## 进程的地址空间：

1. 进程的地址空间由允许进程使用的全部线性地址组成。每个进程所看到的线性地址集合是不同的。内核可以通过增加或删除某些线性地址区间来动态的修改进程的地址空间。
2. 内核通过所有线性区的资源来表示线性地址区间，线性区是由起始线性地址、长度和一些访问权限来描述的。为了效率起见，起始地址和线性区的长度都必须是4096（B）的倍数，以便每个线性区所识别的数据完全填满分配给它的页框（如果长度是4096，即是4KB的倍数，自然能一块一块的充分填满。假设前一个页自然从4096的倍数的起始地址开始，自然填充到4096的倍数的地址结束（按块填充）。下一个地址自然也是从4096的倍数开始）。
3. 与创建、删除线性区相关的系统调用：

brk() : 改变进程堆的大小

execve()：装入一个新的可执行文件，从而改变进程的地址空间

\_exit()：结束当前进程并撤销它的地址空间

fork()：创建一个新进程，并为它创建新的地址空间

mmap()，mmap2()：为文件创建一个内存映射，从而扩大进程的地址空间

mremap()：扩大或缩小线性区

remap\_file\_pages()：为文件创建非线性映射

munmap()：撤销对文件的内存映射，从而缩小进程的地址空间

shmat()：创建一个共享线性区

shmdt()：撤销一个共享线性区

4、IPC：进程间通信

## 内存描述符：

1. 概念：与进程地址空间有关的全部信息都包含在一个叫做内存描述符的数据结构中，结构类型为mm\_struct，进程描述符的mm字段就指向这个结构。
2. 内存描述符中的字段:

类型 字段 说明

struct vm\_area\_struct\* mmap 指向线性区对象的链表头

struct rb\_root mm\_rb 指向线性区对象的红-黑树的根

struct vm\_area\_struct\* mmap\_cache 指向最后一次引用的线性区的描述符的地址

unsigned long(\*)() get\_unmapped\_area 在进程地址空间中搜索有效线性地址区间的方法

void (\*)() unmap\_area 释放线性地址区间时调用的方法

unsigned long mmap\_base 标识第一个分配的匿名线性区或文件内存映射的线性地址

unsigned long free\_area\_cache 内核从这个地址开始搜索进程的地址空间

pgd\_t\* pgd 指向页全局目录

atomic\_t mm\_users 次使用计数器

atomic\_t mm\_count 主使用计数器

int map\_count 线性区的个数

struct rw\_semaphore mmap\_sem 线性区的读/写信号量

spinlock\_t page\_table\_lock 线性区的自旋锁和页表中的自旋锁

struct list\_head mmlist 指向内存描述符链表中的相邻元素

unsigned long start\_code 可执行代码的起始地址

unsigned long end\_code 可执行代码的最后地址

unsigned long start\_data 已初始化数据的起始地址

unsigned long end\_data 已初始化数据的最后地址

unsigned long start\_brk 堆的起始地址

// ……..续 P370

1. 如果内核想确保内存描述符在一个长操作的中间不被释放，就应该增加mm\_users字段而不是mm\_count字段的值。因为mm\_users的增加确保了mm\_count不变为0，即使拥有这个内存描述符的所有轻量级进程全部死亡。
2. mm\_alloc()函数用来获得一个新的内存描述符
3. mmput()函数递减内存描述符的mm\_users字段。如果这个字段变为0，这个函数就释放局部描述符表、线性区描述符及内存描述符所引用的页表，并调用mmdrop()把mm\_count字段减1，如果该字段变为0，就释放mm\_struct数据结构。

## 内核线程的内存描述符：

1. 内核线程仅运行在内核态，因此，它们永远不会访问低于TASK\_SIZE(等于PAGE\_OFFSET，通常为0xc0000000)的地址。与普通进程相比，内核线程不用线性区，因此，内存描述符的很多字段对内核线程是没有意义的。
2. 为了避免无用的TLB和高速缓存刷新，内核线程使用一组最近运行的普通进程的页表。结果，在每个进程描述符中包含了两种内存描述符指针：mm和active\_mm。
3. 进程描述符中的mm字段指向进程所拥有的内存描述符，而active\_mm字段指向进程运行时所使用的内存描述符。对于普通进程而言，这个两个字段存放相同的指针。但是，内核线程不拥有任何内存描述符，因此，它们的mm字段总是为NULL。当内核线程得以运行时，它的active\_mm字段被初始化为前一个运行进程的active\_mm值。
4. 然而，事情有点复杂。只要处于内核态的一个进程的为“高端”线性地址（高于TASK\_SIZE）修改页表项，那么，它就也应当更新系统中所有进程页表集合中的相应表项。事实上，一旦内核态的一个进程进行了设置，那么，映射应该对内核态的所有其他进程都有效。触及所有进程的页表项集合是相当费事的操作。因此，Linux采用一种延迟操作。
5. 每一个高端地址必须被重新映射时，内核就更新目录在swapper\_pg\_dir主内核页全局目录中的常规页表集合。这个页全局目录被主内存描述符的pgd字段所指向，而主内存描述符存放于Init\_mm变量。P372

## 线性区：

1. Linux通过类型为vm\_area\_struct的对象实现线性区，字段如下: P373

类型 字段 说明

struct mm\_struct\* vm\_mm 指向线性区所在的内存描述符

unsigned long vm\_start 线性区内的第一个线性地址

unsigned long vm\_end 线性区之后的第一个线性地址

struct vm\_area\_struct\* vm\_next 进程链表中的下一个线性区

struct vm\_area\_struct\* vm\_page\_prot 线性区中页框的访问许可权

unsigned long vm\_flags 线性区的标志

struct rb\_node vm\_rb 用于红-黑树的数据

union shared 链接到反映射所使用的数据结构

struct list\_head anon\_vma\_node 指向匿名线性区链表的指针

struct anon\_vma\* anon\_vma 指向anon\_vma数据结构的指针

struct vm\_operations\_struct\* vm\_ops 指向线性区的方法

unsigned long vm\_pgoff 在映射文件中偏移量。对匿名页，它等于0或vm\_start / PAGE\_SIZE

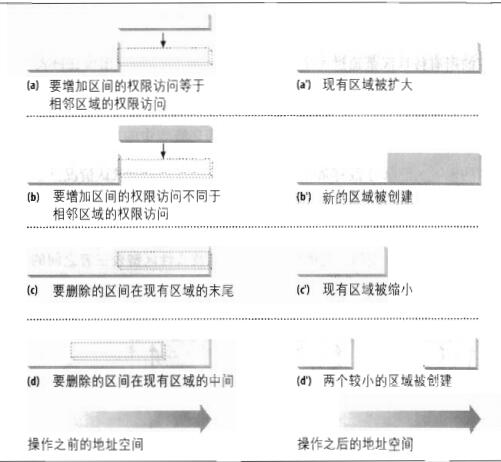
struct file\* vm\_file 指向映射文件的文件对象

void\* vm\_private\_data 指向内存区的私有数据

unsigned long vm\_truncate\_count 释放非线性地址文件存映射中的一个线性地址空间区间时使用。

注意：每个线性区描述符表示一个线性地址空间。vm\_start字段包含区间的第一个线性地址。vm\_end字段包含区间之外的第一个线性地址。vm\_end\_vm\_start表示线性区的长度。vm\_mm字段指向拥有这个区间的进程的mm\_struct内存描述符。

1. 进程所拥有的线性区从来不重叠，并且内核尽力把新分配的线性区与紧邻的现有线性区进行合并。如果两个相邻区的访问权限相匹配，就能把它们合并在一起。
2. 如下图所示，当一个新的线性区间加入到进程的地址空间时，内核就检查一个已经存在的线性地址是否可以扩大（情况a）。如果不能，就创建一个新的线性区。类似的，如果从进程的地址空间删除一个线性地址区间，内核就要调整受影响的线性区的大小（情况c）。有些情况下，调整大小迫使一个线性区被分成两个更小的部分。



1. vm\_ops字段指向vm\_operations\_struct数据结构，该结构中存放的是线性区的方法。只有4种方法可应用于UMA系统。

open：当线性区增加到进程所拥有的线性区集合时调用

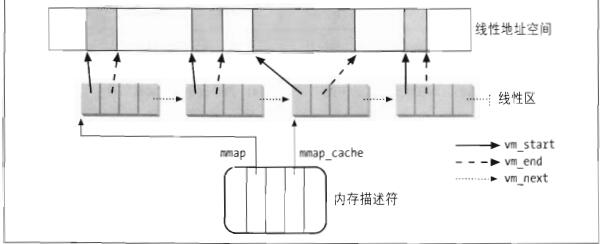
close：当从进程所拥有的线性区集合删除线性区时调用

nopage：当进程试图访问RAM（内存）中不存在的一个页但该页的线性地址属于线性区时，由缺页异常处理程序调用

populate：设置线性区的线性地址（预缺页）所对应的页表项时调用。主要用于非线性文件内存映射。

## 线性区数据结构：

1. 进程所拥有的所有线性区是通过一个简单的链表连接在一起的。出现在链表中的线性区是按内存地址的升序排列的。不过，每两个线性区可以由未使用的内存地址区隔开。每个vm\_area\_struct元素的vm\_next字段指向链表的下一个元素。内核通过进程的内存描述符mmap字段来查找线性区，其中，mmap字段指向链表中的第一个线性区描述符。
2. 内存描述符的map\_count字段存放进程所拥有的线性区的数目。默认情况下，一个进程可以最多拥有65536个不同的线性区。
3. 下图显示了进程的地址空间、它的内存描述符及线性区链表之间的关系：



1. Linux2.6把线性区描述符（vm\_area\_struct）存放在叫做红-黑树的数据结构中。在红-黑树中，每个元素（或节点）通常有两个孩子：左海子和右孩子。树种元素被排序。对每个节点N，N的左子树上的所有元素都排在N之前，相反，N的右子树上的所有元素都排在N之后。节点的关键字被写入节点内部。此外，红-黑树必须满足下列4条规则：
2. 每个节点必须为黑或为红
3. 树的根必须为黑
4. 红节点的孩子必须为黑
5. 从一个节点到后代子节点的每个路径都包含相同数量的黑节点。当统计黑节点个数时，空指针也算作黑节点。
6. 上述这四条规则确保具有n个内部节点的任一红-黑树高度最多为2xlog（n+1）。在红-黑树中搜索一个元素因此变得非常高效，因为其操作的执行时间与树大小的对数成比例。在红-黑树中插入和删除一个元素也是高效的，因为算法能很快的遍历树以确定插入元素的位置或删除元素的位置。任何新节点必须作为一个叶子插入并着成红色。如果插入以后违背了上述规则，就必须移动或重新着色树的几个节点。

## 线性区访问权限：

1. 每个线性区都有一组连续的页所构成。
2. 线性区标志 P378
3. 同一线性区所有页标志的初值必须一样。页表标示的初值被存放在vm\_area\_struct描述符的vm\_page\_prot字段中，当增加一个页时，内核根据vm\_page\_prot字段的值设置相应页表项中的标志。

## 线性区的处理：

1. find\_vma()：查找给定地址的最邻近区。它查找线性区的vm\_end字段大于addr的第一个线性区的位置。**P380**
2. find\_vma\_prev()和find\_vma()类似，不同的是它把函数选中的线性区的前一个线性区的线性描述符赋给参数ppre。
3. find\_vma\_prepare()确定新叶子节点在与给定线性地址对应的红-黑树中的位置，并返回前一个线性区的地址和要插入的叶子节点的父节点的地址。
4. find\_vma\_intersection()：查找一个与给定的地址区间相重叠的第一个线性区。mm参数指向进程的内存描述符，而线性地址start\_addr和end\_addr指定这个区间。
5. get\_unmapped\_area()搜查进程的地址空间以找到一个可以使用（空闲，未被使用）的线性地址区间。len参数指定区间的长度，而非空的addr参数指定从哪个地址开始查找。查找失败返回错误码-ENOMEM。
6. insert\_vm\_struct()：向内存描述符插入一个线性区。
7. \_\_vma\_unlink()：从内存描述符的中线性区链表和红-黑树中删除vma，如果mm->mmap\_cache字段指向刚被删除的线性区，则还要对mm->mmap\_cache进行更新。

## 分配、释放线性地址区间：

1. do\_mmap()：函数为当前进程创建并初始化一个新的线性区。分配成功之后，可以把这个新的线性区与进程已有的其他线性区进行合并。P384
2. do\_munmap()：从当前进程的地址空间中删除一个线性地址区间。P389
3. split\_vma()函数：把线性地址区间交叉的线性区划分为两个较小的区，一个在线性地址区间的外部，另一个在区间的内部。P390
4. unmap\_region()：遍历线性区链表并释放它们的页框。

## 缺页异常处理程序：

1. 首先了解什么是缺页异常：CPU通过地址总线可以访问连接在地址总线上的所有外设，包括物理内存、IO设备等等。但从CPU发出的访问地址是一个虚拟地址，由MMU将虚拟地址转化成物理地址，再从地址总线上发出。MMU上这种虚拟地址和物理地址的转化关系是需要创建的，并且MMU还可以设置这个物理页是否可以进行写操作，当没有创建一个虚拟地址到物理地址的映射，或者，创建了这样的映射，但那个物理页不可写的时候，MMU将会通知CPU产生了一个缺页异常。
2. do\_page\_fault()函数是80x86上的缺页异常处理程序。P391 395

## 请求调页：

1. 概念：如果被访问的页不存在，也就是说，这个页还没被存放在任何一个页框中，那么，内核分配一个新的页框并适当地初始化。一种动态内存分配技术，它把页框的分配推迟到不能再推迟为止，也就是说，一直推迟到要访问的页不在RAM中时为止，由此引发一个缺页异常。
2. 请求调页技术的原因：程序开始运行的时候，并不访问其他地址空间中的全部地址；事实上，有一部分地址也许永远不被进程使用。此外，程序的局部性原理保证了在程序执行的每个阶段，真正引用的进程页只有一小部分，因此临时用不着的页所在的页框可以由其他进程其他进程来使用。因此，对于全局分配（一开始就给进程分配所需要的页框，直到程序结束才释放这些页框）来说，请求调页是首选的，因为它增加了系统中空闲页框的平均数，从而更好地利用空闲内存。
3. 缺点：增加系统额外的开销：由请求调页所引发的每个缺页异常必须由内核处理，这将兰妃CPU的时钟周期。辛运的是，局部性原理保证了一旦进程开始在一组页上运行，在接下来相当长的一段时间内它会一直停留在这些页上而不去访问其他的页。
4. 被访问的页不在主存中，其原因或者是进程从没访问或该页，或者是内核已经回收了相应的页框。在这两种情况下，缺页处理程序必须为进程分配新的页框。P401

## 写时复制：

1. 概念：父进程和子进程共享页框而不是复制页框。然而，只要页框被共享，它们就不能被修改。无论父进程还是子进程何时试图写一些共享的页框，就产生一个异常，这是内核就把这个页复制到一个新的页框中并标记为可写。原来的页框仍然是写保护的：当其他进程试图写入时，内核就检查写进程是否是这个页框的唯一属主，如果是，就把这个页框标记为对这个进程是可写的。
2. 页描述符的\_count字段用于跟踪共享相应页框的进程数目。只要进程释放一个页框或者在它上面执行写时复制，它的\_count字段就减少。当\_count变为-1时，这个页框才被释放。
3. handle\_pte\_fault()实现写时复制

## 创建和删除进程的地址空间：

1. 当创建一个新的进程时内核调用copy\_mm()函数，这个函数通过建立新进程的所有页表和内存描述符来创建进程的地址空间。
2. 通常，每个进程都有自己的地址空间，但是轻量级进程可以通过调用clone（）函数来创建。这些轻量级进程共享同一地址空间，也就是说，运行它们对同一组页进行寻址。
3. 轻量级进程使用进程的地址空间。Linux实现轻量级进程很简单，即不复制父进程的地址空间。创建轻量级的进程比创建普通进程相应要快得多，而且只要父进程和子进程很谨慎地协调它们的访问，就可以认为页的共享是有益的。P406
4. 内核调用exit\_mm()函数释放函数的地址空间。P408

## 堆的管理：

1. 每个Unix进程都拥有一个特殊的线性区，这个线性区就是所谓的堆（heap），堆用于满足进程的动态内存请求。内存描述符的start\_brk与brk字段分别限定了这个区的开始地址和结束地址。
2. malloc(size)：请求size个字节的动态内存。如果分配成功，就返回所分配单元第一个字节的线性地址。
3. calloc(n, size)：请求含有n个大小为size的元素的一个数组。如果分配成功，就把数组元素初始化为0，并返回第一个元素的线性地址。
4. realloc(ptr, size)：改变由前面的malloc()或calloc()分配的内存区字段的大小。
5. free(addr)：释放有malloc或calloc()分配的起始地址为addr的线性区。
6. brk(addr)：直接修改堆的大小。addr参数指定current->mm->brk的新值，返回值是线性区新的结束地址。
7. sbrk(incr)：类似于brk()，不过其中的incr参数指定是增加还是减少以字节为单位的堆大小。