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

담당 교수 / 분반 : 김영재 교수님 / 2반

이름 / 학번 : 20191138 / 이현규

개발 기간 : 10/15 ~ 11/08

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술
* create, remove, open, close, filesize, read, write, seek, tell 등 file system에 대한 System call을 구현한다. 또한, 한 파일에 여러 프로세스가 접근할 경우, 데이터 손상 및 시스템 불안정성이 발생할 수 있기 때문에 이에 대한 동기화 코드를 구현한다. 그리고, 실행 중인 executable file에 다른 프로세스가 접근하여 이를 수정하거나 삭제할 경우 오류가 발생할 수 있기 때문에 file\_deny\_write를 이용하여 이 파일을 보호한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

프로세스가 자신이 연 파일에 다시 접근할 때 이용할 수 있도록 file descriptor를 구현해야 한다. 이를 통해 프로세스는 open한 해당 파일에 대해 create, remove, open, close, filesize, read, write, seek, tell 등의 동작을 할 수 있다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

create, remove, open, close, filesize, read, write, seek, tell 등의 system call이 구현된다면, 프로세스는 각 파일을 생성, 제거, 열기, 닫기, 사이즈 조회, offset 이동, offset 조회를 할 수 있게 된다.

3. Synchronization in Filesystem

File system에서 여러 프로세스가 하나의 파일에 동시에 접근하여 읽기나 쓰기 작업을 한다면, 파일이 손상되고, 데이터 무결성에 영향을 미쳐 시스템이 불안정해질 수 있다. 이를 막기 위해서 semaphore 기반의 synchronization을 구현해야 한다. 이를 통해서 이러한 파일 오염 문제를 해결할 수 있다. 또한, 현재 실행되고 있는 실행 파일에 수정이 가해진다면 오류가 발생할 수 있기 때문에, 이 역시 막아야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

인덱싱과 관리가 용이한 배열을 사용하여 file descriptor를 구현하였다. PintOS 매뉴얼 35페이지의 ' You may impose a limit of 128 open files per process'라는 문장에 따라, linked list에 비해 구현이 용이한 128개의 배열을 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)

bool **create**(const char \*file, unsigned initial\_size);

filesys\_create를 호출해 file이라는 이름을 가진 initial\_size 바이트 크기의 파일을 생성한다. 파일 생성에 성공하면 true, 실패했다면 false를 반환한다. open과의 차이점에 유의해야 한다.

bool **remove**(const char \*file);

filesys\_remove를 호출해 file이라는 이름을 가진 파일을 삭제한다. 삭제에 성공했다면 true, 실패했다면 false를 반환한다. close와의 차이점에 유의해야 한다.

int **open**(const char \*file);

file이라는 이름의 파일을 filesys\_open를 통해 열고, 이를 현재 thread(thread\_current())의 file descriptor에서 STDIN, STDOUT을 제외한 [2]번째 부터 [127]번째까지 중 가장 먼저 비어있는 곳에 배치하며, 이곳의 인덱스를 반환한다. filesys\_open 과정에서 실패했거나, file descriptor에 빈 자리가 없는 경우 -1을 반환한다.

void **close**(int fd);

file descriptor의 [fd]번쩨 파일을 (존재 한다면) close한다. 프로세스를 exit하거나 종료할 경우에는 자동적으로 모든 fd를 close 해야한다.

int **filesize**(int fd);

현재 thread의 file descriptor[fd]인 파일의 크기를 file\_length(cur->fd[fd])를 통해 반환한다.

int **read**(int fd, void \*buffer, unsigned size);

fd가 0, 즉 STDIN이 아닌 경우를 구현하여, file\_read(thread\_current()->fd[fd], buffer, size)를 호출하여 현재 프로세스의 file descriptor[fd] 파일의 내용을 size 크기만큼 읽어와 buffer에 read 한다. 실제로 read된 파일의 크기(오류일 경우 -1)을 반환한다.

int **write**(int fd, const void \*buffer, unsigned size);  
fd가 1, 즉 STDOUT이 아닌 경우를 구현하여, file\_write(thread\_current()->fd[fd], buffer, size)를 호출하여 현재 프로세스의 file descriptor[fd] 파일에 buffer의 내용을 size 크기 만큼 write한다. 실제로 write된 크기를 반환한다.

void **seek**(int fd, unsigned position);  
fd 파일에서 새롭게 read/write될 위치를 position 바이트로 이동한다. 즉, 0일 경우 파일의 시작점이 되는 셈이다.

unsigned **tell**(int fd);

seek에 대한 조회와 비슷한 역할을 하는 system call로, file descriptor[fd] 파일의 읽거나 쓸 파일 시작점으로부터의 위치를 반환한다.

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

syscall.c에 struct lock file\_using\_lock;라는 lock을 구현하여, 한번에 단 한 프로세스만 파일에 관련된 작업을 할 수 있도록 하였다. 이를 통해 여러 프로세스가 파일에 접근하여 발생할 수 있는 데이터 일관성 문제와 충돌을 방지할 수 있다. 한번에 한 프로세스만 파일 접근을 얻고(acquire), 작업이 완료된 후 lock을 해제(release) 하며 다른 프로세스가 안전하게 파일을 사용할 수 있도록 하였다. 이를 통해 자연스럽게 synchorization이 구현되게 한다.

lock은 초기값이 1이고, lock을 잠근 프로세스만 그 잠금을 해제할 수 있다는 제약 조건이 추가된 semaphore라고 볼 수 있다. 따라서, 파일을 독점적으로 사용하고, 잠금/잠금 해제의 주체가 명확해야 하는 file system synchronization을 위해서는 semaphore보다는 lock이 더 적합하다. 하지만 lock 대신 1인 semaphore를 init하고, 이에 대해 lock\_acquire 대신 sema\_down, 끝난 후 sema\_up 하는 식으로도 (다른 프로세스가 잠금을 해제할 수 있다는 점을 감안하고) 구현할 수 있다.

case SYS\_WRITE:

is\_valid\_vaddr(esp + 5);

lock\_acquire(&file\_using\_lock);

f->eax = write(\*(int \*)(esp + 1), \*(const void \*\*)(esp + 2), \*(unsigned \*)(esp + 3));

lock\_release(&file\_using\_lock);

break;

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

**10/15~10/17:** Pintos Manual 학습

**10/18~10/22:** File Descriptor 구현

**10/23~10/29:** 새로운 System Call 구현

**10/30~11/05:** Synchronization 구현

**11/06~11/09:** 리팩토링 및 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
  + File Descriptor  
    가장 먼저 각각의 스레드가 자신과 관련된 파일을 추적할 수 있도록 src/threads/thread.h에 fd 배열을 선언할 것이다. 또한, 이 배열의 각 요소를 NULL로 초기화하는 코드를 src/threads/thread.c의 init\_thread()과 thread\_create()에 추가한다. file descriptor를 사용하는 과정에서 잘못된 접근의 문제가 일어날 수 있기 때문에, src/userprog/syscall.c에 해당 fd의 유효성을 체크하는 함수를 추가적으로 작성할 것이다.
  + System Calls  
    새로운 system call을 포함하도록 src/userprog/syscall.c의 syscall handler를 수정하고, 각각의 함수(read, write를 제외한 create, remove, open, close, filesize, seek, tell)도 pintos에 있는 filesys에 관련된 코드를 사용하도록 새로 작성할 것이다. read와 write의 경우 각각 STDIN, STDOUT이 아닌 경우에 return -1 하도록 되어 있는 부분을 수정하여 파일 read, write 기능을 구현할 것이다.
  + Synchronization 구현  
    src/userprog/syscall.c에 struct lock file\_using\_lock;를 추가하고, 선술했듯이 파일에 괸련된 system call 전에 lock\_acquire, 완료 후 lock\_release 하도록 추가적으로 작성해야 한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
* File descriptor  
  **스크린샷, 키보드이(가) 표시된 사진

  자동 생성된 설명**
* File System System Calls (기타 색상 부분)
* Synchronization (노란색 부분)

**스크린샷, 다채로움, 사각형, 라인이(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
* File Descriptor
  + file descriptor 배열 선언 (src/threads/thread.h)

struct thread

{

/\* Owned by thread.c. \*/

tid\_t tid; /\* Thread identifier. \*/

enum thread\_status status; /\* Thread state. \*/

... 생략

struct list\_elem elem; /\* List element. \*/

#ifdef USERPROG

/\* Owned by userprog/process.c. \*/

uint32\_t \*pagedir; /\* Page directory. \*/

// project 01

struct list\_elem child\_elem; // my parents' children

... 생략 ...

// project 02

struct file \*fd[128];

struct file \*executable\_file;

#endif

/\* Owned by thread.c. \*/

unsigned magic; /\* Detects stack overflow. \*/

};

* + struct thread 구조체에 각 thread가 open한 파일의 포인터의 배열 fd[128]을 추가하였다.
  + file descriptor 배열 초기화 (src/threads/thread.c)

for (int i = 0; i < 128; i++)

{

initial\_thread->fd[i] = NULL;

}

* + init\_thread()와 thread\_create()의 알맞은 부분(#ifdef USERPROG 블록 안)에 해당 코드를 작성하여, 첫 스레드나, 새로운 스레드의 file descriptor 배열들이 올바르게 초기화 되도록 하였다.
  + file descriptor 유효성 함수 추가 (src/userprog/syscall.c)

static void is\_valid\_fd(int fdPos)

{

if (thread\_current()->fd[fdPos] == NULL)

{

exit(-1);

}

}

* + 여러 파일 관련 system call에서 지속적으로 요구되는, 특정 file에 접근하기 전 해당 file descriptor가 존재하는지 확인하는 함수를 별도로 작성하였다.
* system calls
  + syscall handler 추가

case SYS\_CREATE:

{

const char \*file = \*(const char \*\*)(f->esp + 4); // 첫 번째 인자: 파일 이름

unsigned initial\_size = \*(unsigned \*)(f->esp + 8); // 두 번째 인자: 초기 크기

is\_valid\_vaddr(file); // 파일 이름의 유효성 확인

lock\_acquire(&file\_using\_lock);

f->eax = create(file, initial\_size); // 반환 값을 eax에 저장

lock\_release(&file\_using\_lock);

break;

}

* + 추가적인 system call을 해당 system call의 호출 양식에 맞게 switch 문을 작성하여 구현하였다. (보고서에는 create만 포함함.) 또한, 파일이 유효한 파일인지 사전에 검사한 후, system call을 호출하는 형태로 syscall\_handler를 작성하였다.

* + create

bool create(const char \*file, unsigned initial\_size)

{

return filesys\_create(file, initial\_size);

}

* + 파일에 대한 검증은 이미 syscall\_handler가 진행한 상태이기 때믄에, 곧바로 PintOS가 제공하는 filesys\_create를 사용하여 파일 create를 시도한다. 이 함수는 initial\_size 크기의 이름이 file인 파일을 생성하며, 이에 성공하면 true, 아니면 false를 반환한다.
  + remove

bool remove(const char \*file)

{

return filesys\_remove(file);

}

* + PintOS가 제공하는 filesys\_remove를 파일명 file을 인자로 호출하여, 해당 파일을 remove 한다. 성공할 경우 true, 실패할 경우 false를 반환한다.
  + open

int open(const char \*file)

{

// try to open the file

struct file \*fp = filesys\_open(file);

if (fp == NULL)

{

return -1; // fopen error detect

}

// get the most front spot with no file i fd

int loc;

struct thread \*cur = thread\_current();

for (loc = 2; loc < 128; ++loc)

{

if (cur->fd[loc] == NULL)

{

cur->fd[loc] = fp; // put it there

return loc; // finish

}

}

// no empty space in fd

return -1;

}

* + 우선 인자로 전달받은 file을 PintOS가 제공하는 filesys\_open()을 사용하여 열어 struct file \*fp에 저장한다. 이에 실패할 경우, -1을 반환한다. STDIN, STDOUT을 제외한 fd에서 가장 빠른 빈 공간을 찾아, 이곳에 해당 파일 포인터를 저장한다. 이 과정을 마무리 했다면, 저장된 fp position loc을 반환한다. 만약 모든 fd가 차있다면 for 문이 완수되고 -1이 반환된다.
  + close

void close(int fd)

{

struct thread \*cur = thread\_current();  
 file\_close(cur->fd[fd]);

cur->fd[fd] = NULL;

}

* + 현재 thread의 fd 번째 파일의 fd가 비어있지 않다면, 이를 close 해준 후, 다시 접근할 수 없도록, 일종의 비움을 나타내기 위해 fd[fd]를 NULL로 바꾼다. 즉 이 자리는 빈 자리가 되었기 때문에, 새로운 파일을 open한다면, (이 파일이 가장 앞쪽에 있다면) 지금의fd 자리가 다시 사용된다.
  + filesize

int filesize(int fd)

{

struct thread \*cur = thread\_current();

return file\_length(cur->fd[fd]);

}

* + 직관적으로, 현재 thread의 fd[fd] 파일의 크기를 PintOS 기본 제공 코드인 file\_length를 통해 반환한다.
  + read 개선

int read(int fd, void \*buffer, unsigned size)

{

is\_valid\_vaddr(buffer);

if (fd == 0)

{ // File descriptor 0 is stdin

... 생략

}

else // files

{

// Implementing with fd

is\_valid\_fd(fd);

return file\_read(thread\_current()->fd[fd], buffer, size);

}

}

* + fd==0이 아닐 경우, 즉 표준입출력이 아닌 파일에서 읽어오기를 실행하는 경우,  해당 file descriptor에 파일이 존재하는지 확인한 후, PintOS가 제공하는 file\_read를 호출하여 thread\_current()->fd[fd] 파일을 size만큼 읽어와 buffer에 저장한다.
  + write 개선

int write(int fd, const void \*buffer, unsigned size)

{

is\_valid\_vaddr(buffer);

if (fd == 1)

{ // File descriptor 1 is stdout

... 생략

}

else

{

// Implementing with fd

is\_valid\_fd(fd);

return file\_write(thread\_current()->fd[fd], buffer, size);

}

}

* + read와 마찬가지로, fd가 표준 입출력이 아닐 경우, 해당 file descriptor에 파일이 존재하는지 확인한 후, PintOS가 제공하는 file\_write를 호출하도록 하였다.
  + seek

void seek(int fd, unsigned position)

{

struct thread \*cur = thread\_current();

file\_seek(cur->fd[fd], position);

}

* + 시작점 포인터를 수정하고자 하는 파일 인덱스 fd와 수정하고자 하는 위치인 position을 인자로 받아와, 현재 thread의 fd[fd] 파일의 시작점을 position 만큼 이동할 수 있는 PintOS 제공 함수인 file\_seek()을 호출한다.
  + tell

unsigned tell(int fd)

{

struct thread \*cur = thread\_current();

return file\_tell(cur->fd[fd]);

}

* + seek()한 결과값인 position(초창기에는 0)을 반환할 수 있도록, PintOS 기본 함수인 file\_tell을 사용하여 구현하였다. 이 지점부터 read와 write가 이루어진다.
* synchronization 구현
  + 구현 방식  
    구현이 간단한 방식인 lock을 사용하여, 파일에 관련된 부분을 함부로 동시에 사용하지 못하도록 하였다. 구현을 위해, syscall.c 파일의 가장 위에 Lock인 file\_using\_lock을 정의하였다.

#include "userprog/syscall.h"

#include <stdio.h>

#include <syscall-nr.h>

#include "threads/interrupt.h"

#include "threads/thread.h"

#include "threads/vaddr.h"

// project02

#include "filesys/filesys.h"

#include <stdbool.h>

struct lock file\_using\_lock;

static void syscall\_handler(struct intr\_frame \*);

* + lock은 일종의 initial value가 1인 semaphore와 같다. lock\_release는 sema\_up, lock\_acquire는 sema\_down과 같다고 보면 된다. semaphore와의 차이점은, lock은 잠근 사람만이 이를 풀 수 있다는 것이다.
  + lock은 void lock\_init (struct lock \*lock)을 통해 시작되어야 하기 때문에, 모든 syscall 작동 이전에 작동되는 void syscall\_init()에 lock\_init(&file\_using\_lock);를 추가하여 초기화 하였다.
  + lock\_acquire(file\_using\_lock)가 호출된다면, 현재의 주인이 이 lock을 release 할 때 까지 기다린다. 따라서, 한번에 한 프로세스만 파일에 접근할 수 있는 식으로 수정하였기에 동기화 문제를 해결할 수 있다.
  + 과정 예시

case SYS\_WRITE:

is\_valid\_vaddr(esp + 5);

lock\_acquire(&file\_using\_lock);

f->eax = write(\*(int \*)(esp + 1), \*(const void \*\*)(esp + 2), \*(unsigned \*)(esp + 3));

lock\_release(&file\_using\_lock);

break;

* + 파일과 관련된 system call이 호출되었을 경우, 파라미터argument들이 모두 정상적인 경우, lock\_acquire(&file\_using\_lock);을 통해 file\_using\_lock이 나에게 확보되기를 기다린다.
  + 그 후, 해당하는 함수를 작동시킨다.
  + 작동이 완료된 후, lock\_release(&file\_using\_lock)을 통해 lock을 해제하고, 다른 프로세스도 파일을 사용할 수 있도록 한다.
* 실행 파일 쓰기 방지  
  프로세스가 실행 중일때 그 프로세스의 실행 파일이 수정되지 않도록 하기 위해, 자신의 실행 파일을 관리하는 파일 포인터 struct file \*executable\_file;를 추가적으로 정의하고, 이를 process.c의 두 함수에서 file\_deny\_write, file\_allow\_write를 통해 방지하였다..
  + 실행 파일에 쓰기 금지 - src/userprog/process.c load 함수에서 file 수정 금지하기

/\* Open executable file. \*/

file = filesys\_open(program\_name);

if (file == NULL)

{

goto done;

}

// no writing on executables

file\_deny\_write(file);

t->executable\_file = file;

* + 프로세스가 파일을 열 때, 쓰기 방지를 적용한다. file = filesys\_open 이후에 해당 파일에 file\_deny\_write를 사용하여 쓰기를 금지한 후, 이를 해제할 수 있도록 현재 thread의 executable\_file에 포인터를 저장하였다. 해당 포인터 역시 fd 배열처럼, init\_thread와 thread\_create()에서 NULL로 일차적으로 초기화된다.
  + 실행 파일에 쓰기 제한 해제하기 - src/userprog/process.c process\_exit 함수에서 file 다시 수정 허용하기

for (int i = 0; i < 128; i++)

{

file\_close(cur->fd[i]);

cur->fd[i] = NULL;

}

if (cur->executable\_file != NULL)

{

file\_allow\_write(cur->executable\_file);

file\_close(cur->executable\_file);

cur->executable\_file = NULL;

}

* + 프로세스를 종료할 때, 이 프로세스의 file descriptor를 해제할 뿐만 아니라, 이 프로젝트에 실행가능한 파일이 지정되어 있다면, 이에 대해 file\_allow\_write 한 후, 이 파일을 닫고, 자신의 실행파일 포인터를 NULL로 저장하여 참조 문제를 예방한다.

**C. 시험 및 평가 내용**

* make check 수행 결과를 캡처하여 첨부  
  텍스트, 스크린샷, 소프트웨어, 컴퓨터이(가) 표시된 사진

  자동 생성된 설명