Pintos Project 2_2. User Program

(설계 프로젝트 수행 결과)

과목명: [CSE4070-01] 운영체제
담당교수: 서강대학교 컴퓨터공학과 이혁준
조원: 54조 고창영

개발기간: 2014. 10. 26. - 2014. 10. 27.

최 종 보 고 서

프로젝트 제목: Pintos 프로젝트 2_2. User Program
제출일: 2014. 11. 01.
참여조원: 54조 고창영

1 개발목표

최소한의 기능을 갖는 교육용 OS Pintos를 만들어나가는 과정에서 OS의 open()과 close(), write()/read(), 그리고 create()와 remove()의 Synchronous한 구현을 통해 semaphore 및 lock, 더불어 file descriptor와 file system에 대한 개념들을 직접 실습함으로써 이들에 대한 이해도를 높인다.

2 개발 범위 및 내용

2.1 개발 범위

1. create()
2. remove()
3. open()
4. close()
5. read()
6. write()
7. seek(),tell(),filesize()
8. Synchronization
9. Read-only-executables
10. Resource deallocation

2.2 개발 내용

2.2.1 create(const char* filename, unsigned initial_size)

initial_size만큼의 크기를 갖는 새로운 파일을 생성한다, 성공시 true, 그렇지 않으면 false를 리턴한다.
주의할 점은 새로운 파일을 생성한다고 해서 이 파일을 open해주는 것은 아니란 것이다.

2.2.2 remove(const char* filename)

파일을 삭제한다, 성공시 true, 그렇지 않으면 false를 리턴한다.
주의할 점은 현재 이 파일을 open한 프로세스가 존재할 경우에는 데이터를 정말 삭제하는 것이 아니라 그러한 프로세스들이 모두 close하고 나면 그때 삭제해야 한다.

2.2.3 open(const char *filename)

파일을 open한다. 이제부터 리턴받은 fd를 이용해서 이 파일에 쓰거나 이 파일을 읽을 수 있다.
실패시 -1을 리턴하며, 한 프로세스가 한 파일을 여러 번 open할 수 있다.
그리고 물론 같은 파일을 open해서 얻은 fd들이 같은 파일의 다른 위치를 가리킬 수 있다.

2.2.4 close(int fd)

파일을 close한다. 이제부터 이 fd로는 해당 파일에 접근할 수 없다.

2.2.5 read(int fd, void *buffer, unsigned size)

open에서 얻은 fd를 이용, fd가 가리키는 파일의 현재 위치로부터 size만큼 읽어서 그 내용을 buffer에 저장한다.

2.2.6 write(int fd, const void *buffer, unsigned size)

open에서 얻은 fd를 이용, buffer의 내용을 size만큼 fd가 가리키는 파일의 현재 위치에 쓴다.

2.2.7 seek(int fd, unsigned position),tell(int fd),filesize(int fd)

seek: 현재 fd가 가리키는 위치를 position으로 설정한다.
tell: 현재 fd가 가리키는 위치를 리턴한다.
filesize: 현재 fd가 가리키는 파일의 크기를 리턴한다.

2.2.8 Synchronization

여러 프로세스가 한 파일에 동시에 쓰거나 읽으려고 할 경우, 정상적으로 써지지 않을 수 있다.
따라서 파일을 쓰거나 읽는 작업에는 적절한 Synchronization이 필요하고,
이를 위해 write()와 read()와 같은 함수에는 lock과 같은 mutex가 필요하다.
또한, **Read-only-executables**를 구현하기 위해서도 exec()가 수행될 때마다 프로세스가 성공적으로 로드되었는지를 판별하고, 그럴 때에만 write를 막기 위해 load()가 끝날 때까지 리턴을 지연시키는 Synchronization이 필요하다.

2.2.9 Read-only-executables

현재 실행중인 프로세스의 실행파일에 대해 write()가 수행되는 것을 방지한다.

2.2.10 Resource deallocation

프로세스가 수행을 끝낼 때, open한 file들과 그들을 나타내는 fd들, **Read-only-exectables**를 위해 열었던 file들을 모두 deallocation함으로써 memory leak을 없애고, OS가 안정적으로 계속 역할을 수행할 수 있도록 한다.

3 추진 일정 및 개발 방법

3.1 추진 일정

- 10.26 - 10.26: Pintos 매뉴얼 정독 및 Read-only-exectable을 먼저 구현.
- 10.27 - 10.27: `filesys/file.c/h` 및 `filesys/filesys.c/h` 정독 및 `open(), read(), ...` 등 File 관련 System call 구현.

3.2 개발 방법

개발은 다음과 같은 원칙을 바탕으로 진행했다:

1. Pintos Manual의 내용을 숙지한다.
2. 이미 작성되어있는 Pintos 내부의 코드(file.c 등)를 숙지한다.
3. 구현해야할 각 기능의 목표와 작동, Synchronization의 필요성을 숙지한다.
4. 구현시 한 함수를 작성하면 해당 함수의 작동을 완벽히 분석한 다음 다른 함수를 작성한다.
5. 각 기능 구현의 기한을 정하고, 이에 맞춰 작성할 수 있도록 시간을 잘 안배한다.

또한, 다음과 같이 도표를 그려서 각 함수의 연결과 그 흐름을 확인했다.

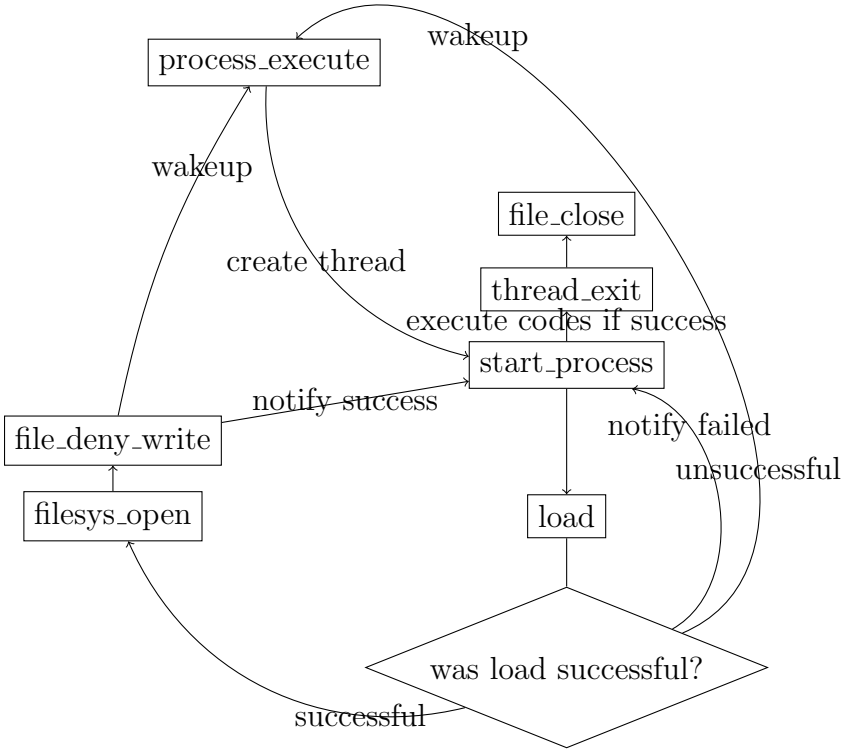

Figure 1: Read-only-executables 구현 흐름도

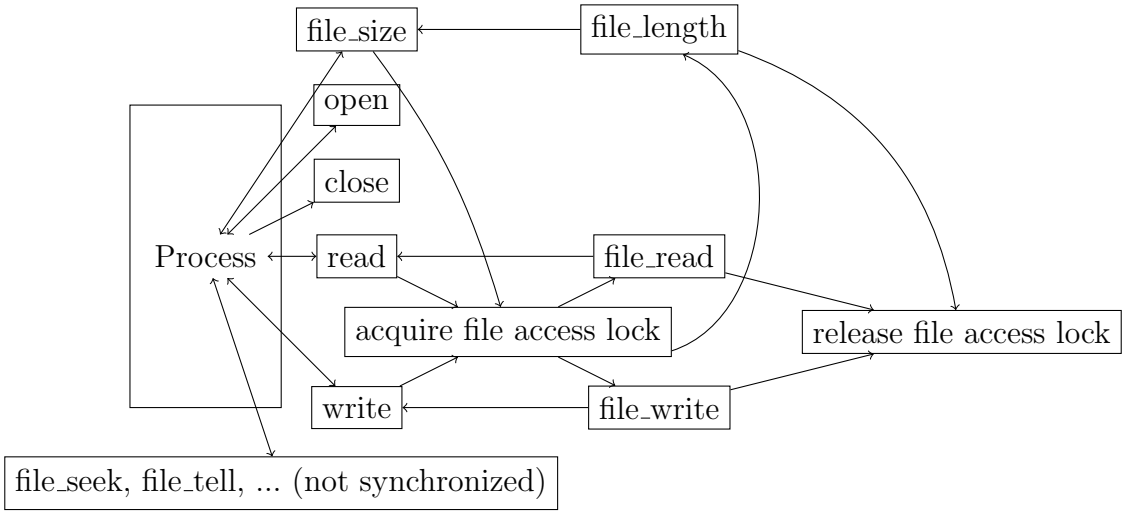

Figure 2: filesystem 관련 system call의 구현 흐름도

3.3 연구원 역할 분담

혼자서 하는 프로젝트였으므로 역할을 분담하기보다는 위에서 적은 것과 같이 날짜를 기준으로 작업을 분배하였다.

4 연구 결과

결과적으로 다음과 같이 모든 테스트에 대해 pass를 받을 수 있었다:

```
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
```

```
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 76 tests passed.
```

4.1 합성 내용

4.1.1 create(const char* filename, unsigned initial_size)

filesys/filesys.h의 filesys_create를 이용해서 새로운 파일을 생성한다.
같은 종류의 인자를 필요로 하는 함수이기 때문에 그대로 넘겨서 호출하고, 그 리턴값을 리턴하면 된다.

4.1.2 remove(const char* filename)

filesys/filesys.h의 filesys_remove를 이용해서 파일의 삭제를 시도한다.
같은 종류의 인자를 필요로 하는 함수이기 때문에 그대로 넘겨서 호출하고, 그 리턴값을 리턴하면 된다. 이 함수 내부에(정확히는 struct inode와 관련 함수에) 해당 file을 open하고 있는 프로세스가 있다면 데이터의 삭제를 그러한 프로세스들이 모두 close할때까지 지연시키는 루틴이 구현되어 있어서 특별히 신경쓸 부분은 없다.

4.1.3 open(const char *filename)

filesys/filesys.h의 filesys_open를 이용해서 파일의 open을 시도한다.
하지만 이 함수가 리턴하는것은 struct file*이고, 앞으로 사용할 write, read같은 system call 함수에서 내부적으로 호출하는 file_read와 같은 함수들은 모두 struct file*을 바탕으로 동작하기 때문에 struct file*을 따로 리스트에 저장하고, 이와 짝을 지은 int타입의 fd를 역시 list에 저장한 다음, 리턴한다. fd가 겹치지 않게 하도록 위해 fd=2로 스레드별로 초기화한 다음, 스레드별로 open이 실행될 때마다 fd가 계속 증가하도록 하였다.

4.1.4 close(int fd)

filesys/file.h의 file_close를 이용해서 파일의 close를 시도한다.
우선 fd에 해당하는 struct file*를 찾고, 존재한다면 이를 close한다.

4.1.5 read(int fd, void *buffer, unsigned size)

fd=0(stdin)의 경우에는 그냥 input_getc를 이용하면 되고, 그렇지 않은 경우에는
filesys/file.h의 file_read를 이용해서 파일의 read를 시도한다.
우선 file_access_lock을 acquire하고,
fd에 해당하는 struct file*를 찾은 다음, 존재한다면 이를 file_read한다.
그 다음 file_read의 수행이 끝나면 다시 file_access_lock을 release한다.
이는 모두 read와 write 등 file과 관련된 critical한 작업들을 synchronize하기 위함이다.

4.1.6 write(int fd, const void *buffer, unsigned size)

fd=1(stdout)의 경우에는 그냥 putbuf를 이용하면 되고, 그렇지 않은 경우에는
filesys/file.h의 file_write를 이용해서 파일의 write를 시도한다.
우선 file_access_lock을 acquire하고,
fd에 해당하는 struct file*를 찾은 다음, 존재한다면 이를 file_write한다.

그 다음 file_write의 수행이 끝나면 다시 file_access_lock을 release한다.
이는 모두 read와 write 등 file과 관련된 critical한 작업들을 synchronize하기 위함이다.

4.1.7 seek(int fd, unsigned position),tell(int fd),filesize(int fd)

세 함수 모두 fd를 인자로 받기에, 우선 이를 이용해서 struct file*을 찾는다.
그후 seek은 filesys/file.h의 file_seek, tell은 filesys/file.h의 file_tell을 호출하여 그대로 리턴하면 되는데, 이중 filesize의 경우에는synchronization을 위해 struct file*을 찾기 전에 file_access_lock을 acquire하고 struct file*을 찾은 뒤 filesys/file.h의 file_length를 호출한 다음 file_access_lock을 release하고 file_length의 리턴값을 리턴한다.

4.1.8 Synchronization

여러 프로세스가 한 파일에 동시에 쓰거나 읽으려고 할 경우, 정상적으로 써지지 않을 수 있다.
따라서 파일을 쓰거나 읽는 작업에는 적절한 Synchronization이 필요하고,
이를 file_access_lock이라는 이름의 lock 하나로 해결했다. 이렇게 함으로써 파일을 읽거나 쓰는 부분에는 최대 한 개의 프로세스만이 동시에 진입할 수 있으므로 race condition이 발생하지 않게 된다. 자세한 설계는 위의 설명들과 Figure 2에서 확인할 수 있다.

4.1.9 Read-only-executables

현재 실행중인 exectable file을 수정할 수 없게 하기 위해서, filesys/file.h의 file_deny_write를 이용했다. 이를 호출하는 것은 Figure 1과 같이 load가 성공하면 현재 실행 파일을 filesys_open하고 이를 file_deny_write하는 것으로 구현하였다.
한편 다시 deny를 해제하는 것은 thread_exit()가 실행될 때 file_close를 하는 것으로 해결했다(file_close는 자동으로 deny를 해제해주도록 되어있다).

4.1.10 Resource deallocation

프로세스가 수행을 끝낼 때, thread_exit에서 그동안 이 프로세스가 open하였지만 close하지 않은 file들과 그들을 나타내는 fd들을 list를 순회함으로써 모두 close하고, 동시에 list를 비우고, 메모리 또한 해제하도록 했다. 그후 마지막으로 Read-only-executables를 위해 열었던 file(자기자신) 역시 close함으로써 모든 메모리를 해제하도록 했다.

4.2 제작 내용

4.2.1 create(const char* filename, unsigned initial_size)

filesys/filesys.h의 filesys_create를 이용해서 새로운 파일을 생성한다.
같은 종류의 인자를 필요로 하는 함수이기 때문에 그대로 넘겨서 호출하고, 그 리턴값을 리턴하도록 구현했다.

```
static bool
create_handler (const char *file_name, unsigned initial_size)
```

```
{
    if (!is_user_vaddr(file_name) ||
        !is_valid_user_addr(file_name) ||
        !is_valid_user_addr(file_name+(strlen(file_name) ? strlen(file_name)-1 : 0)))
        exit_handler(-1);

    return filesys_create(file_name, initial_size);
}
```

4.2.2 remove(const char* filename)

filesys/filesys.h의 filesys_remove를 이용해서 파일의 삭제를 시도한다.
같은 종류의 인자를 필요로 하는 함수이기 때문에 그대로 넘겨서 호출하고, 그 리턴값을 리턴하면 된다.

```
static bool
remove_handler (const char *file_name)
{
    if (!is_user_vaddr(file_name) ||
        !is_valid_user_addr(file_name) ||
        !is_valid_user_addr(file_name+(strlen(file_name) ? strlen(file_name)-1 : 0)))
        exit_handler(-1);

    return filesys_remove(file_name);
}
```

4.2.3 open(const char *filename)

filesys/filesys.h의 filesys_open를 이용해서 파일의 open을 시도한다.
하지만 이 함수가 리턴하는것은 struct file*이고, 앞으로 사용할 write, read같은 system call 함수에서 내부적으로 호출하는 file_read와 같은 함수들은 모두 struct file*을 바탕으로 동작하기 때문에 struct file*을 따로 리스트에 저장하고, 이와 짝을 지은 int타입의 fd를 역시 list에 저장한 다음, 리턴한다. fd가 겹치지 않게 하도록 위해 fd=2로 스레드별로 초기화한 다음, 스레드별로 open이 실행될 때마다 fd가 계속 증가하도록 하였다.
fd와 struct file*를 짝지어서 관리하는것은 둘을 묶는 구조체 struct fdesc를 만들고, 이를 struct thread 안의 list로 관리함으로써 해결했다.

```
#ifdef USERPROG
struct fdesc {
     int fd;
 struct file *file;
 struct list_elem elem;
};
```

```
#endif
struct thread {
 ...
#ifdef USERPROG
 ...
 /* file, and file descriptors. */
 struct list files;
 /* will be initialized to 2 */
 int last_fd;
 ...
#endif
 ...
}
/* syscall 'open' handler */
static int
open_handler (const char *file_name)
{
    if (!is_user_vaddr(file_name) ||
        !is_valid_user_addr(file_name) ||
        !is_valid_user_addr(file_name+(strlen(file_name) ? strlen(file_name)-1 : 0)))
        exit_handler(-1);

    struct file *fp = NULL;
    struct fdesc *new_open = NULL;
    int new_fd = 0;

    if (list_size(&thread_current() -> files) + 2 >= MAX_FILE_PER_THREAD) {
        return -1;
    }

    if ((fp = filesys_open(file_name)) != NULL) {
        /* allocate and initialize new 'struct fdesc' entry */
        new_open = malloc(sizeof(*new_open));
        new_open -> file = fp;
        /* to prevent overlap of fd */
        new_fd = thread_current() -> last_fd++;
        new_open -> fd = new_fd;
        list_push_back(&thread_current() -> files, &new_open -> elem);
        return new_fd;
    } else {
        return -1;
    }
}
```

4.2.4 close(int fd)

filesys/file.h의 file_close를 이용해서 파일의 close를 시도한다.
우선 fd에 해당하는 struct file*를 찾고, 존재한다면 이를 close한다.

```
/* Function-ize repeated iteration. */
static struct list_elem *
find_file_from_thread(int fd)
{
    struct list_elem *iter = NULL;
    for (iter = list_begin(&thread_current() -> files);
iter != list_end(&thread_current() -> files); iter = list_next(iter)) {
        struct fdesc *entry = list_entry(iter, struct fdesc, elem);
        if (entry -> fd == fd) return iter; /* found match! */
    }

    return NULL; /* not found.. */
}

static void
close_handler (int fd)
{
    struct list_elem *iter = NULL;
    struct fdesc *entry = NULL;
    /* Do not close invalid(or std-in/out) files! */
    if (fd < 2) return;

    iter = find_file_from_thread(fd);

    if (iter != NULL) {
        list_remove(iter);
        entry = list_entry(iter, struct fdesc, elem);
        file_close(entry -> file);
/* deallocate memory */
        free(entry);
    }

    return;
}
```

4.2.5 read(int fd, void *buffer, unsigned size)

fd=0(stdin)의 경우에는 그냥 input_getc를 이용하면 되고, 그렇지 않은 경우에는
filesys/file.h의 file_read를 이용해서 파일의 read를 시도한다.
우선 file_access_lock을 acquire하고,

fd에 해당하는 struct file*를 찾은 다음, 존재한다면 이를 file_read한다.
그 다음 file_read의 수행이 끝나면 다시 file_access_lock을 release한다.

```
static int
read_handler (int fd, void *buffer, unsigned size)
{
    if (!is_user_vaddr(buffer) ||
        !is_user_vaddr(buffer+(size > 0 ? size-1 : 0)) ||
        !is_valid_user_addr(buffer))
        exit_handler(-1);

    int bytes_read = 0;
    uint8_t *buf = buffer;
    if (fd == 0) {
        input_init();
        while (size > 0) {
            *buf = input_getc();
            ++buf;
            ++bytes_read;
            --size;
        }
    } else {
        lock_acquire(&file_access_lock);
        struct list_elem *iter = find_file_from_thread(fd);
        struct fdesc *entry = NULL;
        if (iter == NULL) {
            lock_release(&file_access_lock);
            return -1;
        }
        entry = list_entry(iter, struct fdesc, elem);
        bytes_read = file_read(entry->file, buffer, size);
        lock_release(&file_access_lock);
    }

    return bytes_read;
}
```

4.2.6 write(int fd, const void *buffer, unsigned size)

fd=1(stdout)의 경우에는 그냥 putbuf를 이용하면 되고, 그렇지 않은 경우에는
filesys/file.h의 file_write를 이용해서 파일의 write를 시도한다.
우선 file_access_lock을 acquire하고,
fd에 해당하는 struct file*를 찾은 다음, 존재한다면 이를 file_write한다.

그 다음 file_write의 수행이 끝나면 다시 file_access_lock을 release한다.

```
static int
write_handler (int fd, const void *buffer, unsigned size)
{
    if (!is_user_vaddr(buffer) ||
        !is_user_vaddr(buffer+(size > 0 ? size-1 : 0)) ||
        !is_valid_user_addr(buffer))
        exit_handler(-1);

    int bytes_written = 0;
    if (fd == 1) {
        putbuf(buffer, size);
        bytes_written = size;
    } else {
        lock_acquire(&file_access_lock);
        struct list_elem *iter = find_file_from_thread(fd);
        struct fdesc *entry = NULL;
        if (iter == NULL) {
            lock_release(&file_access_lock);
            return -1;
        }
        entry = list_entry(iter, struct fdesc, elem);
        bytes_written = file_write(entry->file, buffer, size);
        lock_release(&file_access_lock);
    }

    return bytes_written;
}
```

4.2.7 seek(int fd, unsigned position),tell(int fd),filesize(int fd)

세 함수 모두 fd를 인자로 받기에, 우선 이를 이용해서 struct file*을 찾는다.
그후 seek은 filesys/file.h의 file_seek, tell은 filesys/file.h의 file_tell을 호출하여 그대로 리턴하면 되는데, 이중 filesize의 경우에는synchronization을 위해 struct file*을 찾기 전에 file_access_lock을 acquire하고 struct file*을 찾은 뒤 filesys/file.h의 file_length를 호출한 다음 file_access_lock을 release하고 file_length의 리턴값을 리턴한다.

```
static int
filesize_handler (int fd)
{
    struct list_elem *iter = find_file_from_thread(fd);
    struct fdesc *entry = NULL;
```

```
    int file_len = 0;
    if (iter == NULL) return -1;

    entry = list_entry(iter, struct fdesc, elem);
    lock_acquire(&file_access_lock);
    file_len = file_length(entry -> file);
    lock_release(&file_access_lock);
    return file_len;
}

static void
seek_handler (int fd, unsigned position)
{
    struct list_elem *iter = find_file_from_thread(fd);
    struct fdesc *entry = NULL;
    if (iter == NULL) return;

    entry = list_entry(iter, struct fdesc, elem);
    file_seek(entry -> file, position);
}

static unsigned
tell_handler (int fd)
{
    struct list_elem *iter = find_file_from_thread(fd);
    struct fdesc *entry = NULL;
    if (iter == NULL) return 0;

    entry = list_entry(iter, struct fdesc, elem);
    return file_tell(entry -> file);
}
```

4.2.8 Synchronization

여러 프로세스가 한 파일에 동시에 쓰거나 읽으려고 할 경우, 정상적으로 써지지 않을 수 있다.
따라서 파일을 쓰거나 읽는 작업에는 적절한 Synchronization이 필요하고,
이를 `file_access_lock`이라는 이름의 `lock` 하나로 해결했다. 이렇게 함으로써 파일을 읽거나 쓰는 부분에는 최대 한 개의 프로세스만이 동시에 진입할 수 있으므로 `race condition`이 발생하지 않게 된다. 자세한 설계는 위의 설명들과 `Figure 2`에서 확인할 수 있다.
정의와 초기화는 다음과 같다:

```
/* Mutex for file access. */
static struct lock file_access_lock;
```

```
void
syscall_init (void)
{
  lock_init(&file_access_lock);
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}
```

4.2.9 Read-only-executables

현재 실행중인 exectable file을 수정할 수 없게 하기 위해서, filesys/file.h의 file_deny_write를 이용했다. 이를 호출하는 것은 Figure 1과 같이 load가 성공하면 현재 실행 파일을 filesys_open하고 이를 file_deny_write하는 것으로 구현하였다. 이 과정에는 process_execute가 load가 끝날때까지 리턴하지 않고 기다리는 것이 필요한데, 이는 semaphore를 이용해서 구현할 수 있었다(다음 코드 참조).
한편 다시 deny를 해제하는 것은 thread_exit()가 실행될 때 file_close를 하는 것으로 해결했다(file_close는 자동으로 deny를 해제해주도록 되어있다).

```
struct thread {
 ...
#ifdef USERPROG
 ...
  struct semaphore exec_lock;
 ...
  /* for Read-Only-Executable. */
  struct file *myself;
 ...
#endif
 ...
}
tid_t
process_execute (const char *cmd_line)
{
  ...
  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid != TID_ERROR) {
  /* wait until load is completed */
      sema_down(&thread_from_tid(tid) -> exec_lock);
  /* Something went wrong while creating.. */
      if (thread_from_tid(tid) -> is_alive == false)
  ...
  }
  ...
```

```
}
static void
start_process (void *file_name_)
{
  ...
  success = load (file_name, &if_.eip, &if_.esp);
  /* If load failed, quit. */
  if (!success)
...
  sema_up(&thread_current() -> exec_lock);
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
}
bool
load (const char *cmd_line, void (**eip) (void), void **esp)
{
  ...
  /* Set up stack. */
  ...
  /* Parse & push arguments. */
  ...
  /* Start address. */
  ...
  success = true;

 done:
  /* We arrive here whether the load is successful or not. */
  file_close (file);
  if (success) {
      /* Denial of writing on myself. */
      t->myself = filesys_open(file_name);
      file_deny_write(t->myself);
  }
  return success;
}
void
thread_exit (void)
{
  ...
#ifdef USERPROG
  ...
  if (thread_current() -> myself != NULL)
  file_close(thread_current()->myself);
  ...
#endif
```

```
  ...
}
```

4.2.10 Resource deallocation

프로세스가 수행을 끝낼 때, thread_exit에서 그동안 이 프로세스가 open하였지만 close하지 않은 file들과 그들을 나타내는 fd들을 list를 순회함으로써 모두 close하고, 동시에 list를 비우고, 메모리 또한 해제하도록 했다. 그후 마지막으로 Read-only-executables를 위해 열었던 file(자기자신) 역시 close함으로써 모든 메모리를 해제하도록 했다.

```
void
thread_exit (void)
{
  ...
#ifdef USERPROG
  ...
  if (!list_empty(&thread_current() -> files)) {
  struct list_elem *item = NULL;
  struct fdesc *entry = NULL;
  while (!list_empty(&thread_current() -> files)) {
  item = list_pop_front(&thread_current() -> files);
  entry = list_entry(item, struct fdesc, elem);
  file_close(entry -> file);
  free(entry);
  }
  }

  process_exit ();
  if (thread_current() -> myself != NULL)
  file_close(thread_current()->myself);
  ...
#endif
  ...
}
```

4.3 시험 및 평가내용

평가는 기본적으로 make check의 pass / fail로 평가했으며, 특정 문제를 해결하기 힘든 경우에는 특정 프로그램을 특정 resource와 함께 시험해보도록 다음과 같은 커맨드를 이용했다.

```
$ pintos -v -k -T 60 --bochs --filesys-size=2 -p tests/userprog/read-normal
-a read-normal -p ./../tests/userprog/sample.txt -a sample.txt -- -q -f run
read-normal
```

pass / fail에 대한 결과는 다음과 같이 모든 테스트에 대해 pass이다. 이 평가 내용들은 각각 개발 목표의 열 부분, 그리고 생산성과 내구성으로 나눌 수 있다.

1a. create(생산성 - 프로그램이 기능하는가)
pass tests/userprog/create-normal

1b. create(내구성 - 프로그램이 예외상황에 대처 가능한가)
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound

2a. open(생산성 - 프로그램이 기능하는가)
pass tests/userprog/open-normal
pass tests/userprog/open-twice

2b. open(내구성 - 프로그램이 예외상황에 대처 가능한가)
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr

3a. close(생산성 - 프로그램이 기능하는가)
pass tests/userprog/close-normal

3b. close(내구성 - 프로그램이 예외상황에 대처 가능한가)
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd

4a. read(생산성 - 프로그램이 기능하는가)
pass tests/userprog/read-normal
pass tests/userprog/read-stdout

4b. read(내구성 - 프로그램이 예외상황에 대처 가능한가)
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-bad-fd
pass tests/userprog/bad-read
pass tests/userprog/bad-read2

5a. write(생산성 - 프로그램이 기능하는가)

```
pass tests/userprog/write-normal
pass tests/userprog/write-stdin

5b. write(내구성 - 프로그램이 예외상황에 대처 가능한가)
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-bad-fd
pass tests/userprog/bad-write
pass tests/userprog/bad-write2

6. 종합적 내구성(프로그램이 극한의 상황 및 예외상황에 대처 가능한가)
pass tests/userprog/multi-child-fd
pass tests/userprog/no-vm/multi-oom
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2

7. Read-only-executables
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild

8. 종합적 생산성(다양한 경우를 통해 프로그램이 기능하는지 확인한다)
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random

9. read 및 write, 그리고 remove의 Synchronization
(내구성 - race condition과 같은 문제에 대처 가능한가)
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
```

4.3.1 보건 및 안정

OS는 기본적으로 어떤 환경에서도 죽어서는 안 된다. 그러므로 모든 system call의 인자로 들어오는 pointer들은 pagedir을 통해 실제 할당된 메모리인지 확인했고, load를 강제로

실패시키는 경우에도 pid를 -1로 리턴시키고 thread를 종료시키도록 처리했다.

5 기타

5.1 inode

아이노드(inode)는 UFS와 같은 전통적인 유닉스 계통 파일 시스템에서 사용하는 자료구조이다. **아이노드는 정규 파일, 디렉터리 등 파일 시스템에 관한 정보를 가지고 있다.** 파일들은 각자 1개의 아이노드를 가지고 있으며, 아이노드는 소유자 그룹, 접근 모드(읽기, 쓰기, 실행 권한), 파일 형태, 아이노드 숫자(inode number, i-number, 아이넘버) 등 해당 파일에 관한 정보를 가지고 있다. **파일시스템 내의 파일들은 고유한 아이노드 숫자를 통해 식별 가능하다.** 일반적으로 파일 시스템을 생성할 때 전체 공간의 약 1퍼센트를 아이노드를 위해 할당한다. 아이노드를 위한 공간이 한정되어 있는 만큼 파일시스템이 가질 수 있는 파일의 최대 개수도 한정되어 있다. 그러나 대부분의 경우, 사용자가 느끼기에 거의 무한개에 가까운 파일을 생성하고 관리할 수 있다.
아이노드에는 보통 다음과 같은 정보가 포함되어있다.
1. 파일 모드: 파일과 관계된 접근과 실행 권한을 저장하는 16비트 플래그.

비트	내용
12-14	파일 형식(일반, 디렉터리, 문자 또는 블록 특별, 선입선출 파이프)
9-11	실행 플래그
8	소유자 읽기 허가
7	소유자 쓰기 허가
6	소유자 실행 허가
5	그룹 읽기 허가
4	그룹 쓰기 허가
3	그룹 실행 허가
2	다른 사용자 읽기 허가
1	다른 사용자 쓰기 허가
0	다른 사용자 실행 허가

Table 1: inode의 파일 모드 플래그

2. 링크 수 : 이 아이노드에 대한 디렉터리 참조 수
3. 소유자 아이디 : 파일의 소유자
4. 그룹 아이디 : 이 파일과 관계된 그룹 소유자
5. 파일 크기 : 파일의 바이트 수
6. 파일 주소 : 주소 정보(39바이트)
7. 마지막 접근 : 마지막으로 파일에 접근한 시각
8. 마지막 수정 : 마지막으로 파일을 수정한 시각
9. 아이노드 수정 : 마지막으로 아이노드를 수정한 시각

5.2 연구 조원 기여도

고창영 : 설계 및 코드 작성, 그리고 보고서(100%).

5.3 느낀점

OS의 원리와 이론들을 직접 구현하면서 실습할 수 있는 기화가 되어 유익했다.
특히 open()의 구현에서 많은 것을 배울 수 있었고 inode와 같은 새로운 개념도 생각해볼 수 있어서 좋은 기회였다.
하지만 test가 획일적이고 적은 것과, pintos의 에러(특히 multi-oom)를 잡기 힘든 것이 좀 아쉽고 힘들었다.