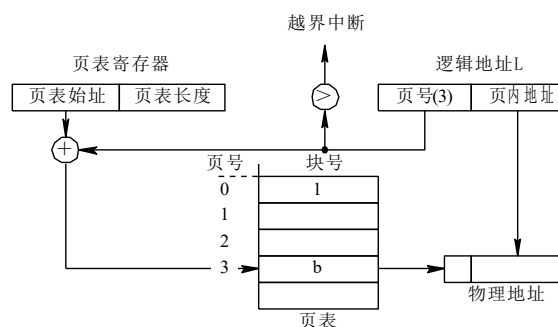
4.3.7.1.1.5 数据结构

页表：页面对应的物理块，每个进程有一个页表。

物理页面表：整个系统有一个，物理内存分配情况。

请求表：整个系统有一个，各页表位置和大小。

4.3.7.1.1.6 地址变换机构



4.3.7.1.2 ★基本分段式

4.3.7.1.2.1 特点

- 方便编程
- 信息共享
- 信息保护
- 动态增长
- 动态链接

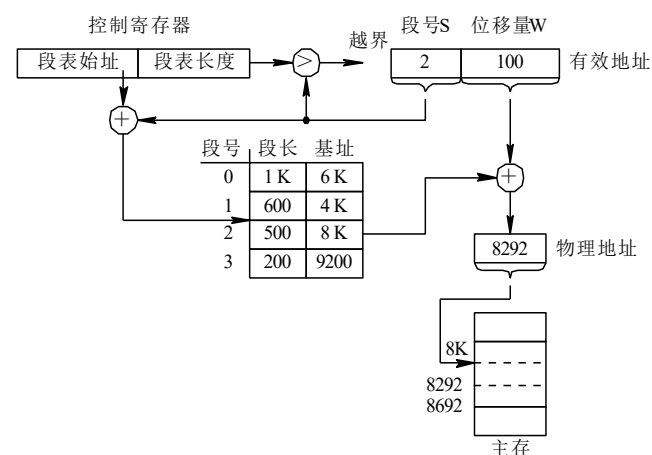
4.3.7.1.2.2 数据结构

段表：**逻辑段**与**物理内存分区**的映射关系。记录**基址**和**段长**。

4.3.7.1.2.3 地址结构

段号	段内地址
----	------

4.3.7.1.2.4 地址变换机构



4.3.7.1.2.5 与分页式比较

- 页是**信息物理单位**，出于**系统管理**的需要；
 - 段是**信息逻辑单位**，出于**用户应用**的需要。
- 例：一条指令可能跨越两页，不会跨越两段。

- 页大小系统固定；
段大小通常不固定，编译时确定。
- 通常段比页大，因而段表比页表短，可以缩短查找时间，提高访问速度
- 逻辑地址表示中：
分页是一维的，各模块链接时必须组织成同一个地址空间；
分段是二维的，各模块链接时可以每个段组织成一个地址空间。

4.3.7.1.3 段页式

将用户程序分为几个段，再把每个段分为若干页。

4.3.7.1.3.1 地址结构

段号	段内页号	页内地址
----	------	------

4.3.7.2 虚拟存储器

4.3.7.2.1 ★定义

指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。

4.3.7.2.2 ★特点

多次性：将一个作业分成多次调入内存。

对换性：允许暂不使用的程序从内存调至对换区。

虚拟性：逻辑上的扩充，并不实际存在。

4.3.7.2.3 ★原理

程序装入时，只将需要执行的部分读入内存。

程序执行时，若所需执行的指令或访问的数据不在内存（缺页、缺段），则处理器通知操作系统将相应的页或段调入内存。

内存中暂不使用的页或段，调出保存在外存上。

4.3.7.2.4 ★硬件支持

系统必须具有一定容量的内存和较大容量的外存。

需要得到页（段）表机制、缺页（段）中断机构和地址变换机构的支持。

4.3.7.2.5 实现方法

4.3.7.2.5.1 请求分页

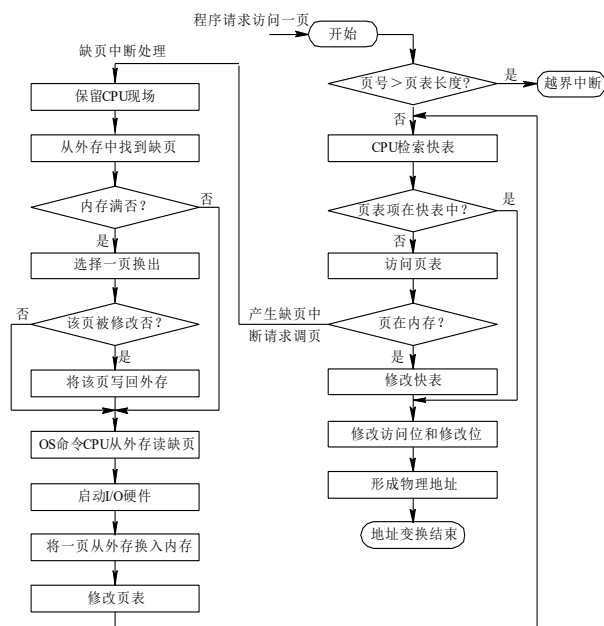
4.3.7.2.5.1.1 扩充

- 页表扩充

页号	物理块号	状态位	访问字段	修改位	外存地址
----	------	-----	------	-----	------

- 缺页中断机构

4.3.7.2.5.1.2 地址变换过程



4.3.7.2.5.1.3 内存分配

4.3.7.2.5.1.3.1 最小物理块数的确定

与计算机硬件结构有关。

4.3.7.2.5.1.3.2 物理块分配策略

- 固定分配局部置换
- 可变分配全局置换
- 可变分配局部置换

4.3.7.2.5.1.3.3 物理块分配算法

- 平均分配算法
- 按比例分配算法
- 考虑优先权的分配算法

4.3.7.2.5.1.4 调页策略

4.3.7.2.5.1.4.1 何时

- 预调页

● 请求调页

4.3.7.2.5.1.4.2 从何处

系统拥有足够的对换区空间：全部从对换区调入。

系统缺少足够的对换区空间：从文件区调入。

UNIX 方式：未运行过的从文件区调入；曾运行但被换出的，从文件区调入。

4.3.7.2.5.1.5 ★缺页率

$$\frac{\text{缺页次数}}{\text{内存访问次数}}$$

4.3.7.2.5.1.6 ★页面置换算法

4.3.7.2.5.1.6.1 最佳置换

理论上的，无法实现。

4.3.7.2.5.1.6.2 先进先出 (FIFO)

引用率

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

页框

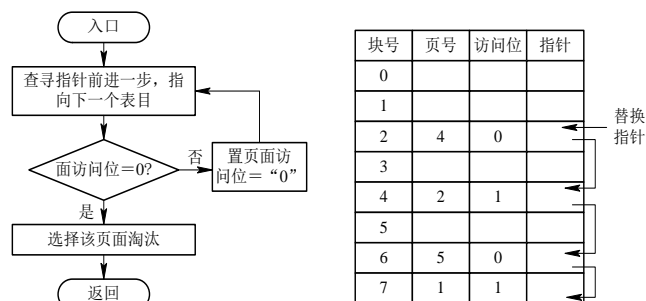
4.3.7.2.5.1.6.3 最近最久未使用 (LRU)

引用率

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

页框

4.3.7.2.5.1.6.4 最近未使用 (Clock 置换)



4.3.7.2.5.1.6.4.1 改进型 Clock 置换算法

将一个页面换出时，若该页已被修改过，则需重新写入磁盘；若未修改，则不必写入。

选择页面换出时，要未使用过的页面，且未修改的页面最好。（访问位：A，修改位：M）

1. 从指针位置开始，扫描循环队列，寻找 A=0，M=0 的页面。遇到的第一个作为淘汰页面。
2. 若未找到，第二轮扫描，寻找 A=0，M=1 的页面，所有扫描过的页面令 A=0，遇到的第一个作为淘汰页面。
3. 若未找到，将所有 A 置 0，重复第 1 步和第 2 步，总能找到淘汰页。

4.3.7.2.5.1.6.5 最少使用 (LFU)

每个页面配置计数器。

对一个新调入页面，可能会因为访问次数最少而被淘汰；而某些页面在某时被访问多次，即使以后不再使用，也不会被淘汰。此时 LFU 算法十分低效。

4.3.7.2.5.1.6.6 页面缓冲算法 (Page Buffering)

FIFO 选择被置换页，将被置换页面放入链表中。

有两个链表，页面未修改和页面已修改的链表，分类放入。

已修改页面达到一定数目后，统一调入外存。

4.3.7.2.5.2 请求分段

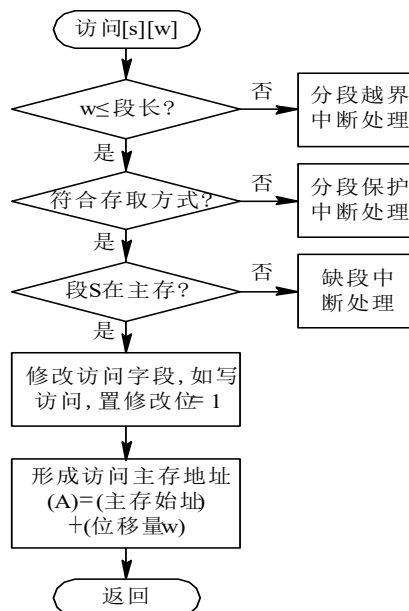
4.3.7.2.5.2.1 扩充

● 段表机制

段名	段长	段的基址	存取方式	访问字段	修改位	存在位	增补位	外存始址
----	----	------	------	------	-----	-----	-----	------

● 缺段中断机构

4.3.7.2.5.2.2 地址变换机构



4.4 ★例题

4.4.1 页表地址变换

某系统拥有逻辑空间 32 个页面，每页 2K，主存 16M。对一个 6 页大某进程，系统为用户它的第 0、1、2、3 页分配了物理块号为 5，10，6，7，而该用户作业的长度为 6 页，试将十六进制的虚拟地址 0A5C，103C，2A5C，3B4D 转换成物理地址。

0A5C: 0000 1 | 010 0101 1100
第 1 页 25C : 第 10 块，即 101 0010 0101 1100
525C

103C: 0001 0 | 000 0011 1100
第 2 页 3C : 第 6 块，即 11 0000 0011 1100
303C

2A5C: 0010 1 | 010 0101 1100
第 4 页，缺页中断。

3BAD: 0011 1 | 011 1010 1101
第 7 页，越界中断

4.4.2 段表地址变换

对于下表所示的段表，请将逻辑地址(0，1145)，(2，5500)，(3，3600)，(5，1230)转换成物理地址。

段 号	内存地址	段 长
0	50K	10K
1	60K	3K
2	70K	5K
3	120K	8K
4	150K	4K

(0，1145)
 $1145 < 10K, 50K + 1145 = 52345$

(2，5500)
 $5500 > 5120$ ，越界中断

(3，3600)
 $3600 < 8K, 120K + 3600 = 126480$

(5，1230)，5 不在段表中，越界中断

5 文件管理

5.1 ★文件的定义

指由创建者所定义的、具有文件名的一组相关数据元素的集合。

可分为：

- 有结构（记录式）文件
- 无结构（流式）文件。

5.1.1 文件的属性

文件类型、文件长度、文件的物理位置、文件的建立时间等。

5.2 ★文件的逻辑结构

从用户角度观察，文件的数据组织形式。

5.2.1 有结构文件

5.2.1.1 基本分类

定长记录、变长记录

5.2.1.2 改进方案

顺序文件、索引文件、索引顺序文件

5.2.2 无结构文件

源程序、可执行文件、库函数等。

以字节为基本访问单位。

UNIX 中所有文件视为流式文件。

5.3 ★外存分配方式

5.3.1 连续分配

5.3.1.1 特点

为每一个文件分配一组相邻接的盘块。

物理上形成了顺序文件结构。

外存上会出现“碎片”，用“紧凑”的方法解决。

5.3.1.2 目录项包含

始址、总块数、最后一块、字节数

5.3.1.3 优缺点

顺序访问容易、速度快

不能灵活删除和插入记录

要求有连续的存储空间（有时需做紧凑处理），必须实现知道文件长度，很难实现文件动态增长。

5.3.2 链接分配

离散分配方式，消除了“碎片”，有利于增删改。

5.3.2.1 隐式链接

目录项中，含有指向链接文件第一盘块和最后一个盘块的指针。

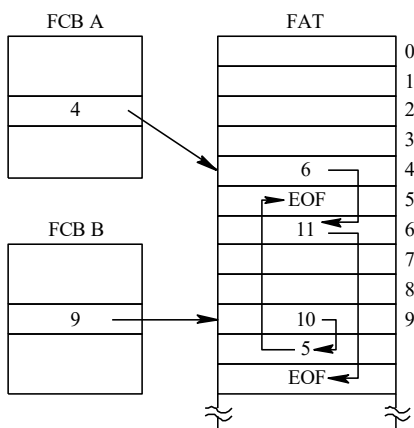
只适合于顺序访问。

5.3.2.2 显式链接

把用于链接文件各物理块的指针，显式地存放在内存的一张“连接表”中。查找在内存中进行。

FCB→FAT。

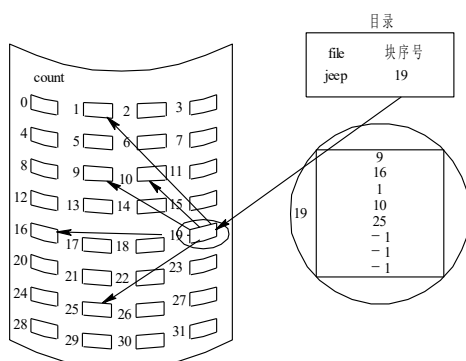
5.3.2.3 MS-DOS FAT12 (FAT表项12位)



5.3.3 索引分配

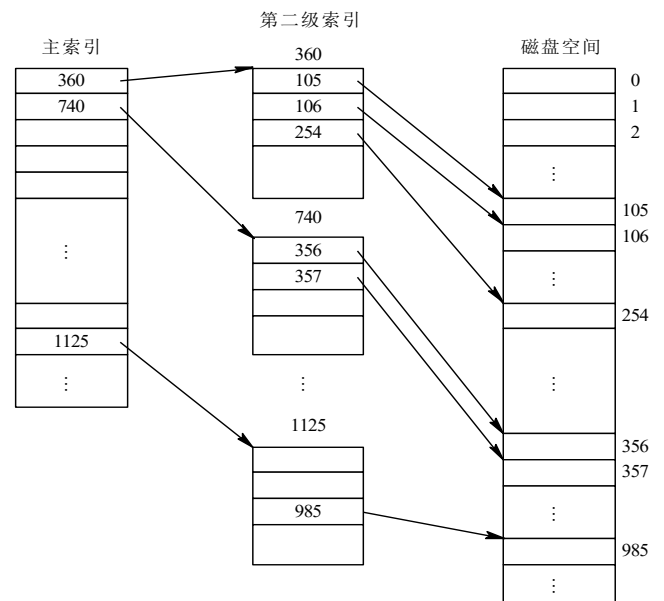
5.3.3.1 单级索引分配

每个文件一个索引块（表）



5.3.3.2 多级索引分配

文件较大时，需要多个索引块。为各索引块建立一个索引块。

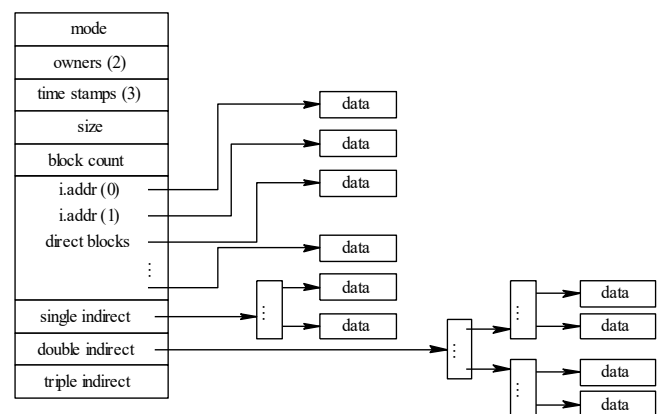


5.3.3.3 混合索引分配

直接地址：iaddr(0) ~ iaddr(9)

间接地址

- iaddr(10)：一次间接地址
- iaddr(11)：二次间接地址
- iaddr(12)：三次间接地址



5.4 目录管理

5.4.1 ★定义

目录：用于标识系统中文件及其物理地址的一种数据结构，供检索使用。

5.4.2 ★要求

- 实现“按名存取”
- 提高对目录的检索速度
- 文件共享
- 允许文件重名

5.4.3 文件控制块和索引节点

5.4.3.1 ★文件控制块 (FCB)

描述和控制文件的数据结构。

5.4.3.1.1 信息项

- 基本信息类：文件名、物理位置等
- 存取控制信息类：文件主等存取权限
- 使用信息类：修改日期等

5.4.3.2 ★索引/节点

5.4.3.2.1 引入

文件访问的查找过程。

5.4.3.2.2 磁盘索引节点

文件主标识符、类型、存取权限、物理地址、长度、连接技术、存取时间

5.4.3.2.3 内存索引节点

对应的磁盘索引节点的内容、索引节点编号、状态、访问计数、所属文件系统的逻辑设备号、链接指针

5.5 ★文件存储空间管理

以磁盘块（扇区）为单位。

5.5.1 空闲表法

连续分配方式

5.5.1.1 空闲表

表项序号、空闲区起始盘块号、盘块数等。

按地址排序。

5.5.1.2 分配与回收

分配：首次适应算法、循环首次适应算法

回收：考虑邻接的前后空闲区拼接合并

5.5.2 空闲链表法

离散分配方式

5.5.2.1 空闲盘块链

分配、回收一个盘块，简单

5.5.2.2 空闲盘区链

与内存的动态分区管理类似

5.5.3 位示图法

用二进制一位来表示磁盘中一个盘块的使用情况。

$m \times n$ 的数组。

5.5.3.1 盘块分配

- 顺序扫描位示图，找到一个或一组为“0”的位
- 将找到的位（第 i 行，第 j 列）转换成与之对应的盘块号

$$b = n(i - 1)j$$

- 修改位示图

5.5.3.2 盘块回收

- 将回收的盘块号转换为位示图的行列号

$$i = \left\lfloor \frac{(b-1)}{n+1} \right\rfloor$$

$$j = (b-1) \bmod (n+1)$$

- 修改位示图

5.5.3.3 优点

- 容易找到一个或一组相邻接的空闲盘块
- 位示图占用空间小，可常驻内存

5.5.4 成组链接法

5.5.4.1 组织

- 空闲盘块号栈：用来存放当前可用的一组空闲盘块号，和空闲盘块数。
- 文件区中的所有空闲盘块，被分成若干组。
- 每一组含有的盘块总数 N 和改组所有的盘块号，计入其前一组的第一个盘块的 $S.free(0) \sim S.free(99)$ 。即由各组的第一个盘块链成一条链。
- 将第一组的盘块数和所有盘块号，记入空闲盘块号栈。

- 最末一组少一个盘块，存放在第二末组的 S.free(1)~S.free(99)中，S.free(0)存放结束标识。

5.5.4.2 分配

栈顶取出空闲盘块号，分配，栈顶指针下移一格，数量减一。

若已是栈底，将该栈底盘块对应内容读入栈中（即下一组空闲盘块），并分配原栈底盘块。

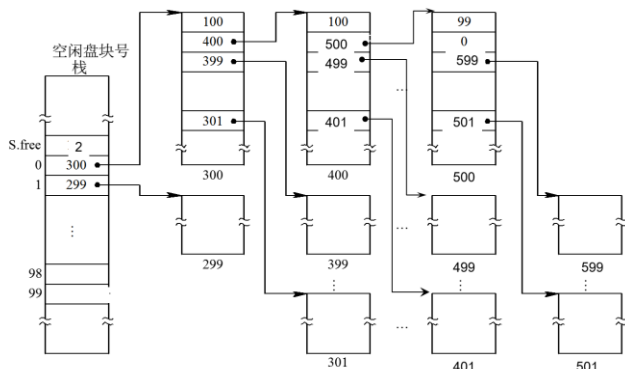
5.5.4.3 回收

回收盘块号记入空闲盘块号栈顶部，数量加一。

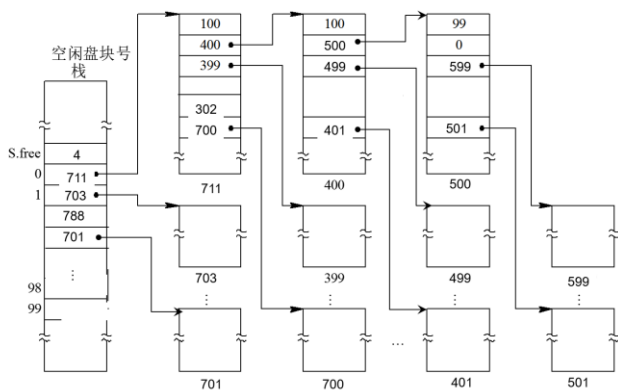
若栈已满，将现有栈中的盘块号记入新回收的盘块中，将其盘块号作为新栈底。

5.5.4.4 例

目前盘块的状态如下。在为某个文件分配 3 个盘块后，系统要删除另一个文件，并回收它所占的 5 个盘块：700，711，703，788，701。



回收后



6 设备管理

6.1 ★I/O 系统的功能

I/O 系统管理的主要对象是 I/O 设备和相应的设备控制器。

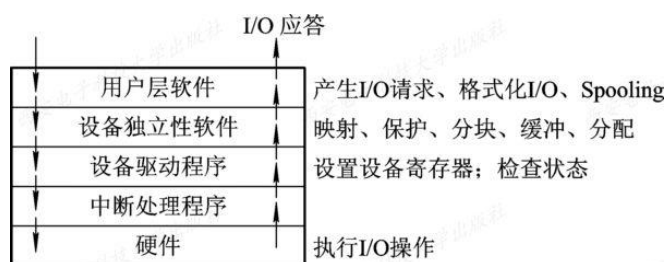
6.1.1 主要任务

- 完成用户提出的 I/O 请求
- 提高 I/O 速率
- 提高设备的利用率
- 为高层的进程方便地使用这些设备提供手段

6.1.2 基本功能

- 隐藏物理设备细节
- 与设备的无关性
- 提高处理机和 I/O 设备的利用率
- 对 I/O 设备进行控制
- 能确保对设备的正确共享
- 错误处理

6.2 ★I/O 软件的层次结构



6.3 ★I/O 系统接口

6.3.1 块设备接口

- 数据的存取和传输以数据块为单位
速率高、可寻址、DMA
- 隐藏磁盘的二维结构
- 将抽象命令映射为底层操作

6.3.2 流设备接口

- 字符设备
速率低、不可寻址、中断 I/O 方式
- get 和 put 操作
采取顺序存取方式
- in-control 指令
大多数流设备都属于独占设备

6.3.3 网络通信接口

现代 OS 中，提供了面向网络的功能。

6.4 ★I/O 设备

6.4.1 设备控制器

主要负责控制一个或多个 I/O 设备，以实现 I/O 设备和计算机之间的数据交换。

6.4.1.1 组成

- 设备控制器与处理机的接口
- 设备控制器与设备的接口
- I/O 逻辑

6.5 ★I/O 通道

是一种特殊的处理机，具有执行 I/O 指令的能力，并通过执行通道程序来控制 I/O 操作。

6.5.1 特点

- 指令类型单一
- 通道没有自己的内存
与 CPU 共享内存

6.5.2 类型

- 字节多路通道
- 数组选择通道
- 数组多路通道

6.6 ★中断技术

- 进程间的切换是通过中断完成的。
- 中断是设备管理的基础。

- 中断处理程序是 I/O 系统中最低的一层，它是整个 I/O 系统的基础。

6.6.1 类型

外中断：来自处理机外部的中断。

I/O 中断、外部信号（ESC 键按下）、定时器、调试程序设置断点等。

内中断：处理机内部产生的中断，一般称“陷入”。各种错误、时间片中断等。

6.6.2 中断向量表

每种设备配以相应的中断处理程序，**中断服务程序**的入口地址称为**中断向量**。

按中断类型号升序排列，存放中断向量的存储区叫**中断向量表**。

6.6.3 ★中断处理过程

中断请求→中断号→中断处理程序的入口地址→转入中断

- 唤醒被阻塞的驱动程序进程
- 保护被中断进程的 CPU 环境
- 分析中断原因，转入相应的设备处理程序
- 中断处理程序→恢复被中断进程的 CPU 现场
- 返回被中断的进程，继续执行

6.7 设备驱动程序

6.7.1 ★功能

- 将命令中的抽象要求转换为与设备相关的底层操作序列
- 检查用户 I/O 请求的合法性
- 发出 I/O 命令
- 及时响应由设备控制器发来的中断请求

6.7.2 特点

- 主要是在请求 I/O 进程与设备控制器之间的一个通信和转换程序
- 与硬件特性紧密相关，不同硬件驱动程序不同
- 与 I/O 设备采用的 I/O 控制驱动方式紧密相关

- 其中一部分必须汇编编写
- 应允许重入

6.8 ★I/O 控制方式

6.8.1 程序 I/O 控制方式

用户进程直接控制内存或 CPU 和外围设备之间的信息传送。

6.8.2 中断驱动 I/O 控制方式

进程要启动某个 I/O 设备工作时，由 CPU 向相应的设备控制器发出 I/O 命令，然后立即返回继续执行原来的任务。

百倍提高 CPU 利用率。

6.8.3 直接存储器访问 DMA I/O 控制方式

在外围设备和内存之间开辟直接的数据交换通路。

6.8.4 I/O 通道控制方式

DMA 方式的发展, 进一步减少 CPU 干预。

把对每个数据块的干预, 减少为对一组数据块和有关的控制和管理干预。

6.9 ★与设备无关的 I/O 软件

6.9.1 设备独立性（设备无关性）

应用程序独立于具体使用的物理设备，使用逻辑设备名来请求某类设备，系统在执行时，则使用该类设备的物理设备名。

6.9.2 设备分配

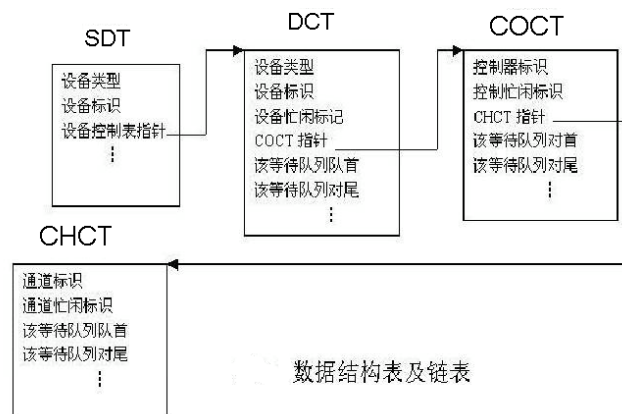
6.9.2.1 数据结构

DCT 设备控制表

COCT 控制器控制表

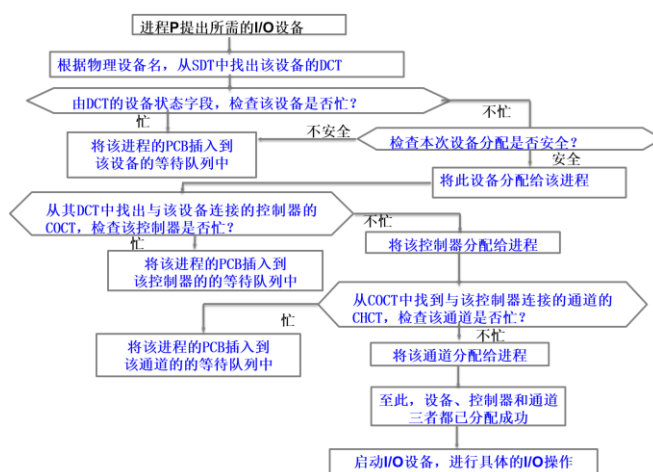
CHCT 通道控制表

SDT 系统设备表



6.9.2.2 基本流程

系统设备表→设备控制表→判断忙闲→控制器控制表→判断忙闲→通道控制表→判断忙闲→分配成功

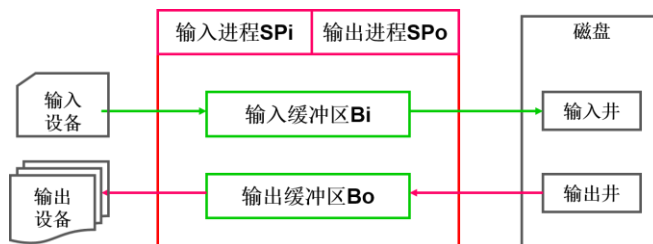


6.9.3 逻辑设备名到物理设备名的映射

逻辑设备表 LUT

逻辑设备名	系统设备表指针
/dev/tty	3
...	...

6.10 ★SPOOLING 技术（假脱机技术）



6.10.1 特点

- 提高 I/O 速度，缓和 CPU 与低速 I/O 设备速度不匹配的矛盾。

- 独占设备改造为共享设备
- 实现虚拟设备功能，用户都感到独占了一台设备

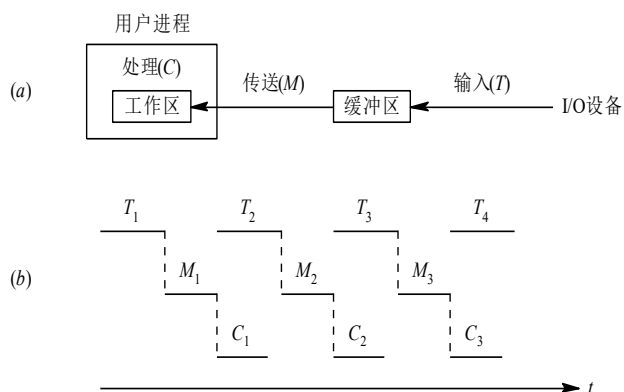
6.11 ★缓冲管理

6.11.1 引入

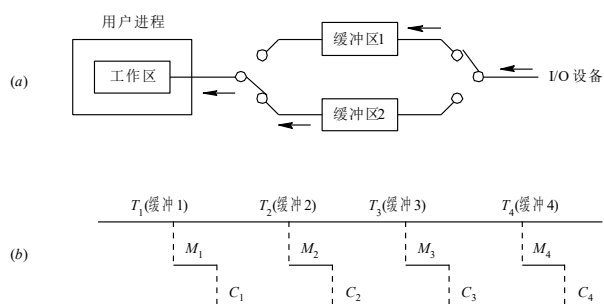
- 缓和 CPU 和 I/O 设备速度不匹配的矛盾
- 减少对 CPU 的中断频率，放宽对中断响应时间的限制
- 解决数据粒度不匹配的问题
- 提高 CPU 和 I/O 设备的并行性

6.11.2 分类

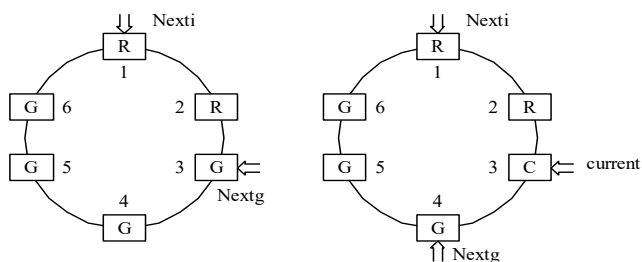
6.11.2.1 单缓冲



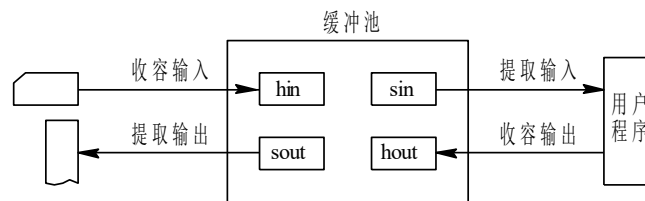
6.11.2.2 双缓冲



6.11.2.3 循环缓冲



6.11.2.4 缓冲池



6.12 ★磁盘存储

6.12.1 磁盘访问时间

6.12.1.1 寻道时间

m : 硬件相关常数, n : 磁道数, s : 磁盘启动时间

$$T_s = m \times n + s$$

一般磁盘 $m = 0.2, s = 5 \sim 30\text{ms}$

高速磁盘 $m \leq 0.1, s \approx 2\text{ms}$

6.12.1.2 旋转延迟时间

r : 转速

$$T_r = \frac{1}{2r}$$

6.12.1.3 传输时间

b : 每次读写字节数, N : 一条磁道字节数

$$T_t = \frac{b}{rN}$$

6.12.1.4 磁盘访问时间

$$T_a = T_s + T_r + T_t$$

6.12.2 磁盘调度算法

6.12.2.1 先来先服务 (FCFS)

缺点: T_s 可能较长

6.12.2.2 最短寻道时间优先 (SSTF)

选择要求访问的磁道与当前磁头所在磁道最近的进程。

缺点: 不能保证平均寻道时间最短, 可能导致“饥饿”

6.12.2.3 扫描算法 (电梯调度算法)

磁头每次只做单方向移动，到边缘，再反向移动。

选择在磁头方向且最近的请求。

消除“饥饿”。

6.12.2.4 循环扫描算法

只做由内向外的单方向扫描，到边缘后，返回最内侧磁道。

消除对边缘磁道的不公平。

- 保护 CPU 环境
- 保存用户定义的参数
- 转入相应的系统调用处理子程序
- 恢复进程环境，继续执行

7.3 图形用户接口

略

8 考试

填空题、选择题、判断题、大题

7 ★操作系统接口

7.1 联机命令接口 (SHELL 命令)

Shell 是命令语言，也是程序设计语言。

7.2 系统调用

提供了用户程序和操作系统之间的接口。

系统调用不仅供所有的**应用程序调用**，而且也供**OS 其他部分调用**。

7.2.1 与一般过程调用的差别

- 运行在不同的系统状态
- 通过软中断进入
- 返回问题
- 嵌套调用

7.2.2 类型

- 进程控制类
- 文件操纵类
- 进程通信类

7.2.3 实现

- 中断和陷入硬件机构
- 系统调用号和参数的设置
- 系统调用步骤
 - 用户态→系统态