彎曲評論
科技 · 人物 · 潮流

《孙钟秀 . 操作系统教程》注释(稿)

(第四章--内存管理)

编注: 陈怀临

CPU与主存之间的Bus（总线）通常是瓶颈。

第四章

Memory Management
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——主存储器。由于任何程序和数据必须占用主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。主存储器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，但高效的主存管理仍然是操作系统设计中的重要课题。

On－Chip Memory

主存空间一般分为两部分：一部分是系统区，用于存放操作系统内核程序和数据结构等；另一部分是用户区，用于存放应用程序和数据。存储管理对核心区和用户区都提供相应的支持和进行管理，当然也包括对辅助存储器（磁盘）空间的管理。尽管现代计算机的主存容量不断增大，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主存的利用率。此外，有效的存储管理也是多道程序设计系统的关键支撑。具体地说，存储管理包含以下一些功能。

例如，Linux的Kernel。

（1）分配和去配

进程可请求对主存区的独占式使用，主存区的请求和释放即主存空间的分配和去配操作由存储管理完成。

（2）抽象和映射

每个进程是独自的4G虚拟地址空间。

主存储器被抽象，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，或者把主存储器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用逻辑地址来映射到物理主存单元，实现地址的转换。

（3）隔离和共享

系统负责隔离已分配给进程的主存区，互不干扰免遭破坏，确保进程对存储单元的独占式使用，以实现存储保护功能；系统允许多个进程共享主存区，在这种情况下，超越隔离机制并授权进程允许共享访问，达到既能提高主存利用率又能共享主存某区内信息的目的。

（4）存储扩充

物理主存容量不应限制应用程序的大小，主存和辅助存储器被抽象为虚拟主存，允许用户的

保护（Protection）是内存管理系统最主要的功能之一。

空间局部性：Spatial Locality ：旁边的内存也会被访问。
时间局部性：被访问的地方会很快被再被访问

虚拟地址空间大于主存物理地址空间，存储管理自动在不同的存储层次中移动信息。

本章在介绍计算机存储器的层次之后，先后分析连续存储管理方法、页式和段式存储管理方法，再讨论虚拟存储管理系统，最后介绍 Linux 虚拟存储管理及 Windows 2003 虚拟存储管理。

4.1 存 储 器

http://en.wikipedia.org/wiki/Locality_of_reference

4.1.1 存储器的层次

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，获得较好的性能/价格比。计算机系统的存储器层次结构分为寄存器、高速缓存、主存储器、磁盘、磁带等 5 层。如图 4.1 所示，存储介质的访问速度由下而上越来越快，价格也越来越高。其中，寄存器、高速缓存和主存储器均属于操作系统存储管理的管辖范畴，掉电后它们所存储的信息不复存在；磁盘和磁带属于设备管理的管辖对象，它们所存储的信息将被持久性保存。

图 4.1 计算机系统存储器层次结构

可执行程序必须被保存在主存储器中，与设备相交换的信息也依托于主存地址空间。由于处理器在执行指令时的主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的执行。

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，或用做地址寄存器，或用做变址寄存器，以加快地址的转换速度。

高速缓存的容量较寄存器稍大，其访问速度快于主存。利用高速缓存来存放主存中经常访问的一些信息，以提高程序执行速度。目前，计算机的典型配置为：CPU 寄存器 1 KB，存取周期 1 ns；高速缓存 1 MB，存取周期 2 ns；主存储器 512 MB，存取周期 10 ns；磁盘 80 GB，存取周期 10 ms；磁带 100 GB，存取周期 100 s，这样的机器足可用于中小型科研项目的开发。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。

由于程序在执行和处理数据时往往存在顺序性和局部性，执行时并不需要将其全部调入主

Locality Principle
空间局部性：Spatial Locality
时间局部性：Temporal Locality

Cache有L1， L2和L3 Cache

编译和链接之后的程序都是基于虚拟地址，或者逻辑地址。在运行的时候，通过加载，地址转换来确定具体的物理地址。

存，仅调入当前使用的一部分，其他部分待需要时再逐步调入。这样，计算机系统为了容纳更多的作业，或为了处理更大批量的数据，可在磁盘上建立磁盘高速缓存以扩充主存储器的存储空间，计算程序和所处理的数据可装入磁盘高速缓存，操作系统自动实现主存储器和磁盘高速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主存容量大得多的存储空间。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

大多数应用程序、操作系统和实用程序都用高级程序设计语言或汇编语言编写，所编写的程序称为源程序，源程序中的符号名集合所限定的空间称为程序名字空间。源程序是不能被计算机直接运行的，需要通过如图 4.2 所示的 3 个阶段处理后才能装入主存运行。

图 4.2 程序的编译、链接、装入和执行

Text，Data，BSS，Stack

1. 编译

源程序经过编译程序(compiler)或汇编程序(assembly)的处理来获得目标代码(也称目标模块)。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，在 C 程序设计模型中，至少分为 3 个程序模块：文本段、数据段和堆栈段。由于模块包含外部引用，即指向其他模块中的数据或指令的操作数地址，或包含对库函数的引用，编译程序负责记录引用的发生位置，编译或汇编的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。

初始化的全局变量，例如，int x = 0；x会放在DATA段。未初始化的全局变量，例如，int y；y会是在BSS。DATA占据可执行文件的大小；BSS不会。

程序缺省的地址都是从0开始。但可以通过链接的选项指定start address。

2. 链接

链接程序(linker)的作用是把多个目标模块链接成一个完整的可重定位程序(其中包括应用程序要调用的标准库函数、所引用的其他模块中的子程序),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。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 函数的主存位置就是调用 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;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。

可重定位目标程序又称装载代码模块,它存放于磁盘中,由于程序在主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。

3. 装入

在加载一个装载代码模块之前,存储管理程序总会分配一块实际主存区给进程,装入程序(loader)根据指定的主存区首地址,再次修改和调整被装载模块中的逻辑地址,将逻辑地址绑定到物理地址,使之成为可执行二进制代码。这样,就可用逻辑地址来引用分配到的主存物理块内相应的物理地址,将再次修改和调整的装载代码模块复制到指定主存区中,以便进程在物理地址空间中执行。

磁盘中的装载代码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;逻辑地址空间也可以是二维的,这时整个程序被分为若干段,每段都有不同的段号,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通常不可能同原来的逻辑地址一致。物理主存储器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。需要注意的是,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主存容量。把逻辑地址转换(绑定)为物理地址的过程称为地址重定位、地址映射或地址转换,有以下两种方式。

1. 静态地址重定位

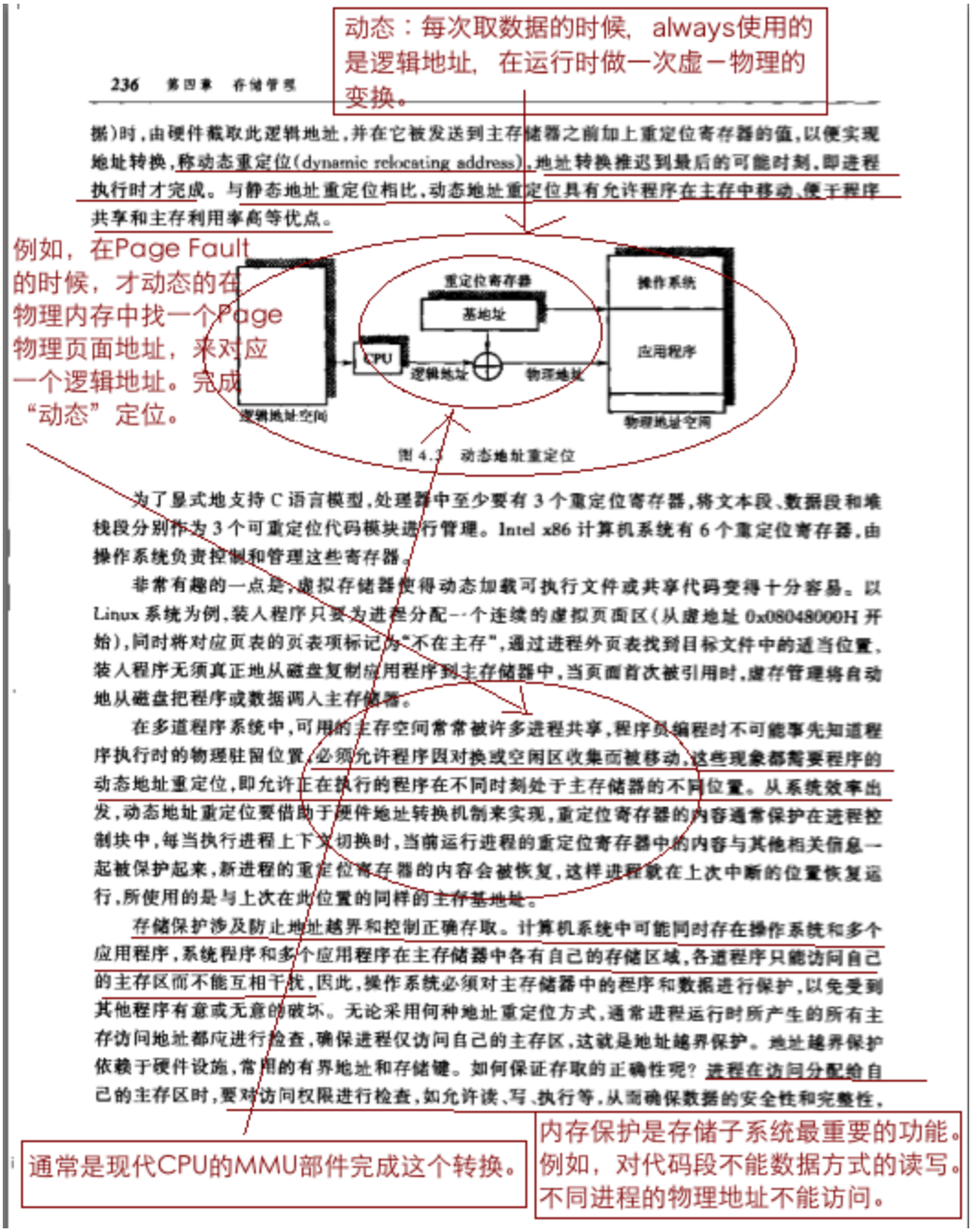
基本上不太用 静态地址重定位。

由装入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主存指定区域,其中的所有逻辑地址修改成主存物理地址,称静态重定位(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

由装入程序实现装载代码模块的加载,把它装入分配给进程的主存指定区域,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,程序主存起始地址被置入硬件专用寄存器——重定位寄存器,如图 4.3 所示。程序执行过程中,每当 CPU 引用主存地址(访问程序和数

ELF格式的文件还含有许多控制信息；真正调入内存中执行的是BIN格式的代码段，数据段（含BSS）。



[注释]： 掌握现代操作系统内存管理系统时，可以把握几个基本知识点。1. 了解ELF展开后的格式。TXT，DATA / BSS / Stack / Heap的关系。2. 任何一个CPU的load / store操作都是基于逻辑（或者说虚拟地址），通过MMU转换一次，成为物理地址，完成“动态”定位。

堆（Heap）管理就是连续存储区域的分配问题。

防止有意或无意的误操作而破坏主存信息，这就是信息存取保护。

4.2 连续存储空间管理

Contiguous Memory Block

4.2.1 固定分区存储管理

固定分区存储管理的基本思想是：主存空间被划分成数目固定不变的分区，各分区的大小不等，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的最简单的存储管理技术。固定分区(fixed partition)存储管理又称为定长分区或静态分区模式，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。

为了说明各分区的分配和使用情况，需要设置一张"主存分配表"，记录主存储器中划分的分区及其使用情况。主存分配表指出各分区的起始地址和长度，"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。主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 4.4 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主存区时，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主存分割成若干连续区域，分割时各分区在主存分配表中可按地址顺序排列，那么，其主存分配算法就十分简单。

基本上不存在这种内存的使用模式了。除非是通信系统。在通信系统里，为了简单，快速，会把一些内存显示的划分，专门使用。

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

图 4.4 固定分区存储管理的主存分配表

固定分区的一项任务是何时及如何把主存空间划分成分区。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，系统操作员根据当天的作业情况把主存储器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

内存区域的显示（Explicitely）的划分，
比较适合专用系统；不适合通用的操作系统，例如，桌面，服务器等。

时，就装入队首进程执行。这样做的好处是可使装入分区的未用空间最小，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。

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主存空间利用率低的问题。其缺点是：首先，由于预先已规定分区的大小，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，这样不但加重用户的负担，而且极不方便；其次，主存空间利用率不高，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 的主存空间，但它们却占用 16 KB 和 32 KB 的区域，共有 20 KB 的主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主存空间，采用固定分区是相当困难的；最后，因为分区的数目是在系统初启时确定的，这就会限制多道运行的程序的个数，特别不适应分时系统交互型用户及主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. 可变分区主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，按照作业的大小来划分分区，但划分的时间、大小、位置都是动态的。系统把作业装入主存时，根据其所需要的主存容量查看是否有足够的空间，若有，则按需分割一个分区分配给此作业；若无，则令此作业等待主存资源。由于分区的大小是按照作业的实际需求量而定的，且分区的数目也是可变的，所以，可变分区能够克服固定分区中的主存资源的浪费，有利于多道程序设计，提高主存资源的利用率。使用可变分区存储管理的一个例子是 IBM 操作系统 OS/MVT(Multiprogramming with a Variable Number of Tasks)。

在可变分区模式下，在系统初启且用户作业尚未装入主存储器之前，整个用户区是一个大空闲分区，随着作业的装入和撤离，主存空间被分成许多分区，有的分区被占用，而有的分区是空闲的。主存中分区的数目和大小随着作业的执行而不断改变，为了方便主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变为"空"，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 4.5 所示，同时应修改主存分配表。

内存分配最需要注意的是：在动态分配过程中，产生的内存碎片（Fragment）的问题。结果是导致了明明有许多内存，但没有足够大的内存块（block）来满足任务task的需求。

关于 碎片： http://en.wikipedia.org/wiki/Fragmentation_(computing)

图 4.5 可变分区回收情况

由于分区的数目不定，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分区链接起来，每个主存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) 最先适应分配算法

最先适应(first fit)分配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，为进程分配主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，有利于大作业的装入；但这样做会使主存储器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的登记项。

(2) 下次适应分配算法

下次适应(next fit)分配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主存储器一端。

(3) 最优适应分配算法

最优适应(best fit)分配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，查找时总是从最小的一个空闲区开始，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。此算法的主存利用率好，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。

可能导致 Internal Frag碎片多。分配的内存块比实际需要的大很多。

可能导致 External Frag 碎片多。一段时间后，系统缺乏连续的大内存块。

(4) 最坏适应分配算法

最坏适应(worst fit)分配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) 快速适应分配算法

快速适应(quick fit)分配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,只要按用户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,但归还主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,操作系统可把分区的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 CDC6600 便采用这种方案。

C 语言程序会被编译成至少 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 平台提供专用的存放段基址的寄存器,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其他地址。在有 N 个重定位寄存器的机器中,允许每个进程获得 N 个不同的主存段,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的主存区存放一套,从而主存利用率低。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

http://en.wikipedia.org/wiki/Buddy_memory_allocation

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 4.6 所示，让进程的共享部分取相同的基址/限长值。

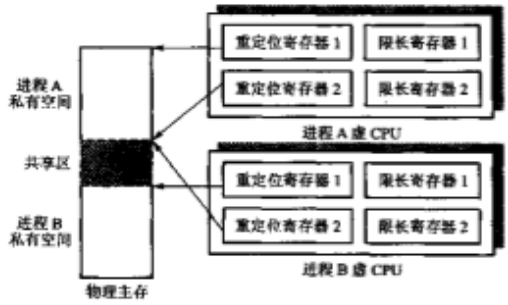


图 4.6 多对重定位寄存器支持主存共享

4.2.3 伙伴系统 Buddy System

1. 伙伴系统原理

伙伴系统(Knuth,1973 年)又称 buddy 算法，是一种固定分区和可变分区折中的主存管理算法，其基本原理是：任何尺寸为 2^i 的空闲块都可被分为两个尺寸为 2^{i-1} 的空闲块，这两个空闲块称作伙伴，它们可以被合并成尺寸为 2^i 的原先的空闲块。

伙伴系统通常用在以固定尺寸为单位分配的系统中，单位为字长或固定字长序列。假设主存储器包含 2^m 个分配单位，则最大空闲块为 2^m 个单位，小一些的为 2^{m-1} 个单位，依此类推，最小空闲块尺寸为 2^0，即一个单位。为了实现这一算法，需要位图和空闲链表作为辅助工具，位图用来跟踪主存块的使用情况，空闲链表用做维护主存中尚未使用的主存块。

伙伴系统维护数组 free 来记录空闲块，它包含 $m+1$ 个列表头，每种空闲块的尺寸有一个，即 free[i]把所有尺寸为 2^i 的空闲块用双向指针链接在一起。系统处理 n 个存储单位请求的步骤如下：查找满足 $n \leqslant 2^i$ 的最小空闲块尺寸($i=0,1,\cdots$)，若此尺寸空闲块列表非空，则从列表中移走此空闲块，将其分配给请求进程，处理结束；否则，再检查 2^{i+1} 的空闲块尺寸，循环直至找到一个非空列表。假设找到 free[$i+1$]，把此列表的第一个空闲块取出，分成大小减半的两个空闲块，分别称为 L 和 R。空闲块 R 被移入列表 free[i]中，如果 L 的尺寸是请求的可能的最小尺寸，就把空闲块 L 分配给请求进程；如果 L 仍然太大，重复执行等分操作，即把 L 再拆分为大小减半的两个空闲块，其中一个空闲块移入尺寸更小一级的空闲块列表中，再考虑分配另一半。

当已分配尺寸为 2^i 的主存块被释放时，要进行相反的操作，系统检查被释放块的伙伴是否被占用，如果被占用，则新空闲块被移入尺寸为 2^i 的空闲块列表；否则，从 2^i 尺寸的空闲块列表

相关算法可以参阅数据结构课程。

[注释] ： 1. 连续地址空间管理，通常也可以理解为“堆管理” - Heap。也可以逻辑地址的堆管理，例如进程的Heap。也可以是一块连续的物理地址空间，例如，Linux Kernel为Slab分配器提供连续物理页面的内存。2. 内存管理要注意两种碎片：External Fragment和Internal Fragment

满足80K的最小的2的幂是2^7＝128K。所以，分拆大块为512，128

中移出伙伴，两个空闲块合并成一个尺寸为2^{i+1}的空闲块；重复这种操作，直到生成可能的最大空闲块。为了提高操作速度，空闲块合并是通过搜索位图快速找到被释放块的对应位，并检查伙伴块的对应位是否为 0 来实现的，回收过程是递归的，一直进行到没有空闲伙伴为止。

如图 4.7 所示的例子用来说明伙伴算法的申请和归还原理。初始时 1 MB 的链表仅有一个表项包含 1 MB 的空闲块，其他链表均为空。当有作业被调入主存储器时，假如所需主存空间如下：A 为 80 KB、B 为 50 KB、C 为 100 KB 和 D 为 60 KB，则伙伴算法的申请和归还情况列于图 4.7 中。伙伴系统分配和合并操作的速度快，其缺点是：当所申请的主存空间大小非 2 的整数次幂时，内部碎片较大。

128 KB	256 KB		384 KB	512 KB	640 KB	768 KB	896 KB	1 MB
A	128 KB		256 KB		512 KB			
A	B	64 KB	256 KB		512 KB			
A	B	64 KB	C	128 KB	512 KB			
128 KB	B	64 KB	C	128 KB	512 KB			
128 KB	B	D	C	128 KB	512 KB			
128 KB	64 KB	D	C	128 KB	512 KB			
256 KB			C	128 KB	512 KB			
1 024 KB								

满足A
满足B
满足C

做一次Merge。Buddy合并。

图 4.7 伙伴算法的申请和归还原理

2. Linux 伙伴系统

Linux 采用请求分页存储管理，分配时并不需要连续分布的页框，然而在 I/O 操作及特殊操作时，会绕过分页机制，要求获得连续分布的页框，直接实现磁盘与主存间的数据传送。为此，引进伙伴系统，使用 buddy 算法及下面的数据结构来实现。

(1) 以 page 结构为数组元素的 mem_map[ ]数组。

(2) 以 free_area_struct 结构为数组元素的 free_area 数组：

```
struct free_area_struct {
    struct page * next;
    struct page * prev;
    unsigned int * map;
};
static free_area_struct free_area[NR_MEM_LISTS];
```

此数组记录空闲主存页框，共 11 个元素(NR_MEM_LISTS 默认值)，维护 11 个不同空闲页框数的链表，大小从 $2^0=1$ 到 $2^{10}=1\,024$，第 i 个元素代表 mem_map 数组中第 i 组空闲块链表头。

这个时候不合并。不是左右的Buddy。128 和64不是buddy。

基于页面的分配机制导致了许多宝贵的内存块被浪费。

(3) 位图数组(bitmap)共 11 个，每个空闲页框块数的链表对应一张，用二进制数表示主存页框的使用情况，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，如果其中的位置 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 2^i 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 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_ok()。

3. Linux 基于伙伴系统的 slab 分配器

伙伴系统以页框为基本分配单位，在很多情况下，内核所需要的主存量远远小于页框大小，如 inode、vma、task_struct 等。为了更经济地使用内核主存资源，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的空间。这样做的好处是：充分利用主存储器，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，从而提高效率。

slab 分配器的结构如图 4.8 所示，主存中建立多个 cache，每个 cache 有一个 slab 链，每个 slab 由一个或最多 32 个物理连续的页框组成，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，等等。

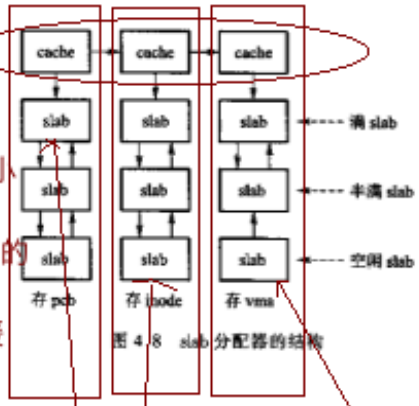


图 4.8 slab 分配器的结构

slab机制：
1. 建立在基于页面的伙伴分配器之上。
2. 一个cache／slab就是一个内存池。
3. 在cache／slab基础上，可以定义任何大小的内存对象，例如，小粒度的数据结构。
4. 在3的基础上，可以缓存一些经常需要使用的数据结构，例如，task struct等。有数据表明，初始化复杂数据结构的时候很大。通过缓存，不放回堆（Heap）里，可以省下许多时间。

一个slab可以有一个page，或者多个page。

Cache链。从而系统可以在cache中的size参数做Best Fit算法，选择用哪个cache／slab

从图 4.8 中可见，每个 slab 均处于三种状态之一：满的(所有对象都已被分配，排在上面)、半满的(尚有空闲对象，排在中间)和空闲的(所有对象均空闲，排在下面)。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，这种策略能减少主存碎片。下面来看 task_struct 的例子，内核用一个全局变量存放指向 task_struct 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，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。

如果直接用伙伴分配机制，因为粒度是Page（4K size），就会造成许多浪费。

除了这些特定对象的缓冲存储器之外，Linux 系统还提供 13 种通用缓存，其存储对象的大小分别为 32 B、64 B、128 B、256 B、512 B、1 KB、2 KB、4 KB、8 KB、16 KB、32 KB、64 KB 和 128 KB，用来满足特定对象之外的普通主存空间需求，单位大小呈级数增长，保证内部碎片率不超过 50%。

slab 分配器的主要操作如下。

(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) kmem_cache_grow()与 kmem_cache_reap()函数：kmem_cache_create()函数只是建立所需的专用缓冲区队列的基础设施，所形成的 slab 是一个空队列。具体 slab 的创建则要等到需要分配缓冲区却发现并无空闲的缓冲区可供分配时，通过 kmem_cache_grow()来进行，它向伙伴系统申请空间；kmem_cache_reap()用于减少 slab。

(4) 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。

(5) kmem_getpages()与 kmem_freepages()函数：slab 与页框级分配器的接口。

slab向内存管理要物理页面

分配的内存对象是create这个cache时指定的大小，例如，可以时32bytes。

4.2.4 主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主存区域，由于进程不断地装入和撤销，导致主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，不但浪费主存资源，还会限制进入主存的进程数目。

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主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，也叫做主存紧凑。第一种方法是把所有当前占用的分区移动到主存的一端；第二种方法是把占用分区移动到主存的一端，但当产生足够大小的空闲区时就停止移动。

移动操作需要把主存中的进程"搬家"，即读出每个字并写回主存，凡涉及地址的信息均应修改，如基址寄存器、地址指针等。移动分配的示例如图 4.9 所示。移动虽然可以汇集主存空闲区，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，由于块设备在与主存储器交换信息时，通道或 DMA 总是按确定的主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。

DMA必须通过绝对的，物理地址来运作。DMA引擎不会通过CPU／MMU的虚存转换机制／部件。

基本上不存在这样的系统和机制了。不太有必要掌握。知道就可以了。

在虚拟地址转换的支持下，每次swap的时候，把进程的各种状态保护好（全局变量，指令执行的当前位置，各种数据和控制寄存器的当前值）。这样就可以之后恢复执行了。

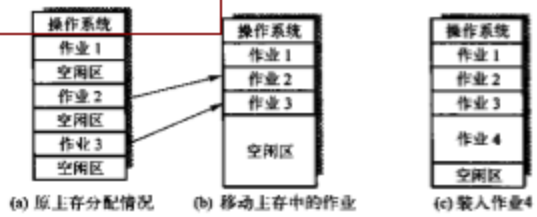


图 4.9 移动分配示例

步骤 3：移动主存的相关分区信息；修改主存分配表的有关项；修改被移动者的基址寄存器等信息；

步骤 4：分配 x KB 主存；修改主存分配表的有关项；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。

移动操作也为进程运行过程中动态扩充主存空间提供了方便。当进程在执行过程中要求增加主存分配区时，只需适当移动邻近的占用分区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，以解决主存容量不足的问题，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，此时选择其中的一个进程，将其暂时移出主存，腾出空间给其他进程使用，同时把磁盘中的某个进程换入主存，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主存映像对换到磁盘；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，一旦主存可用，立即把此进程对换回主存。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主存区域并重置重定位寄存器的值。

为了有效地实施对换，首先，要解决选择哪个进程换出。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将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主存，其未修改部分(如代码)在主存与磁盘中始终保持一致，这些信息不必保存，当进程换回主存时，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的角度考虑，开辟一块特殊的磁盘区域作为对换空间，它包含连续的柱面和磁道，可通过低层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主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，对换进主存的进程其主存位置未必还在换出

代码执行的都是编译和链接之后的逻辑地址。具体这些逻辑地址在物理内存的什么地方，动态的通过《虚－－实》映射来实现。

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 $2k$ 个磁盘块输入/输出的时间，再加上进程重新请求主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，只需更少的主存访问。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、进程动态扩充主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、在主存已逗留的时间等。Windows对换空闲线程负责把进程从主存挪出，此线程每隔4 s被唤醒，查找已空闲一定时间(秒级)的线程，将其核心栈换出，当一个进程的所有线程都被换出时，余下部分(包括线程共享的代码和数据)也被换出，那么，这意味着整个进程被对换出去。

很少使用了。早期的许多病毒代码是通过overlay技术。现代OS的代码段通常不让读写，做保护。不能overlay。

3. 覆盖技术

移动和对换技术解决因其他程序存在而导致主存区不足的问题，这种主存短缺只是暂时的；如果程序的长度超出物理主存总和，或超出固定分区的大小，则出现主存永久性短缺，大程序无法运行，前述两种方法无能为力，解决方法之一是采用覆盖(overlaying)技术。覆盖是指程序执行过程中程序的不同模块在主存中相互替代，以达到小主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，当进程装入运行时，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主存的是哪些程序段，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主存管理工作转给程序员，他们必须根据可用物理主存空间来设计和编写程序。此外，同时运行的代码量超出主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 分页存储管理的基本原理

用分区方式管理存储器，每道程序要求占用主存的一个或多个连续存储区域，导致主存中产生"碎片"。有时为了接纳新作业，往往需要移动已在主存的信息，这样不仅不方便，而且处理器的开销太大。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，又可充分利用主存空间，消除动态分区法中的"碎片"问题，从而提高主存空间的利用率。分页存储管理涉及的基本概念如下。

CPU指令执行是基于虚拟地址（逻辑地址）是可以通过不连续的物理页面，"营造"成一个连续的逻辑空间的重要保障机制。逻辑地址必须是连续的；物理地址可以是Page和Page散开的。

Paging Based Memory Management

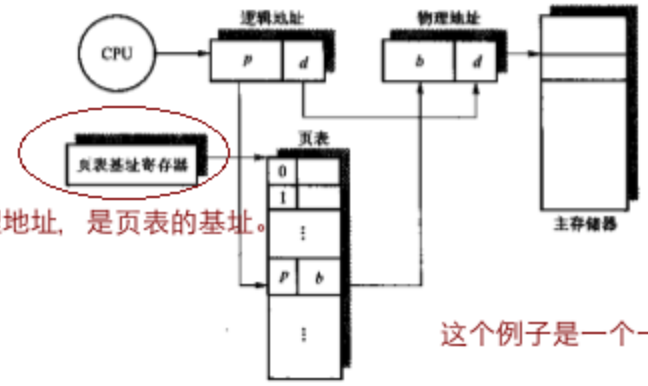
[注释]： 1. 基于4K大小的页面（Page）的分配粒度太大，Linux Kernel的Slab机制就是为了实现细粒度内存频繁分配和释放的一种memory pool的机制。2. 通过架在基于Page的Buddy算法内存管理之上，Slab可以不需要频繁的把常用的数据结构来来回回放回HEAP里，从而提高了效率。

TLB是＋＋页表＋＋在CPU内部的Cache（缓存）。不要和指令和数据缓冲搞混。

物理地址＝页框号×块长＋页内位移

计算出欲访问的主存单元。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的物理地址进行存取。

如图 4.10 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，就组成访问主存储器的绝对地址。



指向一个物理地址，是页表的基址。

这个例子是一个一级页表。

图 4.10 分页存储管理的地址转换

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。

4.3.2 快表 TLB

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；页表也可存放在主存中，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，至少访问主存两次：一次访问页表，另一次根据物理地址访问指令或存取数据，这将降低运算速度，比通常执行指令时的速度慢一半。

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 Lookaside Buffer，TLB)或快表，它是分页存储管理的重要组成部分。快表的存取时间远小于主存，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的页框号。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，再查主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

进程切换的时候，要切换页表基址寄存器。不通的进程有不同的页表。

一个术语：逻辑页面叫Page（页面）。物理页面叫Frame（页框）。地址映射其实就是做Page到Frame的转换。

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，把主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，每个区是一个物理块或页框，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

前面部分表示地址所在页面的编号，后面部分表示页内位移。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 2^{20} 个页面。

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，从而确定主存分块大小。进程在主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，进程主存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. 页表和地址转换

每个进程有自己的页表。对Linux而言，就是每个进程有自己的0-3G的虚拟地址的Page Table

在进行存储分配时，以页框为单位，进程的信息有多少页，那么，把它装入主存时就分配多少块，虽然进程的逻辑地址分成页面后是连续的，但被装入后的相应物理块（页框）未必紧邻，即进程的信息按页面分散存放在主存不相邻的页框中，那么，当进程的程序和数据被分散存放在主存中后，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）。页表是操作系统为进程建立的，是程序页面和主存对应页框的对照表，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，从数学的角度而言，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页框域。为了减少系统开销，不用硬件而是在主存中开辟存储区以存放进程页表，系统另设置专用的硬件——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。系统应为主存中的进程进行存储分配，并建立页表，指出逻辑地址页号与主存页框号之间的对应关系，页表的长度随进程大小而定。

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，运行时借助于硬件的地址转换机构，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：

物理页面是散布的。但通过物理到虚拟地址的映射。进程的访问还是做连续的（虚拟）地址空间上运行。否则，乱了。例如，数组必须是一个连续的内存块。。

TLB命中页表项

简单的策略是"先进先出"，总是淘汰最先登记的页。

通过快表实现主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的主存访问都通过快表实现，几乎不用页表；反之，当进程访问遍布主存页面的跳跃性地址时，命中率近乎为 0，这意味着每次访问主存都要使用页表。采用快表后，地址转换时间大大下降，假定访问主存的时间为 100 ns，访问快表的时间为 20 ns，快表为 32 个单元时的查找命中率若为 90%，主存数据要先存入快表，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：

$$(100+20)\times 90\% + (100+100+20)\times(1-90\%) = 130 \text{ ns}$$

比两次访问主存的时间 200 ns 缩短 35%。

需要注意的是，快表和高速缓存是不同的，两者在一个系统中可同时使用，前者记录最近转换的页号及页框号，而后者保存最近实际访问的指令或数据的副本。

4.3.3 分页存储空间的分配和去配

分页存储管理中，系统要建立一张主存物理块表，用来记录页框的状态，管理主存物理块的分配，所包含的信息有主存总块数、哪些为空闲块、哪些块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 4.11 所示。

有时需要连续的物理页，例如DMA操作。

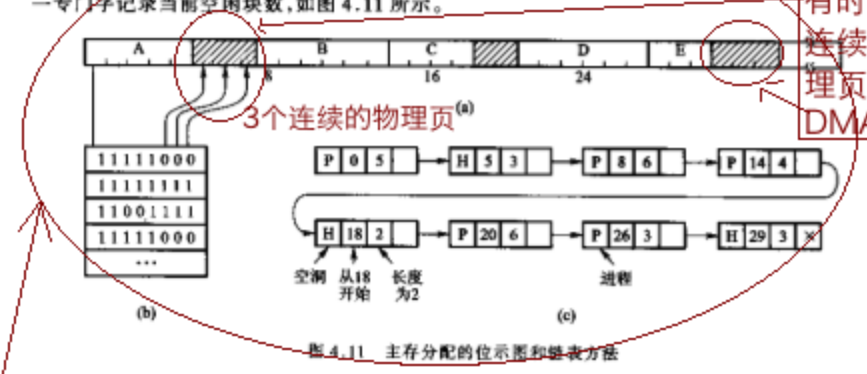


图 4.11 主存分配的位示图和链表方法

分页存储管理页框分配算法如下：进行主存分配时，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。进程执行结束归还主存时，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 4.11(c)是主存分配的链表方法，表中各项都包含以下内容：是进程占用区(P)还是空闲

当出现"缺页"（Page Fault）的异常处理时，操作系统要试图从 物理内存块中分配相应的页框。

完成 虚－实 的mapping

不同经常的虚拟地址Va, Vb, 可以在各自的Page Table里指向通一个物理地址。

区(H)、起始地址、长度和指向下一表项的指针。在本例中,链表是按照地址从小到大排序的,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 分页存储空间的页面共享和保护

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,这些共享信息在主存中保留副本,分页存储管理能实现多个进程共享程序和数据,共享页面信息可大大提高主存空间的利用率。

实现页面共享,必须区分数据共享和程序共享。实现数据共享时,允许不同进程对共享的数据页用不同的页号,只要让各自页表中的有关表项指向共享的数据页框;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,进程依赖于这些地址才能执行,所以,不同进程正确执行共享代码页面,必须为它们在所有逻辑地址空间中指定同样的页号。假定有一个被共享的编辑程序 EDIT,此程序一定是可再入的,假设其中含有指向其自身的转移指令 branch n,64,其中,n 是共享代码的页号,64 是页内位移。问题是 n 要依赖于执行 EDIT 的进程,当进程 P_1 执行时,目标地址必须为(n_1,64),当进程 P_2 执行时,目标地址必须为(n_2,64)。因为一个存储器单元在任何时刻只能有一个值,所以,页号 n_1 和 n_2 必须相同,即共享页面在进程 P_1 和 P_2 的页表中必须被分配同样的记录,因此,对共享的程序必须规定统一的页号,这样在需要时才能转换成相同的物理地址。

实现信息共享必须解决共享信息的保护问题。通常的做法是在页表中增加标志位,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 System/370 系列操作系统的存储保护键保护机制。

2. 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,这样就增加了对主存容量的要求。如果应用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,影响系统的效率。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。

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的主存区域,并在主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 UNIX/Linux 系统中,共享库的共享代码通常用后缀 .so 来表示,共享库在任何给定的文件系统中,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

通过多级页表，可以做到，不需要的时候，不分配来装载（hold）相应的二级页表的物理页面Frame。另外，不同的二级页表的Page可以是离散的分布在物理内存中。不需要是连续的了。页目录都是通过指针来索引跟踪了。

一级页表使得系统必须开辟一个大的连续物理内存块来做Page Table。开销很大，而且不灵活。

4.3.5 多级页表

现代计算机普遍支持 $2^{32}-2^{64}$ B 容量的逻辑地址空间，采用分页存储管理时，页表相当大。以 Windows 为例，其运行的 Intel x86 平台具有 32 位地址，规定页面 4 KB(2^{12})，那么，4 GB(2^{32})的逻辑地址空间由 1 M(2^{20})个页组成，若每个页表项占用 4 B，则需要占用 4 MB(2^{22})连续主存空间来存放页表，这还是一个进程的地址空间。对于地址空间为 64 位的系统而言，问题将变得更加复杂。为此，页表和页面一样也要进行分页，这就形成多级页表的概念，具体做法是：把整个页表分割成许多小页表，每个小页表称为页表页，其大小与页框长度相同，于是，每个页表页含有若干页表表项。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，其表项指出页表页的起始地址。系统为每个进程建立一张页目录表，其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之间的对应关系。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 3 个部分组成：页目录位移、页表页位移和页内位移。

如图 4.13 所示是二级页表实现逻辑地址到物理地址转换的过程，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主存起始地址，加上“页目录位移”作为索引，可找到页表页在主存的起始地址，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。

图 4.13 二级页表地址转换过程

上述方法能解决分散存放页表页的问题，并未解决页表页如何占用主存空间的问题，解决方法如下：进程运行涉及页面的页表页应存放在主存，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，指示对应的页表页是否已调入主存，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”中断信号，请求操

基于页表的虚－实转换是一个“算法”，是一种mapping。把相应的bits拿来做索引（index）

通过context（上下文）ID，配合逻辑地址，这样就可以去掉相同的VA虚拟地址的二义性。例如，每个进程都可以有相同的一个VA，但配上context ID索引，歧义就没有了。

作系统将页表页调入主存。

二级页表地址转换需 3 次访问主存，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。

SUN 微系统公司计算机使用 SPARC 芯片，采用如图 4.14 所示的三级分页结构。为了避免进程切换时重新装入页目录表指针，硬件支持多达 4 096 个上下文，每个进程一个上下文，当新进程装入主存时，操作系统分配一个上下文号，进程保持这个上下文号直到终止。当 CPU 访问主存时，上下文号和逻辑地址一并送入地址转换机构，使用上下文号作为上下文表的索引，找到顶级页目录，然后，使用逻辑地址中的索引值查找下一级页目录表项，直至找到访问页面，形成物理地址。在分页系统中，为了加快地址转换的过程，都会使用快表。

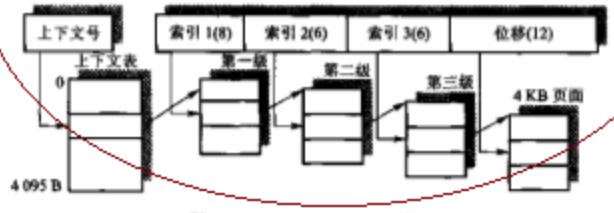


图 4.14　SPARC 三级分页结构

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。Motorola 68030 系列计算机为操作系统设计者提供选择，页表级数和每级的位数可以编程控制。

通过一个物理基址寄存器作为出发点，层层定位Page Table。各个索引其实就是各个页面数组的下标。

4.3.6　反置页表

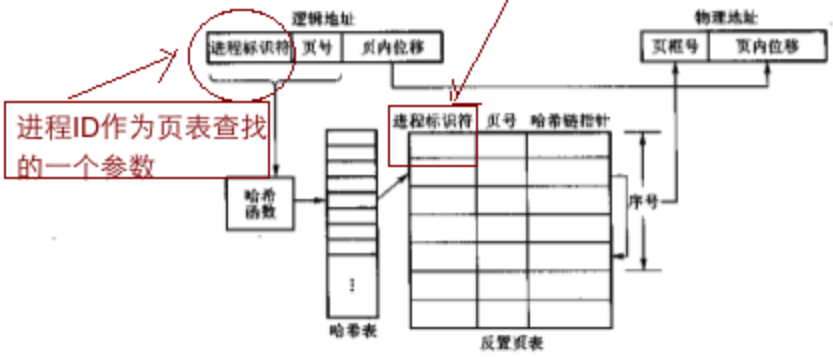
计算机逻辑地址空间越来越大，页表所占用的主存空间也越来越多，页表尺寸与虚地址空间呈正比增长。为了减少主存空间开销，不得不使用多级页表，但也有些操作系统，如 IBM AS/400、PowerPC、UltraSPARC 和 IA－64 体系结构中均采用反置页表（Inverted Page Table，IPT），它的主要优点是只需为所有进程维护一张表。此表为主存中的每个物理块建立一个 IPT 表项并按照块号进行排序，其表项包含：在此页框中页面的页号、页面所属进程的标识符和哈希链指针，用来完成逻辑地址到物理地址的转换。与此相适应，逻辑地址由进程标识符、页号和页内位移 3 个部分组成。

如图 4.15 所示是反置页表及其地址转换的过程：需要访问主存地址时，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

因为不同的进程都可以有相同的虚拟地址，多级页表需要每个进程有自己的Page Table。IPT是通过把进程ID考虑进去，从而只需要维护一个大的，但只是一个系统范围内的Page Table。

进程ID作为页表项的一个参数，供查找时做匹配。

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，说明此页不在主存，触发缺页中断，请求操作系统通过页表调入。



进程ID作为页表查找的一个参数

图 4.15 反置页表及其地址转换

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。

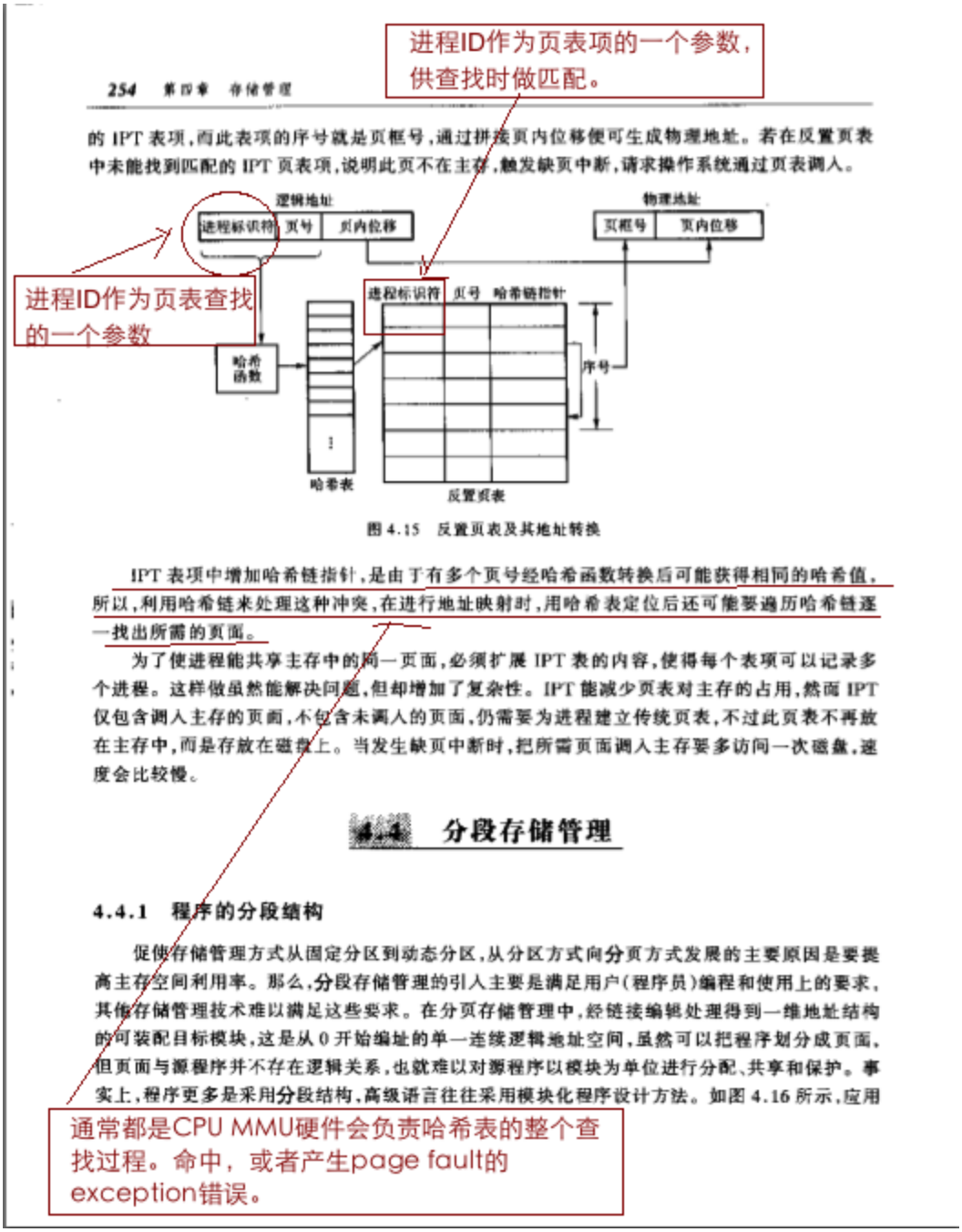
为了使进程能共享主存中的同一页面，必须扩展 IPT 表的内容，使得每个表项可以记录多个进程。这样做虽然能解决问题，但却增加了复杂性。IPT 能减少页表对主存的占用，然而 IPT 仅包含调入主存的页面，不包含未调入的页面，仍需要为进程建立传统页表，不过此页表不再放在主存中，而是存放在磁盘上。当发生缺页中断时，把所需页面调入主存要多访问一次磁盘，速度会比较慢。

4.4 分段存储管理

4.4.1 程序的分段结构

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主存空间利用率。那么，分段存储管理的引入主要是满足用户(程序员)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，经链接编辑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，高级语言往往采用模块化程序设计方法。如图 4.16 所示，应用

通常都是CPU MMU硬件会负责哈希表的整个查找过程。命中，或者产生page fault的exception错误。



[注释]：1. 每个进程都可以有相同的虚拟地址，例如Va。因此每个进程都必须有自己的页表，从而无歧义的完成虚实转换。2. 多级页表可以使得页表空间不需要连续物理块，例如，二级页表，可以Page by page的分配。3. 通过进程ID，可以避免全局页表项目的二义性。

分段存储管理基本上不太用了。算早期技术。可以略微了解就够了。

分段管理在早期x86中普及。分段是一个需要编程时，显示（Explicite）来控制和干涉的。例如，有多个代码段，数据段。通过段地址来把程序的不同模块加载在不同的内存部分。可以理解，一个程序可以变成N个模块。每个模块通过对应一个段来装填。

4.4 分段存储管理

4.4.1 程序的分段结构

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主存空间利用率。那么，分段存储管理的引入主要是满足用户（程序员）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，经链接编辑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，高级语言往往采用模块化程序设计方法。如图4.16所示，应用

Segment，段是一个编程可见的机制。段之间的访问类似远程调用的call，或者jump。

程序由若干程序段(模块)组成，例如，由主程序段、子程序段、数据段和工作区段组成，每段都从0开始编址，有各自的名字和长度，且实现不同的功能。

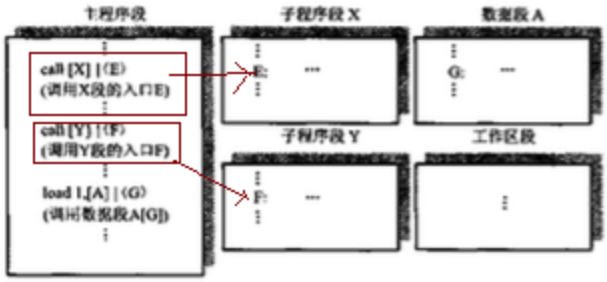


图 4.16 程序的分段结构

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，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。

4.4.2 分段存储管理的基本原理

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

！！！！

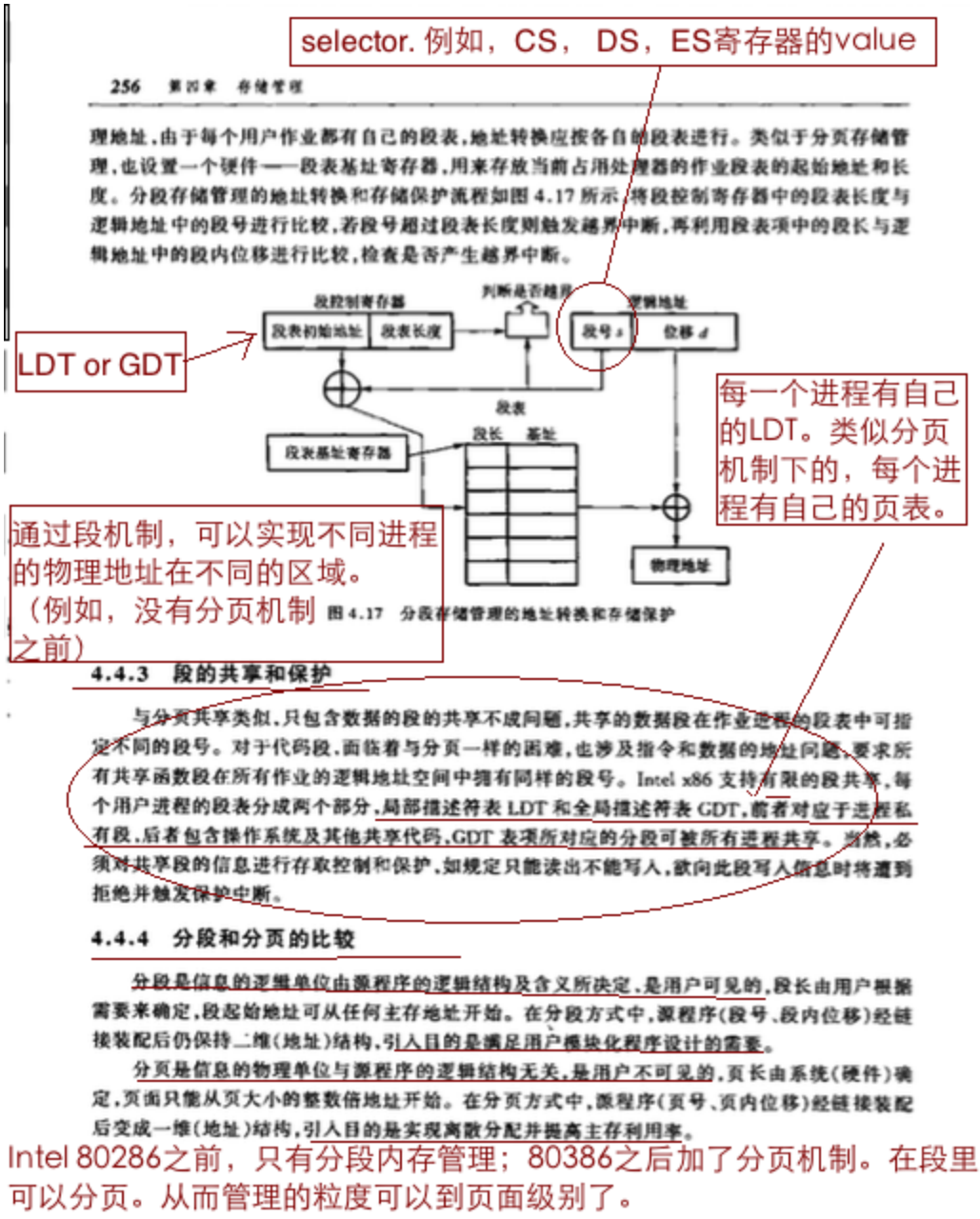
在分页存储管理中，页的划分，即逻辑地址划分为页号和页内位移，是用户不可见的，连续的地址空间将根据页面的大小自动分页；而在分段存储管理中，地址结构是用户可见的，用户知道逻辑地址如何划分为段和段内位移，在设计程序时，段的最大长度由地址结构规定，程序中所允许的最多段数会受到限制。例如，PDP-11/45 的段地址结构为：段号占 3 位，段内位移占 13 位，一个作业最多分为 8 段，各段长度可达 8 KB。

分段存储管理的实现基于可变分区存储管理的原理。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的主存空间，而各段之间不一定连续。在进行存储分配时，应为进入主存的作业建立段表，各段在主存中的情况可由段表来记录，它指出主存中各分段的段号、段起始地址和段长度。在撤销进程时，回收所占用的主存空间，并清除此进程的段表。

段表表项实际上起到基址/限长寄存器的作用，进程运行时通过段表可将逻辑地址转换成物

段之间通过远程调用指令来访问

分页管理是一个Implicite的管理。程序语言是不需要感知的。



[注释]： 分段管理与分页管理是区别和联系是：段是应用编程可感知的；页是应用不感知的，段是早期，例如intel 80286之前的内存管理；80386之后有了分页（Page）了。可以“基于分段的分页内存管理”。使得在段的基础上再加入页面机制，使得内存使用粒度更加小。

没有分页之前，Intel LDT的目的类似之后的Page Table。每个进程有自己的LDT从而管理自己的物理内存。全局的上通过GDT，各个进程之间共享。有了分页后，段基本上不用了，通过把seg selector都设为0，大家都在一个段上，从而，例如，Linux，都（只）用分页机制了。

附图是一个比较好段和页机制的关系图。在段做一次映射之后（通过寄存器索引做一次LDT和GDT查表），如果页机制打开的，查表出来的地址不被当作（not treated as）物理地址而是再做一次基于Page Table的查表。然后出来的数据才与offset合并为物理地址去内存取数据。

General facts

Paging translates linear addresses (what is left after segmentation translated logical addresses) into physical addresses (what actually goes go to RAM wires):

```
(logical) ------------------> (linear) ------------> (physical)
             segmentation                  paging
```

Paging is only available on protected mode. The use of paging protected mode is optional. Paging is on iff the `PG` bit of `cr0` is set.

保护模式下，如果打开页面机制，从段机制mapping出来的地址，还不当作物理地址，要再通过Page Table一次，才出物理地址。

One major difference between paging and segmentation is that:

- paging splits RAM into equal sized chunks called pages
- segmentation splits memory into chunks of arbitrary sizes

This is the main advantage of paging, since equal sized chunks make things more manageable.

segmentation是可见（操作）的。例如，CS，DS，ES等寄存器。Paging是不可见的。

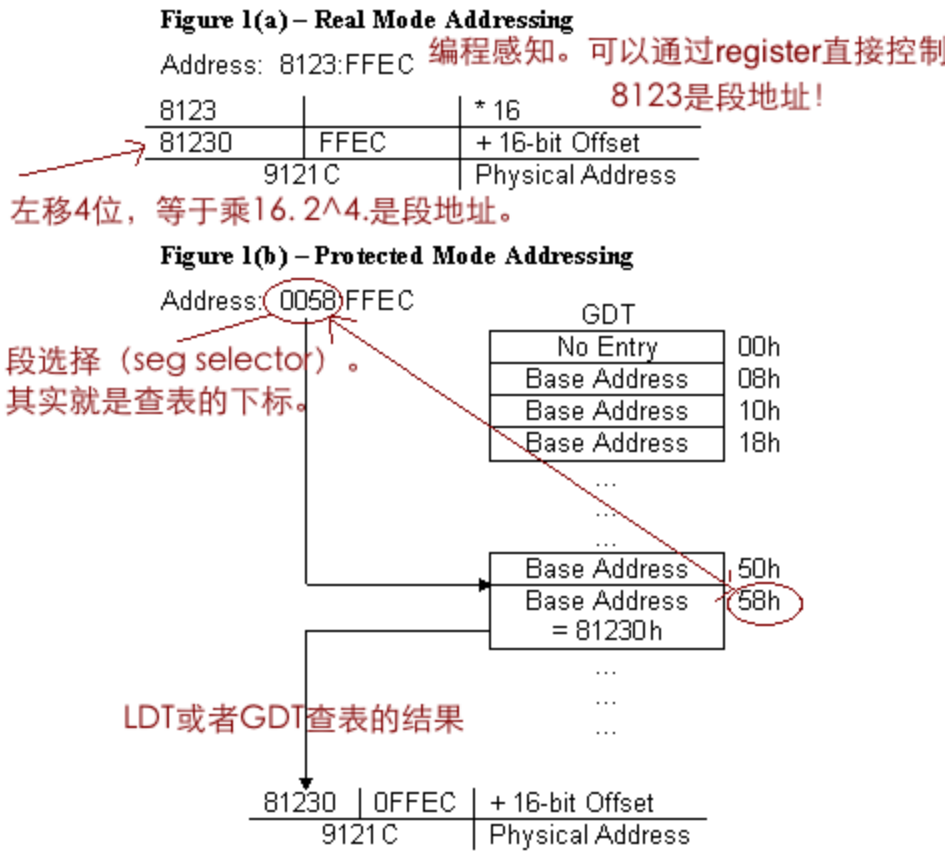
Application

Paging is used to implement processes virtual address spaces on modern OS. With virtual addresses the OS can fit two or more concurrent processes on a single RAM in a way that:

- both programs need to know nothing about the other
- the memory of both programs can grow and shrink as needed
- the switch between programs is very fast
- one program can never access the memory of another process

Paging historically came after segmentation, and largely replaced it for the implementation of virtual memory in modern OSs such as Linux since it is easier to manage the fixed sized chunks of memory of pages instead of variable length segments.

[注释]：现代CPU，例如，Power，PowerPC，MIPS，ARM，都没有段（segmentation）机制，而是直接采取不需要编程感知的Paging虚拟内存机制了。段可以认为是x86/IA32的一个过度历史技术了。



[注释]： Intel CPU的段模式不能关闭。那么如何无缝的略过，只用页机制呢？ 是通过把CS，DS等段选择符预先设置好（参阅宏定义）。然后把GDT里面的描述符都手工设置基址为0，大小是4G。由于基址为0，通过段机制出来的逻辑地址就没有受到任何影响，完整的进入Page机制转换。

Linux/x86对段机制的设置

Note：通过预设CS，DS等selector，和相应的Descriptor的基址为0，完整的把逻辑地址传递给后面的Page机制，做物理地址转换

```
76 #define GDT_ENTRY_KERNEL_BASE   12
77
78 #define GDT_ENTRY_KERNEL_CS     12      (GDT_ENTRY_KERNEL_BASE + 0)
79
80 #define GDT_ENTRY_KERNEL_DS     13      (GDT_ENTRY_KERNEL_BASE + 1)

185 #define __KERNEL_CS     (GDT_ENTRY_KERNEL_CS * 8)  96. 0x60
186 #define __KERNEL_DS     (GDT_ENTRY_KERNEL_DS * 8)  104. 0x68
187 #define __USER_DS       (GDT_ENTRY_DEFAULT_USER_DS* 8 + 3)
```

Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	Segment Selectors
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

Descriptor内容。基址为0

Since x86 segmentation hardware cannot be disabled, Linux just uses NULL mappings

NULL是指segmentation相应的描述符的内容Base为0. segmentation的selector不能为空。

Linux defines four segments

- Set segment base to 0x00000000, limit to 0xffffffff
- segment offset == linear addresses

  例如，通过Intel CPU的EAX等32bit的寄存器。由于Segment的base为0.我们获得没有变化的32 bit的线性地址。进入Page Table方式的转换。
- User code (segment selector: __USER_CS)
- User data (segment selector: __USER_DS)
- Kernel code (segment selector: __KERNEL_CS)
- Kernel data (segment selector: __KERNEL_DATA)

[注释]：技术的发展都是演变的。8086和80286是16位机GPR）。地址总线是20位和24位。要通过段reg + Offset来“拼凑”一个20 / 24的物理地址。386是32位机器了。清一色32位；有了Paging部件。段部件的偏移量也是32位。所以只要段基址为0，就可以使用基于Paging的Flat内存模型

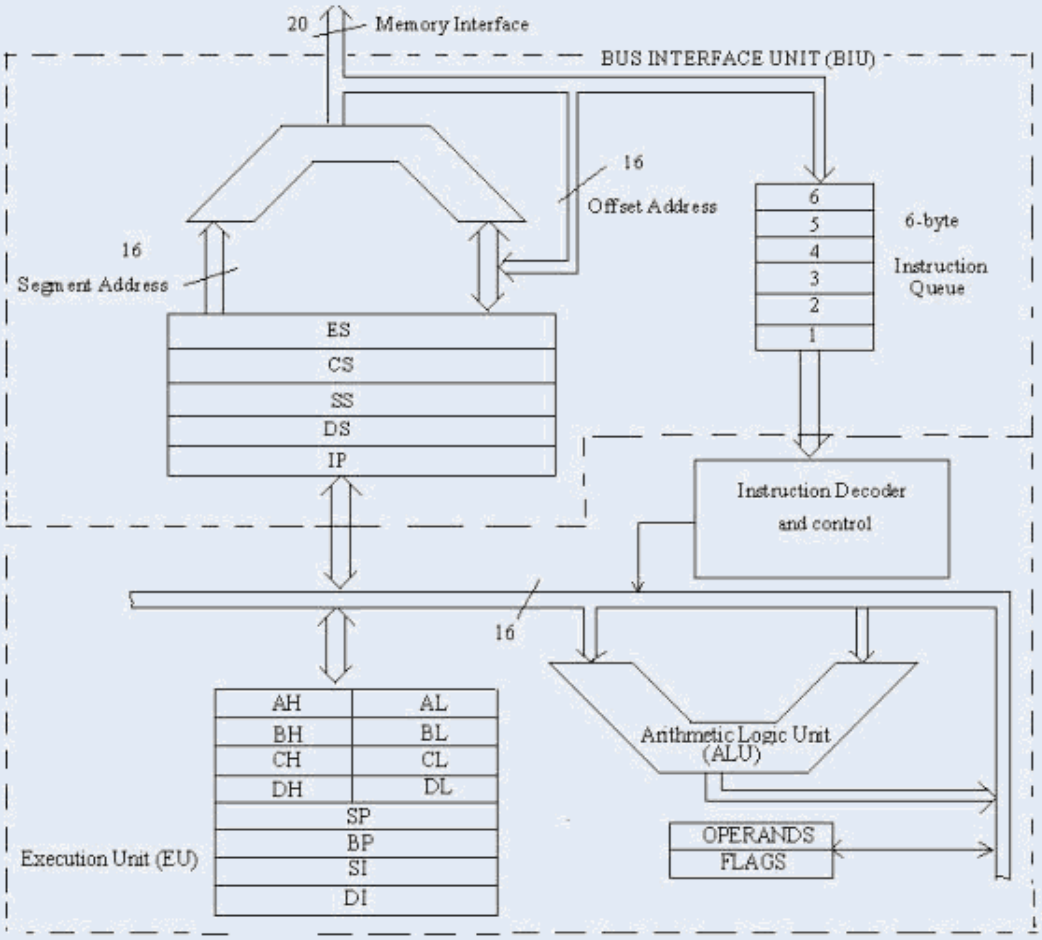
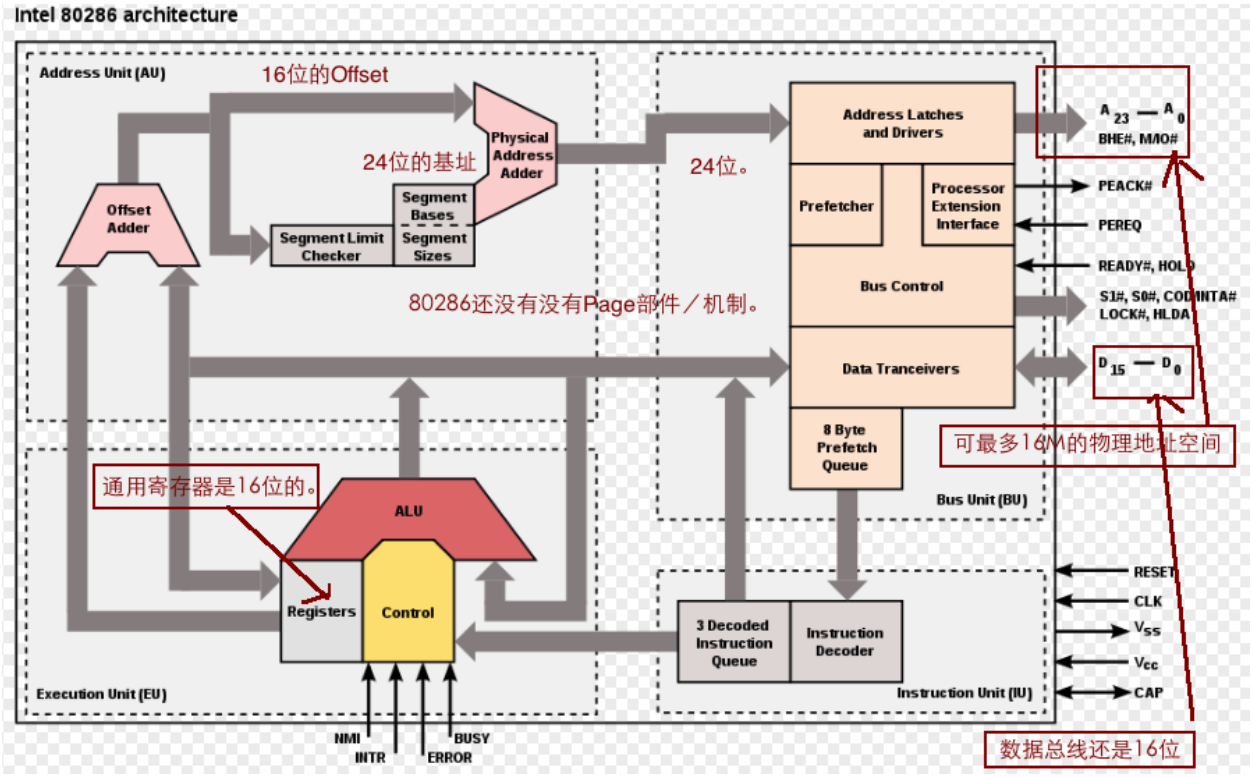
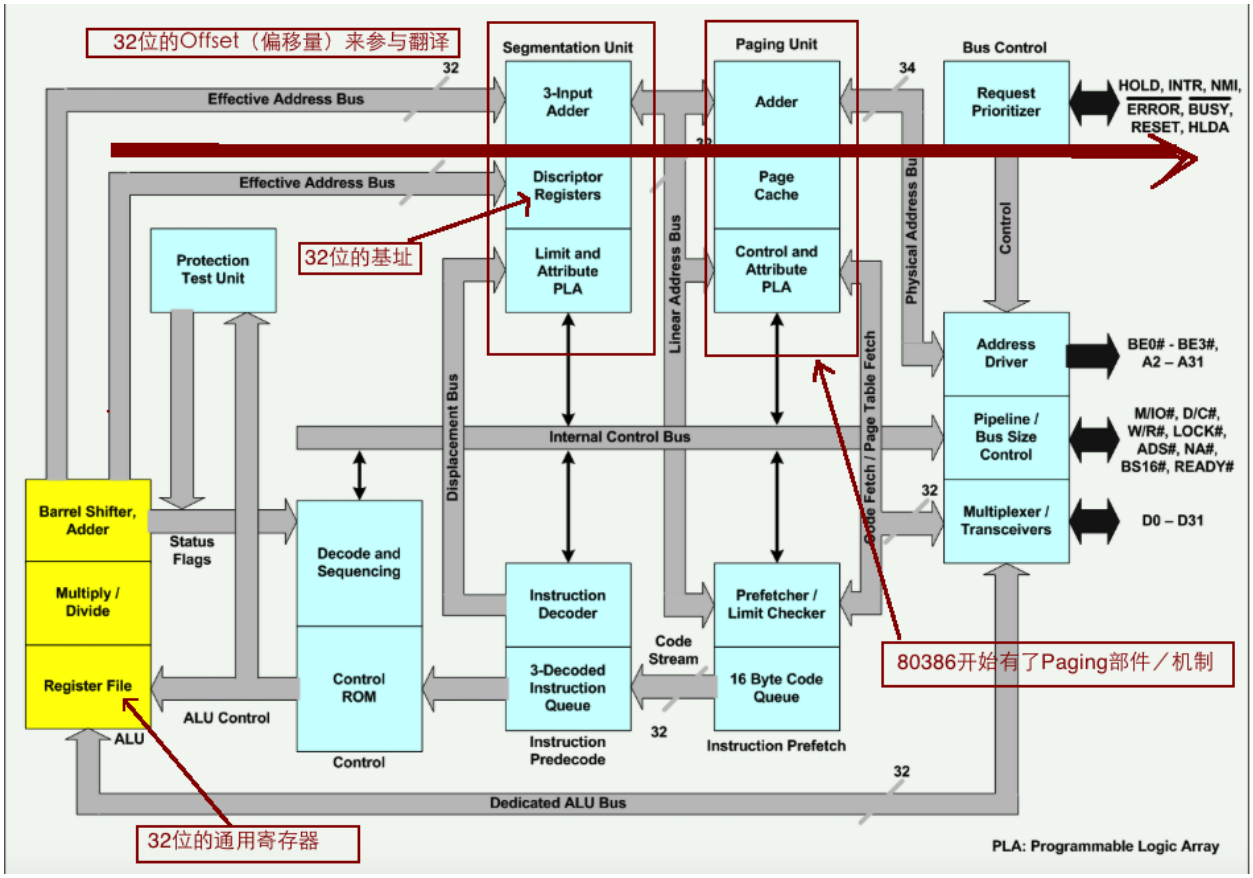


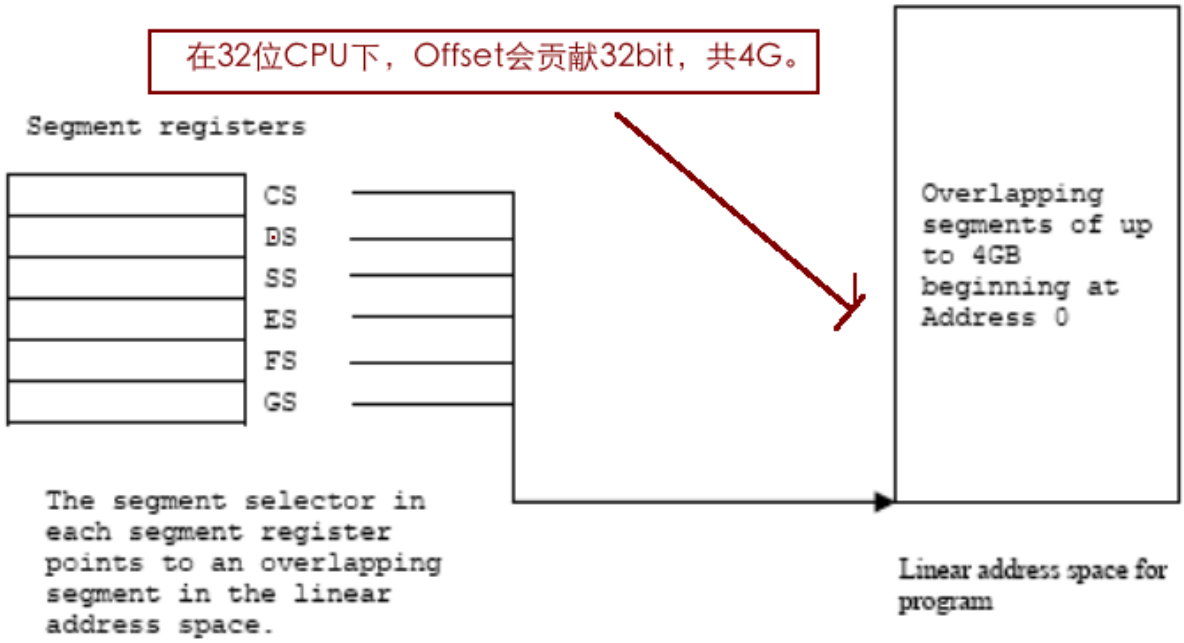
Fig 29.1 Basic Architecture of 8086 Microprocessor


Intel 80286 architecture
Address Unit (AU)
16位的Offset
24位的基址
Physical Address Adder
Offset Adder
Segment Limit Checker
Segment Bases
Segment Sizes
24位。
Address Latches and Drivers
Prefetcher
Processor Extension Interface
Bus Control
Data Tranceivers
8 Byte Prefetch Queue
Bus Unit (BU)
A23 — A0
BHE#, M/IO#
PEACK#
PEREQ
READY#, HOLD
S1#, S0#, COD/INTA#
LOCK#, HLDA
D15 — D0
80286还没有没有Page部件／机制。
可最多16M的物理地址空间
通用寄存器是16位的。
ALU
Registers
Control
3 Decoded Instruction Queue
Instruction Decoder
Instruction Unit (IU)
Execution Unit (EU)
RESET
CLK
Vss
Vcc
CAP
NMI
INTR
BUSY
ERROR
数据总线还是16位



平坦内存模型

在32位CPU下，Offset会贡献32bit，共4G。



编译链接后，每个进程都是自己的0-4G逻辑地址。

4.5 虚拟存储管理

4.5.1 虚拟存储器的概念

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的主存空间，装入其全部信息，否则进程无法运行。把进程的全部信息装入主存后，实际上并非同时使用，有些部分运行一遍，有些部分甚至从不使用，进程在运行时不用的，或暂时不用的，或某种条件下才用的程序和数据，全部驻留于主存是对宝贵的存储资源的一种浪费，会降低主存利用率，这种做法很不合理。于是，提出新的想法：不必装入进程的全部信息，仅将当前使用部分先装入主存，其余部分存放在磁盘中，待使用时由系统自动将其装进来，这就是虚拟存储管理技术的基本思路。当进程所访问的程序和数据在主存中时，可顺利执行；如果处理器所访问的程序或数据不在主存中，为了继续执行，由系统自动将这部分信息从磁盘装入，这叫做"部分装入"；如若此刻没有足够的空闲物理空间，便把主存中暂时不用的信息移至磁盘，这叫做"部分替换"。如果"部分装入、部分替换"能够实现，那么，当主存空间小于进程的需要量时，进程也能运行；更进一步地，当多个进程的总长超出主存总容量时，也可将进程全部装入主存，实现多道程序运行。这样，不仅能充分地利用主存空间，而且用户编程时不必考虑物理空间的实际容量，允许用户的逻辑地址空间大于主存物理地址空间，对于用户而言，好像计算机系统具有一个容量硕大的主存储器，称其为"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主存容量大得多的、可寻址的"主存储器"。实际上，虚拟存储器对用户隐蔽可用物理存储器的容量和操作细节，虚拟存储器的容量与物理主存大小无关，而受限于计算机的地址结构和可用的磁盘容量，如 Intel x86 的地址线是 32 位，则程序可寻址范围是 4 GB，Windows 和 Linux 都为应用进程提供一个 4 GB 的逻辑主存。

如图 4.18 所示是虚拟存储器的概念图，其中，逻辑地址是从进程角度所看到的逻辑主存单元，而物理地址是从处理器角度所看到的物理主存单元，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实际物理主存加部分硬盘区域所组成的存储空间。

图 4.18 虚拟存储器概念图

MMU转换，段 or／and页表

局部性原理是计算机科学最重要的一个原理。
http://en.wikipedia.org/wiki/Locality_of_reference

下面讨论进程信息不全部装入主存时能否正确运行。早在 1968 年，P.Denning 研究程序执行时的局部性原理，对此进行研究的人士还有 Knuth（分析一组学生的 FORTRAN 程序）、Tanenbaum（分析操作系统的过程）、Huck（分析通用科学计算程序），发现程序和数据的访问都有聚集成群的倾向。某存储单元被使用之后，其相邻的存储单元也很快会被使用（称为空间局部性，spatial locality），或者最近访问过的程序代码和数据很快又被访问（称为时间局部性，temporal locality）。对程序的执行进行分析可以发现：第一，程序中只有少量分支和过程调用，大都是顺序执行的指令；第二，程序往往含有若干循环结构，由少量代码组成，而被多次执行；第三，过程调用的深度限制在小范围内，因而，指令引用通常被局限在少量的过程中；第四，许多计算涉及数组、记录之类的数据结构，对它们的连续引用是对位置相邻的数据项的操作；第五，程序中有些部分彼此互斥，不是每次运行时都用到，例如，出错处理部分在正常情况下用不到。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主存，只装入一部分进程信息的假设是合理的，此时只要调度得当，不仅可正确运行进程，而且能在主存中放置更多的进程，充分利用处理器和存储空间。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主存中。这种将物理空间和逻辑空间分开编址、互相隔离，但又统一管理和使用的技术为用户编程提供了极大的方便。

虚拟存储管理与对换技术虽说都是在主存储器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，当其所需主存空间大于当前系统的拥有量时，进程无法被对换进主存工作；而虚拟存储管理以页或段为单位，即使进程所需主存空间大于当前系统拥有的主存总量，仍然能正常运行，因为系统可将其他进程的一部分页或段换出至磁盘。

虚拟存储器的思想早在 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，逐步为广大计算机研制者和用户所接受。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。为了实现虚拟存储器，必须解决好以下问题：主存与辅助存储器（磁盘）统一管理问题、逻辑地址到物理地址的转换问题、部分装入和部分替换问题。实现虚拟存储器要付出一定的开销，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主存与辅助存储器交换页或段的 I/O 开销等。目前，虚拟存储管理主要采用以下几种技术实现：请求分页、请求分段和请求段页虚拟存储管理。

4.5.2 请求分页虚拟存储管理

目前主要是Paging分页管理了。

1. 请求分页虚拟存储管理的硬件支撑

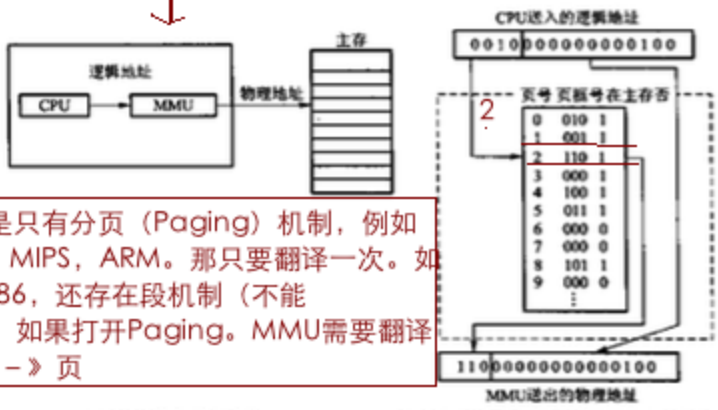
操作系统的存储管理依靠低层硬件的支撑来完成任务，此硬件称为主存管理部件（Memory Management Unit，MMU），它提供地址转换和存储保护的功能，并支持虚拟存储管理和多任务管理。MMU 由一组集成电路芯片组成，逻辑地址作为输入，物理地址作为输出，直接送达总线，对主存单元进行寻址，其位置和功能如图 4.19(a)所示，其内部执行过程如图 4.19(b)所示。主要

大多数CPU都配置MMU部件。没有的很少了，不讨论。MMU主要完成需逻辑地址或者虚拟地址到物理地址的转换。

有的CPU叫做逻辑地址，有的叫虚拟地址。道理都一样：不是最后的物理地址。要送给MMU翻译一次。

Offset是12个bit。4K大小。

功能列举如下。



(a) MMU的位置和功能　　(b) 16个4KB页面情况下MMU的内部操作

图 4.19　主存管理部件

MMU可以是只有分页（Paging）机制，例如PowerPC，MIPS，ARM。那只要翻译一次。如果像Intel x86，还存在段机制（不能disable），如果打开Paging。MMU需要翻译两次：段－－》页

高位4个bit可以来区分16个页面。

(1) 管理硬件页表基址寄存器

每当发生进程上下文切换时，系统负责把将要运行的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面号和页内位移，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，找到相应的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，待缺页中断处理结束，把相应的页面和页框号装入。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，则可访问主存完成指令，同时，把此页面信息装入 TLB。

(5) 发出相应中断　Page Fault Exception

当查出页表中有页失效位或页面访问越界位时，发出缺页中断或越界中断，并将控制权交给内核存储管理。

(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。

不同的CPU情况会有略微不同。清空是防止发生另外一个进程的虚拟地址A的TLB Entry被当作了其他进程的条目了。

”被解释为“的理解很重要。是一个隐含的对待，不是存在类似一个段寄存器CS，或者DS，可以指定一个段。

下面考察 MMU 的工作过程。在图 4.19(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“在主存否”位为 1，表明此页在主存，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 位的物理地址(二进制表示为 110000000000100)，并把它送到主存总线；如果“在主存否”位为 0，则将触发缺页中断，陷入操作系统进行调页处理。

2. 请求分页虚拟存储管理的基本原理

请求分页虚拟存储管理是将进程信息的副本存放在辅助存储器中，当它被调度投入运行时，并不把程序和数据全部装入主存，仅装入当前使用的页面，进程执行过程中访问到不在主存的页面时，再把所需信息动态地装入。使用频率较高的分页虚拟存储管理是请求分页(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主存时，产生缺页(page fault)异常，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页面不会被装入主存。

请求分页虚拟存储管理与分页实存管理不同，仅让进程当前使用部分装入，必然会发生某些页面不在主存中的情况。那么，如何发现页面不在主存中呢？所采用的办法是：扩充页表项的内容，增加驻留标志位，又称页失效异常位，用来指出页面是否装入主存。当访问一个页面时，如果某页的驻留标志位为 1，表示此页已经在主存中，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，操作系统根据磁盘地址将这个页面调入主存，然后重新启动相应的指令。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，通常规定扇区的长度等于页面的长度，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，并把进程程序页面装入辅助存储器，外页表也按进程页号的顺序排列。为了节省主存空间，外页表可存放在磁盘中，当发生缺页中断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主存时由硬件所产生的一种特殊中断信号，CPU 通常在一条指令执行完成后才检查是否有中断到达，也就是说只能在两条指令之间响应中断，但缺页中断却是在指令执行期间产生并获得系统处理的。而且，一条指令可能涉及多个页面，例如，指令本身跨页、指令所处理的数据跨页，完全有可能在执行一条指令的过程中发生多次缺页异常。

为了对页面实施保护和淘汰等各种控制，可在页表中增加标志位，其他标志位包括修改位、引用位和访问权限位等，用来跟踪页面的使用情况。当页面被修改后，硬件自动设置修改位，一旦修改位被设置，当此页被调出主存时必须先被写回磁盘；引用位则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；访问权限位则限定页面允许的访问权限(如读、写、执行等)。

可见，在请求分页虚拟存储管理中，页表中存放的是把逻辑地址转换成物理地址时硬件所需要的信息，其作用主要有：

page on demand是虚拟存储管理重要的手段。

这个地址是而且也必须是 物理地址。不需要任何转换的。

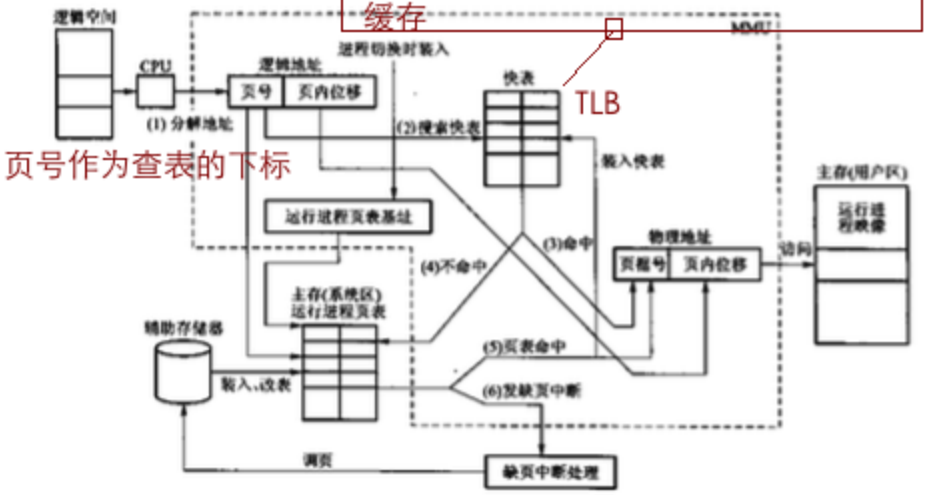
(1) 获得页框号以实现虚实地址转换；

(2) 设置各种访问控制位，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、驻页标志、访问标志、锁定标志和淘汰标志等)。

下面讨论使用快表但是页表存放在主存的情况下，请求分页虚实地址转换的过程：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中，此后，进程开始运行并要访问某个虚地址，MMU 工作，它将完成图 4.20 虚线框内的任务，可看到地址转换过程如下。

TLB是Page Entry在CPU芯片内部的快速缓存



页号作为查表的下标

图 4.20 请求分页虚实地址转换

(1) MMU 接收 CPU 传送过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进程页表，页表基址由硬件页表基址寄存器指出；

(5) 如果在页表中找到此页面，说明所访问页面已在主存中，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

操作系统的许多保护都是通过这个来实现的。例如，用户态的进程不通去访问内核的地址空间。

Page Fault的异常处理是操作系统的一个预先设置好的。在相关的控制寄存器里会存放相关的引起发生这个异常的逻辑地址的信息。

面的信息装入快表 TLB,以备再次访问;

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

MMU 发现缺页并发出缺页中断,存储管理接收控制,进行缺页中断处理的过程如下。

步骤 1:挂起请求缺页的进程;

转向异常处理程序

异常处理程序

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:查看主存是否有空闲页框,如有则找出,修改主存管理表和相应页表项的内容,转步骤 6;

步骤 4:如果主存中无空闲页框,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;若是则转步骤 5;

步骤 5:淘汰页面被写过或修改过,将其内容写回磁盘的原先位置;

步骤 6:进行调页,把页面装入主存所分配的页框中,同时修改进程页表项;

步骤 7:返回进程断点,重新启动被中断的指令。

在分页虚拟存储系统中,由于页面在需要时是根据进程的请求装入主存的,因此称为请求分页虚拟存储管理,IBM/370 系统的 VS/1、VS/2 和 VM/370,Honeywell 的 MULTICS 以及 UNIVAC 系列 70/64 的 VMS 等都采用请求分页虚拟存储管理。这种技术的优点是:进程的程序和数据可按页分散存放在主存中,既有利于提高主存利用率,又有利于多道程序运行。这种技术的缺点是:要有一定的硬件支持,要进行缺页中断处理,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,则主存的损失仍然很大。

3. 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主存,有两种策略可供选择:请页式(demand paging)和预调式(prepaging)。

请页式是仅当需要访问程序和数据时,通过缺页中断并由缺页中断处理程序分配页框,把所需页面装入主存。进程运行时缺页中断很频繁,随着越来越多的页面被装入主存,根据局部性原理,大多数将要访问的程序和数据页面都在最近被装入主存,于是,缺页中断率就会下降。这一策略的优点是确保只有被访问的页面才会调入主存,节省主存空间;其缺点是处理缺页中断的次数多,调页的系统开销大,由于每次仅调用一页,磁盘 I/O 操作次数猛增。

预调式装入主存的页面并非缺页中断所请求的页面,是由操作系统依据某种算法,动态预测进程最可能要访问的那些页面。在使用页面前预先调入主存,尽量做到进程要访问的页面已经调入主存,且每次调入若干页面,而不是仅调入一页。由于进程的页面大多连续存放在磁盘中,一次调入多个连续存放的页面能够减少磁盘 I/O 启动次数,节省寻道和搜索的时间。但是,如果所调入的页面大多未被使用,则效率就很低,可见,预调页要建立在可靠预测的基础之上。Windows 系统使用簇式分页的预调度策略,根据物理主存储器的大小,代码页面会装入 3~8 页,数据页面会装入 2~4 页,这种期望是基于程序的局部性,访问到预取的页面而不致产生额外的缺页。

页面清除策略与装入策略相对应,要考虑何时把修改过的页面写回辅助存储器,常用的方法

通用操作系统基本上上请求式，Paging on Demand；专用操作系统，例如，通信设备，往往是预装，避Page fault的延时。

通信系统最不能承受这样的延迟。

是：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，写出的页仍然在主存中，直到页替换算法选中此页从主存中移出。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。

4. 页面分配策略

请求分页虚拟存储管理排除主存实际容量的约束，使更多的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的代价，由于页面的调入和调出要增加 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，还是主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。

很难有一个最优的替换算法

如果在进程的生命周期中，保持页框数固定不变，称其为固定分配，在创建进程时，根据进程的类型和系统的规则决定页框数，只要有一个缺页中断产生，进程就会有一页被替换；如果在进程的生命周期中，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，可减少所分配的页框数。

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，称为全局页面替换算法；如果页面替换算法的作用范围局限于进程自身，称为局部页面替换算法。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。在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，这是比较困难的任务。

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，如果发生缺页中断，只能从进程在主存的页面中选出一页替换，以保证进程的页框总数不变。这种策略的难点在于：应给进程分配多少页框才合适。分配少了，缺页中断率高；分配多了，会使主存中能同时运行的进程数减少，进而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，从系统空闲页框中对其分配，把所缺页面调入此页框，这样产生缺页的进程的主存空间会逐渐增大，有助于减少系统的缺页中断总次数。当系统所剩有的空闲页框几近耗尽时，要从主存中选择页面淘汰，可以是主存中任一进程的页面，这样又会使某进程的页框数减少，缺页中断率上升。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：

http://en.wikipedia.org/wiki/Page_replacement_algorithm

(1) 新进程装入主存时，根据应用类型、程序要求，分配一定数目的页框，可用请页式或预调式完成这个分配；

(2) 当产生缺页中断时，从进程的页面中选择一个淘汰；

(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，以改善系统的总性能。

5. 缺页中断率

虚拟存储器能够给用户提供一个容量很大的存储空间，但当主存空间已满而又要装入新页时，必须按照预定算法把已在主存中的页面写回，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"(thrashing)，一个好的调度算法应减少和避免抖动现象。

首先定义缺页中断率，假定进程 P 共计 n 页，此进程分得的主存块为 m 块(m、n 均为正整数，且 $1 \leqslant m \leqslant n$)，即主存中最多只能容纳进程的 m 页。如果进程 P 在运行中成功访问的次数为 S，不成功访问的次数为 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) 主存页框数：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中断次数；

(4) 程序特性：程序局部性要好，它对缺页中断率有很大影响。例如，一个程序将 128×128 的数组置初值"0"，假定它仅分得一个主存块，页面尺寸为 128 个字，数组中的元素各行分别存放在一页中，开始时第一页在主存中。若程序按如下左边编写：

```
int A[128][128];
for(int j=0;j<128;j++)
  for(int i=0;i<128;i++)
    A[i][j]=0;
```

```
int A[128][128];
for(int i=0;i<128;i++)
  for(int j=0;j<128;j++)
    A[i][j]=0;
```

按行（row）来赋值

则每执行一次 A[i][j]=0 将产生一次缺页，共产生(128×128−1)次缺页中断；若按如上右边重新编写这个程序，共只产生(128−1)次缺页中断。显然，虚拟存储器的效率与程序局部性程度密切相关，局部性程度因程序而异。

6. 全局页面替换策略

在多道程序的正常运行过程中，属于不同进程的页面被分散存放在主存页框中，当发生缺页中断时，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 一个理论算法。要知道整体情况

1966 年，Belady 提出最佳页面替换算法(OPTimal replacement，OPT)。当要调入一页面必

按照列来写。因为程序只有一个内存快，所以，每写一个value就产生一个缺页。产生颠簸现象。 A[1][0]与A[2][0]相隔了一页！

[注释]：Paging on Demand是现代通用操作系统VM管理重要的机制。是一种滞后算法，当需要时来完成虚拟页面（Page）和物理页框（Frame）的分配，提高效率。但数据通信系统中，为了避免Page Fault带来的延迟和不确定性，往往是事先完成页框的分配从而确保报文处理的实时性。

OPT算法是一个理想调度。需要对内存页的未来使用情况有了解。

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要访问的页面。它所产生的缺页数最少，然而，却需要预测程序的页面引用串，这是无法预知的，不可能对程序的运行过程做出精确的断言，不过此理论算法可用做衡量各种具体算法的标准。

FIFO (2) 先进先出页面替换算法

基本上不存在用FIFO算法的操作系统了。

基于程序总是按线性顺序来访问物理空间这一假设，总是淘汰最先调入主存的页面，即淘汰在主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：

$$P[0],P[1],\cdots,P[m-1]$$

k顺序左移一个位置。如果到边界，折回到第一个位置。

其中，每个 $P[i](i=0,1,\cdots,m-1)$存储一个装入主存中的页面的页号。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

$$P[k]=\text{新页的页号};k=(k+1)\%m;$$

假定主存中已经装入 m 页，k 的初值为 0，那么，第一次淘汰页的页号应为 P[0]，而调入新页后，P[0]的值为新页的页号，k 取值为 1；……；第 m 次淘汰页的页号为 $P[m-1]$，调入新页后，$P[m-1]$的值为新页的页号，k 取值为 0；第 $m+1$ 次页面淘汰时，淘汰页的页号为 P[0]所指向的页面，因为它是在主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，面对具有其他特性的程序则效率不高，因为在主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主存时间最长的页，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 算法还伴有一种奇怪的现象，称 Belady 异常，增加可用主存块的数量会导致更多的缺页。

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，前者是由修改页面的页框所构成的链表；后者是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，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，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，必须维护一个特殊的队列——页面淘汰队列，此队

LRU算法目前是最广泛使用的替换算法。
但LRU开销比较大，例如，队列操作。所以许多近似算法被采用。

页面4被“最近”访问了。类似于刷了存在感。
因此页面3是最“久”没有被用的了。

列存放当前在主存中的页号，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，发生缺页中断时总淘汰队列头所指的页；面执行页面访问后，需要从队列中把此页调整到队列尾。例如，给某进程分配 3 个主存块，依次访问的页号为：4,3,0,4,1,1,2,3,2。于是当访问这些页时，淘汰页面序列的变化情况如下所示：

访问页号	淘汰页面序列	被淘汰页面
4	4	
3	4,3	
0	4,3,0	
4	3,0,4	
1	0,4,,1	3
1	0,4,1	
2	4,1,2	0
3	1,2,3	4
2	1,3,2	

页面1要进来，必须空一个位置
选择页面3淘汰

页2变为最新访问的。

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。

模拟方法一是引用位法，又称最近未使用页面替换算法(Not Recently Used,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，在选中要淘汰的页后，也将其他页的引用位 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，缺页中断时所有页的 R 值均为 1；t 小了，缺页中断时可能所有页的 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。

模拟方法二是计数器法，每当页面被引用时，硬件计数器自动计数，更换访问页面时，把硬件计数器的值记录到页表的计数值字段，经过时间 t 后，将所有计数器全部清除。页面置换时，系统检查所有页表项，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,NFU)。

模拟方法三是记时法，为每页增设一个记时单元，每当页面被引用时，把系统的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。

(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，这是最早进入主存的页面，如果其“引用位”是 0，那么这个页面既时间长又没有用，选择此页面淘汰；如果其“引用位”是 1，说明虽然它进入主存的时间较早，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

LRU算法实现代价比较大。往往用一些近似算法，例如Pseudo – LRU等。

是算法效率优化。算法本身等价于SCR。

面替换算法(Second Chance Replacement,SCR),其含义是最先进入主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主存中。如果主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,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面上的表针,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,仍然要使用页表中的"引用位",把进程已调入主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 一个页面首次装入主存时,其"引用位"置0;

② 主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1,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4.21给出Clock算法的一个例子。当发生缺页中断时,将装入主存的页面是Page727,指针所指向的是Page45(在页框2中),Clock算法执行过程如下:Page45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Page556(在页框4中)的"引用位"是0,于是Page556被Page727替换,并把Page727的"引用位"置1,指针前进到下一页Page13(在页框5中),算法执行到此结束。

淘汰页面时,如果此页面已被修改过,必须将它先重新写回磁盘;但如果所淘汰的是未被修改过的页面,就不需要写磁盘操作,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($r=1, m=1$)。

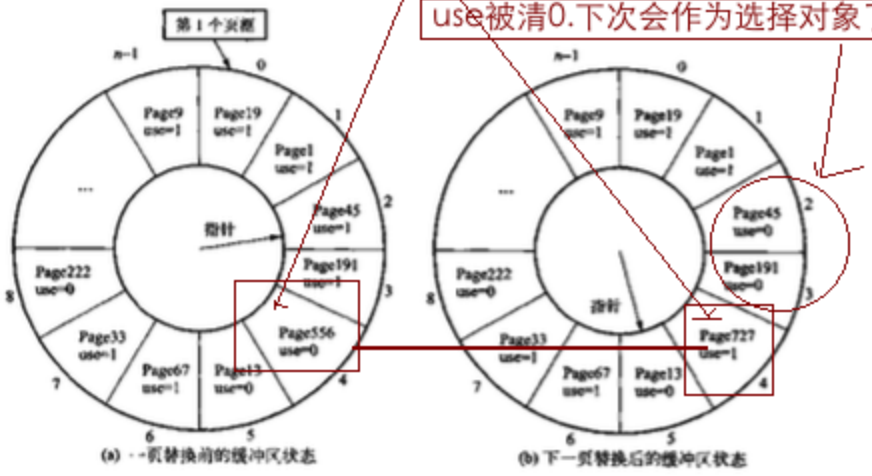
于是,改进的Clock页面替换算法可如下执行:

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。

修改过会涉及I/O操作。所以,尽量选择没有修改过的。

第一个use＝0的，被替换。

use被清0.下次会作为选择对象了。



(a) 一页替换前的缓冲区状态　　(b) 下一页替换后的缓冲区状态

图 4.21　时钟页面替换算法

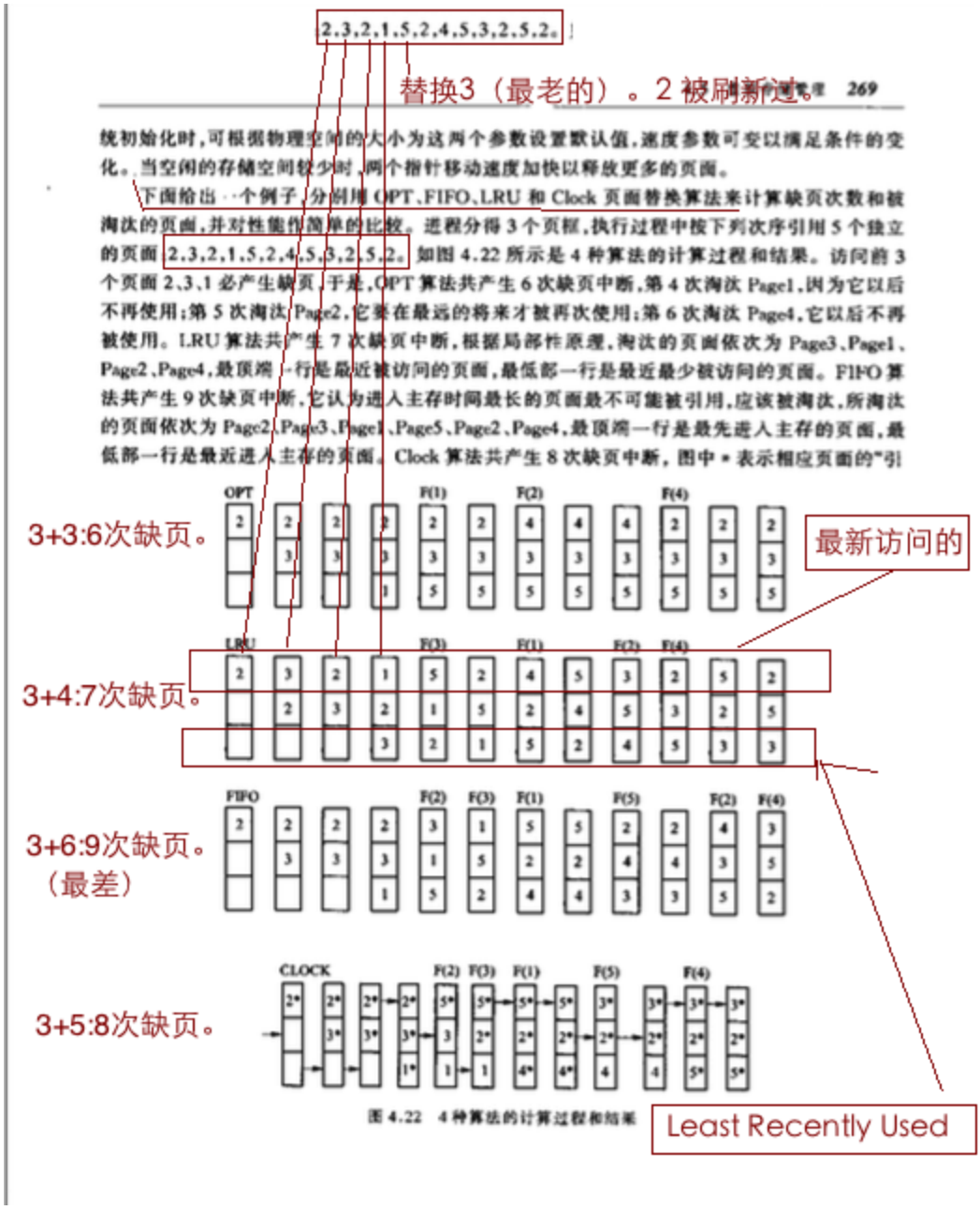
步骤 2：如果步骤 1 失败，再次从原位置开始，查找 $r=0$，$m=1$ 的页面，把遇到的第一个这样的页面作为淘汰页，而在扫描过程中把指针所经过的页面的"引用位"r 置 0。

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。

UNIX SVR4 使用改进的 Clock 页面替换算法，称双指针 Clock 算法，实现思想如下：主存中的每个页面都有"引用位"相连，当一个页面被装入主存时，"引用位"置 0；当一个页面被引用时，"引用位"置 1。系统设置两个时钟指针，称为前指针和后指针，页面守护进程被周期性地唤醒工作，两个指针都扫描一遍：前指针先依次扫过页面并把"引用位"置 0；再延迟一定的时间之后，后指针依次扫描页面，若此页面的"引用位"为 1，表明这个期间页面被引用过，则跳过此页面继续扫描，若此页面的"引用位"为 0，表明页面未被引用过，那么，此页面可作为候选的淘汰者。双指针 Clock 算法的关键：一是两个指针扫描页表的速度；二是两个指针之间时间间隔的选择，在系

后指针来判断是否是可替换的页。这样就可以控制算法做解决的时间间隔。。。



[注释]：页面替换算法只有知道未来页面使用情况，才能达到最优。所以，OPT（OPTimal Replacement）是一个理想。现实中，有LRU和相应的近似算法（NRU，NFU），FIFO，SCR，Clock和各种变种。其中LRU和Clock比较好。在CPU的cache设计中，芯片的cache set的算法说Pseudo - LRU。

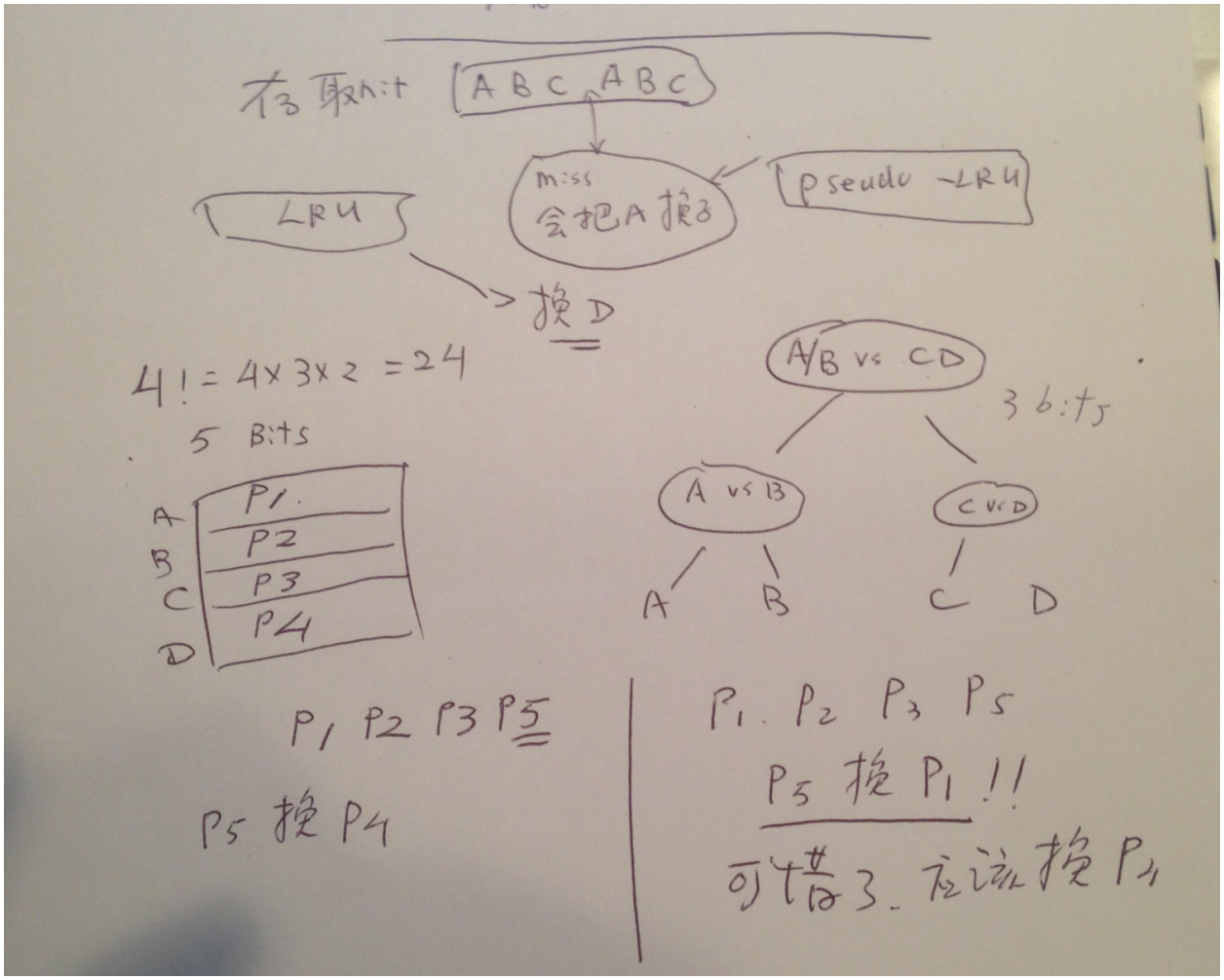
LRU代价比较大。需要维持所有页面使用情况来定位"最近没有被使用的页面"。例如如果有4个页面其使用情况时4！=24种。如果是8个页面，使用

顺序有40320种。想想256个页面，代价太大。现实系统中，需要实时的，多采用一些近似算法，例如“尽可能最近没有使用的”

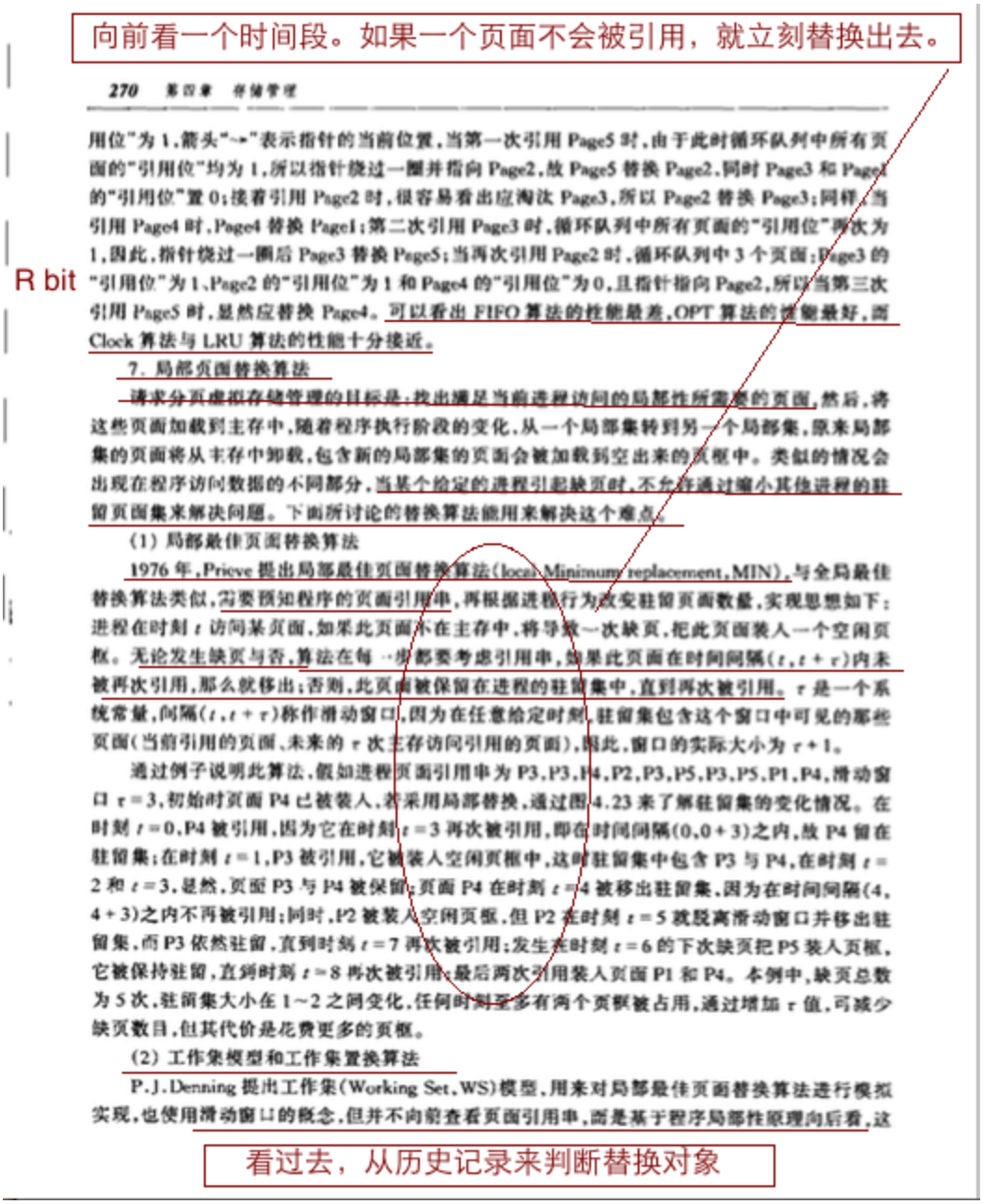
One interesting point is that true LRU replacement can quickly gobble up lots of bits of memory. It can be easily observed that there are $N!$ orders in which N letters of the alphabet can be ordered. A four-Way cache must have five LRU bits for each line to represent the 24 (4!) following possible states of use of the cache contents (order in which Ways A, B, C, and D were used):

ABCD	ABDC	ACBD	ADBC	ACDB	ADCB
BACD	BADC	CABD	DABC	CADB	DACB
BCAD	BDAC	CBAD	DBAC	CDAB	DCAB
BCDA	BDCA	CBDA	DBCA	CDBA	DCBA

since these 24 states require five bits ($2^5 = 32 > 24$) to encode.



[注释]：在CPU的Cache替换算法中，常采用Pseudo - LRU。就是一个近似算法。“差不多就行了”。换出去的一定不会是最近刚用的，但确实可能不是最不用的。如图， 在ABC都是hit引用之后，如果有 miss，精确LRU算法应该是把D换掉 [D就没有用过] 。但如果是PLRU算法，会换了A。



270 第四章 存储管理

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 Page4 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，所以当第三次引用 Page5 时，显然应替换 Page4。可以看出 FIFO 算法的性能最差，OPT 算法的性能最好，而 Clock 算法与 LRU 算法的性能十分接近。

7. 局部页面替换算法

请求分页虚拟存储管理的目标是：找出满足当前进程访问的局部性所需要的页面，然后，将这些页面加载到主存中，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主存中卸载，包含新的局部集的页面会被加载到空出来的页框中。类似的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，如果此页面不在主存中，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$(t, t+\tau)$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$(t, t+\tau)$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 次主存访问引用的页面)，因此，窗口的实际大小为 $\tau+1$。

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 $\tau=3$，初始时页面 P4 已被装入，若采用局部替换，通过图4.23来了解驻留集的变化情况。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的页框。

(2) 工作集模型和工作集置换算法

P.J.Denning 提出工作集(Working Set，WS)模型，用来对局部最佳页面替换算法进行模拟实现，也使用滑动窗口的概念，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，这

P4朝前看3步，不会别引用。被立刻换出去。

时刻 t	0	1	2	3	4	5	6	7	8	9	10
引用串	P4	P3	P3	P4	P2	P3	P5	P3	P5	P1	P4
P1	–	–	–	–	–	–	–	–	–	√	–
P2	–	–	–	–	√	–	–	–	–	–	–
P3	–	√	√	√	√	√	√	√	–	–	–
P4	√	√	√	√	–	–	–	–	–	–	√
P5	–	–	–	–	–	–	√	√	√	–	–
In_t		P3			P2		P5			P1	P4
Out_t					P4	P2			P3	P5	P1

图 4.23 局部最佳页面替换算法示例

这个时候并没有缺页发生。但工作集算法每次扫描清理工作集

意味着在任何给定的时刻，一个进程不久的将来所需主存页框数可通过考查其最近时间内的主存需求做出估计。

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 $W(t,\Delta)$ 表示在时刻 $t-\Delta$ 到时刻 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"。

在图 4.24 的例子中，页面引用串与上例相同，工作集窗口尺寸 $\Delta=3$。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。第一次缺页中断发生在时刻 $t=1$，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 $t=4$，缺页中断会把 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 $t=6$，发生缺页中断并装入 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留；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 $t=10$ 发生两次缺页中断，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的大小。

① 监视每个进程的工作集，只有属于工作集的页面才能驻留在主存；

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主存时，进程才能执行。

Windows 的页面替换机制结合工作集模型和 Clock 页面替换算法的优点，采用局部替换算法，进程缺页时，不会逐出其他进程的页面。系统为每个进程维护一个工作集，系统指定工作集的最小尺寸(20～50 个页框)和最大尺寸(45～345 个页框)；发生缺页中断时，把引用到的页面添

加至进程工作集中，直至达到最大值，此时，若还发生缺页中断，必须从工作集中移出一个页面；淘汰页面的选择使用模拟 LRU 和 Clock 策略的变种，每个页框有一个访问位 u 及一个计数器 count。此页被引用时，u 位被硬件置 1；当在工作集中寻找淘汰页面时，工作集管理程序扫描工作集中页面的访问位，并执行操作。如果 u=1，那么，把 u 和 count 清 0；否则，count 加 1，扫描结束时，将 count 值最大的页面移出。从工作集中移出的页面被放入两个主存队列之一：一是保存暂时移出并已被修改过的页面；二是保存暂时移出并为"只读"的页面。如果其中的页面被再次引用，可迅速地从队列中找回，而不会产生缺页中断，仅当实际的空闲页框队列为空时，它们才被用来满足缺页需求。

看历史，P5太"旧"了。替换出去。

时刻 t	0	1	2	3	4	5	6	7	8	9	10
引用串	P1	P3	P5	P4	P2	P3	P5	P3	P5	P1	P4
P1	√	√	√	√	–	–	–	–	–	√	√
P2	–	–	–	–	√	√	√	√	–	–	–
P3	–	√	√	√	√	√	√	√	√	√	√
P4	√	√	√	√	√	√	√	–	–	–	√
P5	√	√	–	–	–	–	√	√	√	√	√
In_t		P3			P2		P5			P1	P4
Out_t			P5		P1			P4	P2		

图 4.24 工作集替换示例

(3) 模拟工作集替换算法 Aging算法

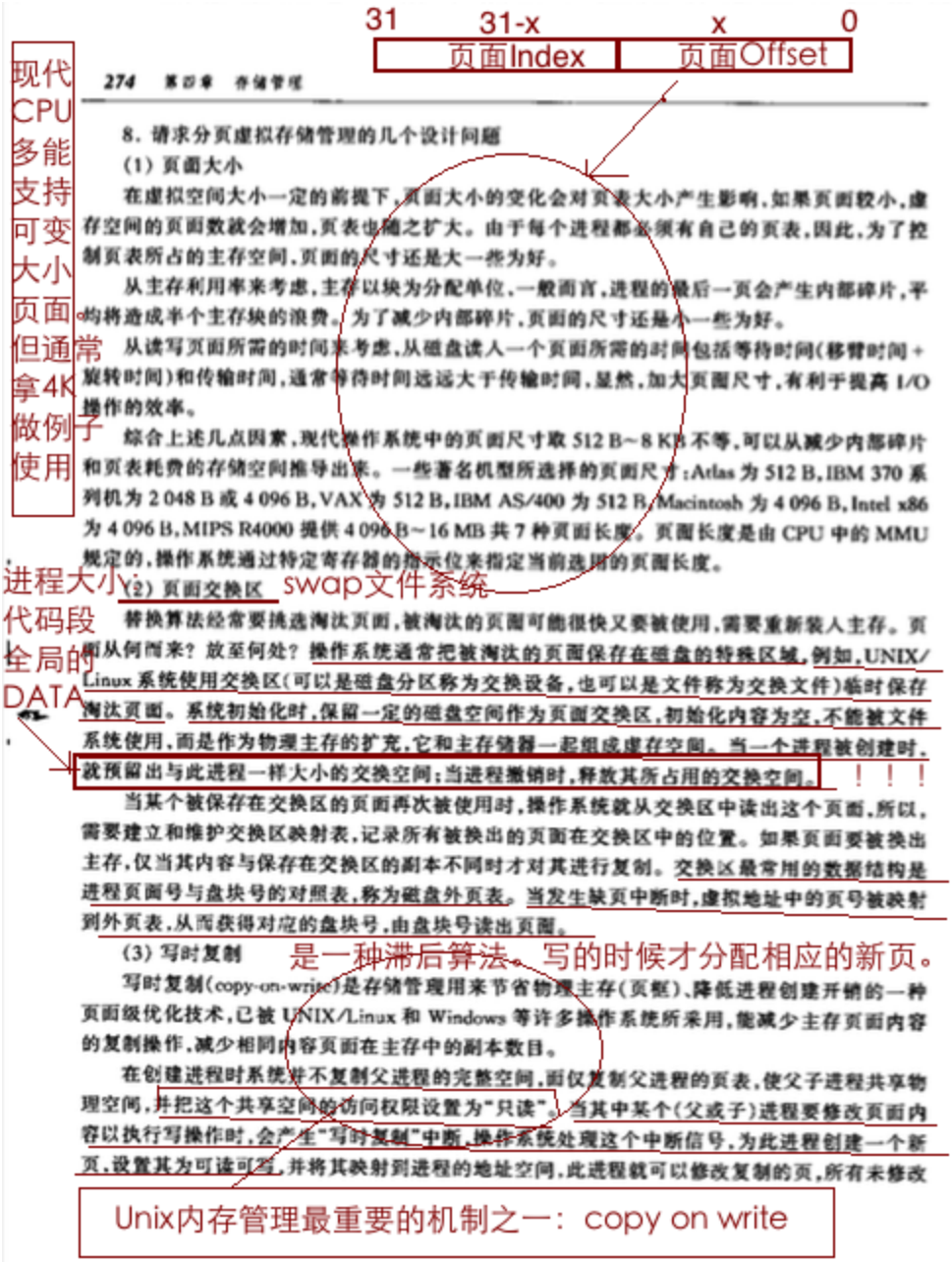
工作集策略在概念上很有吸引力，但实现中监督驻留页面变化的开销却很大，估算合适的窗口 Δ 的大小也是一个难题。为此，已经设计出各种模拟工作集替换算法，下面介绍两种。

进程在运行前要把它的工作集预先装入主存，为每个页设置引用位 r 及年龄寄存器，寄存器初始值为 0，每隔时间 t，系统扫描主存中的所有页面，先将寄存器右移一位，再把引用位 r 的值加到对应寄存器的最左边，这样，未引用页面其年龄寄存器的值逐步减小，当达到下限或值 0 时，由于页面已经落在窗口之外，就可以把它从工作集中移出去。

例如，年龄寄存器共有 4 位，时间间隔 t 定为 1 000 次存储器引用(即 1 000 个指令周期)，页面 P 在时刻 $t+0$ 时年龄寄存器的值为"1 000"，在时刻 $t+1\,000$ 时年龄寄存器的值为"0100"，在时刻 $t+2\,000$ 时年龄寄存器的值为"0010"，在时刻 $t+3\,000$ 时年龄寄存器的值为"0001"，在时刻 $t+4\,000$ 时年龄寄存器的值为"0000"，此时，页面 P 被移出工作集，这就有效地模拟窗口大小为 1 000×4 的一个工作集。

修改后的算法称为"老化(aging)算法"，年龄寄存器各位的累加值反映页面最近使用的情况，访问次数越多，累加值越大，而较早被访问的页面随着寄存器各位的右移，由于老化使得其作

[注释]：局部页面替换算法概念很直接。就是每个进程的页面替换不影响其他进程的。这样可以防止系统的颠簸。Working set的变种叫做Aging算法。目的都是一个，逐步收敛和定位一个被替换的页面，例如某个算法中衡量参数的最小值。Aging中每次右移一个bit，就是一种衰减过程。


31
31-x
x
0
页面Index
页面Offset
现代CPU多能支持可变大小页面，但通常拿4K做例子使用
274 第四章 存储管理
8. 请求分页虚拟存储管理的几个设计问题
(1) 页面大小
在虚拟空间大小一定的前提下，页面大小的变化会对页表大小产生影响。如果页面较小，虚存空间的页面数就会增加，页表也随之扩大。由于每个进程都必须有自己的页表，因此，为了控制页表所占的主存空间，页面的尺寸还是大一些为好。
从主存利用率来考虑，主存以块为分配单位，一般而言，进程的最后一页会产生内部碎片，平均将造成半个主存块的浪费。为了减少内部碎片，页面的尺寸还是小一些为好。
从读写页面所需的时间来考虑，从磁盘读入一个页面所需的时间包括等待时间(移臂时间+旋转时间)和传输时间，通常等待时间远远大于传输时间，显然，加大页面尺寸，有利于提高 I/O 操作的效率。
综合上述几点因素，现代操作系统中的页面尺寸取 512 B~8 KB 不等，可以从减少内部碎片和页表耗费的存储空间推导出来。一些著名机型所选择的页面尺寸：Atlas 为 512 B，IBM 370 系列机为 2 048 B 或 4 096 B，VAX 为 512 B，IBM AS/400 为 512 B，Macintosh 为 4 096 B，Intel x86 为 4 096 B，MIPS R4000 提供 4 096 B~16 MB 共 7 种页面长度。页面长度是由 CPU 中的 MMU 规定的，操作系统通过特定寄存器的指示位来指定当前选用的页面长度。
(2) 页面交换区
swap文件系统
进程大小，代码段，全局的DATA
替换算法经常要挑选淘汰页面，被淘汰的页面可能很快又要被使用，需要重新装入主存。页面从何而来？放至何处？操作系统通常把被淘汰的页面保存在磁盘的特殊区域，例如，UNIX/Linux 系统使用交换区(可以是磁盘分区称为交换设备，也可以是文件称为交换文件)临时保存淘汰页面。系统初始化时，保留一定的磁盘空间作为页面交换区，初始化内容为空，不能被文件系统使用，而是作为物理主存的扩充，它和主存储器一起组成虚存空间。当一个进程被创建时，就预留出与此进程一样大小的交换空间；当进程撤销时，释放其所占用的交换空间。
！！！
当某个被保存在交换区的页面再次被使用时，操作系统就从交换区中读出这个页面，所以，需要建立和维护交换区映射表，记录所有被换出的页面在交换区中的位置。如果页面要被换出主存，仅当其内容与保存在交换区的副本不同时才对其进行复制。交换区最常用的数据结构是进程页面号与盘块号的对照表，称为磁盘外页表。当发生缺页中断时，虚拟地址中的页号被映射到外页表，从而获得对应的盘块号，由盘块号读出页面。
(3) 写时复制
是一种滞后算法。写的时候才分配相应的新页。
写时复制(copy-on-write)是存储管理用来节省物理主存(页框)、降低进程创建开销的一种页面级优化技术，已被 UNIX/Linux 和 Windows 等许多操作系统所采用，能减少主存页面内容的复制操作，减少相同内容页面在主存中的副本数目。
在创建进程时系统并不复制父进程的完整空间，而仅复制父进程的页表，使父子进程共享物理空间，并把这个共享空间的访问权限设置为"只读"。当其中某个(父或子)进程要修改页面内容以执行写操作时，会产生"写时复制"中断，操作系统处理这个中断信号，为此进程创建一个新页，设置其为可读可写，并将其映射到进程的地址空间，此进程就可以修改复制的页，所有未修改
Unix内存管理最重要的机制之一：copy on write

不太需要关心这个。即使对x86，现在都是在使用基于分页的Flat内存管理了。

页仍可与其他进程共享。可见，采用写时复制技术之后，就可推迟到修改时才对共享页面做出副本，避免对"只读"页的复制，从而减少页面复制操作并节省副本所占用的空间。

4.5.3 请求分段虚拟存储管理

请求分段虚拟存储管理也为用户提供比主存实际容量大得多的虚拟主存空间。请求分段虚拟存储系统把作业的所有分段的副本都存放在辅助存储器中，当作业被调度投入运行时，首先把当前需要的段装入主存，在执行过程中访问到不在主存的段时再将其动态装入。因此，在段表中必须增设供管理使用的若干表项，如特征位、存取权限、扩充位、标志位、主存起始地址和段长等。其中，特征位指示段是否在主存中：0(不在主存)，1(在主存)；存取权限给出段的可访问模式：可执行、可读、可写等；扩充位指出段可否动态扩充：0(固定长度)，1(可扩充)；标志位又分为修改位、访问位、可否移动位等。

在作业执行过程中访问某段时，由硬件的地址转换机制查段表，若所需的段在主存中，则按分段存储管理方法进行地址转换以得到绝对地址；若所需的段不在主存中，触发缺段中断，操作系统处理此中断时，查找主存分配表，找出一个足够大的连续区域容纳此分段；如果找不到足够大的连续区域则检查空闲区的总和，若空闲区总和能满足分段要求，那么，进行适当移动后，将此分段装入主存；若空闲区总和不能满足要求，可调出一个或多个分段到磁盘上，再将此分段装入主存。

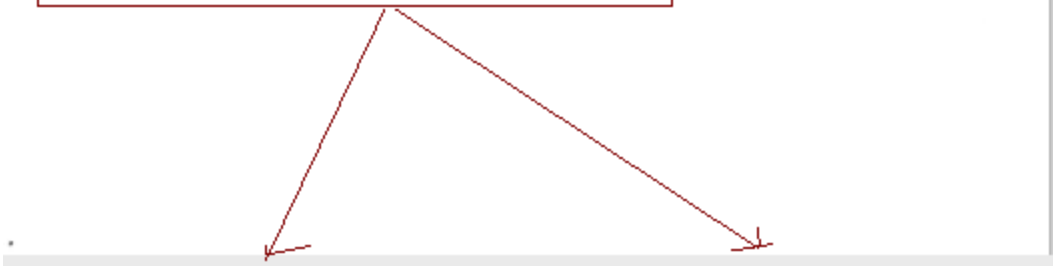
在执行过程中，有些表格或数据段随着输入数据的多少而变化。例如，某个分段在执行期间因表格空间用完而要求扩大分段，这只需在此分段后添置新信息即可，添加后的长度应不超过硬件所允许的每段的最大长度。对于这种变化的数据段，当向其中添加新数据时，由于欲访问的地址超出原有的段长，硬件将产生一个越界中断，系统处理这个中断时，先判别此段的"扩充位"标志，如果可以扩充，则增加段的长度，必要时还要移动或调出分段以腾出主存空间；如果此段不允许扩充，那么，这个越界中断就表示程序出错。缺段中断和段扩充处理流程如图 4.26 所示。

请求分段虚拟存储管理便于实现分段的共享和保护。为了实现分段的共享，除了原有的进程段表外，还要在系统中建立一张段共享表，每个共享分段占一个表项，每个表项包含两部分内容：第一部分含有共享段名、段长、主存起始地址、状态位(如是否在主存位)、磁盘地址、共享进程个数计数器；第二部分含有共享此段的所有进程名、状态、段号、存取控制位(通常为"只读")。分段共享涉及段的动态链接，技术实现比较复杂，可参考 MULTICS 动态链接和共享机制。

分段存储器系统中，每个分段在逻辑上是独立的，实现存储保护时也很方便。一是越界检查，在段表寄存器中存放段长信息，在进程段表中存放每个段的段长，每当存储访问时，首先把指令逻辑地址中的段号与段表长度进行比较，如果段号等于或大于段表长度，将发出地址越界中断；其次，还需检查段内位移是否等于或大于段长，若是，则产生地址越界中断，从而确保每个进程只在自己的地址空间内运行。二是存取控制检查，在段表的每个表项中，均设有存取控制字段，用于规定此段的访问方式，通常设置的访问方式有"只读"、"可读写"、"只执行"等。

段机制在x86里是不能disable的。Paging机制反而可以disable／enable。80386之后才有Paging。

一个地址，需要经过CPU两个部件的依次翻译，然后才把得出的作为物理地址，去内存DRAM中取数据。



粒度不均匀。

4.5.4 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；页式存储是基于系统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。如果把两者的优点结合起来，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拟存储管理的基本原理。

(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页框大小相仿的页面，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，$d' = p \times$ 块长 $+ d$。

(5) 请求段页式虚拟存储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主存

”段内页号“： 每个段都是独立编址的逻辑单元。所以，页号都是“某个段”的页。例如，段A的第x页；段B的第x页。不矛盾。

首先还是看CPU内部的TLB缓存是否有hit。

的信息，及此段页表的起始地址，页表中包含页是否在主存的信息(中断位)、对应的主存块号。

请求段页式虚拟存储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 的页帧号 p'，并与位移 d 一起拼装得到访问主存的实地址，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，由此得到页帧号 p'；这时一方面把 s 段 p 页和页帧号 p' 置换进快表，另一方面用 p' 和 d 生成主存物理地址，完成地址转换。

上述过程假设所需信息都在主存中，事实上，许多情况都会发生，如查段表时，发现 s 段不在主存，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，并将段调入主存；如查页表时，发现 s 段 p 页不在主存，于是产生"缺页中断"，引起操作系统查找 s 段 p 页在磁盘上的位置，并将此页调入主存。当主存已无空闲页帧时，就会导致淘汰页面。如图 4.27 所示是段页式动态地址转换和存储保护示意图。

一个32位的寄存器，或者立即数。

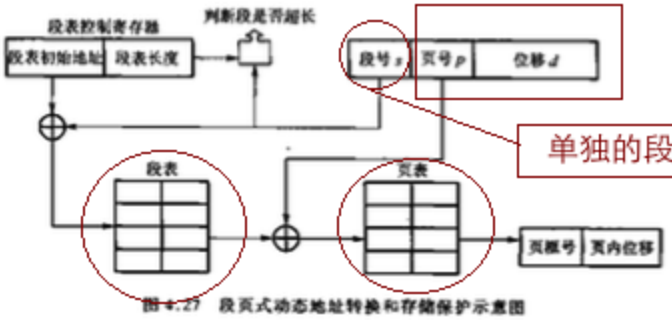


图 4.27 段页式动态地址转换和存储保护示意图

单独的段寄存器

8086

例如，16位的CS／DS／ES等

4.6 Intel x86 分段和分页存储结构

Intel x86 系列 CPU 提供 3 种工作模式：实地址模式、保护模式和虚拟 8086 模式。实地址模式采用段式实存或单一连续分区，无特权级，不启用分页机制，寻址范围为 1 MB；保护模式采用分段机制并可启用分页机制，共有段式、页式、段页式这 3 种虚拟存储管理模式；虚拟 8086 模式是在保护模式下对实地址模式的仿真，允许多个 8086 应用程序同时在 386 以上 CPU 上运行。DOS 在实地址模式下工作，而 Windows 和 Linux 等操作系统在保护模式下运行。

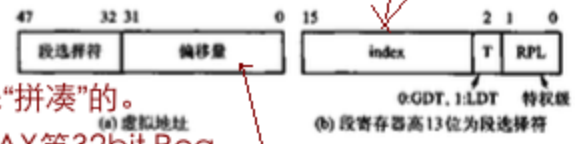
换言之，TLB Entry要有相应的 set，page，page frame的三元组的mapping。

286开始有保护模式
386开始有Paging分页模式

2^13=8K个段Descriptor选择。

Intel x86 实现虚拟存储管理的核心是维护主存中的两张表：局部描述符表(Local Descriptor Table，LDT)和全局描述符表(Global Descriptor Table，GDT)，分别用 GDTR 和 LDTR 两个寄存器指向主存中的描述符表。每个进程都有其私有的 LDT，描述局部于它的分段，包括代码段、数据段、堆栈段、扩充段等的基地址、段大小和有关控制信息等；系统的所有进程共享一个 GDT，描述系统段，包括操作系统的基地址、段大小、相关控制信息和所有软件共享的系统资源。

为了访问一个段，必须把段选择符装入机器的 6 个段寄存器之一，如 CS 寄存器保存代码段选择符，DS 寄存器保存数据段选择符和 SS 寄存器保存堆栈段选择符等。硬件还配有 6 个 8 B 的高速缓存寄存器，存放取自 LDT/GDT 的描述字，每次访问主存时不必都到主存中读取描述字，大大地加快存取速度。控制寄存器 CR0 给出分页标志 PG，保护允许标志 PE 等；CR2 保存缺页中断时所缺页的线性地址；CR3 存放页目录基址。物理地址空间最大为 $2^{32}=4$ GB，但虚存地址空间最大可为 64 TB。为此，定义如图 4.28 所示的虚拟地址。



(a) 虚拟地址　(b) 段寄存器高13位为段选择符

图 4.28 Intel x86 虚拟地址和段选择符

通过两个寄存器来“拼凑”的。

偏移量本身可以是AX等32bit Reg

在 16 位的段寄存器中，第 0、1 位 RPL 是描述符请求的特权级，它不参与段选择；第 2 位 T 为 0 或 1 分别指明选择 GDT 或 LDT；高 13 位 index 用做访问段描述符表中某个描述符的下标。由此可知虚拟地址空间共包含 $2^{14}=16$ K 个存储器分段，其中 GDT 映射一半(8 192 个)全局虚拟地址空间，LDT 映射另一半(8 192 个)局部虚拟地址空间。当发生进程切换时，LDT 更新为待运行进程的 LDT，而 GDT 保持不变。由于段内偏移量 32 位即 $2^{32}=4$ GB，整个虚存地址空间为 $2^{14+32}=64$ TB。

描述符表中的描述符是存储管理硬件 MMU 管理虚存空间分段的一种依据，一个描述符直接对应于虚存空间中的一个主存分段，定义段的基址、大小和属性。GDT 和 LDT 拥有相同格式的描述符，一个段描述符由 8 B 组成，如图 4.29 所示。

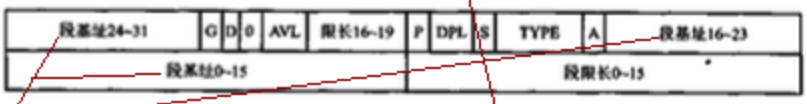


图 4.29 描述符

其中的主要内容有：段基址，共 32 位，加上 32 位偏移量形成线性地址；段限长，共 20 位，限定段描述符寻址的主存段的长度。各种特征位如下：

32为段基址＋32位偏移量。可以完整的跨越4G空间。

可以用来控制 用户态／核心态 的保护。

(1) G 位:描述段长的单位,G=0 表示以字节为单位;G=1 表示以页面为单位,页长固定为 4 KB,于是段的长度分别为 2^{20} B 或 $2^{20}\times 4$ KB $=2^{32}$ B $=4$ GB。

(2) D 位:当 D=1 时,为 32 位代码段;当 D=0 时,为 16 位代码段。

(3) P 位:若为 1,表示段包含有效基址和界限;若为 0,无定义。

(4) DPL 位:指明描述符特权级(0~3)。

(5) S 位:是段内容标志,S 为 1 时,表示代码和数据段描述符;S 为 0 时,表示系统段描述符。

(6) TYPE 字段:表示段类型和保护方式,如可执行代码段、只读数据段等。

(7) A 位:是访问位,0 表示未访问;1 表示访问过,为淘汰页面做准备。

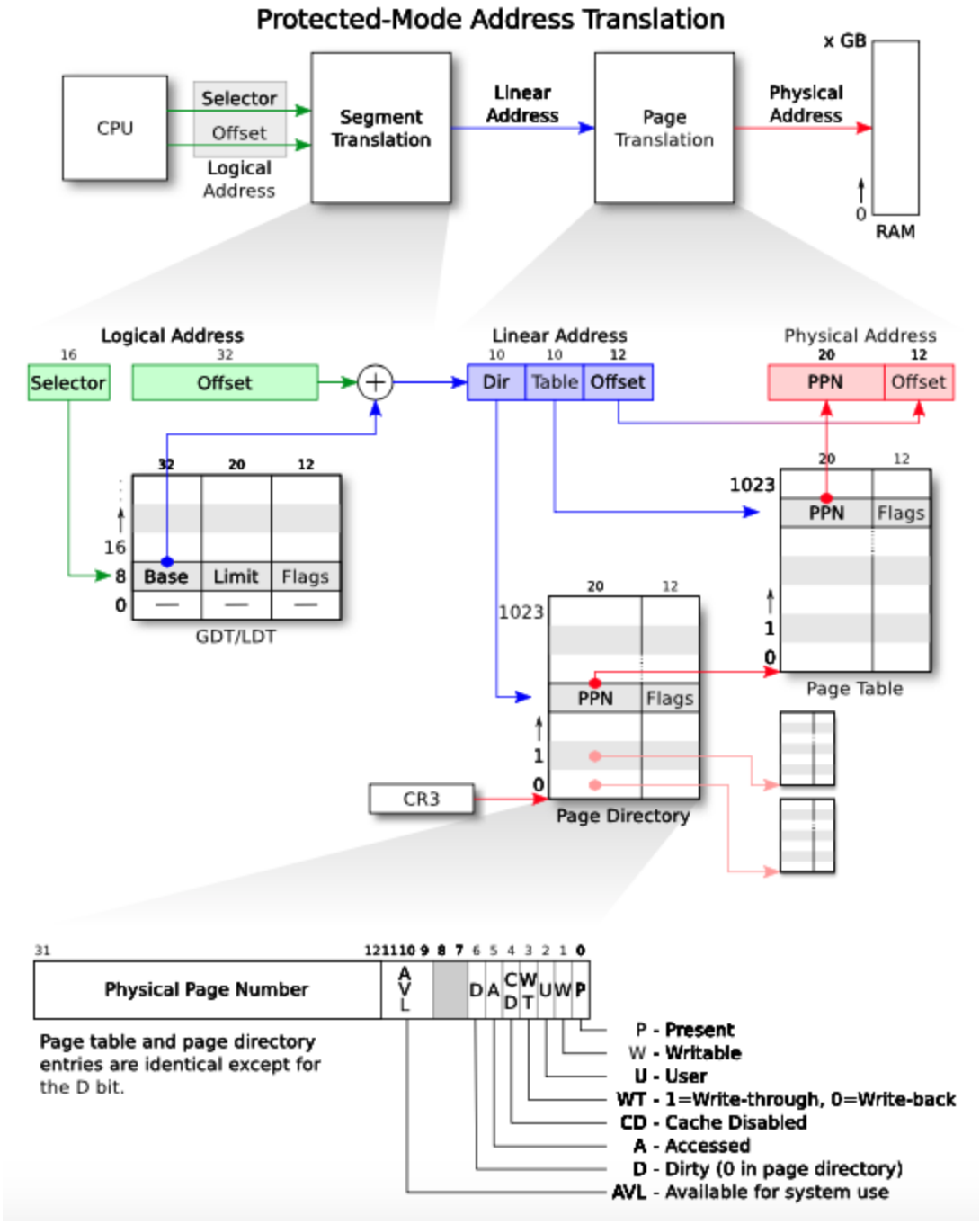
(8) AVL 位:用户编程可用位。

从虚拟地址(16 位选择符 + 32 位偏移量)到物理地址的转换要分两步走,MMU 使用分段机制把 48 位虚拟地址先转换成 32 位线性地址,转换过程是通过描述符表中的描述符来实现的。在保护模式下,段选择符被装入段选择符寄存器时,从选择符的 T 位就知道是选 LDT 还是 GDT,再根据 index 由硬件自动从 LDT 或 GDT 中取出描述符装入段描述符高速缓存寄存器中,以实现 16 位选择符到 32 位段基址的转换,再把描述符中的 32 位段基址与 32 位偏移量相加以形成 32 位线性地址。

如果控制寄存器 CR0 的 PE 位为 1,处理器工作在保护模式下,而 CR0 的控制寄存器的 PG 位为 0 时,表明此时禁止分页,那么,线性地址就是访问存储器的物理地址,这是纯分段结构。如果把 6 个段寄存器设置为同一个段选择符来实现不分段(实际仅有一段),且段描述符基址设置为 0,段大小设置为最大值 4 GB,此时是单段且也有 CR0 的 PE 位为 1,同时 PG 位为 1,系统运行于分页模式,32 位地址空间中单段分页运行,实际上就是纯分页结构。

当 PG 为 1 时,启用分页机制,此时线性地址并不是最终访问的物理地址,需要通过分页机制进行第二次地址转换。由分段得到的线性地址又分成 3 个域:10 位页目录 dir、10 位页 page 和 12 位偏移量 offset。在进行页转换时,根据控制寄存器 CR3 给出的页目录表的起始地址,用 dir 作为索引在页目录表中找到指向页表的起始地址,再用 page 作为索引在页表中找到页框起始地址,最后,把偏移量加到页框起始地址上,得到访问单元的物理地址。页目录表和页表的结构是类似的,均为 32 位,其中左边 20 位是页表或页框的基地址,保证页表或页框对齐 4 KB 边界。页表表项的右 12 位包括由硬件设置的供操作系统使用的中断位、引用位、修改位、保护位和其他一些有用的位。当从页表中读出页表项时,就被硬件同时装入快表,以便继续引用时无须再次查找页表。Intel x86 存储管理机制的设计非常巧妙,实现了避免互相冲突的目标:段页式存储管理、纯页式存储管理、纯段式存储管理,并能够同实地址模式兼容。图 4.30 给出最复杂的保护模式下段页式虚拟地址到物理地址转换的过程。

Flat Memory Model。通过把段基址对齐为0，并启动分页。


Protected-Mode Address Translation
CPU
Selector
Offset
Logical Address
Segment Translation
Linear Address
Page Translation
Physical Address
x GB
0
RAM
Logical Address
16
32
Selector
Offset
Linear Address
10
10
12
Dir
Table
Offset
Physical Address
20
12
PPN
Offset
32
20
12
16
8
0
Base
Limit
Flags
GDT/LDT
1023
20
12
PPN
Flags
1
0
Page Table
1023
20
12
PPN
Flags
1
0
CR3
Page Directory
31
12 11 10 9 8 7 6 5 4 3 2 1 0
Physical Page Number
AVL
D
A
CD
WT
U
W
P
Page table and page directory entries are identical except for the D bit.
P - Present
W - Writable
U - User
WT - 1=Write-through, 0=Write-back
CD - Cache Disabled
A - Accessed
D - Dirty (0 in page directory)
AVL - Available for system use

操作系统存储管理与相应的CPU体系结构关系比较紧密。x86，MIPS，ARM，PowerPC都有着不同的区别。有的CPU系列即使是不同的市场目标，都有一些区别。但总体概念和机制是一样的。通过Paging／MMU来实现逻辑地址到虚拟地址的变换。在变换过程中，由于有了系统软件Page Table的参与和干涉，可以提供许多保护的选项。

缺页异常是操作系统设计需要小心避免的。特别是通信系统。

段机制基本上不用了。都使用分页机制。或者说Flat Memory Model了。

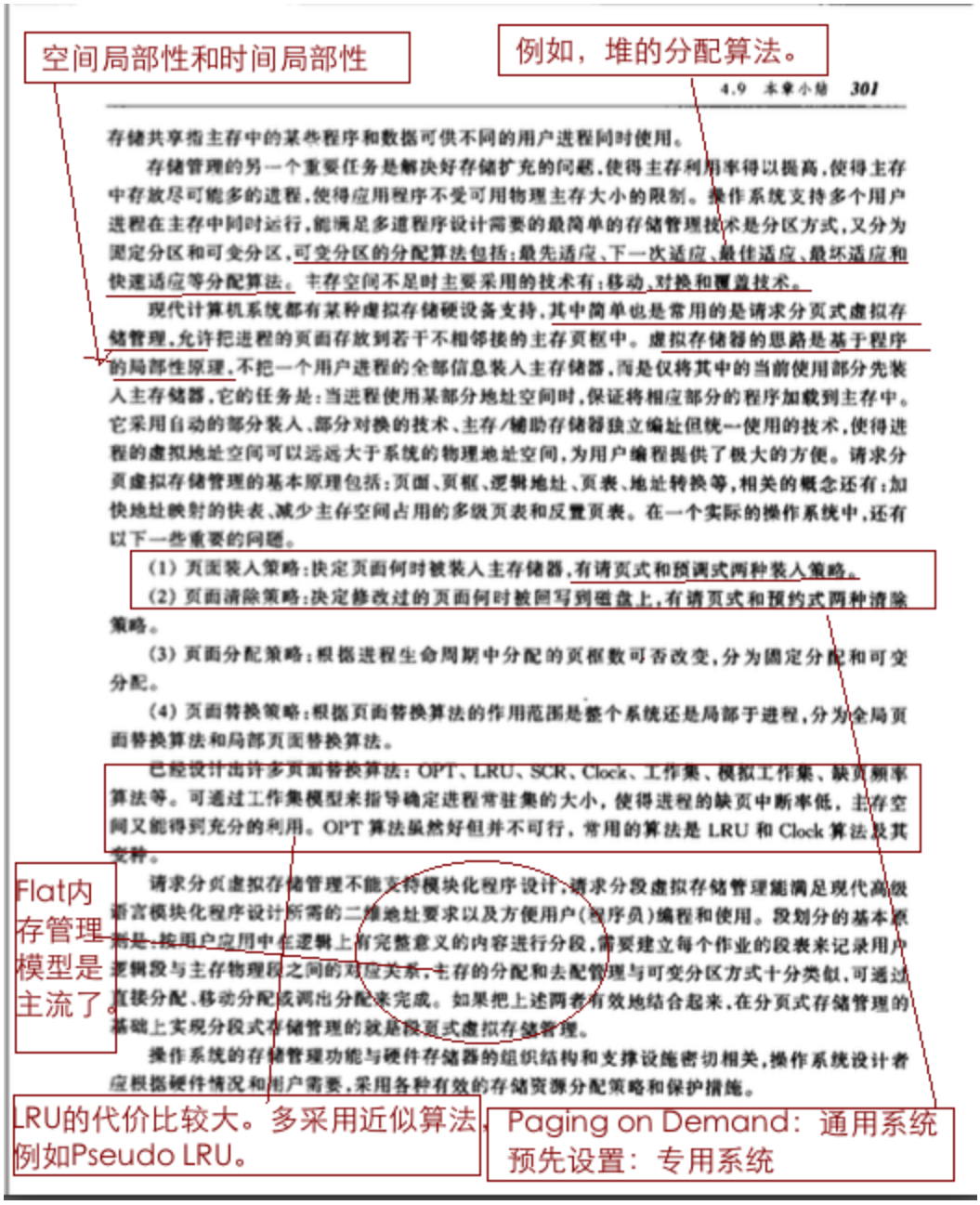
除了Page Table机制，许多现代CPU也提供一些大Page或者大范围的虚拟映射机制，例如，Power／PowerPC的BAT机制。

任何一种页面替换算法都有局限性。没有最优。

本章小结

操作系统存储管理的基本功能有：存储分配、地址转换和存储保护、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主存空间；地址转换是把逻辑地址空间中的应用程序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，以便应用程序执行；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；

虚拟内存机制带来的保护，提供了进程间的隔离，非常重要！



[注释]：内存管理与CPU联系比较紧密。不同的CPU都略微不同。对通用系统而言，Paging on demand；copy on write是操作系统算法层面比较常用的。但对专用系统并不合适；另外，由于启动虚拟到物理的映射，使得操作系统存在一个机会干预内存的分配，保护，从而可以指导策略。