**第四章 存 储 管 理**

存储管理是操作系统的重要组成部分，负责管理计算机系统的重要资源--内存储器。由于任何程序和数据必须占用内存空间才能得以执行和处理，因此，存储管理的优劣直接影响系统性能。内存对数据的存取比处理器处理数据的速度慢得多，硬件技术的不断发展还在进一步拉大这种距离，通过高速缓存可以部分缩小差距，但高效的存储管理仍然是操作系统设计中的重要课题。内存空间一般分为两部分：系统区，用于存放操作系统内核程序和数据结构等；用户区，用于存放应用程序和数据。存储管理对内核区和用户区都提供相应支持和进行管理，当然，也涉及对外存（磁盘）空间的某些管理。尽管现代计算机内存容量不断增大，但仍然不能保证有足够大的空间支持大型应用和系统程序及数据的使用。因此，操作系统的主要任务之一是尽可能方便用户使用和提高内存利用率。此外，有效的存储管理也是多道程序设计技术的关键支撑。具体地说，存储管理包含以下功能：

（1）存储分配。为进程分配内存空间以便运行，完成内存区的分配和去配工作。在虚存管理系统中，由于运行程序部分处于内存，部分位于外存，故不但涉及内存也涉及外存空间的申请与释放工作。

（2）地址映射。内存被抽象成一维地址空间，使得进程认为分配给它的地址空间是一个大且连续地址所组成的数组，或者把内存抽象成二维地址空间，以支持模块化程序设计；同时建立抽象机制支持进程使用逻辑地址来访问物理内存单元，实现地址映射。

（3）存储保护。系统负责隔离分配给进程的内存区，互不干扰免遭破坏，确保进程对存储单元的独占式使用，防止地址越界或操作越权，以实现存储保护功能。

（4）存储共享。为了节省内存空间和实现进程通信，系统允许多个进程共享内存区。在这种情况下，超越隔离机制并授权进程允许共享访问，达到既能共享内存区内信息又能提高内存利用率的目的。

（5）存储扩充。内存速度快但容量小，外存容量大却速度慢，应用程序的大小不应受到物理内存容量限制。为此，操作系统把内存和外存结合起来，形成一个容量近似外存、速度近似内存的虚拟存储器，允许用户的逻辑地址空间大于内存物理地址空间，虚存管理自动在内存和外存之间移动信息。

本章在介绍计算机存储器层次之后，先后分析连续存储管理技术、分页存储管理技术和分段存储管理技术，再讨论虚拟存储管理技术，最后，简单介绍Linux的物理内存管理和虚拟存储管理。

**4.1 存储器工作原理**

**4.1.1 存储器层次**

目前，计算机系统均采用层次结构的存储子系统，以便在容量大小、速度快慢、价格高低等诸多因素中取得平衡点，获得较好性能/格比。计算机系统的存储器层次结构自上至下依次分为：寄存器、缓存、内存、磁盘、磁带等５层。存储介质的访问速度由下而上越来越快，容量越来越小，价格越来越高。其中，寄存器、缓存和内存均属于操作系统存储管理的管辖范畴，掉电后它们存储的信息不复存在；磁盘和磁带属于文件管理和设备管理的管辖对象，它们所存储的信息将被持久性保存。

可执行程序必须被保存在内存中，与设备交换的信息也依托于内存地址空间。由于处理器在执行指令时的内存访问时间远大于其处理时间，所以，寄存器和高速缓存被引入来加快指令执行。寄存器是访问速度最快但价格最昂贵的存储器，其容量较小，一般以字为单位，一个计算机系统可能包括几十个寄存器，用于加速存储访问速度，如用寄存器存放操作数，用做地址寄存器，或用做变址寄存器，以加快地址转换速度。高速缓存的容量较寄存器稍大，其访问速度快于内存。利用高速缓存来存放内存中经常访问的信息，以提高程序执行速度。多层次的存储体系十分有效和可靠，能达到很高的性能/价格比。

由于程序处理数据时存在顺序性和局部性，执行时仅需调入当前运行使用的一部分，其他部分待需要时再逐步调入。据此，计算机系统为了容纳更多作业，或为了处理更大批量数据，可在磁盘上建立磁盘缓存以扩充内存的存储空间，计算程序和所处理的数据可装入磁盘缓存，操作系统自动实现内存和磁盘缓存之间的程序和数据的调进调出，从而，向用户提供比实际内存容量大得多的虚拟内存。基于这个原理，就可以设计出多级层次式体系结构的存储子系统。

**4.1.2 地址转换与存储保护**

应用程序究竟如何在计算机系统上运行呢？这涉及到程序设计语言、编译程序和汇编程序、操作系统及计算机硬件。首先，用程序设计语言编写和编辑应用程序，所编写的程序称为源程序，其中使用符号名集合所限定的空间称为程序名字空间，源程序不能被计算机直接运行，需要通过如图4-1所示的3个阶段处理。接着编译程序或汇编程序处理源程序并生成目标代码（程序），链接程序把它们链接为一个可重定位代码（程序），此时该程序处在逻辑地址空间中。下一步装载程序将可执行代码装入物理地址空间，直到此时程序才能运行。这些阶段都在操作系统控制下，它以进程机制来执行程序，调度程序调度进程到处理器上运行，按进程需要进行数据I/O，最终获得计算结果。

链接

源程序模块1

源程序模块2

源程序模块n

目标代码1

目标代码2

目标代码n

编译

动态重定位

静态重定位

可重定位目标代码

(装载代码)(外存)

装载

执行

程序名字空间

逻辑地址空间

物理地址空间

动态链接系统库

静态链接系统库

可执行二进代码(内存)

if test –f”$1” //测试$1，

then pr $1 //若为普通文件，则打印文件内容

else if test –d”$1” //否则，测试$1，

then ( cd $1;pr \*) //若为目录名，则作为工作目录并打印下属

//所有文件

else echo”$1 is neither a file nor a directory.” //否则，显示出错信息

fi

fi存)

可执行二进代码(内存)

。。。

。。。

…

…

图4-1 程序的编译、链接、装载和执行

**1. 程序编译**

源程序经过编译程序(compiler)或汇编程序(assembly)的处理生成目标模块（也称目标代码）。一个程序可由独立编写且具有不同功能的多个源程序模块组成，由于模块包含外部引用，即指向其他模块中的数据或指令地址，或包含对库函数的引用，编译程序或汇编程序负责记录引用发生位置，其处理结果将产生相应的多个目标模块，每个目标模块都附有供引用使用的内部符号表和外部符号表。符号表中依次给出各个符号名及在本目标模块中的名字地址，在模块被链接时进行转换。例如，编写一个名为simplecomputing的源程序，其主程序main中有函数和子程序调用指令：求平方根SQRT和转子程序 SUB1。SQRT是函数库中已被编译成可链接的目标模块的标准子程序，SUB1是另一个模块中定义的已被编译成可链接的子程序，这时所调用的入口地址均是未知的；编译程序或汇编程序将在外部符号表中记录外部符号名SQRT和SUB1，同时两条调用指令指向函数和子程序的位置。

**2. 程序链接**

链接程序（linker）的作用是根据目标模块之间的调用和依赖关系，将主调模块、被调模块、以及所用到的库函数装配和链接成一个完整的可装载执行模块。根据程序链接发生的时刻和链接方式，程序链接可分成三种方式：

（1）静态链接

是指在程序装载到内存和运行前，就已将它的所有目标模块及所需要的库函数进行链接和装配成一个完整的可执行程序且此后不再拆分。静态方式使得链接过程与装载过程相对独立，链接程序和装载程序可独立设计，但不支持内存空间中目标模块的单副本、不利于模块共享。

仍采用上例，linker首先将主程序调入工作区，然后，扫描外部符号表，获得外部符号名SQRT，用此名字从标准函数库中找出函数的sqrt.o并装入工作区，拼接在主程序的下面；SQRT函数的内存位置就是调用SQRT指令的入口地址，将此指令代真；调用SUB１的链接过程与此相似，只是从另一个模块中找到sub1.o的位置并进行指令代真；需要解析内部和外部符号表，把对符号名的引用转换为数值引用，要将涉及名字地址的程序入口点和数据引用点转换为数值地址。经过链接处理后，主程序main与SQRT函数和SUB1子程序链接成完整的可重定位目标程序simplecomputing.o(使用了图4.1中用实线框住的静态系统库)。

可重定位目标程序又称装载代码模块，它存放于磁盘中，由于程序在内存中的位置不可预知，链接时程序地址空间中的地址总是相对某个基准（通常为０）开始编号的顺序地址，称为逻辑地址或相对地址。

（2）动态链接

是指在程序装入内存前并未事先进行程序各目标模块的链接，而是在程序装载时，一边装载一边链接，生成一个可执行程序。在装载目标模块时，若发生外部模块调用，将引发相应外部目标模块的搜索、装载和链接。动态链接方式使得各目标模块相对独立存在，便于个别目标模块的修改或更新，且不影响程序的装载和执行；同时，若发现所需某目标模块已在内存，可直接进行链接且无须再次装载，支持目标模块的共享。但由于装载和链接过程交织在一起，装载程序和链接程序将合二为一，增加了设计和开发难度。

**（3）运行时链接**

是指将某些目标模块或库函数的链接推迟到执行时才进行。在程序执行过程中，若发现被调用模块或库函数尚未链接，先在内存中进行搜索以查看其是否装入内存；若已装入，则直接将其链接到调用者程序中，否则才进行该模块在外存上的搜索，以及装入内存和进行链接，生成一个可执行程序。这样可避免事先无法知道本次要运行哪些目标模块，避免程序执行过程中不被调用的某些目标模块在执行前进行链接和装载而引起的开销，提高系统资源利用率和系统效率。现代操作系统都支持和采用动态链接系统库（Dynamic Link Library，DLL)及运行时链接，程序执行所需的库函数所在的部分目标模块是伴随着其被调用才动态进行装载和链接，而这些目标模块可能因其他程序调用已被调入内存，也可能因没有程序调用尚未装入内存。具体做法是：不必将程序所需的外部函数代码从系统库中提取出来并链入目标模块中，而仅是在程序调用处登记调用信息，记录函数名及入口号，形成调用链接。一旦DLL库调入内存后，就可以确定所调函数在内存的物理地址(使用图4.1中用虚线框住的动态系统库)，系统库及动态链接调用过程详见本章4.3.4节。

**3. 程序装载**

通常，装载程序（loader）把可执行程序装入内存的方式有三种：(1)绝对装载。装载模块中的指令地址始终与其内存中的地址相同，即在模块中出现的所有地址都是内存绝对地址。(2)可重定位装载。根据内存当时使用情况，决定将装载代码模块放入内存的物理位置。模块内使用的地址都是相对地址。(3)动态运行时装载。为提高内存利用率，装入内存的程序可换出到磁盘上，适当时候再换入到内存中，对换前后程序在内存中的位置可能不同，即允许进程的内存映像在不同时候处于不同位置，此时模块内使用的地址必为相对地址。

磁盘中的装载模块所使用的是逻辑地址，其逻辑地址集合称为进程的逻辑地址空间。逻辑地址空间可以是一维的，这时逻辑地址限制在从0开始顺序排列的地址空间内；也可以是二维的，这时整个程序被分为若干段，每段都有不同段号，段内地址从０开始顺序编址。进程运行时，其装载代码模块将被装入物理地址空间中，此时程序和数据的实际地址不可能同原来的逻辑地址一致。物理内存从统一的基地址开始顺序编址的存储单元称为物理地址或绝对地址，物理地址的总体构成物理地址空间。注意，物理地址空间是由存储器地址总线扫描出来的空间，其大小取决于实际安装的内存容量。

可执行程序逻辑地址转换（绑定）为物理地址的过程称地址重定位、地址映射或地址转换，基于上述程序装入方式，可区分三种地址重定位。

（1）静态地址重定位

由装载程序实现装载代码模块的加载和地址转换，把它装入分配给进程的内存指定区域，其中的所有逻辑地址修改成内存物理地址，称静态地址重定位（static relocating address）。地址转换工作在进程执行前一次完成，无须硬件支持，易于实现，但不允许程序在执行过程中移动位置，这种技术只在早期单用户单任务系统中使用过。

（2）动态地址重定位

由装载程序实现装载代码模块的加载，把它装入分配给进程的内存指定区域，但对链接程序处理过的应用程序的逻辑地址则不做任何修改，程序内存起始地址被置入硬件专用寄存器--重定位寄存器，如图4-2所示。程序执行过程中，每当CPU引用内存地址（访问程序和数据）时，由硬件截取此逻辑地址，并在它被发送到内存之前加上重定位寄存器的值，以便实现地址转换，称动态地址重定位（dynamic relocating address）。与静态地址重定位相比，动态地址重定位具有允许程序在内存中移动、便于程序共享和内存利用率高等优点。

（3）运行时链接地址重定位

对于静态和动态地址重定位装载方式而言，装载代码模块是由整个程序的所有目标模块及库函数目标模块经链接和整合构成的可执行程序，即在程序启动执行前已经完成了程序的链接过程。可见，装载代码的正文结构是静态的，在程序的整个执行期间保持不变，且同一程序每次运行时的装载模块都是相同的。但在实际应用场合，每次要运行的装载模块可能并不相同，如果由于事先无法知道本次要运行哪些模块就采取将整个程序所有模块在装载时或装载前全部链接在一起的处理方法，必然造成内存空间利用率及系统执行效率低。为此，现代操作系统通常支持动态链接系统库及运行时链接装载方式，不再要求启动执行程序时就已装载整个程序的所有目标模块。

为支持运行时链接地址重定位，在程序装载代码模块前缀部分不仅应包括程序名、程序大小、重定位表和执行起始地址，还应包括动态链接表以指明对哪些动态链接库的哪些函数进行调用；同时、动态链接库装载模块前缀部分也应包括程序库名、程序库大小、重定位表和动态链接表，且该动态链接表不仅应给出库中所有可共享函数的名称，还应指明其执行过程需要进一步调用和执行哪些动态链接库及函数。至于相对地址到物理地址的转换则推迟到程序执行指令时，显然，运行时链接装载方式必然采用动态重定位的地址转换方法。

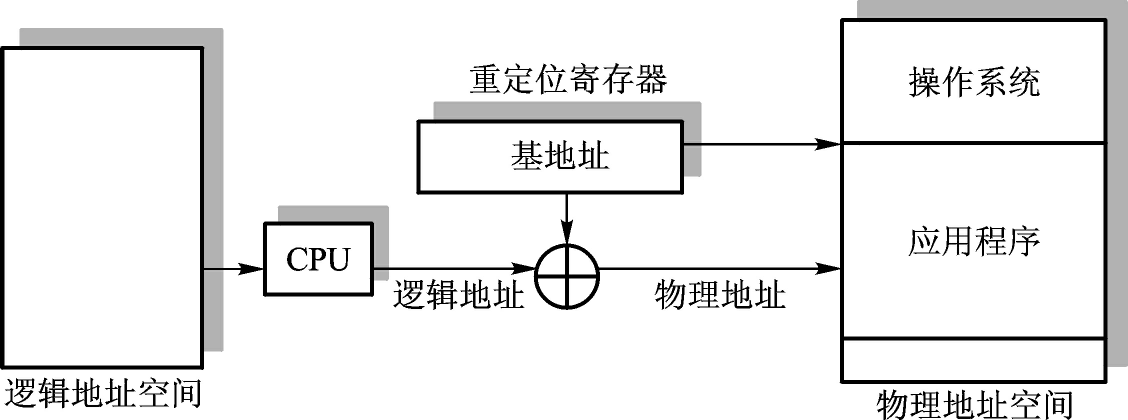


图4-2 动态地址重定位

非常有趣的一点，虚拟存储器使得动态加载可执行代码或共享代码变得十分容易。以Linux系统为例，装载程序只要为进程分配一个连续虚存页面区（从虚地址0x08048000H开始），同时将对应页表的页表项标记为“不在内存”，通过进程外页表找到目标文件中的适当位置，装载程序无须真正地从磁盘复制应用程序到内存中，当页面首次被引用时，通过缺页异常，虚存管理将自动地从磁盘把程序或数据调入内存。

**在多处理器系统之中，可获得的主存是被一定数量的处理器共享的。典型的，程序员是不可能提前知道他所编写的程序在运行的时候，还有多少的程序是在主存中存储的。进一步说，我们希望能有更多的程序可以被交换进或者交换出主存，凭借提供一个可运行程序的储备池，来最大化理器的整体性能。一旦一个程序被交换到磁盘中，很难保证当这个程序被交换回来的时候，仍然占用的是原来的内存区。因此我们就需要对进程进行重定向，把它存储在内存的另一个区域。**

**因此，我们无法提前知道程序会被放置在什么地方，而且我们还必须考虑到这种可能性，那就是程序会因为交换被移动到其他的区域。这些问题引起了一些关于地址的技术型的考量。在图表中，我们假设程序运行在一个连续的地址空间之中，显然，操作系统需要知道程序控制信息，运行栈以及程序入口的地址，因为操作系统要进行内存管理，并且负责把进程放在主存之中，这些地址很容易处理。但是，处理器还必须应对程序中存在的一些地址引用问题。在分支指令中会包含之后运行指令的地址引用，数据引用指令包含了之后被引用数据的地址。处理器硬件和操作系统软件必须能够将在程序的代码中找到的存储器引用转换为实际的物理存储器地址，反映程序在主存储器中的当前位置。**

在多道程序系统中，可用的内存空间常常被许多进程共享，程序员编程时不可能事先知道程序执行时的物理驻留位置，必须允许程序因对换或空闲区收集而被移动，这些现象都需要程序的动态地址重定位，即允许正在执行的程序在不同时刻处于内存的不同位置。从系统效率出发，动态地址重定位要借助于硬件地址转换机制来实现，重定位寄存器的内容通常保护在进程控制块中，每当执行进程上下文切换时，当前运行进程的重定位寄存器中的内容与其他相关信息一起被保护起来，新进程的重定位寄存器的内容会被恢复，这样进程就在上次中断的位置恢复运行，所使用的是与上次在此位置的同样的内存基地址。

存储保护涉及防止地址越界和控制正确存取。计算机系统中可能同时存在操作系统和多个应用程序，系统程序和多个应用程序在内存中各有自己的存储区域，各道程序只能访问自己的内存区而不能互相干扰，因此，操作系统必须对内存中的程序和数据进行保护，以免受到其他程序有意或无意的破坏。无论采用何种地址重定位方式，通常进程运行时产生的所有内存访问地址都应进行检查，确保进程仅访问自己的内存区，这就是地址越界保护。地址越界保护依赖于硬件设施，常用的有界地址和存储键。如何保证存取的正确性呢？进程在访问分配给自己的内存区时，要对访问权限进行检查，如允许读、写、执行等，从而，确保数据的安全性和完整性，防止有意或无意的误操作而破坏内存信息，这就是信息存取保护。

**4.2　连续存储管理**

**4.2.1 固定分区存储管理**

固定分区（fixed partition）存储管理又称静态分区模式，基本思想是：内存空间被划分成数目固定不变的分区，各分区大小不等，每个分区只装入一个作业，若多个分区中都装有作业，则它们可以并发执行，这是支持多道程序设计最简单的存储管理技术。早期 IBM操作系统OS／MFT（Multiprogramming with a Fixed Number of Tasks）就采用这种存储管理技术。

为了说明各分区分配和使用情况，需要设置一张“内存分配表”，记录内存中划分的分区及其使用情况。内存分配表指出各分区起始地址和长度，“占用标志”用来指示此分区是否被使用，当其值为“0”时，表明此分区尚未被占用。~~内存分配时总是选择那些“占用标志”为“0”的分区，当某分区被分配给一个长度小于或等于分区长度的作业后，则在“占用标志”中填入占用此分区的作业名。在图4-3中，第2、5分区分别被作业Job1和Job2占用，其余分区空闲，当分区中的程序执行结束归还内存区时，相应分区的“占用标志”置“0”，其占用的分区又变成空闲，可被重新分配使用。由于固定分区是预先将内存分割成若干连续区域，分割时各分区在内存分配表中可按地址顺序排列，那么，其内存分配算法就十分简单。~~

固定分区管理的一项任务是何时及如何把内存空间划分成分区。这项工作通常由系统管理员和操作系统初始化模块协同完成。系统初次启动时，管理员根据当天作业情况把内存划分成大小不等但数目固定的分区。

~~作业进入分区有两种排队策略：一是每个分区有单独的作业等待队列，调度程序选中作业后，创建用户进程并将其排入一个能够装入它的最小分区的进程等待队列尾部，当此分区空闲时，就装入队首进程执行。这样做的好处是可使装入分区浪费的空间最小，但如果等待处理的作业的大小很不均匀，将导致分区有的空闲而有的忙碌；二是所有等待处理的作业排成一个等待队列，每当有分区空闲时，就从队首起依次搜索分区长度能容纳的作业以便装入执行，为了防止小作业占用大分区，也可以搜索分区长度所能容纳的最大作业装入执行。~~

|  |  |  |  |
| --- | --- | --- | --- |
| ~~分区号~~ | ~~起始地址~~ | ~~长度~~ | ~~占用标志~~ |
| ~~1~~ | ~~8K~~ | ~~8K~~ | ~~0~~ |
| ~~2~~ | ~~16K~~ | ~~16K~~ | ~~Job1~~ |
| ~~3~~ | ~~32K~~ | ~~16K~~ | ~~0~~ |
| ~~4~~ | ~~48K~~ | ~~64K~~ | ~~0~~ |
| ~~5~~ | ~~64K~~ | ~~32K~~ | ~~Job2~~ |
| ~~6~~ | ~~96K~~ | ~~32K~~ | ~~0~~ |

~~图4-3　固定分区存储管理的内存分配表~~

固定分区能够解决单道程序运行在并发环境下不能与CPU速度匹配的问题，同时也解决了单道程序运行时内存空间利用率低的问题。缺点是：首先，由于预先已规定分区大小，使得大作业无法装入，用户不得不采用覆盖等技术加以补救，这样不但加重用户负担，而且极不方便；其次，内存空间利用率不高，作业很少会恰好填满分区~~。例如，图4-4中若Job1和Job2两个作业实际只需10 KB和18 KB的内存空间，但它们却占用16 KB和32 KB的区域，共有20 KB的内存区域占而不用被白白浪费，出现分区内的“碎片”；~~再者，如果一个作业在运行过程中要求动态扩充内存空间，采用固定分区是相当困难的；最后，分区数目是在系统初启时确定的，这就会限制多道运行程序的道数，特别不适应分时系统交互型用户及内存需求变化很大的情形。然而，固定分区方法实现简单，因此，对于程序大小和出现频率已知的情形，还是比较合适的。

**4.2.2 可变分区存储管理**

**1. 可变分区内存空间的分配和去配**

可变分区（variable partition）存储管理又称动态分区模式，按照作业大小来划分分区，但划分的时间、大小、位置都是动态的。系统把作 业装入内存时，根据其所需要的内存容量查看是否有足够空间，若有，则按需分割一个分区分配给此作业；若无，则令此作业等待内存资源。由于分区大小是按照作业的实际需求量而定的，且分区数目也是可变的，所以，可变分区能够克服固定分区内存资源的浪费问题，有利于多道程序设计，提高内存资源利用率。使用可变分区存储管理的例子是IBM操作系统OS/MVT（Multiprogramming with a Variable Number of Tasks）。

在可变分区模式下，系统初启且用户作业尚未装入内存之前，整个用户区是一个大空闲分区，随着作业的装入和撤离，内存空间被分成许多分区，有的分区被占用，而有的分区仍空闲。内存中分区数目和大小随着作业的执行而不断改变，为了方便内存空间的分配和去配，用于管理的数据结构可由两张表组成：“已分配区表”和“未分配区表”。当装入新作业时，从未分配区表中找出一个足够容纳它的空闲区，将此区分成两部分，一部分用来装入作业，成为已分配区；另一部分仍是空闲区（若有）。这时，应从已分配区表中找出一个空栏目登记新作业的起始地址、占用长度，同时修改未分配区表中空闲区的长度和起始地址。当作业撤离时，已分配区表中的相应状态改为“空闲”，而将收回的分区登记到未分配区表中，若有相邻空闲区再将其连接后登记。可变分区的回收算法较为复杂，当一个作业X撤离时，可分成４种情况：其邻近都有作业（A和B），其一边有作业（A或B），其两边均为空闲区（黑色区域）。可变分区回收情况如图４-4 所示，同时应修改内存分配表 。

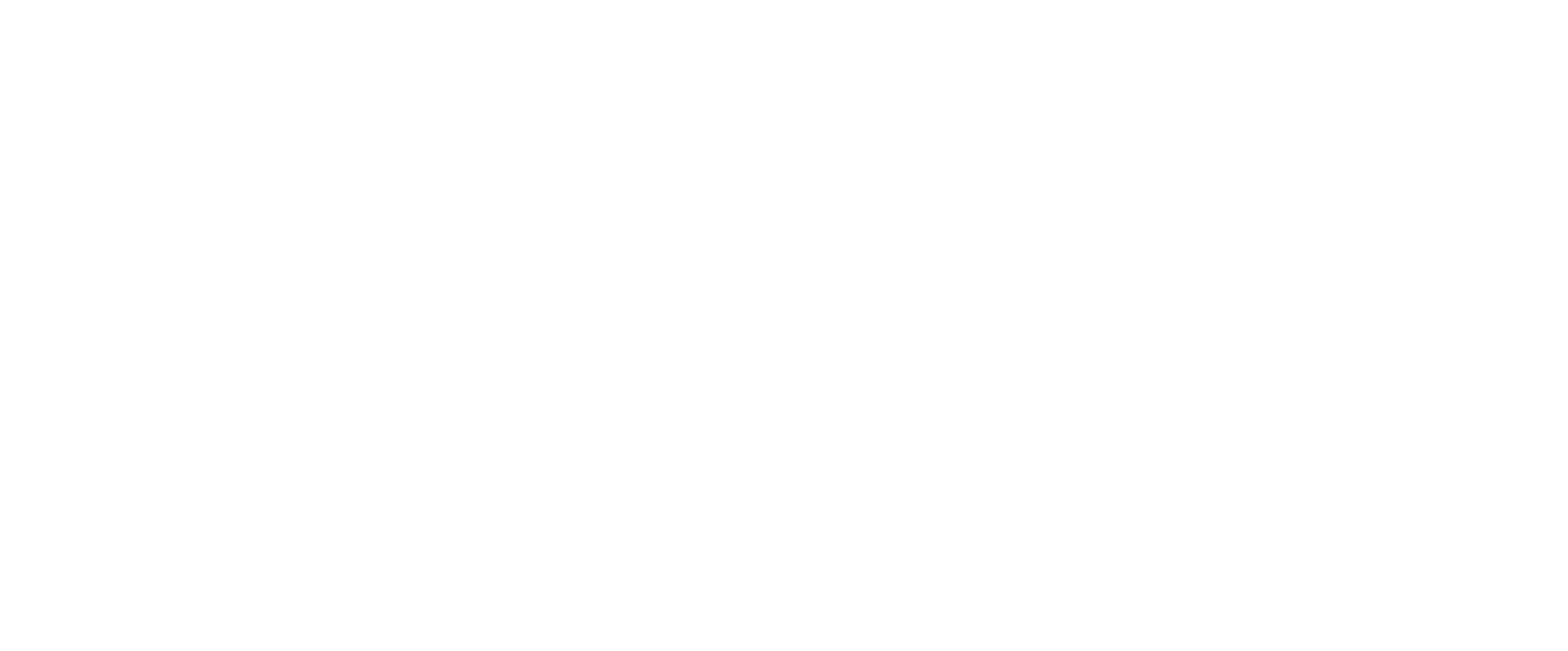


图4-4 可变分区回收情况

由于分区数目不定，采用链表是另一种较好的空闲区管理方法，用链指针把所有空闲区链接起来，每个空闲区的开头单元存放本空闲区长度及下一个空闲区起始地址指针，系统设置指向空闲区链的头指针。在使用时，沿链查找并取一个长度能满足要求的空闲区分配给进程，再修改链表；归还时，把此空闲区链入空闲区链表的相应位置即可。空闲区链表管理比空闲区表格管理要复杂，但其优点是链表自身并不占用存储单元。

无论空闲区表格管理还是空闲区链表管理，表格和链表中的空闲区都可按一定规则排列。例如，按空闲区大小，从大到小或从小到大排列；或按空闲区地址，从大到小或从小到大排列，以方便空闲区的查找和回收。常用的可变分区分配算法有以下5种：

（1）最先适应（first fit）分配算法。该算法顺序查找未分配区表或链表，直至找到第一个能满足长度要求的空闲区为止，分割此分区，一部分分配给作业，另一部分仍为空闲区（若有）。采用这一分配算法时，未分配区表或链表中的空闲区通常按地址从小到大排列。这样，为进程分配内存空间时从低地址部分的空闲区开始查找，可使高地址部分尽可能少用，以保持一个大空闲区，有利于大作业装入；但这样做会使内存低地址和高地址两端的分区利用不均衡，也将给回收分区带来麻烦，需要搜索未分配区表或链表来确定它在表格或链表中的位置且要移动相应登记项。

（2）下次适应（next fit）分配算法。该算法总是从未分配区的上次扫描结束处顺序查找未分配区表或链表，直至找到第一个能满足长度要求的空闲区为止，分割这个未分配区，一部分分配给作业，另一部分仍为空闲区（若有）。这一算法是最先适应分配算法的变种，能够缩短平均查找时间，且存储空间利用率更加均衡，不会导致小空闲区集中于内存一端。

（3）最优适应（best fit）分配算法。该算法扫描整个未分配区表或链表，从空闲区中挑选一个能满足用户进程要求的最小分区进行分配。此算法保证不会分割一个更大的区域，使得装入大作业的要求容易得到满足，同时，通常把空闲区按长度递增顺序排列，查找时总是从最小一个空闲区开始，直至找到满足要求的分区为止，这时，最优适应分配算法等同于最先适应分配算法。此算法的内存利用率好，所找出的分区如果正好满足要求则是最合适的。如果比所要求的分区略大则分割后会使剩下的空闲区很小，难以利用，其查找时间也是最长的。

（4）最坏适应（worst fit）分配算法。该算法扫描整个未分配区表或链表，总是挑选一个最大的空闲区分割给作业使用，其优点是使剩下的空闲区不致过小，对中小型作业有利。采用此分配算法可把空闲区按长度递减顺序排列，查找时只需看第一个分区能否满足进程要求，这样使最坏适应分配算法的查找效率很高，此时，最坏适应分配算法等同于最先适应分配算法。

（5）快速适应（quick fit）分配算法。该算法为那些经常用到的长度的空闲区设立单独的空闲区链表。例如，有一个n项的表，此表第一项是指向长度为2 KB的空闲区链表表头的指针，第二项是指向长度为4 KB的空闲区链表表头的指针，第三项是指向长度为8 KB的空闲区链表表头的指针，依此类推。像9 KB这样的空闲区既可放在8 KB的链表中也可放在一个特殊的空闲区链表中。此算法查找十分快速，只要按进程长度直接搜索能容纳它的最小空闲区链表并取第一块分配，但归还内存空间时与相邻空闲区的合并既复杂又费时。

由于最先适应分配算法简单、快速，在实际操作系统中用得较多，其次是下次适应分配 算法和最优适应分配算法。

**2 地址转换与存储保护**

对固定分区采用静态地址重定位，进程运行时使用绝对地址，可由加载程序进行地址越界检查。对可变分区则采用动态地址重定位，进程的程序和数据的地址转换由硬件完成，硬件设置两个专用控制寄存器：基址寄存器和限长寄存器，基址寄存器存放分配给进程使用的分区的起始地址，限长寄存器存放进程所占用的连续存储空间的长度。当进程占有CPU运行后，操作系统可把分区起始地址和长度送入基址寄存器和限长寄存器，在执行指令或访问数据时，由硬件根据基址寄存器进行地址转换得到绝对地址。当逻辑地址小于限长值时，逻辑地址加基址寄存器的值就可获得绝对地址；当逻辑地址大于限长值时，表示进程所访问的地址超出所分得的区域，此时不允许访问，达到存储保护的目的。

在多道程序系统中，硬件只需设置一对基址/限长寄存器，一个进程在执行过程中出现等待事件时，操作系统把基址/限长寄存器的内容随同此进程的其他信息，如PSW、通用寄存器等一起保存起来，另一个进程被选中执行时，则将其基址/限长值再送入基址/限长寄存器。世界上最早的巨型机CDC 6600便采用这种方案。

C语言程序被编译成至少3个段：代码段、数据段、堆栈段，UNIX进程模型是在这种模块化基础上形成的；相应地，Intel x86平台提供专用的6个段基址寄存器，由操作系统负责控制和管理。代码段寄存器CS在指令执行期间重定位指令地址，堆栈段寄存器SS为栈指令的执行重定位地址，数据段寄存器DS在指令执行周期内重定位数据地址。在有N个重定位寄存器的机器中，允许每个进程获得N个不同的内存段，并在运行时进行动态地址重定位。如果每个进程只能占用一个分区，那么，就不允许各个进程之间有公共区域，这样，当多个进程共享例行程序时就只好在各自内存区存放一套，从而，内存利用率低。提供两对或多对基址/限长寄存器的机器中，允许一个进程占用两个或多个分区。可规定某对基址/限长寄存器的区域是共享的，用来存放共享的程序和数据，当然，共享区域中的信息只能读出不能写入，于是多个用户进程共享的例行程序就可放在限定的公用区域中，如图4-5所示，让进程的共享部分取相同的基址/限长值。

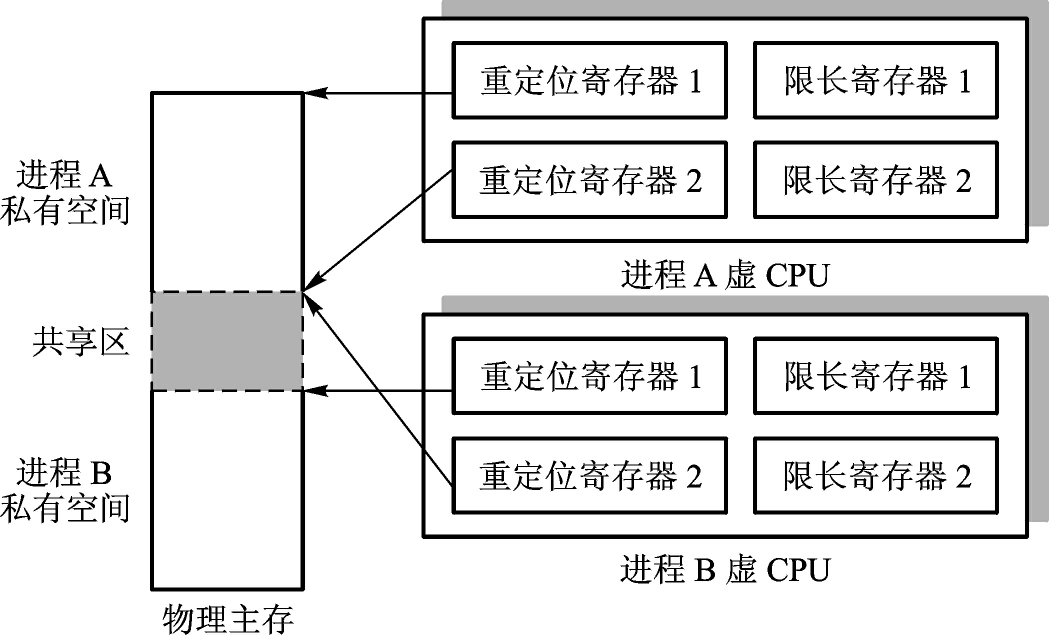


图4-5 多对重定位寄存器支持内存共享

**4.2.3 内存不足的存储管理技术**

**1. 移动技术**

可变分区法中，必须把进程装入一个连续的内存区域，由于进程不断地装入和撤销，导致内存中常常出现分散的小空闲区，称之为“碎片”。有时“碎片”会小到竟然连小进程都容纳不下，这样，不但浪费内存资源，还会限制调入内存的进程数目。

当在未分配区表中找不到足够大的空闲区来装入新进程时，可采用移动技术把已在内存中的进程分区连接到一起，使分散的空闲区汇集成片，这就是移动技术，也叫内存紧凑。第一种方法是把所有当前占用的分区内容移动到内存的一端；第二种方法是把占用分区内容移动到内存的一端，但当产生足够大小的空闲区时就停止移动。

移动操作需要把内存中的进程“搬家”，即读出每个字并写回内存，凡涉及地址的信息均应修改，如基址寄存器、地址指针等，移动分配示例如图4-6所示。移动虽然可以汇集内存空闲区，但其开销很大，现代操作系统都不再采用。“搬家”不是任何时候都能进行的，由于块设备在与内存交换信息时，通道或DMA总是按确定的内存绝对地址完成信息传输，所以，当一道程序正在与设备交换数据时往往不能移动，系统应设法减少移动，比如，在装入时总是先挑选不经移动即可装入的进程，在不得不移动时应力求所移动的道数最少。那么，何时进行移动呢？一是进程撤销之后释放分区时，如果它不与空闲区邻接，立即实施移动，于是，系统始终保持只有一个空闲区；二是进程装入分区时，若空闲区的总和够用，但没有一个空闲区能容纳此进程时实施移动。

假设进程A请求分配x KB内存区，采用移动技术分配内存空间的算法如下。步骤1：查内存分配表，若有大于x KB的空闲区，则转步骤4。步骤2：若空闲区总和小于x KB，则令进程A等待内存资源。步骤3：移动内存的相关分区信息；修改内存分配表有关项；修改被移动者的基址寄存器等信息。步骤4：分配x KB内存；修改内存分配表有关项；设置进程 A 的基址寄存器；有申请者等待时即予以释放，算法结束。

操作系统

作业1

空闲区

作业2

空闲区

作业3

空闲区

操作系统

作业1

作业2

作业3

空闲区

操作系统

作业1

作业2

作业3

空闲区

作业4

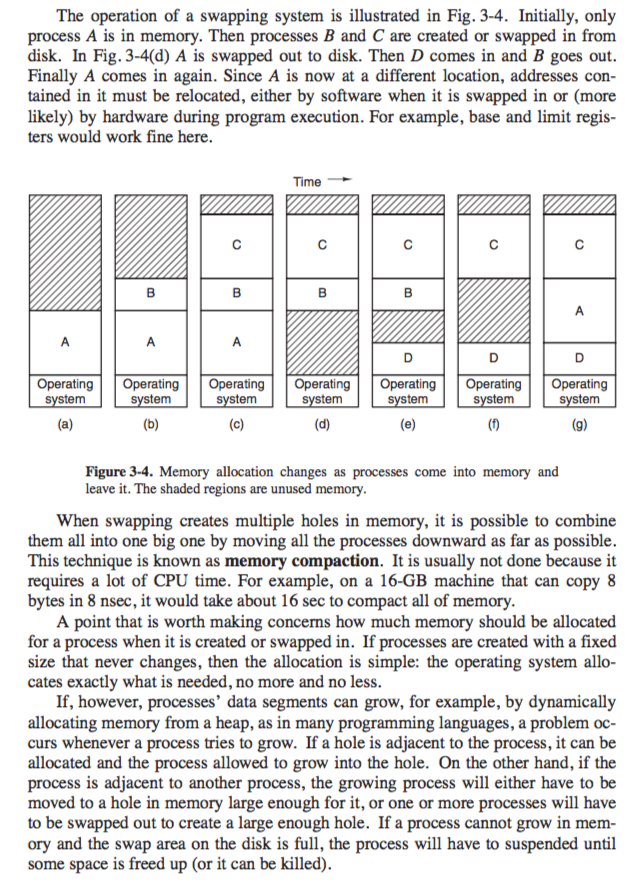
（A）原内存分配情况 （B）移动内存中的作业 （C）装入作业4

图4-6移动分配示例

移动操作也为进程运行过程中动态扩充内存空间提供了方便，当进程在执行过程中要求增加内存空间时，只需适当移动邻近的占用分区内容就可增加其所占有的连续区的长度，移动后的基址值和经扩大的限长值都应相应修改。

**2. 对换技术**

对换技术（swapping）广泛应用于分时系统的调度中，以解决内存容量不足问题，使分时用户获得快速响应时间；也可用于批处理系统，以平衡系统负载。如果当前一个或多个驻留进程都处于阻塞态，此时选择其中一个进程，将其暂时移出内存，腾出空间给其他进程使用，同时把磁盘中的某个进程换入内存，让其投入运行，这种互换称为对换。例如，当一个进程执行某系统调用时变成阻塞态，这时存储管理程序会收到进程管理的通知，以决定是否将此进程的内存映像对换到磁盘；反之，当进程管理程序把已对换出去的进程转换为就绪态时，会通知存储管理程序，一旦内存可用，立即把此进程对换回内存。由于有硬件地址重定位寄存器的支持，对换进来的进程映像被复制到新分配的内存区域并重置重定位寄存器的值。



**内存交换的过程如图所示：其中过程如下**

**a：最初只有A一个进程**

**b：B进程被创建或者从磁盘中换入**

**c：C进程被创建或者从磁盘中换入**

**d：A进程被交换进磁盘**

**e：D进程被创建，或者从磁盘中换入，占用了原来A进程的一部分内存**

**f：B进程被交换进磁盘**

**g：在B被交换之后恰好有一个较大的空间，A进程被换入**

**显然，在上述的过程中，A进程的地址整个发生了变化，在进程中的程序的运行地址也要进行重定向，这个工作可以在换入的时候由软件完成，或者是在运行的时候由硬件完成，硬件实现的例子也很简单，用简单的基址偏移寄存器即可实现。**

**观察此过程图可以发现，在内存交换的过程中可能会产生很多的空洞，处理这个问题最简单的办法就是把内存中的程序逐个向上或者向下尽可能的平移，这个技术也叫内存压缩，但是由于时间消耗比较大一般不会采用。**

**当进程被创建或者被换入的时候应当分配多少的内存，这是一个值得关注的问题。如果进程本身的大小是固定的，那么操作系统只要给他所需要的大小就足够了，然而有的进程本身的段会增长，（很多编程语言都支持动态分配堆内存，这就会引起数据段的增长）这时候问题就出现了。如果说一个进程紧邻的是一个空闲区域，进程在段增长的时候就可以占用这块内存，但是如果进程需要段增长的时候紧邻着另一个进程呢？这时候就需要把进程移动到另一个足够大小的空间中或者把相邻的空闲进程交换出去了。**

为了有效地实施对换，首先，选择哪个进程换出。如果选择不当，将造成系统效率欠佳。通常系统把时间片耗尽或优先级较低的进程换出，因为短时间内它们不会投入运行。

其次，决定把进程的哪些信息移出去。开始时，进程从可执行文件被装入内存，其未修改部分（如代码）在内存与磁盘中始终保持一致，这些信息不必保存，当进程换回内存时，只需简单地从最初的可执行文件再加载一次。数据区和堆栈是进程运行时所创建和修改的，操作系统可通过文件系统把这些可变信息作为特殊文件保存。有些系统从降低开销角度考虑，开辟一块特殊磁盘区域作为对换空间，它包含连续的柱面和磁道，可通过磁盘读写实现高效访问。

最后，需要确定对换时机，在批处理系统中，当进程要求动态扩充内存空间且得不到满足时可触发对换；在分时系统中，对换可与调度结合，每个时间片结束或执行I/O操作时实施，调度程序启动一个挪出的进程换入，这样，轮到它执行时立即可以启动，对换进内存的进程其内存位置未必还在换出之前的位置上，所以，需要解决对换过程中进程的地址重定位问题。假设一个被对换的进程映像占用k个磁盘块，那么，一次进程对换的所有开销是2 k个磁盘块I/O的时间，再加上进程重新请求内存资源所造成的时间延迟。对换比移动技术更有效，移动不能保证得到一个满足请求的空闲区，而利用对换技术总可按需挪出若干驻留的阻塞进程，且对换仅涉及少量进程，只需更少的内存访问。与移动不同的是，对换要访问磁盘，这是一个I/O集中型操作，会影响对用户的响应时间，但系统可让对换与计算型进程并行工作，不会造成系统性能的显著下降。

UNIX早期版本通过称作对换器的专门进程实施对换，每当创建新进程、进程动态扩充内存空间时便挪出一个或多个驻留进程，每隔4 s，对换器进行检查以保证把挪出已久的进程换进。换出的候选者当首选被阻塞的进程，否则就挑选就绪进程。需要考虑进程属性，如已消耗CPU时间、在内存已逗留时间等。

**3. 覆盖技术**

移动和对换技术解决因多个程序存在而导致内存区不足问题，这种内存短缺只是暂时的；如果程序长度超出物理内存总和，或超出固定分区大小，则出现内存永久性短缺，大程序无法运行，前述两种方法无能为力，解决的一种方法是采用覆盖 （overlaying）技术。覆盖是指程序执行过程中程序的不同模块在内存中相互替代，以达到小内存执行大程序的目的，基本的实现技术是：把用户空间分成固定区和一个或多个覆盖区，把控制或不可覆盖部分放在固定区，其余按调用结构及先后关系分段并存放在磁盘上，运行时依次调入覆盖区。系统必须提供覆盖控制程序及相应的系统调用，当程装入运行时，由系统根据用户给出的覆盖结构进行覆盖处理，程序员必须指明同时驻留在内存的是哪些程序段，哪些是被覆盖的程序段，这种声明可从程序调用结构中获得。覆盖技术的不足是把存储管理工作转给程序员，他们必须根据可用物理内存空间来设计和编写程序。此外，同时运行的代码量超出内存容量时仍不能运行，所以，现代操作系统极少采用覆盖技术。

**4.3 分页存储管理**

**4.3.1 分页存储管理基本概念**

用分区方式管理存储器，每道程序要求占用内存的一个或多个连续存储区域，导致内存中产生“碎片”。有时为了接纳新作业，往往需要移动已在内存的信息，这样不仅不方便，而且处理器开销太大。采用分页存储管理允许程序存放到若干不相邻的空闲块中，既可免除移动信息工作，又可充分利用内存空间，消除动态分区法中的“碎片”问题，从而，提高内存空间利用率，分页存储是一种特殊的动态分区方式。分页存储管理涉及的基本概念如下。

1. 页面。进程逻辑地址空间分成大小相等的区，每个区称为页面或页，页号从0开始依次编号。

2. 页框。页框又称页帧，把内存物理地址空间分成大小相等的区，其大小与页面大小相等，每个区是一个页框（物理块），块号从0开始依次编号。

3. 逻辑地址。与此对应，分页存储器的逻辑地址由两部分组成：页号和页内位移，格式如下：

页号 页内位移

前者表示地址所在页面的编号，后者表示页内位移。计算机地址总线通常是32位，页面尺寸若规定为12位（页长4 KB），那么，页号共20位，表示地址空间最多包含220个页面。

4. 内存页框表。该表长度取决于内存划分的物理块数，编号可与物理块号一致，页框表的表项给出物理块使用情况，0为空闲，1为占用。有的系统还会增加保护位。

5. 页表。在进行内存分配时，以页框为单位，进程的信息有多少页，那么，把它装入内存时就分配多少页框，虽然进程的逻辑地址划分成编号连续的页面，但被装入内存后的相应页框未必紧邻，即进程的信息按页面分散存放在内存不相邻的页框中。那么，当进程的程序和数据被分散存放在内存中后，其页面与被分配的页框如何建立联系呢？逻辑地址（页面）如何转换成物理地址（页框）呢？进程被装入后的物理地址空间由连续变成分散后，如何保证程序正确执行呢？仍然采用动态地址重定位技术，让程序在执行时动态地进行地址变换，由于程序以页面为单位存储，所以，每个进程都会维护一个记录页信息的表格，其中，每一个页的信息由特殊的数据结构存储，这种数据结构叫做页表项（page table entry），这些页信息表格的整体则称为页表（page table）

~~所以，为每个页面设立一个重定位寄存器，这些重定位寄存器的集合称为页表（page table）。~~

**6.页表项：**



**页表项是页表中所存储的数据单元，不同的硬件架构下，页表项的格式也各不相同，但是页表项所提供的信息基本相似。如下表所示，页表项中最重要的部分就是页框号部分，因为页表项最基本的作用就是在请求页的时候返回这个值。**

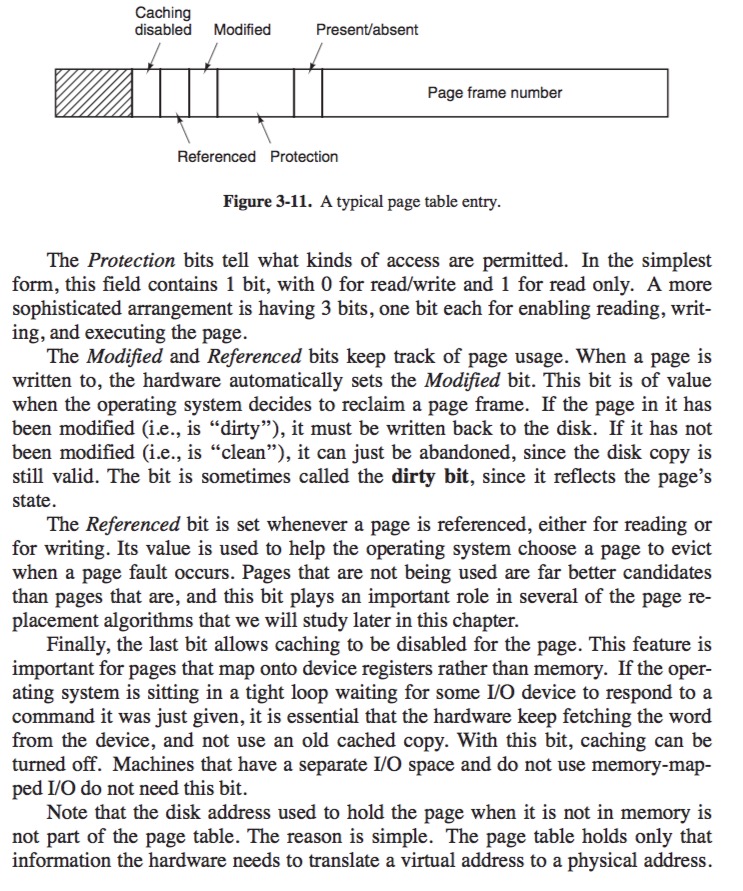
**驻留标志位表示的是该页是否在内存中，为1表 示该页在内存中，可用，为0表示该虚拟页地址不在内存中，访问一个可用位为0的页会引起缺页错误。**

**保护位的作用是声明对该页可用的访问的类型，最简单的情况是只有一位来表示，0表示可读可写，1表示只读。更为复杂的情况则是用三个位来分别表示是否可读，是否可写和是否在运行。**

**修改和引用位用于跟踪页面的使用。当页面发生了写入，硬件会自动更新页表项中的修改位。修改位的作用主要体现在操作系统要替换页框的时候，如果页面是已经被修改过的（“脏页”），就需要被写回到磁盘中，不然的话就没有被写回的必要了。**

**引用位用于标志页面是否被引用，引用的方式分为读和写两种。引用位的主要作用在于帮助操作系统决定在发生缺页错误的时候应该把哪些页替换出去。那些从来没有被使用过的页显然更适合被替换出去。许多页面替换算法都会使用到引用位。**

//从页式管理的重定向，保护，共享等方面给出一个概念性的页表项格式



页表是操作系统为进程建立的，是程序页面和内存页框的对照表，页表中的每一栏指明程序中的一个页面和分得页框之间的对应关系。使用页表的目的是把页面映射为页框，从数学角度而言，页表是一个函数，其变量是页面号，函数值为页框号，通过页表可以把逻辑地址中的逻辑页面域替换成物理地址中的页框域。为了减少系统开销，不用硬件而是在内存中开辟存储区以存放进程页表，系统另设置专用硬件--页表基址寄存器，存放当前运行进程的页表起始地址，以加快地址转换速度。系统应为内存中的进程进行存储分配并建立页表，页表长度随进程大小而定。采用分页存储管理时，逻辑地址是连续的，用户在编制程序时仍使用相对地址，不必考虑如何分页，由硬件地址转换机构和操作系统的管理需要来决定页面尺寸，从而，确定内存分块大小。进程在内存中的每个页框内的地址是连续的，但页框之间的地址可以不连续，进程映象在内存的存放地址由连续到离散的变化为虚拟存储器的实现奠定了基础。

4.3.1分页存储的地址转换

分页作为一种动态分配方式，其地址转化方式和动态分配方式相似。

首先，进程运行前由系统把它的页表基地址送入页表基址寄存器，运行时借助于硬件地址转换机构，按页面动态地址重定位，当CPU获得逻辑地址后，由硬件自动按设定的页面尺寸分成两部分：页号p和页内位移d，先从页表基址寄存器找到页表基地址，再用页号p作为索引查页表，得到对应的页框号，根据关系式：

物理地址＝页框号 × 块长＋页内位移

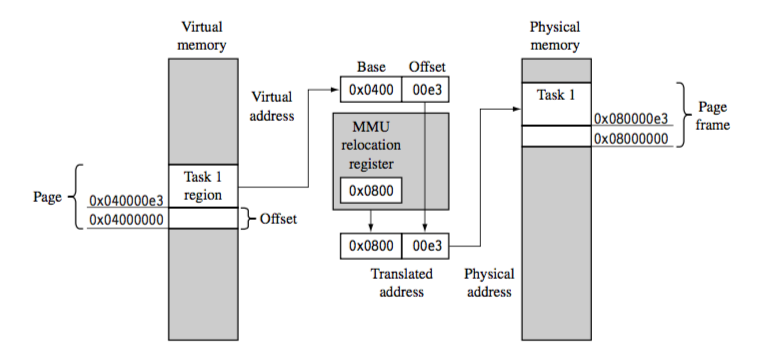
计算出欲访问的内存单元地址。因此，虽然进程存放在若干不连续的页框中，但在执行过程中总能按正确物理地址进行存取。~~如图4-7所示是分页存储管理的地址转换，在实际进行地址转换时，只要把逻辑地址中的页内位移d作为绝对地址中的低地址，根据页号p从页表中查得页框号b作为绝对地址中的高地址，拼接后就组成访问内存的绝对地址。~~

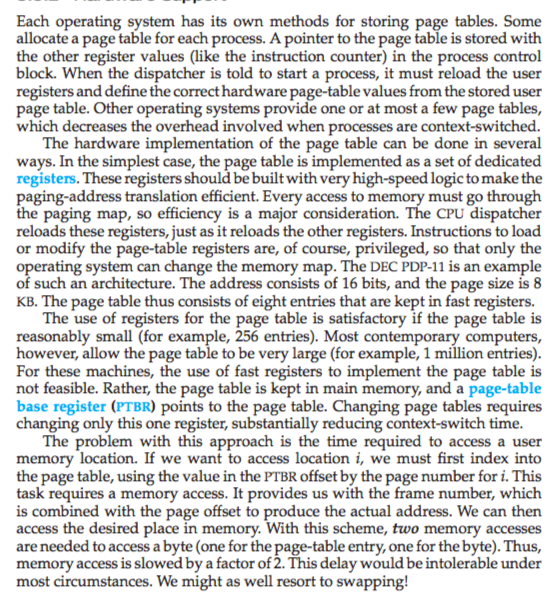
整个系统只有一个页表基址寄存器，只有占用CPU的进程才占有它。在多道程序中，当某道程序让出处理器时，应同时让出此寄存器供其他进程使用。

**当一个处理器核心生成一个虚拟地址的时候，MMU会取得他的顶部位，并用重定向寄存器中存储的物理基址代替，虚拟地址的地位则表示的是偏移，用来进一步定位在物理内存中的位置。**

**下图展示了ARM架构的计算机中将虚拟内存中地址为0x400000的任务被编译运行的过程，重定向寄存器把这个虚拟地址映射到0x800000的物理地址**

**假如有另一个任务，编译运行在同一个虚拟地址0x400000，并不会产生冲突，因为分页的起始地址不同，所存在的物理地址就不同。只要存储的位置是0x10000 64kb的倍数。都可以通过更改重定向寄存器的内容进行访问。**





每一个操作系统对于存储页表都以自己的方法，有的操作系统为每一个进程分配一个页表，并在其他的寄存器中存储这个页表的指针，这个寄存器的值就存储在PCB中，当有调度器来唤起这个进程的时候，他就会重新装载用户的寄存器并且从存储好的用户页表中正确的设置页表有关的硬件，其他的操作系统提供一个或者少量的几个页表，从而减少上下文切换带来的消耗。

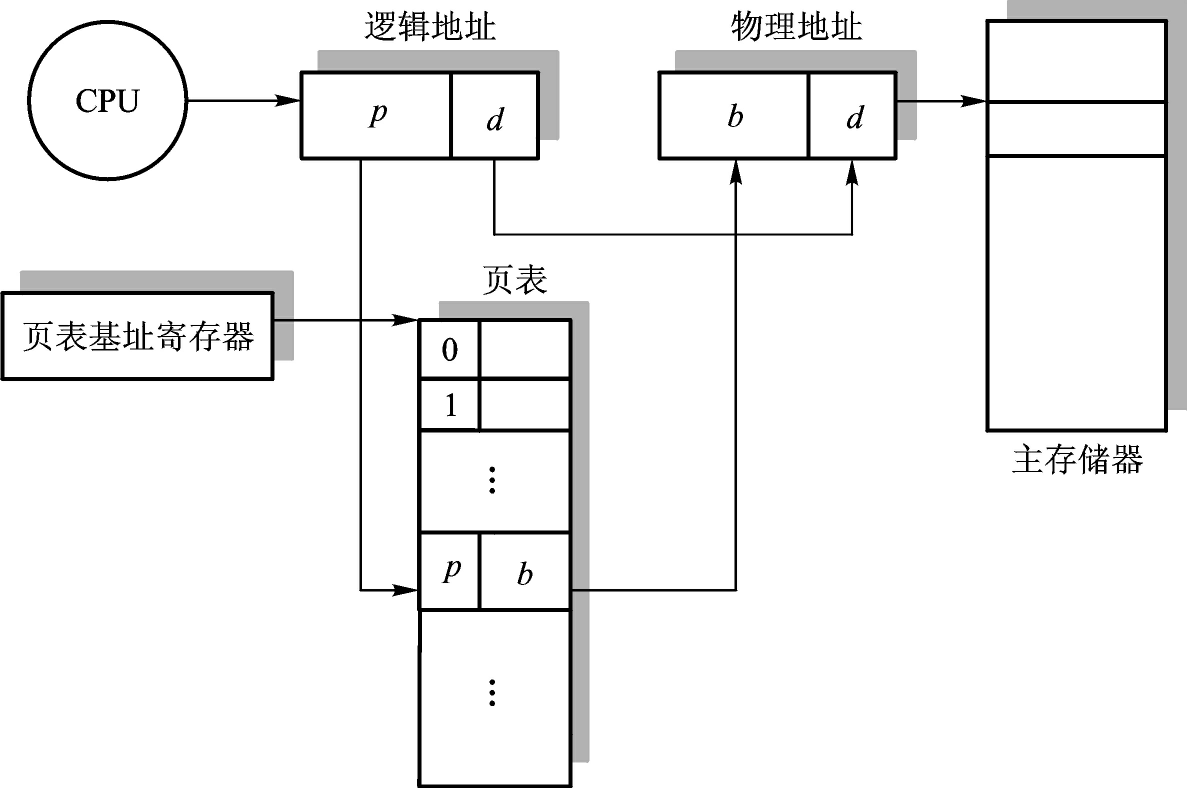


图4-7 分页存储管理的地址转换

硬件实现可以若干方式实现，最简单的方式就是把页表用若干的专用的寄存器存储，这些寄存器必须要用很快的硬件逻辑实现，从而使得页地址转换的效率足够高，对内存的每一个访问都必须经过地址映射，所以转换的效率就是第一位要考虑的问题，CPU调度器像重载其他寄存器一样重载这些寄存器，修改页表寄存器的指令必须在保护模式下运行，所以只用操作系统才可以修改地址映射。

但是使用寄存器存储的方式仅仅可以满足小体量的程序的要求，面对占用大量内存的程序，一般采用的方式是用一个寄存器指向这个进程的页表的起地址，（PTBR），这样做的另一个好处就是在上下文切换的时候只需要改变一个寄存器的值就可以了，减少切换的时间。

虽然减少了切换的时间，但是这种设计获取用户所需要的数据的效率很低，因为要取得一个数据必须要首先根据页表号从内存中的页表得到页号，再根据页号得到需要的数据。这整个过程进行了两次内存访问，很多情况下这样做产生的时延是巨大的。

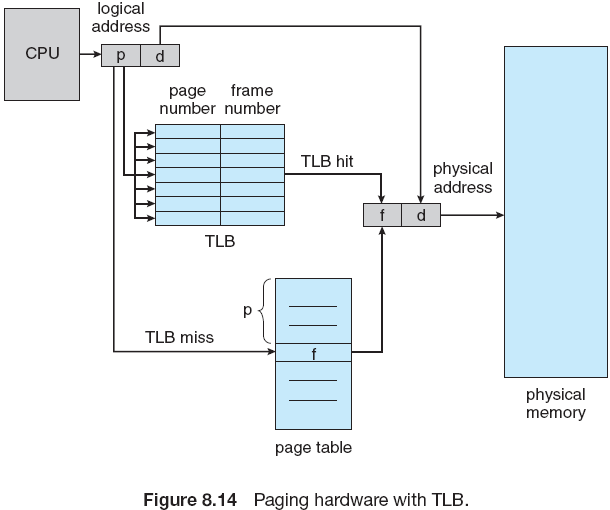
**4.3.2 翻译快表**

页表可存放在一组寄存器中，地址转换时只要从相应寄存器中取值就可得到页框号，这样做虽然能加快地址转换，但硬件代价太高；页表也可存放在内存中，这样做可以降低系统开销，但是按照给定逻辑地址进行读写操作时，至少访问内存两次：一次访问页表，另一次访问指令或存取数据，这将降低运算速度，比~~通常执行指令~~不采用分页方式时的速度慢一半。

为了提高运算速度，在硬件中设置相联存储器，用来存放进程最近访问的部分页表项，也称转换后援缓冲（Translation Look\_aside Buffer，TLB）或翻译快表，它是分页存储管理的重要组成部分。**TLB是一组高速的关联存储区，TLB中的每一个表项都是一个键值对。当接收到一个地址的时候，关联存储器会同时比较所有的键，如果发现了对应的键，就会返回对应的值。这个搜索过程是非常迅速的，在现代的硬件架构中，TLB查找是作为流水线的一个部分，本质上不会产生任何的性能损失。为了保证TLB查找能在一个流水线阶段中完成，TLB的大小一般都在32-1024个表项之间。**

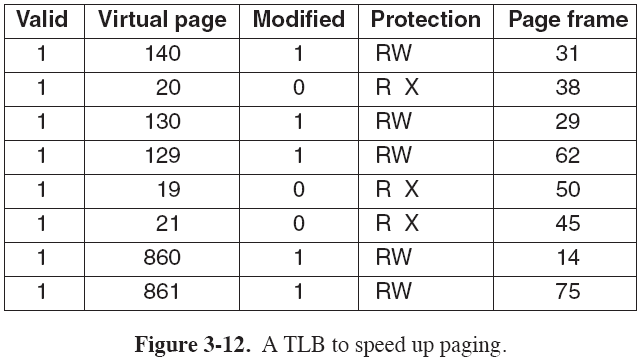
**TLB和页表是按照如下方式运行的，TLB中包含了一部分的页表项，当CPU产生了一个逻辑地址之后，逻辑上的也好就会被交给TLB，如果在TLB中发现了该页，这个页的页框号就会被返回并用于访问主存，就像上文中提到的，这些步骤都是在一个处理器流水线中完成的，相比于没有采用分页机制的系统，不会产生任何的时间损耗。如果TLB没有包含对应的页号（也被称作快表未命中），这时就需要对内存中的页表进行一次访问，这也可以通过硬件中断来实现。在得到了页框号之后，系统就可以访问对应的内存了，除此之外，系统还会把这次未命中的页表项添加到TLB中以便下次能快速访问到。**

**当快表的被填满时，如果有新的表项需要加入，就需要采用替换策略来替换出可能较少使用或者不使用的表项。在这个过程中采取的策略一般有先进先出，最近最久未使用等。另外，有的页表项也会长期驻留在快表中，无论按照替换策略是否应当将其替换出去，典型的，操作系统的核心代码所在页的页表项就会常驻在快表中。**

****

**//Operating System Concepts (9th Edition), page 373**

**//TLB示意**

****

**//Modern Operating Systems (4th edition), page 203**

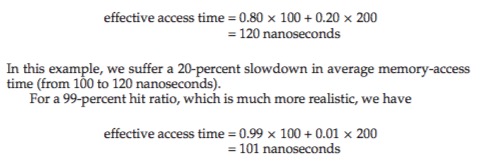
**//TLB示意**

~~快表的存取时间远小于内存，速度快但造价高，故容量较小，只能存放几十个页表项。快表项包含页号及对应页框号，当把页号交给快表后，它通过并行匹配同时对所有快表项进行比较，如果找到，则立即输出页框号，并形成物理地址；如果找不到，再查内存中的页表以形成物理地址，同时将页号及页框号登记到快表中；当快表已满且要登记新页时，系统需要淘汰旧的快表项，最简单的策略是“先进先出”，总是淘汰最先登记的页面。~~

通过快表实现内存访问的比率称为命中率，命中率越高，性能越好，接近100%的命中率表明绝大部分内存访问都通过快表实现，几乎不用页表；反之，当进程访问遍布内存页面的跳跃性地址时，命中率近乎为0，这意味着每次访问内存都要使用页表。采用快表后，地址转换时间大大下降，假定访问内存的时间为100 ns，访问快表的时间为20 ns，根据统计数据获得——当快表为32个单元时的查找命中率为90%，内存数据地址要先存入快表，然后，再由处理器存取。于是，按逻辑地址进行存取的平均时间为：

(100＋20)×90%＋(100＋100＋20)×(1－90%)＝130 ns

比两次访问内存的时间200 ns 缩短35%。



First thing first. *CPU Cache* is a fast memory which is used to improve latency of fetching information from Main memory (RAM) to CPU registers. So CPU Cache sits between Main memory and CPU. And this cache stores information temporarily so that the next access to the same information is faster. A CPU cache which used to store executable instructions, it’s called Instruction Cache (I-Cache). A CPU cache which is used to store data, it’s called Data Cache (D-Cache). So I-Cache and D-Cache speeds up fetching time for instructions and data respectively. A modern processor contains both I-Cache and D-Cache. For completeness, let us discuss about D-cache hierarchy as well. D-Cache is typically organized in a hierarchy i.e. Level 1 data cache, Level 2 data cache etc.. It should be noted that L1 D-Cache is faster/smaller/costlier as compared to L2 D-Cache. But the basic idea of ‘*CPU cache*‘ is to speed up instruction/data fetch time from Main memory to CPU.

*Translation Lookaside Buffer (i.e. TLB)* is required only if Virtual Memory is used by a processor. In short, TLB speeds up translation of virtual address to physical address by storing page-table in a faster memory. In fact, TLB also sits between CPU and Main memory. Precisely speaking, TLB is used by MMU when physical address needs to be translated to virtual address. By keeping this mapping of virtual-physical addresses in a fast memory, access to page-table improves. It should be noted that page-table (which itself is stored in RAM) keeps track of where virtual pages are stored in the physical memory. In that sense, TLB also can be considered as a cache of the page-table.

But the scope of operation for *TLB* and *CPU Cach*e is different. TLB is about ‘speeding up address translation for Virtual memory’ so that page-table needn’t to be accessed for every address. CPU Cache is about ‘speeding up main memory access latency’ so that RAM isn’t accessed always by CPU. TLB operation comes at the time of address translation by MMU while CPU cache operation comes at the time of memory access by CPU. In fact, any modern processor deploys all I-Cache, L1 & L2 D-Cache and TLB.

//CPU cache与TLB的区别

cache作为cpu中的临时存储器，存储的是指令或者数据，用以加速CPU的速度，但是TLB则是为了加速mmu检索页地址的速度

**处理器高速缓存（cache）和TLB是两个比较容易混淆的概念，但是两者在作用和工作方式上都有较大的区别，处理器高速缓存是存在于内存和CPU之间的存储部件，其存储的内容一般是将要运行的指令或者是将要使用的数据，而TLB则是存在于MMU内部，其作用是为了加速页地址的查找，在TLB中存储的是包含页信息的页表项。**

**现代的处理器架构渐渐开始采用分级的TLB来加速页表的访问，分为指令TLB和数据TLB，这样可以获得双倍的可用表项，因为指令和数据的查找在流水线中是两个分开的过程，在一级快表中未命中的页表项，会消耗6个时钟周期在二级快表中寻找，而在二级快表中仍未命中的数据则会消耗上百个时钟周期来通过访问内存取得具体地址。**

**快表是一个硬件的功能，所以对于操作系统的设计者以及操作系统本身来说可能意义并不大，但是设计者必须了解快表的作用以及功能，不同的硬件平台会提供不同的快表功能，而相应的操作系统也应当根据快表的设计方式来使用分页功能。同样的，硬件平台在快表设计上的变化也会对上层操作系统的设计产生影响。**

**4.3.3 分页存储空间分配和去配**

分页存储管理中，系统要建立一张内存物理块表，用来记录页框状态，管理内存物理块的分配，所包含的信息有内存总块数、哪些为空闲块 、哪些已分配及分给哪个进程等，最简单的方法可用位示图来记录分配情况，每位与一个页框相对应，用0/1表示对应块为空闲/已占用，用另一专门字记录当前空闲块数，如图4-8（a）(b)所示。

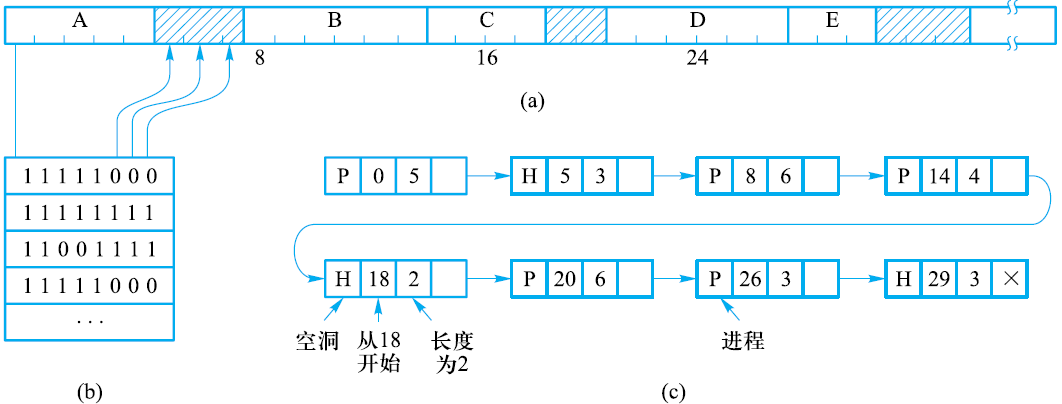


图4-8 内存分配的位示图和链表方法

分页存储管理页框分配算法如下：进行内存分配时，先查空闲块数能否满足用户进程的要求，若不能，令进程等待；若能，则查位示图，找出为“0”的那些位，置占用标志，从空闲块数中减去本次占用块数，按所找到的位的位置计算对应页框号，填入此进程的页表。进程执行结束归还内存时，根据归还的页框号，计算出对应位在位示图中的位置，将占有标志清“0”，并将归还块数加入空闲块数中。

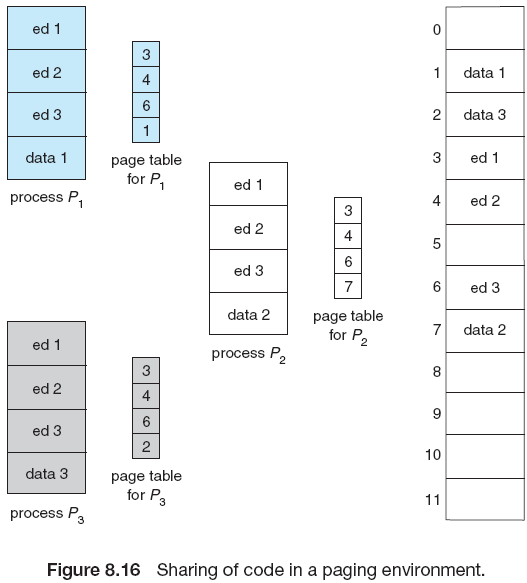
图4.8（c）是内存分配的链表方法，表中各项包含以下内容：是进程占用区（P）还是空闲区（H）、起始地址、长度和指向下一表项的指针。在本例中，链表按照地址从小到大排序，其优点是链表的更新和修改比较方便，运行结束的进程通常有两个邻居，既可能是进程也可能是空闲区，只需修改邻近的链表项 。

**4.3.4 分页存储空间页面共享和保护**

**1. 页面共享和保护**

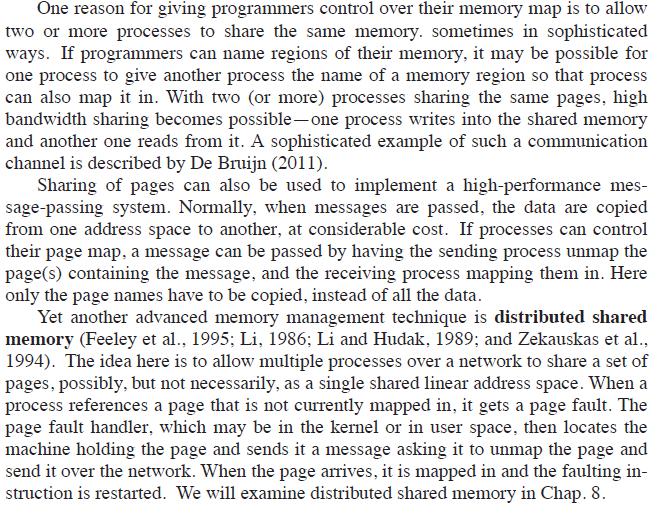
在多道程序系统中，编译程序、编辑程序、解释程序、公共子程序、公用数据等都是可共享的，这些共享信息在内存中只需保留一个副本，分页存储管理能实现页面共享，共享页面信息可大大提高内存空间利用率。

实现程序共享时，由于指令包含指向其他指令或数据的地址，程序依赖于这些地址才能执行。对于采用静态链接的页式存储管理要求逻辑地址空间必须连续，也就是说，程序中每个逻辑地址的页号是唯一和确定的，因而，多个进程只能以相同页号来共享程序。假定有一个被共享的编辑程序EDIT，共有三页，现有二进程共享它。P1和P2的逻辑空间中均空出初始部分0页、1页和2页，而内存管理让这三页分别指向驻留在内存中的4、7、9页框中的EDIT。二个进程执行时用自己的页表进行地址映射，物理地址并不连续但逻辑地址是连续的，于是便可正确实现两个进程对编辑程序EDIT的页面共享。

****

**//Operating System Concepts (9th Edition), page 377**

**//页的共享**

****

**//之所以给程序员控制内存堆的权限就是尽可能的让两个进程共享内存**

**Modern Operating Systems (4th edition)P233**

实现信息共享必须解决共享信息保护问题。通常的做法是在页表**项**中增加标志位，指出此页的信息只读/读写/只可执行/不可访问等，进程访问此页时核对访问模式。例如，欲向只读块写入信息则指令停止执行，产生违例异常信号。另外，也可采取存储保护键作为保护机制，本书第七章将介绍IBM OS/370系列操作系统的存储保护键保护机制。

**2. 运行时动态链接**

当进程需要使用各种标准库函数时，需要采用静态方式全部链接到应用程序中，每个可执行代码中都有库函数的副本，这样就增加了对内存容量的要求。如果程序仅使用其中一小部分，采用静态链接不但麻烦，而且开销大，影响系统效率。为此，可把函数定位和链接推迟到运行时刻，只在实际调用发生时才进行。

运行时动态链接需使用共享库（shared library），它包含共享函数的目标代码模块，在运行时可加载到任意内存区域，并在内存中和一个程序链接起来，这个过程称为动态链接（dynamic linking），这是通过动态链接器（dynamic linker）来执行的。在Linux 系统中，共享库的共享代码通常用后缀.so来表示，对于一个库只有一个.so文件，所有引用此库的可执行目标代码共享此.so文件中的代码和数据，而不是像静态链接那样被复制和嵌入引用它们的可执行应用程序中 。

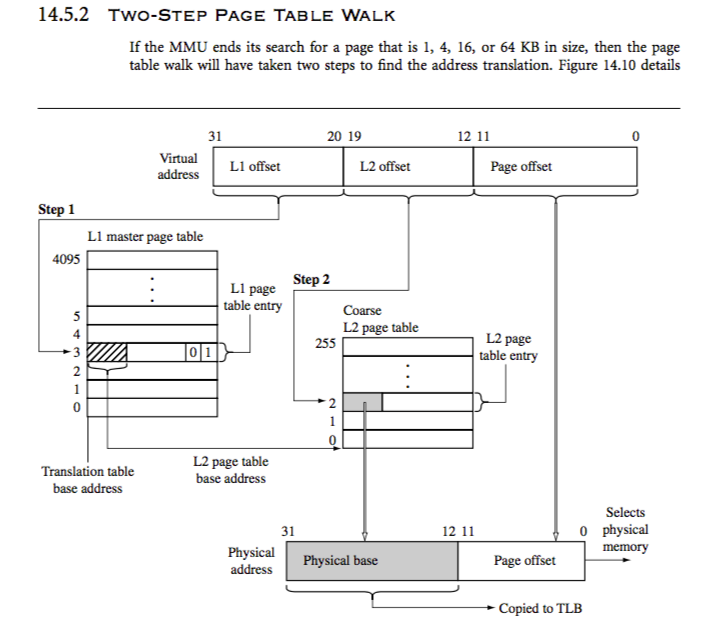
假如，应用程序main1.c需要使用库函数，头文件中包含函数原型stdio.h等定义，下面列出编译和动态链接共享库的过程。（1）编译时，给出main1.c，并包含＃include ＜stdio.h＞ 等头文件。（2）链接器对编译输出信息main1.o和标准共享库libc.so的重定位和符号表信息进行静态链接。获得部分链接的可执行目标代码命名为Exmain1。（3）当装载器（execve()）加载和运行Exmain1时，发现包含动态链接器的路径名，动态链接器本身是一个共享目标代码（如Linux系统上的LD-LINU X.so），装载器不像通常那样将控制传递给应用程序，取而代之的是加载和运行这个动态链接器。（4）动态链接器通过执行下面的重定位完成链接任务。① 重定位libc.so的文本和数据到某个内存段。在Linux系统中，标准共享库被加载到从地址0x40000000H0 开始的区域中。② 重定位 Exmain1中所有对由libc.so定义的符号的引用。③ 动态链接器将控制传递给应用程序，从此时开始，共享库的位置便固定，并在程序执行过程中不会再改变。

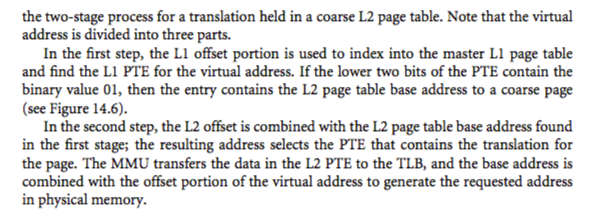
**4.3.5 多级页表**

现代计算机普遍支持232～264容量的逻辑地址空间，采用分页存储管理时，页表相当大，以Linux系统为例，其运行的Intel x86平台具有32位地址，规定页面4 KB(212)时，那么，4 GB(232)的逻辑地址空间由1兆(220)个页组成，若每个页表项占用4个字节，则需要占用4 MB(222)连续内存空间存放页表，这还是一个进程的地址空间，对于地址空间为64位的系统来说，问题将变得更加复杂。为此，页表和页面一样也要进行分页，这就形成了多级页表概念。具体做法是：把整个页表分割成许多小页表，每个称为页表页，它的大小与页框长度相同，于是，每个页表页含有若干个页表表项。页表页从0开始顺序编号，允许被分散存放在不连续的页框中，为了找到页表页，应建立地址索引，称为页目录表，其表项指出页表页起始地址。系统为每个进程建一张页目录表，它的每个表项指出一个页表页，而页表页的每个表项给出页面和页框的对应关系，页目录表是一级页表，页表页是二级页表，共同构成二级页表机制。于是，逻辑地址结构由三部分组成：页目录位移、页表页位移和页内位移。

如图4-9所示是二级页表实现逻辑地址到物理地址转换过程，具体步骤如下：由硬件页目录表基址寄存器指出当前运行进程的页目录表的内存起始地址，加上“页目录位移”作为索引，可找到页表页在内存的起始地址，再以“页表页位移”作为索引，找到页表页的表项，此表项中包含一个页面对应的页框号，由页框号和“页内位移”便可生成物理地址。

上述方法能解决分散存放页表页问题，并未解决页表页如何占用内存空间问题，解决方法如下：进程运行涉及页面的页表页应存放在内存，而其他页表页使用时动态调入，为此，需要在页目录表中增加标志位，指示对应页表页是否已调入内存，地址转换机制根据逻辑地址中的“页目录位移”来查页目录表对应表项的标志位，如未调入，应产生“缺页表页”异常信号，请求操作系统将页表页调入内存。





在进行二级也变转化的过程中，逻辑地址被分割成了三个部分，在第一步中L1偏移被用作在L1页表中检索出L1页表项，如果L1页表中显示这个页表是一个二级页表的指针，那么这个就包含了L2页表的基地址，

第二部和第一步的类似，利用逻辑地址中的L2偏移找到L2页表中的页表项，MMU把L2页表中的页表项转化成实际地址，并且把这个页的实际地址拷贝到TLB中，这个页的实际地址作为基址加上页内偏移就产生了该页的物理地址

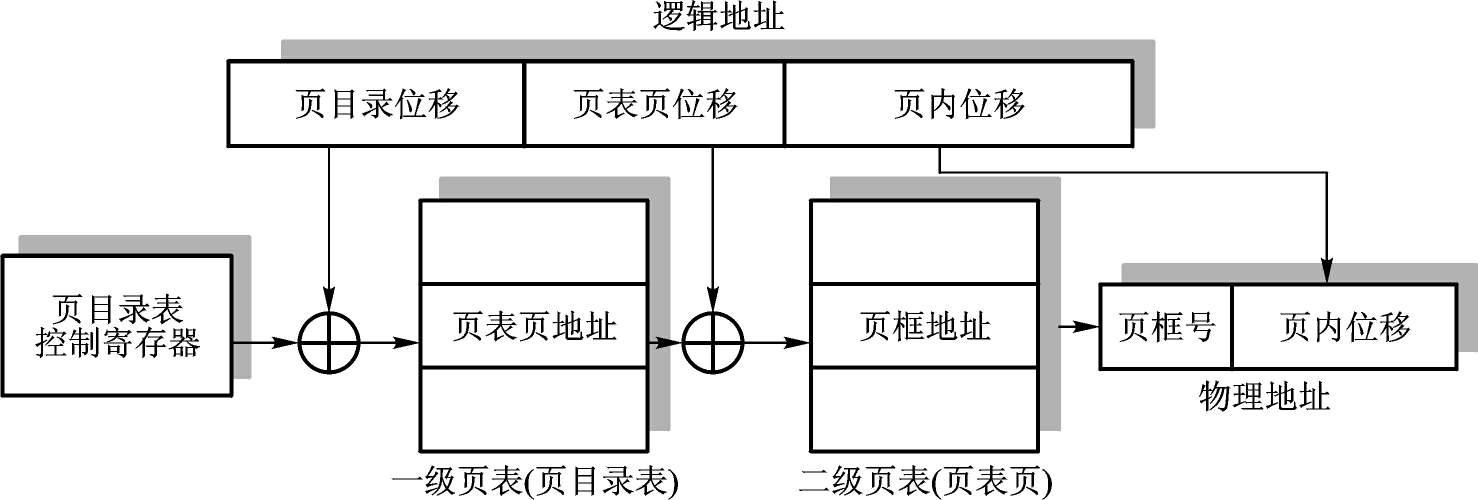


图4-9 二级页表地址转换过程

二级页表地址转换需3次访问内存，一次访问页目录、一次访问页表页、一次访问指令 或数据，随着64位地址的出现，二级页表仍不够用，所以，三级、四级页表也已被引入系统。多级页表结构的本质是多级不连续，导致多级索引。以二级页表结构为例，应用程序的页面不连续存放，需要有页面地址索引，此索引就是进程页表。由于进程页表是不连续存放的多个页表页，故这些页表页也需要页表页地址索引，此索引就是页目录，这就形成二级索引，页目录项是页表页的索引，而页表页项是程序页面的索引。

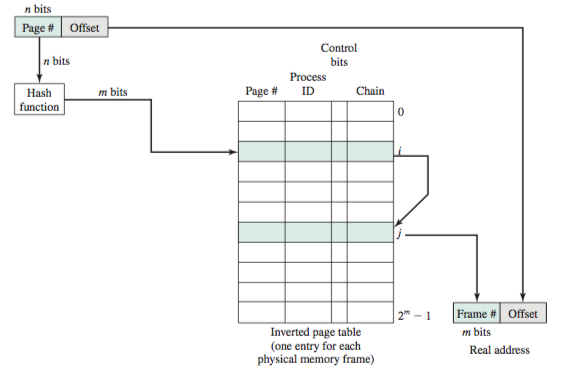
**4.3.6 反置页表**

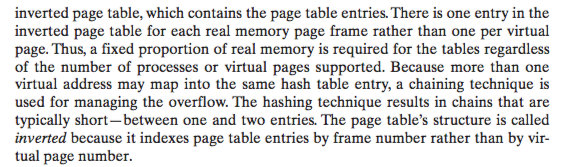
计算机逻辑地址空间越来越大，页表所占用的内存空间也越来越多，页表尺寸与虚地址空间呈正比增长。为了减少内存空间开销，不得不使用多级页表，但也有些操作系统，如IBM AS/400、Power PC和UltraSPARC中均采用反置页表（Inverted Page Table，IPT），它的主要优点是只需为所有进程维护一张表，对于一个具有128 MB的计算机，若页面尺寸为4 KB，则反置页表仅占用128 KB内存。此表为内存中的每个页框建立一个IPT表项并按照块号进行排序，其表项包含：在此页框中页面的页号、页面所属进程的标识符和哈希链指针，用来完成逻辑地址到物理地址的转换，与此相适应，逻辑地址由进程标识符、页号和页内位移3个部分组成。如图4-10所示是反置页表及其地址转换的过程：需要访问内存地址时，地址转换机制用进程标识符与页号作为输入，由哈希函数先映射到哈希表，哈希表项存放的是指向IPT表项的指针，此指针要么就是指向匹配的IPT表项，否则，遍历哈希链直至找到进程标识符与页号均匹配的IPT表项，而此表项的序号就是页框号，通过拼接页内位移便可生成物理地址。若在反置页表中未能找到匹配的IPT页表项，说明此页不在内存，触发缺页异常，请求操作系统通过页表调入。

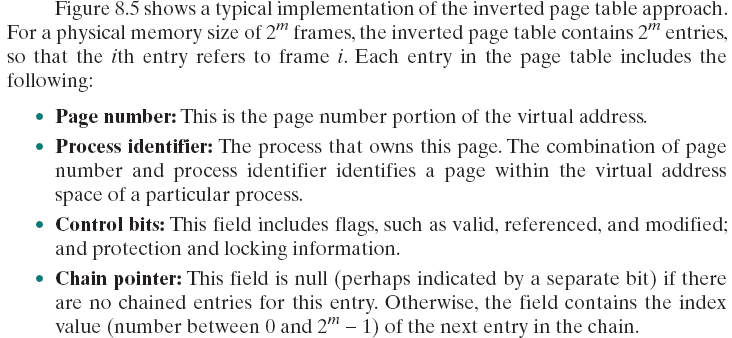
IPT表项中增加哈希链指针，是由于有多个页号经哈希函数转换后可能获得相同的哈希 值，所以，利用哈希链来处理这种冲突，在进行地址映射时，用哈希表定位后还可能要遍历哈希链逐一找出所需的页面。

图4-10 反置页表及其地址转换

为了使进程能共享内存中的同一页面，必须扩展IPT表的内容，使得每个表项可以记录多个进程。这样做虽然能解决问题，但却增加了复杂性。IPT能减少页表对内存的占用，然而，IPT仅包含调入内存的页面，不包含未调入的页面，仍需要为进程建立传统页表，不过此页表不再放在内存中，而是存放在磁盘上。当发生缺页异常时，把所需页面调入内存要多访问一次磁盘，速度会比较慢。





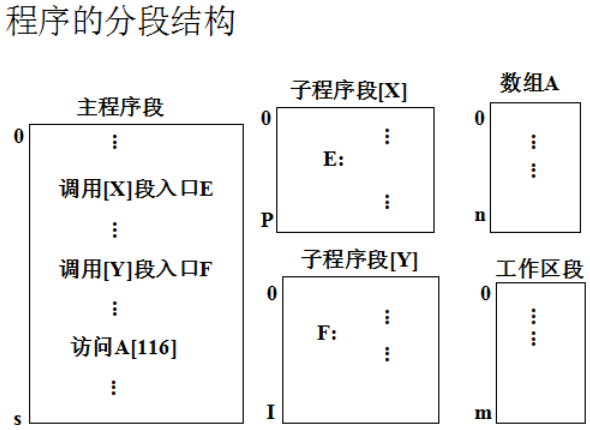
****

**反制页表中，不是为每一个虚拟页配置一个页表项而是每一个物理页框具有一个页表项，所以在内存大小固定的情况下，总的页表的大小也是固定的，无论支持的进程或者虚拟地址有多少。由于虚拟地址可能会重复，所以需要在页表项中添加一个指针形成一个哈希链**

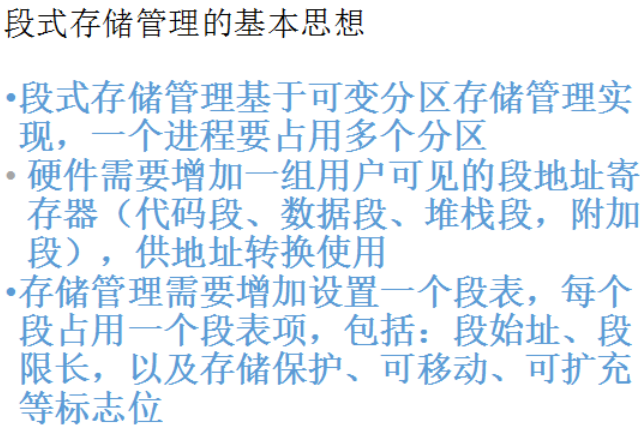
**//Operating Systems: Internals and Design Principles (8th edition), page 348**

**//反置页表的页表项**

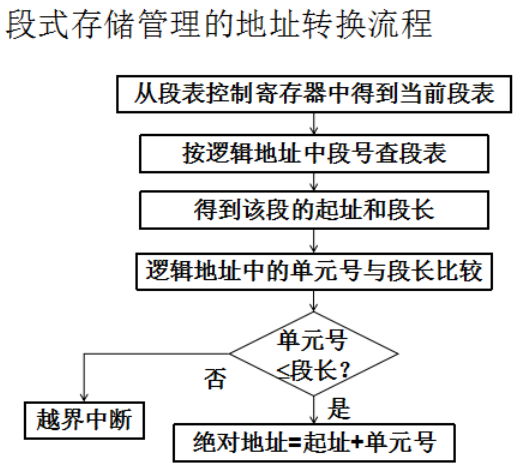
**4.4 分段存储管理**

****

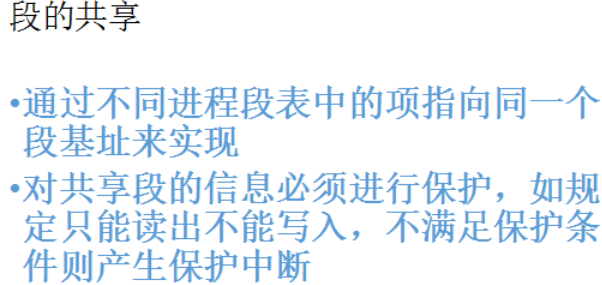
**这是一个程序的分段结构，特可以包括主程序段，若干个子程序段，若干个数据段等等。**

****

**下面来介绍一下段式存储管理的基本思想，段式存储管理是基于可变分区存储管理实现，一个进程有多个段，就有可能要占据多个分区。硬件是由一组用户可见的段地址及存器，至少有一个代码段，一个数据段，一个堆栈段，再加上一个附加段，你可以灵活使用，当然附加段有的计算机系统可能有不止一个寄存器附加段，这些东西都可以被我们的地址转换所借用，段式存储需要为每个进程增加设计一个页表，每个段占用一个段表项，它包括，段始址，段限长以及存储保护，段是不是可移动，是不是可扩充长度等等标志位。**

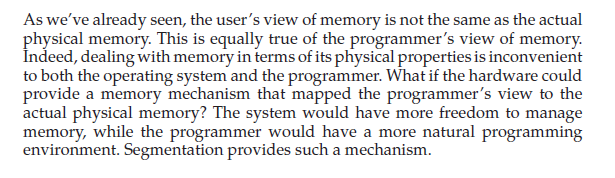
****

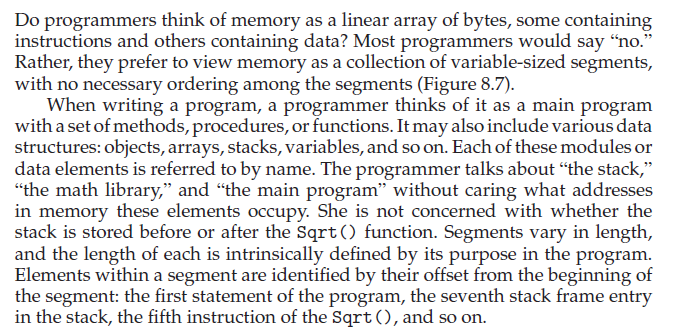
**在段式存储管理当中它的地址转换流程是这样的，我们从段表控制寄存器中得到当前段表，然后我们根据逻辑地址中的段号来查段表，得到该段始址和限长，然后我们把单元号和段长作比较，如果超过了段长就要发生越界中断，因为他是个非法的地址访问，否则的话我们就可以把我们的基址加单元号相加，形成我们的一个绝对地址。**

****

**我**

**//这部分内容来自操作系统慕课课件《3.12段式储**管理》。





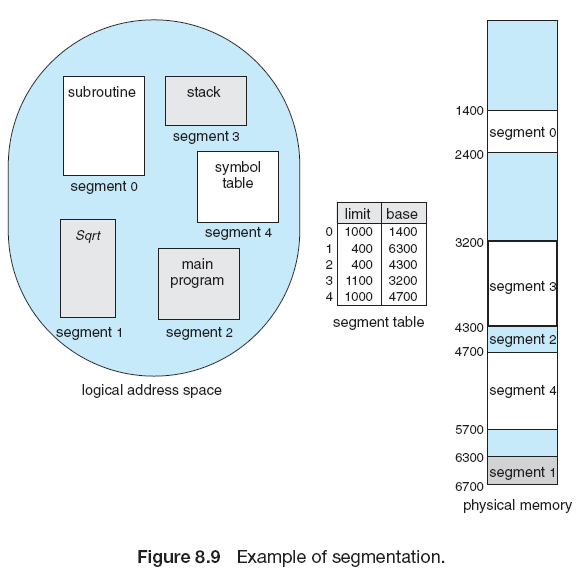
**就像之前提到的，用户对于内存的认知并不是和物理内存一样的，软件的编程人员也是一样，事实上，基于硬件属性对内存进行操作无论是对操作系统还是用户都是非常不方便的。所以，我们设想一种机制，可以让编程人员按照更自然的方式理解内存。分段就是这么一个机制**

**4.4.1 程序分段结构**

**对于编程人员来说，把内存当做线性的字节数组来处理是非常不方便的。更多的编程人员都更倾向于把内存当做不同段的集合，这些段的编址互相独立。在进行编程的时候，编程人员一般把这当作构建一个主程序，这个主程序包含了很多方法，过程和子函数，**

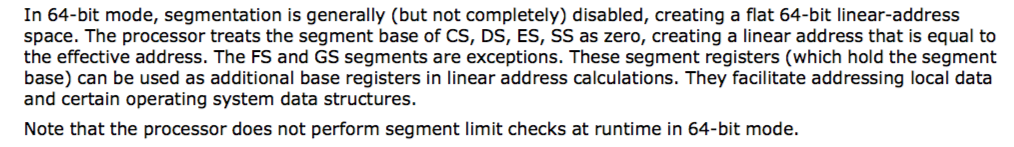
促使存储管理方式从固定分区到动态分区，从分区方式向分页方式发展的主要原因是要提高内存空间利用率。那么，分段存储管理引入主要目的是满足用户（编程人员）编程和使用上的要求，其他存储管理技术难以满足这些要求。在分页存储管理中，经编译链接处理得到一维地址结构的可装配目标模块，这是从0开始编址的单一连续逻辑地址空间，虽然可以把程序划分成页面，但页面与源程序并不存在逻辑关系，也就难以对源程序以模块为单位进行分配、共享和保护。事实上，程序更多是采用分段结构。**主程序包含的各种各样的方法，过程和子函数，以及数据格式：对象、数组、堆栈、变量等等，这些元素都依靠名字来引用。程序编写者只需要知道栈的名字，数学库的名字就可以找到对应的位置，而不用知道他在内存中具体的位置。**高级语言往往采用模块化程序设计方法。应用程序由若干程序段（模块）组成，例如，由主程序段（M）、子程序段(X)、数据段(D)和工作区段(W)组成，每段都从０开始编址，有各自的名字和长度，且实现不同功能。

在源程序中，可用符号形式（指出段名和入口）调用某段的功能，例如，主程序有：调用X段的入口E--call[x]︱<E>；把数据段D的G标号数据装入寄存器1--load 1,[D]︱<G>；等等。源程序经编译或汇编后，仍按照自身逻辑关系分为若干段，每段有一个段号，段之间的地址不一定连续，而段内地址是连续的。可见这是二维地址结构，模块化的程序被装入物理地址空间后，仍保持二维地址结构，这种地址结构需要编译程序的支持，但对程序员而言是透明的。

****

**//Operating System Concepts (9th Edition), page 367**

**//程序分段示意**

****

**在64位情况下，分段功能是基本上被禁用的，生成一个普通的64位地址空间。每一个段的基址都被表示为0，生成一个和有效地址相同的线性地址，之后借由每一个段维护的页表查询所在物理地址**

**4.4.2 分段存储管理基本原理**

分段存储管理把进程的逻辑地址空间分成多段，提供如下形式的二维逻辑地址：

段号 段内位移

在分页存储管理中，页的划分，即逻辑地址划分为页号和页内位移，是用户不可见的，连续的地址空间将根据页面的大小自动分页；而在分段存储管理中，地址结构是用户可见的，用户知道逻辑地址如何划分为段和段内位移，在设计程序时，段的最大长度由地址结构规定，程序中所允许的最多段数会受到限制。~~例如，PDP-11/45的段地址结构为：段号占3位，段内位移占13位，一个作业最多分为8段，各段长度可达8 KB。~~

分段存储管理的实现基于可变分区存储管理原理。可变分区以整个作业为单位来划分和连续存放，也就是说，作业在分区内是连续存放的，但独立作业之间不一定连续存放。而分段方法是以段为单位来划分和连续存放，为作业的各段分配一个连续内存空间，而各段之间不一定连续。在进行存储分配时，应为进入内存的作业建立段表，各段在内存中的情况可由段表来记录，它指出内存中各分段的段号、段起始地址和段长度。在撤销进程时，回收所占用的内存空间，并清除此进程的段表。

段表表项实际上起到基址/限长寄存器作用，进程运行时通过段表可将逻辑地址转换成物理地址，由于每个用户作业都有自己的段表，地址转换应按各自的段表进行。类似于分页存储管理，也设置一个硬件--段表基址寄存器，用来存放当前占用处理器的作业段表的起始地址和长度。~~分段存储管理的地址转换和存储保护流程如图 4-11所示，将段控制寄存器中的段表长度与逻辑地址中的段号进行比较，若段号超过段表长度则触发越界中断，再利用段表项中的段长与逻辑地址中的段内位移进行比较，检查是否产生越界中断。~~

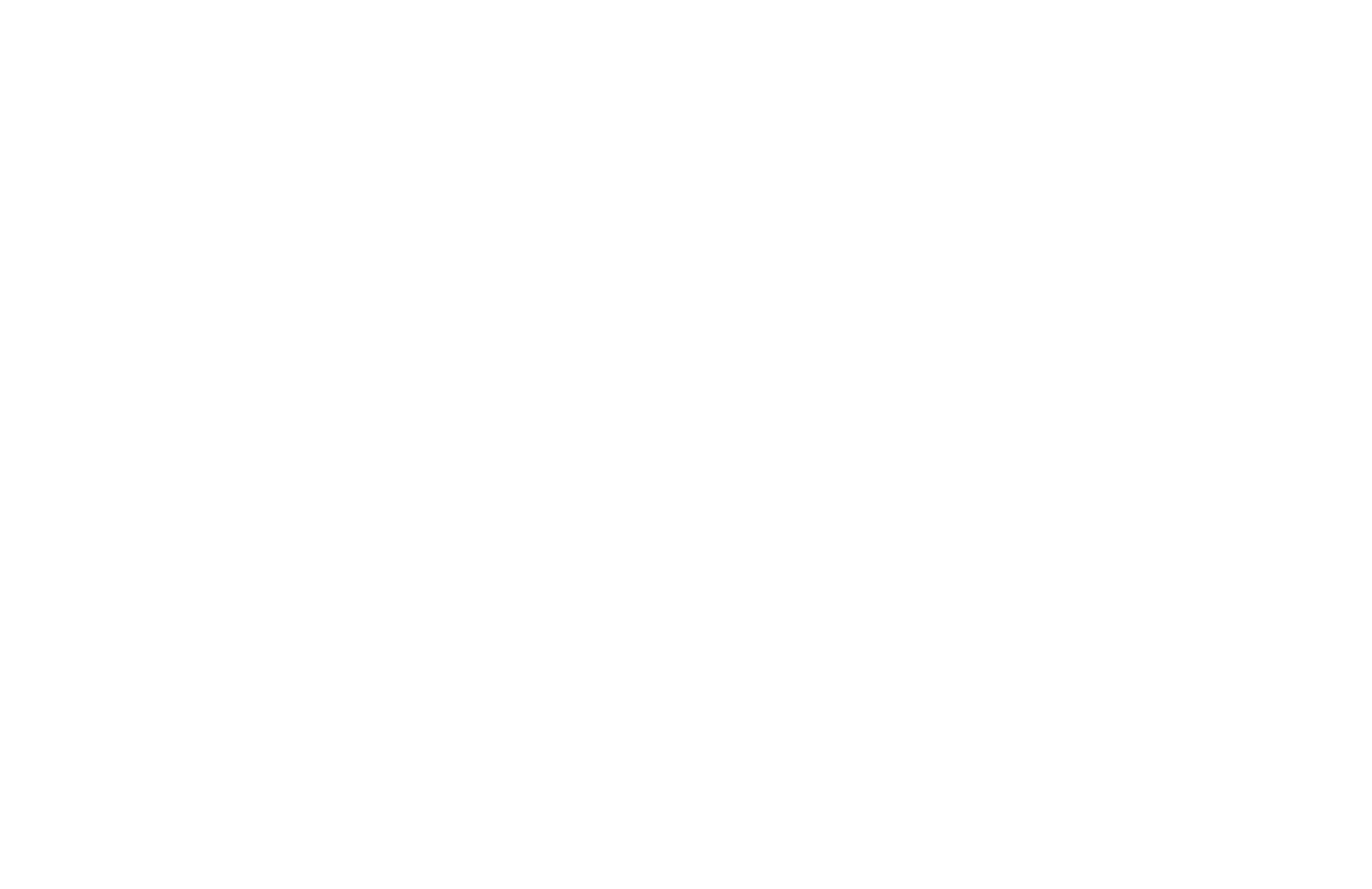
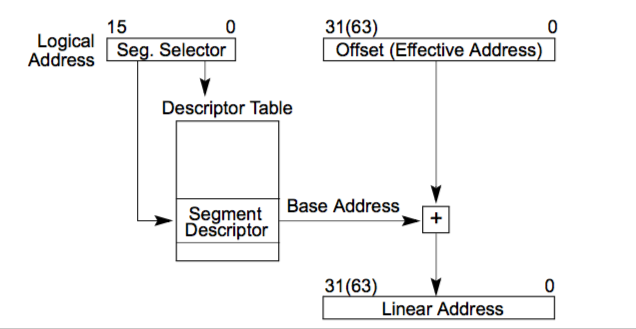


图4-11 分段存储管理的地址转换和存储保护

****

**上图为intel ia-32架构中对于段式存储的地址变化方式，由于处理器运行模式的不同，段地址的长度分为32位和64位两种情况，但是段号的长度都是一致的。可以从中看出段式存储地址变化的具体流程**

1. **从段表控制寄存器中得到当前段表。**
2. **根据逻辑地址中的段号来查段表，得到该段始址和限长。**
3. **把单元号和段长作比较**
   1. **超过了段长就要发生越界中断，因为他是个非法的地址访问，**
   2. **否则把基址和单元号相加，形成程序所请求的绝对地址。**

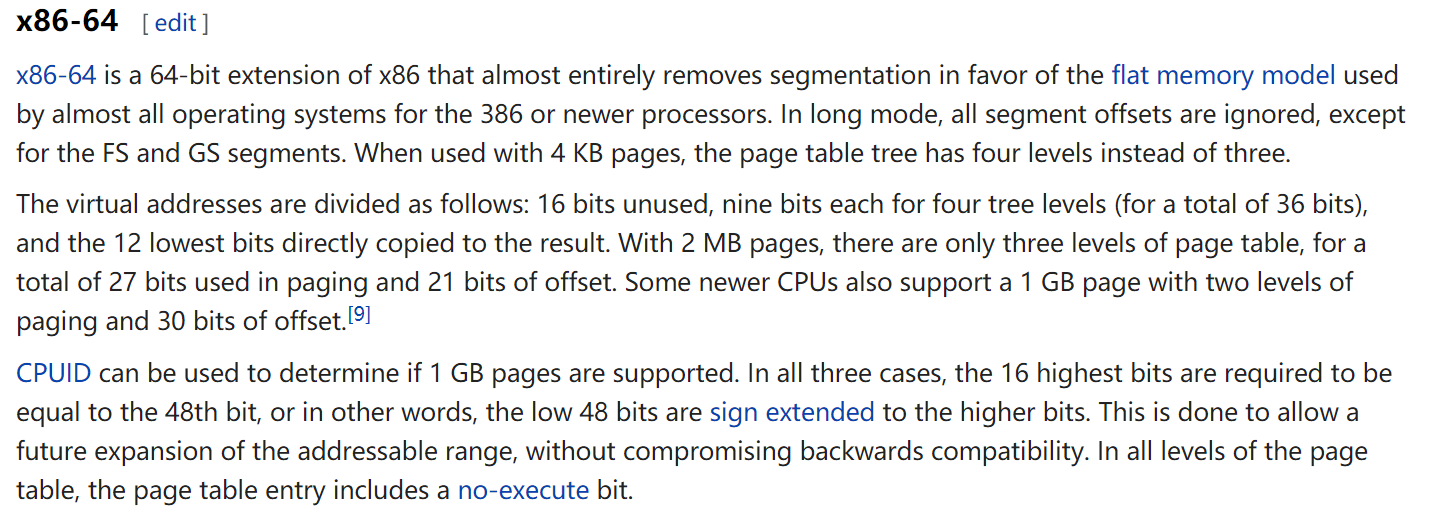
**4.4.3 分段存储管理共享和保护**

对于分段存储管理来说，由于它是基于逻辑含义来划分的信息单位，逻辑上相关的程序和数据不会被分割，而安排在内存中一个连续分段中，故地址转换比较简单。与分页共享类似，若多个进程段表中的某段指向内存相同地址，内存中以该处为起始地址的某段便可被共享。为了便于共享，系统中常建立一张共享段表，用来记录所有共享段，内容有：段名、共享计数、段长、段首址、保护位等。当多个进程共享同一段时，这些进程的段表中的表项指向共享段表中的同一表目。保护位用来对共享段实施保护，如禁写、禁修改等。

**4.4.4 分段和分页比较**

分段是信息的逻辑单位，由源程序的逻辑结构及含义所决定，是用户可见的，段长由用户根据需要来确定，段起始地址可从任何内存地址开始。在分段方式中，源程序 （段号、段内位移）经链接装配后仍保持二维（地址）结构，引入目的是满足用户模块化程序设计的需要。

分页是信息的物理单位，与源程序的逻辑结构无关，是用户不可见的，页长由系统 （硬件）确定，页面只能从页大小的整数倍地址开始。在分页方式中，源程序 （页号、页内位移）经链接装配后变成一维（地址）结构，引入目的是实现离散分配并提高内存利用率 。

****

**//分段存储在x86-64架构上的使用渐渐消失**

**//https://en.wikipedia.org/wiki/Memory\_management\_unit**

**4.5 虚拟存储管理**

**4.5.1 虚拟存储器概念**

前面所介绍的存储管理称为实存管理，必须为进程分配足够内存空间，装入其全部信息，否则就无法运行。把进程全部信息装入内存后，实际上并非同时使用，有些部分运行一遍，有些部分暂时不用，或某种条件下才使用，让进程全部信息驻留于内存是对宝贵内存资源的极大浪费，会降低内存利用率。于是，提出新的想法：能否把物理内存扩大到大容量磁盘上，把磁盘空间当作内存的一部分，进程的程序和数据通常部分放在内存中，部分放在磁盘上。程序运行时，它执行的指令或访问的数据~~究竟在哪儿则~~的存储位置则由存储管理负责判断，并针对情况采取相应步骤，这就是虚存储管理技术的基本思路。当程序执行的指令或访问的数据在内存中时，可顺利执行；如果在磁盘上，为了继续执行，由系统自动将这部分信息从磁盘装入，这称“部分装入”；如此刻没有足够的空闲内存空间，便把内存中暂时不用的信息移至磁盘，这称“部分替换”。如果“部分装入、部分替换” 能够实现，那么，当内存空间小于进程的需要量时，进程也能运行；更进一步，当多个进程的总长超出内存总容量时，也可将进程全部装入内存，实现多道程序运行。这样，不仅能充分地利用内存空间，而且用户编程时不必考虑物理内存的实际容量，允许用户的逻辑地址空间大于内存物理地址空间，对于用户而言，好像计算机系统具有一个容量硕大的“内存”，称其为“虚拟内存或虚拟存储器”（virtual memory）（Fotheringham，1961年）。

虚拟存储器可定义如下：在具有层次结构存储器的计算机系统中，自动实现部分装入和部分替换功能，能从逻辑上为用户提供一个比物理内存容量大得多的、可寻址的“内存储器”。实际上，虚存对用户隐蔽可用物理内存的容量和操作细节，虚存的容量与物理内存大小无关，而受限于计算机地址结构和磁盘容量，如Intel x86地址线是32位，则程序可寻址范围是4 GB，Linux 系统为应用进程提供一个4 GB逻辑内存。

如图4-12所示是虚存概念图，其中，逻辑地址是从进程角度所看到的逻辑内存单元，而物理地址是从处理器角度所看到的物理内存单元，虚拟地址可以说是将逻辑地址映射到物理地址的一种手段。逻辑地址空间是程序员的编程空间，物理地址空间是程序的执行空间，虚拟地址空间等同于物理内存加部分硬盘区域所组成的存储空间。实现虚存的关键问题是：把程序编程和访问地址与内存物理地址分隔开来，用户无需知道内存工作方式，只需在逻辑地址空间中编程，大大方便了程序编写工作，系统负责内存分配和地址映射，把逻辑地址最终转换为物理地址。

物理地址

物理地址空间

虚拟地址

逻辑地址空间

处理器

存储管理部件

内存

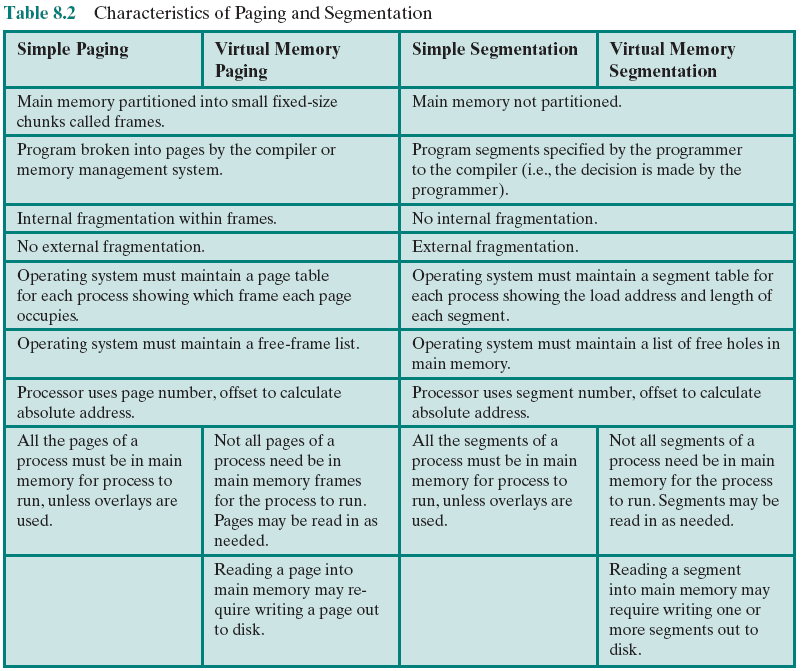
外存

图4-12 虚拟存储器概念

下面讨论进程信息不全部装入内存时能否正确运行。早在1968年，P.Denning、Knuth及Huck等人研究程序执行时的局部性原理，发现程序和数据的访问都有聚集成群的倾向，程序对内存空间的90%的访问局限于内存空间的10%的区域中。某存储单元被访问，该单元及其相邻存储单元很可能被访问（空间局部性），或者最近访问过的存储单元很快又被访问（时间局部性）。1.顺序局部性：程序中只有少量分支和过程调用，大都是顺序执行的指令；2.时间局部性：程序中存在迭代和循环结构，处理器最近访问的单元在不久将来再次被访问或多次循环访问；3.空间局部性：程序对一定范围的数组、或数据堆栈进行集中访问，对它们的连续引用是对位置相邻的数据项的操作；种种情况说明，程序具有局部性，进程运行时没有必要把全部信息调入内存，只装入一部分的假设是合理的，此时只要调度得当，不仅可正确运行，而且能在内存中放置更多进程，充分利用处理器和内存空间。虚拟存储器是基于程序局部性原理的一种假想的而非物理存在的存储器，其主要任务是：基于程序的局部性特点，当进程使用某部分地址空间时，保证将相应部分加载至内存中。

虚存管理与对换技术虽说都是在内存和磁盘之间交换信息，但却存在很大区别。对换技术以进程为单位，当其所需内存空间大于当前系统的拥有量时，进程无法被对换进内存工作；而虚存管理以页或段为单位，即使进程所需内存空间大于当前系统拥有的内存总量，仍然能正常运行，因为系统可将其他进程的一部分页换出至磁盘。虚存的实质是：实现内存资源的时分复用，每次把应用程序的一部分装入内存运行，运行结束后被换出，再装入应用程序的另一部分到内存中运行。通过“部分装入，部分替换”置换功能，便实现了应用程序的各个部分，分时地进入内存。

虚存思想早在20世纪60年代初就在英国Atlas计算机上出现，到20世纪60年代中期，较完整的虚拟存储器在分时系统MULTICS和IBM系列操作系统中得到实现，20世纪70年代开始推广应用，渐渐为广大计算机研制者和用户接受。这项技术不仅用于大型计算机，也逐步被用到微型计算机系统中。为了实现虚存，必须解决好以下问题：内存与外存（磁盘）统一管理问题 、逻辑地址和物理地址转换问题、部分装入和部分替换问题。实现虚存要付出一定开销，其中包括：管理地址转换的各种数据结构所用的存储开销、执行地址转换指令所花费的时间开销和内存与外存交换页或段的I/O开销等。目前，虚存管理主要采用以下技术实现：请求分页和请求段页式虚存储管理。

****

**//Operating Systems: Internals and Design Principles (8th edition), page 343**

**4.5.2 请求分页虚拟存储管理**

**1. 请求分页虚存管理的硬件支撑**

操作系统的存储管理依靠低层硬件支撑来完成任务，此硬件称为存储管理部件 （Memory Management Unit，MMU），它提供地址转换和存储保护功能，并支持虚存管理和多任务管理。MMU由一组集成电路芯片组成，**接受CPU提供的**逻辑地址作为输入，物理地址作为输出，直接送达总线，对内存单元进行寻址，其位置和功能如图4-13（A）所示，其内部执行过程如图4-13(B)所示。主要功能列举如下。

（1）管理硬件页表基址寄存器。每当发生进程上下文切换时，系统负责把运行进程的页表基地址装入硬件页表基址寄存器，此页表便成为活动页表，MMU 只对由硬件页表基址寄存器所指出的活动页表进行操作。

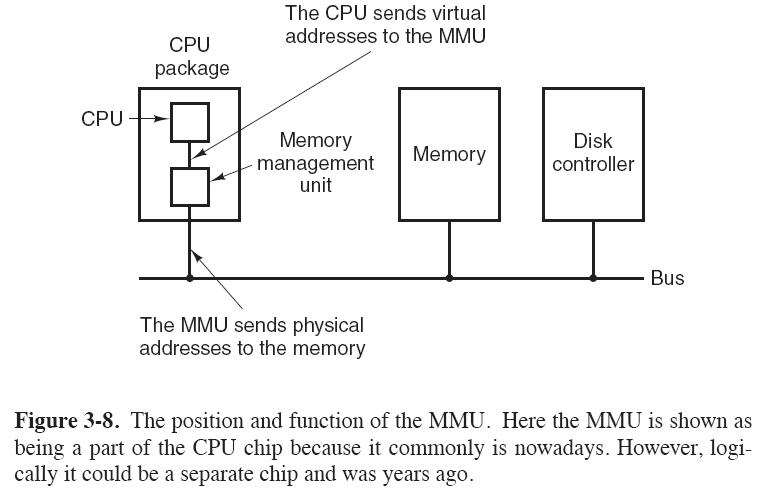
（2）分解逻辑地址。把逻辑地址分解为页号和页内位移，以便进行地址转换。

（3）管理快表。对TLB进行管理，一是直接查找快表，找到相应页框后去拼接物理地址；二是执行TLB的基本操作：装入表目和清除表目，每次发生快表查找不命中的情况后，待缺页异常处理结束，把相应页面号和页框号装入。此外，每次写硬件页表基址寄存器时，负责清除快表项，将TLB清空。

（4）访问页表。当TLB不命中时，根据页表基址寄存器直接访问进程页表，若所需页面已装入，则可访问内存，同时，把此页面信息装入TLB。

（5）发出异常。当查出页表中有页失效位或页面访问越界位时，发出缺页异常或越界中断，并将控制权交给存储管理程序。

（6）管理特征位。负责设置和检查页表中的引用位、修改位、有效位和保护权限位等各个特征位。

****

**//Operating System Concepts (9th Edition), page195**

CPU送入逻辑地址(8196)

0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0

页号 页框号 在内存否

0 010 1

1 001 1

2 110 1

3 000 1

4 100 1

5 011 1

6 000 0

7 000 0

8 101 1

9 000 0

…

逻辑地址

物理地址

CPU

MMU

内存

存

1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

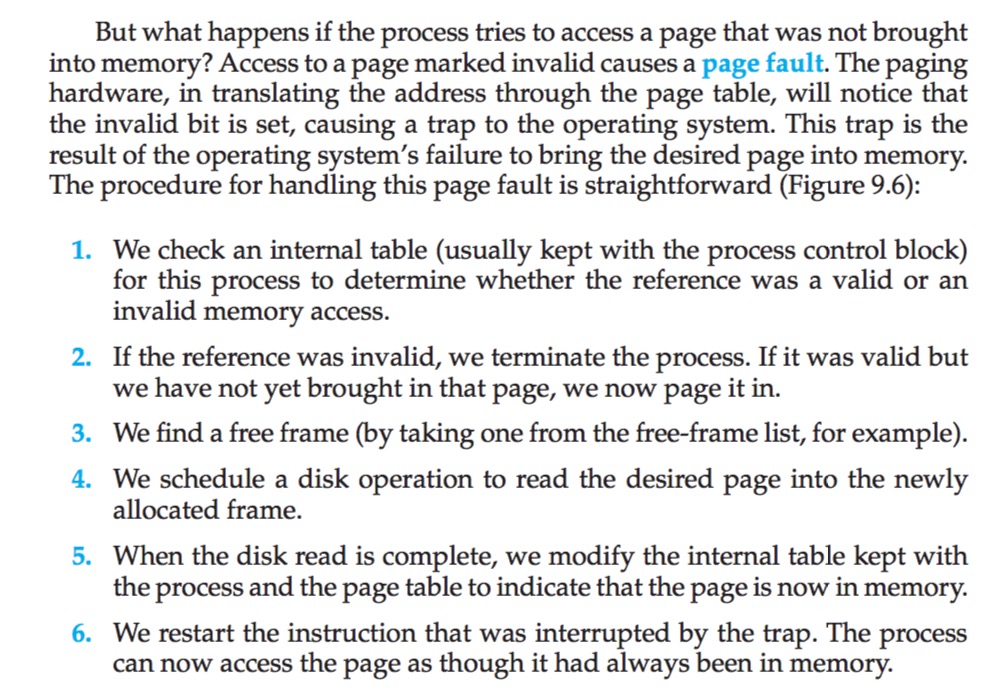
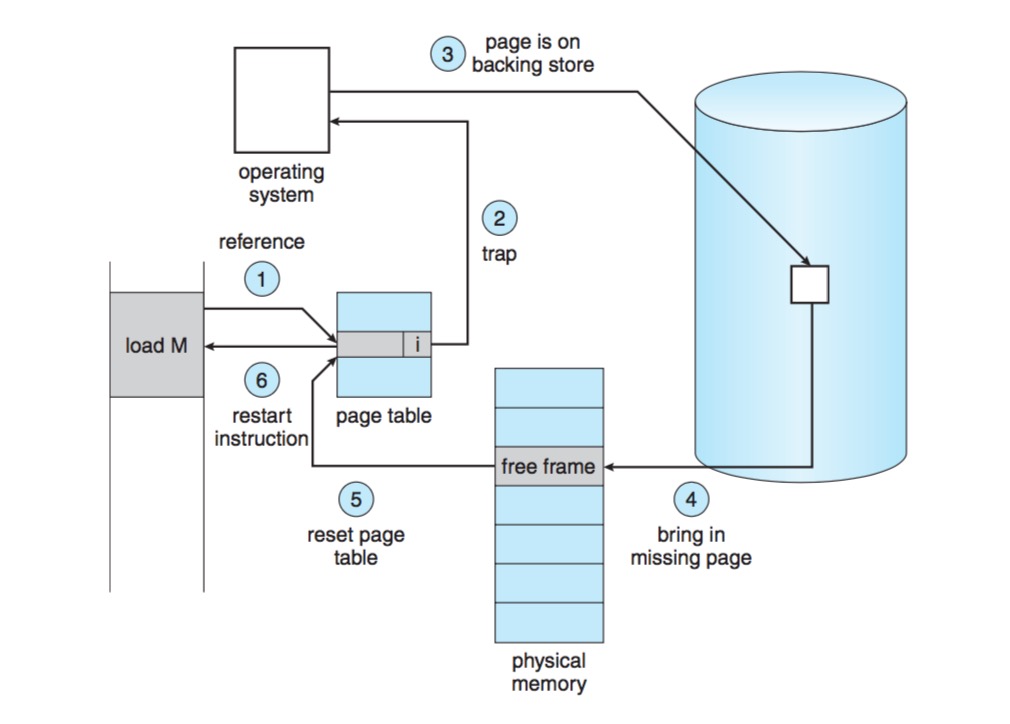
MMU送出物理地址

(A) MMU的位置和功能

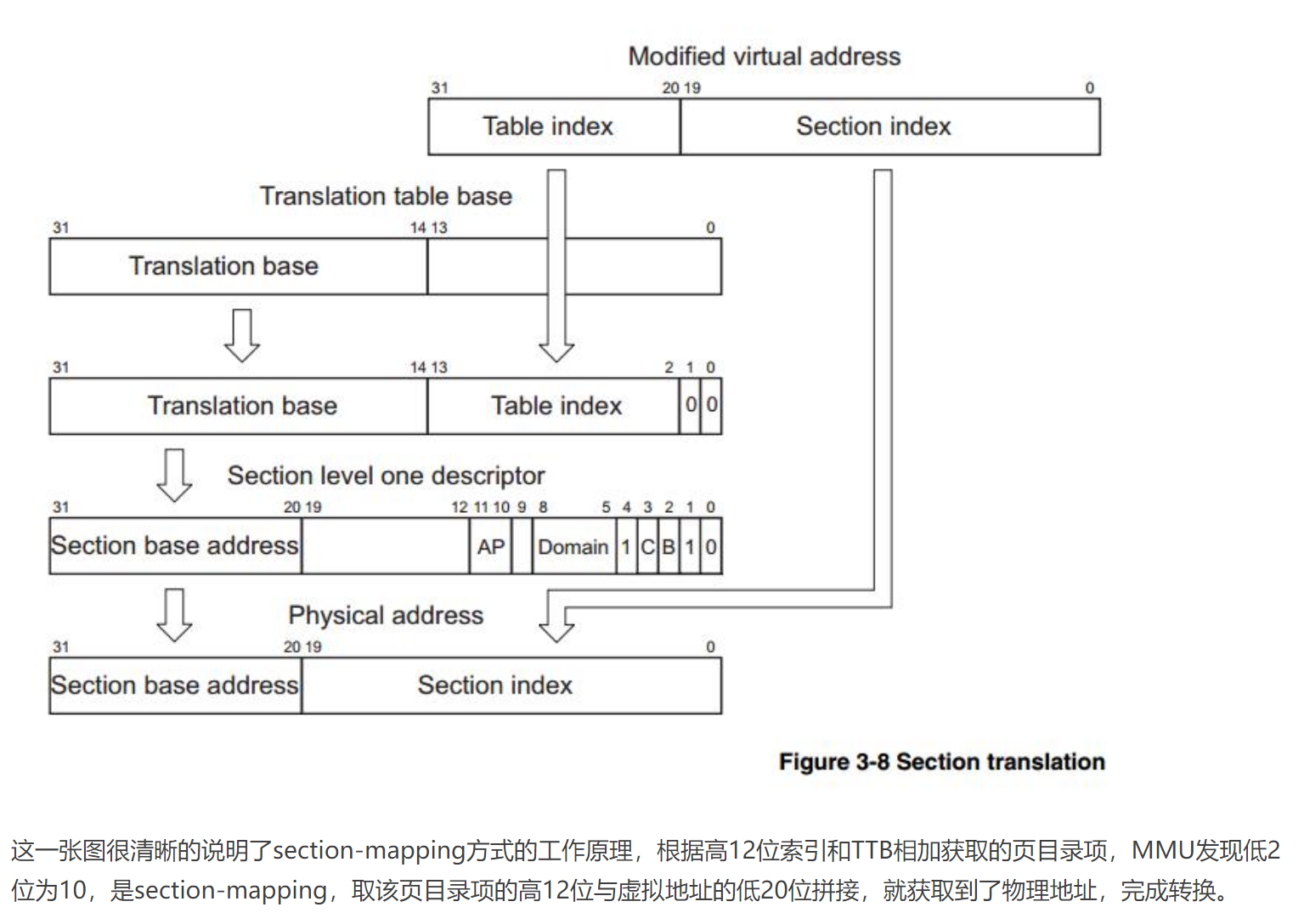
(B)16个4KB页面情况下MMU内部操作

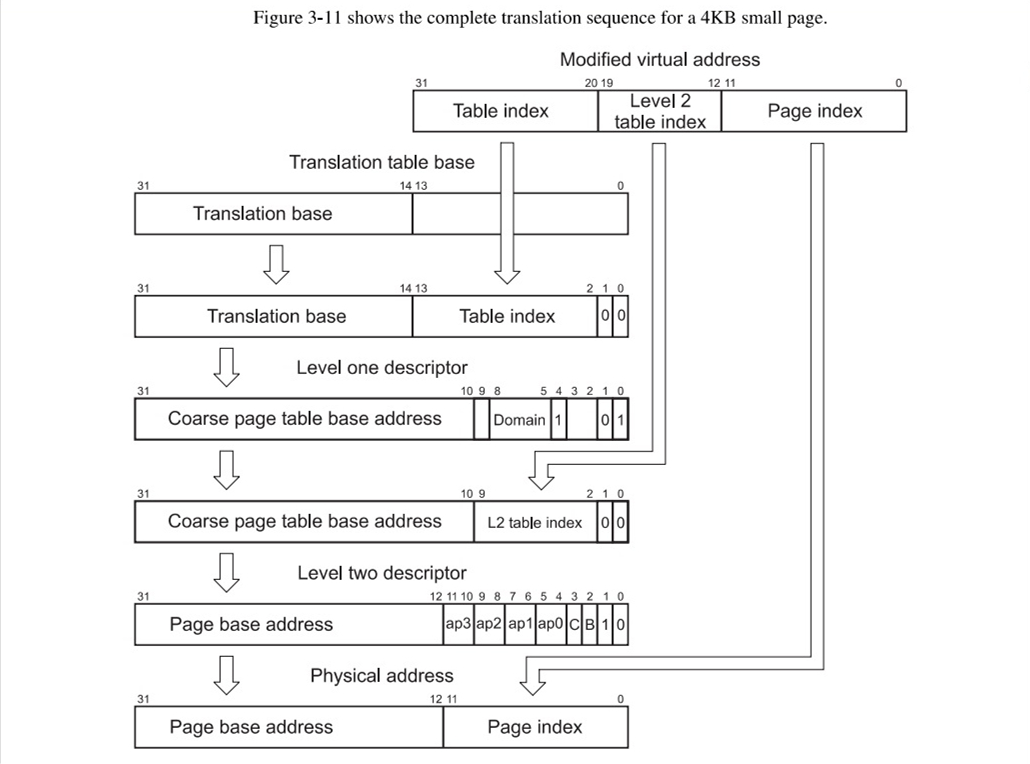
图4-13 存储管理部件MMU

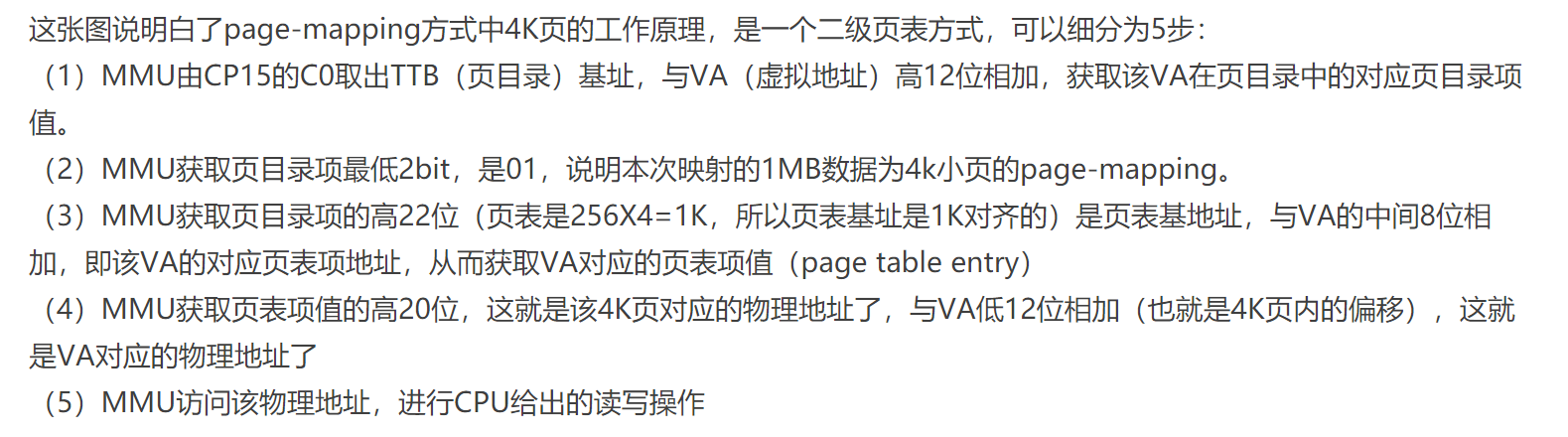
下面考察MMU的工作过程。在图4-12（B）中，给出一个逻辑地址8196（二进制表示为0010000000000100），MMU进行映射，输入的16位逻辑地址被解释为4位页号和12位页内位移。用页号作为索引，找出虚页所对应的页框号，如果“在内存否”位为1，表明此页在内存，把页框号复制到输出寄存器的高3位，再加上逻辑地址中的12位页内位移，生成15位物理地址（二进制表示为110000000000100），并把它送到内存总线；如果“在内存否”位为0，则将触发缺页异常，陷入操作系统进行调页处理。



**//arm架构下的mmu存储系统的概要，并且arm架构中有两种处理虚拟地址的方式，一种是section类型，一种是paging类型，两种的处理方式分别如下**

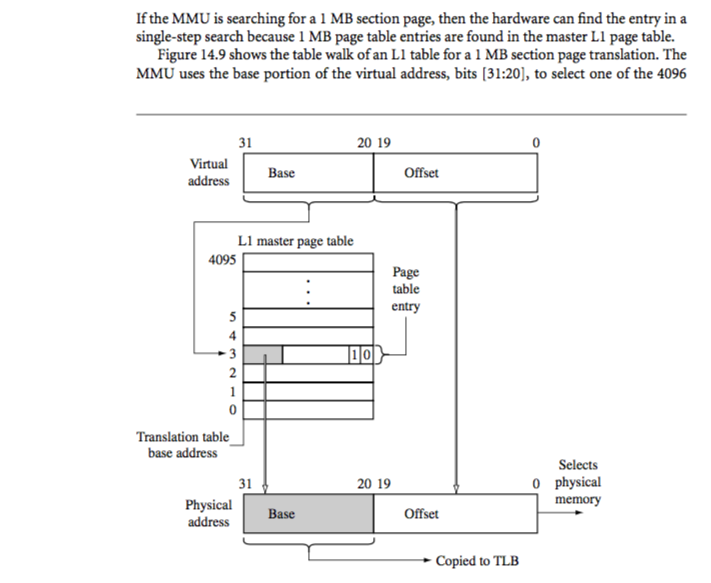
****



****

**//来自博客http://blog.csdn.net/skyflying2012/article/details/51044705###;**

**//mmu的工作过程的介绍**



在实际的硬件操作中

**2. 请求分页虚存管理的基本原理**

请求分页虚存管理是将进程信息副本存放在外存中，当它被调度投入运行时，程序和数据没有全部装进内存，仅装入当前使用页面，进程执行过程中访问到不在内存的页面时，再由系统自动调入。用得较多的分页虚存管理是请求分页（demand paging），当需要执行某条指令或使用某个数据而发现它们不在内存时，产生缺页异常（page fault），系统从磁盘中把此指令或数据所在的页面装入，这样做能够保证用不到的面页不会被装入内存。

这个部分再具体提页表项的内容是否在结构上有些分散

~~那么，如何发现页面不在内存中呢？页面不在内存中该如何处理？采用的办法是：扩充页表项内容，增加驻留标志位，以及用来跟踪页面使用情况和对页面实施保护及淘汰等所需的各种控制位，通常，页表项至少包含以下信息。~~

页号 …… 驻留 标志位 引用位 修改位 保护位 内存块号

~~“引用位”是在页面被引用即无论是读或写时设置，其值帮助系统进行页面淘汰；“修改位”是当页面被修改后，硬件自动置位，一旦修改位被设置，此页被调出内存时必须先被写回磁盘；“保护位”则限定页面允许的访问权限 （如读 、写 、执行等）；“内存块号”是页面对应的页框号，用于地址转换。~~

~~“驻留标志位”用来指出页面是否已装入内存。当访问一个页面时，如果某页的驻留标志位为1，表示此页已在内存，可被正常访问；如果某页的驻留标志位为0，不能立即访问，产生缺页异常，存储管理根据磁盘地址将这个页面调入内存，然后，重新启动相应指令。那么，如何查找页面对应的磁盘地址呢？ 磁盘的物理地址由磁盘机号、柱面号、磁头号和扇区号所组成，通常规定扇区长度等于页面长度，页面与磁盘物理地址的对应表称外页表，由操作系统管理。进程启动运行前系统为其建立外页表，并把进程页面装入外存，外页表也按进程页号顺序排列，所以，查找外页表就找到页面对应的磁盘存放地址。为了节省内存空间，外页表可存放在磁盘中，当发生缺页异常需要查用时才被调入。~~

缺页异常是由于发现当前访问页面不在内存时由硬件所产生的一种特殊中断信号，是在指令执行期间产生并由系统处理的。而且，一条指令可能涉及多个页面，例如，指令本身跨页、指令处理的数据跨页，完全有可能在执行一条指令的过程中发生多次缺页异常。

由上可以总结出，在请求分页虚存管理中，页表中存放的是把逻辑地址转换成物理地址时硬件需要的信息，其**组成以及具体的**作用有：（1）获得页框号以实现虚实地址转换；（2）~~设置~~访问控制位，对页面信息进行保护；（3）设置各种标志位来实现相应的控制功能（如缺页标志、脏页标志、访问标志、锁定标志和淘汰标志等）。

驻留标志位：标识页面是否在内存中

引用位。。。

修改位。。。

下面讨论使用快表但是页表存放在内存的情况下，请求分页虚实地址转换过程：当进程被调度到CPU上运行时，操作系统自动把此进程PCB中的页表起始地址装入硬件页表基址寄存器，此后，进程开始运行并访问某个逻辑地址，MMU 开始工作，它将完成图4-14虚线框内的任务，可看到地址转换过程如下：

（1）MMU接收CPU传送来的逻辑地址并自动按页面大小把它从某位起分解成两部分：页号和页内位移；

（2）以页号为索引快速搜索快表TLB；

（3）如果命中，立即送出页框号，并与页内位移拼接成物理地址，然后，进行访问权限检查，如获通过，进程就可以访问物理地址；

（4）如果不命中，由硬件以页号为索引搜索页表，页表基址由硬件页表基址寄存器指出；

（5）如果页表被命中，说明访问页面已在内存中，可送出页框号，并与页内位移拼接成物理地址，然后，进行访问权限检查，如获通过，进程就可以访问物理地址，同时要把这个页面和页框信息装入快表TLB，以备再次访问；

（6）如果发现页表中的对应页面失效，MMU 发出缺页异常，请求操作系统进行处理，MMU工作到此结束。

外存

②查快表

⑦发缺页中断

缺页中断处理

进程切换时装入

⑤页表命中

③命中

④不命中

cpu

逻辑空间

地址

①分解地址

④查页表

运行进程页表基址

逻辑地址

页号 页内地址

内存(系统区)运行进程页表

⑧调页

⑨调入、改表

⑽访问

MMU

物理空间

运行进

程映象

内存(用户区)

⑥装入快表

物理地址

页框 页内地址

快表

图4-14 请求分页虚存地址转换过程

****

MMU发现缺页并发出缺页异常，存储管理接收控制，进行缺页异常处理的过程如下：步骤1：挂起请求调页的进程。步骤2：根据页号搜索外页表，找到存放此页的磁盘物理地址。步骤3：查看内存是否有空闲页框，如有则分配一个，转步骤6。步骤4：如果内存中无空闲页框，按照替换算法选择淘汰页面，检查其是否被写过或修改过，若否则转步骤6；若是则转下一步。步骤5：将修改过的淘汰页内容写回磁盘原先位置。步骤6：进行调页，把页面装入内存所分配的页框中，同时修改系统页框表和进程页表项。步骤7：返回进程断点，重新启动被中断指令。

**当进程访问内存中没有的页面时，就会引发缺页中断，页面转换硬件在页表中转换地址时可用位被置零,操作系统就会被通知产生了异常，这个异常引发了操作系统对页面缺失的处理：**

**1检查内置的表格，一般是进程自带的PCB中的数据，来决定页面是否在内存中**

**2.如果内存中没有对应的页面，这个页面真实存在只不过没有被放到内存中，那么就把他放到内存中。**

**3.找到一个空闲页框**

**4.准备一系列磁盘操作来把目标页放到分配的页框中**

**5.当磁盘读取完成之后，要对内置表格进行修改，把对应的页标记为在内存中**

**6.把当时暂停的指令重新启动，程序就可以访问该页了**

在分页虚存系统中，由于页面在需要时是根据进程请求装入内存的，因此，称为请求分页虚存管理。优点是：进程的程序和数据可按页分散存放在内存中，既有利于提高内存利用率，又有利于多道程序运行。缺点是：要有一定硬件支持，要进行缺页异常处理，机器成本增加，系统开销加大，此外，页内会出现碎片，如果页面较大，则内存仍有一定损失。

**3. 交换区**

在虚存系统中，内存只能为每个进程保存部分页面，当要访问的页面不在内存时，就必须将其装入。显然，新装入内存的页面将替换掉一个原有页面，而这个被替换的页面有可能在将来又要再次换入内存，不能简单地将它丢弃。于是，系统将该页面内容保存在外存（磁盘）上。操作系统在磁盘上定义一个交换区用来保存临时换出的页面，交换区由磁盘上的一个或多个磁盘分区组成，系统在安装配置时就已保留好若干磁盘分区作为交换区。交换区是物理内存的扩展，它和内存一起组成虚拟存储空间，因此，不允许文件系统使用交换区所占用的磁盘分区。当系统启动时，该交换分区为空，系统记录它的起始位置和大小。简单做法是：当一个进程启动时，留出与这个进程一样大的交换分区，剩余大小为总空间量减去这个交换分区大小。进程运行结束后，会释放其磁盘上的交换区。

与每个进程对应的是其交换区的磁盘地址，即进程映像所保存的位置。这一信息记录在进程的外页表中。写回一个页面时，计算写回地址的过程很简单，将进程虚存空间中的页号，经简单计算再加上交换区的开始地址。但在进程启动前必须初始化交换区，可将整个进程映像复制到交换区，以便随时可将所需页面装入。

但这种简单模式有一个问题：进程在启动后可能增大，尽管程序正文通常是固定的，但数据有时增长，堆栈会增长。这样，最好为正文、数据和堆栈分别保留交换区，并且为这些交换在磁盘上多留几块。另一个极端做法是事先什么也不分配，在页面换出时为其分配磁盘空间，并在换入时回收磁盘空间，这样内存中的进程不必固定于任何交换空间位置。其缺点是内存中每个页面都要记录相应变动的磁盘地址，管理较为复杂。

交换区管理重点是维护交换区映射表，该表用来记录所有被换出内存的页面在交换区中的位置，以便在需要时再次换入内存。当页面第二次被换出内存时，仅当页面修改过（脏页）才写入交换区，否则，因已有副本就直接将它丢弃。

**4. 页面装入策略和清除策略**

页面装入策略决定何时把页面装入内存，有两种基本策略：请页式（demand paging）和 预调式（prepaging）。

请页式是当产生缺页异常时调入页面。该策略优点：确保只有被访问页面才会调入，节省内存空间；缺点：缺页异常处理次数多，调页系统开销大；由于每次仅调用一页，磁盘I/O操作次数猛增。预调式在使用页面前预先调入内存。操作系统依据某种算法，动态预测进程最可能要访问的页面。尽量做到进程要访问的页面已经调入内存，且每次调入若干页面，而不是仅调入一页。该策略优点：减少磁盘I/O启动次数，节省寻道和搜索时间。缺点：如果所调页面大多未被使用，则效率就很低，可见，预调页要建立在可靠预测的基础之上。

页面清除策略与装入策略相对应，考虑何时把修改过的页面写回外存，常用方法有：

请页式和预约式。请页式清除是仅当一页被选中进行替换且其内容被修改过，才把它写回磁盘。预约式清除是对于所有更改过的页面，在需要替换之前把它们都写回磁盘，可成批进行。对于预约式清除，写出的页仍然在内存中，直到页替换算法选中此页从内存中移出。但如若刚刚写回很多页面，在它们被替换之前，其中大部分又被修改过，那么，预约式清除就毫无意义。对于请页式清除，写出一页是在读进新页之前进行的，它要成对操作，虽然仅需写回一页，但进程不得不等待两次 I/O 操作完成，可能会降低系统性能。

**5. 页面分配策略**

请求分页虚存管理排除内存实际容量的约束，使更多进程能同时多道运行，从而，提高系统效率，但是，缺页处理要付出很大代价，由于页面调入和调出要增加 I/O 操作负担，因此，应尽可能地减少缺页次数。那么，究竟如何为进程分配页框呢？当出现缺页时，页面替换算法的作用范围应局限于此进程的页面，还是内存中所有进程的页面？这两个问题均涉及进程驻留页面的管理。

如果在进程生命周期中，保持页框数固定不变，称其为固定分配，在创建进程时，根据进程类型和系统规则决定页框数，只要有一个缺页异常产生，进程就会有一页被替换；如果在进程生命周期中，所分得的页框数可变，称为可变分配，当进程运行的某一阶段缺页率较高，说明目前局部性较差，系统可多分页框降低缺页率，反之，说明局部性较好，可减少所配给的页框数。

页面替换可采用两种策略：局部替换和全局替换。如果页面替换算法的作用范围是整个系统，不考虑进程属主，称全局页面替换算法；如果作用范围局限于进程自身，称局部页面替换算法。要为每个进程维护一组页面，称其为工作集，其大小随进程的执行而变化，应自动地排除不再在工作集中的页面。当工作集大小会在进程运行期间发生较大变化时，全局算法比局部算法好，但是系统必须不断地确定应给每个进程分配的页框数，这是困难的任务。

固定分配往往和局部替换策略配合使用，进程运行期间分得的页框数不再改变，如果发生缺页异常，只能从进程在内存的页面中选出一页替换，以保证进程页框总数不变。该策略难点在于：应给进程分配多少页框才合适？少了，缺页异常率高；多了，内存中能同时运行的进程数减少，造成处理器和其他设备空闲。常用的固定分配算法有：平均分配、比例分配、优先权分配等。

可变分配往往和全局替换策略配合使用，为系统中的进程分配一定数目的页框，操作系统保留若干空闲页框。进程发生缺页时，从系统空闲页框中为其分配，把所缺页面调入此页框，于是产生缺页的进程的内存空间会逐渐增大，有助于减少系统缺页异常总次数。当系统拥有的空闲页框几近耗尽时，要从内存中选择页面淘汰，可以是内存中任一进程的页面，这样又会使某进程页框数减少，缺页异常率上升。这种方法的难点在于选择哪个页面作替换，应用某一种淘汰策略选页时，并未确定哪个进程会失去页面。如果选择某个进程，此进程工作集的缩小会严重影响其运行，那么，这个选择就不是最佳的。

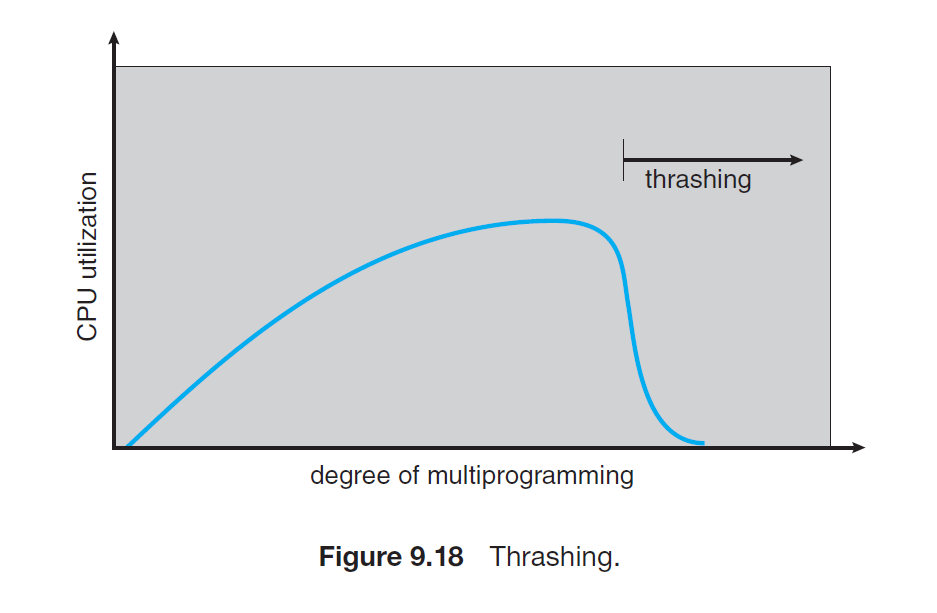
可变分配配合局部替换可克服可变分配配合全局替换时存在的缺点，实现要点如下：（1）新进程装入内存时，根据应用类型、程序要求，分配给一定数目页框，可用请页式或预调式完成分配；（2）当产生缺页异常时，从进程的页面中选择一页淘汰；（3）不时重新评价进程的缺页率，增加或减少分配给进程的页框数，以改善系统总体性能。

**6. 缺页中断率**

虚存能够给用户提供一个容量很大的存储空间，但当内存空间已满而又要装入新页时，必须按照预定算法把已在内存中的页面写回，这项工作称为页面替换。用来确定被淘汰页的算法称为淘汰算法。如果选用不合适的算法，会出现这样的现象：刚被淘汰的页面立即又要调用，而调入不久随即被淘汰，淘汰不久再被调入，如此反复，使得整个系统的页面调度非常频繁，以致大部分时间都花费在来回调度页面上，而不是执行计算任务，这种现象叫做“抖动”或”颠簸”（thrashing），一个好的调度算法应减少和避免抖动现象。

一个系统随着同时运行的进程的数量的增加，系统的效率的变化曲线如下

//对于抖动的解释放在下面的原因解释之后更合适

****

**//添加一个图更容易理解抖动带来的影响**

**// Operating System Concept (9th Edition) p426**

首先，定义缺页中断率（Linux系统中，缺页中断也称缺页异常），假定进程P共计n页，此进程分得页框m块（m、n均为正整数，且１≤m≤n），即内存中最多只能容纳进程的m页。如果进程P在运行中成功访问次数为S，不成功访问次数为F，则总的访问次数 A＝S＋F，定义：

f＝F/A

称f为缺页中断率。 影响缺页中断率f的因素有：（1）内存页框数：进程所分得的块数多，缺页中断率低，反之缺页中断率就高；（2）页面大小：页面大，缺页中断率低，否则缺页中断率就高；（3）页面替换算法：算法的优劣影响缺页异常次数；（4）程序特性：程序局部性要好，它对缺页中断率有很大影响。

产生抖动的主要原因是，内存中同时运行的进程太多，而分配给每个进程的页框太少，不能满足它们正常运行需要。由于出现频繁缺页异常，使得排队等待页面对换的进程数目剧增，从而，磁盘的有效时间大都花费在对换页面上，处理器的有效时间利用率则大大降低，这时系统产生了抖动。由于页面大小是由操作系统预先规定的，为了降低缺页中断率，防止抖动发生，可从3个方面下手：增加配给进程的页框数、挑选页面替换算法和改进应用程序结构。

例如，一个程序将128×128的数组置初值“0”，假定它仅分得一个内存页框，页面尺寸为128个字，数组中的元素各行分别存放在一页中，开始时第一页在内存中。若程序按如下左边编写：

int A[128][128]；　 　 　 　 　 int A[128][128]；

for（int j＝０；j＜128；j＋＋） 　 for（int i＝０ ；i＜128；i＋＋）

　 for（int i＝０；i＜128；i＋＋） 　 for（int j＝０；j＜128；j＋＋）

　 　 A[i][j]＝０；　 　 A[i][j]＝０；

则每执行一次A[i][j]＝0产生一次缺页，共产生（128×128－1）次缺页异常；若按如上右边重新编写这个程序，共只产生（128－1）次缺页异常。显然，虚存效率与程序局部性程度密切相关，局部性程度因程序而异 。

**7. 全局页面替换策略**

在多道程序正常运行过程中，属于不同进程的页面被分散存放在内存页框中，当发生缺页异常时，如果已无空闲页框，系统要选择一个驻留页面进行淘汰。在此讨论的是所有驻留页面都可作为置换对象的情况，而不管页面所属进程的全局页面置换算法。

（1）最佳页面替换算法

1966年，Belady提出最佳页面替换算法（OPTimal replacement，OPT）。当要调入一页而必须淘汰旧页时，应该淘汰以后不再访问的页，或距现在最长时间后才访问的页。然而，程序页面引用串是无法预知的，不可能对程序的运行过程做出精确断言，不过此理论算法可用做衡量各种具体算法的标准。

（2）先进先出页面替换算法

基于程序总是按线性顺序来访问物理空间这一假设，总是淘汰最先调入内存的页面，即淘汰在内存中驻留时间最长的页面，认为驻留时间最长的页不再使用的可能性较大。先进先出页面替换算法（First-In First-Out replacement，FIFO）的一种实现方法是系统中设置一张具有m个元素的页号表：P[0]，P[1]，…，P[m－1]。其中，每个P[i]（i＝0，1，…，m－1）存储一个装入内存中的页面的页号。假设用索引k指示当前调入新页时应淘汰页在页号表中的位置，则淘汰页的页号应是P[k]。每当调入新页后，执行：P[k]＝新页的页号；k＝(k＋1)%m，就可以了，因为，它是在内存中驻留时间最长的页面 。

这种算法较易实现，对具有线性顺序特性的程序比较适用，而对具有其他特性的程序则效率不高，因为，在内存中驻留时间最长的页面未必是最长时间后才使用的页面，很可能是最近要被访问的页。也就是说，如果某个页面经常被使用，采用FIFO算法，在一定时间后此页面变成驻留内存时间最长的页，这时若淘汰它，可能立即又要用到，必须重新调入。据估计，采用FIFO调度算法，缺页中断率为最佳算法的2～3倍。

FIFO调度算法伴有一种奇怪现象，增加可用物理页框数量会导致更多的缺页异常。假如，采用请求分页调入页面，今有5个页面的访问序列为：4,3,2,1,4,3,5,4,3,2,1,5。现在分配给进程的物理页框为3个和4个，其执行过程分别如下。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **访问序号** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **页框1** | **4** | **4** | **4** | **3** | **2** | **1** | **4** | **4** | **4** | **3** | **5** | **5** |
| **页框2** |  | **3** | **3** | **2** | **1** | **4** | **3** | **3** | **3** | **5** | **2** | **2** |
| **页框3** |  |  | **2** | **1** | **4** | **3** | **5** | **5** | **5** | **2** | **1** | **1** |
| **淘汰页面** |  |  |  | **4** | **3** | **2** | **1** |  |  | **4** | **3** |  |
| **缺页异常** | **√** | **√** | **√** | **√** | **√** | **√** | **√** | **0** | **0** | **√** | **√** | **0** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **访问序号** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **页框1** | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 2 | 1 | 5 | 4 | 3 |
| **页框2** |  | 3 | 3 | 3 | 3 | 3 | 2 | 1 | 5 | 4 | 3 | 2 |
| **页框3** |  |  | 2 | 2 | 2 | 2 | 1 | 5 | 4 | 3 | 2 | 1 |
| **页框4** |  |  |  | 1 | 1 | 1 | 5 | 4 | 3 | 2 | 1 | 5 |
| **淘汰页面** |  |  |  |  |  |  | 4 | 3 | 2 | 1 | 5 | 4 |
| **缺页异常** | **√** | **√** | **√** | **√** | 0 | 0 | **√** | **√** | **√** | **√** | **√** | **√** |

分配给进程3个页框时，产生缺页异常9次。分配给进程4个页框时，缺页异常非但没有减少，反而增加了一次，这种现象便称为Belady异常。但这仅仅是特殊情况，只要系统有足够内存资源，应该给进程增加物理页框数，通过降低整个系统的缺页异常来改善性能。

页面缓冲算法是对FIFO替换算法的一种改进，其策略如下：系统维护两个FIFO队列，修改页面队列和非修改（空闲）页面队列，前者由修改页面的页框所构成的链表；后者由可直接用于装入页面的页框所构成的链表，只不过有些未修改的淘汰页暂时还留在其中，当进程再次访问这些页面时，可不经I/O操作而快速找回。当发生缺页中断时，按照FIFO算法选出淘汰页，并不立即抛弃它，而是根据其内容是否被修改过而进入两个队列之一的末尾，需要装入的页面被读进非修改队列的队首所指向的页框中，不必等待淘汰页写回，使得进程能快速恢复运行。当选中的淘汰页被写回磁盘时，只需把此页占用的页框链接到非修改队列的末尾即可。每当修改页面队列中的页面达到一定数量时，将成批地写回磁盘，并把空闲页框加入非修改页面队列尾部 。

（3）最近最少使用页面替换算法

最近最少使用页面替换算法（Least Recently Used replacement，LRU）淘汰的页面是在最近一段时间内最久未被访问的那一页，它是基于程序局部性原理来考虑的，认为那些刚被使用过的页面可能还要立即被使用，而那些在较长时间内未被使用的页面可能不会立即使用。

为了能准确地淘汰最近最少使用的页面，必须维护一个特殊队列--页面淘汰队列，此队列存放当前在内存中的所有页号，每访问一页时就调整一次，使队列尾总是指向最近访问的页，队列头就是最近最少使用的页，显然，发生缺页异常时总是淘汰队列头所指页面；而执行页面访问后，需要从队列中把此页调整到队列尾。LRU算法的实现需要硬件支持，关键是确定页 面最后访问以来所经历的时间，可采用多种模拟方法。

模拟方法一：引用位法。又称最近未使用页面替换算法（Not Recently Used replacement，NRU）。此方法为每页设置引用位R，每次访问某页时，由硬件将此页的R位置1，间隔时间 t，周期性地将所有页的R位清0。页面置换时，从引用位R为0的那些页中挑选页面进行淘汰，在选中要淘汰的页面后，也将其他页面的引用位R清0。这种实现方法开销小，但t的大小不易确定且精确性差。t大了，缺页异常时所有页的R值均为1；t小了，缺页异常时可能所有页的R值均为0，这样就很难挑选出应淘汰的页面，通常把t定为一个或数个时钟中断周期 。

模拟方法二：计数法。每当页面被引用时，页引用计数器自动计数，更换访问页面时，把页引用计数器的值记录到页表的计数值字段，经过时间t后，将所有页引用计数器全部清除。页面置换时，系统检查所有页表项，页引用计数值最小的页面就是最不经常使用的页，故称最不经常使用页面替换算法（Not Frequently Used replacement，NFU）。

模拟方法三：记时法。为每页增设一个记时单元，每当页面被引用时，把当前绝对时间置入记时单元。经过时间 t 后，将所有记时单元全部清除。页面置换时，系统对各页面的记时值进行比较，值最小的页面就是最久未使用的页面从而淘汰之。

模拟方法四：老化算法。为每个页设置一个多位寄存器r。当页面被访问时，对应寄存器的最左边位置1；每隔时间t，将r寄存器右移一位；在发生缺页中断时，找最小数值的r寄存器对应的页面淘汰。例如，r寄存器共有四位，页面P0、P1、P2在T1、T2、T3时刻的r寄存器内容如下：

时刻

页面 T1  T2  T3

P0 1000 0100 1010

P1 1000 1100 0110

P2 0000 1000 0100

在时刻T3时，应该淘汰的页面是P2。这是因为，同P0比较，它不是最近被访问的页面；同P1比较，虽然它们在时刻T3都没有被访问，且在时刻T2都被访问过，但在时刻T1时P2没有被访问，反映了它最近（3个周期中）被访问的次数最少。越久远发生的事件对决策的影响越小，故此模拟方法称为“老化算法”(aging replacement)，优于前面几种模拟方法，性能好、易实现，常被操作系统采用。

（4）第二次机会页面替换算法

FIFO 算法会把经常使用的页面淘汰掉，为了避免这一点，可对算法进行改造，把 FIFO 算法 与页表中的“引用位”结合起来使用，实现思想如下：首先检查 FIFO 页面队列中的队首，这是最早进入内存的页面，如果其“引用位”是0，那么，这个页面最早进入并且最长时间未被使用，选择此页面淘汰；如果其“引用位”是1，说明虽然它进入内存时间较早，但最近仍在使用，于是将其 “引用位”清0，并把这个页面移至队尾，把它看做一个新调入的页，再给一次机会。这一算法称为第二次机会页面替换算法0（Second Chance Replacement，SCR），其含义是最先进入内存的页面如果最近还在被使用（其“引用位”总保持为1），仍然有机会像新调入页面一样留在内存 中。如果内存中的页面都被访问过，即它们的“引用位”均为1，那么，第一遍检查把所有页面的“引用位”清0，第二遍又找出队首，并把此页面淘汰，此时，SCR算法便退化为FIFO算法 。

（5）时钟页面替换算法

如果利用标准队列机制构造FIFO队列，SCR 算法可能产生频繁的出队和入队，实现代价较高，作为SCR的一种改进，可采用循环队列机制构造页面队列，形成类似于钟表面的环形表，队列指针相当于钟表表针，指向可能要淘汰的页面，这就是时钟页面替换算法（Clock policy replacement，Clock）的得名。此算法与SCR算法在本质上没有区别，仅仅是实现方法有所改进，仍要使用页表中的“引用位”，把进程已调入内存的页面链接成循环队列，用指针指向循环队列中下一个将被替换的页面。算法实现要点如下：

① 一个页面首次装入内存时，其“引用位”置1；

② 内存中的任何一个页面被访问时，其“引用位”置1；

③ 淘汰页面时，存储管理从指针当前指向的页面开始扫描循环队列，把所遇到的“引用位”是１的页面的“引用位”清0，并跳过这个页面；把所遇到的“引用位”是0的页面淘汰，指针推进一步；

④ 扫描循环队列时，如果遇到所有页面的“引用位”均为，指针就会环绕整个循环队列一圈，把碰到的所有页面的“引用位”清0；指针停在起始位置，并淘汰这一页，然后，指针推进一步。

图4-15给出Clock算法的一个例子。当发生缺页异常时，将装入内存的页面是Page 727，指针所指向的是 Page 45（在页框2中），Clock算法执行过程如下：Page 45的“引用位”是1，它不能被淘汰，仅将其“引用位”清0，指针推进；同样道理，Page191（在页框3中）也不能被替换，将其“引用位”清0，指针继续推进；下一页Page 556（在页框4中）的“引用位”是0，于是Page 556被Page 727替换，并把Page 727的“引用位”置１，指针前进到下一页Page 13（在页框5中），算法执行到此结束。

Page9 use=1

Page19use=1

Page1 use=1

Page45use=1

Page191use=1

Page556use=0

Page13use=0

Page67use=1

Page222use=0

Page33 use=1

．

指针

n-1

0

1

2

3

4

5

6

7

8

（A）一个页替换前的缓

冲区状态

Page9 use=1

Page19use=1

Page1 use=1

Page45use=0

Page191use=0

Page727use=1

Page13use=0

Page67use=1

Page33 use=1

Page222 use=0

．

n-1

0

1

2

3

4

5

6

7

8

（B）下一页替换后的缓冲区状态

第1页框

图4-15 时钟页面替换算法

淘汰页面时，如果此页面已被修改过，必须将它重新写回磁盘；但如果所淘汰的是未被修改过的页面，就不需要写盘操作，这样看来淘汰修改过的页面比淘汰未被修改过的页面的开销要大。如果把页表项的“引用位”和“修改位”结合起来使用，可以改进Clock算法，页面共组合成4种情况 ：

① 最近未被引用，未被修改（r＝0，m＝0）；

② 最近被引用，未被修改（r＝1，m＝0）；

③ 最近未被引用，但被修改过（r＝0，m＝1）；

④ 最近被引用，也被修改过（r＝1，m＝1）。

于是，改进的Clock页面替换算法可如下执行：步骤1：选择最佳淘汰页面。从指针当前位置开始扫描循环队列，扫描过程中不改变“引用位”，把遇到的第一个r＝0，m＝0的页面作为淘汰页。步骤2：如果步骤1失败，再次从原位置开始，查找r＝0，m＝1的页面，把遇到的第一个这样的页面作为淘汰页，而在扫描过程中把指针所经过的页面的“引用位r置0。步骤3：如果步骤2失败，指针再次回到起始位置，由于此时所有页面的“引用位”r均为0，再转向步骤1或步骤2操作，这次一定能挑出一个可淘汰的页面。

改进的Clock页面替换算法就是扫描循环队列中的所有页面，寻找既未被修改且最近又未被引用的页面作为首选页面淘汰，因为未曾被修改过，淘汰时不用把它写回磁盘；如果步骤１失败，算法再次扫描循环队列，欲寻找一个被修改过但最近未被引用的页面，虽然这种淘汰页面需要写回磁盘，但依据程序局部性原理，这类页面不会立刻被再次使用；如果步骤2也失败，则所有页面已被标记为最近未被引用，可进入第三次扫描，也称为“第三次机会时钟替换算法”，因为，一个被修改过的页面直到指针已经完成对队列的两次完全扫描之前，将不会被移出，与未被修改的页面相比，它在被选中替换之前还有额外一次机会被引用。

下面给出一个例子，分别用OPT、FIFO、LRU和Clock页面替换算法来计算缺页次数和被淘汰的页面，并对性能作简单的比较 。进程分得3个页框，执行过程中按下列次序引用5个独立的页面：2，3，2，1，5，2，4，5，3，2，5，2。如图4-16所示是4种算法的计算过程和结果。前3个页面2、3、1必产生缺页，于是，OPT算法共产生6次缺页异常；LRU 算法产生7次缺页异常；FIFO算法共产生9次缺页异常；Clock算法共产生8次缺页异常，图中\*表示相应页面的“引用位”为1，箭头“→”表示指针的当前位置，当第一次引用Page5时，由于此时循环队列中所有页面的“引用位”均为1，所以指针绕过一圈并指向Page2，故Page5替换Page2，同时 Page3和 Page1的“引用位”置0；接着引用Page2时，很容易看出应淘汰Page3，所以Page2替换Page3；同样，当引用 Page4时，Page4替换Page1；第 二次引用Page3时，循环队列中所有页面的“引用位”再次为1，因此，指针绕过一圈后Page3替换Page5；当再次引用Page2时，循环队列中3个页面：Page3的“引用位”为1、Page2的“引用位”为1和Page4的“引用位”为0，且指针指向Page2，所以，第三次引用Page5时，显然应替换Page4。可以看出FIFO算法的性能最差，OPT算法性能最好，而Clock算法与LRU算法的性能十分接近。

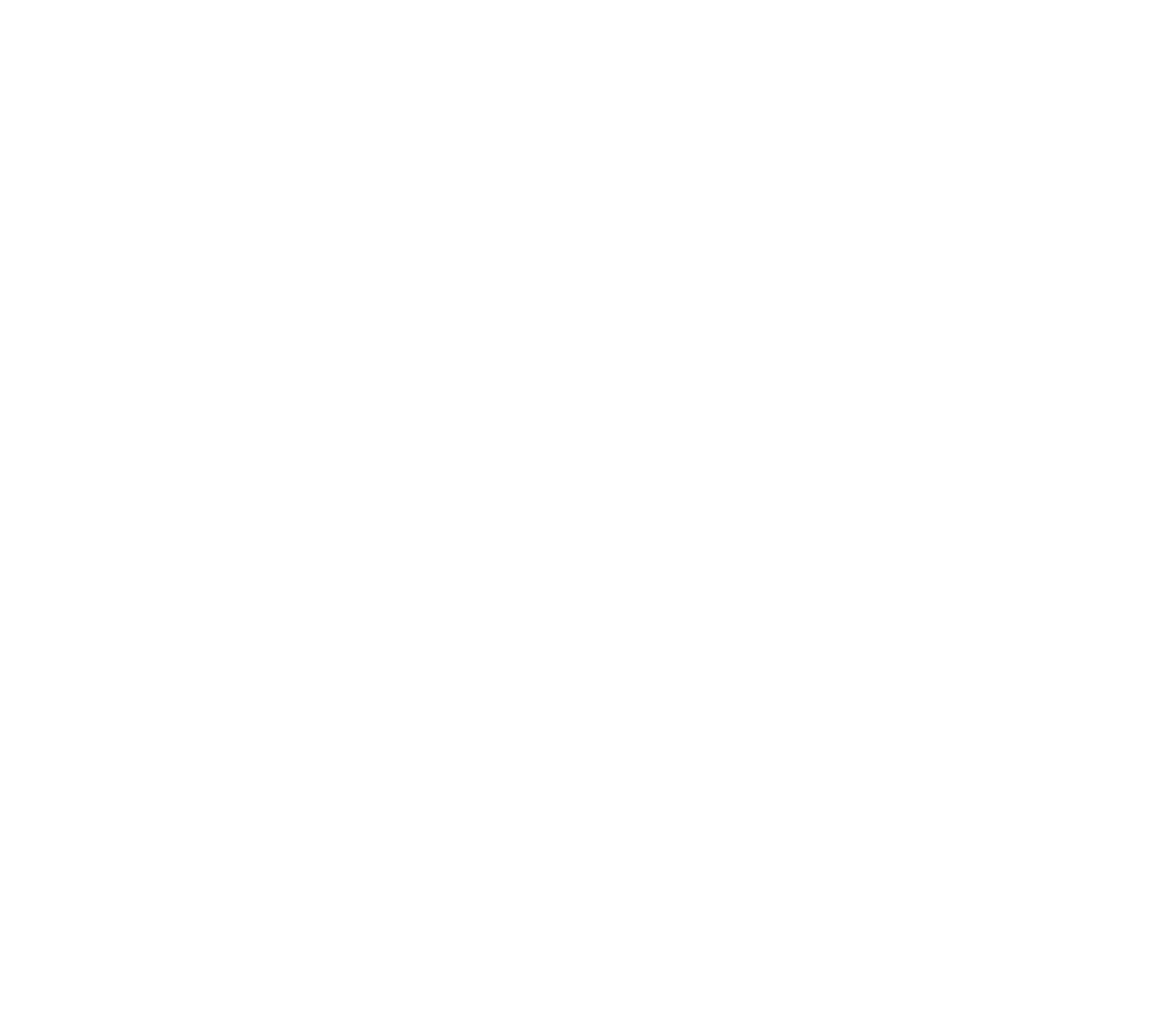


图4-16　4种算法的计算过程和结果

8. 局部页面替换算法

请求分页虚存管理的目标是：找出满足当前进程访问的局部性所需要的页面，然 后，将这些页面加载到内存中，随着程序执行阶段的变化，从一个局部集转到另一个局部集，原来局部集的页面将从内存中卸载，包含新的局部集的页面会被加载到空出来的页框中。类似情况会出现在程序访问数据的不同部分，当某个给定的进程引起缺页时，不允许通过缩小其他进程的驻留页面集来解决问题。下面所讨论的替换算法能用来解决这个难点。

（1）局部最佳页面替换算法

1976年，Prieve提出局部最佳页面替换算法（local Minimum replacement，MIN），与全局最佳替换算法类似，需要预知程序的页面引用串，再根据进程行为改变驻留页面数量，实现思想如下：进程在时刻t访问某页面，如果此页面不在内存中，将导致一次缺页，把此页面装入一个空闲页框。无论发生缺页与否，算法在每一步都要考虑引用串，如果此页面在时间间隔（t，t＋τ）内未被再次引用，那么就移出；否则，此页面被保留在进程的驻留集中，直到再次被引用。τ是一个系统常量，间隔（t，t＋τ)）称作滑动窗口，因为，在任意给定时刻，驻留集包含这个窗口中可见的那些页面（当前引用的页面、未来的 τ次内存访问引用的页面），因此，窗口的实际大小为τ＋１。

通过例子说明此算法，假如进程页面引用串为P3，P3，P4，P2，P3，P5，P3，P5，P1，P4；滑动窗口τ＝3；初始时页面P4已被装入。若采用局部替换算法，通过图４-17来了解驻留集变化情况。在时刻t＝0，P4被引用，因为它在时刻t＝3再次被引用，即在时间间隔（0，0＋3）之内，故P4留在驻留集；在时刻t＝1，P3被引用，它被装入空闲页框中，这时驻留集中包含P3与P4，在时刻t＝2和t＝3，显然，页面P3与P4被保留；页面P4在时刻t＝4被移出驻留集，因为在时间间隔（4，4＋3）之内不再被引用；同时，P2被装入空闲页框，但P2在时刻t＝5就脱离滑动窗口并移出驻留集，而P3依然驻留，直到时刻t＝7再次被引用；发生在时刻t＝6的下次缺页把 P5装入页框，它被保持驻留，直到时刻t＝8再次被引用；最后两次引用装入页面P1和P4。本例中，缺页总数为5次，驻留集大小在1～2之间变化，任何时刻至多有两个页框被占用，通过增加τ值，可减少缺页数目，但其代价是花费更多页框。

（2）工作集模型和工作集置换算法

p.j.denning 提出工作集（Working Set replacement，WS）模型，用来对局部最佳页面替换算法进行模拟实现，也使用滑动窗口概念，但并不向前查看页面引用串，而是基于程序局部性原理向后看，在任何给定时刻，一个进程不久的将来所需内存页框数可通过考查其最近时间内的内存需求做出估计。

进程工作集指“在某一段时间间隔内进程运行所需访问的页面集合”，用W(t，Δ)表示在时刻t－Δ到时刻t（下例中表示为（t－Δ，t））之间所访问的页面集合，它就是进程在时刻t的工作集。变量Δ称为“工作集窗口尺寸”，可通过窗口来观察进程的行为，工作集中所包含的页面数目称为“工作集尺寸”。

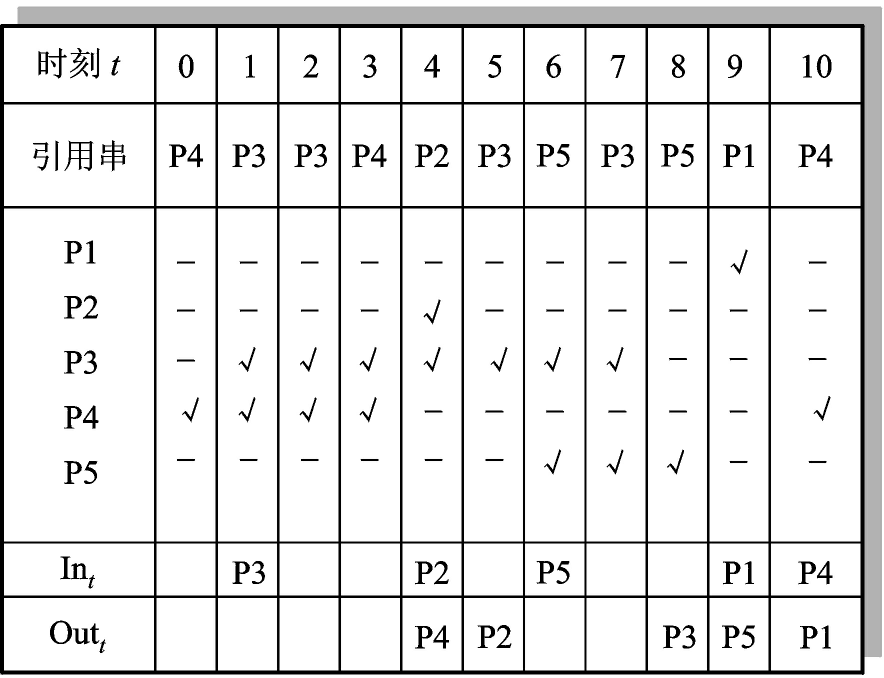


图4-17　 局部最佳页面替换算法示例

在图4-18的例子中，页面引用串与上例相同，工作集窗口尺寸Δ＝3。如果系统有空 闲页框供分配，并且在时刻t＝0时，初始工作集为（P1，P4，P5），其中，P1在时刻t＝0被引用，P4在时刻t＝-1被引用，而P5在时刻t＝-2被引用。第一次缺页异常发生在时刻t＝1，页面P3被装入一个空闲页框，另外3个当前驻留页面P1、P4和P5在窗口（1-3，1）中仍然可见，并被保留；在时刻t＝2，页面P5离开当前窗口（2-3，2），它被移出工作集；在时刻t＝4，缺页异常会把P2装入，它占用移出的页面P1的位置，因为P1已离开当前窗口（4-3，4）；在时刻t＝6，发生缺页异常并装入P5，并且当前驻留页面P2、P3和P4作为由当前窗口（6-3，6）定义的当前工作集的一部分被保；在下面两次引用中，工作集会缩小到仅两个页面P3和P5，并因为在时刻t＝9和t＝10发生两次缺页异常，使工作集再次增长到4个页面。此算法总的缺页数为5次，工作集尺寸在2～4个页框间波动。

工作集是程序局部性的近似表示，可通过它来确定驻留集大小。

① 监视每个进程工作集，只有属于工作集的页面才能驻留在内存；

② 定期地从进程驻留集中删去那些不在工作集中的页面 ；

③ 仅当一个进程的工作集在内存时，进程才能执行。

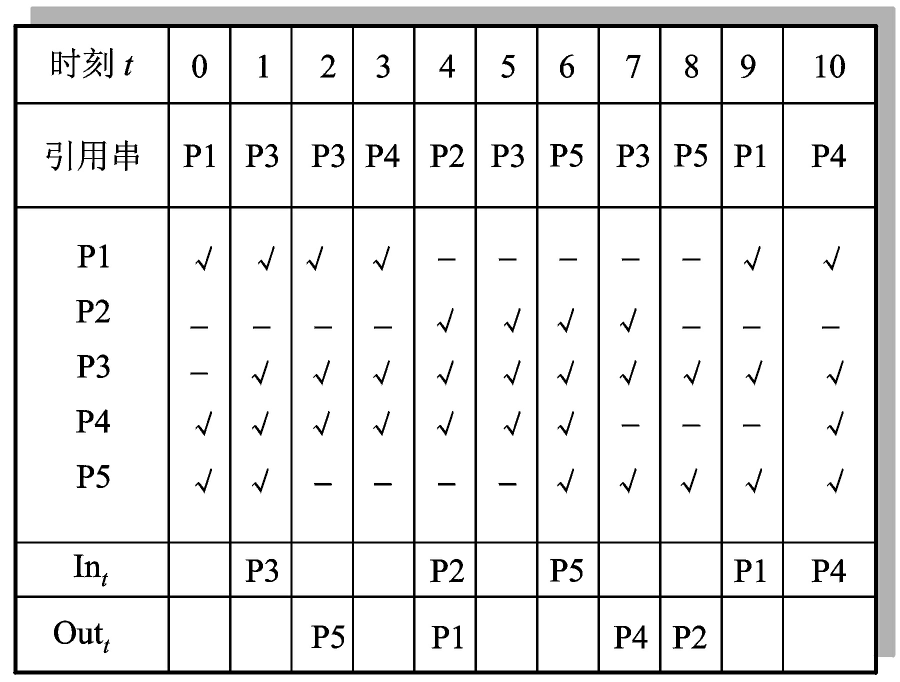


图4-18　 工作集替换示例

（3）模拟工作集替换算法

工作集策略在概念上很有吸引力，但实现中监督驻留页面变化的开销却很大，估算合适的窗口Δ的大小也是一个难题，为此，已经设计出各种模拟工作集替换算法，下面介绍一种。为每个页面设置引用位及关联的时间戳，通过超时中断，至少每隔若干条指令就周期性地检查引用位及时间戳，当发现引用位为1时，就将其置0并把这次改变的时间作为时间戳记录下来。每当发现引用位为0时，通过系统当前时间减去时间戳的时间，计算出从上次使用以来未被再次访问的时间量，记作t\_off。t\_off值会随着每次超时中断的 处理而不断增加，除非页面在此期间被再次引用，导致其引用位为1。把t\_off与系统时间参数t\_max相比，若t\_off＞t\_max，就把页面从工作集中移出，释放相应页面并回收此页框 。

（4）缺页频率替换算法

在工作集算法中，保证最少缺页次数是通过调整工作集大小来间接实现的，一种直接改善系统性能的方法是使用缺页频率替换算法（Page Fault Frequency replacement，PFF）。这种算法根据连续的缺页之间的时间间隔来对缺页频率进行测量，每次缺页时，利用测量时间调整进程工作集尺寸。其规则是：如果本次缺页与前次缺页之间的时间间隔超过临界值τ，那么，在这个时间间隔内未引用的所有页面都被移出工作集。这就能保证进程工作集不会过分地扩大，与工作集模型相比，实现效率高，只在发生缺页异常时才调整页面，而不是每次引用时都需要调整 。

如图4-19所示的例子再次使用上述引用串，并设临界值τ＝2，在时刻t＝0，驻留集合中包含页面P1、P4和P5。在时刻t＝1发生第一次缺页，假设前一次缺页刚刚发生，故本次无页面被移出；下次缺页发生在时刻t＝4，因为本次缺页时刻（4）－上次缺页时刻（1）＞τ成立，所以，在时间间隔（1，4）内未被引用的页面P1和P5应当移出；继而缺页发生在时刻t＝6，但由于本次缺页时刻（6）－上次缺页时刻（4）≯τ，所以，这次无页面被移出；在时刻t＝9发生下次缺页时，因移出条件为“真”，故页面P2和P4被移出；在时刻t＝10发生最后一次缺页时，无页面需要移出。缺页频率替换算法对此引用串共产生5次缺页，工作集页面数在3～4之间波动。

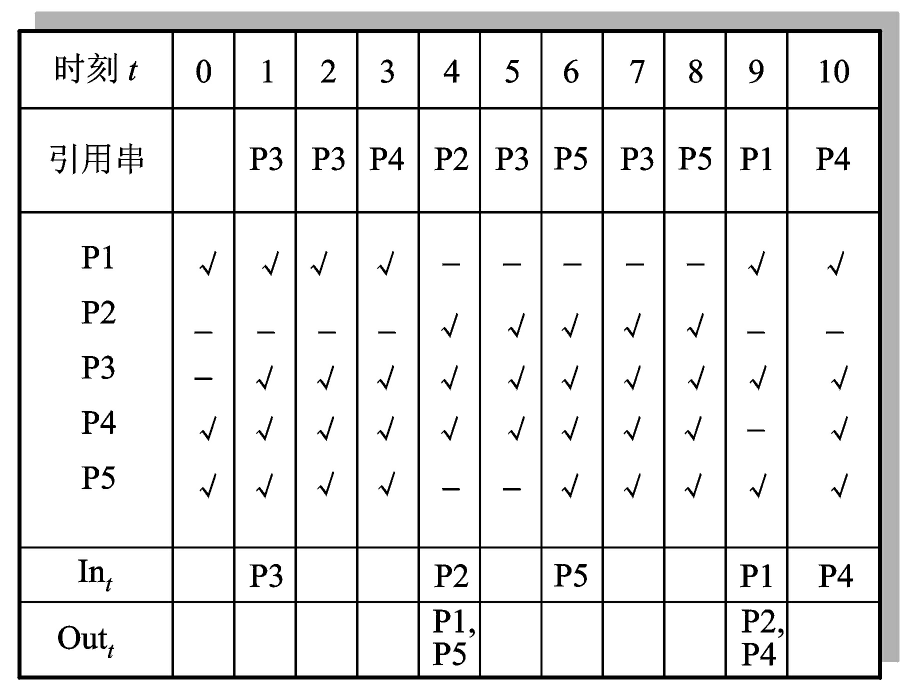
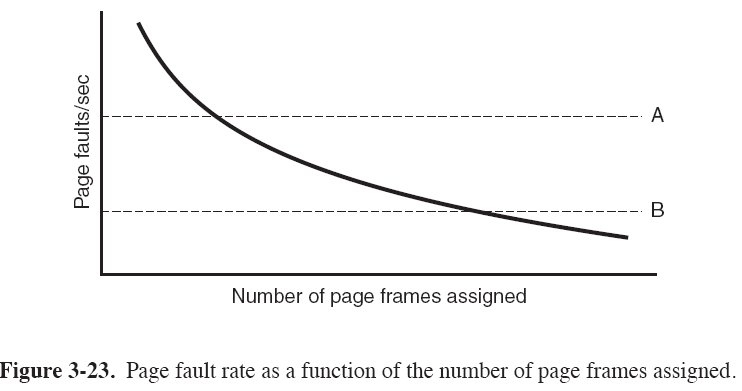


图４-19 缺页频率替换示例

****

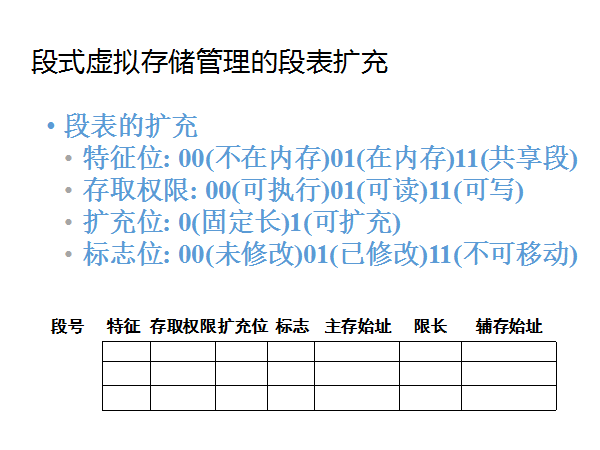
**//Modern Operating Systems (4th edition), page 224.**

**//缺页率与页框数的分配**

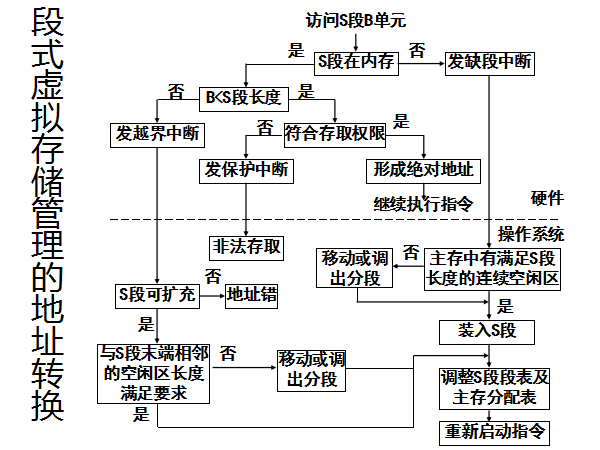
请求段式虚拟存储管理是否要添加一个4级标题

4.5.3\*

**段式虚拟存储管理是指，进程在转入内存时，把进程的所有分段都存放在辅存中，进程运行时先把当前需要的一段或几段装入主存，在执行过程中访问到不在主存的段时再把它动态装入，需要指出的是段式虚拟存储管理中段的调进调出是由操作系统自动实现，对用户透明，和段覆盖技术不同，这是由用户控制的动态扩充技术，操作系统是不感知的。**

****

**但是虚拟存储管理需要对段表进行扩充，显然他既要包括普通段表当中的主存的始址，还要有辅存的位置，显然，主存与限长构成了其实际的内存位置，而辅存始址与限长构成了其辅存的地址，同时标志位也进行扩充，这些标志位可以包括特征位，比如说不在内存，在内存，共享段等等，包括存取权限位，可执行，刻度，可写，扩充位，它的长度能不能动态扩充，还有一些标志位是不是修改过，是不是可以移动等等，基于这样一个段表扩充，段式虚拟存储管理可以来实现相应的地址转换。**

****

**当访问到一个内存当中的地址的时候，硬件系统首先要判断这一段在不在内存，如果这一段不在内存则发出缺段中断，由操作系统来处理，操作系统来判断主存内有没有满足S段长度的连续空闲区，如果没有的话需要调出一个分段，生成一个可以装入S段的空闲区，调整段表，重新启动执行指令。这是段不在内存情况，如果段在内存中，硬件会检测地址是否超越了段的长度，超越了段长则会发出越界中断，同样请求操作系统服务，操作系统查标志位看这个段能不能被扩充，如果不能被扩充显然越界中断，就会产生一个地址错，如果是可扩充的他就要把相应的段扩充，当然相应的段要扩充的时候同样涉及到一个空闲区的长度要求，如果空闲区不够长的话需要移动和调出段，总之有了足够的空间之后，我们就可以把段进行扩充重新执行指令，假如这条路也没有走它是一条内存合法的地址之后，我们就要看他是不是负荷存取权限，不符合存取权限，保护这个进程就该被中止，否则的话这个段在内存，我们就可以形成绝对地址了，那至于怎么形成绝对地址，显然我们可以用可变分区的地址转换的基本思想或者页式地址转换的基本的思想。**

**4.5.3 请求段页式虚拟存储管理**

段式存储是基于应用程序结构的存储管理技术，有利于模块化程序设计，便于段的扩充、动态链接、共享和保护，但往往会产生段之间的碎片，浪费存储空间；页式存储是基于物理存储器结构的存储管理技术，存储利用率高，便于系统管理，但不易实现存储共享、保护和动态扩充。如果把两者优点结合起来，在分段存储管理的基础上实现分页存储管理就是段页式存储管理。

下面介绍请求段页式虚存管理的基本原理。（1）虚地址以程序的逻辑结构划分成段，这是段页式存储管理的段式特征。（2）实地址划分成位置固定、大小相等的页框（块），这是段页式存储管理的页式特征。（3）将每一段的线性地址空间划分成与页框大小相同的页面，于是形成段页式存储管理的特征。（4）逻辑地址分为3个部分：段号 s、段内页号p、页内位移d。对于用户而言，段式虚拟地址应该由段号s和段内位移d′组成，操作系统内部自动把d′解释成两部分：段内页号p和页内位移d，也就是说，d′＝p×块长＋d。（5）请求段页式虚存管理的数据结构较为复杂，包括作业表、段表和页表三级结构。作业表中登记进入系统的所有作业及此作业段表的起始地址；段表中至少包含此段是否在内存的信息，及此段页表的起始地址；页表中包含页是否在内存的信息（中断位）、对应的内存块号。

请求段页式虚存管理的动态地址转换机构由段表、页表和快表构成，当前运行作业的段表起始地址已被操作系统置入段表控制寄存器，其动态地址转换过程如下：从逻辑地址出发，先以段号s和页号p作为索引去查找快表，如果找到，立即获得页p的页框号p′，并与位移d一起拼装得到访问内存的实地址，从而，完成动态地址转换；若查找快表失败，就要通过段表和页表进行地址转换，用段号s作为索引，找到相应的表目，得到s段页表的起始地址s′，再以页号p作为索引得到s段p页所对应的表目，由此得到页框号p′；这时一方面把s段p页和页框号p′置换进快表，另一方面用p′和d生成内存物理地址，完成地址转换。

上述过程假设所需信息都在内存中，事实上，许多情况都会发生，如查段表时，发现s段不在内存，于是产生“缺段中断”，引起操作系统查找s段在磁盘上的位置，并将段调入内存；如查页表时，发现s段p页不在内存，于是产生“缺页异常”，引起操作系统查找s段p页在磁盘上的位置，并将此页调入内存。当内存已无空闲页框时，就会导致淘汰页面。如图4-20所示是段页式动态地址转换和存储保护示意图。

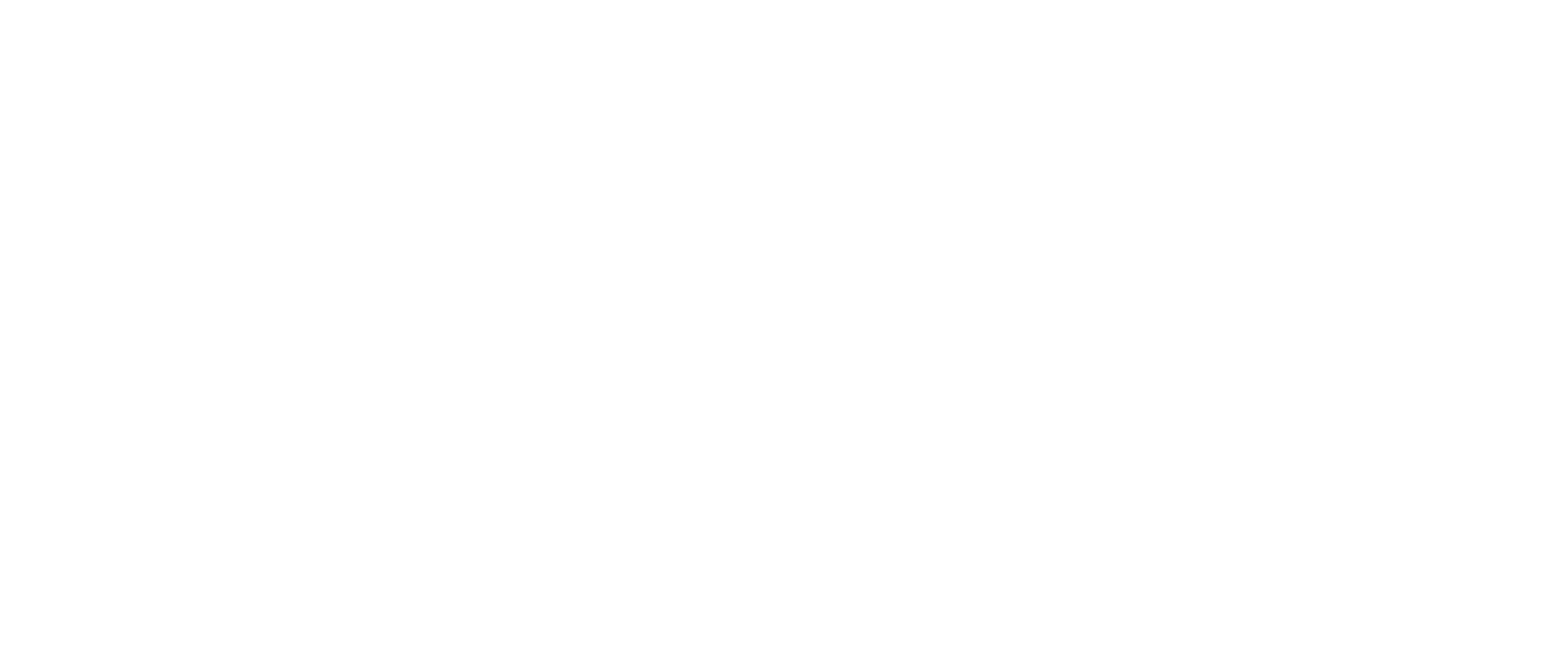
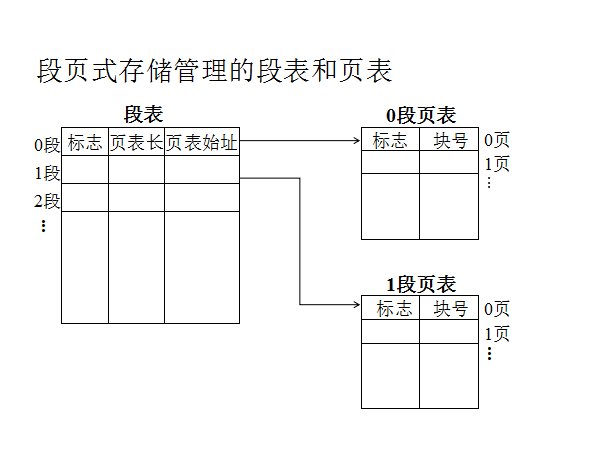
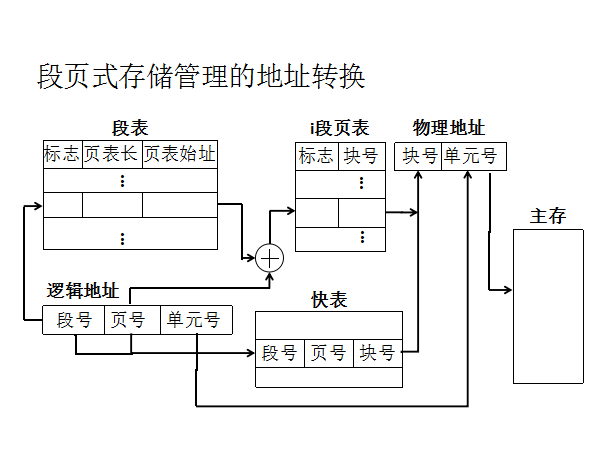


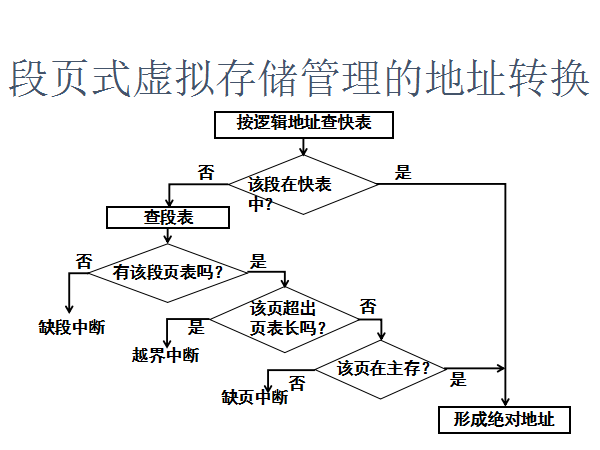
图4-20 段页式动态地址转换和存储保护示意图

****

**在段页式存储管理当中，既有段表，也有页表，页表可能没啥特殊的，而段表不再记录基址和限长，而是页表的基址和限长。**

****

**我们来简单的看一下段页式存储管理的地址转换的示意，我们的一个逻辑地址一个物理地址，我们的段表，页表，快表，以及主存，这样的6个实体，我们根据逻辑地址单元中段号，页号，到快表当中去查，如果查到了那很简单，块号就形成了，页架号就形成了，如果快表当中查不到，那这个过程就稍微复杂了一点，它首先要去查段表，再从段表进行存储保护看有没有超出页进行一个比较，如果过了存储保护，他再去找到他的页表，再从他的页表当中获得他的块号，无论哪条路在往下走再把单元号拼接进去就形成了物理地址，就可以访问主存了，这是段页式存储管理的地址转换的简单示意，当然要完成这样的一个地址转换，他的具体实现流程是很复杂的，他要综合应用我们前面讲到的相应的机制，我们再来总结一下段页式虚拟存储管理的地址转换，刚才讲的是段页式存储管理的地址转换全部在内存中，现在要讲的是虚拟存储管理，同样我是从宏观的角度来看一下，不涉及细节，细节在前面讲页式存储管理，可变分区存储管理当中细致的介绍了。**

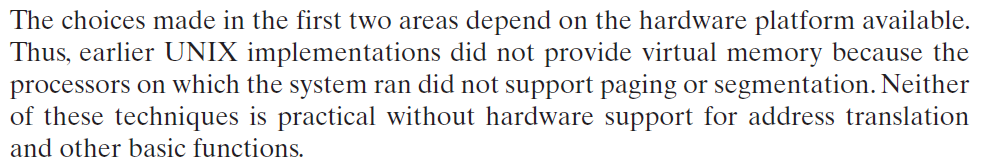
****

**我们首先去查快表，如果该段该页在快表当中，绝对地址就完了，不在快表当中我们就需要去查段表，找到段表之后我们就看他的页表在不在内存当中，要是他不在内存中，整个段都没有装入内存中我就要去发缺段中断，至于缺段中断怎么处理我们就不去细讲他了，前面已经讲过了就是把这个表调入内存，如果页表在内存当中，我们就进行存储保护我们来看一看他有没有超过页表的长度，如果超过了就是越界中断，越界中断怎么处理也就不细说了，通过了存储保护我们就去查该页是不是在内存中，也就是说处理页表，如果该页不在内存中，缺页中断，细节处理不去说他，如果该页在内存当中，显然通过段表存储保护和页表查询我们也得到了一个组成的页架号，我们就可以形成绝对地址，当然我这张图中没有画出他还需要在快表当中补充登记。**

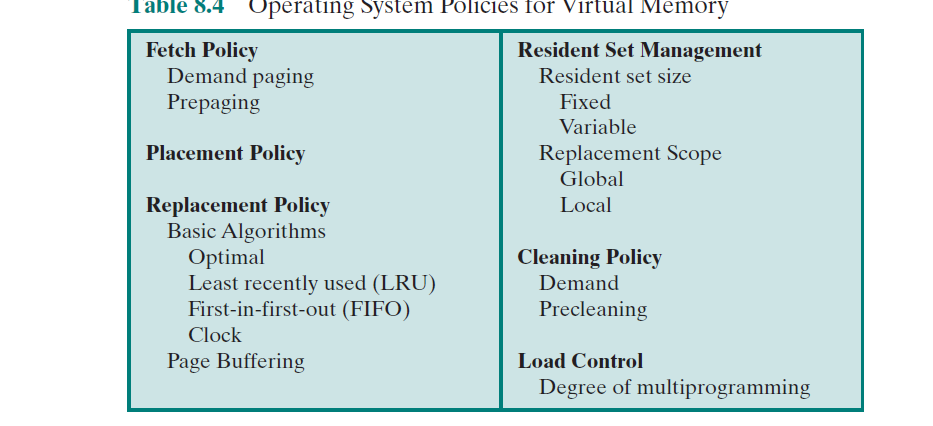
**//操作系统慕课课件《段页式存储管理的地址转换》**

**//这部分内容来自操作系统慕课课件《3.14段页式存储管理》。**

**4.6 存储管理方案及虚存页面替换算法小结**



**//对于操作系统的构建需要考虑的角度如下**

****

**//operating-systems-internals-and-design-principles-8th**

**操作系统对于内存管理方式设计基于以下三个方面的选择：**

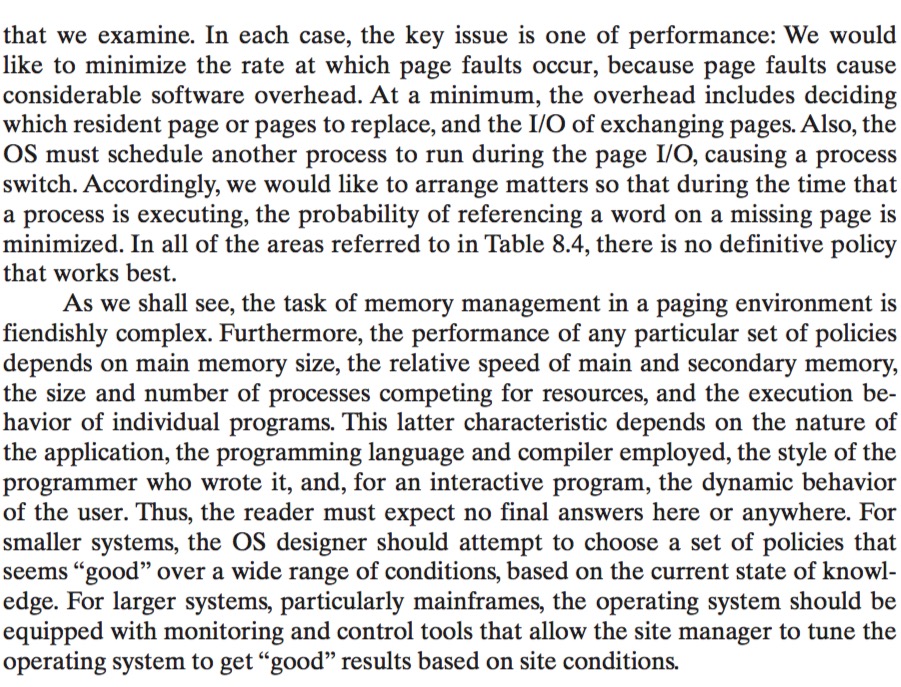
**是否使用虚拟内存技术**

**是否使用分页和分段或者两者都是用**

**在内存管理的各个方面使用的算法**

**前面两个方面的选择是基于硬件平台是否支持，所以，早期的UNIX系统没有提供任何有关虚拟内存的功能，就是因为当时运行操作系统的处理器不支持分页和分段，这些技术没有地址转化等基础的功能，是不可能实现的。**

**第三个领域上的选择是基于以下的考虑，最小化缺页率**



**1. 存储管理方案小结**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 分类 | 管理方案 | 维数 | 多道支持 | 共享内存 | 重定位 | 内存扩充 | 保护机制 | | 硬件支持 |
| 实存  管理 | 单连续分区 | 一维 | 单道 | 不能 | 静态 | 覆盖与对换 | 无 | | 无 |
| 固定分区 | 多道 | 可以(多对界地址) | 静态 | 界地址或键保护 | | 单或多对重定位寄存器 |
| 可变分区 | 动态 |
| 分页 | 可以 | 页表 | 越界保护和存取控制保护 | 地址转换机制与保护机制+快表 |
| 分段 | 二维 | 段表 |
| 段页式 | 段页表 |
| 虚存  管理 | 请求分页 | 一维 | 虚存(内存是外存的缓冲) | 页表 | 同上+动态链接机制+中断机制 |
| 请求段页 | 二维 | 段页表 |

**2. 虚存页面替换算法小结**

|  |  |  |
| --- | --- | --- |
| 算法名称 | 特点 | 比较说明 |
| OPT(最优算法) | 淘汰不用的页或最长时间后才访问的页 | 理论算法，不可能实现，作为衡量标准 |
| FIFO(先进先出算法) | 淘汰最先调入内存的页 | 可能会调出经常使用的页 |
| PageBuf(页面缓冲算法) | 维护修改页、空闲页两个队列，便于再访问页的找回 | FIFO算法的改进，实用和性能好， |
| LRU(最近最少用算法) | 淘汰最近最少使用的页 | 性能好，实现难，常采用近似算法 |
| NRU(最近未使用算法) | 淘汰最近未使用的页 | LRU的近似算法，粗糙 |
| NFU(最不经常使用算法) | 淘汰最不经常使用的页 | LRU的近似算法，粗糙 |
| aging(老化算法) | 通过年龄寄存器各位的累加值，找出应淘汰的页 | 近似LRU算法，但开销低，性能好，易实现 |
| SCR(第二次机会算法) | 先进入内存的、还在使用的页，让其像新页一样留在内存中 | 性能比FIFO算法有改善 |
| CLOCK1(时钟算法算法) | 循环机制构造页面队列，采用单指针，加多条淘汰规则。 | 是SCR算法的改进，实用，性能适中 |
| CLOCK2(改进时钟算法) | “引用位”，采用双指针，加多条淘汰规则。 | 实用，总体性能优于CLOCK1算法 |
| ws(工作集算法) | 引入滑动窗口概念，向后查看页引用串，估算出不久将来所需内存页框数 | 性能好，实现开销大 |
| PFF(缺页频率替换算法) | 根据连续缺页之间的时间间隔来对缺页频率进行测量，找出应淘汰的页 | ws算法的改进，实现效率高 |

**4.7　Linux虚拟存储管理**

**4.7.1　物理内存管理**

**1. 物理内存数据结构**

在传统计算机系统中，物理地址空间是均匀、一致的，CPU访问一个地址所需时间相同，这种内存称均质存储结构。在多处理器系统结构中，CPU访问本地存储器的速度很快，但通过系统总线访问其他CPU的存储器的速度就比较慢。在这种系统中，物理存储空间虽然在逻辑上统一编址，物理上却不在一处，导致访问不同位置的存储器的时间并不一样，这种内存称非均质存储结构。Linux系统为了管理后者，引入了存储节点概念，访问时间相同的存储空间称一个存储节点。为了保证执行效率，一般不进行跨节点操作。对应于Intel x86，实际上只有一个节点。Linux物理内存分成3个层次管理：存储节点→管理区→页框 。

（1）页框管理

物理内存划分成页框(PAGE)，其大小与页面相等，通常为4 KB或8 KB，每个页框由一个struct page结构描述，内核用此结构体来管理内存中的页框，随时了解每个页框的使用情况，例如，是否空闲?是否被占用?谁拥有页框内的页面(如用户进程、内核数据或页缓冲)。系统初始化时通过free\_area\_init()函数创建page结构的数组mem\_map，它本身是由mem\_map\_t组成，每个mem\_map\_t描述一个页框，整个数组就代表系统中的全部页框，数组下标就是物理页框序号，其定义为：

struct page {

struct list\_head list； /\*通用双向链队列结构，链接page\*/

struct address\_sparce \*mapping; /\*指明文件映射空间地址\*/

unsigned long index; /\*映射文件中或对换空间中的偏移量\*/

struct page \*next\_ hash； /\* page cache的hash表中的后继指针\*/

atomic\_t count； /\*访问此页框的进程个数\*/

unsigned long flags； /\*标志位\*/

unsigned dirty； /\*修改标志\*/

struct list\_head lru； /\*页面换出LRU链表或活跃LRU链表\*/

unsigned long age； /\*页面年龄，越小越先换出\*/

unsigned long map\_nr； /\*页框在mem\_map表中的下标、即页框号\*/

struct page \*\*pprev\_hash； /\*page cache的hash表中的前向指针\*/

struct buffer\_head \*buffers； /\*若该页框用做缓冲区，指示缓冲区地址\*/

struct inode \*inode； /\*页框内存放代码或数据所属文件的inode\*/

unsigned long offset； /\*页框内存放代码或数据所属文件的位移\*/

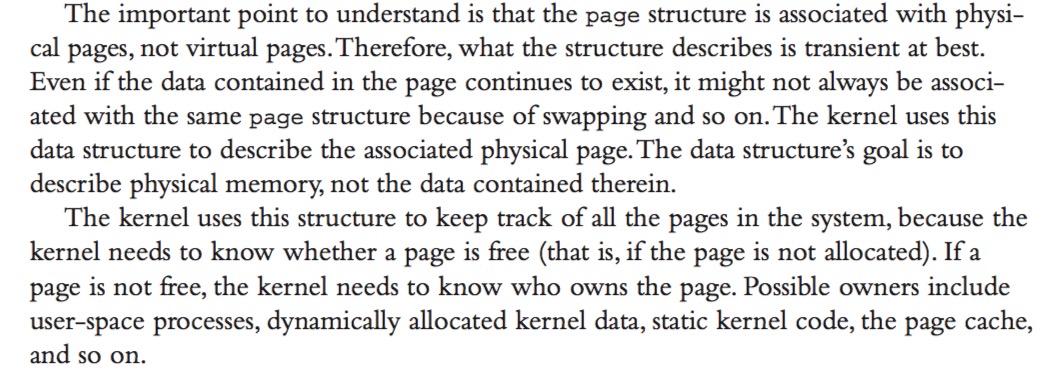
struct zone\_struct \*zone； /\*页框所在管理区\*/

void \*virtual; /\*页面虚存地址\*/

…

}mem\_map\_t；

其中，flags共有32位，描述页框状态，如：页锁定、页被修改过、页正在回写、页是否由slab分配、页不能换出、调页时发生I/O错、完成读操作、从高速缓存或磁盘换入页、DMA传输、内核专用等。此结构中的引用计数变量count指出页框中的页面被引用的次数，当计数为0时，表明没有进程使用该页面，内核可重新分配该页面占用的页框，系统提供函数page\_count()用于检查该变量的值。



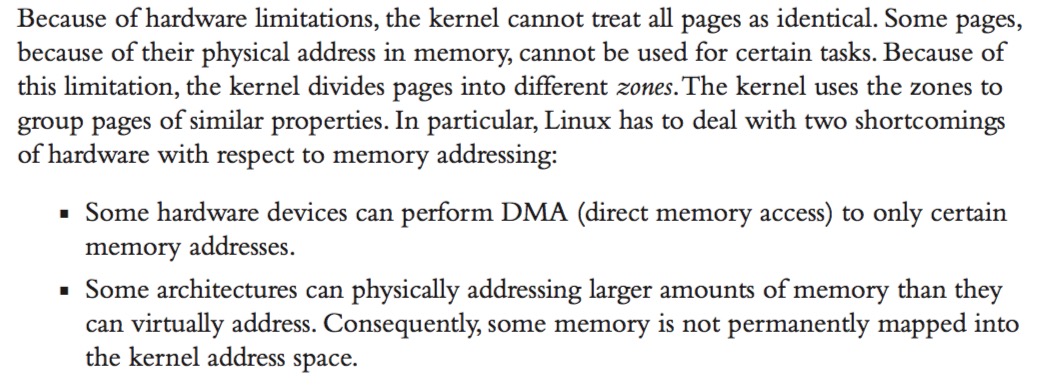
// Linux kernel development p259

Page这个结构体是与具体的物理页面相关联的，而不是虚拟页面，所以这个数据结构描述的是暂时性的内容。即使某个应用的数据重复的出现在内存中，也不能保证他总是被联系到同一个数据结构中，因为内存交换等操作会对物理地址的位置产生影响，所以这个结构是为了描述物理页，而不是其中包含的数据。

\*是否和反置页表有关

随着用户程序的执行和结束，页框不断地被分配和释放，必然导致在已分配的内存块中分散许多小块空闲页框，即使这些小块空闲页框加起来足以满足所请求的页面，但是要分配一个大块的连续页面可能就无法满足。为此，Linux系统采用著名的伙伴（Buddy）算法来解决碎片问题。

（2）管理区管理



// Linux kernel development p260

由于一些硬件的局限性，内核中不能把所有的页面都同样对待，某些页面由于物理地址的特殊性，不能被用作某些工作，由于这些局限性，内核会把页面分到不同的管理区，并用zone来管理具有相同属性的页面，举例来说，一般linux要处理两类由于内存地址引发的硬件缺陷：

1. 一些硬件设备中设备只能对一小部分的物理地址进行DMA操作
2. 某些架构的物理地址范围大于逻辑地址范，所以某些物理内存无法永久的映射到系统的地址空间

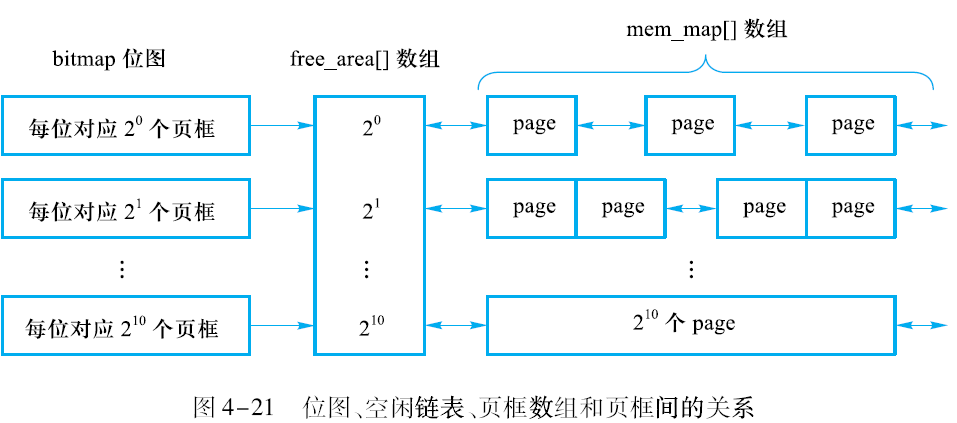
管理区(ZONE)位于同一个存储节点内，所有页框被划分成3个区：ZONE\_DMA区，物理地址低于16 MB，专供DMA使用；ZONE\_NORMAL 区，物理地址介于16 MB-896 MB，被内核直接映射；ZONE\_HIGHMEM区，物理地址大于896 MB，内核不能直接映射区。设置ZONE\_DMA 是保证磁盘I/O操作所需的连续物理页框；用户进程申请页框时，先选定存储节点，再到HIGHMEM内存区选，若不能满足要求，依次再尝试NORMAL、DMA内存区。每个管理区都有数据结构zone\_struct，其中含有一组空闲区队列，空闲区的管理则采用伙伴算法，其内容有：ZONE自旋锁、此区在mem\_map中的起始页框号和下标、空闲区页框数、空闲区位图、伙伴系统中空闲页框链表数组、此区所在存储节点、此区内存映射表、此区起始物理地址、管理区物理内存大小等。内存的管理区分配程序分别管理每个管理区，包括各自的页框高速缓存与伙伴系统。

（3）存储节点管理

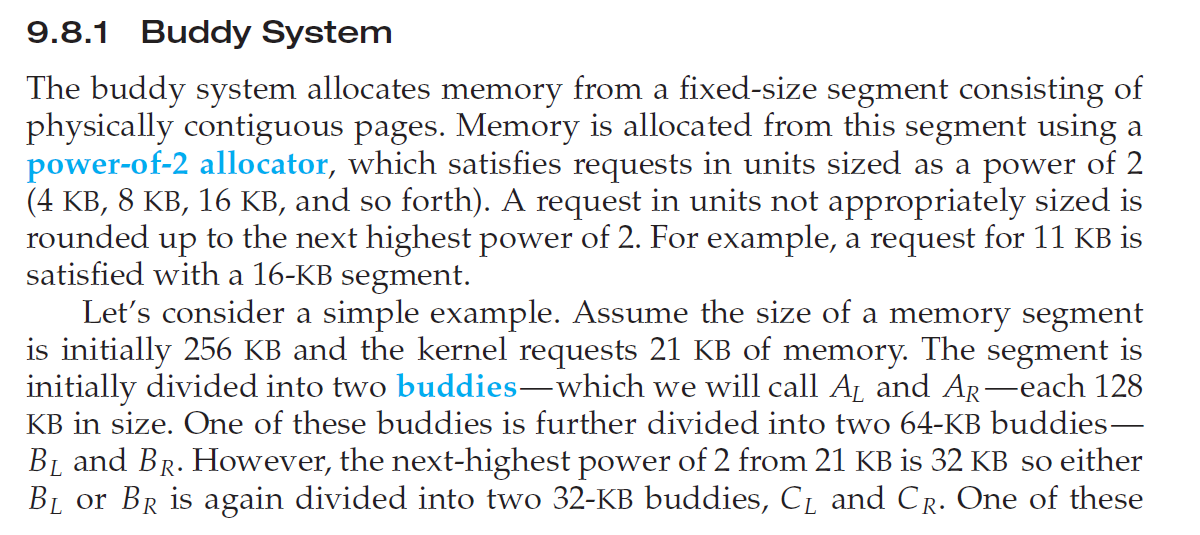
存储节点(NODE)为非均质存储结构而设置，IntelX86平台上只有一个节点。pglist data是为执行跨节点操作而建立的结构体，其内容有：存储节点的管理区数组、存储节点内存映射表、存储节点管理区数目、位示图表示的有效地址、存储节点起始物理地址、存储节点物理内存大小、下一存储节点指针等。

**2．伙伴算法**

Linux系统采用请求分页虚存管理，分配时页框并不需要连续，然而，在执行I/O操作时，会绕过分页机制，要求获得地址连续的页框，直接实现磁盘与内存间的DMA数据传送，这时就需要分配出连续的可使用的内存空间。为此，引进伙伴系统，使用buddy算法来实现连续内存页框的分配与回收。其主要数据结构有：位图、空闲链表、页框数组，见图4-21。



**//这是第五版教材印刷原图**



**//Operating System Concepts (9th Edition), page 437**

**//对伙伴系统的介绍**

(1)页框数组

页框page结构的数组为mem\_map[]。

(2)空闲链表

以free\_area\_struct结构为数组元素的空闲链表free\_area 数组定义如下：

struct free\_area\_struct ｛

　 struct page \*next ；

　 struct page \*prev ；

　 unsig ned int \*map ；

｝free \_area [11]；

此数组记录空闲内存页框，共11个块链表 ，每个链表中的一个块含有2的幂次个页框，把这种块称页框块，大小依次20(1个页框)、21(2个页框)、…、直到210(1024个页框)。因此，free＿area[]数组中的第i个元素代表位图中第i组中空闲块链表的表头。例如，凡是由一个空闲页框组成的空闲链表的表头由free\_area[0]引导；而比邻的2i个空闲物理页框组成的空闲链表，则由free\_area[i]引导。

(3)位图数组

位图数组bitmap共11个，每个空闲页框块的链表对应一张，用二进制数表示主页框使用情况，第0组的每一位表示单个页框的使用情况，为1表示此页框正在使用，为0表示空闲；第1组的每一位表示相邻两个页框的使用情况，如果其中一位置1，表示所对应的两个页框正在使用，依此类推；第i组中的每一位表示相邻2i个页框被使用的情况，例如，第6组中的某位置1，说明对应的26=64个相邻页框正在被使用，仅当64个页框全部回收后，此位才能清0。直接向伙伴系统申请空间和释放空间的函数是alloc\_pages()和free\_pages()。

(4) 页框的分配和回收

内存中满足以下条件的两组页框块称伙伴：两组页框块大小相同(设为b)，两组页框块地址连续，位置居前的一组页框块的首页框号必为2b的倍数。通过例子来说明伙伴算法的工作原理。假设要求分配的页框块其大小为64个页面。该算法先在页框块大小为64个页面的链表中查找。如果有空闲块就直接分配；如果没有则会查找下一个更大的页框块，就是在页框块大小为128个页框的链表中查找空闲块。如果存在这样的空闲块，就把这128个页框分为两半，一半分配出去，另一半插入到页框块大小为64个页框的链表中。如果在页框块大小为128个页框的链表中也没有找到空闲页框，继续找更大页框块。如果存在这样的块，内核就从256个页框的页框块中分出64个页框满足请求，然后，从192个页面中取出128个页框插入到页框块大小为128个页框的链表中，把剩余64个页框插入到页框块大小为64个页框的链表中。

回收过程相反，伙伴算法把满足以上条件的两个页框块合并为一个更大的页框块，该算法是迭代算法，如果合并后的块还可以跟相邻的块进行合并，那么，该算法就继续合并。

**3. slab分配机制**

伙伴系统以页框为基本分配单位，在很多情况下，内核所需要的内存量远远小于页框大小，如inode、vm\_area\_struct、mm\_struct、task­\_struct等。为了更经济地使用内核内存资源，引入solaris操作系统中首创的基于伙伴系统的slab分配器，其基本思想是：为经常使用的小对象建立缓冲存储，小对象的申请和释放都通过slab 分配器来管理，仅当缓冲存储不够用时才向伙伴系统申请更多空间。这样做的好处是：充分利用内存，减少内部碎片，对象管理局部化，尽可能少地与伙伴系统打交道，从而，提高系统效率 。

slab描述符struct slab用来描述每个slab：

struct slab {

struct list\_head; /\*slab满、半满或空闲链表\*/

unsigned long colouoff; /\*slab着色偏移量\*/

void \*s\_mem; /\*slab中的第一个对象\*/

unsigned int inuse; /\*已分配的对象数\*/

kmem\_bufctl\_t free /\*第一个空闲对象\*/

…

}；

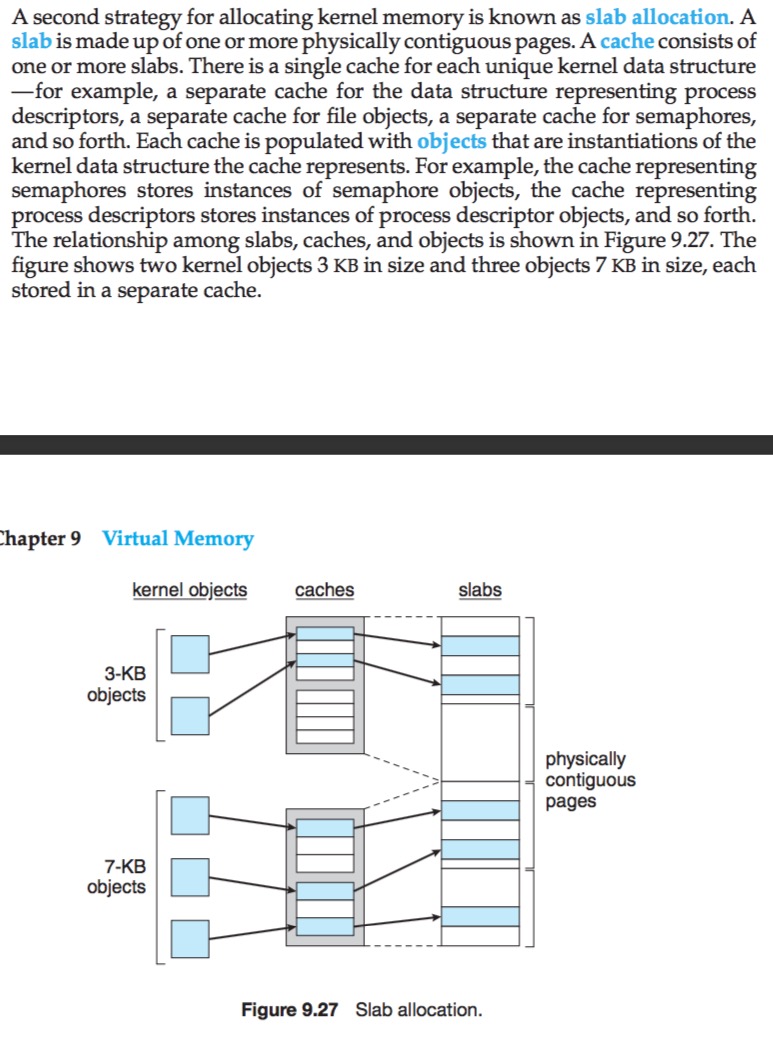
slab分配器的结构和对象空间分配过程如下：内核在内存中建立多个cache，每个cache可保存一种类型对象，每个cache有一个或多个slab组成并形成slab链，每个slab由一个或最多32个物理上连续的页框组成，用于存放对象。例如，一个slab 中存放task\_struct 对象，另一个slab中存放inode对象。每个slab均处于三种状态之一：满（所有对象都已被分配）、半满（尚有空闲对象）和空闲（所有对象均空闲）。当内核分配一个对象时，先从半满的slab中查找，然后，从空闲slab中查找，如果没有空闲对象就应创建一个slab以供分配，这种策略能减少内存碎片。下面来看task\_struct的例子，内核用一个全局变量存放指向task\_struct的slab的指针：kmem\_struct\_t\*task\_struct\_cachep；当内核初始化时，在 fork\_init()中创建高速缓存cache，其中可以存放类型为task\_struct的对象。每当进程调用fork()时，调用内核函数do\_fork()，由它使用 kmem\_cache\_alloc()函数在对应的 slab 中建立一个task\_struct对象，进程运行结束后，task\_struct对象被释放，返还给task\_struct\_cachep slab。

除了这些特定对象的缓冲存储外，Linux系统还提供多种通用缓冲存储，用来满足特定对象之外的普通内存空间需求，单位大小呈级数增长，保证内部碎片率不超过50%。

slab分配器主要操作如下。（1）kmem\_cache\_create()函数：创建特定对象的slab结构，并加入cache所管理的队列。（2）kmem\_cache\_alloc()与kmem\_cache\_free()函数：分别用于分配和取消一个拥有专用 slab队列的对象。（3）kmalloc()与kfree()函数：分别用来从通用缓冲区队列中申请和释放空间。（4）kmem\_getpages()与kmem\_freepages()函数：slab与页框分配器的接口。

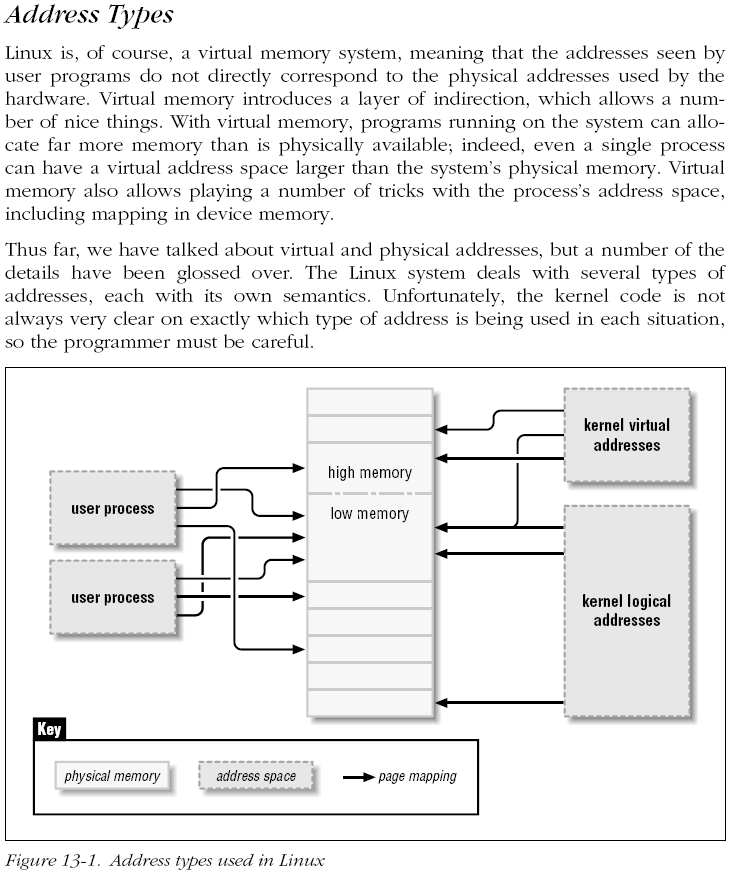
//Operating System Concept 438

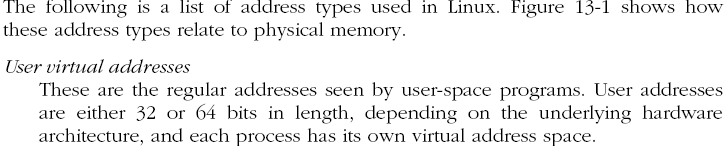
第二种用于分配系统内核内存的方法叫做slab分配，一个slab可以由若干个连续的物理页组成，一个cache包含了多个slab，每一种内核中特定的数据结构都会由一个slab来存储，比如有一个单独的cache用来存储过程描述符的数据结构，有一个单独的cache用来存储文件队形，有的用来存储同步信号量。每一个cache中都保存了特定内核数据结构的实体化对象



**4.7.2 进程虚拟地址空间管理**

**Linux作为现代的操作系统，包含了必须的虚拟内存管理系统，这也就意味着，程序课件的地址并不是直接与物理地址相关的，在之前我们已经了解了虚拟地址和物理地址的关系，现在我们来详细的料及linux操作系统中的实现细节**

****

****

****

**//Linux Device Drivers 3rd Edition, page371**

**//修改理由：Linux系统的虚拟内存的页映射**

在linux中，区分如下几种地址类型，

用户虚拟地址

这是一般的程序员所能看到的地址空间，用户的地址长度可以使32位或者是64位，这取决于底层的硬件架构，每一个进程都会由肚子的虚拟地址空间

物理地址

在处理器和系统内存之间适用的地址，也会区分32位和64位，但是即使是32位的操作系统，在某些情况下也可以使用64位的地址

总线地址

外围总线和存储使用的地址，一般来说处理器不会区分总线地址和物理地址但是也有例外，总线地址与硬件架构密切相关。

内核逻辑地址

这个地址构成了内核的普通地址空间，这个地址映射了大多数或者全部的内存，并且很多时候都被当做物理地址来对待，在很多的架构中，逻辑地址和他们相关联的物理地址唯一的区别就是逻辑地址存在一个偏移值的常量，逻辑地址使用的是硬件本身的指针大小，因此有的时候在重量级的系统中无法取址所有的物理内存，逻辑地址一般存储在unsigned long或者void\*类型的变量中。通过kmalloc方法分配的内存会返回一个逻辑地址

内核虚拟地址

内核虚拟地址和逻辑地址的区别体现在虚拟地址并不一定直接映射到物理地址，所有的逻辑地址都是内核虚拟地址。通过vmalloc方法分配的内存会有一个虚拟地址，但是不一定直接分配到物理地址，虚拟地址信息一般存储在一个指针类型的变量中

不同的内核方法会需要不同的类型的地址，如果说C中定义了不同类型，每一种需要的地址都会被显示的声明，那么很多工作都会变的很方便，但是事实并不是这样

//书中之后的章节讲了32位操作系统如何处理内存容量大于逻辑地址的问题，是否添加存在疑问：

1. -现在的64位系统中暂时不会存在物理地址大于可索引地址的问题
2. +采用高位地址和低位地址的处理方式可能会对理解逻辑地址和虚拟地址的区别有所帮助

虚存空间管理以进程为基础，每个进程最大可拥有3 GB私有虚存空间，进程的内核空间被所有进程共享，进程虚存空间由mm\_struct结构和vm\_area\_struct结构描述。

（1）虚存区描述符vm\_area\_struct

用户进程地址空间很大，有必要显式地表示真正被进程用到的那部分虚存区，进程所用虚存中的各区域未必连续但绝不重叠，由于采用面向对象方法管理，使得虚存区结构体可以代表多种类型的内存区，如代码段、数据段、堆栈段、共享段和内存映射文件等，形成若干离散的虚存区，因此，对虚存区的抽象是一个重要问题。内核将进程的每个虚存区作为一个单独的对象管理，每个虚存区都拥有相同属性，比如访问权限等。vm\_area\_struct结构用来描述进程用户空间的一个虚存区 vma（virtual memory area），而所有vma用来表示此进程实际用到的虚地址空间。vm\_area\_struct结构如下：

struct vm\_area\_struct { /\*虚存区结构\*/

struct mm\_struct \*vm\_mm; /\*虚存区所在mm\_struct指针\*/

unsigned long vm\_start; /\*虚存区起址\*/

unsigned long vm\_end; /\*虚存区末址\*/

unsigned long vm\_flags; /\* 虚存区标志\*/

struct vm\_area\_struct \*vm\_next; /\*下一个虚存区链接指针\*/

…

struct vm\_operations\_struct \*vm\_ops; /\* 虚存区操作函数\*/

struct file \*vm\_file; /\*虚存区映射的文件\*/

unsigned long vm\_pgoff; /\*虚存区中文件的偏移量\*/

…

}vma;

除上面说明的成员外，还有以下内容： vma 存取权限、按地址降序排列链接下一个vma的指针、vma红黑树结构、存放预读信息区等。vm\_flags给出此vma页面的访问权限和相关信息：如“只读”页、“可写”页 、“可执行”页、“可共享”页、vma页面被锁住、vma可作为共享内存、vma可向下增长、vma可向上增长、vma映射不可写文件、vma映射可执行文件、vma不能换出、vma是I/O设备的地址空间、页被连续访问、页被随机访问等。

vma中设置了与磁盘文件相关的域，这是因为在虚存区中的页会与磁盘发生关系。一种是磁盘交换区，当内存不够分配时，一些久未使用的页面可以被交换到磁盘交换区，腾出物理页框供进程使用。另一种是提供系统调用可以将一个打开的文件映射到进程的虚存空间，此后就可以像访问内存中的字符数组那样来访问文件。

为什么要把进程的用户空间要划分为许多vma？这是因为每个虚存区可能来源不同，有可能来自可执行映像，有可能来自共享库，而有的则可能是动态分配的内存区，对不同的vma区具有不同访问权限和不同操作。因此，把进程的用户空间分割管理，把一个vma看成一个对象，用vma\_area\_struct描述这个对象的属性，其中的vm\_operation结构描述在这个对象上的操作，open()、close()分别用于虚存区的打开、关闭，而nopage()是当虚存页面不在物理内存而引起的“缺页异常”时调用的函数。

（2）内存描述符mm\_struct

每个进程有一个内存描述符mm\_struct结构，在进程的task\_struct结构中，有一个指向该结构的指针，mm\_struct结构是对整个用户虚存空间的描述，其定义如下。

struct mm\_struct {

struct vm\_area\_struct \*mmap; /\*指向vma的链表头\*/

struct vm\_area\_struct \*mmap\_cache; /\*最近用到的vma在高速缓存中的指针\*/

pgd\_t pgd; /\*进程页目录基址，运行时写入CR3\*/

atomic\_t mm\_users; /\*共享地址空间的进程数\*/

atomic\_t mm\_count; /\*mm\_struct的引用计数，实现线程使用\*/

int map\_count; /\*进程vma个数\*/

struct list\_head mmlist; /\*内存mm\_list链表\*/

unsigned long start\_code,end\_code,start\_data,end\_data;

/\*进程代码段、数据段起址和末地\*/

unsigned long start\_brk,brk,start\_stack;/\*堆首尾地址及用户栈起址\*/

unsigned long rss,total\_vm,locked\_vm;

/\*进程在内存中的页面数，所需总页面数，被锁住的页面数\*/

…..

};

除已说明的外，还包含的成员有：指向vma的红黑树的根、读写信号量、自旋锁、访问标志等。

//对于linux的编写，结合代码是必须的，但是要做到什么程度，我认为应当把主要的流程性概念介绍一下，linux在处理硬件问题上采用的策略等，而不应当是把所有的方法使用指南都罗列一番，毕竟有些不伦不类

下面讨论内存描述符的分配和撤销，进程描述符的mm域存放进程所使用的内存描述符，故current-＞mm便指向当前进程的内存描述符。fork()函数利用copy\_mm()函数复制父进程的内存描述符，即把current-＞mm域交给其子进程，而子进程中的 mm\_struct结构则是通过内核的allocate\_mm()来建立，每个进程都有唯一的 mm\_struct结构，即唯一的进程虚存空间，子进程的虚存空间也就创建好了。如果父进程希望和子进程共享地址空间，可在调用clone()时设置CLONE\_VM标志，这样的子进程称作线程。当CLONE\_VM 被指定后，内核就无须调用allocate\_mm()函数，而仅需要在调用copy\_mm()函数时将 mm 域指向其父进程的内存描述符。进程退出时，内核会调用 exit\_mm()函数执行常规的撤销工作，同时更新统计量，如exit\_mm()函数会调用 mm\_put()函数减少内存描述符中的mm\_users用户计数，如果用户计数降至0，继续调用 mm\_drop()函数，减少mm\_count计数；如果 mm\_count计数也等于0，说明此内存描述符不再有使用者，调用free\_mm()宏通过 kmem\_cache\_free()函数回收mm\_struct结构。如图4-22所示是进程虚存管理数据结构。进程描述符是内核中的核心数据结构，task\_struct中含有指向mm\_struct的指针，而该内存管理结构中包含了进程的可执行信息，还包含指向vma的链接指针mmap，并且可找到按照降序用 vm\_next 链接起来的进程的所有vma。此外，每个进程有一个页目录pgd，存储此进程所使用的页面情况，根据缺页调度原则，只分配所用到的内存页面，从而，避免页表占用过多物理内存空间。

进程任务结构

task\_struct

\*mm

虚存区结构

vm\_area\_struct

\*vm\_mm

vm\_\_start

vm\_end

\*vm\_ops

\*vm\_next

页目录表

pgd

图4-22 进程虚存管理数据结构

内存管理结构

mm\_struct

\*mmap

……

\*pgd

封装的操作集vm\_operations\_struct

open( )

close( )

unmap( )

swapin( )

页表

PTE

页框

PF

(共享库)

进程虚拟内存

虚拟内存段

(0x40000000)

(data)

虚拟内存段

(0x0804a020)

(text)

虚拟内存段

(0x08048000)

虚存区结构

vm\_area\_struct

\*vm\_mm

vm\_\_start

vm\_end

\*vm\_ops

\*vm\_next

虚存区结构

vm\_area\_struct

\*vm\_mm

vm\_\_start

vm\_end

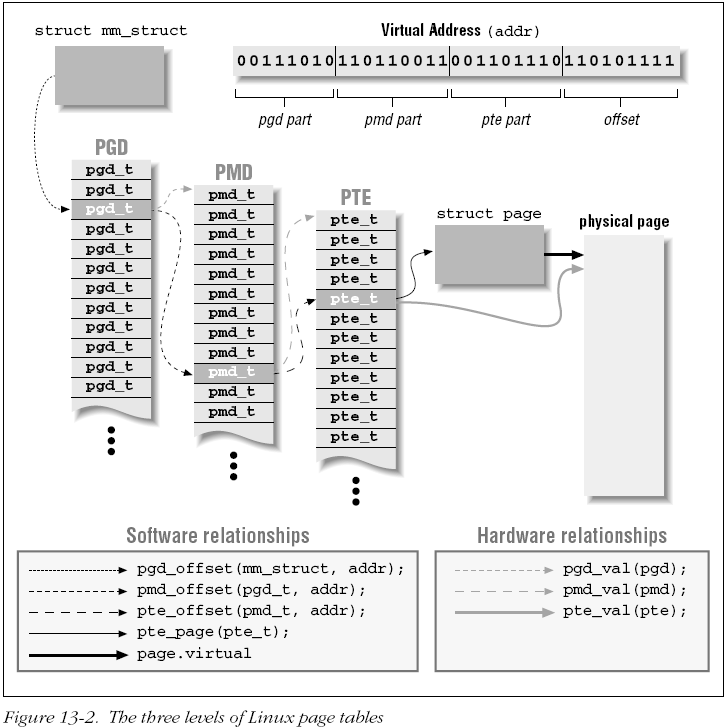
\*vm\_ops

\*vm\_next

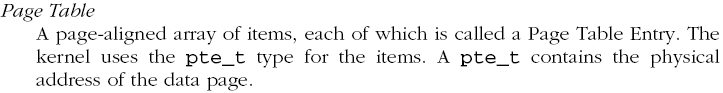
…

图４-22 进程虚存管理数据结构

**4.7.3 页表机制**

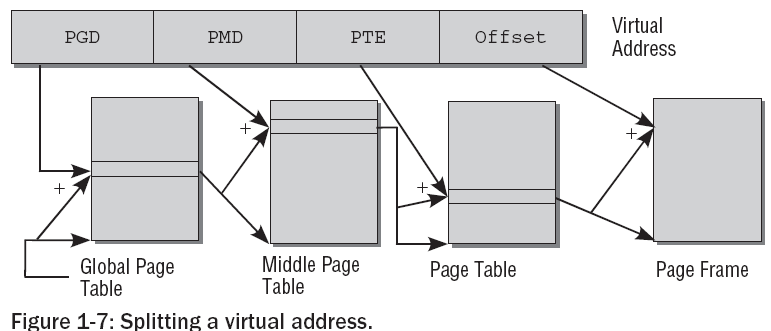
****

****

****

**//Linux Device Drivers 3rd Edition, page376**

**//Linux系统的三级页表设计**

****

**//Professional Linux Kernel Architecture, page 12**

**//Linux三级页表**

在linux内核中，使用三级页表来映射虚拟地址到物理地址，这种多层的结构使得分散的内存存储方式也可以实现，现代的操作系统会在一个巨大的虚拟内存空间上展开，这么安排是有意义的，在系统运行的过程中提供较好的适应性。

注意到即使是在只提供两层页表映射虚拟内存的硬件架构上，linux依然可以使用三级页表。由于三级页表的实现方式是和处理器型号无关的，这种方便的方式并不会在只有二级页表的设备上带来性能损耗，因为编译器会自动的分辨出不适用的页表层级。

页目录

中间级页目录

页表

操作系统内核不必处理缺页中断后去查找所需内存的工作，因为硬件会完成这部分工作，但是操作系统内核要安排好硬件查找所需要的页表，这样在发生缺页中断的时候就能让硬件查找到对应的页。

不考虑CPU使用的机制，linux的软件操作方式是基于三层页表的，至于实际中如何操作，不同的架构会采用不同的策略

//书中的内容太复杂，而且没有涉及到linux是如何具体的实现不依赖架构的三级页表方法的

在Intel x86处理器上，可区分两种地址：线性地址和物理地址。线性地址用来表达4 GB虚拟地址空间。后者指内存实地址，用于内存单元寻址。Linux系统采用分页机制实现虚存管理，每个进程都有自己独立的虚拟地址空间，进程虚拟地址空间通过进程的页目录表和页表实现与物理内存的映射。进程创建时分配一个虚拟地址空间，直到实际对物理内存进行操作时，才通过请页机制分配物理内存。虚存以页为单位，大小等同于物理页框。通常页大小定义为4 KB，页面可以映射到任一物理页框。进程线性地址为32位，转换为虚拟地址，分为以下3个部分：页目录表(高10位)——记录在页目录中的索引；页表(中间10位)——记录在页表中的索引；偏移量(低12位)——用来表示4KB页框中的位移。每个进程有一个页目录，每当它运行时，寄存器CR3指向该页目录基地址，显示从线性地址到物理地址的映射过程。页目录用一个物理页框存储，保存页表的基地址，每个数据项占4 B，因而，页目录有1024个页目录项。以页目录值为索引，在页目录中找到页表基地址。页表也用一个物理页框存储，用来保存物理页框号，每个数据项占4 B，因而，页表有1024个页表项。物理页框号拼接偏移量即得到对应物理地址。

Linux系统的页表项共32位，除4位备用外，其定义如下：页框基地址(12-31位)，其余为页框使用情况和保护标志位。Page Size(7位)置位表示2KB页面(否则4KB)，Dirty(6位)置位表示页面被修改过，Access(5位)置位表示页面被访问过，PCD(4位)置位表示关闭页面高速缓冲(否则可缓存)，PWT(3位)置位表示页面缓冲采用同直写缓冲和内存(否则仅写缓冲)，User/Kernal(2位)置位表示用户级保护(否则内核级保护)，Read/Write(1位)置位表示页面可读可写(否则只读)，Present(0位)置位表示页面在内存。笫0位是异常位，为0表示对应页尚不存在，当访问该线性地址对应页面时，将产生页面异常，导致页面装入并填充页表项。

**4.7.4 缺页异常处理**

1. 缺页异常处理过程

~~页面失效在Linux系统中被称为缺页异常~~，为了在内存资源有限情况下实现多进程并发执行，在可执行程序第一次被装载时，仅其中一小部分映像被装入内存，当处理器执行到一个不在内存的页面时，CPU控制单元就会引发一次缺页异常。处理器自动终止当前程序的执行，转去执行缺页异常处理程序，它将需要执行的页面调入内存后，再恢复被中止的程序执行。Linux系统缺页异常处理程序 do\_page\_fault()能够区分两种情况：如果编程错误而引起的对保护页面非法访问的异常，这时应根据 vma结构中的vm\_flags域来区别情况，分门别类地进行处理；如果进程在执行时，处理器访问到一个有效虚地址，但该地址对应页面尚未调入内存，这时产生缺页异常，并把缺页的虚地址保存在控制寄存器CR2中，连同缺页时访问虚存的模式一并传给缺页异常处理程序 ，它读取引起缺页的虚拟地址，进行以下步骤处理。

（1）检查异常发生时处理器是否正在处理中断，或正在执行内核线程，如是则表示内核发生错误，进行出错处理。

（2）调用 find\_vma 找到发生缺页异常的虚地址所在的vm\_area\_struct，以确定此错误的虚地址是否包含在进程虚地址空间中，或在堆栈的合理扩展区中 。如为非法虚地址或非法类型，内核发出信号终止进程。

（3）若错误由写访问引起，则检查此vma是否可写。若禁写，需发信号终止进程，否则可采用“写时复制”操作，调用do\_wp\_page()，再调用get\_free\_page()分配一个新页框，再把旧页面内容拷入到新页框中。

（4）若异常是由读或执行访问所引起的 ，则检查此页面是否已经在内存中 ，若不在内存中且虚地址区的访问权限与引起异常的访问类型相匹配，则执行调用handl\_mm\_fault()，再调用do\_no\_page()分配一个页框并初始化，再“请求调页”处理。

（5）检查进程页表项中的特征位，区分缺页所对应的页面是在交换空间，还是在磁盘文件映像，最后 ，调用swap\_in( )进行“页面调入”操作 。

2. Linux页面替换算法

Linux页面替换算法是一种改进的最少使用频率策略。最早用access标志指示页面被访问情况，以便为页面淘汰作准备。为了精确地做好页面淘汰，页表项中设置一个8位的age变量来替代access，将根据内存中当前页面的年龄来决定是否将其替换。每个被调入内存的页面都有一个起始年龄，其默认值是3。如果页面长久没有被访问过，它会变老。系统规定，页面每次被访问时其age增加3，kswapd内核线程周期性地扫描全局页池 ，每个页面的年龄将减1，直到0时为止。显然age的值越小，所对应的页面越老，交换进程只替换老页面，因为它是最近没有被访问过的页面。age值越大，此页最近被使用过的频率就越高，也就越不适合替换。如果被替换的页面已修改过，必须在释放前将其写回到交换空间。

在Linux系统中，内核将暂时不用的内存数据或代码以页为单位从物理内存转储到交换区中。当需要这些页面时，再从交换区调回内存。交换区是用预留的若干硬盘分区来保存换出的页面，这些硬盘分区称为交换设备。内核为每个交换设备建立一个swap\_info\_struct结构，并定义一个交换设备数组swap\_info[]，结构内容有：发生交换的设备号及优先级、指向文件或文件目录设备的目录项、计数器数组指针(每个元素对应一个页框)、交换设备中首个/末个可用页、搜索指针、交换设备可用页面数、交换区大小、链接指针等。

当系统中的物理内存空间减少时，存储管理必须释放物理页框，这个任务由内核线程kswapd完成，其目标是保证系统中有足够空闲页框来满足应用程序需求和系统运行效率，它被内核对换定时器周期性地调用，检查系统中的空闲页框数是否太少，用两个变量free\_pages\_high 和 free\_pages\_low来判断是否释放页面；如果系统中的空闲页框数大于上限，kswapd就不做任何工作，并睡眠到下一次定时器到时为止；如果系统中的空闲页框数在上限甚至下限以下时，将通过3途径来减少系统中所使用的页框的数量。（1）换出页面缓存和数据缓存的页面。（2）换出System V IPC的共享内存页面。（3）换出或者丢弃进程占有的页面。如果系统中的空闲页面数低于下限，kswapd将在下次运行之前释放6个页面，否则它只释放3个页面。将依次使用上述途径直到系统释放足够的空闲页框，然后，kswapd线程再次睡眠到下次定时器到时为止。如果导致释放页面的原因是系统中的空闲页框数小于下限，则它只睡眠平时的一半时间，一旦空闲页面数大于free\_pages\_low，那么，它的睡眠时间又会延长。

Linux内存管理系统中还有另一个守护进程pdflush，通常每隔500 ms，它被周期性地唤醒，完成两项任务：（1）把非常老的“脏（dirty）”页面写回到磁盘；(2）当可用的内存水平下降到一个阈值时，由内核唤醒，把页面缓存的“脏”页面写回到磁盘。

**4.8 本 章 小 结**

操作系统存储管理的基本功能有：存储分配、地址转换、存储保护、存储共享和存储扩充。存储分配是指为选中的多道运行作业分配内存空间；地址转换是把逻辑地址空间中的程序和数据通过静态或动态地址重定位转换和映射到分配的物理地址空间中，以便程序执行；存储保护指各道程序只能访问自己的存储区域，而不能互相干扰，以免其他程序受到有意或无意的破坏；存储共享指内存中的某些程序和数据可供不同的用户进程同时使用。

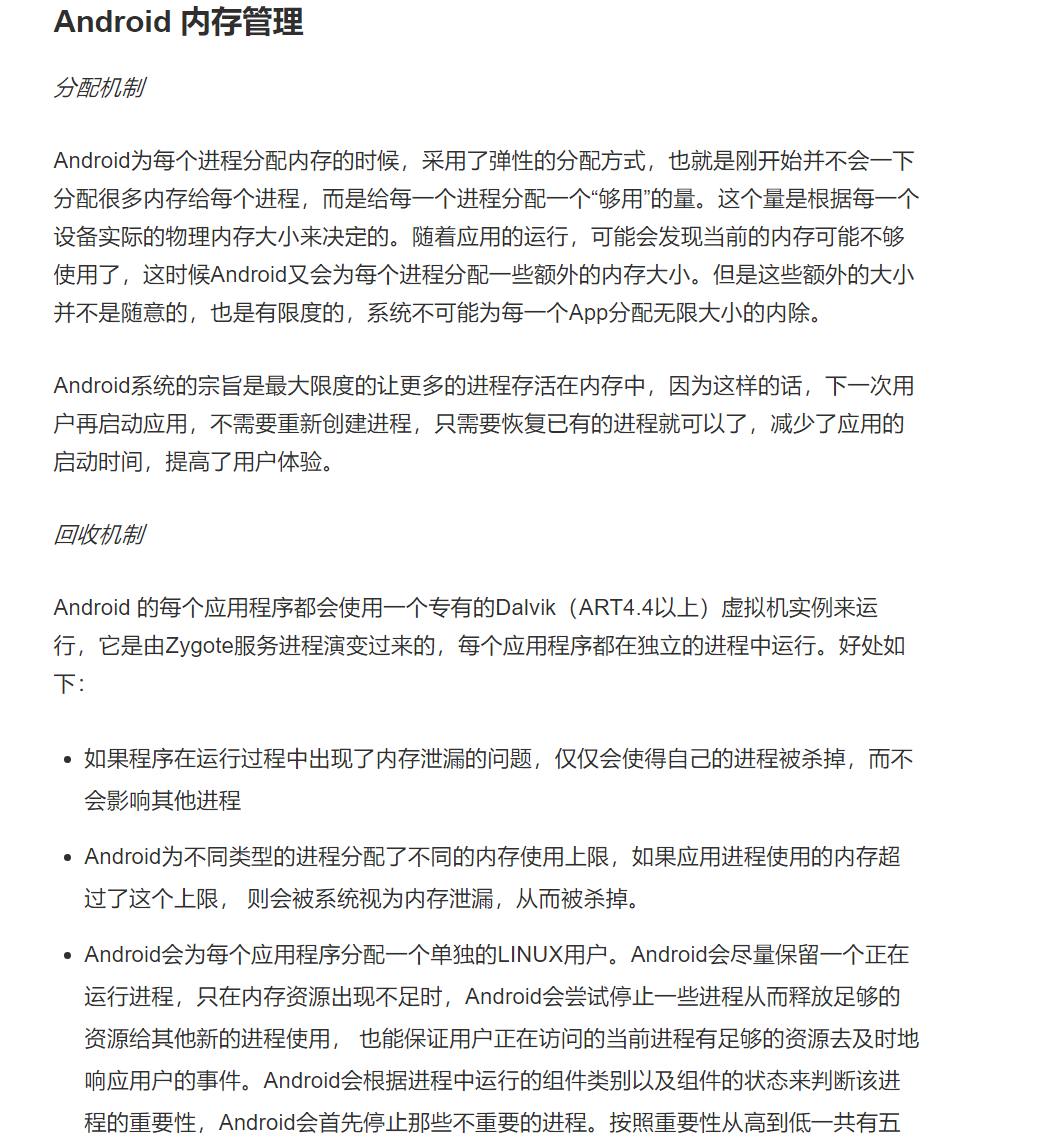
存储管理的另一个重要任务是解决好存储扩充问题，使得内存中进入尽可能多的进程，使得内存利用率得以提高，使得应用程序不受可用物理内存大小的限制。操作系统支持多个用户进程在内存中同时运行，能满足多道程序设计需要的最简单的存储管理技术是分区方式，又分为固定分区和可变分区，可变分区的分配算法包括：最先适应、下次适应、最佳适应、最坏适应和快速适应等分配算法。内存空间不足时主要采用的技术有：移动、对换和覆盖技术。

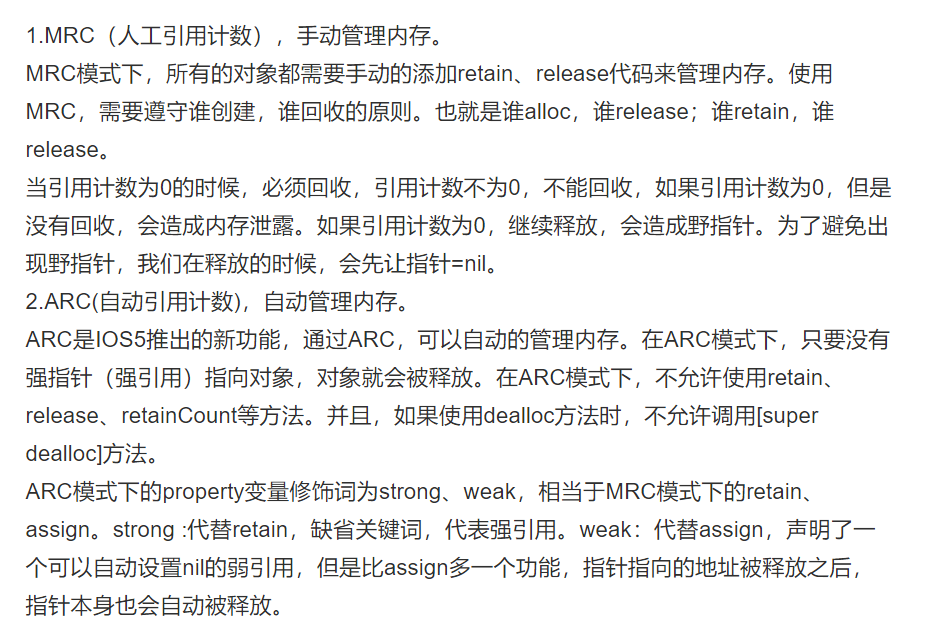
现代计算机系统都有某种虚拟存储硬设备支持，其中简单也是常用的是请求分页式虚存管理，允许把进程的页面存放到若干不相邻接的内存页框中。虚拟存储器的思路是：基于程序的局部性原理，不把一个进程的全部信息装入内存，而是仅将其中的当前使用部分先装入内存，当进程使用某部分地址空间时，保证将相应部分的程序加载到内存中。它采用自动“部分装入、部分对换”技术、内存和外存独立编址统一使用的技术，使得进程的虚拟地址空间可以远远大于系统的物理地址空间，为用户编程提供了极大方便。请求分页虚存管理的基本原理包括：页面、页框、逻辑地址、页表、地址转换等，相关概念还有：加快地址映射的快表、减少内存空间占用的多级页表等。在一个实际操作系统中，还有以下一些重要问题：页面装入策略：决定页面何时被装入内存，有请页式和预调式两种装入策略。页面清除策略：决定修改过的页面何时被回写到磁盘，有请页式和预约式两种清除策略。页面分配策略：根据进程生命周期中分配的页框数可否改变，分为固定分配和可变分配。页面替换策略：根据页面替换算法的作用范围是整个系统还是局部于进程，分为全局页面替换算法和局部页面替换算法。

已经设计出许多页面替换算法，全局页面替换算法有：OPT、FIFO、LRU、SCR、Clock等。局部页面替换算法有：工作集、模拟工作集、缺页频率算法等。可通过工作集模型来指导确定进程常驻集的大小，使得进程的缺页中断率低，内存空间又能得到充分利用。OPT算法虽然好但并不可行，常用的算法是 LRU和 Clock算法及其变种。

请求分页虚存管理不能支持模块化程序设计，在分页式虚存管理的基础上实现分段式虚存管理就是段页式虚存管理。它能满足现代高级语言模块化程序设计所需的二维地址要求以及方便用户（程序员）编程和使用。请求段页式虚存管理的基本原理：（1）虚地址以程序的逻辑结构划分成段。（2）实地址划分成位置固定、大小相等的页框。（3）将每一段线性地址空间划分成与页框大小相同的页面，于是形成段页式存储管理的特征。（4）逻辑地址分为4个部分：段号s、段内页号p、页内位移d 。对于用户而言，段式虚拟地址应该由段号s和段内位移d′组成，操作系统内部自动进行地址变换。

操作系统的存储管理功能与硬件存储器的组织结构和支撑设施密切相关，操作系统设计者应根据硬件情况和用户需要，采用各种有效的存储资源分配策略和保护措施。





**//IOS与Android的内存管理机制和普通平台不同**

[**https://www.jianshu.com/p/66b5d43b6ac4**](https://www.jianshu.com/p/66b5d43b6ac4)

**https://www.jianshu.com/p/5bd50b15177f**