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

담당 교수 / 분반 :1분반

이름 / 학번 :서윤혁 / 20191597

개발 기간 : 3주

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

프로젝트1에서 구현하지 않은 추가 system call 구현 및 read, write system call들을 stdin과 stdout뿐 아니라 file read, write도 가능하도록 수정.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목을 구현해야 하는 이유, 혹은 구현 시 기대되는 결과를 간략히 서술
  + 1. File Descriptor

System call을 통해 파일을 open, read, write할 때 파일에 접근을 해야하는데, 이 때 file descriptor를 사용하게 된다. 파일들은 위에서부터 순서대로 file descriptor를 채우게 된다. 그리고 핀토스에서 기본적으로 file descriptor가 0은 STDIN 1은 STDOUT으로 세팅되어있다..

* + 1. (이번 프로젝트에서 구현해야 하는) System Calls

Create, remove, open, seek, filesize, tell, close 즉 파일을 handling할 때 필요한 system call들을 구현해야하고 프로젝트1에서 구현했던 read와 write를 STDIN, STDOUT뿐 아니라 일반적인 file read, write까지 되도록 추가 구현해준다.

3. Synchronization in Filesystem

여러 process에서 하나의 파일에 접근 또는 사용을 할 때 문제가 일어날 수 있는데, 이를 방지하기 위해 semaphore나 lock방식을 사용하여 해결한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

Array를 이용하여 구현.  
thread는 파일에 접근할 때 array의 index인 file descriptor를 통해서 접근을하게 될건데, array는 index를 알면 O(1)에 찾을 수 있기 때문에 효율적이고, 구현도 쉬워서 해당 자료구조를 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)

Craete: 파일명과 size를 받아서 파일을 생성해주는 system call

Remove: 파일명을 받아서 해당 파일명을 가진 파일을 삭제하는 system call

Open: file명을 받아서 해당 파일명을 가진 파일을 file descriptor에서 찾아서 open 해주는system call

Seek: fd의 file descriptor를 갖는 파일의 position을 변경하는 system call

Filesize: fd의 file descriptor를 갖는 파일의 size를 반환하는 system call

Tell: fd의 file descriptor를 갖는 파일의 position을 반환하는 system call

Close: fd의 file descriptor를 갖는 열려있는 파일을 close해주는 system call

Read: fd의 file descriptor를 갖는 파일을 size만큼 read하여 buffer에 저장하는 system call  
리턴값은 실제로 읽은 byte 수 or -1

Write: fd의 file descriptor를 갖는 파일을 size만큼 buffer로부터 write 하는 system call  
리턴값은 실제로 쓴 byte 수 or -1

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

File을 read, write, open할 때 lock을 이용한다.  
 하나 이상의 쓰레드가 file에 접근할 때 파일 수정이 이루어지면 안되므로 lock을 이용해서 read나 open이 되고 있는 상황에 write를 할 수 없도록 한다.

한 편 semaphore도 사용하게 되는데 process\_execute 함수와 start\_process함수에 사용한다. Thread를 생성하고나서 start\_process의 load가 일어나기 전에 부모 process가 끝나버리게 되지 않도록 semaphore를 이용해서 자식의 load가 일어나기 전까지 부모 process의 semaphore down을 해주고 load가 끝난 후 up을 해준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10/21일: 추가 system\_call 구현

10/22일: synchronization 문제 해결 (multi-oom) 및 코드 정리

10/28일: 보고서 작성

* 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

Thread.h: file descriptor 배열 추가 및 syncronization을 위한 semaphore 추가

Thread.c: semaphore 0으로 초기화 및 file descriptor null로 초기화

Process.c: process\_execute, start\_process 함수에 세마포어를 이용하여 child process가 load되기 이전에 parent process가 먼저 끝나는 문제를 방지

Syscall.c: 추가 구현 할 system call 함수의 business logic 작성

Filesize(), seek(), create(), remove(), open(), close(), tell() 함수 추가

Read(), write() 함수 수정

Syscall.h: 추가 구현 할 system call의 함수를 정의

1. **연구 결과**
   1. **Flow Chart**

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

**텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명**

* + 1. System call

텍스트, 평행, 직사각형, 도표이(가) 표시된 사진

자동 생성된 설명

* + 1. Synchronization

Using lock

**텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명**

Using semaphore

**라인, 도표, 직사각형, 평행이(가) 표시된 사진

자동 생성된 설명**

* 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

**텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

Thread.h 파일에 file descriptor를 위한 fd배열을 추가해주었고, syncronization을 위한 sync\_sema를 추가해주었다. 또한 parent thread의 접근이 필요한 경우가 있어서 추가해주었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명위에서 추가 정의해준 애들은 thread.c 파일에서 초기화 시켜주는데, fd는 null값으로 모두 초기화해주고, sync\_sema는 0으로 parent는 현재 running중인 thread를 넣어준다.

이렇게 초기 세팅을 해준 후 추가로 구현해준 system call들을 살펴보자.

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

File의 size를 리턴해주는 sys\_filesize system call이다.

Fd를 받아 해당 파일이 file descriptor에 있다면 파일의 크기를 return 해준다.

이때 file.c의 file\_length 함수를 이용하는데 이는 inode의 data length를 리턴해준다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

File의 position을 변경해주는 system call이다.

Fd를 받아 해당 파일이 file descriptor에 있다면 해당 파일의 position을 파라미터 값으로 받은 positon으로 바꾼다.

이때도 file.c의 file\_seek 함수를 이용하는데 이 함수는 file의 pos값을 position으로 업데이트 해주는 함수이다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

File을 생성해주는 system call이다.

이때 filesys.c의 filesys\_create함수를 이용하는데, initial\_size만큼 file이름으로 파일을 생성해주는 함수이고, 파일 생성에 성공하면 1을 리턴한다.

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

File을 제거해주는 system call이다.

Filesys.c의 filesys\_remove함수를 이용하고 파라미터로 받은 file명과 같은 파일을 제거한다. 성공하면 1을 리턴한다.

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

현재 파일의 position을 리턴하는 system call이다.

File.c의 file\_tell함수를 사용하는데, file의 pos를 리턴해주는 함수이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

파일을 닫는 system call이다.

Fd를 받아 file descriptor 테이블의 값을 null로 바꿔주고, file.c의 file\_close함수를 호출한다. File\_close 함수는 닫으려는 file의 inode를 close해주고 file을 free 해준다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

파일을 열어주는 system call이다.

파일명을 파라미터로 받아서 filesys\_open을 통해 파일을 찾아온다. Filesys\_open 함수는 create system call로 만들어둔 파일을 파일명을 이용해서 찾은 파일을 리턴해준다. 우리는 lock을 이용해서 다른 쓰레드가 동시에 파일에 접근하지 못하도록 막는다.

파일이 open되는동안 write되는 것을 방지하기 위해서 file\_deny\_write함수를 이용해서 해당 file의 deny\_write 값을 true로 바꿔준다.  
그리고 나서 file descriptor 테이블의 앞쪽부터 탐색하여 null값을 가진 fd에 넣어준다. 그리고 해당 fd를 리턴해준다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

파일을 읽는 system call이다.

여기서도 마찬가지로 여러 thread에서 동시 접근이 되지 않도록 막기 위해 lock을 사용하였고, file.c의 file\_read 함수를 통해 파일을 읽고 실제로 읽은 byte를 return 해준다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

File을 write할 때 사용하는 system call이다.

여러 thread에서 동시에 접근하지 못하도록 lock을 사용하였고, file.c의 file\_write함수를 이용하여 file에 쓰도록 구현했다. 실제로 쓴 byte만큼 return을 한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Process.c의 process\_execute, start\_process 함수 부분이다.

처음에 이 부분을 구현하지 않고 진행을 했는데 multi-oom테스트 말고는 모두 통과했다. Multi-oom은 메모리 누수 테스트인데 왜 실패하는지 처음에는 잘몰랐고, 해당 테스트 구현 파일을 직접 뜯어 디버깅을 해보고나서 깨닫게 되었다.   
  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명  
  
이 부분에서 chlid를 생성해주는데 wait가 제대로 되지않아 자꾸 실패를 하였고 이는 child thread가 생성되고 load되기 전에 parent가 먼저 죽어서 제대로 wait되지 않기 때문에 발생하는 것이었다.

따라서 semaphore를 이용해서 thread가 create되고나서 load될때까지 parent를 semaphore를 이용하여 lock을 걸어두었다.

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

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명