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

담당 교수 / 분반 : 김영재 - 2분반

이름 / 학번 : 장수길 - 20161634

개발 기간 : 2021/10/1 - 2021/10/11

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

운영체제 프로젝트 2 에서는 지난 프로젝트 1에 이어 각종 시스템 콜들을 구현하고 File Descriptor들로 구성된 pintos의 기본적인 file system을 구현하는 것을 목표로 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

각 프로세스에 대해서 file descriptor table을 구현하고 파일 open 및 close 할 경우 file descriptor을 반환하는 것을 통해 핀토스는 표준 입출력을 포함한 파일 시스템에 접근하는 것이 가능하게 된다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

프로젝트2에서구현해야 할 시스템 콜들은 다음과 같다:  
create, remove - 파일 생성 및 삭제  
open, close - 파일 열기 및 닫기  
filesize - 파일 크기   
read, write - 파일 읽기 쓰기   
seek, tell - 파일 포인터 조정

3. Synchronization in Filesystem

동일한 파일에 여러 개의 thread가 동시에 접근할 경우 생길 수 있는 동기화 문제를 해결한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

File descriptor의 경우, 배열을 사용하였다. Linear한 자료구조의 후보로 연결 리스트 및 배열이 있었는데, 가장 친숙하고 메모리 누수 문제에 있어서 비교적 자유로운 배열 자료구조를 선택하여 사용하였다. 또한 file descriptor의 인덱스 값이 0 이상의 자연수로 한정되어있었기 때문에 배열로 관리하는 것이 용이하다고 판단하였다.

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: 파일을 생성하는 system call이다.   
remove: 파일을 삭제하는 system call 이다.  
open: 파일 포인터를 입력으로 받아 파일을 열어준 뒤 정상적으로 열람될경우 해당하는 file descriptor의 번호를 반환한다. 정상적으로 열리지 않을 경우 -1을 반환한다.  
close: file descriptor의 번호를 받아 해당하는 파일을 닫아준다.   
filesize: file descriptor을 받아 해당하는 파일의 크기를 알려준다.  
read: file descriptor을 받아 해당하는 파일에서 size만큼의 byte를 받아 buffer로 넘겨준다.   
write: read system call 과 유사하게 file descriptor을 받아 해당하는 파일에서 buffer 인자의 내용을 file descriptor을 통해서 연 파일에 값을 써준다.  
seek: 해당하는 file descriptor을 입력으로 받아 원하는 위치로 파일 포인터를 이동시켜준다. 단위는 바이트 단위이다.  
tell: file descriptor을 입력으로 받아 현재 파일 포인터를 반환한다.

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

파일 시스템을 단순히 주어진 시스템 콜을 호출하는 방식으로 구현할 경우, 몇가지 동기화 문제가 발생할 여지가 존재한다. 만약 현재 하나의 쓰레드가 읽고있는 파일에 다른 쓰레드가 접근하여 쓰기를 할경우, 즉 공통 변수인 critical section에 대한 다수의 프로세스의 접근이 이루어 질 경우 정확한 결과를 보장할 수 없는 문제가 발생한다. 이 문제를 해결하기 위해 pintos 에서는 lock과 semaphore 인터페이스를 제공한다. Lock이란 세마포어와 쓰레드를 하나의 구조체로 묶은 형태로, 다른 프로세스의 접근을 막을 수 있는 기능들을 제공한다.  
특정 파일에 프로세스가 접근하는 경우, 즉 파일을 read/write 할 경우에는 lock\_acquire을 통한 semaphore down 명령을 통해 다른 프로세스의 접근을 차단하고, 작업이 끝난 이후 lock\_release를 통한 타 프로세스 접근 허용이 가능하다.  
**Why locks not semaphores? 왜 semaphore가 아닌 lock을 쓰는 것인가?**  
이유는 semaphore과는 구분되는 Lock의 차별점에 있다. Pintos manual의 appendix 부분에 언급 되어있듯이, lock은 semaphore와는 다르게 lock을 건 쓰레드만이 lock을 푸는것이 가능하다. 이 부분에 비추어 보면 먼저 파일에 쓰기/읽기 작업을 하고 있던 프로세스만이 자신의 작업을 종료하고 다른 프로세스의 접근을 허용할 수 있는 권한이 존재하는 file system의 synchronization 문제를 해결하기 위해 lock이 적합하다는것을 파악할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2021/10/2 – 2021/10/9 : 시스템 콜 구현  
2021/10/10 – 2021/10/12 : 테스트 및 디버깅  
2021/10/13 – 2021/10/16: 문서화 및 제출

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

남아있는 시스템 콜을 구현하는 것이 주인 프로젝트이므로 대부분의 작업은 syscall.c 함수에서 진행되었다. 별도 유틸리티 멤버 함수들의 구현은 process.c에서 진행하였다.   
지난 프로젝트에 이어서 create, remove, open, close, read, write, open, seek, tell 시스템 콜 함수들을 syscall.c 에서 구현하였다.  
추가적으로 File descriptor을 구현하기 위해서 thread 구조체가 들어있는 thread.h 파일과 thread의 생성을 담당하는 create\_thread 함수가 위치한 thread.c 파일을 수정하였다.

* + 수정하거나 추가해야 하는 자료구조

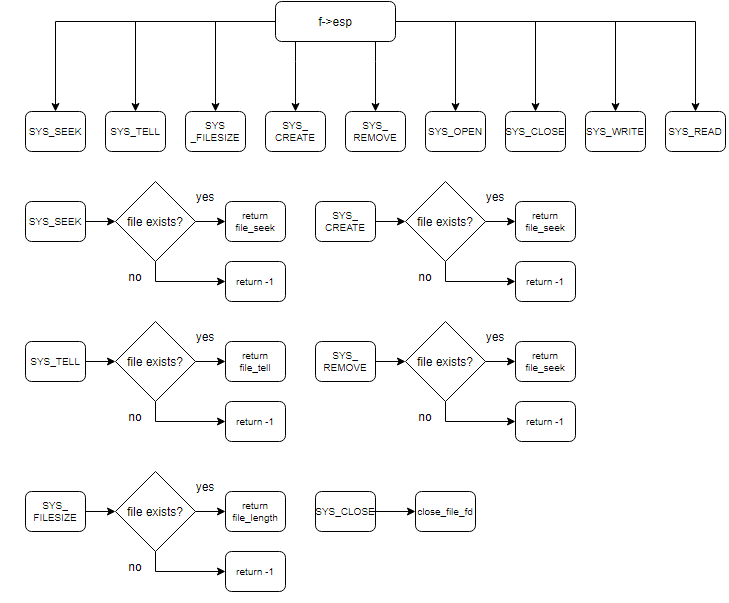
추가해야 하는 자료구조는 단순하다. 각 쓰레드에 File descriptor 자료구조를 추가한다. File descriptor의 최대 개수가 128개이므로 총 128의 크기를 갖는 포인터 배열을 thread.h 파일 내의 thread구조체 내부에 작성하고, 순차적으로 file descriptor을 쌓아나가기 때문에 file descriptor의 최댓값을 하나의 정수로 선언하여 추가해준다. 이 최댓값은 file descriptor가 할당될때마다 갱신된다.

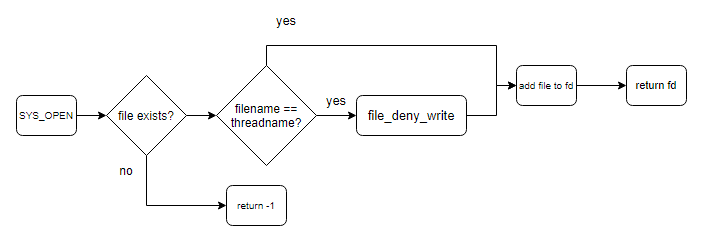
* + 수정하거나 추가해야 하는 함수

이번 프로젝트는 독립적으로 존재하는 file 구조체와 file descriptor가 유기적으로 서로 맞물리도록 구현해야 하는 시스템 콜들이 다수 존재한다. 따라서 파일 구조체를 통해서 file descriptor을 얻을 수 있도록, 그리고 그 반대의 경우도 가능하도록 몇가지 멤버함수들을 process.c에 추가하였다.   
file 구조체를 받아 새로운 file descriptor을 할당하고 file descriptor index를 갱신해주는 add\_file\_to\_fd 함수를 구현하였고, file descriptor을 입력으로 받아 해당하는 file 구조체를 반환해주는 get\_file\_by\_fd함수를 구현하였다.   
또한 file descriptor을 입력받아서 해당하는 file\_descriptor의 할당을 해제시켜주는 close\_file\_fd와 모든 file descriptor들의 할당을 해제시켜주는 close\_all\_fd 를 구현하였다.  
나머지 함수들은 모두 syscall.c 에 구현하였는데, 내용은 모두 프로젝트의 명세로 주어진 시스템 콜 함수들이다.   
목록은 다음과 같다 : open, close, create, remove, read, write, seek, tell, filesize

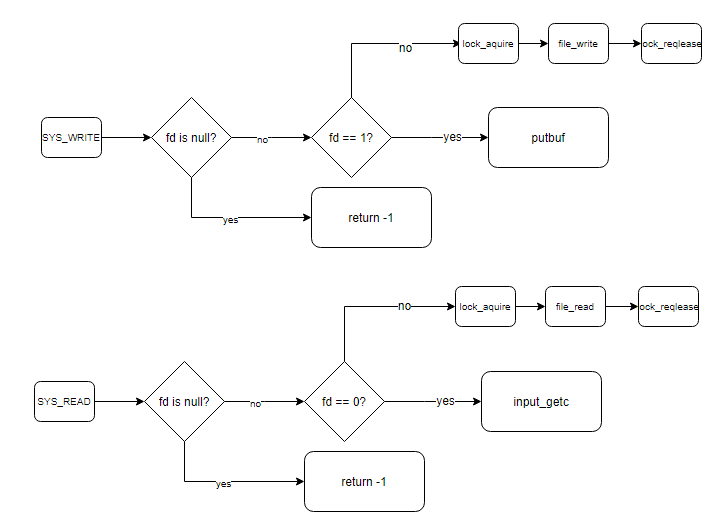
1. **연구 결과**
   1. **Flow Chart**

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

*<그림 1. seek, tell, filesize, create, remove, close syscalls flowchar>*



*<그림2. SYS\_OPEN flowchart>*

*<그림3. SYS\_READ, SYS\_WRITE flowchart>*

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

File descriptors: thread.h 에 선언되어있는 thread 구조체에 다음과 같은 선언을 통해 추가하였다: **struct file \*fd[128], int next\_fd**

System calls: 대부분의 작업은 syscall.c 에서 이루어졌다. 각 시스템 콜 함수 원형을 그대로 선언하여 내부에 작성하였다.   
**시스템 콜 구현 내용:  
SYS\_SEEK:** 파일 포인터를 지정 위치로 옮겨주는 시스템 콜로, 주어진 file descriptor에 해당하는 파일이 존재하는지 확인해준 이후 존재하지 않는다면 -1을 반환하고, 존재한다면 file\_seek함수를 호출시켜준 후 반환한다.  
**SYS\_TELL:** 현재 파일 포인터를 반환하는 시스템콜로, 마찬가지로 파일에 대한 null check 이후 file\_tell함수를 반환시켜준다.  
**SYS\_FILESIZE:** 파일의 크기를 반환하는 시스템콜로, null check 이후 file\_length함수를 반환한다.   
**SYS\_CREATE:** 파일을 생성하는 시스템 콜로, 파일 구조체에 대한 null check 이후 filesys\_create 함수를 실행시키고 반환한다.   
**SYS\_REMOVE:** 파일을 제거하는 시스템 콜로, 파일 구조체에 대한 null check 이후 filesys\_remove 함수를 실행시키고 반환한다.   
**SYS\_OPEN:** 파일을 여는 시스템 콜이다. 입력으로 받은 파일 구조체에 대한 null check를 수행한 후 add\_file\_to\_fd함수를 이용해 file descriptor을 할당해 주었다.  
다만, 여기서 끝낼경우 rox-simple 등 rox관련 test들이 fail하는 문제가 발생하는데, 이유는 pintos 는 현재 실행하고 있는 load가 끝난 실행 파일을 삭제하는 것을 불허하기 때문이다. 따라서 현재 프로세스의 이름이 열고자 하는 파일명과 동일한 경우에는 file\_deny\_write 함수를 이용하여 접근을 차단해준다.   
**SYS\_CLOSE**: 파일을 닫아주는 시스템 콜이다. 이전 유틸리티 함수로 구현한 close\_file\_fd를 호출하여 파일을 닫아준다.

**SYS\_READ:** 파일의 내용을 읽어오는 시스템 콜이다.  
인자로 file descriptor을 입력받기 때문에 해당하는 파일 구조체를 먼저 찾아준 뒤, null check를 진행해준다. 앞서 언급한 파일에 대한 동기화 문제를 해결하기 위해서 lock\_aquire 함수를 호출해준다. 이후에는 기존에 작성한 표준입출력 코드에 대응하도록 분기문을 설정하여 표준 입출력일 경우 input\_getc를, 아닌 경우 file\_read 함수를 호출하고 lock을 release하도록 작성하였다.   
  
**SYS\_WRITE:** 파일에 내용을 작성하는 시스템 콜이다.  
read 시스템 콜과 유사하게 파일 구조체를 먼저 찾고, null check를 수행한다.  
마찬가지로 동기화 문제에 대한 대안으로 lock\_acquire함수를 호출하고 분기문을 작성하여 표준 출력일 경우 putbuf함수를, 아닐 경우 file\_write함수를 호출해주었다.

* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명

**add\_file\_to\_fd:** 파일 구조체를 입력으로 받아 nullcheck 이후 해당하는 file descriptor table에 추가해준다. 이후 file descriptor next\_fd 인덱스를 증가시켜준다. **get\_file\_by\_fd:** file descriptor을 입력으로 받아 유효한지 확인한다.  
file descriptor는 절대 next\_fd(다음 file descriptor을 할당할 index)보다 낮기 때문에 이에 대한 확인 및 음수 여부를 확인하여 유효성 확인을 진행한다.  
이후 해당하는 파일 구조체를 반환한다. **close\_file\_fd:** 역시 유사하게 file descriptor에 대한 유효성 체크 이후 해당하는 file descriptor의 할당을 해제한다.  **close\_all\_fd:** 프로세스가 종료될 경우 호출되는 함수로, 열려있는 모든 file descriptor들의 할당을 해제시켜준다.

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

가장 해결하기 까다로웠던 문제는 write 함수를 구현하는 도중 일어난 일로, bad-read 및 bad-write test를 통과하기 위해서 작성한 validity check 분기문 중 파일의 포인터가 null일경우 return -1 의 코드를 잘못된 위치에 작성하여 프로세스 자체의 stdout 이 모두 실행되지 않아 기존 테스트들이 모두 fail이 나오는 현상이었다.   
다행히 git 에 틈틈히 기능 단위로 구현을 백업해둔 repository를 만들어 두었기 때문에 문제가 일어나기 전 commit으로 roll back 하여 다시 구현하였다.  
두번째는 bad series 들이었다. robustness test들 중 커널 코드에 대한 접근이 이루어지는 경우가 있었는데, 항상 page fault 를 동반한 kernel panic 을 출력하였기 때문에 디버깅하기 쉽지 않았다. exception.c 의 page fault 출력 조건을 수정하는것으로 해결하였다.

* 1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부  
  텍스트이(가) 표시된 사진

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명