OS Project 3

- b03902071 資工二 葉奕廷
- b03902078 資工二 林書瑾

Result

***All the test is under** *Ubuntu 12.04.5 LTS 32-bit, Linux 3.2.54* **environment.**

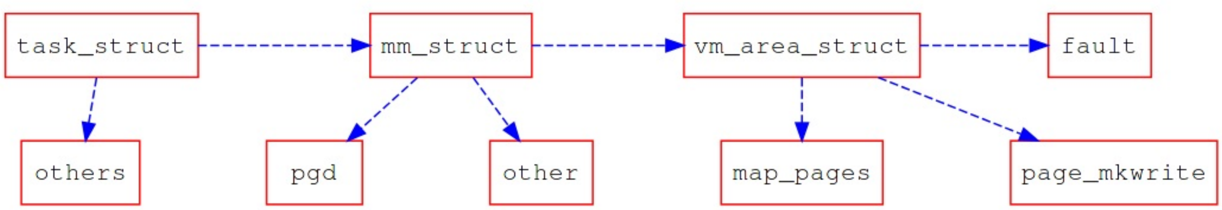
Original version:

```
$ ./test | tail -3
# of major pagefault: 4157
# of minor pagefault: 2651
# of resident set size: 26616 KB
```

Revised version:

```
$ ./test | tail -3
# of major pagefault: 6574
# of minor pagefault: 238
# of resident set size: 26608 KB
```

Trace *mmap()*



In `linux/sched.h`, *task_struct* (per process) contains *mm_struct*, which is defined in `linux/mm_types.h`. *mm_struct* is the memory descriptor, storing information about usage of memory. Its first member is *vm_area_struct* (i.e. memory region), defined in `linux/mm_types.h`. *vm_area_struct* is a linked list of virtual memory area (VMA).

In addition, for file-backed memory regions, the operation is assigned in `linux/filemap.c`:

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

As a result, when a page fault occurs, it will invoke *filemap_fault()*.

Trace *filemap_fault()*

When a page fault occurs, the invoked `filemap_fault()` will check whether the required page is in the page cache by `find_get_page()` at first.
If we found the page in page cache, we will try to do async_readahead to read pages after the page we found. If not, we will

do sync_readahead to read the required page and readahead other pages to cache. Both `do_async_mmap_readahead()` and `do_sync_mmap_readahead()` will check whether that vma is randomly reading or not by `VM_RandomReadHint()`.

```
/* 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))
```

If `VM_RandomReadHint()` returns true, there is no need to do readahead so both functions will return immediately.

We will find the page by async_readahead and sync_readahead if MADV_RANDOM isn't in effect i.e. 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:
    /*
     * We're only likely to ever get here if MADV_RANDOM is in
     * effect.
     */
```

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

Implementation

By our tracing of `filemap_fault()`, we find the most easy way to do pure demand paging is to make both `do_async_mmap_readahead()` and `do_sync_mmap_readahead()` return immediately.

```
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)
{
    return;
```

```
static void do_sync_mmap_readahead(struct vm_area_struct *vma,
                    struct file_ra_state *ra,
                    struct file *file,
                    pgoff_t offset)
{
    return;
```

Pure Demand Paging vs. Readahead Algorithm

	Pure Demand Paging	Readahead Algorithm
Major Pagefault	6574	4157
Minor Pagefault	238	2651
Resident Set Size(KB)	26608	26616

In *Readahead Algorithm*, the pager will read more than needed data into memory. This guess is often correct because we usually perform sequential I/O; thus improve the performance. However, guessing incorrectly will lead to overhead and inefficiency. In the table above, fewer major pagefaults occur due to readahead.

In contrast, *Pure Demand Paging* does not guess or read more than needed data; it only reads exactly what users request. In other words, it's impossible to read any redundant data in *Pure Deman Paging*. Nevertheless, when a user perform sequential I/O, it optimizes nothing. In the table above, lots of major pagefault occur since it doesn't read ahead anything.

Bonus

Introduction

We modify the original readahead function in `mm/readahead.c`. At first we trace the `ondemand_readahead()` function because it is called by both *async_readahead* and *sync_readahead*. We find `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 */
};
```

Thus we want to know where the members of *file_ra_state are* changed, and we find `get_next_ra_size()` decides the size of readahead when we assume the file is sequential access. So we try to change the growing speed of readahead size by modifying the parameters.(We also change the `get_init_ra_size()` function.)

```
/*
 *  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);
}
```

We try to increase the growing speed of readahead size for more readahead. Our experiment results are shown below.

Implementation

In `mm/readahead.c`:

```
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 / 32)
        newsize = 16 * cur;
    else if (cur < max / 16)
        newsize = 8 * cur;
    else
        newsize = 4 * cur;

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

Result

```
$ ./test | tail -3
# of major pagefault: 1259
# of minor pagefault: 5551
# of resident set size: 26532 KB
```

Comparison

By increasing the readahead size, the number of major page fault decreases dramatically. Thus, many faults turn to be minor because it reads ahead more data into memory.

On the other side, the resident size of them is almost equal to default *Readahead Algorithm*. Memory usage of both algorithm are approximately 26 MB in the test program.