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

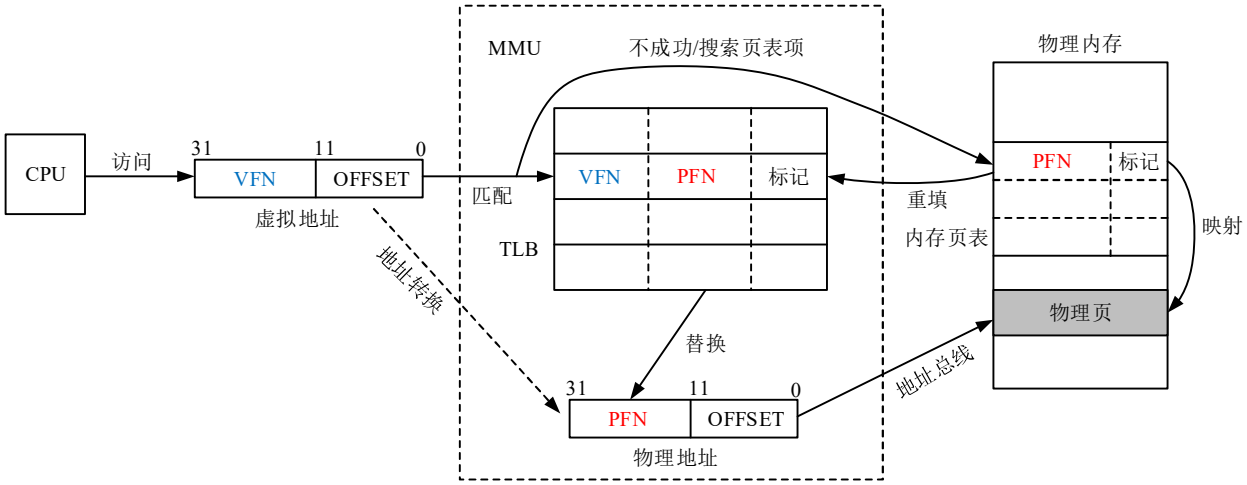
前面第 3 章介绍过，内核按页对物理内存进行管理。虚拟地址到物理地址的转换也是按页进行的，一个虚拟页对应一个物理页，也可以多个虚拟页对应同一个物理页。记录虚拟页与物理页映射关系的数据结构称为页表项。页表是页表项的集合（数组）包含进程整个虚拟地址空间与物理内存的映射关系，保存在内存中。MMU 中的 TLB 是内存页表的缓存，CPU 进行地址转换时，在 TLB 中查找匹配的项，若未找到则到内存页表中查找对应页表项，填充至 TLB 中后再进行转换。

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

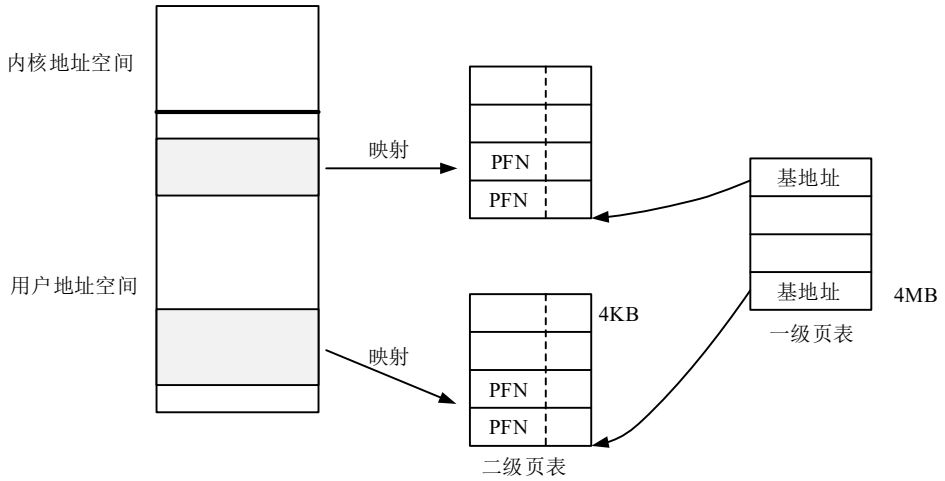
构与 TLB 内页表项的结构可能有所不同，在 TLB 重填过程中完成两者之间的转换。MIPS 处理器由软件（异常处理程序）完成 TLB 页表项的重填，也有的处理器架构由硬件实现页表项重填。

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。

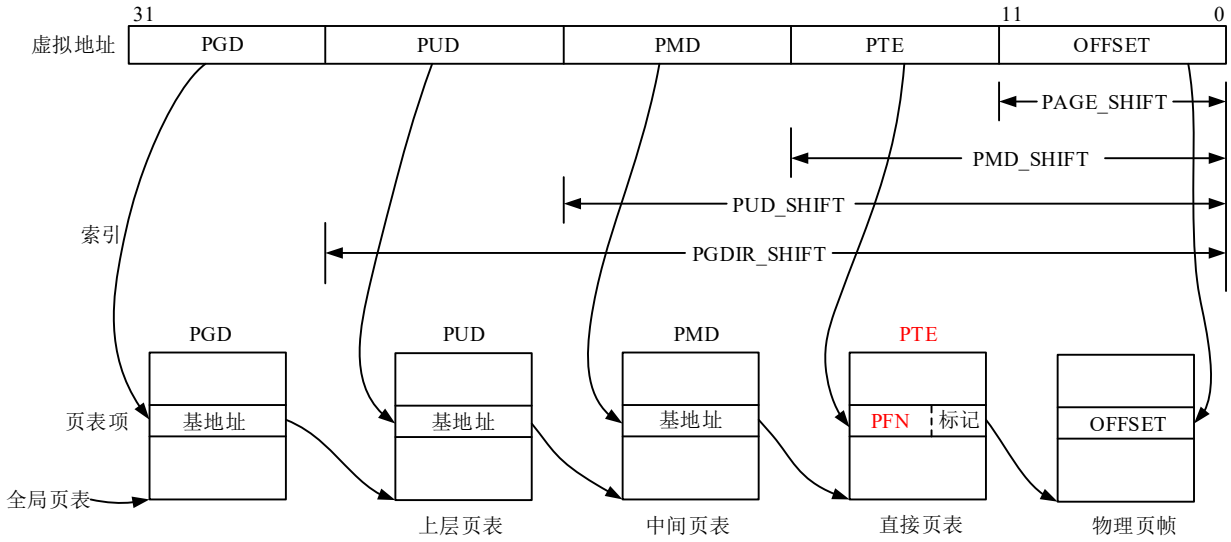
内存页表是一个页表项数组，虚拟页编号是数组项索引值，每个数组项代表一个页的映射关系，保存了映射的页帧号和访问标记。MIPS32 体系结构中，内核地址空间和用户进程地址空间分别为 2GB，假设页大小为 4KB，则内核和进程地址空间分别需要 500K 条页表项，每条页表项占用 4 个字节，则内核和每个进程页表需要占用 2MB 的空间，并且这 2MB 的空间必须是连续的！这几乎是不可能实现的。

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*/
```

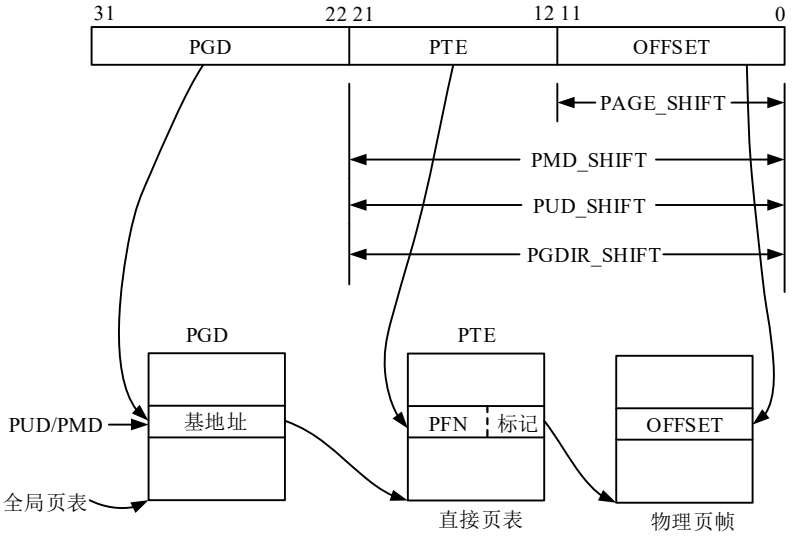
__builtin_ffs(x)表示计算整数 x 第一个为 1 的比特位位置（编号从 1 开始），如：x=1，返回 1，x=2，返回 2。因此 PTE_T_LOG2宏的值为__builtin_ffs(4)-1 结果为 2。

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))

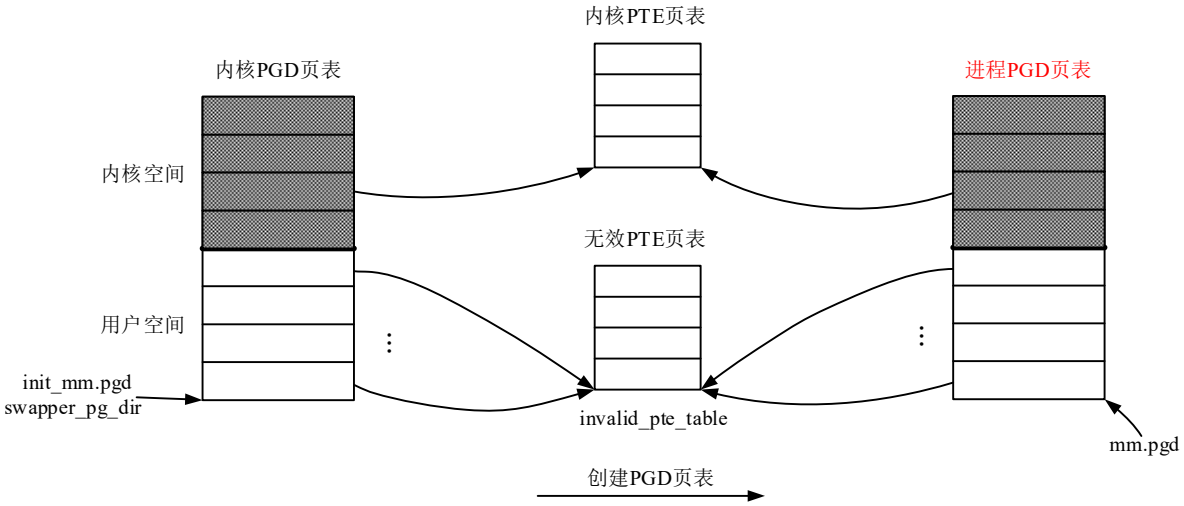
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD-1)) /*页表项索引值*/

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：

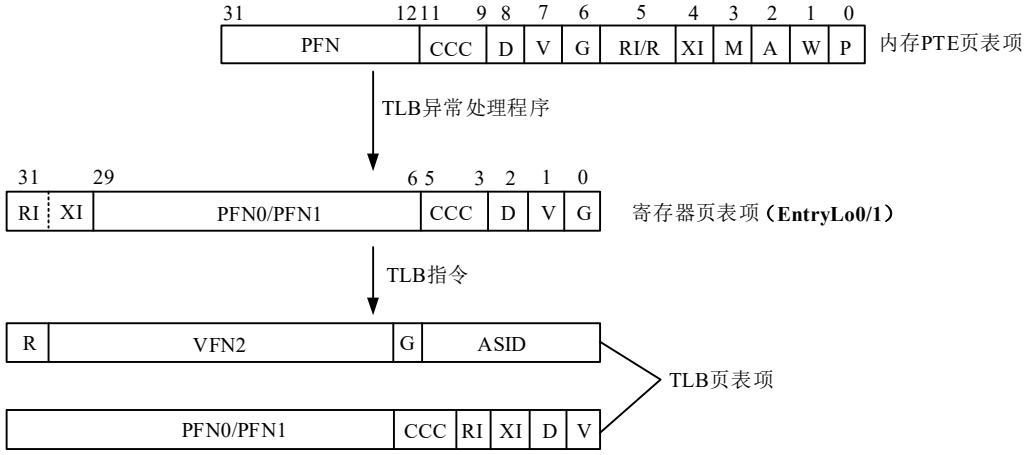
●static inline void **pte_free_kernel**(struct mm_struct *mm, pte_t *pte)：函数内直接调用伙伴系统释放页函数。

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：

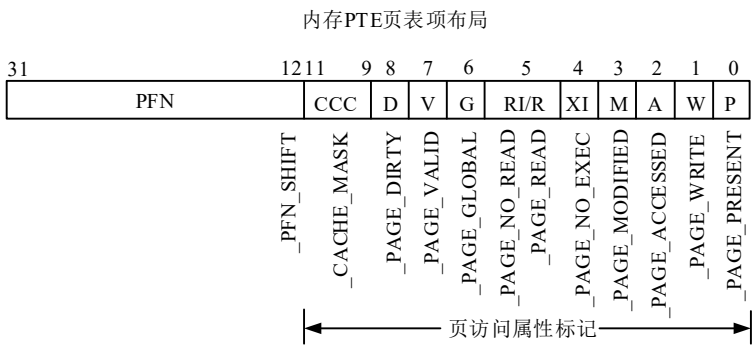
```
typedef struct { unsigned long pte; }   pte_t;          /*PTE 页表项数据结构*/
#define   pte_val(x)      ((x).pte)                     /*页表项数值*/
#define   __pte(x)    ((pte_t) { (x) } )                /*将整数 x 转换成 pte_t 类型数据*/

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）。

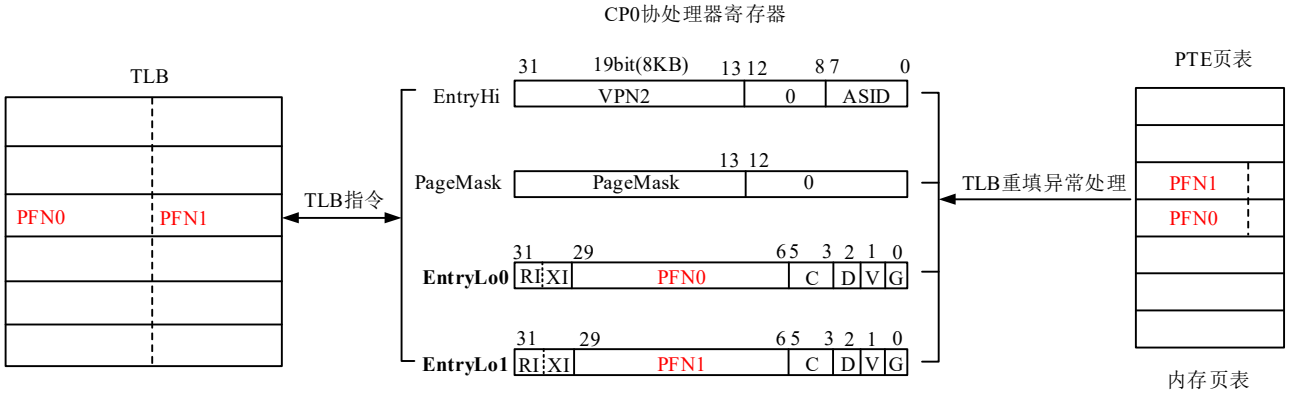
4.1.4 TLB 控制

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（不考虑 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

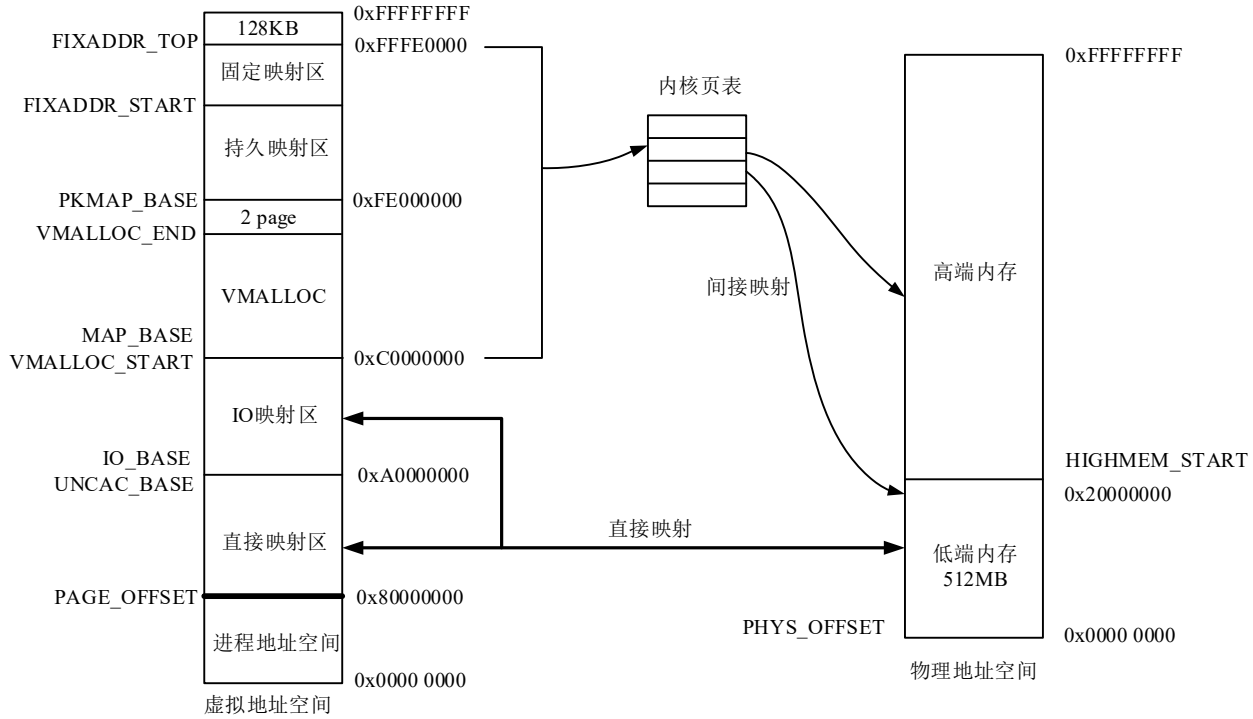
MIPS32 体系结构中，处理器在内核态和用户态下虚拟内存的映射关系如下图所示：

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。

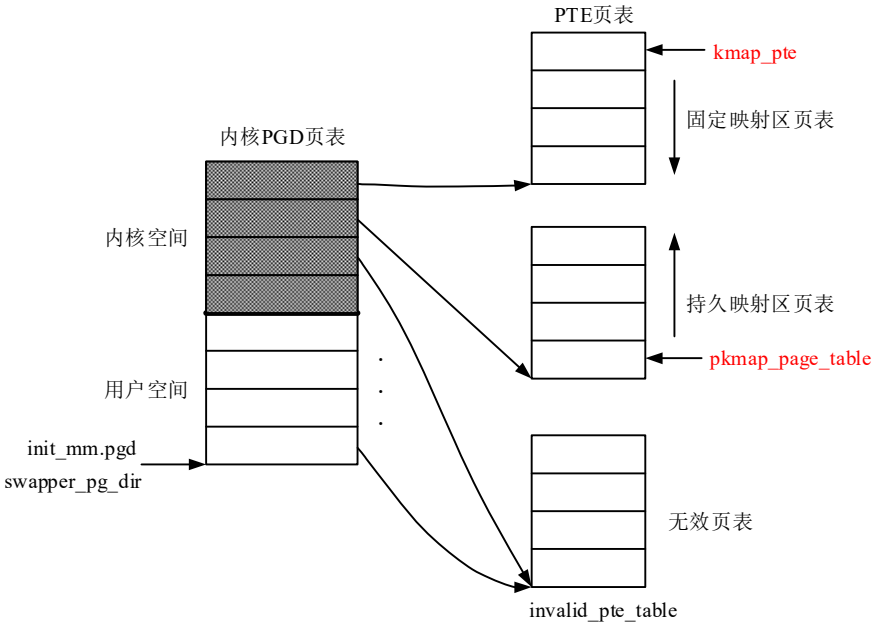
pagetable_init()函数将内核 PGD 页表项都指向无效的 PTE 页表 invalid_pte_table[]，然后为内核地址空

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）*/
```

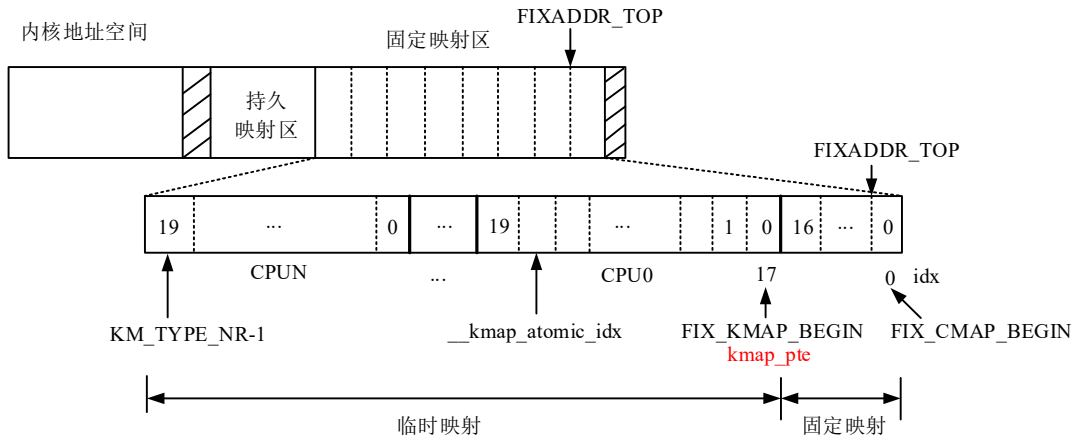
__end_of_fixed_addresses 表示固定映射区的页数量，如果内核配置没有选择支持高端内存则映射页数量为 17，表示只有固定映射区，没有临时映射区。如果内核配置支持高端内存，还将为每个 CPU 核划分 KM_TYPE_NR 数量的虚拟页，用于临时映射。

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。

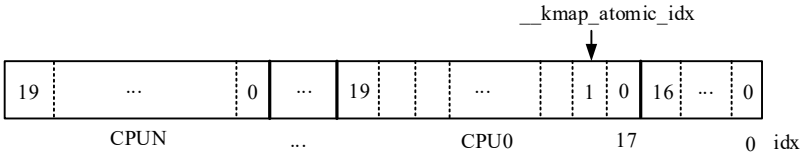
1 建立临时映射

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。

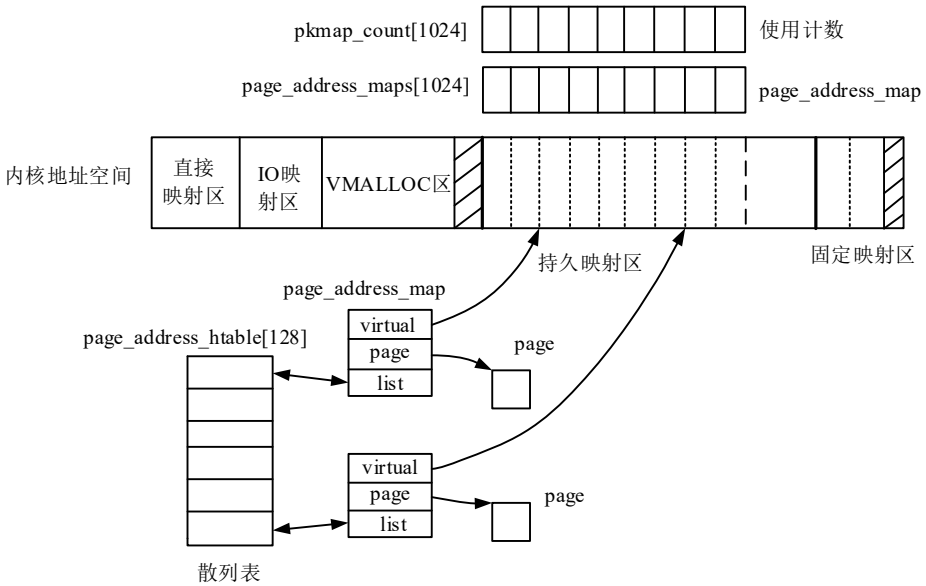
在建立映射前，创建者需先向内核申请页帧（伙伴系统），调用 kmap(page)函数建立映射，使用完后调用 kunmap(page)函数解除映射，并释放页帧。建立映射时会修改映射页对应的内存页表项，并刷新 TLB 中表项。在解除映射时，对应内存页表项并不会马上清空，而是由内核在适当的时候清空。建立映射时，调用进程可能会进入睡眠。

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。

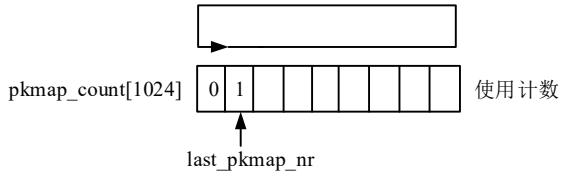
内核 pkmap_count[LAST_PKMAP]数组用来记录持久映射区虚拟页的使用计数，正常新创建映射的页其使用计数值为 2。解除映射时，解除映射函数只是对页使用计数值减 1。在建立持久映射时，首先扫描使用计数值为 0 的页，如果存在使用计数值为 0 的页，则在此虚拟页上建立映射。如果没有使用计数值为 0 的页，则扫描使用计数值为 1 页，对其解除映射，然后再试图建立映射。

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：

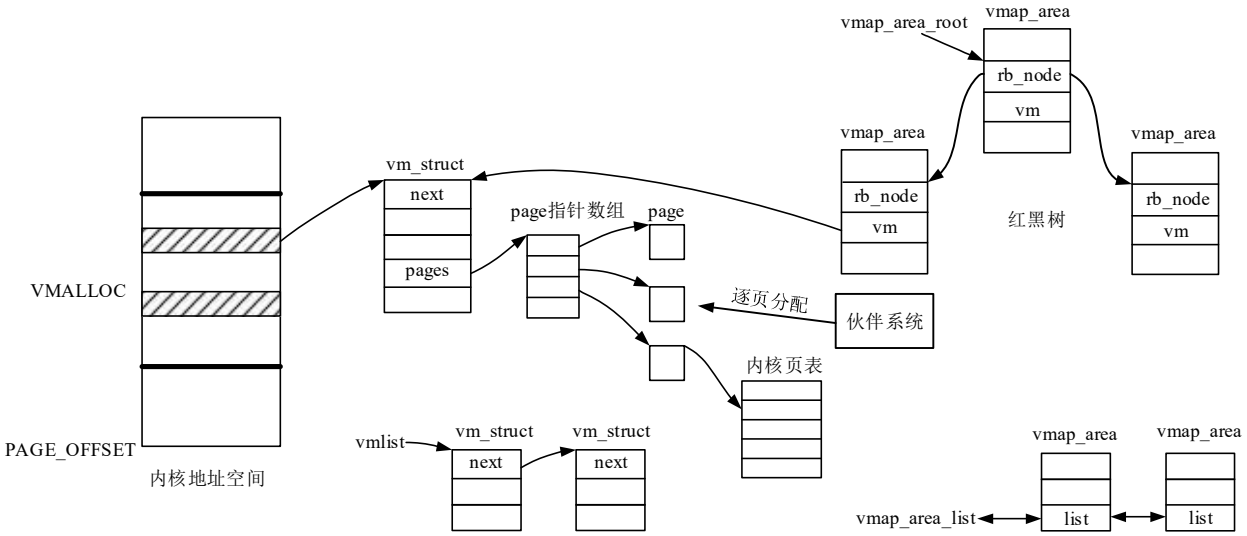
```
#define   MAP_BASE    _AC(0xc0000000, UL)
#define   VMALLOC_START      MAP_BASE            /*起始地址*/
```

```
#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。

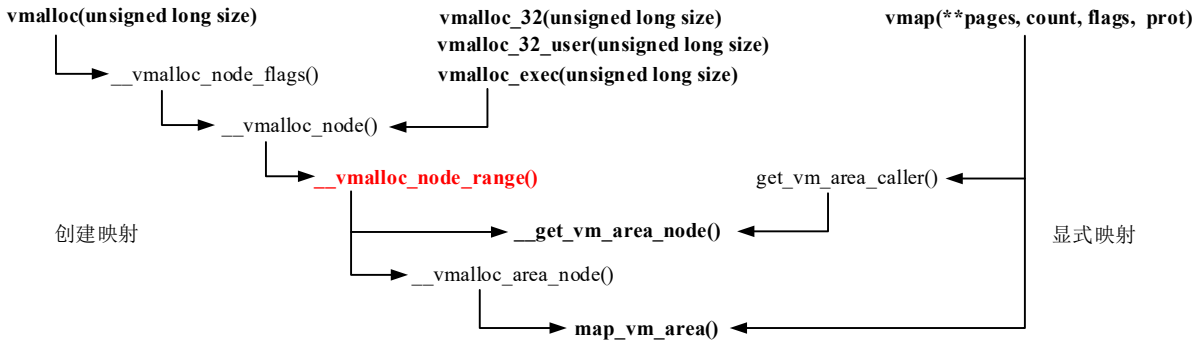
●void **vm_unmap_ram**(const void *mem, unsigned int count)：解除映射函数。

●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：



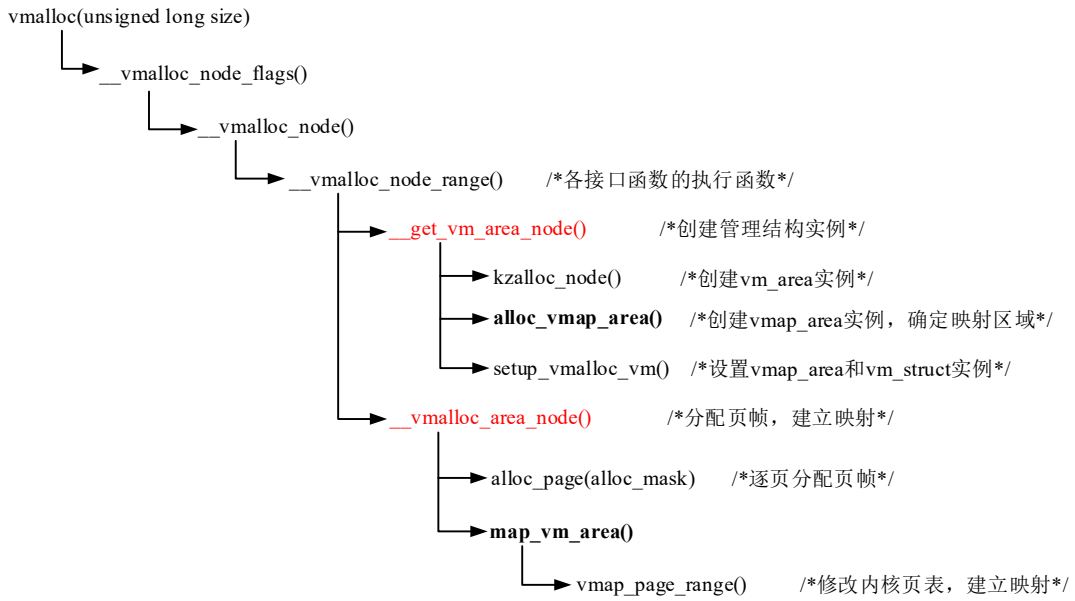
创建映射函数只需指定创建映射区域的大小（以及映射页帧来自哪个内存结点），由内核确定映射区域的位置，分配物理页帧，修改内核页表等，从而建立映射，函数返回映射区起始虚拟地址。

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。

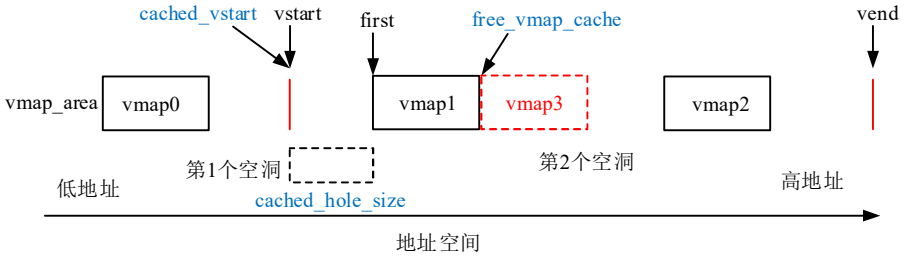
下面介绍一下创建 vmap_area 实例的 alloc_vmap_area()函数实现，其它函数代码请读者自行阅读。

●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。

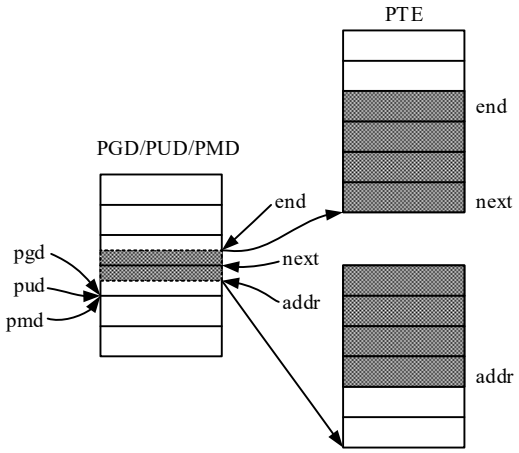
vmalloc()函数在调用__vmalloc_node_flags()函数时分配掩码设置了__GFP_HIGHMEM 标记位，因此分配函数优先从高端内存中分配页帧。__vmalloc_area_node()函数最后调用 map_vm_area()函数修改内核页表完成映射的创建，返回映射区起始虚拟地址。

●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;
}
```

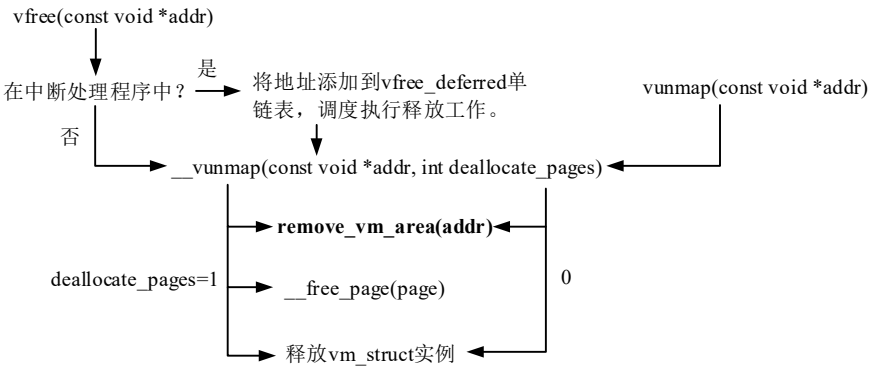
get_vm_area_caller()函数内直接调用__get_vm_area_node()函数，如下所示：

```
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：

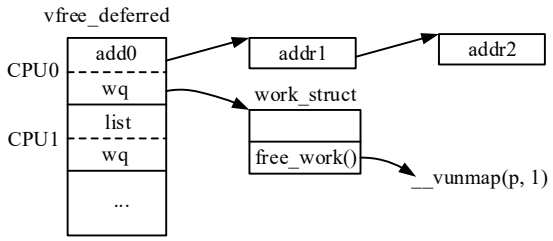
```
struct vfree_deferred {
    struct llist_head list;         /*单链表节点*/
    struct work_struct wq;      /*工作*/
};
```

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。

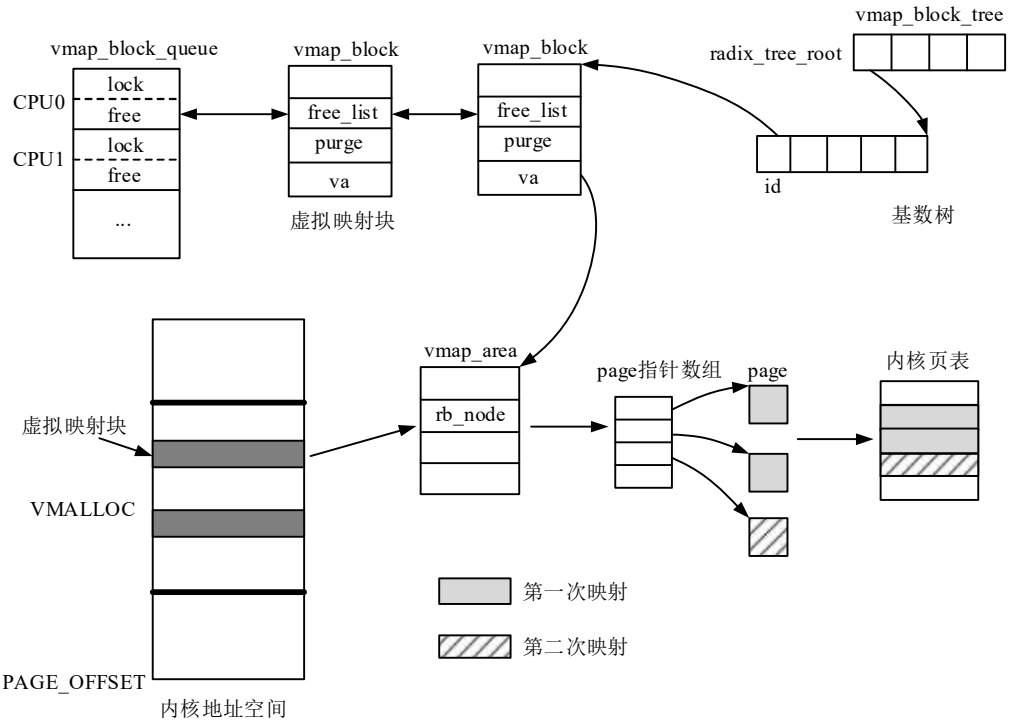
每 CPU 分配器框架如下图所示，分配器也是在 VMALLOC 区创建映射区域，但是每次创建映射区的大小是固定的，为 VMAP_BLOCK_SIZE 字节，所含页数为 VMAP_BBMAP_BITS（在/mm/vmalloc.c 文件内定义）。每个映射区域称为虚拟映射块，由 vmap_block 结构体表示。vmap_block 实例关联到映射区域对应的 vmap_area 实例，但是没有创建关联的 vm_struct 实例。

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 2^{order}，但是实际建立映射（修改页表项）的页数可能小于 2^{order}。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。

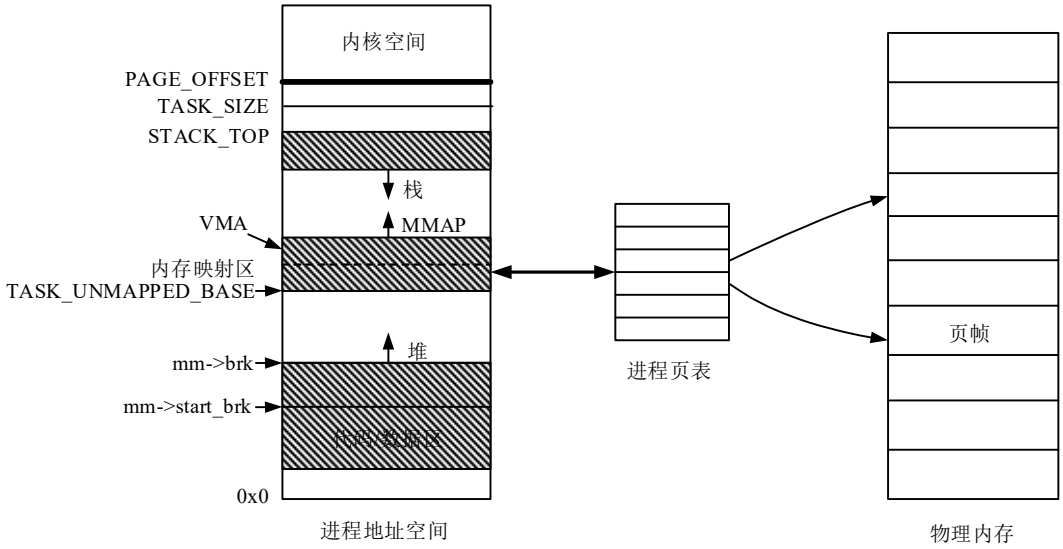
本节介绍进程虚拟地址空间的布局，各内存区的功能和映射关系，以及表示进程地址空间、虚拟内存区域的 mm_struct 和 vm_area_struct 数据结构的定义和组织结构。

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

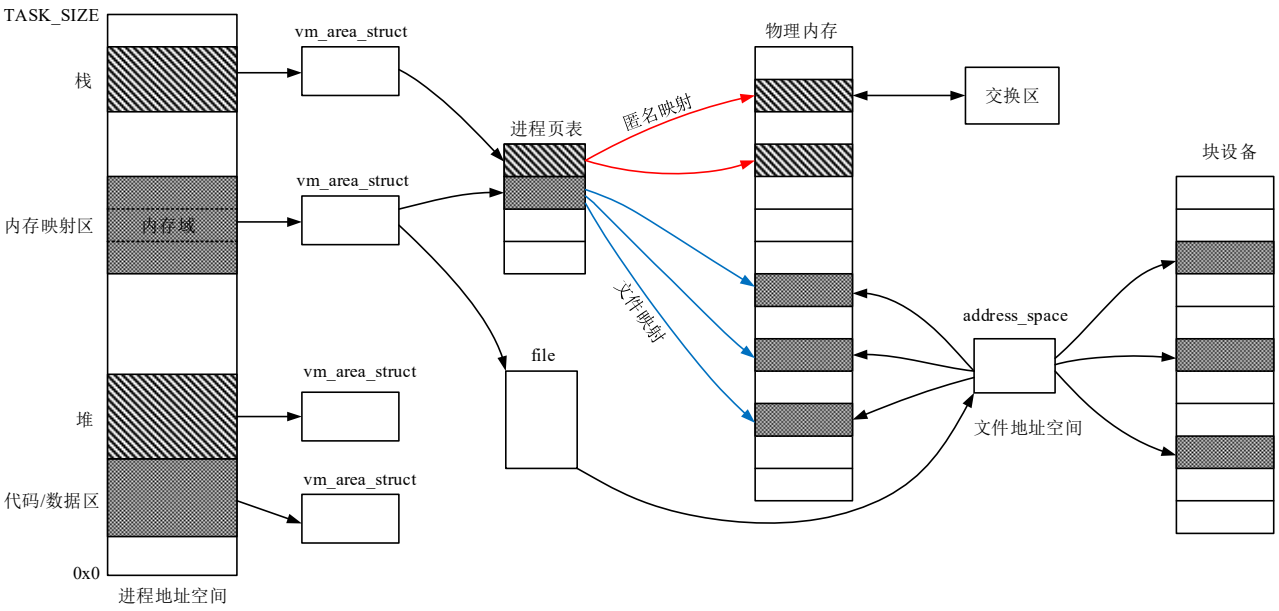
所谓映射就是虚拟内存落实到物理内存的转换关系。进程虚拟内存全部通过页表映射到物理内存，进程使用的虚拟内存区域被划分成小块，每块称为虚拟内存域，由 vm_area_struct 结构体实例表示。

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：

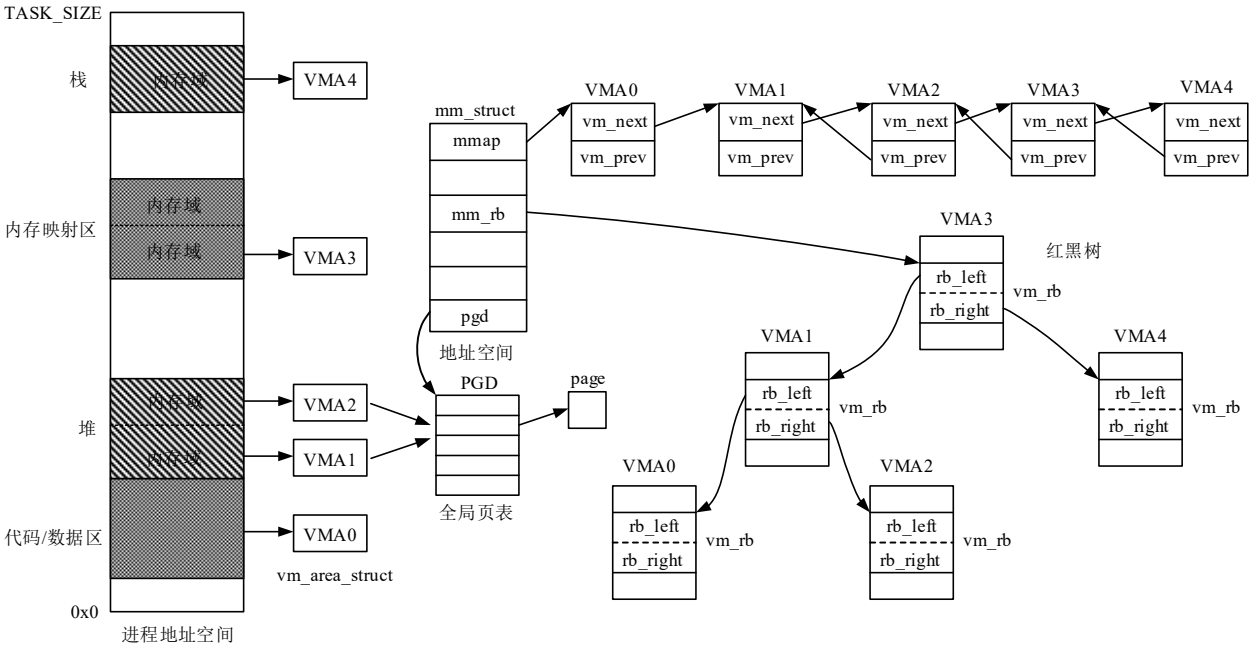
```
struct mm_struct {
    struct vm_area_struct    *mmap;    /*指向 vm_area_struct 实例，构成链表*/
    struct rb_root    mm_rb;            /*（扩展）红黑树根节点*/
    ...
}
```

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。

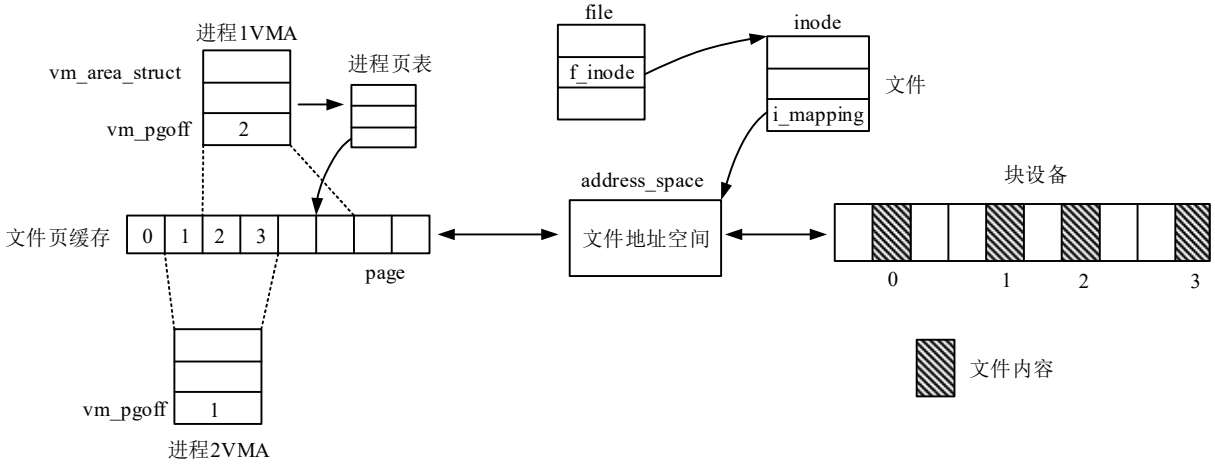
●static ITSTRUCT ***ITPREFIX ## _subtree_search**(ITSTRUCT *node, ITTYPE start, ITTYPE last)：在 node 为根节点的子树中查找区间值与[start,last]有重叠的子树中最左边节点（对象）。

●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

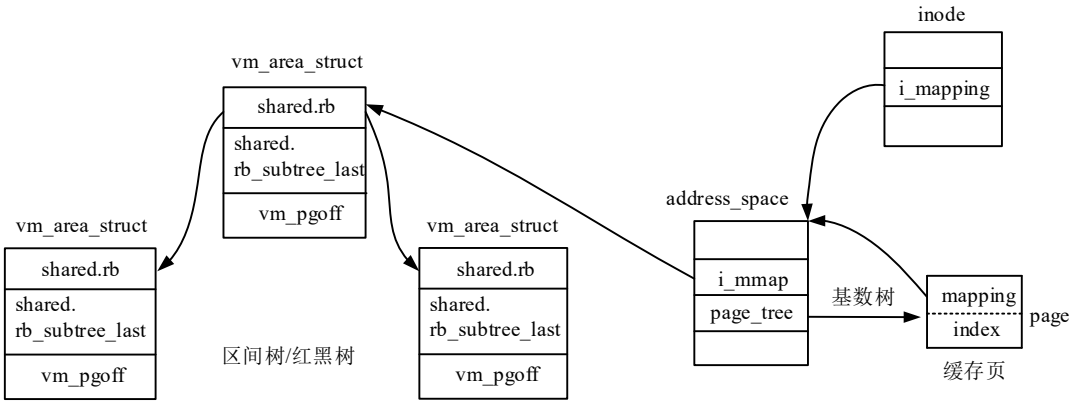
文件映射的反向映射结构如下图所示，在内核中文件由 inode 实例唯一表示（file 是进程文件的表示），inode 结构体包含文件地址空间 address_space 结构体实例，address_space 实际上是文件内容在内存中的缓存，address_space 通过基数树管理缓存文件内容的页帧（页缓存）。不同进程的 VMA 或者同一进程的不同 VMA 可以同时映射到同一个文件，如图中所示，进程 1 和进程 2 都有 VMA 映射到文件。vm_area_struct 结构体中包含 VMA 映射到文件内容的起始偏移量（vm_pgoff，页偏移量）和长度信息（映射区域长度）。不同 VMA 映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*/
};
```

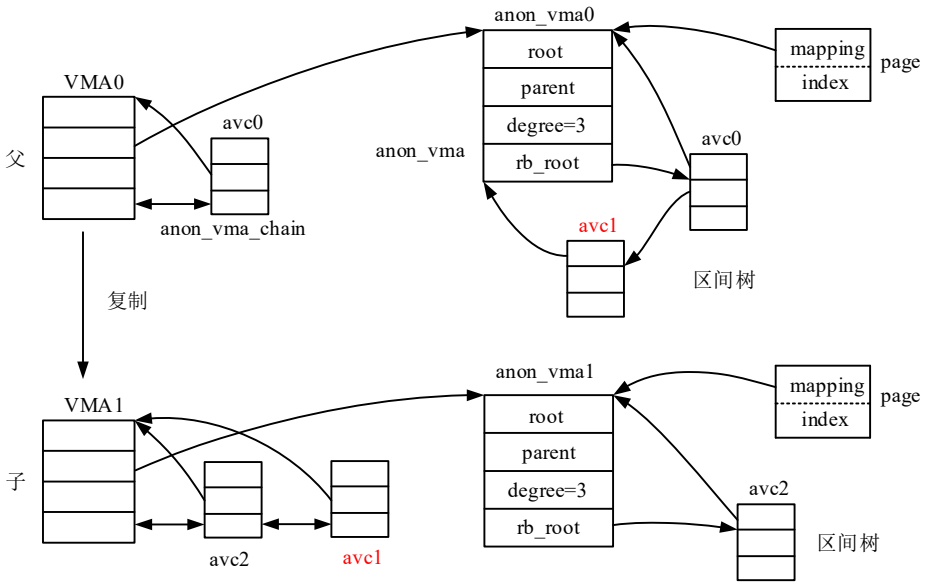
anon_vma_chain 结构体用于关联 vm_area_struct 和 anon_vma 实例，定义如下（/include/linux/rmap.h）：

```
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）：

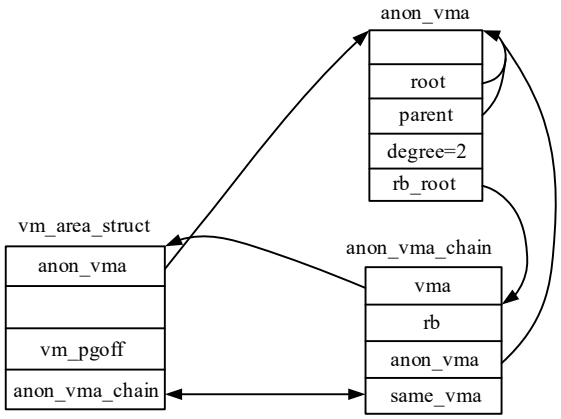
```
static inline struct anon_vma *anon_vma_alloc(void)
{
    struct anon_vma *anon_vma;

    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*/
}
```

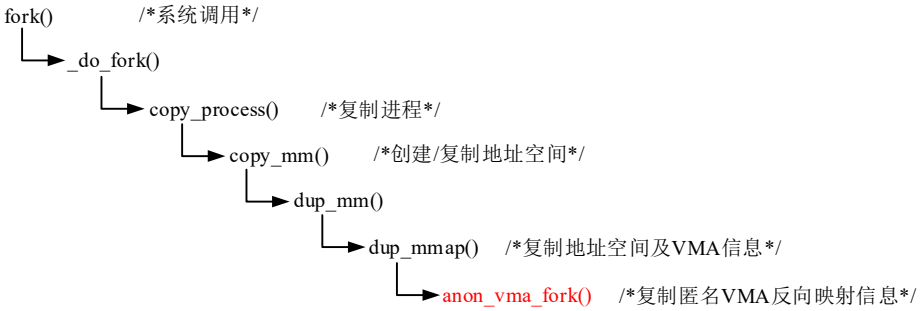
anon_vma_interval_tree_insert()函数调用__anon_vma_interval_tree_insert()函数将 anon_vma_chain 添加到 anon_vma->rb_root 管理的区间树。

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：

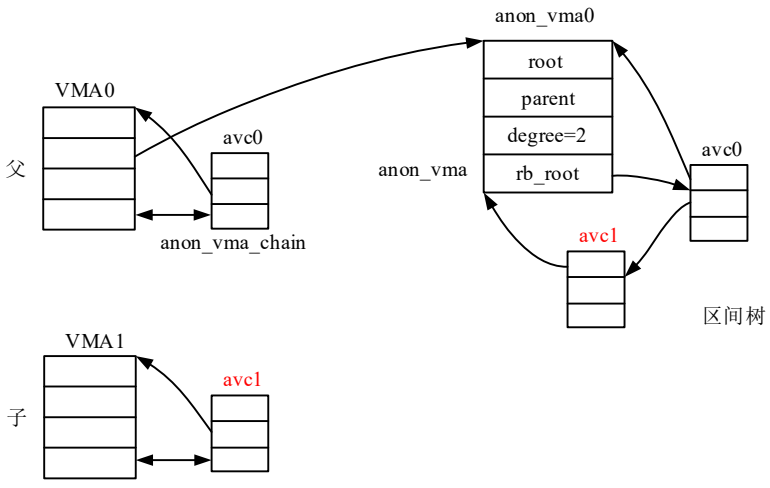
（1）调用 anon_vma_clone()函数扫描父 VMA 的 anon_vma_chain 双链表，对每个 anon_vma_chain 实例（旧实例）为子 VMA 创建一个新 anon_vma_chain 实例，新实例关联到旧实例关联的 anon_vma 实例，以及子进程 VMA 实例。新 anon_vma_chain 实例添加到子 VMA 实例中的 anon_vma_chain 双链表头部，以及旧 anon_vma_chain 实例关联 anon_vma 实例的红黑树。

anon_vma_clone(dst, src)函数定义如下（/mm/rmap.c）：

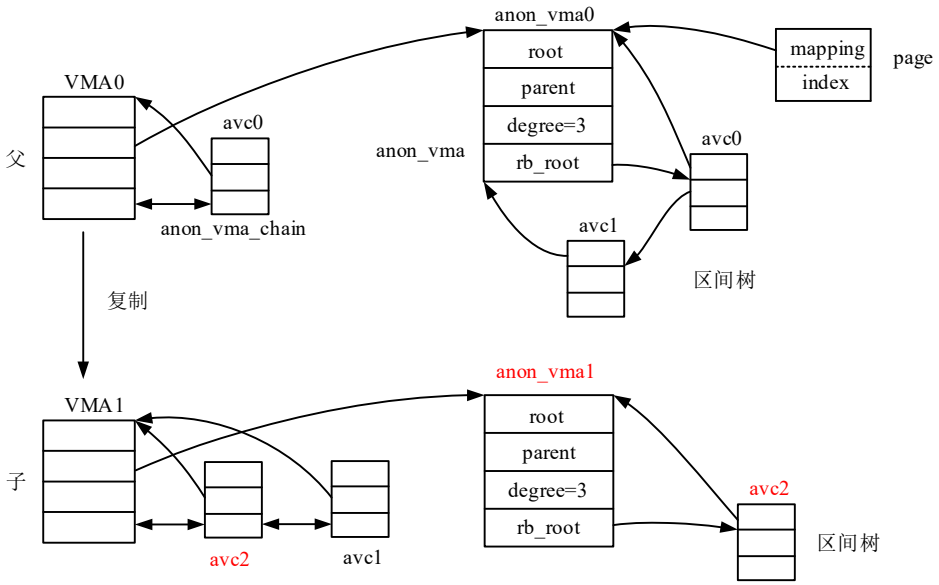
```
int anon_vma_clone(struct vm_area_struct *dst, struct vm_area_struct *src)
/*dst：子进程 VMA 指针，src：父进程 VMA 指针*/
{
    struct anon_vma_chain *avc, *pavc;
    struct anon_vma *root = NULL;

    /*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*/
    ...
}
```

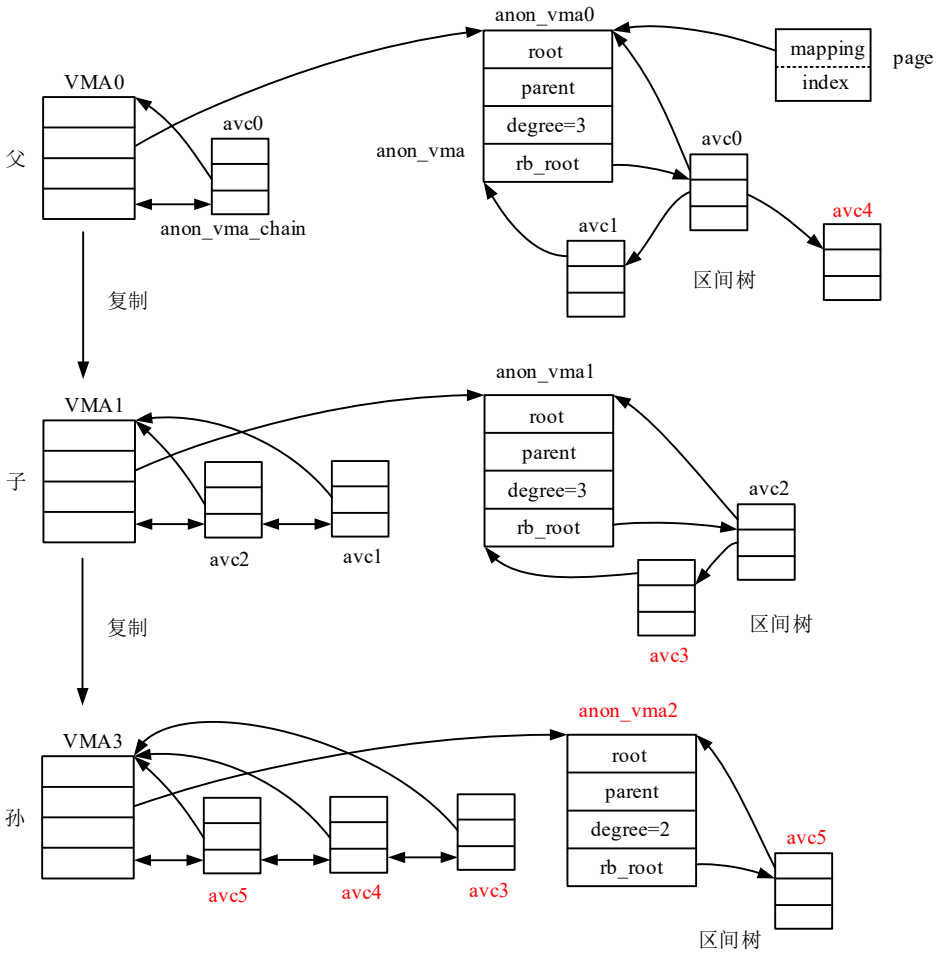
下面通过一个例子来说明 anon_vma_clone(dst, src)函数的功能。如下图所示，假设子进程 VMA1 复制于父进程 VMA0。由于 VMA0 中 anon_vma_chain 双链表只有一个成员 avc0，因此在 anon_vma_clone()函数中将只创建一个 anon_vma_chain 实例 acv1。acv1 关联到子进程 VMA1 以及 avc0 关联的 anon_vma 实例 anon_vma0。acv1 添加到 VMA1 中的双链表和 anon_vma0 中的区间树。



（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

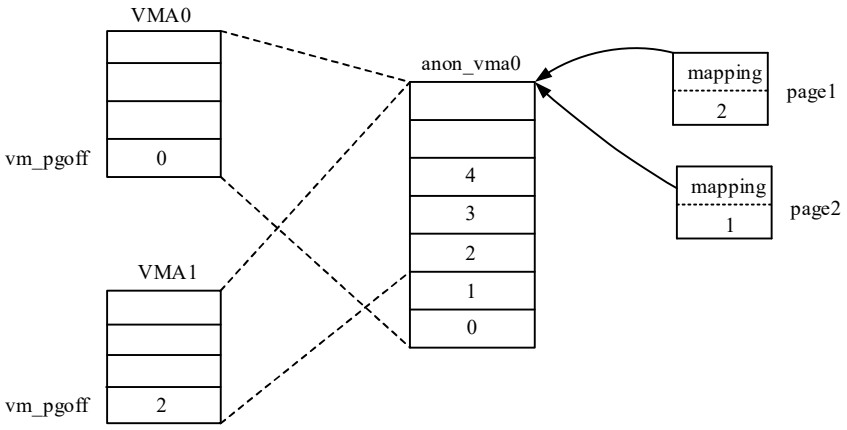
上图中的 anon_vma1 和 anon_vma2 实例的 root 成员都指向 anon_vma0 实例。

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？

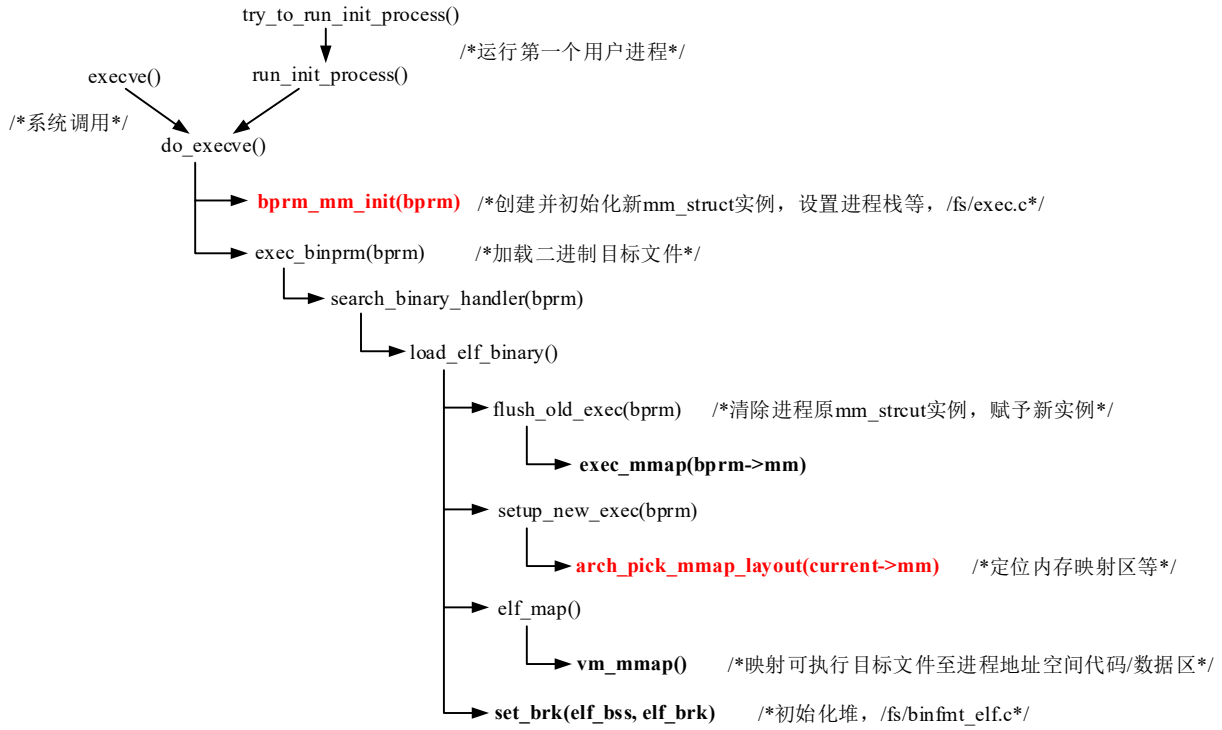
内核自身地址空间 mm_struct 实例 init_mm 定义在/mm/init-mm.c 文件内，内核自身可看成系统内的第一个进程（内核线程），其它进程（线程）通过复制而来。由于内核（内核线程）不使用用户地址空间，因此，在 init_mm 实例中并没有表示用户进程地址空间的信息。在运行第一个用户进程，加载可执行目标文件时，将创建进程地址空间。

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）：

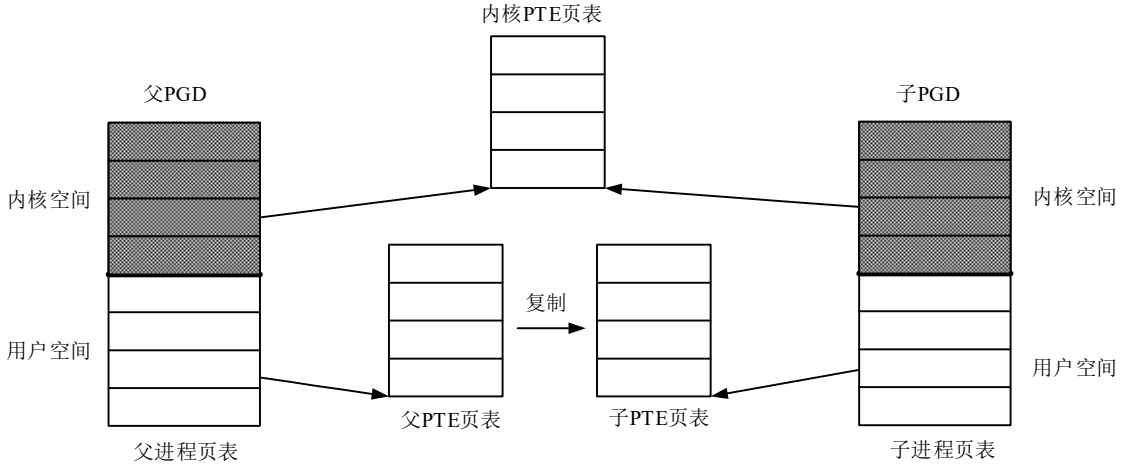
```
static inline unsigned long copy_one_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm,   \
                        pte_t *dst_pte, pte_t *src_pte, struct vm_area_struct *vma,unsigned long addr, int *rss)
/*dst_pte：目标 PTE 页表项指针，src_pte：源 PTE 页表项指针*/
{
    unsigned long vm_flags = vma->vm_flags;     /*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。

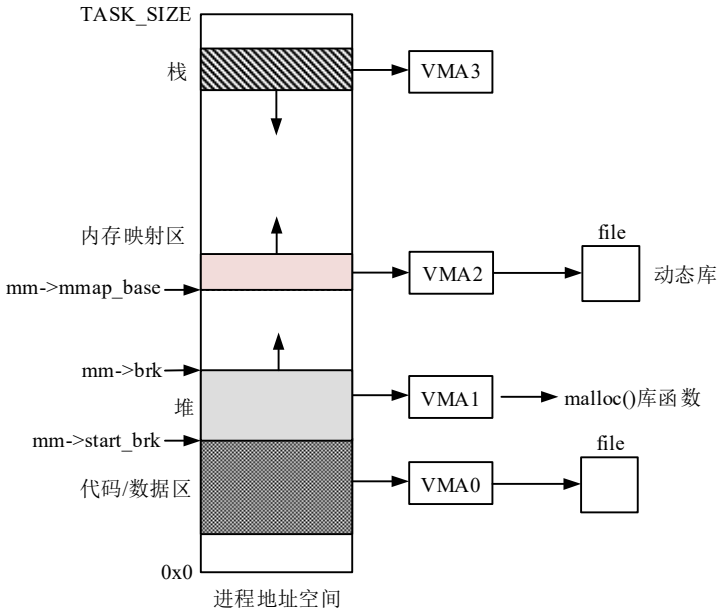
如果父（源）页表项 P 标记位为 1，表示映射到了物理页帧，则需要检查 VMA 标记，根据需要修改父、子页表项访问属性标记，增加 page 实例的引用和映射计数值，最后将修改后页表项数据写入子进程页表。

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）。

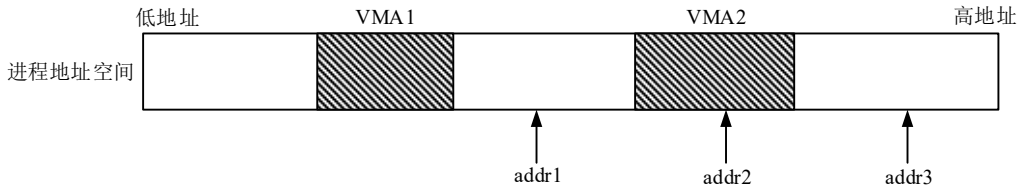
用户进程通过系统调用可以在内存映射区创建匿名映射或文件映射。栈在进程运行过程中由内核自动扩展。

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。

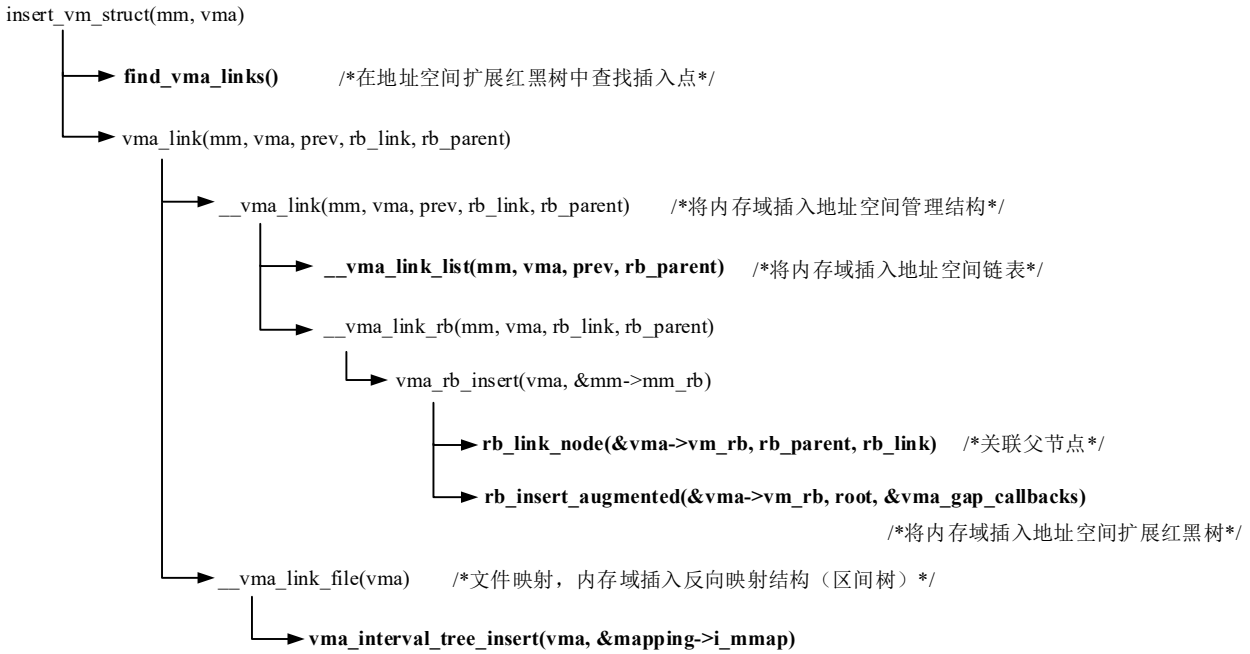
4.5.3 插入/移出 VMA

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;
}
```

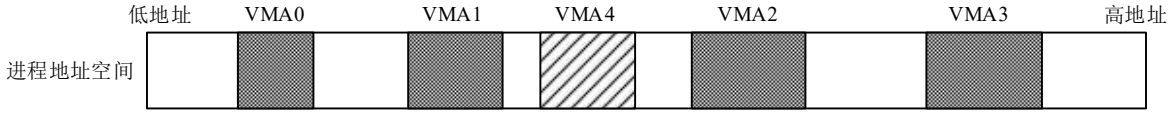
insert_vm_struct()函数首先判断 VMA 是否为匿名映射，如果是则 vm_area_struct 实例 vm_pgoff 成员赋值 VMA 起始虚拟页帧号。然后，调用 find_vma_links()函数查找 VMA 在地址空间（扩展）红黑树中的插入位置（关联父节点），新插入 VMA 不能与现有 VMA 有重叠，否则将返回错误码。最后，调用 vma_link()函数将 VMA 实例插入到地址空间管理链表和扩展红黑树中。如果是文件映射，还需要将 VMA 插入到文件地址空间管理的反向映射结构中，匿名映射在第一次建立映射时创建反向映射结构。

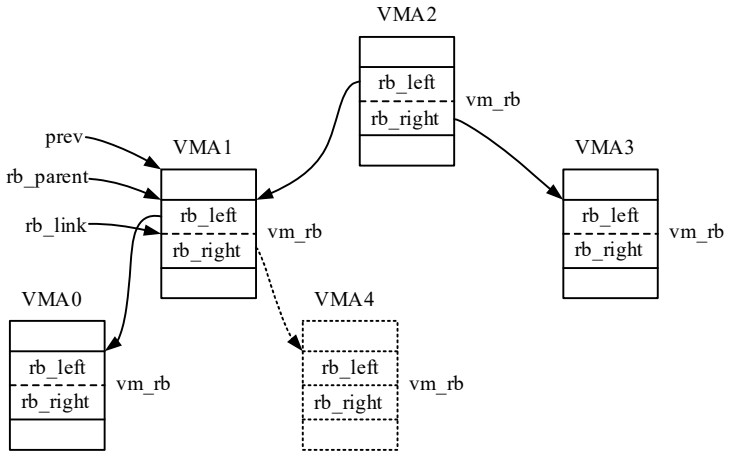
insert_vm_struct()函数调用关系如下图所示：



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。

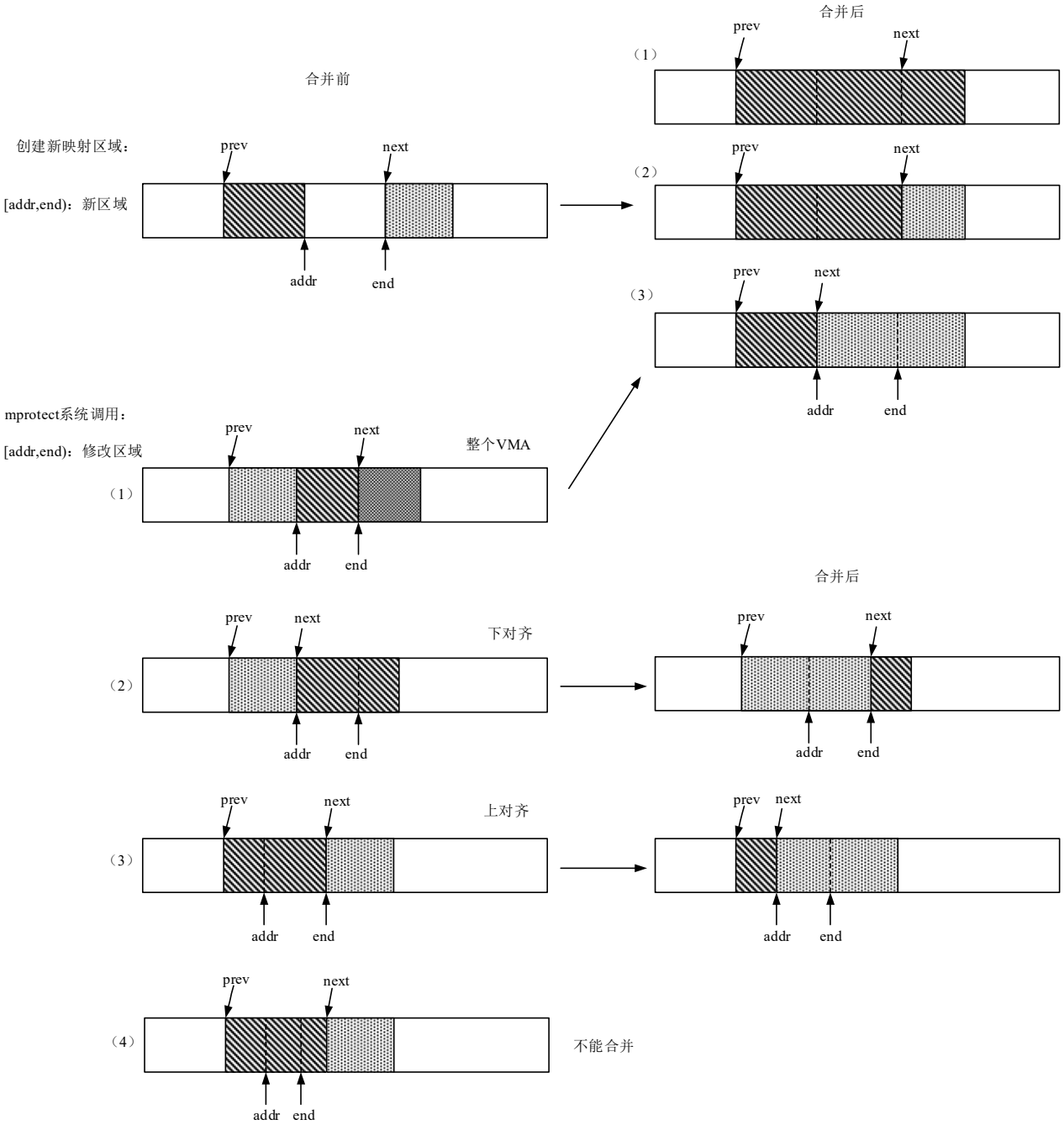
一、创建新映射时，获取的未映射区域[addr,end)与现有 VMA 不会有重叠。合并 VMA 时，判断[addr,end)能否与其前后 VMA 合并，能合并则执行合并，一共会有 4 种情况出现，如下所示：

（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。

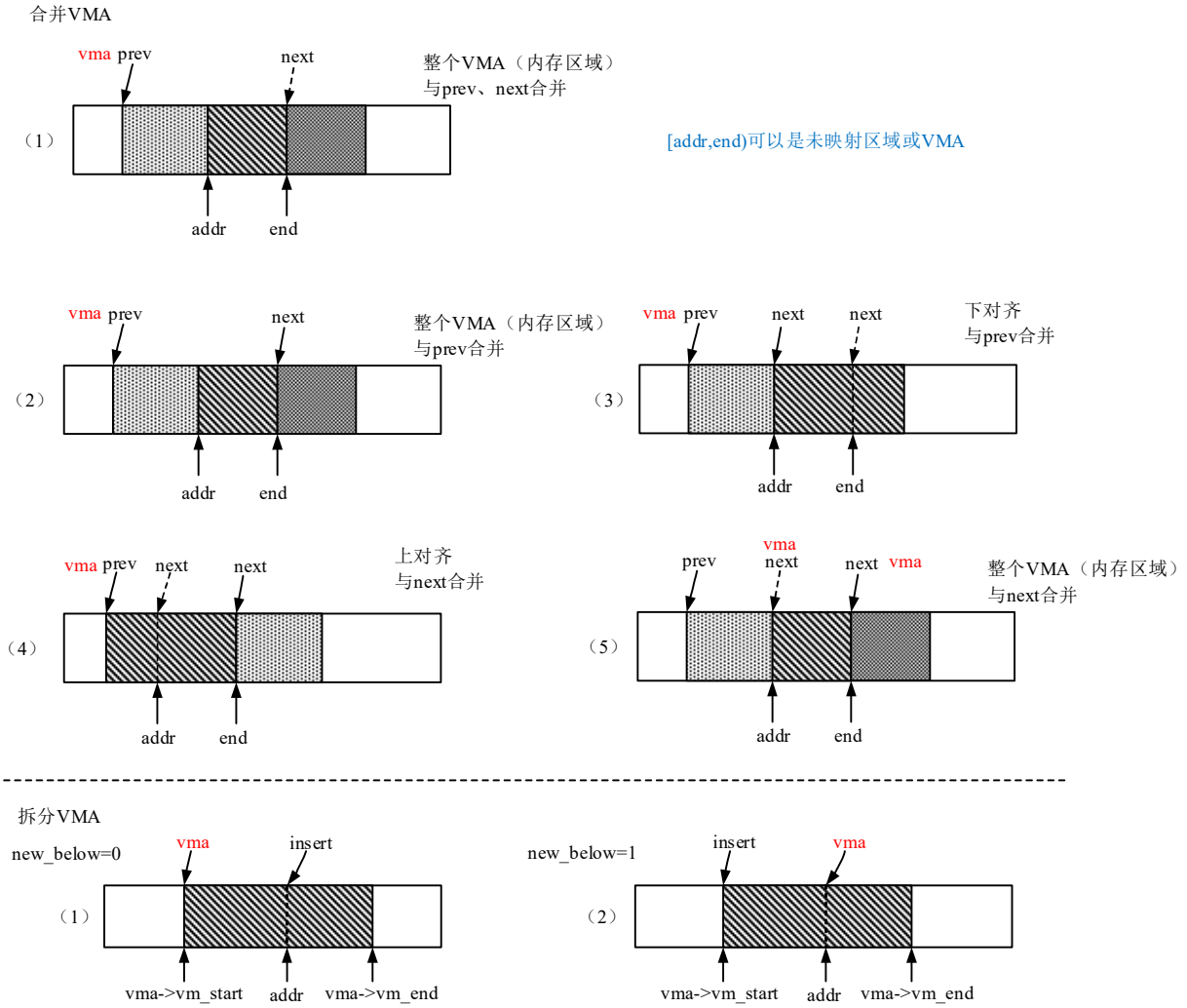
1 调整 VMA

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*/
```

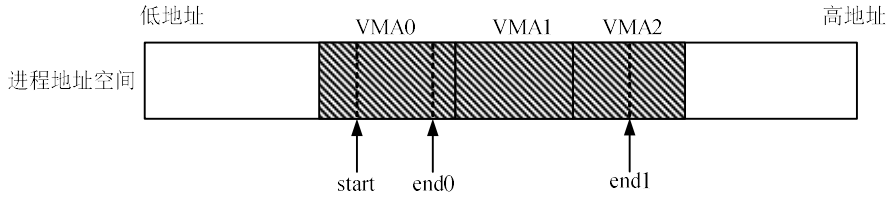
```
            vma = prev->vm_next;
            if (!vma || vma->vm_start != nstart) {       /*锁定/解锁内存区必须位于连续的 VMA 内部*/
                error = -ENOMEM;
                break;
            }
        }      /*遍历 VMA 结束*/
        return error;
    }
```

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

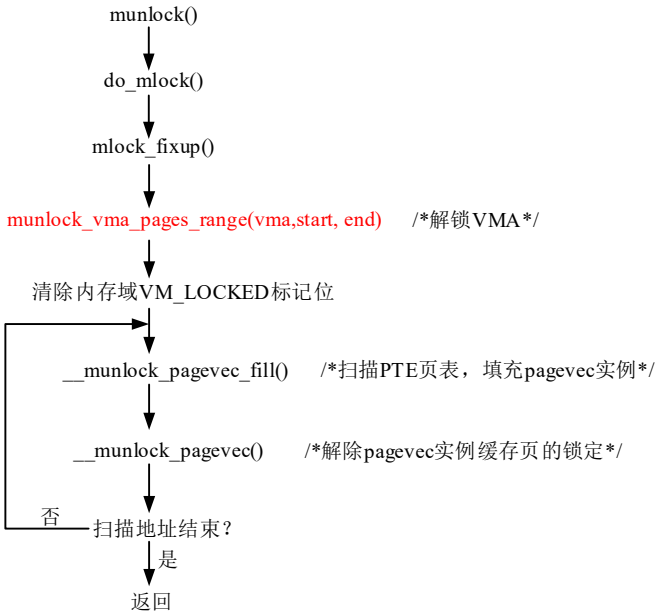
解锁内存区操作是解除内存区域的锁定（清除 VMA 标记 VM_LOCKED 标记位），并将映射页帧 page 实例添加到可回收 LRU 链表。注意，内核既可对已经锁定的内存区域执行解锁操作，也可以对没有锁定的内存区域执行解锁操作。

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

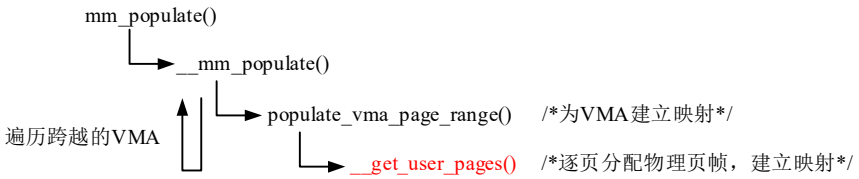
在前面介绍的锁定内存区和锁定所有内存域的系统调用中，将调用 mm_populate()或__mm_populate()函数为锁定区域逐页分配页帧，修改页表项，建立映射。这两个函数为用户空间一段连续使用的虚拟内存立即建立映射。

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。

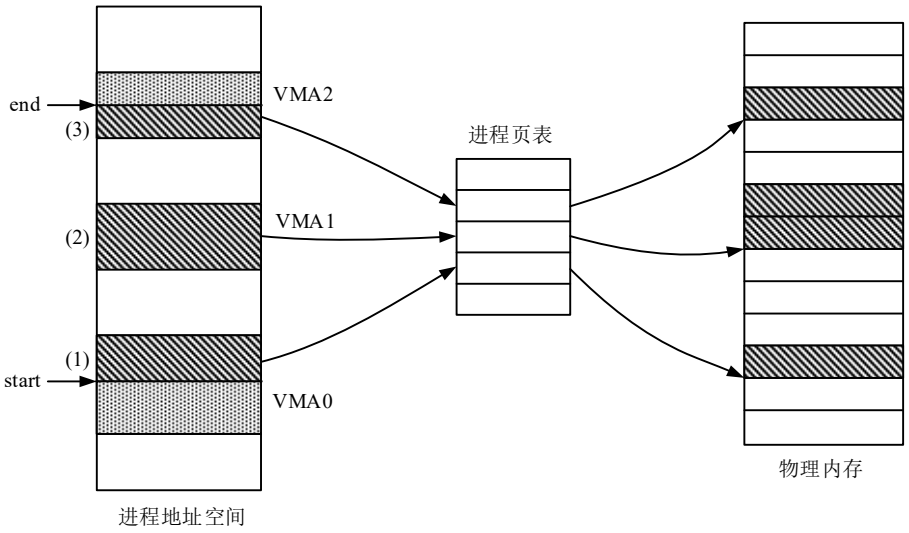
```
int   __mm_populate(unsigned long start, unsigned long len, int ignore_errors)
/*start：起始虚拟地址，len：长度，字节数，ignore_errors：是否忽略错误，1 忽略，0 不忽略*/
{
    struct mm_struct *mm = current->mm;      /*当前进程地址空间*/
    unsigned long end, nstart, nend;
    struct vm_area_struct *vma = NULL;
    int locked = 0;        /*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。

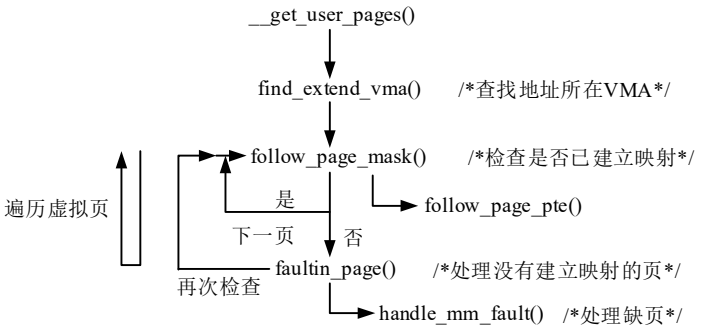
2 建立用户空间映射

__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;
```

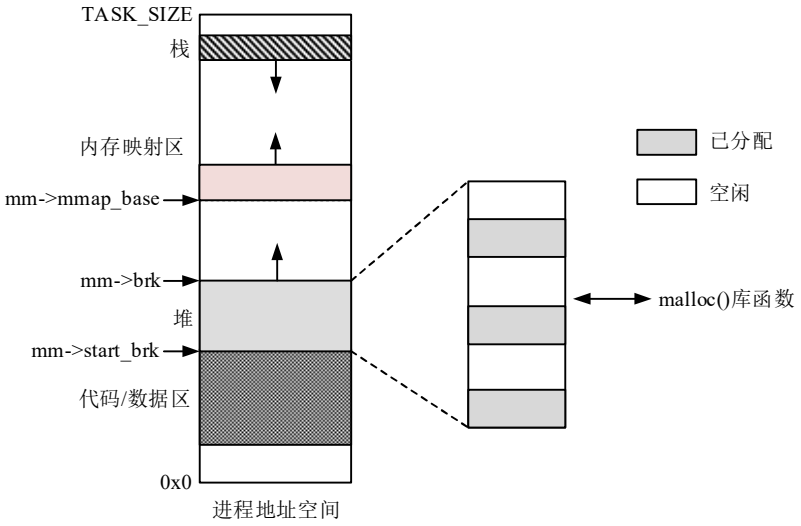
...

}

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。

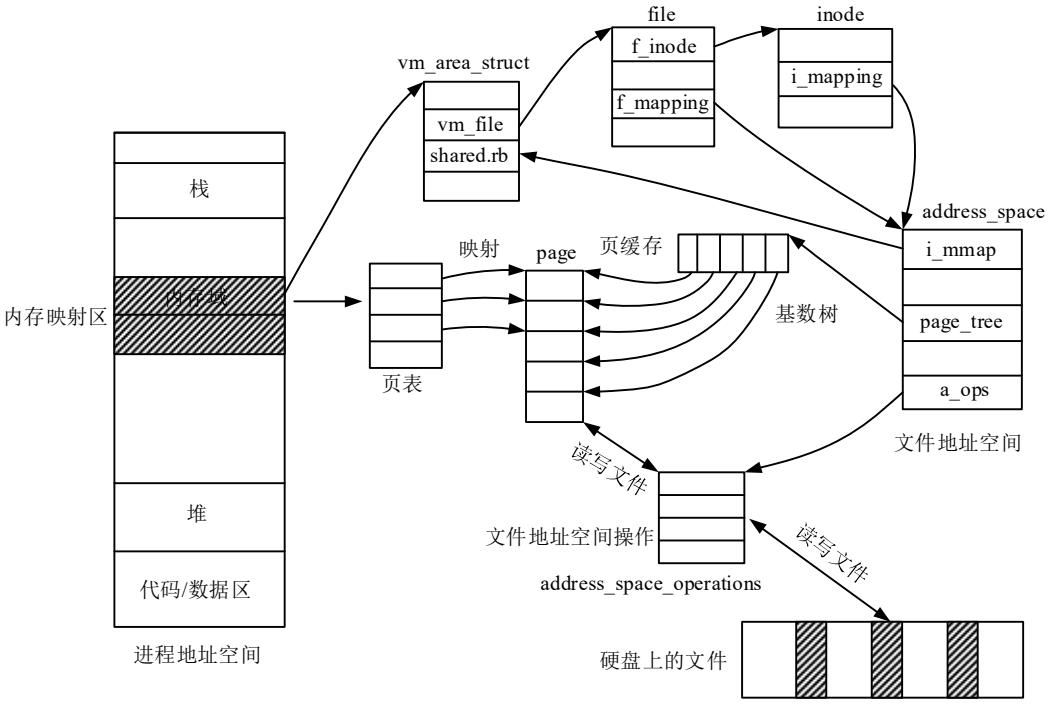
4.6.4 内存映射区

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：

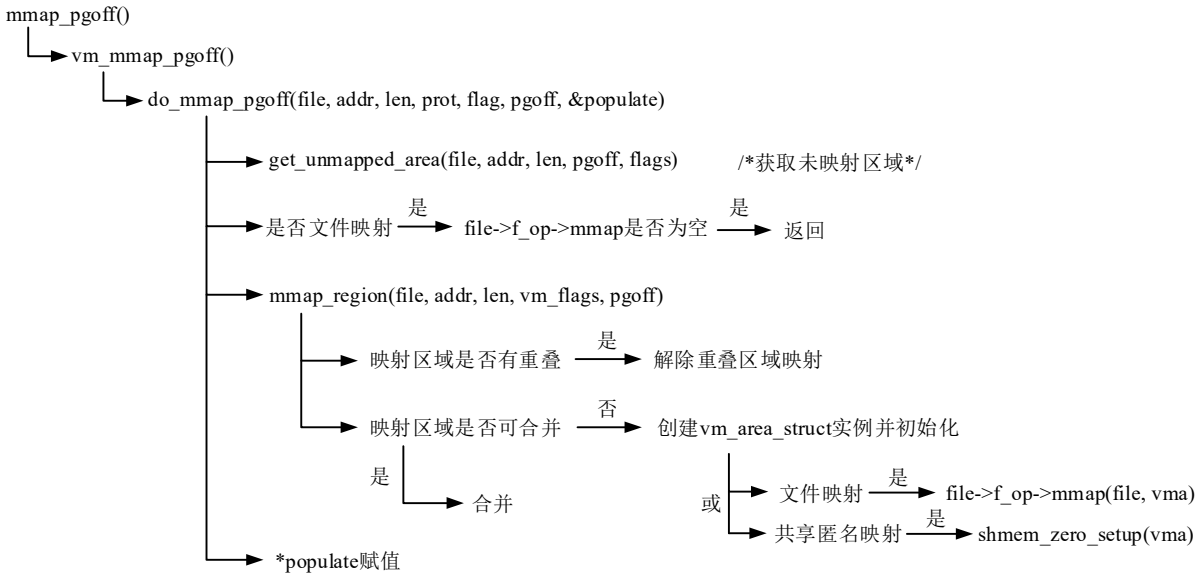
```
unsigned long vm_mmap_pgoff(struct file *file, unsigned long addr,unsigned long len, unsigned long prot, \
                                                    unsigned long flag, unsigned long pgoff)
{
    unsigned long ret;
    struct mm_struct *mm = current->mm;
    unsigned long populate;              /*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。

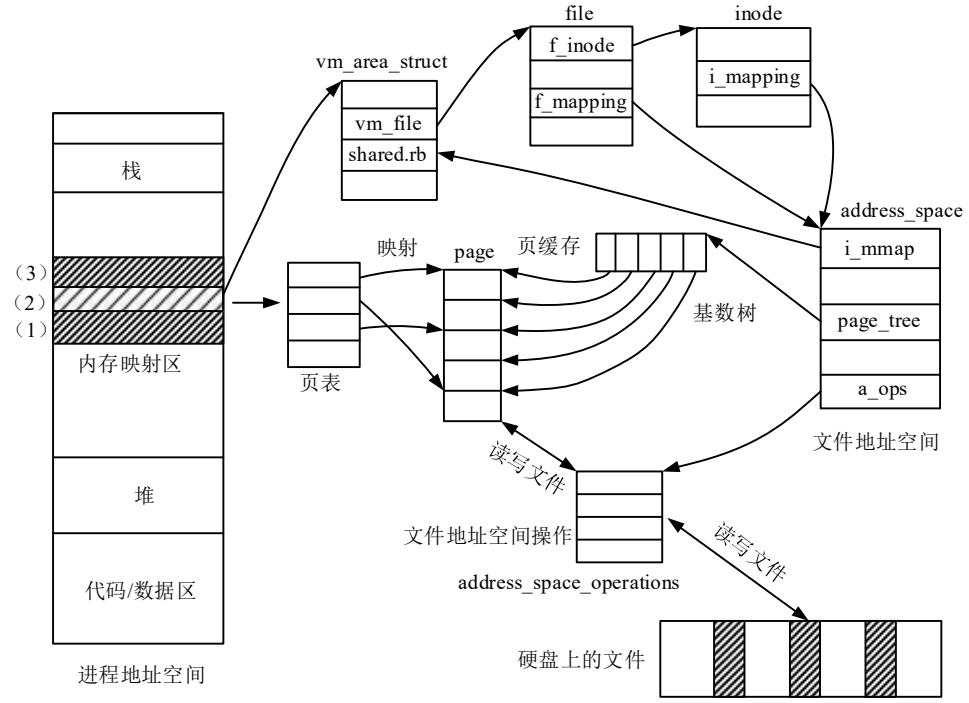
映射区域 vm_area_struct 实例将被插入到地址空间管理结构中，如果是文件映射 VMA 还将被插入文件地址空间区间树（反向映射结构），如果是匿名映射，反向映射结构初始化为空。mmap_region()函数在设置内存域实例 vm_flags 和 vm_page_prot 成员后，返回映射区域起始虚拟地址。

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。

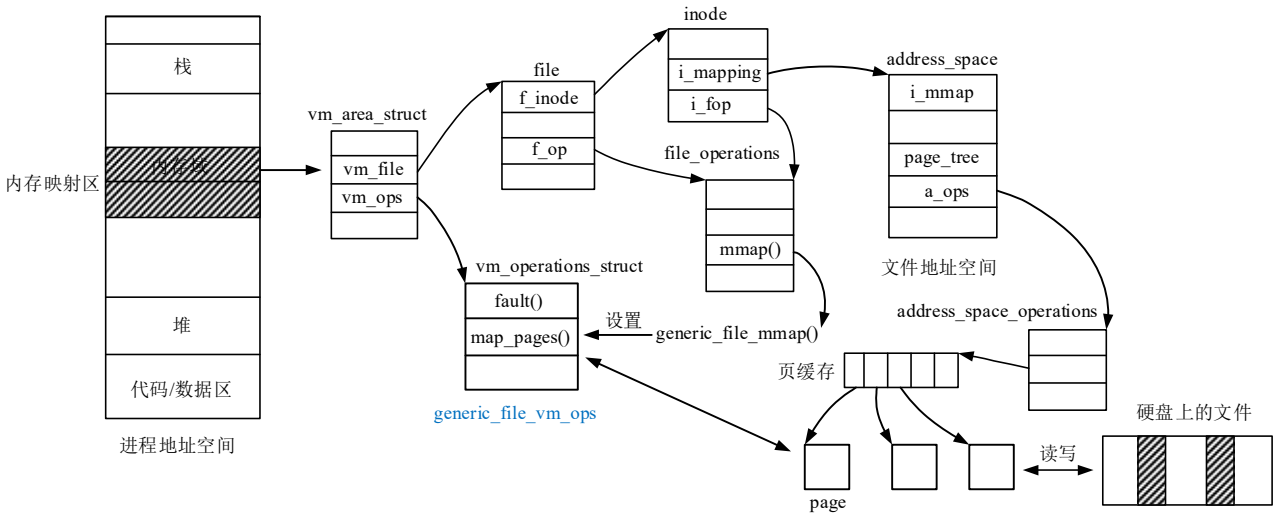
重映射操作调用前面介绍过的 do_mmap_pgoff()函数在重映射区域创建映射。do_mmap_pgoff()函数内部最终会调用 do_munmap()函数对原 VMA 进行拆分，拆分成三个 VMA，前后两个内存域映射保持不变，中间 VMA 解除映射后用于重映射。

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)
{...}
```

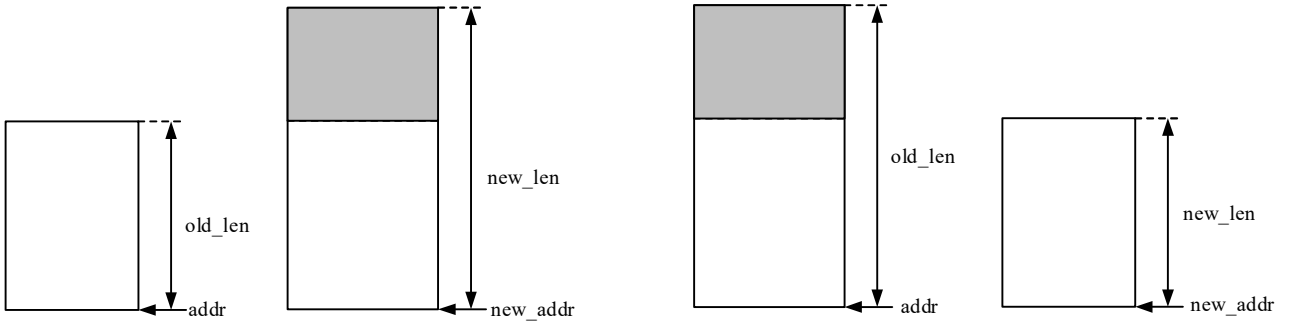
下面先分几种情形来介绍函数的功能，然后再分析其源代码：

（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。

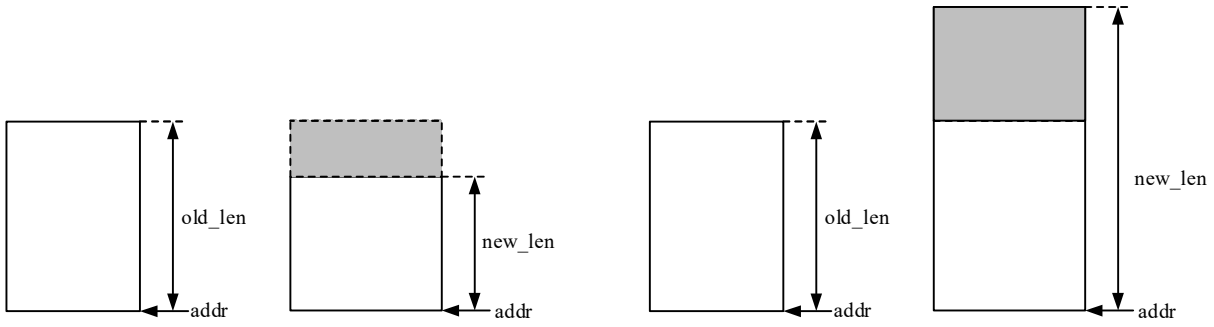
假设旧区域比新区域短（扩展 VMA），如下图中左侧所示，则在进程地址空间 new_addr 处 new_len 长度的未映射区域创建 VMA（若不能在此地址创建映射，返回错误码），复制旧区域映射至新 VMA（含页表项数据），解除旧区域映射。若需要为新区域立即建立映射，则还需要为扩展的区域立即建立映射（图中阴影部分）。

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：

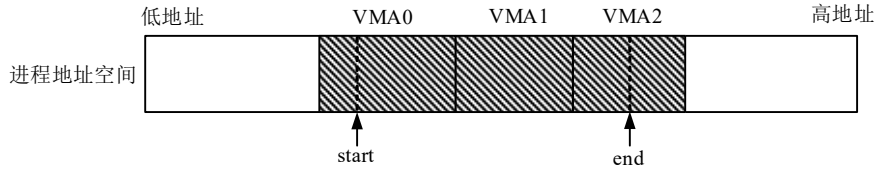
```
SYSCALL_DEFINE3(mprotect, unsigned long, start, size_t, len,unsigned long, prot)
/*start：被修改区域起始地址，必须页对齐且位于现有 VMA 内部，
*len：长度，字节数，prot：新访问权限。*/
{
    unsigned long vm_flags, nstart, end, tmp, reqprot;
    struct vm_area_struct *vma, *prev;
    int error = -EINVAL;
    const int grows = prot & (PROT_GROWSDOWN|PROT_GROWSUP);
    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*/
```

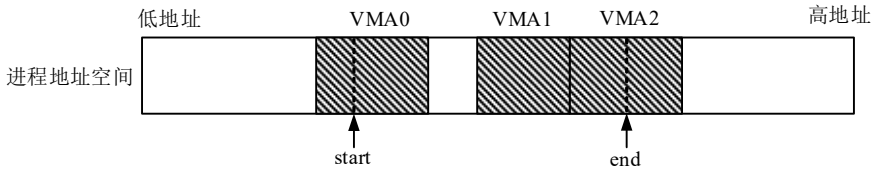
madvise()系统调用实现函数如下（/mm/madvise.c）：

```
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。

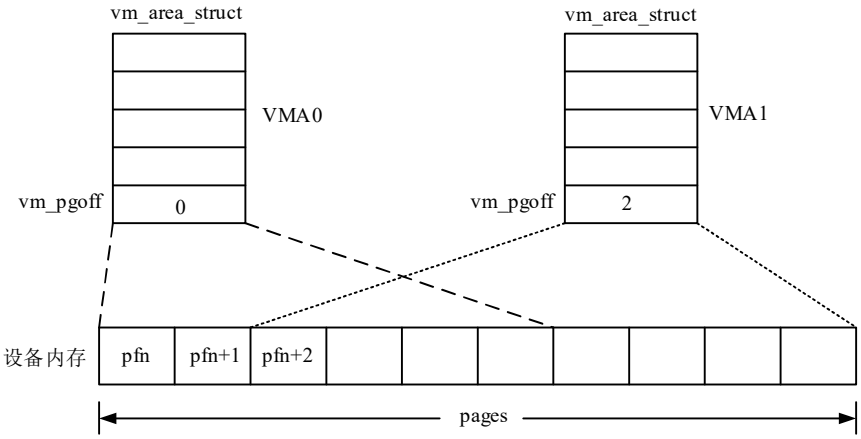
2 映射设备内存

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。

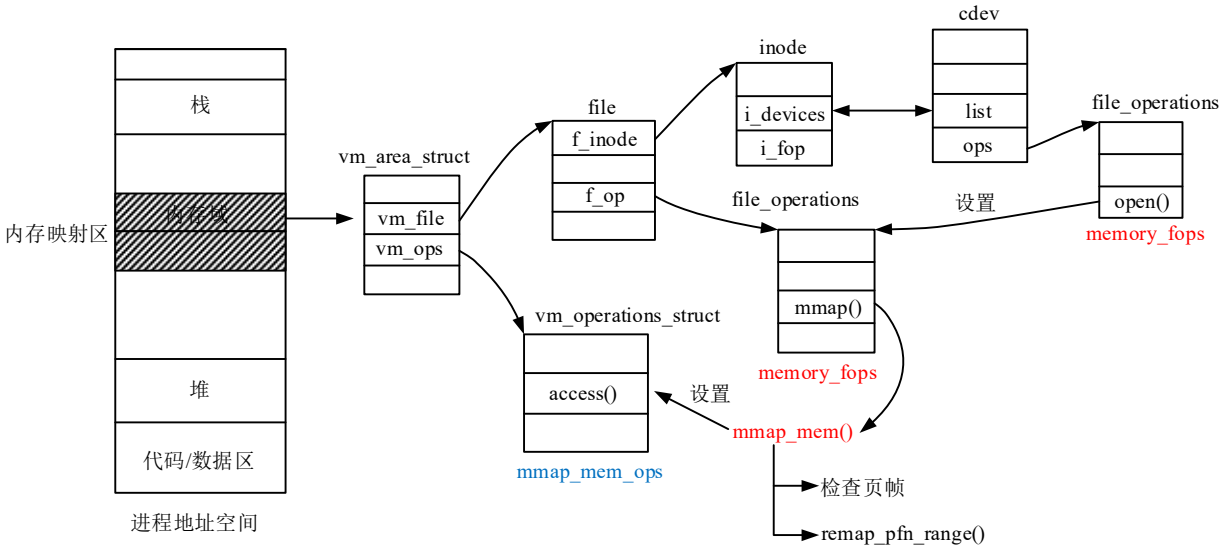
设备文件/dev/mem 表示系统中的物理内存，文件的内容就是物理内存中的数据。读者可以学习完第 7 章文件系统，以及第 9 章字符设备驱动程序后再回过头来学习本小节。

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）：

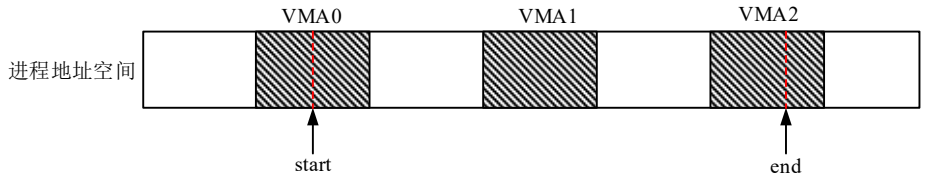
```
int vm_munmap(unsigned long start, size_t len)
{
    int ret;
    struct mm_struct *mm = current->mm;

    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。

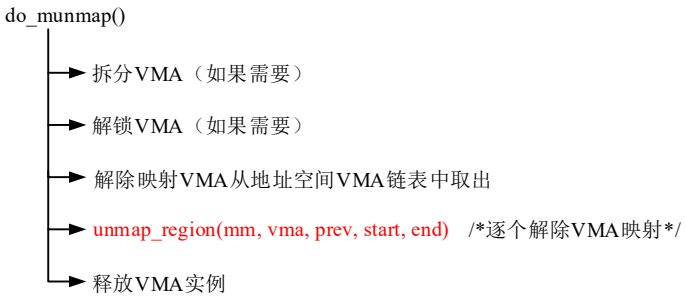
2 接口函数

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）：

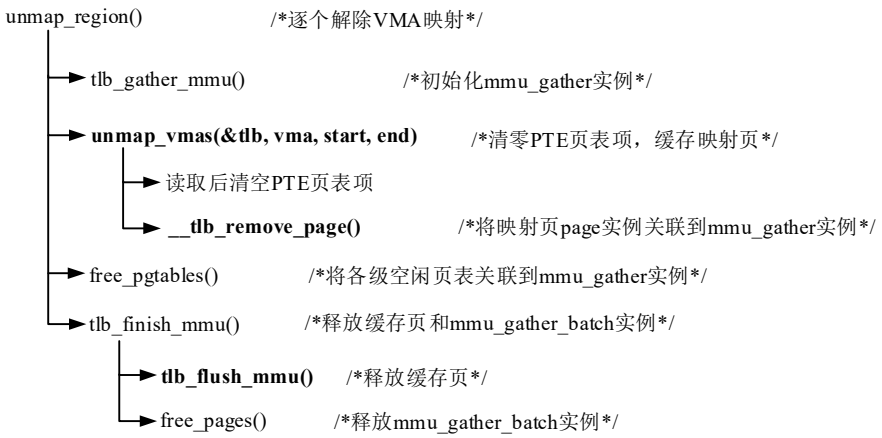
```
static void unmap_region(struct mm_struct *mm,       \
        struct vm_area_struct *vma, struct vm_area_struct *prev,unsigned long start, unsigned long end)
/*
*vma：指向解除映射 VMA 链表第一个成员，prev：指向 vma 前一个 VMA，
*start：起始虚拟地址，end：结束虚拟地址。
*/
{
    struct vm_area_struct *next = prev ? prev->vm_next : mm->mmap;
    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：

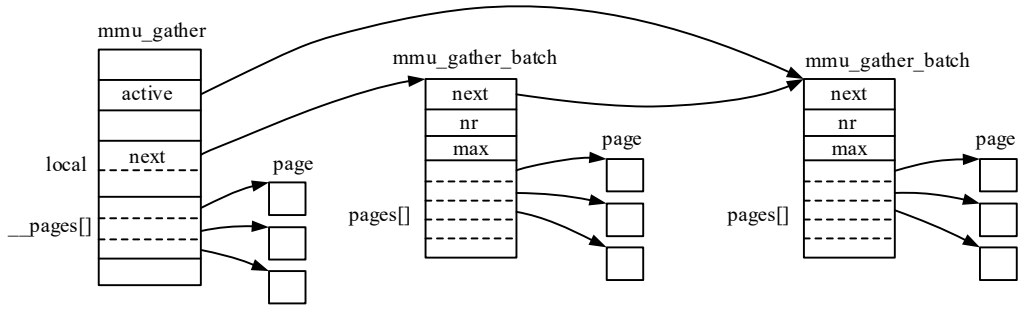
```
struct mmu_gather_batch {
    struct mmu_gather_batch   *next;    /*指向下一实例*/
    unsigned int        nr;           /*当前实例缓存页数量*/
    unsigned int        max;          /*当前实例缓存页最大数量，即 pages 指针数组项数*/
    struct page         *pages[0];    /*page 指针数组，必须是最后一个成员*/
```

};

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。

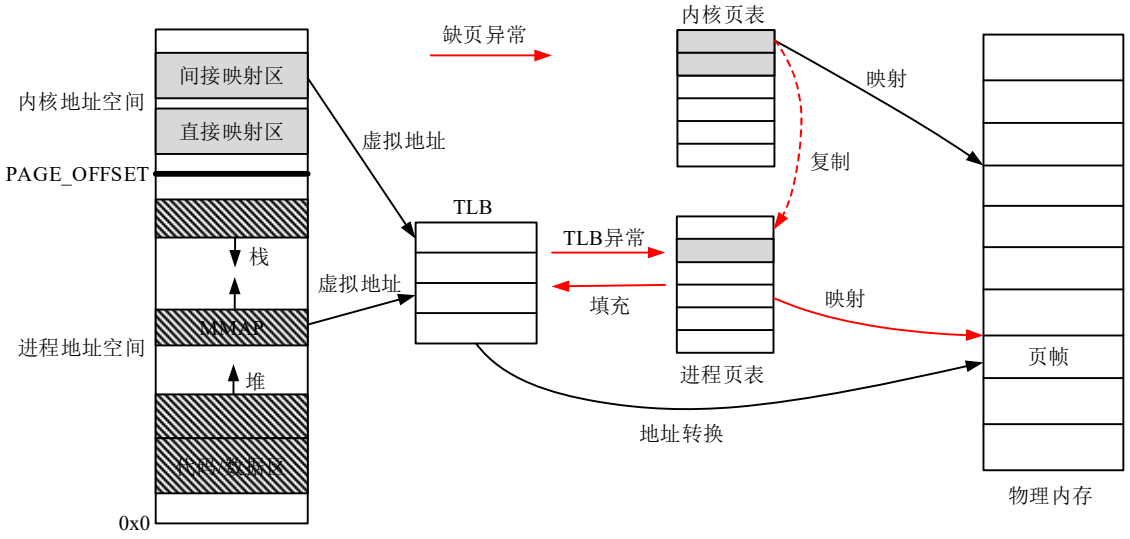
在创建进程时，进程地址空间未建立映射的 PTE 表项都写入了无效的表项。CPU 访问未建立映射的虚拟地址时将先引发 TLB 重填异常，重填异常处理程序会将无效的（但与虚拟地址匹配）页表项写入 TLB，重填异常返回。重填异常返回后，虚拟地址将匹配到无效的 TLB 表项，随后将触发 TLB 缺页（无效）异常，在缺页异常处理程序中建立虚拟页到物理页的映射。

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。

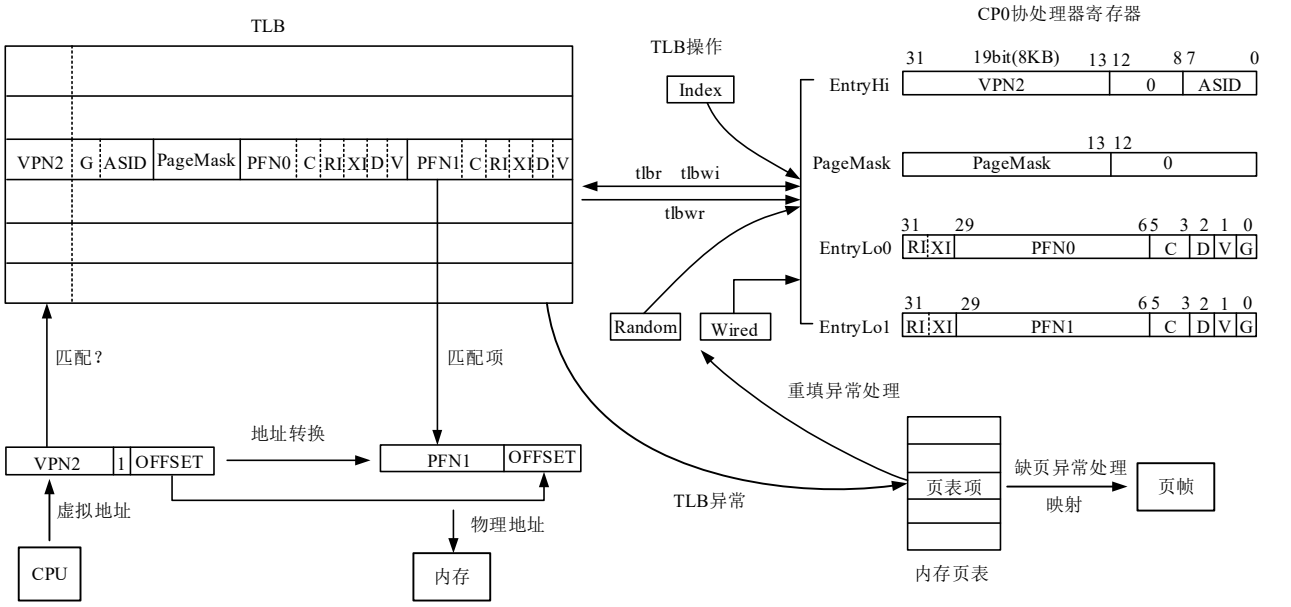
（3）CPU 访问未建立映射的文件映射页时，从文件地址空间中查找或创建文件内容的缓存页（从块设备中读取数据），依缓存页生成进程页表项，写入 TLB，随后可以正常地进行地址转换了。

（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。

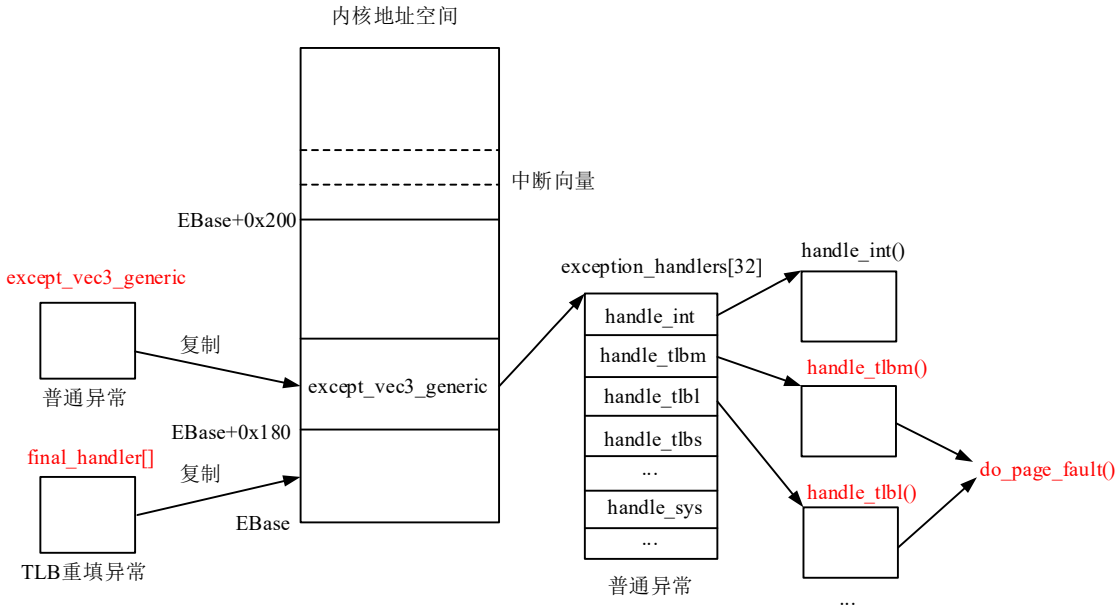
MIPS32 体系结构 TLB 异常的类型，发生的条件，异常的名称及异常处理程序的入口地址如下表所示：

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。

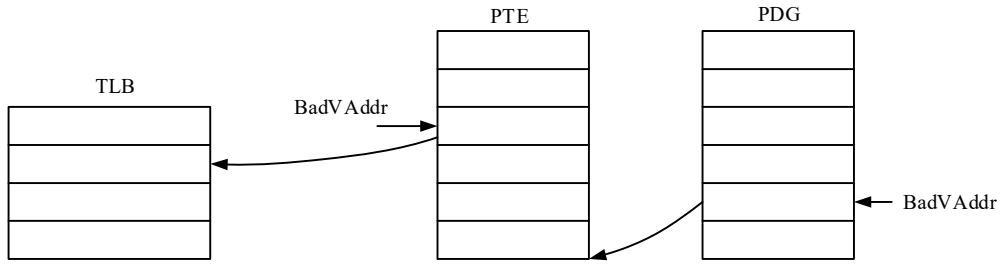
3 异常处理

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，则中断也有自己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）。

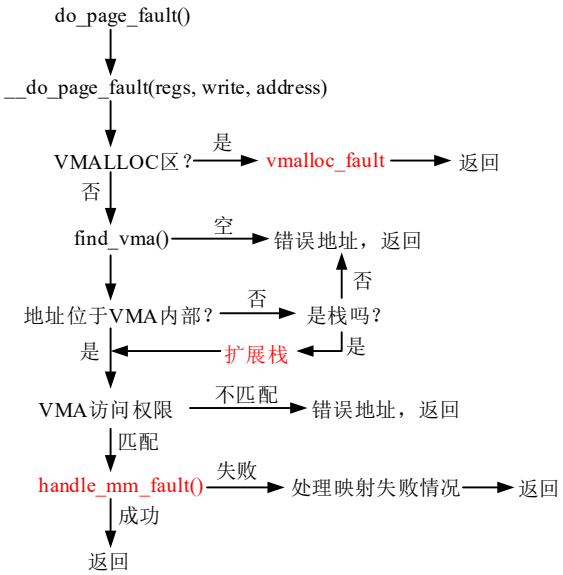
●void pagefault_disable(void)：关闭缺页异常处理，pagefault_disabled 加 1。

●#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。

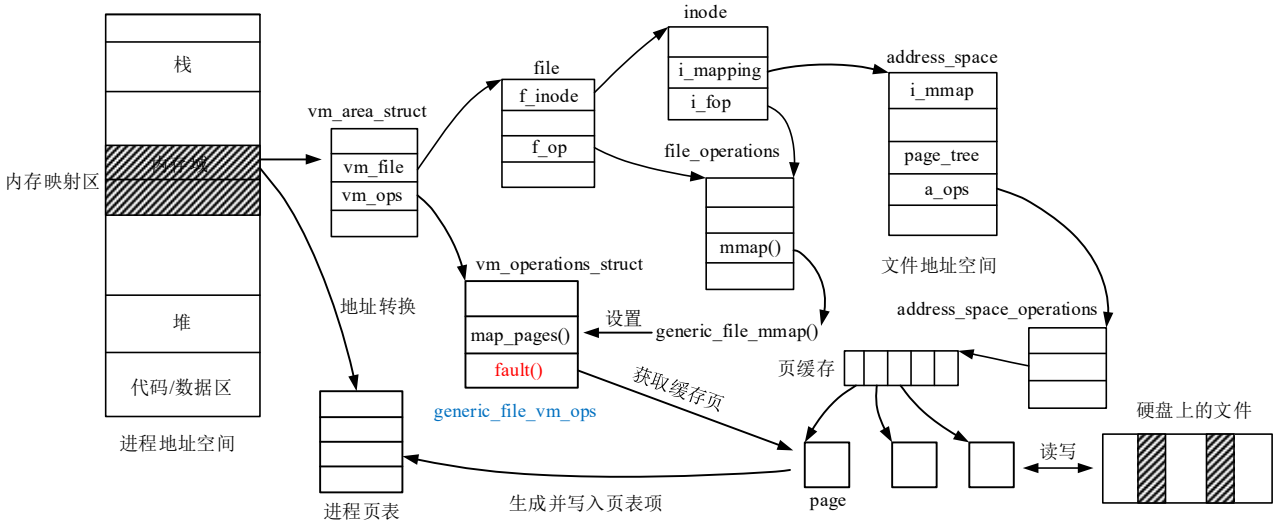
（3）对共享映射写操作触发的异常，需要查找（或创建）缓存页，并建立映射。

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。