# 第4章 虚拟内存管理

现代处理器通常具备几种工作状态，例如内核态、用户态等。处理器处于不同的工作状态，具有不同的权限，即访问处理器资源的权限，如内存空间、寄存器等。一般内核态具有最高的权限，可以访问处理器中所有的资源，用户态权限较低，只能访问处理器部分资源。

这里我们只关心处理器各工作状态下对内存的访问。MIPS32处理器将虚拟地址空间分成上下两部分，用户态时处理器只能访问地址空间的下半部分（通过页表映射访问），不能访问上半部分地址空间。内核态时可以访问所有的虚拟地址空间。

Linux内核将处理器虚拟地址空间划分成内核空间和用户空间，内核空间位于上部，用户空间位于下部。内核镜像加载到处理器虚拟地址空间的上半部分，在处理器内核态下运行。用户程序加载到虚拟地址空间的下半部分，在用户态下运行。

处理器指令使用的是虚拟地址，而真实的指令和数据保存在物理内存中。虚拟地址到物理地址需要一个转换，称之为映射。映射为指令、数据在物理内存中的存放提供了便利，也就是说指令、数据可以存放到任意的物理内存中，只需要修改映射关系即可。这也是现代操作系统运行的基础。

地址映射由处理器中的MMU（内存管理）单元、内存页表及相关软件实现。内存页表保存的是虚拟地址和物理地址对（页编号），即虚拟地址对应的物理地址。处理器访问相应虚拟地址时，由软件（异常处理程序）或硬件将内存页表项添加到MMU中，MMU根据加载的页表项，将虚拟地址转换成物理地址，输出到处理器地址总线上。

内核地址空间包括直接映射区、IO映射区、间接映射区等。MIPS32体系结构中，直接映射区和IO映射区不需要使用页表，直接线性映射到物理内存底端。间接映射区需要通过（内核）页表转换，可映射到任意物理内存。

用户地址空间总是通过页表建立映射，可映射到任意物理内存。

前面介绍物理内存管理时，我们知道内核按页对物理内存进行管理。在建立映射时，也是按页进行的。每个页表项代表一页，页表项中是虚拟页号与物理页帧号对，以实现虚拟内存到物理内存的映射。

虚拟内存管理的本质是从伙伴系统获取物理内存，建立物理内存与内核或用户虚拟内存之间的映射关系，以使内核或进程有内存可用。建立映射就是从伙伴系统分配物理页帧，将页帧号写入内核或用户进程对应的页表项，处理器访问对应的虚拟内存时，会将页表项导入MMU，以实现地址的转换。解除映射就是清除内核或用户进程对应的页表项，断开虚拟内存与物理内存的映射关系，释放物理内存至伙伴系统。

虚拟内存管理之所以复杂的原因是物理内存不足和虚拟内存需求巨大之间存在矛盾。假如物理内存的数量是无限的，创建进程时内核就直接分配4GB的物理内存供其永久使用，直至进程退出（或主动释放），那么虚拟内存的管理就简单多了。但是，现实就是物理内存总是有限的，进程数在不断增加，进程所需物理内存在不断增加，物理内存永远是不够用的。进程虚拟内存与物理内存之间的映射不是固定的，或者说虚拟内存不是总是能映射到物理内存，内核需要将有限的物理内存在进程之间来回倒腾，以保证进程们都能正常地运行。

内核本身使用的物理内存并不是很多，而且内核代码/数据使用频繁，所以分配给内核使用的物理内存由其永久使用，除非内核主动释放。在创建进程时，内核并没有为其分配物理内存，而只是创建了管理进程虚拟地址空间的数据结构，进程在访问没有建立映射的虚拟地址时将产生缺页异常，在异常处理程序中建立虚拟内存与物理内存之间的映射。进程退出时解除映射（或主动解除），物理内存释放回伙伴系统。除此之外，分配给进程的物理内存，在物理内存紧张时，页面回收机制在进程运行期间也可以视情对其使用的内存进行回收（释放回伙伴系统）。

本章将介绍虚拟地址到物理地址转换的机制，内核地址空间的布局以及映射的建立和解除，以及用户进程地址空间管理和映射的建立及解除，页面回收机制将在第11章中再做介绍。

## 4.1页表

处理器访问内核地址空间间接映射区和用户进程地址空间时，虚拟地址需要通过页表项映射到物理地址。内核及用户进程的页表保存在内存中，处理器的MMU中包含一个（或多个）TLB（地址转换缓存），TLB中缓存了部分内核或用户进程页表项，以加快地址转换的速度。

程序产生的虚拟地址将在TLB中查找是否有匹配且可用的页表项，如果有则可以直接转换，获得物理地址，否则需要到内存页表中去查找虚拟地址对应的页表项，填充至TLB后，再进行转换。如果内存页表中的页表项也不存在或不可用，则需要为虚拟页建立正确的映射，生成页表项，写入内存页表和TLB，然后才能进行地址转换。

本节先介绍处理器MMU实现地址转换的机制，然后介绍内核对内存页表的组织和管理。

### 4.1.1访问内存

前面第3章介绍过，内核按页对物理内存进行管理。虚拟地址到物理地址的转换也是按页进行的，一个虚拟页对应一个物理页，也可以多个虚拟页对应同一个物理页。记录虚拟页与物理页映射关系的数据结构称为页表项。页表是页表项的集合（数组）包含进程整个虚拟地址空间与物理内存的映射关系，保存在内存中。MMU中的TLB是内存页表的缓存，CPU进行地址转换时，在TLB中查找匹配的项，若未找到则到内存页表中查找对应页表项，填充至TLB中后再进行转换。

#### 1地址转换机制

处理器将间接映射的虚拟地址转换成物理地址的过程如下图所示。程序产生的虚拟地址划分成两部分，一部分是虚拟页号VFN，另一部分是页内偏移量OFFSET。如果页大小配置成4KB，则低12位表示页内偏移量，高20位表示虚拟页号。虚拟页号与物理页号对应，位数相同。

处理器MMU根据虚拟地址页号，在TLB中查找是否有匹配且可用的页表项，如果有则用匹配项内保存的物理页号PFN替换虚拟地址页号VFN，页内偏移量OFFSET保持不变，从而组合成物理地址，输出到地址总线访问内存。

如果TLB中没有匹配的项或匹配的项不可用，则会触发处理器TLB异常，由硬件或软件（异常处理程序）查找内核/进程在内存中的页表，找到虚拟地址对应的页表项填充至TLB后，再进行转换。



如果内存页表中的表项也不存在或不可用，则TLB异常处理程序则需要为虚拟页建立正确的到物理页的映射，然后生成页表项，写入内存页表和TLB，随后CPU才能进行地址转换。

在地址转换过程中页内偏移量是不变的，所以TLB（内存页表）中的页表项页内偏移量位域（bit[11..0]）可用于其它用途，一般主要是用于标注页的访问属性，如：可读、可写、脏页等。内存页表中页表项的结构与TLB内页表项的结构可能有所不同，在TLB重填过程中完成两者之间的转换。MIPS处理器由软件（异常处理程序）完成TLB页表项的重填，也有的处理器架构由硬件实现页表项重填。

#### 2内存页表模型

处理器MMU中TLB是内存页表项的缓存，用于实现地址转换。各体系结构中TLB页表项格式不尽相同，这由处理器硬件决定。内存页表项格式可能与TLB中页表项格式有所不同，这由体系结构相关的内核代码决定，内存页表项到TLB页表项的转换由硬件或软件（异常处理程序）实现。

内存页表是一个页表项数组，虚拟页编号是数组项索引值，每个数组项代表一个页的映射关系，保存了映射的页帧号和访问标记。MIPS32体系结构中，内核地址空间和用户进程地址空间分别为2GB，假设页大小为4KB，则内核和进程地址空间分别需要500K条页表项，每条页表项占用4个字节，则内核和每个进程页表需要占用2MB的空间，并且这2MB的空间必须是连续的！这几乎是不可能实现的。

内核/用户进程地址空间并不是连续被使用的，也不一定会用到整个地址空间，也就是说地址空间的使用是离散的，因此没有必要为内核/用户进程的整个地址空间建立页表项。为此内核采用了分级页表的思想，如下图所示：



上图中假设进程分别使用了地址空间的底端和高端两个内存区域，中间的未使用。分级页表的思想是先将地址空间按比较大的单位进行划分（如4MB），再对划分的区域按页建立页表（或按更小的大小划分）。对未使用的地址空间不需要创建页表，而只对使用了的地址空间创建页表，高一级的页表项保存的是低一级页表的基地址。如上图中所示，假设使用了两级页表，一级页表中每个表项代表4MB空间（共1K个条表项），表项内容是二级页表的基地址，二级页表保存的是真正的页表项。搜索页表项时，先用虚拟地址高位域搜索一级页表，找到对应表项，再从其指示的二级页表中搜索页表项。分级页表即可以不需要分配过大的连续内存用于保存页表，又可以省去未使用地址空间的页表。分级页表其实就是将原本连续的页表拆分成小段，未使用地址空间的页表可以省略。

内核定义了四级页表的模型，如下图所示，四级页表分别是全局页表PGD，上层页表PUD，中间页表PMD和直接页表PTE。



越高级的页表，页表项代表的内存区域越大，越低级页表项代表的内存区越小，最后的PTE页表项表示内存页。由于进程通常只使用虚拟地址空间的一小部分，因此除全局页表外，其它各级页表都是按需创建的，进程需要使用某段地址空间时再为其创建各级页表，否则不创建。

四级页表模型将虚拟地址划分成五段，每一段用于索引本级页表项，最后一段为页内偏移地址。除PTE页表项外，其余各级页表项内保存的是下一级页表的起始地址。

进程结构中保存了全局页表基地址，地址转换中没有在TLB中找到匹配的表项时，将在进程页表中查找到对应的页表项填充至TLB，然后再进行转换。查找页表项是按级进行的，从高到低将虚拟地址位域当成索引值，依次在各级页表中搜索相应的表项，最后搜索到的PTE页表项，将填充至TLB。

### 4.1.2 PGD页表

内核定义了四级页表的模型，但是对于32位的系统并不需要使用如此复杂的页表模型。MIPS32体系结构只使用了两级页表，PUD、PMD合并到了PGD中，也就是说只使用了PGD和PTE两级页表。

内核在/arch/mips/include/asm/page.h头文件内定义了以下宏：

#ifdef CONFIG\_PAGE\_SIZE\_4KB

#define **PAGE\_SHIFT** **12**

#endif

...

#define PAGE\_SIZE (\_AC(1,UL) << PAGE\_SHIFT) /\*页大小\*/

#define PAGE\_MASK (~((1 << PAGE\_SHIFT) - 1)) /\*屏蔽页内偏移量（用于取页帧号）\*/

PGDIR\_SHIFT宏定义在/arch/mips/include/asm/pgtable-32.h头文件内：

#define **PGDIR\_SHIFT** (2 \* PAGE\_SHIFT + PTE\_ORDER - PTE\_T\_LOG2) /\*22\*/

#define **PGDIR\_SIZE**  (1UL << PGDIR\_SHIFT) /\*PGD页表项表示的内存区大小（4MB）\*/

#define PGDIR\_MASK (~(PGDIR\_SIZE-1))

#define \_\_PGD\_ORDER (32 - 3 \* PAGE\_SHIFT + PGD\_T\_LOG2 + PTE\_T\_LOG2) /\*0\*/

#define **PGD\_ORDER**  (\_\_PGD\_ORDER >= 0 ? \_\_PGD\_ORDER : 0) /\*PGD页表阶数，0\*/

PTE\_ORDER宏表示PTE页表占用内存的阶数，定义为0，即占用1个页帧。

PTE\_T\_LOG2宏定义在/arch/mips/include/asm/pgtable.h头文件：

#define PGD\_T\_LOG2 (\_\_builtin\_ffs(sizeof(pgd\_t)) - 1) /\*2\*/

#define PTE\_T\_LOG2 (\_\_builtin\_ffs(sizeof(pte\_t)) - 1) /\*2\*/

\_\_builtin\_ffs(x)表示计算整数x第一个为1的比特位位置（编号从1开始），如：x=1，返回1，x=2，返回2。因此PTE\_T\_LOG2 宏的值为\_\_builtin\_ffs(4)-1结果为2。

由PGD\_T\_LOG2值可计算得PGDIR\_SHIFT=22，同理PGD页表阶数PGD\_ORDER为0，表示PGD页表占1个页帧。

在/arch/mips/include/asm/pgtable-32.h中包含了体系结构无关的/include/asm-generic/pgtable-nopmd.h头文件，后者又包含/include/asm-generic/pgtable-nopud.h头文件，在这两个头文件内定义了以下宏：

#define **PUD\_SHIFT PGDIR\_SHIFT**

#define PTRS\_PER\_PUD 1 /\*PUD页表中页表项数目\*/

#define PUD\_SIZE (1UL << PUD\_SHIFT)

#define PUD\_MASK (~(PUD\_SIZE-1))

#define  **PMD\_SHIFT** **PUD\_SHIFT**  /\*PUD页表中页表项数目\*/

#define PTRS\_PER\_PMD 1

#define PMD\_SIZE (1UL << PMD\_SHIFT)

#define PMD\_MASK (~(PMD\_SIZE-1))

实际上PUD、PMD页表就是PGD页表项，两级页表的组织结构如下图所示：



#### 1创建PGD页表

内核在/arch/mips/include/asm/page.h头文件内定义了PGD页表项数据结构：

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

#define pgd\_val(x) ((x).pgd) /\*PDG页表项数值\*/

#define \_\_pgd(x) ((pgd\_t) { (x) } )  /\*将整型数x转化成PGD页表项类型\*/

PGD页表项数据结构内只包含一个无符号整型数，用于保存下级页表的起始地址，使用两级页表时PUD、PMD页表项结构定义成与pgd\_t相同。对PGD页表项的操作需要通过接口函数进行，不能直接操作，后面再介绍操作函数。

内核自身及每个进程/线程都具有一个全局页表，表示进程的task\_struct结构体内mm\_struct（地址空间结构）成员具有指向全局页表的指针成员：

struct mm\_struct{

...

**pgd\_t \* pgd**; /\*指向PGD页表\*/

...

}

内核自身地址空间实例init\_mm静态定义在/mm/init-mm.c文件内：

struct mm\_struct init\_mm = {

...

.pgd = **swapper\_pg\_dir**, /\*全局页表地址，在链接文件内定义\*/

...

};

内核自身全局页表保存在内核链接文件的指定段内，swapper\_pg\_dir表示指定段的起始地址。

MIPS32体系结构将虚拟地址空间低2GB的空间定义为用户空间，高2GB的空间定义为内核空间。内核页表的高半部分用于内核地址空间映射（部分映射），低半部分PGD页表项指向无效的PTE页表，因为内核没有用户地址空间。

用户进程在用户空间运行，它们通过系统调用进入内核，内核只有一个，所以所有进程页表的高半部分与内核页表的高半部分相同（PGD页表项指向相同的PTE页表）。

在创建进程时需要为进程创建全局页表。进程全局页表的高半部分与内核页表的高半部分内容相同，低半部分中PGD页表项初始化为指向无效的PTE页表（尚未建立映射）。

创建进程全局页表的函数为pgd\_alloc()，定义在/arch/mips/include/asm/pgalloc.h头文件：

static inline pgd\_t \*pgd\_alloc(struct mm\_struct \*mm)

{

pgd\_t \*ret, \*init;

ret = (pgd\_t \*) **\_\_get\_free\_pages**(GFP\_KERNEL, PGD\_ORDER); /\*从伙伴系统分配页帧\*/

if (ret) {

**init =** **pgd\_offset(&init\_mm, 0UL)**; /\*获取内核页表基地址\*/

**pgd\_init**((unsigned long)ret); /\*初始化进程PDG页表低半部分指向无效的PTE页表\*/

/\*/arch/mips/mm/pgtable-32.c\*/

**memcpy(ret + USER\_PTRS\_PER\_PGD, init + USER\_PTRS\_PER\_PGD,**

**(PTRS\_PER\_PGD - USER\_PTRS\_PER\_PGD) \* sizeof(pgd\_t))**;

/\*复制内核页表高半部分（内核空间映射）至进程页表\*/

}

return ret;

}

pgd\_alloc()函数首先调用伙伴系统分配函数为全局页表分配内存，然后调用pgd\_offset()函数获取内核页表的起始地址，最后将进程页表低半部分页表项指向无效PTE页表，高半部分PGD页表项内容复制内核PGD页表项的内容，即指向相同的PTE页表（所有进程共用内核空间页表）。

pgd\_offset(mm, addr)函数定义在/arch/mips/include/asm/pgtable-32.h头文件，用于查找在mm表示的地址空间中，包含addr地址对应的PGD页表项，返回PGD页表项指针：

#define **pgd\_offset(mm, addr)** ((mm)->pgd + pgd\_index(addr))

#define pgd\_index(address) (((address) >> PGDIR\_SHIFT) & (PTRS\_PER\_PGD-1)) /\*页表项索引值\*/

内核在/arch/mips/mm/init.c文件内定义了无效的PTE页表（页表项全为0，表示无效）：

**pte\_t invalid\_pte\_table[PTRS\_PER\_PTE] \_\_page\_aligned\_bss;**

pgd\_init(unsigned long page)函数定义在/arch/mips/mm/pgtable-32.c文件内，用于设置page（地址值）表示的PGD页表低半部分的页表项指向无效的invalid\_pte\_table页表。

pgd\_alloc(mm)函数最终创建的新PGD页表如下图所示：



内核在创建进程时，将调用mm\_init()函数复制父进程地址空间，函数内会调用pgd\_alloc()函数为子进程创建PGD页表并初始化，随后会复制父进程PGD页表内容至新进程PGD页表。

内核在/arch/mips/include/asm/pgalloc.h头文件内定义了释放PGD页表的函数：

static inline void pgd\_free(struct mm\_struct \*mm, pgd\_t \*pgd)

{

free\_pages((unsigned long)pgd, PGD\_ORDER); /\*伙伴系统释放页函数，pgd为全局页表基地址\*/

}

#### 2页表项操作

前面介绍了PGD页表的创建、初始化，以及释放操作，下面介绍PGD页表项的操作函数。

**■设置全局页表项**

内核在/include/asm-generic/pgtable-nopud.h头文件定义了设置PGD页表项值的操作：

#define  **set\_pgd**(pgdptr, pgdval) set\_pud((pud\_t \*)(pgdptr), (pud\_t) { pgdval })

在/include/asm-generic/pgtable-nopmd.h头文件中定义了设置PUD页表项的操作：

#define set\_pud(pudptr, pudval) set\_pmd((pmd\_t \*)(pudptr), (pmd\_t) { pudval })

在/arch/mips/include/asm/pgtable.h头文件定义了设置PMD页表项的操作函数：

#define set\_pmd(pmdptr, pmdval) do { **\*(pmdptr) = (pmdval)**; } while(0)

set\_pgd(pgdptr, pgdval)函数最终的功能就是将pdgval值写入pgdptr指向的PGD页表项中。

**■其它操作函数**

内核在/include/asm-generic/pgtable-nopud.h头文件定义了PGD页表项的其它操作函数：

static inline int pgd\_none(pgd\_t pgd) { return 0; } /\*PGD页表项是否不存在，一直存在\*/

static inline int pgd\_bad(pgd\_t pgd) { return 0; }

static inline int pgd\_present(pgd\_t pgd) { return 1; } /\*PGD页表项一直存在，且有效\*/

static inline void pgd\_clear(pgd\_t \*pgd) { }

由于创建PGD页表时，没有映射的页表项都指向无效PTE页表，因此PGD页表项不会为空，始终有效（PTE页表项可以无效）。

### 4.1.3 PTE页表

PTE页表是内存页表模型中的最后一级，其页表项内容是要填入处理器TLB中的，与内存访问直接相关，因此PTE页表稍微复杂一些。注意，这里所有讨论的都是内存页表，没有涉及TLB中的页表项，两者之间的转换由TLB异常处理程序（或硬件）完成，只有将内存页表项填入到TLB之后，页表项才能真正的用于地址转换。

#### 1页表操作

内核在/arch/mips/include/asm/pgalloc.h头文件等定义了PTE页表的分配和释放函数，例如：

●static inline pte\_t \***pte\_alloc\_one\_kernel**(struct mm\_struct \*mm,unsigned long address)：创建PTE页表并清零，返回页表起始虚拟地址，mm和address参数多余。

●static inline struct page \***pte\_alloc\_one**(struct mm\_struct \*mm,unsigned long address)：创建PTE页表并清零，返回page实例指针，mm和address参数多余。

●int **\_\_pte\_alloc\_kernel**(pmd\_t \*pmd, unsigned long address)：创建PTE页表，并关联到pmd指向的中间页表项，函数定义在/mm/memory.c文件内。

●int **\_\_pte\_alloc**(struct mm\_struct \*mm, struct vm\_area\_struct \*vma,pmd\_t \*pmd, unsigned long address)：创建PTE页表，并关联到pmd指向的中间页表项，函数定义在/mm/memory.c文件内。

●**pte\_alloc\_kernel**(pmd, address)：返回address地址所在页对应的PTE页表项指针，可能需要创建PTE页表，并关联到pmd指向的中间页表项，函数定义在/include/linux/mm.h头文件：

#define **pte\_alloc\_kernel**(pmd, address) \

((unlikely(pmd\_none(\*(pmd))) && **\_\_pte\_alloc\_kernel(pmd, address)**)? \ /\*/mm/memory.c\*/

NULL: pte\_offset\_kernel(pmd, address))

释放PTE页表的函数：

●static inline void **pte\_free\_kernel**(struct mm\_struct \*mm, pte\_t \*pte)：函数内直接调用伙伴系统释放页函数。

#### 2页表项定义

内存PTE页表项数据结构定义在/arch/mips/include/asm/page.h头文件：

**typedef struct { unsigned long pte; } pte\_t;** /\*PTE页表项数据结构\*/

#define pte\_val(x) ((x).pte) /\*页表项数值\*/

#define \_\_pte(x) ((pte\_t) { (x) } ) /\*将整数x转换成pte\_t类型数据\*/

**typedef struct { unsigned long pgprot; } pgprot\_t;**  /\*访问属性标记位，页表项低12位\*/

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

#define \_\_pgprot(x) ((pgprot\_t) { (x) } )

在介绍内存PTE页表项的布局前,先通过下图了解一下内存页表项、寄存器页表项和TLB中页表项的关系，如下图所示：



上图中最上面是保存在内存中的PTE页表项，在TLB异常处理程序中，将异常地址对应的页表项填充至CPU寄存器（EntryLo0/EntryLo1），然后通过TLB指令将寄存器中页表项写入TLB，TLB页表项是最终可用于地址转换的页表项。由上图可知，各页表项的布局有所不同，其中内存PTE页表项中低6位（或4位，MIPSR3）没有写入寄存器页表项和TLB页表项，详细内容请参考MIPS32体系结构手册。

内存PTE页表项布局定义在/arch/mips/include/asm/pgtable\_bits.h头文件内，对MIPSr2（及以上）版本的32位处理器，内存PTE页表项的布局如下图所示：



各标记位语义如下：

**P**：是否有映射的页，置位表示页数据在物理内存；页数据在交换区时，标记位值为0；

**W**：页可写标记；

**A**：置位表示CPU刚访问了该页，0表示页面较老，较长时间未被访问；

**M**：置位表示页内容被修改；

**XI**：执行障碍，置位表示不可从页读取指令；

**RI/R**：可读，或读障碍标记（MIPSr3），\_PAGE\_READ或\_PAGE\_NO\_READ；

**G**：全局页标记；

**V**：页表项及映射有效性标记；

**D**：脏页标记，置位暗示页可写，清零暗示页不可写，若对清零页执行写操作将触发TLB修改异常；

**CCC**：缓存标记；

**PFN**：映射的物理页帧号。

页表项缓存域（CCC）属性定义在/arch/mips/include/asm/pgtable\_bits.h头文件：

#define \_CACHE\_CACHABLE\_NO\_WA (0<<\_CACHE\_SHIFT)

#define \_CACHE\_CACHABLE\_WA (1<<\_CACHE\_SHIFT)

#define  **\_CACHE\_UNCACHED**  (2<<\_CACHE\_SHIFT) /\*不可缓存\*/

#define **\_CACHE\_CACHABLE\_NONCOHERENT**  (3<<\_CACHE\_SHIFT) /\*非一致缓存\*/

#define \_CACHE\_CACHABLE\_CE (4<<\_CACHE\_SHIFT)

#define \_CACHE\_CACHABLE\_COW (5<<\_CACHE\_SHIFT)

#define \_CACHE\_CACHABLE\_CUW (6<<\_CACHE\_SHIFT)

#define \_CACHE\_UNCACHED\_ACCELERATED (7<<\_CACHE\_SHIFT)

内核在/arch/mips/include/asm/pgtable\_bits.h头文件内定义了各标记位的组合：

#define **\_PAGE\_SILENT\_READ \_PAGE\_VALID**

#define **\_PAGE\_SILENT\_WRITE \_PAGE\_DIRTY**

#define \_PFN\_MASK (~((1 << (\_PFN\_SHIFT)) - 1)) /\*用于屏蔽标记字段\*/

#define \_\_READABLE (\_PAGE\_SILENT\_READ | \_PAGE\_READ | \_PAGE\_ACCESSED)

#define \_\_WRITEABLE (\_PAGE\_SILENT\_WRITE | \_PAGE\_WRITE | \_PAGE\_MODIFIED)

#define \_PAGE\_CHG\_MASK (\_PAGE\_ACCESSED | \_PAGE\_MODIFIED | \_PFN\_MASK | \

\_CACHE\_MASK)

pgprot\_t结构体用于表示页表项中的页访问属性标记，内核在/arch/mips/include/asm/pgtable.h头文件定义了pgprot\_t结构体实例，用于表示页访问属性。\_\_pgprot(x)宏用于将标记组合转换成pgprot\_t实例。

#define PAGE\_NONE \_\_pgprot(\_PAGE\_PRESENT | \_CACHE\_CACHABLE\_NONCOHERENT)

#define PAGE\_SHARED \_\_pgprot(\_PAGE\_PRESENT | \_PAGE\_WRITE | \_PAGE\_READ | \

\_page\_cachable\_default)

/\*共享页属性\*/

#define PAGE\_COPY \_\_pgprot(\_PAGE\_PRESENT | \_PAGE\_READ | \_PAGE\_NO\_EXEC | \

\_page\_cachable\_default)

#define PAGE\_READONLY \_\_pgprot(\_PAGE\_PRESENT | \_PAGE\_READ | \

\_page\_cachable\_default)

/\*只读页属性\*/

#define **PAGE\_KERNEL** \_\_pgprot(\_PAGE\_PRESENT | \_\_READABLE | \_\_WRITEABLE | \

\_PAGE\_GLOBAL | \_page\_cachable\_default)

/\*内核映射页的属性，内核间接映射区页帧属性\*/

#define PAGE\_KERNEL\_NC \_\_pgprot(\_PAGE\_PRESENT | \_\_READABLE | \_\_WRITEABLE | \

\_PAGE\_GLOBAL | \_CACHE\_CACHABLE\_NONCOHERENT)

#define PAGE\_USERIO \_\_pgprot(\_PAGE\_PRESENT | \_PAGE\_READ | \_PAGE\_WRITE | \

\_page\_cachable\_default)

#define PAGE\_KERNEL\_UNCACHED \_\_pgprot(\_PAGE\_PRESENT | \_\_READABLE | \

\_\_WRITEABLE | \_PAGE\_GLOBAL | \_CACHE\_UNCACHED)

/\*内核非缓存页属性\*/

内存页表由页表项组成，可以认为是页表项数组，页表项中没有保存虚拟页编号。页表项在页表中的位置即表示对应的虚拟页帧号，内核通过虚拟页帧号在页表中索引相应的页表项。

内存PTE页表项的低6位（或4位）将不会写入TLB页表项中，详见下文。

#### 3页表项操作

下面介绍内存PTE页表项的操作函数。

**■标记位操作**

内核在/arch/mips/include/asm/pgtable.h头文件定义了各标记位的检测、设置及清除函数：

|  |  |
| --- | --- |
| 函 数 | 功 能 |
| pte\_present(pte) | 返回P标记位值，页数据是否在物理内存中。 |
| pte\_none(pte) | 检测PTE页表项值是否为0，是则返回1，非0返回0。 |
| pte\_clear(mm, addr, ptep) | 清零ptep指向的页表项。 |
| pte\_dirty(pte\_t pte) | 返回D标记位值，脏标记。 |
| pte\_write(pte\_t pte) | 返回W标记位值，是否可写。 |
| pte\_young(pte\_t pte) | 返回A标记位值，是否刚被访问过。 |
| pte\_wrprotect(pte\_t pte) | 清除写权限（写保护），清零W和D标记位。 |
| pte\_mkclean(pte\_t pte) | 清零M，D标记位。 |
| pte\_mkold(pte\_t pte) | 清零A，V标记位，。 |
| pte\_mkwrite(pte\_t pte) | 设置W标记位，如果M置位，还需设置D标记位。 |
| pte\_mkdirty(pte\_t pte) | 设置M标记位，如果页可写还需设置D标记位。 |
| pte\_mkyoung(pte\_t pte) | 设置A，V标记位，表示页刚被访问过且有效。 |
| pte\_special(pte\_t pte) | 返回0。 |
| pte\_mkspecial(pte\_t pte) | 返回pte。 |
| pte\_modify(pte\_t pte, pgprot\_t newprot) | 设置页表项访问属性部分为newprot。 |
| pgprot\_noncached(pgprot\_t \_prot) | 设置缓存属性为不可缓存，返回pgprot\_t。 |
| pgprot\_writecombine(pgprot\_t \_prot) | 缓存属性设为cpu\_data[0].writecombine。 |

PTE页表项中的标记位与page实例flags成员中的标记位有些是类似的，前者是体系结构相关的，用于CPU对物理页帧的访问，后者是通用标记，用于内核对页帧的管理。

**■生成页表项**

**mk\_pte**(page, pgprot)宏用于生成PTE页表项，page为页结构实例指针，pgport为页表项访问属性，函数定义在/arch/mips/include/asm/pgtable.h头文件：

#define mk\_pte(page, pgprot) pfn\_pte(page\_to\_pfn(page), (pgprot))

pfn\_pte()宏定义在/arch/mips/include/asm/pgtable-32.h头文件，由页帧号和访问属性生成页表项：

#define pfn\_pte(pfn, prot) \_\_pte(((unsigned long long)(pfn) << \_PFN\_SHIFT) | pgprot\_val(prot))

对于用户地址空间，页访问权限参数pgprot来自于虚拟内存域vm\_area\_struct实例**vm\_page\_prot**成员，而此成员值又由vm\_area\_struct实例**vm\_flags**成员和系统调用参数生成的，详见后面用户进程系统调用的实现函数。

**■设置页表项**

**set\_pte**(pte\_t \*ptep, pte\_t pteval)函数用于将页表项内容pteval写入内存页表中，ptep指向页表项写入位置，函数在/arch/mips/include/asm/pgtable.h头文件实现：

static inline void set\_pte(pte\_t \*ptep, pte\_t pteval)

{

**\*ptep = pteval**; /\*写入页表项内容\*/

#if !defined(CONFIG\_CPU\_R3000) && !defined(CONFIG\_CPU\_TX39XX)

if (pte\_val(pteval) & \_PAGE\_GLOBAL) { /\*如果是全局页表项，处理配对页表项\*/

pte\_t \*buddy = ptep\_buddy(ptep); /\*/arch/mips/include/asm/page.h\*/

/\*TLB中一个条目包含奇偶两个相邻页表项，两页表项都要设置全局属性\*/

#ifdef CONFIG\_SMP

... /\*多处理器需保证操作的原子性，由汇编代码实现\*/

#else /\*单核处理器\*/

if (pte\_none(\*buddy)) /\*设置配对页表项的全局属性\*/

pte\_val(\*buddy) = pte\_val(\*buddy) | \_PAGE\_GLOBAL;

#endif

}

#endif

}

#define  **set\_pte\_at**(mm, addr, ptep, pteval) set\_pte(ptep, pteval)

**pte\_clear**(struct mm\_struct \*mm, unsigned long addr, pte\_t \*ptep)函数用于清零ptep指向的页表项。

**■由地址查找页表项**

**pte\_offset**(dir, address)宏定义在/arch/mips/include/asm/pgtable-32.h头文件，返回address地址所在页的PTE页表项指针，dir表示address地址对应的PMD页表项的地址（页表项保存的是PTE页表基地址）。

### 4.1.4 TLB控制

内存页表项（PTE页表项），只有导入到处理器MMU中的TLB后，才能用于地址的转换。TLB中的页表项格式与内存页表项格式有所不同。

TLB中每条缓存表项中包含2个页表项，分别表示是偶数页和奇数页。CPU协处理器0中相关寄存器和TLB指令是操作TLB的接口。如下图所示，EntryLo0和EntryLo1寄存器与TLB中的一个页表项对应。在写入TLB表项时，将表项内容写入这两个寄存器，然后执行TLB写指令，就会将寄存器中保存的表项内容写入TLB。读TLB表项时，执行TLB读指令后，这两个寄存器就保存了读出表项的信息。

程序产生的地址（虚拟地址）将在TLB中查找匹配且有效的页表项，如果没有查找到匹配的页表项，将触发TLB重填异常。此异常处理程序中，将在内存页表中查找虚拟地址对应的PTE页表项，并将其写入处理器协处理器相应寄存器中，最后执行TLB写指令将寄存器页表项信息写入TLB。

当TLB中有匹配的表项但访问权限不对，或者重填进来的页表项也不可用时，将触发TLB缺页异常，在异常处理程序中将处理虚拟页与物理页的映射，最后生成页表项，写入内存页表和TLB，异常返回后就可以正确地进行地址转换了。



前面介绍的PTE页表项是位于内存页表中的页表项，寄存器中页表项格式如图中寄存器所示，访问权限标记只占低6位，各标记位语义与前面介绍的PTE页表项相同。由前面内存PTE页表项格式可知，PTE页表项bit[31...6]位段与寄存器页表项格式相同（不考虚RI和XI位），因此将PTE页表项右移6位即可得寄存器页表项，可写入寄存器。在TLB重填异常处理程序中，就需要将内存中的页表项右移6位后写入EntryLo0和EntryLo1寄存器。

TLB中页表项与寄存器页表项格式也不完全相同，不过它们之间的转换由硬件完成，用户只需要正确填充寄存器页表项，执行相应的TLB指令即可。

在内核空间建立映射时，会立即分配物理页帧，生成页表项填入内存页表。因此在内核空间通常只发生TLB重填异常，在异常处理程序中找到虚拟地址对应页表项写入TLB即可。

在用户空间创建映射时，通常只创建管理数据结构实例，并没有分配物理页帧，因此也不可能生成页表项。在CPU访问到未建立映射的虚拟地址时，将引发TLB缺页异常，在异常处理程序中分配物理页帧，生成页表项写入内存页表和TLB，以建立映射。

内核在/arch/mips/mm/tlb-r4k.c文件内定义了TLB的操作函数，例如：

**●\_\_update\_tlb**(vma, address, pte)：将ptep指向的页表项写入TLB中，如果TLB中已有address地址匹配的TLB表项，则覆盖它，没有则随机写入TLB中。

●**local\_flush\_tlb\_one**(unsigned long page)：如果page虚拟地址在本地CPU核TLB中有匹配的项，则对其进行清零。

●**local\_flush\_tlb\_kernel\_range**(vmaddr, end)：清零TLB中匹配[vmaddr,end)地址范围虚拟页的项。

●**local\_flush\_tlb\_all**(void)：清零本地CPU核整个TLB。

在写入TLB表项的\_\_update\_tlb()函数中（TLB重填异常不是调用这个函数），将调用**pte\_to\_entrylo**()函数将内存页表项转换成适用于EntryLo0和EntryLo1寄存器的页表项（右移6位）。

## 4.2内核地址空间管理

处理器虚拟地址空间分为内核空间和用户空间，内核空间位于上半部分，用户空间位于下半部分。内核地址空间又划分成几个更小的区域，主要有直接映射区、IO映射区（直接映射）和间接映射区。

直接映射区虚拟地址直接线性映射到底端物理内存（地址值减去一个固定的偏移量），不需要通过页表项转换。伙伴系统分配函数中，从低端内存分配的页帧映射到直接映射区，物理地址加上一个固定的偏移量即得到映射的虚拟地址。

间接映射区划分成VMALLOC区、持久映射区和固定映射区，间接映射区需要通过内核页表建立与物理内存的映射。间接映射区可映射到物理内存的任意位置。

本节主要介绍内核地址空间的布局，以及各间接映射区映射的创建和解除。

### 4.2.1内核地址空间

本小节介绍内核地址空间的布局，以及内核页表的初始化。

#### 1布局

MIPS32体系结构中，处理器在内核态和用户态下虚拟内存的映射关系如下图所示：

|  |  |  |
| --- | --- | --- |
| **虚拟地址** | **物理地址** | |
| **内核态** | **用户态** |
| 0xC000 0000-0xFFFF FFFF(1GB) | 任意位置（间接映射） | 地址错误 |
| 0xA000 0000-0xBFFF FFFF(512MB) | 0x0000 0000-0x1FFF FFFF(512MB) | 地址错误 |
| 0x8000 0000-0x9FFF FFFF(512MB) | 0x0000 0000-0x1FFF FFFF(512MB) | 地址错误 |
| 0x0000 0000-0x7FFF FFFF(2GB) | 任意位置（间接映射） | 任意位置（间接映射） |

处理器在内核态下，低2GB内存通过页表间接映射访问物理内存，2GB-3GB内存分成2部分，都直接映射到物理内存低512MB，最高的1GB内存通过页表间接映射到物理内存。

处理器在用户态下，只能访问低2GB内存，并通过页表间接映射到物理内存。

Linux内核在处理器内核态下运行，所使用的虚拟地址空间范围是0x8000 0000-0xFFFF FFFF的2GB空间，用户进程使用0x0000 0000-0x7FFF FFFF的2GB虚拟地址空间。Linux内核将2GB的内核虚拟地址空间按映射方式进行了划分，如下图所示：



内核虚拟地址空间划分为五个区域：直接映射区、外设IO映射区、VMALLOC区（映射不连续的物理内存）、持久映射区和固定映射区。各区域起止地址定义在以下两个头文件内：

（1）/arch/mips/include/asm/mach-generic/spaces.h

（2）/arch/mips/include/asm/pgtable-32.h

下面简要介绍一下各区域：

**（1）直接映射区：**内核主要的代码/数据段位于此区域，虚拟地址线性映射到物理内存低512MB空间（可缓存），访问无需经过页表，内核可执行目标文件加载到此区域。直接映射区起始地址为PAGE\_OFFSET，大小为0x20000000字节（512MB）：

#ifndef PAGE\_OFFSET /\*space.h\*/

#define PAGE\_OFFSET (CAC\_BASE + PHYS\_OFFSET)

#endif

#ifndef PHYS\_OFFSET

#define PHYS\_OFFSET \_AC(0, UL) /\*\_AC(x,y)直接返回x，include/uapi/linux/const.h\*/

#endif

#define CAC\_BASE \_AC(0x80000000, UL)

直接映射区起始地址**PAGE\_OFFSET**为0x80000000，此区域虚拟地址与物理地址的转换关系如下（/arch/mips/include/asm/page.h）：

#define **\_\_pa(x)** ((unsigned long)(x) - PAGE\_OFFSET + PHYS\_OFFSET) /\*虚拟地址转物理地址\*/

#define  **\_\_va(x)** ((void \*)((unsigned long)(x) + PAGE\_OFFSET - PHYS\_OFFSET))

/\*物理地址转虚拟地址\*/

直接映射区可直接通过调用伙伴系统分配函数获取物理内存，然后将物理地址进行线性变换即得虚拟地址。例如：\_\_get\_free\_pages(mask,order)函数，返回结果直接就是物理内存映射到此区域的虚拟地址。

物理内存中能映射到直接映射区的内存，称为低端内存，低端内存之上的物理内存称为高端内存。高端内存起始地址为：

#define HIGHMEM\_START \_AC(0x20000000, UL)

内核只有通过页表将高端内存映射到间接映射区，才能对高端内存进行访问，而低端内存可以直接映射，也可以通过页表映射到间接映射区。

**（2）IO映射区：**虚拟地址0xA000 0000至0xBFFF FFFF的512MB内存空间也是直接映射到物理内存低512MB空间，不同之处在于此段内存不能使用处理器缓存。Linux内核将此区域映射到外部IO设备控制寄存器地址空间，此区域起始地址为（spaces.h）：

#define IO\_BASE \_AC(0xa0000000, UL)

IO映射区之上的区域处理器需要通过页表才能访问物理内存，这里统称之为间接映射区，其起始地址为（spaces.h）：

#define MAP\_BASE \_AC(0xc0000000, UL)

间接映射区又被划分成VMALLOC区、持久映射区和固定映射区，间接映射区从伙伴系统分配内存后需要修改内核页表项以建立映射关系。

**（3）固定映射区：**该区域位于内核虚拟地址空间的顶端，用于将高端物理内存固定地映射到内核地址空间，或用于临时映射，此区域结束地址为（space.h）：

#define FIXADDR\_TOP ((unsigned long)(long)(int)0xfffe0000) /\*顶部保留128KB空间\*/

固定映射区起始地址及大小定义在/arch/mips/include/asm/fixmap.h头文件：

#define FIXADDR\_SIZE (\_\_end\_of\_fixed\_addresses << PAGE\_SHIFT)

#define FIXADDR\_START (FIXADDR\_TOP - FIXADDR\_SIZE)

\_\_end\_of\_fixed\_addresses表示此区域以页为单位的大小，其值后面再做介绍。

固定映射区的布局是体系结构相关的，它实际上包含两个区域，一个是固定映射区，用于将某个物理页帧映射到固定的虚拟地址（MIPS32未使用），也就是说固定映射区每个虚拟页内核指定了其用途；另一个是为每个CPU核建立的临时映射区，用于内核对高端物理内存的操作，操作前建立映射，操作结束后解除映射（临时使用）。

**（4）持久映射区：**用于将高端物理内存持久地映射到内核地址空间。只有在选择了HIGHMEM配置选项支持高端内存时才存在此区域。持久映射区起始地址为：

#define PKMAP\_BASE (0xfe000000UL) /\*pgtable-32.h（L64），距顶部32MB\*/

持久映射区结束地址为固定映射区的起始地址FIXADDR\_START。

**（5）VMALLOC区：**用于将物理上不连续的页帧映射到内核连续的虚拟内存区，该区域的典型应用就是用于向内核加载模块。此区域起止地址定义如下（pgtable-32.h）：

#define MAP\_BASE \_AC(0xc0000000, UL) /\*spaces.h\*/

#define **VMALLOC\_START** MAP\_BASE

#ifdef CONFIG\_HIGHMEM

# define **VMALLOC\_END**  (PKMAP\_BASE-2\*PAGE\_SIZE)

#else

# define VMALLOC\_END (FIXADDR\_START-2\*PAGE\_SIZE)

#endif

VMALLOC区域起始地址为0xC0000000，若内核配置了支持高端内存，则结束地址与持久映射区起始地址间隔2个页面，否则结束于与固定映射区起始地址间隔2个页面。

下表总结了内核虚拟地址空间各区域的虚拟地址段信息：

|  |  |  |
| --- | --- | --- |
| **起始地址** | **名称** | **备注** |
| FIXADDR\_START--0xFFFE 0000 | 固定映射区 | 间接映射 |
| 0xFE00 0000--(FIXADDR\_START-1) | 持久映射区 | 间接映射（需支持高端内存） |
| 0xC000 0000--VMALLOC\_END | VMALLOC区 | 间接映射，加载模块等 |
| 0xA000 0000--0xBFFF FFFF(512MB) | IO映射区 | 直接映射、不可缓存 |
| 0x8000 0000--0x9FFF FFFF (512MB) | 直接映射区 | 直接映射、可缓存 |

#### 2初始化内核页表

内核自身使用的PGD页表定义在/arch/mips/mm/init.c文件内：

pgd\_t swapper\_pg\_dir[\_PTRS\_PER\_PGD] \_\_section(.bss..swapper\_pg\_dir); /\*链接到指定段内\*/

链接文件/arch/mips/kernel/vmlinux.lds将内核PGD页表置于目标文件.bss段起始位置，并保证起始地址64KB对齐。

内核在/mm/init-mm.c文件内定义了自身地址空间数据结构实例：

struct mm\_struct init\_mm = {

.mm\_rb = RB\_ROOT,

**.pgd = swapper\_pg\_dir,**  /\*内核PGD页表基地址\*/

.mm\_users = ATOMIC\_INIT(2),

.mm\_count = ATOMIC\_INIT(1),

.mmap\_sem = \_\_RWSEM\_INITIALIZER(init\_mm.mmap\_sem),

.page\_table\_lock = \_\_SPIN\_LOCK\_UNLOCKED(init\_mm.page\_table\_lock),

.mmlist = LIST\_HEAD\_INIT(init\_mm.mmlist),

INIT\_MM\_CONTEXT(init\_mm)

};

内核PGD页表是静态创建的，在启动阶段要对内核页表进行初始化。内核在启动阶段调用paging\_init()函数完成内核PGD页表的初始化工作，函数定义如下（/arch/mips/mm/init.c）：

void \_\_init paging\_init(void)

{

unsigned long max\_zone\_pfns[MAX\_NR\_ZONES]; /\*各内存域最大物理页帧号\*/

unsigned long lastpfn \_\_maybe\_unused;

**pagetable\_init()**; /\*初始化内核页表，/arch/mips/mm/pgtable-32.c\*/

#ifdef CONFIG\_HIGHMEM

**kmap\_init()**; /\*获取临时映射区最高页PTE页表项指针，/arch/mips/mm/highmem.c\*/

#endif

...

free\_area\_init\_nodes(max\_zone\_pfns); /\*初始化结点、内存域结构实例\*/

}

pagetable\_init()函数在/arch/mips/mm/pgtable-32.c文件内实现，主要对持久映射区和固定映射区建立各级页表，最后PTE页表项内容在建立映射时填充。

pagetable\_init()函数代码如下：

void \_\_init pagetable\_init(void)

{

unsigned long vaddr;

pgd\_t \*pgd\_base;

#ifdef CONFIG\_HIGHMEM

pgd\_t \*pgd;

pud\_t \*pud;

pmd\_t \*pmd;

pte\_t \*pte;

#endif

/\*初始化内核PGD页表，先初始化内核页表中用户地址空间页表（低2GB），

\*再初始化内核地址空间页表（高2GB）\*/

**pgd\_init((unsigned long)swapper\_pg\_dir)**; /\*/arch/mips/mm/pgtable-32.c\*/

**pgd\_init((unsigned long)swapper\_pg\_dir+ sizeof(pgd\_t) \* USER\_PTRS\_PER\_PGD)**;

/\*PGD页表项全保存无效PTE页表地址\*/

pgd\_base = swapper\_pg\_dir; /\*内核页表基地址\*/

/\*创建固定映射区页表\*/

vaddr = \_\_fix\_to\_virt(\_\_end\_of\_fixed\_addresses - 1) & PMD\_MASK; /\*起始页虚拟地址\*/

**fixrange\_init(vaddr, vaddr + FIXADDR\_SIZE, pgd\_base)**; /\*创建各级页表，/arch/mips/mm/init.c\*/

#ifdef CONFIG\_HIGHMEM

vaddr = PKMAP\_BASE; /\*创建持久映射区页表\*/

**fixrange\_init(vaddr, vaddr + PAGE\_SIZE\*LAST\_PKMAP, pgd\_base)**;

/\*创建各级页表，/arch/mips/mm/init.c\*/

pgd = swapper\_pg\_dir + \_\_pgd\_offset(vaddr);

pud = pud\_offset(pgd, vaddr);

pmd = pmd\_offset(pud, vaddr);

pte = pte\_offset\_kernel(pmd, vaddr);

**pkmap\_page\_table = pte**; /\*全局变量，指向持久映射区起始PTE页表项，/mm/highmem.c\*/

#endif

}

fixrange\_init(unsigned long start, unsigned long end,pgd\_t \*pgd\_base)函数用于在pgd\_base指向的全局页表下创建虚拟地址start至end的各级页表（PTE页表项不填充），注意创建页表调用的是自举分配器提供的分配内存函数，因为此时伙伴系统尚未启用，源代码请读者自行阅读。

pagetable\_init()函数将内核PGD页表项都指向无效的PTE页表invalid\_pte\_table[]，然后为内核地址空间持久映射区和固定映射区创建页表，最后将全局变量pkmap\_page\_table指向持久映射区起始PTE页表项。

kmap\_init(void)函数定义在/arch/mips/mm/highmem.c文件内，主要是查找固定映射区中临时映射区地最高页PTE页表项，并将其指针赋予全局变量：

void \_\_init kmap\_init(void)

{

unsigned long kmap\_vstart;

kmap\_vstart = \_\_fix\_to\_virt(FIX\_KMAP\_BEGIN);

/\*临时映射区最高页基地址，/include/asm-generic/fixmap.h\*/

**kmap\_pte** = kmap\_get\_fixmap\_pte(kmap\_vstart); /\*/arch/mips/include/asm/fixmap.h\*/

}

固定映射区从高到低依次是固定映射区，各CPU核的临时映射区，各页从高到低依次编号（倒序）。kmap\_vstart即表示CPU0临时映射区起始页的基地址，FIX\_KMAP\_BEGIN表示临时映射区起始页的编号，\_\_fix\_to\_virt()函数将页编号转为虚拟基地址。全局变量kmap\_pte指向临时映射区起始页的PTE表项（倒序）。内核页表初始化完成后的结构简列如下图所示：



至此，内核页表初始化工作完成，注意这项工作是在结点、内存域数据结构实例初始化之前进行的，所以保存页表的页帧是从自举分配器中分配的。

### 4.2.2固定映射区

固定映射区在内核地址空间中始终存在，它分为固定映射区和临时映射区。固定映射区用于将某个物理页帧映射到固定的虚拟页，固定映射区中每个虚拟页都有固定的用途，这由体系结构定义。临时映射区用于将高端内存临时地映射到内核空间，用于实现内核对高端内存页的操作，操作完成后解除映射。

固定映射区位于内核地址空间的最顶端，结束地址为：

#define FIXADDR\_TOP ((unsigned long)(long)(int)0xfffe0000) /\*地址固定\*/

固定映射区大小（页面数量）由枚举类型确定（/arch/mips/include/asm/fixmap.h）：

enum fixed\_addresses {

#define FIX\_N\_COLOURS 8

FIX\_CMAP\_BEGIN, /\*0\*/

FIX\_CMAP\_END = FIX\_CMAP\_BEGIN + (FIX\_N\_COLOURS \* 2), /\*16，用于固定映射\*/

#ifdef CONFIG\_HIGHMEM /\*临时映射\*/

FIX\_KMAP\_BEGIN = FIX\_CMAP\_END + 1, /\*17，临时映射页起始编号\*/

FIX\_KMAP\_END = FIX\_KMAP\_BEGIN+(**KM\_TYPE\_NR**\***NR\_CPUS**)-1,

#endif

\_\_end\_of\_fixed\_addresses /\*FIX\_KMAP\_BEGIN+(KM\_TYPE\_NR\*NR\_CPUS)\*/

};

#define FIXADDR\_SIZE (\_\_end\_of\_fixed\_addresses << PAGE\_SHIFT)

/\*固定映射区大小（字节数）\*/

\_\_end\_of\_fixed\_addresses表示固定映射区的页数量，如果内核配置没有选择支持高端内存则映射页数量为17，表示只有固定映射区，没有临时映射区。如果内核配置支持高端内存，还将为每个CPU核划分KM\_TYPE\_NR数量的虚拟页，用于临时映射。

KM\_TYPE\_NR宏定义在/include/asm-generic/kmap\_types.h头文件内：

#ifdef \_\_WITH\_KM\_FENCE /\*需选择DEBUG\_HIGHMEM配置选项\*/

# define KM\_TYPE\_NR 41

#else

**# define KM\_TYPE\_NR 20**

#endif

如果没有选择DEBUG\_HIGHMEM配置选项，则KM\_TYPE\_NR取值为20。CPU核数量NR\_CPUS并不是处理器芯片实际具有的核数量，而是内核配置选项设置的内核最大支持的CPU核数，简单地说它是根据配置选项NR\_CPUS定义的常数。

固定映射区布局如下图所示，固定映射页和临时映射页是从高到低倒序排列的：



如图中所示，固定映射区每个虚拟页按从高到低赋予一个页索引值，用idx表示。由索引值idx确定虚拟页起始地址以及由虚拟页起始地址确定索引值的宏义如下（/include/asm-generic/fixmap.h）：

#define \_\_fix\_to\_virt(x) (FIXADDR\_TOP - ((x) << PAGE\_SHIFT)) /\*0页起始地址为FIXADDR\_TOP\*/

#define \_\_virt\_to\_fix(x) ((FIXADDR\_TOP - ((x)&PAGE\_MASK)) >> PAGE\_SHIFT)

/\*虚拟地址转索引值\*/

在固定映射区创建固定映射的函数为**set\_fixmap**(idx, phys)，idx表示虚拟页索引值，phys表示物理地址，函数声明在/include/asm-generic/fixmap.h头文件，函数由体系结构代码定义，MIPS架构没有定义此函数，这里就不做介绍了。解除固定映射的函数为clear\_fixmap(idx)，也是由体系结构代码定义。

固定映射区的每个虚拟页都是有定义的，有指定用途，在内核启动阶段建立映射，MIPS32没有使用固定映射页。

建立/解除临时映射的函数分别为**kmap\_atomic**(struct page \*page)和**kunmap\_atomic**(addr)。这两个函数的操作是原子的，不能在可能引起睡眠的地方调用，用于短时间临时地将高端内存映射到内核空间。

如果内核配置没有选择HIGHMEM选项，则这两个函数在/include/linux/highmem.h头文件内实现，建立映射时，禁止内核抢占，返回page页帧映射到直接映射区的虚拟地址。解除映射时，使能内核抢占即可。

内核配置支持高端内存时，建立/解除临时映射的函数在/arch/mips/mm/highmem.c文件内实现，下面看一下这两个函数的实现。

#### 1建立临时映射

在临时映射区，每个CPU核都有属于自己的内存空间。内核定义了percpu变量用于记录CPU核当前建立临时映射的页数量。CPU临时映射区相当于一个后进先出的队列（类似栈），后建立的映射先解除映射。

内核在/include/linux/highmem.h头文件内定义了全局percpu变量\_\_kmap\_atomic\_idx：

DECLARE\_PER\_CPU(int, \_\_kmap\_atomic\_idx);

\_\_kmap\_atomic\_idx变量用于记录各CPU核临时映射区已建立映射页的数量。建立映射时，从右至左依次建立各页的映射，解除映射时，从左至右依次解除映射，如下图所示。



\_\_kmap\_atomic\_idx-1表示最近建立映射页的索引值。建立映射\_\_kmap\_atomic\_idx表示本次建立映射虚拟页的索引值，建立映射后使其值加1，解除映射时减1。

**kmap\_atomic\_idx\_push()**函数用于获取本次建立映射页的索引值（从0开始）：

static inline int kmap\_atomic\_idx\_push(void) /\*/include/linux/highmem.h\*/

{

int idx = **\_\_this\_cpu\_inc\_return**(\_\_kmap\_atomic\_idx) - 1;

/\*\_\_kmap\_atomic\_idx值加1，/include/linux/percpu-defs.h\*/

#ifdef CONFIG\_DEBUG\_HIGHMEM

...

#endif

**return idx**; /\*返回\_\_kmap\_atomic\_idx-1值，建立映射页索引值\*/

}

**kmap\_atomic\_idx()**函数返回最近建立映射页的索引值，函数定义如下（/include/linux/highmem.h）：

static inline int kmap\_atomic\_idx(void)

{

return \_\_this\_cpu\_read(\_\_kmap\_atomic\_idx) - 1; /\*\_\_kmap\_atomic\_idx值减1\*/

}

**kmap\_atomic\_idx\_pop()**函数用于将\_\_kmap\_atomic\_idx值减1，在解除映射时调用，函数定义如下：

static inline void kmap\_atomic\_idx\_pop(void) /\*/include/linux/highmem.h\*/

{

#ifdef CONFIG\_DEBUG\_HIGHMEM

...

#else

**\_\_this\_cpu\_dec**(\_\_kmap\_atomic\_idx); /\*\_\_kmap\_atomic\_idx减1\*/

#endif

}

建立临时映射函数**kmap\_atomic()**定义在/arch/mips/mm/highmem.c文件内，函数代码如下：

void \*kmap\_atomic(struct page \*page)

{

unsigned long vaddr;

int idx, type;

preempt\_disable(); /\*禁止内核抢占\*/

pagefault\_disable(); /\*禁止缺页异常处理\*/

if (!PageHighMem(page))

return page\_address(page); /\*非高端内存页帧返回直接映射的虚拟地址\*/

type = **kmap\_atomic\_idx\_push()**; /\*获取当前CPU建立映射页索引值，/include/linux/highmem.h\*/

idx = type + KM\_TYPE\_NR\*smp\_processor\_id(); /\*映射页在整个临时映射区的索引值\*/

vaddr = **\_\_fix\_to\_virt(FIX\_KMAP\_BEGIN + idx)**; /\*临时映射区页索引值转虚拟地址\*/

#ifdef CONFIG\_DEBUG\_HIGHMEM

BUG\_ON(!pte\_none(\*(kmap\_pte - idx)));

#endif

**set\_pte**(**kmap\_pte**-idx, mk\_pte(page, PAGE\_KERNEL));

/\*设置页表项，kmap\_pte指向临时映射区最高页表项\*/

**local\_flush\_tlb\_one((unsigned long)vaddr)**; /\*刷新vaddr对应本地CPU核TLB表项\*/

return (void\*) **vaddr**; /\*返回映射虚拟地址\*/

}

建立临时映射函数首先获取当前CPU临时映射页索引值，然后将其转换成虚拟地址，并生成和写入内核页表项（刷新TLB，但没有写入），最后返回映射页起始虚拟地址。

**kmap\_atomic\_pfn**(unsigned long pfn)函数与kmap\_atomic(struct page \*page)函数功能相同，只不过参数不是映射页帧page实例指针，而是页帧起始物理地址，返回映射页的虚拟地址。另外，kmap\_atomic\_pfn()函数不管物理页帧是位于低端内存还是高端内存，都映射到临时映射区。

#### 2解除临时映射

解除临时映射函数**kunmap\_atomic(addr)**定义在/include/linux/highmem.h头文件内：

#define kunmap\_atomic(addr) \

do { \

BUILD\_BUG\_ON(\_\_same\_type((addr), struct page \*)); \

**\_\_kunmap\_atomic(addr)**; \

} while (0)

如果内核配置选择了HIGHMEM选项，\_\_kunmap\_atomic(addr)函数定义在/arch/mips/mm/highmem.c文件内，代码如下：

void \_\_kunmap\_atomic(void \*kvaddr)

{

unsigned long vaddr = (unsigned long) kvaddr & PAGE\_MASK; /\*地址页对齐\*/

int type \_\_maybe\_unused;

if (vaddr < FIXADDR\_START) { /\*虚拟地址不在固定映射区\*/

pagefault\_enable(); /\*使能缺页异常处理\*/

preempt\_enable();

return;

}

type **= kmap\_atomic\_idx()**; /\*读CPU核\_\_kmap\_atomic\_idx变量值，并减1\*/

#ifdef CONFIG\_DEBUG\_HIGHMEM /\*如果选择DEBUG\_HIGHMEM选项，则清页表项\*/

{

int idx = type + KM\_TYPE\_NR \* smp\_processor\_id(); /\*虚拟页在固定映射区索引值\*/

BUG\_ON(vaddr != \_\_fix\_to\_virt(FIX\_KMAP\_BEGIN + idx));

pte\_clear(&init\_mm, vaddr, kmap\_pte-idx); /\*清空内核PTE页表项\*/

local\_flush\_tlb\_one(vaddr); /\*清零vaddr对应本地CPU核TLB页表项\*/

}

#endif

**kmap\_atomic\_idx\_pop()**; /\*CPU核\_\_kmap\_atomic\_idx变量减1\*/

pagefault\_enable(); /\*使能缺页异常处理\*/

preempt\_enable(); /\*使能内核抢占\*/

}

解除临时映射操作比较简单，只是将CPU核对应\_\_kmap\_atomic\_idx值减1，并没有清除内核页表项和TLB中的对应表项，只有选择了DEBUG\_HIGHMEM配置选项时，解除映射操作才会清空内核页表项和TLB页表项。

临时映射的解除必须按照建立映射相反的顺序进行。内核使用临时映射时，通常是建立映射后，对页帧进行操作（读写等），操作完后立即解除映射。

使用临时映射的一个例子是在伙伴系统分配函数中，如果需要对分配的页帧清零，在prep\_new\_page()函数内将逐页调用clear\_highpage(page)函数对页帧清零。

clear\_highpage(page)函数定义如下（/include/linux/highmem.h）：

static inline void clear\_highpage(struct page \*page)

{

void \*kaddr = **kmap\_atomic(page)**; /\*建立临时映射\*/

clear\_page(kaddr); /\*页帧清零\*/

**kunmap\_atomic(kaddr)**; /\*解除临时映射\*/

}

### 4.2.3持久映射区

持久映射区用于将高端内存页（每次只能映射一页）持久地映射到内核空间。内核需选择HIGHMEM配置选项，持久映射区才存在，否则不存在持久映射区。持久映射区位于固定映射区下方。

在建立映射前，创建者需先向内核申请页帧（伙伴系统），调用kmap(page)函数建立映射，使用完后调用kunmap(page)函数解除映射，并释放页帧。建立映射时会修改映射页对应的内存页表项，并刷新TLB中表项。在解除映射时，对应内存页表项并不会马上清空，而是由内核在适当的时候清空。建立映射时，调用进程可能会进入睡眠。

#### 1映射状态管理

持久映射区起始地址为PKMAP\_BASE，其为固定值(0xfe000000UL)，结束地址为固定映射区起始地址FIXADDR\_START。

持久映射区页数量为LAST\_PKMAP，这由体系结构定义，通常为1024。由于持久映射区起始地址是固定的，因此各虚拟页的基地址也是固定的。由虚拟页在持久映射区的位置（数组项索引）及持久映射区起始地址即可计算得映射页的基地址。

映射状态管理是指对持久映射区虚拟页映射物理页帧状态的管理。持久映射区映射状态管理如下图所示：



page\_address\_map结构体用来记录虚拟页的映射状态和使用计数，结构体实例数组page\_address\_maps[]与虚拟页一一对应。数组pkmap\_count[]表示每个映射页的使用计数或状态，计数值为0表示没有建立映射（可在此页上建立映射），1表示映射已经建立（但页未被使用），已修改对应页表项，内核可解除计数值为1的页的映射状态，计数值为n（大于1），表示内核中有n-1处在使用该映射页，映射不能解除。

全局散列表page\_address\_htable[]用于管理建立了映射的page\_address\_map实例，实例通过映射页帧page实例计算散列值，确定散列链表。

page\_address\_map结构体定义在/mm/highmem.c文件内：

struct page\_address\_map {

struct page \*page; /\*指向映射页帧page实例\*/

void \*virtual; /\*虚拟页基地址\*/

struct list\_head list; /\*双链表成员\*/

};

内核在/mm/highmem.c文件内静态定义了page\_address\_map结构体实例数组和映射页使用计数的数组：

static struct page\_address\_map page\_address\_maps[LAST\_PKMAP];

static int pkmap\_count[LAST\_PKMAP]; /\*映射使用计数数组\*/

LAST\_PKMAP宏表示数组项数，定义在/arch/mips/include/asm/highmem.h头文件内：

#define LAST\_PKMAP 1024

内核在/mm/highmem.c文件内定义了散列表用于管理page\_address\_map实例，便于快速查找实例：

#define PA\_HASH\_ORDER 7

static struct page\_address\_slot {

struct list\_head lh; /\*散列表头\*/

spinlock\_t lock; /\*保护自旋锁\*/

} \_\_\_\_cacheline\_aligned\_in\_smp page\_address\_htable[1<<PA\_HASH\_ORDER]; /\*128项\*/

内核在/mm/highmem.c文件内定义了初始化散列表的page\_address\_init(void)函数，此函数由内核启动函数start\_kernel()调用。

散列函数**page\_slot(page)**通过映射页帧page实例，确定散列链表page\_address\_slot实例，映射页对应的page\_address\_map实例添加到此链表。

##### **■查找页帧映射地址**

在持久映射区建立映射前需要检查页帧是否已经映射到了持久映射区，如果是则不需要再建立映射了，只需要增加现有映射的使用计数即可，如果没有建立映射，再建立映射。

page\_address(page)函数用于判断给定page页帧是否已经映射到了持久映射区，如果是则返回映射的虚拟地址，否则返回NULL，代码如下（/mm/highmem.c）：

void \*page\_address(const struct page \*page)

{

unsigned long flags;

void \*ret;

struct page\_address\_slot \*pas;

if (!PageHighMem(page)) /\*不是高端内存页\*/

return lowmem\_page\_address(page); /\*返回直接映射地址，/include/linux/mm.h\*/

pas = **page\_slot(page)**; /\*获取散列链表，/mm/highmem.c\*/

ret = NULL;

spin\_lock\_irqsave(&pas->lock, flags);

if (!list\_empty(&pas->lh)) { /\*链表不为空，搜索链表成员，寻找匹配项\*/

struct page\_address\_map \*pam;

list\_for\_each\_entry(pam, &pas->lh, list) { /\*遍历链表中page\_address\_map实例\*/

if (**pam->page == page**) { /\*检查已映射页是否是page页\*/

**ret = pam->virtual**; /\*是则返回映射的虚拟地址\*/

goto done;

}

}

} /\*if结束\*/

done:

spin\_unlock\_irqrestore(&pas->lock, flags);

return ret;

}

page\_address(page)函数内根据page实例搜索散列链表，检查链表中page\_address\_map实例是否关联到page实例，如果是则返回映射的虚拟地址，如果page页尚未映射到虚拟页则返回NULL。

##### ■设置映射状态

在建立和解除持久映射区映射时，需要修改相应的内核页表项，另外还需要将映射关系反映到虚拟页对应的page\_address\_map实例中。

set\_page\_address()函数用于在建立/解除映射时向管理数据结构登记映射关系，函数代码如下：

void set\_page\_address(struct page \*page, void \*virtual) /\*/mm/highmem.c\*/

/\*page：page实例指针，virtual：虚拟页基地址，NULL表示解除映射，非0表示建立映射\*/

{

unsigned long flags;

struct page\_address\_slot \*pas;

struct page\_address\_map \*pam;

BUG\_ON(!PageHighMem(page));

pas = **page\_slot(page)**; /\*获取散列表链表\*/

if (virtual) { /\*建立映射操作\*/

pam = &page\_address\_maps[PKMAP\_NR((unsigned long)virtual)];

/\*查找虚拟页对应page\_address\_map实例\*/

pam->page = page; /\*page实例赋予pam->page\*/

pam->virtual = virtual; /\*虚拟页基地址\*/

spin\_lock\_irqsave(&pas->lock, flags);

**list\_add\_tail(&pam->list, &pas->lh)**; /\*将page\_address\_map实例添加到散列链表末尾\*/

spin\_unlock\_irqrestore(&pas->lock, flags);

} else { /\*解除映射\*/

spin\_lock\_irqsave(&pas->lock, flags);

list\_for\_each\_entry(pam, &pas->lh, list) { /\*查找匹配page\_address\_map实例，从链表移除\*/

if (pam->page == page) {

**list\_del(&pam->list);**  /\*将page\_address\_map实例从散列链表移除\*/

spin\_unlock\_irqrestore(&pas->lock, flags);

goto done;

}

}

spin\_unlock\_irqrestore(&pas->lock, flags);

}

done:

return;

}

set\_page\_address()函数在建立映射时找到映射虚拟页对应的page\_address\_map实例，对其进行赋值，并将其添加到全局散列表。解除映射时，在散列链表中查找映射页对应的page\_address\_map实例，将其从散列表中移除即可。

#### 2建立映射

在持久映射区建立映射的函数为kmap(page)，建立映射前调用者需获取页帧page实例。kmap(page)函数定义在/arch/mips/mm/highmem.c文件内，代码如下：

void \*kmap(struct page \*page)

{

void \*addr;

might\_sleep();

if (!PageHighMem(page)) /\*不是高端内存页帧\*/

return page\_address(page); /\*返回直接映射虚拟地址\*/

**addr = kmap\_high(page)**; /\*建立映射，/mm/highmem.c\*/

**flush\_tlb\_one((unsigned long)addr)**; /\*清零addr对应TLB页表项\*/

return addr; /\*返回映射页虚拟地址\*/

}

kmap()函数判断页帧是否位于高端内存域，若不是则返回其映射到直接映射区的虚拟地址。如果是高端内存，则调用kmap\_high(page)函数建立映射，清零映射虚拟页在TLB中的对应页表项，最后返回映射虚拟地址。

kmap\_high(page)函数是一个体系结构无关的函数，函数在/mm/highmem.c文件内实现：

void \*kmap\_high(struct page \*page)

{

unsigned long vaddr;

lock\_kmap();

vaddr = (unsigned long)page\_address(page); /\*判断页帧是否已经建立映射\*/

if (!vaddr) /\*未建立映射\*/

vaddr = **map\_new\_virtual(page)**; /\*建立映射，计数值置1，/mm/highmem.c\*/

pkmap\_count[PKMAP\_NR(vaddr)]++;

/\*不管是已建立的映射还是新建立的映射使用计数都加1，计数值最小为2\*/

BUG\_ON(pkmap\_count[PKMAP\_NR(vaddr)] < 2);

unlock\_kmap();

return (void\*) vaddr; /\*返回虚拟地址\*/

}

kmap\_high(page)函数内判断页帧是否已经映射到持久映射区，如果是则增加映射页使用计数，返回已映射的虚拟地址。如果尚未建立映射，则调用**map\_new\_virtual(page)**函数建立新映射后，再增加映射使用计数，最后返回映射页虚拟地址。

内核pkmap\_count[LAST\_PKMAP]数组用来记录持久映射区虚拟页的使用计数，正常新创建映射的页其使用计数值为2。解除映射时，解除映射函数只是对页使用计数值减1。在建立持久映射时，首先扫描使用计数值为0的页，如果存在使用计数值为0的页，则在此虚拟页上建立映射。如果没有使用计数值为0的页，则扫描使用计数值为1页，对其解除映射，然后再试图建立映射。

kmap\_high(page)函数中调用的建立新映射的函数为map\_new\_virtual(page)，定义如下（/mm/highmem.c）：

static inline unsigned long map\_new\_virtual(struct page \*page)

{

unsigned long vaddr;

int count;

unsigned int last\_pkmap\_nr; /\*扫描的使用计数数组项（索引值），初始值为0\*/

unsigned int color = get\_pkmap\_color(page); /\*返回0\*/

**start:**

count = get\_pkmap\_entries\_count(color); /\*返回LAST\_PKMAP，/mm/highmem.c\*/

for (;;) { /\*循环遍历pkmap\_count[]数组，最开始从数组项1开始\*/

last\_pkmap\_nr = get\_next\_pkmap\_nr(color); /\*下一数组项，/mm/highmem.c\*/

if **(no\_more\_pkmaps**(last\_pkmap\_nr, color)) {

/\*last\_pkmap\_nr是否为0，为0表示遍历完了pkmap\_count[]数组，/mm/highmem.c\*/

**flush\_all\_zero\_pkmaps()**;

/\*遍历完数组仍没有找到计数值为0的项，则解除计数值为1的页的映射\*/

count =get\_pkmap\_entries\_count(color);

}

if (!**pkmap\_count[last\_pkmap\_nr]**) /\*使用计数值为0\*/

break; /\*找到了可以建立映射的页，跳出循环\*/

if (--count)

continue;

/\*若解除使用计数值为1的页的映射后，仍没有可用映射页，则进程进入睡眠等待\*/

{

DECLARE\_WAITQUEUE(wait, current); /\*等待队列\*/

wait\_queue\_head\_t \*pkmap\_map\_wait =**get\_pkmap\_wait\_queue\_head(color)**;

\_\_set\_current\_state(TASK\_UNINTERRUPTIBLE); /\*进程进入不可中断睡眠状态\*/

add\_wait\_queue(**pkmap\_map\_wait**, &wait); /\*添加到等待队列\*/

unlock\_kmap();

schedule(); /\*进程调度，唤醒后从此处开始执行，在解除映射函数中唤醒睡眠进程\*/

remove\_wait\_queue(pkmap\_map\_wait, &wait); /\*唤醒后从等待队列移出\*/

lock\_kmap();

if (page\_address(page)) /\*如果其它进程已经建立了映射，返回映射虚拟地址\*/

return (unsigned long)page\_address(page);

**goto start**; /\*进程唤醒后尝试重新建立映射\*/

}

} /\*for循环结束\*/

/\*找到了可以建立映射的虚拟页，建立映射\*/

vaddr = PKMAP\_ADDR(last\_pkmap\_nr); /\*计算虚拟地址\*/

**set\_pte\_at(&init\_mm, vaddr,&(pkmap\_page\_table[last\_pkmap\_nr]), mk\_pte(page, kmap\_prot))**;

/\*写入内核PTE页表项，pkmap\_page\_table为持久映射区起始PTE页表项地址\*/

**pkmap\_count[last\_pkmap\_nr] = 1**; /\*使用计数值设为1\*/

**set\_page\_address(page, (void \*)vaddr)**; /\*登记映射状态\*/

return **vaddr**; /\*返回映射虚拟地址\*/

}

内核在get\_next\_pkmap\_nr()函数中定义了静态变量last\_pkmap\_nr（不是map\_new\_virtual()函数中的同名局部变量）用于记录最近一次建立映射虚拟页的索引值。get\_next\_pkmap\_nr()函数中将last\_pkmap\_nr值加1，如果达到最大值则返回0，如下图所示。



map\_new\_virtual(page)函数执行流程如下：调用get\_next\_pkmap\_nr()函数从上次建立映射页的下一项开始，循环遍历pkmap\_count[]数组，如果找到使用计数值为0的项，则在其对应的虚拟页上建立映射；如果扫描到pkmap\_count[0]项时，则调用**flush\_all\_zero\_pkmaps()**函数解除使用计数值为1的页的映射，然后再扫描一遍pkmap\_count[]数组，查找空闲页，以建立映射；如果仍未找到可以建立映射的页，则进程加入pkmap\_map\_wait等待队列，进入睡眠，在解除映射的函数中将唤醒进程。若找到了可以建立映射的页，建立映射操作就比较简单了，主要是生成映射页页表项，并写入内核页表，登记虚拟页映射状态，最后返回映射虚拟页基地址。

##### ■解除未使用映射

map\_new\_virtual(page)函数调用的**flush\_all\_zero\_pkmaps()**函数，用于解除使用计数值为1的虚拟页的映射，函数代码如下（/mm/highmem.c）：

static void flush\_all\_zero\_pkmaps(void)

{

int i;

int need\_flush = 0;

flush\_cache\_kmaps(); /\*flush\_cache\_all()刷新CPU缓存，/arch/mips/include/asm/highmem.h\*/

for (i = 0; i < LAST\_PKMAP; i++) { /\*扫描pkmap\_count[]数组\*/

struct page \*page;

if (pkmap\_count[i] != 1) /\*值不为1的数组项跳过\*/

continue;

**pkmap\_count[i] = 0**; /\*数组项值清0\*/

BUG\_ON(pte\_none(pkmap\_page\_table[i]));

page = **pte\_page(pkmap\_page\_table[i]);** /\*映射页page实例指针\*/

**pte\_clear**(&init\_mm, PKMAP\_ADDR(i), &pkmap\_page\_table[i]);

/\*清空虚对应内核PTE页表项\*/

**set\_page\_address(page, NULL)**; /\*将对应page\_address\_map实例从散列表中移出\*/

**need\_flush = 1**;

}

if (need\_flush)

**flush\_tlb\_kernel\_range**(PKMAP\_ADDR(0), PKMAP\_ADDR(LAST\_PKMAP));

/\*刷新持久映射区在TLB中所有匹配表项，/arch/mips/include/asm/tlbflush.h\*/

}

flush\_all\_zero\_pkmaps()函数用于解除pkmap\_count[]数组中所有使用计数值为1的虚拟页的映射关系，并刷新持久映射区在TLB中所有的匹配表项。

#### 3解除映射

解除持久映射的函数为kunmap(struct page \*page)，函数定义在/arch/mips/mm/highmem.c文件内：

void kunmap(struct page \*page)

{

BUG\_ON(in\_interrupt());

if (!PageHighMem(page)) /\*非高端内存页无需解除\*/

return;

**kunmap\_high(page)**; /\*/mm/highmem.c\*/

}

kunmap\_high(page)函数定义在/mm/highmem.c文件内，它是体系结构无关的函数：

void kunmap\_high(struct page \*page)

{

unsigned long vaddr;

unsigned long nr;

unsigned long flags;

int need\_wakeup;

unsigned int color = get\_pkmap\_color(page); /\*返回0\*/

wait\_queue\_head\_t \*pkmap\_map\_wait;

lock\_kmap\_any(flags);

**vaddr = (unsigned long)page\_address(page)**; /\*page映射的虚拟地址\*/

BUG\_ON(!vaddr);

nr = PKMAP\_NR(vaddr); /\*虚拟页在持久映射区中的索引值\*/

need\_wakeup = 0;

switch (**--pkmap\_count[nr]**) { /\*引用计数值减1\*/

case 0:

BUG();

**case 1:** /\*计数值为1的映射可以解除，唤醒等待建立映射的进程\*/

pkmap\_map\_wait = get\_pkmap\_wait\_queue\_head(color); /\*等待队列头\*/

**need\_wakeup = waitqueue\_active(pkmap\_map\_wait)**; /\*等待队列是否非空，是返回true\*/

}

unlock\_kmap\_any(flags);

if (need\_wakeup)

**wake\_up(pkmap\_map\_wait)**; /\*如果有进程在等待建立映射，唤醒它\*/

}

kunmap\_high(page)函数就比较简单了，它将映射页使用计数值减1，若值为1，则检查是否有在等待建立映射的睡眠进程，如果有则唤醒它，然后函数返回，如果没有睡眠等待进程则函数直接返回。

在前面介绍的建立映射函数中，如果没有使用计数值为0的虚拟页，也没有使用计数值为1的页，建立映射函数将进入睡眠。解除映射函数将其唤醒后，建立映射函数将重新执行，此时可以解除使用计数值为1的页（解除映射的页）的映射，以获取空闲页，从而建立映射。

### 4.2.4 VMALLOC区

VMALLOC映射区可称之为虚拟内存分配区。为什么要这个区呢？有下面两个主要的原因：

（1）使内核可以获得大块的连续的可用虚拟内存。连续的内存对内核是最有利的，内核常要使用大块的连续的内存，如加载模块。伙伴系统可以分配连续的物理内存，如果从低端内存分配，则伙伴系统分配的内存可以直接由内核使用（线性映射）。但是，随着系统的运行，物理内存将越来越碎片化，想要获得连续的大块物理内存越来越难。VMALLOC区位于内核地址空间的间接映射区，通过页表访问物理内存，因此虚拟内存可以按页映射到任意物理内存，不要求物理内存连续。即可以将连续的虚拟内存映射到离散的物理内存。

（2）使内核可以使用高端内存。内核可以通过伙伴系统从低端内存中获取物理内存，但低端内存数量有限，使用紧张，而高端内存更大，却不能直接映射到内核地址空间。使用VMALLOC区可以将高端内存（也可以是低端内存）通过页表映射到内核地址空间，使内核可以使用高端内存。

本小节介绍VMALLOC区的管理数据结构，以及创建和解除映射函数的实现。

#### 1概述

VMALLOC区位于持久映射区或固定映射区之下，与它们有2个页的间隔。VMALLOC区起止地址如下：

#define MAP\_BASE \_AC(0xc0000000, UL)

#define **VMALLOC\_START** MAP\_BASE /\*起始地址\*/

#ifdef CONFIG\_HIGHMEM

# define **VMALLOC\_END** (PKMAP\_BASE-2\*PAGE\_SIZE) /\*结束地址\*/

#else

# define VMALLOC\_END (FIXADDR\_START-2\*PAGE\_SIZE)

#endif

在VMALLOC区建立映射时，只需要指明区域大小、访问属性等参数，由内核确定映射区域的位置，以及分配物理页帧并建立映射。不需要像临时映射或持久映射一样，由调用者去获取物理页帧。

##### **■管理结构**

VMALLOC区用于动态分配连续的虚拟内存区域，并建立映射。每个映射区域由vm\_struct结构体表示，如下图所示。vm\_struct结构体中主要包含映射区域起始地址、大小、映射页帧指针等信息。建立映射的区域需要设置相应的内核页表项。所有的vm\_struct实例通过vmap\_area结构体实例添加到红黑树中。vm\_struct实例按起始地址从小到大在红黑树中从左至右排列。



所有的vmap\_area实例还将添加到全局双链表vmap\_area\_list。在VMALLOC映射初始化之前创建的vm\_struct实例将添加到全局的vmlist双链表，在VMALLOC映射初始化时将为其创建对应的vmap\_area实例，并将其添加到红黑树和双链表中。

内核将VMALLOC区划分成小块的映射区域，每一个映射区域对应有vm\_struct和vmap\_area结构体实例。vm\_struct结构体定义如下（/include/linux/vmalloc.h）：

struct vm\_struct {

struct vm\_struct \*next; /\*指向下一个vm\_struct实例，vmalloc\_init()执行前创建的实例\*/

void \*addr; /\*起始虚拟地址\*/

unsigned long size; /\*映射区域大小，字节数\*/

unsigned long flags; /\*映射区域标记\*/

struct page \*\*pages; /\*page指针数组，表示映射的页帧\*/

unsigned int nr\_pages; /\*页帧数量\*/

phys\_addr\_t phys\_addr; /\*ioremap()映射使用，表示物理内存起始地址\*/

const void \*caller; /\*创建映射函数后调用的回调函数\*/

};

vm\_struct结构体中各成员语义都比较明确，下面主要看一下flag标记成员的取值，定义如下：

#define VM\_IOREMAP 0x00000001 /\*IO映射，不是物理内存\*/

#define VM\_ALLOC 0x00000002 /\*vmalloc()函数创建的区域\*/

#define VM\_MAP 0x00000004 /\*vmap()函数显式映射区域\*/

#define VM\_USERMAP 0x00000008 /\*适用于remap\_vmalloc\_range \*/

#define VM\_VPAGES 0x00000010 /\*page指针数组位于VMALLOC区\*/

#define VM\_UNINITIALIZED 0x00000020 /\*vm\_struct实例未完全初始化\*/

#define VM\_NO\_GUARD 0x00000040 /\*映射区域最后不增加一页保护页\*/

#define VM\_KASAN 0x00000080 /\* has allocated kasan shadow memory \*/

内核在/mm/vmalloc.c文件内定义了全局双链表用于链接内核启动初期创建的vm\_struct实例：

static struct vm\_struct \*vmlist \_\_initdata; /\*vmalloc\_init()执行前，内核创建的vm\_struct实例\*/

vmap\_area结构体定义如下（/include/linux/vmalloc.h）：

struct vmap\_area {

unsigned long va\_start; /\*起始虚拟地址\*/

unsigned long va\_end; /\*结束虚拟地址\*/

unsigned long flags; /\*标记\*/

struct rb\_node **rb\_node;** /\*红黑树结点\*/

**struct list\_head list**; /\*链表元素，链接到vmap\_area\_list全局链表\*/

struct list\_head purge\_list; /\*将实例添加到延迟释放双链表，在解除映射时使用\*/

**struct vm\_struct \*vm**; /\*指向vm\_struct实例\*/

struct rcu\_head rcu\_head; /\*释放vmap\_area实例时的回调函数\*/

};

vmap\_area结构体成员语义也比较明确，主要看一下flags成员的取值，定义如下（/mm/vmalloc.c）：

#define VM\_LAZY\_FREE 0x01 /\*vmap\_area实例需要延迟释放\*/

#define VM\_LAZY\_FREEING 0x02 /\*vmap\_area实例正在延迟释放\*/

#define VM\_VM\_AREA 0x04 /\*具有对应的vm\_struct实例\*/

内核在/mm/vmalloc.c文件内定义了全局双链表及红黑树根结点用于管理vmap\_area实例：

LIST\_HEAD(vmap\_area\_list);

static struct rb\_root vmap\_area\_root = RB\_ROOT; /\*管理vmap\_area实例的红黑树根节点\*/

vmap\_area实例在内核中由双链表和红黑树共同管理，双链表中按内存区域地址从低到高排列，按内存区域地址从低到高在红黑树中从左至右排序。

\_\_insert\_vmap\_area(va)接口函数用于将vmap\_area实例添加到红黑树和双链表中。

##### **■初始化**

内核在内存管理初始化函数mm\_init()中调用vmalloc\_init()函数完成VMALLOC区映射的初始化，函数定义在/mm/vmalloc.c文件内，代码如下：

void \_\_init vmalloc\_init(void)

{

struct vmap\_area \*va;

struct vm\_struct \*tmp;

int i;

for\_each\_possible\_cpu(i) { /\*初始化静态vmap\_block\_queue和vfree\_deferred结构体实例\*/

struct vmap\_block\_queue \*vbq; /\*每CPU分配器使用的数据结构，见下文\*/

struct vfree\_deferred \*p; /\*释放映射区域时使用的数据结构，见下文\*/

vbq = &per\_cpu(vmap\_block\_queue, i);

spin\_lock\_init(&vbq->lock);

INIT\_LIST\_HEAD(&vbq->free);

p = &per\_cpu(vfree\_deferred, i);

init\_llist\_head(&p->list);

INIT\_WORK(&p->wq, **free\_work**); /\*工作成员执行函数，释放映射区域\*/

}

/\*为vmalloc\_init()函数执行前创建的vm\_struct实例创建立vmap\_area实例，并添加到管理结构\*/

for (tmp = vmlist; tmp; tmp = tmp->next) { /\*扫描vmlist双链表中的vm\_struct实例\*/

va = **kzalloc(sizeof(struct vmap\_area)**, GFP\_NOWAIT); /\*分配vmap\_area实例\*/

va->flags = VM\_VM\_AREA; /\*设置vmap\_area实例\*/

va->va\_start = (unsigned long)tmp->addr;

va->va\_end = va->va\_start + tmp->size;

**va->vm = tmp**; /\*指向vm\_struct实例\*/

**\_\_insert\_vmap\_area(va)**; /\*将vmap\_area实例添加到红黑树和双链表，/mm/vmalloc.c\*/

}

vmap\_area\_pcpu\_hole = VMALLOC\_END;

**vmap\_initialized = true;** /\*vmalloc初始化完成\*/

}

初始化函数vmalloc\_init()主要完成的工作有两项，一是初始化percpu变量vmap\_block\_queue和vfree\_deferred结构实例；二是为已经创建的vm\_struct实例创建对应的vmap\_area实例，并调用函数\_\_insert\_vmap\_area(va)将其添加到红黑树和双链表中。

##### **■接口函数**

内核地址空间VMALLOC区创建映射区分为全局分配器和每CPU处理器，全局分配器接口函数简列如下所示（/mm/vmalloc.c）：

**●**void \***vmalloc(unsigned long size)**：在VMALLOC区创建指定大小的映射区域，由内核自动分配页帧，修改内核页表项建立映射，返回起始虚拟地址。

**●**void \***vzalloc(unsigned long size)：**与vmalloc()相同，且对物理内存清零。

●void \***vmalloc\_32(unsigned long size)：**与vmalloc()相似，且保证分配页帧为32位地址可寻址。

●void \***vmalloc\_32\_user(unsigned long size)：**与vmalloc\_32()相似，且保证分配的页帧清零。

●void \***vmalloc\_exec(unsigned long size)：**与vmalloc()相似，保证映射区域具有可执行属性。

●void \***vmap(struct page \*\*pages, unsigned int count,unsigned long flags, pgprot\_t prot)**：显式映射函数，显式地将page指针数组指定的页帧映射到VMALLOC区。注意，这里由函数参数指定物理页帧，不需要内核分配。

**●void vfree(const void \*addr)**：释放vmalloc()， vmalloc\_32()或\_\_vmalloc()创建的映射区。

●**void** **vunmap(const void \*addr)：**释放vmap()创建的映射区。

每CPU分配器适用于小块、短时间的映射。每次创建映射时，分配固定大小的映射区，创建管理数据结构，建立映射时，只是在部分区域内建立映射。也就是说多次映射共用一个虚拟内存区域，以提高建立映射的速度。

每CPU分配器接口函数简列如下所示（/mm/vmalloc.c）：

●void \***vm\_map\_ram**(struct page \*\*pages, unsigned int count,int node, pgprot\_t prot)：创建映射函数，如果映射的页数小于VMAP\_MAX\_ALLOC，此函数比vmap()函数快些。映射区适用于保存短期对象。

●void **vm\_unmap\_ram**(const void \*mem, unsigned int count)：解除映射函数。

**●**void **vm\_unmap\_aliases**(void)：用于释放目前没有任何映射的虚拟映射块。

下面将介绍全局分配器创建/解除映射函数的实现，然后简要介绍一下每CPU分配器创建/解除映射函数的实现。

#### 2创建映射

在VMALLOC区创建全局映射的接口函数调用关系简列如下图所示：



创建映射函数只需指定创建映射区域的大小（以及映射页帧来自哪个内存结点），由内核确定映射区域的位置，分配物理页帧，修改内核页表等，从而建立映射，函数返回映射区起始虚拟地址。

显式映射就是将指定页帧显式地映射到VMALLOC区域，不需要内核再分配页帧了，映射区域位置仍由内核确定，并根据给定页帧修改内核页表，完成映射的建立，返回映射区起始虚拟地址。

创建映射函数最终都由**\_\_vmalloc\_node\_range()**函数执行，函数内调用\_\_get\_vm\_area\_node()函数创建映射区域管理数据结构实例（vm\_struct、vmap\_area），并确定映射区起止地址；调用\_\_vmalloc\_area\_node()函数逐页分配物理页帧，并调用map\_vm\_area()函数修改内核页表，使相应页表项指向分配的物理页帧，完成映射的建立。

显示映射更为简单一些，因为省去了分配物理页帧的过程，创建映射区域数据结构体实例后，直接调用map\_vm\_area()函数，完成映射的建立。

下面以常见的创建映射函数vmalloc(unsigned long size)为例，介绍其实现过程，函数调用关系如下图所示，所有函数都定义在/mm/vmalloc.c文件内：



vmalloc()创建映射函数主要分两个步骤，一是调用\_\_get\_vm\_area\_node()函数创建映射区管理数据结构实例，确定映射区起止地址；二是调用\_\_vmalloc\_area\_node()函数逐页分配页帧，修改页表项建立映射。

下面从\_\_vmalloc\_node\_range()函数开始，介绍创建映射函数的实现：

void \*\_\_vmalloc\_node\_range(unsigned long size, unsigned long align, unsigned long start, \

unsigned long end, gfp\_t gfp\_mask, pgprot\_t prot, unsigned long vm\_flags, int node,const void \*caller)

/\*

\*size：映射区大小，字节数；

\*align：映射区起止地址对齐要求，这里为1字节；

\*start：可分配映射区的起始虚拟地址，这里为VMAOLLOC区起始虚拟地址；

\*end：可分配映射区的结束虚拟地址，这里为VMAOLLOC区结束虚拟地址；

\*gfp\_mask：伙伴系统分配掩码，这里为GFP\_KERNEL | **\_\_GFP\_HIGHMEM**（从高端内存分配）；

\*prot：页访问属性，这里为PAGE\_KERNEL；

\*vm\_flags：vm\_struct标记，这里为0；

\*node：内存结点，这里为NUMA\_NO\_NODE（-1）；

\*caller：返回地址，这里为NULL（0）。

\*/

{

struct vm\_struct \*area;

void \*addr;

unsigned long real\_size = size;

size = PAGE\_ALIGN(size); /\*映射区大小页对齐\*/

if (!size || (size >> PAGE\_SHIFT) > totalram\_pages)

goto fail;

**area = \_\_get\_vm\_area\_node(size, align, VM\_ALLOC | VM\_UNINITIALIZED | \**

**vm\_flags, start, end, node, gfp\_mask, caller);**  /\*创建管理数据结构实例\*/

if (!area)

goto fail;

**addr = \_\_vmalloc\_area\_node(area, gfp\_mask, prot, node)**; /\*分配页帧、修改页表，建立映射\*/

if (!addr)

return NULL;

clear\_vm\_uninitialized\_flag(area); /\*清除vm\_struct->flags成员VM\_UNINITIALIZED标记位\*/

kmemleak\_alloc(addr, real\_size, 2, gfp\_mask); /\*/mm/kmemleak.c\*/

return addr; /\*返回映射区起始虚拟地址\*/

fail:

...

return NULL; /\*失败返回NULL\*/

}

\_\_vmalloc\_node\_range()函数的主要工作就是创建映射区管理数据结构实例，以及分配页帧、修改内核页表建立映射。下面分步介绍函数的实现。

##### **■创建管理数据结构**

\_\_vmalloc\_node\_range()函数中调用\_\_get\_vm\_area\_node()函数为映射区域创建vm\_struct和vmap\_area结构体实例、在VMALLOC区申请地址空间、将映射区起止地址保存到数据结构实例中，并建立vm\_struct和vmap\_area实例之间的关联。

static struct vm\_struct \*\_\_get\_vm\_area\_node(unsigned long size,unsigned long align, unsigned long flags, \

unsigned long start,unsigned long end, int node, gfp\_t gfp\_mask, const void \*caller)

/\*flags：标记，此处为VM\_ALLOC | VM\_UNINITIALIZED\*/

{

struct vmap\_area \*va;

struct vm\_struct \*area;

BUG\_ON(in\_interrupt()); /\*不能处于中断处理程序序中\*/

if (flags & VM\_IOREMAP) /\*设置IO映射区域对齐要求\*/

align = 1ul << clamp\_t(int, fls\_long(size),PAGE\_SHIFT, IOREMAP\_MAX\_ORDER);

**size = PAGE\_ALIGN(size)**; /\*映射区域大小页对齐\*/

...

**area = kzalloc\_node**(sizeof(\*area), gfp\_mask & GFP\_RECLAIM\_MASK, node);

/\*创建vm\_struct实例，并清零\*/

...

if (!(flags & VM\_NO\_GUARD)) /\*没有设置VM\_NO\_GUARD标记位\*/

size += PAGE\_SIZE; /\*映射区域增加一页作为保护页\*/

**va = alloc\_vmap\_area(size, align, start, end, node, gfp\_mask)**;

/\*创建设置vmap\_area实例（插入红黑树和全局双链表），并确定映射区起止地址\*/

....

**setup\_vmalloc\_vm(area, va, flags, caller)**; /\*设置vmap\_area和vm\_struct实例\*/

return area;

}

\_\_get\_vm\_area\_node()函数先从通用缓存中分配vm\_struct实例并清零，然后调用alloc\_vmap\_area()函数创建设置vmap\_area实例，并确定映射区域的起止地址赋予vmap\_area实例，最后调用setup\_vmalloc\_vm()函数初始化vm\_struct和vmap\_area实例。

下面介绍一下创建vmap\_area实例的alloc\_vmap\_area()函数实现，其它函数代码请读者自行阅读。

###### ●创建并设置vmap\_area实例

在介绍alloc\_vmap\_area()函数前，先看几个在搜索空洞（可用）区域过程中会用到的几个全局变量，这几个变量主要用于缓存最近一次创建操作的数据：

static struct rb\_node \*free\_vmap\_cache; /\*指向新创建的vmap\_area实例\*/

static unsigned long cached\_hole\_size; /\*最近一次创建操作中搜索到的最大空洞区大小\*/

static unsigned long cached\_vstart; /\*最近一次创建操作中指定分配区域的起始虚拟地址\*/

static unsigned long cached\_align; /\*最近一次创建映射操作指定的对齐字节数\*/

static unsigned long vmap\_area\_pcpu\_hole; /\*用于percpu内存的分配\*/

下面结合一个例子来说明以上变量的语义。alloc\_vmap\_area()函数的功能是在[vstart,vend]范围内的虚拟内存区中查找一个空洞区域，在其中创建一个大小为size（对齐）的映射区，并建立映射。如下图所示，表示虚拟内存区域的vmap\_area实例按起始地址从低到高排成一个双链表：



假设最开始时存在3个区域，分别是vmap0、vmap1和vmap2，现在要在[vstart,vend]区域内查找一个不小于size大小的空洞区域用于创建映射区。alloc\_vmap\_area()函数首先在红黑树中查找第一个结束地址不小于vstart的vmap\_area实例，局部变量first指向此实例。然后，从vstart开始查找空洞区，如上图中所示，假设第一个空洞区长度不够，而第2个空洞区长度足够，则在第2个空洞区分配区域创建vmap\_area实例vmap3。

cached\_hole\_size变量表示最近一次创建操作中搜索到的最大空洞区大小，cached\_vstart表示最近一次创建操作设置的查找地址范围的起始虚拟地址，free\_vmap\_cache指向新创建vmap\_area实例中的红黑树节点成员。cached\_align表示最近一次创建映射操作指定的创建区域对齐字节数。

下一次创建操作时，如果设置的查找地址范围起始地址不小于cached\_vstart，且分配区域大小不小于cached\_hole\_size、对齐要求不小于cached\_align，则first指向的起始搜索vmap\_area实例是free\_vmap\_cache指向的实例，而不必再到红黑树中去查找。

在释放vmap\_area实例时，如果实例结束地址小于cached\_vstart，则free\_vmap\_cache设为NULL。如果释放vmap\_area实例的结束地址大于等于cached\_vstart，并且起始地址小于等于free\_vmap\_cache指向实例的起始地址，则free\_vmap\_cache指向释放vmap\_area实例之前的实例（详见释放函数）。

下面来看一下alloc\_vmap\_area()函数的实现，代码如下：

static struct vmap\_area \*alloc\_vmap\_area(unsigned long size,unsigned long align,unsigned long vstart, \

unsigned long vend,int node, gfp\_t gfp\_mask)

{

struct vmap\_area \*va;

struct rb\_node \*n;

unsigned long addr;

int purged = 0;

struct vmap\_area \*first;

... /\*参数有效性检测\*/

va = **kmalloc\_node**(sizeof(struct vmap\_area),gfp\_mask & GFP\_RECLAIM\_MASK, node);

/\*从通用缓存中为vmap\_area实例分配空间\*/

...

kmemleak\_scan\_area(&va->rb\_node, SIZE\_MAX, gfp\_mask & GFP\_RECLAIM\_MASK);

/\*以下是在vmap\_area实例红黑树中查找起始搜索的vmap\_area实例\*/

retry:

spin\_lock(&vmap\_area\_lock); /\*获取自旋锁\*/

if (!free\_vmap\_cache ||size < cached\_hole\_size ||vstart < cached\_vstart ||align < cached\_align) {

nocache: /\*不能利用上次缓存的结果\*/

cached\_hole\_size = 0;

free\_vmap\_cache = NULL;

}

cached\_vstart = vstart; /\*搜索区域起始地址\*/

cached\_align = align; /\*对齐要求\*/

if (free\_vmap\_cache) { /\*可以利用上次搜索缓存的结果\*/

**first** = rb\_entry(free\_vmap\_cache, struct vmap\_area, rb\_node);

addr = ALIGN(first->va\_end, align); /\*free\_vmap\_cache指向实例结束地址\*/

if (addr < vstart) /\*结束地址若大于等于vstart，则free\_vmap\_cache为第一个搜索实例\*/

goto nocache;

if (addr + size < addr)

goto overflow;

} else { /\*在红黑树中查找第一个搜索的vmap\_area实例\*/

**addr** = ALIGN(vstart, align); /\*查找地址范围起始地址（对齐）\*/

if (addr + size < addr)

goto overflow;

n = vmap\_area\_root.rb\_node; /\*红黑树根节点\*/

first = NULL;

while (n) { /\*n为NULL跳出循环\*/

struct vmap\_area \*tmp;

tmp = rb\_entry(n, struct vmap\_area, rb\_node);

if (tmp->va\_end >= addr) { /\*vmap\_area实例结束地址大于等于addr\*/

**first = tmp**;

if (tmp->va\_start <= addr) /\*起始地址小于addr\*/

break;

n = n->rb\_left; /\*起始地址大于addr，查找左边实例\*/

} else /\*vmap\_area实例结束地址小于addr，查找右边实例\*/

n = n->rb\_right;

}

if (!first) /\*first为NULL表示vstart地址之后还没有vmap\_area实例，可以直接创建\*/

goto found;

}

/\*以下是从first指向实例开始，在vmap\_area实例链表中搜索空洞区域\*/

while (addr + size > first->va\_start && addr + size <= vend) { /\*创建区域与现有区域重叠\*/

if (addr + cached\_hole\_size < first->va\_start)

**cached\_hole\_size = first->va\_start - addr**; /\*最大空洞区大小\*/

**addr = ALIGN(first->va\_end, align)**; /\*vmap\_area实例后空洞区起始地址\*/

if (addr + size < addr)

goto overflow; /\*溢出了\*/

if (list\_is\_last(&first->list, &vmap\_area\_list))

goto found;

first = list\_entry(first->list.next,struct vmap\_area, list); /\*遍历下一个vmap\_area实例\*/

}

/\*查到了可用的空洞区，起始地址为addr，创建映射区\*/

found:

if (addr + size > vend)

goto overflow;

**va->va\_start = addr;** /\*设置映射区起始地址\*/

**va->va\_end = addr + size**; /\*设置映射区大小\*/

**va->flags = 0**; /\*标记\*/

**\_\_insert\_vmap\_area(va)**; /\*将vmap\_area实例添加到红黑树和双链表\*/

**free\_vmap\_cache = &va->rb\_node**; /\*指向新创建的vmap\_area实例\*/

spin\_unlock(&vmap\_area\_lock); /\*释放自旋锁\*/

...

return **va**; /\*返回vmap\_area实例指针\*/

overflow:

...

return ERR\_PTR(-EBUSY);

}

alloc\_vmap\_area()函数首先从通用缓存中为vmap\_area实例分配内存，然后根据指定的虚拟内存范围，在vmap\_area实例双链表查找第一个可以创建映射区的空洞区域，创建映射区，将起止地址赋予vmap\_area实例，最后返回vmap\_area实例指针。

\_\_get\_vm\_area\_node()函数在调用alloc\_vmap\_area()函数查找到空洞区域，创建、设置vmap\_area实例后，调用setup\_vmalloc\_vm()函数继续设置vm\_struct和vmap\_area实例。

##### **■建立映射**

\_\_vmalloc\_node\_range()函数在调用\_\_get\_vm\_area\_node()函数创建并设置了vm\_struct和vmap\_area实例后，创建映射最艰巨的任务就完成了，剩下的工作就比较简单了。那就是从伙伴系统中逐页申请页帧，修改内核页表项，建立映射，这项工作由\_\_vmalloc\_area\_node()函数完成。

\_\_vmalloc\_area\_node()函数代码如下：

static void \*\_\_vmalloc\_area\_node(struct vm\_struct \*area, gfp\_t gfp\_mask,pgprot\_t prot, int node)

/\*gfp\_mask：伙伴系统分配掩码，这里为GFP\_KERNEL | **\_\_GFP\_HIGHMEM**（从高端内存分配）\*/

{

const int order = 0; /\*分配阶为0，逐页分配\*/

struct page \*\*pages;

unsigned int nr\_pages, array\_size, i;

const gfp\_t nested\_gfp = (gfp\_mask &GFP\_RECLAIM\_MASK) | \_\_GFP\_ZERO;

const gfp\_t alloc\_mask = gfp\_mask | \_\_GFP\_NOWARN; /\*分配掩码\*/

nr\_pages = get\_vm\_area\_size(area) >> PAGE\_SHIFT; /\*映射区大小，页数量\*/

array\_size = (nr\_pages \* sizeof(struct page \*)); /\*page指针数组大小\*/

area->nr\_pages = nr\_pages;

if (array\_size > PAGE\_SIZE) { /\*如果page指针数组大小大于一页，则将数组放在VMALLOC区\*/

pages = \_\_vmalloc\_node(array\_size, 1, nested\_gfp|\_\_GFP\_HIGHMEM,

PAGE\_KERNEL, node, area->caller);

area->flags |= VM\_VPAGES;

} else { /\*page指针数组大小不到一页，则从通用缓存分配内存\*/

**pages = kmalloc\_node(array\_size, nested\_gfp, node)**;

}

**area->pages = pages**; /\*指向页指针数组\*/

...

for (i = 0; i < area->nr\_pages; i++) { /\*从伙伴系统逐页分配页帧\*/

struct page \*page;

if (node == NUMA\_NO\_NODE)

page = **alloc\_page(alloc\_mask)**; /\*分配页帧\*/

else

page = alloc\_pages\_node(node, alloc\_mask, order);

...

**area->pages[i] = page**; /\*指向page实例\*/

if (gfp\_mask & \_\_GFP\_WAIT) /\*视情重调度\*/

cond\_resched();

} /\*for循环结束，分配页帧结束\*/

if (**map\_vm\_area(area, prot, pages)**) /\*修改内核页表项，/mm/vmalloc.c\*/

goto fail;

return area->addr; /\*返回虚拟区起始虚拟地址\*/

...

}

\_\_vmalloc\_area\_node()函数首先由映射区域大小计算得所需page指针数组的大小，如果数组大小大于一页则将page数组也放在VMALLOC区（递归调用\_\_vmalloc\_node()函数），如果小于一页则直接从通用缓存中分配内存，然后逐页从伙伴系统分配页帧初始化page指针数组。

vmalloc()函数在调用\_\_vmalloc\_node\_flags()函数时分配掩码设置了\_\_GFP\_HIGHMEM标记位，因此分配函数优先从高端内存中分配页帧。\_\_vmalloc\_area\_node()函数最后调用map\_vm\_area()函数修改内核页表完成映射的创建，返回映射区起始虚拟地址。

###### ●修改内核页表

map\_vm\_area()函数用于修改内核页表，全部需修改的页表项修改成功返回0，否则返回实际修改页表项数量，函数代码如下：

int map\_vm\_area(struct vm\_struct \*area, pgprot\_t prot, struct page \*\*pages)

/\*area：vm\_struct 实例指针；prot：页表项访问属性，PAGE\_KERNEL；page：指针数组基址\*/

{

unsigned long addr = (unsigned long)area->addr; /\*映射区起始地址，页对齐\*/

unsigned long end = addr + get\_vm\_area\_size(area); /\*结束地址\*/

int err;

err = **vmap\_page\_range(addr, end, prot, pages)**; /\*修改内核页表，返回值为映射页数量\*/

return err > 0 ? 0 : err;

}

vmap\_page\_range()函数执行真正的修改内核页表的操作，介绍函数代码之前，先来简要介绍一下修改页表的机制。在两级页表模型中，真实页表中只存在PGD页表和PTE页表，PUD和PMD页表中只有一项那就是其上级页表中对应该页表项，即PGD页表项，如下图所示。PGD页表项中对应的PUD和PMD页表就是其本身（PGD页表项），因此对PUD和PMD的操作都将省略掉。



如上图中所示，假设我们要为地址从addr至end（页对齐）的虚拟内存区填充页表项，内存区跨越了两个PGD页表项。首先由addr找到起始的PGD页表项，截取地址范围[addr,next)，其中next为下一个PGD页表项代表的虚拟内存起始地址。然后判断PGD页表项是否为空，如果为空则需要创建PTE页表，不为空则不需要创建，接下来就是逐一搜索[addr,next)地址在PTE页表中的对应项，将映射物理页帧号和访问属性生成的页表项填入其中。处理完第一个PGD页表项后，接着处理第二个页表项，地址范围是[next,end)，重复上面的操作。如果虚拟内存区跨越了多个PGD页表项，则中间要处理的地址范围就是PGD页表项表示的全部内存范围。

下面我们来看一下vmap\_page\_range()函数的实现，代码如下：

static int vmap\_page\_range(unsigned long start, unsigned long end,pgprot\_t prot, struct page \*\*pages)

/\*start：起始地址，end：结束地址，prot：访问属性，pages：page指针数组\*/

{

int ret;

ret = **vmap\_page\_range\_noflush(start, end, prot, pages)**;

**flush\_cache\_vmap(start, end)**;

/\*刷新page对应缓存，清零，/arch/mips/include/asm/cacheflush.h\*/

return ret;

}

vmap\_page\_range()函数内又将工作委托给vmap\_page\_range\_noflush()函数，定义如下：

static int vmap\_page\_range\_noflush(unsigned long start, unsigned long end, \

pgprot\_t prot, struct page \*\*pages)

{

pgd\_t \*pgd;

unsigned long next;

unsigned long addr = start;

int err = 0;

int nr = 0;

BUG\_ON(addr >= end);

**pgd = pgd\_offset\_k(addr)**; /\*返回addr对应内核PGD页表项指针\*/

do {

next = **pgd\_addr\_end(addr, end)**; /\*/include/asm-generic/pgtable.h\*/

err = **vmap\_pud\_range(pgd, addr, next, prot, pages, &nr)**; /\*创建下级页表映射\*/

if (err)

return err;

} while (**pgd++**, **addr = next**, **addr != end**); /\*addr变为next，addr==end则跳出循环\*/

return **nr**; /\*返回映射页数\*/

}

pgd\_offset\_k(addr)函数计算addr地址在内核全局页表中对应的页表项。pgd\_addr\_end(addr, end)返回的是在[addr,end]地址范围内与addr映射同一PGD页表项的最大地址。假设，如上图中所示地址范围跨越了页表项边界，则返回页表项边界地址，如果在同一页表项内，则返回end。

vmap\_page\_range\_noflush()函数中每个循环处理一个PGD页表项的映射，页表项下级页表的映射由vmap\_pud\_range()函数完成。前面讲过在两级两页表模型中，PUD和PMD页表的操作都被压缩掉了（省略掉了），最后传递到了对PTE页表的操作。

PTE页表的操作函数如下，可以认为用vmap\_pte\_range()函数直接代替上面的vmap\_pud\_range()函数：

static int vmap\_pte\_range(pmd\_t \*pmd, unsigned long addr,unsigned long end, \

pgprot\_t prot, struct page \*\*pages, int \*nr)

{

pte\_t \*pte;

**pte = pte\_alloc\_kernel(pmd, addr)**; /\*PTE页表是否存在，不存在则创建\*/

if (!pte)

return -ENOMEM;

do {

**struct page \*page = pages[\*nr]**; /\*映射页帧\*/

if (WARN\_ON(!pte\_none(\*pte)))

return -EBUSY;

if (WARN\_ON(!page))

return -ENOMEM;

**set\_pte\_at(&init\_mm, addr, pte, mk\_pte(page, prot))**; /\*生成并写入内核页表项\*/

**(\*nr)++;**

} while (**pte++**, **addr += PAGE\_SIZE**, **addr != end**); /\*页表项、映射地址增加\*/

return 0;

}

pte\_alloc\_kernel(pmd, addr)函数判断PTE页表是否存在，不存在则创建，返回addr对应的PTE页表项指针。vmap\_pte\_range()函数从addr对应的PTE页表项开始生成并写入内核页表项，设置完一项后，地址加PAGE\_SIZE，表示下一页，直至到达地址边界end。nr表示映射的页帧数，映射页帧依次从page指针数组中获取。至此，创建VMALLOC区映射映射终于完成。

##### ■处理初始映射区

内核在启动阶段，执行vmalloc\_init()函数前，有时会静态定义VMALLOC区映射区vm\_struct实例，设置映射区大小、标记等（不设置起始地址），然后调用vm\_area\_register\_early()函数向内核注册vm\_struct实例。

vm\_area\_register\_early()函数内会对注册的vm\_struct实例设置起始地址，并将其添加到vmlist双链表。在vmalloc\_init()初始化函数中，将扫描vmlist双链表，对各vm\_struct实例创建对应的vmap\_area实例，并将实例添加到红黑树和双链表。

下面看一下早期注册vm\_struct实例的vm\_area\_register\_early()函数的定义：

void \_\_init vm\_area\_register\_early(struct vm\_struct \*vm, size\_t align)

/\*vm：vm\_struct实例，align：映射区起始地址对齐要求，一般为PAGE\_SIZE\*/

{

static size\_t **vm\_init\_off** \_\_initdata; /\*记录已创建映射区的总大小\*/

unsigned long addr;

addr = ALIGN(VMALLOC\_START + vm\_init\_off, align); /\*本次创建映射区起始地址\*/

**vm\_init\_off** = PFN\_ALIGN(addr + **vm->size**) - VMALLOC\_START; /\*更新已创建映射区总大小\*/

vm->addr = (void \*)addr; /\*设置映射区起始地址\*/

**vm\_area\_add\_early(vm)**; /\*将vm\_struct实例添加到vmlist双链表\*/

}

在vm\_area\_register\_early()函数中，从VMALLOC区起始地址VMALLOC\_START开始，依次向上为各注册的映射区分配内存区域。静态变量vm\_init\_off记录了分配映射区的总大小，每次分配时在其上增加映射区，（VMALLOC\_START +vm\_init\_off）（对齐）将做为本次注册映射区的起始地址。最后将vm\_struct实例添加到vmlist双链表。在vmalloc\_init()函数中将最终完成早期注册vm\_struct实例的注册工作。

#### 3显式映射

显式映射就是将一组指定页帧映射到VMALLOC区的连续虚拟内存上。理解了创建映射的操作后，显式映射就简单多了，因为显式映射与创建映射的操作基本是相同的，只不过不需要从伙伴系统中申请页帧了，操作更加简单。

显式映射实现函数为**vmap()**，代码如下：

void \*vmap(struct page \*\*pages, unsigned int count,unsigned long flags, pgprot\_t prot)

/\*pages：page实例指针数组，count：映射页数，flags：映射区标记，prot：访问属性\*/

{

struct vm\_struct \*area;

might\_sleep();

if (count > totalram\_pages)

return NULL;

**area = get\_vm\_area\_caller((count << PAGE\_SHIFT), flags,\_\_builtin\_return\_address(0))**;

/\*直接调用\_\_get\_vm\_area\_node()函数创建管理数据结构实例\*/

if (!area)

return NULL;

if (**map\_vm\_area(area, prot, pages)**) { /\*修改内核页表，建立映射\*/

vunmap(area->addr);

return NULL;

}

return area->addr;

}

get\_vm\_area\_caller()函数内直接调用\_\_get\_vm\_area\_node()函数，如下所示：

struct vm\_struct \*get\_vm\_area\_caller(unsigned long size, unsigned long flags,const void \*caller)

{

return \_\_get\_vm\_area\_node(size, 1, flags, VMALLOC\_START, VMALLOC\_END,

NUMA\_NO\_NODE, GFP\_KERNEL, caller);

}

vmap()函数调用get\_vm\_area\_caller()函数创建管理数据结构vm\_struct和vmap\_area实例，然后调用map\_vm\_area()函数修改内核页表项，建立映射，最后返回起始虚拟地址。

#### 4解除映射

前面介绍了创建映射函数的实现，下面看一下解除映射函数的实现。解除VMALLOC区映射的函数为vfree(addr)和vunmap(addr)函数，前者用于解除正常创建的映射，后者用于解除显式映射，函数调用关系如下：



vfree(addr)和vunmap(addr)函数都只有一个参数，即虚拟地址，此地址只要是映射区内的地址即可，不必是映射区起始地址。vfree()函数可以在中断处理程序中调用，而vunmap()函数不能。

如果vfree()函数在中断处理程序中被调用，则虚拟地址会添加到vfree\_deferred结构体实例管理的单链表，并调度执行工作，在工作执行函数中调用\_\_vunmap()函数解除映射。vunmap()函数直接调用\_\_vunmap()函数解除映射。

\_\_vunmap()函数中的deallocate\_pages参数表示是否释放映射页帧至伙伴系统，对于vfree()函数需要释放页帧，而vunmap()函数不释放页帧，释放页帧的操作由vunmap()函数的调用者执行。

\_\_vunmap()函数首先调用remove\_vm\_area(const void \*addr)函数，由地址addr查找vmap\_area红黑树，找到对应的vmap\_area实例，清除虚拟内存区地址对应CPU缓存，清除解除映射区域对应的内核页表项和TLB表项，设置vmap\_area实例的VM\_LAZY\_FREE标记位，随后（或延迟）将扫描设置VM\_LAZY\_FREE标记位的vmap\_area实例并释放。remove\_vm\_area()函数返回vmap\_area实例关联vm\_struct实例指针。

\_\_vunmap()函数然后调用伙伴系统释放函数逐页释放页帧，最后释放vm\_struct实例。

下面看一下另一个解除映射函数vfree()的实现，代码如下：

void vfree(const void \*addr)

{

BUG\_ON(in\_nmi());

kmemleak\_free(addr);

if (!addr)

return;

if (unlikely(in\_interrupt())) { /\*是否在中断处理程序中\*/

**struct vfree\_deferred \*p = this\_cpu\_ptr(&vfree\_deferred)**; /\*CPU关联vfree\_deferred实例\*/

if (llist\_add((struct llist\_node \*)addr, &p->list)) /\*虚拟地址添加到单链表\*/

**schedule\_work(&p->wq)**; /\*调度执行工作\*/

} else

**\_\_vunmap(addr, 1)**; /\*解除映射，源代码请读者自行阅读\*/

}

vfree\_deferred结构体定义在/mm/vmalloc.c文件内：

struct vfree\_deferred {

struct llist\_head list; /\*单链表节点\*/

struct work\_struct wq; /\*工作\*/

};

内核为每个CPU核定义了vfree\_deferred结构体实例（percpu变量）：

static DEFINE\_PER\_CPU(struct vfree\_deferred, vfree\_deferred); /\*percpu变量实例\*/

在初始化函数vmalloc\_init()中会初始化各CPU核关联vfree\_deferred结构体实例，包括初始化单链表成员，并设置工作成员的执行函数为**free\_work()**。

在vfree()函数中，如果是在中断处理程序中调用，则将地址addr添加到vfree\_deferred实例单链表中，并调度执行工作，如下图所示：



工作执行函数为free\_work()，函数内将扫描单链表，对每个虚拟地址调用\_\_vunmap(addr, 1)函数解除映射。

#### 5每CPU分配器

前面介绍的在内核地址空间VMALLOC区创建、解除映射的机制称为全局虚拟内存分配器。内核还实现了一个称为每CPU的分配器，用于在VMALLOC区创建、解除映射。

每CPU分配器框架如下图所示，分配器也是在VMALLOC区创建映射区域，但是每次创建映射区的大小是固定的，为VMAP\_BLOCK\_SIZE字节，所含页数为VMAP\_BBMAP\_BITS（在/mm/vmalloc.c文件内定义）。每个映射区域称为虚拟映射块，由vmap\_block结构体表示。vmap\_block实例关联到映射区域对应的vmap\_area实例，但是没有创建关联的vm\_struct实例。

创建虚拟映射块时，并不是立即在整个虚拟内存块上建立映射，而是只在部分区域上建立映射。下一次创建映射时就不需要重新创建数据结构实例了，可以继续在已有虚拟内存块上建立映射，以提高速度。可以理解成创建映射区域是批发，而建立映射是零售。

vmap\_block\_queue结构体包含一个双链表，管理尾部具有空闲页（未映射页）的虚拟映射块，这是一个percpu变量，每个CPU核对应一个实例。另外，vmap\_block实例还由基数树管理，实例索引值由映射块起始地址、映射块大小等参数计算得来。vmap\_block\_queue和vmap\_block结构体定义在/mm/vmalloc.c文件内，如下所示：

struct vmap\_block\_queue { /\*虚拟映射块链表\*/

spinlock\_t lock;

struct list\_head free; /\*链接具有空闲页的vmap\_block实例\*/

};

static DEFINE\_PER\_CPU(struct **vmap\_block\_queue**, vmap\_block\_queue); /\*percpu变量实例\*/

struct vmap\_block { /\*虚拟映射块\*/

spinlock\_t lock;

struct vmap\_area \*va; /\*指向vmap\_area实例\*/

unsigned long **free**, dirty; /\*free：虚拟映射块尾部空闲页数量，dirty：脏页数量\*/

unsigned long dirty\_min, dirty\_max; /\*脏区域（建立映后又解除）大小，页数\*/

struct list\_head **free\_list**; /\*将实例添加到vmap\_block\_queue实例双链表\*/

struct rcu\_head rcu\_head;

struct list\_head purge; /\*将全部映射解除的实例添加到临时链表，等待释放\*/

};



vmap\_block结构体中的free成员记录的是虚拟映射块尾部尚未建立映射页的数量。创建映射时，总是从虚拟映射块开头开始，依次建立映射，每次分配出去建立映射的页数为2order，但是实际建立映射（修改页表项）的页数可能小于2order。free记录的是从未建立映射的页的数量。如果曾建立映射的页解除了，也不会增加free的值，不视其为空闲页，也就是说free总是单调递减的。在解除映射操作中，可以将没有建立映射的虚拟内存块注销，但是只要还有映射的页就不会注销。

利用虚拟内存块创建映射具有专门的接口函数，如下所示：

●void \***vm\_map\_ram**(struct page \*\*pages, unsigned int count,int node, pgprot\_t prot)：将page指针数组关联的页通过每CPU分配器映射到VMALLOC区。

●void **vm\_unmap\_ram**(const void \*mem, unsigned int count)：解除通过每CPU分配器创建的映射。如果虚拟内存块已全部建立了映射，且在此函数中解除了最后的映射，则虚拟内存块将释放。

●void **vm\_unmap\_aliases**(void)：释放所有没有建立任何映射的虚拟内存块。

每CPU分配器主要用于创建小块、短时间的映射，创建操作会比vmap()函数快一些。如果将长时间映射和短时间映射混在一起，则会造成许多虚拟内存碎片。

##### ■创建映射

每CPU分配器创建映射的函数为**vm\_map\_ram**()，函数代码如下：

void \*vm\_map\_ram(struct page \*\*pages, unsigned int count, int node, pgprot\_t prot)

/\*pages：page实例指针数组，count：映射页数量，node：内存结点，prot：访问属性\*/

{

unsigned long size = count << PAGE\_SHIFT;

unsigned long addr;

void \*mem;

/\*若映射页数小于等于VMAP\_MAX\_ALLOC（32，32位系统），则采用每CPU分配器\*/

if (likely(count <= VMAP\_MAX\_ALLOC)) {

mem = **vb\_alloc**(size, GFP\_KERNEL); /\*查找或创建虚拟映射块，返回映射基地址\*/

...

**addr** = (unsigned long)mem; /\*映射区基地址\*/

} else { /\*映射页数大于VMAP\_MAX\_ALLOC，使用全局分配器\*/

struct vmap\_area \*va;

va = **alloc\_vmap\_area**(size, PAGE\_SIZE,

VMALLOC\_START, VMALLOC\_END, node, GFP\_KERNEL);

/\*创建vmap\_area实例（没有vm\_struct实例）\*/

...

addr = va->va\_start;

mem = (void \*)addr;

}

if (**vmap\_page\_range**(addr, addr + size, prot, pages) < 0) { /\*建立映射\*/

vm\_unmap\_ram(mem, count);

return NULL;

}

return mem; /\*返回映射基地址\*/

}

vm\_map\_ram()函数根据创建映射的页数是否大于VMAP\_MAX\_ALLOC确定是采用每CPU分配器还是全局分配器。

vb\_alloc()函数用于在CPU虚拟映射块链表中查找可以建立映射的现有虚拟块，返回映射起始地址；若没有找到可用虚拟映射块，则创建新虚拟映射块，返回虚拟块起始地址。在虚拟内存块中查找空闲页时，只查找内存块尾部的空闲区域，而不检查中间的空闲区域，并且以get\_order(count)分配阶（上对齐）为单位查找并分配空闲区（填充页表项以count数量计），函数源代码请读者自行阅读。

vmap\_page\_range()用于修改内核页表项，建立映射。建立映射后，vm\_map\_ram()函数返回映射区起始地址。

##### ■解除映射

解除每CPU分配器映射的接口函数是vm\_unmap\_ram(const void \*mem, unsigned int count)，参数mem是创建映射vm\_map\_ram()函数返回的地址，count表示解除映射页数，应与创建映射时的页数相同，不能解除部分映射。解除映射操作稍微复杂一些。

vm\_unmap\_ram()函数定义如下：

void vm\_unmap\_ram(const void \*mem, unsigned int count)

{

unsigned long size = count << PAGE\_SHIFT; /\*大小，字节数，页对齐\*/

unsigned long addr = (unsigned long)mem;

BUG\_ON(!addr); /\*地址有效性检查\*/

BUG\_ON(addr < VMALLOC\_START);

BUG\_ON(addr > VMALLOC\_END);

BUG\_ON(addr & (PAGE\_SIZE-1));

debug\_check\_no\_locks\_freed(mem, size);

vmap\_debug\_free\_range(addr, addr+size);

if (likely(count <= VMAP\_MAX\_ALLOC)) /\*页数不大于VMAP\_MAX\_ALLOC\*/

**vb\_free**(mem, size); /\*每CPU分配器解除映射\*/

else /\*页数大于VMAP\_MAX\_ALLOC\*/

free\_unmap\_vmap\_area\_addr(addr); /\*解除正常全局分配器的映射\*/

}

vm\_unmap\_ram()函数根据映射页数确定是每CPU分配器映射还是全局分配器映射。全局分配器映射的解除与vunmap(addr)函数解除映射相似，主要区别是不需要释放vm\_struct实例（因为没有创建）。

每CPU分配器映射的解除映射函数为**vb\_free**(mem, size)，函数代码如下：

static void vb\_free(const void \*addr, unsigned long size)

/\*addr：起始地址，size：大小，页对齐（字节数）\*/

{

unsigned long offset;

unsigned long vb\_idx;

unsigned int order;

struct vmap\_block \*vb;

BUG\_ON(size & ~PAGE\_MASK);

BUG\_ON(size > PAGE\_SIZE\*VMAP\_MAX\_ALLOC);

flush\_cache\_vunmap((unsigned long)addr, (unsigned long)addr + size); /\*刷新缓存\*/

order = get\_order(size); /\*映射内存区大小对应阶数，上对齐，例如：3页，order为2\*/

offset = (unsigned long)addr & (VMAP\_BLOCK\_SIZE - 1); /\*addr在虚拟映射块中偏移量\*/

offset >>= PAGE\_SHIFT; /\*页偏移量\*/

vb\_idx = addr\_to\_vb\_idx((unsigned long)addr); /\*虚拟映射块编号\*/

rcu\_read\_lock();

vb = radix\_tree\_lookup(&vmap\_block\_tree, vb\_idx); /\*在基数树中查找虚拟映射块实例\*/

rcu\_read\_unlock();

BUG\_ON(!vb);

**vunmap\_page\_range**((unsigned long)addr, (unsigned long)addr + size); /\*清除内核页表项\*/

spin\_lock(&vb->lock);

/\*扩展被释放的区域（记录脏区域），即建立映射后又解除的区域\*/

vb->dirty\_min = min(vb->dirty\_min, offset); /\*dirty页最小偏移量\*/

vb->dirty\_max = max(vb->dirty\_max, offset + (1UL << order)); /\*dirty页最大偏移量\*/

vb->dirty += 1UL << order; /\*累加dirty页数量\*/

if (vb->dirty == VMAP\_BBMAP\_BITS) { /\*虚拟映射块全部建立映射后，又全部解除\*/

BUG\_ON(vb->free);

spin\_unlock(&vb->lock);

**free\_vmap\_block(vb)**; /\*释放vmap\_block实例\*/

} else

spin\_unlock(&vb->lock);

}

vb\_free()函数先在基数树中查找虚拟映射块实例，清除解除映射区域的内核页表项，然后记录脏区域的起止页偏移量及数量，如果整个虚拟映射块都建立过映射，而且又全部解除了，则调用**free\_vmap\_block()**函数释放虚拟映射块。

free\_vmap\_block()函数定义如下：

static void free\_vmap\_block(struct vmap\_block \*vb)

{

struct vmap\_block \*tmp;

unsigned long vb\_idx;

vb\_idx = addr\_to\_vb\_idx(vb->va->va\_start);

spin\_lock(&vmap\_block\_tree\_lock);

tmp = radix\_tree\_delete(&vmap\_block\_tree, vb\_idx); /\*从基数树中移除vmap\_block实例\*/

spin\_unlock(&vmap\_block\_tree\_lock);

BUG\_ON(tmp != vb);

**free\_vmap\_area\_noflush(vb->va)**; /\*释放vmap\_area实例\*/

**kfree\_rcu**(vb, rcu\_head); /\*释放vmap\_block实例\*/

}

在解除映射的vm\_unmap\_ram()函数中只会释放全部建立映射又全部解除的虚拟内存块，如果某个虚拟映射块只部分建立了映射，且这部分映射又全部解除了，如果释放此虚拟映射块呢？

**vm\_unmap\_aliases**(void)函数用于释放目前没有任何映射的虚拟映射块，它会扫描各vmap\_block实例双链表，释放没有映射的vmap\_block实例，包括部分建立过映射，又解除映射的虚拟映射块，函数源代码请读者自行阅读。

### 4.2.5 IO映射区

内核虚拟地址空间IO映射区用于映射到外设IO控制寄存器地址空间，在MIPS32体系结构中IO映射区直接映射到物理地址低512MB的范围内。

Linux内核中，体系结构相关的代码必须提供将外设IO地址映射到内核虚拟地址的函数，以及相应的解除映射函数。

本小节介绍MIPS32体系结构提供的建立IO映射和解除映射的函数。外设IO物理地址范围如果位于低512MB的范围内，则直接线性映射到内核IO映射区，否则映射到内核VMALLOC区。

#### 1创建映射

MIPS32体系结构在/arch/mips/include/asm/io.h头文件内定义了各IO映射函数，以及解除映射函数。下面以ioremap(offset, size)函数为例，介绍创建IO映射函数的实现。

ioremap(offset, size)函数定义在/arch/mips/include/asm/io.h头文件内：

#define ioremap(offset, size) \

\_\_ioremap\_mode((offset), (size), **\_CACHE\_UNCACHED**) /\*设置不可缓存标记\*/

offset表示IO起始物理地址，size为地址空间大小，字节数。映射成功函数返回offset映射的虚拟地址，不成功返回NULL。

\_\_ioremap\_mode()函数也在/arch/mips/include/asm/io.h头文件内定义，代码如下：

static inline void \_\_iomem \* \_\_ioremap\_mode(phys\_addr\_t offset, unsigned long size,unsigned long flags)

/\*offset：起始物理地址，size：大小，字节数，flags：访问属性\*/

{

void \_\_iomem \*addr = plat\_ioremap(offset, size, flags); /\*提供平台实现映射的机会\*/

/\*返回空指针，/arch/mips/include/asm/mach-generic/ioremap.h，可由平台实现\*/

if (addr) /\*平台代码实现了映射\*/

return addr;

#define \_\_IS\_LOW512(addr) (!((phys\_addr\_t)(addr) & (phys\_addr\_t) ~0x1fffffffULL))

/\*物理地址是否为低512MB\*/

if (cpu\_has\_64bit\_addresses) {

...

} else if (\_\_builtin\_constant\_p(offset) && \

\_\_builtin\_constant\_p(size) && \_\_builtin\_constant\_p(flags)) {

phys\_addr\_t phys\_addr, last\_addr; /\*起始、结束地址\*/

phys\_addr = fixup\_bigphys\_addr(offset, size);

/\*直接返回offset，/arch/mips/include/asm/mach-generic/ioremap.h\*/

last\_addr = phys\_addr + size - 1; /\*结束地址\*/

if (!size || last\_addr < phys\_addr)

return NULL;

if (\_\_IS\_LOW512(phys\_addr) && \_\_IS\_LOW512(last\_addr) && \

**flags == \_CACHE\_UNCACHED**)

return (void \_\_iomem \*) (unsigned long)CKSEG1ADDR(phys\_addr);

/\*如果是在低512MB物理地址范围内，且不可缓存，则返回映射到kseg1的虚拟地址\*/

}

return **\_\_ioremap(offset, size, flags)**;

/\*IO物理地址空间超出低512MB范围，或未设置不可缓存标记\*/

#undef \_\_IS\_LOW512

}

\_\_ioremap\_mode()函数首先判断平台代码有没有实现plat\_ioremap()函数，如果实现了则由此函数执行映射，这提供了一个让平台实现IO映射的机会。如果平台没有实现plat\_ioremap()函数，然后判断IO地址范围是否在低512MB的范围内，并且设置了不可缓存标记，如果是则返回IO起始地址直接映射到内核IO映射区的虚拟地址（物理地址加0xA000 0000）。如果IO地址范围超过了低512MB范围或没有设置不可缓存标记，则调用\_\_ioremap(offset, size, flags)函数，将IO地址空间映射到内核VMALLOC区。

\_\_ioremap()函数在/arch/mips/mm/ioremap.c文件内实现：

void \_\_iomem \* \_\_ioremap(phys\_addr\_t phys\_addr, phys\_addr\_t size, unsigned long flags)

{

struct vm\_struct \* area;

unsigned long offset;

phys\_addr\_t last\_addr;

void \* addr;

phys\_addr = fixup\_bigphys\_addr(phys\_addr, size); /\*直接返回phys\_addr\*/

last\_addr = phys\_addr + size - 1; /\*结束地址\*/

if (!size || last\_addr < phys\_addr)

return NULL;

/\*判断IO地址范围是否在低512MB范围内，且不可缓存\*/

if (IS\_LOW512(phys\_addr) && IS\_LOW512(last\_addr) &&flags == \_CACHE\_UNCACHED)

return (void \_\_iomem \*) CKSEG1ADDR(phys\_addr);

/\*IO地址范围超过低512MB范围，或没有设置不可缓存标记\*/

if (phys\_addr < virt\_to\_phys(high\_memory)) { /\*high\_memory为实际低端内存最高地址\*/

/\*若IO地址空间位于实际低端物理内存中，除非对应page实例设置了预留标记位，

\*否则函数返回NULL，因为不允许重映射低端物理内存至IO映射区。\*/

char \*t\_addr, \*t\_end;

struct page \*page;

t\_addr = \_\_va(phys\_addr);

t\_end = t\_addr + (size - 1);

for(page = virt\_to\_page(t\_addr); page <= virt\_to\_page(t\_end); page++)

if(!PageReserved(page)) /\*对应page实例没有设置预留标记，返回NULL\*/

return NULL;

}

/\*（1）IO地址范围超过了512MB地址空间；

\*（2）IO地址范围在512MB地址空间内，且没有设置不可缓存标记。

\* 满足以上两个条件之一，映射到VMALLOC区。\*/

offset = phys\_addr & ~PAGE\_MASK;

/\*起始地址页内偏移量，PAGE\_MASK=(~(PAGE\_SIZE-1))\*/

phys\_addr &= PAGE\_MASK; /\*起始地址页对齐\*/

size = PAGE\_ALIGN(last\_addr + 1) - phys\_addr; /\*映射区大小\*/

area = **get\_vm\_area(size, VM\_IOREMAP)**; /\*创建VMALLOC区映射区管理数据结构实例\*/

if (!area)

return NULL;

**addr = area->addr**; /\*映射区起始虚拟地址\*/

if (**remap\_area\_pages((unsigned long) addr, phys\_addr, size, flags)**) {

/\*修改页表，建立映射，物理页帧起始于phys\_addr，/arch/mips/mm/ioremap.c\*/

vunmap(addr);

return NULL;

}

return (void \_\_iomem \*) (offset + (char \*)addr);

/\*返回IO起始地址映射的虚拟地址，映射区起始地址加IO起始地址页内偏移量\*/

}

\_\_ioremap()函数内再次判断IO地址范围是否位于低512MB范围内，如果是且设置了不可缓存标记，则返回IO地址空间映射到内核IO映射区的起始地址。如果IO地址空间超过了低512MB的范围或没有设置不可缓存标记，在判断了映射的有效性之后将IO地址范围映射到内核VMALLOC区，函数最后返回IO起始地址映射的虚拟地址。

get\_vm\_area()函数用于创建VMALLOC区映射的数据结构实例，remap\_area\_pages()函数由MIPS体系结构代码实现，用于设置内核页表项。

ioremap(offset, size)函数调用\_\_ioremap\_mode()函数时设置了\_CACHE\_UNCACHED（不可缓存）标记，通常如果IO地址范围全部位于低512MB范围内，则映射到IO映射区，若IO地址空间在低512MB空间之上，则映射到VMALLOC区。

另外，在/arch/mips/include/asm/io.h头文件内还定义了其它的IO映射函数，如下所示：

#define **ioremap\_nocache**(offset, size) \

\_\_ioremap\_mode((offset), (size), \_CACHE\_UNCACHED) /\*与ioremap(offset, size)相同\*/

#define  **ioremap\_cachable**(offset, size) \

\_\_ioremap\_mode((offset), (size), \_page\_cachable\_default)

#define ioremap\_cacheable\_cow(offset, size) \

\_\_ioremap\_mode((offset), (size), \_CACHE\_CACHABLE\_COW)

#define ioremap\_uncached\_accelerated(offset, size) \

\_\_ioremap\_mode((offset), (size), \_CACHE\_UNCACHED\_ACCELERATED)

#### 2解除映射

解除IO映射的iounmap()函数要简单许多，定义如下（/arch/mips/include/asm/io.h）：

static inline void iounmap(const volatile void \_\_iomem \*addr)

{

if (plat\_iounmap(addr)) /\*对应的平台解除映射函数\*/

return;

#define \_\_IS\_KSEG1(addr) (((unsigned long)(addr) & ~0x1fffffffUL) == CKSEG1)

if (cpu\_has\_64bit\_addresses || (\_\_builtin\_constant\_p(addr) && \_\_IS\_KSEG1(addr)))

return; /\*映射到内核IO映射区的映射无需解除\*/

**\_\_iounmap(addr)**; /\*解除映射到VMALLOC区的IO映射，/arch/mips/mm/ioremap.c\*/

#undef \_\_IS\_KSEG1

}

若IO地址范围映射到内核IO映射区，则不用解除映射，因为这是直接映射区，不需要清除页表项，也不用释放页帧（不是普通内存）。若映射到VMALLOC区，则调用\_\_iounmap()函数解除映射，函数代码如下：

void \_\_iounmap(const volatile void \_\_iomem \*addr)

{

struct vm\_struct \*p;

f (IS\_KSEG1(addr)) /\*直接映射无需解除\*/

return;

p = **remove\_vm\_area((void \*) (PAGE\_MASK & (unsigned long \_\_force) addr))**;

/\*释放vmap\_area实例和清除内核页表项（TLB表项），无需释放页帧\*/

if (!p)

printk(KERN\_ERR "iounmap: bad address %p\n", addr);

kfree(p); /\*释放vm\_struct实例\*/

}

如果IO地址范围是映射到内核IO映射区，则无需解除。如果是映射到内核VMALLOC区，则调用前面介绍过的remove\_vm\_area()函数释放vmap\_area结构体实例并清除内核页表项和TLB表项，但无需释放映射页帧，因为IO物理地址空间不归伙伴系统管理，最后还需要释放vm\_struct实例。

### 4.2.6小结

至此，内核地址空间管理介绍完了。内核地址空间从下至上，依次是直接映射区、IO映射区、VMALLOC区、持久映射区和固定映射区。

**直接映射区**：映射到物理内存低512MB空间，内核代码及数据主要位于此区域内。

**IO映射区**：映射到物理内存低512MB空间，映射到IO地址空间，不可缓存。

**VMALLOC区**：用于分配连续的虚拟内存，映射到离散的物理内存，优先映射到高端内存。内核模块等位于此区域。

**持久映射区：**内核需支持高端内存，持久映射区才存在。主要用于将高端内存持久地映射到内核空间。

**固定映射区**：实际上包括两个部分，一部分是固定映射区，一部分是临时映射区。固定映射区由体系结构确定，用于将物理内存映射到固定的内核虚拟地址。临时映射区对每个CPU对应一个映射区域，主要用于将高端内存页临时地映射到内核地址空间。

直接映射区和IO映射区直接映射到低512MB物理内存，不需要经过页表映射。伙伴系统从低端内存域分配的内存可以直接映射到直接映射区，只需将物理地址线性转换成虚拟地址即可。

IO地址若位于低512MB地址空间且不可缓存，则直接映射到IO映射区，否则映射到VMALLOC区。

VMALLOC区用于为内核分配连续的虚拟内存，创建映射时将逐页分配页帧（优先从高端内存分配），修改内核页表项，建立映射。

持久映射区和固定映射区都是按页建立和解除映射的，即一次只分配一个页帧，修改内核页表项，建立映射。持久映射区和固定映射区的映射一般都是临时的，通常是内核需要对某个页面进行操作时（高端内存页），就将其映射到持久映射区或固定映射区，操作结束后，立即解除映射。

从下一节开始将介绍进程地址空间的管理。进程虚拟地址总是通过页表进行映射，转换成物理地址。每个进程都有自己的地址空间和页表，进程与进程之间的地址空间是隔离的。进程地址空间按功能也划分成不同的区域。每个虚拟内存区域也由对应的数据结构实例管理。

作者认为内核地址空间与进程地址空间管理的主要区别有：

（1）内核地址空间只有一个实例，而每个进程都有自己的地址空间实例。内核地址空间对于所有进程来说是共用的。内核只需要一个页表，而每个进程都有自己的页表。内核地址空间始终存在，而进程地址空间随着进程的创建而建立，随着进程的退出而销毁。

（2）建立映射时，内核地址空间立即分配页帧，修改页表项，完成映射的建立。进程地址空间创建映射时，通常只是创建管理虚拟内存区域的数据结构实例，而没有分配页帧、修改页表项，即映射没有立即建立。在CPU访问到没有建立映射的地址时，再分配页帧建立映射，即映射是按需建立的。另外，分配给进程的页帧还有可能被回收（即使进程还在运行），而分配给内核的页帧不会被回收，除非内核主动释放。

## 4.3进程地址空间

用户进程使用处理器虚拟地址空间的低半部分（0x0000 0000至0x7FFF FFFF），进程地址空间与内核地址空间相似也按功能被划分成不同的区，进程虚拟内存始终通过页表映射到物理内存。进程虚拟地址空间划分成代码/数据区、堆、内存映射区、栈等。

进程地址空间信息由mm\_struct结构体表示，进程使用的每段虚拟内存区域由vm\_area\_struct结构体表示，内核只为进程具有的vm\_area\_struct实例包含的虚拟内存区域建立映射，进程访问vm\_area\_struct实例之外的虚拟地址将是非法的。

本节介绍进程虚拟地址空间的布局，各内存区的功能和映射关系，以及表示进程地址空间、虚拟内存区域的mm\_struct和vm\_area\_struct数据结构的定义和组织结构。

### 4.3.1地址空间描述

本小节介绍进程地址空间的布局，以及相关管理数据结构。

#### 1布局

进程地址空间布局如下图所示：



进程地址空间通常按功能进行划分，分为以下区域：

●**代码/数据区：**存储用户进程可执行目标文件代码、数据段，也称为text段，代码段并不是从0x0地址开始的，而是有一个偏移量，这由ELF标准（目标文件格式）确定。

●**堆**：存储进程运行过程中动态产生的数据，malloc()库函数申请的内存区位于此区域。

●**内存映射区：**用于将文件内容等映射到进程地址空间，从而进程对文件的操作可转换成对内存的操作，如：映射动态库文件。

●**栈：**用于保存程序运行的局部变量，实现函数调用功能，以及保存进程进入中断处理程序前的上下文信息等。

通常，在进程运行过程中，堆和内存映射区通常向上生长，栈向下生长。所有虚拟内存通过页表映射到物理内存。

内核在/arch/mips/include/asm/processor.h头文件内定义了进程地址空间各内存区的边界：

#define **TASK\_SIZE** 0x7fff8000UL /\*用户进程地址空间最高地址值\*/

#define STACK\_TOP\_MAX TASK\_SIZE

#define SPECIAL\_PAGES\_SIZE PAGE\_SIZE

#define  **STACK\_TOP**  ((TASK\_SIZE & PAGE\_MASK) - SPECIAL\_PAGES\_SIZE)

/\*栈最高地址值\*/

#define **TASK\_UNMAPPED\_BASE** PAGE\_ALIGN(TASK\_SIZE / 3)

/\*内存映射区基地址，进程地址空间1/3处\*/

#define HAVE\_ARCH\_PICK\_MMAP\_LAYOUT 1

/\*体系结构具有自身arch\_pick\_mmap\_layout()函数（创建布局）\*/

TASK\_SIZE表示进程地址空间的最大地址值，其值不等于PAGE\_OFFSET的原因是顶部空间用于存储进程运行时的环境变量和命令行参数等信息。STACK\_TOP表示栈最高地址，与TASK\_SIZE保有一页的安全区。TASK\_UNMAPPED\_BASE位于进程地址空间的1/3处，是内存映区的起始地址。堆的起始地址设置成数据段的结束地址，也就是说进程堆紧接着代码/数据区。

进程地址空间按功能进行划分后，各区域根据实际需要和使用情况，又划分成更小的区域，称为虚拟内存域，简称VMA。虚拟内存域在进程虚拟内存管理中是非常重要的概念，它是进程对地址空间操作的基本单位，例如建立映射、解除映射、设置访问权限等都是按虚拟内存域进行。

#### 2数据结构

进程地址空间由mm\_struct结构体表示，虚拟内存域由vm\_area\_struct结构体表示。下面先看一下地址空间mm\_struct结构体的定义（/include/linux/mm\_types.h），下一小节将介绍vm\_area\_struct结构体的定义：

struct mm\_struct {

struct vm\_area\_struct **\*mmap**; /\*指向vm\_area\_struct实例链表第一个对象\*/

struct rb\_root  **mm\_rb**; /\*红黑树根节点，用于管理vm\_area\_struct实例\*/

u32 vmacache\_seqnum; /\* per-thread vmacache \*/

#ifdef CONFIG\_MMU

unsigned long **(\*get\_unmapped\_area)** (struct file \*filp, unsigned long addr, unsigned long len, \

unsigned long pgoff, unsigned long flags);

/\*在地址空间中获取未映射（使用）区域函数，返回起始虚拟地址\*/

#endif

unsigned long **mmap\_base**; /\*内存映射区起始地址 \*/

unsigned long mmap\_legacy\_base; /\*内存映射区向下生长时，表示起始地址\*/

unsigned long task\_size; /\*进程地址空间大小 \*/

unsigned long highest\_vm\_end; /\*最高虚拟内存域结束地址\*/

pgd\_t \* **pgd**; /\*指向进程全局页表\*/

atomic\_t mm\_users; /\*地址空间用户数\*/

atomic\_t mm\_count; /\*地址空间使用计数 \*/

atomic\_long\_t nr\_ptes; /\* PTE页表映射的页数\*/

#if CONFIG\_PGTABLE\_LEVELS > 2

atomic\_long\_t nr\_pmds; /\* PMD页表映射页数\*/

#endif

int  **map\_count**; /\*虚拟内域数量，vm\_area\_struct实例数量\*/

spinlock\_t page\_table\_lock; /\*页表保护自旋锁\*/

struct rw\_semaphore **mmap\_sem**; /\*读写信号量\*/

struct list\_head **mmlist**; /\*双链表成员，链接可能交换出去的地址空间\*/

unsigned long hiwater\_rss; /\*未初始化数据段使用内存的高水印值\*/

unsigned long hiwater\_vm; /\*虚拟内存使用的高水印值\*/

unsigned long total\_vm; /\*映射页总数量\*/

unsigned long locked\_vm; /\*设置VM\_LOCKED（锁定）标记的页总数量\*/

unsigned long pinned\_vm; /\* Refcount permanently increased \*/

unsigned long shared\_vm; /\*共享映射页数量，文件映射(files) \*/

unsigned long exec\_vm; /\* VM\_EXEC & ~VM\_WRITE，可执行页数量\*/

unsigned long stack\_vm; /\* VM\_GROWSUP/DOWN，栈映射页数量\*/

unsigned long  **def\_flags**; /\*创建虚拟内存域时赋予的默认标记\*/

unsigned long start\_code, end\_code, start\_data, end\_data; /\*代码段，数据段起止地址\*/

unsigned long start\_brk, brk, start\_stack; /\*堆起始结束地址，栈起始地址\*/

unsigned long **arg\_start, arg\_end, env\_start, env\_end**; /\*进程命令行参数、环境变量起止地址\*/

unsigned long saved\_auxv[AT\_VECTOR\_SIZE]; /\*保存可执行ELF文件信息等\*/

struct mm\_rss\_stat **rss\_stat**; /\*统计量\*/

struct linux\_binfmt **\*binfmt**; /\*可执行目标文件二进制文件格式数据结构指针\*/

cpumask\_var\_t  **cpu\_vm\_mask\_var**; /\*地址空间由哪个CPU核使用，位图标记位\*/

mm\_context\_t **context**; /\*特定于体系结构的保存进程地址空间上下文信息的数据结构\*/

unsigned long **flags**; /\*地址空间标记，位操作必须是原子操作\*/

struct core\_state \*core\_state; /\*mm\_types.h\*/

#ifdef CONFIG\_AIO

spinlock\_t ioctx\_lock;

struct kioctx\_table \_\_rcu \*ioctx\_table;

#endif

#ifdef CONFIG\_MEMCG /\*内存控制组\*/

struct task\_struct \_\_rcu \*owner;

#endif

struct file \_\_rcu **\*exe\_file**; /\*进程可执行文件file实例指针\*/

...

struct uprobes\_state uprobes\_state; /\*需配置UPROBES，否则为空结构，/include/linux/uprobes.h\*/

...

};

mm\_struct结构体主要成员简介如下：

●**mmap**：指向vm\_area\_struct实例链表，用于链接地址空间中所有vm\_area\_struct实例。

●**mm\_rb：**红黑树根节点，用于管理vm\_area\_struct实例。

●**get\_unmapped\_area：**在进程地址空间获取一段未映射空闲（未使用）区域的函数指针。

●**pgd：**全局页表指针。

●**mmap\_sem：**读写信号量rw\_semaphore结构体实例，详见第5章。

●**rss\_stat**：mm\_rss\_stat结构体实例，结构体定义在/include/linux/mm\_types.h头文件：

struct mm\_rss\_stat {

atomic\_long\_t count[NR\_MM\_COUNTERS]; /\*统计各类型页数量\*/

};

数组项NR\_MM\_COUNTERS常数定义在枚举类型中（/include/linux/mm\_types.h）：

enum {

MM\_FILEPAGES, /\*文件缓存页\*/

MM\_ANONPAGES, /\*匿名映射页\*/

MM\_SWAPENTS, /\*交换到外部交换区的页\*/

NR\_MM\_COUNTERS /\*统计量项数\*/

};

mm\_rss\_stat结构体用于统计进程地址空间中各类型映射页的数量。

●**context：**mm\_context\_t结构体实例，mm\_context\_t为特定于体系结构的保存进程地址空间上下文信息的数据结构，结构体定义在/arch/mips/include/asm/mmu.h头文件：

typedef struct {

unsigned long **asid**[NR\_CPUS]; /\*进程地址空间ASID值，进程切换时设置，匹配TLB时使用\*/

void \*vdso;

atomic\_t fp\_mode\_switching;

} mm\_context\_t;

●**core\_state：**core\_state结构体指针，结构体定义在/include/linux/mm\_types.h头文件：

struct core\_state {

atomic\_t nr\_threads; /\*线程数量\*/

struct core\_thread dumper;

struct completion startup; /\*完成量\*/

};

struct core\_thread {

struct task\_struct \*task; /\*进程结构指针\*/

struct core\_thread \*next;

};

●**flags：**进程地址空间标记，取值定义在/include/linux/sched.h头文件：

#define MMF\_DUMPABLE\_BITS 2

#define MMF\_DUMPABLE\_MASK ((1 << MMF\_DUMPABLE\_BITS) - 1)

低2位（bit[0,1]）的取值定义如下：

#define SUID\_DUMP\_DISABLE 0 /\* No setuid dumping \*/

#define SUID\_DUMP\_USER 1 /\* Dump as user of process \*/

#define SUID\_DUMP\_ROOT 2 /\* Dump as root \*/

#define MMF\_DUMP\_ANON\_PRIVATE 2 /\*bit[2]\*/

#define MMF\_DUMP\_ANON\_SHARED 3

#define MMF\_DUMP\_MAPPED\_PRIVATE 4

#define MMF\_DUMP\_MAPPED\_SHARED 5

#define MMF\_DUMP\_ELF\_HEADERS 6

#define MMF\_DUMP\_HUGETLB\_PRIVATE 7

#define MMF\_DUMP\_HUGETLB\_SHARED 8

#define MMF\_DUMP\_FILTER\_SHIFT MMF\_DUMPABLE\_BITS

#define MMF\_DUMP\_FILTER\_BITS 7

#define MMF\_DUMP\_FILTER\_MASK \

(((1 << MMF\_DUMP\_FILTER\_BITS) - 1) << MMF\_DUMP\_FILTER\_SHIFT)

#define MMF\_DUMP\_FILTER\_DEFAULT \

((1 << MMF\_DUMP\_ANON\_PRIVATE) | (1 << MMF\_DUMP\_ANON\_SHARED) |\

(1 << MMF\_DUMP\_HUGETLB\_PRIVATE) | MMF\_DUMP\_MASK\_DEFAULT\_ELF)

#ifdef CONFIG\_CORE\_DUMP\_DEFAULT\_ELF\_HEADERS

# define MMF\_DUMP\_MASK\_DEFAULT\_ELF (1 << MMF\_DUMP\_ELF\_HEADERS)

#else

# define MMF\_DUMP\_MASK\_DEFAULT\_ELF 0

#endif

/\* leave room for more dump flags \*/

#define MMF\_VM\_MERGEABLE 16 /\* KSM may merge identical pages \*/

#define MMF\_VM\_HUGEPAGE 17 /\* set when VM\_HUGEPAGE is set on vma \*/

#define MMF\_EXE\_FILE\_CHANGED 18 /\* see prctl\_set\_mm\_exe\_file() \*/

#define MMF\_HAS\_UPROBES 19 /\* has uprobes \*/

#define MMF\_RECALC\_UPROBES 20 /\* MMF\_HAS\_UPROBES can be wrong \*/

#define MMF\_INIT\_MASK (MMF\_DUMPABLE\_MASK | MMF\_DUMP\_FILTER\_MASK)

在mm\_struct结构体中还包含地址空间各虚拟内存区的起止地址等信息。

在表示进程的task\_struct结构体中包含以下与进程地址空间相关的成员：

task\_struct{

...

**struct mm\_struct \*mm, \*active\_mm**;  /\*指向进程地址空间mm\_struct实例\*/

u32 vmacache\_seqnum;

**struct vm\_area\_struct \*vmacache[VMACACHE\_SIZE]**;

/\*虚拟内存域缓存，最近查找过的vm\_area\_struct实例，指针数组\*/

#if defined(SPLIT\_RSS\_COUNTING)

struct task\_rss\_stat rss\_stat;

#endif

...

}

task\_struct结构体中相关成员简介如下：

●**mm：**指向进程地址空间实例，如果是内核线程，则为NULL，用户进程指向其mm\_struct实例。

**●active\_mm**：指向当前系统活跃的进程地址空间实例，当前运行的内核线程，active\_mm指向最近一次运行的用户进程的mm\_struct实例（内核线程不运行时，active\_mm也设为NULL）。用户进程的active\_mm与mm相同，始终指向其mm\_struct实例。

●**vmacache[VMACACHE\_SIZE]**：虚拟内存域vm\_area\_struct结构体指针数组，缓存进程最近查找过的虚拟内存域实例。VMACACHE\_SIZE宏定义在/include/linux/sched.h头文件，目前为4。vm\_area\_struct实例通过虚拟地址计算索引值选择数组项（/include/linux/vmacache.h），关联到指针数组。

内核自身（初始）地址空间mm\_struct实例在/mm/init-mm.c文件内定义：

struct mm\_struct init\_mm = {

.mm\_rb = RB\_ROOT, /\*初始化红黑树根节点\*/

.pgd = **swapper\_pg\_dir**, /\*指向内核页表\*/

.mm\_users = ATOMIC\_INIT(2),

.mm\_count = ATOMIC\_INIT(1),

.mmap\_sem = \_\_RWSEM\_INITIALIZER(init\_mm.mmap\_sem),

.page\_table\_lock = \_\_SPIN\_LOCK\_UNLOCKED(init\_mm.page\_table\_lock),

.mmlist = LIST\_HEAD\_INIT(init\_mm.mmlist),

INIT\_MM\_CONTEXT(init\_mm)

};

内核自身进程结构实例init\_task定义在/init/init\_task.c文件内：

struct task\_struct init\_task = INIT\_TASK(init\_task);

INIT\_TASK(tsk)宏定义在/include/linux/init\_task.h头文件：

#define INIT\_TASK(tsk) \

{

...

**.mm = NULL**, \ /\*内核线程mm为空\*/

.**active\_mm** = **&init\_mm**, \ /\*当前活跃地址空间，/mm/init-mm.c\*/

...

}

#### 3映射类型

所谓映射就是虚拟内存落实到物理内存的转换关系。进程虚拟内存全部通过页表映射到物理内存，进程使用的虚拟内存区域被划分成小块，每块称为虚拟内存域，由vm\_area\_struct结构体实例表示。

虚拟内存域映射类型主要分为两类，一是匿名映射，二是文件映射，映射关系如下图所示。

匿名映射表示进程数据只存储在物理内存中，不存诸到外部块设备中（可能会存储到交换区），如堆、栈就是匿名映射区。匿名映射数据只有进程运行时有效，进程退出即丢弃。

文件映射是指将外部块设备存储的文件内容缓存至物理内存，再将物理内存映射到进程虚拟内存，从而进程可以通过对内存的操作执行对文件的操作，进程退出或映射解除时将修改过的文件内容回写至块设备。进程虚拟地址空间的内存映射区主要用于文件映射，进程代码/数据区其实也是文件映射。

进程虚拟内存通过页表映射到物理内存，文件映射中，物理内存与外部块设备之间的映射关系由文件地址空间负责实现，详见第11章。



在页面回收机制中，会对分配给进程的页帧进行回收，以释放更多的空闲页帧。内核对匿名映射页的回收方法是将其内容写出到外部块设备交换区，将交换区位置存入page实例private成员，释放页帧，修改页表项，当下次需要页数据时再从交换区中读回。对文件缓存页的回收就更简单了，如果页面没有被修改，则直接释放页帧、修改页表项即可，如果被修改过则将其回写到块设备后（写文件）再释放页帧，下次需要时再从块设备中（文件中）读取即可。

另外，虚拟内存域也可以映射到由PFN管理的物理内存（没有对应的page实例）、IO内存等非普通物理内存页。

### 4.3.2虚拟内存域

进程一般只使用虚拟地址空间中的部分内存空间，每个使用的内存区域由vm\_area\_struct结构体表示，称为虚拟内存域（VMA）。进程在使用某段虚拟内存前，必须先建立对应的vm\_area\_struct实例，若访问了没有对应vm\_area\_struct实例的内存地址，将视为无效的地址。

创建映射时一般只是申请一段虚拟内存，为其创建vm\_area\_struct实例，而并没有真正映射到物理内存。在进程访问未映射到物理内存的地址时，将触发缺页异常，在异常处理程序中分配物理页帧，修改页表项，建立映射。

本小节介绍虚拟内存域数据结构以及虚拟内存域的操作函数，后面将介绍进程地址空间映射的创建。

#### 1数据结构

虚拟内存域vm\_area\_struct实例由进程地址空间mm\_struct实例管理，mm\_struct结构体中包含管理vm\_area\_struct实例的链表和红黑树根节点成员。

vm\_area\_struct结构体定义在/include/linux/mm\_types.h头文件内，读者需要将其与内核地址空间中VMALLOC区映射使用的vm\_struct结构体区分开来，前者用于用户进程地址空间而后者用于内核地址空间。

struct vm\_area\_struct {

unsigned long vm\_start; /\*内存域起始地址\*/

unsigned long vm\_end; /\*内存域结束地址（内存域不包含此地址）\*/

struct vm\_area\_struct \*vm\_next, \*vm\_prev; /\*组成内存域链表\*/

**struct rb\_node vm\_rb**; /\*红黑树节点，添加到mm\_struct->mm\_rb红黑树\*/

unsigned long rb\_subtree\_gap; /\*以本内存域为根节点的子树当中最大空洞区长度\*/

struct mm\_struct \*vm\_mm; /\*指向mm\_struct实例\*/

pgprot\_t  **vm\_page\_prot**; /\*用于设置页表项中页访问属性，由vm\_flags成员及系统调用参数\*/

unsigned long **vm\_flags**; /\*内存域标记，/include/linux/mm.h\*/

struct {

**struct rb\_node rb;** /\*文件映射，将实例链接到地址空间address\_space->i\_mmap区间树\*/

unsigned long rb\_subtree\_last; /\*子树所有内存域中映射偏移量最大值\*/

} shared;

struct list\_head  **anon\_vma\_chain;** /\*链接anon\_vma\_chain实例，用于匿名反向映射结构\*/

struct anon\_vma  **\*anon\_vma;** /\*本内存域对应的匿名映射域实例指针\*/

/\*文件映射相关成员\*/

const struct vm\_operations\_struct  **\*vm\_ops;** /\*内存域操作结构指针，主要用于缺页异常\*/

unsigned long  **vm\_pgoff**; /\*起始页偏移量，文件映射、匿名映射语义不同\*/

struct file \* **vm\_file**; /\*指向映射文件结构实例\*/

void \* vm\_private\_data; /\*指向私有数据\*/

#ifndef CONFIG\_MMU

...

#endif

#ifdef CONFIG\_NUMA

...

#endif

};

vm\_area\_struct结构体中大部分成员的语义很明确，下面简要说明其中部分成员：

●**vm\_start：**VMA起始虚拟地址。

●**vm\_end：**VMA结束虚拟地址（实际结束地址的下一个字节）。

●**vm\_rb：**红黑树节点成员，将实例插入到mm\_struct（扩展）红黑树中。

●**rb\_subtree\_gap**：地址空间扩展红黑树中的扩展信息，表示以本节点为根的子树包含的VMA之间最大空洞区的长度（字节数）。

●**vm\_page\_prot**：表示VMA映射页的访问权限属性，用于合成PTE页表项。访问权限值来自于创建映射的系统调用参数或VMA标记成员vm\_flags，vm\_get\_page\_prot(flags)函数（后面将介绍）用于从VMA标记成员中提取访问权限值。访问权限取属性值定义在/arch/mips/include/uapi/asm/mman.h头文件：

#define PROT\_NONE 0x00 /\*不可访问页\*/

#define PROT\_READ 0x1 /\*可读页\*/

#define PROT\_WRITE 0x2 /\*可写页\*/

#define PROT\_EXEC 0x4 /\*可执行页\*/

#define PROT\_SEM 0x8 /\*页可能用于原子操作\*/

#define PROT\_GROWSDOWN 0x01000000 /\*mprotect系统调用标记，vma 向下生长\*/

#define PROT\_GROWSUP 0x02000000 /\*mprotect系统调用标记，vma向上生长 \*/

●**vm\_flags**：VMA标记成员，指示VMA的各种属性，取值定义在/include/linux/mm.h头文件：

#define VM\_NONE 0x00000000 /\*空标记\*/

#define VM\_READ 0x00000001 /\*可读\*/

#define VM\_WRITE 0x00000002 /\*可写\*/

#define VM\_EXEC 0x00000004 /\*可执行\*/

#define VM\_SHARED 0x00000008 /\*共享VMA\*/

#define VM\_MAYREAD 0x00000010 /\*仅供mprotect()系统调用使用\*/

#define VM\_MAYWRITE 0x00000020 /\*写时复制VMA\*/

#define VM\_MAYEXEC 0x00000040

#define VM\_MAYSHARE 0x00000080

#define VM\_GROWSDOWN 0x00000100 /\*VMA向下生长\*/

#define **VM\_PFNMAP** 0x00000400 /\*映射页没有page实例，如PFN管理的页\*/

#define VM\_DENYWRITE 0x00000800 /\*不可写\*/

#define **VM\_LOCKED** 0x00002000 /\*锁定内存域，立即创建，映射页不可回收\*/

#define **VM\_IO**  0x00004000 /\*映射到IO内存\*/

#define VM\_SEQ\_READ 0x00008000 /\*用户程序将顺序访问内存\*/

#define VM\_RAND\_READ 0x00010000 /\*用户程序将随机访问内存\*/

#define VM\_DONTCOPY 0x00020000 /\*fork()系统调用不复制本VMA\*/

#define VM\_DONTEXPAND 0x00040000 /\*禁止用mremap()扩展内存域\*/

#define VM\_ACCOUNT 0x00100000 /\*指定区域是否归入overcommit特性计算中\*/

#define VM\_NORESERVE 0x00200000 /\* should the VM suppress accounting \*/

#define VM\_HUGETLB 0x00400000 /\*TLB中巨型页\*/

#define VM\_ARCH\_1 0x01000000 /\*特定体系结构标记\*/

#define VM\_ARCH\_2 0x02000000

#define VM\_DONTDUMP 0x04000000 /\* Do not include in the core dump（核心转储）\*/

#ifdef CONFIG\_MEM\_SOFT\_DIRTY

# define VM\_SOFTDIRTY 0x08000000 /\* Not soft dirty clean area \*/

#else

# define VM\_SOFTDIRTY 0

#endif

#define **VM\_MIXEDMAP** 0x10000000 /\*可映射page管理页帧和PFN管理页\*/

#define VM\_HUGEPAGE 0x20000000 /\* MADV\_HUGEPAGE marked this vma \*/

#define VM\_NOHUGEPAGE 0x40000000 /\* MADV\_NOHUGEPAGE marked this vma \*/

#define VM\_MERGEABLE 0x80000000 /\* KSM may merge identical pages \*/

...

#ifndef VM\_GROWSUP

#define VM\_GROWSUP VM\_NONE

#endif

#define VM\_STACK\_INCOMPLETE\_SETUP (VM\_RAND\_READ | VM\_SEQ\_READ)

#ifndef VM\_STACK\_DEFAULT\_FLAGS

#define VM\_STACK\_DEFAULT\_FLAGS **VM\_DATA\_DEFAULT\_FLAGS**

/\*栈区域默认标记，/arch/mips/include/asm/page.h \*/

#endif

#ifdef CONFIG\_STACK\_GROWSUP /\*栈区域标记\*/

#define VM\_STACK\_FLAGS (VM\_GROWSUP | VM\_STACK\_DEFAULT\_FLAGS | VM\_ACCOUNT)

#else

#define  **VM\_STACK\_FLAGS** (VM\_GROWSDOWN |VM\_STACK\_DEFAULT\_FLAGS| \

VM\_ACCOUNT)

#endif

#define **VM\_SPECIAL** (VM\_IO | VM\_DONTEXPAND | VM\_PFNMAP | VM\_MIXEDMAP)

/\*特殊VMA标记，如映射IO内存的VMA\*/

#define VM\_INIT\_DEF\_MASK VM\_NOHUGEPAGE

其中VM\_DATA\_DEFAULT\_FLAGS标记定义在/arch/mips/include/asm/page.h头文件内，这是一个体系结构相关的标记：

#define VM\_DATA\_DEFAULT\_FLAGS (VM\_READ | VM\_WRITE | VM\_EXEC | \

VM\_MAYREAD | VM\_MAYWRITE | VM\_MAYEXEC)

vm\_flags成员是vm\_area\_struct结构体中非常重要的成员，指示了VMA的各种属性，在VMA操作、创建映射、缺页异常处理等时机将被广泛使用。vm\_flags与vm\_page\_prot成员具有对应的转换关系，可以认为vm\_page\_prot是vm\_flags标记属性在PTE页表项中的体现。

在/include/linux/mman.h头文件中定义的calc\_vm\_prot\_bits(prot)函数用于将通用访问权限属性转换成VMA标记值。

在没有访问权限参数的系统调用中创建内存域时，vm\_get\_page\_prot(flags)函数用于由vm\_flags成员值生成访问权限属性值（/mm/mmap.c）。

●**shared：**结构体成员，内部包含两个成员，用于文件映射VMA的反向映射结构。rb为红黑树节点成员，将VMA插入到文件地址空间address\_space实例i\_mmap成员表示的红黑树（区间树）中。

rb\_subtree\_last成员为区间树的扩展信息，表示在区间树中以本VMA为根的子树中VMA结束偏移量的最大值（区间结束最大值）。

●**anon\_vma\_chain：**双链表头，链接anon\_vma\_chain实例，用于构成匿名映射VMA的反向映射结构，文件映射此成员为空。

●**anon\_vma**：指向本VMA对应的匿名映射域anon\_vma结构体实例，用于反向映射结构，文件映射此成员为空。

●**vm\_ops**：VMA操作结构vm\_operations\_struct结构体指针，表示文件映射VMA的操作函数，结构体定义在/include/linux/mm.h头文件内：

struct vm\_operations\_struct {

void (\*open) (struct vm\_area\_struct \* area); /\*创建VMA时调用，通常为NULL\*/

void (\*close) (struct vm\_area\_struct \* area); /\*删除VMA时调用，通常为NULL\*/

int (**\*fault**) (struct vm\_area\_struct \*vma, struct vm\_fault \*vmf); /\*文件映射缺页异常时调用的函数\*/

void (\***map\_pages**) (struct vm\_area\_struct \*vma, struct vm\_fault \*vmf); /\*映射缓存页至VMA\*/

int (\***page\_mkwrite**) (struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

int (\***pfn\_mkwrite**) (struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

int (\*access) (struct vm\_area\_struct \*vma, unsigned long addr,void \*buf, int len, int write);

const char \*(\*name) (struct vm\_area\_struct \*vma);

#ifdef CONFIG\_NUMA

...

#endif

struct page \*(\*find\_special\_page)(struct vm\_area\_struct \*vma, unsigned long addr);

};

内核在为地址空间创建文件映射时对vm\_ops成员进行赋值，文件映射VMA此成员不能为空，各操作函数到第11章再做介绍。

**●vm\_pgoff**：VMA起始偏移量，若为匿名共享映射VMA此值为0，匿名私有映射VMA为vm\_start表示的虚拟页帧号（vm\_start>>PAGE\_SHIFT）。如果是文件映射VMA则表示VMA起始页映射的文件内容在文件中的页偏移量，例如，VMA从文件内容pgoff处开始映射，则vm\_pgoff取值为pgoff>>PAGE\_SHIFT。

**●vm\_file：**文件映射VMA，指向映射文件file结构体实例。

内核启动阶段，start\_kernel()函数内调用proc\_caches\_init()函数为mm\_struct和vm\_area\_struct等数据结构创建slab缓存，函数在/kernel/fork.c文件内定义：

void \_\_init proc\_caches\_init(void)

{

...

**mm\_cachep** = kmem\_cache\_create("mm\_struct",

sizeof(struct **mm\_struct**), ARCH\_MIN\_MMSTRUCT\_ALIGN,

SLAB\_HWCACHE\_ALIGN|SLAB\_PANIC|SLAB\_NOTRACK, NULL);

**vm\_area\_cachep** = KMEM\_CACHE(**vm\_area\_struct**, SLAB\_PANIC);

mmap\_init(); /\*初始化计数值（vm\_committed\_as，percpu变量），/mm/mmap.c\*/

...

}

内核代码中需要创建mm\_struct和vm\_area\_struct实例时只需直接从对应的slab缓存分配即可。

#### 2管理结构

进程地址空间mm\_struct结构体中通过红黑树和链表管理VMA实例，相关成员如下所示：

struct mm\_struct {

struct vm\_area\_struct \*mmap; /\*指向vm\_area\_struct实例，构成链表\*/

struct rb\_root mm\_rb; /\*（扩展）红黑树根节点\*/

...

}

内存域vm\_area\_struct结构体中与mm\_struct中管理结构相关的成员如下：

struct vm\_area\_struct {

...

struct vm\_area\_struct \*vm\_next, \*vm\_prev; /\*指向前一个、下一个VMA\*/

struct rb\_node vm\_rb; /\*红黑树节点，添加到mm\_struct->mm\_rb红黑树\*/

...

}

vm\_area\_struct实例按起始虚拟地址升序在地址空间mm\_struct实例中的链表和红黑树中从左至右排序。mm\_struct实例mmap成员指向进程地址空间第一个内存域（最低地址），随后vm\_area\_struct实例通过vm\_next和vm\_prev成员组成链表。

mm\_struct实例mm\_rb成员为红黑树根节点，指向vm\_area\_struct实例vm\_rb成员，vm\_area\_struct实例通过vm\_rb成员插入到（扩展）红黑树中。以上结构如下图所示（假设地址空间中有5个VMA）：



mm\_struct结构体中VMA红黑树采用了扩展红黑树（参考第1章）。内核在/mm/mmap.c文件内定义了扩展红黑树的rb\_augment\_callbacks结构体实例：

RB\_DECLARE\_CALLBACKS(static, **vma\_gap\_callbacks**, struct vm\_area\_struct, **vm\_rb**, \

unsigned long, **rb\_subtree\_gap**, **vma\_compute\_subtree\_gap**)

rb\_augment\_callbacks结构体实例名称为vma\_gap\_callbacks，扩展信息更新的是内存域vm\_area\_struct实例rb\_subtree\_gap成员的值，计算扩展信息的函数为vma\_compute\_subtree\_gap()，定义如下：

static long vma\_compute\_subtree\_gap(struct vm\_area\_struct \*vma)

{

unsigned long max, subtree\_gap;

max = vma->vm\_start;

if (vma->vm\_prev) /\*若有前节点，则为当前节点与前节点之间空洞区长度，字节数\*/

max -= vma->vm\_prev->vm\_end;

/\*取max、左节点内存域rb\_subtree\_gap成员、右节点内存域rb\_subtree\_gap成员当中的最大值\*/

if (vma->vm\_rb.rb\_left) {

subtree\_gap = rb\_entry(vma->vm\_rb.rb\_left,struct vm\_area\_struct, vm\_rb)->rb\_subtree\_gap;

if (subtree\_gap > max)

max = subtree\_gap;

}

if (vma->vm\_rb.rb\_right) {

subtree\_gap = rb\_entry(vma->vm\_rb.rb\_right,struct vm\_area\_struct, vm\_rb)->rb\_subtree\_gap;

if (subtree\_gap > max)

max = subtree\_gap;

}

return max;

}

由于每次插入节点时，都会计算rb\_subtree\_gap值，rb\_subtree\_gap值会在红黑树中往上传递。因此，

rb\_subtree\_gap成员记录的是以vm\_area\_struct实例为根表示的子树当中，最大空洞区的长度。rb\_subtree\_gap成员值用于在创建映射前在地址空间中查找最合适的创建新映射的位置。

向地址空间红黑树中插入vm\_area\_struct实例节点的函数为：

rb\_insert\_augmented(&vma->vm\_rb, root, &vma\_gap\_callbacks)

从地址空间红黑树中移除vm\_area\_struct实例的函数为：

rb\_erase\_augmented(&vma->vm\_rb, root, &vma\_gap\_callbacks)

vm\_area\_struct实例依据红黑树在mm\_struct实例中的链表中排序，后面将介绍向地址空间插入VMA实例的操作函数。

### 4.3.3反向映射

进程虚拟内存页通过页表项映射到物理页帧，这称为正向映射（简称映射）。反向映射就是从物理页帧的角度往进程虚拟内存看，看物理页帧被哪些进程VMA映射（访问）。反向映射机制用来管理页帧映射到的VMA，以便于以解除映射。

对于文件映射，反向映射由文件地址空间address\_space结构体管理，映射文件内容的各vm\_area\_struct实例添加到address\_space实例管理的红黑树中（区间树）。

对于匿名映射，各进程每个匿名映射VMA对应一个匿名映射anon\_vma实例。映射页帧page实例关联到anon\_vma实例，anon\_vma实例中的红黑树（区间树）管理着所有后代进程（含本进程）源自于本VMA的VMA。通过扫描anon\_vma实例红黑树中关联的VMA，可获知page被映射到了哪些VMA。

在创建新VMA实例（文件映射）或在第一次为VMA建立映射时（匿名映射）会将vm\_area\_struct实例添加到反向映射管理结构中。在复制VMA（复制进程时）、调整（拆分、扩展等）VMA时会同时复制和调整vm\_area\_struct实例在反向映射结构中的状态。

本小节介绍反向映射管理结构，以及通过反向映射结构解除页所有映射的机制。

#### 1区间树

在介绍反向映射的结构之前，先来了解一下反向映射结构中使用的数据结构--区间树，区间树是一种扩展的红黑树（参考1.4小节）。在反向映射结构中区间树管理的是同一物理内存区域（物理页帧）被映射到的VMA。由于相同物理内存被映射到的各进程VMA之间虚拟地址可能相同或重叠，因此不能使用简单的红黑树来管理被映射到的VMA。

区间树中每个节点代表的对象包含一个区间信息，例如：地址区间[start,end]。对象实例在区间树中按区间起始值升序从左至右排序，起始值相同的实例在父节点的右子树。使用区间树时需要定义计算对象（节点）区间起始值和结束值的函数。区间树中每个对象的扩展信息是在以本对象为根节点的子树中，所有实例区间结束值的最大值。

内核定义区间树的宏在/include/linux/interval\_tree\_generic.h头文件声明：

#define INTERVAL\_TREE\_DEFINE(ITSTRUCT, ITRB, ITTYPE, ITSUBTREE, \

ITSTART, ITLAST, ITSTATIC, ITPREFIX) \

/\*

\*ITSTRUCT：区间树管理的对象数据结构，ITRB：对象数据结构中红黑树节点成员名称，

\*ITTYPE：扩展信息数据类型，ITSUBTREE：表示扩展信息的数据结构在对象结构中的成员，

\*ITSTART：计算对象区间起始值的函数指针，ITLAST：计算对象区间结束值的函数指针，

\*ITSTATIC：static或为空，ITPREFIX：各函数名称前缀。

\*/

static inline ITTYPE ITPREFIX ## \_compute\_subtree\_last(ITSTRUCT \*node) \

/\*计算指定节点（对象）扩展信息的函数，取子树节点中扩展信息最大值\*/

{ \

ITTYPE **max = ITLAST(node)**, subtree\_last; \ /\*扩展信息变量声明\*/

if (node->ITRB.rb\_left) { \

subtree\_last = rb\_entry(node->ITRB.rb\_left, \

ITSTRUCT, ITRB)->ITSUBTREE; \

if (max < subtree\_last) \

max = subtree\_last; \

} \

if (node->ITRB.rb\_right) { \

subtree\_last = rb\_entry(node->ITRB.rb\_right, \

ITSTRUCT, ITRB)->ITSUBTREE; \

if (max < subtree\_last) \

max = subtree\_last; \

} \

return max; \ /\*扩展信息为子树各节点区间结束值的最大值\*/

} \

\

**RB\_DECLARE\_CALLBACKS**(static, ITPREFIX ## \_augment, ITSTRUCT, ITRB, \

ITTYPE, ITSUBTREE, ITPREFIX ## \_compute\_subtree\_last) \

\

/\*定义扩展红黑树rb\_augment\_callbacks实例，名称为ITPREFIX ## \_augment\*/

/\*以下是定义区间树节点操作函数，如插入节点、删除节点等\*/ \

ITSTATIC void **ITPREFIX ## \_insert**(ITSTRUCT \*node, struct rb\_root \*root) \ /\*插入节点\*/

{ \

struct rb\_node \*\*link = &root->rb\_node, \*rb\_parent = NULL; \

ITTYPE start = ITSTART(node), last = ITLAST(node); \ /\*区间起始、结束地址\*/

ITSTRUCT \*parent; \

\

while (\*link) { \

rb\_parent = \*link; \

parent = rb\_entry(rb\_parent, ITSTRUCT, ITRB); \

if (parent->ITSUBTREE < last) \

parent->ITSUBTREE = last; \

if (start < ITSTART(parent)) \

link = &parent->ITRB.rb\_left; \

else \

link = &parent->ITRB.rb\_right; \

} \

\

node->ITSUBTREE = last; \ /\*节点扩展信息值\*/

**rb\_link\_node**(&node->ITRB, rb\_parent, link); \ /\*关联父节点（含颜色值）\*/

**rb\_insert\_augmented(&node->ITRB, root, &ITPREFIX ## \_augment)**; \ /\*插入节点\*/

} \

\

ITSTATIC void **ITPREFIX ## \_remove**(ITSTRUCT \*node, struct rb\_root \*root) \ /\*删除节点\*/

{ \

rb\_erase\_augmented(&node->ITRB, root, &ITPREFIX ## \_augment); \

}

\

/\*在以node为根节点的子树中查找区间值与[start,last]有重叠的子树中最左边节点（对象）\*/

static ITSTRUCT \* **ITPREFIX ## \_subtree\_search**(ITSTRUCT \*node, ITTYPE start, ITTYPE last) \

{ \

while (true) { \

if (node->ITRB.rb\_left) { \

ITSTRUCT \*left = rb\_entry(node->ITRB.rb\_left, \

ITSTRUCT, ITRB); \

if (start <= left->ITSUBTREE) { \

node = left; \

continue; \

} \

} \

if (ITSTART(node) <= last) { /\* Cond1 \*/ \

if (start <= ITLAST(node)) /\* Cond2 \*/ \

return node; /\* node is leftmost match \*/ \

if (node->ITRB.rb\_right) { \

node = rb\_entry(node->ITRB.rb\_right, \

ITSTRUCT, ITRB); \

if (start <= node->ITSUBTREE) \

continue; \

} \

} \

return NULL; /\* No match \*/ \

} \

} \

\

/\*在以root为根节点的树中，查找区间值与[start,last]有重叠的树中最左边节点（对象）\*/

ITSTATIC ITSTRUCT \* \

ITPREFIX ## \_iter\_first(struct rb\_root \*root, ITTYPE start, ITTYPE last) \

{ \

ITSTRUCT \*node; \

\

if (!root->rb\_node) \

return NULL; \

node = rb\_entry(root->rb\_node, ITSTRUCT, ITRB); \

if (node->ITSUBTREE < start) \

return NULL; \

return ITPREFIX ## \_subtree\_search(node, start, last); \

} \

\

/\*查找node节点之后，区间值与[start,last]有重叠的第一个节点（对象）\*/

ITSTATIC ITSTRUCT \* \

ITPREFIX ## \_iter\_next(ITSTRUCT \*node, ITTYPE start, ITTYPE last) \

{ \

struct rb\_node \*rb = node->ITRB.rb\_right, \*prev; \

\

while (true) { \

if (rb) { \

ITSTRUCT \*right = rb\_entry(rb, ITSTRUCT, ITRB); \

if (start <= right->ITSUBTREE) \

return ITPREFIX ## \_subtree\_search(right, \

start, last); \

} \

\

/\* Move up the tree until we come from a node's left child \*/ \

do { \

rb = rb\_parent(&node->ITRB); \

if (!rb) \

return NULL; \

prev = &node->ITRB; \

node = rb\_entry(rb, ITSTRUCT, ITRB); \

rb = node->ITRB.rb\_right; \

} while (prev == rb); \

\

/\* Check if the node intersects [start;last] \*/ \

if (last < ITSTART(node)) /\* !Cond1 \*/ \

return NULL; \

else if (start <= ITLAST(node)) /\* Cond2 \*/ \

return node; \

} \

}

INTERVAL\_TREE\_DEFINE()宏实际就是定义了区间树（节点）操作函数，简列如下所示：

●void **ITPREFIX ## \_insert**(ITSTRUCT \*node, struct rb\_root \*root)：向root为根节点的区间树插入node节点。

●void **ITPREFIX ## \_remove**(ITSTRUCT \*node, struct rb\_root \*root)：在区间树中移除node节点。

●static ITSTRUCT \***ITPREFIX ## \_subtree\_search**(ITSTRUCT \*node, ITTYPE start, ITTYPE last)：在node为根节点的子树中查找区间值与[start,last]有重叠的子树中最左边节点（对象）。

●ITSTRUCT \***ITPREFIX ## \_iter\_first**(struct rb\_root \*root, ITTYPE start, ITTYPE last)：在以root为根节点的树中，查找区间值与[start,last]有重叠的树中最左边节点（对象）。

●ITSTRUCT \***ITPREFIX ## \_iter\_next**(ITSTRUCT \*node, ITTYPE start, ITTYPE last)：查找node节点之后，区间值与[start,last]有重叠的第一个节点（对象）。

#### 2文件反向映射

文件映射的反向映射结构如下图所示，在内核中文件由inode实例唯一表示（file是进程文件的表示），inode结构体包含文件地址空间address\_space结构体实例，address\_space实际上是文件内容在内存中的缓存，address\_space通过基数树管理缓存文件内容的页帧（页缓存）。不同进程的VMA或者同一进程的不同VMA可以同时映射到同一个文件，如图中所示，进程1和进程2都有VMA映射到文件。vm\_area\_struct结构体中包含VMA映射到文件内容的起始偏移量（vm\_pgoff，页偏移量）和长度信息（映射区域长度）。不同VMA映射到文件内容的不同区域，映射内容有可能会重叠。如下图中所示，进程1VMA映射文件内容的起始页偏移量为2，长度为4页，进程2VMA映射文件内容的起始页偏移量为1，长度为3页



文件地址空间address\_space实例中通过页缓存，管理文件内容缓存。页缓存线性映射到块设备中的文件内容。文件映射内存域对应页表项指向文件缓存页，以实现文件映射。缓存页page实例的mapping成员指向所属文件地址空间address\_space实例，index成员表示缓存的文件内容在文件中的页偏移量，例如，index=0表示缓存的是文件首页的内容。注意index成员值并不是表示page在VMA中的偏移量。

映射到文件的虚拟内存域vm\_area\_struct实例通过shared.rb成员插入到以文件地址空间address\_space实例i\_mmap成员为根节点的区间树，如下图所示。



映射页page实例通过mapping成员指向所属文件地址空间address\_space实例。其中mapping成员值低2位用于标识类型，bit0为0表示文件映射，为1表示匿名映射，bit1为1表示为KSM映射。

address\_space实例通过基数树管理映射页page实例，页在基数树中的索引值为映射文件内容的页偏移量。

若要查找page实例表示的页映射到哪些VMA，可查找文件地址空间中的区间树节点（vm\_area\_struct实例），各节点的区间值是映射文件内容的页偏移量范围。如果page实例index值位于节点区间值内部，表示此页映射到此VMA。

例如，如上图中所示，对于index=3的page实例，查找内区间树，可以发现进程1和进程2的VMA都映射了此页，进程1VMA起始偏移量为2，则page映射到VMA中的第2页，进程2VMA起始偏移量为1，则page映射到VMA中的第3页。

vm\_area\_struct结构体中与文件映射反向映射结构相关的成员如下所示：

struct vm\_area\_struct {

...

struct {

struct rb\_node rb;/\*将实例添加到文件地址空间address\_space->i\_mmap区间树/红黑树\*/

unsigned long rb\_subtree\_last; /\*子树所有VMA中映射偏移量最大值，扩展信息\*/

} **shared;**

...

}

vm\_area\_struct结构体中shared.rb成员用于将实例添加到文件地址空间区间树，shared.rb\_subtree\_last成员记录扩展信息，即子树VMA中映射文件内容最大页偏移量。

内核在/mm/interval\_tree.c文件内定义了文件反向映射结构中区间树的操作函数：

INTERVAL\_TREE\_DEFINE**(struct vm\_area\_struct, shared.rb, \**

**unsigned long, shared.rb\_subtree\_last, vma\_start\_pgoff, vma\_last\_pgoff,, vma\_interval\_tree)**

区间树中节点区间值为VMA映射文件内容的起止页偏移量，计算函数如下：

static inline unsigned long vma\_start\_pgoff(struct vm\_area\_struct \*v) /\*计算区间起始值函数\*/

{

return v->vm\_pgoff; /\*映射内容起始页偏移量\*/

}

static inline unsigned long vma\_last\_pgoff(struct vm\_area\_struct \*v) /\*计算区间结束值函数\*/

{

return v->vm\_pgoff + ((v->vm\_end - v->vm\_start) >> PAGE\_SHIFT) - 1; /\*映射内容最大页偏移量\*/

}

由以上定义可知，vm\_area\_struct实例以起始映射页偏移量升序在区间树中从左至右排序，相同起始偏移量的实例在右子树中。

以上INTERVAL\_TREE\_DEFINE()宏中定义的插入vm\_area\_struct实例的函数为：

void **vma\_interval\_tree\_insert**( (struct vm\_area\_struct \*node, struct rb\_root \*root)；

移除vm\_area\_struct实例的函数为：

void **vma\_interval\_tree\_remove**(struct vm\_area\_struct \*node, struct rb\_root \*root)；

以下两个函数配合用于遍历区间树中所有与[start,last]区间值有重叠的VMA实例：

struct vm\_area\_struct \***vma\_interval\_tree\_iter\_first**(struct rb\_root \*root,

unsigned long start, unsigned long last);

struct vm\_area\_struct \***vma\_interval\_tree\_iter\_next**(struct vm\_area\_struct \*node,

unsigned long start, unsigned long last);

用户进程在创建或复制文件映射VMA时，需要将VMA插入文件地址空间区间树，删除VMA时从区间树中移除。如果是对现有VMA进行调整，则需要先将其从区间树中移出，调整完后再插入即可。

释放文件缓存页时，需要扫描所属文件地址空间VMA区间树，找到所有映射了该页的VMA，清零各VMA映射该页的页表项。如果页内容被修改了（脏页）则通知文件地址空间将其回写，页引用计数值减1，当页引用计数为0时，将释放页。

#### 3匿名反向映射

进程匿名映射VMA会通过复制进程传递给子进程，各VMA之间构成父子关系（祖先和后代关系）。因为本VMA的映射关系会通过VMA复制传递给所有后代匿名映射VMA，因此VMA对应的反向映射结构需要管理所有源于本VMA（含本VMA）的后代VMA，以便在解除映射时找到所有映射了本页的VMA。

##### ■管理结构

vm\_area\_struct结构体中与匿名反向映射结构相关的成员有：

struct vm\_area\_struct {

...

struct list\_head anon\_vma\_chain; /\*双链表头，链接关联本VMA的anon\_vma\_chain实例\*/

struct anon\_vma \*anon\_vma; /\*指向本VMA对应的anon\_vma实例\*/

...

}

vm\_area\_struct结构体中相关成员简介如下：

●anon\_vma：指向本VMA对应的反向映射结构anon\_vma实例。

●anon\_vma\_chain：双链表头，链接anon\_vma\_chain结构体实例，用于将内存域与自身对应anon\_vma实例关联，以及将内存域关联到所有祖先内存域对应的anon\_vma实例。

anon\_vma结构体定义在/include/linux/rmap.h头文件内：

struct anon\_vma {

struct anon\_vma \*root; /\*指向源VMA关联的anon\_vma实例（最早的祖先）\*/

struct rw\_semaphore rwsem; /\*读写信号量\*/

atomic\_t refcount; /\*用于root节点anon\_vma实例统计后代VMA数量（不含自身）\*/

unsigned degree; /\*degree减1值表示直接由本VMA复制产生的VMA数量（含自身）\*/

struct anon\_vma \*parent; /\*指向父VMA对应的anon\_vma实例\*/

struct rb\_root **rb\_root**; /\*区间树根节点，管理anon\_vma\_chain实例，用于集合后代VMA\*/

};

anon\_vma\_chain结构体用于关联vm\_area\_struct和anon\_vma实例，定义如下（/include/linux/rmap.h）：

struct anon\_vma\_chain {

struct vm\_area\_struct \***vma**; /\*关联的VMA\*/

struct anon\_vma **\*anon\_vma**; /\*anon\_vma可能是本VMA关联的实例，

\*也可能是祖先VMA关联的VMA\*/

struct list\_head **same\_vma**; /\*添加到vm\_area\_struct.anon\_vma\_chain双链表\*/

struct rb\_node **rb**; /\*红黑树节点，添加到anon\_vma实例中区间树\*/

unsigned long **rb\_subtree\_last**; /\*本节点表示的子树关联的VMA最大结束偏移量\*/

#ifdef CONFIG\_DEBUG\_VM\_RB

unsigned long cached\_vma\_start, cached\_vma\_last;

#endif

};

anon\_vma结构体类似于文件地址空间的address\_space结构体，也就是每个VMA都视为一个地址空间，所有后代的VMA（含自身）都添加到anon\_vma实例管理的区间树，区间树通过anon\_vma\_chain实例关到各VMA。VMA中包含一个anon\_vma\_chain实例双链表，用于将VMA添加到自身及所有祖先VMA关联的anon\_vma实例区间树中。

下面通过一个简单的例子来说明以上数据结构的关系，如下图所示。假设父进程中具有VMA0，创建子进程时，在子进程生成VMA1。VMA0中建立页映射时创建anon\_vma实例anon\_vma0，以及anon\_vma\_chain实例avc0，avc0添加到VMA0中的双链表和anon\_vma0中区间树。VMA0中映射页帧page实例的mapping成员指向anon\_vma0实例。

复制生成VMA1时，将扫描父VMA中anon\_vma\_chain双链表中实例，对每个实例复制一个副本，副本关联到VMA1，以及原anon\_vma\_chain实例关的anon\_vma实例。下图中avc1复制于avc0，avc1关联到VMA1和**anon\_vma0**。然后，还要为VMA1创建属于自己的anon\_vma和anon\_vma\_chain实例，并建立关联，如图中所示。



内核在启动函数start\_kernel()内调用anon\_vma\_init()函数为匿名反向映射数据结构创建slab缓存，函数定义如下（/mm/rmap.c）：

void \_\_init anon\_vma\_init(void)

{

anon\_vma\_cachep = kmem\_cache\_create("anon\_vma", sizeof(struct **anon\_vma**),

0, SLAB\_DESTROY\_BY\_RCU|SLAB\_PANIC, anon\_vma\_ctor);

anon\_vma\_chain\_cachep = KMEM\_CACHE(**anon\_vma\_chain**, SLAB\_PANIC);

}

内核在/mm/interval\_tree.c文件内定义了匿名反向映射结构中区间树的操作函数，与文件反向映射区间树类似：

INTERVAL\_TREE\_DEFINE(**struct anon\_vma\_chain, rb, unsigned long, rb\_subtree\_last, \**

**avc\_start\_pgoff, avc\_last\_pgoff,static inline, \_\_anon\_vma\_interval\_tree**)

static inline unsigned long avc\_start\_pgoff(struct anon\_vma\_chain \*avc) /\*获取区间起始值函数\*/

{

return vma\_start\_pgoff(avc->vma); /\*VMA起始虚拟页帧号\*/

}

static inline unsigned long avc\_last\_pgoff(struct anon\_vma\_chain \*avc) /\*获取区间结束值函数\*/

{

return vma\_last\_pgoff(avc->vma); /\*VMA结束虚拟页帧号\*/

}

向区间树插入节点anon\_vma\_chain实例的函数为：

**void \_\_anon\_vma\_interval\_tree\_insert(struct anon\_vma\_chain \*node,struct rb\_root \*root);**

从区间树移出节点的函数为：

**void \_\_anon\_vma\_interval\_tree\_remove(struct anon\_vma\_chain \*node,struct rb\_root \*root);**

以下两个函数配合用于遍历区间树中所有与[start,last]区间值有重叠的anon\_vma\_chain实例：

struct anon\_vma\_chain \***anon\_vma\_interval\_tree\_iter\_first**(

struct rb\_root \*root, unsigned long start, unsigned long last);

struct anon\_vma\_chain \***anon\_vma\_interval\_tree\_iter\_next**(

struct anon\_vma\_chain \*node, unsigned long start, unsigned long last);

##### **■创建反向映射结构**

进程新创建匿名映射内存域，在第一次创建物理页帧映射时创建对应的anon\_vma实例，函数定义如下（/mm/rmap.c）：

int anon\_vma\_prepare(struct vm\_area\_struct \*vma)

{

struct anon\_vma \*anon\_vma **= vma->anon\_vma**; /\*VMA原关联的anon\_vma实例\*/

struct anon\_vma\_chain \*avc;

might\_sleep();

if (unlikely(!anon\_vma)) { /\*vma->anon\_vma是否为空，若不为空则函数直接返回0\*/

struct mm\_struct \*mm = vma->vm\_mm;

struct anon\_vma \*allocated;

avc = **anon\_vma\_chain\_alloc(GFP\_KERNEL)**; /\*从slab缓存中分配anon\_vma\_chain实例\*/

if (!avc)

goto out\_enomem;

anon\_vma = find\_mergeable\_anon\_vma(vma);

/\*检查是否可与相邻VMA共用anon\_vma实例，/mm/mmap.c\*/

allocated = NULL;

if (!anon\_vma) { /\*需要分配anon\_vma实例\*/

anon\_vma = **anon\_vma\_alloc()**;

/\*从slab缓存中分配anon\_vma实例，parent，root指向成员自身\*/

if (unlikely(!anon\_vma))

goto out\_enomem\_free\_avc;

allocated = anon\_vma;

}

anon\_vma\_lock\_write(anon\_vma);

spin\_lock(&mm->page\_table\_lock);

if (likely(!vma->anon\_vma)) {

vma->anon\_vma = anon\_vma; /\*anon\_vma实例赋予VMA\*/

**anon\_vma\_chain\_link(vma, avc, anon\_vma)**; /\*将实例添加到管理结构，/mm/rmap.c\*/

**anon\_vma->degree++**; /\*degree=2\*/

allocated = NULL;

avc = NULL;

}

spin\_unlock(&mm->page\_table\_lock);

anon\_vma\_unlock\_write(anon\_vma);

...

} /\*if结束\*/

return 0;

...

}

anon\_vma\_prepare()函数内判断vma->anon\_vma成员是否为空，若不为空则函数直接返回0，不需要创建反向映射结构。若为空则表示需要创建anon\_vma实例，随后首先创建anon\_vma\_chain实例，调用函数find\_mergeable\_anon\_vma(vma)判断是否可以与相邻VMA共用anon\_vma实例，若可以则共用（不创建），若不能共用则调用anon\_vma\_alloc()函数创建并初始化anon\_vma实例。最后，调用anon\_vma\_chain\_link()函数将创建的anon\_vma\_chain和anon\_vma实例添加到管理结构中，并设置anon\_vma\_chain实例。

anon\_vma\_alloc()函数定义如下（/mm/rmap.c）：

static inline struct anon\_vma \*anon\_vma\_alloc(void)

{

struct anon\_vma \*anon\_vma;

anon\_vma = kmem\_cache\_alloc(anon\_vma\_cachep, GFP\_KERNEL); /\*分配anon\_vma实例\*/

if (anon\_vma) {

atomic\_set(&anon\_vma->refcount, **1**); /\*引用计数值设为1\*/

anon\_vma->degree = **1**; /\*degree值设为1\*/

anon\_vma->parent = anon\_vma; /\*指向自身\*/

anon\_vma->root = anon\_vma; /\*指向自身\*/

}

return anon\_vma; /\*返回anon\_vma实例指针\*/

}

anon\_vma\_chain\_link()函数定义如下（/mm/rmap.c）：

static void anon\_vma\_chain\_link(struct vm\_area\_struct \*vma,struct anon\_vma\_chain \*avc, \

struct anon\_vma \*anon\_vma)

{

avc->vma = vma;

avc->anon\_vma = anon\_vma;

**list\_add(&avc->same\_vma, &vma->anon\_vma\_chain)**;

/\*anon\_vma\_chain添加到vm\_area\_struct 链表\*/

**anon\_vma\_interval\_tree\_insert(avc, &anon\_vma->rb\_root)**; /\*/mm/interval\_tree.c\*/

/\*anon\_vma\_chain添加到anon\_vma->rb\_root管理的区间树\*/

}

anon\_vma\_interval\_tree\_insert()函数调用\_\_anon\_vma\_interval\_tree\_insert()函数将anon\_vma\_chain添加到anon\_vma->rb\_root管理的区间树。

anon\_vma\_prepare()函数创建的数据结构实例如下图所示：



##### **■复制反向映射结构**

用户进程通过fork()等系统调用创建进程时，子进程将复制父进程的信息，包括进程地址空间信息。复制进程地址空间信息的函数调用关系简列如下：



在dup\_mm()函数中将为子进程创建mm\_struct实例，复制父进程mm\_struct实例数据，并调用函数dup\_mmap()复制父进程各VMA信息至子进程。在dup\_mmap()函数中将遍历父进程VMA，对每个VMA在子进程中复制一个VMA实例（包括VMA对应的页表项），并调用anon\_vma\_fork()函数为子进程VMA复制/创建反向映射信息。

anon\_vma\_fork()函数用于为子进程复制匿名映射VMA的反向映射信息，函数定义如下（/mm/rmap.c）：

int anon\_vma\_fork(struct vm\_area\_struct \*vma, struct vm\_area\_struct \*pvma)

/\*vma：子进程vm\_area\_struct实例，pvma：父进程vm\_area\_struct实例\*/

{

struct anon\_vma\_chain \*avc;

struct anon\_vma \*anon\_vma;

int error;

if (!pvma->anon\_vma) /\*anon\_vma成员为空的VMA，不用复制，直接返回。

return 0; /\*可能是还没有建立任何映射的匿名映射VMA，也可能是文件映射VMA\*/

**vma->anon\_vma = NULL**; /\*清空anon\_vma指针\*/

error = **anon\_vma\_clone(vma, pvma)**; /\*复制父VMA反向映射信息，成功返回0，/mm/rmap.c\*/

if (error)

return error;

if (vma->anon\_vma)

return 0;

/\*创建VMA自身对应的anon\_vma实例\*/

anon\_vma = anon\_vma\_alloc(); /\*分配anon\_vma实例并初始化\*/

if (!anon\_vma)

goto out\_error;

avc = anon\_vma\_chain\_alloc(GFP\_KERNEL); /\*分配anon\_vma\_chain实例\*/

if (!avc)

goto out\_error\_free\_anon\_vma;

anon\_vma->root = pvma->anon\_vma->root; /\*设置根anon\_vma实例\*/

anon\_vma->parent = pvma->anon\_vma; /\*设置父anon\_vma实例\*/

**get\_anon\_vma(anon\_vma->root)**;

/\*增加根anon\_vma实例refcount引用计数，/include/linux/rmap.h\*/

**vma->anon\_vma = anon\_vma**; /\*指向新创建anon\_vma实例\*/

anon\_vma\_lock\_write(anon\_vma);

**anon\_vma\_chain\_link(vma, avc, anon\_vma)**; /\*添加anon\_vma\_chain实例到反向映射管理结构\*/

**anon\_vma->parent->degree++**; /\*增加父anon\_vma实例degree计数值\*/

anon\_vma\_unlock\_write(anon\_vma);

return 0;

...

}

anon\_vma\_fork()函数执行的工作主要分两步，简述如下：

（1）调用anon\_vma\_clone()函数扫描父VMA的anon\_vma\_chain双链表，对每个anon\_vma\_chain实例（旧实例）为子VMA创建一个新anon\_vma\_chain实例，新实例关联到旧实例关联的anon\_vma实例，以及子进程VMA实例。新anon\_vma\_chain实例添加到子VMA实例中的anon\_vma\_chain双链表头部，以及旧anon\_vma\_chain实例关联anon\_vma实例的红黑树。

anon\_vma\_clone(dst, src)函数定义如下（/mm/rmap.c）：

int anon\_vma\_clone(struct vm\_area\_struct \*dst, struct vm\_area\_struct \*src)

/\*dst：子进程VMA指针，src：父进程VMA指针\*/

{

struct anon\_vma\_chain \*avc, \*pavc;

struct anon\_vma \*root = NULL;

/\*遍历父VMA中anon\_vma\_chain双链表\*/

list\_for\_each\_entry\_reverse(**pavc**, &src->anon\_vma\_chain, same\_vma) {

struct anon\_vma \*anon\_vma;

avc = anon\_vma\_chain\_alloc(GFP\_NOWAIT | \_\_GFP\_NOWARN);

/\*分配新anon\_vma\_chain实例\*/

if (unlikely(!avc)) {

... /\*不成功，再次尝试分配anon\_vma\_chain实例\*/

}

anon\_vma **= pavc->anon\_vma**; /\*旧anon\_vma\_chain实例关联的anon\_vma实例\*/

root = lock\_anon\_vma\_root(root, anon\_vma); /\*根anon\_vma实例\*/

**anon\_vma\_chain\_link(dst, avc, anon\_vma)**; /\*新anon\_vma\_chain实例添加到管理结构\*/

if (!dst->anon\_vma && anon\_vma != src->anon\_vma &&anon\_vma->degree < 2)

/\*复用anon\_vma实例，此条件何时满足？？？\*/

dst->anon\_vma = anon\_vma;

} /\*遍历anon\_vma\_chain链表结束\*/

if (dst->anon\_vma) /\*如果复用了现有anon\_vma实例，增加其degree值\*/

dst->anon\_vma->degree++;

unlock\_anon\_vma\_root(root);

return 0; /\*成功返回0\*/

...

}

下面通过一个例子来说明anon\_vma\_clone(dst, src)函数的功能。如下图所示，假设子进程VMA1复制于父进程VMA0。由于VMA0中anon\_vma\_chain双链表只有一个成员avc0，因此在anon\_vma\_clone()函数中将只创建一个anon\_vma\_chain实例acv1。acv1关联到子进程VMA1以及avc0关联的anon\_vma实例anon\_vma0。acv1添加到VMA1中的双链表和anon\_vma0中的区间树。



（2）anon\_vma\_fork()函数的第二步是为子进程VMA创建自己的anon\_vma和anon\_vma\_chain实例，并添加到管理结构中。如下图所示，为VMA1创建了anon\_vma实例anon\_vma1，anon\_vma\_chain实例avc2，并将avc2实例添加到VMA1双链表及anon\_vma1中区间树。



假设子进程又创建了子进程（孙进程），VMA2复制于VMA1，则VMA2的反向映射数据结构如下图所示：



为VMA3复制反向映射结构时，anon\_vma\_clone()函数扫描VMA1中anon\_vma\_chain实例双链表，分别创建avc3（对应avc2）和avc4（对应avc1）实例，avc3添加到anon\_vma1实例的区间树，avc4添加到anon\_vma0实例的区间树。avc5添加到VMA3对应anon\_vma2实例的区间树。

另外，各anon\_vma实例之间通过parent成员指示父子关系，root成员指向祖先anon\_vma实例。例如上图中的anon\_vma1和anon\_vma2实例的root成员都指向anon\_vma0实例。

由上可知，anon\_vma\_chain实例用于建立anon\_vma实例与vm\_area\_struct实例之间的关联。anon\_vma实例通过区间树中的anon\_vma\_chain实例建立与其所有后代（含自己）VMA及反向映射结构之间的关联。

vm\_area\_struct实例通过anon\_vma\_chain双链表中的实例建立与其所有祖先VMA及反向映射结构之间的关联。

在为VMA建立映射时，映射页帧page实例mapping成员指向VMA关联的anon\_vma实例，index成员值为VMA中vm\_pgoff值加上映射虚拟页相对于VMA起始位置的页偏移量，其实就是虚拟页的虚拟页帧号。在复制VMA时会复制VMA对应的页表项，也就是说映射关系会随复制操作传递给子进程。

若要查找page实例映射到哪些VMA中，可通过mapping成员获取初始建立映射VMA对应的anon\_vma实例，由搜索其区间树anon\_vma\_chain实例可获知page映射到了哪些VMA，以便于解除page页帧的所有映射。

举个例子有助于理解，如下图所示，假设VMA1由VMA0复制而来，anon\_vma0代表VMA0对应的anon\_vma实例，page1和page2的映射在复制VMA1前创建，anon\_vma0区间树中关联了VMA0和VMA1。VMA0从虚拟页帧0开始（vm\_pgoff=0），而VMA1在复制之后进程解除了0号和1号虚拟页帧的映射（vm\_pgoff=2）。查找page1映射的虚拟页时，由anon\_vma0搜索到VMA0和VMA1，VMA0起始偏移量vm\_pgoff=0，小于page1实例index值（1），说明此映射区包含了此页，且相对VMA起始页偏移量为index-vm\_pgoff=1，即映射到VMA的第二页。VMA1起始虚拟页帧号vm\_pgoff=2大于page1实例index值，表示page1没有映射到此VMA。同理可知，page2同时映射到VMA0和VMA1，只不过VMA内页偏移量不同。



如果进程在调用fork()系统调用创建进程后，子进程调用了execve()系统调用运行新的可执行目标文件，则子进程的地址空间将被复位，将清除fork()系统调用中复制的VMA及其反向映射信息等。

#### 4解除页所有映射

在页面回收操作中，被选中回收的页在被释放前需要解除其所有映射关系（详见第11章）。页面回收操作中调用**try\_to\_unmap**(struct page \*page, enum ttu\_flags flags)函数解除page表示页的所有映射关系。参数flags表示解除页映射标记，取值定义如下（/include/linux/rmap.h）：

enum ttu\_flags {

**TTU\_UNMAP = 1**, /\*解除映射模式\*/

TTU\_MIGRATION = 2, /\* migration mode \*/

TTU\_MUNLOCK = 4, /\*不解除映射\*/

TTU\_IGNORE\_MLOCK = (1 << 8), /\*忽略VMA的锁定标记，强制解除\*/

TTU\_IGNORE\_ACCESS = (1 << 9), /\* don't age \*/

TTU\_IGNORE\_HWPOISON = (1 << 10), /\* corrupted page is recoverable \*/

};

在介绍解除映射函数前，先了解一下rmap\_walk\_control控制结构，定义如下（/include/linux/rmap.h）：

struct rmap\_walk\_control {

void \*arg; /\*传递给rmap\_one()函数的参数\*/

int (\***rmap\_one**)(struct page \*page, struct vm\_area\_struct \*vma,unsigned long addr, void \*arg);

/\*解除单页page到VMA映射关系的函数\*/

int (\*done)(struct page \*page); /\*解除页到VMA的映射后的回调函数\*/

struct anon\_vma \*(\*anon\_lock)(struct page \*page);

bool (\*invalid\_vma)(struct vm\_area\_struct \*vma, void \*arg); /\*检查VMA是否无效的函数\*/

};

解除页映射函数try\_to\_unmap()定义在/mm/rmap.c文件内，代码如下：

int try\_to\_unmap(struct page \*page, enum ttu\_flags flags)

/\*页面回收操作中调用此函数flags参数为**TTU\_UNMAP**\*/

{

int ret;

struct rmap\_walk\_control rwc = { /\*设置rmap\_walk\_control实例\*/

.rmap\_one = **try\_to\_unmap\_one**, /\*解除页到某个VMA映射的函数，/mm/rmap.c\*/

.arg = (void \*)**flags**, /\*标记参数\*/

.done = page\_not\_mapped, /\*返回!page\_mapped(page)，/mm/rmap.c\*/

.anon\_lock = page\_lock\_anon\_vma\_read,

};

VM\_BUG\_ON\_PAGE(!PageHuge(page) && PageTransHuge(page), page);

if ((flags & TTU\_MIGRATION) && !PageKsm(page) && PageAnon(page))

rwc.invalid\_vma = invalid\_migration\_vma;

ret = **rmap\_walk(page, &rwc)**; /\*扫描页映射到的VMA并逐个解除映射，/mm/rmap.c\*/

if (ret != SWAP\_MLOCK && !page\_mapped(page))

ret = SWAP\_SUCCESS;

return ret;

}

try\_to\_unmap()函数定义并设置rmap\_walk\_control控制结构，调用rmap\_walk()函数解除映射，函数定义如下（/mm/rmap.c）：

int rmap\_walk(struct page \*page, struct rmap\_walk\_control \*rwc)

{

if (unlikely(PageKsm(page))) /\*KSM页（内核相同页合并机制）\*/

return rmap\_walk\_ksm(page, rwc);

else if (PageAnon(page))

return **rmap\_walk\_anon(page, rwc)**; /\*解除匿名映射页\*/

else

return **rmap\_walk\_file(page, rwc)**; /\*解除文件映射页\*/

}

rmap\_walk()函数对匿名映射页和文件映射页调用不同的处理函数，但是函数执行流程类似，都是遍历反向映射结构，找出page映射到各VMA的虚拟地址，对每个VMA调用rwc->rmap\_one()函数解除page到此VMA的映射。

##### ■扫描映射VMA

rmap\_walk\_anon()函数扫描匿名映射页关联的反向映射结构，逐个解除到各VMA的映射，函数定义如下（/mm/rmap.c）：

static int rmap\_walk\_anon(struct page \*page, struct rmap\_walk\_control \*rwc)

{

struct anon\_vma \*anon\_vma;

pgoff\_t pgoff;

struct anon\_vma\_chain \*avc;

int ret = SWAP\_AGAIN;

anon\_vma = **rmap\_walk\_anon\_lock**(page, rwc); /\*匿名反映射结构anon\_vma实例\*/

if (!anon\_vma) /\*文件映射返回NULL\*/

return ret;

pgoff = page\_to\_pgoff(page); /\*映射页的虚拟页帧号\*/

anon\_vma\_interval\_tree\_foreach(avc, &anon\_vma->rb\_root, pgoff, pgoff) {

/\*遍历反向映射结构中与[pgoff,paoff]有重叠的anon\_vma\_chain实例\*/

struct vm\_area\_struct \*vma = avc->vma; /\*关联的VMA\*/

unsigned long address = vma\_address(page, vma); /\*page在VMA中的映射（起始）地址\*/

if (rwc->invalid\_vma && rwc->invalid\_vma(vma, rwc->arg))

continue;

**ret** = **rwc->rmap\_one**(page, vma, address, rwc->arg);

/\*调用**try\_to\_unmap\_one()**函数，解除映射\*/

if (**ret != SWAP\_AGAIN**) /\*如果返回值不是**SWAP\_AGAIN，**遍历操作中止，跳出循环\*/

break;

if (**rwc->done && rwc->done(page)**) /\*调用rwc->done(page)函数\*/

break;

} /\*遍历反向映射结构结束\*/

anon\_vma\_unlock\_read(anon\_vma);

return ret;

}

rmap\_walk\_anon()函数获取page关联的anon\_vma实例（在建立映射时赋值，详见下文），遍历其中的区间树，找到与[pgoff,paoff]区间值有重叠的anon\_vma\_chain实例，对每个anon\_vma\_chain实例关联的VMA调用rwc->rmap\_one()函数，即**try\_to\_unmap\_one()**函数，解除页到此VMA的映射。

rmap\_walk\_file()函数用于解除文件映射页的映射，函数定义如下（/mm/rmap.c）：

static int rmap\_walk\_file(struct page \*page, struct rmap\_walk\_control \*rwc)

{

struct address\_space \*mapping = page->mapping; /\*文件地址空间，建立映射时赋值\*/

pgoff\_t pgoff;

struct vm\_area\_struct \*vma;

int ret = SWAP\_AGAIN;

VM\_BUG\_ON\_PAGE(!PageLocked(page), page);

if (!mapping)

return ret;

pgoff = page\_to\_pgoff(page); /\*page在文件地址空间中的页偏移量\*/

i\_mmap\_lock\_read(mapping);

vma\_interval\_tree\_foreach(vma, &mapping->i\_mmap, pgoff, pgoff) {

/\*遍历文件地址空间中区间树，找到与[pgoff,paoff]区间值有重叠的VMA实例\*/

unsigned long address = vma\_address(page, vma); /\*页映射到VMA的（起始）地址\*/

if (rwc->invalid\_vma && rwc->invalid\_vma(vma, rwc->arg)) /\*VMA是否无效\*/

continue;

ret = **rwc->rmap\_one**(page, vma, address, rwc->arg); /\*调用try\_to\_unmap\_one()函数\*/

if (ret != SWAP\_AGAIN)

goto done;

if (rwc->done && rwc->done(page)) /\*回调函数\*/

goto done;

}

done:

i\_mmap\_unlock\_read(mapping);

return ret;

}

rmap\_walk\_file()函数与rmap\_walk\_anon()函数相似，只不过此处遍历的是文件地址空间中的区间树，对每个区间值与[pgoff,paoff]有重叠的VMA实例调用try\_to\_unmap\_one()函数解除映射。

##### ■解除单个映射

try\_to\_unmap\_one()函数用于解除页到单个VMA的映射，它同时适用于解除匿名映射页和文件映射页，函数定义如下（/mm/rmap.c）：

static int try\_to\_unmap\_one(struct page \*page, struct vm\_area\_struct \*vma, \

unsigned long address, void \*arg)

{

struct mm\_struct \*mm = vma->vm\_mm; /\*进程地址空间\*/

pte\_t \*pte;

pte\_t pteval;

spinlock\_t \*ptl;

int ret = SWAP\_AGAIN;

enum ttu\_flags flags = (enum ttu\_flags)arg;

pte = **page\_check\_address(page, mm, address, &ptl, 0)**;

/\*页所对应的页表项指针，/include/linux/rmap.h\*/

if (!pte)

goto out;

if (!(flags & TTU\_IGNORE\_MLOCK)) { /\*没有设置TTU\_IGNORE\_MLOCK标记\*/

if (vma->vm\_flags & VM\_LOCKED) /\*VMA锁定了页\*/

goto out\_mlock;

if (flags & TTU\_MUNLOCK)

goto out\_unmap;

}

if (!(flags & TTU\_IGNORE\_ACCESS)) {

if (ptep\_clear\_flush\_young\_notify(vma, address, pte)) {

ret = SWAP\_FAIL;

goto out\_unmap;

}

}

flush\_cache\_page(vma, address, page\_to\_pfn(page)); /\*刷新缓存\*/

pteval = **ptep\_clear\_flush**(vma, address, pte); /\*清零内存页表项，返回原页表项值\*/

if (pte\_dirty(pteval)) /\*脏页\*/

**set\_page\_dirty(page);**  /\*设置页脏，会调用文件地址空间中相应操作，如果需要\*/

update\_hiwater\_rss(mm); /\*更新水印值\*/

if (PageHWPoison(page) && !(flags & TTU\_IGNORE\_HWPOISON)) {

if (!PageHuge(page)) {

if (PageAnon(page))

dec\_mm\_counter(mm, MM\_ANONPAGES);

else

dec\_mm\_counter(mm, MM\_FILEPAGES);

}

set\_pte\_at(mm, address, pte,swp\_entry\_to\_pte(make\_hwpoison\_entry(page)));

} else if (pte\_unused(pteval)) {

if (PageAnon(page))

dec\_mm\_counter(mm, MM\_ANONPAGES);

else

dec\_mm\_counter(mm, MM\_FILEPAGES);

} else if (**PageAnon(page)**) { /\*处理在使用的匿名映射页\*/

swp\_entry\_t entry = { .**val = page\_private(page)** };

/\*页交换机制中交换区位置已写入page->private\*/

pte\_t swp\_pte;

if (PageSwapCache(page)) {

if (**swap\_duplicate(entry)** < 0) { /\*交换区swap\_map[]数组项值加1，/mm/swapfile.c\*/

set\_pte\_at(mm, address, pte, pteval); /\*写出到交换区失败，重新写回页表项\*/

ret = SWAP\_FAIL;

goto out\_unmap;

}

if (list\_empty(&mm->mmlist)) {

spin\_lock(&mmlist\_lock);

if (list\_empty(&mm->mmlist))

list\_add(&mm->mmlist, &init\_mm.mmlist);

spin\_unlock(&mmlist\_lock);

}

dec\_mm\_counter(mm, MM\_ANONPAGES);

inc\_mm\_counter(mm, MM\_SWAPENTS);

} else if (IS\_ENABLED(CONFIG\_MIGRATION)) {

BUG\_ON(!(flags & TTU\_MIGRATION));

entry = make\_migration\_entry(page, pte\_write(pteval));

}

**swp\_pte = swp\_entry\_to\_pte(entry)**; /\*swp\_entry\_t转pte\_t内存页表项\*/

if (pte\_soft\_dirty(pteval))

swp\_pte = pte\_swp\_mksoft\_dirty(swp\_pte);

**set\_pte\_at(mm, address, pte, swp\_pte);** /\*pte\_t写入内存页表项\*/

} else if (IS\_ENABLED(CONFIG\_MIGRATION) &&(flags & TTU\_MIGRATION)) {

swp\_entry\_t entry;

entry = make\_migration\_entry(page, pte\_write(pteval));

set\_pte\_at(mm, address, pte, swp\_entry\_to\_pte(entry));

} else

dec\_mm\_counter(mm, MM\_FILEPAGES); /\*文件映射页，减小地址空间页统计值\*/

**page\_remove\_rmap(page)**; /\*将页从反向映射结构中移除，映射计数减1等，/mm/rmap.c\*/

**page\_cache\_release(page)**; /\*释放页，引用计数值\_count减1\*/

out\_unmap:

pte\_unmap\_unlock(pte, ptl);

if (ret != SWAP\_FAIL && !(flags & TTU\_MUNLOCK))

mmu\_notifier\_invalidate\_page(mm, address);

out:

return ret; /\*函数返回\*/

out\_mlock:

pte\_unmap\_unlock(pte, ptl);

if (down\_read\_trylock(&vma->vm\_mm->mmap\_sem)) {

if (vma->vm\_flags & VM\_LOCKED) {

**mlock\_vma\_page(page)**; /\*标记页锁定，放回不活跃链表，/mm/mlock.c\*/

ret = SWAP\_MLOCK;

}

up\_read(&vma->vm\_mm->mmap\_sem);

}

return ret;

}

try\_to\_unmap\_one()函数中解除页到VMA的映射，涉及到VMA对页的锁定，页回收和页交换机制等。读者可以在学习完后面的内容后再来详细研究此函数。

现在仅对try\_to\_unmap\_one()函数做简要的解释：函数根据映射页虚拟地址address找到对应的内存页表项。如果VMA设置了VM\_LOCKED标记位（锁定映射页），则不解除页映射。若要解除映射，则对内存页表项清零，并将原页表项内容写入pteval局部变量。

对于正在使用的匿名映射页，解除映射前已将其数据写出到交换区，交换区位置保存在page->private成员中（swp\_entry\_t结构体实例），将page->private转换成内存页表项格式写入到映射页的页表项中。在下次再次访问该页时，由缺页异常处理程序根据页表项中保存的交换区信息，从交换区恢复数据。

对于文件映射页的处理要简单一些，如果页脏，则在set\_page\_dirty()函数中会调用文件地址空间中的函数标记页脏，由页缓存机制将页数据回写到文件中。在page\_cache\_release()函数中会释放页，当\_count值减为0时，将释放页帧。

## 4.4地址空间操作

前面介绍了进程地址空间布局和管理数据结构，以及反向映射结构。那么，进程初始的地址空间、虚拟内存域实例从何而来呢？

内核自身地址空间mm\_struct实例init\_mm定义在/mm/init-mm.c文件内，内核自身可看成系统内的第一个进程（内核线程），其它进程（线程）通过复制而来。由于内核（内核线程）不使用用户地址空间，因此，在init\_mm实例中并没有表示用户进程地址空间的信息。在运行第一个用户进程，加载可执行目标文件时，将创建进程地址空间。

由进程创建进程时，子进程将复制父进程地址空间信息，子进程在执行execve()系统调用，运行新的可执行目标文件时，将重新创建和设置进程地址空间，与创建第一个用户进程时相同。

本节介绍用户进程地址空间的创建，以及复制进程时复制地址空间的操作。

### 4.4.1创建地址空间

在运行第一个用户进程或进程通过execve()等系统调用加载新程序运行时，内核将为进程创建新地址空间实例，函数调用关系简列如下图所示（大部分函数在/fs/exec.c文件内实现）：



以上主要函数功能简介如下（详情请参考第5章）：

●**bprm\_mm\_init()**：创建一个新的mm\_struct实例并初始化（创建全局页表等），创建进程栈对应的VMA实例，并插入到mm\_struct管理结构中。

●flush\_old\_exec()：释放在复制进程时创建的mm\_struct实例，并将bprm\_mm\_init()函数内创建的新实例赋予进程。如果是execve()系统调用，在创建子进程时，子进程已经复制了父进程的mm\_struct实例，因此在此处需要将其释放。

●**arch\_pick\_mmap\_layout()：**用于定位内存映射区的起始地址，设置mm\_struct实例get\_unmapped\_area函数指针成员等，这是一个体系结构相关的函数。

●**elf\_map()**：将可执行目标文件中的加载段，如代码/数据段，以文件映射的形式映射到进程地址空间中的代码/数据区，函数内调用vm\_mmap()函数完成文件映射操作。

vm\_mmap()函数定义在/mm/util.c文件内：

unsigned long vm\_mmap(struct file \*file, unsigned long addr,unsigned long len, unsigned long prot, \

unsigned long flag, unsigned long offset)

{

if (unlikely(offset + PAGE\_ALIGN(len) < offset))

return -EINVAL;

if (unlikely(offset & ~PAGE\_MASK))

return -EINVAL;

return **vm\_mmap\_pgoff(file, addr, len, prot, flag, offset >> PAGE\_SHIFT)**;

}

vm\_mmap\_pgoff()函数功能是为进程地址空间创建文件映射，在后面讲解内存映射时再介绍此函数的实现。

●set\_brk(elf\_bss, elf\_brk)：在[elf\_bss,elf\_brk)区域创建匿名映射，并将堆起始结束地址都设为elf\_brk。这个函数用于为可执行文件未初始化数据段（.bss段）创建匿名映射。到第5章再介绍此函数的实现。

下面将分别介绍bprm\_mm\_init()和arch\_pick\_mmap\_layout()函数的实现。

#### 1创建mm\_struct实例

bprm\_mm\_init(bprm)函数用于为新进程创建地址空间mm\_struct实例，并初始化，函数定义在/fs/exec.c文件内，代码如下：

static int bprm\_mm\_init(struct linux\_binprm \*bprm)

{

int err;

struct mm\_struct \*mm = NULL;

bprm->mm = mm = **mm\_alloc()**; /\*创建mm\_struct实例并初始化，/kernel/fork.c\*/

err = -ENOMEM;

if (!mm)

goto err;

err = **\_\_bprm\_mm\_init(bprm)**; /\*创建并初始化进程栈对应的vm\_area\_struct实例，/fs/exec.c\*/

if (err)

goto err;

return 0;

...

}

bprm\_mm\_init(bprm)函数调用mm\_alloc()函数为进程创建并初始化mm\_struct实例，\_\_bprm\_mm\_init()函数用于初始化进程栈。

mm\_alloc()函数定义在/kernel/fork.c文件内，完成mm\_struct实例的创建和初始化，代码如下：

struct mm\_struct \*mm\_alloc(void)

{

struct mm\_struct \*mm;

mm = allocate\_mm(); /\*从slab缓存中分配mm\_struct实例，/kernel/fork.c\*/

if (!mm)

return NULL;

memset(mm, 0, sizeof(\*mm)); /\*mm\_struct实例清零\*/

return **mm\_init(mm, current)**; /\*初始化mm\_struct实例，包括创建PGD页表等，/kernel/fork.c\*/

}

mm\_alloc()函数从slab缓存中分配mm\_struct实例，然后调用mm\_init()函数初始化实例。

mm\_init()函数在/kernel/fork.c文件内实现，主要完成mm\_struct实例各成员的初始化，包括分配PGD全局页表（复制内核页表中内核地址空间对应的PGD表项至进程页表），从父进程mm\_struct实例中继承flags和def\_flags成员值等，源代码请读者自行阅读。

\_\_bprm\_mm\_init(bprm)函数在/fs/exec.c文件内实现，主要用于为进程栈创建vm\_area\_struct实例，函数代码如下：

static int \_\_bprm\_mm\_init(struct linux\_binprm \*bprm)

{

int err;

struct vm\_area\_struct \*vma = NULL;

struct mm\_struct \*mm = bprm->mm; /\*新创建的mm\_struct实例\*/

**bprm->vma = vma = kmem\_cache\_zalloc(vm\_area\_cachep, GFP\_KERNEL)**;

/\*分配vm\_area\_struct实例\*/

if (!vma)

return -ENOMEM;

down\_write(&mm->mmap\_sem);

vma->vm\_mm = mm; /\*指向mm\_struct实例\*/

BUILD\_BUG\_ON(VM\_STACK\_FLAGS & VM\_STACK\_INCOMPLETE\_SETUP);

vma->vm\_end = STACK\_TOP\_MAX; /\*栈顶位置\*/

vma->vm\_start = vma->vm\_end - PAGE\_SIZE; /\*栈大小初始值设为一页，运行时会扩展\*/

vma->vm\_flags = VM\_SOFTDIRTY | VM\_STACK\_FLAGS | VM\_STACK\_INCOMPLETE\_SETUP;

vma->vm\_page\_prot = vm\_get\_page\_prot(vma->vm\_flags);

INIT\_LIST\_HEAD(&vma->anon\_vma\_chain); /\*初始化双链表\*/

err = **insert\_vm\_struct**(mm, vma); /\*将vma添加到地址空间管理结构中，/mm/mmap.c\*/

if (err)

goto err;

mm->stack\_vm = mm->total\_vm = 1;

arch\_bprm\_mm\_init(mm, vma); /\*空操作，/include/asm-generic/mm\_hooks.h\*/

up\_write(&mm->mmap\_sem);

**bprm->p = vma->vm\_end - sizeof(void \*)**; /\*设置用户栈顶地址\*/

return 0;

...

}

\_\_bprm\_mm\_init(bprm)函数为进程栈创建了vm\_area\_struct实例，并对其进行初始化，该虚拟内存域对应的虚拟地址空间是用户栈顶部的一页，在进程运行过程中将扩展栈。

注意，bprm\_mm\_init()函数只是为进程准备好了mm\_struct实例，而并没有启用（没有赋予task\_struct实例）。在加载可执行目标文件时调用的flush\_old\_exec()函数中将新mm\_struct实例赋予进程，并释放进程原mm\_struct实例（复制于父进程的mm\_struct实例）。

#### 2定位内存映射区

arch\_pick\_mmap\_layout()函数用于定位内存映射区的起始地址，设置mm\_struct实例get\_unmapped\_area函数指针成员等，这是一个体系结构相关的函数。

MIPS32体系结构此函数定义在/arch/mips/mm/mmap.c文件内：

void arch\_pick\_mmap\_layout(struct mm\_struct \*mm)

{

unsigned long random\_factor = 0UL;

if (current->flags & PF\_RANDOMIZE) /\*内存映射区起始地址加上随机偏移量\*/

random\_factor = **arch\_mmap\_rnd()**; /\*/arch/mips/mm/mmap.c\*/

if (mmap\_is\_legacy()) { /\*/arch/mips/mm/mmap.c\*/

mm->mmap\_base = **TASK\_UNMAPPED\_BASE + random\_factor**; /\*内存映射区基地址\*/

mm->get\_unmapped\_area = **arch\_get\_unmapped\_area**; /\*获取未映射区域函数指针\*/

} else {

mm->mmap\_base = mmap\_base(random\_factor);

mm->get\_unmapped\_area = arch\_get\_unmapped\_area\_topdown;

}

}

mm\_struct实例中的get\_unmapped\_area()函数指针成员赋值为**arch\_get\_unmapped\_area()**函数指针，这是一个体系结构相关的函数，MPIS32体系结构定义/arch/mips/mm/mmap.c文件内，下一节将介绍此函数的实现。

以上介绍的是创建地址空间的操作，在创建子进程时，将复制父进程地址空间至子进程，下一小节将介绍复制操作的实现。

### 4.4.2复制地址空间

内核在创建进程/线程时（详见第5章），在复制进程的copy\_process()函数内调用copy\_mm()函数复制父进程地址空间至子进程，函数定义在/kernel/fork.c文件内：

static int copy\_mm(unsigned long clone\_flags, struct task\_struct \*tsk)

/\*clone\_flags：复制进程标记，tsk：子进程数据结构指针\*/

{

struct mm\_struct \*mm, \*oldmm;

int retval;

tsk->min\_flt = tsk->maj\_flt = 0;

tsk->nvcsw = tsk->nivcsw = 0;

#ifdef CONFIG\_DETECT\_HUNG\_TASK

tsk->last\_switch\_count = tsk->nvcsw + tsk->nivcsw;

#endif

**tsk->mm = NULL**;

**tsk->active\_mm = NULL**;

oldmm = **current->mm**; /\*当前进程（父进程）地址空间实例指针，内核线程mm为空\*/

if (!oldmm) /\*如果当前进程为内核线程，则直接返回，不用复制（没有用户地址空间）\*/

return 0;

vmacache\_flush(tsk); /\*清零tsk->vmacache成员，/include/linux/vmacache.h\*/

if (clone\_flags & **CLONE\_VM**) { /\*创建线程，父子进程共用地址空间，不用复制\*/

atomic\_inc(&oldmm->mm\_users); /\*增加地址空间用户数\*/

mm = oldmm;

goto good\_mm;

}

/\*由进程创建进程，需要复制地址空间\*/

retval = -ENOMEM;

mm = **dup\_mm(tsk)**; /\*复制用户进程地址空间，/kernel/fork.c\*/

if (!mm)

goto fail\_nomem;

good\_mm:

**tsk->mm = mm**; /\*设置子进程地址空间\*/

**tsk->active\_mm = mm**;

return 0;

...

}

由copy\_mm()函数可知，只有进程创建进程时才需要复制地址空间，此时调用dup\_mm()函数为子进程创建和复制地址空间实例。dup\_mm()函数定义如下（/kernel/fork.c）：

static struct mm\_struct \*dup\_mm(struct task\_struct \*tsk)

{

struct mm\_struct \*mm, \***oldmm = current->mm**; /\*父进程地址空间\*/

int err;

mm = **allocate\_mm()**; /\*从slab缓存分配mm\_struct实例\*/

if (!mm)

goto fail\_nomem;

**memcpy(mm, oldmm, sizeof(\*mm));**  /\*复制父进程实例数据至子进程\*/

if (!**mm\_init(mm, tsk)**) /\*初始化实例，主要成员清零，创建全局页表并初始化等，/kernel/fork.c\*/

goto fail\_nomem;

err = **dup\_mmap(mm, oldmm);**  /\*复制VMA，/kernel/fork.c\*/

if (err)

goto free\_pt;

mm->hiwater\_rss = get\_mm\_rss(mm); /\*设置水印值\*/

mm->hiwater\_vm = mm->total\_vm;

if (mm->binfmt && !try\_module\_get(mm->binfmt->module))

goto free\_pt;

return mm; /\*返回新mm\_struct实例指针\*/

...

}

dup\_mm()函数从slab缓存中分配新mm\_struct实例，随后复制父进程mm\_struct实例数据到新实例，并对新实例进行初始化，主要内容是对部分成员清零，创建全局页表并初始化等。最后调用dup\_mmap()函数复制父进程地址空间VMA信息至子进程地址空间，并同时复制VMA对应的页表页数据。

dup\_mm()函数返回新mm\_struct实例指针。新mm\_struct实例（子进程）具有了自身的VMA实例和页表，但是与父进程VMA映射到相同的物理内存。子进程对虚拟页进行写操作时，可能需要重新分配页帧替换原有映射页帧，以实现父子进程之间的隔离，详见下文复制VMA页表的操作。

下面介绍一下复制地址空间VMA的dup\_mmap()函数的实现。

#### 1复制内存域

dup\_mmap()函数用于复制父进程VMA至子进程，函数代码如下（/kernel/fork.c）：

static int dup\_mmap(struct mm\_struct \*mm, struct mm\_struct \*oldmm)

/\*mm：子进程地址空间，oldmm：父进程地址空间\*/

{

struct vm\_area\_struct \*mpnt, \*tmp, \*prev, \*\*pprev;

struct rb\_node \*\*rb\_link, \*rb\_parent;

int retval;

unsigned long charge;

uprobe\_start\_dup\_mmap();

down\_write(&oldmm->mmap\_sem);

flush\_cache\_dup\_mm(oldmm);

uprobe\_dup\_mmap(oldmm, mm);

down\_write\_nested(&mm->mmap\_sem, SINGLE\_DEPTH\_NESTING);

RCU\_INIT\_POINTER(mm->exe\_file, get\_mm\_exe\_file(oldmm));

mm->total\_vm = oldmm->total\_vm; /\*地址空间映射页统计量\*/

mm->shared\_vm = oldmm->shared\_vm;

mm->exec\_vm = oldmm->exec\_vm;

mm->stack\_vm = oldmm->stack\_vm;

rb\_link = &mm->mm\_rb.rb\_node; /\*VMA红黑树根节点成员\*/

**rb\_parent = NULL**;

**pprev = &mm->mmap**; /\*第1个VMA\*/

retval = ksm\_fork(mm, oldmm);

if (retval)

goto out;

retval = khugepaged\_fork(mm, oldmm);

if (retval)

goto out;

prev = NULL;

for (mpnt = oldmm->mmap; mpnt; mpnt = mpnt->vm\_next) { /\*遍历地址空间所有VMA实例\*/

struct file \*file;

if (mpnt->vm\_flags & VM\_DONTCOPY) { /\*跳过不需要复制的VMA\*/

vm\_stat\_account(mm, mpnt->vm\_flags, mpnt->vm\_file,-vma\_pages(mpnt)); /\*更新统计量\*/

continue;

}

charge = 0;

if (mpnt->vm\_flags & VM\_ACCOUNT) {

unsigned long len = vma\_pages(mpnt); /\*VMA长度\*/

if (security\_vm\_enough\_memory\_mm(oldmm, len)) /\* sic \*/

goto fail\_nomem;

charge = len;

}

tmp = **kmem\_cache\_alloc(vm\_area\_cachep, GFP\_KERNEL)**; /\*创建vm\_area\_struct实例\*/

if (!tmp)

goto fail\_nomem;

**\*tmp = \*mpnt**; /\*复制vm\_area\_struct实例数据\*/

INIT\_LIST\_HEAD(&tmp->anon\_vma\_chain); /\*初始化anon\_vma\_chain双链表\*/

retval = vma\_dup\_policy(mpnt, tmp); /\*UMA系统直接返回0\*/

if (retval)

goto fail\_nomem\_policy;

**tmp->vm\_mm = mm**; /\*指向mm\_struct实例\*/

if (**anon\_vma\_fork(tmp, mpnt)**) /\*创建反向映射结构，见上文，/mm/rmap.c\*/

goto fail\_nomem\_anon\_vma\_fork;

**tmp->vm\_flags &= ~VM\_LOCKED**; /\*清零VMA的VM\_LOCKED标记位\*/

tmp->vm\_next = tmp->vm\_prev = NULL;

file = tmp->vm\_file; /\*映射文件，如果存在\*/

if (file) { /\*处理文件映射VMA\*/

struct inode \*inode = file\_inode(file);

struct address\_space \***mapping = file->f\_mapping**;

get\_file(file);

if (tmp->vm\_flags & VM\_DENYWRITE)

atomic\_dec(&inode->i\_writecount);

i\_mmap\_lock\_write(mapping);

if (tmp->vm\_flags & VM\_SHARED) /\*共享映射\*/

atomic\_inc(&mapping->i\_mmap\_writable);

flush\_dcache\_mmap\_lock(mapping);

**vma\_interval\_tree\_insert\_after(tmp, mpnt,&mapping->i\_mmap)**;

/\*VMA插入文件地址空间反向映射区间树，/mm/interval\_tree.c\*/

flush\_dcache\_mmap\_unlock(mapping);

i\_mmap\_unlock\_write(mapping);

}

if (is\_vm\_hugetlb\_page(tmp))

reset\_vma\_resv\_huge\_pages(tmp);

\*pprev = tmp; /\*VMA插入地址空间中链表\*/

pprev = &tmp->vm\_next;

tmp->vm\_prev = prev;

prev = tmp;

**\_\_vma\_link\_rb(mm, tmp, rb\_link, rb\_parent)**; /\*VMA插入地址空间中的（扩展）红黑树\*/

rb\_link = &tmp->vm\_rb.rb\_right;

rb\_parent = &tmp->vm\_rb;

mm->map\_count++;

retval = **copy\_page\_range(mm, oldmm, mpnt)**; /\*复制VMA对应页表项，/mm/memory.c\*/

if (tmp->vm\_ops && tmp->vm\_ops->open)

**tmp->vm\_ops->open(tmp)**; /\*调用文件映射操作结构中的open()操作函数\*/

if (retval)

goto out;

} /\*for循环遍历VMA结束\*/

arch\_dup\_mmap(oldmm, mm);

retval = 0;

out:

up\_write(&mm->mmap\_sem);

flush\_tlb\_mm(oldmm);

up\_write(&oldmm->mmap\_sem);

uprobe\_end\_dup\_mmap();

return retval;

...

}

dup\_mmap()函数的主要工作是：遍历父进程地址空间VMA实例，对没有设置VM\_DONTCOPY标记位的VMA逐个复制一个副本至子进程，并复制对应的页表项。复制操作首先创建新vm\_area\_struct实例，复制父VMA实例数据到新VMA，去除新VMA的VM\_LOCKED标记位；然后处理VMA反向映射数据结构，如果是匿名映射，则复制父VMA反向映射数据结构，如果是文件映射，只需将VMA插入映射文件地址空间反向映射区间树即可；随后需要将新VMA插入子进程地址空间链表和（扩展）红黑树中，最后复制VMA页表（并调用文件映射打开内存域操作函数open()）至子进程页表。

下面再详细介绍一下复制父VMA对应页表项至子进程页表的操作。

##### ■复制内存域页表

在复制父进程VMA的操作中，复制完VMA实例数据和反向映射数据结构后，需要复制父VMA对应页表项至子进程页表，以使父子进程映射到相同的物理内存。但是，在复制页表项的过程中可能需要修改子进程的访问属性标记。

复制页表函数为copy\_page\_range()，定义如下（/mm/memory.c）：

int copy\_page\_range(struct mm\_struct \*dst\_mm, struct mm\_struct \*src\_mm,struct vm\_area\_struct \*vma)

/\*dst\_mm：子进程地址空间指针，src\_mm：父进程地址空间指针，vma：父进程VMA指针\*/

{

pgd\_t \*src\_pgd, \*dst\_pgd;

unsigned long next;

unsigned long addr = vma->vm\_start; /\*VMA起始地址\*/

unsigned long end = vma->vm\_end; /\*VMA结束地址\*/

unsigned long mmun\_start; /\* For mmu\_notifiers \*/

unsigned long mmun\_end; /\* For mmu\_notifiers \*/

bool is\_cow;

int ret;

/\*VMA页表项不复制的情形\*/

if (!(vma->vm\_flags & (VM\_HUGETLB | VM\_PFNMAP | VM\_MIXEDMAP)) &&!vma->anon\_vma)

return 0;

/\*没有设置VM\_HUGETLB 、VM\_PFNMAP和VM\_MIXEDMAP中任一标记位，

\*并且是文件映射或没有建立任何映射的匿名映射VMA，不复制页表项。

\*/

if (is\_vm\_hugetlb\_page(vma))

return copy\_hugetlb\_page\_range(dst\_mm, src\_mm, vma);

if (unlikely(vma->vm\_flags & **VM\_PFNMAP**)) { /\*PFN映射\*/

ret = track\_pfn\_copy(vma); /\*返回0\*/

if (ret)

return ret;

}

is\_cow = is\_cow\_mapping(vma->vm\_flags); /\*可能写的私有映射，/mm/internal.h\*/

/\*是否是写时复制VMA，VM\_MAYWRITE置位，且VM\_SHARED为零。\*/

mmun\_start = addr;

mmun\_end = end;

if (is\_cow)

mmu\_notifier\_invalidate\_range\_start(src\_mm, mmun\_start,mmun\_end);

ret = 0;

dst\_pgd = pgd\_offset(dst\_mm, addr); /\*PGD页表项指针\*/

src\_pgd = pgd\_offset(src\_mm, addr);

do {

next = pgd\_addr\_end(addr, end);

if (**pgd\_none\_or\_clear\_bad(src\_pgd)**)

continue;

if (unlikely(**copy\_pud\_range(dst\_mm, src\_mm, dst\_pgd, src\_pgd,vma, addr, next)**)) {

ret = -ENOMEM;

break;

}

} while (dst\_pgd++, src\_pgd++, addr = next, addr != end);

if (is\_cow)

mmu\_notifier\_invalidate\_range\_end(src\_mm, mmun\_start, mmun\_end);

return ret;

}

在两级页表模型中copy\_pud\_range()函数最终调用copy\_pte\_range()函数为子进程创建PTE页表，并调用copy\_one\_pte()函数逐个复制PTE页表项至子进程PTE页表，函数定义如下（/mm/memory.c）：

static inline unsigned long copy\_one\_pte(struct mm\_struct \*dst\_mm, struct mm\_struct \*src\_mm, \

pte\_t \***dst\_pte**, pte\_t \***src\_pte**, struct vm\_area\_struct \*vma,unsigned long addr, int \*rss)

/\*dst\_pte：目标PTE页表项指针，src\_pte：源PTE页表项指针\*/

{

unsigned long vm\_flags = **vma->vm\_flags**; /\*父VMA标记\*/

**pte\_t pte = \*src\_pte**;

struct page \*page;

/\*PTE页表项中包含交换区位置信息，复制页表项\*/

if (unlikely(!pte\_present(pte))) { /\*P标记位为0，检查是否包含交换区信息\*/

swp\_entry\_t entry = pte\_to\_swp\_entry(pte); /\*/include/linux/swapops.h\*/

/\*PTE页表项转换成SWP页表项\*/

if (likely(!non\_swap\_entry(entry))) {

if (**swap\_duplicate(entry) < 0**) /\*swap\_map[]对应数组项引用计数值加1\*/

return entry.val; /\*如果没有对应数组项，返回负值，表示页数据不在交换区\*/

/\*映射页数据在交换区\*/

if (unlikely(list\_empty(&dst\_mm->mmlist))) {

spin\_lock(&mmlist\_lock);

if (list\_empty(&dst\_mm->mmlist))

list\_add(&dst\_mm->mmlist,&src\_mm->mmlist);

spin\_unlock(&mmlist\_lock);

}

rss[MM\_SWAPENTS]++;

} else if (is\_migration\_entry(entry)) { /\*迁移页表项\*/

page = migration\_entry\_to\_page(entry);

if (PageAnon(page))

rss[MM\_ANONPAGES]++;

else

rss[MM\_FILEPAGES]++;

if (is\_write\_migration\_entry(entry) &&is\_cow\_mapping(vm\_flags)) {

make\_migration\_entry\_read(&entry);

pte = swp\_entry\_to\_pte(entry);

if (pte\_swp\_soft\_dirty(\*src\_pte))

pte = pte\_swp\_mksoft\_dirty(pte);

set\_pte\_at(src\_mm, addr, src\_pte, pte);

}

} /\*if (likely(!non\_swap\_entry(entry)))结束\*/

goto **out\_set\_pte**; /\*跳至设置页表项处\*/

} /\*if (unlikely(!pte\_present(pte)))结束\*/

/\*P标记位为1的页表项，映射到物理页帧\*/

if (**is\_cow\_mapping(vm\_flags)**) {

/\*写时复制VMA，设置了VM\_MAYWRITE标记，但没有设置VM\_SHARED标记\*/

**ptep\_set\_wrprotect(src\_mm, addr, src\_pte)**;

/\*设置源页表项写保护，/include/asm-generic/pgtable.h\*/

**pte = pte\_wrprotect(pte)**; /\*设置目标页表项写保护，清零W和D标记位\*/

}

if (vm\_flags & VM\_SHARED) /\*共享VMA\*/

**pte = pte\_mkclean(pte)**; /\*清除目标页表项M，D标记位，/arch/mips/include/asm/pgtable.h\*/

pte = pte\_mkold(pte); /\*清零目标页表项A，V标记位，设置最近没有被访问且无效\*/

page = vm\_normal\_page(vma, addr, pte); /\*是否是普通页帧（由page实例管理），返回page实例\*/

if (page) {

get\_page(page); /\*增加page实例**\_count**成员计数值\*/

**page\_dup\_rmap(page)**; /\*增加page实例\_mapcount成员计数值（映射计数）\*/

if (PageAnon(page))

rss[MM\_ANONPAGES]++; /\*更新匿名映射页统计量\*/

else

rss[MM\_FILEPAGES]++; /\*更新文件缓存页统计量\*/

}

out\_set\_pte: /\*设置页表项\*/

**set\_pte\_at(dst\_mm, addr, dst\_pte, pte)**; /\*设置子进程页表项\*/

return 0;

}

copy\_one\_pte()函数首先判断父页表项P标记位是否为0，如果是则检查是否含有交换区信息，即是否是交换出去的匿名映射页，如果是则增加交换区的引用计数，直接将父页表项内容复制到子进程页表即可。

如果父（源）页表项P标记位为1，表示映射到了物理页帧，则需要检查VMA标记，根据需要修改父、子页表项访问属性标记，增加page实例的引用和映射计数值，最后将修改后页表项数据写入子进程页表。

复制操作后的父子进程页表简略如下图所示：



## 4.5 VMA操作

在创建第一个用户进程，或进程调用execve()系统调用运行新程序时，内核将为进程创建初始的地址空间，如下图所示：



进程地址空间的代码/数据区是文件映射，映射到可执行目标文件。堆起始只包含未初始化数据段，堆向上生长。内存映射区设置了起始地址，如果程序使用了动态库，在加载程序时还会将动态库映射到内存映射区。栈位于地址空间顶部，栈在进程运行过程中向下生长。

代码/数据区的映射在进程运行过程中不会改变。堆用于为进程动态分配（小块）内存，用户空间库函数malloc()管理着堆，库函数将堆区域划分成小块，为进程动态分配内存，这类似于内核空间的slab分配器。当堆区域空闲内存不足时，malloc()函数通过系统调用brk()系统调用扩展堆（也可收缩堆）。

用户进程通过系统调用可以在内存映射区创建匿名映射或文件映射。栈在进程运行过程中由内核自动扩展。

用户创建/解除映射的系统调用都涉及对VMA的操作，主要包括在地址空间中申请空闲区域用于创建VMA，VMA的查找、添加、拆分、合并等。本节介绍VMA的相关操作，为下节介绍系统调用的实现打下基础。

### 4.5.1查找VMA

查找内存域find\_vma(struct mm\_struct \*mm, unsigned long addr)函数，用于在地址空间mm内查找第一个结束地址在addr之后的VMA实例，成功返回VMA指针，否则返回NULL。如下图所示，对虚拟地址addr1和addr2调用find\_vma()函数，返回的都是VMA2实例指针，对addr3调用查找函数将返回NULL，因为其后没有VMA实例了。



查找函数find\_vma()定义在/mm/mmap.c文件内：

struct vm\_area\_struct \*find\_vma(struct mm\_struct \*mm, unsigned long addr)

{

struct rb\_node \*rb\_node;

struct vm\_area\_struct \*vma;

vma = vmacache\_find(mm, addr);

/\*在地址空间vm\_area\_struct缓存中查找是否有包含addr地址的实例，/mm/vmacache.c\*/

if (likely(vma))

return vma;

**rb\_node = mm->mm\_rb.rb\_node**; /\*VMA（扩展）红黑树根节点\*/

vma = NULL; /\*保存返回值\*/

while (rb\_node) { /\*在红黑树中查找\*/

struct vm\_area\_struct \*tmp;

tmp = rb\_entry(rb\_node, struct vm\_area\_struct, vm\_rb);

if (tmp->vm\_end > addr) {

vma = tmp;

if (tmp->vm\_start <= addr)

break;

rb\_node = rb\_node->rb\_left;

} else

rb\_node = rb\_node->rb\_right;

}

if (vma)

**vmacache\_update(addr, vma)**; /\*VMA更新到VMA缓存，/mm/vmacache.c\*/

return vma;

}

VMA查找函数代码比较简单，函数在地址空间VMA缓存中查找是否具有包含addr地址的VMA实例，有则返回VMA实例，没有则在红黑树中查找。在红黑树中从根节点开始，比较addr与节点VMA结束地址值，直至查找到结束地址在addr之后的第一个VMA，如果没有符合条件的VMA，则返回NULL。

内核基于find\_vma()函数还定义了其它的查找函数：

●struct vm\_area\_struct \***find\_vma\_intersection**(struct mm\_struct \* mm, unsigned long start\_addr, unsigned long end\_addr)：（/include/linux/mm.h）返回第一个与[start\_addr,end\_addr]地址范围有重叠的vm\_area\_struct实例指针。

●struct vm\_area\_struct \***find\_vma\_prev**(struct mm\_struct \*mm, unsigned long addr,struct vm\_area\_struct \*\*pprev)：（/mm/mmap.c）返回find\_vma(mm, addr)函数查找的内存域实例指针，\*pprev指向find\_vma()查找的VMA的前一个VMA。若find\_vma()返回NULL，则\*pprev指向地址空间中最后一个VMA实例（红黑树中最右边节点）。

●struct vm\_area\_struct \***find\_extend\_vma**(struct mm\_struct \*mm, unsigned long addr)：返回包含addr的VMA实例（/mm/mmap.c）。如果addr处于空洞区则判断其后内存域是否为栈，是则扩展栈后返回栈VMA实例，如果其后VMA不是栈，则返回NULL，此函数实际用于扩展栈。

### 4.5.2获取未映射区域

在进程地址空间创建映射区域，首先需要申请分配一段进程尚未使用的地址段，已使用的地址段都有对应的vm\_area\_struct实例。申请成功空闲的地址段后，再依此创建并设置vm\_area\_struct实例，并将其插入到地址空间管理结构和反向映射结构。

获取未映射区域的函数get\_unmapped\_area()定义在/mm/mmap.c文件内，若获取/申请未映射区域成功，则返回未映射区域起始虚拟地址，否则返回错误码：

unsigned long get\_unmapped\_area(struct file \*file, unsigned long addr, unsigned long len, \

unsigned long pgoff, unsigned long flags)

/\*

\*file：文件映射，指向映射文件file实例，匿名映射为NULL；

\*addr：指定未映射区域起始虚拟地址，为0表示由内核确定起始地址；

\*len：未映射区域大小，字节数；

\*pgoff：VMA页偏移量，文件映射为文件内容页偏移量，匿名映射为起始VFN；

\*flags：映射标记，定义在/arch/mips/include/uapi/asm/mman.h头文件，例如：

\* #define MAP\_SHARED 0x001 /\*共享映射\*/

\* #define MAP\_PRIVATE 0x002 /\*进程私有映射\*/

\* #define MAP\_TYPE 0x00f /\*映射类型掩码\*/

\* #define MAP\_FIXED 0x010 /\*固定地址映射，参数指定起始地址，不能由内核确定\*/

\* ...

\*/

{

unsigned long (\*get\_area)(struct file \*, unsigned long,unsigned long, unsigned long, unsigned long);

/\*函数指针\*/

unsigned long error = arch\_mmap\_check(addr, len, flags); /\*直接返回0，/mm/mmap.c\*/

if (error)

return error;

if (len > TASK\_SIZE)

return -ENOMEM;

**get\_area = current->mm->get\_unmapped\_area;**  /\*mm\_struct->get\_unmapped\_area函数指针\*/

if (file && file->f\_op->get\_unmapped\_area) /\*赋值文件操作get\_unmapped\_area()函数\*/

get\_area = file->f\_op->get\_unmapped\_area;

**addr = get\_area(file, addr, len, pgoff, flags);**  /\*调用get\_area()函数\*/

... /\*判断地址有效性\*/

addr = arch\_rebalance\_pgtables(addr, len); /\*返回addr，/mm//mmap.c\*/

error = security\_mmap\_addr(addr);

return error ? error : addr;

}

get\_unmapped\_area()函数首先获取当前进程地址空间实例中定义的get\_unmapped\_area()函数指针，赋予get\_area变量；再判断是否为文件映射，如果是则判断文件操作结构中是否定义了get\_unmapped\_area()函数，如果定义了，则将其赋予get\_area变量，最后，调用get\_area()函数，返回获取未映射区域起始虚拟地址。

内核在运行新进程的execve()系统调用中调用arch\_pick\_mmap\_layout()函数（/arch/mips/mm/mmap.c）为mm\_struct实例get\_unmapped\_area函数指针成员赋值，赋值**arch\_get\_unmapped\_area()**函数指针（见见上文）。

若体系结构相关代码中没有定义HAVE\_ARCH\_UNMAPPED\_AREA宏，则arch\_get\_unmapped\_area()函数在/mm/mmap.c文件内实现，否则由体系结构相关代码实现。

MIPS32体系结构在/arch/mips/include/asm/pgtable.h头文件内定义了该宏：

#define HAVE\_ARCH\_UNMAPPED\_AREA

#define HAVE\_ARCH\_UNMAPPED\_AREA\_TOPDOWN

因此，MPIS32体系结构在/arch/mips/mm/mmap.c文件内定义了arch\_get\_unmapped\_area()函数，代码如下：

unsigned long arch\_get\_unmapped\_area(struct file \*filp, unsigned long addr0, \

unsigned long len, unsigned long pgoff, unsigned long flags)

{

return arch\_get\_unmapped\_area\_common(filp,addr0, len, pgoff, flags, **UP**); /\*映射区向上生长\*/

}

arch\_get\_unmapped\_area()函数内调用arch\_get\_unmapped\_area\_common()函数完成具体的操作。在讨论函数代码前先来看一下vm\_unmapped\_area\_info结构体的定义（/include/linux/mm.h），它用于在函数内部传递未映射区域的信息：

struct vm\_unmapped\_area\_info {

#define VM\_UNMAPPED\_AREA\_TOPDOWN 1

unsigned long flags; /\*标记，标记内存域向上还是向下生长\*/

unsigned long length; /\*未映射区域长度\*/

unsigned long low\_limit; /\*未映射区域最低地址限制\*/

unsigned long high\_limit; /\*未映射区域最高地址限制\*/

unsigned long align\_mask;

unsigned long align\_offset; /\*\*/

};

arch\_get\_unmapped\_area\_common()函数代码如下（/arch/mips/mm/mmap.c）：

static unsigned long arch\_get\_unmapped\_area\_common(struct file \*filp,unsigned long addr0, \

unsigned long len, unsigned long pgoff,unsigned long flags, enum **mmap\_allocation\_direction** dir)

{

/\*枚举类型mmap\_allocation\_direction：enum mmap\_allocation\_direction {**UP**, DOWN};\*/

struct mm\_struct \*mm = current->mm; /\*指向当前进程mm\_struct实例\*/

struct vm\_area\_struct \*vma;

unsigned long addr = addr0;

int do\_color\_align;

struct vm\_unmapped\_area\_info info; /\*/include/linux/mm.h\*/

if (unlikely(len > TASK\_SIZE)) /\*未映射区域大小超过进程地址空间大小，返回错误码\*/

return -ENOMEM;

if (flags & **MAP\_FIXED**) { /\*设置了固定地址映射标记\*/

if (TASK\_SIZE - len < addr) /\*指定地址往上没有足够大空闲地址空间\*/

return -EINVAL;

if ((flags & MAP\_SHARED) && ((addr - (pgoff << PAGE\_SHIFT)) & shm\_align\_mask))

return -EINVAL;

/shm\_align\_mask定义在/arch/mips/mm/mmap.c，PAGE\_SIZE - 1\*/

/\*共享固定地址映射不接受未页对齐的起始地址\*/

return addr;

/\*设置了MAP\_FIXED且地址合法（未检测是否与现有VMA重叠），直接返回addr\*/

}

/\*未设置MAP\_FIXED标记位\*/

do\_color\_align = 0;

if (filp || (flags & MAP\_SHARED)) /\*文件映射或共享映射\*/

do\_color\_align = 1;

if (addr) { /\*指定了未映射区起始地址\*/

if (do\_color\_align) /\*文件映射或共享映射\*/

addr = COLOUR\_ALIGN(addr, pgoff);

/\*addr页对齐后加pgoff页偏移量，/arch/mips/mm/mmap.c\*/

else /\*匿名映射或私有映射\*/

addr = PAGE\_ALIGN(addr); /\*addr已经页对齐返回addr，否则返回下一页起始地址\*/

**vma = find\_vma(mm, addr)**; /\*查找结束地址在addr之后的第一个内存域，见上文\*/

if (TASK\_SIZE - len >= addr && (!vma || addr + len <= vma->vm\_start)) /\*地址无重叠\*/

**return addr**; /\*未映射区与现有vma无重叠，或未映射区位于地址空间末尾\*/

}

/\*未设置MAP\_FIXED标记位，且未指定未映射区域起始地址，

\*或指定地址与现有VMA有重叠，则由内核确定映射地址。\*/

info.length = len;

info.align\_mask = do\_color\_align ? (PAGE\_MASK & shm\_align\_mask) : 0;

info.align\_offset = **pgoff << PAGE\_SHIFT**;

if (dir == DOWN) { /\*如果未映射区向下生长\*/

info.flags = VM\_UNMAPPED\_AREA\_TOPDOWN;

info.low\_limit = PAGE\_SIZE;

info.high\_limit = mm->mmap\_base;

addr = vm\_unmapped\_area(&info);

if (!(addr & ~PAGE\_MASK))

return addr;

}

/\*未映射区向上生长，MIPS架构是这样\*/

info.flags = 0;

**info.low\_limit = mm->mmap\_base**; /\*内存映射区基地址\*/

info.high\_limit = TASK\_SIZE;

return **vm\_unmapped\_area(&info)**; /\*/include/linux/mm.h\*/

}

arch\_get\_unmapped\_area\_common()函数首先判断参数flags是否设置了MAP\_FIXED标记，如果是且指定地址addr0合法，则返回addr0，注意这里并没有判断未映射区是否与现有内存域有重叠；然后在没有设置MAP\_FIXED标记时，判断是否指定了未映射区起始地址（addr0!=0，且页对齐），如果指定了起始地址，则检查指定未映射区是否为空闲（不与现有内存域重叠），是则表示可以在addr0处创建新VMA，函数返回addr0（页对齐）；如果在指定起始地址创建映射区会与现有VMA有重叠或者函数未指定起始地址（addr0=0），则调用vm\_unmapped\_area() 函数，由内核确定未映射区的位置，返回未映射区起始地址。

vm\_unmapped\_area()函数定义在/include/linux/mm.h头文件：

static inline unsigned long vm\_unmapped\_area(struct vm\_unmapped\_area\_info \*info)

{

if (info->flags & VM\_UNMAPPED\_AREA\_TOPDOWN)

return unmapped\_area\_topdown(info); /\*映射区向下生长\*/

else

return **unmapped\_area(info)**; /\*在VMA红黑树中查找空洞区，/mm/mmap.c\*/

}

vm\_unmapped\_area()函数info参数vm\_unmapped\_area\_info结构体中指定了查找未映射区域的起始地址（mm->mmap\_base）及标记等信息，如果从指定起始地址往上搜索未映射区域则调用unmapped\_area(info)，如果是从指定地址往下搜索未映射区域则调用unmapped\_area\_topdown(info)函数。

MIPS体系结构中内存映射区向上生长，因此调用的是unmapped\_area(info)函数。此函数内在地址空间VMA红黑树中查找合适的空洞区域，查找过程中利用了前面介绍的rb\_subtree\_gap成员值的信息（最大空洞区长度），函数返回查找到空洞区的起始虚拟地址，源代码请读者自行阅读。

获取未映射区域时，如果必须在指定起始地址处创建内存域，则需设置MAP\_FIXED标记位和指定起始地址；如果没有设置MAP\_FIXED标记位，但指定了起始地址，内核首先判断在指定起始地址处能否创建新内存域，能则直接返回起始地址，如果不能则由内核在内存映射区选择适当的位置；如果既没有设置MAP\_FIXED标记位也没有指定起始地址，则直接由内核在内存映射区选择适当的位置用于创建新内存域。

### 4.5.3插入/移出VMA

插入内存域是指将新创建的vm\_area\_struct实例插入到进程地址空间mm\_struct的管理结构中（含文件映射反向结构）。插入内存域函数insert\_vm\_struct()定义在/mm/mmap.c文件内，操作成功返回0，否则返回错误码。

int insert\_vm\_struct(struct mm\_struct \*mm, struct vm\_area\_struct \*vma)

{

struct vm\_area\_struct \*prev;

struct rb\_node \*\*rb\_link, \*rb\_parent;

if (!vma->vm\_file) { /\*匿名映射内存域，vm\_pgoff设为VMA起始虚拟页帧号\*/

BUG\_ON(vma->anon\_vma);

**vma->vm\_pgoff = vma->vm\_start >> PAGE\_SHIFT**;

}

if (**find\_vma\_links(mm, vma->vm\_start, vma->vm\_end, &prev, &rb\_link, &rb\_parent)**)

/\*确定插入位置，如果VMA被已有VMA包含，返回错误码，/mm/mmap.c\*/

return -ENOMEM;

if ((vma->vm\_flags & VM\_ACCOUNT) && \

security\_vm\_enough\_memory\_mm(mm, vma\_pages(vma)))

return -ENOMEM;

**vma\_link(mm, vma, prev, rb\_link, rb\_parent)**; /\*VMA实例插入链表和红黑树，/mm/mmap.c\*/

return 0;

}

insert\_vm\_struct()函数首先判断VMA是否为匿名映射，如果是则vm\_area\_struct实例vm\_pgoff成员赋值VMA起始虚拟页帧号。然后，调用find\_vma\_links()函数查找VMA在地址空间（扩展）红黑树中的插入位置（关联父节点），新插入VMA不能与现有VMA有重叠，否则将返回错误码。最后，调用vma\_link()函数将VMA实例插入到地址空间管理链表和扩展红黑树中。如果是文件映射，还需要将VMA插入到文件地址空间管理的反向映射结构中，匿名映射在第一次建立映射时创建反向映射结构。

insert\_vm\_struct()函数调用关系如下图所示：



插入VMA函数首行要找到VMA插入位置（红黑树和链表中位置）。在向地址空间红黑树中插入节点时，插入的位置都是叶节点。例如，如下图所示，假设进程地址空间中现有4个VMA，VMA0至VMA3，地址空间红黑树的结构如图中实线所示。现在假设要插入的内存域为VMA4，位于VMA1和VMA2之间，调用find\_vma\_links()函数查找可得VMA4的插入点为VMA1的右节点。插入函数中prev指向VMA4的前一个内存域VMA1，rb\_parent指向父节点（VMA1）rb\_node成员，rb\_link指向VMA1实例vm\_rb成员的rb\_right分量，也就是说VMA4为VMA1的右节点。

注意，前节点与父节点不一定是同一个内存域，如果插入点为右节点，则前节点和父节点相同。如果是左节点，则前节点是其祖先节点中第一个（从下往上）位于右子树的节点的父节点。例如，假设插入点是VMA3的左节点，则其父节点为VMA3，前节点为VMA2，VMA3是其祖先中第一个位于右子树的节点，而VMA2为其父节点。其实这是由红黑树的结构决定的，节点键值从左至右递增，请读者仔细推敲。



插入VMA函数随后调用vma\_link(mm, vma, prev, rb\_link, rb\_parent)函数将VMA插入到管理结构中，其中\_\_vma\_link\_list(mm, vma, prev, rb\_parent)函数用于将VMA插入到地址空间链表中，vma\_rb\_insert(vma, &mm->mm\_rb)函数用于将VMA插入到地址空间（扩展）红黑树中。\_\_vma\_link\_file(vma)函数用于将文件映射VMA插入到文件地址空间address\_space实例管理的区间树中，建立反向映射结构。各函数源代码位于/mm/mmap.c文件内，请读者自行阅读。

移出VMA的函数为\_\_vma\_unlink()，函数定义如下（/mm/mmap.c）：

static inline void \_\_vma\_unlink(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

struct vm\_area\_struct \*prev)

{

struct vm\_area\_struct \*next;

**vma\_rb\_erase(vma, &mm->mm\_rb)**; /\*从地址空间（扩展）红黑树中移出VMA\*/

prev->vm\_next = next = vma->vm\_next; /\*从地址空间链表中移出VMA\*/

if (next)

next->vm\_prev = prev;

/\*将VMA从地址空间VMA缓存中移除\*/

vmacache\_invalidate(mm);

}

\_\_vma\_unlink()函数将VMA从地址空间红黑树和链表中移出（并没有释放），但没有处理反向映射结构（仍保留）。从址空间红黑树中移出VMA的函数定义如下（/mm/mmap.c）：

static void vma\_rb\_erase(struct vm\_area\_struct \*vma, struct rb\_root \*root)

{

validate\_mm\_rb(root, vma); /\*没有选择DEBUG\_VM\_RB配置选项为空操作\*/

**rb\_erase\_augmented(&vma->vm\_rb, root, &vma\_gap\_callbacks)**; /\*从红黑树中移出节点\*/

}

### 4.5.4拆分VMA

拆分VMA是指将VMA从指定地址处将其一分为二，拆分成两个VMA。拆分VMA需要调整原VMA（缩减）并创建一个新的VMA。

拆分VMA函数split\_vma()定义如下（/mm/mmap.c）：

int split\_vma(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long addr, int new\_below)

{

if (mm->map\_count >= sysctl\_max\_map\_count)

return -ENOMEM;

return  **\_\_split\_vma**(mm, vma, addr, new\_below);

}

split\_vma()函数内直接调用\_\_split\_vma()函数拆分VMA。\_\_split\_vma()函数定义在/mm/mmap.c文件内，VMA拆分成功返回0，否则返回错误码。\_\_split\_vma()函数代码如下：

static int \_\_split\_vma(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long **addr**, int new\_below)

/\*mm：地址空间，vma：被拆分的VMA，addr：拆分地址，new\_below：新VMA在下部或上部\*/

{

struct vm\_area\_struct \*new;

int err = -ENOMEM;

if (is\_vm\_hugetlb\_page(vma) && (addr &~(huge\_page\_mask(hstate\_vma(vma)))))

return -EINVAL;

new = kmem\_cache\_alloc(vm\_area\_cachep, GFP\_KERNEL); /\*创建vm\_area\_struct实例\*/

if (!new)

goto out\_err;

**\*new = \*vma**; /\*复制原VMA实例所有数据至新VMA\*/

**INIT\_LIST\_HEAD(&new->anon\_vma\_chain)**; /\*初始化anon\_vma\_chain双链表为空\*/

if (new\_below) /\*新VMA位于原VMA下部（低地址）\*/

new->vm\_end = addr; /\*结束地址为拆分地址\*/

else { /\*新VMA位于原VMA上部\*/

new->vm\_start = addr; /\*起始地址为拆分地址\*/

new->vm\_pgoff += ((addr - vma->vm\_start) >> PAGE\_SHIFT);

}

err = vma\_dup\_policy(vma, new); /\*返回0\*/

if (err)

goto out\_free\_vma;

err = **anon\_vma\_clone(new, vma)**;

/\*复制匿名反向映射数据结构，关联相同的anon\_vma实例（**\*new = \*vma**），见4.3.3小节\*/

if (err)

goto out\_free\_mpol;

if (new->vm\_file) /\*文件映射\*/

get\_file(new->vm\_file); /\*增加file实例引用计数，/include/linux/fs.h\*/

if (new->vm\_ops && new->vm\_ops->open) /\*调用文件映射VMA打开操作函数\*/

new->vm\_ops->open(new);

if (new\_below) /\*新VMA位于下部，原VMA位于上部\*/

err = **vma\_adjust**(vma, addr, vma->vm\_end, vma->vm\_pgoff +

((addr - new->vm\_start) >> PAGE\_SHIFT), new); /\*调整原VMA，插入新VMA\*/

else /\*新VMA位于上部，原VMA位于下部\*/

err = **vma\_adjust**(vma, vma->vm\_start, addr, vma->vm\_pgoff, new);

/\*调整原VMA，插入新VMA\*/

/\*成功返回0\*/

if (!err)

return 0;

...

}

\_\_split\_vma()函数首先为新VMA创建vm\_area\_struct实例，并复制原vm\_area\_struct实例数据到新VMA实例，并设置新VMA起止地址。

如果是匿名映射VMA，则需要复制原VMA反向映射信息到新VMA。复制函数anon\_vma\_clone(new, vma)前面介绍过了，执行完此函数后新VMA与原VMA具有相同的反向映射结构，anon\_vma成员指向同一实例（因为前面的\*new = \*vma操作）。

\_\_split\_vma()函数最后调用vma\_adjust()函数调整原vm\_area\_struct实例，并将新vm\_area\_struct实例插入到管理结构中。\_\_split\_vma()函数参数new\_below如果为零，表示新VMA位于上部，地址范围是[addr，vma->vm\_end)，原VMA地址范围调整为[vma->vm\_start，addr)。如果new\_below非零，表示新VMA位于下部，地址范围是[vma->vm\_start，addr)，原VMA地址范围调整为[addr，vma->vm\_end)，如下图所示。



vma\_adjust()函数用于调整现有内存域，这里主要是对原VMA起止地址进行调整，并将新VMA添加到管理结构。vma\_adjust()函数在合并VMA的vma\_merge()函数中也将调用，后面再介绍此函数的实现。

### 4.5.5合并VMA

合并VMA用于判断指定VMA（修改访问属性后）或新申请的未映射区域（创建映射时）能否与地址空间中地址相邻的前后VMA合并。VMA合并的条件包含VMA起止地址要无缝衔接，访问属性要相同，映射类型要相同且映射的内容要连续等。映射类型要相同是指匿名映射VMA只能与匿名映射VMA合并，文件映射VMA只能与文件映射VMA合并。映射内容要连续是指：对于文件映射，两个相邻VMA映射的文件内容也必需是连续的，中间不能有文件内容的空洞；对于匿名映射就是VMA的虚拟地址要是连续的。

内核中需要执行合并VMA的时机有：一是在进程地址空间创建新映射时，判断新获取的未映射区域能否与相邻VMA合并，能合并则只需调整现有VMA，无需创建新VMA；二是在mprotect()等系统调用中，修改现有VMA（或其中一部分的）访问属性后（或映射关系后），判断被修改部分能否与相邻VMA合并，能合并则调整相邻（及被修改）VMA，不能合并则需要拆分被修改VMA。

下图示意了合并VMA的情形。

一、创建新映射时，获取的未映射区域[addr,end)与现有VMA不会有重叠。合并VMA时，判断[addr,end)能否与其前后VMA合并，能合并则执行合并，一共会有4种情况出现，如下所示：

（1）新区域能与前一内存域合并，并且合并后还能与后一内存域合并，合并成一个大的内存域。

（2）新区域能与前一内存域合并，但不能与后一内存域合并，只需调整前一内存域。

（3）新区域不能与前一内存域合并，但能与后一内存域合并，只需调整后一内存域。

（4）新区域不能与前后内存域合并，新区域需要创建新的内存域实例（下图中未画出）。



二、mprotect()等系统调用在修改[addr,end)地址范围的的访问属性时（或映射关系时），也会调用合并内存域函数。地址范围[addr,end)保证不会超过所在VMA的范围，也就是说[addr,end)位于一个VMA内部。此时也有4种情形，如下所示：

（1）[addr,end)区域正好是所在VMA的起止地址范围，区域边界与VMA边界相同，此时与创建新映射区的情形相同，判断能否与其前后VMA合并。

（2）[addr,end)区域起始地址与VMA起始地址相同，但结束地址与VMA结束地址不同（下对齐），此时只需判断[addr,end)区域修改属性后能否与前一VMA合并，能则合并，否则拆分内存域（一分为二）。

（3）[addr,end)区域结束地址与VMA结束地址相同，但起始地址与VMA起始地址不同（上对齐），此时只需判断[addr,end)区域修改属性后能否与下一VMA合并，能则合并，否则拆分VMA（一分为二）。

（4）[addr,end)区域完全位于所在VMA内部，没有触及上下边界，此时不能执行合并操作，需要拆分所在VMA，一分为三。

下面来看一下合并VMA的vma\_merge()函数的实现，函数判断[addr,end)内存区域能否与其前后的VMA合并，如果可以则合并，返回合并后的VMA指针，否则返回NULL。

vma\_merge()函数定义在/mm/mmap.c文件内：

struct vm\_area\_struct \*vma\_merge(struct mm\_struct \*mm,

struct vm\_area\_struct \*prev, unsigned long addr,unsigned long end, unsigned long vm\_flags,

struct anon\_vma \*anon\_vma, struct file \*file,pgoff\_t pgoff, struct mempolicy \*policy)

/\*

\*mm：进程地址空间实例指针；

\*prev：新内存区域[addr,end)之前的VMA（或包含[addr,end)且不是下对齐的VMA）实例指针；

\*addr,end,vm\_flags：[addr,end)内存区域起始、结束地址、内存域标记；

\*anon\_vma：[addr,end)内存区域关联anon\_vma实例指针，或为NULL（创建新映射区域或文件映射时）；

\*file：[addr,end)内存区域映射的文件，或为 NULL；

\*pgoff：[addr,end)内存区域页偏移量，文件映射为文件内容页偏移量，匿名映射为虚拟页帧号；

\*policy：只在NUMA系统上需要，这里忽略。

\*/

{

pgoff\_t pglen = (end - addr) >> PAGE\_SHIFT; /\*新内存区域长度，页数量\*/

struct vm\_area\_struct \*area, \*next;

int err;

if (vm\_flags & VM\_SPECIAL) /\*特殊VMA不能与其它VMA合并\*/

return NULL;

if (prev)

next = prev->vm\_next; /\*prev之后的VMA\*/

else

**next = mm->mmap**; /\*地址空间第一个VMA\*/

**area = next**;

if (next && next->vm\_end == end) /\*mprotect()系统调用情形（1），修改整个VMA\*/

next = next->vm\_next;

if (prev && prev->vm\_end == addr &&mpol\_equal(vma\_policy(prev), policy) \

&&can\_vma\_merge\_after(prev, vm\_flags,anon\_vma, file, pgoff)) {

/\*判断prev能否与其后的[addr,end)内存区域合并\*/

if (next && end == next->vm\_start &&mpol\_equal(policy, vma\_policy(next)) &&

can\_vma\_merge\_before(next, vm\_flags,anon\_vma, file, pgoff+pglen) &&

is\_mergeable\_anon\_vma(prev->anon\_vma,next->anon\_vma, NULL)) {

/\*再判断next能否与其前面的[addr,end)内存区域合并\*/

err = **vma\_adjust(prev, prev->vm\_start,next->vm\_end, prev->vm\_pgoff, NULL)**;

/\*[addr,end)内存区域能同时与其前后VMA合并，全部合并到prev内存域\*/

/\*mprotect()系统调用情形（1）\*/

} else

err = **vma\_adjust(prev, prev->vm\_start,end, prev->vm\_pgoff, NULL)**;

/\*[addr,end)内存区域只能与其前一VMA合并，合并到prev内存域\*/

/\*mprotect()系统调用情形（2）\*/

if (err)

return NULL;

khugepaged\_enter\_vma\_merge(prev, vm\_flags);

return **prev**; /\*返回合并后VMA实例指针\*/

}

/\*[addr,end)内存区域不能与其前一内存域合并，再判断能否与其后一VMA合并\*/

if (next && end == next->vm\_start &&mpol\_equal(policy, vma\_policy(next)) &&

can\_vma\_merge\_before(next, vm\_flags,anon\_vma, file, pgoff+pglen)) {

/\*判断next能否与其前面的[addr,end)内存区域合并\*/

if (prev && addr < prev->vm\_end) /\*mprotect()系统调用情形（3）\*/

err = **vma\_adjust(prev, prev->vm\_start,addr, prev->vm\_pgoff, NULL)**;

/\*收缩前一内存域，[addr,end)内存区域合并至下一VMA\*/

else

err = **vma\_adjust(area, addr, next->vm\_end,next->vm\_pgoff - pglen, NULL)**;

/\*[addr,end)与前一VMA没有重叠，合并到下一VMA\*/

if (err)

return NULL;

khugepaged\_enter\_vma\_merge(area, vm\_flags);

**return area**; /\*返回合并后VMA实例指针（后一VMA）\*/

}

return NULL; /\*不能合并，返回NULL\*/

}

vma\_merge()函数根据[addr,end)内存区域确定前后VMA实例，由prev和next指针变量指向它们；然后，判断prev能否与其后的[addr,end)内存区域合并，如果能则再判断[addr,end)内存区域能否与next合并，如果能则调整prev实例，将三个内存区域合并成一个，如果[addr,end)内存区域不能与next合并，则只与prev合并，函数返回合并后VMA指针；如果[addr,end)内存区域不能与prev合并，则再判断其能否与next合并，能则执行合并，函数返回合并后VMA指针。如果[addr,end)内存区域不能与前后内存域合并，函数返回NULL。

can\_vma\_merge\_before()和can\_vma\_merge\_after()函数用于判断给定VMA能否与其前/后的由参数指定的映射区域合并，能合并则函数返回1，否则返回0。这两个函数定义在/mm/mmap.c文件内，源代码请读者自行阅读。

如果可以合并在vma\_merge()函数中将调用vma\_adjust()函数调整生成合并后的VMA。被调整的VMA可能是[addr,end)内存区域之前的VMA，也可能是其后的VMA。下面将介绍vma\_adjust()函数的实现。

#### 1调整VMA

vma\_adjust()函数用于调整现有VMA，应用的时机主要有：

（1）合并VMA时，如果能执行合并操作，则调用此函数对现有VMA进行调整。

（2）拆分VMA时（以VMA内某个地址将VMA一分为二，后面介绍），调用此函数调整被拆分的VMA，并将新创建VMA插入管理结构。拆分VMA是将VMA一分为二，因此需要新建一个vm\_area\_struct实例。

下面先用图示的方式来说明调整VMA的操作，如下图所示：



图中红色标记的vma表示传递给vma\_adjust()函数的需调整的VMA实例指针。

合并VMA中有5种调整VMA的情形：

（1）[addr,end)能与prev和next合并，生成一个大的VMA，调整prev（向上扩展）。

（2）[addr,end)只能与prev合并，与nxet无重叠且不能合并，调整prev（向上扩展）。

（3）[addr,end)是next的一部分，能与prev合并，调整prev（向上扩展）。

（4）[addr,end)是prev的一部分，能与next合并，调整prev（向下收缩）。

（5）[addr,end)只能与next合并，与prev无重叠且不能合并，[addr,end)为已有VMA，调整区域所在VMA，[addr,end)为未映射区域，调整next（向下扩展）。

需要注意的是，在情形（1）（2）（5）中，中间区域可以是一个已存在的VMA，也可以是未建立映射（欲建立映射）的区域，在阅读下面vma\_adjust()函数代码时要注意这两种情况。

拆分VMA时有2种调整VMA的情形：

（1）新VMA（insert）位于上部，调整原vma位于下部（向下收缩）。

（2）新VMA（insert）位于下部，调整原vma位于上部（向上收缩）。

vma\_adjust()函数定义如下（/mm/mmap.c）：

int vma\_adjust(struct vm\_area\_struct \*vma, unsigned long start,unsigned long end, \

pgoff\_t pgoff, struct vm\_area\_struct \*insert)

/\*vma：被调整VMA，start,end：调整后VMA新的起止地址，pgoff：被调整VMA新的页偏移量，

\*insert：新建VMA指针，只在拆分VMA时使用，其它时机为NULL（合并操作中为NULL）。\*/

{

struct mm\_struct \*mm = vma->vm\_mm;

struct vm\_area\_struct \***next = vma->vm\_next**; /\*下一VMA\*/

struct vm\_area\_struct \*importer = NULL; /\*扩展VMA指针\*/

struct address\_space \*mapping = NULL;

struct rb\_root \*root = NULL;

struct anon\_vma \*anon\_vma = NULL;

struct file \***file = vma->vm\_file**; /\*映射文件\*/

bool start\_changed = false, end\_changed = false;

long adjust\_next = 0;

int remove\_next = 0;

if (next && !insert) { /\*合并VMA\*/

struct vm\_area\_struct \*exporter = NULL; /\*收缩VMA指针\*/

if (end >= next->vm\_end) { /\*合并情形（1）（2）（5）\*/

/\*（1）调整整个VMA访问属性：

\* 只能与前一VMA或下一VMA合并，remove\_next=1；

\* 能与前一VMA和下一VMA合并，remove\_next=2。

\*（2）中间为未映射区域：

\* 能与前一VMA和下一VMA合并，remove\_next=1。

\*remove\_next表示需要释放VMA实例的数量，即合并现有VMA次数。

\*/

**again:** **remove\_next = 1 + (end > next->vm\_end)**; /\*vma向后合并次数\*/

end = next->vm\_end;

exporter = next; /\*收缩（释放）的VMA\*/

importer = vma; /\*扩展的VMA\*/

} else if (end > next->vm\_start) {

/\*合并情形（3），下一VMA的一部分能与前一VMA合并\*/

adjust\_next = (end - next->vm\_start) >> PAGE\_SHIFT; /\*前一VMA扩展的页数大小\*/

exporter = next;

importer = vma;

} else if (end < vma->vm\_end) { /\*合并情形（4），本VMA一部分与下一VMA合并\*/

adjust\_next = -((vma->vm\_end - end) >> PAGE\_SHIFT); /\*收缩大小页数\*/

exporter = vma; /\*本VMA缩小\*/

importer = next; /\*下一VMA扩展\*/

}

if (exporter && exporter->anon\_vma && !importer->anon\_vma) {

int error; /\*扩展VMA复制收缩VMA匿名反向映射结构\*/

**importer->anon\_vma = exporter->anon\_vma**;

error = anon\_vma\_clone(importer, exporter);

if (error)

return error;

}

} /\*if (next && !insert)结束，确定合并VMA次数或调整量\*/

if (file) { /\*文件映射\*/

mapping = file->f\_mapping; /\*文件地址空间\*/

**root = &mapping->i\_mmap**; /\*红黑树根节点\*/

uprobe\_munmap(vma, vma->vm\_start, vma->vm\_end);

if (adjust\_next)

uprobe\_munmap(next, next->vm\_start, next->vm\_end);

i\_mmap\_lock\_write(mapping);

if (insert) {

**\_\_vma\_link\_file(insert)**; /\*拆分VMA时，将新VMA插入文件反向映射结构\*/

}

}

vma\_adjust\_trans\_huge(vma, start, end, adjust\_next);

anon\_vma = vma->anon\_vma; /\*被调整VMA匿名反向映射结构\*/

if (!anon\_vma && adjust\_next) /\*匿名反向映射结构为空，且需要合并/收缩部分VMA区域\*/

**anon\_vma = next->anon\_vma**; /\*共用下一VMA的反向映射结构\*/

if (anon\_vma) { /\*反向映射结构不为空\*/

VM\_BUG\_ON\_VMA(adjust\_next && next->anon\_vma &&anon\_vma != next->anon\_vma, next);

anon\_vma\_lock\_write(anon\_vma);

**anon\_vma\_interval\_tree\_pre\_update\_vma(vma)**; /\*/mm/mmap.c\*/

/\*扫描vma->anon\_vma\_chain链表，链表成员从各anon\_vma区间树中移出\*/

if (adjust\_next)

anon\_vma\_interval\_tree\_pre\_update\_vma(next); /\*对下一VMA执行同样的操作\*/

}

if (root) { /\*文件地址空间中红黑树根节点\*/

flush\_dcache\_mmap\_lock(mapping);

**vma\_interval\_tree\_remove(vma, root)**; /\*被调整VMA从红黑树中移出\*/

if (adjust\_next)

**vma\_interval\_tree\_remove(next, root)**; /\*下一VMA也移出红黑树\*/

}

if (start != vma->vm\_start) { /\*合并VMA情形（5），拆分VMA情形（2）\*/

vma->vm\_start = start;

start\_changed = true;

}

if (end != vma->vm\_end) { /\*合并VMA情形（1）（2）（3）（4），拆分VMA情形（1）\*/

vma->vm\_end = end;

end\_changed = true;

}

vma->vm\_pgoff = pgoff; /\*VMA偏移量\*/

if (adjust\_next) { /\*对下一VMA进行调整，扩展或收缩，合并VMA情形（3）（4）\*/

next->vm\_start += adjust\_next << PAGE\_SHIFT;

next->vm\_pgoff += adjust\_next;

}

if (root) { /\*文件映射\*/

if (adjust\_next)

**vma\_interval\_tree\_insert(next, root)**; /\*下一VMA重新插入文件地址空间红黑树\*/

**vma\_interval\_tree\_insert(vma, root)**; /\*被调整VMA重新插入文件地址空间红黑树\*/

flush\_dcache\_mmap\_unlock(mapping);

}

if (remove\_next) { /\*合并VMA情形（1）（2）（5）中，有需要释放的VMA\*/

\_\_vma\_unlink(mm, next, vma); /\*将next从mm\_struct管理链表和红黑树中移出\*/

if (file)

\_\_remove\_shared\_vm\_struct(next, file, mapping);

/\*将next从文件地址空间红黑树中移出，/mm/mmap.c\*/

} else if (insert) { /\*拆分VMA\*/

**\_\_insert\_vm\_struct(mm, insert)**; /\*将新VMA插入地址空间管理结构，没有处理反向映射\*/

} else { /\*不需要释放VMA的情形\*/

if (start\_changed) /\*vma基地址改变了\*/

vma\_gap\_update(vma); /\*更新VMA中rb\_subtree\_gap成员\*/

if (end\_changed) { /\*VMA结束地址变了\*/

if (!next)

mm->highest\_vm\_end = end;

else if (!adjust\_next)

vma\_gap\_update(next);

}

}

if (anon\_vma) { /\*vma的反向映射结构\*/

**anon\_vma\_interval\_tree\_post\_update\_vma(vma)**; /\*/mm/mmap.c\*/

/\*扫描vma->anon\_vma\_chain链表，链表成员插入各anon\_vma区间树\*/

if (adjust\_next) /\*对下一VMA进行同样的处理\*/

**anon\_vma\_interval\_tree\_post\_update\_vma(next)**;

anon\_vma\_unlock\_write(anon\_vma);

}

if (mapping)

i\_mmap\_unlock\_write(mapping);

if (root) { /\*文件映射\*/

uprobe\_mmap(vma);

if (adjust\_next)

uprobe\_mmap(next);

}

if (remove\_next) { /\*需要移除VMA的情形\*/

if (file) {

uprobe\_munmap(next, next->vm\_start, next->vm\_end);

fput(file);

}

if (next->anon\_vma)

anon\_vma\_merge(vma, next); /\*能否合并匿名反向映射结构\*/

mm->map\_count--;

mpol\_put(vma\_policy(next));

**kmem\_cache\_free(vm\_area\_cachep, next)**; /\*释放vm\_area\_struct实例\*/

**next = vma->vm\_next**;

if **(remove\_next == 2)**

**goto again**; /\*合并三个VMA的情形，再执行一次合并\*/

else if (next)

vma\_gap\_update(next);

else

**mm->highest\_vm\_end = end**;

}

if (insert && file)

uprobe\_mmap(insert);

validate\_mm(mm); /\*主要用于输出信息，/mm/mmap.c\*/

return 0;

}

vma\_adjust()函数需要考虑的情形比较多，逻辑比较复杂，下面对此函数所做工作进行一个小结，请读者结合小结阅读源代码。

（1）拆分VMA时，vma\_adjust()函数所做的工作主要是调整原VMA实例，向上或向下收缩，调整原VMA在mm\_struct管理结构中位置及反向映射结构信息等，并将新VMA实例插入mm\_struct管理结构和反向映射结构中等。

（2）在合并VMA时，如果是由对原有VMA中的部分区域进行访问属性调整（情形（3）（4））引起的，处理起来比较简单，就是调整前后两个VMA实例起止地址（adjust\_next不为0），以及在mm\_struct管理结构中位置和反向映射信息。

（3）当调整VMA操作是由于合并未映射区域或整个VMA改变访问属性引起的，情况要复杂一些：

a：未映射区域只能与前一VMA或后一VMA合并，只需要调整前一VMA或后一VMA实例，包括在地址空间管理结构中位置和反向映射结构信息。

b：未映射区域能同时与前一VMA和后一VMA合并，调整前一VMA，释放后一VMA（remove\_next=1）。

c：调整整个VMA访问属性时，若能与前一VMA或后一VMA合并，则调整前一VMA或本VMA，释放本VMA或后一VMA（remove\_next=1）。

d：调整整个VMA访问属性时，能与前一VMA合并，还能与后一VMA合并，则要执行两次调整操作。第一次将本VMA与前一VMA合并，释放本VMA，第二次是将第一次合并生成的VMA与下一VMA合并，释放下一VMA（remove\_next=2）。

remove\_next变量表示的是合并现有VMA的次数，合并一次就要将后面一个VMA释放。如果不需要合并现有VMA，则只是对现有VMA进行调整。adjust\_next变量表示在调整VMA时（不需要合并VMA），前一VMA的调整量（页数），正值表示扩展（后一VMA收缩），负值表示收缩（后一VMA扩展）。

## 4.6用户空间映射管理

前面介绍过，内核在创建进程加载可执行目标文件时，将创建并初始化进程地址空间。进程地址空间初始状态包括代码/数据映射区（文件映射）、堆、内存映射区（可能映射了动态库）和栈等。进程在运行过程中可以通过系统调用对自身地址空间（虚拟内存）进行操作，范围主要包括堆和内存映射区，操作内容有创建映射、解除映射、锁定/解锁映射等。栈在进程运行过程中由内核管理，后面将介绍。

本节介绍进程地址空间管理相关的系统调用，这对用户空间编程来说是非常重要的内容，是用户进程操作其地址空间（虚拟内存）的接口。

### 4.6.1锁定与解锁内存区

锁定内存区是指设置内存区跨越的VMA标记成员的MAP\_LOCKED标记位，立即为VMA分配物理页帧建立映射，并且设置映射页帧page实例的PG\_mlocked标记位，page实例添中到内存域zone实例中的不可回收LRU链表中，表示页帧不可以被回收。

解锁内存区操作是指解除内存区的锁定，清除跨越VMA标记成员的VM\_LOCKED标记位，并将映射页帧page实例添加到可回收LRU链表。注意，内核可对已经锁定的内存区域执行解锁操作，也可以对没有锁定的内存区域执行解锁操作。

内核提供的锁定/解锁进程虚拟内存区的系统调用如下（/mm/mlock.c）：

●mlock(start, len)：锁定进程地址空间从start开始，长度为len（字节数）的区域。

●mlockall(flags)：锁定进程所有映射区域。

●munlock(start,len)：解锁进程地址空间指定start、len表示的区域。

●mlockall(flags)：解锁进程所有映射区域。

#### 1锁定操作

锁定操作包括锁定指定内存区和锁定整个进程地址空间，下面分别介绍。

##### **■锁定指定内存区**

锁定指定内存区是指对进程地址空间中指定连续的内存区进行锁定操作，并立即为其分配物理页帧，修改页表项，建立映射。锁定内存区必须位于VMA内部，可以跨越多个地址连续的VMA，但中间不能有空洞。

锁定内存区mlock()系统调用实现函数定义如下（/mm/mlock.c）：

SYSCALL\_DEFINE2(mlock, unsigned long, start, size\_t, len)

/\*start：必须位于现有某个VMA内部，len：锁定内存区长度\*/

{

unsigned long locked;

unsigned long lock\_limit;

int error = -ENOMEM;

if (!can\_do\_mlock()) /\*是否可以执行锁定操作，检查有没有超过限制值等，/mm/mlock.c\*/

return -EPERM;

**lru\_add\_drain\_all()**; /\*清除pagevec页缓存，详见第11章\*/

len = PAGE\_ALIGN(len + (start & ~PAGE\_MASK)); /\*起始地址、长度页对齐，下对齐\*/

start &= PAGE\_MASK;

lock\_limit = rlimit(RLIMIT\_MEMLOCK);

lock\_limit >>= PAGE\_SHIFT;

locked = len >> PAGE\_SHIFT; /\*本次锁定页数\*/

down\_write(&current->mm->mmap\_sem);

**locked += current->mm->locked\_vm**; /\*执行完本次锁定操作后，进程锁定页总数量\*/

/\*锁定页总数量不能超过限制值\*/

if ((locked <= lock\_limit) || capable(CAP\_IPC\_LOCK))

**error = do\_mlock(start, len, 1)**; /\*执行锁定操作，执行对VMA的操作，/mm/mlock.c\*/

up\_write(&current->mm->mmap\_sem);

if (error)

return error;

error = **\_\_mm\_populate(start, len, 0)**; /\*分配物理页帧，修改页表项，建立映射，/mm/gup.c\*/

if (error)

return \_\_mlock\_posix\_error\_return(error);

return 0;

}

mlock()系统调用实现函数首先判断进程锁定页数量是否超过进程资源的限制值，以确定是否可以执行本次锁定操作；如果可以执行，再调用do\_mlock()函数执行锁定操作，主要是对VMA实例进行操作；最后调用\_\_mm\_populate()函数为锁定内存区逐页分配物理页帧，修改页表项，建立映射。

\_\_mm\_populate()函数的操作比较复杂，下一小节将专门介绍。下面介绍do\_mlock()函数的实现，此函数主要是扫描start、len内存区跨越的VMA实例，对各VMA实例进行操作。

do\_mlock()函数定义如下（/mm/mlock.c）：

static int do\_mlock(unsigned long start, size\_t len, int on)

/\*start：锁定内存区起始虚拟地址，len：锁定内存区长度，on：1表示锁定，0表示解锁\*/

{

unsigned long nstart, end, tmp;

struct vm\_area\_struct \* vma, \* prev;

int error;

VM\_BUG\_ON(start & ~PAGE\_MASK); /\*必须页对齐\*/

VM\_BUG\_ON(len != PAGE\_ALIGN(len));

**end = start + len**; /\*内存区结束地址，页对齐\*/

if (end < start)

return -EINVAL;

if (end == start)

return 0;

**vma** = find\_vma(current->mm, start); /\*结束地址在start之后的VMA\*/

if (**!vma || vma->vm\_start > start**) /\*start必须位于现有VMA内部\*/

return -ENOMEM;

prev = vma->vm\_prev; /\*前一VMA\*/

if **(start > vma->vm\_start)**  /\*start在VMA内部，不是VMA起始地址，VMA需要被拆分**\*/**

prev = vma;

for (nstart = start ; ; ) { /\*遍历[start,end)跨越的VMA\*/

vm\_flags\_t newflags;　　　　/\*VMA新标记成值\*/

/\*vma->vm\_start <= nstart < vma->vm\_end. \*/

newflags = **vma->vm\_flags & ~VM\_LOCKED**; /\*清除VM\_LOCKED标记位\*/

if (**on**) /\*锁定操作，设置VM\_LOCKED标记位\*/

**newflags |= VM\_LOCKED**;

tmp = vma->vm\_end;

if (tmp > end)

tmp = end;

error = **mlock\_fixup(vma, &prev, nstart, tmp, newflags)**; /\*锁定/解锁VMA，/mm/mlock.c\*/

if (error)

break;

**nstart = tmp**;

if (nstart < prev->vm\_end) /\*是否遍历到跨越的最后一个VMA\*/

nstart = prev->vm\_end;

if (nstart >= end)

break; /\*遍历到最后一个VMA，退出循环\*/

vma = prev->vm\_next;

if (!vma || vma->vm\_start != nstart) { /\*锁定/解锁内存区必须位于连续的VMA内部\*/

error = -ENOMEM;

break;

}

} /\*遍历VMA结束\*/

return error;

}

mlock()系统调用锁定的内存区域[start,end)（解锁区域也一样）必须位于某一VMA内部或者位于地址连续的VMA内部，中间不能有空洞，如下图所示。

do\_mlock()函数遍历[start,end)跨越的VMA，对每个VMA调用mlock\_fixup()函数。mlock\_fixup()函数即可用于锁定VMA，也可用于解锁VMA。

mlock\_fixup()函数内判断锁定/解锁区域是否只是VMA的一部分，是则需要拆分VMA。如下图所示，假设锁定区域[start,end0)位于VMA0内部，则需要对VMA0执行两次拆分操作，需要锁定/解锁的内存区域划分出来形成新的VMA。如果锁定/解锁区域是整个VMA（如下图中VMA1，假设结束地址为end1）则无需拆分。

随后，对需要锁定/解锁的VMA（拆分后VMA）执行相应的操作，如果是锁定操作则设置VMA标记成员的VM\_LOCKED标记位，如果是解锁操作则对VMA调用munlock\_vma\_pages\_range()函数完成解锁操作。



mlock\_fixup()函数代码如下（/mm/mlock.c）：

static int mlock\_fixup(struct vm\_area\_struct \*vma, struct vm\_area\_struct \*\*prev, \

unsigned long start, unsigned long end, vm\_flags\_t newflags)

/\*newflags：VMA新标记\*/

{

struct mm\_struct \*mm = vma->vm\_mm;

pgoff\_t pgoff;

int nr\_pages;

int ret = 0;

int **lock = !!(newflags & VM\_LOCKED)**; /\*VMA新标记是否设置了VM\_LOCKED标记位\*/

if (newflags == vma->vm\_flags || (vma->vm\_flags & VM\_SPECIAL) ||

is\_vm\_hugetlb\_page(vma) || vma == get\_gate\_vma(current->mm))

goto out; /\*不需要执行操作的情况\*/

pgoff = vma->vm\_pgoff + ((start - vma->vm\_start) >> PAGE\_SHIFT);

/\*以start开始的下一VMA的偏移量\*/

\*prev = vma\_merge(mm, \*prev, start, end, newflags, vma->anon\_vma,vma->vm\_file, pgoff, \

vma\_policy(vma));

/\*修改VMA标记后检查能否与前后VMA合并，能合并则合并\*/

if (\*prev) { /\*可以合并则跳至success\*/

vma = \*prev;

goto success;

}

/\*不能合并，判断是否需要拆分VMA\*/

if (start != vma->vm\_start) {　　　　　　/\*以start拆分VMA\*/

ret = **split\_vma(mm, vma, start, 1)**; /\*新VMA位于下部，vma位于上部\*/

if (ret)

goto out;

}

if (end != vma->vm\_end) { /\*以end拆分VMA\*/

ret = **split\_vma(mm, vma, end, 0)**; /\*新VMA位于上部，vma位于下部\*/

if (ret)

goto out;

}

success:

nr\_pages = (end - start) >> PAGE\_SHIFT;

if (!lock)

nr\_pages = -nr\_pages;

mm->locked\_vm += nr\_pages; /\*更新锁定页数量\*/

if (lock) /\*锁定内存域\*/

**vma->vm\_flags = newflags**;

/\*设置锁定区域对应VMA的标记成员，置位VM\_LOCKED标记位\*/

else

**munlock\_vma\_pages\_range(vma, start, end)**; /\*解锁内存域，后面再做介绍\*/

out:

**\*prev = vma**;

return ret;

}

对于需要锁定的VMA，这里只是简单地设置VMA标记成员的VM\_LOCKED标记位，在mlock()系统调用随后调用的\_\_mm\_populate()函数中将为VMA分配页帧，修改页表项，建立映射。

对于需要解锁的VMA，将调用munlock\_vma\_pages\_range()函数进行处理，后面讲解锁系统调用时再介绍此函数的实现。

##### **■锁定所有内存域**

锁定进程整个虚拟内存区的系统调用为mlockall(int flags)，内核在/arch/mips/include/uapi/asm/mman.h头文件定义了标记参数flags的取值：

#define MCL\_CURRENT 1 /\*锁定进程已有VMA\*/

#define MCL\_FUTURE 2 /\*锁定进程将来创建的VMA\*/

mlockall()系统调用flags参数必须设置两个标记位中的一位或两位，并且不能设置除低两位之外的其它位，因此flags参数有3种状态，对应的操作如下。

●**MCL\_CURRENT**：清除进程地址空间mm->def\_flags标记成员VM\_LOCKED标记位，对现有VMA执行锁定操作（立即分配物理页帧建立映射），之后创建的VMA默认不锁定。

●**MCL\_FUTURE**：设置进程地址空间mm->def\_flags标记成员VM\_LOCKED标记位，对现有VMA不做修改，之后创建的VMA默认锁定。

●**MCL\_CURRENT|MCL\_FUTURE**：设置进程地址空间mm->def\_flags标记成员VM\_LOCKED标记位，对现有VMA执行锁定操作（立即分配物理页帧建立映射），之后创建的VMA默认锁定。这是真正地锁定整个进程地址空间。

mlockall系统调用实现函数如下（/mm/mlock.c）：

SYSCALL\_DEFINE1(mlockall, int, flags)

{

unsigned long lock\_limit;

int ret = -EINVAL;

if (!flags || (flags & ~(MCL\_CURRENT | MCL\_FUTURE))) /\*参数检查\*/

goto out;

ret = -EPERM;

if (!can\_do\_mlock()) /\*是否可以锁定\*/

goto out;

if (flags & MCL\_CURRENT)

**lru\_add\_drain\_all()**; /\*flush pagevec \*/

lock\_limit = rlimit(RLIMIT\_MEMLOCK);

lock\_limit >>= PAGE\_SHIFT;

ret = -ENOMEM;

down\_write(&current->mm->mmap\_sem);

if (!(flags & MCL\_CURRENT) || (current->mm->total\_vm <= lock\_limit) ||capable(CAP\_IPC\_LOCK))

ret = **do\_mlockall(flags)**; /\*执行锁定操作，/mm/mlock.c\*/

up\_write(&current->mm->mmap\_sem);

if (!ret && (**flags & MCL\_CURRENT**)) /\*如果设置了MCL\_CURRENT标记\*/

**mm\_populate(0, TASK\_SIZE)**; /\*为现有所有VMA分配物理页帧，修改页表项，建立映射\*/

out:

return ret;

}

如果进程锁定页数量没有超过限制值，不管flags参数有没有设置MCL\_CURRENT标记位，都将调用

do\_mlockall(flags)函数，对内存域执行锁定操作，这里的锁定只是修改VMA标记位，没有建立映射。

如果flags参数设置了MCL\_FUTURE标记位，在do\_mlockall(flags)函数中将设置地址空间默认VMA标记的VM\_LOCKED标记位，以后创建的VMA默认锁定。

如果flags参数设置了MCL\_CURRENT标记位，在do\_mlockall(flags)函数中将设置现有所有VMA的VM\_LOCKED标记位。随后，在mm\_populate(0, TASK\_SIZE)函数中将为进程所有现有VMA分配物理页帧，修改页表项，建立映射。mm\_populate()函数内调用\_\_mm\_populate()函数，后面再介绍此函数的实现。

do\_mlockall()函数根据flags值修改地址空间默认标记值及现有VMA标记值，函数定义如下：

static int do\_mlockall(int flags) /\*/mm/mlock.c\*/

{

struct vm\_area\_struct \* vma, \* prev = NULL;

if (flags & MCL\_FUTURE) /\*设置了MCL\_FUTURE标记位\*/

current->mm->**def\_flags |**= VM\_LOCKED; /\*设置默认VMA标记中VM\_LOCKED标记位\*/

else /\*只设置了MCL\_CURRENT标记位\*/

current->mm->def\_flags &= ~VM\_LOCKED; /\*清除默认VMA标记中VM\_LOCKED标记位\*/

if (flags == MCL\_FUTURE) /\*只设置了MCL\_FUTURE标记位，函数返回\*/

goto out;

/\*如果设置了MCL\_CURRENT标记，则对现有VMA执行锁定操作\*/

for (vma = current->mm->mmap; vma ; vma = prev->vm\_next) {

vm\_flags\_t newflags;

**newflags = vma->vm\_flags & ~VM\_LOCKED**; /\*清除VM\_LOCKED标记位\*/

if (flags & MCL\_CURRENT)

newflags |= VM\_LOCKED; /\*设置VM\_LOCKED标记位\*/

**mlock\_fixup(vma, &prev, vma->vm\_start, vma->vm\_end, newflags)**; /\*锁定VMA\*/

cond\_resched\_rcu\_qs();

}

out:

return 0;

}

#### 2解锁操作

解锁内存区操作是解除内存区域的锁定（清除VMA标记VM\_LOCKED标记位），并将映射页帧page实例添加到可回收LRU链表。注意，内核既可对已经锁定的内存区域执行解锁操作，也可以对没有锁定的内存区域执行解锁操作。

解锁进程指定虚拟内存区域的系统调用为munlock()，实现函数如下（/mm/mlock.c）：

SYSCALL\_DEFINE2(munlock, unsigned long, start, size\_t, len)

/\*start：起始地址，len：长度\*/

/\*[start,start+len)内存区域必须位于VMA或连续VMA内部，不能有空洞\*/

{

int ret;

len = PAGE\_ALIGN(len + (start & ~PAGE\_MASK)); /\*起始地址、长度页对齐\*/

start &= PAGE\_MASK;

down\_write(&current->mm->mmap\_sem);

**ret = do\_mlock(start, len, 0)**; /\*解锁内存区域，标记参数为0，/mm/mlock.c\*/

up\_write(&current->mm->mmap\_sem);

return ret;

}

munlock()系统调用调用前面介绍过的do\_mlock()函数，逐个对[start,start+len)跨越的VMA进行拆分，需要解锁的区域生成新的VMA，调用**munlock\_vma\_pages\_range()**函数对新VMA进行解锁操作，函数调关系简列如下。



munlock\_vma\_pages\_range()函数首先清除VMA的VM\_LOCKED标记位，然后定义一个pagevec结构体实例，用于缓存解锁的page实例，pagevec结构体中包含page指针数组，数组项数为14。随后，调用

\_\_munlock\_pagevec\_fill()函数从start虚拟地址对应的PTE页表项开始扫描，获取映射页帧page实例并关联到pagevec实例指针数组。如果指针数组填满，\_\_munlock\_pagevec\_fill()函数返回，继续调用函数\_\_munlock\_pagevec()解锁指针数组缓存页，然后再调用\_\_munlock\_pagevec\_fill()函数获取需要解锁的页，如此循环直至扫描完整个VMA的PTE页表项。

pagevec结构体定义在/include/linux/pagevec.h头文件，表示页向量，用于缓存需要解锁页的page实例：

struct pagevec {

unsigned long nr; /\*缓存中页数量\*/

unsigned long cold; /\*\*/

struct page \*pages[PAGEVEC\_SIZE]; /\*指针数组，14项\*/

};

munlock\_vma\_pages\_range()函数代码如下（/mm/mlock.c）：

void munlock\_vma\_pages\_range(struct vm\_area\_struct \*vma,unsigned long start, unsigned long end)

/\*vma：VMA指针，start：VMA起始地址，end：VMA结束地址\*/

{

vma->vm\_flags &= ~VM\_LOCKED; /\*清除VMA的VM\_LOCKED标记位\*/

while (start < end) {

/\*依次扫描VMA对应PTE页表项，每次循环填满pagevec指针数组，并解锁缓存页\*/

struct page \*page = NULL;

unsigned int page\_mask;

unsigned long page\_increm;

**struct pagevec pvec**; /\*pagevec实例\*/

struct zone \*zone;

int zoneid;

pagevec\_init(&pvec, 0); /\*初始化pagevec实例，指针数组清空\*/

**page = follow\_page\_mask(vma, start, FOLL\_GET | FOLL\_DUMP,&page\_mask)**;

/\*获取start虚拟页映射页帧page实例，/mm/gup.c\*/

/\*start映射页帧page实例存在\*/

if (page && !IS\_ERR(page)) {

if (PageTransHuge(page)) {

...

} else {

**pagevec\_add(&pvec, page)**; /\*添加到pagevec指针数组，返回0表示缓存已满\*/

zone = page\_zone(page); /\*物理内存域指针\*/

zoneid = page\_zone\_id(page); /\*物理内存域编号\*/

start = **\_\_munlock\_pagevec\_fill(&pvec, vma,zoneid, start, end)**;

/\*扫描PTE页表，填满pagevec实例页缓存，/mm/mlock.c\*/

**\_\_munlock\_pagevec(&pvec, zone)**; /\*解锁pagevec实例缓存页，/mm/mlock.c\*/

goto next; /\*条件重调度后再次执行循环\*/

}

}

VM\_BUG\_ON((start >> PAGE\_SHIFT) & page\_mask);

page\_increm = 1 + page\_mask;

start += page\_increm \* PAGE\_SIZE;

next:

cond\_resched();

} /\*while循环结束\*/

}

munlock\_vma\_pages\_range()函数内是一个循环，每次循环初始化pagevec实例，调用follow\_page\_mask()函数从页表项中获取首页page实例，添加到缓存；然后调用\_\_munlock\_pagevec\_fill()函数扫描VMA对应页表项，获取随后映射页的page实例，添加到缓存，直至pagevec实例缓存填满，返回下一映射页地址；最后，调用\_\_munlock\_pagevec(&pvec, zone)函数解锁缓存页。以上操作将循环进行，直至扫描完VMA对应的所有页表项，即解锁VMA所有映射页。

##### ■扫描页表

\_\_munlock\_pagevec\_fill()函数用于扫描VMA页表项，获取映射页page实例，缓存至pagevec实例，直至扫描完指定页表项或pagevec实例缓存填满，函数定义如下（/mm/mlock.c）：

static unsigned long \_\_munlock\_pagevec\_fill(struct pagevec \*pvec, \

struct vm\_area\_struct \*vma, int zoneid,unsigned long start,unsigned long end)

/\*start：扫描范围起始地址，end：扫描范围结束地址\*/

{

pte\_t \*pte;

spinlock\_t \*ptl;

pte = get\_locked\_pte(vma->vm\_mm, start, &ptl);

/\*start对应PTE页表项指针，首页已添加到缓存\*/

/\*确保扫描不跨越页表边界，即在同一个PTE页表中扫描\*/

end = pgd\_addr\_end(start, end);

end = pud\_addr\_end(start, end);

end = pmd\_addr\_end(start, end);

start += PAGE\_SIZE; /\*start对应page实例已经添加到pagevec实例指针数组，因此跳过\*/

while (start < end) {

struct page \*page = NULL;

pte++;

if (pte\_present(\*pte))

page = **vm\_normal\_page(vma, start, \*pte)**;

/\*返回普通页帧page实例\*/

if (!page || **page\_zone\_id(page) != zoneid**) /\*确保页帧在同一物理内存域\*/

break;

get\_page(page); /\*增加page实例\_count成员计数值，/include/linux/mm.h\*/

start += PAGE\_SIZE;

if (**pagevec\_add(pvec, page) == 0**) /\*page实例添加到pagevec缓存\*/

/\*返回0表示指针数组填满，/include/linux/pagevec.h\*/

break;

}

pte\_unmap\_unlock(pte, ptl); /\*释放自旋锁\*/

return start; /\*返回下一扫描地址\*/

}

\_\_munlock\_pagevec\_fill()函数负责填充pagevec实例page指针数组，在扫描过程中确保扫描的PTE页表项位于同一个PTE页表中，并且页帧不跨越物理内存域。指针数组填满或扫描地址范围结束时函数返回，返回值是下一次扫描页的虚拟地址。

##### ■解锁页

\_\_munlock\_pagevec()函数用于解除pagevec实例中缓存page实例锁定，函数执行流程简列如下图所示：



锁定VMA映射的页帧page实例设置了PG\_mlocked标记位，并添加到不可回收LRU链表中，解除锁定则是清除PG\_mlocked标记位，并将其移至可回收LRU链表。如果页帧被多个内存域锁定，则减少其映射计数后，还需要将其释放回不可回收LRU链表。对于没有设置PG\_mlocked标记位的页，则减小其引用计数，如果引用计数为0，则将其释放回伙伴系统。

下面来看一下\_\_munlock\_pagevec()函数的实现，代码如下（/mm/mlock.c）：

static void \_\_munlock\_pagevec(struct pagevec \*pvec, struct zone \*zone)

{

int i;

int nr = **pagevec\_count(pvec)**; /\*缓存page实例数量\*/

int delta\_munlocked;

struct pagevec **pvec\_putback**;

int pgrescued = 0;

pagevec\_init(&pvec\_putback, 0); /\*初始化pagevec实例pvec\_putback\*/

/\*

\*第一步：（1）将设置PG\_mlocked标记位的page实例从（不可回收）LRU链表中移出；

\* （2）释放没有设置PG\_mlocked标记的page实例。

\*/

spin\_lock\_irq(&zone->lru\_lock);

for (i = 0; i < nr; i++) { /\*逐页扫描缓存页\*/

struct page \*page = pvec->pages[i];

if (TestClearPageMlocked(page)) { /\*处理设置PG\_mlocked标记位page实例，标记位清零\*/

if (**\_\_munlock\_isolate\_lru\_page(page, false)**)

/将page从（不可回收）LRU链表中移出，成功返回true\*/

continue; /\*扫描下一页\*/

else /\*page不在LRU链表\*/

\_\_munlock\_isolation\_failed(page); /\*报告事件，/mm/mlocked\*/

}

/\*未设置PG\_mlocked标记位page转移到pvec\_putback页缓存\*/

pagevec\_add(&pvec\_putback, pvec->pages[i]);

pvec->pages[i] = NULL;

}

delta\_munlocked = -nr + pagevec\_count(&pvec\_putback);

\_\_mod\_zone\_page\_state(zone, NR\_MLOCK, delta\_munlocked); /\*修改物理内存域统计量\*/

spin\_unlock\_irq(&zone->lru\_lock);

**pagevec\_release(&pvec\_putback)**; /\*释放未设置PG\_mlocked标记位page实例\*/

/\*page引用计数减1，如果为0则释放页到伙伴系统，/include/linux/pagevec.h\*/

/\*第二步：处理设置PG\_mlocked标记位的page实例\*/

for (i = 0; i < nr; i++) {

struct page \*page = pvec->pages[i];

if (page) {

**lock\_page(page)**; /\*锁定页，设置PG\_locked标记位，/include/linux/pagemap.h\*/

/\*页是否可解除锁定\*/

if (!**\_\_putback\_lru\_fast\_prepare(page, &pvec\_putback,&pgrescued)**) {

/\*不可解除锁定页，/mm/mlock.c\*/

get\_page(page);

**\_\_munlock\_isolated\_page(page)**; /\*释放回（不可回收）LRU链表，/mm/mlock.c\*/

**unlock\_page(page)**;

/\*清除PG\_locked标记位，唤醒等待页解锁进程，/mm/filemap.c\*/

put\_page(page); /\* from follow\_page\_mask() \*/

}

}

}

/\*第三步：将可解除锁定的页释放回（可回收）LRU链表\*/

if (pagevec\_count(&pvec\_putback))

**\_\_putback\_lru\_fast(&pvec\_putback, pgrescued)**; /\*将page放回到LRU链表，/mm/mlock.c\*/

}

这里所说的锁定是指内存域对页帧的锁定（页帧不可回收），关联的是page实例PG\_mlocked标记位，与关联PG\_locked标记位的页锁定不同，后者是指锁定页内容，内核不可对其内容进行读写。

解除VMA对页帧的锁定只是将页帧释放到可回收页LRU链表，而不是解除映射，因此无需处理页表项。page实例在物理内存域LRU链表中的操作详见第11章。

解锁进程所有内存区的系统调用为munlockall()（/mm/mlock.c），实现函数调用do\_mlockall(**0**)函数清除mm->def\_flags成员VM\_LOCKED标记位，对进程现有各VMA逐个调用mlock\_fixup()函数执行解锁操作，源代码请读者自行阅读。

### 4.6.2内存区建立映射

在前面介绍的锁定内存区和锁定所有内存域的系统调用中，将调用mm\_populate()或\_\_mm\_populate()函数为锁定区域逐页分配页帧，修改页表项，建立映射。这两个函数为用户空间一段连续使用的虚拟内存立即建立映射。

内核在创建映射的系统调用中（后面介绍），如果地址空间默认VMA标记设置了VM\_LOCKED标记位或系统调用标记参数设置了MAP\_POPULATE/MAP\_LOCKED标记位，在创建VMA实例后，将调用mm\_populate()函数立即为VMA建立映射。

本小节将介绍mm\_populate()/\_\_mm\_populate()函数的实现。

#### 1接口函数

内核中调用mm\_populate()/\_\_mm\_populate()函数为进程地址空间某段连续的内存区域逐页分配物理页帧，修改页表项，建立映射，函数调用关系简列如下图所示：



mm\_populate()函数参数指定了建立映射内存区的起始虚拟地址和长度，内部直接调用\_\_mm\_populate()函数。\_\_mm\_populate()函数遍历内存区跨越的每个VMA（不需要拆分）分别调用populate\_vma\_page\_range()函数，逐个为VMA建立映射（可以对VMA部分建立映射）。

populate\_vma\_page\_range()函数内调用\_\_get\_user\_pages()函数为VMA内指定虚拟内存段逐页分配页帧，修改页表项，建立映射。

mm\_populate()函数定义在/include/linux/mm.h头文件内：

static inline void mm\_populate(unsigned long addr, unsigned long len)

/\*addr：映射区起始地址，页对齐，len：映射区长度，字节数，页对齐，1：忽略错误\*/

{

(void) **\_\_mm\_populate(addr, len, 1)**; /\*/mm/gup.c\*/

}

mm\_populate()函数内直接调用\_\_mm\_populate(addr, len, 1)函数执行建立映射操作，\_\_mm\_populate()函数定义在/mm/gup.c文件内。

int \_\_mm\_populate(unsigned long start, unsigned long len, int ignore\_errors)

/\*start：起始虚拟地址，len：长度，字节数，ignore\_errors：是否忽略错误，1忽略，0不忽略\*/

{

struct mm\_struct \***mm = current->mm**; /\*当前进程地址空间\*/

unsigned long end, nstart, nend;

struct vm\_area\_struct \*vma = NULL;

int locked = 0; /\*初始值为0\*/

long ret = 0;

VM\_BUG\_ON(start & ~PAGE\_MASK); /\*start、len必须页对齐\*/

VM\_BUG\_ON(len != PAGE\_ALIGN(len));

**end = start + len**; /\*映射区域结束地址\*/

for (**nstart = start; nstart < end; nstart = nend**) { /\*遍历映射区域跨越的VMA\*/

if (!locked) {

**locked = 1**;

down\_read(&mm->mmap\_sem);

**vma = find\_vma(mm, nstart)**; /\*跨越的第一个VMA\*/

} else if (nstart >= vma->vm\_end)

**vma = vma->vm\_next**; /\*跨越的下一VMA\*/

if (!vma || vma->vm\_start >= end)

break;

nend = min(end, vma->vm\_end); /\*本次遍历的结束地址\*/

if (vma->vm\_flags & (VM\_IO | VM\_PFNMAP)) /\*跳过映射IO和PFN映射区\*/

continue;

if (**nstart < vma->vm\_start**)

**nstart = vma->vm\_start**; /\*取VMA的起始地址\*/

**ret = populate\_vma\_page\_range(vma, nstart, nend, &locked)**; /\* **\*locked=1**\*/

/\*为VMA内部指定内存段建立映射（或整个VMA），返回建立映射的页数，/mm/gup.c\*/

if (ret < 0) {

if (ignore\_errors) {

ret = 0;

continue; /\*忽略错误，返回值为0\*/

}

break;

}

**nend = nstart + ret \* PAGE\_SIZE**; /\*下次遍历起始地址\*/

ret = 0;

} /\*遍历VMA结束\*/

if (locked)

up\_read(&mm->mmap\_sem);

return ret; /\*返回0或错误码，忽略错误始终返回0\*/

}

\_\_mm\_populate()函数遍历虚拟内存区[start,end)跨越的VMA，调用populate\_vma\_page\_range()函数对VMA中需建立映射的区域建立映射。这里建立映射区域可以是整个VMA，也可以是VMA的一部分。

例如，如下图所示，假设[start,end)跨越了三个VMA，分别是VMA0、VMA1和VMA2，其中VMA0和VMA2只跨越了一部分（不需要拆分VMA）。\_\_mm\_populate()函数需调用populate\_vma\_page\_range()函数三次，分别为VMA0、VMA1和VMA2中的区域建立映射。

注意，\_\_mm\_populate()函数只为位于VMA内部的内存区建立映射，空洞区将忽略，不建立映射。



populate\_vma\_page\_range()函数为某个VMA内的虚拟内存区（或整个VMA）分配物理页帧，修改页表项，建立映射。内核调用此函数时保证内存区域位于VMA内部，不越界。

populate\_vma\_page\_range()函数代码如下（/mm/gup.c）：

long populate\_vma\_page\_range(struct vm\_area\_struct \*vma,unsigned long start, unsigned long end, \

int \*nonblocking)

/\*vma：目标VMA，start,end：起始结束地址\*/

{

struct mm\_struct \*mm = **vma->vm\_mm**;

unsigned long **nr\_pages = (end - start) / PAGE\_SIZE**; /\*建立映射页数量\*/

int gup\_flags;

VM\_BUG\_ON(start & ~PAGE\_MASK); /\*参数有效性判断\*/

VM\_BUG\_ON(end & ~PAGE\_MASK);

VM\_BUG\_ON\_VMA(start < vma->vm\_start, vma);

VM\_BUG\_ON\_VMA(end > vma->vm\_end, vma);

VM\_BUG\_ON\_MM(!rwsem\_is\_locked(&mm->mmap\_sem), mm);

gup\_flags = **FOLL\_TOUCH | FOLL\_POPULATE**; /\*设置\_\_get\_user\_pages()函数标记参数\*/

/\*根据VMA标记成员设置gup\_flags参数\*/

if ((vma->vm\_flags & (VM\_WRITE | VM\_SHARED)) == VM\_WRITE)

gup\_flags |= **FOLL\_WRITE**;

if (vma->vm\_flags & (VM\_READ | VM\_WRITE | VM\_EXEC))

gup\_flags |= **FOLL\_FORCE**;

return **\_\_get\_user\_pages(current, mm, start, nr\_pages, gup\_flags,NULL, NULL, nonblocking)**;

/\*为内存段逐页建立映射，返回建立映射的页数，/mm/gup.c\*/

}

populate\_vma\_page\_range()函数计算出需要建立映射的页数，设置标记参数gup\_flags后，调用函数\_\_get\_user\_pages()为VMA内存段（或整个VMA）逐页分配物理页帧，修改页表项、建立映射。

下面将介绍\_\_get\_user\_pages()函数的实现。

#### 2建立用户空间映射

\_\_get\_user\_pages()函数用于为用户进程地址空间一段虚拟内存建立映射，函数定义在/mm/gup.c文件内（get user pages），在这个文件内还定义了其它的些函数，它们都是\_\_get\_user\_pages()函数的包装器。

在介绍\_\_get\_user\_pages()函数实现之前，先了解一下函数标记参数的取值（/include/linux/mm.h），标记参数gup\_flags用于控制建立映射操作或设置映射页的访问权限属性：

#define **FOLL\_WRITE** 0x01 /\*确保页表项具有写权限\*/

#define **FOLL\_TOUCH** 0x02 /\*标记页被访问过\*/

#define FOLL\_GET 0x04 /\*对映射页page调用get\_page()，增加\_count引用计数\*/

#define **FOLL\_DUMP** 0x08 /\*如果空洞页（或全零页）报错\*/

#define **FOLL\_FORCE** 0x10 /\*分配页赋予读写权限\*/

#define FOLL\_NOWAIT 0x20 /\*如需发起IO操作（如读磁盘），发起后不等待\*/

#define **FOLL\_POPULATE** 0x40 /\*\*/

#define FOLL\_SPLIT 0x80 /\*不返回巨型页，将其拆分\*/

#define FOLL\_HWPOISON 0x100 /\*确保页是hwpoisoned \*/

#define FOLL\_NUMA 0x200 /\* force NUMA hinting page fault \*/

#define **FOLL\_MIGRATION** 0x400 /\*\*/

#define FOLL\_TRIED 0x800 /\* a retry, previous pass started an IO \*/

\_\_get\_user\_pages()函数执行流程简列如下图所示：



\_\_get\_user\_pages()函数首先由起始虚拟地址查找所在VMA实例；然后，遍历内存地址段中每个虚拟页，调用follow\_page\_mask()函数检查是否已经映射了页帧（或交换到交换区等），如果映射已经存在则无需建立映射，对已映射页page实例进行相应的操作后，返回page实例指针；如果映射页不存在，则调用faultin\_page()函数模拟产生了一个用户空间缺页异常，函数内调用缺页处理函数handle\_mm\_fault()建立页映射（后面介绍此函数实现），而后再次调用follow\_page\_mask()函数检查映射情况，返回映射页帧page实例指针，最后跳至下一页继续执行。如此循环，直至遍历完内存段中所有虚拟页。

\_\_get\_user\_pages()函数代码如下：

long \_\_get\_user\_pages(struct task\_struct \*tsk, struct mm\_struct \*mm,unsigned long start, unsigned long \

nr\_pages,unsigned int gup\_flags, struct page \*\*pages,struct vm\_area\_struct \*\*vmas, int \*nonblocking)

/\*

\*tsk：当前进程结构实例指针；mm：进程地址空间实例指针，start：建立映射起始虚拟地址；

\*nr\_pages：建立映射虚拟页数量；**gup\_flags**：标记参数；

\*pages：指向page指针数组，用于关联建立映射时分配页帧的page实例；

\*vmas：指向vm\_area\_struct 指针数组，用于关联内存段跨越的VMA实例；

\* \*nonblocking：是否等待IO操作或mmap\_sem信号量。

\*/

{

long i = 0;

unsigned int **page\_mask**;

struct vm\_area\_struct \***vma = NULL**; /\*初始化为NULL\*/

if (!nr\_pages)

return 0;

VM\_BUG\_ON(!!pages != !!(gup\_flags & FOLL\_GET));

if (!(gup\_flags & FOLL\_FORCE)) /\*标记参数没有设置FOLL\_FORCE标记位\*/

gup\_flags |= FOLL\_NUMA;

do { /\*循环开始\*/

struct page \*page;

unsigned int **foll\_flags = gup\_flags**;

unsigned int page\_increm;

if (!vma || start >= vma->vm\_end) { /\*跳过空洞区\*/

**vma = find\_extend\_vma(mm, start)**; /\*查找start所在的内VMA，/mm/mmap.c\*/

if (!vma && in\_gate\_area(mm, start)) {

/\*处理start处于空洞区的情况，如果没有定义\_\_HAVE\_ARCH\_GATE\_AREA宏，

\*in\_gate\_area()函数返回NULL，/include/linux/mm.h。\*/

int ret;

ret = get\_gate\_page(mm, start & PAGE\_MASK,gup\_flags, &vma, \

pages ? &pages[i] : NULL);

if (ret)

return i ? : ret;

page\_mask = 0;

goto next\_page;

}

if (**!vma** || **check\_vma\_flags(vma, gup\_flags)**)

/\*检查标记有效性，有效返回0，/mm/gup.c\*/

return i ? : -EFAULT;

if (is\_vm\_hugetlb\_page(vma)) {

i = follow\_hugetlb\_page(mm, vma, pages, vmas,&start, &nr\_pages, i,gup\_flags);

continue;

}

} /\*if结束，主要是查找VMA和标记有效性检查\*/

**retry:**

if (unlikely(fatal\_signal\_pending(current))) /\*检查进程是否有挂起的SIGKILL信号\*/

return i ? i : -ERESTARTSYS;

cond\_resched(); /\*条件重调度\*/

page = **follow\_page\_mask(vma, start, foll\_flags, &page\_mask)**;

/\*检查映射页是否存在，存在返回page指针，否则返回NULL，/mm/gup.c\*/

if (!page) { /\*如果映射页不存在，则建立映射\*/

int ret;

ret = **faultin\_page(tsk, vma, start, &foll\_flags,nonblocking)**; /\*建立页映射，/mm/gup.c\*/

switch (ret) {

**case 0:** /\*映射建立成功\*/

**goto retry**; /\*再次检查页映射情况\*/

case -EFAULT:

case -ENOMEM:

case -EHWPOISON:

return i ? i : ret;

case -EBUSY:

return i;

case -ENOENT:

**goto next\_page**; /\*跳至下一页\*/

}

BUG();

} /\*if 结束，处理映射页不存在结束\*/

/\*页映射已经建立，保存映射页page和vm\_area\_struct实例信息\*/

if (IS\_ERR(page))

return i ? i : PTR\_ERR(page);

if (pages) {

**pages[i] = page**; /\*关联到指针数组\*/

flush\_anon\_page(vma, page, start); /\*/arch/mips/include/asm/cacheflush.h\*/

flush\_dcache\_page(page); /\*/arch/mips/include/asm/cacheflush.h\*/

page\_mask = 0;

}

next\_page:

if (vmas) {

**vmas[i] = vma**; /\*关联VMA\*/

page\_mask = 0;

}

/\*循环控制\*/

**page\_increm = 1 + (~(start >> PAGE\_SHIFT) & page\_mask)**; /\*本次建立映射页数量\*/

if (page\_increm > nr\_pages)

page\_increm = nr\_pages;

**i += page\_increm**;

**start += page\_increm \* PAGE\_SIZE**; /\*增加地址\*/

**nr\_pages -= page\_increm**; /\*未建立映射页数量减少\*/

} while (nr\_pages); /\*遍历下一页，do\_while循环结束\*/

return i; /\*返回建立映射页数量\*/

}

\_\_get\_user\_pages()函数的执行流程比较清晰，follow\_page\_mask()函数用于检查虚拟页映射情况，是没有建立映射，映射到页帧，又或是页数据在交换区等。如果映射存在则对映射页帧page实例进行相应的操作后返回page指针，否则返回NULL。

如果虚拟页映射不存在，\_\_get\_user\_pages()函数将调用faultin\_page()函数模拟产生了一个用户空间缺页异常，在此函数内将调用handle\_mm\_fault()函数为虚拟页建立映射。执行完handle\_mm\_fault()函数后，还将调用follow\_page\_mask()函数获取映射页帧page实例。

下面分别介绍faultin\_page()和follow\_page\_mask()函数的实现。

##### ■获取映射页

follow\_page\_mask()函数用于检查虚拟页映射是否已经存在，若存在则由PTE页表项获取映射页帧page实例，并对page实例做相应的操作后返回page指针，否则返回NULL。

follow\_page\_mask()函数代码如下（/mm/gup.c）：

struct page \*follow\_page\_mask(struct vm\_area\_struct \*vma,unsigned long address, \

unsigned int flags,unsigned int \*page\_mask)

/\*address：页起始地址，flags：标记，指定映射需要具有的属性等，如**FOLL\_WRITE。**\*/

{

pgd\_t \*pgd;

pud\_t \*pud;

pmd\_t \*pmd;

spinlock\_t \*ptl;

struct page \*page;

struct mm\_struct \*mm = vma->vm\_mm;

\*page\_mask = 0;

... /\*处理巨型页\*/

pgd = pgd\_offset(mm, address); /\*PGD页表项指针\*/

if (pgd\_none(\*pgd) || unlikely(pgd\_bad(\*pgd)))

return no\_page\_table(vma, flags); /\*对应全局页表项为空，返回NULL，表示映射未创建\*/

pud = pud\_offset(pgd, address); /\*PUD页表项指针\*/

if (pud\_none(\*pud))

return no\_page\_table(vma, flags);

... /\*处理巨型页\*/

if (unlikely(pud\_bad(\*pud)))

return no\_page\_table(vma, flags);

pmd = pmd\_offset(pud, address); /\*PMD页表项指针\*/

if (pmd\_none(\*pmd))

return no\_page\_table(vma, flags);

... /\*处理巨型页\*/

if ((flags & FOLL\_NUMA) && pmd\_protnone(\*pmd))

return no\_page\_table(vma, flags);

... /\*巨型页\*/

return **follow\_page\_pte(vma, address, pmd, flags)**; /\*检查pte页表项，/mm/gup.c\*/

}

以上follow\_page\_mask()函数代码中忽略了对巨型页处理的代码后，函数结构就比较简单了。函数内依次判断虚拟地址address对应的的PGD、PUD、PMD页表项是否为空，若其中之一为空则表示映射还未建立，函数返回NULL。若以上三级页表项都存在，则最后调用follow\_page\_pte()函数检查PTE页表项的情况，函数定义如下（/mm/gup.c） ：

static struct page \*follow\_page\_pte(struct vm\_area\_struct \*vma,unsigned long address, \

pmd\_t \*pmd, unsigned int flags)

{

struct mm\_struct \*mm = vma->vm\_mm;

struct page \*page;

spinlock\_t \*ptl; /\*保护页表项自旋锁\*/

pte\_t \*ptep, pte; /\*PTE页表项指针，实例\*/

retry:

if (unlikely(pmd\_bad(\*pmd)))

return no\_page\_table(vma, flags);

**ptep = pte\_offset\_map\_lock(mm, pmd, address, &ptl)**; /\*address对应PTE页表项指针\*/

pte = \*ptep; /\*复制页表项内容至pte\*/

if (!**pte\_present(pte)**) { /\*映射页不在内存中（没有映射页或在交换区中）\*/

swp\_entry\_t entry;

if (likely(!(flags & FOLL\_MIGRATION)))

/\*不有设置FOLL\_MIGRATION标记，不跟踪交换区中页\*/

goto no\_page; /\*返回NULL\*/

if (**pte\_none(pte)**) /\*页表项为空，返回NULL\*/

goto no\_page;

**entry = pte\_to\_swp\_entry(pte)**; /\*不为空，页数据在交换区，/include/linux/swapops.h\*/

/\*将体系结构相关的pte实例转换成体系结构无关的swp\_entry\_t实例\*/

if (!is\_migration\_entry(entry)) /\*是否是正在合并的页\*/

goto no\_page;

pte\_unmap\_unlock(ptep, ptl);

migration\_entry\_wait(mm, pmd, address);

goto retry;

}

/\*映射到页帧\*/

if ((flags & FOLL\_NUMA) && pte\_protnone(pte))

goto no\_page;

/\*指定页需要写权限\*/

if (**(flags & FOLL\_WRITE) && !pte\_write(pte)**) { /\*pte没有写权限，返回NULL\*/

pte\_unmap\_unlock(ptep, ptl);

return NULL; /\*返回NULL\*/

}

page = **vm\_normal\_page(vma, address, pte)**;

/\*检查是否是普通映射页，是则返回page指针，/mm/memory.c\*/

if (unlikely(!page)) { /\*PFN映射页或映射全零页帧时（体系结构分配的）\*/

if ((flags & FOLL\_DUMP) || !is\_zero\_pfn(pte\_pfn(pte)))

goto bad\_page;

page = pte\_page(pte); /\*pte转page（PFN映射）\*/

}

/\*映射页帧具有page实例\*/

if (flags & **FOLL\_GET**)

**get\_page\_foll(page)**; /\*增加page实例\_count引用计数，/mm/internal.h\*/

if (flags & **FOLL\_TOUCH**) { /\*标记页被访问过\*/

if ((flags & FOLL\_WRITE) &&!pte\_dirty(pte) && !PageDirty(page))

**set\_page\_dirty(page)**; /\*设置页脏，需在回写\*/

**mark\_page\_accessed(page)**; /\*标记页被访问过，修改页标记及LRU链表等，/mm/swap.c\*/

}

/\*默认设置了FOLL\_POPULATE标记位，锁定VMA映射页添加到LRU链表\*/

if **((flags & FOLL\_POPULATE) && (vma->vm\_flags & VM\_LOCKED))** {

if (page->mapping && trylock\_page(page)) {

**lru\_add\_drain()**; /\*将页添加到LRU链表，/mm/swap.c\*/

**mlock\_vma\_page(page)**; /\*/mm/mlock.c\*/

/\*设置page实例PG\_mlocked标记，如果未插入不可回收链表，则添加至链表\*/

unlock\_page(page);

}

}

pte\_unmap\_unlock(ptep, ptl);

return **page;**  /\*返回page实例指针\*/

...

}

follow\_page\_pte()函数通过检查pte页表项内容，确定映射页状态。如果pte页表项内容为空，则函数直接返回NULL。页表项内容不为空，则映射页可能映射到普通的页帧，页数据可能在交换区中，也可能映射到IO内存等。

如果具有映射，则还需要检查页表项写权限是否与标记参数匹配，不匹配也将返回NULL。如果映射页在内存中且写权限也匹配，则调用vm\_normal\_page()函数检查映射页是否是普通内存页（由page实例管理的页），是则返回page实例指针。如果不是普通映射页，可能是PFN映射页等，如果需要也由PFN转换成page实例。然后，还需要根据flags参数对page实例进行相应的操作（修改标记位，添加到LRU链表），最后返回page实例指针。

vm\_normal\_page()用于检查pte页表项映射页是否是普通内存页，映射页可分为两种，一种是普通的由page实例管理的物理内存页，第二种是特殊页，如映射IO内存的页，PFN管理的页等。对于普通映射页vm\_normal\_page()函数返回page实例指针，特殊页返回NULL。

vm\_normal\_page()函数定义如下（/mm/memory.c）：

struct page \*vm\_normal\_page(struct vm\_area\_struct \*vma, unsigned long addr,pte\_t pte)

{

unsigned long **pfn = pte\_pfn(pte)**; /\*页表项转物理页帧号，/arch/mips/include/asm/pgtable-32.h\*/

if (HAVE\_PTE\_SPECIAL) { /\*处理特殊页表项，MIPS32未定义\*/

...

}

if (unlikely(vma->vm\_flags & (VM\_PFNMAP|VM\_MIXEDMAP))) { /\*PFN或混合映射VMA\*/

if (vma->vm\_flags & VM\_MIXEDMAP) { /\*混合映射VMA，地址有效，返回page指针\*/

if (!pfn\_valid(pfn))

return NULL;

goto out;

} else { /\*PFN映射VMA\*/

unsigned long off;

off = (addr - vma->vm\_start) >> PAGE\_SHIFT;

if (pfn == vma->vm\_pgoff + off) /\*直接映射，虚拟地址与物理地址相同\*/

return NULL;

if (!**is\_cow\_mapping(vma->vm\_flags)**)

/\*PFN映射（写时复制VMA）仍需返回page实例\*/

return NULL;

}

}

if (**is\_zero\_pfn(pfn**)) /\*映射全零页帧，返回NULL，/include/asm-generic/pgtable.h\*/

return NULL;

check\_pfn:

if (unlikely(pfn > highest\_memmap\_pfn)) {

print\_bad\_pte(vma, addr, pte, NULL);

return NULL;

}

**out:**

return **pfn\_to\_page(pfn)**; /\*PFN转page实例指针，/include/asm-generic/memory\_model.h\*/

}

对于映射到普通物理内存的页，以及写时复制形式的PFN映射页，返回对应的page实例指针，其余返回NULL。

##### ■建立页映射

\_\_get\_user\_pages()函数中对虚拟页先调用follow\_page\_mask()函数检查映射情况，如果尚未建立映射则调用faultin\_page()函数为虚拟页建立映射，函数定义如下（/mm/gup.c）：

static int faultin\_page(struct task\_struct \*tsk, struct vm\_area\_struct \*vma,unsigned long address, \

unsigned int \*flags, int \*nonblocking)

{

struct mm\_struct \*mm = vma->vm\_mm; /\*地址空间\*/

unsigned int **fault\_flags = 0**; /\*handle\_mm\_fault()函数的flags标记参数，缺页异常标记\*/

int ret;

/\*跳过栈保护页，populate\_vma\_page\_range()函数设置了flags的FOLL\_POPULATE标记位\*/

if ((\*flags & **FOLL\_POPULATE**) &&(stack\_guard\_page\_start(vma, address) || \

stack\_guard\_page\_end(vma, address + PAGE\_SIZE)))

return -ENOENT;

if (\*flags & FOLL\_WRITE) /\*设置缺页异常标记\*/

fault\_flags |= FAULT\_FLAG\_WRITE; /\*模拟写操作异常\*/

if (**nonblocking**) /\*等待磁盘IO\*/

fault\_flags |= FAULT\_FLAG\_ALLOW\_RETRY; /\*缺页异常处理中允许重试\*/

if (\*flags & FOLL\_NOWAIT)

fault\_flags |= FAULT\_FLAG\_ALLOW\_RETRY | FAULT\_FLAG\_RETRY\_NOWAIT;

if (\*flags & FOLL\_TRIED) {

VM\_WARN\_ON\_ONCE(fault\_flags & FAULT\_FLAG\_ALLOW\_RETRY);

fault\_flags |= FAULT\_FLAG\_TRIED;

}

ret = **handle\_mm\_fault**(mm, vma, address, fault\_flags); /\*用户空间缺页处理函数，/mm/memory.c\*/

/\*建立映射不成功时，设置相应的错误码，返回给调用函数\*/

if (ret & VM\_FAULT\_ERROR) {

if (ret & VM\_FAULT\_OOM)

return -ENOMEM;

if (ret & (VM\_FAULT\_HWPOISON | VM\_FAULT\_HWPOISON\_LARGE))

return \*flags & FOLL\_HWPOISON ? -EHWPOISON : -EFAULT;

if (ret & (VM\_FAULT\_SIGBUS | VM\_FAULT\_SIGSEGV))

return -EFAULT;

BUG();

}

if (tsk) {

if (ret & VM\_FAULT\_MAJOR)

tsk->maj\_flt++;

else

tsk->min\_flt++;

}

if (ret & VM\_FAULT\_RETRY) {

if (nonblocking)

\*nonblocking = 0;

return -EBUSY;

}

if ((ret & VM\_FAULT\_WRITE) && !(vma->vm\_flags & VM\_WRITE))

\*flags &= ~FOLL\_WRITE;

**return 0**; /\*建立映射成功，返回0\*/

}

faultin\_page()函数将\_\_get\_user\_pages()函数传递下来的标记参数，转换成缺页处理函数的fault\_flags标记参数，调用用户空间缺页处理函数**handle\_mm\_fault()**，模拟产生了一个用户空间缺页异常，在缺页处理函数中为虚拟页分配页帧，修改页表项，建立映射。建立映射成功，faultin\_page()函数返回0，不成功则设置相应的错误码返回给调用函数。

handle\_mm\_fault()函数需要处理的情况比较多，例如虚拟页是否位于栈区域，是匿名映射还是文件映射等，又或者是写保护页等，后面介绍缺页异常处理程序时，再详细介绍此函数的实现。

在创建映射时，如果地址空间默认VMA标记（def\_flags成员值）没有设置VM\_LOCKED标记位，并且创建映射系统调用中没有设置MAP\_POPULATE或MAP\_LOCKED标记位，则创建映射时，只是创建映射区域对应的VMA实例，而没有分配页帧和修改页表项，没有建立映射。当CPU访问到没有建立映射的地址（或访问权限不匹配）时，将触发CPU异常，称为缺页异常。在异常处理程序中，如果是访问用户空间地址引发的异常，将调用handle\_mm\_fault()函数处理，在此函数中建立页映射后，从异常返回，随后CPU就要以正常地访问引发异常的地址了。

在\_\_get\_user\_pages()函数中，调用faultin\_page()函数建立映射后，将再次调用follow\_page\_mask()函数检查映射页情况，获取映射页page实例，follow\_page\_mask()函数返回后\_\_get\_user\_pages()函数将处理下一虚拟页。

### 4.6.3堆的管理

内核在创建进程，加载新可执行目标文件时，将为进程创建初始的堆，包括对应的VMA实例和反向映射结构。

地址空间mm\_struct结构体中包含了堆的起始地址和当前结束地址，相关成员如下所示：

mm\_struct {

...

unsigned long  **start\_brk, brk**, start\_stack;

...

}

start\_brk成员表示堆的起始地址，运行新可执行目标文件时设置，之后不再改变，brk成员表示当前堆的结束地址。用户进程可通过brk()系统调用改变brk值，即可以向上扩展堆或收缩堆。

用户程序一般不会直接调用brk()系统调用，而是通过malloc()库函数间接地管理堆。malloc()库函数用于为进程动态分配/释放（小块）内存，类似于内核使用的slab分配器。

malloc()库函数将堆划分成小块区域，并管理其使用情况，如下图所示。



用户进程调用malloc()库函数分配内存时，将会返回获取空闲内存起始地址。空闲内存不足时，库函数将调用brk()系统调用扩展堆，也可以通过brk()系统调用收缩堆。malloc()库函数将堆划分成小块内存进行管理的方式由其自身实现，与内核无关。brk()系统调用只负责建立映射，不负责管理。

brk()系统调用在扩展堆时，一般只是创建/修改VMA实例数据，而没有分配物理页帧，没有修改页表项建立映射，除非地址空间默认VMA标记def\_flags成员设置了VM\_LOCKED标记位。如果没有设置此标记位，则在CPU访问到未映射的虚拟内存时，在缺页异常处理程序中按页建立映射，详见下一节。

brk()系统调用在/mm/mmap.c文件内实现，唯一的参数是堆新的结束地址：

SYSCALL\_DEFINE1(brk, unsigned long, brk)

/\*brk：小于mm->start\_brk时，返回当前堆结束地址，例如，用0参数获取当前堆顶位置\*/

{

unsigned long retval;

unsigned long **newbrk, oldbrk**; /\*堆的新旧结束地址\*/

struct mm\_struct \*mm = current->mm;

unsigned long min\_brk;

bool populate;

down\_write(&mm->mmap\_sem); /\*获取信号量\*/

#ifdef CONFIG\_COMPAT\_BRK

if (current->brk\_randomized)

min\_brk = mm->start\_brk;

else

min\_brk = mm->end\_data;

#else

**min\_brk = mm->start\_brk**; /\*堆起始地址\*/

#endif

if (**brk < min\_brk**) /\*跳转至out处，返回当前堆结束地址\*/

goto out;

if (check\_data\_rlimit(rlimit(RLIMIT\_DATA), brk, mm->start\_brk,mm->end\_data, mm->start\_data))

goto out; /\*限制值检查\*/

newbrk = PAGE\_ALIGN(brk); /\*新结束地址页对齐\*/

oldbrk = PAGE\_ALIGN(mm->brk); /\*旧结束地址页对齐，mm->brk保存堆当前结束地址\*/

if (oldbrk == newbrk) /\*新旧结束地址相等，不需要对堆进行收缩/扩展操作\*/

goto set\_brk;

if (brk <= mm->brk) { /\*新结束地址更小，收缩堆\*/

if (!**do\_munmap(mm, newbrk, oldbrk-newbrk)**)

/\*若新堆结束地址在堆现有结束地址之下，则解除部分映射，后面将介绍\*/

goto set\_brk;

goto out;

}

if (find\_vma\_intersection(mm, oldbrk, newbrk+PAGE\_SIZE))

goto out; /\*如果扩展的新区域被与现有内存域有重叠则跳转至out\*/

if (**do\_brk(oldbrk, newbrk-oldbrk)** != oldbrk)

/\*创建/扩展VMA实例，oldbrk为新VMA起始地址，/mm/mmap.c\*/

goto out;

set\_brk:

mm->brk = brk; /\*堆新的结束地址\*/

populate = **newbrk > oldbrk** && (**mm->def\_flags & VM\_LOCKED**) != 0;

up\_write(&mm->mmap\_sem);

if (**populate**) /\*扩展堆且VMA设置了VM\_LOCKED标记位\*/

**mm\_populate(oldbrk, newbrk - oldbrk)**;

/\*立即为扩展区域建立映射，见上文\*/

return **brk**; /\*返回堆结束地址\*/

out:

retval = mm->brk; /\*堆当前结束地址\*/

up\_write(&mm->mmap\_sem);

return retval; /\*返回堆当前结束地址\*/

}

brk系统调用判断新设置的堆结束地址与当前堆结束地址的值，如果前者大，则是扩展堆，否则是收缩堆。但是，如果新堆结束地址小于堆起始地址，则直接返回堆当前结束地址。

如果是收缩堆，则调用do\_munmap()函数解除收缩区域的映射。如果是扩展堆，则调用do\_brk()函数，为扩展区域申请未映射区域（起始地址指定为堆当前结束地址），并判断新区域能否与相邻的VMA合并，如果能合并则合并，不能合并则为新区域创建新的vm\_area\_struct实例，并插入到地址空间管理结构中。最后，如果地址空间mm->def\_flags 成员设置了 VM\_LOCKED标记位，则需要立即为扩展区域分配物理内存建立映射，这项工作由mm\_populate()函数完成。

请读者记住解除区域映射的**do\_munmap()**函数，在后面介绍解除映射的munmap()系统调用时再做介绍。

下面看一下do\_brk()函数的实现（/mm/mmap.c）：

static unsigned long do\_brk(unsigned long addr, unsigned long len)

/\*addr：扩展区域起始地址，为oldbrk；len：扩展区域长度newbrk-oldbrk（所有地址页对齐）\*/

{

struct mm\_struct \*mm = current->mm;

struct vm\_area\_struct \*vma, \*prev;

unsigned long flags;

struct rb\_node \*\*rb\_link, \*rb\_parent;

pgoff\_t **pgoff = addr >> PAGE\_SHIFT**; /\*扩展区域起始虚拟页帧号\*/

int error;

len = PAGE\_ALIGN(len);

if (!len) /\*长度为0，返回addr\*/

return addr;

flags = **VM\_DATA\_DEFAULT\_FLAGS | VM\_ACCOUNT | mm->def\_flags**; /\*VMA标记\*/

error = **get\_unmapped\_area(NULL, addr, len, 0, MAP\_FIXED)**;

/\*在指定地址获取未映射区域，匿名映射\*/

if (error & ~PAGE\_MASK)

return error;

error = mlock\_future\_check(mm, mm->def\_flags, len);

/\*检查VM\_LOCKED标记位是否有效，是否可锁定VMA，/mm/mmap.c\*/

if (error)

return error;

verify\_mm\_writelocked(mm); /\*需选择DEBUG\_VM配置选项，/mm/mmap.c\*/

while (**find\_vma\_links(mm, addr, addr + len, &prev, &rb\_link, &rb\_parent)**) { /\*查找VMA\*/

if (do\_munmap(mm, addr, len)) /\*解除旧的映射\*/

return -ENOMEM;

}

if (!may\_expand\_vm(mm, len >> PAGE\_SHIFT)) /\*判断扩展VMA有效性，/mm/mmap.c\*/

return -ENOMEM;

if (mm->map\_count > sysctl\_max\_map\_count)

return -ENOMEM;

if (security\_vm\_enough\_memory\_mm(mm, len >> PAGE\_SHIFT))

return -ENOMEM;

vma = **vma\_merge(mm, prev, addr, addr + len, flags,NULL, NULL, pgoff, NULL)**;

/\*合并VMA\*/

if (vma) /\*可以合并，跳至out\*/

goto out;

/\*扩展区域不能与现有VMA合并\*/

**vma = kmem\_cache\_zalloc(vm\_area\_cachep, GFP\_KERNEL)**; /\*创建vm\_area\_struct实例\*/

...

INIT\_LIST\_HEAD(&vma->anon\_vma\_chain); /\*初始化anon\_vma\_chain双链表\*/

vma->vm\_mm = mm /\*设置VMA\*/

vma->vm\_start = addr;

vma->vm\_end = addr + len;

**vma->vm\_pgoff = pgoff**; /\*映射区起始虚拟页帧号\*/

**vma->vm\_flags = flags**;

vma->**vm\_page\_prot** = **vm\_get\_page\_prot(flags)**; /\*VMA标记转页访问权限，/mm/mmap.c\*/

**vma\_link(mm, vma, prev, rb\_link, rb\_parent)**;

/\*VMA插入地址空间管理结构和文件反向映射结构中\*/

out:

perf\_event\_mmap(vma);

mm->total\_vm += len >> PAGE\_SHIFT;

if (**flags & VM\_LOCKED**)

mm->locked\_vm += (len >> PAGE\_SHIFT);

**vma->vm\_flags |= VM\_SOFTDIRTY**;

return addr;

}

do\_brk()函数实际上是要为进程虚拟内存[addr,addr+len)区域创建VMA实例。函数首先在进程地址空间中以固定起始地址的方式申请未映射区域；然后在地址空间（扩展）红黑树中查找新区域的插入点（父节点及前节点），如果新区域与现有内存域重叠，则解除[addr,addr+len)区域的映射；随后调用VMA合并函数检查新区域能否与现有VMA合并，能则合并，函数返回；如果不能合并，则为新区域创建VMA实例，并初始化，最后将实例插入地址空间管理结构和文件映射反向结构中（如果是文件映射），匿名反向映射数据结构初始化为空。

brk系统调用用于扩展或收缩堆，如果是收缩堆则立即解除收缩区的映射。如果是扩展堆则为扩展区建立vm\_area\_struct实例（或与现有内存域合并），如果VMA标记成员设置了VM\_LOCKED标记位，则立即为扩展区域分配物理内存建立映射，否则将不建立映射，直至进程访问VMA地址产生缺页异常时，在异常处理程序中逐页建立映射。

### 4.6.4内存映射区

内存映射区应该是用户进程使用最多的区域了，用于进程创建文件映射和匿名映射。本小节介绍在内存映射区创建映射的系统调用。

#### 1线性映射

进程地址空间中的内存映射区可用于文件映射和匿名映射。文件映射是将块设备中的文件内容映射到进程地址空间，从而进程可以通过对内存的操作来完成对文件内容的操作。匿名映射通常用于创建大块物理内存映射，小块内存可直接通过malloc()函数分配，匿名映射页还可以用于进程间共享或传递信息。

下图以文件映射为例，示意了文件映射的结构：



文件映射的物理页帧由被称为文件地址空间的address\_space结构体管理。address\_space结构体中通过基数树管理映射页帧的page实例，基数树管理的是文件内容在内存中的缓存，按页进行，因此称为页缓存。

建立VMA到文件内容的映射，就是修改VMA对应页表项，使其映射到文件内容在页缓存中的页。访问文件内容转换成对映射内存页的访问。

address\_space结构体中的地址空间操作结构address\_space\_operations负责建立文件内容缓存页与块设备中的文件内容之间的映射关系，完成两者之间的数据传输。

文件映射内存域vm\_area\_struct实例vm\_file成员指向映射文件file实例（进程文件表示），file结构体具有指向inode结构体实例（文件在内核中的唯一表示）的指针成员，inode结构体和file结构体中具有指向文件地址空间address\_space结构体的指针成员。文件映射vm\_area\_struct实例插入到address\_space结构体管理的区间树内，用于构成反向映射结构（哪些内存域映射了文件内容）。文件系统相关内容请参考第7章，文件地址空间及页缓存相关内容请参考第11章。

匿名映射VMA将关联反向映射anon\_vma结构体实例（第一次建立映射时创建或复制于父进程），匿名映射页没有映射到块设备，数据只存在于内存中。

用户进程可通过mmap()/mmap2()系统调用将文件内容映射到进程地址空间或创建匿名映射。MIPS32体系结构这2个系统调用编号定义如下（/arch/mips/include/uapi/asm/unistd.h）：

#define \_\_NR\_mmap (\_\_NR\_Linux + 90) /\*4090，mmap()系统调用编号\*/

...

#define \_\_NR\_mmap2 (\_\_NR\_Linux + 210) /\*4210\*/

mmap()/mmap2()系统调用实现函数名称（指针）如下（/arch/mips/kernel/scall32-o32.S）：

EXPORT(sys\_call\_table) /\*系统调用实现函数列表\*/

PTR sys\_syscall /\* 4000 \*/

PTR sys\_exit

PTR \_\_sys\_fork

PTR sys\_read

PTR sys\_write

PTR sys\_open /\* 4005 \*/

PTR sys\_close

...

PTR **sys\_mips\_mmap** /\* **4090**，mmap()系统调用实现函数，由编号定位列表\*/

...

PTR sys\_mips\_mmap2 /\* 4210 ，mmap2()系统调用实现函数\*/

...

mmap()/mmap2()系统调用实现函数定义在/arch/mips/kernel/syscall.c文件内，下面将以mmap()系统调用为例说明其实现，实现函数为**sys\_mips\_mmap()**。mmap()系统调用的实现类似，只不过偏移量参数是页偏移量，而不是字节数。

mmap()系统调用中prot和flags参数用于指示映射页的访问类型和映射的属性，这两个参数取值定义在/arch/mips/include/uapi/asm/mman.h头文件：

/\*访问权限prot参数\*/

#define PROT\_NONE 0x00 /\*页不可以被访问\*/

#define PROT\_READ 0x01 /\*可读\*/

#define PROT\_WRITE 0x02 /\*可写\*/

#define PROT\_EXEC 0x04 /\*可执行\*/

#define PROT\_SEM 0x10 /\*页可用于原子操作\*/

#define PROT\_GROWSDOWN 0x01000000

/\*mprotect()系统调用标记，权限更改扩展至向下生长的VMA起始位置\*/

#define PROT\_GROWSUP 0x02000000

/\*mprotect()系统调用标记，权限更改扩展至向上生长的VMA结束位置\*/

/\*映射内存域标记flags参数\*/

#define MAP\_SHARED 0x001 /\*共享映射\*/

#define MAP\_PRIVATE 0x002 /\*私有映射\*/

#define MAP\_TYPE 0x00f /\*映射类型掩码，映射类型必须二选一\*/

#define MAP\_FIXED 0x010 /\*固定地址映射\*/

... /\*linux不使用的标记\*/

/\*linux特有标记\*/

#define MAP\_NORESERVE 0x0400 /\* don't check for reservations \*/

#define **MAP\_ANONYMOUS** 0x0800 /\*匿名映射\*/

#define MAP\_GROWSDOWN 0x1000 /\*VMA向下生长，如栈\*/

#define MAP\_DENYWRITE 0x2000 /\* ETXTBSY \*/

#define MAP\_EXECUTABLE 0x4000 /\*标记为可执行VMA\*/

#define **MAP\_LOCKED** 0x8000 /\*锁定内存域，创建时立即建立映射\*/

#define **MAP\_POPULATE** 0x10000 /\*立即分配物理内存，建立映射\*/

#define MAP\_NONBLOCK 0x20000 /\* do not block on IO \*/

#define MAP\_STACK 0x40000 /\*为进程/线程栈给出一个最合适地址\*/

#define MAP\_HUGETLB 0x80000 /\*创建巨型TLB页映射\*/

共享映射指多个进程可以共同映射同一页，任一进程对映射页帧的改变对其它进程可见，修改后的内容会同步到块设备文件中（如果是文件映射）。

私有映射指创建一个写时复制的映射，当进程对页进行写操作时，则复制页面内容到一个新页，修改页表项指向新页，进程对页面的写操作对其它进程不可见。文件映射修改内容不会同步到块设备。

匿名映射中的共享映射通常用于进程间共享内存，私有映射通常用于分配大块内存。

mmap()系统调用实现函数定义如下（/arch/mips/kernel/syscall.c）：

SYSCALL\_DEFINE6(**mips\_mmap**, unsigned long, addr, unsigned long, len,

unsigned long, prot, unsigned long, flags, unsigned long,fd, off\_t, offset)

/\*定义的函数名称为sys\_mips\_mmap()，

\*addr：映射区域起始虚拟地址，通常设为0，由内核确定；

\*len：映射区域长度，字节数；

\*port：映射页访问权限；

\*flags：映射属性，如共享映射、私有映射等；

\***fd**：映射文件描述符，先要打开文件，获取描述符；

　　\***offset**：文件内容偏移量，字节数，mmap2()系统调用为页偏移量。

\*/

{

unsigned long result;

result = -EINVAL;

if (**offset & ~PAGE\_MASK**) /\*偏移量offset必须页对齐，后面要转为页偏移量\*/

goto out;

result = **sys\_mmap\_pgoff**(addr, len, prot, flags, fd, **offset >> PAGE\_SHIFT**);

/\*mmap\_pgoff()系统调用实现函数\*/

out:

return result;

}

sys\_mips\_mmap()函数调用mmap\_pgoff()系统调用的实现函数，定义在/mm/mmap.c文件，代码如下：

SYSCALL\_DEFINE6(mmap\_pgoff, unsigned long, addr, unsigned long, len, \

unsigned long, prot, unsigned long, flags,unsigned long, fd, unsigned long, pgoff)

/\*pgoff：页偏移量\*/

{

struct file \*file = NULL;

unsigned long retval = -EBADF;

if (!(flags & MAP\_ANONYMOUS)) { /\*不是匿名映射，即文件映射\*/

audit\_mmap\_fd(fd, flags);

**file = fget(fd)**; /\*由进程文件描述符获取文件file实例指针\*/

if (!file)

goto out;

... /\*处理巨型页情况\*/

}

else if (flags & MAP\_HUGETLB) { /\*匿名映射，且是巨型页\*/

...

}

flags &= ~(MAP\_EXECUTABLE | MAP\_DENYWRITE); /\*屏蔽可执行和不可写标记位\*/

retval = **vm\_mmap\_pgoff(file, addr, len, prot, flags, pgoff)**;

/\*创建映射，返回映射起始虚地址，/mm/util.c\*/

out\_fput:

if (file)

fput(file);

out:

return retval; /\*返回映射区域起始虚地址\*/

}

用户程序如果是要创建匿名映射则flags成员必须设置MAP\_ANONYMOUS标记位，如果是要创建文件映射则不能设置MAP\_ANONYMOUS标记位，fd表示进程文件描述符。

系统调用内首先判断是否是文件映射，如果是则由文件描述符fd获取file实例，然后对巨型TLB页进行处理，这里我们忽略它。随后，屏蔽flags标记的可执行和不可写标记位，最后将实际创建映射工作委托给vm\_mmap\_pgoff(file, addr, len, prot, flags, pgoff)函数，函数在/mm/util.c文件内实现：

unsigned long vm\_mmap\_pgoff(struct file \*file, unsigned long addr,unsigned long len, unsigned long prot, \

unsigned long flag, unsigned long pgoff)

{

unsigned long ret;

struct mm\_struct \*mm = current->mm;

unsigned long populate; /\*需要立即建立映射时，保存映射区长度\*/

ret = security\_mmap\_file(file, prot, flag);

if (!ret) {

down\_write(&mm->mmap\_sem);

ret = **do\_mmap\_pgoff(file, addr, len, prot, flag, pgoff, &populate)**; /\*/mm/mmap.c\*/

up\_write(&mm->mmap\_sem);

**if (populate)**  /\*如果需要立即建立映射则调用mm\_populate()函数\*/

**mm\_populate(ret, populate)**; /\*建立映射函数，见上文，/mm/gup.c\*/

}

return ret;

}

vm\_mmap\_pgoff()函数调用do\_mmap\_pgoff()函数在地址空间中确定映射区域位置，创建vm\_area\_struct实例（或与现有VMA合并）并初始化，并将VMA实例插入到进程地址空间管理结构中，如果是文件映射还需要将vm\_area\_struct实例插入address\_space实例区间树中。

如果VMA标记成员设置了VM\_LOCKED标记位，或者flags参数设置了MAP\_POPULATE标记位且没有设置 MAP\_NONBLOCK标记位，则populate指向的整数赋值映射区域的长度值（字节数）。最后，若populate参数值非零，调用mm\_populate()函数立即为映射区域分配物理页帧建立映射。

mm\_populate()函数在前面介绍过了，这里主要看一下do\_mmap\_pgoff()函数的实现，函数调用关系如下图所示：



do\_mmap\_pgoff()函数首先调用get\_unmapped\_area()函数获取未映射区域，如果是文件映射则文件操作结构mmap函数指针不能为空，否则返回错误码。然后，调用mmap\_region()函数完成映射内存域数据结构的处理：如果申请的未映射区域被已有内存域覆盖，则解除覆盖区域映射，判断未映射区域能否与现有内存域合并，能则合并，不能合并则创建新VMA实例并初始化。需要注意的是，如果是文件映射，创建VMA实例后还需要调用文件操作结构的mmap()函数，通常是给vm\_area\_struct实例vm\_ops成员赋值。 如果是共享匿名映射则调用shmem\_zero\_setup(vma)函数创建共享匿名映射。do\_mmap\_pgoff()函数最后根据VMA标记及flags参数判断是否需要对\*populate赋值，以确定是否立即分配物理页帧建立映射。

do\_mmap\_pgoff()函数定义在/mm/mmap.c文件内，代码如下（/mm/mmap.c）：

unsigned long do\_mmap\_pgoff(struct file \*file, unsigned long addr,unsigned long len, unsigned long prot, \

unsigned long flags, unsigned long pgoff,unsigned long \*populate)

{

struct mm\_struct \*mm = current->mm;

vm\_flags\_t vm\_flags;

\*populate = 0;

if (!len)

return -EINVAL;

if ((prot & PROT\_READ) && (current->personality & READ\_IMPLIES\_EXEC))

if (!(file && (file->f\_path.mnt->mnt\_flags & MNT\_NOEXEC)))

prot |= PROT\_EXEC;

if (!(flags & MAP\_FIXED)) /\*未设置MAP\_FIXED标记位\*/

addr = round\_hint\_to\_min(addr); /\*指定地址不能在内存映射区基地址之下，/mm/mmap.c\*/

len = PAGE\_ALIGN(len); /\*映射区长度页对齐\*/

if (!len)

return -ENOMEM;

if ((pgoff + (len >> PAGE\_SHIFT)) < pgoff)

return -EOVERFLOW;

if (mm->map\_count > sysctl\_max\_map\_count)

return -ENOMEM; /\*以上是参数检查\*/

**addr = get\_unmapped\_area(file, addr, len, pgoff, flags);** /\*获取未映射区域，返回起始虚拟地址\*/

if (addr & ~PAGE\_MASK) /\*如果起始地址没有页对齐，不能创建映射\*/

return addr;

vm\_flags = **calc\_vm\_prot\_bits(prot) | calc\_vm\_flag\_bits(flags) |**

**mm->def\_flags | VM\_MAYREAD | VM\_MAYWRITE | VM\_MAYEXEC**;

/\*由prot、flags等参数设置VMA标记\*/

if (flags & MAP\_LOCKED) /\*检查是否可以执行锁定操作（有没有超过限制值）\*/

if (!can\_do\_mlock())

return -EPERM;

if (mlock\_future\_check(mm, vm\_flags, len))

return -EAGAIN;

**if (file)** { /\*文件映射的处理，检查/设置VMA标记\*/

struct inode \*inode = file\_inode(file); /\*表示文件的inode实例\*/

switch (flags & MAP\_TYPE) {

case **MAP\_SHARED**: /\*文件共享映射\*/

if ((prot&PROT\_WRITE) && !(file->f\_mode&FMODE\_WRITE))

return -EACCES;

if (IS\_APPEND(inode) && (file->f\_mode & FMODE\_WRITE))

return -EACCES;

if (locks\_verify\_locked(file))

return -EAGAIN;

**vm\_flags |= VM\_SHARED | VM\_MAYSHARE**;

if (!(file->f\_mode & FMODE\_WRITE))

vm\_flags &= ~(VM\_MAYWRITE | VM\_SHARED);

case **MAP\_PRIVATE**: /\*文件私有映射\*/

if (!(file->f\_mode & FMODE\_READ))

return -EACCES;

if (file->f\_path.mnt->mnt\_flags & MNT\_NOEXEC) {

if (vm\_flags & VM\_EXEC)

return -EPERM;

vm\_flags &= ~VM\_MAYEXEC;

}

**if (!file->f\_op->mmap)** /\*文件操作函数结构必须定义mmap()函数指针\*/

return -ENODEV;

if (vm\_flags & (VM\_GROWSDOWN|VM\_GROWSUP))

return -EINVAL;

break;

default:

return -EINVAL;

}

}

**else** { /\*匿名映射的处理，检查/设置VMA标记\*/

switch (flags & MAP\_TYPE) {

case **MAP\_SHARED**: /\*匿名共享映射\*/

if (vm\_flags & (VM\_GROWSDOWN|VM\_GROWSUP))

return -EINVAL;

**pgoff = 0**; /\*VMA偏移量设为0\*/

**vm\_flags |= VM\_SHARED | VM\_MAYSHARE**;

break;

case **MAP\_PRIVATE**: /\*匿名私有映射\*/

**pgoff = addr >> PAGE\_SHIFT**; /\*VMA偏移量设为起始虚拟页帧号\*/

break;

default:

return -EINVAL;

}

}

if (flags & MAP\_NORESERVE) {

if (sysctl\_overcommit\_memory != OVERCOMMIT\_NEVER)

vm\_flags |= VM\_NORESERVE;

if (file && is\_file\_hugepages(file))

vm\_flags |= VM\_NORESERVE;

}

addr = **mmap\_region(file, addr, len, vm\_flags, pgoff)**;

/\*处理VMA实例，新创建或合并现有VMA，/mm/mmap.c\*/

if (!IS\_ERR\_VALUE(addr) && (**(vm\_flags & VM\_LOCKED)** || \

(flags & (**MAP\_POPULATE** | MAP\_NONBLOCK)) == MAP\_POPULATE))

/\*VMA设置了VM\_LOCKED标记位，

\*或flags参数设置了MAP\_POPULATE标记位且没有设置 MAP\_NONBLOCK标记位。

\*/

**\*populate = len**; /\*赋值映射区域长度\*/

return addr; /\*返回映射区起始地址\*/

}

do\_mmap\_pgoff()函数内调用get\_unmapped\_area()函数获取未映射区域，根据映射类型设置内存域标记，然后调用函数mmap\_region()创建并设置映射区域vm\_area\_struct实例或者与现有VMA合并。如果需要立即建立映射则将映射区域长度值赋予参数populate参数指向的整数，最后返回映射区域起始虚拟地址。

匿名共享映射VMA偏移量设为0，匿名私有映射VMA偏移量设为起始虚拟页帧号。

mmap\_region()函数在/mm/mmap.c文件内实现，代码如下：

unsigned long mmap\_region(struct file \*file, unsigned long addr,unsigned long len, vm\_flags\_t vm\_flags, \

unsigned long pgoff)

{

struct mm\_struct \*mm = current->mm;

struct vm\_area\_struct \*vma, \*prev;

int error;

struct rb\_node \*\*rb\_link, \*rb\_parent;

unsigned long charged = 0;

/\*进程能否扩展内存域（没超过资源限制），/mm/mmap.c\*/

if (!may\_expand\_vm(mm, len >> PAGE\_SHIFT)) {

unsigned long nr\_pages;

if (!(vm\_flags & MAP\_FIXED))

return -ENOMEM;

nr\_pages = count\_vma\_pages\_range(mm, addr, addr + len); /\*映射区域页数量\*/

if (!may\_expand\_vm(mm, (len >> PAGE\_SHIFT) - nr\_pages))

return -ENOMEM;

}

error = -ENOMEM;

while (**find\_vma\_links(mm, addr, addr + len, &prev, &rb\_link,&rb\_parent)**) {

if (**do\_munmap(mm, addr, len)**) /\*如果映射区域与现有VMA有重叠，则解除已有映射\*/

return -ENOMEM;

}

if (accountable\_mapping(file, vm\_flags)) { /\*是否是私有且可写映射，/mm/mmap.c\*/

charged = len >> PAGE\_SHIFT;

if (security\_vm\_enough\_memory\_mm(mm, charged)) /\*/include/linux/security.h\*/

return -ENOMEM;

**vm\_flags |= VM\_ACCOUNT**;

}

vma = **vma\_merge(mm, prev, addr, addr + len, vm\_flags, NULL, file, pgoff,NULL)**;

/\*新映射区域能否与现有VMA合并，能则合并\*/

if (vma)

goto out; /\*能合并，跳转至out处\*/

vma = **kmem\_cache\_zalloc(vm\_area\_cachep, GFP\_KERNEL);**

/\*不能合并，创建VMA实例\*/

...

/\*设置VMA实例\*/

vma->vm\_mm = mm;

vma->vm\_start = addr;

vma->vm\_end = addr + len;

vma->vm\_flags = **vm\_flags**; /\*VMA标记\*/

vma->vm\_page\_prot = **vm\_get\_page\_prot(vm\_flags)**; /\*设置用于页表项的访问权限，/mm/mmap.c\*/

**vma->vm\_pgoff = pgoff**; /\*VMA偏移量，匿名共享映射为0\*/

**INIT\_LIST\_HEAD(&vma->anon\_vma\_chain);** /\*初始化匿名反向映射链表为空\*/

if (file) { /\*文件映射\*/

if (vm\_flags & VM\_DENYWRITE) {

error = deny\_write\_access(file);

if (error)

goto free\_vma;

}

if (vm\_flags & VM\_SHARED) {

error = mapping\_map\_writable(file->f\_mapping);

if (error)

goto allow\_write\_and\_free\_vma;

}

**vma->vm\_file = get\_file(file)**; /\*指向file实例\*/

**error = file->f\_op->mmap(file, vma)**;

/\*调用file->f\_op->mmap(file, vma)映射函数，非常重要的操作\*/

/\*磁盘文件通常为generic\_file\_mmap()函数，主要内容是为vm\_ops成员赋值\*/

if (error)

goto unmap\_and\_free\_vma;

WARN\_ON\_ONCE(addr != vma->vm\_start);

**addr = vma->vm\_start**;

**vm\_flags = vma->vm\_flags**;

} else if (vm\_flags & VM\_SHARED) { /\*共享匿名映射\*/

error = **shmem\_zero\_setup(vma)**;

/\*创建共享匿名映射，文件映射，映射文件为/dev/zero，/mm/shmem.c\*/

if (error)

goto free\_vma;

}

**vma\_link(mm, vma, prev, rb\_link, rb\_parent)**; /\*插入地址空间管理结构和文件反向映射结构\*/

if (file) { /\*文件映射\*/

if (vm\_flags & VM\_SHARED)

mapping\_unmap\_writable(file->f\_mapping);

/\*地址空间结构i\_mmap\_writable成员减1，/include/linux/fs.h\*/

if (vm\_flags & VM\_DENYWRITE)

allow\_write\_access(file);

}

file = vma->vm\_file;

out:

perf\_event\_mmap(vma);

vm\_stat\_account(mm, vm\_flags, file, len >> PAGE\_SHIFT); /\*更新VMA统计量\*/

if (vm\_flags & VM\_LOCKED) {

if (!((vm\_flags & VM\_SPECIAL) || is\_vm\_hugetlb\_page(vma) || \

vma == get\_gate\_vma(current->mm)))

mm->locked\_vm += (len >> PAGE\_SHIFT);

else

vma->vm\_flags &= ~VM\_LOCKED;

}

if (file)

uprobe\_mmap(vma); /\*/kernel/events/uprobes.c\*/

vma->vm\_flags |= VM\_SOFTDIRTY;

**vma\_set\_page\_prot(vma)**; /\*修改VMA实例vm\_page\_prot成员值，/mm/mmap.c\*/

return addr;

...

}

mmap\_region()函数在地址空间内存域（扩展）红黑树中查找新区域插入点，如果新区域与现有VMA有重叠，则调用**do\_munmap()**函数解除重叠区的映射（拆分VMA），然后判断新区域能否与现有VMA合并，能则合并，不能则创建新vm\_area\_struct实例并初始化。如果新映射区域是文件映射，则调用文件操作结构中的mmap()函数，如果是共享匿名映射则调用shmem\_zero\_setup(vma)函数创建映射。

映射区域vm\_area\_struct实例将被插入到地址空间管理结构中，如果是文件映射VMA还将被插入文件地址空间区间树（反向映射结构），如果是匿名映射，反向映射结构初始化为空。mmap\_region()函数在设置内存域实例vm\_flags和vm\_page\_prot成员后，返回映射区域起始虚拟地址。

总之，mmap\_pgoff()系统调用先在进程地址空间获取未映射区域，判断其能否与现有VMA合并，能则合并，否则创建新VMA实例，然后设置VMA实例并插入地址空间管理结构和反向映射结构，最后判断是否需要立即为映射区域建立映射，需要则建立映射，最终返回映射区域起始虚拟地址。

#### 2非线性映射

内存映射可将文件中连续的某段内容映射到进程地址空间连续的内存区（VMA）中，而文件重映射（非线性映射）是指将文件中的某段内容映射到已有文件映射VMA内部，解除原内存段映射，映射文件必须是同一个文件。

非线性映射结构如下图所示：



重映射区域必须位于现有映射VMA内部，重映射后原VMA被拆分成三个VMA，前后VMA映射保持不变，中间VMA映射文件另一段内容。

文件重映射系统调用为remap\_file\_pages()，系统调用实现函数在/mm/mmap.c文件内，代码如下：

SYSCALL\_DEFINE5(remap\_file\_pages, unsigned long, start, unsigned long, size, \

unsigned long, prot, unsigned long, pgoff, unsigned long, flags)

/\*

\*start：重映射区域起始虚拟地址；

\*size：重映射区域长度，字节数，重映射区域必须在现有VMA内部；

\*port：重映射区域访问权限；

\*pgoff：重映射文件内容在文件内的页偏移量；

\*flags：映射标记。

\*/

{

struct mm\_struct \*mm = current->mm;

struct vm\_area\_struct \*vma;

unsigned long populate = 0;

unsigned long ret = -EINVAL;

struct file \*file;

... /\*输出信息\*/

if (prot)

return ret;

start = start & PAGE\_MASK;

size = size & PAGE\_MASK; /\*起始地址，映射区域长度页对齐（下对齐）\*/

if (start + size <= start)

return ret;

if (pgoff + (size >> PAGE\_SHIFT) < pgoff)

return ret;

down\_write(&mm->mmap\_sem);

**vma = find\_vma(mm, start)**; /\*查找VMA\*/

if (!vma || !(vma->vm\_flags & **VM\_SHARED**)) /\*必须是共享映射\*/

/\*若未找到VMA或没有设置VM\_SHARED标记，映射失败\*/

goto out;

if (start < vma->vm\_start || start + size > vma->vm\_end)

goto out; /\*重映射区域必须在已有VMA内部，可以是整个VMA\*/

if (pgoff == linear\_page\_index(vma, start)) {

/\*如果重映射内容与原内容相同，则返回0，/include/linux/pagemap.h\*/

ret = 0;

goto out;

}

prot |= vma->vm\_flags & VM\_READ ? PROT\_READ : 0; /\*设置访问权限\*/

prot |= vma->vm\_flags & VM\_WRITE ? PROT\_WRITE : 0;

prot |= vma->vm\_flags & VM\_EXEC ? PROT\_EXEC : 0;

flags &= MAP\_NONBLOCK; /\*只保留MAP\_NONBLOCK标记位值\*/

flags |= **MAP\_SHARED | MAP\_FIXED | MAP\_POPULATE**;

/\*固定地址、共享映射、立即建立映射\*/

if (vma->vm\_flags & VM\_LOCKED) { /\*原VMA锁定，解锁新映射区域\*/

flags |= MAP\_LOCKED;

**munlock\_vma\_pages\_range(vma, start, start + size)**;

}

**file = get\_file(vma->vm\_file)**; /\*增加file引用计数，返回vma->vm\_file\*/

ret = **do\_mmap\_pgoff(vma->vm\_file, start, size,prot, flags, pgoff, &populate)**;

/\*创建映射，返回起始虚拟地址\*/

fput(file);

out:

up\_write(&mm->mmap\_sem);

if (populate)

**mm\_populate(ret, populate)**; /\*建立映射\*/

if (!IS\_ERR\_VALUE(ret))

ret = 0;

return ret; /\*返重映射起始虚拟地址\*/

}

理解了前面的线性内存映射后，文件重映射就不难理解了。重映射区域必须在已有VMA范围内，不能越界（可以是整个VMA），并且原VMA必须是共享映射。重映射区域保留原VMA的读写执行权限，如果原VMA是锁定的，则需要对重映射区域解锁。

重映射操作调用前面介绍过的do\_mmap\_pgoff()函数在重映射区域创建映射。do\_mmap\_pgoff()函数内部最终会调用do\_munmap()函数对原VMA进行拆分，拆分成三个VMA，前后两个内存域映射保持不变，中间VMA解除映射后用于重映射。

remap\_file\_pages()最后判断如果populate变量不为0，则调用mm\_populate()函数立即分配物理页帧，建立映射，返回重映射起始虚拟地址。

#### 3文件映射说明

前面介绍的线性映射和非线性映射都可用于文件映射。由于文件映射涉及文件操作和文件地址空间操作等，因此下面先仅对文件映射做一个简要的介绍，到第11章将介绍具体函数的实现。

文件映射数据结构如下图所示（普通磁盘文件）：



对于普通磁盘文件，其文件操作结构file\_operations实例中mmap()函数设为generic\_file\_mmap()，在此函数中将设置VMA关联的vm\_operations\_struct实例为**generic\_file\_vm\_ops**，实例中的函数实现对文件地址空间中文件内容页缓存的操作，包括获取指定页映射到VMA，回写缓存页等。文件地址空间操作负责页缓存与块设备中文件内容的交互。

generic\_file\_vm\_ops实例定义如下（/mm/filemap.c）：

const struct vm\_operations\_struct generic\_file\_vm\_ops = {

.fault = **filemap\_fault**, /\*缺页处理函数，获取映射的缓存页\*/

.map\_pages = filemap\_map\_pages,

.page\_mkwrite = filemap\_page\_mkwrite,

};

在创建文件映射时将调用file\_operations实例中mmap()函数设置VMA关联的vm\_operations\_struct实例。在CPU访问到文件映射VMA时，若映射未建立，将调用vm\_operations\_struct实例中的fault()函数，从页缓存中获取页，以此修改VMA页表项，映射到VMA，本章后面将介绍缺页异常的处理。文件地址空间操作到第11章再做介绍。

对于设备文件，file\_operations实例中mmap()函数由驱动程序定义，函数内直接将驱动程序申请的内存或IO内存映射到用户进程地址空间，本节后面将介绍驱动程序中mmap()函数调用的映射函数。

### 4.6.5 mremap()

mremap()系统调用用于扩展或收缩现有映射区域，隐含移动映射区域。系统调用标记参数flags取值定义在/include/uapi/linux/mman.h头文件内：

#define MREMAP\_MAYMOVE 1 /\*可能移动内存域\*/

#define MREMAP\_FIXED 2 /\*新内存域设在固定位置\*/

参数flags不能只设置MREMAP\_FIXED标记位，若设置了MREMAP\_FIXED标记位必须同时设置标记位MREMAP\_MAYMOVE。可以只设置MREMAP\_MAYMOVE标记位或两个都不设置。

mremap()系统调用实现函数在/mm/mremap.c文件内，原型如下：

SYSCALL\_DEFINE5(mremap, unsigned long, addr, unsigned long, old\_len, \

unsigned long, new\_len, unsigned long, flags,unsigned long, new\_addr)

{...}

下面先分几种情形来介绍函数的功能，然后再分析其源代码：

（1）设置了MREMAP\_FIXED和MREMAP\_MAYMOVE标记位，收缩或扩展并移动VMA至指定地址。

旧区域[addr,addr+old\_len)必须位于某个VMA内部，新区域地址new\_addr必须页对齐，新区域长度为new\_len。新区域与旧区域不能有重叠。

假设旧区域比新区域短（扩展VMA），如下图中左侧所示，则在进程地址空间new\_addr处new\_len长度的未映射区域创建VMA（若不能在此地址创建映射，返回错误码），复制旧区域映射至新VMA（含页表项数据），解除旧区域映射。若需要为新区域立即建立映射，则还需要为扩展的区域立即建立映射（图中阴影部分）。

若旧区域比新区域长，则在new\_addr地址创建VMA，长度为new\_len（不能创建映射返回错误码），复制旧区域[addr,addr+new\_len)映射至新VMA，解除整个旧区域的映射。



（2）只设置了MREMAP\_MAYMOVE标记位，新VMA地址由内核确定。

旧区域[addr,addr+old\_len)必须位于某个VMA内部，新地址new\_addr参数不使用。如下图左侧所示，假设新区域比旧区域短，则只需要解除原VMA中旧区域比新区域多出部分的映射即可（阴影部分）。



如上图中右侧所示，假设旧区域正好包含整个VMA，且new\_len大于old\_len，即扩展VMA。如果可以直接扩展原VMA，则直接扩展VMA即可。

如果上例中不能直接扩展VMA，或者旧区域只是所在VMA的一部分，则由内核分配一段未映射区域，创建映射（VMA），复制旧区域VMA映射信息至新VMA，解除旧区域映射。如果需要立即建立映射，则为扩展区域建立映射。这与（1）的区别就是新VMA地址由内核确定，而不是由new\_addr参数指定（上图中未画出）。

（3）MREMAP\_FIXED和MREMAP\_MAYMOVE标记位都没有设置。

这时只处理两种情形，一是收缩旧区域的情形，即old\_len >= new\_len，主要是解除收缩区域的映射；二是new\_len>old\_len，旧区域正好是整个VMA，且扩展区域能合并到现有VMA的情形，直接调整现有VMA。除这两种情形外，其它情形将返回错误码，也就是说不会移动VMA，只收缩或扩展VMA。

mremap()系统调用实现函数代码如下：

SYSCALL\_DEFINE5(mremap, unsigned long, addr, unsigned long, old\_len, \

unsigned long, new\_len, unsigned long, flags,unsigned long, new\_addr)

/\*addr：旧起始虚拟地址，必须页对齐；

\*new\_addr：新起始虚拟地址，必须页对齐（移动VMA至指定地址）；

\*old\_len：旧映射区域长度，[addr,addr+old\_len)必须位于某个VMA内部，不能跨越或超过VMA边界；

\*new\_len：新映射区域长度（字节数），flags：标记。

\*/

{

struct mm\_struct \*mm = current->mm;

struct vm\_area\_struct \*vma;

unsigned long ret = -EINVAL;

unsigned long charged = 0;

bool locked = false;

if (flags & ~(MREMAP\_FIXED | MREMAP\_MAYMOVE)) /\*只能设置标记参数低2位\*/

return ret;

if (flags & MREMAP\_FIXED && !(flags & MREMAP\_MAYMOVE)) /\*标记参数检查\*/

return ret;

if (addr & ~PAGE\_MASK) /\*起始地址必须页对齐\*/

return ret;

old\_len = PAGE\_ALIGN(old\_len); /\*长度值页对齐\*/

new\_len = PAGE\_ALIGN(new\_len);

if (!new\_len) /\*新长度不能为0\*/

return ret;

down\_write(&current->mm->mmap\_sem);

if (flags & **MREMAP\_FIXED**) { /\*设置了MREMAP\_FIXED和MREMAP\_MAYMOVE标记位\*/

ret = **mremap\_to**(addr, old\_len, new\_addr, new\_len,&locked); /\*/mm/mremap.c\*/

/\*处理情形（1），在new\_addr地址创建映射，复制旧映射，解除旧映射等\*/

goto out;

}

/\*只设置了MREMAP\_MAYMOVE标记位，或两个标记位都没有设置\*/

if (old\_len >= new\_len) { /\*收缩VMA\*/

ret = **do\_munmap**(mm, addr+new\_len, old\_len - new\_len); /\*解除收缩部分映射\*/

if (ret && old\_len != new\_len)

goto out;

ret = addr;

goto out;

}

/\*需要扩展VMA（并移动）\*/

vma = **vma\_to\_resize**(addr, old\_len, new\_len, &charged); /\*/mm/mremap.c\*/

/\*addr必须位于vma内部，函数判断能否扩展vma，若不能返回错误码\*/

/\*若vma设置了VM\_ACCOUNT标记位，charged保存扩展页数\*/

...

if (old\_len == vma->vm\_end - addr) { /\*addr正好是vma起始地址，old\_len正好是其长度\*/

if (vma\_expandable(vma, new\_len - old\_len)) { /\*如果可以扩展vma\*/

int pages = (new\_len - old\_len) >> PAGE\_SHIFT;

if (vma\_adjust(vma, vma->vm\_start, addr + new\_len,vma->vm\_pgoff, NULL)) {

ret = -ENOMEM; /\*调整现有VMA即可\*/

goto out;

}

vm\_stat\_account(mm, vma->vm\_flags, vma->vm\_file, pages);

if (vma->vm\_flags & VM\_LOCKED) {

mm->locked\_vm += pages;

locked = true;

new\_addr = addr;

}

ret = addr;

goto out;

} /\*如果不能扩展vma，继续往下执行创建新VMA\*/

}

/\*旧区域不是正好包含整个VMA，或者不能扩展当前VMA，需要创建新VMA，并移动\*/

ret = -ENOMEM;

/\*运行至此，如果没有设置 MREMAP\_MAYMOVE标记位将跳过下面if语句，返回错误码\*/

if (flags & MREMAP\_MAYMOVE) {

unsigned long map\_flags = 0;

if (vma->vm\_flags & VM\_MAYSHARE)

map\_flags |= MAP\_SHARED;

new\_addr = **get\_unmapped\_area**(vma->vm\_file, 0, new\_len,

vma->vm\_pgoff +((addr - vma->vm\_start) >> PAGE\_SHIFT),

map\_flags); /\*获取未映射区域，地址设为0，由内核分配，长度为new\_len\*/

if (new\_addr & ~PAGE\_MASK) {

ret = new\_addr;

goto out;

}

ret = **move\_vma**(vma, addr, old\_len, new\_len, new\_addr, &**locked**); /\*移动VMA\*/

/\*在[new\_addr,new\_addr+new\_len)区域创建VMA，复制[addr,addr+old\_len)映射至新VMA，

\*解除旧区域映射，返回新VMA起始地址。\*/

} /\*if (flags & MREMAP\_MAYMOVE)结束\*/

out:

if (ret & ~PAGE\_MASK)

vm\_unacct\_memory(charged);

up\_write(&current->mm->mmap\_sem);

if (locked && new\_len > old\_len) /\*扩展VMA，且需要立即建立映射\*/

**mm\_populate**(new\_addr + old\_len, new\_len - old\_len);

return ret; /\*返回新VMA起始地址\*/

}

由函数代码可知，如果设置了MREMAP\_FIXED和MREMAP\_MAYMOVE标记位，表示将原区域映射移动到指定起始地址的映射区并扩展或收缩映射区域，不成功将返回错误码。

如果只设置了MREMAP\_MAYMOVE标记位，则可能是收缩/扩展原VMA，或收缩/扩展并移动VMA，新区域地址由内核确定。

如果MREMAP\_FIXED和MREMAP\_MAYMOVE标记位都没有设置，则只会收缩或扩展原VMA，不会移动VMA。移动VMA等函数请读者自行阅读源代码。

### 4.6.6 mprotect()

mprotect()系统调用用于修改映射区的访问权限，访问权限取值定义如下：/\*/arch/mips/include/uapi/asm/mman.h\*/

#define PROT\_NONE 0x00 /\*映射页不可被访问\*/

#define PROT\_READ 0x01 /\*可读\*/

#define PROT\_WRITE 0x02 /\*可写\*/

#define PROT\_EXEC 0x04 /\*可执行\*/

#define PROT\_SEM 0x10 /\*页可能被用于原子操作\*/

#define PROT\_GROWSDOWN 0x01000000

/\*对向下生长的内存域，访问权限修改扩展到内存域起始位置\*/

#define PROT\_GROWSUP 0x02000000

/\*对向上生长的内存域，访问权限修改扩展到内存域结束位置\*/

mprotect()系统调用实现函数位于/mm/mprotect.c文件内，代码如下：

SYSCALL\_DEFINE3(mprotect, unsigned long, start, size\_t, len,unsigned long, prot)

/\*start：被修改区域起始地址，必须页对齐且位于现有VMA内部，

\*len：长度，字节数，prot：新访问权限。\*/

{

unsigned long vm\_flags, nstart, end, tmp, reqprot;

struct vm\_area\_struct \*vma, \*prev;

int error = -EINVAL;

const int grows = prot & (PROT\_GROWSDOWN|PROT\_GROWSUP);

prot &= ~(PROT\_GROWSDOWN|PROT\_GROWSUP);

if (grows == (PROT\_GROWSDOWN|PROT\_GROWSUP)) /\*不同时设置\*/

return -EINVAL;

if (**start & ~PAGE\_MASK**) /\*起始地址必须页对齐\*/

return -EINVAL;

if (!len)

return 0;

len = PAGE\_ALIGN(len); /\*长度页对齐\*/

**end = start + len;**  /\*被修改区域长度\*/

if (end <= start)

return -ENOMEM;

if (!arch\_validate\_prot(prot))

return -EINVAL;

reqprot = prot;

if ((prot & PROT\_READ) && (current->personality & READ\_IMPLIES\_EXEC))

prot |= PROT\_EXEC;

vm\_flags = **calc\_vm\_prot\_bits(prot)**; /\*访问权限转VMA标记\*/

down\_write(&current->mm->mmap\_sem);

vma = **find\_vma**(current->mm, start); /\*查找VMA，start必须位于VMA内部\*/

error = -ENOMEM;

if (!vma)

goto out;

prev = vma->vm\_prev; /\*前一个VMA\*/

if (unlikely(grows & PROT\_GROWSDOWN)) { /\*VMA向下生长\*/

if (vma->vm\_start >= end)

goto out;

start = vma->vm\_start;

error = -EINVAL;

if (!(vma->vm\_flags & VM\_GROWSDOWN))

goto out;

} else {　　　　 /\*VMA向上生长\*/

if (vma->vm\_start > start)

goto out;

if (unlikely(grows & PROT\_GROWSUP)) {

end = vma->vm\_end;

error = -EINVAL;

if (!(vma->vm\_flags & VM\_GROWSUP))

goto out;

}

}

if (start > vma->vm\_start) /\*start在VMA内部，不是起始地址\*/

prev = vma;

for (nstart = start ; ; ) { /\*遍历被修改区域跨越的VMA\*/

unsigned long newflags;

/\* Here we know that vma->vm\_start <= nstart < vma->vm\_end. \*/

newflags = vm\_flags; /\*访问权限确定的VMA标记\*/

newflags |= (vma->vm\_flags & ~(VM\_READ | VM\_WRITE | VM\_EXEC)); /\*设置VMA标记\*/

/\* newflags >> 4 shift VM\_MAY% in place of VM\_% \*/

if ((newflags & ~(newflags >> 4)) & (VM\_READ | VM\_WRITE | VM\_EXEC)) {

error = -EACCES;

goto out;

}

error = security\_file\_mprotect(vma, reqprot, prot);

if (error)

goto out;

tmp = vma->vm\_end; /\*VMA结束地址\*/

if (tmp > end)

tmp = end;

error = **mprotect\_fixup**(vma, &prev, nstart, tmp, newflags);

/\*修改VMA访问权限，/mm/mprotect.c\*/

if (error)

goto out;

nstart = tmp;

if (nstart < prev->vm\_end)

nstart = prev->vm\_end;

if (nstart >= end)

goto out;

vma = prev->vm\_next;

if (!vma || vma->vm\_start != nstart) { /\*被修改区域不能有空洞\*/

error = -ENOMEM;

goto out;

}

}

out:

up\_write(&current->mm->mmap\_sem);

return error; /\*成功返回0\*/

}

mprotect()系统调用的执行流程与前面介绍的锁定内存区域的mlock()系统调用类似。假设修改访问权限的内存区域如下图所示（不能含有空洞），系统调用内将遍历跨越的三个VMA，对每个VMA调用函数mprotect\_fixup()修改跨越区域的访问权限，修改后需要考虑能否合并VMA，如果不能则保持原VMA（如VMA1）或拆分VMA（如VMA0和VMA2）。mprotect\_fixup()函数源代码请读者自行阅读。



### 4.6.7 madvise()

madvise()系统调用用于进程向内核提供它将如何访问映射页，以帮助内核预读之前或之后的页数据和确定缓存策略。

madvise()系统调用实现函数原型如下（/mm/madvise.c）：

SYSCALL\_DEFINE3(madvise, unsigned long, start, size\_t, len\_in, int, behavior);

start表示内存区域起始地址（必须页对齐），len表示长度（字节数），behavior参数标识进程访问内存的方式。behavior参数取值定义如下（/arch/mips/include/uapi/asm/mman.h）：

#define MADV\_NORMAL 0 /\*不需要做特殊的对待\*/

#define MADV\_RANDOM 1 /\*随机访问\*/

#define MADV\_SEQUENTIAL 2 /\*可能顺序访问，且只访问一次\*/

#define MADV\_WILLNEED 3 /\*通知内核预读页\*/

#define MADV\_DONTNEED 4 /\*进程完成了对指定区域映射页的访问\*/

#define MADV\_REMOVE 9 /\*进程要移除指定区域映射页以及关联后备存储中页\*/

#define **MADV\_DONTFORK** 10 /\*指定映射区域不传递给子进程（不复制VMA）\*/

#define MADV\_DOFORK 11 /\*取消MADV\_DONTFORK标记位（复制VMA）\*/

#define MADV\_MERGEABLE 12 /\* KSM may merge identical pages \*/

#define MADV\_UNMERGEABLE 13 /\* KSM may not merge identical pages \*/

#define MADV\_HWPOISON 100 /\* poison a page for testing \*/

#define MADV\_HUGEPAGE 14 /\* Worth backing with hugepages \*/

#define MADV\_NOHUGEPAGE 15 /\* Not worth backing with hugepages \*/

#define MADV\_DONTDUMP 16 /\*核心转储时排除此区域\*/

#define MADV\_DODUMP 17 /\*清除 MADV\_NODUMP标记\*/

madvise()系统调用实现函数如下（/mm/madvise.c）：

SYSCALL\_DEFINE3(madvise, unsigned long, start, size\_t, len\_in, int, behavior)

{

unsigned long end, tmp;

struct vm\_area\_struct \*vma, \*prev;

int unmapped\_error = 0;

int error = -EINVAL;

int write;

size\_t len;

struct blk\_plug plug;

#ifdef CONFIG\_MEMORY\_FAILURE

if (behavior == MADV\_HWPOISON || behavior == MADV\_SOFT\_OFFLINE)

return madvise\_hwpoison(behavior, start, start+len\_in);

#endif

if (!madvise\_behavior\_valid(behavior))

return error;

if (start & ~PAGE\_MASK) /\*起始地址必须页对齐\*/

return error;

len = (len\_in + ~PAGE\_MASK) & PAGE\_MASK; /\*长度页对齐，上对齐\*/

/\* Check to see whether len was rounded up from small -ve to zero \*/

if (len\_in && !len)

return error;

**end** = start + len; /\*结束地址\*/

if (end < start)

return error;

error = 0;

if (end == start)

return error;

write = madvise\_need\_mmap\_write(behavior); /\*是否需要修改vma->vm\_flags成员值\*/

if (write) /\*需要修改vma->vm\_flags成员值，获取写锁，否则获取读锁\*/

down\_write(&current->mm->mmap\_sem);

else

down\_read(&current->mm->mmap\_sem);

/\* [start,end)区域可以包含空洞，只不过处理完成后返回-ENOMEM错误码\*/

vma = find\_vma\_prev(current->mm, start, &prev);

if (vma && start > vma->vm\_start) /\*start位于vma内部\*/

prev = vma;

blk\_start\_plug(&plug);

for (;;) { /\*遍历[start,end)区域跨越的VMA\*/

/\* Still start < end. \*/

error = -ENOMEM;

if (!vma)

goto out;

/\* Here start < (end|vma->vm\_end). \*/

if (start < vma->vm\_start) { /\*遇到了空洞，跳过，处理随后的VMA\*/

unmapped\_error = -ENOMEM;

start = vma->vm\_start;

if (start >= end)

goto out;

}

/\* Here vma->vm\_start <= start < (end|vma->vm\_end) \*/

tmp = vma->vm\_end;

if (end < tmp)

tmp = end;

/\* Here vma->vm\_start <= start < tmp <= (end|vma->vm\_end). \*/

error = **madvise\_vma**(vma, &prev, start, tmp, behavior); /\*对跨越的VMA进行处理\*/

if (error)

goto out;

start = tmp;

if (prev && start < prev->vm\_end)

start = prev->vm\_end;

error = unmapped\_error;

if (start >= end) /\*遍历跨越VMA结束\*/

goto out;

if (prev)

vma = prev->vm\_next; /\*下一VMA\*/

else /\* madvise\_remove dropped mmap\_sem \*/

vma = find\_vma(current->mm, start);

} /\*遍历跨越的VMA结束\*/

out:

blk\_finish\_plug(&plug);

if (write)

up\_write(&current->mm->mmap\_sem);

else

up\_read(&current->mm->mmap\_sem);

return error;

}

madvise()系统调用的处理方式与前面的mprotect()系统调用类似，也是遍历[start,end)区域跨越的VMA，对每个VMA调用**madvise\_vma()**函数进行处理，只不过这里可以跨越空洞区。如下图所示，VMA0与VMA1之间存在空洞，madvise()系统调用会跳过空洞区，继续处理后面跨越的VMA，只过最后返回-ENOMEM错误码。



madvise\_vma()函数根据behavior参数值调用不同的处理函数，代码如下（/mm/madvise.c）：

static long madvise\_vma(struct vm\_area\_struct \*vma, struct vm\_area\_struct \*\*prev,

unsigned long start, unsigned long end, int behavior)

{

switch (behavior) {

case MADV\_REMOVE:

return madvise\_remove(vma, prev, start, end); /\*移除映射页，会造成文件空洞\*/

case MADV\_WILLNEED:

return madvise\_willneed(vma, prev, start, end); /\*执行预读\*/

case MADV\_DONTNEED:

return madvise\_dontneed(vma, prev, start, end); /\*不需要页了，可以回收\*/

default:

return madvise\_behavior(vma, prev, start, end, behavior); /\*主要是修改vma->vm\_flags标记\*/

}

}

以上处理函数主要是对vma->vm\_flags标记成员进行修改（修改后考虑VMA合并），或通知内核执行预读、回收页等操作，函数源代码请读者自行阅读。

### 4.6.8 msync()

msync()系统调用用于同步文件映射的文件，即将进程修改过的文件内容写回磁盘（块设备）。此系统调用标记参数flags取值定义在/arch/mips/include/uapi/asm/mman.h头文件：

#define MS\_ASYNC 0x0001 /\*同步映射文件内容（不立即发起IO操作，无操作）\*/

#define MS\_INVALIDATE 0x0002 /\*使映射和缓存无效\*/

#define MS\_SYNC 0x0004 /\*立即同步文件内容\*/

msync()系统调用实现函数定义如下（/mm/msync.c）

SYSCALL\_DEFINE3(msync, unsigned long, start, size\_t, len, int, flags)

/\*start：同步映射区域起始地址，必须页对齐，len：长度，字节数，flags：标记\*/

{

unsigned long end;

struct mm\_struct \*mm = current->mm;

struct vm\_area\_struct \*vma;

int unmapped\_error = 0;

int error = -EINVAL;

if (flags & ~(MS\_ASYNC | MS\_INVALIDATE | MS\_SYNC))

goto out;

if (start & ~PAGE\_MASK) /\*起始地址必须页对齐\*/

goto out;

if ((flags & MS\_ASYNC) && (flags & MS\_SYNC)) /\*不能同时设置这两个标记位\*/

goto out;

error = -ENOMEM;

len = (len + ~PAGE\_MASK) & PAGE\_MASK; /\*长度页对齐，上对齐\*/

end = start + len; /\*同步区域结束地址\*/

if (end < start)

goto out;

error = 0;

if (end == start)

goto out;

/\*[start,end)区域可以跨越空洞，但是最后返回-ENOMEM错误码\*/

down\_read(&mm->mmap\_sem);

vma = find\_vma(mm, start); /\*查找VMA\*/

for (;;) { /\*遍历跨越的VMA\*/

struct file \*file;

loff\_t fstart, fend;

/\* Still start < end. \*/

error = -ENOMEM;

if (!vma)

goto out\_unlock;

/\* Here start < vma->vm\_end. \*/

if (start < vma->vm\_start) {

start = vma->vm\_start;

if (start >= end)

goto out\_unlock;

unmapped\_error = -ENOMEM;

}

/\* Here vma->vm\_start <= start < vma->vm\_end. \*/

if ((flags & MS\_INVALIDATE) &&(vma->vm\_flags & VM\_LOCKED)) {

error = -EBUSY;

goto out\_unlock;

}

file = vma->vm\_file;

/\*文件内容起始结束偏移量，字节数\*/

fstart = (start - vma->vm\_start) +((loff\_t)vma->vm\_pgoff << PAGE\_SHIFT);

fend = fstart + (min(end, vma->vm\_end) - start) - 1;

start = vma->vm\_end;

if (**(flags & MS\_SYNC) && file &&(vma->vm\_flags & VM\_SHARED)**) {

get\_file(file);

up\_read(&mm->mmap\_sem);

error = **vfs\_fsync\_range**(file, fstart, fend, 1); /\*同步文件内容，/fs/sync.c\*/

fput(file);

if (error || start >= end)

goto out;

down\_read(&mm->mmap\_sem);

vma = find\_vma(mm, start);

} else {

if (start >= end) {

error = 0;

goto out\_unlock;

}

vma = vma->vm\_next;

}

}

out\_unlock:

up\_read(&mm->mmap\_sem);

out:

return error ? : unmapped\_error;

}

msync()系统调用与前面介绍的系统调用处理方式类似，也是遍历[start,end)区域跨越的VMA（可以包含空洞），对每个VMA进行处理。

msync()系统调用只在flags参数设置了MS\_SYNC标记位时，对设置了**VM\_SHARED**标记位的文件映射VMA调用vfs\_fsync\_range()函数同步文件内容，其它情形不对VMA映射页进行处理。

vfs\_fsync\_range()函数调用文件操作结构中file->f\_op->fsync(file, start, end, datasync)函数同步文件内容，详情见第11章。

### 4.6.9设备文件映射

Linux内核中设备也由文件表示，设备文件操作file\_operations实例中可以定义mmap()函数，用于将设备驱动程序使用的内存（或IO内存）映射到用户进程地址空间。

将指定页帧映射到进程VMA的函数有remap\_pfn\_range()、vm\_insert\_pfn()等，这些函数通常由设备文件操作结构file\_operations实例中mmap()函数调用。本小节将介绍这些函数的定义，并介绍/dev/mem设备文件的映射函数。

#### 1映射物理内存

remap\_pfn\_range()/vm\_insert\_pfn()函数将指定连续的物理内存（单页或多页）映射到进程地址空间VMA中指定起始地址的连续虚拟内存区。

##### ■remap\_pfn\_range()

remap\_pfn\_range()函数用于将一段连续的物理内存映射到用户地址空间，代码如下（/mm/memory.c）：

int remap\_pfn\_range(struct vm\_area\_struct \*vma, unsigned long addr, \

unsigned long pfn, unsigned long size, pgprot\_t prot)

/\*

\*vma：建立映射的VMA实例指针，addr：映射区域起始虚拟地址，pfn：物理内存起始页帧号，

\*size：映射区大小，prot：访问权限。

\*/

{

pgd\_t \*pgd;

unsigned long next;

unsigned long end = addr + PAGE\_ALIGN(size); /\*映射区域结束虚拟地址\*/

struct mm\_struct \*mm = vma->vm\_mm;

int err;

if (is\_cow\_mapping(vma->vm\_flags)) { /\*写时复制VMA，/mm/internal.h\*/

/\*设置了VM\_MAYWRITE标记，但没有设置VM\_SHARED标记\*/

if (addr != vma->vm\_start || end != vma->vm\_end) /\*映射区域必须正好包含整个VMA\*/

return -EINVAL;

**vma->vm\_pgoff = pfn**; /\*PFN映射内存域，vm\_pgoff保存映射物理页帧号pfn\*/

}

err = track\_pfn\_remap(vma, &prot, pfn, addr, PAGE\_ALIGN(size)); /\*直接返回0\*/

if (err)

return -EINVAL;

**vma->vm\_flags |= VM\_IO | VM\_PFNMAP | VM\_DONTEXPAND | VM\_DONTDUMP**;

/\*设置VMA标记（特殊内存域）\*/

BUG\_ON(addr >= end);

**pfn -= addr >> PAGE\_SHIFT**; /\*先减，后面再加\*/

pgd = pgd\_offset(mm, addr);

flush\_cache\_range(vma, addr, end);

do { /\*扫描PGD页表项\*/

next = pgd\_addr\_end(addr, end);

err = **remap\_pud\_range(mm, pgd, addr, next,pfn + (addr >> PAGE\_SHIFT), prot)**;

/\*设置页表项\*/

if (err)

break;

} while (pgd++, addr = next, addr != end);

if (err)

untrack\_pfn(vma, pfn, PAGE\_ALIGN(size));

return err;

}

remap\_pfn\_range()函数设置完VMA标记后，逐级扫描页表，填充页表项。扫描页表的步骤与内核空间VMALLOC区建立映射时相似。在两级页表模式下，remap\_pud\_range()函数直接调用remap\_pte\_range()函数，扫描PGD页表项对应PTE页表，修改PTE页表项，建立映射。

remap\_pte\_range()函数代码如下（/mm/memory.c）：

static int remap\_pte\_range(struct mm\_struct \*mm, pmd\_t \*pmd, \

unsigned long addr, unsigned long end,unsigned long pfn, pgprot\_t prot)

{

pte\_t \*pte;

spinlock\_t \*ptl;

pte = pte\_alloc\_map\_lock(mm, pmd, addr, &ptl); /\*addr对应PTE页表项指针\*/

if (!pte)

return -ENOMEM;

arch\_enter\_lazy\_mmu\_mode(); /\*空操作\*/

do {

BUG\_ON(!pte\_none(\*pte));

**set\_pte\_at(mm, addr, pte, pte\_mkspecial(pfn\_pte(pfn, prot)))**; /\*设置页表项\*/

**pfn++**;

} while (pte++, addr += PAGE\_SIZE, addr != end);

arch\_leave\_lazy\_mmu\_mode(); /\*空操作\*/

pte\_unmap\_unlock(pte - 1, ptl);

return 0;

}

这里需要注意的是生成页表项调用的是pte\_mkspecial()函数，由物理页帧号和访问权限生成特殊页表项，在MIPS32体系结构中特殊页表项与普通页表项相同。在设置内存PTE页表项之后，并没有刷新TLB页表项，因为此函数不是在缺页异常处理函数中调用的。

##### ■vm\_insert\_pfn()

内核除了提供向进程地址空间映射一段物理内存的remap\_pfn\_range()函数外，还提供了将一个给定页帧映射（一次只映射一页）到进程指定VMA虚拟页的接口函数，如下所示（/mm/memory.c）：

int vm\_insert\_pfn(struct vm\_area\_struct \*vma, unsigned long addr,unsigned long pfn)

/\*将pfn页帧映射到addr地址的虚拟页\*/

{

int ret;

pgprot\_t pgprot = **vma->vm\_page\_prot**;

BUG\_ON(!(vma->vm\_flags & (VM\_PFNMAP|VM\_MIXEDMAP))); /\*标记检查\*/

BUG\_ON((vma->vm\_flags & (VM\_PFNMAP|VM\_MIXEDMAP)) ==

(VM\_PFNMAP|VM\_MIXEDMAP));

BUG\_ON((vma->vm\_flags & VM\_PFNMAP) && is\_cow\_mapping(vma->vm\_flags));

BUG\_ON((vma->vm\_flags & VM\_MIXEDMAP) && pfn\_valid(pfn));

if (addr < vma->vm\_start || addr >= vma->vm\_end) /\*addr必须在VMA内部\*/

return -EFAULT;

if (track\_pfn\_insert(vma, &pgprot, pfn))

return -EINVAL;

ret = **insert\_pfn(vma, addr, pfn, pgprot)**; /\*/mm/memory.c\*/

return ret;

}

vm\_insert\_pfn()函数内获取映射VMA的访问权限属性，检查VMA标记成员必须是设置VM\_PFNMAP或者VM\_MIXEDMAP标记位，然后调用insert\_pfn()函数完成页帧映射的建立。

insert\_pfn()函数代码如下（/mm/memory.c）：

static int insert\_pfn(struct vm\_area\_struct \*vma, unsigned long addr,unsigned long pfn, pgprot\_t prot)

/\*vma：VMA实例指针，addr：虚拟页地址，pfn：物理页帧号，prot：访问权限\*/

{

struct mm\_struct \*mm = vma->vm\_mm;

int retval;

pte\_t \*pte, entry;

spinlock\_t \*ptl;

retval = -ENOMEM;

**pte = get\_locked\_pte(mm, addr, &ptl)**; /\*PTE页表项指针\*/

if (!pte)

goto out;

retval = -EBUSY;

if (**!pte\_none(\*pte)**) /\*原PTE页表项必须为空才能建立映射\*/

goto out\_unlock;

**entry = pte\_mkspecial(pfn\_pte(pfn, prot));** /\*生成特殊PTE页表项\*/

**set\_pte\_at(mm, addr, pte, entry)**; /\*设置PTE页表项\*/

**update\_mmu\_cache(vma, addr, pte)**; /\*刷新TLB页表项和CPU缓存\*/

retval = 0;

out\_unlock:

pte\_unmap\_unlock(pte, ptl);

out:

return retval; /\*成功返回0\*/

}

insert\_pfn()函数比较简单，在获取PTE页表项指针后，判断原PTE页表项内容是否为空，如果不为空则不能建立映射。如果为空，则由pfn和prot生成特殊页表项，写入PTE页表，并刷新TLB页表项和CPU缓存。

#### 2映射设备内存

内核提供了将设备内存映射到进程虚拟内存的函数**vm\_iomap\_memory()**，定义如下（/mm/memory.c）：

int vm\_iomap\_memory(struct vm\_area\_struct \*vma, phys\_addr\_t start, unsigned long len)

/\*vma：VMA实例指针，start：设备内存起始物理地址，len：设备内存长度（字节数）\*/

{

unsigned long vm\_len, pfn, pages;

if (start + len < start) /\*检查物理地址有效性\*/

return -EINVAL;

len += start & ~PAGE\_MASK; /\*地址、长度值页对齐\*/

pfn = start >> PAGE\_SHIFT; /\*起始页帧号\*/

**pages = (len + ~PAGE\_MASK) >> PAGE\_SHIFT**; /\*设备内存包含页数量\*/

if (pfn + pages < pfn)

return -EINVAL;

if (vma->vm\_pgoff > pages)

return -EINVAL;

pfn += vma->vm\_pgoff; /\*pfn必须与vm\_pgoff匹配，vm\_pgoff为0则映射从pfn开始\*/

pages -= vma->vm\_pgoff; /\*vm\_pgoff不为0，映射页帧数量要减去vma->vm\_pgoff\*/

vm\_len = vma->vm\_end - vma->vm\_start;

if (vm\_len >> PAGE\_SHIFT > pages) /\*VMA长度必须小于等于设备内存长度\*/

return -EINVAL;

return **io\_remap\_pfn\_range(vma, vma->vm\_start, pfn, vm\_len, vma->vm\_page\_prot)**;

/\*即remap\_pfn\_range()函数，/include/asm-generic/pgtable.h\*/

}

vm\_iomap\_memory()函数先对映射IO内存起始地址和长度进行页对齐，然后判断vma->vm\_pgoff成员值，如果为0则表示VMA映射从起始页帧pfn开始，否则VMA映射从pfn+vma->vm\_pgoff页帧开始。如下图所示，假设VMA0中vm\_pgoff值为0，则VMA映射到页帧号为[pfn，pfn+vm\_len)的设备内存区域，VMA1中vm\_pgoff值为2，则VMA映射到页帧号为[pfn+2，pfn+vm\_len)的设备内存区域。

vm\_iomap\_memory()函数最后调用io\_remap\_pfn\_range()函数完成映射的建立，在MIPS32体系结构中io\_remap\_pfn\_range()函数定义成与remap\_pfn\_range()函数相同。



#### 3 /dev/mem文件映射

本小节介绍的函数都是用于设备文件操作结构file\_operations实例中的mmap()函数，用于将指定物理内存或设备内存映射到进程地址空间。下面介绍一个特殊的设备文件/dev/mem，以及此文件的映射函数。

设备文件/dev/mem表示系统中的物理内存，文件的内容就是物理内存中的数据。读者可以学习完第7章文件系统，以及第9章字符设备驱动程序后再回过头来学习本小节。

在Linux内核中设备由设备文件文件表示，设备文件位于/dev目录下。每个设备具有一个主设备号和次设备号，mem设备主设备号为1，定义如下（/include/uapi/linux/major.h）：

MEM\_MAJOR 1

mem设备驱动程序在/drivers/char/mem.c文件内，在驱动程序初始化函数chr\_dev\_init()中将注册mem设备驱动程序，并创建设备文件。创建的设备文件包括/dev/mem（需选择DEVMEM配置选项）、/dev/kmem（需选择DEVKMEM配置选项）、/dev/zero等。下面以/dev/mem设备文件为例（次设备号为1），介绍其映射函数的实现。

如下图所示，在内核中字符设备由cdev实例示，实例关联file\_operations实例。在打开字符设备时将调用cdev实例关联file\_operations实例的open()函数。mem设备file\_operations实例的open()函数将根据设备文件中的次设备号，对文件file实例赋予不同的file\_operations实例，例如，/dev/mem设备文件file实例关联的实例为**mem\_fops**实例。



用户进程在映射/dev/mem文件时，要先通过open()系统调用打开设备文件，在打开操作中会调用设备文件关联mem\_fops实例中的open()函数，在此函数中将检查用户进程是否具有CAP\_SYS\_RAWIO权限，具有此权限的用户进程才可以打开/dev/mem文件。

用户进程打开/dev/mem文件获取文件描述符后，通过mmap()/mmap2()系统调用将指定物理内存（文件内容）映射到进程地址空间。在mmap()/mmap2()系统调用实现函数中将调用设备文件关联mem\_fops实例中定义的mmap()函数，即**mmap\_mem()**函数，用于执行真正的建立映射操作。

**mmap\_mem()**函数内必须对进程权限进行检查，如果随意将物理内存映射到用户进程地址空间，使用户进程可以随意访问物理内存，将是一件非常危险的事情。

mmap\_mem()函数定义如下（/drivers/char/mem.c）：

static int mmap\_mem(struct file \*file, struct vm\_area\_struct \*vma)

{

size\_t size = vma->vm\_end - vma->vm\_start; /\*映射内存大小\*/

if (!valid\_mmap\_phys\_addr\_range(vma->vm\_pgoff, size)) /\*返回1，/drivers/char/mem.c\*/

return -EINVAL;

if (!private\_mapping\_ok(vma)) /\*返回1，/drivers/char/mem.c\*/

return -ENOSYS;

if (!**range\_is\_allowed**(vma->vm\_pgoff, size))

return -EPERM; /\*选择STRICT\_DEVMEM选项，检查内存，/drivers/char/mem.c\*/

if (!phys\_mem\_access\_prot\_allowed(file, vma->vm\_pgoff, size,&vma->vm\_page\_prot))

return -EINVAL; /\*默认返回1（如果体系结构没有定义），/drivers/char/mem.c\*/

vma->vm\_page\_prot = **phys\_mem\_access\_prot**(file, vma->vm\_pgoff,size,vma->vm\_page\_prot);

/\*没有定义pgprot\_noncached，直接返回vma->vm\_page\_prot\*/

vma->vm\_ops = &mmap\_mem\_ops; /\*设置vm\_operations\_struct实例，只定义了access()函数\*/

if (**remap\_pfn\_range**(vma,vma->vm\_start,**vma->vm\_pgoff,**size,vma->vm\_page\_prot)) {

return -EAGAIN; /\*映射物理内存至VMA\*/

}

return 0; /\*成功返回0\*/

}

mmap\_mem()函数的目的是将物理内存映射到VMA，VMA起始虚拟地址为vma->vm\_start，物理内存起始页帧号为vma->vm\_pgoff。

mmap\_mem()函数需要检查映射物理内存的有效性（安全性），调用remap\_pfn\_range()函数映射物理内存至VMA。

range\_is\_allowed()函数用于逐页检查页帧，体系结构需定义STRICT\_DEVMEM配置选项，否则此函数直接返回1。MIPS32体系结构没有定义STRICT\_DEVMEM配置选项，因此在这里不检查页帧。

在创建/dev/mem，/dev/kmem设备文件时，设置了它们的访问权限为0，在open()系统调用中将检查用户进程的权限（见第7章），在文件操作结构的open()函数中还将检查用户进程是否具有CAP\_SYS\_RAWIO权限。总之，要将/dev/mem等文件映射到进程地址空间，用户需要较高的权限。

### 4.6.10解除映射

前面介绍了在用户进程地址空间创建映射及管理映射的方法，本小节介绍映射的解除。

进程地址空间解除映射总是以VMA为单位进行的，如果只是解除VMA中部分映射，则先对VMA进行拆分，然后再进行解除。解除映射的主要工作是：清空VMA对应的页表(项)，断开映射关系，减小映射页的计数值，将映射物理页帧释放回伙伴系统，将内存域vm\_area\_struct实例从管理结构中移出并释放回slab分配器。

#### 1系统调用

解除虚拟内存区映射的系统调用为**munmap()**，系统调用实现函数定义如下（/mm/mmap.c）：

SYSCALL\_DEFINE2(munmap, unsigned long, addr, size\_t, len)

/\*addr：解除映射区起始虚拟地址，必须页对齐，len：解除映射区长度，字节数\*/

{

profile\_munmap(addr);

return **vm\_munmap(addr, len)**; /\*/mm/mmap.c\*/

}

munmap()系统调用参数需指定解除映射内存区的起始虚拟地址和长度，系统调用由vm\_munmap(addr, len)函数实现。vm\_munmap()函数定义如下（/mm/mmap.c）：

int vm\_munmap(unsigned long start, size\_t len)

{

int ret;

struct mm\_struct \*mm = current->mm;

down\_write(&mm->mmap\_sem);

ret = **do\_munmap(mm, start, len);**  /\*/mm/mmap.c\*/

up\_write(&mm->mmap\_sem);

return ret;

}

do\_munmap()函数是内核中解除映射的接口函数，在前面介绍的映射管理中也可能调用此函数。下面将介绍do\_munmap()函数的实现。

#### 2接口函数

do\_munmap()函数用于解除进程地址空间某段虚拟内存的映射。函数内只对位于现有VMA内部的区域执行解除操作，对空洞区将不做处理。如果解除内存区只包含VMA的一部分，需要先对VMA进行拆分，再解除拆分后VMA的映射。

例如，如下图所示，假设要解除[start,end)区域映射，内含2个空洞区，start和end分别位于VMA0和VMA2内部，则需要以start和end为界对VMA0和VMA2进行拆分，然后再依次解除各VMA的映射。



do\_munmap()函数执行流程简列如下图所示：



do\_munmap()函数首先判断是否需要对首尾VMA进行拆分，如果需要则拆分VMA，对需解除映射的锁定VMA执行解锁操作，然后将需要解除映射的VMA从地址空间管理链表中移出（同时从红黑树移出），调用**unmap\_region()**函数逐个解除VMA映射，最后释放VMA实例（包括调用关联VMA操作结构中的close()函数等）。

do\_munmap()函数定义如下（/mm/mmap.c）：

int do\_munmap(struct mm\_struct \*mm, unsigned long start, size\_t len)

/\*mm：地址空间实例指针，start：解除映射区域起始虚拟地址，len：解除映射区域长度\*/

{

unsigned long end;

struct vm\_area\_struct \*vma, \*prev, \*last;

if (**(start & ~PAGE\_MASK)** || start > TASK\_SIZE || len > TASK\_SIZE-start) /\*参数有效性检查\*/

return -EINVAL;

len = PAGE\_ALIGN(len); /\*长度页对齐\*/

if (len == 0)

return -EINVAL;

/\*查找结束地址在start之后的第一个VMA \*/

vma = find\_vma(mm, **start**);

if (!vma) /\*start之后没有VMA，则无需执行解除操作，返回0\*/

return 0;

prev = vma->vm\_prev;

end = start + len; /\*解除映射区域结束地址\*/

if (vma->vm\_start >= end) /\*解除映射整个区域位于空洞区，什么也不用做，返回0\*/

return 0;

if (start > vma->vm\_start) { /\*start在首个VMA内部\*/

int error;

if (end < vma->vm\_end && mm->map\_count >= sysctl\_max\_map\_count)

/\*VMA数量超过了限制值不能再拆分VMA了\*/

return -ENOMEM;

error = **\_\_split\_vma(mm, vma, start, 0)**;

/\*将VMA以start为地址进行拆分，vma指向前半部分内存域实例\*/

if (error)

return error;

**prev = vma**;

}

last = find\_vma(mm, end);

if (last && end > last->vm\_start) { /\*如果end也位于VMA内部，拆分结尾VMA\*/

int error = **\_\_split\_vma(mm, last, end, 1)**; /\*last指向后半部分VMA实例\*/

if (error)

return error;

}

vma = prev ? prev->vm\_next : mm->mmap; /\*解除映射区域第一个VMA实例\*/

/\*解锁设置VM\_LOCKED标记位的VMA\*/

if (mm->locked\_vm) {

struct vm\_area\_struct \***tmp = vma**;

while (tmp && tmp->vm\_start < end) { /\*遍历解锁内存区VMA实例\*/

if (tmp->vm\_flags & **VM\_LOCKED**) {

mm->locked\_vm -= vma\_pages(tmp);

**munlock\_vma\_pages\_all**(tmp); /\*/mm/internal.h\*/

/\*调用前面介绍的munlock\_vma\_pages\_range()函数解锁VMA\*/

}

tmp = tmp->vm\_next; /\*下一个VMA\*/

}

}

**detach\_vmas\_to\_be\_unmapped(mm, vma, prev, end);**

/\*将解除映射VMA从地址空间VMA链表、红黑树中取出，/mm/mmap.c\*/

**unmap\_region(mm, vma, prev, start, end)**;

/\*逐个解除VMA映射，/mm/mmap.c\*/

arch\_unmap(mm, vma, start, end); /\*空操作\*/

**remove\_vma\_list(mm, vma)**;

/\*释放解除映射VMA实例，/mm/mmap.c\*/

return 0; /\*成功返回0\*/

}

do\_munmap()函数判断是否需要对首尾VMA进行拆分，如果需要则执行拆分操作，然后对锁定的VMA执行解除操作。

detach\_vmas\_to\_be\_unmapped()函数将解除映射内存区包含的VMA从地址空间mm\_struct实例中VMA链表、（扩展）红黑树中移出，参数vma指向解除映射VMA链表的第一个成员。

unmap\_region()函数遍历解除映射VMA链表，逐个对VMA执行解除映射操作，后面将详细介绍此函数的实现。

最后，do\_munmap()函数调用remove\_vma\_list()函数释放VMA实例，主要工作是更新统计量，如果是文件映射还需调用执行vma->vm\_ops->close()函数，最后释放vm\_area\_struct实例至slab分配器。

detach\_vmas\_to\_be\_unmapped()和remove\_vma\_list()函数源代码请读者自行阅读，下面将详细介绍函数**unmap\_region()**的实现。

#### 3解除VMA映射

unmap\_region()函数扫描解除映射VMA实例链表，收集各VMA映射的页帧，解除页帧映射并清空各VMA对应的页表（项）。

unmap\_region()函数定义如下（/mm/mmap.c）：

static void unmap\_region(struct mm\_struct \*mm, \

struct vm\_area\_struct \*vma, struct vm\_area\_struct \*prev,unsigned long start, unsigned long end)

/\*

\*vma：指向解除映射VMA链表第一个成员，prev：指向vma前一个VMA，

\*start：起始虚拟地址，end：结束虚拟地址。

\*/

{

struct vm\_area\_struct \*next = prev ? prev->vm\_next : mm->mmap;

struct mmu\_gather **tlb**; /\*mmu\_gather实例用于收集需要释放的映射页\*/

lru\_add\_drain(); /\*移出CPU页向量中页至LRU链表，详见第11章，/mm/swap.c\*/

**tlb\_gather\_mmu(&tlb, mm, start, end)**; /\*初始化mmu\_gather实例，/mm/memory.c\*/

update\_hiwater\_rss(mm); /\*更新mm\_struct实例hiwater\_rss成员值，/include/linux/mm.h\*/

**unmap\_vmas**(&tlb, vma, start, end); /\*断开VMA映射，/mm/memory.c\*/

**free\_pgtables**(&tlb, vma, prev ? prev->vm\_end : FIRST\_USER\_ADDRESS,

next ? next->vm\_start : USER\_PGTABLES\_CEILING);

/\*释放空闲页表，/mm/memory.c\*/

**tlb\_finish\_mmu**(&tlb, start, end);

/\*释放映射页及mmu\_gather\_batch实例，/mm/memory.c\*/

}

unmap\_region()函数内定义并初始化了mmu\_gather结构体实例，用于收集（缓存）需要释放页帧的page实例。unmap\_vmas()函数遍历解除映射VMA链表，对每个VMA读取并清零其对应的PTE页表项，断开映射，由页表项获取映射页帧page实例，添加到mmu\_gather实例。

free\_pgtables()函数扫描解除映射内存区可以释放的各级页表，将可释放的页表对应的page实例也添加到mmu\_gather实例。

tlb\_finish\_mmu()函数释放mmu\_gather实例缓存的页，以及mmu\_gather\_batch结构体实例占用的页帧（mmu\_gather实例中包含一个mmu\_gather\_batch实例链表，见下文）。

unmap\_region()函数调用关系简列如下图所示：



##### ■收集映射页

内核定义了mmu\_gather结构体用于收集页帧page实例，unmap\_region()函数中扫描各VMA对应PTE页表项，获取映射页page实例交由mmu\_gather实例缓存，扫描结束后统一将缓存的映射页释放。

mmu\_gather结构体定义在/include/asm-generic/tlb.h头文件，用于缓存页帧page实例：

struct mmu\_gather {

struct mm\_struct \*mm; /\*地址空间实例指针\*/

#ifdef CONFIG\_HAVE\_RCU\_TABLE\_FREE

struct mmu\_table\_batch \*batch;

#endif

unsigned long start; /\*映射区域起始虚拟地址\*/

unsigned long end; /\*映射区域结束虚拟地址\*/

unsigned int fullmm : 1, /\*是否包括整个地址空间\*/

need\_flush\_all : 1; /\*需要刷新整个TLB\*/

struct mmu\_gather\_batch **\*active;**  /\*指向当前活跃mmu\_gather\_batch实例\*/

struct mmu\_gather\_batch **local**; /\*内嵌mmu\_gather\_batch实例，后接page指针数组\*/

struct page **\*\_\_pages[MMU\_GATHER\_BUNDLE]**; /\*数组项数为8\*/

unsigned int batch\_count; /\*链表中mmu\_gather\_batch实例数量（不含local实例）\*/

};

#define MMU\_GATHER\_BUNDLE 8

mmu\_gather结构体中最主要的成员是包含一个mmu\_gather\_batch结构体实例链表，内嵌的local实例是链表中第一个成员，batch\_count表示链表中mmu\_gather\_batch实例的数量（不含local实例）。

mmu\_gather\_batch结构体定义在/include/asm-generic/tlb.h头文件，其中包含page实例指针数组，用于缓存页帧page实例：

struct mmu\_gather\_batch {

**struct mmu\_gather\_batch \*next;** /\*指向下一实例\*/

unsigned int nr; /\*当前实例缓存页数量\*/

unsigned int max; /\*当前实例缓存页最大数量，即pages指针数组项数\*/

**struct page \*pages[0]**; /\*page指针数组，必须是最后一个成员\*/

};

mmu\_gather\_batch实例在内核中是动态分配的，并且是直接从伙伴系统分配一个页帧来保存结构体实例，最后一个成员pages实际上是一个page指针数组，也就是说分配页帧内除了用于保存结构体前三个成员的空间外，其它的都用于page指针数组。

mmu\_gather结构体中\_\_pages[]成员表示内嵌的local实例包含的指针数组（8项）。

以上数据结构组织关系如下图所示：



向mmu\_gather实例中添加page实例时，总是从mmu\_gather\_batch链表的第一个实例开始，填满一个实例的指针数组后再添加到下一下实例的指针数组。mmu\_gather结构体中active成员指向链表中当前活跃的（可添加page实例的）mmu\_gather\_batch实例。

###### ●初始化mmu\_gather实例

内核在/mm/memory.c文件内定义了初始化mmu\_gather实例的函数：

void tlb\_gather\_mmu(struct mmu\_gather \*tlb, struct mm\_struct \*mm, unsigned long start, \

unsigned long end)

{

tlb->mm = mm; /\*指向地址空间\*/

tlb->fullmm = !(start | (end+1)); /\*地址范围是0至~0（包含整个地址空间），置1\*/

tlb->need\_flush\_all = 0;

tlb->local.next = NULL;

tlb->local.nr = 0;

**tlb->local.max = ARRAY\_SIZE(tlb->\_\_pages)**; /\*8个指针数组项\*/

**tlb->active = &tlb->local**; /\*指向内嵌mmu\_gather\_batch实例\*/

tlb->batch\_count = 0;

#ifdef CONFIG\_HAVE\_RCU\_TABLE\_FREE

tlb->batch = NULL;

#endif

\_\_tlb\_reset\_range(tlb); /\*/include/asm-generic/tlb.h\*/

/\*根据tlb->fullmm值设置mmu\_gather起始结束虚拟地址\*/

}

###### ●添加page实例

向mmu\_gather实例添加页帧page实例的函数为\_\_tlb\_remove\_page()，定义如下（/mm/memory.c）：

int **\_\_tlb\_remove\_page**(struct mmu\_gather \*tlb, struct page \*page)

{

struct mmu\_gather\_batch \*batch;

VM\_BUG\_ON(!tlb->end);

batch = tlb->active;

batch->pages[batch->nr++] = page;

if (batch->nr == batch->max) {

if (!**tlb\_next\_batch(tlb)**) /\*获取（分配）下一个mmu\_gather\_batch实例，/mm/memory.c\*/

return 0;

**batch = tlb->active**;

}

VM\_BUG\_ON\_PAGE(batch->nr > batch->max, page);

return batch->max - batch->nr; /\*返回当前活跃mmu\_gather\_batch实例指针数组空闲项数量\*/

}

**\_\_tlb\_remove\_page**()函数内由mmu\_gather实例active成员获取当前活跃的mmu\_gather\_batch实例，直接将page实例关联到其指针数组，如果添加page后指针数组被填满，则调用tlb\_next\_batch(tlb)获取或分配下一个活跃的mmu\_gather\_batch实例。注意，分配实例的操作是直接从伙伴系统中分配一个物理页帧，用于构建mmu\_gather\_batch实例，最后函数返回当前mmu\_gather\_batch实例指针数组空闲项的数目。

另外，tlb\_remove\_page()函数也用于向mmu\_gather实例添加page实例（/include/asm-generic/tlb.h）：

static inline void tlb\_remove\_page(struct mmu\_gather \*tlb, struct page \*page)

{

if (!\_\_tlb\_remove\_page(tlb, page)) /\*返回0时，释放缓存页\*/

tlb\_flush\_mmu(tlb); /\*释放mmu\_gather实例中缓存页\*/

}

tlb\_remove\_page()函数中调用\_\_tlb\_remove\_page(tlb, page)函数向mmu\_gather实例添加page实例，当\_\_tlb\_remove\_page()函数返回0时，即分配下一个mmu\_gather\_batch实例不成功时，将释放mmu\_gather实例中缓存的页。

##### **■断开映射**

unmap\_vmas()函数扫描解除映射区域对应的页表项，读取PTE页表项内容后对其清零，从PTE页表项中数据获取映射页帧page实例，添加到mmu\_gather实例。

unmap\_vmas()定义如下（/mm/memory.c）：

void unmap\_vmas(struct mmu\_gather \*tlb, \

struct vm\_area\_struct \*vma, unsigned long start\_addr,unsigned long end\_addr)

/\*vma：指向VMA链表中首个VMA\*/

{

struct mm\_struct \*mm = vma->vm\_mm;

mmu\_notifier\_invalidate\_range\_start(mm, start\_addr, end\_addr);

for ( ; vma && vma->vm\_start < end\_addr; vma = vma->vm\_next) /\*遍历解除映射VMA\*/

**unmap\_single\_vma(tlb, vma, start\_addr, end\_addr, NULL)**; /\*/mm/memory.c\*/

mmu\_notifier\_invalidate\_range\_end(mm, start\_addr, end\_addr);

}

unmap\_vmas()函数内遍历解除映射VMA链表，对每个VMA调用unmap\_single\_vma()函数，函数代码如下：

static void unmap\_single\_vma(struct mmu\_gather \*tlb,struct vm\_area\_struct \*vma, unsigned long start\_addr,

unsigned long end\_addr,struct zap\_details \*details)

/\*details参数为NULL\*/

{

unsigned long start = **max(vma->vm\_start, start\_addr)**; /\*VMA起始地址\*/

unsigned long end;

if (start >= vma->vm\_end)

return;

**end = min(vma->vm\_end, end\_addr)**; /\*VMA结束地址\*/

if (end <= vma->vm\_start)

return;

if (vma->vm\_file) /\*处理文件映射\*/

uprobe\_munmap(vma, start, end);

/\*没有选择UPROBES配置选项为空操作，/include/linux/uprobes.h\*/

if (unlikely(vma->vm\_flags & VM\_PFNMAP)) /\*处理PFN映射\*/

untrack\_pfn(vma, 0, 0); /\*没有定义\_\_HAVE\_PFNMAP\_TRACKING则为空操作\*/

if (start != end) {

if (unlikely(is\_vm\_hugetlb\_page(vma))) { /\*巨型TLB映射VMA\*/

if (vma->vm\_file) {

i\_mmap\_lock\_write(vma->vm\_file->f\_mapping);

\_\_unmap\_hugepage\_range\_final(tlb, vma, start, end, NULL);

i\_mmap\_unlock\_write(vma->vm\_file->f\_mapping);

}

} else

**unmap\_page\_range(tlb, vma, start, end, details)**; /\*details为NULL，/mm/memory.c\*/

}

}

对于普通映射VMA，最后调用unmap\_page\_range()函数进行处理，函数定义在/mm/memory.c文件内：

static void unmap\_page\_range(struct mmu\_gather \*tlb,struct vm\_area\_struct \*vma, \

unsigned long addr, unsigned long end, struct zap\_details \*details)

/\*details参数为NULL\*/

{

pgd\_t \*pgd;

unsigned long next;

if (details && !details->check\_mapping)

details = NULL;

BUG\_ON(addr >= end);

tlb\_start\_vma(tlb, vma); /\*如有需要刷新CPU缓存，/arch/mips/include/asm/tlb.h\*/

pgd = pgd\_offset(vma->vm\_mm, addr);

do {

next = pgd\_addr\_end(addr, end); /\*下一个PGD页表项表示的起始虚拟地址\*/

if (pgd\_none\_or\_clear\_bad(pgd)) /\*PGD页表项是否为空，/include/asm-generic/pgtable.h\*/

continue;

next = **zap\_pud\_range(tlb, vma, pgd, addr, next, details)**; /\*/mm/memory.c\*/

} while (pgd++, addr = next, addr != end);

tlb\_end\_vma(tlb, vma); /\*空操作\*/

}

unmap\_page\_range()函数逐级扫描各级页表，zap\_pud\_range()函数最终调用zap\_pte\_range()函数，获取VMA对应PTE页表项并清零，根据PTE页表项获取映射页帧page实例，并添加到mmu\_gather实例。

zap\_pte\_range()函数定义如下（/mm/memory.c）:

static unsigned long zap\_pte\_range(struct mmu\_gather \*tlb,struct vm\_area\_struct \*vma, pmd\_t \*pmd, \

unsigned long addr, unsigned long end,struct zap\_details \*details)

{

struct mm\_struct \*mm = tlb->mm;

int force\_flush = 0;

int rss[NR\_MM\_COUNTERS]; /\*映射页类型统计量\*/

spinlock\_t \*ptl;

pte\_t \*start\_pte;

pte\_t \*pte;

swp\_entry\_t entry;

**again:**

init\_rss\_vec(rss); /\*初始化统计量\*/

**start\_pte = pte\_offset\_map\_lock(mm, pmd, addr, &ptl)**; /\*起始PTE页表项指针\*/

**pte = start\_pte**; /\*指向PTE页表项\*/

arch\_enter\_lazy\_mmu\_mode();

do { /\*遍历PTE页表项\*/

pte\_t ptent = \*pte;

if (pte\_none(ptent)) { /\*PTE页表项为空，跳过\*/

continue;

}

/\*处理映射页在内存中的情况\*/

if (pte\_present(ptent)) {

struct page \*page;

page = **vm\_normal\_page(vma, addr, ptent)**; /\*返回普通映射页page实例指针\*/

if (unlikely(details) && page) { /\*details参数为NULL\*/

if (details->check\_mapping &&details->check\_mapping != page->mapping)

continue;

}

**ptent = ptep\_get\_and\_clear\_full(mm, addr, pte,tlb->fullmm)**;

/\*读取原PTE页表项内容，并对其清空，/include/asm-generic/pgtable.h\*/

**tlb\_remove\_tlb\_entry(tlb, pte, addr)**;

/\*调整tlb起始结束地址，/include/asm-generic/tlb.h\*/

if (unlikely(!page)) /\*page为NULL，跳过\*/

continue;

if (PageAnon(page)) /\*匿名映射页\*/

rss[MM\_ANONPAGES]--; /\*匿名映射页数量减1\*/

else { /\*文件映射页\*/

if (pte\_dirty(ptent)) { /\*脏页\*/

**force\_flush = 1**;

**set\_page\_dirty(page)**; /\*设置页脏（需执行回写）\*/

}

if (pte\_young(ptent) &&likely(!(vma->vm\_flags & VM\_SEQ\_READ)))

**mark\_page\_accessed(page)**;

rss[MM\_FILEPAGES]--;

}

**page\_remove\_rmap(page)**; /\*映射计数减1，/mm/rmap.c\*/

if (unlikely(page\_mapcount(page) < 0)) /\*映射计数不能小于0\*/

print\_bad\_pte(vma, addr, ptent, page);

if (unlikely(!**\_\_tlb\_remove\_page(tlb, page)**)) { /\*将page实例添加到mmu\_gather实例\*/

/\*此处需要处理的情况是mmu\_gather\_batch实例创建不成功，

\*则释放mmu\_gather实例中缓存页后，再添加page实例。

\*/

force\_flush = 1;

addr += PAGE\_SIZE;

break;

}

continue; /\*扫描下一个PTE页表项\*/

} /\*if (pte\_present(ptent))结束\*/

/\*处理映射页不在内存中的情况，如映射未建立，映射页在交换缓存中等\*/

if (unlikely(details))

continue;

**entry** = **pte\_to\_swp\_entry(ptent);**

if (!non\_swap\_entry(entry))

rss[MM\_SWAPENTS]--;

else if (is\_migration\_entry(entry)) {

struct page \*page;

page = migration\_entry\_to\_page(entry);

if (PageAnon(page))

rss[MM\_ANONPAGES]--;

else

rss[MM\_FILEPAGES]--;

}

if (unlikely(!**free\_swap\_and\_cache(entry)**)) /\*释放交换缓存中的页\*/

print\_bad\_pte(vma, addr, ptent, NULL);

**pte\_clear\_not\_present\_full**(mm, addr, pte, tlb->fullmm); /\*清空PTE页表项\*/

} while (**pte++, addr += PAGE\_SIZE, addr != end**); /\*遍历PTE页表项结束\*/

**add\_mm\_rss\_vec(mm, rss)**; /\*统计量更新到地址空间实例\*/

arch\_leave\_lazy\_mmu\_mode();

if (force\_flush)

tlb\_flush\_mmu\_tlbonly(tlb); /\*调整mmu\_gather实例起始结束地址等\*/

pte\_unmap\_unlock(start\_pte, ptl);

if (**force\_flush**) {

force\_flush = 0;

**tlb\_flush\_mmu\_free(tlb)**; /\*释放mmu\_gather实例中缓存页，/mm/memory.c\*/

if (addr != end)

**goto again**; /\*继续执行扫描PTE页表项操作\*/

}

return addr;

}

zap\_pte\_range()函数扫描PTE页表项，获取页表项内容并清零页表项，根据页表项内容获取映射页page实例，减小其映射计数，page实例添加到mmu\_gather实例缓存中，扫描完页表项后如果force\_flush为1，则调用函数tlb\_flush\_mmu\_free(tlb)立即释放映射页，如果force\_flush为0则需要等到解除映射函数最后调用tlb\_finish\_mmu()函数时才释放映射页。

##### **■收集空闲页表**

前面unmap\_vmas()函数中已经清零了VMA对应的页表（项），free\_pgtables()函数用于将解除映射VMA从反向映射结构中移出，并将各级页表中的空闲页表对应的page实例添加到mmu\_gather实例，以便在最后一起释放。free\_pgtables()函数定义如下（/mm/memory.c）：

void free\_pgtables(struct mmu\_gather \*tlb, struct vm\_area\_struct \*vma,unsigned long floor,\

unsigned long ceiling)

{

while (vma) { /\*遍历解除映射VMA链表\*/

struct vm\_area\_struct \*next = vma->vm\_next;

unsigned long addr = vma->vm\_start;

**unlink\_anon\_vmas(vma)**; /\*VMA从匿名反向映射结构中移出，/mm/rmap.c\*/

**unlink\_file\_vma(vma);** /\*VMA从文件反向映射结构中移出，/mm/mmap.c\*/

if (is\_vm\_hugetlb\_page(vma)) {

hugetlb\_free\_pgd\_range(tlb, addr, vma->vm\_end,floor, next? next->vm\_start: ceiling);

} else {

while (next && next->vm\_start <= vma->vm\_end + PMD\_SIZE

&& !is\_vm\_hugetlb\_page(next)) {

vma = next;

next = vma->vm\_next;

unlink\_anon\_vmas(vma);

unlink\_file\_vma(vma);

}

**free\_pgd\_range(tlb, addr, vma->vm\_end,floor, next? next->vm\_start: ceiling)**;

/\*扫描VMA各级页表，空闲页表page实例添加到mmu\_gather实例，/mm/memory.c\*/

}

vma = next; /\*下一个VMA\*/

}

}

free\_pgtables()函数扫描解除VMA链表，对每个VMA将其从反向映射结构中移出，扫描VMA对应的各级页表，如果页表为空将最后通过tlb\_remove\_page()函数将空闲页表对应page实例添加到mmu\_gather实例中。

##### ■释放映射页

unmap\_region()函数调用**tlb\_finish\_mmu()**函数释放mmu\_gather实例缓存的页以及mmu\_gather\_batch实例占用的物理页帧。

tlb\_finish\_mmu()函数定义如下（/mm/memory.c）：

void tlb\_finish\_mmu(struct mmu\_gather \*tlb, unsigned long start, unsigned long end)

{

struct mmu\_gather\_batch \*batch, \*next;

**tlb\_flush\_mmu(tlb)**; /\*调用tlb\_flush\_mmu\_free(tlb)函数，/mm/memory.c\*/

check\_pgt\_cache();

for (batch = tlb->local.next; batch; batch = next) { /\*遍历mmu\_gather\_batch实例链表\*/

next = batch->next;

**free\_pages((unsigned long)batch, 0)**; /\*释放mmu\_gather\_batch实例占用页帧\*/

}

**tlb->local.next = NULL**; /\*链表清空\*/

}

tlb\_finish\_mmu()函数中调用tlb\_flush\_mmu(tlb)函数释放实例中缓存的页，调用free\_pages()函数释放mmu\_gather\_batch实例占用页帧。

下面看一下tlb\_flush\_mmu(tlb)函数的定义（/mm/memory.c）：

void tlb\_flush\_mmu(struct mmu\_gather \*tlb)

{

tlb\_flush\_mmu\_tlbonly(tlb); /\*刷新TLB，/mm/memory.c\*/

**tlb\_flush\_mmu\_free(tlb)**; /\*释放缓存页，/mm/memory.c\*/

}

tlb\_flush\_mmu\_free(tlb)函数，释放mmu\_gather实例中缓存的页，函数定义如下（/mm/memory.c）：

static void tlb\_flush\_mmu\_free(struct mmu\_gather \*tlb)

{

struct mmu\_gather\_batch \*batch;

for (batch = &tlb->local; batch && batch->nr; batch = batch->next) {

**free\_pages\_and\_swap\_cache(batch->pages, batch->nr)**; /\*/mm/swap\_state.c\*/

**batch->nr = 0**; /\*缓存page实例数量清零\*/

}

**tlb->active = &tlb->local**; /\*返回初始状态\*/

}

tlb\_flush\_mmu\_free()函数内遍历mmu\_gather实例中mmu\_gather\_batch实例链表，对每个实例调用函数free\_pages\_and\_swap\_cache()释放pages指针数组缓存的页。free\_pages\_and\_swap\_cache()函数具体实现到第11章再做介绍。执行完释放函数后mmu\_gather实例中的mmu\_gather\_batch实例链表依然存在没有释放，只不过active成员又指向了链表中第一个成员。tlb\_finish\_mmu()函数随后会释放mmu\_gather\_batch实例。

至此，munmap()系统调用通过unmap\_region()函数解除了各VMA的映射，最后将调用remove\_vma\_list()函数释放VMA实例，解除映射操作大功告成！

## 4.7缺页异常处理

在进程地址空间创建映射时，如果系统调用flags标记参数未设置MAP\_POPULATE或MAP\_LOCKED标记位，或地址空间VMA默认标记未设置VM\_LOCKED标记位，则在创建映射时只为映射区创建VMA等数据结构实例，而并没有分配物理内帧，也没有生成内存页表项，映射还没有真正建立。

在创建进程时，进程地址空间未建立映射的PTE表项都写入了无效的表项。CPU访问未建立映射的虚拟地址时将先引发TLB重填异常，重填异常处理程序会将无效的（但与虚拟地址匹配）页表项写入TLB，重填异常返回。重填异常返回后，虚拟地址将匹配到无效的TLB表项，随后将触发TLB缺页（无效）异常，在缺页异常处理程序中建立虚拟页到物理页的映射。

TLB缺页异常不仅包括TLB无效异常，还包括因访问权限不匹配引发的异常等。TLB缺页异常处理程序需要处理这些页表项不可用的情形，详见第6章。

### 4.7.1缺页异常

#### 1异常的产生

缺页异常就是在TLB中没有找到虚拟地址匹配的TLB项，或者有匹配的表项，但是访问权限不匹配时，而产生的TLB异常。



如上图所示，我们用图示的方式先来阐述一下缺页异常在何时产生。

当CPU在执行内核线程时，使用的是内核页表，在内核地址空间创建映射时，立即分配物理页帧，生成并写入内核页表项，建立映射。当TLB中没有CPU访问间接映射区虚拟地址匹配的表项时，将触发TLB重填异常，在重填异常处理程序中只需要根据虚拟页号查询内核页表，将查找到的页表项填充至TLB，异常即可返回。如果匹配内核页表项是无效的，说明CPU访问了非法的地址，因为内核空间创建映射和建立映射是同时进行的，正常的映射区不可能不存在映射。

当CPU在执行用户进程时，使用的是进程页表，这时缺页异常的情况要复杂一些，主要分以下几种情况：

（1）用户进程通过系统调用或中断进入内核空间运行时，此时使用的是进程页表而非内核页表。当TLB中没有与内核空间地址匹配的TLB表项时，TLB重填异常处理程序将到用户进程页表中查找页表项填充至TLB。

在pgd\_alloc()函数中为进程创建PGD页表时，将复制内核PGD页表中内核空间对应的表项至进程页表，也就是说进程与内核共用PGD以下的各级页表。但是，如果内核PGD页表项是在进程创建之后建立映射的，则此时尚未同步到进程页表。TLB重填异常处理程序将填充无效的页表项至TLB，随后又将触发TLB无效异常，在此异常处理程序中需要将内核PGD页表项内容复制到用户进程PGD页表（同步）。异常返回后，将再次触发TLB无效异常，此时就会将可用的页表项写入TLB。

注意：除用户态TLB重填异常外，其它任何TLB异常处理程序，都会先检查内存页表项是否可用，若可用则写入TLB，异常返回；若不可用再处理缺页异常的情况（详见第6章）。

（2）CPU访问未建立映射的匿名映射地址时，异常处理程序中需要分配页帧，修改对应页表项，写入TLB，随后就可以进行正常的地址转换了。如果匿名映射页数据在交换区中，则还需要将交换区中页数据读入分配的页帧。

（3）CPU访问未建立映射的文件映射页时，从文件地址空间中查找或创建文件内容的缓存页（从块设备中读取数据），依缓存页生成进程页表项，写入TLB，随后可以正常地进行地址转换了。

（4）当对CPU写保护（写时复制）页进行写操作时，需要分配新页帧，复制原映射页帧数据至新页帧，依新页帧修改页表项，写入TLB，然后就可以对页进行正常的写操作了。

#### 2异常的类型

下面以MIPS32体系结构为例，详细说明TLB异常的类型，以及产生的条件。

MIPS32处理器产生的程序（虚拟地址）转换成物理地址的过程如下图所示，处理器以虚拟地址页帧号VPN2（2代表2个物理页帧）为索引在MMU硬件TLB中查找匹配的项，如果有匹配的项（VPN2匹配），并且访问权限也匹配，则使用TLB页表项中的物理页帧号PFN替换虚拟地址中的虚拟页帧号，页内偏移量不变，合成物理地址输出到地址总线。



如果TLB中没有虚拟地址匹配的TLB表项（虚拟页帧号或ASID标识不匹配等），将触发TLB重填异常。这是一个频繁发生的异常，因此具有专用的异常处理程序（异常向量）。TLB重填异常处理程序中从进程（内核）页表中查找虚拟地址对应的页表项（不检查有效性），填充至TLB（页表项可能无效或访问权限不匹配），异常返回。

如是TLB中有虚拟地址匹配的表项，但访问权限不匹配也将触发异常，例如，读障碍异常、修改异常等。

MIPS32体系结构通过协处理器CP0寄存器操作TLB，包括写入TLB表项，读出TLB表项等，寄存器是操作TLB的窗口。向TLB写入表项时，由程序将内存页表项写入CP0寄存器，然后执行TLB写指令将寄存器中表项内容写入TLB表项中。读出TLB表项时，先执行TLB读指令，TLB表项中内容被传送到CP0寄存器，程序再从寄存器中读取表项数据。

内存页表中的页表项与CP0寄存器中页表项内容不完全一样，它们之间的转换由写入TLB表项的程序完成。CP0寄存器中页表项与TLB中页表项内容也不完全一样，它们之间的转换由处理器硬件完成。

CP0寄存器和TLB页表项中各标记位（位域）语义如下，注意与内存页表项中标记的区分：

**ASID**：页所属进程地址空间ID号，所有进程使用相同的虚拟地址，ASID号用于区分页表项属于哪个进程。

**G**：全局标记位，全局页适用于所有进程，内核空间映射页设置此标记位，对所有进程可用。

**V**：页表项有效性标记，1表示有效，0表示无效，访问此页将触发TLB无效异常。

**D**：脏标记位，1表示映射页可写，0表示不可写，写置0的页将触发TLB修改异常。

**C**：缓存和一致性标记（3位），定义如下：

|  |  |  |
| --- | --- | --- |
| C（5...3） | 描述 | 备注 |
| 0 | 用户定义 | 可选 |
| 1 | 用户定义 | 可选 |
| 2 | **不可缓存** | 必选 |
| 3 | **可缓存** | 必选 |
| 4 | 用户定义 | 可选 |
| 5 | 用户定义 | 可选 |
| 6 | 用户定义 | 可选 |
| 7 | 用户定义 | 可选 |

**RI**：读障碍标记，1表示页不可读，0表示可读，只有在PageGrain寄存器RIE位置1时，写CP0寄存器时才会将RI值写入EntryLo0和EntryLo1寄存器，否则寄存器中RI位始终为0。

**XI**：执行障碍标记，1表示页不可取指令，0表示可取指令，只有在PageGrain寄存器XIE位置1时，写CP0寄存器时才会将XI值写入EntryLo0和EntryLo1寄存器，否则寄存器中XI位始终为0。

MIPS32体系结构TLB异常的类型，发生的条件，异常的名称及异常处理程序的入口地址如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **异常**  **类型** | **发生条件** | **异常名称** | **异常向量地址** |
| TLBL | TLB中没有匹配项，加载数据或取指令，Status(EXL)=0。 | TLB重填异常 | **EBase** |
| TLB中没有匹配项，加载数据或取指令，Status(EXL)=1。 | 异常级TLB重填异常（内核态） | EBase+0x180 |
| TLB中有匹配项，V=0，加载数据或取指令。 | TLB无效异常 |
| TLB中有匹配项，RI=1，  PageGrain(RIE)=1，PageGrain(IEC)=0，加载数据。 | 读障碍异常 |
| TLB中有匹配项，XI=1，  PageGrain(XIE)=1，PageGrain(IEC)=0，取指令。 | 执行障碍异常 |
| TLBS | TLB中没有匹配项，存储数据，Status(EXL)=0。 | TLB重填异常 | **EBase** |
| TLB中没有匹配项，存储数据，Status(EXL)=1。 | 异常级TLB重填异常 | EBase+0x180 |
| TLB中有匹配的项，V=0，存储数据。 | TLB无效异常 |
| TLBM | LB中有匹配的项且有效，存储操作，D=0（页不可写）。 | TLB修改异常 |
| TLBRI | TLB中有匹配项，RI=1，  PageGrain(RIE)=1，PageGrain(IEC)=1，加载数据。 | 读障碍异常 |
| TLBXI | TLB中有匹配项，XI=1，  PageGrain(XIE)=1，PageGrain(IEC)=1，取指令。 | 执行障碍异常 |

注：更详细的信息请参考MIPS32体系结构手册。

#### 3异常处理

MIPS32体系结构规定的各异常处理程序位置（异常向量），如下图所示，其中EBase为异常向量基地址。TLB重填异常处理程序位于EBase基地址处，其它的异常处理程序入口地址都是EBase+0x180。但是，如果中断采用向量模式，则中断也有自已专门的处理程序入口，基地址为EBase+0x200。



内核在初始化函数trap\_init()中将为各异常向量填充代码（指令）。TLB重填异常处理程序代码先保存在final\_handler[]数组中，然后复制到异常向量基地址处。普通异常处理程序为except\_vec3\_generic()，它是一个分配器，根据异常的类型跳转至exception\_handlers[32]数组项中保存的地址处，执行特定于异常类型的代码。在trap\_init()函数中会为各TLB异常处理程序生成并填充代码，并将处理程序入口地址写入exception\_handlers[32]对应数组项中。

TLB重填异常处理程序根据CP0寄存器BadVAddr中保存的引发异常的虚拟地址，逐级搜索当前地址空间PGD页表，将虚拟地址对应的PTE页表项写入TLB，异常返回，如下图所示。



TLBL、TLBS、TLBM异常处理程序包含两个路径，一是快速路径：查找虚拟地址对应的内存页表项，如果还可以继续使用，则重设内存页表项访问权限，将其写入TLB，异常返回。二是慢速路径：虚拟地址对应的内存页表项不存在或不可用，则需要重新分配页帧（写入数据），生成页表项，写入内存页表项和TLB，异常返回。

TLBRI和TLBXI异常处理程序没有快速路径，只有慢速路径。

下表给出了各TLB异常处理程序入口的标号或慢速路径中调用的函数：

|  |  |  |
| --- | --- | --- |
| 异常类型（编号） | 处理程序入口地址 | 备注 |
| TLBL（2） | handle\_tlbl | **tlb\_do\_page\_fault\_0()** |
| ebase | 重填异常 |
| TLBS（3） | handle\_tlbs | **tlb\_do\_page\_fault\_1()** |
| ebase | 重填异常 |
| TLBM（1） | handle\_tlbm | **tlb\_do\_page\_fault\_1()** |
| TLBRI（19） | **tlb\_do\_page\_fault\_0** | /arch/mips/mm/tlb-fault.S |
| TLBXI（20） | **tlb\_do\_page\_fault\_0** | /arch/mips/mm/tlb-fault.S |

TLB重填异常处理程序和快速路径中的程序都是由汇编指令编写的，只涉及对内存页表项和TLB的操作，不涉及对映射页的操作。慢速路径中都是调用tlb\_do\_page\_fault\_0()/tlb\_do\_page\_fault\_1()函数，需要对映射页（页帧），内存页表项和TLB进行操作。

tlb\_do\_page\_fault\_0()和tlb\_do\_page\_fault\_1()函数其实是一个函数，只不过参数不同，定义如下：

.macro tlb\_do\_page\_fault, write /\*write为0表示读，1表示写\*/

NESTED(tlb\_do\_page\_fault\_\write, PT\_SIZE, sp)

**SAVE\_ALL**

/\*保存CPU寄存器信息至进程栈pt\_regs实例，/arch/mips/include/asm/stackframe.h\*/

MFC0 a2, CP0\_BADVADDR /\*a2保存引发异常的虚拟地址，第三个参数\*/

**KMODE**  /\*CPU进入内核态，中断状态不变，/arch/mips/include/asm/stackframe.h\*/

/\*Status：**CU0=1，KSU=00（内核态），EXL=0，IE保持不变**\*/

move a0, sp /\*a0保存pt\_regs实例指针，do\_page\_fault()函数第一个参数\*/

REG\_S a2, PT\_BVADDR(sp) /\*保存引发异常地址至pt\_regs实例\*/

li a1, \write /\*a1保存引发异常的是写操作还是读操作，第二个参数\*/

PTR\_LA ra, **ret\_from\_exception** /\*跳转延迟指令，异常返回地址ret\_from\_exception保存至ra\*/

j **do\_page\_fault**  /\*调用通用的缺页异常处理函数，a0，a1，a2为参数\*/

/\*do\_page\_fault()函数返回后，从ret\_from\_exception处开始执行\*/

END(tlb\_do\_page\_fault\_\write)

.endm

tlb\_do\_page\_fault 0 /\*定义tlb\_do\_page\_fault\_0()函数，读操作异常处理函数\*/

tlb\_do\_page\_fault 1 /\*定义tlb\_do\_page\_fault\_1()函数，写操作异常处理函数\*/

tlb\_do\_page\_fault\_0()函数处理读操作产生的异常，tlb\_do\_page\_fault\_0()函数处理写操作产生的异常。这两个函数内部都调用do\_page\_fault()函数处理缺页异常（不含TLB重填异常），函数的三个参数分别是pt\_regs实例指针、读/写操作、产生异常的虚拟地址，后面将详细介绍do\_page\_fault()函数的实现。

注意：在调用do\_page\_fault()函数前CPU的中断状态没有改变，还是TLB异常前的状态。

### 4.7.2缺页处理函数

程序产生的虚拟地址由硬件将其与TLB中表项进行比较，如果有与虚拟地址匹配的项（没有匹配的项由重填异常处理），但是页表项内容无效或访问权限不匹配等，处理器将触发TLB异常，这里称之为缺页异常。缺页异常处理程序在慢速路径中，根据虚拟地址映射类型及页表项内容，确定采取哪种建立映射的措施。映射建立后写入内存页表项，并写入TLB中，从而处理器就可以正确地访问物理内存了。

触发缺页异常的虚拟地址可能位于用户地址空间，也可能位于内核地址空间。如果是位于内核地址空间（处理器处于核心态），那么只可能是用户进程由于系统调用、中断等进入了内核空间运行，访问了间接映射区，准确地说是VMLLOC区。运行内核线程时，只会产生内核态TLB重填异常，而不会触发缺页异常，除非访问了非法的地址。

这里还需要说明一下，当进程进入内核空间运行时如果访问了固定映射区或持久映射区，为什么不会产生缺页异常？在前面介绍的内核页表初始化中，固定映射区和持久映射区对应的各级页表已经分配好了，只是没有填充PTE表项，创建进程时进程页表中内核地址空间对应的PGD表项，与内核页表是一样的（共享），在固定映射区和持久映射区创建映射时会填充PTE页表项（用户进程自动更新，因为内核、进程的PGD页表项指向相同的PTE页表）。因此，不管是用户进程还是内核线程访问固定映射区或持久映射区产生的TLB重填异常，都能找到正确的PTE页表项填充至TLB，不会产生缺页异常。

触发缺页异常地址位于进程地址空间时（处理器处于用户态），只有位于进程VMA范围内或栈区域以下的地址（扩展栈）才是合法的，空洞区地址是无效（非法）的，这里我们只讨论对合法地址访问产生缺页异常的处理。

进程数据结构task\_struct中包含与缺页异常处理相关的成员：

struct task\_struct {

...

**int pagefault\_disabled;**  /\*是否关闭缺页异常处理，>0关闭，0使能\*/

struct thread\_struct thread;

}

pagefault\_disabled成员值大于0，表示关闭缺页异常处理，等于0表示使能缺页异常处理。

内核在/include/linux/uaccess.h头文件内定义了关闭、使能缺页异常处理的函数及判断是否可执行缺页异常处理的函数：

●void pagefault\_enable(void)：使能缺页异常处理，pagefault\_disabled减1（不能小于0）。

●void pagefault\_disable(void)：关闭缺页异常处理，pagefault\_disabled加1。

●#define pagefault\_disabled() (current->pagefault\_disabled != 0) /\*是否关闭缺页异常处理\*/

●#define **faulthandler\_disabled()** (**pagefault\_disabled()** || in\_atomic()) /\*是否关闭缺页异常处理\*/

#### 1通用处理函数

do\_page\_fault()函数是通用的缺页异常处理函数，主要用于为进程虚拟页分配页帧，建立正确的映射关系，其执行流程简列如下图所示：



缺页处理函数判断异常地址是否位于内核地址空间VMALLOC区，如果是则跳转至vmalloc\_fault处执行，主要工作是将异常地址在内核PGD页表中对应的PGD表项复制到进程页表（同步）。

如果异常地址不是位于VMALLOC区，则应该是位于进程地址空间。处理函数调用find\_vma()函数查找异常地址所处的VMA，如果异常地址不在VMA内部，则再判断是否位于栈区域下方，是则扩展栈，否则地址错误，返回。如果异常地址位于VMA内部，还需要检查VMA标记成员是否规定了相应的读写权限（产生异常的操作权限），例如，没有指定写权限的VMA，其写操作异常将不被处理。如果权限匹配则调用**handle\_mm\_fault()**函数为用户空间虚拟页分配物理页帧，修改页表项，建立映射，建立成功函数返回0，否则返回错误码。

do\_page\_fault()函数在/arch/mips/mm/fault.c文件内实现，代码如下：

asmlinkage void \_\_kprobes do\_page\_fault(struct pt\_regs \*regs,unsigned long write, unsigned long address)

/\*regs：pt\_regs实例指针，保存CPU寄存器信息，write：读/写操作，address：引发异常的地址\*/

{

enum ctx\_state prev\_state;

prev\_state = exception\_enter(); /\*没有配置CONTEXT\_TRACKING选项时返回0\*/

**\_\_do\_page\_fault(regs, write, address)**; /\*/arch/mips/mm/fault.c\*/

exception\_exit(prev\_state); /\*没有配置CONTEXT\_TRACKING选项时为空操作\*/

}

\_\_do\_page\_fault()函数在/arch/mips/mm/fault.c文件内实现，代码如下：

static void \_\_kprobes \_\_do\_page\_fault(struct pt\_regs \*regs, unsigned long write,unsigned long address)

{

struct vm\_area\_struct \* vma = NULL;

struct task\_struct \*tsk = current;

struct mm\_struct **\*mm = tsk->mm**;

const int field = sizeof(unsigned long) \* 2;

**siginfo\_t info**; /\*表示信号属性的结构体，/arch/mips/include/uapi/asm/siginfo.h\*/

int fault;

unsigned int **flags = FAULT\_FLAG\_ALLOW\_RETRY | FAULT\_FLAG\_KILLABLE**; /\*缺页标记\*/

/\*默认设置的标记值传递给handle\_mm\_fault()函数，/include/linux/mm.h\*/

static DEFINE\_RATELIMIT\_STATE(ratelimit\_state, 5 \* HZ, 10);

#ifdef CONFIG\_KPROBES

if (notify\_die(DIE\_PAGE\_FAULT, "page fault", regs, -1, \

(regs->cp0\_cause >> 2) & 0x1f, SIGSEGV) == NOTIFY\_STOP)

return;

#endif

info.si\_code = SEGV\_MAPERR;

#ifdef CONFIG\_64BIT

# define VMALLOC\_FAULT\_TARGET **no\_context**

#else /\*32位系统\*/

# define VMALLOC\_FAULT\_TARGET **vmalloc\_fault** /\*内核空间缺页处理入口地址\*/

#endif

if (unlikely(address >= VMALLOC\_START && address <= VMALLOC\_END))

goto **VMALLOC\_FAULT\_TARGET**; /\*缺页地址为内核空间VMALLOC区\*/

#ifdef MODULE\_START

if (unlikely(address >= MODULE\_START && address < MODULE\_END))

goto VMALLOC\_FAULT\_TARGET;

#endif

/\*以下处理用户地址空间缺页异常\*/

if (**faulthandler\_disabled**() || !mm) /\*中断处理程序或内核线程内的异常，/include/linux/uaccess.h\*/

goto bad\_area\_nosemaphore; /\*出错了\*/

if (user\_mode(regs)) /\*用户空间产生的缺页异常\*/

**flags |= FAULT\_FLAG\_USER**;

retry:

down\_read(&mm->mmap\_sem); /\*获取锁\*/

**vma = find\_vma(mm, address)**; /\*查找address对应的VMA\*/

if (!vma) /\*地址没有对应VMA，访问了非法的地址\*/

goto bad\_area;

if (vma->vm\_start <= address) /\*address位于VMA内部，地址有效\*/

goto **good\_area**; /\*跳转至缺页处理\*/

/\*address不在VMA内部，检查是否需要扩展栈\*/

if (!(vma->vm\_flags & VM\_GROWSDOWN)) /\*address不是位于栈区域，访问了非法的地址\*/

goto bad\_area;

if (**expand\_stack(vma, address)**) /\*address位于栈区域，扩展栈\*/

goto bad\_area;

good\_area:

info.si\_code = SEGV\_ACCERR;

if (write) { /\*如果是写操作产生的异常\*/

if (!(vma->vm\_flags & VM\_WRITE)) /\*如果VMA没有写权限，出错\*/

goto bad\_area;

**flags |= FAULT\_FLAG\_WRITE**; /\*VMA具有写权限\*/

} else { /\*读操作产生的异常\*/

if (cpu\_has\_rixi) { /\*如果处理器具有读、执行障碍标记\*/

if (address == regs->cp0\_epc && !(vma->vm\_flags & VM\_EXEC)) {

...

goto bad\_area;

}

if (!(vma->vm\_flags & VM\_READ) &&exception\_epc(regs) != address) {

...

goto bad\_area;

}

} else { /\*处理器没有读、执行障碍标记，VMA需具有读或写或执行权限\*/

if (!(vma->vm\_flags & (VM\_READ | VM\_WRITE | VM\_EXEC)))

goto bad\_area;

}

}

**fault =** **handle\_mm\_fault(mm, vma, address, flags)**; /\*处理用户空间地址缺页异常，/mm/memory.c\*/

/\*以下是检查用户地址空间缺页处理函数返回结果\*/

if ((fault & VM\_FAULT\_RETRY) && fatal\_signal\_pending(current))

return;

perf\_sw\_event(PERF\_COUNT\_SW\_PAGE\_FAULTS, 1, regs, address);

if (unlikely(fault & **VM\_FAULT\_ERROR**)) {

if (fault & VM\_FAULT\_OOM)

goto **out\_of\_memory**;

else if (fault & VM\_FAULT\_SIGSEGV)

goto bad\_area;

else if (fault & VM\_FAULT\_SIGBUS)

goto **do\_sigbus**;

BUG();

}

if (flags & FAULT\_FLAG\_ALLOW\_RETRY) {

if (fault & VM\_FAULT\_MAJOR) {

perf\_sw\_event(PERF\_COUNT\_SW\_PAGE\_FAULTS\_MAJ, 1,regs, address);

tsk->maj\_flt++;

} else {

perf\_sw\_event(PERF\_COUNT\_SW\_PAGE\_FAULTS\_MIN, 1,regs, address);

tsk->min\_flt++;

}

if (fault & **VM\_FAULT\_RETRY**) {

flags &= ~FAULT\_FLAG\_ALLOW\_RETRY;

flags |= FAULT\_FLAG\_TRIED;

goto **retry;**  /\*重试\*/

}

}

up\_read(&mm->mmap\_sem);

**return**; /\*函数返回\*/

bad\_area:

up\_read(&mm->mmap\_sem);

bad\_area\_nosemaphore: /\*处理用户空间禁止缺页异常的情况，向用户进程发送信号\*/

if (user\_mode(regs)) {

tsk->thread.cp0\_badvaddr = address;

tsk->thread.error\_code = write;

if (show\_unhandled\_signals &&unhandled\_signal(tsk, SIGSEGV) && \_\_ratelimit(&ratelimit\_state)) {

... /\*输出信息\*/

}

info.si\_signo = SIGSEGV;

info.si\_errno = 0;

info.si\_addr = (void \_\_user \*) address;

force\_sig\_info(SIGSEGV, &info, tsk); /\*向进程发送SIGSEGV信号\*/

return;

}

no\_context: /\*内核空间错误地址处理\*/

if (**fixup\_exception(regs)**) { /\*/arch/mips/mm/extable.c\*/

current->thread.cp0\_baduaddr = address;

return;

}

bust\_spinlocks(1);

... /\*输出信息\*/

die("Oops", regs);

out\_of\_memory:

up\_read(&mm->mmap\_sem);

if (!user\_mode(regs))

goto no\_context;

**pagefault\_out\_of\_memory()**; /\*空闲内存不足，触发OOM机制，/mm/oom\_kill.c\*/

return;

do\_sigbus: /\*向用户进程发送信号\*/

up\_read(&mm->mmap\_sem);

if (!user\_mode(regs))

goto no\_context;

else

... /\*输出信息\*/

tsk->thread.cp0\_badvaddr = address;

info.si\_signo = SIGBUS;

info.si\_errno = 0;

info.si\_code = BUS\_ADRERR;

info.si\_addr = (void \_\_user \*) address;

force\_sig\_info(SIGBUS, &info, tsk); /\*向进程发送SIGBUS信号\*/

return;

#ifndef CONFIG\_64BIT

**vmalloc\_fault**: /\*内核空间VMALLOC区缺页处理，同步进程页表和内核页表\*/

{

int offset = \_\_pgd\_offset(address);

pgd\_t \*pgd, \*pgd\_k;

pud\_t \*pud, \*pud\_k;

pmd\_t \*pmd, \*pmd\_k;

pte\_t \*pte\_k;

**pgd = (pgd\_t \*) pgd\_current[raw\_smp\_processor\_id()] + offset**; /\*当前进程PGD页表项\*/

pgd\_k = **init\_mm.pgd** + offset; /\*内核PGD页表项\*/

if (!pgd\_present(\*pgd\_k))

goto no\_context;

**set\_pgd(pgd, \*pgd\_k)**; /\*复制内核PGD页表项至用户进程PGD页表\*/

pud = pud\_offset(pgd, address);

pud\_k = pud\_offset(pgd\_k, address);

if (!pud\_present(\*pud\_k))

goto no\_context;

/\*复制PMD页表项\*/

pmd = pmd\_offset(pud, address);

pmd\_k = pmd\_offset(pud\_k, address);

if (!pmd\_present(\*pmd\_k))

goto no\_context;

**set\_pmd(pmd, \*pmd\_k);**

**pte\_k = pte\_offset\_kernel(pmd\_k, address)**; /\*检检PTE页表项是否有效\*/

if (!pte\_present(\*pte\_k))

goto no\_context;

return; /\*没有写入TLB，再次触发TLB异常后会写入TLB\*/

}

#endif

}

\_\_do\_page\_fault()函数首先判断产生异常的地址是否位于内核地址空间VMALLOC区，如果是则跳转至vmalloc\_fault处同步进程页表和内核页表。如果内核空间缺页异常发生在VMALLOC区之外的区域则由fixup\_exception(regs)函数处理。

缺页异常产生在用户空间时，有两种情况是正常的，一是异常地址位于栈下方（栈向下生长），此时需要先扩展栈，然后再进行正常的用户空间缺页处理，二是异常地址位于某个VMA内部，直接进行用户空间缺页处理。

expand\_stack()函数用于扩展栈，**handle\_mm\_fault()**函数用于处理用户空间缺页，处理成功函数返回0，否则返回错误码。缺页异常处理函数根据其返回值还需要做进一步的处理，例如，处理不成功时向用户进程发送信号等。

下面介绍扩展栈expand\_stack()函数的实现，下一节将单独介绍用户空间缺页的处理函数。

#### 2扩展栈

在缺页异常处理函数中，如果异常地址位于栈区域下部（栈向下生长），则需要扩展栈，即扩展表示栈的VMA实例。

expand\_stack(vma, address)函数用于扩展栈，参数vma表示栈区域第一个结束地址在address之后的VMA实例指针，address表示异常地址。

扩展栈函数expand\_stack()在/mm/mmap.c文件内实现：

int expand\_stack(struct vm\_area\_struct \*vma, unsigned long address)

{

struct vm\_area\_struct \*prev;

address &= PAGE\_MASK; /\*地址页对齐，下对齐\*/

prev = vma->vm\_prev;

if (prev && prev->vm\_end == address) { /\*如果与前一VMA重叠返回错误码\*/

if (!(prev->vm\_flags & VM\_GROWSDOWN))

return -ENOMEM;

}

return **expand\_downwards(vma, address)**; /\*向下扩展VMA，/mm/mmap.c\*/

}

expand\_downwards(vma, address)函数在/mm/mmap.c文件内实现，主要是对vma实例向下扩展，使其包含指定地址address，函数代码如下：

int expand\_downwards(struct vm\_area\_struct \*vma, unsigned long address)

{

int error;

if (unlikely(**anon\_vma\_prepare(vma)**)) /\*若匿名反向映射结构不存在，则创建\*/

return -ENOMEM;

address &= PAGE\_MASK;

error = security\_mmap\_addr(address);

if (error)

return error;

vma\_lock\_anon\_vma(vma);

if (address < vma->vm\_start) { /\*扩展vma\*/

unsigned long size, grow;

size = vma->vm\_end - address;

grow = (vma->vm\_start - address) >> PAGE\_SHIFT;

error = -ENOMEM;

if (grow <= vma->vm\_pgoff) {

error = acct\_stack\_growth(vma, size, grow);

if (!error) {

spin\_lock(&vma->vm\_mm->page\_table\_lock);

**anon\_vma\_interval\_tree\_pre\_update\_vma(vma)**; /\*从反向映射结构中移出\*/

vma->vm\_start = address; /\*调整VMA大小\*/

vma->vm\_pgoff -= grow;

**anon\_vma\_interval\_tree\_post\_update\_vma(vma)**; /\*重新插入反向映射结构\*/

**vma\_gap\_update(vma)**; /\*更新扩展红黑树信息\*/

spin\_unlock(&vma->vm\_mm->page\_table\_lock);

perf\_event\_mmap(vma);

}

}

}

vma\_unlock\_anon\_vma(vma);

khugepaged\_enter\_vma\_merge(vma, vma->vm\_flags);

validate\_mm(vma->vm\_mm);

return error;

}

expand\_downwards()函数实现比较简单，栈区域是匿名映射区，首先要保证vma匿名反向映射数据结构存在，不存在则创建，随后将vma从反向映射结构中移出，待调整完VMA大小和偏移量后重新插入反向映射结构中。

扩展栈之后，还需要继续调用下面介绍的handle\_mm\_fault()函数，为异常地址address所在页分配页帧建立映射。

## 4.8用户空间缺页处理

至此，在用户空间产生的缺页异常，已经可以保证位于某个VMA内部。缺页处理函数随后调用用户空间缺页处理函数**handle\_mm\_fault()**，处理用户空间缺页。

用户地址空间产生缺页异常的情形有：

（1）映射页不在内存中，页表项为空，需要重新创建匿名映射页或文件映射页。

（2）映射页不在内存中，页表项不为空，表示页数据被写入到交换区（匿名映射页），此时需要分配页帧，从交换区中读回数据，建立映射。

（3）映射页在内存中，但具有写保护（异常由写操作触发），需要采用COW（写时复制）机制，分配新页帧，复制映射页数据至新页帧，建立映射。

### 4.8.1缺页处理函数

在介绍用户空间缺页处理函数前，先了解一下该函数的标记参数和返回值，标记参数flags用于指示缺页异常发生的属性，各标记位定义在/include/linux/mm.h头文件：

#define FAULT\_FLAG\_WRITE 0x01 /\*写操作触发的异常\*/

#define FAULT\_FLAG\_MKWRITE 0x02 /\*写时复制触发的异常\*/

#define FAULT\_FLAG\_ALLOW\_RETRY 0x04 /\*阻塞时重试\*/

#define FAULT\_FLAG\_RETRY\_NOWAIT 0x08 /\*等待时不要释放mmap\_sem信号量\*/

#define FAULT\_FLAG\_KILLABLE 0x10 /\* The fault task is in SIGKILL killable region \*/

#define FAULT\_FLAG\_TRIED 0x20 /\*第二次重试\*/

#define FAULT\_FLAG\_USER 0x40 /\*用户地址空间异常\*/

函数返回值定义在/include/linux/mm.h头文件内，表示函数执行结果或不成功的原因等：

#define VM\_FAULT\_MINOR 0 /\* For backwards compat. Remove me quickly. \*/

#define VM\_FAULT\_OOM 0x0001 /\*空闲物理内存严重不足，需要启动OOM机制\*/

#define VM\_FAULT\_SIGBUS 0x0002

#define VM\_FAULT\_MAJOR 0x0004

#define VM\_FAULT\_WRITE 0x0008 /\*get\_user\_pages()函数返回值\*/

#define VM\_FAULT\_HWPOISON 0x0010 /\*Hit poisoned small page \*/

#define VM\_FAULT\_HWPOISON\_LARGE 0x0020 /\* \*/

#define VM\_FAULT\_SIGSEGV 0x0040

#define VM\_FAULT\_NOPAGE 0x0100 /\*修改PTE页表项，没有page实例\*/

#define VM\_FAULT\_LOCKED 0x0200 /\*锁定映射页\*/

#define VM\_FAULT\_RETRY 0x0400 /\*需要重试\*/

#define VM\_FAULT\_FALLBACK 0x0800 /\*巨型页处理失败\*/

#define VM\_FAULT\_HWPOISON\_LARGE\_MASK 0xf000 /\*\*/

#define VM\_FAULT\_ERROR (VM\_FAULT\_OOM | VM\_FAULT\_SIGBUS | VM\_FAULT\_SIGSEGV | \

VM\_FAULT\_HWPOISON | VM\_FAULT\_HWPOISON\_LARGE |VM\_FAULT\_FALLBACK)

/\*错误码集合\*/

**handle\_mm\_fault()**函数定义在/mm/memory.c文件内，代码如下：

int handle\_mm\_fault(struct mm\_struct \*mm, struct vm\_area\_struct \*vma,unsigned long address, \

unsigned int **flags**)

/\*flags：标记参数\*/

{

int ret;

\_\_set\_current\_state(TASK\_RUNNING); /\*设置进程状态为可运行\*/

count\_vm\_event(PGFAULT);

mem\_cgroup\_count\_vm\_event(mm, PGFAULT);

check\_sync\_rss\_stat(current);

if (flags & **FAULT\_FLAG\_USER**) /\*用户地址空间产生的异常\*/

mem\_cgroup\_oom\_enable();

ret = **\_\_handle\_mm\_fault(mm, vma, address, flags)**; /\*/mm/memory.c\*/

if (flags & **FAULT\_FLAG\_USER**) {

mem\_cgroup\_oom\_disable();

if (task\_in\_memcg\_oom(current) && !(ret & VM\_FAULT\_OOM))

mem\_cgroup\_oom\_synchronize(false);

}

return ret;

}

handle\_mm\_fault()函数内调用\_\_handle\_mm\_fault()函数处理缺页异常，函数定义在/mm/memory.c文件内：

static int \_\_handle\_mm\_fault(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long address, unsigned int flags)

{

pgd\_t \*pgd;

pud\_t \*pud;

pmd\_t \*pmd;

pte\_t \*pte;

if (unlikely(is\_vm\_hugetlb\_page(vma)))

return hugetlb\_fault(mm, vma, address, flags);

pgd = pgd\_offset(mm, address); /\*创建各级页表\*/

pud = pud\_alloc(mm, pgd, address);

if (!pud)

return VM\_FAULT\_OOM;  /\*创建页表不成功，物理内存严重不足\*/

pmd = pmd\_alloc(mm, pud, address);

if (!pmd)

return VM\_FAULT\_OOM; /\*创建页表不成功，物理内存严重不足\*/

if (pmd\_none(\*pmd) && transparent\_hugepage\_enabled(vma)) {

... /\*忽略巨型页情形\*/

} else {

pmd\_t orig\_pmd = \*pmd;

int ret;

barrier();

if (pmd\_trans\_huge(orig\_pmd)) {

... /\*忽略巨型页情形\*/

}

}

if (**unlikely(pmd\_none(\*pmd)) &&unlikely(\_\_pte\_alloc(mm, vma, pmd, address))**)

return VM\_FAULT\_OOM; /\*创建PTE页表不成功，表示物理内存严重不足\*/

if (unlikely(pmd\_trans\_huge(\*pmd)))

return 0;

**pte = pte\_offset\_map(pmd, address)**; /\*address对应PTE页表项指针\*/

return **handle\_pte\_fault(mm, vma, address, pte, pmd, flags)**; /\*根据PTE页表项情况进行处理\*/

}

\_\_handle\_mm\_fault()函数如果忽略掉对巨型TLB页的处理后，实现比较简单，函数内逐级查找address对应的页表，如果哪级页表不存在则创建页表，若创建页表不成功说明空闲物理内存严重不足，函数返回错误码VM\_FAULT\_OOM。

最后查找到address对应的PTE页表项，据此调用**handle\_pte\_fault()**函数完成对缺页的处理。address对应的PTE页表项可能为空，表示映射尚未创建，也可能已经存在，但因访问权限不匹配或映射页被交换出交换区等原因触发了缺页异常。

下面来看handle\_pte\_fault()函数如何根据PTE页表项的内容处理各种情况（/mm/memory.c）：

static int handle\_pte\_fault(struct mm\_struct \*mm,struct vm\_area\_struct \*vma, unsigned long address, \

pte\_t \*pte, pmd\_t \*pmd, unsigned int flags)

{

pte\_t entry;

spinlock\_t \*ptl;

**entry = \*pte**; /\*PTE页表项指针\*/

barrier();

if (!pte\_present(entry)) { /\*映射页不在内存中（可在交换区），PTE页表项P标记为0\*/

if (pte\_none(entry)) { /\*PTE页表项为空\*/

if (vma->vm\_ops) /\*文件映射\*/

return **do\_fault(mm, vma, address, pte, pmd,flags, entry)**; /\*创建文件缓存页\*/

return **do\_anonymous\_page(mm, vma, address, pte, pmd,flags)**; /\*创建匿名映射页\*/

}

/\*PTE页表项不为空，页被交换到交换区\*/

return **do\_swap\_page(mm, vma, address,pte, pmd, flags, entry)**;

/\*处理换出到交换区的匿名映射页，见第11章\*/

}

/\*映射页在内存中（P=1），但访问权限不匹配，或页表项无效V=0\*/

if (pte\_protnone(entry)) /\*返回0，/include/asm-generic/pgtable.h\*/

return do\_numa\_page(mm, vma, address, entry, pte, pmd);

ptl = pte\_lockptr(mm, pmd);

spin\_lock(ptl);

if (unlikely(!pte\_same(\*pte, entry)))

goto unlock;

if (flags & FAULT\_FLAG\_WRITE) { /\*写操作触发的异常\*/

if (!pte\_write(entry)) /\*原PTE页表项没有写权限\*/

return  **do\_wp\_page(mm, vma, address,pte, pmd, ptl, entry)**; /\*处理写保护页\*/

entry = pte\_mkdirty(entry); /\*原PTE页表项有写权限，设置脏标记\*/

}

/\*读操作触发的异常，页表项无效\*/

**entry = pte\_mkyoung(entry);** /\*设置页刚被访问过，设置页表项A、V标记位，页回收\*/

if (ptep\_set\_access\_flags(vma, address, pte, entry, flags & FAULT\_FLAG\_WRITE)) {

**update\_mmu\_cache(vma, address, pte)**; /\*写入TLB页表项和刷新CPU缓存\*/

} else {

if (flags & FAULT\_FLAG\_WRITE)

flush\_tlb\_fix\_spurious\_fault(vma, address);

}

unlock:

pte\_unmap\_unlock(pte, ptl);

return 0;

}

handle\_pte\_fault()函数内判断PTE页表项映射的页帧是否在内存中，若不在内存中，分两种情况，一是页表项为空，表示需要创建新匿名映射页或文件映射页，二是页表项不为空，表示匿名映射页被交换到块设备交换区，需要分配物理页帧，从交换区中读回页内容，重新建立映射。

若PTE页表项映射的页帧在内存中，则表示异常是由访问权限不匹配触发的，对具有写保护的页执行写操作时，将执行写时复制操作。本节后面将具体介绍各种情形下的处理函数，对交换到交换区的页的处理在第11章再做介绍。

### 4.8.2新建匿名映射页

用户空间缺页异常处理函数中，如果异常地址对应页表项为空，且VMA不是文件映射，则为匿名映射，调用do\_anonymous\_page()函数创建新的匿名映射页，函数定义在/mm/memory.c文件内：

static int do\_anonymous\_page(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long address, pte\_t \*page\_table, pmd\_t \*pmd,unsigned int flags)

{

struct mem\_cgroup \*memcg;

struct page \*page;

spinlock\_t \*ptl;

pte\_t entry;

pte\_unmap(page\_table);

/\*不能是共享匿名映射，因为共享匿名映射实际是/dev/zero文件映射\*/

if (**vma->vm\_flags & VM\_SHARED**)

return VM\_FAULT\_SIGBUS;

if (check\_stack\_guard\_page(vma, address) < 0) /\*检查是否要在栈中增加保护页\*/

return VM\_FAULT\_SIGSEGV;

/\*读操作异常，则虚拟页映射到内核全局全零页\*/

if (!(flags & FAULT\_FLAG\_WRITE) && !mm\_forbids\_zeropage(mm)) {

entry = pte\_mkspecial(pfn\_pte(**my\_zero\_pfn(address)**,vma->vm\_page\_prot));

page\_table = pte\_offset\_map\_lock(mm, pmd, address, &ptl);

if (!pte\_none(\*page\_table))

goto unlock;

**goto setpte**;

}

if (unlikely(**anon\_vma\_prepare(vma)**)) /\*准备匿名反向映射数据结构，/mm/rmap.c\*/

goto oom;

page = **alloc\_zeroed\_user\_highpage\_movable(vma, address)**; /\*分配全零、可移动、高端内存页\*/

if (!page)

goto oom;

if (mem\_cgroup\_try\_charge(page, mm, GFP\_KERNEL, &memcg))

goto oom\_free\_page;

**\_\_SetPageUptodate(page)**; /\*设置page实例页数据有效PG\_uptodate标记位\*/

entry = **mk\_pte(page, vma->vm\_page\_prot)**; /\*生成PTE页表项\*/

if (vma->vm\_flags & VM\_WRITE)

entry = pte\_mkwrite(pte\_mkdirty(entry)); /\*赋予页表项可写权限\*/

page\_table = pte\_offset\_map\_lock(mm, pmd, address, &ptl); /\*PTE页表项指针\*/

if (!pte\_none(\*page\_table)) /\*PTE页表项应为空\*/

goto release;

inc\_mm\_counter\_fast(mm, MM\_ANONPAGES); /\*增加统计量\*/

**page\_add\_new\_anon\_rmap(page, vma, address)**;

/\*page关联到反向映射结构，设置PG\_swapbacked标记位，/mm/rmap.c\*/

mem\_cgroup\_commit\_charge(page, memcg, false);

**lru\_cache\_add\_active\_or\_unevictable(page, vma)**; /\*page添加到LRU链表，/mm/swap.c\*/

setpte:

**set\_pte\_at(mm, address, page\_table, entry)**; /\*设置PTE页表项\*/

**update\_mmu\_cache(vma, address, page\_table)**;

/\*写入TLB页表项、刷新CPU缓存，/arch/mips/include/asm/pgtable.h\*/

unlock:

pte\_unmap\_unlock(page\_table, ptl);

return 0;

...

oom:

return VM\_FAULT\_OOM;

}

do\_anonymous\_page()函数不能对共享的匿名映射页进行处理（由文件映射缺页处理），函数内为VMA

准备反向映射数据结构anon\_vma实例，然后调用伙伴系统分配页帧，优先从高端内存域可移页分配，并清零，随后依据分配的page实例和VMA访问权限成员生成PTE页表项，将page关联到反向映射结构和LRU链表，最后设置内存页表项，写入TLB页表项和刷新CPU缓存。

#### 1关联反向映射结构

page\_add\_new\_anon\_rmap()函数将分配的物理页帧page实例关联到匿名映射反向映射结构，函数定义如下（/mm/rmap.c）:

void page\_add\_anon\_rmap(struct page \*page,struct vm\_area\_struct \*vma, unsigned long address)

{

do\_page\_add\_anon\_rmap(page, vma, address, **0**); /\*/mm/rmap.c\*/

}

page\_add\_new\_anon\_rmap()函数内直接调用do\_page\_add\_anon\_rmap()函数，定义如下：

void do\_page\_add\_anon\_rmap(struct page \*page,struct vm\_area\_struct \*vma, unsigned long address, \

int exclusive)

{

int first = **atomic\_inc\_and\_test(&page->\_mapcount)**; /\*增加\_mapcount计数，并返回是否为0\*/

if (first) { /\*\_mapcount初始值为-1，加1后为0表示是第一次映射\*/

if (PageTransHuge(page))

\_\_inc\_zone\_page\_state(page,NR\_ANON\_TRANSPARENT\_HUGEPAGES);

\_\_mod\_zone\_page\_state(page\_zone(page), NR\_ANON\_PAGES,hpage\_nr\_pages(page));

/\*增加物理内存域匿名映射页统计数量\*/

}

if (unlikely(PageKsm(page)))

return;

VM\_BUG\_ON\_PAGE(!PageLocked(page), page);

if (first) /\*第一次映射\*/

**\_\_page\_set\_anon\_rmap(page, vma, address, exclusive)**; /\*第一次映射，/mm/rmap.c\*/

else

\_\_page\_check\_anon\_rmap(page, vma, address); /\*没有选择DEBUG\_VM配置选项，为空操作\*/

}

\_\_page\_set\_anon\_rmap()将page实例mapping成员指向anon\_vma实例（最低位置1），index成员设置成vma->vm\_pgoff加上address相对于VMA起始虚拟地址的页偏移量。

#### 2添加到LRU链表

lru\_cache\_add\_active\_or\_unevictable()函数用于将匿名映射页page实例添加到物理内存域LRU链表，函数定义如下（/mm/swap.c）：

void lru\_cache\_add\_active\_or\_unevictable(struct page \*page,struct vm\_area\_struct \*vma)

{

VM\_BUG\_ON\_PAGE(PageLRU(page), page);

if (likely((vma->vm\_flags & (VM\_LOCKED | VM\_SPECIAL)) != VM\_LOCKED)) {

/\*没有设置VM\_LOCKED或同时设置了VM\_LOCKED和VM\_SPECIAL\*/

**SetPageActive**(page); /\*设置page实例PG\_active标记位，将page插入活跃LRU链表\*/

**lru\_cache\_add**(page); /\*page添加到活跃或不活跃LRU链表，/mm/swap.c\*/

return;

}

if (!TestSetPageMlocked(page)) {

\_\_mod\_zone\_page\_state(page\_zone(page), NR\_MLOCK,hpage\_nr\_pages(page));

count\_vm\_event(UNEVICTABLE\_PGMLOCKED);

}

**add\_page\_to\_unevictable\_list(page)**;

/\*锁定VMA的page实例添加到不可回收LRU链表，/mm/swap.c\*/

}

由函数可知，如果VMA标记没有设置VM\_LOCKED或同时设置了VM\_LOCKED和VM\_SPECIAL标记位，则将page实例添加到物理内存域zone实例活跃或不活跃LRU链表，如果VMA标记成员设置了VM\_LOCKED标记位则将page添加到zone实例不可回收页面LRU链表。第11章将详细介绍page实例在LRU链表中的操作。

#### 3写入TLB和刷新缓存

update\_mmu\_cache()函数用于将生成PTE页表项写入TLB中并刷新CPU缓存，显然这是一个体系结构相关的函数，函数定义在/arch/mips/include/asm/pgtable.h头文件内：

static inline void update\_mmu\_cache(struct vm\_area\_struct \*vma,unsigned long address, pte\_t \*ptep)

{

pte\_t pte = \*ptep;

**\_\_update\_tlb(vma, address, pte)**; /\*/arch/mips/mm/tlb-r4k.c\*/

**\_\_update\_cache(vma, address, pte)**; /\*/arch/mips/mm/cache.c\*/

}

\_\_update\_tlb()函数定义在/arch/mips/mm/tlb-r4k.c文件内，函数内将ptep指向的页表项写入TLB中，如果TLB中已有address匹配的TLB表项，则覆盖它，没有则随机写入TLB中。

\_\_update\_cache()定义在/arch/mips/mm/cache.c文件内，调用处理器定义的刷新处理器数据缓存函数，刷新原page数据在CPU中的缓存。

内核在以上两个体系结构相关的文件及/arch/mips/mm/c-r4k.c文件内定义了处理器TLB和缓存的操作函数，源代码由于与处理器密切相关，请读者自行阅读。

### 4.8.3新建文件映射页

用户空间缺页异常处理函数中，如果异常地址对应页表项为空，且VMA实例vma->vm\_ops成员不为空，说明VMA为文件映射，则调用do\_fault()函数处理文件映射缺页异常。

do\_fault()函数定义在/mm/memory.c文件内，代码如下：

static int do\_fault(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long address, pte\_t \*page\_table, pmd\_t \*pmd,unsigned int flags, pte\_t orig\_pte)

{

pgoff\_t pgoff = (((address & PAGE\_MASK)- vma->vm\_start) >> PAGE\_SHIFT) + vma->vm\_pgoff;

/\*address对应映射文件内容的页偏移量\*/

pte\_unmap(page\_table); /\*返回page\_table\*/

if (!vma->vm\_ops->fault) /\*VMA操作函数中没有定义fault()函数，返回错误码\*/

return VM\_FAULT\_SIGBUS;

if (!(flags & FAULT\_FLAG\_WRITE)) /\*读操作触发的缺页异常，/mm/memory.c\*/

return **do\_read\_fault(mm, vma, address, pmd, pgoff, flags,orig\_pte)**;

if (!(vma->vm\_flags & VM\_SHARED)) /\*私有映射写操作触发的异常，/mm/memory.c\*/

return **do\_cow\_fault(mm, vma, address, pmd, pgoff, flags,orig\_pte)**;

return **do\_shared\_fault(mm, vma, address, pmd, pgoff, flags, orig\_pte)**;

/\*共享映射写操作触发的异常，/mm/memory.c\*/

}

文件映射缺页异常中又分三种情况：

（1）读操作引发的缺页异常，文件地址空间中缓存页未映射到进程地址空间或缓存页未创建，需要查找（或创建）缓存页，建立映射，可预读文件内容。

（2）对私有映射写操作触发的异常，需要查找（或创建）缓存页，分配新页帧，并复制缓存页内容至新页帧，然后将新页帧映射到进程虚拟内存。

（3）对共享映射写操作触发的异常，需要查找（或创建）缓存页，并建立映射。

#### 1通用函数

文件映射缺页异常分三种情况，在介绍各处理缺页异常的函数前，先介绍两个通用的处理函数：一是从文件地址空间中获取缓存页的\_\_do\_fault()函数，二是由缓存页生成并设置进程页表项的do\_set\_pte()函数。

文件映射的结构如下图所示，VMA映射的物理页帧由文件地址空间address\_space结构中的基数树管理，物理页帧缓存文件的内容，因此称为页缓存。

基数树中缓存页可以不映射到进程虚拟内存，页缓存可以看成是一个缓存池，进程根据需要建立与缓存页的映射关系（依缓存页填入PTE页表项），多个进程（或VMA）可同时映射到相同的缓存页。



文件映射页表项为空时，对以上三种情况的首要任务都是由异常地址获取缓存页实例，然后再考虑是将缓存页直接映射到进程VMA，还是复制新页帧映射到进程VMA等。

通用函数\_\_do\_fault()用于获取指定虚拟地址映射页的page实例，函数内将调用vm\_operations\_struct实例中的fault()函数，完成缓存页的获取。

##### ■获取缓存页

通用函数\_\_do\_fault()用于在文件地址空间基数树中查找（或创建）缓存页，并返回缓存页page实例指针。介绍函数实现之前先看一下vm\_fault结构体的定义（/include/linux/mm.h），vm\_fault结构体用于向内存域操作结构vm\_operations\_struct实例中fault()函数传递参数。

struct vm\_fault {

unsigned int flags; /\*缺页异常标记 FAULT\_FLAG\_xxx\*/

pgoff\_t pgoff; /\*缺页异常页在VMA中的偏移量（文件内容页偏移量）\*/

void \_\_user \*virtual\_address; /\*异常虚拟地址\*/

struct page \*cow\_page; /\*写时复制页page指针\*/

struct page \*page; /\*文件地址空间缓存页page指针\*/

/\* vm\_operations\_struct->map\_pages()函数专用成员\*/

pgoff\_t max\_pgoff; /\*映射页最大偏移量\*/

pte\_t \*pte; /\*页表项指针\*/

};

\_\_do\_fault()函数定义如下（/mm/memory.c）：

static int \_\_do\_fault(struct vm\_area\_struct \*vma, unsigned long address,pgoff\_t pgoff, unsigned int flags, \

struct page \*cow\_page, struct page \*\*page)

{

**struct vm\_fault vmf**;

int ret;

vmf.virtual\_address = (void \_\_user \*)(address & PAGE\_MASK); /\*异常地址页对齐\*/

vmf.pgoff = pgoff;

vmf.flags = flags;

vmf.page = NULL;

vmf.cow\_page = cow\_page;

**ret = vma->vm\_ops->fault(vma, &vmf)**; /\*通常为filemap\_fault()函数指针\*/

if (unlikely(ret & (VM\_FAULT\_ERROR | VM\_FAULT\_NOPAGE | VM\_FAULT\_RETRY)))

return ret;

if (!vmf.page)

goto out;

if (unlikely(PageHWPoison(vmf.page))) {

if (ret & VM\_FAULT\_LOCKED)

unlock\_page(vmf.page);

page\_cache\_release(vmf.page);

return VM\_FAULT\_HWPOISON;

}

if (unlikely(!(ret & VM\_FAULT\_LOCKED))) /\*如果返回值没有设置VM\_FAULT\_LOCKED标记\*/

**lock\_page(vmf.page)**; /\*锁定page\*/

else

VM\_BUG\_ON\_PAGE(!PageLocked(vmf.page), vmf.page);

out:

**\*page = vmf.page**; /\*缓存页page实例指针\*/

return ret;

}

\_\_do\_fault()函数内定义了vm\_fault结构体变量并对其进行初始化，调用VMA操作结构中的fault()函数完成缓存页的查找或创建，fault()函数通常赋值为filemap\_fault()函数（磁盘文件）。

filemap\_fault()函数内首先在页缓存中查找address对应缓存页，如果存在且有效则将缓存页page实例地址赋予vmf.page成员，如果缓存页存在但数据失效了则需要重新从文件中读入数据，如果缓存页不存在，则创建缓存页并从文件中读入数据，最后都将缓存页page实例赋予vmf.page成员。filemap\_fault()函数的定义到第11章再做详细介绍。

\_\_do\_fault()函数随后根据vm\_ops**->**fault()函数返回值还需要执行相应的操作，如果返回值中没有设置VM\_FAULT\_LOCKED标记位则调用lock\_page(vmf.page)函数锁定页，函数返回。注意，在调用\_\_do\_fault()函数后需要调用unlock\_page(vmf.page)函数解除对page的锁定。

##### ■设置页表项

do\_set\_pte()函数用于由物理页帧号生成并填充PTE页表项，刷新TLB页表项，建立映射关系，并将page实例关联到反向映射结构中。

do\_set\_pte()函数定义如下（/mm/memory.c）：

void do\_set\_pte(struct vm\_area\_struct \*vma, unsigned long address,struct page \*page, pte\_t \*pte, \

bool write, bool anon)

{

pte\_t entry;

flush\_icache\_page(vma, page);

entry = **mk\_pte(page, vma->vm\_page\_prot)**; /\*生成PTE页表项，使用VMA中的访问权限\*/

if (write)

entry = maybe\_mkwrite(pte\_mkdirty(entry), vma); /\*设置写属性\*/

if (anon) { /\*匿名映射页\*/

inc\_mm\_counter\_fast(vma->vm\_mm, MM\_ANONPAGES);

**page\_add\_new\_anon\_rmap(page, vma, address)**; /\*关联到匿名反向映射结构\*/

} else { /\*文件映射页\*/

inc\_mm\_counter\_fast(vma->vm\_mm, MM\_FILEPAGES);

**page\_add\_file\_rmap(page)**;

/\*文件反向映射，增加\_mapcount映射计数，更新统计量，/mm/rmap.c\*/

}

**set\_pte\_at(vma->vm\_mm, address, pte, entry)**; /\*设置内存PTE页表项\*/

**update\_mmu\_cache(vma, address, pte)**; /\*写入TLB页表项、刷新CPU缓存\*/

}

do\_set\_pte()函数主要工作是根据page实例和内存域访问权限生成PTE页表项，将page实例关联到反向映射结构（并增加引用计数，映射计数），最后将生成PTE页表项写入内存页表，写入TLB和刷新CPU缓存。

#### 2处理读异常页

在文件映射缺页处理函数do\_fault()中，如果异常地址PTE页表项为空，且是由读操作触发的缺页，则调用函数do\_read\_fault()函数处理缺页情形。

do\_read\_fault()函数定义如下（/mm/memory.c）：

static int do\_read\_fault(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long address, pmd\_t \*pmd,pgoff\_t pgoff, unsigned int flags, pte\_t orig\_pte)

{

struct page \*fault\_page;

spinlock\_t \*ptl;

pte\_t \*pte;

int ret = 0;

if (**vma->vm\_ops->map\_pages** && fault\_around\_bytes >> PAGE\_SHIFT > 1) {

pte = pte\_offset\_map\_lock(mm, pmd, address, &ptl);

**do\_fault\_around(vma, address, pte, pgoff, flags)**; /\*周边页尝试建立映射，/mm/memory.c\*/

if (!pte\_same(\*pte, orig\_pte)) /\*如果已经有进程修改了页表项\*/

goto unlock\_out;

pte\_unmap\_unlock(pte, ptl);

}

**ret = \_\_do\_fault(vma, address, pgoff, flags, NULL, &fault\_page)**;

if (unlikely(ret & (VM\_FAULT\_ERROR | VM\_FAULT\_NOPAGE | VM\_FAULT\_RETRY)))

return ret; /\*创建映射不成功返回错误码\*/

pte = pte\_offset\_map\_lock(mm, pmd, address, &ptl);

if (unlikely(!pte\_same(\*pte, orig\_pte))) { /\*如果已经有进程修改了页表项，则释放页\*/

pte\_unmap\_unlock(pte, ptl);

**unlock\_page(fault\_page)**; /\*解锁page\*/

page\_cache\_release(fault\_page); /\*等同put\_page(page)，减1引用计数，为0则释放页\*/

return ret;

}

**do\_set\_pte(vma, address, fault\_page, pte, false, false)**; /\*设置页表项，增加page计数值\*/

**unlock\_page(fault\_page)**; /\*解锁page\*/

unlock\_out:

pte\_unmap\_unlock(pte, ptl);

return ret;

}

do\_read\_fault()函数根据内存访问局部性的原理首先尝试调用do\_fault\_around()函数将缺页异常的周边页与文件地址空间中的缓存页建立映射，注意这里只是尝试与页缓存中已经存在的页建立映射，而不是创建缓存页，函数内部调用vm\_ops->map\_pages()函数尝试将周边页建立映射。

do\_read\_fault()函数随后调用\_\_do\_fault()函数在页缓存中创建（或查找）对应的页，返回后\*fault\_page指向映射页page实例，如果在调用\_\_do\_fault()函数过程中其它进程已经为虚拟页建立了映射（PTE页表项被修改了），则放弃本次操作。最后，调用do\_set\_pte()函数生成并设置PTE页表项，写入TLB和刷新CPU缓存，建立映射。

#### 3处理写时复制页

在文件映射缺页处理函数do\_fault()中，如果是由写私有VMA操作触发的缺页，则调用do\_cow\_fault()函数进行处理。对私有映射的写操作，需要启用写时复制（COW）机制，分配新物理页帧，将原缓存页中的数据复制到新页帧，再将虚拟页与新页帧建立映射关系（新页帧不在文件内容页缓存中），写操作不会传递到文件缓存页，不会同步到磁盘文件。如果对应文件缓存页不存在，则需要先创建。也就是说，写时复制页并不是映射到页缓存中的页，而是另外使用一个副本。

do\_cow\_fault()函数定义如下（/mm/memory.c）：

static int do\_cow\_fault(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long address, pmd\_t \*pmd,pgoff\_t pgoff, unsigned int flags, pte\_t orig\_pte)

{

struct page \*fault\_page, \*new\_page;

struct mem\_cgroup \*memcg;

spinlock\_t \*ptl;

pte\_t \*pte;

int ret;

if (unlikely(anon\_vma\_prepare(vma))) /\*写进复制VMA视为匿名VMA，创建反向映射结构\*/

return VM\_FAULT\_OOM;

new\_page = **alloc\_page\_vma(GFP\_HIGHUSER\_MOVABLE, vma, address)**; /\*分配物理页帧\*/

if (!new\_page)

return VM\_FAULT\_OOM;

if (mem\_cgroup\_try\_charge(new\_page, mm, GFP\_KERNEL, &memcg)) {

**page\_cache\_release(new\_page)**;

return VM\_FAULT\_OOM;

}

ret = **\_\_do\_fault(vma, address, pgoff, flags, new\_page, &fault\_page)**; /\*创建或查找缓存页\*/

if (unlikely(ret & (VM\_FAULT\_ERROR | VM\_FAULT\_NOPAGE | VM\_FAULT\_RETRY)))

goto uncharge\_out;

if (fault\_page)

**copy\_user\_highpage(new\_page, fault\_page, address, vma)**; /\*复制缓存页内容至新页帧\*/

**\_\_SetPageUptodate(new\_page)**; /\*设置page有效标记位\*/

pte = pte\_offset\_map\_lock(mm, pmd, address, &ptl);

if (unlikely(!pte\_same(\*pte, orig\_pte))) { /\*如果映射已被其它进程创建，放弃本次操作\*/

pte\_unmap\_unlock(pte, ptl);

if (fault\_page) {

unlock\_page(fault\_page);

page\_cache\_release(fault\_page);

} else {

i\_mmap\_unlock\_read(vma->vm\_file->f\_mapping);

}

goto uncharge\_out;

}

**do\_set\_pte(vma, address, new\_page, pte, true, true)**; /\*生成并设置PTE页表项等\*/

mem\_cgroup\_commit\_charge(new\_page, memcg, false);

**lru\_cache\_add\_active\_or\_unevictable(new\_page, vma)**; /\*新页帧page添加到LRU链表\*/

pte\_unmap\_unlock(pte, ptl);

if (fault\_page) {

**unlock\_page(fault\_page)**; /\*解锁缓存页\*/

**page\_cache\_release(fault\_page)**; /\*引用计数值减1，为0则释放页\*/

} else {

i\_mmap\_unlock\_read(vma->vm\_file->f\_mapping);

}

return ret;

uncharge\_out:

mem\_cgroup\_cancel\_charge(new\_page, memcg);

page\_cache\_release(new\_page);

return ret;

}

do\_cow\_fault()函数不难理解，首先从伙伴系统分配高端内存域、可移动页帧，调用\_\_do\_fault()函数查找或创建页缓存中对应页，复制缓存页数据至新分配页帧，虚拟页与新页帧建立映射关系。新页帧被添加到物理内存域页LRU链表，原缓存页引用计数值减1，若减1后为0则释放缓存页。

#### 4处理写共享页

在文件映射缺页处理函数do\_fault()中，若缺页是由写共享映射页操作触发的，则调用do\_shared\_fault()函数进行处理。共享页对所有进程可见，一个进程对共享页的写操作对其它进程可见，并且会回写到块设备文件中。

do\_shared\_fault()函数定义如下（/mm/memory.c）：

static int do\_shared\_fault(struct mm\_struct \*mm, struct vm\_area\_struct \*vma, \

unsigned long address, pmd\_t \*pmd,pgoff\_t pgoff, unsigned int flags, pte\_t orig\_pte)

{

struct page \*fault\_page;

struct address\_space \*mapping;

spinlock\_t \*ptl;

pte\_t \*pte;

int dirtied = 0;

int ret, tmp;

ret = **\_\_do\_fault(vma, address, pgoff, flags, NULL, &fault\_page)**; /\*查找或创建缓存页\*/

if (unlikely(ret & (VM\_FAULT\_ERROR | VM\_FAULT\_NOPAGE | VM\_FAULT\_RETRY)))

return ret;

if (**vma->vm\_ops->page\_mkwrite**) { /\*使能写操作函数\*/

unlock\_page(fault\_page);

tmp = **do\_page\_mkwrite(vma, fault\_page, address)**;

/\*调用vm\_ops->page\_mkwrite()函数，通知文件地址空间页将变成可写，/mm/memory.c\*/

if (unlikely(!tmp ||(tmp & (VM\_FAULT\_ERROR | VM\_FAULT\_NOPAGE)))) {

page\_cache\_release(fault\_page);

return tmp;

}

}

pte = pte\_offset\_map\_lock(mm, pmd, address, &ptl);

if (unlikely(!pte\_same(\*pte, orig\_pte))) { /\*如果已经有进程建立了映射，放弃本次操作\*/

pte\_unmap\_unlock(pte, ptl);

unlock\_page(fault\_page);

page\_cache\_release(fault\_page);

return ret;

}

**do\_set\_pte(vma, address, fault\_page, pte, true, false)**; /\*生成并设置页表项等\*/

pte\_unmap\_unlock(pte, ptl);

if (**set\_page\_dirty(fault\_page)**) /\*标记页脏，/mm/page-writeback.c\*/

dirtied = 1;

mapping = fault\_page->mapping;

unlock\_page(fault\_page);

if ((dirtied || vma->vm\_ops->page\_mkwrite) && mapping) {

**balance\_dirty\_pages\_ratelimited(mapping);**  /\*脏页平衡，触发数据回写，见第11章\*/

}

if (!vma->vm\_ops->page\_mkwrite) /\*没有定义page\_mkwrite()函数\*/

file\_update\_time(vma->vm\_file); /\*更新文件时间元信息，/fs/inode.c\*/

return ret;

}

共享文件映射页写操作缺页异常的处理与前面介绍的缺页处理相似，如果VMA操作结构实例中定义了page\_mkwrite()函数指针，则需要调用此函数通知文件地址空间使能对页的写操作（设置脏标记等），触发回写操作。

### 4.8.4处理写保护页

在handle\_pte\_fault()函数中，如果映射页PTE页表项存在，是由对只读页（没有写权限，PTE页表项W标记位为0）进行写操作触发的缺页（写保护页），则调用do\_wp\_page()函数处理此种情况。

do\_wp\_page()函数定义如下（/mm/memory.c）：

static int do\_wp\_page(struct mm\_struct \*mm, struct vm\_area\_struct \*vma,unsigned long address, \

pte\_t \*page\_table, pmd\_t \*pmd,spinlock\_t \*ptl, pte\_t orig\_pte) \_\_releases(ptl)

{

struct page \*old\_page;

old\_page = **vm\_normal\_page(vma, address, orig\_pte)**; /\*是否是普通映射页，是则返回page指针\*/

if (!old\_page) { /\*不是普通映射页，如PFN映射页等\*/

if ((vma->vm\_flags & (VM\_WRITE|VM\_SHARED)) == (VM\_WRITE|VM\_SHARED))

return **wp\_pfn\_shared(mm, vma, address, page\_table, ptl, orig\_pte, pmd)**;

/\*可写共享页，调用vma->vm\_ops->pfn\_mkwrite(vma, &vmf)，

\*设置页表项的脏、可写标记，写入TLB表项，刷新缓存，

\*触发数据回写等，/mm/memory.c。\*/

pte\_unmap\_unlock(page\_table, ptl);

return **wp\_page\_copy(mm, vma, address, page\_table, pmd, orig\_pte, old\_page)**;

/\*分配新页帧，复制原映射页数据，映射到新页帧，

\*写入TLB表项，刷新address对应CPU缓存等，/mm/memory.c。

\*/

}

/\*以下是处理普通页\*/

if (PageAnon(old\_page) && !PageKsm(old\_page)) { /\*处理匿名映射页\*/

if (!**trylock\_page(old\_page)**) {

page\_cache\_get(old\_page); /\*增加引用计数\*/

pte\_unmap\_unlock(page\_table, ptl);

lock\_page(old\_page);

page\_table = pte\_offset\_map\_lock(mm, pmd, address,&ptl);

if (!pte\_same(\*page\_table, orig\_pte)) {

unlock\_page(old\_page);

pte\_unmap\_unlock(page\_table, ptl);

page\_cache\_release(old\_page);

return 0;

}

page\_cache\_release(old\_page);

}

if (**reuse\_swap\_page(old\_page)**) {

/\*如果页引用计数小于等于1，修改页表项可继续使用映射页，/mm/swapfile.c\*/

page\_move\_anon\_rmap(old\_page, vma, address);

unlock\_page(old\_page);

return **wp\_page\_reuse(mm, vma, address, page\_table, ptl, orig\_pte, old\_page, 0, 0)**;

/\*修改PTE页表项标记（设置脏、可写标记），写入TLB表项，刷新缓存，

\*数据回写等，/mm/memory.c\*/

}

unlock\_page(old\_page);

}

/\*处理共享文件映射页\*/

else if (unlikely((vma->vm\_flags & (VM\_WRITE|VM\_SHARED)) ==

(VM\_WRITE|VM\_SHARED))) {

return **wp\_page\_shared(mm, vma, address, page\_table, pmd,ptl, orig\_pte, old\_page)**;

/\*设置页表项可写、脏标记，写入TLB表项，刷新CPU缓存，触发回写等\*/

}

/\*处理需要复制新页，建立新映射的情况\*/

page\_cache\_get(old\_page);

pte\_unmap\_unlock(page\_table, ptl);

return **wp\_page\_copy(mm, vma, address, page\_table, pmd,orig\_pte, old\_page);**

/\*分配新页帧，复制数据，设置PTE页表项，写入TLB表项，刷新CPU缓存等\*/

}

创建进程时，子进程复制父进程的页表项，对父进程写时复制VMA的页表项会清除可写权限，因此当子进程对其进行写操作时会触发写保护异常。对写保护页的处理与前面处理文件映射写时复制页类似，不过要根据映射页类型分别处理。

（1）映射的不是普通内存页（如设备内存），如果是共享映射则调用wp\_pfn\_shared()函数进行处理，主要工作是调用vm\_ops->pfn\_mkwrite(vma, &vmf)函数，设置页表项的脏、可写权限，触发回写等。

（2）匿名映射页，如果只有当前进程映射到该页（页引用计数小于等于1），只需将其关联到VMA反向映射结构中，修改页表项脏、可写权限等。如果页引用计数大于1，则需要分配新页帧，复制页数据，建立新映射。

（3）共享文件映射页，则调用wp\_page\_shared()函数进行处理，同（1）。对私有文件映射页，调用wp\_page\_copy()函数，分配新页帧，复制旧页内容，建立新映射。

在以上操作中在写入PTE页表项之后都会将此PTE页表项写入TLB（以避免再次触发缺页异常），并刷新异常地址对应的CPU缓存。

至此，用户空间缺页异常处理函数就介绍完了，CPU产生缺页异常的两个主要原因：一是映射数据是不在内存中，可能映射未建立，也可能页数据在交换区，二是页表项不为空，映射数据在内存中，但是访问权限不匹配。

对于第一种情况处理方法如下：

（1）如果匿名映射页，直接从伙伴系统分配页帧，修改内存 PTE页表项，建立映射，同时将page实例添加到反向映射结构和LRU链表中。

（2）如果是文件映射页，在文件地址空间页缓存中查找（或创建）页或复制缓存页内容至新页帧，修改内存 PTE页表项，建立映射。

（3）对于交换至交换区的匿名映射页，分配物理页帧，从交换区中读回数据，修改内存 PTE页表项，建立映射。

对于第二种情况处理方法如下：

（1）如果是共享映射页，则调用执行内存域操作结构中的pfn\_mkwrite()函数，设置PTE页表项的可写、脏标记位，增加映射计数（普通映射页），触发回写等。

（2）如果是私有映射页，则分配新页帧，复制旧页内容至新页帧，生成并设置PTE页表项，建立新映射。

以上所有操作，在设置或修改PTE页表项后，都要将PTE页表项写入TLB，并刷异常地址对应的CPU缓存。

## 4.9释放地址空间

在execve()系统调用运行新进程时，以及进程退出的exit()系统调用中将调用mm\_release()函数解除进程与mm\_struct实例的关联，随后调用mmput()函数释放不再被使用的mm\_struct实例。

### 4.9.1 mm\_release()

mm\_release()函数用于解除指定进程与mm\_struct实例之间的关联，即进程不再使用此地址空间实例了，函数定义如下（/kernel/fork.c）：

void mm\_release(struct task\_struct \*tsk, struct mm\_struct \*mm)

{

#ifdef CONFIG\_FUTEX

...

#endif

uprobe\_free\_utask(tsk); /\*没有选择UPROBES配置选项为空操作，/include/linux/uprobes.h\*/

**deactivate\_mm(tsk, mm)**;

/\*MIPS32体系结构定义为空操作，/arch/mips/include/asm/mmu\_context.h\*/

if (tsk->clear\_child\_tid) {

if (!(tsk->flags & PF\_SIGNALED) &&atomic\_read(&mm->mm\_users) > 1) {

**put\_user(0, tsk->clear\_child\_tid)**; /\*tsk->clear\_child\_tid写入0\*/

sys\_futex(tsk->clear\_child\_tid, FUTEX\_WAKE,1, NULL, NULL, 0);

}

tsk->clear\_child\_tid = NULL;

}

if (**tsk->vfork\_done**)

**complete\_vfork\_done(tsk)**;

/\*唤醒在tsk->vfork\_done完成量上睡眠等待的进程，/kernel/fork.c\*/

}

### 4.9.2 mmput()

mmput()函数用于减少mm\_struct实例的用户数，如果用户数为0，将释放mm\_struct实例。mmput()函数定义如下（/kernel/fork.c）：

void mmput(struct mm\_struct \*mm)

{

might\_sleep();

if (atomic\_dec\_and\_test(&mm->mm\_users)) { /\*用户数减1后判断是否为0，为0则执行if语句\*/

uprobe\_clear\_state(mm);

exit\_aio(mm);

ksm\_exit(mm);

khugepaged\_exit(mm); /\* must run before exit\_mmap \*/

**exit\_mmap(mm)**; /\*解除所有映射，/mm/mmap.c\*/

**set\_mm\_exe\_file**(mm, NULL); /\*修改可执行目标文件引用计数，/kernel/fork.c\*/

if (!list\_empty(&mm->mmlist)) {

spin\_lock(&mmlist\_lock);

list\_del(&mm->mmlist);

spin\_unlock(&mmlist\_lock);

}

if (mm->binfmt)

module\_put(mm->binfmt->module);

**mmdrop(mm)**; /\*释放mm\_struct实例，/include/linux/sched.h\*/

}

}

mmput()函数调用exit\_mmap(mm)函数解除地址空间中所有VMA映射，并释放VMA，函数源代码请读者自行阅读。

mmdrop(mm)函数用于释放mm\_struct实例，函数定义如下（/include/linux/sched.h）：

static inline void mmdrop(struct mm\_struct \* mm)

{

if (unlikely(atomic\_dec\_and\_test(&mm->mm\_count))) /\*引用计数减1后是否为0\*/

**\_\_mmdrop(mm)**; /\*/kernel/fork.c\*/

}

mmdrop(mm)函数对mm\_struct实例的引用计数减1，结果为0，则调用\_\_mmdrop()函数释放mm\_struct实例。\_\_mmdrop()函数定义如下（/kernel/fork.c）：

void \_\_mmdrop(struct mm\_struct \*mm)

{

BUG\_ON(mm == &init\_mm);

mm\_free\_pgd(mm); /\*释放PGD页表，/kernel/fork.c\*/

destroy\_context(mm); /\*空操作\*/

mmu\_notifier\_mm\_destroy(mm); /\*注册通知，没有选择MMU\_NOTIFIER配置选项为空操作\*/

check\_mm(mm); /\*输出信息，/kernel/fork.c\*/

free\_mm(mm); /\*将mm\_struct实例释放回slab缓存，/kernel/fork.c\*/

}

## 4.10内核与用户空间复制数据

内核经常需要从用户空间复制数据到内核空间，如系统调用中用户需要向内核传递参数和数据，内核空间也需要将数据返回给用户空间。内核可以访问处理器所有虚拟地址空间，包括用户空间，但是用户不能访问内核空间。内核在与用户交换数据时，必须检查地址的有效性，包括地址是否合法，虚拟地址是否映射到物理内存。

内核提供了内核与用户空间交换数据的标准函数（/arch/mips/include/asm/uaccess.h），例如：

●**copy\_from\_user**(to, from, n)：从from指向的用户空间复制n字节长度的数据至内核空间to处。

●**copy\_to\_user**(to, from, n)：从from指向的内核空间复制n字节长度的数据至用户空间to处。

●**get\_user**(x,ptr)：从ptr表示的用户空间地址处复制简单数据结构数据至内核x变量。

●**put\_user**(x,ptr)：将内核空间变量值x复制到ptr表示的用户空间地址处。

表示用户空间地址的变量前需加关键字\_\_user标记，以便编译过程中检查工具检查指针的有效性。

## 4.11小结

CPU执行程序产生的地址是程序地址，或称为虚拟地址，虚拟地址通过MMU（TLB）转换或线性映射转换成物理地址，用于访问物理内存。

CPU的虚拟地址空间分为内核地址空间和用户地址空间，内核地址空间位于上半部分，用户地址空间位于下半部分。内核在内核地址空间运行，用户进程在用户地址空间运行。

MIPS32位系统中内核地址空间分为直接映射区、IO映射区、VMALLOC区、持久映射区和固定映射区。其中直接映射区、IO映射区线性映射到物理内存低512MB空间，不需要通过TLB转换，其它区域需要通过页表转换。

内核需要在直接映射区分配/释放内存时，直接使用伙伴系统的分配/释放函数，分配的物理内存地址线性转换成内核空间虚拟地址。

在内核地址空间VMALLOC区创建映射分为全局分配器和每CPU分配器，全局分配器接口函数简列如下：

**●**void \***vmalloc(unsigned long size)**：在VMALLOC区创建指定大小的映射区域，由内核自动分配页帧，修改内核页表项建立映射，返回起始虚拟地址。

**●**void \***vzalloc(unsigned long size)：**与vmalloc()相同，且对物理内存清零。

●void \***vmalloc\_32(unsigned long size)：**与vmalloc()相似，且保证分配页帧为32位地址可寻址。

●void \***vmalloc\_32\_user(unsigned long size)：**与vmalloc\_32()相似，且保证分配的页帧清零。

●void \***vmalloc\_exec(unsigned long size)：**与vmalloc()相似，保证映射区域具有可执行属性。

●void \***vmap(struct page \*\*pages, unsigned int count,unsigned long flags, pgprot\_t prot)**：显式映射函数，显式地将page指针数组指定的页帧映射到VMALLOC区。注意，这里由函数参数指定物理页帧，不需要内核分配。

**●void vfree(const void \*addr)**：释放vmalloc()， vmalloc\_32()或\_\_vmalloc()创建的映射区。

●**void** **vunmap(const void \*addr)：**释放vmap()创建的映射区。

VMALLOC区每CPU分配器接口函数如下（用于分配短时使用的小块内存）：

●void \***vm\_map\_ram**(struct page \*\*pages, unsigned int count,int node, pgprot\_t prot)：将page指针数组关联的页通过每CPU分配器映射到VMALLOC区。

●void **vm\_unmap\_ram**(const void \*mem, unsigned int count)：解除通过每CPU分配器创建的映射。如果虚拟内存块已全部建立了映射，且在此函数中解除了最后的映射，则虚拟内存块将释放。

●void **vm\_unmap\_aliases**(void)：释放所有没有建立任何映射的虚拟内存块。

在内核地址空间固定映射区建立/解除临时映射的接口函数如下：

●void \***kmap\_atomic(struct page \*page)**：将page页帧映射到内核地址空间固定映射区，返回虚拟地址。

●void **kunmap\_atomic(addr)**：解除固定映射。

在内核地址空间持久映射区建立/解除临时映射的接口函数如下：

●void \***kmap(struct page \*page)**：将page页帧映射到内核地址空间持久映射区，返回虚拟地址。

●void **kunmap(struct page \*page)：**解除持久映射。

在内核地址空间IO映射区建立/解除临时映射的接口函数如下：

● void \_\_iomem \***ioremap(offset, size)**：将起始物理地址为offset长度为size的IO内存映射到内核地址空间IO映射区或VMALLOC区，返回虚拟地址。

●void **iounmap(addr)**：解除IO映射。

内核只有一个，内核地址空间是公用的，而用户进程有许多。每个用户进程都有自己的地址空间，并且彼此间应相互隔离。每个用户进程拥有自己的页表，通过页表将地址空间映射到不同的物理内存，CPU在切换进程时，同时切换使用的页表，以达到不同进程间地址空间的隔离。

用户地址空间分为代码/数据区、堆、内存映射区、栈等区域。

代码/数据区为文件映射区，在加载可执行目标文件时，由内核创建并映射到可执行目标文件。

堆用于为进程动态分配（小块）内存，brk()系统调用用于扩展或收缩堆，用户程序通常通过malloc()库函数间接地使用堆。

栈用于保存函数调用过程中的参数和变量值，在加载可执行目标文件内，内核将为进程创建初始栈，在程序运行过程中栈会动态生长。

内存映射区用于分配大块内存或创建文件映射，动态库即加载到内存映射区。

用户地址空间管理相关系统调用简列如下：

●**mlock**(start, len)：锁定进程地址空间从start开始，长度为len（字节数）的区域。

●**mlockall**(flags)：锁定进程所有映射区域。

●**munlock**(start,len)：解锁进程地址空间指定start、len表示的区域。

●**mlockall**(flags)：解锁进程所有映射区域。

●**mmap()/mmap2()**：在内存映射区创建映射（分配内存）。

●**mremap()：**重映射，收缩/扩展映射（隐含移动动映射区）。

●**mprotect()**：改变指定内存区域的映射属性。

**●munmap()：**解除映射。

内核地址空间创建映射时，立即分配页帧、修改页表项，完成映射的建立，映射到内核地址空间的页帧不会被回收。

在用户地址空间创建映射时，如果映射区域不是锁定的，或创建映射系统调用标记参数中没有指定MAP\_POPULATE/MAP\_LOCKED标记位，则只是创建表示映射区域的数据结构实例，而没有分配物理页帧和修改页表项。在CPU访问到没有建立映射的地址时，在缺页异常（TLB异常）处理程序中逐页建立映射。分配给用户进程使用的页帧可能被回收，除非对内存区域执行锁定操作。