MEMORY MANAGEMENT (4): FAQS

April 20, 2019 (Http://Www.Mysixue.Com/?P=117) JL
(Http://Www.Mysixue.Com/?Author=1)

4 FAQs

4.1 Flags :

The various flags related to the memory management system are summarized as follows :

- each physical page has its own flags, defined in struct page -> unsigned Long flags ; detailed in 2.2.2 section of the " page "
- each memory block (s page composed of a memory block) has its own pageblock_flags , defined struct Zone -> unsigned Long * pageblock_flags detailed in 2.4.3 "Auxiliary functions and variables" section
- at the time of application memory , it requires the use of gfp flags, as detailed in 2.4.5 section "Assigning mask GFP_XXX "
- ❄♒♏ kmem_cache_create function needs to use SLAB flags. For details , see the introduction of kmem_cache_create in section 2.5.3 " APIs "
- ❄♒♏ page table only needs to store the address of the physical page , so the 0-PAGE_SHIFT bits of each item (pte) of the page table are free . These free bits can be used for storage protection , see section 3.1.1 " pte flags " for details 》
- The sub-area of the vmalloc area is represented by vm_struct , each vm_struct has its own flags, see section 3.2.4 " vm_struct " for details
- each sub-region of memory mapped area with vm_area_struct representation , each vm_area_struct has its own vm_flags, see 3.3.4 Day "sub-region (vm_area_struct)"

4.2 When to allocate physical page frames & create page tables

- When you create a new memory mapping , unless specified MAP_LOCKED flag , it would not apply to the physical page frame buddy system , is not good to create a page table .
  Only when a certain virtual address is accessed and it is found that it does not have a corresponding physical page , a page fault exception will be triggered . At this time, the page fault exception handler will apply to the partner system for a physical page frame and establish a page table .
- specified when creating maps MAP_LOCKED sign , or is in the stack to expand the time , will by mm_populate trigger for each virtual page page-missing fault , then the page fault handler will start running . That these two operations successfully returned , The physical page frame and page table are all ready .

4.3 The relationship between CONFIG_HIGHMEM and physical memory size

If the physical memory is very small , in `` 2.3.3 Determining the boundary value of low-end / high-end memory", it will be judged that the system does not have high-end memory , which means that no matter whether CONFIG_HIGHMEM is enabled or not , the system does not have high-end memory .

If the physical memory is large , but CONFIG_HIGHMEM is not enabled , the kernel can only use linearly mapped physical memory , and cannot use excess physical memory .
Such a case , " 2.3.3 to determine the low-end / when high memory boundary value" , the kernel will print message , reminding the user enable CONFIG_HIGHMEM.

If large physical memory , but also enabled CONFIG_HIGHMEM, the system can use the high memory . General use " HIGHMEM " referred to this situation .

4.4 vmalloc and HIGHMEM it matter ?

Regardless of whether there is HIGHMEM or not , vmalloc can be used : when HIGHMEM exists , vmalloc allocates physical page frames from ZONE_HIGHMEM first ; otherwise, vmalloc allocates physical page frames from ZONE_NORMAL .

But PKMAP domain CONFIG_HIGHMEM relationship , and only when the system is enabled CONFIG_HIGHMEM time , you can use PKMAP domain , see " 3.2.3 PKMAP & FIXADDR " .

4.5 vmalloc & buddyinfo

What if I use vmalloc apply for a block of memory , will buddyinfo affect the output of it ? Try this test it , the BBB on the board , write a ko, init stage with vmalloc application 8 a page, exit stage release the 8 th page. Observe the changes in buddyinfo during the period .

The example code is as follows :

```
    vmalloc_ptr = vmalloc ( PAGE_SIZE * 8 );
    if ( vmalloc_ptr != NULL ) {
        printk ( KERN_ALERT "vmalloc_ptr %p\n" , vmalloc_ptr );
        *( unsigned int *) vmalloc _ptr = 88 ;
         printk ( KERN_ALERT "value of vmalloc_ptr %d\n" , *( unsigned int *)
vmalloc_ptr );
    } else {
```

```
        printk ( KERN_ALERT "vmalloc alloc failed\n" );
    }
```

The results of the operation are as follows :

```
root@embest:/home# cat /proc/buddyinfo
Node 0, zone   Normal      6     17     37     48     21     33      6      3      3      1      4     45
root@embest:/home# insmod HelloWorld.ko
[33868.694343] vmalloc_ptr e0c5a000
[33868.698114] value of vmalloc_ptr 88
root@embest:/home# cat /proc/buddyinfo
Node 0, zone   Normal      6     17     37     48     21     33      6      3      3      1      4     45
```

Conclusion will not affect buddyinfo output , because vmalloc is the application of physical pages page by page , it will be from pcp list get inside pages , only when pcp list is insufficient page . Pcp list will apply to the partner system page , The output of buddyinfo will be affected at this time . For details, see " 2.4.6 buffered_rmqueue "

4.6 Will the page table of the user process contain the kernel virtual address space ?

In ARM32 on , when the fork when the process , a kernel page table directory will copy to the user process page table task_struct-> mm_struct-> pgd in . In charge copy of the code is mm_alloc (https://elixir.bootlin.com/linux/v4.20/source/kernel/fork.c#L1030) -> mm_init -> mm_alloc_pgd -> pgd_alloc (https://elixir.bootlin.com/linux/v4.20/source/arch/arm/mm/pgd.c#L33) (for arm32): (https://elixir.bootlin.com/linux/v4.20/source/kernel/fork.c#L1030) (https://elixir.bootlin.com/linux/v4.20/source/arch/arm/mm/pgd.c#L33)

```
pgd_t *pgd_alloc(struct mm_struct *mm)
{
        pgd_t *new_pgd, *init_pgd;
        pud_t *new_pud, *init_pud;
        pmd_t *new_pmd, *init_pmd;
        pte_t *new_pte, *init_pte;

        new_pgd = __pgd_alloc();
        if (!new_pgd)
                goto no_pgd;

        memset(new_pgd, 0, USER_PTRS_PER_PGD * sizeof(pgd_t));

        /*
         * Copy over the kernel and IO PGD entries
         */
        init_pgd = pgd_offset_k(0);
        memcpy(new_pgd + USER_PTRS_PER_PGD, init_pgd + USER_PTRS_PER_PGD,
                       (PTRS_PER_PGD - USER_PTRS_PER_PGD) * sizeof(pgd_t));
```

But in the ARM64 on , the user process's page table is stored in ttbr0_el1 , kernel mode page table is stored in ttbr1_el1 , so no copy.

arch/arm64/mm/ pgd.c (https://elixir.bootlin.com/linux/v4.20/source/arch/arm64/mm/pgd.c#L33)

```
pgd_t *pgd_alloc(struct mm_struct *mm)
{
        if (PGD_SIZE == PAGE_SIZE)
                return (pgd_t *)__get_free_page(PGALLOC_GFP);
        else
                return kmem_cache_alloc(pgd_cache, PGALLOC_GFP);
}
```

4.7 Copy_ to {,} _user from () Consideration

http://www.wowotech.net/memory_management/454.html (http://www.wowotech.net/memory_management/454.html) , this document discusses this issue very well . A personal summary is as follows :

copy_xxx_user and memcpy comparison , provides the following additional features :

- It will call access_ok to ensure that the user space address passed to the kernel belongs to the current process . Otherwise, this vulnerability (https://www.cnblogs.com/linhaostudy/archive/2018/07/16/9317683.html) may be used to obtain root privileges . (https://www.cnblogs.com/linhaostudy/archive/2018/07/16/9317683.html)
- If the user space address passed to the kernel is an illegal address (the vm_area domain of the user space does not contain this address) , the kernel will not oops , but through the help of the two sections of .fixup and __ex_table , the kernel can return to the user normally Space . If memcpy is used , the kernel will directly oops in this case .
- in enabling CONFIG_ARM64_SW_TTBR0_PAN or CONFIG_ARM64_PAN (in the case of hardware support to be effective) time, when switching to kernel space , will modify the page tables so that the kernel can not access user address space . At this point we can only use copy_ {to, from}_user() This interface (copy_xxx_user will temporarily restore the user space page table at its entrance , and set the user space page table to an invalid value again when it exits) .

If you do not consider the above 3 Dian difference , when accessing the following two user address space in kernel space , copy_xxx_user with memcpy can work :

- A valid user space address , and it has been mapped to physical memory
- a valid user address space , but the map has not been established . For the address , either kernel mode or user mode access it , Page Fault process is almost the same, will help us create and apply physical memory mappings.
