Chapter 3 内存管理

一、内存管理概念

1. 内存管理的基本原理和要求

内存管理的主要功能：

- 内存空间的分配和回收：
  - 由OS负责内存空间的分配和管理，记录内存的空闲空间，内存的分配情况，并回收结束进程所占用的内存空间。
- 地址转换：
  - 程序的逻辑地址与内存中的物理地址不可能一致，因此存储管理必须提供地址变换功能，将逻辑地址转成物理地址
- 内存空间的扩充：
  - 利用虚拟存储技术从逻辑上扩充内存
- 内存共享：
  - 允许多个进程访问内存的同一部分
- 存储保护：
  - 保证各个进程在各自的存储空间内运行，互不干扰

(1) 逻辑地址与物理地址

编译后，每个目标模块都从0号单元开始编址，这称为该目标模块的相对地址（逻辑地址）

当链接程序将各个模块链接成一个完整的可执行目标程序时，链接程序顺序依次按各个模块的相对地址构成统一的从0号单元开始编址的**逻辑地址空间**（或虚拟地址空间）。

对于32位系统，逻辑地址空间的范围为0~$2^{32}-1$

进程在运行时，看到和使用的都是逻辑地址，用户程序和程序员只需要知道逻辑地址，而内存管理的具体机制是完全透明的。

不同进程可以有相同的逻辑地址，因为相同的逻辑地址可以映射到主存地不同位置。

物理地址空间：指内存中物理单元的集合，它是地址转换的最终地址，进程在运行时执行指令和访问数据，最后都要通过物理地址从主存中存取。

当装入程序将可执行代码装入内存时，必须通过地址转换将逻辑地址转换为物理地址，这个过程为重定位。

操作系统通过**MMU(内存管理部件)**将逻辑地址转换为物理地址，

逻辑地址通过页表映射到物理内存，页表由操作系统维护并被处理器使用。

（2）程序的链接与装入

将用户源程序变为可在内存中执行的程序，经过以下几个步骤：

- 编译：由编译程序将用户源代码编译成若干目标模块
- 链接：由链接程序将编译后形成的一组目标模块，与他们所需要的库函数链接在一起，成一个完整的装入模块。
- 装入：由装入程序将装入模块装入内存运行。

a. 三种链接方式

对目标模块进行连接时，根据链接时间的不同，可以分为三类：

- 静态连接：
  - 在程序运行前，将各目标模块及他们所需的库函数链接成一个完整的装入模块，以后不再拆开。
  - **需要解决问题：修改相对地址，变换外部调用符号，将每个模块中所用的外部调用符号都变换成相对地址**
- 装入时动态链接：
  - 将用户源程序编译后所得到的一组目标模块，在装入内存时，采用边装入边链接的方式，
  - **优点：便于更新和修改，便于实现对目标模块的共享**
- 运行时动态链接：
  - 在程序执行中，需要某目标模块时，才对他进行链接，凡在程序运行中未用到的模块，都不会被调入内存和连接到装入模块上。
  - **优点：加快程序的装入过程，节省内存空间。**

b. 三种装入方式：

- 绝对装入：

（3）进程的内存映像

一个进程的内存映像有几个要素：

- 代码段：程序的二进制代码，只读，被多个进程共享
- 数据段：程序运行时加工处理的对象，包括全局变量和静态变量
- 进程控制块（PCB）：存放在系统区，从用户空间的最大地址向往低地址方向增长。
- 堆：用来存放动态分配的变量，通过调用malloc函数动态地向高地址分配空间。
- 栈：用来实现函数调用，从用户空间的最大地址向低地址方向增长。

数据段和代码段在程序调入内存时就制定了大小，

堆可以在运行时动态地扩展和收缩，（如调用malloc，free）

用户栈在程序运行期间可以动态地扩展和收缩，每次调用一个函数，栈就会增长，从一个函数返回，栈就会收缩。

只读代码段：

- `.init`：是程序初始化时调用的`_init()`函数
- `.text`：是用户程序的机器代码

读写数据段：

- `.data`：已初始化的全局变量和静态变量
- `.bss`：未初始化及所有初始化为0的全局变量和静态变量

（4）内存保护

内存分配前，需要保护操作系统不收进程的影响，同时保护用户进程不受其他用户进程的影响。

方法：

- 在CPU中设置一对**上下限寄存器**，存放用户进程在主存中的上限和下限地址
  - 当CPU要访问一个地址时，分别和两个寄存器的值比较，判断有无越界
- 采用**重定位寄存器**（基址寄存器）和**界地址寄存器**（限长寄存器）进行越界检查
  - 重定位寄存器存放进程的起始物理地址
  - 界地址寄存器存放进程的最大逻辑地址
  - 内存管理部件将逻辑地址与界地址寄存器进行比较，若未发生地址越界，则加上重定位寄存器的值后映射成物理地址，再交送内存单元

> 重定位寄存器是用来“加”的，逻辑地址加上重定位寄存器中的值就是物理地址
>
> 界地址寄存器是用来“比”的，通过比较界地址寄存器中的值与逻辑地址的值来判断是否越界。

（5）内存共享

并不是所有的进程内存空间都适合共享，只有那些**只读的区域才可以共享。**

可重入代码，也称作纯代码，是允许多个进程访问，但不允许被任何进程修改的代码

实际运行时，可以为每个进程配以**局部数据区**，将执行中可能改变的部分复制到该数据区

这样，程序在执行时，只需要对该私有数据区中的内存进行修改，并不去改变共享的代码。

（6）内存分配与回收

存储管理方式的变化：

单一连续分配——>固定分区分配——>动态分区分配

为了提高内存利用率，从连续分配发展到离散分配方式。（页式存储管理）

2. 连续分配管理方式

连续分配方式：为一个用户程序分配一个连续的内存空间。

（1）单一连续分配

（2）固定分区分配

是最简单的一种多道程序存储管理方式，将用户内存空间划分为若干固定大小的分区，每个分区只装入一道作业。

当有空闲分区时，再从外存的后备作业队列中选择适当大小的作业装入该分区。

划分分区的两种方法：

- 分区大小相等：程序小了会造成浪费，程序大了无法被装入，缺乏灵活性
- 分区大小不等：划分为多个较小的分区，适量的中等大小的分区，和少量大分区。

维护一个分区使用表，按照分区大小排队，各表项对应分区的起始地址，大小及状态（是否已分配）

回收时，只需要将对应表项设置为“未分配”即可。

问题：

- 程序太大而放不进任何一个分区
- 程序小于固定分区大小时，也要占用一个完整的内存分区，即内部碎片
- 固定分区无外部碎片，但是不能实现多进程共享一个主存区，内存利用率低。

（3）动态分区分配

a. 动态分区分配的基本原理

也叫作可变分区分配，指进程装入内存时，根据进程的实际需求，动态地分配内存，并使分区的大小正好适合进程的要求。

系统中分区的大小和数量可变。

动态分区在开始时很好，但是随着时间推移，内存中会产生越来越多的小内存块，内存的利用率也下降，**这些小内存块叫做外部碎片**

存在于所有分区的外部，与固定分区中的内部碎片相对。

外部碎片可以通过紧凑技术克服，即操作系统不时地对进程进行移动和整理

需要重定位寄存器的支持，且相对费时，紧凑空间类似于Windows系统中的磁盘碎片管理程序，后者是对外存空间的紧凑。

b. 动态分区分配的内存分配与回收

设置一张空闲分区链（表），按起始地址排序。

检索空闲分区链，找到所需的分区，若其大小大于请求大小，从该区中按请求大小割出一块空间分配给装入进程。余下部分仍留在空闲分区链中。

回收类似，可能出现以下情况：

- 回收区与插入点的前（后）一空闲分区相邻，将这两个分区合并，修改前（后）一分区表项大小为两者之和

- 回收区同时与插入点的前，后两个分区相邻，将三个分区合并，修改前一分区表项大小为三者之和，取消后一分区表项。
- 回收区没有相邻空闲分区，为回收区新建一个表项，填写起始地址，大小，并插入空闲分区链。

c. 基于顺序搜索的分配方法

将作业装入主存时，要按照一定的分配算法，从空闲分区链（表）中选择一个分区，以分配给该作业。

按分区检索方式，可分为：顺序分配算法，索引分配算法

顺序分配算法：依次搜索空闲分区链上的空闲分区，以寻找一个满足大小要求的分区。

- **首次适应算法（First-Fit）**：
  - 空闲分区按照地址递增的顺序排列，每次分配时，顺序查找一个能满足大小的分区，分配给作业。
  - 保留了内存高地址部分的大空闲分区，有利于后续大作业的装入
  - 使内存低地址部分出现许多小碎片，从而每次查找都要经过这些区域，增加了开销。
- **邻近适应算法（Next-Fit）：**
  - 循环首次适应算法
  - 由首次适应算法演变而来，不同的是，分配内存时从上次查找结束的位置开始继续查找。
  - 让内存低，高地址部分的空闲分区以同等概率被分配，导致内存高地址部分没有大空闲分区可用。
  - 通常比首次适应算法更差
- **最佳适应算法（Best-Fit）：**
  - 空闲分区按容量递增的次序排列，
  - 每次分配内存时，顺序查找到**第一个**满足大小的空闲分区，即最小的空闲分区
  - 虽然能留下大空闲分区，但性能很差，因为每次分配会留下越来越多很小的内存块，从而产生最多的外部碎片。
- **最坏适应算法（Worst-Fit）：**
  - 空闲分区按容量递减的次序排列
  - 每次分配内存，顺序查找到第一个能满足要求的空闲分区，即最大的空闲分区，从中割
  - 给作业
  - 看似不容易产生碎片，但是会很快导致没有大空闲分区可用，性能也很差

d. 基于索引搜索的分配算法

当系统很大时，空闲分区链可能很长，此时采用顺序分配算法可能很慢，在大，中，型系统中采用索引分配方式

索引分配算法的思想是：根据其大小对空闲分区进行分类，对于每类（大小相同）空闲分区，单独设立一个空闲分区链，并设置一张索引表来管理这些空闲分区链。

为进程分配空间时，先在索引表中查找所需空间大小对应的表项，并从中得到空闲分区链的头指针。

再在空闲分区链中得到一个空闲分区。

索引分配算法有三种：

- 快速适应算法：
  - 根据进程长度，再索引表中找到可以容纳地最小空闲分区链表

- 从链表中取下第一块进行分配
- 优点：查找效率高，不会产生内存碎片
- 缺点：回收时需要有效合并分区，算法复杂，开销大

- 伙伴系统：
  - 2）伙伴系统。规定所有分区的大小均为 2 的 k 次幂（k 为正整数）。当需要为进程分配大小为 n 的分区时（$2^{i-1} < n \leqslant 2^i$），在大小为 2^i 的空闲分区链中查找。若找到，则将该空闲分区分配给进程。否则，表示大小为 2^i 的空闲分区已耗尽，需要在大小为 2^{i+1} 的空闲分区链中继续查找。若存在大小为 2^{i+1} 的空闲分区，则将其等分为两个分区，这两个分区称为一对伙伴，其中一个用于分配，而将另一个加入大小为 2^i 的空闲分区链。若不存在，则继续查找，直至找到为止。回收时，需要要将相邻的空闲伙伴分区合并成更大的分区。
- 哈希算法：
  - 根据空闲分区链表分布规律，建立哈希函数，构建一张以空闲分区大小为关键字的哈希表
  - 每个表项记录一个对应空闲分区链的头指针
  - 分配时，通过哈希函数计算得到哈希表中的位置

非连续分配方式的存储密度低于连续分配方式，非连续分配方式根据**分区大小是否固定**，分为**分页存储管理**和**分段存储管理**

在分页存储管理中，又根据运行作业时**是否要将作业的所有页面都装入内存**才能运行，又分为**基本分页存储管理，请求分页存储管理**

3. 基本分页存储管理

固定分区会产生内部碎片，动态分区会产生外部碎片，这两种技术对内存的利用率都比较低。

为了避免碎片的产生，引入分页的思想：

- 将内存空间分为若干固定大小（如4KB）的分区，称为**页框，页帧，物理块**
- 进程的**逻辑地址空间**也分为与块大小相等的若干区域，称为**页或页面**
- 操作系统以页框为单位为各个进程分配内存空间

分页的方法像是分区相等的固定分区技术，分页管理不产生外部碎片。

不同点：

- 块的大小相对分区小很多，**进程也按照块划分**，进程运行时按块申请主存可用空间并执行。
- 这样进程只会在为最后一个不完整的块申请一个主存块空间时，才产生主存碎片。
- 即使有内部碎片，相对于进程也比较小。每个进程平均只产生半个块大小的内部碎片（页内碎片）

（1）分页存储的几个概念

a. 页面和页面大小

- 页号：进程的逻辑地址空间中的每个页面都有一个编号，从0开始
- 页框号（物理号）：内存空间中的每个页框也有一个编号，从0开始

进程在执行的时候需要申请内存空间，即为每个页面分配内存中的可用页框，产生了页号和页框号一一对应

- 页面大小是2的整数次幂（方便地址转换）
- 页面小会使进程页面过多，页表过长，增加硬件地址转换开销，降低页面换出换入效率

- 页面过大会使页内碎片增多，降低内存利用率

b. 地址结构

31 … 12	11 … 0
页号 P	页内偏移量 W

图 3.7　某个分页存储管理的逻辑地址结构

- 前一部分：页号P，最多允许2^{20}页
- 后一部分：页内偏移量，每页大小2^{12}
- 页号，页内偏移，逻辑地址有可能是十进制给出，需要转换

c. 页表

为方便找到进程的每个页面在内存中存放的位置，系统为每个进程建立一个**页面映射表**，简称页表

进程的每个页面对应一个页表项：由页号和块号组成。

记录了页面在内存中对应的物理块号

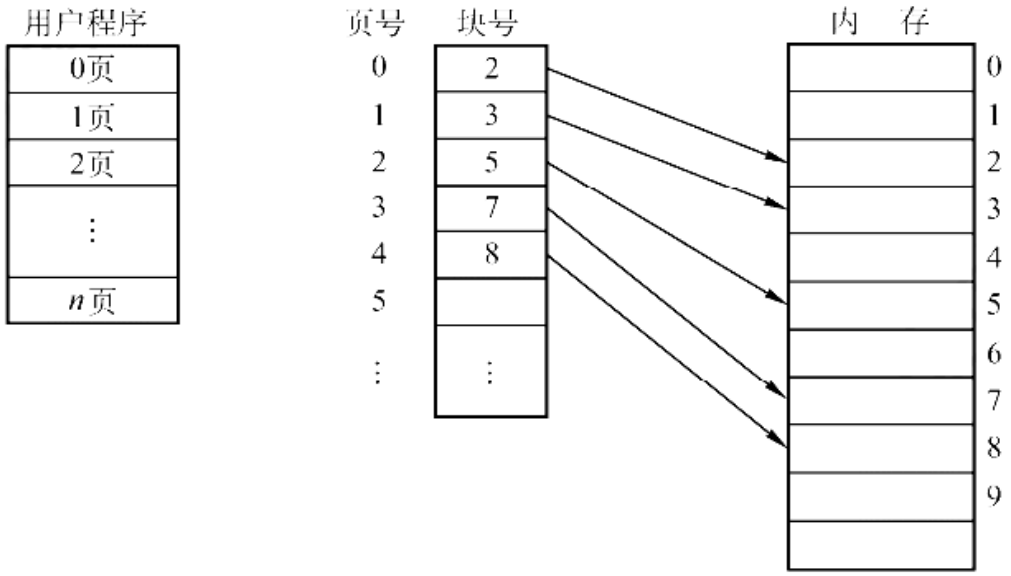


图 3.8　页表的作用

通过查找页表，就可以找到每页在内存中的物理块号。

页表的作用：实现从页号到物理块号的地址映射。

（2）基本地址变换机构

借助于页表实现，**实现逻辑地址转换为内存中的物理地址**

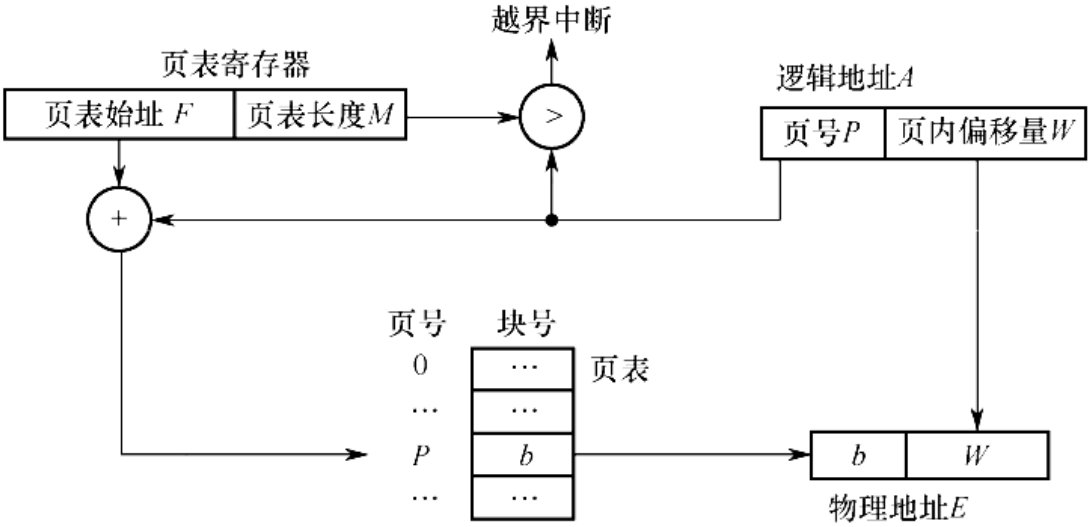


图 3.9　分页存储管理系统中的地址变换机构

页表中，页表项连续存放，因此页号可以隐含，不占空间

页表寄存器(PTR)：存放页表在内存的起始地址F和页表长度M。

CPU系统中只有一个页表寄存器，进程未执行时，页表的起始地址和页表长度放在本进程的PCB中。

当进程被调度后，将页表起始地址和长度放在页表寄存器中。

分页管理存在的问题：

- 每次访存都需要进行逻辑地址到物理地址的转换，地址转换必须快
- 每个进程引入页表，用于存储映射机制，页表不能太大，否则内存利用率会降低

(3) 具有快表的地址变换机构

若页表全部放在内存中，则存取一条数据或一条指令至少需要两次访存：

- 第一次：访问页表，确定所需存取的数据或指令的物理地址
- 第二次：根据该地址存取数据或指令

在地址变换机构中设置一个具有并行查找能力的告诉缓冲存储器——快表（TLB），也称为相联存储器，用来存放当前访问的若干页表项，加速地址变化过程。

与之对应，主存中的页表常称为慢表

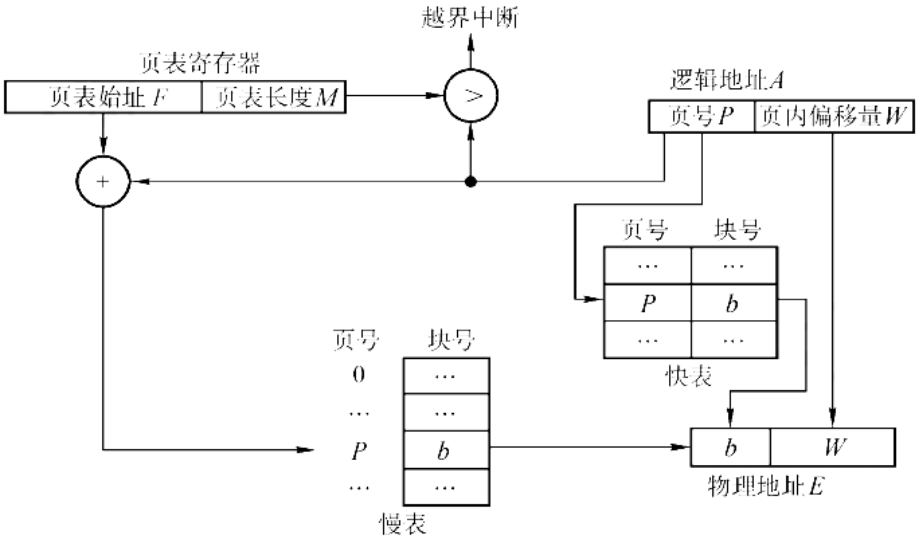


图 3.10　具有快表的地址变换机构

地址变换过程：

- CPU给出逻辑地址，由硬件进行地址转换，将页号与快表中的所有页号进行比较
- 若找到匹配的页号，直接取出该页对应的物理块号，与页内偏移量拼接成物理地址（存取数据只要一次访存）
- 若没有找到匹配的页号，则需要访问内存中的页表。（需要两次访存）

> 找到页表项后，应同时存入快表，以便后边再次访问，若快表已满，则需要按照算法淘汰掉一个旧页表项
>
> 一般块表中的命中率可达90%，这样分页带来的速度损失就降低到10%以下，快表的有效性基于**局部性原理**

（4）两级页表

引入分页后，进程只需要在执行时将保存有映射关系的页表调入内存。

解决页表过大的方法：

- 对于页表所需的内存空间，采用离散分配方式，用索引表来记录各个页表的存放位置
- 只将当前需要的部分页表调入内存，其余页表仍驻留磁盘，需要时再调入（虚拟内存的思想）

实际离散分配页表就是再建立一张页表，称为外层页表（页目录）

一级页号或页目录号10位	二级页号或页号10位	页内偏移12位

图 3.11　逻辑地址空间的格式

两级页表是在普通页表结构上再加一层页表，其结构如图 3.12 所示。

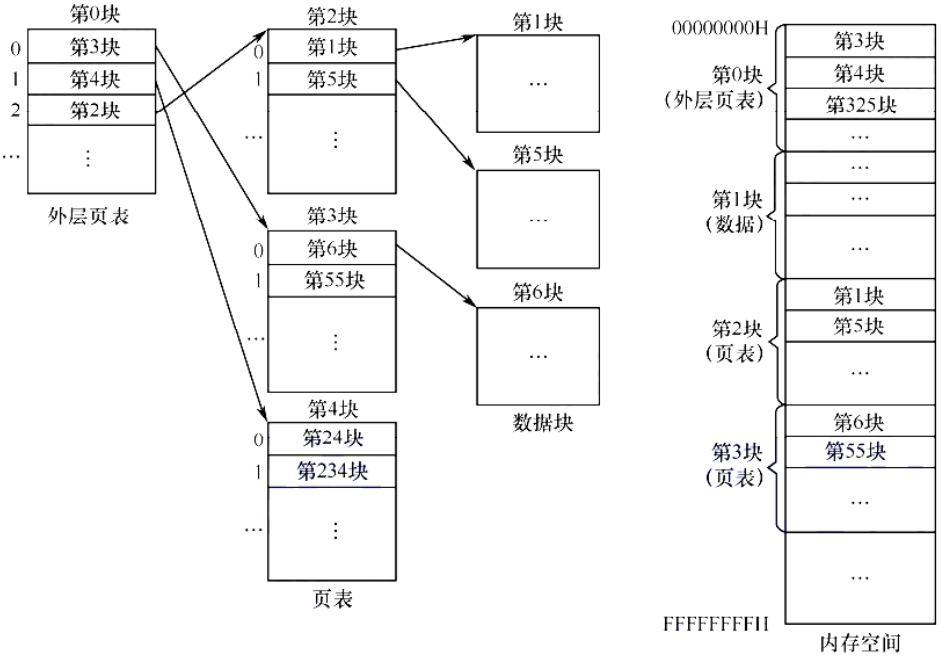


图 3.12　两级页表结构示意图

在页表的每个表项中，存放的是进程的某页对应的物理块号，如0号页放在1号物理块中，1号页放在5号物理块中。

在外层页表的每个表项中，存放的是某个页表分页的始址，如0号页表放在3号物理块中，可以用外层页表和页表来实现进程从逻辑地址到物理地址的变换。

系统中增设一个外层页表寄存器（页目录基址寄存器），用于存放页目录始址。

- 将逻辑地址中的页目录号座位页目录的索引，从中找出对应页表的开始地址（在内存中）
- 再用二级页号作为页表分页的索引，找出对应的页表项；
- 将页表项中的物理块号和页内偏移拼接成物理地址
- 再用该物理地址访问内存单元
- **共进行了三次访存**

4. 基本分段存储方式

分页管理的目的是提高内存利用率，提升计算机的性能，分页通过硬件基址实现，对用户完全透明（意思是不可见）

分段管理则考虑了用户和程序员，以方便编程，信息保护，共享，动态增长，动态链接。

（1）分段

分段系统将用户进程的逻辑地址划分为大小不等的段。

用户进程：主程序段，两个子程序段，栈段，数据段。

可以将这个进程划分为5段，每段从0开始编址，并分配一段连续的地址空间（段内连续，段间不连续，进程的地址空间是二维的）

分段存储的逻辑地址：段号S+段内偏移量W。

31 … 16	15 … 0
段号 S	段内偏移量 W

图 3.13　分段系统中的逻辑地址结构

在页式系统中，逻辑地址的页号和页内偏移量对用户透明

在分段系统中，段号和页内偏移量必须由用户显式提供，高级语言程序汇总，由编译程序完成。

（2）段表

每个进程都有一张逻辑地址与内存空间映射的段表，进程的每个段对应一个段表项，段表项记录了该段在内存中的始址和段长。

段　号	段　长	本段在主存的始址

图 3.14　段表的内容

配置段表后，进程可以通过查找段表，找到每段对应的内存区。

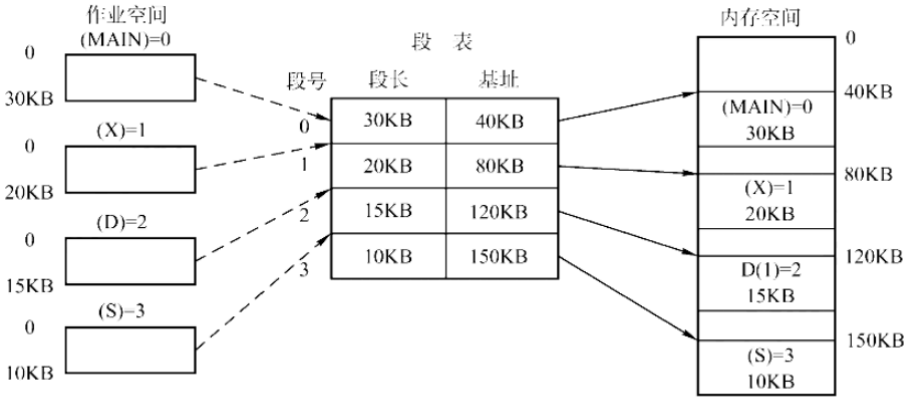


图 3.15　利用段表实现物理内存区映射

段表连续存放，段号可以隐含。

(3) 地址变换机构

系统中设置一个段表寄存器，用于存放段表地址F和段表长度M。

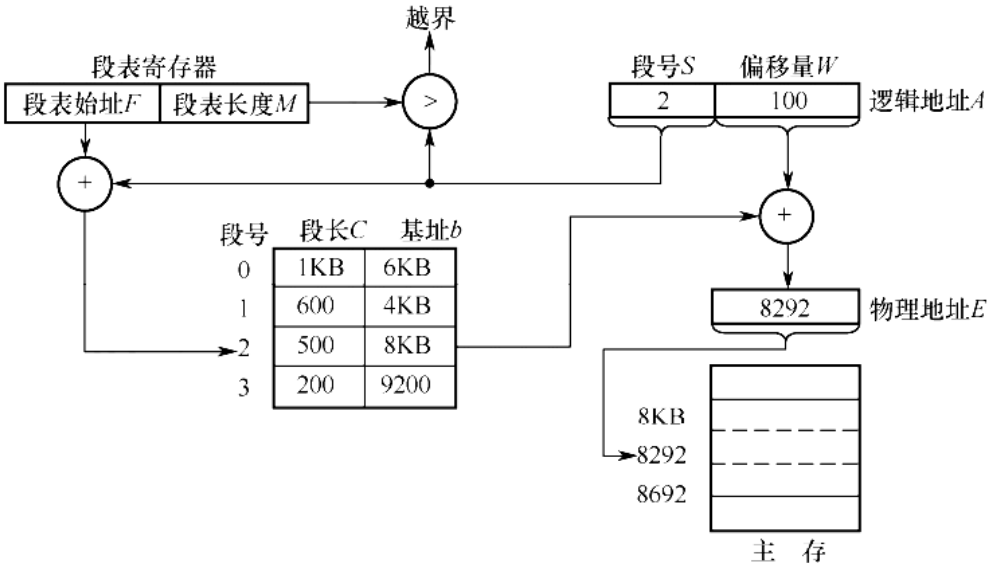


图 3.16　分段系统的地址变换过程

- 从逻辑地址A中取出前几位为段号S，后几位为段内偏移量W
- 判断段号是否越界，段号S>段表长度M，则产生越界中断，否则继续执行
- 段表中查询到对应的段表项，段号S对应的段表地址=段表起始地址F+段号S×段表项长度
- 取出段表项中该段的段长C，若W>C，则产生越界中断
- 取出段表项中该段的起始地址b，计算物理地址E=b+W，用物理地址E去访存。

(4) 分页和分段的对比

分页和分段都是非连续分配方式，需要通过地址映射机构实现地址变换。

不同点表现在：

- 页是信息的**物理单位**，段是信息的**逻辑单位**
    - 分页是系统行为，主要为了提高内存利用率，对用户不可见
    - 分段为了满足用户需求，用户按照逻辑关系将程序划分为若干段，分段对用户可见

- **页的大小固定**，由系统决定，段的长度不固定，由用户编写的程序决定
- 分页管理的地址固定一维的，段式管理不能给一个整数便确定对应的物理地址，因为每段长度不固定，**无法通过除法得到段号**，**无法通过求余得出段内偏移**，所以要显示给出段号和段内偏移，因此分段管理的地址空间是二维的

（5）段的共享与保护

- 若被共享的代码占N个页框，则在每个进程的页表中就要建立N个页表项，指向被共享的N个页框。
- 而在分段系统中，不管该段多大，都只需要为这个段设置一个段表项，因此容易实现共享。
- 为了实现段共享，在系统中配置一张共享段表，所有共享的段都在共享段表中占一个表项。
- 表项中记录了共享段的段号，段长，内存地址，状态（存在）位，外存地址，共享进程计数count等信息。
    - count记录有多少进程正在共享该段。当count=0时，回收该段占用的内存区。
- 对于一个共享段，不同进程可以具有不同地的段号，每个进程用自己的段号去访问该共享段。

不被任何进程修改的代码成为可重入代码或纯代码，允许多个进程同时访问。

为了防止程序在执行时修改共享代码，在每个进程中都必须配局部数据区，将执行过程中可能改变的部分复制到数据区，这样进程就可以对数据区中的内容进行修改。

分段管理的保护方法有两种：

- 存取控制保护
- 地址越界保护。
    - 将段表寄存器中的段表长度（有几个段表项）与逻辑地址中的段号对比，若段号大于段表长，产生越界中断
    - 再将段表项中的段长和逻辑地址中的段内偏移量对比，若段内偏移量大于段长，产生越界中断。
- 分页管理只需要判断页号是否越界，页内偏移不会越界。

5. 段页式存储管理

进程先被分成若干逻辑段，每段有自己的段号，将各段分成若干大小的固定的页。

对内存空间的管理与分页存储管理一样，将其分成若干和页面大小相同的物理块，对内存的分配以物理块为单位。

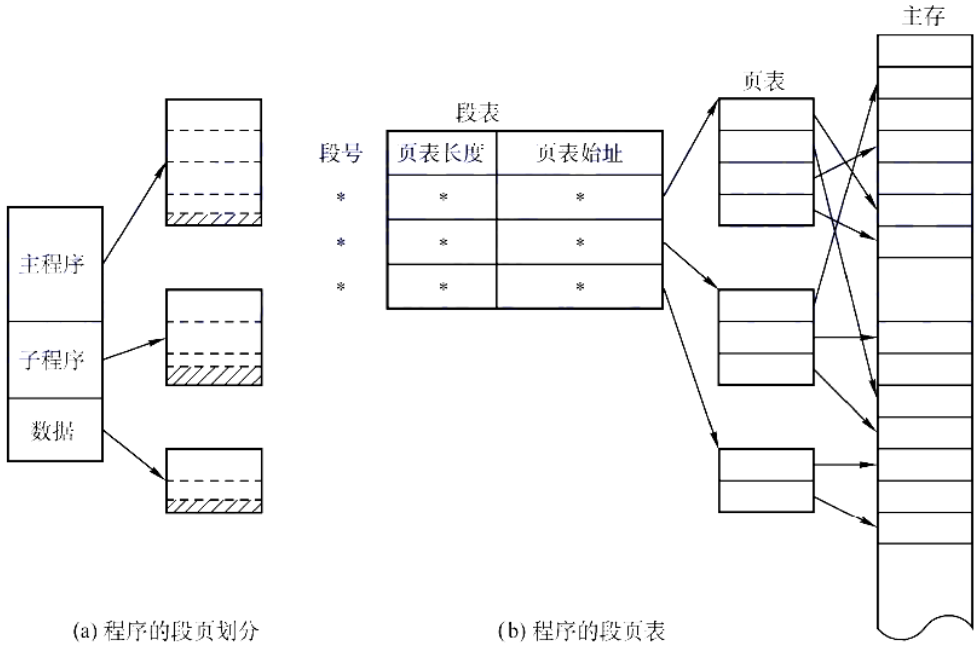


图 3.17　段页式管理方式

在段页式系统中，逻辑的物理地址：

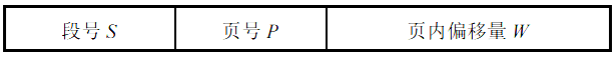


图 3.18　段页式系统的逻辑地址结构

系统为每个进程设置一张段表，每个段对应一个段表项，每个段表项至少包括段号（隐含），页表长度，页表地址

每个段有一张页表，每个页表项至少包括页号（隐含）和块号。

此时系统中还有一个段表寄存器，指出进程的段表起始地址和段表长度

段表寄存器和页表寄存器的作用都有两个：

- 在短表或页表中寻址
- 判断是否越界

在进行地址变换时，先通过短表查到页表地址，再通过页表找到物理块号，最后形成物理地址。

需要三次访存：

- 第一次，访存找到段表寄存器，通过段表寄存器找到段表地址
- 第二次，查询段表，通过段号，找到对应的页表地址
- 第三次，访存查询页表，找到对应的物理块号，

可以用快表来加快查找速度，关键字由段号，页号组成，值是对应的物理块号和保护码

二、虚拟内存管理

问题：

- 为什么引入虚拟地址
- 虚拟内存空间的大小由什么因素决定
- 虚拟内存是怎么解决问题的，会带来什么问题

1. 虚拟内存的基本概念

（1）传统存储管理方式的特征

之前讨论的内存管理策略都是为了将多个进程保存到内存中，以便进程多道程序设计。具有以下特征：

- 一次性：
  - 作业必须一次性全部装入内存后，才能开始运行，会导致两个问题
    - 当作业很大不能被全部装入内存，该作业无法运行
    - 当大量作业要求运行时，由于内存不能容纳所有作业，只能使少数作业先运行，导致程序并发度下降。
- 驻留性：
  - 作业被装入内存后，就一直驻留内存中，任何部分都不会被换出，直到作业运行结束
  - 运行中的进程会因为等待I/O而被阻塞，肯呢个处于长期等待状态

可知，许多程序运行中不用或暂时不用的程序（数据）占据了大量的内存空间，而一些需要运行的作业又无法装入运行，浪费了资源

（2）局部性原理

从广义上来讲，快表，页高速缓存，及虚拟内存技术都属于高速缓存技术，这个技术依赖的原理就是局部性原理

局部性原理适用于程序结构，数据结构，局部性原理表现在两个方面：

- **时间局部性：**
  - 程序内的某条指令一旦被执行，不久后该指令可能再次执行
  - 一个数据被访问过，一段时间后可能再次被访问
  - 因为存在着大量的循环
- **空间局部性：**
  - 一旦程序访问了某个存储地址，不久后，其周围的存储单元也将被访问，即程序在一段时间内访问的地址，可能集中在一个范围内
  - 因为指令通常是按顺序存放，执行的，数据一般按照向量，数组，表等形式簇集存储的
- 时间局部性通过将近来使用的指令和数据保存到高速缓存中，并使用高速缓存的层次结构实现。
- 空间局部性通常使用较大的高速缓存，并将**预取机制集成到高速缓存控制逻辑中实现。**
- 虚拟内存技术实际上**建立了“内存-外存”的两级存储器结构**，利用局部性原理实现高速缓存

（3）虚拟存储器的定义和特征

基于局部性原理，在程序装入时，仅需将程序当前运行要用到的少数页面（或段）装入内存，而将其余部分留在外存，便可以启动程序执行。

在程序执行过程中，**若访问信息不在内存，由操作系统将所需信息从外存调入内存，然后继续执行程序，这个过程就是请求调页（请求调段）。**

当内存空间不足时，由**操作系统将暂时不用的信息换出到外存，从而腾出空间存放要调入内存的信息，这个过程就是页面置换（段置换）**

虚拟存储器，并不存在，只是由于系统提供了部分装入，请求调入，置换功能后（对用户透明），给用户的感觉好像有一个比实际物理内存大的多的存储器，但是容量大只是一种错觉。

虚拟存储器的特征：

- **多次性：**
  - 无需在作业运行时一次性全部装入内存，而是允许分成多次调入内存
- **对称性：**
  - 在作业运行时无需一直常驻内存，而是允许在作业运行过程中，将那些暂不使用的程序和数据从内存调至外存的对换区（换出），以后需要时，再将他们从外存调入内存（换进）
- 虚拟性：
  - 逻辑上扩充了内存的容量，最重要特征，使用户看到的内存容量远大于实际容量

（4）虚拟内存技术的实现

虚拟内存的实现需要建立在离散分配的内存管理方式的基础上：

虚拟内存的实现有三种方式：

- 请求分页存储管理
- 请求分段存储管理
- 请求段页存储管理

不管哪种方式，都需要一定的硬件支持，一般的支持需要几个方面：

- 一定容量的内存和外存
- 页表机制（段表机制），作为主要数据结构
- 中断机构，当用户要访问的部分尚未调入内存时，产生中断
- 地址变换机构，逻辑地址到物理地址的变换

2. 请求分页管理方式

请求分页在基本分页系统的基础上，为支持虚拟存储器功能**增加了请求调页和页面置换功能**

作业执行中

- 当作业不再内存中时，通过请求调页将其从外存调入内存
- 当内存空间不足时，通过页面置换功能将内存中暂时用不到的页面换出到外存

为了实现请求分页，系统必须提供一定的硬件支持，除了内存及外存的计算机系统，还需要页表机制，缺页中断机构，地址变换机构。

（1）页表机制

在请求分页系统中，

- **为了实现请求调页**，操作系统需要知道每个页面是否已经调入内存。
  - 若未调入，需要知道该页在外存中的存放地址。
- **为了实现页面置换**，操作系统需要通过某些指标来决定换出哪个页面
  - 对于要换出的页面，需要知道其是否被修改过，以决定是否写回外存

为此，在请求页表项中增加了4个字段：

页　　号	物理块号	状态位 P	访问字段 A	修改位 M	外存地址

图 3.20　请求分页系统中的页表项

- 状态位P：标记该页是否已被 调入内存，供程序访问时参考
- 访问字段A：记录本页在一段时间内被访问的次数，或者记录本页最近有多久未被访问过，供置换算法换出页面时参考
- 修改位M：标记该页在调入内存后是否被修改过，以决定是否要写回外存
- 外存地址：该页在磁盘（程序文件/交换区）中的位置，用于缺页调页和换入/换出管理
- 物理块号：指的是页框编号，加上offset可以转换为物理地址

> **外存地址不一定只在"换出后"才有**
>
> - 有的系统会一直记录：这页对应的程序文件位置（可执行文件/映射文件中的位置）或交换区位置；
> - 一旦该页被换出到交换区（swap），外存地址可能会更新成交换区中的位置。
>
> 但在很多教材语境里，确实可以理解为：
> **这页不在内存时，它在外存（尤其是交换区）的位置。**

（2）缺页中断机构

请求分页系统中，当要访问的页面不在内存中时，产生一个缺页中断，请求操作系统的缺页中断处理程序处理。

- 缺页的进程阻塞，放入阻塞队列，调入完成后再唤醒，放回就绪队列。
- 若内存中有空闲页框，为进程分配一个页框，将所缺页面从外存装入页框，并修改页表中的表项
- 若没有空闲页框，则用页面置换算法选择一个页面淘汰，若改业在内存期间被修改过，则还要将其写回外存，未修改的不用写回外存

缺页中断，要保护CPU环境，分析中断原因，转入缺页中断处理程序，恢复CPU环境等几个步骤，与一般中断相比，有两个明显区别：

- 在指令执行期间而非一条指令执行完后产生和处理中断，属于内部异常
- 一条指令在执行期间，可能发生多次缺页中断

（3）地址变换机构

在基本分页系统的地址变换机构的基础上，为实现虚拟内存，增加了产生和处理缺页中断，及从页面中换出一页的功能。

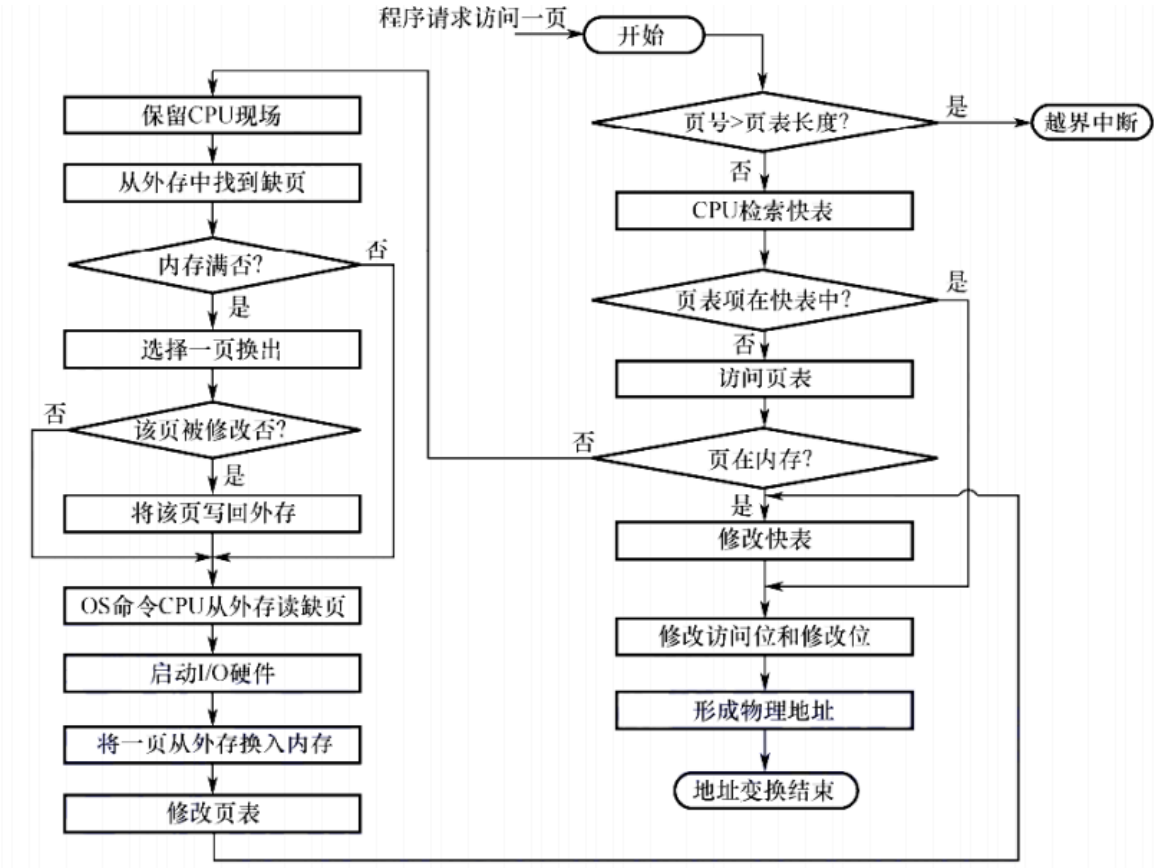


图 3.21　请求分页中的地址变换过程

请求分页的地址变换过程：

- **先检索快表：**
  - 若命中，取出该页的物理块号，修改页表项中的访问位，对于写指令，修改位置1
- **快表未命中，到页表中查找：**
  - 若找到：从相应表项取出物理块号，将该页表项写入快表，如果快表已满，采用某种策略替换
- **若页表中未找到：**
  - 需要进行缺页中断处理，请求系统将该页从外存换入内存，页面被调入内存后，由OS负责更新页表和快表，并获得物理块号
- **利用得到的物理块号和页内地址偏移拼接成物理地址，用该地址访存**

3. 页框分配

（1）驻留集大小

操作系统必须决定读取多少页，即给特定进程分配几个页框。

给一个进程分配的页框的集合，就是这个进程的驻留集。

需要考虑：

- 驻留集越小，驻留在内存中的进程就越多，可以提供多道程序的并发度，但是分配给每个进程的页框太少，会使缺页率高，CPU需要花费大量时间处理缺页
- 驻留集越大，当分配给进程的页框超过某个数量时，再增加，对缺页率改善不大，会浪费内存空间，并且导致多道程序并发度下降。

（2）内存分配策略

在请求分页系统中，可以采取两种**内存分配方式，**即**固定和可变分配策略。**

在进行**置换**时，可采用两种策略，即**全局和置换局部。**

于是可以组合成下面三种适用的策略：

a. 固定分配局部置换

- **固定分配：**
  - 为每个内存分配固定数量的物理块，在运行期间不再改变。
- **局部置换：**
  - 若进程在运行期间发生缺页，只能从分配给进程在内存中的页面选择一页换出，再将所缺页面调入，以保证分配给这个进程的内存空间不变
- **缺点：**
  - 难以确定为每个进程分配的物理块数量
  - 太少会频繁出现缺页中断，太多会降低CPU和其他资源利用率

> 全局置换比如会导致一个进程拥有的物理块数量变化，不能和固定分配结合

b. 可变分配全局置换

- **可变分配：**
  - 为每个进程分配一定数量的物理块，在运行期间可以根据情况适当的增加或减少
- **全局置换：**
  - 若进程在运行中发生缺页，则系统从空闲物理块队列中取出一块分配给改进程，并将所缺页面调入。
- **优缺点：**
  - 比固定分配局部置换更灵活，动态增加进程的物理块，
  - 弊端是可能盲目的给进程增加物理块，导致系统多道程序的并发能力下降

c. 可变分配局部置换

- **可变分配：**
  - 为每个进程分配一定量的物理块，可变
- **局部置换：**
  - 当发生缺页时，只允许该进程在内存的页面中选出一页换出，不影响其他进程的工作
- 当进程**频繁的发生缺页中断，**则系统再为该进程分配若干物理块，直到该进程的缺页率趋于适当程度
- 反之，如果进程的**缺页率特别低**，可以适当减少给进程的物理块，但不能引起缺页率的明显变化。
- **优缺点：**
  - 保证进程不会过多的调页的同时，保持了系统的多道程序并发能力
  - 需要更复杂的实现，更大的开销，但是对于频繁的换入、换出所浪费的计算机资源，这种牺牲是值得的。

（3）物理块调入算法

采用固定分配策略时，将系统中的空闲物理块分配给各个进程，可采用以下算法：

- **平均分配算法**
  - 将系统中所有可供分配的物理块均分给各个进程
- **按比例分配算法**
  - 根据进程大小按比例分配
- **优先权分配算法**
  - 为重要和紧急的进程分配较多的物理块

通常，将所有可分配的物理块分成两部分，一部分按比例分配给各个进程，一部分根据优先权分配。

（4）调入页面的时机

为确定系统将进程运行时所缺的页面何时调入内存，采用以下两种调页策略：

- **预调页策略：**
  - 根据局部性原理，一次调入若干个相邻的页面会比一次调入一页更高效，但若提前调入的页面都没有被访问，则又是低效的
  - 因此，可以预测不就后可能被访问的页面，将他们预先调入内存，但是目前预测成功率仅为50%，因此这种策略只适用于进程的首次调入。
- **请求调页策略**：
  - 进程在运行中需要访问的页面不在内存中，便提出请求， 由系统将所需的页面调入内存，比较容易实现
  - 缺点是每次仅调入一页，增加了磁盘的I/O开销

预调页就是运行前的调入，请求调页是运行期间的调入。

（5）从何处调入页面

请求分页系统中的外存分为两部分：

- 用于存放文件的**文件区**
- 用于存放对换页面的**对换区，也称交换区**

对换区采用离散分配方式，对换区采用离散分配方式，对换区的磁盘I/O速度比文件区的更快。

这样当发生缺页请求时，系统从何处将缺页调入内存就分为了三种情况：

- **系统拥有足够的对换区域：**
  - 可以全部从对换区调入所需页面，提高调页速度，为此，程序在运行前，需将该进程有关的文件从文件区复制到对换区
- **系统缺少足够的对换区空间：**
  - 凡是不会被修改的文件都直接从文件区调入，当换出这些页面时，由于不会被修改而不需要被换出，
  - 对于那些可能修改的部分，在将他们换出时必须放在对换区，以后需要时再从对换区调入。

> 没有被写的页面被置换出内存时不用写入对换区（swap），因为以后要用时可以直接从原文件区再读入，就是说，直接丢弃即可，只有可能被修改（变脏）的页面才必须写到对换区保存

- **UNIX方法：**

- 与进程有关的都放在文件区，未运行过的页面都从文件区调入，
- 曾经运行过但又被换出的页面，放在对换区，下次调入时从对换区调入。

进程请求的共享页面若被其他进程调入内存，则不需要再从对换区调入。

（6）如何调入页面

当进程所访问的页面不在内存中时（存在位为0），向CPU发出缺页中断，中断响应后转入中断处理程序。

- **该程序通过查找页表得到该页的物理块，**
  - 此时**若内存未满**，则启动磁盘I/O，将所缺页面调入内存，并修改页表。
  - **若内存已满，**则按照某种置换算法从内存中选出一页准备换出
    - 若该页**未被修改过**（修改位为0），不需要将页写回磁盘，直接丢弃
    - 若该页**被修改过**，写回磁盘
  - 将所缺页面调入内存，并修改页表中对应表项，设置其存在位为1
- 调入完成后，就可以利用修改过的页表形成要访问数据的内存地址

4. 页面置换算法

进程运行时，若其访问的页面不再内存中，需要将其调入，但内存没有空闲，需要从内存中调出一页，换出到外存，选择调出哪个页面的算法就叫做页面置换算法。

> 页面置换算法的特点，哪种可能导致Belady异常

（1）最佳OPT置换算法

选择淘汰的页面时以后永不使用的，**或者在最长时间内不再被访问的页面**，以便保证获得最低的缺页率

然而目前无法预测进程在内存中的若干页面哪个是**以后最长时间内不再被访问**的，因此算法无法实现，但可以用来评价其他算法。

- **最长时间不被访问**（下一次被访问的时间最晚）
  - 从当前时间开始看，很长一段时间都不会被访问，适合换出
- **以后访问次数最少**区分
- 可以举个例子：
  - A：下一次再很远的地方才访问，但之后会用很多次
  - B：下一次马上就用，但是之后几乎不用
  - **OPT应该换出A，是因为他的下一次最远，而不是因为次数少**

（2）先进先出（FIFO）页面置换算法

FIFO算法选择淘汰的页面时最早进入内存的页面。

实现简单，根据调入的先后顺序排成一个队列，需要换出的时候，选择队头的页面，但是该算法没有利用局部性原理，性能较差。

FIFO算法会产生当为进程分配的物理块增多，缺页次数不降反增的异常现象，成为Belady异常，只有FIFO算法会出现Belady异常。

（3）最近最久未使用（LRU）算法

选择淘汰最近最长时间未被使用的页面，认为过去一段时间内未被访问过的页面，在最近的将来也不会被访问。

为各个页面设置一个访问字段，记录页面自从上次被访问以来所经历的时间，淘汰页面时选择现有页面中值最大的页面。

LRU根据各页之前的使用情况来判断，最佳置换算法根据各页以后的使用情况来判断，前后无必然联系

- OPT性能最好，但无法实现
- FIFO实现简单，但性能差
- LRU性能好，接近OPT，但是实现需要寄存器和栈的硬件支持，开销大。

（4）时钟（CLOCK）置换算法

CLOCK 是一种实现成本低、效果接近 LRU 的页面置换算法。核心思想：用一个“环形队列 + 指针（钟表指针）”在页框中循环检查，利用“访问位（R 位）”判断页面近期是否被访问过。

a. 简单的CLOCK置换算法（也叫 Second-Chance，二次机会）

- **需要的硬件/数据**
  - 每个页表项（或页框记录）有一个 **访问位 R（Referenced）**
    - 页面被访问（读/写）时，硬件把 R 置 1
  - 把驻留内存的页框组织成一个环
  - 一个指针 hand 指向“当前候选页框
- **换页时的规则（找牺牲页）**

当需要换出一页时，从 hand 指向的页开始循环：

1. 若当前页 **R = 0**
   - 说明最近没被用过
   - **选它换出**
2. 若当前页 **R = 1**
   - 说明最近用过，给它“第二次机会”
   - 把 **R 置 0**，hand 向前移动到下一页
   - 继续检查

不断循环，直到找到 R=0 的页。

- **为什么能近似 LRU**
  - 最近被访问过的页 R=1 会被“放过一次”，同时 R 被清零
  - 如果它之后又被访问，R 会再次变 1，就更不容易被淘汰
  - 如果它之后一直不被访问，下一圈遇到它时 R=0，就会被淘汰
- **特点**
  - 优点：实现简单、开销小（只用一个访问位和一个指针）
  - 缺点：只区分“最近是否被访问过”，**不区分访问频率**；并且可能需要扫描多次才能找到可换出的页（但通常比纯 FIFO 好很多）

b. 改进型CLOCK置换算法（考虑“访问位 R + 修改位 M”）

改进型 CLOCK 的目标：**优先换出“既没被访问也没被修改”的页**，因为这种页换出最省事——不需要写回磁盘。

- **需要的硬件/数据**
  - **访问位 R**
    - **修改位 M（Dirty）**
      - 页面被写过时 M=1
  - 同样的环形队列和指针 hand
- **页面分类（常考）**
  - 按 (R, M) 把页分为四类：
    - **(0,0)**：没访问过、没修改过 —— 最优先换出（直接丢弃）
    - **(0,1)**：没访问过、但修改过 —— 次优（要写回磁盘）
    - **(1,0)**：访问过、没修改 —— 先不给换（刚用过）
    - **(1,1)**：访问过、且修改 —— 最不想换（刚用过还得写回）

典型选择策略（两趟扫描的写法最常见）

当需要置换时，从 hand 开始按顺序找：

第一趟：找 (0,0)

- 如果遇到 R=1 的页，先不换，通常只跳过（有的版本会顺便把 R 清 0，但常见是第二趟统一清）

若第一趟找不到：

第二趟：找 (0,1)，并在扫描过程中把所有 R=1 的页清成 0

- 这样做的效果是：把“最近访问过”的页在下一轮变成“未访问”，除非它在这期间再次被访问把 R 置回 1

如果仍然找不到（理论上经过清 R 后最终总能找到），就继续循环，后续会出现 (0,0) 或 (0,1)。

直观理解

- 先找**不需要写回**的干净页（0,0）
- 找不到再找**需要写回**的脏页（0,1）
- 对于最近访问过的页（R=1），先给机会，让它们变成 R=0 后再比较谁更适合被换出

特点

- 优点：相比简单 CLOCK，更倾向于换出“干净且冷”的页，减少磁盘写回
- 缺点：实现更复杂；最坏情况下仍可能扫描较多页

做题策略：

- 简单 CLOCK：
  - 只看 R，指针循环，遇到 R=1 清零跳过，遇到 R=0 换出
- 改进 CLOCK：
  - 看 (R,M)，优先找 (0,0)，再找 (0,1)，必要时把 R 清零让下一轮更容易选出“冷页”

5. 抖动和工作集

（1）抖动

在页面置换的过程中，最糟糕的情况是，刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出内存，这种频繁的页面调度行为**称为抖动或者颠簸**

抖动的根本原因：给每个进程分配的物理块太少，不能满足进程的正常运行的基本要求，致使每个进程在运行时频繁的出现缺页，必须请求系统将所缺页面调入内存。

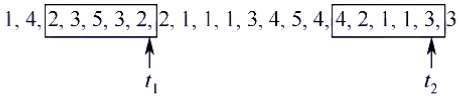
对磁盘的访问时间也急剧增加，造成每个进程的大部分时间都用于页面的换入换出，而不能做什么有效的工作，进而导致发生CPU的利用率急剧下降并区域0的情况

抖动的发生与系统为进程分配物理块（驻留集）有关

（2）工作集

工作集是指：某段时间间隔内，进程要访问的页面集合。

一般来说，工作集W可以由时间t和工作窗口尺寸来确定。



命题追踪 ▶ 工作集的应用分析（2016）

假设工作集窗口尺寸 Δ 设置为 5，则在 t_1 时刻，进程的工作集为{2, 3, 5}，t_2 时刻，进程的工作集为{1, 2, 3, 4}。实际应用中，工作集窗口会设置得很大，对于局部性好的程序，工作集大小一般会比工作集窗口 Δ 小很多。工作集反映了进程在接下来的一段时间内很有可能频繁访问的页面集合，因此驻留集大小不能小于工作集大小，否则进程在运行过程中会频繁缺页。

6. 页框回收

（1）页面缓冲算法

在页式虚拟存储系统中，页面的换入、换出开销会对系统性能产生很大影响

影响页面换入，换出效率的因素主要包括：

- 对页面进行页面置换的算法
  - 好的苏娜发可以使有较低的缺页率，减少页面换入换出的开销
- 将已修改页面写回磁盘的频率
  - 若每当一个页面要被换出时，就将他写回磁盘，则会导致很大的磁盘I/O开销
- 将磁盘内容读入内存的频率
  - 若进程的每次访问都需要将磁盘内容读入内存，会导致很大的磁盘I/O频率，增加页面换入的开销。

页面缓冲算法

- 是在原有的页面置换算法的基础上，增设已修改页面链表，保存已修改且需要被换出的页面，等被换出的页面数量达到一定值时，再一起换出到磁盘，减少页面换出的开销。

页面缓冲算法的特点：

- 显著降低页面换入换出的频率，使磁盘的I/O开销大为减少，进而减少页面换入，换出的开销。

- 由于换入换出的开销减小，当采用较简单的置换策略如FIFO时，不需要特殊硬件的支持，实现简单。

为降低页面换入换出频率，增加两个链表在内存中：

- **空闲页面链表：**（这里边的页框可以立刻调配给新调入的页）
  - 空闲页框链表，当进程需要读入一个页面时，便从空闲页面链表中取链首的页框并装入该页
  - 当有一个为被修改的页面换出时，实际上并不把他换出到磁盘，而是将他所在的页框挂在空闲链表的表尾（就是我们之前说的直接丢）
  - 这些改在空闲链表内且未被修改的页面中有数据，若以后某个进程需要这些数据，从空闲链表上将他们取下，从而避免磁盘读入的操作，减少页面换入开销

> 这里指的是名义上空闲，里边还有旧数据，如之后访问到同一页，只要他还没有被其他页覆盖，就能直接复用，不用再从磁盘读。

- **修改页面链表：**（这里边的页框暂时不要分配，因为里边是脏页）
  - 当进程要将一个已修改的页面换出时，系统并不立即将他换出到磁盘，而是将他所在的页框挂在修改页面链表的尾部
  - 这样做的目的是降低将已修改页面写回磁盘的频率

直接写盘的问题是：

- 每次换出都写磁盘，写操作频繁、慢
- 写盘会阻塞很多事情（尤其机械盘时代更明显）
- 有时这个脏页过一会儿又会被访问，如果立刻写盘等于白写一次

所以系统常做的是：

- 先把脏页从进程的工作集里“拿掉”（逻辑上不再属于该进程可用页）
- 但**不马上写回**，而是挂到“修改页面链表”里
- 由后台线程（写回守护/页写回进程）**异步、批量**写回磁盘

这叫“延迟写 / 异步写回”。

> “挂在修改页面链表尾部”到底意味着什么？
>
> 意味着：
>
> - 这个页框现在属于一种“缓存状态”：**内容是脏的、需要写回**
> - 它**不再映射给原进程使用**（页表会更新：该页不在内存 or 标记为换出中）
> - 但它也**还不能算真正空闲**，因为内容必须先安全落盘（或写到 swap），否则一旦被覆盖就丢数据

它主要降低的是 **写磁盘的频率和同步等待开销**，不是说“换出动作完全消失”。

具体收益：

- **批量写**：攒多个脏页一起写，磁盘效率更高
- **异步写**：进程被换出脏页时不必等写盘完成（减少阻塞）
- **写合并**：同一页如果在短时间内多次被修改，延迟写只写“最后结果”，避免多次写
- **更平滑**：后台慢慢写，不让前台频繁抖动

（2）页框回收

将物理页面换出的过程，即为页框回收。

当系统课分配的资源不足时，就必须回收一些页框，但不是所有页框都是可以回收的。

属于内核的大部分页框（内核栈，内核数据段，大部分内核使用的页框不能回收）

而由进程使用的页框（进程代码段，进程数据段，进程堆栈，进程访问文件时映射的文件页，进程间共享内存使用的页框）大部分可回收。

7. 内存映射文件

内存映射文件(Memory-Mapped Files)是操作系统向应用程序提供的一个系统调用，与虚拟内存有些像，在磁盘文件与进程的虚拟地址空间之间建立映射关系。

进程通过该系统调用，将文件当做内存中的一个大字符数组来访问，而不通过文件I/O操作来访问，更便利，磁盘文件的读写由操作系统负责，对进程是透明的。

在映射进程的页面时，不会实际读入文件的内容，而只在访问页面时次啊被每次一页地读入。

当进程退出或关闭文件映射时，所有改动的页面才会被写回磁盘文件。

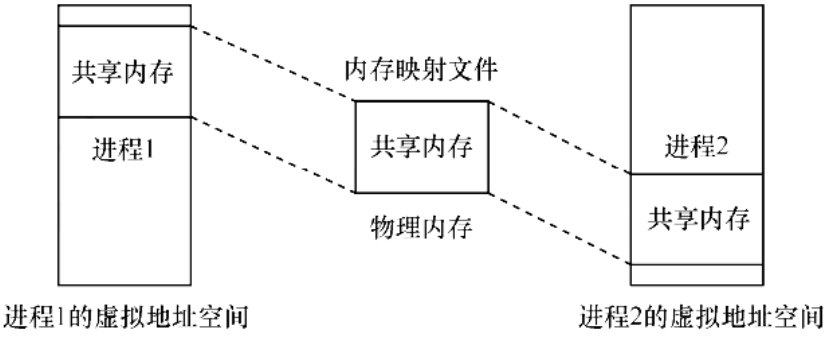


图 3.28　采用内存映射 I/O 的共享内存

进程可以通过共享内存来通信，共享内存很多时候就是通过映射到相同文件到通信进程的虚拟地址空间来实现的。

当多个进程映射到一个文件时，各个进程的虚拟地址空间都相对独立，但操作系统将对应的这些虚拟地址空间映射到相同的物理内存（用页表实现）

一个进程在共享内存上完成了写操作，此刻当另一个进程在映射到这个文件的虚拟地址空间执行读操作时，就立刻看到上一个进程写操作的结果。

内存映射的好处：

- 使编程简单，已建立映射的文件，只需要按照访问内存的方式进行读写
- 方便多个进程共享同一个磁盘文件

8. 虚拟存储器性能影响因素

缺页率是影响虚拟存储器性能的主要因素。

缺页率又收到页面大小，分配给进程的物理块数，页面置换算法，以及程序的编址方法的影响

- **页面大小：**
    - **页面大：**
        - 根据局部性原理，页面较大缺页率则较低

        - 虽然可以减少页表长度，但会使页内碎片增大
    - 页面小
        - 缺页率相对较高
        - 一方面减少了内部碎片，有利于提供内存利用率
        - 另一方面，每个进程有更多的页面，导致页表过长
- **置换算法：**
    - 好的置换算法可让进程在运行过程中有较低的缺页率
    - 选择LRU,CLOCK等置换算法，将未来有可能访问的页面尽可能的留在内存中，有利于提高页面访问速度。
- **分配给进程的物理块数：**
    - 越多
        - 缺页率低，但有阈值，但降低并发度
    - 越少：
        - 提高并发度，但是可能有抖动
- **写回磁盘的频率：**
    - 每次换出修改的页面都写回，每次都启动磁盘，效率低
    - 建立已修改的换出页面链表，数量到一定值时，一起写回，减少磁盘I/O次数，即减少页面换出开销
    - 若对这些页面再次访问，直接从链表中访问，无需从外存调入。
- **程序局部化程度：**
    - 越高，执行时缺页率越低
    - 若存储时采用的是按行存储，则访问时要采用相同的访问方式，避免按列访问导致缺页率过高的情况

9. 地址翻译

设某系统满足：

- 有一个TLB和一个data cache
- 存储器以字节为编址单位
- 虚拟地址14位
- 物理地址12位
- 页面大小64B
- TLB为思路组相联，共有16个条目
- data Cache是物理寻址，直接映射的，行大小为4B，共有16组