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

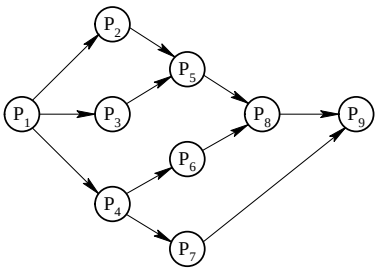
无结构（整体式）、模块化、分层、微内核

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

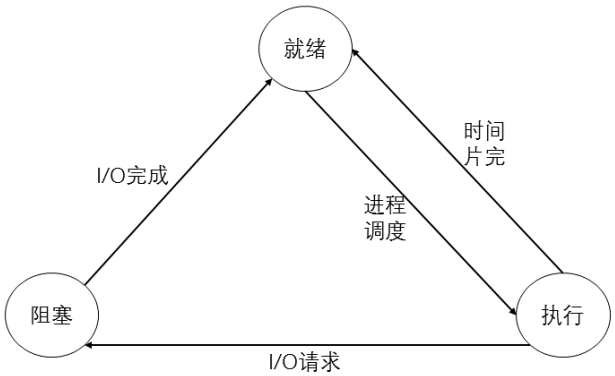
2.1.2.1 ★结构

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, ⋯, S_n) Ssignal(S_1, S_2, ⋯, 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, ⋯, S_n, t_n, d_n);　　t 下限，d 需求

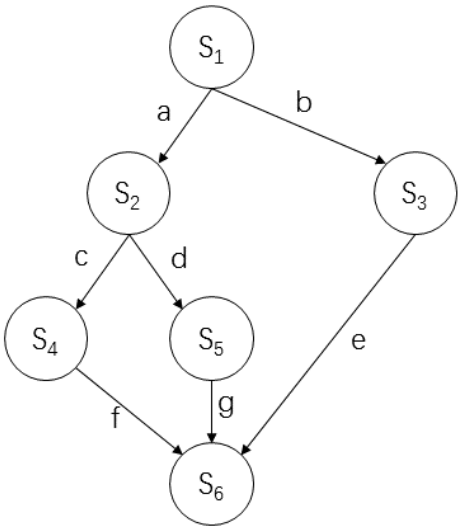
Ssignal(S_1, d_1, ⋯, S_n, d_n);

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
    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
    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
    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(wcmutex);
                if writecount = 0 then
                    wait(S);
                writecount := writecount + 1;
                signal(wcmutex);

                wait(wmutex)
                写
                signal(wmutex);

                wait(wcmutex);
                writecount := writecount - 1;
                if writecount = 0 then
                    signal(S);
                signal(wcmutex);
            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

3.2.4.4.2.2.3 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

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

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

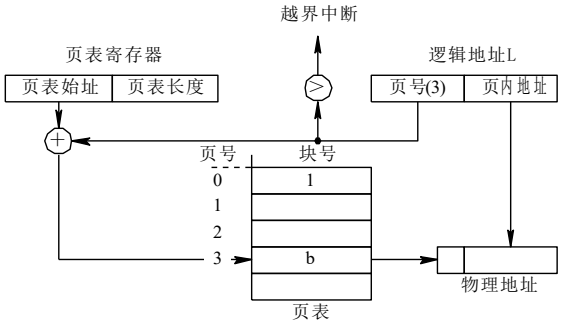
4.3.7.1.1.5 数据结构

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

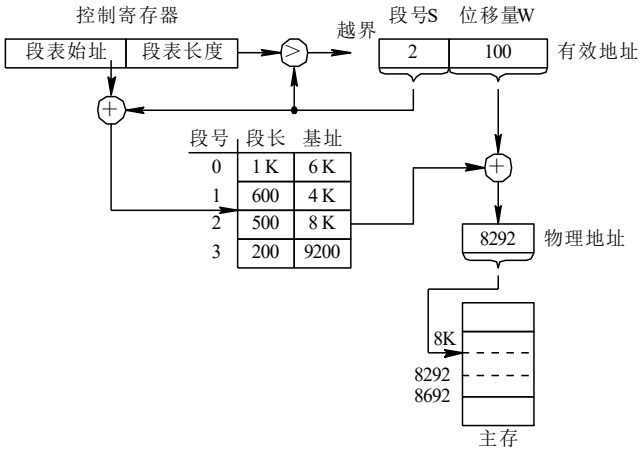
4.3.7.1.2.2 数据结构

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