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

담당 교수 / 분반 : 박성용 / 1 분반

이름 / 학번 : 이우진 / 20181669

개발 기간 : 2023/10/09 – 2023/10/29

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

본 프로젝트에서는 pintos 상에서의 파일 시스템에 관련된 system call을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

2. (이번 프로젝트에서 구현해야 하는) System Calls

3. Synchronization in Filesystem

파일 시스템에 관련된 system call을 구현하기 위해서, 먼저 각 thread별로 열여 놓은 파일에 대한 정보를 저장하기 위한 file descriptor table을 구현한다. 각 process는 file descriptor table에서의 위치를 의미하는 file descriptor을 사용하여 파일 시스템에 관한 시스템 콜을 쉽게 호출할 수 있고, 실제로 파일에 관한 system call을 구현하는데 있어 file descriptor table에 저장해 놓은 정보를 이용할 수 있다.

구현해야 하는 system call은 create, remove, open, close, filesize, read, write, seek, tell 이 있다. 상기한 system call을 구현함으로써 process에서 파일 입출력 및 관리를 OS에 요청하여 수행할 수 있다.

위와 같은 system call을 구현할 때, 여러 process에서 동시에 하나의 파일에 대한 입출력을 요청할 수 있다. 이 때 파일에 관한 동기화 문제가 발생할 수 있는데, 예를 들어 하나의 process가 특정 파일을 읽는 도중에 다른 process가 같은 파일을 수정한다면, 예상하지 못한 결과가 발생할 수 있다. 따라서, 같은 파일에 대해 문제가 발생할 수 있는 부분(critical section)은 동시에 하나의 process만 접근 가능하도록 구현해야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

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)

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

File descriptor table을 구현하는데 array와 linked list 두 가지 자료구조를 선택하여 사용할 수 있다.   
Array을 이용하는 경우, file descriptor을 이용해 바로 file에 대한 접근이 가능하므로 기존에 open한 file에 대한 접근을 쉽고 빠르게 수행할 수 있다는 장점이 있다. 하지만, 동시에 열 수 있는 file의 개수에 제한이 있고 새로운 file을 open 할 경우 비어 있는 file descriptor을 찾는데 시간이 필요하다는 단점이 있다.   
반대로 linked list을 이용하는 경우, 새로운 파일을 추가하는데 필요한 시간이 적고 동시에 열 수 있는 file의 개수에 제한이 없다는 장점이 있으나 file에 대한 접근에 시간이 필요하다는 단점이 있다.   
본 프로젝트의 경우 파일을 동시에 최대 128개 까지만 사용한다는 제한조건이 존재하므로, array을 사용해 구현한다.

본 프로젝트에서 구현할 system call은 다음과 같은 기능을 수행한다.

* + 1. bool create (*const char* \*file, *unsigned* initial\_size)  
        *initial\_size* 만큼의 크기를 초기값으로 가지는 *file* 이라는 이름의 file을 새롭게 생성한다. 성공할 경우 true, 실패할 경우 false을 반환한다.
    2. bool remove (*const char* \*file)  
        *file* 이라는 이름을 가지는 file을 삭제한다. 성공할 경우 true, 실패할 경우 false을 반환한다.
    3. int open (*const char* \*file)  
        *file* 이라는 이름을 가지는 file을 새롭게 open하여 file descriptor table에 추가한다. 성공할 경우 새로운 file descriptor을, 실패할 경우 -1을 반환한다.
    4. void close (*int* fd)  
        file descriptor *fd* 에 해당하는 file을 file descriptor table에서 제거한다.
    5. int filesize (*int* fd)  
        file descriptor *fd* 에 해당하는 file의 크기를 반환한다.
    6. int read (*int* fd, *void* \*buffer, *unsigned* size)  
        file descriptor *fd*에 해당하는 file에서 최대 *size*만큼 입력 받아 *buffer*에 저장한다. 성공할 경우 실제로 읽어 들인 크기를 반환하고, 실패할 경우 -1을 반환한다.
    7. int write (*int* fd, *const void* \*buffer, *unsigned* size)  
        file descriptor *fd*에 해당하는 file에 *buffer*에 저장된 값을 최대 *size*만큼 출력한다. 성공할 경우 실제로 출력한 크기를 반환하고, 실패할 경우 -1을 반환한다.
    8. void seek (*int* fd, *unsigned* position)  
        file descriptor *fd*에 해당하는 file에서 다음에 read/write 할 위치를 *position* 으로 변경한다.
    9. unsigned tell (*int* fd)  
        file descriptor *fd*에 해당하는 file에서 다음에 read/write 할 위치를 반환한다.

File system에 대한 synchronization을 위해, lock과 semaphore 두 가지 방법을 사용하여 synchronization을 구현할 수 있다. Lock을 이용하면 특정 process가 file system에 대한 system call의 수행을 시작할 때 lock을 acquire하고, 끝날 때 release하여 파일 시스템에 대한 system call은 동시에 하나만 수행할 수 있도록 할 수 있다. 이를 remove, read, write와 같은 critical section에 적용하여 synchronization을 구현할 수 있다.  
semaphore을 사용하면 다른 구현이 가능하다. read의 경우 몇 개의 프로세스가 동시에 수행하더라도 문제가 없으므로 read을 시작할 때 semaphore을 up 하고, 종료할 때 semaphore을 down하도록 구현하고 나서 lock은 semaphore가 0일 때만 acquire 가능하도록 구현한다. Write을 시작할 때 lock을 acquire하도록 구현하면 read중인 process가 없을 때만 write을 할 수 있도록 구현할 수 있다. 하지만, 이는 write을 요청 하고 나서 계속해서 read요청이 들어오는 경우 write에 대한 처리가 너무 늦어질 수 있다는 단점이 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2023/10/09 – 2023/10/15 : file descriptor 및 file system call을 구현한다.

2023/10/27 – 2023/10/28 : file descriptor에 대한 synchronization issue을 해결한다.

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

File descriptor을 구현하기 위해 각각의 thread에 file descriptor table을 추가하여야 한다. 그러므로, threads/thread.h 파일의 *struct* *thread* 에 *struct \*file* 타입의 array *file\_descriptor*을 추가하고, threads/thread.c 파일의 *thread\_create( )* 함수에서 *file\_descriptor* 내부의 값을 모두 NULL로 초기화 한다.

File system과 관련된 system call의 경우 모두 userprog/syscall.c 파일에서 구현한다. 기존의 *syscall\_handler( )*에 새롭게 추가할 system call에 대한 조건문을 추가하고, 각각의 system call을 수행하는 함수를 호출하도록 구현한다. 새롭게 추가될 함수는 *sys\_create( ), sys\_remove( ), sys\_open( ), sys\_seek( ), sys\_close( )* 이며, *sys\_write( ), sys\_read( )* 함수는 기존에 stdin, stdout에 대해서만 구현 했으므로 파일 입출력에 대한 구현을 추가한다.

위 system call을 모두 구현하고 나서, userprog/syscall.c 파일에 *struct lock* 변수를 추가하고, *syscall\_init( )*에서 초기화 해준다. 각각의 system call중 *sys\_remove( ), sys\_write( ), sys\_remove( )* 에 대해, 시작 부분에 lock을 acquire하고, 끝나는 부분에 release하는 코드를 추가한다.

1. **연구 결과**
   1. **Flow Chart**

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

**텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명**

**Filedescriptor에 대한 초기화 flow chart**

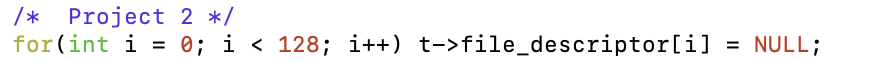
**텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

자동 생성된 설명**

**System call 구현 및 synchronization에 대한 flow chart**

* 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

File descriptor을 구현하기 위해, threads/thread.h 파일의 *struct thread* 에 *struct file\* file\_descriptor[128]* 을 추가하였다. 해당 배열은 threads/thread.c에 위치한 새로운 thread의 생성이 이루어 지는 *thread\_create( )* 함수에서 0에서 128번 까지 모두 NULL로 초기화 해주었다.

**  
file\_descriptor의 초기화 코드**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
file\_descriptor 배열을 추가한 thread struct**

system call의 구현을 위해, userprog/syscall.c 파일에서 기존의 *syscall\_handler( )* 함수에서 lib/syscall-nr.h 파일의 system call의 enum 값을 참조하여 *syscall\_num*의 값이 SYS\_CREATE, SYS\_REMOVE, SYS\_OPEN, SYS\_CLOSE, SYS\_FILESIZE, SYS\_SEEK 일 경우를 추가하였고, lib/user/syscall.h에 위치한 system call의 함수 원형을 참조하여 각각에 대한 함수를 userprog/syscall.h와 userprog/syscall.c에 추가하고, *syscall\_handler( )* 에서 적절히 stack에서 적절히 변수를 가져와 인자로 넘길 수 있도록 구현하였다.

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명  
userprog/syscall.h에 추가한 함수 원형**

위에 선언한 함수 원형에 대해 각각 system call을 구현하였다. 이 때, filesys/file.h에 구현된 함수들을 이용하여 각각 구현하였다. Filesys/file.h에 구현된 함수에 필요한 인자와 userprog/syscall.c 에서 선언한 함수들의 인자가 대체로 같으므로, 인자를 그대로 넘겨주어 구현할 수 있었다. 이 때, 인자로 *fd*를 입력받는 함수들의 경우 현재 thread의 *file\_descriptor* 배열의 *fd* 위치를 접근하여 file pointer을 구하여 인자로 전달해야 했는데 이 때 file pointer가 존재하지 않는 경우 error로 출력하였다.

System call을 구현하면서, 현재 실행중인 파일에 대한 write system call은 deny되도록 구현하여야 했다. 이를 위해, file open시에 open하고자 하는 file의 이름과 현재 thread에서 실행중인 파일의 이름을 비교하고, 같다면 filesys/file.h 에 위치한 *file\_deny\_write( )* 함수를 호출하여 앞으로 해당 파일에 대한 write system call은 deny되도록 구현하였다.

마지막으로 file system call에 대한 synchronization issue을 해결하기 위해 userprog/syscall.c에 *struct lock filesys\_lock* 변수를 추가하였다. 이후, write, read, remove system call의 경우 시작하기 전에 *filesys\_lock*을 acquire한 이후에 실행되도록 하였고, 종료될 때 *filesys\_lock*을 release 하도록 구현하였다.

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

텍스트, 스크린샷, 스티치, 패턴이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 흑백, 폰트이(가) 표시된 사진

자동 생성된 설명