第六章 存储管理

- 存储管理是指存储器资源（主要指内存并涉及外存）的管理。
  - 存储器资源的组织（如内存的组织方式）
  - 地址变换（逻辑地址与物理地址的对应关系维护）
  - 虚拟存储的调度算法

6.1 引言

返回

6.1.1 存储组织

- 存储器的功能是保存数据，存储器的发展方向是高速、大容量和小体积。
  - 内存在访问速度方面的发展：DRAM、SDRAM、SRAM等；
  - 硬盘技术在大容量方面的发展：接口标准、存储密度等；
- 存储组织是指在存储技术和CPU寻址技术许可的范围内组织合理的存储结构。
  - 其依据是访问速度匹配关系、容量要求和价格。
  - “寄存器-内存-外存”结构
  - “寄存器-缓存-内存-外存”结构；
- 微机中的存储层次组织：
  - 访问速度越慢，容量越大，价格越便宜；
  - 最佳状态应是各层次的存储器都处于均衡的繁忙状态（如：缓存命中率正好使主存读写保持繁忙）；

返回

存储层次结构

- 快速缓存：
  - Data Cache
  - TLB(Translation Lookaside Buffer)
- 内存：DRAM, SDRAM等；
- 外存：软盘、硬盘、光盘、磁带等；

6.1.2 存储管理的功能

- 存储分配和回收：分配和回收算法及相应的数据结构。
- 地址变换：
  - 可执行文件生成中的链接技术
  - 程序加载(装入)时的重定位技术
  - 进程运行时硬件和软件的地址变换技术和机构
- 存储共享和保护：
  - 代码和数据共享
  - 地址空间访问权限（读、写、执行）
- 存储器扩充：存储器的逻辑组织和物理组织；
  - 由应用程序控制：覆盖；
  - 由OS控制：交换（整个进程空间），虚拟存储的请求调入和预调入（部分进程空间）

返回

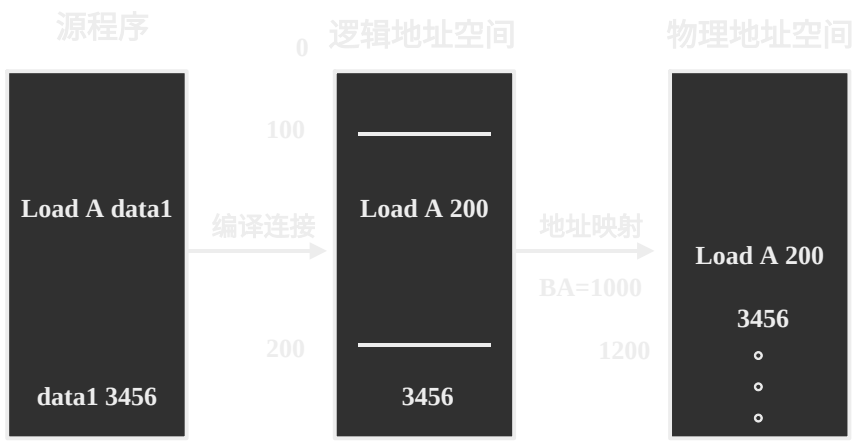
6.1.3 重定位方法

- 重定位：在可执行文件装入时需要解决可执行文件中地址（指令和数据）和内存地址的对应。由操作系统中的装入程序loader来完成。
- 程序在成为进程前的准备工作
  - 编辑：形成源文件(符号地址)
  - 编译：形成目标模块(模块内符号地址解析)
  - 链接：由多个目标模块或程序库生成可执行文件(模块间符号地址解析)
  - 装入：构造PCB，形成进程(使用物理地址)
- 重定位方法：
  - 绝对装入
  - 可重定位装入
  - 动态装入

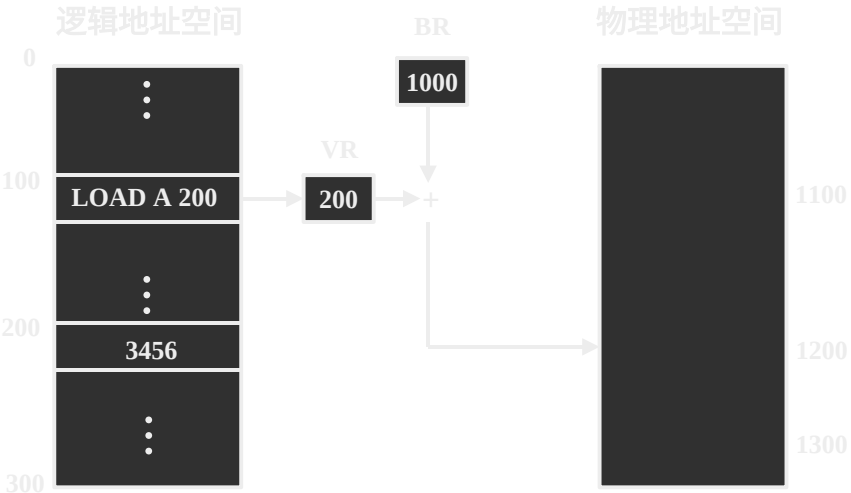
返回

1. 逻辑地址、物理地址和地址映射

- 逻辑地址（相对地址，虚地址）：用户的程序经过汇编或编译后形成目标代码，目标代码通常采用相对地址的形式。
  - 其首地址为0，其余指令中的地址都相对于首地址来编址。
  - 不能用逻辑地址在内存中读取信息。
- 物理地址（绝对地址，实地址）：内存中存储单元的地址。物理地址可直接寻址。
- 地址映射：将用户程序中的逻辑地址转换为运行时由机器直接寻址的物理地址。
  - 当程序装入内存时, 操作系统要为该程序分配一个合适的内存空间，由于程序的逻辑地址与分配到内存物理地址不一致, 而CPU执行指令时，是按物理地址进行的，所以要进行地址转换。



逻辑地址、物理地址和地址映射



地址映射

2. 绝对装入(absolute loading)

在可执行文件中记录内存地址，装入时直接定位在上述(即文件中记录的地址)内存地址。

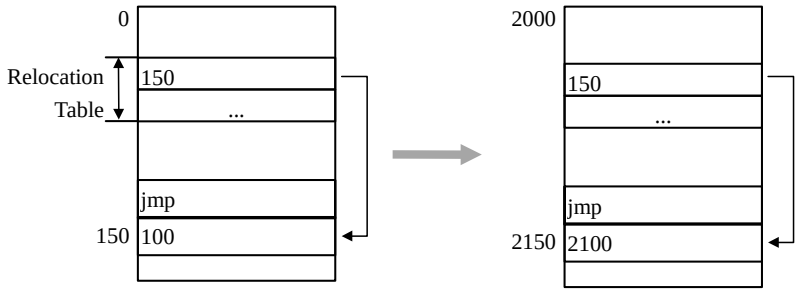
- 优点：装入过程简单。
- 缺点：过于依赖于硬件结构，不适于多道程序系统。

3. 可重定位装入(relocatable loading)

在可执行文件中，列出各个需要重定位的地址单元和相对地址值。当用户程序被装入内存时，一次性实现逻辑地址到物理地址的转换，以后不再转换（一般在装入内存时由软件完成）。即：装入时根据所定位的内存地址去修改每个重定位地址项，添加相应偏移量。

- 优点：不需硬件支持，可以装入有限多道程序（如MS DOS中的TSR）。
- 缺点：一个程序通常需要占用连续的内存空间，程序装入内存后不能移动。不易实现共享。

可执行文件在内存中的重定位



- 说明：重定位表中列出所有修改的位置。如：重定位表的150表示相对地址150处的内容为相对地址(即100为从0起头的相对位置)。在装入时，要依据重定位后的起头位置(2000)修改相对地址。
  - 重定位修改：重定位表中的150->绝对地址2150(=2000+150)
  - 内容修改：内容100变成2100(=100+2000))。

4. 动态装入(dynamic run-time loading)

在可执行文件中记录虚拟内存地址，装入和执行时通过硬件地址变换机构，完成虚拟地址到实际内存地址的变换。

- 优点：
  - OS可以将一个程序分散存放于不连续的内存空间，可以移动程序，有利用实现共享。
  - 能够支持程序执行中产生的地址引用，如指针变量（而不仅是生成可执行文件时的地址引用）。
- 缺点：需要硬件支持（通常是CPU），OS实现较复杂。它是虚拟存储的基础。

6.1.4 链接

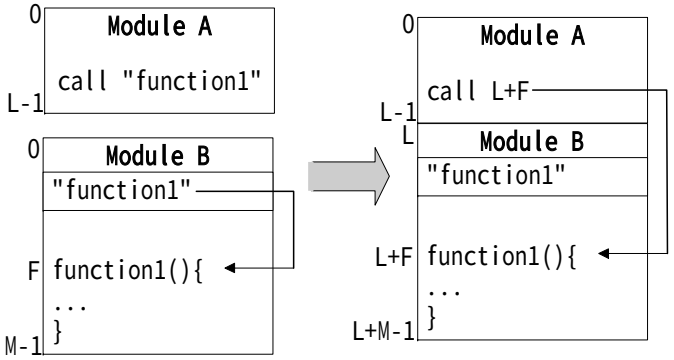
链接是指多个目标模块在执行时的地址空间分配和相互引用。

返回

6.1.4.1 链接方法

1. 静态链接(static-linking)

静态链接是在生成可执行文件时进行的。在目标模块中记录符号地址(symbolic address)，而在可执行文件中改写为指令直接使用的数字地址。



返回

2. 动态链接(dynamic-linking)

在装入或运行时进行链接。通常被链接的共享代码称为动态链接库(DLL, Dynamic-Link Library)或共享库(shared library)。

- 优点
  - 共享：多个进程可以共用一个DLL，节省内存，减少文件交换。
  - 部分装入：一个进程可以将多种操作分散在不同的DLL中实现，而只将当前操作相应的DLL装入内存。
  - 便于局部代码修改：即便于代码升级和代码重用；只要函数的接口参数（输入和输出）不变，则修改函数及其DLL，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，就可以适应多种使用环境和提供不同功能。如：不同的显示卡只需厂商为其提供特定的DLL，而OS和应用程序不必修改。
- 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。

6.1.4.2 Windows NT动态链接库

1. 构造动态链接库

DLL是包含函数和数据的模块，它的调用模块可为EXE或DLL，它由调用模块在运行时加载；加载时，它被映射到调用进程的地址空间。在VC中有一类工程用于创建DLL。

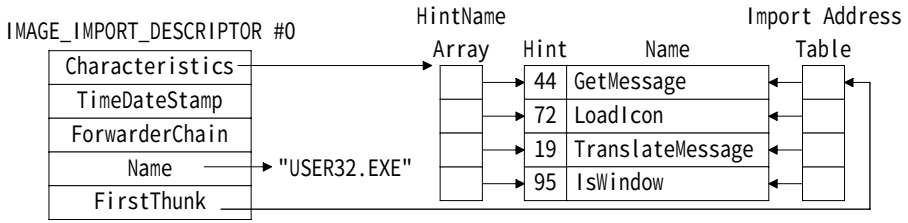
- 库程序文件 .C：相当于给出一组函数定义的源代码；
- 模块定义文件 .DEF：相当于定义链接选项，也可在源代码中定义；如：DLL中函数的引入和引出(dllimport和dllexport)。
- 编译程序利用 .C文件生成目标模块 .OBJ
- 库管理程序利用 .DEF文件生成DLL输入库 .LIB和输出文件 .EXP
- 链接程序利用 .OBJ和 .EXP文件生成动态链接库 .DLL。

返回

2. DLL的装入方法

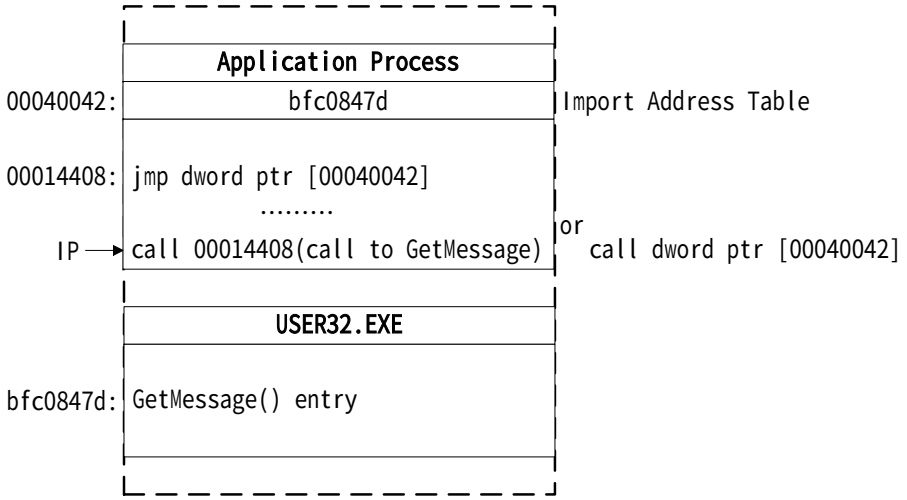
- 装入时动态链接(load-time)：
  - 在编程时显式调用某个DLL函数，该DLL函数在可执行文件中称为引入(import)函数。
  - 链接时需利用 .LIB文件。在可执行文件中为引入的每个DLL建立一个IMAGE_IMPORT_DESCRIPTOR结构。
  - 在装入时由系统根据该DLL映射在进程中的地址改写Import Address Table中的各项函数指针。Hint是DLL函数在DLL文件中的序号，当DLL文件修改后，就未必指向原先的DLL函数。在装入时，系统会查找相应DLL，并把它映射到进程地址空间，获得DLL中各函数的入口地址，定位本进程中对这些函数的引用；

装入时动态链接过程



注：Import Address Table是在装入时依据DLL模块的加载位置确定。

DLL函数的调用过程



- 运行时动态链接(run-time)：在编程时通过LoadLibrary（给出DLL名称，返回装入和链接之后该DLL的句柄）, FreeLibrary, GetProcAddress（其参数包括函数的符号名称，返回该函数的入口指针）等API来使用DLL函数。这时不再需要引入库（import library）。
  - LoadLibrary或LoadLibraryEx把可执行模块映射到调用进程的地址空间，返回模块句柄；
  - GetProcAddress获得DLL中特定函数的指针，返回函数指针；
  - FreeLibrary把DLL模块的引用计数减1；当引用计数为0时，拆除DLL模块到进程地址空间的映射；

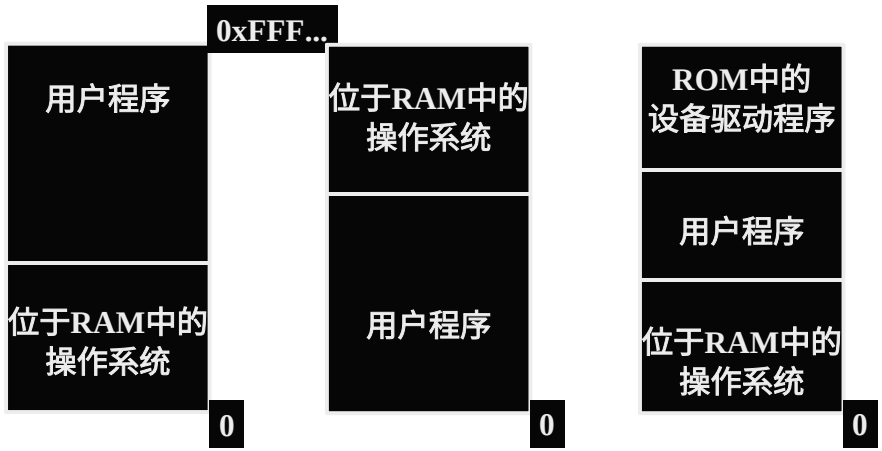
运行时动态链接的例子

```
HINSTANCE hInstLibrary;//模块句柄定义
DWORD (WINAPI *InstallStatusMIF)(char*, char*, char*,
char*, char*, char*, char*, BOOL);//函数指针定义
if (hInstLibrary = LoadLibrary("ismif32.dll"))//映射
       {
         InstallStatusMIF = (DWORD (WINAPI
*)(char*,char*,char*, char*, char*, char*, char*,
BOOL)) GetProcAddress(hInstLibrary,
"InstallStatusMIF");//获得函数指针
            if (InstallStatusMIF)
            {
        if (InstallStatusMIF(“office97”, “Microsoft”,
“Office 97”, “999.999”, “ENU”, “1234”, ”Completed
successfully”, TRUE) !=0)//调用DLL模块中的函数
                  {
                  }
            }
            FreeLibrary(hInstLibrary);//拆除映射
       }
```

6.2 单一连续区存储管理

- 内存分为两个区域：系统区，用户区。应用程序装入到用户区，可使用用户区全部空间。
- 最简单，适用于单用户、单任务的OS。
- 优点：易于管理。
- 缺点：对要求内存空间少的程序，造成内存浪费；程序全部装入，很少使用的程序部分也占用内存。

返回



单一连续区存储管理

6.3 分区存储管理

返回

6.3.1 原理

- 把内存分为一些大小相等或不等的分区(partition)，每个应用进程占用一个或几个分区。操作系统占用其中一个分区。
- 特点：适用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。
- 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：占用分区之间难以利用的空闲分区（通常是小空闲分区）。

返回

- 分区的数据结构：分区表，或分区链表
  - 可以只记录空闲分区，也可以同时记录空闲和占用分区
  - 分区表中，表项数目随着内存的分配和释放而动态改变，可以规定最大表项数目。
  - 分区表可以划分为两个表格：空闲分区表，占用分区表。从而减小每个表格长度。空闲分区表中按不同分配算法相应对表项排序。

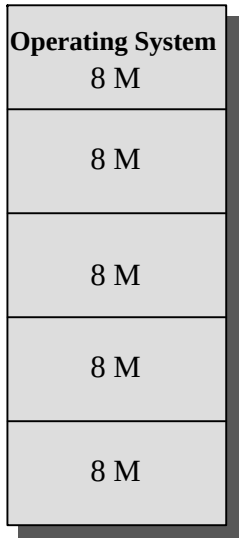
- 内存紧缩(compaction)：将各个占用分区向内存一端移动。使各个空闲分区聚集在另一端，然后将各个空闲分区合并成为一个空闲分区。
  - 对占用分区进行内存数据搬移占用CPU时间
  - 如果对占用分区中的程序进行"浮动"，则其重定位需要硬件支持。
  - 紧缩时机：每个分区释放后，或内存分配找不到满足条件的空闲分区时

6.3.2 固定分区(fixed partitioning)

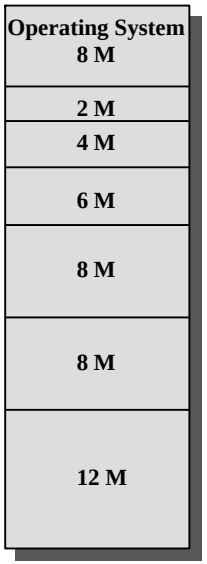
把内存划分为若干个固定大小的连续分区。

- 分区大小相等：只适合于多个相同程序的并发执行（处理多个类型相同的对象）。
- 分区大小不等：多个小分区、适量的中等分区、少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。

返回



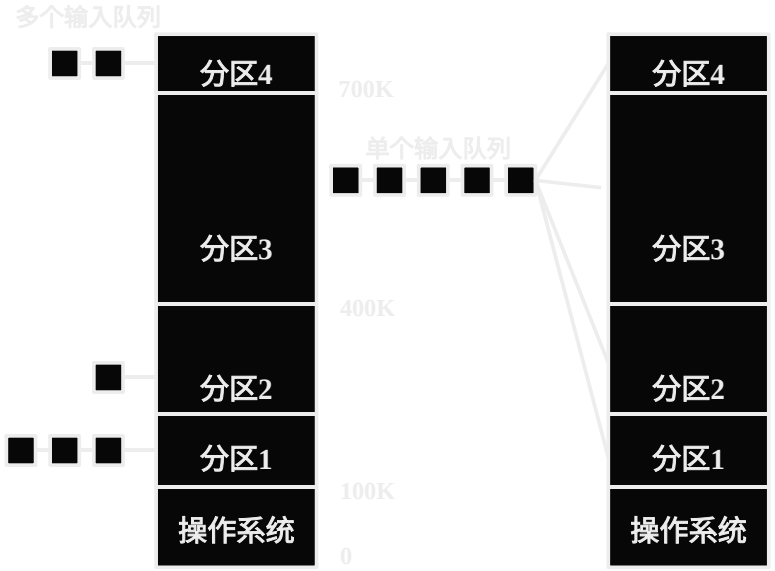
固定分区(大小相同)



固定分区(多种大小)

- 优点：易于实现，开销小。
- 缺点：
  - 内碎片造成浪费
  - 分区总数固定，限制了并发执行的程序数目。
- 可以和覆盖、交换技术配合使用。
- 采用的数据结构：分区表－－记录分区的大小和使用情况

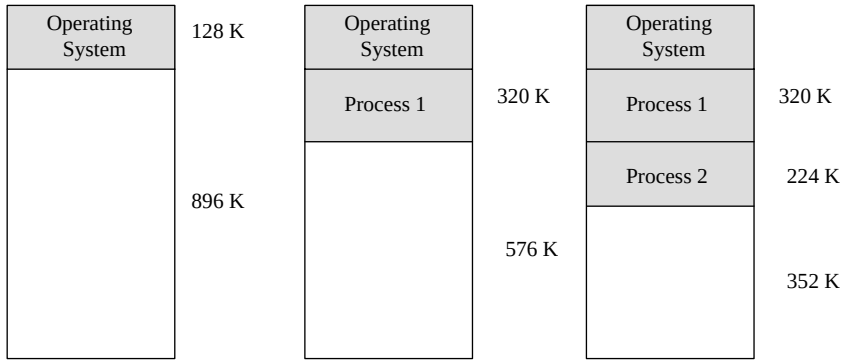
固定分区的进程队列：多队列适用于大小不等的固定分区；单队列适用于大小相同的固定分区，也可用于大小不等的情况。

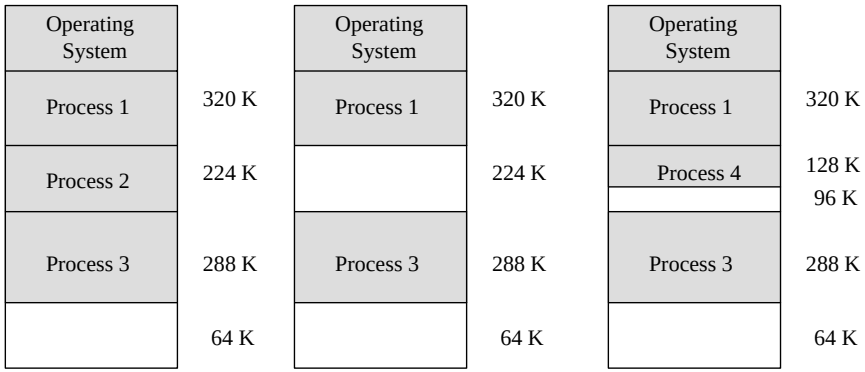


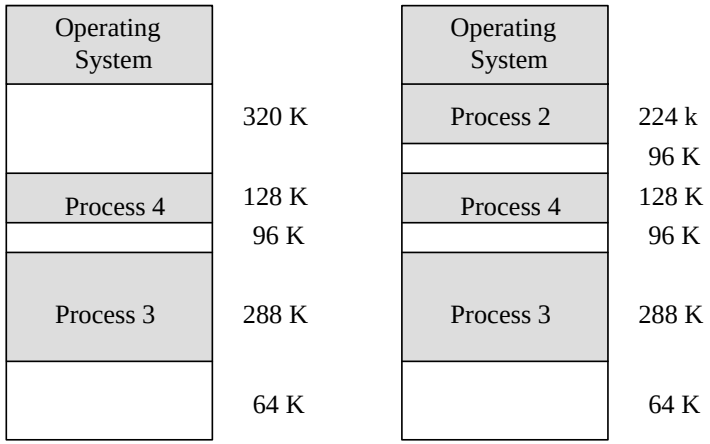
6.3.3 动态分区(dynamic partitioning)

- 动态创建分区：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。
- 优点：没有内碎片。
- 缺点：有外碎片；如果大小不是任意的，也可能出现内碎片。

返回





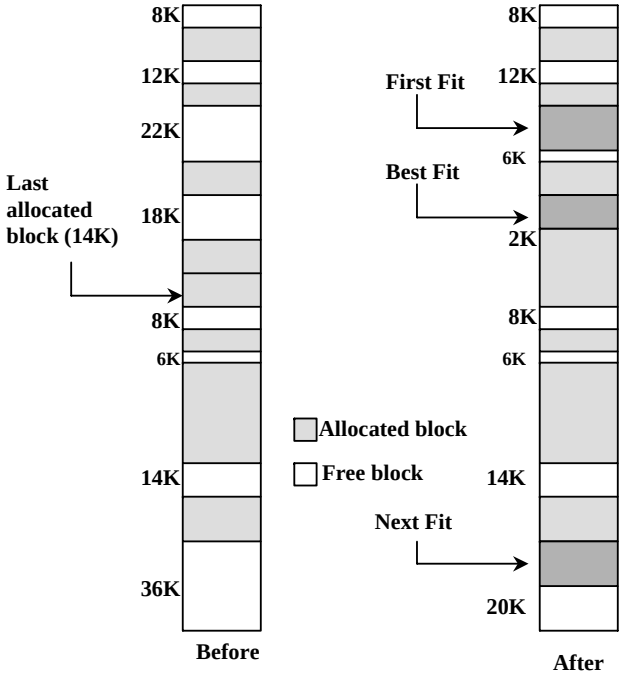


6.3.4 分区分配算法

- 分区分配算法：寻找某个空闲分区，其大小需大于或等于程序的要求。若是大于要求，则将该分区分割成两个分区，其中一个分区为要求的大小并标记为“占用”，而另一个分区为余下部分并标记为“空闲”。分区的先后次序通常是从内存低端到高端。
- 分区释放算法：需要将相邻的空闲分区合并成一个空闲分区。(这时要解决的问题是：合并条件的判断和合并时机的选择)

返回

- 最先匹配法(first-fit)：按分区的先后次序，从头查找，找到符合要求的第一个分区
  - 该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。
  - 但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。
- 下次匹配法(next-fit)：按分区的先后次序，从上次分配的分区起查找（到最后分区时再回到开头），找到符合要求的第一个分区
  - 该算法的分配和释放的时间性能较好，使空闲分区分布得更均匀，但较大的空闲分区不易保留。
- 最佳匹配法(best-fit)：找到其大小与要求相差最小的空闲分区
  - 从个别来看，外碎片较小，但从整体来看，会形成较多外碎片。较大的空闲分区可以被保留。
- 最坏匹配法(worst-fit)：找到最大的空闲分区
  - 基本不留下小空闲分区，但较大的空闲分区不被保留。



6.3.5 MS DOS中的分区存储管理

- DOS提供动态分区管理；
- 通过DOS功能调用int 21h，支持分区的创建(48h)、释放(49h)和改变分区大小(4Ah)
- 设置或查询分区的分配策略(58h)：
  - 最先匹配法、
  - 最佳匹配法、
  - 最后匹配法（last-fit，从内存高端向低端查找）

返回

数据结构为单向链表。每个分区以一个MCB(Memory Control Block)结构开始，MCB占16个字节，按段边界对齐（起始地址可被16整除）

PSP(Program Segment Prefix)：起进程控制块的作用，其起始地址可作为进程ID

```
typedef struct {
   BYTE type;          /* 'M' = in chain; 'Z' = at end  */
   WORD owner;         /* PSP of the owner process, 0 = free */
   WORD size;          /* in 16-byte paragraphs */
   BYTE unused[3];
   BYTE dos4[8];       /* program name(DOS 4.x)    */
 } MCB;
```

6.4 覆盖和交换技术

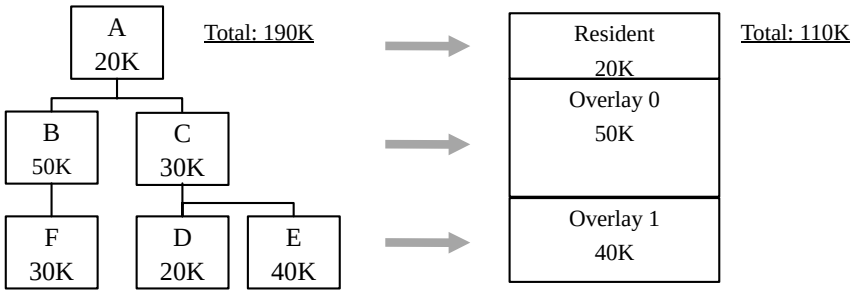
返回

6.4.1 覆盖(overlay)

- 引入：其目标是在较小的可用内存中运行较大的程序。常用于多道程序系统，与分区存储管理配合使用。
- 原理：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。
  - 将程序的必要部分（常用功能）的代码和数据常驻内存；
  - 可选部分（不常用功能）在其他程序模块中实现，平时存放在外存中（覆盖文件），在需要用到时才装入到内存；
  - 不存在调用关系的模块不必同时装入到内存，从而可以相互覆盖。(即不同时用的模块可共用一个分区)

返回

覆盖技术



注：另一种覆盖方法：(100K)
- A(20K)占一个分区：20K；
- B(50K)、D(20K)和E(40K)共用一个分区：50K；
- F(30K)和C(30K)共用一个分区：30K；

- 缺点：
  - 编程时必须划分程序模块和确定程序模块之间的覆盖关系，增加编程复杂度。
  - 从外存装入覆盖文件，以时间延长来换取空间节省。
- 实现：函数库（操作系统对覆盖不得知），或操作系统支持

6.4.2 交换(swapping)

- 引入：多个程序并发执行，可以将暂时不能执行的程序送到外存中，从而获得空闲内存空间来装入新程序，或读入保存在外存中而目前到达就绪状态的进程。交换单位为整个进程的地址空间。常用于多道程序系统或小型分时系统中，与分区存储管理配合使用。又称作"对换"或"滚进/滚出(roll-in/roll-out)"；
  - 程序暂时不能执行的可能原因：处于阻塞状态，低优先级（确保高优先级程序执行）；
- 原理：暂停执行内存中的进程，将整个进程的地址空间保存到外存的交换区中（换出swap out），而将外存中由阻塞变为就绪的进程的地址空间读入到内存中，并将该进程送到就绪队列（换入swap in）。

返回

- 优点：增加并发运行的程序数目，并且给用户提供适当的响应时间；编写程序时不影响程序结构
- 缺点：对换入和换出的控制增加处理机开销；程序整个地址空间都进行传送，没有考虑执行过程中地址访问的统计特性。
- 考虑的问题：
  - 程序换入时的重定位；
  - 减少交换中传送的信息量，特别是对大程序；
  - 对外存交换区空间的管理：如动态分区方法；

6.5 页式和段式存储管理

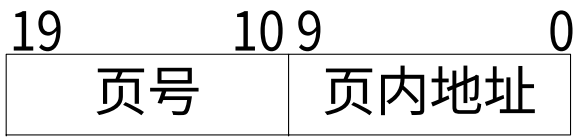
页式和段式存储管理是通过引入进程的逻辑地址，把进程地址空间与实际存储位置分离，从而增加存储管理的灵活性。

返回

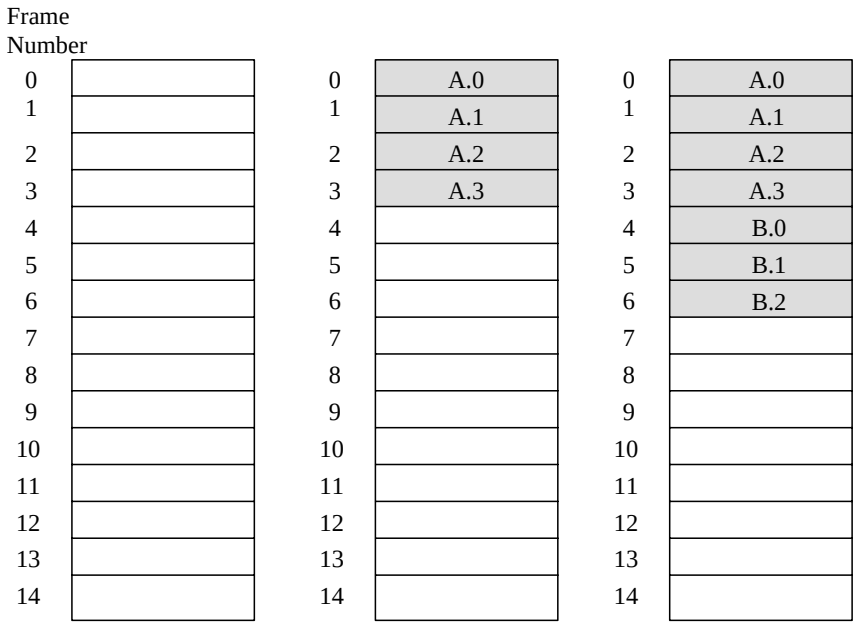
6.5.1 简单页式(simple paging)

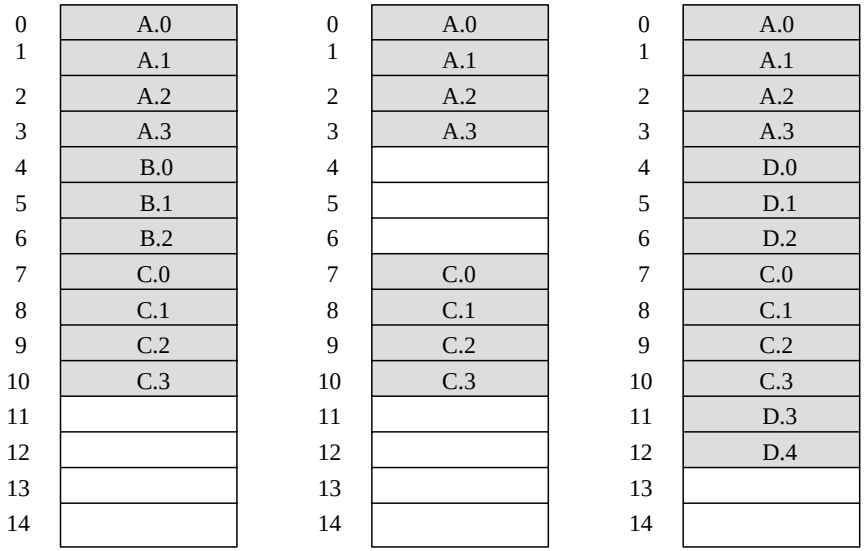
1. 简单页式管理的基本原理

将程序的逻辑地址空间和物理内存划分为固定大小的页或页面(page or page frame)，程序加载时，分配其所需的所有页，这些页不必连续。需要CPU的硬件支持。



返回

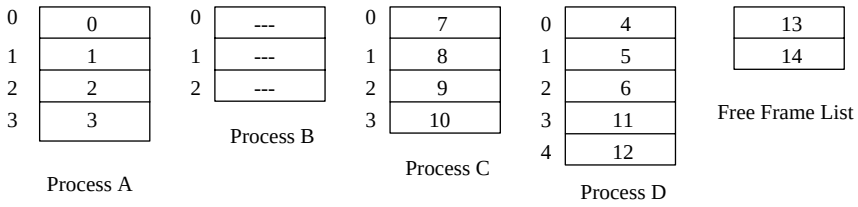




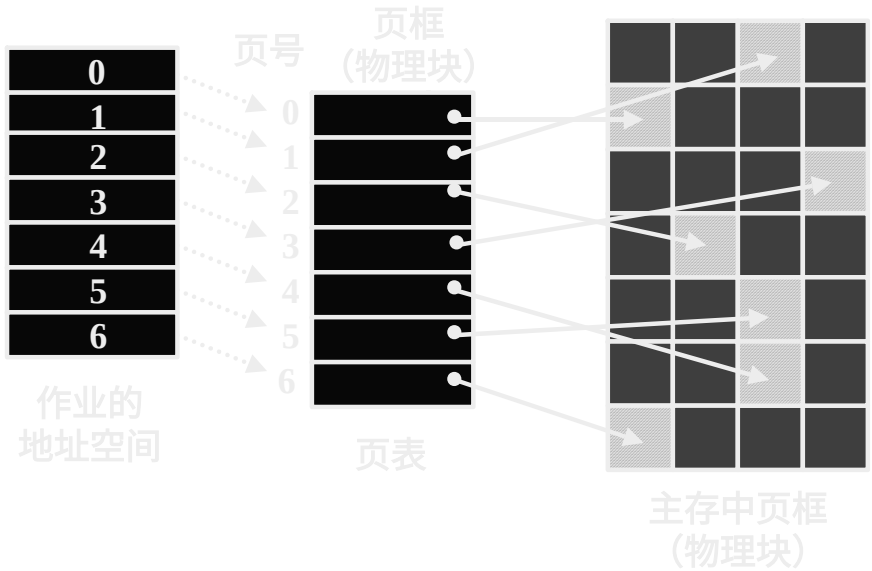
- 优点：
  - 没有外碎片，每个内碎片不超过页大小。
  - 一个程序不必连续存放。
  - 便于改变程序占用空间的大小。即随着程序运行而动态生成的数据增多，地址空间可相应增长。
- 缺点：程序全部装入内存。

2. 简单页式管理的数据结构

- 进程页表：每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序；
  - 逻辑页号（本进程的地址空间）－>物理页面号（实际内存空间）；
- 物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用状况。
  - 数据结构：位示图，空闲页面链表；
- 请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换，也可以结合到各进程的PCB里；

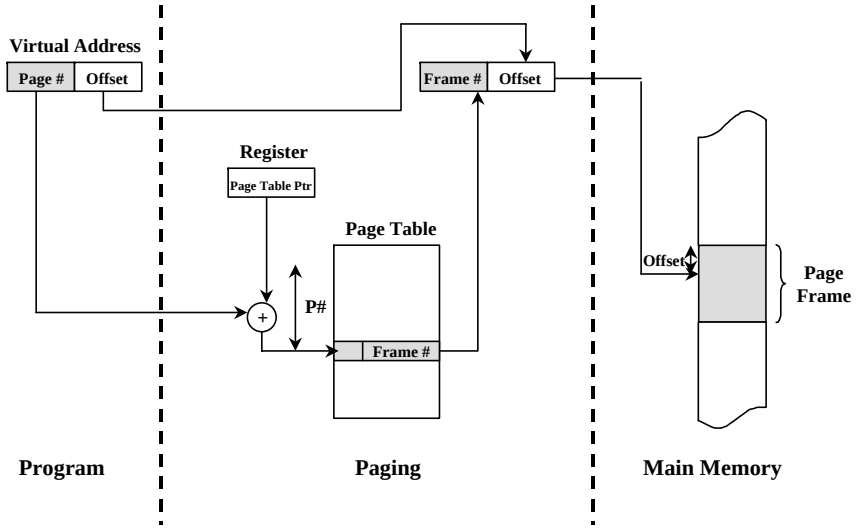


进程页表

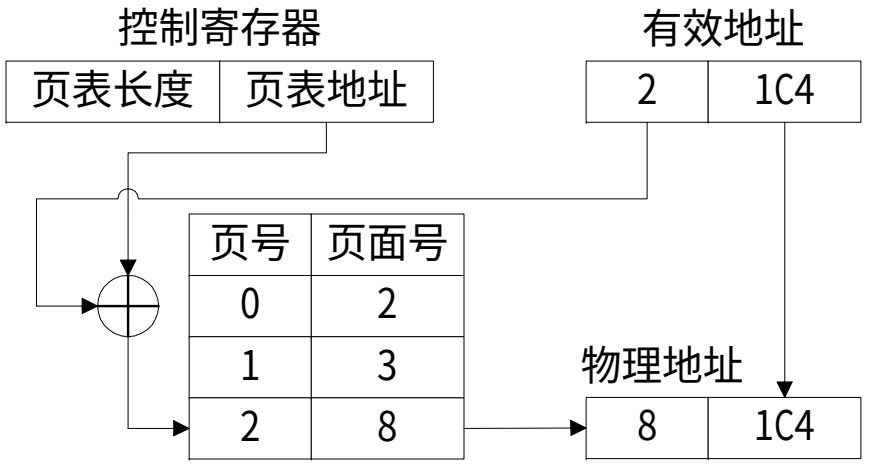


3. 简单页式管理的地址变换

- 指令所给出地址分为两部分：逻辑页号，页内偏移地址－>查进程页表，得物理页号－>物理地址
  - 为缩短查找时间，可以将页表从内存装入到关联存储器(TLB, Translation Lookaside Buffer)，按内容查找(associative mapping)，即逻辑页号－>物理页号



页式地址变换



页式地址变换举例

5. 页的大小

- 通常是：几KB到几十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。
  - 和目前计算机的物理内存大小有关：4MB到256MB，不太大。

6.5.2 简单段式(simple segmentation)

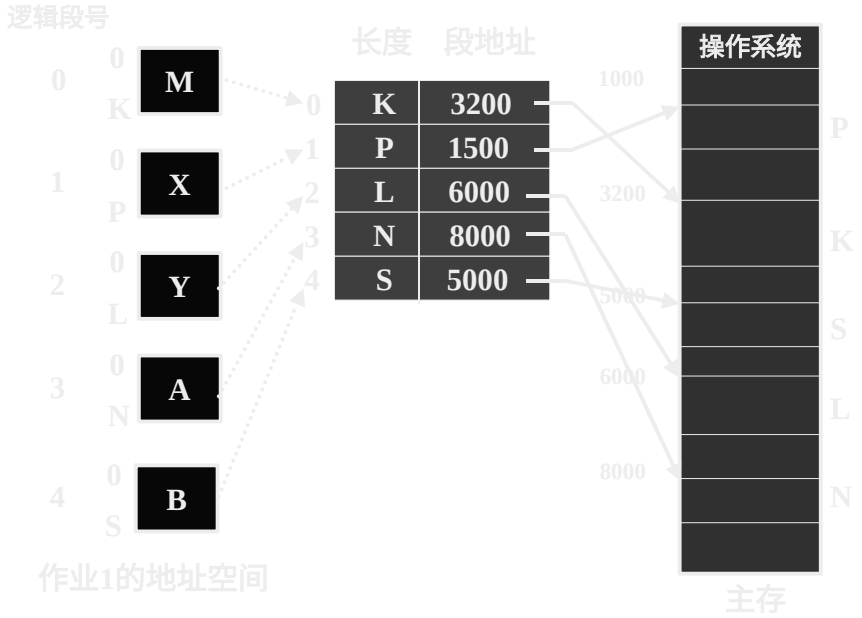
页式管理是把内存视为一维线性空间；而段式管理是把内存视为二维空间，与进程逻辑相一致。

1. 简单段式管理的基本原理

将程序的地址空间划分为若干个段(segment)，程序加载时，分配其所需的所有段（内存分区），这些段不必连续；物理内存的管理采用动态分区。需要CPU的硬件支持。

返回

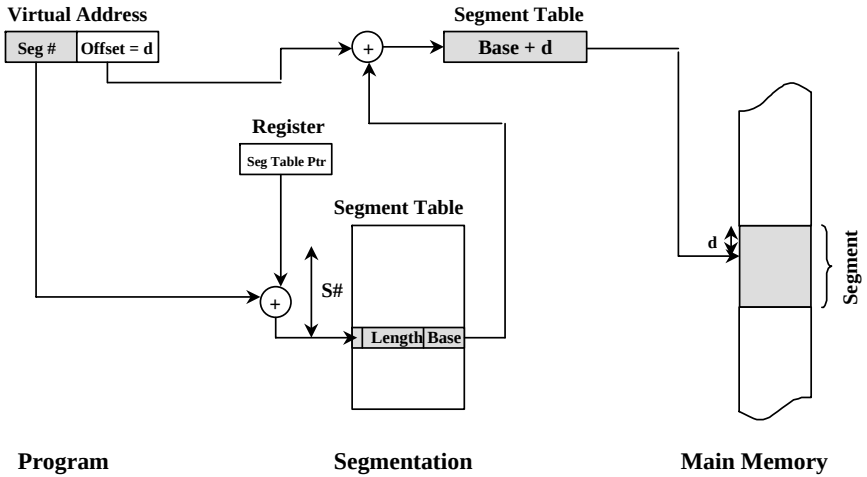
- 程序通过分段(segmentation)划分为多个模块，如代码段、数据段、共享段。
  - 可以分别编写和编译
  - 可以针对不同类型的段采取不同的保护
  - 可以按段为单位来进行共享，包括通过动态链接进行代码共享
- 优点：
  - 没有内碎片，外碎片可以通过内存紧缩来消除。
  - 便于改变进程占用空间的大小。
- 缺点：
  - 进程全部装入内存。

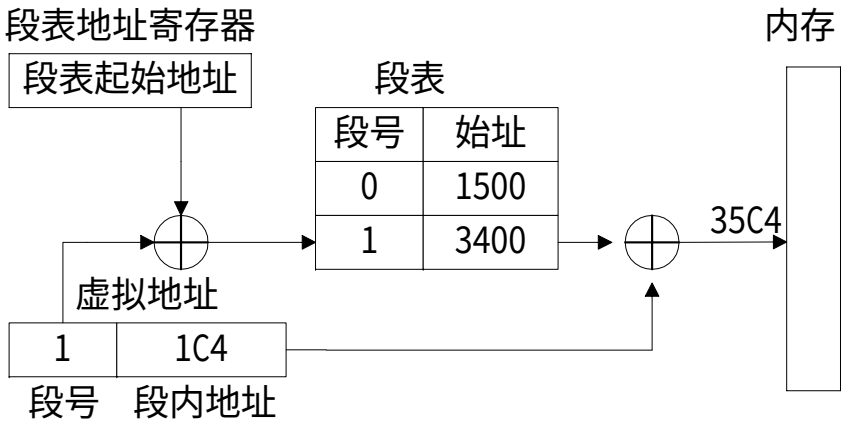


2. 简单段式管理的数据结构

- 进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址(base address)和段长度
- 系统段表：系统内所有占用段
- 空闲段表：内存中所有空闲段，可以结合到系统段表中

3. 简单段式管理的地址变换



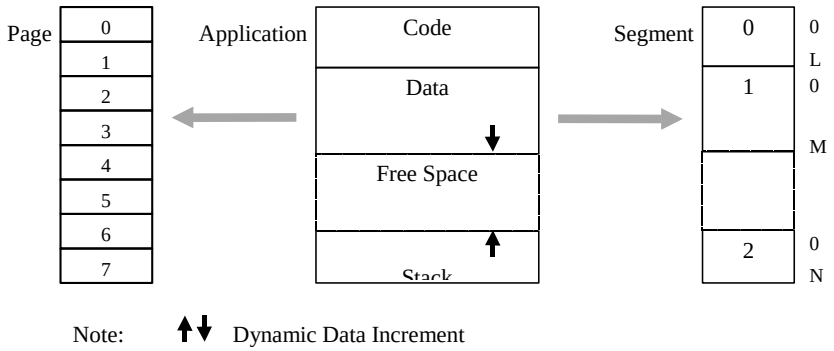


段式地址变换举例

6.5.3 页式管理和段式管理的比较

- 分页是出于系统管理的需要，分段是出于用户应用的需要。
  - 一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处。
- 页大小是系统固定的，而段大小则通常不固定。
- 逻辑地址表示：
  - 分页是一维的，各个模块在链接时必须组织成同一个地址空间；
  - 分段是二维的，各个模块在链接时可以每个段组织成一个地址空间。
- 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。

返回



页式管理与段式管理的比较

6.6 虚拟存储器(VIRTUAL MEMORY)

返回

6.6.1 局部性原理

- 局部性原理(principle of locality)：指程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。还可以表现为：
  - 时间局部性：一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内；
  - 空间局部性：当前指令和邻近的几条指令，当前访问的数据和邻近的数据都集中在一个较小区域内。

返回

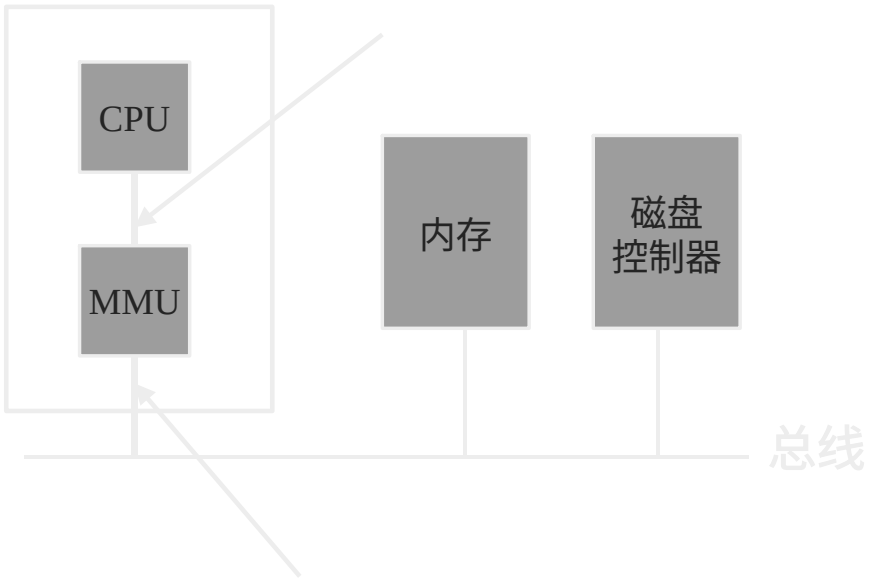
- 局部性原理的具体体现
  - 程序在执行时，大部分是顺序执行的指令，少部分是转移和过程调用指令。
  - 过程调用的嵌套深度一般不超过5，因此执行的范围不超过这组嵌套的过程。
  - 程序中存在相当多的循环结构，它们由少量指令组成，而被多次执行。
  - 程序中存在相当多对一定数据结构的操作，如数组操作，往往局限在较小范围内。

6.6.2 虚拟存储器的原理

1. 虚拟存储的基本原理

- 在程序装入时，不必将其全部读入到内存，而只需将当前需要执行的部分页或段读入到内存，就可让程序开始执行。
- 在程序执行过程中，如果需执行的指令或访问的数据尚未在内存（称为缺页或缺段），则由处理器通知操作系统将相应的页或段调入到内存，然后继续执行程序。
- 另一方面，操作系统将内存中暂时不使用的页或段调出保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段。只需程序的一部分在内存就可执行。

返回



处理机中的存储管理单元

2. 引入虚拟存储技术的好处

- 大程序：可在较小的可用内存中执行较大的用户程序；
- 大的用户空间：提供给用户可用的虚拟内存空间通常大于物理内存(real memory)
- 并发：可在内存中容纳更多程序并发执行；
- 易于开发：与覆盖技术比较，不必影响编程时的程序结构

3. 虚拟存储技术的特征

- 不连续性：物理内存分配的不连续，虚拟地址空间使用的不连续（数据段和栈段之间的空闲空间，共享段和动态链接库占用的空间）
- 部分交换：与交换技术相比较，虚拟存储的调入和调出是对部分虚拟地址空间进行的；
- 大空间：通过物理内存和快速外存相结合，提供大范围的虚拟地址空间
  - 总容量不超过物理内存和外存交换区容量之和

6.6.3 虚拟存储技术的种类

返回

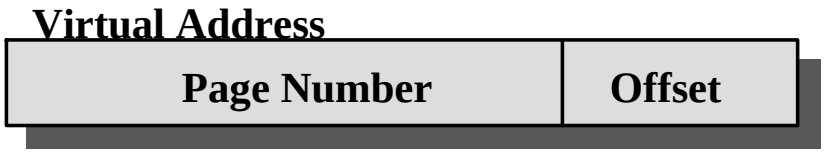
6.6.3.1 虚拟页式(virtual paging)

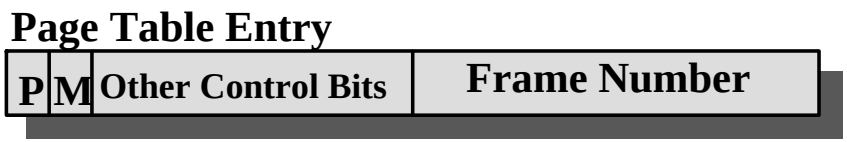
在简单页式存储管理的基础上，增加请求调页和页面置换功能。

1. 对进程页表的修改

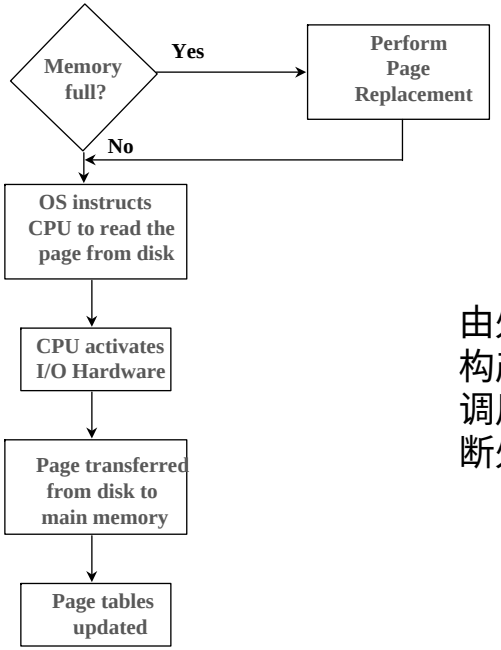
- 需要在进程页表中添加若干项
  - 标志位：存在位（present bit，内存页和外存页），修改位(modified bit)
  - 访问统计：在近期内被访问的次数，或最近一次访问到现在的时间间隔
  - 外存地址

返回





虚拟页式的进程页表

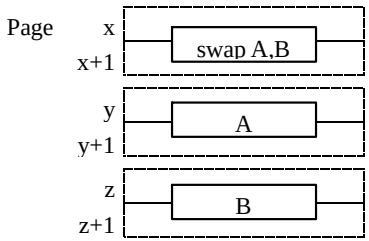


2. 缺页中断

由处理器的地址变换机构产生缺页中断，然后调用操作系统提供的中断处理例程。

缺页中断的特殊性

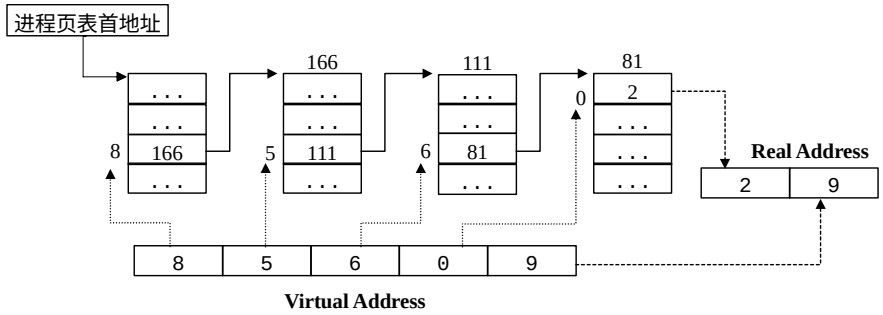
- 缺页中断在指令执行期间产生和进行处理，而不是在一条指令执行完毕之后。所缺的页面调入之后，重新执行被中断的指令。
- 一条指令的执行可能产生多次缺页中断，如：swap A, B而指令本身和两个操作数A, B都跨越相邻外存页的分界处，则产生6次缺页中断。



3. 多级页表

虚拟地址空间很大而每页比较小，则进程页表太长。采用两级或多级页表。在两级页表时，指令所给出的地址分为三部分：页表号，页号，偏移地址。如SUN SPARC处理器支持三级页表，Motorola 68030支持四级页表。

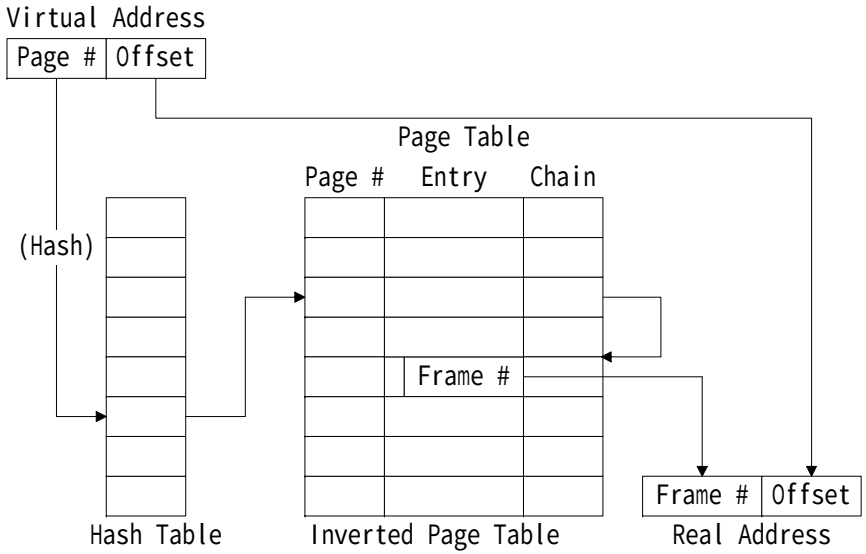
- 为缩短查找时间，多级页表中的每级都可以装入到关联存储器（即页表的高速缓存）中，并按照cache的原理进行更新。
- 多级页表结构中，指令所给出的地址除偏移地址之外的各部分全是各级页表的页表号或页号，而各级页表中记录的全是物理页号，指向下级页表或真正的被访问页。



多级页表地址映射

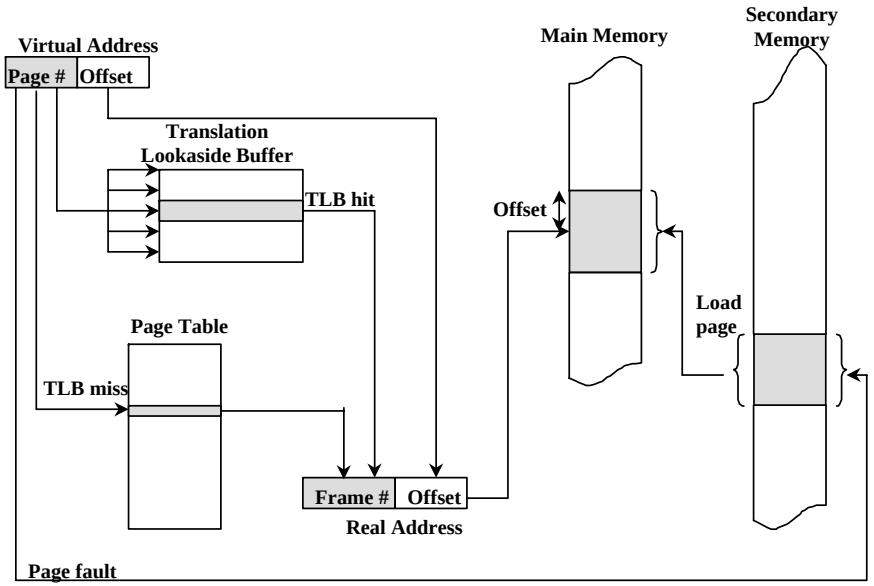
5. 反置页表(inverted page table)

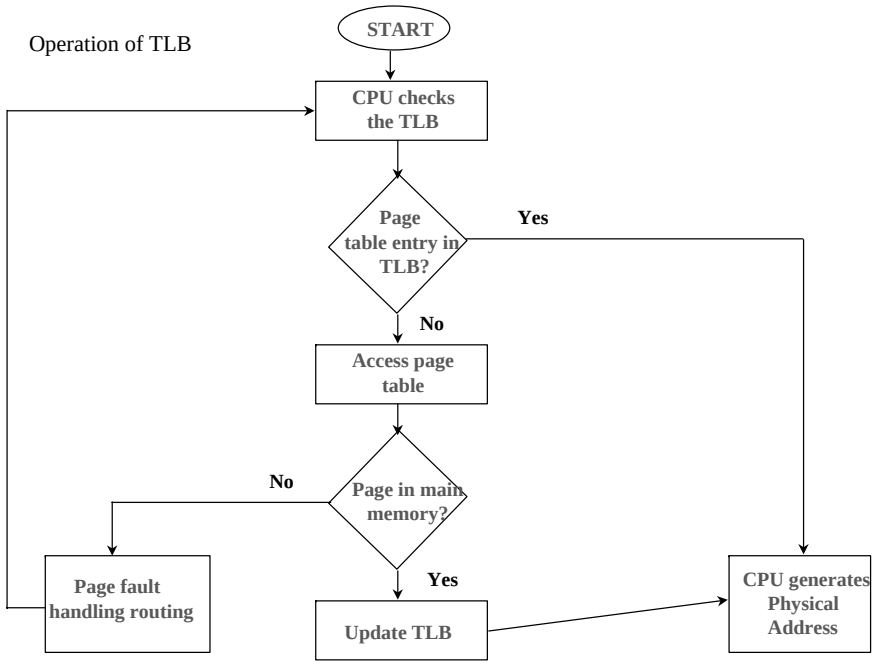
- 反置页表不是依据进程的逻辑页号来组织，而是依据该进程在内存中的物理页面号来组织（即：按物理页面号排列）。
- 每个进程一个反置页表，通过哈希表(hash table)查找可由逻辑页号得到物理页面号。虚拟地址中的逻辑页号通过哈希表指向反置页表中的表项链头（因为哈希表可能指向多个表项），得到物理页面号。
- 反置页表的大小只与物理内存的大小相对关，与逻辑空间大小和进程数无关。
- 如PowerPC, IBM AS/4000



反置页表

5. 关联存储器TLB





6. 缺页率(page fault rate)

缺页率表示“缺页次数 / 内存访问次数”(比率)或“缺页的平均时间间隔的倒数”；

- 缺页率的影响因素
  - 页面大小：
    - 页面很小－>每个进程的内存页较多，通过调页很快适应局部性原理的要求，缺页率低；
    - 页面很大－>进程使用的大部分地址空间都在内存，缺页率低；
    - 页面中等大小－>局部性区域只占每页的较小部分，缺页率高(不能很快适应，也不能大部分在内存)。
  - 分配给进程的页面数目：
    - 数目越多－>缺页率越低。
    - 页面数目的下限，应该是一条指令及其操作数可能涉及的页面数目的上限，以保证每条指令都能被执行。

- 发展趋势：采用多种页面大小
  - 内存增大
  - －>应用程序增大，而局部性下降（面向对象技术和多线程使数据和指令流分散）
  - －>TLB命中率下降，成为性能瓶颈
  - －>增加TLB容量，则成本上升；增加页面大小，则缺页率上升；
  - 采用多种页面大小
- 针对不同规模的数据结构采用不同的页面大小。
  - 代码段是大规模的
  - 每个线程的栈是小规模的
  - 处理器的例子有R4000, DEC Alpha和SUN SuperSPARC。其中R4000支持7种页面大小，从4KB到16MB

7. 磁盘I/O访问时间与页面大小

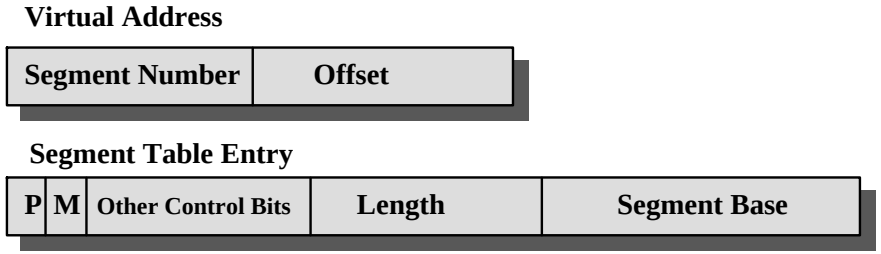
由旋转等待时间和读写时间组成，其中前者占80~90%（如一个柱面由40个扇区组成，则旋转等待时间大约占19/20=95%）。因此，采用较大的页面，相应交换区中由多个磁盘扇区组成存储单位，可以提高从交换区调页的I/O效率。

6.6.3.2 虚拟段式(virtual segmentation)

在简单段式存储管理的基础上，增加请求调段和段置换功能。

- 需要在进程页表中添加若干项：
  - 标志位：存在位(present bit)，修改位(modified bit/dirty bit)，增长位（该段是否增长过，在虚拟页式中没有该位）
  - 访问统计：如使用位(use bit)
  - 存取权限：如读R，写W，执行X
  - 外存地址
- 地址变换和缺段中断：指令和操作数必定不会跨越在段边界上

返回



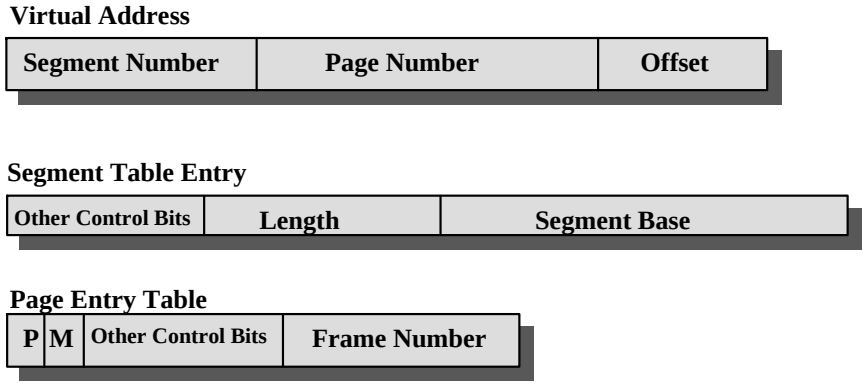
虚拟段式管理的段表

6.6.3.3 段页式(combined paging and segmentation)

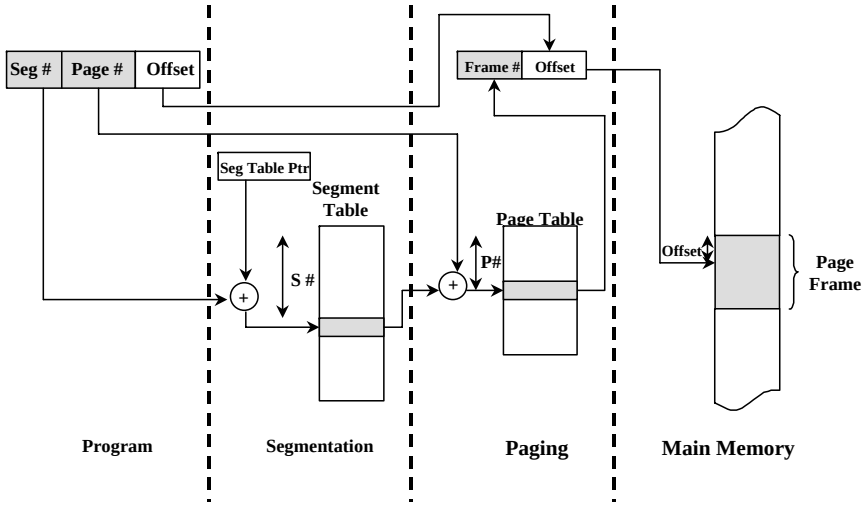
是虚拟页式和虚拟段式存储管理的结合。

- 存储管理的分配单位是：段，页
- 逻辑地址的组成：段号，页号，页内偏移地址。
- 地址变换：先查段表，再查该段的页表。缺段中断和缺页中断。

返回



虚拟段页式管理中的段表和页表



段页式地址变换

6.6.4 存储保护和共享

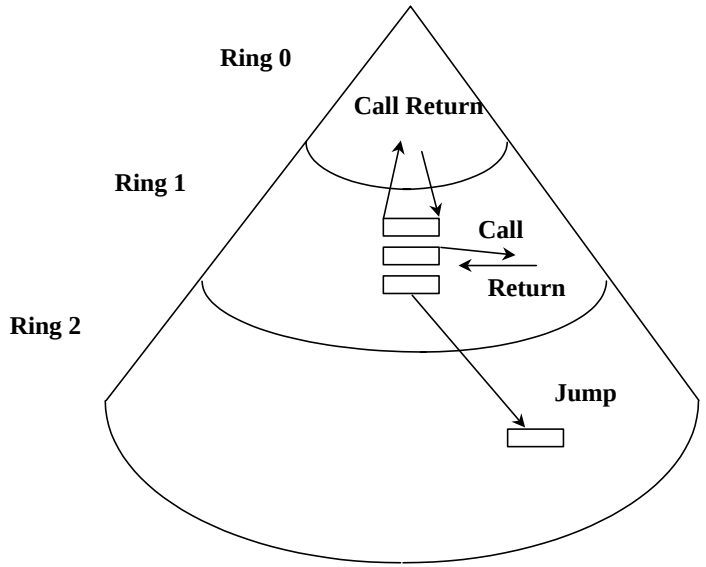
1. 存储保护

由于存储保护检查是针对每个存储访问操作进行的，必须由相应的处理器硬件机构支持。

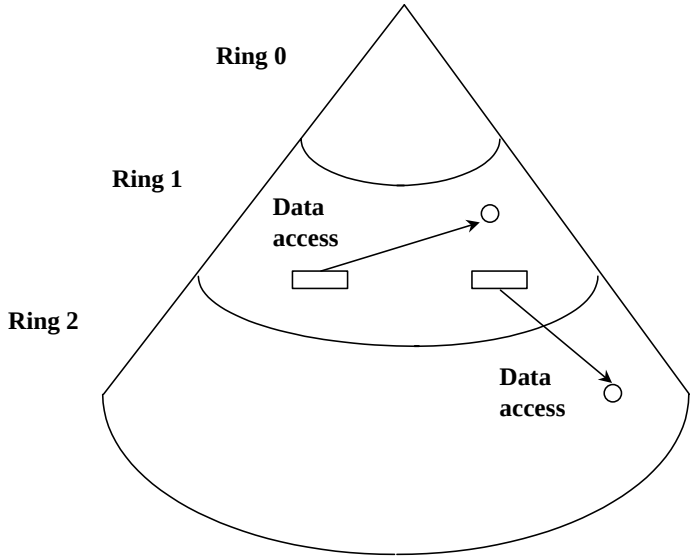
- 存储保护的目的：
  - 保护系统程序区不被用户侵犯（有意或无意的）
  - 不允许用户程序读写不属于自己地址空间的数据（系统区地址空间，其他用户程序的地址空间）

返回

- 存储保护类型
  - 界限保护（上界寄存器/下界寄存器或基址寄存器/限长寄存器）：所有访问地址必须在上下界之间；每个进程都有自己独立的进程空间，如果一个进程在运行时所产生的地址在其地址空间之外，则发生地址越界。当程序要访问某个内存单元时，由硬件检查是否允许，如果允许则执行，否则产生地址越界中断，由操作系统进行相应处理。
  - 访问方式保护（保护键）：通过保护键匹配来判断存储访问方式是否合法。对于允许多个进程共享的存储区域，每个进程都有自己的访问权限。如果一个进程对共享区域的访问违反了权限规定，则发生操作越权(即读写保护)。对每个内存区域指定一个键值和若干禁止的访问方式，进程中也指定键值，如果访问时键值不匹配而且是被禁止的访问方式，则出错；
  - 环保护：处理器状态分为多个环(ring)，分别具有不同的存储访问特权级别(privilege)，通常是级别高的在内环，编号小（如0环）级别最高；可访问同环或更低级别环的数据；可调用同环或更高级别环的服务。



对同环或更高级别环服务的调用



对同环或更低级环数据的访问

2. 存储分配的安全性考虑

把一个页面分配给进程之前，先要清除页面中的数据（如全部填充为0），以免该进程读取前一进程遗留在页面中的数据；

3. 存储共享

通过引用计数(reference count)来描述存储区的共享，引用计数表示共享它的进程的数目；具体的存储共享方式有两种：

- 共享段：被连接或被释放一次则对引用计数加1或减1，计数减至0则可将该共享段删除；目的在于进程间的信息交流。
- 共享页面：在物理页面表中有引用计数。只能共享不被修改的页面。这对用户应用是透明的，完全由操作系统控制，目的在于减少系统内的物理页面总数。
  - 可用于装入同一个程序而创建的几个进程，或是父进程通过fork()创建子进程；
  - 写时复制(copy on write)：如果一个进程要改写共享页面，则先把该页面复制一份，让该进程访问复制后的页面，而让其他进程访问复制前的页面。

6.6.5 虚拟存储的调入策略、分配策略和清除策略

1. 调入策略 (fetch policy)

调入策略确定在外存中的页面调入时机。在虚拟页式管理中有两种常用策略。

- 请求调页(demand paging)：只调入发生缺页时所需的页面。
  - 优点：容易实现。
  - 缺点：对外存I/O次数多，开销较大
- 预调页(prepaging)：在发生缺页需要调入某页时，一次调入该页以及相邻的几个页。
  - 优点：提高调页的I/O效率。
  - 缺点：基于预测，若调入的页在以后很少被访问，则效率低。常用于程序装入时的调页。

返回

2. 分配策略 (assignment policy)

- 在虚拟段式管理中，如何对物理内存进行分配，可采用最佳适应、最先适应等。
- 在虚拟页式和段页式管理中，地址变换最后通过页表进行，因此不必考虑分配策略。

3. 调入页面的来源

通常对外存交换区的I/O效率比文件区的高。关于调入页面的来源，这里有两种做法：

- 进程装入时，将其全部页面复制到交换区，以后总是从交换区调入。执行时调入速度快，要求交换区空间较大。
- 凡是未被修改的页面，都直接从文件区读入，而被置换时不需调出；已被修改的页面，被置换时需调出到交换区，以后从交换区调入。节省交换区空间。
  - 可能引发问题。如：装入可执行文件a从而创建进程P，如果在P执行时，改写了可执行文件a（如重新编译和链接），而此后P发生缺页需要从a中调入页面，则可能会因为各个页面内容无法配合而出错(如Bus Error或Segmentation Fault)

4. 清除策略(cleaning policy)

在虚拟页式管理中，何时将已修改页面调出到外存上。有两种常用清除策略：

- 请求清除(demand cleaning)：该页被置换时才调出，把清除推迟到最后一刻。
  - 缺点：调入所缺页面之前还要调出已修改页面，缺页进程的等待时间较长，
- 预清除(precleaning)：该页被置换之前就调出，因而可以成批调出多个页面。
  - 缺点：可能形成不必要的开销。已修改页面被调出之后仍停留在内存，如果这些页面被置换之前就被再次修改，则这些页面可以返还到进程的常驻集，而之前所做的调出操作就成为不必要的开销。这种策略发展成为页面缓冲算法(page buffering)。

6.6.6 置换算法

- 功能：需要调入页面时，选择内存中哪个物理页面被置换。称为replacement policy。
- 目标：把未来不再使用的或短期内较少使用的页面调出，通常只能在局部性原理指导下依据过去的统计数据进行预测；
- 页面锁定(frame locking)：用于描述必须常驻内存的操作系统的关键部分或时间关键(time-critical)的应用进程。实现方法为在页表中加上锁定标志位(lock bit)。

返回

- 最佳算法(OPT, optimal)
- 最近最久未使用算法(LRU, Least Recently Used)
- 先进先出算法(FIFO)
- 轮转算法(clock)
- 最不常用算法(LFU, Least Frequently Used)
- 页面缓冲算法(page buffering)

1. 最佳算法(OPT, optimal)

选择“未来不再使用的”或“在离当前最远位置上出现的”页面被置换。这是一种理想情况，是实际执行中无法预知的，因而不能实现。可用作性能评价的依据。

2. 最近最久未使用算法 (LRU, Least Recently Used)

选择内存中最久未使用的页面被置换。这是局部性原理的合理近似，性能接近最佳算法。但由于需要记录页面使用时间的先后关系，硬件开销太大。硬件机构如：

- 一个特殊的栈：把被访问的页面移到栈顶，于是栈底的是最久未使用页面。
- 每个页面设立移位寄存器：被访问时左边最高位置1，定期右移并且最高位补0，于是寄存器数值最小的是最久未使用页面。

3. 先进先出算法(FIFO)

选择建立最早的页面被置换。可以通过链表来表示各页的建立时间先后。性能较差。较早调入的页往往是经常被访问的页，这些页在FIFO算法下被反复调入和调出。并且有Belady现象。

- Belady现象：采用FIFO算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多，缺页率反而提高的异常现象。
- Belady现象的描述：一个进程P要访问M个页，OS分配N个内存页面给进程P；对一个访问序列S，发生缺页次数为PE（S,N）。当N增大时，PE(S, N)时而增大，时而减小。
- Belady现象的原因：FIFO算法的置换特征与进程访问内存的动态特征是矛盾的，即被置换的页面并不是进程不会访问的。

Belady现象的例子

进程P有5页程序访问页的顺序为：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5；

如果在内存中分配3个页面，则缺页情况如下：12次访问中有缺页9次；

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	1	2	5	5	5	3	4	4
页 1		1	2	3	4	1	2	2	2	5	3	3
页 2			1	2	3	4	1	1	1	2	5	5
缺页	x	x	x	x	x	x	x	√	√	x	X	√

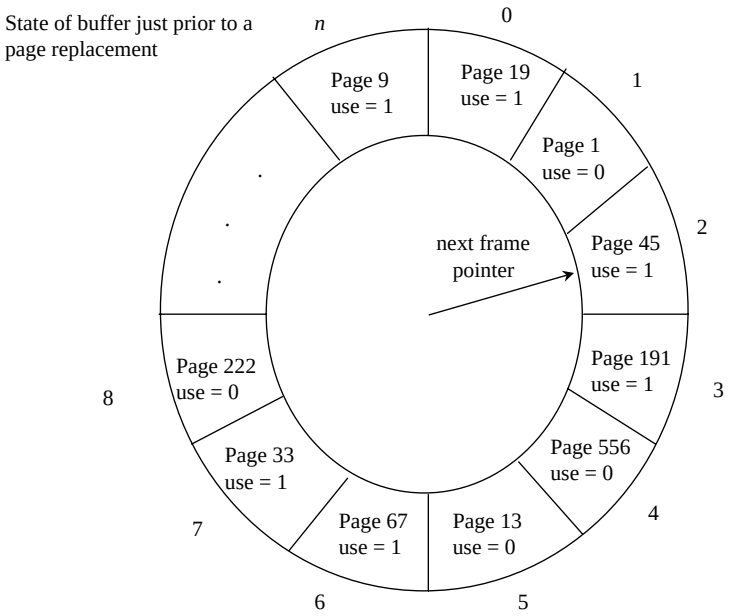
如果在内存中分配4个页面，则缺页情况如下：12次访问中有缺页10次；

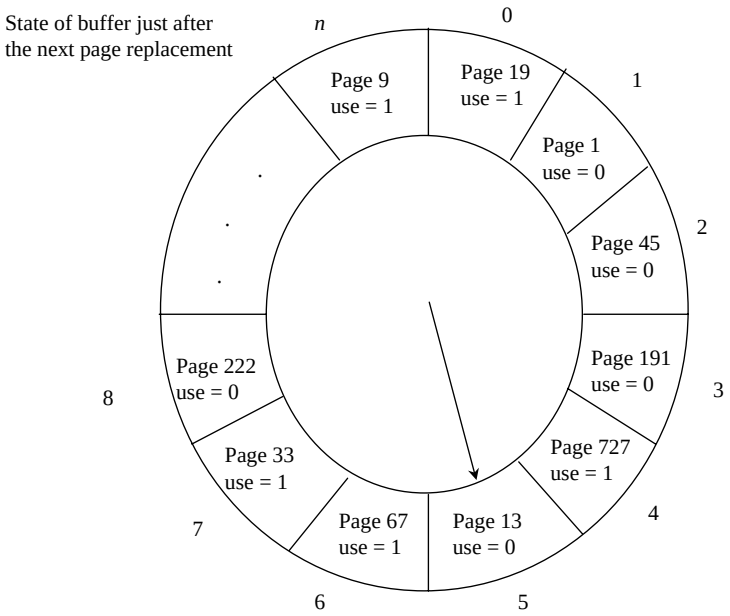
FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	4	4	5	1	2	3	4	5
页 1		1	2	3	3	3	4	5	1	2	3	4
页 2			1	2	2	2	3	4	5	1	2	3
页 3				1	1	1	2	3	4	5	1	2
缺页	x	x	x	x	√	√	x	x	x	x	x	x

4. 轮转算法(clock)

也称最近未使用算法(NRU, Not Recently Used)，它是LRU(最近最久未使用算法)和FIFO的折衷。

- 每页有一个使用标志位(use bit)，若该页被访问则置user bit=1。
- 置换时采用一个指针，从当前指针位置开始按地址先后检查各页，寻找use bit=0的页面作为被置换页。
- 指针经过的user bit=1的页都修改user bit=0，最后指针停留在被置换页的下一个页。





5. 最不常用算法(LFU, Least Frequently Used)

- 选择到当前时间为止被访问次数最少的页面被置换；
- 每页设置访问计数器，每当页面被访问时，该页面的访问计数器加1；
- 发生缺页中断时，淘汰计数值最小的页面，并将所有计数清零；

6. 页面缓冲算法(page buffering)

它是对FIFO算法的发展，通过被置换页面的缓冲，有机会找回刚被置换的页面；

- 被置换页面的选择和处理：用FIFO算法选择被置换页，把被置换的页面放入两个链表之一。
  - 如果页面未被修改，就将其归入到空闲页面链表的末尾
  - 否则将其归入到已修改页面链表。

- 需要调入新的物理页面时，将新页面内容读入到空闲页面链表的第一项所指的页面，然后将第一项删除。
- 空闲页面和已修改页面，仍停留在内存中一段时间，如果这些页面被再次访问，只需较小开销，而被访问的页面可以返还作为进程的内存页。
- 当已修改页面达到一定数目后，再将它们一起调出到外存，然后将它们归入空闲页面链表，这样能大大减少I/O操作的次数。

7. 置换算法举例

某程序在内存中分配三个页面，初始为空，页面走向为4，3，2，1，4，3，5，4，3，2，1，5。

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

页1 4 3 2 1 1 1 5 5 5 2 1 1

页2 4 3 3 3 3 3 3 3 5 5 5

页3 4 4 4 4 4 4 4 4 4 4

x x x x x x x x x

共缺页中断7次

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

页1 4 3 2 1 4 3 5 4 3 2 1 5

页2 4 3 2 1 4 3 5 4 3 2 1

页3 4 3 2 1 4 3 5 4 3 2

x x x x x x x x x x x

共缺页中断10次

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

页1 4 3 2 1 4 3 5 5 5 2 1 1

页2 4 3 2 1 4 3 3 3 5 2 2

页3 4 3 2 1 4 4 4 3 5 5

x x x x x x x x x x

共缺页中断9次

6.6.7 常驻集和工作集策略

这里要讨论的问题是分给每个进程多少物理页面，以及如何动态调整各进程的物理页面数。

1. 常驻集(resident set)

常驻集指虚拟页式管理中给进程分配的物理页面数目。

- 常驻集与缺页率的关系：
  - 每个进程的常驻集越小，则同时驻留内存的进程就越多，可以提高并行度和处理器利用率；另一方面，进程的缺页率上升，使调页的开销增大。
  - 进程的常驻集达到某个数目之后，再给它分配更多页面，缺页率不再明显下降。该数目是"缺页率－常驻集大小"曲线上的拐点(curve)。

返回

- 常驻集大小的确定方式：
  - 固定分配(fixed-allocation)：常驻集大小固定。各进程平均分配，根据程序大小按比例分配，优先权
  - 可变分配(variable-allocation)：常驻集大小可变，按照缺页率动态调整（高或低－>增大或减小常驻集），性能较好。增加算法运行的开销。
- 置换范围(replacement scope)：被置换的页面局限在本进程，或允许在其他进程。
  - 局部置换(local replacement)：容易进行性能分析
  - 全局置换(global replacement)：更为简单，容易实现，运行开销小

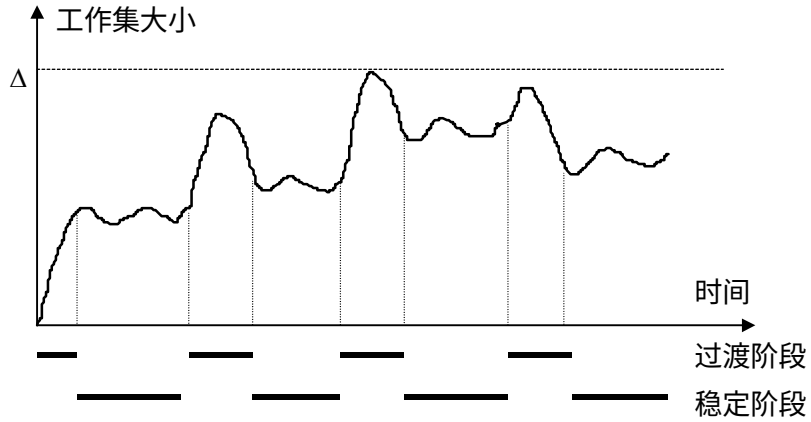
- 常驻集大小和置换范围的配合：三种策略
  - 固定分配+局部置换：这时的主要问题进程开始前要依据进程类型决定分配多少页面。多了会影响并发水平，少了会使缺页率过高。
  - 可变分配+局部置换：局部转换但在大尺度上进行可变分配。
  - 可变分配+全局置换：这时OS会一直维持一定数目的空闲页面，以快速置换；这时的主要问题是置换策略的选择，如何决定哪个进程的页面将被调出。较好的选择是页面缓冲算法。
  - 不包括“固定分配全局置换”（因为对各进程进行固定分配时不可能进行全局置换）。

2. 工作集策略(working set strategy)

1968年由Denning提出，引入工作集的目的是依据进程在过去的一段时间内访问的页面来调整常驻集大小。这里要讨论的内容是常驻集大小的动态调整策略。

- 工作集是一个进程执行过程中所访问页面的集合，可用一个二元函数$W(t, \Delta)$表示
  - t是执行时刻；
  - Δ是一个虚拟时间段，称为窗口大小(window size)，它采用"虚拟时间"单位(即阻塞时不计时)，大致可以用执行的指令数目，或处理器执行时间来计算；
  - 工作集是在$[t - \Delta, t]$时间段内所访问的页面的集合；
  - $| W(t, \Delta) |$ 指工作集大小即页面数目；

工作集大小的变化：进程开始执行后，随着访问新页面逐步建立较稳定的工作集。当内存访问的局部性区域的位置大致稳定时，工作集大小也大致稳定；局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值。



- 工作集的性质：
  - 随Δ单调递增：$W(t, \Delta) \subseteq W(t, \Delta + a)$，其中$a>0$；
- 利用工作集进行常驻集调整的策略：
  - 记录一个进程的工作集变化；
  - 定期从常驻集中删除不在工作集中的页面；
  - 总是让常驻集包含工作集；
- 困难：
  - 工作集的过去变化未必能够预示工作集的将来大小或组成页面的变化；
  - 记录工作集变化要求开销太大；
  - 对工作集窗口大小Δ的取值难以优化，而且通常该值是不断变化的；

3. 缺页率算法(PFF, page fault frequency)

- 页面被访问时的处理：每个页面设立使用位(use bit)，在该页被访问时设置use bit=1；
- 缺页时的处理：每次缺页时，由操作系统计算与上次缺页的"虚拟时间"间隔t，如处理器执行时间
- 缺页时对常驻集的调整：定义一个"虚拟时间"间隔的阈值(threshold)F。依据t和F来修改常驻集。
  - 如果t小于F，则所缺页添加到常驻集中；
  - 否则，将所有use bit=0的页面从物理内存清除并缩小常驻集；随后，对常驻集中的所有页面设置use bit=0；

注：可以和页面缓冲算法(page buffering)配合使用，获得良好的性能

- 主要缺点：
  - 当局部性区域的位置改变时，工作集的变化处于过渡阶段，其快速扩张使较多新页面添加到进程的常驻集中；
  - 其中较少使用的页面至少还要经过F虚拟时间才会被淘汰，因而带来较多不必要的调页开销。

4. 可变采样间隔算法
(VSWS, variable-interval sampled working set)

- 使用以下3个参数：采样间隔通常情况下由缺页次数Q来控制，而M和L提供异常情况下的界限条件。
  - M，采样间隔时间的下限；
  - L，采样间隔时间的上限；
  - Q，一个采样间隔内允许发生的缺页次数的上限
- 常驻集的调整：
  - 每个页面设立使用位(user bit)
  - 在每个采样间隔开始时设置各页的user bit=0
  - 在每个采样间隔结束时只保留user bit=1的页面在常驻集中，其余页面从常驻集中删除。
  - 在采样间隔内发生缺页的页面，就添加到常驻集中。

- 采样间隔划分：每次缺页时，对缺页次数加1。
  - 如果上次采样以来经过时间已达到L而缺页次数未达到Q，则结束当前采样间隔，并开始下一个采样间隔；
  - 如果缺页次数达到Q而经过时间已达到M，则结束当前采样间隔，并开始下一个采样间隔；
  - 如果缺页次数达到Q而经过时间未达到M，则不作处理。

VSWS算法的优点：在局部性区域位置改变时，缺页率上升，通过缩短采样间隔，删除无用页面，从而降低常驻集扩张的峰值；

6.6.8 虚拟存储中的负载控制

这里要讨论的是OS要在内存中驻留多少个并发进程是较好的。

1. 改善时间性能的途径

- 降低缺页率：缺页率越低，虚拟存储器的平均访问时间延长得越小；
- 提高外存的访问速度：外存和内存的访问时间比值越大，则达到同样的平均访问时间，所要求的缺页率就越低；即在平均访问时间不变的条件下，外存速度越快，要求的缺页率就越松(高)；

返回

2. 抖动问题(thrashing)

随着驻留内存的进程数目增加，或者说进程并发水平(multiprogramming level)的上升，处理器利用率先是上升，然后下降。

这里处理器利用率下降的原因通常称为虚拟存储器发生“抖动”，也就是：每个进程的常驻集不断减小，缺页率不断上升，频繁调页使得调页开销增大。OS要选择一个适当的进程数目，以在并发水平和缺页率之间达到一个平衡。

3. 负载控制(load control)策略

决定驻留内存的进程数目，在避免出现抖动的前题下，尽可能提供进程并发水平。

OS不能完全控制进程的创建，但它可通过进程挂起来减少驻留内存的进程数目。即：需要减少驻留内存的进程数目时，可以将部分进程挂起并全部换出到外存上。如：低优先级的、缺页率高的、常驻集最小的、页面最多的，等等。

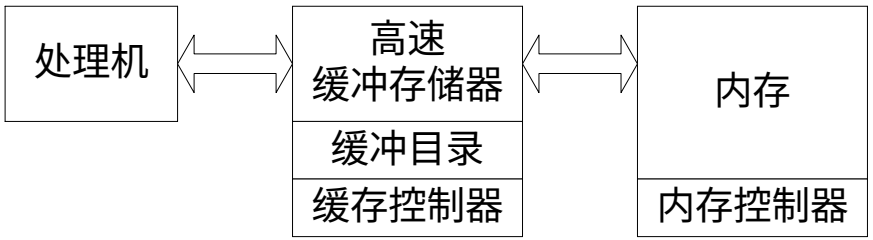
- 基于工作集策略的算法（如PFF, VSWS）：它们隐含负载控制策略，只有那些常驻集足够大的进程才能运行，从而实现对负载的自动和动态控制。
- "L = S判据"策略（Denning, 1980）：让缺页的平均间隔时间（是指真实时间而不是虚拟时间）等于对每次缺页的处理时间（即缺页率保持在最佳水平），这时CPU的利用率达到最大。(假设系统中CPU和缺页处理两部分同时满负荷工作时，系统利用率最高)
- 基于轮转置换算法的负载控制策略：
  - 定义一个轮转计数，描述轮转的速率（即扫描环形页面链的速率）。
  - 当轮转计数少于一定的阈值时，表明缺页较少或存在足够的空闲页面；
  - 当轮转计数大于阈值时，隐含系统的进程并发水平过

6.7 高速缓冲存储器

高速缓存是为了匹配CPU的处理速率与内存的访问而增加的高速存储器，其目标是提高CPU的利用率。实际系统中的高速缓存命中率为98%以上。高速缓存的使用对用户是透明的。

返回

6.7.1 高速缓存的组织



由于CPU的指令处理速度与内存中指令的访问速度存在差异（可达到一个数量级的差异，如：Intel Pentium Pro平均一个时钟周期可执行3条指令），为提高CPU的利用率，在CPU与内存之间组织一个高速缓存结构。

返回

高速缓存由三部分组成：缓冲存储器、缓存目录和缓存控制器。

- 缓存控制器：负责缓存目录的维护和利用缓存淘汰算法进行缓存的更新；
- 缓冲存储器：缓存内存中数据；
  - 缓冲存储器分为若干块。Intel Pentium有16KB L1高速缓存（8KB为数据，8KB为代码）。如：16KB（2^{14}）缓存可分为8个区，每个区为2KB（2^{11}）；每个2KB的区又分成32（2^5）个块，每个块为64（2^6）字节，各块用列号标识；从而，可用区号和列号描述每一个缓冲存储器块。

- 缓冲目录：描述各缓冲存储器块的状态。缓冲目录的表项与缓冲存储器块一一对应。每32个缓冲目录中表项构成一行，对应于一个缓存区。它包括内存地址行号、状态位，以及用于缓存淘汰算法的缓存访问信息。
  - 内存地址行号（对24位地址总线为13位）：
    - 相应缓存块对应的内存地址(24位)为行号（13位）+列号（5位）+字节数（6位）；
  - 3个状态位：描述相应缓存块的状态；
    - 有效位：相应缓存块中的数据是否有效；0表示有效，1表示无效；
    - 修改位：相应缓存块中的数据是否已经被修改过；0表示未修改，1表示已修改；
    - 故障位：相应缓存块是否出了故障；0表示无故障，1表示有故障；

6.7.2 缓存的工作过程

在不同类型的内存操作时，缓存会有不同的工作过程。具体的缓存工作过程如下：

- CPU读数据：缓存控制器自动查找缓存目录，确定相应内存数据是否在缓存中；
  - 查找过程：
    - 依据读操作的地址确定查找缓存目录的列号；
    - 比较缓存目录相应列的各区行号与地址行号；
    - 判断有效位是否为0；
  - 依据查找结果，有两种可能的操作：
    - 如果在缓存，则从缓存中读数据，并修改访问标志；
    - 如果不在缓存，则从内存中读数据，同时该块内容被送到缓存相应列的某块。

返回

- CPU写数据：查找缓存目录，确定相应地址是否在缓存中，有两种可能的操作：
  - 如果在缓存，则修改缓存内容，并把缓存目录中相应修改位置1；这时有两种做法：
    - 立即写：内存与缓存的相应块同时写；
    - 惰性写：数据只写入缓存，不马上写入内存；当该缓存块被淘汰时，才写回内存；
  - 如果不在缓存，则先把内存读入缓存，再在缓存中修改；
- 通道向内存写数据：数据被写入内存；缓存控制器同时查找缓存目录，如果有，则修改相应表项的有效位为1(表示无效)；
  - 注：通道向内存写数据时，不会同时向缓冲区写数据。
- 通道从内存读数据：
  - 如果在缓存中，则从缓存中读数据到通道；
  - 如果不在缓存中，则从内存中读数据到通道，但并不同时送到缓存；

6.8 存储管理举例

返回

6.8.1 Solaris中的存储管理

由于UNIX可在多种平台上运行，其存储管理差异十分大。这里我们介绍Solaris 2.x的存储管理系统。Solaris的存储管理系统分成两部分：

- 对换系统（paging system）：为进程和磁盘缓存提供页式虚拟存储管理；
- 内核存储分配器（kernel memory allocator）：为内核提供内存分配服务；

返回

6.8.1.1 对换系统

1. 对换系统的数据结构

- 页表(page table)：每个进程有一个页表，进程逻辑空间的每页对应页表中的一个表项；
  - 页面号(page frame number)：物理内存页面的位置；
  - 有效标志(valid)：表示该页是否在内存中。
  - 未访问时间(age)：表示该页面未被访问的时间；长度与内容都与处理机类型相关，不同的置换算法对该字段的使用也不同；
  - 修改标志(modify)：表示该页面被修改过。
  - 引用标志(reference)：表示该页面被访问过。该标志在装入时清0，并且会被页置换算法周期性清0。
  - 访问保护标志(protect)：表示该页是否允许写操作。
  - 写时复制标志(copy on write)：多个进程共享该页面时，设置该标志；某进程在向该页面写入前，必须先复制该页后再写入。这种做法可把复制操作延迟到写入数据前进行，从而避免不必要的复制。

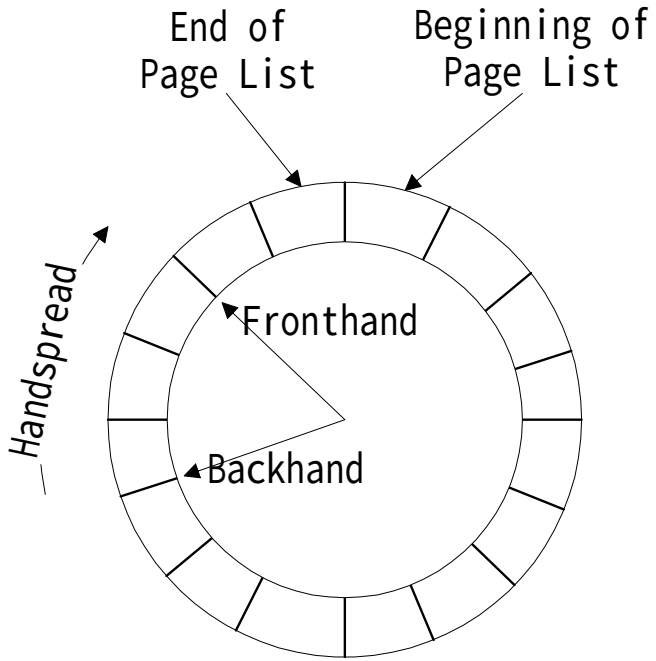
- 硬盘块描述表(disk block descriptor)：该表表项与页表表项一一对应，描述该页在磁盘上的副本；
  - 对换设备号(swap device number)：该页对应外存页所在设备的逻辑设备号。该字段的使用允许在一个系统中使用多个对换分区。
  - 设备块号(device block number)：该页对应外存页在对换分区中的位置。
  - 存储类型(type of storage)：表示该页对应外存页位于对换分区或可执行文件。
    - 利用可执行文件创建进程时，只有部分代码和数据被装入内存；
    - 出现缺页时再装入其他部分，并在对换区中建立相应页面副本。

- 页面表(page frame date table)：按物理页面号排序，描述每个物理页面；
  - 页面状态(frame state)：指示该页面是否空闲。
    - 被占用状态又分成：与对换分区对应、与可执行文件对应和正在进行对换；
  - 引用计数(reference count)：表示引用该页的进程数目；
  - 逻辑设备号(logical device)：表示包含本页面复本的逻辑设备号；
  - 设备块号(block number)：表示页面副本在逻辑设备上的位置；
  - 空闲页面表项指针(pfdata pointer)：指向其他空闲页面表项的指针；用于空闲页面链表；

- 对换区表(swap-use table)：每个磁盘对换区有一个对换表，该对换区上的每个页副本对应一个表项；
  - 引用计数(reference count)：指向该页的页表表项数目；
  - 页标识(page/storage unit number)：在存储单元对应的页标识；

2. 双指针轮转置换算法 (Two-Handed Clock Page-Replacement Algorithm)

该算法利用页面表上的几个指针维护一个空闲页面表。当空闲页面数少于一定阈值时，该算法置换一些页面加入空闲页面表。该算法是一种改进的轮转算法。



- 算法数据结构
  - 我们把页面表视为一个环形链
  - 每个页面对应一个引用位（reference bit）描述该页面的引用情况；
  - 在页面表上定义两个指针
    - 前指针（fronthand）
    - 后指针（backhand）；
- 算法操作：
  - 系统在访问一个页面时，把相应引用位置1；
  - 前指针在扫描页面表时，把各页面的引用位清0；
  - 后指针在扫描页面表时，把引用位为0的页面对换到外存，其他页面的引用位清0；

- 算法参数：OS在一定范围内调节两个参数。当空闲页较少时，加快扫描速度或缩小扫描窗口；反之，放慢扫描速度或加大扫描窗口。
  - 扫描速度（scanrate）：前后指针扫描页面表的速度，单位为每秒扫描的页面数；
  - 扫描窗口（handspread）：前指针与后指针间的页面数，描述页面引用的统计范围；

6.8.1.2 内核存储分配器

1. 内核存储分配特征

- 内核需要分配和释放小内存块，用于内核的数据结构和缓冲区；
- 这些内存块通常比物理页面要小许多；
- 这些内存块的分配和释放要求快速进行；
- 分配和释放的内存块的大小变化是缓慢的；这是一个分析结果。

2. SVR 4的惰性动态分区算法（lazy buddy system）

- 惰性动态分区算法：该算法是一种动态分区算法。
  - 分配的分区大小可变，但只能为$2^K (L \le K \le U)$。
  - 释放的分区并不马上合并，而是维持一定数目的各种大小的空闲分区；当空闲分区超过一定数目时，才合并。

- 该算法维护的参数：
  - N_i表示大小为2^i的分区数目；
  - A_i表示大小为2^i的已分配分区数目；
  - G_i表示大小为2^i的适合合并空闲（globally free）分区数目；两个可合并成一个2^{i+1}分区相邻的2^i分区将合并成一个；
  - L_i表示大小为2^i的不适合合并空闲（locally free）分区数目；
- 有如下关系：
  - $N_i = A_i + G_i + L_i$
- 合并的判断条件：
  - 要求$Li \le Ai$；依据它们的差来决定是否合并：差为0或1时进行合并，大于1时不合并。

SVR 4的惰性动态分区算法算法的实现

定义变量$D_i = A_i - L_i = N_i - 2L_i - G_i$
D_i的初值为0；
在分配和释放一个大小为2^i的分区时，对D_i进行更新；

在分配分区时，进行的操作为：

如果有相应大小的空闲分区，选择一个空闲分区进行分配；
　　如果被分配的空闲分区不适合合并，
　　　　则D_i加2(即A_i加1的同时L_i减1)，
　　否则D_i加1；(即A_i加1的同时L_i不变)
否则先找一个大小为2^{i+1}的分区，把它分成两个较小的分区；
　　分配其中的一个分区，另一个标记为不适合合并分区；
　　D_i保持不变；

在释放分区时，进行的操作为：

按D_i的大小分成三种情况分别进行处理：

D_i大于等于2时：标记被释放分区为不适合合并；D_i减2；(即A_i减1的同时L_i加1)

D_i等于1时：标记被释放分区为适合合并，进行可能的合并；D_i改为0；(即A_i减1的同时L_I不变)

D_i等于0时：标记被释放分区为适合合并，进行可能的合并；
找一个不适合合并的分区，改为适合合并，进行可能的合并；
D_i不变；(即A_i减1的同时L_I不变)

注：为了提高系统分配和释放分区的速度，通常把空闲分区按大小组织成不同的双向链表；每个链表中的分区大小都是相同的；不适合合并的分区放在队头，适合合并的分区放在队尾。这样，通常情况下分配和释放的分区都是不适合合并的分区。

3. Solaris 2.4的slab分配器

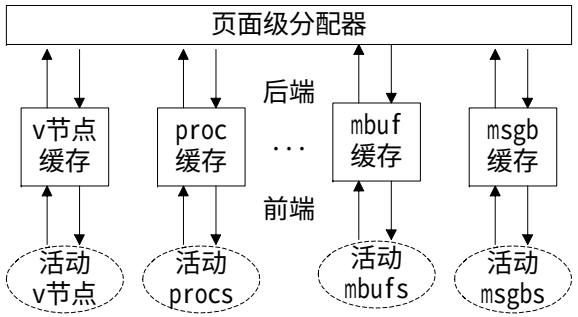
slab分配器的特点：改进分配器的性能和内存利用率；

- 利用对象复用，优化内存的分配和初始化；
  - 分配器对内核对象的操作顺序：
    - 分配内存
    - 构造(初始化)对象
    - 使用对象
    - 释放对象
    - 释放内存
  - 在某些情况下，对象在释放前又恢复到初始化后的状态；
  - 缓存并利用对象比分配内存并初始化要好；

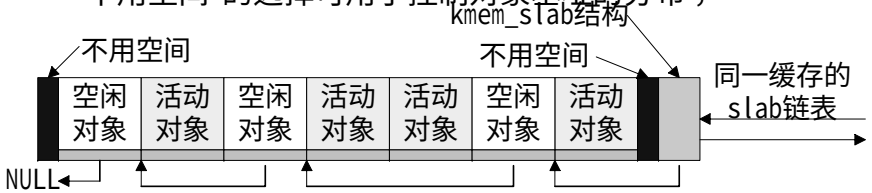
- 利用数据对齐，优化高速缓存的的利用率；
  - 由于下列原因，高速缓存的某些列竞争激烈，而其余部分利用率很低；
    - 高速缓存是按区对齐，分列存放的。如Intel Pentium的L1 cache中每个2KB的区就分成32列。A10至A6相同的块一定在同一列。
    - 内核中的内存分配大小为2的整次幂，并按相应尺寸对齐；
    - 大部分内核对象都在对象的开始部分保存经常访问的数据；

slab分配器的改进之处

- 设置一组对象缓存，每个缓存保存一种对象类型；
  - 每个缓存分为前端和后端两部分。前端与客户交互，后端与页面级分配器交互。
  - 客户从缓存中获取构造好的对象，返回用完的对象；后端与页面级分配器交换页面。



- 通过把内核小对象组织成内核页面结构，使对象的基址分布均衡，从而提高和均衡高速缓存和总线的使用率。
  - 内核页面结构分成：kmem_slab结构、对象集和不用空间；
  - kmem_slab结构中记录本块内的使用对象计数、块内空闲对象的单向链表和同缓存块间的双向链；
  - 对象集中记录对象和空闲对象链表指针；
  - "不用空间"的选择可用于控制对象基址的分布；



- 同一缓存中的所有内核页面块构成的双向slab链表中，所有对象都在使用的页面块排在前，部分对象在使用的页面块居中，空闲的页面块排在最后。
  - 在分配对象时，最后使用完全空闲的页面块；
  - 页面级分配器回收页面时，首先回收完全空闲的页面块；

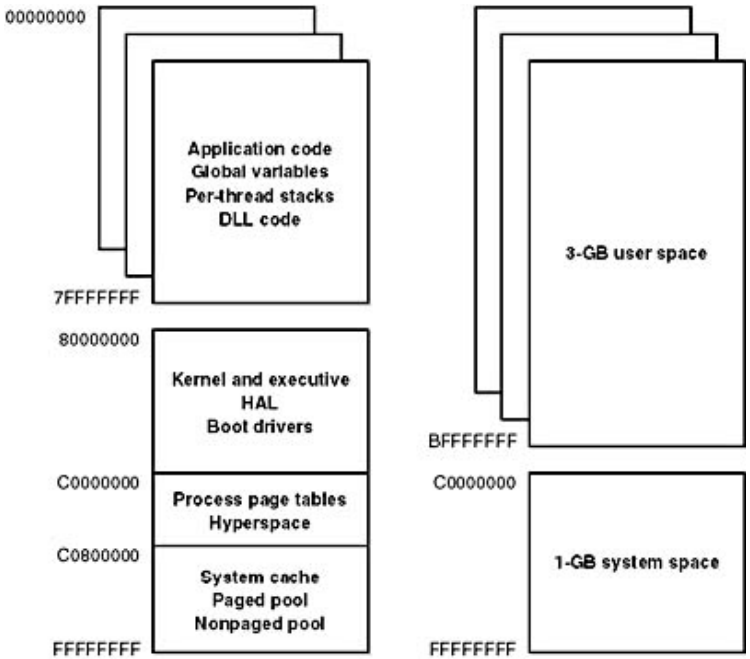
6.8.2 Windows NT中的存储管理

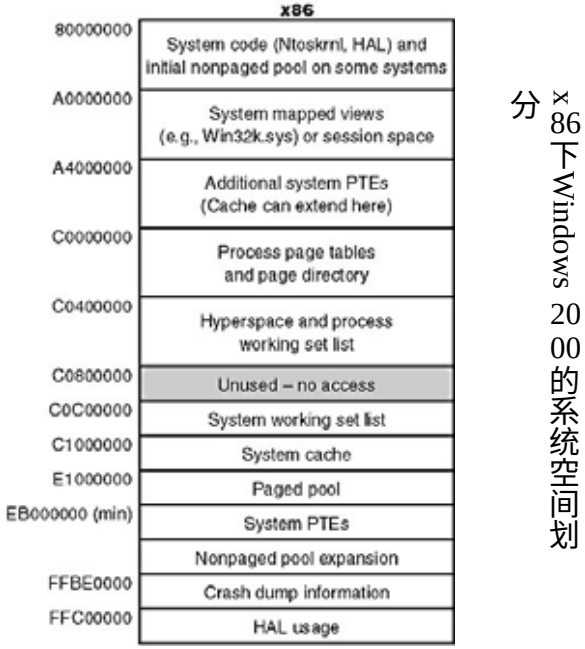
1. 虚拟地址空间布局

- NT使用的页面大小为4KB（2^{12}）。每个NT进程地址空间为4GB（2^{32}），其中：
- 用户存储区：在用户态和核心态都可访问的用户存储区为2GB；用户存储区为页交换区，可对换到外存；用户存储区的内容包括：
  - 专用进程地址空间：用户代码、数据和堆栈；
  - 线程环境块（TEB）：用户态代码可修改的线程控制信息；
  - 进程环境块（PEB）：用户态代码可修改的进程控制信息；
  - 共享用户数据页：系统存储区映像，为用户态可访问的系统空间，目的在于避免用户态与核心态的频繁切换；如：系统时间。

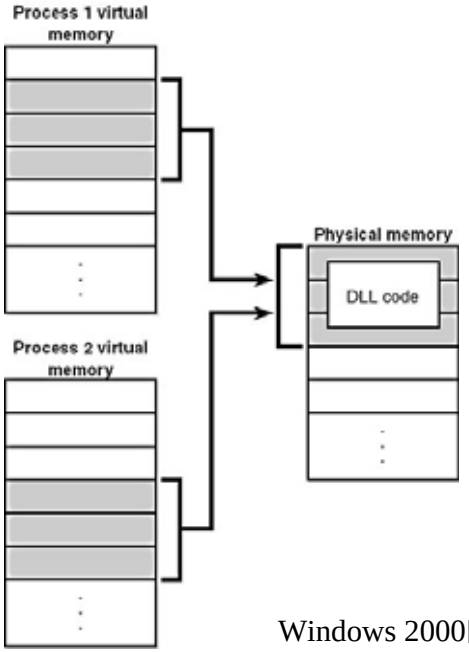
返回

- 系统存储区：在核心态可访问的系统存储区为2GB；按交换特征，系统存储区可分为：
  - 固定页面区：永不被换出内存的页面；如：HAL特定的数据结构；
  - 页交换区：非常驻内存的系统代码和数据；如：进程页表和页目录；
  - 直接映射区：常驻内存且寻址由硬件直接变换的页面，访问速度最快；用于存放内核中频繁使用且要求快速响应的代码。

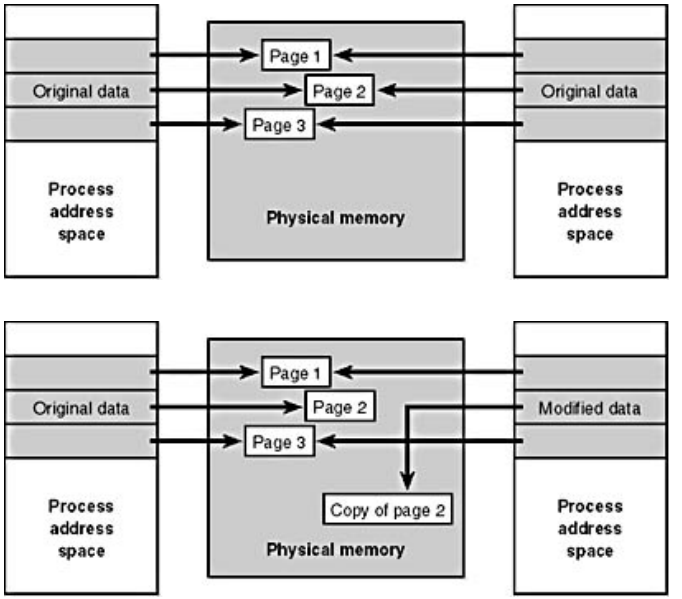




分x86下Windows 2000的系统空间划



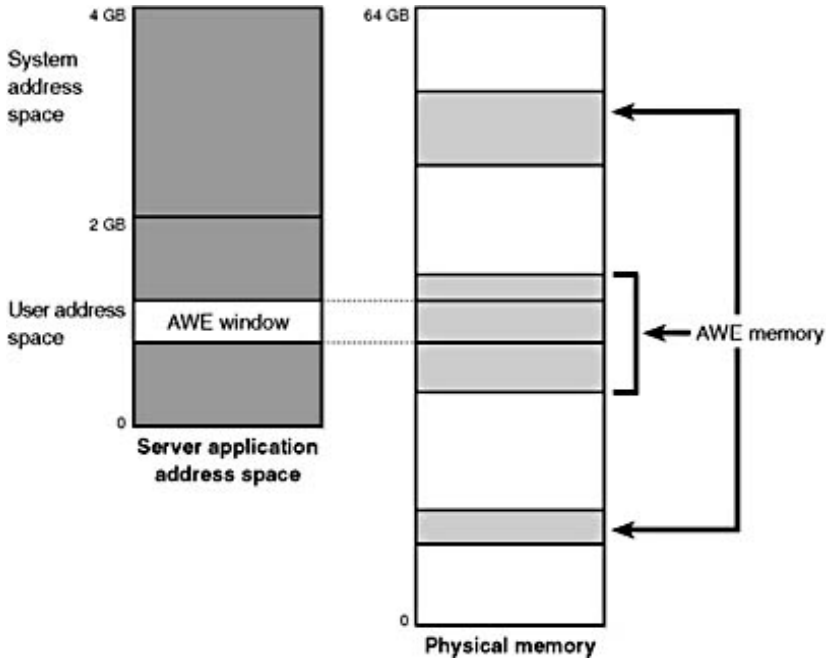
Windows 2000的内存共享



写时复制

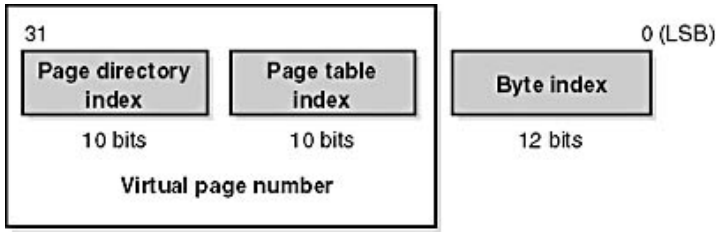
Windows2000的地址空间扩展

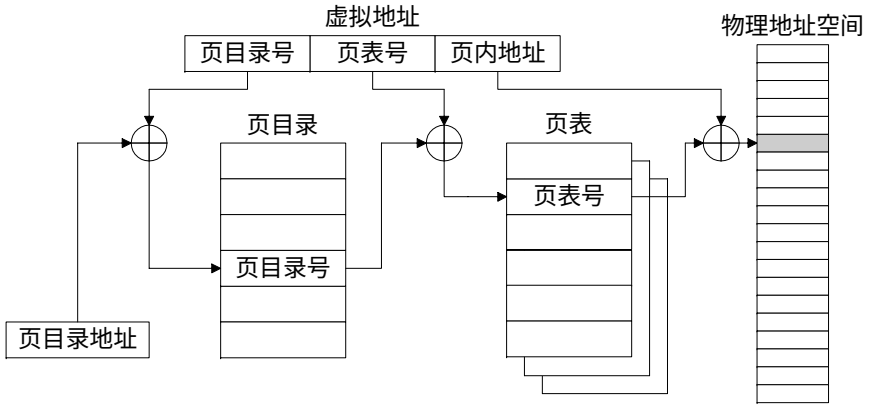
- 利用窗口映射(AWE, Address Windowing Extensions)方法可在一个进程中使用大于2GB或3GB的物理内存空间。使用步骤分成3步：
  - 分配物理内存(可大于2GB)；
  - 在进程虚拟地址空间创建一个窗口区域(小于2GB)；
  - 把物理内存空间的一个区域映射到窗口区域，从而可访问在区域的内存；

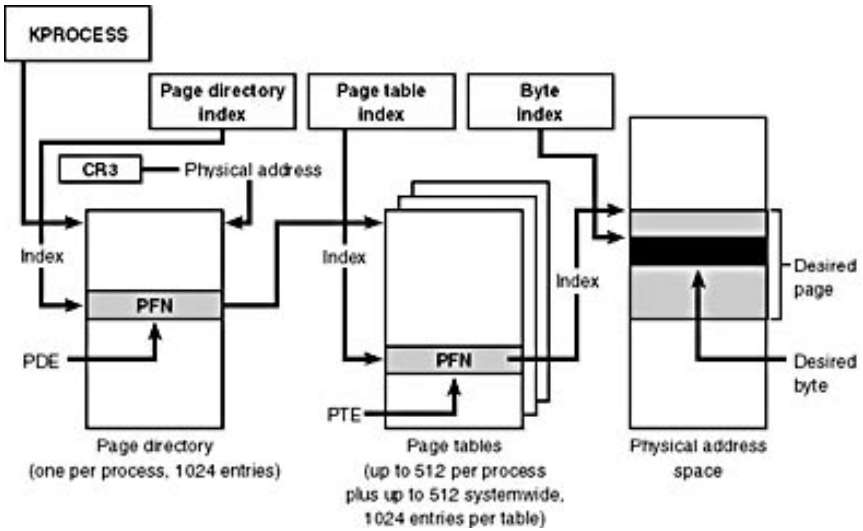


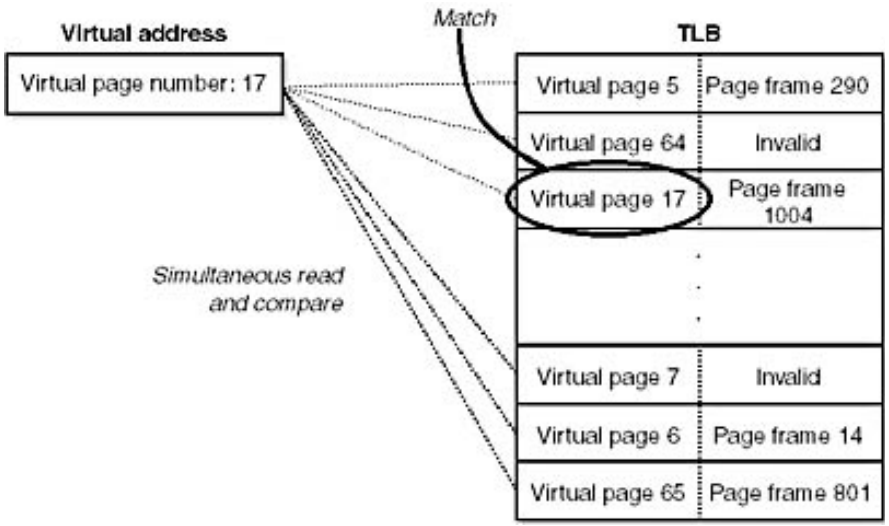
2. 地址转换机构

NT使用2级页表结构转换虚拟地址，第一级称为页目录（每个进程一个页目录），第二级称为页表。每个页目录或页表有1024(2^{10})个表项，每个表项为4字节。由于每个页面为4KB，每个进程的地址空间可为4GB（$2^{10}*2^{10}*2^{12}$）。



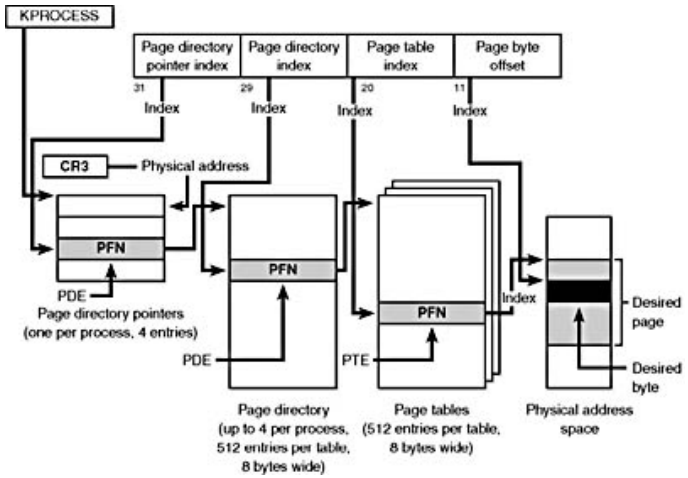






物理地址扩展(PAE, Physical Address Extension)

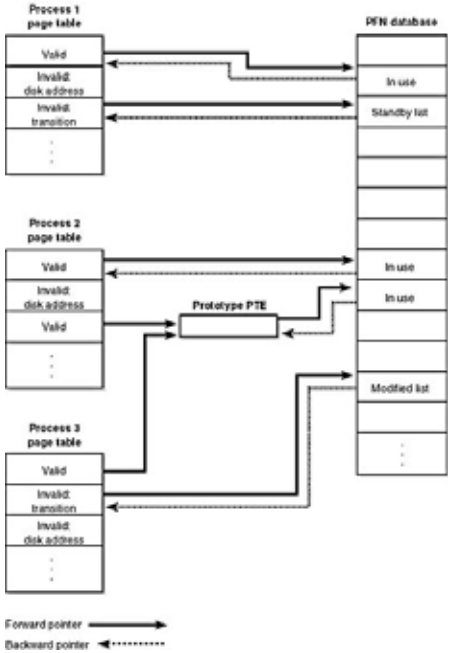
每个PDE和PTE表项都是8个字节，物理页面号为24Bit，可访问的物理地址空间可达64GB。

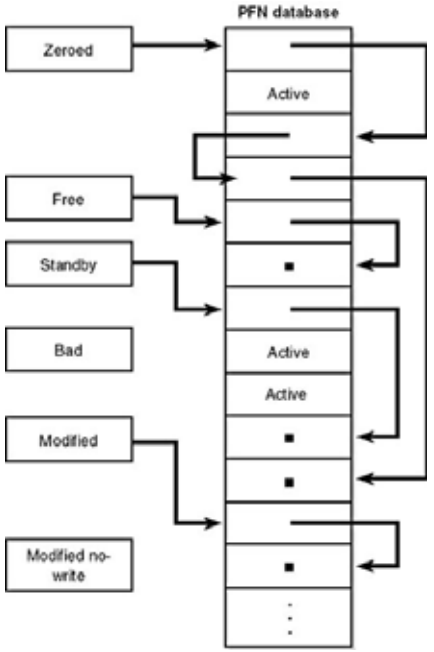


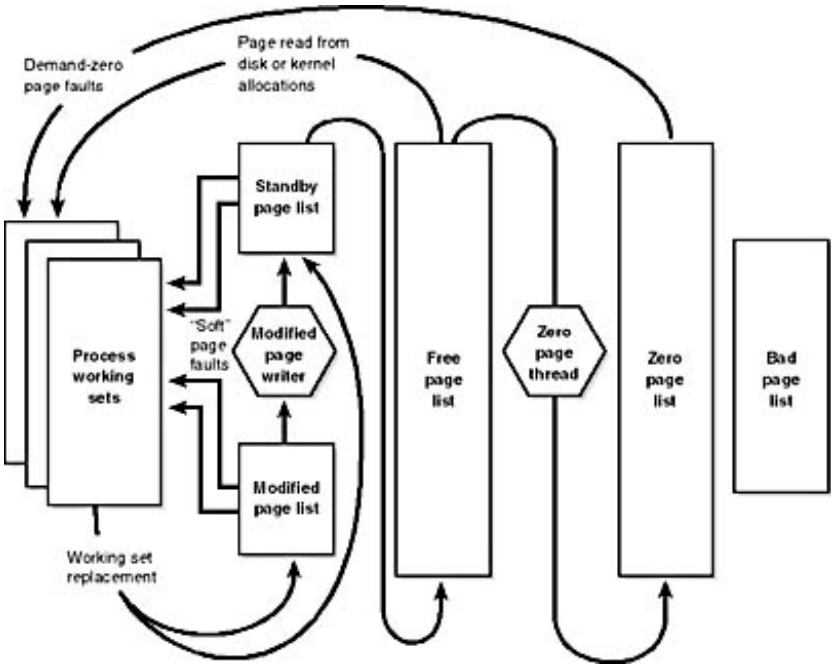
3. 页面状态

Windows NT和Windows 2000的页面有8种状态。

- 有效状态(active or valid)：某进程正在使用该页面，也可能不属于任何进程(如不可对换的内核页面)；
- 过渡状态(transition)：页面处于不属于任何进程的过渡状态，用于避免页面冲突。如正在进行页面与I/O设备间的数据传送。清零状态(zeroed)：空闲且已被清零；
- 修改状态(modified)：已标记为无效，但对该页面内容的修改尚未写入外存，可快速回到有效状态；
- 不保存的修改状态(modified no-write)：已标记不需要写入外存的修改状态。如，该状态可用于NTFS的事务交易日志状态的记录过程。
- 备用状态(standby)：已标记为无效，但可快速回到有效状态；
- 空闲状态(free)：空闲但尚未被清零；
- 坏页状态(bad)：该页面产生硬件错，不能再用；







4. 页面调度策略

页面调度策略包括取页策略、置页策略和淘汰策略。

- 取页策略：NT采用按进程需要进行的请求取页和按集群方法进行的提前取页。集群方法是指在发生缺页时，不仅装入所需的页，而且装入该页附近的一些页。
- 置页策略：在线性存储结构中，简单地把装入的页放在未分配的物理页面即可。
- 淘汰策略：采用局部FIFO置换算法。在本进程范围内进行局部置换，利用FIFO算法把驻留时间最长的页面淘汰出去。

5. 工作集策略

NT根据内存负荷和进程缺页情况自动调整工作集。

- 进程创建时，指定一个最小工作集（可用SetProcessWorkingSetSize函数指定）；
- 当内存负荷不太大时，允许进程拥有尽可能多的页面；
- 系统通过自动调整保证内存中有一定的空闲页面存在；

小结

- 引言：存储组织（层次），存储管理功能
- 重定位和装入（绝对装入、可重定位装入、动态运行期装入）
- 静态链接和动态链接（可重入代码）
- 存储管理方式：单一连续区管理，分区管理（静态和动态分区）
- 覆盖，交换
- 页式和段式存储管理：原理，优缺点，数据结构，地址变换，分段的意义，两者比较

- 虚拟存储器：
  - 局部性原理，虚拟存储器的原理
  - 种类（虚拟页式、段式、段页式），缺页中断，发展趋势
  - 存储保护和共享
  - 调入策略、分配策略和清除策略
  - 置换策略
  - 常驻集和工作集策略
  - 虚拟存储器的性能分析
  - 负载控制策略
- 高速存储器
- 存储管理实例

作业

- 实验三