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

담당 교수 / 분반 : 박성용 교수님

이름 / 학번 : 20181662 이건영

개발 기간 : 2022.10.06~2022.10.09

1. **개발 목표**

file system과 관련된 system call(create, remove, open, close, filesize, read, write, seek, tell)을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

지난 Project 1에서는 STDIN / STDOUT에 대한 입출력만 구현하였다. 그러나 pintos에서는 파일에 대한 입출력도 동작하여야 하므로 프로세스에서 파일을 관리하기 위한 file descriptor(fd)와 file descriptor table을 구현하여 file system과 관련된 system call이 동작할 수 있도록 한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

create : 파일을 생성하는 system call이다.

remove : 파일을 제거하는 system call이다.

open : 주어진 파일을 열어 해당 파일의 file descriptor를 얻는 system call이다.

filesize : fd에 해당하는 파일의 크기를 반환하는 system call이다.

read : 해당 fd에 해당하는 파일에서 데이터를 읽어오는 system call이다.

write : 해당 fd에 해당하는 파일에 데이터를 출력하는 system call이다.

seek : 해당 fd의 position을 바꾸는 system call이다.

tell : 해당 fd의 현재 position을 반환하는 system call이다.

close : 해당 fd의 파일을 닫는 system call이다.

3. Synchronization in Filesystem

Pintos에서는 여러 프로세스들이 동시에 같은 파일에 접근할 수 있다. 만약 하나의 프로세스가 critical section에 들어갔을 때 다른 프로세스가 파일에 접근하게 된다면 서로 다른 내용의 데이터에 접근하는 Synchronization 문제가 발생할 수 있다. 따라서 Semaphore 혹은 Lock을 사용하여 이를 방지해야 데이터에 안전하게 접근할 수 있다.

* 1. **개발 내용**

1. File Descriptor

File Descriptor table은 각 프로세스 별로 존재하므로 threads/thread.h에 선언된 struct thread에 struct file\*의 배열인 fd\_table을 추가하였다. Pintos 메뉴얼을 확인한 결과 한 프로세스의 fd table의 크기를 특정해야 한다면 128로 권장하고 있기에 128 크기의 배열로 선언하였다.

2. System Calls

create : 파일명과 초기 size를 인자로 받아 root directory에 파일을 생성한다. 성공 여부(true / false)를 반환한다.

remove : 파일명을 인자로 받아 해당하는 파일을 제거한다. 성공 여부를 반환한다.

open : 파일명을 인자로 받아 해당 파일을 열어 그 file pointer를 해당 프로세스의 fd table에 저장한다. 해당 포인터가 저장된 fd table의 index를 fd로 반환한다.

filesize : fd를 인자로 받아 해당하는 파일의 크기를 반환한다.

read : fd와 buffer, size를 인자로 받는다. fd에 해당하는 파일로부터 size byte만큼 데이터를 buffer로 읽어온다. 읽어온 size를 반환한다.

write : fd와 buffer, size를 인자로 받는다. buffer에 담긴 내용을 fd에 해당하는 파일에 size만큼 출력한다. 출력한 byte 수를 반환한다.

seek : fd와 position을 인자로 받아 fd에 해당하는 파일의 위치를 position으로 이동시킨다.

tell : fd를 인자로 받아 해당하는 파일의 현재 position을 반환한다.

close : fd를 인자로 받아 해당 파일을 닫고 fd table에서 삭제한다.

3. Synchronization in Filesystem

open과 read, write system call을 처리할 때 발생하는 synchronization 문제를 방지하기 위해 Lock을 사용할 수 있다. 우선 어떤 프로세스에서 Critical section에 들어가기 전에 lock\_acquire 함수를 통해 다른 프로세스가 해당 영역에 접근하는 것을 막은 후, Critical section에서 빠져나가기 전에 lock\_release 함수를 통해 lock을 해제하여 다른 프로세스에서도 접근이 가능하도록 만든다.

Semaphore를 통해서도 synchronization 문제를 방지할 수 있다. struct semaphore의 value는 sema\_down과 sema\_up 함수를 통해서 증감이 가능하다. sema\_down을 수행하면 value가 0이 아니게 될 때까지 프로세스 수행을 멈추고 대기한다. sema\_up을 통해 value를 증가시키면 정지되었던 프로세스가 다시 작동하는 방식이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

22.10.06~22.10.07 : System call 구현

22.10.07~22.10.09 : Synchronization 문제 해결

* 1. **개발 방법**

1. File Descriptor

File descriptor table을 각 thread에 추가하기 위해 struct file\* fd\_table 배열을 struct thread에 추가하였다. 배열의 크기는 위에서 언급한대로 128로 설정하였다. 또 thread를 초기화하는 init\_thread 함수에서 해당 table의 값들을 NULL로 초기화하기 위해 배열을 초기화하는 부분을 추가하였다.

2. System calls

각 system call에 해당하는 기능을 수행하기 위해 filesys/file.c와 filesys/filesys.c의 함수들을 실행하도록 하였다. filesys\_create, filesys\_remove, filesys\_open, file\_length, file\_read, file\_write, file\_seek, file\_tell, file\_close 함수가 사용되었다.

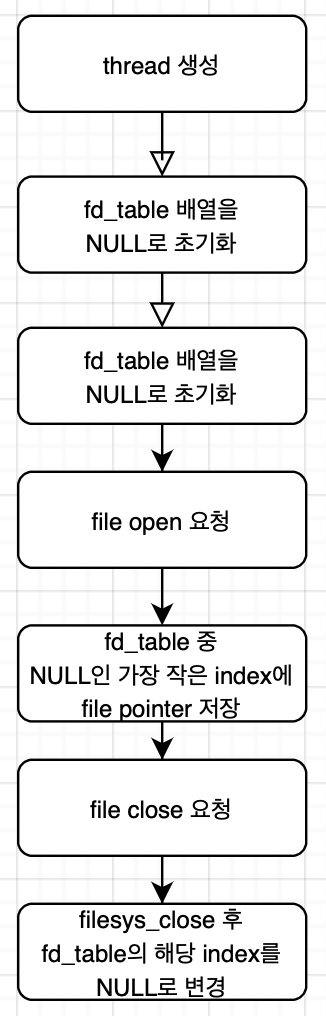
또한 file descriptor table의 접근에 용이하게 함수를 추가하였다. 우선 file pointer를 인자로 받아 해당 thread의 fd table의 빈 자리, 즉 NULL인 index를 찾아 file pointer를 할당하고 해당 index를 return하는 process\_add\_file 함수, fd를 인자로 받아 해당하는 file pointer를 반환하는 process\_get\_file 함수, fd를 인자로 받아 table에서 해당 파일을 닫고 제거하는 process\_close\_file 함수를 구현하였다.

3. Synchronization in Filesystem

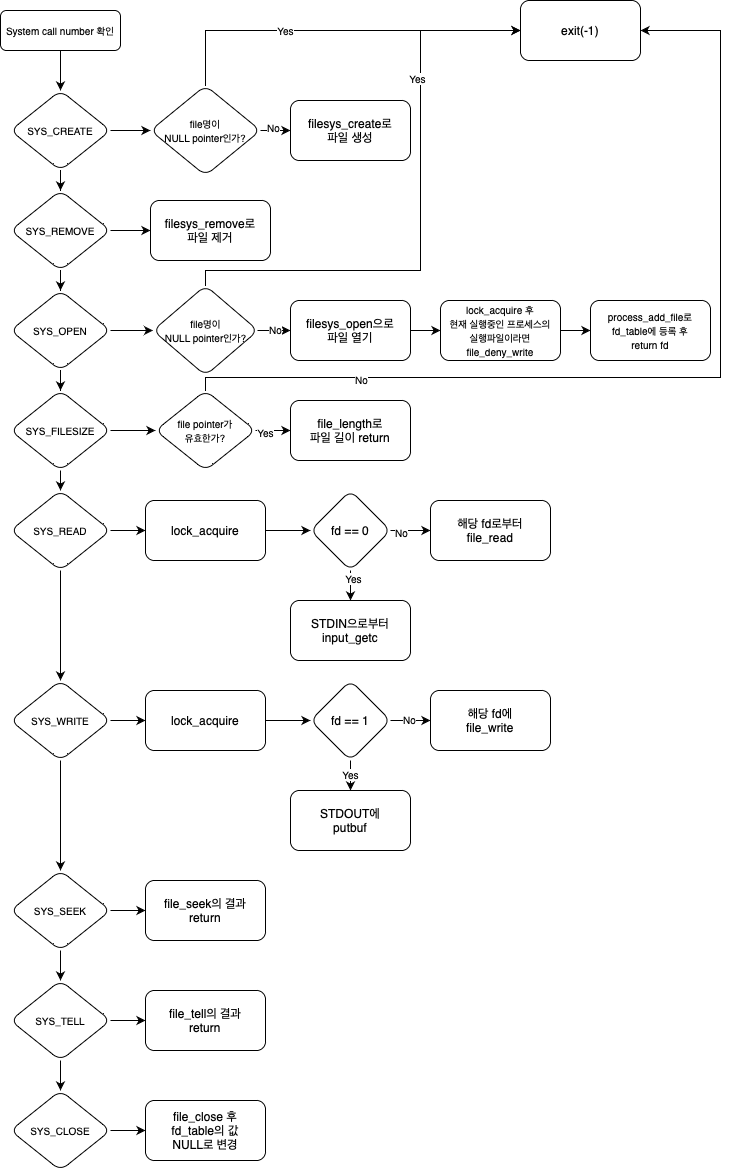
syscall.c의 전역변수로 struct lock 타입의 file\_lock 변수를 추가하여 open, read, write 시 lock을 설정하도록 하였다. 또 자식 프로세스의 실행 중 자식 프로세스의 load 과정이 끝나기 전 부모 프로세스가 종료되는 문제를 막기 위해 thread에 추가적인 sem\_mem semaphore를 추가하였다. 그리고 load의 결과를 저장하던 지역변수 success 대신 thread별로 mem\_success 변수를 두어 load 결과를 저장할 수 있도록 수정하였다.

1. **연구 결과**
   1. **Flow Chart**

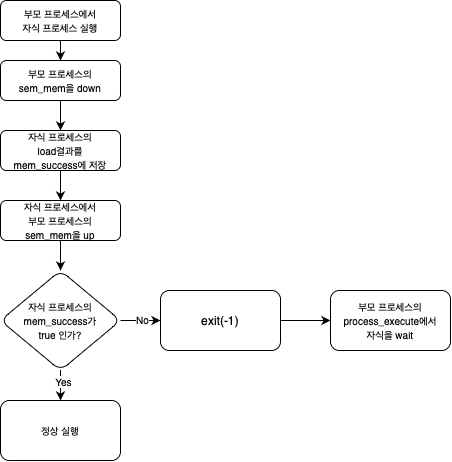
1. File Descriptor



2. System calls



3. Synchronization in Filesystem



* 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

**텍스트이(가) 표시된 사진

자동 생성된 설명**

우선 threads/thread.h 의 struct thread 구조체에 위와 같은 내용들을 추가하였다. fd\_table은 파일 포인터를 담는 배열로 사용하였다. mem\_flag와 mem\_success, sem\_mem은 synchronization 문제를 해결하기 위해서 선언되었다. parent\_thread는 부모 thread의 포인터를 저장하는 변수이다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

다음으로는 threads/thread.c의 init\_thread 함수 부분에 위의 thread 구조체 변수들을 초기화하는 부분을 추가하였다. sema\_init과 for 반복문, 부모 thread에 현재 thread를 담는 것과 mem\_success 변수에 1을 저장하는 부분이 이에 해당한다.

****

userprog/syscall.c의 전역 변수로 struct lock 변수인 file\_lock을 선언하였다.



create와 remove system call에 대해서는 위와 같이 filesys/filesys.c의 함수 filesys\_create와 filesys\_remove를 사용하여 수행하도록 구현하였다. 각각 파일을 생성 / 제거한 후 성공 여부를 반환하는 함수이다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

open system call은 마찬가지로 filesys/filesys.c의 filesys\_open 함수로 연결되도록 구성하였다. filesys\_open 함수는 파일명을 인자로 받아 파일을 열고 파일 포인터를 반환하는 함수이다. 파일명이 유효하지 않은 경우와 파일이 제대로 열리지 않았을 경우를 각각 예외처리(exit(-1)) 한 후 lock\_acquire 함수로 lock을 진행하였다. 이 후 실행 중인 프로세스의 실행파일은 수정되면 안 된다는 점을 보장하기 위해 open한 파일명과 현재 프로세스의 이름(struct thread의 name)을 비교하였다. 프로세스를 실행할 때, thread의 이름을 process\_execute에서 파일명으로 주었기 때문에 위와 같이 구현하였다. 만약 둘이 동일하다면 file\_deny\_write 함수를 사용해서 실행파일에 쓰기를 불가능하게 하였다. 이후 직접 구현한 process\_add\_file 함수를 사용하여 thread의 file descriptor table에 새로운 파일 포인터를 추가하고 fd table의 index를 new\_fd에 전달받아 반환한다. return 전에는 lock\_release 함수를 통해 다른 프로세스에 걸린 lock을 해제한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

filesize system call은 filesys/file.c의 함수 file\_length를 이용하였다. 해당 라이브러리의 함수들은 인자로 파일 포인터를 사용하기 때문에 fd table의 파일 포인터에 쉽게 접근할 수 있도록 process\_get\_file이라는 함수를 추가하였다. 이는 fd를 인자로 받아 해당하는 파일 포인터를 반환하는 함수이다. 이를 통해 file\_length를 실행한 후 return 값인 파일의 길이를 반환하였다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

read system call은 filesys/file.c의 file\_read 함수를 사용하였다. 우선 project 1에서 구현했던 STDIN으로부터의 read는 유지하면서, STDIN이 아닌 fd로부터의 read는 process\_get\_file 함수로 파일 포인터를 구해 file\_read 함수의 인자로 넘겨주었다. 이 과정에서도 함수의 시작과 끝에 lock\_acquire와 lock\_release를 이용해 lock을 걸어주었다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

write system call 역시 기존의 STDOUT으로의 출력 기능은 유지하면서 다른 fd로의 출력에 대해서는 filesys/file.c의 file\_write 함수로 연결되게 구현하였다. 또한 함수의 시작과 끝에 lock\_acquire와 lock\_release 함수도 추가하였다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

seek, tell system call은 filesys/file.c의 file\_seek, file\_tell 함수에 각각 연결되도록 구현하였다.

close system call은 filesys/file.c의 close 함수를 수행한 후, 현재 thread의 fd table에서 해당하는 fd의 파일 포인터 값을 NULL로 변경하여 해당 인덱스를 비워주도록 구현하였다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

위는 userprog/process.c에 추가적으로 구현한 함수들이다.

먼저 process\_add\_file 함수는 파일 포인터를 인자로 받는다. 현재 thread의 fd table에서 비어 있는 공간 중 가장 인덱스가 작은 곳에 파일 포인터를 담고 해당 인덱스를 반환하는 함수이다.

process\_get\_file 함수는 위에서 언급했듯 fd를 인자로 받아 해당하는 파일 포인터를 반환하는 함수이다.

process\_close\_file 함수는 close system call과 같은 기능을 수행한다. process\_exit 부분에서 사용하기 위해 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위와 같이 process\_exit의 마지막 부분에 STDIN / STDOUT / STDERR를 제외한 fd에 대한 파일을 닫고 fd table의 값들을 초기화하는 부분을 추가하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

process\_execute에서는 thread\_create 함수를 통해 thread를 생성하고 start\_process 함수를 실행하게 한다. 이 때 pintos test case의 syn-read / syn-write 부분에서 여러 프로세스를 실행하는 부분에서 에러를 겪었다. 2번째 프로세스 부터는 userprog/process.c의 load 함수가 실패하는 오류였는데, 상황을 파악해본 결과 부모 프로세스가 자식 프로세스의 실행 파일이 메모리에 load되는 것을 다 기다려주지 못한 것으로 생각되었다. 따라서 semaphore를 이용하여 부모 프로세스가 자식 프로세스의 load 과정을 기다려 주도록 수정하였다.

thread\_create 함수 이후 부모 프로세스의 semaphore인 sem\_mem을 sema\_down 해준다. 이후 아래의 start\_process 함수에서 sema\_up을 시켜주고, 마지막으로 자식 프로세스 리스트를 순회하며 load에 실패한 자식 프로세스들을 wait해서 불필요한 정보들을 삭제하도록 하였다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

start\_process 에서는 기존의 load의 return 값을 지역 변수 success에 저장했으나, 각 thread의 mem\_success 변수에 저장하도록 변경하였다. 이를 통해 부모 프로세스에서 위의 process\_execute 함수 마지막 부분에 load에 실패한 자식 프로세스를 완전히 제거할 수 있도록 도와주었다.

load 함수를 호출한 이후에는 현재 프로세스의 부모 프로세스의 semaphore인 sem\_mem을 sema\_up 해서 부모 프로세스가 재개될 수 있도록 하였다.

이후엔 자신의 mem\_success를 확인하여 정상적으로 load되지 않았다면 exit(-1) 하도록 하였다.

* 1. **시험 및 평가 내용**

텍스트이(가) 표시된 사진

자동 생성된 설명

성공적으로 모든 test를 pass하였다.