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

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070] 운영체제

담당 교수 : 박성용 교수님

조 / 조원 : 11조, 20151547, 20151561

개발 기간 : 2019/11/4 - 2019/11/17

**프로젝트 제목 : Pintos Project 1 User Program (2)**

**제출일 : 2019년 11월 17일**

**참여 조원 : 박재상, 신용하**

1. **개발 목표**

* pintos는 **File system**에 대한 **access**를 위해서 **syscall**에 접근할 수 있어야한다. 그렇지 않으면 제대로된 Filesystem관련 함수( ex) read, write **)**를 진행할 수 없다. 또한 각 file에 대해 synchronization에 대한 구현도 제대로 되어있지 않은 상황이다.
* 이번 프로젝트에서는 크게 **File System** 의 제대로된 기능을 위해 적절한 **syscall**을 사용할 수 있는 **API**를 구현하고, **File의 Consistency를** 위해 **Synchronization**을 통해서 **File**에 접근하는 각 **process**의 순서를 실행 규칙을 정하고, 각각의 **process**의 **Critical Section**을 보호하는 기능을 구현할 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **1. API for File System**

기존 project1에서 syscall\_handler는 filesystem에 대한 syscall을 포함하지 않았다. read,write의 경우에도 stdin,stdout인 경우에 해당하는 구현만 진행하였다. 이 syscall\_handler에 적절한 기능을 추가하여 사용자가 올바른 filesystem의 사용을 할 수 있도록 만들어야한다.

* **2. Synchronization**

Pintos에서 제공하는 filesystem에는 몇 가지 한계가 있다. 그 중, 가장 중요한 synchronizaion에 대한 구현이 없다. 하지만 원활한 전체 program의 진행을 위해서는, file에 대한 access를 잘 통제해야 한다. 이를 위해, struct file을 새로 구현하여 syscall\_handler에 의해 call되는 함수에서 접근하려는 file에 대한 정보를 확인 후 task를 진행하도록 만들어야 한다. 또한 자식 process가 load 되기 전에 부모 process가 끝나지 않도록 적절한 synchronization 기법이 필요하다.

* 1. **개발 내용**

**1. API for filesystem**

기존 Project1에서 구현한 sycall\_handler에 filesystem을 사용하는 syscall을 추가하기 위해 struct file을 추가한다. 또한, syscall\_handler의 switch문에서 뒷부분에 더 많은 case가 추가될 것이다. 또한, 기존의 syscall.c에 구현되어 있는 함수의 경우에도, filesystem에 대한 기능이 추가될 것이다.

주요 개념:

1) deny write: 파일의 내용을 동시에 여러 process에서 변경하려 해선 안된다.

2) critical section: 여러 process가 공유하는 영역. 동시에 실행될 경우엔 적절한 synchronization을 해야 한다. ex) filesystem

* **2. Synchronization**

기존 pintos의 filesystem이 synchronization에 대한 기능이 없기 때문에, 우리가 직접 구현해야 한다.

이는 syscall.c에 struct file을 추가하여 file의 정보를 읽어올 수 있도록 할 것이고, 여기 있는 함수들에 synchronization 기능을 추가하는 것으로 구현할 것이다.

또한, struct thread에서 현재 열려 있는 file을 관리할 수 있도록 struct file pointer 멤버를 추가할 예정이다.

주요 개념:

1) mutex lock: critical section을 가진 thread들의 running time이 서로 겹치지 않게 한 번에 한 thread만 접근하도록 한다. synchronization 대상이 하나일 때 사용한다. 사용한 thread에서 직접 해제하기 때문이다.

2) semaphore: critical section을 가진 process들의 running time이 서로 겹치지 않게 한 번에 한 process만 접근하도록 한다. synchronization 대상이 여러 개일 때 사용한다. 다른 process에서 해제하기 때문이다.

3) mutex lock vs semaphore: mutex lock 은 상태가 두 개인 semaphore로 볼 수 있으므로 범위로 보자면 mutex lock < semaphore 이다. semaphore는 count를 사용해 접근하는 process들의 개수에 따라 synchronization을 할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **2019/11/4 - 2019/11/17**
  1. **개발 방법**

**1. API for filesystem**

기존 Project1에서 구현한 userprog/syscall.c의 sycall\_handler()에 filesystem을 사용하는 syscall을 추가하기 위해 struct file을 추가한다. 이 struct에는 file의 inode를 관리하는 struct inode pointer(이 프로젝트에서는 사용하지 않음), 그리고 현재 위치를 의미하는 pos, 그리고 writable을 알려주는 deny\_write가 포함되어 있다.

그리고 filesystem에 대한 syscall을 부르는 경우에 대한 case를 syscall\_handler의 switch문에 추가한다.

또한, 기존의 syscall.c에 구현되어 있는 함수의 경우에도, filesystem에 대한 기능이 추가하기 위해 몇 가지 손을 보았다.

userprog/exception.c 에 새로운 pagefault 조건을 추가했다. (조건 강화)

exit()의 경우, 기존에는 exit\_status를 넘기고 thread\_exit()을 call했지만, 새로 작성한 exit()에서는, 현재 thread가 가지고 있는 모든 file을 close() 후에 앞의 기능을 하도록 수정했다.

파일 포인터가 NULL값으로 넘어오면 exit(-1)을 하도록 하였다.

나머지 filesystem에 관한 함수들은 새로 추가되는 함수들인데, pintos manual에 따라 작성하였다.

* **2. Synchronization**

기존 pintos의 filesystem이 synchronization에 대한 기능이 없기 때문에, 우리가 직접 구현해야 한다.

synchronization기능을 구현하기 전, 우선적으로 추가해줘야 할 변수들이 있다.

우선 struct lock file\_lock 변수인데. 이는 한 process의 thread들이 공유하는 변수이다. 즉, file\_lock 값을 확인하여 현재 thread가 더 이상 진행해도 되는지를 확인하며 synchronization을 구현할 수 있다.

또한, threads/thread.h에 선언되어 있는 struct thread에 struct file\* file\_list[131]를 추가한다. 131개의 배열로 선언을 한 이유는, linked list로 동적으로 관리하기가 어려웠고 앞의 세 개는 사용하지 않고 최대 파일 개수가 128개라는 pintos의 특징을 알고 있기 때문이다.

이 file\_list는, 현재 thread가 가지고 있는 file에 대한 모든 목록을 포함하기 때문에 fd 값을 이용하여 어떤 file을 사용할 지를 결정할 수 있게 된다.

우리가 synchronization 기능을 추가한 함수는 read(), write(), open()이다.

현재 thread가 open 하려고 하는 file의 consistency를 지키기 위해 deny\_write 값을 변경하려고 할 때, 다른 thread가 해당 file에 대한 write를 진행하게 된다면, 원래는 write할 수 없는 file인데, 다른 thread에서 write를 진행해버리는 상황이 발생한다. 이런 상황을 방지하기 위해 open()에서 deny\_write값을 변경하는 과정을 critical section으로 구성하고, lock\_acquire(&file\_lock)과 lock\_release(&file\_lock)사이에 배치함으로써 CS를 보호하고, sync를 맞출 수 있도록 구현하였다.

wirte()함수에서 file에 대한 write를 진행하는 과정과 read()함수에서 file에 대한 read를 진행할 때 모두 CS이기 때문에, open()에서와 같이 file\_lock을 통해 CS를 보호하고, sync를 맞췄다.

자식 process가 load 되기 전에 부모 process가 끝나지 않도록 threads/thread.h에 새로운 semaphore인 load\_lock을 만들었다. 기존의 두 개의 semaphore로는 이 기능을 수행할 수 없었기 때문이다. userprog/process.c의 process\_execute() 에서 load\_lock은 부모가 자식 thread를 만들고 sema\_down을 하고 자식의 load가 완료되면 sema\_up을 하여 부모의 execute전에 자식의 load가 완료될 수 있도록 하였다.

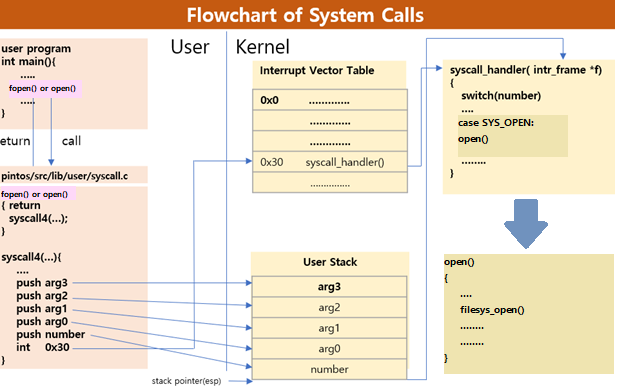
만약 load가 실패했다면, 부모가 자식이 강제종료 되었는지 확인하고 process\_wait()을 해주어 부모가 자식을 완료하여 회수할 수 있도록 하였다.

* 1. **연구원 역할 분담**

**신용하: Synchronization, error handling**

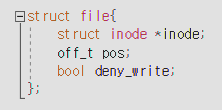
**박재상: Syscall\_handler, Additional Implementation**

1. **연구 결과**
   1. **합성 내용**

****

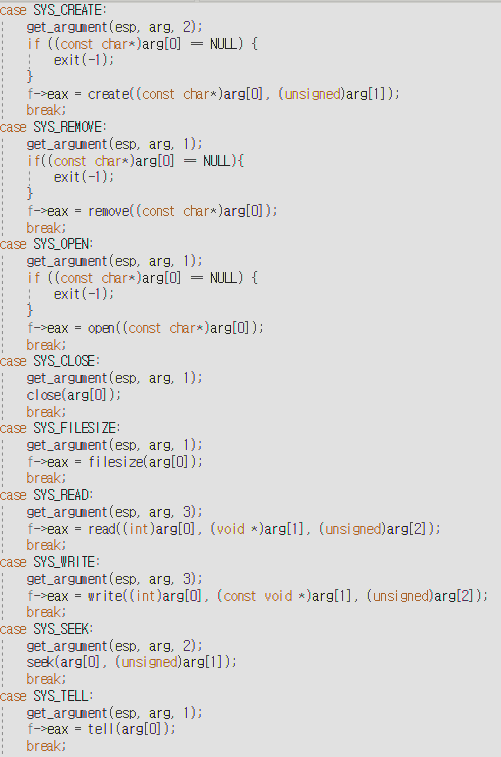
* 1. **제작 내용**
* **B-1 API for file system.**

새롭게 추가된 struct file은 아래와 같다.

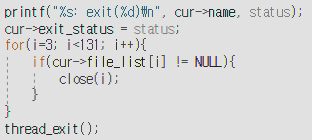
****

syscall\_handler()에서 filesystem에 대한 syscall을 관리하기 위해, switch문에

새로운 case들을 추가하였다.

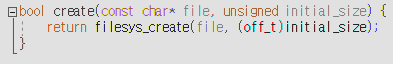


기존의 eixt()함수에, 현재 thread가 open한 모든 file을 close하고나서 종료되도록 아래와 같은 코드를 추가하였다.



다음은 새롭게 추가된 file system syscall함수들이다.

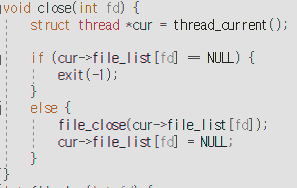
create()



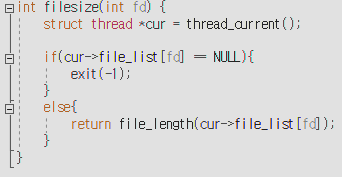
remove()



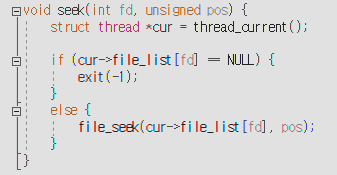
close()



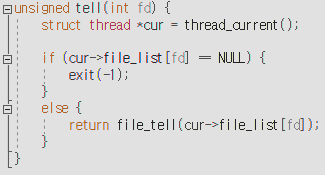
filesize()



seek()



tell()

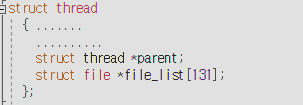


* **B-2 Synchronization**

가장 먼저 synchronization을 구현하기 위해서 file\_lock이라는 변수를 선언했다.



또한, thread.h에 선언된 struct thread에 멤버 변수를 추가하였다.

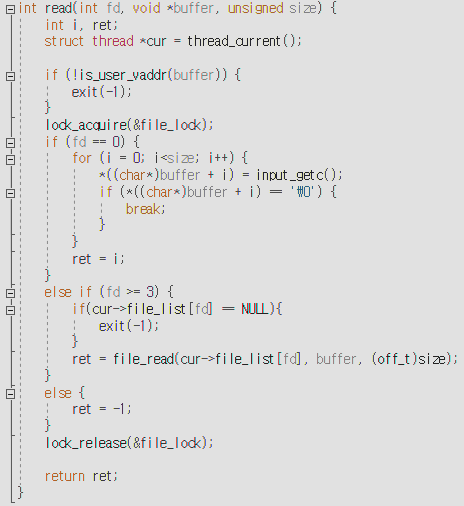


수정된open()은 아래와 같다.



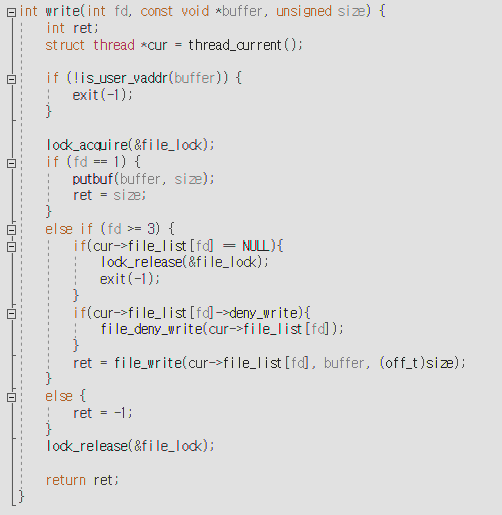
해당 thread가 oepn한 file의 deny\_write값을 변경하는 코드가 추가되어 있다.

read()의 수정버전은 아래와 같다.



critical section에 진입하기 전, lock\_acquire(&file\_lock)을 통해 다른 thread에서 critical section에 진입하지 못하도록 했다. 또한 fd값이 3이상일 때 file\_read()라는 syscall함수를 호출하는 모습을 볼 수 있다. 모든 read과정이 끝난다면, lock\_release를 통해 다른 thread의 CS진입을 허용한다.

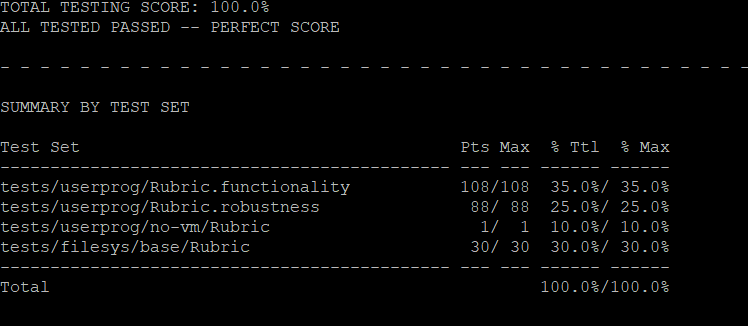
write()의 수정버전은 아래와 같다.



read와 마찬가지로, lock\_acquire와 lock\_release를 통해 CS를 보호하는 것을 볼 수 있다.

그리고, write를 진행하기 전, 해당 file에 write를 진행해도 상관없는지 확인하기 위해, deny\_write의 값을 확인하는 모습을 볼 수 있다.

* 1. **시험 및 평가 내용**

****

**정상적으로 모든 테스트를 통과하였다.**

1. **기타**
   1. **연구 조원 기여도**

* **신용하(50%) 박재상(50%)**
  1. **소감**

**신용하: synchronization을 저번에 이어서 하게 되었다. 가장 어려운 부분은 아무래도 multi-oom과 syn-read, syn-write이었는데, 처음에 syn-read, syn-write가 read, write systemcall에서의 오류로 생각하고 작성하였는데, 나중에 알고 보니 load 와 exec의 synchronization이어서 찾기 어려웠다. 또한 multi-oom은 올바른 값을 내뿜고 완료가 되어도 정상적으로 모든 파일을 닫지 않고 리스트에 남아 있다면 통과가 되지 않았다. 파일 시스템을 동적할당 하지 않고 배열로 만든 것이 테스트 통과를 그나마 쉽게 할 수 있는 발판이 되었다고 생각한다.**

**박재상: 이제 pintos에 대해 어느정도 알고 있는 상황에서 project를 진행하다 보니 이전 project보다는 괜찮았다. 밑바닥부터 다시 채워야 했던 이전 project에 비해, syscall의 추가와, synchronization 기능을 추가하는 것이 이번 project였기 때문에 기존에 비해 많은 시간이 단축되었다. 여러모로 synchronization기능을 구현하느라 고생한 조원에게 감사할 따름이다.**