

第4章 存储体系

Ø 主要包括:

Ø 几种常用的内存管理方法（分区、分页、分段）

Ø 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）

Ø 虚拟存储器的概念（部分装入）

Ø 控制主存和外存之间的数据流动方法

Ø 地址变换技术和内存数据保护与共享技术等

Ø 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

(1) 内存空间的分配与回收:

存储管理要为运行进程的程序和数据分配内存空间,并在不需要时回收它们占据的空间,系统会建立一张“主存空间表”,记录主存空间的分配情况。

(2) 实现地址变换:

存储管理必须配合硬件进行地址变换工作,把用户使用的逻辑地址(又称相对地址或虚拟地址)转换成处理器能访问的绝对地址(又称物理地址或实地址)。

(3) 主存空间的共享与保护:

主存空间的共享是指若干个进程能够共同访问公共程序所占的主存区。存储保护是为了防止多程序在执行中互相干扰并保护区域内的信息不被破坏。

(4) 主存空间的扩充:

当内存容量不足时,操作系统要采取某种措施在不改变实际内存容量的前提下,借助于大容量的外存解决内存不够用的问题。

虚拟存储器

- Ø 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。
- Ø 外存尽管访问速度较慢，但价格便宜，适合于存放大量信息。
- Ø 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:

- Ø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;
- Ø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

- Ø 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。
- Ø 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。
- Ø 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。
- Ø 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

- Ø 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。
- Ø 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。
- Ø 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

- Ø 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）
- Ø 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；
- Ø 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

Ø 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。

Ø 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

- Ø 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。
- Ø 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
- Ø 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。

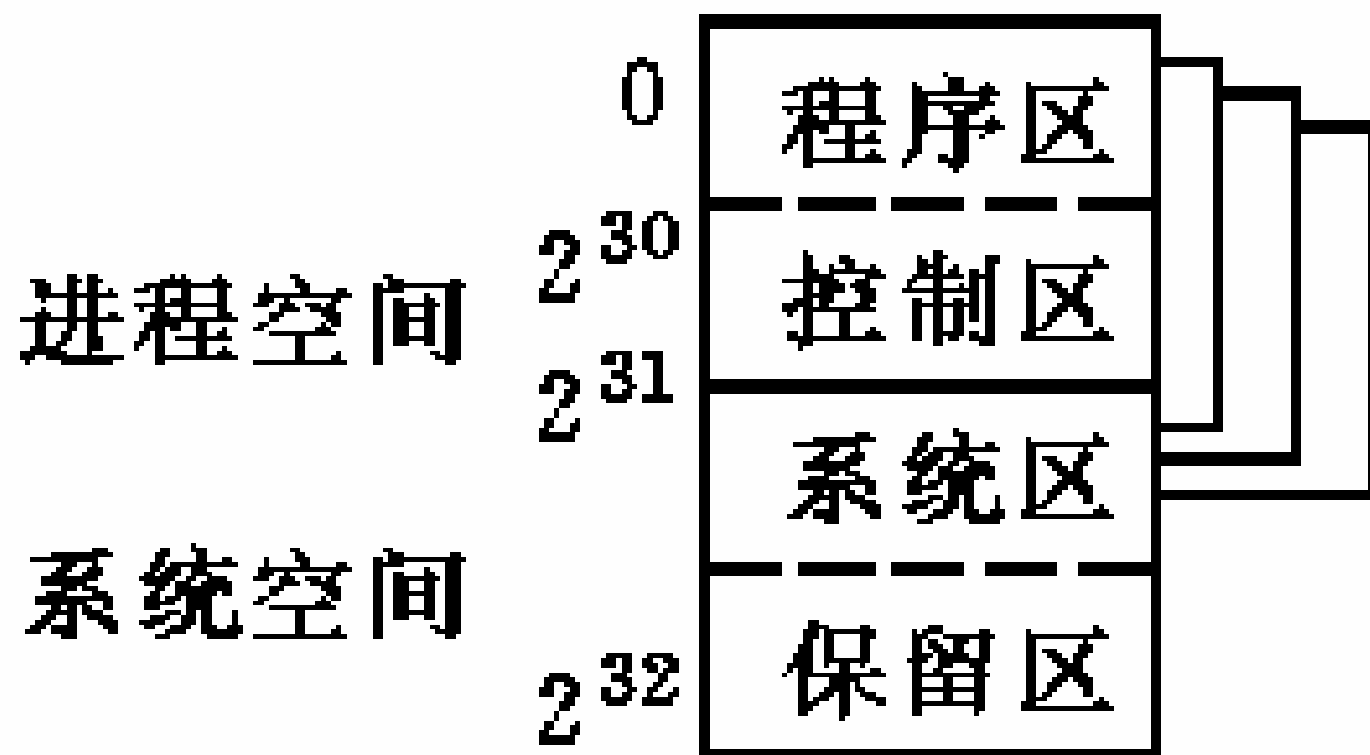


图5.2 虚拟空间的划分

重定位方法

Ø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。

Ø程序在成为进程前的准备工作

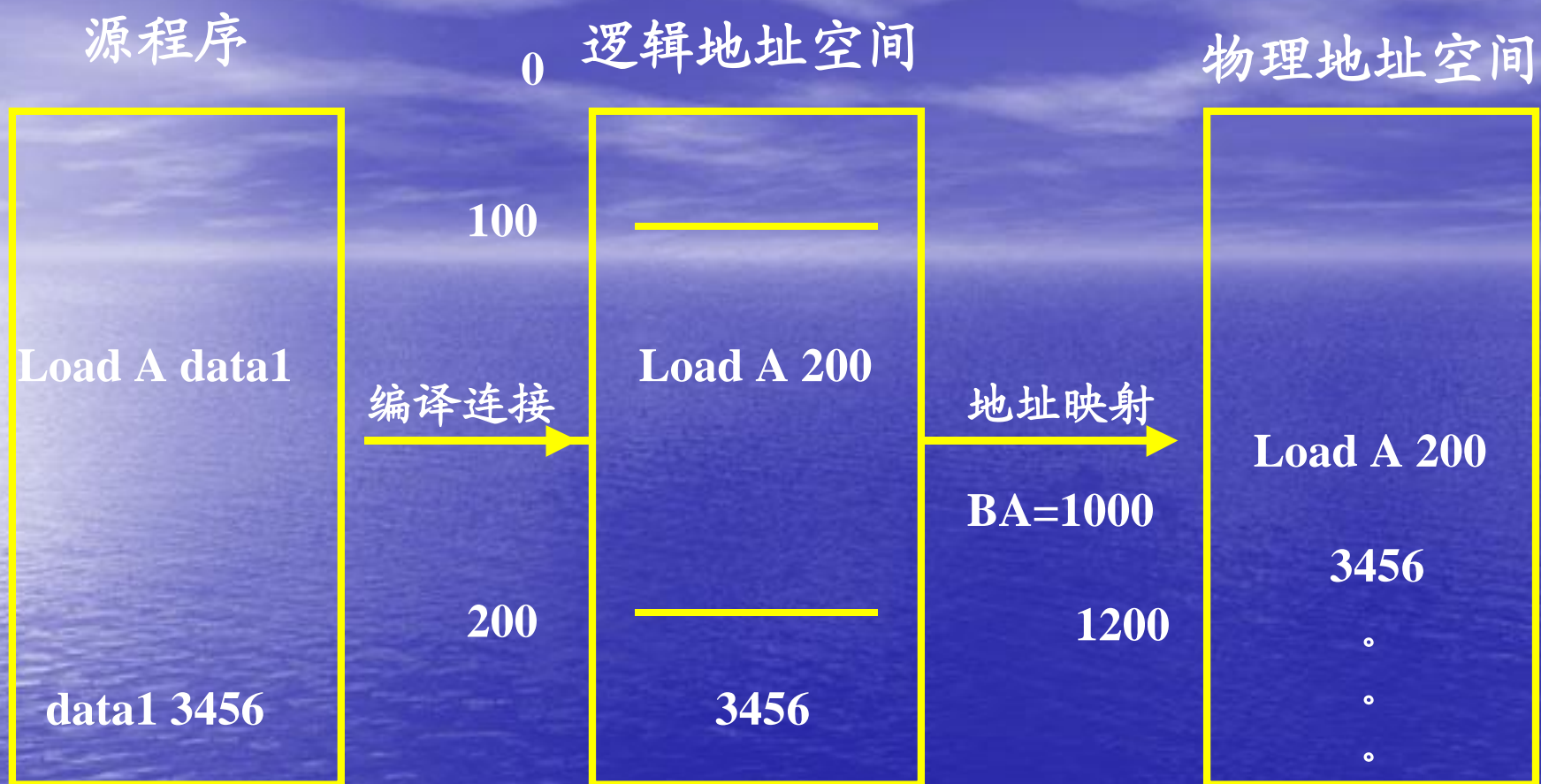
- 编辑：形成源文件(符号地址)
- 编译：形成目标模块(模块内符号地址解析)
- 链接：由多个目标模块或程序库生成可执行文件(模块间符号地址解析)
- 装入：构造PCB，形成进程(使用物理地址)

逻辑地址、物理地址和地址映射

逻辑地址：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。

物理地址：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。



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。

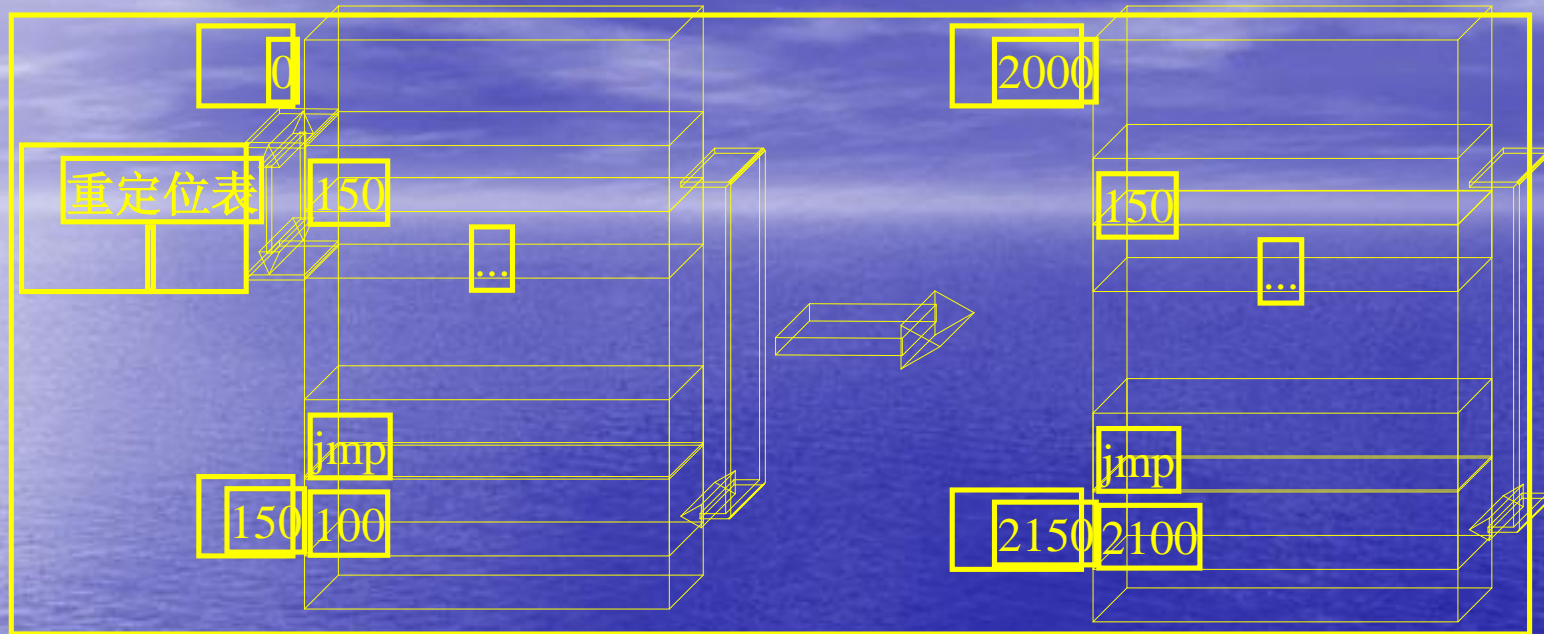
(2)动态地址重定位

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- \rightarrow 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:

- Ø 可以装入的程序道数受限;
- Ø 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。
- Ø 使用静态重定位方法进行地址变换无法实现虚拟存储器。
- Ø 必须占用连续的内存空间和难以做到程序和数据
的共享

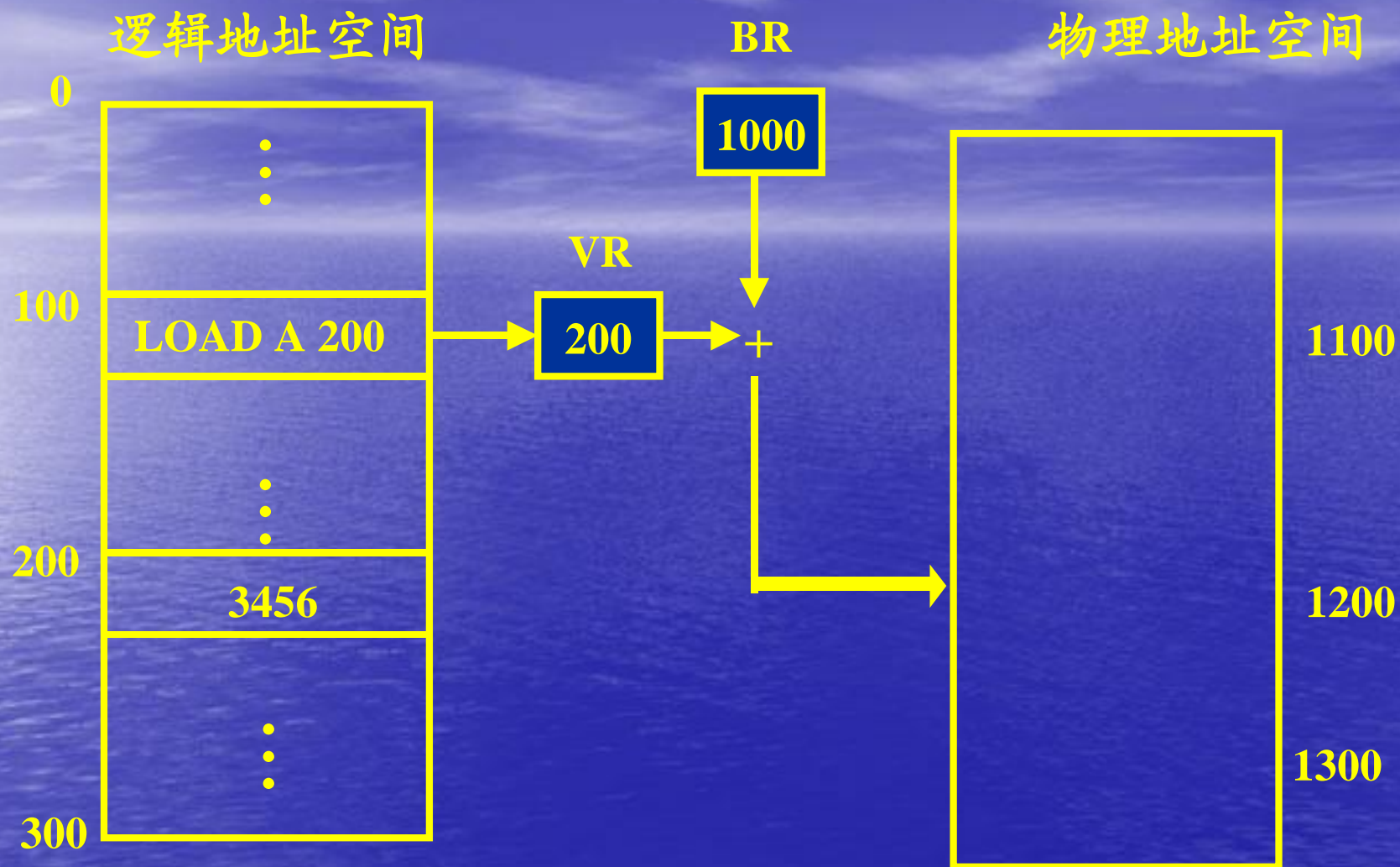
动态地址重定位

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。

- 1 动态重定位依靠硬件地址变换机构完成。
- 2 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： $MA = (BR) + (VR)$
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。



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

- Ø 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。
- Ø 最简单，适用于单用户、单任务的OS。
- Ø 优点：易于管理。
- Ø 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

- Ø 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。
- Ø 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。

Ø 用户程序自己控制内外存之间的数据交换的例子是覆盖。

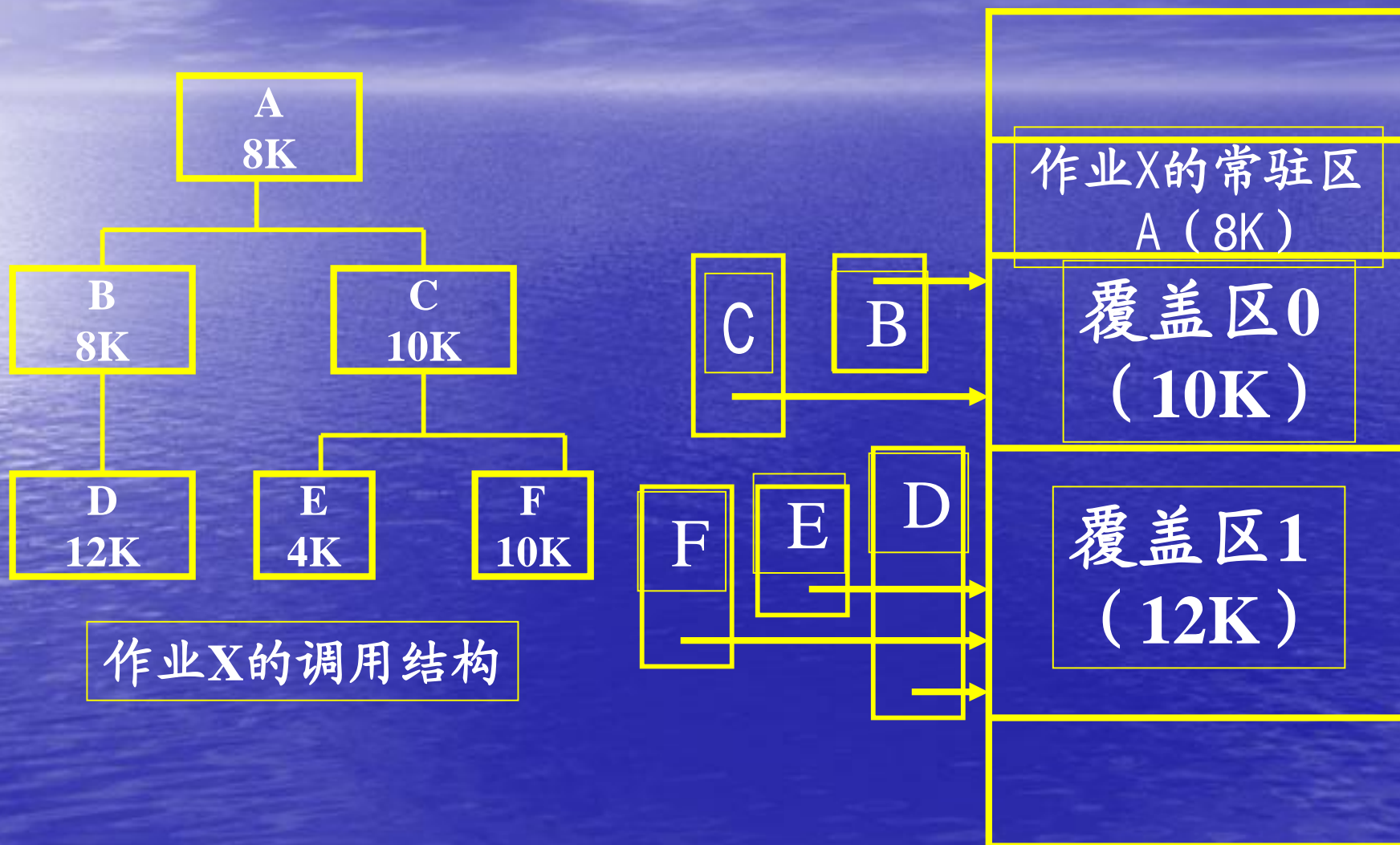
Ø 覆盖技术要求用户清楚地了解程序的结构，并指定各程序段调入内存的先后次序。

Ø 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，然后由操作系统完成自动覆盖

覆盖技术例（图）



Ø 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。

Ø 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

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

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

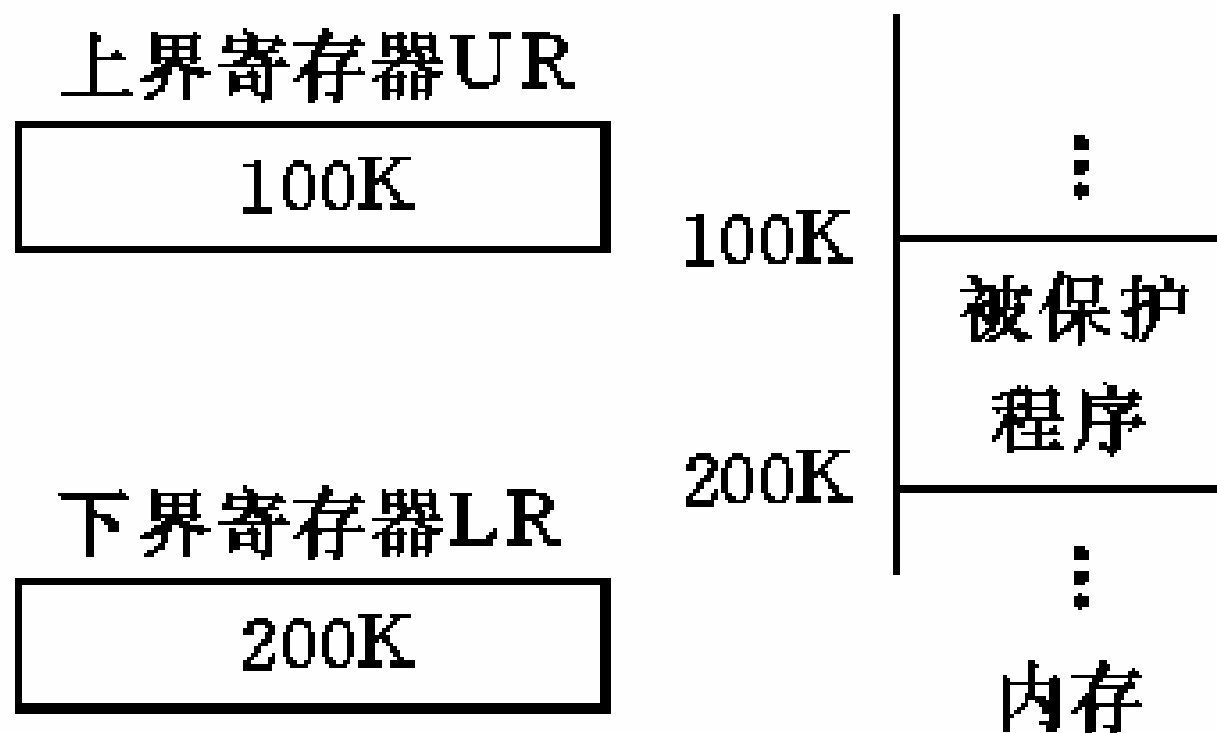
在多道程序设计环境下，内存中的许多用户或系统程序和数据段可供不同的用户进程共享。这种资源共享将会提高内存的利用率。

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。



$100K \leq \text{被访问地址} \leq 200K$

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。

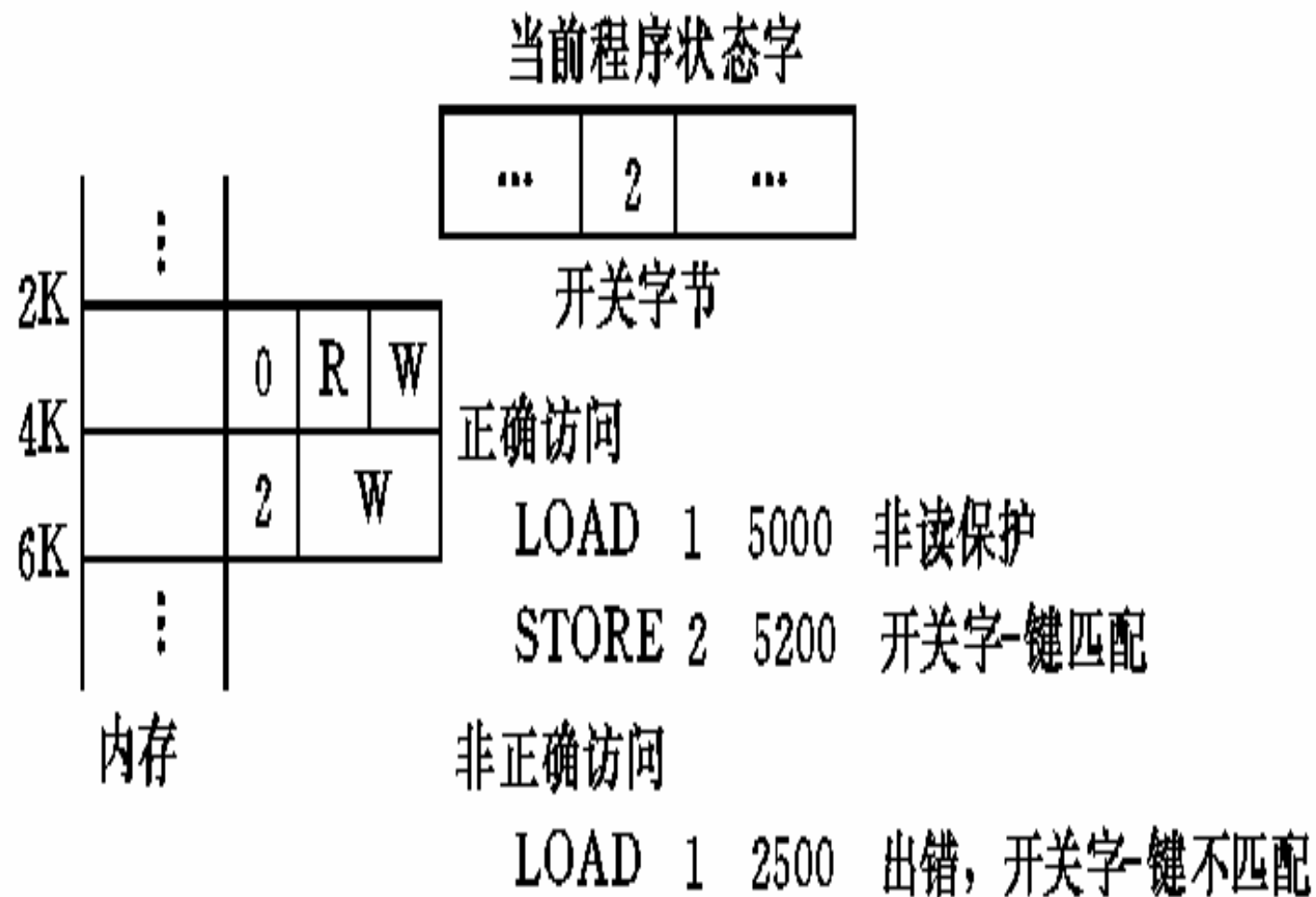


图5.5 保护键保护法

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

Ø 把内存分为一些大小相等或不等的分区，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。

Ø 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。

Ø 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）。

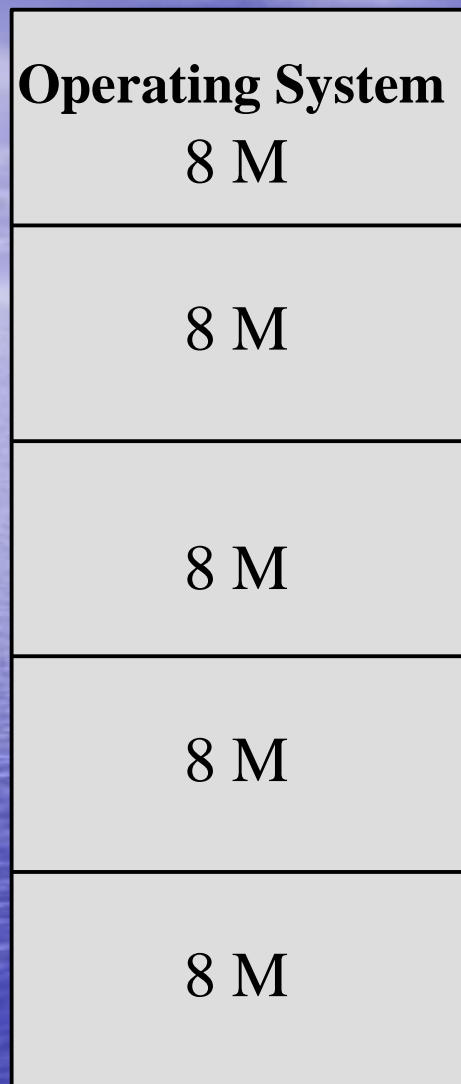
分区管理基本原理

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,以连续存储各进程序的程序和数据,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。

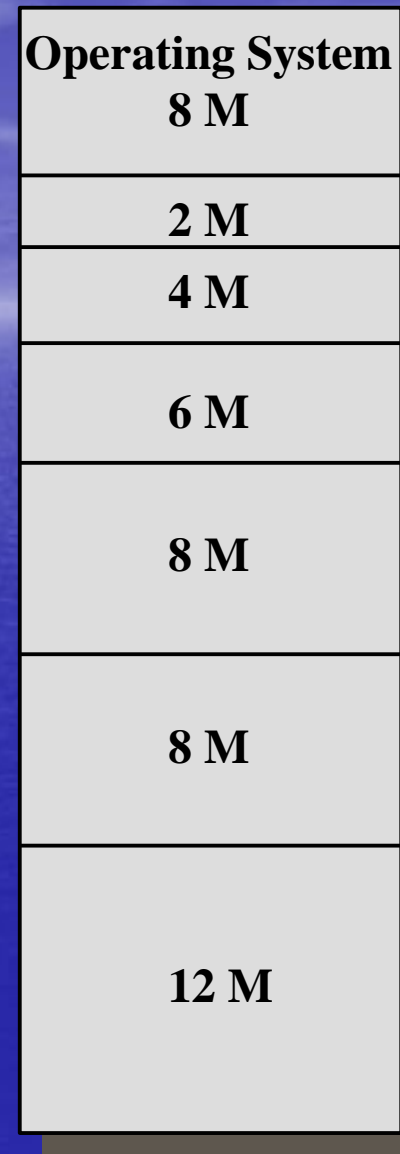
1. 固定分区法□

Ø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。



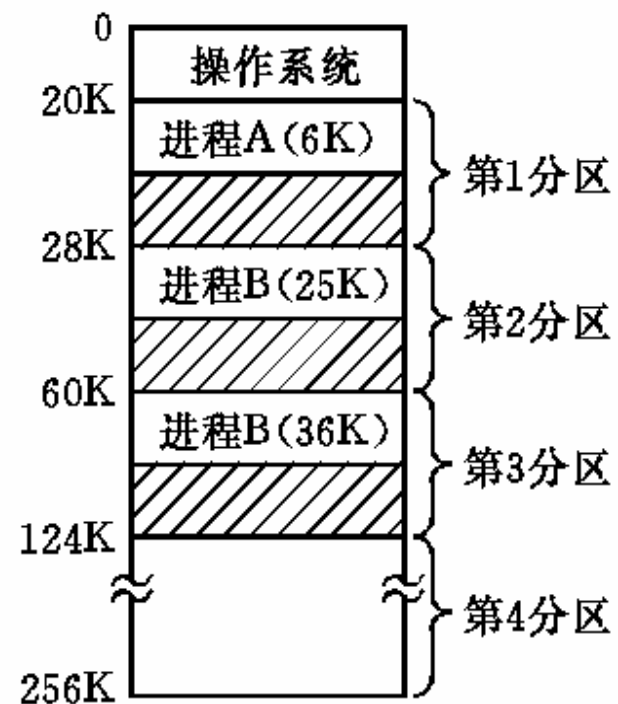
固定分区(大小相同)



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

- Ø 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
- Ø 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。

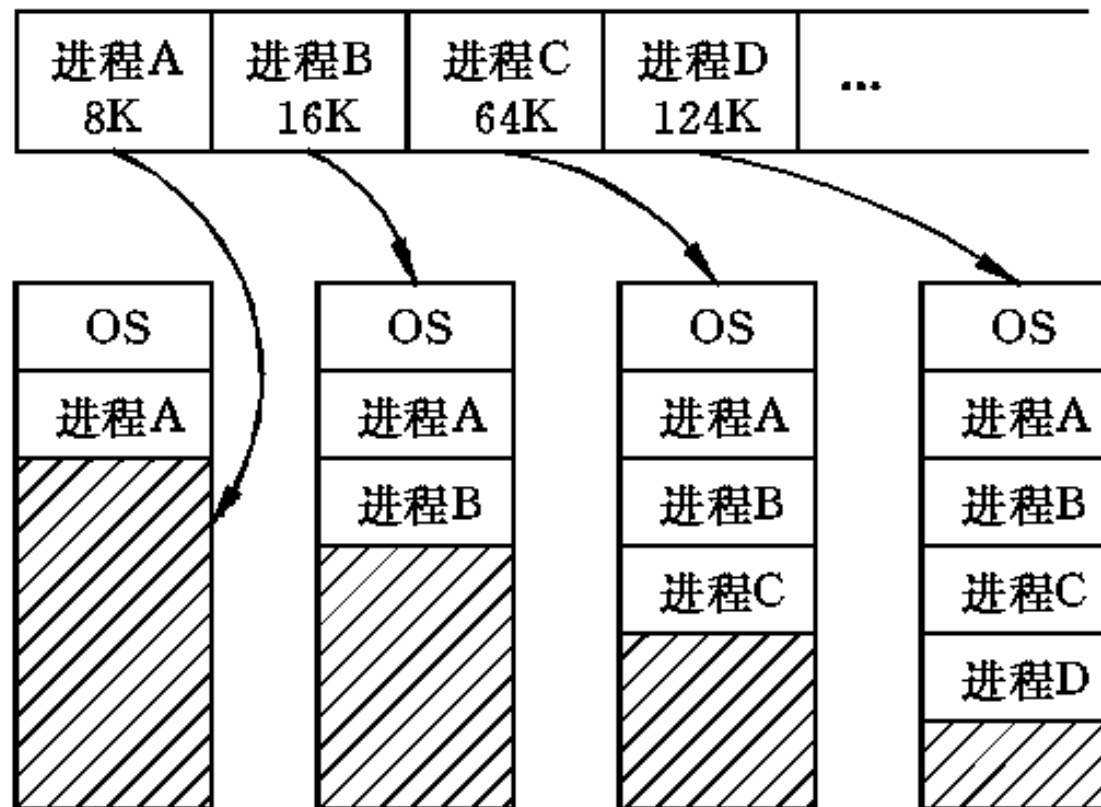


图5.7 内存初始分配情况

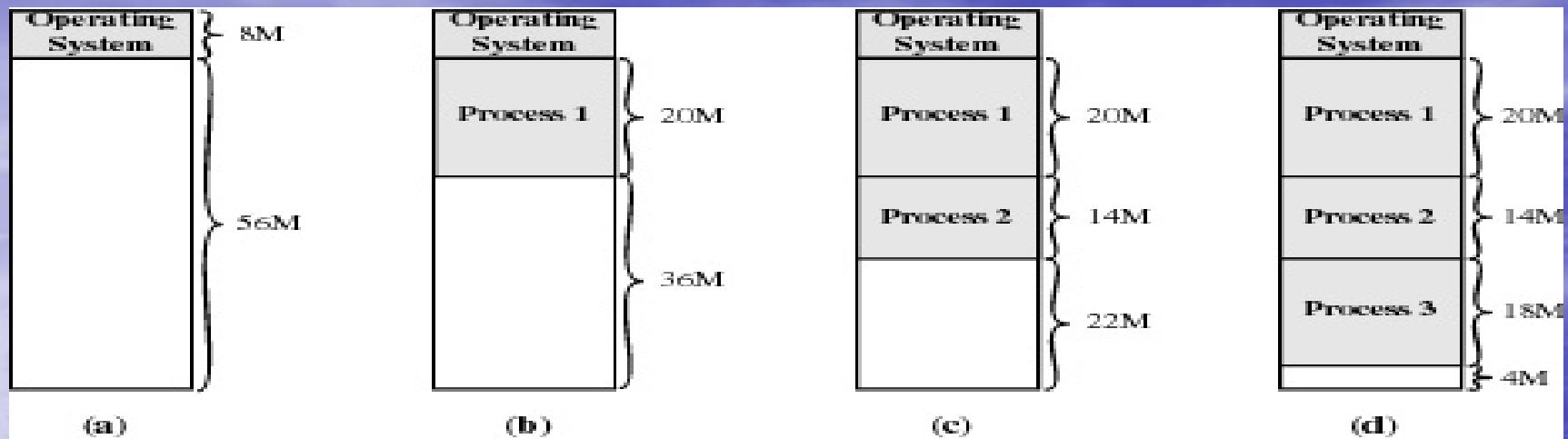
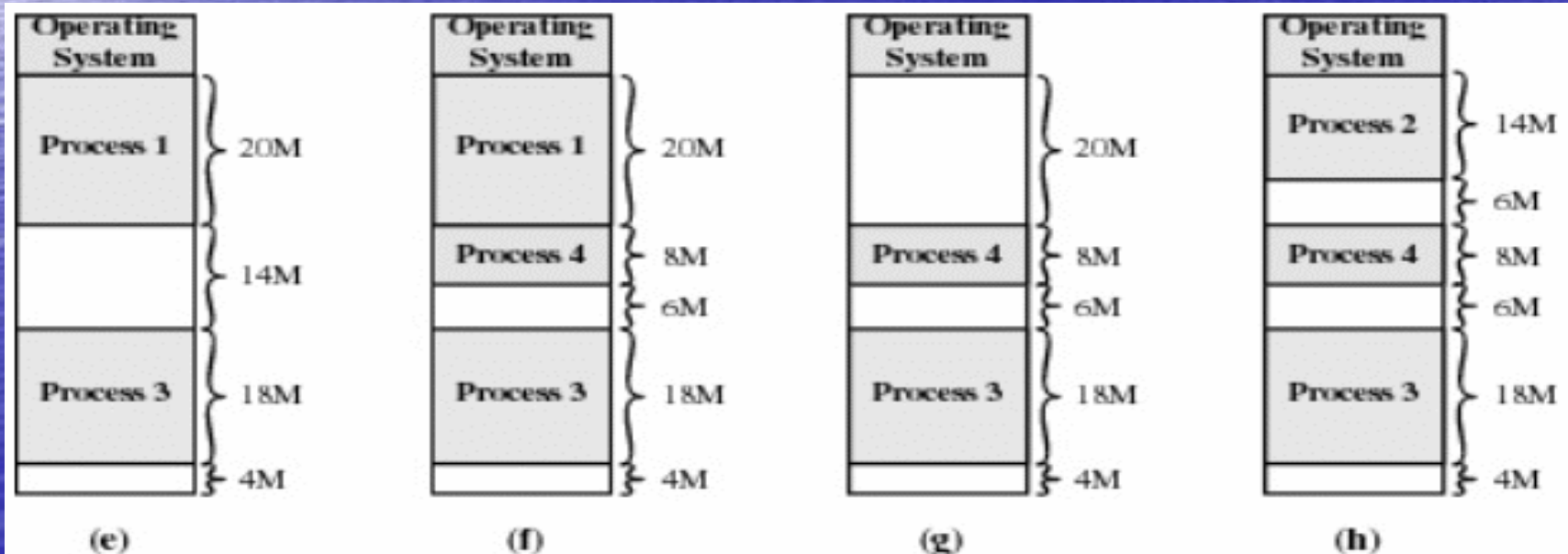


Figure 7.4 The Effect of Dynamic Partitioning



Ø 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）

Ø 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

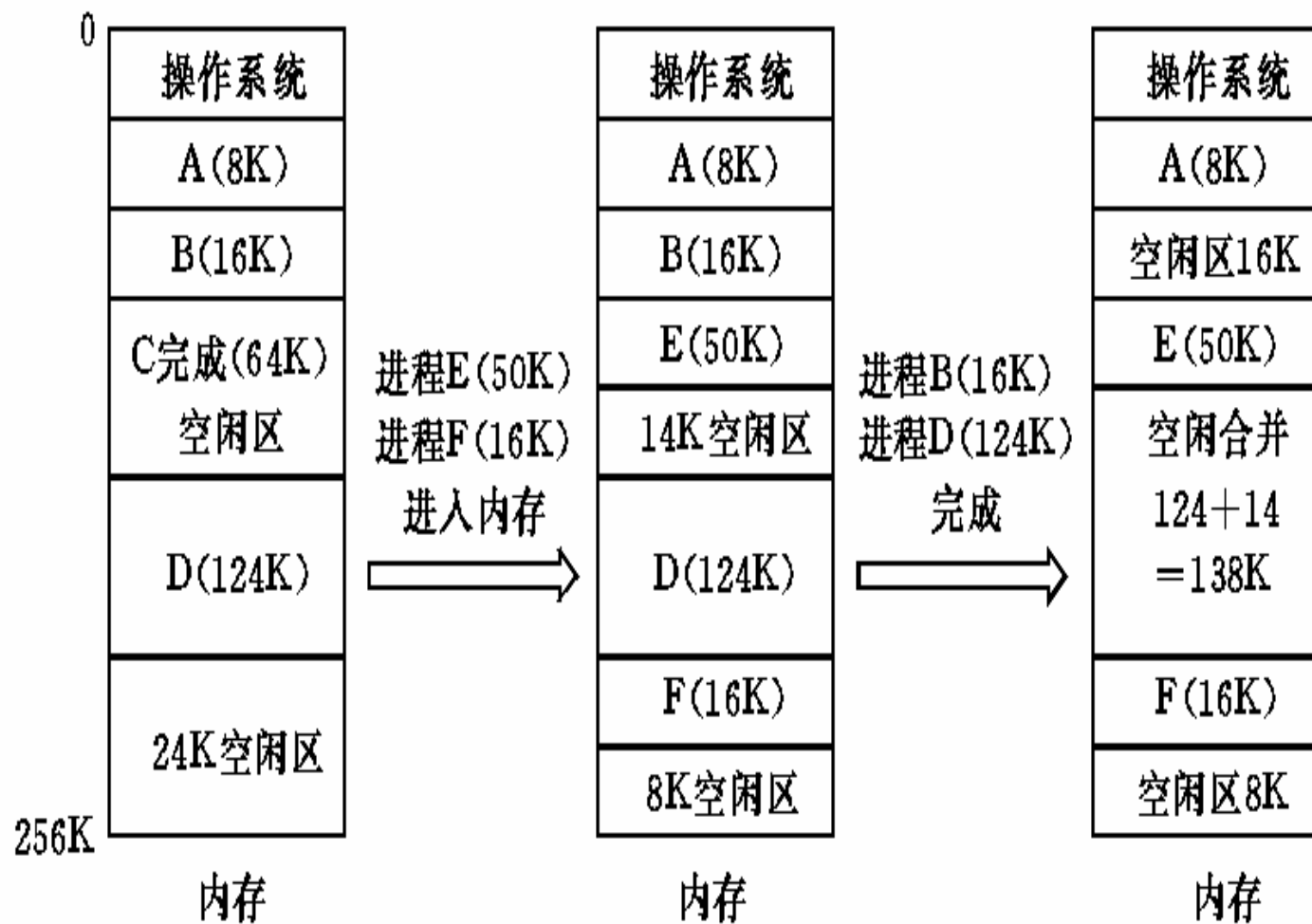
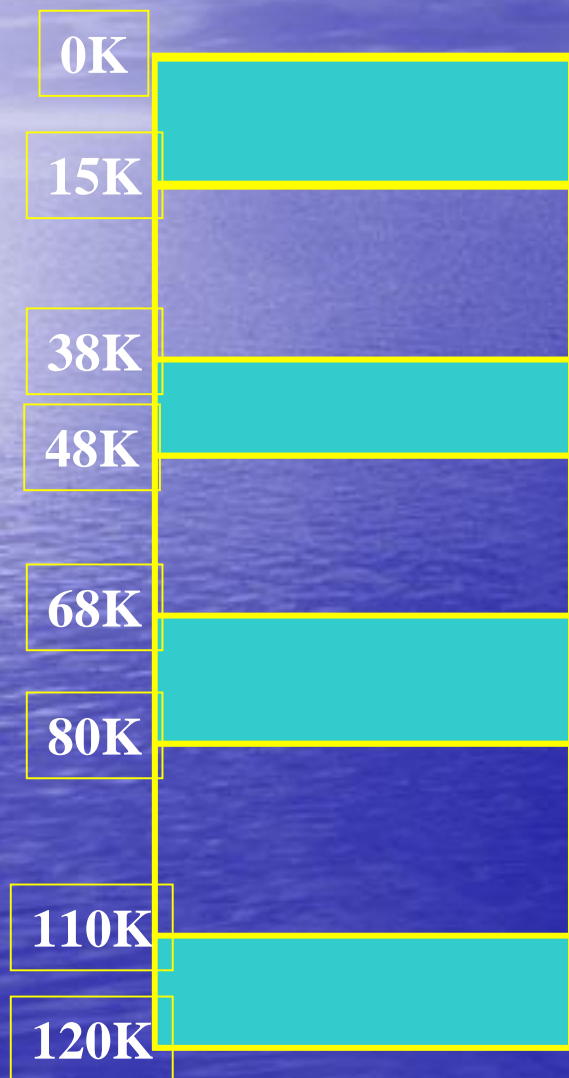


图5.8 内存分配变化过程



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

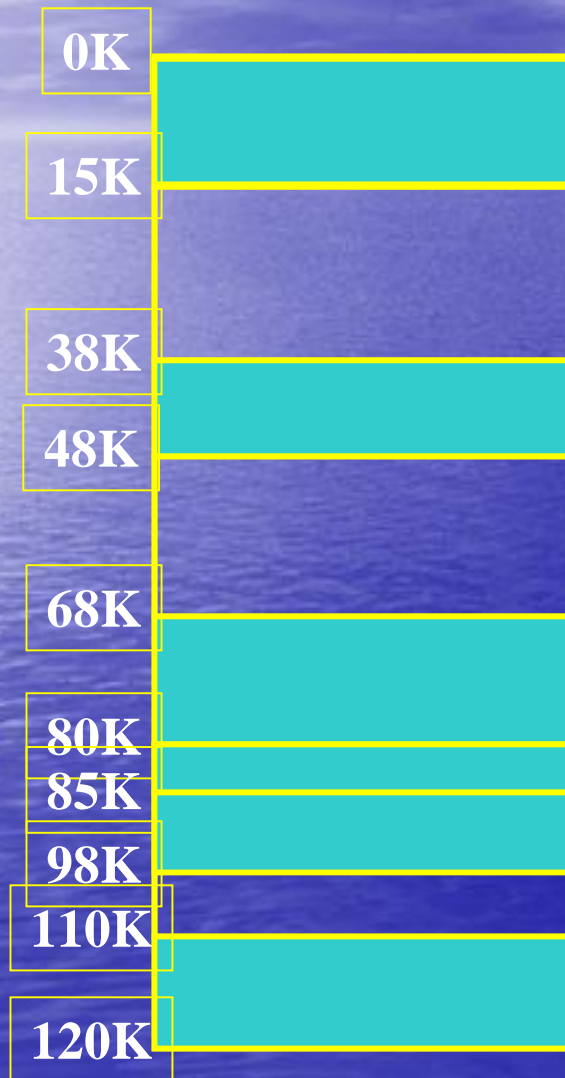
始址	长度	标志
0K	15K	J1
38K	10K	J2
68K	12K	J3
110K	10K	J4
		空
		空

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

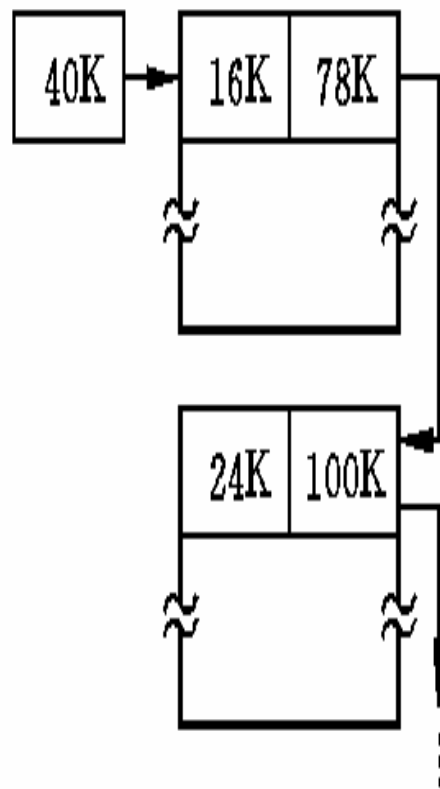


除了分区说明表之外，动态分区法还把内存中的可用分区单独构成可用分区表或可用分区自由链，以描述系统内的内存资源。与此相对应，请求内存资源的作业或进程也构成一个内存资源请求表。图5.9给出了可用表，自由链和请求表的例子。

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。

区号	分区长度	起始地址
1	16K	40K
3	24K	78K
5	9K	100K

(a) 可用表

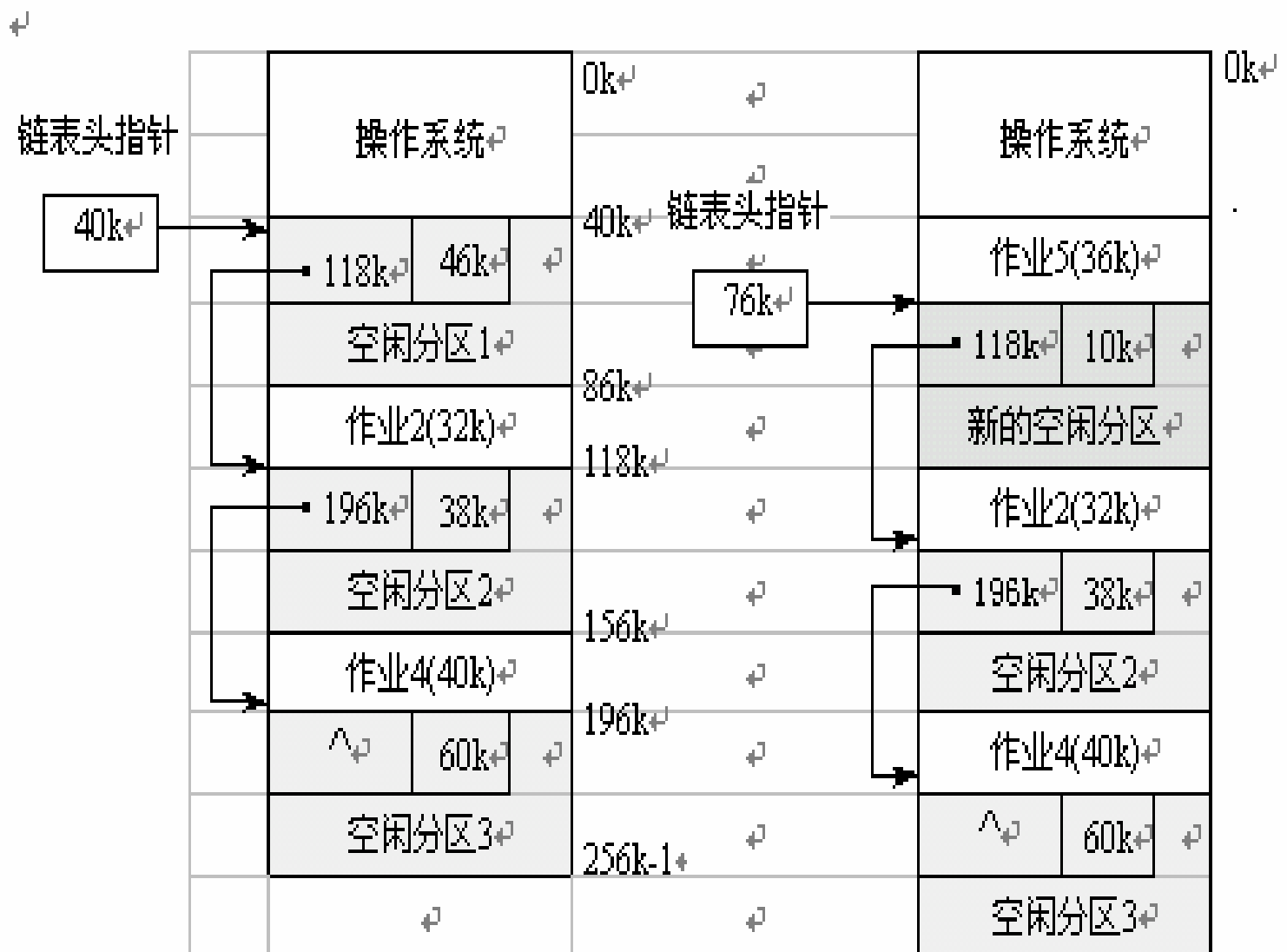


(b) 自由链

作业(进程)号	请求长度
P ₁	13K
P ₂	20K
	⋮

(c) 请求表

图5.9 可用表、自由链及请求表



(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。



Ø 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。

Ø 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)

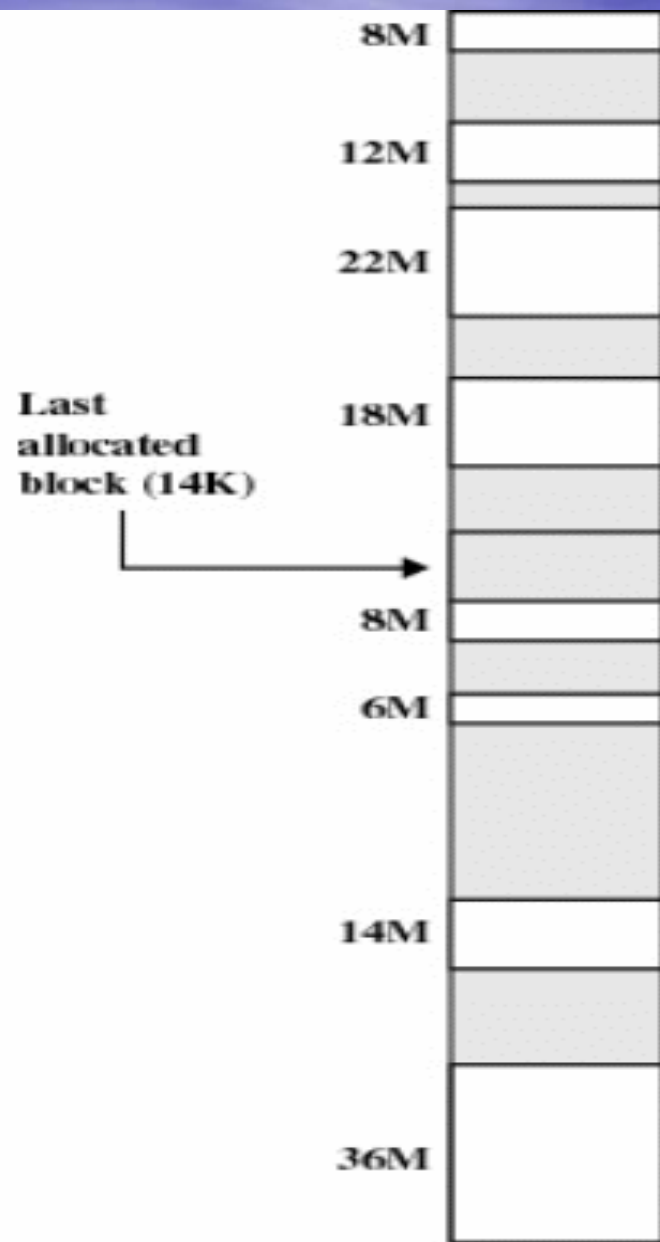
动态分区时的分配与回收

动态分区时的分配与回收主要解决三个问题:□

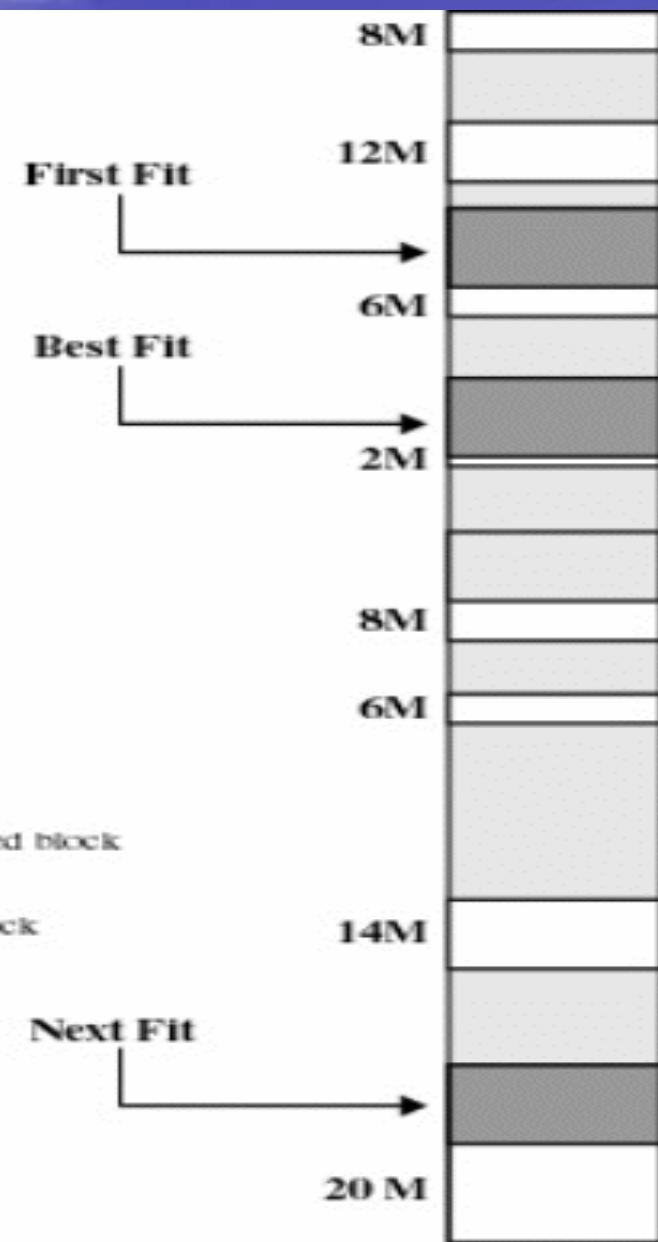
- (1) 对于请求表中的要求内存长度，从可用表或自由链中寻找出合适的空闲区分配程序。
- (2) 分配空闲区之后，更新可用表或自由链。
- (3) 进程或作业释放内存资源时，和相邻的空闲区进行链接合并，更新可用表或自由链。

- 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。



(a) Before



(b) After

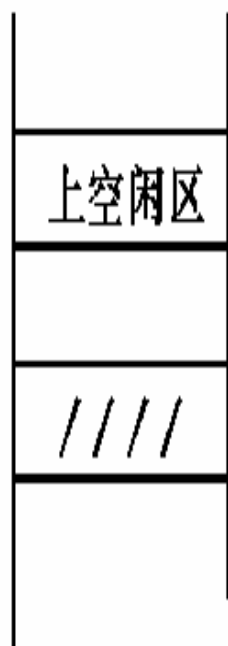
动态分区时的回收与拼接

Ø 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：

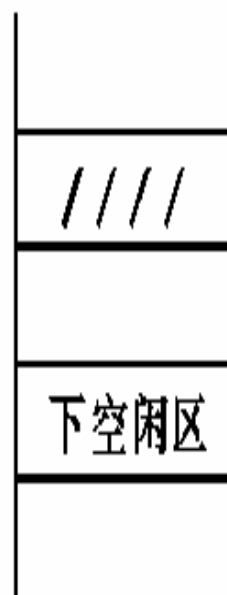
- a) 该空闲区的上下两相邻分区都是空闲区。
- b) 该空闲区的上相邻区是空闲区
- c) 该空闲区的下相邻区是空闲区
- d) 两相 邻区都不是空闲区。



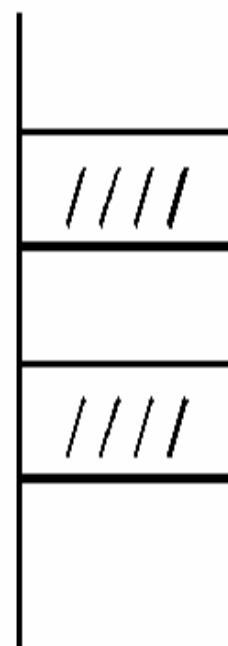
(a) 上下相邻区
都是空闲区



(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

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:

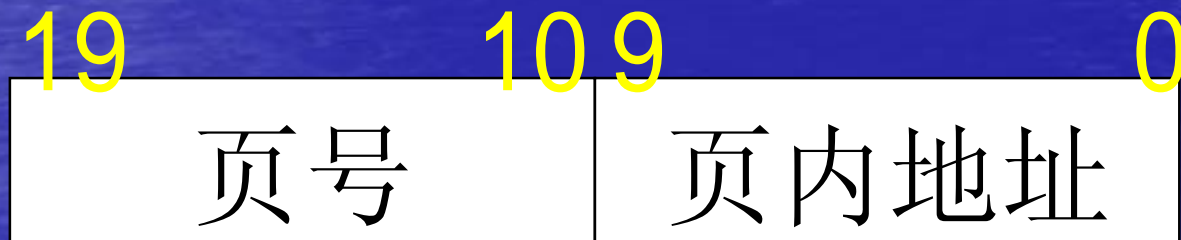
分区式管理存在着严重碎片问题,各作业或进程连续存放,进程的大小仍受分区大小或内存可用空间的限制。

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

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） \rightarrow 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	

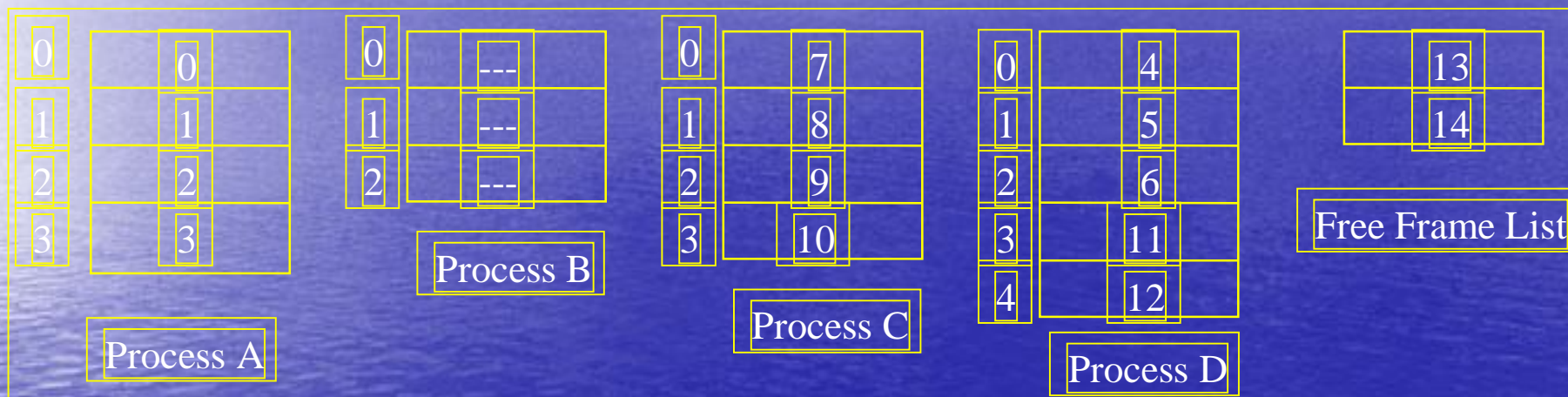
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

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	

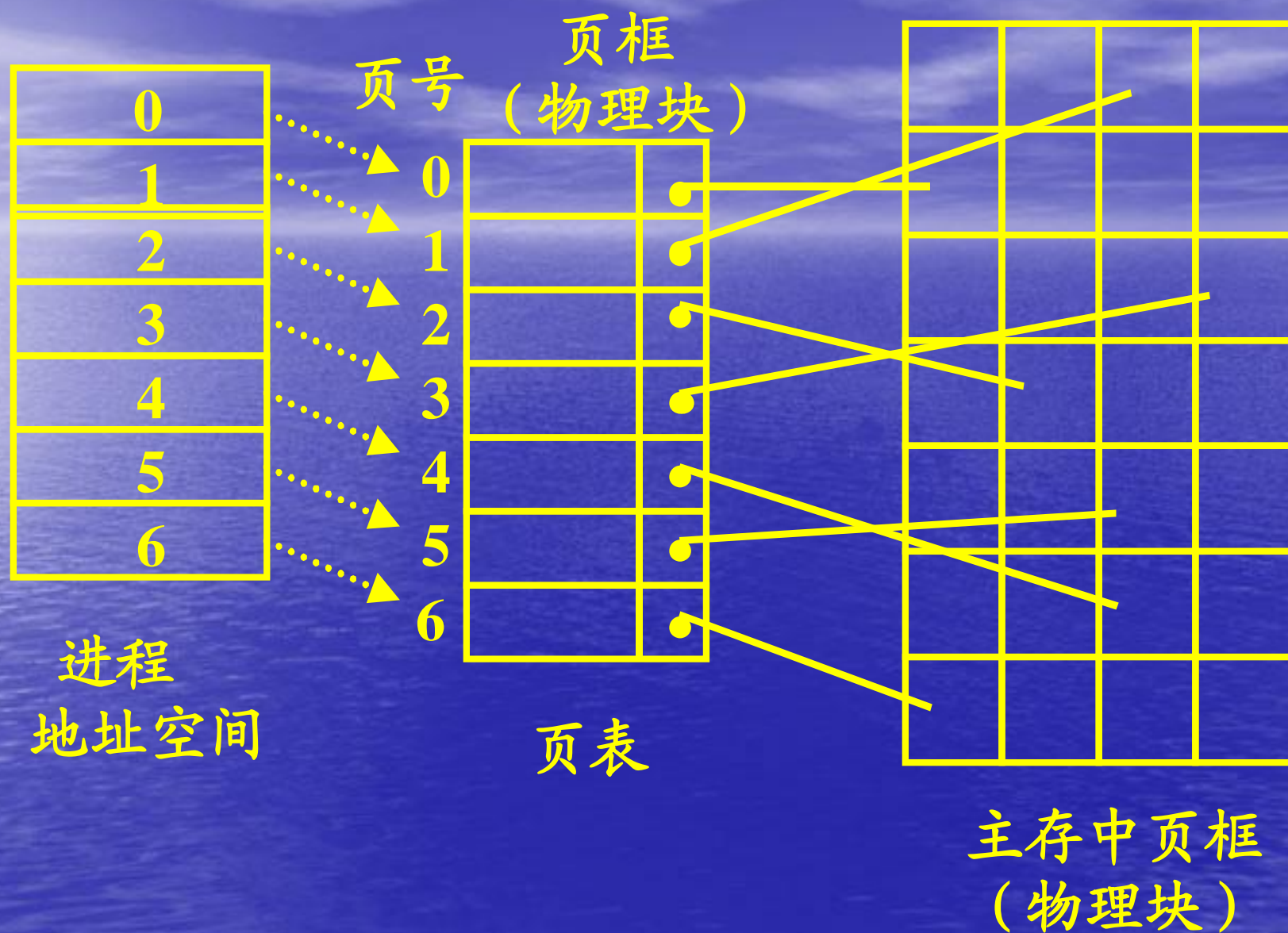
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

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	



进程页表



(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。

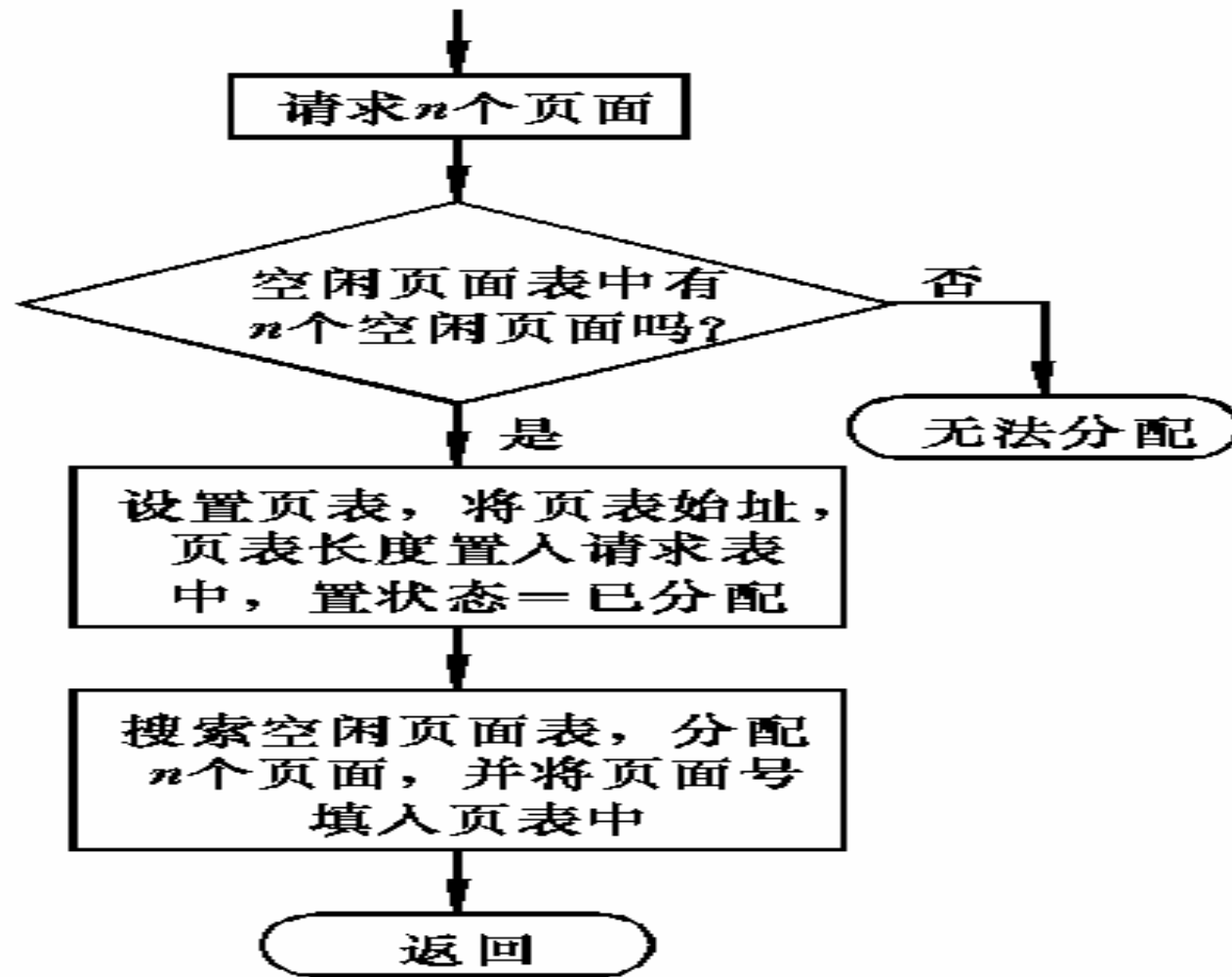
19	18	17	16	15	4	3	3	2	0
0	1	1	1	1	1	1	0	1	1
0	0	0	1	1	0	0	1	1	0
0	0	1	1	1	0	0	0	0	0

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

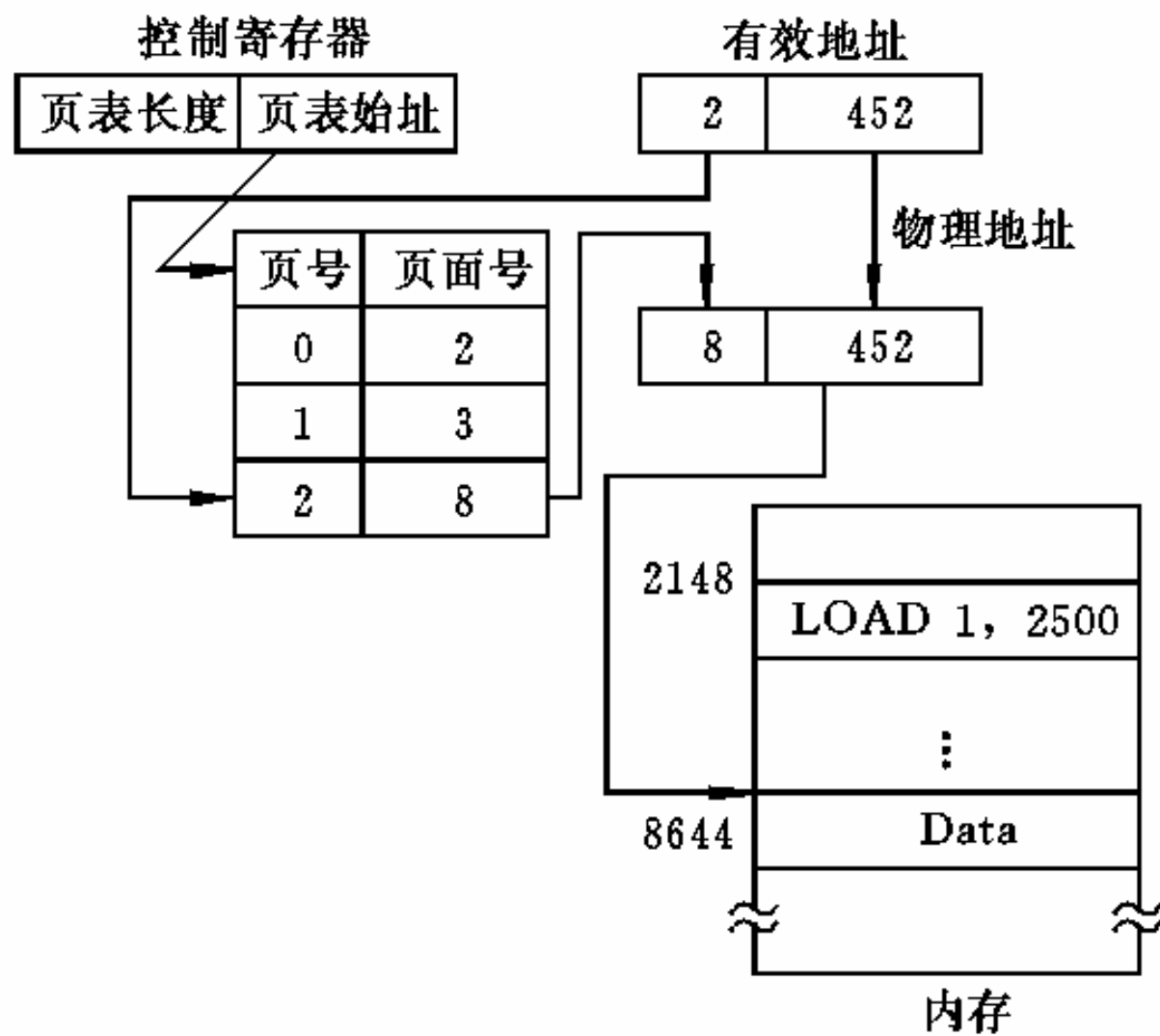


地址变换

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。



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？

$a[0][0], a[0][1], \dots, a[0][99]$

$a[1][0], a[1][1], \dots, a[1][99]$

...

$a[99][0], a[99][1], \dots, a[99][99]$

$100/2=50$ 次缺页中断。

$a[0][0], a[1][0], \dots, a[99][0]$

$a[0][1], a[1][1], \dots, a[99][1]$

...

$a[0][99], a[1][99], \dots, a[99][99]$

$10000/2 = 5000$ 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：

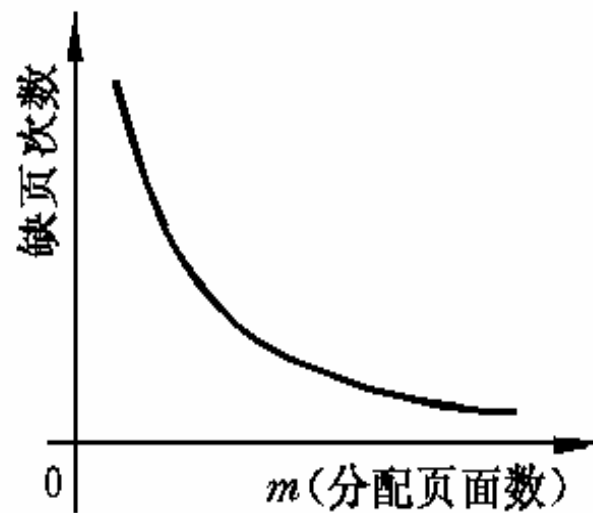
- (1) 随机淘汰算法。在系统设计人员认为无法确定哪些页被访问的概率较低时，随机地选择某个用户的页面并将其换出将是一种明智的作法。
- (2) 轮转法和先进先出算法。轮转法循环换出内存可用区内一个可以被换出的页，无论该页是刚被换进或已换进内存很长时间

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。

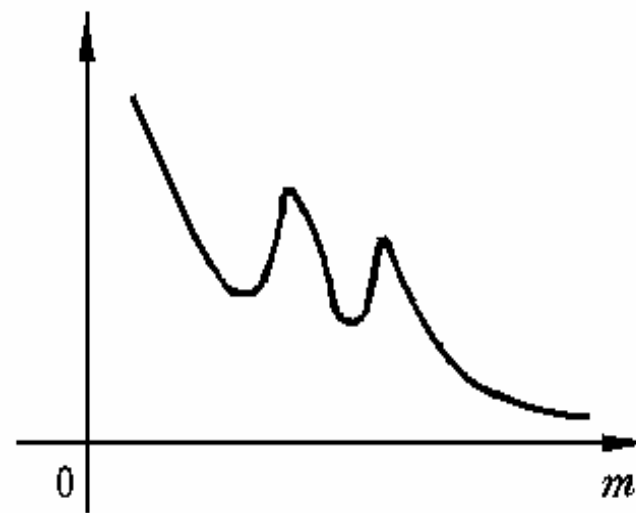
由实验和测试发现FIFO算法和RR算法的内存利用率不高。这是因为，这两种算法都是基于CPU按线性顺序访问地址空间的这个假设上。事实上，许多时候，CPU不是按线性顺序访问地址空间的，例如执行循环语句时。因此，那些在内存中停留时间最长的页往往也是经常被访问的页。尽管这些页变“老”了。但它们被访问的概率仍然很高。

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
0																	
页	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0
0																	
页		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1
1																	
页			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2
2																	
缺	X	x	x	x	√	x	x	x	x	x	x	√	√	x	x	√	√
页																	

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。

[illegible]

[illegible]

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。

[illegible]

O P T	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。

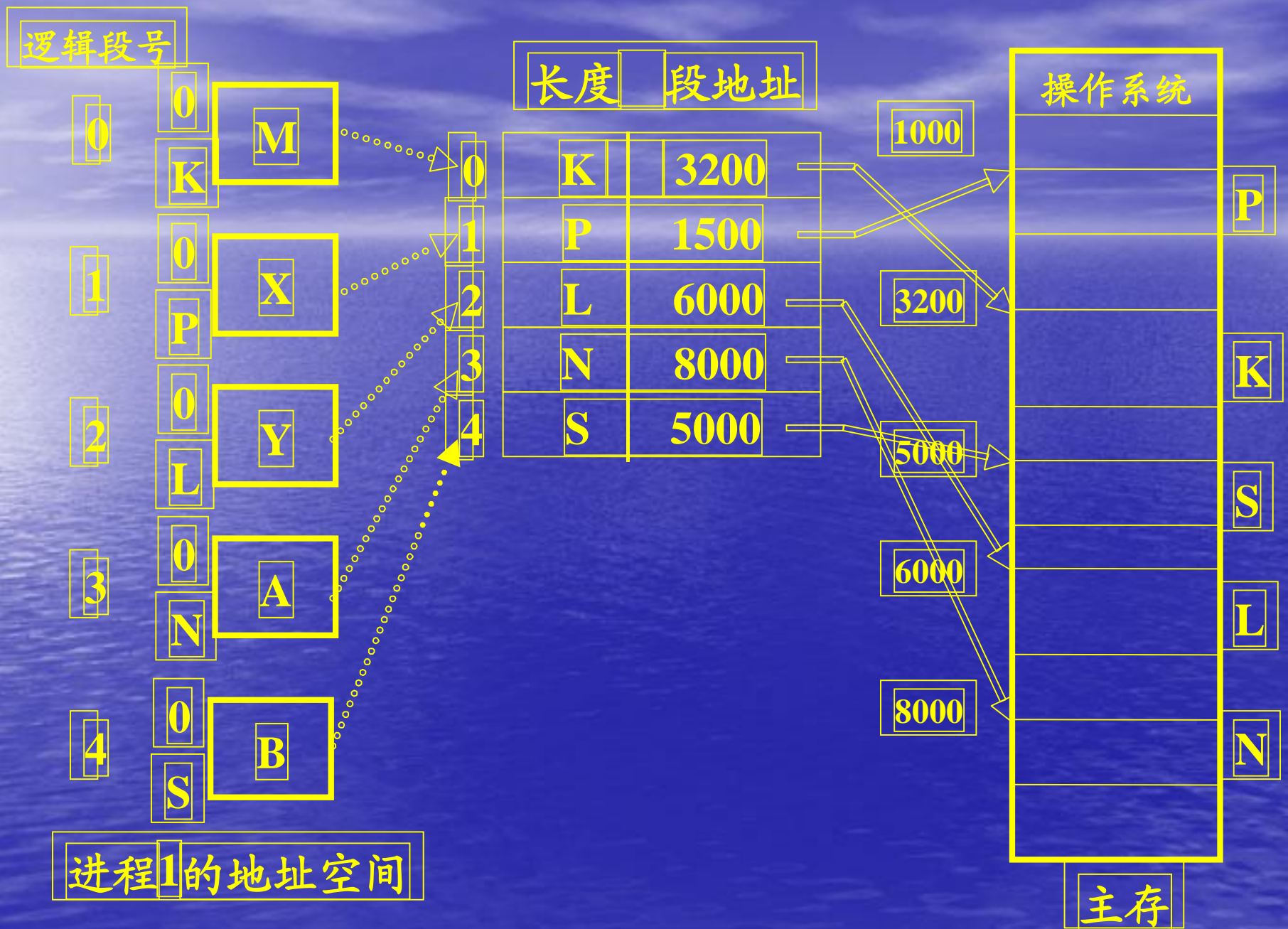
- 可以分别编写和编译
- 可以针对不同类型的段采取不同的保护
- 可以按段为单位来进行共享，包括通过动态链接进行代码共享

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。



简单段式管理的数据结构

- Ø 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
- Ø 系统段表：系统内所有占用段
- Ø 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

(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	

虚拟地址

1 | 120

段号 段内地址

3400



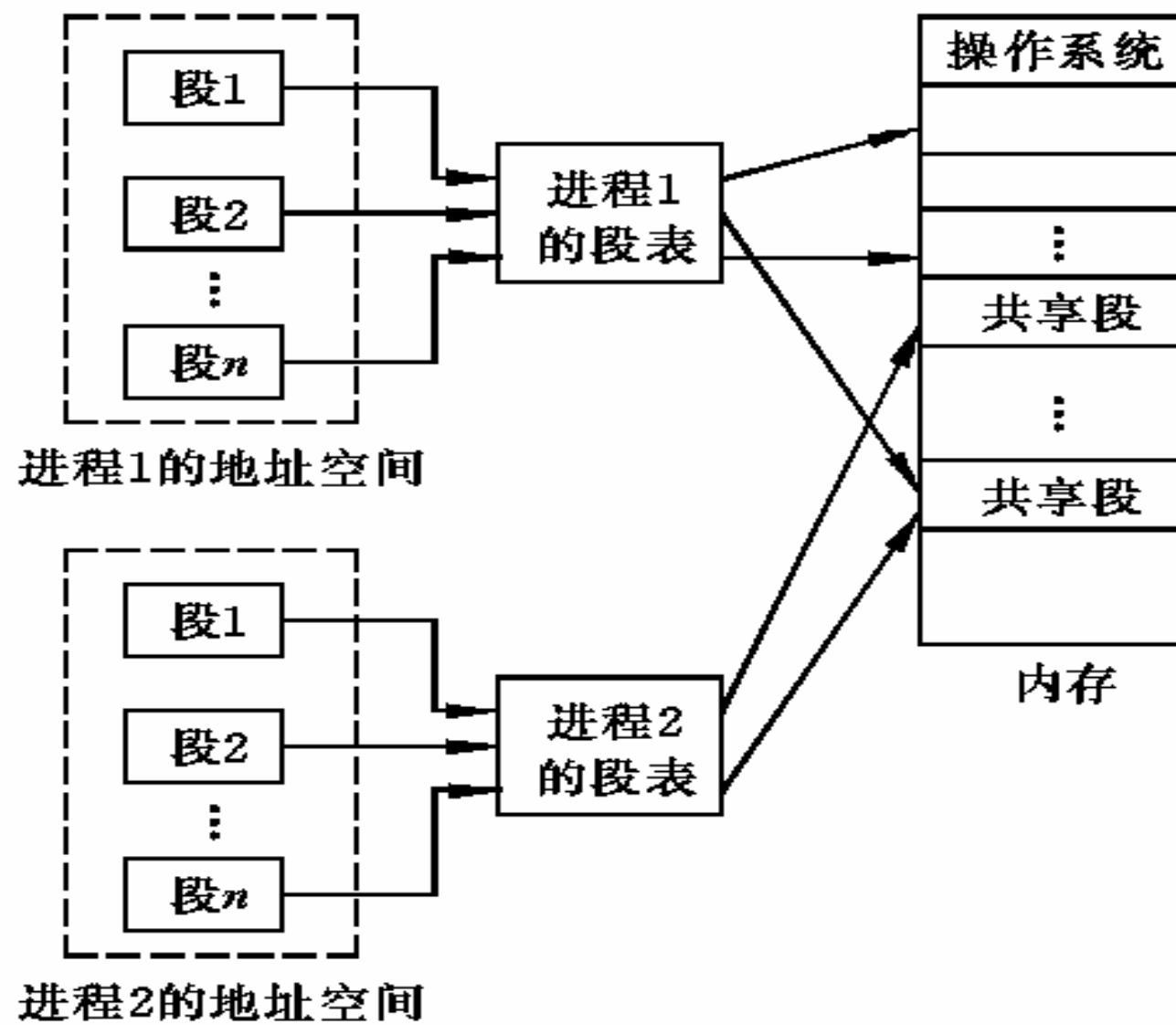
3520

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。

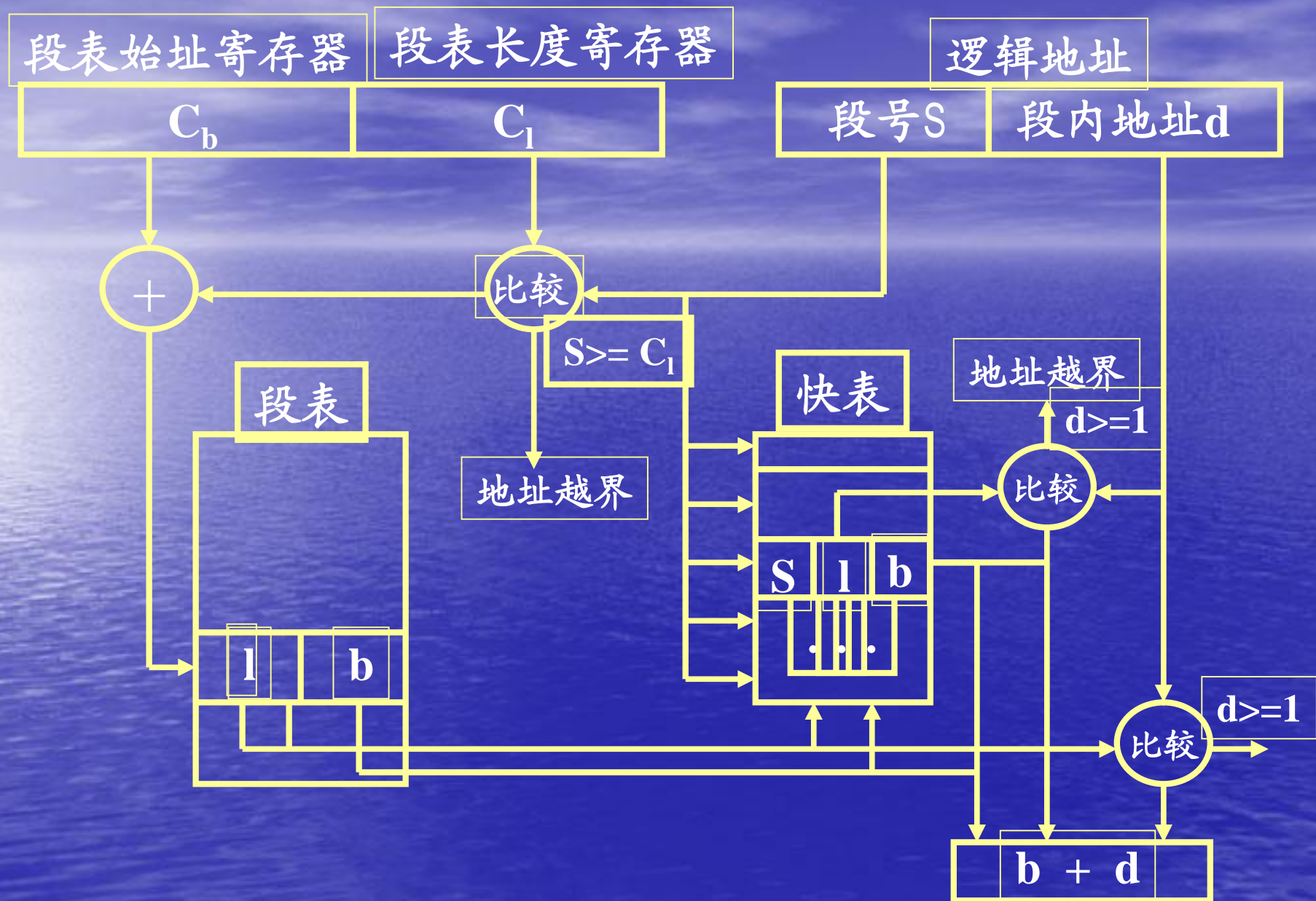


只要用户使用相同的段名，就可新的段表中填入已存在于内存之中的段的起始地址，并置以适当的读写控制权，就可做到共享一个逻辑上完整的内存段信息。

(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。



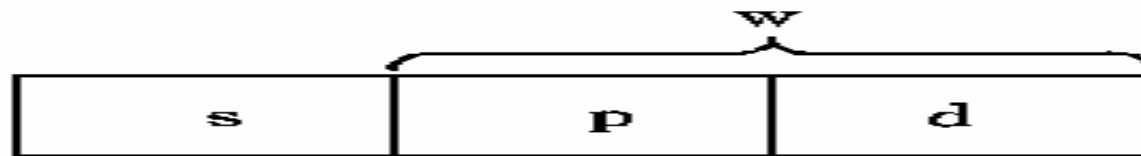
段页式管理的实现原理

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

Ø 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。

Ø 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 。



- Ø 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。
- Ø 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

Ø 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，并能提前将它们调入内存，将会大大降低缺页率，减少置换工作，提高CPU的利用率。

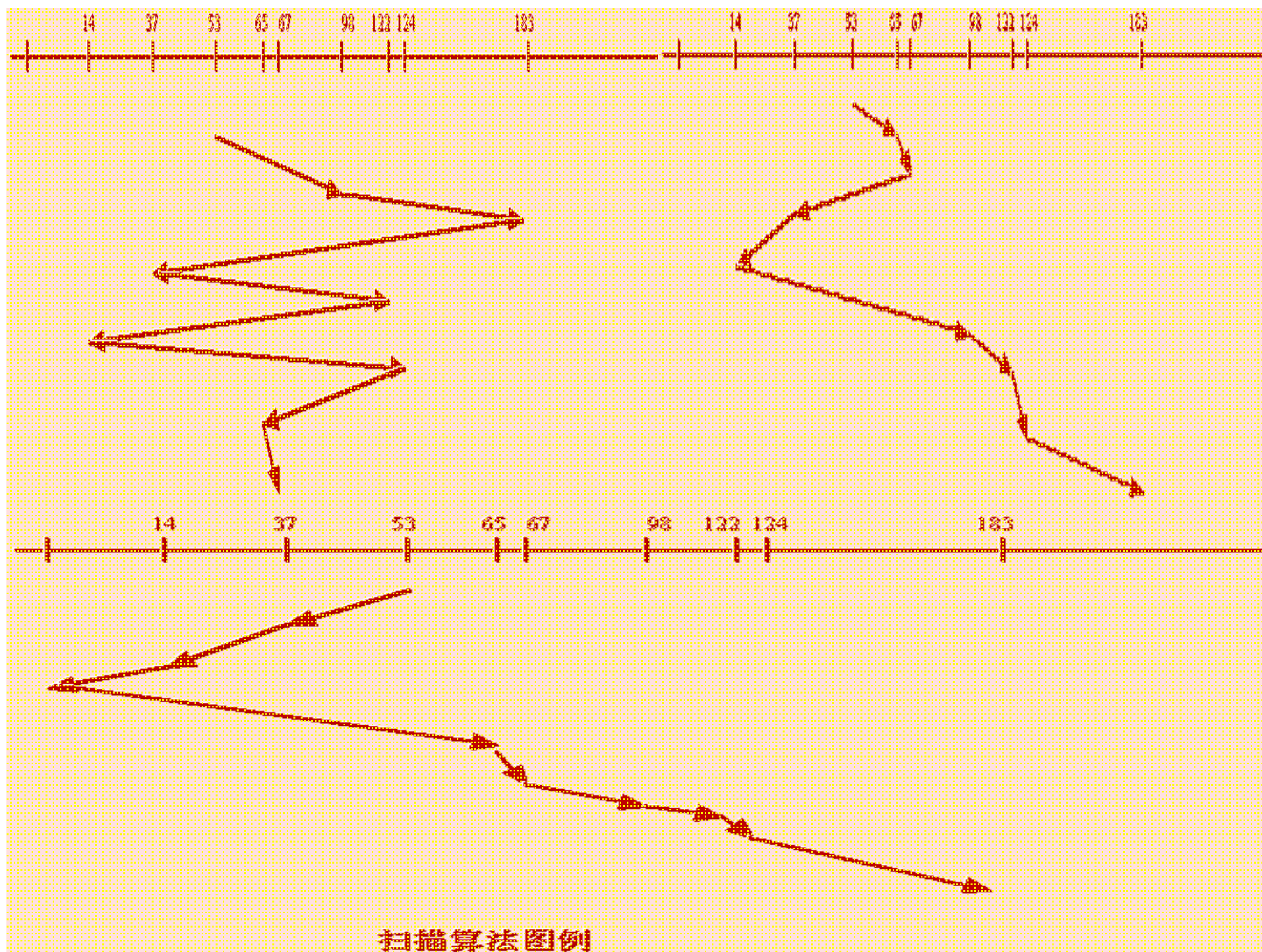
Ø 所谓工作集是指在某段时间间隔A里，进程实际要访问的页面集合。Denning认为，虽然程序只需少量的几页已在内存就可运行，但为使程序能有效地运行，较：p地产生缺页，就必须使程序的工作集全部在内存中

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.
- (2) 最短寻道时间优先
- (3) 扫描(SCAN)算法
- (4) 循环扫描(CSCAN)算法



置换算法

如同请求调页(段)一样，在将磁盘中的盘块数据读入高速缓存时，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。

00000000

2GB用户进程空间
分配给每一个进程的

应用程序代码
全局数据
每个线程的堆栈
DLL代码

7FFFFFFF
80000000

内核与执行体
HAL
引导驱动程序

C0000000

进程页表
超空间

C0800000

系统高速缓存
页面交换区
非页面交换区

FFFFFFFF

2GB系统空间，留给操作系统的

Windows2000/XP进程地址空间布局

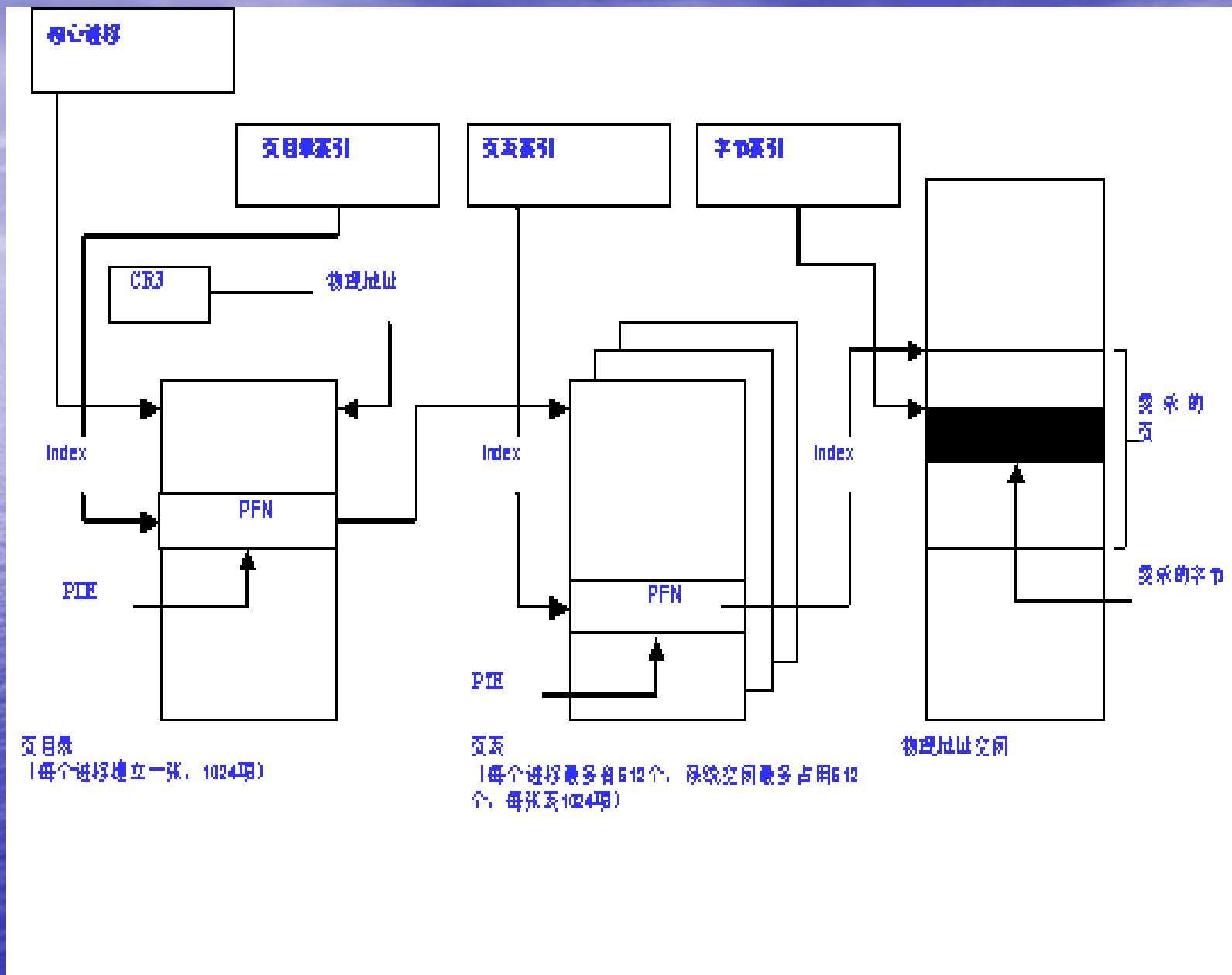
范围	大小	功能
0x0到0xFFFF	64KB	拒绝访问区域，用于帮助程序员避免引用错误的指针；试图访问这个区域地址的操作将会导致访问侵犯。
0x10000到0x7FFEFFFF	2GB减掉至少192KB	专用进程地址空间
0x7EFDE000到0x7EFDEFFF	4KB	用于第1个线程的线程环境块（TEB）。系统会在这一页的前面创建附加的TEB（从地址0x7FFDD000开始向上）。
0x7FFDF000到0x7FFDFFFF	4KB	进程环境块（PEB）
0x7FFE0000到0x7FFE0FFF	4KB	共享的用户数据页，这个只读方式的页面被映射到系统空间中包含系统时间、时钟计数和版本号信息的一个页面。这个页面的存在使数据在用户态下可以直接读取而不必请求核心态转换。
0x7FFE1000到0x7FFEFFFF	64KB	拒绝访问区域（共享用户数据页面以后剩余的64KB）。
0x7FFF0000到0x7FFFFFFF	64KB	拒绝访问区域，用于防止线程跨越用户/系统空间边界传送缓存区。在MmUserProbeAddress中包含此页的起始地址。

x86系统的系统内存地址空间布局

范围	大小	功能
0x80000000到0x9FFFFFFF	512MB	启动系统的系统代码（NTOSKRNL.EXE、HAL.DLL和启动驱动程序）和非页交换区的初始部分。在有2GB系统空间和32MB或更多RAM的x86系统中，前4MB是用一个x86大页面PTE映射的。
0xA0000000到0xA2FFFFFF	48MB	用于系统映射视图的空间（目前用于映射Win32子系统的核心态部分Win32k.sys，以及它所用的核心态图形驱动程序）。
0xA3000000到0xBFFFFFFF	464MB	大多数Windows NT系统不使用。
0xC0000000到0xC03FFFFFFF	4MB	进程页表（页目录在0xC0300000，大小为4KB）。这是映射到系统空间内的每一个进程的数据。
0xC0400000到0xC07FFFFFFF	4MB	工作集表和超空间。这是映射到系统空间内的每一个进程的数据。
0xC0800000到0xC0BFFFFFFF	4MB	未使用
0xC0C00000到0xC0FFFFFFF	4MB	系统工作集表
0xC1000000到0xE0FFFFFFF	512MB（最大值）	系统高速缓存（启动时计算它的大小）
0xE1000000到0xECFFFFFFF	192MB（最大值）	页交换区（启动时计算它的大小）
0xEB000000到0xFFBDBFFF	331.875MB	系统PTE和非页交换区（启动时计算它的大小）
0xFFBE0000到0xFFFFFFFF	4.125MB	故障转储结构和专用HAL数据结构。

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($2^{12}=4096$)。



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Win32API VirtualAlloc和VirtualAllocEx函数实现的。

Ø 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。

Ø 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)。

Ø 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

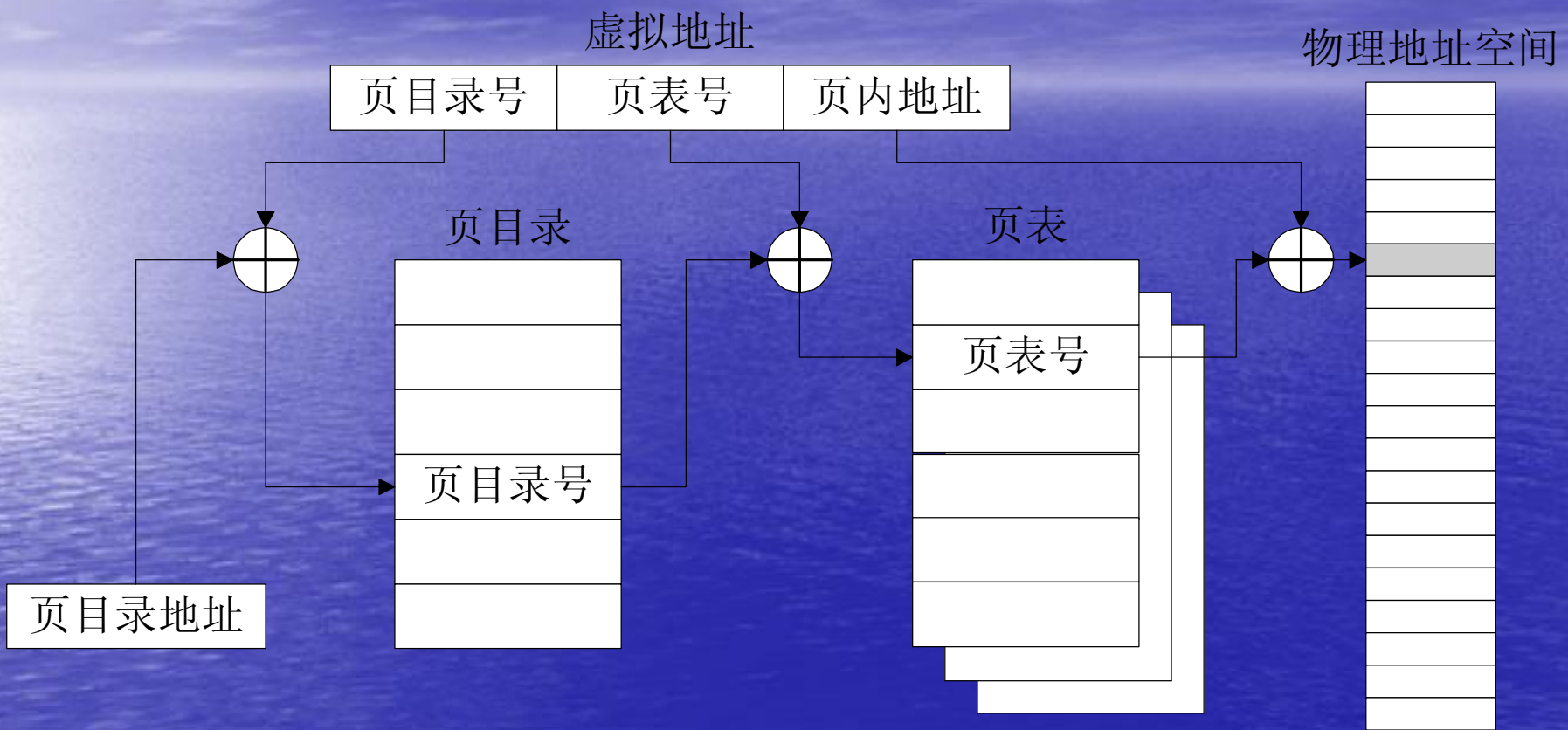
Ø现代操作系统能够使磁盘空间看起来像内存一样，为进程提供了虚拟存储器。Windows2000 / XP中磁盘上的部分通常称为“页文件”。如果计算机有64MB物理内存，同时在磁盘上有100MB的页文件，那么应用程序就可以认为计算机总共拥有164MB内存。

Ø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。

Windows2000的地址空间扩展

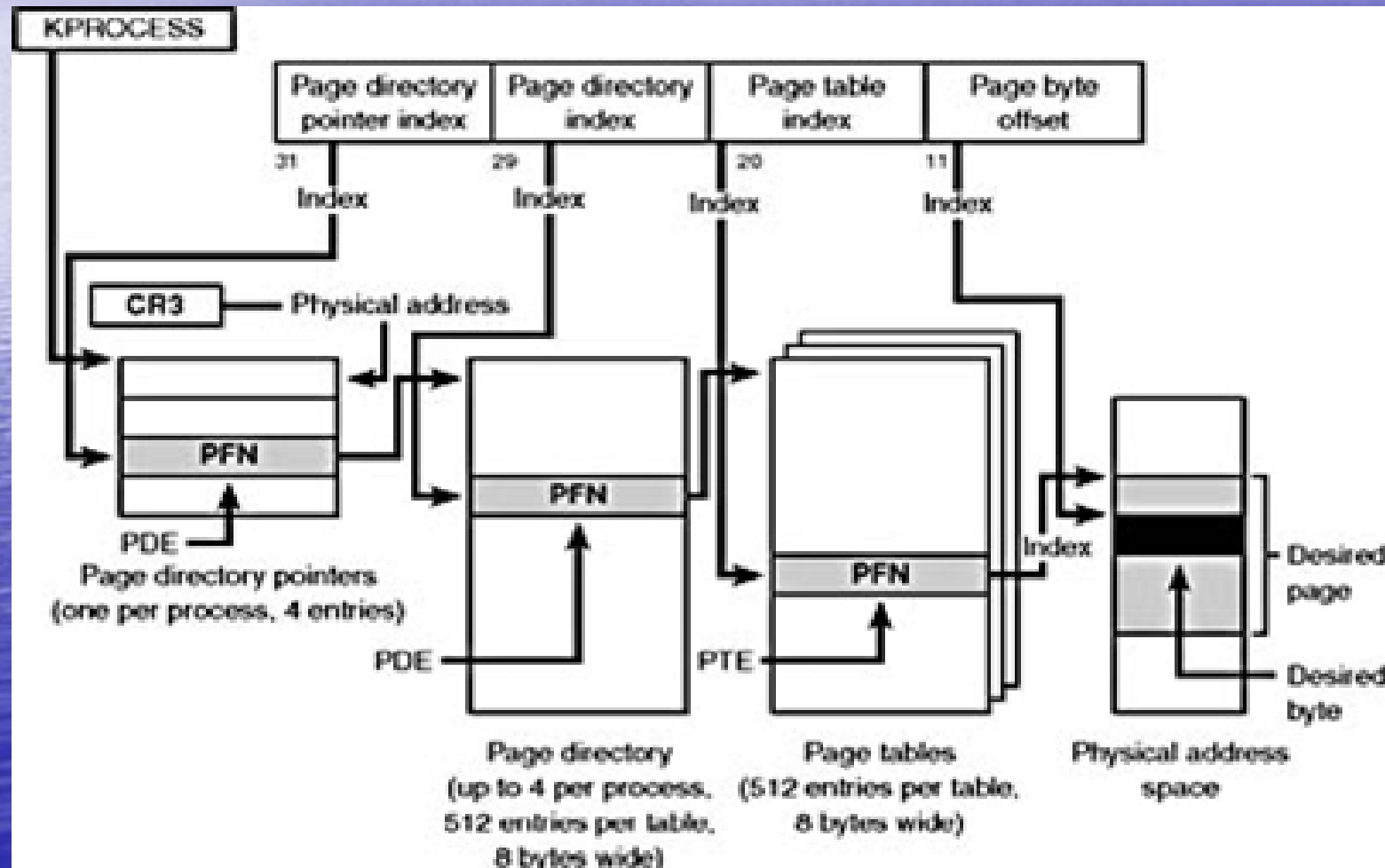
利用窗口映射(AWE, Address Windowing Extensions)方法可在一个进程中使用大于2GB或3GB的物理内存空间。使用步骤分成3步:

- 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;



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

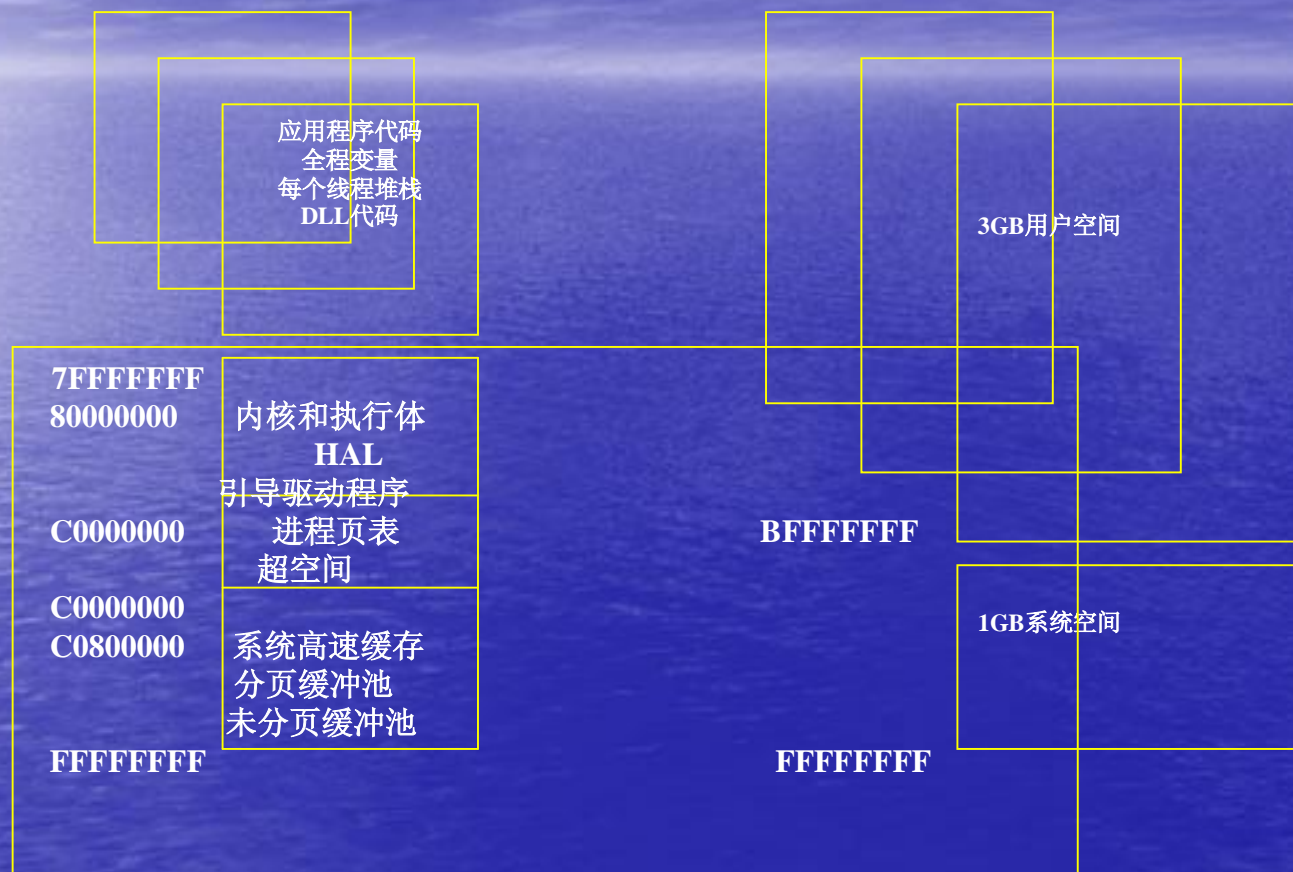
每个PDE和PTE表项都是8个字节，物理页面号为24Bit，可访问的物理地址空间可达64GB。



组成部分

- Ø 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。
- Ø 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。
- Ø 六个的关键组件

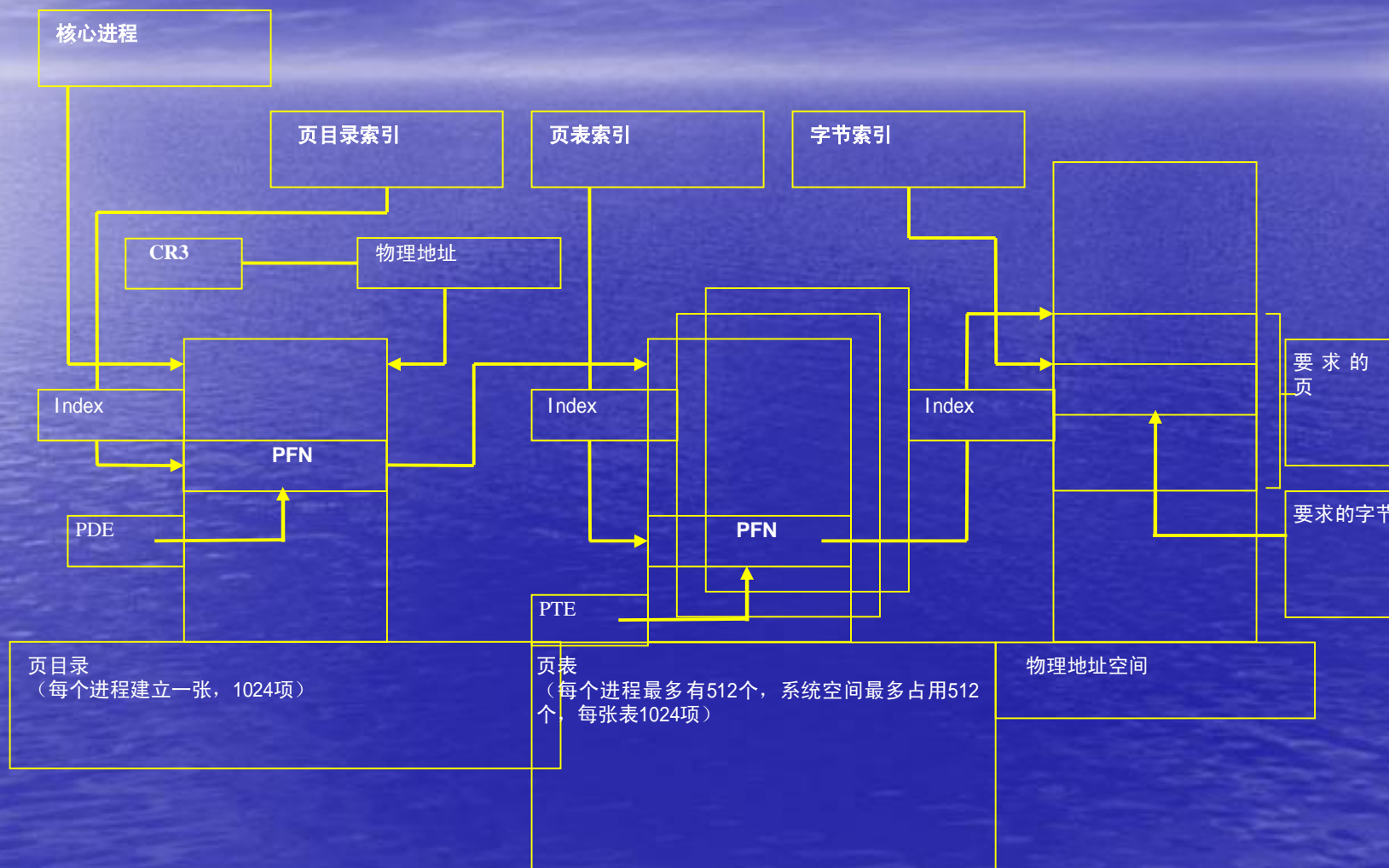
X86系统虚拟地址空间布局



X86系统空间布局

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

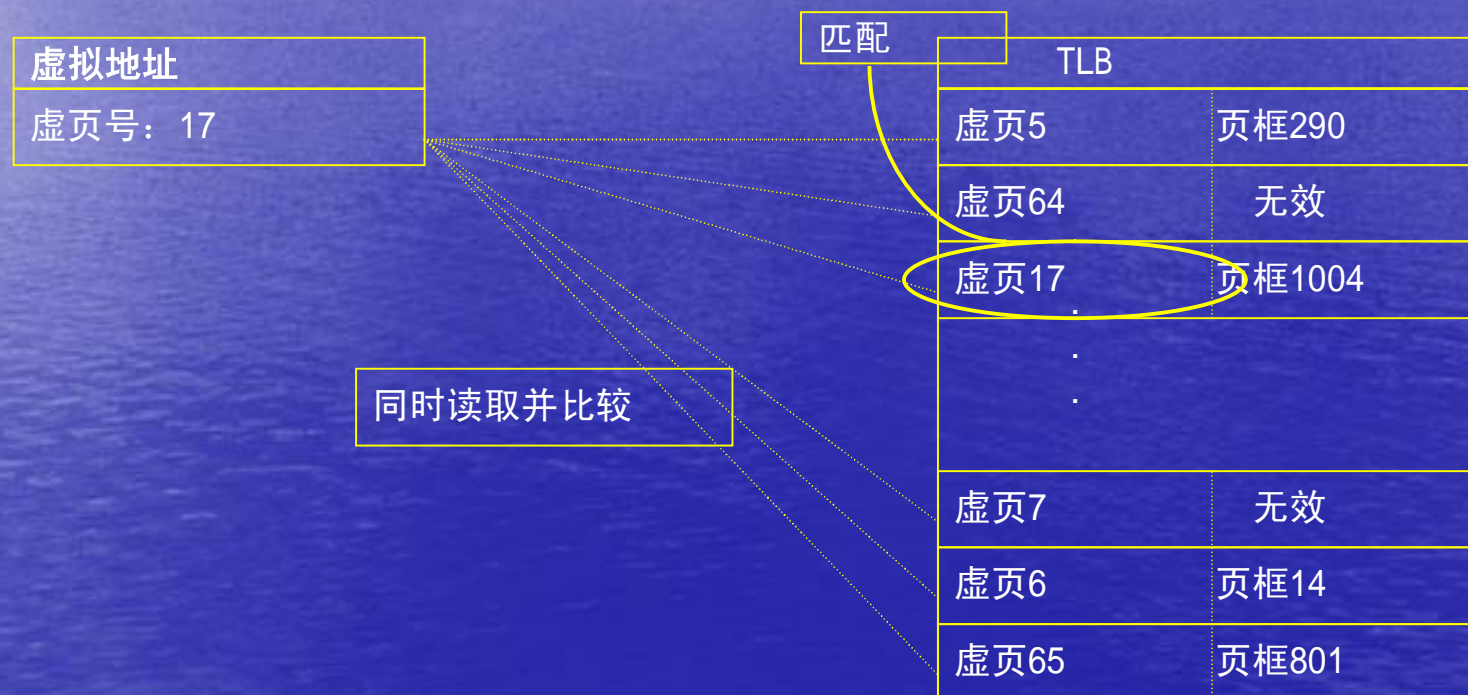
虚拟地址变换过程



系统页表与进程私有页表



快表TLB



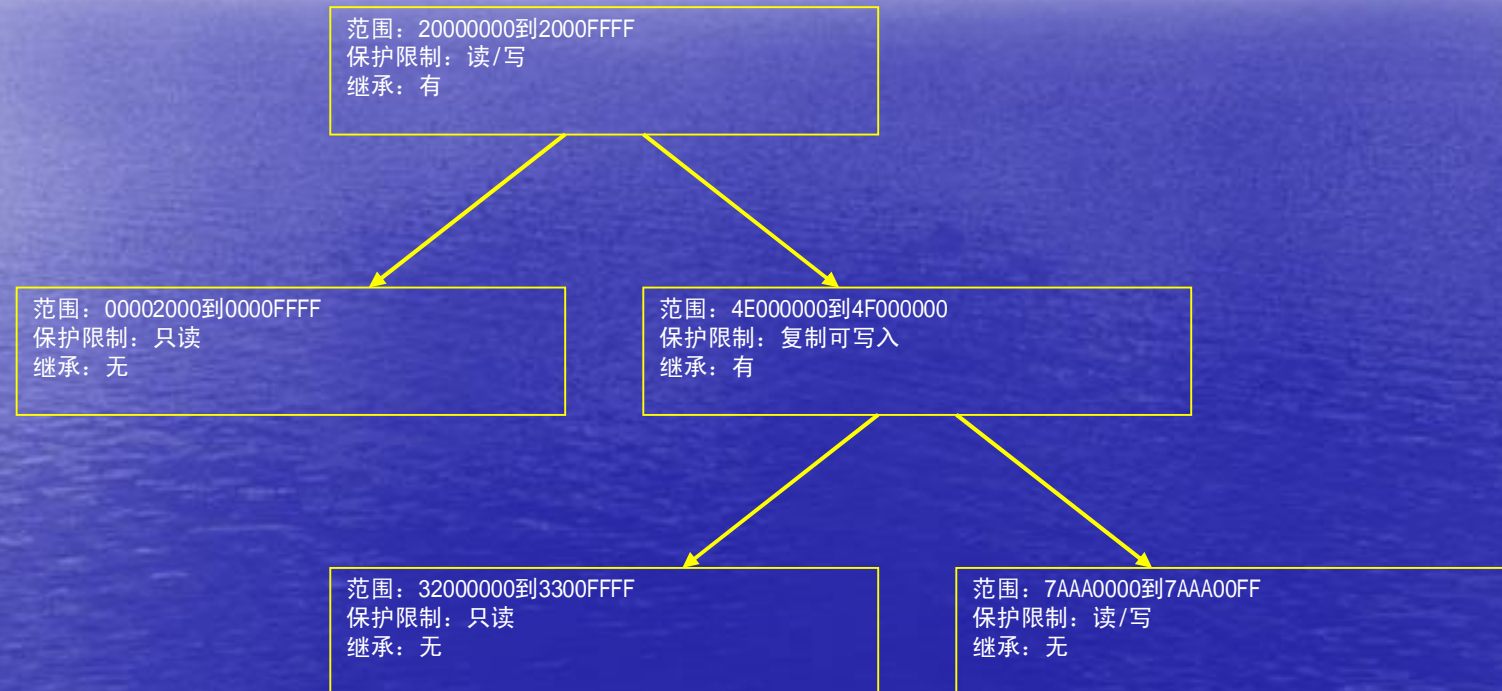
内存分配方式

Ø 以页单位的虚拟内存函数

Ø 内存映射文件函数

Ø 堆函数

虚拟地址描述符



区域对象

对象类型	区域
对象体属性	最大规模 页保护限制 页文件/映射文件 基准的/非基准的
服务程序	创建区域 打开区域 扩展区域 映射/非映射视图 查询区域

内部区域结构

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。

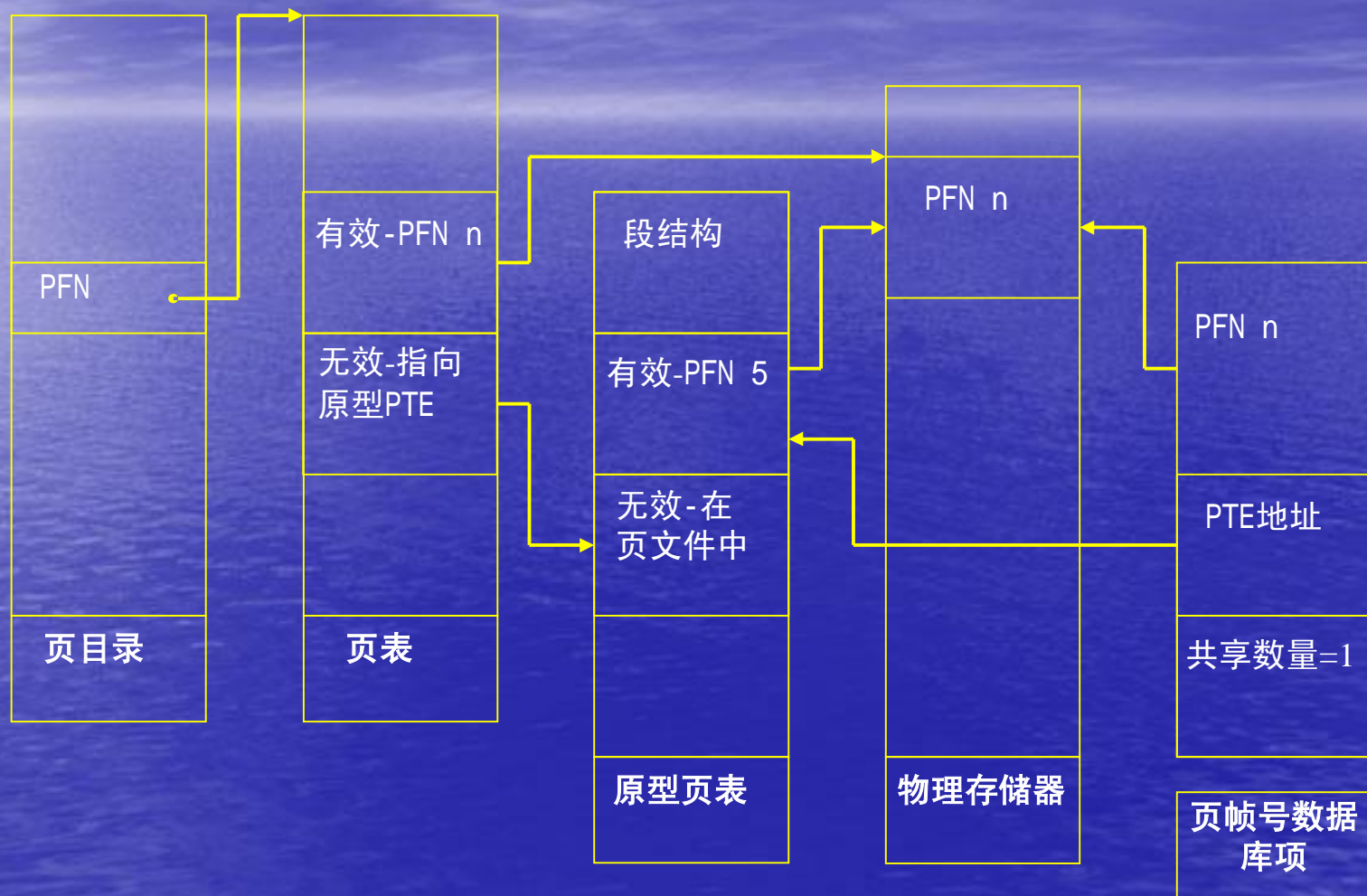
缺页处理

1 无效的页表项

- ┆ 页文件
- ┆ 请求零页
- ┆ 转换
- ┆ 未知

2 原型页表项

原型页表项



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

冲突页错误

页面调度程序检测

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。

工作集

页面调度策略

工作集管理

平衡集管理和交换程序

系统工作集

物理内存管理

页面状态:

活动（又称有效）

过渡(Transition)

后备(stand by)

修改

修改不写入

空闲

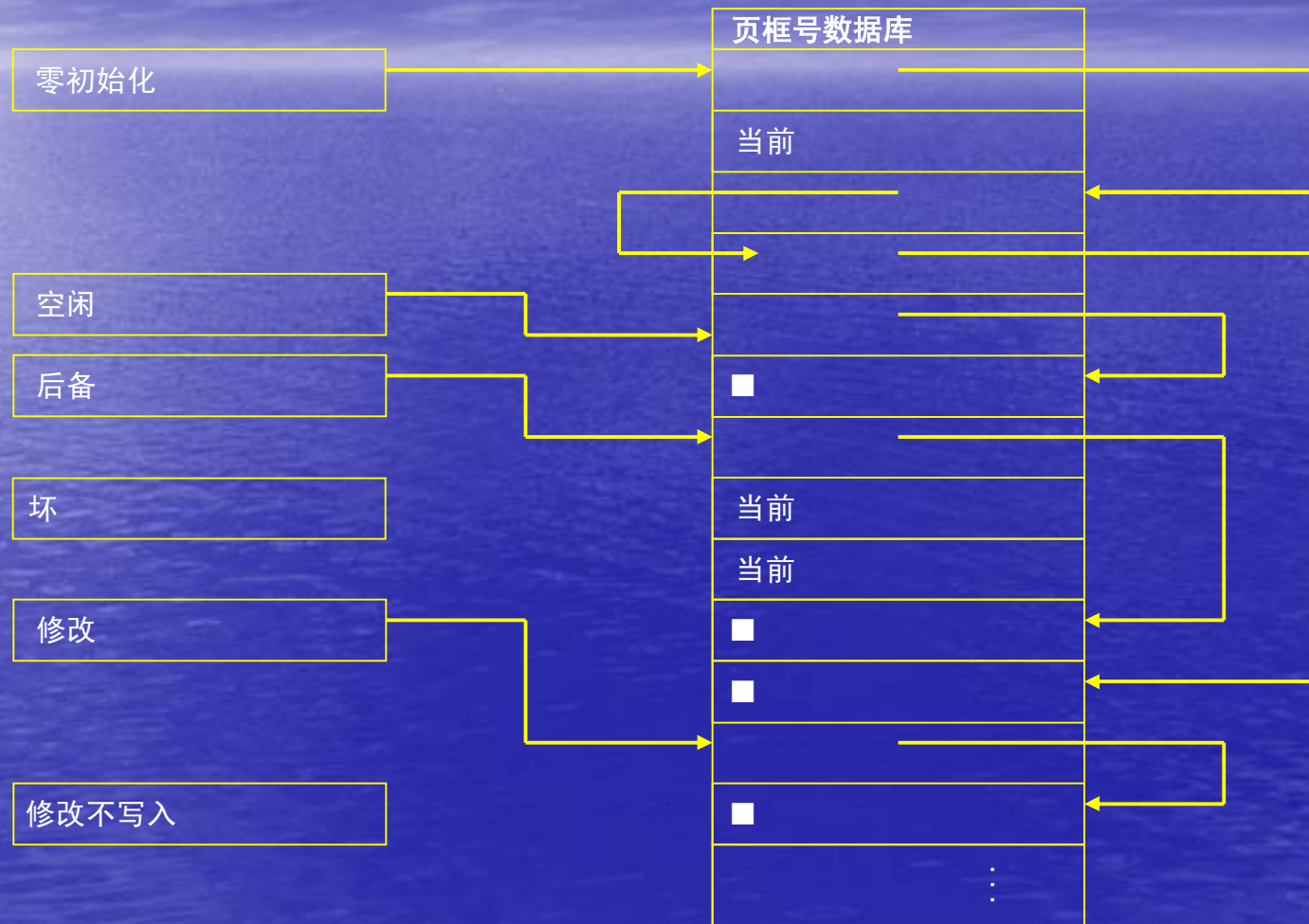
零初始化(zeroed)

坏

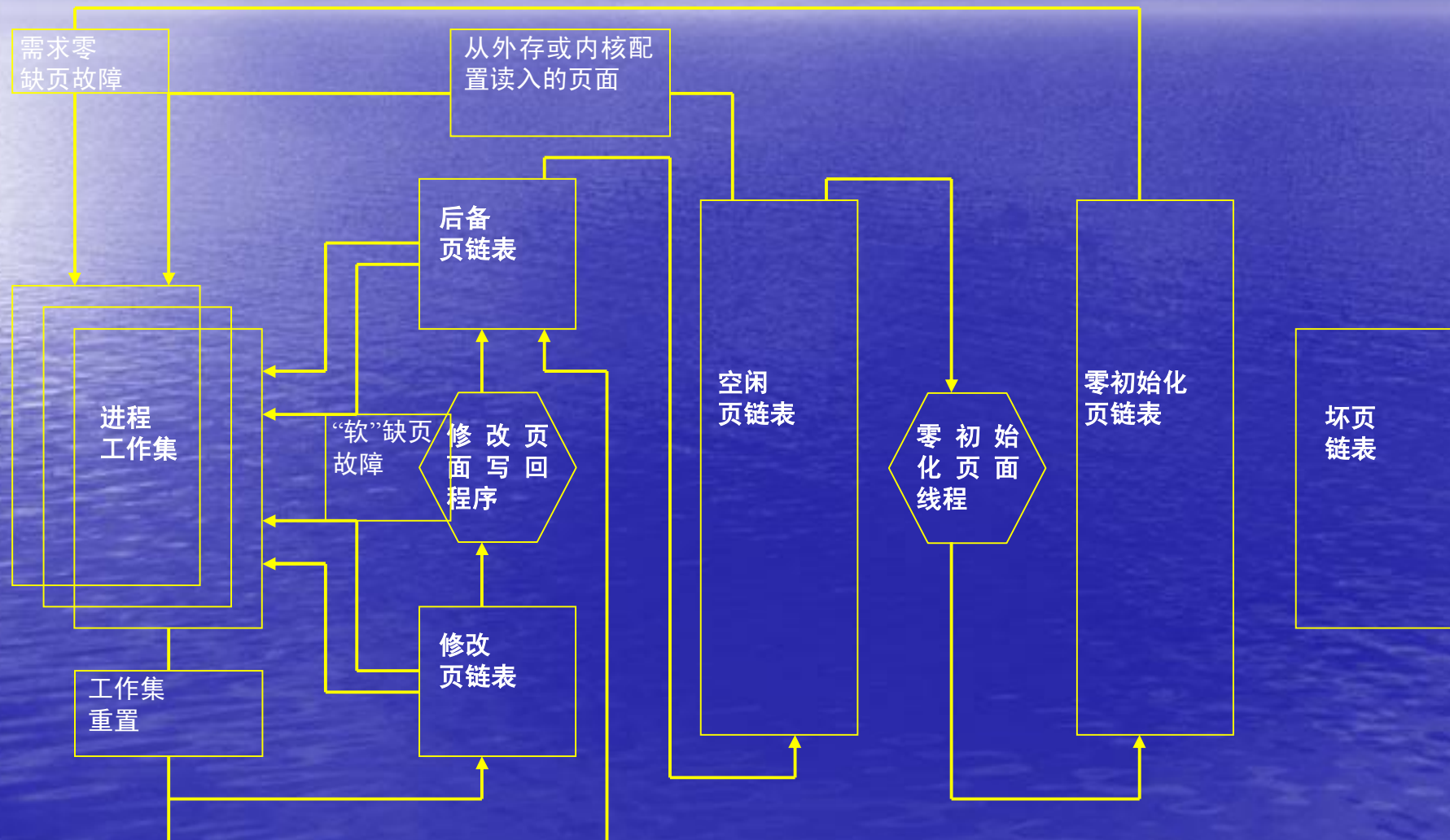
页表与页框号数据库



页框号数据库中的页链表



页框的状态图



锁内存

设备驱动程序可以调用核心态函数

Win32应用程序可以调用VirtualLock函数锁住进程工作集中的页面。

分配粒度

内存保护机制

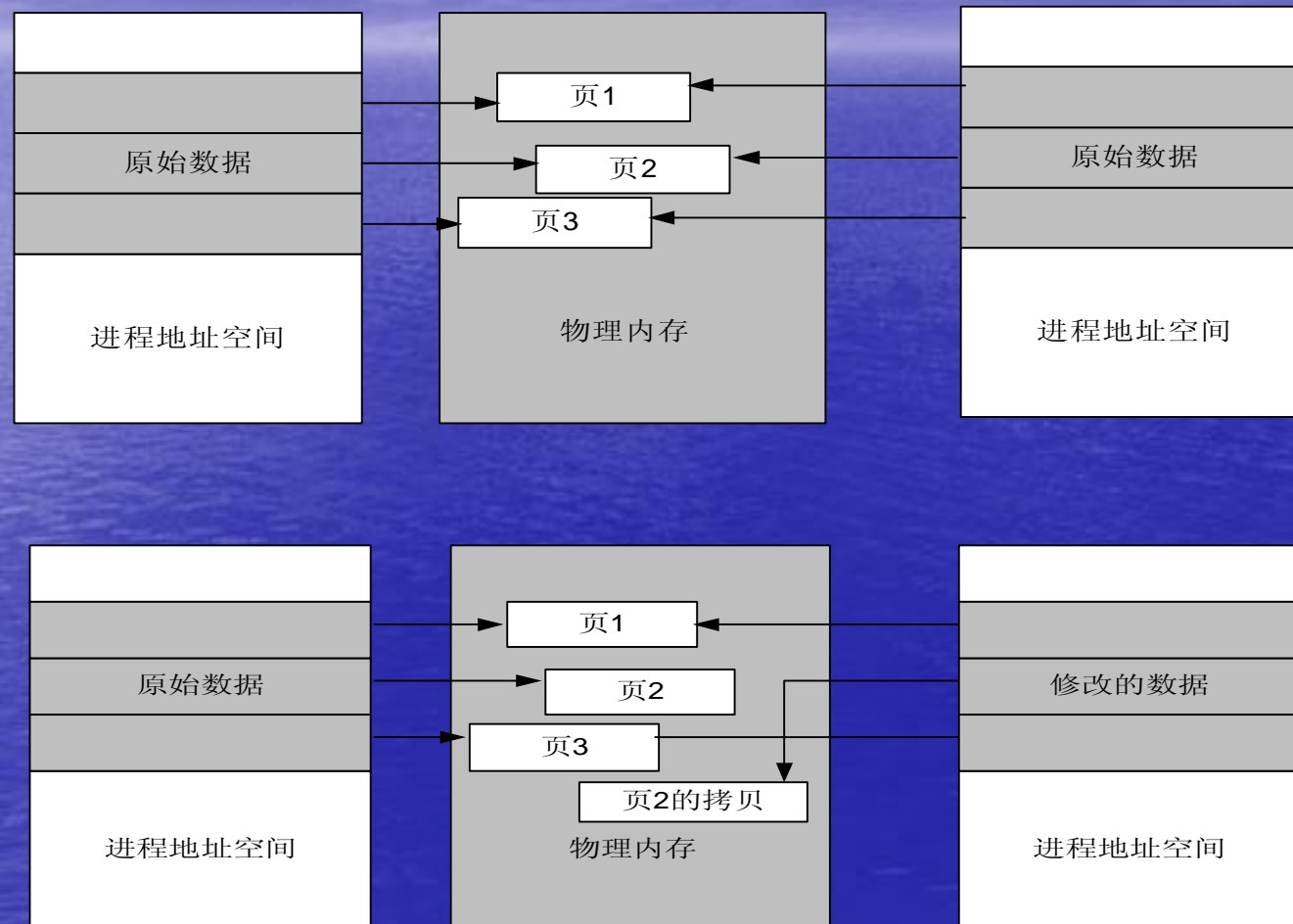
首先，所有系统范围内核心态组件使用的数据结构和内存缓冲池只能在核心态下访问。

第二，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。

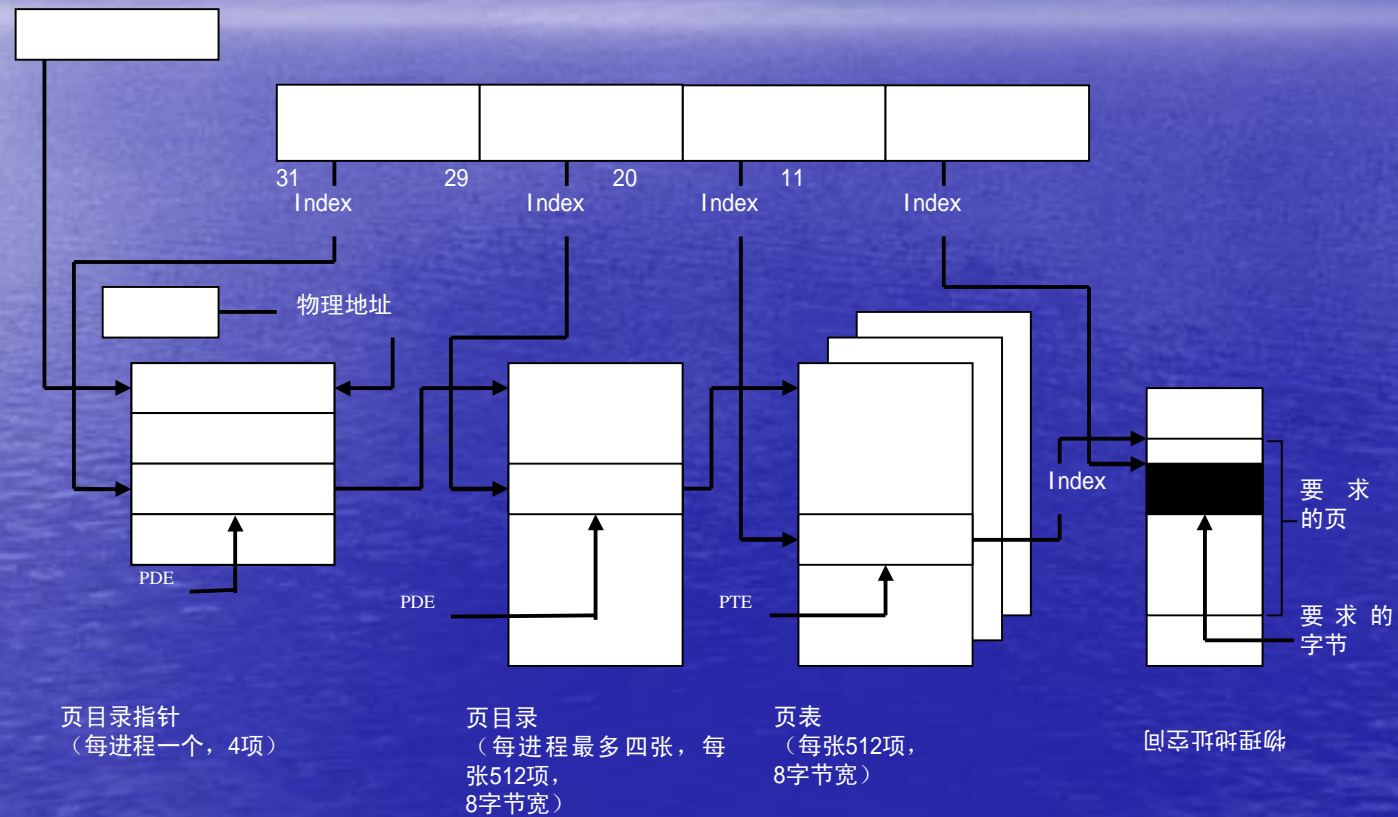
第三，Windows2000/xp支持的处理器还提供了一些硬件内存保护措施（如读/写，只读等）。

最后，共享内存区域对象具有标准的Windows2000/XP存取控制表（ACL）

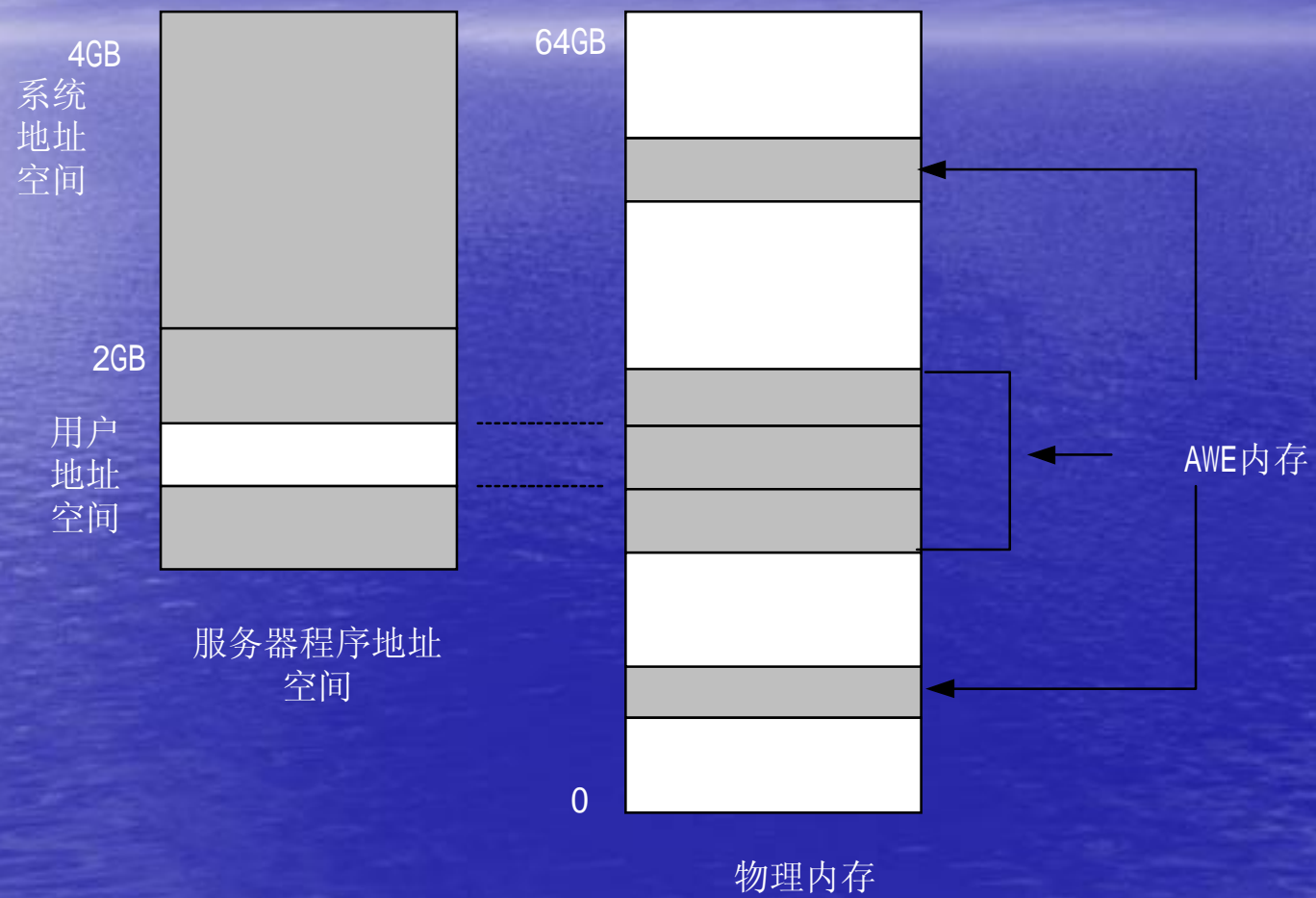
写时复制



物理地址扩展



地址窗口扩充



Windows2000/xp外存管理

- 1 Windows 2000/XP存储的演变
- 2 分区
- 3 驱动程序
- 4 动态盘管理

Windows2000/xp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）

基本术语

- 盘是一种物理存储设备。
- 扇被分为扇区，这是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

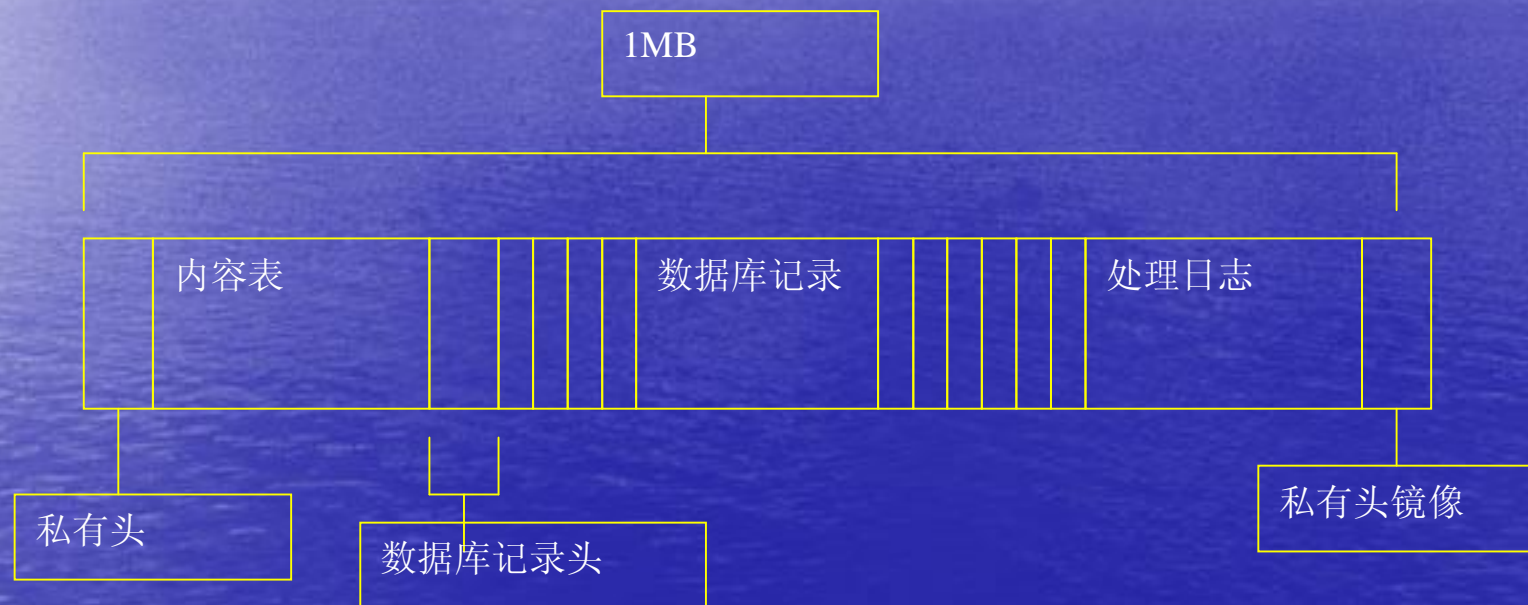
LDM

- Ø LDM的数据库存在于每个动态盘最后的1MB保留空间中。
- Ø 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。
- Ø 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



LDM的数据库



驱动程序

- 1 磁盘驱动程序
- 2 设备命名
- 3 基本盘管理
- 4 动态盘管理

多重分区管理

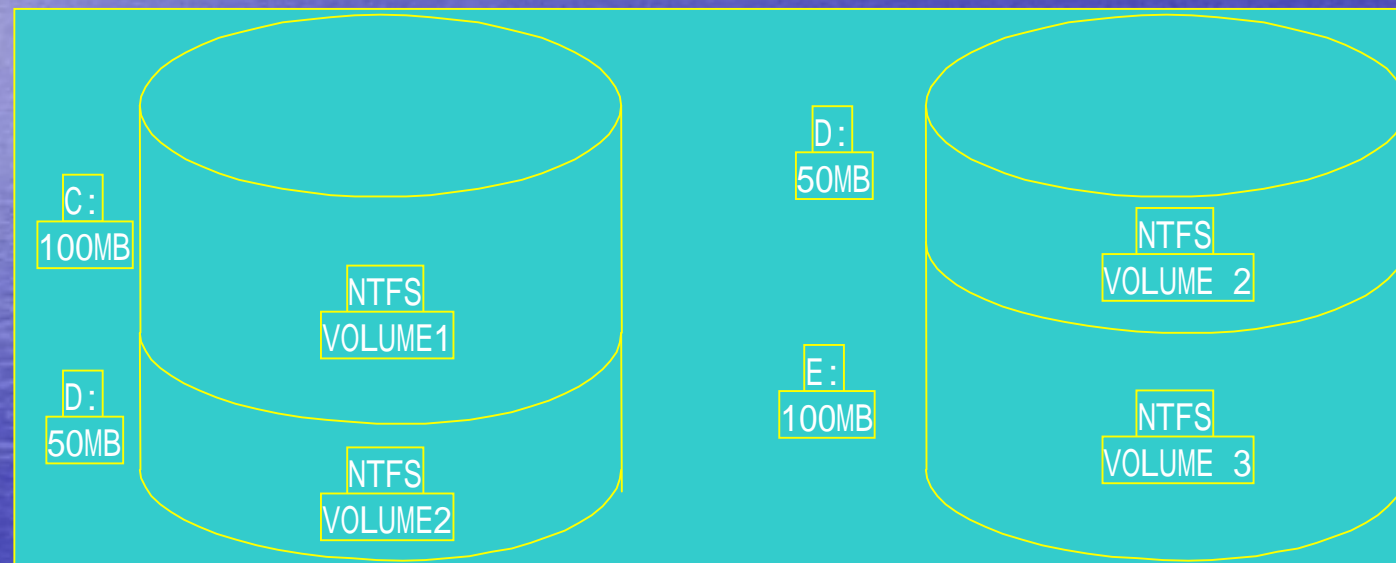
跨分区卷(spanned volume)

条带卷 (striped volume)

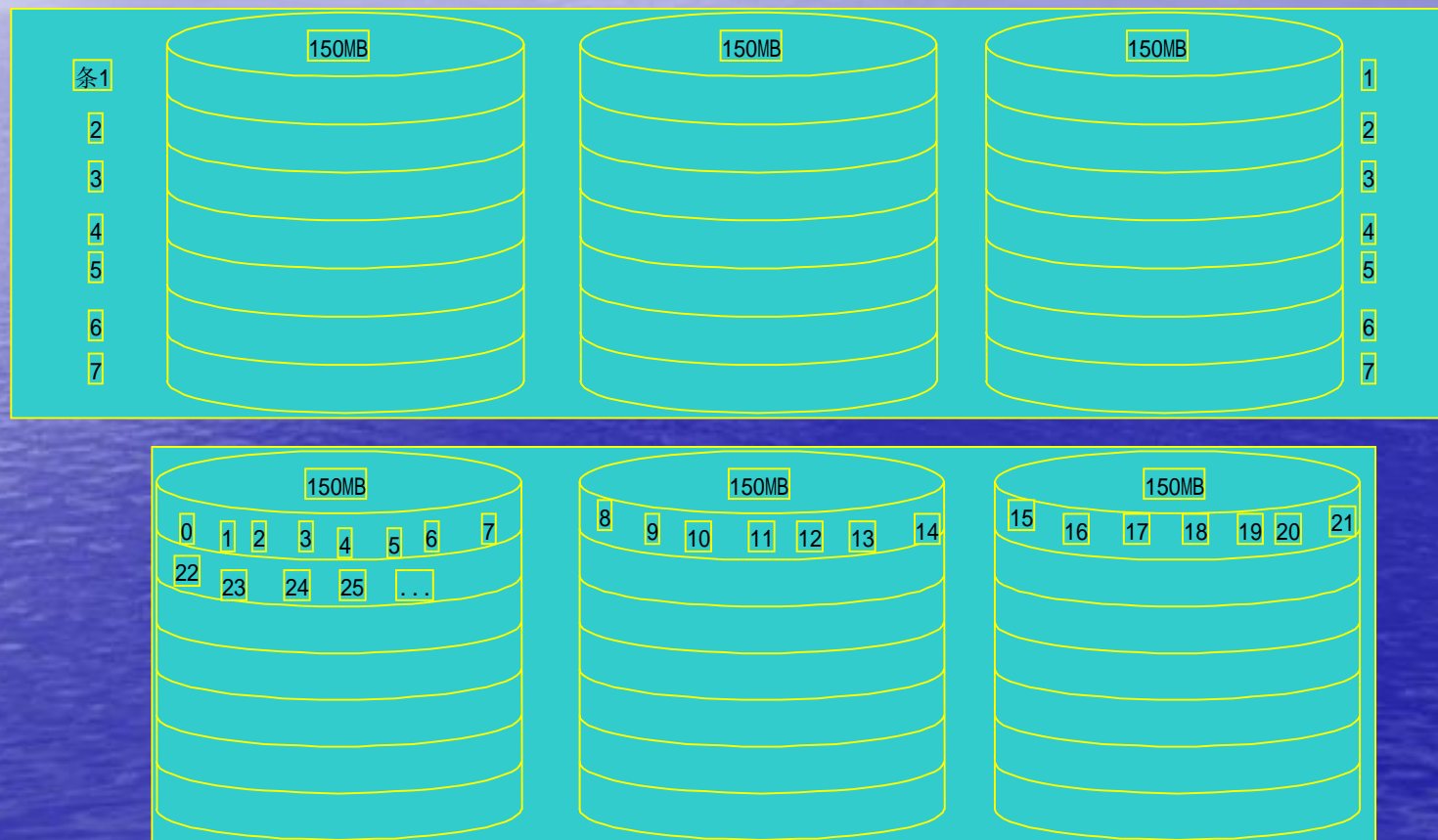
镜像卷 (mirrored volume)

廉价冗余磁盘阵列5卷 (RAID-5 volume)

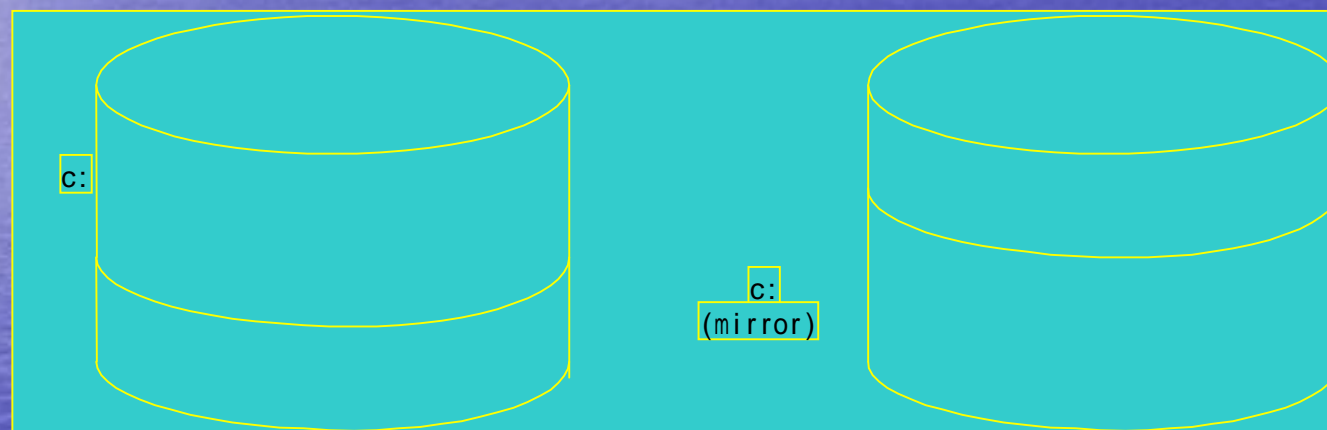
跨分区卷



条带卷 (RAID-0卷)



镜像卷

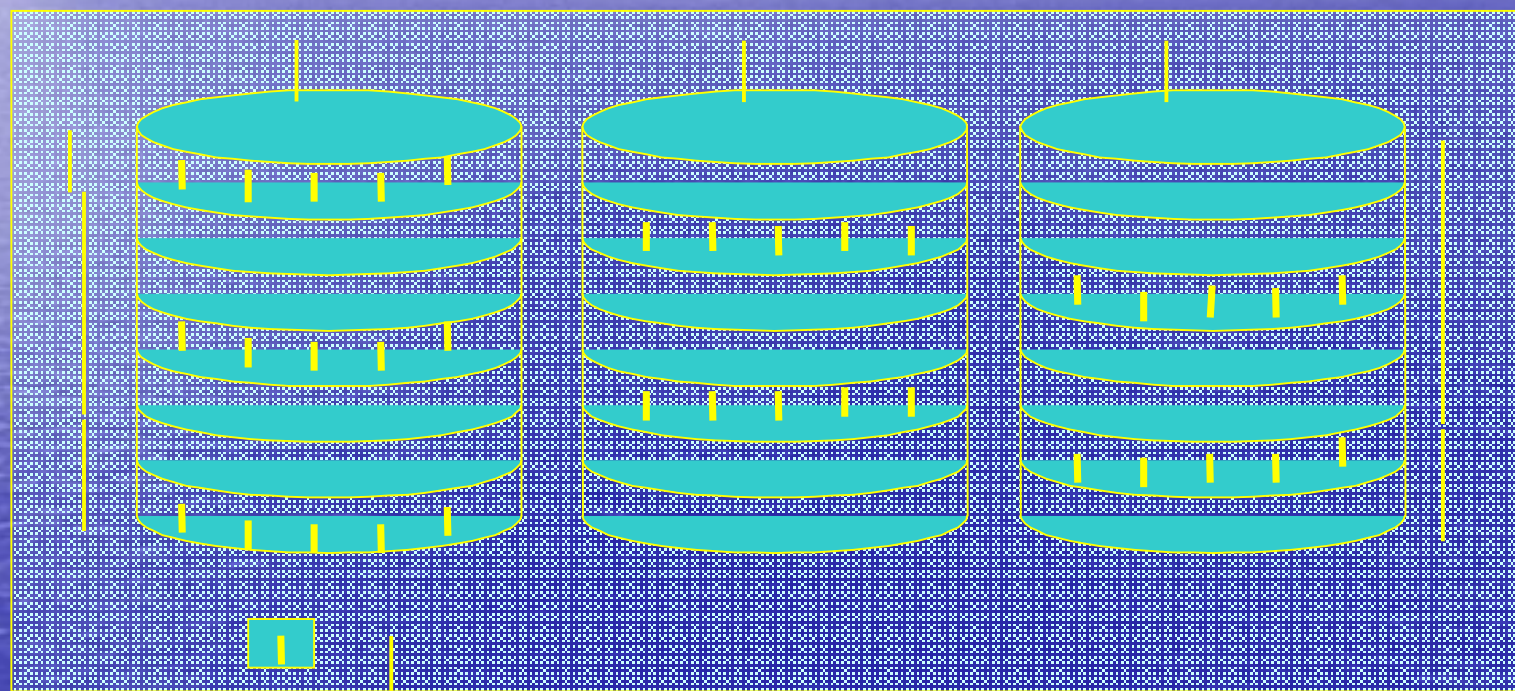


一个磁盘上分区的内容被复制另一个磁盘与它等大小的分区中。镜像卷有时也被称为RAID-1。

镜像卷能够在负载很重的系统中提高系统I/O吞吐量。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。

RAID-5卷



卷名字空间

- 1 安装管理器
- 2 安装点
- 3 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(Repase Point)技术。

C:\Project\CurrentProject\Description.txt

C:\Projects\CurrentProject\Description.txt

卷安装

Windows2000/xp高速缓存管理

1 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）

2 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

3 高速缓存的一致性



4 虚拟块缓存

Windows 2000/XP高速缓存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：

- 1) 它使智能的文件预读成为可能。
- 2) 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）。

5 基于流的缓存

6 可恢复的文件系统支持

- 1) 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。
- 2) 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。
- 3) 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。
- 4) 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。

高速缓存的结构

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

系统高速缓存

视图0

视图1

视图2

视图3

视图4

视图5

视图6

视图7

视图8

视图n

文件A(500KB)

节0

节1

文件B(750KB)

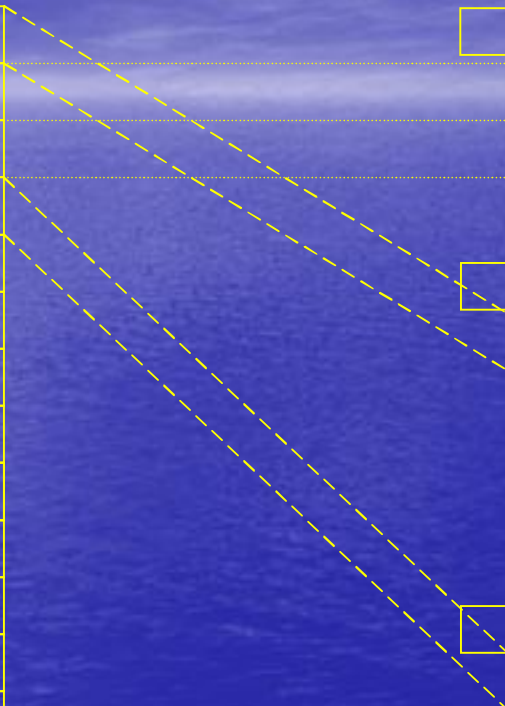
节0

节1

节2

文件C(100KB)

节0



高速缓存的大小

- 1 缓存区的虚拟大小
- 2 缓存的物理大小

高速缓存的数据结构

- 1) 在系统高速缓存的每个256 KB的槽由VACB描述。
- 2) 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。
- 3) 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。

系统范围的高速缓存数据结构

- 虚拟地址控制块 (VACB)

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

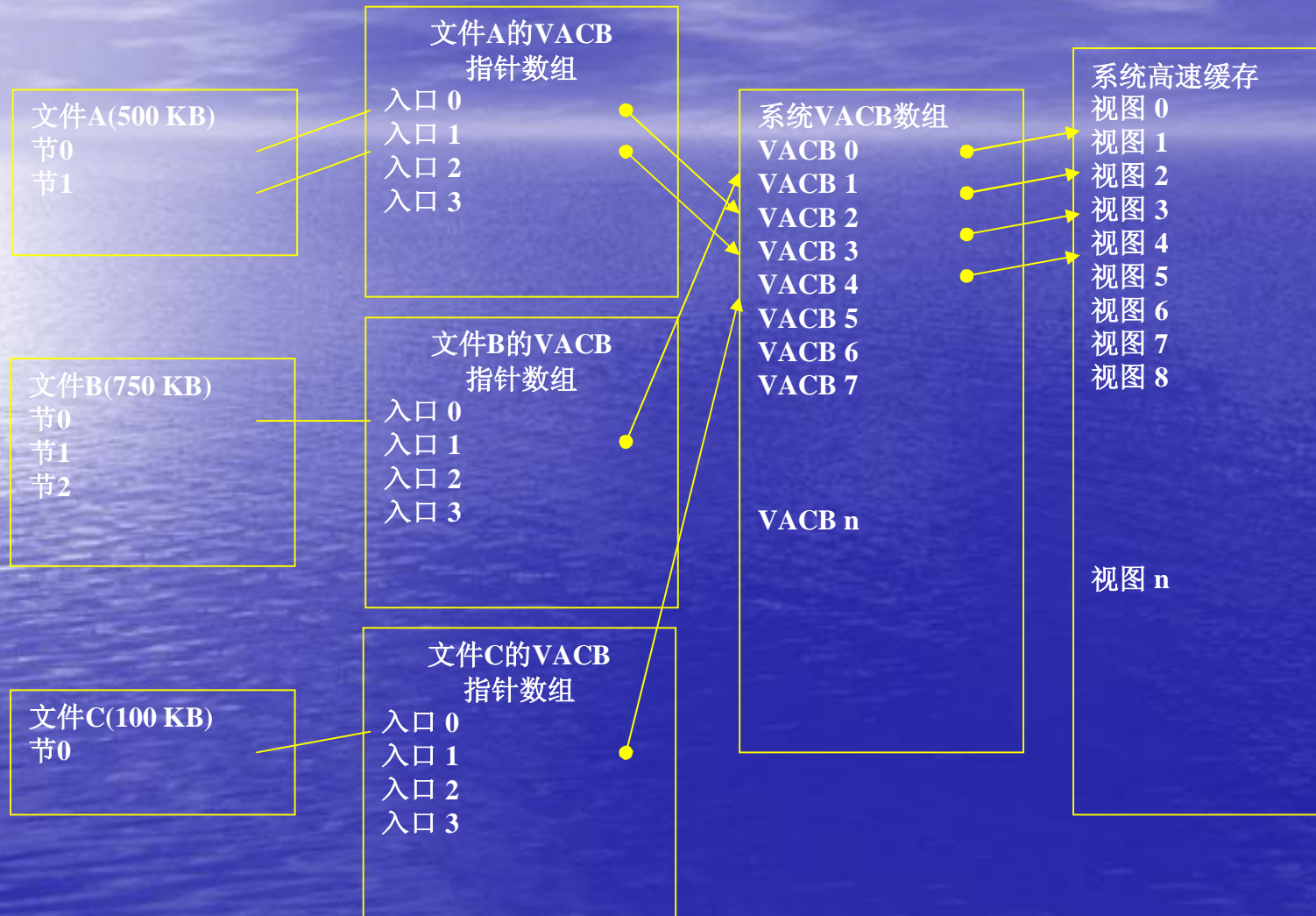
View 0
View 1
View 2
View 3
View 4
View 5
View 6
View 7
View 8
View n

系统VACB数组

- 每个缓存文件有一个共享的缓存映射结构，该结构描述了缓存文件的状态，包括它的大小和为了安全原因它的有效数据的长度。
- 有效数据长度字段的功能在“回写缓存和延迟写”部分解释。
- 共享缓存映射同时还指向由内存管理器维护的描述了文件的在虚拟地址的映射区域对象；与文件相连的私有缓存映射链表；和所有描述在系统缓存的文件的当前映射视图的所有VACB。

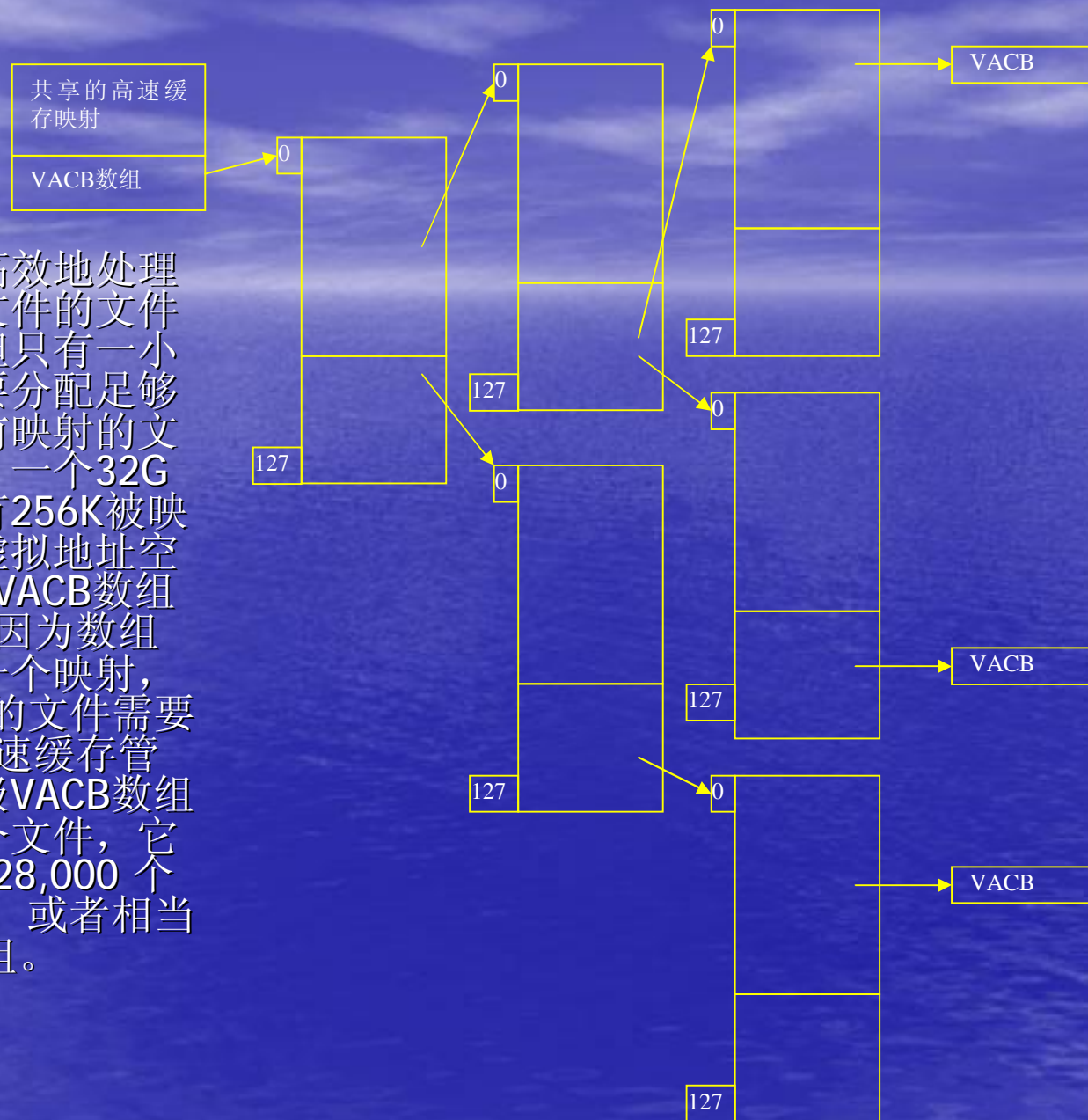
- 当请求从一个特定文件读数据，高速缓存管理器必须回答以下两个问题：
 - 1 文件在高速缓存中吗？
 - 2 如果在，哪个VACB（如果有）指向请求的地址？

单文件的缓存区数据结构



高速缓存管理器维护了一个指针数组指向VACB, 称为VACB索引数组。VACB索引数组的第一项指向文件的第一个256KB, 第二项指向文件的第二个256KB, 依此类推。来自三个不同文件的四个不同区域当前被映射到系统高速缓存。

- 这个方案是为了高效地处理稀疏文件，稀疏文件的文件大小可能很大，但只有一小块有效数据，只要分配足够的数组，处理当前映射的文件的视图。例如，一个32G的稀疏文件，只有256K被映射到高速缓存的虚拟地址空间，需要分配1个VACB数组和3个索引数组，因为数组只有一个分支有一个映射，32G（235字节）的文件需要3级数组。如果高速缓存管理器不是使用多级VACB数组进行优化，为这个文件，它需要分配一个有128,000个入口的VACB数组，或者相当于1000个索引数组。



高速缓存的操作

- 1 回写缓存和延迟写
- 2 计算脏页域值
- 3 屏蔽对文件延迟写
- 4 强制写缓存到磁盘
- 5 刷新被映射的文件

6 智能预读

7 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

8 带历史信息的异步预读

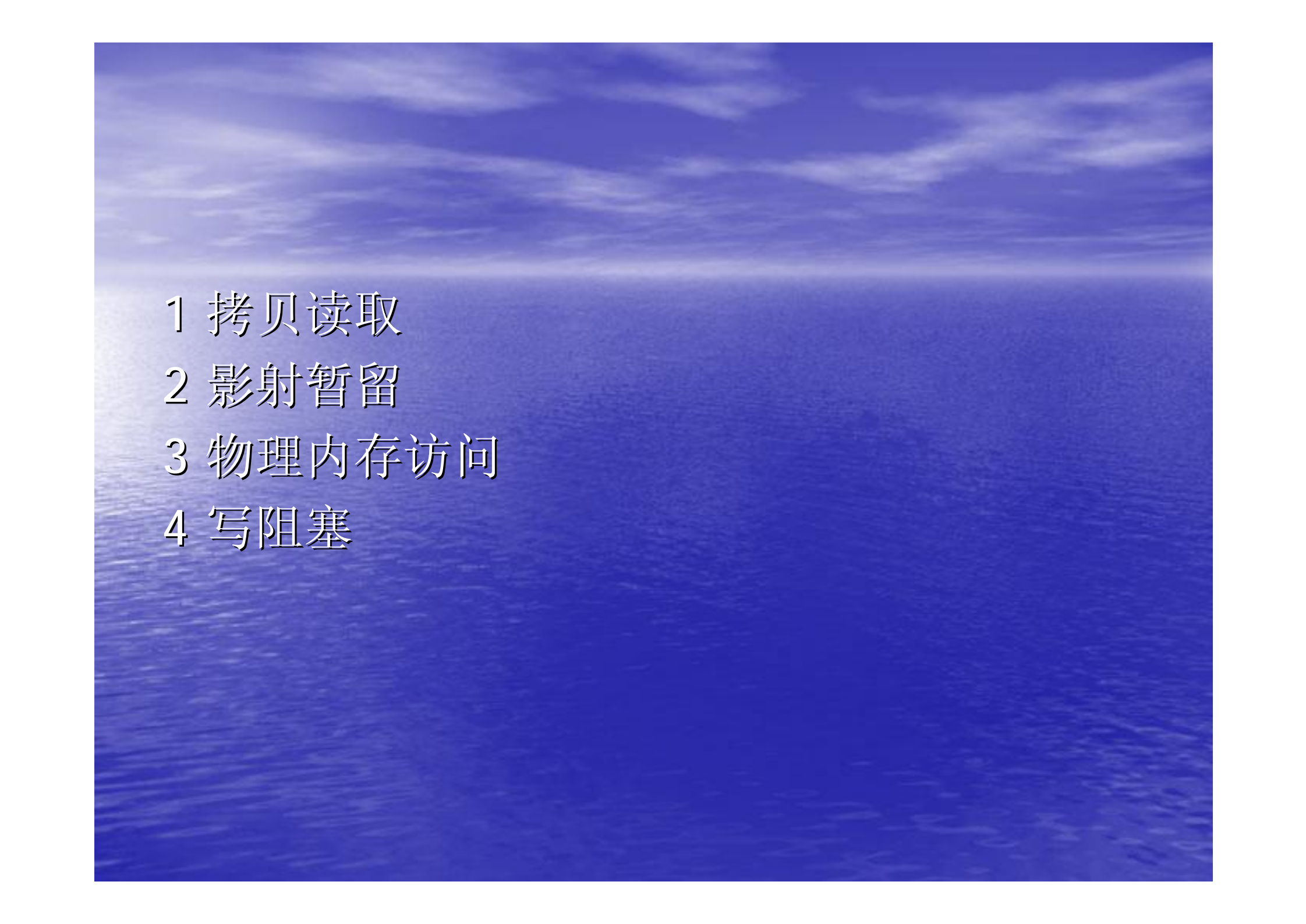
- 高速缓存管理器在文件的私有缓存映射结构中为正在被访问的文件句柄保存最后两次读请求的历史信息

9 系统线程

10 快速I/O

高速缓存支持例程

- 1 “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；
- 2 “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。
- 3 “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。

- 
- 1 拷贝读取
 - 2 影射暂留
 - 3 物理内存访问
 - 4 写阻塞