# 内存管理，

内存中的每一个小房间称为存储单元，内存地址从0开始，每个地址对应一个存储单元

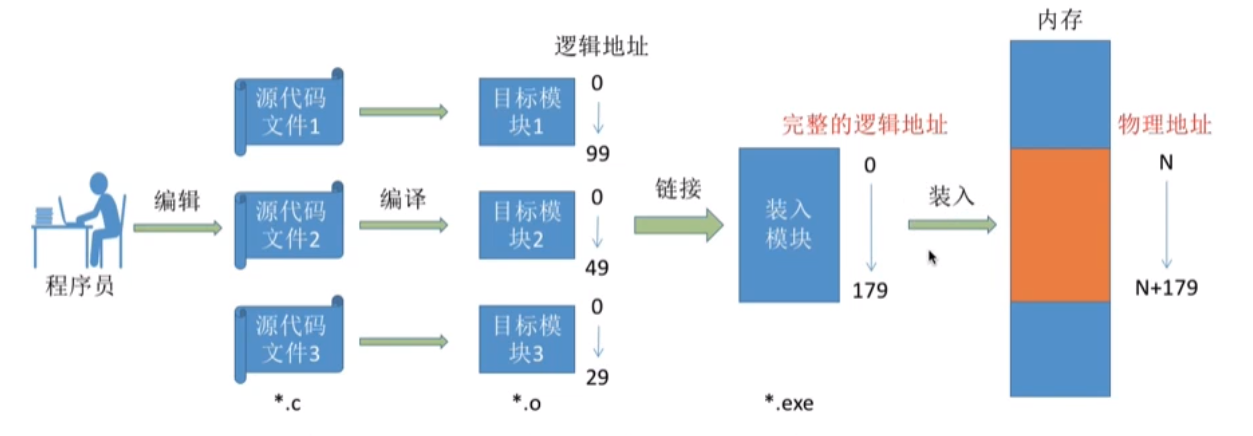
如果计算机按字节编址，则每个存储单元大小为1字节，即1B。

如果计算机按字编址，则每个存储单元的大小为1个字，每个字的大小为16位二进制（字长16位）

## 进程的运行原理—指令

操作码 数据地址

由于再编译的过程中不知道数据会放到什么为止，所以编译生成的指令一般是使用逻辑地址（相对地址） 通过相对位置得到绝对位置



装入模块（exe文件）

# 装入的三种方式

绝对装入：在编译时，如果知道程序放到内存中的哪个位置，编译程序将产生绝对地址的目标代码，装入程序按照装入模块中的地址，将程序数据装入内存。

（只适用于单道程序环境）

静态重定位：可重定位装入，编译链接后的装入模块地址都是从0开始，指令中的地址，数据存放的地址都是相对于起始地址而言的逻辑地址，可根据实际情况，将装入的地址进行重定位，将逻辑地址转变为物理地址（装入时一次性完成地址变换）

动态重定位：运行时装入，编译链接后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，并不会立即吧逻辑地址转换为物理地址Merseyside吧地址转换推到程序真正要执行的时候才执行，因此装入内存和所有的地址依旧是逻辑地址，这种方式需要一个重定位寄存器。（重定位寄存器设置为装入模块的的开始，并且加上逻辑地址，才能得到物理地址）

## 链接的三种方式

静态链接，装入时动态链接 运行时动态链接

内存空间与分配与回收

内存虚拟化

地址转换 绝对装入 静态重定位 动态重定位

内存保护（只能访问自己的地址）

1. 上限寄存器 下限寄存器
2. 重定位寄存器（基址寄存器） 界地址寄存器

# 内存空间扩增（替换，交换）

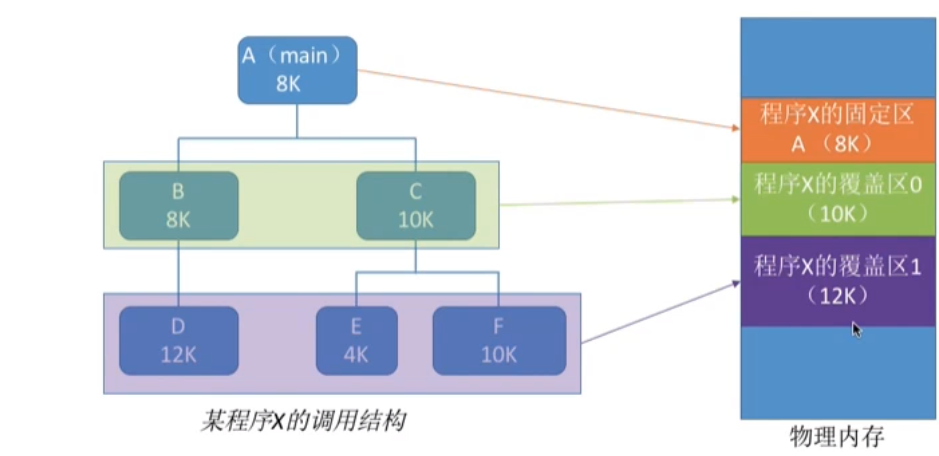
# 覆盖（一个进程）

覆盖的思想：将程序分为多个短，常用的段常驻内存，不常用的段在需要的时候调入内存

分一个 固定区 覆盖区

固定区放需要常驻的段，调入后就不再调出

不常用的段放在覆盖区，需要的时候调入内存，不需要则调出内存

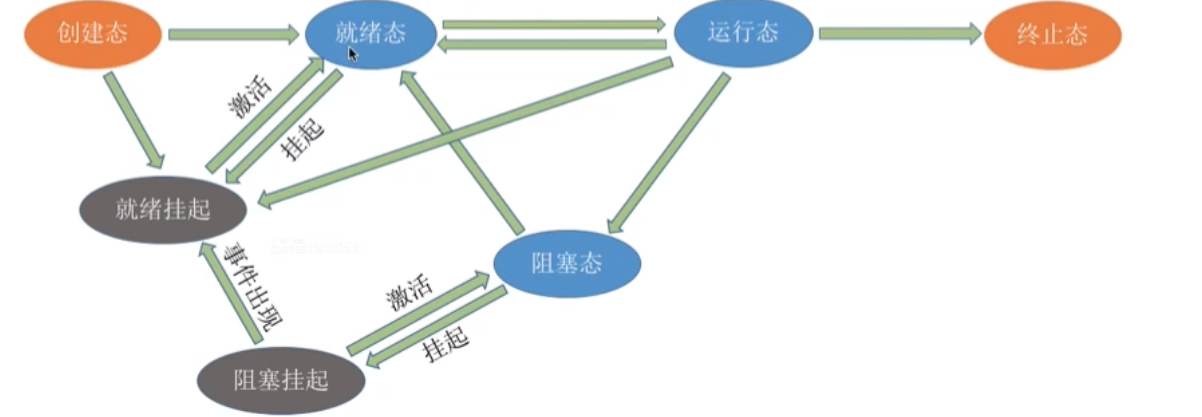


缺点：必须由程序员声明覆盖结构，操作系统完成自动覆盖，对用户不透明，增加编程负担。

# 交换技术（多个进程之间）

内存空间紧张的时候，将内存的进程换出外存，但是PCB要保存，放到挂起队列里，并且换出的位置放到PCB中，并且将外存的进程换入内存。

挂起态：就需挂起，阻塞挂起



问题：

1. 应该在外存的什么位置保存被换出的进程 ？
   1. 外存分成 “对换区” 和 “文件区” ，对换区采用连续存储，对换区速度大于文件区
2. 应该什么时候发生交换？
   1. 内存吃紧的时候
3. 应该换出那些进程？
   1. 优先换出阻塞进程 优先级比较低的进程

# 内存空间的分配与回收

## 连续分配管理方式

单一连续分配：内存分为系统区和用户区，内存中只能有一个程序，整个用户独占整个用户区。

优点：简单，没有外部碎片

缺点：只能用于单用户，单任务的操作系统，并且会产生内部碎片（本来给进程分配的，但是进程没有用上）

固定分区分配：（分区大小相等，分区大小不相等）

将整个用户空间划分为若干个固定大小的分区，在每个分区内只装入一道作业，形成了最早的多道程序管理方式。

分区大小相等：缺乏灵活性，但是适合于计算机控制多个相同对象的场合。（n个程序相同）

分区大小不等：增加了灵活性，可以满足多个不同大小程序的内存要求。

如何记录内存中分区的空闲或者分配状态：创建分区说明表（数据结构）

优点：没有外部碎片

缺点：如果程序太大，无法分区，所以采用覆盖技术。产生内部碎片。

动态分区分配：

又称为可变分区分配，这种分配方式不会预先划分内存分区，而是装入的时候按照进程大小动态建立分区，因此分区的数目和大小是可变的。

记录数据结构：空闲分区表：使用表记录空闲的分区

空闲分区链：使用链式存储方式将多个分区连接起来

多个空闲分区都满足的时候，应该选择哪个分区进行分配？

动态分配分区算法：从空闲分区选择出一个分区进行分配作业。

分区的分配与回收：

分配：有两种情况第一种是空闲分区表中的分区大小大于作业的需求大小，这样的话空闲分区表的记录数量是不变的，只需改变相应记录的剩余分区大小

第二种情况是空闲分区表中的分区大小等于作业的需求，这样分区表的记录数量惠改变，减少一个记录。

回收：如果回收的两个空闲分区相邻，那么要将其合并并且更新到空闲分区表中。

如果有多个空闲分区相邻，那么将多个分区合并，并且更新到表中。

如果前后都没有空闲分区相邻，那么直接加入到空闲分区表中。

内部碎片：分配给某进程的内存区域中，有些部分没有用上。

内部碎片是已经被分配出去的的内存空间大于请求所需的内存空间。

外部碎片：内存中的某些空闲分区由于太小而难以利用。

还没有分配出去，但是由于大小太小而无法分配给申请空间的新进程的内存空间空闲块。

外部碎片是除了任何已分配区域或页面外部的空闲存储块。这些存储块的总和可以满足当前申请的长度要求，但是由于它们的地址不连续或其他原因，使得系统无法满足当前申请。

拼凑技术来解决外部碎片：将不相连的进程移位。（动态重定位的装入方式）

动态分区和固定分区分配方式相比，内存空间的利用率要高些。但是，总会存在一些分散的较小空闲分区，即外部碎片，它们存在于已分配分区之间，不能充分利用。可以采用拼接技术加以解决。固定分区分配方式存在内部碎片，而无外部碎片，而动态分区分配方式存在外部碎片，无内部碎片。

动态分区分配算法

多个分区都满足的时候，应该选择哪个？

首次适应算法 First Fit

从低地址开始查找，找到第一个能满足大小的分区（地址从低到高排列）

低地值出现很多很小的空闲分区

临近适应算法 next fit

将空闲分区链设置为循环链表，每次都从上次查找之后的那个分区开始往后查找，解决了首次适应算法低地值出现很多很小的空闲分区

由于首次适应算法都要从低地值开始检索，但是低地值都是小分区，不如从上一次开始往后

以上开销好

最佳适应算法 Best Fit

优先使用更小空闲区，保证大片的空闲区

将空闲分区以容量递增的方式分配，并且找到大小能满足要求的第一个分区

每次选择的都是最小的分区，所以会产生很多的外部碎片

最坏适应法 Largest Fit

为了解决最佳适应算法留下太多外部碎片的问题，所以每次优先分配最大的区域

空闲分区按照容量递减连接，每次找到满足要求的第一个分区

缺点：造成内部碎片 不适用于大进程

## 非连续分配管理方式

分页存储管理的基本概念：

内存分为一个个大小相等的分区(例如每个分区4k)，每个分区就是一个页框/页帧/内存块/物理块，每个框有一个编号 “页框号“ 页框号从0开始

将用户进程地址空间也分为与页框大小相等的页面，每个页面也有一个编号“页号“，从0开始。

最后一个页面可能没有那么大，因此页框不能太大，否则产生内部碎片

页面与页框一一对应，且页面与页框不一定一一对应

如何实现逻辑地址->物理地址的转换

页初始地址+页偏移量

例如 80 在1号页面内 所以逻辑地址=450+（80-50）=480//50个单元一个页面

1. 求页号 逻辑地址/页面长度=页号
2. 求该页面起始地址 操作系统会记录
3. 求页面中的偏移量 逻辑地址%页面长度=偏移量
4. 物理地址=页面地址+偏移量

计算机通常将页面大小设置为2的整数次幂



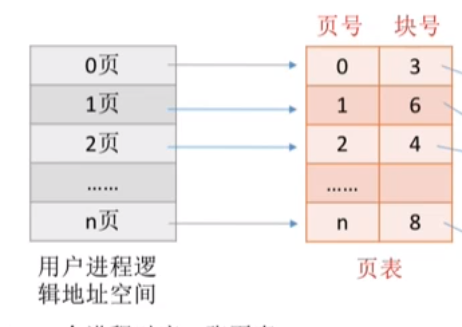
即前20位代表页面号，后12位代表页内逻辑地址

一个地址结构分为页号和页内偏移量（k位偏移量 则页面大小为2^k）

有M位表示页号，那么有2^M个页面

页表 为了知道进程的每个页面在内存中的位置，操作系统要为每个进程建立一张页表

注意表内记录的是块号 通过块号\*页长度可以得到页面的初始地址



每一项都是页表项

块号\*每个块的大小=起始地址

每个页表项的长度是相同的，但是页号是隐藏的(存放起始地址即可)

基本地址变换机构

逻辑地址到物理地址的一组硬件机构

页表寄存器 PTR 存放页表内存中的起始地址F和页表长度M，没有上粗处理机的时候，放在PCB ,上处理机之后放入PTR

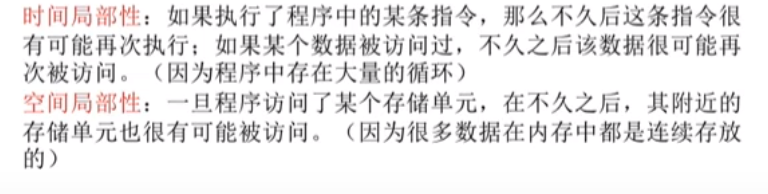
页号需要与M对比 如果页号>=M则异常



分页管理是一维的 即我们只需要告诉cpu逻辑地址的值

具有块表的地址变换机构 TLB联想存储器

局部性原理：



先查找TLB 没找到再查找慢表

# 两级页表

页表需要占用很大的空间，所以可以让他不连续的占用内存空间

可以将很长的页表分组 ，使得每个内存块刚好可以放入一个分组，比如页面大小4KB，每个页表项4B,这样一个页面可以存放1K个页表项，因此每1K个页表项为一组，每组刚好占用一个内存块，再将各组离散的放到各内存块中。

另外，就需要再创建一张页表，称为页目录表，或者外层/顶层页表

同样的 将逻辑地址拆成三个部分即：

一级页号 二级页号 偏移地址

采用多级页表的时候，各级页表的大小不能超过一个页面

某系统按字节编制，采用40位逻辑地址，页面大小位4KB,页表项大小为4B，假设采用纯页式存储，要采用(3)级页表，页内偏移量为(12)位。

页面大小4KB,偏移量为2^12 那么还剩下28位，所以偏移量位12位

页表项大小位4B，所以一页可以存储2^10个页表，所以28位需要分成8 10 10 12

两级页表的访存次数分析（无TLB）

1. 访问一级页表
2. 访问二级页表
3. 访问内存

Implementing LRU

Timer too expensive

时钟轮转法

转的快/慢、

被修改过的界面（写磁盘 ）

# 分段内存管理

进程的地址空间按照自身的逻辑关系划分为若干个段，每个段都有一个段名，每段从0开始编址

内存分配规则，以段位单位进行分配，每个段再内存中占据连续空间，但是各段之间不相邻。

地址分为

|  |  |
| --- | --- |
| 段号 16位 | 段内地址 16位 |

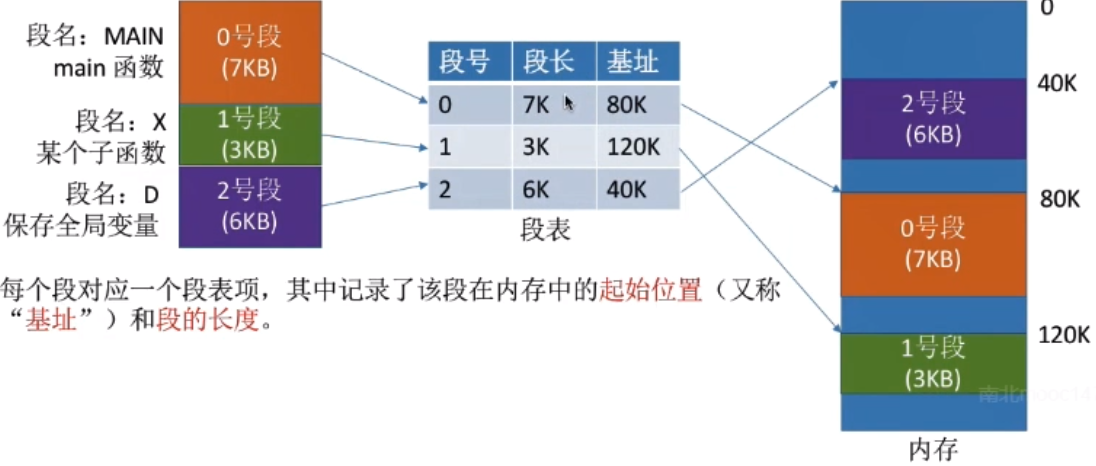
段号的位置决定内个进程最多分多少个段，段内地址决定了每个段的最大长度是多少

每个进程最多有2^16=64K个段 并且每个段内最大长度为64K

[x]翻译成对应的段号 <A>翻译成段内地址 [x]|<A>

段表

段号 段长 基址



段长需要记录，因为段的大小不相同

在页表中，页表记录的是块号，并且页号还是隐藏的，所以4GB 内存 4B页长共有

1G=2^20个空间，所以用三个字节代表一个页表段

段需要有短号 段长 段基址 逻辑结构用16位代表段号，段内地址16位，32位内存 4GB，段号可以是隐藏的，不占空间，所以共有48位即6B代表段表的一个映射。即段表的基址是M，那么K号段就是M+6K

段表寄存器

段表初始地址F 段表长度M

逻辑地址

|  |  |
| --- | --- |
| 段号S | 段内地址W |

如果段号大于等于M 那么产生中断

通过F+段表长度\*S找到相应的段基址

检查段内地址W是否大于段长（页内不需要检查，因为页的长度是相同的）

物理地址=段基址+偏移量

分页与分段的区别

页是信息的物理单位，主要实现离散分配，系统管理对用户不可见

段的信息的逻辑单位，主要是方便用户编程，对用户可见，用户编程的时候需要提供段号

页的长度固定 段的长度不固定，与用户编程有关

分页管理中，用户只需要给出逻辑地址空间，是一维的

分段管理中，用户需要给出段名和偏移量，是二维的

分段比分页更容易实现信息的共享和保护

  
分段和分页都是需要访问两次内存，但是分段也可以引入TLB快表 （2）

分页管理方式不容易实现保护是因为，多个段可能在同一个页面里面，这样一个页面只有一部分是需要保护的，另一部分不需要保护（允许多个进程访问），这样就不如段式管理将整个空间都设置为临界资源来的方便

段叶式管理方式



由于段式管理有多少分配多少，所以不产生内部碎片，但是会产生外部碎片

（采用紧凑方式）

765



也就是将之前的段内地址进一步进行拆分，也就是将一个段进行拆分，查分成多个页放到多个长度相等的页帧里面

段号长度决定了进程可以分几个段，页号决定了一个段可以分多少叶，页内偏移量表示一个页有多长（内存块有多大）



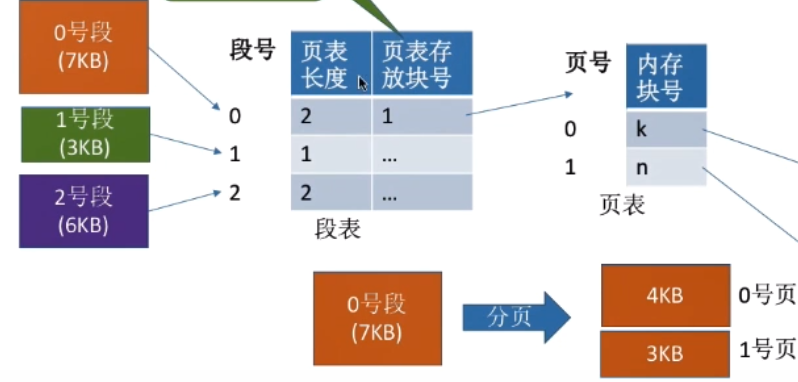
段号占16位，故一共可以分2^16=64K个段

页号占4位，即可以分16个页

页内偏移量占12为，故一个页的大小为2^12B=4KB

对于用户来说 页号 页内偏移量对用户是不可见的，所以用户的编程只用关心段号，段内地址，所以段叶式管理方式是二维的

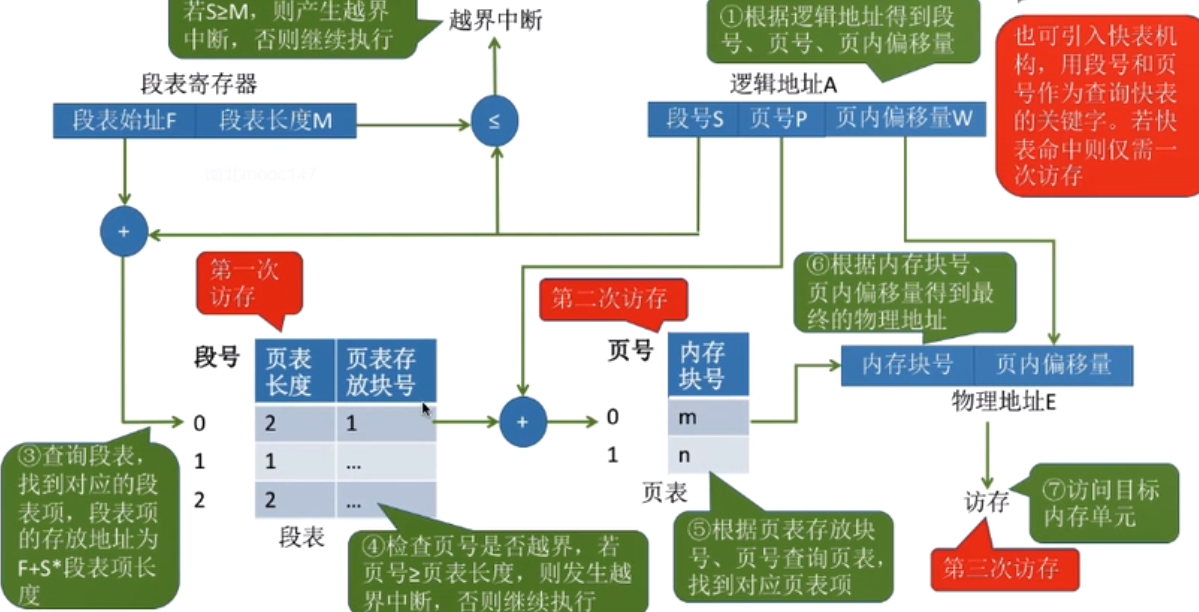






每一个段对应一个段表项，每个段表项由段号，页表长度，页表起始地址组成，每个段表项长度相等，所以段号是隐藏的

需要放完三次内存





# 虚拟存储技术（对内存空间扩充）

传统内存管理

一次性 大作业无法一次性执行



驻留性 一旦装入内存，就会一直留在内存中



内存的物理容量实际上是没有变，但是感觉上内存大多了

虚拟内存的最大容量，是由计算机CPU的寻址范围决定，比如32位就是2^32即可4GB

虚拟内存的实际容量,是min(计算机内存+外存，虚拟内存的最大容量)

多次性 一次性

对换性 驻留性

虚拟性

虚拟内存技术建立在离散分配方式的基础上

全新需求：

1. 在程序执行的过程中，当访问信息不再内从的时候，由操作系统负责将所需信息从外存调入内存，然后执行程序。（请求调页/段）
2. 如果内存空间不够，由操作系统将内存中暂时用不到的信息换到外存

（段/页置换）

请求分页管理方式

对比基本分页管理方式

操作系统需要知道每个页面是否已经调入内存，如果没有调入则需要知道其在外存中的位置

当内存空间不足的时候，要实现页面置换，操作系统需要通过某些指标来决定到底换出哪个界面，有的页面没有被修改过，操作系统也就不需要再浪费时间将其写回外存，可以直接覆盖。

所以请求分页管理方式的页表分为：

内存块号 状态位 访问字段 修改位 外存地址

缺页中断机构

通过请求分页管理页表里的状态位判断页面是否在内存中，如果不在那么产生一个缺页中断，然后由操作系统的缺页中断处理程序处理中断。

这个时候缺页的进程阻塞，调页完成之后将其唤醒

如果内存中有空闲块，那么就将所缺页面装入该块，并修改相应的页表项

如果内存中没有空闲块，那么需要使用页面替换算法进行替换（淘汰一个页面）

（指令产生 属于内中断）

内中断 陷阱 故障（可以恢复，比如缺页中断） 终止（不可恢复的错误造成的结果）

外中断 I/O请求，人工干预

地址变换机构的区别

1. 查找页面对应的页表项的过程中，要判断是否在内存中
2. 没有调入内存，但是又没有空闲块的过程中，需要用替换算法进行替换
3. 调入调出需要修改也表中的新增表项

（如果能在快表中找到的页表项，那么该页面一定在内存中，如果页面被换出，那么TLB中的相应内容被删除）

页面调入之后，不仅要修改慢表，也要讲该界面加入到快表当中

# 请求分页的页面置换算法

最佳置换算法 OPT optional：选择以后不使用/在最长时间内不再被访问的页面

例如 7 0 1 2 0 3 0 4 2 4 0 4 2 1 2 0 1 7 0 1 有三个页面

内1 7 7 7 这里产生缺页，要看目前在内存里的 7 0 1 哪一个在以后的时间内最晚被使用

内2 0 0 可以看到是 7 所以换出7 将2放到内存块1中

内3 1

缺页 f f f

缺页中断之后，不一定发生页面置换，只有在页面已经满了的时候，才发生页面置换

缺页率 = 缺页发生的次数 / 总共访问次数

（只是一种理想的算法，因为操作系统在程序没执行的时候也不知道将来要执行那个页面）

# FIFO先入先出

选择淘汰最早进入内存的页面

把调入内存的页面一次排成一个队列，队列的长度取决于系统为进程分配了多少个内存块

例如：

3 2 1 0 3 2 4 3 2 1 0 4

内1 3 3 3 这时候 队列是 3 <-2<-1 这里面要换出3号 因为最早 序列变成 2->1->0

内2 2 2 然后3号进来替换掉2 即1->0->3

内3 1 然后2进来替换掉1 即0->3->2

缺页 f f f

（Belady 异常 ： 当分配更多的内存块的时候，却也次数不减反增，针对FIFO）

FIFO 效能差

# LRU Least Recently used 最近最久未使用替换算法

替换掉最近最久未使用的页面，实现方法是，在页面对应的页表中，在访问字段记录被访问以来所经历的时间t，淘汰的时候只要淘汰掉t最大的

1 8 1 7 8 2 7 2 1 8 3 8 2 1 3 1 7 1 3 7

内1 1 1 11 1 | 这里开始发生缺页，逆向扫描发现7号最晚出现，所以替换掉7

内2 8 8 8 8 缺页 替换掉8号

内3 7 7

内4 2

缺页

性能好，但是需要硬件支持，比较麻烦，但是性能最接近OPT最佳置换算法

# Clock时钟置换算法（最近未用算法）

简单的时钟置换算法

每个页面设置访问为，所有的在内存中的页面通过指针链接成一个循环队列

当被访问的时候，访问位置位1 如果要淘汰一个页面，则检查访问位是不是为0，如果是0则换出，如果是1则不换出，且置为0，继续访问下一个页面，如果第一轮扫描中所有的页面都是1，那么就进行第二轮扫描（简单始终算法 最多会经过两轮）

e.g.

1 3 4 2 5 6 3 4 7 五个内存块

1 3 4 2 5

访问位 1 1 1 1 1

0 0 0 0 0

6 3 4 2 5

1 0 0 0 0

1 1 1 0 0

1 1 1 7 0

改进型时钟置换算法

（如果页面没有被修改过，那么不需要IO操作写回外存，直接替换 所以还要考虑该页面是否被修改）

增加一个修改位 使用访问位和修改位这样的二元组

将所有可能被置换的页面排成一个循环队列 ，第一轮扫描到一个(0,0)的帧用于替换，本轮扫描不修改任何标志位（淘汰最近没访问 也没修改的页面）

第二轮，如果第二轮扫描失败，则重新扫描，查找第一个(0，1)的帧用于替换，将所有访问过的访问位置位0 （淘汰最近没有被访问但是修改过的页面 ）

第三轮 若第二轮失败，则重新查找第一个(0，0)用于替换，不修改任何标志位

（最近访问过但是没有修改过）

第四轮 若第三轮失败，查找第一个(0，1)用于替换，一定会找到 不修改标志位 最多四轮

（最近访问过，也被修改过 ）

# 驻留集

指的是请求分页管理中分给进程的物理块的集合，在采用了虚拟存储技术的系统中，驻留级的大小与进程的总大小相同

系统应该选择合适的驻留集大小

固定分配：大小不变

可变分配：可变大小

局部置换：缺页的时候只能置换进程自己的物理块

全局置换：缺页的时候进程可以得到其他进程或者操作系统保留的物理块

|  |  |  |
| --- | --- | --- |
|  | 局部置换 | 全局置换 |
| 固定分配 | T | F(全局置换块数必然改变) |
| 可变分配 | T | T |

固定分配局部置换 快数不改变 缺页只能从该进程的块内置换

缺点：不知道分配多少块合适

可变分配全局置换 一开始分配一部分，所以可以从进程以外内存分配 如果空闲物理块用完，那么选择一个未锁定的换出

可变分配，局部置换 缺页的时候只允许换自己的内存 ，并且操作系统动态的分配内存给进程

区别 可变分配全局置换 只要缺页就分配 可变分配局部置换 系统会动态的分配

何时调入页面

预调页策略

一次性调用很多相邻的界面 （进程首次调用，程序员指出）

请求调页策略，运行过程中调入所却的页面 但是开销大

何时调入页面 对换区 快 小 文件区 慢 大

（运行之前 且有足够的对换区空间 先从文件区放到对换区）

缺少对换区，不被修改的数据直接从文件区读入

UNIX 第一次从文件区读 以后就从对换区读取

抖动/颠簸

刚刚换出的界面马上又要换入内存 刚刚换出的内存马上又要换入内存 （进程频繁访问的页面数量高于可用物理块数量，系统大量时间处理幻如幻出）

工作集 某段时间里，进程实际访问页面的集合 （区分驻留集）窗口尺寸 工作集可能会小于窗口尺寸

工作集小 可以分配更小的块 但是驻留级不能小于工作集