
河南大学
操作系统
计算机学院

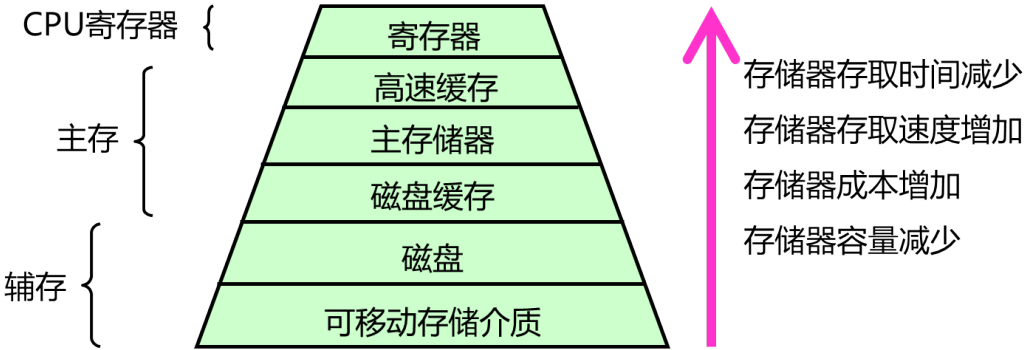
第 4 章

存储器管理

大纲

存储器的层次结构

- 按照速度、容量和成本划分，存储器系统构成一个层次结构。

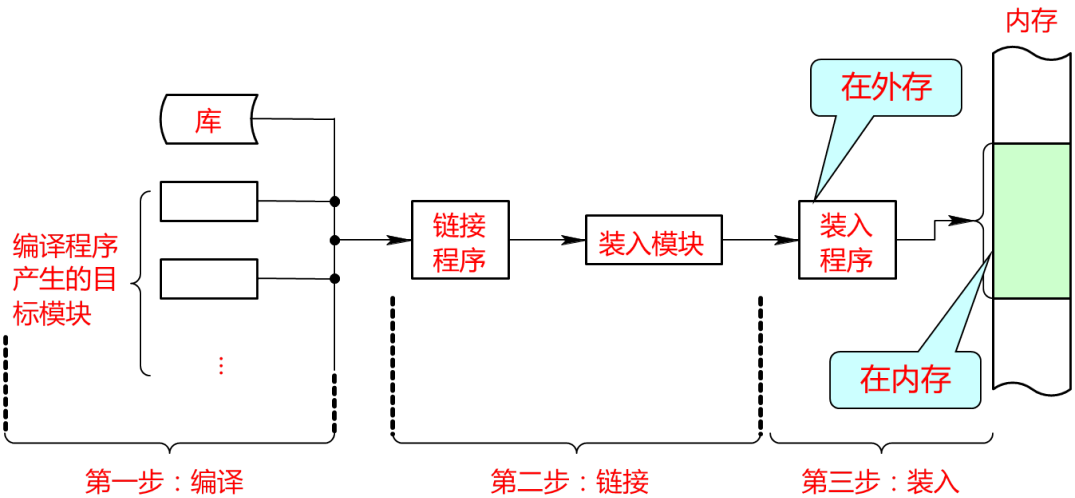


程序的装入和链接

一个用户源程序要变为在内存中可执行的程序，通常要进行以下处理：

- 编译：由编译程序将用户源程序编译成若干个目标模块。
- 链接：由链接程序将目标模块和相应的库函数链接成装入模块。
- 装入：由装入程序将装入模块装入内存。

程序的装入和链接



基本概念

- 逻辑地址（相对地址，虚地址）
  - 其首地址为 0，其余指令中的地址都相对于首地址而编址。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。
  - 不能用逻辑地址在内存中读取信息。
- 物理地址（绝对地址，实地址）
  - 内存中存储单元的地址，可直接寻址。
- 地址映射（地址转换）
  - 为了保证 CPU 执行指令时可正确访问存储单元，需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，这一过程称为地址映射。

程序的装入

程序的装入方式

1. 绝对装入方式
2. 可重定位装入方式
3. 动态运行时装入方式

绝对装入方式

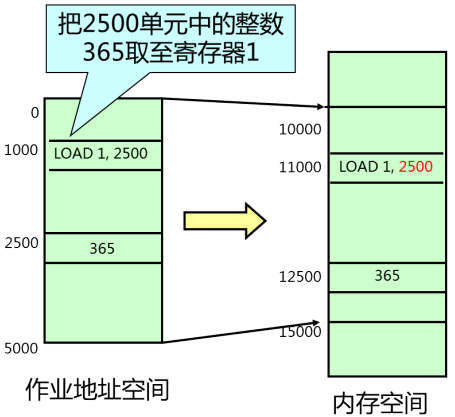
- 在编译时，如果能够事先知道程序将驻留在内存的什么位置，那么编译程序将产生绝对地址的目标代码。
- 绝对装入程序按照装入模块中的地址，将程序和数据装入内存。装入模块被装入内存后，由于程序中的逻辑地址与实际内存中的地址完全相同，故不需对程序和数据的地址进行修改。
- 该装入方式只适用于单道程序环境。
- 绝对地址既可在编译时给出，也可由程序员直接赋予。
- 要求程序员熟悉内存的使用情况。一旦程序或数据被修改后，可能要改变程序中的所有地址。

可重定位装入方式

- 重定位：由于一个作业装入到与其地址空间不一致的存储空间所引起的，需对其有关地址部分进行调整的过程就称为重定位（实质是一个地址变换过程/地址映射过程）。
- 为什么需要重定位：当程序装入内存时，操作系统要为该程序分配一个合适的内存空间，由于程序的逻辑地址与分配到内存物理地址不一致，而 CPU 执行指令时是按物理地址进行的，所以要进行地址转换。

可重定位装入方式

可重定位装入方式：事先不知用户程序在内存的驻留位置，装入程序在装入时根据内存的实际情况把相对地址（逻辑地址）转换为绝对地址，装入到适当的位置。（在装入时进行地址转换）



指令地址：相对地址1000-物理地址11000
数据地址：相对地址2500-物理地址12500

重定位基本概念

- 根据地址变换进行的时间及采用技术手段不同，可分为静态重定位和动态重定位两类。
- 静态重定位
  - 当用户程序被装入内存时，一次性实现逻辑地址到物理地址的转换，以后不再转换。
  - 一般在装入内存时由软件完成。
- 动态重定位
  - 在程序运行过程中要访问数据时再进行地址变换（即在逐条指令执行时完成地址映射）。一般为了提高效率，此工作由硬件地址映射机制来完成。
  - 硬件（寄存器）支持，软硬件结合完成。

动态运行时装入方式

- 可重定位装入方式不允许程序运行时在内存中移动位置。而实际情况是程序在内存中的位置经常要改变。
- 程序在内存中的移动意味着它的物理位置发生了变化，这时必须对程序和数据的地址 (绝对地址) 进行修改后方能运行。
- 为了保证程序在内存中的位置可以改变。装入程序把装入模块装入内存后，并不立即把装入模块中相对地址转换为绝对地址，而是在程序运行时才进行。
- 这种方式需要一个重定位寄存器来支持，在程序运行过程中进行地址转换。

程序的链接

根据链接时间的不同，可将链接分成三种：

程序的链接方式

1. 静态链接
2. 装入时动态链接
3. 运行时动态链接

静态链接

- 静态链接方式是一种事先链接方式，即在程序运行之前，先将各目标模块及它们所需的库函数，链接成一个完整的装入模块 (执行文件)，以后不再拆开。
- 实现静态链接应解决的问题：
  1. 相对地址的修改
  2. 变换外部调用符号
- 存在问题：
  - 不便于对目标模块的修改和更新。
  - 无法实现对目标模块的共享。

静态链接

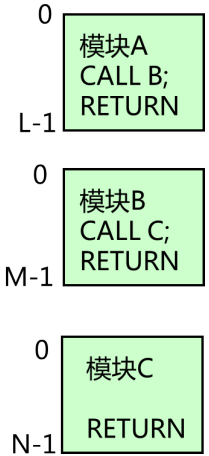
相对地址的修改

B中的所有相对地址都加上L，C中的所有相对地址都加上L+M。

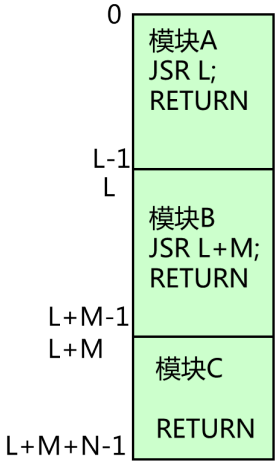
变换外部调用符号

B、C在目标模块中是外部调用符号。

B的起始地址变换为L，把C的起始地址变换为L+M。CALL改为JSR（跳转）。



(a)目标模块



(b)装入模块

装入时动态链接

- 装入时动态链接方式是把一组目标模块在装入内存时边装入边链接的方式。
- 各目标模块分开存放，便于修改和更新。
- 便于实现对目标模块的共享。
- 存在问题：
  - 由于把程序运行所有可能用到的目标模块在装入时均全部链接在一起，所以将会把一些可能不会运行的目标模块也链接进去。如程序中的错误处理模块。

运行时动态链接

- 运行时动态链接方式
  - 由于事先无法知道要运行哪些模块，装入时动态链接方式只能是将所有可能要运行到的模块都全部装入内存，并在装入时全部链接在一起，显然这是低效的。
  - 运行时动态链接方式在程序运行中需要某些目标模块时，才对它们进行链接的方式。具有高效且节省内存空间的优点。

连续分配方式

连续分配方式（分区技术）：指为一个用户程序分配一片连续的内存空间。

- 静态分区：作业装入时一次完成，分区大小及边界在运行时不能改变。
- 动态分区：根据作业大小动态地建立分区，分区的大小、数目可变。

连续分配方式

连续分配方式（分区技术）

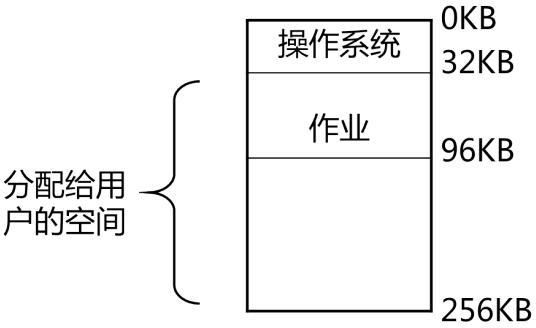
- 单一连续分区分配（静态分区技术）：仅用于单用户单任务系统
- 固定分区分配（静态分区技术）：可用于多道系统
- 动态分区分配（动态分区技术）
- 动态可重定位分区分配（动态分区技术）：引入了动态重定位和内存紧凑技术
- 伙伴系统（动态分区技术）

单一连续分区分配

- 存储管理方法：将内存分为系统区（内存低端，分配给 OS 用）和用户区（内存高端，分配给用户用）。
- 采用静态分配方式，即作业一旦进入内存，就要等待它运行结束后才能释放内存。
- 最简单的一种存储管理方式，但只能用于单用户、单任务的 OS 中。
- 主要特点：管理简单，只需少量的软件和硬件支持，便于用户了解和使用。但因内存中只装入一道作业运行，内存空间浪费大，各类资源的利用率也不高。

单一连续分区分配

- 一个容量为 256KB 的内存，操作系统占用 32KB，剩下 224KB 全部分配给用户作业，如果一个作业仅需 64KB，那么就有 160KB 的存储空间被浪费。

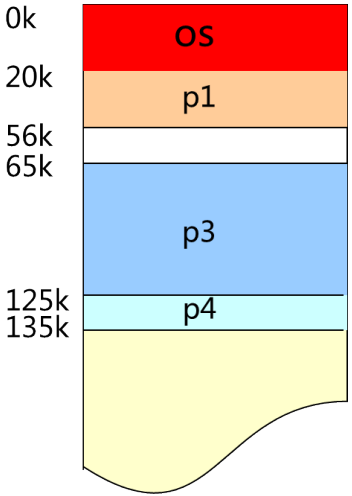


固定分区分配方式

- 固定分区分配方式是最早使用的一种可运行多道程序的存储管理方法。
- 存储管理方法：将内存空间划分为若干个固定大小的分区，除 OS 占一区外，其余的一个分区装入一道程序。分区的大小可以相等，也可以不等，但事先必须确定，在运行时不能改变。即分区大小及边界在运行时不能改变。
- 系统需建立一张分区说明表或使用表，以记录分区号、分区大小、分区的起始地址及状态（已分配或未分配）。

固定分区分配方式

区号	起址	大小	状态
1	20k	36k	已分配
2	56k	9k	未分配
3	65k	60k	已分配
4	125k	10k	已分配



固定分区分配方式

- 划分分区的方法
  - 分区大小相等

    用于利用一台计算机去控制多个相同对象的场合。

    缺点是缺乏灵活性。
  - 分区大小不等

    把内存区划分成含有多个较小的分区、适量的中等分区及少量的大分区。

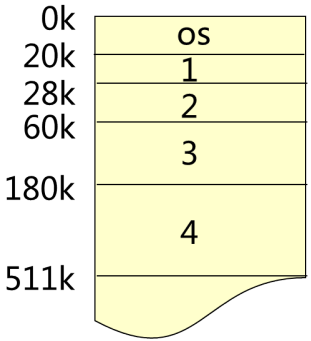
固定分区分配方式

内存分配

- 当某个用户程序要装入内存时，通常将分区按大小进行排队，由内存分配程序检索分区说明表，从表中找出一个满足要求的尚未分配的分区分配该程序，同时修改说明表中相应分区的状态；若找不到大小足够的分区，则拒绝为该程序分配内存。
- 程序执行完毕，释放占用的分区，管理程序修改说明表中相应分区的状态为未分配，实现内存资源的回收。
- 主要特点：管理简单，但因作业的大小并不一定与某个分区大小相等，从而使一部分存储空间被浪费。所以主存的利用率不高。

固定分区分配方式

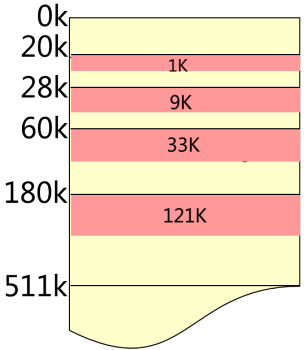
例：在某系统中，采用固定分区分配管理方式，内存分区（单位：字节）情况如图所示，现有大小为 1K、9K、33K、121K 的多个作业要求进入内存。



区号	起址	大小	状态
1	20k	8k	未分配
2	28k	32k	未分配
3	60k	120k	未分配
4	180k	331k	未分配

固定分区分配方式

根据分区说明表，将 4 个分区依次分配给 4 个作业，同时修改分区说明表，其内存分配和分区说明表如下所示：



区号	起址	大小	状态
1	20k	8k	已分配
2	28k	32k	已分配
3	60k	120k	已分配
4	180k	331k	已分配

浪费空间 =(8-1)+(32-9)+(120-33)+(331-121)=327(k)

动态分区分配方式（可变分区分配）

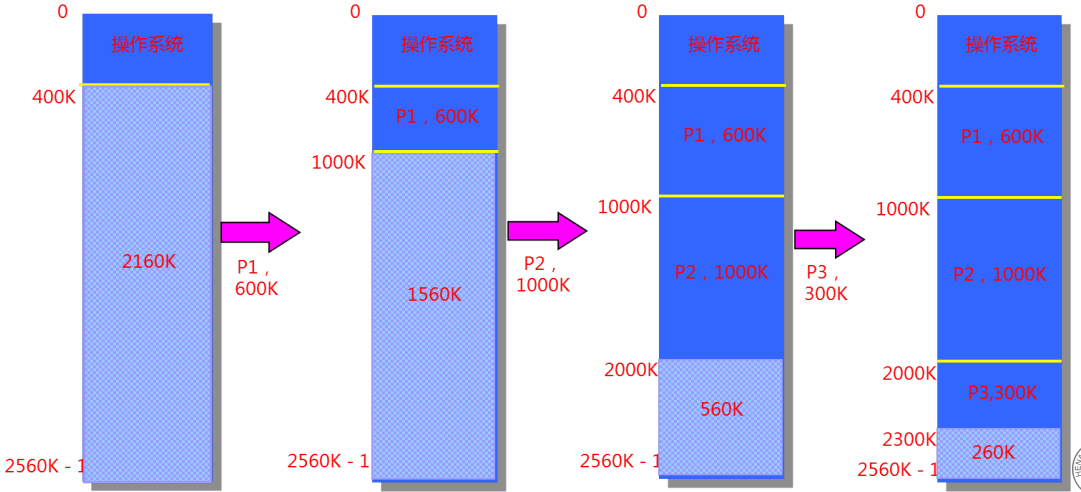
- 动态分区分配是一种动态划分存储器的分区方法。
- 存储管理方法
  - 内存不是预先划分好的，作业装入时，根据作业的需求和内存空间的使用情况来决定是否分配。
  - 若有足够的空间，则按需要分割一部分分区给该进程；否则令其等待内存空间。
- 主要特点：管理简单，只需少量的软件和硬件支持，便于用户了解和使用，主存的利用率有所提高。

动态分区分配方式

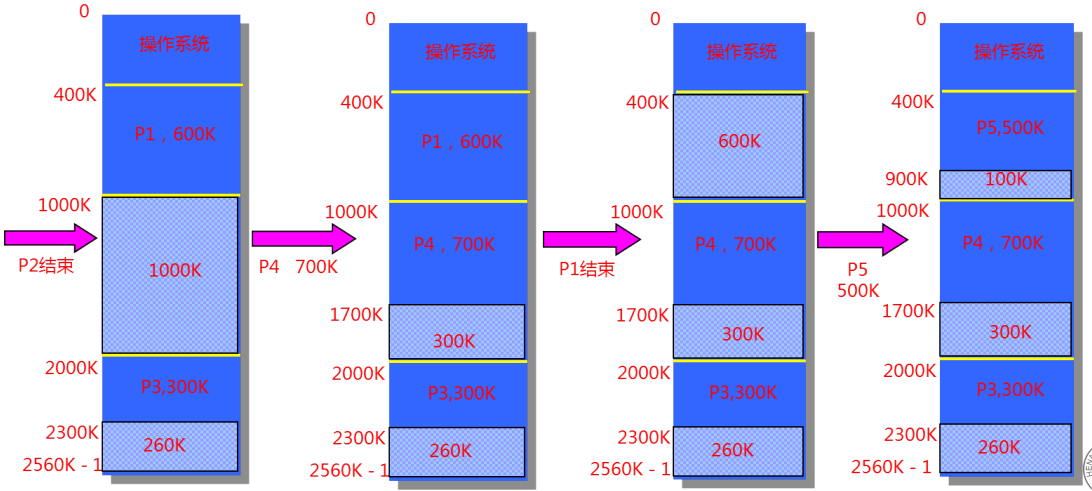
例：一个计算机有 2560K 内存，采用可变分区模式管理内存，操作系统占用低地址的 400K 空间，剩余 2160K 的空间为用户区。

- 最初整个用户区是空闲的，只有一个分区。然后随着用户程序的创建和撤销。分区的个数和大小位置开始变化。

动态分区分配方式

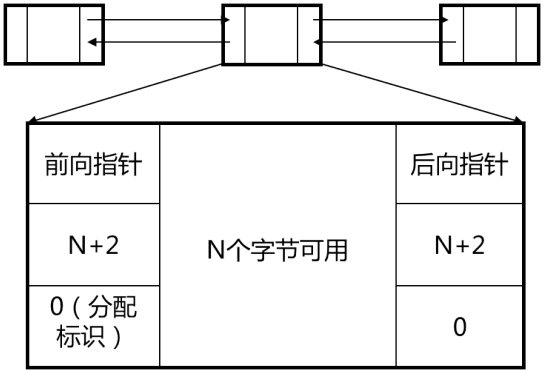

0
操作系统
400K
2160K
2560K - 1
P1，600K
0
操作系统
400K
P1，600K
1000K
1560K
2560K - 1
P2，1000K
0
操作系统
400K
P1，600K
1000K
P2，1000K
2000K
560K
2560K - 1
P3，300K
0
操作系统
400K
P1，600K
1000K
P2，1000K
2000K
P3,300K
2300K
260K
2560K - 1

动态分区分配方式


0
操作系统
400K
P1，600K
1000K
1000K
2000K
P3,300K
2300K
260K
2560K - 1
P2结束
P4　700K
0
操作系统
400K
P1，600K
1000K
P4，700K
1700K
300K
2000K
P3,300K
2300K
260K
2560K - 1
P1结束
0
操作系统
400K
600K
1000K
P4，700K
1700K
300K
2000K
P3,300K
2300K
260K
2560K - 1
P5
500K
0
操作系统
400K
P5,500K
900K
100K
1000K
P4，700K
1700K
300K
2000K
P3,300K
2300K
260K
2560K - 1

动态分区分配中的数据结构

- 空闲分区表：用来登记系统中的空闲分区（分区号、分区起始地址、分区大小及状态）。
- 空闲分区链：

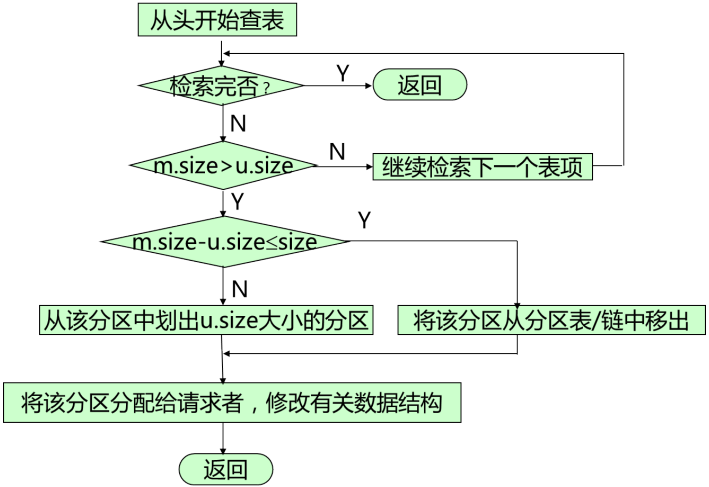


- 前、后向链接指针用于把所有的空闲分区链接成一个双向链。当分区被分配出去以后，前、后向指针无意义。
- 状态位 0：未分配
- 状态位 1：已分配

分区分配操作（分配内存）

- 系统利用某种分配算法，从空闲分区表/链中找到所需大小的分区。
- 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≤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（分配内存）


从头开始查表
检索完否？
Y
返回
N
m.size>u.size
N
继续检索下一个表项
Y
m.size-u.size≤size
Y
N
从该分区中划出u.size大小的分区
将该分区从分区表/链中移出
将该分区分配给请求者，修改有关数据结构
返回

分区分配操作（回收内存）

- 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。
- 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。
- 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。
- 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。

动态分区分配算法

动态分区分配算法

1. 基于顺序搜索的动态分区分配算法
   - 首次适应算法（First Fit）
   - 循环首次适应算法（Next Fit）
   - 最佳适应算法（Best Fit）
   - 最坏适应算法（Worst Fit）
2. 基于索引搜索的动态分区分配算法
   - 快速适应算法（Quick Fit）
   - 伙伴系统（Buddy system）
   - 哈希算法

基于顺序搜索的动态分区分配算法

按照一定的分配算法从空闲分区表（链）中选出一个满足作业需求的分区分割，一部分分配给作业，剩下的部分仍然留在空闲分区表（链）中，同时修改空闲分区表（链）中相应的信息。

顺序搜索动态分区分配算法

- 首次适应算法（First Fit）
- 循环首次适应算法（Next Fit）
- 最佳适应算法（Best Fit）
- 最坏适应算法（Worst Fit）

首次适应算法

- 空闲分区（链）按地址递增的次序排列。
- 在进行内存分配时，从空闲分区表/链首开始顺序查找，直到找到第一个满足其大小要求的空闲分区为止。
- 然后再按照作业大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲分区表（链）中。

首次适应算法

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的内存分配情况及分配后空闲分区表。

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

首次适应算法

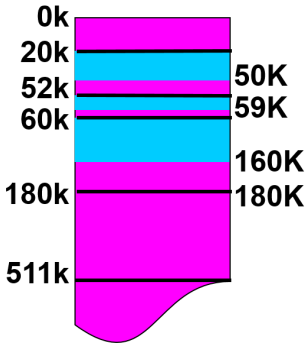
例：系统中的空闲分区表如下表示，现有三个作业分配申请内存空间 100K、30K 及 7K，给出按首次适应算法的内存分配情况及分配后空闲分区表。

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

按首次适应算法，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。

首次适应算法



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

首次适应算法的特点：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。

循环首次适应算法

- 循环首次适应算法又称为下次适应算法，由首次适应算法演变而来。在为作业分配内存空间时，不再每次从空闲分区表/链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直到找到第一个能满足其大小要求的空闲分区为止。
- 然后再按照作业大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲分区表（链）中。

循环首次适应算法

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，给出按循环首次适应算法的内存分配情况及分配后空闲分区表。

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

循环首次适应算法

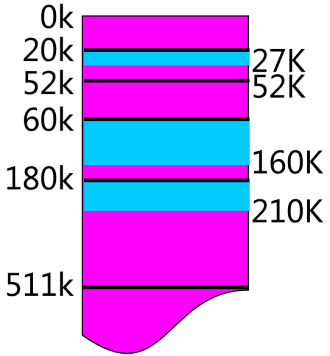
例：系统中的空闲分区表如下表示，现有三个作业分配申请内存空间 100K、30K 及 7K，给出按循环首次适应算法的内存分配情况及分配后空闲分区表。

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

按循环首次适应算法，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 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。

循环首次适应算法



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

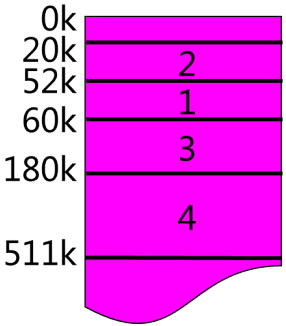
循环首次适应算法的特点：使存储空间的利用更加均衡，不致使小的空闲区集中在存储区的一端，但这会导致缺乏大的空闲分区。

最佳适应算法

- 空闲分区表/链按容量大小递增的次序排列。在进行内存分配时，从空闲分区表/链首开始顺序查找，直到找到第一个满足其大小要求的空闲分区为止。
- 按这种方式为作业分配内存，就能把既满足作业要求又与作业大小最接近的空闲分区分配给作业。将剩余空闲分区仍留在空闲分区表/链中。
- 第一次找到的能满足要求的空闲区必然是最佳的。

最佳适应算法

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，给出按最佳适应算法的内存分配情况及分配后空闲分区表。



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

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

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

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

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

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

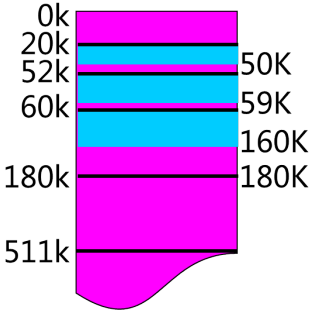
作业100K分配后

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

最佳适应算法



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

最佳适应算法的特点：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，从而在存储器中留下许多难以利用的小空闲区（碎片）。

最坏适应算法

- 空闲分区表/链按容量大小递减的次序排列。在进行内存分配时，从空闲分区表/链首开始顺序查找，直到找到第一个满足其大小要求的空闲分区为止。
- 总是挑选一个最大的空闲区分割给作业使用，其优点是可使剩下的空闲区不至于太小，产生碎片的几率最小，对中、小作业有利。

最坏适应算法

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，给出按最佳适应算法的内存分配情况及分配后空闲分区表。

按容量大小递减的次序排列

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

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

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

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

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

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

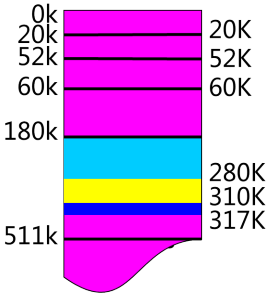
作业100K分配后

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

最坏适应算法



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

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，其存储空间的申请往往会得不到满足。

分区存储管理算法练习题

例：采用可变分区方式管理内存时，若内存中按地址顺序依次有五个大小依次为 15k、28k、10k、226k 和 110k 的空闲区。现有五个作业 Ja、Jb、Jc、Jd 和 Je，它们各需要内存 10k、15k、102k、26k 和 180k。

问：

① 如果采用首次适应分配算法，能将这五个作业按 Ja～Je 的次序全部装入内存吗？

② 用什么分配算法可以装入这 5 个作业？

分区存储管理算法练习题

① 按首次适应分配算法，不能将这五个作业按 Ja　Je 的次序全部装入内存。因为内存中前两个空闲分区能依次装入 Ja(10k) 和 Jb(15k)，第 3 个 10KB 的空闲区和刚刚划分出来的两个大小分别为 5KB 和 13KB 的空闲区均无法分配给 Jc(102k) ，第 4 个空闲区可以分 2 次装入作业 Jc(102k) 和 Jd(26k)，最后作业 Je(180k) 无法装入内存。

② 用最佳适应分配算法可装入这 5 个作业。

空闲区	10k	15k	28k	110k	226k
作业	J_a(10k)	J_b(15k)	J_d(26k)	J_c(102k)	J_e(180k)

首次适应算法空闲分区表

分区号	大小	起始地址
1	30K	20K
2	20K	100K
3	5K	160K
4	46K	210K

作业序列

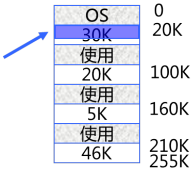
作业A
18K

首次适应算法空闲分区表

分区号	大小	起始地址
1	30K　12K	20K　38K
2	20K	100K
3	5K	160K
4	46K	210K

作业序列

作业A
18K



首次适应算法空闲分区表

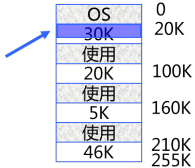
分区号	大小	起始地址
1	30K　12K	20K　38K
2	20K	100K
3	5K	160K
4	46K	210K

最佳适应算法空闲分区表

分区号	大小	起始地址
1	5K	160K
2	20K	100K
3	30K	20K
4	46K	210K

作业序列

作业A
18K



作业序列

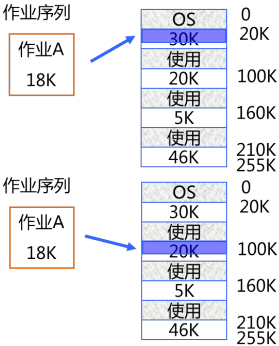
作业A
18K

首次适应算法空闲分区表

分区号	大小	起始地址
1	30K　12K	20K　38K
2	20K	100K
3	5K	160K
4	46K	210K

最佳适应算法空闲分区表

分区号	大小	起始地址
1	5K	160K
2	20K　2K	100K　118K
3	30K	20K
4	46K	210K



首次适应算法空闲分区表

分区号	大小	起始地址
1	30K　12K	20K　38K
2	20K	100K
3	5K	160K
4	46K	210K

最佳适应算法空闲分区表

分区号	大小	起始地址
1	5K	160K
2	20K　2K	100K　118K
3	30K	20K
4	46K	210K

最坏适应算法空闲分区表

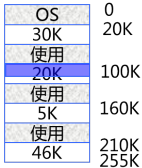
分区号	大小	起始地址
1	46K	210K
2	30K	20K
3	20K	100K
4	5K	160K

作业序列

作业A
18K

作业序列

作业A
18K



作业序列

作业A
18K

首次适应算法空闲分区表

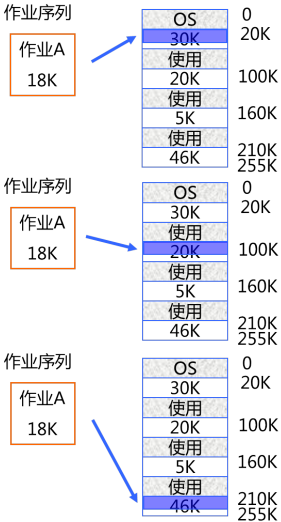
分区号	大小	起始地址
1	30K　12K	20K　38K
2	20K	100K
3	5K	160K
4	46K	210K

最佳适应算法空闲分区表

分区号	大小	起始地址
1	5K	160K
2	20K　2K	100K　118K
3	30K	20K
4	46K	210K

最坏适应算法空闲分区表

分区号	大小	起始地址
1	46K　28K	210K　228K
2	30K	20K
3	20K	100K
4	5K	160K



基于索引搜索的动态分区分配算法

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。

索引搜索动态分区分配算法

- 快速适应算法（Quick Fit）
- 伙伴系统（Buddy system）
- 哈希算法

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。
- 优点是查找效率高，仅需要根据进程的长度，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。
- 缺点是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”。

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 2，4，8，...，2^m 字节。

例：对于 1M 的内存，空闲块的大小最多可以有多少种不同的取值？

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 2，4，8，...，2^m 字节。

例：对于 1M 的内存，空闲块的大小最多可以有多少种不同的取值？　20

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。

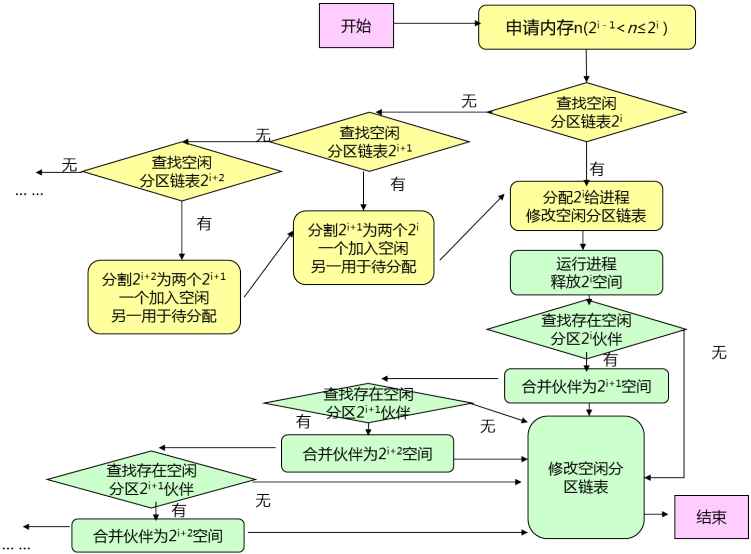
伙伴系统的内存释放

- 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。

伙伴系统

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					


开始
申请内存n($2^{i-1}<n\leq 2^i$)
查找空闲分区链表2^i
无
有
查找空闲分区链表2^{i+1}
无
有
查找空闲分区链表2^{i+2}
无
有
... ...
分配2^i给进程修改空闲分区链表
运行进程释放2^i空间
分割2^{i+2}为两个2^{i+1}一个加入空闲另一用于待分配
分割2^{i+1}为两个2^i一个加入空闲另一用于待分配
查找存在空闲分区2^i伙伴
有
无
合并伙伴为2^{i+1}空间
查找存在空闲分区2^{i+1}伙伴
有
无
合并伙伴为2^{i+2}空间
查找存在空闲分区2^{i+1}伙伴
无
有
... ...
合并伙伴为2^{i+2}空间
修改空闲分区链表
结束

伙伴系统

- 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。

哈希算法

- 哈希函数（Hash），也叫散列函数，是把任意长度的输入通过散列算法，变换成固定长度的输出，该输出就是散列值。
- 哈希既是一种查找技术，也是一种存储技术。可以用哈希函数建立从关键字空间到存储地址空间的映射。
- 若关键字为 k，计算出 Hash 函数值 Hash(k)，把这个值（Hash 地址）存储为一个线性表，称为散列表或哈希表。
- 查找时，对于给定关键字，求哈希值，然后直接在哈希表中取得所查记录。（不必顺序查表，查找速度比较快）

哈希算法

- 哈希函数是一种单向密码体制，即它是一个从明文到密文的不可逆映射，只有加密过程，没有解密过程。
- 哈希函数可用于文件校验和数字签名。
- 哈希函数的构造
  - 哈希函数应是一个压缩映像函数，应具有较大的压缩性以节省存储空间。
  - 哈希函数应具有较好的散列性，以尽量减少冲突（collision）现象的出现。
- 冲突：不同的关键字可能得到相同的哈希值。即 key1≠key2，而 Hash(key1)=Hash(key2)。

哈希算法

- 上述的分类搜索算法和伙伴系统算法中，都是将空闲分区根据分区大小进行分类，对于每一类具有相同大小的空闲分区，单独设立一个空闲分区链表。
- 哈希算法：构造一张以空闲分区大小为关键字的哈希表，该表的每一个表项记录了一个对应的空闲分区链表表头指针。
- 进行空闲分区分配时，根据所需空闲分区大小，通过哈希函数得到在哈希表中的位置，从中得到相应的空闲分区链表的表头指针。

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、分页式存储管理和请求页式存储管理都会出现内部碎片。
- 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。

系统中的碎片

- 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。
- 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。
- 另一种浪费内存的方式是额外开销（Overhead）。Overhead：内存头，记录一些分配信息以便释放内存。

动态可重定位分区分配用到的技术

- 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。
- 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。

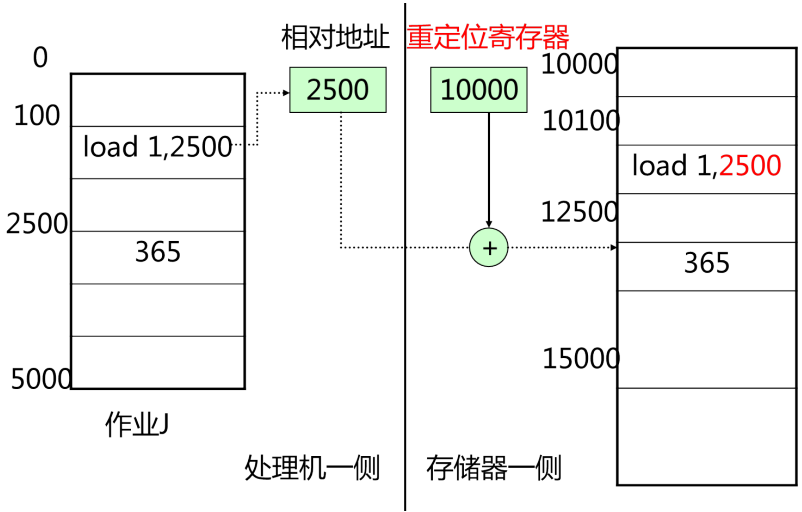
紧凑技术

紧凑技术


操作系统
作业A
20KB
作业B
30KB
作业C
15KB
作业D
25KB


操作系统
作业A
作业B
作业C
作业D
20KB
30KB
15KB
25KB

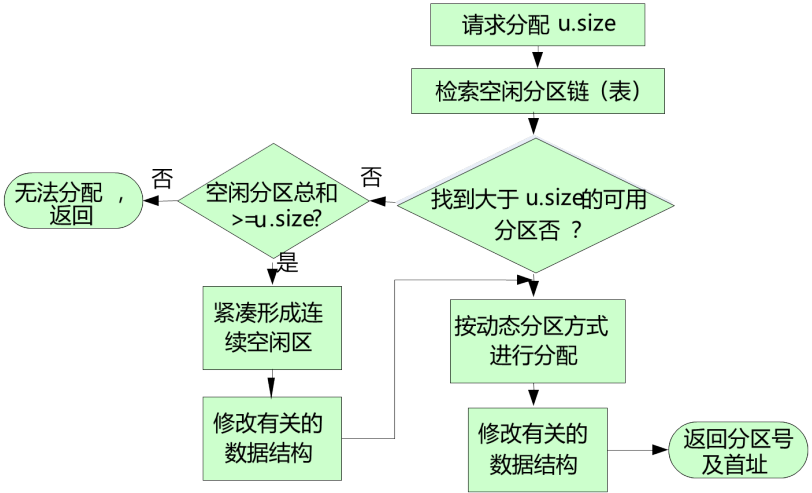
动态重定位的实现


相对地址
重定位寄存器
0
100
2500
5000
load 1,2500
365
2500
10000
+
10000
10100
12500
15000
load 1,2500
365
作业J
处理机一侧
存储器一侧

动态可重定位分区分配算法

- 在分区存储管理方式中，必须把作业装入到一片连续的内存空间。如果系统中有若干个小的分区，其总容量大于要装入的作业，但由于它们不相邻接，也将致使作业不能装入内存。
- 动态可重定位分区分配相当于引入了动态重定位和内存紧凑技术的动态分区分配。

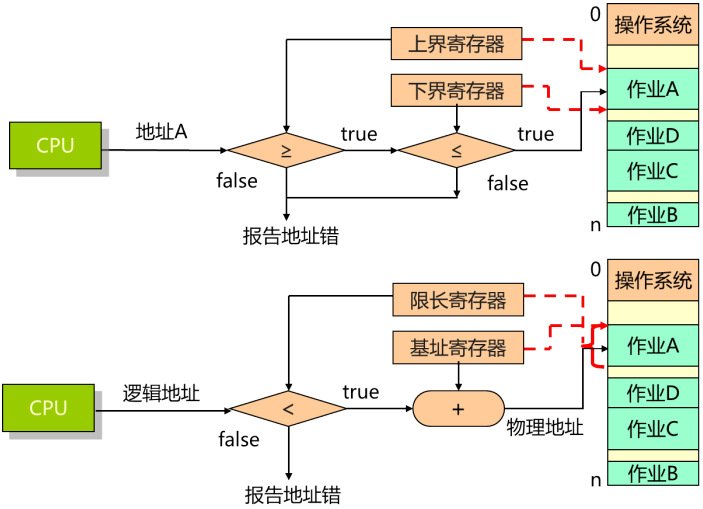
动态可重定位分区分配算法



分区的存储保护

- 存储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。常用的存储保护方法有：
- 界限寄存器方法：①上下界寄存器方法 ② 基址、限长寄存器 (BR,LR) 方法
- 存储保护键方法：给每个存储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。

分区的存储保护

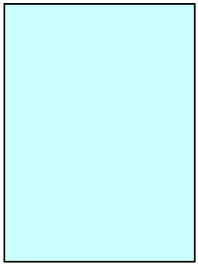

0
操作系统
上界寄存器
下界寄存器
作业A
作业D
作业C
作业B
n
CPU
地址A
≥
true
≤
true
false
false
报告地址错
0
操作系统
限长寄存器
基址寄存器
作业A
作业D
作业C
作业B
n
CPU
逻辑地址
<
true
+
物理地址
false
报告地址错

覆盖与对换

- 覆盖与对换技术是在多道程序环境下用来扩充内存的两种方法。
- 覆盖与对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，对换技术则用在现代 OS 中。

覆盖 (Overlay)

- 覆盖技术主要用在早期的 OS 中（内存 <64KB），可用的存储空间受限，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。

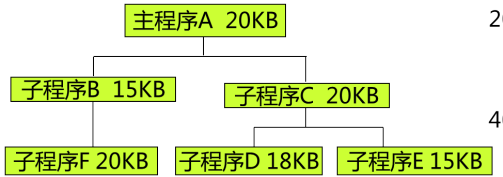


大作业：108KB

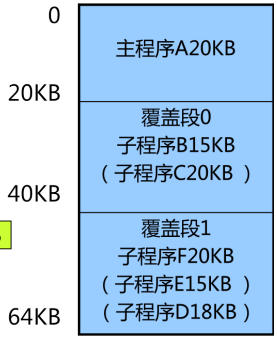
内存：64KB

覆盖 (Overlay)

- 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。



大作业：108 KB
覆盖结构



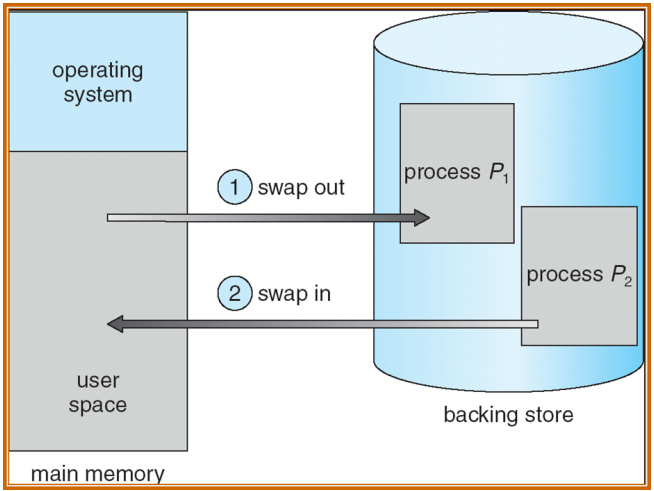
覆盖 (Overlay)

- 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。

对换（Swapping）

- 对换（Swapping）
- 所谓“对换”，是指将暂时不用的某个进程及数据（首先是处于阻塞状态优先级最低的）部分（或全部）从内存移到到外存（备份区或对换区）中去，让出内存空间，同时将某个需要的进程调入到内存中，让其运行。
- 交换技术也是“扩充”内存容量和提高内存利用率的有效措施。
- 交换到外存的进程需要时可以被再次交换回（选择换出时间最久的）内存中继续执行。

对换（Swapping）


operating system
1 swap out
process P_1
process P_2
2 swap in
user space
backing store
main memory

对换（Swapping）

- 对换的类型
  - 整体对换：进程对换，解决内存紧张问题。（中级调度）
  - 部分对换：页面对换/分段对换，提供虚存支持。
- 对换空间的管理
  - 具有对换功能的 OS 中，通常把外存分为文件区和对换区。前者用于存放文件，后者存放从内存换出的进程。
  - 对换区比文件区侧重于对换速度。因此对换区一般采用连续分配。
- 进程的换出与换入
  - 选择换出进程：优先级，进程状态。
  - 选择换入进程：优先级，进程状态，换出时间等。

覆盖与对换技术的区别

- 覆盖可减少一个进程运行所需的空间。对换可让整个进程暂存于外存中，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。对换技术不要求程序员给出程序段之间的覆盖结构。
- 覆盖技术主要在同一个作业或进程中进行。对换主要在作业或进程之间进行。

谢 谢

School of Computer & Information Engineering
Henan University
Kaifeng, Henan Province
475001
China