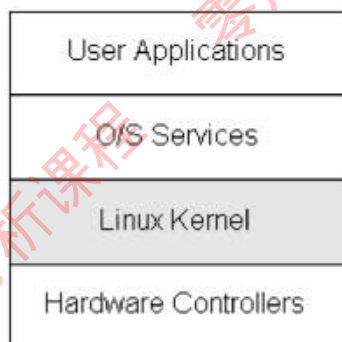


第 0008 讲 内存映射原理机制



一、Linux 内存映射原理机制

创建内存映射时，在进程的用户虚拟地址空间中分配一个虚拟内存区域。内核采用延迟分配物理内存的策略，在进程第一次访问虚拟页的时候，产生缺页异常。如果是文件映射，那么分配物理页，把文件指定区间的数据读到物理页中，然后在页表中把虚拟页映射到物理页。如果是匿名映射，就分配物理页，然后在页表中把虚拟页映射到物理页。

Linux 内核中的内存映射是指将一个文件或设备的内容映射到进程的虚拟地址空间，使得进程可以像访问内存一样访问这些内容。通过内存映射，进程可以避免频繁的磁盘读写操作，提高文件和设备的访问效率。分为两种：

(1) 文件映射：文件支持的内存映射，把文件的一个区间映射到进程的虚拟地址空间，数据源是存储设备上的文件。

文件映射是一种将磁盘上的文件映射到内存中的技术，通过这种方式可以使程序直接访问内存中的数据，而不需要

通过繁琐的读写操作。在文件映射中，系统会为文件分配一个虚拟地址空间，并将磁盘上的数据按需载入到该地址空间中。这样一来，在程序访问这个地址空间时就相当于直接访问了文件中对应位置的数据。文件映射通常用于处理大型文件或数据库等需要频繁读写操作的应用场景。

(2) 匿名映射：没有文件支持的内存映射，把物理内存映射到进程的虚拟地址空间，没有数据源。

在 Linux 中，匿名映射是指通过 `mmap()` 系统调用创建的一种内存映射方式，它不需要指定文件或设备作为映射对象，而是直接将一个匿名的、由操作系统管理的虚拟内存区域映射到进程的地址空间中。

这种匿名映射通常用于实现共享内存、进程间通信等功能。

由于没有实际的文件或设备对象与之对应，因此它不需要进行任何 I/O 操作，也不会影响任何磁盘上的数据。另外，匿名映射还可以设置访问权限和共享标志等参数，以满足具体需求。

需要注意的是，在使用匿名映射时需要确保所申请的虚拟内存区域大小不能超过系统可用内存总量。同时，在使用匿名映射时也要遵循合适的安全措施以防止出现安全漏洞。

二、虚拟内存区域数据结构 (vm_area_struct)

vm_area_struct 是用来描述一个进程的虚拟内存区域的数据结构，在 Linux 系统中，每个进程都有一个虚拟地址空间，由多个不同的 vm_area_struct 组成。主要核心成员如下：

```
C mm_types.h X
include > linux > C mm_types.h
529 /*
530  * This struct describes a virtual memory area. There is one of these
531  * per VM-area/task. A VM area is any part of the process virtual memory
532  * space that has a special rule for the page-fault handlers (ie a shared
533  * library, the executable area etc).
534  */
535 struct vm_area_struct {
536     /* The first cache line has the info for VMA tree walking. */
537
538     unsigned long vm_start;    /* Our start address within vm_mm. */
539     unsigned long vm_end;     /* The first byte after our end address
540                                within vm_mm. */
541
542     struct mm_struct *vm_mm;   /* The address space we belong to. */
543
544     /*
545      * Access permissions of this VMA.
546      * See vmf_insert_mixed_prot() for discussion.
547      */
548     pgprot_t vm_page_prot;
549     unsigned long vm_flags;    /* Flags, see mm.h. */
550 }
```

linux 内核使用 vm_area_struct 结构来表示一个独立的虚拟内存区域，由于每个不同质的虚拟内存区域功能和内部机制都不同，因此一个进程使用多个 vm_area_struct 结构来分

别表示不同类型的虚拟内存区域。各个 `vm_area_struct` 结构使用链表或者树形结构链接，方便进程快速访问，如下图所示：

结构体 `struct vm_area_struct`

它表示的是一块连续的虚拟地址空间区域，给进程使用的，地址空间范围是 $0 \sim 3G$ ，对应的物理

结构体 `struct vm_struct`

表示一块连续的虚拟地址空间区域。给内核使用，地址空间范围是 $(3G + 896M + 8M) \sim 4G$ ，又

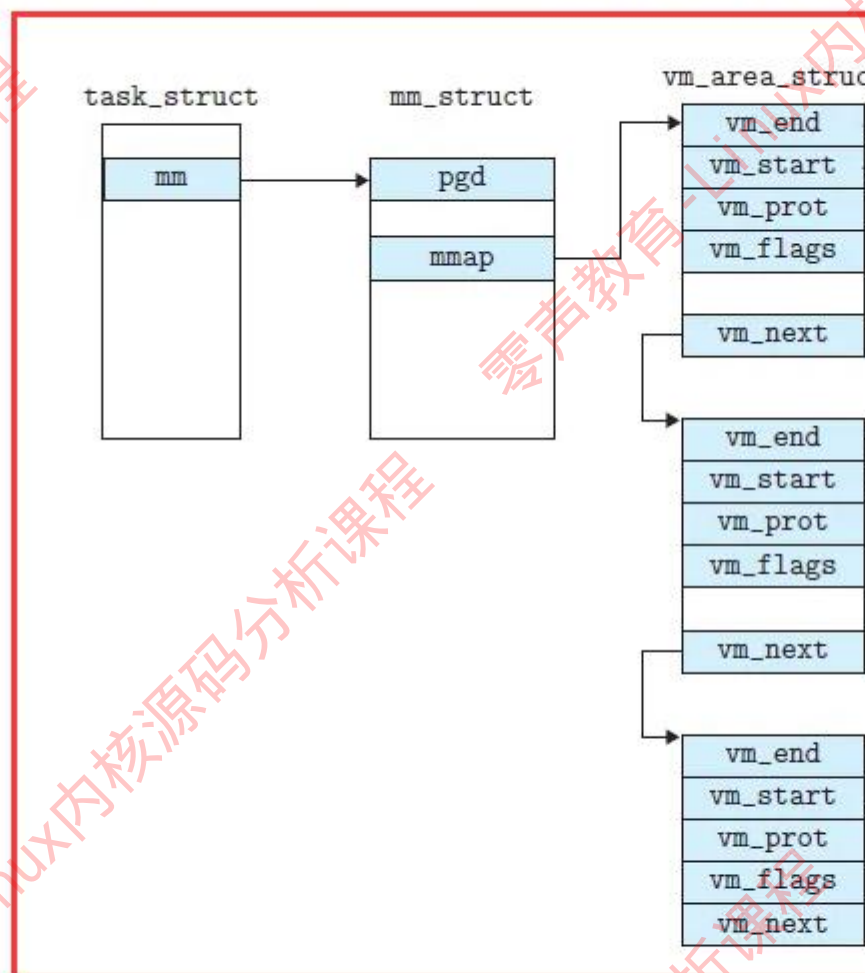
两者的区别是什么呢？

在Linux中，`struct vm_area_struct`表示的虚拟地址是给进程使用的，而`struct vm_struct`表示的
`vm_area_struct`表示的地址空间范围是 $0 \sim 3G$ ，而`struct vm_struct`表示的地址空间范围是 $(3G +$
 $G)$ 呢？

原来， $3G \sim (3G + 896M)$ 范围的地址是用来映射连续的物理页面的，这个范围的虚拟地址和对

而 $(3G + 896M) \sim (3G + 896M + 8M)$ 是安全保护区域（例如，所有指向这8M地址空间的指针都
来映射非连续的物理页面。

linux内核使用vm_area_struct结构来表示一个独立的虚拟内存区域，由于每个不同质的虚拟内存来分别表示不同类型的虚拟内存区域。各个vm_area_struct结构使用链表或者树形结构链接



vm_area_struct结构中包含区域起始和终止地址以及其他相关信息，同时也包含一个vm_ops结构。程序对某一虚拟内存区域的任何操作需要用的信息，都可以从vm_area_struct中获得。mmap地址相连

三、内存映射（系统调用实战操作）

1、mmap()

系统调用是 Linux 中的一种内存映射文件的方式，它可以将一个文件或者设备映射到进程的虚拟地址空间中，从而实现对该文件或设备的访问。mmap() 函数的原型如下：

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

mmap() 系统调用是 Linux 中的一种内存映射文件的方式，它可以将一个文件或者设备映射到进程的虚拟地址空间中，从而实现对该文件或设备的访问。

参数说明：

- addr：期望映射的起始地址，通常为 NULL 表示由系统自动分配
- length：需要映射到进程地址空间中的字节数
- prot：设置内存区域保护方式，可选项包括 PROT_READ、PROT_WRITE、PROT_EXEC 等。其中 PROT_READ 表示允许读取该区域数据，PROT_WRITE 表示允许写入该区域数据，PROT_EXEC 表示允许执行该区域代码。
- flags：控制映射对象的类型和属性。可选项包括 MAP_SHARED、MAP_PRIVATE 等。其中 MAP_SHARED 表示共享映射，多个进程都能够修改这段内存；MAP_PRIVATE 表示私有

映射，在多个进程之间不共享修改。

- fd: 要被映射到内存中的文件描述符
- offset: 被映射对象在文件中偏移量

mmap() 函数成功返回新创建虚拟地址空间部分首地址指针，失败则返回错误码并设置 errno 变量。通过 munmap() 函数可以释放该虚拟地址空间。

mmap() 系统调用常用于实现进程间共享内存，以及在进程中动态加载共享库等操作。

2、munmap()

在 Linux 中，munmap() 系统调用用于释放由 mmap() 函数创建的内存映射区域。munmap() 函数的原型如下：

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

参数说明：

- addr: 要释放的内存区域的起始地址
- length: 要释放的内存区域大小

munmap() 函数成功返回 0，失败则返回-1 并设置 errno 变量。

需要注意的是，munmap() 只会解除对内存区域的映射关系，

并不会将该内存区域所占用的物理内存清空。因此，在使用 `mmap()` 释放共享内存时，应先确保所有进程都已经解除了对该共享内存的映射关系，否则可能导致数据丢失或者损坏。

以下是 Linux 内核中 `mmap()` 系统调用的实例代码：

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

int main() {
    // 创建一个匿名的共享内存区域

    void* addr = mmap(NULL, 4096, PROT_READ |
    PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, -1, 0);

    if (addr == MAP_FAILED) {
        perror("mmap");
        return -1;
    }

    // 在共享内存中写入数据

    char* str = (char*) addr;
    sprintf(str, "Hello world!");
```



```
// 打印共享内存中的数据
```

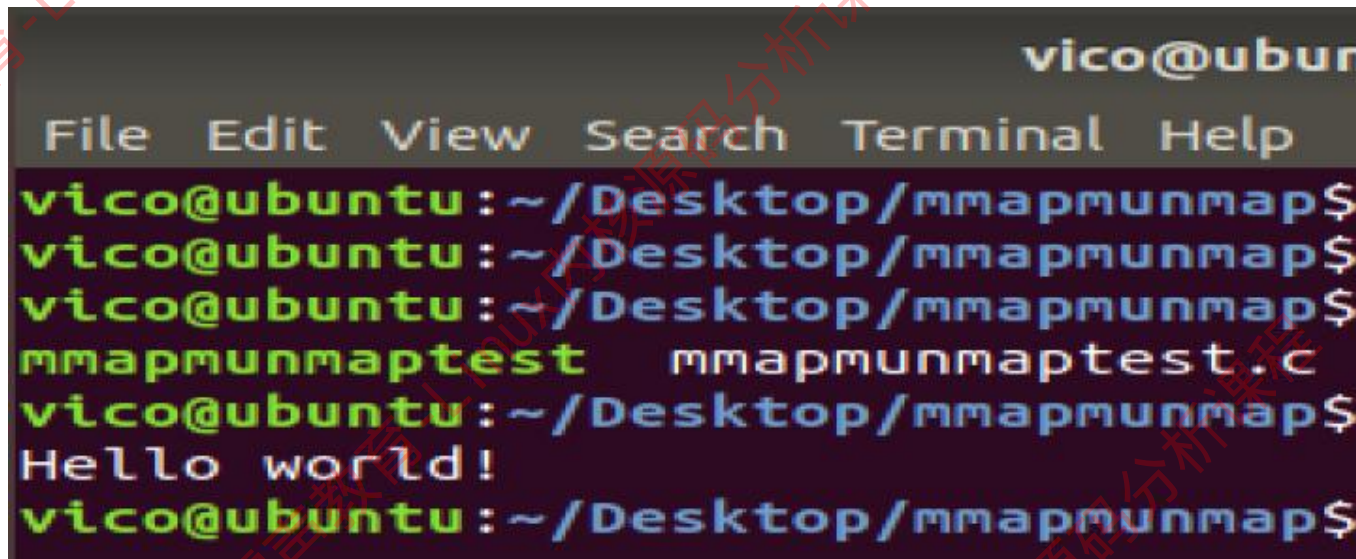
```
printf("%s\n", str);
```

```
// 解除映射并释放资源
```

```
munmap(addr, 4096);
```

```
return 0;
```

```
}
```



```
vico@ubuntu:~/Desktop/mmapmunmap$  
vico@ubuntu:~/Desktop/mmapmunmap$  
vico@ubuntu:~/Desktop/mmapmunmap$  
mmapmunmaptest mmapmunmaptest.c  
vico@ubuntu:~/Desktop/mmapmunmap$  
Hello world!  
vico@ubuntu:~/Desktop/mmapmunmap$
```

此示例创建一个大小为 4KB 的匿名共享内存区域，将其映射到进程地址空间，并在其中写入字符串"Hello world!"。然后，它打印该字符串并解除映射以释放资源。

以下是一个简单的示例代码，演示了如何使用 Linux 内核中的 `mmap()` 系统调用：

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

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

```
int main() {
```

```
    char* message = "Hello, world!";
```

```
    size_t len = sizeof(char) * 14;
```

```
    // 创建一个匿名共享内存区域，大小为 len 字节
```

```
    void* addr = mmap(NULL, len, PROT_READ |  
    PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

```
    if (addr == MAP_FAILED) {
```

```
        perror("mmap");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    // 将消息复制到共享内存区域中
```

```
    memcpy(addr, message, len);
```

```
    // 打印共享内存区域中的消息
```

```
    printf("%s\n", (char*) addr);
```

```
// 解除映射并释放资源
```

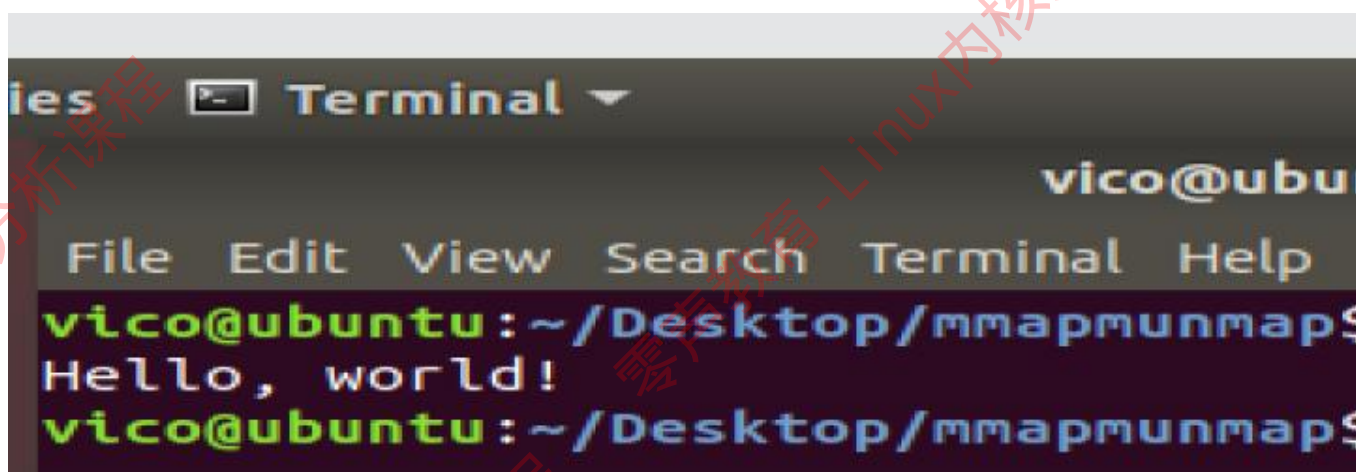
```
munmap(addr, len);
```

```
return EXIT_SUCCESS;
```

```
}
```

此程序创建了一个大小为 14 个字符的匿名共享内存区域，并将字符串 "Hello, world!" 复制到其中。然后，它打印该字符串并解除映射以释放资源。

在这个示例中，我们使用了 `MAP_PRIVATE` 标志来指定只有当前进程可以访问该共享内存区域。如果我们想要允许其他进程也能够访问该区域，则应该使用 `MAP_SHARED` 标志来创建一个可共享的映射。



```
ies Terminal vico@ubu
File Edit View Search Terminal Help
vico@ubuntu:~/Desktop/mmapmunmap$ ./mmapmunmap
Hello, world!
vico@ubuntu:~/Desktop/mmapmunmap$
```

3、mprotect()

在 Linux 中，mprotect() 系统调用用于修改内存区域的保护属性。mprotect() 函数的原型如下：

```
int mprotect(void *addr, size_t len, int prot);
```

参数说明：

- addr：要修改保护属性的内存区域起始地址
- len：要修改保护属性的内存区域大小
- prot：新的保护属性，可以是以下几种值之一：
 - PROT_NONE：禁止读、写和执行操作
 - PROT_READ：允许读取操作
 - PROT_WRITE：允许写入操作
 - PROT_EXEC：允许执行操作

mprotect() 函数成功返回 0，失败则返回-1 并设置 errno 变

量。

需要注意的是，`mprotect()` 只能修改由 `mmap()` 或者 `shmat()` 创建的映射关系所对应的内存区域的保护属性。如果要修改常规堆栈分配的内存区域的保护属性，可以使用 `mmap()` 创建匿名映射，并将其与所需内存区域进行关联。

以下是一个简单的示例代码，演示了如何使用 Linux 内核中的 `mprotect()` 系统调用：

下面是一个使用 `mprotect()` 系统调用来修改内存保护权限的示例代码：

```
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main() {
 // 分配 4096 字节大小的内存空间
 void* ptr = mmap(NULL, 4096, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
 if (ptr == MAP_FAILED) {
```

```

 perror("mmap failed");
 }
 exit(1);
}

// 输出分配的内存地址和初始值
printf("Memory allocated at %p with value %d\n", ptr,
(int)ptr);

// 修改内存保护权限为只读
if (mprotect(ptr, 4096, PROT_READ) == -1) {
 perror("mprotect failed");
 exit(1);
}

// 尝试写入新值,会导致“段错误”(Segmentation fault)
(int)ptr = 123;

return 0;
}

```

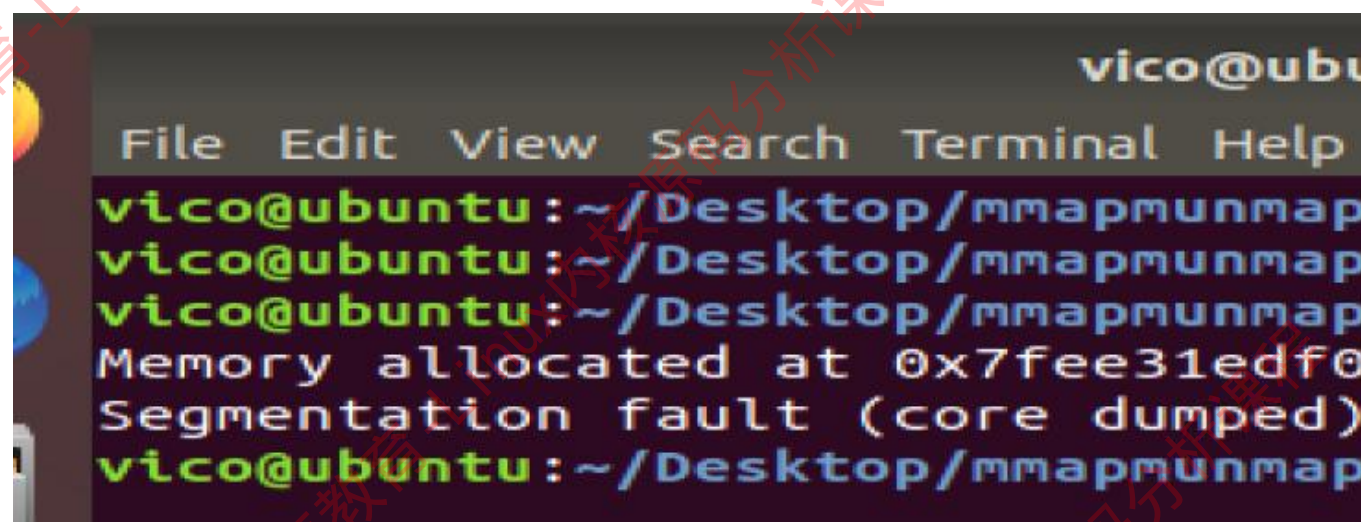
该程序通过调用 `mmap()` 函数在进程虚拟地址空间中分配了



一个大小为4096字节、可读可写的内存空间。然后使用 printf 输出了分配的内存地址和初始值。

接着，使用 mprotect() 函数将这个内存区域的保护权限修改为只读。最后尝试向只读区域写入新值，会触发“段错误”。

注意：以上代码仅供参考，实际使用时需要谨慎处理指针类型、访问权限等问题，避免导致不可预期的错误或安全漏洞。

A terminal window screenshot from an Ubuntu system. The window title is "vico@ubu". The menu bar shows "File Edit View Search Terminal Help". The terminal output shows the user "vico@ubuntu" in the directory "~/Desktop/mmapmunmap" running a command. The output displays "Memory allocated at 0x7fee31edf0" followed by "Segmentation fault (core dumped)". The prompt "vico@ubuntu:~/Desktop/mmapmunmap" is visible at the bottom.

```
vico@ubu
File Edit View Search Terminal Help
vico@ubuntu:~/Desktop/mmapmunmap
vico@ubuntu:~/Desktop/mmapmunmap
vico@ubuntu:~/Desktop/mmapmunmap
Memory allocated at 0x7fee31edf0
Segmentation fault (core dumped)
vico@ubuntu:~/Desktop/mmapmunmap
```

“Segmentation fault (core dumped)”是一种错误提示，通常出现在运行 C 或 C++ 程序时。它意味着程序试图访问无效的内存地址或对受保护的内存区域进行了非法操作。

在 Linux 系统中，当发生段错误时，操作系统会向终端输出“Segmentation fault (core dumped)”信息，并生成一个称为“core dump”的文件。这个文件包含了程序崩溃时的内存映像，可以用于调试和分析问题。

当我们看到“Segmentation fault (core dumped)”错误信息

时，应该检查代码中是否存在指针未初始化、数组越界、空指针引用等错误。如果找到了这些问题，则需要修复它们以避免出现段错误。此外，还可以使用调试工具如 GDB 来分析生成的核心转储文件以更深入地了解问题所在。