第4章　存储管理

知识要点

- 掌握
  - 存储器层次、逻辑地址空间和物理地址空间及其关系；
  - 地址重定位、存储保护机制；
  - 分区原理、交换原理、覆盖原理、对换原理；
  - 分页存储管理基本概念、实现思想及优点；
  - 分段存储管理基本概念、实现思想及优点；
  - 虚拟存储器、程序局部性原理；
  - 各种虚存管理实现思想及地址变换过程；
  - 各种分页虚存页面替换算法。
- 了解
  - Linux物理内存管理
  - Linux进程虚拟地址空间管理

存储管理的主要功能

- 主存分配和回收：
  - 主要任务：将主存分配给多个程序，以提高主存利用率。
  - 选择合适的分配和回收算法及相应的数据结构，以提高主存利用率和分配、回收的速度。
- 地址转换和重定位：
  - 主要任务：屏蔽物理内存使用细节，解决用户程序装入（可以部分装入）。
  - 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

存储管理的主要功能

- 存储保护和主存共享：
  - 解决如何在多用户和多任务环境下，实现程序代码和数据共享和保护。
  - 代码和数据共享
  - 地址空间访问权限（读、写、执行）
- 存储器扩充：
  - 解决用户对内存容量要求与内存实际容量之间的矛盾，使运行的程序不受主存大小的限制。
  - 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）

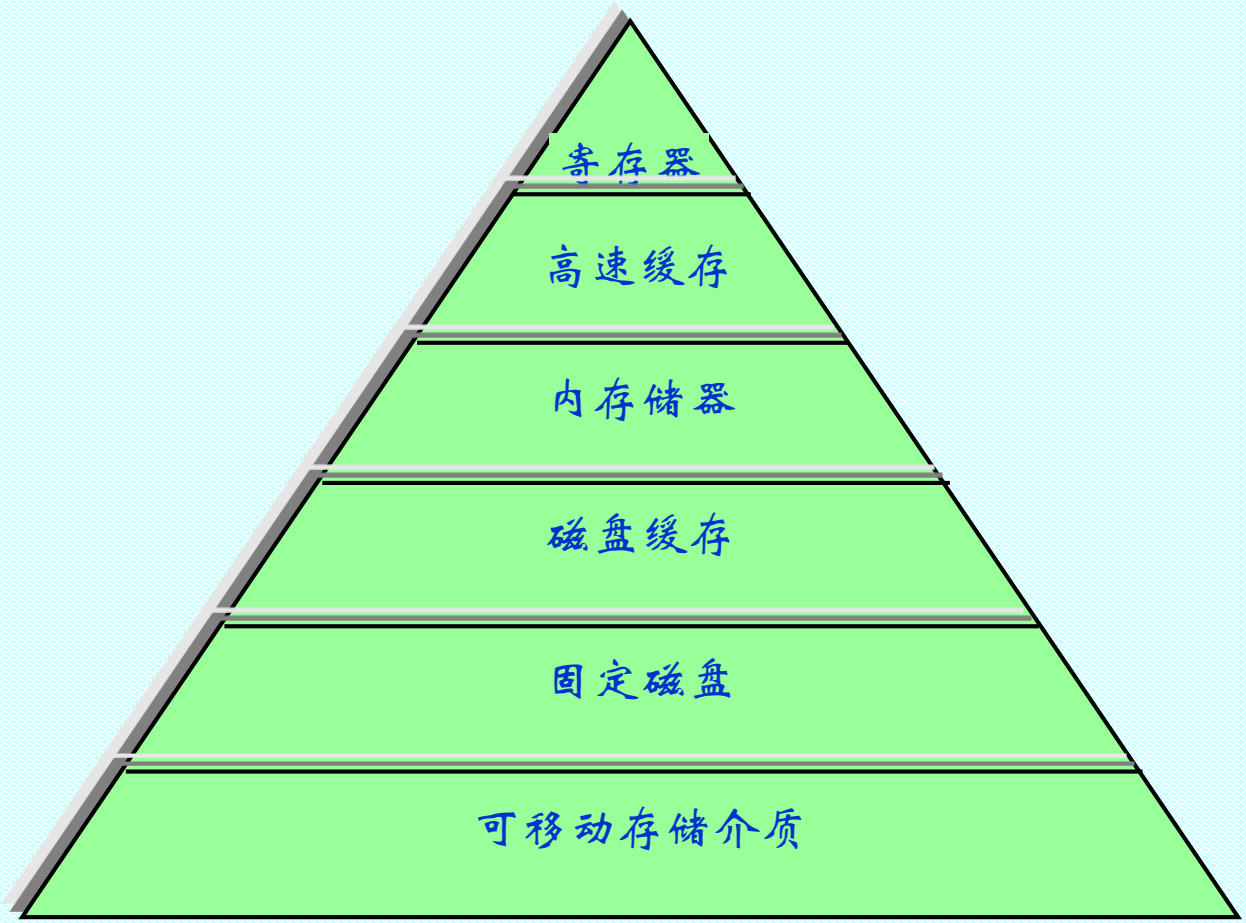
4.1 存储器工作原理

- 4.1.1存储器层次
- 4.1.2地址转换与存储保护

- 内存储器（简称内存、主存、物理存储器）
- 处理机能直接访问的存储器。用来存放系统和用户的程序和数据，其特点是存取速度快，存储方式是以新换旧，断电信息丢失。

- 外存储器（简称外存、辅助存储器）
- 处理机不能直接访问的存储器。用来存放用户的各种信息，存取速度相对内存而言要慢得多，但它可用来长期保存用户信息。在文件系统中介绍。

4.1.1 存储器层次



各级存储器性能

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

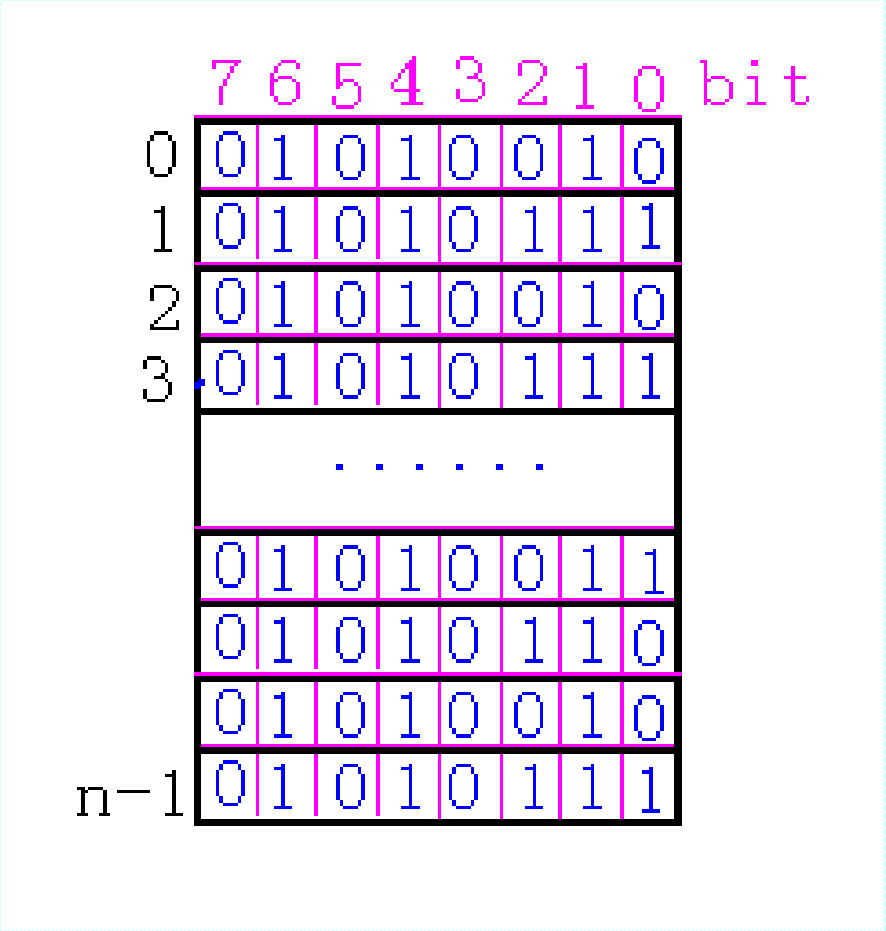
内存的物理组织

- **物理地址：**

  把内存分成若干个大小相等的存储单元，每个单元给一个编号，这个编号称为内存地址（物理地址、绝对地址、实地址），存储单元占8位，称作字节（byte）。

- **物理地址空间：**

  物理地址的集合称为物理地址空间（主存地址空间），它是一个一维的线性空间。

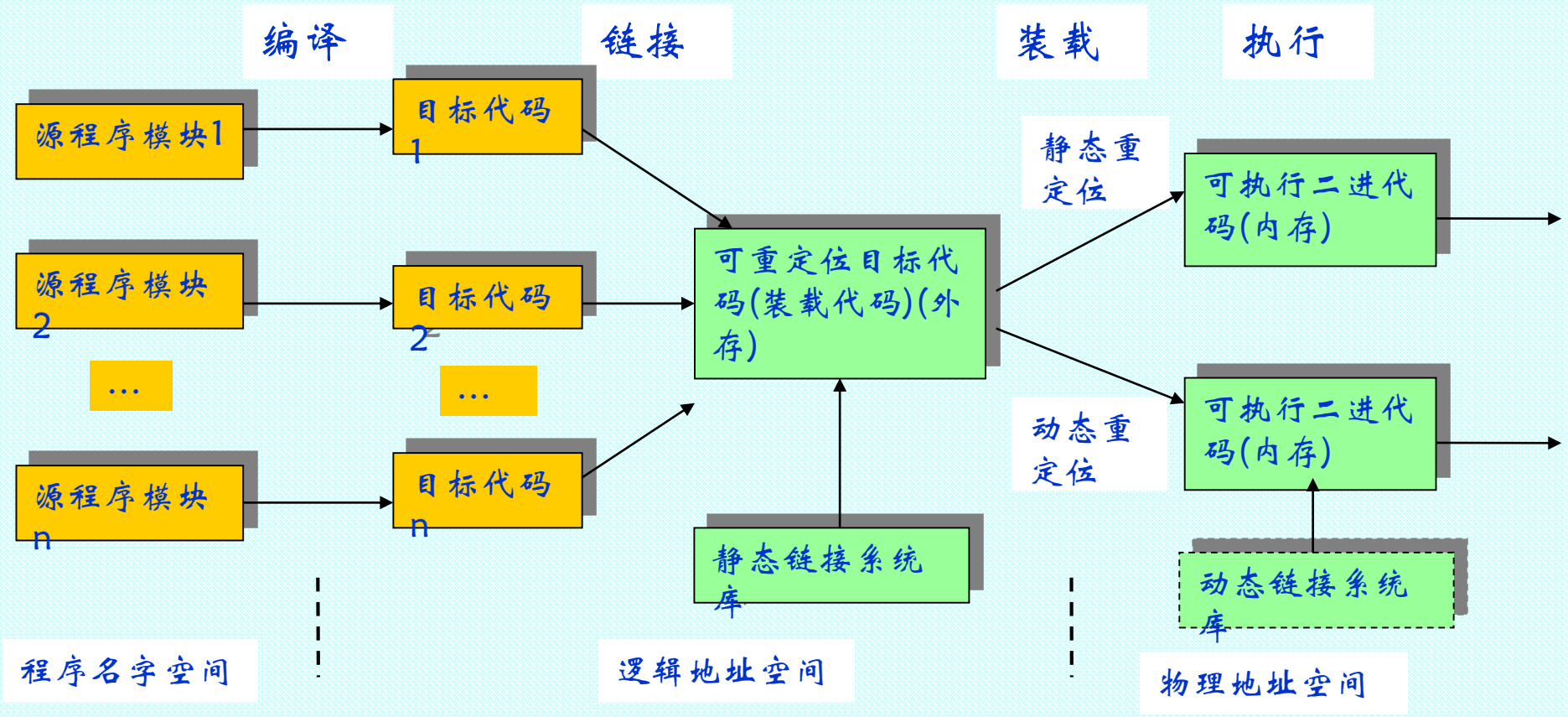


程序的逻辑结构

- 程序地址：用户编程序时所用的地址（或称逻辑地址、虚地址），基本单位可与内存的基本单位相同，也可以不相同。
- 程序地址空间（逻辑地址空间、虚地址空间）：用户的程序地址的集合称为逻辑地址空间，它的编址总是从0开始的，可以是一维线性空间，也可以是多维空间。

- 为什么程序使用逻辑地址而不是物理地址？
  - 用户需要精确计算空间与存放地址；
  - 支持多道程序运行十分困难；
  - 程序的可移植性差。

4.1.2 地址转换与存储保护(1)



程序的编译、链接、装载和执行

1. 编译、链接、装载(1)

- 编译程序负责记录引用发生的位置，编译或汇编的结果产生相应的多个目标代码模块，每个都附有供引用使用的内部符号表和外部符号表。符号表中依次给出每个符号名及在本目标代码模块中的名字地址，在模块被链接时进行转换。
- 链接需要解析内部和外部符号表，把对符号名字引用转换为数值引用，要转换每个涉及名字地址的程序入口点和数据引用点成为数值地址。
- 装入时根据指定的内存块首地址，再次修改和调整被装载模块中的每个逻辑地址，将逻辑地址绑定到物理地址。

1. 编译、链接、装载(2)

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

1. 编译、链接、装载(3)

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

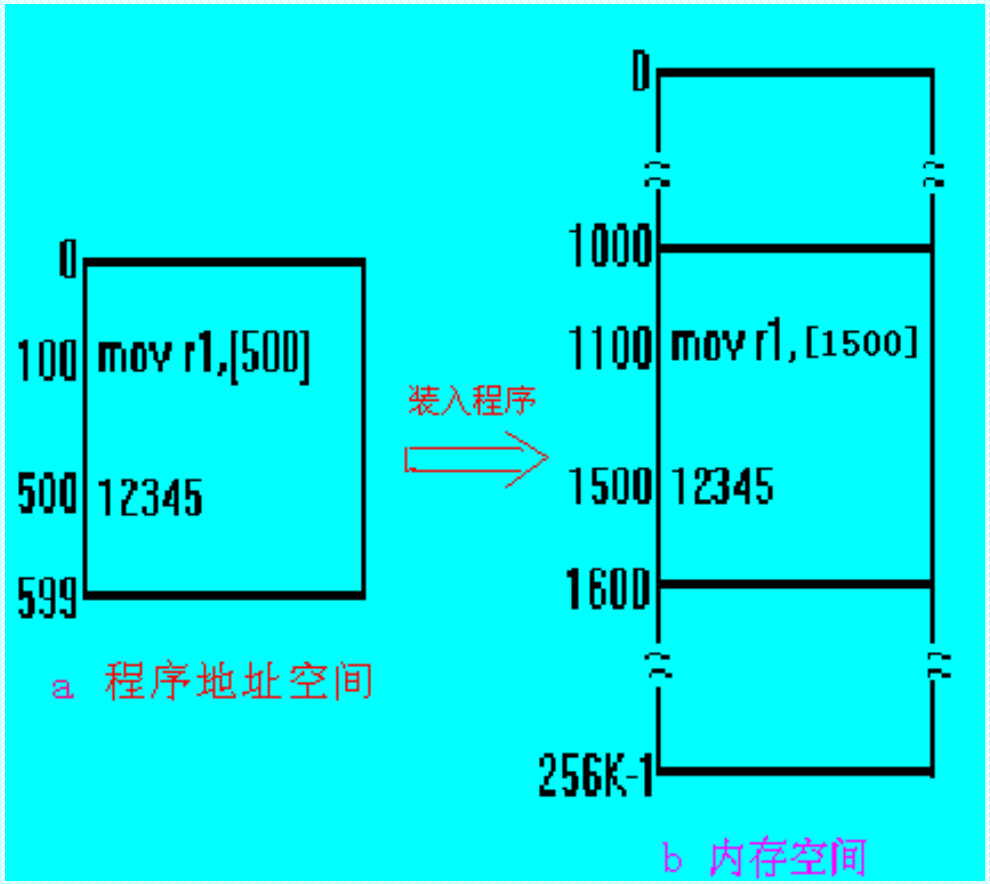
2. 地址重定位

- 可执行程序逻辑地址转换（绑定）为物理地址的过程称地址重定位、地址映射或地址转换，基于上述程序装载方式，可区分三种地址重定位。
  - (1) 静态地址重定位
  - (2) 动态地址重定位
  - (3) 运行时链接地址重定位

2. 地址重定位

- 静态重定位：程序装入内存时由连接装入程序完成从逻辑地址到物理地址的转换。
- 在一些早期的系统中都有一个装入程序（加载程序），它负责将用户程序装入系统，并将用户程序中使用的访问内存的逻辑地址转换成物理地址。
- 评价:
  - 优点是实现简单，不要硬件的支持。
  - 缺点是程序一旦装入内存，移动就比较困难。有时间上的浪费。在程序装入内存时要将所有访问内存的地址转换成物理地址。

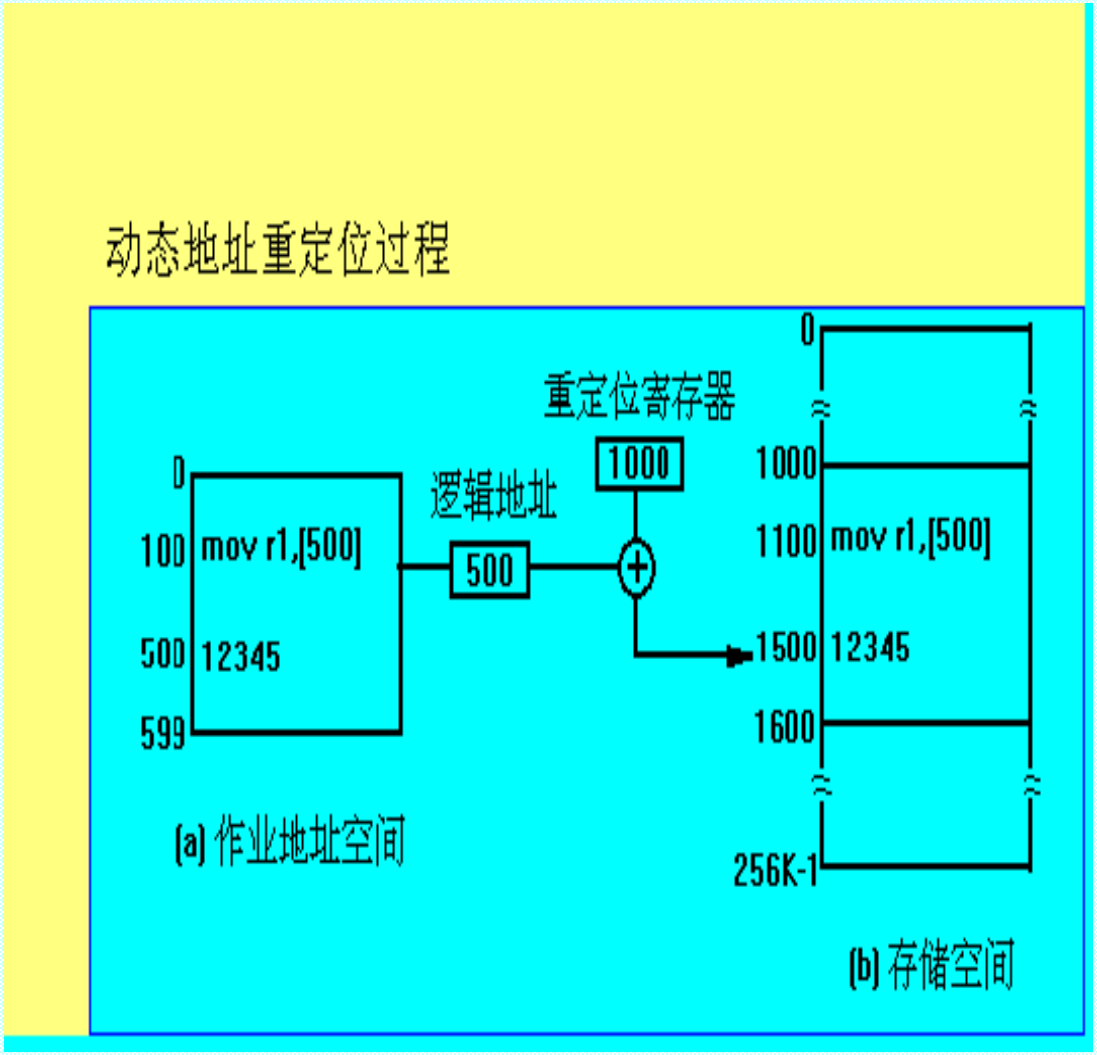
2. 地址重定位


0
100
mov r1,[500]
500
12345
599
a 程序地址空间
装入程序
0
1000
1100
mov r1,[1500]
1500
12345
1600
256K-1
b 内存空间

2. 地址重定位

- 动态重定位：在程序执行时由系统硬件完成从逻辑地址到物理地址的转换。
- 系统中设置了重定位寄存器。
- 动态重定位是由硬件地执行时完成的，程序中不执行的程序就不做地址映射的工作，这样节省了CPU的时间 。
- 重定位寄存器的内容由操作系统用特权指令来设置，比较灵活。
- 实现动态地址映射必须有硬件的支持，并有一定的执行时间延迟。现代计算机系统中都采用动态地址映射技术。

2. 地址重定位


动态地址重定位过程
重定位寄存器
1000
逻辑地址
500
0
100 mov r1,[500]
500 12345
599
(a) 作业地址空间
0
1000
1100 mov r1,[500]
1500 12345
1600
256K-1
(b) 存储空间

2. 地址重定位

- 动态地址映射技术能满足以下目标：
  - 具有给一个用户程序任意分配内存区的能力；
  - 可实现虚拟存储；
  - 具有重新分配的能力；
  - 对于一个用户程序，可以分配到多个不同的存储区。

3. 存储保护

- 涉及：防止地址越界和控制正确存取。
- 地址越界保护：
  - 各道程序只能访问自己的内存区而不能互相干扰，必须对内存中的程序和数据进行保护，以免受到其他程序有意或无意的破坏。
  - 对进程执行时所产生的所有内存访问地址进行检查，确保进程仅访问它自己的内存区，就是地址越界保护。
  - 越界保护依赖于硬件设施，常用的有：界地址和存储键
- 信息存取保护：
  - 进程访问分配给自己的内存区时，要对访问权限进行检查，如允许读、写、执行等，从而确保数据的安全性和完整性，防止有意或无意的误操作而破坏内存信息。

3. 存储保护

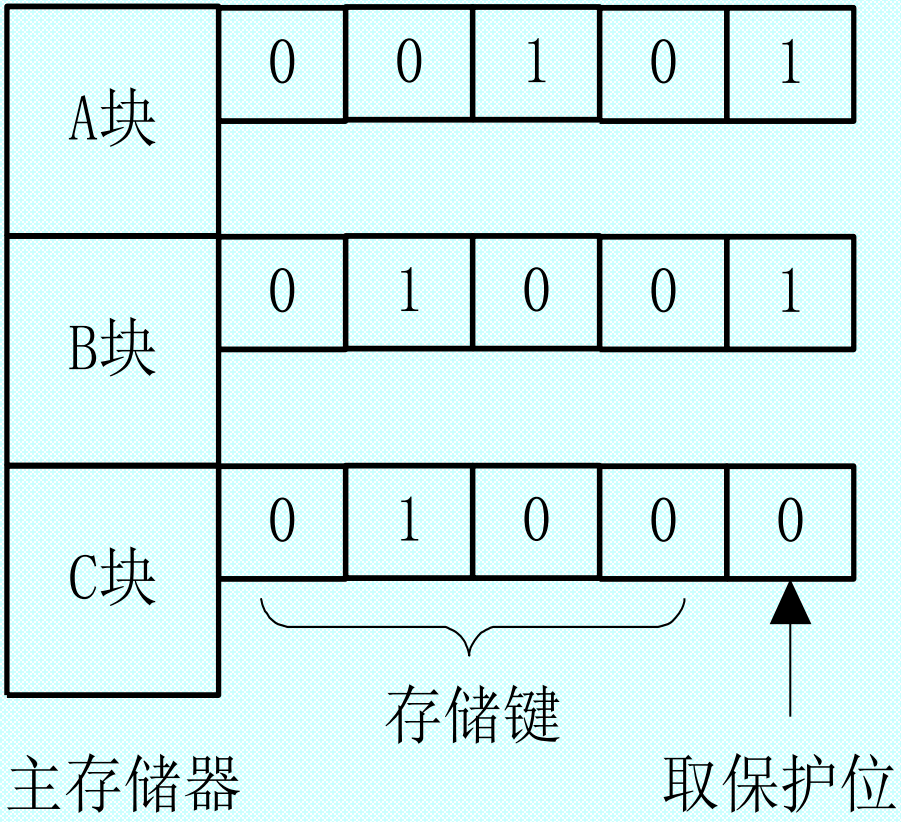
- 界地址寄存器（界限寄存器）：

  在CPU 中设置一对界限寄存器来存放该用户作业在主存中的下限和上限地址，分别称为下限寄存器和上限寄存器。

- 存储保护键：

  每个存储块都有一个存储保护键，附加在每个存储块上。当操作系统挑选作业运行时，操作系统同时将该作业的存储键号存放到程序状态字PSW的存储键(“钥匙”)域中。每当CPU访问主存时，都将对主存块的存储键与PSW中的“钥匙”进行比较。以判断访问是否合法。

3. 存储保护



4.2 连续存储空间管理

- 4.2.1 固定分区存储管理
- 4.2.2 可变分区存储管理
- 4.2.3 内存不足的存储管理技术

4.2.1 固定分区存储管理

1、基本概念：

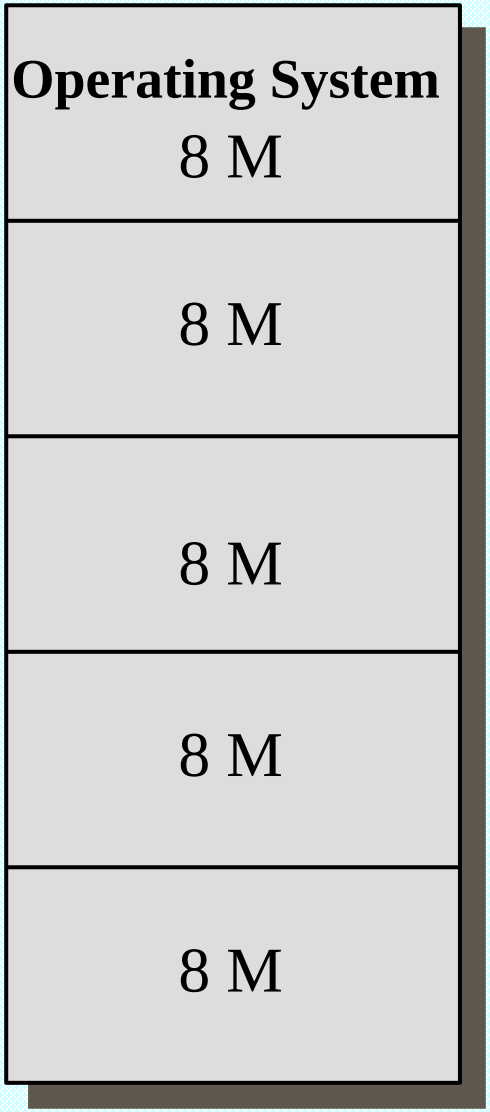
把主存分成若干个固定大小的存储区，每个分区给一个作业使用，直到该作业完成后才将该区归还系统。

- 固定指各分区的位置和大小固定。通常在系统启动后就确定了。
- 分区可分为用户分区和系统分区，用户分区存放用户程序，系统分区存放系统程序和管理信息。

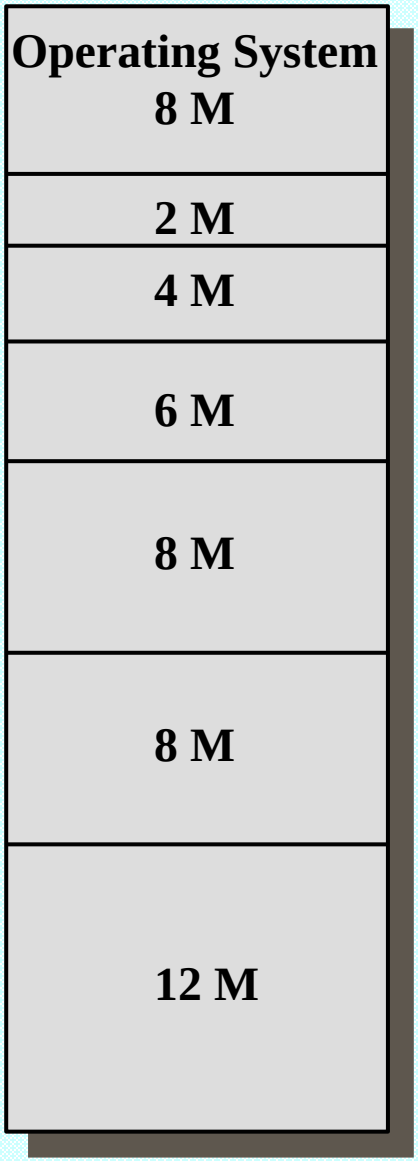
4.2.1 固定分区存储管理

2、用户分区的划分可用两种方式

- 分区大小相等：指所有的用户分区大小都相等。
  - 缺点：
    - 程序小于分区大小，可能出现内部碎片，造成主存浪费
    - 程序大于分区，程序无法在一个分区内装入，导致程序无法运行。
- 分区大小不等：指所有用户分区的大小并不都相等
  - 克服分区大小相等的缺点，一般划分出多个较小的分区、适量中等分区和少量大分区。小程序分配小分区。



固定分区(大小相同)



固定分区(多种大小)

4.2.1 固定分区存储管理

3、存储分块表（MBT）

- 当分区大小不等时，系统需要对每个分区的信息进行记录，以便管理。
- 用来存储分区管理信息的数据基。
- MBT中一般记录三项信息
  - 大小：存储块的大小，以字节为单位
  - 位置：存储块在主存中的起始地址
  - 状态：存储块是否使用标记

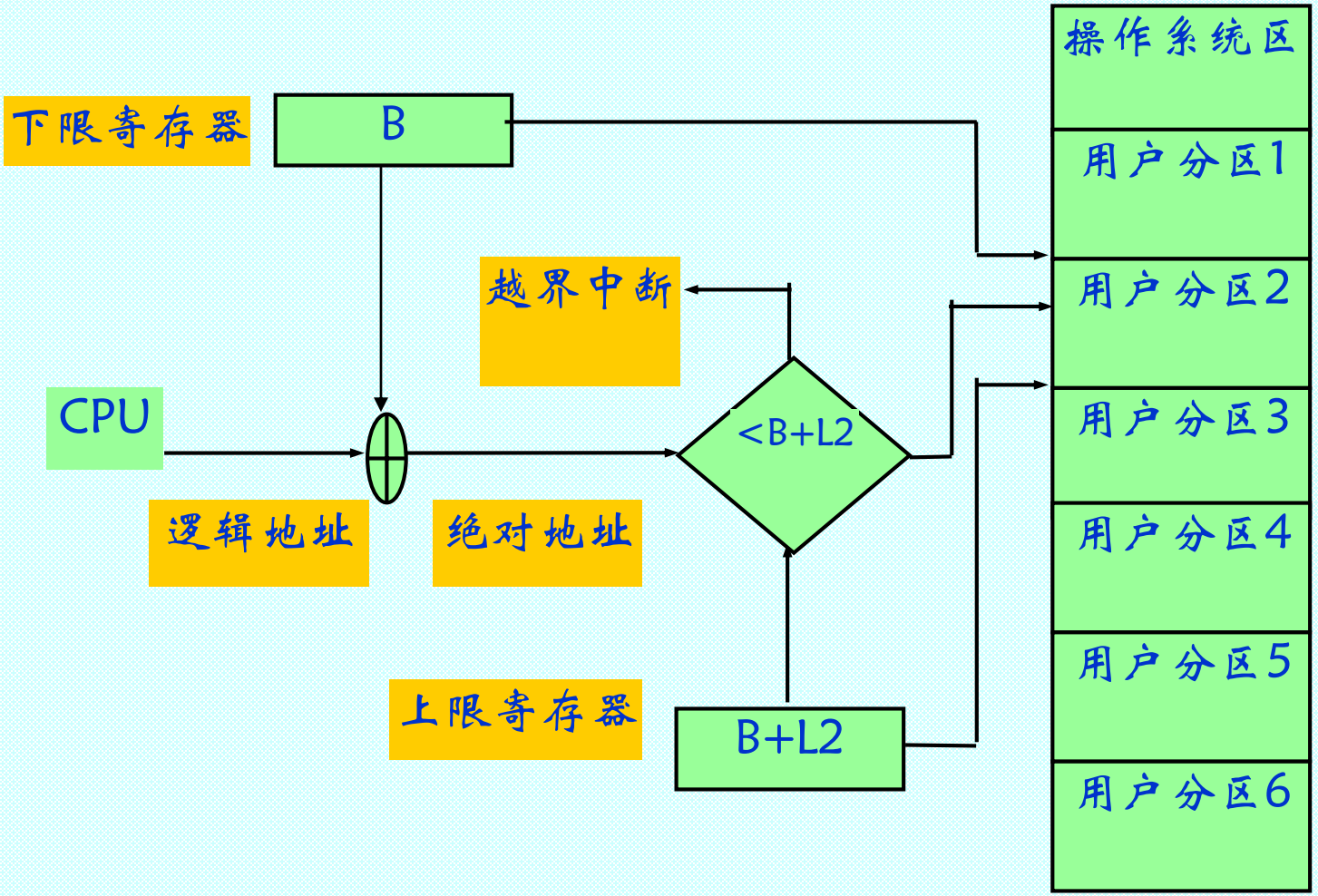
4.2.1 固定分区存储管理

- MBT一般放在系统分区内，通常由存储分配和释放两个模块对它进行操作。
- MBT在系统分区占用一个连续的内存空间

大小	位置	状态
8K	**300K**	正使用
8K	**308K**	未使用
16K	**316K**	正使用
16K	**332K**	正使用
32K	**348K**	未使用
128K	**380K**	正使用

4.2.1 固定分区存储管理

地址转换和存储保护



4.2.1 固定分区存储管理

- 优点
  - 管理简单；
  - 硬件支持要求少，一对界地址寄存器；
- 缺点
  - 主存利用率不高，存在内部碎片。
  - 分区总数固定，限制了并发执行的程序数目。
- 采用静态重定位。
- 可以采用一对界地址寄存器实现储存器保护。

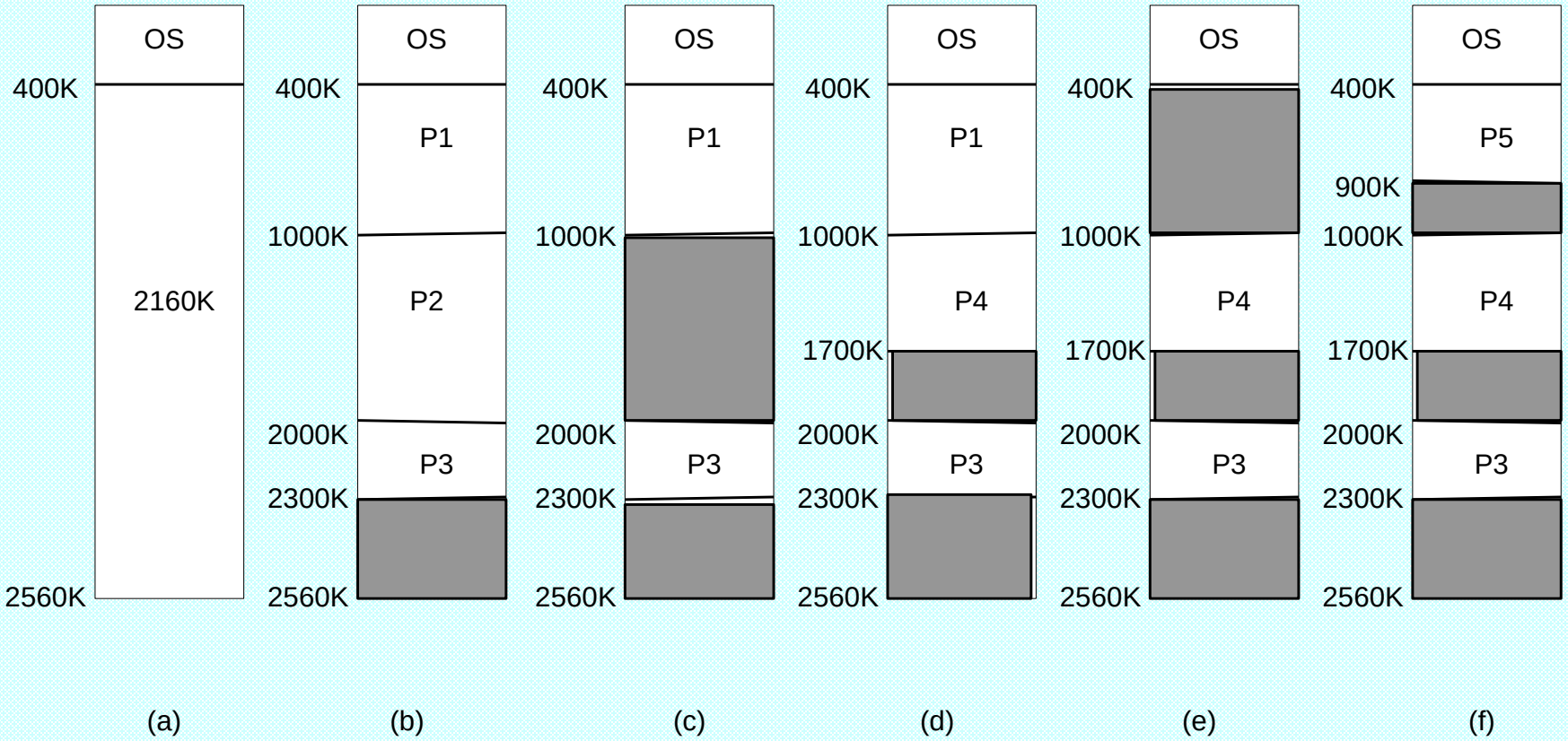
4.2.2 可变分区存储管理

- 起因：固定分区主存利用率不高，使用不灵活。
- 定义：指事先并未将主存划分为一块块分区，而是在作业进入主存时，按作业的大小动态地建立分区，且分区个数也是随机的，实现多个作业对内存的共享，进一步提高内存资源利用率。

工作过程

例子：计算机系统有2560KB主存，按照可变分区方式，系统首先为OS分配一个系统分区，剩余的作为一个整的分区作为用户分区。OS需要400KB，则用户区为2160KB。系统启动后，其主存分配图（a），此时有5个作业依次进入内存，其内存要求和进入时间如表：

进程	主存	时间
P1	**600KB**	**10**
P2	**1000KB**	**5**
P3	**300KB**	**20**
P4	**700KB**	**8**
P5	**500KB**	**15**


OS
400K
2160K
2560K
(a)
OS
400K
P1
1000K
P2
2000K
P3
2300K
2560K
(b)
OS
400K
P1
1000K
2000K
P3
2300K
2560K
(c)
OS
400K
P1
1000K
P4
1700K
2000K
P3
2300K
2560K
(d)
OS
400K
1000K
P4
1700K
2000K
P3
2300K
2560K
(e)
OS
400K
P5
900K
1000K
P4
1700K
2000K
P3
2300K
2560K
(f)

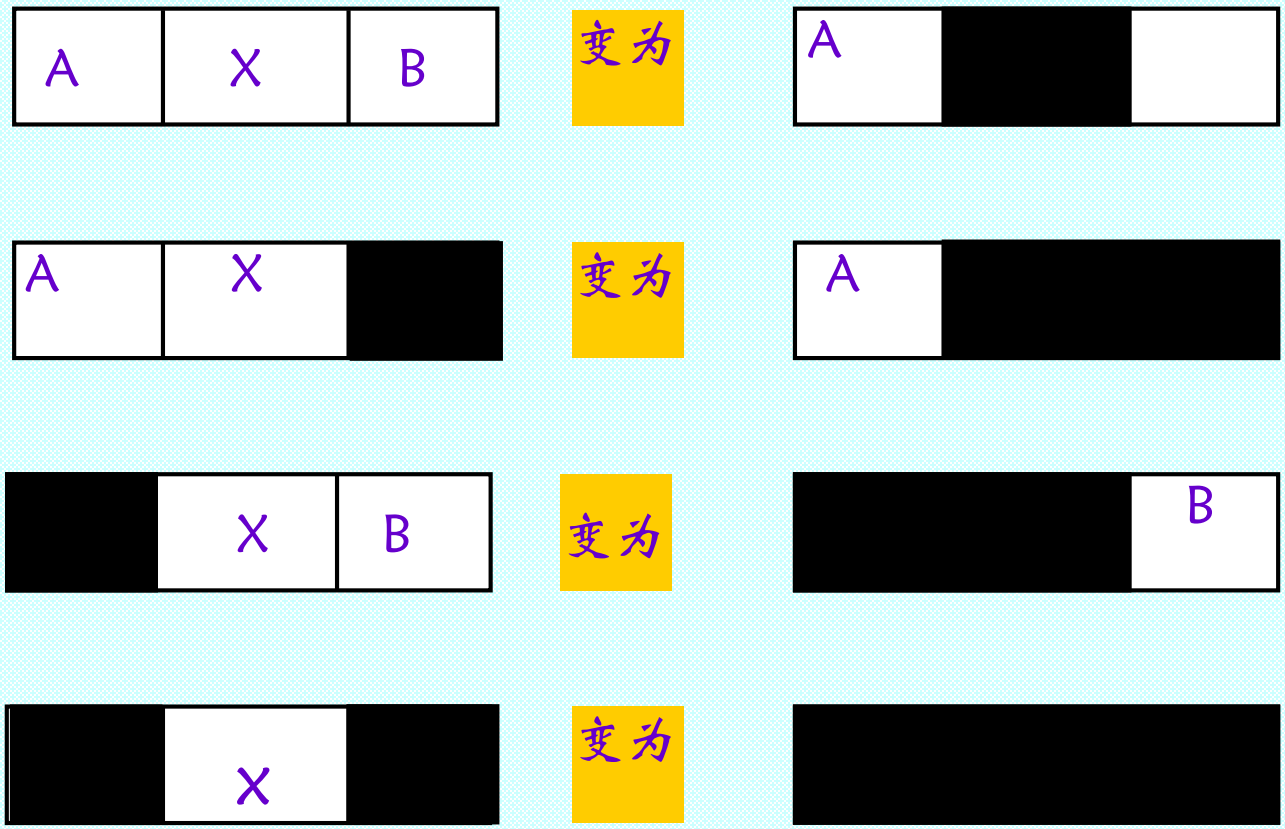
4.2.2 可变分区存储管理

- 由于作业的大小以及进入主存的时间不同。形成以下特点：
  - 分区个数可变，分区大小不固定。
  - 主存中分布着个数和大小都是变化的自由分区。
- 必须解决的问题
  - 记录分区信息的数据结构
  - 分配算法
  - 分配和回收操作

可变分区存储管理数据结构

- 固定分区的MBT结构应用于可变分区有以下缺点：
  - 表长难确定，由于分区个数变化，因此MBT表项也需变化；
  - 查找速度慢，由于空闲分区在表中一般没有按大小排序，查找一个可供分配的分区需要察看更多的表项。
- 可变分区内存分配表可由两张表格组成：
  - “已分配区表”：存放已在分配使用的分区信息。
  - “未分配区表”：存放空闲、尚未分配使用的分区信息。
- 分配内存时只需查找FBT，提高了内存分配速度。

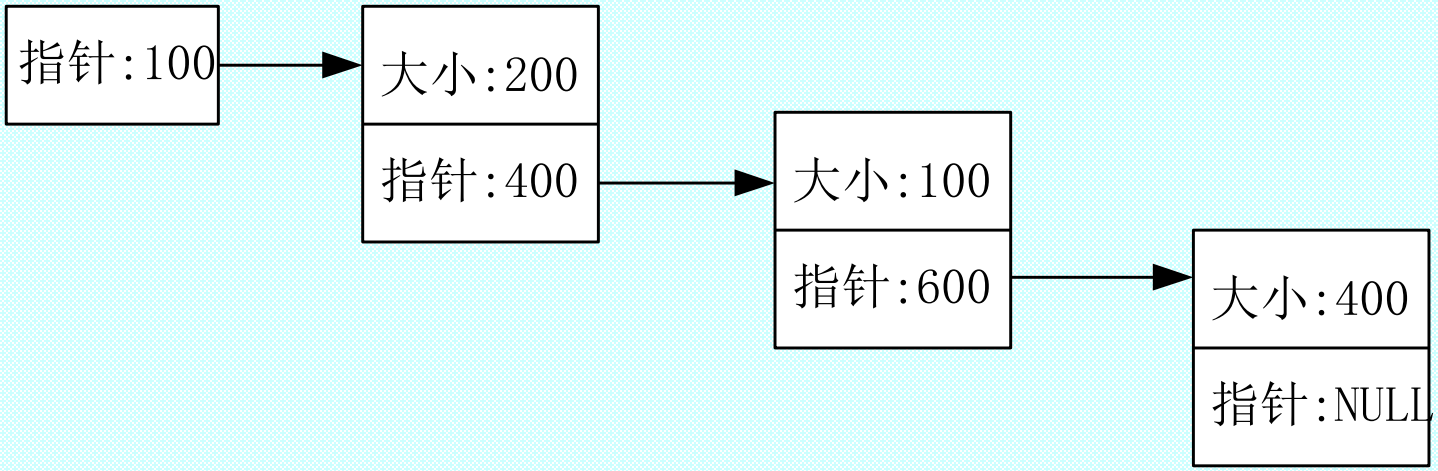
可变分区回收算法



链表空闲区管理方法

- 原因：以上两种方式采用表格方式，表长难以确定的问题仍未解决。
- 定义：采用链指针方式将空闲分区块链结在一起。
- 实现方法：空闲区开头单元存放本空闲区长度及下个空闲区起始地址，把所有空闲区都链接起来，设置第一块空闲区地址指针，让它指向第一块空闲区地址。
- 已分存储块的管理：由于存储块分配给作业或进程后，存储块信息（大小和起始位置）在JCB或PCB中有记录，无需链表来管理。

链表空闲区管理方法



可变分区管理分配算法

- 最先适应分配算法
- 下次适应分配算法
- 最优适应分配算法
- 最坏适应分配算法
- 快速适应分配算法

最优适应法

- 定义：按分区的在内存的次序从头查找，找到其大小与要求相差最小的满足要求的空闲分区进行分配。
- 思想：避免“大材小用”，使分区内未用部分最少。
- 为了便于查找，一般对空闲存储块由小到大顺序排列，这样，第一次找到的满足要求的空闲块就是最佳的空闲块。
- 缺点：孤立地看，该方法似乎是最优的，然而，从宏观和长远看，由于每次剩余的部分重是最小的，这样，在主存中会留下许多难以利用的小空闲区（外部碎片）。
- 优点：较大的空闲分区可以被保留。

适应分配法例子

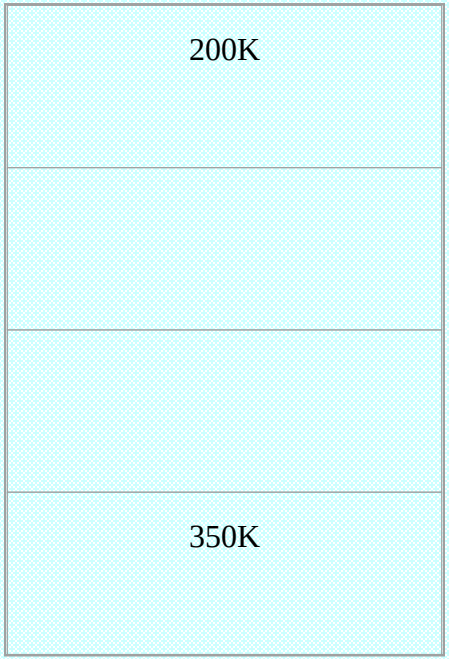
- 以上分配方法各有其有优缺点，不同情况下，不同的结果 。

例：某一时刻，内存分布如右图，有进程P1(190K)，P2（300K），P3(20K)。

问：下列情况下，采用哪种方式可使所有进程装入内存？

(a)：进入内存次序为P1,P2,P3。

(b):进入内存次序为P3,P2,P1。



适应分配法例子

- (a)：进入内存次序为P1,P2,P3时，最佳、最先可以；最坏不可以。
- (b)：进入内存次序为P3,P2,P1时，最坏可以；最佳、最先不可以。

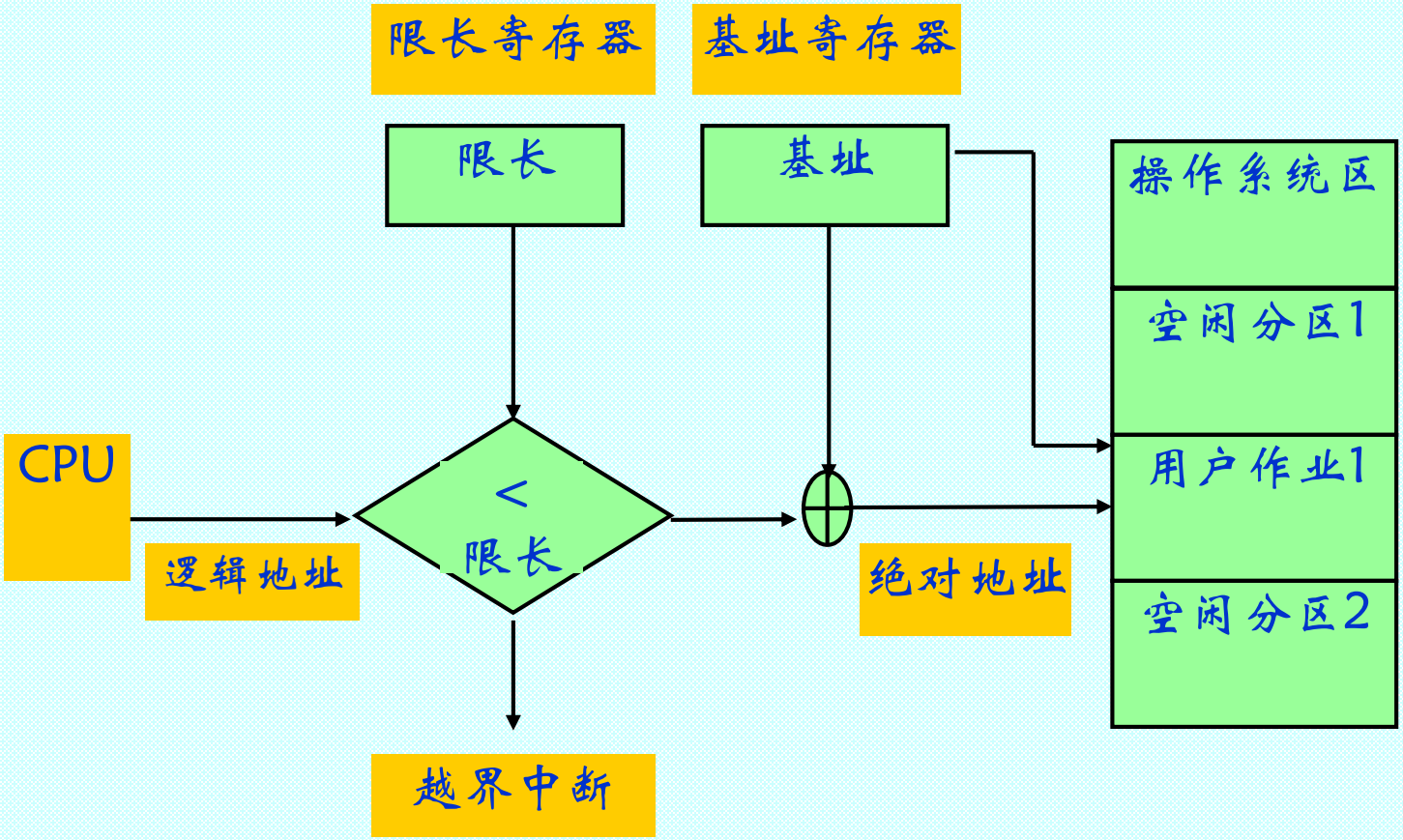
最先适应法

- 定义：按分区在内存的先后次序从头查找，找到符合要求的第一个分区进行分配。
- 分析：由于分区序号通常由低向高排列，因此，该算法倾向于优先利用主存的低地址部分的空闲分区，高地址的空闲分区很少被利用，可以保留高端大空闲区。
- 一般要求对空闲分区按地址递增的次序排列。
- 优点：该算法的分配和回收的时间性能较好，可以保留高端大空闲区。
- 缺点：随着低端分区不断划分，低端会出现很多较小的空闲分区，由于分配查找从低端开始，因此查找时间开销会增大。

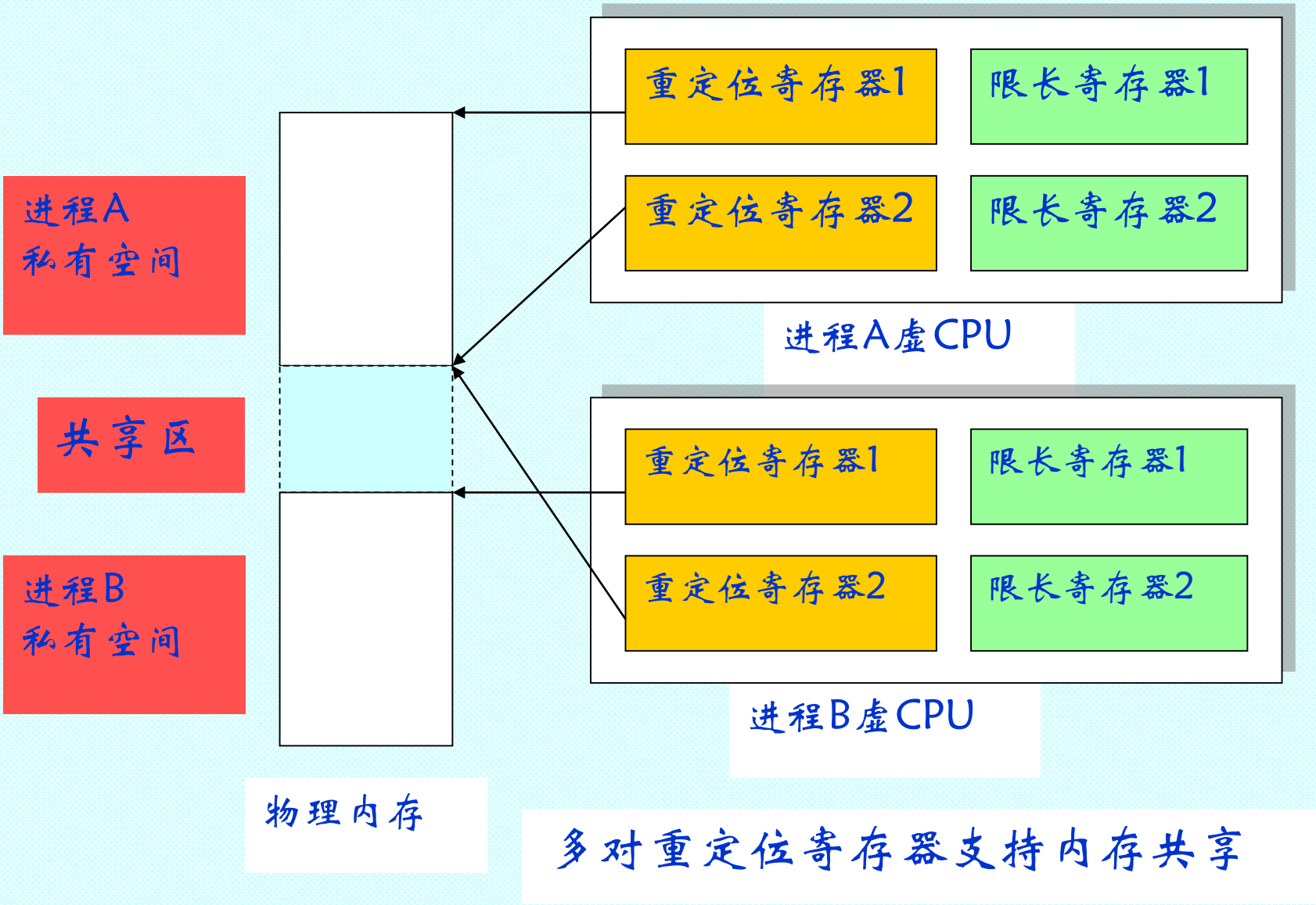
最坏适应法

- 定义：按分区在内存的先后次序从头查找，找到最大的满足要求的空闲分区进行分配。
- 一般要求对空闲存储块按其大小以递增顺序排列。
- 优点：减少了最佳适应法中出现过多外部碎片的缺点。
- 缺点：不利于大作业的分配。

可变分区地址转换与存储保护



多对基址/限长寄存器



多对重定位寄存器支持内存共享

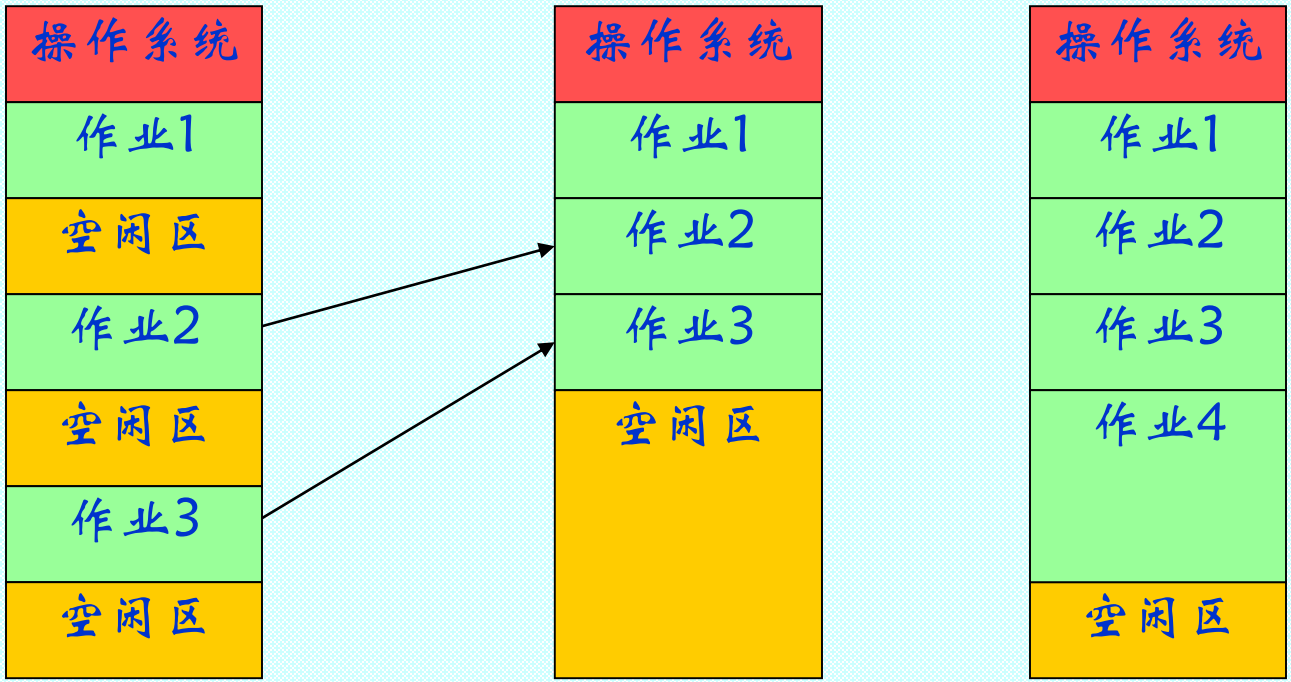
4.2.3内存不足的存储管理技术

- 移动技术
- 对换技术
- 覆盖技术

1.移动技术（1）

- 起因：在可变分区中，会出现大量小的空闲分区，即使这些分区的总容量大于一个用户程序的要求，由于地址离散，而不能为程序所用，形成外部碎片，造成内存的浪费。
- 措施：通过移动程序，将碎片集中起来形成一个大分区。
- 内存紧凑：也叫存储器紧缩，指在主存中把离散的碎片集中起来形成一个完整的大分区的方法。
- 程序浮动：指在主存中将用户程序移动。

1.移动技术（2）


操作系统
作业1
空闲区
作业2
空闲区
作业3
空闲区
操作系统
作业1
作业2
作业3
空闲区
操作系统
作业1
作业2
作业3
作业4
空闲区

1. 移动技术（3）

- 内存紧凑和程序浮动带来的问题：
  - 经过紧缩后，用户程序在内存中的位置发生了变化，若要程序能正确运行，必须对程序代码和数据的地址进行变换，即进行重定位。
  - 静态重定位不行，最好的方法是采用动态重定位。
  - 采用动态重定位技术，由于地址转换在程序执行期间，随着对每条指令和数据的访问而自动进行，因此，当系统进行紧缩和程序浮动时，不需要对程序做任何修改，只需将程序在主存的起始地址进行更新即可。

移动算法的实施时机

- 方式1：在某分区被释放后立即进行紧缩。
  - 优点：系统主存非常整洁，只有一个连续的空闲分区，没有任何碎片，有利于空闲分块表的管理和主存分配。
  - 缺点：紧缩工作需要耗费系统资源，会降低CPU利用率和系统吞吐量。
- 方法2：当“请求分配模块”找不到足够大的空闲分区时，再进行紧缩
  - 优点：减少紧缩次数，提高CPU利用率和系统吞吐量。
  - 缺点：增加了空闲分块表管理的复杂性。

2. 对换技术（1）

- 作用：解决多个程序存在而导致内存区不足。
- 对换技术：通过选择一个进程，把其暂时移出到磁盘，腾出空间给其他进程使用，同时把磁盘中的某个进程再换进内存，让其投入运行，这种互换称对换。
- 对换进程选择：把时间片耗尽或优先级较低的进程换出，因为短时间内它们不会被投入运行；
- 数据区和堆栈是进程运行时创建和修改的，可通过文件系统把这些可变信息作为特殊文件移出。
- 由OS控制内存与外存信息交换。

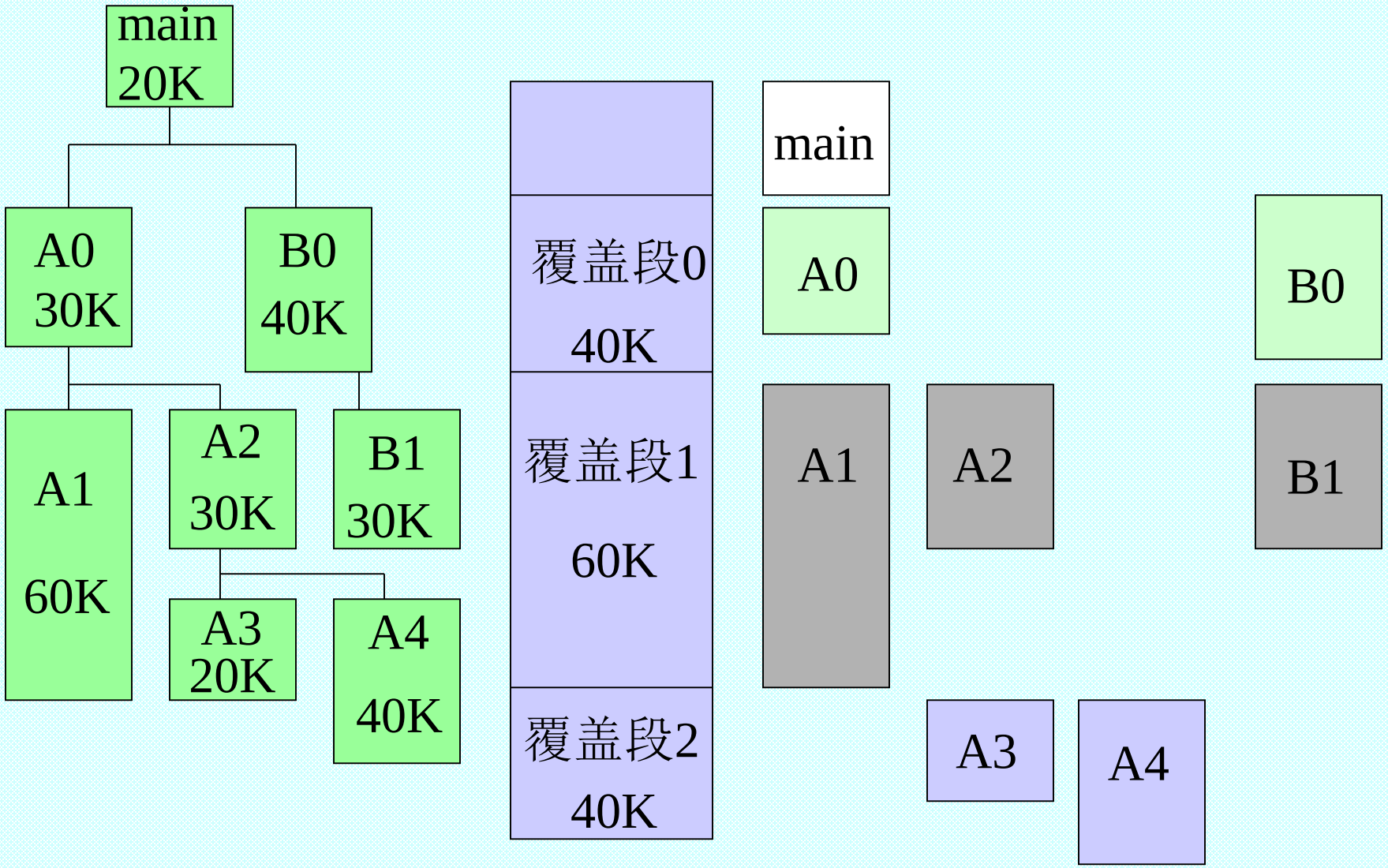
2. 对换技术（2）

- 批处理系统中，当有进程要求动态扩充内存且得不到满足时可触发对换；
- 分时系统中，对换可与调度结合在一起，每个时间片结束或执行I/O操作时实施。
  - 进程一旦因时间片到或因等待事件而不能运行时，它不但让出CPU，而且也要释放出其所占有的主存空间，并且把该进程的程序和数据以文件的形式保存在外存中。
  - 直到调度程序再次调度到它时，才重新进入主存运行，这时又把它的程序和所需数据送入主存。

3. 覆盖技术（1）

- 作用：解决用户程序长度超出物理内存总和。
- 覆盖技术：指一个作业的若干程序段（或数据段）间，或几个作业的某些部分间共享某主存空间。
  - 用于OS：OS常用部分常驻内存，不常用部分存于外存;
  - 用于用户作业：用户指定各程序段调入内存的先后次序，以及内存中可以覆盖的程序段位置。
- 不足：由用户程序自己控制内外存信息交换，用户负担很重，且程序不宜过长，用于早期的OS。

3. 覆盖技术（2）


main
20K
A0
30K
B0
40K
A1
60K
A2
30K
B1
30K
A3
20K
A4
40K
覆盖段0
40K
覆盖段1
60K
覆盖段2
40K
main
A0
A1
A2
A3
A4
B0
B1

4.3 分页存储管理

- 基本思想
  - 固定和可变分区有碎片存在，解决碎片有两种基本思路；
  - 可变分区采取内存紧凑技术消除碎片；
  - 分页存储采取由连续分配变为离散分配的方法。

4.3.1 分页存储管理基本原理

- 等分主存
  - 把主存划分为相同大小的存储块，称为页架，并按其在内存中的地址顺序从0开始对其编号，记为页架号或块号。
  - 不同系统，页架的大小不相同，但对一个特定的计算机系统来说，其大小固定不变。
- 用户逻辑地址空间分页：
  - 将用户程序的逻辑地址空间划分成若干个与页架大小相等的部分，每个部分称为页，同样，按逻辑地址顺序从0开始对页进行编号，记为页号。

1、逻辑地址表示

- 逻辑地址表示：

在分页系统中，每个逻辑地址用一个数对表示：

（p,d）

其中：p：页号；

d：页内偏移地址。

例如：逻辑地址A，页大小为 L，则：

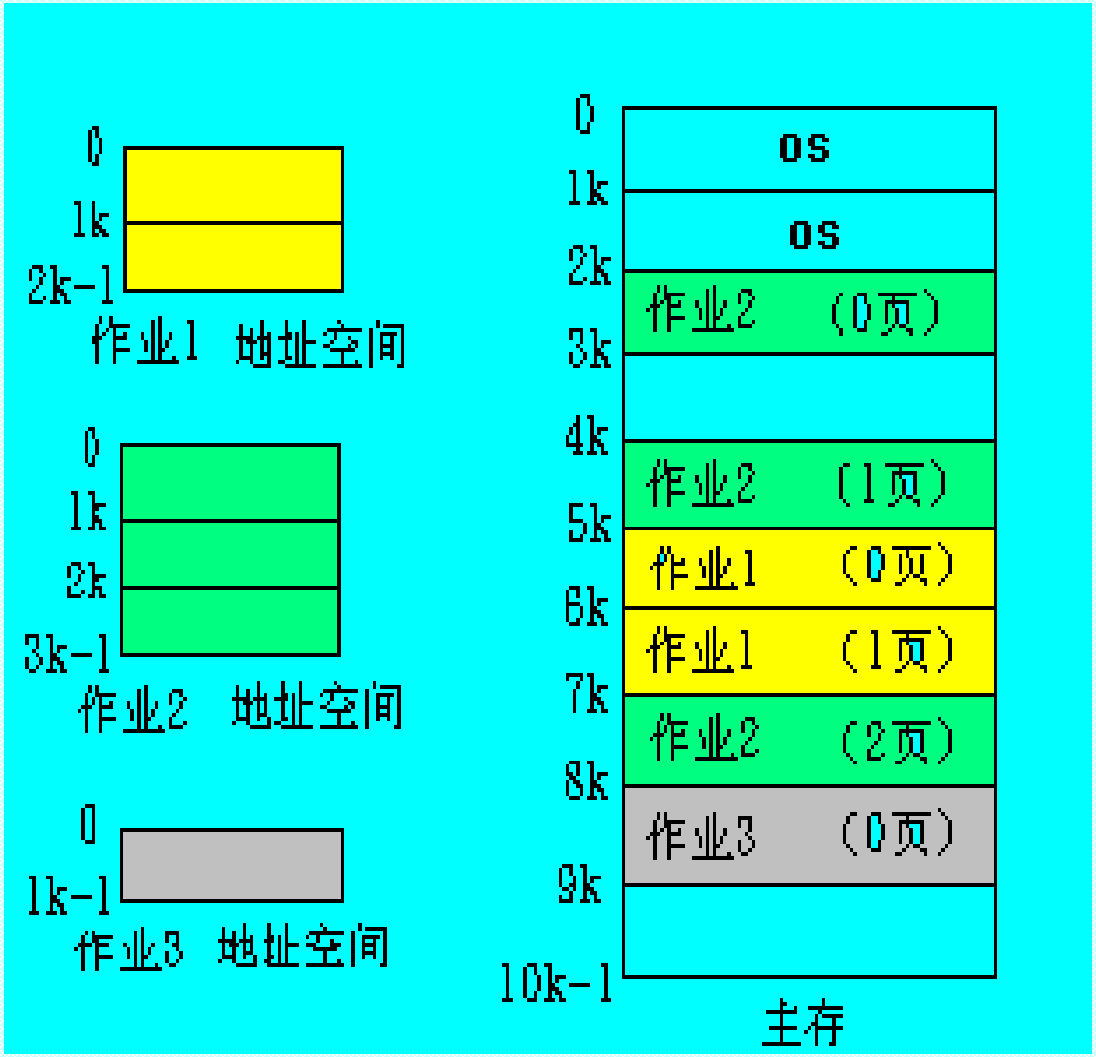
p = INT[A/L]　　d = A mod L

逻辑地址2500，页大小1024，则p = 2，d = 452，

逻辑地址可表示为（2，452）。

2、主存分配原则

- 主存以页架为单位进行分配；
- 分配的页架可以连续，也可以不连续；
- 可以将作业的任意一页放入主存的任意一页架中；
- 作业所有页一次性全部装入主存，若主存空间不够，则作业等待。


0
1k
2k-1
作业1 地址空间
0
1k
2k
3k-1
作业2 地址空间
0
1k-1
作业3 地址空间
0
1k
2k
3k
4k
5k
6k
7k
8k
9k
10k-1
OS
OS
作业2 (0页)
作业2 (1页)
作业1 (0页)
作业1 (1页)
作业2 (2页)
作业3 (0页)
主存

3、页表

- 定义：用于管理和维护进程页和页架映射关系的数据结构，称为页表，也叫页映象表，记为PMT。

页号	块号
0	2
1	4
2	6
3	8

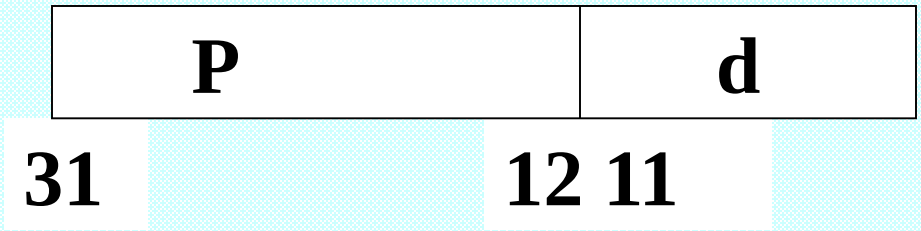
3、页表

- 系统创建进程时，同时为其产生一个PMT。进程结束时，PMT删除。
- 每个进程的页表存放在主存的一个连续地址空间中。
- 系统中有一个页表寄存器，用来存放当前正在运行的进程的页表起始地址和页表长度。

4、页面大小的确定

- 页面大小由机器的地址结构所决定
  - 地址场分两部分：页号和页内偏移；
  - 地址场的长度决定最大逻辑地址空间。
- 页面较小，可使页内碎片小，有利于提高主存利用率，但会使页面数量增多，导致页表过长，占用过多主存。
- 页面较大，可减少页表长度，但又会使页内碎片增加。
- 页面大小的选择应结合计算机指令运算的效率，通常页大小取2的幂。

- 如：Intel X86，其逻辑地址结构为：

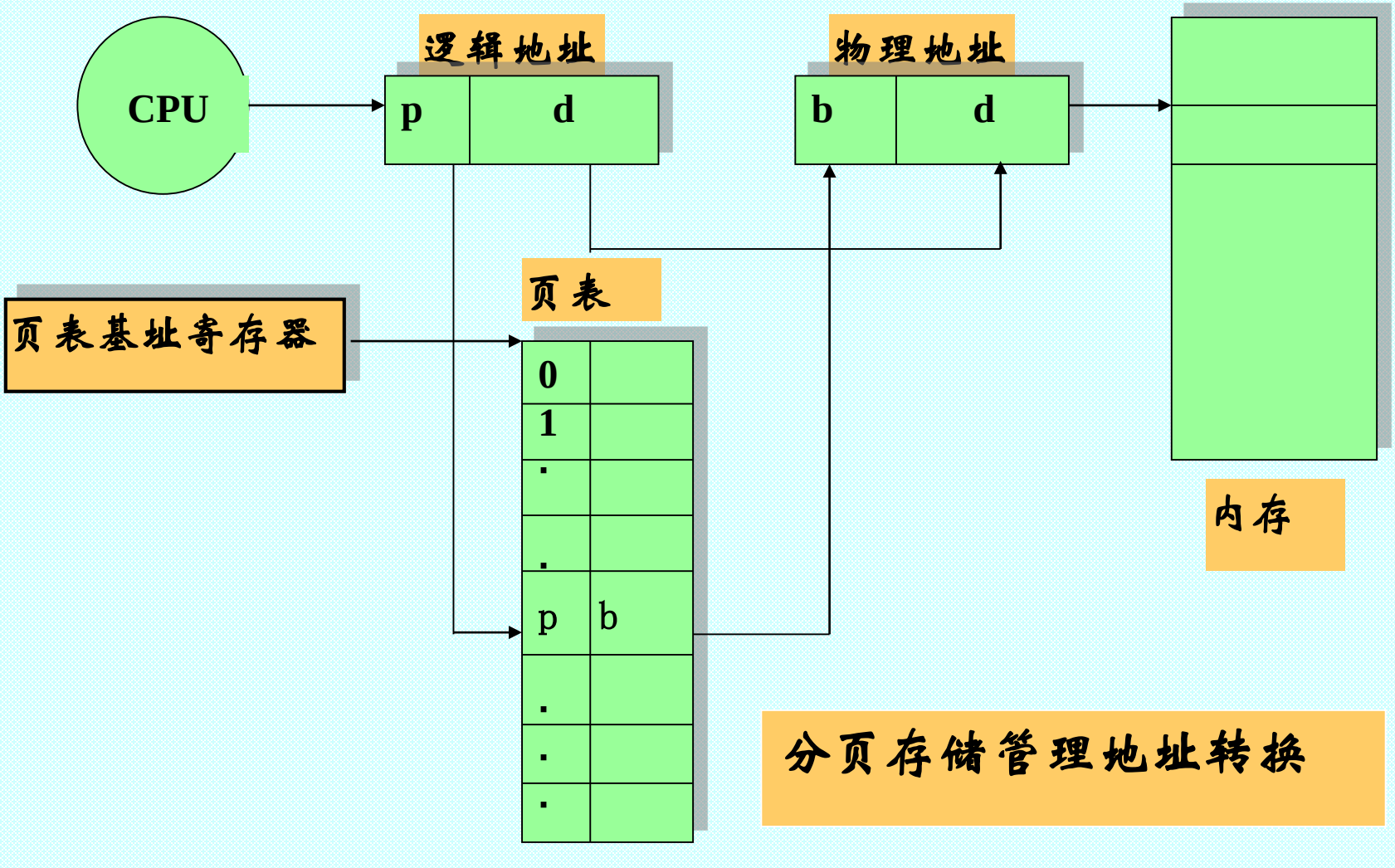


可知其地址场长度为32位，
逻辑地址空间为：4G。
页大小为4K(2^{12})，
页表长为1M(2^{20})

5、地址转换与存储保护

- 从逻辑地址中求出页号和页偏移；
- 以页号为索引查找页表，得到相应的块号；
- 将块号转换为块的物理内存地址，并与页偏移相加获得相应的物理地址。

5、地址转换和存储保护



6、实例

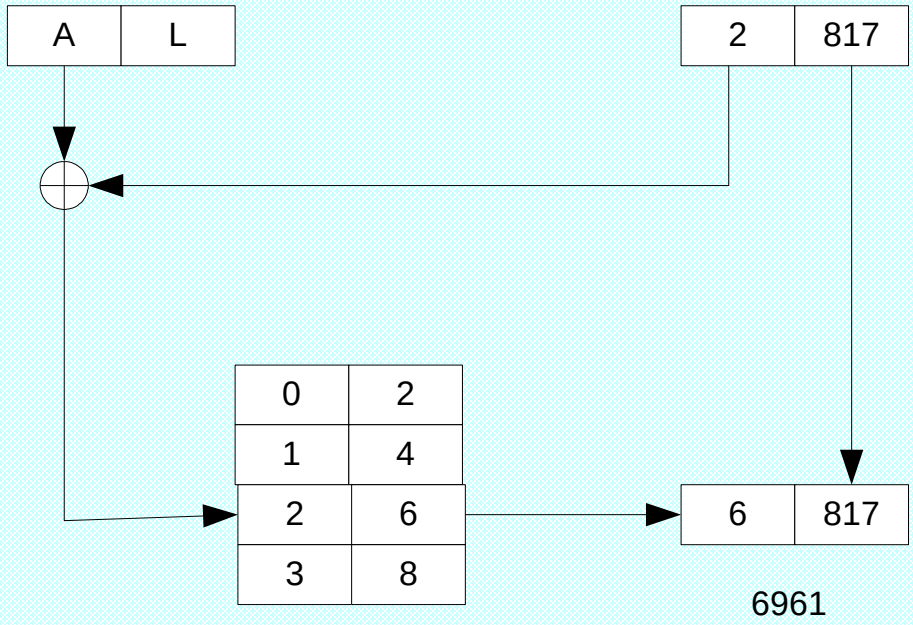
例：一个进程的PMT如图，每页1024字节，求出逻辑地址为2865的物理地址。

0	2
1	4
2	6
3	8

解： P = INT[2865/1024] = 2

d = 2865 mod 1024 = 817

物理地址：6x1024+817 = 6961



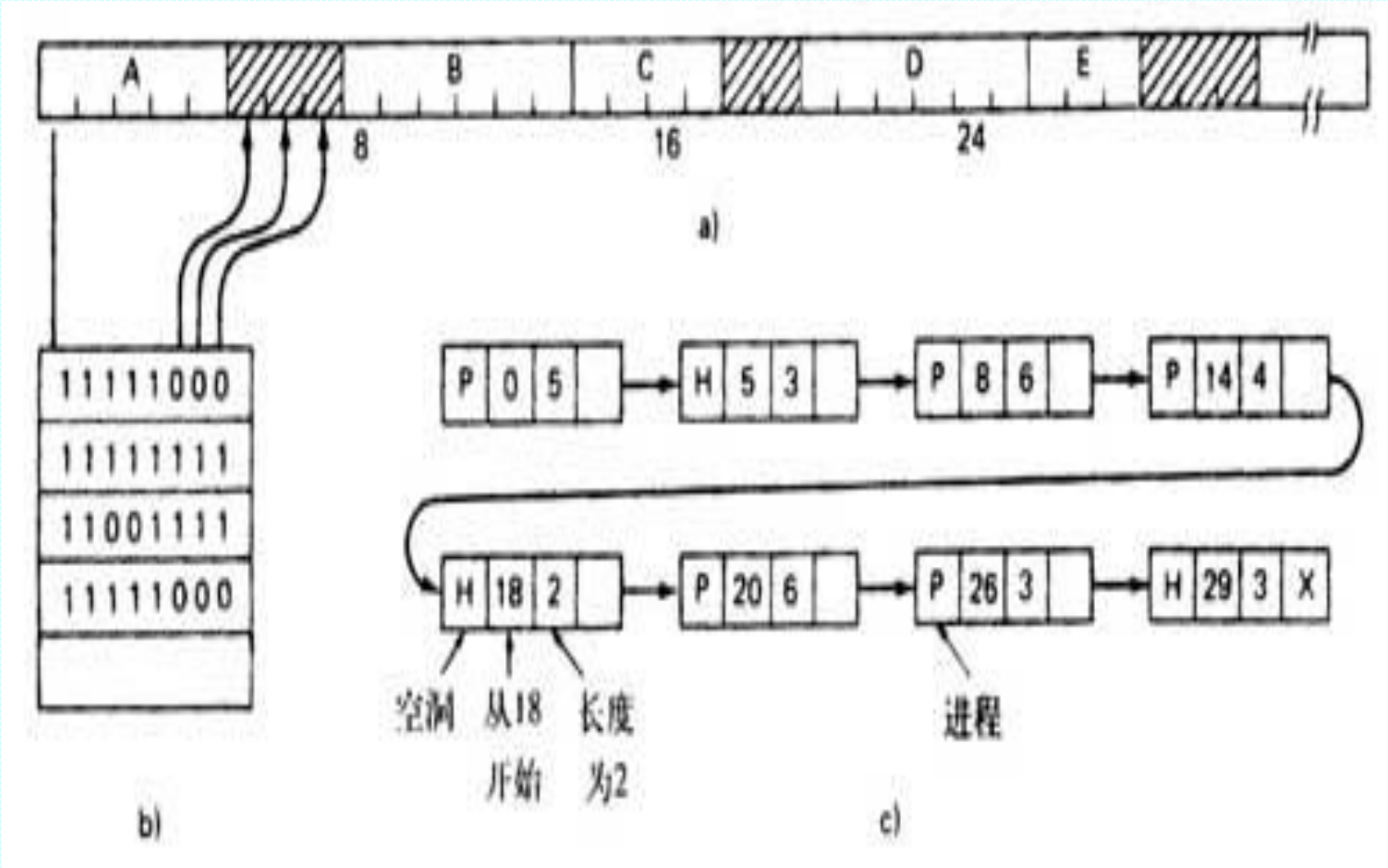
7、简单分页优缺点

- 优点
  - 主存利用率高，不存在页外碎片，极少页内碎片，存在于每个进程最后页内。
  - 主存分配和释放快。
  - 分区管理简单。
- 缺点：
  - 要求一次将进程全部页装入主存；
  - 存在页内碎片。

4.3.2分页存储空间分配和去配

- 位示图法：用每位的状态来表示相应内存物理块是否已分配。
- 链表方法：用链表来表示内存物理块的分配情况。
- 分配算法：

内存分配的位示图和链表方法



4.3.4分页存储空间页面共享和保护

- 程序共享
- 标志位保护方法
- 键保护方法。

1．页面共享

- 程序共享--由于指令包含指向其他指令或数据的地址，进程依赖于这些地址才能执行，不同进程中正确执行共享代码页面，必须为它们在所有逻辑地址空间中指定同样页号。
- 假定有一个被共享的编辑程序 EDIT，共有三页，现有两进程共享它。P1和p2的逻辑空间中均空出初始部分0页、1页和2页，而内存管理让这三页分别指向驻留在内存中的4、7、9页框中的EDIT。二个进程执行时用自己的页表进行地址映射，物理地址并不连续但逻辑地址是连续的，于是便可正确实现两个进程对编辑程序 EDIT的页面共享。

2. 页面保护

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

3. 运行时动态链接

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

3. 运行时动态链接

- 应用程序 main 1 .c 需要使用动态链接共享库：
- 编译时，给出main1.c，并包含#include <stdio.h>等头文件；
- 链接器对编译输出信息：main1.o和标准共享库libc.so的重定位和符号表信息进行静态链接，获得链接的可执行目标代码命名为Exmain1；
- 装入器(execve( ))加载和运行Exmain1时，发现包含动态链接器的路径名，它本身是一个共享目标代码（如，在Linux系统上的LD-LINUX.so），装入器不再像它通常那样将控制传递给应用程序，取而代之是加载和运行这个动态链接器。
- 动态链接器通过执行下面重定位完成链接任务：
  - (1) 重定位libc.so的文本和数据到某个内存段。在Linux系统中，标准共享库被加载到从地址0x40000000开始的区域中；
  - (2) 重定位Exmain1中所有对由libc.so定义的符号的引用；
  - (3) 动态链接器将控制传递给应用程序，从这个时刻开始，共享库的位置便固定，并在程序执行过程中都不会改变。

4.4 分段存储管理

- 4.4.1 程序的分段结构
- 4.4.2 分段存储管理的基本原理
- 4.4.3 分段存储管理共享和保护
- 4.4.4 分段和分页比较

4.4.1 程序分段结构

- 分段存储管理引入的主要原因
- 模块化程序设计的分段结构
- 分页存储管理---一维地址结构
- 分段存储管理---二维地址结构

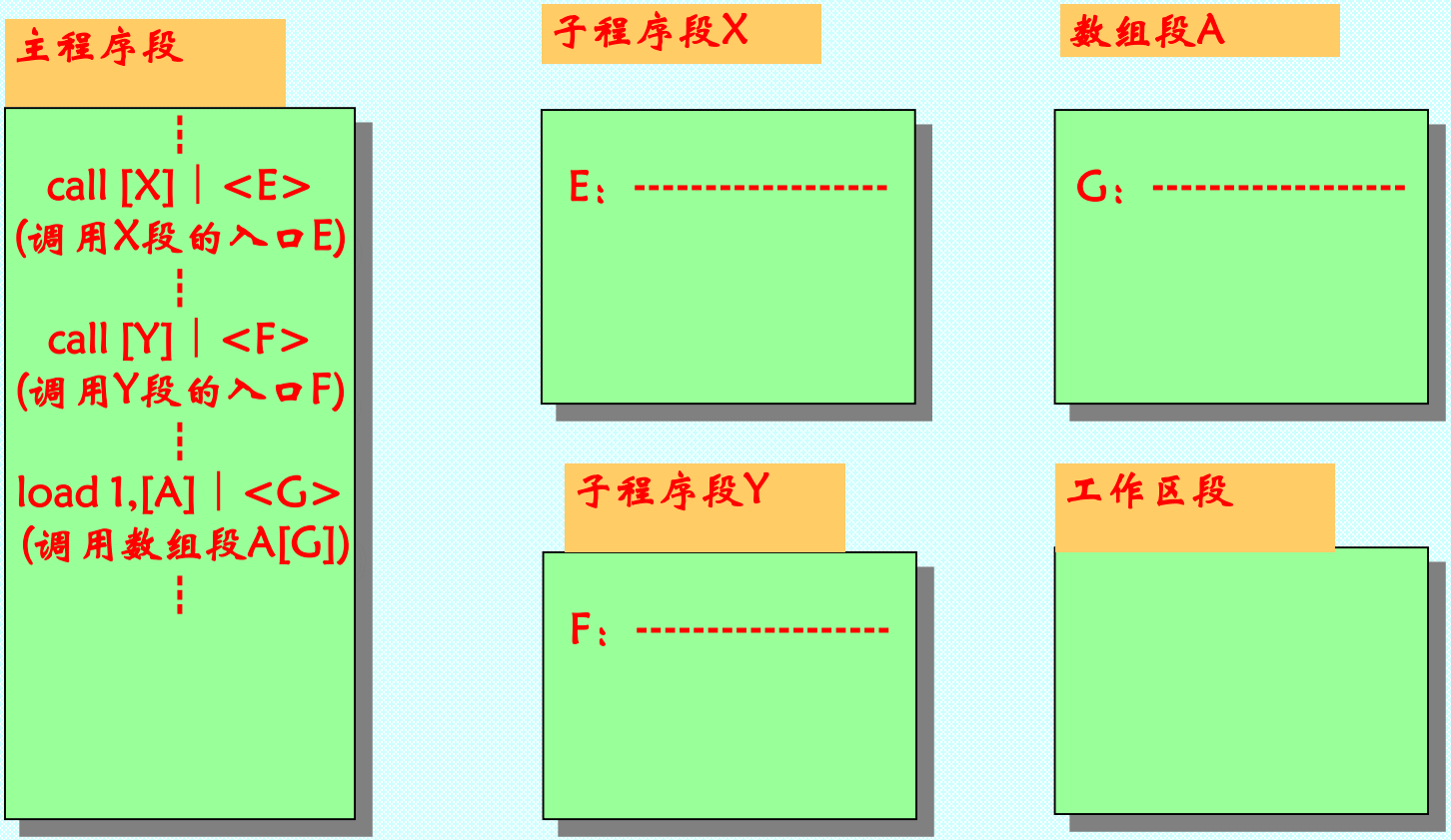
分段存储管理引入的主要原因

1、段的定义：一组逻辑信息的集合。

2、分段的引入：主要目的是为了满足用户在编程和内存使用上要求：

- 方便编程：通常，一个程序是由若干个自然段组成，因而，用户希望能够把程序按逻辑关系分成若干个段，每个段有段名和长度，用户程序在执行时可按段名和段内地址进行访问。
- 共享和保护：在实现程序和数据共享和保护时，都是以信息逻辑单位为基础的，比如，共享例程和函数。而在分页系统中，每页是存放信息的物理单位，本身并没有完整的意义，因而不便于实现信息共享和保护，而段是信息的逻辑单元 。

模块化程序设计的分段结构

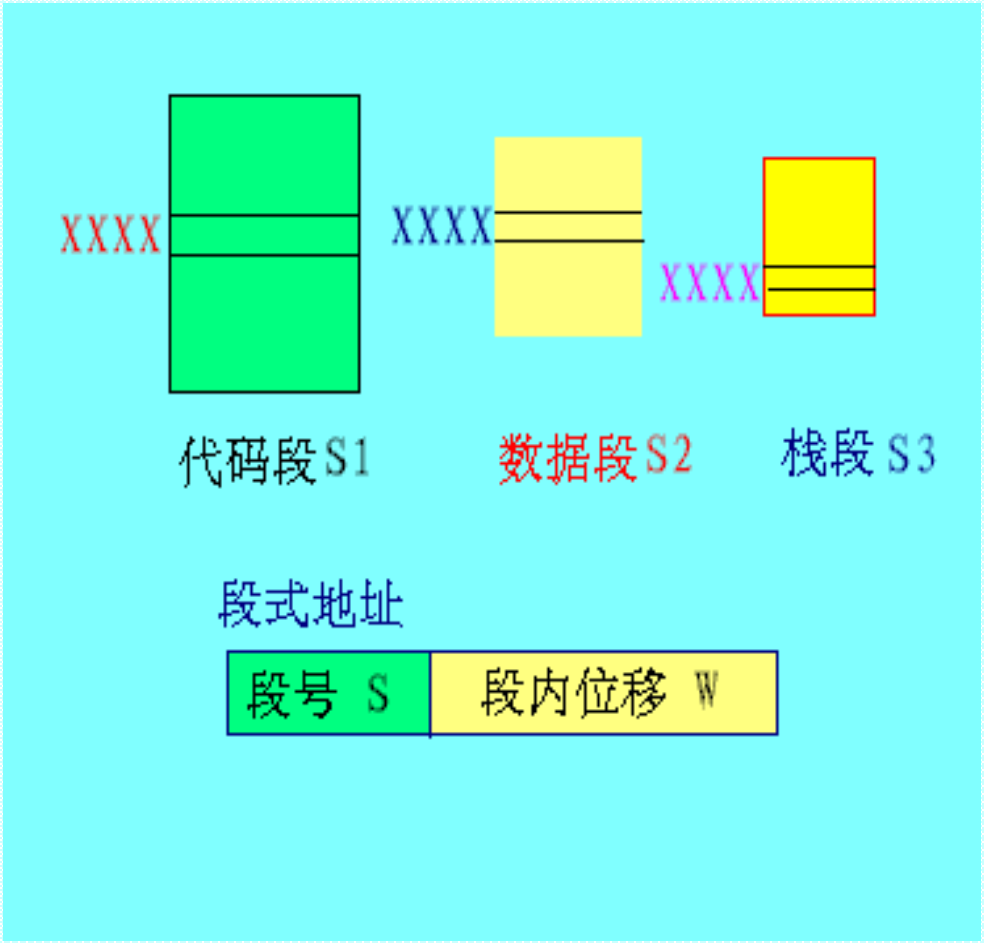


4.4.2 分段存储管理基本原理

- 逻辑地址空间划分和表示：
  - 每个进程的地址空间被划分为若干段，每段有段名；
  - 每段都从0开始连续编址，段的长度由相应的逻辑信息组的长度决定。
  - 段间可以不连续编址。
  - 采用二维地址空间来表示，

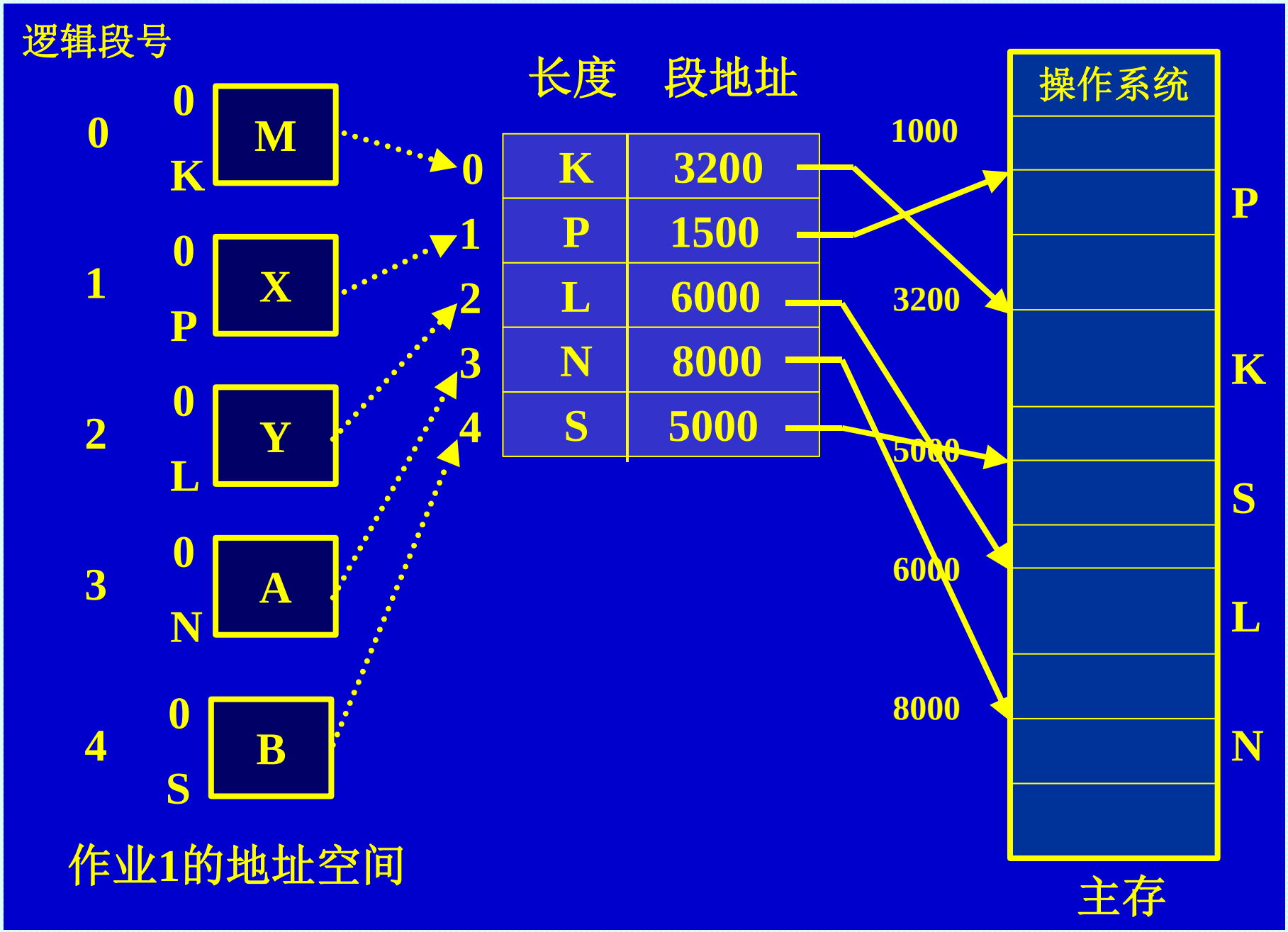
    V =（S，W）；

    其中， S：段号，W：段内地址。


XXXX
XXXX
XXXX
代码段S1
数据段S2
栈段 S3
段式地址
段号 S
段内位移 W

4.4.2 分段存储管理基本原理

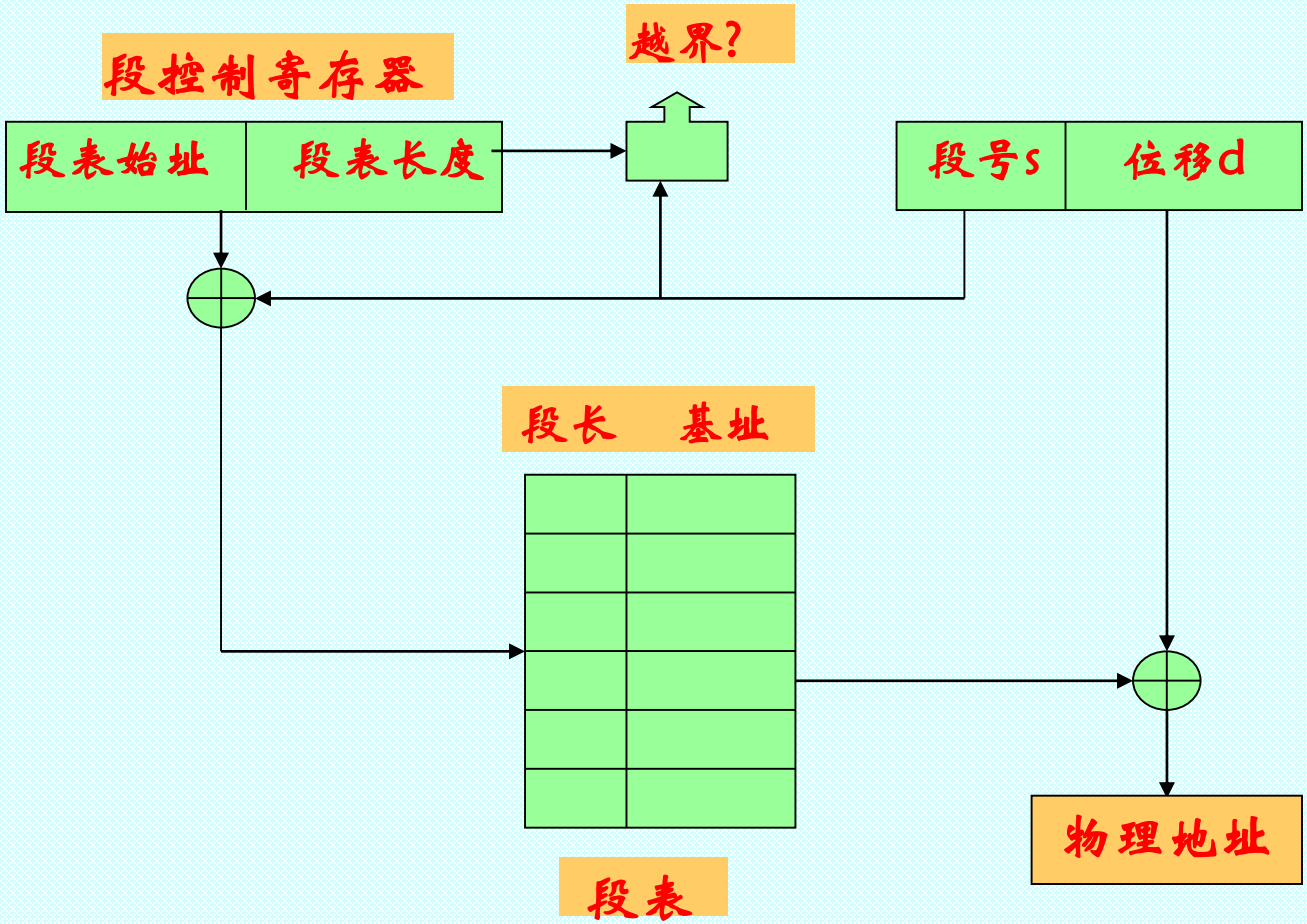
- 主存分配：
  - 以段为单位进行主存分配；
  - 段内连续存放：每段分配一个连续的主存物理空间；
  - 段间可以不连续：段和段之间在主存中地址可以是离散的。


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

4.4.2 分段存储管理基本原理

- 段表：
  - 定义：用于记录和管理进程分段信息的数据表称为段表。段表应包含：段号、段长和段的主存起始地址。
  - 系统创建进程时,同时为其产生一个段表。进程结束时，段表删除。
  - 每个进程的段表存放在主存的一个连续地址空间中。
  - 系统中由一个段表寄存器中，存放进程的段表的起始地址和段表长度。

4.4.2 分段存储管理基本原理



4.4.3分段存储管理共享和保护

- 多对基址/限长寄存器
- 段的共享，是通过不同作业段表中的项指向同一个段基址来实现。
- 几道作业共享的例行程序就可放在一个段中，只要让各道作业的共享部分有相同的基址/限长值。
- 对共享段的信息必须进行保护，保护位用来对共享段实施保护，如禁写、禁修改等。

地址转换实例

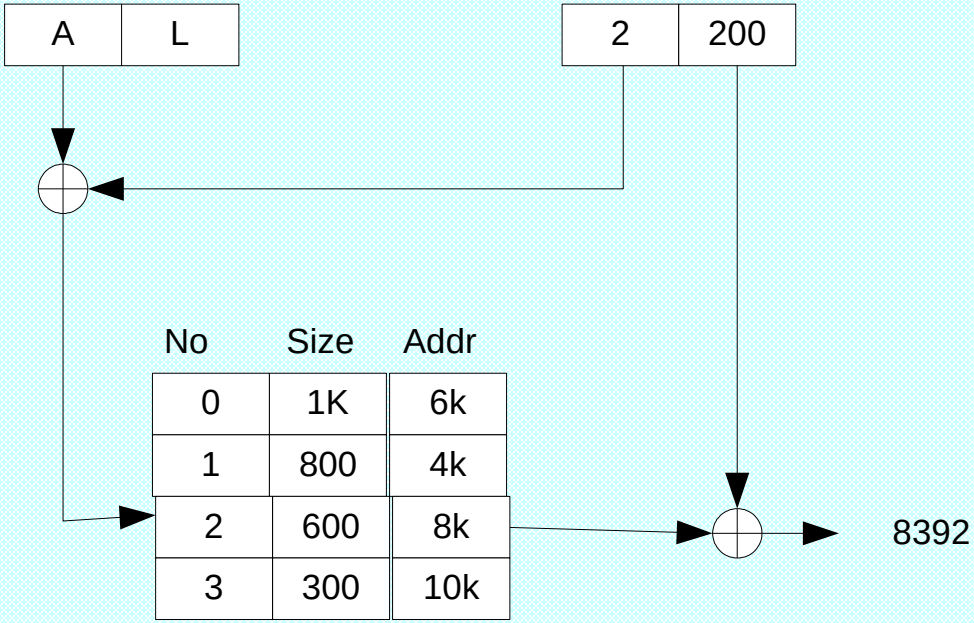
段表：

段号	段长	起始地址
0	1K	6K
1	800	4K
2	300	10K

地址转换实例

- 段的地址转换：

  已知逻辑地址V = (2,200)，　（1K=1024字节）



- 简单分段的特点：
  - 没有内部碎片；
  - 便于共享和保护；
  - 存在外部碎片；
  - 由于段内连续分配，段的长度受内存空闲区大小的限制；
  - 需要更多硬件支持。

4.4.4 页式和段式系统的区别

- 不同点：
  - 页是信息的物理单位，分页是为了实现离散分配方式，以减少内存的碎片，提高内存利用率。或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。

    段是信息的逻辑单位，它含有一组其意义相对完整的信息，分段的目的是为了能更好地满足用户的需要。
  - 页的大小固定且由系统确定，把逻辑地址划分为页号和页内地址两部分是由硬件实现的，因而，一个系统只能有一种大小的页面。

    段的长度不固定，取决于用户编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

4.4.4 页式和段式系统的区别

- 分页的逻辑地址空间是一维的，即单一的线性地址空间，程序员只需利用一个地址符。即可表示一个地址。

  分段的逻辑地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

- 相同点
  - 采用离散分配方式；
  - 通过地址映射机构实现地址变换。

4.5虚拟存储管理

- 4.5.1 虚拟存储器概念
- 4.5.2 请求分页虚拟存储管理
- 4.5.3 请求段页式虚拟存储管理
- 4.5.4 页面管理策略

4.5.1 虚拟存储器概念

- 为什么要引入虚拟存储器?
  - 实存管理需要将作业一次全部内存方能运行，使作业的大小受到内存的极大限制。
  - 解决的方法两种：
    - 物理上增加主存：硬件成本增加，并且往往受计算机系统本身的限制。
    - 逻辑上扩充主存：虚拟存储技术 。

程序运行的局部性特性

- 指程序在执行过程中的一个较短时间内，所执行的指令地址或操作数地址分别局限于一定的存储区域中。又可细分时间局部性和空间局部性。
- 时间局部性
  - 指程序中的某条指令一旦被执行，则在较短的时间内，该指令可能被再执行，某个数据被访问，则在较短的时间内，该数据可能被再次访问。
- 空间局部性
  - 指一旦程序访问了某个存储单元，在较短的时间内，其附近的存储单元很可能也被访问。即程序在一段时间内所访问的地址，可能集中在一定的范围内。

产生局部性的典型原因

- 第一：程序中只有少量分支和过程调用，大都是顺序执行的指令；
- 第二：程序含有若干循环结构，由少量代码组成，而被多次执行；
- 第三：过程调用的深度限制在小范围内，因而，指令引用通常被局限在少量过程中；
- 第四：涉及数组、记录之类的数据结构，对它们的连续引用是对位置相邻的数据项的操作；
- 第五：程序中有些部分彼此互斥，不是每次运行时都用到。

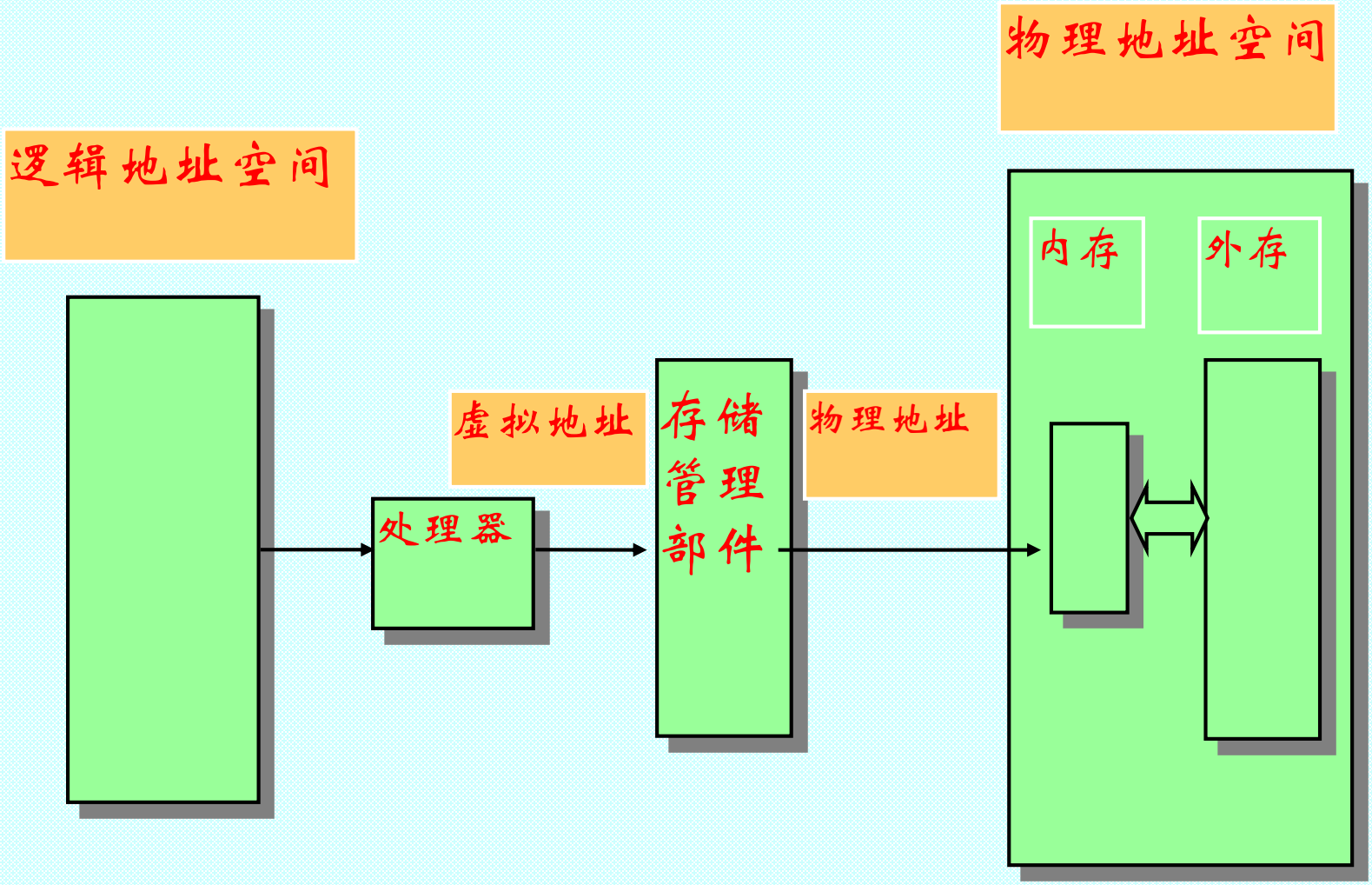
4.5.1 虚拟存储器概念

- 实现虚拟存储器的基本思路
  - 从局部性特性可以知道，一个作业在运行之前，没有必要全部装入内存，而仅需将那些当前要运行的部分指令和数据，先装入内存即可启动运行。
  - 在程序执行过程中，如果需要的指令和数据不在内存中，则由CPU通知OS将相应的页或段调入到内存，然后继续执行。
  - 另一方面，OS将内存中暂时不使用的页或段调出，保存在外存上，以腾出较多的内存。
  - 从用户角度看，该系统具有的容量，将比实际的内存容量大得多。

4.5.1 虚拟存储器概念

- 虚拟存储技术：为用户提供一种不受物理存储器结构和容量限制的存储技术称为虚拟存储技术。
- 虚拟存储器：在具有层次结构存储器的计算机系统中，采用自动实现部分装入和部分对换功能，为用户提供一个比物理内存容量大得多的，可寻址的一种“内存储器”。
- 物质基础
  - 相当容量的辅存
  - 一定容量的主存
  - 地址变换机构
- 虚拟空间的限制
  - 指令中的地址场长度限制；（主要原因）
  - 外部存储器大小的限制。

虚拟存储器概念图



实现虚拟存储器须解决的问题

- 内存外存统一管理问题
- 逻辑地址到物理地址的转换问题
- 部分装入和部分对换问题

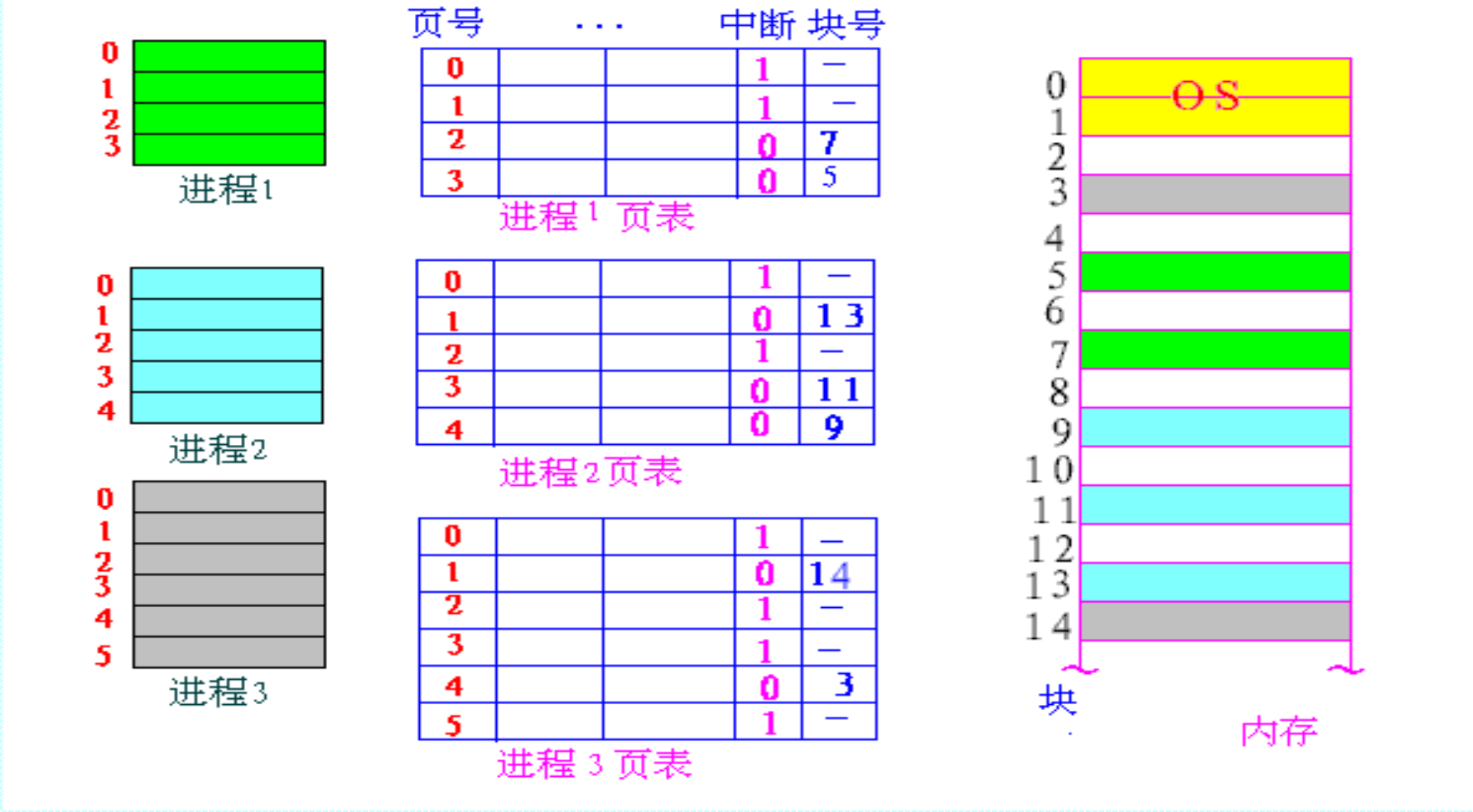
虚拟存储管理实现技术

- 请求分页虚拟存储管理
- 请求段页式虚拟存储管理

4.5.2 分页虚拟存储系统

- 与实存分页管理基本相同
- 主要区别在于：
  - 实存中需要将所有页一次装入内存，虚存中只需将部分页装入内存。 在执行过程中访问到不在内存的页面时，产生缺页中断，再从磁盘动态地装入 。
- 如何发现页面不在内存中？怎样处理这种情况呢？
  - 扩充页表的内容，增加驻留标志位等信息。
  - 增加了请求调页和页面置换功能。

4.5.2 分页虚拟存储系统



请求分页虚存管理页表扩展

页号	……	驻留标志位	引用位	修改位	保护位	内存块号

- 驻留标志位(又称中断位)
- 修改位（Modified）
- 引用位（Renferenced）
- 保护位 (Protected)
- 页面号 (Page Number)
- 内存块号(Frame Number)

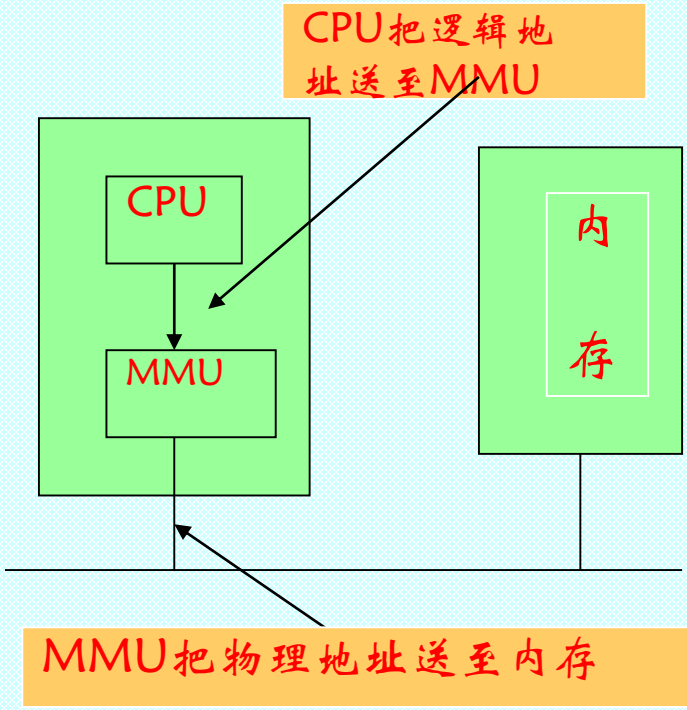
外页表

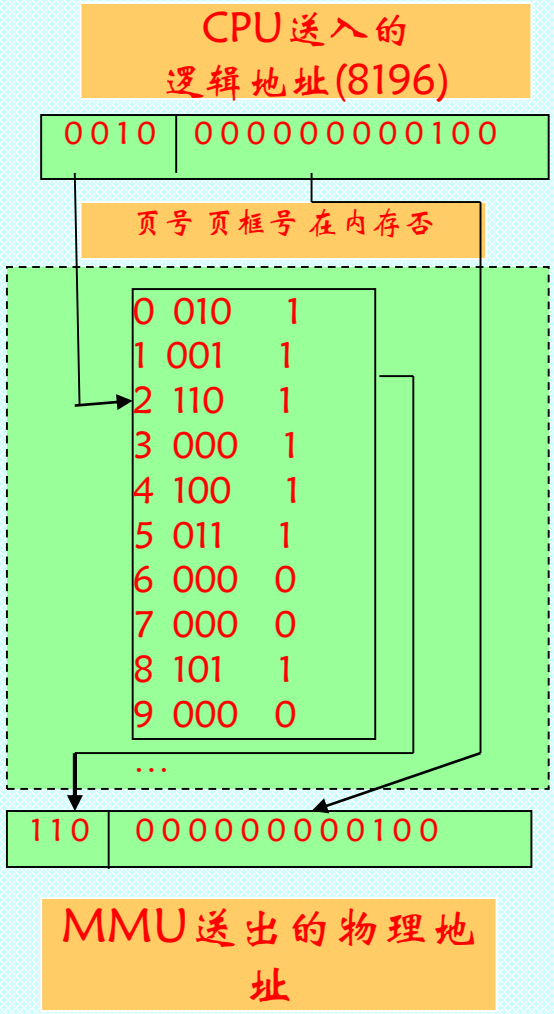
- 页面与磁盘物理地址的对应表，由操作系统管理，进程启动运行前系统为其建立外页表，并把进程程序页面装入外存。
- 该表按进程页号的顺序排列，为节省内存，外页表可存放在磁盘中，当发生缺页中断需要查用时才被调入。

1、分页虚存系统硬件支撑

- 内存管理单元MMU完成逻辑地址到物理地址的转换功能，它接受逻辑地址作为输入，物理地址作为输出，直接送到总线上，对内存单元进行寻址。

1、分页虚存系统硬件支撑





MMU的位置、功能和16个4KB页面情况下MMU的内部操作

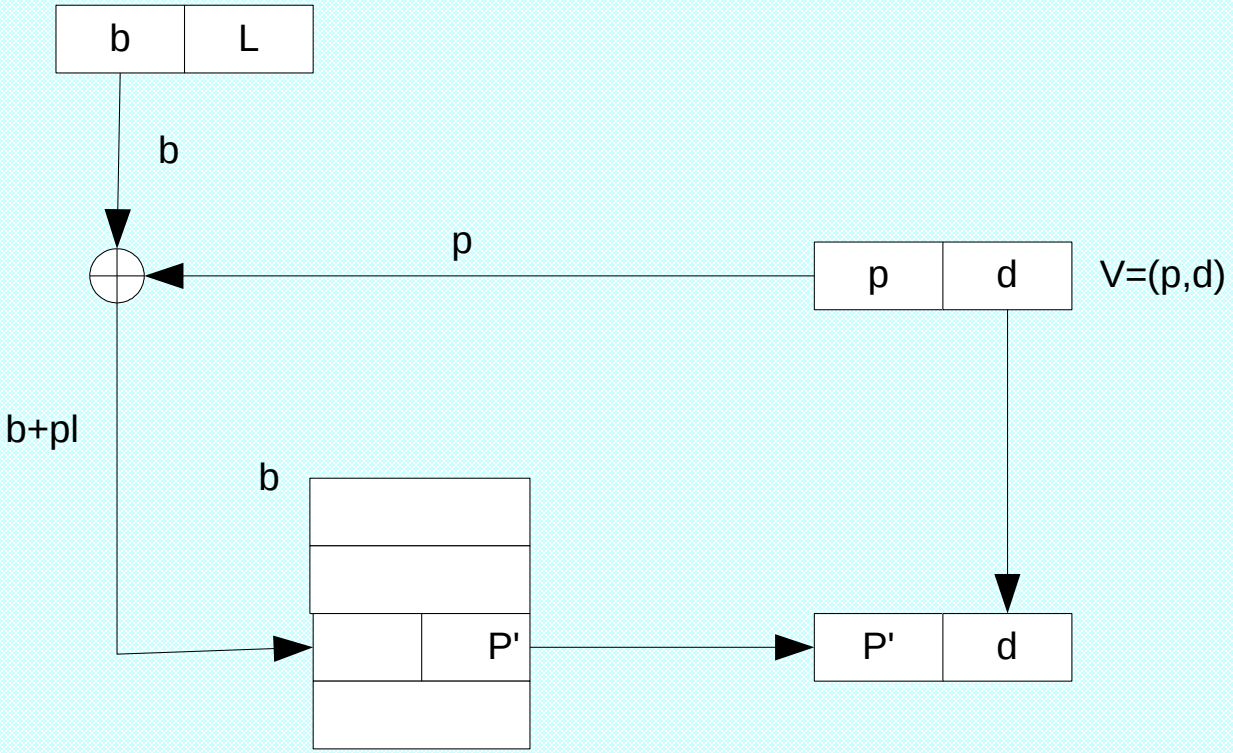
MMU主要功能

- (1)管理硬件页表基址寄存器。
- (2)分解逻辑地址。
- (3)管理快表TLB。
- (4)访问页表。
- (5)发出缺页中断或越界中断，并将控制权交给内核存储管理处理。
- (6)设置和检查页表中各个特征位。

缺页中断处理的过程

- 步1：挂起请求缺页的进程；
- 步2：根据页号查外页表，找到该页存放的磁盘物理地址；
- 步3：查看内存是否有空闲页框，如有则找出一个，修改内存管理表和相应页表项内容，转步6；
- 步4：如内存中无空闲页框，按替换算法选择淘汰页面，检查它曾被写过或修改过吗？若未则转步6；若是则转步5；
- 步5：该淘汰页面被写过或修改过，则把它的内容写回磁盘原先位置；
- 步6：进行调页，把页面装入内存所分配的页框中，同时修改进程页表项；
- 步7：返回进程断点，重新启动被中断的指令。

2、直接映象的页地址转换



2、直接映象的页地址转换

- 特点：
  - 降低了CPU执行指令的速度，访问一个实际物理地址，至少需要访问两次内存，从而使速度将为1/2。
  - 每个进程一个页表，页表的全部表项装在一个地址连续的内存空间。
  - 只适合小进程地址空间。

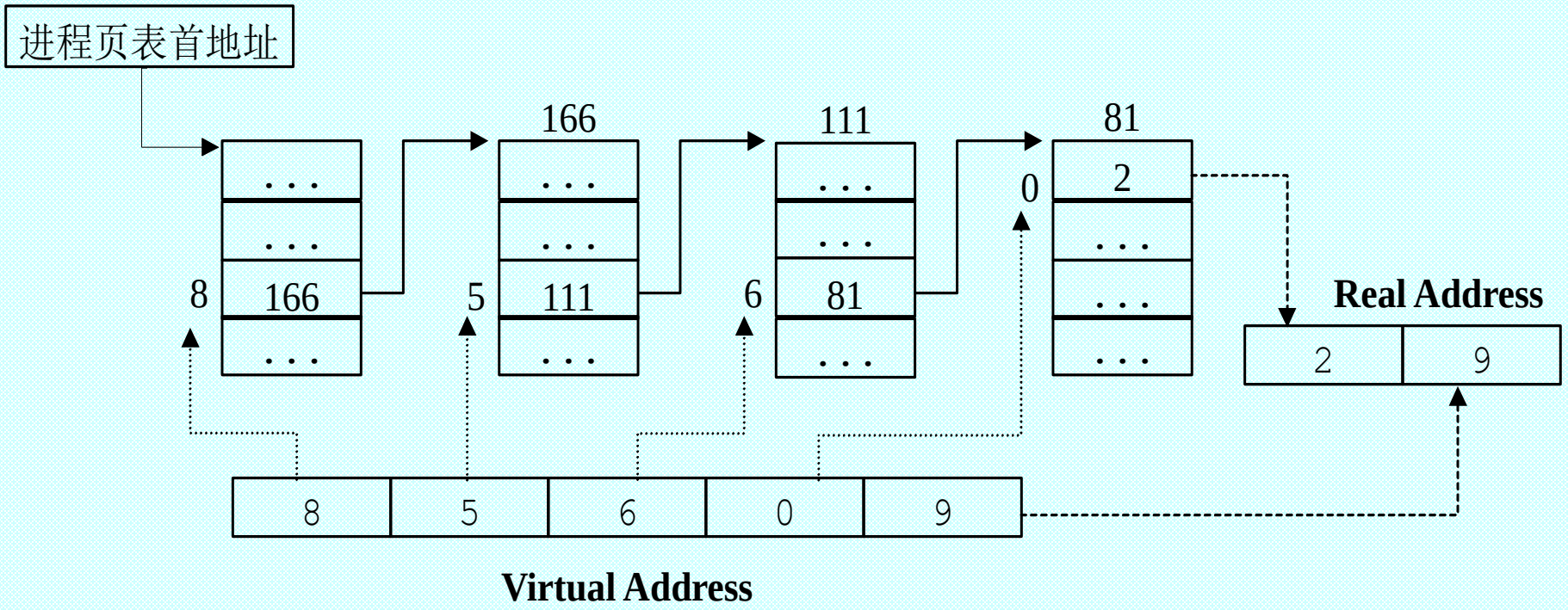
2、直接映象的页地址转换

- 问题
  - 页表可能会非常大。
    - 如：32位OS，意味着一个进程的虚拟地址空间可达4G，如果页大小为4K，则页表有1M个页表目，如果一个表目占4个字节，则一个进程的页表需要4M内存。
  - 地址映射应非常快。

3、多级页表的地址转换

- 为了减少进程页表对内存的占有，通常采取以下两种措施：
  - 对进程页表所需的主存空间，采用离散分配方式。
  - 只将当前需要的部分页表目调入主存，其余保留在虚拟存储器中，需要时才调入。
- 系统为每个进程建一张页目录表，它的每个表项对应一个页表页，而页表页的每个表项给出了页面和页框的对应关系，页目录表是一级页表，页表页是二级页表。
- 逻辑地址结构有三部分组成：页目录、页表页和位移。

3、多级页表地址映射

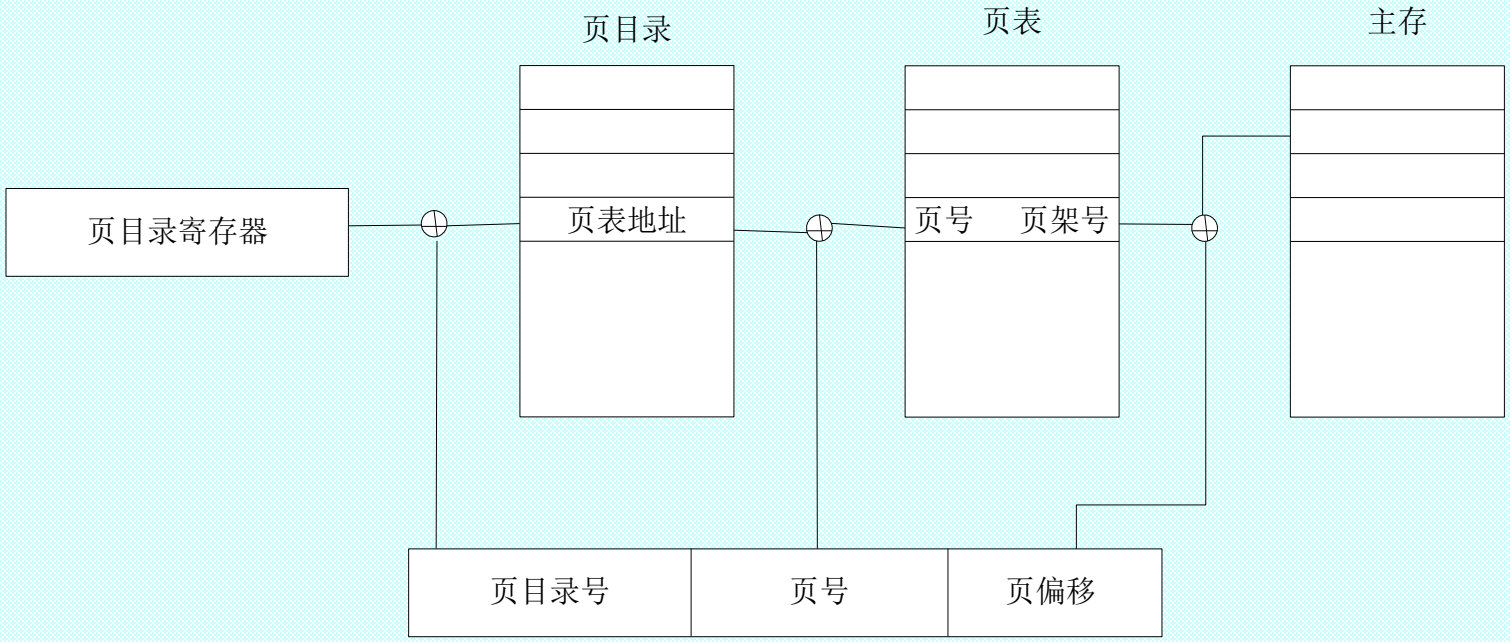

进程页表首地址
166
111
81
8
166
5
111
6
81
0
2
Real Address
2
9
8
5
6
0
9
Virtual Address

3、多级页表地址映射

- 多级不连续导致多级索引。
- 以二级页表为例，用户程序的页面不连续存放，要有页面地址索引，该索引是进程页表；进程页表又是不连续存放的多个页表页，故页表页也要页表页地址索引，该索引就是页目录。
- 页目录项是页表页的索引，而页表页项是进程程序的页面索引。

3、多级页表地址映射

- 二级页表结构及地址映射过程



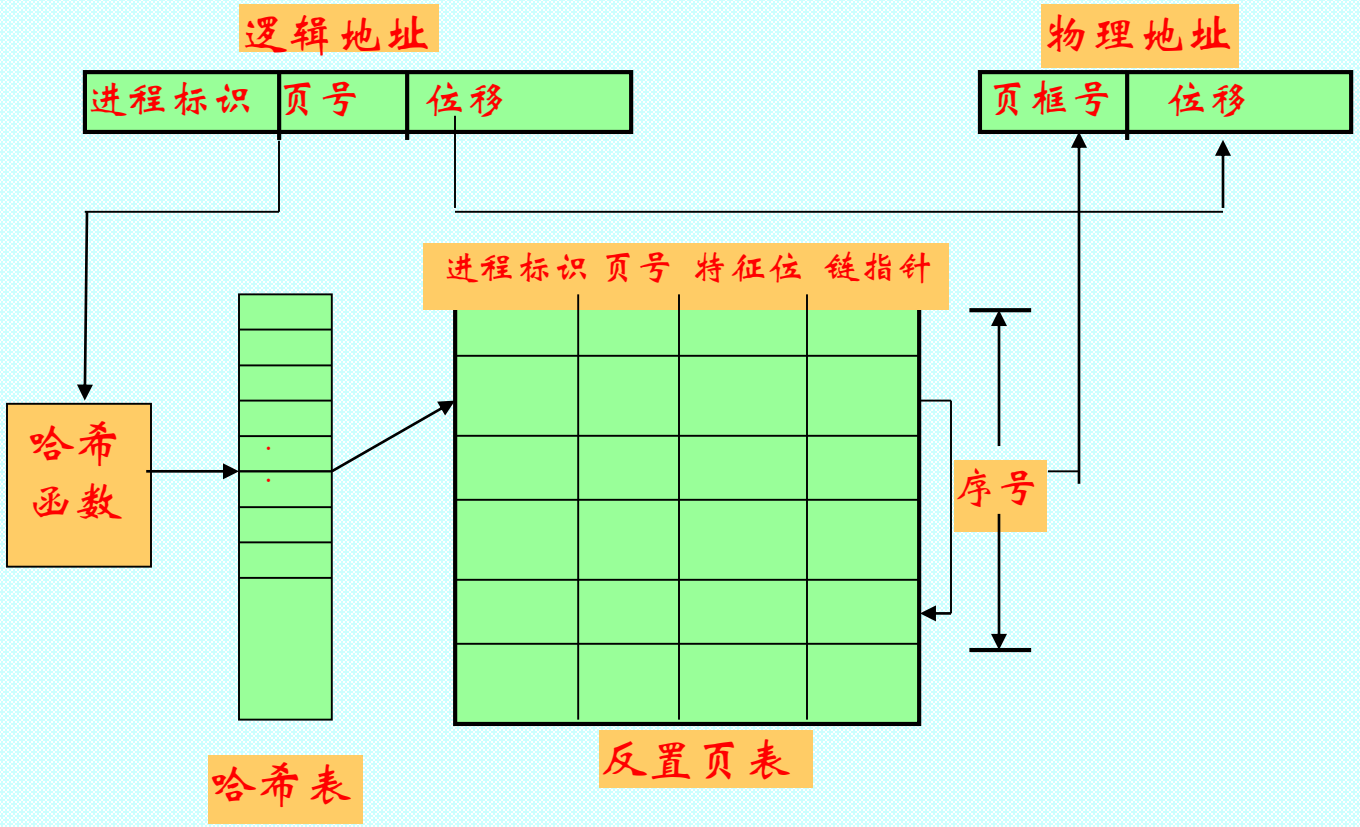
3、多级页表地址映射

- 二级页表地址转换特点
  - 访问数据需要三次访问主存。
  - 页目录调入主存，页表并不全部调入主存
  - 一般情况下，页目录表占一个页架，每页占一个页架。

4、反置页表地址转换

- 目的：减少页表占用主存空间
- 思想：反置页表不是依据进程的逻辑页号来组织，而是依据该进程在内存中的物理页面号来组织（为每一个页架设置一个页表项，并按页架号排序。
  - 如：64M主存，若页面大小为4K，则反向页表只需64KB
- 实现：
  - 每个进程一个反置页表；
  - 虚拟地址中的逻辑页号，经K＝hash（p)获得hash值，并查找hash表；
  - 以Hash表的索引值查找反向表获得页架

4、反置页表地址转换

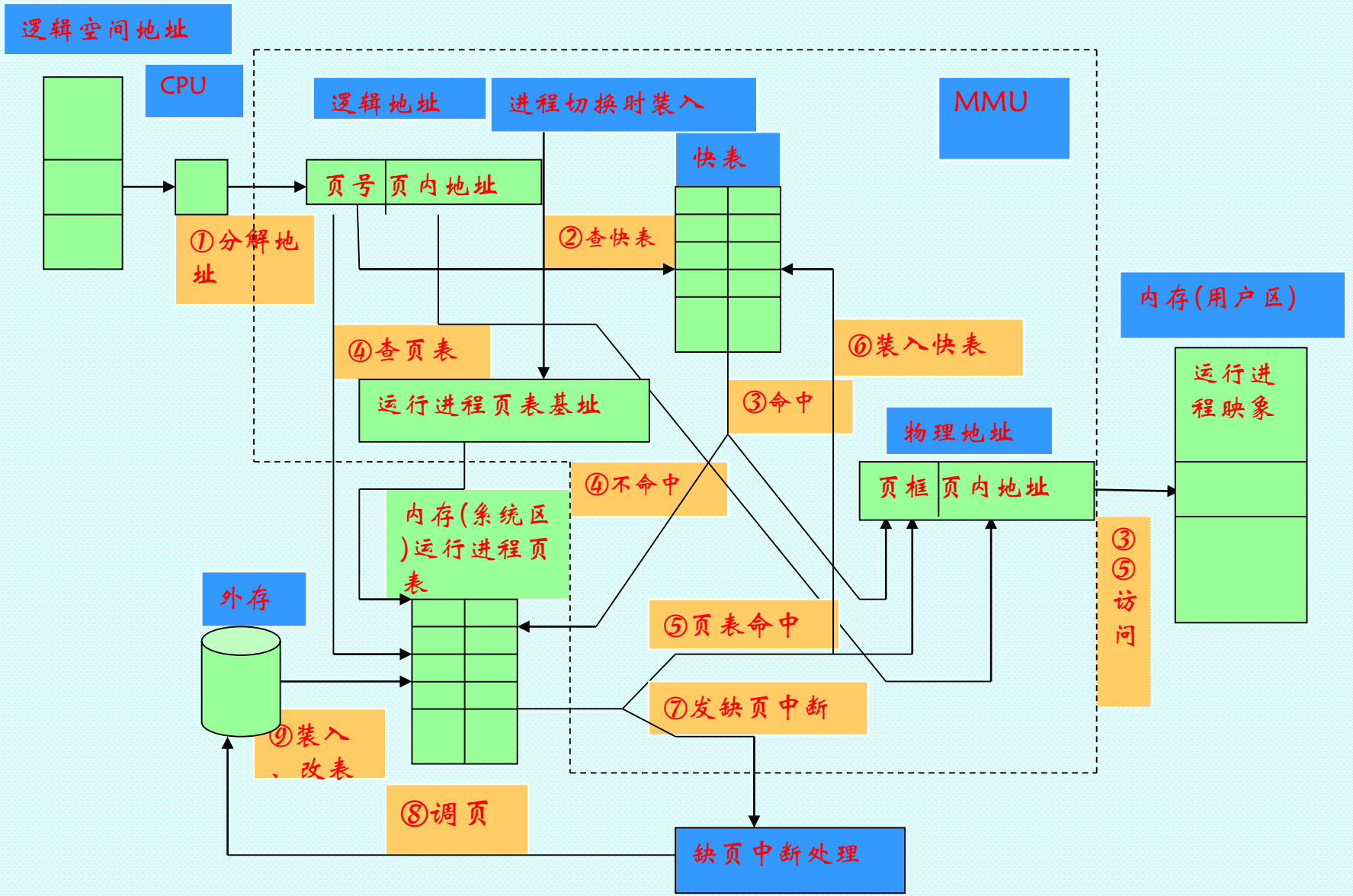


反置页表及其地址转换

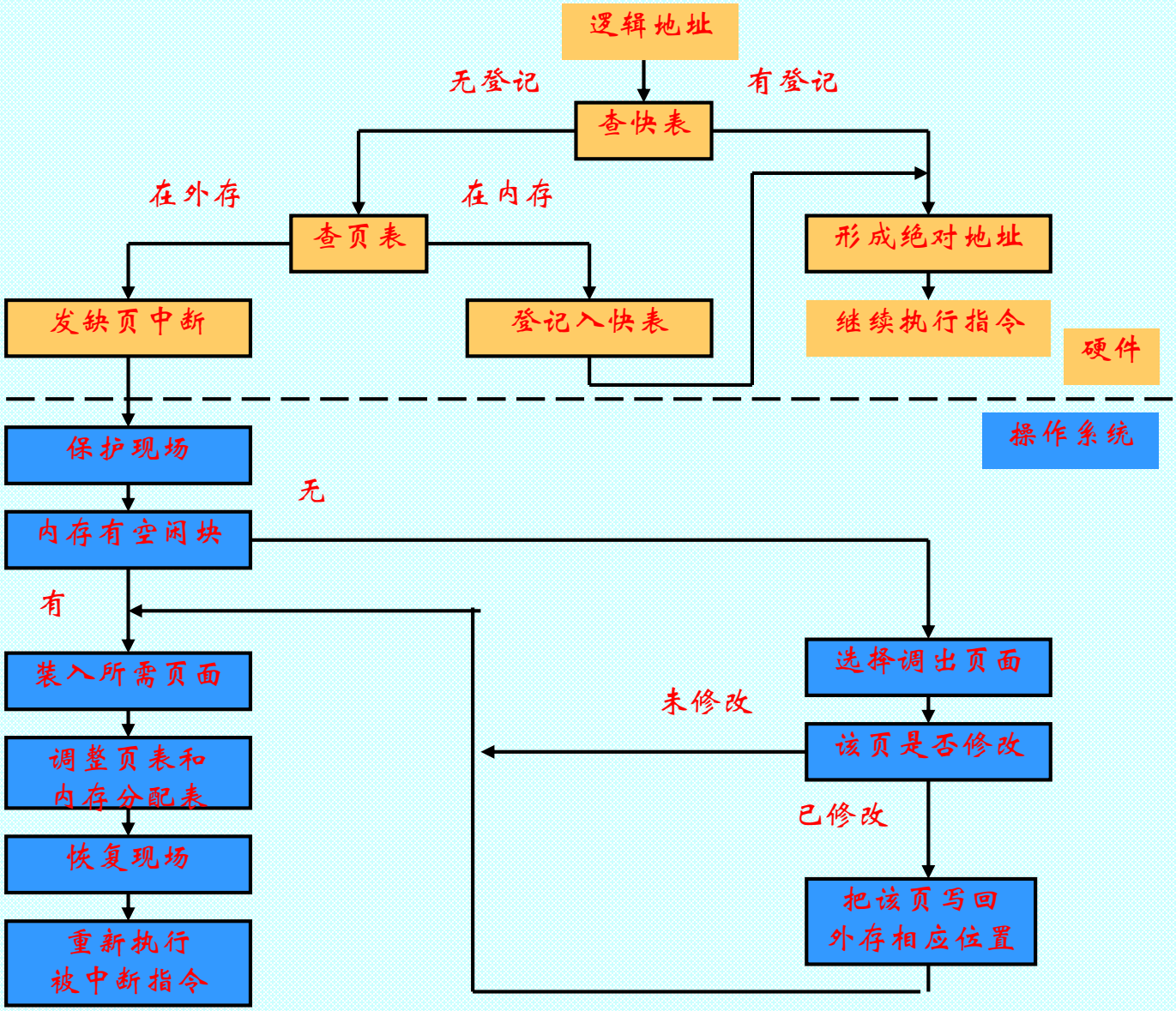
4、反置页表地址转换

- 特点：
  - 反置页表只含已调入主存的页面，不在主存的页面需借助其他手段（传统页表方式）从外存获得；
  - 减少页表占用的主存空间，反置页表的大小只与物理内存的大小相对关，与逻辑空间大小和进程数无关
  - 一个Hash值可能对应多个页号。

5、请求分页虚存地址转换过程(1)



5、请求分页虚存地址转换过程(2)



5、请求分页虚存地址转换过程(3)

- 采用快表的依据
  - 程序执行局部性特性
  - 高速缓存查找速度快
- 快表（TLB）：存放在关联高速缓存中的页表。快表表目包含：页号、页架号、所属进程和页面保护权限。
- 特点
  - 查找时对快表中的各表目同时比较内容，速度快；
  - 采用关联高速缓存成本高 。

请求分页虚存系统优缺点

- 优点：
  - 作业的程序和数据可按页分散存放在内存中，减少移动开销，有效解决了碎片问题；
  - 既有利于改进内存利用率，又有利于多道程序运行。
- 缺点：
  - 要有硬件支持，要进行缺页中断处理，机器成本增加，系统开销加大。

4.5.3请求段页式虚存管理

- 段式存储是基于用户程序结构的存储管理技术，有利于模块化程序设计，便于段的扩充、动态链接、共享和保护,但会生成段内碎片浪费存储空间
- 页式存储是基于系统存储器结构的存储管理技术, 存储利用率高,便于系统管理，但不易实现存储共享、保护和动态扩充
- 把两者结合起来就是段页式存储管理

请求段页式虚存管理的基本原理

1、基本概念

- 主存划分
  - 把整个主存分成大小相等的存储块，即页架。并由低地址开始从0顺序编号，记为页架号。
- 逻辑地址空间划分
  - 先分段：将进程的逻辑地址空间按程序的自然逻辑关系分成若干个段，每个段有外部段名和内部段号；
  - 后分页：将进程的每个段按主存页架大小分成若干个页，每段都从0开始顺序编号。

- 逻辑地址表示
  - 一个逻辑地址用三个参数表示：

    V=(s,p,d)；

    其中S：段号，P：页号，d：页内地址偏移量。

段号(s)	段内页号 (p)	页内位移(d)

主存分配

- 以页架为单位进行内存分配；
- 无需将所有段的所有页面一次性全部转入主存页架。

段页式虚存管理的数据结构

- 段页式存储管理的数据结构包括：
  - 作业表：登记进入系统中的所有作业及该作业段表的起始地址，
  - 段表：包含这个段是否在内存，以及该段页表的起始地址等，
  - 页表：包含该页是否在内存(中断位)、对应内存块号。

地址转换(1)

- 从逻辑地址出发，先以段号s和页号p作索引去查快表，如果找到，那么立即获得页p的页框号p’，并与位移d一起拼装得到访问内存的实地址，从而完成了地址转换。

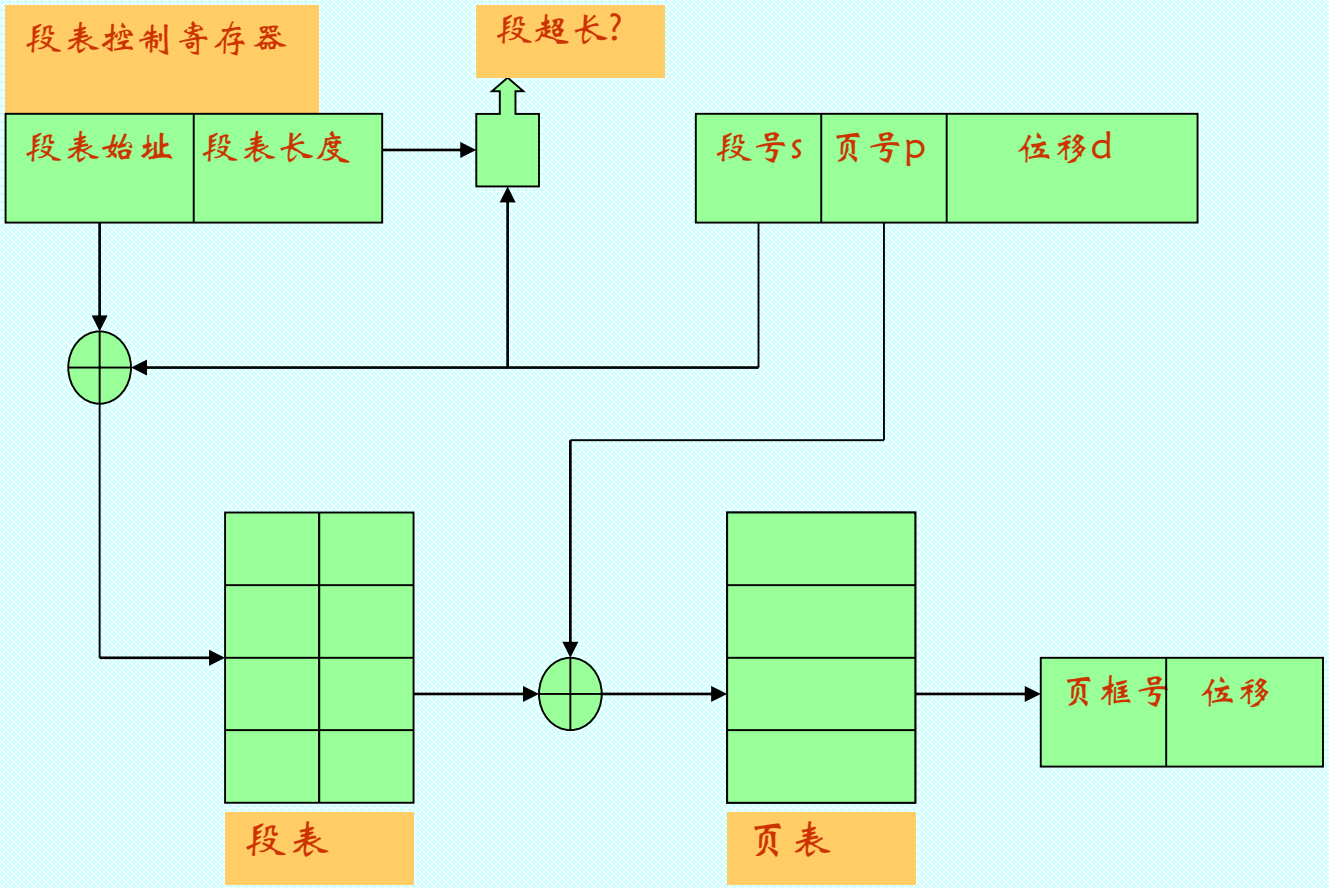
地址转换(2)

- 若查快表失败,就要通过段表和页表作地址转换了，用段号s作索引，找到相应表目，由此得到s段的页表起址s’，再以p作索引得到s段p页对应的表目,得到页框号p’；这时一方面把s段p页和页框号p’置换进快表，另一方面用p’和d生成内存实地址，从而完成地址转换。

地址转换(3)

- 如查段表时，发现s段不在内存，产生“缺段中断”，引起系统查找s段在外存的位置，将该段页表调入内存；
- 如查页表时，发现s段的p页不在内存，产生“缺页中断”，引起系统查找s段p页在外存的位置，并将该页调入内存，当内存已无空闲页框时，就会导致淘汰页面。

地址转换(4)



段页式虚存管理优缺点

- 优点
  - 基本上结合了段式和页式的优点，克服了两者的缺点
  - 以页架为单位分配主存，主存利用率比段式高，消除了段式的外部碎片，无紧缩问题；
  - 段页式的共享和保护实现与段式一样，比页式好，自需段表中的相应表目指向共享段在主存中页表地址即可。
  - 段页式的动态扩充实现的比段式和页式都好，既不受逻辑相邻的限制，也不受物理相邻的限制。

段页式虚存管理优缺点

- 缺点
  - 和页式一样存在页内碎片，其页内碎片比页式多。页式平均每个程序有一页有碎片，段页式则是平均每段有一页有碎片。
  - 增加了硬件成本。
  - 增加了软件复杂性和管理开销。

存储管理方案小结

<table>
<tr><th>分类</th><th>管理方案</th><th>维数</th><th>多道支持</th><th>共享内存</th><th>重定位</th><th>内存扩充</th><th colspan="2">保护机制</th><th>硬件支持</th></tr>
<tr><td rowspan="5">实存管理</td><td>单连续分区</td><td rowspan="3">一维</td><td>单道</td><td>不能</td><td>静态</td><td rowspan="5">覆盖与对换</td><td colspan="2">无</td><td>无</td></tr>
<tr><td>固定分区</td><td rowspan="4">多道</td><td rowspan="2">可以(多对界地址)</td><td>静态</td><td colspan="2" rowspan="2">界地址或键保护</td><td rowspan="2">单或多对重定位寄存器</td></tr>
<tr><td>可变分区</td><td rowspan="4">动态</td></tr>
<tr><td>分页</td><td rowspan="3">可以</td><td>页表</td><td rowspan="4">越界保护和存取控制保护</td><td rowspan="2">地址转换机制与保护机制+快表</td></tr>
<tr><td>分段</td><td>二维</td><td>段表</td></tr>
<tr><td rowspan="2">虚存管理</td><td>请求分页</td><td>一维</td><td rowspan="2">虚存(内存是外存的缓冲)</td><td>页表</td><td rowspan="2">同上+动态链接机制+中断机制</td></tr>
<tr><td>请求段页</td><td>二维</td><td>段页表</td></tr>
</table>

4.5.4 页面管理策略

- 1、页面装入策略
- 2、页面清除策略
- 3、页面分配策略
- 4、页面置换策略
- 5、缺页中断率
- 6、全局页面替换算法
- 7、局部页面替换算法

1、页面装入策略

- 请页式调度：指当进程需要访问某页面时，才将该页调入主存。
  - 问题：当进程刚启动时，会引起大量持续的缺页现象。
- 预调式调度：指在进程访问某页面之前，就预先将该页面调入主存。
  - 系统有无能力主动装入页面？
  - 采用何策略，将哪些页面装入主存？

2、页面清除策略

- 何时把一个修改过的页面写回外存储器，有两种策略：
  - 请页式清除：仅当一页被选中进行替换且内容被修改过才把它写回磁盘。
  - 预清除：对所有更改过的页面，在需要替换之前把它们都写回磁盘。

3、页面分配策略

- 系统为进程分配内存，需考虑因素：
  - ①分给进程的空间越小，同一时间处于内存的进程就越多，至少有一个进程处于就绪态的可能性就越大；
  - ②如果进程只有小部分在内存里，即使局部性很好，缺页中断率还会相当；
  - ③因程序的局部性原理，分给进程的内存超过一定限度后，再增加内存空间，不会明显降低进程的缺页中断率。

3、页面分配策略

- 固定分配
  - 进程保持页框数固定不变，称固定分配；
  - 进程创建时，根据进程类型和程序员的要求决定页框数，只要有一个缺页中断产生，进程就会有一页被替换。
- 可变分配
  - 进程分得的页框数可变, 称可变分配；
  - 进程执行的某阶段缺页率较高，说明目前局部性较差，系统可多分些页框以降低缺页率，反之说明进程目前的局部性较好，可减少分给进程的页框数。

4、页面替换策略

- 全局替换
  - 如果页面替换算法的作用范围是整个系统，称全局页面替换算法，它可以在运行进程间动态地分配页框。
- 局部替换
  - 如果页面替换算法的作用范围局限于本进程，称为局部页面替换算法，它实际上需要为每个进程分配固定的页框。

固定分配和局部替换策略配合使用

- 进程分得的页框数不变，发生缺页中断，只能从进程的页面中选页替换，保证进程的页框总数不变。
- 策略难点：应给每个进程分配多少页框?给少了，缺页中断率高；给多了，使内存中能同时执行的进程数减少，进而造成处理器和其它设备空闲。

固定分配和局部替换策略配合使用

- 采用固定分配算法，系统把页框分配给进程，采用：
  - ①平均分配
  - ②按比例分配
  - ③优先权分配

可变分配和全局替换策略配合使用

- 先每个进程分配一定数目页框，os保留若干空闲页框，进程发生缺页中断时，从系统空闲页框中选一个给进程,这样产生缺页中断进程的内存空间会逐渐增大，有助于减少系统的缺页中断次数。
- 系统拥有的空闲页框耗尽时，会从内存中选择一页淘汰，该页可以是内存中任一进程的页面，这样又会使那个进程的页框数减少，缺页中断率上升。
- 典型应用：SVR4

可变分配和局部替换配合使用

- (1)新进程装入内存时，根据应用类型、程序要求，分配给一定数目页框，可用请页式或预调式完成这个分配。
- (2)产生缺页中断时，从该进程驻留集中选一个页面替换。
- (3)不时重新评价进程的分配，增加或减少分配给进程的页框以改善系统性能。
- (4)典型应用：Windows NT

5、缺页中断率

- 页面替换
- 页面淘汰算法
- “抖动”（Thrashing）现象
- （注：Linux系统中，缺页中断也称缺页异常）

影响缺页中断率的因素

- 假定作业p共计n页，系统分配给它的内存块只有m块（1≤m≤n）。如果作业p在运行中成功的访问次数为s，不成功的访问次数为F，则总的访问次数A为：

$$A = S + F$$

- 缺页中断率：

$$f = F / A$$

- 影响缺页中断率f的因素有：
  - 内存页框数。
  - 页面大小。
  - 页面替换算法。
  - 程序特性。

程序局部性例子

```
A)  for j=1 TO  100  DO
        for j:=1   TO  100  DO
          A[i, j]:=0;
```

```
B) for j:=1  TO  100   DO
        for i:=1   T0  100    DO
    A[i,j]:=0
```

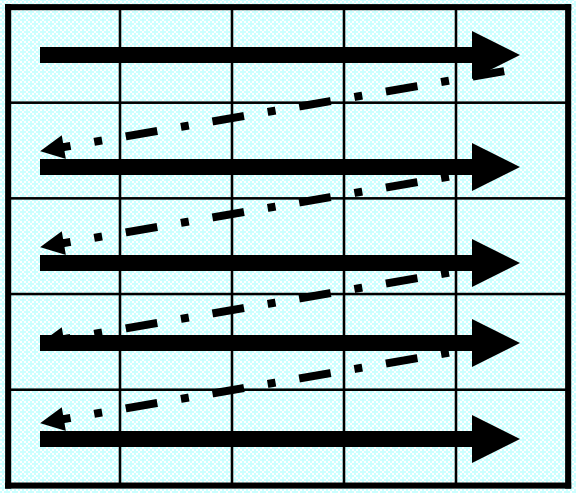
- 程序将数组置为“0”，进程有3页内存空间， 1页存放程序， 2页存放数据，页面大小可容纳200个数；按A多少次缺页中断，按B多少次缺页中断

解：数组A[i，j]，10000个数，存放顺序

A程序，按行进行，一页存放200个数，调入一页后，全部置0即可

10000/200=50次缺页中断

B按列进行，调入一页后，只访问2个数，置0，10000/2=5000次缺页中断



6、全局页面替换策略

- 1) 最佳页面替换算法OPT
- 2) 先进先出页面替换算法FIFO
- 3) 最近最少用页面替换算法LRU
- 4) 第二次机会页面替换算法SCR
- 5) 时钟页面替换算法Clock

1)最佳页面替换算法

- 基本思想：选择“将来不再使用的”或“在最远的将来才被访问的”页面被置换。
- 特点：
  - 无法实现，因为无法预知未来页面的访问情况
  - 常用作评价其他算法的依据

2)先进先出页面替换算法

- 基于程序总是按线性顺序来访问物理空间这一假设。
- 算法淘汰最先调入内存的页，或者说在内存中驻留时间最长的页。
- 实现技术
  - (1)设置具有m个元素的页号表，控制换页P［0］，P［1］， …，P［m-1］
  - (2)引入指针链成队列

FIFO 调度算法的Belady 异常现象

- 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，有时就会出现增加可用物理页框数量反而会导致缺页率增加的异常现象。
- Belady现象的原因：FIFO算法的置换特征与进程访问内存的动态特征是矛盾的，即被置换的页面并不是进程不会访问的。
- 今有5个页面的访问序列为：

  4, 3, 2, 1, 4, 3, 5, 4,3, 2, 1, 5。
- 现在分配给进程的物理页框为3个和4个，其执行过程分别如下。

Belady 异常现象举例

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

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

页面缓冲算法

- 维护两个FIFO队列，修改页面队列和非修改(空闲)页面队列，前者是由修改页面的页框构成的链表；后者是由可直接用于装入页面的页框构成的链表，未修改的淘汰页暂时还留在其中，当进程再次访问这些页面时，可不经I/O而快速找回。
- 按FIFO选出淘汰页，并不立即抛弃它，根据它的内容是否被修改过进入两个队列之一的末尾，需要装入的页面被读进非修改队列的队首指向的页框中，不必等待淘汰页写回，使得进程能快速恢复运行。

3)最近最少用页面替换算法

- 算法淘汰的页面是在最近一段时间里较久未被访问的那页。
- 算法依据：根据程序局部性原理，那些刚被使用过的页面，可能马上还要被使用，而较长时间未被使用的页面，可能不会马上需要使用。

LRU算法实现：页面淘汰队列(1)

- 队列中存放当前在内存中的页号，每当访问一页时就调整一次，使队列尾总指向最近访问的页，队列头就是最近最少用的页。
- 发生缺页中断时总淘汰队列头所指示的页；执行一次页面访问后，需要从队列中把该页调整到队列尾。

LRU算法实现：页面淘汰队列(2)

- 例子：给某作业分配了三块内存，该作业依次访问的页号为：4，3，0，4，1，1，2，3，2。当访问这些页时，页面淘汰序列变化情况如下

LRU算法实现：页面淘汰队列(3)

访问页号	页面淘汰序列	被淘汰页面
4	4	
3	4 3	
0	4 3 0	
4	3 0 4	
1	0 4 1	3
1	0 4 1	
2	4 1 2	0
3	1 2 3	4
2	1 3 2	

LRU算法实现：引用位法

- 每页设置一个引用位R，访问某页时，由硬件将页标志位R置1，隔一定时间t将所有页的标志R均清0。
- 发生缺页中断时，从标志位R为0的页中挑选一页淘汰。挑选到要淘汰的页后，也将所有页的标志位R清0。
- 又称最近未使用页面替换算法（Not Recently Used replacement，NRU）

LRU算法实现：计数法

- 每个页面设置一个多位计数器，又叫最不常用页面替换算法LFU。每当访问一页时，就使它对应的计数器加 1 。
- 当发生缺页中断时，可选择计数值最小的对应页面淘汰，并将所有计数器全部清 0 。

LRU算法实现：计时法

- 为每个页面设置一个多位计时器，每当页面被访问时，系统的绝对时间记入计时器。
- 比较各页面的计时器的值，选最小值的未使用的页面淘汰，因为，它是最“老”的未使用的页面。

LRU算法实现：老化算法

- 为每个页设置一个多位寄存器R。当页面被访问时，对应寄存器的最左边位置1；每隔时间t，将R寄存器右移一位；发生缺页中断时，找最小数值的R寄存器对应的页面淘汰。
- 举例说明，应淘汰的页为P2。

页号	时 刻		
	T1	T2	T3
P0	1000	0100	1010
P1	1000	1100	0110
P2	0000	1000	0100

4)第二次机会页面替换算法

- 改进FIFO算法，把FIFO与页表中的“引用位”结合起来使用：
- 检查FIFO中的队首页面(最早进入内存页面)，如果它的“引用位”是0，这个页面既老又没有用，选择该页面淘汰；
- 如果“引用位”是1，说明它进入内存较早，但最近仍在使用。把它的“引用位”清0，并把这个页面移到队尾，把它看作是一个新调入的页；
- 算法含义：最先进入内存的页面，如果最近还在被使用的话，仍然有机会作为像一个新调入页面一样留在内存中。

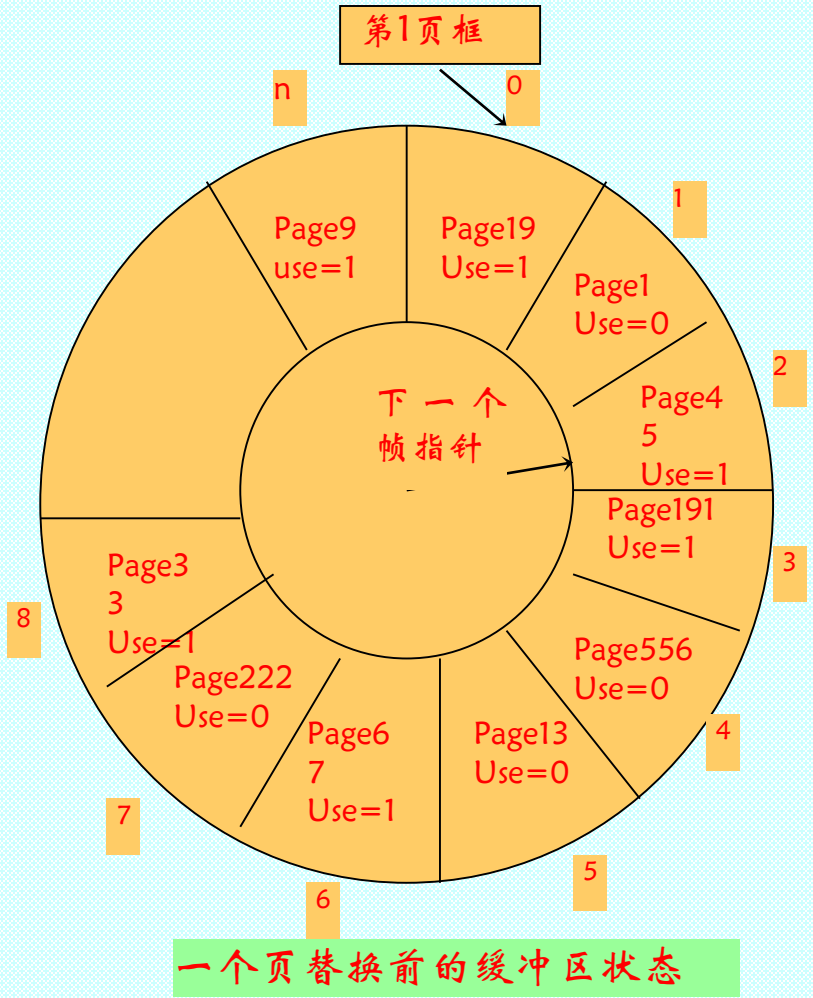
5）时钟页面替换算法(1)

- 改善二次机会置换算法，改善算法开销。
- 实现方式与二次机会置换基本一样，只不过将线形链表构成一个环形链表。
- 算法实现要点(1)：
  - 一个页面首次装入内存，其“引用位”置1 。
  - 内存中的任何页面被访问时，“引用位”置1。
  - 淘汰页面时,从指针当前指向的页面开始扫描循环队列，把遇到的“引用位”是1的页面的“引用位”清0,跳过这个页面；把所遇到的“引用位”是0的页面淘汰掉,指针推进一步。

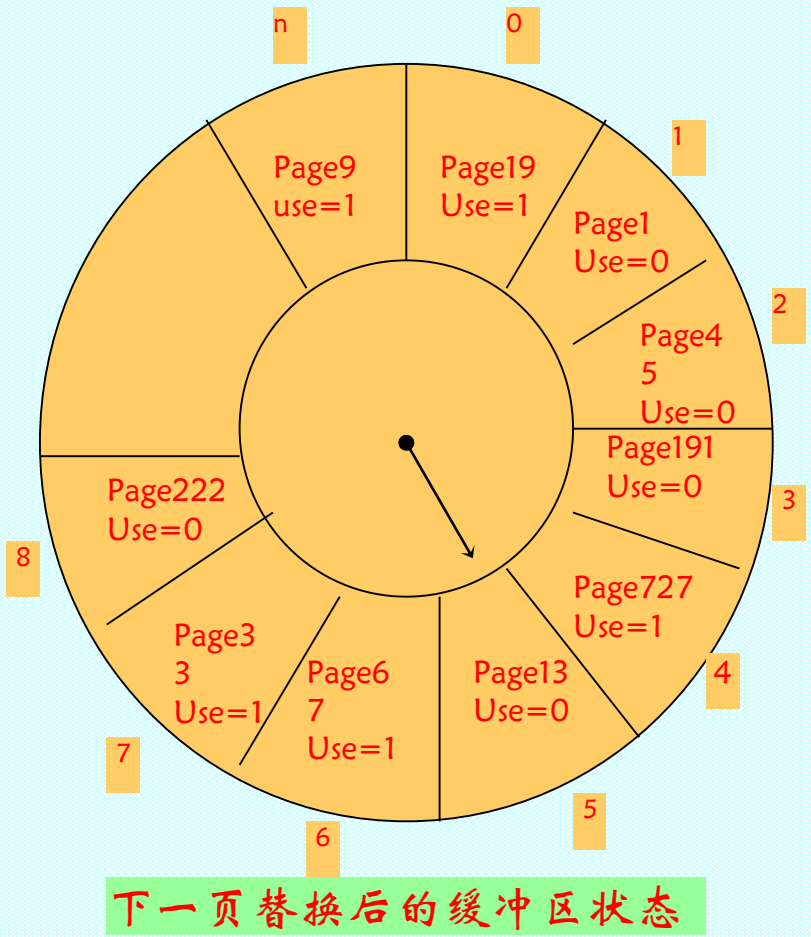
时钟页面替换算法(2)

- 算法实现要点(2)：
  - 扫描循环队列时，如果遇到的所有页面的“引用位”为1，指针就会绕整个循环队列一圈，把碰到的所有页面的“引用位”清0；指针停在起始位置，并淘汰掉这一页，然后，指针推进一步。

时钟页面替换算法的一个例子



一个页替换前的缓冲区状态



下一页替换后的缓冲区状态

时钟页面替换改进算法(1)

- 把“引用位”和“修改位”结合起来使用，共组合成四种情况：
  - (1)最近没有被引用,没有被修改(r=0,m=0)
  - (2)最近被引用,没有被修改(r=1,m=0)
  - (3)最近没有被引用,但被修改(r=0,m=1)
  - (4)最近被引用过,也被修改过(r=1,m=1)

时钟页面替换改进算法(2)

- 步1：选择最佳淘汰页面，从指针当前位置开始，扫描循环队列。扫描过程中不改变“引用位”，把遇的第一个r=0,m=0的页面作为淘汰页面。
- 步2：如果步1失败，再次从原位置开始，查找r=0且m=1的页面，把遇到的第一个这样的页面作为淘汰页面，而在扫描过程中把指针所扫过的页面的“引用位”r置0。
- 步3：如果步2失败，指针再次回到了起始位置，由于此时所有页面的“引用位”r均已为0，再转向步1操作，必要时再做步2操作，这次一定可以挑出一个可淘汰的页面。

例子--计算缺页中断次数和被淘汰页面(1)

- 假设采用固定分配策略，进程分得三个页框，执行中按下列次序引用5个独立的页面：

  2　3　2　1　5　2　4　5　3　2　5　2。

例子：计算缺页中断次数和被淘汰页面(2)

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

- OPT

F(1) F(2) F(4)

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

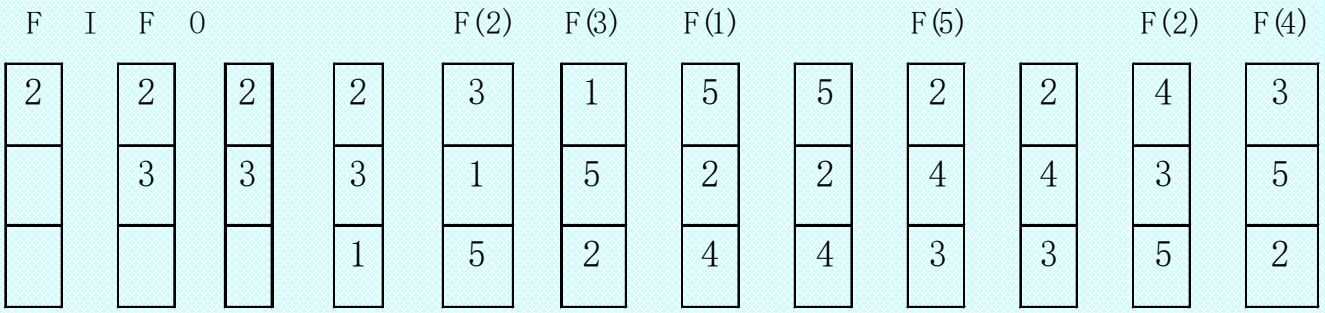
- LRU

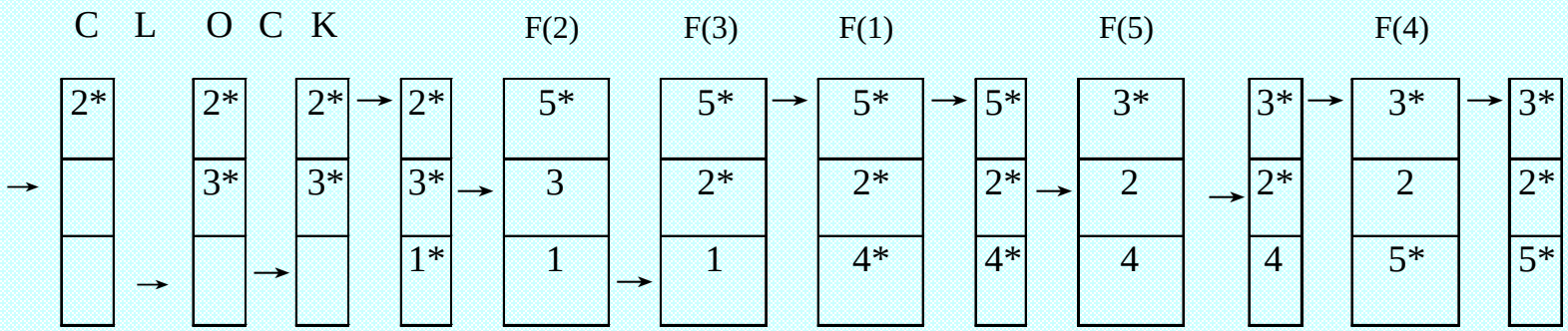
F(3) F(1) F(2) F(4)

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

例子--计算缺页中断次数和被淘汰页面(3)

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





例子--计算缺页中断次数和被淘汰页面(4)

性能比较

- OPT　　F(1)　F(2)　F(4)　　,6次
- LRU　　F(3)　F(1)　F(2)　F(4)　　,7次
- CLOCK　F(2)　F(3)　F(1)　F(5)　F(4)　　,8次
- FIFO　　F(1)　F(3)　F(1)　F(5)　F(2)　F(4)　,9次

7、局部页面替换算法

1）局部最佳页面替换算法

2）工作集模型和工作集置换算法

3）模拟工作集替换算法

4）缺页频率替换算法

1) 局部最佳页面替换算法(1)

- 实现思想：进程在时刻t访问某页面，如果该页面不在内存中，导致一次缺页，把该页面装入一个空闲页框。
- 不论发生缺页与否，算法在每一步要考虑引用串，如果该页面在时间间隔(t, t+ τ)内未被再次引用，那么就移出；否则，该页被保留在进程驻留集中。
- τ 为一个系统常量，间隔(t, t+ τ)称作滑动窗口 。例子中 τ =3。

局部最佳页面替换算法(2)

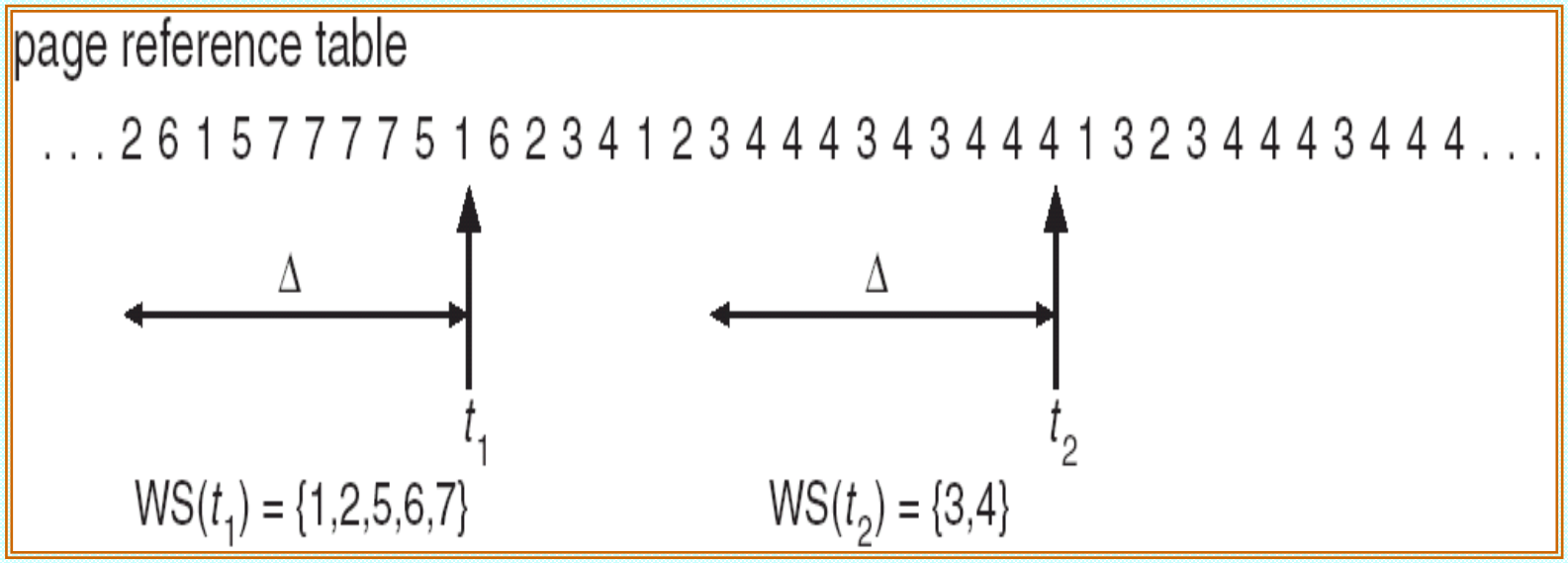
时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串	p4	p3	p3	p4	p2	p3	p5	p3	p5	p1	p4
p1	—	—	—	—	—	—	—	—	—	√	—
p2	—	—	—	—	√	—	—	—	—	—	—
p3	—	√	√	√	√	√	√	√	—	—	—
p4	√	√	√	√	—	—	—	—	—	—	√
p5	—	—	—	—	—	—	√	√	√	—	—
In t		p3			p2		p5			p1	p4
OUT t					p4	p2			p3	p5	p1

局部最佳页面替换算法示例

局部最佳页面替换算法(3)

时刻	驻留集		Outi	滑动窗口
	已驻留	Ini		
T0	P4			(0,0+3)看到p4
T1	P4	P3		(1,1+3)看到p3, p4
T2	P3,p4			(2,2+3)看到p3, p4
T3	P3,p4			(3,3+3)看到p3, p4
T4	P3	P2	p4	(4,4+3)中看不到p4
T5	P3		P2	(5,5+3)中看不到p2
T6	P3	P5		(6,6+3)看到p3, p5
T7	P3, P5			(7,7+3)看到p3, p5
T8	P5		P3	(8,8+3)中看不到p3
T9		P1	P5	(9,9+3)中看不到p5
T10		P4	P1	(10,10+3)中看不到p1

工作集例子


page reference table
. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .
Δ
Δ
t_1
t_2
WS(t_1) = {1,2,5,6,7}
WS(t_2) = {3,4}

2) 工作集模型和工作集置换算法

- 进程工作集指“在某一段时间间隔内进程运行所需访问的页面集合”。
- 实现思想：工作集模型用来对局部最佳页面替换算法进行模拟实现，不向前查看页面引用串，而是基于程序局部性原理向后看。
- 任何给定时刻，进程不久的将来所需内存页框数，可通过考查其过去最近的时间内的内存需求做出估计。

进程工作集

- 指“在某一段时间间隔内进程运行所需访问的页面集合”，W(t，Δ)表示在时刻t-Δ到时刻t之间((t-Δ，t))所访问的页面集合，进程在时刻t的工作集。
- Δ是系统定义的一个常量。变量Δ称为“工作集窗口尺寸”，可通过窗口来观察进程行为，还把工作集中所包含的页面数目称为“工作集尺寸”。
- Δ=3。

工作集替换示例

时刻t =	0	1	2	3	4	5	6	7	8	9	10
引用串	p1	p3	p3	p4	p2	p3	p5	p3	p5	p1	p4
p1	√	√	√	√	—	—	—	—	—	√	√
p2	—	—	—	—	√	√	√	√	—	—	—
p3	—	√	√	√	√	√	√	√	√	√	√
p4	√	√	√	√	√	√	√	—	—	—	√
p5	√	√	—	—	—	—	√	√	√	√	√
Int		p3			p2		p5			p1	p4
OUT t			p5		p1			p4	p2		

工作集替换示例进一步说明

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
引用串	p5	p4	p1	p3	p3	p4	p2	p3	p5	p3	p5	p1	p4
p1			√	√	√	√	—	—	—	—	—	√	√
p2			—	—	—	—	√	√	√	√	—	—	—
p3			—	√	√	√	√	√	√	√	√	√	√
p4		√	√	√	√	√	√	√	√	—	—	—	√
p5	√	√	√	√	—	—	—	—	√	√	√	√	√
In_t				p3			p2		p5			p1	p4
OUT_t					p5		p1			p4	p2		

工作集页面替换算法

时刻	工作集		Outi	(t-Δ，t)= (t-3，t)
	已驻留	Ini		
T-2		P5		
T-1	P5	P4		
T0	P4,p5	P1		
T1	P1,P4,p5	P3		(1-3,1)看到p1, p3 p4, p5
T2	P1,P3,p4		p5	(2-3,2)看到p1, p3, p4。 P5出。
T3	P1,P3,p4			(3-3,3)看到p1, p3, p4
T4	P3,p4	P2	P1	(4-3,4)看到p2, p3, p4。 P1出。
T5	P2,P3, P4			(5-3,5)看到p2, p3, p4
T6	P2,P3, P4	P5		(6-3,6)看到p2, p3, p4, p5
T7	P2,P3, P5		P4	(7-3,7)看到p2, p3, p5。 P4出。
T8	P3, P5		P2	(8-3,8)看到p3, p5。 P2出。
T9	P3, P5	P1		(9-3,9)看到p1, p3, P5
T10	P1,P3, P5	P4		(10-3,10)看到p1, p3, P54,p5

通过工作集确定驻留集大小

- (1)监视每个进程的工作集，只有属于工作集的页面才能留在内存；
- (2)定期地从进程驻留集中删去那些不在工作集中的页面；
- (3)仅当一个进程的工作集在内存时，进程才能执行。

3) 模拟工作集替换算法(1)

- 模拟工作集算法
  - 工作集策略在概念上很有吸引力，监督驻留页面变化的开销很大，估算合适的窗口Δ大小也是个难题，为此，使用工作集近似算法 。

模拟工作集替换算法(2)
时间戳算法(1)

- 为页面设置引用位及关联时间戳，通过超时中断，至少每隔若干条指令周期性地检查引用位及时间戳：
  - 当引用位为1时，就把它置0，并把这次改变的时间作为时间戳记录下来。
  - 当引用位为0时，系统当前时间减去时间戳时间，计算出从它上次使用以来未被再次访问的时间量，记入t_off；

模拟工作集替换算法(3)
时间戳算法(2)

- t_off值随着每次超时中断的处理而不断增加，除非页面在此期间被再次引用，导致其使用位为1；
- 把t_off与系统时间参数t_max相比，若t_off>t_max，就把页面从工作集中移出，释放相应页框。

举例

页面P的引用情况：

- T0　P被引用，引用位=1，则令引用位=0，并将时刻T0记入时间戳。
- T1　P未被引用，引用位=0，则t_off=T1-T0。
- T2　P未被引用，引用位=0，则t_off=T2-T0。
- t_off值越来越大，除非页面在此期间被再次引用，导致其使用位为1；
- 当t_off>t_max,则将页面P从工作集中移出。

4）缺页频率替换算法

- 缺页频率替换算法根据连续的缺页之间的时间间隔来对缺页频率进行测量，每次缺页时，利用测量时间调整进程工作集尺寸。
- 规则：如果本次缺页与前次缺页之间的时间超过临界值τ，那么，所有在这个时间间隔内没有引用的页面都被移出工作集。
- τ=2。

PFF替换示例

时刻t	0	1	2	3	4	5	6	7	8	9	10
引用串		p3	p3	p4	p2	p3	p5	p3	p5	p1	p4
p1	√	√	√	√	—	—	—	—	—	√	√
p2	—	—	—	—	√	√	√	√	√	—	—
p3	—	√	√	√	√	√	√	√	√	√	√
p4	√	√	√	√	√	√	√	√	√	—	√
p5	√	√	√	√	—	—	√	√	√	√	√
Int		p3			p2		p5			p1	p4
OUT t					p1 ,p5					p2,p4	

虚存页面替换算法小结

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