操作系统 A 期末

1. 基本存储管理

KEY WORDS：地址重定位，单一连续区，虚拟内存，工作集，驻留集，逻辑地址，物理地址，固定分区，虚拟存储空间，工作集，存储保护，可变分区，虚拟地址，物理地址，置换策略，存储共享，页式，段式，页表/页表项，清除策略，覆盖技术，TLB，页缓冲，交换技术，段页式，页错误，加载控制

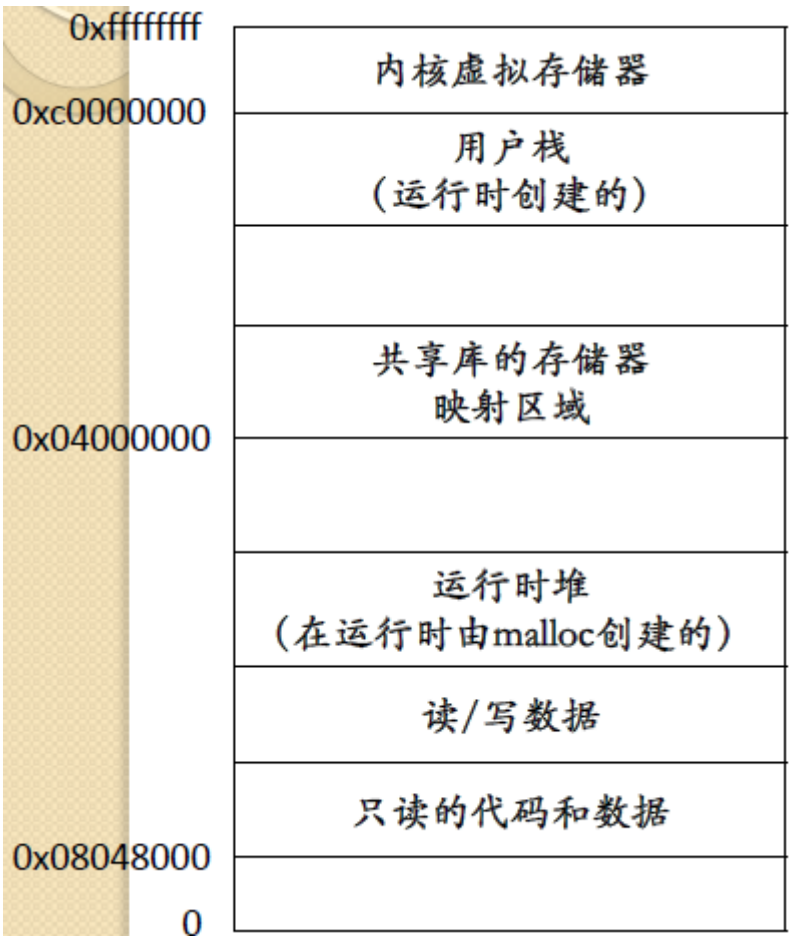
基本概念

内存管理的工作：支持地址重定位，支持地址保护。

内存管理的目标：分配进程的内存（地址空间），向内存加载内容（映射进程地址空间到物理内存），实现存储保护（越界和权限的检查），管理共享内存，最小化存储访问时间。

需要解决：单个进程地址空间的连续性-多个进程地址空间的离散性；单个进程地址空间的驻留性-多个进程地址空间的交换性；单个进程地址空间使用的一次性-相同代码不同进程地址空间使用的多次性。

进程地址空间



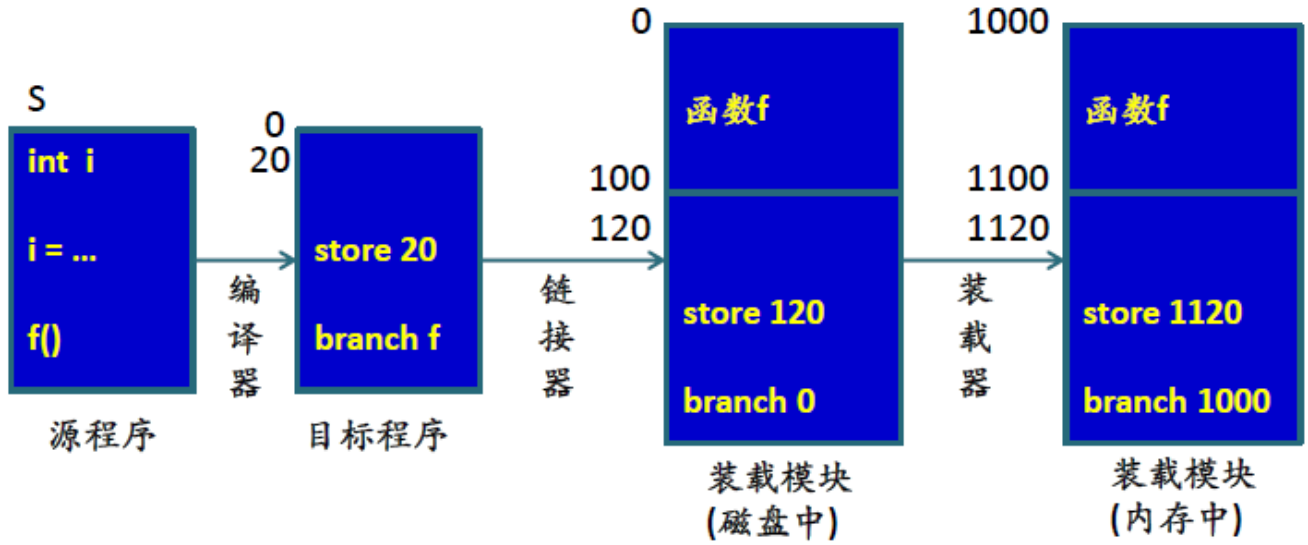
程序执行的准备过程：源程序i->编译->目标文件i->链接->装载模块（磁盘中）->装载->装载模块（内存中）->运行。装载模块也可以使系统库。

地址重定位：讲用户程序中的逻辑地址装换为运行时可以由机器直接寻址的物理地址的过程，目的是保证CPU执行指令时正确地访问内存单元。

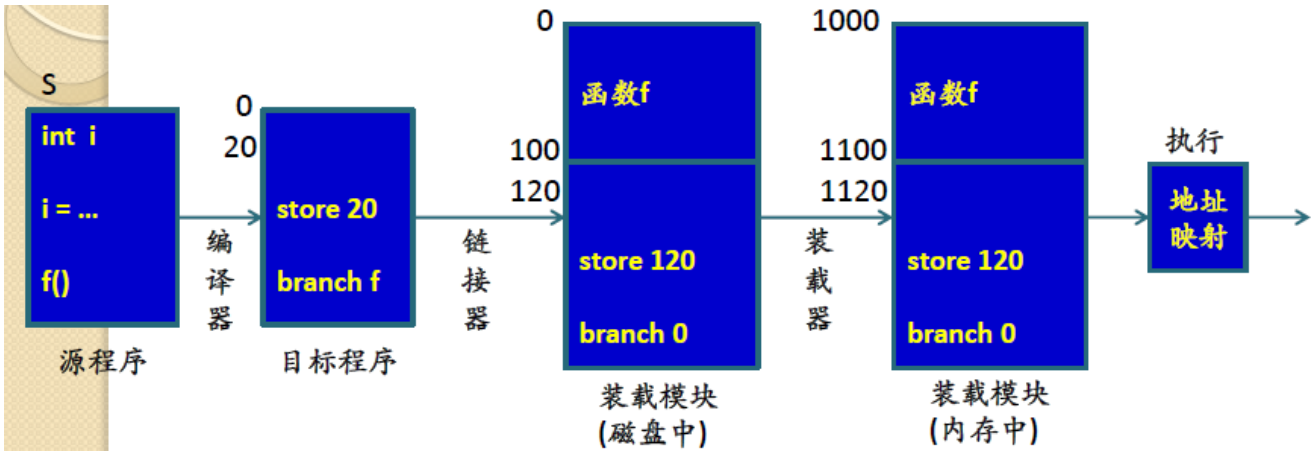
逻辑地址/相对地址/虚拟地址：用户程序经过编译、汇编后形成目标代码，采取相对地址形式，首地址为0，其余指令中的地址相对于0首地址编址。不可以使用逻辑地址在内存中寻址。

物理地址/绝对地址/实地址：内存中存储单元的地址，可以直接进行寻址。

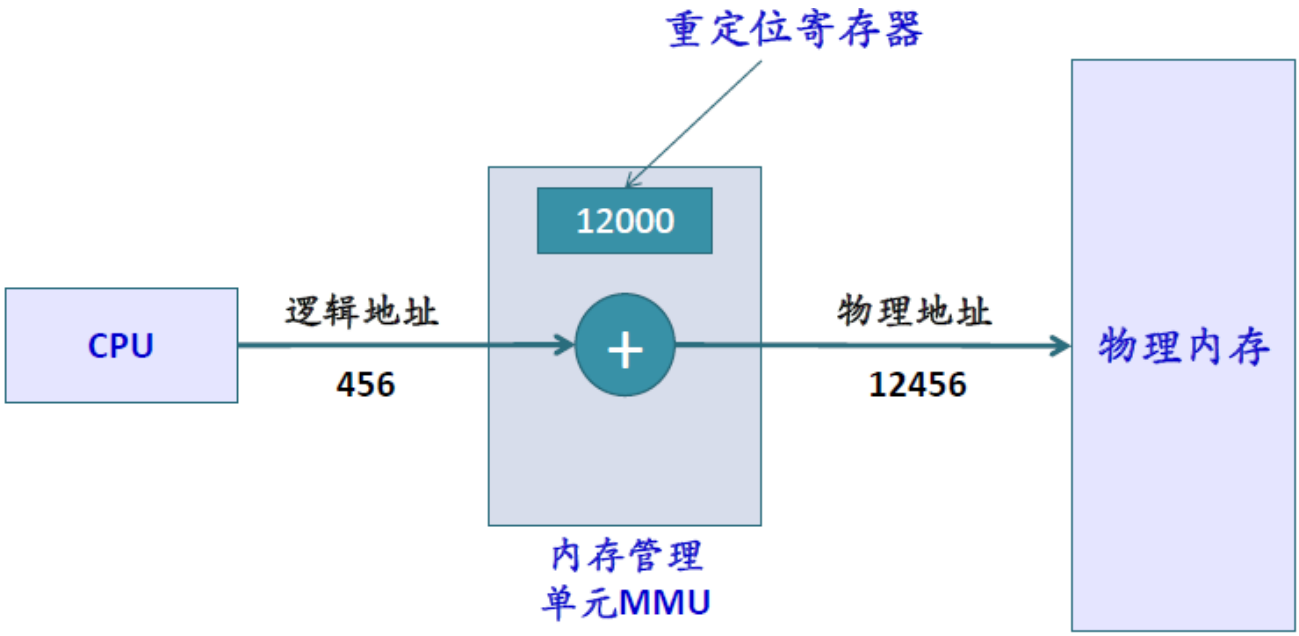
静态地址重定位：用户程序加载到内存，一次性地将逻辑地址转换成物理地址，可以由装载器软件完成。



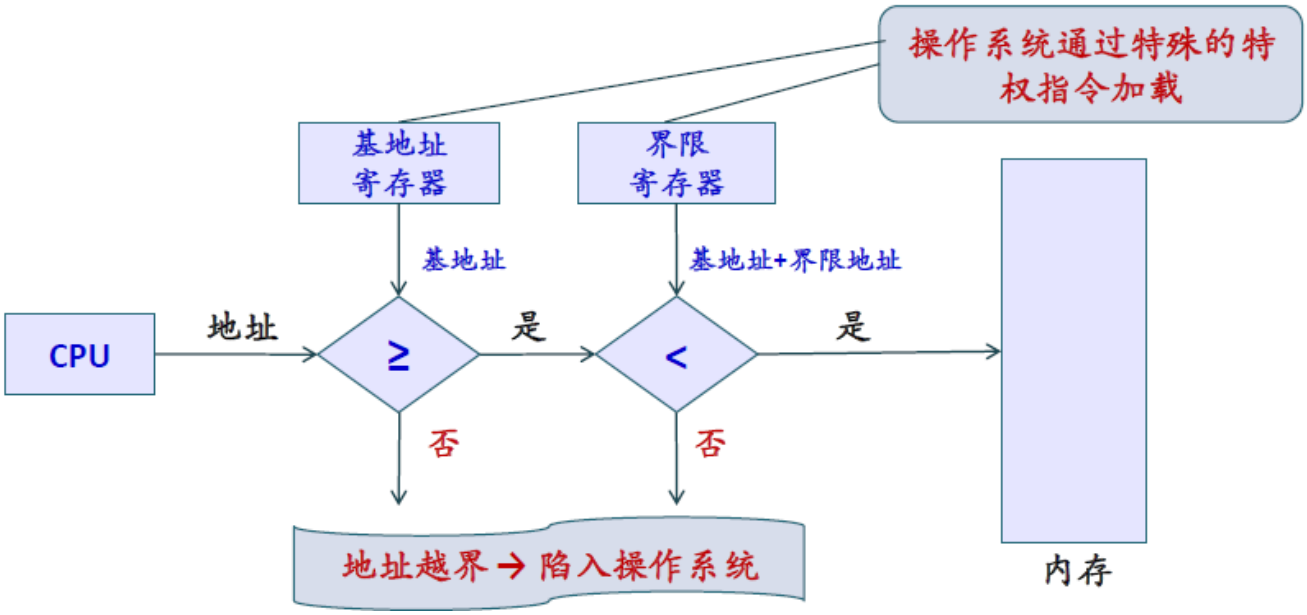
动态地址重定位：进程执行过程中逐条指令执行时完成地址变换。可以要求硬件支持，也可以软件制作映射器。



MMU动态重定位实现：添加重定位寄存器。



地址保护：确保每个进程由独立的地址空间，确保每个进程都只可访问其合法地址。



物理内存管理方案

空闲内存管理的**数据结构**：位图，空闲区表+已分配区表（记录空闲区/已分配区的起始地址、长度、标志等），空闲块链表。采用等长划分管理（位图，空闲块链表）或不等长划分管理（空闲区表，已分配区表）。

内存分配算法：首次适配（first fit，空闲区表中找到第一个满足进程要求的空闲区），下次适配（next fit，从上次找到的空闲区进行查找），最佳适配（best fit，查找整个空闲区表，找到满足要求的最小空闲区），最差适配（worst fit，总是分配满足进程要求的最大空闲区）。找到空闲区后，将其划分为两部分，一部分供进程使用，另一部分形成新的空闲区。

内存回收算法：某一块归还后，前后空闲空间合并，修改内存空闲区表。需要考虑四种情况——上相邻，下相邻，上下相邻，上下不相邻。

伙伴系统：Linux低层内存管理所采用的算法，是一种特殊的分离适配算法。将内存按2的幂次划分，组成若干空闲块链表，查找该链表找到满足进程需要的最佳匹配块。

初始化：内存视为一整个块，大小为2^U

进程申请空间大小为s

若满足2^(U-1)<s<=2^U，则分配整个块

否则，将块划分为两个大小相同的伙伴块，大小都为2^(U-1)，一枝花分下去直到产生不小于s的最小块

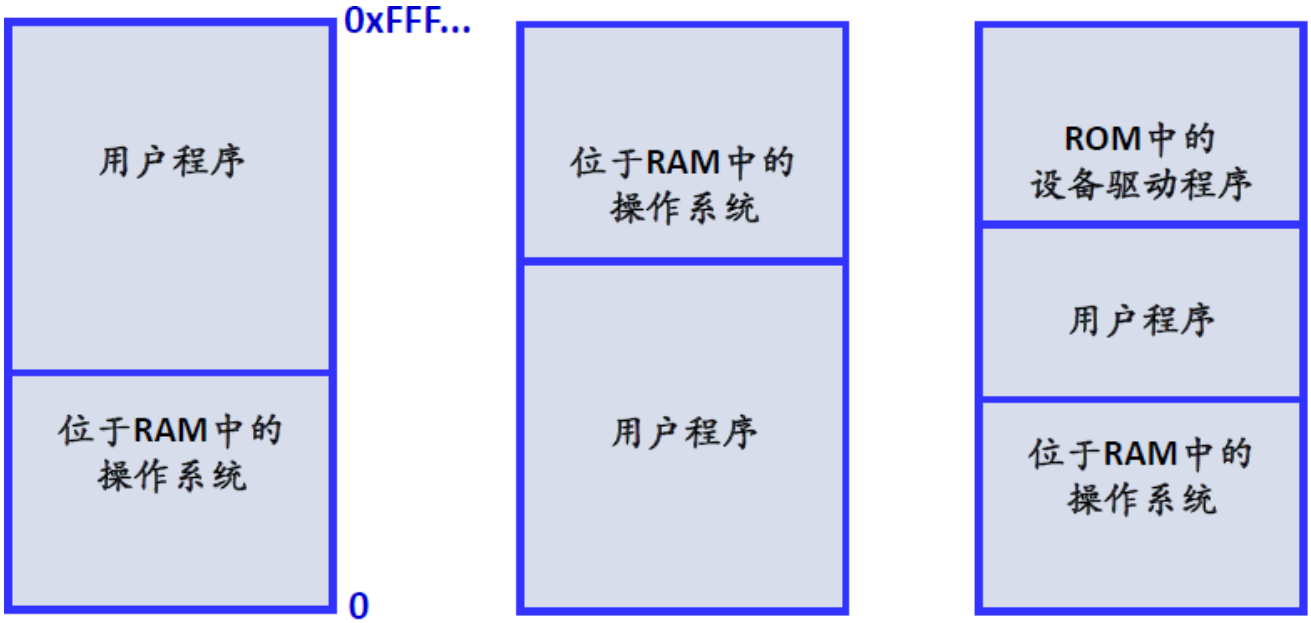
进程释放空间

若被释放的块的伙伴空闲，则递归地进行合并

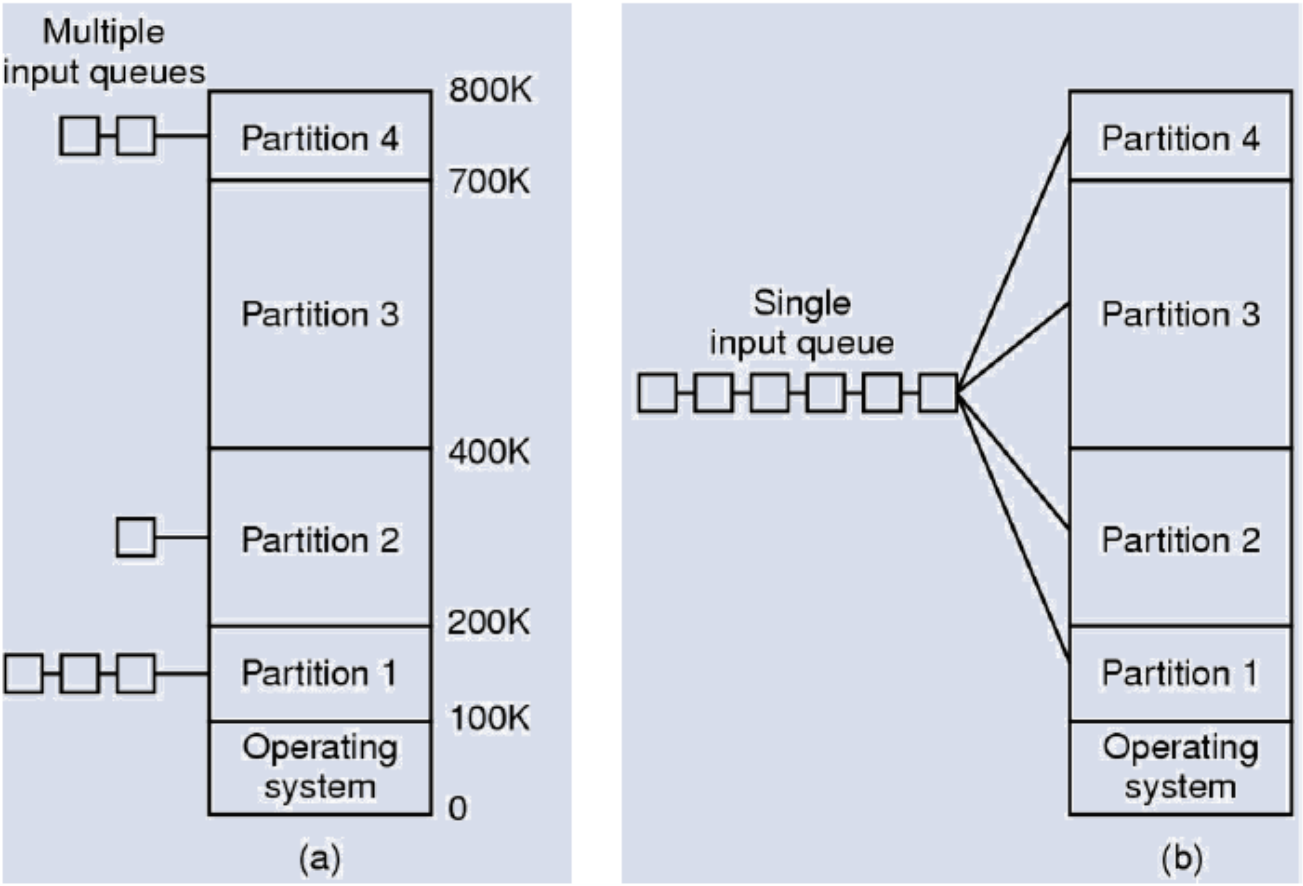
否则直接释放该块

内存管理的基本方案

单一连续区：一段时间只有一个进程在内存，简单但内存利用率很低。



固定分区：将内存空间分割为若干个大小相同或不同的区域，每个区域为一个分区，每个分区大小固定不变，每个分区一次至多只能装入一个进程。



可变分区：根据进程的需要，将内存空闲空间分割出一个分区，分配给该进程，剩余部分成为新的空闲区。

碎片：很小的，不易于利用的空闲区，导致内存利用率下降。

外部碎片：实际的物理空闲充裕，但由于不连续而使得无法被利用。

解决方案：紧缩技术/压缩技术/紧致技术/搬家技术，在内存内移动程序，将小的空闲区合并为大的空闲区，需要考虑移动的系统开销和移动的时机。

页式存储管理：将用户程序地址空间划分为大小相等的页，内存空间也按页的大小划分为大小相等的区域，称作内存块/物理页面/页框/页帧，按页为单位，按进程需要的页数进行分配。在地址空间里逻辑相邻的页，在物理内存上并不一定相邻。

典型页面大小：4K或4M

逻辑地址：由页内偏移（低位）和页号（高位）拼接而成。32位系统，4K页面大小的页式存储，低12位为偏移量，高20位为页号。

页式存储管理的数据结构：**页表**，进程地址空间与内存地址映射/分配关系的记录表。每个进程对应一个页表，页表内记录该进程的所有页面的分配情况。

页表项：记录逻辑页号和页框号的对应关系。

页表的起始位置：存放在PCB中，在进程切换时将其加载到CR寄存器中（x86体系结构）。

地址转换：需要由硬件提供支持。CPU取逻辑地址，自动划分出页号和偏移量，用页号查找页表得到页框号，将页框号与偏移量拼接得到物理地址。

内部碎片：由于页面未被填充满而使之无法被利用。

段式存储管理：用户程序地址空间按照自身的逻辑关系划分为若干个程序段，每个程序段有一个段名；内存空间被动态地划分为若干长度不同的区域，成为物理段，每个物理段起始地址和长度确定。分配时以段为单位进行分配，每个段在内存中占据连续空间，但各段之间允许断开。

段式存储管理的数据结构：**段表**，记录段号，段首地址和段长度，每个进程一个段表，存放在内存中。

段表项：记录段号，段首地址，段长度的对应关系。

段表的起始位置：存放在PCB中，进程切换时加载到CR寄存器（x86）。

地址转换：CPU取到逻辑地址，用段号查找段表，得到该段的起始地址，再与段内偏移地址计算出物理地址。

段页式存储管理：结合段式和页式，尽量克服二者缺点。

段式划分：对于用户而言，按段式的逻辑关系进行划分

页式划分：对于操作系统而言，按页式划分每一段。最终的内存划分和分配，也按页式存储管理，以页为单位进行分配。

$$逻辑地址 = 段号 + 段内地址$$
$$段内地址 = 页号 + 页内地址$$

段页式存储管理的数据结构：**段表**，记录每一段的页表起始地址和页表长度；**页表**，记录逻辑页号与内存块号的对应关系。每段一个页表，一个程序可能有多个页表。

空闲区管理：同页式管理

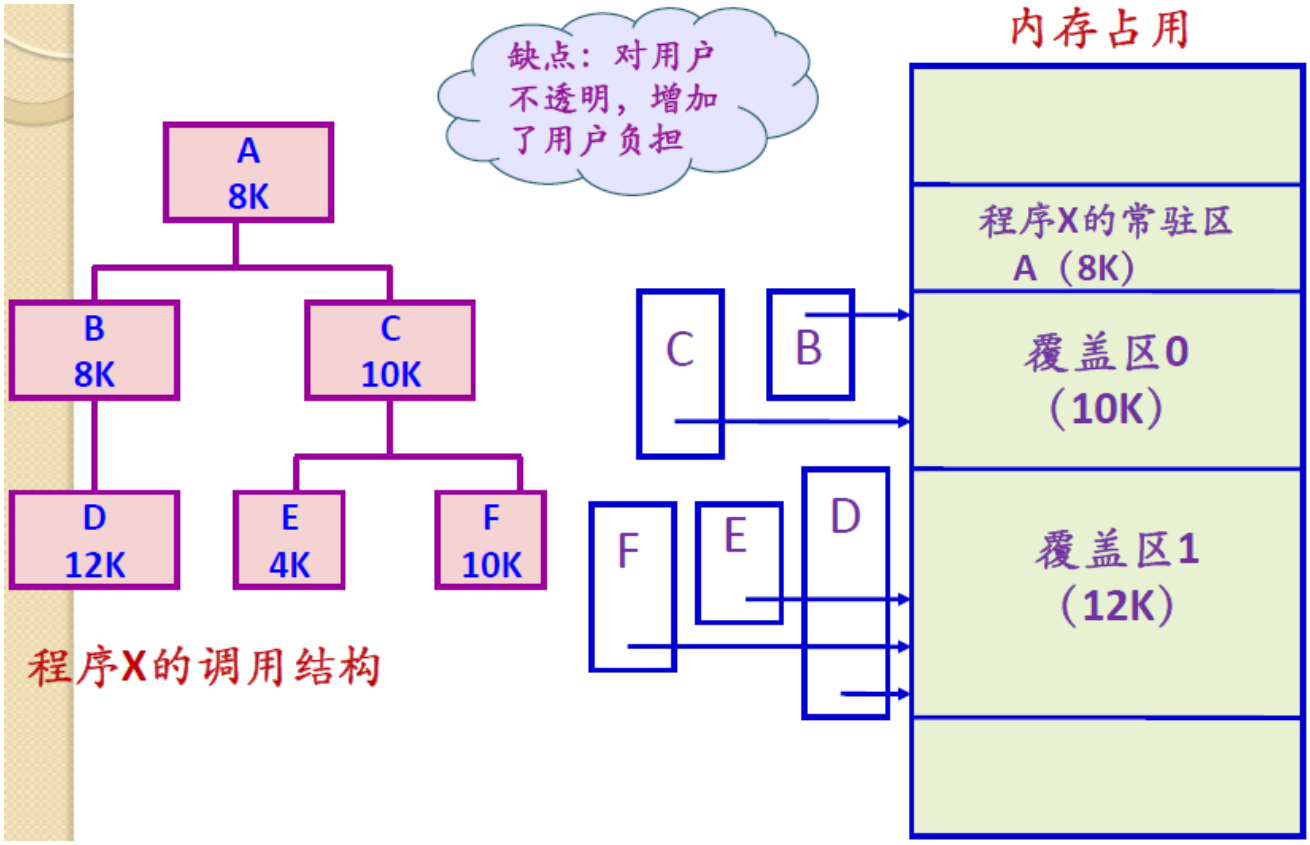
内存分配回收：同页式管理

内存“扩充”

内存紧凑：移动内存内容，将小的空闲区合并得到大的空闲区，从而使得内存外部碎片减少。

覆盖技术：意图解决程序大小超过物理内存总和的问题，在早期的操作系统上应用。程序执行的过程中，使得程序的不同部分相互替代，这些部分不会同时执行。

程序各个模块间都必须有明确的调用结构，要求程序员声明如何覆盖，增加编程复杂度；从外存装入覆盖模块，增加了执行时间，用时间换取空间。



交换技术：内存空间紧张时，将内存中某些进程移动到外存，将外存中某些进程换至内存占据前者所占用的区域，从而实现内存外存之间的动态调整。最早是用于小型分时系统。

固定不变的段：代码段，数据段等

运行时创建或修改的段：堆栈段

交换区：操作系统指定的一块特殊的磁盘区域作为交换空间，这一空间包含连续的磁道，操作系统可以使用底层的磁盘读写操作对其进行高效访问。

交换的时机：不用或很少再用时换出；内存空间不够或有不够的危险时换出。交换需要与调度器相结合。

不应换出处于I/O状态的进程——I/O将数据放置在内存中，换出该进程，会导致出错。

换出换入一次，进程不一定会回到原来的位置。

进程空间增长：通过动态分配堆栈段完成。在分配内存时给进程留出一部分空闲区域用作动态增长的堆栈段，换出时不需要将空闲的内存区域换出；空闲区域空间不足时则分配更大的内存区域或者直接换出。

虚拟存储：交换技术要求进程分配到的内存连续，即进程要么都在内存中，要么都在外存中。而这样的交换开销很大，虚拟存储允许进程在只有一部分存在于内存的情况下运行，从而降低开销，也提升内存的利用率。

虚存时构建在存储体系之上，提供给用户的一个比物理内存大得多的地址空间的“幻象”。

虚拟内存：将物理内存与磁盘结合使用，得到的一个容量大得多的内存。虚存大小受到计算机寻址机制和磁盘容量的限制。

虚拟地址：虚拟内存中的某一个位置，该位置可以被访问，看起来就是内存的一部分。

虚拟地址空间：分配给进程的虚拟内存。

虚拟存储技术：进程运行时，部分装入内存，其余暂存在磁盘，当要执行的指令/访问的数据不存在在内存中时，由OS自动将其从磁盘调入内存的技术。

虚拟页式存储管理

基本想法：装载程序时将几个甚至零个页面装入内存，在进程执行时发现需要的页面不在内存，引发page fault，动态地装入所需的页面；在需要的情况下，将内存中暂时不用的页面交换到磁盘以便获取更多的内存空间。

资源交换技术：以CPU时间和磁盘空间换区物理内存空间。

调页方式：请求调页（demand paging），预先调页（pre-paging）。

MMU：内存管理单元，将CPU的虚拟地址的访存转换为物理地址，并且进行权限检查等工作。

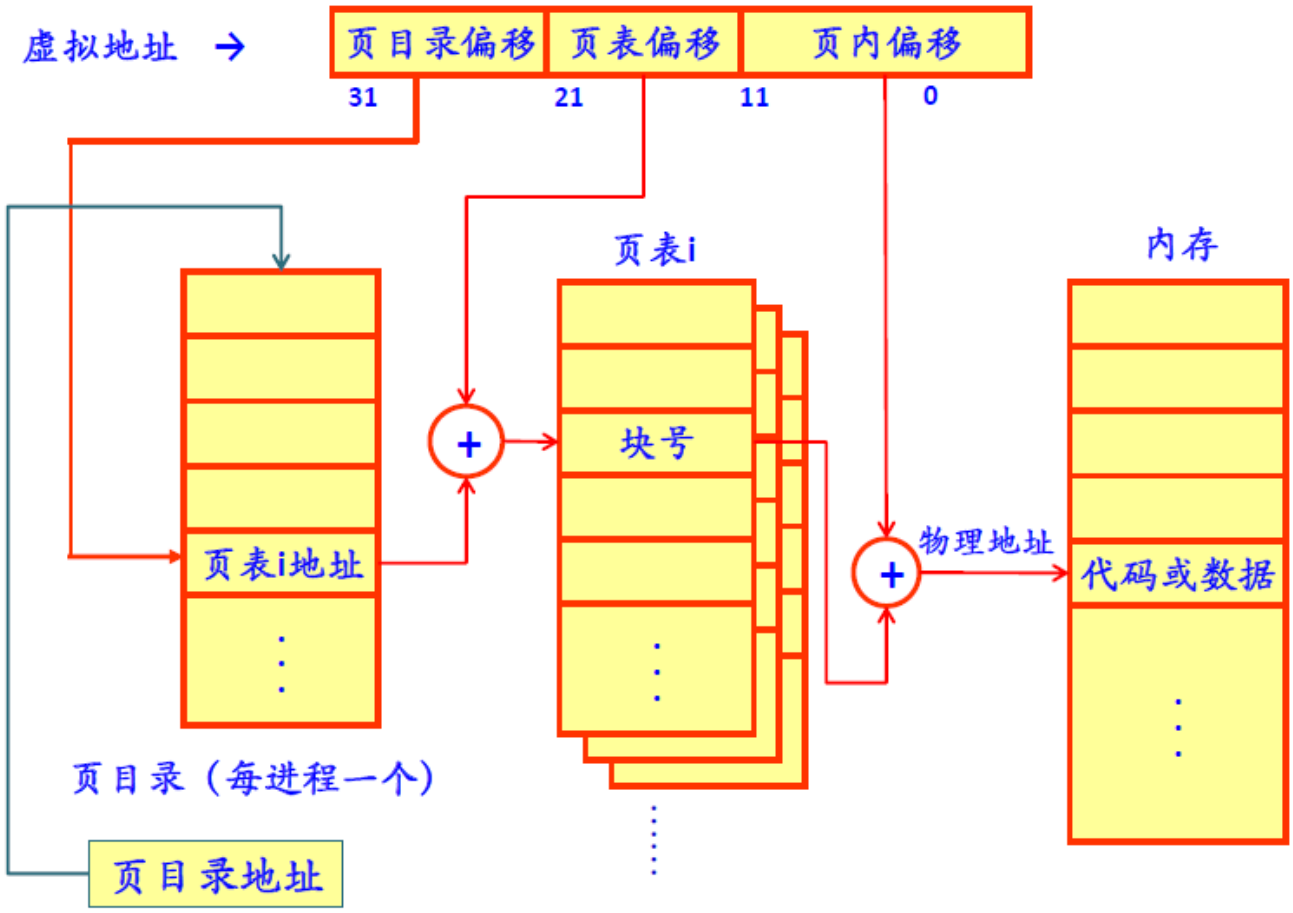
页表表项：页框号/内存块号/物理页面好/页帧号，有效位/驻留位（Valid，Present，表示该页在内存还是在磁盘），访问位/引用位（Referenced，Accessed，记录该页是否被读写过，用于page fault时寻找替换页面），修改位（Dirty，Modified，记录此页是否在内存中被修改过），保护位（Protection，读R写W执行E/X保护）。

禁止缓存位PCD：在映射为设备寄存器的页面进行设置，防止其进入高速缓存，而cache在设备寄存器变化时不再更新。

二级页表：一个线性的页表太大了，同时由于多进程的存在，每个进程一个页表，最终导致页表在内存中都存放不下，因此需要树形的二级/多级页表来压缩页表大小。

页目录：一个进程的页表的各页在进程中不连续存放，其地址索引为页目录。

地址映射：页目录地址固定，通过页目录偏移找到页目录项；页目录项存放下一级页表的页面起始位置，与页表偏移量相加找到下一级页表的页表项；页表项与页内偏移量相加得到虚拟地址对应的物理地址。

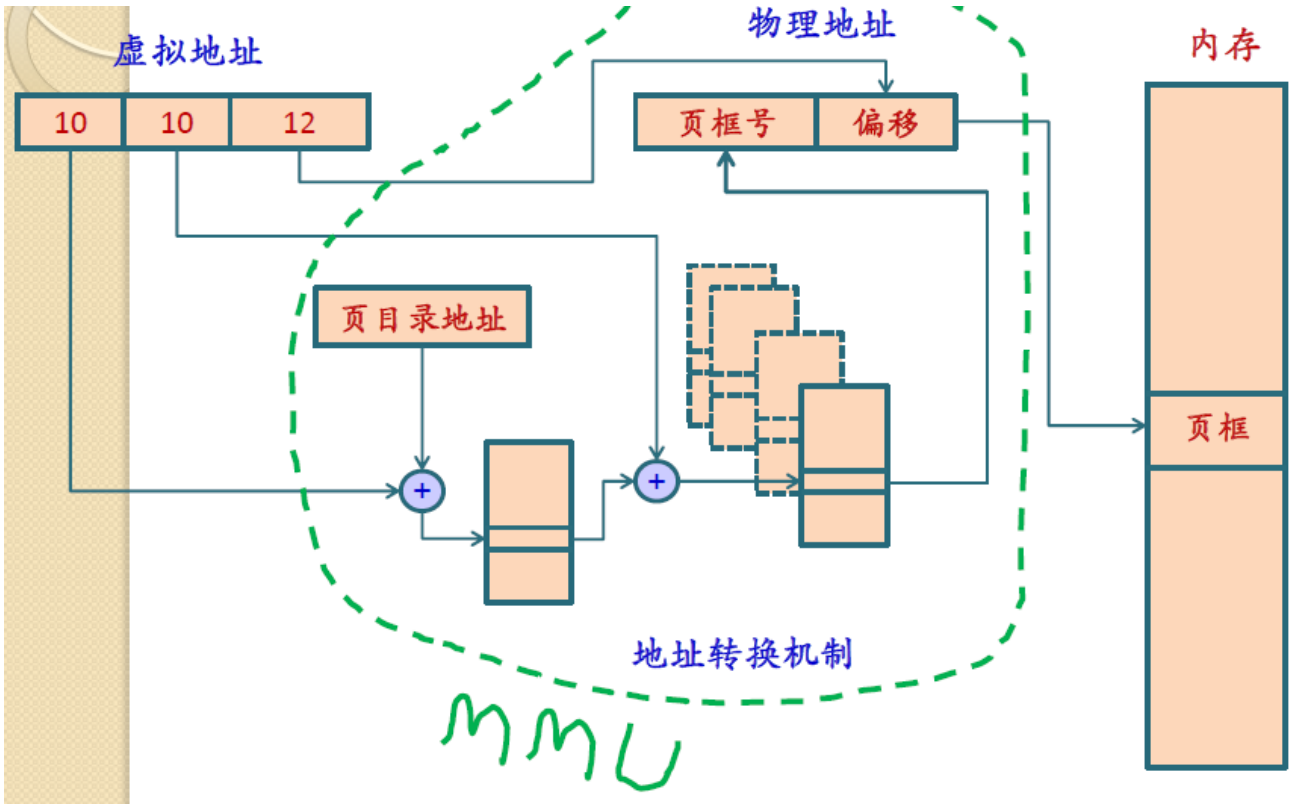


反转页表：另一种缩小页表的技术。页表用VPN作索引PPN作值，反转页表用PPN作索引VPN作值，从物理地址空间出发建立页表。优势在于可以极大地压缩页表大小，页表大小与物理内存成比例而与进程无关；但由于使用PPN作索引VPN作值，每次地址转换都需要搜索整个倒排表。

散列表：将虚拟地址页号与pid进行散列，散列表的值指向一个反转页表项，需要使用拉链法解决冲突问题。

地址转换过程：MMU硬件完成。

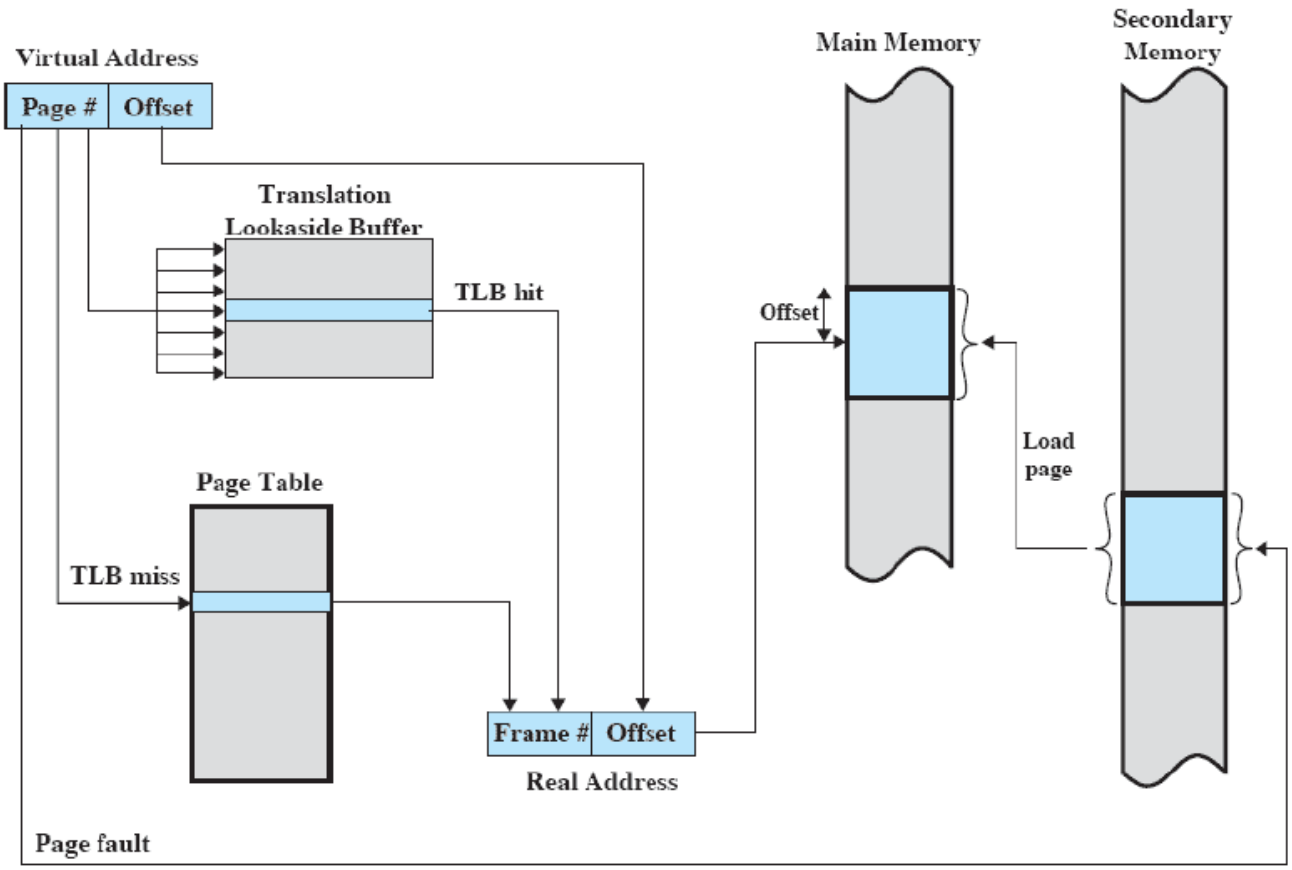
```
if (虚拟页面不在内存 or 页面非法 or 页面被保护)
{
  硬件产生异常(Page Fault)
  陷入操作系统
  执行page fault服务程序
}
else
{
  PPN = PT[VPN]
  PHYS_ADDR = [PPN | OFFSET]
}
```



快表TLB：一次页表访问至少需要两次以上（至少需要访问一次内存中的页表项，和访问一次与逻辑地址对应的物理地址）的内存访问，很容易成为计算机执行速度的瓶颈，因此利用程序的局部性原理，设计页表的"高速缓存"TLB。

硬件TLB一般位于MMU之中，包含少量的表项，相当于全相联/单路组相联的cache，按内容查找。TLB中保存正在运行进程的页表的活跃子集。

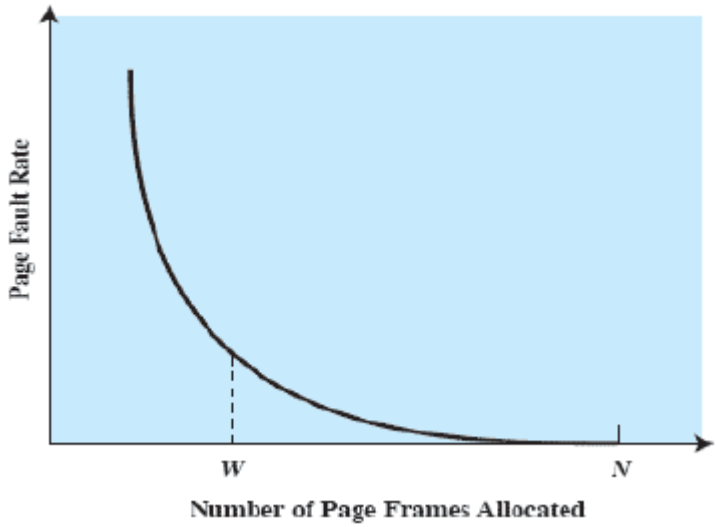
TLB使用：取得VPN，将其切为Tag和Index两段，其中Index用于找到表项，Tag用于检查该表项是否为所需表项，若是且该表项有效，则TLB Hit，直接得到PPN，产生物理地址；否则TLB Miss，需要查找页表得到PPN。



缺页异常Page Fault：地址映射时，硬件检查页表发现需要访问的页不在内存中，产生Page Fault异常。OS在Page Fault后调用异常处理程序，获得磁盘地址，启动磁盘并将该页调入内存中，若内存中有空闲页框，则分配一页，将新调入页装入内存并修改页表中相应表项的Valid位和PPN；若没有空闲页框，则置换某一页，若该页被写过则将其写回磁盘。

```
A. 在内存中有空闲物理页面时，分配页帧f，装至E步
B. 依据页面置换算法选择被替换的页帧f，其对应逻辑页q
C. 若q被修改过，则写回外存（磁盘）
D. 修改q的页表项中Valid位为0
E. 将需要访问的页p装入页框f
F. 修改p的页表项Valid位为1，PPN为f
G. 重新执行产生Page Fault的指令
```

驻留集：进程驻留集为进程可以停留在内存中的部分的集合，驻留集大小对应一个进程分配的页框数目。驻留集越大，Page Fault率越低；当驻留集超出一阈值N（或无限大）后，Page Fault率降为0；一般取使得Page Fault足够低但又没有N那么大的W作为驻留集大小。



固定分配策略：进程创建时确定一个固定的W，根据进程类型（交互、批处理、应用）或者程序员的定义或者系统管理员的定义来确定。

可变分配策略：根据缺页率来评估进程局部性表现。缺页率高时增加W，缺页率小时减少W。需要权衡系统开销。

置换问题：在产生Page Fault后，发现页框已满，需要选择一个页框进行置换。

局部置换策略：仅在产生本次缺页的进程的驻留集中选择。

全局置换策略：将进程中所有未锁定的页框作为置换的候选。

页框锁定：禁止操作系统将进程使用的页面换出内存，避免产生交换过程带来的不确定的延迟。用于OS核心代码和关键数据结构，以及I/O缓冲区（尤其是正在I/O的页面）等。

置换策略：决定置换当前内存中的哪个页框，目标是置换最近最不可能访问的页。一般情况下，可以利用局部性原理，基于过去的行为来预测将来的行为。置换策略越精致越复杂，一般实现其的硬件开销也就越大。

置换策略不得选择被锁定的页框进行置换。

清除：分页系统在最佳工作状态下，若发生Page Fault，应该可以在系统中找到大量空闲页框。因此**保证一定数目的页框供给比会使得在需要时搜索一个页框有更好的性能**。因而需要对页框进行清除。

设计一个分页守护进程paging daemon，大多数时间睡眠，定期被OS唤醒以检查内存状态。假如空闲页框过少，那么paging daemon通过预设的页面置换算法选择页框换出内存；如果页面被写过，则将其写回磁盘，以保证所有空闲页框都是干净的。

清除策略：基本想法是在需要使用一个已置换出的页框时，若该页框还没有被覆盖，则将其从空闲页框缓冲池中移出即可恢复页面。

清除策略实现：使用双指针时钟，前指针由paging daemon控制，当其指向一个dirty页面时，则将该页面写回磁盘，前指针向前移动；当其指向干净页面时，则仅仅向前移动。后指针用于页面置换，与标准十种算法一样。由于paging daemon前指针已经扫清所有脏页面，后指针命中干净页面的概率就会大大增加。

页缓冲技术：置换出的页面不被丢弃，若未被修改，则放到空闲页链表；若被修改，则放到修改页链表。修改页链表按簇的方式写回到磁盘，减少I/O次数。被置换的页面实际还保留在内存之中，一旦进程又要访问该页，则可以以很小的代价将其重新加入到该进程的驻留集之中。

加载控制：系统并发度对系统负载的影响存在峰值。利用这一特性，调节并发进程数来控制系统负载。

系统并发度：驻留在内存中的进程数目。

页面置换算法

最佳置换OPT：置换以后不在需要的，或者最远的将来才会用到的页面。不可能或者很难实现，需要先运行一遍程序，记录其访存过程，之后再重新重新运行一遍并根据记录选择替换的页面。OPT算法是不可能实用的，但可以用来作为上界来评价其他的置换算法。

先进先出置换FIFO：选择在内存中驻留时间最长的页并置换它。可以通过页面链表，或者页面队列进行实现。

第二次机会置换SRC：按照先进先出选择某一页面，检查其访问位R位，若为0则置换该页；否则为1，给该页面第二次机会，将其装入队尾，将R位置0并设置装入时间为当前时刻，之后从队首继续按SRC查找。

时钟算法CLOCK：SRC需要将链表节点来回移动，会造成比较大的开销，同时也不是很有必要。CLOCK算法将所有页面保存在一个环形链表中，用一个表针指向最老的页面，Page Fault时检查表针指向的页面，若R=0则置换该页面，否则R=1则清除R位并向前移动表针。

最近未使用NRU：选择最近一段时间内未使用过的一页置换。需要检查访问位R位和修改位M位，进程启动时R、M位都置为0，R位定期复位。根据R位和M位，将页面分为4类：第0类，近期没有访问，也从未被修改过，R=0，M=0；第1类，近期没有访问，但被修改过，R=0，M=1；第2类，近期有访问，但没有被修改过；第3类，近期有访问，也被修改过。从编号最小的非空的一类中选择一个页面进行置换。

NRU CLOCK：1. 指针位置起，选择第一个R=0，M=0的页框进行置换，该过程中R位不做修改；2. 第1步失败，重新扫描，选择第一个R=0，M=1的页框进行置换，该过程中将每个跳过的页框R设置为0；3. 第2步失败，指针回到初始位置，并且此时所有页框R位都被置为0，重复第1步和第2步。和NRU的区别在于是否随机。

最近最少使用LRU：选择最后一次访问时间距离当前时间最长的一页进行置换。性能接近于OPT，但由于必须对每个页面设置一个时间戳，或者维护一个访问页面的栈，开销很大（前者搜索的时间开销大，后者空间开销大）。

硬件实现：有P个页，则需要一个P*P的0-1表用于记录访问信息，每次访问时，先将访问的页的整行置为1，再将整列置为0。1最多的行则对应最近访问的页面，全0的行则对应最近最少访问的页。

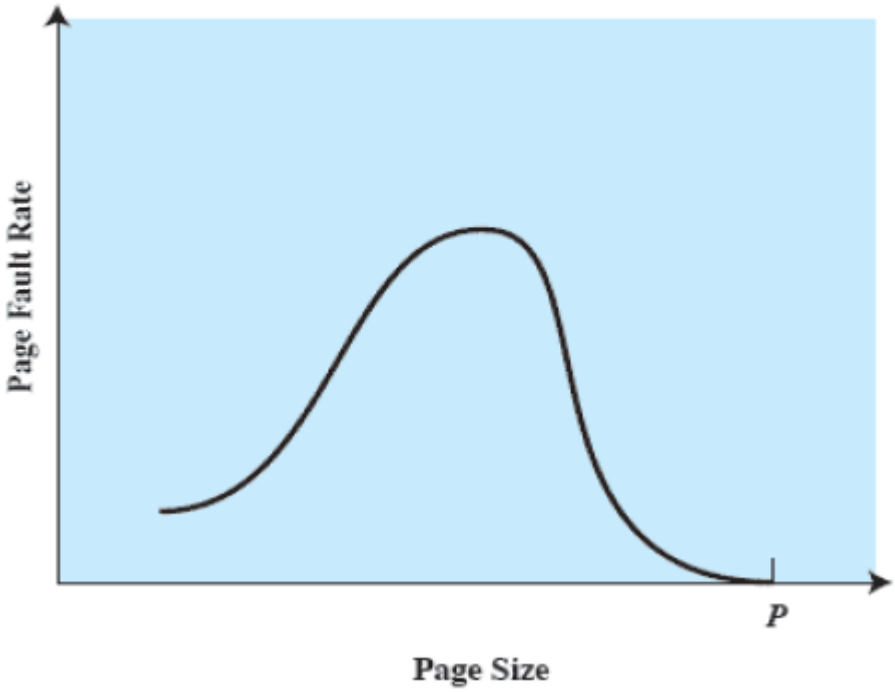
最不经常使用NFU：选择访问次数最少的页面进行置换，通过软件实现的LRU。对每一页设置一个软件计数器，初值为0，发生时钟中断时，页面计数器加上该页的R位；在发生缺页中断时，选择计数器值最小的页置换。

老化算法Aging：NFU从不忘记，一个被频繁使用的页面可能在很长时间之后，其计数器的值依然很大而没有被替换出去，这不符合LRU的想法。为了模拟LRU，采用Aging，计数器在加上R位前先右移1位，之后R位加到计数器的最左端。

Belady现象：FIFO置换产生的异常现象，当分配给进程的物理页面数增加时，缺页次数反而增加。

影响Page Fault次数的因素：页面置换算法，页面自身大小，程序编制方法，分配给进程的物理页面数。

页面大小p：只考虑内部碎片，页表大小的情况下，最优的$p = \sqrt{2se}$，其中s为进程的平均大小（byte），e为页表项的大小（byte）。在内存无限大时，缺页率与页面大小近似成反比；但由于Belady现象的存在，在页面增大的过程中，缺页率会有一个小幅的先升再降的过程。



程序的局部性越好，缺页率越低。

进程分得的页框数越多，缺页率越低，近似于反比例。

工作集模型：一般情况下，由于局部性原理，进程在一段时间内总是集中访问一些活跃页面。若进程能够分得的页框数大于等于活跃页面数，则可以降低缺页率；反之，活跃页面无法全部装入内存中，需要频繁交换，缺页率随之提高。

工作集W(t, △)：在当前时刻t时，进程在过去的△个虚拟时间单位中使用的虚拟页面的集合。工作集的内容受三个因素的影响——访存序列特性，当前时刻t，工作集窗口大小△。

驻留集：当前时刻，进程实际驻留在内存中的页框集合。

只有当一个进程的驻留集包含工作集，即工作集全部在内存中时，才可以更好的工作。

工作集算法Working Set：找出一个不在工作集中的页面并置换，使得内存中所有页面都尽量为工作集页面。设置窗口大小的时间值T，每个页表项中有一个字段记录该页面最后一次被访问的时间，根据访问时间落在t-T之前还是之后判断其在工作集之外还是之内。

```
扫描所有页表项，执行如下操作：
    若页面的R位为1，则将最后一次访问时间设置为t，R位清零
    否则R位为0，检查访问时间是否在t-T之前
        若是，则该页面在工作集之外，需要被置换
        否则记录当前所有被扫描过的页面的最后访问时间的最小值，重复1、2
```

工作集时钟置换WS Clock：WS的开销很大，因为需要扫描所有页表项才能确定被置换的页面。WS Clock使用循环表结构，初始时表空，随着页面装入，表逐渐被填充满。当指针指向的页面R=1时，该页面不能被置换，将R位清零，指针向前移动一位；否则R=0，检查访问时间是否落在t-T之前，若是则其不在工作集中，可以进行置换。考虑置换的两种情况，若M=0，则可以直接进行置换，将访问时间设置为t，R位置1；否则M=1，需要进行回写，避免频繁I/O，当需要回写的页面数到达I/O的最大限制时再进行会写。当指针转了一圈后，若进行过至少一次写操作，则等待I/O完成后从被写回的干净页面中选择一个进行置换；若没有进行过写操作，则所有页面都在工作集中，随机挑选一个进行置换。

算法	评价
OPT	不可实现，但可作为基准
NRU	LRU的很粗略的近似
FIFO	可能淘汰重要的页面
Second Chance	比FIFO有很大的改善
Clock	现实的
LRU	很优秀，但很难实现
NFU	LRU的相对粗略的近似
Aging	非常近似LRU的有效算法
Working set	实现起来开销很大
WSClock	好的有效的算法

Intel x86虚拟内存机制

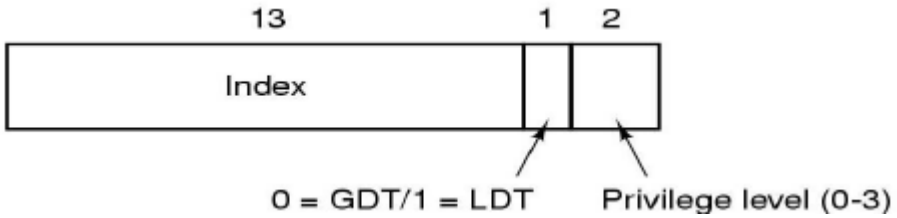
i386提供实模式，保护模式和虚拟8086模式（v86模式）3种工作模式。

实模式寻址：段式寻址，物理地址[19:0] = 段基址[15:0] * 16 + 偏移值[15:0]。段基址由CS，DS，ES，SS，FS和GS六个段寄存器提供，偏移值由通用寄存器或者指令立即数提供。

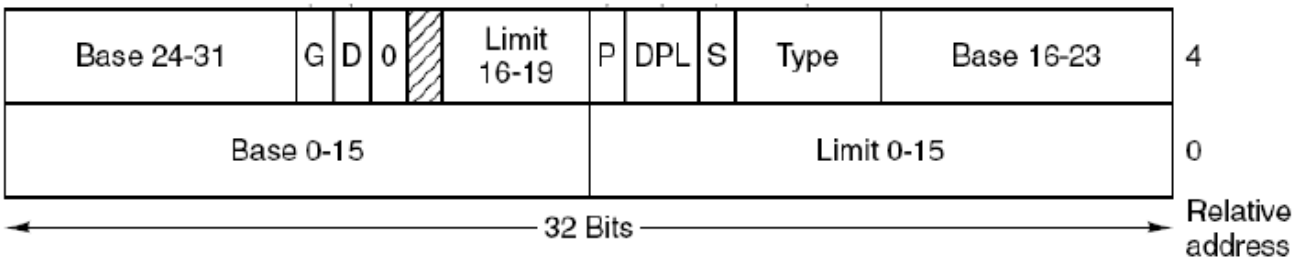
段寄存器：i386中，段寄存器共6个，每个16位，CS为代码段寄存器，DS为数据段寄存器，SS为堆栈段寄存器，ES，FS和GS都为附加段寄存器。

局部描述符表LDT和全局描述符表GDT：每个程序都有自己的LDT，一台计算机上仅有一个GDT，所有程序共享。LDT描述每个程序局部的段，GDT描述系统段，包括OS本身。

段选择符：总共16位。低2位描述0-3的特权级；第2位为0则说明该段位于GDT中，为1说明该段位于LDT中；高13位为段描述符在LDT或GDT中的索引。段选择符存放在段寄存器之中，为0表示不可用。



段描述符：记录一个段的基本信息，左右两端的Base拼接起来为该段的基地址，其余位域描述了段的长度，对齐方式，权限，是否在内存中，保护类型等等。



保护模式寻址：根据段寄存器中存放的段描述符，选择GDT或LDT，找到相应的段描述符，检查越界和权限等进行地址保护，若可以通过检查，则将段基址与偏移量相加得到一个线性地址。

若禁止分页，则该地址为物理地址，这种情况下寻址方式是段式寻址。

若允许分页，则该地址作为逻辑地址传给MMU，转化为物理地址后再读取内存，这种情况下寻址方式是段页式寻址。

线性地址分页：线性地址划分为10（目录域）+10（页面域）+12（偏移量域），地址转换时首先使用目录域和页面域共同查找TLB，若命中则得到PPN，拼接偏移量即完成地址转换；否则TLB Miss，需要查找页目录得到页表首地址，之后查找页表得到PPN。

$$逻辑地址 \rightarrow 段式转换 \rightarrow 线性地址 \rightarrow (允许分页)页式转换 \rightarrow 物理地址$$

绕过x86分段机制：将段基址设置为0，长度设置为最大，则只是用了一个最大的段，本质上就是绕过了分段的纯粹的分页寻址。

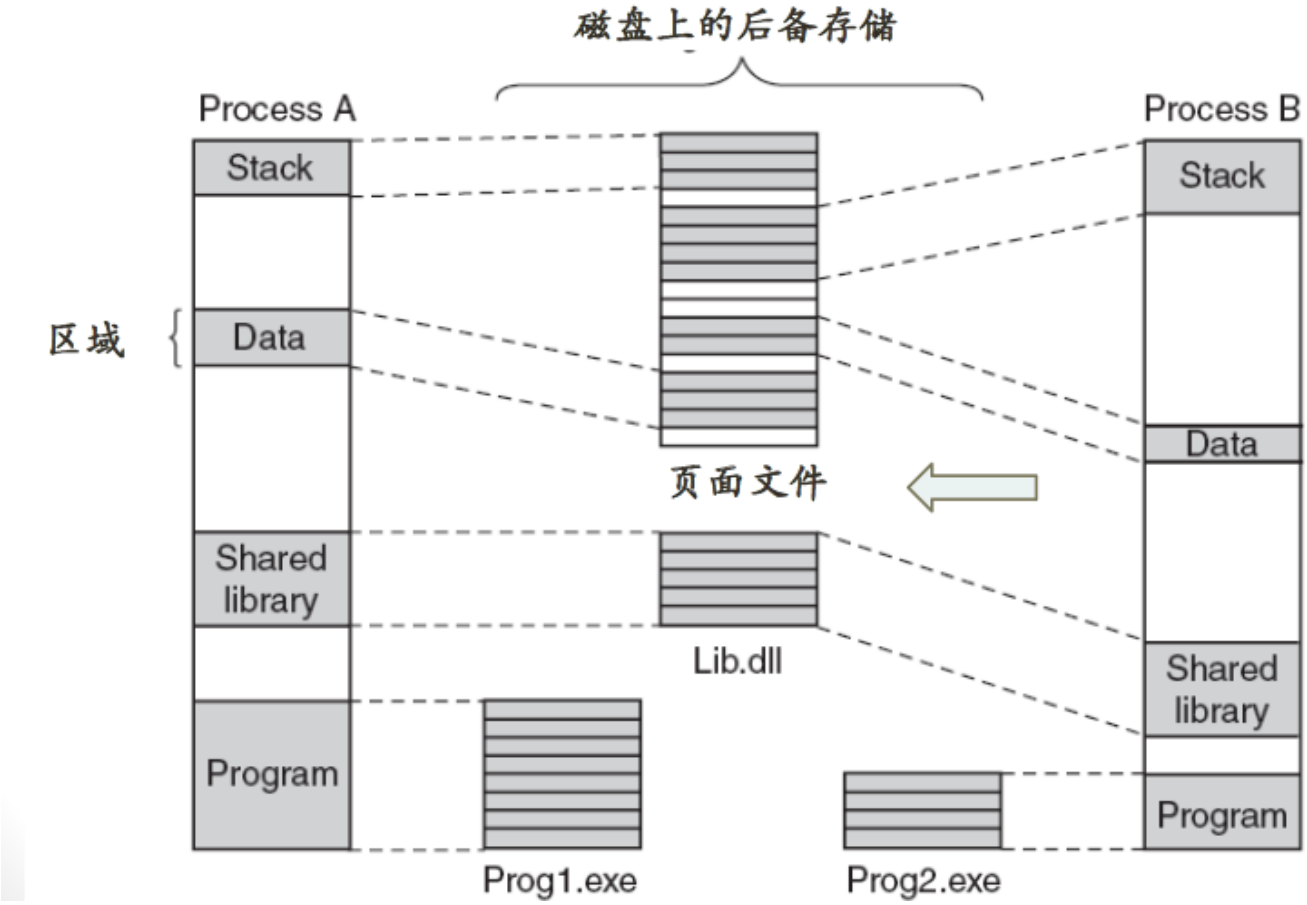
控制寄存器：CR0-CR3为i386的控制寄存器，其中CR1未定义留作以后使用；CR0控制系统状态，可以设置保护模式，允许分页等等；CR2为Page Fault线性地址寄存器，保存最后一次出现的32位的缺页线性地址；CR3位页目录基址寄存器，保存了页目录的基址，由于第12位始终对其，因此在页目录中低12位写入任何内容都不做理会。i486添加CR4，用于控制虚拟8086模式。

Windows虚拟存储管理

内存管理器：一组执行体系统服务程序，用于VM分配回收和管理；一个Page Fault陷阱处理程序，用于解决硬件检测到的内存管理异常，分配页框或将磁盘数据装入内存。

工作集管理器，进程/栈交换器，修改页面写出器，映射页面写出器，零页线程...

Windows虚存内容：EXE，DLL，动态分配的堆，操作系统支持



地址转换：各个进程私有页目录，通过页目录查找进入进程页表或系统页表，进程页表私有，系统页表共享。

TLB：同时读取比较所有TLB表项，选出VPN相同的即为Hit，否则为Miss。

有效PTE：页面存在于物理内存中，此时PDE和PTE的最低位都是1，说明V=1有效，CPU可以自动进行地址转换，不需要OS介入。当PTE无效时才需要OS负责。

无效PTE：Page Fault，可能的情况有——引用的页面没有提交，尝试访问高权限页面，修改一个共享的COW页面，栈需要扩大，引用的页已经被提交单尚未映射，请求一个零页面。

Page Fault异常：CPU地址转换的过程中发生PTE无效，引发Page Fault异常，CPU自动将虚拟地址存入CR2中；根据中断号，在中段描述符表中找到相应的中段描述符，再找到异常处理程序；异常处理程序通过CR2中的地址，计算出PDE/PTE地址，分析PTE内容发现是哪一种情况引起的异常，根据不同情况作出相应的处理。

页目录：映射进程所有页表位置的特殊的页表。物理地址保存在PCB之中，x86使用专用的CR3寄存器保存PD物理地址，可以通过虚地址进行访问，x86中将其映射为0xC0300000。

页目录自映射：PD[0xC0300000>>22] = PD[0x300] = PD，即PD中第0x300的表项存放的内容是PD的物理地址的基地址。通过在PD的0x300表项上自映射三次（0xC0300C00）即可得到页目录的物理地址。使用相同的方法，在PD的0x300表项上自映射两次，最后进入其余的页目录项即可得到对应的页表的物理地址；在PD的0x300表项上自映射一次，之后通过PD其他表项找到PT，再进入PT表项，即可得到页表项的物理地址（即逻辑地址对应的页框的基地址）。

```
MiGetPdeAddress():给定虚拟地址va，计算对应的PDE
    (PMMPTE)((((((ULONG)(va))>>22)<<2)+PDE_BASE))
MiGetPteAddress():给定虚拟地址va，计算对应的PTE
    (PMMPTE)((((((ULONG)(va))>>12)<<2)+PDE_BASE))
```

以页为单位的VM分配：用户程序经过保留和提交两个阶段使用一段地址范围。

保留：为线程将来使用所保留的一块虚拟地址。

提交：在已保留的区域中，提交页面支出将PM提交到何处以及提交多少。

虚拟地址描述符：每个进程维护一组VAD来描述一段被分配的进程虚拟空间的状态。VAD被构造为一颗自平衡二叉树来使得查找更有效率。通过VAD，可以快速查找到指定地址空间是否已经被分配（提交/保留）

利用区域对象实现内存映射文件：区域对象是一块可以被两个或多个进程所共享的内存块，使用区域对象将文件映射到进程的地址空间，之后访问这个文件就像访问内存中的一个数组，而不需要读写文件；若出现Page Fault，则内存管理器自动将页面从映射文件调入内存之中；若页面被修改，则常规调度时内存管理器将其写回文件。

2. 基本文件管理

KEY WORDS: 文件系统，文件，文件分类，文件控制块FCB，文件目录，目录文件，文件系统布局，文件逻辑结构，文件物理结构，文件描述符/文件句柄，FAT/UNIX，文件基本操作，内存结构，文件共享，磁盘空间管理

基本概念

文件：对磁盘的抽象，时一组带标识（文件名）的，在逻辑上具有完整意义的信息项的序列。

信息项：构成文件内容的基本单位，可以使单字节，也可以是多个字节，各个信息项具有顺序关系。

文件系统：OS中统一管理信息资源（持久性数据）的子系统，管理文件的存储、检索、更新，提供安全可靠的共享和保护，方便用户使用。

```
统一管理磁盘空间，实施磁盘空间的分配和回收
完成文件名空间和磁盘空间的映射，实现文件的按名存取
实现文件信息的贡献，提供数据可靠性和安全保障
向用户提供方便使用和维护的接口，并向用户提供相关信息
提高文件系统性能
提供与I/O系统的统一的接口
```

UNIX下按文件性质和用途分类：普通文件，目录文件，特殊文件（设备文件），管道文件，套接字，符号链接文件。

普通文件：包含了用户信息，一般为ASCII或Binary文件

目录文件：管理文件系统的系统文件

特殊文件：字符设备文件（模仿串行I/O设备）或块设备文件（模仿磁盘）

文件逻辑结构：字节序列文件，记录序列文件，记录树文件等

字节序列文件：操作系统不关心也不知道文件的内容，操作系统只看到字节的序列，文件内容的解释由用户程序进行。Windows和UNIX都采用这一方法。

记录序列文件：文件的基本单位是具有固定长度的"记录"，每个记录都有其内部结构。在历史上穿孔卡片还是主流时，这一结构被大型机系统普遍采用。

记录树文件：每个记录长度不一定相同，但每个记录的固定位置有一KEY字段，按照KEY进行排序从而可以快速查找。用户可以写入和读出，但不能指定位置；操作系统决定文件如何组织。处理商业数据的大型机采用这一结构。

文件存取方式：顺序存取（按字节依次读取），随机存取（从任意位置读写，需要提供读写的位置，eg. UNIX中的seek）。

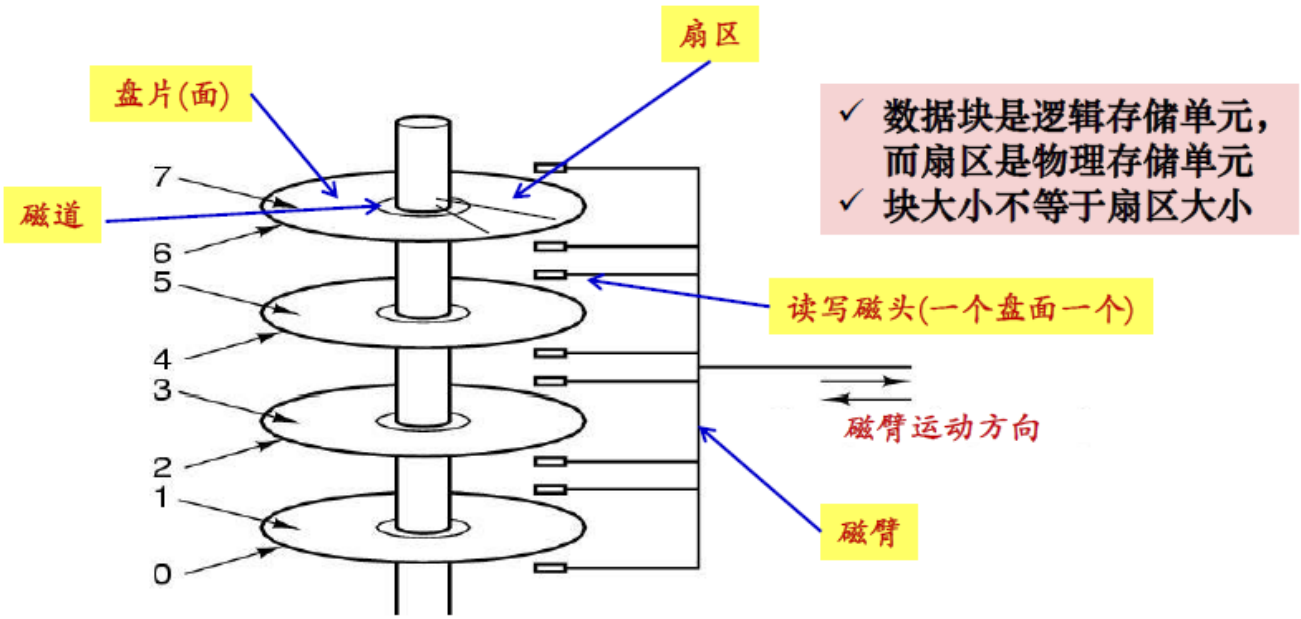
典型存储介质：磁盘，SSD盘，磁带，光盘，软盘，U盘...

物理块/块/数据块：存储设备被划分位大小相等的块，统一编号，在信息存储传输分配时以块为单位进行。

磁盘结构：盘片/盘面，磁道，扇区，柱面，读写磁头，磁臂。

物理地址形式：磁头号/盘面号+磁道号/柱面号+扇区号

输入输出数据流以位串形式出现，任何时刻都只有一个刺头处于活动状态。



SSD：将数据存放在闪存之中，没有碟片的外形，也没有可以移动的部分。

磁盘访问：读/写，磁盘地址（设备号，柱面号，磁头号，扇区号），内存地址（源/目标）。

寻道：磁头移动定位到指定的磁道

旋转延迟：等待指定的扇区从磁头下旋转经过

数据传输：数据在磁盘与内存之间实际传输

文件控制块FCB：操作系统为了管理文件而设置的数据结构，存放了管理文件所需要的所有有关信息（文件属性/元数据）。

常用属性：文件名，文件号，保护类型，口令，创建者，当前拥有者，文件地址，文件大小，文件类型，共享计数，创建时间，最后修改时间，最后访问时间，标志（只读、隐藏、系统、ASCII/二进制，顺序/随机访问...）

文件操作：create，delete，open，close，read，write，append，seek，rename，get_attributes，set_attributes...文件访问模式规定，进程访问文件数据之前必须先显式地打开文件（open）。

文件目录：统一管理每个文件的信息，将左右文件的管理信息组织在一起，形成文件目录。

目录项：构成文件目录的基本单元，目录项可以是FCB，目录是FCB的有序集合。

目录文件：文件目录按照文件的形式存放在磁盘上，是一种特殊类型的文件，内容由目录项组成。为了保证安全性和映射的完整性，只允许内核修改目录，应用程序通过系统调用进行访问。

树形目录结构：根节点为**根目录**，由系统所有，记录所有用户目录；根节点下一层为用户目录，每个目录下可能有若干用户子目录和用户文件。

绝对路径名：从根目录开始，UNIX系统下一般以"/"开头表示。

相对路径名：从当前目录开始，UNIX系统下一般以"./"开头表示。

当前目录/工作目录：每个进程拥有一个，可以进行改变，用于解析相对路径名的文件名。

文件系统的实现-1

实现文件系统需要考虑磁盘与内存中的内容布局。在磁盘上如何启动存储的操作系统，如何在磁盘上存放和管理目录文件和普通文件；进程在内存中使用文件时，OS如何进行支持和管理。

磁盘分区partition：将一个物理磁盘的存储空间划分为几个相互独立的部分，每个部分为一个分区。

文件卷volume：逻辑分区，由一个或多个物理块（簇）组成。

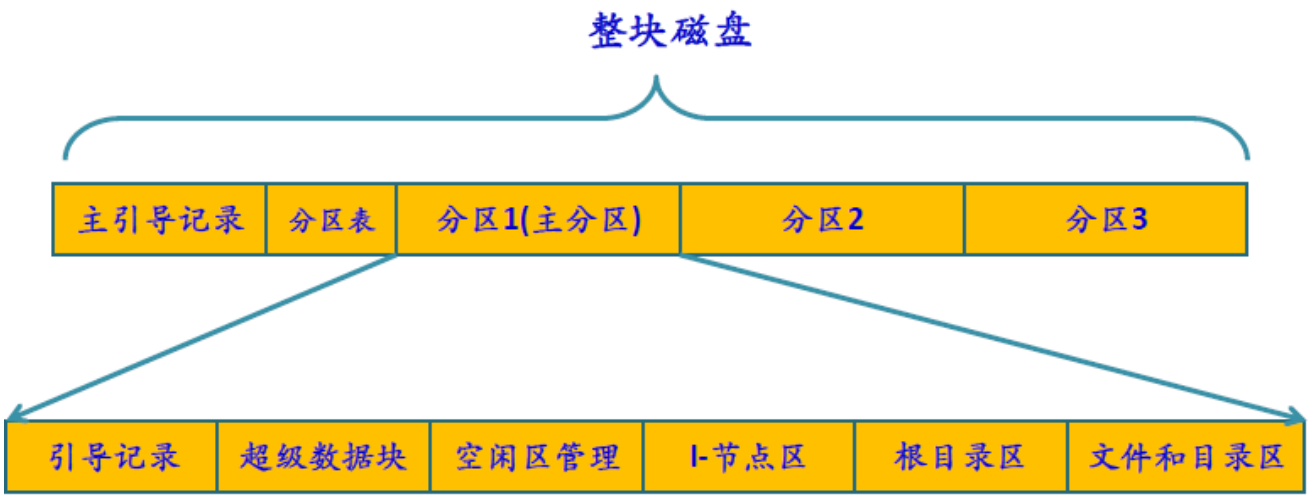
物理块block/簇cluster：一个或多个（2^n个）连续的扇区，可寻址数据块。存储设备被划分成大小相等的物理块并统一编号，物理块是逻辑存储传输分配的独立单位。

一个文件卷可以是整个磁盘或部分磁盘或跨盘（RAID），在同一个个文件卷中使用同一份管理数据进行文件分配和磁盘空闲空间管理，不同的文件卷中的管理数据独立。具体的一个文件卷包括，文件系统信息，一组目录文件和用户文件，以及未分配的空间。

格式化format：在一个文件卷上建立文件系统的过程，即建立并初始化用于文件分配和磁盘空闲空间管理的**元数据**/管理数据。

磁盘上的内容：**引导区**，包括从该卷引导操作系统所需要的信息，每个卷/分区一个，通常为第一个扇区。卷/分区信息，包括该卷/分区的块/簇数，块/簇大小，空闲块/簇的数量和指针，空闲FCB数量和指针。目录结构/目录文件和用户文件。

UNIX文件系统布局：主引导记录MBR确定活动分区，读入并执行引导块（第一个块）；分区表给出每个分区的起始和结束地址；引导块中存储一段程序，用于装载该分区中的操作系统；超级数据块包含文件系统的关键数据，一般包括Magic Number，块数量等信息。



文件的物理结构：文件在物理介质上的存放方式，系统分配给文件的物理块的位置和顺序。需要考虑存储效率（外部碎片问题）和读写性能（访问速度）。

连续/顺序结构：文件的信息存放在若干连续的物理块之中。FCB记录起始地址和终止地址即可。

简单高效，支持顺序存取和随机存取，磁盘寻道次数和寻道时间最少，可以同时读入多个块；文件不能动态增长，如果预留空间会造成浪费，移动和重新分配回造成开销，造成外部碎片，需要压缩紧致，不利于文件内容删除和插入。

链接结构：一个文件的信息存放在若干不连续的物理块之中，各个块之间通过指针链接，一个物理块指向下一个物理块。FCB中记录起始文件块地址即可。

提高磁盘空间利用率，不存在外部碎片问题，有利于文件内容插入和删除，有利于动态扩充；不适合随机存取，存取速度缓慢，寻道次数多，寻道时间长，链接指针固定占用空间，可靠性不足，链接指针出错会引起文件系统整个出错。

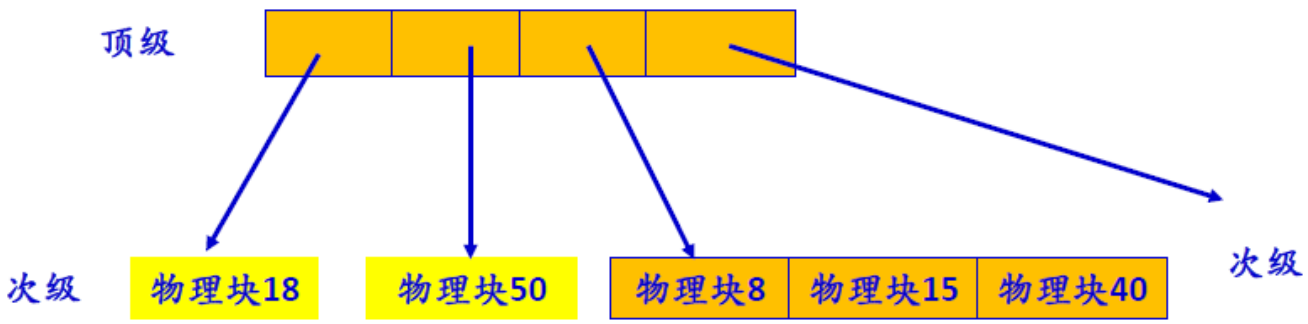
文件分配表FAT：在内存中建立一个表，每个表项对应一个磁盘块指针，该指针即为链接结构中的下一块指针。FCB中记录起始块号即可。

随机存取只需要在内存FAT表上行走一遍即可，数据块可以全部用来存放数据；内存中存放整个FAT表，空间开销很大，对于大型磁盘甚至无法存放在内存之中。

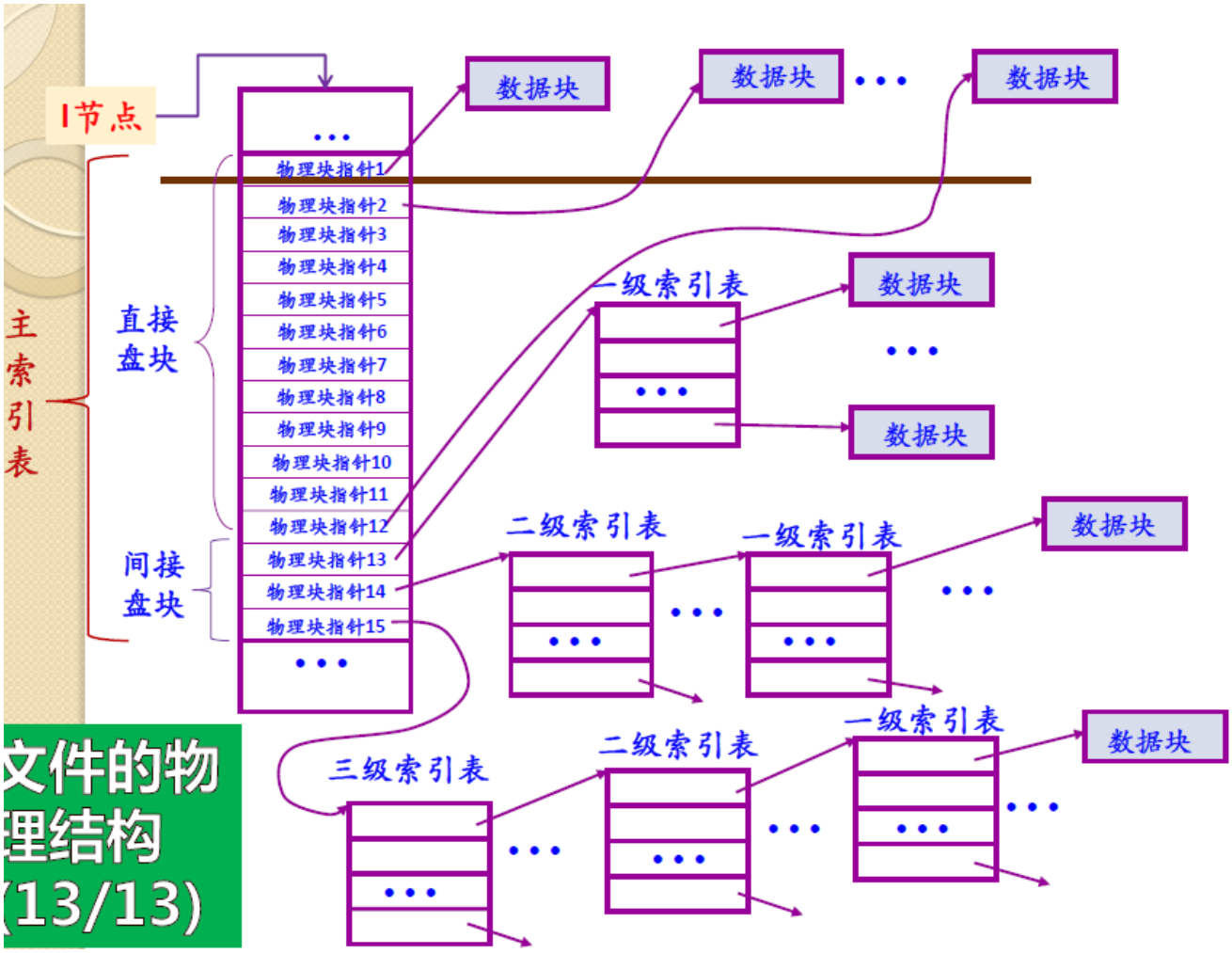
索引结构：一个文件的信息存放在若干不连续的物理块之中，系统为每个文件建立一个专用的索引表存放这些块号，索引表的第i个条目指向文件的第i块。FCB中记录索引表的数据块地址即可。

保持链接结构优点，同时既能顺序存取又能随机存取，满足文件动态增长和插入删除的需要；需要较多的寻到次数和较长的寻道时间，索引表本身会带来系统开销。

扩充索引表：当索引表需要存放的物理块很多时，需要进行扩充。链接方式将多个索引表链接起来，多级索引方式将二级索引地址存放在一级索引之中，综合方式将直接索引和间接索引相结合。



UNIX文件系统采用多级综合索引。每个文件的索引表有15个索引，每项2 Byte，最前面的12项直接登记存放文件信息的物理块号；第13项指向一个物理块，该物理块作为一级索引表，最多存放256个文件块号；第14项指向二级索引表，第15项指向三级索引表。



目录文件的组织方式：将目录项组织为FCB，散列表（将目录项组织为散列表，查找速度快），B/B+树。

文件目录检索：目录检索+文件寻址

目录检索：根据用户给出的文件名查找目录项/FCB，需要进行文件名解析，将逻辑名字装换成物理资源找到目录项/FCB

文件寻址：根据目录项/FCB中的文件物理地址信息，计算文件中的记录在存取介质上的地址。

加速目录检索的方法：将FCB分解为两部分，一部分为符号目录项（文件名，文件号），另一部分为基本目录项（除去文件名外的所有字段）。将符号目录项连续存放，基本目录项连续存放。

文件系统实例

UNIX文件系统：FCB=目录项+i节点，目录项=文件名+i节点号，i节点记录文件属性和索引表。每个文件由一个目录项，一个i节点和若干个磁盘块组成。

查找路径：首先定位根目录，根目录的i节点在磁盘上固定，使用该i节点直接找到根目录；之后读根目录查找用户目录，得到用户目录的i节点号，根据该i节点号在i节点区查找目录的i节点；读目录，递归地查找，直至得到普通文件的i节点号；根据i节点的索引打开该文件。

Windows FAT16文件系统：文件系统数据记录在引导扇区中，FAT描述簇的分配状态并标注下一簇的簇号。根目录大小固定，FAT表项2 byte，目录项32 byte。

引导区	文件分配表1	文件分配表2	根目录	其他目录和文件

MBR：主引导记录位于0号扇区，用于引导整个磁盘和加载分区表。

DBR：每个分区的第一个记录都是引导区，用于加载该分区的操作系统。

FAT：看做一个整数数组，没个整数代表一个簇号，内容为该簇状态（未使用，坏簇，系统保留，文件占用存放下一簇簇号，最后一簇0xFFFF），簇0和簇1保留，从簇3起可以被文件使用。

FAT16目录项：存放文件属性，包括文件名，文件扩展名，文件属性字节，保留段，最后一次修改时间，最后一次修改日期，起始簇号，文件大小等。

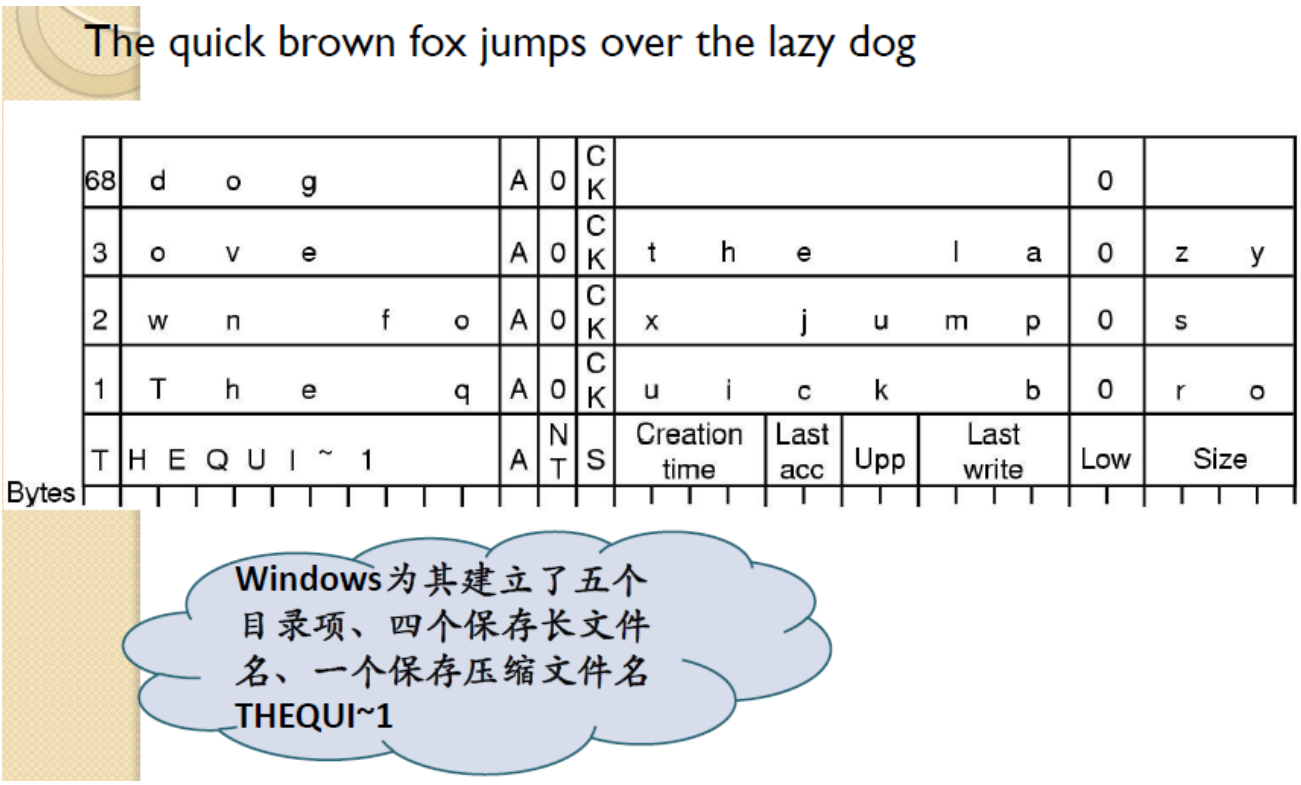
FAT32：根目录区大小和区域不固定，而是作为数据区的一部分，采用与子目录文件相同的管理方式。目录项32byte，添加新的属性，支持长文件名。

FAT32目录项：FAT16目录项，添加创建时间，文件创建时间，文件创建日期，文件最后访问日期，起始簇号高16位和低16位（FAT16中的起始簇号）

长文件名实现：1. 不定长目录项；2. 定长目录项+命名块，目录项中使用一个指针指向命名块的该名字的起始位置；3. 使用长文件名目录项扩展。

FAT32长文件名目录项：32byte，用于扩展短目录项的文件名，使用unicode编码存放文件名，一个长文件名目录项可以存放13个unicode字符。将长文件名拼接在目录项之后，用于扩展文件名。

偏移	长度	含义
00h	1	位0-5给出序号,位6 表示长文件最后一个目录项
01h	10	长文件名(unicode码)① 前5个字符
0Bh	1	0x0F(长文件名目录项标志)
0Ch	1	保留
0Dh	1	校验值(由短文件名计算得出)
0Eh	12	长文件名(unicode码)② 6个字符
1Ah	2	文件起始簇号，常置0
1Ch	4	长文件名(unicode码)③ 2个字符



文件系统的实现-2

磁盘空间的管理：位图法，空闲块表，空闲块链表。

已知块号，则磁盘地址：

柱面号=[块号/（磁头数×扇区数）]

磁头号=[（块号mod （磁头数×扇区数））/扇区数]

扇区号=（块号mod （磁头数×扇区数））mod 扇区数

已知磁盘地址：

块号=柱面号×（磁头数×扇区数）+磁头号×扇区数+扇区号

位图计算公式：

已知字号i，位号j：块号=i×字长+j

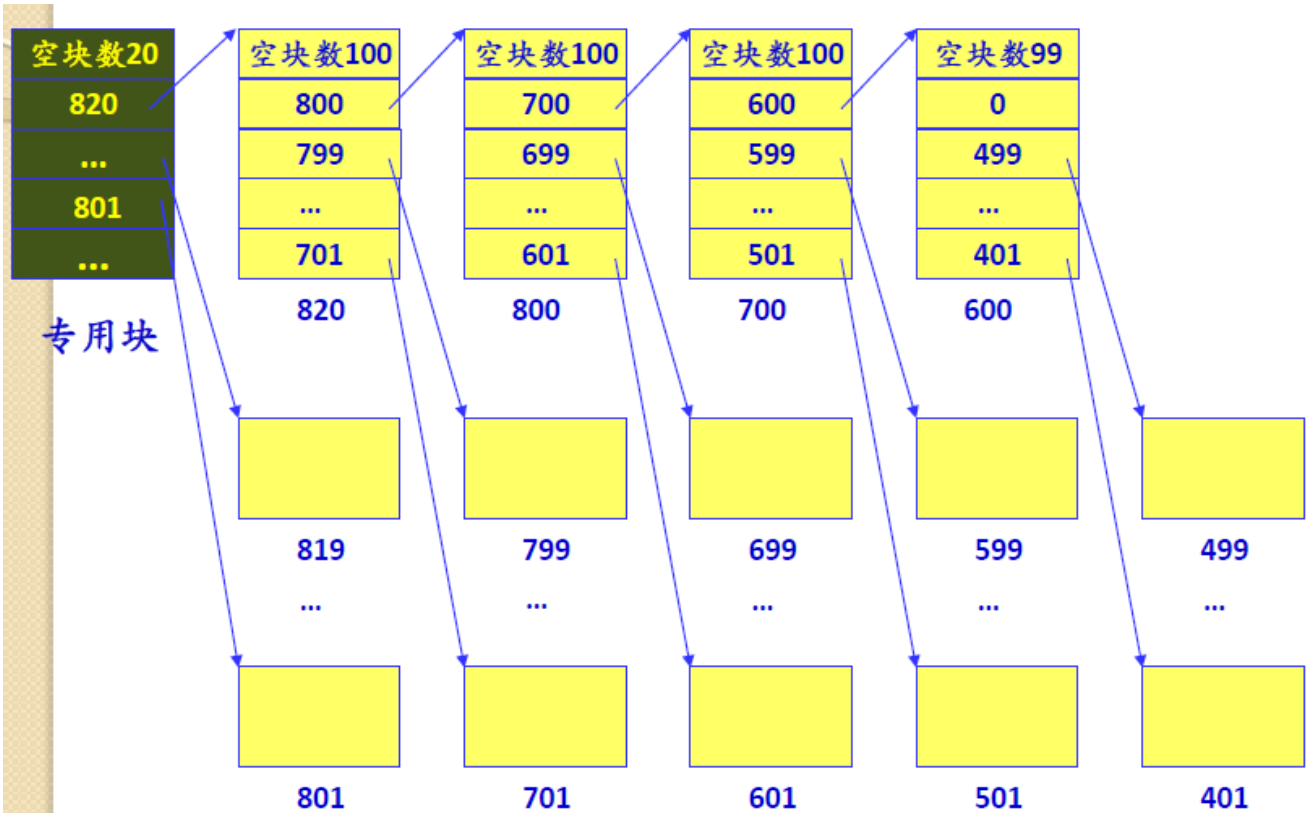
已知块号：字号=[块号/字长]　位号=块号 mod 字长

位图：用一串2进制位反应磁盘空间中的分配情况，分配物理块位0，空闲物理块为1。

空闲块表：所有空闲块记录在一个表之中。

空闲块链表：所有空闲块连接成链，可以使用成组链接法扩展。

成组链接法：将空闲块成组进行链接，类似综合的多级索引，UNIX下的空闲块管理策略。将空闲块分为100个一组，每组的第一个空闲块记录了空闲块总数和下一组空闲块的物理盘块号，空闲块号为0则说明是最后一组。一个100个块的组，实际上只有99个空闲块可用。分配时从前往后分配，回收时从后向前回收。



分配一个空闲块：

将专用块内容读入内存L开始的区域，查L单元（空闲块数）内容

若空闲块数>1，i<-L+空闲块数，从i单元得到空闲块号，分配给申请者，空闲块数减1

若空闲块数=1，取出下一盘号

若L+1单元内容=0，则没有空闲块，申请者等待

若不等于0，则将该块内容复制到专用块，将该块分配给申请者

归还一个空闲块：

将专用块内容读入内存L开始的区域，查L单元的空闲块数

若空闲块数<100，空闲块数加1，j<-L+空闲块数，归还块号填入j单元

若空闲块数=100，则把内存中登记的所有信息写入归还的块中，将L置为1并将归还块号填入L+1单元

文件表：用于保存打开的FCB，属于运行时文件结构，存放在内存之中。

系统打开文件表：整个系统只有一张，存放在内存中，保存已打开的FCB；表项包括FCB的i节点信息，引用计数，修改标志等。

用户打开文件表：每个进程一张，维护打开文件的状态和信息，在PCB中记录；表项包括文件描述符，打开方式，读写指针，系统打开文件表入口等，其中系统打开文件表入口指向系统打开文件表中的一个表项。

文件描述符fd：打开文件的标识。

文件操作：创建、删除、打开、关闭、指针定位、读、写。除去创建和删除操作，其余所有文件操作都必须先打开文件，获得其描述符或文件句柄。

创建文件：分配必要的磁盘空间，建立新文件与目录（文件系统）的联系。

```
1. 检查参数合法性（命名是否符合规范，有无重名等）
2. 申请空目录项，填写相关内容
3. 申请磁盘块
4. 返回
```

打开文件：为文件读写操作作准备，给出路径名，获得文件句柄或描述符。

```
fd = open(文件路径名，打开方式)
1. 根据文件路径名查找目录，找到对应的目录项（或i节点号）
2. 根据文件号查系统打开文件表，查看是否已经打开
    若是，则共享计数加一
    否则将目录项（或i节点）信息填入打开文件表空表项，共享计数置为1
3. 根据打开方式，共享说明和用户身份检查访问的合法性
4. 用户打开文件表中建立空表项，填写打开方式并指向系统打开文件表对应表项
5. 返回文件描述符fd，用于读写文件的非负整数
```

指针定位：设置读写指针相对于文件开头的偏移位置，从而实现随机读写。

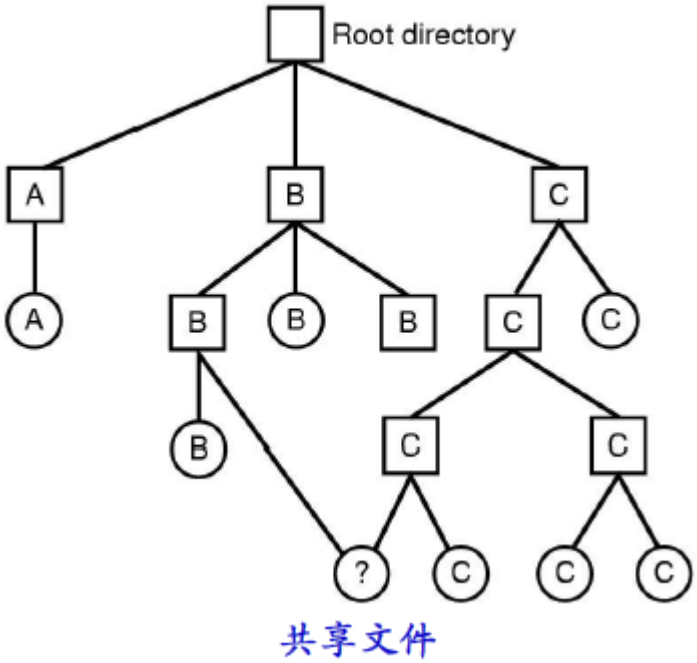
```
seek (fd，新指针位置)
1. 由fd查用户打开文件表，找到对应的入口
2. 将用户打开文件表中的读写指针设置为新的指针位置
```

读文件：文件基本操作之一，读取文件内容至内存。

```
read(fd，读指针，读取长度，内存目的地址)
1. 根据fd，找到相应的FCB（目录项），检查读操作合法性（权限，文件是否打开等）
2. 将文件的逻辑块号转换为物理块号，根据读指针和读取长度确定块号，块数和快内偏移
3. 申请缓冲区
4. 启动磁盘I/O，将磁盘块中的信息读入缓冲区，再传送到指定的内存区中
5. 反复执行34，直至读取全部数据或读至文件末尾
```

文件共享：一个文件被多个用户或进程使用，为了交换信息或节省时间和存储空间。可以通过文件别名的方式实现。

硬链接：多个路径名描述同一个共享文件，多个目录项指向同一个共享文件。



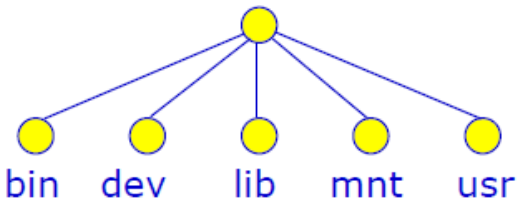
共享文件

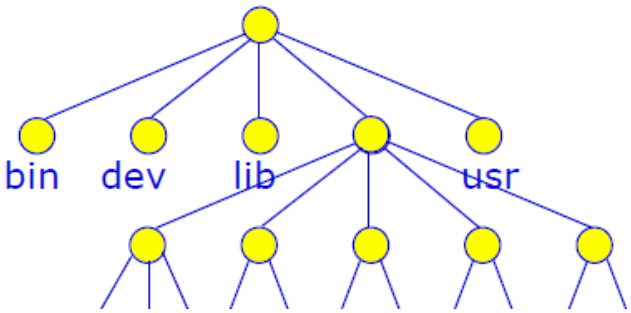
Linux的链接机制：Linux下目录项分文文件名和索引节点，多个文件名可以链接到同一个索引节点，即对一个索引节点（即文件）建立起多个平等的别名；别名数目记录在索引节点的链接计数中，删除会使其减1，减至0时则文件被删除。

软链接/符号链接/快捷方式：一种特殊类型的文件，其内容是要共享的文件的路径名。别名关系并不对等，只有真正的文件所有者才持有i节点。系统开销大，目录结构可能成环；计算机网络环境下有事明显，可以建立任意的别名关系，任意链接其他计算机。

挂载：将一个文件系统加入另一个文件系统中，用户需要提供被挂载的文件系统的根目录/挂载点。

卸载：将挂载的子文件系统从文件系统中去除。





文件系统的管理

可靠性：抵御和预防各种物理性破坏和人为性破坏的能力。其中物理性破坏很重要的就是**坏块问题**。

备份：通过转储操作，形成文件或文件系统的多个副本。

全量转储：定期将所有文件拷贝到后援存储器。

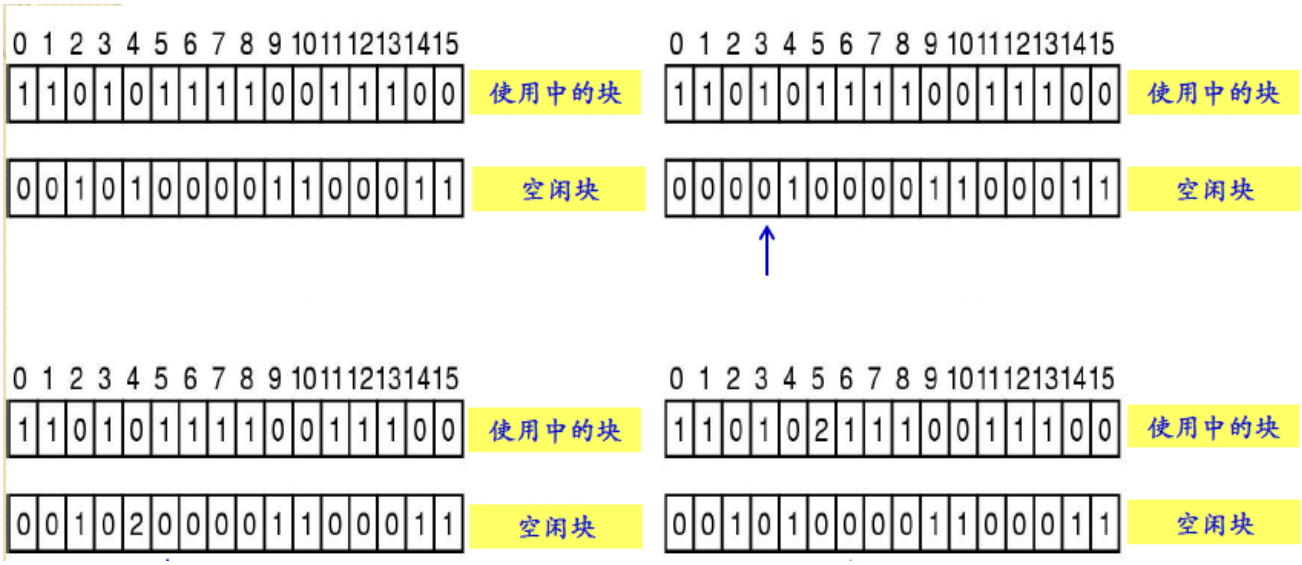
增量转储：只转储修改过的文件，减少系统开销。

物理转储：从磁盘第0块开始，将所有磁盘块按序输出到后备存储。

逻辑转储：从一个或几个指定目录起，递归地转储自给定日期后更改的所有文件和目录。

文件系统一致性：文件读写必须将磁盘块内容读入内存，在内存中完成读写操作，若有修改再写回磁盘。但如果在写回磁盘前，系统崩溃，则会出现文件系统不一致的问题。

UNIX一致性检查：**fsck**实用程序。使用空闲块和分配块的两张表，fsck检查所有i节点并填充分配块表，每个块出现一次则在表中计数器加1；接着检查空闲组链接表并填充空闲块表。若没有一致性问题，则两表的值均为0-1且互补；若出现两表某位均为0，则出现块丢失；若空闲块表某位大于1，则出现重复空闲块；若分配块表某位大于1，则出现重复数据块。



文件系统写入方式：通写，延迟写，可恢复写。

通写：FAT使用的方法，内存修改时立刻写回磁盘，性能差但一致性基本不受影响。

延迟写：利用缓存Write back的方法实现高速文件操作，但可恢复性差。

可恢复写：NTFS使用的方法，采用事务日志的方式来实现文件系统的写入，既考虑到了安全性，又考虑到了性能。

安全性：确保未授权用户不能存取某些文件。

数据丢失：可以通过备份解决。

入侵者：需要考虑哪一类入侵者。

文件保护机制：文件保护。对于拥有权限的用户，允许其操作，否则禁止，防止没有权限的用户冒充拥有权限的用户进行操作。

用户身份验证：口令或物理鉴定（磁卡指纹等）。

访问控制：访问控制表和能力表，均存放在内核空间。访问控制表每个文件一个，记录UID和权限；能力表（权限表）每个用户一个，记录文件名和访问权限。

UNIX的文件保护：二级存取控制，审查用户权限，审查本次操作的合法性。对用户分类为文件主（owner），文件主同组用户（group）和其他用户（other），用于第一级对访问者进行识别；对操作分类为读（r），写（w），执行（x）和不能执行任何操作（-），用于第二级对操作权限惊醒识别。

文件系统的性能

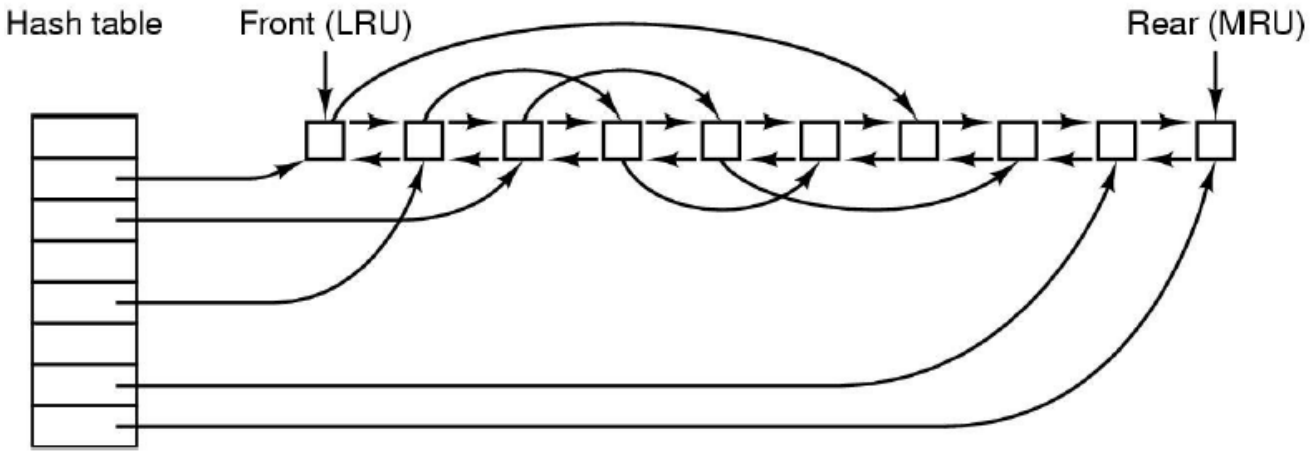
提升文件系统的性能，就需要尽可能少地访问磁盘。

磁盘高速缓存：内存中为磁盘块设置的一个缓冲区，保存了磁盘中某块的副本。需要考虑高速缓存的组织，置换问题，以及文件系统一致性问题。

> 当出现对某个特定块的I/O请求时，检查该块是否位于高速缓存中
>
> 若是，则直接进行读写
>
> 否则，先将数据从磁盘读入高速缓存，之后再拷贝到内存所需位置进行读写

散列组织：通过拉链Hash表快速查找某一块是否在高速缓存中。

LRU替换：由于高速缓存中块的数目通常不大（千级别），并且访问并不频繁，因此可以使用真实的LRU算法，使用一个链表来维护LRU结构。



一致性保证：除去数据块，其余会影响到文件系统一致性的某一块被修改，则立刻写回磁盘，不论其在LRU链表中的位置。数据块写回越频繁一致性越好，但性能越差；反之，性能越高但一致性越差。

UNIX的一致性保证：UNIX采用LRU链表+散列表的组织形式，并且提供一个SYNC系统调用，强制性写回所有被修改过的高速缓存中的数据块。后台update进程，每30秒调用一次sync写回所有修改过的数据块。因此UNIX即使系统崩溃，也不会丢失超过30s的数据。

Windows的一致性保证：组织形式类似，目前使用FlushFileBuffers系统调用，类似于UNIX的sync。历史上，采用通写策略，每一次高速缓存块修改都将直接写磁盘。

提前读取：依据空间局部性原理，每次访问磁盘时，多读入几个磁盘块。这一方法开销较小，针对顺序文件的情况，随机读取没有任何收益。

Windows文件访问：三种方式——不使用文件缓冲（普通方式），使用文件缓冲（预读取，写回时机需要考虑一致性问题），异步模式（CPU与I/O并发，不等待磁盘操作完成）。用户对磁盘的访问都是通过访问文件缓存实现的，由系统cache manager实现对缓存的控制，用户写磁盘数据时只更改cache内容，cache manager决定何时更新磁盘（一般是1s）。

合理分配磁盘空间：将有可能顺序存取的块，尽量分配在同一柱面上，减少磁臂移动次数。

使用i节点的文件系统，访问一个文件至少需要读盘两次，第一次获得i节点，第二次获得数据块。若i节点放在磁盘最外侧，那么第二次访问数据块的平均磁臂移动距离就是柱面数/2；若将i节点放在磁盘中部，那么第二次访问数据块的平均磁臂移动距离就只有柱面数/4，降为原先的一半。

磁盘调度：多个访盘请求在等待时，采用一定的策略，对请求的服务顺序调整安排，以降低平均磁盘服务时间，尽可能达到公平高效。

公平：一次I/O在有限时间内完成；高效：减少设备机械运动的时间开销。

访盘时间 = 寻道时间 + 旋转时间 + 传输时间

FCFS调度：按访问请求的先后顺序服务。简单公平，但磁头反复移动，效率低，设备损耗大。

最短寻道优先调度：选择距离当前磁头最近的访问进行服务。改善磁盘平均服务时间，会出现某些访问请求长时间得不到服务（饥饿）

扫描/电梯调度SCAN：无请求时，磁头不移动；当有请求时，磁头按一个方向移动，移动过程中对所有遇到的访问请求服务并判断该方向上是否还有请求；若没有请求则改变移动方向，为反向的经过的请求服务。折中策略，避免饥饿。

单向扫描调度C-SCAN：总是从0号柱面向里扫描，扫描过程中按柱面号顺序对请求进行服务；到达最内层煮面后，磁臂立刻带动磁头快速返回0号柱面，返回过程中不提供任何服务。减少了新情求的最大延迟。

多步扫描N-step-SCAN：当磁盘请求队列少于等于N时，按SCAN进行服务；当请求队列大于N时，每N个请求切分为一个子队列，按SCAN进行服务；再服务一个等待队列时，新的请求不允许加入正在服务的队列，而必须添加到其他队列中。N较大时，性能接近于SCAN，N=1时，退化为FCFS。克服磁头臂的粘性。

磁头臂的粘性：重复请求同一磁道，会垄断整个设备，造成其他请求的长时间等待。“将磁头粘在重复访问的磁道”。

双队列扫描FSCAN：两个子队列，相互对称。扫描开始时，一个对列为空，另一个队列装有所有请求；扫描过程中，新到达的请求放入另一个队列之中；一个队列扫描完成后则扫描另一个队列。对新请求的服务延迟到处理完所有的老请求之后，客服磁头臂的粘性。

旋转调度算法：根据延迟时间来决定执行次序的调度。若干请求访问同一磁头上的不同扇区，和若干请求访问不同磁头上的不同编号的扇区，总是让首先到达磁头位置下的扇区先进行传送；若干请求访问不同磁头上具有相同编号的扇区，各扇区同时到达，任意选择一个进行传送。

信息的分布优化：记录在磁道上的排列方式也会影响I/O操作时间。

记录成组：将若干逻辑记录合并为一组，并存放在一块。提高存储空间利用率，减少I/O次数，提高工作效率。eg. 目录文件。

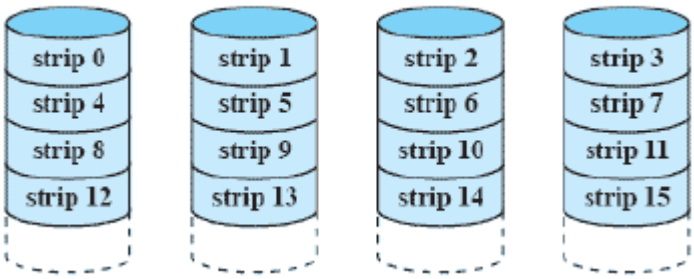
成组：将若干逻辑记录合并为一组并存放在一块的操作，一个物理块上存放的逻辑记录数量为**块因子**，逻辑记录不能跨块，因此块因子是正整数。

分解：从一个物理块上的一组逻辑记录中将某个逻辑记录分离出来的操作。

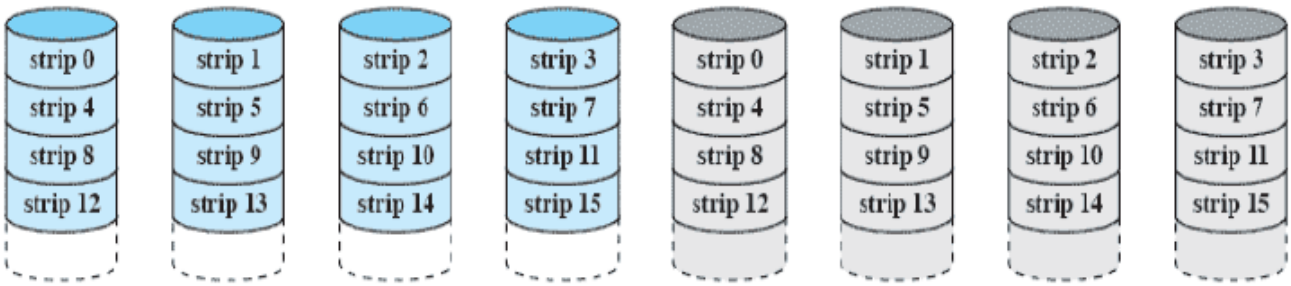
独立磁盘冗余阵列RAID技术：提高可靠性和磁盘性能，将多个磁盘按照一定的要求组合构成一个整体，OS将其视为一个独立的存储设备。

基本想法：将多个磁盘组织在一起作为一个逻辑卷提供磁盘跨越功能；通过将数据分为多个数据块，并行写入/读出以提高数据传输率（数据分条stripe）；通过镜像/校验提高容错能力（冗余）。

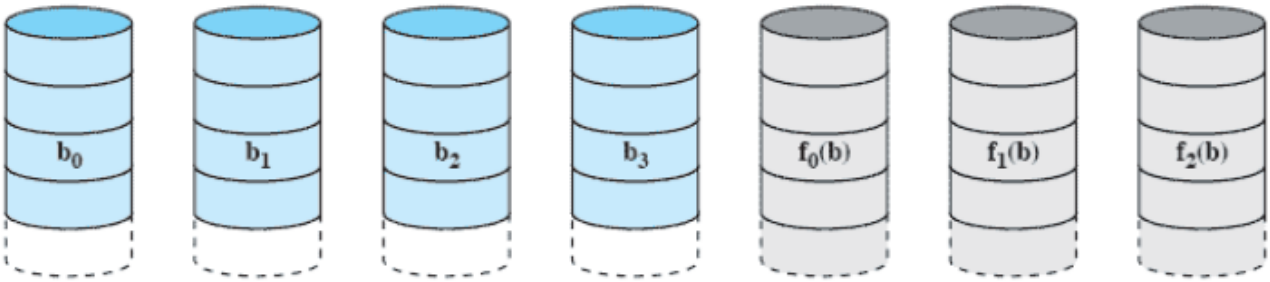
RAID0：条带化，数据分布在阵列的所有磁盘上。多个磁盘可以并行操作，充分利用总线带宽，使得数据吞吐提高，性能最佳；但没有冗余，没有错误校验和错误控制。



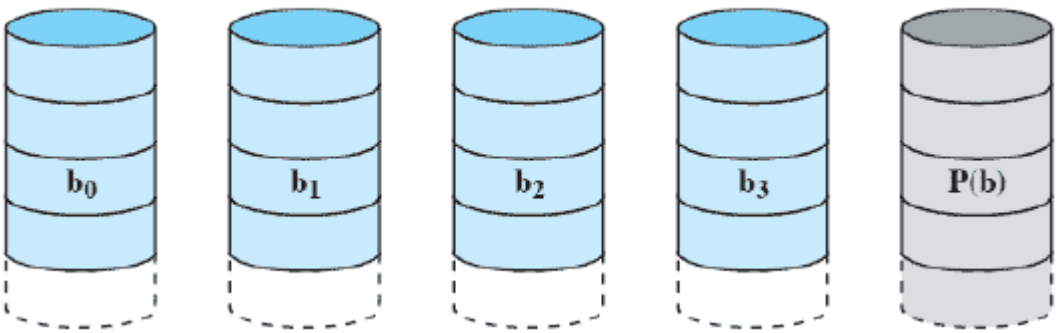
RAID1：镜像，所有数据同时存在于两个磁盘的相同位置，镜像存储。数据安全性最好，数据可恢复性最好；磁盘利用率仅有50%。



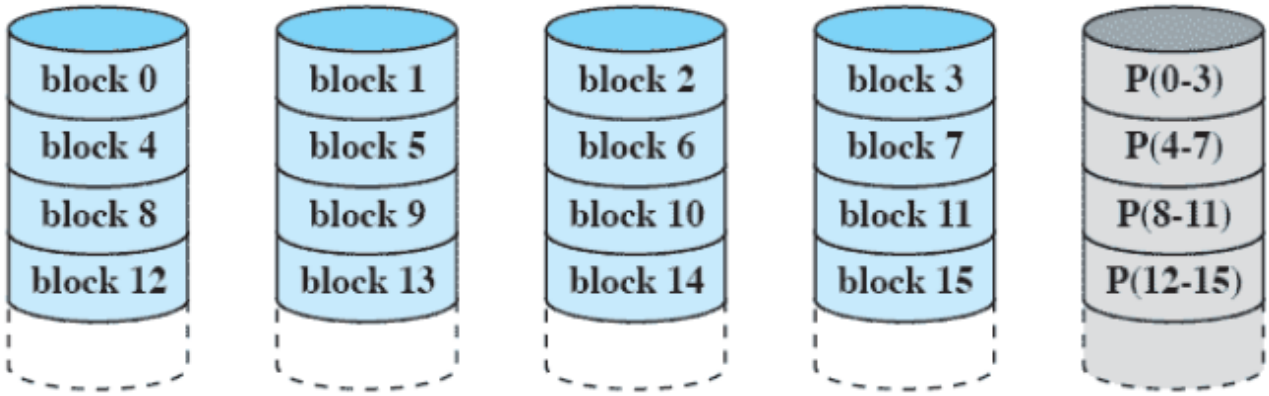
RAID2：并行访问，Hamming码校验。数据条带化（以byte或bit为单位）分布于不同磁盘，各个条带的Hamming码写入专用的磁盘的对应位置，存取数据时，整个磁盘阵列一起动作，在各个磁盘的相同位置并行存取。磁盘共轴同步平行存取，存取时间最优，Hamming码查错纠错性能良好；空间开销较大。



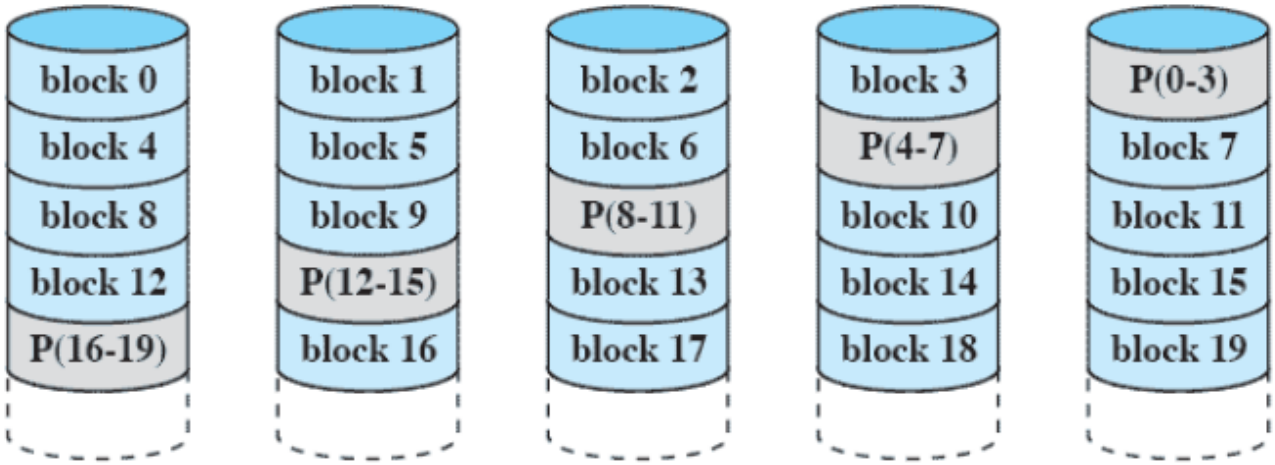
RAID3：交错位奇偶校验。类似RAID2，不使用Hamming码校验，而使用奇偶校验。数据以byte为单位拆分为条带，交叉写入数据盘。



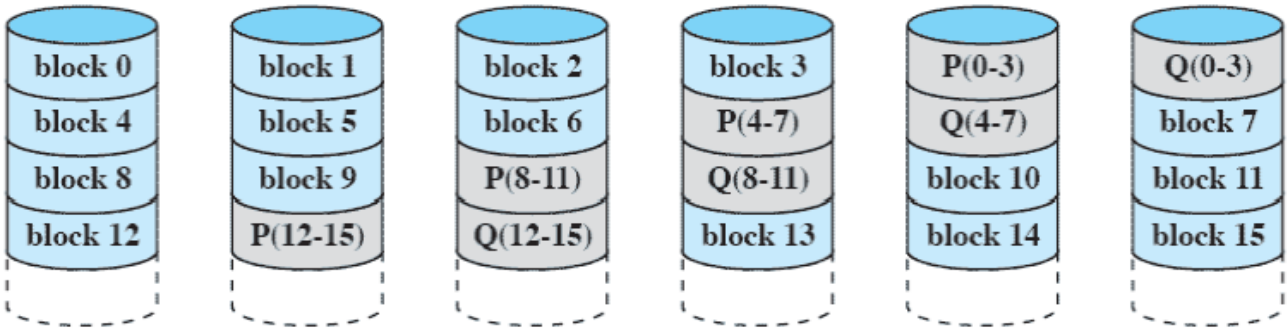
RAID4：交错块奇偶校验。类似RAID3，按数据块为单位进行拆分和校验。



RAID5：交错块分布式奇偶校验。类似RAID4，奇偶校验块交叉分布于各个磁盘。数据读出效率高，磁盘利用率好，可靠性高；数据写入效率一般，有写损失。



RAID6：交错块双重分布式奇偶校验。RAID5的基础上，设置两个校验码，交叉写入两个磁盘。数据恢复能力强；磁盘利用率降低，写能力降低。



RAID7：最优化异步高I/O速率和高数据传输率。自带操作系统和存储管理工具，是可以独立于主机运行的存储计算机；每个磁盘有独立I/O通道，与主通道相连；OS直接对每个磁盘的访问控制，让每个磁盘在不同的时段进行数据读写。性能最优；价格最高。

文件系统的结构设计

文件系统通用模型：应用程序文件系统接口<->逻辑文件系统层+文件组织模块层+基本文件系统层+基本I/O控制层<->物理磁盘

文件系统接口：定义一组使用和操作文件的方法。

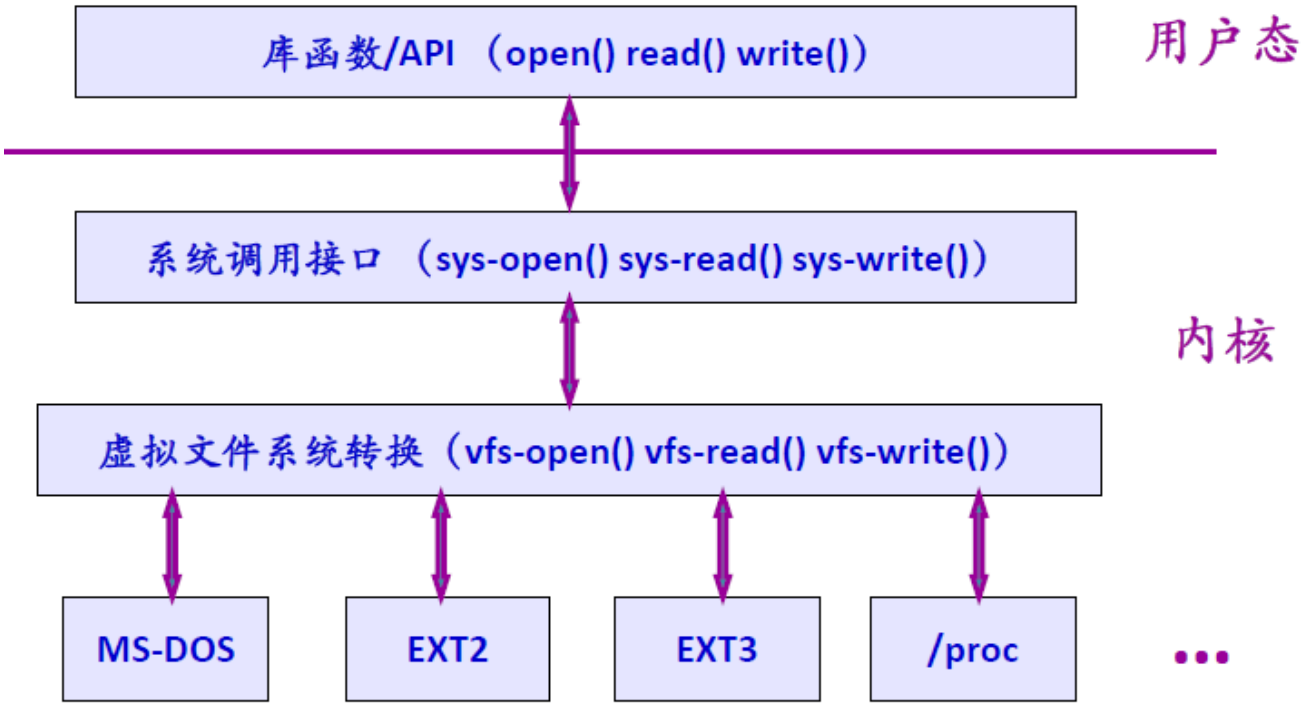
逻辑文件系统层：使用目录结构为文件组织提供需要的信息，并且负责文件的保护和安全。

文件组织模块层：负责对具体文件以及文件的逻辑块物理块进行操作。

基本文件系统层：向相应的设备驱动程序发出读写物理块的一般命令。

基本I/O控制层：由设备驱动和中断处理程序组成，实现内存和磁盘的信息传输。

虚拟文件系统：对多个不同的文件系统的抽象。提供相同的接口，管理所有文件系统关联的数据结构等。



日志结构文件系统LFS：提高磁盘写效率。将磁盘看作一个日志，每次写到末尾，避免寻找写的位置；集中零散的随机写操作，按段写入日志末尾；i节点和文件内容一起写入；为了统一管理分散的i节点，建立i节点表，存放在磁盘和高速缓存中。

清理线程：内存有限时，最终会被段填满，因此需要进行清理。清理线程检查i节点表，将需要清理的内容直接丢弃；要保留的内容先写入内存的待写入段中，然后回收；最终所有垃圾均被清理，所有要保留的内容规整地写入日志中。

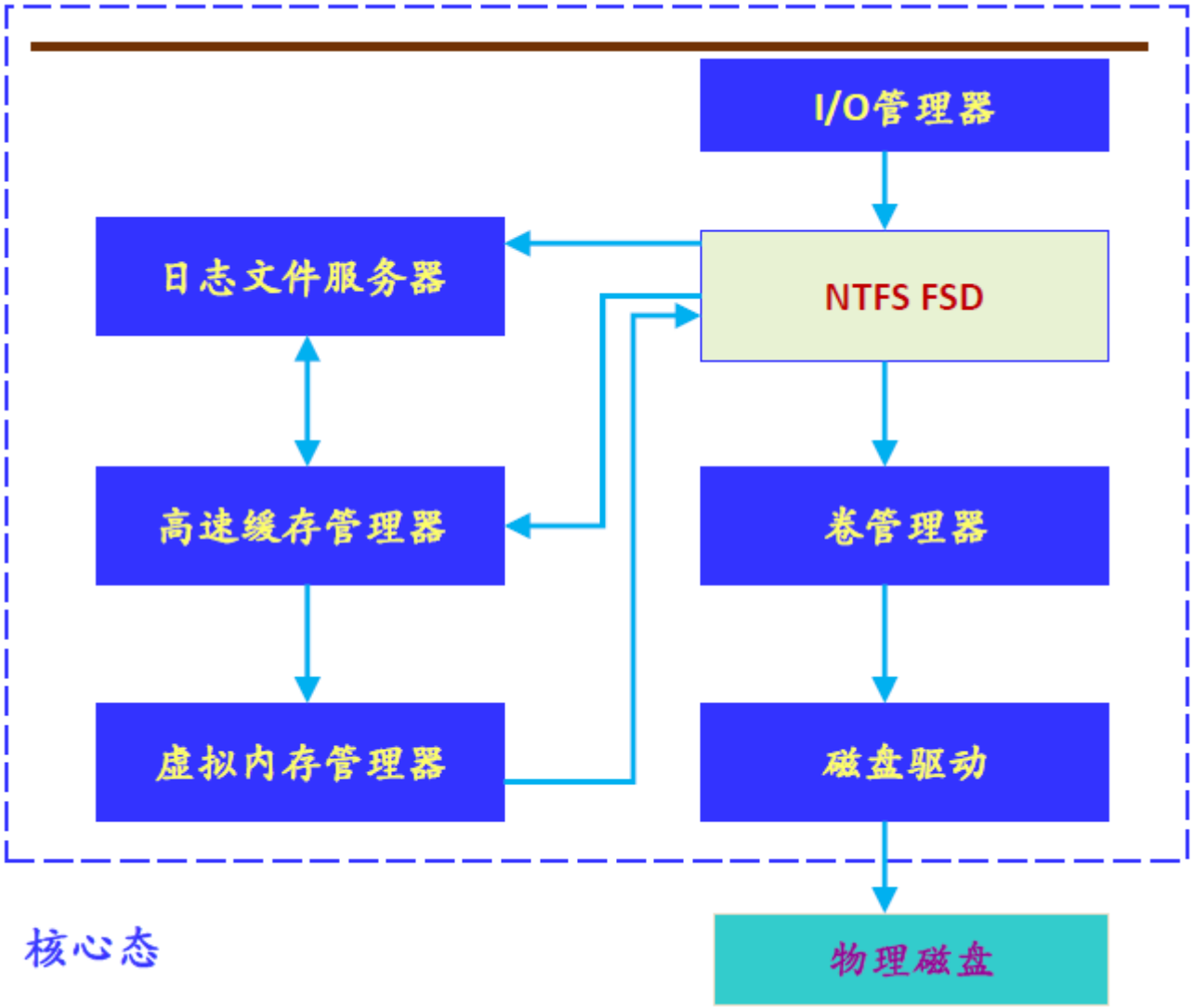
大量零碎随机写操作，LFS性能明显优于UNIX，读写大块数据，LFS性能不低于UNIX；鲁棒性好；与现有文件系统差异很大，管理复杂度高。

日志文件系统LFS：借鉴LFS的鲁棒性设计思路，保存一个记录下一步做什么的日志，在系统出错后，可以通过查看日志进行回复并继续完成未完成的操作。要求日志中记录的操作均为原子操作。

1. 写日志项
2. 将日志项写入磁盘日志
3. 执行操作
4. 执行完成后擦除日志项

NTFS文件系统 - Windows文件系统

FSD文件系统驱动程序：Windows所有文件系统相关的操作都需要通过FSD完成。包括显式文件I/O，高速缓存滞后写，高速缓存提前读，内存脏页写，内存缺页处理。



卷：建立在磁盘分区之上，一个磁盘可以有多个卷，一个卷也可以由多个磁盘组成。已格式化的卷上的数据分为元数据和用户数据。

簇：NTFS系统磁盘空间分配和回收的基本单位，簇的大小在格式化卷时确定。

LCN逻辑簇号：对整个卷中所有簇从头到尾进行简单编号。

VCN虚拟簇号：对属于特定文件的簇从头到尾进行编号，方便引用文件中的数据。

NTFS的卷最大为2^{64}字节。

NTFS文件组织：文件名称，主控文件表，文件记录，常驻属性和非常驻属性

文件名称：每个文件/目录名长度可以达到255 byte，可以包含unicode，多个空格和句点。每个文件由64位文件引用号唯一标识，文件引用号由48位文件号（标识该文件在MFT中的位置）和16位文件顺序号组成。

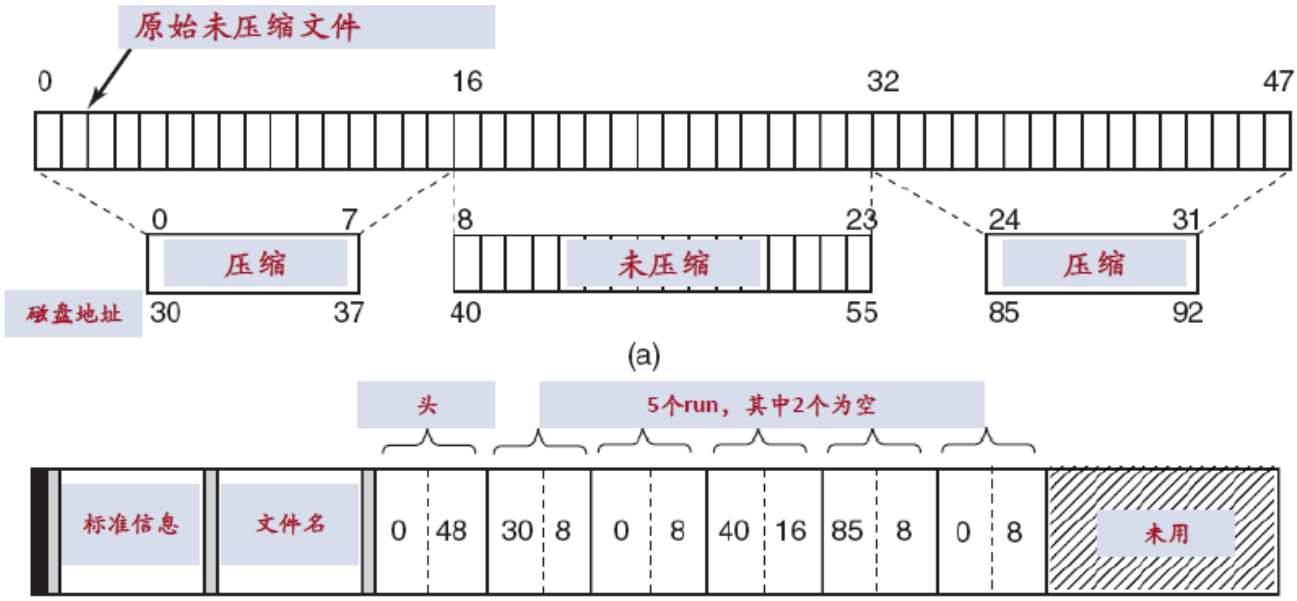
主控文件表MFT：NTFS卷结构核心，包含了卷中所有文件的信息，以文件记录数组的形式实现。MFT本身也是一个文件。访问文件时先从MFT中寻找相应的MFT项，再根据MFT项记录的信息找到文件。

文件记录：每个MFT项（文件记录）描述了卷中的一个文件/目录；卷中的每个文件/目录至少有一个MFT项。

属性：NTFS中的所有信息都是属性，每个属性由单个的流组成。NTFS文件是属性/属性值的集合，文件内容指的是"未命名数据属性流"。**常驻属性**：标准信息，文件名，索引根等。**非常驻属性**：存放在MFT之外的区域的run/extent。

数据压缩：NTFS的重要特征，可以对单个文件，整个目录或者整个目录树进行压缩；仅对用户数据压缩，不能压缩文件系统元数据；压缩可以减少磁盘空间，但会导致卷的性能下降。

压缩以16个簇为单位进行，NTFS决定这16个簇压缩后能否腾出一个簇，若能则只分配所需簇数，否则直接写入磁盘。每个run的VCN都按照16对齐，并且每个run不大于16。



可恢复性：通过日志记录实现，所有操作都被记录在日志文件中。

分布式文件系统

分布式计算机系统：由多台分散的计算机互联形成的计算机系统，资源、任务、功能和控制的全面分布，各个资源单元相互协作又高度自治，可以在全系统范围内实现资源管理，动态地进行资源分布和功能分配。

分布式文件系统：永久性存储和共享文件，允许用户直接存取文件而不需要讲他们复制到本地。

系统透明性：系统的内部实现细节，对于用户而言是隐藏的。

Hadoop分布式文件系统HDFS：分布式，部署硬件价格低廉，高容错性，高吞吐量，适用于超大数据集。

数据块存储：基本存储单元是数据块，默认大小64M，若文件大于数据块容量，将会被拆分，若小于也不会独占数据块。数据块为单位更便于数据备份。

流式读写：只支持文件末尾添加数据，不支持任意修改；并发读文件，不支持并发写文件。一次写入多次读取，吞吐量高。

数据备份：复制因子默认为3，三个副本均衡存放在本地节点，同机架（rack）其他节点，其他机架的节点。

3. 基本I/O管理

I/O管理概述

I/O性能经常会成为系统性能的瓶颈，并且I/O的资源多、杂且并发，会使得OS庞大而复杂。

设备控制器：CPU和I/O设备之间的接口，向CPU提供特殊指令和寄存器（控制寄存器，状态寄存器，数据寄存器）。

设备按数据特征分类：字符设备，块设备，网络设备

字符设备：以byte为单位进行存储和传输，传输速率低，不可寻址。eg. 键盘，鼠标，串口设备。get()和put()命令；使用文件访问接口。

块设备：以数据块为单位进行存储和传输，传输速率较高，可随机读写（可寻址）。eg. 磁盘，光驱，磁带。原始I/O命令或文件系统接口；内存映射文件访问。

网络设备：格式化报文交换。eg. 以太网，WIFI，蓝牙。send/receive网络报文；通过网络接口支持网络协议。

设备按资源分配分类：独占设备，共享设备，虚设备。

独占设备：一段时间内只能有一个进程使用的设备，一般是低速I/O设备。eg. 打印机，磁盘。

共享设备：一段时间内可以有多个进程以交叉的方式共同使用的设备，资源利用率高。eg. 硬盘。

虚设备：在一类设备上模拟另外一类设备，常用共享设备模拟独占设备，用高速设备模拟低速设备，被模拟的设备为虚设备。将慢速的独占设备改造成为多个用户可以共享的设备，从而提高设备的利用率。eg. SPOOLing技术用硬盘来模拟I/O设备。

I/O管理的任务：

按照用户请求，控制各个设备完成各种操作，完成I/O设备与内存之间的数据交换，最终完成用户I/O请求。需要分配和回收设备，执行设备驱动程序以实现真正的I/O，进行设备中断处理，以及管理I/O缓冲区。

建立方便的统一的独立于设备的接口。**方便性**：用户编程时不需要考虑设备的复杂的物理特性。**统一性**：对不同设备采取统一的操作，用户使用的是逻辑设备。

充分利用各种技术（通道，中断，缓冲，异步I/O）提高CPU与设备、设备与设备之间的秉性工作能力，充分利用资源，提高资源利用率。

保护，使得设备传送和数据管理时安全的，不被破坏的，保密的。

I/O硬件组成

机械部分：设备本身的物理装置

电子部分：**设备控制器/适配器**。完成（端口）地址译码；按照主机和设备之间的约定的格式和过程接受计算机发来的数据和控制信号，或向主机发送数据和状态信号；将计算机的数字信号转换为机械部分可以识别的模拟信号，或将模拟信号转换为数字信号；实现设备内部硬件缓冲、数据加工等提高性能或增强功能。

控制器的作用：设备接口。

OS将命令写入控制寄存器中，命令设备进行某种I/O动作，之后OS从状态寄存器获取状态信息，向/从数据寄存器/数据缓冲区写入/读取数据。

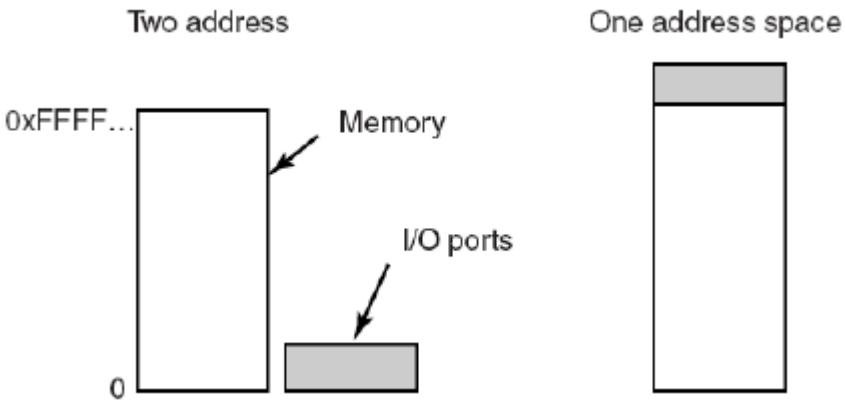
控制器接受一条命令后，便可以独立于CPU完成指定的动作，CPU同时执行其他的工作；命令完成后，控制器产生一个中断，CPU响应中断，OS通过读取状态寄存器信息获知设备状态和操作结果。

控制器将串行的位流转换为字节块，存入控制器内部的缓冲期中，进行必要的错误检查和修正后，再将其复制到内存中。

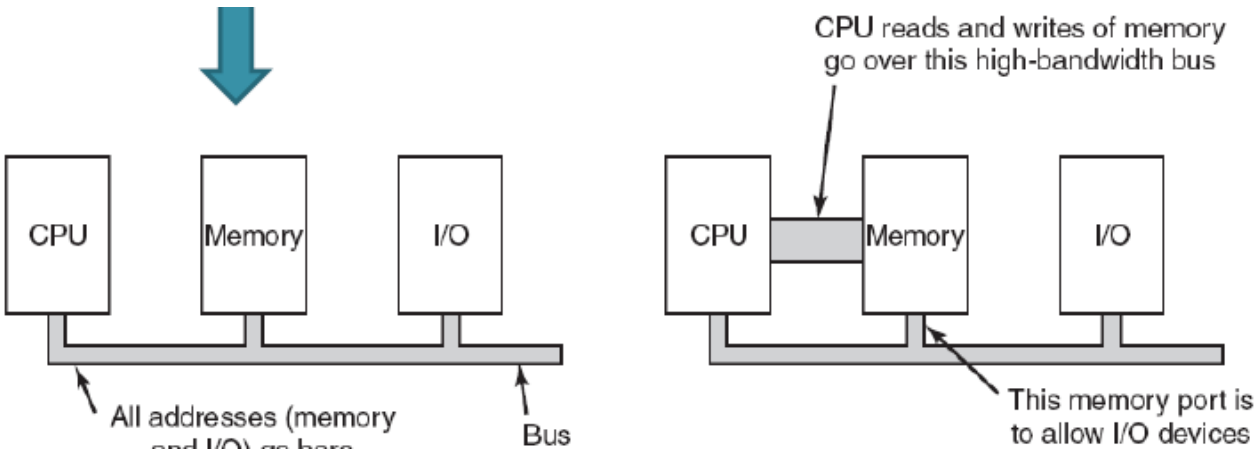
I/O端口地址：接口中的每个寄存器具有的惟一的地址，所有的I/O端口地址形成**I/O端口空间**，I/O端口空间受到OS保护。

I/O独立编址：使用I/O专用指令，I/O端口空间与内存空间独立且分离。外设不占用内存地址空间，编程时易于区分内存和I/O操作；I/O专用指令少，操作不灵活。eg. 8086 CPU分配64K的I/O端口空间，使用in和out进行读写。

内存映射编址：I/O端口空间与内存空间统一编址，将I/O端口视为内存单元，对I/O的读写等同于对内存的读写。不需要专门的I/O指令，I/O端口空间易于扩展，I/O操作可用指令多（内存操作指令均可用）；I/O端口空间占据内存空间。



内存映射编址不需要特殊的保护机制来组织用户进程执行I/O，OS不将包含控制寄存器的部分的地址空间放入任何用户的虚拟地址空间即可；任何引用内存的指令都可以直接引用控制寄存器。然而内存映射编址的情形下，设备寄存器不可以被Cache，因而每个页面都需要具有禁用Cache的选项位，OS据此选择是否进行Cache，使得硬件和OS的复杂度上升；内存映射编址中所有设备和内存模块都需要检查内存引用是不是自己，若计算机为单一总线则没有任何问题，但如果CPU和主存间具有高速总线，则会出现I/O设备无法向内存一样快速进行访问的问题。



I/O控制方式：可编程I/O（轮询），中断驱动I/O，DMA方式。

可编程I/O：CPU代表进程向I/O模块发出I/O命令，之后进入忙等待，直到操作完成后才继续执行。

中断驱动I/O：CPU向I/O设备发出命令，之后进行其他操作，当I/O设备准备好数据后产生I/O中断，CPU接受中断后将数据传送到内存，完成一次I/O操作。

DMA方式：DMAC可以独立于CPU获得总线，从而完成I/O设备到内存之间的数据传送工作。CPU配置DMAC，设置内存传送起点，传送长度等控制寄存器后，CPU进行其他操作；DMAC控制I/O设备和内存之间的数据传送，当传送都完成后，产生中断提示CPU；CPU接受中断，继续其I/O后的工作。

	无中断	使用中断
CPU实现I/O-Mem传送	可编程I/O	中断驱动I/O
I/O-Mem直接传送	-	DMA

I/O软件组成

分层设计思想：将I/O软件组织成多个层次；每层都执行OS所需功能的一个相关子集，该层功能依赖于低层的更原始的功能（隐藏低层功能细节），向高层提供服务；较低层考虑硬件特性，较高层不依赖于硬件，而是像用户提供一个友好清晰，简单但功能强大的接口。

I/O软件层次：用户级I/O软件，设备无关I/O软件，设备驱动程序，中断处理程序。

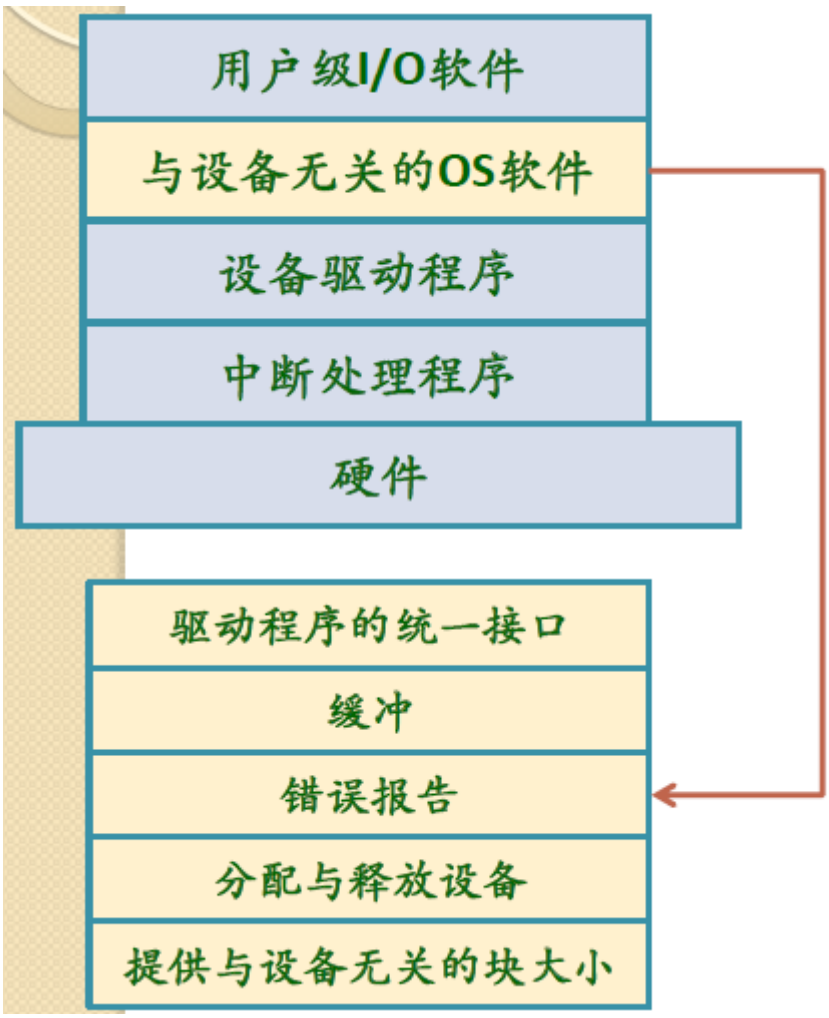
用户进程层：执行输入输出系统调用，对I/O数据格式化，为假脱机I/O作准备。

独立于设备的I/O软件：实现设备命名，设备保护，成块处理，缓冲技术以及设备分配和回收。

设备驱动程序：设置设备寄存器，检查设备执行状态。

中断处理程序：负责I/O完成时唤醒设备驱动程序进程，进行中断处理。

硬件层：实现物理I/O操作。



设备独立性/设备无关性：用户编写的程序可以访问任意的I/O设备，无需实现指定设备。设备分配灵活，易于I/O重定向。

用户角度：用户编程时使用逻辑设备名，OS实现逻辑设备到物理设备的转换，并实现I/O操作。

系统角度：设计并实现I/O软件时，除了直接与设备打交道的低层，其余部分不依赖于硬件。

I/O相关技术

缓冲技术：解决CPU与I/O设备速度不匹配的问题；提高CPU与I/O设备的并行性；减少I/O设备对CPU的中断请求数目，放宽CPU对中断响应时间的要求。

硬缓冲：硬件寄存器实现，设备中设置好的缓冲区。

软缓冲：在内存中开辟一个空间，用作缓冲区。

单缓冲：一个连续的缓冲区，一方（CPU或I/O设备）填充满后，只有另一方处理完毕才可以继续填充。

双缓冲：两个对称的缓冲区，一方填充一个缓冲区时，另一方可以处理另一个被填充满的缓冲区，双方工作都完成后交换缓冲区。

缓冲池/多缓冲：统一管理多个缓冲区，采取有界缓冲区的生产者-消费者模型对缓冲池中的缓冲区进行循环使用。

UNIX System V缓冲技术：缓冲池技术；结合提前读和延迟写技术，加速重复性I/O和阵发性I/O，尽可能少地减少磁盘I/O。

缓冲池：200个缓冲区，512/1024 byte。

缓冲区：缓冲控制块/缓冲首部+缓冲数据区。OS通过缓冲控制块来实现对缓冲区的管理。

av链：空闲缓冲区队列，队列头部为bfreelist。

b链：设备缓冲队列，连接所有分配给各类设备的缓冲区。采用散列多链表的方式组织，通过设备号等进行散列。

逻辑设备号和盘块号：标志出文件系统和数据所在的盘块号，时缓冲区的唯一标志。

状态项：指明缓冲区状态，空闲/占用，上锁/开锁，延迟写/立即写...

av指针和b指针：分配管理缓冲池。

每个缓冲区开始时（未被使用时）位于av链；开始I/O请求后位于设备I/O请求队列和b链；I/O完成后位于av链和设备b链。

> 缓冲区的近似LRU分配：
> 进程想从指定盘块读取数据时，OS根据盘块号在b链中查找
> 　　若找到缓冲区，则将该缓冲区从av链取下，完成从缓冲区到内存用户数据区的传送
> 　　否则在b链中没有找到，从av链首摘取一个缓冲区，插入设备I/O请求队列，并将该块从原来的b链位置取下（若原来在b链中的话），插入到新的b链位置。当数据从磁盘块读入到缓冲区后，从I/O请求队列取下该缓冲区；当OS完成缓冲区到用户数据区后，释放缓冲区，加入av链尾。
> 数据读入缓冲区并传送到内存后，该缓冲区保留在原b链位置，即该设备保留其缓冲区；只有当该设备长期未使用缓冲区时，缓冲区慢慢移动到av链首，被其他某个设备夺走，重新分配后就得盘块数据才被置换。

设备管理有关的数据结构

设备表/部件控制块：OS为每个部件，每个设备设置一张表格，描述设备的类型、标识符、状态、当前使用者的PID等。

同类资源队列：OS为了方便对I/O设备的管理，在设备表的基础上通过指针将相同物理属性的设备连成队列。

I/O请求包：每当进程发出I/O请求，OS建立一张表格，将此次I/O请求的参数、系统缓冲区地址等填入表格。I/O完成后I/O请求包失效而被删除。

I/O队列：请求包队列。

独占设备的分配策略：独占设备一旦被进程占用，就不再运去其他进程申请使用直至被释放。分配独占设备，需要考虑效率问题，也需要避免死锁。

静态分配：进程运行前，分配设备；运行结束后，收回设备。设备利用率很低。

动态分配：进程运行过程中，当用户提出设备请求时进行分配，一旦停止使用立即收回。效率高，但不好的分配策略会导致思索。

分时式共享设备的分配策略：采用队列方式，不同进程的I/O操作以队列方式分时使用设备。分时单位为一次I/O用时。

I/O设备驱动：每个驱动处理一种设备类型，其接收来自与设备无关的上层软件的抽象请求，并执行这个请求。驱动程序负责向控制器释放命令（控制字），并监督它们正确执行。

驱动程序释放命令后，一般情况下执行该驱动的进程必须等待命令完成，因而当命令开始执行后，它阻塞自己直至I/O完成后中断处理程序解除阻塞；其他情况下，命令执行不必延迟就可以很快地完成。

驱动接口：与操作系统的接口，与系统引导的接口（初始化），与设备的接口。

驱动接口函数：初始化函数（向OS登记接口函数等，OS启动或加载驱动时执行），卸载函数，申请设备函数，释放设备函数，I/O操作函数（独占设备，包含启动I/O的指令；共享设备，将I/O请求包为请求包排至请求队列），中断处理函数（I/O完成后的善后处理，唤醒等待I/O的被阻塞的进程，使之完成后续工作，以及启动请求队列的下一个I/O请求）。

I/O进程：专门处理系统中的I/O请求和I/O中断工作。

I/O请求进入：用户程序调用send向I/O进程发送I/O请求，之后调用block将自己阻塞，直至I/O任务完成；OS利用wakeup唤醒I/O进程，完成用户要求的I/O处理。

I/O中断进入：I/O中断发生时，内核的中断处理程序发送消息给I/O进程，I/O进程负责判断和处理中断。

```
I/O进程：最高优先级系统进程，快速抢占CPU并运行
关闭终端，调用receive接受信息
    若没有信息，则开中断，阻塞自己
    否则有消息，判断消息类型
        若为I/O请求，则准备通道程序，发送启动I/O指令，继续判断有无消息
        若为I/O中断，判断正常/异常结束
            正常结束：唤醒请求I/O操作的进程进行后续工作
            异常结束：转入相应的错误处理程序
```

I/O性能问题

目标是CPU尽可能拜托I/O，或至少CPU利用率尽可能不被I/O拉低。

缓冲技术：缓解CPU和I/O设备的速度差异

异步I/O：使得CPU不用等待I/O

DMA & 通道 & I/O处理机：使得CPU摆脱I/O数据传送操作

同步I/O：I/O处理过程中，CPU空闲等待；CPU处理数据时，I/O无法同时进行。

异步I/O：应用程序可以启动一个I/O，然后在I/O请求执行的同时继续进行处理，尽量填充I/O操作间的等待CPU的时间。

系统实现的异步I/O：切换至其他线程运行以保证CPU利用率；但对少量数据的I/O操作会使得切换开销很大

用户实现的异步I/O：发出I/O指令后，不阻塞自己，而是进行其他操作；当需要使用I/O数据或结果时，再阻塞自己等待其完成。可以尽量填充I/O等待时间，但有时无法避免阻塞；不引入线程切换，系统开销少。

4. 死锁问题

KEY WORDS：死锁，活锁，饥饿，死锁预防，死锁避免，死锁检测，死锁解除，资源有序分配法，银行家算法，安全状态，资源分配图，哲学家就餐问题

死锁的基本概念

经典哲学家就餐问题：五个哲学家，围坐一个圆桌，桌中央一盘通心粉，每人面前一只空盘子，每两人之间一只叉子；哲学家的行为是思考，感到饥饿，和吃通心粉；为了吃到通心粉，每个哲学家必须拿到两只叉子，并且每个人只能取自己左右的叉子。要求筷子互斥使用，不能出现死锁，不能出现某个哲学家饥饿。

死锁：一组进程中每个进程都无限等待被该组中另一个进程占有的资源，因此永远无法得到资源的现象。这一组进程成为死锁进程。

死锁是由于资源数量有限，锁和信号量错误使用造成的。

资源使用方式：**“申请-分配-使用-释放”模式**

可重用资源：可以被多个进程多次使用的资源。eg. CPU，I/O设备，内存，文件...

可消耗资源：只可使用一次的可以创建和销毁的资源。eg. 信号，中断，消息...

活锁：加锁，轮询，没有阻塞但也没有进展。

饥饿：看起来合理的资源分配策略，使得某些进程在某种情况下永远得不到服务。

死锁的必要条件：互斥使用；占有且等待；不可抢占，不可剥夺；循环等待。

互斥使用：每个资源每次只能给一个进程使用。

占有且等待：一个进程在申请新资源的同时保持对原有资源的占有。

不可抢占：自愿申请者不能强行从占有者手中夺取资源，资源只能由占有者资源释放。

循环等待：存在一个进程等待队列$\{P_1, P_2, \cdots, P_n\}$，其中$P_i$等待$P_{i+1}$，$i = 1, 2, \cdots, n-1$，$P_n$等待$P_1$，形成进程等待环。

资源分配图RAG

RAG：用有向图描述资源和进程的状态。结点由进程和资源两部分组成，边由资源结点指向进程节点，或由进程节点指向资源节点。

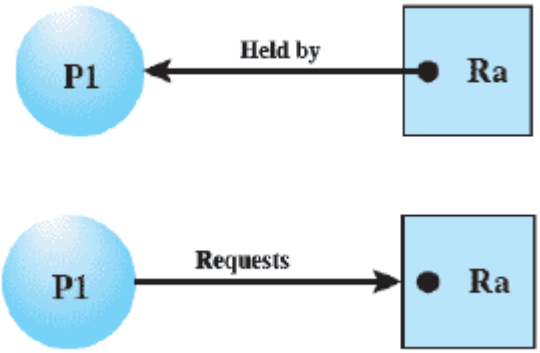
$$
\begin{aligned}
&G =< V, E > \\
&V = P \cap R,\ P = \{P_1, P_2, \cdots P_n\},\ R = \{R_1, R_2, \cdots, R_m\} \\
&E = \{(P_i, R_j)\ or\ (R_j, P_i),\ i = 1, 2, \cdots, n;\ j = 1, 2, \cdots, m\}
\end{aligned}
$$

资源类：系统由若干类资源构成，每一类资源为一个资源类，每个资源类中的若干同种资源，为**资源实例**。

方框表示资源类，方框中的黑点表示资源实例，圆圈表示进程。

分配边：资源实例->进程，一条有向边。

申请边：进程->资源类，一条有向边。



死锁定理：如果资源分配图中没有环路，则系统中没有死锁；如果图中存在环路，则系统中可能存在死锁。如果每个资源类中只包含一个资源实例，则环路是死锁存在的充分必要条件。

RAG化简算法

```
1. 找非孤立点且只有分配边的进程节点，去除分配边，使其变为孤立节点
2. 将相应资源分配给一个等待的进程，即将进程的申请边变为分配边
3. 重复1,2
```

进程依赖图：去除RAG中的资源类和资源实例，结点仅有进程，有向边$< P_i, P_j >$表示P_i依赖于P_j的资源，即P_i要等待P_j释放资源，才能运行。

解决死锁

鸵鸟算法：假装事情没有发生，不考虑这一问题。

让死锁发生：**死锁检测与解除**

不让死锁发生：**死锁预防**（静态策略，设计合适资源分配算法）；**死锁避免**（动态策略，跟踪并评估资源分配过程，根据评估结果决策是否分配）

死锁预防：在设计系统时，通过确定资源资源分配算法，排除发生死锁的可能性。具体做法是预防产生死锁的四个条件中的任何一个。

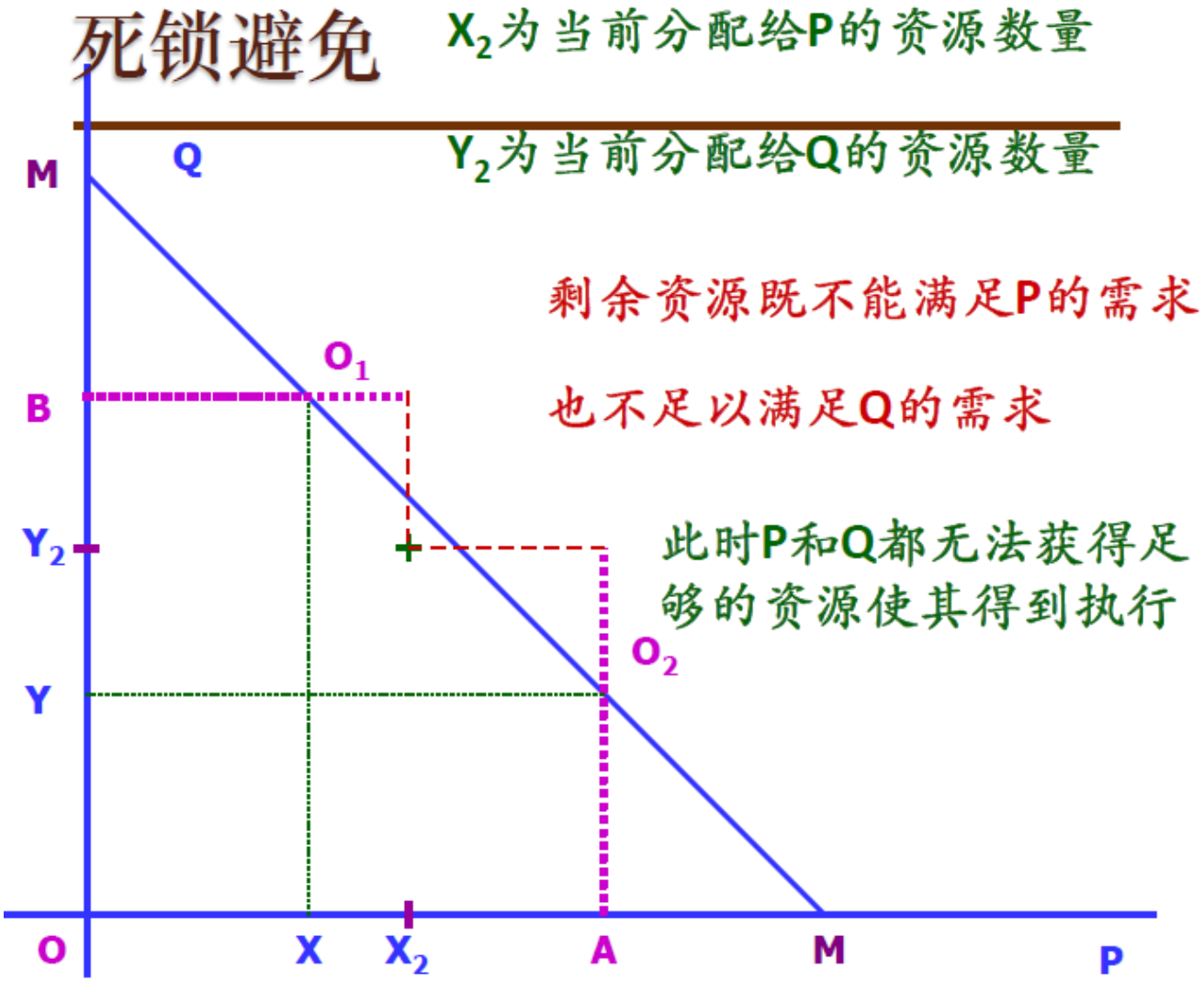
破坏互斥使用条件：将独占资源变为共享资源。eg. SPOOLing技术，解决不允许任何进程直接占有打印机的问题，设计"精灵daemon"进程负责管理打印机，任何请求都发送给daemon，由它完成打印任务。

破坏占有且等待条件：要求每个进程在运行前一次性申请要用的所有资源，仅当所有资源均可满足时才一次性分配（资源利用率很低，会有饥饿现象）；允许动态申请资源，但一个进程在申请新的资源不能立即得到满足而变为等待之前，释放所有已经占有的资源，需要时再次申请。

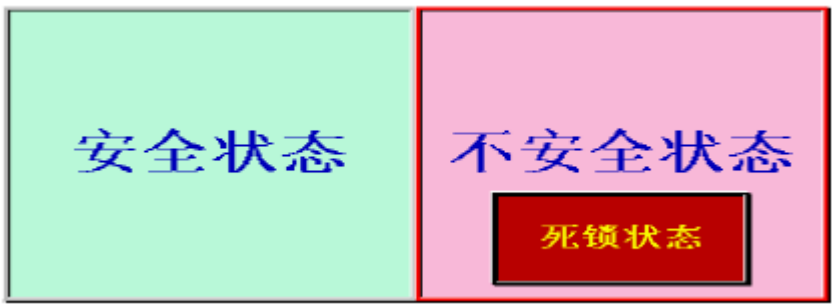
破坏不可抢占条件：虚拟化资源，当进程申请的资源被其它进程占用时，可以通过操作系统抢占这一资源。仅适用于状态易于保存和恢复的资源。

破坏循环等待条件：资源有序分配法，将系统中所有资源编号，进程在申请资源时必须严格按照资源编号的递增次序进行，否则OS不予分配。变体策略为禁止进程申请编号低于持有资源编号的资源。

死锁避免：在系统运行过程中，对进程发出的每个系统能够满足的资源申请进行动态检查，并且根据检查结果决定是否分配，若分配后有可能或确定会发生死锁，则不予分配，否则可以分配。



安全序列：一个进程序列$\{P_1, \cdots, P_n\}$是安全的，若对于每一个进程$P_i(1 \leq i \leq n)$，它以后尚需要的资源量不超过系统当前剩余的资源量与所有进程P_j $(j < i)$当前占有的资源量之和，则系统处于安全状态。安全状态下一定没有死锁发生，不安全状态一定会导致死锁。



银行家算法：Dijkstra提出。

系统具有的特征：**固定数量**的进程共享固定数量的资源；每个进程预先指定完成工作所需的**最大资源数量**；进程不能申请比系统中可用资源总数还多的资源；进程等待资源的时间**有限**；若系统满足进程对资源的最大需求，那么进程应该在**有限**时间内使用资源，之后归还给系统。

银行家算法：n进程总数，m资源总数；Available[m]可用资源计数；Max[n,m]进程对资源的最大需求量；Allocation[n,m]进程分得的资源量；Need[n,m]进程在最大需求量约束下还可以获得的资源量；Request[n,m]进程申请的资源量
进程pi提出申请时：
1. 若Request[i]<=Need[i]，则转至第2步；否则出错，返回
2. 若Request[i]<=Available，则转至第3步；否则进程等待
3. 若系统允许资源分配，则有新状态如下：
    Available = Available - Request[i]
    Allocation[i] = Allocation[i] + Request[i]
    Need[i] = Need[i] - Request[i]
4. 若系统新状态安全，则分配完成；否则新状态不安全，恢复原状态，进程等待

安全性检查：Work[m]系统可以提供给进程继续运行所需的各类资源;Finish[n]系统是否具有足够的资源分配给进程使之运行完成。
1. 初始化
    Work = Available
    Finish = false
2. 寻找i，使得如下条件满足；若存在转至3，否则不存在，转至4
    Finish[i] = false
    Need[i] <= Work
3. 更新后转至2
    Work = Work + Allocation[i]
    Finish[i] = true
4. 若对所有i，均有Finish[i] = true，则系统处于安全状态，否则处于不安全状态

死锁检测：允许死锁发生，OS不断检视系统状态，判断死锁是否发生；当判断死锁发生后，立刻采取专门措施解除死锁，并以最小代价恢复系统运行。

检测时机：当进程由于资源请求不满足而等待时检测（系统开销大）；定时检测；资源利用率降低时检测。

简单的检测死锁方法：每个进程和资源唯一编号，设置资源分配表和进程等代表，若在两表上出现环，则出现了死锁。

死锁解除：以**最小代价**恢复OS运行。撤销所有死锁进程；或进程回退再启动；或按某种选择原则逐一撤销死锁进程，直到满足某一条件；或按照某种原则OS逐一抢占资源，资源被OS抢占的进程回退到之前的状态，直到满足某一条件。