Operating System Homework 3 : Readahead Algorithm

黃千儀 B05902101
黃麗璿 B05902102

Part 1

How readahead is called when page faults occur?

A page fault occurs when either a virtual memory address is not mapped well, or no physical page is actually mapped to the address, or a write is attempted on a read-only mapping. When the mmap() is called, filemap_fault() is set as page fault handler through this process below :

```
//<arch/x86/kernel/sys_x86_64.c>
void *mmap(){
    error =sys_mmap_pgoff(...);
}
 //<mm/util.c>
mmap_pgoff() {
    rtval = do_mmap_pgoff(...)
}
//<mm/mmap.c>
do_mmap_pgoff() {
    struct mm_struct *mm = current->mm;
    //<asm/current.h>，current point to the current task_struct
    len = PAGE_ALIGN(len); //put len on 4096
    addr = get_unmapped_area(...);
    //search for the memory that hasn't been map
    return mmap_region(addr, len, ...);
}
//<mm/mmap.c>
mmap_region() {
munmap_back:
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent)
    //start from addr,search for vma，store the one that move forward to prev
    if (vma && vma->vm_start < addr + len) {
    //if we found vma before addr+len
        if(do_munmap(mm, addr, len)) //clear it
            return -ENOMEM;
        goto munmap_back;
        //keep doing it until the thing inside len are cleared
    }
    if(!may_expand_vm(mm, len >> PAGE_SHIFT))
    //check the given page number, to determine whether we can expand vm or not
        return -ENOMEN;
    vma = vma_merge(mm, prev, addr, addr+len, flags, NULL, NULL, pgoff, NULL)
    //decide whether we want to merger with prev
    if(vma) // If we successfully merge
        goto out;
    vma = kmem_cache_zalloc(...)
    //if we decided not to merge, then store this vma
```

```
    if(file) {
        vma->vm_file = file;
        error = file->f_op->mmap;
    }
}
//<linux/fs.h>
struct file {
       struct file_operations *f_op;
};
//<linux/fs.h>
struct file_operations {
    int (*mmap) (struct file *, struct vm_area_struct *);
};
//<fs/ext4/file.c>
static const struct file_operations ext4_file_operations(...) = {
    .mmap = ext4_file_mmap,
};
//<fs/ext4/file.c>
ext4_file_mmap() {
    vma->vm_ops = &ext4_file_vm_ops;
}
//<fs/ext4/file.c>
static const struct vm_operations_struct ext4_file_vm_ops = {
    .fault = filemap_fault,
    //This will invoke filemap_fault when page fault
};
```

While the filemap_fault() is invoked, the readahead algorithm will work as follow. There are two branches to be processed depending on whether or not the page can be found in the page cache. However whatever the branch it will choose, it will be ended up calling page_cache_async_readahead() and then calling ondemand_readahead(). Before it returns a value, it will first call __do_page_cache_readahead(). In the implementation of __do_page_cache_readahead(), it preallocate as many pages as we will need,nr_to_read. If the page is already in the page cache, then grab it from the cache. Otherwise, we need to call page_cache_alloc_cold() to clean a space to place the page. Lastly, call read_pages() to start execute I/O.

```
//Part 2
//<mm/filemap.c>
filemap_fault() {
    if(likely(page)) {
        do_async_mmap_readahead(...);
    }
    else {
        do_async_mmap_readahead(...);
    }
}
//<mm/filemap.c>
do_async_mmap_readahead() {
    if(PageReadahead(page))
        page_cache_async_readahead(...)
}
//<mm/readahead.c>
page_cache_async_readahead() {
    ondemand_readahead(...)
}
//<mm/readahead.c>
ondemand_readahead() {
    return __do_page_cache_readahead(...)
}
//<mm/readahead.c>
__do_page_cache_readahead() {}
```

Part 2

Revise the readahead algorithm for smaller response time

```
ra->ra_pages = VM_MAX_READAHEAD * 1024 / PAGE_CACHE_SIZE
page_cache_async_readahead(mapping, ra, file, page, offset, ra->ra_pages)
```

We find that in include/linux/mm.h, there is a macro called in include/linux/mm.h :

#VM_MAX_READAHEAD 128

So, for the smaller response time of this readahead algorithm, we increase the value of **VM_MAX_Readahead to be 512 kbytes or 1028 kbytes.**

```
/* readahead.c */
#define VM_MAX_READAHEAD        128     /* kbytes */
```

```
/* readahead.c */
#define VM_MAX_READAHEAD        512     /* kbytes */
```

OR

```
/* readahead.c */
#define VM_MAX_READAHEAD        1028    /* kbytes */
```

VM_MAX_Readahead = 128 kbytes :

```
# of major pagefault: 4159
# of minor pagefault: 2562
# of resident set size: 26616 KB
```

```
[ 1974.628572]  page fault test program starts !
[ 1977.168851]  page fault test program ends !
syn@syn-VirtualBox:~/Downloads/hw3$
```

VM_MAX_Readahead = 512 kbytes :

```
# of major pagefault: 354
# of minor pagefault: 6365
# of resident set size: 26680 KB
```

```
[   50.378945]  page fault test program starts !
[   52.473703]  page fault test program ends !
```

VM_MAX_Readahead = 1028 kbytes :

```
# of major pagefault: 186
# of minor pagefault: 6534
# of resident set size: 26680 KB
```

```
[   52.896399]  page fault test program starts !
[   53.594969]  page fault test program ends !
```

Table Comparison

Maximum Readahead Size (kbytes)	Number of Major Page Fault	Number of Minor Page Fault	Number of Resident Set Size (KB)	Execution Time (s)
128 (Default)	4159	2562	26616	2.54
240	4158	2642	26636	2.46
512	354	6365	26680	2.09
1028	186	6534	26680	0.69857

Explanation and Discussion

From the result above, if we use maximum readahead size of 1028 kbytes, the consuming time will decrease up to 72.5%. However, if we use the maximum readahead size of 512 kbytes, the consuming time will only decrease 17.72%. Therefore, we conclude that it is better to use the size of 1028 kbytes for smaller response time.