



操作系统 Operating System

第三章 内存管理(1)

沃天宇

woty@buaa.edu.cn

2024年3月8日





小练习

- 在传统x86体系结构下，计算机引导过程中以下哪项不是BIOS负责的任务？
 - 读取MBR并装载到内存特定地址
 - 自检系统，当有设备故障时暂停启动过程并告警
 - 选择引导磁盘
 - 装载打印机驱动
 - 启动显示器
 - 解压缩操作系统Kernel
- 以下说法正确的是
 - BIOS是只读存储器，
 - 如果硬盘只有一个分区，则不需要MBR引导
 - 引导程序装载进内存后不会变更位置
 - BIOS程序运行时可以访问超过4GB的地址空间



* 1. 1.1. 以下项功能不是操作系统核心的任务：（多选）【多选题】

- ☐ A. 调度多个任务执行
- ☐ B. 管理外部设备
- ☐ C. 响应键盘按键事件
- ☐ D. 对摄像头采集的图像进行美颜处理
- ☐ E. 驱动打印机
- ☐ F. 执行数据库查询
- ☐ G. 实现进程间的同步互斥

* 2. 2. 第一次课中提到的Meltdown的根源是：（单选）

- ☐ A. 操作系统漏洞
- ☐ B. CPU设计缺陷
- ☐ C. 计算机病毒

* 3. 3. Meltdown和 Spectre是否可以通过升级操作系统进行修复？

- ☐ A. 可以
- ☐ B. 不可以

* 4. 4. 以下*没有*获得过图灵奖的人是：

- ☐ A. Ken Thompson
- ☐ B. Dennis Ritchie
- ☐ C. Frederick Brooks
- ☐ D. Brian Kernighan

* 5. 目前正在使用的操作系统有多少种？

- ☐ <10
- ☒ 10-50
- ☐ 51-100
- ☐ 100+

* 6. 你使用过那些系统调用，列举3个例子。



* 1. 计算机系统中下面哪部分增长速度最快？

- ☐ A. CPU主频；
- ☐ B. 内存大小；
- ☐ C. 磁盘大小；
- ☐ D. 网络带宽

* 2. DOS系统增加了网络功能，它属于（ ）。

- ☐ A. 分布式操作系统；
- ☐ B. 网络操作系统

* 3. 哪个软件管理内存？

- ☐ A. 操作系统；
- ☐ B. 编译系统

* 4. 哪个软件管理寄存器？

- ☐ A. 操作系统；
- ☐ B. 编译系统

* 5. 故障?(fault)是（ ）

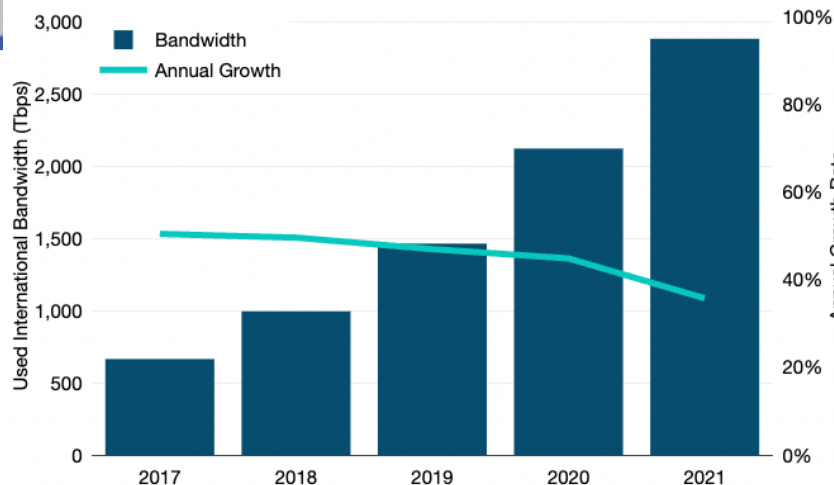
- ☐ A. 异步异常；
- ☐ B. 同步异常

* 6. 系统调用是（ ）

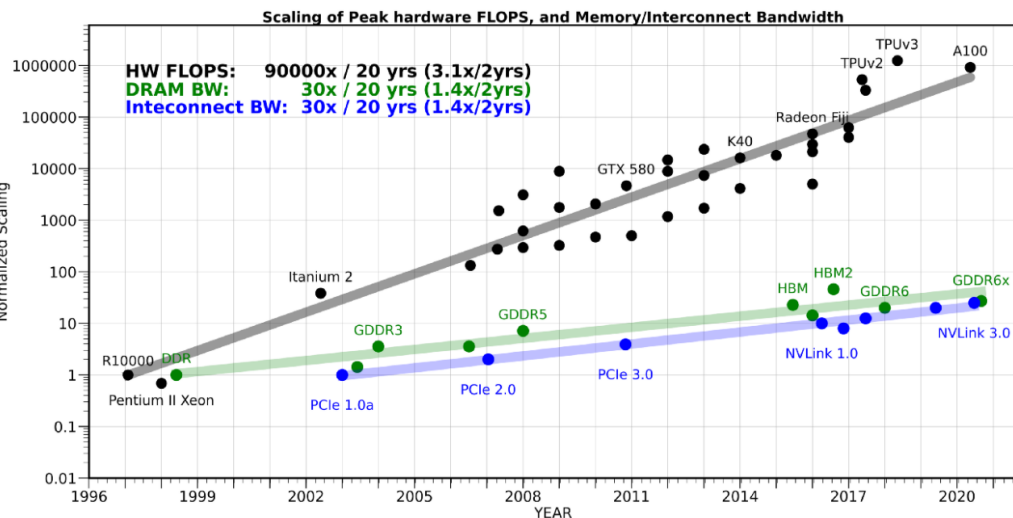
- ☐ A. 异步异常；
- ☐ B. 同步异常

* 7. 下面哪一项不是微内核结构的优点？

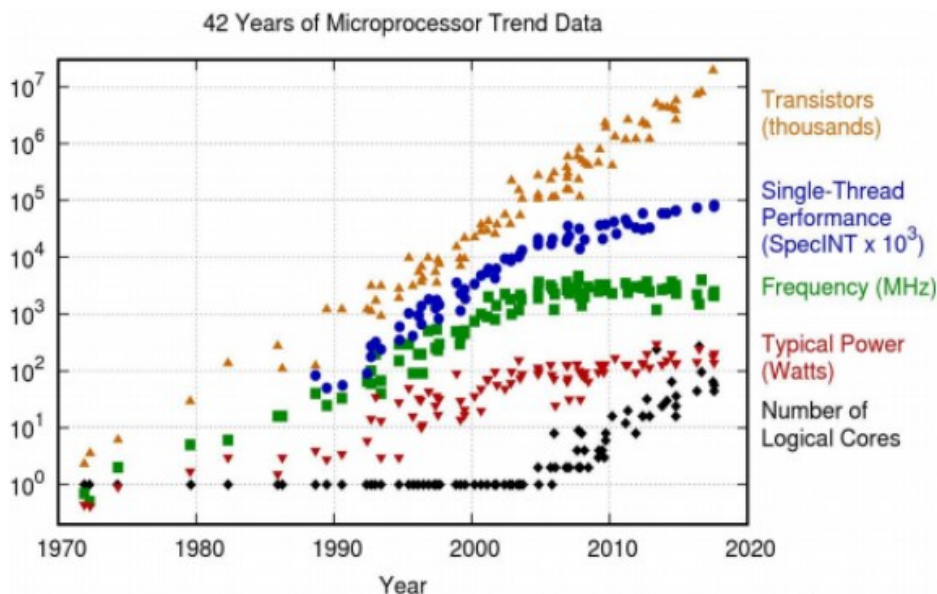
- ☐ A. 可移植性好；
- ☐ B. 配置灵活；
- ☐ C. 适应分布式环境；
- ☐ D. 速度快



国际带宽及增长率

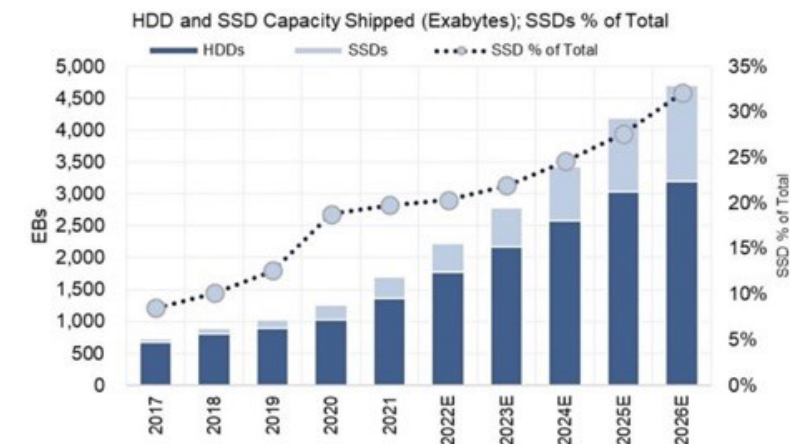


硬件算力vs内存、网络带宽



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
 New plot and data collected for 2010-2017 by K. Rupp

微处理器发展趋势



Source: Gartner; Wells Fargo Securities, LLC.

磁盘发货量



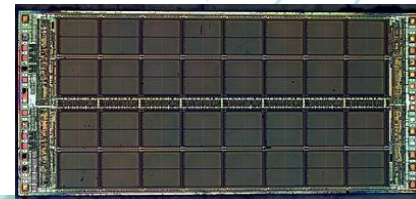
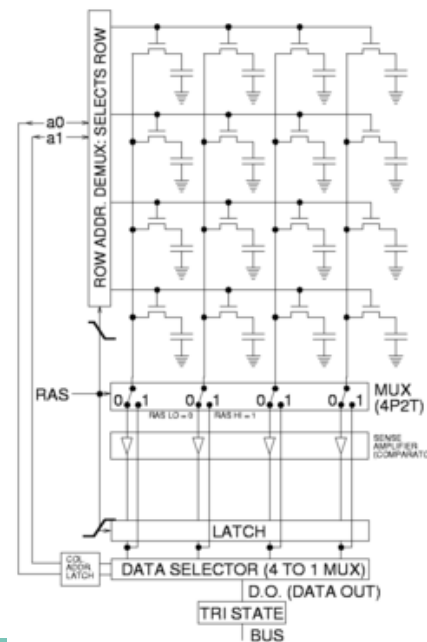
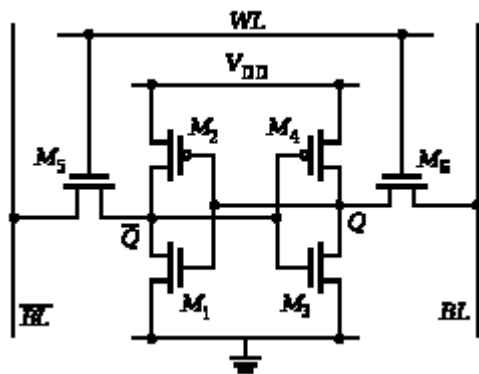
存储器的管理

- 存储**分配**：主要是讨论和解决多道作业之间**共享主存**的存储空间的问题。
- 存储器资源的**组织**（如内存的组织方式）
- **地址变换**（逻辑地址与物理地址的对应关系维护）
- **虚拟存储**的调度算法



存储组织

- 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；硬盘技术在大容量方面的发展：接口标准、存储密度等；
 - 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**



存储组织

- 存储组织的功能是在存储技术和CPU寻址技术许可的范围内组织合理的存储结构，其依据是访问速度匹配关系、容量要求和价格。如：“寄存器-内存-外存”结构和“寄存器-缓存-内存-外存”结构；
- 现在微机中的存储层次组织：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）；



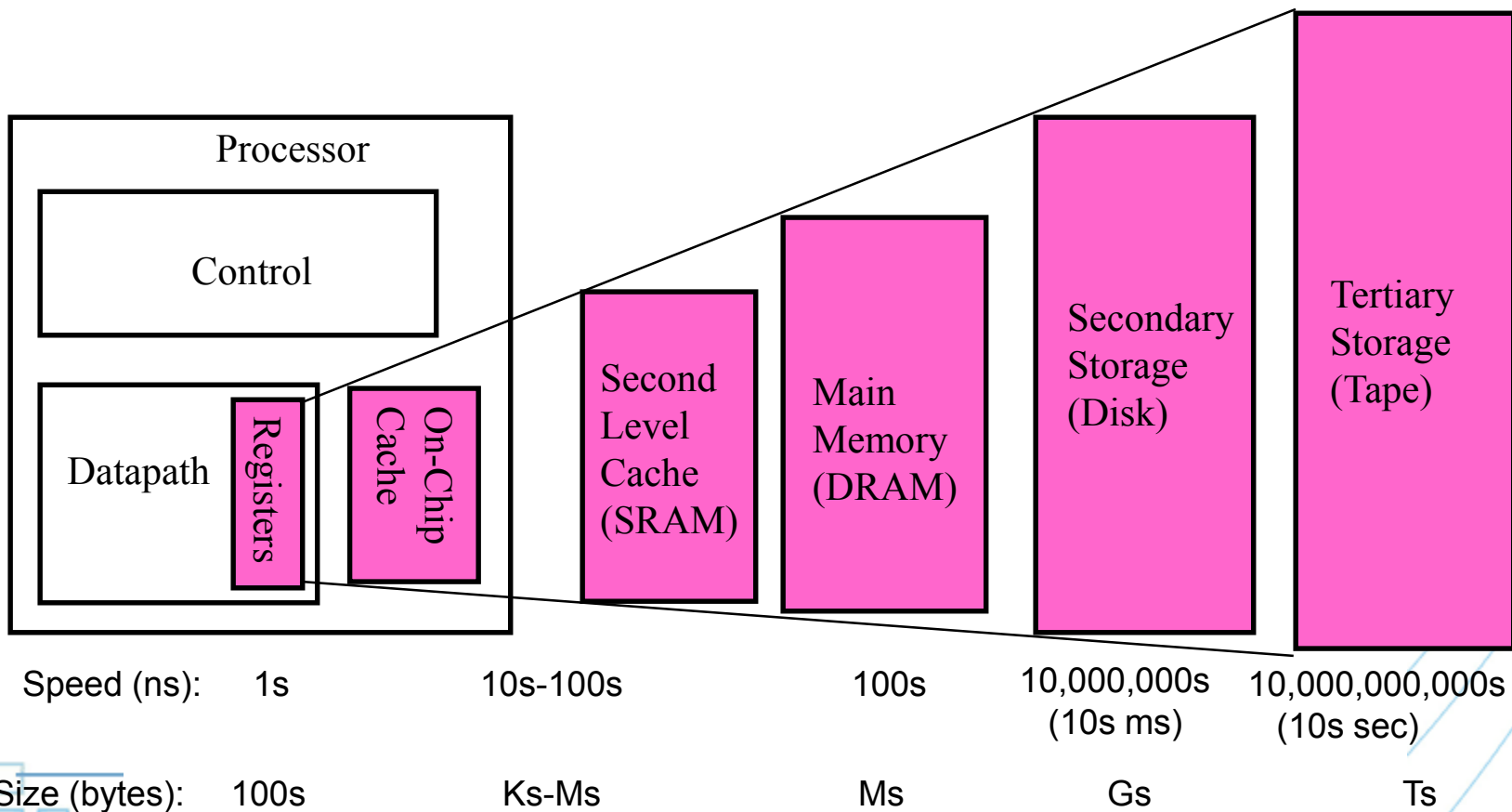
存储层次

寄存器(register)

快速缓存(cache)

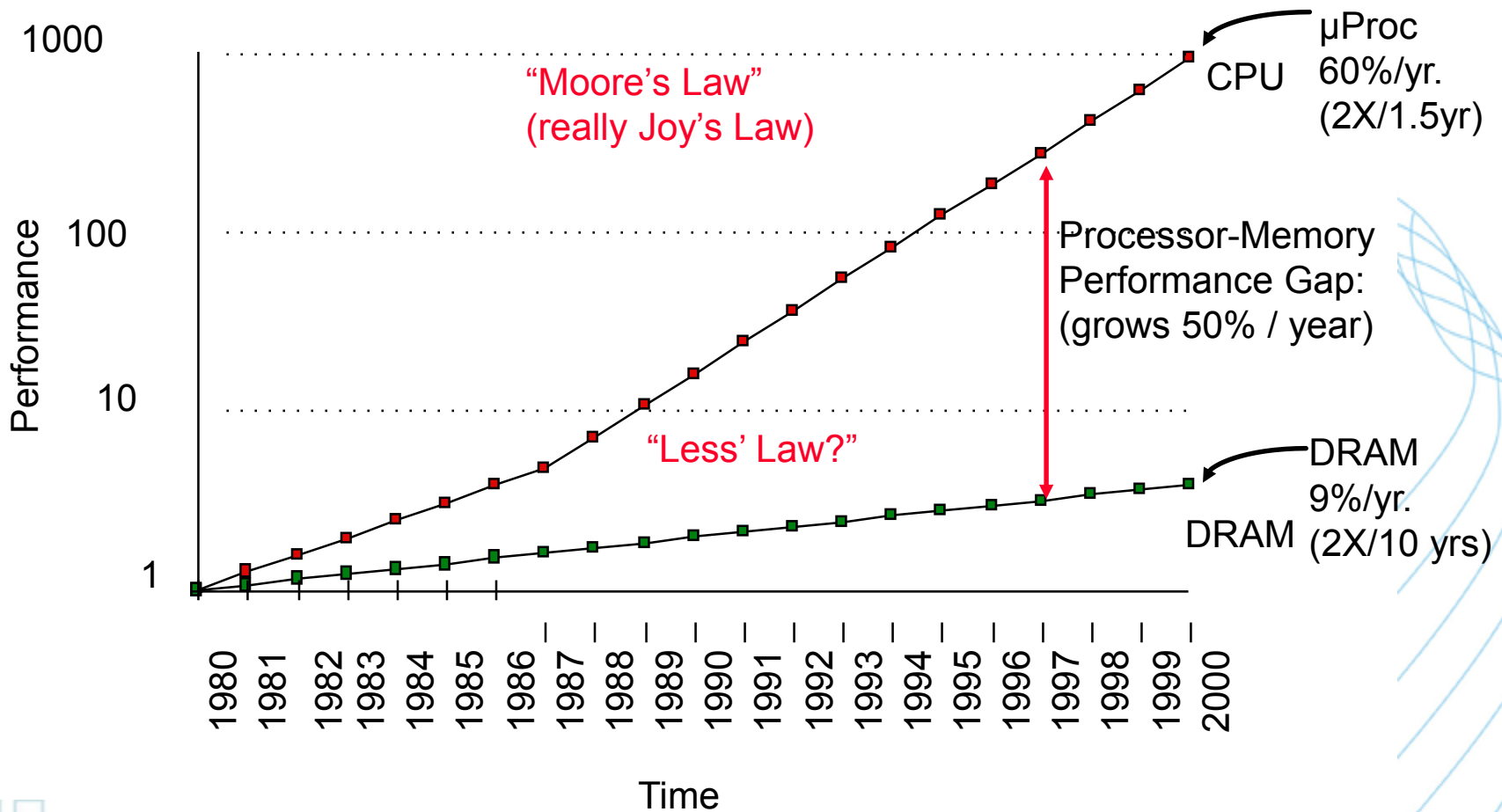
主存(primary storage)

外存(secondary storage)



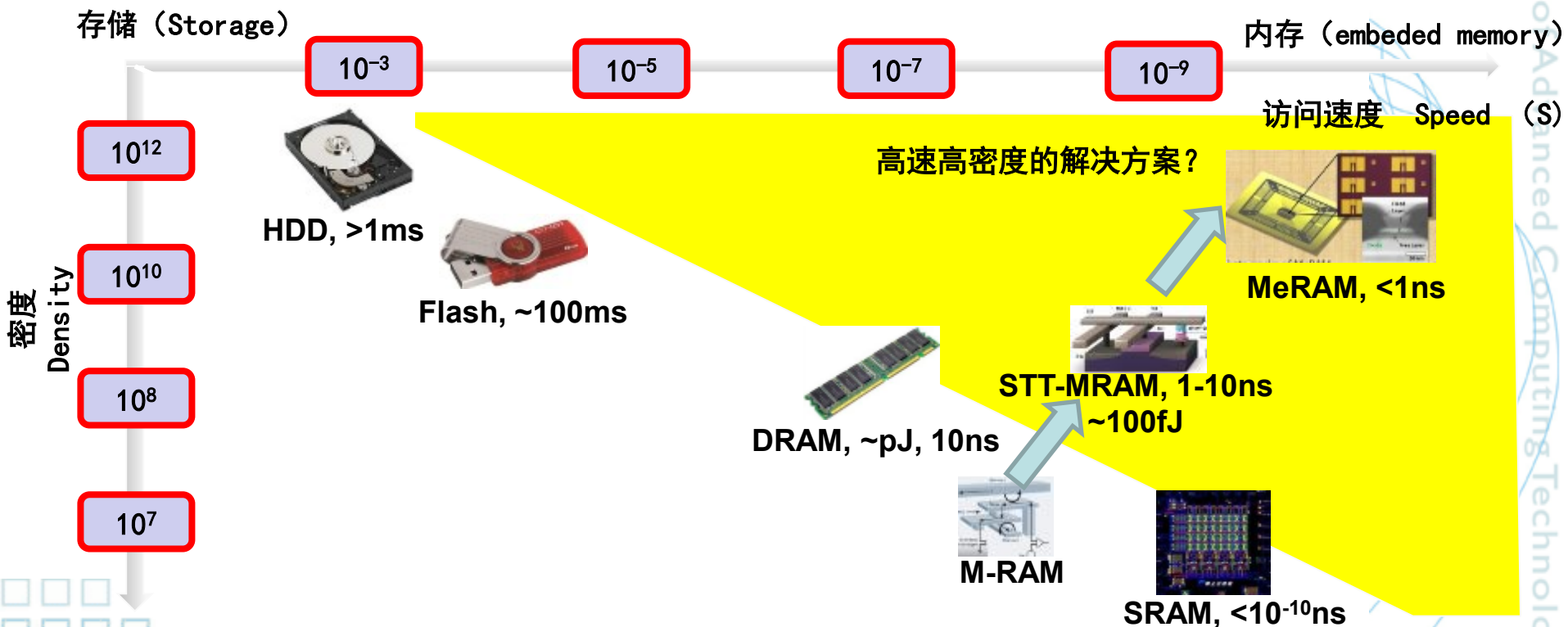


Processor-DRAM Memory Gap (latency)



蕴含巨大的科学问题和原创突破

- 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





为什么进行存储管理

- 重要的资源
- 帕金森定律(Parkinson):
 - 存储器有多大，程序有多长。
 - 多道程序，永远不够用（共享→竞争→管理）

需求与存储管理的目标

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；



需求与存储管理的目标

存储管理的基本目标：

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

存储管理要解决的问题：**分配和回收**





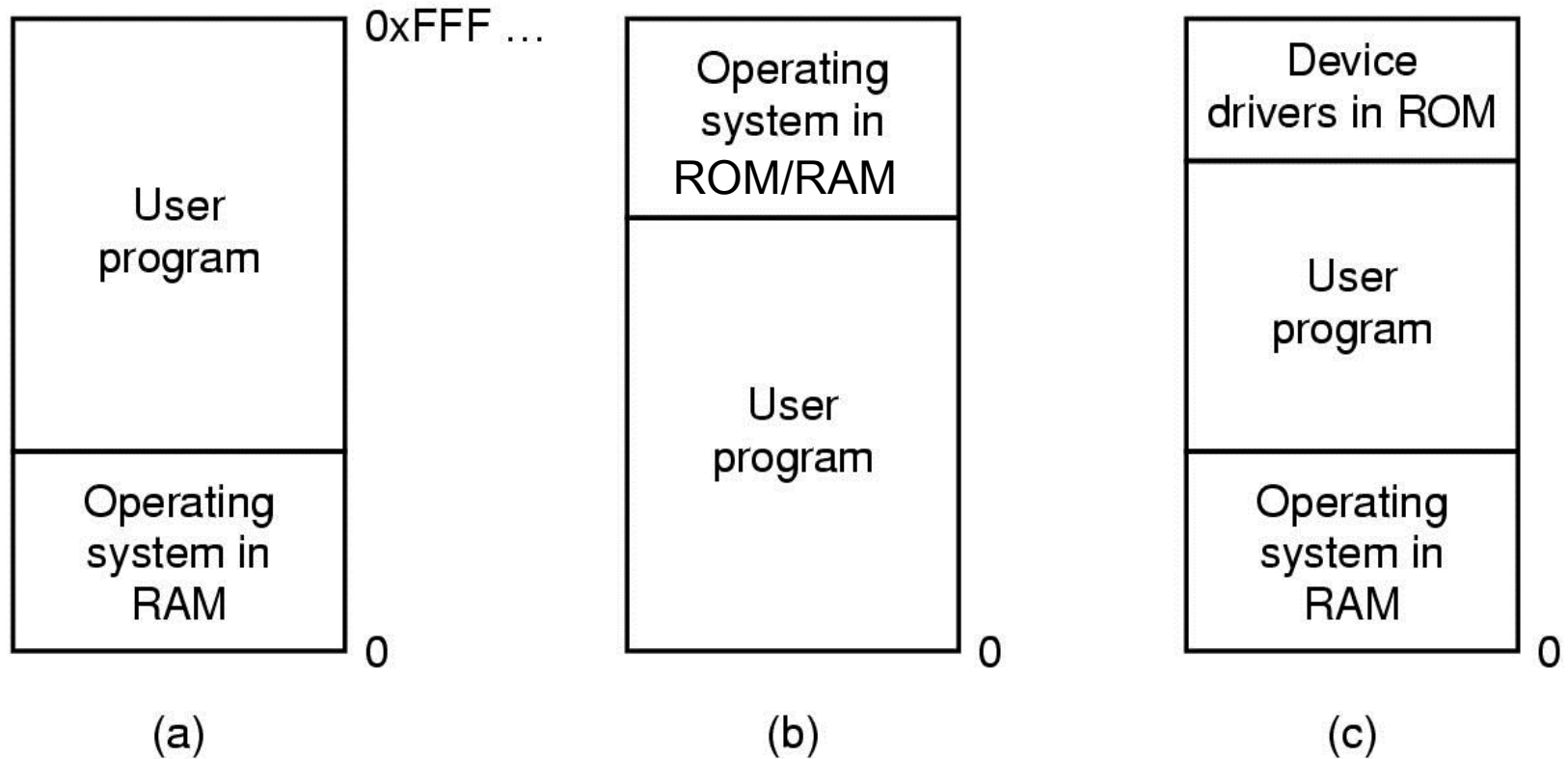
存储管理的功能

- **存储分配和回收**：是存储管理的主要内容。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。
- **存储共享和保护**：代码和数据共享，对地址空间的访问权限（读、写、执行）。
- **存储器扩充**：它涉及存储器的逻辑组织和物理组织；
 - 由应用程序控制：覆盖；
 - 由OS控制：交换（整个进程空间），请求调入和预调入（部分进程空间）

几个概念

- 1. **地址空间**：源程序经过编译后得到的目标程序，存在于它所限定的地址范围内，这个范围称为地址空间。简言之，地址空间是**逻辑地址**的集合。
- 2. **存储空间**：存储空间是指主存中一系列存储信息的物理单元的集合，这些单元的编号称为物理地址或绝对地址。简言之，存储空间是**物理地址**的集合。

操作系统在内存中的位置



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：一个用户程序和操作系统。
- 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。

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：

- 最简单，适用于单用户、单任务的OS。CP/M和DOS
- 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；不区分常用/非常用数据；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？



多道程序的存储管理

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：

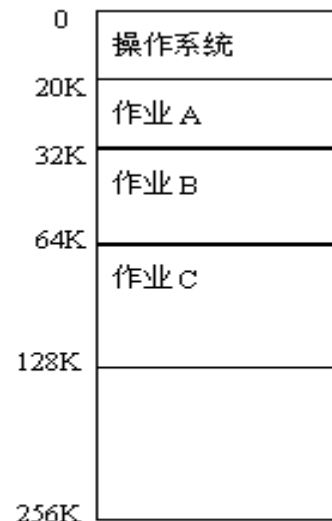
- 固定（静态）式分区分配，程序适应分区。
- 可变（动态）式分区分配，分区适应程序。

固定式分区

- 固定式分区（静态存储区域）：当系统初始化时，把存储空间划分成若干个任意大小的区域；然后，把这些区域分配给每个用户作业。

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



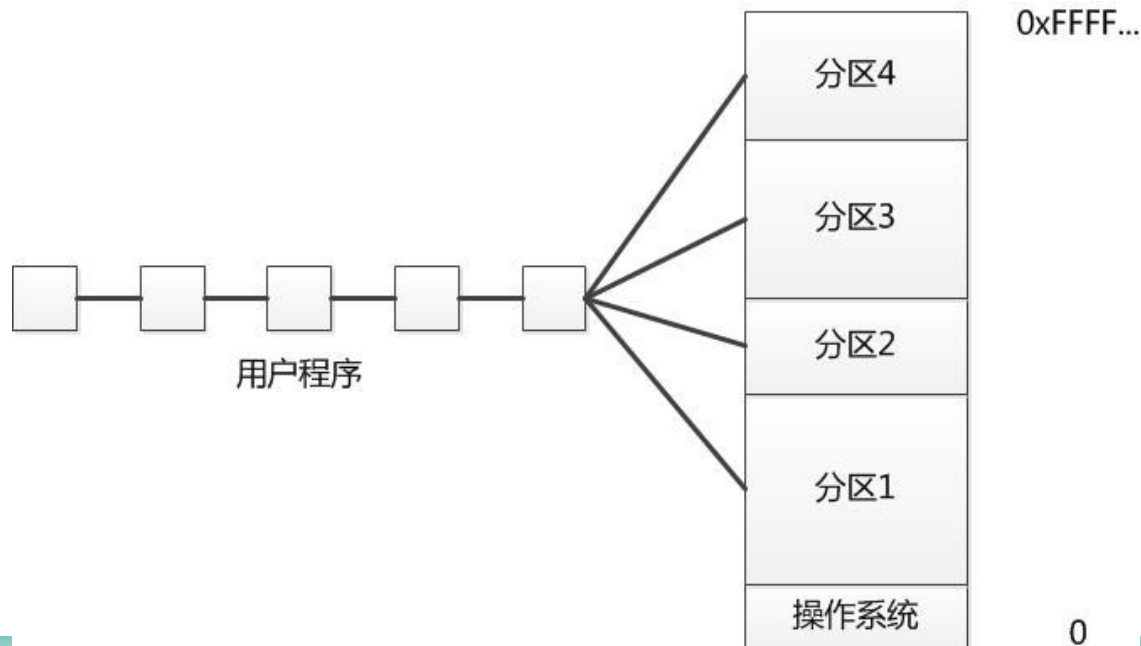
存储空间分配情况

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。
- 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

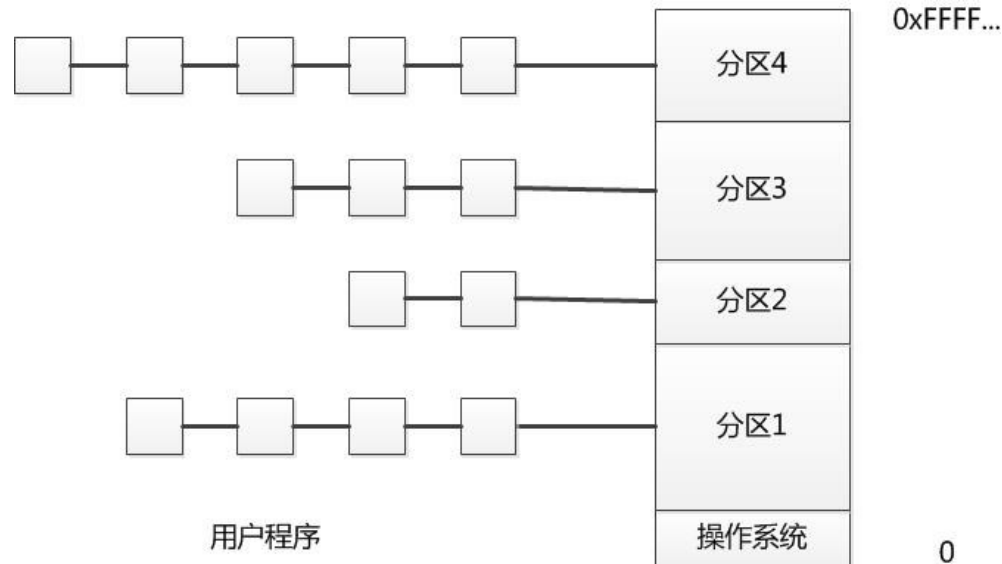
单一队列的分配方式

- 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。





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。



系统中的碎片

- 内存中无法被利用的存储空间称为碎片。

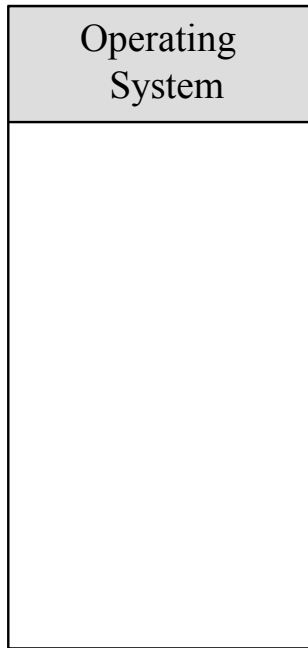
内部碎片：

- 指分配给作业的存储空间中未被利用的部分，如固定分区中存在的碎片。
- 单一连续区存储管理、固定分区存储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

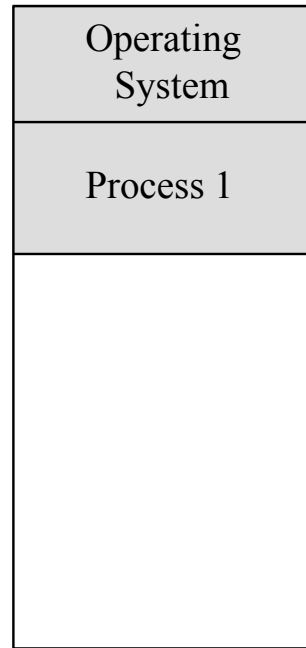
外部碎片：

- 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。



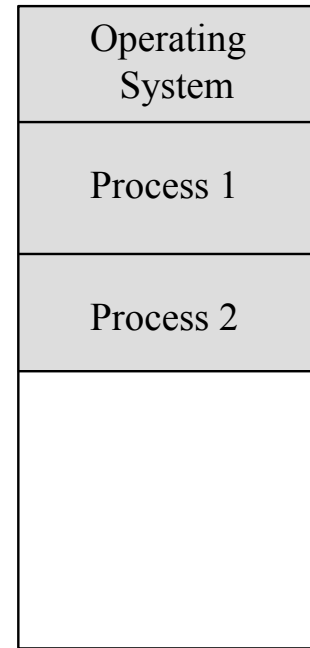
128 K

896 K



320 K

576 K



320 K

224 K

352 K



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



Operating System
Process 4
Process 3

320 K

128 K

96 K

288 K

64 K

Operating System
Process 2
Process 4
Process 3

224 k

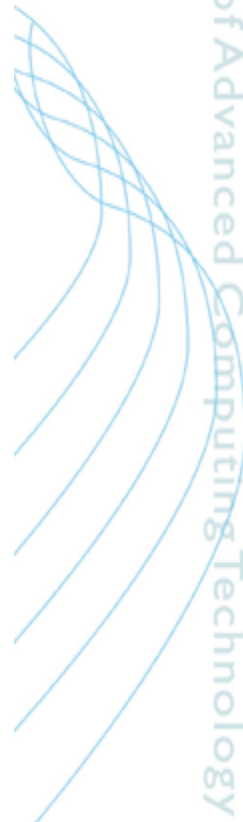
96 K

128 K

96 K

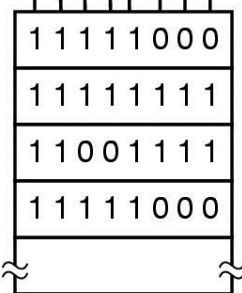
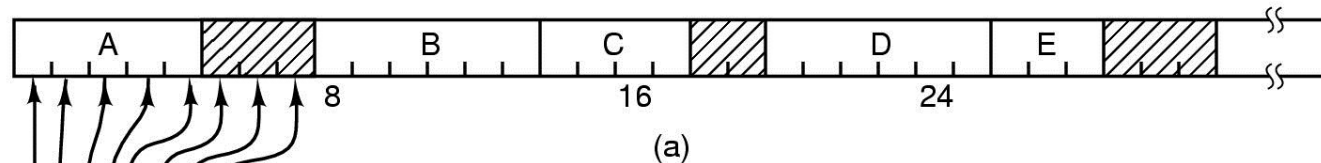
288 K

64 K

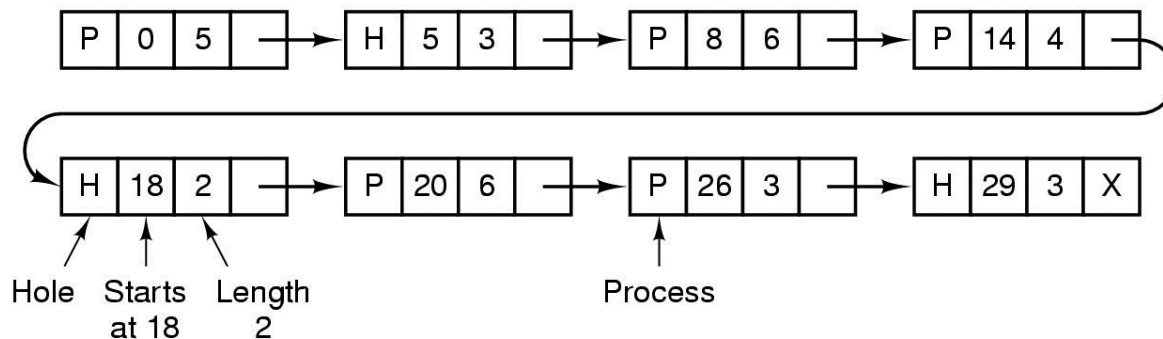


闲置空间的管理

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



(b)



(c)

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。

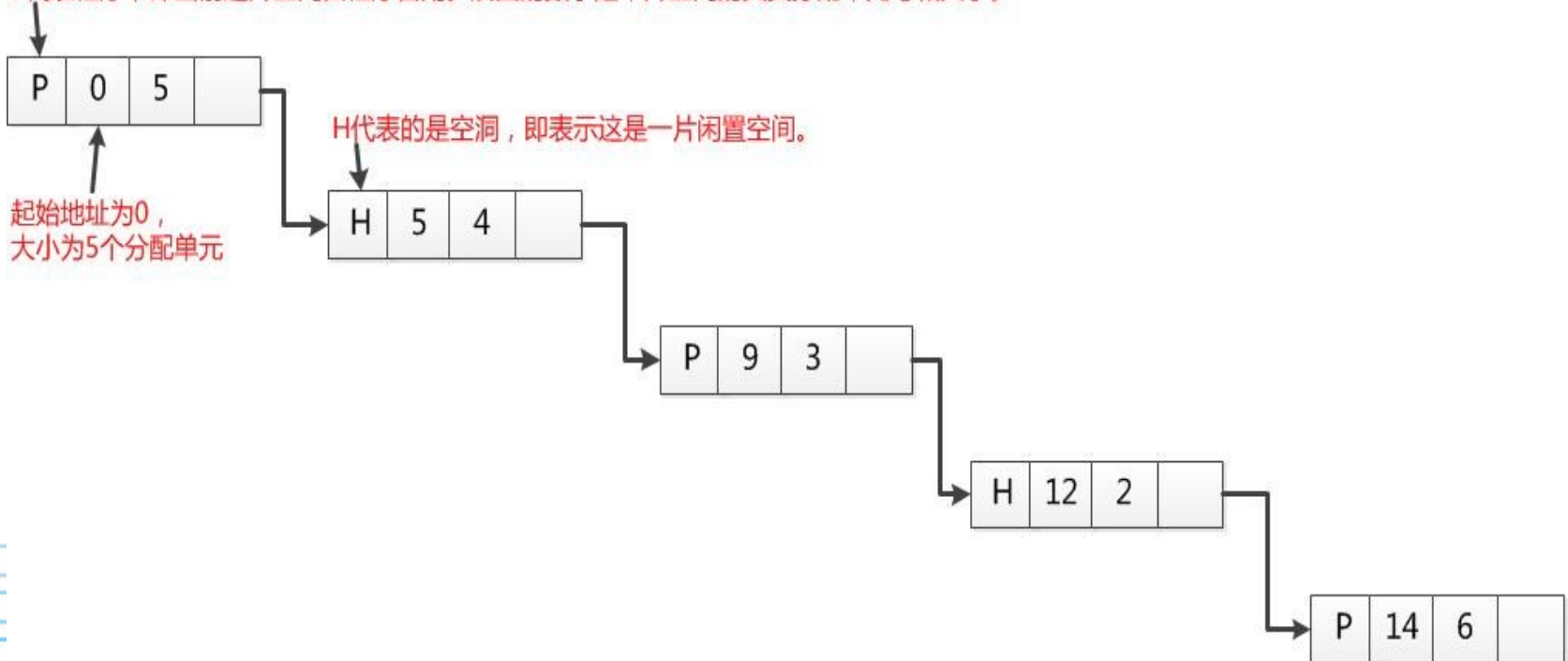
1	1	1	1	1	1	0	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---

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。如上图所示的位图所表示的内存分配状态，使用链表来表示的话则会如下图所示

P代表程序，即当前这片空间由程序占用。后面的数字是本片空间的其实分配单元号和大小。



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：

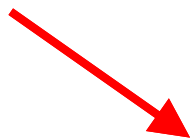
- 空间成本：取决于程序的数量。
- 时间成本：链表扫描通常速度较慢，还要进行链表项的插入、删除和修改。
- 有一定容错能力：因为链表有被占空间和闲置空间的表项，可以相互验证。

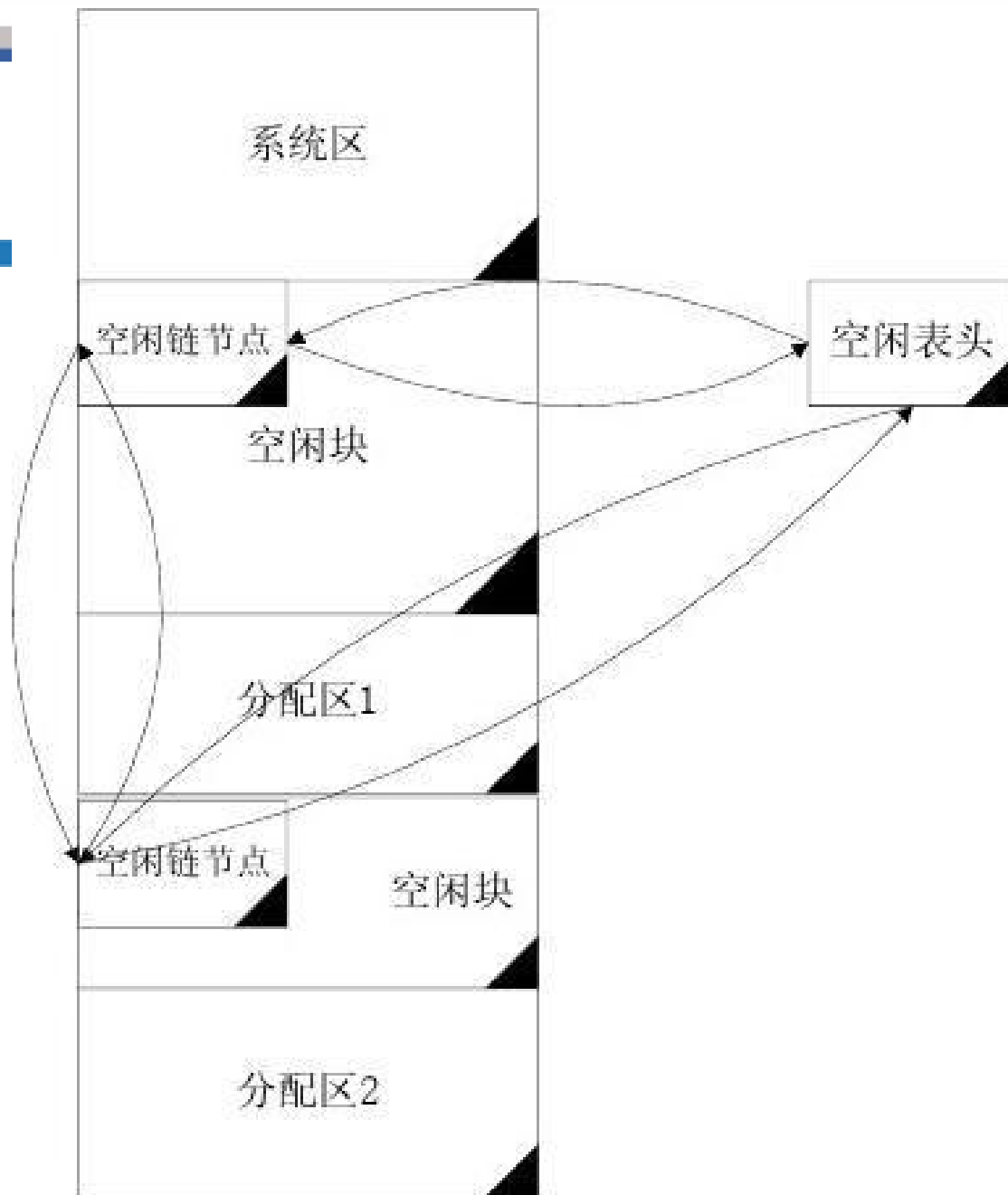
可变分区的管理

- 内存分配采用两张表：已分配分区表和未分配分区表。
- 每张表的表项为存储控制块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

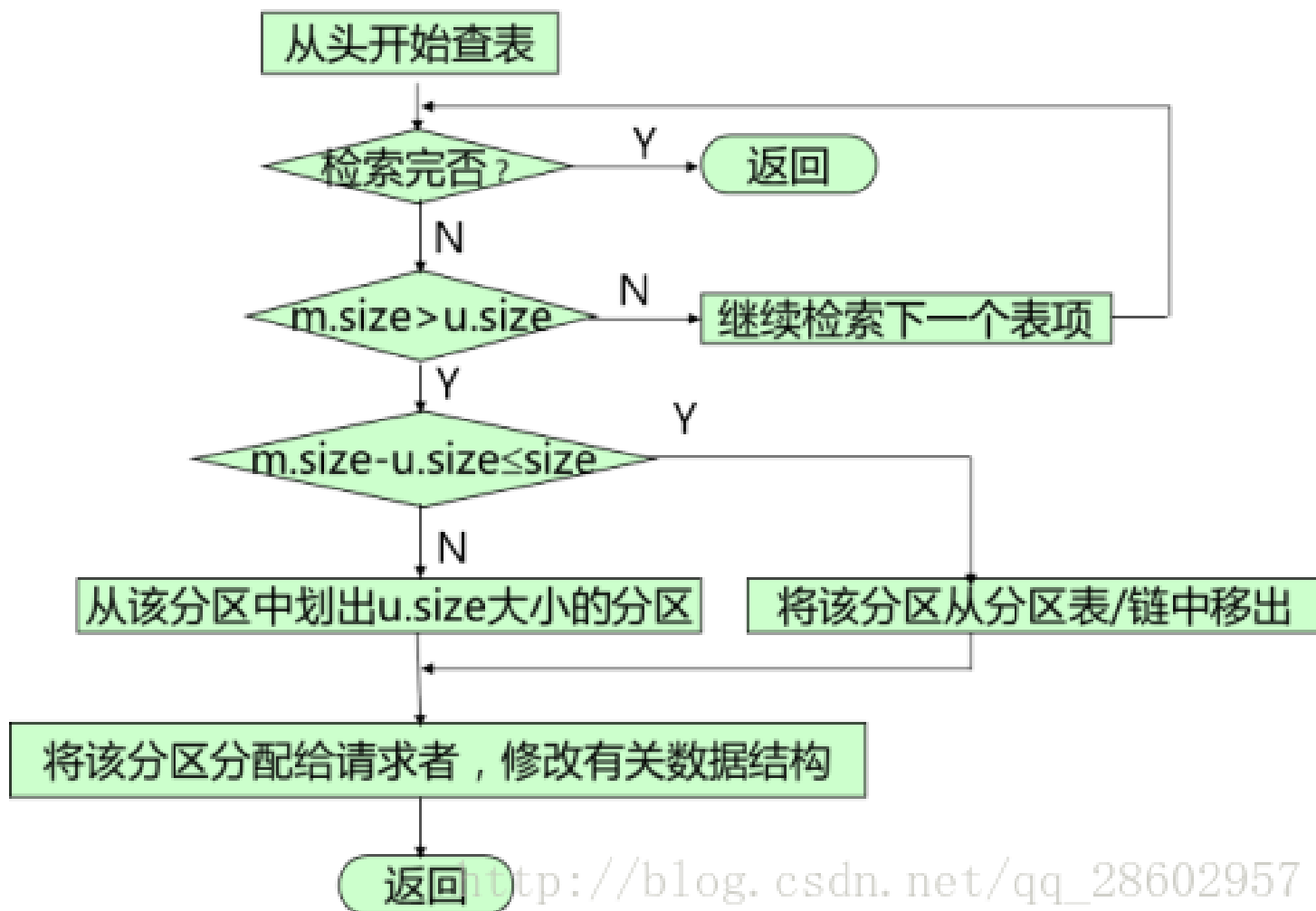




分区分配操作（分配内存）

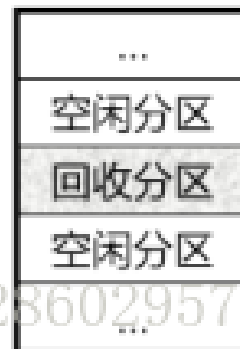
分配内存

- 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 $m.size - u.size \leq size$, 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。



分区分配操作（回收内存）

- **情况1：上空下不空。**回收分区上邻接一个空闲分区，合并后首地址为空闲分区的首地址，大小为二者之和。
- **情况2：下空上不空。**回收分区下邻接一个空闲分区，合并后首地址为回收分区的首地址，大小为二者之和。
- **情况3：上下皆空。**回收分区上下邻接空闲分区，合并后首地址为上空闲分区的首地址，大小为三者之和。
- **情况4：上下皆不空。**回收分区不邻接空闲分区，这时在空闲分区表中新建一表项，并填写分区大小等信息。



基于顺序搜索的分配算法：

1. **首次适应算法（First Fit）**：每个空白区按其在存储空间中地址递增的顺序连在一起，在为作业分配存储区域时，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）**：把存储空间中空白区构成一个循环链，每次为存储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**的存储区域。
4. **最坏适应算法（Worst Fit）**：为作业选择存储区域时，**总是寻找最大的空白区**。



动态分区的操作和数据结构

序号P	大小	起址	状态
1	8	20K	已分配
2	32	28K	已分配
3	--	--	
4	120	92K	已分配
5	--	--	
...

已分配分区（P表）

序号F	大小	起址	状态
1	32	60K	空闲
2	300	212K	空闲
3	--	--	
4	--	--	
5	--	--	
...

空闲分区（F表）

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。（不合并相邻空闲分区）

按地址递增的次序排列

区号	大小	起址	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

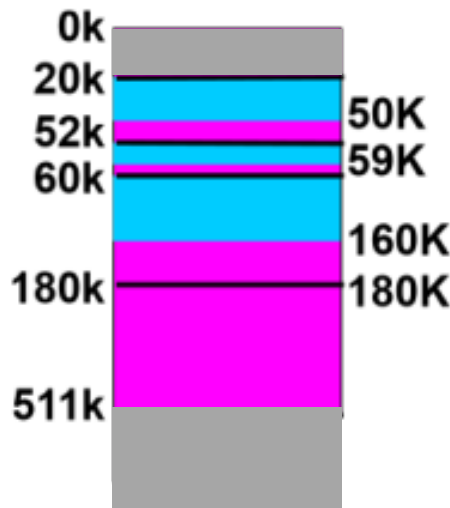
http://blog.csdn.net/yq_20092457

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；
- 申请作业 30k，分配 1 号分区，剩下分区为 2k，起始地址 50k；
- 申请作业 7k，分配 2 号分区，剩下分区为 1k，起始地址 59k。

区号	大小	起址	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

分配前



区号	大小	起址	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

分配后

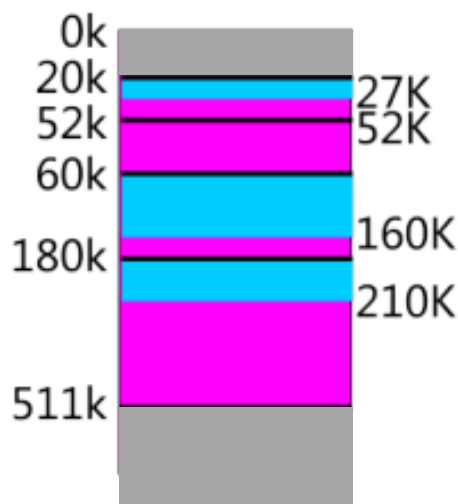


下次适应算法

- 申请作业 100k，分配 3 号分区，剩下分区为20k，起始地址 160k；
- 申请作业 30k，分配 4 号分区，剩下分区为301k，起始地址 210k；
- 申请作业 7k，分配 1 号分区，剩下分区为25k，起始地址 27k；

区号	大小	起址	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

分配前



区号	大小	起址
1	25k	27k
2	8k	52k
3	20k	160k
4	301k	210k

分配后



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

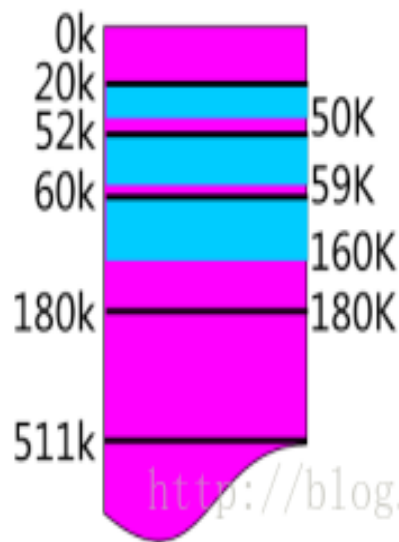
按容量递增的次序重新排列

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

http://blog.csdn.net/qq_28602957



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

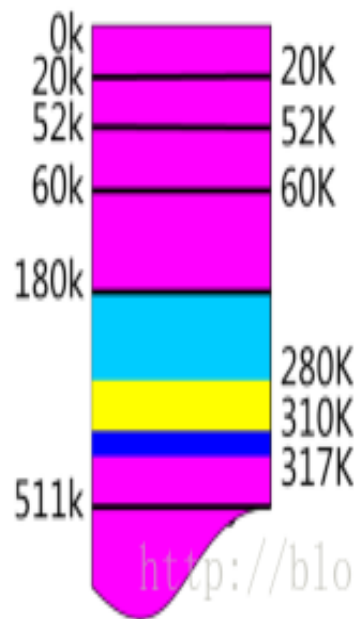
按容量递减的次序重新排列

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

http://blog.csdn.net/qq_28602957

算法特点

- 首次适应：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，在低地址会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：使存储空间的利用更加均衡，不致使小的空闲区集中在存储区的一端，但这会导致缺乏大的空闲分区。

算法特点

- 最佳适应：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，从而在存储器中留下许多难以利用的小空闲区（碎片）。
- 最坏适应：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，其存储空间的申请往往会得不到满足。

申请分配一个 x_k
大小的分区

置空闲区号 $F=1$

$F=F+1$

《 F 已超出最大项号?

是

本次无法分配

否

是

《 F 的状态=未填表项?

否

$Loc \leftarrow F$ 的起始地址

否

《 F 的大小 $\geq x_k$?

大于

F 的大小 $- x_k$ =新空闲块大小
 $Loc + x_k$ =新起始地址

等于

置 F 的状态=未填表项

在已分配表中找一个
状态=未填表项的序号 P

置 P 的大小= x_k
置 P 的始址= Loc
置 P 的=已分配

返回序号 P

FirstFit算法

可变分区的分配算法

请求回收分区R

Size ← 分区R的大小
Loc ← 分区的起始地址

已分配区说明表中
置R的状态 = 未填表项

否 分区R与F₂邻接?

是

Size ← Size + F₂的大
小

是

分区R与F₁邻接?

否

否

分区R与F₁邻接?

是

在空闲分区表中
找一个未填表项

置新空闲分区
的大小=Size
始址=Loc
状态=空闲

在空闲分区表中置F₂为未填表项

置空闲分区F₁的大小
=Size + F₁的大小

置空闲分区F₂
的大小=Size
始址=Loc

返回

空闲分区回收算法

已分配分区说明

序号P	大小	起址	状态
1	8K	20K	已分配
2	32K	28K	已分配
3	—	—	未填表项
4	120K	92K	已分配
5	—	—	未填表项
...

空闲分区说明表

序号F	大小	起址	状态
1	32K	60K	空闲
2	300K	212K	空闲
3	—	—	未填表项
4	—	—	未填表项
5	—	—	未填表项
...

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。

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**：在分配存储块时将一个大的存储块分裂成两个**大小相等**的小块，这两个小块就称为“伙伴”。

Linux内核使用二进制伙伴算法来管理和分配物理内存页面。

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

ACT

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。

http://t.qq_28602957

伙伴系统的内存分配

ACT

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。

http://blog.csdn.net/qq_28602957

伙伴系统的内存释放

ACT

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。

注意：

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。

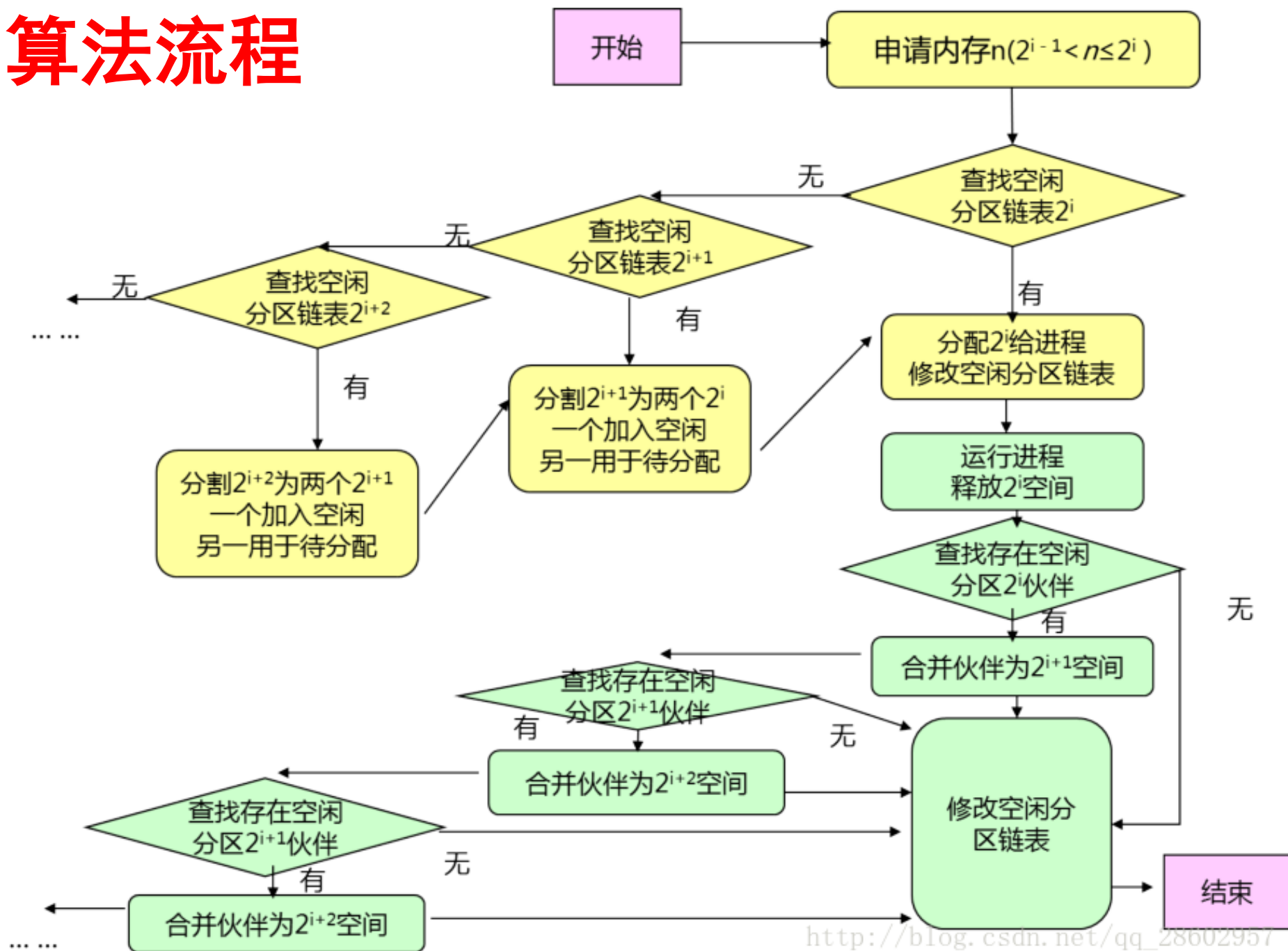


一个例子

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 http://blog.csdn.net/qq_28602957					

算法流程



伙伴系统

1. 是否会产生内碎片？
2. 是否会产生外碎片？
3. 最大内碎片多大？
4. 如果既会产生内碎片，又会产生外碎片，那伙伴系统有什么优势？



碎片



相邻相同大小块，不一定是buddy

伙伴系统特点

AS

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。



linux: /proc/buddyinfo

```
$ cat /proc/buddyinfo
```

Node 0, zone	DMA.	0	0	0	0	0	0	...
Node 0, zone	DMA32	2	1	2	3	2	0	...
Node 0, zone	Normal	32000	30873	8499	8073	1048	1251	...

- 有32000个 2^0 个页组成的buddy
- 有30873个 2^1 个页组成的buddy
- 有8499个 2^2 个页组成的buddy
- 有8073个 2^3 个页组成的buddy
- 有1048个 2^4 个页组成的buddy
- 有1251个 2^5 个页组成的buddy
- ...