Linux kernel Project: Memory Space

Mtest

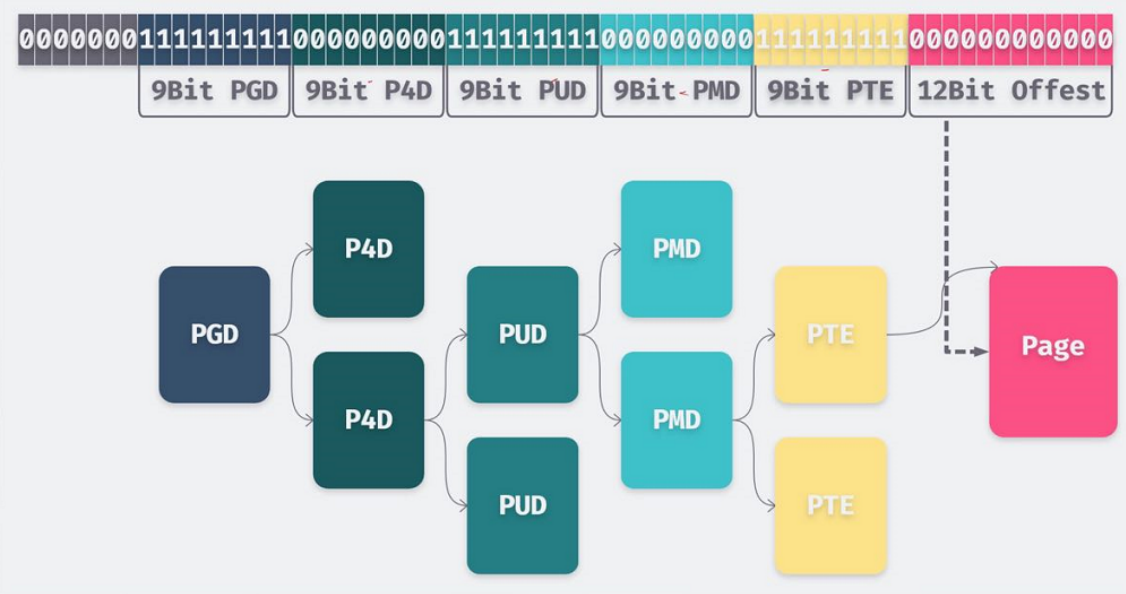
- **运用相关字符串处理函数对输入进行解析：**

```
char *strsep(char **stringp, const char *delim);
/*
参数1：指向字符串的指针的指针
参数2：指向规定分隔符的指针
功能：以参数2所指的字符作为分界符，将参数1的值所指的字符串分割开，返回值为被参数2分开的左边的那个字符串，同时会导致参数1的值（指向位置）发生改变，即，参数1的值会指向分隔符号右边的字符串的起始位置
*/
```

```
int kstrtoul (  const char * s, unsigned int base, unsigned long * res);
/*
参数1：指向需要解析的目标字符串，该字符串必须以 null 结尾，并且在其结束 null 之前还可以包含一个换行符。
参数2：要使用的基数。最大支撑基数为16。如果以0为基数，那么字符串的基数就会用传统的语义自动检测到，如果以0x 开头，那么这个数字        就会被分析为十六进制(不区分大小写)
参数3：指向存储转换成功的结果
*/
```

- **线性地址——>物理地址**

  Linux 中为各种硬件的页式寻址提供了一套通用的抽象模型，其支持5级页表，分别是：



  - 页全局目录：PGD
  - 页四级目录：P4D
  - 页上级目录：PUD
  - 页中级目录：PMD
  - 页表项：PTE

在实验过程中，我们首先需要将特定进程的线性地址`addr`转换成物理地址。而线性地址转换成物理地址的过程其实就是不断访问各级页表的过程。

我们通过`task_struct-->mm-->pgd`字段获得该进程页全局目录PGD的基地址。之后进行页表的转换，具体过程如下：

1. pgd基地址+offset（addr中PGD字段）=p4d基地址
2. p4d基地址+offset（addr中P4D字段）=pud基地址
3. pud基地址+offset（addr中PUD字段）=pmd基地址
4. pmd基地址+offset（addr中PMD字段）=pte基地址
5. pte基地址+offset（addr中PTE字段）=page基地址

为实现上述过程，我们可以使用linux内核所提供的相关宏来进行实现，具体代码如下：

```
#define pgd_index(addr)     ((addr) >> PGDIR_SHIFT)
#define pgd_offset(mm, addr)    (((mm)->pgd) + pgd_index(addr))
#if CONFIG_PGTABLE_LEVELS > 3
/* In 4 level paging, p4d_* macros work on pgd */
#define p4d_none(x)     (!p4d_val(x))
#define p4d_bad(x)      ((p4d_val(x) & ~PAGE_MASK))
/*
 * 2nd level paging: pud
 */
/*
 * In 3 level paging, pud_* macros work on pgd
 * In 4 level paging, pud_* macros work on pud
 */
#define pud_none(x)     (!pud_val(x))
#define pud_bad(x)      ((pud_val(x) & ~PAGE_MASK))
/*
 * Due to the strange way generic pgtable level folding works, the pmd_*
macros
 *  - are valid even for 2 levels (which supposedly only has pgd - pte)
 *  - behave differently for 2 vs. 3
 * In 2  level paging        (pgd -> pte), pmd_* macros work on pgd
 * In 3+ level paging (pgd -> pmd -> pte), pmd_* macros work on pmd
 */
#define pmd_none(x)     (!pmd_val(x))
#define pmd_bad(x)      ((pmd_val(x) & ~PAGE_MASK))
/*
 * 4th level paging: pte
 */
#define pte_none(x)     (!pte_val(x))
#define pte_present(x)      (pte_val(x) & _PAGE_PRESENT)
#define pte_clear(mm,addr,ptep) set_pte_at(mm, addr, ptep, __pte(0))
#define pte_page(pte)       pfn_to_page(pte_pfn(pte))
```

其中，`xxx_offset()`的主要功能为根据相关页表的基地址和addr中特定字段的偏移量计算出相应的页表项地址。

在经过5次页表查询后，获得了指向线性地址addr所对应的页框的物理基地址的PTE表项pte。在此基础上，我们可以使用`pte_pfn()`来获得addr所对应的页框的物理基地址，再使用实验指导书中所提供的系统调用：`pfn_to_page()`来得到该物理页所对应的结构体`struct page`。为了程序的简洁性，我们也可以直接使用`pte_page()`来直接获得该物理页所对应的结构体。其实`pte_page()`的定义如下：

```
#define pte_page(pte)   pfn_to_page(pte_pfn(pte))
```

至此，我们已经获得了线性地址addr所对应的物理页的结构体page。

- **物理地址——>内核空间**

  正如实验指导书中所说，因为模块代码处于内核内存空间中，所以并不能直接访问该页的内容，还需要将该页映射到内核内存空间中。`kmap_local_page()`系统调用可以完成这项工作，其在linux内核中的定义如下：

```
/**
 * kmap_local_page - Map a page for temporary usage
 * @page:   Pointer to the page to be mapped
 *
 * Returns: The virtual address of the mapping
 *
 * Can be invoked from any context.
 *
 * Requires careful handling when nesting multiple mappings because the map
 * management is stack based. The unmap has to be in the reverse order of
 * the map operation:
 *
 * addr1 = kmap_local_page(page1);
 * addr2 = kmap_local_page(page2);
 * ...
 * kunmap_local(addr2);
 * kunmap_local(addr1);
 **/
static inline void *kmap_local_page(struct page *page);
```

  获得的返回参数就是该物理页在内核空间中所对应的基地址，之后我们需要将基地址与addr最后12bit的offset相拼接，最后得到addr在内核态下所对应的虚拟地址kernel_addr。相关代码截图如下：

```
unsigned long kernel_base = kernel_addr;
kernel_base = kernel_base & PAGE_MASK;
unsigned long kernel_offset = addr & (~PAGE_MASK);
kernel_addr = kernel_base |kernel_offset;
```

  至此，我们完成了将特定进程中的指定线性地址转换为内核态所对应的虚拟地址。最后，只需要通过对`kernel_addr`进行读写就能完成访问和修改mtest_test进程的功能了。

实验截图：

首先在终端运行测试程序，获得预期的实验效果：

```
chn@chn-virtual-machine:~/CS353/project3/mtest$ ./mtest_test
pid: 17007
addr: 7fff2148613a
content: 42
read result: 42
content: 1
```

之后使用`dmesg`命令查看内核的输出情况：

```
[15344.027675] the command :r 17007 7fff2148613a ,the length=21
[15344.027679] pid=17007
[15344.027680] address=7fff2148613a
[15344.027681] kernel base addr = 0xffff967a95a54000
[15344.027682] base = 0xffff967a95a54000,offset = 0x13a, kernel addr = 0xffff967a95a5413a
[15344.027713] the command :w 17007 7fff2148613a 1 ,the length=23
[15344.027714] pid=17007
[15344.027715] address=140733751779642
[15344.027715] content=1
[15344.027716] kernel base addr = 0xffff967a95a54000
```

Maptest

- **mmap系统调用**

  mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上`mmap` 函数的原型如下：

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

  - `addr`：指定映射的虚拟内存地址，可以设置为 NULL，让 Linux 内核自动选择合适的虚拟内存地址。
  - `length`：映射的长度。
  - `prot`：映射内存的保护模式，可选值如下：
  - - `PROT_EXEC`：可以被执行。
    - `PROT_READ`：可以被读取。
    - `PROT_WRITE`：可以被写入。
    - `PROT_NONE`：不可访问。
  - `flags`：指定映射的类型，常用的可选值如下：
  - - `MAP_FIXED`：使用指定的起始虚拟内存地址进行映射。
    - `MAP_SHARED`：与其它所有映射到这个文件的进程共享映射空间（可实现共享内存）。
    - `MAP_PRIVATE`：建立一个写时复制（Copy on Write）的私有映射空间。
    - `MAP_LOCKED`：锁定映射区的页面，从而防止页面被交换出内存。
    - ...
  - `fd`：进行映射的文件句柄。
  - `offset`：文件偏移量（从文件的何处开始映射）。

本次实验要求我们创建一个proc文件/proc/maptest，之后使用系统调用mmap将该文件映射到用户的虚拟空间中。为实现上述目标，我们首先需要在创建proc文件时在内核空间分配一个物理页，并向该页中写入指定内容content。在分配物理页时要求我们使用alloc_page()系统调用，详情如下。

- **alloc_page()**

  alloc_pages( )函数以gfp_mask分配方式分配2的order次方（1<<order）个连续的物理页。

  ```
  static inline struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
  ```

  gfp_mask：是分配标志，内核分配内存有多种方式，该参数告诉内核如何分配以及在哪分配所需的内存，内存分配最终总是调用*get_free_pages( )来实现，这也是GFP*前缀的由来。其中分配标志（gfp_mask）的取值定义见文件linux-3.19.3/include/linux/gfp.h可以取以下各值,比如：

  - GFP_KERNEL：该分配方式最常用，是内核内存的正常分配，它可能睡眠。

有的标志用双下划线作为前缀，它们可与上面标志“或”起来使用，以控制分配方式，比如：

- __GFP_DMA：要求分配可用于DMA的内存。

order：指要释放的物理页数，其取值为2的order次方个。

返回参数说明：

alloc_pages( )函数返回page结构体指针，指向所分配的物理页中的第一个页，如果分配不成功，则返回NULL。

内核用struct page结构表示系统中的每个页框。

在分配了页面并获得了返回了struct page之后，我们可以参照之前task1中将物理页映射到内核空间中的步骤，将分配的物理页映射到内核空间后，使用memcy()函数将content的内容写入指定的物理页中。具体代码如下:

```
page = alloc_pages(GFP_KERNEL,0);
unsigned long *kernel_addr;
kernel_addr = (unsigned long*)kmap_local_page(page);
//copy the content
memcpy(kernel_addr,content,strlen(content));
```

之后，我们需要设置mmap的回调函数来实现文件映射的功能。根据实验指导书的要求通过设置 struct proc_ops 结构体的 proc_mmap来完成回调函数的设置。其中实现具体功能的关键函数为 **remap_pfn_range()**, 详细情况如下。

- **remap_pfn_range()**

  remap_pfn_range（）函数的原型:

  ```
  int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr,
  unsigned long pfn, unsigned long size, pgprot_t prot);
  ```

  该函数的功能是创建页表。其中参数vma是内核根据用户的请求自己填写的，而参数addr表示内存映射开始处的虚拟地址，因此，该函数为addr~addr+size之间的虚拟地址构造页表。另外，pfn（Page Fram Number）是虚拟地址应该映射到的物理地址的页面号，实际上就是物理地址右移PAGE_SHIFT位。如果PAGE_SHIFT为4kb，则PAGE_SHIFT为12，因为PAGE_SHIFT等于1<<PAGE_SHIFT。最后一个参数prot是新页所要求的保护属性。

  在本次实验中，我们首先将需要获得page对应的物理页帧号，将其作为调用remap_pfn_range()中的pfn参数。具体代码如下:

  ```
  unsigned long kernel_addr_frame = page_to_pfn(page);
  if (remap_pfn_range(vma, vma->vm_start, kernel_addr_frame, (vma->vm_end - vma->vm_start),vma->vm_page_prot))
  {
      pr_info("remap failed!");
      return -1;
  }
  ```

至此我们完成了对物理页的分配和内容拷贝，和mmap系统调用回调函数的设置。

实验截图：

使用测试程序，获得了向page中写入的特定内容：

```
chn@chn-virtual-machine:~/CS353/project3/maptest$ ./maptest_test
I am Haonan Chen,
Listen to me say thanks
Thanks to you, I'm warm all the time
I thank you
For being there
The world is sweeter
I want to say thanks
Thanks to you, love is in my heart
I thank you, for being there
To bring happiness
```

思考与感悟

1. 在实验过程中，使用`kmap_local_page()`函数可以将特定的物理页映射到内核空间中，并返回该页在虚拟空间中的基地址。在查阅资料后发现，使用`page_address()`同样也可以获得该物理页在内核空间中虚拟基地址。下面是相关代码和实验过程截图：

```
unsigned long *kernel_addr;
kernel_addr = (unsigned long*)kmap_local_page(page);
pr_info("kernel base addr = 0x%lx",kernel_addr);
kernel_addr = (unsigned long*)page_address(page);
pr_info("kernel base addr = 0x%lx",kernel_addr);
```

```
22275.879953] kernel base addr = 0xffff967a49c48000
22275.879954] kernel base addr = 0xffff967a49c48000
hn@chn-virtual-machine:~/CS353/project3/mtest$
```

   我们可以发现，使用以上两种系统调用所获得的虚拟内存地址是相同的。

2. 在完成任务2的过程中，一开始因为没有完全理解remap_pfn_range()中参数pfn的意义，导致误将内核空间中page的页帧号传入函数，使得程序无法正常工作。最后在查阅源码注释后发现pfn应该是物理页帧号，与进行memcpy内容拷贝时的目的地址不同。

3. 本次实验作为Linux内核这么课的理论教学之下的实践性作业，从理论走向实践，让我进一步学习到了Linux内核中内存管理方面的知识。自己动手进行页表的访问和虚拟地址的转换，查阅详细资料与源码。慢慢的理解并应用LInux内核所提供的api接口使我收获颇多。与此同时，助教的实验指导书里面的内容也很具有指导意义，为我指明了方向。最后感谢陈全老师和各位助教的悉心解答。