Linux系统开发专栏

linux内核分析笔记----进程地址空间

前边我已经说过了内核是如何管理物理内存。但事实是内核是操作系统的核心，不光管理本身的内存，还要管理进程的地址空间。linux操作系统采用虚拟内存技术，所有进程之间以虚拟方式共享内存。进程地址空间由每个进程中的线性地址区组成，而且更为重要的特点是内核允许进程使用该空间中的地址。通常情况况下，每个进程都有唯一的地址空间，而且进程地址空间之间彼此互不相干。但是进程之间也可以选择共享地址空间，这样的进程就叫做线程。

内核使用内存描述符结构表示进程的地址空间，由结构体mm_struct结构体表示，定义在linux/sched.h中，如下：

```
struct mm_struct {
        struct vm_area_struct  *mmap;               /* list of memory areas */
        struct rb_root         mm_rb;               /* red-black tree of VMAs */
        struct vm_area_struct  *mmap_cache;         /* last used memory area */
        unsigned long          free_area_cache;     /* 1st address space hole */
        pgd_t                  *pgd;                /* page global directory */
        atomic_t               mm_users;            /* address space users */
        atomic_t               mm_count;            /* primary usage counter */
        int                    map_count;           /* number of memory areas */
        struct rw_semaphore    mmap_sem;            /* memory area semaphore */
        spinlock_t             page_table_lock;     /* page table lock */
        struct list_head       mmlist;              /* list of all mm_structs */
        unsigned long          start_code;          /* start address of code */
        unsigned long          end_code;            /* final address of code */
        unsigned long          start_data;          /* start address of data */
        unsigned long          end_data;            /* final address of data */
        unsigned long          start_brk;           /* start address of heap */
        unsigned long          brk;                 /* final address of heap */
        unsigned long          start_stack;         /* start address of stack */
        unsigned long          arg_start;           /* start of arguments */
        unsigned long          arg_end;             /* end of arguments */
        unsigned long          env_start;           /* start of environment */
        unsigned long          env_end;             /* end of environment */
        unsigned long          rss;                 /* pages allocated */
        unsigned long          total_vm;            /* total number of pages */
        unsigned long          locked_vm;           /* number of locked pages */
        unsigned long          def_flags;           /* default access flags */
        unsigned long          cpu_vm_mask;         /* lazy TLB switch mask */
        unsigned long          swap_address;        /* last scanned address */
        unsigned               dumpable:1;          /* can this mm core dump? */
        int                    used_hugetlb;        /* used hugetlb pages? */
        mm_context_t           context;             /* arch-specific data */
        int                    core_waiters;        /* thread core dump waiters */
        struct completion      *core_startup_done;  /* core start completion */
        struct completion      core_done;           /* core end completion */
        rwlock_t               ioctx_list_lock;     /* AIO I/O list lock */
        struct kioctx          *ioctx_list;         /* AIO I/O list */
        struct kioctx          default_kioctx;      /* AIO default I/O context */
};
```

mm_users记录了正在使用该地址的进程数目(比如有两个进程在使用，那就为2)。mm_count是该结构的主引用计数，只要mm_users不为0，它就为1。但其为0时，后者就为0。这时也就说明再也没有指向该mm_struct结构体的引用了，这时该结构体会被销毁。内核之所以同时使用这两个计数器是为了区别主使用计数器和使用该地址空间的进程的数目。mmap和mm_rb描述的都是同一个对象：该地址空间中的全部内存区域。不同只是前者以链表，后者以红黑树的形式组织。所有的mm_struct结构体都通过自身的mmlist域连接在一个双向链表中，该链表的首元素是init_mm内存描述符，它代表init进程的地址空间。另外需要注意，操作该链表的时候需要使用mmlist_lock锁来防止并发访问，该锁定义在文件kernel/fork.c中。内存描述符的总数在mmlist_nr全局变量中，该变量也定义在文件fork.c中。

我前边说过的进程描述符中有一个mm域，这里边存放的就是该进程使用的内存描述符，通过current->mm便可以指向当前进程的内存描述符。fork函数利用copy_mm()函数就实现了复制父进程的内存描述符，而子进程中的mm_struct结构体实际是通过文件kernel/fork.c中的allocate_mm()宏从mm_cachep slab缓存中分配得到的。通常，每个进程都有唯一的mm_struct结构体。

前边也说过，在linux中，进程和线程其实是一样的，唯一的不同点就是是否共享这里的地址空间。这个可以通过CLONE_VM标志来实现。linux内核并不区别对待它们，线程对内核来说仅仅是一个共向特定资源的进程而已。好了，如果你设置这个标志了，似乎很多问题都解决了。不再要allocate_mm函数了，前边刚说作用。而且在copy_mm()函数中将mm域指向其父进程的内存描述符就可以了，如下：

```
if (clone_flags & CLONE_VM) {
        /*
         * current is the parent process and
         * tsk is the child process during a fork()
         */
         atomic_inc(&current->mm->mm_users);
         tsk->mm = current->mm;
}
```

最后，当进程退出的时候，内核调用exit_mm()函数，这个函数调用mmput()来减少内存描述符中的mm_users用户计数。如果计数降为0，继续调用mmdrop函数，减少mm_count使用计数。如果使用计数也为0，则调用free_mm()宏通过kmem_cache_free()函数将mm_struct结构体归还到mm_cachep slab缓存中。

但对于内核而言，内核线程没有进程地址空间，也没有相关的内存描述符，内核线程对应的进程描述符中mm域也为空。但内核线程还是需要使用一些数据的，比如页表，为了避免内核线程为内存描述符和页表浪费内存，也为了当新内核线程运行时，避免浪费处理器周期向新地址空间进行切换，内核线程将直接使用前一个进程的内存描述符。回忆一下我刚说的进程调度问题，当一个进程被调度时，进程结构体中mm域指向的地址空间会被装载到内存，进程描述符中的active_mm域会被更新，指向新的地址空间。但我们这里的内核是没有mm域(为空)，所以，当一个内核线程被调度时，内核发现它的mm域为NULL，就会保留前一个进程的地址空间，随后内核更新内核线程对应的进程描述符中的active域，使其指向前一个进程的内存描述符。所以在需要的时候，内核线程便可以使用前一个进程的页表。因为内核线程不妨问用户空间的内存，所以它们仅仅使用地址空间中和内核内存相关的信息，这些信息的含义和普通进程完全相同。

内存区域由vm_area_struct结构体描述，定义在linux/mm.h中，内存区域在内核中也经常被称作虚拟内存区域或VMA.它描述了指定地址空间内连续区间上的一个独立内存范围。内核将每个内存区域作为一个单独的内存对象管理，每个内存区域都拥有一致的属性。结构体如下：

```
struct vm_area_struct {
        struct mm_struct             *vm_mm;        /* associated mm_struct */
        unsigned long                vm_start;      /* VMA start, inclusive */
        unsigned long                vm_end;        /* VMA end , exclusive */
        struct vm_area_struct        *vm_next;      /* list of VMA's */
        pgprot_t                     vm_page_prot;  /* access permissions */
        unsigned long                vm_flags;      /* flags */
        struct rb_node               vm_rb;         /* VMA's node in the tree */
        union {         /* links to address_space->i_mmap or i_mmap_nonlinear */
                struct {
                        struct list_head        list;
                        void                    *parent;
                        struct vm_area_struct   *head;
                } vm_set;
                struct prio_tree_node prio_tree_node;
        } shared;
        struct list_head             anon_vma_node;     /* anon_vma entry */
        struct anon_vma              *anon_vma;         /* anonymous VMA object */
        struct vm_operations_struct  *vm_ops;           /* associated ops */
        unsigned long                vm_pgoff;          /* offset within file */
        struct file                  *vm_file;          /* mapped file, if any */
        void                         *vm_private_data;  /* private data */
};
```

每个内存描述符都对应于地址进程空间中的唯一区间。vm_mm域指向和VMA相关的mm_struct结构体。两个独立的进程将同一个文件映射到各自的地址空间，它们分别都会有一个vm_area_struct结构体来标志自己的内存区域；但是如果两个线程共享一个地址空间，那么它们也同时共享其中的所有vm_area_struct结构体。

在上面的vm_flags域中存放的是VMA标志，标志了内存区域所包含的页面的行为和信息，反映了内核处理页面所需要遵循的行为准则，如下表下述：

Flag	Effect on the VMA and Its Pages
VM_READ	Pages can be read from.
VM_WRITE	Pages can be written to.
VM_EXEC	Pages can be executed.
VM_SHARED	Pages are shared.
VM_MAYREAD	The VM_READ flag can be set.
VM_MAYWRITE	The VM_WRITE flag can be set.
VM_MAYEXEC	The VM_EXEC flag can be set.
VM_MAYSHARE	The VM_SHARE flag can be set.
VM_GROWSDOWN	The area can grow downward.
VM_GROWSUP	The area can grow upward.
VM_SHM	The area is used for shared memory.
VM_DENYWRITE	The area maps an unwritable file.
VM_EXECUTABLE	The area maps an executable file.
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_SEQ_READ	The pages seem to be accessed sequentially.
VM_RAND_READ	The pages seem to be accessed randomly.
VM_DONTCOPY	This area must not be copied on fork().
VM_DONTEXPAND	This area cannot grow via mremap().
VM_RESERVED	This area must not be swapped out.
VM_ACCOUNT	This area is an accounted VM object.
VM_HUGETLB	This area uses hugetlb pages.
VM_NONLINEAR	This area is a nonlinear mapping.

上表已经相当详细了，而且给出了说明，我就不说了。在vm_area_struct结构体中的vm_ops域指向域指定内存区域相关的操作函数表，内核使用表中的方法操作VMA。vm_area_struct作为通用对象代表了任何类型的内存区域，而操作表描述针对特定的对象实例的特定方法。操作函数表由vm_operations_struct结构体表示，定义在linux/mm.h中，如下:

```
struct vm_operations_struct {
        void (*open) (struct vm_area_struct *);
        void (*close) (struct vm_area_struct *);
        struct page * (*nopage) (struct vm_area_struct *, unsigned long, int);
```

```
        int (*populate) (struct vm_area_struct *, unsigned long, unsigned long,pgprot_t, unsigned long,
int);
};
```

open:当指定的内存区域被加入到一个地址空间时，该函数被调用。

close：当指定的内存区域从地址空间删除时，该函数被调用。

nopages:当要访问的页不在物理内存中时，该函数被页错误处理程序调用。

populate：该函数被系统调用remap_pages调用来为将要发生的缺页中断预映射一个新映射。

记性好的你一定记得内存描述符中的mmap和mm_rb域都独立地指向与内存描述符相关的全体内存区域对象。它们包含完全相同的vm_area_struct结构体的指针，仅仅组织方式不同而已。前者以链表的方式进行组织，所有的区域按地址增长的方向排序，mmap域指向链表中第一个内存区域，链中最后一个VMA结构体指针指向空。而mm_rb域采用红--黑树连接所有的内存区域对象。它指向红--黑输的根节点。地址空间中每一个vm_area_struct结构体通过自身的vm_rb域连接到树中。关于红黑二叉树结构我就不细讲了，以后可能会详细说这个问题。内核之所以采用这两种结构来表示同一内存区域，主要是链表结构便于遍历所有节点，而红黑树结构体便于在地址空间中定位特定内存区域的节点。我么可以使用/proc文件系统和pmap工具查看给定进程的内存空间和其中所包含的内存区域。这里就不细说了。

内核也为我们提供了对内存区域操作的API，定义在linux/mm.h中：

(1)find_vma<定义在mm/mmap.c>中，该函数在指定的地址空间中搜索一个vm_end大于addr的内存区域。换句话说，该函数寻找第一个包含
addr或者首地址大于addr的内存区域，如果没有发现这样的区域，该函数返回NULL；否则返回指向匹配的内存区域的vm_area_struct结构体指针。
(2)find_vma_prev().函数定义和声明分别在文件mm/mmap.c中和文件linux/mm.h中，它和find_vma()工作方式相同，但返回的是第一个小于addr的VMA.
(3)find_vma_intersection().定义在文件linux/mm.h中，返回第一个和指定地址区间相交的VMA,该函数是一个内敛函数。

接下来要说的两个函数就非常重要了，它们负责创建和删除地址空间。

内核使用do_mmap()函数创建一个新的线性地址空间。但如果创建的地址区间和一个已经存在的地址区间相邻，并且它们具有相同的访问权限的话，那么两个区间将合并为一个。如果不能合并，那么就确实需要创建一个新的vma了，但无论哪种情况，do_mmap()函数都会将一个地址区间加入到进程的地址空间中。这个函数定义在linux/mm.h中，如下：

```
unsigned long do_mmap(struct file *file, unsigned long addr, unsigned long len, unsigned long prot,unsigned
long flag, unsigned long offset)
```

这个函数中由file指定文件，具体映射的是文件中从偏移offset处开始，长度为len字节的范围内的数据，如果file参数是NULL并且offset参数也是0，那么就代表这次映射没有和文件相关，该情况被称作匿名映射。如果指定了文件和偏移量，那么该映射被称为文件映射(file-backed mapping)，其中参数prot指定内存区域中页面的访问权限，这些访问权限定义在asm/mman.h中，如下：

Flag	Effect on the Pages in the New Interval
PROT_READ	Corresponds to VM_READ
PROT_WRITE	Corresponds to VM_WRITE
PROT_EXEC	Corresponds to VM_EXEC
PROT_NONE	Page cannot be accessed

flag参数指定了VMA标志，这些标志定义在asm/mman.h中，如下：

Flag	Effect on the New Interval
MAP_SHARED	The mapping can be shared
MAP_PRIVATE	The mapping cannot be shared
MAP_FIXED	The new interval *must* start at the given address addr
MAP_ANONYMOUS	The mapping is not file-backed, but is anonymous
MAP_GROWSDOWN	Corresponds to VM_GROWSDOWN
MAP_DENYWRITE	Corresponds to VM_DENYWRITE
MAP_EXECUTABLE	Corresponds to VM_EXECUTABLE
MAP_LOCKED	Corresponds to VM_LOCKED
MAP_NORESERVE	No need to reserve space for the mapping
MAP_POPULATE	Populate (prefault) page tables
MAP_NONBLOCK	Do not block on I/O

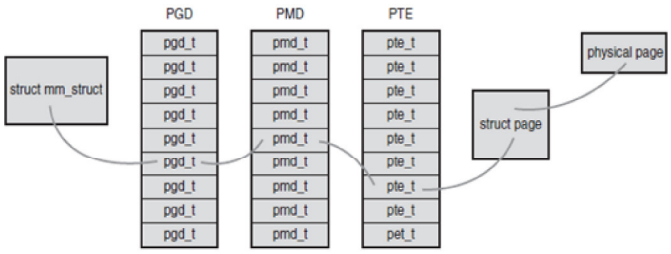
如果系统调用do_mmap的参数中有无效参数，那么它返回一个负值；否则，它会在虚拟内存中分配一个合适的新内存区域，如果有可能的话，将新区域和临近区域进行合并，否则内核从vm_area_cach
ep长字节缓存中分配一个vm_area_struct结构体，并且使用vma_link()函数将新分配的内存区域添加到地址空间的内存区域链表和红黑树中，随后还要更新内存描述符中的total_vm域，然后才返回新分配的地址区间的初始地址。在用户空间，我们可以通过mmap()系统调用获取内核函数do_mmap()的功能，这个在unix环境高级编程中讲的很详细，我就不好意思继续说了。我们继续往下走。
我们说既然有了创建，当然要有删除了，是不？do_mummp()函数就是干这事的。它从特定的进程地址空间中删除指定地址空间，该函数定义在文件linux/mm.h中，如下：

```
int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
```

第一个参数指定要删除区域所在的地址空间，删除从地址start开始，长度为len字节的地址空间，如果成功，返回0，否则返回负的错误码。与之相对应的用户空间系统调用是munmap。

下面开始最后一点内容：页表

我们知道应用程序操作的对象是映射到物理内存之上的虚拟内存，但是处理器直接操作的确实物理内存。所以当应用程序访问一个虚拟地址时，首先必须将虚拟地址转化为物理地址，然后处理器才能解析地址访问请求。这个转换工作需要通过查询页面才能完成，概括地讲，地址转换需要将虚拟地址分段，使每段虚地址都作为一个索引指向页表，而页表项则指向下一级别的页表或者指向最终的物理页面。linux中使用三级页表完成地址转换。多数体系结构中，搜索页表的工作由硬件完成，下表描述了虚拟地址通过页表找到物理地址的过程：



在上面这个图中，顶级页表是页全局目录(PGD)，二级页表是中间页目录(PMD).最后一级是页表(PTE),该页表结构指向物理页。上图中的页表对应的结构体定义在文件asm/page.h中。为了加快查找速度，在linux中实现了快表(TLB),其本质是一个缓冲器，作为一个将虚拟地址映射到物理地址的硬件缓存，当请求访问一个虚拟地址时，处理器将首先检查TLB中是否缓存了该虚拟地址到物理地址的映射，如果找到了，物理地址就立刻返回，否则，就需要再通过页表搜索需要的物理地址。