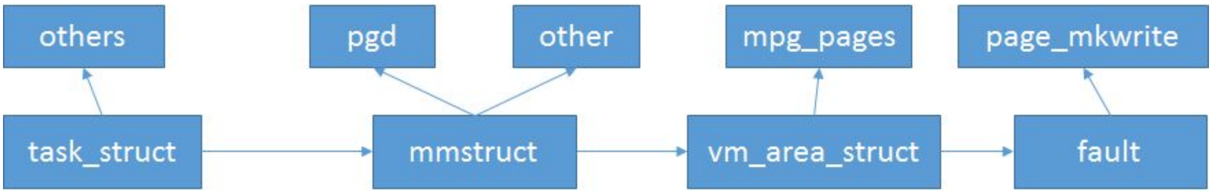
OS project 3

張耿健 R05922092
孫凡耘 B04902045

* Environment: Ubuntu 12.04.5 LTS 32-bit

Part 1

Trace *mmap( )*



在linux/sched.h中，task_struct 包含 mm_struct類別 (在linux/mm_types.h中被定義)

```
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 result */
        unsigned long (*get_unmapped_area) (struct file *filp,
                                unsigned long addr, unsigned long len,
                                unsigned long pgoff, unsigned long flags);
        void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
        unsigned long mmap_base;                /* base of mmap area */
        unsigned long task_size;                /* size of task vm space */
        unsigned long cached_hole_size;         /* if non-zero, the largest hole below free_area_cache */
        unsigned long free_area_cache;          /* first hole of size cached_hole_size or larger */
        pgd_t * pgd;
        atomic_t mm_users;                      /* How many users with user space? */
        atomic_t mm_count;                      /* How many references to "struct mm_struct" (users count as 1) */
        int map_count;                          /* number of VMAs */
        struct rw_semaphore mmap_sem;
        spinlock_t page_table_lock;             /* Protects page tables and some counters */

        struct list_head mmlist;                /* List of maybe swapped mm's.  These are globally strung
                                                 * together off init_mm.mmlist, and are protected
                                                 * by mmlist_lock
                                                 */
```

mm_struct 是一種 memory descriptor資料結構，主要用來儲存記憶體的資訊。他的第一個成員是vm_area_struct (在linux/mm_types.h中被定義)

```
struct vm_area_struct {
        struct mm_struct * vm_mm;       /* The address space we belong to. */
        unsigned long vm_start;         /* Our start address within vm_mm. */
        unsigned long vm_end;           /* The first byte after our end address
                                           within vm_mm. */

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

        pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
        unsigned long vm_flags;         /* Flags, see mm.h. */

        struct rb_node vm_rb;
```

vm_area_struct 是一種 virtual memory area (VMA) 的linked list

在 file-backed 記憶體區域的操作被定義在linux/filemap.c：

occurs , it will

```
const struct vm_operations_struct generic_file_vm_ops = {
    .fault = filemap_fault,
    .map_pages = filemap_map_pages,
    .page_mkwrite = filemap_page_mkwrite,
};
```

當一個 page fault 產生的時候，就會 invoke filemap_fault()。

Trace *filemap_fault( )*

當 page fault 發生時,filemap_fault() 會先去使用 find_get_page() 去確認是否所有的 required page 在 page cache 裡面.
如果有在 page cache 裡面的話 , 就試著去用 async_readahead 去讀 pages.萬一找不到的話使用 sync_readahead 去讀 required page and readhead other pages to cache.
這兩個 function do_async_mmap_readahead() and do_sync_mmap_readahead()都會去確認 vma 是否 randomly reading by VM_RandomReadHint().

If VM_RandomReadHint() returns true,那上述兩個函數都會直接 return..

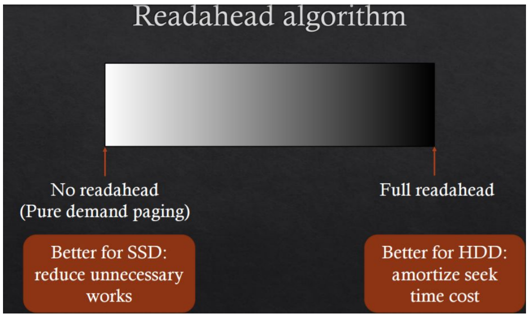
We will find the page by async_readahead and async_readahead and sync_readahead if MADV_RANDOM isn't in effect which mean vma isn't randomly reading.If we found the page(checking by find_get_page()), we will lock the page and check whether it is truncated and up-to-date.
After checking its size under page lock , we return the required page,

If MADV_RANDOM is in effect , we goto no_cached_page.

no_cached_page will simply do page_cache() to read the required page and go back to do find_get_page() again.

Part 2

跟助教確認我們決定這個部分直接實作pure demand paging. 如下圖所示，屬於一個極端情況。



追蹤fielmap.c的時候(*filemap_fault()*)，發現最簡單實作pure demand paging的方式就是讓do_sync_mmap_readahead跟do_async_mmap_readahead這兩個函式直接return不做後面的任何處理。

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

Result:

Readahead

```
[sudo] password for willy:
# of major pagefault: 4201
# of minor pagefault: 2550
# of resident set size: 26676 KB
```

Demand paging

```
[sudo] password for willy:
# of major pagefault: 6567
# of minor pagefault: 184
# of resident set size: 26676 KB
willy@willy-VirtualBox:~/Downloads/hw3$
```

Readahead: the pager will read more than needed data into memory. Often, it can enhance the performance since programs usually perform sequential I/O. But, guessing incorrectly will lead to inefficieny and more overhead.

Pure Demand Paging: It does not guess or read more than the needed data. As a result, when the user perform sequential I/O, it optimizes nothing. We can observe that lots of major page fault occur since it doesn't read ahead anything.

Bonus

Introduction

We modify the origin readahead function in mm/readahead.c.

Instead of trying a whole new algoithm, we decide to alter the parameters in the readahead algorithm. We found out that struct file_ra_state defined in include/linux/fs.h controls the readahead state of the file.

```
/*
 * Track a single file's readahead state
 */
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 */
};
```

We want to know where the members of file_ra_state are changed. We found get_next_ra_size() and get_init_ra_size() decides the size of readahead. Originally, the implementation inside linux2.6.32.60 uses 2 and 4 as the *parameter*(commented out in the following figure).

```
/*
 *  Get the previous window size, ramp it up, and
 *  return it as the new window size.
 */
static unsigned long get_next_ra_size(struct file_ra_state *ra,
                                      unsigned long max)
{
    unsigned long cur = ra->size;
    unsigned long newsize;

    if (cur < max / 16)
        newsize = /*parameter*/ * cur;
    else
        newsize = /*parameter*/ * cur;

    return min(newsize, max);
}
```

Our experiment results are shown below.

```
static unsigned long get_next_ra_size(struct file_ra_state *ra,
                                                 unsigned long max)
{
        unsigned long cur = ra->size;
        unsigned long newsize;
        if (cur < max / 64)
                newsize = 16 * cur;
        else if( cur < max / 32 )
                newsize = 8 * cur;
        else if (cur < max / 16)
                newsize = 4 * cur;
        else
                newsize = 2 * cur;

        return min(newsize, max);
}
```

```
[sudo] password for willy:
# of major pagefault: 4204
# of minor pagefault: 2548
# of resident set size: 26676 KB
```

```
static unsigned long get_next_ra_size(struct file_ra_state *ra,
                                                    unsigned long m
ax)
{
        unsigned long cur = ra->size;
        unsigned long newsize;
        if (cur < max / 64)
                newsize = 32 * cur;
        else if( cur < max / 32 )
                newsize = 16 * cur;
        else if (cur < max / 16)
                newsize = 8 * cur;
        else
                newsize = 4 * cur;

        return min(newsize, max);
}
```

```
[sudo] password for willy:
# of major pagefault: 4204
# of minor pagefault: 2547
# of resident set size: 26676 KB
```

After trying two different set of parameters in get_next_ra_size(), we found out our modification doesn't really make any changes. So we made an assumption, that is, when we try to increase the growing speed of the readahead size, it grows too fast so it exceeds *max* pretty fast.

To prove this idea, we made another modification. That is, directly return max inside get_next_ra_size(). The result is shown below and is still pretty much the same. So we proved our idea that the readahead size grows rapidlly to max so that moidfying parameters inside get_next_ra_size() doesn't make any significant changes (At least that's true when using the testing program that the TA gave us).

```
static unsigned long get_next_ra_size(struct file_ra_state *ra, unsigned long max) {
    return max;
}
```

```
# of major pagefault: 4201
# of minor pagefault: 2593
# of resident set size: 26676 KB
```