目录

[2.1 Linux内存管理的基本框架 1](#_Toc399341366)

[2.2 地址映射的全过程 1](#_Toc399341367)

[2.3 几个重要的数据结构和函数 2](#_Toc399341368)

[2.4 越界访问 4](#_Toc399341369)

[2.5 用户堆栈的拓展 5](#_Toc399341370)

[2.6 物理页面的使用和交换 6](#_Toc399341371)

[2.7 物理页面的分配 7](#_Toc399341372)

[2.8 页面的定期换出 8](#_Toc399341373)

[2.9 页面的换入 10](#_Toc399341374)

[2.10 内核缓冲区的管理 10](#_Toc399341375)

[2.11 外部设备存储空间的地址映射 12](#_Toc399341376)

[2.12 系统调用brk() 12](#_Toc399341377)

[2.13 系统调用mmap() 12](#_Toc399341378)

## 2.1 Linux内存管理的基本框架

Linux内核设计需要先设计一种通用的模型，再分别落实到各种具体的CPU上，具体到页式存管，两层映射机制对于32位也许合理，却无法满足地址宽度大于32位的CPU，因此映射机制为三层PGD PMD PT

对于PAE模式(物理地址扩充功能)，虚存映射为三层模式，其它的i386来说只有两层映射 #ifCONFIG\_X86\_PAE

#include<asm/pgtable-3level.h>

#else

# include<asm/pgtable-2level.h>

#endif

pgtable-2level.h

#define PGDIR\_SHIFT 22

#define PTRS\_PER\_PGD 1024

#define PMD\_SHIFT 22

#define PTRS\_PER\_PMD 1

#define PTRS\_PER\_PTE 1024

Pgtable.h

#define PGDIR\_SIZE (1UL<< PGDIR\_SHIFT) //PGD每个表项所代表的空间为//1\*1022，每个PGD表中有1024个指针，这与PGD位段长度对应210=1024

PMD只有一个表项，且位长度为0，页目录直接指向页面表，跳过页中间目录

虽然系统空间占据每个虚存空间的最高1G字节，在物理的内存中却总是从最低的地址(0)开始，对于内核来说只需要简单的线性映射，0XC0000000就是两者的位移也称为PAGE\_OFFSET定义于page.h中

全局段描述表GDT中对于每个进程至少有两项LDT TSS，由于段寄存器用作GDT表下标的位段宽度为13，也就是GDT有8192个描述项，除去系统开销，尚有8180个表项可用，故理论上系统上最大的进程数为4090

## 2.2 地址映射的全过程

当要将一部分物理空间的内容换出到磁盘中时，在段式存储中要将整个段都换出，而页式存储按页进行，效率要高很多

% objdump -d hello //反汇编一段二进制代码

在elf格式的可执行代码中，ld总是从0x8000000开始安排程序的“代码段”

在Linux中，对于I386，每个段都是从0地址开始的整个4GB虚存空间，虚地址到线性地址的映射保持不变

指向进程页面目录的PGD保存在每个进程的mm\_struct结构中，内核据此通过\_\_pa()函数将其转为物理地址设置CR3，由于是在内核中完成的，此时所有进程都具有相同的页面映射，不会给进程的执行带来不便

系统调用modify\_ldt()用来改变当前进程的局部段描述符，用于在Linux上建立运行Windows的环境，由于Linux中是通过页式存储管理保证安全性的，故系统仍然是安全的。

系统调用vm86()用来在linux上模拟运行DOS软件，即实地址模式软件

## 2.3 几个重要的数据结构和函数

Include/asm-i386/page.h

typedef struct { unsigned long pte\_low;} pte\_t;

typedef struct { unsigned long pmd;} pmd\_t;

typedef struct { unsigned long pgd;} pgd\_t;

当采用32位地址时，pgd\_t pmd\_t pte\_t 实际就是长整数，而采用36位地址时是long long整数。

不直接定义为长整数的原因是可以让gcc在编译时加以更严格的类型检查

页面保护结构typedef struct { unsigned long pgprot;} pgprot\_t; 这与i386MMU页面表项的低12位相对应

页表的指针部分(高20位)与页面保护结构(低12位)通过pgtable.h中定义的宏操作mk\_pte完成

#define \_\_mk\_pte(page\_nr,pgprot) \_\_pte(((page\_nr)<<PAGE\_SHIFT)|pgprot\_val(pgprot))

#define pgprot\_val(x) ((x).pgprot)

#define \_\_pte(x) ((pte\_t){(x)})

页面表项的高20位，对于软件是一个物理页面的序号，这个序号作下标可以从mem\_map找到代表这个物理页面的page数据结构；对于硬件在低位补12个0后就是物理页面的起始地址

宏操作set\_pte()用来把一个表项的值设置到一个页面表项中 见include/asm-i386/pgtable-2level.h #define set\_pte(pteptr,pteval)(\*(pteptr)=pteval)

检测页面表项的内容的宏操作 #define pte\_none(x) (!(x).pte\_low)

#define pte\_present(x) ((x).pte\_low&(\_PAGE\_PRESENT|\_PAGE\_PROTNONE)) 对于软件，页面表项为0表示尚未为这个表项(所代表的虚存映射)建立映射，而如果页面表项不为0，但P标志位为0，表示映射已建立，但是所映射的物理页面不在内存中

根据虚存地址找到对应的物理地址的page结构

#define virt\_to\_page(kaddr) (mem\_map+(\_\_pa(kaddr)>>PAGE\_SHIFT))

由页表项找到目标物理页面的page数据结构

#define pte\_page(x) (mem\_map+((unsigned long)(((x).pte\_low>>PAGE\_SHIFT))))

Index作为page结构的成员：当页面的内容来自一个文件时，index代表该页面在文件中的序号；当页面的内容被换出到交换设备上，但还保留着内容作为缓冲时，index指明页面的去向

物理页面划分为ZONE\_DAM ZONE\_NORMAL两个管理区(若物理地址超过1G，还有第三个管理区ZONE\_HIGHMEM)

为什么DMA使用的页面要单独加以管理

1.DMA使用的页面是磁盘I/O必需的，如果所有的物理页面都分配完了，就无法进行页面和盘区的交换；

2.在I386中页式存储管理是在CPU内部实现的，所以DMA不经过MMU提供的地址映射。这样外部设备就要直接提供访问物理页面的地址，可是有些外设(特别是插在ISA总线上的外设接口卡)在这方面往往有限制，要求用于DMA的物理地址不能过高

3.当DMA所需的缓冲区超过一个物理页面的大小时，要求两个页面在物理上连续，因为此时DMA控制器不能依靠在CPU内部的MMU将连续的虚存页面映射到物理上不连续的页面。所以用于DMA的物理页面要单独加以管理

每个管理区都有一个数据结构，即zone\_struct，其中有一组“空闲区间”(free\_area\_t)队列，由于常常需要成“块”地分配物理空间中连续的多个页面，所以要按“块”的大小加以管理，通常以1,2,22---210来保持，即4M字节

管理区中的offset表示分区在mem\_map中起始页面号，物理页面属于哪一个管理区取决于页面的起始地址

物理页面的page结构通过其成员 struct list\_head list;进入管理区的空闲区链表free\_area\_t free\_area[MAX\_ORDER]中的

CPU访问空间中任何一个地址所需的时间都相同，称为“匀质存储结构”UMA

在NUMA“非均质存储结构”系统中，分配连续的若干物理页面时一般都要求分配在质地相同的区间（称为node，即“节点”）

由于NUMA的引入，每个存储节点至少有两个管理区，page结构不再是全局性的而是从属于具体的节点，节点的pglist\_data结构通过指针node\_next形成一个单链队列，每个结构中的指针node\_mem\_map指向具体节点的page数据结构

节点的pglist\_data结构中数组node\_zones[]就是该节点的最多三个页面管理区，在管理区结构zone\_struct结构中也有一个指向所属节点的指针zone\_pgdats

在节点的结构体中设置了数组node\_zonelists[]，其中有一个zones[]的指针数组，各个元素按特定的顺序指向具体的管理区，这些管理区可以属于不同的节点。

物理空间管理有一个总的物理页面仓库，而虚存空间管理以进程为基础

物理空间从“供”的角度来管理，也就是“仓库还有些什么”；虚存空间从“需”的角度来管理，也就是“我们需要用虚存空间的哪些部分”

物理页面通常使用的变量名为page或map；虚存空间通常使用变量名为vma

虚存区间vm\_start <= vma < vm\_end，区间的划分不仅取决于连续性，还取决于区间的其它属性如访问属性。包含在同一个区间里的所有页面具有相同的访问权限。

由于区间要求所有页面具有相同的访问权限，一个进程的虚存很可能划分为大量区间，由于给定虚拟地址而要找出其所属区间是一个频繁的操作，因此在区间数量大为之建立一个AVL树，方便搜索。这是由find\_vma实现的，若没有建立区间，可以通过insert\_vm\_struct()将新建的虚存区间插入mm\_struct的线性队列或AVL树中

系统调用mmap()可以将一个已经打开的磁盘文件映射到用户空间中

mm\_struct是进程用户空间的抽象。一个进程只使用一个mm\_struct结构，当进程创建一个子进程时，子进程就可能与父进程共享一个mm\_struct结构

## 2.4 越界访问

Page Fault Exception产生可能原因：

1.页面目录项或页表项为空，也就是线性地址与物理地址未建立映射；

2.相应的物理页面不在内存中；

3.指令中规定的访问方式与页面的权限不符

页面异常服务函数do\_page\_fault

1.将映射失败的线性地址放入address中

2.取得当前进程的task\_struct结构，并得到mm\_struct

3.判断异常是否发生在中断或者进程映射还未建立的时候

4.上锁

5.判断映射失败地址是否发生在已经建立的区间，是否越界或者是空洞

6.向进程发出“强制”信号，之后结束

在从中断/异常返回之前，在检查到信号后，会使进程撤销

find\_vma()找到在虚存空间试图找到结束地址大于给定地址的第一个区间，由于堆栈位于用户区间的顶部，因此如果找不到出现映射异常的地址所在区间就应该是超过3G的地址，自然也就是越界了

在用户空间存在两种不同的空洞：第一种是堆栈区以下的空洞，代表供动态分配(通过系统调用brk())而仍未分配出去的空间；第二种是因为一个映射区间被撤销而留下的，或者在建立映射时跳过了一块地址

## 2.5 用户堆栈的拓展

当出现页面出错异常即映射失败时，可能会是一次因越界访问而造成映射失败从而引起进程流产，也可能是导致用户堆栈的扩展的堆栈操作

通常，一次压入堆栈的是4个字节，所以该地址应该是%esp-4，但是i386CPU有一条pusha指令，可以将32个字节(8个32位寄存器的内容)压入堆栈，因此检查的准则是%esp-32

在页面异常服务函数do\_page\_fault发现要对用户堆栈扩展时，首先通过expand\_stack()函数用户堆栈所在虚存区间vma进行拓展，并注意是否超过rlim结构数组中的限制；最后在handle\_mm\_fault()函数中建立页面目录项

页面目录总是在的，相应的目录项也许已经指向一个页面表，此时需要根据给定的地址在表中找到相应的页面表项。或者，目录项也可能是空的，那样的话就需要先分配一个页面表，再在页面表中找到相应的表项。这样，才可以为下面分配物理内存页面并建立映射做准备。准备工作就是在pte\_alloc()函数中完成的

在handle\_mm\_fault()函数中，pte\_alloc()函数分配一个页面表，相应的页面表项总为空。函数handle\_pte\_fault()>do\_no\_page()> do\_anonymous\_page()完成物理内存页面的分配，具体页面表项。

当释放一个页面表时，内核将释放的页面先保存在一个缓冲池中，而先不将其物理页面释放。只有在缓冲池已满的情况下才真的将页面表所占的物理内存页面释放，这样在分配一个页面表时，就可以先看一下缓冲池，这就是get\_pte\_fast()。要是缓冲池已空，就只能通过get\_pte\_kernel\_slow()获得（在物理内存页面用完时需要把内存已占有的页面交换到磁盘上，因此可能执行比较慢）

物理内存页面的分配与文件操作的关系

在虚存结构中有一个vm\_ops指针，里面有一些文件操作的指针，包括物理内存页面的分配。这对于可能的文件共享很有意义。当多个进程将同个文件映射到各自的虚存空间时，内存中通常只要保存一份物理页面。只有当一个进程需要写入该文件时才有必要另外复制一份独立的副本，称为“copy on write”或者COW。这样当通过mmap()将一个虚存空间跟一个已打开文件建立映射后，就可以通过函数调用将对内存的操作转换为对文件的操作。另一方面，物理页面的盘区交换显然也与文件操作有关

函数do\_anonymous\_page()中：如果是读操作，那么由mk\_pte()构筑的映射表项要通过pte\_wrprotect()加以修正，所映射的物理页面总是ZERO\_PAGE即empty\_zero\_page；而如果是写操作，则通过pte\_mkwrite()加以修正，并通过alloc\_page()分配物理页面

当CPU从一次页面错异常处理返回到用户空间时，将会重新执行因映射失败而中途夭折的那条指令，然后才继续执行，这是异常处理的特殊性。

中断和自陷发生时，CPU都会将下一条指令的地址压入堆栈作为返回地址，而异常将夭折的指令本身的地址压入堆栈。

## 2.6 物理页面的使用和交换

页面的交换，是典型的以时间换空间。在要求实时性的系统中，是不宜采用页面交换的，它使程序的执行有了较大的不确定性。

在系统初始化阶段，内核根据检测到的物理内存的大小，为每一个页面都建立了一个page数据结构，形成一个page结构的数组，并使一个全局量mem\_map指向这个数组。同时，又按需要将这些页面拼成物理地址连续的许多内存页面“块”，再根据块的大小建立起若干“管理区”，而在每个管理区中设置一个空闲块队列，以便物理内存页面的分配使用。

当页面在内存时，页面表中的相应表项确定了地址的映射关系；而当页面不在内存时，则指明了物理页面的去向和所在

交换设备的物理页面在内存中也有类似的数据结构swap\_info\_struct，实际上只是一个计数，表示页面是否已经分配，有几个用户共享

数据结构swap\_info\_struct中的指针swap\_map指向一个数组，该数组的每一个无符号短整数即代表盘上（或普通文件中）的一个物理页面，数组的下标决定了页面在盘上或文件中的位置。数组的大小取决于pages，它表示该页面交换设备或文件的大小。设备上（或文件中）的第一个页面，也即swap\_map[0]代表的页面不用于交换，包含该设备的一些信息以及一个表明哪些页面可用的位图。结构中的lowest\_bit和highest\_bit说明文件中从什么地方开始到什么地方为止是供页面交换的。max字段表示该设备或文件中最大的页面号。

Linux内核允许使用多个页面交换设备，所以在内核建立了一个swap\_info\_struct结构(代表一个交换设备或文件)的数组swap\_info。与页面表项pte\_t对应的swp\_entry\_t结构，其中的type字段为swap\_info交换设备数组的下标，表明是哪一个交换设备，offset字段表明在该设备的具体位置，也是swap\_info\_struct结构中swap\_map数组的下标，其中的数字表示有几个用户在使用这个磁盘物理页面。

#define SWP\_TYPE(x) (((x).val>>1)&0x3f) 该函数返回的实际就是页面交换设备的序号，即其swap\_info\_struct结构在swap\_info[]数组中的下标

需要注意的是释放磁盘页面的操作实际上并不涉及磁盘操作，而只是在内存中“账面”上的操作，表明磁盘上页面的使用情况，所以花费的代价很小。

凡是映射到系统空间的页面都不会被换出

内核代码和内核中的全局变量所占的内存页面既不需要经过分配也不需要释放，这部分空间是静态的。除此之外，内核中使用的内存页面也要经过动态分配，但永远保存在内存中

一类是使用完毕便无保存价值，所以立即释放回收

1.内核中通过kmalloc()或vmalloc分配、用作临时性使用和为管理目的而设的数据结构，如vma\_area\_struct。

2.内核中通过alloc\_pages()分配，用作某些临时性使用和管理目的的内存页面

另一类是虽然使用完毕，但是其内容仍有保存价值，这类页面在“释放”后要放入一个LRU队列，经过一段时间缓冲让其“老化”，直到条件不允许才加以回收

1.在文件系统操作中用来缓冲存储一些文件目录结构dnetry的空间

最简单的页面交换策略是：需要是才进行交换；比较积极的方法是在空闲的时候挑选一些页面进行交换

全局的address\_space数据结构swapper\_space，把所有可交换内存页面管理起来，每个可交换内存页面的page数据结构都通过队列头结构list链入其中一个队列

内核在为某个需要换入的页面分配了一个空闲内存页面后，就通过add\_to\_swap\_cache()将其page结构链入相应的队列，其中的函数add\_to\_page\_cache\_locked()中，page结构通过其队列头list链入swapper\_space中的clean\_pages、对应的杂凑队列以及内核全局性的LRU队列active\_list中。

页面的缓冲不光是为页面交换而设的，文件的读/写也要用到这种缓冲机制。通常来自同一个文件的页面就通过一个address\_space数据结构来管理，而代表一个文件的inode数据结构中有一个成分i\_data，那就是一个address\_space数据结构。因此，从某种意义来说，用来管理可交换页面的address\_space数据结构swapper\_space只是一个特例。

## 2.7 物理页面的分配

当需要分配若干内存页面时，用于DMA的内存页面必须是连续的，因为DMA不经过MMU提供的地址映射。

在连续空间UMA结构中只有一个节点contig\_page\_data，而在NUMA结构或不连续空间UMA结构中则有多个。

具体页面的分配\_\_alloc\_pages()

如果要求分配的页面只是单个页面，而且要等待分配完成，又不是用于管理目的，则可以从相应页面管理区的“不活跃干净页面”缓冲队列中回收。当发现可分配页面短缺时，还要唤醒kswapd和bdflush两个内核线程，设法腾出一些内存页面。函数扫描队列中的所有节点的管理区，如果空闲页面在“低水位”以上，就通过rmqueue()来从管理区分配。要是发现管理区中的空闲页面数量降到最低，而且有进程（实际上只能是内核线程kreclaim）在一个等待队列中睡眠，就唤醒让其帮助回收一些页面备用。

要是分配策略中的所有的页面管理区都失败了，一是降低对页面管理区中保持“水位”的要求，二是把缓冲在管理区中的“不活跃干净页面”也考虑进去

还是不行的话，唤醒内核线程kswapd，让它设法换出一些页面。如果分配策略宁可等待，就让系统来一次调度，并让当前进程为其它进程让一下路。这样一来让kswapd有可能被调度执行，二来其它进程也有可能会释放出一些页面

当进程再次调度执行时，如果还是分配不成。就要看谁在要求分配内存页面了

1.如果是分配页面的进程kswapd或kreclaimd，这些进程的PF\_MEMALLOC为1 ，是在执行释放页面的公务，只能“不惜老本”地获取；

2.如果是一般的进程，分配页面失败的原因可能有：一种是可分配页面总量实在太少了；另一种是总量不少，但是所要求的页面大小不能满足。此时可能不少页面在inactive\_clean\_pages或者在inactive\_dirty\_pages队列中，通过page\_launder()把“脏页面洗干净”，通过for循环在各个页面管理区中回收和释放“干净”页面，\_\_free\_page释放页面时会把空闲页面拼装在尽可能大的页面块

函数rmqueue()首先在恰好满足大小的物理内存块的队列中分配，如果不行的话就试试更大的队列中分配，成功的话，通过expand()函数就把分配到的大块中剩余的部分分解成小块而链入相应的队列中

## 2.8 页面的定期换出

linux内核中为页面定期（每1s）换出专门设置kswapd线程分为两部分。第一部分是在发现物理页面短缺的情况下进行，目的在于预先找出若干页面，且将这些页面的映射断开，使物理页面从活跃状态转入不活跃状态，为页面换出作准备。第二部分是每次都要执行的，目的在于把已经处于不活跃状态的“脏”页面写入交换设备，使他们成为不活跃干净继续缓冲，或进一步回收这种页面成为空闲页面

inactive\_shortage()检查潜在的物理页面供应总量是否足够；free\_shortage()检查是否有某个具体管理区中有严重短缺。

do\_try\_to\_free\_pages()函数，首先通过page\_launder()回收不活跃“脏”页面为不活跃“干净”页面；其次如果仍然不够，就会从以下三方面回收：一是在打开文件的过程中要分配和使用代表着目录项的dentry数据结构和代表着文件索引节点的inode数据结构，在文件关闭后往往放在LRU队列中后备，经过一段时间就会占据数量可观的物理页面；二是内核在运行时也需要动态分配许多数据结构，称为“slab”机制；三是断开映射，使活跃页面变为不活跃页面。

page\_launder()函数会检查PG\_reference标志位，page的age以及是否还存在映射用来将不活跃脏队列中的页面恢复到活跃页面中。并且如果所属的页面的address\_space数据结构没有提供页面写出操作，就只好转回page\_active处了。

refill\_inactive ()主要有两件事：一是通过refill\_inactive\_scan()扫描活跃页面队列，试图从中找到可以转入不活跃状态的页面；二是通过swap\_out找出一个进程，然后扫描其映射表，从中找出转入不活跃状态的页面

refill\_inactive\_scan()首先根据priority值决定扫描页面的数量，根据页面是否受到访问，决定再加或减少页面的寿命，如果发现寿命为0且不存在用户空间映射，就将其转入不活跃状态，对于没有转入不活跃的要将其移至队列尾。一般来说，活跃页面队列的使用计数都大于1，而当swap\_out()断开一个页面映射而使其转入不活跃状态时，页面已经转入不活跃队列。所以在我看来refill\_inactive\_scan()函数主要用来根据最近是否受到访问来改变页面的寿命age，swap\_out()函数在断开映射后会将活跃页面自动转入不活跃页面。

每个进程的虚存空间已经建立映射的页面并且在内存中的页面构成“驻内页面集合”，其大小称为rss；mm->rss反映一个进程占用的内存页面数量，而mm->swap\_cnt反映了进程在一轮换出内存页面的努力中尚未受到考察的页面数量。直到所有进程的mm->swap\_cnt为0时，mm->swap\_cnt会重新置为mm->swap\_cnt。

swap\_out函数会通过找到所有进程中mm->swap\_cnt最大的进程来考察，找到的进程会通过swap\_out\_mm()来处理，遍历其虚存空间，并通过try\_to\_swap\_out()函数来试图换出一个页面表项所指的内存页面。

函数try\_to\_swap\_out()通过pte\_page()函数得到页面表项所指物理内存页面的page指针，并将mm->swap\_cnt减1，检查其是否年轻

一个页面是否换出，取决于这个页面是否年轻，每当内存映射机制访问这个物理地址时，就会将页面表项的\_PAGE\_ACCESSED标志位设为1，而该函数的每一次检测都会通过ptep\_test\_and\_clear\_young()函数清零。

如果页面还“年轻”：如果页面还活跃，就通过SetPageReference()将page数据结构中的PG\_referenced设为1，也就是说，将页面表项中表示受到过访问的信息转移到页面的数据结构中；如果页面不在活跃页面队列中，则通过age\_page\_up增加页面可以留下来“以观后效”的时间。

页面有映射却不在活跃队列的原因

在do\_swap\_page()中看到，当因页面异常而恢复一个不活跃页面的映射时，并不立即把它转入活跃页面队列，而把这项工作留给page\_launder()，让其在比较空闲来处理

如果页面的page结构已经在为页面换入/换出而设置的队列中（相应的页面是个普通换入/换出页面），即数据结构swapper\_space()，那么页面的内容已经在交换设备上，只要把映射暂时断开，表示进程已经同意释放这个页面。page\_lanuger()函数会将脏页面写到磁盘上

函数swap\_duplicate()的作用：一是对索引项的内容做一些检验；二是要递增相应盘上页面的共享计数

函数deactivate\_page()有条件的将页面设为不活跃状态，并将页面的page结构从活跃页面转移到不活跃页面队列

物理页面的page结构中有个计数器count，空闲页面的这个计数为0，在分配页面时将其设为1，以后每当页面增加一个“用户”，如建立或恢复一个映射时，就使count加1,。这样，如果计数器的值为2，就说明刚断开的映射已经就是该物理页面的最后一个映射。既然是最后一个映射，这页面当然是不活跃的。

不活跃“脏”页面只有一个，那就是inactive\_dirty\_list；而不活跃“干净”页面队列则有很多，每个页面管理区中都有inactive\_clean\_list队列。

在断开映射后，try\_to\_swap\_out()->page\_cache\_release()->\_\_free\_pages()->put\_page\_testzero()负责将page结构中的count的值减1。

函数try\_to\_swap\_out()只有当mm->swap\_cnt达到0才返回1，正是这样，才使swap\_out\_mm能够依次考察和处理一个进程的所有页面。

Swap\_out()是在一个for循环中调用swap\_out\_mm()的，所以每次调用swap\_out()都会换出若干进程的若干页面，而refill\_inactive()又是在嵌套的while()循环中调用swap\_out()的，一直到系统中可供分配的页面，包括潜在可供分配的页面在内不再短缺时为止。到那时，do\_try\_to\_free\_pages()就结束了。

函数kswapd()和kreclaimd()都将task\_struct结构中的flag字段的PF\_MEMALLOC标志位设为1，表示这两个内核线程都是页面管理机制的维护者。

函数kreclaimd()通过reclaim\_page()扫描各个页面管理区中的不活跃“干净”页面队列，从中回收页面加以释放。

## 2.9 页面的换入

CPU在映射过程中只要看到页面表项或目录项的P标志位为0，就会产生一次“页面异常”，并且认为其余各个位段无意义，至于当页面不在内存中，利用页面表项指向一个盘上页面，那是软件的事。

页面异常处理函数handle\_pte\_fault()首先检查表项中的P标志，看看物理页面是否在内存中，进而通过pte\_none()检查表项是否为空，如果为空，尚未建立映射，就调用do\_no\_page()；反之，如果非空，就调用do\_swap\_page()，从交换设备换入这个页面。

函数do\_swap\_page()首先通过lookup\_swap\_cache()函数检查内存页面是否还留在swapper\_space的换入/换出队列中；由于寻道时间较长，会通过swapin\_readahead()函数多读几个页面进来，如果因为内存空间不足而失败，内核会调度其它进程先运行，再通过read\_swap\_cache()函数读且只读一个页面

无论是swapin\_readahead()还是read\_swap\_cache()在分配内存页面时都会将调用参数gfp\_mak中的\_\_GFP\_WAIT标志位置为1，所以分配不到内存页面时都会自愿暂时礼让，让其它进程先运行。

## 2.10 内核缓冲区的管理

1.一个slab可能由1个，2个，4个……最多32个连续物理页面构成。slab的具体大小因对象的大小而异，初始化时通过计算得出最适合的大小。

2.在每个slab的前端是该slab的描述结构slab\_t。用于同一种对象的多个slab通过描述结构中的队列头形成一条双向链队列。每个slab双向链队列在逻辑上分为三截，第一截是各个slab上所有的对象都分配使用的；第二截部分使用；第三截全部空闲。

3.每个slab上都有一个对象区，这是个对象数据结构的数组，以对象的序号为下标就可得到具体对象的起始地址。

4.每个slab上还有个对象链接数组，用来实现一个空闲对象链。

5.每个slab的描述结构中有一个表明该slab上的第一个空闲对象的字段，这个字段与对象链接数组结合一起形成一条空闲对象链。

6.在slab的描述结构中还有一个已经分配使用的对象的计数器

7.当释放一个对象时，只需要调整链接数组中的相应元素以及slab描述结构中的计数器，并且根据该slab的使用情况而调整在slab队列中的位置（在第几截上）。

8.每个slab的头部有一个小小区域是不用的，称为“着色区”（colouring area）。着色区的大小使slab中的每个对象的起始地址都按高速缓存中的“缓冲行”大小对齐。每个slab都是从一个页面边界开始的，所以本来就自然按高速缓存的缓冲行对齐，而着色区的设置只是将第一个对象的起始地址往后推到另一个与缓冲行对齐的位置。同一个对象的缓冲队列的各个slab的着色区的大小尽可能安排为不同的大小，使得不同slab上同一相对位置的对象的起始地址在高速缓存中错开，改善高速缓存的效率。

9.每个slab最后一个对象之后有一部分是不用的，这一部分对于同一个对象是相同的。

10.一个slab上的所有对象的起始地址都必然是按高速缓存的缓冲行对齐的，否则会在对象的数据结构中增加若干字节使其对齐。

总根cache\_cache是一个对列头结构kmem\_cache\_t结构，用来维持第一层slab队列，其对象都是队列头，并用其维持第二层slab队列，下面都是具体对象的队列

分配了一个kmem\_cache\_t结构以后，kmem\_cahe\_create()就进行一系列的计算，以确定最佳的slab构成。包括：每个slab由几个页面构成，划分成多少个缓冲区（即对象）；slab的控制机构kmem\_slab\_t应该在slab外面集中存放还是就放在每个slab的尾部；每个缓冲区的链接指针应该在slab外面集中存放，还是在slab上与相应的缓冲区紧挨在一起；还有“颜色”的数量等等。

函数alloc\_skb()是具体设备驱动程序对kmem\_cache\_alloc()的包装，在此基础上建立起自己的缓冲区机制，包括一个sk\_buff数据结构的缓冲池，以加快分配数据的速度，并防止因具体驱动程序分配/释放缓冲区不当而引起问题。

函数kmem\_cache\_alloc()中的kmem\_cache\_alloc\_one()函数根据firstnotfull指针判断队列中是否有空闲的slab对象。如果有，就通过kmem\_cache\_alloc\_one\_tail()分配一个空闲对象；否则通过kmem\_cache\_grow()来分配一块新的slab。

位于最高层的是缓冲区的分配，例如alloc\_skb()，具体则是先通过skb\_head\_from\_pool()，从缓冲池。即已经分配的slab块中分配。如果失败，就往下跑一层从slab队列中通过kmem\_cache\_alloc()分配。要是slab队列中已经没有载有空闲缓冲区的slab，那就再往下跑一层，通过kmem\_cache\_grow()，分配若干个页面构筑slab块。

页面page结构中链头list内，原用于队列链接的指针prev，指向页面所属的slab，所以通过宏操作GET\_PAGE\_SLAB就可以得到这个slab指针。

除了上述slab专用缓冲区，内核中还有一个通用缓冲池cache\_sizes，里面根据缓冲区的大小分成若干队列。由kmalloc()函数来分配，实际上也是调用\_kmem\_cache\_alloc()从队列中分配一个缓冲区。

## 2.11 外部设备存储空间的地址映射

函数ioremap()首先对地址进行检查，不能太大，超过32位地址空间的限制，其次必须在high\_memory（系统初始化时，根据检测的物理内存大小设置物理内存地址的上限）之上，即high\_memory<phys\_addr phys\_addr+size<4G，再通过get\_vm\_area()在内核虚存空间队列去寻找。

get\_vm\_area()函数从内存地址空间分配一处虚拟地址

内核专用的内核空间抽象结构是init\_mm，remap\_area\_pages()完成映射的建立

## 2.12 系统调用brk()

mm\_struct结构中有一个成分brk，表示动态分配区当前的底部。系统调用brk()在内核中的实现为sys\_brk()，其中的参数brk，要求不低于代码段的终点（即每个进程所必须的分配好的空间，位于数据段的上方），并且与页面对齐。如果低于老边界，表示释放空间，否则申请分配用户空间。

释放空间：通过do\_munmap()函数来解除映射区间，将要释放的虚拟区间的vm\_struct结构链入free链表中，再通过for循环，通过zap\_page\_range解除若干连续页面的映射。

分配空间：通过find\_vma\_intersecton()检查所要求的那部分空间是否与已经存在的某一区间冲突，再通过vm\_enough\_memory()检查系统是否有足够的空闲内存空间。

## 2.13 系统调用mmap()