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

담당 교수 / 분반 : 박성용 / 1분반

이름 / 학번 : 최원빈 / 20181693

개발 기간 : 2022.10.05 ~ 2022.10.26

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트에서의 목표는 파일 시스템과 관련된 system call(ex. create, remove, open, close, filesize, read, write, seek, tell 등)들이 잘 수행되도록 구현하는 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor: 현재 핀토스에서 제공하는 base file system에서는 파일 오픈 시 파일 포인터를 반환하고, read·write를 할 때도 파일 포인터를 parameter로 넘겨 받아 작업을 한다. 즉, 어떤 프로세스(쓰레드)에서 파일 관련 작업을 하려면 파일 오픈 후 포인터를 유지해야 하는데 이를 관리하기 위해 파일 디스크립터를 구현한다. 구현 시 여러 파일 포인터들을 관리하며 효율적으로 사용할 수 있을 것이다.

2. (이번 프로젝트에서 구현해야 하는) System Calls: 기본적으로 위에서 언급한 시스템 콜들을 구현한다. 이때 지난 프로젝트1에서 구현한 read·write도 수정해야 한다. 이런 시스템 콜들을 구현함으로써 우리는 user program에서 파일을 열거나 읽고 쓰는 작업들을 수행할 수 있을 것이다.

3. Synchronization in Filesystem: 만약 synchronization 없이 파일을 열거나 읽고 쓰면 여러 프로세스에서 동시에 특정 critical section에 접근할 수 있다. 이 경우 정보 업데이트 속도가 달라져 기대하는 작업 결과가 안 나올 수 있다. 때문에 critical section에 접근하는 작업들은 한 번에 한 프로세스만이 이용 가능하도록 동기화 작업이 필요하다. 구현 시 안정적인 작업을 수행할 수 있을 것이다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

구현에 이용할 자료구조로 배열을 선택했다. 후보로는 linked list가 있었는데, 이를 이용 시 크기를 자유롭게 조절할 수 있고, 삽입 및 삭제가 쉽다는 장점이 예상되었다. 그러나 핀토스 매뉴얼에서 파일디스크립터 개수에 필요하다면 128개의 제한을 두어도 된다는 조건이 있는 것을 확인하였다. 때문에 굳이 삽입 및 삭제가 필요하며 접근 속도도 느린 linked list보다 그냥 한 번에 128개 크기의 포인터배열을 선언해 사용하는 것이 훨씬 효율적일 것이라 생각해 이를 선택했다.

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: 인자로 받은 file이라는 파일을 initial\_size의 크기로 생성한다.

-remove: 인자로 받은 file이라는 파일을 제거한다.

-open: 인자로 받은 file이라는 파일을 open 한다.

-filesize: 현재 쓰레드의 파일 디스크립터 중 인자로 받은 fd번째와 연결된 파일의 szie를 반환한다.

-read: 현재 쓰레드의 파일 디스크립터 중 인자로 받은 fd번째와 연결된 파일을 size 만큼 읽어 buffer에 넘긴다. 그후 실제로 읽은 byte를 반환한다.

-write 현재 쓰레드의 파일 디스크립터 중 인자로 받은 fd번째와 연결된 파일에 buffer에 있는 내용을 size 만큼 넘긴다. 그후 실제로 쓴 byte를 반환한다.

-seek: 현재 쓰레드의 파일 디스크립터 중 인자로 받은 fd번째와 연결된 파일의 다음 읽거나 쓰일 byte를 position으로 변환한다

-tell: 현재 쓰레드의 파일 디스크립터 중 인자로 받은 fd번째와 연결된 파일의 다음 읽거나 쓰일 위치의 byte를 반환한다

-close: 현재 쓰레드의 파일 디스크립터 중 인자로 받은 fd번째와 연결된 파일을 닫는다. 포인터도 NULL값으로 바꾼다.

-exit: 추가로 지난 프로젝트 1에서 구현한 exit 시스템 콜을 프로세스 종료 시 열려 있는 디스크립터들을 모두 닫도록 수정한다.

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

한 파일에 한 쓰레드가 read 혹은 write 작업 수행을 할 때, 다른 쓰레드가 접근(open)을 하면 문제가 생길 수 있으므로 read/write/open 시스템 콜을 수행할 때 자체적으로 락을 잡고 종료 시 락을 회수하도록 구현한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2022.10.05 ~ 2022.10.19: 파일 디스크립터 생성 및 시스템 콜 구현

2022.10.05 ~ 2022.10.26: open read write 시스템 콜 수행 시 synchronization 작업 구현

* 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

1. File Descriptor: thread.h에 있는 thread 구조체에 struct file\* fd[128]을 추가 및 thread.c에서 init\_thread 할 때 초기화(fd는 다 NULL로). Syscall.c의 exit 함수에서 아직 close 호출을 안 한 파일 디스크립터 존재 시 호출 후 종료.

2. (이번 프로젝트에서 구현해야 하는) System Calls: syscall.c에 create, remove, open, close, filesize, seek, tell 함수 추가, read, write, exit 수정

3. Synchronization in Filesystem: Syscall.c에서 전역변수로 struct lock read\_and\_write\_lock 생성 및 syscall\_init 할 때 초기화. Open read write 함수에서 시작 시 lock을 잡고, 종료 전에 해제하도록 수정.

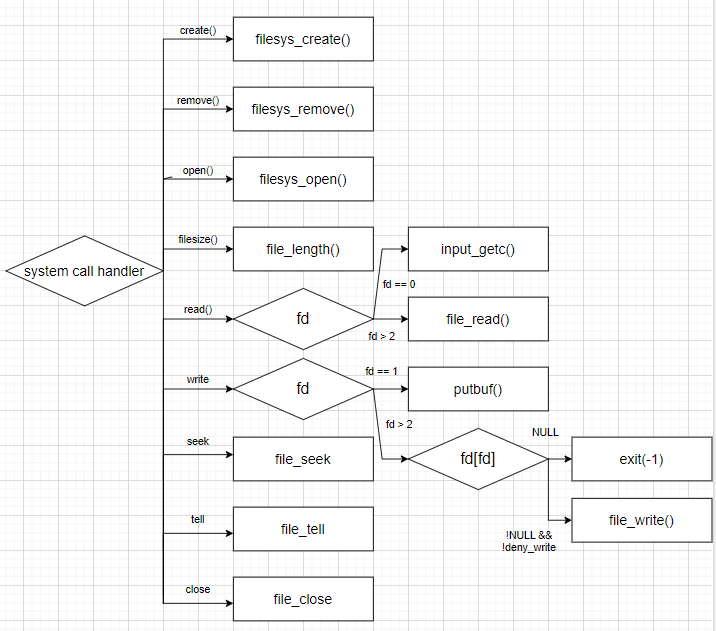
1. **연구 결과**
   1. **Flow Chart**

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

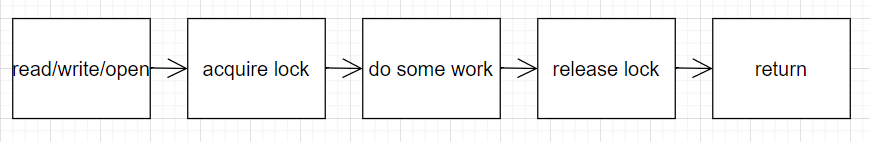
텍스트, 지도, 실내이(가) 표시된 사진

자동 생성된 설명

* 2. (이번 프로젝트에서 구현해야 하는) 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
* 1. File Descriptor: thread.h의 thread 구조체에 struct file\* fd[128]을 추가한다. thread.c의 init\_thread에 모든 fd를 반복문을 돌면서 NULL로 초기화 해주는 코드를 작성한다.
* 2. (이번 프로젝트에서 구현해야 하는) System Calls: syscall.c에 다음 함수들을 추가한다. 기본적으로 모든 함수들은 포인터를 받은 경우 NULL값이면 exit(-1)을 호출한다. 아니면 프로젝트 1에서 만들었던 check\_user\_addr 함수를 호출해 valid한 주소인지 확인한다. fd를 받은 경우 thread\_current()->fd[fd]가 NULL인지를 먼저 확인하고 맞다면 역시 exit(-1)을 호출한다.

bool create(const char \*file, unsigned initial\_size): 받은 인자로 filesys\_create() 함수를 호출한다. filesys\_create() 함수는 받은 인자로 파일 이름과 크기를 설정해 생성해준다.

bool remove(const char \*file): 받은 인자로 filesys\_remove() 함수를 호출한다. filesys\_create() 함수는 받은 인자로 파일 이름으로 파일을 제거해준다.

int open(const char \*file) : 받은 파일명을 인자로 넘겨 filesys\_open 함수로 파일 포인터를 fp에 받는다. 이후 만약 NULL이면 -1을 리턴하고, 아니라면 반복문으로 I = 3 부터 127까지 돌면서 thread\_current()->fd[i]가 NULL이 아닌 최소값을 찾아서 반환한다. 이때 만약 현재 쓰레드와 이름이 같은 파일을 열려고 시도했다면 file\_deny\_write를 설정해준다. 핀토스에서는 실행중은 프로그램의 excutable file을 제거하는 것을 원하지 않기 때문이다. 위에서 인덱스를 설정할 때 2부터 할지 고민을 했었는데, 메뉴얼에서는 0, 1만 언급되었지만 그냥 배운대로 3부터 했다.

int filesize(int fd): 인자로 받은 fd로 file\_length를 호출해 size를 반환한다. file\_length는 받은 포인터로 data의 size를 반환한다.

int read(int fd, void \*buffer, unsigned size) : fd가 0이면 input\_getc()로 입력을 받고, 2보다 크면 file\_read로 파일을 size 만큼 읽어 buffer에 저장한다. 이후 읽은 바이트 수를 반환한다.

int write(int fd, void \*buffer, unsigned size) : fd가 1이면 putbuf()로 출력을 하고, 2보다 크면 file\_write로 파일에 buffer에 있는 내용을 size 만큼 저장한다. 이때 만약 deny\_write가 활성화되어 있다면, file\_deny\_write() 함수를 호출한다. 이 함수에서는 deny\_write가 활성화가 되어있지 않으면 아무 것도 하지 않는다.

void seek(int fd, unsigned position) : file\_seek 함수로 받은 fd가 가리키는 파일의 다음 읽거나 쓸 바이트를 position으로 이동한다.

unsigend tell(int fd) : file\_tell 함수로 받은 fd가 가리키는 파일의 다음 읽거나 쓸 바이트를 반환한다.

void close(int fd) : thread\_current()->fd[fd]에 NULL 값을 넣는다. 이후 file\_close 함수로 원래 있던 포인터를 닫아준다.

3. Synchronization in Filesystem: Syscall.c에서 전역변수로 struct lock read\_and\_write\_lock 생성한다. 그리고 syscall\_init을 할 때 lock\_init으로 같이 초기화해준다. 즉 처음 세마포어 값은 1이다. 이후 Open read write 함수에서 시작 시 lock\_acquire 함수를 호출한다. 이 함수에서 sema\_down() 함수를 호출하므로 다른 프로세스에서는 접근을 하려 해도 value가 0이니 락을 sema\_down이 정상적으로 종료가 되지 않아 락을 획득할 수 없게 된다. 이후 종료 전에 lock\_release()로 sema\_up()을 해주며 다시 락을 반환한다. 만약 락을 획득하고 싶어 했던 프로세스가 존재한다면 이때부터 획득이 가능할 것이다. 이를 통해 critical section에 동시에 여러 프로세스가 접근할 수 없음을 보장할 수 있게 되었다. 사실 맨 처음에는 시스템 프로그래밍 시간에 배웠던 reader&writer problem을 생각하며, 동시에 여러 프로세스가 read는 가능하게 구현을 할지 고려했었는데, 우선 간단히 하나의 lock만 구현한 결과 테스트 케이스를 모두 통과해 굳이 바꾸지는 않았다. 그러나 추후 필요하다면 충분히 바꿔줄 수 있을 것이라 생각이 든다. 또 파일시스템은 아니지만 동기화를 해준 부분이 하나 더 있다. 바로 프로세스와 자식 프로세스 사이에서 자식 프로세스가 load가 끝나기 전에 부모 프로세스가 너무 빨리 죽는 경우를 고려하지 않아 테스트 케이스를 통과하지 못한 부분이 있어 그 부분을 고쳤다. thread 구조체에 부모를 가리키는 쓰레드 포인터 parent를 추가해줬다. 또 load\_lock을 추가한 뒤 0으로 초기화도 해준다. 이후 start\_process를 할 때 자신의 부모 프로세스의 load\_lock을 sema\_up을 해준다. 이후 process\_excute에서 thread\_create 호출 이후 load\_lock을 sema\_down 해준다. 만약 여기 부분이 먼저 실행되어도 초기에는 0이므로 끝나지 않을 것이다. 추가로 load가 실패했을시 exit(-1)을 해준다. 이 경우 process\_excute에서 끝나기 전에 반복문을 돌며 자신의 자식의 종료상태가 -1인 경우 그 tid로 process\_wait을 호출하여 정상적으로 종료하고 자식 리스트에서 제거되도록 한다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명