第4章 存储体系

- 主要内容包括：
- 几种常用的内存管理方法（分区、分页、分段）
- 内存的分配和释放算法（最先适应、最佳适应、最坏适应、临近适应）
- 虚拟存储器的概念（部分装入）
- 控制主存和外存之间的数据流动方法
- 地址变换技术和内存数据保护与共享技术等
- Windows2000/xp内存管理

- 存储器是计算机系统的重要资源之一。
- 　任何程序和数据以及各种控制用的数据结构都必须占用一定的存储空间
- 存储器由内存（primary storage）和外存（secondary storage）组成。
- 存储管理是指存储器资源（主要指内存并涉及外存）的管理。
- 内存由顺序编址的块组成，每块包含相应的物理单元。

存储层次结构

- 微机中的存储层次组织：
  - –访问速度越慢，容量越大，价格越便宜；
  - –最佳状态应是各层次的存储器都处于均衡的繁忙状态

存储管理的功能

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

虚拟存储器(VIRTUAL MEMORY)

- 内存价格昂贵，不可能用大容量的内存存储所有被访问的或不被访问的程序与数据段。
- 外存尽管访问速度较慢，但价格便宜，适合于存放大量信息。
- 存储管理系统把进程中那些不经常被访问的程序段和数据放入外存中，待需要访问它们时再将它们调入内存。

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

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

虚拟存储器的原理

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

虚拟存储技术的特征

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

- 用户编写的源程序，首先要由编译程序编译成CPU 可执行的目标代码。然后，链接程序把一个进程的不同程序段链接起来以完成所要求的功能。
- 不同的程序段，应具有不同的地址。如何安排这些编译后的目标代码的地址。

地址变换

- 内存地址的集合称为内存空间或物理地址空间。内存空间是一维线性空间。
- 几个虚存的一维线性空间或多维线性空间变换到内存的唯一的一维物理线性空间
- 一个是虚拟空间的划分问题。例如进程的正文段和数据段应该放置在虚拟空间的什么地方。虚拟空间的划分使得编译链接程序可以把不同的程序模块，链接到一个统一的虚拟空间中去。

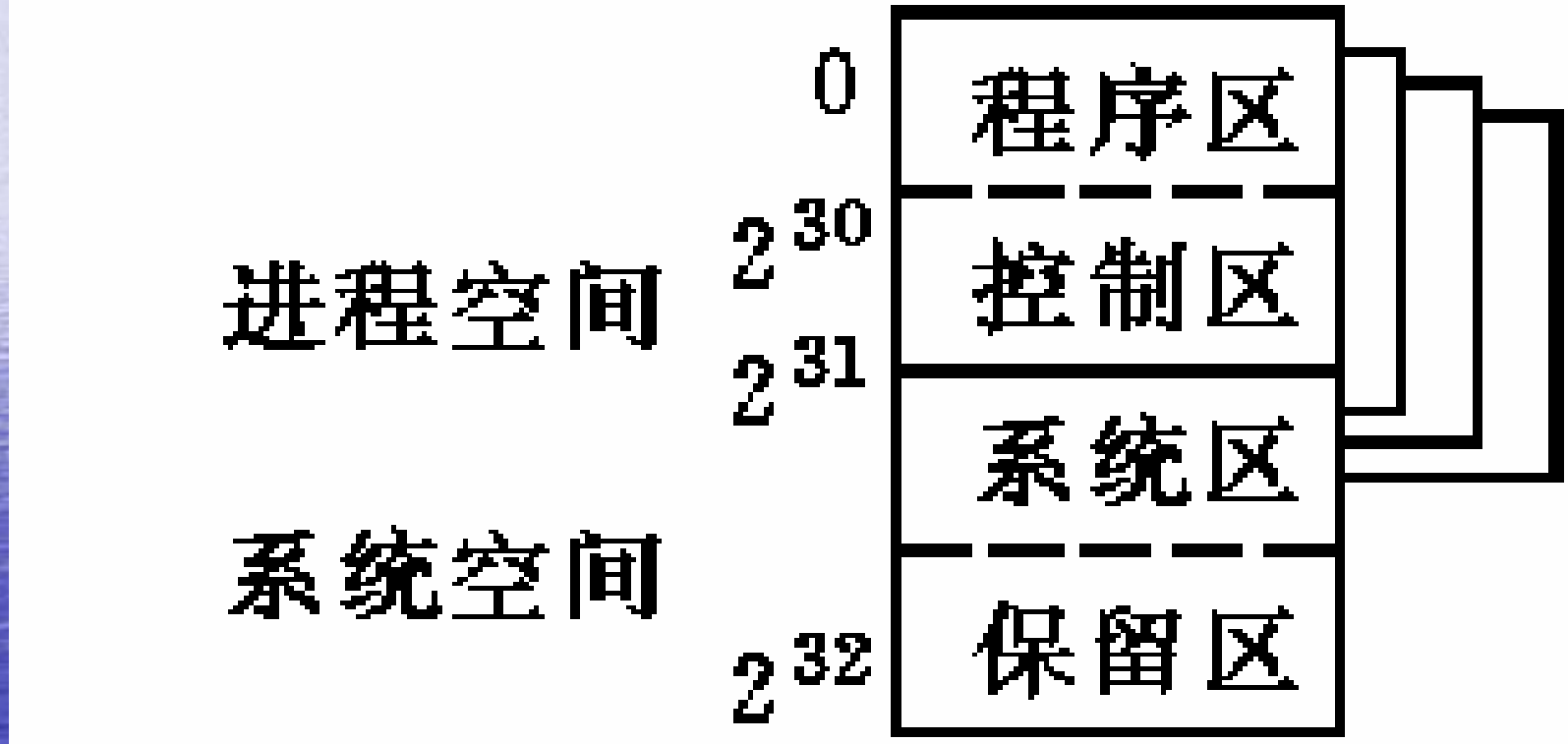


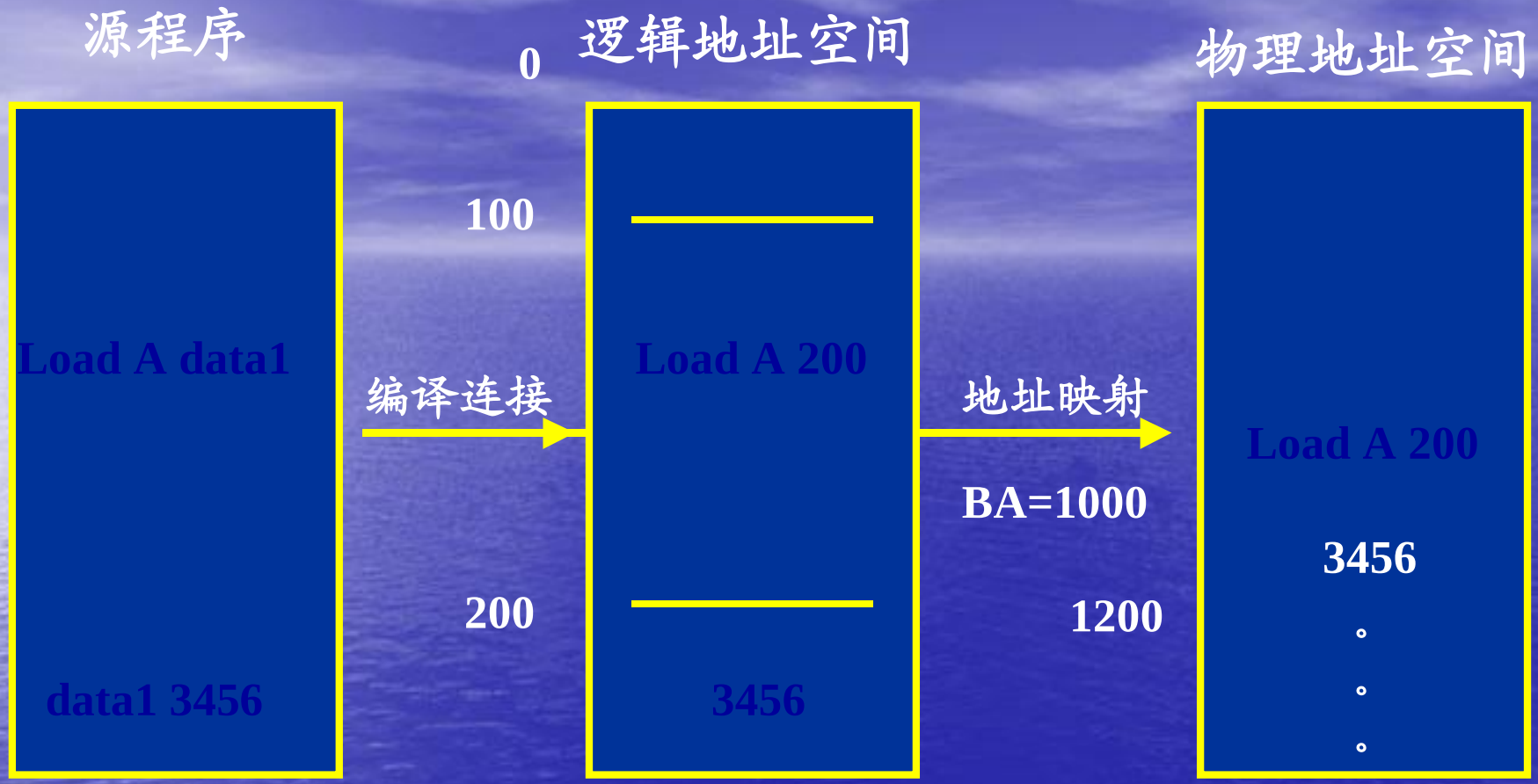
图5.2 虚拟空间的划分

重定位方法

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

逻辑地址、物理地址和地址映射

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



逻辑地址、物理地址和地址映射

重定位方法（地址映射）

- 虚拟地址与内存地址的关系

重定位：

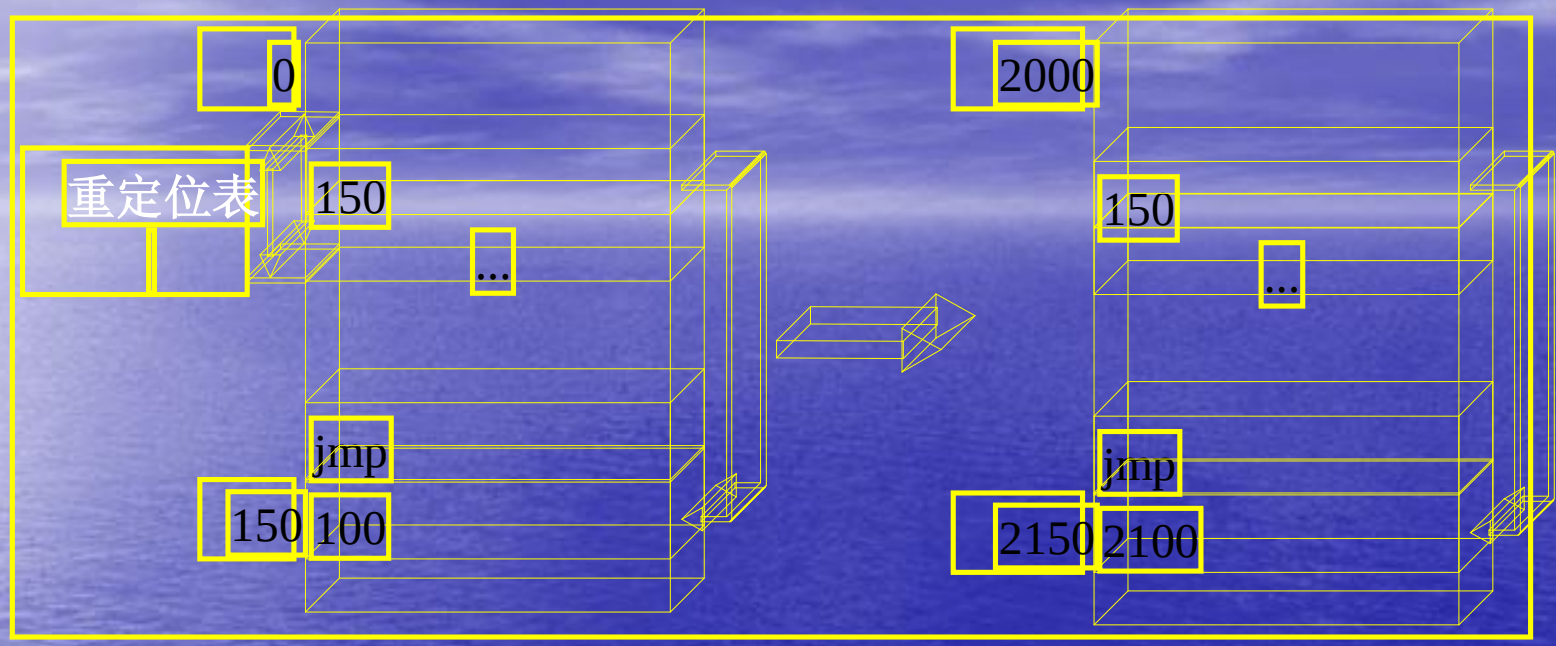
1. 确定一个待执行程序在内存的位置
2. 将程序中的逻辑地址转换成物理地址

- (1)静态地址重定位
- 静态地址重定位是在程序执行之前由操作系统的重定位装入程序完成的。
- (2)动态地址重定位
- 动态地址重定位是在程序执行期间进行的。

静态地址重定位可执行文件结构

- 在可执行文件中，列出各个需要重定位的地址单元和相对地址值(文件头有一个重定位地址表)。
- 装入时根据所定位的内存地址去修改每个重定位地址项，添加相应偏移量。

静态地址重定位（续）



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

静态重定位的缺点：

- 可以装入的程序道数受限；
- 静态重定位方法将程序一旦装入内存之后就不能再移动且必须在程序执行之前将有关部分全部装入。
- 使用静态重定位方法进行地址变换无法实现虚拟存储器。
- 必须占用连续的内存空间和难以做到程序和数据的共享

动态地址重定位

- 动态地址重定位是在程序执行过程中，在CPU访问内存之前，将要访问的程序或数据地址转换成内存地址。
  - 动态重定位依靠硬件地址变换机构完成。
  - 地址重定位机构需要一个(或多个)基地址寄存器BR和一个(或多个)程序虚地址寄存器VR。指令或数据的内存地址MA与虚地址的关系为：MA＝(BR)十(VR)

    这里，(BR)与(VR)分别表示寄存器BR与VR中的内容

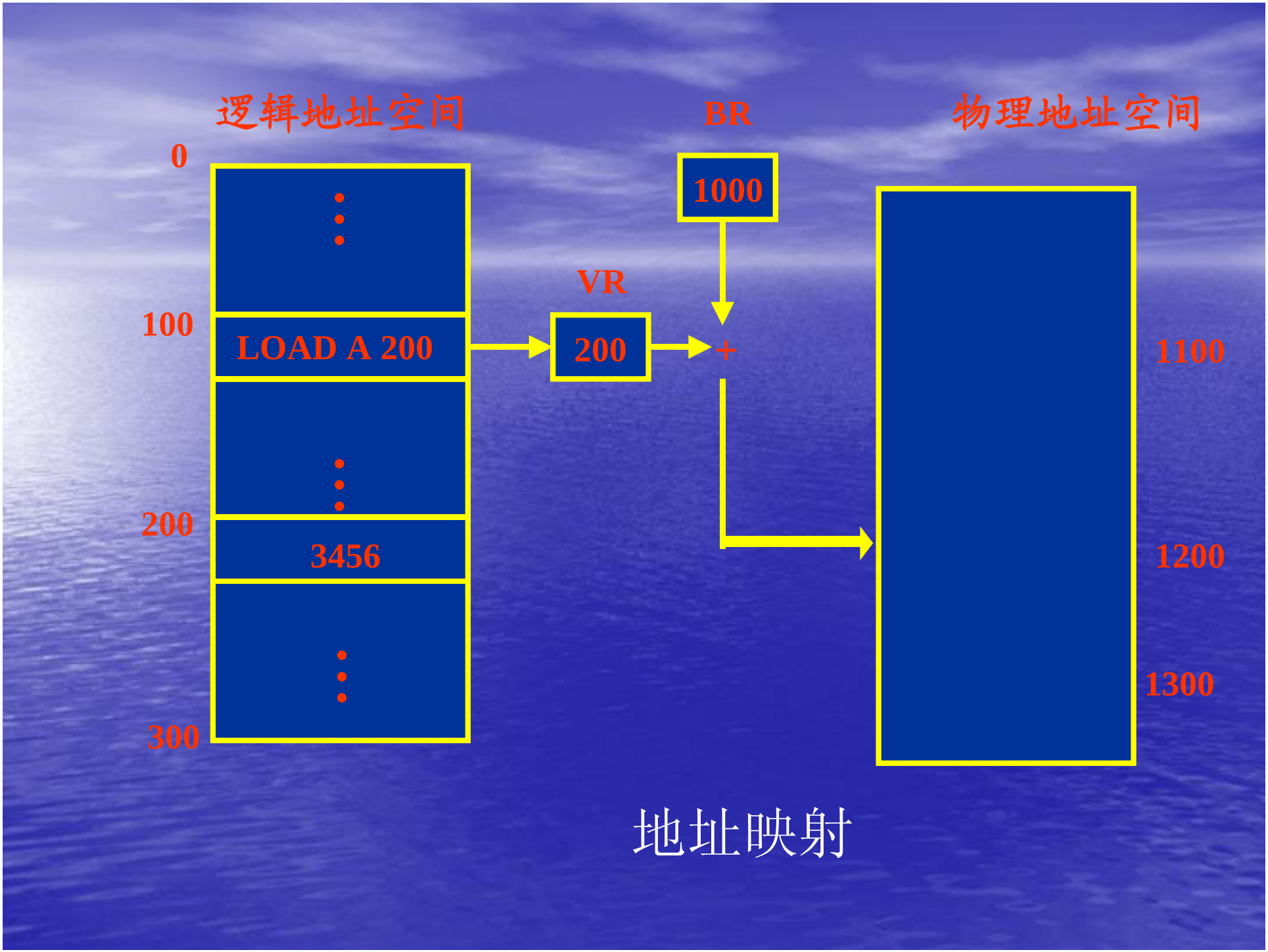
其具体过程是：

(1)设置基地址寄存器BR，虚地址寄存器VR。

(2)将程序段装入内存，且将其占用的内存区首地址送(BR)中。(BR)＝1000。

(3)在程序执行过程中、将所要访问的虚地址送入VR中，例如在上图中执行LOAD A 500语句时，将所要访问的虚地址500放入VR中。

(4)地址变换机构把VR和BR的内容相加，得到实际访问的物理地址。


逻辑地址空间
BR
物理地址空间
0
100
LOAD A 200
200
3456
300
VR
200
1000
+
1100
1200
1300

地址映射

动态重定位的主要优点有：

(1)可以对内存进行非连续分配。显然，对于同一进程的各分散程序段，只要把它们在内存中的首地址统一存放在不同的BR中，则可以由地址变换机构变换得到正确的待访问内存地址。

(2)将程序装入内存之后仍可再移动。

(3)动态重定位提供了实现虚拟存储器的基础。因为动态重定位不要求在作业执行前为所有程序分配内存，也就是说、可以部分地、动态地分配内存。

(4)有利于程序段的共享。

单一连续区存储管理

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

内外存数据传输的控制

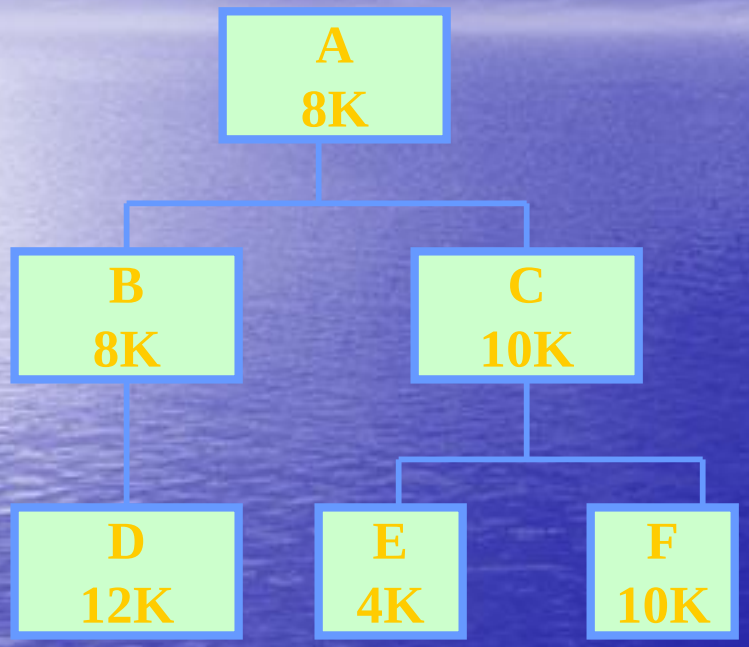
- 把那些即将执行的程序和数据段调入内存，而把那些处于等待状态的程序和数据段调出内存。
- 最基本的控制办法有两种。一种是用户程序自己控制，另一种是操作系统控制。

- 用户程序自己控制内外存之间的数据交换的例子是覆盖。
- 覆盖技术要求用户清楚地了解程序的结构，并指定各程序段调入内存的先后次序。
- 覆盖是一种早期的主存扩充技术。使用覆盖技术，用户负担很大，且程序段的最大长度仍受内存容量限制。因此，覆盖技术不能实现虚拟存储器。

覆盖技术

- 把程序划分为若干个功能上相对独立的程序段，按照其自身的逻辑结构将那些不会同时执行的程序段共享同一块内存区域
- 程序段先保存在磁盘上，当有关程序段的前一部分执行结束，把后续程序段调入内存，覆盖前面的程序段（内存“扩大”了）
- 覆盖：一个作业的若干程序段，或几个作业的某些部分共享某一个存储空间
- 一般要求作业各模块之间有明确的调用结构，程序员要向系统指明覆盖结构，然后由由操作系统完成自动覆盖

覆盖技术例（图）



作业X的调用结构

操作系统控制方式又可进一步分为两种。一种是交换(swapping)方式，另一种是请求调入(on demand)方式和预调入(on prefetch)方式。

请求调入方式是在程序执行时，如果所要访问的程序段或数据段不在内存中，则操作系统自动地从外存将有关的程序段和数据段调入内存的一种操作系统控制方式。

而预调入则是由操作系统预测在不远的将来会访问到的那些程序段和数据段部分，并在它们被访问之前系统选择适当的时机将它们调入内存的一种数据流控制方式。

交换

- 引入：多个程序并发执行，可以将暂时不能执行的程序送到外存中，从而获得空闲内存空间来装入新程序，或读入保存在外存中而目前到达就绪状态的进程。交换单位为整个进程的地址空间。
  - 程序暂时不能执行的可能原因：处于阻塞状态，低优先级（确保高优先级程序执行）；

- 原理：暂停执行内存中的进程，将整个进程的地址空间保存到外存的交换区中（换出swap out），而将外存中由阻塞变为就绪的进程的地址空间读入到内存中，并将该进程送到就绪队列（换入swap in）

例子：

- 只要不用就换出（很少再用）
- 只在内存空间不够或有不够的危险时换出

分时系统，时间片轮转法或基于优先数的调度算法，在选择换出进程时，要确定换出的进程是要长时间等待的

与覆盖技术相比，交换技术不要求用户给出程序段之间的逻辑覆盖结构；而且，交换发生在进程或作业之间，而覆盖发生在同一进程或作业内。此外，覆盖只能覆盖那些与覆盖段无关的程序段

只有请求调入方式和预调入方式可以实现进程大小不受内存容量限制的虚拟存储器。

内存的分配与回收

存储管理模块要为每一个并发执行的进程分配内存空间。另外，当进程执行结束之后，存储管理模块又要及时回收该进程所占用的内存资源，以便给其他进程分配空间。

分配和回收的策略和数据结构

(1)分配结构：用来登记内存使用情况和供分配程序使用的表格与链表。例如内存空闲区表、空闲区队列等。

(2)放置策略：用来确定调入内存的程序和数据在内存中的放置位置。这是一种选择内存空闲区的策略。

(3)交换策略：在需要将某个程序段和数据调入内存时，如果内存中没有足够的空闲区，交换策略被用来确定把内存中的哪些程序段和数据段调出内存，以便腾出足够的空间。

(4)调入策略：外存中的程序段和数据段什么时间按什么样的控制方式进入内存。调入策略与前一节中所述内外存数据流动控制方式有关。

(5)回收策略：回收策略包括二点．一是回收的时机是对所回收的内存空闲区和已存在的内存空闲区的调整。

内存信息的共享与保护

在多道程序设计环境下，内存中的许多用户或系统程序和数据段可供不同的用户进程共享。这种资源共享将会提高内存的利用率。

但是，反过来说，除了被允许共享的部分之外，又要限制各进程只在自己的存储区活动，各进程不能对别的进程的程序和数据段产生干扰和破坏，因此须对内存中的程序和数据段采取保护措施。

常用的内存信息保护方法有硬件法、软件法和软硬件结合三种。

上下界保护法是一种常用的硬件保护法。上下界存储保护技术要求为每个进程设置一对上下界寄存器。上下界寄存器中装有被保护程序和数据段的起始地址和终止地址。

在程序执行过程中，检查经过重定位后的内存地址是否在上、下界寄存器所规定的范围之内。

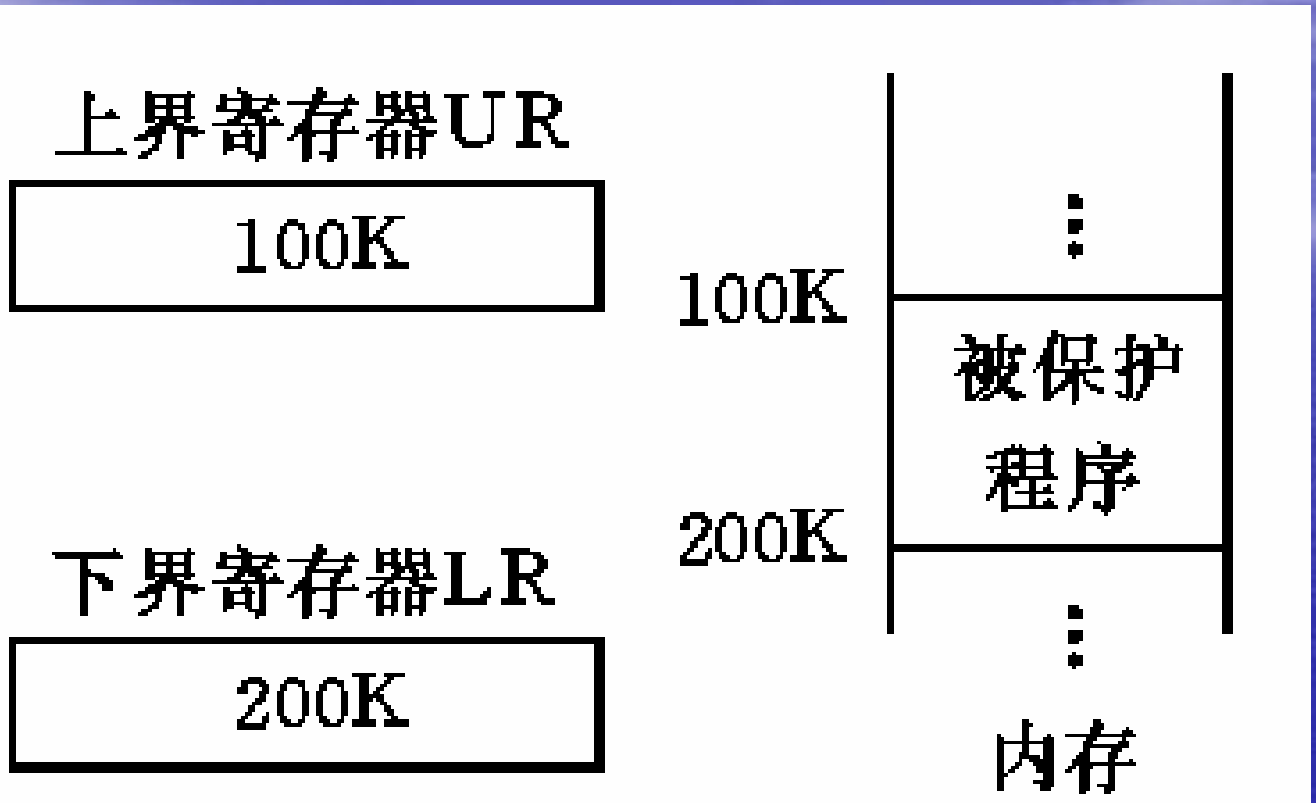


图5.4 上、下界寄存器保护法

保护键法也是一种常用的存储保护法。保护键法为每一个被保护存储块分配一个单独的保护键。在程序状态字中则设置相应的保护键开关字段，对不同的进程赋予不同的开关代码和与被保护的存储块中的保护键匹配。

保护键可设置成对读写同时保护的或只对读，写进行单项保护的。例如，图5.5中的保护键0，就是对2K到4K的存储区进行读写同时保护的，而保护键2则只对4K到6K的存储区进行写保护。如果开关字与保护键匹配或存储块未受到保护，则访问该存储块是允许的，否则将产生访问出错中断。

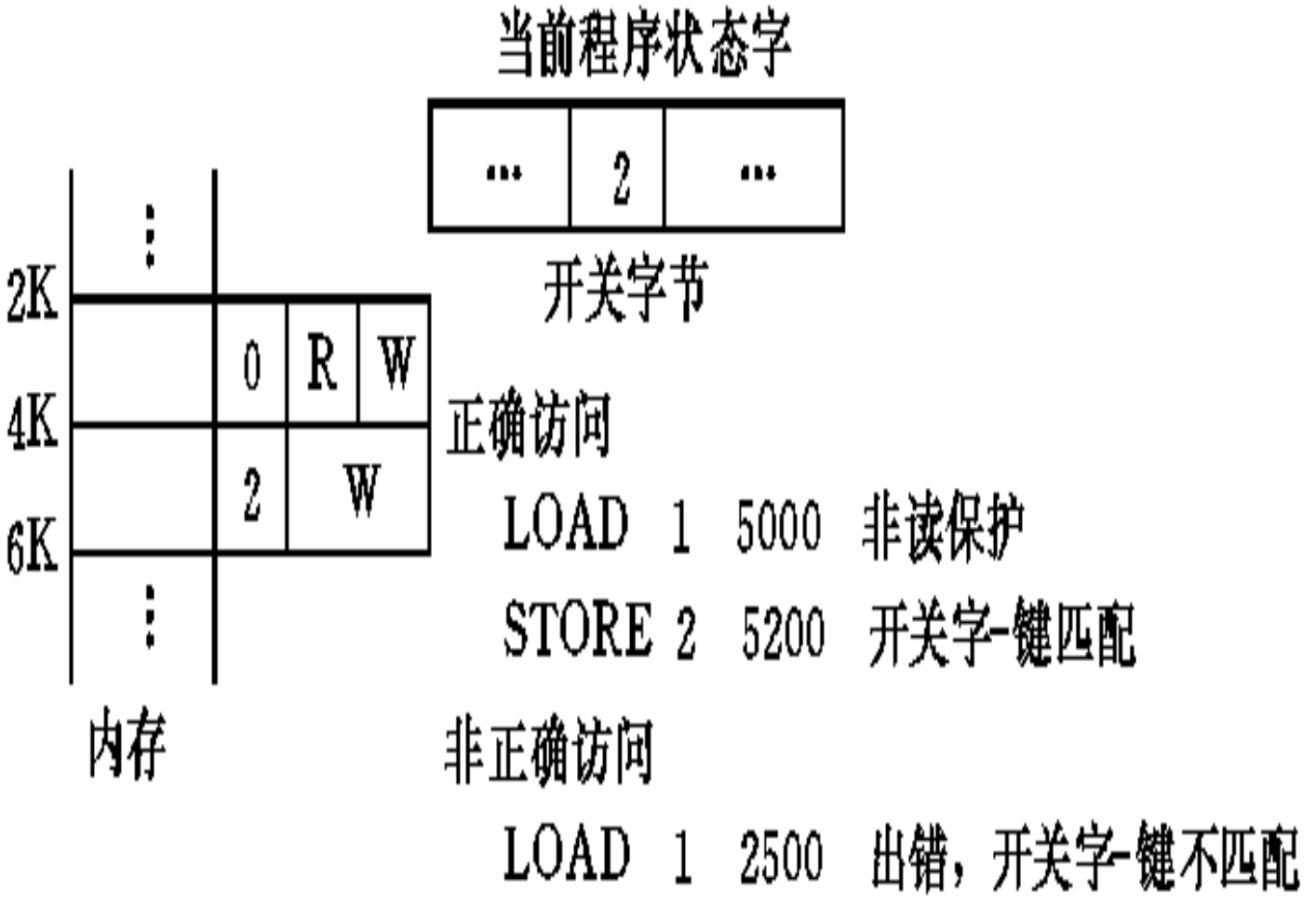

当前程序状态字
... 2 ...
开关字节
2K
4K
6K
0 R W
2 W
内存
正确访问
LOAD 1 5000 非读保护
STORE 2 5200 开关字-键匹配
非正确访问
LOAD 1 2500 出错，开关字-键不匹配

图5.5 保护键保护法

另外一种常用的内存保护方式是：界限寄存器与CPU的用户态或核心态工作方式相结合的保护方式。

在这种保护模式下，用户态进程只能访问那些在界限寄存器所规定范围内的内存部分，而核心态进程则可以访问整个内存地址空间。UNIX系统就是采用的这种内存保护方式。

分区存储管理

- 把内存分为一些大小相等或不等的分区(partition)，每个应用进程占用一个或几个分区。操作系统占用其中一个分区。
- 特点：适用于多道程序系统和分时系统
  - 支持多个程序并发执行
  - 难以进行内存分区的共享。
- 问题：可能存在内碎片和外碎片。
  - 内碎片：占用分区之内未被利用的空间
  - 外碎片：占用分区之间难以利用的空闲分区（通常是小空闲分区）。

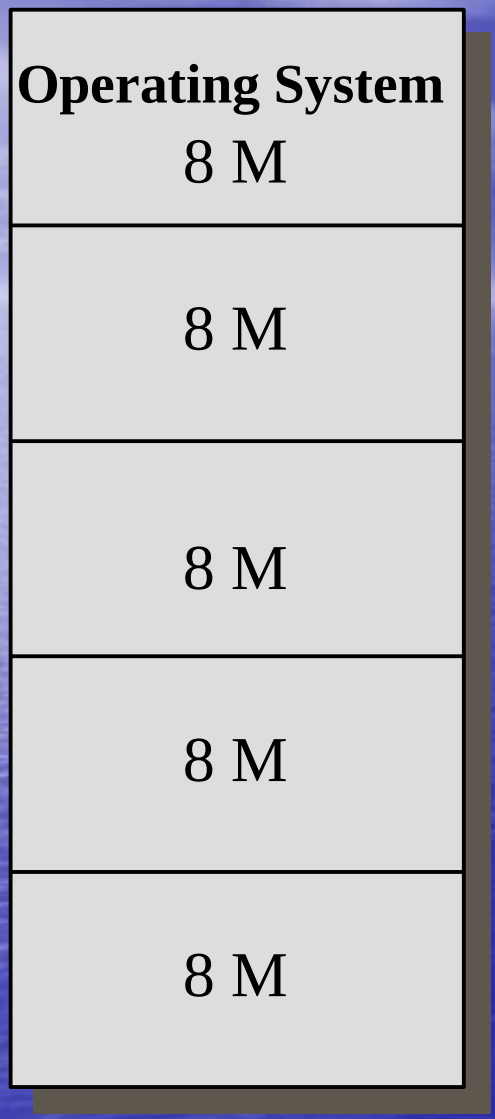
分区管理基本原理

分区管理的基本原理是给每一个内存中的进程划分一块适当大小的存储区,以连续存储各进程的程序和数据，使各进程得以并发执行。按分区的时机，分区管理可以分为固定分区和动态分区两种方法。

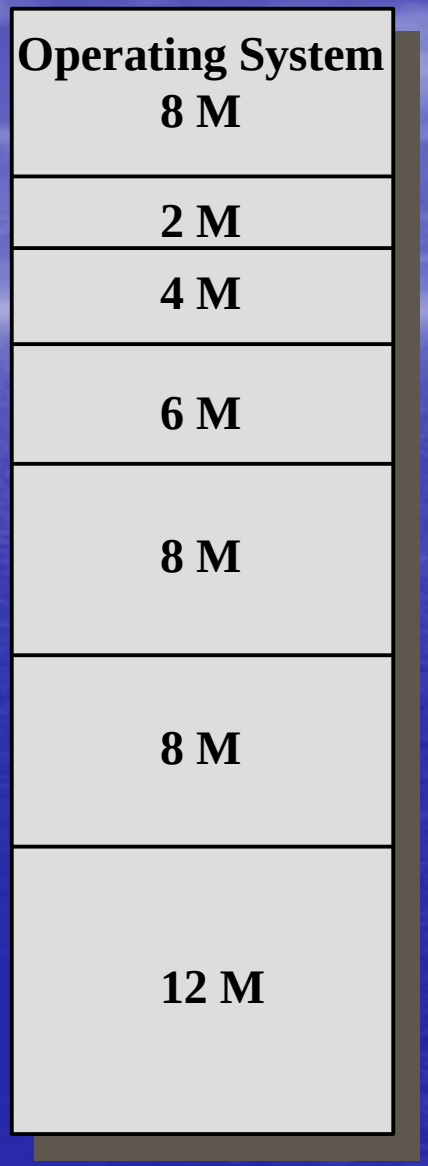
1. 固定分区法□

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

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



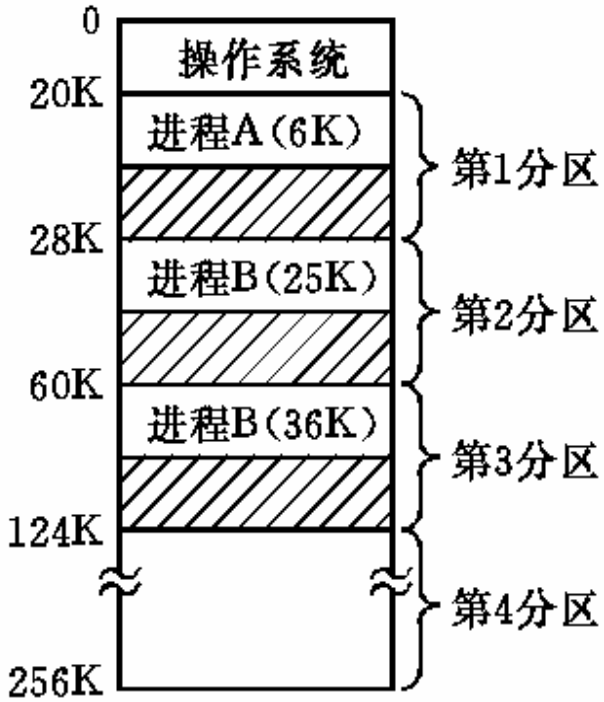
固定分区(大小相同)



固定分区(多种大小)

区号	分区长度	起始地址	状态
1	8K	20K	已分配
2	32K	28K	已分配
3	64K	60K	已分配
4	132K	124K	已分配

(a) 分区说明表



(b) 内存状态

图5.6 固定分区法

2. 动态分区法

动态分区法在作业执行前并不建立分区，分区的建立是在作业的处理过程中进行的，且其大小可随作业或进程对内存的要求而改变。这就改变了固定分区法中那种即使是小作业也要占据大分区的浪费现象，从而提高了内存的利用率

采用动态分区法，在系统初启时，除了操作系统中常驻内存部分之外，只有一个空闲分区。随后，分配程序将该区依次划分给调度选中的作业或进程。图5.7给出了FIFO调度方式时的内存初始分配情况。

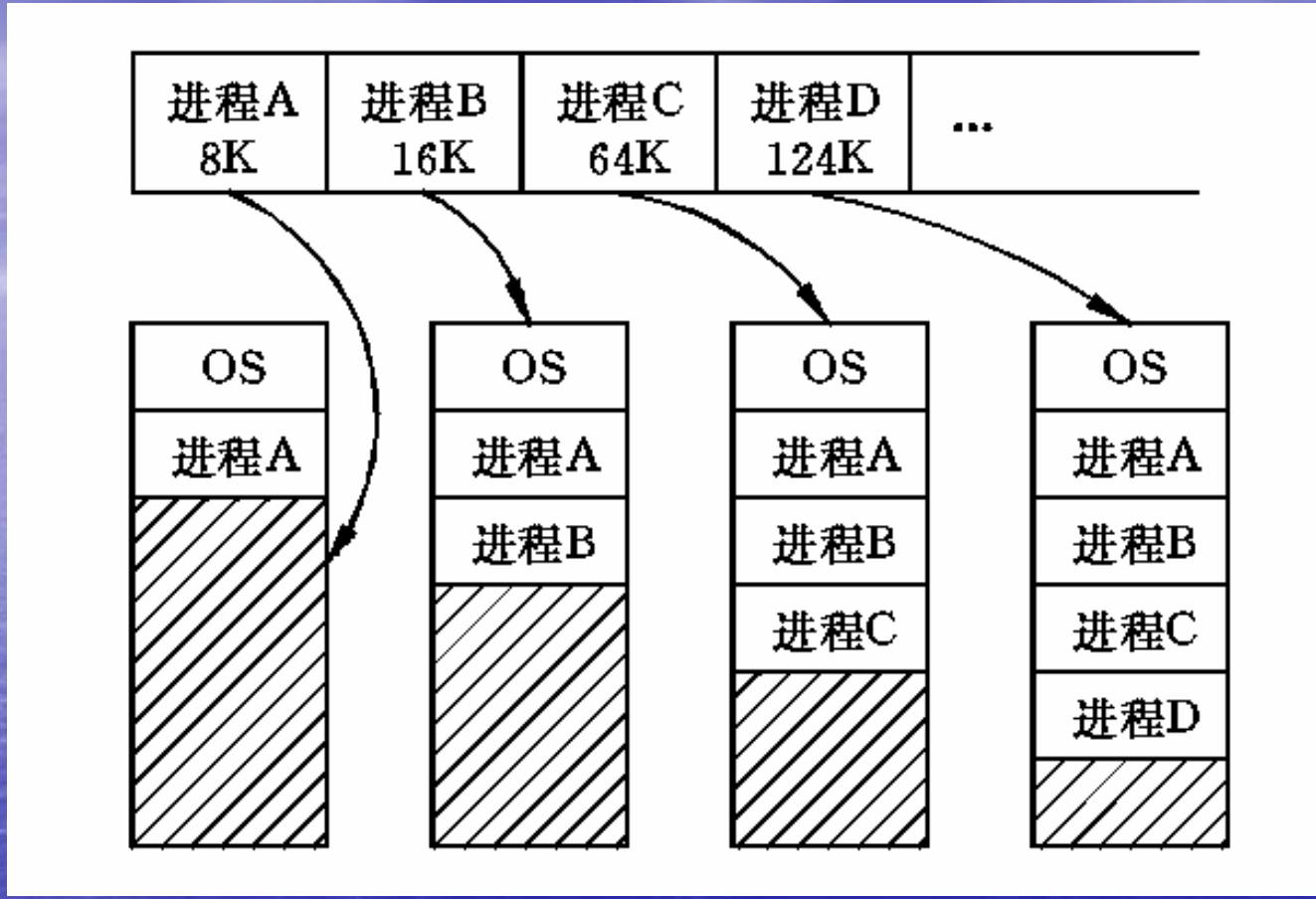


图5.7 内存初始分配情况

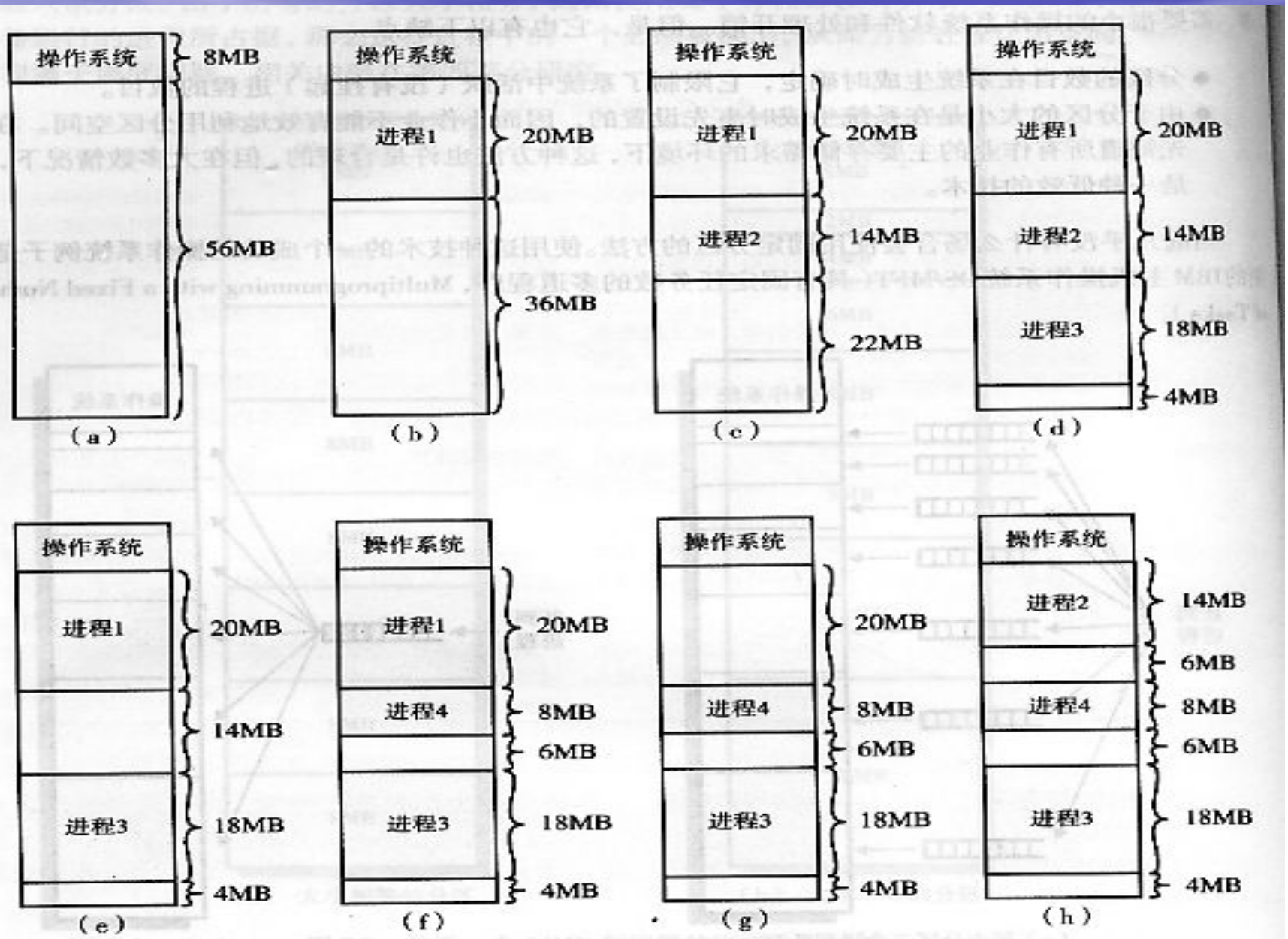

操作系统
8MB
56MB
(a)
操作系统
进程1
20MB
36MB
(b)
操作系统
进程1
20MB
进程2
14MB
22MB
(c)
操作系统
进程1
20MB
进程2
14MB
进程3
18MB
4MB
(d)
操作系统
进程1
20MB
14MB
进程3
18MB
4MB
(e)
操作系统
进程1
20MB
进程4
8MB
6MB
进程3
18MB
4MB
(f)
操作系统
20MB
进程4
8MB
6MB
进程3
18MB
4MB
(g)
操作系统
进程2
14MB
6MB
进程4
8MB
6MB
进程3
18MB
4MB
(h)

图 7.4　动态分区的结果

- 开始很好，最后在存储器出现许多空洞。随着时间的推移，存储器产生了越来越多的碎片。（外部碎片）

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

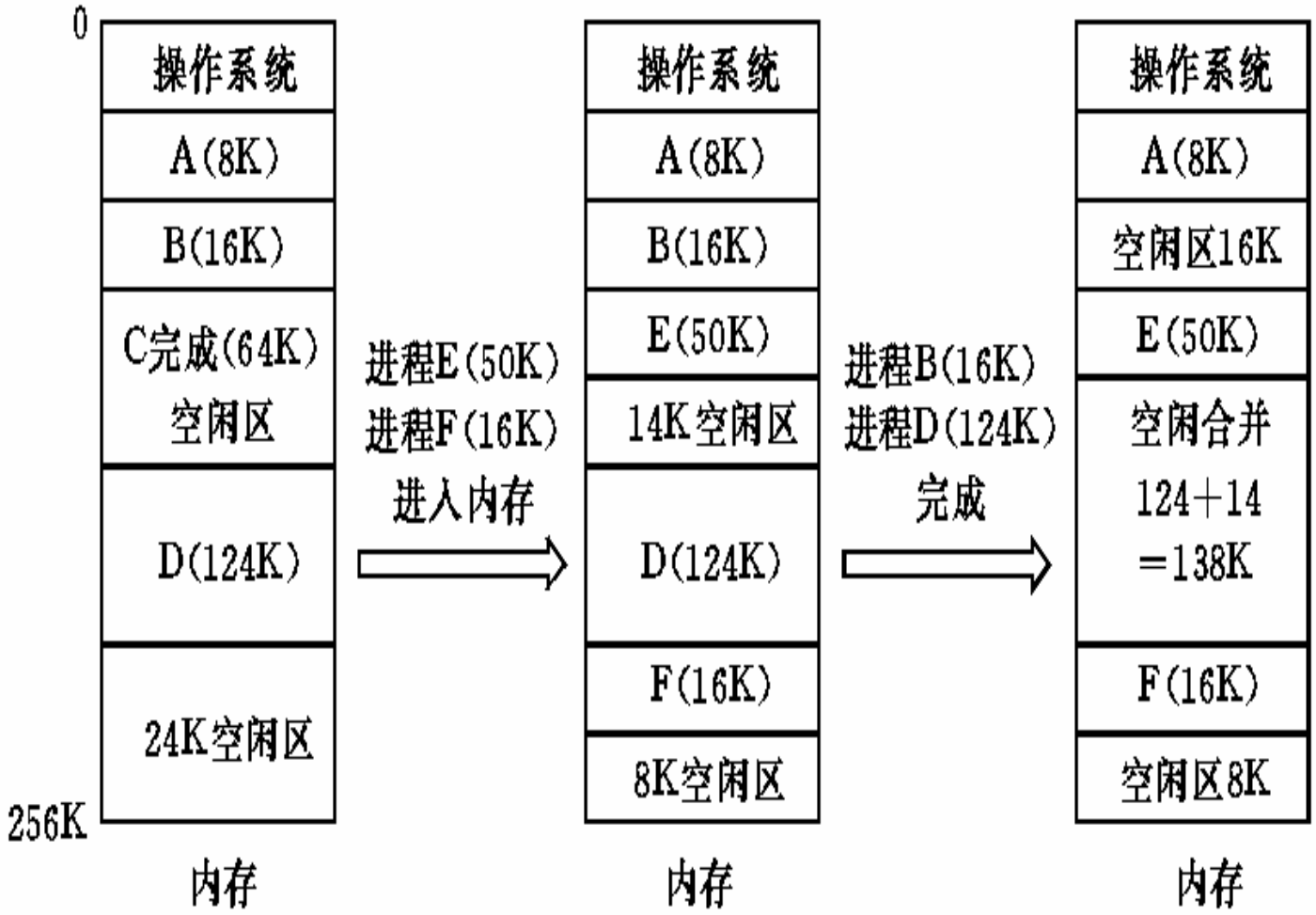
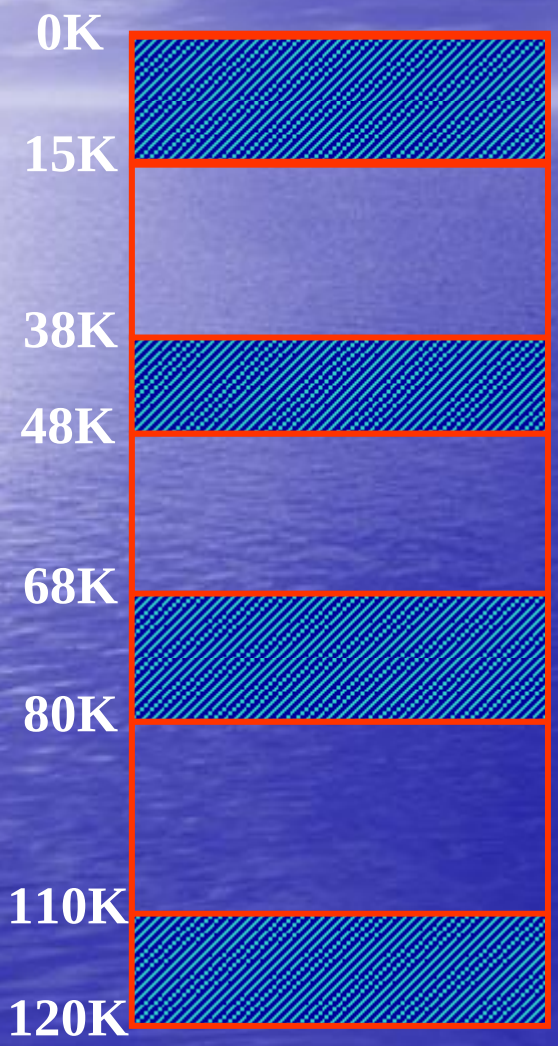


图5.8 内存分配变化过程

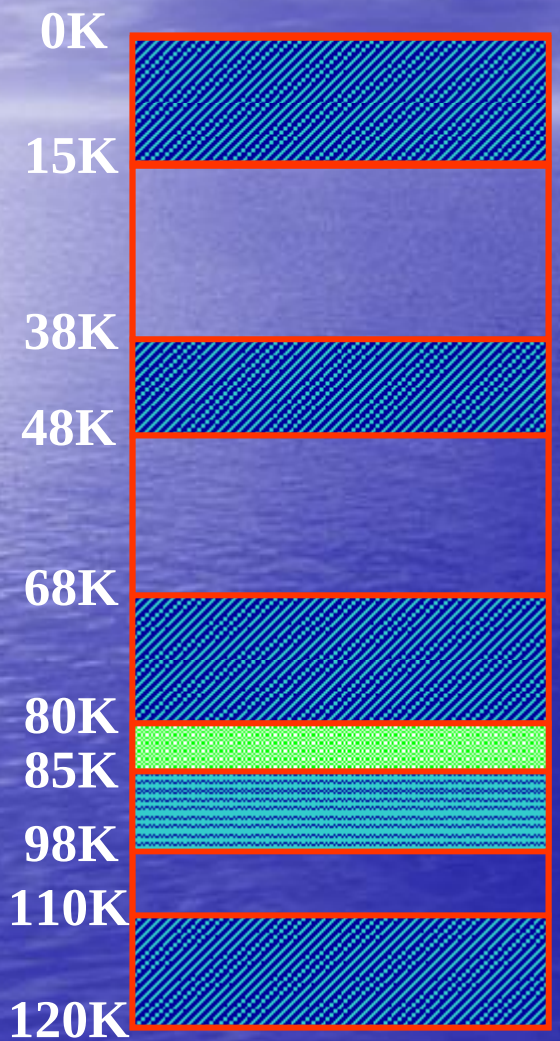


空闲区表

始址	长度	标志
15K	23K	未分配
48K	20K	未分配
80K	30K	未分配
		空
		空

已分配区表

始址	长度	标志
0K	15K	J1
38K	10K	J2
68K	12K	J3
110K	10K	J4
		空
		空



空闲区表

始址	长度	标志
15K	23K	未分配
48K	20K	未分配
98K	12K	未分配
		空
		空

已分配区表

始址	长度	标志
0K	15K	J1
38K	10K	J2
68K	12K	J3
110K	10K	J4
80K	5K	J5
85K	13K	J6

除了分区说明表之外，动态分区法还把内存中的可用分区单独构成可用分区表或可用分区自由链，以描述系统内的内存资源。与此相对应,请求内存资源的作业或进程也构成一个内存资源请求表。图5.9给出了可用表，自由链和请求表的例子。

可用表的每个表目记录一个空闲区，主要参数包括区号、长度和起始地址。采用表格结构，管理过程比较简单，但表的大小难以确定，可用表要占用一部分内存。

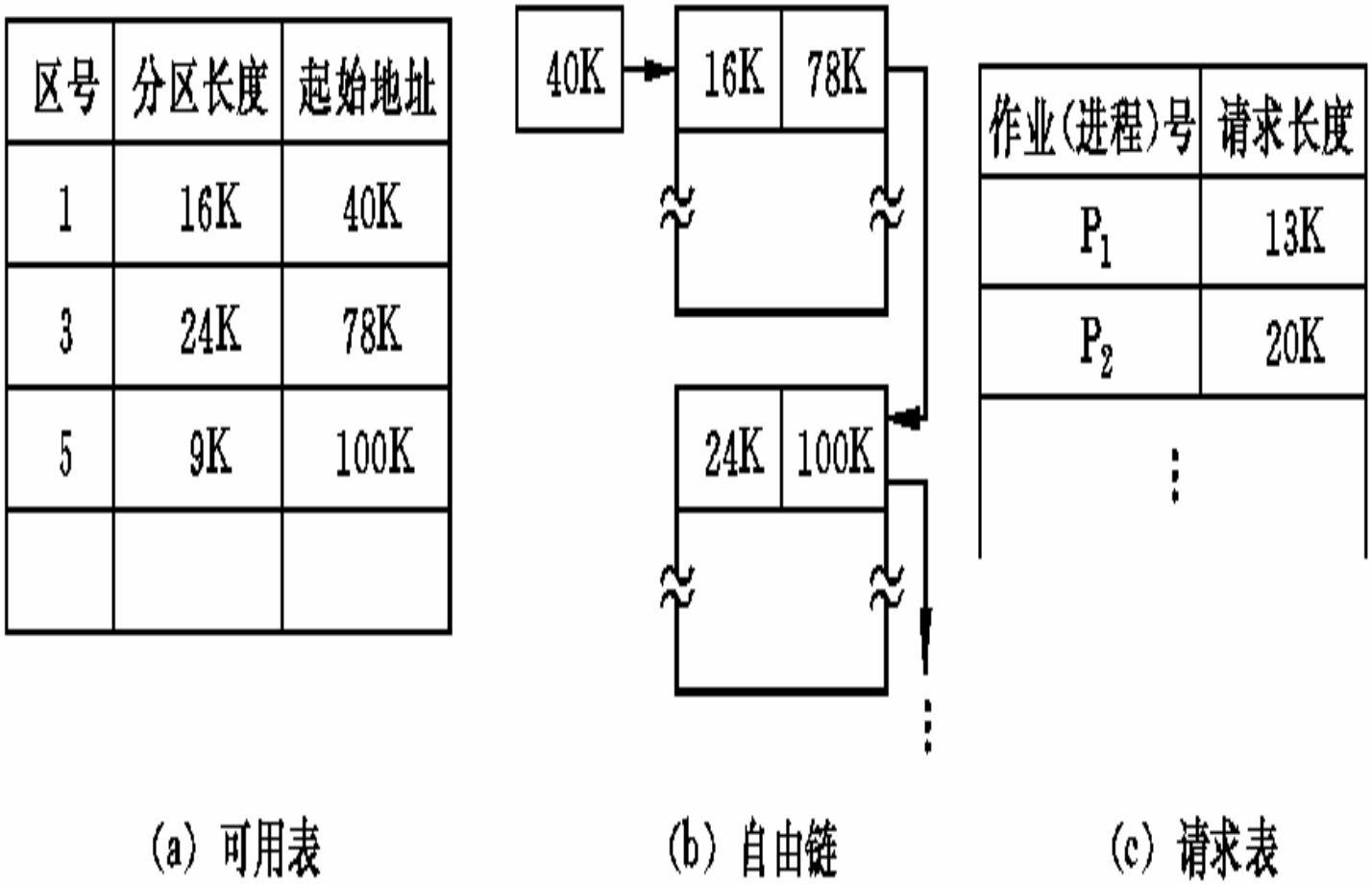
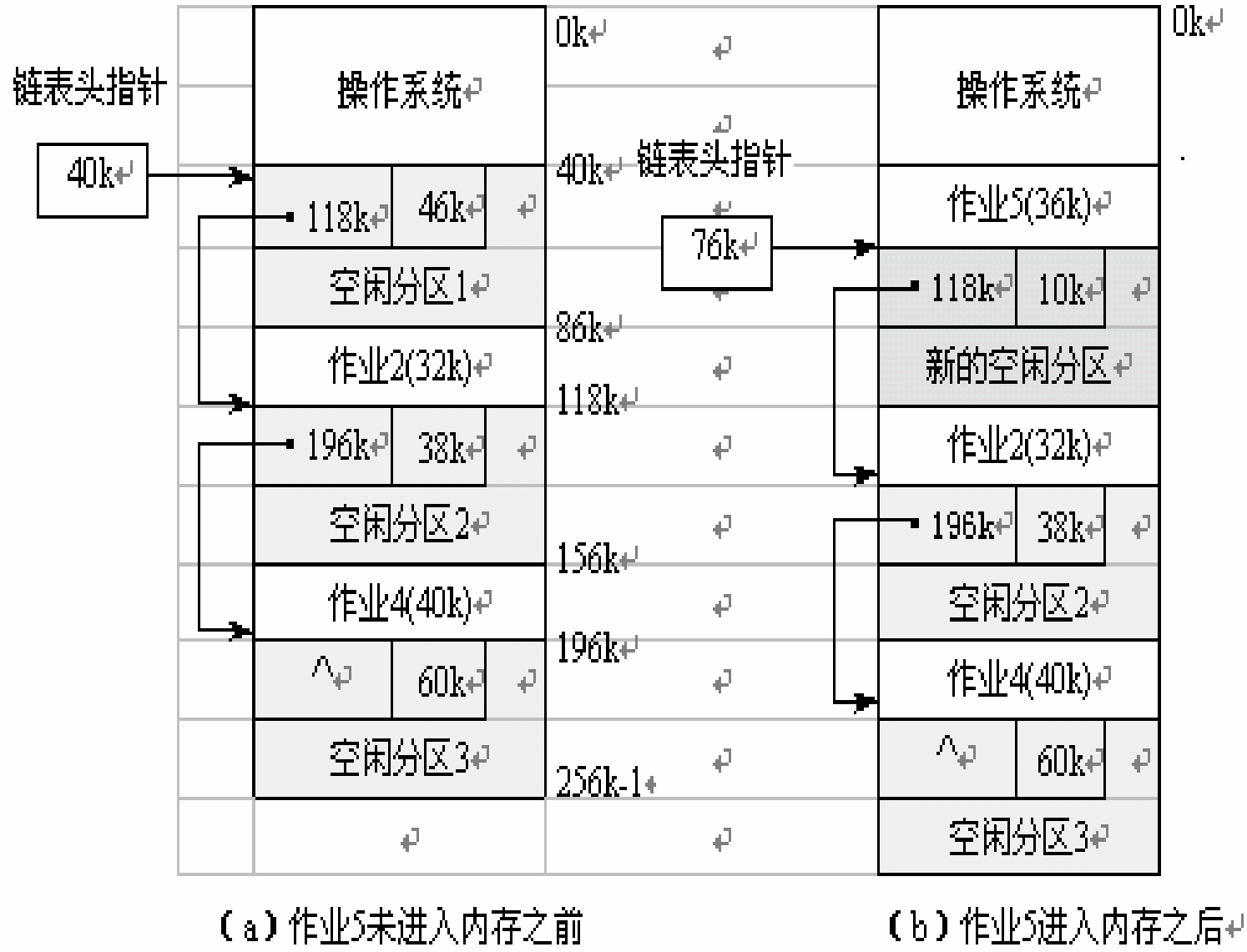


图5.9 可用表、自由链及请求表


链表头指针
40k
操作系统
118k
46k
空闲分区1
作业2(32k)
196k
38k
空闲分区2
作业4(40k)
^
60k
空闲分区3
0k
40k
86k
118k
156k
196k
256k-1
链表头指针
76k
操作系统
作业5(36k)
118k
10k
新的空闲分区
作业2(32k)
196k
38k
空闲分区2
作业4(40k)
^
60k
空闲分区3
0k
(a) 作业5未进入内存之前
(b) 作业5进入内存之后

固定分区时的分配与回收

通过请求表提出内存分配要求和所要求的内存空间大小。

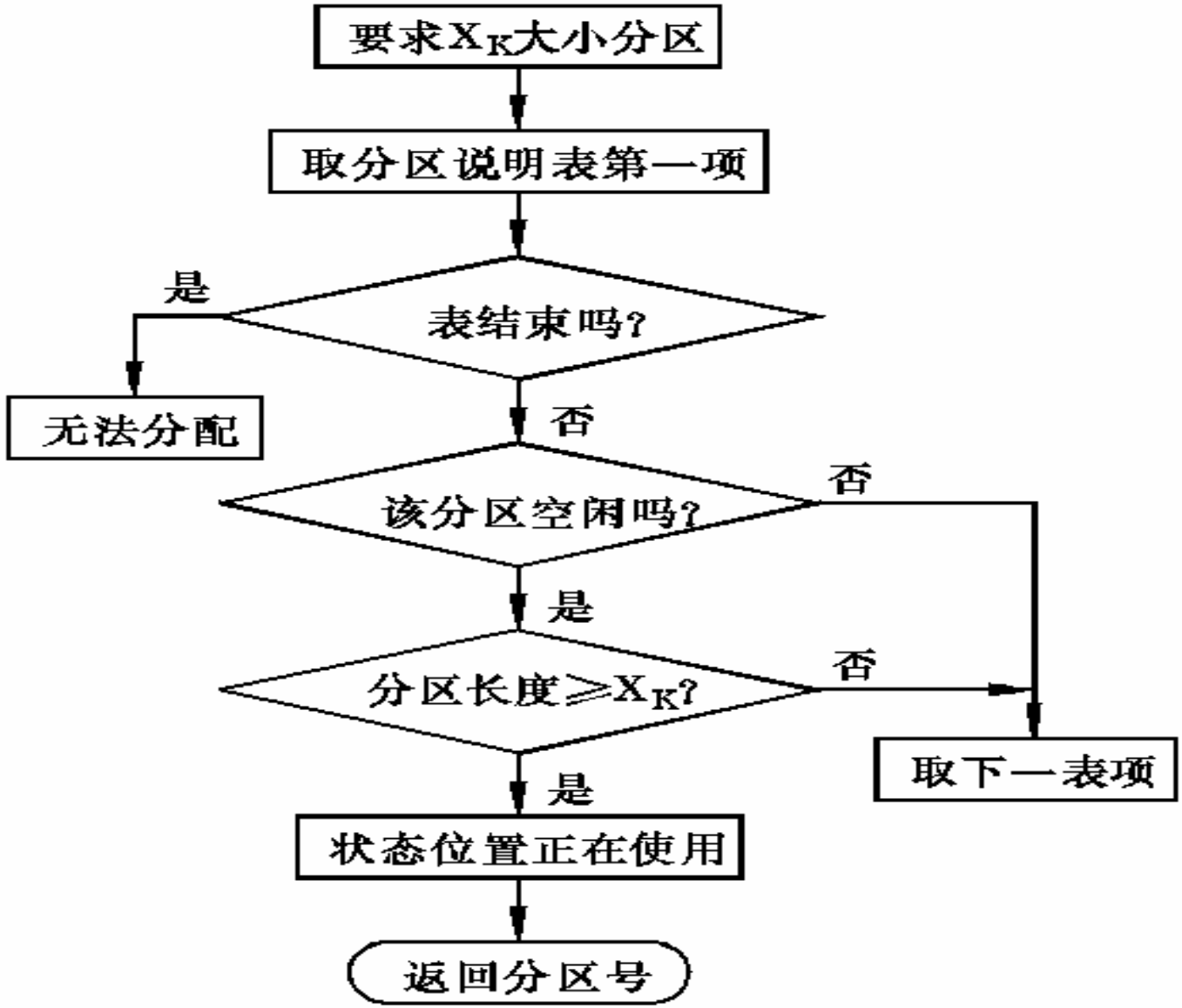
存储管理程序根据请求表查询分区说明表，从中找出一个满足要求的空闲分区，并将其分配给申请者。

□□

- 分区分配算法：寻找某个空闲分区，其大小需大于或等于程序的要求。

  若是大于要求，则将该分区分割成两个分区，其中一个分区为要求的大小并标记为“占用”，而另一个分区为余下部分并标记为“空闲”。分区的先后次序通常是从内存低端到高端。

- 分区释放算法：需要将相邻的空闲分区合并成一个空闲分区。(这时要解决的问题是：合并条件的判断和合并时机的选择)


要求X_K大小分区
取分区说明表第一项
表结束吗?
是
无法分配
否
该分区空闲吗?
否
取下一表项
是
分区长度≥X_K?
否
是
状态位置正在使用
返回分区号

动态分区时的分配与回收

动态分区时的分配与回收主要解决三个问题:□

(1) 对于请求表中的要求内存长度，从可用表或自由链中寻找出合适的空闲区分配程序。

(2) 分配空闲区之后，更新可用表或自由链。

(3) 进程或作业释放内存资源时，和相邻的空闲区进行链接合并，更新可用表或自由链。

- 最先匹配法(first-fit)：按分区的先后次序，从头查找，找到符合要求的第一个分区
  - 该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。
  - 但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。
- 最佳匹配法(best-fit)：找到其大小与要求相差最小的空闲分区
  - 从个别来看，外碎片较小，但从整体来看，会形成较多外碎片。较大的空闲分区可以被保留。

- 最坏匹配法(worst-fit)：找到最大的空闲分区
  - 基本不留下小空闲分区，但较大的空闲分区不被保留。
- 下次匹配法(next-fit)：按分区的先后次序，从上次分配的分区起查找（到最后分区时再回到开头），找到符合要求的第一个分区
  - 该算法的分配和释放的时间性能较好，使空闲分区分布得更均匀，但较大的空闲分区不易保留。

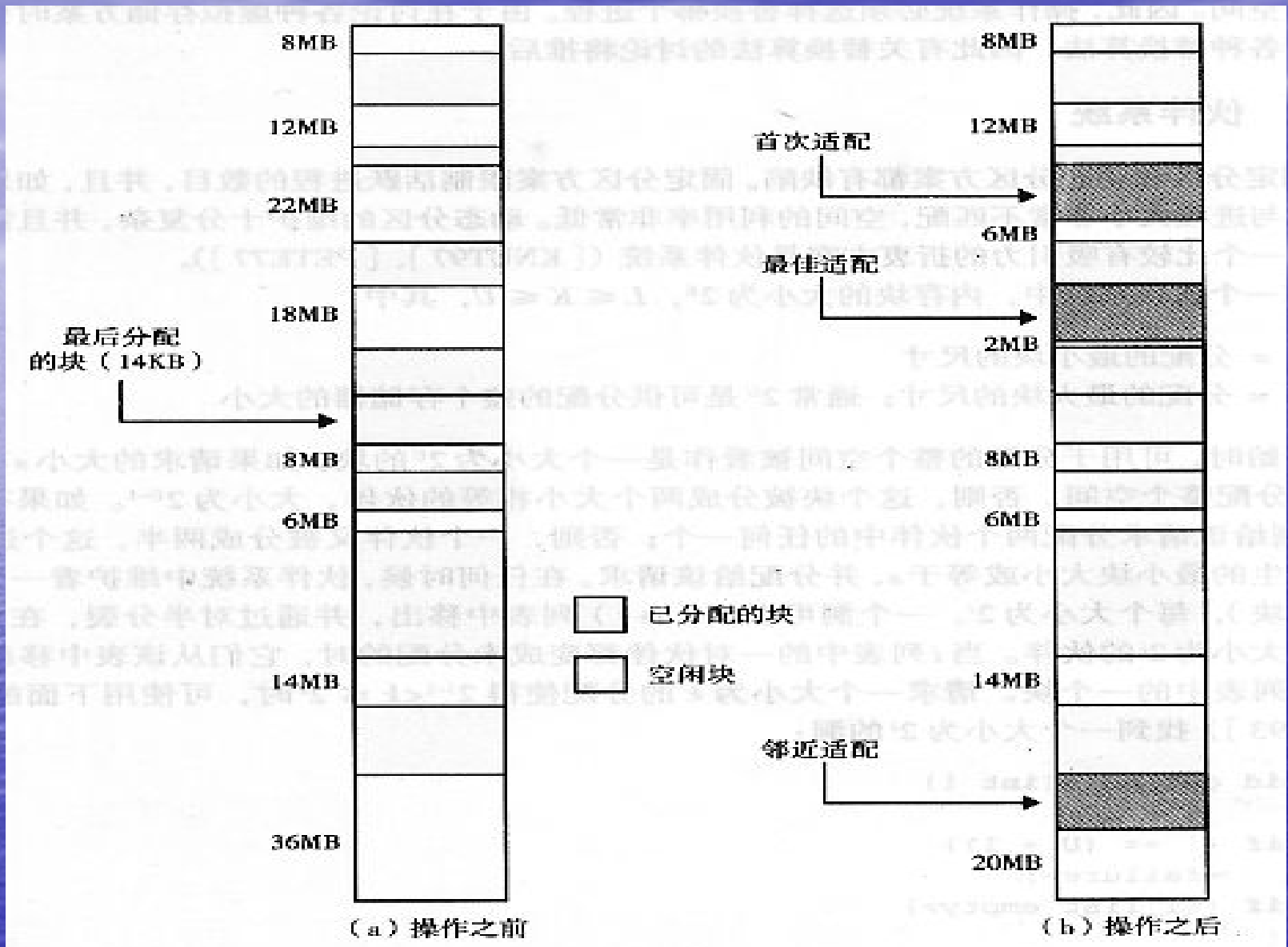


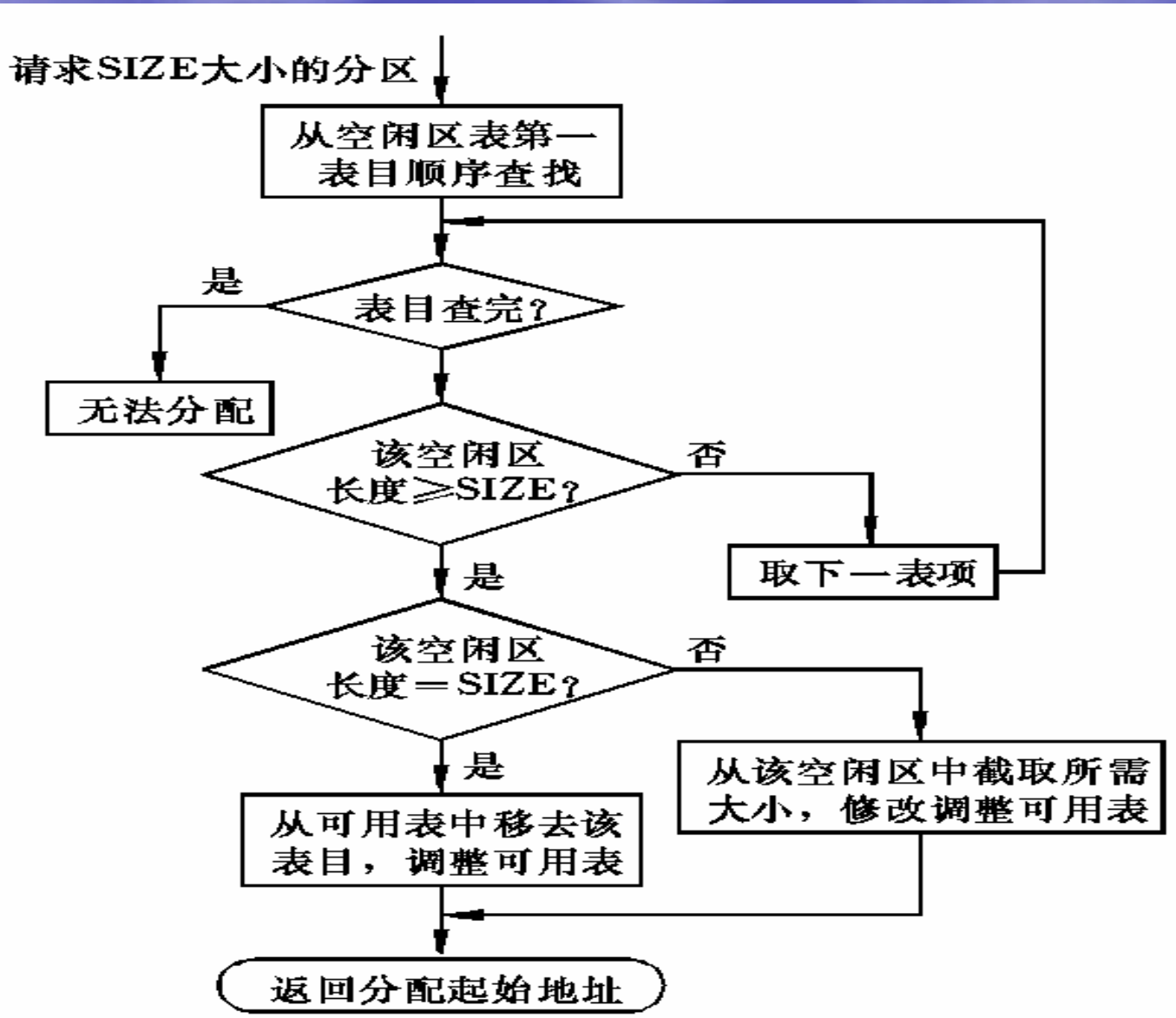
图 7.5　16MB 的块分配操作之前和之后的存储器配置例子

(1) 最先适应法

最先适应法要求可用表或自由链按起始地址递增的次序排列。

该算法的最大特点是一旦找到大于或等于所要求内存长度的分区，则结束探索。

然后，该算法从所找到的分区中划出所要求的内存长度分配给用户，并把余下的部分进行合并(如果有相邻空闲区存在)后留在可用表中，但要修改其相应的表项。 最先适应算法如图5.11所示。


请求SIZE大小的分区
从空闲区表第一表目顺序查找
表目查完？
是
无法分配
该空闲区长度≥SIZE？
否
取下一表项
是
该空闲区长度＝SIZE？
否
是
从可用表中移去该表目，调整可用表
从该空闲区中截取所需大小，修改调整可用表
返回分配起始地址

(2) 最佳适应算法□□□

最佳适应算法要求从小到大的次序组成空闲区可用表或自由链。

当用户作业或进程申请一个空闲区时，存储管理程序从表头开始查找，当找到第一个满足要求的空闲区时，停止查找。如果该空闲区大于请求表中的请求长度，则与最先适应法时相同，将减去请求长度后的剩余空闲区部分留在可用表中。

(3) 最坏适应算法

最坏适应算法要求空闲区按其大小递减的顺序组成空闲区可用表或自由链。

当用户作业或进程申请一个空闲区时，先检查空闲区可用表或自由链的第一个空闲可用区的大小是否大于或等于所要求的内存长度，若可用表或自由链的第一个项长度小于所要求的，则分配失败，否则从空闲区可用表或自由链中分配相应的存储空间给用户，然后修改和调整空闲区可用表或自由链。

动态分区时的回收与拼接

- 在将一个新空闲可用区插入可用表或队列时，该空闲区和上下相邻区的关系是下述4种关系之一：
  a)该空闲区的上下两相邻分区都是空闭区。
  b)该空闲区的上相邻区是空闲区
  c)该空闲区的下相邻区是空闲区
  d)两相 邻区都不是空闲区。

如图5.12所示。

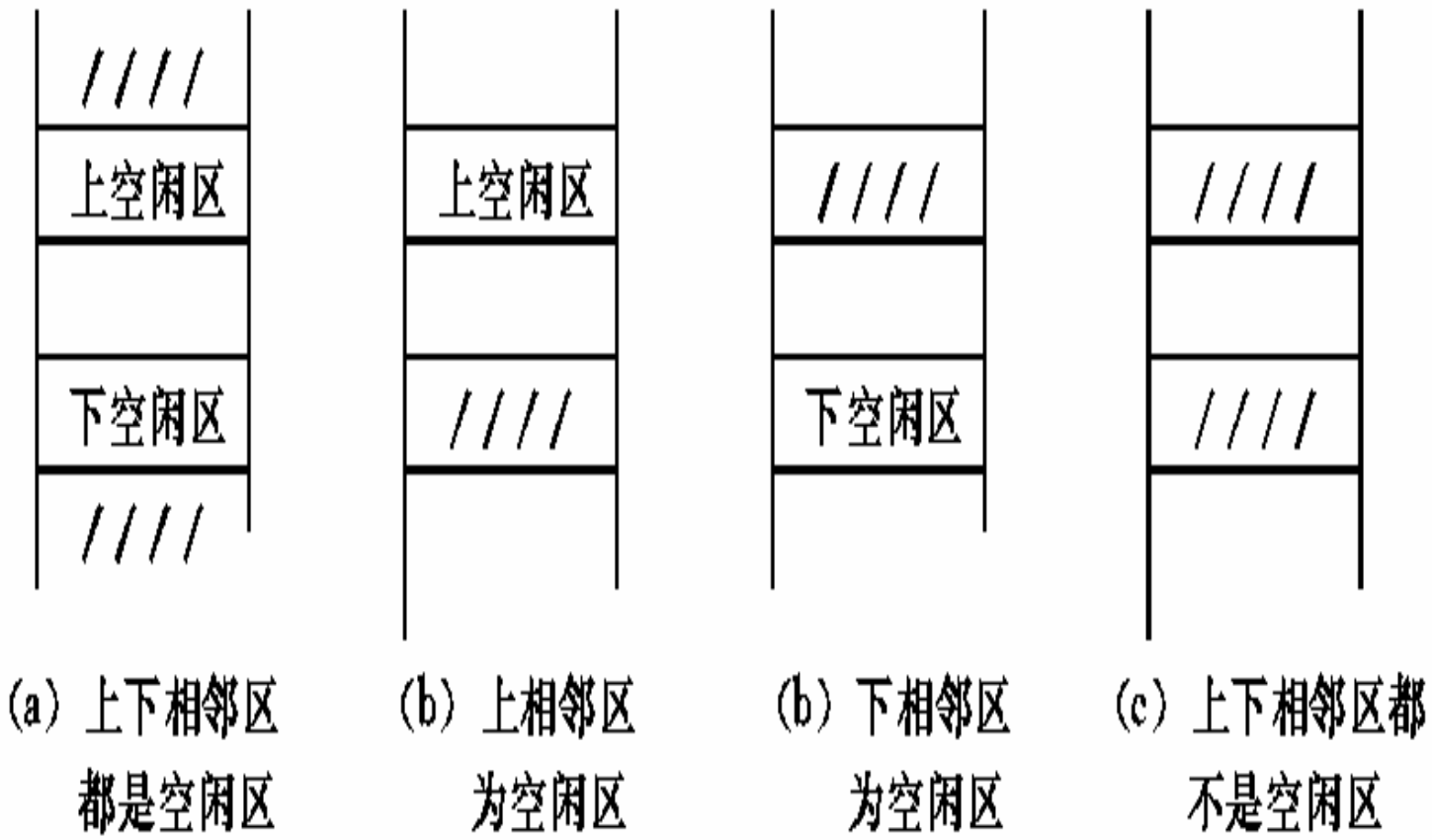

上空闲区
下空闲区
上空闲区
下空闲区
(a) 上下相邻区都是空闲区
(b) 上相邻区为空闲区
(b) 下相邻区为空闲区
(c) 上下相邻区都不是空闲区

图5.12 空闲区的合并

几种分配算法的比较

首先，从搜索速度上看，最先适应算法具有最佳性能。尽管最佳适应算法或最坏适应算法看上去能很快地找到一个最适合的或最大的空闲区。

再者，从回收过程来看，最先适应算法也是最佳的。因为使用最先适应算法回收某一空闲区时，无论被释放区是否与空闲区相邻，都不用改变该区在可用表或自由链中的位置，只需修改其大小或起始地址。

最先适应算法的另一个优点就是尽可能地利用了低地址空间，从而保证高地址有较大的空闲区来放置要求内存较多的进程或作业。

最佳适应法找到的空闲区是最佳的。不过，在某些情况下并不一定提高内存的利用率。

最坏适应算法正是基于不留下碎片空闲区这一出发点的。它选择最大的空闲区来满足用户要求，以期分配后的剩余部分仍能进行再分配。

总之，上述三种算法各有特长，针对不同的请求队列，效率和功能是不一样的。

页式和段式存储管理

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

页 式 管 理

页式管理的基本原理:

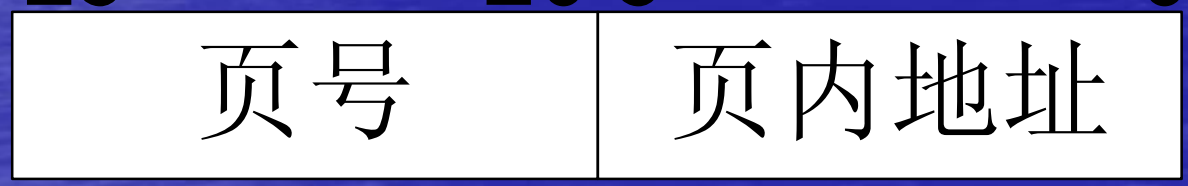
分区式管理存在着严重碎片问题,各作业或进程连续存放,进程的大小仍受分区大小或内存可用空间的限制。

页式管理减少碎片,只在内存存放那些反复执行或即将执行的程序段与数据部分

简单页式(simple paging)

1. 简单页式管理的基本原理

将程序的逻辑地址空间和物理内存划分为固定大小的页或页面(page or page frame)，程序加载时，分配其所需的所有页，这些页不必连续。需要CPU的硬件支持。例如，一个页长为1 K,拥有1 024页的虚拟空间地址结构：



- 页式管理还把内存空间也按页的大小划分为片或页面(page frame)。
- 分页管理时，用户进程在内存空间内除了在每个页面内地址连续之外，每个页面之间不再连续。

优点：1 实现了内存中碎片的减少，因为任一碎片都会小于一个页面。

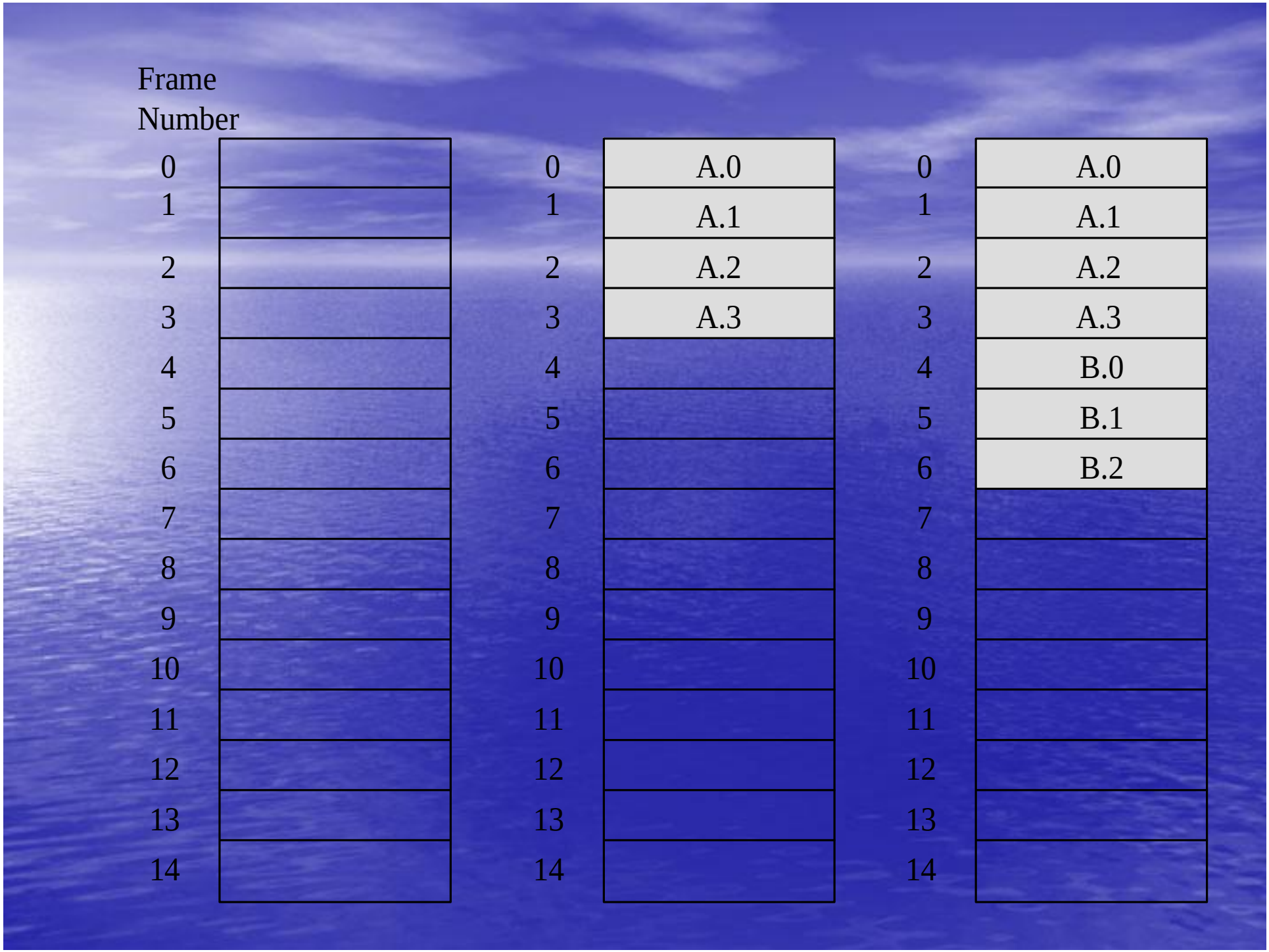
2 实现了由连续存储到非连续存储这个飞跃

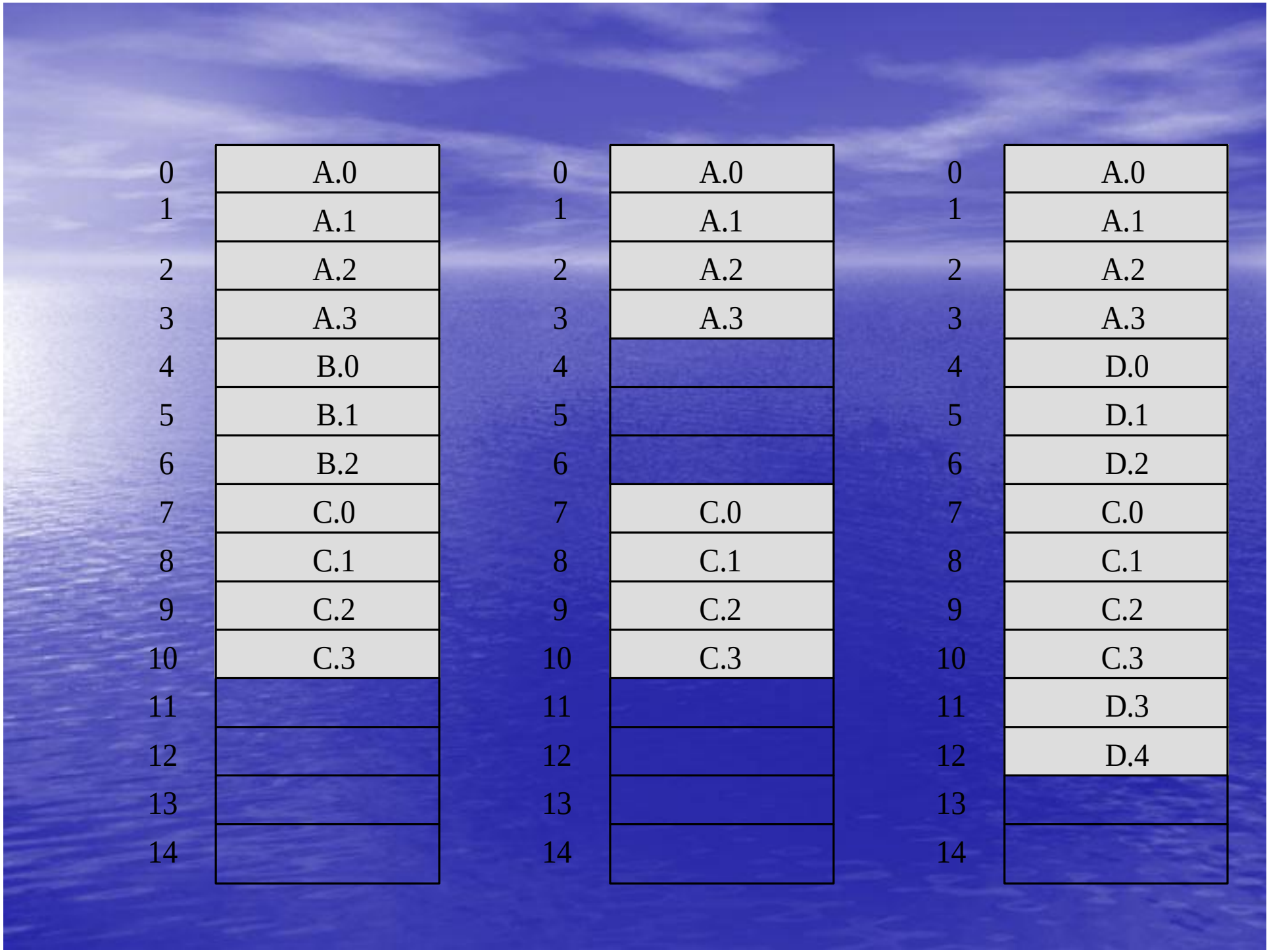
页式管理把页式虚地址与内存页面物理地址建立一一对应页表，并用相应的硬件地址变换机构，来解决离散地址变换问题。页表方式实质上是动态重定位技术的一种延伸。

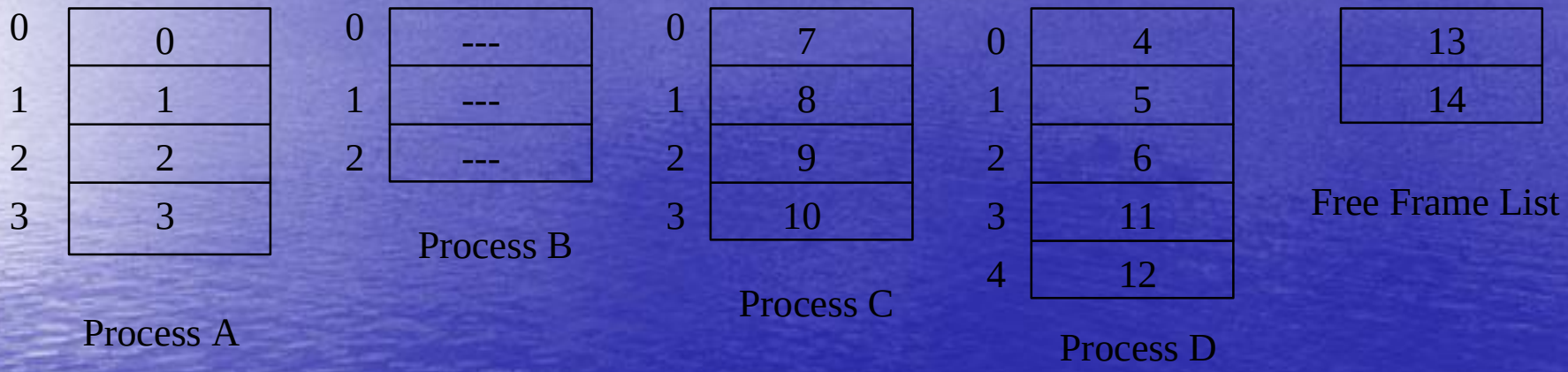
页式管理采用请求调页或预调页技术实现了内外存存储器的统一管理。请求调页或预调页技术是基于工作区的局部性原理的

简单页式管理的数据结构

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


Frame
Number
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
0
A.0
1
A.1
2
A.2
3
A.3
4
5
6
7
8
9
10
11
12
13
14
0
A.0
1
A.1
2
A.2
3
A.3
4
B.0
5
B.1
6
B.2
7
8
9
10
11
12
13
14

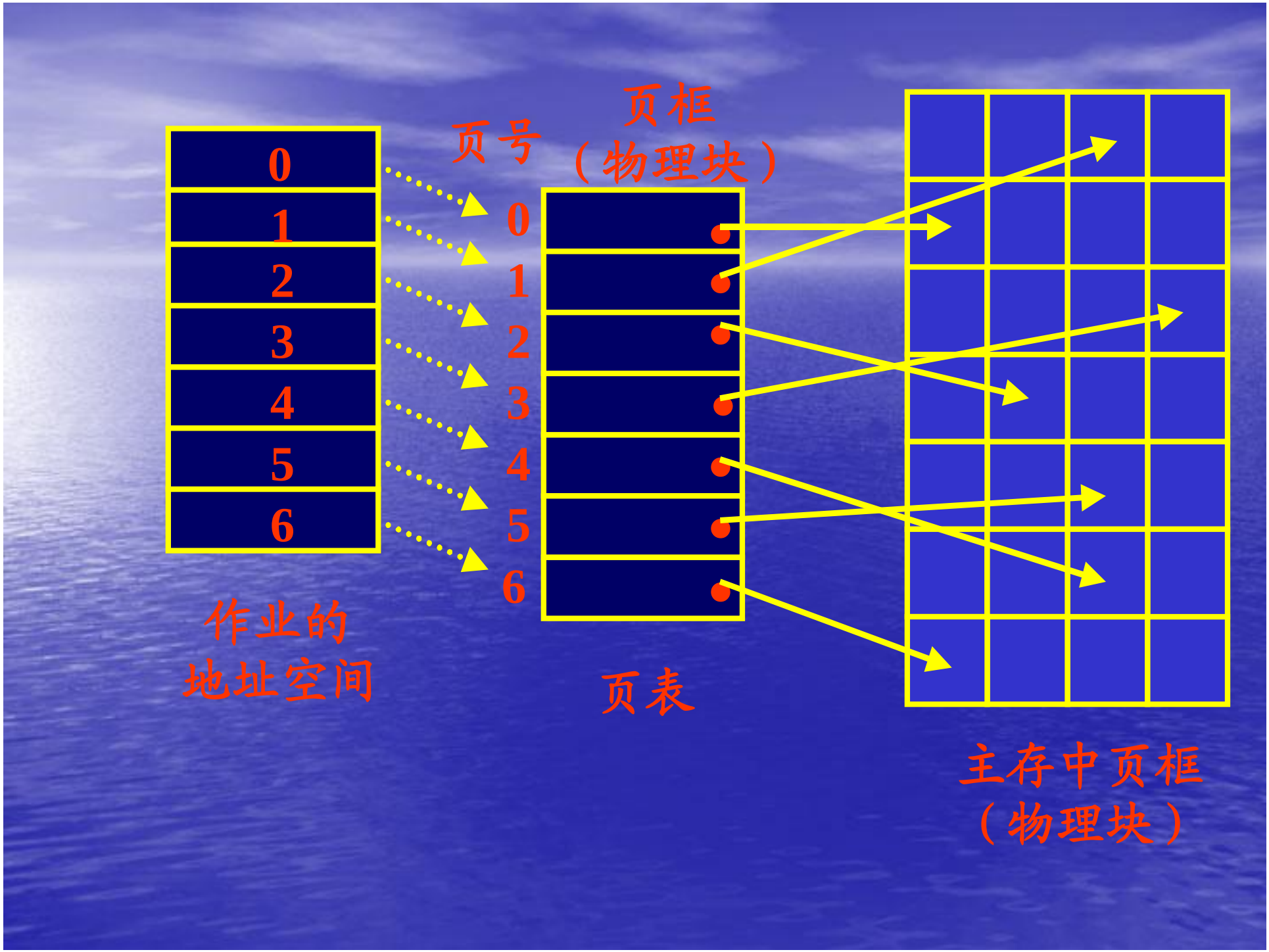

0 A.0
1 A.1
2 A.2
3 A.3
4 B.0
5 B.1
6 B.2
7 C.0
8 C.1
9 C.2
10 C.3
11
12
13
14
0 A.0
1 A.1
2 A.2
3 A.3
4
5
6
7 C.0
8 C.1
9 C.2
10 C.3
11
12
13
14
0 A.0
1 A.1
2 A.2
3 A.3
4 D.0
5 D.1
6 D.2
7 C.0
8 C.1
9 C.2
10 C.3
11 D.3
12 D.4
13
14


0
1
2
3
0
1
2
Process A
0
1
2

Process B
0
1
2
3
7
8
9
10
Process C
0
1
2
3
4
4
5
6
11
12
Process D
13
14
Free Frame List

进程页表

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


0
1
2
3
4
5
6
作业的
地址空间
页号
0
1
2
3
4
5
6
页框
（物理块）
页表
主存中页框
（物理块）

(1) 页表

最简单的页表由页号与页面号组成。如图5.15所示。

页表在内存中占有一块固定的存储区。 页表的大小由进程或作业的长度决定。例如，对于一个每页长1 K，大小为20 K的进程来说，如果一个内存单元存放一个页表项，则只要分配给该页表20个存储单元即可。显然，页式管理时每个进程至少拥有一个页表。□

(2) 请求表

- 请求表用来确定作业或进程的虚拟空间的各页在内存中的实际对应位置。

进程号	请求页面数	页表始址	页表长度	状态
1	20	1 024	20	已分配
2	34	1 044	34	已分配
3	18	1 078	18	已分配
4	21	…	…	未分配
…	…			…

- (3) 存储页面表

存储页面表指出内存各页面是否已被分配出去。存储页面表也有两种构成方法，一种是在内存中划分 一块固定区域，每个单元的每个比特代表一个页面。如果该页面已被分配，则对应比特位置1，否则置0。这种方法称为位示图法。如图5.17所示。

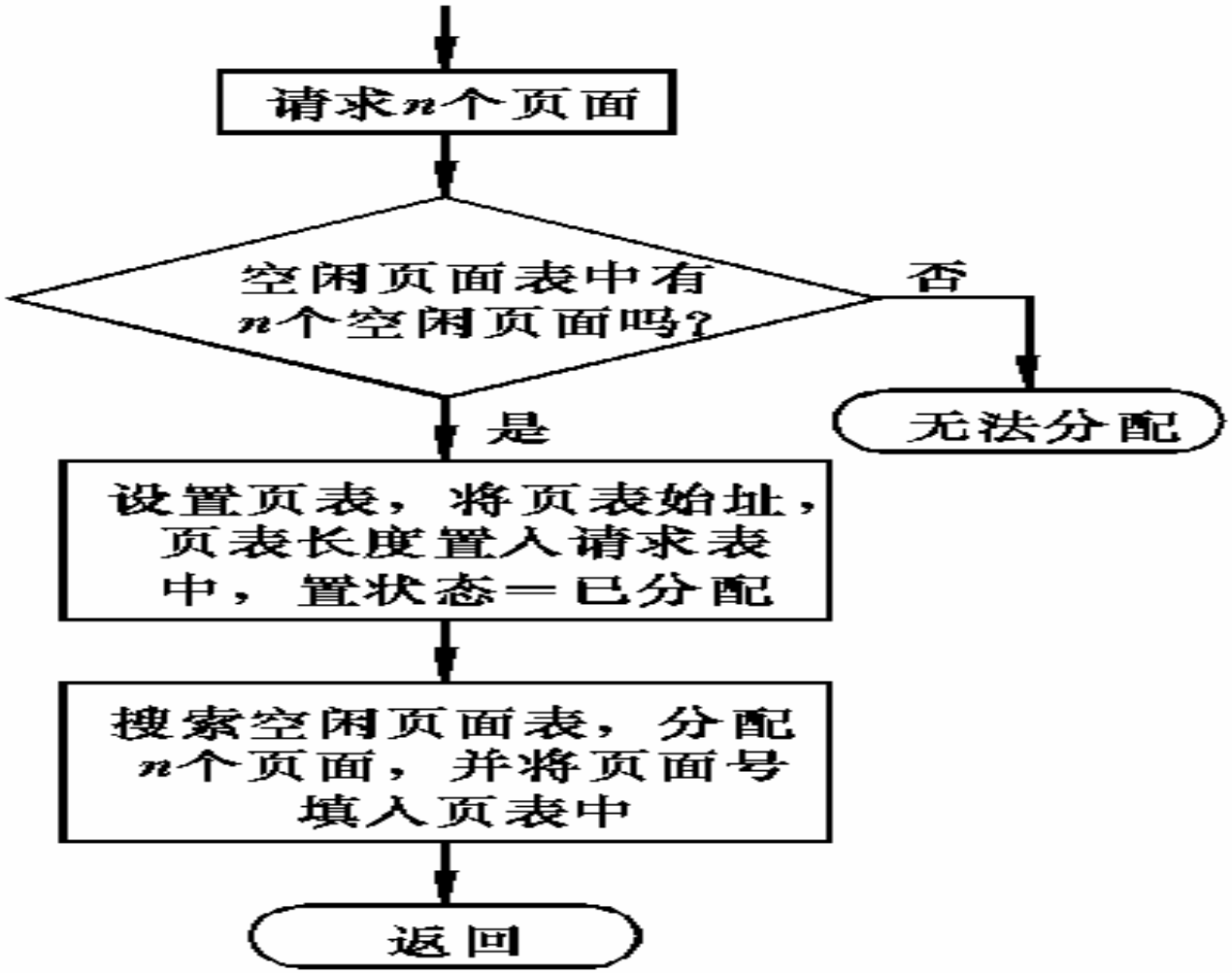
19	18	17	16	15	……	4	3	3	2	0
0	1	1	1	1	……	1	1	0	1	1
0	0	0	1	1	……	0	0	1	1	0
0	0	1	1	1	……	0	0	0	0	0

图5.17 位示图

位示图要占据一部分内存容量，例如，一个划分为1 024个页面的内存，如果内存单元长20比特，则位示图要占据1 024/20=52个内存单元。

存储页面表的另一种构成办法是采用空闲页面链的方法。在空闲页面链中，队首页面的第一个单元和第二个单元分别放入空闲页面总数与指向下一个空闲页面的指针。其他页面的第一个单元中则分别放入指向下一个页面的指针。

分配算法


请求n个页面
空闲页面表中有n个空闲页面吗?
否
无法分配
是
设置页表，将页表始址，页表长度置入请求表中，置状态＝已分配
搜索空闲页面表，分配n个页面，并将页面号填入页表中
返回

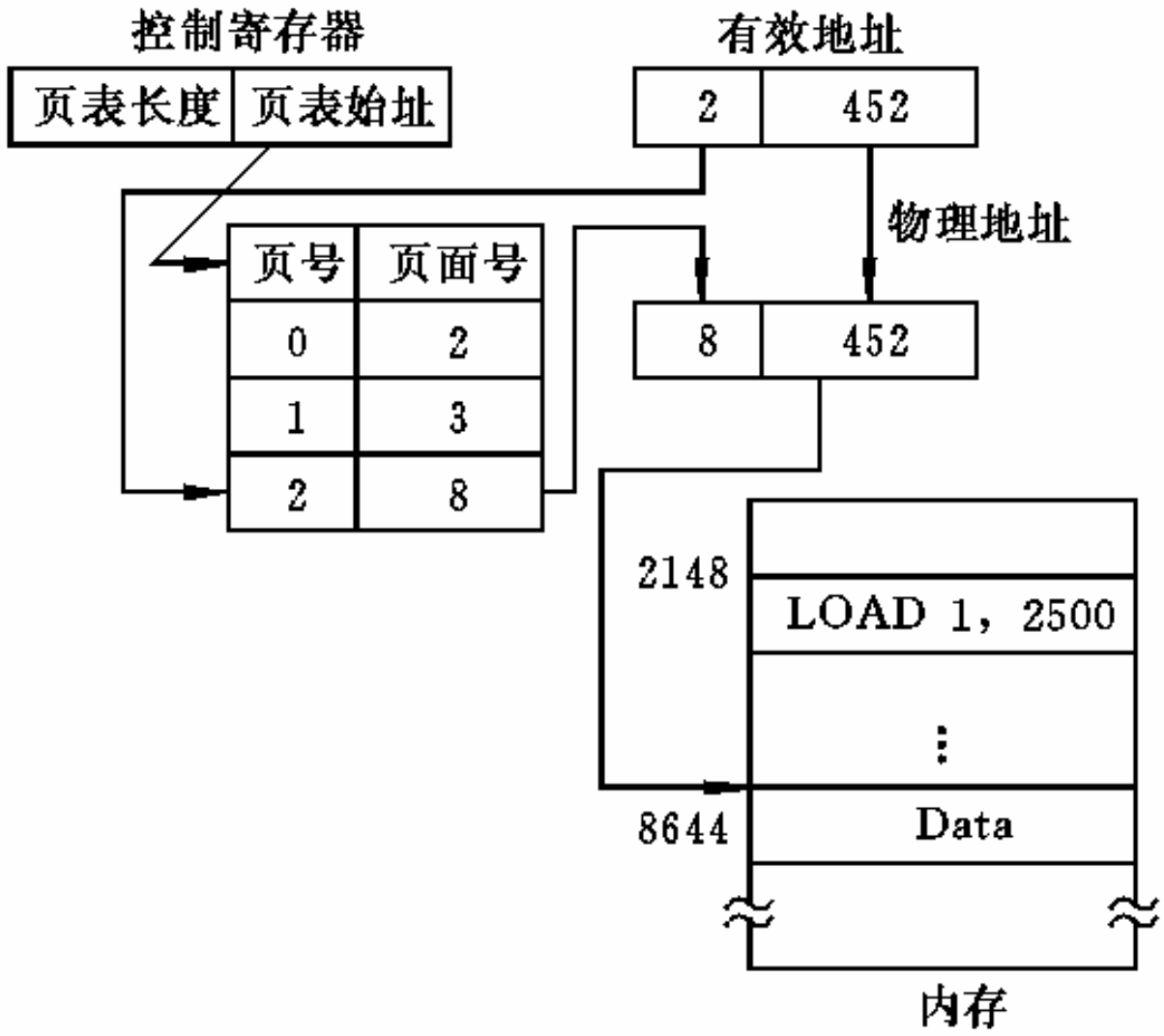
地址变换

- 由页号和页内相对地址变换到内存物理地址的问题

由地址分配方式知道，在一个作业或进程的页表中，连续的页号对应于不连续的页面号。例如，设一个3页长的进程具有页号0，1，2，但其对应的页面号则为2，3，8。如图5.19所示。

每个页面长度为1K，指令LOAD 1，2500的虚地址为100，怎样通过图5.19所示页表来找到该指令所对应的物理地址呢？下面使用该例子说明地址变换过程。

- 进程未执行，页表起始地址和长度存放在进程控制块中，进程执行时，才存入页表寄存器中。
- 当进程访问某个逻辑地址中的信息时，分页地址变换机构自动将逻辑地址分为页号和页内位移，再以页号为索引检索页表。如果页号超越页表长度，发生地址越界中断。
- 最后将块号与逻辑地址中页位移拼接一起，形成访问内存的物理地址。


控制寄存器
页表长度
页表始址
有效地址
2
452
物理地址
页号
页面号
0
2
1
3
2
8
8
452
2148
LOAD 1，2500
8644
Data
内存

另外，由于页表是驻留在内存的某个固定区域中，因此，取一个数据或指令至少要访问内存两次以上。一次访问页表以确定所取数据或指令的物理地址，另一次是根据地址取数据或指令。这比通常执行指令的速度慢了一倍。

提高查找速度一个最直观的办法就是把页表放在寄存器中而不是内存中，但由于寄存器价格太贵，这样做是不可取的。另一种办法是在地址变换机构中加入一个高速联想存储器，构成一张快表。在快表中，存入那些当前执行进程中最常用的页号与所对应的页面号，从而以提高查找速度。

- 引入快表的地址变换过程：

Cup给出逻辑地址后，地址变换机构将逻辑地址分为页号和页内位移

将页号与联想存储器中的所有页号进行并行比较，if匹配该页表项在联想存储器中，取出页号与页内地址拼接形成物理地址 else 再访问内存的页表，取出页号与页内地址拼接形成物理地址。将这次查询到页表项加入联想存储器。若联想存储器满，淘汰出一个表项

- 静态页式管理解决了分区管理时的碎片问题。
- 但是，由于静态页式管理要求进程或作业在执行前全部装入内存，如果可用页面数小于用户要求时，该作业或进程只好等待。
- 而且，作业或进程的大小仍受内存可用页面数的限制。这些问题将在动态(请求)页式管理中解决。

动态页式管理

- 分为请求页式管理和预调入页式管理。

相同：在作业或进程开始执行之前，都不把作业或进程的程序段和数据段一次性地全部装入内存，而只装入被认为是经常反复执行和调用的工作区部分。其他部分则在执行过程中动态装入。

区别:

请求页式管理的调入方式是，当需要执行某条指令而又发现它不在内存时或当执行某条指令需要访问其他的数据或指令时，这些指令和数据不在内存中，从而发生缺页中断，系统将外存中相应的页面调入内存。

- 预调入方式是，系统对那些在外存中的页进行调入顺序计算，估计出这些页中指令和数据的执行和被访问的顺序，并按此顺序将它们顺次调入和调出内存

怎样发现这些不在内存中的虚页以及怎样处理这种情况，是请求页式管理必须解决的两个基本问题。

第一个问题可以用扩充页表的方法解决。即与每个虚页号相对应，除了页面号之外，再增设该页是否在内存的中断位以及该页在外存中的副本起始地址。扩充后的页表如图5.21。

页号	页面号	中断位	外存始址
0			
1			
2			
3			

虚页不在内存时的处理

- 第一，采用何种方式把所缺的页调入内存。
- 第二，采用什么样的策略来淘汰已占据内存的页。
- 如果在内存中的某一页被淘汰，且该页曾因程序的执行而被修改，则显然该页是应该重新写到外存上加以保存的。因此，在页表中还应增加一项以记录该页是否曾被改变。

页号	页面号	中断位	外存始址	改变位
0				
1				
2				
3				

图5.22 加入改变位后的页表

如果置换算法选择不当，有可能产生刚被调出内存的页又要马上被调回内存，调回内存不久又马上被调出内存，如此反复的局面。这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主存和辅存之间的来回调入调出上。这种现象被称为抖动(thrashing)现象。

- 有一个矩阵 int a[100][100],以行为先进行存储。假设有一个虚拟存储系统，物理内存有3页，其中1页用来存放程序，其余2页用于存放数据。假设程序已在内存中占1页，其余2页空闲。

- 程序A

```
for (i=0;i<=99;i++)
    for(j=0;j<=99;j++)
        a[i][j]=0;
```

- 程序B

```
for(j=0;j<=99;j++)
    for (i=0;i<=99;i++)
        a[i][j]=0;
```

- 若每页可存放200个整数，程序A和程序B的执行过程各会发生多少次缺页？若每页只能存放100个整数呢？以上说明了什么问题？

- a[0][0],a[0][1], …. a[0][99]
- a[1][0],a[1][1], …. a[1][99]
- …
- a[99][0],a[99][1], …. a[99][99]
- 100/2=50次缺页中断。

- a[0][0],a[1][0], …. a[99][0]
- a[0][1],a[1][1], …. a[99][1]
- …
- a[0][99],a[1][99], …. a[99][99]
- 10000/2=5000次中断

- 100次
- 10000次
- 缺页中断次数和数据存放方法及程序访问数据有很大关系。

请求页式管理中的置换算法

置换算法应该置换那些被访问概率最低的页，将它们移出内存。比较常用的置换算法有以下几种:

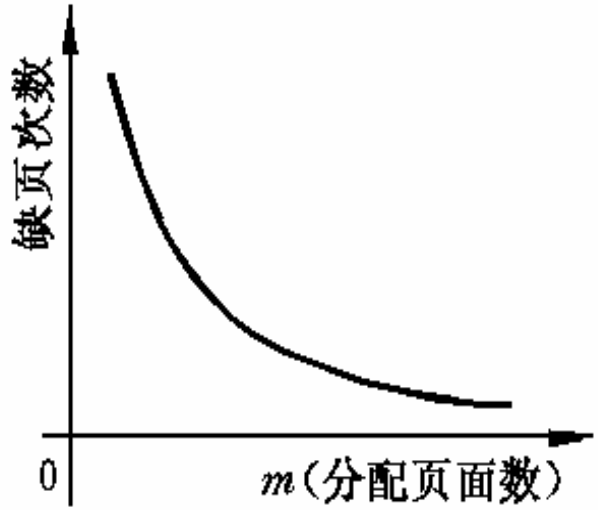
(1) 随机淘汰算法。在系统设计人员认为无法确定哪些页被访问的概率较低时，随机地选择某个用户的页面并将其换出将是一种明智的作法。

(2) 轮转法和先进先出算法。轮转法循回换出内存可用区内一个可以被换出的页，无论该页是刚被换进或已换进内存很长时间

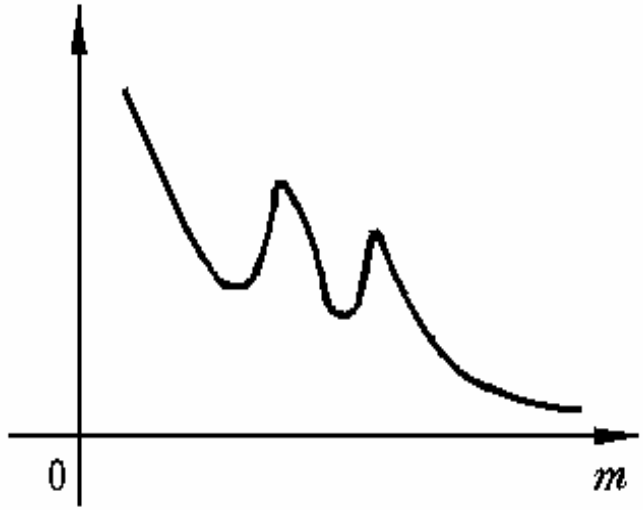
- FIFO算法认为先调入内存的页不再被访问的可能性要比其他页大，因而选择最先调入内存的页换出。
- 由实验和测试发现FIFO算法和RR算法的内存利用率不高。这是因为，这两种算法都是基于CPU按线性顺序访问地址空间的这个假设上。事实上，许多时候，CPU不是按线性顺序访问地址空间的，例如执行循环语句时。因此，那些在内存中停留时间最长的页往往也是经常被访问的页。尽管这些页变“老”了。 但它们被访问的概率仍然很高。

Belady现象

- Belady现象：采用FIFO算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多，缺页率反而提高的异常现象。



（a）正常情况



（b）Belady现象

图5.24 FIFO算法的Belady现象

下面的例子可以用来说明FIFO算法的正常换页情况和Belady现象。例：设进程P共有8页，且已在内存中分配有3个页面，程序访问内存的顺序(访问串)为7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1。这里，这些自然数代表进程P所建的程序和数据的页号。内存中有关进程P所建的程序和数据的各页面变化情况如图5.25所示。

FIFO	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
页0	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0
页1		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1
页2			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2
缺页	X	x	x	x	√	x	x	x	x	x	x	√	√	x	x	√	√

由图5.25可以看出，实际上发生了12次缺页。如果设缺页率为缺页次数与访问串的访问次数之比，则该例中的缺页率为12/17=70.5%。如果分4个页面，同理算得9/17=52.9%

作业

- 设进程分为5页，访问串为1，2，3，4，1，2，5，1，2，3，4，5时，
- 进程P分得3个页面时，缺页9次
- 进程P分得4个页面时，缺页10次

最近最久未使用算法
(LRU, Least Recently Used)

算法的基本思想是: 当需要淘汰某一页时，选择离当前时间最近的一段时间内最久没有使用过的页先淘汰。该算法的主要出发点是，如果某页被访问了，则它可能马上还要被访问。或者反过来说，如果某页很长时间未被访问，则它在最近一段时间也不会被访问。

LRU	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
页0																				
页1																				
页2																				
页3																				
缺页																				

LRU	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
页0	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
页1		1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3
页2			1	2	3	4	2	1	5	6	6	1	2	3	7	6	3	3	1	2
页3				1	**1**	3	4	2	1	5	5	**6**	**1**	2	2	**7**	**6**	**6**	**6**	**1**
缺页	x	x	x	x			x	x				x	x	x			x			

要完全实现LRU算法是十分困难的。因为要找出最近最久未被使用的页面的话，就必须对每一个页面都设置有关的访问记录项，而且每一次访问都必须更新这些记录。这显然要花费巨大的系统开销。因此，在实际系统中往往使用LRU的近似算法。

比较常用的近似算法有:

- 最不经常使用页面淘汰算法LFU(least frequently used)。
- 选择到当前时间为止被访问次数最少的页面被置换；
- 每页设置访问计数器，每当页面被访问时，该页面的访问计数器加1；
- 发生缺页中断时，淘汰计数值最小的页面，并将所有计数清零；

理想型淘汰算法OPT

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

OPT	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
页0																				
页1																				
页2																				
页3																				
缺页																				

OPT	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
页0	1	2	3	4	4	4	5	6	6	6	6	6	7	7	7	7	1	1	1	1
页1		1	2	3	3	3	3	3	3	3	3	3	6	6	6	6	6	6	6	6
页2			1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
页3				1	**1**	1	1	1	1	1	1	**1**	**2**	2	2	**2**	**2**	**2**	**2**	**2**
缺页	x	x	x	x			x	x					x				x			

页式管理的优缺点

综上所述，页式管理具有如下优点:

(1) 由于它不要求作业或进程的程序段和数据在内存中连续存放，从而有效地解决了碎片问题。

(2) 动态页式管理提供了内存和外存统一管理的虚存实现方式，使用户可以利用的存储空间大大增加。这既提高了主存的利用率，又有利于组织多道程序执行。

其主要缺点是:

(1) 要求有相应的硬件支持。例如地址变换机构，缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。这增加了机器成本。

(2) 增加了系统开销，例如缺页中断处理等。

(3) 请求调页的算法如选择不当，有可能产生抖动现象。

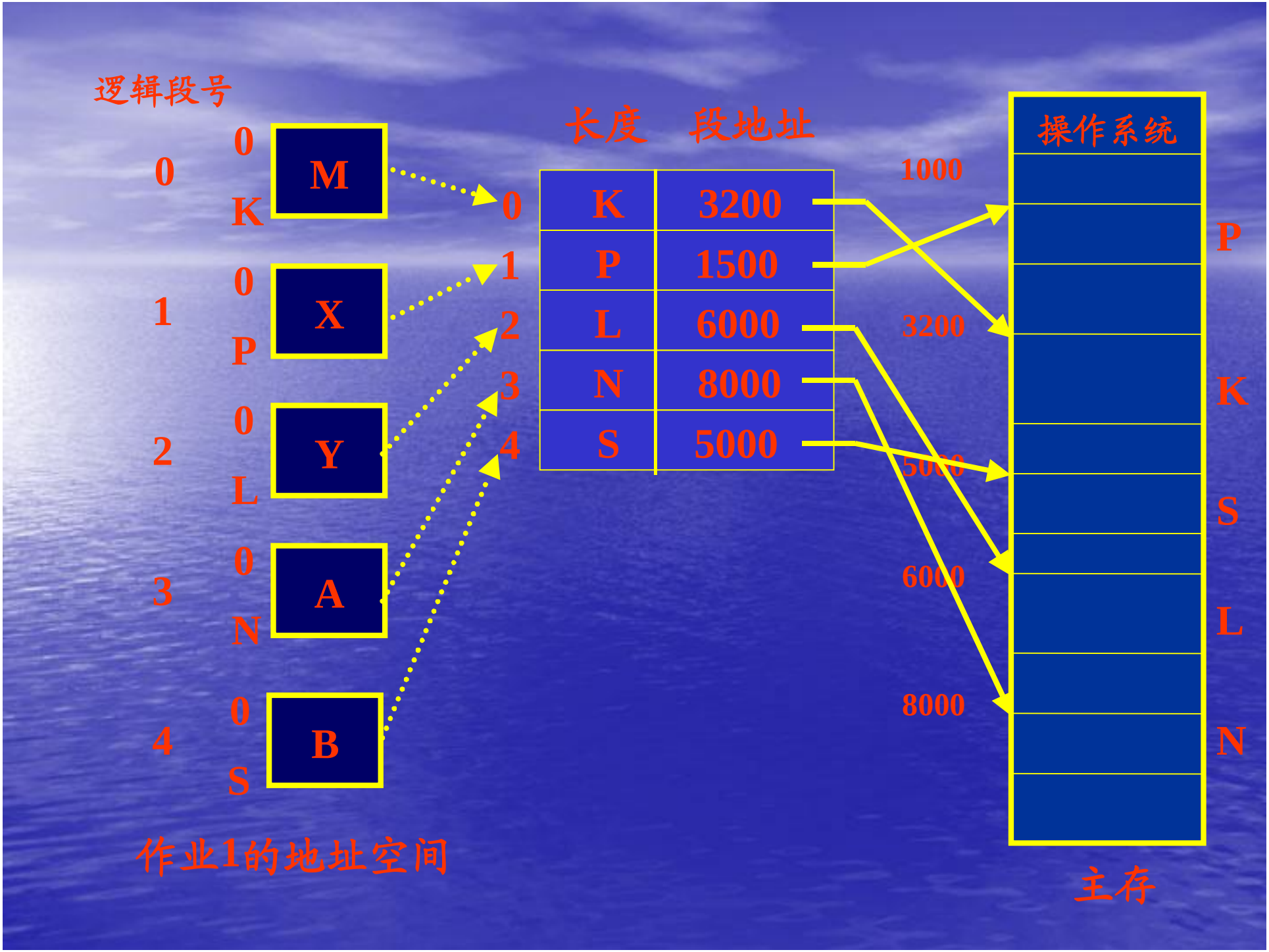
(4) 虽然消除了碎片，但每个作业或进程的最后一页内总有一部分空间得不到利用。如果页面较大，则这一部分的损失仍然较大。

简单段式(simple segmentation)

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

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

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


逻辑段号
0
0
K
M
1
0
P
X
2
0
L
Y
3
0
N
A
4
0
S
B
长度
段地址
0
K
3200
1
P
1500
2
L
6000
3
N
8000
4
S
5000
1000
3200
5000
6000
8000
操作系统
P
K
S
L
N
作业1的地址空间
主存

简单段式管理的数据结构

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

段式管理的地址变换

CPU如何感知到所要访问的段不在内存而启动中断处理程序呢?

还有，段式虚拟地址属于一个二维的虚拟空间。一个二维空间的虚拟地址怎样变换为一个一维的线性物理地址呢? 这些都由段式地址变换机构解决。

(1) 段表(segment mapping table)

和页式管理方案类似，段式管理程序在进行初始内存分配之前，首先根据用户要求的内存大小为进程建立一个段表，以实现动态地址变换和缺段中断处理及存储保护等。如图5.30所示

段号	始址	长度	存取方式	内外	访问位

(2) 动态地址变换

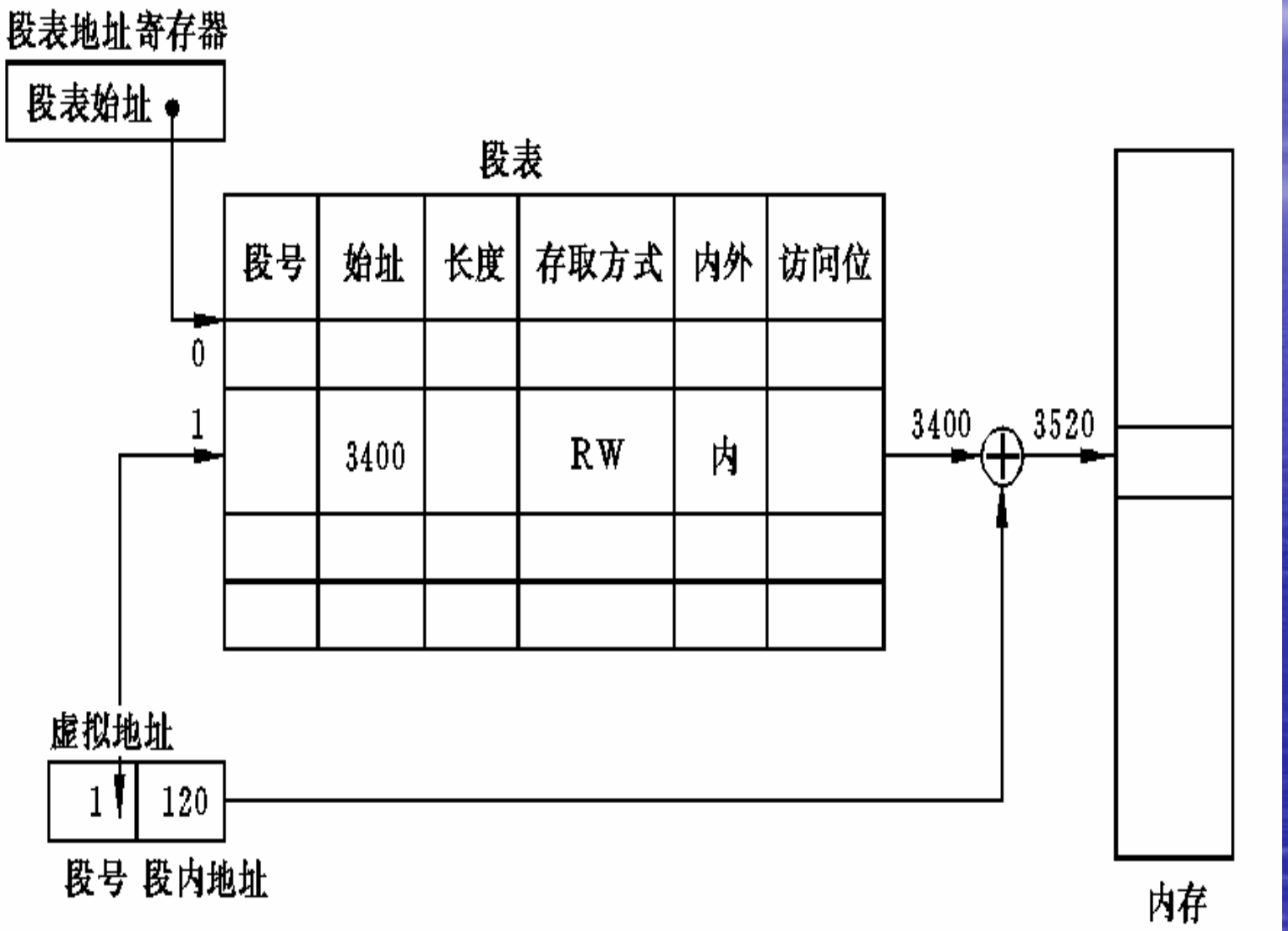
一般在内存中给出一块固定的区域放置段表。当某进程开始执行时，管理程序首先把该进程的段表始址放入段表地址寄存器。

通过访问段表寄存器，管理程序得到该进程的段表始址从而可开始访问段表。然后，由虚地址中的段号s为索引，查段表。

若该段在内存，则判断其存取控制方式是否有错。如果存取控制方式正确，则从段表相应表目中查出该段在内存的起始地址，并将其和段内相对地址w相加，从而得到实际内存地址。

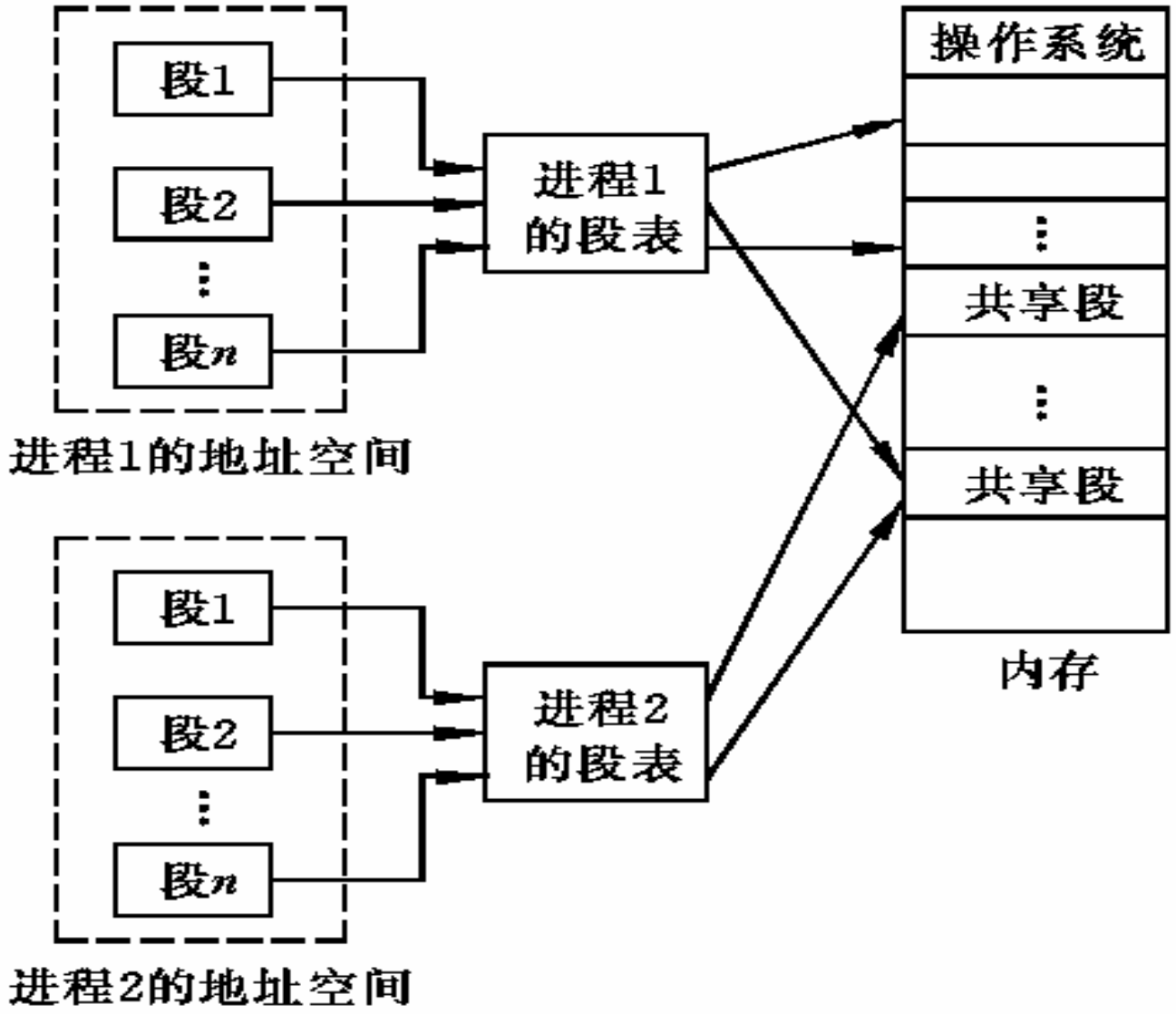
如果该段不在内存，则产生缺段中断将CPU控制权交给内存分配程序。内存分配程序首先检查空闲区链，以找到足够长度的空闲区来装入所需要的段。如果内存中的可用空闲区总数小于所要求的段长时，则检查段表中访问位，以淘汰那些访问概率低的段并将需要段调入。段式地址变换过程如图5.31所示。

与页式管理时相同，段式管理时的地址变换过程也必须经过二次以上的内存访问。 即首先访问段表以计算得到待访问指令或数据的物理地址，然后才是对物理地址进行取数据或存数据操作。为了提高访问速度，页式地址变换时使用的高速联想寄存器的方法也可以用在段式地址变换中。


段表地址寄存器
段表始址
段表
段号
始址
长度
存取方式
内外
访问位
0
1
3400
RW
内
3400
3520
虚拟地址
1
120
段号 段内地址
内存

(1) 段的共享

- 常常有许多子程序和应用程序是被多个用户所使用的
- 如果每个用户进程或作业都在内存保留它们共享程序和数据的副本，那就会极大地浪费内存空间。最好的办法是内存中只保留一个副本，供多个用户使用，称为共享。


段1
段2
段n
进程1的地址空间
进程1
的段表
操作系统
共享段
共享段
内存
段1
段2
段n
进程2的地址空间
进程2
的段表

只要用户使用相同的段名，就可在新的段表中填入已存在于内存之中的段的起始地址，并置以适当的读写控制权，就可做到共享一个逻辑上完整的内存段信息。

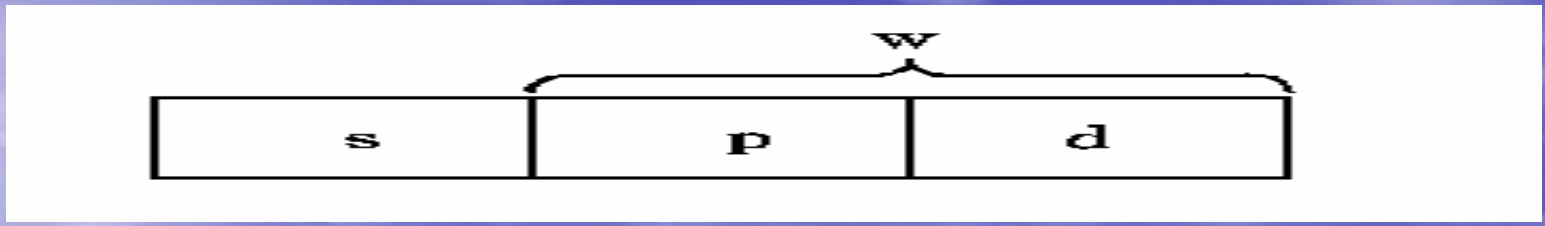
(2) 段的保护

- 一种是地址越界保护法，另一种是存取方式控制保护法。
- 而地址越界保护则是利用段表中的段长项与虚拟地址中的段内相对地址比较进行的。若段内相对地址大于段长，系统就会产生保护中断。

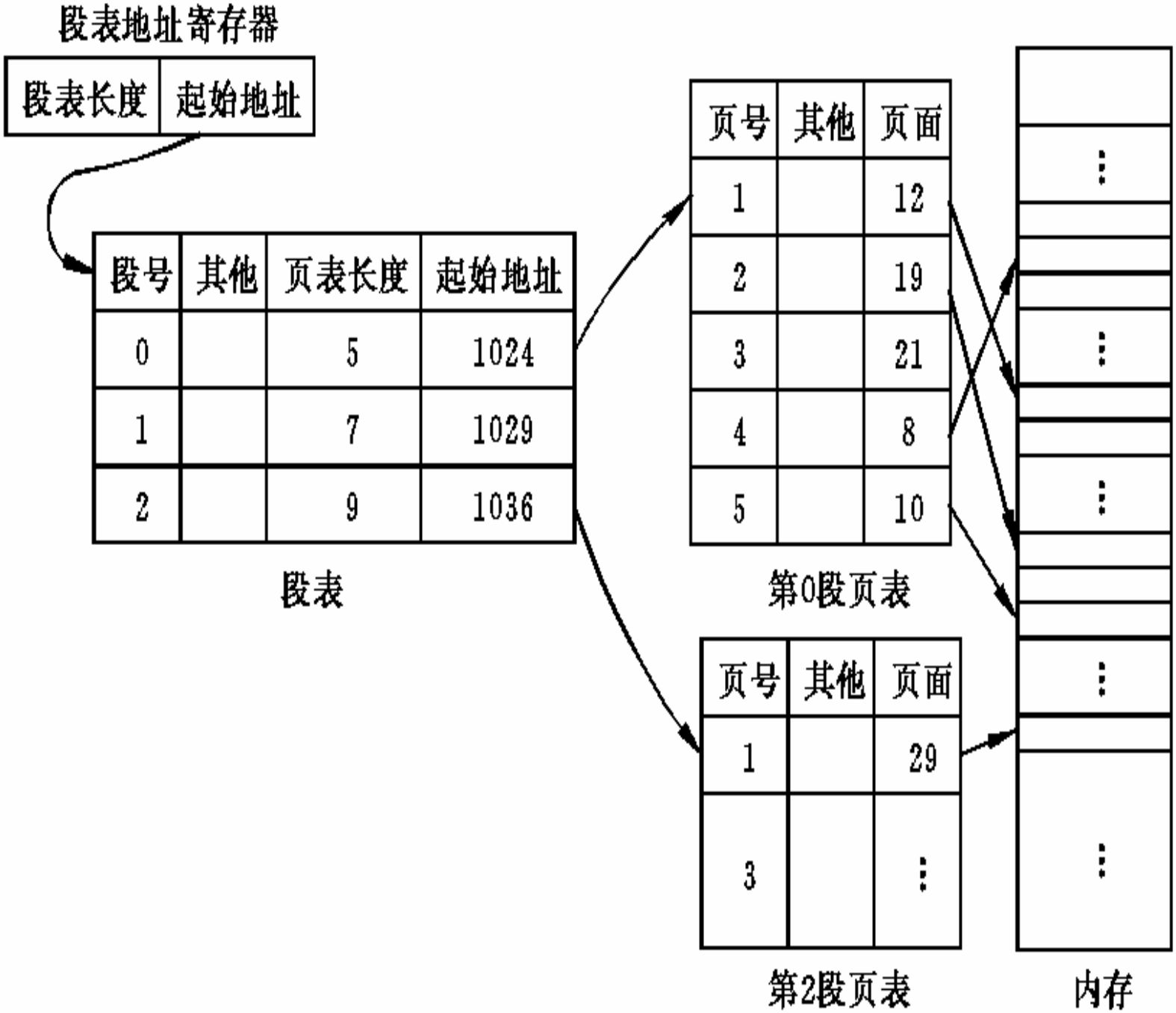
段页式管理的实现原理

1. 虚地址的构成

- 段页式管理时，一个进程仍然拥有一个自己的二维地址空间
- 一个进程中所包含的具有独立逻辑功能的程序或数据仍被划分为段，并有各自的段号s，对于段s中的程序或数据，则按照一定的大小将其划分为不同的页。和页式系统一样，最后不足一页的部分仍占一页。
- 段页式管理时的进程的虚拟地址空间中的虚拟地址由三部分组成: 即段号s，页号p和页内相对地址d。



- 程序员可见的仍是段号s和段内相对地址w。p和d 是由地址变换机构把w的高几位解释成页号p，以及把剩下的低位解释为页内地址d而得到的。
- 由于虚拟空间的最小单位是页而不是段，从而内存可用区也就被划分成为若干个大小相等的页面，且每段所拥有的程序和数据在内存中可以分开存放。分段的大小也不再受内存可用区的限制。


段表地址寄存器
段表长度
起始地址
段号
其他
页表长度
起始地址
0
5
1024
1
7
1029
2
9
1036
段表
页号
其他
页面
1
12
2
19
3
21
4
8
5
10
第0段页表
页号
其他
页面
1
29
3
第2段页表
内存

工作集

- 工作集理论是在1968年由Denning提出并推广的。Denning认为程序在运行时对页面的访问是不均匀的：即往往在某段时间内的访问仅局限于较少的页面；而在另一段时间内，则又可能仅局限于对另一些较少的页面进行访问。如果能够预知程序在某段时间间隔内要访问哪些页面，并能提前将它们调人内存，将会大大降低缺页率，减少置换工作，提高CPU的利用率。

- 所谓工作集是指在某段时间间隔A里，进程实际要访问的页面集合。Denning认为，虽然程序只需少量的几页已在内存就可运行，但为使程序能有效地运行，较：p地产生缺页，就必须使程序的工作集全部在内存中

磁盘存储管理

- ·为文件分配必要的存储空间；
- ·提高磁盘存储空间的利用率；
- ·提高对磁盘的I／O速度，以改善文件系统的性能；
- ·采取必要的冗余措施，来确保文件系统的可靠性。

磁盘调度算法

- (1)先来先服务．(First-Come，First-Served，FCFS)
- 这是一种简单的磁盘调度算法。它根据进程请求访问磁盘的先后次序进行调度。此算法的优点是公平、简单，且每个进程的请求都能依次得到处理，不会出现某一进程的请求长期得不到满足的情况。但此算法由于未对寻道进行优化，致使平均寻道时间可能较长

- (2)最短寻道时间优先(ShortestSeekTimeFirst，SSTF)
- 该算法选择这样的进程，其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短，但这种调度算法却不能保证平均寻道时间最短。

- 扫描(SCAN)算法
- SSTF算法虽然获得较好的寻道性能，但它可能导致某些进程发生“饥饿”(starvation)。SCAN算法不仅考虑到欲访问的磁道与当前磁道的距离，更优先考虑的是磁头的当前移动方向。
- 例如，当磁头正在自里向外移动时，SCAN算法所选择的下一个访问对象应是其欲访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直到最外的磁道需要访问才将磁臂换向，自外向里移动。由于这种算法中磁头移动的规律颇似电梯的运行，露故又称为电梯调度算法。

循环扫描(CSCAN)算法

- 处理该进程的请求，致使该进程的请求被严重地推迟。为了减少这种延迟，CSCAN算法规定磁头单向移动。例如，只自里向外移动，当磁头移到最外的被访问磁道时，磁头立即返回到最里的欲访磁道，即将最小磁道号紧接着最大磁道号构成循环，进行扫描。

磁盘高速缓存的形式

- 这里所说的磁盘高速缓存，并非通常意义下在内存和CPU之间所增设的一个小容量高速存储器，而是指利用内存中的存储空间，来暂存从磁盘中读出的一系列盘块中的信息。因此，这里的高速缓存是一组在逻辑上属于磁盘，而物理上是驻留在内存中的盘块

置换算法

- 如同请求调页(段)一样，在将磁盘中的盘块数据读人高速缓存时，同样会出现因高速缓存中已装满盘块数据，而需要将高速缓存中的数据先换出的问题。相应地，也存在着采用哪种置换算法的问题。较常用的置换算法仍然是最近最久未使用(LRU)算法、最近未使用(NRU)算法及最不常用(LFU)算法等。

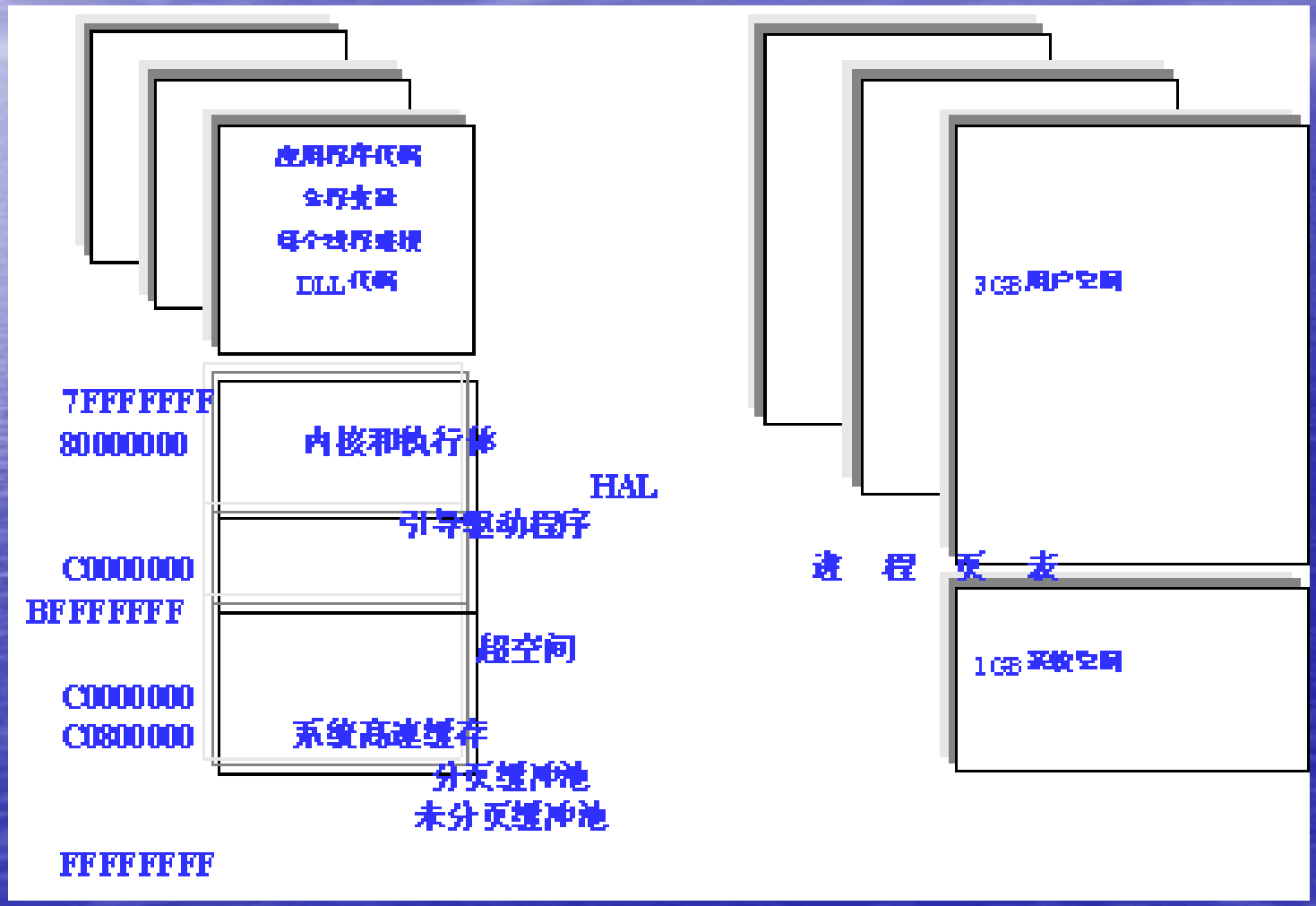
Windows 2000 / XP内存管理

内存管理器是Windows 2000 / XP执行体的一部分，位于Ntoskrnl．exe文件中

- ·一组执行体系统服务程序，用于虚拟内存的分配、回收和管理。大多数这些服务都是以Win32API或核心态的设备驱动程序接口形式出现。
- ·一个转换无效和访问错误陷阱处理程序，用于解决硬件检测到的内存管理异常，并代表进程将虚拟页面装入内存。
- ·运行在六个不同的核心态系统线程上下文中的几个关键组件

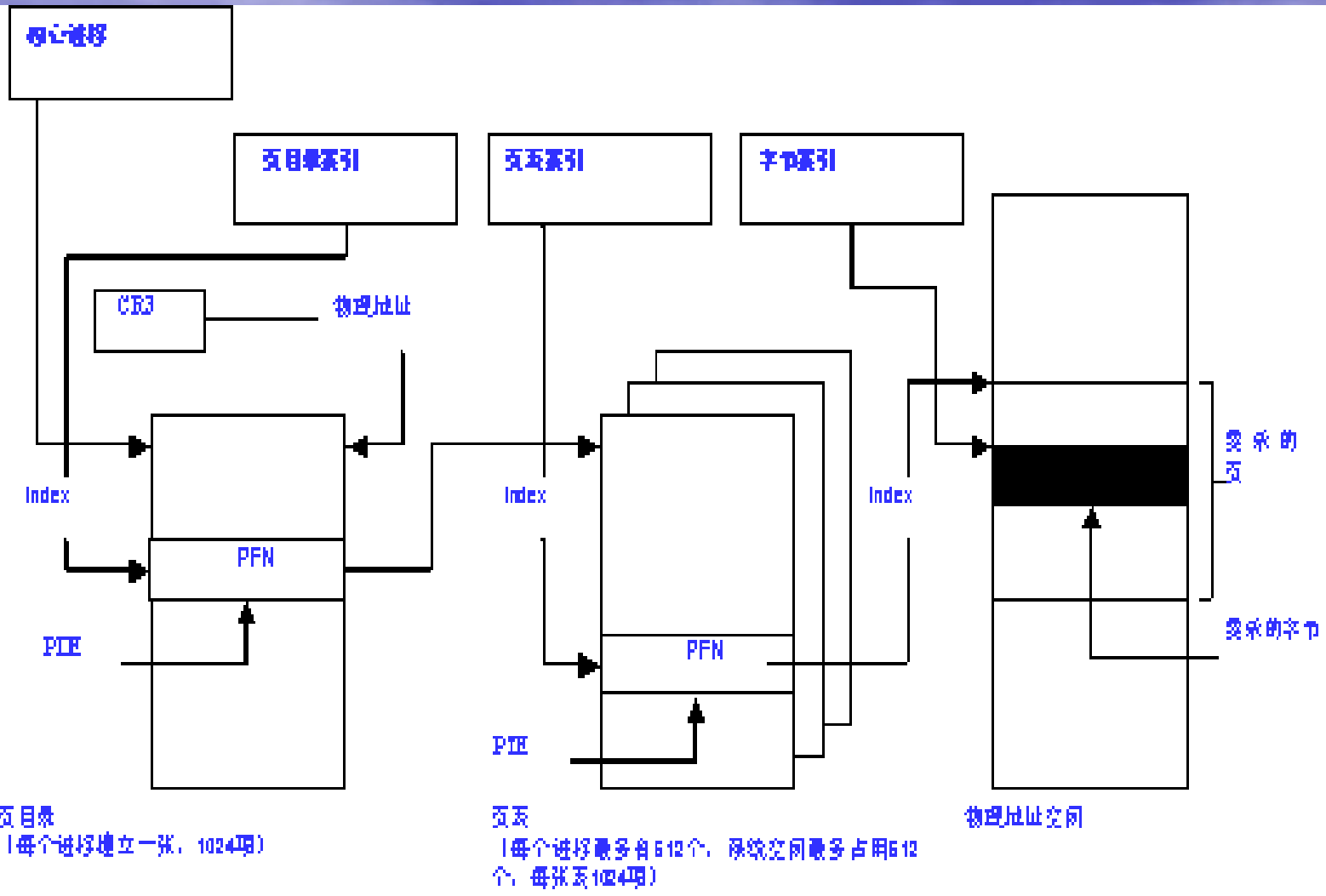
地址空间的布局

- 32位Windows 2000／XP上每个用户进程可以占有2GB的私有地址空间(address space)；操作系统占有剩下的2GB地址空间。Windows 2000／XP高级服务器和Windows 2000／XP数据中心服务器支持一个引导选项，允许用户拥有3GB的地址空间。


应用程序代码
全局变量
每个线程堆栈
DLL代码
7FFFFFFF
80000000
内核和执行体
HAL
引导驱动程序
C0000000
BFFFFFFF
超空间
C0000000
C0800000
系统高速缓存
分页缓冲池
未分页缓冲池
FFFFFFFF
3GB用户空间
进 程 页 表
1GB系统空间

地址转换机制

- Windows 2000 / XP在x86体系结构上利用二级页表结构来实现虚拟地址向物理地址的变换。
- (运行物理地址扩展(PAE)内核的系统是利用三级页表——下面的讨论假定系统为非PAE系统。)
- 一个32位虚拟地址被解释为三个独立的分量——页目录索引、页表索引和字节索引——它们用于找出描述页面映射结构的索引。
- 。比如，在x86系统中，因为一页包含4096字节，于是字节索引被确定为12位宽(212=4096)。


32位地址
页目录索引
页表索引
字节索引
CR3
物理地址
Index
PFN
PTE
Index
PFN
PTE
Index
要求的页
要求的字节
页目录
（每个进程建立一张，1024项）
页表
（每个进程最多有512个，系统空间最多占用512个，每张表1024项）
物理地址空间

以页为单位的虚拟内存分配方式

- 在进程的地址空间中的页面或是空闲的(free)，或被保留(reserved)，或被提交(committed)。应用程序可以首先保留地址空间，然后向此地址空间提交物理页面。它们也可以通过一个函数调用同时实现保留和提交。这些功能是通过Win32~irtualAlloc和VirtualAllocEx函数实现的。
- 保留地址空间是为线程将来使用所保留的一块虚拟地址。试图访问已保留内存会造成访问冲突，因为这时内存页面还没有映射到一个可以满足这次访问的存储器上。在已保留的区域中，提交页面必须指出将物理存储器提交到何处以及提交多少。提交页面在访问时会转变为物理内存中的有效页面。

- 分两步保留和提交内存可以直到需要时才提交页面，这样减少了内存的使用。保留内存是Windows 2000 / XP中既快速又便宜的操作，因为它不消耗任何物理页面(一种珍贵的系统资源)或进程页文件配额(进程可以消耗的提交页面数量的限制)。所需要更新或构造的是相对较小的代表进程地址空间状态的内部数据结构VAD。

  。

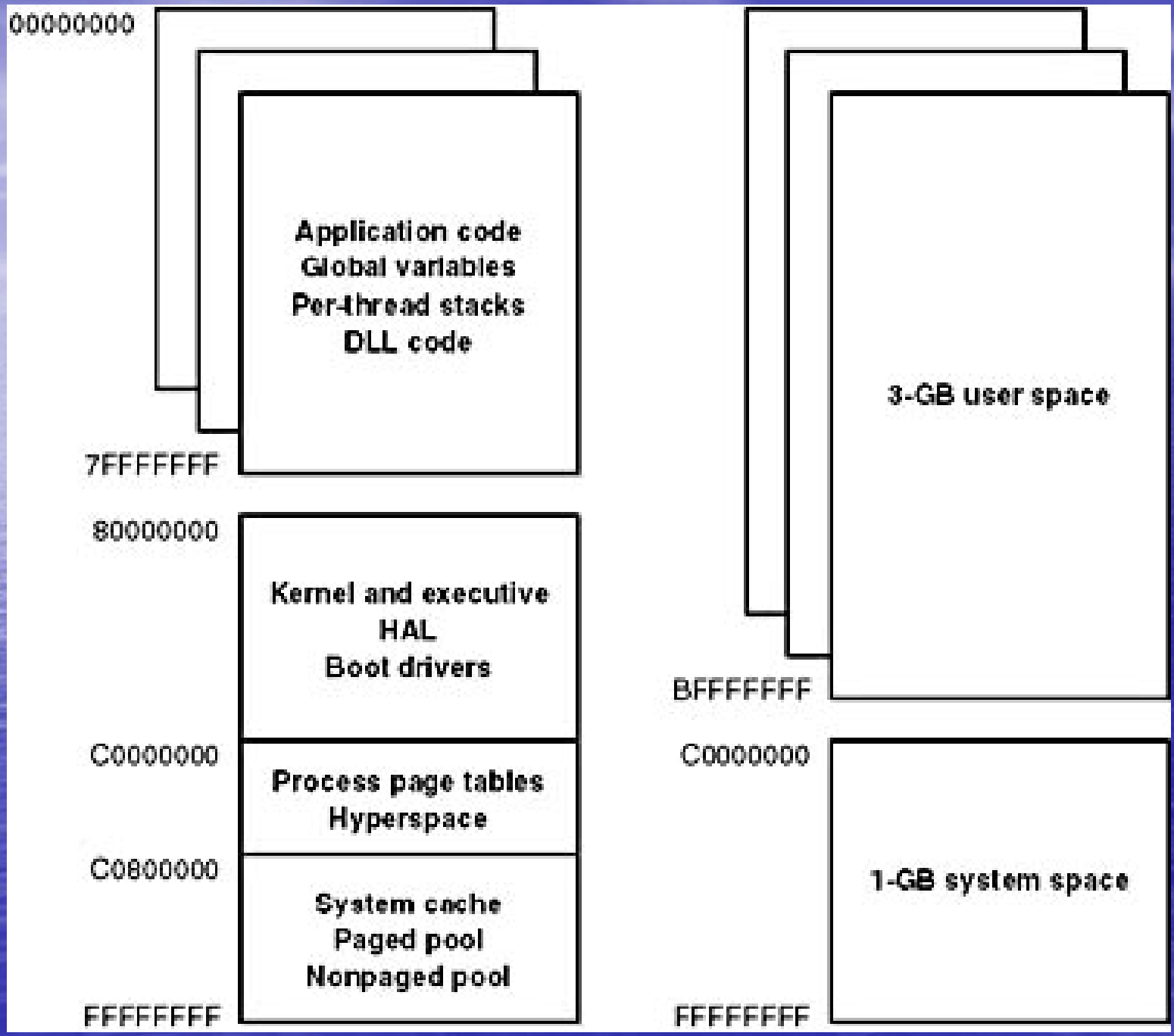
页文件

- 现代操作系统能够使磁盘空间看起来像内存一样，为进程提供了虚拟存储器。Windows2000／XP中磁盘上的部分通常称为“页文件”。如果计算机有64MB物理内存，同时在磁盘上有100MB的页文件，那么应用程序就可以认为计算机总共拥有164MB内存。
- 性能计数器中的Process：PageFileBytes实际上就是被提交的进程私有内存总和。这些内存可能有一些，或全部在页文件中，也可能全都不在页文件中。内存管理器一直在追踪已提交私有内存的使用情况，该情况对整个系统而言称为“提交量”，对各个进程而言称为“页文件限额”。(此外，内存使用情况并不反映页文件使用情况——它只反映提交的私有内存使用情况。)无论何时向虚拟地址提交物理页面，都会查询提交量和页文件限额。一旦达到系统整体提交限制(物理内存和页文件都满了)，这次虚存分配就会失败，直到一些进程释放内存(比如，当一个进程撤销)。

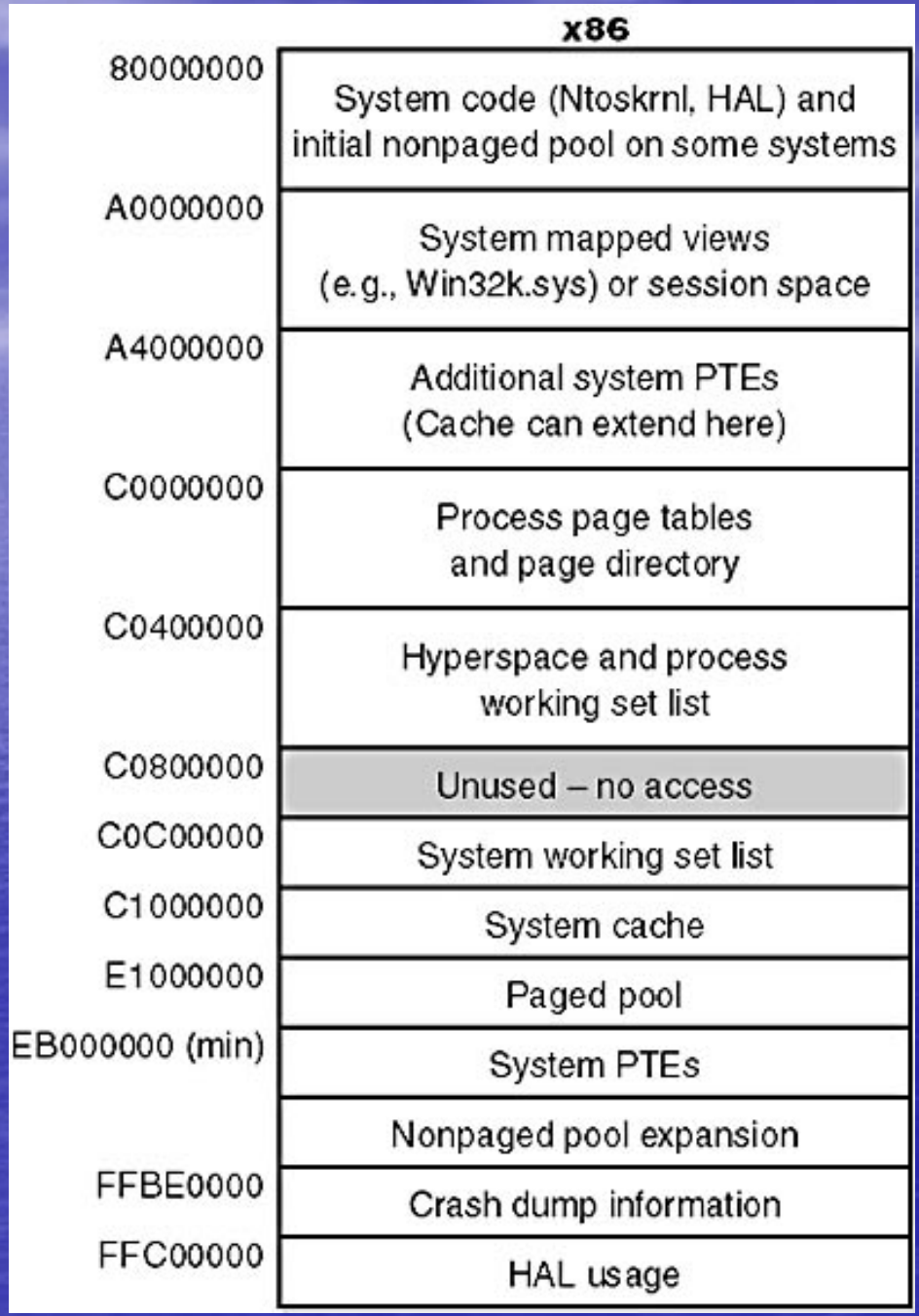
Windows 中的存储管理

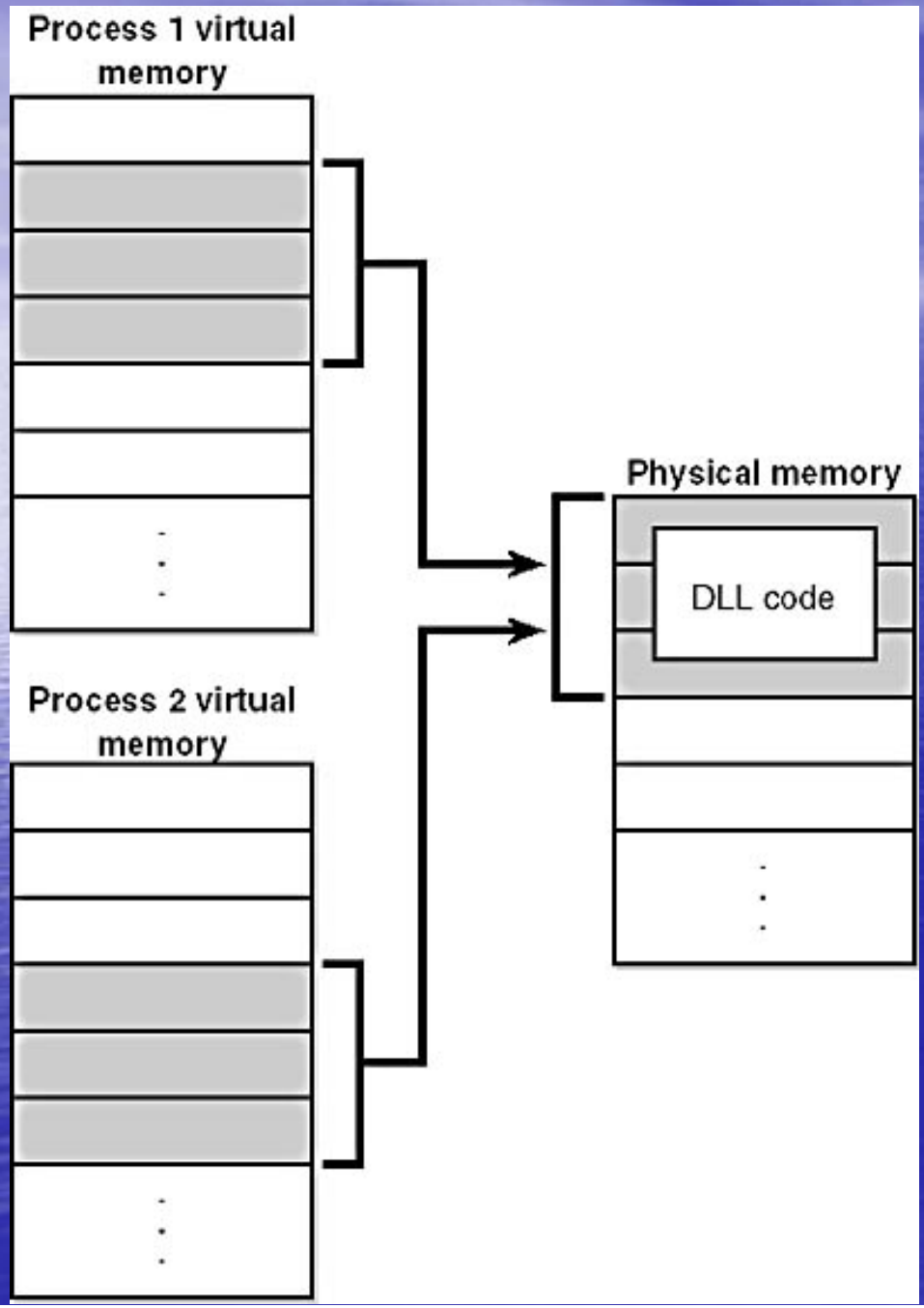
- windows使用的页面大小为4KB（2^{12}）。每个NT进程地址空间为4GB（2^{32}），其中：
- 用户存储区：在用户态和核心态都可访问的用户存储区为2GB；用户存储区为页交换区，可对换到外存；用户存储区的内容包括：
  - 专用进程地址空间：用户代码、数据和堆栈；
  - 线程环境块（TEB）：用户态代码可修改的线程控制信息；
  - 进程环境块（PEB）：用户态代码可修改的进程控制信息；
  - 共享用户数据页：系统存储区映像，为用户态可访问的系统空间，目的在于避免用户态与核心态的频繁切换；如：系统时间。

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

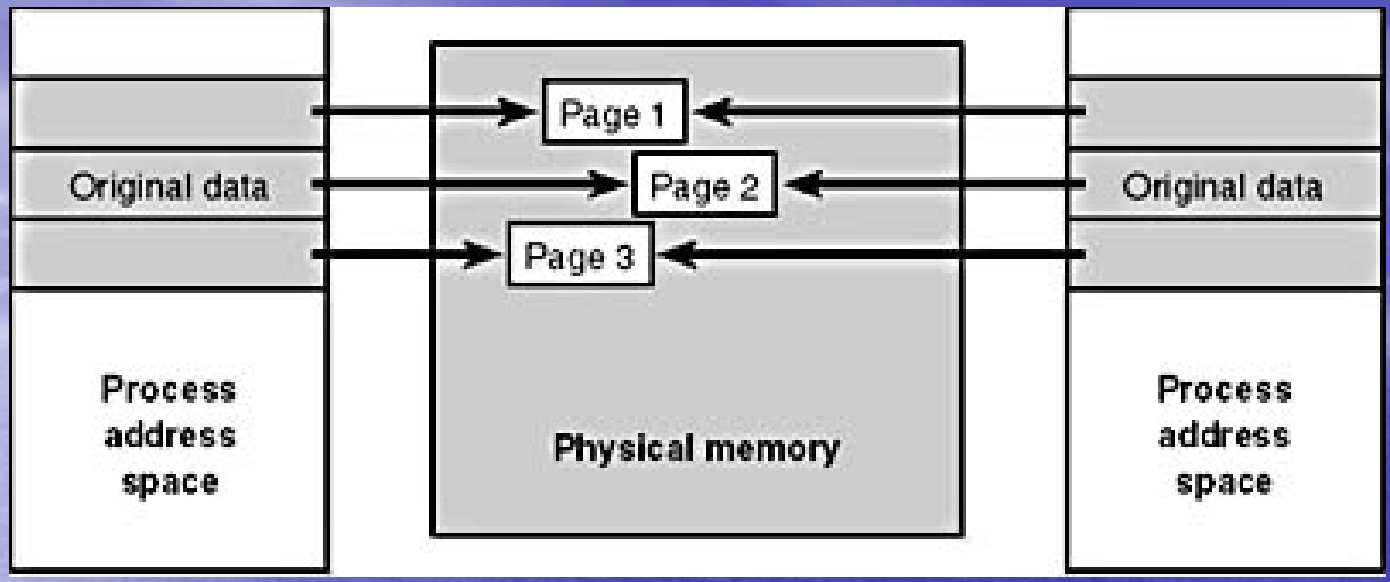

00000000
Application code
Global variables
Per-thread stacks
DLL code
7FFFFFFF
80000000
Kernel and executive
HAL
Boot drivers
C0000000
Process page tables
Hyperspace
C0800000
System cache
Paged pool
Nonpaged pool
FFFFFFFF
3-GB user space
BFFFFFFF
C0000000
1-GB system space
FFFFFFFF

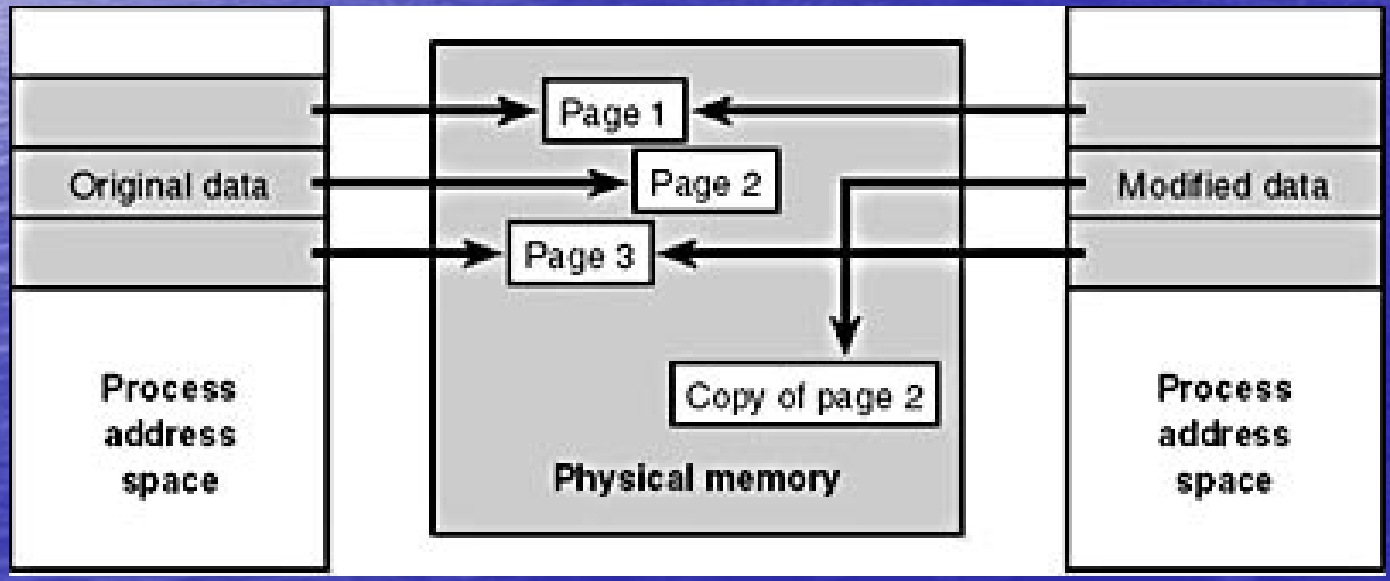
分x86下Windows 2000的系统空间划





Windows 2000的内存共享

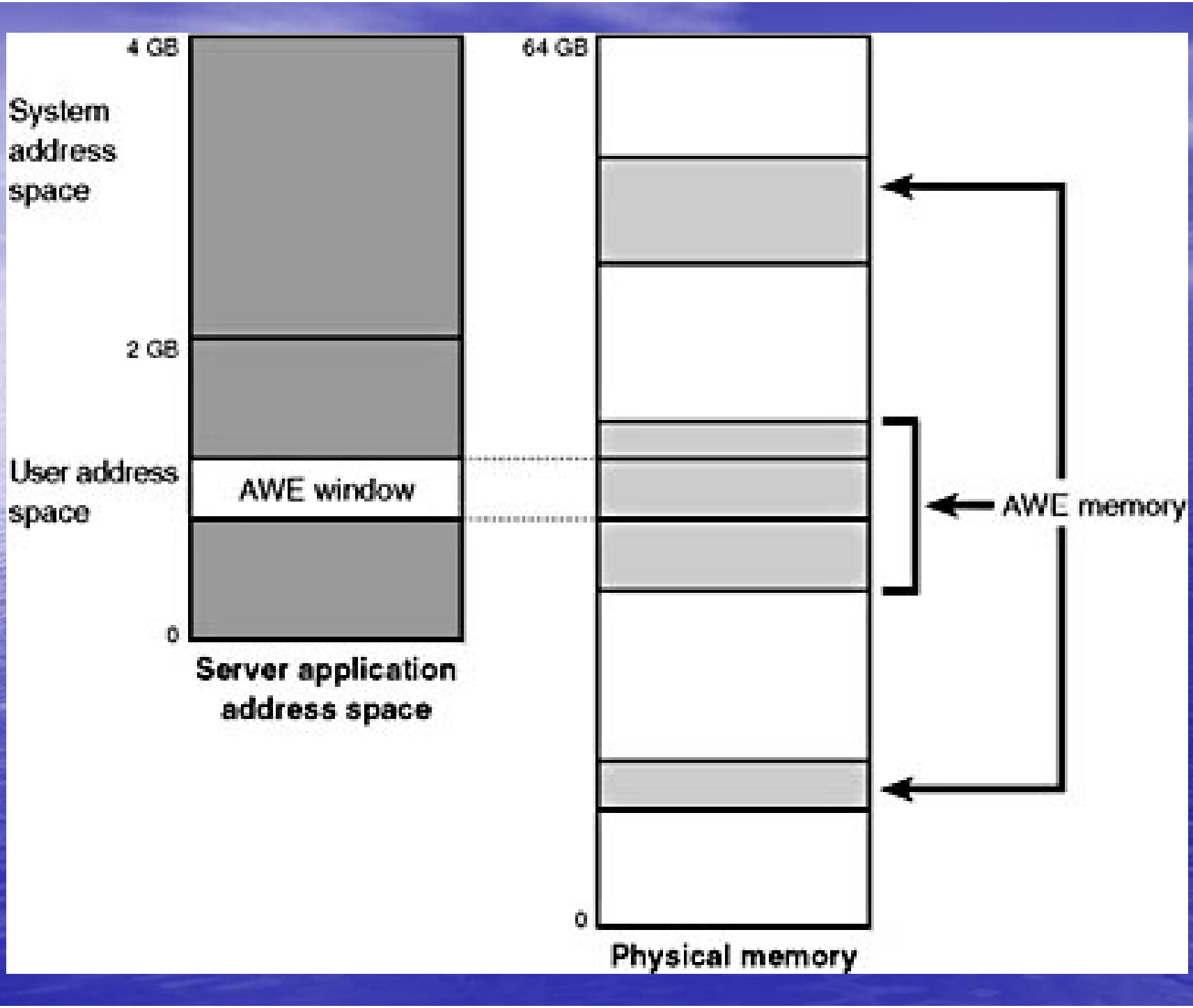

Page 1
Page 2
Page 3
Original data
Process address space
Physical memory
Original data
Process address space


Page 1
Page 2
Page 3
Original data
Process address space
Copy of page 2
Physical memory
Modified data
Process address space

写时复制

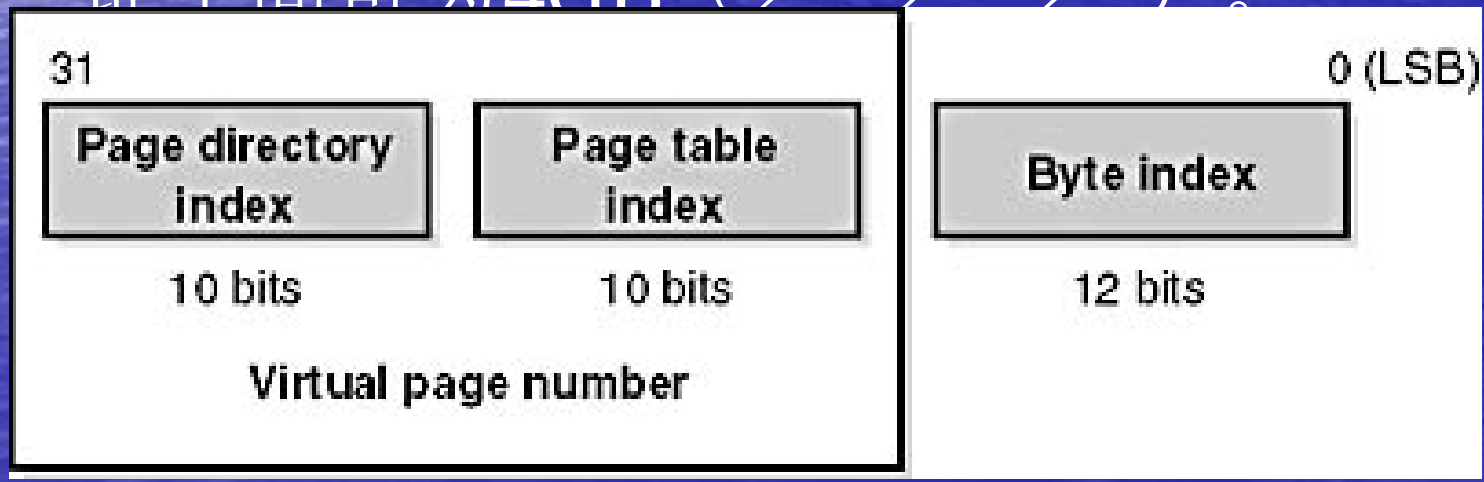
Windows2000的地址空间扩展

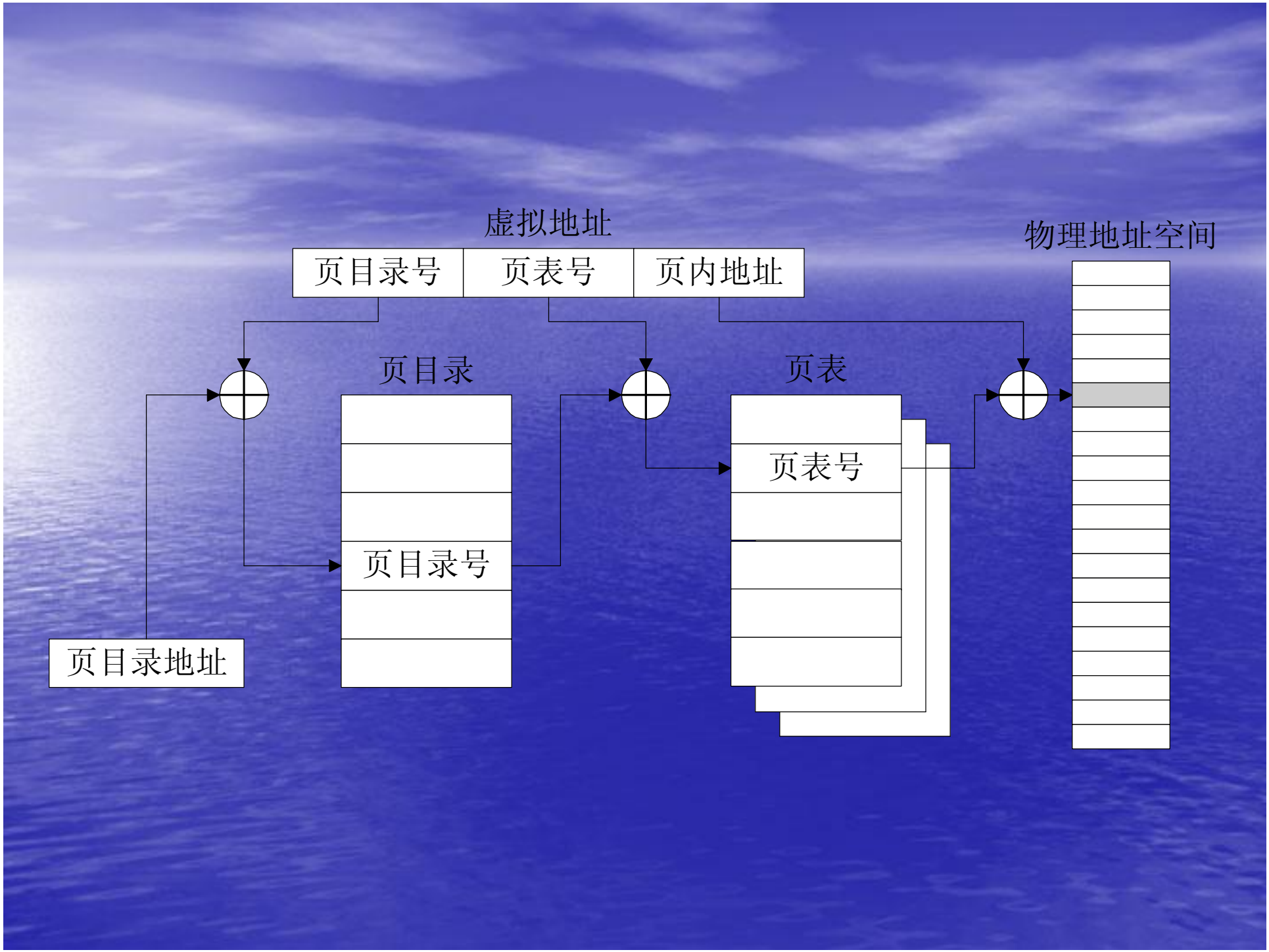
- 利用窗口映射(AWE, Address Windowing Extensions)方法可在一个进程中使用大于2GB或3GB的物理内存空间。使用步骤分成3步：
  - 分配物理内存(可大于2GB)；
  - 在进程虚拟地址空间创建一个窗口区域(小于2GB)；
  - 把物理内存空间的一个区域映射到窗口区域，从而可访问在区域的内存；

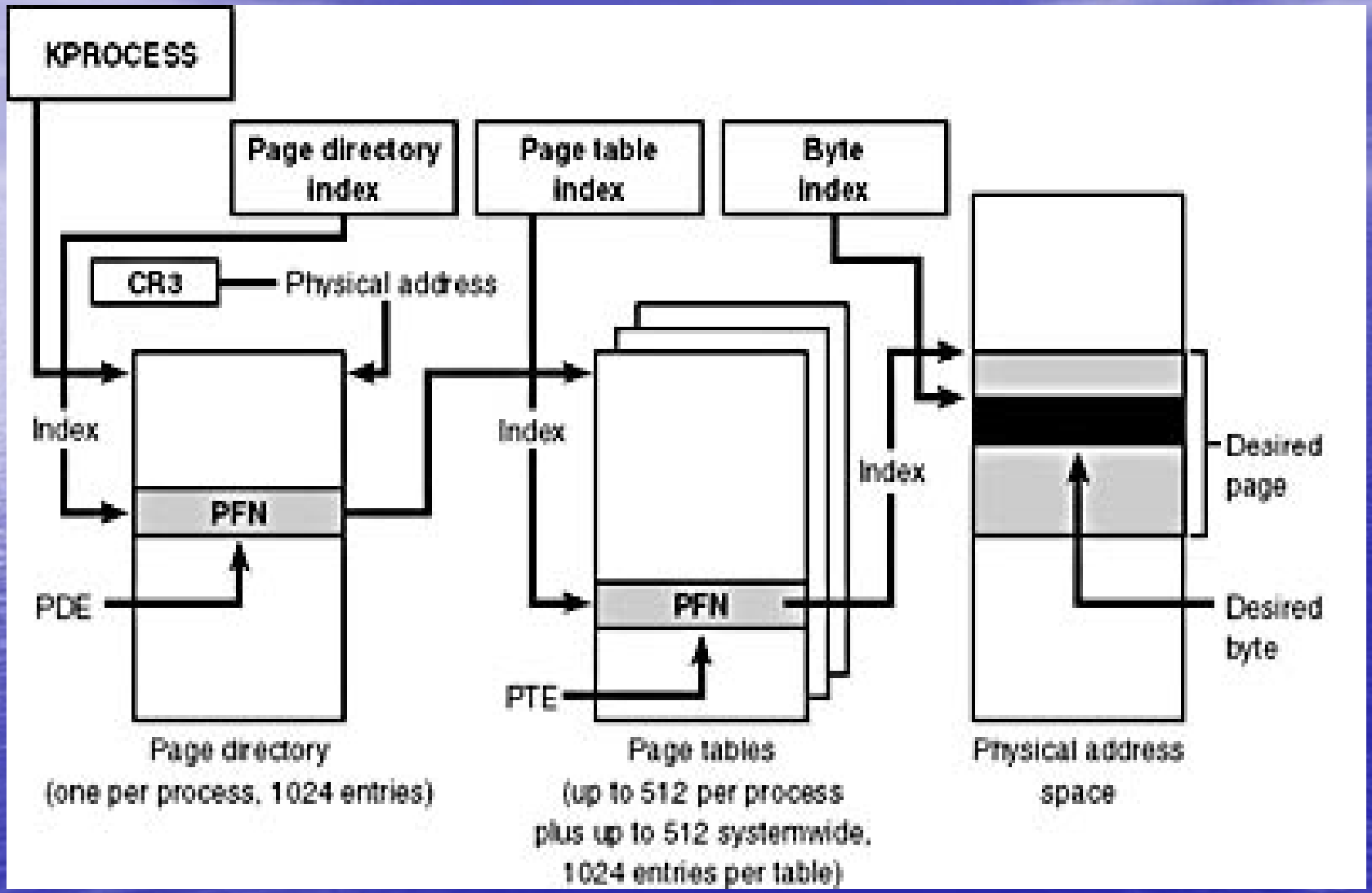

4 GB
System address space
2 GB
User address space
AWE window
0
Server application address space
64 GB
0
Physical memory
AWE memory

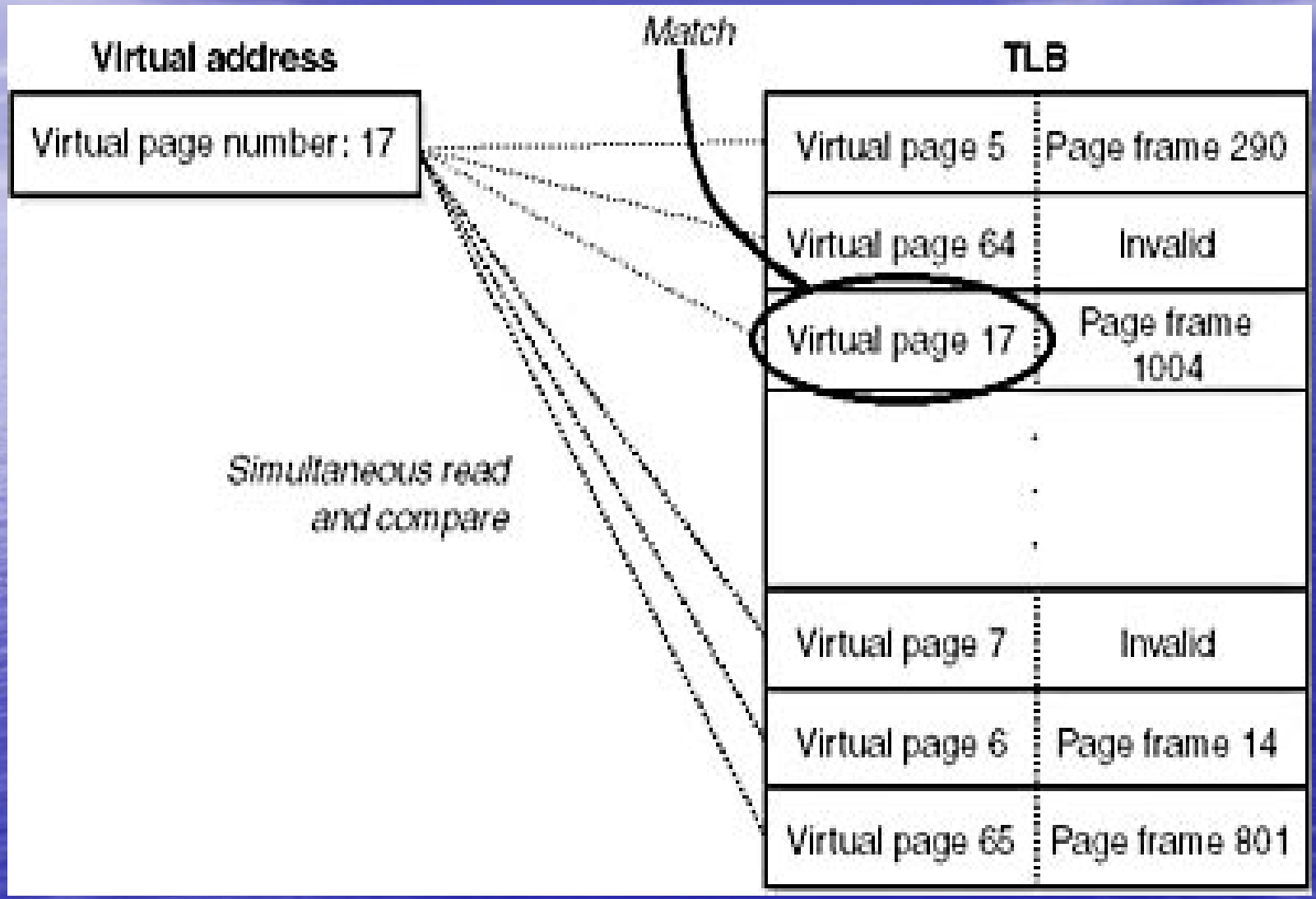
2. 地址转换机构

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



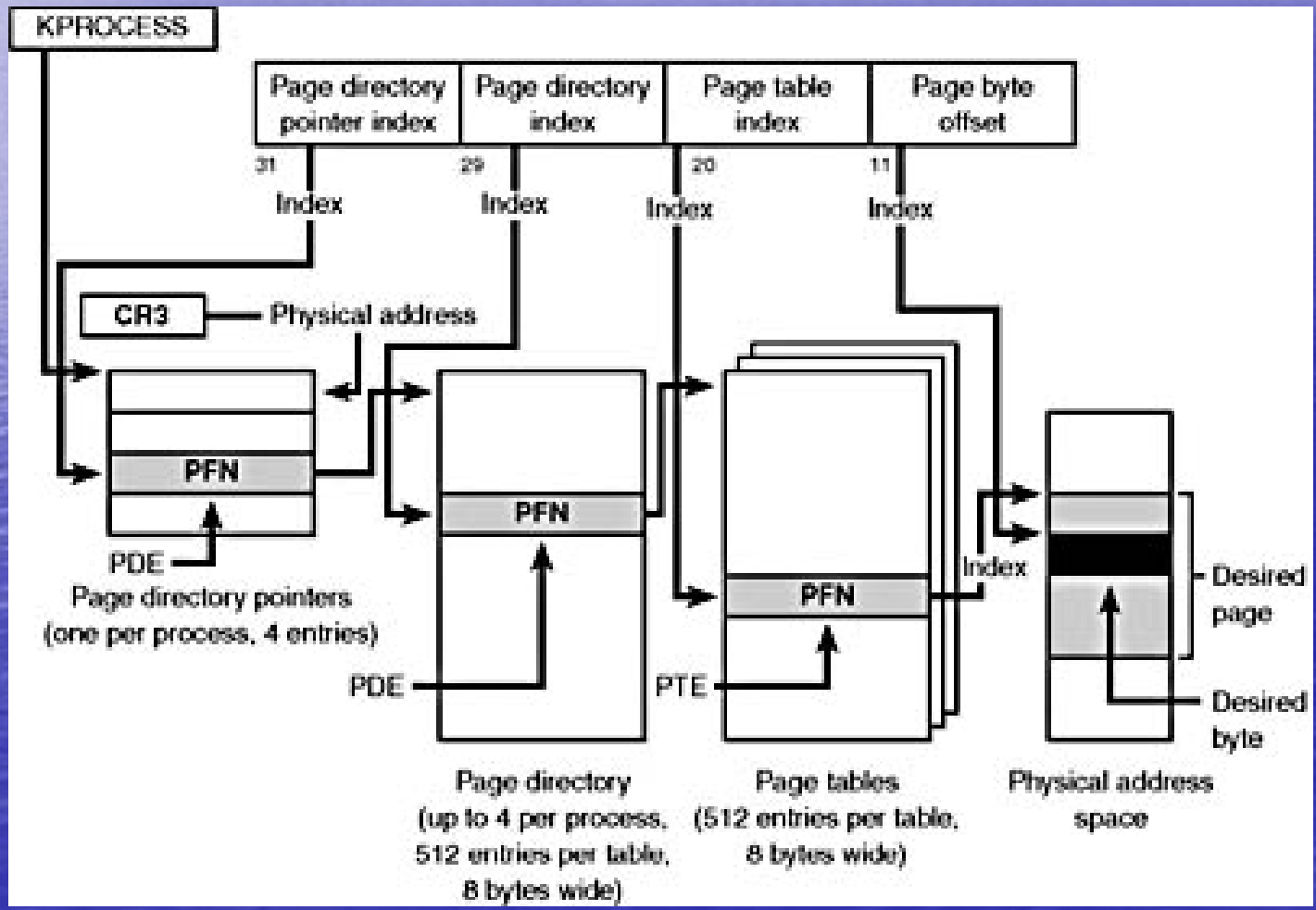

虚拟地址
页目录号
页表号
页内地址
物理地址空间
页目录
页表
页表号
页目录号
页目录地址


KPROCESS
Page directory index
Page table index
Byte index
CR3
Physical address
Index
Index
Index
PFN
PDE
PFN
PTE
Desired page
Desired byte
Page directory
(one per process, 1024 entries)
Page tables
(up to 512 per process
plus up to 512 systemwide,
1024 entries per table)
Physical address space


Virtual address
Virtual page number: 17
Match
TLB
Virtual page 5
Page frame 290
Virtual page 64
Invalid
Virtual page 17
Page frame 1004
Simultaneous read and compare
Virtual page 7
Invalid
Virtual page 6
Page frame 14
Virtual page 65
Page frame 801

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

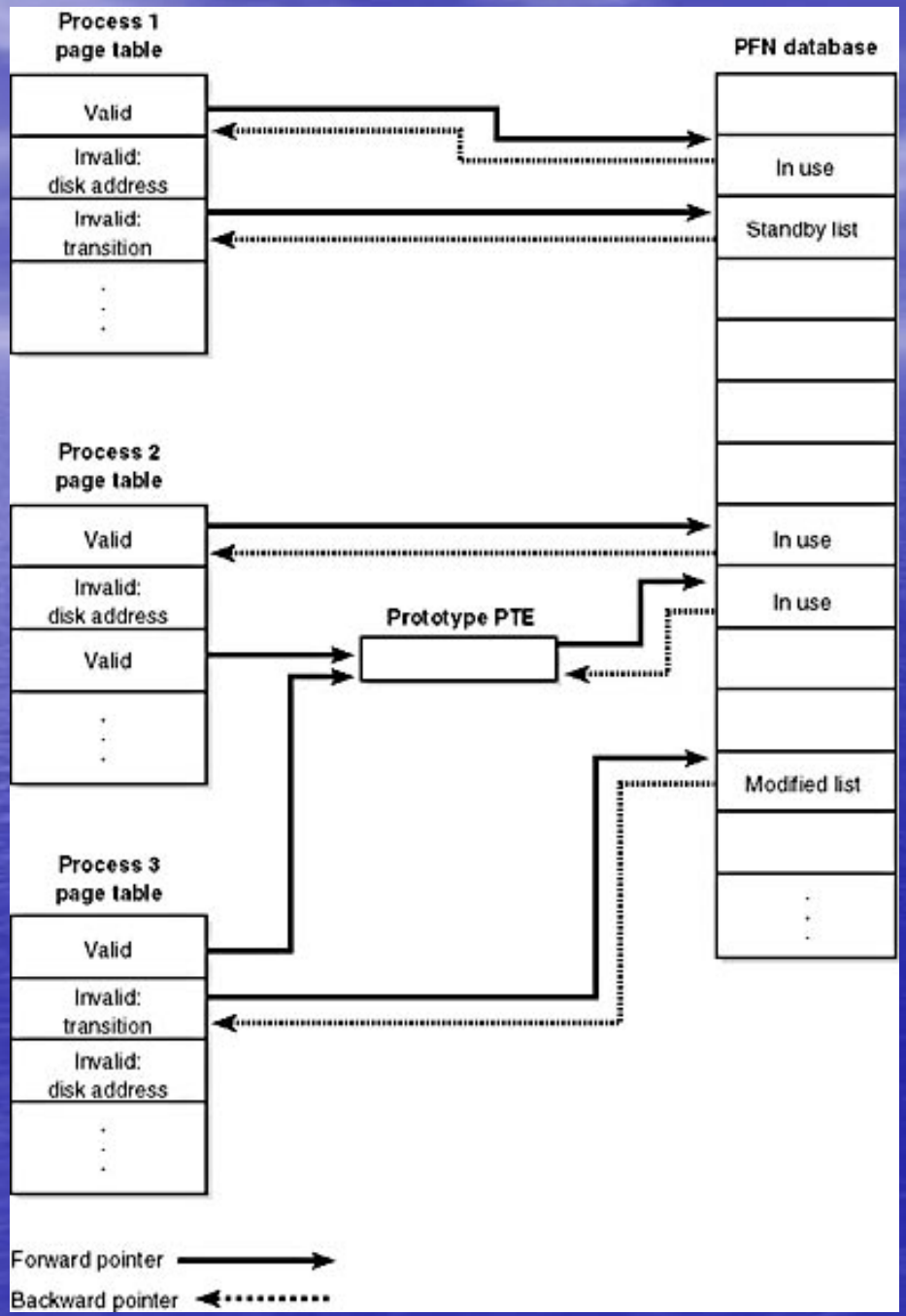
每个PDE和PTE表项都是8个字节，物理页面号为24Bit，可访问的物理地址空间可达64GB。

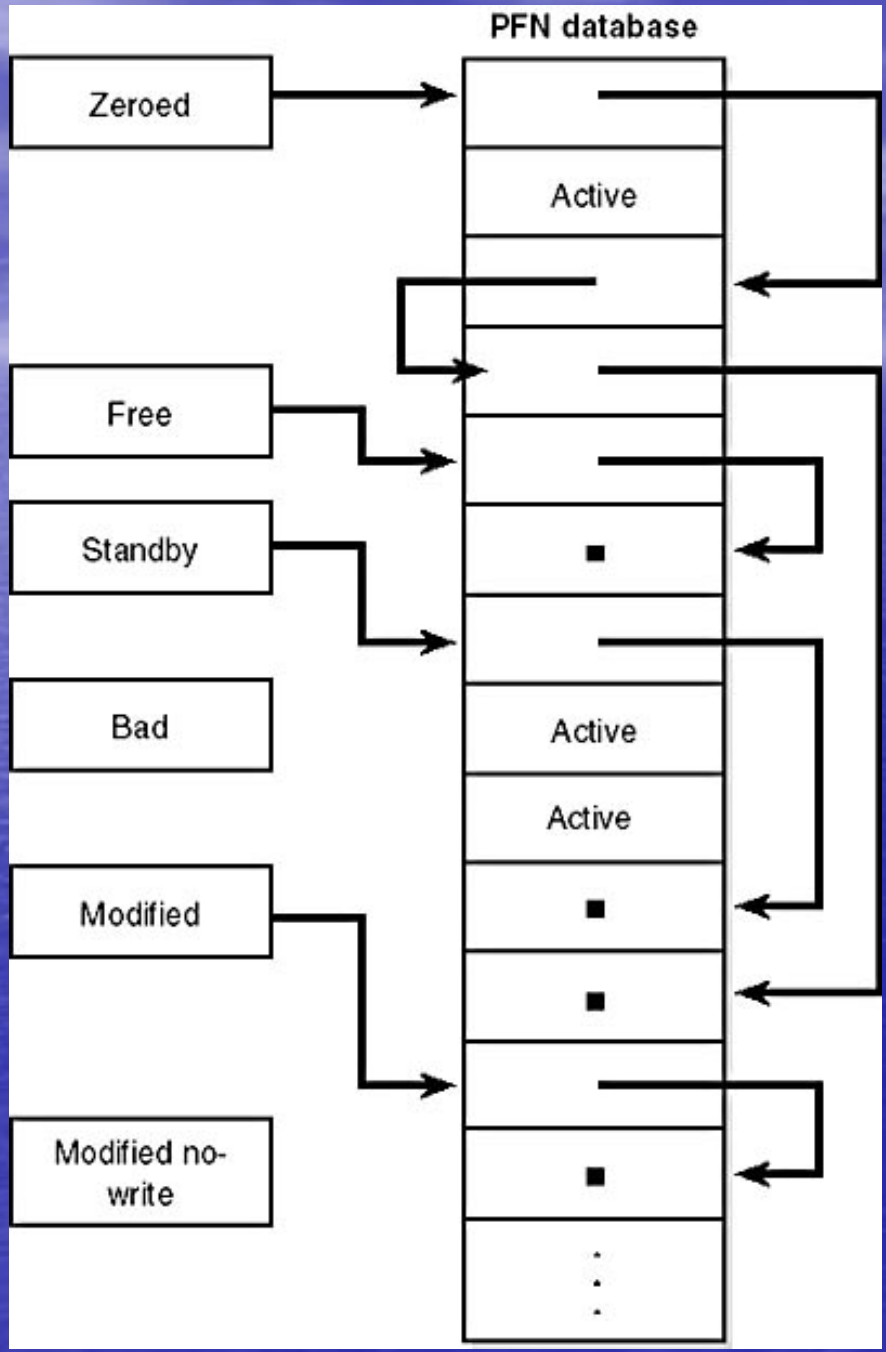


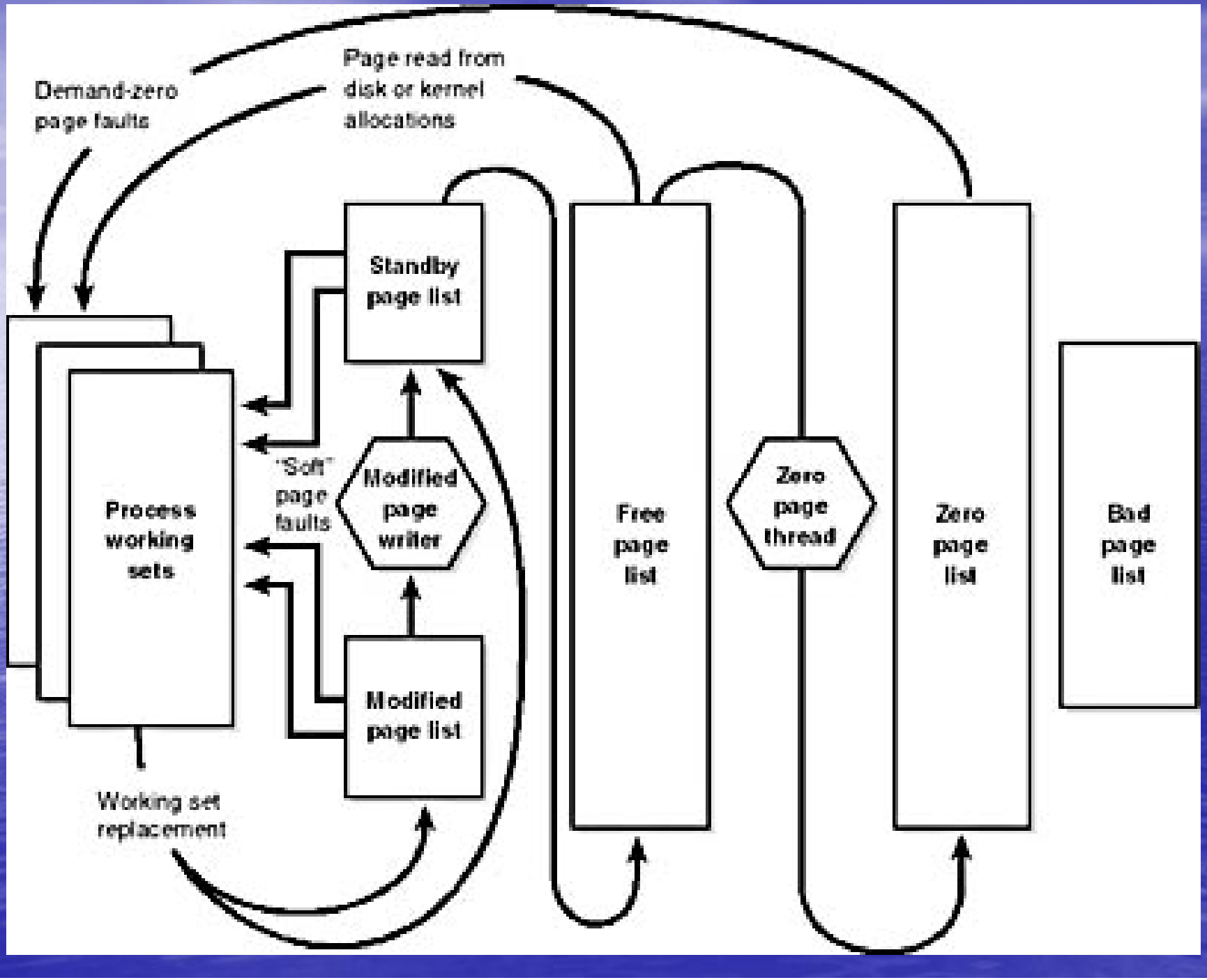
3. 页面状态

Windows 2000的页面有8种状态。

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


Process 1
page table
Valid
Invalid:
disk address
Invalid:
transition
Process 2
page table
Valid
Invalid:
disk address
Valid
Process 3
page table
Valid
Invalid:
transition
Invalid:
disk address
Prototype PTE
PFN database
In use
Standby list
In use
In use
Modified list
Forward pointer
Backward pointer


PFN database
Zeroed
Free
Standby
Bad
Modified
Modified no-write
Active
Active
Active


Demand-zero
page faults
Page read from
disk or kernel
allocations
Standby
page list
Process
working
sets
"Soft"
page
faults
Modified
page
writer
Free
page
list
Zero
page
thread
Zero
page
list
Bad
page
list
Modified
page list
Working set
replacement

4. 页面调度策略

页面调度策略包括取页策略、置页策略和淘汰策略。

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

5. 工作集策略

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

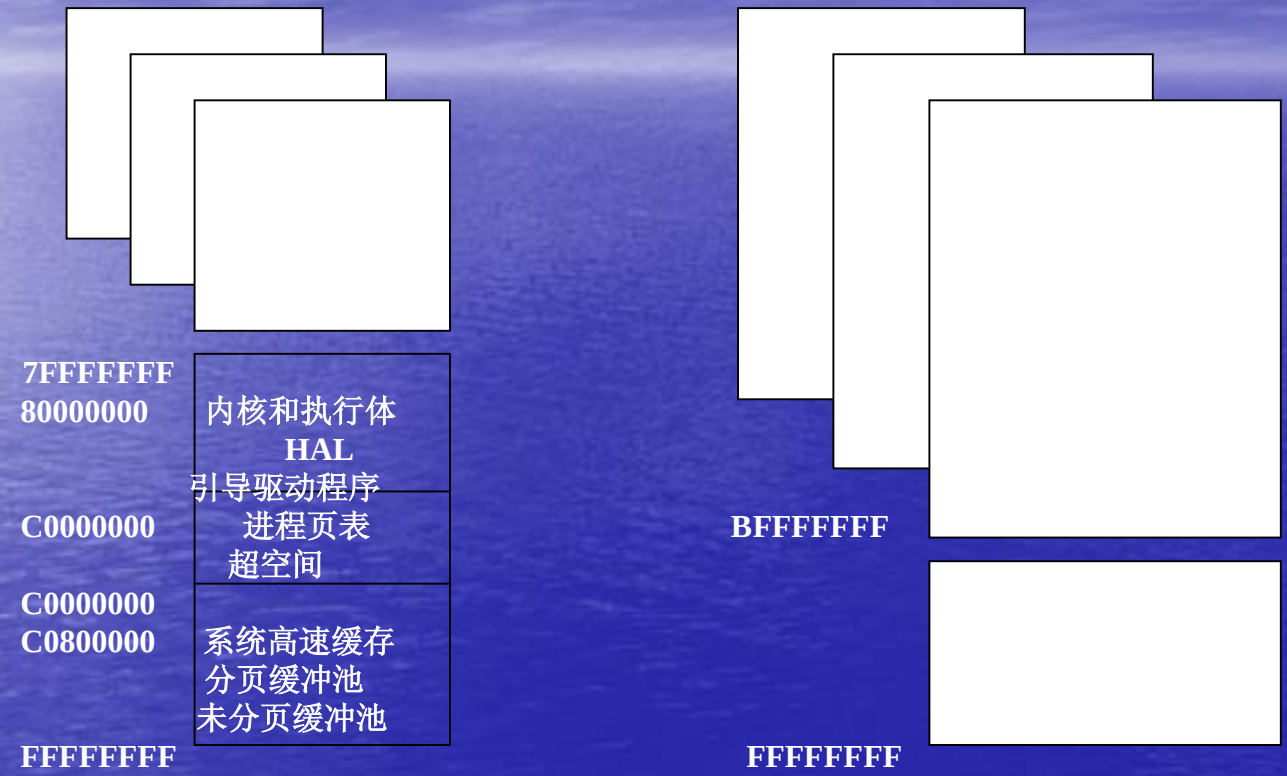
组成部分

- 一组执行体系统服务程序，用于虚拟内存的分配、回收和管理。大多数这些服务都是通过Win32 API 或内核态的设备驱动程序接口形式出现。
- 一个转换无效和访问错误陷阱处理程序用于解决硬件监测到的内存管理异常，并代表进程将虚拟页面装入内存。
- 六个的关键组件

- 工作集管理器（16优先)：当空闲内存低于某一界限时，便启动所有的内存管理策略，如：工作集的修整、老化和已修改页面的写入等。
- 进程/堆栈交换程序（23优先）：完成进程和内核线程堆栈的换入和换出操作。
- 已修改页面写入器（17优先）：将修改链表上的“脏”页写回到适当的页文件。

- 映射页面写入器（17优先）：将映射文件中脏页写回磁盘。
- 废弃段线程（18优先）：负责系统高速缓存和页面文件的扩大和缩小。
- 零页线程（0优先）：将空闲链表中的页面清零。

内存布局

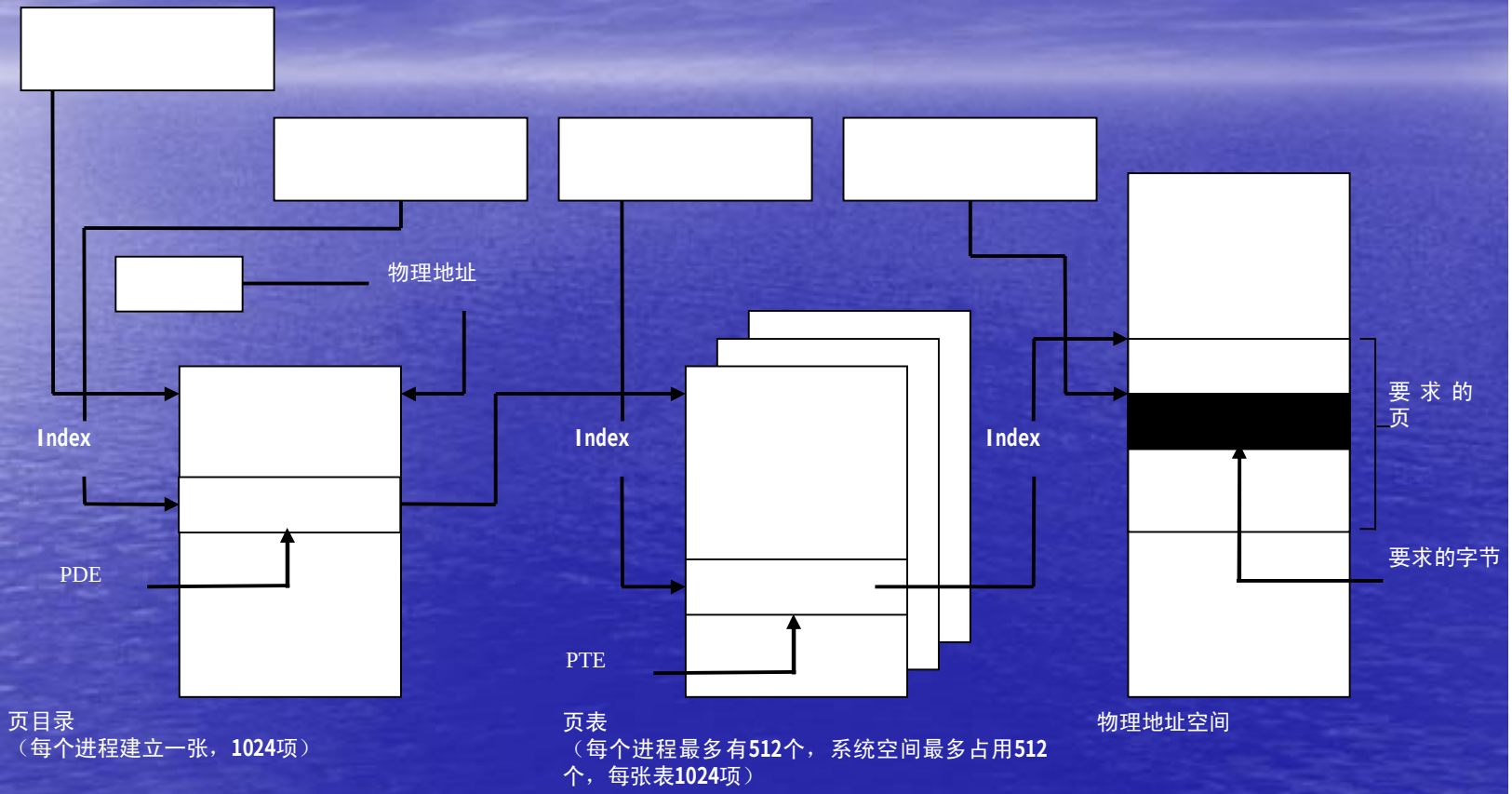

7FFFFFFF
80000000
内核和执行体
HAL
引导驱动程序
进程页表
超空间
C0000000
C0000000
C0800000
系统高速缓存
分页缓冲池
未分页缓冲池
FFFFFFFF
BFFFFFFF
FFFFFFFF


80000000
系统代码(Ntoskrnl,HAL) 和一些系统中初始的未分页缓冲池
A0000000
系统映射视图（例如，Win32k.sys)或者会话空间
A4000000
附加的系统PTE（高速缓存可以扩展到这）
C0000000
进程的页表和页目录
C0400000
超空间和进程工作集列表
C0800000
没有使用,不可访问
C0C00000
系统工作集列表
C1000000
系统高速缓存
E1000000
分页缓冲池
EB000000(min)
系统PTE
未分页缓冲池扩充
FFBE0000
故障转储信息
FFC00000
HAL使用

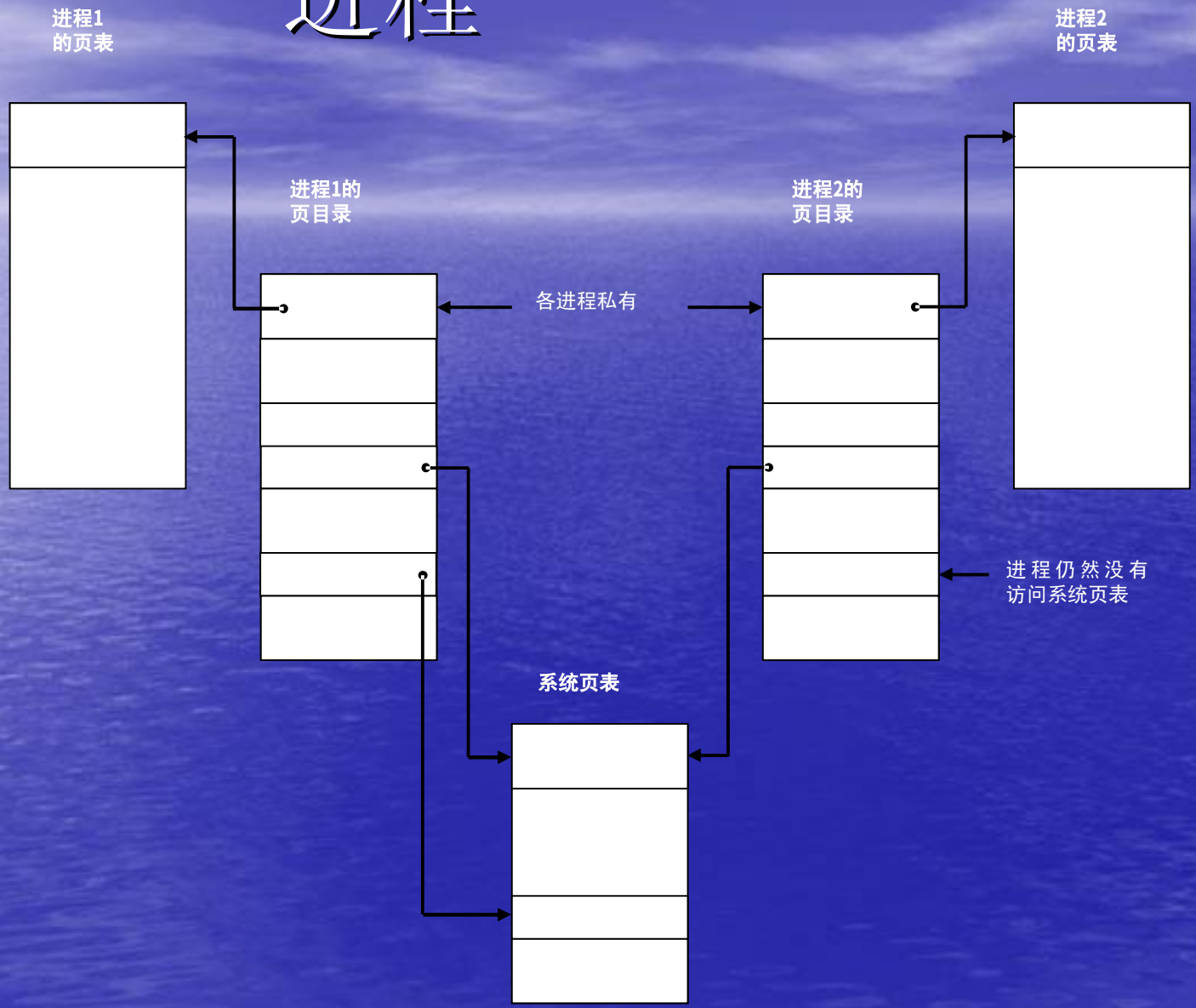
- 系统代码　包括操作系统映像、HAL和用于引导系统的设备驱动程序。
- 系统映射视图　用来映射Win32子系统可加载的核心态部分Win32k.sys，以及它使用的核心态图形驱动程序。
- 会话空间　用来映射一个用户的会话信息。
- 进程页表和页目录　描述虚拟地址映射的结构。
- 超空间　一个特殊的区域用来映射进程工作集链表，并为创建临时映射物理页面。

- 系统工作集链表　描述系统工作集的工作集链表数据结构。
- 系统高速缓存　用来映射在系统高速缓存中打开的文件的虚拟空间。
- 分页缓冲池　可分页系统内存堆。
- 系统页表项　系统PTE缓冲池，用来映射系统页面。
- 非分页缓冲池　不可分页的系统内存堆。

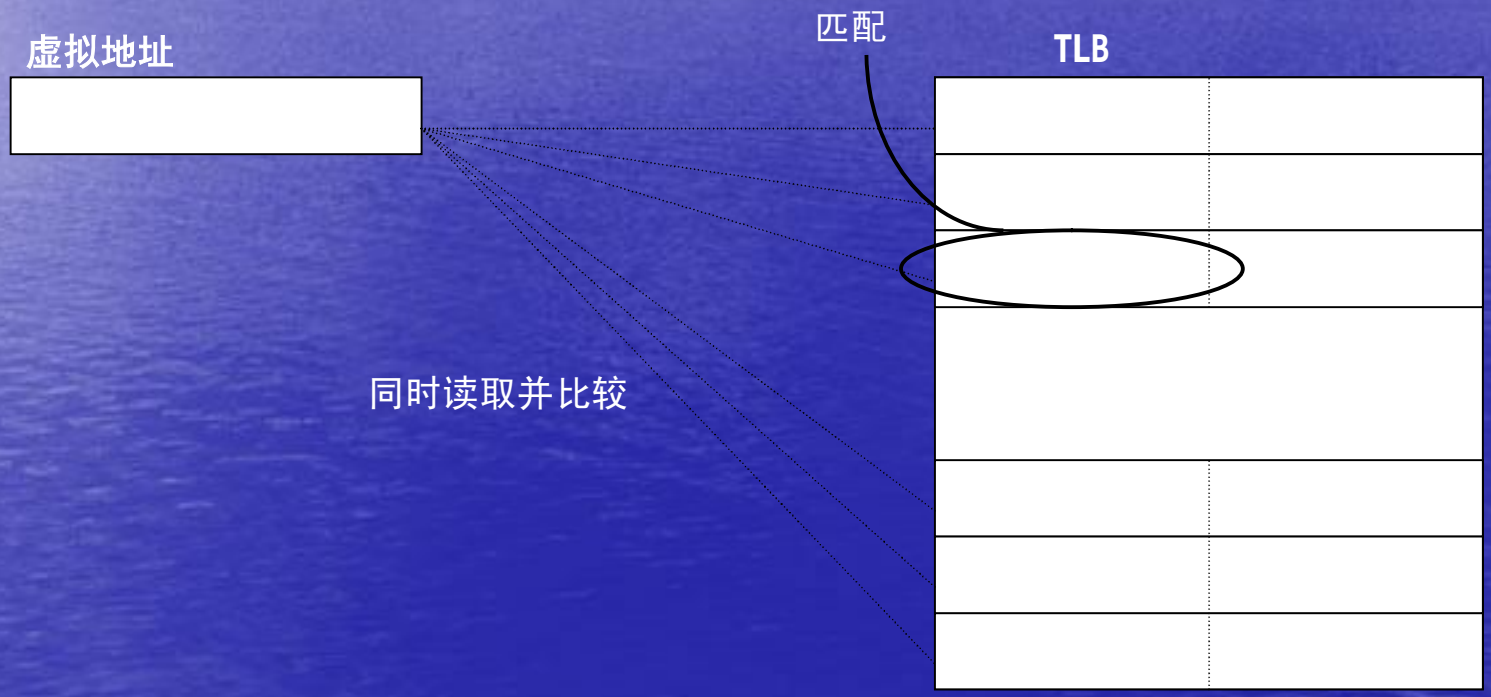
地址变换过程


物理地址
Index
Index
Index
PDE
PTE
要求的页
要求的字节
页目录
（每个进程建立一张，1024项）
页表
（每个进程最多有512个，系统空间最多占用512个，每张表1024项）
物理地址空间

进程



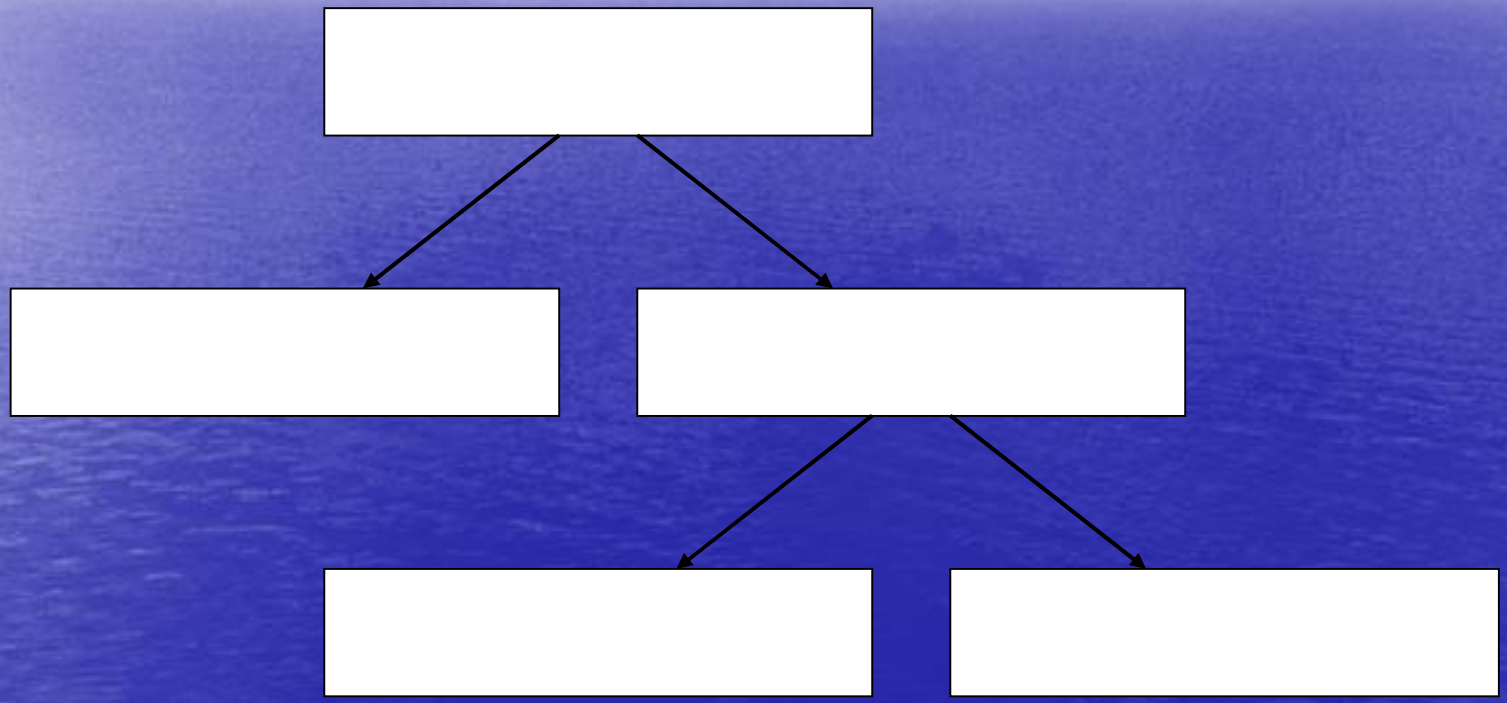
快表TLB



内存分配方式

- 以页单位的虚拟内存函数（VirtualXxx），
  –保留与提交
- 内存映射文件函数（CreateFileMapping, MapViewOfFile），
- 堆函数（HeapXxx 和早期的接口LocalXxx和GlobalXxx）。

虚拟地址描述符



内存映射文件

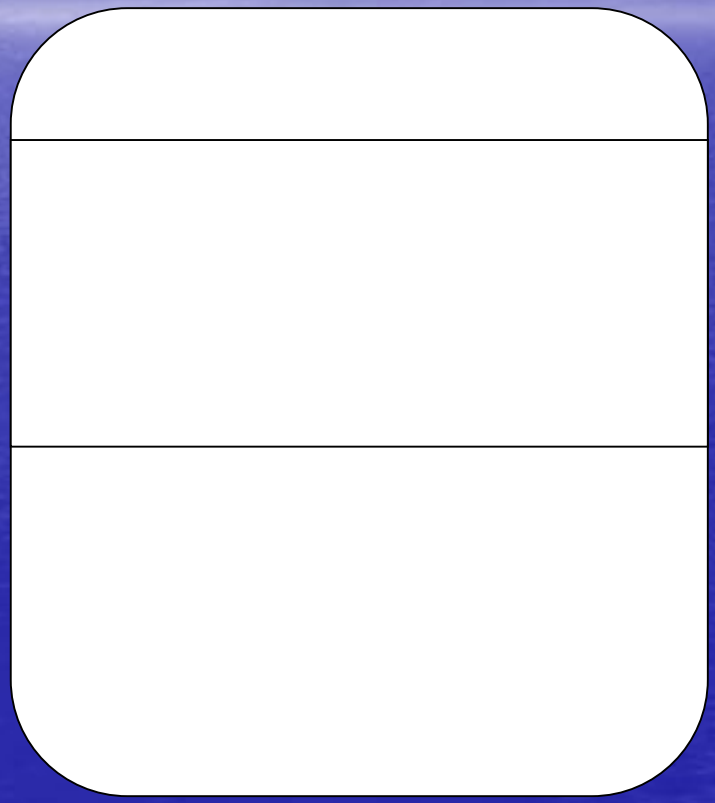
- 加载和执行.exe和dll文件，这可以节省应用程序启动所需的时间；
- 访问磁盘上的数据文件，这可以减少文件I/O，并且不必对文件进行缓存；
- 实现多个进程间的数据共享。

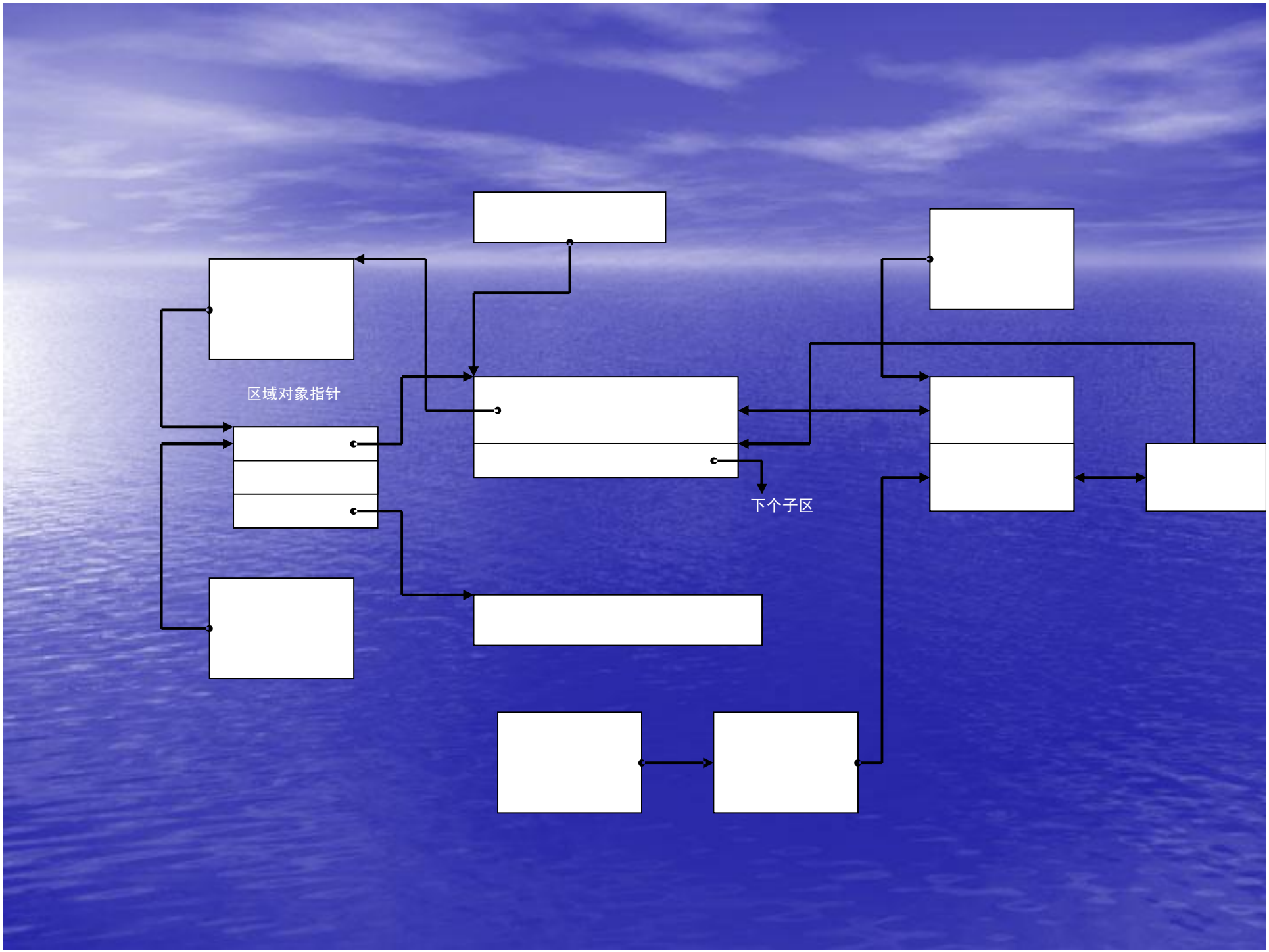
区域对象(section object)

对象类型

对象体属性

服务程序




区域对象指针
下个子区

建立过程

- 打开文件，区域对象可以连接到已打开的磁盘文件（映射文件），或是已提交的内存（提供共享内存）。
- 可以调用Win32函数CreateFileMapping创建区域对象，其参数包括映射到区域对象的文件句柄（或是INVALID_HANDLE_VALUE表示页文件支持区域）。如果区域有名字，其它进程可以用OpenFileMapping打开它。
- 设备驱动程序也可以使用ZwOpenSection, ZwMapViewOfSection, 和ZwUnmapViewOfSection函数操纵区域对象。
- MapViewOfFile函数映射区域对象的一部分，并指定映射范围。

堆函数

- 缺省进程堆，通常是1MB大小
- HeapCreate函数创建另外的私有堆，HeapDestroy删除。
- 串行化选项。

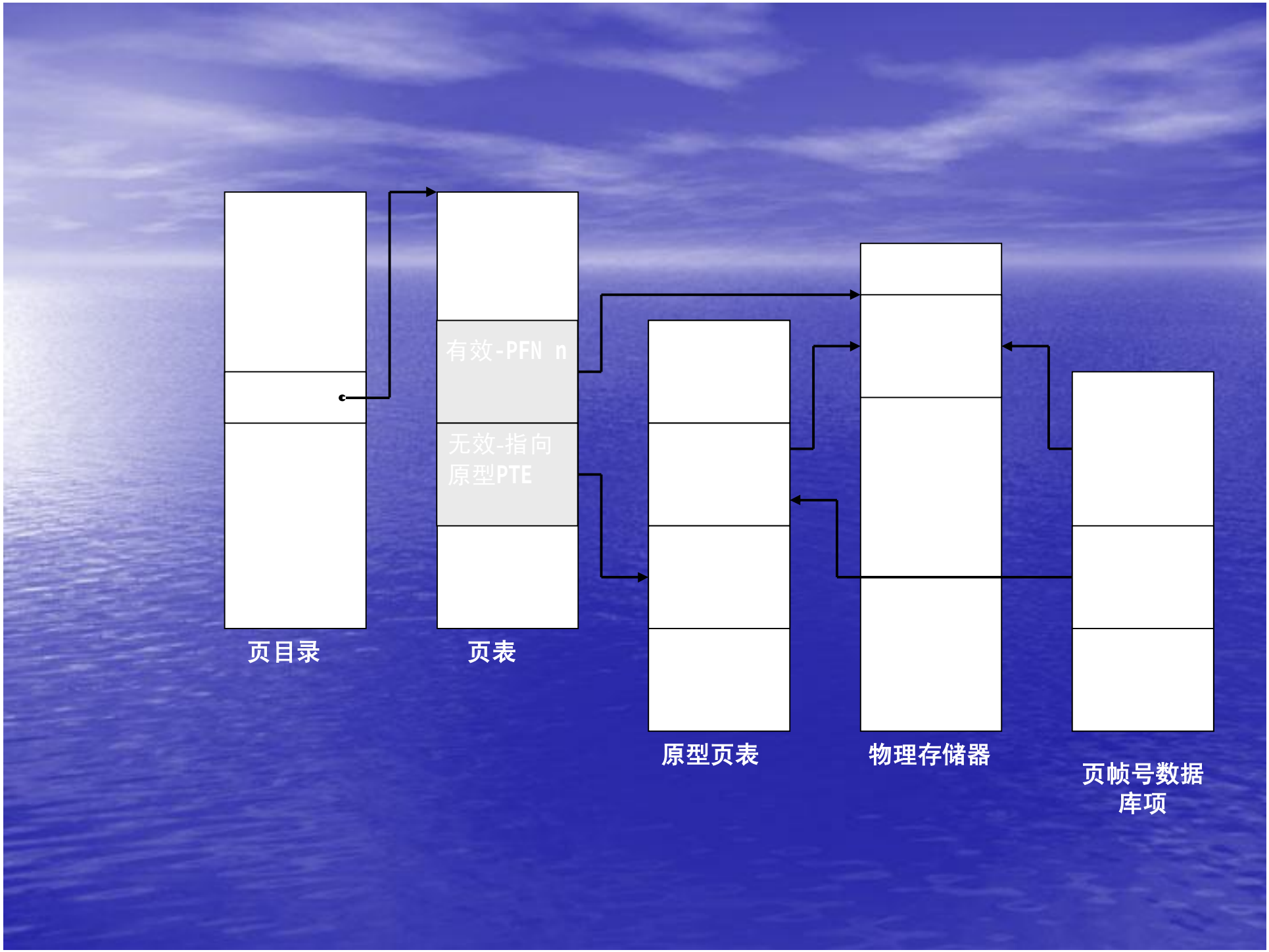
系统内存分配

- *非分页缓冲池* 由系统虚拟地址组成，它们长期驻留在物理内存中，在任何时候都可以被访问到（从任何IRQL级和任何进程上下文），而不会发生页错误。需要未分页缓冲池的一个原因是：页错误不能满足在DPC/调度级或更高。
- *分页缓冲池* 是系统可以被分页和分出系统的空间中虚拟内存的一个区域。不必从DPC/调度级或更高一级访问内存的设备驱动程序可以使用分页缓冲池。它从任何进程上下文都是可访问的。

- 系统有两种非分页缓冲池：一种在一般情况下使用，另一种小型的（4页）缓冲池在非分页缓冲池已满并且调用者不能允许分配失败时，紧急使用。
- 单处理机系统有三个分页缓冲池；多处理机系统有五个。
- 后备链表(Look-Aside Lists)。
- Ex…

缺页处理

- 无效的页表项
  - 页文件
  - 请求零页
  - 转换
  - 未知
- 原型页表项


有效-PFN n
无效-指向
原型PTE
页目录
页表
原型页表
物理存储器
页帧号数据
库项

页面调入I/O

- 向文件（页或映射文件）发出读操作来解决缺页问题
- 同步的

问题

- 同一进程中的另一线程，或一个其它的进程，都可能由于一个相同的页面导致缺页错误。（称为“冲突页错误”，将在下节中介绍）。
- 页面可能已经从虚拟地址空间中被删除（并重新映射）。
- 页面的保护限制可能发生了变化。
- 错误可能是由一个原型页表项引发的，并且这个原型页表项所映射的页面可能并不在工作集中。

冲突页错误(collided page fault)

- 页面调度程序检测
- 等待操作
- I/O操作完成后，所有等待该事件的线程都会被唤醒
- 第一个获得页框号数据库锁的线程负责执行页面调入完成操作。

页文件

- 最多16个页文件
- 以非压缩的形式被创建

工作集

- 进程工作集
- 系统工作集

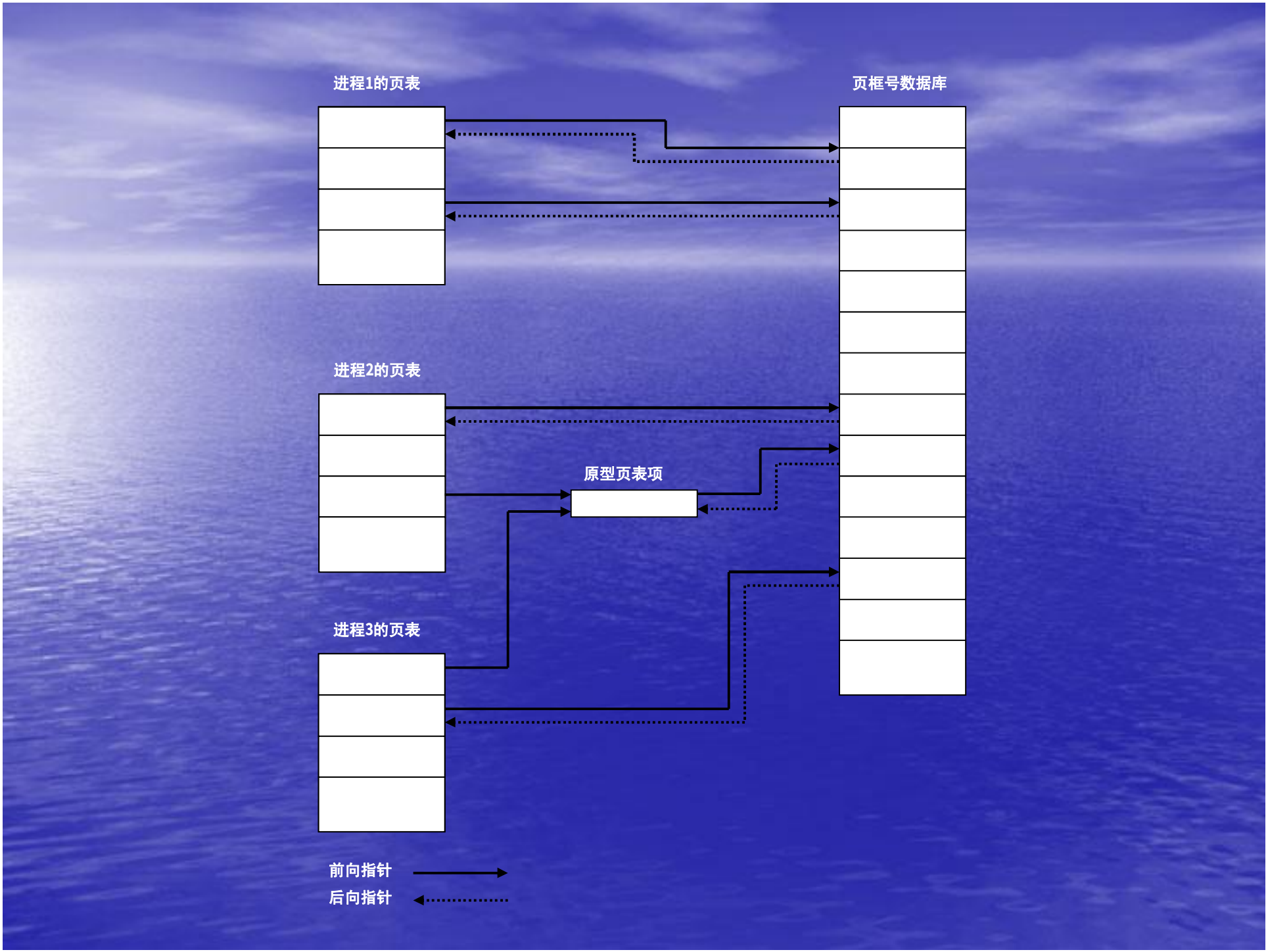
系统工作集

- 系统高速缓存页面
- 分页缓冲池
- Ntoskrnl.exe中可分页的代码和数据
- 设备驱动程序中可分页的代码和数据
- 系统映射视图（部分映射在0xA0000000处，如Win32k.sys）

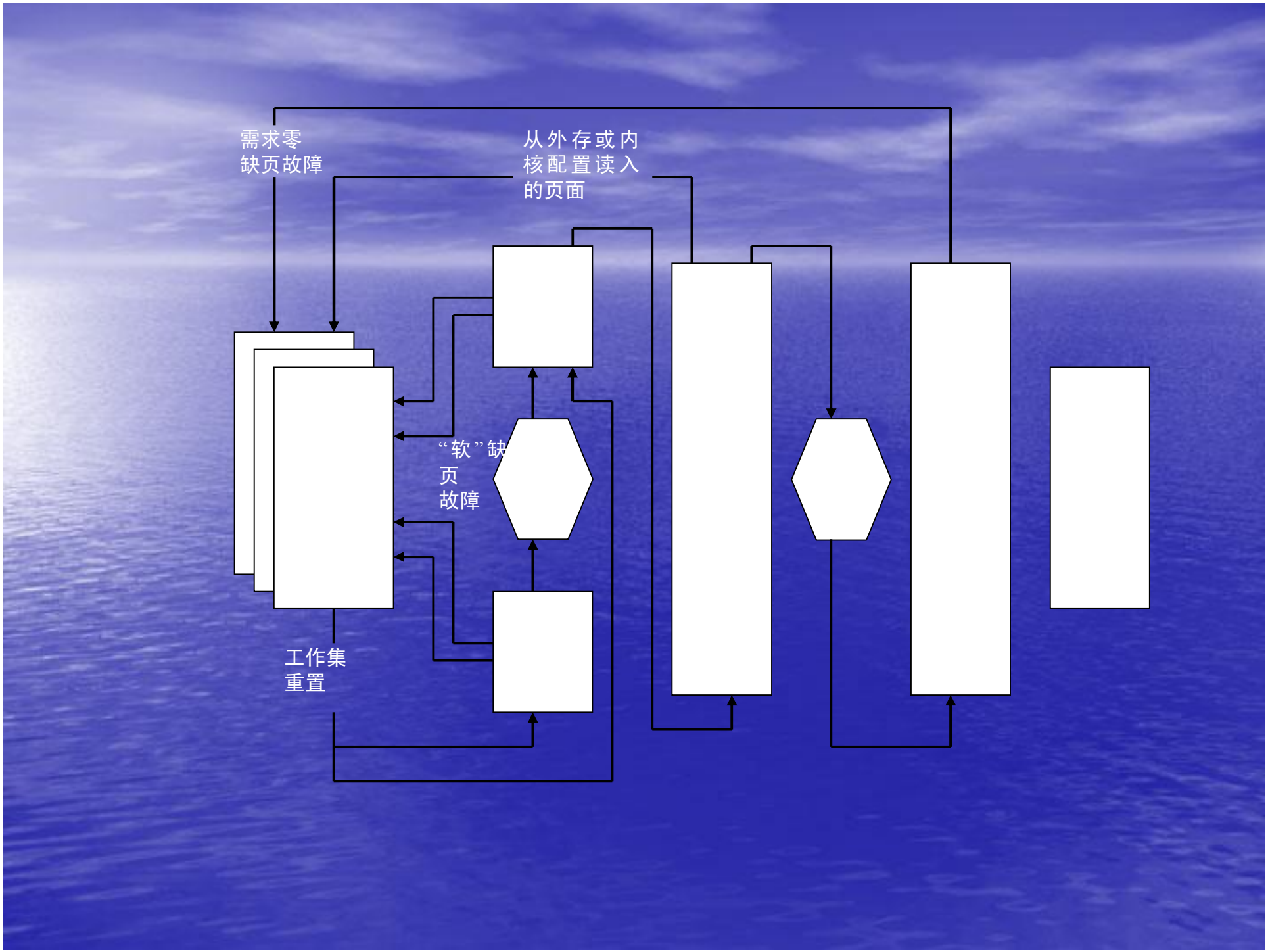
- 取页策略：内存管理器利用请求式页面调度算法以及簇方式将页面装入内存
- 置页策略：选择页框应使CPU内存高速缓存不必要的震荡最小
- 换页策略
  - 在多处理器系统中，Windows 2000/XP采用了局部先进先出置换策略。而在单处理器系统中，Windows 2000/XP的实现更接近于最近最少使用策略（LRU）(称为“轮转算法”，用于大多数版本的UNIX)。

物理内存管理

- 活动（又称有效）
- 过渡(Transition)
- 后备(stand by)
- 修改
- 修改不写入
- 空闲
- 零初始化(zeroed)
- 坏


进程1的页表
页框号数据库
进程2的页表
原型页表项
进程3的页表
前向指针
后向指针

物理内存管理


需求零
缺页故障
从外存或内
核配置读入
的页面
“软”缺
页
故障
工作集
重置

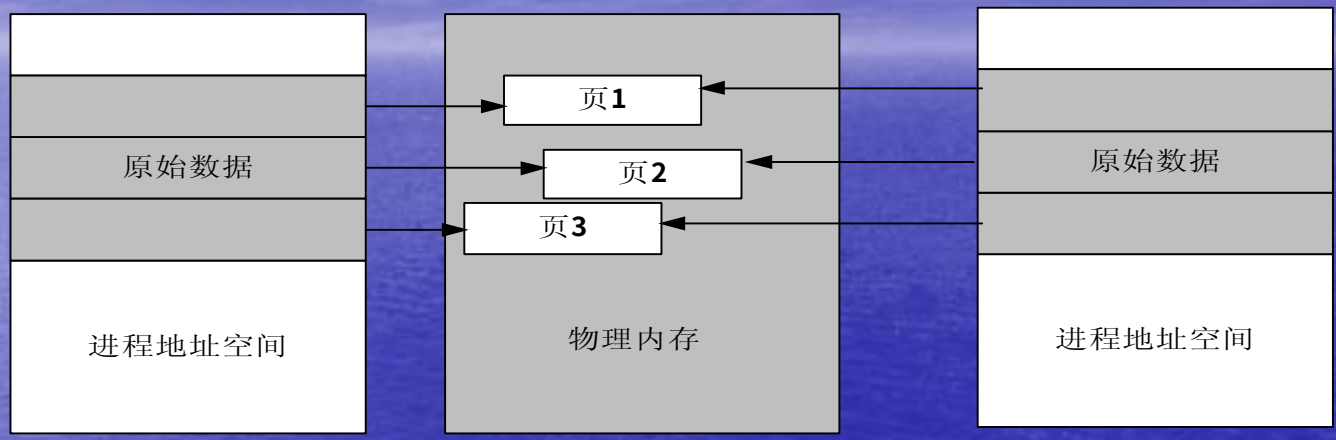
锁内存

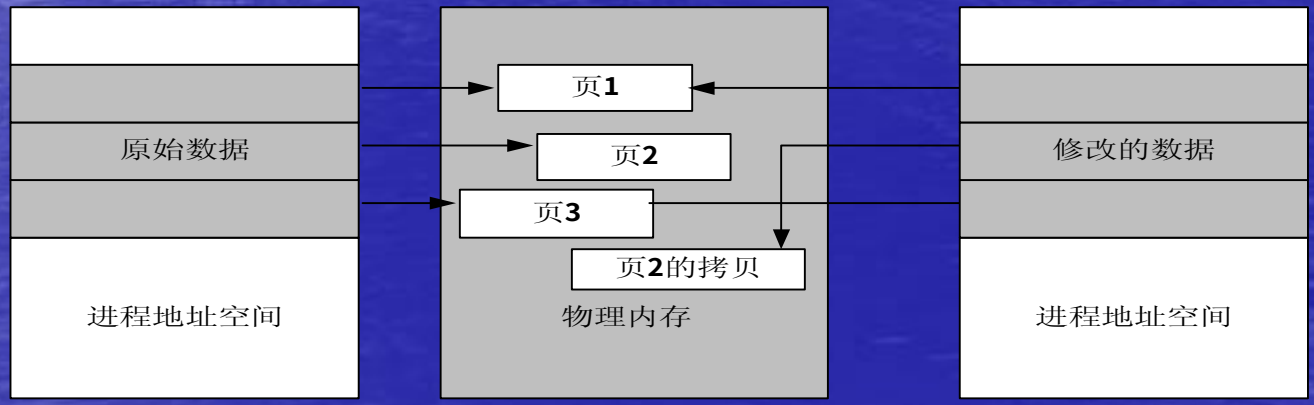
- 设备驱动程序可以调用核心态函数MmProbeAndLockPages，MmLockPagableCodeSection，MmLockPagableDataSection，或者MmLockPagableSection ByHandle。
- Win32应用程序可以调用VirtualLock函数锁住进程工作集中的页面。

内存保护机制

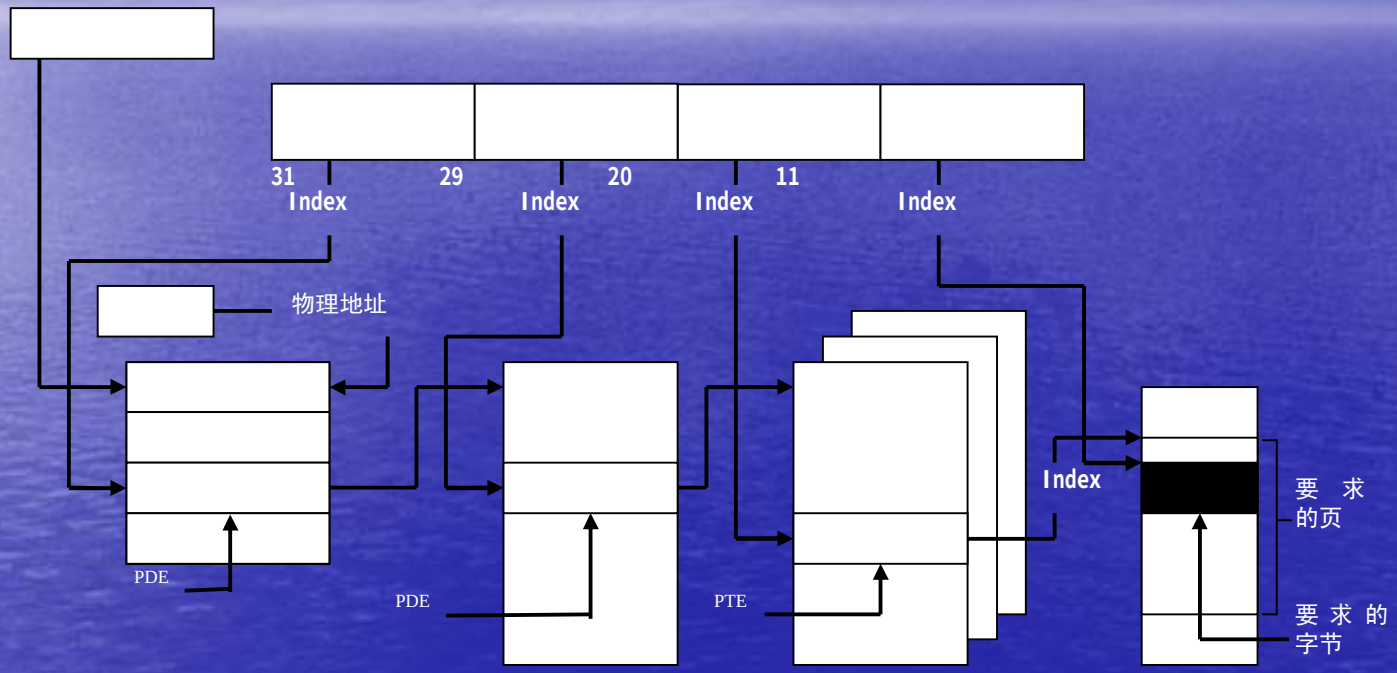
- 所有系统范围内核心态组件使用的数据结构和内存缓冲池只能在核心态下访问。
- 每个进程有一个独立、私有的地址空间，禁止其它进程的线程访问。
- 支持的处理机还提供了一些硬件内存保护措施（如读/写，只读等）。
- 共享内存区域对象具有标准的Windows2000/XP存取控制表（ACLs）

写时复制

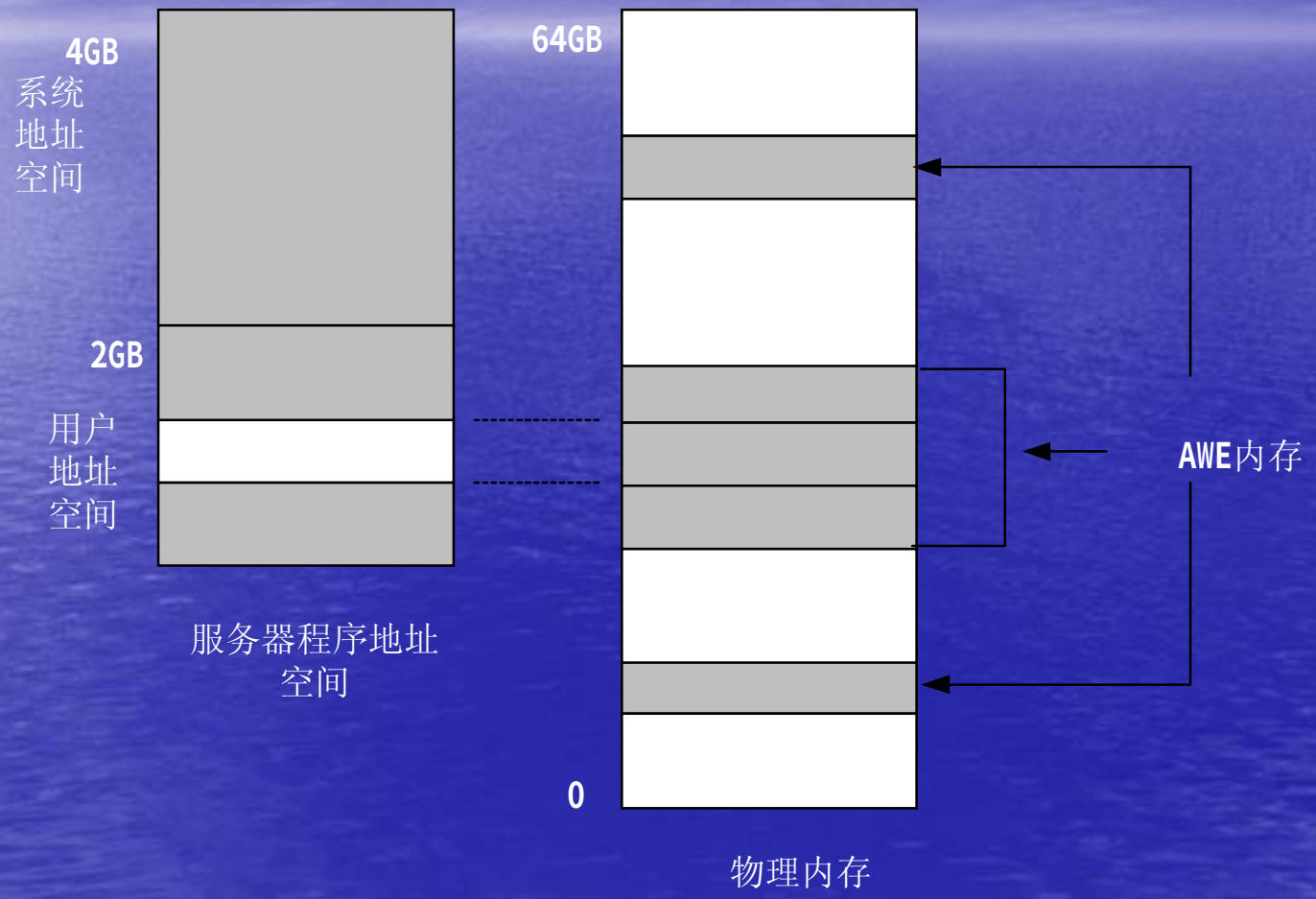

原始数据
进程地址空间
页1
页2
页3
物理内存
原始数据
进程地址空间


原始数据
进程地址空间
页1
页2
页3
页2的拷贝
物理内存
修改的数据
进程地址空间

物理地址扩展


31
29
20
11
Index
Index
Index
Index
物理地址
PDE
PDE
PTE
Index
要 求
的页
要 求 的
字节
页目录指针
（每进程一个，4项）
页目录
（每进程最多四张，每张512项，
8字节宽）
页表
（每张512项，
8字节宽）
物理地址空间

地址窗口扩充



外存管理

- Windows 2000/XP存储的演变
- 分区（Partitioning)
- 驱动程序(Drivers)
- 多重分区管理(Multipartition Volume Management)
- 卷名字空间(The Volume Namespace)

存储的演变

- 让MS—DOS在一个物理盘上采用多个分区，也就是逻辑盘
- Windows NT借鉴了MS—DOS的分区机制，扩展了MS—DOS分区的基本概念，支持企业级操作系统所需的一些存储管理的特征：跨磁盘管理（disk spanning）和容错（fault tolerance）

早期磁盘管理的缺点

- 对大多数磁盘设置的改变需要重启操作系统才能生效
- NT的注册表中为MS—DOS方式的分区保存了多分区磁盘的配置信息
- 每个卷有一个唯一的从A到Z的驱动器名

- 盘一种物理存储设备。
- 扇区可寻址的大小固定的块。
- 分区是盘上连续扇区的集合。
- 简单卷代表文件系统驱动程序作为一个独立单元管理来自一个分区的所有扇区。
- 多分区卷它代表文件系统驱动程序作为一个独立单元管理来自多个分区的所有扇区。多分区卷提供简单卷所不支持的性能、可靠性和大小等特性。

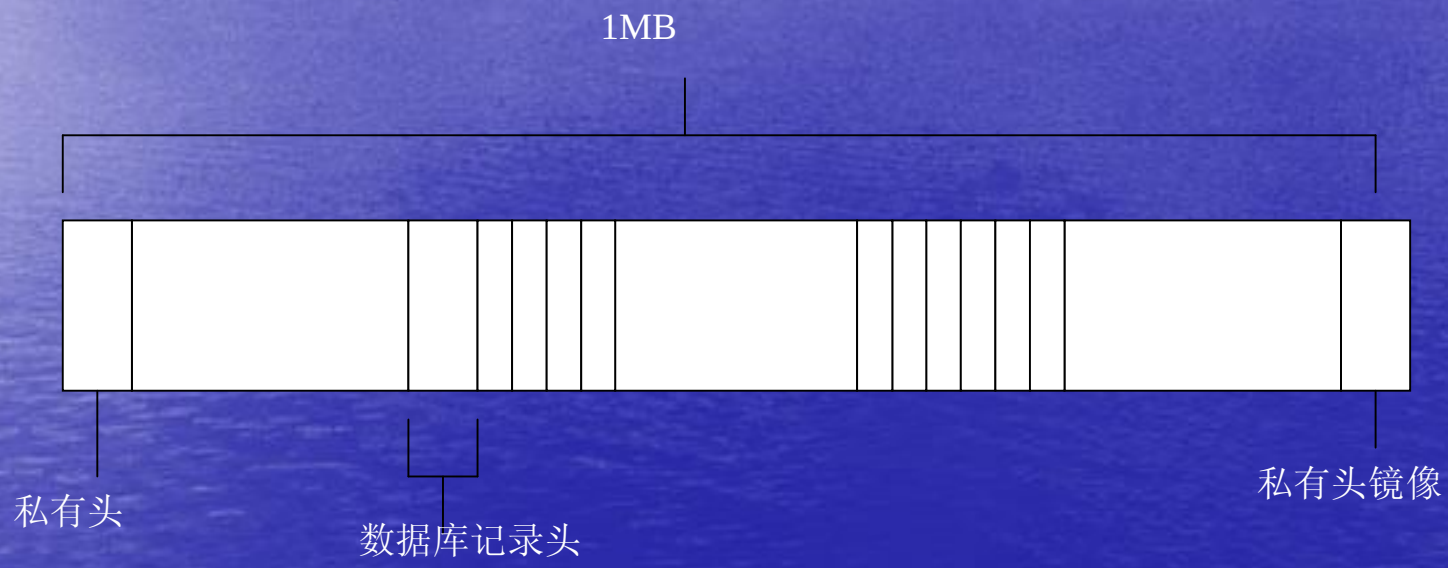
分区

- 基本分区
- 动态分区
  - 逻辑磁盘管理子系统(LDM)负责

LDM

- LDM的数据库存在于每个动态盘最后的1MB保留空间中。
- LDM实现了一个MS DOS的分区表，这是为了继承一些在Windows2000/XP下运行的磁盘管理工具，或是在双引导环境中让其它系统不至于认为动态盘还没有被分区。
- 由于LDM分区在磁盘的MS DOS分区表中并没有体现出来，所以被称为软分区，而MS DOS分区被称为硬分区。

动态盘的内部组织


1MB
私有头
数据库记录头
私有头镜像

数据库结构

- 私有头：GUID，磁盘组的名字（该名字是由Dg0和计算机的名字一起组成，例如SusanDg0，意味着计算机的名字是Susan）和一个指向数据库内容表的指针。为了保证可靠性，LDM在磁盘的最后一个扇区保存了私有头的拷贝。
- 数据库内容表有16个扇区大小，其中包含关于数据库布局的信息。
- 数据库记录区域紧接着内容表，并将内容表后第一个扇区作为数据库记录头。这个扇区中存储了数据库记录区的信息，包括其所包含的记录个数，数据库相关的磁盘组的名字和GUID，以及LDM用于创建下一项的序列号。

- 数据库中的每一项可以是如下四种类型之一：分区，磁盘，组件，卷。LDM把每一项与内部对象的标识符联系到一起。在最低的级别，分区项描述软分区，它是在一个盘上的连续区域。存储在分区项中的标识符把这个项与一个组件和一个磁盘项联系起来。磁盘项代表一个磁盘组中的动态盘，包括磁盘的GUID。组件项像一条链子把一个或多个分区项和与分区相连的卷项联系起来。卷项存放这个卷的GUID，卷的大小和状态，驱动器的名字。比一个数据库记录大的磁盘项占用多个记录的空间，分区项、组件项和卷项很少占用多个记录的空间。

- LDM需要三个项来描述一个简单卷：分区项、组件项和卷项。分区项描述系统分配给某个卷的磁盘上的一个区域，组件项把一个分区项和一个卷项联系起来，卷项中包含Windows 2000/XP内部用来识别卷的GUID。多分区卷需要的项数多于三个。例如，一个条带卷包括最少两个分区项，一个组件项和一个卷项。唯一一种含有一个以上组件项的卷的类型是：镜像卷。镜像卷含有两个组件项，每个只表示这个镜像的一半。LDM为每个镜像卷使用两个组件项的目的是：当一个镜像破坏时LDM能够在组件一级将他们分割开来，并创建两个各含有一个组件项的卷。因为简单卷需要三个项，而1MB数据库空间大约可以容纳8000个项，所以在Windows 2000/XP中可以创建的卷数目的有效上界大约是2500个。

- LDM数据库的最后部分是事务处理日志区，它包含的几个扇区在数据库信息改变时用来存储备份信息。这样确保在系统崩溃或断电时，LDM能够利用日志把系统恢复到一个正确的状态。

驱动程序

- 系统卷中引导扇区中的代码负责执行Ntldr。
- Ntldr从系统卷中读取Boot.ini文件，把计算机的引导选项显示给用户。Boot.ini指定分区名为mult(0)disk(0)rdisk(0)partition(1)的形式。
- Ntldr把Boot.ini中用户指定的项转换为正确的引导分区，然后将Windows 2000/XP系统文件（从注册表、Ntoskrnl.exe、引导驱动程序开始）装入内存，继续引导过程。

- Windows 2000/XP的存储驱动程序
  - 类：实现所有存储设备共同的功能
  - 端口：基于某种特定总线设备的共同功能，如SCSI、IDE
  - 小端口：OEM提供

- 磁盘的类驱动程序使用I/O管理器的IoReadPartitionTable函数识别表示分区的设备对象
- 设备名
  - \Device\Harddisk0\DP(1)0x7e000-0x7ff50c00+2

- Windows 2000/XP保存了两个不同的名字空间子目录供Win32使用，其中之一是：\??子目录（另一个是\BaseNamedObjects 子目录）。
- 在\??子目录中，Windows 2000/XP创建了一些与Win32程序交互的硬件对象，包括串口和并口，还有磁盘。

- 由于磁盘对象实际上存在于其它的子目录中，所以Windows 2000/XP使用符号链接，把在\??子目录下的名字与在名字空间其它地方的对象联系起来。I/O管理器为系统中的每一个物理盘都创建一个\??\PhysicalDriveX的链接，指向\Device\HarddiskX\Partition0（从零开始的数字来替代X）。
- 那些直接访问磁盘扇区的WIN 32应用程序可以调用Win32函数CreateFile，通过指定\\.\PhysicalDriveX（X是一个磁盘的号码）作为参数来打开磁盘。Win32 的应用层先把名字转化为\??\PhysicalDriveX，然后在把名字提交给Windows2000/XP对象管理器。

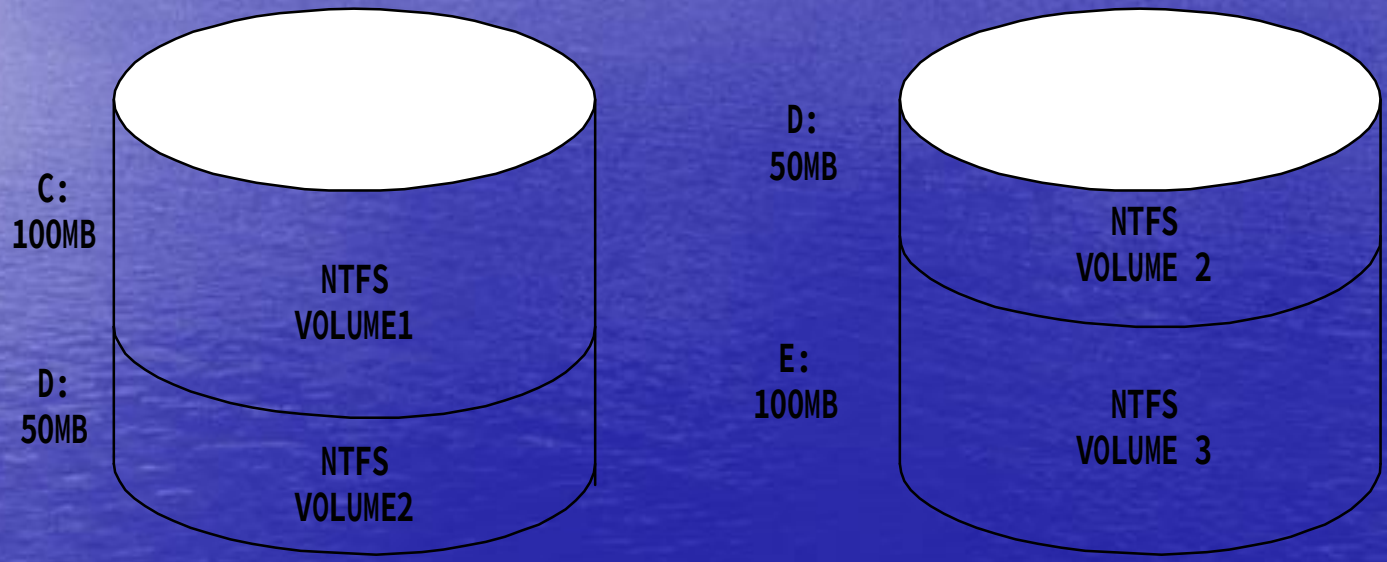
管理工具

- FtDisk和 DMIO负责识别文件系统驱动程序管理的卷，并将I/O直接从卷映射到组成卷的底层分区。
- 对简单卷来说，通过把卷的偏移量加上卷在磁盘中的起始地址，卷管理器可以保证卷的偏移量被转换成盘的偏移量。
- 对于多分区卷这就复杂多了，因为组成卷的分区可以是不邻接的分区，甚至可以在不同的磁盘中。有一些多分区卷使用数据冗余技术，所以它们需要更多的卷到磁盘的转换工作。

多重分区管理

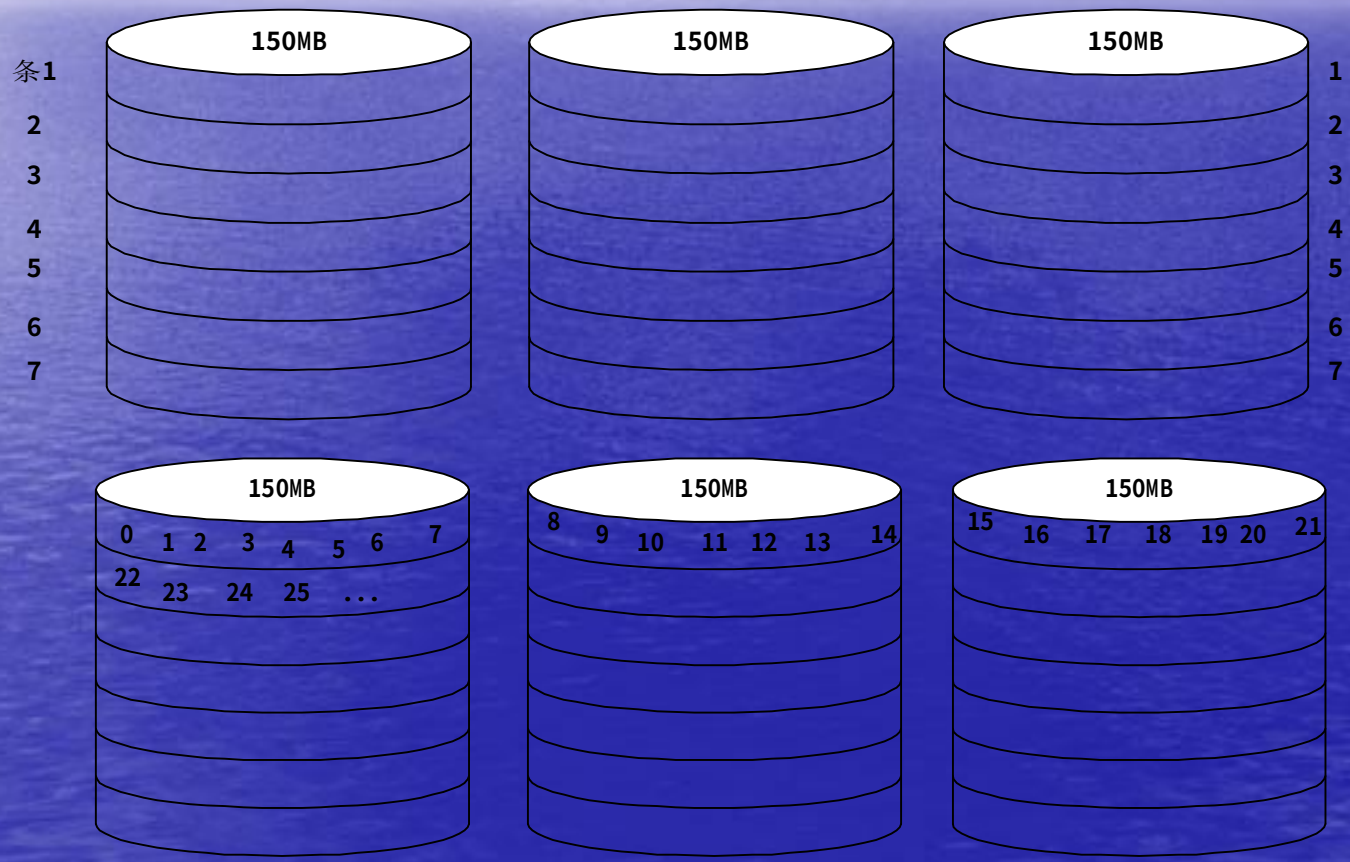
- 跨分区卷(spanned volume)
- 条带卷（striped volume）
- 镜像卷（mirrored volume）
- 廉价冗余磁盘阵列5卷（RAID-5 volume）

跨分区卷


C:
100MB
NTFS
VOLUME1
D:
50MB
NTFS
VOLUME2
D:
50MB
NTFS
VOLUME 2
E:
100MB
NTFS
VOLUME 3

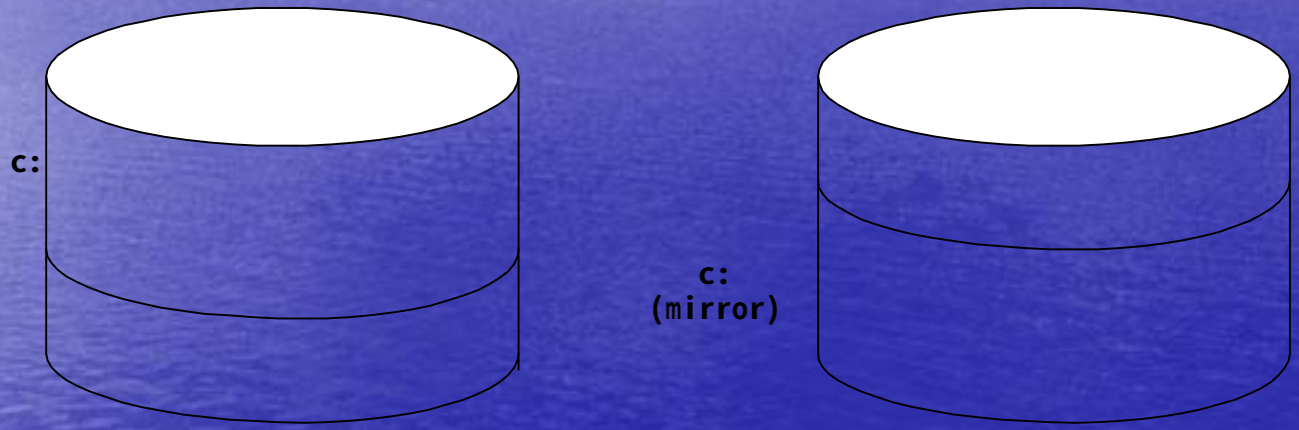
- 一个单独的逻辑卷，最多由在一个或多个磁盘上的32个空闲分区组成。
- 跨分区卷可以用来把小的磁盘空闲区域，或者把两个或更多的小磁盘组成大的卷。
- 卷管理器对Windows 2000/XP的文件系统隐藏了磁盘物理配置信息。

条带卷（RAID-0卷）


150MB
150MB
150MB
条1
2
3
4
5
6
7
1
2
3
4
5
6
7
150MB
0 1 2 3 4 5 6 7
22 23 24 25 ...
150MB
8 9 10 11 12 13 14
150MB
15 16 17 18 19 20 21

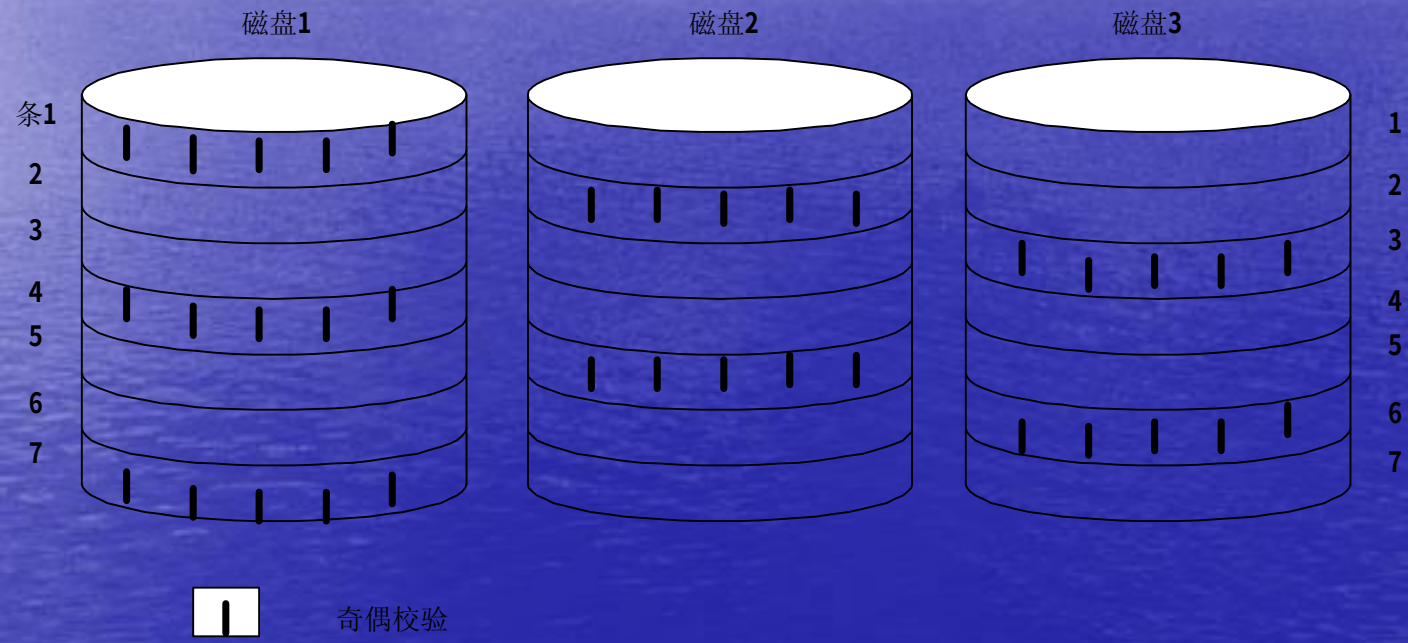
- 一系列分区组成的单独的逻辑卷，最多有32个分区并且每个盘一个分区。
- 条带卷中的一个分区不需要占据整个磁盘，唯一的限制是每个盘上的分区大小相同
- 数据能够被平均分配到每个磁盘上

镜像卷



- 一个磁盘上分区的内容被复制另一个磁盘与它等大小的分区中。镜像卷有时也被称为RAID-1。
- 镜像卷能够可以在主分区和镜像分区之间平衡I/O操作。两个读操作可以同时进行，所以理论上只用一半时间就可以完成。当修改一个文件时，必须写入镜像卷的两个分区，但是磁盘写操作可以异步进行，所以用户态程序的性能一般不会被这种额外的磁盘更新所影响。
- 镜像卷是唯一一种支持系统卷和引导卷的多分区卷。

廉价冗余磁盘阵列5卷



卷名字空间

- 安装管理器
- 安装点
- 卷安装

安装管理器

- 安装管理器（Mountmgr.sys）是Windows 2000/XP中新驱动程序，为在Window 2000/XP安装后创建的动态磁盘卷和基本磁盘卷分配驱动器名。
- 卷管理器创建卷时都将通知它。当接到通知时，确定新的卷GUID或者磁盘标记；
- 安装管理器使用卷GUID（或者标识）在内部数据库中进行查询
- 安装管理者使用第一个未分配的驱动器名，为这次分配创建一个符号链接（例如，\??\D:）

安装点

- 实现安装点的技术是再解析点(Reparse Point)技术。
- C:\Project CurrentProject\Description.txt
- C:\Projects\CurrentProject\Description.txt

卷安装

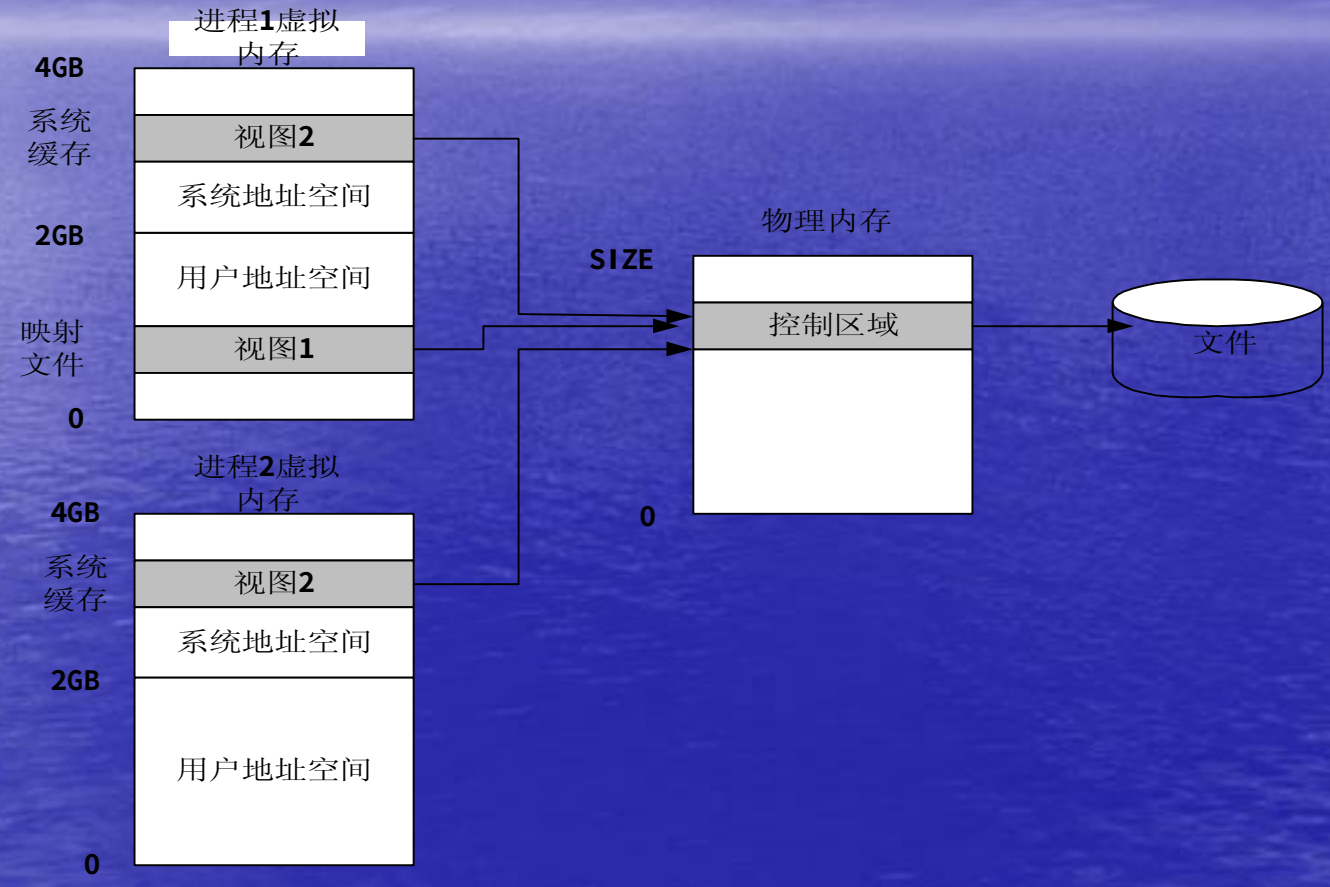
- 每个文件系统驱动程序在初始化时都向I/O管理器注册
- 每个设备对象包含一个卷参数块VPB，但是I/O管理器认为只有卷设备对象的VPB是有意义的
- 安装请求

- 安装管理器把D:分配给系统中的第二个卷
- 它产生符号连接\??\D:指向设备对象\Device\HarddiskVolume2。
- 一个WIN 32应用程序试图打开D:上的文件\Temp\Test.txt时，它将会指定路径D:\Temp\Test.txt。

- WIN 32子系统在调用NtCreateFile之前将路径转化为\??\D: \Temp\Test.txt。
- I/O管理器将检查\Device\HarddiskVolume2的VPB是否引用一个文件系统。

高速缓存

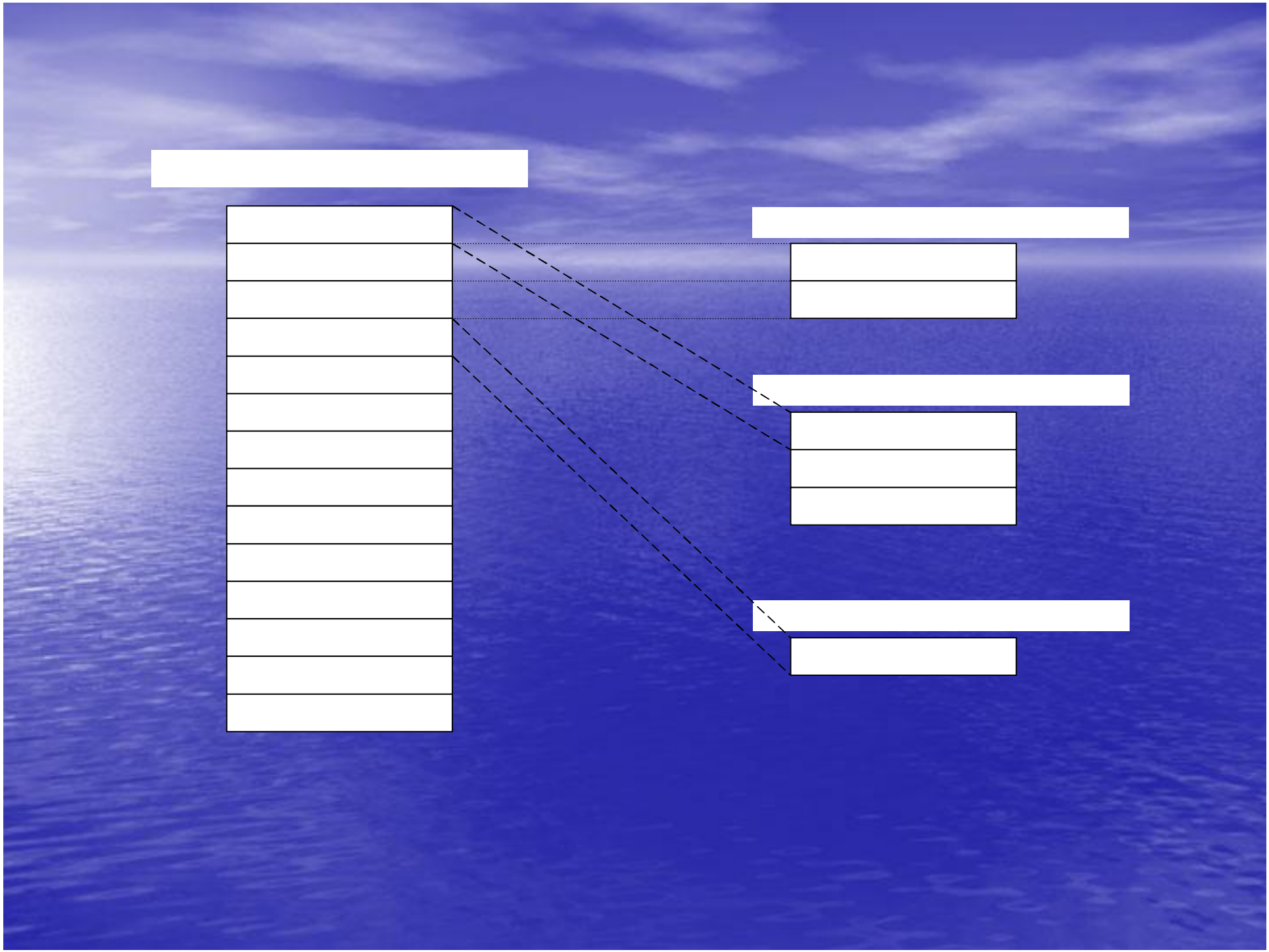
- 单一集中式系统高速缓存
  - 任何数据都能被高速缓存，无论它是用户数据流（文件内容和在这个文件上正在进行读和写的活动）或是文件系统的元数据（metadata）（例如目录和文件头）
- 与内存管理器结合
  - 因为它采用将文件视图映射到系统虚拟空间的方法访问数据
- 高速缓存的一致性


进程1虚拟内存
4GB
系统缓存
视图2
系统地址空间
2GB
用户地址空间
映射文件
视图1
0
进程2虚拟内存
4GB
系统缓存
视图2
系统地址空间
2GB
用户地址空间
0
SIZE
物理内存
控制区域
0
文件

- 虚拟块缓存
  - Windows 2000/XP高速缓速管理器用一种虚拟块缓存方式，管理器对缓存中文件的某些部分进行追踪。通过内存管理器的特殊系统高速缓存例程将256-KB大小的文件视图映射到系统虚拟地址空间，高速缓存管理器能够管理文件的这些部分。这种方式有以下几个主要特点：
  - 它使智能的文件预读成为可能。
  - 它允许I/O系统绕开文件系统访问已经在缓存中的数据（快速I/O）。

- 基于流的缓存
- 可恢复的文件系统支持
  - 文件系统写一个日志文件记录，记录将要进行的卷修改操作。
  - 文件系统调用高速缓存管理器将日志文件记录刷新到磁盘上。
  - 文件系统把卷修改内容写入高速缓存，即修改文件系统在高速缓存的元数据。
  - 高速缓存管理器将被更改的元数据刷新到磁盘上，更新卷结构。


80000000
系统代码(Ntoskrnl,HAL) 和一些系统中初始的未分页缓冲池
A0000000
系统映射视图（例如，Win32k.sys)或者会话空间
A4000000
附加的系统PTE（高速缓存可以扩展到这）
C0000000
进程的页表和页目录
C0400000
超空间和进程工作集列表
C0800000
没有使用,不可访问
C0C00000
系统工作集列表
C1000000
系统高速缓存
E1000000
分页缓冲池
EB000000(min)
系统PTE
未分页缓冲池扩充
FFBE0000
故障转储信息
FFC00000
HAL使用



高速缓存的大小

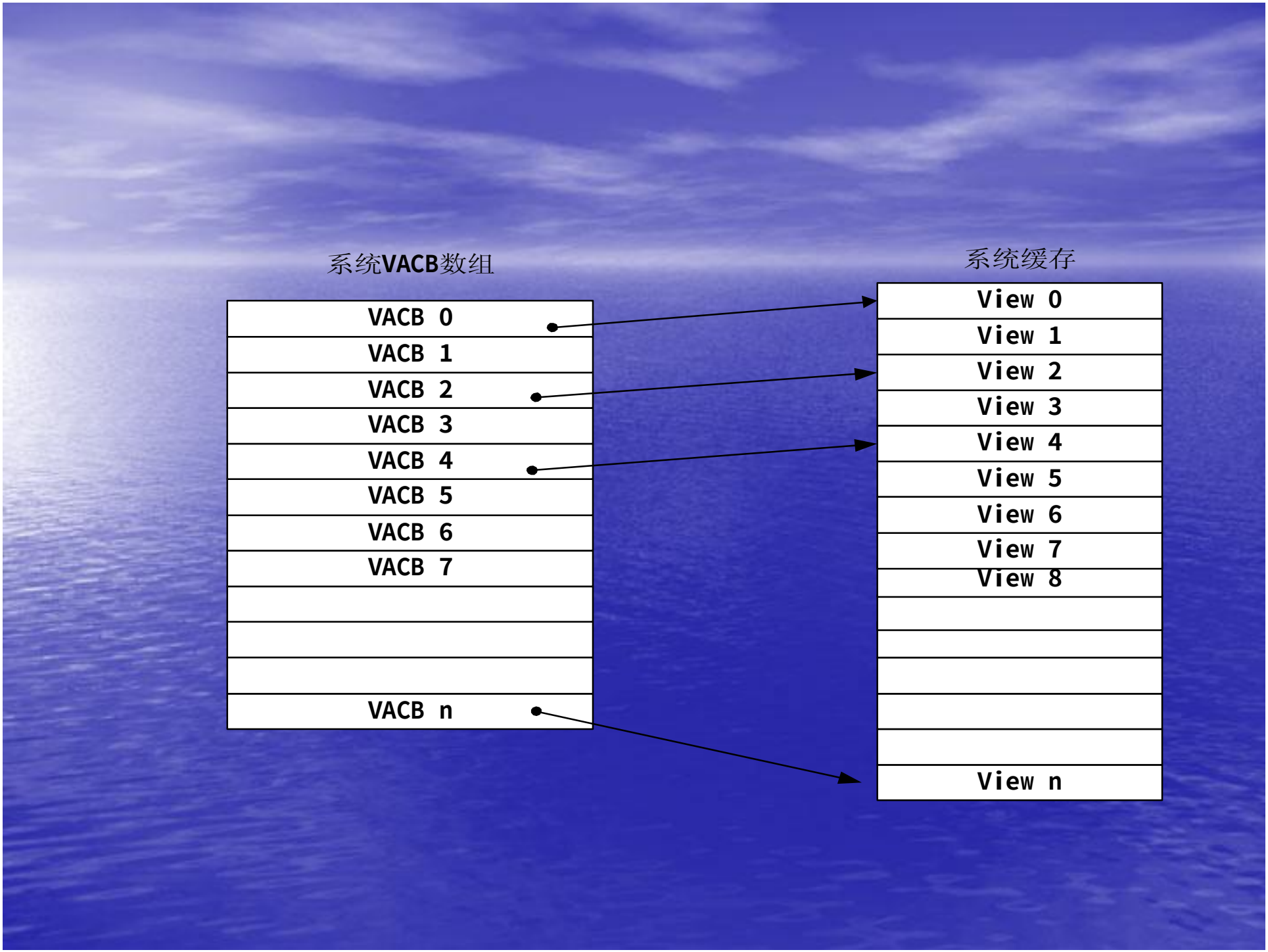
- 缓存区的虚拟大小
- 缓存的物理大小

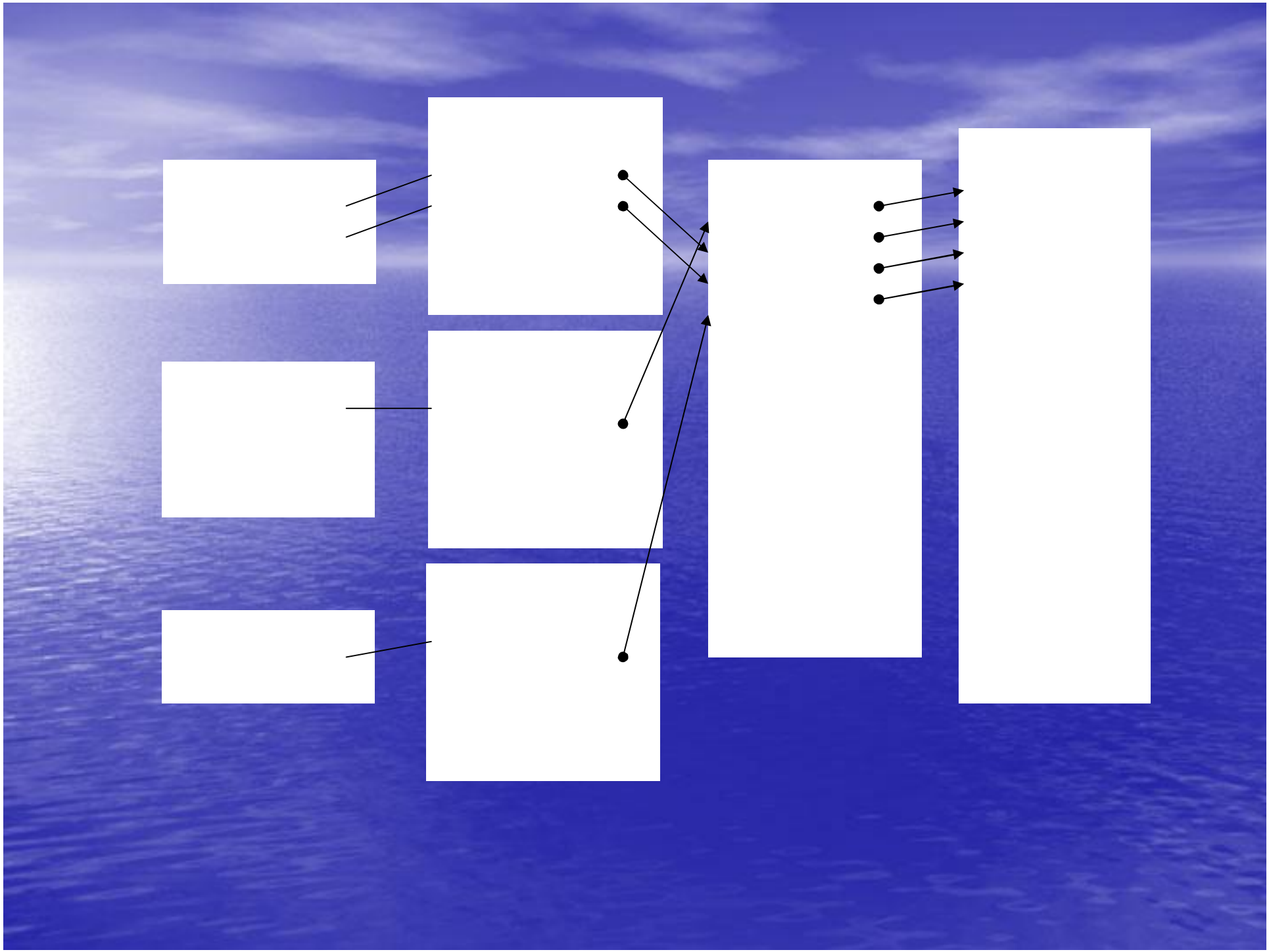
高速缓存的数据结构

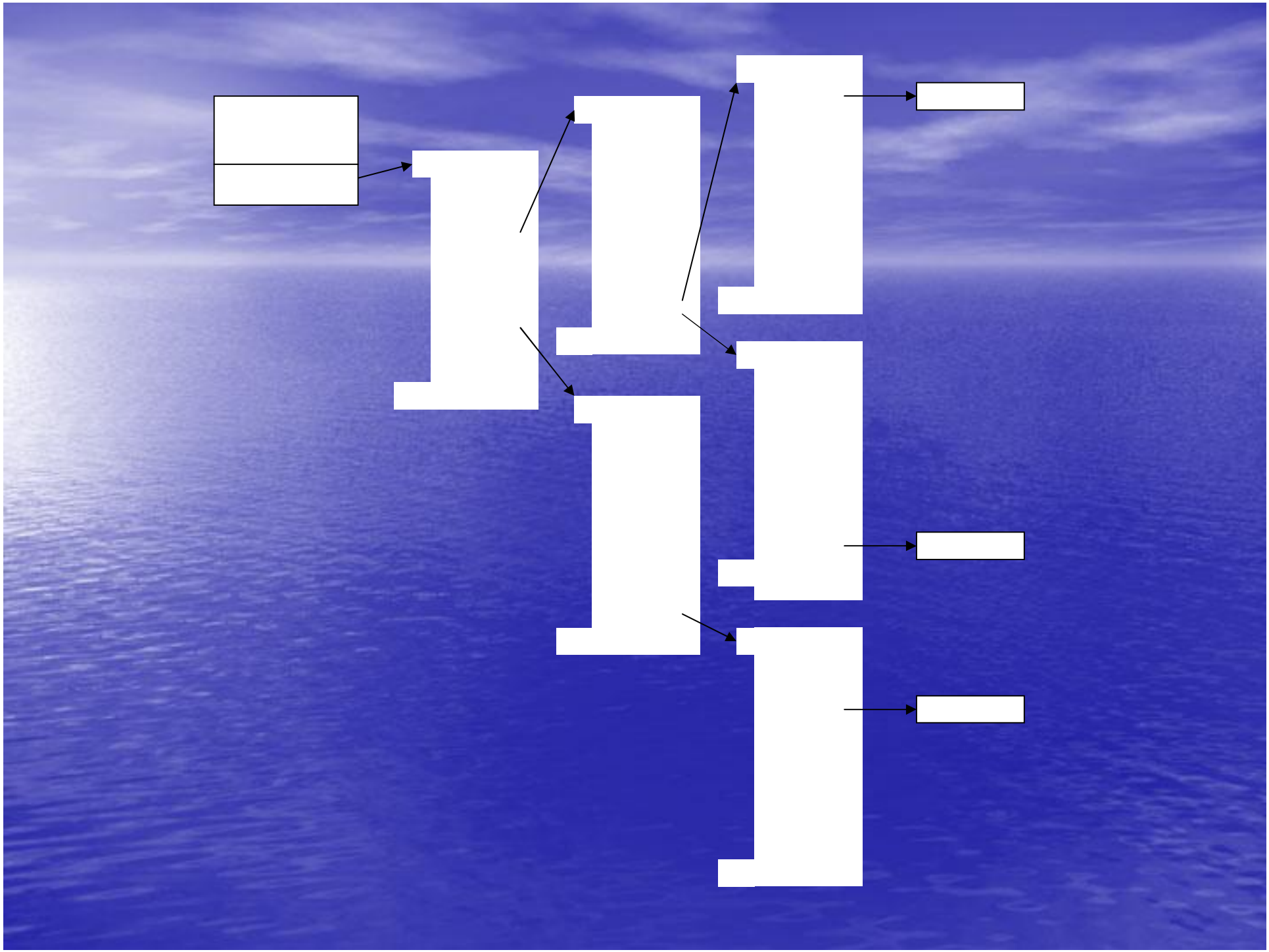
- 在系统高速缓存的每个256 KB的槽由一个VACB描述。
- 每个打开的被缓存文件有一个专用的缓存映射，它包含了用于控制文件预读的信息。
- 每个被缓存的文件有一个单独的共享缓存映射结构，它指向系统缓存中包含此文件映射视图的槽。

虚拟地址控制块（VACB）

- 系统高速缓存中数据的虚拟地址
- 指向共享高速缓存映射的指针
- 文件偏移
- 活动计数


系统VACB数组
VACB 0
VACB 1
VACB 2
VACB 3
VACB 4
VACB 5
VACB 6
VACB 7
VACB n
系统缓存
View 0
View 1
View 2
View 3
View 4
View 5
View 6
View 7
View 8
View n





高速缓存的操作

- 回写缓存和延迟写
  - 写入文件的数据首先被存储在高速缓存页面的内存中，然后再被写入磁盘。因此，写操作允许在短时间内积累，并一次性刷新到磁盘，这可以减少磁盘的I/O次数。
- 屏蔽对文件延迟写
  - 在调用Win32 CreateFile函数时指定FILE_ATTRIBUTE_TEMPORARY标志创建一个临时文件，延迟写器就不会将脏页写回磁盘，除非物理内存严重不足或文件关闭。
- 强制写缓存到磁盘
- 刷新被映射的文件

智能预读

- 虚拟地址预读
  - 将被访问页面相近的几个页一起读到内存中。内存管理器的这种方法唯一缺点是：必须同步进行
- 带历史信息的异步预读
  - 高速缓存管理器在文件的私有缓存映射结构中为正在被访问的文件句柄保存最后两次读请求的历史信息

- 快速I/O(fast I/O)
- 写阻塞

访问缓存数据的方法

- “拷贝读取”方法在系统空间中的高速缓存数据缓冲区和用户空间中的进程数据缓冲区之间拷贝用户数据；
- “映射暂留”方法使用虚拟地址直接读写高速缓存的数据缓冲区。
- “物理内存访问”方法使用物理地址直接读写高速缓存的数据缓冲区。