进程地址空间也就是每个进程所使用的内存，内核对进程地址空间的管理，也就是对用户态程序的内存管理。

**主要内容**：

* 地址空间(mm\_struct)
* 虚拟内存区域(VMA)
* 地址空间和页表

### **1. 地址空间(mm\_struct)**

地址空间就是每个进程所能访问的内存地址范围。

这个地址范围不是真实的，是虚拟地址的范围，有时甚至会超过实际物理内存的大小。

现代的操作系统中进程都是在保护模式下运行的，地址空间其实是操作系统给进程用的一段连续的虚拟内存空间。

地址空间最终会通过页表映射到物理内存上，因为内核操作的是物理内存。

虽然地址空间的范围很大，但是进程也不一定有权限访问全部的地址空间(一般都是只能访问地址空间中的一些地址区间)，

进程能够访问的那些地址区间也称为 内存区域。

进程如果访问了有效内存区域以外的内容就会报 “段错误” 信息。

内存区域中主要包含以下信息：

* - 代码段(text section)，即可执行文件代码的内存映射
* - 数据段(data section)，即可执行文件的已初始化全局变量的内存映射
* - bss段的零页(页面信息全是0值)，即未初始化全局变量的内存映射
* - 进程用户空间栈的零页内存映射
* - 进程使用的C库或者动态链接库等共享库的代码段，数据段和bss段的内存映射
* - 任何内存映射文件
* - 任何共享内存段
* - 任何匿名内存映射，比如由 malloc() 分配的内存

**注**：**bss**是 block started by symbol 的缩写。

linux中内存相关的概念稍微整理了一下，供参考：

|  |  |
| --- | --- |
| **英文** | **含义** |
| SIZE | 进程映射的内存大小，这不是进程实际使用的内存大小 |
| RSS(Resident set size) | 实际驻留在“内存”中的内存大小，不包含已经交换出去的内存 |
| SHARE | RSS中与其他进程共享的内存大小 |
| VMSIZE | 进程占用的总地址空间，包含没有映射到内存中的页 |
| Private RSS | 仅由进程单独占用的RSS，也就是进程实际占用的内存 |

#### **1.1 mm\_struct介绍**

linux中的地址空间是用 mm\_struct 来表示的。

下面对其中一些关键的属性进行了注释，有些属性我也不是很了解......

struct mm\_struct {

struct vm\_area\_struct \* mmap; /\* [内存区域]链表 \*/

struct rb\_root mm\_rb; /\* [内存区域]红黑树 \*/

struct vm\_area\_struct \* mmap\_cache; /\* 最近一次访问的[内存区域] \*/

unsigned long (\*get\_unmapped\_area) (struct file \*filp,

unsigned long addr, unsigned long len,

unsigned long pgoff, unsigned long flags); /\* 获取指定区间内一个还未映射的地址，出错时返回错误码 \*/

void (\*unmap\_area) (struct mm\_struct \*mm, unsigned long addr); /\* 取消地址 addr 的映射 \*/

unsigned long mmap\_base; /\* 地址空间中可以用来映射的首地址 \*/

unsigned long task\_size; /\* 进程的虚拟地址空间大小 \*/

unsigned long cached\_hole\_size; /\* 如果不空的话，就是 free\_area\_cache 后最大的空洞 \*/

unsigned long free\_area\_cache; /\* 地址空间的第一个空洞 \*/

pgd\_t \* pgd; /\* 页全局目录 \*/

atomic\_t mm\_users; /\* 使用地址空间的用户数 \*/

atomic\_t mm\_count; /\* 实际使用地址空间的计数， (users count as 1) \*/

int map\_count; /\* [内存区域]个数 \*/

struct rw\_semaphore mmap\_sem; /\* 内存区域信号量 \*/

spinlock\_t page\_table\_lock; /\* 页表锁 \*/

struct list\_head mmlist; /\* 所有地址空间形成的链表 \*/

/\* Special counters, in some configurations protected by the

\* page\_table\_lock, in other configurations by being atomic.

\*/

mm\_counter\_t \_file\_rss;

mm\_counter\_t \_anon\_rss;

unsigned long hiwater\_rss; /\* High-watermark of RSS usage \*/

unsigned long hiwater\_vm; /\* High-water virtual memory usage \*/

unsigned long total\_vm, locked\_vm, shared\_vm, exec\_vm;

unsigned long stack\_vm, reserved\_vm, def\_flags, nr\_ptes;

unsigned long start\_code, end\_code, start\_data, end\_data; /\* 代码段，数据段的开始和结束地址 \*/

unsigned long start\_brk, brk, start\_stack; /\* 堆的首地址，尾地址，进程栈首地址 \*/

unsigned long arg\_start, arg\_end, env\_start, env\_end; /\* 命令行参数，环境变量首地址，尾地址 \*/

unsigned long saved\_auxv[AT\_VECTOR\_SIZE]; /\* for /proc/PID/auxv \*/

struct linux\_binfmt \*binfmt;

cpumask\_t cpu\_vm\_mask;

/\* Architecture-specific MM context \*/

mm\_context\_t context;

/\* Swap token stuff \*/

/\*

\* Last value of global fault stamp as seen by this process.

\* In other words, this value gives an indication of how long

\* it has been since this task got the token.

\* Look at mm/thrash.c

\*/

unsigned int faultstamp;

unsigned int token\_priority;

unsigned int last\_interval;

unsigned long flags; /\* Must use atomic bitops to access the bits \*/

struct core\_state \*core\_state; /\* coredumping support \*/

#ifdef CONFIG\_AIO

spinlock\_t ioctx\_lock;

struct hlist\_head ioctx\_list;#endif

#ifdef CONFIG\_MM\_OWNER

/\*

\* "owner" points to a task that is regarded as the canonical

\* user/owner of this mm. All of the following must be true in

\* order for it to be changed:

\*

\* current == mm->owner

\* current->mm != mm

\* new\_owner->mm == mm

\* new\_owner->alloc\_lock is held

\*/

struct task\_struct \*owner;#endif

#ifdef CONFIG\_PROC\_FS

/\* store ref to file /proc/<pid>/exe symlink points to \*/

struct file \*exe\_file;

unsigned long num\_exe\_file\_vmas;#endif

#ifdef CONFIG\_MMU\_NOTIFIER

struct mmu\_notifier\_mm \*mmu\_notifier\_mm;#endif

};

**补充说明1：** 上面的属性中，mm\_users 和 mm\_count 很容易混淆，这里特别说明一下：(下面的内容有网上查找的，也有我自己理解的)

mm\_users 比较好理解，就是 mm\_struct 被用户空间进程(线程)引用的次数。

如果进程A中创建了3个新线程，那么 进程A(这时候叫线程A也可以)对应的 mm\_struct 中的 mm\_users = 4

补充一点，linux中进程和线程几乎没有什么区别，就是看它是否共享进程地址空间，共享进程地址空间就是线程，反之就是进程。

所以，如果子进程和父进程共享了进程地址空间，那么父子进程都可以看做线程。如果父子进程没有共享进程地址空间，就是2个进程

mm\_count 则稍微有点绕人，其实它记录就是 mm\_struct 实际的引用计数。

简单点说，当 mm\_users=0 时，并不一定能释放此 mm\_struct，只有当 mm\_count=0 时，才可以确定释放此 mm\_struct

从上面的解释可以看出，可能引用 mm\_struct 的并不只是用户空间的进程(线程)

当 mm\_users>0 时， mm\_count 会增加1, 表示有用户空间进程(线程)在使用 mm\_struct。不管使用 mm\_struct 的用户进程(线程)有几个， mm\_count 都只是增加1。

也就是说，如果只有1个进程使用 mm\_struct，那么 mm\_users=1，mm\_count也是 1。

如果有9个线程在使用 mm\_struct，那么 mm\_users=9，而 mm\_count 仍然为 1。

那么 mm\_count 什么情况下会大于 1呢？

当有内核线程使用 mm\_struct 时，mm\_count 才会再增加 1。

内核线程为何会使用用户空间的 mm\_struct 是有其他原因的，这个后面再阐述。这里先知道内核线程使用 mm\_struct 时也会导致 mm\_count 增加 1。

在下面这种情况下，mm\_count 就很有必要了：

* - 进程A启动，并申请了一个 mm\_struct，此时 mm\_users=1, mm\_count=1
* - 进程A中新建了2个线程，此时 mm\_users=3, mm\_count=1
* - 内核调度发生，进程A及相关线程都被挂起，一个内核线程B 使用了进程A 申请的 mm\_struct，此时 mm\_users=3, mm\_count=2
* - CPU的另一个core调度了进程A及其线程，并且执行完了进程A及其线程的所有操作，也就是进程A退出了。此时 mm\_users=0, mm\_count=1
* 在这里就看出 mm\_count 的用处了，如果只有 mm\_users 的话，这里 mm\_users=0 就会释放 mm\_struct，从而有可能导致 内核线程B 异常。
* - 内核线程B 执行完成后退出，这时 mm\_users=0，mm\_count=0，可以安全释放 mm\_struct 了

**补充说明2**：为何内核线程会使用用户空间的 mm\_struct？

对Linux来说，用户进程和内核线程都是task\_struct的实例，

唯一的区别是内核线程是没有进程地址空间的（内核线程使用的内核地址空间），内核线程的mm描述符是NULL，即内核线程的tsk->mm域是空（NULL）。

内核调度程序在进程上下文的时候，会根据tsk->mm判断即将调度的进程是用户进程还是内核线程。

但是虽然内核线程不用访问用户进程地址空间，但是仍然需要页表来访问内核自己的空间。

而任何用户进程来说，他们的内核空间都是100%相同的，所以内核会借用上一个被调用的用户进程的mm\_struct中的页表来访问内核地址，这个mm\_struct就记录在active\_mm。

简而言之就是，对于内核线程，tsk->mm == NULL表示自己内核线程的身份，而tsk->active\_mm是借用上一个用户进程的mm\_struct，用mm\_struct的页表来访问内核空间。

对于用户进程，tsk->mm == tsk->active\_mm。

**补充说明3**：除了 mm\_users 和 mm\_count 之外，还有 mmap 和 mm\_rb 需要说明以下：

其实 mmap 和 mm\_rb 都是保存此 进程地址空间中所有的内存区域(VMA)的，前者是以链表形式存放，后者以红黑树形式存放。

用2种数据结构组织同一种数据是为了便于对VMA进行高效的操作。

#### **1.2 mm\_struct操作**

1. 分配进程地址空间

参考 kernel/fork.c 中的宏 allocate\_mm

#define allocate\_mm() (kmem\_cache\_alloc(mm\_cachep, GFP\_KERNEL))#define free\_mm(mm) (kmem\_cache\_free(mm\_cachep, (mm)))

其实分配进程地址空间时，都是从slab高速缓存中分配的，可以通过 /proc/slabinfo 查看 mm\_struct 的高速缓存

# cat /proc/slabinfo | grep mm\_struct

mm\_struct 35 45 1408 5 2 : tunables 24 12 8 : slabdata 9 9 0

2. 撤销进程地址空间

参考 kernel/exit.c 中的 exit\_mm() 函数

该函数会调用 mmput() 函数减少 mm\_users 的值，

当 mm\_users=0 时，调用 mmdropo() 函数， 减少 mm\_count 的值，

如果 mm\_count=0，那么调用 free\_mm 宏，将 mm\_struct 还给 slab高速缓存

3. 查看进程占用的内存：

cat /proc/<PID>/maps

或者

pmap PID

### **2. 虚拟内存区域(VMA)**

内存区域在linux中也被称为虚拟内存区域(VMA)，它其实就是进程地址空间上一段连续的内存范围。

#### **2.1 VMA介绍**

VMA的定义也在 <linux/mm\_types.h> 中

struct vm\_area\_struct {

struct mm\_struct \* vm\_mm; /\* 相关的 mm\_struct 结构体 \*/

unsigned long vm\_start; /\* 内存区域首地址 \*/

unsigned long vm\_end; /\* 内存区域尾地址 \*/

/\* linked list of VM areas per task, sorted by address \*/

struct vm\_area\_struct \*vm\_next, \*vm\_prev; /\* VMA链表 \*/

pgprot\_t vm\_page\_prot; /\* 访问控制权限 \*/

unsigned long vm\_flags; /\* 标志 \*/

struct rb\_node vm\_rb; /\* 树上的VMA节点 \*/

/\*

\* For areas with an address space and backing store,

\* linkage into the address\_space->i\_mmap prio tree, or

\* linkage to the list of like vmas hanging off its node, or

\* linkage of vma in the address\_space->i\_mmap\_nonlinear list.

\*/

union {

struct {

struct list\_head list;

void \*parent; /\* aligns with prio\_tree\_node parent \*/

struct vm\_area\_struct \*head;

} vm\_set;

struct raw\_prio\_tree\_node prio\_tree\_node;

} shared;

/\*

\* A file's MAP\_PRIVATE vma can be in both i\_mmap tree and anon\_vma

\* list, after a COW of one of the file pages. A MAP\_SHARED vma

\* can only be in the i\_mmap tree. An anonymous MAP\_PRIVATE, stack

\* or brk vma (with NULL file) can only be in an anon\_vma list.

\*/

struct list\_head anon\_vma\_node; /\* Serialized by anon\_vma->lock \*/

struct anon\_vma \*anon\_vma; /\* Serialized by page\_table\_lock \*/

/\* Function pointers to deal with this struct. \*/

const struct vm\_operations\_struct \*vm\_ops;

/\* Information about our backing store: \*/

unsigned long vm\_pgoff; /\* Offset (within vm\_file) in PAGE\_SIZE

units, \*not\* PAGE\_CACHE\_SIZE \*/

struct file \* vm\_file; /\* File we map to (can be NULL). \*/

void \* vm\_private\_data; /\* was vm\_pte (shared mem) \*/

unsigned long vm\_truncate\_count;/\* truncate\_count or restart\_addr \*/

#ifndef CONFIG\_MMU

struct vm\_region \*vm\_region; /\* NOMMU mapping region \*/#endif

#ifdef CONFIG\_NUMA

struct mempolicy \*vm\_policy; /\* NUMA policy for the VMA \*/#endif

};

这个结构体各个字段的英文注释都比较详细，就不一一翻译了。

上述属性中的 vm\_flags 标识了此VM 对 VMA和页面的影响：

vm\_flags 的宏定义参见 <linux/mm.h>

|  |  |
| --- | --- |
| **标志** | **对VMA及其页面的影响** |
| VM\_READ | 页面可读取 |
| VM\_WRITE | 页面可写 |
| VM\_EXEC | 页面可执行 |
| VM\_SHARED | 页面可共享 |
| VM\_MAYREAD | VM\_READ 标志可被设置 |
| VM\_MAYWRITER | VM\_WRITE 标志可被设置 |
| VM\_MAYEXEC | VM\_EXEC 标志可被设置 |
| VM\_MAYSHARE | VM\_SHARE 标志可被设置 |
| VM\_GROWSDOWN | 区域可向下增长 |
| VM\_GROWSUP | 区域可向上增长 |
| VM\_SHM | 区域可用作共享内存 |
| VM\_DENYWRITE | 区域映射一个不可写文件 |
| VM\_EXECUTABLE | 区域映射一个可执行文件 |
| VM\_LOCKED | 区域中的页面被锁定 |
| VM\_IO | 区域映射设备I/O空间 |
| VM\_SEQ\_READ | 页面可能会被连续访问 |
| VM\_RAND\_READ | 页面可能会被随机访问 |
| VM\_DONTCOPY | 区域不能在 fork() 时被拷贝 |
| VM\_DONTEXPAND | 区域不能通过 mremap() 增加 |
| VM\_RESERVED | 区域不能被换出 |
| VM\_ACCOUNT | 该区域时一个记账 VM 对象 |
| VM\_HUGETLB | 区域使用了 hugetlb 页面 |
| VM\_NONLINEAR | 该区域是非线性映射的 |

#### **2.2 VMA操作**

vm\_area\_struct 结构体定义中有个 vm\_ops 属性，其中定义了内核操作 VMA 的方法

/\*

\* These are the virtual MM functions - opening of an area, closing and

\* unmapping it (needed to keep files on disk up-to-date etc), pointer

\* to the functions called when a no-page or a wp-page exception occurs.

\*/struct vm\_operations\_struct {

void (\*open)(struct vm\_area\_struct \* area); /\* 指定内存区域加入到一个地址空间时，该函数被调用 \*/

void (\*close)(struct vm\_area\_struct \* area); /\* 指定内存区域从一个地址空间删除时，该函数被调用 \*/

int (\*fault)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf); /\* 当没有出现在物理页面中的内存被访问时，该函数被调用 \*/

/\* 当一个之前只读的页面变为可写时，该函数被调用，

\* 如果此函数出错，将导致一个 SIGBUS 信号 \*/

int (\*page\_mkwrite)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

/\* 当 get\_user\_pages() 调用失败时, 该函数被 access\_process\_vm() 函数调用 \*/

int (\*access)(struct vm\_area\_struct \*vma, unsigned long addr,

void \*buf, int len, int write);

#ifdef CONFIG\_NUMA

/\*

\* set\_policy() op must add a reference to any non-NULL @new mempolicy

\* to hold the policy upon return. Caller should pass NULL @new to

\* remove a policy and fall back to surrounding context--i.e. do not

\* install a MPOL\_DEFAULT policy, nor the task or system default

\* mempolicy.

\*/

int (\*set\_policy)(struct vm\_area\_struct \*vma, struct mempolicy \*new);

/\*

\* get\_policy() op must add reference [mpol\_get()] to any policy at

\* (vma,addr) marked as MPOL\_SHARED. The shared policy infrastructure

\* in mm/mempolicy.c will do this automatically.

\* get\_policy() must NOT add a ref if the policy at (vma,addr) is not

\* marked as MPOL\_SHARED. vma policies are protected by the mmap\_sem.

\* If no [shared/vma] mempolicy exists at the addr, get\_policy() op

\* must return NULL--i.e., do not "fallback" to task or system default

\* policy.

\*/

struct mempolicy \*(\*get\_policy)(struct vm\_area\_struct \*vma,

unsigned long addr);

int (\*migrate)(struct vm\_area\_struct \*vma, const nodemask\_t \*from,

const nodemask\_t \*to, unsigned long flags);#endif

};

除了以上的操作之外，还有一些辅助函数来方便内核操作内存区域。

这些辅助函数都可以在 <linux/mm.h> 中找到

1. 查找地址空间

/\* Look up the first VMA which satisfies addr < vm\_end, NULL if none. \*/

extern struct vm\_area\_struct \* find\_vma(struct mm\_struct \* mm, unsigned long addr);extern struct vm\_area\_struct \* find\_vma\_prev(struct mm\_struct \* mm, unsigned long addr,

struct vm\_area\_struct \*\*pprev);

/\* Look up the first VMA which intersects the interval start\_addr..end\_addr-1,

NULL if none. Assume start\_addr < end\_addr. \*/

static inline struct vm\_area\_struct \* find\_vma\_intersection(struct mm\_struct \* mm, unsigned long start\_addr, unsigned long end\_addr)

{

struct vm\_area\_struct \* vma = find\_vma(mm,start\_addr);

if (vma && end\_addr <= vma->vm\_start)

vma = NULL;

return vma;

}

2. 创建地址区间

static inline unsigned long do\_mmap(struct file \*file, unsigned long addr,

unsigned long len, unsigned long prot,

unsigned long flag, unsigned long offset)

{

unsigned long ret = -EINVAL;

if ((offset + PAGE\_ALIGN(len)) < offset)

goto out;

if (!(offset & ~PAGE\_MASK))

ret = do\_mmap\_pgoff(file, addr, len, prot, flag, offset >> PAGE\_SHIFT);out:

return ret;

}

3. 删除地址区间

extern int do\_munmap(struct mm\_struct \*, unsigned long, size\_t);

### **3. 地址空间和页表**

地址空间中的地址都是虚拟内存中的地址，而CPU需要操作的是物理内存，所以需要一个将虚拟地址映射到物理地址的机制。

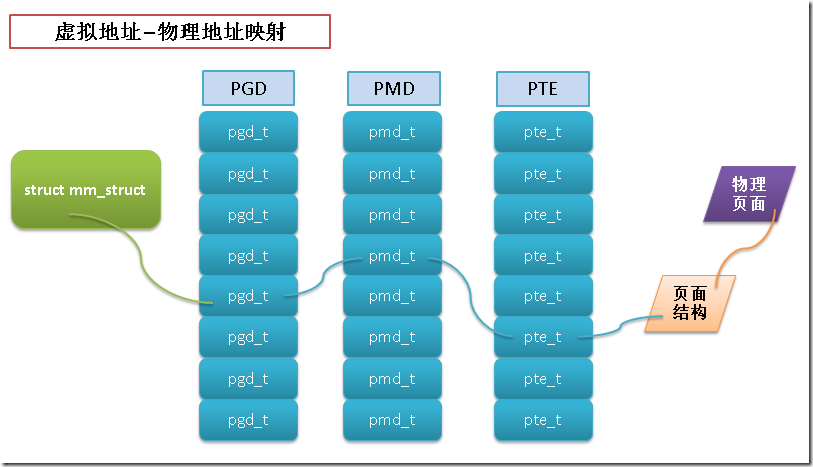
这个机制就是页表，linux中使用3级页面来完成虚拟地址到物理地址的转换。

1. PGD - 全局页目录，包含一个 pgd\_t 类型数组，多数体系结构中 pgd\_t 类型就是一个无符号长整型

2. PMD - 中间页目录，它是个 pmd\_t 类型数组

3. PTE - 简称页表，包含一个 pte\_t 类型的页表项，该页表项指向物理页面

虚拟地址 - 页表 - 物理地址的关系如下图：

[](https://images0.cnblogs.com/blog/83005/201310/04162809-8efa30e1f1644caf94b1650ad9b9a5b0.png)