操作系统 Operating System

第三章 内存管理

3.1 内存管理基础-2

原仓周

yuancz@buaa.edu.cn


2025第4次课堂小测试

3.1内容提要

- 3.1 内存管理基础
- 3.2 页式内存管理
- 3.3 段式内存管理
- 3.4 虚拟内存管理
- 3.5 内存管理实例
- 3.6 页目录自映射

3.1内容提要

- 3.1 内存管理基础
  - 程序的链接与加载
    - 过程原理分析
      - » ELF文件格式
      - » 链接
      - » 加载
    - C程序实例分析(MIPS)
      - » 符号表和重定位表
      - » 链接
      - » 加载（含Linux)
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理

3.1内容提要

- 3.1 内存管理基础
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理

存储器硬件的分类

- RAM: Random Access Memory
  - SRAM
  - DRAM
  - SDRAM、DDR SDRAM

易失性

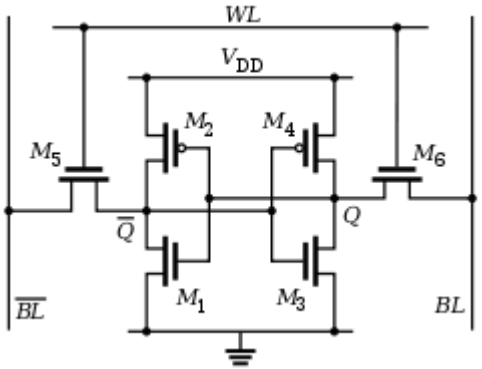
- ROM: Read Only Memory
  - PROM、EPROM、EEPROM等
- Flash memory
  - NOR
  - NAND
- Disk
- Tape

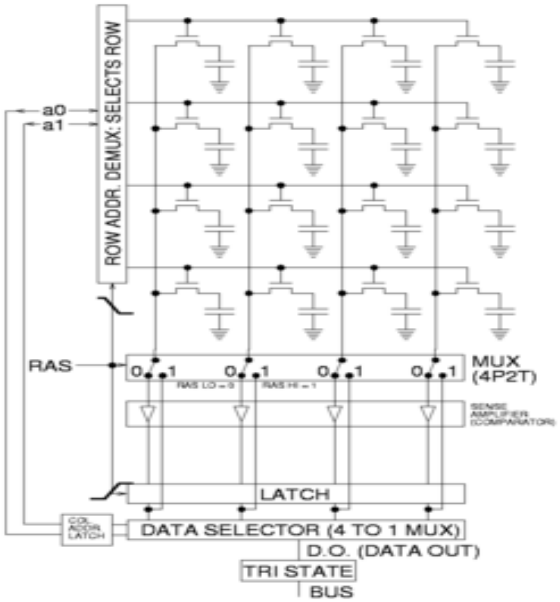
外存

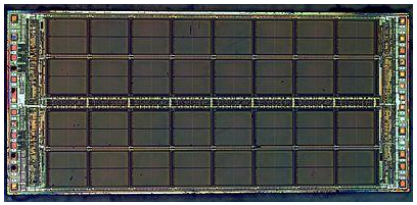
非易失性

存储器硬件的功能和发展方向

- 存储器的功能：保存数据
- 存储器的发展方向：是高速、大容量和小体积。
  - 如：内存在访问速度方面的发展：DRAM、SDRAM（DDR）、SRAM等；
    - DDR4理论上每根DIMM模块能达到512GiB的容量
    - DDR4-3200带宽可达51.2GB/s
  - 硬盘技术在大容量方面的发展：接口标准、存储密度等；







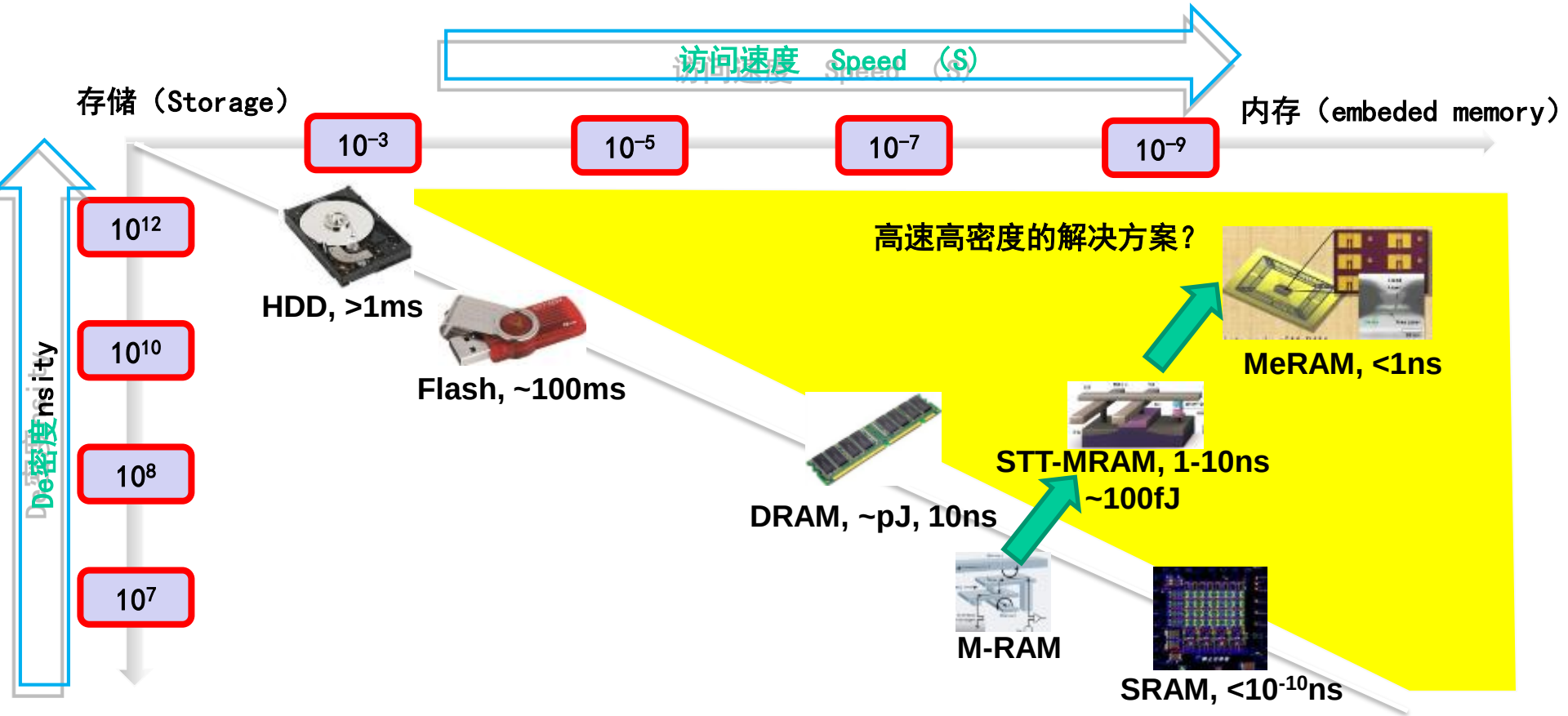
静态和动态内存

- 静态存储器（SRAM）：读写速度快，生产成本高，多用于容量较小的高速缓冲存储器。
- 动态存储器（DRAM）：读写速度较慢，集成度高，生产成本低，多用于容量较大的主存储器。

	SRAM	DRAM
存储信息方式	触发器（RS）	电容
破坏性读出	否	是
定期刷新	不需要	需要
送地址方式	行列同时送	行列分两次送
运行速度	快	慢
发热量	大	小
存储成本	高	低
集成度	低	高

蕴含巨大的挑战

- 非易失性存储器件（NVM）的发展是否有突破的机会？
- SSD（Flash）、PCM、忆阻器、自旋电子器件



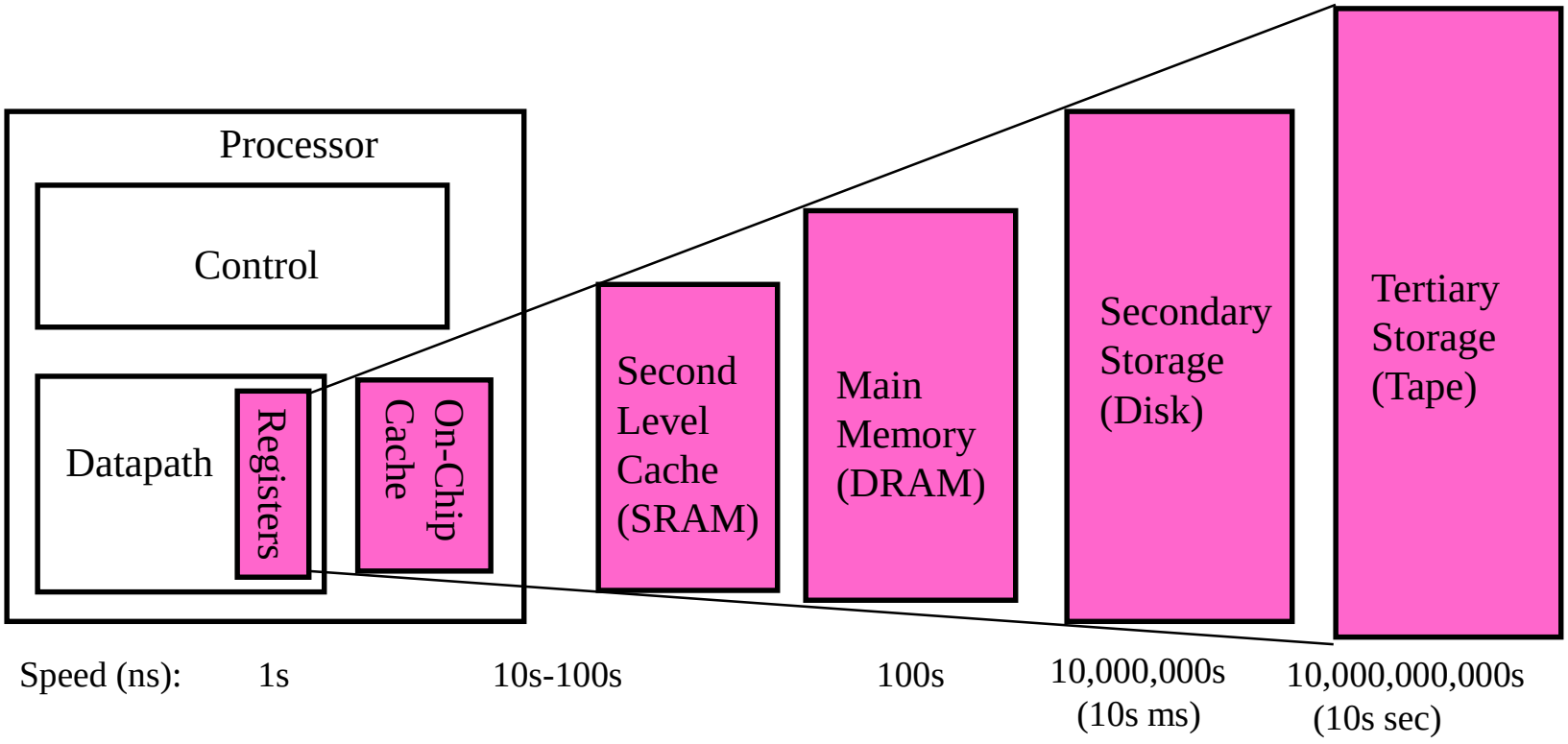
K.L.Wang, Spin orbit interaction engineering of magnetic memory for energy efficient electronics systems. NVMTS 2015

存储的组织结构

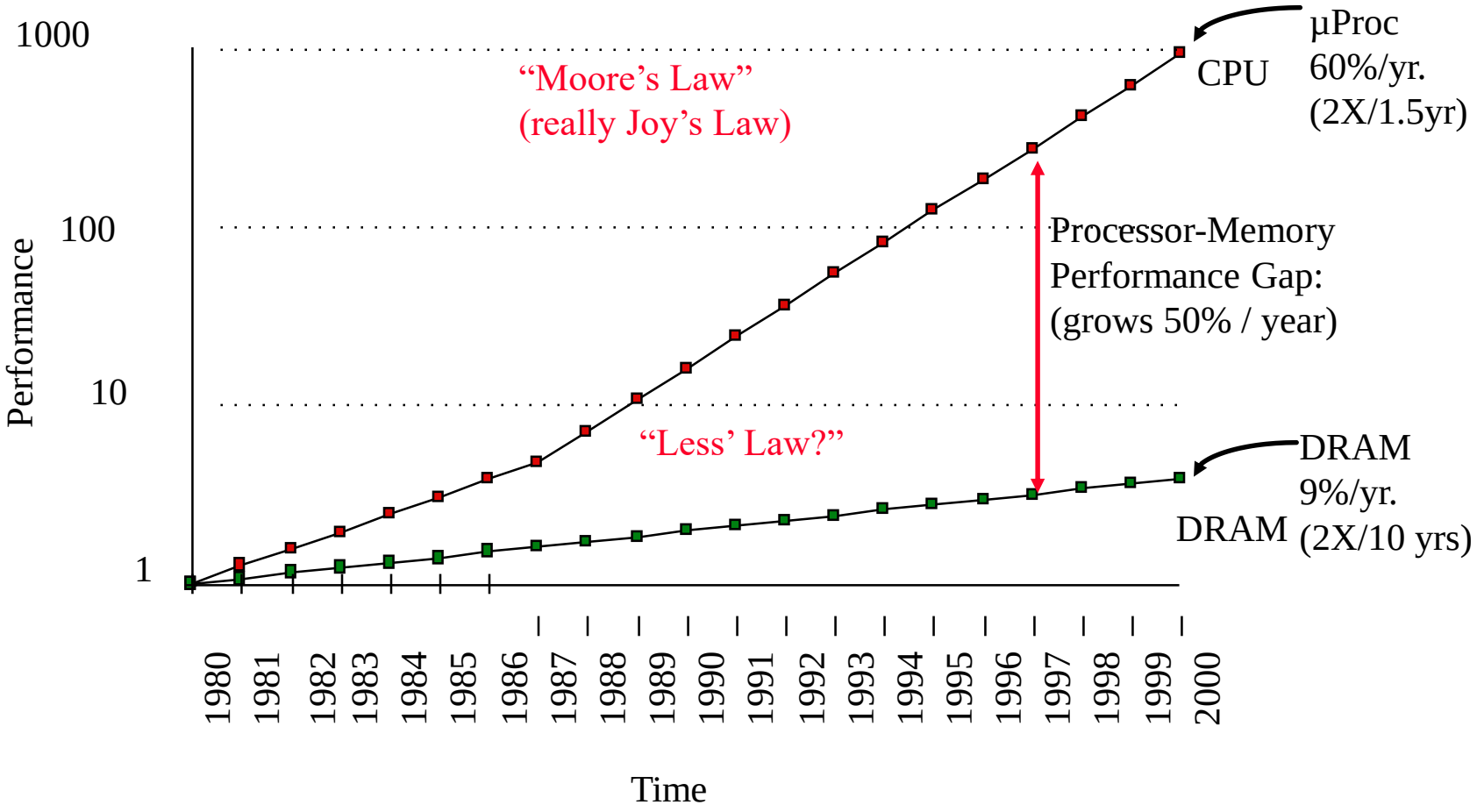
- 存储组织结构：
  - 在存储技术和CPU寻址技术许可的范围内组织合理的存储结构
- 依据是：
  - 访问速度匹配关系、容量要求和价格
    - 例如：“寄存器-内存-外存”结构
    - “寄存器-缓存-内存-外存”结构
- 典型的层次式存储组织：
  - 访问速度越来越慢，容量越来越大，价格越来越便宜
- 最佳状态应是各层次的存储器都处于均衡的繁忙状态（如：缓存命中率正好使主存读写保持繁忙）

存储层次结构


寄存器(register)
快速缓存(cache)
主存(primary storage)
外存(secondary storage)


Processor
Control
Datapath
Registers
On-Chip Cache
Second Level Cache (SRAM)
Main Memory (DRAM)
Secondary Storage (Disk)
Tertiary Storage (Tape)
Speed (ns):
1s
10s-100s
100s
10,000,000s (10s ms)
10,000,000,000s (10s sec)
Size (bytes):
100s
Ks-Ms
Ms
Gs
Ts

Processor-DRAM Memory Gap (latency)



3.1内容提要

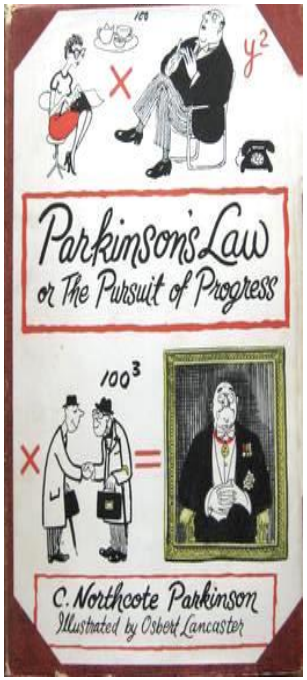
- 3.1 内存管理基础
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理

内存管理的目标

- 程序使用内存的特点：
  - 以内存为中心的存储资源
  - 程序在内存中运行
- 用户角度（程序员的目标）
  - 容量大
  - 速度快（性能）
  - 独立拥有，不受干扰（安全）
- 资源管理角度
  - 为多用户提供服务
  - 效率、利用率、能耗

帕金森定律(Parkinson)

work expands so as to fill the time available for its completion



无论存储器空间有多大，程序都能将其耗尽

内存管理的需求分析

需求：

- 从每个计算机使用者（程序员）的角度
  1. 整个空间都归我使用
  2. 不希望任何第三方因素妨碍我的程序的正常运行
- 从计算机平台提供者的角度
  - 尽可能同时为多个用户提供服务

分析：

- 计算机至少同时存在两个程序：
  - 一个用户程序和一个服务程序（操作系统）
- 每个程序具有的地址空间应该是相互独立的
- 每个程序使用的空间应该得到保护

内存管理的基石

1. 地址独立：程序发出的地址与物理地址无关
2. 地址保护：一个程序不能访问另一个程序的地址空间

内存管理的功能

- **内存分配和回收**：物理的和逻辑的
  - 讨论其算法和相应的数据结构
- **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
- **内存共享和保护**：代码和数据共享
  - 对地址空间的访问权限（读、写、执行）
- **内存扩充**：它涉及内存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），请求调入和预调入（部分进程空间）

3.1内容提要

- 3.1 内存管理基础
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理
- 3.2 页式内存管理
- 3.3 段式内存管理
- 3.4 虚拟内存管理
- 3.5 内存管理实例

3.1内容提要

- 3.1 内存管理基础
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理
      - 固定式分区
      - 可变式分区
        - 基于顺序搜索的分配算法
        - 基于索引搜索的分配算法
        - 紧凑技术
      - 多重分区
      - 内外协同，内存复用

存储分配的三种方式

- 直接指定方式
  - 程序员在编程序时,或编译程序(汇编程序)对源程序进行编译(汇编)时,所用的是实际地址
- 静态分配(Static Allocation)
  - 程序员编程时,或由编译程序产生的目的程序,均可从其地址空间的零地址开始；当装配程序对其进行连接装入时才确定它们在主存中的地址
- 动态分配(Dynamic Allocation)
  - 作业在存储空间中的位置,在其装入时确定,在其执行过程中可根据需要申请附加的存储空间,而且一个作业已占用的部分区域不再需要时,可以要求归还给系统

3.1内容提要

- 3.1 内存管理基础
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理
      - 固定式分区
      - 可变式分区
        - 基于顺序搜索的分配算法
        - 基于索引搜索的分配算法
        - 紧凑技术
      - 多重分区
      - 内外协同，内存复用

单道程序的内存管理

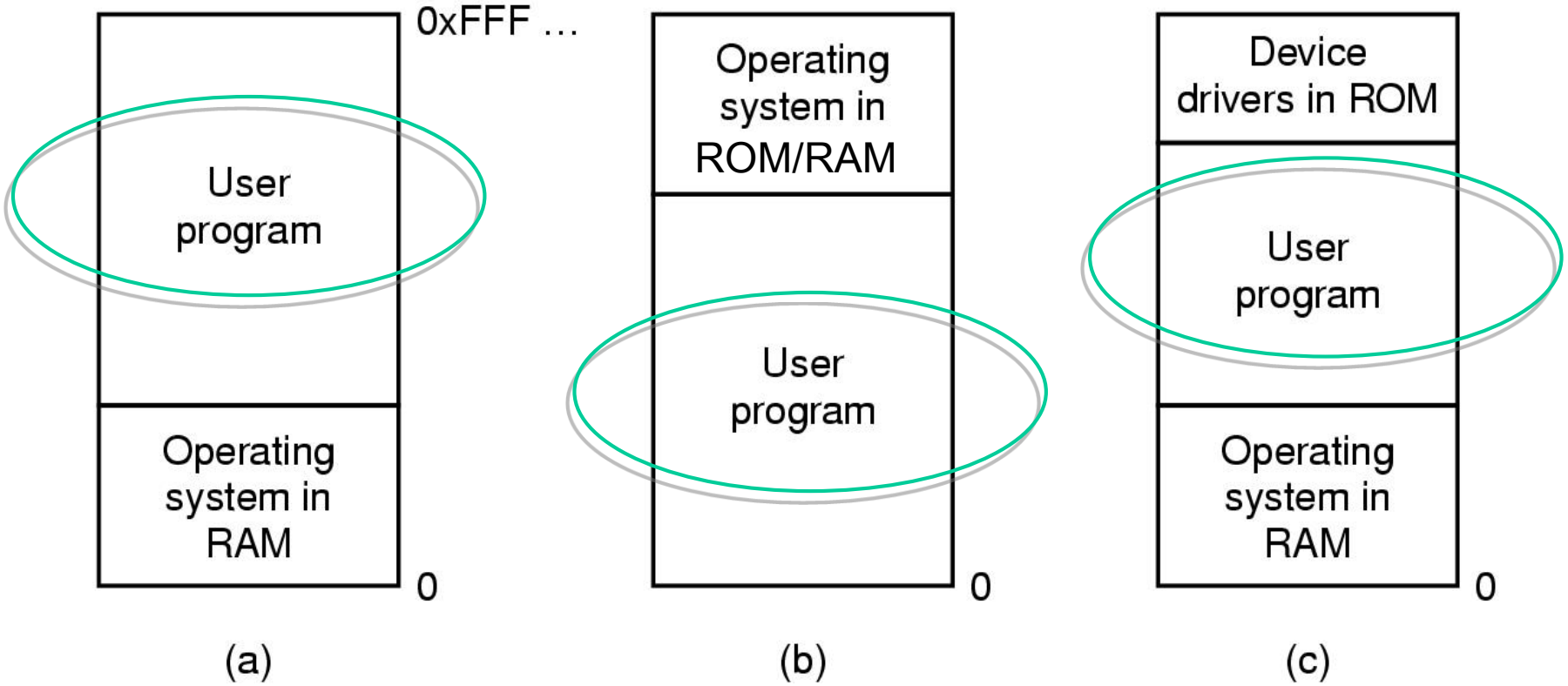
条件：

- 在单道程序环境下，整个内存里只有两个程序
  - 一个用户程序
  - 操作系统
- 操作系统所占的空间是固定的
- 因此可以将用户程序永远加载到同一个地址
  - 即用户程序永远从同一个地方开始运行

结论：

- 用户程序的地址在运行之前可以计算

操作系统在内存中的位置



单道程序的内存管理

方法：

- 静态地址翻译
  - 即在程序运行之前就计算出所有物理地址
  - 静态翻译工作可以由加载器实现

分析：

- 地址独立？YES.
  - 因为用户无需知道物理内存的相关知识
- 地址保护？YES.
  - 因为没有其它用户程序

单道程序的内存管理

优点：

- 执行过程中无需任何地址翻译工作，程序运行速度快

缺点：

- 比物理内存大的程序无法加载，因而无法运行
- 造成资源浪费
  - 小程序会造成空间浪费
  - I/O时间长会造成计算资源浪费

思考：

- 程序可加载到内存中，就一定可以正常运行吗?
- 用户程序运行会影响操作系统吗?

3.1内容提要

- 3.1 内存管理基础
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理
      - 固定式分区
      - 可变式分区
        - 基于顺序搜索的分配算法
        - 基于索引搜索的分配算法
        - 紧凑技术
      - 多重分区
      - 内外协同，内存复用

多道程序的内存管理

空间的分配：分区式分配

- 把内存分为一些大小相等或不等的分区(partition)
  - 每个应用程序占用一个或几个分区
  - 操作系统占用其中一个分区
- 适用于多道程序系统和分时系统
  - 支持多个程序并发执行
  - 但难以进行内存分区的共享

方法：

- 固定（静态）式分区分配，程序适应分区
- 可变（动态）式分区分配，分区适应程序

3.1内容提要

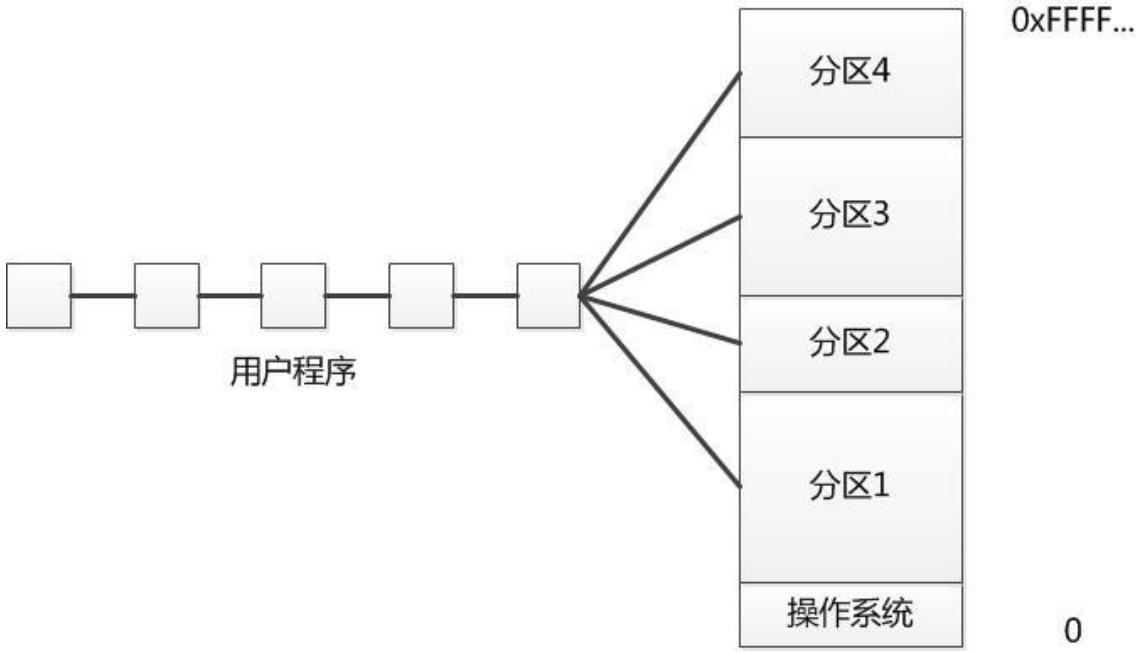
- 3.1 内存管理基础
  - 存储器硬件发展
  - 内存管理的目标
  - 内存分配方法
    - 单道程序的内存管理
    - 多道程序的内存管理
      - 固定式分区
      - 可变式分区
        - 基于顺序搜索的分配算法
        - 基于索引搜索的分配算法
        - 紧凑技术
      - 多重分区
      - 内外协同，内存复用

固定式分区

- 把内存划分为若干个固定大小的连续分区
  - 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

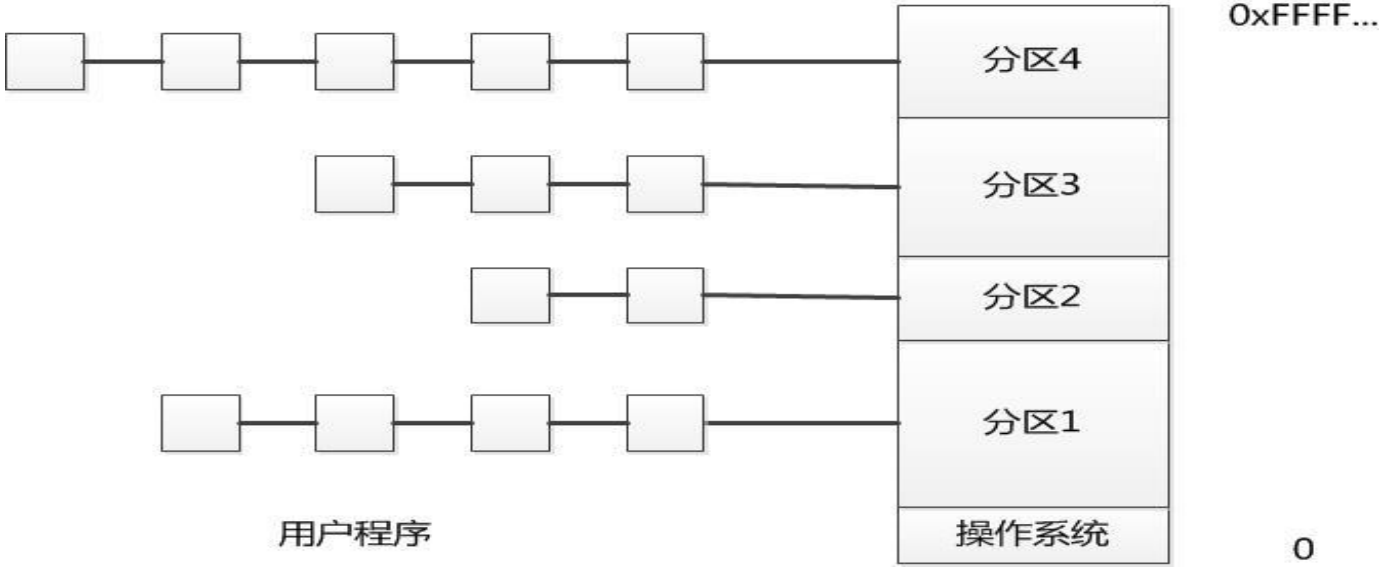
单一队列的分配方式

- 当需要加载程序时，选择一个当前闲置且容量足够大的分区进行加载，可采用共享队列的固定分区（多个用户程序排在一个共同的队列里面等待分区）分配。



多队列分配方式

- 由于程序大小和分区大小不一定匹配，有可能形成一个小程序占用一个大分区的情况
  - 从而造成内存里虽然有小分区闲置但无法加载大程序的情况
- 这时，可以采用多个队列，给每个分区一个队列，程序按照大小排在相应的队列里

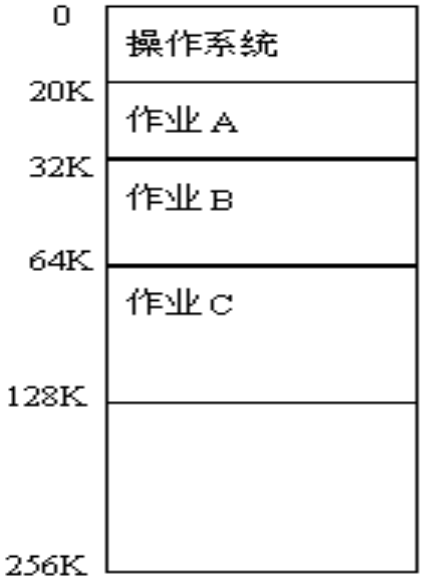


固定式分区的管理

- 优点：易于实现，开销小
- 缺点：内碎片造成浪费，分区总数固定，限制了并发执行的程序数目
- 采用的数据结构：分区表——记录分区的大小和使用情况

分区号	大小	起址	状态
1	12K	20K	已分配
2	32K	32K	已分配
3	64K	64K	已分配
4	128K	128K	未分配

分区说明表



存储空间分配情况

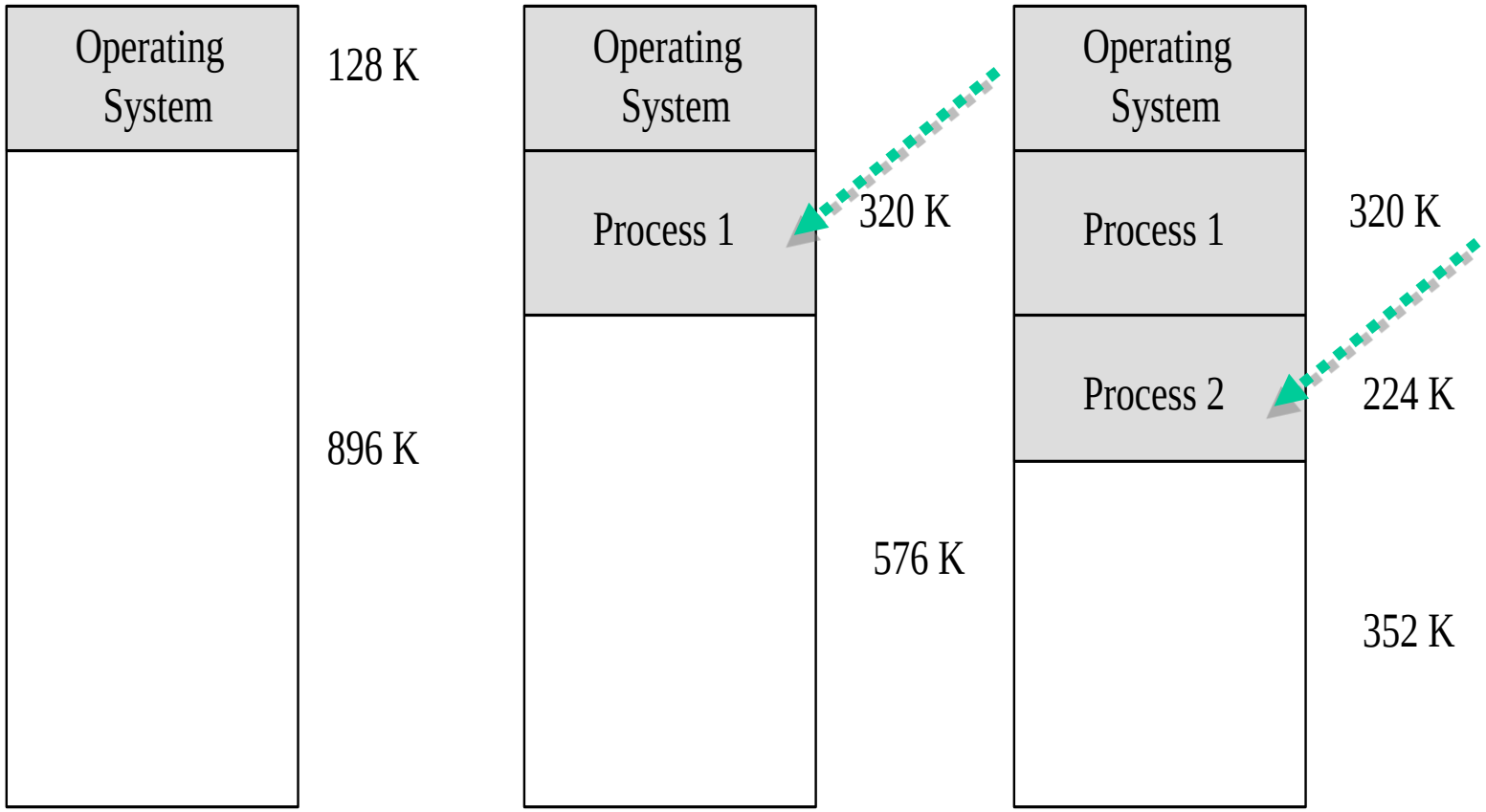
3.1内容提要

- 3.1 内存管理基础
  - 单道程序的内存管理
  - 多道程序的内存管理
    - 固定式分区
    - 可变式分区
      - » 基于顺序搜索的分配算法
      - » 基于索引搜索的分配算法
      - » 紧凑技术
    - 多重分区
      - » 界限寄存器方法
      - » 内存保护键方法
    - 内外协同，内存复用
      - » 覆盖
      - » 交换

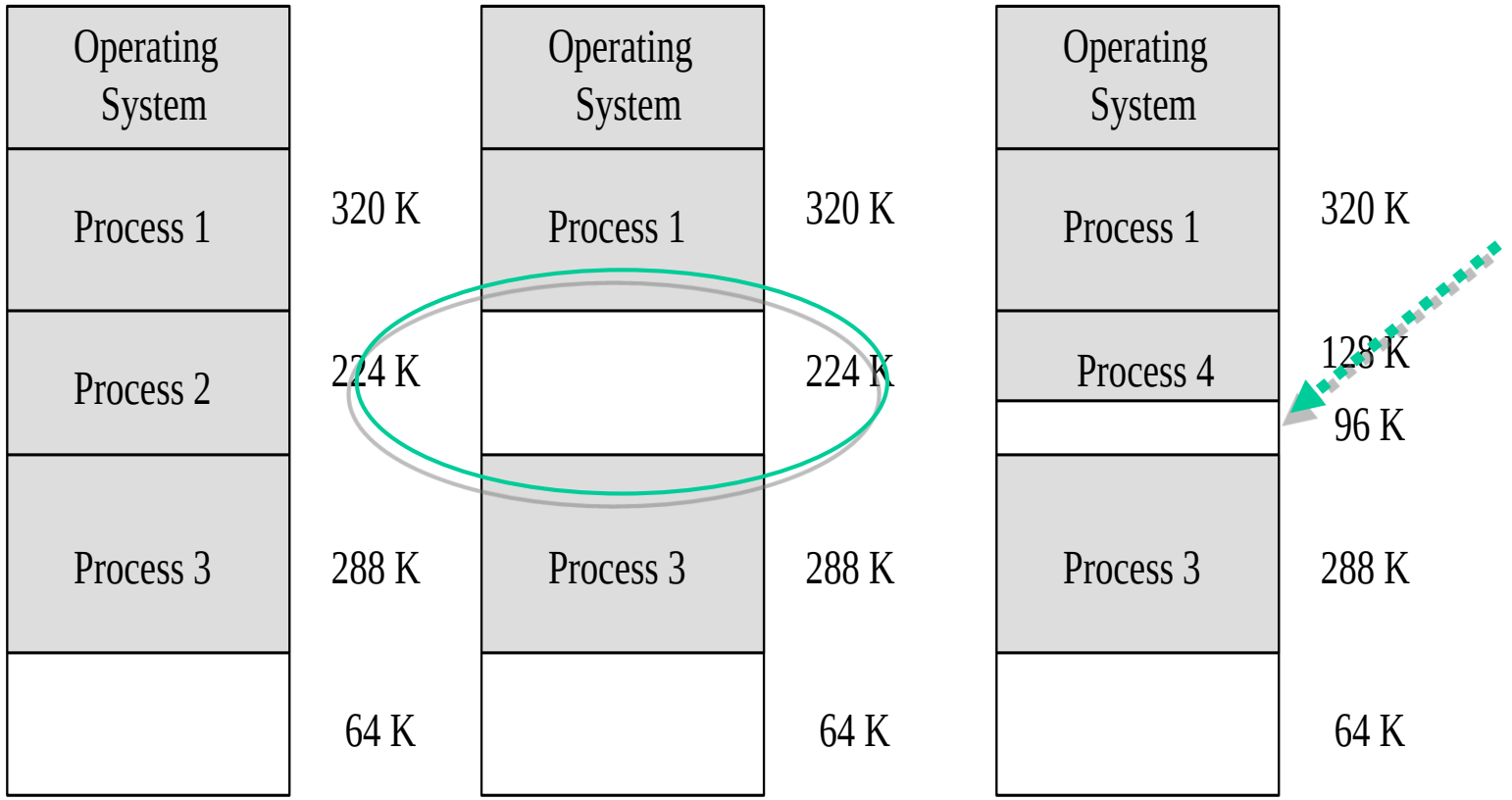
可变式分区

- 可变式分区：分区的边界可以移动
  - 即分区的大小可变
- 优点：
  - 没有内碎片
- 缺点：
  - 有外碎片

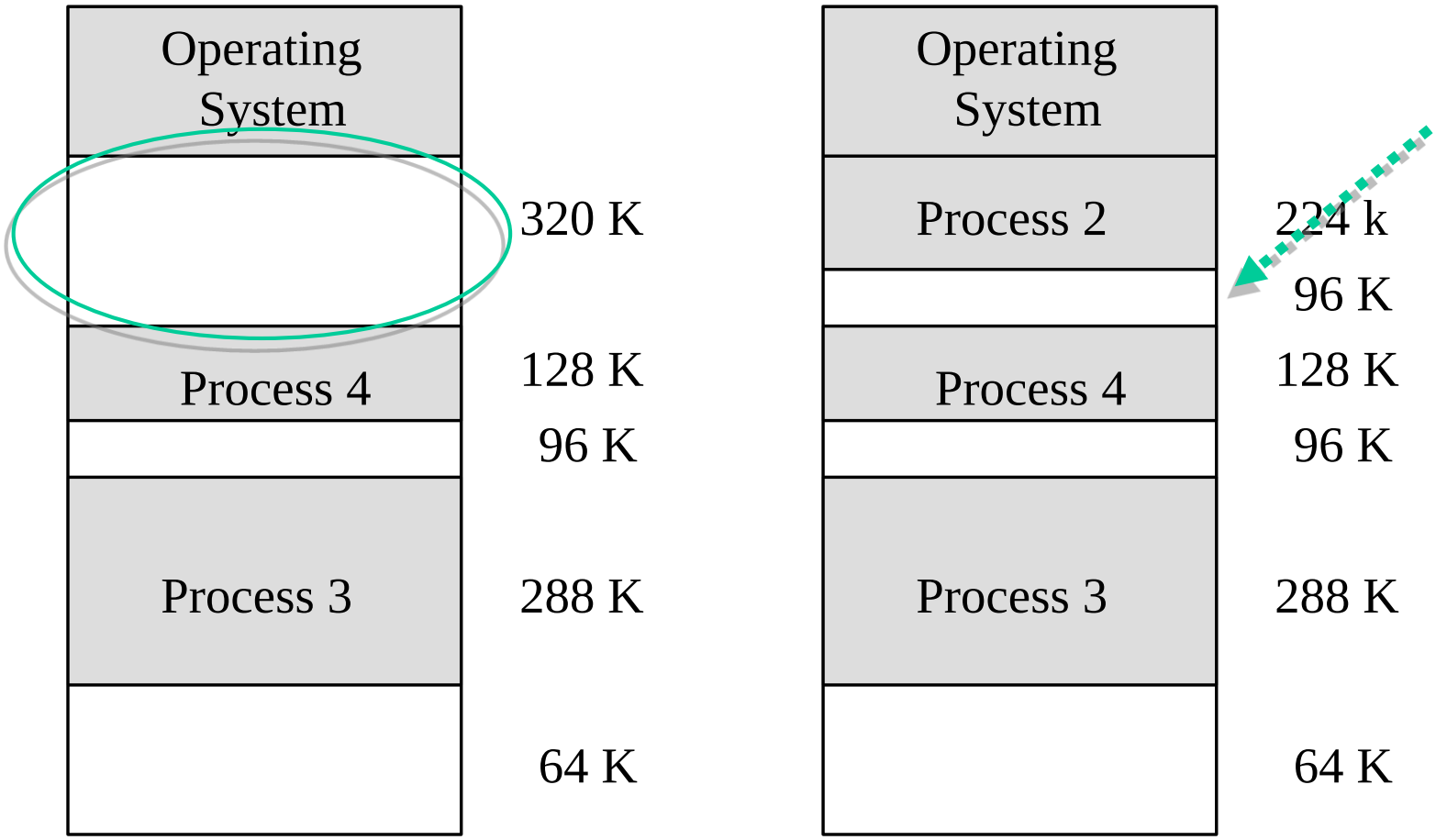
可变式分区


Operating System
128 K
896 K
Operating System
Process 1
320 K
576 K
Operating System
Process 1
320 K
Process 2
224 K
352 K

可变式分区

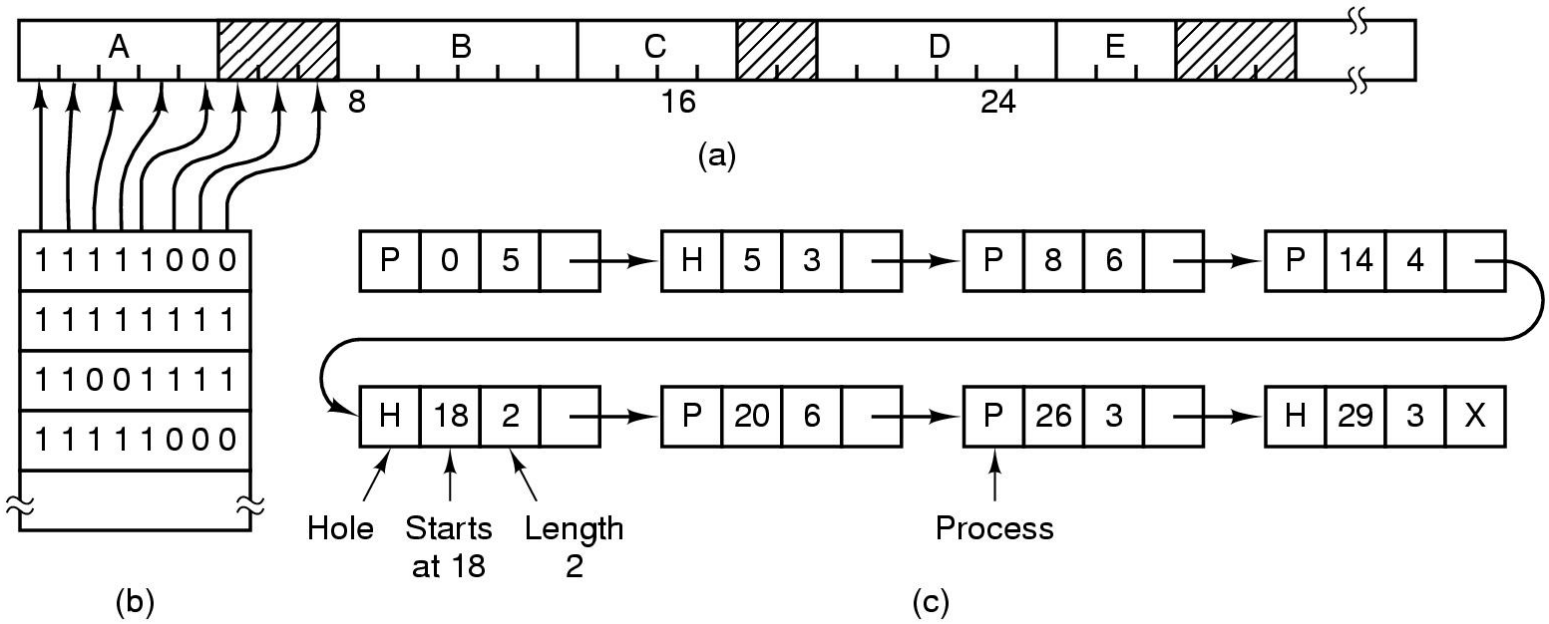

Operating System
Process 1
Process 2
Process 3
320 K
224 K
288 K
64 K
Operating System
Process 1
Process 3
320 K
224 K
288 K
64 K
Operating System
Process 1
Process 4
Process 3
320 K
128 K
96 K
288 K
64 K

可变式分区


Operating System
320 K
Process 4
128 K
96 K
Process 3
288 K
64 K
Operating System
Process 2
224 k
96 K
Process 4
128 K
96 K
Process 3
288 K
64 K

空闲空间的表示方法

- 在管理内存的时候，OS需要知道内存空间有多少空闲？跟踪内存使用的办法有两种：
  - 位图表示法（分区表）
  - 链表表示法（分区链表）



位图表示法

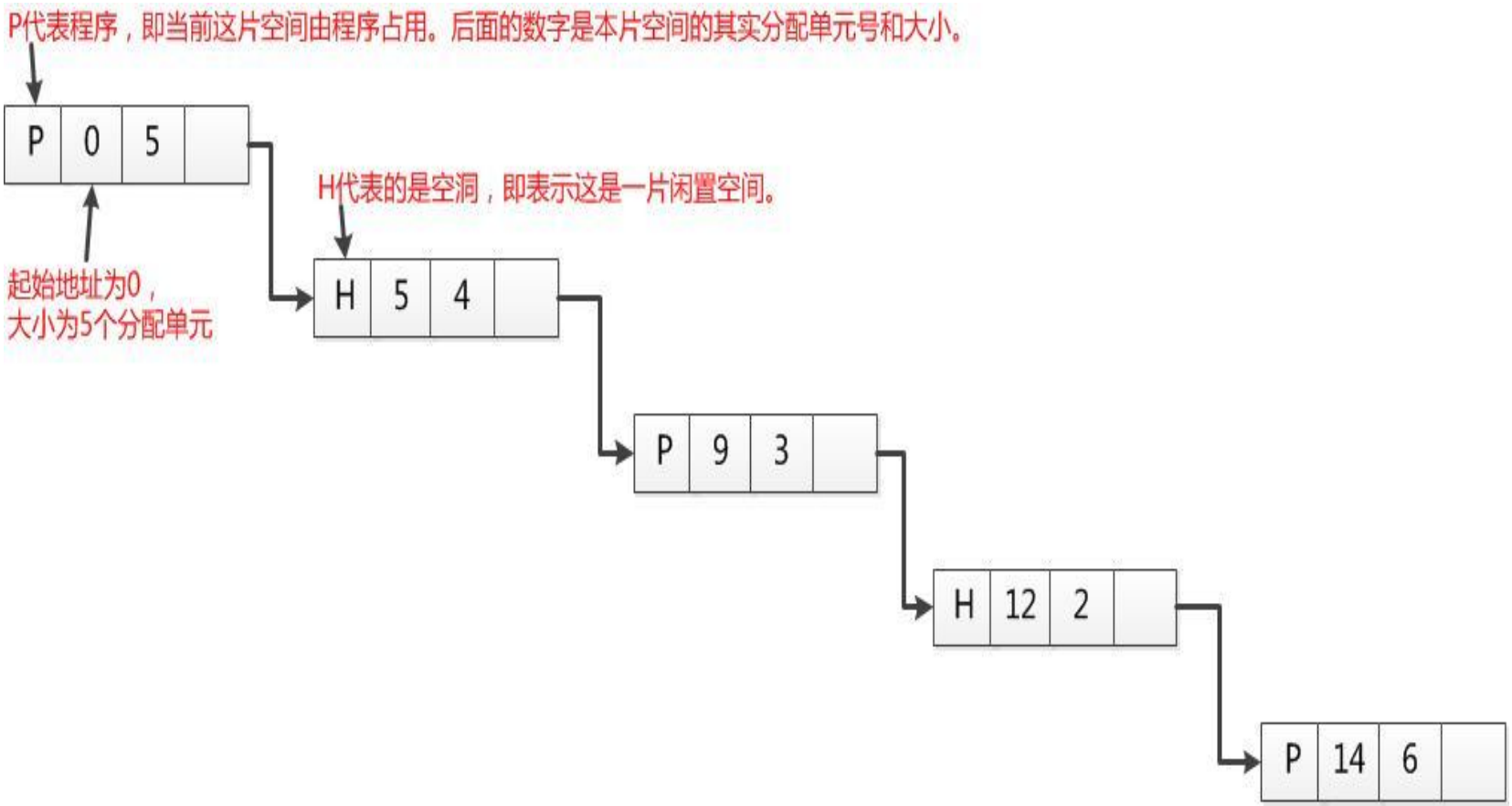
- 给每个分配单元赋予一个字位，用来记录该分配单元是否闲置
  - 例如，字位取值为0表示单元闲置，取值为1则表示已被占用，这种表示方法就是位图表示法

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

内存分配位图表示

链表表示法

- 将分配单元按照是否闲置链接起来，这种方法称为链表表示法



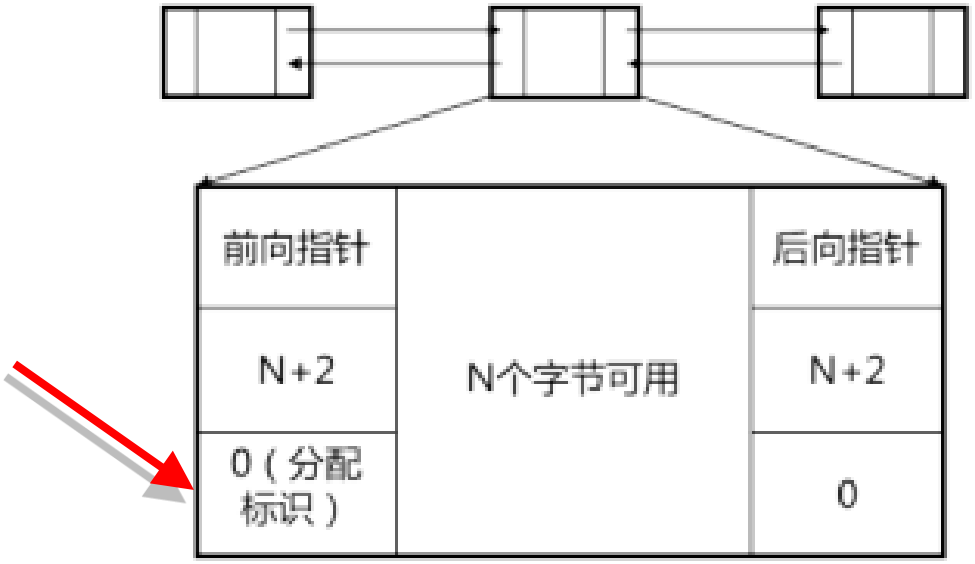
两种方法的特点

- 位图表示法：
  - 空间成本固定：不依赖于内存中的程序数量
  - 时间成本低：操作简单，直接修改其位图值即可
  - 没有容错能力：如果一个分配单元为1，不能肯定应该为1还是因错误变成1
- 链表表示法：
  - 空间成本：取决于程序的数量
  - 时间成本：链表扫描通常速度较慢，还要进行链表项的插入、删除和修改
  - 有一定容错能力：因为链表有被占空间和闲置空间的表项，可以相互验证

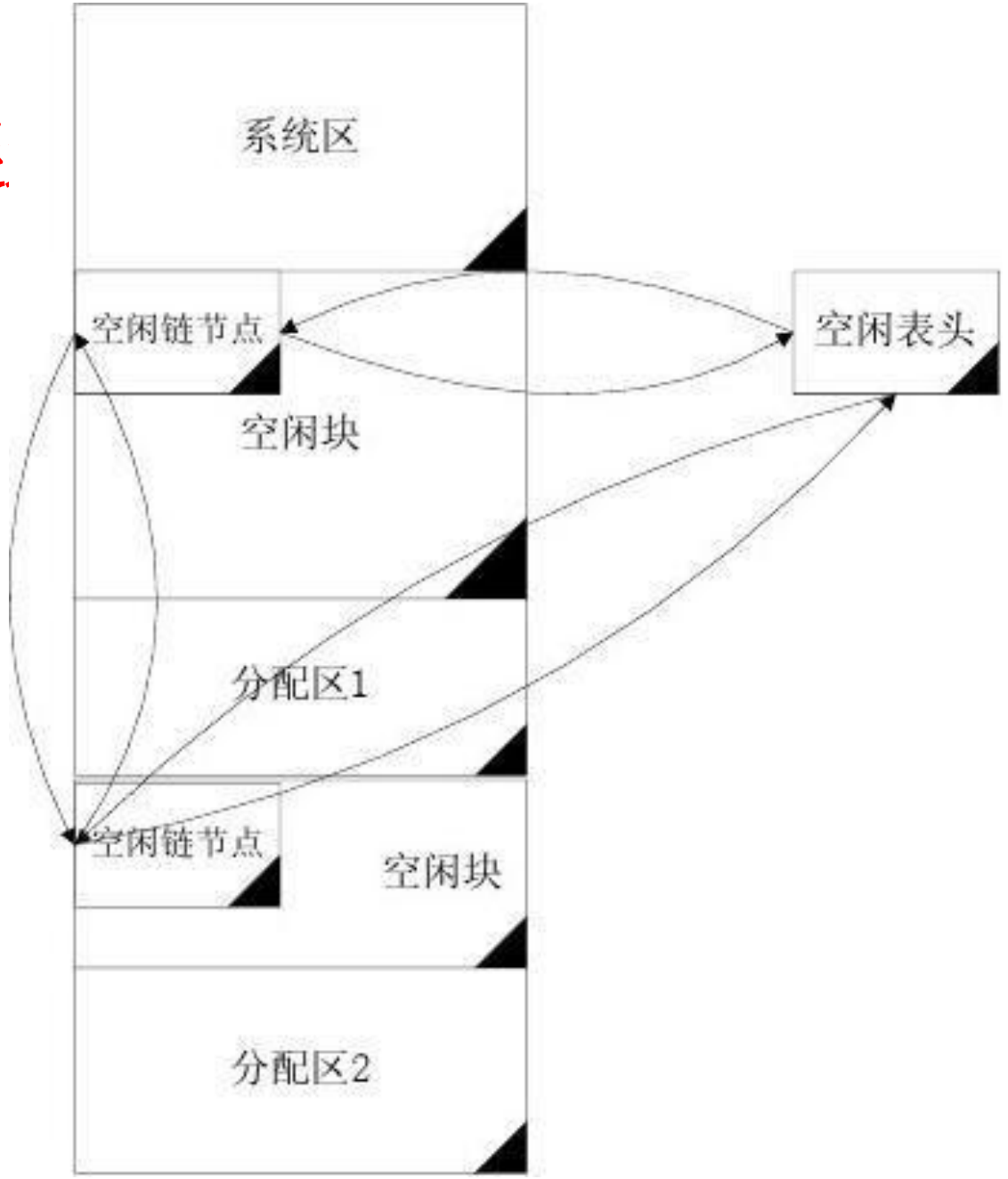
可变分区的表示方法

- 内存分配采用两张表：已分配分区表和未分配分区表
  - 每张表的表项为内存控制块MCB（Memory Control Block），包括AMCB（Allocated MCB）和FMCB（Free MCB）
- 空闲分区控制块按某种次序构成FMCB链表结构。当分区被分配出去以后，前、后向指针无意义。

分配标识：
0：未分配
1：已分配



空闲块双向链



3.1内容提要

- 3.1 内存管理基础
  - 单道程序的内存管理
  - 多道程序的内存管理
    - 固定式分区
    - 可变式分区
      - 基于顺序搜索的分配算法
      - 基于索引搜索的分配算法
      - 紧凑技术
    - 多重分区
      - 界限寄存器方法
      - 内存保护键方法
    - 内外协同，内存复用
      - 覆盖
      - 交换

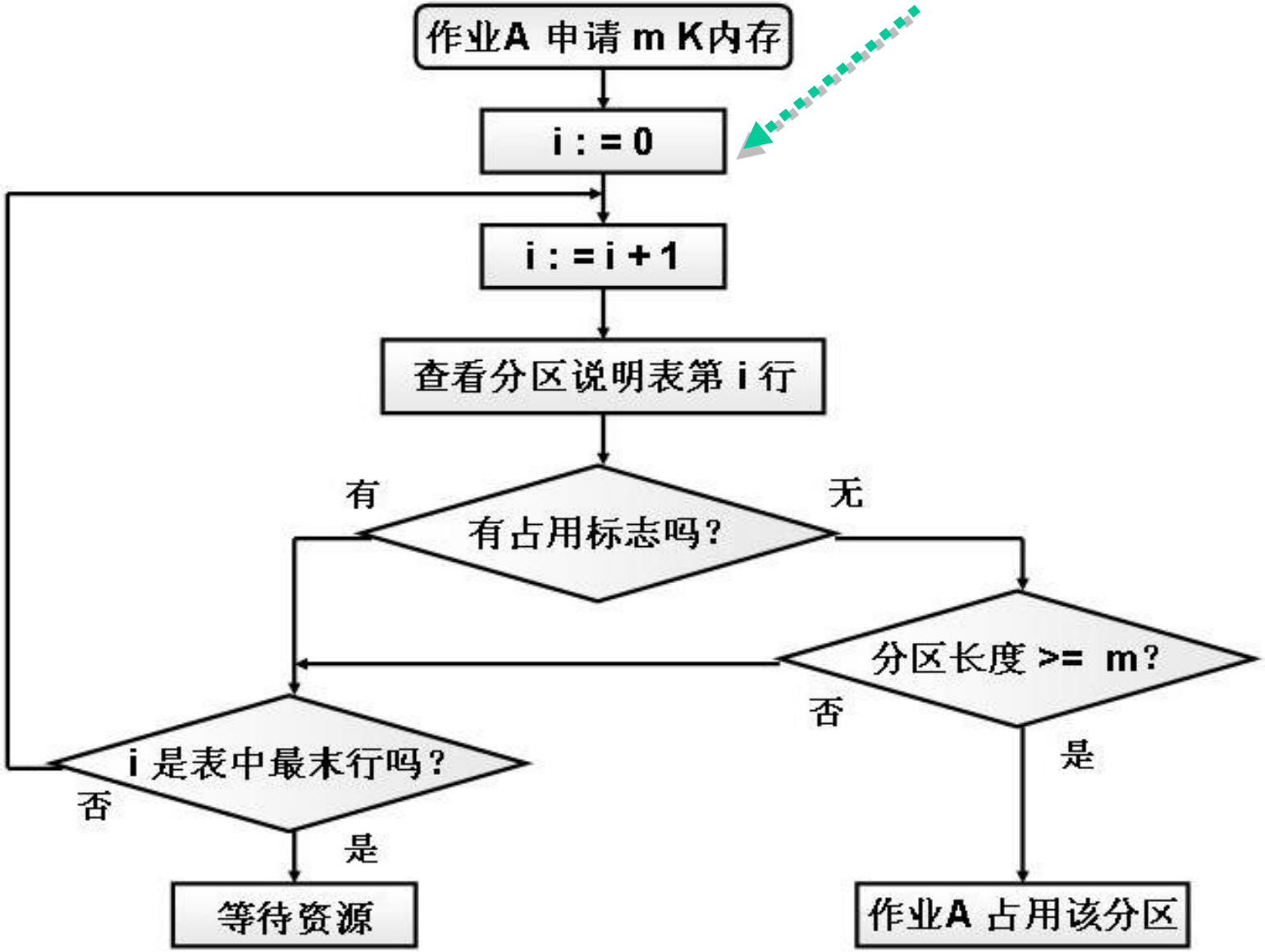
可变分区分配操作

- 事先规定 size 是不再切割的剩余分区的大小
- 设请求的分区大小为 u.size，空闲分区的大小为 m.size
- 若 m.size-u.size≤size，将整个分区分配给请求者
- 否则，从该分区中按请求的大小划分出一块内存空间分配出去，余下的部分仍留在空闲分区表/链中

基于顺序搜索的分配算法

1. 首次适应算法（First Fit）：每个空白区按其在内存空间中地址递增的顺序连在一起，在为作业分配内存区域时，从这个空白区域链的始端开始查找，选择第一个足以满足请求的空白块
2. 下次适应算法（Next Fit）：把内存空间中空白区构成一个循环链，每次为内存请求查找合适的分区时，总是从上次查找结束的地方开始，只要找到一个足够大的空白区，就将它划分后分配出去
3. 最佳适应算法（Best Fit）：为一个作业选择分区时，总是寻找其大小最接近于作业所要求的内存区域
4. 最坏适应算法（Worst Fit）：为作业选择内存区域时，总是寻找最大的空白区

FirstFit算法


作业A 申请 m K内存
i : = 0
i : = i + 1
查看分区说明表第 i 行
有
有占用标志吗?
无
分区长度 >= m?
否
是
i 是表中最末行吗?
否
是
等待资源
作业A 占用该分区

FirstFit算法

- 优点
  - 分配和释放的时间性能较好
  - 较大的空闲分区保留在内存的高端
- 缺点
  - 随着低端内存被不断分配，会产生很多小分区，开销会增大

算法举例

- 例：系统中的空闲分区表如下表示，现有三个作业分配申请内存空间 100K、30K 及 7K
- 给出按首次适应算法、下次适应算法、最佳适应算法和最坏适应算法的内存分配情况及分配后空闲分区表

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	120K	60K	未分配
4	331K	180K	未分配

首次适应算法

- 申请作业 100k，分配 3 号分区，剩下分区为 20k，起始地址 160K
- 100K、30K 、7K

大小	起始地址
32K	20K
8K	52K
120K	60K
331K	180K

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

首次适应算法

- 申请作业 30K，分配 1 号分区，剩下分区为 2K，起始地址 50K 。
- 100K、30K 、7K

大小	起始地址
32K	20K
8K	52K
20K	160K
331K	180K

区号	大小	起始地址	状态
1	2K	50K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

首次适应算法

- 申请作业 7K，分配 2 号分区，剩下分区为 1K，起始地址59K。
- 100K、30K 、7K

大小	起始地址
2K	50K
8K	52K
20K	160K
331K	180K

区号	大小	起始地址	状态
1	2K	50K	未分配
2	1K	59K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

下次适应算法

- 按下次适应算法，申请作业100k，分配3号分区，剩下分区为20k，起始地址160K；
- 100K、30K、7K

大小	起始地址
32K	20K
8K	52K
120K	60K
331K	180K

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	331K	180K	未分配

下次适应算法

- 按下次适应算法，申请作业 30K，分配 4 号分区，剩下分区为301K;
- 100K、30K 、7K

大小	起始地址
32K	20K
8K	52K
20K	160K
331K	180K

区号	大小	起始地址	状态
1	32K	20K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	301K	210K	未分配

下次适应算法

- 按下次适应算法，申请作业 7k，分配 1号分区
- 100K、30K 、7K

大小	起始地址
32K	20K
8K	52K
20K	160K
301K	210K

区号	大小	起始地址	状态
1	25K	27K	未分配
2	8K	52K	未分配
3	20K	160K	未分配
4	301K	210K	未分配

算法特点

- 首次适应的特点：
  - 优先利用内存低地址部分的空闲分区
  - 但由于低地址部分不断被划分，留下许多难以利用的很小的空闲分区（碎片或零头）
  - 而每次查找又都是从低地址部分开始，增加了查找可用空闲分区的开销
- 下次适应的特点：
  - 使内存空间的利用更加均衡，不致使小的空闲区集中在内存区的一端
  - 但这会导致缺乏大的空闲分区

最佳适应算法

按容量大小递增的次序排列

分配前的空闲分区表

区号	大小	起址
1	8k	52k
2	32k	20k
3	120k	60k
4	331k	180k

作业30K分配后

区号	大小	起址
2	2k	50k
1	8k	52k
3	20k	160k
4	331k	180k

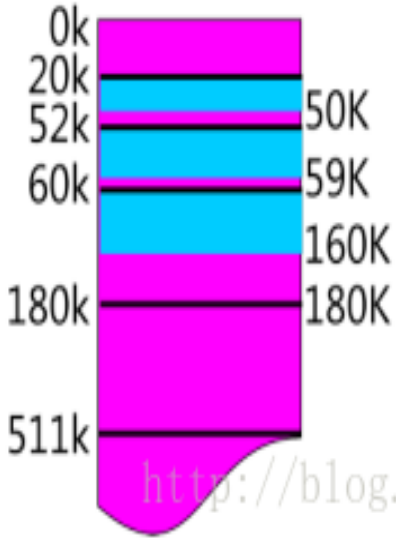
按容量递增的次序重新排列

作业100K分配后

区号	大小	起址
1	8k	52k
3	20k	160k
2	32k	20k
4	331k	180k

作业7K分配后

区号	大小	起址
1	1k	59k
2	2k	50k
3	20k	160k
4	331k	180k



区号	大小	起址
1	1k	59k
2	2k	50k
3	20k	160k
4	331k	180k

最坏适应算法

按容量大小递减的次序排列

分配前的空闲分区表

区号	大小	起址
1	331k	180k
2	120k	60k
3	32k	20k
4	8k	52k

作业30K分配后

区号	大小	起址
1	201k	310k
2	120k	60k
3	32k	20k
4	8k	52k

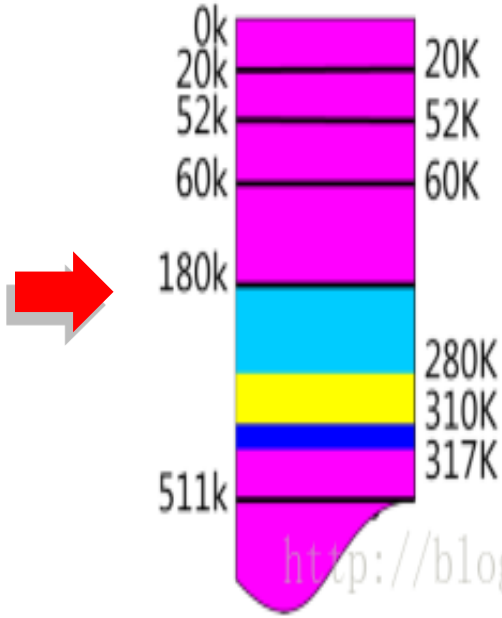
按容量递减的次序重新排列

作业100K分配后

区号	大小	起址
1	231k	280k
2	120k	60k
3	32k	20k
4	8k	52k

作业7K分配后

区号	大小	起址
1	194k	317k
2	120k	60k
3	32k	20k
4	8k	52k



区号	大小	起址
1	194k	317k
2	120k	60k
3	32k	20k
4	8k	52k

算法特点

- 最佳适应的特点：
  - 若存在与作业大小一致的空闲分区,则它必然被选中
  - 若不存在与作业大小一致的空闲分区，则只划分比作业稍大的空闲分区，从而保留了大的空闲分区
  - 最佳适应算法往往使剩下的空闲区非常小，从而在内存中留下许多难以利用的小空闲区（碎片）
- 最坏适应算法的特点：
  - 总是挑选满足作业要求的最大的分区分配给作业。这样使分给作业后剩下的空闲分区也较大，可装下其它作业
  - 由于最大的空闲分区总是因首先分配而划分，当有大作业到来时，其内存空间的申请往往会得不到满足

练习题

- 在下列内存管理算法中，内存分配和释放平均时间之和为最大的是:

A 首次适应 B 下次适应 C 最佳适应 D 最差适应

练习题

- 在下列内存管理算法中，内存分配和释放平均时间之和为最大的是:

A 首次适应 B 下次适应 C 最佳适应 D 最差适应

需要寻找满足需要且最小的空闲块，
释放要找到上下邻空闲区，修改插入链表

练习题

- 可变分区又称为动态分区，它是在系统运行过程中___动态建立的:

A 作业未装入　　B 在作业装入

C 在作业创建　　D 在作业完成

练习题

- 可变分区又称为动态分区，它是在系统运行过程中___动态建立的:

A 作业未装入　　B 在作业装入

C 在作业创建　　D 在作业完成

分区大小在程序装入时大小动态确定，量身定制

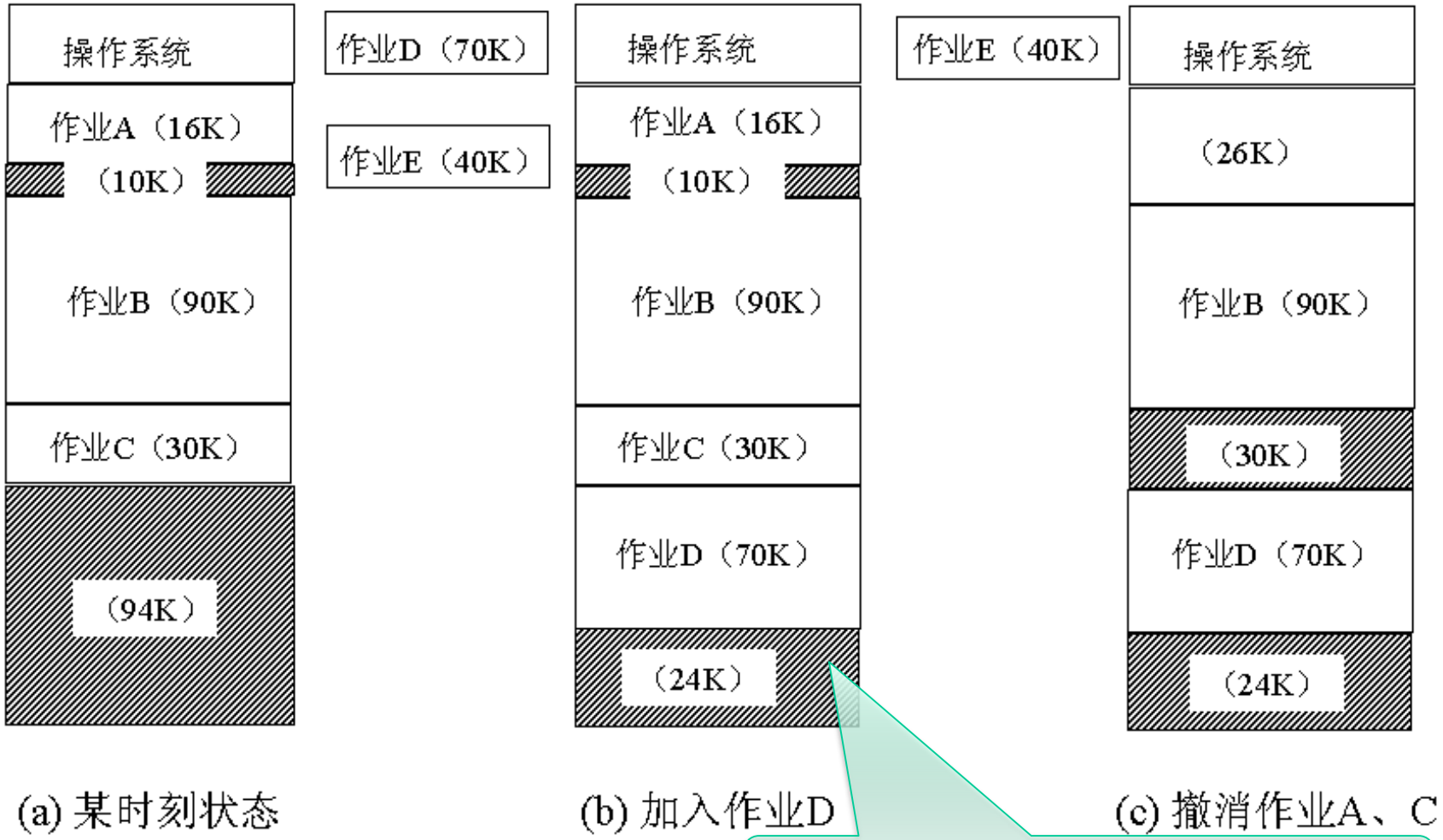
练习：可变分区的内存分配

- 假设一个可变分区系统中内存按照顺序包含如下空闲区：10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB, and 15 KB.
- 假设后续的连续内存分配请求是：(a) 12 KB (b) 10 KB (c) 9 KB
- 对于首次适应那个空闲区会被分配?对于 最佳适应, 最差适应, 和 下次适应 又会分配那些空闲区?

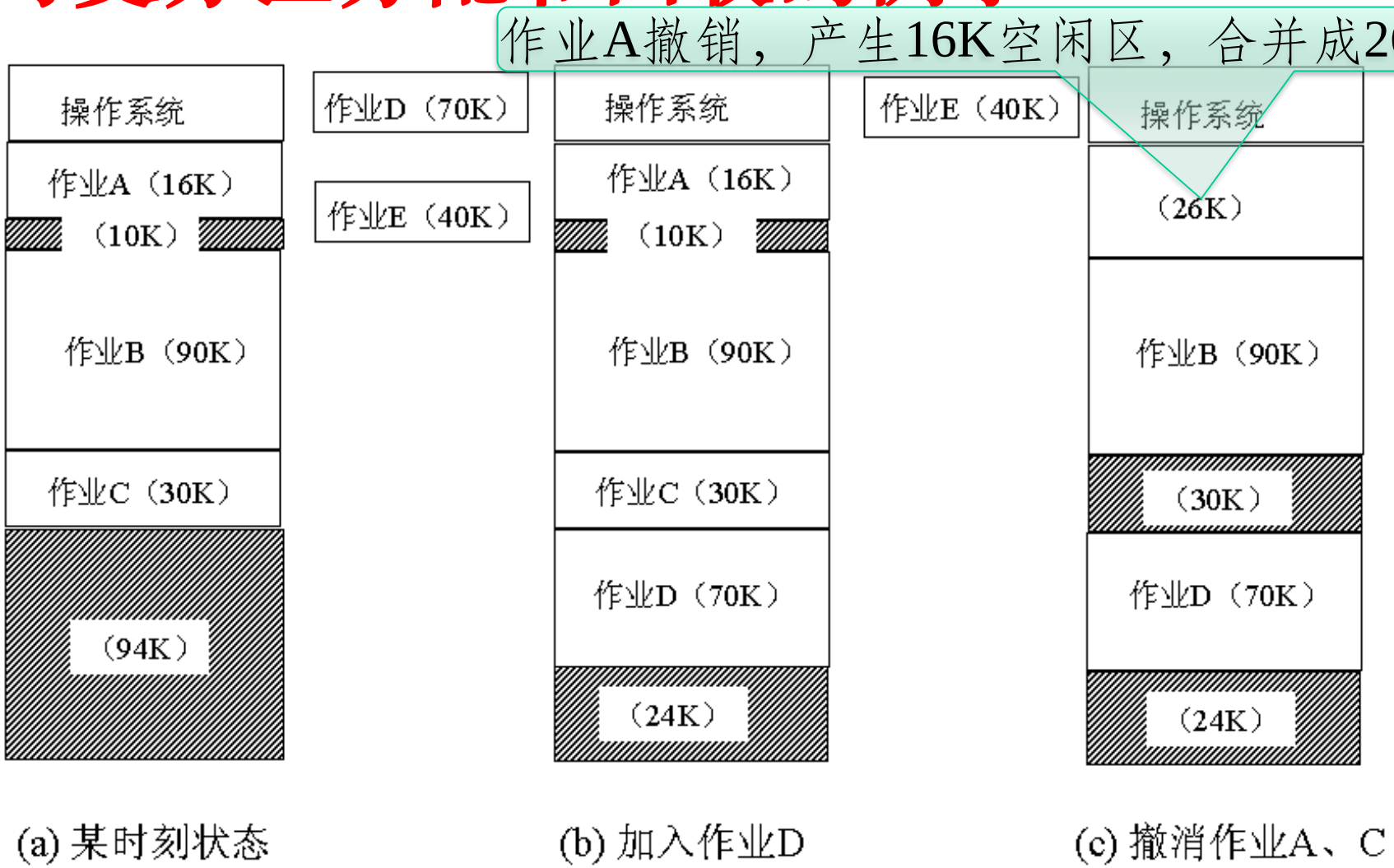
练习：可变分区的内存分配

- 首次适应 20 KB, 10 KB, 18 KB.
- 最佳适应 12 KB, 10 KB, 9 KB.
- 最差适应 20 KB, 18 KB, 15 KB.
- 下次适应 20 KB, 18 KB, 9 KB.

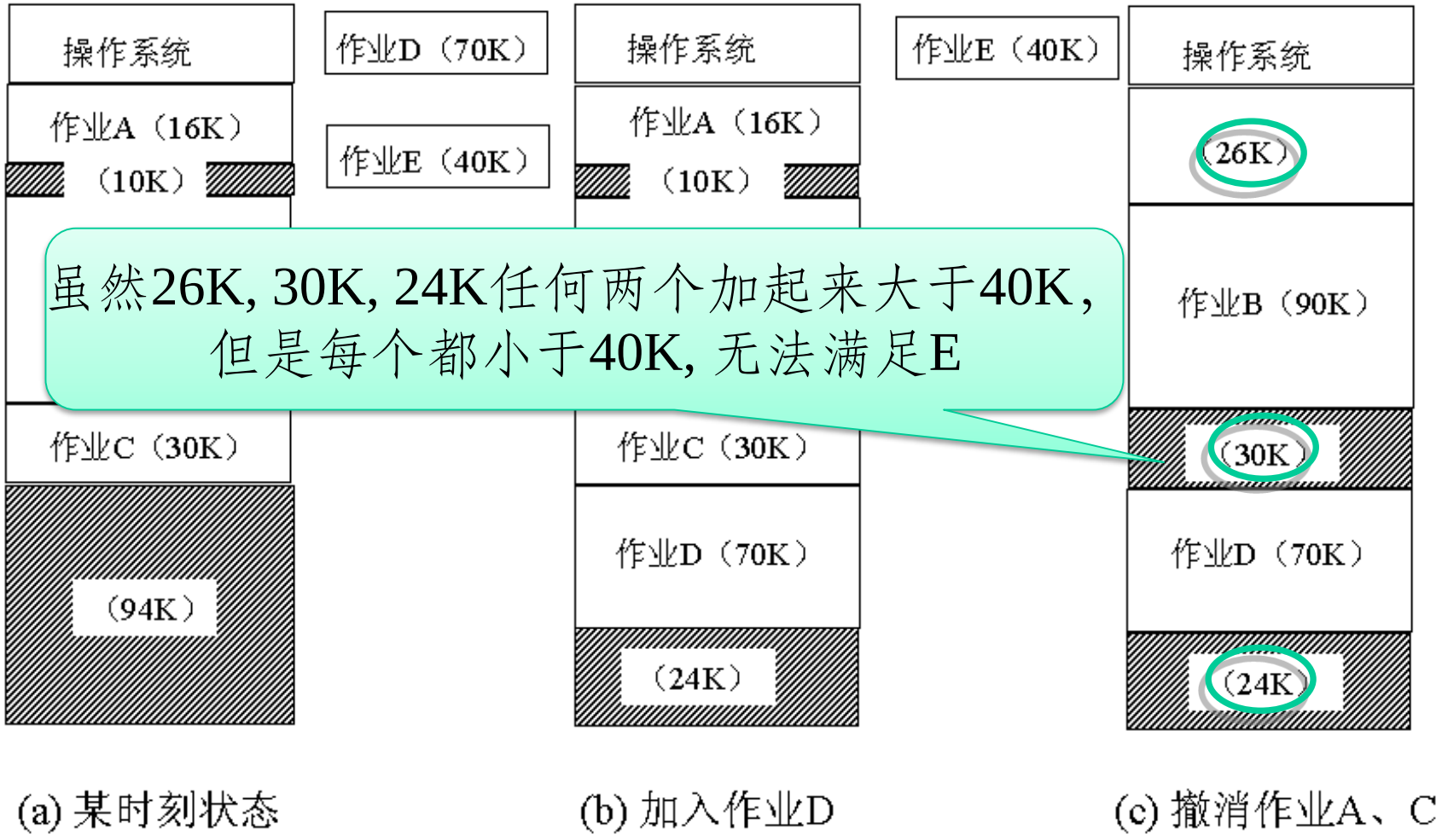
可变分区分配和回收的例子



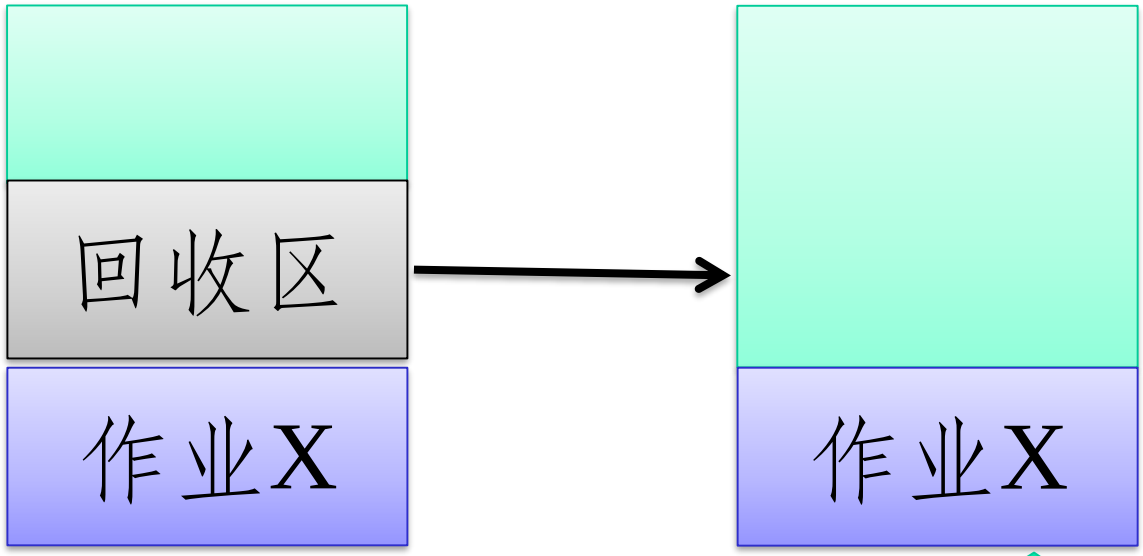
可变分区分配和回收的例子



可变分区分配和回收的例子



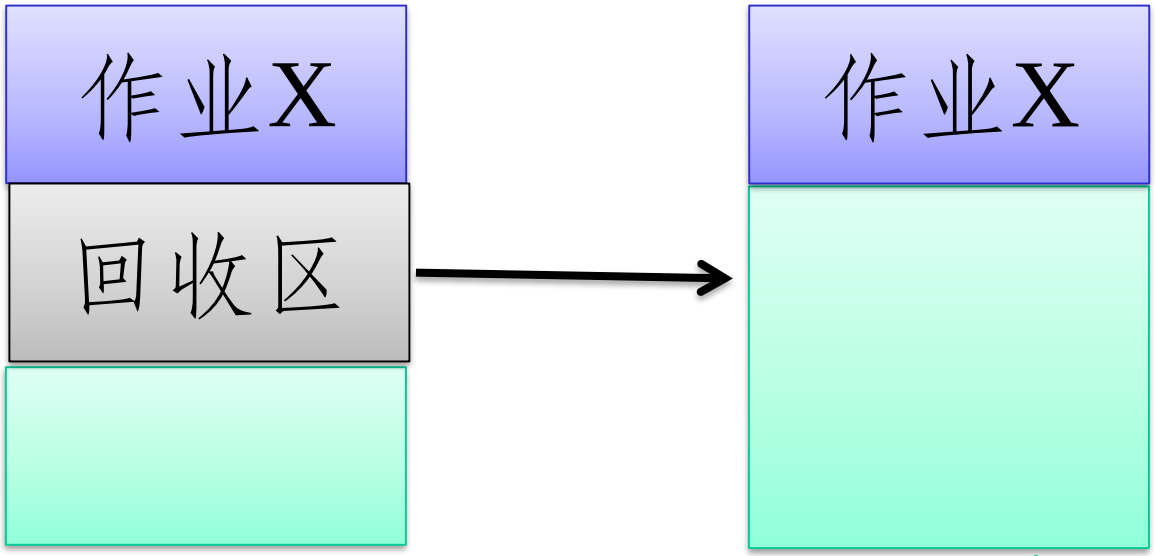
回收区域空白区邻接的四种情况



绿色表示空闲区

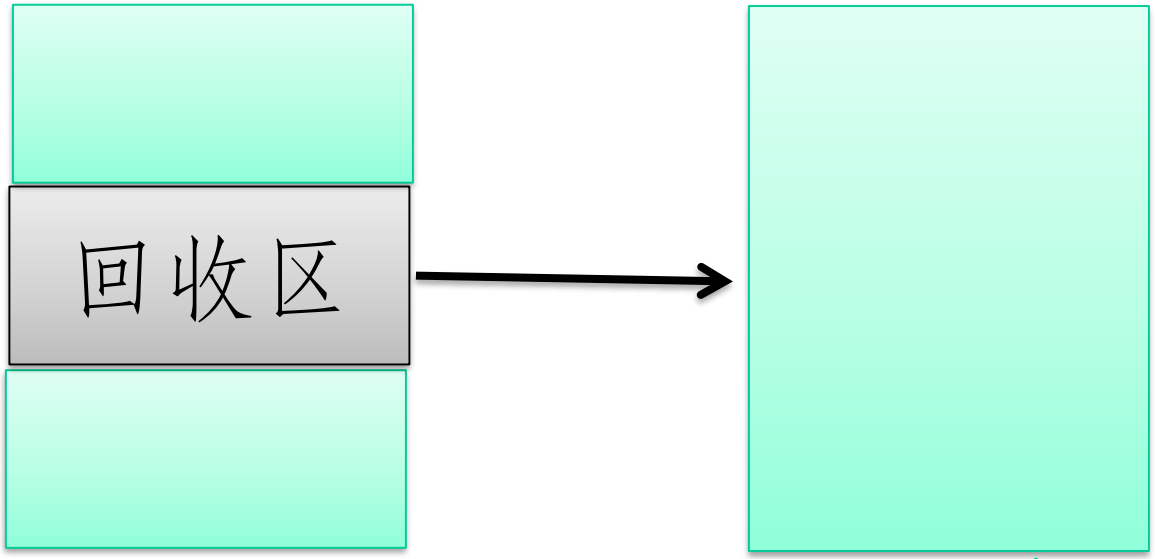
合并空闲区域，修改空闲区表

回收区域空白区邻接的四种情况



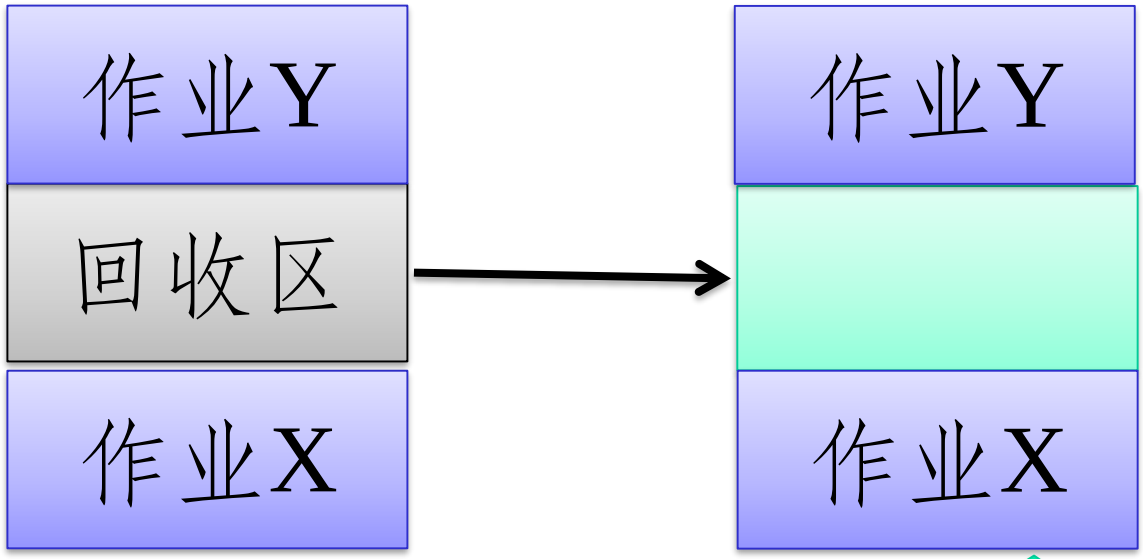
合并空闲区域，修改空闲区表

回收区域空白区邻接的四种情况



合并空闲区域，修改空闲区表

回收区域空白区邻接的四种情况



绿色表示空闲区

直接修改空闲区表

3.1内容提要

- 3.1 内存管理基础
  - 单道程序的内存管理
  - 多道程序的内存管理
    - 固定式分区
    - 可变式分区
      - 基于顺序搜索的分配算法
      - 基于索引搜索的分配算法
      - 紧凑技术
    - 多重分区
      - 界限寄存器方法
      - 内存保护键方法
    - 内外协同，内存复用
      - 覆盖
      - 交换

基于索引搜索的分配算法

- 基于顺序搜索的动态分区分配算法一般只是适合于较小的系统
- 如果系统的分区很多，空闲分区表（链）可能很大（很长），检索速度会比较慢
- 为了提高搜索空闲分区的速度，大中型系统采用了基于索引搜索的动态分区分配算法

快速适应算法

- 快速适应算法，又称为分类搜索法
- 把空闲分区按容量大小进行分类
  - 经常用到长度的空闲区设立单独的空闲区链表
  - 系统为多个空闲链表设立一张管理索引表

优点：

- 查找效率高，仅需要根据程序的长度，寻找到能容纳它的最小空闲区链表，取下第一块进行分配即可
- 该算法在分配时，不会对任何分区产生分割，所以能保留大的分区，也不会产生内存碎片

缺点：

- 在分区归还主存时算法复杂，系统开销较大
- 在分配空闲分区时是以进程为单位，一个分区只属于一个进程，存在一定的浪费。空间换时间

伙伴系统

- 固定分区方式不够灵活，当进程大小与空闲分区大小不匹配时，内存空间利用率很低。
- 动态分区方式算法复杂，回收空闲分区时需要进行分区合并等，系统开销较大。
- 伙伴系统 (buddy system)是介于固定分区与可变分区之间的动态分区技术。
- 伙伴：在分配内存块时将一个大的内存块分裂成两个大小相等的小块，这两个小块就称为“伙伴”。

Linux系统采用，同学们回去自学

伙伴系统

- 伙伴系统规定，无论已分配分区或空闲分区，其大小均为 2 的 k 次幂，k 为整数，$n \leq k \leq m$，其中：2^n 表示分配的最小分区的大小，2^m 表示分配的最大分区的大小，通常 2^m 是整个可分配内存的大小。
- 在系统运行过程中，由于不断的划分，可能会形成若干个不连续的空闲分区。
- 内存管理模块保持有多个空闲块链表，空闲块的大小可以为 1，2，4，8，…，2^m 字节。

伙伴系统的内存分配

系统初启时，只有一个最大的空闲块（整个内存）。

当一个长度为 n 的进程申请内存时，系统就分给它一个大于或等于所申请尺寸的最小的 2 的幂次的空闲块。

如果 $2^{i-1} < n \leq 2^i$，则在空闲分区大小为 2^i 的空闲分区链表中查找。

例如，某进程提出的 50KB 的内存请求，将首先被系统向上取整，转化为对一个 64KB 的空闲块的请求。

若找到大小为 2^i 的空闲分区，即把该空闲分区分配给进程。否则表明长度为 2^i 的空闲分区已经耗尽，则在分区大小为 2^{i+1} 的空闲分区链表中寻找。

伙伴系统的内存分配

若存在 2^{i+1} 的一个空闲分区，把该空闲分区分为相等的两个分区，这两个分区称为一对伙伴，其中的一个分区用于分配，另一个加入大小为 2^i 的空闲分区链表中。

若大小为 2^{i+1} 的空闲分区也不存在，需要查找大小为 2^{i+2} 的空闲分区，若找到则对其进行两次分割：第一次，将其分割为大小为 2^{i+1} 的两个分区，一个用于分割，一个加入到大小为 2^{i+1} 的空闲分区链表中；第二次，将用于分割的空闲区分割为 2^i 的两个分区，一个用于分配，一个加入到大小为 2^i 的空闲分区链表中。

若仍然找不到，则继续查找大小为 2^{i+3} 的空闲分区，以此类推。

伙伴系统的内存释放

首先考虑将被释放块与其伙伴合并成一个大的空闲块，然后继续合并下去，直到不能合并为止。

例如：回收大小为 2^i 的空闲分区时，若事先已存在 2^i 的空闲分区时，则应将其与伙伴分区合并为大小为 2^{i+1} 的空闲分区，若事先已存在 2^{i+1} 的空闲分区时，又应继续与其伙伴分区合并为大小为 2^{i+2} 的空闲分区，依此类推。

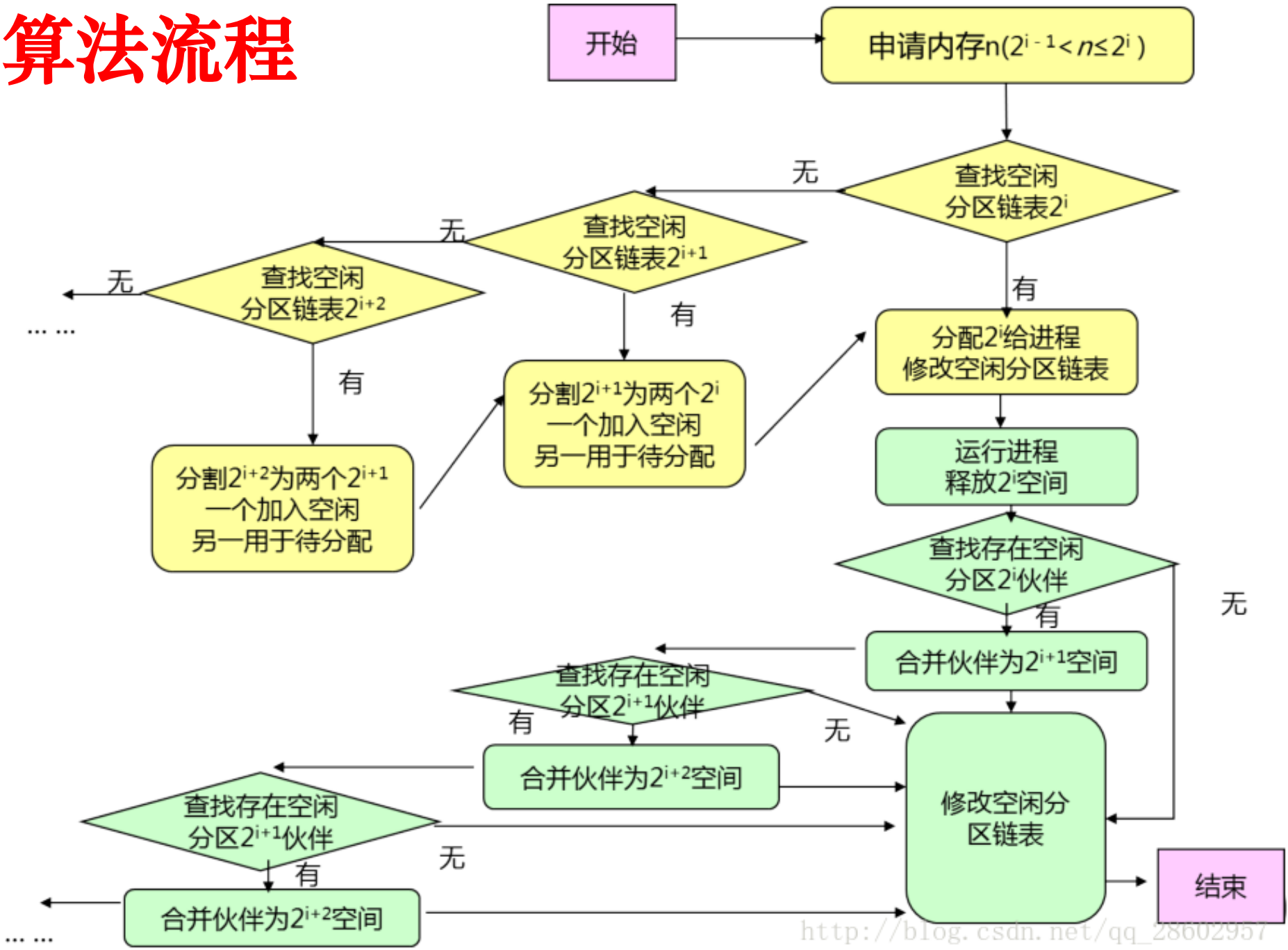
如果有两个存储块大小相同，地址也相邻，但不是由同一个大块分裂出来的（不是伙伴），则不会被合并起来。

一个例子

伙伴系统示例（1M 内存）

Action	Memory					
Start	1M					
A请求150kb	A	256k			512k	
B请求100kb	A	B	128k		512k	
C请求50kb	A	B	C	64k	512k	
释放B	A	128k	C	64k	512k	
D请求200kb	A	128k	C	64k	D	256k
E请求60kb	A	128k	C	E	D	256k
释放C	A	128k	64k	E	D	256k
释放A	256k	128k	64k	E	D	256k
释放E	512k				D	256k
释放D	1M					

算法流程



伙伴系统特点

伙伴系统利用计算机二进制数寻址的优点，加速了相邻空闲分区的合并。

当一个 2^i 字节的块释放时，只需搜索 2^i 字节的块，而其它算法则必须搜索所有的块，伙伴系统速度更快。

伙伴系统的缺点：不能有效地利用内存。进程的大小不一定是 2 的整数倍，由此会造成浪费，内部碎片严重。例如，一个 257KB 的进程需要占用一个 512KB 的分配单位，将产生 255KB 的内部碎片。

伙伴系统不如基于分页和分段的虚拟内存技术有效。

伙伴系统目前应用于 Linux 系统和多处理机系统。

3.1内容提要

- 3.1 内存管理基础
  - 单道程序的内存管理
  - 多道程序的内存管理
    - 固定式分区
    - 可变式分区
      - 基于顺序搜索的分配算法
      - 基于索引搜索的分配算法
      - 紧凑技术
    - 多重分区
      - 界限寄存器方法
      - 内存保护键方法
    - 内外协同，内存复用
      - 覆盖
      - 交换

系统中的（内部）碎片

- 内存中无法被利用的内存空间，称为碎片。

内部碎片

- 指分配给作业的内存空间中未被利用的部分，如固定分区中存在的碎片
- 单一连续区内存管理、固定分区内存管理等都会出现内部碎片
- 内部碎片无法被整理，但作业完成后会得到释放。它们其实已经被分配出去了，只是没有被利用

系统中的外部碎片

外部碎片：

- 指系统中无法利用的小的空闲分区。如分区与分区之间存在的碎片。这些不连续的区间就是外部碎片。动态分区管理会产生外部碎片
- 外部碎片才是造成内存系统性能下降的主要原因。外部碎片可以被整理后清除
- 消除外部碎片的方法：紧凑技术

紧凑技术（Compaction）

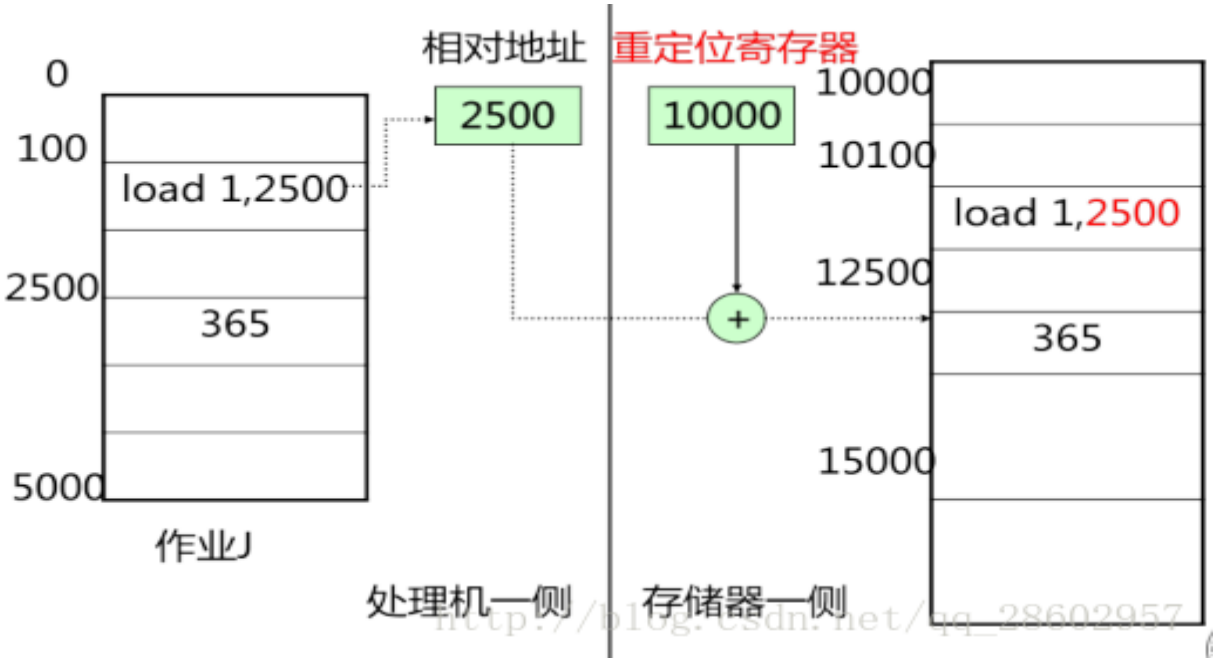
- 通过移动作业从把多个分散的小分区拼接成一个大分区的方法称为紧凑（拼接或紧缩）
- 目标：消除外部碎片，使本来分散的多个小空闲分区连成一个大的空闲区
- 紧凑时机：找不到足够大的空闲分区且总空闲分区容量可以满足作业要求时

实现支撑

动态重定位：作业在内存中的位置发生了变化，这就必须对其地址加以修改或变换。

可重定位分区分配

结合紧凑技术的动态分区技术



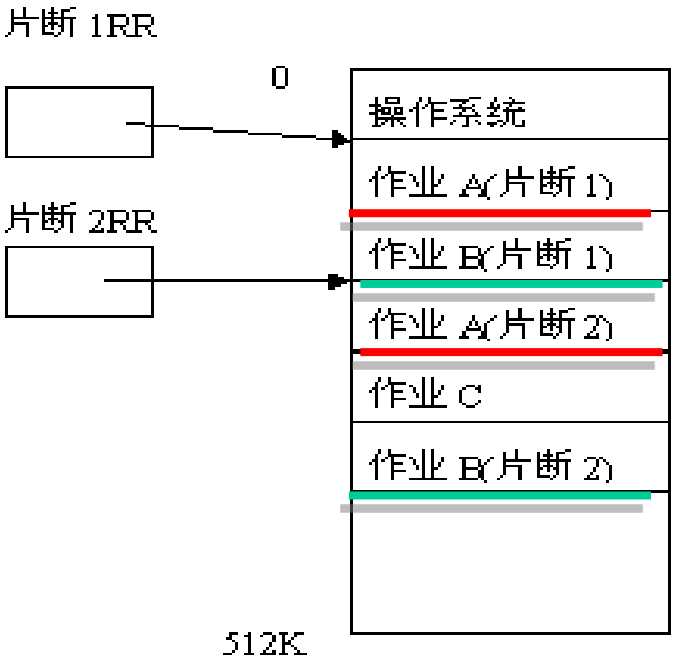
动态重定位的实现

3.1内容提要

- 3.1 内存管理基础
  - 单道程序的内存管理
  - 多道程序的内存管理
    - 固定式分区
    - 可变式分区
      - 基于顺序搜索的分配算法
      - 基于索引搜索的分配算法
      - 紧凑技术
    - 多重分区
      - 界限寄存器方法
      - 内存保护键方法
    - 内外协同，内存复用
      - 覆盖
      - 交换

多重分区分配

- 多重分区分配：一个作业往往由相对独立的程序段和数据段组成，将这些片断分别装入到内存空间中不同的区域内的分配方式。

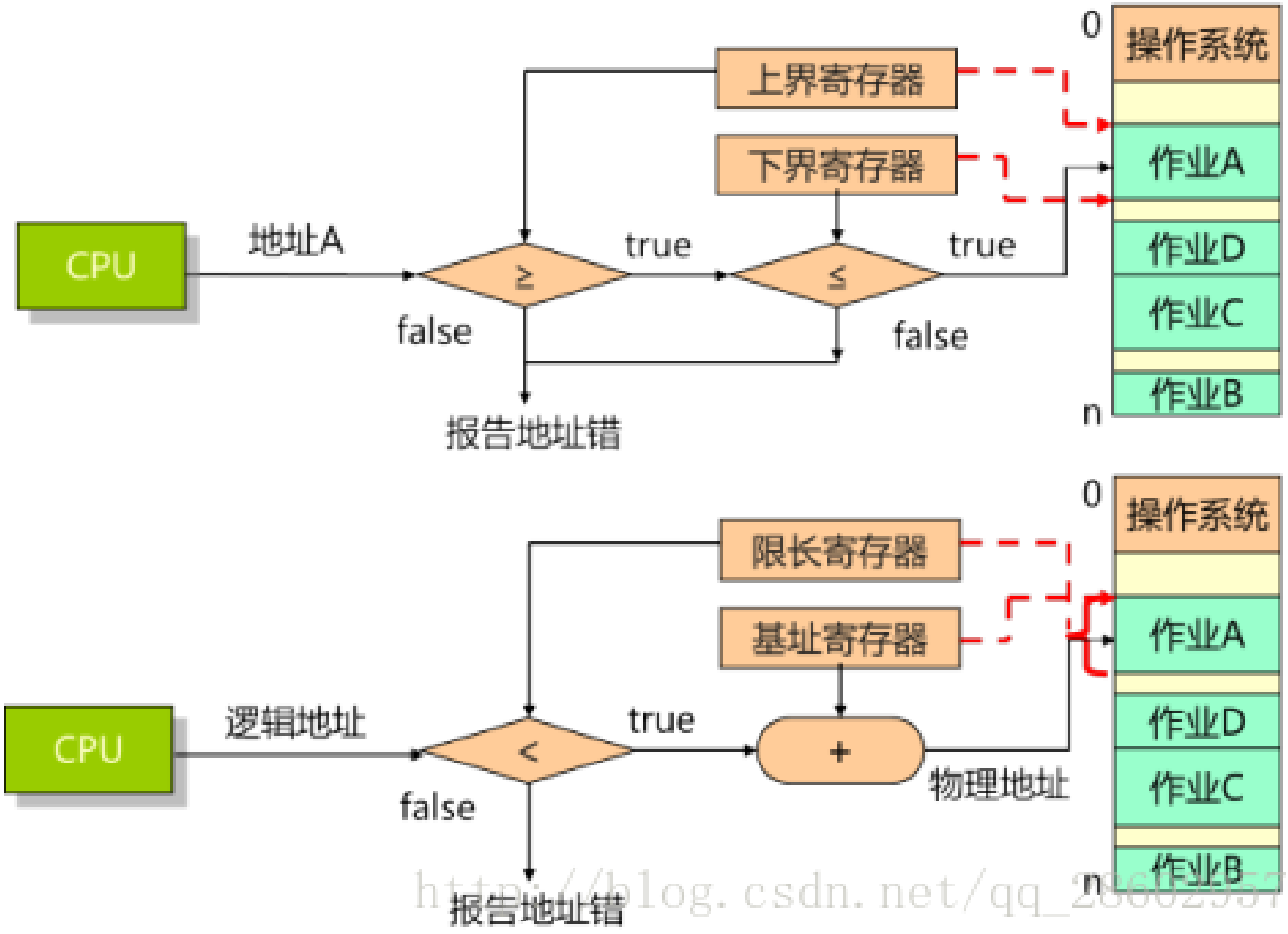


分区的内存保护

内存保护是为了防止一个作业有意或无意地破坏操作系统或其它作业。常用的内存保护方法有

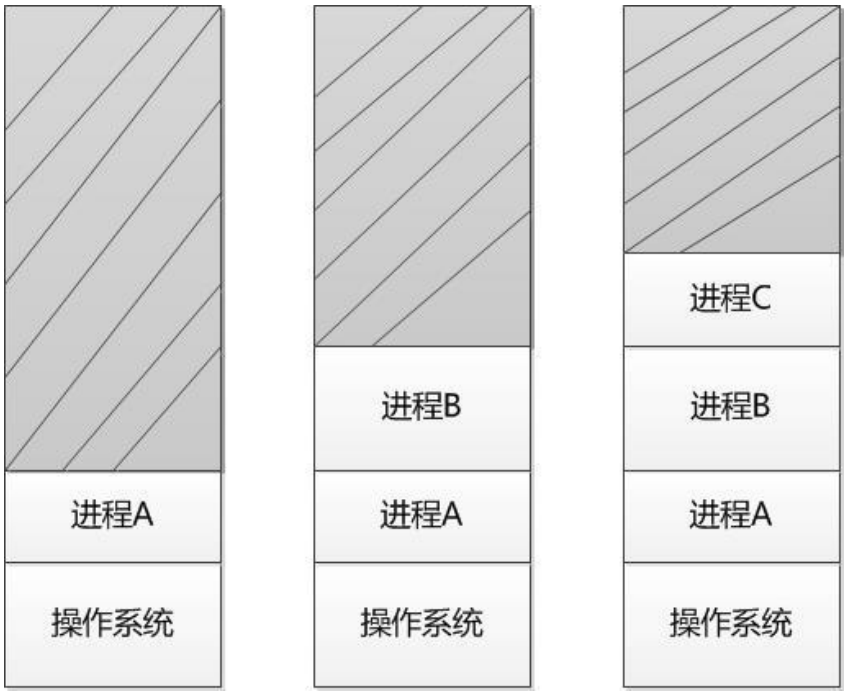
- 界限寄存器方法：
  - 上下界寄存器方法
  - 基址、限长寄存器 (BR，LR) 方法
- 内存保护键方法：
  - 给每个内存块分配一个单独的保护键，它相当于一把锁。进入系统的每个作业也赋予一个保护键，它相当于一把钥匙。当作业运行时，检查钥匙和锁是否匹配，如果不匹配，则系统发出保护性中断信号，停止作业运行。

界限寄存器法



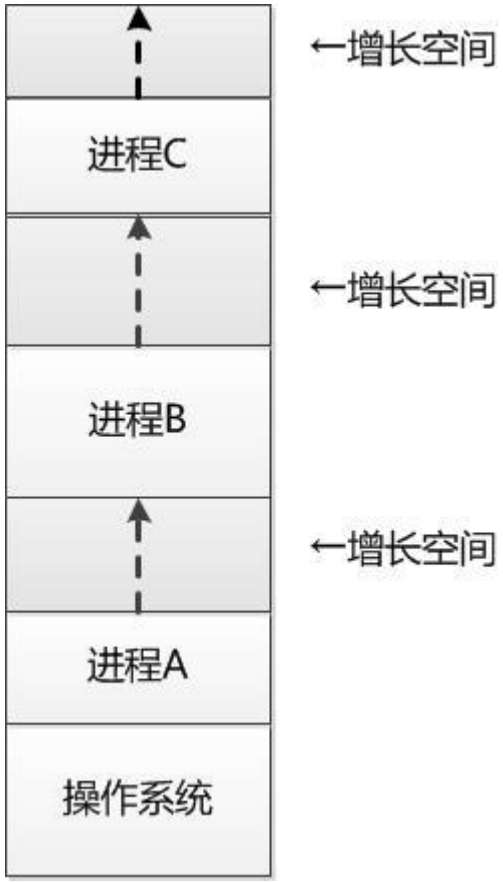
分区管理中的进程扩张问题

- 例如，一开始内存中只有OS，这时候进程A来了，于是分出一片与进程A大小一样的内存空间
- 随后，进程B来了，于是在进程A之上分出一片给进程B；然后进程C来了，就在进程B上面再分出一片给C



大作业小内存问题

- 每个程序像叠罗汉一样累计，如果程序B在成型过程中需要更多空间怎么办？（例如在实际程序中，很多递归嵌套函数调用的时候回造成栈空间的增长）
- 预留一定的空间？OS怎么知道应该分配多少空间给一个程序呢？分配多了，就是浪费；而分配少了，则可能造成程序无法继续执行。



必须解决大作业在小内存中运行的问题。

3.1内容提要

- 3.1 内存管理基础
  - 单道程序的内存管理
  - 多道程序的内存管理
    - 固定式分区
    - 可变式分区
      - 基于顺序搜索的分配算法
      - 基于索引搜索的分配算法
      - 紧凑技术
    - 多重分区
      - 界限寄存器方法
      - 内存保护键方法
    - 内外协同，内存复用
      - 覆盖
      - 交换

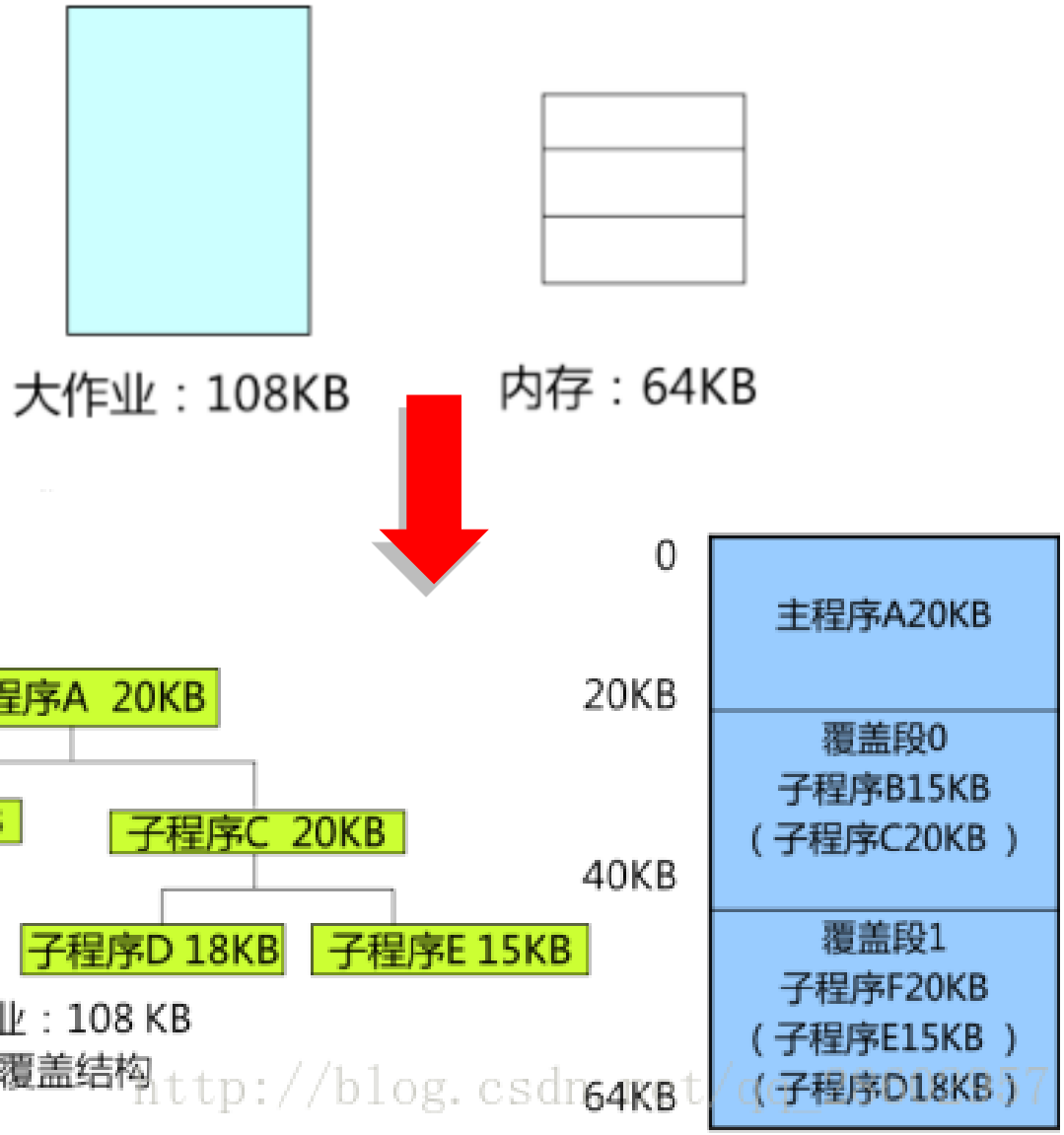
解决的方法

- **覆盖**与**交换**技术是在多道程序环境下用来扩充内存的两种方法
- 覆盖与交换可以解决在小的内存空间运行大作业的问题，是“扩充”内存容量和提高内存利用率的有效措施
- 覆盖技术主要用在早期的 OS 中，交换技术则用在现代OS 中

覆盖 (Overlay)

- 覆盖技术主要用在早期的 OS 中（内存 <64KB），可用的内存空间受限，某些大作业不能一次全部装入内存，产生了大作业与小内存的矛盾
- 覆盖：把一个程序划分为一系列功能相对独立的程序段，让执行时不要求同时装入内存的程序段组成一组（称为覆盖段），共享主存的同一个区域，这种内存扩充技术就是覆盖
  - 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）
  - 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
- 缺点：对用户不透明，增加了用户负担

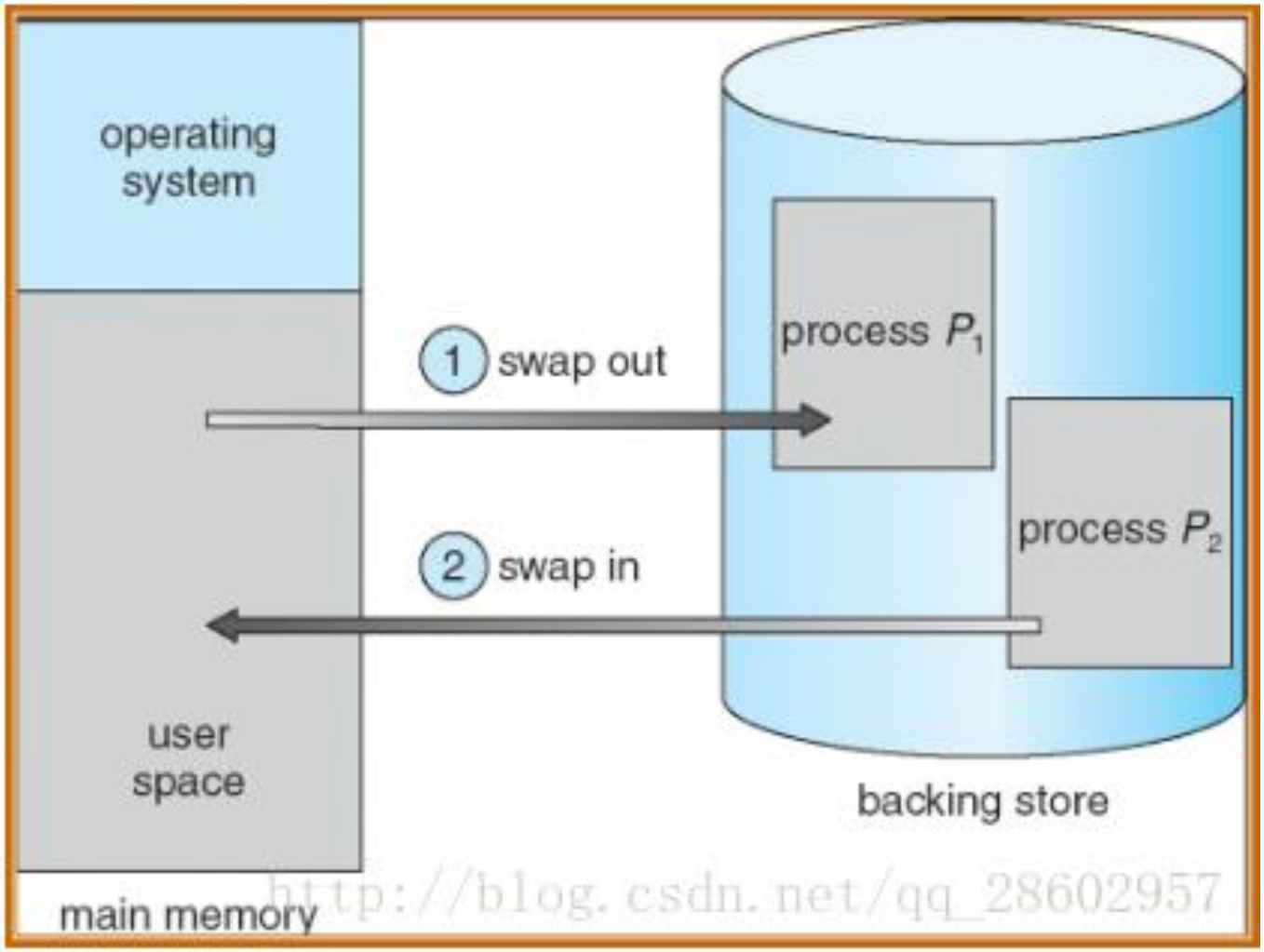
覆盖示例


大作业：108KB
内存：64KB
主程序A 20KB
子程序B 15KB
子程序C 20KB
子程序F 20KB
子程序D 18KB
子程序E 15KB
大作业：108 KB
覆盖结构
0
20KB
40KB
64KB
主程序A20KB
覆盖段0
子程序B15KB
（子程序C20KB ）
覆盖段1
子程序F20KB
（子程序E15KB ）
（子程序D18KB ）

交换（Swapping）

- 交换
  - 广义的说，所谓交换就是把暂时不用的某个（或某些）程序及其数据的部分或全部从主存移到辅存中去，以便腾出必要的内存空间
  - 接着把指定程序或数据从辅存读到相应的主存中，并将控制转给它，让其在系统上运行
- 优点
  - 增加并发运行的程序数目，并且给用户提供适当的响应时间；
  - 编写程序时不影响程序结构
- 缺点
  - 对换入和换出的控制增加处理机开销
  - 程序整个地址空间都进行传送，没有考虑执行过程中地址访问的统计特性

交换（Swapping）


operating system
process P_1
1 swap out
process P_2
2 swap in
user space
backing store
main memory
http://blog.csdn.net/qq_28602957

交换技术的几个问题

- 选择原则，即将哪个进程换出/内存?
  - 等待I/O的进程
- 交换时机的确定，何时需发生交换?
  - 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
- 交换时需要做哪些工作?
- 换入回内存时位置的确定

覆盖与交换技术的区别

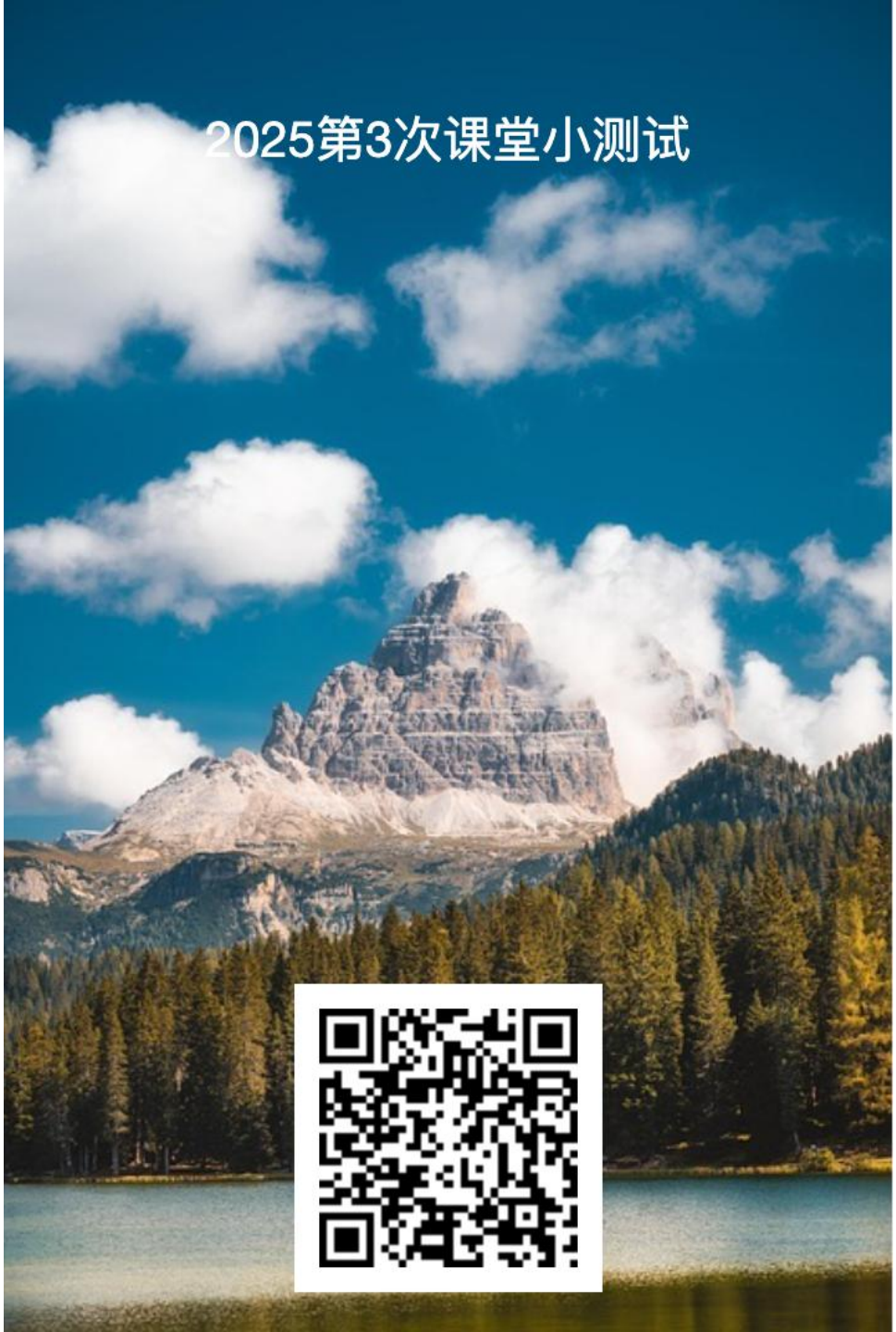
- 覆盖可减少一个程序运行所需的空间
  交换可让整个程序暂存于外存中，让出内存空间
- 覆盖是由程序员实现的，操作系统根据程序员提供的覆盖结构来完成程序段之间的覆盖
  交换技术不要求程序员给出程序段之间的覆盖结构
- 覆盖技术主要对同一个作业或程序进行
  交换换主要在作业或程序间之间进行

讨论

是否满足:

- 地址独立?
- 地址保护?

- 碎片问题、小内存运行大程序问题


2025第3次课堂小测试