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

담당 교수 / 분반 : 김영재 1반

이름 / 학번 :이호성 20171680

개발 기간 :10.3~10.4

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트2는 프로젝트1에서 기본적인 system call을 구현한 것에 이어 filesystem과 관련된 system call을 구현하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor : STDIN STDOUT 이외에도 다양한 input output resource들도 활용할 수 있어야 하기 때문에 file descriptor를 이용하여 다양한 resource들을 활용할 수 있게 구현한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls : 이번 프로젝트에서는 create remove open close filesize read write seek tell을 구현해야 한다. 이 system call을 통해 다양한 IO resource 들을 다룰 수 있게 구현한다.

3. Synchronization in Filesystem : 한 File 을 수정하고 있는 상태에서 그 파일을 참조하면 잘못된 정보가 전달 될 수 있기 때문에 수정을 하는 상태에서는 다른 스레드들이 그 파일을 참조할 수 없게 만들어야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술 : 가장 간단한 array를 구현하였다. pintos manual 에 따르면 최대 128개의 fd 가 들어올 수 있다고 되어 있으므로 구현이 쉬운 array로 선택하였다

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)

int open(const char \*file): 인자로 넘어온 ‘file’ 이라는 파일을 open한다.

void close(int fd): file descriptor 중 fd를 닫는다.

int filesize(int fd): file descriptor 중 fd 로 열린 파일의 size를 반환한다.

int read(int fd,void \*buffer, unsigned size): file descriptor 중 fd에 해당하는 파일을 size만큼 읽어 buffer로 넘긴다 그후 실제로 읽은 byte를 반환한다..

int write(int fd, const void \*buffer, unsigned size): file descriptor 중 fd에 해당하는 파일에 buffer를 size 만큼 write한다. 실제로 적힌 byte 수를 return 한다.

void seek(int fd, unsigned position): file descriptor 중 fd로 열린 파일의 다음 읽거나 쓰일 다음 byte를 position으로 변경한다.

unsigned tell(int fd): file descriptor 중 fd로 열린 파일의 다음 읽거나 쓸 다음 위치의 바이트를 반환한다.

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

한 파일을 한 스레드가 read/write를 할 때 다른 스레드가 접근(open)을 하게 되면 문제가 발생하므로 write/read/open을 시작할 때 lock을 걸고 종료할 때 lock을 회수하게 구현한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

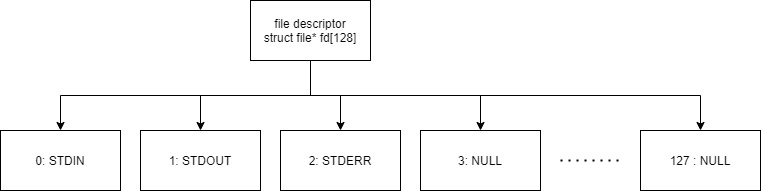
10.3 file descriptor 생성, open read write를 제외한 system call 구현

10.4 open read write system call 구현. synchronization in filesystem 구현

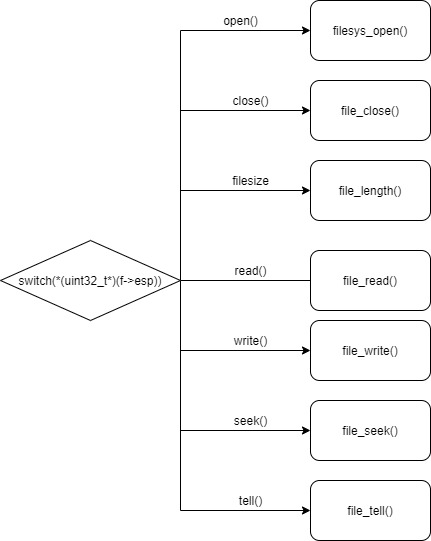
* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드: 메모리 누수 방지를 위해 자식 프로세스가 생성되기도 전에 부모 프로세스가 죽는 일은 없어야 하기 때문에 thread\_create()가 있는 process\_execute()와 생성되는 단계인 process\_start()를 수정해야한다.
  + 수정하거나 추가해야 하는 자료구조: struct thread에 file descriptor를 추가하고 read\_write lock을 생성해야 한다. 그리고 test case에 있는 누수메모리를 위한 lock과 parent thread를 알 수 있는 변수도 추가하여야 한다.
  + 수정하거나 추가해야 하는 함수: open close filesize read write seek tell 함수를 추가하여야 한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
  + 1. file descriptor



* + 1. System Calls



* + 1. Synchronization in File systems



* 1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1. File descriptor **:** array로 구현할 것이므로 struct thread 에 struct file\* fd[128]을 추가한다.

2. System Calls :

우선 (int fd)와 같이 file descriptor를 인자로 받는 system call 함수의 경우 닫으려는 fd가 존재하지 않으면 exit(-1)을 해준다. (const char \*file)과 같이 파일명을 인자로 받는 함수의 경우 해당 이름의 파일이 존재하지 않으면 exit(-1)을 하는 부분을 추가해준다.

**int open(const char \*file):** 인자로 넘어온 ‘file’ 이라는 파일을 struct file\* fp = filesys\_open()을 통해 접근한다. 비어 있는 file descriptor의 자리를 찾은 다음 할당한다. 그러곤 할당된 file\_descriptor의 번호를 반환한다.

**void close(int fd):** fd[fd]의 값을 NULL 로 바꿔주고 file\_close()를 호출한다.

**int filesize(int fd):** fd 로 열린 파일의 size를 file\_length()를 통해 반환한다.

**int read(int fd,void \*buffer, unsigned size**): file descriptor 중 fd에 해당하는 파일을 size만큼 읽어 buffer로 넘긴다 그후 실제로 읽은 byte를 반환한다. STDIN(fd == 0)인 경우 프로젝트 1에서 구현한 것과 같이 input\_getc()를 사용하며 다른 file descriptor들의 경우 file\_read()를 사용하여 읽은 바이트수를 반환한다.

**int write(int fd, const void \*buffer, unsigned size):** file descriptor 중 fd에 해당하는 파일에 buffer를 size 만큼 write한다. 실제로 적힌 byte 수를 반환한다. STDOUT(fd == 1)의 경우 putbuf()를 이용하여 작성한 바이트 수를 반환한다. 다른 file descriptor들의 경우 file\_write()를 활용하여 작성한 바이트 수를 반환한다.

**void seek(int fd, unsigned position**): fd로 열린 파일의 다음 읽거나 쓰일 다음 byte를 position으로 file\_seek()를 통해 변경한다.

**unsigned tell(int fd):** file descriptor 중 fd로 열린 파일의 다음 읽거나 쓸 다음 위치의 바이트를 file\_tell()을 통해 반환한다.

3. Synchronization in Filesystems.

File systems에서 구현할 synchronization은 한 thread 가 file에 read/write/open를 하고 있을 때 다른 스레드가 read()/write()/open()를 하게 되면 수정중인 정보에 접근을 하여 잘못된 정보를 읽게 되는 상황이 발생한다. 그러기 위해서는 우선 rw\_lock을 만들어 read()/write()/open()를 시작하면 lock\_acquire()를 통해 lock을 습득하고 read()/write()/open()가 끝나면 lock\_release()를 통해 lock을 반환한다.

이에 추가로 Denying Writes to Executable files 에 관하여 file\_deny\_write() / file\_allow\_write() 를 사용하라고 되어있다. write()/read()를 하려면 open()을 해야 하므로 open()에 file\_deny\_write()를 close에 file\_allow\_write()를 하면 될 것 같았다. 확인해보니 close()에서 활용하는 file\_close(fp)에서 file\_allow\_write()를 해주기 때문에 open에만 file\_deny\_write()하는 부분을 추가하였다.

여기까지 완료하였을 때 make check를 해보면 multi-oom, syn-read, syn-write 이 안되었다. synchronization은 맞게 구현한 것 같은데 fail이 뜨는 관계로 multio-oom을 해결하면 syn test 도 해결할 수 있을 것 같아 multi-oom을 해결하기로 하엿다. multi-oom의 경우 자식 thread가 load가 되기 전에 parent 가 끝나버린 경우 parent가 자식을 wait 하지 못하기 때문에 이부분에 대하여 semaphore를 사용하여 lock을 걸어야 할 것 같았다. 그래서 struct thread에 oom\_lock을 추가하고 init\_thread()에 이를 초기화하는 부분을 추가한다. 그 다음 thread\_create() 가 있는 process\_execute 부분으로 가 thread\_create(…,start\_process,…)를 한 다음 sema\_down(&thread\_current()->oom\_lock)을 추가해 준다. 그 다음 start\_process 로 가 자식 프로세스가 load가 다 되엇다면 sema\_up(&thread\_current()->oom\_lock)을 하여 부모 프로세스가 진행 될 수 있게 한다. 그리고 테스트 케이스에서 load를 실패 하였다면 -1을 기대하였다고 하여 (!success) 다음 thread\_exit() 을 exit(-1)로 변경 해 준다. 그럼에도 안 되었는데 이는 exit(-1)로 종료된 스레드가 parent에서 정상적으로 안 지워졌기 때문인 것 같다. 그러므로 process\_execute의 마지막 종료하기 전 thread\_current의 자식들을 검사하여 exit\_status 가 -1 이면 process\_wait()을 이용하여 강제종료 된 child\_thread 도 회수하도록 하였다. 여기까지 완료하였더니 앞서 언급했던 syn test들까지 해결되며 모든 테스트 케이스가 pass하였다.

* 1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부
* 