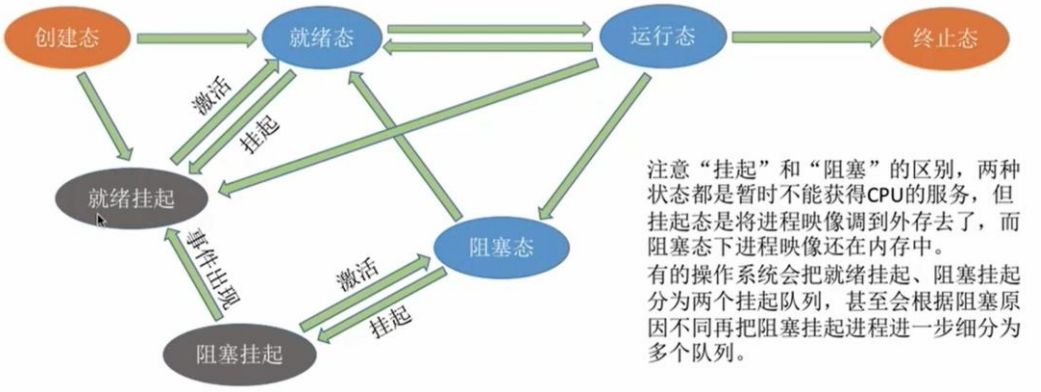
答：“用户态→核心态”是通过中断实现的。并且中断是唯一途径。
“核心态→用户态”的切换是通过执行一个特权指令，将程序状态字（PSW）的标志位设置为“用户态”

由于内存空间有限，有时无法将用户提交的作业全部放入内存，因此就需要确定某种规则来决定将作业调入内存的顺序。
高级调度（作业调度）。按一定的原则从外存上处于后备队列的作业中挑选一个（或多个）作业，给他们分配内存等必要资源，并建立相应的进程（建立PCB），以使它（们）获得竞争处理机的权利。
高级调度是辅存（外存）与内存之间的调度。每个作业只调入一次，调出一次。作业调入时会建立相应的PCB，作业调出时才撤销PCB。高级调度主要是指调入的问题，因为只有调入的时机需要操作系统来确定，但调出的时机必然是作业运行结束才调出。

引入了虚拟存储技术之后，可将暂时不能运行的进程调至外存等待。等它重新具备了运行条件且内存又稍有空闲时，再重新调入内存。
这么做的目的是为了提高内存利用率和系统吞吐量。
暂时调到外存等待的进程状态为挂起状态。值得注意的是，PCB并不会一起调到外存，而是会常驻内存。PCB中会记录进程数据在外存中的存放位置，进程状态等信息，操作系统通过内存中的PCB来保持对各个进程的监控、管理。被挂起的进程PCB会被放到的挂起队列中。

中级调度（内存调度），就是要决定将哪个处于挂起状态的进程重新调入内存。
一个进程可能会被多次调出、调入内存，因此中级调度发生的频率要比高级调度更高。



注意“挂起”和“阻塞”的区别，两种状态都是暂时不能获得CPU的服务，但挂起态是将进程映像调到外存去了，而阻塞态下进程映像还在内存中。
有的操作系统会把就绪挂起、阻塞挂起分为两个挂起队列，甚至会根据阻塞原因不同再把阻塞挂起进程进一步细分为多个队列。

不能进行进程调度与切换的情况
1. 在处理中断的过程中。中断处理过程复杂，与硬件密切相关，很难做到在中断处理过程中进行进程切换。
2. 进程在操作系统内核程序临界区中。
3. 在原子操作过程中（原语）。原子操作不可中断，要一气呵成（如之前讲过的修改PCB中进程状态标志，并把PCB放到相应队列）

$$\text{带权周转时间} = \frac{\text{作业周转时间}}{\text{作业实际运行的时间}} = \frac{\text{作业完成时间} - \text{作业提交时间}}{\text{作业实际运行的时间}}$$

带权周转时间必然 ≥ 1

带权周转时间与周转时间都是越小越好

$$\text{平均带权周转时间} = \frac{\text{各作业带权周转时间之和}}{\text{作业数}}$$

对于进程来说，等待时间就是指进程建立后等待被服务的时间之和，在等待I/O完成的期间其实进程也是在被服务的，所以不计入等待时间。
对于作业来说，不仅要考虑建立进程后的等待时间，还要加上作业在外存后备队列中等待的时间。

最短剩余时间优先算法：每当有进程加入就绪队列改变时就需要调度，如果新到达的进程剩余时间比当前运行的进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列。另外，当一个进程完成时也需要调度

2. 很多书上都会说“SJF 调度算法的平均等待时间、平均周转时间最少”
严格来说，这个表述是错误的，不严谨的。之前的例子表明，最短剩余时间优先算法得到的平均等待时间、平均周转时间还要更少
应该加上一个条件“在所有进程同时可运行时，采用SJF调度算法的平均等待时间、平均周转时间最少”；
或者说“在所有进程都几乎同时到达时，采用SJF调度算法的平均等待时间、平均周转时间最少”；

高响应比优先（HRRN, Highest Response Ratio Next）

中国大学MOOC

HRRN

算法思想	要综合考虑作业/进程的等待时间和要求服务的时间
算法规则	在每次调度时先计算各个作业/进程的响应比，选择响应比最高的作业/进程为其服务
用于作业/进程调度	即可用于作业调度，也可用于进程调度
是否可抢占？	非抢占式的算法。因此只有当前运行的作业/进程主动放弃处理机时，才需要调度，才需要计算响应比
优缺点	综合考虑了等待时间和运行时间（要求服务时间） 等待时间相同时，要求服务时间短的优先（SJF 的优点） 要求服务时间相同时，等待时间长的优先（FCFS 的优点） 对于长作业来说，随着等待时间越来越久，其响应比也会越来越大，从而避免了长作业饥饿的问题
是否会导致饥饿	不会

$$\text{响应比} = \frac{\text{等待时间+要求服务时间}}{\text{要求服务时间}}$$

响应比 ≥ 1

多级反馈队列调度算法

中国大学MOOC

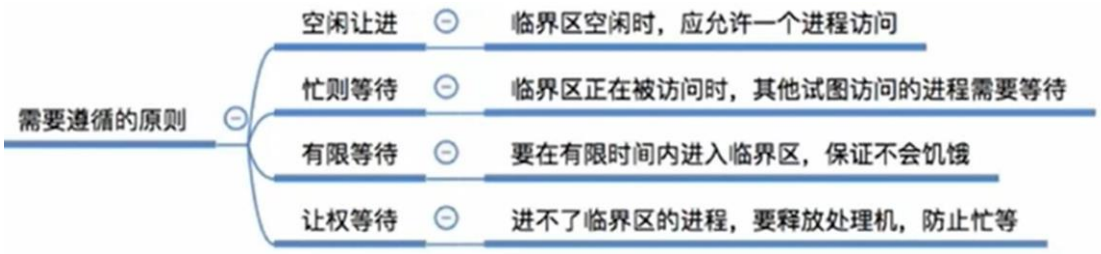
算法思想	对其他调度算法的折中权衡
算法规则	1. 设置多级就绪队列，各级队列优先级从高到低，时间片从小到大 2. 新进程到达时先进入第1级队列，按FCFS原则排队等待被分配时间片，若用完时间片进程还未结束，则进程进入下一级队列队尾。如果此时已经是在最下级的队列，则重新放回该队列队尾 3. 只有第 k 级队列为空时，才会为 k+1 级队头的进程分配时间片
用于作业/进程调度	用于进程调度
是否可抢占？	抢占式的算法。在 k 级队列的进程运行过程中，若更上级的队列（1~k-1级）中进入了一个新进程，则由于新进程处于优先级更高的队列中，因此新进程会抢占处理机，原来运行的进程放回 k 级队列队尾。

知识回顾与重要考点

中国大学MOOC

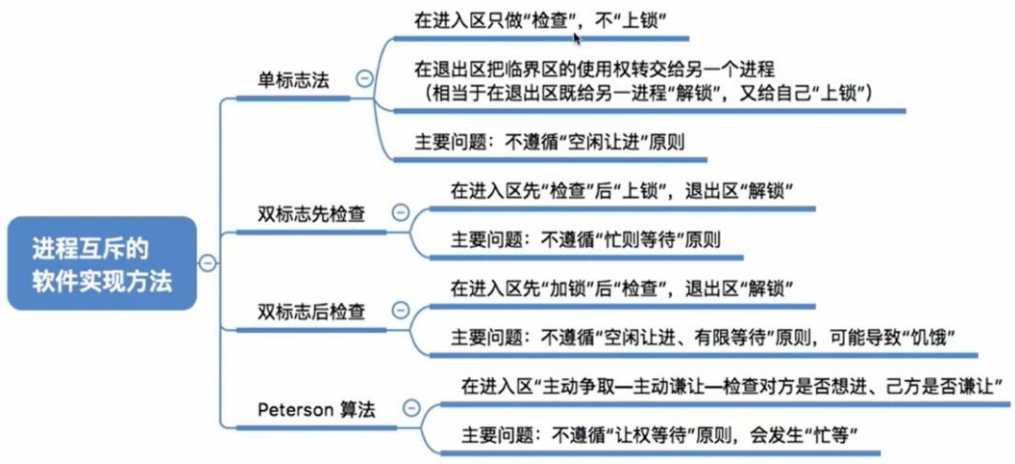
算法	思想&规则	可抢占?	优点	缺点	会导致饥饿?	补充
时间片轮转		抢占式	公平，适用于分时系统	频繁切换有开销，不区分优先级	不会	时间片太大或太小有何影响?
优先级调度		有抢占式的，也有非抢占式的。注意做题时的区别	区分优先级，适用于实时系统	可能导致饥饿	会	动态/静态优先级。各类型进程如何设置优先级？如何调整优先级？
多级反馈队列	较复杂，注意理解	抢占式	平衡优秀666	一般不说它有缺点，不过可能导致饥饿	会	

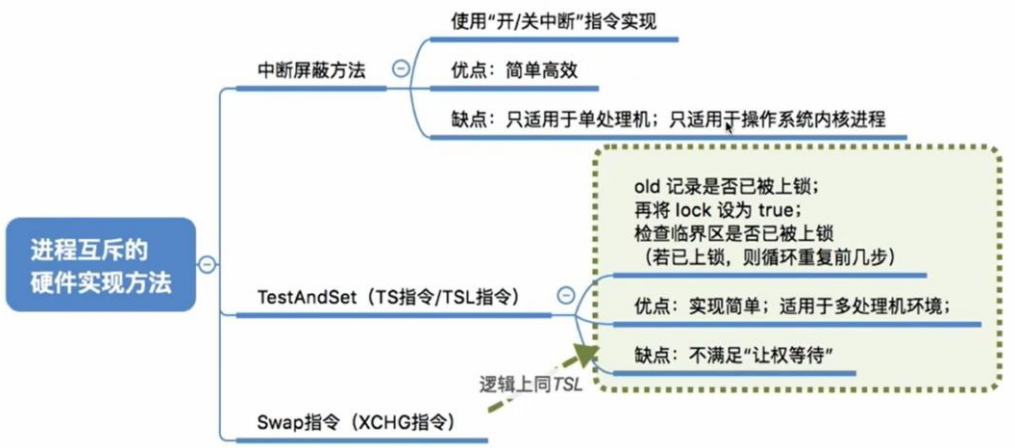
注：比起早期的批处理操作系统来说，由于计算机造价大幅降低，因此之后出现的交互式操作系统（包括分时操作系统、实时操作系统等）更注重系统的响应时间、公平性、平衡性等指标。而这几种算法恰好也能较好地满足交互式系统的需求。因此这三种算法适合用于交互式系统。（比如UNIX使用的就是多级反馈队列调度算法）




```
bool flag[2];          //表示进入临界区意愿的数组，初始值都是false
int turn = 0;          //turn 表示优先让哪个进程进入临界区
P0 进程:
flag[0] = true;                    ①
turn = 1;                          ②   进入区
while (flag[1] && turn==1);        ③
critical section;                  ④
flag[0] = false;                   ⑤
remainder section;
P1 进程:
flag[1] = true;                    ⑥ //表示自己想进入临界区
turn = 0;                          ⑦ //可以优先让对方进入临界区
while (flag[0] && turn==0);        ⑧ //对方想进，且最后一次是自己“让梨”，那自己就循环等待
critical section;                  ⑨
flag[1] = false;                   ⑩ //访问完临界区，表示自己已经不想访问临界区了
```

进入区：1. 主动争取；2. 主动谦让；3. 检查对方是否也想使用，且最后一次是不是自己说了“客气话”





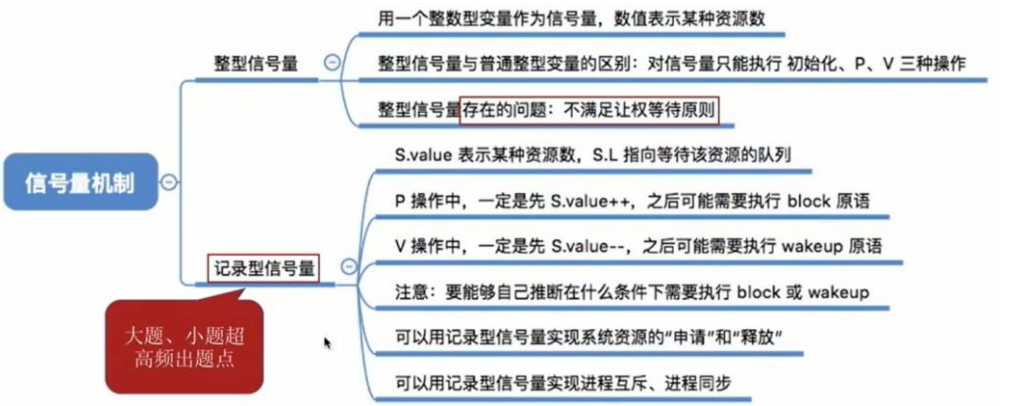

```c
/*记录型信号量的定义*/
typedef struct {
  int value;          //剩余资源数
  struct process *L;  //等待队列
} semaphore;
```

```c
/*某进程需要使用资源时，通过 wait 原语申请*/
void wait (semaphore S) {
  S.value--;
  if (S.value < 0 ) {
    block (S.L);
  }
}
```

如果剩余资源数不够，使用block原语使进程从运行态进入阻塞态，并把挂到信号量 S 的等待

```c
/*进程使用完资源后，通过 signal 原语释放*/
void signal (semaphore S) {
  s.value++;
  if (S.value <= 0) {
    wakeup(S.L);
  }
}
```

释放资源后，若还有别的进程在等待这种资源，则使用 wakeup 原语唤醒等待队列中的一个进程，该进程从阻塞态变



进程 P1 中有句代码 S1，P2 中有句代码 S2 ...P3... P6 中有句代码 S6。这些代码要求按如下前驱图所示的顺序来执行：

其实每一对前驱关系都是一个进程同步问题（需要保证一前一后的操作）
因此，

1. 要为每一对前驱关系各设置一个同步变量
2. 在“前操作”之后对相应的同步变量执行 V 操作
3. 在“后操作”之前对相应的同步变量执行 P 操作

```
P1() {
  ...
  S1;
  V(a);
  V(b);
  ...
}
```

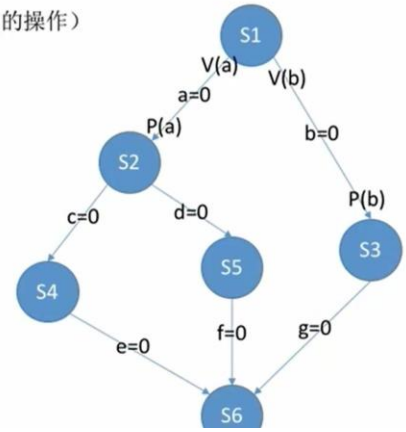
```
P2() {
  ...
  P(a);
  S2;
  V(c);
  V(d);
  ...
}
```

```
P3() {
  ...
  P(b);
  S3;
  V(g);
  ...
}
```

```
P4(){
  ...
  P(c);
  S4;
  V(e);
  ...
}
```

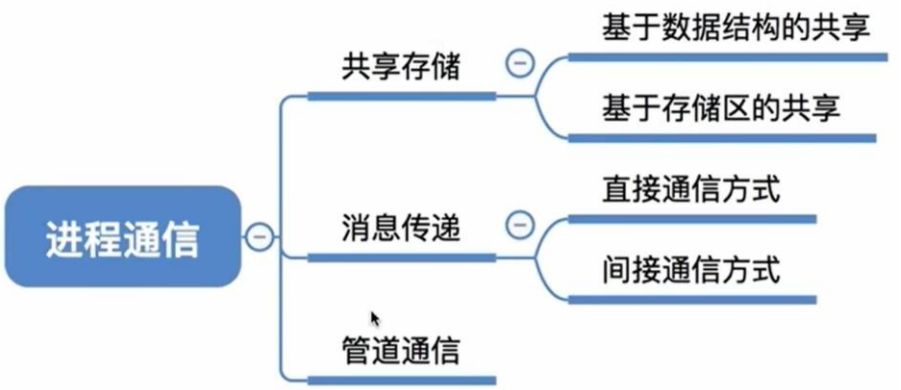
```
P5(){
  ...
  P(d);
  S5;
  V(f);
  ...
}
```

```
P6() {
  ...
  P(e);
  P(f);
  P(g);
  S6;
  ...
}
```



学习技巧：进程控制会导致进程状态的转换。无论哪个原语，要做的无非三类事情：

1. 更新PCB中的信息（如修改进程状态标志、将运行环境保存到PCB、从PCB恢复运行环境）
   a. 所有的进程控制原语一定都会修改进程状态标志
   b. 剥夺当前运行进程的CPU使用权必然需要保存其运行环境
   c. 某进程开始运行前必然要恢复期运行环境
2. 将PCB插入合适的队列
3. 分配/回收资源



重点重点重点：
操作系统只“看得见”内核级线程，因此只有内核级线程才是处理机分配的单位。

例如：左边这个模型中，该进程由两个内核级线程，三个用户级线程，在用户看来，这个进程中有三个线程。但即使该进程在一个4核处理机的计算机上运行，也最多只能被分配到两个核，最多只能有两个用户线程并行执行。

请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放。

可以采用静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不让它投入运行。一旦投入运行后，这些资源就一直归它所有，该进程就不会再请求别的任何资源

循环等待条件：存在一种进程资源的循环等待链，链中的每一个进程已获得的资源同时被下一个进程所请求。

可采用顺序资源分配法。首先给系统中的资源编号，规定每个进程必须按编号递增的顺序请求资源，同类资源（即编号相同的资源）一次申请完。

银行家算法步骤：
①检查此次申请是否超过了之前声明的最大需求数
②检查此时系统剩余的可用资源是否还能满足这次请求
③试探着分配，更改各数据结构
④用安全性算法检查此次分配是否会导致系统进入不安全状态

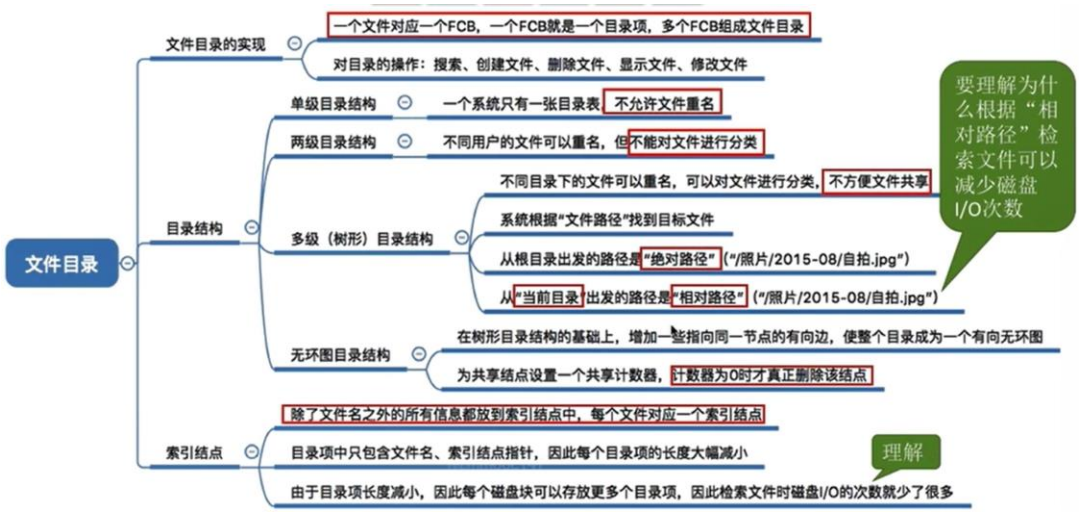
安全性算法步骤：
检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。
不断重复上述过程，看最终是否能让所有进程都加入安全序列。

系统处于不安全状态未必死锁，但死锁时一定处于不安全状态。系统处于安全状态一定不会死锁。

补充：并不是系统中所有的进程都是死锁状态，用死锁检测算法化简资源分配图后，还连着边的那些进程就是死锁进程
解除死锁的主要方法有：

1. 资源剥夺法。挂起（暂时放到外存上）某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿。
2. 撤销进程法（或称终止进程法）。强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来。
3. 进程回退法。让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。

索引顺序文件是索引文件和顺序文件思想的结合。索引顺序文件中，同样会为文件建立一张索引表，但不同的是：并不是每个记录对应一个索引表项，而是一组记录对应一个索引表项。



链接分配采取离散分配的方式，可以为文件分配离散的磁盘块。分为隐式链接和显式链接两种。

隐式链接——除文件的最后一个盘块之外，每个盘块中都存有指向下一个盘块的指针。文件目录包括文件第一块的指针和最后一块的指针。
优点：很方便文件拓展，不会有碎片问题，外存利用率高。
缺点：只支持顺序访问，不支持随机访问，查找效率低，指向下一个盘块的指针也需要耗费少量的存储空间。

显式链接——把用于链接文件各物理块的指针显式地存放在一张表中，即 文件分配表（FAT，File Allocation Table）。一个磁盘只会建立一张文件分配表。开机时文件分配表放入内存，并常驻内存。
优点：很方便文件拓展，不会有碎片问题，外存利用率高，并且支持随机访问。相比于隐式链接来说，地址转换时不需要访问磁盘，因此文件的访问效率更高。
缺点：文件分配表的需要占用一定的存储空间。

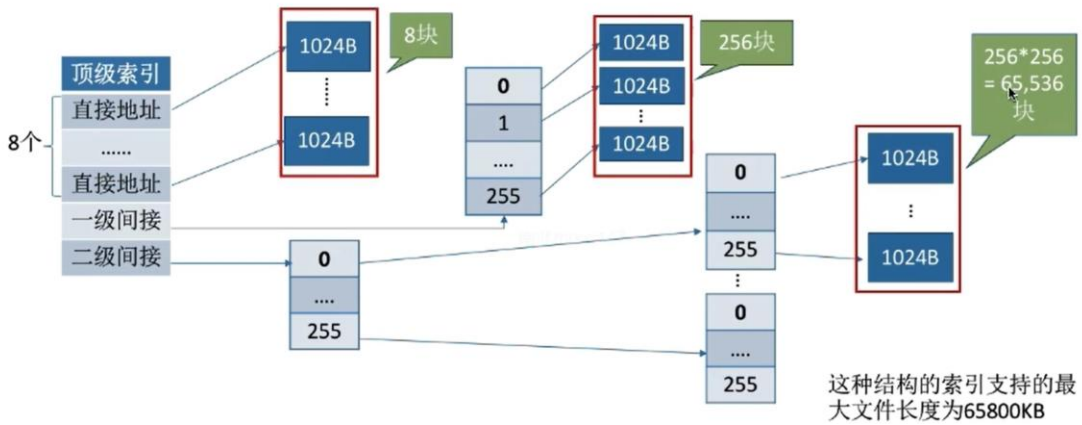
索引分配允许文件离散地分配在各个磁盘块中，系统会为每个文件建立一张索引表，索引表中记录了文件的各个逻辑块对应的物理块（索引表的功能类似于内存管理中的页表——建立逻辑页面到物理页之间的映射关系）。索引表存放的磁盘块称为索引块。文件数据存放的磁盘块称为数据块。
若文件太大，索引表项太多，可以采取以下三种方法解决：
①链接方案：如果索引表太大，一个索引块装不下，那么可以将多个索引块链接起来存放。缺点：若文件很大，索引表很长，就需要将很多个索引块链接起来。想要找到 i 号索引块，必须先依次读入 0~i-1 号索引块，这就导致磁盘I/O次数过多，查找效率低下。

②多层索引：建立多层索引（原理类似于多级页表）。使第一层索引块指向第二层的索引块。还可根据文件大小的要求再建立第三层、第四层索引块。采用 K 层索引结构，且顶级索引表未调入内存，则访问一个数据块只需要 K + 1 次读磁盘操作。缺点：即使是小文件，访问一个数据块依然需要K+1次读磁盘。

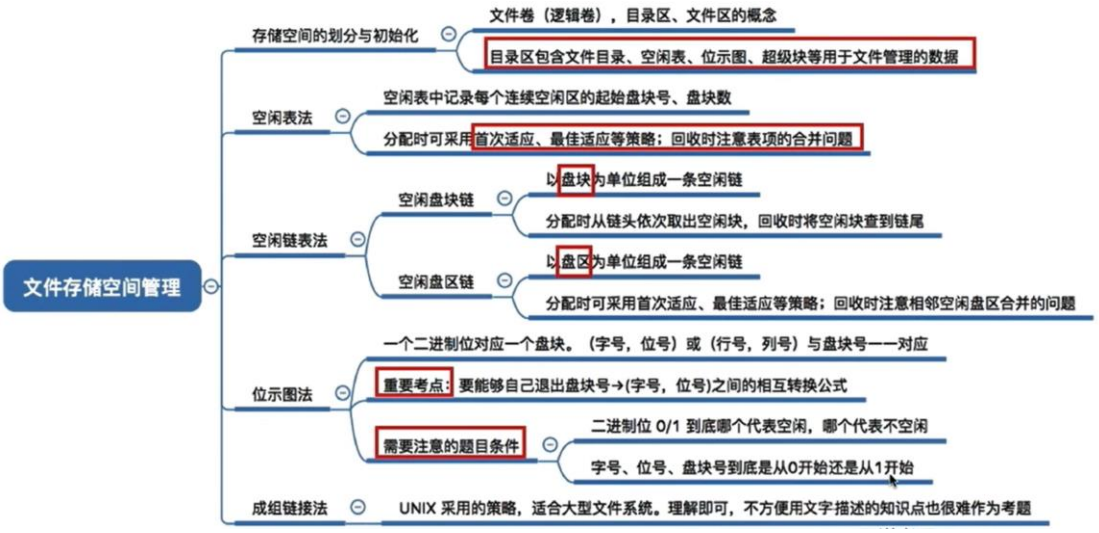
③混合索引：多种索引分配方式的结合。例如，一个文件的顶级索引表中，既包含直接地址索引（直接指向数据块），又包含一级间接索引（指向单层索引表）、还包含两级间接索引（指向两层索引表）。
优点：对于小文件来说，访问一个数据块所需的读磁盘次数更少。

超级超级超级重要考点：①要会根据多层索引、混合索引的结构计算出文件的最大长度（Key：各级索引表最大不能超过一个块）；②要能自己分析访问某个数据块所需要的读磁盘次数（Key：FCB中会存有指向顶级索引块的指针，因此可以根据FCB读入顶级索引块。每次读入下一级的索引块都需要一次读磁盘操作。另外，要注意题目条件——顶级索引块是否已调入内存）

③混合索引：多种索引分配方式的结合。例如，一个文件的顶级索引表中，既包含直接地址索引（直接指向数据块），又包含一级间接索引（指向单层索引表）、还包含两级间接索引（指向两层索引表）。



这种结构的索引支持的最大文件长度为65800KB



用一个例子来辅助记忆文件系统的层次结构：
假设某用户请求删除文件 “D:/工作目录/学生信息.xlsx” 的最后100条记录。

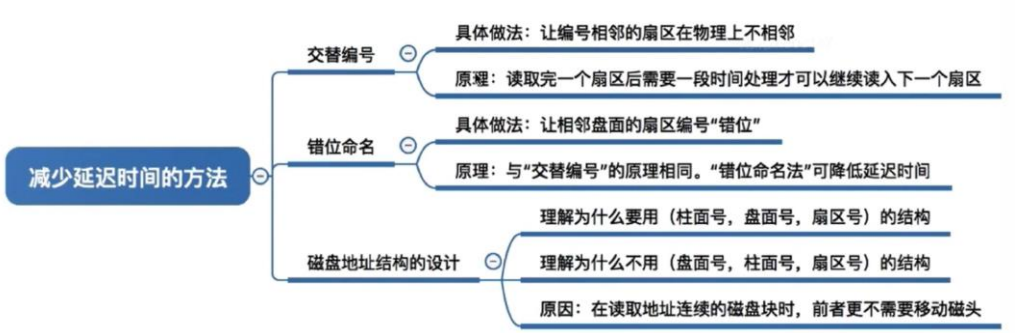
1. 用户需要通过操作系统提供的接口发出上述请求——用户接口
2. 由于用户提供的是文件的存放路径，因此需要操作系统一层一层地查找目录，找到对应的目录项——文件目录系统
3. 不同的用户对文件有不同的操作权限，因此为了保证安全，需要检查用户是否有访问权限——存取控制模块（存取控制验证层）
4. 验证了用户的访问权限之后，需要把用户提供的“记录号”转变为对应的逻辑地址——逻辑文件系统与文件信息缓冲区
5. 知道了目标记录对应的逻辑地址后，还需要转换成实际的物理地址——物理文件系统
6. 要删除这条记录，必定要对磁盘设备发出请求——设备管理程序模块
7. 删除这些记录后，会有一些盘块空闲，因此要将这些空闲盘块回收——辅助分配模块

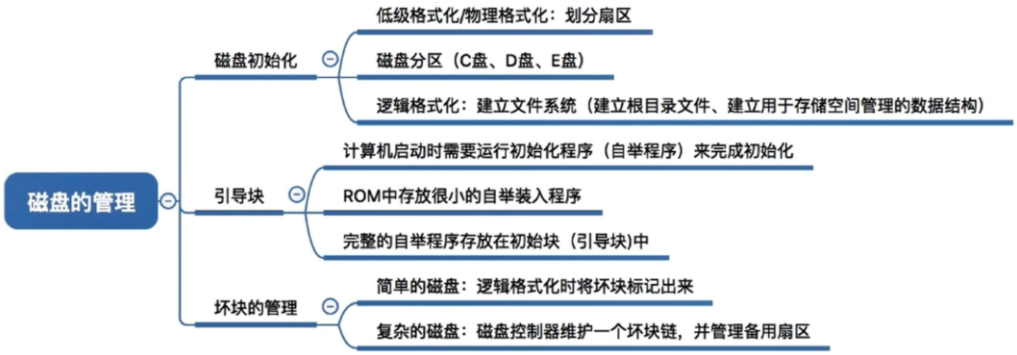
可用（柱面号，盘面号，扇区号）来定位任意一个“磁盘块”。在“文件的物理结构”小节中，我们经常提到文件数据存放在外存中的几号块，这个块号就可以转换成（柱面号，盘面号，扇区号）的地址形式。

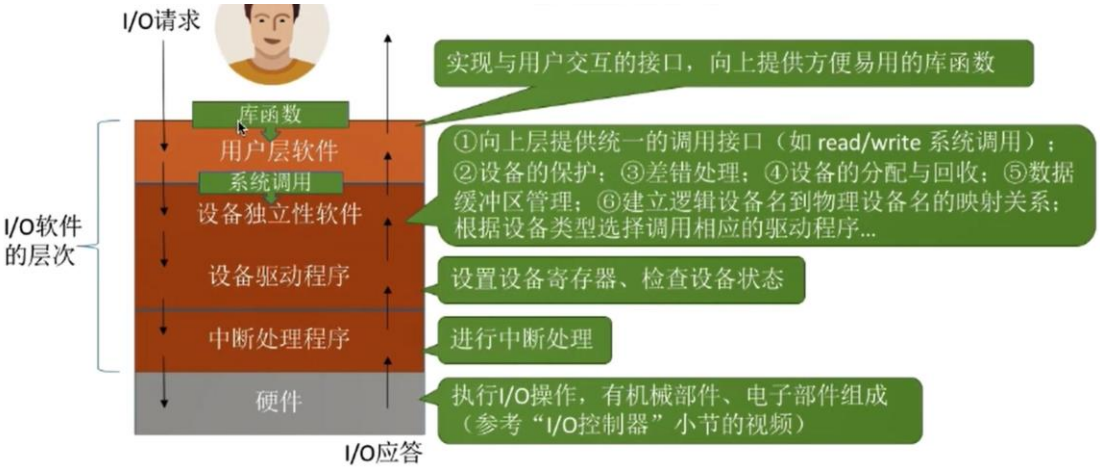
寻找时间（寻道时间）T_S：在读/写数据前，将磁头移动到指定磁道所花的时间。
①启动磁头臂是需要时间的。假设耗时为 s；
②移动磁头也是需要时间的。假设磁头匀速移动，每跨越一个磁道耗时为 m，总共需要跨越 n 条磁道。则：
寻道时间 $T_S = s + m*n$

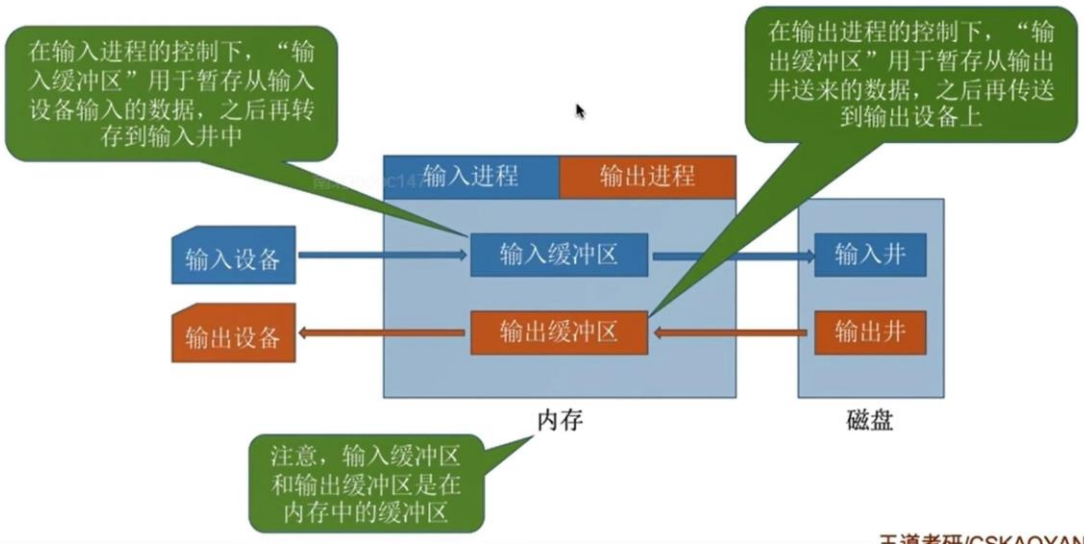
延迟时间T_R：通过旋转磁盘，使磁头定位到目标扇区所需要的时间。设磁盘转速为 r（单位：转/秒，或 转/分），则
平均所需的延迟时间 $T_R = (1/2)*(1/r) = 1/2r$

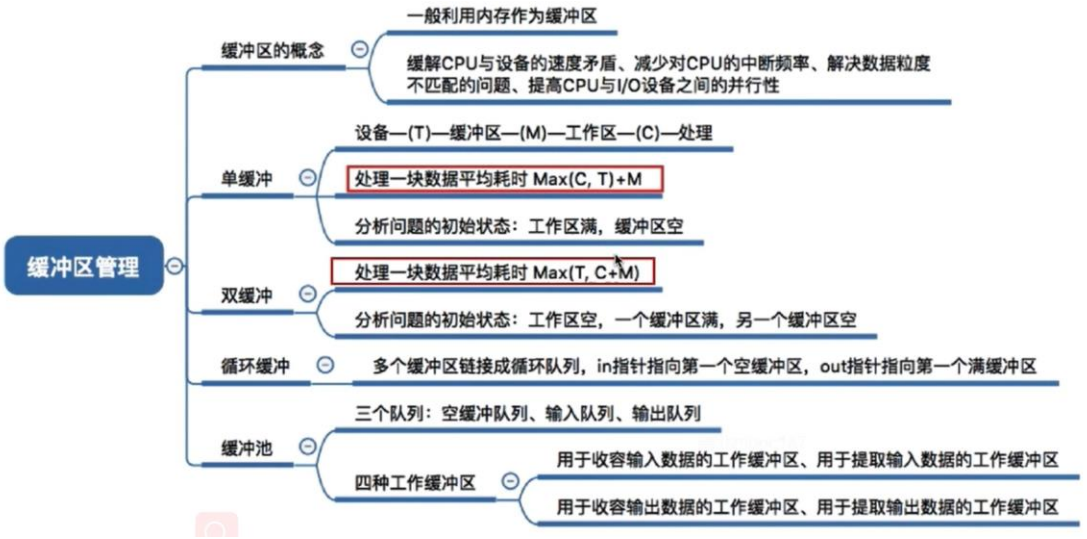
传输时间T_t：从磁盘读出或向磁盘写入数据所经历的时间，假设磁盘转速为 r，此次读/写的字节数为 b，每个磁道上的字节数为 N。则：
传输时间$T_t = (1/r) * (b/N) = b/(rN)$


减少延迟时间的方法
交替编号
具体做法：让编号相邻的扇区在物理上不相邻
原理：读取完一个扇区后需要一段时间处理才可以继续读入下一个扇区
错位命名
具体做法：让相邻盘面的扇区编号“错位”
原理：与“交替编号”的原理相同。“错位命名法”可降低延迟时间
磁盘地址结构的设计
理解为什么要用（柱面号，盘面号，扇区号）的结构
理解为什么不用（盘面号，柱面号，扇区号）的结构
原因：在读取地址连续的磁盘块时，前者更不需要移动磁头


磁盘的管理
磁盘初始化
低级格式化/物理格式化：划分扇区
磁盘分区（C盘、D盘、E盘）
逻辑格式化：建立文件系统（建立根目录文件、建立用于存储空间管理的数据结构）
引导块
计算机启动时需要运行初始化程序（自举程序）来完成初始化
ROM中存放很小的自举装入程序
完整的自举程序存放在初始块（引导块)中
坏块的管理
简单的磁盘：逻辑格式化时将坏块标记出来
复杂的磁盘：磁盘控制器维护一个坏块链，并管理备用扇区


I/O请求
库函数
用户层软件
系统调用
设备独立性软件
设备驱动程序
中断处理程序
硬件
I/O软件的层次
I/O应答
实现与用户交互的接口，向上提供方便易用的库函数
①向上层提供统一的调用接口（如 read/write 系统调用）；②设备的保护；③差错处理；④设备的分配与回收；⑤数据缓冲区管理；⑥建立逻辑设备名到物理设备名的映射关系；根据设备类型选择调用相应的驱动程序...
设置设备寄存器、检查设备状态
进行中断处理
执行I/O操作，有机械部件、电子部件组成（参考“I/O控制器”小节的视频）


在输入进程的控制下，“输入缓冲区”用于暂存从输入设备输入的数据，之后再转存到输入井中
在输出进程的控制下，“输出缓冲区”用于暂存从输出井送来的数据，之后再传送到输出设备上
输入进程
输出进程
输入设备
输入缓冲区
输入井
输出设备
输出缓冲区
输出井
内存
磁盘
注意，输入缓冲区和输出缓冲区是在内存中的缓冲区



①根据进程请求的物理设备名查找SDT（注：物理设备名是进程请求分配设备时提供的参数）
②根据SDT找到DCT，若设备忙碌则将进程PCB挂到设备等待队列中，不忙碌则将设备分配给进程。
③根据DCT找到COCT，若控制器忙碌则将进程PCB挂到控制器等待队列中，不忙碌则将控制器分配给进程。
④根据COCT找到CHCT，若通道忙碌则将进程PCB挂到通道等待队列中，不忙碌则将通道分配给进程。

①根据进程请求的逻辑设备名查找SDT（注：用户编程时提供的逻辑设备名其实就是“设备类型”）
②查找SDT，找到用户进程指定类型的、并且空闲的设备，将其分配给该进程。操作系统在逻辑设备表（LUT）中新增一个表项。
③根据DCT找到COCT，若控制器忙碌则将进程PCB挂到控制器等待队列中，不忙碌则将控制器分配给进程。
④根据COCT找到CHCT，若通道忙碌则将进程PCB挂到通道等待队列中，不忙碌则将通道分配给进程。

静态重定位：又称可重定位装入。编译、链接后的装入模块的地址都是从0开始的，指令中使用的地址、数据存放的地址都是相对于起始地址而言的逻辑地址。可根据内存的当前情况，将装入模块装入到内存的适当位置。装入时对地址进行“重定位”，将逻辑地址变换为物理地址（地址变换是在装入时一次完成的）。

静态重定位的特点是在一个作业装入内存时，必须分配其要求的全部内存空间，如果没有足够的内存，就不能装入该作业。作业一旦进入内存后，在运行期间就不能再移动，也不能再申请内存空间。

动态重定位：又称动态运行时装入。编译、链接后的装入模块的地址都是从0开始的。装入程序把装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要执行时才进行。因此装入内存后所有的地址依然是逻辑地址。这种方式需要一个重定位寄存器的支持。

并且可将程序分配到不连续的存储区中；在程序运行前只需装入它的部分代码即可投入运行，然后在程序运行期间，根据需要动态申请分配内存；便于程序段的共享，可以向用户提供一个比存储空间大得多的地址空间。

交换（对换）技术的设计思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存（进程在内存与磁盘间动态调度）

1. 具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式；对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式（学过文件管理章节后即可理解）。总之，对换区的I/O速度比文件区的更快。
2. 交换通常在许多进程运行且内存吃紧时进行，而系统负荷降低就暂停。例如：在发现许多进程运行时经常发生缺页，就说明内存紧张，此时可以换出一些进程；如果缺页率明显下降，就可以暂停换出。
3. 可优先换出阻塞进程；可换出优先级低的进程；为了防止优先级低的进程在被调入内存后很快又被换出，有的系统还会考虑进程在内存的驻留时间...

（注意：PCB 会常驻内存，不会被换出外存）

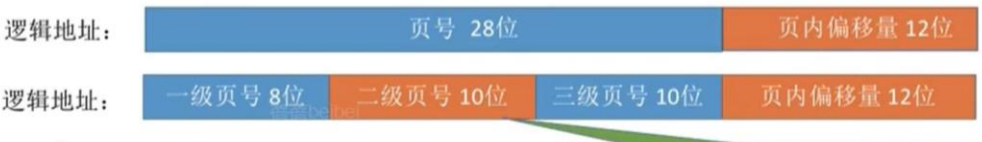
基本地址变换机构可以借助进程的页表将逻辑地址转换为物理地址。
通常会在系统中设置一个页表寄存器（PTR），存放页表在内存中的起始地址F 和页表长度M。
进程未执行时，页表的始址 和 页表长度 放在进程控制块（PCB）中，当进程被调度时，操作系统内核会把它们放到页表寄存器中。

结论：理论上，页表项长度为 3B 即可表示内存块号的范围，但是，为了方便页表的查询，常常会让一个页表项占更多的字节，使得每个页面恰好可以装得下整数个页表项。

1. 若采用多级页表机制，则各级页表的大小不能超过一个页面

例：某系统按字节编址，采用 40 位逻辑地址，页面大小为 4KB，页表项大小为 4B，假设采用纯页式存储，则要采用（）级页表，页内偏移量为（）位？

页面大小 = 4KB =2^{12}B，按字节编址，因此页内偏移量为12位
页号 = 40 - 12 = 28 位
页面大小 = 2^{12}B，页表项大小 = 4B，则每个页面可存放 2^{12} / 4 = 2^{10} 个页表项
因此各级页表最多包含 2^{10} 个页表项，需要 10 位二进制位才能映射到 2^{10} 个页表项，因此每一级的页表对应页号应为10位。总共28位的页号至少要分为三级



2. 两级页表的访存次数分析（假设没有快表机构）
第一次访存：访问内存中的页目录表
第二次访存：访问内存中的二级页表
第三次访存：访问目标内存单元

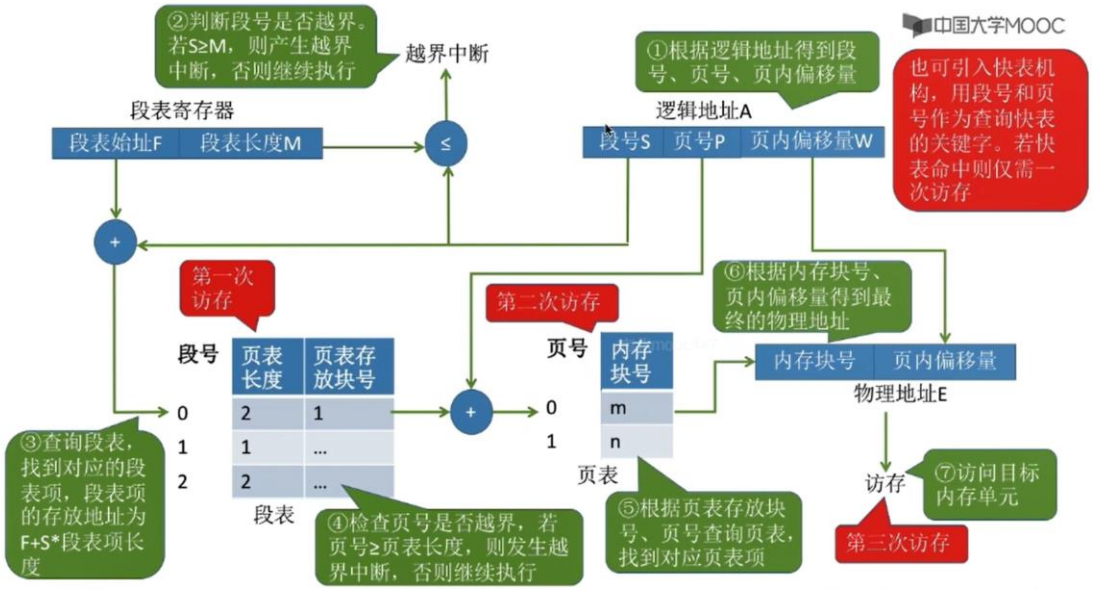
如果只分为两级页表，则一级页号占 18 位，也就是说页目录表中最多可能有 2^{18} 个页表项，显然，一个页面是放不下这么多页表项的。

分段比分页更容易实现信息的共享和保护。不能被修改的代码称为纯代码或可重入代码（不属于临界资源），这样的代码是可以共享的。可修改的代码是不能共享的

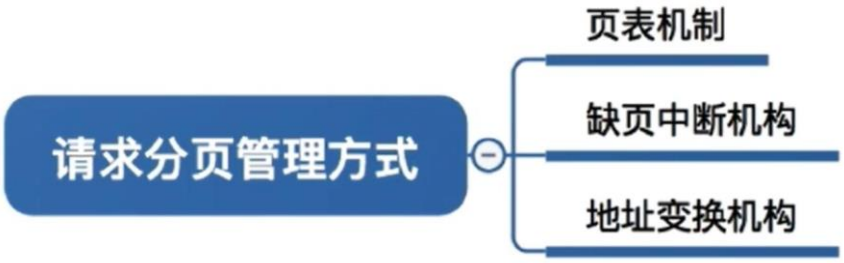
	优点	缺点
分页管理	内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理会产生外部碎片

“分段”对用户是可见的，程序员编程时需要显式地给出段号、段内地址。而将各段“分页”对用户是不可见的。系统会根据段内地址自动划分页号和页内偏移量。
因此段页式管理的地址结构是二维的。

每个段对应一个段表项，每个段表项由段号、页表长度、页表存放块号（页表起始地址）组成。每个段表项长度相等，段号是隐含的。
每个页面对应一个页表项，每个页表项由页号、页面存放的内存块号组成。每个页表项长度相等，页号是隐含的。



虚拟内存的最大容量是由计算机的地址结构（CPU寻址范围）确定的
虚拟内存的实际容量 = min（内存和外存容量之和，CPU寻址范围）



注意：缺页时未必发生页面置换。若还有可用的空闲内存块就不用进行页面置换。

Belady 异常——当为进程分配的物理块数增大时，缺页次数不减反增的异常现象。

只有 FIFO 算法会产生 Belady 异常。另外，FIFO算法虽然实现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此，算法性能差

简单的CLOCK 算法实现方法：为每个页面设置一个访问位，再将内存中的页面都通过链接指针链接成一个循环队列。当某页被访问时，其访问位置为1。当需要淘汰一个页面时，只需检查页的访问位。如果是0，就选择该页换出；如果是1，则将它置为0，暂不换出，继续检查下一个页面，若第一轮扫描中所有页面都是1，则将这些页面的访问位依次置为0后，再进行第二轮扫描（第二轮扫描中一定会有访问位为0的页面，因此简单的CLOCK 算法选择一个淘汰页面最多会经过两轮扫描）

算法规则：将所有可能被置换的页面排成一个循环队列
第一轮：从当前位置开始扫描到第一个（0, 0）的帧用于替换。本轮扫描不修改任何标志位
第二轮：若第一轮扫描失败，则重新扫描，查找第一个（0, 1）的帧用于替换。本轮将所有扫描过的帧访问位设为0
第三轮：若第二轮扫描失败，则重新扫描，查找第一个（0, 0）的帧用于替换。本轮扫描不修改任何标志位
第四轮：若第三轮扫描失败，则重新扫描，查找第一个（0, 1）的帧用于替换。
由于第二轮已将所有帧的访问位设为0，因此经过第三轮、第四轮扫描一定会有一个帧被选中，因此改进型CLOCK置换算法选择一个淘汰页面最多会进行四轮扫描

第一优先级：最近没访问，且没修改的页面
第二优先级：最近没访问，但修改过的页面
第三优先级：最近访问过，但没修改的页面
第四优先级：最近访问过，且修改过的页面

驻留集：指请求分页存储管理中给进程分配的物理块的集合。
在采用了虚拟存储技术的系统中，驻留集大小一般小于进程的总大小。
若驻留集太小，会导致缺页频繁，系统要花大量的时间来处理缺页，实际用于进程推进的时间很少；驻留集太大，又会导致多道程序并发度下降，资源利用率降低。所以应该选择一个合适的驻留集大小。

固定分配：操作系统为每个进程分配一组固定数目的物理块，在进程运行期间不再改变。即，驻留集大小不变
可变分配：先为每个进程分配一定数目的物理块，在进程运行期间，可根据情况做适当的增加或减少。即，驻留集大小可变

局部置换：发生缺页时只能选进程自己的物理块进行置换。
全局置换：可以将操作系统保留的空闲物理块分配给缺页进程，也可以将别的进程持有的物理块置换到外存，再分配给缺页进程。

	局部置换	全局置换
固定分配	√	—
可变分配	√	√

全局置换意味着一个进程拥有的物理块数量必然会改变，因此不可能是固定分配

中国大学MOOC

页面分配、置换策略

固定分配局部置换：系统为每个进程分配一定数量的物理块，在整个运行期间都不改变。若进程在运行中发生缺页，则只能从该进程在内存中的页面中选出一页换出，然后再调入需要的页面。这种策略的缺点是：很难在刚开始就确定应为每个进程分配多少个物理块才算合理。（采用这种策略的系统可以根据进程大小、优先级、或是根据程序员给出的参数来确定为一个进程分配的内存块数）

可变分配全局置换：刚开始会为每个进程分配一定数量的物理块。操作系统会保持一个空闲物理块队列。当某进程发生缺页时，从空闲物理块中取出一块分配给该进程；若已无空闲物理块，则可选择一个未锁定的页面换出外存，再将该物理块分配给缺页的进程。采用这种策略时，只要某进程发生缺页，都将获得新的物理块，仅当空闲物理块用完时，系统才选择一个未锁定的页面调出。被选择调出的页可能是系统中任何一个进程中的页，因此这个被选中的进程拥有的物理块会减少，缺页率会增加。

可变分配局部置换：刚开始会为每个进程分配一定数量的物理块。当某进程发生缺页时，只允许从该进程自己的物理块中选出一个进行换出外存。如果进程在运行中频繁地缺页，系统会为该进程多分配几个物理块，直至该进程缺页率趋势适当程度；反之，如果进程在运行中缺页率特别低，则可适当减少分配给该进程的物理块。

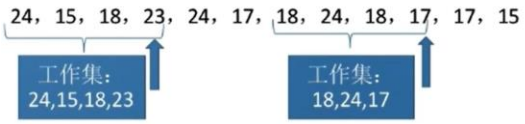
可变分配全局置换：只要缺页就给分配新物理块
可变分配局部置换：要根据发生缺页的频率来动态地增加或减少进程的物理块

1. 系统拥有足够的对换区空间：页面的调入、调出都是在内存与对换区之间进行，这样可以保证页面的调入、调出速度很快。在进程运行前需将进程相关的数据从文件区复制到对换区。

2. 系统缺少足够的对换区空间：凡是不会被修改的数据都直接从文件区调入，由于这些页面不会被修改，因此换出时不必写回磁盘，下次需要时再从文件区调入即可。对于可能被修改的部分，换出时需写回磁盘对换区，下次需要时再从对换区调入。

驻留集：指请求分页存储管理中给进程分配的内存块的集合。
工作集：指在某段时间间隔里，进程实际访问页面的集合。

操作系统会根据“窗口尺寸”来算出工作集。例：
某进程的页面访问序列如下，窗口尺寸为4，各时刻的工作集为？



工作集大小可能小于窗口尺寸，实际应用中，操作系统可以统计进程的工作集大小，根据工作集大小给进程分配若干内存块。如：窗口尺寸为5，经过一段时间的监测发现某进程的工作集最大为3，那么说明该进程有很好的局部性，可以给这个进程分配3个以上的内存块即可满足进程的运行需要。
一般来说，驻留集大小不能小于工作集大小，否则进程运行过程中将频繁缺页。