OS-PRJ3 Report

Team 12： b02902025呂承洋 b02902071陳柏堯

一、Trace Code程式追蹤及kernel相關函式意義探索：

A．根據PRJ2中關於Process的描述，我們知道在fork之後，系統會為我們的process產生一個task_struct結構，我們將之理解為PCB。因此，我們可以在其中找到進程的調度信息和內存信息。于</include/linux/sched.h> 的struct task_struct中 找到如下定義：

```
1287 #endif
1288
1289         struct mm_struct *mm, *active_mm;
1290 #ifdef CONFIG_COMPAT_BRK
1291         unsigned brk_randomized:1;
1292 #endif
```

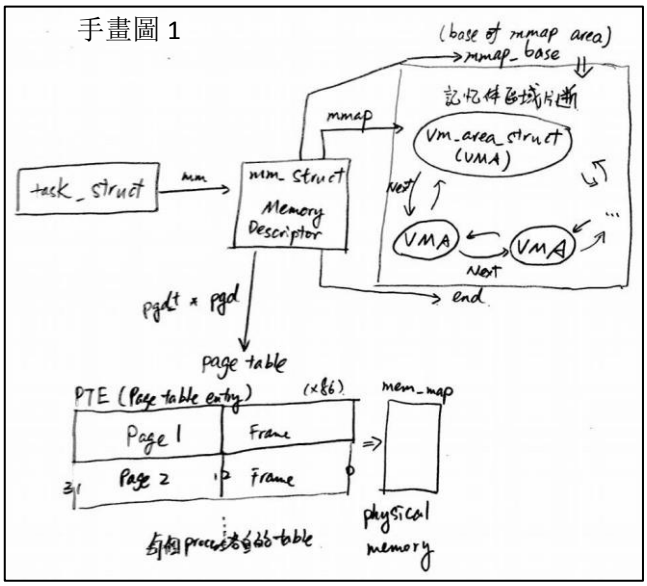
可以得知mm_struct 即該process memory management 的地方。

B．依照《Understanding the Linux Virtual Memory Manager》書中的描述，我們在</include/linux/mm_types.h>中找到mm_struct的定義。(書中舊版kernel是定義在sched.h中。) 對於 task_struct 而言，mm struct就是該進程的memory descriptor 記憶體描述符。

```
struct mm_struct {
        struct vm_area_struct * mmap;           /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache;     /* last find vma res
#ifdef CONFIG_MMU
        unsigned long (*get_unmapped_area) (struct file *filp,
                                unsigned long addr, unsigned long le
                                unsigned long pgoff, unsigned long f
        void (*unmap_area) (struct mm_struct *mm, unsigned long addr
#endif
        unsigned long mmap_base;                /* base of mmap area
        unsigned long mmap_legacy_base;         /* base of mmap area
        unsigned long task_size;                /* size of task vm s
        unsigned long cached_hole_size;         /* if non-zero, the
        unsigned long free_area_cache;          /* first hole of siz
        pgd_t * pgd;
        atomic_t mm_users;                      /* How many users wi
        atomic_t mm_count;                      /* How many referenc
```

VMA

Page Table



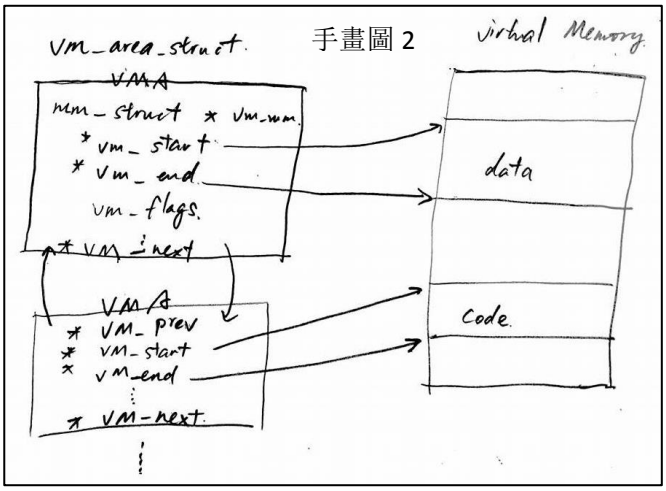
根據上述結構，我們可以畫出如右手畫圖1的 mm_struct 架構。mm_struct 主要指向vm_area_struct和PTE兩個主要結構

C．之後我們在 mm_types.h 中找到了上述vm_area_struct的宣告如下：

```
struct vm_area_struct {
        struct mm_struct * vm_mm;       /* The addre
        unsigned long vm_start;         /* Our start
        unsigned long vm_end;           /* The first
                                           within vm

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

        pgprot_t vm_page_prot;          /* Access pe
        unsigned long vm_flags;         /* Flags, se
```



由此我們可以畫出右手畫圖2，vm_area_struct的vm_start和vm_end與虛擬記憶體地址的關係。

其中vm_mm是指向原來的mm_struct(Memory Descriptor).

D. 在<include/linux/mm.h>中我們找到了vm_operations_struct的宣告，其功能是對vm_area_struct進行操作open\close\fault。

```
struct vm_operations_struct {
        void (*open)(struct vm_area_struct * area);
        void (*close)(struct vm_area_struct * area);
        int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
```

在《Understanding the Linux Virtual Memory Manager》中，我們可以發現在我們調用mmap()映射文件的時候，process會利用這個函式open一塊vm_area出來映射給這個文件內容的地址。與本次PRJ3有關的(*fault)是指，在發生缺頁錯誤的時候，（根據<Understanding……>書中所述）do_no_page()會呼叫這個函式，他會定位這個缺少的page所在的位置或者建立新頁面并填充缺頁的數據并返回它。

E. 接著，我們在</mm/filemap.c>中找到filemap_fault的相關敘述，其中說明了，當mm_struct調用記憶體數據出現page_fault時，vma_operations會調用filemap_fault函式來從外存提取page進來。如下圖：

```
1637  * filemap_fault() is invoked via the vma operations vector for a
1638  * mapped memory region to read in file data during a page fault.
1639  *
1640  * The goto's are kind of ugly, but this streamlines the normal case of having
1641  * it in the page cache, and handles the special cases reasonably without
1642  * having a lot of duplicated code.
1643  */
1644 int filemap_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
```

會調用的定義如下圖：

```
1766 const struct vm_operations_struct generic_file_vm_ops = {
1767         .fault          = filemap_fault,
1768 };
1769
```

因此，我們可以理解在ssize_t do_generic_file_read()中讀取數據發生page fault時，會調用vma_operations的.fault, 而.fault就指向filemap_fault函式。

F． 在filemap_fault()中我們看到以下code:

```
1647         struct file *file = vma->vm_file;
1648         struct address_space *mapping = file->f_mapping;
```

```
1663          page = find_get_page(mapping, offset);
```

如1663行，我們理解為我們在vma的mapping中去尋找我們要的數據位置(offset)，這一步是為了確保我們下一步要做異步預讀取(do_async_readahead)還是同步預讀取(do_sync_readahead)。

G．<filemap.c>中同步和異步的預讀取的相關函式 (sync_readahead & async_readahead)

（1） 同步預讀取是指在page_fault中我們需要的page不在cache中，要調用do_sync_mmap_readaead（），等待需要的page交換回來才能往下做。

```
1673                    do_sync_mmap_readahead(vma, ra, file, offset);
```

在該函式中(如右圖)程式判斷如果vma是隨機訪問(random access)的話，就return，不適用預讀readahead的功能(VM_RandomReadHit(vma)判斷)。

而如果vma是順序存取（sequential access）則具有一定的規律, 則使用readahead讀取資料。（調用page_cache_sync_readahead)。

```
static void do_sync_mmap_readahead(struct vm_area_struct *vma,
                                   struct file_ra_state *ra,
                                   struct file *file,
                                   pgoff_t offset)
{
        unsigned long ra_pages;
        struct address_space *mapping = file->f_mapping;

        /* If we don't want any read-ahead, don't bother */
        if (VM_RandomReadHint(vma))
                return;
        if (!ra->ra_pages)
```

（2）異步預讀取是指需要讀取的page已在cache中，但是還要讀取該page後面一段的page。

```
1670                    do_async_mmap_readahead(vma, ra, file, page, offset);
```

異步的readahead和同步的版本相對應，是讀非當前需要的資料，同樣也只有在順序存取時使用。(若 VM_RAND_READ則return)

最後調用page_cache_async_readahead來讀取資料。

```
static void do_async_mmap_readahead(struct vm_area_struct *vma,
                                    struct file_ra_state *ra,
                                    struct file *file,
                                    struct page *page,
                                    pgoff_t offset)
{
        struct address_space *mapping = file->f_mapping;

        /* If we don't want any read-ahead, don't bother */
        if (VM_RandomReadHint(vma))
                return;
        if (ra->mmap_miss > 0)
                ra->mmap_miss--;
        if (PageReadahead(page))
                page_cache_async_readahead(mapping, ra, file,
                                           page, offset, ra->ra_pages);
}
```

H. 不使用預讀readahead

我們在filemap_fault()中發現，在調用readahead的sync或async函式之後，程式判斷如果page仍然沒有找到則會跳轉到後面的no_cached_page處，如下：

```
1677 retry_find:
1678                    page = find_get_page(mapping, offset);
1679                    if (!page)
1680                            goto no_cached_page;
1681            }
```

而no_cached_page處則會呼叫page_cache_read()函式做普通的讀取，如下：

```
1717 no_cached_page:
1718            /*
1719             * We're only likely to ever get here if MADV_RANDOM is i
1720             * effect.
1721             */
1722            error = page_cache_read(file, offset);
```

其函式主要功能是分配一個page給pagecache, 然後讀回資料并回傳。

```
1539 static int page_cache_read(struct file *file, pgoff_t offset)
1540 {
1541            struct address_space *mapping = file->f_mapping;
1542            struct page *page;
1543            int ret;
1544
1545            do {
1546                    page = page_cache_alloc_cold(mapping);
```

二、將read ahead scheme 改成pure demand paging

根據trace code結果，我們發現只要將do_sync_mmap_readaead()和do_async_mmap_readaead()注釋起來，則當vma發生page fault時，就會跳入no_cached_page並呼叫page_cache_read來進行單純的page讀取。 如下圖：

```
1670                    //do_async_mmap_readahead(vma, ra, file, page, offset);
1671            } else {
1672                    /* No page in the page cache at all */
1673                    //do_sync_mmap_readahead(vma, ra, file, offset);
```

三、read ahead scheme algorithm 預讀的運作方式和結構

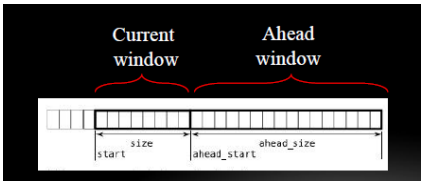
code如第一大點G點所述, read ahead scheme是指檔案系統vma為process一次讀出比預期更多的檔內容並緩存在page cache中，這樣下一次讀請求到來時部分頁面直接從page cache讀取即可。本次要讀的是同步的預讀，而下次可能要讀的是異步的預讀。

read ahead的基本結構是file_ra_state(如下左圖)。於是就形成了一個window(預讀窗口)。(ra->start 起始位置, ra->size同步大小, ra->async_size非同步大小)，也就形成了如下右圖 助教投影片的window結構了。

```
struct file_ra_state{

pgoff_t start;            /* where readahead started */
unsigned int size;        /* # of readahead pages */
unsigned int async_size;    /* do asynchronous readahead when
                  there are only # of pages ahead */

unsigned int ra_pages;        /* Maximum readahead window */
unsigned int mmap_miss;        /* Cache miss stat for mmap accesses */
loff_t prev_pos;        /* Cache last read() position */
}
```



四、read ahead scheme 和 pure demand paging 比較:

（1） 優劣：

pure demand paging之讀取需要運行的paging，好處是可以保留更多記憶體讓更多process來作context switch，可以減少程式運行延遲的潛在因素，對於手機之類電子設備未必需要那麼多高成本記憶體來運行read_ahead.但缺點就是需要更多的時間來讀取page。

Read ahead scheme可以加快使用者的文件讀取的感覺，提升主記憶體使用效率等。

(2) Case Study:

利用助教提供的syslog、test、input.log運行的結果：

(左下圖是pure demand paging, 右下圖是read ahead scheme)

```
# of major pagefault: 6567
# of minor pagefault: 190
# of resident set size: 26608 KB

real    0m56.917s
user    0m0.008s
sys     0m1.276s
```

```
# of major pagefault: 4162
# of minor pagefault: 2593
# of resident set size: 26632 KB

real    0m40.927s
user    0m0.016s
sys     0m1.028s
```

Cache hit是minor page fault，Cache miss是major page fault

我們發現pure demand paging 的cache miss非常高，而read ahead scheme則較低，由於read ahead scheme 都有預先讀取一定的page，所以可以提升cache hit，在cache中找到所需page的效率。同時我們利用time來分別計算他們的時間，real time包括了process被IO block住所等待的時間，我們發現pure demand paging 所花費的時間比read ahead scheme 長很多。

五、本作業中遇到的問題：

1、 網路上、參考書中很多舊版本kernel和kernel3.2.54中的code有很多地方不同。

2、 Filemap.c 中vma、mvf等結構以及相關代碼非常多、很難判斷哪些是重點，要花很多時間。投影片中給予的函式名稱往往很難找到具體的位置在哪裡。

3、 Kernel編寫者叫我們不要bother with read-ahead了:

```
/* Avoid banging the cache line if not needed */
if (ra->mmap_miss < MMAP_LOTSAMISS * 10)
        ra->mmap_miss++;

/*
 * Do we miss much more than hit in this file? If so,
 * stop bothering with read-ahead. It will only hurt.
 */
if (ra->mmap_miss > MMAP_LOTSAMISS)
        return;
```