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

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070] 운영체제

담당 교수 : 김영재 교수님

조 / 조원 : 46조, 20141284, 20141340

개발 기간 : 2019/11/2 ~ 2019/11/17

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

**제출일 : 2019년 11월 17일**

**참여 조원 : 이기현, 장주호**

1. **개발목표**

* Project 1에서 진행했던 User Program의 연장선으로 지난 프로젝트에서 구현하지 않았던 seek, close, tell, filesize, read, write, open 등을 구현한다.   
  또한 read, write 수행에 있어서 read, write가 critical section에서 동시에 수행되지 못하도록 synchronization 기법을 적용한다.   
  또한 실행 중인 프로그램의 executable file이 지워지지 않도록 file\_deny\_write 함수를 구현한다.

1. **개발 범위 및 내용**
2. **개발범위**

/threads/ thread.h thread.c

/userprog/ syscall.c process.c

* **System Calls**

Read, write, open, close, seek, tell, filesize, create를 구현한다.

* **Synchronization**

Critical Section 내부에서 동일한 파일에 read, write혹은 write, write가 동시에 일어나지 못하도록 해야한다.

* **Denying Writes to Executables**

프로세스가 메모리에 로드된 후 실행되더라도 실행 중인 프로그램의 executable file이 지워지지 않도록 한다. File\_deny\_Write()함수를 통해서 구현한다.

1. **개발 내용**
2. **1) System Call**

create, remove, open, close, filesize, seek, tell 을 추가하고, project1에서 추가했던 read, write를 수정한다.

- create : 첫번째 parameter는 file의 이름이고, 두번째 parameter는 파일의 size이다.

- remove : parameter를 이름으로 갖는 file을 제거한다.

- open : parameter를 이름으로 갖는 file을 찾아 open해준다.

- close : parameter에 해당하는 file description의 file을 close해준다.

- filesize : parameter에 해당하는 file description의 file의 size를 return해준다.

- seek : 첫번째 parameter에 해당하는 file description의 file을 찾아 그 file의 위치를

두번째 parameter에 해당하는 position으로 옮겨준다.

- tell : parameter에 해당하는 file을 찾고 그 file의 현재 위치를 return해준다.

- read : 기존에는 fd값이 0인 STDIN을 읽어오는 동작만 구현되어 있었는데, 여기에 fd>2인 다른 file들을 read 동작을 추가 구현한다.

- write : 기존에는 fd값이 1인 STDOUT을 출력하는 동작만 구현되어 있었는데, 여기에 fd>2인다른 file들을 write하는 동작을 추가 구현한다.

**2) Synchronizing**

Synchronizing은 semaphore을 이용한다. Semaphore는 어떤 thread가 critical section에 접근했을 때 다른 스레드가 해당 critical section에 접근하지 못하도록 해주는 역할을 한다.   
sema\_down() 과 sema\_up()을 통해서 critical section에 접근한 쓰레드를 block하고 unblock하는 방법을 구현한다.   
**3) Denying Writes to Executables**

이번 프로젝트에서는 pintos가 실행하는 중에, 실행중인 program의 executable file이 지워지지 않도록 file\_deny\_write함수를 사용해서 구현한다. file이 load될 때, file\_deny\_write 함수를 호출하여 file에 쓰기작업을 하지 못하도록 설정하고, thread가 종료될 때, file에 쓰기를 허용함으로써 executable file이 지워지지 않도록 구현한다

1. **추진 일정 및 개발 방법**
2. **추진 일정**

11/2 (토) ~ 11/6 (수) : ppt 및 매뉴얼 분석

11/7 (목) ~ 11/10 (일) : system call (read, write, tell, seek, create, remove, open)구현

11/11 (월) ~ 11/14 (목) : 코드 디버깅 및 wait 관련 문제 해결

11/15 (금) ~ 11/17 (일) : 최종 완성 및 보고서 작성

1. **개발 방법**

* **System Call**

**- create( )** : file 이름을 인자로 받아서 만약 file 이름이 NULL이라면 exit(-1)을 호출하여 thread를 종료. NULL이 아니라면 filesys\_create 함수를 호출한다. Filesys\_create 함수의 인자로는 파일이름과 초기 크기이며, create를 성공하면 true를, 실패하면 false를 return한다.

**- remove( )** : file 이름을 인자로 받아서 filesys\_remove 함수를 호출한다. 해당 이름과 일치하는 file을 제거하고, 성공하면 true를, 실패하면 false를 return한다.

**- open( )** : file이름을 인자로 받아서 최초에 NULL일 경우 exit(-1)을 수행한다.   
 그 후 임시 file구조체를 생성하여 입력받은 file을 인자로 filesys\_open을 수행한다. 그리 고 나서 다른 스레드에서 이미 open한 적이 있는 파일이라면 file\_deny\_write 함수를 수행해주고 현재 스레드의 file descriptor리스트에 해당 파일을 추가해준다.

**- close( )** : fd값을 인자로 받아서 thread의 file에 대한 정보를 갖는 구조체에서 fd에 해당하는 file을 찾아 closed 되었다는 의미인 0을 file index에 할당해주고, file을 NULL로 바꾸어 준다. 그 후, file\_close 함수를 호출하여 해당 file을 close시켜준다.

**- filesize( )** : fd값을 인자로 받아서 fd에 해당하는 file을 parameter로 갖는 file\_length 함수를 호출한다. 그 후, file의 size를 return해준다.

**- seek( )** : fd 값과 position을 인자로 받아서 fd에 해당하는 file의 현재 위치를 인자로 받은 position으로 옮겨준다. 이를 위해서 file\_seek 함수를 호출해준다.

**- tell( )** : fd (file description)를 parameter로 받아서 fd에 해당하는 file을 parameter로 갖는 file\_tell 함수를 호출한다. 그 후 그 file의 현재 위치를 return 해준다.

**- read( )** : project 1에서 구현했던 read 함수에 fd가 3~127 사이의 값을 갖는 경우, 즉 standard input이 아니라 file을 읽는 경우를 추가해 준다. 만약 fd가 3~128 사이의 값인 경우라면, fd에 해당하는 file이 NULL인 경우에는 -1을 return, 잘못된 address를 접근하는 경우에는 exit(-1)을 호출한다. 이 경우들에 포함되지 않는다면 file을 올바르게 read할 수 있는 경우이며, file\_read함수를 호출하고 이 함수의 return 값을 read( ) 함수의 return값으로 설정해준다. File을 읽을 때, file에 쓰지 못하도록 동기화 시켜 주기 위해서 file\_read 함수 앞뒤로 lock을 걸어주어 구현한다.

* **- write( )** : project 1에서 구현했던 write 함수에 fd가 3~127 사이의 값을 갖는 경우, 즉 standard output이 아니라 file에 쓰는 경우를 추가해 준다. read( )와 비슷한 방법으로 file이 NULL인 경우에는 -1을 return하고, file이 NULL이 아니라면 file\_write 함수를 호출하여 file에 작성한다. 마찬가지로 write를 하는 도중에 read를 할 수 없도록 file\_write 함수 앞뒤로 lock을 걸어주어 구현한다.
* **Synchronization**

synch.c에 구현된 sema\_down 및 sema\_up 의 내용은 다음과 같다. sema\_down은 value가 0이 되면 그 value가 0을 벗어날 때 까지 thread를 block시키고 해당 semaphore의 waiter list에 현재 thread를 추가하는 함수이다. while문을 벗어나면 value를 다시 감소시켜준다. sema\_up은 반대로 waiter list가 NULL이 아니면 waiter 중 하나를 pop한 뒤 unblock시켜주고, 이후 semaphore value를 증가시켜주는 함수이다. 따라서 read나 write함수 등 critical section에 접근하는 부분 전후로 사용한다.. 일반적으로는 semaphore을 1로 초기화해 현재 thread가 임계 영역에 들어가면서 다른 thread의 접근을 막지만 wait 및 exit에서는 wait함수에서 자식 thread가 종료되기전 까지 현재 thread를 block해야하기 때문에 semaphore을 0으로 초기화함에 주의한다. 이후 자식 thread가 exit하기 직전 semaphore 값을 증가시킨다.

* **Denying Writes to Executables**

File이 동일한 thread에서 여러 번 open 되었을 경우를 대비한다  
file\_close시에 file\_allow\_write함수는 적용이 되도록 pintos가 기본적으로 시스템화 되어있기 때문에 open시에 두번 이상 open되었다면 file\_deny\_write()함수를 사용하기만 하면 된다. .

**연구원 역할 분담**

이기현 – system call(read, create, remove, close)구현

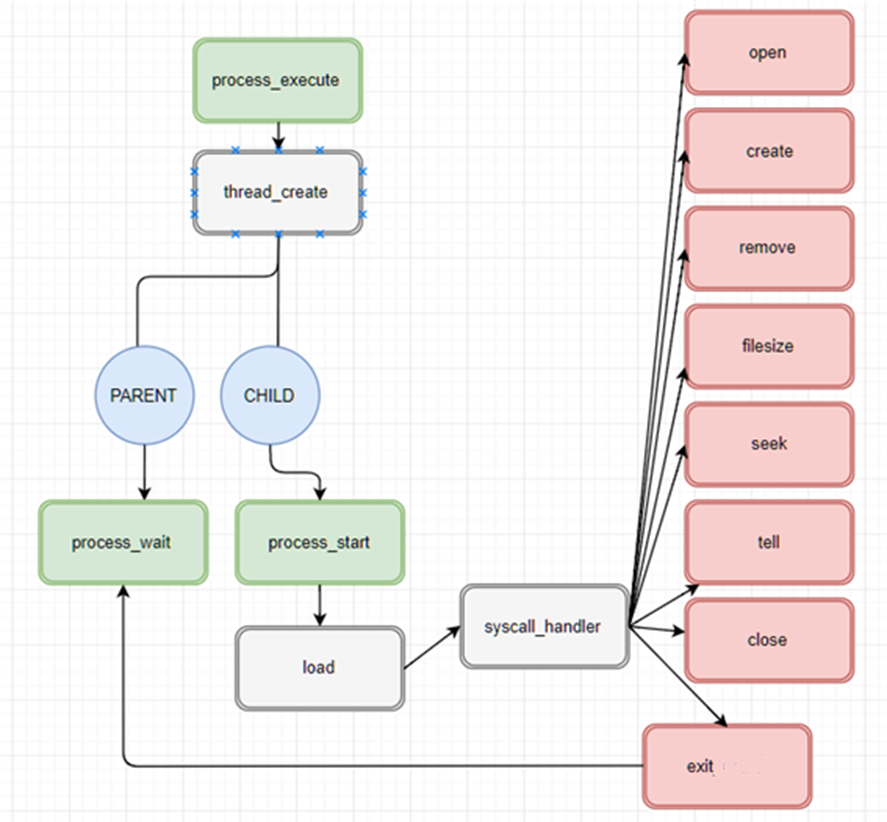
synchronization구현

보고서 작성

장주호 – system call(write, seek, tell, open)구현

보고서 작성

1. **연구결과**
2. **합성 내용**

****

1. **제작 내용**

**/userprog/syscall.c**

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

입력받은 fd값에 해당하는 file에 write작업을 수행한다. fd값이 1인경우 STDOUT을수행하고 아닐경우 fd값에 해당하는 file\_write를 수행한다.  
2=<fd <128 인 경우 synchronization을 수행하기 위해 lock\_acquire을 수행하고 시작한다.  
만약 현재 수행중인 쓰레드에 포함되어 있는 file의 fd값에 해당하는 fd가 들어오지 않으면 -1을 return한다.  
file\_write를 수행하고 난 후 lock\_release를 수행하고 file\_write의 결과값을 return한다.  
0<=fd<128에 포함되지 않는 fd값이 들어온 경우 아무 작업도 수행하지 않고 -1을 return한다.

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

if(fd == 1){

putbuf(buffer, size);

return size;

}

else if(fd >= 2 && fd < 128){

struct file\* cur\_file;

int ret;

cur\_file = thread\_current()->filelist[fd];

if(cur\_file == NULL) {

return -1;

}

lock\_acquire(&sys\_lock);

int ret = file\_write(cur\_file, buffer, size);

lock\_release(&sys\_lock);

return ret;

}

return -1;

}

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

입력받은 fd값에 해당하는 fiile에 대한 read작업을 수행한다.   
0<=fd<128범위에서 진행되며 fd값이 0인경우 STDIN을 수행, 다른 fd값의 경우 해당 fd값과 일치하는 file\_read작업을 수행한다.  
우선 입력받은 buffer와 size의 합이 커널 메모리 영역을 침범하는지 확인하고 침범시에는   
exit(-1)을 수행한다.  
write와 마찬가지로 synchronization을 위해서 작업을 수행하기전 lock\_acquire를 수행하고 시작한다.   
그 후 STDIN에 대해서는 임시 버퍼를 만들어서 read작업을 수행하고   
다른 fd값은 file\_read함수를 수행한다.   
현재 read해야하는 fd값의 파일이 NULL이라면 -1을 return 하고 아니라면 작업을 마친다.   
작업을 마치고 나면 마찬가지로 lock\_release를 수행하고 file\_read의 결과값을 return 해준다.

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

{

if(!is\_user\_vaddr(buffer+size)) exit(-1);

if (fd == 0) {

void \*temp = buffer; int i;

for (i = 0; i < (int)size; i++) {

\*(uint8\_t \*)temp = input\_getc();

if (\*(uint8\_t \*)temp == '\0') break;

(uint8\_t \*)temp++;

}

return i;

}

else if( fd >= 2 && fd < 128) {

struct file \* cur\_file;

int ret;

cur\_file = thread\_current()->filelist[fd];

if(cur\_file == NULL){

return -1;

}

lock\_acquire(&sys\_lock);

ret = file\_read(cur\_file, buffer, size);

lock\_release(&sys\_lock);

return ret;

}

return -1;}

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

입력받은 이름의 file에 대한 open 을 수행한다.   
입력값이 NULL인 경우 exit(-1)을 수행한다.   
임시 파일 구조체를 선언해서 filesys\_open(file)을 수행하고 해당 결과를 임시 파일 구조체에 저장한다.   
만약 filesys\_open의 결과값이 NULL일 경우 -1을 return 한다.   
또한 현재 open한 파일이 이미 동일 스레드에서 open했다면 file\_deny\_write()함수를 수행해서 실행중인 프로그램의 executable file이 지워지지 않도록 한다.   
또한 현재 돌아가고 있는 thread의 filelist에 open한 file을 추가하고   
스레드의 파일의 개수를 check 하는 filecnt++를 한다. 마지막으로 fd값을 리턴하며 종료한다.

int open(const char \*file){

if(!file) exit(-1);

struct thread \*cur;

struct file\* open\_file;

int i;

cur = thread\_current();

open\_file = filesys\_open(file);

if(open\_file == NULL) {

return -1;

}

if(!strcmp(thread\_name(), file)) file\_deny\_write(tmp);

for(i = 2 ; i<128 ; i++){

if(thread\_current() -> filelist[i] == NULL){

cur -> filelist[i] = tmp;

cur -> filecnt++;

return i;

}

}

return -1;

}

**int filesize(int fd)**

입력받은 fd값에 해당하는 file의 size를 return해준다.

int filesize(int fd){

return file\_length(thread\_current()->filelist[fd]);

}

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

입력받은 file과 size를 create한다.   
 입력 file 값이 없을 경우 exit(-1)을 create에 성공했을 경우 true, 아니면 false 를 return한다.

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

if(!file) exit(-1);

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

}

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

입력받은 file을 remove한다..   
입력값이 없을 경우 exit(-1)을, remove에 성공했을경우 true, 아니면 false를 return 한다.

bool remove(const char\* file){

if(!file) exit(-1);

return filesys\_remove(file);

}

**void close(int fd)**

입력받은 fd에 해당하는 file을 close해준다. file\_close에서 메모리 해제 후, NULL처리를 해주지 않기 때문에 별도로 현재 스레드의 filelist에서 해당 fd를 비워둔다.

void close(int fd){

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

if(cur->filelist[fd] != NULL) {

cur->filecnt--;

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

cur->filelist[fd] = NULL;

}

}

**void seek(int fd, unsigned position)**

fd에 해당하는 file의 현재 위치를 인자로 받은 position으로 옮겨준다. 이를 위해서 file\_seek 함수를 호출해준다.

void seek(int fd, unsigned position){

file\_seek(thread\_current()->filelist[fd], position);

}

**unsigned tell(int fd)**

fd에 해당하는 file을 인자로 갖는 file\_tell 함수를 호출한다. 그 후 그 file의 현재 위치를 return 해준다.

unsigned tell(int fd){

return file\_tell(thread\_current()->filelist[fd]);

}

**void exit(int status)**

thread의 exit시에 열려있는 파일을 모두 닫아준다.

void exit(int status) {

struct thread\*cur;

int i;

cur = thread\_current();

cur->exit\_status = status;

printf(“%s: exit(%d)\n”, cur->name, cur->exit\_status);

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

if(cur->filelist[i] != NULL)

close(i);

thread\_exit();

}

**struct lock syslock**

write, read 작업 수행시에 critical section에서의 race condition을 방지하기 위해 lock을 수행하기 위해서 전역변수로 syslock 선언.

#include "threads/thread.h"

#include "threads/vaddr.h"

#include "filesys/file.h"

struct lock sys\_lock;

**void syscall\_init(void)**

기존에 사용하던 syscall\_init함수 내부에 lock\_init을 추가해서 read, write 함수 내부에서 lock\_acquire, lock\_release를 사용 가능 할 수 있게 조치.

Void syscall\_init (void)

{

lock\_init(&syslock);

intr\_register\_int (0x30, 3, INTR\_ON, syscall\_handler, "syscall");

}

**/threads/thread.h**

**struct thread**

자식 thread의 load를 부모 thread가 기다리기 위한 load\_sema

자식 thread가 부모의 wait이전에 exit되어 zombie thread가 되는 걸 방지하기 위한 wait\_sema

프로세스당 open가능한 파일을 128개라고 가정하여, 현재 open된 file의 개수인 filecnt와 open된 file들의 포인터를 저장하는 filelist

#ifdef USERPROG

….

struct semaphore load\_sema;

struct semaphore wait\_sema;

int filecnt;

struct file\* filelist[128];

….

**/threads/thread.c**

**intit\_thread**

load\_sema와 wait\_sema 를 초기화해주고 STDIN과 STDOUT이 open이라는 가정하에 filecnt를 2로 초기화 해주고 2~127의 fd를 갖는 file 포인터들을 NULL로 초기화한다.

static void init\_thread (struct thread \*t, const char \*name, int priority) {

…

sema\_init(&(t->load\_sema), 0);

sema\_init(&(t->wait\_sema), 0);

t->filecnt = 2;

for(i = 2; i < 128; i++) t->filelist[i] = NULL;

…

}

**/userprog/process.c**

**process\_execute**

자식 thread의 생성 후, 자식 thread가 load를 끝낼 때까지 sema down 시켜준다.

tid\_t process\_execute (const char \*file\_name) {

…

tid = thread\_create(tmp, PRI\_DEFAULT ,start\_process, fn\_copy);

sema\_down(&(tid\_thread(tid)->load\_sema));

…

}

**start\_process**

자식 thread에서 load를 마무리하고 성공여부를 받은 이후에, sema\_up 을 통해 process execute 에서 down되어있는 부모 thread를 다시 살린다.

static void start\_process (void \*file\_name) {

…

success = load(file\_name, &if\_.eip, &if\_.esp);

if(!success) {

thread\_current()->tid = -1;

sema\_up(&(thread\_current()->load\_sema));

exit(-1);

}

else {

sema\_up(&(thread\_current()->load\_sema));

}

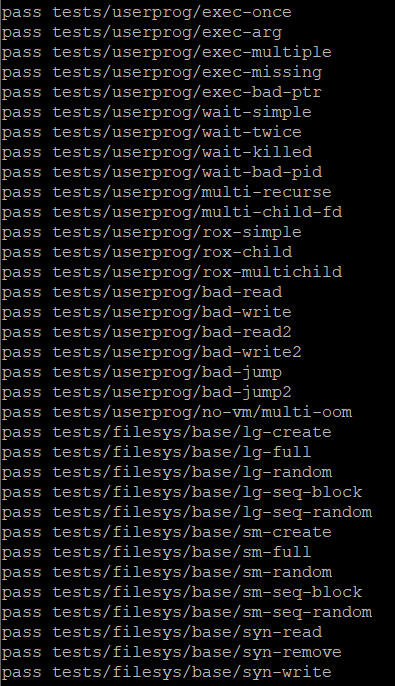
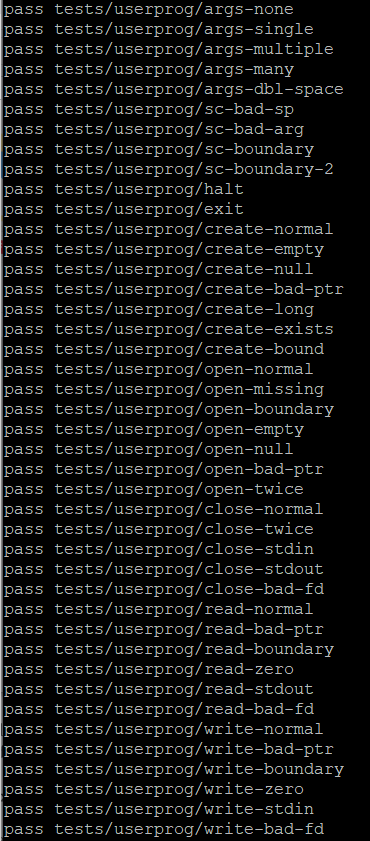
…

}

1. **시험 및 평가 내용**

* test 결과

55 / 55



1. **기타**
2. **연구 조원 기여도**

이기현 : 50%

장주호 : 50%

1. **소감**

이기현 : 지난번 프로젝트 만큼의 충격과 공포는 아니었지만 이번 프로젝트 또한 완벽하게 수행하기에는 여러 장애물들이 있었다.   
프로젝트 내용 이해나 기본적인 system call 구현은 수월하게 진행하였으나 make check 시에 세부적인 몇 개의 test에서 계속해서 fail이 나와서 적잖이 당황했다.   
특히 syn-read, syn-write 테스트에서 완벽하게 구현을 했다고 생각했음에도 계속 fail을 해서 완성시키는데 며칠을 할애했다.   
다행스럽게도 조교님들의 질문시간을 적절하게 활용해서 좋은 팁들을 얻을 수 있었고 문제를 해결 할 수 있었다. 또한 팀원과의 협업으로 낭비되는 시간 없이 문제를 해결 할 수 있었다.

장주호 : 개인적으로 시작하기 전에는 저번 프로젝트에 사용했던 시간에 절반정도면 사용하면 금방 마무리 지을 수 있다고 생각했다.   
저번 프로젝트에서 이미 구현한 뼈대 위에 새로운 코드를 얹는다고 생각해서 금방 할 수 있다고 생각했기 때문이다.   
시작은 순조로웠으나 역시 쉽게 마무리 되지는 않았다.  
중간에 몇가지 테스트가 통과되지 않아서 교착상태에 빠졌을 때 조교님들의 도움으로 힌트를 얻을 수 있었고, 팀원과의 정보공유 및 격려를 통해서 문제를 해결했다.   
여러모로 배울 수 있는게 많았던 프로젝트였다.