mm_struct 结构体定义如下：

```
struct mm_struct {
    struct vm_area_struct * mmap;     /* list of VMAs */      //虚拟地址空间结构体，双向链表包含红黑树节点访问到不能访问的区域。
    struct rb_root mm_rb;                                       //红黑树的根节点
    struct vm_area_struct * mmap_cache; /* last find_vma result */   //mmap 的高速缓冲器，指的是 mmap 最后指向的一个虚拟地址区间
#ifdef CONFIG_MMU
    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
#endif
    unsigned long mmap_base;       /* base of mmap area */                                  //mmap 区域的基地址
    unsigned long mmap_legacy_base;                /* base of mmap area in bottom-up allocations */   //自底向上的配置
    unsigned long task_size;       /* size of task vm space */                              //进程的虚拟地址空间大小
    unsigned long cached_hole_size;     /* if non-zero, the largest hole below free_area_cache */ //缓冲器的最大的大小
    unsigned long free_area_cache;       /* first hole of size cached_hole_size or larger */          //不受约束的空间大小
    unsigned long highest_vm_end;     /* highest vma end address */                          //虚拟地址空间最大结尾地址
    pgd_t * pgd;                                                                           //页表的全局目录
    atomic_t mm_users;           /* How many users with user space? */                     //有多少用户
    atomic_t mm_count;           /* How many references to "struct mm_struct" (users count as 1) */ //有多少用户引用 mm_struct
    atomic_long_t nr_ptes;    /* Page table pages */                                       //页表
    int map_count;               /* number of VMAs */                                      //虚拟地址空间的个数
    spinlock_t page_table_lock;      /* Protects page tables and some counters */          //保护页表和用户
    struct rw_semaphore mmap_sem;                                                          //读写信号
    struct list_head mmlist;       /* List of maybe swapped mm's.    These are globally strung
                    * together off init_mm.mmlist, and are protected
                    * by mmlist_lock
                    */
    unsigned long hiwater_rss;    /* High-watermark of RSS usage */                        //标志
    unsigned long hiwater_vm; /* High-water virtual memory usage */
    unsigned long total_vm;      /* Total pages mapped */
    unsigned long locked_vm;    /* Pages that have PG_mlocked set */
    unsigned long pinned_vm;    /* Refcount permanently increased */
    unsigned long shared_vm;    /* Shared pages (files) */
    unsigned long exec_vm;       /* VM_EXEC & ~VM_WRITE */
    unsigned long stack_vm;      /* VM_GROWSUP/DOWN */
    unsigned long def_flags;
    unsigned long start_code, end_code, start_data, end_data;                              //开始代码段，结束代码。开始数据，结束数据
    unsigned long start_brk, brk, start_stack;                                             //堆的开始和结束。
    unsigned long arg_start, arg_end, env_start, env_end;                                  //参数的起始和结束，环境变量的起始和终点
    unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */
```

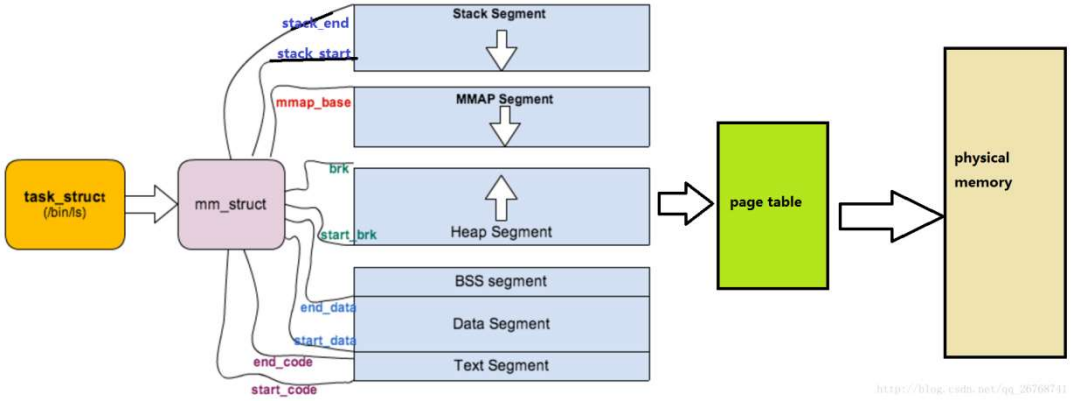
```
        ...
};
```

在 task_struct 中，有两个成员：

```
struct task_struct {
    ...
    struct mm_struct        *mm;
    struct mm_struct        *active_mm;
    ...
}
```

对于用户态进程，其任务描述符（task_struct）的 mm 和 active_mm 相同，都是指向其进程地址空间。对于内核线程而言，其 task_struct 的 mm 成员为 NULL（内核线程没有进程地址空间），但是，内核线程被调度执行的时候，总是需要一个进程地址空间，而 active_mm 就是指向它借用的那个进程地址空间。

mm_struct 是进程的内存描述符，根据上面的结构体定义，大致包含以下内容：



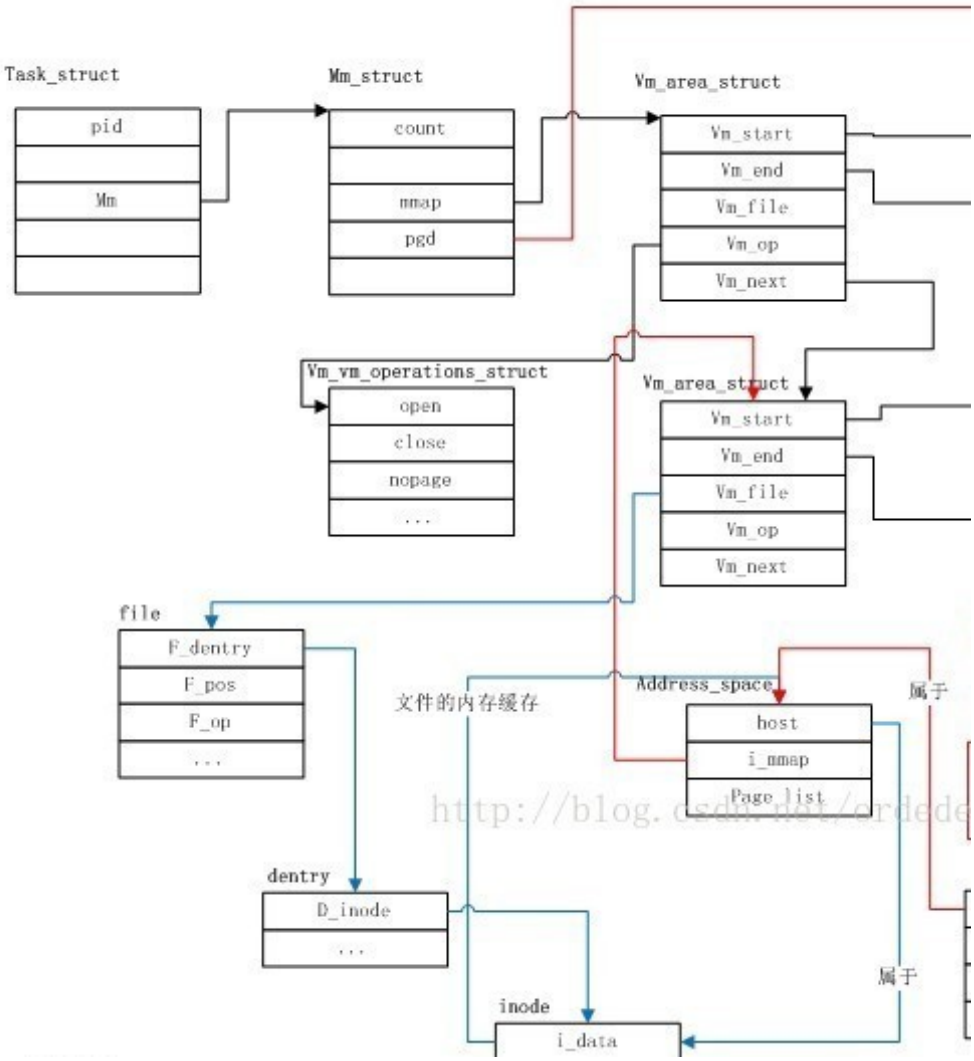
即一个进程对应的栈，mmap 区域(vm_area_struct)，堆，bss,data,text 这些的范围。

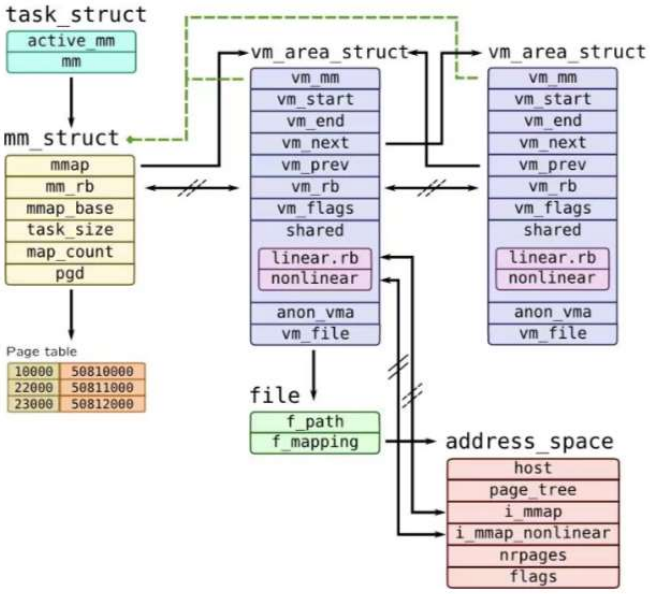
另外其成员 struct vm_area_struct *mmap;，指向的是一段一段虚拟地址空间的链表：

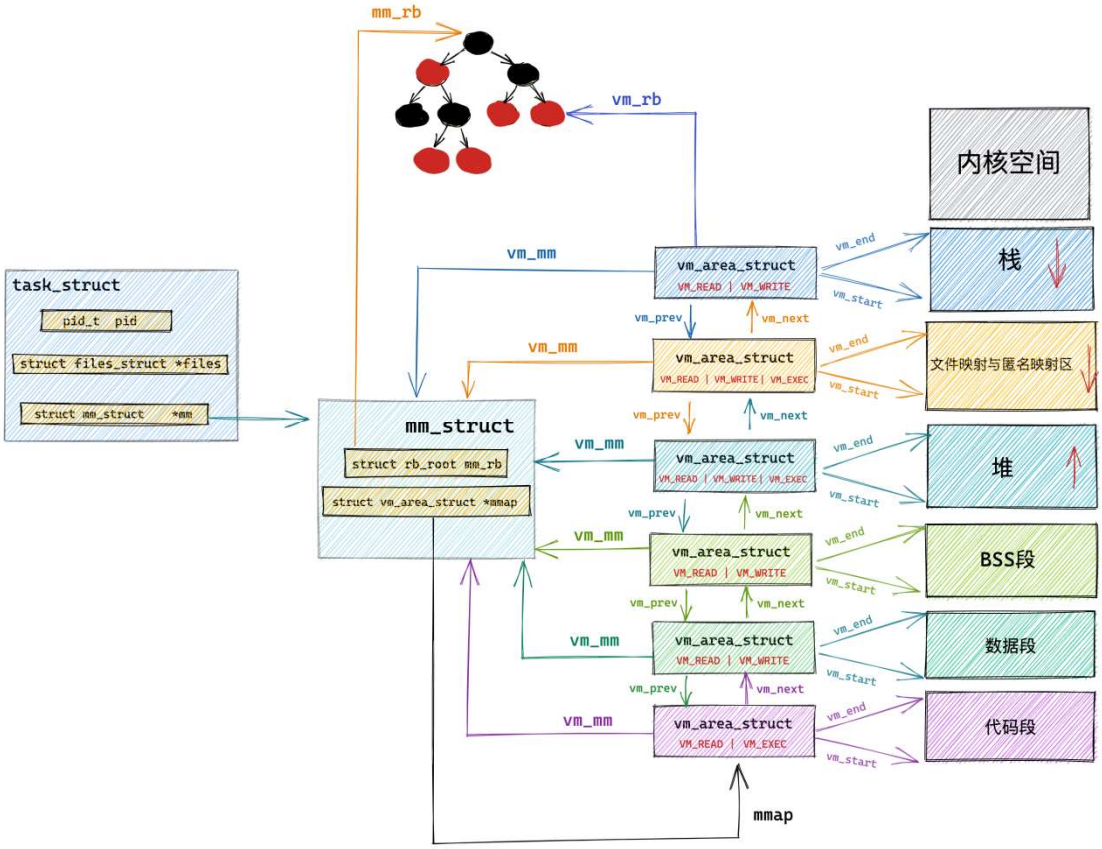
```
/*
 * This struct describes a virtual memory area. There is one of these
 * per VM-area/task. A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */
struct vm_area_struct {
    /* The first cache line has the info for VMA tree walking. */

    unsigned long vm_start;        /* Our start address within vm_mm. */
    unsigned long vm_end;        /* The first byte after our end address
                                    within vm_mm. */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;

    struct rb_node vm_rb;
```

```
	/*
	 * Largest free memory gap in bytes to the left of this VMA.
	 * Either between this VMA and vma->vm_prev, or between one of the
	 * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
	 * get_unmapped_area find a free area of the right size.
	 */
	unsigned long rb_subtree_gap;

	/* Second cache line starts here. */

	struct mm_struct *vm_mm;	/* The address space we belong to. */

	/*
	 * Access permissions of this VMA.
	 * See vmf_insert_mixed_prot() for discussion.
	 */
	pgprot_t vm_page_prot;
	unsigned long vm_flags;		/* Flags, see mm.h. */

	/*
	 * For areas with an address space and backing store,
	 * linkage into the address_space->i_mmap interval tree.
	 */
	struct {
		struct rb_node rb;
		unsigned long rb_subtree_last;
	} 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_chain; /* Serialized by mmap_lock &
					  * page_table_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 */
        struct file * vm_file;          /* File we map to (can be NULL). */
        void * vm_private_data;         /* was vm_pte (shared mem) */

#ifdef CONFIG_SWAP
        atomic_long_t swap_readahead_info;
#endif
#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
        struct vm_userfaultfd_ctx vm_userfaultfd_ctx;
} __randomize_layout;
```

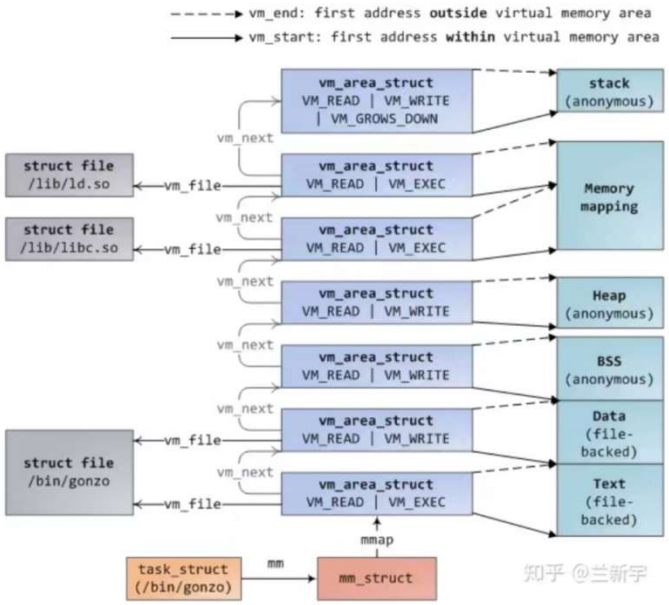






可以看到，每一段（栈，堆等），都由 vm_area_struct 描述。

这其中例如 mmap 的时候，新增的 vm_area_struct，也会被添加到这个链表中，如下图的 memory mapping：



在 linux 系统上通过 `cat /proc/pid/maps` 命令查看虚拟内存结构：

```
c.h
learn_process  sum-scalability.c  thread.h
learn_socket   test.c             vma_test.c
pp
mjf@DESKTOP-SPQ45P7:~/njuos$ ./a.out
PID = 164
```

```
mjf@DESKTOP-SPQ45P7:~/njuos$ cat /proc/164/maps
5612126c4000-5612126c5000 r--p 00000000 08:10 3748                       /home/mjf/njuos/a.out
5612126c5000-5612126c6000 r-xp 00001000 08:10 3748                       /home/mjf/njuos/a.out
5612126c6000-5612126c7000 r--p 00002000 08:10 3748                       /home/mjf/njuos/a.out
5612126c7000-5612126c8000 r--p 00002000 08:10 3748                       /home/mjf/njuos/a.out
5612126c8000-5612126c9000 rw-p 00003000 08:10 3748                       /home/mjf/njuos/a.out
561213271000-561213292000 rw-p 00000000 00:00 0                          [heap]
7f9a1e761000-7f9a1e783000 r--p 00000000 08:10 24809                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f9a1e783000-7f9a1e8fb000 r-xp 00022000 08:10 24809                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f9a1e8fb000-7f9a1e949000 r--p 0019a000 08:10 24809                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f9a1e949000-7f9a1e94d000 r--p 001e7000 08:10 24809                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f9a1e94d000-7f9a1e94f000 rw-p 001eb000 08:10 24809                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f9a1e94f000-7f9a1e955000 rw-p 00000000 00:00 0
7f9a1e961000-7f9a1e962000 r--p 00000000 08:10 24785                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f9a1e962000-7f9a1e985000 r-xp 00001000 08:10 24785                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f9a1e985000-7f9a1e98d000 r--p 00024000 08:10 24785                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f9a1e98e000-7f9a1e98f000 r--p 0002c000 08:10 24785                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f9a1e98f000-7f9a1e990000 rw-p 0002d000 08:10 24785                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f9a1e990000-7f9a1e991000 rw-p 00000000 00:00 0
7fff9a5ed000-7fff9a60e000 rw-p 00000000 00:00 0                          [stack]
7fff9a72b000-7fff9a72f000 r--p 00000000 00:00 0                          [vvar]
7fff9a72f000-7fff9a730000 r-xp 00000000 00:00 0                          [vdso]
mjf@DESKTOP-SPQ45P7:~/njuos$
```

每一列的含义分别为

内核每进程的vm_area_struct项	/proc/pid/maps中的项	含义
vm_start	"-"前一列，如00377000	此段虚拟地址空间起始地址
vm_end	"-"后一列，如00390000	此段虚拟地址空间结束地址
vm_flags	第三列，如r-xp	此段虚拟地址空间的属性。每种属性用一个字段表示，r表示可读，w表示可写，x表示可执行，p和s共用一个字段，互斥关系，p表示私有段，s表示共享段，如果没有相应权限，则用-代替
vm_pgoff	第四列，如00000000	对有名映射，表示此段虚拟内存起始地址在文件中以页为单位的偏移。对匿名映射，它等于0或者vm_start/PAGE_SIZE
vm_file->f_dentry->d_inode->i_sb->s_dev	第五列，如fd:00	映射文件所属设备号。对匿名映射来说，因为没有文件在磁盘上，所以没有设备号，始终为00:00。对有名映射来说，是映射的文件所在设备的设备号
vm_file->f_dentry->d_inode->i_ino	第六列，如9176473	映射文件所属节点号。对匿名映射来说，因为没有文件在磁盘上，所以没有节点号，始终为00:00。对有名映射来说，是映射的文件的节点号
	第七列，如/lib/ld-2.5.so	对有名来说，是映射的文件名。对匿名映射来说，是此段虚拟内存在进程中的角色。[stack]表示在进程中作为栈使用，[heap]表示堆。其余情况则无显示