**Pintos Project 2: User Program (2)**

담당 교수 / 분반 : 박성용 교수님/ 1분반

이름 / 학번 : 권형준 / 20161566

개발 기간 : 11/02~11/17

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

프로젝트 2에서는 간단한 파일처리 시스템을 구현하는 것이 주 목표이다. 프로젝트1에서 미리 구현한 system call handler에 create, remove, open, close와 같은 파일을 생성, 삭제, 열고 닫는 system call과 read, write, seek, tell, filesize와 같은 파일 내에서 읽고, 쓰고 검색하는 등의 system call등을 추가해주어야 한다. 구현이 완벽하게 되었을 때, thread에 현재 열려 있는 모든 파일에 대한 추적이 가능해야 되며, read와 write시에 동기화가 돼야 된다. 모든 테스트를 통과하기 위해서, 이러한 기본적인 파일처리 시스템 외에 2가지를 더 구현해야 한다. 한가지는 rox(read only executable로 추정)를 구현하는 것이다. 즉, 현재 실행되고 있는 process의 executable file을 수정하는 것을 막는 것이다. 다른 한 가지는 multi-oom test를 통과하기 위한 메모리 누수를 막는 것이다. Multi-oom test는 재귀함수를 사용하여 계속 child process를 생성하며 이를 실패할 때까지 반복하는 test이다. 적어도 30개는 수행이 가능해야 되는데, 만약 메모리 누수가 발생하면 이 과정이 정상적으로 작동하지 않을 수 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목을 구현해야 하는 이유, 혹은 구현 시 기대되는 결과를 간략히 서술

1. File Descriptor

File descriptor는 프로세스에서 특정 파일에 붙여준 음이 아닌 정수이다. 보통 0,1,2는 stdin, stdout, stderr를 뜻하며 그 외의 숫자는 각 프로세스마다 다를 수 있다. 예를 들어서 같은 파일을 지칭하는 fd라도 프로세스마다 다를 수 있다. 원래 파일처리 시스템에는 file descriptor table이라는 구조체가 프로세스 내에 존재하여 이를 통하여 kernel의 open file table에 접근을 한다. 이후 inode table을 이용하여 실제 파일에 접근하게 된다. 핀토스에서도 이를 배열과 구조체로 비슷하게 구현해 놓았다. 만약 Fd가 없다면, 특정 파일을 지칭하기 위하여 파일 이름과 같은 문자열을 계속 비교하는 등의 방법을 해야 되는데 이는 매우 비효율적이다.

**🡪 프로세스에서 특정 파일로의 빠른 접근 경로를 제공하기 위한 음수가 아닌 정수**

2. (이번 프로젝트에서 구현해야 하는) System Calls

* Create

프로세스 내에서 결과파일과 같은 새로운 파일을 생성해야 될 경우 필요한system call이다. Create system call을 통해서 파일을 생성할 수 있다. 주어진 이름으로 파일을 새로 생성하고 실패한 경우 false를 반환한다.

* Remove

프로세스 내에서 파일을 삭제할 때 필요한 system call이다. 주어진 이름의 파일을 삭제하며, 성공하면 true를 실패하면 false를 반환한다.

* Open

프로세스 내에서 파일을 열 때 필요한 system call이다. 주어진 이름의 파일을 열기 성공했을 때 해당 파일을 반환하고, 실패했을 시에는 NULL 포인터를 반환한다. 이후 프로세스에서 해당 파일에 fd를 할당하여 접근하기 편리하게 한다. (즉, file descriptor table에 추가한다)

* Close

프로세스 내에서 파일을 닫을 때 사용되는 system call이다. 주어진 file descriptor에 해당하는 파일을 닫고 프로세스에 저장되어 있는 fd의 값을 NULL로 바꿔준다. 즉, 더 이상 프로세스에서 파일에 접근할 수 있는 경로가 없게 된다.

* Filesize

프로세스 내에서 파일의 크기를 읽어올 때 필요한 system call이다. 주어진 file descriptor에 해당하는 파일의 전체 크기를 반환한다. 프로젝트 test 중 read-normal과 같이 check\_file\_handle함수를 사용하는 test에서 이 system call을 사용하기 때문에 구현할 필요가 있다.

* Read

프로세스 내에서 파일의 내용을 읽어올 때 사용한다. 기존에는 stdin(fd=0)인 경우에만 읽을 수 있었지만 이제는 해당하는 file descriptor에서 지정한 size만큼 byte를 읽어올 수 있다. 읽는 도중 새로 write가 이루어지면 안됨으로 write와 동기화가 필요하다.

* Write

프로세스 내에서 파일에 새로운 내용을 기록할 때 사용된다. 기존에는 stdout(fd=1)인 경우에만 쓸 수 있었지만, 이제는 해당하는 file descriptor에 지정한 size만큼 byte를 기록할 수 있다. Open과 read가 진행되는 중이라면 write는 실행될 수 없음으로 동기화 과정이 필요하다.

* Seek

File descriptor에 해당하는 파일의 pos(위치)를 지정한 byte offset으로 옮길 때 필요한 system call이다. 파일의 pos는 file\_read, file\_write 등 해당하는 파일에 읽거나 쓸 때 사용하는 file pointer와 같은 역할을 수행한다.

* Tell

File descriptor에 해당하는 파일의 pos(위치)를 반환한다. 앞에서 설명한 file pointer와 같은 역할을 수행하는 pos의 위치가 몇 byte offset만큼 떨어져 있는지를 알려준다.

3. Synchronization in Filesystem

앞에서 구현해야 되는 system call에 대한 설명 중 write, read의 system call은 동기화(synchronization)이 필요하다고 설명하였다. 만약 read하는 도중 write가 발생하면 실행되는 thread의 순서(scheduling algorithm)에 따라서 매번 읽고 쓸 때마다 그 결과가 다를 수 있다. 이는 흔히 말하는 Readers-Writers problem을 발생시킨다. 따라서 읽는 도중에는 write를 금지시키고, write 도중에는 read와 write를 금지시키는 작업이 필요한데, 이를 semaphore와 lock을 사용하여 구현할 수 있다. 구현이 완료되면 동시에 read하는 것은 허락이 되어도 write는 오직 하나의 thread만 가능하게 된다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

각각의 thread에서 열 수 있는 파일의 수는 128개로 제한이 되어 있는 것을 pintos manual에서 읽었다. 따라서 file 구조체(struct file)의 포인터를 128개 포함하는 배열 fd를 선언하여 사용하였다. (아래는 struct thread내에 포함되는 자료구조이다.)



이때, fd=0,1,2는 앞에서 설명한 것처럼 stdin, stdout, stderr을 위해 남겨놓고 실질적으로는 fd[3]부터 사용할 수 있게 구현을 해 놓았다.

2. System Calls: 구현할 각 system call에 대해 간략히 서술 (하나의 system call 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

* Create

인자로 넘겨 받은 file의 이름이 NULL이면 exit(-1)을 호출한다. Filesys\_create api를 사용하여 해당하는 이름의 file을 생성한다. 성공 여부를 bool로 반환한다.

* Remove

인자로 넘겨 받은 file의 이름이 NULL이면 exit(-1)을 호출한다. Filesys\_remove ap를 사용하여 해당하는 이름의 file을 삭제한다. 성공 여부를 bool로 반환한다.

* Open

인자로 넘겨 받은 file의 이름이 NULL이면 exit(-1)을 호출한다. Filesys\_open을 사용하여 해당하는 이름의 파일을 열고, 현재 thread에 맞는 file descriptor를 할당한다. 할당 받은 file descriptor를 반환한다.

* Close

인자로 넘겨 받은 file descriptor에 해당하는 file을 file\_close api를 사용하여 닫는다. 이후, 사용하고 있던 file descriptor를 다른 파일이 사용할 수 있게 file descriptor 배열의 파일 포인터를 NULL로 바꾼다. 현재 thread가 넘겨 받은 file descriptor에 해당하는 파일이 없다면 exit(-1)을 호출한다.

* Filesize

File\_length api를 사용하여 인자로 넘겨 받은 file descriptor에 해당하는 파일의 크기를 반환한다. 만약 file descriptor에 대응하는 파일이 없다면 exit(-1)을 호출한다.

* Read

앞에서는 stdin으로만 입력을 받을 수 있었던 read에 file\_read api를 사용하여 다른 fd도 인식할 수 있게 한다. 만약 읽고자 하는 파일이 없다면 exit(-1)을 호출한다. Readers-writers-problem을 해결하기 위하여 semaphore와 lock을 사용한다.

* Write

앞에서는 stdout으로만 출력을 내보낼 수 있었던 write에 file\_write api를 사용하여 다른 fd도 인식할 수 있게 한다. 만약 쓰고자 하는 파일이 없다면 exit(-1)을 호출한다. Readers-writers-problem을 해결하기 위하여 semaphore를 사용한다.

* Seek

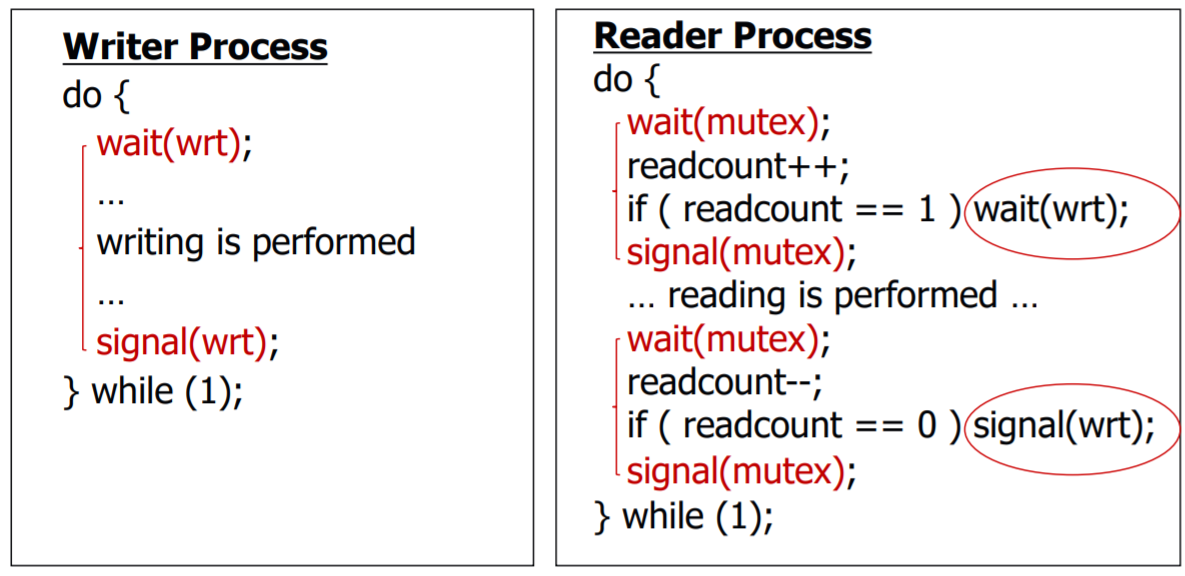
File\_seek api를 사용하여 인자로 넘겨 받은 file descriptor에 해당하는 파일의 position(file pointer)를 원하는 위치로 옮긴다. 만약 file descriptor에 대응하는 파일이 없다면 exit(-1)을 호출한다.

* Tell

File\_tell api를 사용하여 인자로 넘겨 받은 file descriptor에 해당하는 파일의 position의 위치를 반환한다. 만약 넘겨 받은 file descriptor에 대응하는 파일이 없다면, exit(-1)을 호출한다.

3. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명 (다른 방법을 서술해도 되지만 lock과 semaphore는 반드시 포함해야 함)

Lock의 경우 acquire을 한 thread에서만 release를 할 수 있으며, mutex lock과 같이 0과 1의 값 만을 가질 수 있다. 이에 다르게 semaphore는 정수의 값을 가질 수 있으며 하나의 thread가 down을 실시한 semaphore에 다른 thread가 up시킬 수 있다. 이를 바탕으로 수업시간에 배운 readers-writers-problem 해결 방안을 그대로 적용하였다. readcount라는 변수를 사용하여 현재 파일을 읽고 있는 thread가 몇 개인지 추적하고, Read의 경우 처음 mutex lock을 걸어 readcount값을 증가시키고 wrt라는 semaphore를 down시키고 mutex lock을 release했다. 이후 read의 critical section이 끝나면 다시 mutex lock을 걸어 readcount변수의 값을 1 감소시키고 이 값이 0이라면 wrt라는 semaphore을 up시켜 주었다. 이 내용을 그림으로 표현하면 다음과 같다. (운영체제 6-7장 수업자료)



프로젝트에서 구현을 할 때는 mutex semaphore와 같은 역할을 하는 lock 구조체를 사용하였다. 하지만 이러한 synchronization 구현은 몇 가지 의문을 들게 하였다. 이러한 점은 뒤의 “연구 결과-제작 내용”에서 더 추가로 설명하겠다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

|  |  |
| --- | --- |
|  | **개발 범위** |
| **11/02~11/05** | **File descriptor, system call(create, remove)** |
| **11/05~11/10** | **System call(read, write, open, …)** |
| **11/10~11/15** | **Synchronization** |

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명

1. **File descriptor**
   * 수정하거나 추가해야 하는 자료구조
     + Struct thread <Threads/thread.h>

“Struct file\* fd[128]”이라는 포인터 배열을 추가해야 된다. 이는 각 thread마다 열 수 있는 파일의 수를 128개로 제한한 것이며, pintos manual에 나온 그대로 구현하였다.

* + 수정해야하는 소스코드
    - Open <Userprog/syscall.c>

처음 파일을 여는 것을 성공했을 때, 위에서 구현한 포인터 배열에 빈 공간을 찾아서 새로 연 파일을 할당해준다. 만약 할당할 수 있는 fd가 없다면 -1을 반환한다.

* + - Close <Userprog/syscall.c>

파일을 닫을 때, 해당하는 파일의 포인터 배열을 NULL로 바꿔준다. 즉, 다음에 다른 파일을 열었을 때 사용할 수 있게 fd값을 free해준다.

* + - Exit, read, write, etc <Userprog/syscall.c>

그 외에 fd 포인터 배열을 사용하는 함수들이다. 보통 해당 thread에서 해당하는 file descriptor가 가리키는 파일이 있는지 확인하는 용도로 사용이 된다.

1. **System call**
2. Create, Remove
   * 수정해야하는 소스코드

🡪 syscall\_handler <userprog/syscall.c>

Syscall\_handler에서 System call number에 따라 만약 4(=SYS\_CREATE), 5(=SYS\_REMOVE)라면 인자를 추출하여 create 또는 remove 함수를 호출한다.

* + 수정하거나 추가해야 하는 함수

🡪 create, remove <userprog/syscall.c>

함수가 받는 인자는 filename을 포함하고 있다. File name이 NULL이 아니라면 pintos에 미리 구현되어 있는 api를 사용하여 파일을 생성하거나 삭제한다. 성공하면 True를 실패하면 False를 반환한다.

1. Open, Close
   * 수정해야하는 소스코드

🡪 syscall\_handler <userprog/syscall.c>

Syscall\_handler에서 System call number에 따라 만약 6(=SYS\_OPEN), 12(=SYS\_CLOSE)라면 인자를 추출하여 open 또는 close 함수를 호출한다.

* + 수정하거나 추가해야 하는 함수

🡪 open, close <userprog/syscall.c>

Open 함수의 경우 file의 이름을 받아서 pinots의 api를 사용하여 해당하는 파일이 있다면 열고 없다면 -1을 반환한다. 앞에서 구현한 file descriptor를 새로 연 파일을 할당한다. Close의 경우 인자로 file descriptor을 받게 되고, 해당 file descriptor에 해당하는 파일이 없다면 exit(-1)을 호출하고, 정상적으로 파일이 닫혔다면 사용하고 있던 file descriptor는 NULL로 초기화 한다.

1. Read, Write
   * 수정하거나 추가해야 하는 함수

🡪 read, write <userprog/syscall.c>

앞에서는 fd가 0(=stdin), 1(=stdout)인 경우에만 사용이 가능하였다. 이를 확장하여 thread에서 연 파일에 read와 write를 가능하게 한다. 이 과정에서 pintos에서 구현되어 있는 api(file\_write, file\_read)를 사용한다. Readers-writers-problem을 위한 synchronize과정이 필요하며 이는 뒤에서 자세히 설명하겠다.

1. Filesize, seek, tell
   * 수정해야하는 소스코드

🡪 syscall\_handler <userprog/syscall.c>

Syscall\_handler에서 System call number에 따라 만약 7(=SYS\_FILESIZE), 10(=SYS\_SEEK), 11(=SYS\_TELL)라면 인자를 추출하여 filesize, seek, tell 함수를 호출한다.

* + 수정하거나 추가해야 하는 함수

🡪 filesize, seek, tell <userprog/syscall.c>

세 함수 다 인자로 file descriptor를 받으며 (seek만 position이라는 인자를 추가로 받는다) 미리 구현되어 있는 pintos api를 사용하여 해당 system call을 수행한다.

1. **Synchronization**
   * 수정하거나 추가해야 하는 자료구조 (userprog/syscall.c)
     + Struct semaphore wrt

현재 파일에 write system call이 수행될 수 있는지를 표현하는 semaphore이다. 만약 wrt가 0보다 작거나 같으면 write는 수행될 수 없다. Wrt를 조정할 수 있는 함수는 sema\_down, sema\_up, sema\_init 뿐이다.

* + - Struct lock mutex

read함수에서 critical section을 표현하기 위해서 사용하는 lock이며, mutex lock으로 오직 0과 1의 값을 가지는 semaphore와 똑같이 작동한다. 즉, 모든 thread 중 오직 한 thread만이 mutex lock으로 나뉘어 있는 critical section에 접근할 수 있다. Lock\_acquire와 lock\_release를 사용하여 조정한다.

* + - Int readcount

현재 read를 하고 있는 thread의 수를 표현한다. 이 값에 따라서 wrt semphore을 조정할 수 있다. Readers-writers-problem에서 하나라도 read가 있는 경우 write는 발생할 수 없는 경우를 구현하기 위하여 사용된다.

* + 수정하거나 추가해야 하는 함수 (userprog/syscall.c-read, write함수)

파일을 read, write하는 critical section이 있다. 이 부분에서는 file\_Read, file\_write와 같은 api를 사용한다. Read의 경우 다른 read와 동시에 발생할 수 있지만, write는 다른 write 또는 read가 발생하고 있을 시 동시에 발생할 수 없다. 이를 위하여 semaphore와 lock을 사용하여 구현한다.

* + - Read 함수

readcount라는 변수는 현재 파일을 읽고 있는 thread를 나타낸다. 각 thread에서 readcount라는 변수를 조작하는 것 또한 synchronize가 필요하다. 이 경우 겹치는 thread가 존재하면 안됨으로 mutex lock을 사용하여 critical section을 보호한다. 이 구간을 지나면 wrt는 sema\_down이 되어 다른 write가 파일에 대한 critical section의 접근을 금지한다. 파일을 읽는 중이던 thread가 파일 접근에 관한 critical section을 나가 wrt를 sema\_up을 해준 뒤, 다른 write가 실행될 수 있다.

* + - Write 함수

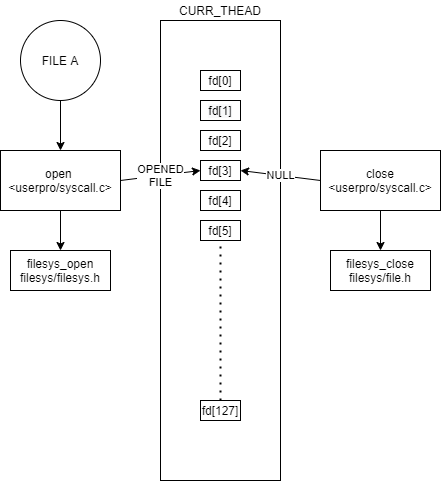
파일에 접근하는 critical section처음에 wrt를 sema\_down하고 write를 완료한 이후 sema\_up을 한다. Sema\_down이 되어있는 경우 다른 write나 read는 파일에 접근하는 critical section에 들어올 수 없어 synchronization이 구현된다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성

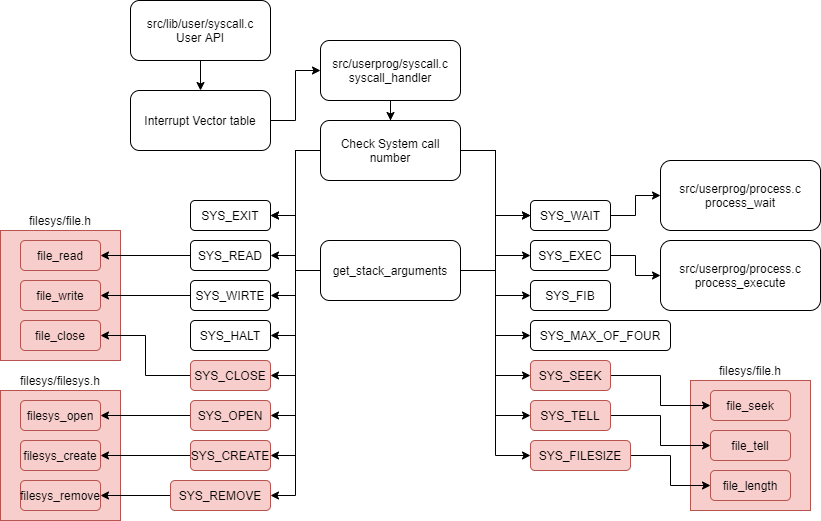
1. **File descriptor**

아래 file descriptor에 대한 흐름도는, system call 중 open과 close를 실행하면서 thread내부의 fd배열에 할당하여 다른 system call에서 사용할 수 있게 되는 과정을 설명한 것이다.

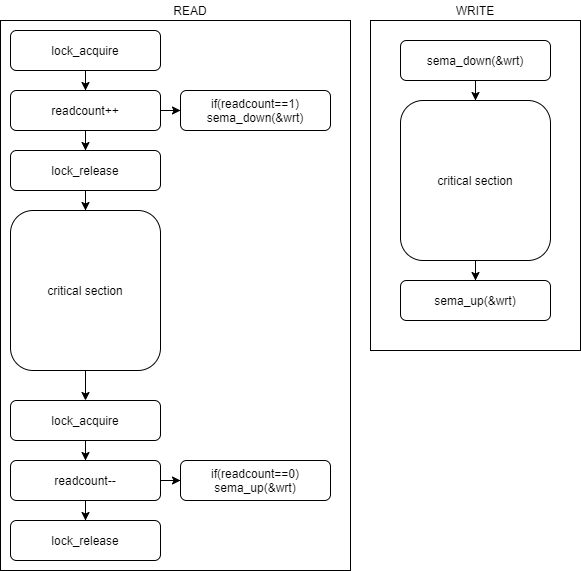


1. **System call**

**(새로 구현한 부분은 붉은색, 그 외는 project1에서 구현한 것)**



1. **Synchronization (userprog/syscall.c의 read, write 함수에 관하여)**



* 1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)

1. File descriptor

각 thread마다 “struct file\* fd[128]”이라는 파일 포인터 배열을 선언한다. 파일을 새로 열 때 이 배열의 빈 file descriptor을 찾아서 파일 포인터를 새로 연 파일에 연결한다. 파일을 닫을 때, 이 포인터를 NULL로 초기화한다. 그 과정을 open과 close의 관점에서 따라가면 다음과 같다. System call number로 SYS\_OPEN이 들어오면 다른 system call들과 마찬가지로 user stack에서 argument를 읽어온다. 이 경우 열고자 하는 file의 이름을 읽어오게 된다. 이를 open 함수로 넘겨준다. 읽어온 file name이 NULL인 경우 exit(-1)을 호출하여 오류를 알린다. 만약 제대로 된 file name이라면 해당하는 file name의 file을 filesys\_open함수를 사용하여 연다. (pintos에서 미리 구현되어 있는 함수는 밑에서 다시 설명) 해당하는 파일이 없거나, 메모리 할당 실패로 파일을 여는데 실패하면 -1을 반환한다. 이후 파일이 정상적으로 열렸으면 file descriptor을 할당하게 된다. 0, 1, 2는 stdin, stdout, stderr을 위해 사용할 수 없음으로 fd[3]부터 검색을 하여 NULL, 즉 사용하지 않는 file descriptor에 연 파일을 넣는다. 이후 새로 생성한 file descriptor을 반환한다.

다음은 close system call이 호출될 경우, 인자로는 file descriptor을 입력 받는다. File descriptor에 대응하는 file이 존재하는지 확인하고, 없다면 exit(-1)을 호출한다. 이후 미리 구현된 (file\_close)를 사용하여 파일을 닫고 file descriptor에 대응하는 포인터를 NULL을 초기화한다.

이렇게 구현한 file descriptor는 filesize, seek, tell등과 같은 system call에서 사용이 된다.

1. System call
   * Create

System call handler에 System call number로 SYS\_CREATE가 들어오면 create함수를 호출한다. 이때, 인자로 들어오는 것은 생성할 파일의 이름과 초기 크기이다. 만약, 파일의 이름이 NULL이라면 exit(-1)을 호출한다. 미리 구현되어 있는 filesys\_create함수를 이용하여 파일을 생성하고, 성공 시 true를 실패 시에는 false를 return한다. 이를 eax레지스터에 반환한다.

* + Remove

System call handler에 system call number로 SYS\_REMOVE가 들어오면 remove함수를 호출한다. 이때, 인자로 들어오는 것은 삭제할 파일의 이름이다. 만약 파일의 이름이 NULL이라면 exit(-1)을 호출한다. Filesys\_remove함수를 사용하여 파일을 삭제하고 성공하면 true를 실패하면 false를 return한다.

* + Open

Open에 관한 대부분의 구조는 앞에 file descriptor을 설명하면서 하였기 때문에 file descriptor와 관련 없는 부분만 설명하면 다음과 같다. 원래 프로그램은 load되고, execute된 뒤에 executable file을 삭제, 수정하여도 문제가 없다. 하지만 pintos는 이를 금지하라고 되어 있으며, rox-simple, rox-child, rox-multichild에서 이를 검사한다. (filesys/file.c)에 구현되어 있는 file\_deny\_write은 이를 쉽게 구현할 수 있게 해준다. thread내에서 파일을 새로 여는 open에서 만약 현재 돌고 있는 thread의 이름과 file의 이름이 같다면 file\_deny\_write을 하여 해당 파일에 write하는 것을 금지한다. 이 점이 어떻게 작동하는지는 밑에 미리 구현된 pintos함수에서 추가로 설명하겠다. 이후에 file\_close를 하게 되면 자동으로 file\_allow\_write이 실행된다.

* + Close

Close는 file descriptor에서 설명한 것과 똑같다.

* + Read

System call number가 SYS\_READ로 들어온 경우, user stack에서 argument를 file descriptor, buffer, buffer의 size를 입력 받는다. Buffer의 주소가 유효한지 확인하는 check\_buffer를 실행하고, file descriptor에 해당하는 파일이 존재하는지 확인한 이후에 file\_read를 사용하여 원하는 내용을 읽어온다. (check\_buffer, file\_read, synchronization에 관한 내용은 밑에서 추가로 설명한다)

* + Write

System call number가 SYS\_WRITE로 들어온 경우, user stack에서 argument를 3개 읽어온다. 사용할 파일의 file descriptor, 파일에 적을 buffer, 그리고 buffer의 크기를 읽어온다. 이전에 구현한 Write에 file descriptor에 해당하는 파일이 존재하는지 확인하는 코드를 추가하고 file\_write함수를 사용하여 파일에 적는다. 인자로 넘기기 전에 buffer의 모든 주소가 유효한지 확인하는 check\_buffer를 사용한다. (이는 밑에서 추가로 설명한다.) 또한 synchronization에 관한 내용도 밑에서 설명한다.

* + Filesize, seek, tell

3개의 system call은 각각의 system call number에 따라서 함수(filesize, seek, tell)을 호출한다. Filesize의 경우 하나의 인자, file descriptor만 입력 받은 후 해당하는 파일이 존재하지 않으면 exit(-1)을 호출하고 존재한다면 미리 구현된 file\_length를 호출한다. Tell의 경우도 이와 동일하며 file\_tell을 호출한다. Seek의 경우 file descriptor와 함께 파일 내로 file pointer를 옮길 위치를 담고 있다.

1. Synchronization

Synchronization은 Readers-writers-problem을 해결하기 위하여 system call read, write에서 구현을 하였다. 하나의 semaphore와 하나의 lock(mutex lock)을 사용하여 파일에 접근하는 critical section에는 동시간에 오직 하나의 write만 들어올 수 있게 한다. 코드의 흐름은 위의 흐름도와 동일하다. Read의 경우 mutex lock을 걸고 readcount(read 중인 thread의 수)를 증가시킨다. 만약 readcount가 1이라면 wrt semaphore를 down시킨다. 이후 다시 mutex lock을 푼다. Read가 끝나면 다시 mutex lock을 걸고 readcount를 1 감소시킨다. 만약 readcount의 값이 0이라면 wrt를 semaphore up 시킨다. 이후에 lock을 푼다. Write의 경우 처음 시작할 때 wrt semaphore down시키고 끝나면 semaphore up시킨다. 이러한 구조가 어떻게 synchronization을 구현하게 되는지 4가지 경우로 나누면 다음과 같다.

* + - Read 중 read가 들어온 경우

Mutex\_lock으로 구분되는 critical section은 겹치지 않지만 실제로 파일에 접근하는 critical section은 다수의 read thread가 겹칠 수 있다. Readcount가 계속 증가한다.

* + - Read 중 write가 들어온 경우

Read가 critical section으로 들어오면서 sema\_down(&wrt)를 했기 때문에 write의 시작에 있는 sema\_down(&wrt)에서 wait를 하게 된다.

* + - Write 중 read가 들어온 경우

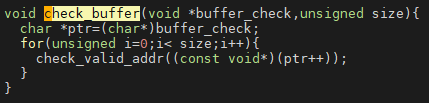
Write의 시작에 있는 sema\_down(&wrt) 때문에 read 초반에 readcount가 1일 때 sema\_dwon(&wrt)에서 wait를 하게 되어 read를 할 수 없게 된다.

* + - Write 도중 write가 들어온 경우

Wrt semaphore로 인해 critical section이 보호된다.

* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
  + Check\_buffer (userprog/syscall.c)

새로 구현한 함수이다. System call 함수 중 read와 write에서 사용하는 buffer가 실제 사용 가능한 주소인지 확인한다. 프로젝트 1에서 구현한 check\_valid\_addr(해당하는 주소가 NULL이 아니며, user address이며, 0x08048000보다 큰지, 즉 유효한 주소인지 확인하는 함수이다).



* + Filesys\_create (filesys/filesys.c)

인자로 들어온 문자열을 이름으로 가지는 파일을 생성한다. 해당하는 파일의 inode를 생성하는 과정을 담고 있으며, root directory에 새로운 파일을 추가하는 과정이 포함되어 있다. 만약 실패 시 false를 반환한다.

* + Filesys\_remove (filesys/filesys.c)

인자로 들어온 이름에 해당하는 파일을 찾아서 삭제한다. Filesys/directory.c에 구현되어 있는 dir\_remove함수를 호출하여 사용한다. 삭제 성공시에 true를 실패시에 false를 반환한다.

* + Filesys\_open (filesys/filesys.c)

인자로 들어온 이름에 해당하는 파일을 root directory에서 찾는다. 없다면 NULL을 반환한다. 성공 시 file\_open함수를 호출하여 찾은 파일의 inode에 대한 정보를 읽어서 파일을 새로운 file 구조체에 담는다. 생성한 file 구조체에 대한 포인터를 반환한다.

* + File\_close (filesys/file.c)

File 구조체에 대한 포인터를 받아와서 NULL이 아니라면 file\_allow\_write를 호출하고 inode\_close함수를 호출한 뒤, file 구조체를 free한다.

* + File\_read (filesys/file.c)

인자로 들어온 파일 구조체의 포인터에서, 구조체 내부의 변수인 position이 가리키는 위치로부터 size만큼 bytes를 읽어서 buffer에 저장한다. 이후 position을 size만큼 뒤로 이동시킨다. 마지막으로 총 읽은 byte 수만큼 반환한다.

* + file\_write (filesys/file.c)

인자로 들어온 파일 구조체의 포인터에서, 구조체 내부의 변수인 position이 가리키는 위치로부터 size만큼의 bytes를 buffer에 적힌 내용을 적는다. 이후, position을 size만큼 뒤로 이동시킨다. 마지막으로 총 쓴 byte 수만큼 반환한다.

* + file\_length (filesys/file.c)

파일이 NULL이 아니라면 inode\_length함수를 사용하여 파일의 크기를 반환한다. 이 함수는 inode구조체 내에 저장되어 있는 data항목의 length를 반환한다.

* + File\_seek (filesys/file.c)

파일이 NULL이 아니고, 새로운 위치가 0보다 크거나 같다면 file 구조체 내부에 position을 나타내는 변수에 새로운 위치를 대입한다.

* + File\_tell (filesys/file.c)

파일이 NULL이 아니라면 file 구조체 내부에 position을 나타내는 변수의 값을 반환한다.

* + File\_deny\_write (filesys/file.c)

File이 NULL이 아니라면, file 구조체 내부에 deny\_write변수를 true로 설정한다. 그 뒤에 inode\_deny\_write를 호출한다. 즉, 이후에 file\_allow\_wirte가 나올 때까지, 파일의 inode에 write를 금지한다.

* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1. Syn\_read와 syn\_write가 synchronization을 구현하지 않아도 작동을 하는 점을 발견하였다. Syn-read.c, syn-write.c, child-syn-read.c, child-syn-write.c등의 코드를 열어본 결과 synchronization을 확인할 수 있는 코드는 아니었다. Synchronization을 test하기 위해서는 10개의 read, write 하나만 하는 child가 아닌 둘을 번갈아 하는 child를 생성해야 될 것이다. 오히려 syn\_read는 child thread가 메모리에 load되기 전에 parent의 execute가 끝나지 않는지 확인하는 check\_child\_load test가 알맞지 않나 싶다.
2. Filesys.h, file.h을 수정하지 않는다는 조건으로 인하여, semaphore등의 동기화 도구들은 system call handler에 오직 한 개를 할당해주었다. 이렇게 구현하게 되면 조금 비효율적인 작동이 예상된다. 예를 들어서 a라는 파일에 쓰는 도중에 b라는 파일은 읽을 수 있지만, b라는 파일에 쓰지는 못한다. 하지만 이는 실제로 아무런 문제도 생성하지 않는 상황이다. 이러한 점은 시스템에 문제를 발생시키지 않지만, 작은 시간도 아끼기 위하여 busy waiting등을 하는 상황에서 critical section하나를 통째로 (필요 없이)기다린 다는 것은 매우 비효율적이다. Lass Forum에 다른 학우의 질문과 조교님의 답변을 통하여, 다른 시스템에서는 inode에 semaphore등을 저장한다는 것을 알게 되었다. 이를 pintos에서 구현할 경우 inode.h, inode.c, filesys.c, file.c 등 파일을 열고, 읽는 함수들을 다 고쳐야 된다.
   1. **시험 및 평가 내용**

* make check 수행 결과를 캡처하여 첨부

