04 存储器管理

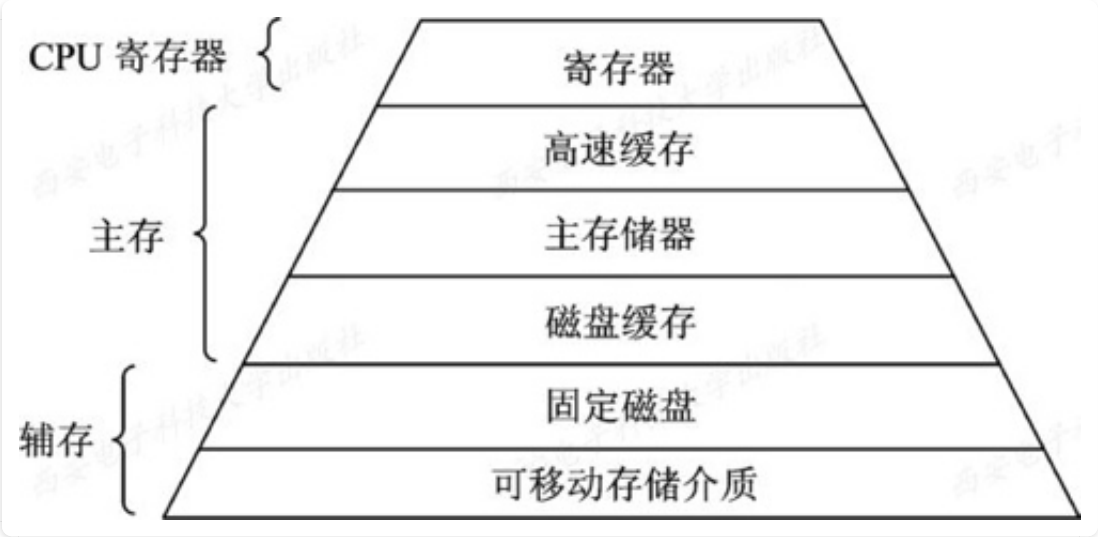
4.1 存储器的结构

采用多层次结构的原因

要求存储器：

1. 速度 必须非常快，能与处理机的速度相匹配
2. 非常大的 容量
3. 价格 便宜

4.1.1 多层结构的存储器系统



层次越高，越靠近CPU，访问速度越快，容量越小，价格越高。

	CPU寄存器和主存	辅存
范畴	存储管理	设备管理
信息保存	掉电后不再存在	长期保存

4.1.2 可执行存储器

> *寄存器和主存又被称为可执行存储器。*

对于存放在内存中的信息，与存放在辅存中的信息相比较而言:

所采用的 `访问机制` 是不同的：进程可以在很少的时钟周期内使用一条 `load` 或 `store` 指令对可执行存储器进行访问。对辅存的访问则需要通过 `I/O设备` 实现。

所需 `耗费的时间` 也是不同的：所耗费的时间一般相差3个数量级甚至更多。

1.寄存器

寄存器具有与处理机相同的速度，故对寄存器的访问速度最快，完全能与CPU协调工作
但价格却十分昂贵
因此容量不可能做得很大。

2.高速缓存

高速缓存介于寄存器和主存储器之间的存储器，主要用于备份主存中较常用的数据，以减少处理机对主存储器的访问次数，这样可大幅度地提高程序执行速度。
高速缓存容量远大于寄存器，而比内存约小两到三个数量级左右，从几十KB到几MB，访问速度快于主存储器。常设置多级。

3.主存储器

主存储器简称内存或主存，是计算机系统中的主要部件，用于保存进程运行时的程序和数据。

4.磁盘缓存

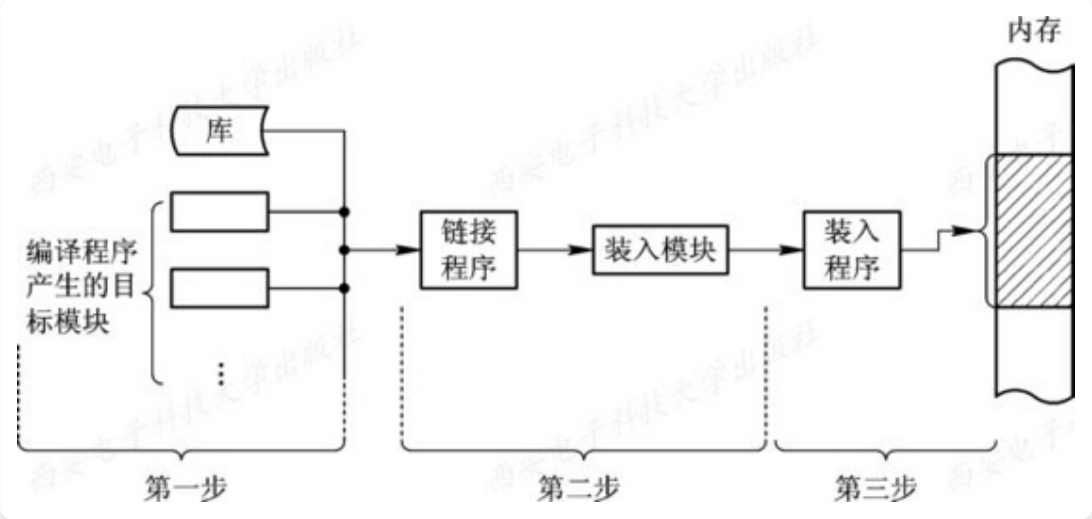
> *由于目前磁盘的I/O速度远低于对主存的访问速度，为了缓和两者之间在速度上的不匹配，而设置了磁盘缓存，主要用于暂时存放频繁使用的一部分磁盘数据和信息，以减少访问磁盘的次数。*

磁盘缓存与高速缓存不同，它本身并不是一种实际存在的存储器，而是利用主存中的部分存储空间暂时存放从磁盘中读出(或写入)的信息。主存也可以看作是辅存的高速缓存，因为，辅存中的数据必须复制到主存方能使用，反之，数据也必须先存在主存中，才能输出到辅存。

4.2 程序的装入和链接

用户程序要在系统中运行，必须先将它装入内存，然后再将其转变为一个可以执行的程序，通常都要经过以下几个步骤：

1. 编译
   由 `编译程序(Compiler)` 对用户源程序进行编译，形成若干个 `目标模块(Object Module)` ；
2. 链接
   由 `链接程序(Linker)` 将编译后形成的一组目标模块以及它们所需要的 `库函数` 链接在一起，形成一个完整的 `装入模块(Load Module)` ；
3. 装入
   由 `装入程序(Loader)` 将装入模块装入内存。



4.2.1 装入的方式

1.绝对装入方式(Absolute Loading Mode)

当计算机系统很小，且仅能运行单道程序时，完全有可能知道程序将驻留在内存的什么位置。此时可以采用绝对装入方式。

用户程序经编译后，将产生绝对地址(即物理地址)的目标代码。

2.可重定位装入方式(Relocation Loading Mode)

> *绝对装入方式只能将目标模块装入到内存中事先指定的位置，这只适用于单道程序环境。*

而在多道程序环境下，编译程序不可能预知经编译后所得到的目标模块应放在

内存的何处。因此，对于用户程序编译所形成的若干个目标模块，它们的起始地址通常都是 `从0开始` 的，程序中的其它地址也都是 `相对` 于起始地址计算的。

3.动态运行时的装入方式(Dynamic Run-time Loading)

可重定位装入方式可将装入模块装入到内存中任何允许的位置，故可用于多道程序环境。
但该方式并不允许程序运行时在内存中移动位置。

4.2.2 链接的方式

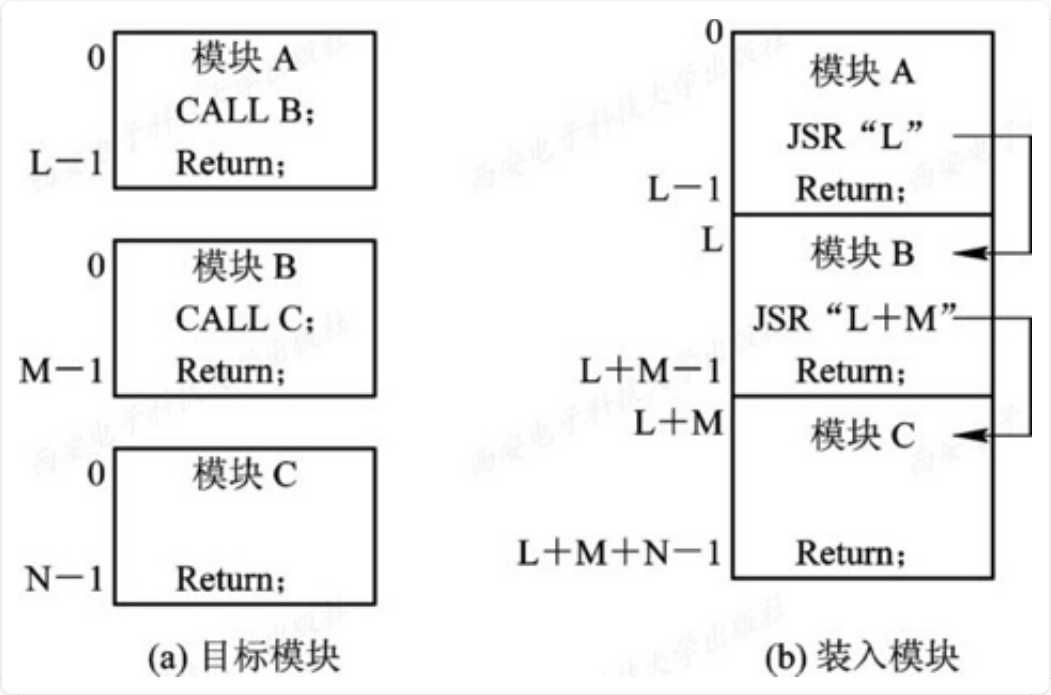
1.静态链接(Static Linking)

> *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配模块，以后不再拆开。*

如下图经过编译后所得到的三个目标模块A、B、C，它们的长度分别为L、M和N。在模块A中有一条语句CALL B，用于调用模块B。在模块B中有一条语句CALL C，用于调用模块C。B和C都属于外部调用符号

在将这几个目标模块装配成一个装入模块时，须解决以下两个问题：

- 对相对地址进行修改
- 变换外部调用符号



2.装入时动态链接(Load-time Dynamic Linking)

> *目标模块在装入内存时，采用边装入边链接的链接方式。*

即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入程序去找出相应的外部目标模块，并将它装入内存，还要按照上图所示的方式修改目标模块中的相对地址。装入时动态链接方式有以下优点：

- 便于修改和更新
- 便于实现对目标模块的共享

3.运行时动态链接(Run-time Dynamic Linking)

> *将对某些模块的链接推迟到程序执行时进行。*

在许多情况下，应用程序在运行时，每次要运行的模块可能是不相同的。但由于事先无法知道本次要运行哪些模块，故只能是将所有可能要运行到的模块全部都装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往会有部分目标模块根本就不运行。比较典型的例子是作为错误处理用的目标模块，如果程序在整个运行过程中都不出现错误，则显然就不会用到该模块。

4.3 连续分配存储管理方式

4.3.1 单一连续分配

> *在单道程序环境下，当时的存储器管理方式是把内存分为系统区和用户区两部分*

分区	系统区	用户区
分配	仅提供给OS使用，它通常是放在内存的低址部分。	仅装有一道用户程序，即整个内存的用户空间由该程序独占。

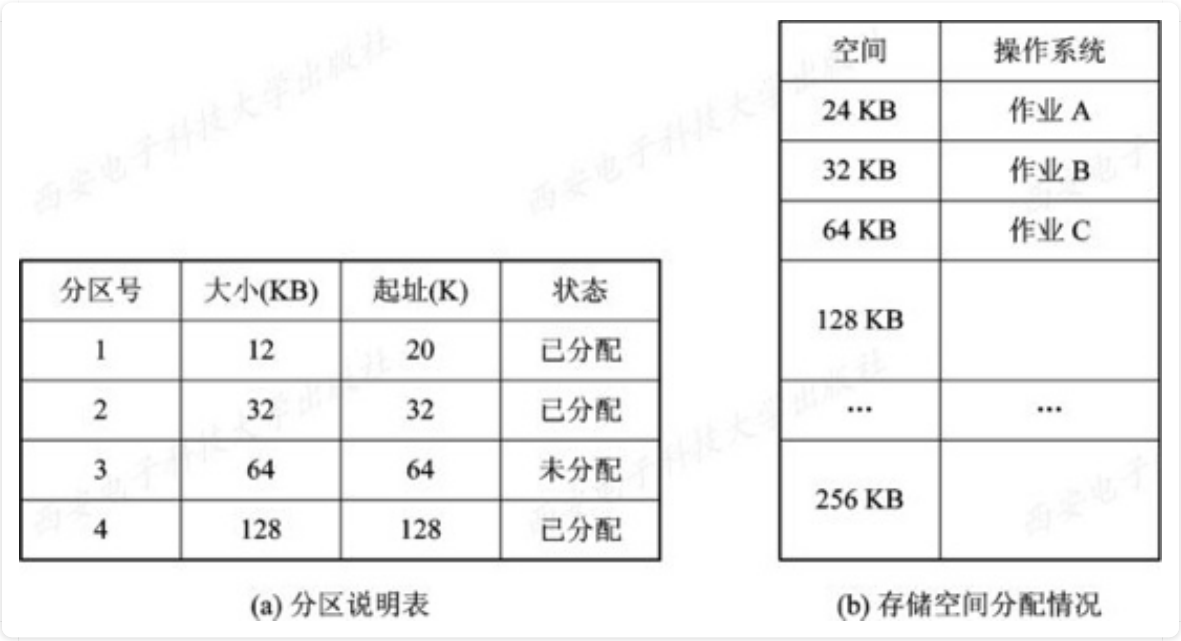
4.3.2 固定分区分配

分区大小相等	分区大小不等

所有内存分区大小相等	多个小分区，适量中分区，少量大分区

> *为了便于内存分配，通常将分区按其大小进行排队，并为之建立一张分区使用表*

其中各表项包括每个分区的 起始地址 、 大小 及 状态(是否已分配)



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

(a) 分区说明表

空间	操作系统
24 KB	作业 A
32 KB	作业 B
64 KB	作业 C
128 KB	
…	…
256 KB	

(b) 存储空间分配情况

4.3.3 动态分区分配

1.动态分区分配数据结构

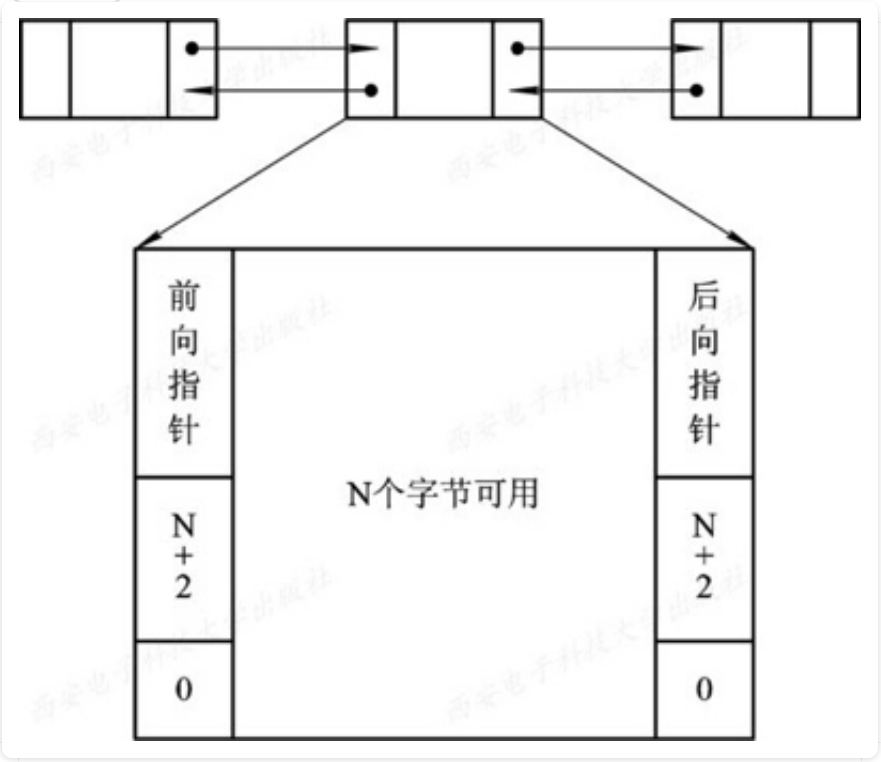
（1）空闲分区表

用于记录每个空闲分区的情况。

分区号	分区大小(KB)	分区始址(K)	状态
1	50	85	空闲
2	32	155	空闲
3	70	275	空闲
4	60	532	空闲
5	…	…	…

（2）空闲分区链

为了实现对空闲分区的分配和链接，在每个分区的起始部分设置一些用于控制分区分配的信息，以及用于链接各分区所用的 前向指针 ，在分区尾部则设置一 后向指针 。通过前、后向链接指针，可将所有的空闲分区链接成一个 双向链 。



2.动态分区的分配算法

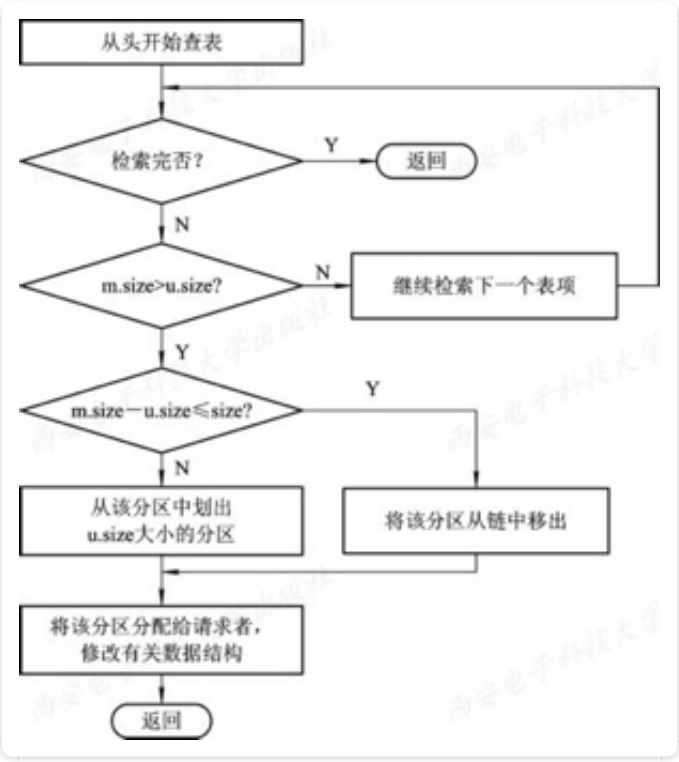
> *为把一个新作业装入内存，须按照一定的分配算法，从空闲分区表或空闲分区链中选出一分区分配给该作业。由于内存分配算法对系*

> *统性能有很大的影响，故人们对它进行了较为广泛而深入的研究，于是产生了许多动态分区分配算法。*

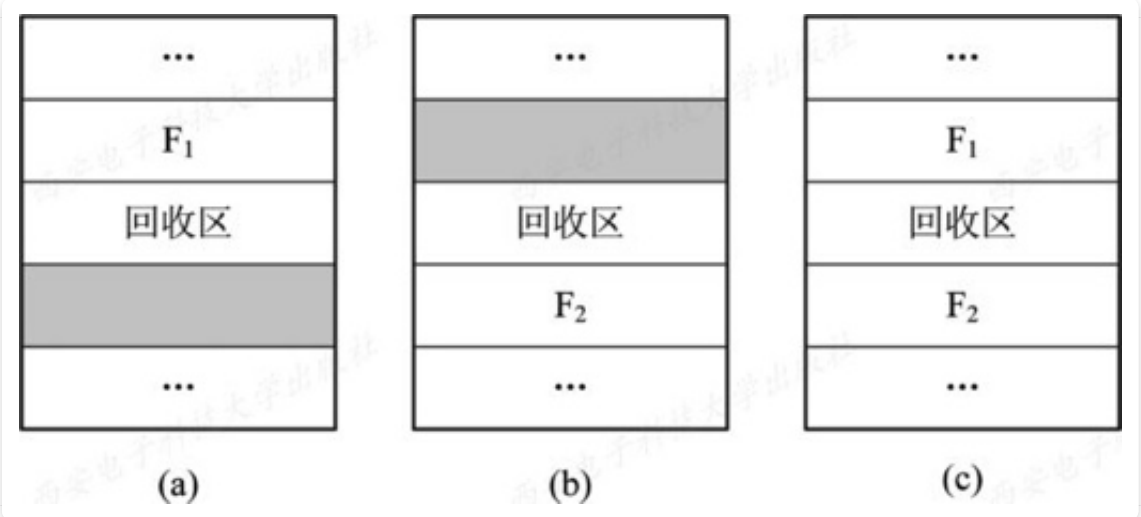
3.动态分区分配的操作

（1）分配内存

系统应利用某种分配算法，从空闲分区链(表)中找到所需大小的分区。设请求的分区大小为u.size，表中每个空闲分区的大小可表示为m.size。

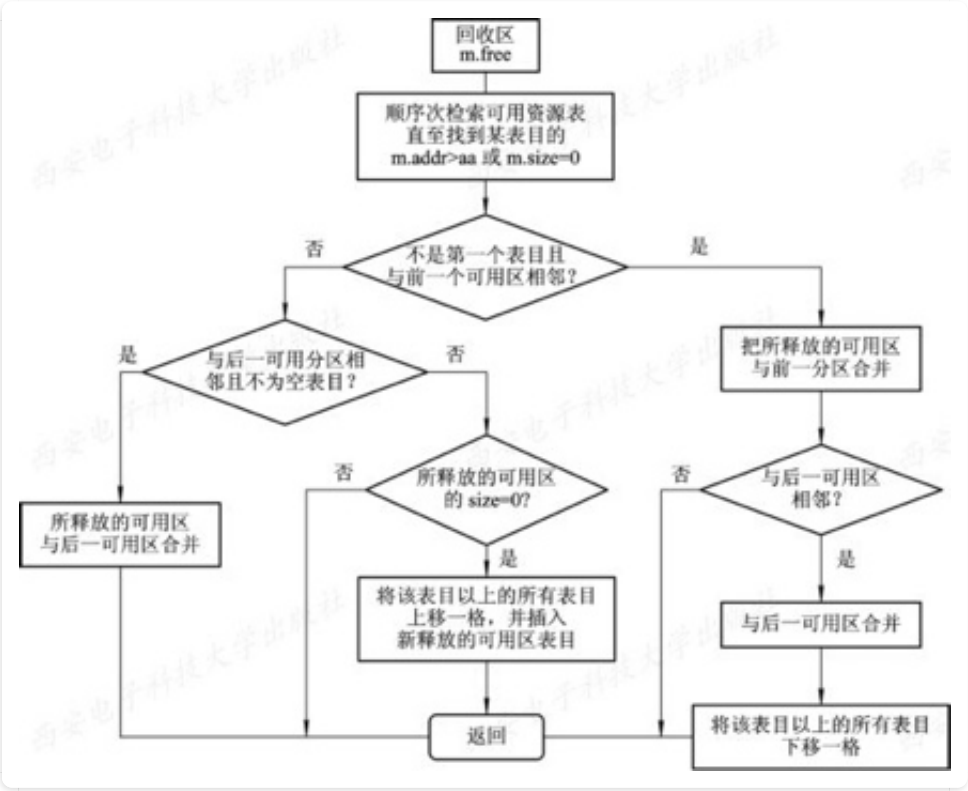


（2）回收内存



1. 回收区与插入点的前一个空闲分区F1相邻接，如上图(a)。此时应将回收区与插入点的前一分区合并，不必为回收分区分配新表项，而只需修改其前一分区F1的大小。
2. 回收分区与插入点的后一空闲分区F2相邻接，如上图(b)。此时也可将两分区合并，形成新的空闲分区，但用回收区的首址作为新空闲区的首址，大小为两者之和。
3. 回收区同时与插入点的前、后两个分区邻接，如上图(c)。此时将三个分区合并，使用F1的表项和F1的首址，取消F2的表项，大小为三者之和。
4. 回收区既不与F1邻接，又不与F2邻接。这时应为回收区单独建立一个新表项，填写回收区的首址和大小，并根据其首址插入到空闲链中的适当位置。

内存回收时的流程：



4.3.4 基于顺序搜索的动态分区分配算法

类型	首次适应(first fit, FF)算法	循环首次适应(next fit，NF)算法	最佳适应(best fit, BF)算法	最坏适应(worst fit，WF)算法
		不再是每	每次为作业	该算法要求将所有

算法	空闲分区链以地址递增的次序链接。在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止。然后再按照作业的大小，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链中。若从链首直至链尾都不能找到一个能满足要求的分区，则表明系统中已没有足够大的内存分配给该进程，内存分配失败，返回。	次都从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找（设置一起始查询指针），直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。	分配内存时，总是把能满足要求、又是最小的空闲分区分配给作业，避免“大材小用”。为了加速寻找，该算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。这样，第一次找到的能满足要求的空闲区必然是最佳的	的空闲分区按其容量以从大到小的顺序形成一空闲分区链。在扫描整个空闲分区表或链表时，总是挑选一个最大的空闲区，从中分割一部分存储空间给作业使用，以至于存储器中缺乏大的空闲分区。
优缺点	优点：优先利用低址部分分区，保留高址部分，为大作业分区创造条件。 缺点：低址部分不断划分，形成许多碎片，增加查找开销。	优点：使内存中空闲分区分布更均匀，减少查找开销。 缺点：缺乏大的空闲分区。	宏观上最佳，仍然会留下许多碎片。	剩下的空闲分区不至于太小，产生碎片的可能性小，对中小作业有利

4.3.5 基于索引搜索的动态分区分配算法

1.快速适应(quick fit)算法

该算法又称为分类搜索法，是将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表，这样系统中存在多个空闲分区链表。同时，在内存中设立一张管理索引表，其中的每一个索引表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针。

2.伙伴系统(buddy system)

该算法规定，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，l≤k≤m)。通常2m是整个可分配内存的大小(也就是最大分区的大小)。假设系统的可利用空间容量为2m 个字，则系统开始运行时，整个内存区是一个大小为2m的空闲分区。在系统运行过程中，由于不断地划分，将会形成若干个不连续的空闲分区，将这些空闲分区按分区的大小进行分类。对于具有相同大小的所有空闲分区，单独设立一个空闲分区双向链表，这样，不同大小的空闲分区形成了k个空闲分区链表。

在伙伴系统中，对于一个大小为2k，地址为x的内存块，其伙伴块的地址则用buddyk(x)表示，其通式为：

3.哈希算法

在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。在为进程分配空间时，需要在一张管理索引表中查找到所需空间大小所对应的表项，从中得到对应的空闲分区链表表头指针，从而通过查找得到一个空闲分区。如果对空闲分区分类较细，则相应索引表的表项也就较多，因此会显著地增加搜索索引表的表项的时间开销。

4.3.6 动态可重定位分区分配

1.紧凑

连续分配方式的一个重要特点是，一个系统或用户程序必须被装入一片连续的内存空间中。当一台计算机运行了一段时间后，它的内存空间将会被分割成许多小的分区，而缺乏大的空闲空间。即使这些分散的许多小分区的容量总和大于要装入的程序，但由于这些分区不相邻接，也无法把该程序装入内存。

紧凑的示意

2.动态重定位

在4.2.1节中所介绍的动态运行时装入的方式中，作业装入内存后的所有地址仍然都是相对(逻辑)地址。而将相对地址转换为绝对(物理)地址的工作被推迟到程序指令要真正执行时进行。为使地址的转换不会影响到指令的执行速度，必须有硬件地址变换机构的支持，即须在系统中增设一个重定位寄存器，用它来存放程序(数据)在内存中的起始地址。程序在执行时，真正访问的内存地址是相对地址与重定位寄存器中的地址相加而形成的。

动态重定位示意图：

3.动态重定位分区分配算法

动态重定位分区分配算法与动态分区分配算法基本上相同，差别仅在于：在这种分配算法中，增加了紧凑的功能。通常，当该算法不能找到一个足够大的空闲分区以满足用户需求时，如果所有的小的空闲分区的容量总和大于用户的要求，这时便须对内存进行“紧凑”，将经“紧凑”后所得到的大空闲分区分配给用户。如果所有的小的空闲分区的容量总和仍小于用户的要求，则返回分配失败信息。

动态分区分配算法流程图：

4.4 对换(Swapping)

对换技术也称为交换技术，最早用于麻省理工学院的单用户分时系统CTSS中。由于当时计算机的内存都非常小，为了使该系统能分时运行多个用户程序而引入了对换技术。系统把所有的用户作业存放在磁盘上，每次只能调入一个作业进入内存，当该作业的一个时间片用完时，将它调至外存的后备队列上等待，再从后备队列上将另一个作业调入内存。这就是最早出现的分时系统中所

用的对换技术。现在已经很少使用。

4.4.1 多道程序环境下的对换技术

1.对换的引入

在多道程序环境下，一方面，在内存中的某些进程由于某事件尚未发生而被阻塞运行，但它却占用了大量的内存空间，甚至有时可能出现在内存中所有进程都被阻塞，而无可运行之进程，迫使CPU停止下来等待的情况；另一方面，却又有着许多作业，因内存空间不足，一直驻留在外存上，而不能进入内存运行。显然这对系统资源是一种严重的浪费，且使系统吞吐量下降。

2.对换的类型

在每次对换时，都是将一定数量的程序或数据换入或换出内存。根据每次对换时所对换的数量，可将对换分为如下两类：

- 整体对换
- 页面(分段)对换

4.4.2 对换空间的管理

1.对换空间管理的主要目标

在具有对换功能的OS中，通常把磁盘空间分为文件区和对换区两部分。

1. 对文件区管理的主要目标
2. 对对换空间管理的主要目标

2.对换区空闲盘块管理中的数据结构

为了实现对对换区中的空闲盘块的管理，在系统中应配置相应的数据结构，用于记录外存对换区中的空闲盘块的使用情况。其数据结构的形式与内存在动态分区分配方式中所用数据结构相似，即同样可以用空闲分区表或空闲分区链。在空闲分区表的每个表目中，应包含两项：对换区的首址及其大小，分别用盘块号和盘块数表示。

3.对换空间的分配与回收

由于对换分区的分配采用的是连续分配方式，因而对换空间的分配与回收与动

态分区方式时的内存分配与回收方法雷同。其分配算法可以是首次适应算法、循环首次适应算法或最佳适应算法等。具体的分配操作也与图4–8中内存的分配过程相同。

4.4.3 进程的换出与换入

1.进程的换出

对换进程在实现进程换出时，是将内存中的某些进程调出至对换区，以便腾出内存空间。换出过程可分为以下两步：

1. 选择被换出的进程
2. 进程换出过程

2.进程的换入

对换进程将定时执行换入操作，它首先查看PCB集合中所有进程的状态，从中找出“就绪”状态但已换出的进程。当有许多这样的进程时，它将选择其中已换出到磁盘上时间最久(必须大于规定时间，如2 s)的进程作为换入进程，为它申请内存。如果申请成功，可直接将进程从外存调入内存；如果失败，则需先将内存中的某些进程换出，腾出足够的内存空间后，再将进程调入。

4.5 分页存储管理方式

- 分页存储管理方式
- 分段存储管理方式
- 段页式存储管理方式

4.5.1 分页存储管理的基本方法

1.页面和物理块

- 页面
- 页面大小

2.地址结构

分页地址中的地址结构如下：

对某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：

3.页表

在分页系统中，允许将进程的各个页离散地存储在内存的任一物理块中，为保证进程仍然能够正确地运行，即能在内存中找到每个页面所对应的物理块，系统又为每个进程建立了一张页面映像表，简称页表。

页表的作用：

4.5.2 地址变换机构

1.基本的地址变换机构

进程在运行期间，需要对程序和数据的地址进行变换，即将用户地址空间中的逻辑地址变换为内存空间中的物理地址，由于它执行的频率非常高，每条指令的地址都需要进行变换，因此需要采用硬件来实现。页表功能是由一组专门的寄存器来实现的。一个页表项用一个寄存器。

分页系统的地址变换机构：

2.具有快表的地址变换机构

由于页表是存放在内存中的，这使CPU在每存取一个数据时，都要两次访问内存。第一次是访问内存中的页表，从中找到指定页的物理块号，再将块号与页内偏移量W拼接，以形成物理地址。第二次访问内存时，才是从第一次所得地址中获得所需数据(或向此地址中写入数据)。因此，采用这种方式将使计算机

的处理速度降低近1/2。可见，以此高昂代价来换取存储器空间利用率的提高，是得不偿失的。

具有快表的地址变换机构

4.5.3 访问内存的有效时间

从进程发出指定逻辑地址的访问请求，经过地址变换，到在内存中找到对应的实际物理地址单元并取出数据，所需要花费的总时间，称为内存的有效访问时间(Effective Access Time，EAT)。假设访问一次内存的时间为t，在基本分页存储管理方式中，有效访问时间分为第一次访问内存时间(即查找页表对应的页表项所耗费的时间t)与第二次访问内存时间(即将页表项中的物理块号与页内地址拼接成实际物理地址所耗费的时间t)之和：

$$EAT = t + t = 2t$$

在引入快表的分页存储管理方式中，通过快表查询，可以直接得到逻辑页所对应的物理块号，由此拼接形成实际物理地址，减少了一次内存访问，缩短了进程访问内存的有效时间。但是，由于快表的容量限制，不可能将一个进程的整个页表全部装入快表，所以在快表中查找到所需表项存在着命中率的问题。所谓命中率，是指使用快表并在其中成功查找到所需页面的表项的比率。这样，在引入快表的分页存储管理方式中，有效访问时间的计算公式即为：

$$EAT=a\times\lambda+(t+\lambda)(1—a)+t=2t+\lambda—t\times a$$

上式中，λ表示查找快表所需要的时间，a表示命中率，t表示访问一次内存所需要的时间。

可见，引入快表后的内存有效访问时间分为查找到逻辑页对应的页表项的平均时间$a \times \lambda + (t + \lambda)(1 - a)$，以及对应实际物理地址的内存访问时间t。假设对快表的访问时间λ为20 ns(纳秒)，对内存的访问时间t为100 ns，则下表中列出了不同的命中率a与有效访问时间的关系：

4.5.4 两级和多级页表

1.两级页表(Two-Level Page Table)

针对难于找到大的连续的内存空间来存放页表的问题，可利用将页表进行分页的方法，使每个页面的大小与内存物理块的大小相同，并为它们进行编号，即

依次为0# 页、1# 页，…，n# 页，然后离散地将各个页面分别存放在不同的物理块中。同样，也要为离散分配的页表再建立一张页表，称为外层页表(Outer Page Table)，在每个页表项中记录了页表页面的物理块号。

两级页表结构：

为了方便实现地址变换，在地址变换机构中，同样需要增设一个外层页表寄存器，用于存放外层页表的始址，并利用逻辑地址中的外层页号作为外层页表的索引，从中找到指定页表分页的始址，再利用P2作为指定页表分页的索引，找到指定的页表项，其中即含有该页在内存的物理块号，用该块号P和页内地址d即可构成访问的内存物理地址。

具有两级页表的地址变换机构

2.多级页表

对于32位的机器，采用两级页表结构是合适的，但对于64位的机器，采用两级页表是否仍然合适，须做以下简单分析。如果页面大小仍采用4 KB即212 B，那么还剩下52位，假定仍按物理块的大小(212位)来划分页表，则将余下的42位用于外层页号。此时在外层页表中可能有4096 G个页表项，要占用16 384 GB的连续内存空间。

4.5.5 反置页表(Inverted Page Table)

1.反置页表的引入

在分页系统中，为每个进程配置了一张页表，进程逻辑地址空间中的每一页，在页表中都对应有一个页表项。在现代计算机系统中，通常允许一个进程的逻辑地址空间非常大，因此就需要有许多的页表项，而因此也会占用大量的内存空间。

2.地址变换

在利用反置页表进行地址变换时，是根据进程标识符和页号，去检索反置页

表。如果检索到与之匹配的页表项，则该页表项(中)的序号i便是该页所在的物理块号，可用该块号与页内地址一起构成物理地址送内存地址寄存器。若检索了整个反置页表仍未找到匹配的页表项，则表明此页尚未装入内存。对于不具有请求调页功能的存储器管理系统，此时则表示地址出错。对于具有请求调页功能的存储器管理系统，此时应产生请求调页中断，系统将把此页调入内存。

4.6 分段存储管理方式

存储管理方式随着OS的发展也在不断地发展。当OS由单道向多道发展时，存储管理方式便由单一连续分配发展为固定分区分配。

4.6.1 分段存储管理方式的引入

1.方便编程

通常，用户把自己的作业按照逻辑关系划分为若干个段，每个段都从0开始编址，并有自己的名字和长度。因此，程序员们都迫切地需要访问的逻辑地址是由段名(段号)和段内偏移量(段内地址)决定的，这不仅可以方便程序员编程，也可使程序非常直观，更具可读性。例如，下述的两条指令便使用段名和段内地址：

LOAD 1，[A]|〈D〉；
STORE 1，[B]|〈C〉；

2.信息共享

在实现对程序和数据的共享时，是以信息的逻辑单位为基础的。比如，为了共享某个过程、函数或文件。分页系统中的“页”只是存放信息的物理单位(块)，并无完整的逻辑意义，这样，一个可被共享的过程往往可能需要占用数十个页面，这为实现共享增加了困难。

3.信息保护

信息保护同样是以信息的逻辑单位为基础的，而且经常是以一个过程、函数或文件为基本单位进行保护的。

4.动态增长

在实际应用中，往往存在着一些段，尤其是数据段，在它们的使用过程中，由于数据量的不断增加，而使数据段动态增长，相应地它所需要的存储空间也会

动态增加。然而，对于数据段究竟会增长到多大，事先又很难确切地知道。对此，很难采取预先多分配的方法进行解决。

5.动态链接

在4.2.2节中我们已对运行时动态链接做了介绍。为了提高内存的利用率，系统只将真正要运行的目标程序装入内存，也就是说，动态链接在作业运行之前，并不是把所有的目标程序段都链接起来。当程序要运行时，首先将主程序和它立即需要用到的目标程序装入内存，即启动运行。而在程序运行过程中，当需要调用某个目标程序时，才将该段(目标程序)调入内存并进行链接。

4.6.2 分段系统的基本原理

1.分段

在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段MAIN、子程序段X、数据段D及栈段S等，如图4–19所示。

分段地址中的地址具有如下结构：

2.段表

在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在分段式存储管理系统中，则是为每个分段分配一个连续的分区。进程中的各个段，可以离散地装入内存中不同的分区中。为保证程序能正常运行，就必须能从物理内存中找出每个逻辑段所对应的位置。

图4–19 利用段表实现地址映射

3.地址变换机构

为了实现进程从逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器，用于存放段表始址和段表长度TL。在进行地址变换时，系统将逻辑地址中的段号与段表长度TL进行比较。若S>TL，表示段号太大，是访问越界，于是产生越界中断信号。若未越界，则根据段表的始址和该段的段号，计算出该段

对应段表项的位置，从中读出该段在内存的起始地址。然后，再检查段内地址d是否超过该段的段长SL。若超过，即d>SL，同样发出越界中断信号。若未越界，则将该段的基址d与段内地址相加，即可得到要访问的内存物理地址。

分段系统的地址变换过程:

4.分页和分段的主要区别

- 页是信息的物理单位
- 页的大小固定且由系统决定
- 分页的用户程序地址空间是一维的

4.6.3 信息共享

1.分页系统中对程序和数据的共享

在分页系统中，虽然也能实现对程序和数据的共享，但远不如分段系统来得方便。我们通过一个例子来说明这个问题。

分页系统中共享editor的示意图:

2.分段系统中程序和数据的共享

在分段系统中，由于是以段为基本单位的，不管该段有多大，我们都只需为该段设置一个段表项，因此使实现共享变得非常容易。我们仍以共享editor为例，此时只需在(每个)进程1和进程2的段表中，为文本编辑程序设置一个段表项，让段表项中的基址(80)指向editor程序在内存的起始地址。

分段系统中共享editor的示意图：

4.6.4 段页式存储管理方式

1.基本原理

段页式系统的基本原理是分段和分页原理的结合，即先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。图4–23(a)示出了一个作业地址空间的结构。该作业有三个段：主程序段、子程序段和数据段；页面大小为 4 KB。在段页式系统中，其地址结构由段号、段内页号及页内地址三部分所组成

作业地址空间和地址结构：

在段页式系统中，为了实现从逻辑地址到物理地址的变换，系统中需要同时配置段表和页表。段表的内容与分段系统略有不同，它不再是内存始址和段长，而是页表始址和页表长度。图4–24示出了利用段表和页表进行从用户地址空间到物理(内存)空间的映射。

利用段表和页表实现地址映射：

2.地址变换过程

在段页式系统中，为了便于实现地址变换，须配置一个段表寄存器，其中存放段表始址和段长TL。进行地址变换时，首先利用段号S，将它与段长TL进行比较。若S < TL，表示未越界，于是利用段表始址和段号来求出该段所对应的段表项在段表中的位置，从中得到该段的页表始址，并利用逻辑地址中的段内页号P来获得对应页的页表项位置，从中读出该页所在的物理块号b，再利用块号b和页内地址来构成物理地址。

段页式系统中的地址变换机构：