

SIXUENET

To Learn Without Thinking Is To Be Useless, To Think Without Learning Is To Lose
(<http://www.mysixue.com/>)

MEMORY MANAGEMENT (3): VIRTUAL MEMORY MANAGEMENT

📅 April 20, 2019 ([Http://Www.Mysixue.Com/?P=114](http://Www.Mysixue.Com/?P=114)) 👤 JL
([Http://Www.Mysixue.Com/?Author=1](http://Www.Mysixue.Com/?Author=1)) 💬

3 Virtual memory management

Virtual memory management mainly involves three aspects : virtual space allocation ; physical page frame allocation ; establishing page tables and linking the two .

These three do not have to operate at the same time , for example :

- can only assign a virtual address , temporarily allocate physical page frame and build page tables . Only when the CPU to access a virtual address , found that it is not associated with the physical page frame time , will be treated by the missing page (Page Fault Handling allocate physical page) mechanism Frame and build the page table .

This situation is common in user space , in the allocation of the virtual address , we usually put this virtual address space associated with a back-up memory by storing the mapping mechanism , backing store is mainly responsible for reading the data disk file . About the memory map The details of the mechanism will be specifically introduced later .

- You can also allocate virtual address space (but not absolute) , and allocate physical page frame pages to establish page tables . This situation is common in kernel space , such as vmalloc.

3.1 Page table

The page table is used to establish the association between the virtual address space of the user process and the physical memory (memory, page frame) of the system.

Each process is its own independent page table, which provides a consistent virtual address space for each process. The address space seen by the application is a contiguous memory area. This table also maps virtual memory pages to physical memory, thus supporting the implementation of shared memory (memory shared by several processes at the same time) . It can also swap out pages to block devices without adding additional physical memory to increase the effectiveness Available memory space.

Kernel memory management always assumes the use of four-level page tables, regardless of whether the underlying processor is the case. Page table management is divided into two parts, the first part depends on the architecture, and the second part is independent of the architecture. Interestingly, all data structures and almost all functions that manipulate data structures are defined in architecture-specific files.

The data structures and functions described in the next few sections are usually based on the interfaces provided in the architecture-related files. The definition can be found in the header files `arch/ " arch " /include/asm/page.h` and `arch/ " arch " /include/asm/pgtable.h` .

3.1.1 Main data structure

In the C language, the `void *` data type is used to define a pointer that may point to any byte location in memory. The number of bits required for this type varies with different architectures. All common processors (including all processors supported by Linux) use 32 -bit or 64 -bit.

The kernel source code assumes that `void *` and unsigned long types require the same number of bits, so they can be forced to convert without loss of information. The form of this assumption is expressed as `sizeof(void *) == sizeof(unsigned long)` , which is correct on all architectures supported by Linux .

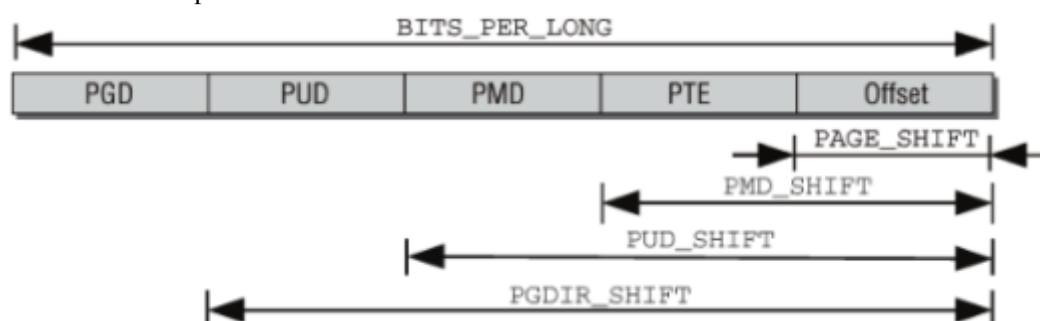
Memory management prefers to use unsigned long variables instead of void pointers, because the former is easier to handle and manipulate. Technically, they are all effective.

Decomposition of virtual addresses

According to the needs of the four-level page table structure, the virtual memory address is divided into 5 parts (4 table entries are used to select the page, and 1 index represents the position in the page) .

Different architectures not only have different address word lengths, but also different ways of address word splitting. Therefore, the kernel defines macros to decompose the address into individual components.

The following figure illustrates how to use bit shift to define the position of each component of the address word. `BITS_PER_LONG` defines the number of bits used for unsigned long variables, so it is also suitable for general pointers to virtual address spaces.



➤ bits at the end of each pointer are used to specify the position within the selected page frame.

The specific number of bits is specified by `PAGE_SHIFT` .

➤ PTE stands for page table

- PMD stands for middle page directory
- PUD stands for upper page directory
- PGD is the global page directory

The number of pointers that can be stored in the page directory / page table at all levels can also be determined by macro definitions. PTRS_PER_PGD specifies the number of entries in the global page directory, PTRS_PER_PMD corresponds to the middle page directory, PTRS_PER_PUD corresponds to the number of entries in the upper page directory, and PTRS_PER_PTE is the number of entries in the page table.

The MMU of some CPUs only supports 2-level page tables, and the architecture of the two-level page table defines PTRS_PER_PMD and PTRS_PER_PUD as 1. This makes the rest of the kernel feel that the architecture also provides a four-level page translation structure, although there are actually only two-level page tables: the middle-level page directory and the upper-level page directory are actually eliminated because there is only one item. Since only a few systems use four-level page tables, the kernel uses the header file include/asm-generic/pgtable-nopud.h to provide all the declarations needed to simulate the upper-level page directory. The header file include/asm-generic/pgtable-nopmd.h is used to simulate the middle-level page table on a system with only two-level address translation.

The addressable address area of an address word with a length of 2^n bits is 2^n bytes. The kernel defines additional macro variables to save the calculated value to avoid repeated calculations. The related macro definitions are as follows:

```
#define PAGE_SIZE (1UL << PAGE_SHIFT)
#define PUD_SIZE      (1UL << PUD_SHIFT)
#define PMD_SIZE      (1UL << PMD_SHIFT)
#define PGDIR_SIZE    (1UL << PGDIR_SHIFT)
```

The value 2^n is easily calculated by shifting n bits from position 0 to the left in binary. The kernel uses this technique in many places.

ARM 32-bit architecture virtual address resolution

I introduced some background knowledge of virtual address differentiation, let's take a look at how virtual addresses are divided in the ARM 32-bit architecture.

The division of virtual addresses is defined in the header file: arch/arm/include/asm/pgtable.h

```
...
#include <asm-generic/pgtable-nopud.h>
...
#ifdef CONFIG_ARM_LPAE
#include <asm/pgtable-3level.h>
#else
#include <asm/pgtable-2level.h>
#endif
```

- the include the pgtable-nopud.h, described no PUD. In pgtable-nopud.h in, will make the following two definitions:

```
#define PUD_SHIFT PGDIR_SHIFT
```

```
#define PTRS_PER_PUD 1
```

Thus , to achieve the elimination of PUD object layer , and is transparent to the code for upper layer , the upper layer can still specify the code . 4 -level page table structures.

- If CONFIG_ARM_LPAE is not defined , the 2 -level page table structure will be used . I have seen several ARM development board cores , but this switch is not defined , so we will focus on the 2 -level page table .

Header file : arch/arm/include/asm/pgtable-2level.h

- First look at the definition of PGD :

```
#define PGDIR_SHIFT 21

#define PTRS_PER_PGD 2048

#define PGDIR_SIZE (1UL << PGDIR_SHIFT)
#define PGDIR_MASK (~(PGDIR_SIZE-1))
```

The SHIFT of the global page directory is 21, so the global directory has a total of $2^{11} = 2048$ entries . That is, if we use an array to store the global page directory , then the array has a total of 2048 elements , if each element occupies 4 A byte , the global directory will occupy a total of 8KB of physical memory , and the size is still acceptable !

- take a look at the definition of PMD :

```
#define PMD_SHIFT 21

#define PTRS_PER_PMD 1

#define PMD_SIZE (1UL << PMD_SHIFT)
#define PMD_MASK (~(PMD_SIZE-1))
```

PMD_SHIFT and PGDIR_SHIFT one , PMD entries only 1 Ge , so PMD is eliminated

- eliminates PUD and PMD, on the left PGD and PTE the , so called 2 -level page table , let's look at PTE

```
#define PTRS_PER_PTE 512
```

PTRS_PER_PTE means that each page table has 512 entries . If an array is used to represent the page table , the number of array elements is 512. Each element of the array stores the address of physical memory , and one element on a 32 -bit system will occupy 4 bytes .

How did this value come from ? $\text{PTRS_PER_PTE} = 2^{((\text{PMD_SHIFT}+1) - \text{PAGE_SHIFT})}$, PAGE_SHIFT is defined as 12 (described later) , so $\text{PTRS_PER_PTE} = 2^{10} = 512$.

Finally , let us look at PAGE_SHIFT defined , PAGE_SHIFT represents the system in a page size , regardless of the physical address space or virtual address space , it will be divided according to the size of one page .

Header file : arch/arm/include/asm/page.h

```
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT 12
#define PAGE_SIZE (_AC(1,UL) << PAGE_SHIFT)
```

```
#define PAGE_MASK (~((1 << PAGE_SHIFT) - 1))
```

The size of each page is $2^{12} = 4\text{KB}$.

In summary , in the ARM 32 -bit system , the address space is divided into 4KB pages , and a 2 -level page table structure (PGD+PTE) is used to manage all virtual address space pages . The global page directory PGD has a total of 2048 items , each A page table PTE has 512 entries .

Page table data structure and related functions

The above definition has established the number of each page directory / page table , but has not defined its structure. The so-called structure refers to the data structure describing each element in the page directory / page table . For 32 -bit system , each element with a u32 integer representation ; for 64 -bit systems , each element with a u64 integer representation .

The kernel provides 4 data structures (defined in page.h) to represent the structure of the page directory / page table .

- pgd_t is used for global page directory entries.
- pud_t is used for the upper page directory item.
- pmd_t is used for the middle page directory item.
- pte_t is used for direct page table entries.

For the ARM system architecture , the above several data structures are defined as follows :

```
//arch/arm/include/asm/pgtable-2level-types.h

typedef u32 pteval_t;
typedef u32 pmdval_t;

/*
 * These are used to make use of C type-checking..
 */
typedef struct {pteval_t pte;} pte_t;
typedef struct {pmdval_t pmd;} pmd_t;
typedef struct {pmdval_t pgd[2];} pgd_t;
typedef struct {pteval_t pgprot;} pgprot_t;
```

Use struct instead of basic types to ensure that the contents of table items can only be processed by related auxiliary functions, and never directly accessed. Table items can also be composed of several basic types of variables. In this case, the kernel must use struct . (For example, when the IA-32 processor uses PAE mode, define pte_t as typedef struct {unsigned long pte_low, pte_high;} . 32 bits are obviously not enough to address all the memory, because this mode can manage more than 4 GiB In other words, the amount of memory available can be greater than the address space of the processor. But since the pointer is still only 32 bits wide, an appropriate subset of the expanded memory space must be selected for user space applications so that each The process still only sees the 4 GiB address space.)

The virtual address is divided into several parts and used as the index of each page table. This is a familiar solution. Depending on the word length of the architecture used, the length of each individual part is less than 32 or 64 bits. As can be seen from the given kernel source code snippets, the kernel (and processor) uses 32 or 64 -

bit types to represent table entries (regardless of the number of levels in the page table) . This means that not all bits of the table entry store useful data, that is, the base address of the next-level table. The extra bits are used to store additional information.

The following table shows the standard functions used to analyze page directories / page tables . According to different architectures , some functions may be implemented as macros while others are implemented as inline functions . I will not distinguish between the two in the following .

Functions	Comment
pgd_val pud_val pmd_val pte_val pgprot_val	Convert variables of types such as pte_t to unsigned long integers
__pgd __pud __pmd __pte __pgprot	pgd_val inverse like function: The unsigned long integer conversion to pgd_t other types of variable
pgd_index (virtual_addr) pud_index (virtual_addr) pmd_index (virtual_addr) pte_index (virtual_addr)	Given a virtual address , the index of the related table entry is proposed from the virtual address . For example : #define pgd_index(addr) ((addr) >> PGDIR_SHIFT) Assuming that PGD[] is a global page directory array , then PGD[pgd_index(addr)] can get the starting address of the next level page table .
pgd_present pud_present pmd_present pte_present	Check whether the _PRESENT bit of the corresponding item is set. If the corresponding page directory or page table is in memory, it will be set
pgd_none pud_none pmd_none pte_none	Invert the logic of the value of the xxx_present function. If it returns true , the corresponding page directory or page table is not in memory
pgd_clear pud_clear pmd_clear pte_clear	Delete the passed entry . Usually set it to zero
pgd_bad pud_bad pmd_bad	Check whether the entries in the middle page table, upper page table, and global page table are invalid. If the function receives input parameters from outside, it cannot be assumed that the parameters are valid. To ensure safety, you can call these functions to check
pud_page pmd_page pte_page	pud_page returns a physical address that points to the location of the pud page directory pmd_page also returns a physical address , pointing to the location of the pte page table pte_page will return the address of the physical page frame

PAGE_ALIGN	It is another standard macro (usually in <code>page.h</code>) that must be defined for every architecture. It takes an address as a parameter and "rounds" the address to the beginning of the next page. If the page size is 4096, the macro always returns its multiple. $PAGE_ALIGN(6000)=8192 = 2 \times 4096$, $PAGE_ALIGN(0x84590860)=0x84591000 = 542097 \times 4096$. Since the CPU cache is a cache physical page by page, to a better use of processor cache resources, align the address to the page boundary is very important.
------------	--

pte flags

The entry (`pte`) in the last-level page table not only contains a pointer to the memory location of the page, but also contains additional information related to the page in the above-mentioned extra bits. Although these data are CPU-specific, they at least provide some information about page access control. The following bits can be found in most CPUs supported by the Linux kernel.

- `_PAGE_PRESENT` specifies whether the virtual memory page exists in the memory. Pages may not always be in memory, and pages may be swapped out to the swap area.
If the page is not in memory, the structure of the page table entry is usually different, because there is no need to describe the location of the page in memory. Instead, information is needed to identify and find the page that was swapped out.
- Each time the CPU accesses a page, it will automatically set `_PAGE_ACCESSED`. The kernel will periodically check this bit to confirm the active level of page usage (pages that are not frequently used are more suitable for swapping out). This bit will be set after read or write access.
- `_PAGE_DIRTY` indicates whether the page is "dirty", that is, whether the content of the page has been modified.
- `_PAGE_FILE` values and `_PAGE_DIRTY` same, but for a different context, i.e., when the page is not in memory. Obviously, a page that does not exist cannot be dirty, so the bit can be reinterpreted. If it is not set, the item points to a page to be swapped out. If the item belongs to non-linear file mapping, you need to set `_PAGE_FILE`, and it will be discussed in the section "Non-linear mapping".
- If the `_PAGE_USER`, the user is allowed to access the page space code. Otherwise, only the kernel can access it (or when the CPU is in the system state).
- `_PAGE_READ`, `_PAGE_WRITE` and `_PAGE_EXECUTE` specify whether the ordinary user process is allowed to read, write, and execute the machine code in the page.

The pages in kernel memory must be protected from being written by user processes.

But even pages belonging to user processes cannot be guaranteed to be writable. This may be intentional or unintentional. For example, it may contain executable code that cannot be modified.

For architectures with less granular access permissions, if there are no further criteria to distinguish read and write access permissions, the `_PAGE_RW` constant will be defined to allow or prohibit read and write access at the same time.

- IA-32 and AMD64 provide `_PAGE_BIT_NX`, for the page is marked as not executable (the IA-32 on the system, only the addressable enable 64 GiB memory page address extension (Page Extension address, the PAE) when the function, This protection bit can be used. For example, it can prevent the execution of the code on the stack page. Otherwise, malicious code may execute code on the stack by means of buffer overflow, resulting in a security hole in the program. The NX bit cannot prevent buffer overflow, but it can suppress its effect, because the process will refuse to execute malicious code. Of course, if the architecture itself provides good access authorization settings for memory pages, the same effect can be achieved. This is the case with some processors (unfortunately, these processors are not very common).

Each architecture must provide two things so that the memory management subsystem can modify the extra bits in the `pte_t` item, that is, `__pgprot` The data type, and the `pte_modify` function to modify these bits. The above flags can be used to select appropriate bits.

The kernel also defines various functions for querying and setting the state of the memory page and the architecture. Some processors may lack hardware support for some given features, so not all processors have all these functions defined.

- `pte_present` checks whether the page pointed to by the page table entry exists in the memory. For example, this function can be used to detect whether a page has been swapped out.
- `pte_dirty` checks whether the page related to the page table entry is dirty, that is, whether its content has been modified since the last kernel check. It should be noted that this function can be called only when `pte_present` confirms that the page is available.
- `pte_write` checks whether the kernel can write to the page .
- `pte_file` is used for non-linear mapping. It provides a different view of the file content by operating the page table (see "Non-linear Mapping" for details) . This function checks whether the page table entry belongs to such a mapping.
Only when `pte_present` returns false , can `pte_file` be called , that is, the page related to the page table entry is not in memory.
Due to the dependence of the kernel's general code on `pte_file` , this function also needs to be defined when a certain architecture does not support non-linear mapping. In this case, the function always returns 0 .

The following table summarizes all the functions used to manipulate PTE items . These functions are often three in one group, which are used to set, delete, and query a specific attribute .

Functions	Comment
<code>pte_present</code>	Is the page in memory
<code>pte_read</code> <code>pte_rdprotect</code> <code>pte_mkrread</code>	Can the page be read from user space Clear the read permission of this page Set read permissions
<code>pte_write</code> <code>pte_wrprotect</code> <code>pte_mkwrite</code>	Can you write to this page Clear the write permission of this page Set write permissions
<code>pte_exec</code> <code>pte_exprotect</code> <code>pte_mkexec</code>	Can the data in this page be executed as binary code? Clear the permission to execute the binary data in this page Allow the content of the page to be executed
<code>pte_dirty</code> <code>pte_mkclean</code> <code>pte_mkdirty</code>	Is the page dirty? Whether its content has been modified "Clear" page, usually means to clear the <code>_PAGE_DIRTY</code> bit Mark page as dirty
<code>pte_file</code>	Does this page table entry belong to a non-linear mapping?
<code>pte_young</code> <code>pte_mkold</code> <code>pte_mkyoung</code>	Is the access bit (usually <code>_PAGE_ACCESS</code>) set Clear access bit Set the access bit, <code>_PAGE_ACCESSED</code> on most architectures

Not all CPU hardware supports the above operations. For example, the IA-32 processor only supports two control methods, which allow reading and writing respectively. In this case, the architecture-related code will try to imitate the required semantics as much as possible.

3.1.2 API

The following lists all the functions used to create the new directory / page table . All architectures must implement the functions in the table so that the memory management code can create and destroy the page table.

Functions	Comment
-----------	---------

mk_pte	Create a page table entry. The page instance and the required page access permissions must be passed as parameters
pte_page	Get the page instance address corresponding to the page described by the page table entry
pgd_alloc pud_alloc pmd_alloc pte_alloc	Allocate and initialize the memory that can hold a complete page directory or page table (not just a table entry) , how much memory space each page directory or page table needs can be calculated based on the previous knowledge
pgd_free pud_free pmd_free pte_free	Release the memory occupied by the page directory or page table
set_pgd set_pud set_pmd set_pte	Set the value of an item in the page table

3.2 Kernel virtual address space

3.2.1 Division of address space

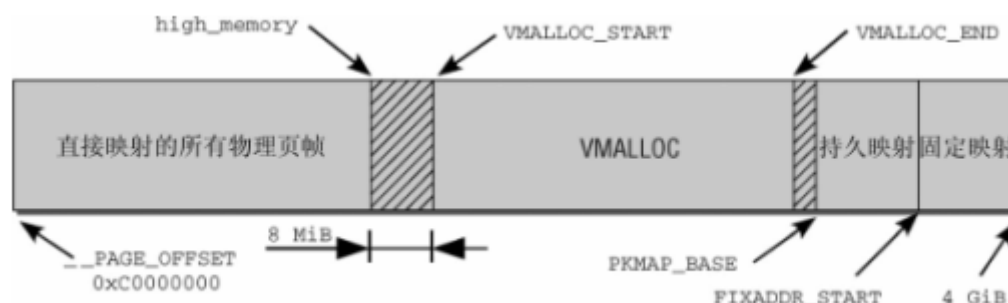
Chapter 1 mentioned that on IA-32 systems, the kernel usually divides the total available virtual address space of 4 GiB at a ratio of 3:1 . The low-end 3 GiB is used for user state applications (each application has an exclusive 3GB virtual address space), while the high-end 1 GiB is dedicated to the kernel. (3:1 is not absolute, some systems use other division methods, but except for special purposes, generally 3:1 division)

The main motivations for this division are as follows :

- When the execution of the user application program switches to the core mode (this will always happen, for example, when a system call is used or a periodic clock interrupt occurs), the kernel must be loaded in a reliable environment. Therefore, it is necessary to allocate a part of the address space to the kernel for exclusive use.
- 物理内存页被映射到内核地址空间的开始，以便内核无需复杂的页表操作即可直接访问。

Although the virtual address part used for user-level processes changes with process switching, the kernel part is always the same.

Dividing the address space according to the ratio of 3 : 1 , but roughly reflects the situation in the kernel, and the kernel address space itself is divided into sections. The details are shown in the figure below :



This picture has been quoted many times before. Note that this picture only indicates the purpose of each area of the virtual address space, which has nothing to do with the allocation of physical memory.

- The first segment of the address space is used to map the physical memory pages of the system to the virtual address space of the kernel. Since the kernel address space starts at offset 0xC0000000, which is often mentioned 3 GiB, each virtual address x corresponds to the physical address $x - 0xC0000000$, so this is a simple linear translation.

Directly map the area from 0xC0000000 to the `high_memory` address. The exact value of `high_memory` will be discussed later. If the total capacity of physical memory exceeds `high_memory`, the kernel cannot directly map all physical memory. In this case, the kernel introduces the concept of high-end memory: 0xC0000000 - `highmem` belongs to the normal area; `highmem` - 4GB belongs to the `highmem` area.

For the normal area, each architecture ported by the kernel must provide two macros for the conversion between physical and virtual addresses (in the end this is a platform-related task)

- ✓ `__pa(vaddr)` returns the physical address related to the virtual address `vaddr`
- ✓ `__va(paddr)` calculates the virtual address corresponding to the physical address `paddr`

Both functions use void pointers and unsigned long operations, because these two data types are equally applicable to representing memory addresses.

Remember, these functions are not suitable for handling arbitrary addresses in the virtual address space, and can only be used for the consistent mapping part of them!

- The second segment of the address space: `highmem` - 4GB. As shown in the figure above, this part has 3 purposes
 - `vmalloc` area: the contiguous memory area in the virtual memory but the discontinuous memory area in the physical memory can be allocated in the `vmalloc` area. This mechanism is usually used in user processes, and the kernel itself will try its best to avoid non-contiguous physical addresses. The kernel usually succeeds because most of the large memory blocks are allocated to the kernel at startup, when the memory fragmentation is not serious. But on a system that has been running for a long time, when the kernel needs physical memory, the available space may be discontinuous. This type of situation mainly occurs when modules are dynamically loaded.

`VMALLOC_START` and `VMALLOC_END` define the beginning and end of the `vmalloc` area. These two values are not directly defined as constants:

`VMALLOC_START` depends on how much virtual address space memory is used when directly mapping physical memory (and therefore also depends on the `high_memory` variable above). Also takes into account the fact that the core, i.e. between the two regions have at least `VMALLOC_OFFSET` a gap, and `vmalloc` region may be from `VMALLOC_OFFSET` start address divisible.

`VMALLOC_END` depends on whether high-end memory support is enabled. If not enabled, then no lasting mapped area, this time `VMALLOC_END = (FIXADDR_START - 2 * PAGE_SIZE)`; otherwise `VMALLOC_END (PKMAP_BASE - 2 * PAGE_SIZE)`. Always leaves two, as `vmalloc` area and two regions The split between.

- `PKMAP` area: Persistent mapping is used to map non-persistent pages in the high-end memory domain to the kernel.
- `FIXADDR` area: Fixed mapping is a virtual address space item associated with a fixed page in the physical address space.

In the ARM architecture, the address of `VMALLOC_END` is defined as a constant, and the value of `high_memory` is related to the size of the `VMALLOC` area. (For details, see 2.3.3 "Determining the boundary value of low-end / high-end memory").

And long-lasting in the mapping area ARM architecture was placed `PAGE_OFFSET` front, fixed mapping area placed `VMALLOC_END` after. Hereinafter simply discussed `KMAP` & `FIXADDR` region.

3.2.2 ARM 64 kernel virtual address space

Recent contacts ARM64 chip , simply looked at the kernel code , in this already known to the kernel virtual address space is divided listed , may not be accurate , subsequent updates . (The Add @ 2017-3-19)

```
//arch/arm64/include/asm/memory.h

/*
 * PAGE_OFFSET - the virtual address of the start of the linear map (top
 * (VA_BITS - 1))
 * KIMAGE_VADDR - the virtual address of the start of the kernel image
 * VA_BITS - the maximum number of bits for virtual addresses.
 * VA_START - the first kernel virtual address.
 * TASK_SIZE - the maximum size of a user space task.
 * TASK_UNMAPPED_BASE - the lower boundary of the mmap VM area.
 */
#define VA_BITS (CONFIG_ARM64_VA_BITS)
#define VA_START (UL(0xffffffffffffffff) << VA_BITS)
#define PAGE_OFFSET (UL(0xffffffffffffffff) << (VA_BITS - 1))
#define KIMAGE_VADDR (MODULES_END)
#define MODULES_END (MODULES_VADDR + MODULES_VSIZE)
#define MODULES_VADDR (VA_START + KASAN_SHADOW_SIZE)
#define MODULES_VSIZE (SZ_128M)
#define VMEMMAP_START (PAGE_OFFSET - VMEMMAP_SIZE)
#define PCI_IO_END (VMEMMAP_START - SZ_2M)
#define PCI_IO_START (PCI_IO_END - PCI_IO_SIZE)
#define FIXADDR_TOP (PCI_IO_START - SZ_2M)
#define TASK_SIZE_64 (UL(1) << VA_BITS)

//arch/arm64/include/asm/pgtable.h
#define VMALLOC_START (MODULES_END)
#define VMALLOC_END (PAGE_OFFSET - PUD_SIZE - VMEMMAP_SIZE - SZ_64K)
#define vmemmap ((struct page *)VMEMMAP_START - (memstart_addr >> PAGE_SHIFT))
```

Process virtual address space : 0 – TASK_SIZE_64

Kernel virtual address space : VA_START – 0xffffffffffffff

1. VA_START – (VA_START + KASAN_SHADOW_SIZE) : KASAN shadow memory virtual address space
2. MODULES_VADDR - MODULES_END : kernel module virtual address space . The size is MODULES_VSIZE (SZ_128M) .
Note #define MODULES_VADDR (VA_START + KASAN_SHADOW_SIZE)
3. VMALLOC_START - VMALLOC_END : the virtual address of the start of the kernel image , which is the vmalloc area .
Note #define VMALLOC_END (MODULES_END)
#define VMALLOC_END (PAGE_OFFSET - PUD_SIZE - VMEMMAP_SIZE - SZ_64K)
4. VMEMMAP_START - PAGE_OFFSET : This size of the area is VMEMMAP_SIZE , it is not immediately VMALLOC_END beginning of , from VMALLOC_END to VMEMMAP_START between there are a number of other areas (now do not know what , later to add) .

To understand the role of this region , to be a little fee something . Kernel memory have 3 Zhong model : FLAT memory model , for discontiguous Memory Model , Sparse Memory Model . To simplify understanding , this paper default is FLAT Memory Model for the background . But with the kernel development , slowly turning Sparse memory model. About this 3 Zhong mode , this article written by very good , you can refer to : http://www.wowotech.net/memory_management/memory_model.html (http://www.wowotech.net/memory_management/memory_model.html) (The content of the link has been copied to " 1.5 Three Memory Models in the Linux Kernel ") .

In FLAT model in , the `mem_map []` array is used to store each `struct page` structure , through this array , can complete a physical address and `struct page` mutual conversion (`pfn_to_page` / `page_to_pfn`) .

In Sparse model in , all of the `struct Page` stored in `VMEMMAP_START` defined address space within , and the corresponding `pfn_to_page` and `page_to_pfn` defined as follows :

```
#define __pfn_to_page(pfn)  (vmemmap + (pfn))
#define __page_to_pfn(page)  (unsigned long)((page) - vmemmap)
```

V MEMMAP similar start address of the array , PFN is . Index `vmemmap` is defined as follows :

```
#define vmemmap      ((struct page *)VMEMMAP_START - \
                     SECTION_ALIGN_DOWN(memstart_addr >> PAGE_SHIFT))
```

There is no doubt that we need to allocate an address in the virtual address space to place the `struct page` array (the array contains all the physical memory span space `page`), which is the definition of `VMEMMAP_START`

5. `PAGE_OFFSET` : The Start of The Virtual address of Map The Linear (linear starting virtual address map), but also is the starting address of the kernel code to be linked .

3.2.3 Create a kernel page table (`paging_init`)

`paging_init` responsible for setting up the kernel page table logical address space, that is, the physical memory of the normal area to one mapping kernel virtual address space of the normal area.

After the page table is created, when the kernel later calls the partner system interface to obtain the physical page frame from the `ZONE_NORMAL` domain, there is no need to change the page table (for example, `alloc_pages` or `kmalloc`). In contrast, if the physical page frame is obtained from the `ZONE_HIGHMEM` domain, the page table entry (for example, `vmalloc`) needs to be changed every time .

The global page directory of the kernel's entire virtual address space is stored in the `swapper_pg_dir` variable . When the kernel is loaded into the memory by `uboot` , the kernel assembly code will run first. The entire assembly code can be divided into two parts : the first part is address-independent, and the second part is address-related (associated to the address `0xc0000000`). The first part of the code will initialize the `swapper_pg_dir` variable , then turn on the MMU, and then jump to the second Part . At this time, the global page directory only maps about 1M bytes of physical memory .

The page table of the entire logical address space is established in the `paging_init` phase , so the main function of `paging_init` is to fill the rest of `swapper_pg_dir` (of course, including the corresponding PUD PMD PTE) .

The general flow of the code is as follows :

setup_arch -> paging_init -> map_lowmem .

map_lowmem (arch/arm/mm/mmu.c):

- ✓ for_each_memblock (Memory, REG) : for memblock a of each block
 - IF (Start >= arm_lowmem_limit) BREAK;; if physical memory starting address is within the high memory region , the end of the loop . Opinion memblock the blocks have is already sorted the
 - Otherwise , according to different situations , set some parameters (mainly to indicate whether this block of memory is RWX or RW) , and then call create_mapping

create_mapping (arch/arm/mm/mmu.c):

- ✓ Do some parameter checks
- ✓ PGD = pgd_offset_k (addr); : The virtual address , from the global page directory swapper_pg_dir extracted corresponding table entry
- ✓ for from addr to addr_length all physical memory , call alloc_init_pud create pud page directory

```
end = addr + length ;  
do {  
    unsigned long next = pgd_addr_end ( addr , end );  
  
    alloc_init_pud ( pgd , addr , next , phys , type );  
  
    phys += next - addr ;  
    addr = next ;  
} while ( pgd ++, addr != end );
```

alloc_init_pud (arch/arm/mm/mmu.c):

Since the ARM architecture does not have PUD, there is no actual alloc action here , but alloc_init_pmd is directly called

alloc_init_pmd (arch/arm/mm/mmu.c):

Since the ARM architecture does not have PMD, there is no actual alloc action here , but alloc_init_pte is directly called

alloc_init_pte (arch/arm/mm/mmu.c):

- ✓ early_pte_alloc : This function first allocates a piece of memory for the pte page table , and sets the memory address to the table entry corresponding to the upper level page directory
- ✓ do { set_pte_ext ...} the while (XXX): then for each physical memory , initializing each of the page table entries , this will also lower the page table entry is provided , for storage protection . On page tables each low bit For the meaning of " 3.1.1 pte flags "

3.2.4 PKMAP & FIXADDR

PKMAP

Different from the X86 architecture , the ARM architecture puts the PKMAP area before the PAGE_OFFSET . The definition is as follows :

```
// arch/arm/include/asm/highmem.h
```

```

#define PKMAP_BASE (PAGE_OFFSET - PMD_SIZE)
#define LAST_PKMAP PTRS_PER_PTE
#define LAST_PKMAP_MASK (LAST_PKMAP - 1)

// arch/arm/include/asm/pgtable-2level.h
#define PMD_SHIFT 21
#define PMD_SIZE (1UL << PMD_SHIFT)

```

From the above code, it can be seen that the starting position of the PKMAP area is PAGE_OFFSET - 2M, and the size is the number of entries in a page table . In the ARM system, this value is 512. It seems that this area can only map 512 addresses .

In addition , if PAGE_OFFSET is 0xc0000000 words , it means that each virtual address of the application are not available 0 - 3GB, otherwise they will be here with the PKMAP regional conflict . In fact it does , ARM system architecture , TASK_SIZE defined as :

```

// arch/arm/include/asm/memory.h

#define TASK_SIZE (UL(CONFIG_PAGE_OFFSET) - UL(SZ_16M))

```

FIXADDR

The start and end positions of the FIXADDR area are defined as follows :

```

// arch/arm/include/asm/fixmap.h

#define FIXADDR_START 0xffc00000UL
#define FIXADDR_END 0xfff00000UL
#define FIXADDR_TOP (FIXADDR_END - PAGE_SIZE)

```

Note that VMALLOC_END is also defined as a fixed value of 0xff800000UL, so there is a 4M partition between VMALLOC_END and FIXADDR_START .

initialization

The initialization of the P KMAP area and the FIXADD area is also completed in paging_init , and the process is as follows :

setup_arch -> paging_init -> kmap_init

```

// arch/arm/mm/mmu.c

static void __init kmap_init ( void )
{
#ifdef CONFIG_HIGHMEM
    pkmap_page_table = early_pte_alloc ( pmd_off_k ( PKMAP_BASE ),
        PKMAP_BASE , _PAGE_KERNEL_TABLE );
#endif

    early_pte_alloc ( pmd_off_k ( FIXADDR_START ), FIXADDR_START ,

```

```

        _PAGE_KERNEL_TABLE );
    }

```

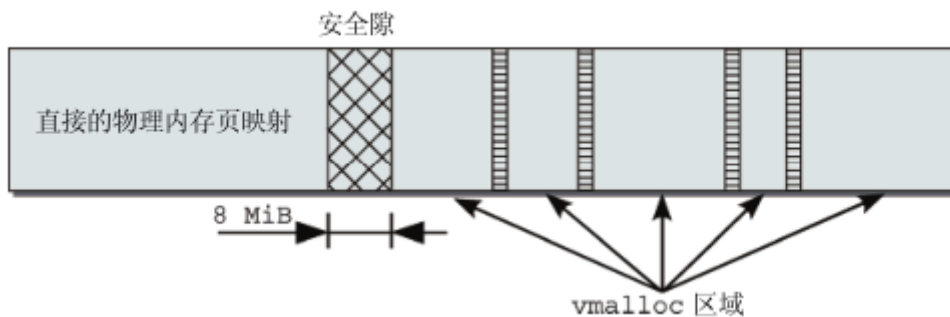
Note that the PKMAP area exists only when CONFIG_HIGHMEM is enabled .

Regarding the API details of PKMAP and FIXADDR , related data structures , working logic, etc. , I will not go into details here . For details , see "In-depth Linux Kernel Architecture 3.5.8 "

3.2.5 vmalloc

Based on the above, we know that physically continuous mapping is the best for the kernel, but it is not always successful. When allocating a large block of memory, you may not be able to find a contiguous block of memory even if you try your best. This is not a problem in user space, because ordinary processes use the processor's paging mechanism to map discontinuous physical pages to continuous virtual address spaces. Of course, this will slow down the speed and occupy TLB .

The same technique can be used in the kernel. There is a VMALLOC area in the kernel virtual address space, which is used to establish continuous mapping. The breakdown of the VMALLOC area is shown in the figure below :



The VMALLOC area is divided into multiple sub-areas , and the sub-areas are separated by a memory page . The separation between different vmalloc sub-areas is also to prevent incorrect memory access operations. This situation will only occur because of a kernel failure and should be reported through system error messages, rather than allowing data in other parts of the kernel to be secretly modified. Because the separation is established in the virtual address space, precious physical memory pages are not wasted.

working principle

I remember the first 3 chapters begins with the introduction of virtual memory management 3 aspects of it : virtual space allocation ; the physical page frame allocation ; build page tables .

The vmalloc system follows this principle .

- space allocation : The virtual address space of the vmalloc system starts at VMALLOC_START, Ends in VMALLOC_END. Whenever the user calls the API to apply for memory , the vmalloc system will create a sub-area in the virtual address space , and a sub-area is represented by a struct vm_struct structure
- page frame allocation : After the virtual space is allocated , the vmalloc system will call the partner system's API to apply for physical page frames from the partner system . Since vmalloc does not require the page frames to be physically continuous , it is one when applying for page frames from the partner system. Page by page
- build page tables : The last step is to build a page table , map physical and virtual addresses page frame up .

data structure

vm_struct

Each sub-area of the vmalloc area is described by a struct vm_struct , and all the sub-areas will be chained through a linked list , so that the kernel can know which virtual addresses in the vmalloc area have been used and which are free .

Note that the user process virtual address space will also be divided into sub-areas , and the kernel code uses struct vm_area_struct to abstract each sub-areas . Although the names and purposes are similar , these two structures cannot be confused .

Header file : include/linux/vmalloc.h

struct vm_struct	Comment
struct vm_struct *next;	Point to the next sub-area , through this pointer to string together each sub-area
void *addr	Defines the starting address of the allocated sub-area in the virtual address space
unsigned long size	size represents the length of the subregion
	<p>flags stores the set of flags associated with the sub-area , and it is only used to specify the type of the sub-area :</p> <pre>include/linux/vmalloc.h #define VM_IOREMAP 0x00000001 /* ioremap() and friends */ #define VM_ALLOC 0x00000002 /* vmalloc() */ #define VM_MAP 0x00000004 /* vmap()ed pages */ #define VM_USERMAP 0x00000008 /* suitable for remap_vmalloc_range */ #define VM_VPAGES 0x00000010 /* buffer for pages was vmalloc'ed */ #define VM_UNINITIALIZED 0x00000020 /* vm_struct is not fully initialized */ #define VM_NO_GUARD 0x00000040 /* don't add guard page */ #define VM_KASAN 0x00000080 /* has allocated kasan shadow memory */</pre>
struct page **pages	<p>pages is a pointer , pointing to page pointer array .</p> <p>Each array member represents a page instance mapped to a physical memory page in the virtual address space</p>
unsigned int nr_pages	Specify the number of array items in pages , that is , the number of physical memory pages involved
phys_addr_t phys_addr	<p>It is only needed when the physical memory area described by the physical address is mapped with ioremap .</p> <p>ioremap used to map a given physical address to vmalloc a sub-region , for access to the kernel . The most common situation is to map the peripheral register address to kernel virtual address space . In this case , does not call the buddy system allocates physical page frame , but will use a given physical address , the address stored in phys_addr in</p>
const void *caller	Generally point to the __builtin_return_address(0) function , I don't know what it is used for

API

Header file : include/linux/vmalloc.h

vmalloc API	Comment
-------------	---------

void *vmalloc(unsigned long size)	<p>This function requires only one parameter , used to specify the desired length of the memory area , and partner systems in different functions discussed , it is not a page but the unit of length bytes , which is very common in user space programming .</p> <p>The most famous example of using vmalloc is the kernel's implementation of modules. Because the module may be loaded at any time, if the module has a lot of data, there is no guarantee that there will be enough continuous memory available, especially when the system has been running for a long time. If you can use small pieces of memory to splice enough memory, then using vmalloc can circumvent this problem.</p> <p>There are about 400 places where vmalloc is called in the kernel , especially in device and sound drivers.</p> <p>Because the memory pages used for vmalloc are not required to be contiguous , pages using the ZONE_HIGHMEM memory domain are better than other memory domains. This allows the kernel to save more valuable lower-end memory domains without any additional harm.</p> <p>This function returns a virtual address , which can be accessed directly by the kernel code .</p>
void *vzalloc(unsigned long size)	<p>With 0 physical page frames allocated to filling , it is clear that the functions except when a page frame to the partner application system __ GPF_ZERO flag</p>
<p>In addition vmalloc addition , there are other ways to create a virtual continuous mapping . These are discussed below based __vmalloc_node function or use a very similar mechanism</p>	
vmalloc_32	<p>It works in the same way as vmalloc , but it will ensure that the physical memory used can always be addressed with ordinary 32 -bit pointers. If the addressing capability of a certain architecture exceeds the range based on word length calculations, then this guarantee is very important. For example, this is the case on an IA-32 system with PAE enabled .</p>
vmap	<p>Use a page array as a starting point to create a virtual contiguous memory area. Compared with vmalloc , the physical memory location used by this function is not implicitly allocated, but needs to be allocated first and passed as a parameter. This type of mapping can be identified by the VM_MAP flag in the vm_map instance .</p>
ioremap	<p>Unlike all the above mapping methods, ioremap is a processor-specific function and must be implemented on all architectures (its header file is not vmalloc.h) .</p> <p>It can map a memory block taken from the physical address space and used by the system bus for I/O operations into the kernel's address space.</p> <p>This function is used a lot in the device driver and can expose the address area used to communicate with the peripheral to the kernel .</p>
void vfree(const void *addr)	<p>Through the virtual address , the sub-area is released , and the physical page frame is also returned to the partner system , and the related items in the page table are deleted.</p> <p>vfree is used to release the area allocated by vmalloc and vmalloc_32 .</p>
vunmap	<p>Used to release the mapping created by vmap or ioremap .</p>

vmalloc_to_page	Converts a kernel virtual address obtained from vmalloc to its corresponding struct page pointer
-----------------	--

Key code analysis

The implementation code of vmalloc is in mm/vmalloc.c , and the main process is as follows :

void *vmalloc(unsigned long size)

- ✓ __vmalloc_node_flags (size, NUMA_NO_NODE, GFP_KERNEL | __GFP_HIGHMEM)
 - __vmalloc_node
 - ◆ __vmalloc_node_range
 - __get_vm_area_node : Create sub-area
 - ⊗ __vmalloc_area_node : the buddy system application page frame , and establishes page table

Create vm_area (__get_vm_area_node)

Before creating a new virtual memory area, an appropriate location must be found. A linked list composed of vm_area instances manages the sub-areas that have been established in the vmalloc area. The global variable vmlist defined in mm/ vmalloc.c is the header.

```
// mm/vmalloc.c

static struct vm_struct * vmlist __initdata ;
```

The kernel provides an auxiliary function get_vm_area in mm/ vmalloc.c . It acts as the front end of the __get_vm_area_node function. According to the length information of the sub-region, the function tries to find an appropriate position in the virtual vmalloc space.

Since 1 page (warning page) needs to be inserted between each vmalloc sub-area as a safety gap, the kernel first appropriately increases the length of memory that needs to be allocated.

```
// mm/vmalloc.c

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 )
{
    ...

    if (!( flags & VM_NO_GUARD ))
        size += PAGE_SIZE ;

    ...

    va = alloc_vmap_area ( size , align , start , end , node , gfp_mask );

    setup_vmalloc_vm ( area , va , flags , caller );
}
```

After the size is set correctly , call `alloc_vmap_area` to create the sub-area , and finally call `setup_vmalloc_vm` to initialize the value of each member variable of `vm_area`.`setup_vmalloc_vm` is nothing beautiful , we mainly focus on the `alloc_vmap_area` function .

alloc_vmap_area

```
// mm/vmalloc.c

static struct vm_area * alloc_vmap_area ( unsigned long size ,
                                         unsigned long align ,
                                         unsigned long vstart , unsigned long vend ,
                                         int node , gfp_t gfp_mask )
{
```

The start and end parameters are set by the caller to `VMALLOC_START` and `VMALLOC_END`, respectively .

This function first loops through all table elements of `vmlist` until it finds an appropriate item.

```
// mm/vmalloc.c

for ( p = & vmlist ; ( tmp = * p ) != NULL ; p = & tmp -> next ) {
    if (( unsigned long ) tmp -> addr < addr ) {
        if (( unsigned long ) tmp -> addr + tmp -> size >= addr )
            addr = ALIGN ( tmp -> size +
                          ( unsigned long ) tmp -> addr , align );
        continue ;
    }
    if (( size + addr ) < addr )
        goto out ;
    if ( size + addr <= ( unsigned long ) tmp -> addr )
        goto found ;
    addr = ALIGN ( tmp -> size + ( unsigned long ) tmp -> addr , align );
    if ( addr > end - size )
        goto out ;
}
}
```

If `size+addr` is not greater than the starting address of the current inspection area (stored in `tmp->addr`) , then the kernel has found a suitable location. Next, initialize the new linked list element with the appropriate value and add it to the `vmlist` linked list.

If no suitable memory area is found, a `NULL` pointer is returned to indicate failure.

The `remove_vm_area` function removes an existing sub-area from the `vmalloc` address space.

Request page frame (__vmalloc_area_node)

When `vmalloc` calls `__get_vm_area_node` to successfully allocate a virtual address space , it will call `__vmalloc_area_node` to request a physical page from the partner system .

The complete code is not given here, it contains uninteresting security checks. We are more interested in the allocation of physical memory area (ignoring the possibility that there are not enough physical memory pages available) .

```
// mm/vmalloc.c

static void * __vmalloc_area_node ( struct vm_struct * area , gfp_t gfp_mask ,
                                   pgprot_t prot , int node )
{
    ...
    for ( i = 0 ; i < area -> nr_pages ; i ++ ) {
        struct page * page ;

        if ( node == NUMA_NO_NODE )
            page = alloc_page ( alloc_mask );
        else
            page = alloc_pages_node ( node , alloc_mask , order );

        if ( unlikely (! page ) ) {
            /* Successfully allocated i pages, free them in __vunmap() */
            area -> nr_pages = i ;
            goto fail ;
        }
        area -> pages [ i ] = page ;
        if ( gfpflags_allow_blocking ( gfp_mask ) )
            cond_resched () ;
    }

    if ( map_vm_area ( area , prot , pages ) )
        goto fail ;
    return area -> addr ;
    ...
}
```

If the node where the page frame is allocated is explicitly specified, the kernel calls `alloc_pages_node` . Otherwise, use `alloc_page` to allocate page frames from the current node.

The allocated page is removed from the partner system of the relevant node. When calling, `vmalloc` sets `gfp_mask` to `GFP_KERNEL | __GFP_HIGHMEM` . The kernel uses this parameter to instruct the memory management subsystem to allocate page frames from the `ZONE_HIGHMEM` memory domain as much as possible . The reason has been given above: the page frame of the low-end memory domain is more precious, so it should not be wasted in the allocation of `vmalloc` , where the page frame of the high-end memory domain can fully meet the requirements.

When allocating memory from the partner system, it is allocated page by page, rather than a large block at a time. This is a key aspect of `vmalloc` . If you can be sure that you can allocate a contiguous memory area, then there is no need to use `vmalloc` . After all, the purpose of this function is to allocate a large memory block, although the

page frames in the memory block may not be continuous due to memory fragmentation. Splitting the allocation unit as small as possible (in other words, in pages) can ensure that vmalloc can still work even if the physical memory is severely fragmented .

Create page table (map_vm_area)

__vmalloc_area_node last calls map_vm_area dispersed continuous physical memory pages mapped to the virtual vmalloc area .

This function traverses the allocated physical memory pages, allocates and initializes the required directory entries / table entries in the page directories / page tables at all levels .

Some architectures need to flush out the CPU cache after modifying the page table . So the kernel calls flush_cache_vmap , and its definition is architecture-specific. Depends on different CPU types, which may include low-level assembly statements used to flush out the cache.

Free up space (__vunmap)

vfree and vunmap to free up space , " API " an introduction through them .

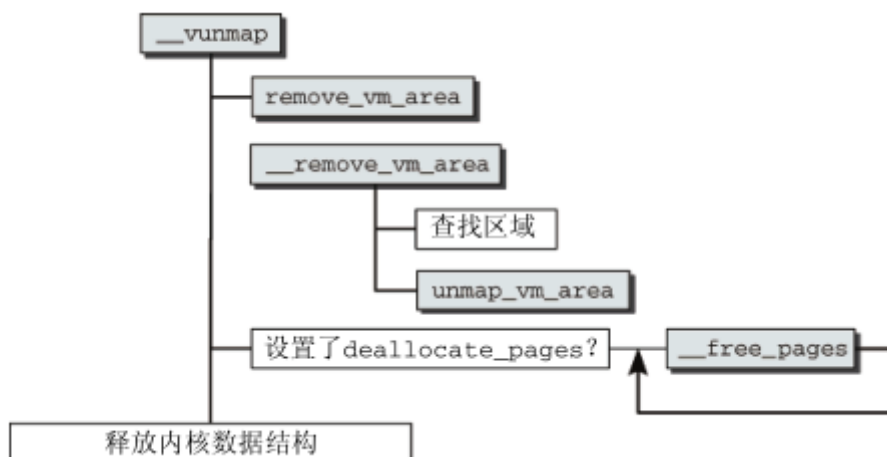
Both functions will eventually call __vunmap :

```
// mm/vmalloc.c

static void __vunmap ( const void * addr , int deallocate_pages )
```

addr represents the starting address of the area to be released, deallocate_pages specifies whether to return the physical memory pages associated with the area to the partner system. Vfree sets the latter parameter to 1 , and vunmap to 0 , because in this case only the mapping is deleted, and the relevant physical memory page is not returned to the partner system.

The following figure shows the code flow of __vunmap :



It is not necessary to explicitly give the length of the area to be released, the length can be derived from the information in the vmlist . So __vunmap first task is to __remove_vm_area (by the remove_vm_area called after the completion of the lock) scanning the list to find items.

unmap_vm_area uses the found instance of vm_area to delete items that are no longer needed from the page table. Similar to when allocating memory, this function needs to manipulate page tables at all levels, but this time you need to delete the items involved. It will also update the CPU cache.

If the parameter deallocate_pages of __vunmap is set to 1 (in vfree), the kernel will traverse all the elements of area->pages , that is, a pointer to the page instance of the physical memory page involved . Then call __free_page for each item to release the page to the partner system.

Finally, the kernel data structure used to manage the memory area must be released.

3.3 User process virtual address space





3.3.1 The layout of the process address space

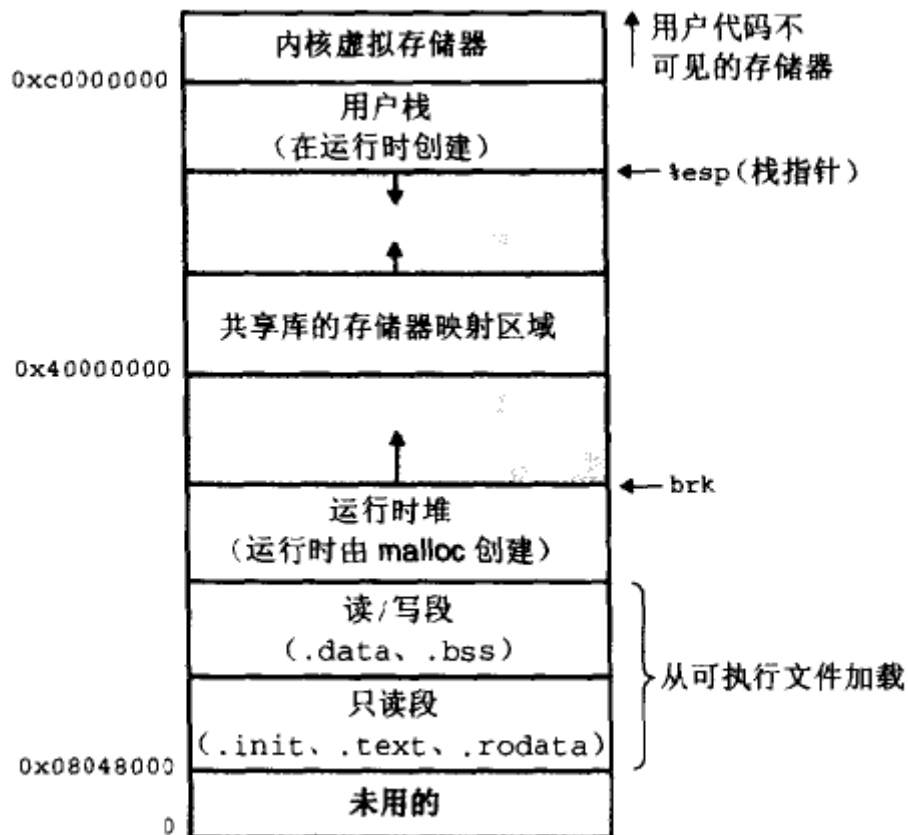
The virtual address space of each process starts at address 0 and extends to $\text{TASK_SIZE} - 1$, above which is the kernel address space. The range of the address space on the IA-32 system can reach $2^{32} = 4 \text{ GiB}$. The total address space is usually divided at a ratio of 3:1 . We will focus on this division in the following. The kernel allocates 1 GiB , and the available part of each user space process is 3 GiB . Other division ratios are also possible, but as discussed above, they can only be useful under very specific configurations and certain workloads.

A very important aspect related to system integrity is that user programs can only access the lower half of the entire address space, not the kernel. If no "agreement" is reached in advance , it is impossible for a user process to manipulate the address space of another process because the address space of the latter is not visible to the former.

No matter which user process is currently active, the contents of the kernel part of the virtual address space are always the same. Depending on the specific hardware, which could be operational by each user process's page table such that the upper half of the virtual address space seemed always with the same. It may also instruct the processor to provide an independent address space for the kernel, which is mapped on the address space of each user.

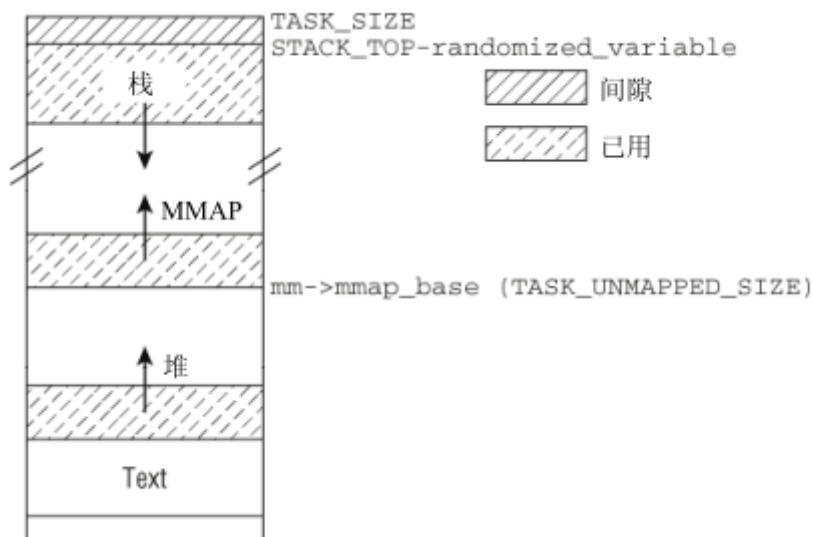
There are several areas in the virtual address space. Its distribution method is specific to the architecture, but all methods have the following common components.

- The binary code of the currently running code. This code is usually called text , and the virtual memory area in which it is located is called the text segment.
- Text constant area , or called read-only data segment .
-  storage area , including R/W data, BSS data. Global variables and static variables are stored here.
-  heap of data (malloc) is dynamically generated .
-  stack used to save local variables and implement function / procedure calls.
-  segment of environment variables and command line parameters.
- Map the contents of the file to the memory mapping area in the virtual address space . This area has the same attributes as the vmalloc area. This area will also be divided into multiple sub-areas, and each sub-area is described by a struct vm_area_struct . Every time a new memory map is created, a sub-area is created in the memory map area.
The memory mapping area can map many things , such as the code of the dynamic library used by the program . The memory mapping will be specifically introduced later .




Two layout methods of memory mapped area

The following figure shows a rough layout diagram, in which certain sections are omitted. This diagram is used to explain why there are two types of address space layouts in the kernel:

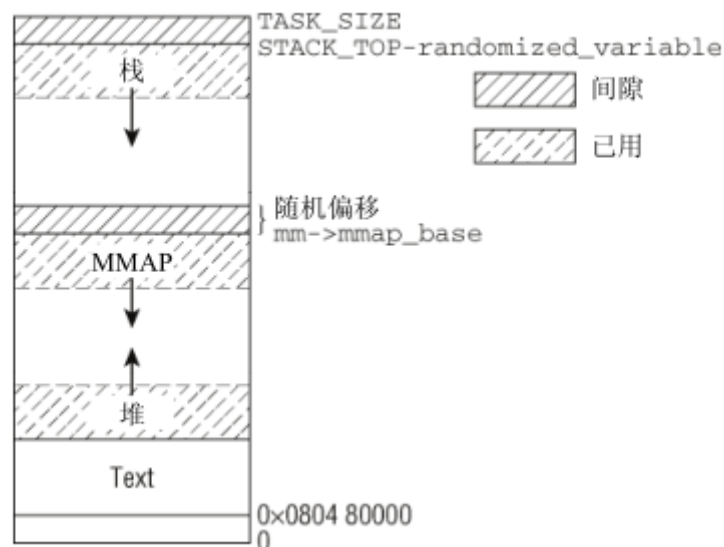


- ✓ How the text segment is mapped into the virtual address space is determined by the ELF standard. Each architecture specifies a specific starting address: The IA-32 system starts at 0x08048000, and there is a space of about 128 MiB between the starting address of the text segment and the lowest available address, which is used to capture the NULL pointer. Other architectures have similar gaps: UltraSparc computers use 0x100000000 as the starting point of the text segment, while AMD64 uses 0x0000000000400000.
- ✓ heap starts immediately after the text segment and grows upward.

- ✓  stack starts at `STACK_TOP` . If the process sets `PF_RANDOMIZE` , the starting point will be reduced by a small random amount. The main purpose of setting `PF_RANDOMIZE` is for security considerations, for example, making it more difficult to attack security vulnerabilities caused by buffer overflows. If an attacker can not find stacks on fixed address, then you want to construct build malicious code through a buffer overflow to gain access to the stack memory area, and then maliciously manipulate the content of the stack will be much more difficult.
Each architecture must define `STACK_TOP` , most of which are set to `TASK_SIZE` , which is the highest available address in the user address space. The parameter list and environment variables of the process are the initial data of the stack.
- ✓ area for memory mapped starting `mm_struct-> mmap_base` , usually set to `TASK_UNMAPPED_BASE` , each architecture will need to be defined. In almost all cases, the value is `TASK_SIZE/3` . It should be noted that if the default configuration of the kernel is used, the starting point of the `mmap` area is not random.

If the computer provides a huge virtual address space, then using the address space layout described above will work very well. However, problems may occur on 32 -bit computers. Consider the case of IA-32 : the virtual address space ranges from 0 to `0xC0000000` , and each user process has 3 GiB available. `TASK_UNMAPPED_BASE` starts at `TASK_SIZE/3` , which is 1 GiB . What's bad is that this means that the heap has only 1 GiB space available, and if it continues to grow, it will enter the `mmap` area, which is obviously not what we want.

To solve this problem, a new virtual address space layout was introduced during the development of kernel version 2.6.7 (the classic layout can still be used) . The new layout is shown below :



The idea is to use a fixed value to limit the maximum length of the stack. Since the stack is bounded, the area where the memory map is placed can start immediately below the end of the stack. Contrary to the classic method, the memory-mapped area is now expanded from top to bottom. Since the heap is still located in a lower area of the virtual address space and grows upward, the `mmap` area and the heap can be relatively expanded until the remaining area in the virtual address space is exhausted. To ensure that the stack does not conflict with the `mmap` area, a security gap is set between the two.

Architecture-specific settings

Each architecture can affect the layout of the virtual address space through several configuration options :

- ✓ If the architecture wants to choose between different address space layouts, you need to set `HAVE_ARCH_PICK_MMAP_LAYOUT` and provide the `arch_pick_mmap_layout` function .
- ✓ When creating a new memory map, unless the user specifies a specific address, the kernel needs to find an appropriate location. If the architecture itself wants to choose a suitable location, it must set the preprocessor symbol `HAVE_ARCH_UNMAPPED_AREA` and define the `arch_get_unmapped_area` function accordingly.

- ✓ When looking for a new memory map low-end memory location, usually start from the lower memory location and gradually search to the higher memory address. The kernel provides a default function `arch_get_unmapped_area` for searching, but if a certain architecture wants to provide a special implementation, you need to set the preprocessor symbol `HAVE_ARCH_GET_UNMAPPED_AREA`.
- ✓ Generally, the stack grows from top to bottom. Architectures with different processing methods need to set the configuration option `CONFIG_STACK_GROWSUP`.

3.3.2 Data structure (mm_struct)

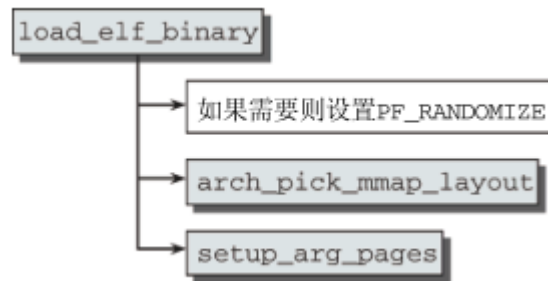
Each process in the system has an instance of struct `mm_struct`, which can be accessed through `task_struct`. This example saves the memory management information of the process. This structure is very large. At present, we only look at the part of this structure that is related to the address space layout:

Header file: `include/linux/mm_types.h`

struct mm_struct	Comment
...	
unsigned long start_code, end_code, start_data, end_data;	<p>The start and end of the virtual address space area occupied by the executable code are marked by <code>start_code</code> and <code>end_code</code> respectively.</p> <p>Similarly, <code>start_data</code> and <code>end_data</code> mark the area containing initialized data.</p> <p>Please note that after the ELF binary file is mapped into the address space, the length of these areas does not change.</p>
unsigned long start_brk, brk, start_stack;	<p>The start address of the heap is stored in <code>start_brk</code>, and <code>brk</code> represents the current end address of the heap area. Although the starting address of the heap is constant during the life cycle of the process, the length of the heap will change, and therefore the value of <code>brk</code> will also change.</p> <p><code>start_stack</code> represents the current position of the stack</p>
unsigned long arg_start, arg_end, env_start, env_end;	The location of the parameter list and environment variables are described by <code>arg_start</code> and <code>arg_end</code> , <code>env_start</code> and <code>env_end</code> , respectively. Both regions are located in the highest region in the stack.
unsigned long mmap_base ulong (*get_unmapped_area) (.....) struct vm_area_struct *mmap	These 3 elements are all related to the memory mapping area and will be introduced in detail in "Managing the Red-Black Tree and Linked List of Sub-Area"
unsigned long task_size	<p>As the name implies, the length of the address space of the corresponding process is stored. This value is usually <code>TASK_SIZE</code>. In the ARM architecture, <code>TASK_SIZE</code> is defined as (memory.h: <code>#define TASK_SIZE (UL(CONFIG_PAGE_OFFSET) - UL(SZ_16M))</code>)</p> <p>But if 32-bit binary code is executed on a 64-bit computer, <code>task_size</code> describes the length of the address space that the binary code is actually visible</p>

3.3.3 Establishing the layout

When using `load_elf_binary` to load an ELF binary file, the above address space layout of the process will be created . The `exec` system call happens to use this function. Loading the ELF file involves a lot of complicated technical details, which has little to do with our subject, so the code flow chart given in the figure below mainly focuses on the various steps required to establish a virtual memory area.



If the global variable `randomize_va_space` is set to 1 , the address space randomization mechanism is enabled. Normally it is enabled, but it will be disabled on Transmeta CPU , because this setting will reduce the speed of such computers. In addition, users can disable this feature through `/proc/sys/kernel/randomize_va_space` .

The work of selecting the layout is done by `arch_pick_mmap_layout` . If the corresponding architecture does not provide a specific function, the default routine of the kernel is used to establish the address space according to the above layout . But we are more interested in how IA-32 chooses between the classic layout and the new layout:

```

arch/x86/mm/mmap_32.c
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    /*
     * 如果设置了personality比特位，或栈的增长不受限制，则回退到标准布局：
     */
    if (sysctl_legacy_va_layout ||
        (current->personality & ADDR_COMPAT_LAYOUT) ||
        current->signal->rlim[RLIMIT_STACK].rlim_cur == RLIM_INFINITY)
    {
        mm->mmap_base = TASK_UNMAPPED_BASE;
        mm->get_unmapped_area = arch_get_unmapped_area;
        mm->unmap_area = arch_unmap_area;
    } else {
        mm->mmap_base = mmap_base(mm);
        mm->get_unmapped_area = arch_get_unmapped_area_topdown;
        mm->unmap_area = arch_unmap_area_topdown;
    }
}
  
```

If the user gives clear instructions through `/proc/sys/kernel/legacy_va_layout` , or if you want to execute a binary file compiled for a different UNIX variant, the old layout is needed . Or the stack can grow indefinitely (the most important point), the system will choose the old layout , because it is difficult to determine the lower bound of the stack, that is , the upper bound of the mmap area.

In the classic configuration, the starting point of the mmap area is `TASK_UNMAPPED_BASE` , its value is `0x4000000` , and the standard function `arch_get_unmapped_area` (although its name has `arch` , this function is not necessarily architecture-specific, and the kernel also provides a standard implementation) Is used to create a new mapping from the bottom up.

When using the new layout, the memory map grows from top to bottom. The standard function `arch_get_unmapped_area_topdown` (I will not describe it in detail) is responsible for this work. The more interesting question is how to choose the base address of the memory map:

```

// arch/x86/mm/mmap_32.c

#define MIN_GAP (128*1024*1024)
#define MAX_GAP (TASK_SIZE/6*5)

static inline unsigned long mmap_base(struct mm_struct *mm)
{
    unsigned long gap = current->signal->rlim[RLIMIT_STACK].rlim_cur;
    unsigned long random_factor = 0;

    if (current->flags & PF_RANDOMIZE)
        random_factor = get_random_int() % (1024*1024);
    if (gap < MIN_GAP)
        gap = MIN_GAP;
    else if (gap > MAX_GAP)
        gap = MAX_GAP;

    return PAGE_ALIGN(TASK_SIZE - gap - random_factor);
}

```

The lowest possible position of the stack can be calculated according to the maximum length of the stack and used as the starting point of the mmap area. But the kernel will ensure that the stack spans at least 128 MiB . In addition, if the specified stack limit is very large, the kernel will ensure that at least a small part of the address space is not occupied by the stack.

If the address space randomization mechanism is required, a random offset will be subtracted from the above position, and the maximum is 1 MiB . In addition, the kernel will ensure that the area is aligned to the page frame, which is a requirement of the architecture.

We go back to `load_elf_binary` . The function finally needs to create the stack in the appropriate location:

```

< FS / binfmt_elf . C >

static int load_elf_binary ( struct linux_binprm * bprm , struct pt_regs * regs
)
{
    ...

    retval = setup_arg_pages ( bprm , randomize_stack_top ( STACK_TOP ),
        executable_stack );

    ...
}

```

The standard function `setup_arg_pages` is used for this purpose. Because this function is only technical, I will not discuss it in detail. This function requires the position of the top of the stack as a parameter. The top of the stack is given by the architecture-specific constant `STACK_TOP` , and then `randomize_stack_top` is called to ensure that the address is randomly offset when the address space randomization is enabled.

3.3.4 Memory mapping

The principle of memory mapping

Everything under Linux is a file , many dynamic libraries (.so), text files (.txt, ...), and many other data are stored in the form of files on the disk . If a user process wants to access these data , it must store these data Copy to physical memory , and then map the physical memory to the virtual address space of the user process , and then the user

process can access the data .

The "layout process address space" previously described , in the virtual address space of a process , specifically a divided region , used in the above map , this area is called memory-mapped region .

However, the actual mapping process is somewhat different from the above steps . We will not first copy the contents of the file to physical memory and then do the mapping . This is unreasonable : if there is a file with a size of 1G , if all are copied to the physical memory will take up 1G memory size , many times the physical memory is simply not so big ; but at some point , we just need to visit a small part of the contents of this document , there is no need to copy the entire file to physical memory , which is Waste of physical memory!

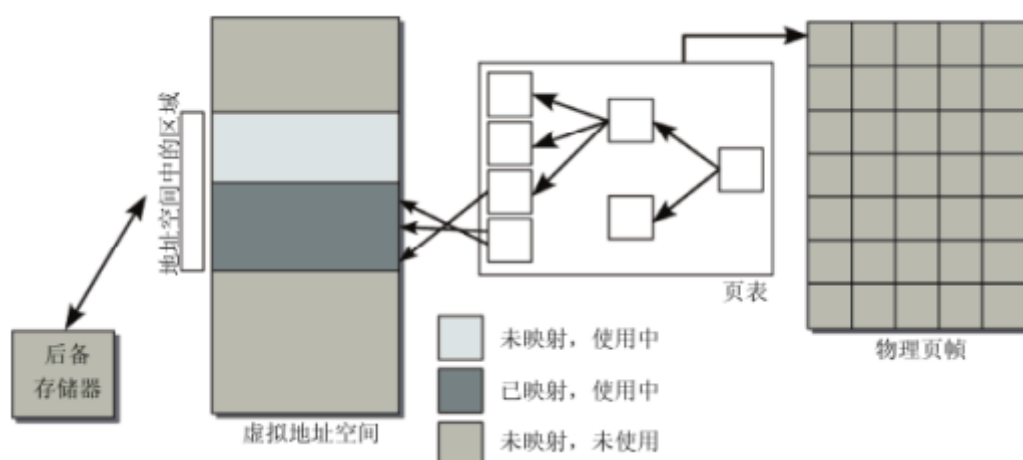
Therefore , the actual processing process is to first establish the mapping between the memory mapping area and the file (at this time, no data will be copied to the physical memory) , and then when the program accesses a virtual address , it is found that the corresponding page table cannot be found. physical memory , but also the region where the address has been mapped to a file , then the system will apply to the partner system memory , data and need to copy memory (note once it is assigned a buddy system PAGE_SIZE amount of memory , and therefore Each copy action will at least copy the contents of PAGE_SIZE) . The kernel's " page fault exception handler " is responsible for applying for the physical page table and copying the data .




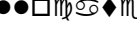
File also has its own address space , the so-called memory-mapped in fact, create a file address space mapping memory-mapped area . Note that once the mapping does not require the address space of the entire file is mapped in the past , this will waste a virtual address space memory map area , we You can map only the part of the file that we need .

After mapping established , it must solve a problem : When a missing page exception occurs , how to read the contents of the file ? File data stored on the hard disk is usually not continuous, and different ways of reading the file may not be the same . kernel use address_space data structure to read the contents of the file abstract methods , no matter what form of documents , is creating a memory map , must be filled address_space interface-related , in order to read the contents of the file back .

In addition , although the above discussion is for the memory mapping area , it is also applicable to other areas of the process address space . For example, the code of the process is stored on the disk, and we map the code file to the code area of the process , and only when a certain part of the code is executed Only then copy the corresponding content into the memory .

Finally , we use a pair of icons to illustrate the processing of " page missing exception " :



- ✓  process tries to access a memory address in the user address space, but the physical address cannot be determined using the page table (there is no associated page in the physical memory) .
- ✓  processor then triggers a page fault exception and sends it to the kernel.
- ✓  kernel will first check whether the " abnormal address " is legal (for example, whether the address belongs to a certain memory mapping area), and then find an appropriate backup memory (`address_space`) .
- ✓  physical memory pages and read the required data from the backing memory to fill .
- ✓ With the help of the page table, the physical memory page is merged into the address space of the user process, and the application program resumes execution.

These operations are transparent to the user process. In other words, the process will not notice whether the page is actually in physical memory, or it needs to be loaded through a " page fault " .

Related data structure

Sub-area (`vm_area_struct`)

The memory mapping area will be divided into multiple sub-areas , each sub- area is represented as an instance of `vm_area_struct` , its definition (simplified form) is as follows :

Header file : `include/linux/mm_types.h`

struct <code>vm_area_struct</code>	Comment
<code>struct mm_struct *vm_mm;</code>	The address space of the owning process
<code>unsigned long vm_start;</code>	The starting position of the subregion in the process virtual address space
<code>unsigned long vm_end;</code>	The ending position of this subregion in the process virtual address space
<code>struct vm_area_struct *vm_next, *vm_prev;</code>	Used to link all sub-areas , the sub-areas in the linked list will be sorted in order of increasing address
<code>struct rb_node vm_rb ;</code>	<code>mm_struct</code> will in all sub-regions under the red-black tree management , <code>vm_rb</code> corresponds to a red-black tree node , for the integration region into a red-black tree book
<code>unsigned long rb_subtree_gap</code>	Largest free memory gap in bytes to the left of this VMA. Either between this VMA and <code>vma->vm_prev</code> , or between one of the VMAs below us in the VMA rbtree and its <code>->vm_prev</code> . This helps <code>get_unmapped_area</code> find a free area of the right size
<code>pgprot_t vm_page_prot</code>	Access permissions of this VMA The definition of <code>pgprot_t</code> is discussed in " 3.1.1 pte flags "

<p>unsigned long vm_flags</p>	<p>A set of signs describing the area :</p> <p>Header file : include/linux/mm.h</p> <ul style="list-style-type: none"> ➤ VM_READ , VM_WRITE , VM_EXEC , VM_SHARED respectively specify whether the content of the page can be read, written, executed, or shared by several processes ➤ VM_MAYREAD , VM_MAYWRITE , VM_MAYEXEC , VM_MAYSHARE are used to determine whether the corresponding VM_* flag can be set . This is required by the mprotect system call. ➤ VM_GROWSDOWN and VM_GROWSUP indicate whether an area can be expanded downward or upward . For the heap area , since the heap grows from bottom to top , it is set to VM_GROWSUP For the stack area , because the stack grows from top to bottom , it is set to VM_GROWSDOWN For the memory mapped area , set according to the actual situation ➤ If the region is likely to start to finish reading order, is set VM_SEQ_READ VM_RAND_READ specifies that the read may be random These two flags are used to " prompt " the memory management subsystem, and block layer , in order to optimize its performance ➤ If you set VM_DONTCOPY , in the region of the associated fork does not copy the system call execution ➤ VM_DONTEXPAND forbidden area to be extended by mmap system call ➤ If the region is based on the structural support system of some giant page, set VM_HUGETLB flag ➤ Whether the specified area of VM_ACCOUNT is included in the calculation of the overcommit feature. These features limit memory allocation in many ways
<pre>struct { struct rb_node rb; unsigned long rb_subtree_last; } shared;</pre>	<p>Through the mm_struct structure of the process , we can easily find all the subregions belonging to the process . Each subregion may be mapped to a certain file , so it is also very convenient to know which files are mapped by a process.</p> <p>However , a file may be mapped to multiple processes . This mapping is called " shared mapping " . For example, the C standard library may be mapped by multiple processes . So the kernel sometimes needs to know a file or a certain file Which processes are the regions mapped to .</p> <p>shared is used to solve this problem , shared.rb is a red-black tree node , used to associate the region with mapping file , a file with a red-black tree is also used to manage all sub-regions associated with it . After More details will be discussed in the article "File Address Space"</p>
<pre>struct list_head anon_vma_chain struct anon_vma *anon_vma</pre>	<p>Don't understand the details for the time being</p>
<pre>const struct vm_operations_struct *vm_ops</pre>	<p>vm_ops is a pointer , pointing to a collection of a number of methods , these methods for the execution of various operating standards region</p> <p>Later " vm_operations_struct " will discuss more details of this structure</p>

unsigned long vm_pgoff	Specifies the offset of the file mapping. This value is used when only part of the content of the file is mapped (if the entire file is mapped , the offset is 0) The unit of the offset is not bytes , but pages (ie PAGE_SIZE) . This is reasonable , because the kernel only supports mapping in units of whole pages , and smaller values are meaningless .
struct file * vm_file	In the Linux system , a file is opened once , and a file instance relative to it will be generated . vm_file points to the file instance and describes a file to be mapped . If the object is not a mapping file, for the NULL means the needle
struct file *vm_pfile	Don't understand the details for the time being
void * vm_private_data	vm_private_data can be used to store private data and is not operated by general memory management routines. The kernel only ensures that the member is initialized to a NULL pointer when creating a new area . Currently, only a few sound and video drivers use this option.
struct vm_userfaultfd_ctx vm_userfaultfd_ctx	Don't understand the details for the time being

vm_operations_struct

Header file : include/linux/mm.h

struct vm_operations_struct	Comment
void (*open)(struct vm_area_struct * area) void (*close)(struct vm_area_struct * area)	When creating and deleting areas , call open and close respectively . These two interfaces are usually not used and set to NULL pointers .
int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf)	Fault is very important. If a virtual memory page in the address space is not in physical memory, the automatically triggered page fault exception handler will call this function . This function reads the corresponding data into a physical memory page mapped in the user address space.
...	There are some other functions , not to introduce them one by one

Manage red-black trees and linked lists of sub-regions

We know that struct mm_struct very important , according to the foregoing discussion , the structure provides all the information necessary to process the layout in memory .

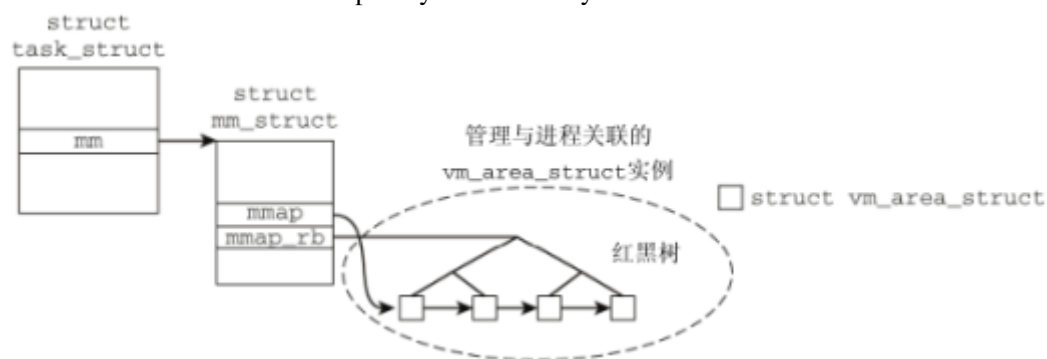
In addition , it also includes the following members , which are used to manage all sub-areas in the memory-mapped area of the user process :

Header file : include/linux/mm_types.h

struct mm_struct	Comment
...	

struct vm_area_struct *mmap	<p>mmap points to a sub-area of the memory mapping area , and each sub-area has its own vm_next & vm_prev pointing to the adjacent sub-area .</p> <p>In this way , through mmap, we can traverse all sub-regions belonging to the process .</p> <p>Note that all of the sub- region will be on the starting address to be included in the list in ascending order</p>
struct rb_root mm_rb	<p>The follow node of the red-black tree , and each sub-area belonging to the process is related to this root node through struct rb_node vm_rb .</p> <p>Red-black tree is a self-balancing binary search tree , very efficient . If only chain management by sub-region , when a large number of sub-areas , will lead to find the low efficiency of a particular sub-region , because it may have to scan the entire Linked list .</p>
unsigned long mmap_base	Represents the starting address for memory mapping in the virtual address space
unsigned long (*get_unmapped_area) (....)	Call get_unmapped_area to find the appropriate location for the new mapping in the memory mapped area

Finally, we use a picture to explain the above management structure . Note that the representation of the picture is only symbolic and does not reflect the complexity of the real layout .



The address space of the file (address_space)

Each open file (and each block device, as these may be performed by a memory-mapped device file) are expressed as a struct File one example , the structure contains a pointer to file address space struct address_space pointer .

```
// include/linux/fs.h

struct file {
...
    struct address_space * f_mapping ;
...
}
```

Each file has its own address space , with struct address_space represent . A file may be mapped to multiple processes , so we need to address_space recorded in the end of this document which is mapped to the process , in order to get at the kernel when necessary The information .

The details of the address_space structure are as follows , we are only about the part related to the mapping for the time being :

Header file : include/linux/fs.h

struct address_space	Comment
...	
struct inode *host;	struct inode representing the file itself , and struct file different , each file is opened regardless of how many times , only there is an inode instance .
struct rb_root i_mmap	Red-black tree with nodes , to mount an all private and shared maps . vm_area_struct ->shared.rb is used to attach vm_area_struct to this follower node . Through i_mmap, we can find all the vm_area_struct related to the file ; and through vm_area_struct ->mm_struct, we can find the corresponding process . Therefore, through i_mmap, we can find all processes that map the file
const struct address_space_operations *a_ops	This data structure provides a general method for obtaining the contents of a file .

address_space_operations

Header file : include/linux/fs.h

struct address_space_operations	Comment
int (*writepage)(struct page *page, struct writeback_control *wbc)	writepage contents page from physical memory to write back to the position corresponding to the block device , in order to save changes permanently
int (*writepages)(struct address_space *, struct writeback_control *)	Write multiple pages at once
int (*readpage)(struct file *, struct page *)	readpage reads a page from a potential block device into physical memory
int (*readpages)(struct file *filp, struct address_space *mapping, struct list_head *pages, unsigned nr_pages)	Read multiple pages at once
...	

The memory mapped area is associated with the file address space

How to establish the connection between vm_operations_struct and address_space ? There is no static link that assigns an instance of one structure to another structure. These two structures are connected using the standard implementation provided by the kernel for vm_operations_struct , which is used in almost all file systems.

mm/filemap.c

```
struct vm_operations_struct generic_file_vm_ops = {
    .fault = filemap_fault,
};
```

The implementation of filemap_fault uses methods such as readpage of address_space_operations .

In this way , when a page fault occurs , the handler will find the vm_area_struct corresponding to the virtual address that triggered the exception , then find the vm_operations_struct, then call its .fault function , and then call methods such as readpage .

Memory mapping area related operations

Introducing the API before , we first introduce the operations associated with the memory-mapped area , because the API function which describes the operational functions will be cited here .

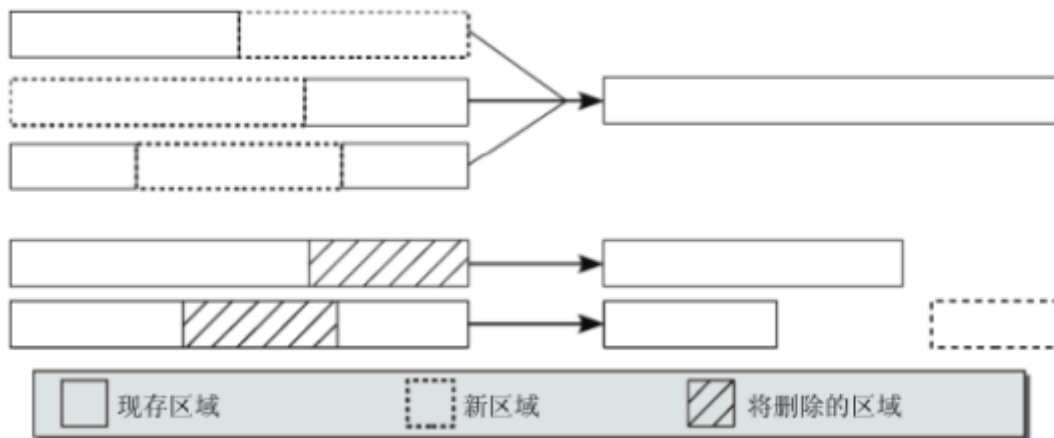
Memory-mapped area is actually a section of the process virtual address space , the area commonly used API has two :

- `mm_create_subarea` mapping : create a new subarea in the memory mapping area .
- Delete map : created before deleting the sub-region

In the two API derived based further on several operations :

- region query : through a given virtual address , find the address belongs to the sub-region
- `mm_check` check : Before creating a sub-region , we need to find a suitable location in the memory mapping area .
- Region merger : If a new region is directly added immediately before and after the existing region , or a new region is inserted between two regions , and all the regions involved have the same access rights, and are continuous data mapped from the same backing memory. At this time, the kernel merges the involved areas into one .
- `mm_split` split : If you delete a part of the head / tail of an existing area , you must cut off the existing area accordingly . In addition, if you delete a part of the middle of an existing area , you must split the existing area into 2 new ones. Area .

The following figure shows an example of area merging and area deletion :



Below , we introduce the codes related to these operations respectively .

Area query (`find_vma`)

Given a certain virtual address, `find_vma` can find the first area in the user address space whose end address is after the given address, that is, the first area that satisfies the condition of `addr < vm_area_struct->vm_end` . The parameters of this function include not only the virtual address (`addr`), but also a pointer to an instance of `mm_struct` , which specifies the address space of which process to scan.

Implementation file : `mm/mmap.c`

```
struct vm_area_struct * find_vma ( struct mm_struct * mm , unsigned long addr )
{
    struct rb_node * rb_node ;
```

```

struct vm_area_struct * vma ;

/* Check the cache first. */
vma = vmacache_find ( mm , addr );
if ( likely ( vma ))
    return vma ;

rb_node = mm -> mm_rb . rb_node ;

while ( rb_node ) {
    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 );
return vma ;
}

EXPORT_SYMBOL ( find_vma );

```

- ✓ kernel first by vmacache_find whether the area to check the previous processing contain addresses that you want . In many cases , we are likely to operate in the same area several times in a row , so the system the last action of the cached area , speed up processing .
- ✓ Otherwise, red-black trees must be searched step by step. rb_node is a data structure used to represent each node in the tree. rb_entry is used to retrieve "useful data" from the node (here is an instance of vm_area_struct)
- ✓ The logic of searching for red-black trees is :
 -   end address of the current area is greater than the target address, start from the left child node
 -  the end address of the current area is less than or equal to the target address, start from the right child node
 - If the end address of a certain area is greater than the target address and the start address is less than the target address , the kernel has found an appropriate node and can exit the while loop and return a pointer to the vm_area_struct instance
 - If the child node of the root node of the tree is a NULL pointer , the kernel can easily determine when to end the search and return a NULL pointer as an error message
- ✓ If you find an appropriate area , then call vmacache_update cache to find the area , because the next time find_vma call to search the same area adjacent address high possibility

Region merge (vm_merge)

When a new area is added to the process's address space, the kernel checks whether it can be merged with one or more existing areas. `vm_merge` merges a new area with the surrounding area when possible. It requires many parameters.

Implementation file : `mm/mmap.c`

```
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 ,
                                     struct vm_userfaultfd_ctx vm_userfaultfd_ctx )
{
    pgoff_t pglen = ( end - addr ) >> PAGE_SHIFT ;
    struct vm_area_struct * area , * next ;

    ...

    return NULL ;
}
```

- ✓ `mm` is an instance of the address space of the related process, and `prev` is the area immediately before the new area
- ✓ `addr` , `end` and `vm_flags` are the start address, end address and flag of the new area respectively
- ✓ If the area belongs to a file mapping, `file` is a pointer to the file instance representing the file
 `pgoff` specifies the offset mapped in the file data
- ✓ `policy` parameter is only required on NUMA systems, I will not discuss it further

The technical details of the implementation are very simple. First check to determine whether the end address of the previous area corresponds to the start address of the new area.

If this is the case, the kernel must check the two areas next to confirm that the flags of the two are the same as the mapped file, and the offset within the file mapping meets the requirements of the continuous area. If the two files are mapped consecutively in the address space, but in the file The medium is not continuous and cannot be merged.

The check is completed by the `can_vma_merge_after` helper function. The work of merging the area with the previous area looks like this:

```
// mm/mmap.c: vma_merge

if ( prev && prev -> vm_end == addr &&
      mpol_equal ( vma_policy ( prev ) , policy ) &&
      can_vma_merge_after ( prev , vm_flags ,
                           anon_vma , file , pgoff ,
                           vm_userfaultfd_ctx ) ) {
    ...
}
```

If so, the kernel next checks whether the latter area can be merged :

```
// mm/mmap.c: vma_merge
```

```

/*
 * OK, it can. Can we now merge in the successor as well?
 */
if ( next && end == next -> vm_start &&
    mpol_equal ( policy , vma_policy ( next )) &&
    can_vma_merge_before ( next , vm_flags ,
                          anon_vma , file ,
                          pgoff + pglen ,
                          vm_userfaultfd_ctx ) &&
    is_mergeable_anon_vma ( prev -> anon_vma ,
                          next -> anon_vma , NULL )) {
    /* cases 1, 6 */
    err = vma_adjust ( prev , prev -> vm_start ,
                      next -> vm_end , prev -> vm_pgoff , NULL );
} else /* cases 2, 5, 7 */
    err = vma_adjust ( prev , prev -> vm_start ,
                      end , prev -> vm_pgoff , NULL );

```

Compared with the previous example, the first difference is that `can_vma_merge_before` is used to check whether the two regions can be merged, instead of `can_vma_merge_after`.

If both the previous and next regions can be merged with the current region, it must be confirmed that the anonymous mappings of the previous and next regions can be merged, and then a single region containing these 3 regions can be created.

In both cases, the auxiliary function `vma_adjust` is called to perform the final merge. It will appropriately modify all data structures involved, including `address_space->i_mmap` and `vm_area_struct` instances, as well as releasing structure instances that are no longer needed.

Area check (`get_unmapped_area`)

In the memory map before creating a new sub-region, the kernel virtual address space must ensure that there is enough free space, available for a given length of the region. The assignment to `get_unmapped_area` helper completed.

```

// mm/mmap.c

unsigned long
get_unmapped_area ( struct file * file , unsigned long addr , unsigned long len
,
                  unsigned long pgoff , unsigned long flags )
{
    unsigned long (* get_area ) ( struct file *, unsigned long ,
                                unsigned long , unsigned long , unsigned long );

    ...

    get_area = current -> mm -> get_unmapped_area ;
    if ( file && file -> f_op -> get_unmapped_area )

```

```

        get_area = file -> f_op -> get_unmapped_area ;
    addr = get_area ( file , addr , len , pgoff , flags );

    ...
}






```

The function parameters are self-explanatory and do n't explain much .

If the struct file itself provides get_unmapped_area , the function provided by the file will be used first .

Otherwise , the get_unmapped_area function provided by the current process will be used .

mm-> get_unmapped_area is initialized when it ? Recall " 3.3.1 Layout process address space : specific architecture Set content" as described , you can get the following conclusions :

- If the architecture-specific code implements get_unmapped_area , use that implementation .
-      architecture does not have it , use the kernel's default implementation :
 - If the layout of the memory-mapped area grows from low to high , use the arch_get_unmapped_area (mm/mmap.c) function
 - If the layout of the memory-mapped area grows from high to low , use the arch_get_unmapped_area_topdown (mm/mmap.c) function

Here , we take a detailed look at the standard function arch_get_unmapped_area used on most systems .

```

// mm/mmap.c

#ifndef HAVE_ARCH_UNMAPPED_AREA
unsigned long
arch_get_unmapped_area ( struct file * filp , unsigned long addr ,
                        unsigned long len , unsigned long pgoff , unsigned long flags )
{
    struct mm_struct * mm = current -> mm ;
    struct vm_area_struct * vma ;
    struct vm_unmapped_area_info info ;

    if ( len > TASK_SIZE - mmap_min_addr )
        return - ENOMEM ;

    if ( flags & MAP_FIXED )
        return addr ;

    if ( addr ) {
        addr = PAGE_ALIGN ( addr );
        vma = find_vma ( mm , addr );
        if ( TASK_SIZE - len >= addr && addr >= mmap_min_addr &&
            (! vma || addr + len <= vma -> vm_start ))
            return addr ;
    }

    info . the_flags = 0 ;

```

```

    info . length = len ;
    info . low_limit = mm -> mmap_base ;
    info . high_limit = TASK_SIZE ;
    info . align_mask = 0 ;
    return vm_unmapped_area (& info ) ;
}
#endif

```

- ✓ First, you must check whether the MAP_FIXED flag is set , which indicates that the mapping will be created at a fixed address . If so , return the address directly
- ✓ call find_vma function , if an end address is greater than found addr area , the vma is not NULL, this time further checks whether the start address of this area is also larger than addr + len, if established , proven to find a hole , can Here , a subregion with a length of len is created , and the existing vma will not be disturbed.
If find_vma does not find a suitable subregion , that is, vma == NULL, this means that the end address of all existing subregions is less than addr, add is in the edge area , and the back is desert , so you can create a length of len here. Sub-area .
- ✓ If it still fails , vm_unmapped_area will be called . Note that when vm_unmapped_area is called , the addr parameter is not passed in , which proves that the return value of vm_unmapped_area has nothing to do with addr .
In other words , vm_unmapped_area purpose are trying to find an area mapped from memory length len cavity , if found , return the empty start address , this address may be associated with a user requested address addr is not the same .

The implementation details of vm_unmapped_area seem to be very complicated , so we won't take a closer look

Region split (split_vma)

split_vma is an auxiliary function , it will directly call __split_vma to achieve the function .

Implemented logical functions is very simple , just familiar with the standard operation of a data structure , it allocates a new region for the new split out vm_area_struct example , fill it with the original data area , and the boundary alignment . Newly inserted area Into the red-black tree of the process .

The code is not analyzed in detail , we just simply post the declaration of __split_vma :

```

// mm/mmap.c

static int __split_vma ( struct mm_struct * mm , struct vm_area_struct * vma ,
                        unsigned long addr , int new_below )

```

APIs

We are already familiar with the principles , data structures and related operations related to memory mapping . In this section , we will further discuss the related APIs for establishing memory mapping . User space code can use these APIs for memory mapping . We focus on establishing the mapping Time the interaction between the kernel and the application .

Create mapping (mmap , do_mmap)

As far as we know , the C standard library provides the mmap function to establish a mapping . Its definition is as follows :

```

#include <sys/mman.h>

```

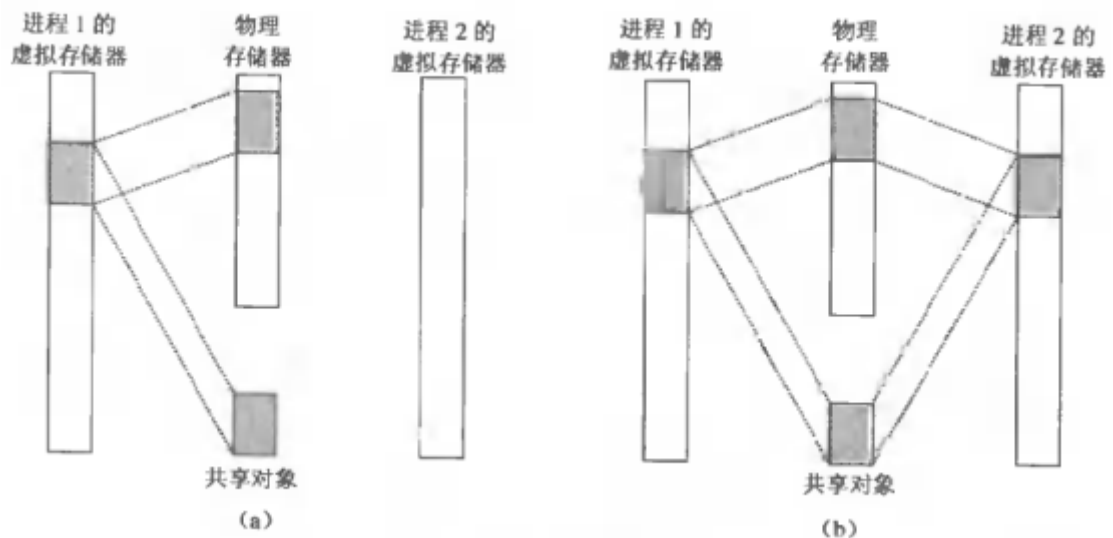
```
void * mmap ( void * start , size_t length , int prot , int flags , int fd ,
off_t off set );
```

The parameters are explained as follows :

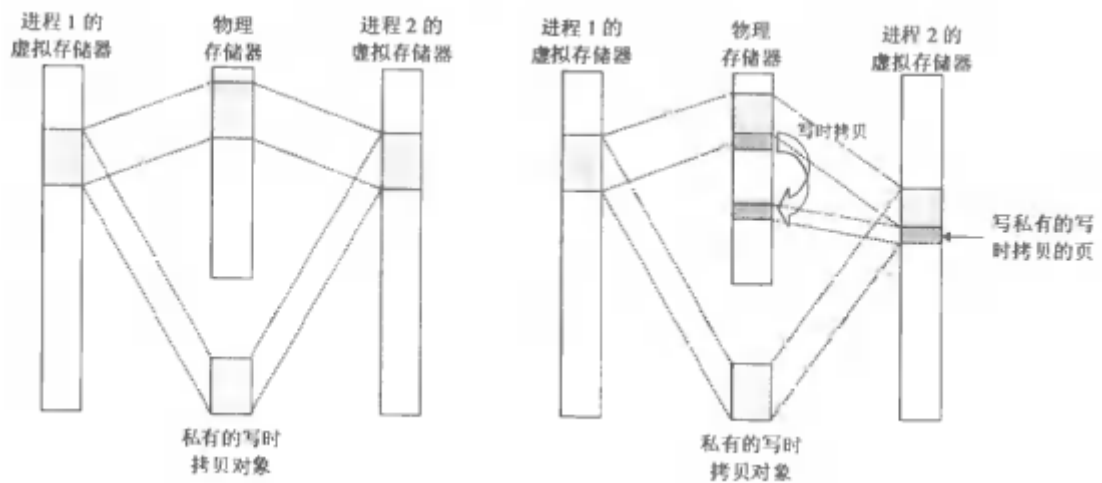
- ✓ start: specifying the desired map to which the virtual address , usually set to NULL, on behalf of the system automatically selected address , return address after the successful mapping . In addition, even if start is not NULL, the return value may also start different addresses pointing , reason For details, please refer to " 3.3.4 get_unmapped_area function"
- ✓ length: representative length mapping , which means that a length of the memory map region length subregions , means how much the map file to memory-mapped region portion
- ✓ prot: Corresponds to vm_area_struct-> **vm_page_prot** , representing the access rights of the newly created memory mapped sub-area
 - ◆ PROT_EXEC : sub- area can be executed
 - ◆ PROT_READ : The sub- area can be read
 - ◆ PROT_WRITE : The sub- area can be written
 - ◆ PROT_NONE : The sub- area cannot be accessed
 - ◆ ...
- ✓ flags: Various characteristics that affect the mapping area :

When you create a map , MAP_SHARED / MAP_PRIVATE must choose one , that is to say either create a shared mapping , or create private maps .

- MAP_SHARED : Create a shared mapping . The so-called sharing means that if multiple processes map the same area in the same file , there will only be one copy in physical memory, and any process write operation can be seen by other processes . and write modify eventually written back to the original file to disk in . One of the benefits of doing so is to save physical memory . example in the following figure :



- MAP_PRIVATE : creating a private map , generate a write operation is mapped area private of " copy on write " (Copy ON the Write , that is, make a copy of the physical memory at the time of writing), written between processes The input operations are invisible to each other , and any modification made to this area will not be written back to the original file on the disk . The example diagram is as follows :



- **MAP_FIXED** : If the address pointed to by the parameter start fails to successfully establish the mapping , the mapping will be abandoned , and the available address will be automatically searched for if it is wrong . The use of this flag is generally discouraged .
 - **MAP_ANONYMOUS** : An anonymous mapping is established . At this time, the parameter fd is ignored , files are not involved , and the mapping area cannot be shared with other processes . The difference between this mapping and file mapping is that when a page fault occurs , the file mapping will be removed from the file. Read the memory and fill the physical page ; the anonymous mapping is to fill the physical page with 0 .
 - **MAP_LOCKED** : Lock the mapped area , which means that the area will not be replaced (swap)
 - ...
- ✓ **fd**: to be mapped into memory file descriptor . If you use an anonymous memory mapping , namely flags set in **MAP_ANONYMOUS** , fd is set to -1 .
- Some systems do not support anonymous memory mapping , you can use **fopen** to open the **/dev/zero** file , and then map the file , which can also achieve the effect of anonymous memory mapping .
- fd** can point to an ordinary file , can also point to a block or character device equipment .
- If **fd** pointing device is a character , e.g. LCD device driver node provides , for such **fd** using **mmap** must be required to achieve a corresponding drive **.mmap** function . Ordinary file mapping is different , for such **fd** of mapping the physical memory is allocated when creating maps and build page tables (by the driver in **.mmap** completion , rather than be handled by the missing pages abnormal program system)
- ✓ **offset** : specifies the offset of the file mapping . This value is used when only part of the content of the file is mapped (if the entire file is mapped , the offset is 0) . Offset must be an integer multiple of **PAGE_SIZE** .

The **mmap** function of the user layer will eventually call the system call provided by the kernel to complete the actual function . In the ARM architecture , this system call function is **sys_mmap_pgoff** .

```
// mm/mmap.c

SYSCALL_DEFINE6 ( mmap_pgoff , unsigned long , addr , unsigned long , len ,
                unsigned long , prot , unsigned long , flags ,
                unsigned long , fd , unsigned long , pgoff )
```

sys_mmap_pgoff -> **vm_mmap_pgoff** -> **do_mmap_pgoff** -> **do_mmap**

do_mmap perform the actual operation , this function is also defined in **mm / mmap.c** of :

```
// mm/mmap.c

/*
 * The caller must hold down_write(&t->mm->mmap_sem) .
```


```

*/
unsigned long do_mmap ( struct file * file , unsigned long addr ,
                        unsigned long len , unsigned long prot ,
                        unsigned long flags , vm_flags_t vm_flags ,
                        unsigned long pgoff , unsigned long * populate )

```

The code flow is roughly as follows :

do_mmap

- get_unmapped_area
-  sign
- mmap_region
 - ✓ find_vma_links & do_munmap
 - ✓ accountable_mapping
 - ✓ vma_merge
 - ✓ Create a new vm_area_struct instance
 - ◆ kmem_cache_zalloc
 - ◆ if (file != NULL) file->f_op->mmap
 - ◆ else shmem_zero_setup
 - ◆ vma_link
 - ✓ Return the starting address of the mapping
- If the VM_LOCKED flag is set , it will cause mm_populate to be called

do_mmap used to be one of the longest functions in the kernel. It has now been divided into two parts, but it is still quite long. One part needs to thoroughly check the parameters passed by the user application, and the second part needs to consider a large number of special circumstances and subtleties. Since the latter part is of little value for understanding the mechanism involved in a general sense, we only examine a representative standard situation: mapping ordinary files with MAP_SHARED . In addition, in order to avoid making the description too lengthy, the code flow chart has also been deleted accordingly.

First of all , before calling the paper describes get_unmapped_area function , to find an appropriate address in the virtual address space used to create the sub-region . We know that , the application can specify a fixed address mapping, address or address a recommendation by the kernel choice .

Then , calculate the flag , the code is as follows :

```

// mm/mmap.c: do_mmap

vm_flags |= calc_vm_prot_bits ( prot ) | calc_vm_flag_bits ( flags ) |
mm -> def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC ;

```

calc_vm_prot_bits and calc_vm_flag_bits the user passed over the space of constant sign, and access flag into one common focus , relatively easy to process in subsequent operations (Map_ and PROT_ AMI prefix VM_ flag)

Interestingly , the kernel includes def_flags in the flag set after obtaining def_flags from the mm_struct instance of the currently running process . The value of def_flags is 0 or VM_LOCK . The former will not change the result flag set, and VM_LOCK means that subsequent mapped pages cannot be swapped out. To set the value of def_flags , the process must issue the mlockall system call, using the above mechanism to prevent all future mappings from being swapped out, even if the VM_LOCK flag is not explicitly specified at the time of creation .

After checking the parameters and setting all the required flags , the remaining work is delegated to mmap_region.

When the appropriate starting address is obtained through `get_unmapped_area`, `mmap_region` first calls `find_vma_links` to check whether `[addr, addr + length]` has been included in an existing mapping area? If so, call `do_munmap(mm, addr, len)` from the existing mapping area. The mapping area is deleted `[addr, addr + length]` of this area, attention at this time may involve splitting area. `do_munmap` will be detailed later.

Then `mmap_region` call `accountable_mapping` check memory limitations, if no `MAP_NORESERVE` flag or kernel parameters `sysctl_overcommit_memory` set `OVERCOMMIT_NEVER` (i.e., does not allow excess), is called `security_vm_enough_memory_mm`. This function is required to select whether the memory allocation operation. If it is not selected `Allocate`, the system call ends and `-ENOMEM` is returned.

`sysctl_overcommit_memory` can be set with the help of `/proc/sys/vm/overcommit_memory`. There are currently 3 overuse options. 1 Allows the application to allocate as much memory as required, even if it exceeds the limit allowed by the system address space.

0 means that the application heuristic is overused. The number of available pages is obtained by calculating the total number of page caches, swap areas, and unused page frames, and requests to allocate a small number of pages are allowed.

2 means strict mode, which is called strict overcommitment, in which the number of pages allowed to be allocated is calculated as follows:

`allowed = (totalram_pages - hugetlb) * sysctl_overcommit_ratio / 100;`

`allowed += total_swap_pages;`

Here `sysctl_overcommit_ratio` is a configurable kernel parameter, usually set to 50. If the total number of used pages exceeds all

According to the calculation result, the kernel refuses to continue allocating memory.

Does it make sense to allow an application to allocate more pages than the theoretical processing limit? Scientific computing applications sometimes require this feature. Some applications tend to allocate a lot of memory, which does not actually need to be used, but from the perspective of the application author, it is always good to allocate more, just in case! If the memory is never used, then physical page frames will not be allocated and there is no problem. Obviously this programming style is a bad practice, but unfortunately, this is not a criterion for evaluating the value of software. In the scientific community outside of computer science, writing clean code usually does not bring rewards. In related fields, it is usually only concerned that the program can work normally under a given configuration, and the work that makes the program still usable or portable in the future does not seem to provide the immediate benefits, so it is usually considered to be of no value.

Next `mmap_region` calls `vma_merge` to see if the newly created subregion can be merged with an existing region. If so, then only adjust the relevant fields of the `vm_area_struct` of this existing region, without creating a new `vm_area_struct` instance.

If not merged, then `kmem_cache_zalloc` assign a new `vm_area_struct` instance, and to perform the initialization operation:

- ✓ initialization `vm_area_struct` the fields
- ✓ If it is a normal file mapping (ie `file != NULL`), call `file->f_op->mmap`, and most file systems assign `f_op->mmap` to `generic_file_mmap`.
`generic_file_mmap` main work is to the new `vm_area_struct` examples of `vm_ops` member to `generic_file_vm_ops`, "3.3.4 memory-mapped file region and address space associated" section introduced `generic_file_vm_ops`, the key element is the `filemap_fault`.
The main purpose of `filemap_fault` is to read the data in the file, and I will not go into details here.
- ✓ If the `file == NULL`, then one anonymous mapping, this time calls `shmem_zero_setup`, will file pointing to `/dev/zero`
- ✓ Finally, call `vma_link` to add the newly created `vm_area_struct` instance to the red-black tree of `mm_struct`

After the new instance is successfully created, the starting virtual address of this subarea is returned.

At the end of `do_mmap` , if `vm_flags & VM_LOCKED` is not 0, `populate` will be assigned to `len`

```
addr = mmap_region(file, addr, len, vm_flags, pgoff);
if ( !IS_ERR_VALUE(addr) &&
    ((vm_flags & VM_LOCKED) ||
    (flags & (MAP_POPULATE | MAP_NONBLOCK)) == MAP_POPULATE) )
    *populate = len;
return addr;
} ? end do mmap ?
```

`vm_mmap_pgoff` will check if `populate` is set in `do_mmap` , and if it is set , `mm_populate` will be called

```
// mm/util.c

unsigned long vm_mmap_pgoff ( struct file * file , unsigned long addr ,
    unsigned long len , unsigned long prot ,
    unsigned long flag , unsigned long pgoff )
{
    ...

    ret = do_mmap_pgoff ( file , addr , len , prot , flag , pgoff ,
        & populate ); //do_mmap_pgoff directly calls do_mmap

    if ( populate )
        mm_populate ( ret , populate );

    ...
}
```

The purpose of this above logic is : If you set `VM_LOCKED` , either sign parameters through a system call explicitly passed in , or through `mlockall` mechanism implicitly set , the kernel will call `mm_populate` sequentially scanned map pages , each page is triggered Page faults are abnormal in order to read its data . Of course , this means that the performance improvement brought by delayed reads is lost , but the kernel can ensure that the pages involved are always in physical memory after the mapping is established . After all, the `VM_LOCKED` flag is used to prevent Pages are swapped out from memory , so these pages must be read in first .

Delete mapping (`munmap`, `do_munmap`)

If the user code wants to delete the mapping established by `mmap` , such as calling `munmap`, its definition is as follows :

```
#include <sys/mman.h>

int munmap ( void * addr , size_t len );
```

It requires two parameters : the start address and length of the unmapped area . Subsequent access to the deleted area will cause a segmentation fault .

The user layer `munmap` function will eventually call the system call provided by the kernel to complete the actual function . In the ARM architecture , this system call function is `sys_munmap` .

```
// mm/mmap.c

SYSCALL_DEFINE2 ( munmap , unsigned long , addr , size_t , len )
{
    profile_munmap ( addr );
    return vm_munmap ( addr , len );
}
```

```
}
```

sys_munmap will call vm_munmap, which directly calls do_munmap.

do_munmap perform the actual operation , this function is also defined in mm / mmap.c of :

```
// mm/mmap.c

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

The code flow is roughly as follows :

do_munmap

- find_vma
- need to split it -?> __Split_vma
- find_vma again
- also necessary to once again split it -?> __Split_vma
- detach_vmas_to_be_unmapped
- unmap_region
- remove_vma_list

First , call find_vma to find the vm_area_struct instance corresponding to the given address .

If (the starting address of the instance vma_area_struct->start) is greater than (the given starting address start), the area must be split , and the area must be split into [vma_area_struct->start, start – 1] and [start, vma_area_struct->end] These two areas .

After calling again split find_vma, looking for a given address with the corresponding vma_area_struct example , the start address at this time to find an example of a given constant equal to the address . In this case the check (the end address of the instance vma_area_struct-> end) whether is greater than (the given start address start + len), if so , the need to split again , the area is split into [vma_area_struct-> start, start + end -1] and [start + end, vma_area_struct-> end] These two areas .

After the above split operation , we will be able to find a vma_area_struct example , a starting address of start, end address for the start + len. And then call for that instance detach_vmas_to_be_unmapped .

Detach_vmas_to_be_unmapped will delete the instance from the red-black tree and linear linked list of the process .

There are two final steps . First, call unmap_region to delete all items related to the mapping from the page table . After completion , the kernel must also ensure that the related items are removed from the TLB or invalidated .

Secondly , with remove_vma_list release vm_area_struct space occupied instance , to complete the removal of the mapping from the kernel work .

Non-linear mapping

According to the above description, ordinary mapping maps a continuous part of the file to an equally continuous part of the virtual memory. If you need to map different parts of the file to a contiguous area of virtual memory in different orders, you usually have to use several mappings. From the point of view of the resources consumed by the mapping , the cost is more expensive (especially the number of vm_area_structs that need to be allocated) . An easy way to achieve the same effect is to use non-linear mapping, which was introduced during the development of kernel version 2.5 . The kernel provides an independent system call (remap_file_pages) specifically for this purpose.

However, in 4.1 the kernel , this feature has been canceled , because not many people will use this feature , but this feature will make virtual memory management code becomes a bit confusing , and will take PTE low-order . So the latest The kernel code has cancelled this feature . For detailed reasons, please see < Documentation/vm/remap_file_pages.txt >

If readers are still interested in nonlinear mapping , please read "In-depth Linux Kernel Architecture : 4.7.3 Nonlinear Mapping"

.mmap some thoughts of [@ 2018.12.24]

After the mmap function is called from the user space , there are three things that the kernel will ultimately accomplish :

- 1) Find an area in the virtual address space of the process (struct vm_area_struct)
- 2) Allocate a piece of memory from physical memory
- 3) Modify the page table to establish a mapping from physical memory to virtual address space

One thing 1 is done by the kernel for us (refer to "Create Mapping") . When the kernel finds the space area , it will call the mmap function implemented by the underlying driver (file->f_op->mmap) , and put the struct representing this area vm_area_struct as a parameter passed to mmap.

There are many ways for the mmap function of the underlying driver . Here are a few ways that have been seen :

- a) One is to implement the functions defined in vm_operations_struct in the mmap function (mainly the fault function) , and then assign vm_operations_struct to vm_area_struct . When the user space accesses the corresponding virtual address , the kernel will trigger a page fault exception , which will eventually be called here the vm_operations_struct .fault . in fault function , we can allocate physical memory page table to establish and modify the map . currently file mmap operation is used in this way (generic_file_vm_ops (<https://elixir.bootlin.com/linux/v4.20/source/mm/filemap.c#L2631>)) .
(<https://elixir.bootlin.com/linux/v4.20/source/mm/filemap.c#L2631>)
- b) The second is mmap distribution inside physical content and build a map , so that you do not need to achieve vm_operations_struct up , because the mapping has been created , no missing pages abnormalities . At present some of the involved DMA access hardware drivers are like this Do it , (code sample not found ...) .
- c) The third way is to mmap before the function is called , in other places driven allocate physical memory (for example during driver initialization , or some later time) , then mmap only responsible for modifying the page table inside a mapping function . At present, many places are doing , such as sound inside snd_pcm_lib_default_mmap (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L3356) , or LCD inside fb_mmap (<https://elixir.bootlin.com/linux/v4.20/source/drivers/video/fbdev/core/fbmem.c#L1379>) .
(https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L3356)
(<https://elixir.bootlin.com/linux/v4.20/source/drivers/video/fbdev/core/fbmem.c#L1379>)

3.3.5 Heap management

The heap is a memory area used to dynamically allocate variables and data in a process, and the management of the heap is not directly visible to the application programmer. Because it relies on various auxiliary functions provided by the standard library (the most important of which is malloc) to allocate an arbitrary length of memory area. The classic interface between malloc and the kernel is the brk system call, which is responsible for expanding / shrinking the heap.

The heap is a continuous memory area that grows from bottom to top as it expands. The `mm_struct` structure mentioned above contains the start and current end addresses (`start_brk` and `brk`) of the heap in the virtual address space .

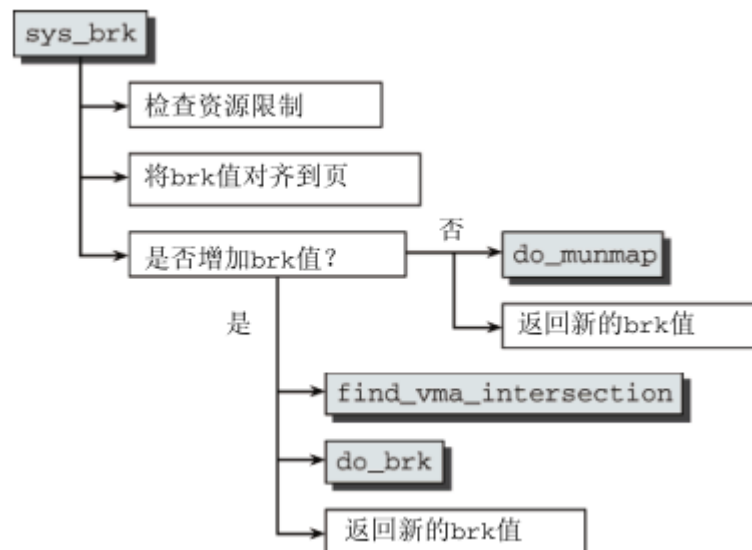
The definition of the `brk` system call is as follows :

```
// mm/mmap.c

SYSCALL_DEFINE1 ( brk , unsigned long , brk )
```

The `brk` system call requires only one parameter, which is used to specify the new end address of the heap in the virtual address space (if the heap is about to shrink, it is less than the current value) .

The code flow is roughly as follows :



The `brk` mechanism is not an independent kernel concept, but is implemented based on anonymous mapping to reduce internal overhead. Therefore, many of the functions discussed in the previous sections for managing memory mapping can be reused when implementing `sys_brk` .

After checking that the new address used as the `brk` value does not exceed the limit of the heap, the first important operation of `sys_brk` is to align the requested address according to the page length.

```
// mm/mmap.c: SYSCALL_DEFINE1(brk, unsigned long, brk)

newbrk = PAGE_ALIGN ( brk );
oldbrk = PAGE_ALIGN ( mm -> brk );
```

The code ensures `brk` new value (original value also) system is a multiple page length . In other words , one is `brk` minimum memory region can be allocated blocks . Therefore, the user needs to another dispenser function space , the pages split into smaller areas . this is is the C standard library `malloc` main tasks .

`Do_munmap` will be called when the heap needs to be shrunk , and we are already familiar with this function in the "Delete Mapping" section .

```
// mm/mmap.c: SYSCALL_DEFINE1(brk, unsigned long, brk)

/* Always allow shrinking brk. */
```

```

    if ( brk <= mm -> brk ) {
        if (! do_munmap ( mm , newbrk , oldbrk - newbrk ))
            goto set_brk ;
        goto out ;
    }

```

If the heap will be expanded , the kernel must first check the maximum heap size limit if the new length exceeds the process . Find_vma_intersection is used to do this thing , it will check the expansion of the heap whether the process in the existing maps overlap . If so , then Do nothing and return immediately .

```

// mm/mmap.c: SYSCALL_DEFINE1(brk, unsigned long, brk)

/* Check against existing mmap mappings. */
if ( find_vma_intersection ( mm , oldbrk , newbrk + PAGE_SIZE ))
    goto out ;

```

After checking , entrust the actual work of expanding the heap to do_brk . The function always returns the new value of mm->brk , no matter whether it is increased, decreased, or unchanged compared with the original value.

```

// mm/mmap.c: SYSCALL_DEFINE1(brk, unsigned long, brk)

/* Ok, looks good - let it rip. */
if ( do_brk ( oldbrk , newbrk - oldbrk ) != oldbrk )
    goto out ;

```

We don't need to discuss do_brk separately , because in essence it is a simplified version of do_mmap , nothing new . Similar to the latter , it creates an anonymous mapping in the user address space , but saves some security checks and improves the code Performance of the handling of special situations .

If everything goes well , it will enter set_brk.

```

// mm/mmap.c: SYSCALL_DEFINE1(brk, unsigned long, brk)

set_brk :
    mm -> brk = brk ;
    populate = newbrk > oldbrk && ( mm -> def_flags & VM_LOCKED ) != 0 ;
    up_write (& mm -> mmap_sem );
    if ( populate )
        mm_populate ( oldbrk , newbrk - oldbrk );
    return brk ;

```

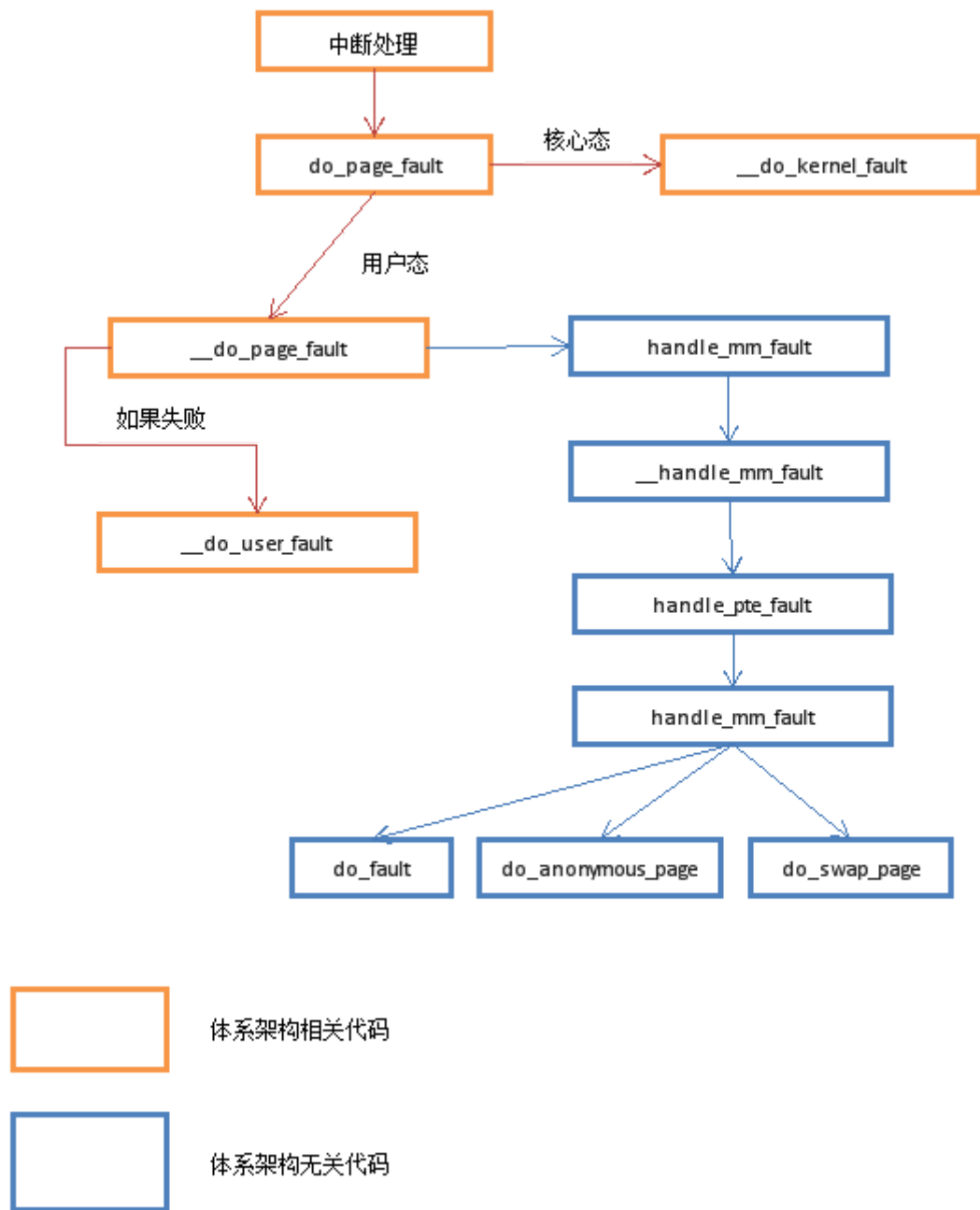
Note , set_brk last call mm_populate, we introduced one in the function "create a map" , the function will turn scanned maps of each page , each page missing page trigger an exception to read its data . This shows that when sys_brk upon successful return , the kernel has applied for a memory page and memory is initialized to the partner system 0 a , page table will be created at this time .

3.4 Handling of page fault exception

Before the data in a certain virtual memory area is actually needed, the association between virtual and physical memory will not be established. If the part of the virtual address space accessed by the process is not yet associated with the page frame, the processor automatically raises a page fault exception, and the kernel must

handle this exception. This is one of the most important and complex aspects of memory management , because countless details must be considered.

The following figure outlines the entire process of page fault processing , and we will introduce the details of each part in turn :



- First , the hardware will trigger abnormal missing page , then branches to the interrupt vector table , then eventually call the interrupt vector table to do_page_fault
- do_page_fault is roughly divided into two parts :
 - First, it will check whether mm_struct is NULL, if it is , it proves that the exception may be triggered by a kernel interrupt ; or if !user_mode(regs) is true , it proves that it is in the kernel mode at this time . At this time, __do_kernel_fault is called
 - it is user mode , that is, mm_struct != NULL, __do_page_fault will be called , and the latter will further call architecture-independent code to deal with page fault exceptions . Here, according to different situations , one of the 3 of do_fault/ do_anonymous_page / do_swap_page will be finally called One .

- If `do_page_fault` is successful , the system has allocated physical memory , and has also written initial data into the memory and established a page table . At this time, the system will re-execute the instruction that triggered the page fault exception .

If `do_page_fault` fails , it will call `__do_user_fault` , the main purpose is sent to the user process `SIGSEGV` and other signals , thereby ending user processes . This logic is simple , later this analysis is not a function .

3.4.1 Interrupt handling process

This section will not focus on the principle of interrupt handling in the Linux kernel (a special article will be published on the principle of interruption) , here is only a rough list of the flow of page fault exception interruption under the ARM system architecture .

First , the hardware triggers Prefetch Abort, and then the hardware will point the PC to the `vector_pabt` of the interrupt vector table .

The definition of the ARM architecture interrupt vector table is as follows :

```
//arch/arm/kernel/entry-armv.S

__vectors_start :
    W ( b )      vector_rst
    W ( b )      vector_und
    W ( ldr )    pc , __vectors_start + 0x1000
    W ( b )      vector_pabt
    W ( b )      vector_dabt
    W ( b )      vector_addrexcptn
    W ( b )      vector_irq
    W ( b )      vector_fiq
```

`vector_pabt` achieved also with a compilation document , the macro " `vector_stub pABT` " definition , eventually calls `__pabt_usr -> pabt_helper -> CPU_PABORT_HANDLER`

The `CPU_PABORT_HANDLER` is as follows :

```
//arch/arm/include/asm/glue-pf.h

...

#ifdef CONFIG_CPU_PABRT_V7
# ifdef CPU_PABORT_HANDLER
#  define MULTI_PABORT 1
# else
#  define CPU_PABORT_HANDLER v7_pabort
# endif
#endif

...
```

According to different architecture versions , different implementations are selected . If the CPU is ARMv7 architecture , it corresponds to `v7_pabort`.

The implementation of `v7_pabort` is `arch/arm/mm/pabort-v7.S` , it will directly call `do_PrefetchAbort` . The implementation of the latter is as follows :

```
//arch/arm/mm/fault.c

asmlinkage void __exception
do_PrefetchAbort ( unsigned long addr , unsigned int ifsr , struct pt_regs *
regs )
{
    const struct fsr_info * inf = ifsr_info + fsr_fs ( ifsr );
    struct siginfo info ;

    if ( ! inf -> fn ( addr , ifsr | FSR_LNX_PF , regs ) )
        return ;

    ...
}
```

`ifsr_info` is actually an array , first by `ifsr` out the corresponding array item from this array , and then call `inf->fn`. That this `fn` who in the end is it ?

Look at the following implementation :

```
//arch/arm/mm/fault.c

struct fsr_info {
    int (* fn ) ( unsigned long addr , unsigned int fsr , struct pt_regs *
regs );
    int      sig ;
    int      code ;
    const char * name ;
};

/* FSR definition */
#ifdef CONFIG_ARM_LPAE
#include "fsr-3level.c"
#else
#include "fsr-2level.c"
#endif

//arch/arm/mm/fsr-2level.c

static struct fsr_info ifsr_info [] = {
    { do_bad , SIGBUS , 0 , "unknown 0" },

    ...

    { do_page_fault , SIGSEGV , SEGV_MAPERR , "page translation fault" },
}
```

```
...
}
```

A structure `fsr_info` is defined in `fault.c`, and an array `ifsr_info` of this structure is defined and initialized in `fsr-2level.c`. As you can see, one of the elements corresponds to the function `do_page_fault`.

As for why you can call to `do_page_fault` corresponding array item, I did not look, there is not much to say, anyway, know the final call to the interrupt handling `do_page_fault` on the line.

3.4.2 do_page_fault

The implementation of `do_page_fault` is as follows. In order to simplify, we have omitted a lot of parts, leaving only the backbone:

```
//arch/arm/mm/fault.c

static int __kprobes
do_page_fault ( unsigned long addr , unsigned int fsr , struct pt_regs * regs )
{
...

    /*
     * If we're in an interrupt or have no user
     * context, we must not take the fault..
     */
    if ( in_atomic () || ! mm )
        goto no_context ;

...

    /*
     * As per x86, we may deadlock here. However, since the kernel only
     * validly references user space from well defined areas of the code,
     * we can bug out early if this is from code which shouldn't.
     */
    if ( ! down_read_trylock ( & mm -> mmap_sem ) ) {
        if ( ! user_mode ( regs ) && ! search_exception_tables ( regs ->
ARM_pc ) )
            goto no_context ;

...

    } else {

...

    }

    fault = __do_page_fault ( mm , addr , fsr , flags , tsk );

...

    /*
     * Handle the "normal" case first - VM_FAULT_MAJOR / VM_FAULT_MINOR
     */
    if ( likely ( !( fault & ( VM_FAULT_ERROR | VM_FAULT_BADMAP |
VM_FAULT_BADACCESS ) ) ) )
```

```

        return 0 ;

/*
 * If we are in kernel mode at this point, we
 * have no context to handle this fault with.
 */
if (! user_mode ( regs ))
    goto no_context ;

...

if ( fault & VM_FAULT_SIGBUS ) {
    /*
     * We had some memory, but were unable to
     * successfully fix up this page fault.
     */
    sig = SIGBUS ;
    code = BUS_ADRERR ;

} else {
    /*
     * Something tried to access memory that
     * isn't in our memory map..
     */
    sig = SIGSEGV ;
    code = fault == VM_FAULT_BADACCESS ?
           SEGV_ACCERR : SEGV_MAPERR ;

}

__do_user_fault ( tsk , addr , fsr , sig , code , regs );
return 0 ;

no_context :
    __do_kernel_fault ( mm , addr , fsr , regs );
    return 0 ;
}

```

- First , note 3 Chu goto no_context statement , summed up if in_atomic () || mm_struct == NULL || ! User_mode (regs) will enter __do_kernel_fault .
- Secondly , __do_page_fault responsible for user process page fault processing , note that the return value fault: If the fault is not wrong , then return 0, represents the treatment was successful ; otherwise it will be based on fault specific value , call __do_user_fault , sent to the user process SIGBUS / SIGSEGV signal .

3.4.3 Kernel page fault exception handling (__do_kernel_fault)

According to the introduction in the previous section , the following situations may cause the kernel page fault exception :

- If the exception address is accessed in the atomic context , in_atomic() == true at this time .

- If the exception address is accessed in the interrupt context or kernel thread , `mm_struct == NULL` at this time

Note, for kernel threads , please refer to "In-depth Linux Kernel Architecture Section 1.3.3 "

- If the user process calls the system call interface provided by the kernel , but the process passes an incorrect address . In this case, it will first try to be handled by `__do_page_fault` , if it is unsuccessful , it will call `__do_kernel_fault`.

The `__do_kernel_fault` code looks relatively simple :

```
//arch/arm/mm/fault.c

static void
__do_kernel_fault ( struct mm_struct * mm , unsigned long addr , unsigned int
fsr ,
                    struct pt_regs * regs )
{
    /*
     * Are we prepared to handle this kernel fault?
     */
    if ( fixup_exception ( regs ))
        return ;

    /*
     * No handler, we'll have to terminate things with extreme prejudice.
     */
    bust_spinlocks ( 1 );
    pr_alert ( " Unable to handle kernel %s at virtual address %08lx\n" ,
              ( addr < PAGE_SIZE ) ? "NULL pointer dereference" :
              "Paging request" , addr );

    show_pte ( mm , addr );
    die ( " Oops " , regs , fsr );
    bust_spinlocks ( 0 );
    do_exit ( SIGKILL );
}
```

- First, we will try to call `fixup_exception` to handle the exception . This is the last chance ! We do not plan to go deep into the `fixup_exception` mechanism for the time being , and those who are interested can read the code by themselves .

➤👉🔍 ♦️🌀 last chance fails , the kernel will print the Oops information we often see ...

3.4.4 User process page fault exception handling (`__do_page_fault`)

`__do_page_fault` will first find the corresponding mapping area through the abnormal address . If the mapping area does not exist , it means that this is an invalid address and will return a `VM_FAULT_BADMAP` error . Then it will check the access permissions of the mapping area (READ/WRITE/EXEC). If the permissions are incorrect , it means It is an illegal access , and `VM_FAULT_BADACCESS` error will be returned at this time .

After the above checks are passed , `handle_mm_fault` will be called for processing .

`handle_mm_fault` not dependent on the underlying architecture , but in the frame memory management system implemented independently . This function is the confirmation page directory levels , leading to the respective page directory entry corresponding to the page table entry address are abnormal Exist , and then call the `handle_pte_fault` function to analyze the cause of the page fault exception .

The main logic of `handle_pte_fault` is as follows :

```
//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 ;
    barrier () ;
    if ( ! pte_present ( entry ) ) {
        if ( pte_none ( entry ) ) {
            if ( vma -> vm_ops ) {
                if ( likely ( vma -> vm_ops -> fault ) )
                    return do_fault ( mm , vma , address ,
pte ,
                                     pmd , flags , entry ) ;
            }
            return do_anonymous_page ( mm , vma , address ,
                                     pte , pmd , flags ) ;
        }
        return do_swap_page ( mm , vma , address ,
                              pte , pmd , flags , entry ) ;
    }

    if ( flags & FAULT_FLAG_WRITE ) {
        if ( ! pte_write ( entry ) )
            return do_wp_page ( mm , vma , address ,
                              pte , pmd , ptl , entry ) ;
        entry = pte_mkdirty ( entry ) ;
    }

    ...
}
```

The parameter `pte` is a pointer to the relevant page table entry (`pte_t`) .

If the page is not in physical memory , i.e. `! pte_present(entry)` , you must distinguish the following three cases :

- If there is no corresponding page table entry (the `p-TE_none == to true`) , description at this time only to establish a mapping between the region and memory-mapped files , not physical memory allocation and create a page table .

- If `vma->ops->fault` is not empty , then call `do_fault` process
 - Otherwise, call `do_anonymous_page` to process
- If the corresponding page table entry exists (`pte_none == true`), but the corresponding page is not in physical memory , it means that the page has been swapped out , because `do_swap_page` needs to be called to swap in the page table from a swap area of the system .

If the page has physical memory , then if (`!pte_present(entry)`) is not established , the above `do_xxx` operations will not be executed . The reason for this is that the mapping area grants write permission to the page , but the page table indicates that it cannot Writing to this page , so an exception was triggered .

This case by the `do_wp_page` handles , `do_wp_page` responsible for creating the copy of the page , and modify the corresponding page table entry process , to point to the new physical page , and then enable write access to the page table entry for the new physical page . The mechanism called copy-on-write (copy ON the write , referred COW).

The most common occasion of the COW mechanism is when the process creates a child process (`fork`), the physical page required by the child process is not copied immediately , but the physical page of the parent process is mapped to the address space of the child process as a "read-only" "Copy" , so as not to spend too much time copying information . Before the actual write access of the child process , no independent copy of the page will be created for the process .

In addition , when creating a memory map , if it is a private map , the COW mechanism will also be used . For details , see 3.3.4 ``Create a Map" .

do_fault

According to an analysis on , into `do_fault` description `vma->ops->fault` is not empty , further speaking , it is mapping must be linked to a file .

Recall " 3.3.4 Create a map" details of : If you are mapping a regular file , whether it is private or shared maps , will be the general document file link ; if it is anonymous & share mapping , will be with / `dev` / zero linked file ; but if it is anonymous & private map , is not linked to any file , this case will be `do_anonymous_page` process , not described in this section of the column .

Let 's take a look at the details of `do_fault` :

```
//mm/memory.c

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 ;

    pte_unmap ( page_table );
    if (!( flags & FAULT_FLAG_WRITE ))
        return do_read_fault ( mm , vma , address , pmd , pgoff , flags
    ,
                                orig_pte );
    if (!( vma -> vm_flags & VM_SHARED ))
        return do_cow_fault ( mm , vma , address , pmd , pgoff , flags
    ,
```



```

                                orig_pte );
    return do_shared_fault ( mm , vma , address , pmd , pgoff , flags ,
orig_pte );
}

```

do_fault will call 3 different processing functions, these 3 common functions that will be called __do_fault and do_set_pte. __do_fault further calls vma->vm_ops->Fault, Fault will be responsible for calling the partner system API allocated physical page frame, and calls address_space_operations provided readpage interfaces, the contents of the file to read physical page frame. do_set_pte will be called mk_pte create a page table entry, and the page table entry points to the physical address of the page frame.

The differences between the 3 functions are :

- do_read_fault: If !(flags & FAULT_FLAG_WRITE) == true, it means that this is a read-only mapping. At this time, do_read_fault simply calls __do_fault and do_set_pte without any additional actions.
- do_cow_fault: if !(Vma->vm_flags & VM_SHARED) == true, stated that this is a private mapping, and (flags & FAULT_FLAG_WRITE) == false (otherwise you will call do_read_fault), illustrates this mapping will write access. In this case, do_cow_fault call __do_fault later, we will apply for a new physical page, copy the data to the new physical page, and then call do_set_pte, the page table to point to this newly created physical page.

Note that although the function name has cow, it is different from the processing of do_wp_page we described earlier: do_cow_fault handles !pte_present(entry) == true copy-on-write, and do_wp_page handles !pte_present(entry) == false copy-on-write.

- do_shared_fault: If it is a shared map, and maps have write access, will be called do_shared_fault. This means that for mapping the physical page write operation will eventually be written back to disk file. Therefore do_shared_fault addition to calling __do_fault and do_set_pte outside, It will also call vma->vm_ops->page_mkwrite to check whether the disk file can be written, and also call set_page_dirty to mark the page as dirty, so that when the physical page is released, the content will be written back to the disk file.

do_anonymous_page

If it is anonymous & private mapping, it will not be linked to any file, this situation will be handled by do_anonymous_page.

The following is the implementation of the function, we have omitted most of the content, leaving only the outline:

```

//mm/memory.c

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 )
{
...
    page = alloc_zeroed_user_highpage_movable ( vma , address );
...

    entry = mk_pte ( page , vma -> vm_page_prot );

```

```

        if ( vma -> vm_flags & VM_WRITE )
            entry = pte_mkwwrite ( pte_mkdirty ( entry ) );
...
}

```

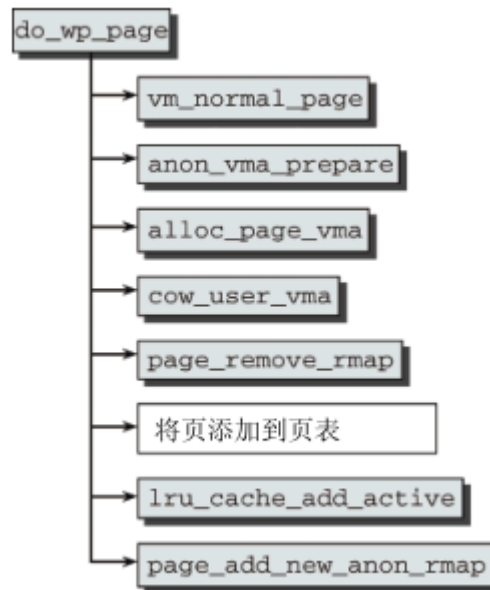
alloc_zeroed_user_highpage_movable is responsible for requesting a physical page from the partner system and clearing the page . mk_pte is responsible for creating a page table entry and pointing to the physical page . If this is a writable anonymous mapping , the corresponding flag of the page table entry will be set to be writable .

do_swap_page

Many details of this function are related to the page swap / recycling mechanism . I won't go into details here . If you are interested, you can read "In-depth Linux Kernel Architecture Chapter 18 "

do_wp_page

The following figure shows the general flow of do_wp_page (note that the order of the actual code may be different from that shown , but it doesn't matter , the main logic is the same):



What we are looking at is a slightly simplified version, which eliminates potential conflicts with the swap cache, and some corners and corners. Because these all complicate the problem and do not help reveal the nature of the mechanism itself.

- kernel first calls `vm_normal_page` and finds the struct page instance of the page through the page table entry . In essence, this function is based on `pte_pfn` and `pfn_to_page` , both of which must be defined in all architectures. The former finds the page number associated with the page table entry, while the latter determines the page instance associated with the page number . This is feasible because the copy-on-write mechanism is only called for pages that actually exist in memory (otherwise, it needs to be automatically loaded through the page fault exception mechanism first) .
- with `page_cache_get` After acquiring the page, next `anon_vma_prepare` ready data structure reverse mapping mechanism, in order to accept a new anonymous area.
- Since the source of the exception is the need to copy a page full of useful data to a new page, the kernel calls `alloc_page_vma` to allocate a new page. `Cow_user_page` then copies the data of the abnormal page to the new page, and the process can then write to the new page.
- Then use `page_remove_rmap` , reverse mapping to delete the original read-only pages.
- new page is added to the page table, and the CPU cache must also be updated at this time .

- Finally, use `lru_cache_add_active` to place the newly allocated page on the active list of the LRU cache, and insert it into the reverse mapping data structure through `page_add_anon_rmap`. Thereafter, user space processes can write data to the page.

Uh, the above is a complete copy of "In-depth Linux Kernel Architecture", I don't have time to understand it carefully, let's put it here first, and then review it later.

3.5 Copy data between kernel and user space

The kernel often needs to copy data from user space to kernel space. For example, when a lengthy data structure is passed indirectly through a pointer in a system call. Conversely, there is also the need to write data from kernel space to user space.

There are two reasons that make it impossible to just pass and dereference pointers. First, user space programs cannot access kernel addresses; second, there is no guarantee that the virtual memory page pointed to by the pointer in the user space is indeed associated with the physical memory page. Therefore, the kernel needs to provide several standard functions to handle the data exchange between the kernel space and the user space, and take these special circumstances into consideration. These APIs are listed below.

3.5.1 APIs

Header file : `include/linux/uaccess.h`

Function	Comment
<code>copy_from_user(to, from, n)</code> <code>__copy_from_user</code>	Copy a string of n bytes from from (user space) to to (kernel space). The return value represents how many bytes of data need to be copied. May sleep.
<code>copy_to_user(to, from, n)</code> <code>__copy_to_user</code>	Copy a string of n bytes from from (kernel space) to to (user space). The return value represents how many bytes of data need to be copied. May sleep.
<code>get_user(type *to, type* ptr)</code> <code>__get_user</code>	Read a simple type variable (char , long , ...) from ptr and write to . According to the type of pointer, the kernel automatically determines the number of bytes to be transferred (1 , 2 , 4 , 8)
<code>put_user(type *from, type *to)</code> <code>__put_user</code>	Copy a simple value from from (kernel space) to to (user space). The corresponding value is automatically judged according to the pointer type <code>put / get_user</code> designed for use with small data , suitable for ioctl medium , as detailed in "character device drivers 5.4 - ioctl Arg "
<code>strncpy_from_user(to, from, n)</code> <code>__strncpy_from_user</code>	The 0 end of the string (up to n characters) from from (user space) copied to to (with n kernel space)
<code>clear_user(to, n)</code> <code>__clear_user</code>	Fill the n bytes after to (user space) with 0

strlen_user(s) __strlen_user	Get the length of a zero- terminated string in user space (including the end character)
strnlen_user(s, n) __strnlen_user	Get the length of a zero- terminated string , but the search is limited to no more than n characters

According to the table of contents , most of the function has two versions . In versions without the underscore prefix , will call `access_user`, the user address space to check . Checks performed in accordance with the architecture and different . For example , a platform-a The check may be to confirm that the pointer is indeed pointing to the location in the user space . Another possibility is that when the page cannot be found in the memory , `handle_mm_fault` is called to ensure that the data has been read into the memory and is ready for processing . All functions apply the above for detection And the correction mechanism to correct the page fault abnormality .

These functions are mainly implemented in assembly language . Because the calls are very frequent , the performance requirements are extremely high . In addition , GNU C must be used to embed the complex structure of the assembly and the link instructions in the code to integrate the abnormal code . I I do not intend to discuss the implementation of each function in detail .

In kernel version 2.5 during development , the compilation process adds an inspection tool . This tool analyzes the source code , checks the user space if the pointer directly between the anti-reference . Originated from the user space pointer must keyword `__user` labeled , so as to distinguish the tool pointer checks need . a specific example is `chroot` system call , it requires a file name as a parameter . the kernel also there are many places contain similar numerals with, parameters from user space .

```
// fs/open.c

asmlinkage long sys_chroot ( const char __user * filename ) {
    ...
}
```