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

담당 교수 / 분반 : 박성용 교수님

이름 / 학번 : 20201635 전찬

개발 기간 :2022.10.05 – 2022.10.14

1. **개발 목표**

이번 프로젝트 2에서는 이전에 구현했던 기본적인 pintos의 system call에서 더 나아가, 여러 system call을 구현해야 한다. 또한 구현한 system call들에 대해서도 프로젝트 1과 동일하게 kernel memory에 접근하는지, 혹은 비정상적으로 종료되었는지, 여러 child process을 만들 수 있는지 등을 파악하며, 이를 보장하도록 system call을 구현하며, 이전에 구현한 함수들을 수정해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

**1. File Descriptor**

project 1에서 구현한 pintos는 STDIN로의 read, STDOUT로의 write, 어떠한 process의 execute 등의 간단한 작업만 수행할 수 있는 운영체제이다. 하지만 실제 program은 디스크 상에 존재하는 file에서 read/write 작업을 원하는 경우가 대부분이다. 이를 구현하기 위해서 file system call을 구현해야 할 뿐만 아니라, 각각의 open한 file에 대한 정보를 저장할 수 있는 구조체가 필요한데, 이를 수행할 수 있는 것이 File Descriptor 이다. 이를 통해 각 file을 어떠한 정수 값을 바탕으로 접근할 수 있다.

**2. (이번 프로젝트에서 구현해야 하는) System Calls**

이번 프로젝트에서 구현해야 하는 system call은 file을 다루는 system call이다. 이러한 system call들은 STDIN로의 입력, STDOUT로의 출력 뿐만 아니라, 어떠한 user program이 디스크 상에 존재하는 file에서 read/write을 수행할 수 있도록 해 준다. 따라서 이를 구현함을 통해 file을 open, read, write, close 등을 수행할 수 있다.

**3. Synchronization in Filesystem**

어떠한 한 file에 대해, 여러 process가 해당 file의 수정을 요청하는 경우, 의도치 않은 결과가 발생할 수 있다. 따라서 각 process들을 동기화 시켜 문제가 발생하지 않도록 구현해 주어야 한다. 이를 통해 여러 process가 한 file에 동시에 연결할 때, 발생할 수 있는 문제들을 방지할 수 있다.

* 1. **개발 내용**

**1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유**

초기 pintos는 file descriptor이 구현되어 있지 않다. 따라서 각 process마다 file descriptor을 만들어 주어야 하며, 이는 struct thread 내부에 추가해 주어야 함을 파악할 수 있다. 따라서 각 file을 point하는 file pointer array를 통해 file pointer을 관리하며, pintos manual에 따라 최대 open 개수가 128이므로, struct file\* fd[128]; 을 각 thread의 file descriptor로 구현할 수 있다. 이러한 array을 자료구조로 선택한 이유는, file descriptor은 최대 개수가 고정되어 있는 형태이기 때문에, 접근 속도가 빠른 array을 선택했다.

**2. System Calls: 구현할 각 system call**

이번 프로젝트에서 구현할 system call은 각각 다음과 같다.

- create : 새로운 file을 만들어내는 system call 이다.

- remove : 어떠한 file을 제거하는 system call 이다.

- open : 존재하는 file을 open하는 System call 이다. 현재 process의 fd(file descriptor) table에서 빈 공간을 찾아, 해당하는 file을 할당하는 작업을 수행한다.

- filesize : 어떠한 file의 총 크기를 알려주는 system call이다.

- read : 프로젝트 1에서 구현했던 STDIN로의 입력에서 더 나아가, 어떠한 file에서 data을 읽어오는 system call 이다.

- write : STDOUT로의 출력 뿐만 아닌, file에 해당 내용을 write하는 system call 이다.

- seek : 해당 file에서 원하는 위치로 이동하는 system call 이다.

- tell : file에서 현재 위치가 어디인지 알려주는 system call 이다.

- close : open 했던 file을 닫아주며, fd table에서 삭제하는 system call 이다.

추가로 이전에 구현했던 exit system call에서 해당 process을 exit할 때, fd table의 모든 file을 close할 수 있도록 구현해야 한다.

**3. Synchronization in Filesystem: Lock, Semaphore의 이용 방법**

file system에서 synchronization을 구현하기 위해 lock 또는 semaphore을 사용할 수 있다. semaphore는 일반적인 형태로, 해당 critical region에 접근하는 process의 개수를 원하는 만큼 지정할 수 있다. 하지만 이번 프로젝트에서는 한 file에 대해 배타적으로 하나의 process만 read / write을 수행할 수 있도록 구현해야 하기 때문에, mutex와 같은 역할을 수행하는 lock 을 통해 이를 구현할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10.05 – 10.07 : system call 구현

10.08 – 10.09 : test 결과를 바탕으로 system call, 이전에 구현한 함수 수정

10.09 – 10.12 : synchronization 구현

10.13 – 10.24 : 구현 레포트 작성

* 1. **개발 방법**

**1. File Descriptor**

file descriptor을 구현하기 위해, 위에서 이야기한 것처럼 structure file\* fd[128] array을 thread.h에 있는 struct thread 구조체 내에 추가해야 한다. 또한 이러한 array을 init\_thread을 수행할 때 각각의 pointer가 NULL이 되도록 설정해 주어야 한다. 추가로 open, close, exit system call 에서, file descriptor의 관리를 적절한 형태로 해 주어야 한다.

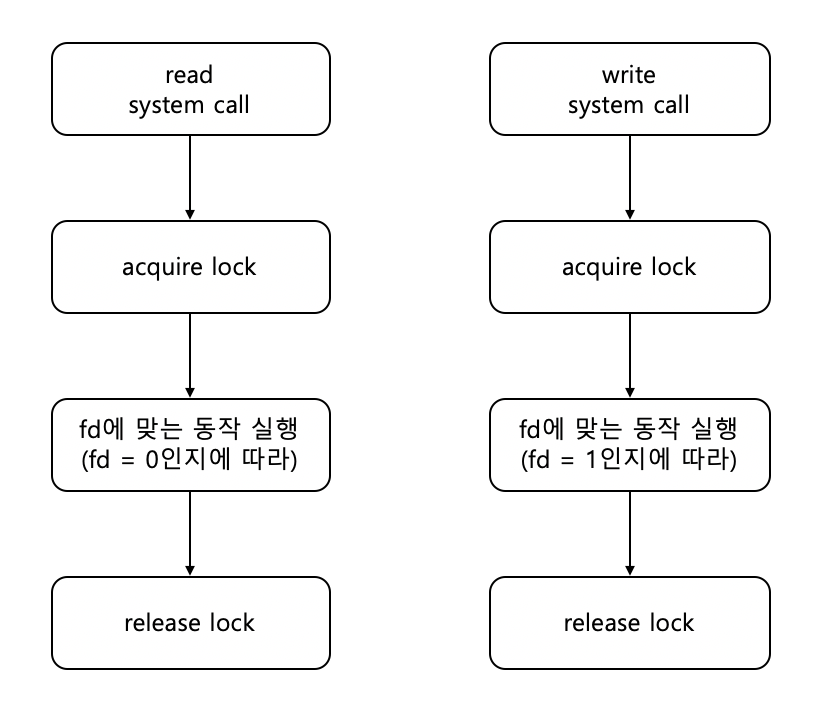
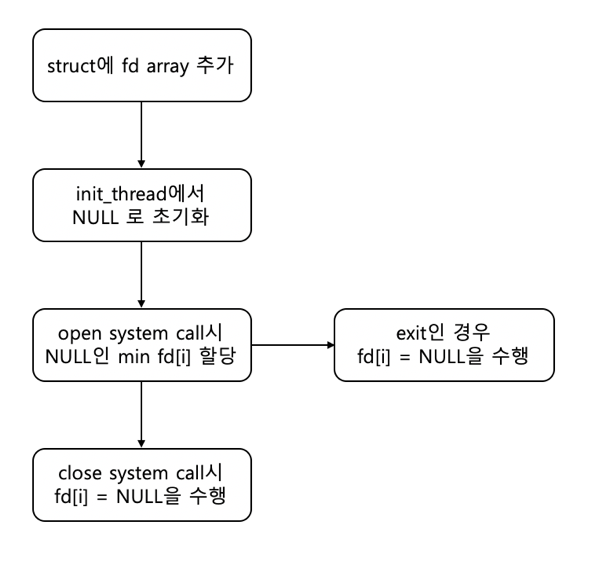
**2. System Calls**

각 system call을 userprog/syscall.c에 구현한다. 이번에 구현한 system call 또한, parameter이 user vaddr에 존재하는지를 확인해야 하며, 추가로 buffer 등의 pointer 등이 parameter로 전달되는 경우, 해당 pointer 또한 user 영역인지를 확인해야 한다. 특히 read, write인 경우, pintos 자체적으로는 STDIN, STDOUT, STDERR에 대해 특별한 조치를 취하지 않기 때문에 조건문을 통해 fd = 0, fd = 1, fd = 2 인 경우에 대해서는 프로젝트 1에서 구현한 코드를 사용하고, 이외에는 새롭게 코드를 구현해 주어야 한다.

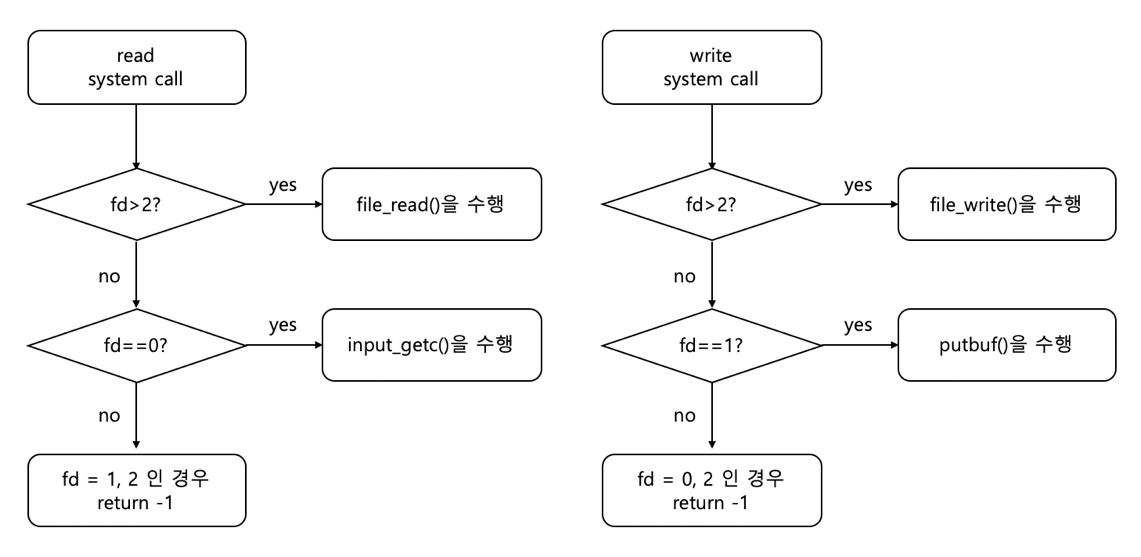
**3. Synchronization in Filesystem**

filesystem에서의 synchronization을 구현한다는 것은, 어떤 프로세스가 read / write 을 수행할 때 다른 process가 read / write을 하지 못하게 하는 것이다. 따라서 모든 process에서 사용할 수 있는 lock이 존재해야 하며, 이는 userprog/syscall.c에 전역 변수 형태로 구현할 수 있다. struct lock syn\_lock; 을 설정하며, syscall\_init 함수에서 lock\_init(&syn\_lock);을 설정해야 한다. 각각 read / write 함수의 시작, 끝 부분에 lock\_acquire(&syn\_lock), lock\_release(&syn\_lock)을 수행하는 형태로 synchronization 문제를 해결할 수 있다. 또한 이러한 함수를 사용하기 위해, “threads/synch.h” 을 먼저 #include 해주어야 한다.

1. **연구 결과**
   1. **Flow Chart**



<1. file descriptor flowchart> <3. synchronization flowchart>



<2. 수정한 read / write system call>

* 1. **제작 내용**

이번 프로젝트에서 구현한 struct thread에서 추가된 부분은 다음과 같다.

/\* project 2의 load-process execute 문제를 해결하기 위해 추가적으로 정의한다. \*/

struct semaphore load\_lock;

struct thread\* parent;

int flag;

/\* project 2 에서 추가되는 부분이다. \*/

struct file\* fd[128];

struct file\* file\_pointer;

<thread/thread.h thread 구조체에 추가된 부분>

load\_lock은 어떠한 프로그램을 실행할 때, parent가 먼저 종료되어 child을 wait하지 못하는 문제가 발생하는데, 이를 방지하기 위해 추가한 semaphore이다. thread\* parent는 child에서 parent의 load\_lock을 sema\_up 해주기 위해, child에서 parent을 파악할 수 있는 pointer 이다. int flag는 어떠한 process가 실행될 때, 해당 process에서 load() 과정을 정상적으로 수행하지 못했을 때 1이 되는 변수이다. 이를 통해 해당 process의 비정상적인 종료 등을 파악할 수 있다. 마지막으로 fd[]는 file descriptor table 이며, file\_pointer은 current process가 filesys\_open 을 수행할 때, 해당 process가 실행하는 file에 대해 write을 방지하기 위해, 구현한 pointer 이다.

위 변수들을 초기화하기 위해 thread/thread.c 의 init\_thread()에 다음 코드를 추가했다.

#ifdef USERPROG

t->parent = running\_thread();

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

t->flag = 0;

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

t->fd[i] = NULL;

}

#endif

}

<thread/thread.c init\_thread()의 추가 부분>

각 fd[i] = NULL 하며, flag = 0, parent = running\_thread() load\_lock setting, flag setting을 수행했다.

위와 같이 구조체를 설정한 이후, 다음과 같은 header file을 userprog/syscall.c에 추가해 주었다.

#include "filesys/filesys.h"

#include "filesys/file.h"

#include "threads/synch.h"

<userprog/syscall.c에서 추가한 header file>

filesys.h은 filesys 함수들을 위해, file.h은 file 함수들을 위해, 그리고 sysnch.h은 read, write에서 lock으로 synchronization을 구현하기 위해 추가해 주었다.

또한 구현한 각 system call은 아래와 같다.

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

{

// create-null test

if(file == NULL){

exit(-1);

}

check\_address(file);

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

}

<create system call>

create system call은 해당하는 char pointer가 point하는 부분이 user 영역인지를 확인하며, user 영역인 경우, fiesys\_create() 함수를 실행시킨다.

bool remove(const char\* file)

{

if(file == NULL){

exit(-1);

}

check\_address(file);

return filesys\_remove(file);

}

<remove system call>

remove system call 또한 해당 char pointer이 user 영역을 point하는지 확인하며, filesys\_remove() 함수를 실행시킨다.

int open(const char\* file)

{

// 모든 open file에 대해, 먼저 file\_deny\_write 을 수행할까?

int return\_value;

// open-null test

if(file == NULL){

exit(-1);

}

// char\*가 user\_vaddr에 존재하는지 판단한다.

check\_address(file);

//lock\_acquire(&lock);

struct thread\* cur;

struct file\* open\_file = filesys\_open(file);

if(open\_file == NULL){

return\_value = -1;

}

else{

cur = thread\_current();

// current\_thread의 fd에서 NULL인 fd를 찾아 새로운 file\* 을 할당한다.

// 만약 I/O redirection, pipe의 구현이 필요하다면 3 -> 0 으로 변경

for(int i=3; i<128; i++){

// 비어 있는 file descripter 을 발견

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

cur->fd[i] = open\_file;

return\_value = i;

// 현재 thread와 open하는 file의 이름이 동일한 경우

//if(strcmp(file, cur->name) == 0){

// file\_deny\_write(open\_file);

//}

break;

}

}

}

//lock\_release(&lock);

return return\_value;

}

<open system call>

open system call 또한 char pointer이 user 영역인지 확인하며, filesys\_open으로 해당 char\* 이름을 가진 file에 대한 struct file pointer을 구해내며, 현재 thread(open을 실행시킨 thread)의 비어 있는 fd table 중 가장 앞에 해당 file pointer을 할당하는 형태이다. 이때 for loop가 3으로 시작하는 이유는, 0, 1, 2는 이미 STDIN, STDOUT, STDERR으로 정해져 있기 때문이다. 추가로 만약에 I/O redirection, pipe 등을 구현하는 경우, for loop의 시작 값을 0 으로 설정해 주어야 더 적합한 pintos을 구현할 수 있다.

int filesize(int fd)

{

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

exit(-1);

}

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

}

<filesize system call>

filesize system call은 해당하는 fd 값에 file pointer이 존재하는지를 파악하며, 존재하는 경우 file\_length() 함수를 호출한다.

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

{

int return\_value;

check\_address(buffer);

lock\_acquire(&syn\_lock);

// standard input이 아닌 경우

if(fd > 2){

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

lock\_release(&syn\_lock);

exit(-1);

}

//lock\_acquire(&lock);

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

//lock\_release(&lock);

//return return\_value;

}

else if(fd == 0){

//lock\_acquire(&lock);

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

if(input\_getc() == '\0'){

//lock\_release(&lock);

return\_value = i;

break;

}

}

}

else{

return\_value = -1;

}

lock\_release(&syn\_lock);

return return\_value;

}

<read system call>

read system call은 fd > 2 / fd == 0 / else 의 경우로 나누어 수행된다. fd > 2인 경우, file\_read()함수를 통해 fd에 해당하는 file pointer에서 read을 수행한다. fd = 0인 경우 프로젝트 1과 동일한 작업을 수행한다. else의 경우는 fd = 1, 2인 경우로 STDOUT, STDERR에서의 입력을 의미한다. 이러한 입력은 부적절한 입력이기 때문에, 해당 경우에는 return\_value = -1로 설정했다. 추가로 lock\_acquire() / lock\_release()를 사용한 것을 코드에서 파악할 수 있는데, 이는 read / write에서 synchronization을 보장하기 위해서 추가한 부분이다. 해당하는 lock은 syscall.c의 전역변수로, syscall\_init()에 의해 초기화 되어야 한다. 따라서 추가되는 코드는 다음과 같다.

lock\_init(&syn\_lock);

<syscall\_init()에서 추가된 부분>

이를 통해 lock을 이용할 수 있다.

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

{

int return\_value;

check\_address(buffer);

// standard output이 아닌 경우

lock\_acquire(&syn\_lock);

if(fd > 2){

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

lock\_release(&syn\_lock);

exit(-1);

}

//lock\_acquire(&lock);

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

//lock\_release(&lock);

//return return\_value;

}

else if(fd == 1){

//lock\_acquire(&lock);

putbuf(buffer, size);

return\_value = size;

//lock\_release(&lock);

//return size;

}

else{

return\_value = -1;

}

lock\_release(&syn\_lock);

return return\_value;

}

<write system call>

write system call 또한 fd > 2 / fd == 1 / else 부분으로 나누어 수행한다. fd >2 인 경우 file\_write을 통해 해당하는 file pointer에 file을 write한다. 또한 동일하게 lock\_acquire() / lock\_release() 을 사용해서 read / write가 동시에 실행되지 않도록 구현했다.

void seek(int fd, unsigned position)

{

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

exit(-1);

}

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

}

<seek system call>

seek system call은 file\_seek을 call 하여 구현할 수 있다.

unsigned tell(int fd)

{

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

exit(-1);

}

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

}

<tell system call>

tell system call 또한 file\_tell system call을 통해 간단하게 구현할 수 있다.

void close(int fd)

{

/\*

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

exit(-1);

}

\*/

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

struct file\* close\_file = cur->fd[fd];

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

for(int i=3; i<128; i++){

if(i != fd && cur->fd[i] == close\_file)

return;

}

// 만약 다른 file descriptor 에서 해당 file pointer을 point하지 않는 경우

// file\_close에서 file\_allow\_write가 실행된다.

file\_close(close\_file);

}

<close system call>

close system call은 해당 fd의 file structure을 file\_close하며, 해당 fd 값을 다시 NULL로 만들어주는 형태이다. 이때 해당하는 file structure을 linux 상의 dup() 등으로 복사하여, 다른 fd에서도 point할 수도 있다. 따라서 해당 file pointer을 point하고 있는 모든 fd을 찾아서 종료해야 한다. 이를 수행하는 코드가 위 코드 상의 for loop 이다.

추가로 exit system call을 수정해 주어야 한다. 추가된 부분은 다음과 같다.

void exit(int status)

{

// int status를 thread sturuct에 할당한다.

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

//cur->status = THREAD\_DYING;

cur->exit\_status = status;

for(int i=3; i<128; i++){

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

close(i);

}

}

// exit 하는 process의 모든 child process을 wait 한다.

struct thread\* child = NULL;

struct list\_elem\* e;

for(e = list\_begin(&(cur->child)); e != list\_end(&(cur->child)); e = list\_next(e)){

struct thread\* check = list\_entry(e, struct thread, child\_elem);

process\_wait(check->tid);

}

thread\_exit();

//printf("EXIT func executed.\n");

}

<exit system call에서 추가한 부분>

exit system call에서 두 가지의 추가적인 부분이 필요한데, 해당 process을 종료하기 이전에 fd table에서 저장하고 있던 모든 file structure을 종료해야 하며, 또한 해당 process을 종료하기 이전에 child process 각각을 wait 하며 reaping 해 주어야 한다. 이를 수행하는 코드가 위 부분이다.

또한 각 system call을 syscall handler에 연결하기 위해, userprog/syscall.c의 syscall\_handler() 함수에 다음 내용을 추가해야 한다.

case SYS\_CREATE:

check\_address(f->esp + 4);

check\_address(f->esp + 8);

f->eax = create(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8));

break;

case SYS\_REMOVE:

check\_address(f->esp + 4);

f->eax = remove(\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_OPEN:

check\_address(f->esp + 4);

f->eax = open(\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_FILESIZE:

check\_address(f->esp + 4);

f->eax = filesize(\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_SEEK:

check\_address(f->esp + 4);

check\_address(f->esp + 8);

seek(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8));

break;

case SYS\_TELL:

check\_address(f->esp + 4);

f->eax = tell(\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_CLOSE:

check\_address(f->esp + 4);

close(\*(uint32\_t \*)(f->esp + 4));

break;

<syscall\_handler()에 추가한 부분>

이를 통해 각 system call을 user program에서 실행시킬 수 있다. 추가로 read / write 는 프로젝트 1에서 구현한 코드와 동일하므로 첨부하지 않았다.

이후에는 test 결과를 바탕으로 구현한 pintos에서 취약점을 파악하고 이를 보완해주는 형식으로 코드를 구현했다. 수정한 코드는 userprog/exception.c, userprog/process.c 이며, 각각은 아래와 같다.

// sc-bad-sp 을 방지하기 위해

// bad-read, bad-write 해결

if(is\_kernel\_vaddr(fault\_addr) || not\_present){

exit(-1);

}

<userprog/exception.c의 page\_fault에 추가한 부분>

page\_fault가 발생하면 해당 page\_fault 함수가 실행하게 된다. 하지만 만약 커널 영역이 커널 영역으로의 접근을 수행하는 경우, 이는 정상적인 접근임에도 kernel panic이 발생한다. 따라서 이를 방지하기 위해 위 코드와 같이 fault addr이 커널 영역인 경우, 혹은 not\_present로 해당 영역이 실제로 존재하지 않은 페이지인 경우, kill로 종료할 필요 없이, exit(-1)으로 kernel panic을 방지할 수 있다.

또한 userprog/process.c에서 여러 기능을 추가해 주어야 한다. 첫 번째로, 실행하고 있는 file에 대해서는 write을 수행할 수 없도록 해 주어야 한다. 이를 위해 load() 함수에서 filesys\_open을 통해 얻어낸 file structure에 대해 다음 과정을 수행해야 한다. 이를 수행하는 코드는 아래와 같다.

t->file\_pointer = file;

file\_deny\_write(file);

<userprog/process.c 의 load 함수에서 추가한 부분>

이를 통해 실행하고 있는 file에 대해서는 write을 방지할 수 있다. 또한 이 과정에서 open한 file은 process\_exit() 함수에서 종료해 주어야 하며, 이 코드는 다음과 같다.

file\_close(cur->file\_pointer);

<userprog/process.c의 process\_exit 함수에서 추가한 부분>

이를 통해 rox 관련 test을 모두 통과할 수 있다.

다음으로는 multi-oom test를 수행하며 여러 부분에 코드를 추가해 주었다. 첫 번째로, process\_execute() 가 실행되면, thread\_create()가 실행되며 parent, child 2개의 process가 존재하게 된다. 이때 parent의 execute가 child가 load하기 이전에 종료되면, 오류가 발생하게 된다. 따라서 이를 방지하기 위해 추가적인 semaphore을 사용했고, 이는 위 thread 구조체에서 추가한 load\_lock이다. 이를 이용한 코드는 아래와 같다.

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

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

<userprog/process.c의 start\_process 함수에서 추가한 부분>

// thread\_create 이후 child thread가 load 과정을 모두 수행할 때까지 기다린다.

sema\_down(&(thread\_current()->load\_lock));

<userprog/process.c의 process\_exit 함수에서 추가한 부분>

위와 같은 코드를 통해 parent가 load 수행 이전에 종료되는 문제를 방지할 수 있다.

또한 위와 같이 비정상적으로 종료된(load 이전에 종료되는) child process는 wait으로 reaping되지 않고, parent의 child process list에 남아있을 수 있다. 따라서 이를 방지하기 위해 아래 코드를 process\_execute()에 추가해 주었다.

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

struct thread\* child = NULL;

struct list\_elem\* e;

for(e = list\_begin(&(cur->child)); e != list\_end(&(cur->child)); e = list\_next(e)){

struct thread\* check = list\_entry(e, struct thread, child\_elem);

if(check->flag == 1 || check->exit\_status == -1){

//return process\_wait(tid);

return process\_wait(tid);

}

}

<userprog/process.c의 process\_execute 함수에 추가한 부분>

이를 통해, 모든 child list에 존재하는 child 중, 비정상적으로 종료된 child가 존재하는 경우, 해당 child를 reaping하는 형식으로 multi-oom test를 통과하며, pintos가 정상적으로 작동함을 보장할 수 있었다.

또한 syn-read / syn-write test에서 문제가 존재했다. system call의 구현에서 설명한 것처럼 각 read / write에서 lock\_acquire / lock\_release을 통해 하나의 process만 해당 작업을 수행할 수 있도록 구현하였는데, 계속해서 오류가 발생했다. 따라서 새로운 process가 실행되는 경로를 따라 찾아보니 process\_execute 함수에서 filesys\_open으로 해당 parameter file\_name에 해당하는 file을 open할 때 문제가 발생함을 파악할 수 있었다. 따라서 filesys\_open 코드를 제거해주니, 모든 test을 통과할 수 있었다. 원인을 정확하게 파악할 수는 없었으나, file\_name이 kernel 영역 등에 존재해서 발생할 수 있는 접근 문제, 혹은 해당 file이 이미 실행되고 있기 때문에 file\_deny\_write을 적용한 file이기 때문에 read 0~9 에 대해서 “read 1”의 filesys\_open(“child-syn-read”)이 정상적으로 작동할 수 없어서 문제가 발생할 수도 있음을 파악하기도 했다.

마지막으로, multi-oom test는 통과했으나, 이전 프로젝트 1에서 esp stack에 data와 pointer을 저장할 때 사용한, malloc으로 heap 영역을 잡아놓고 free 해주지 않은 부분을 찾을 수 있었다. 이를 free 해주어 더 견고한 프로그램을 만들었다.. 이는 아래와 같다.

// 이전 data들에 대해 free 시킨다.

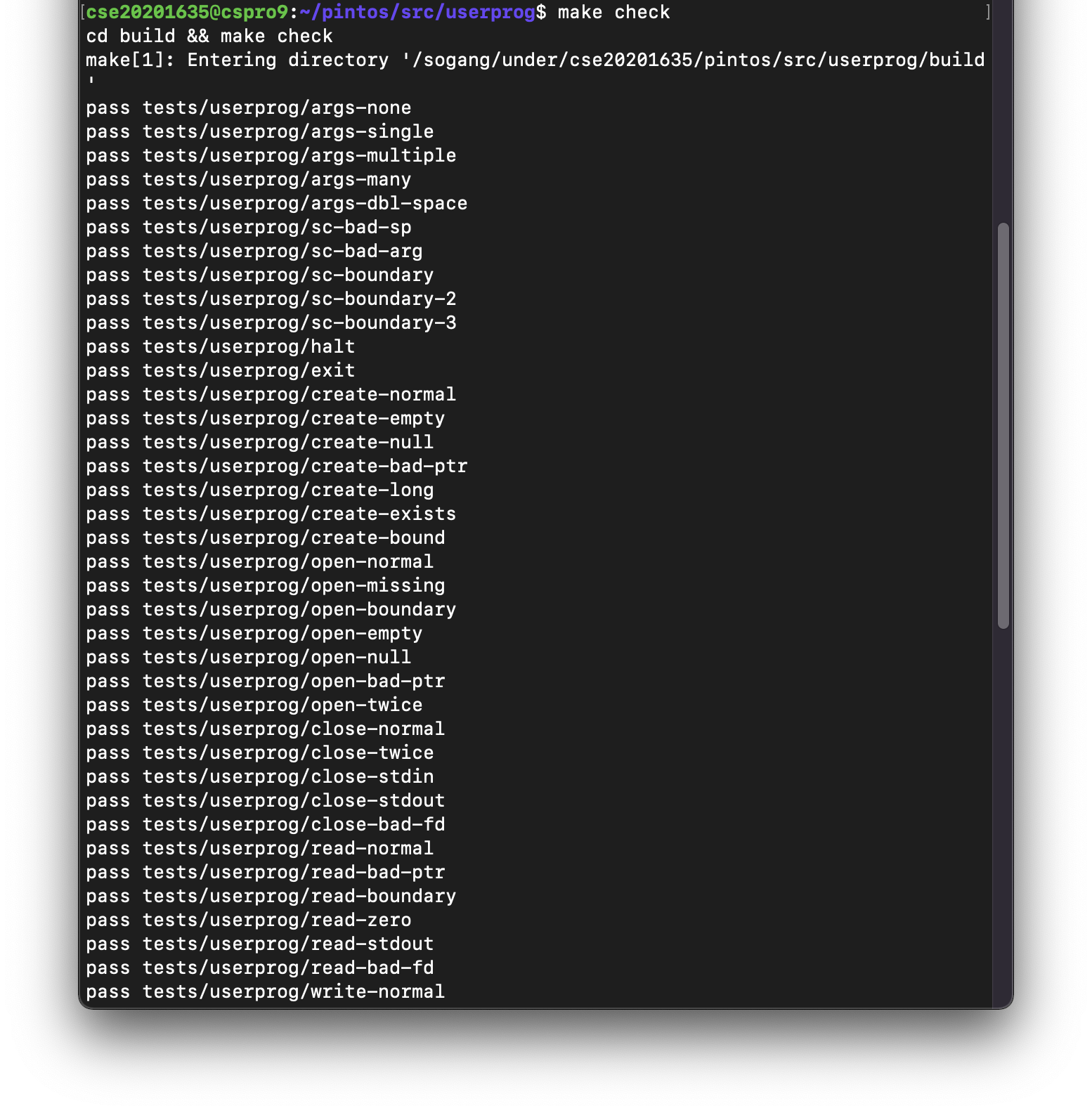
free(argv);

free(file\_name\_save);

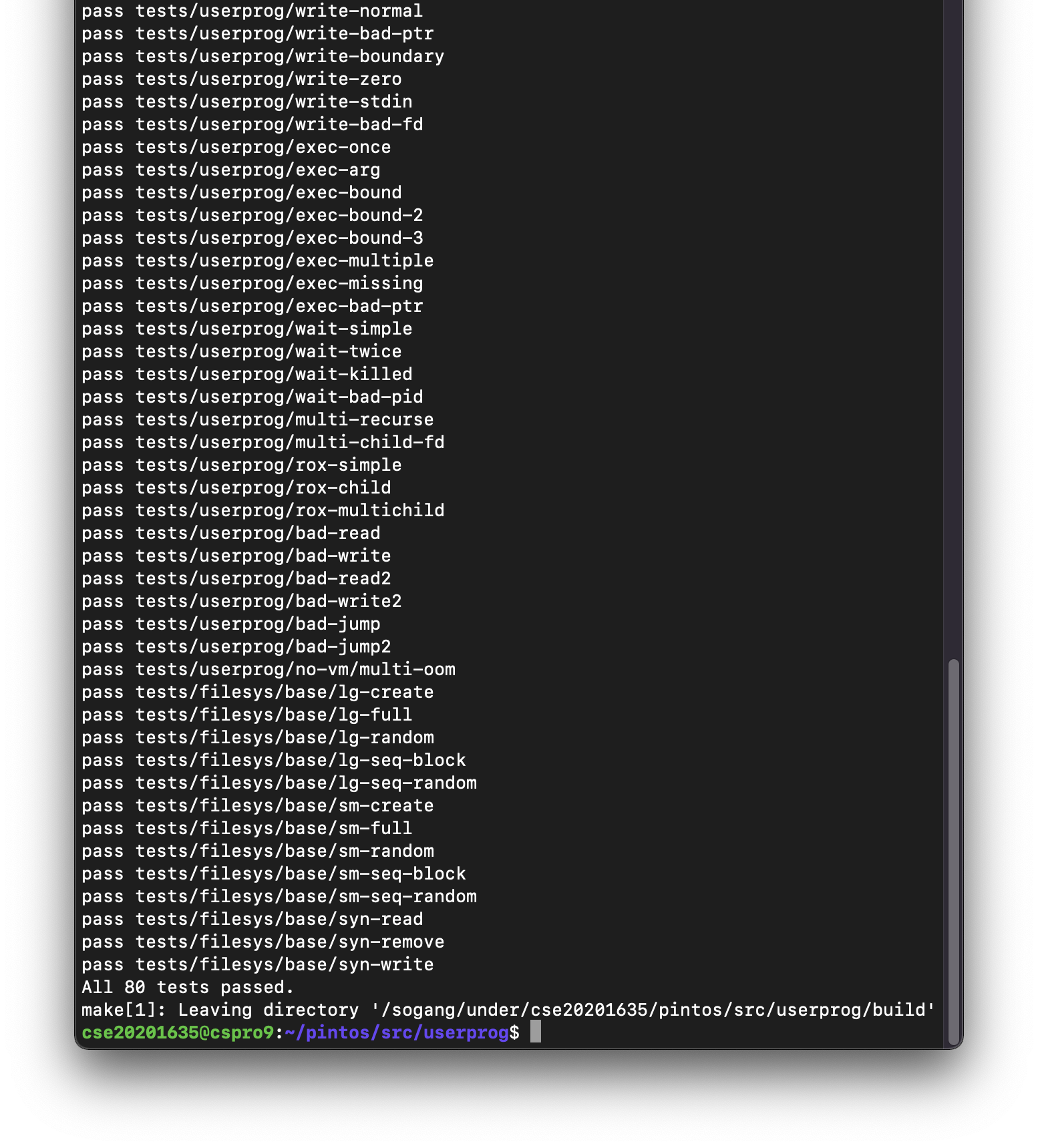
<userprog/process.c의 load 함수에 추가한 부분>

위와 같은 과정들을 통해, 모든 테스트를 통과하는 pintos을 구현할 수 있었다..

* 1. **시험 및 평가 내용**



<make check 실행 결과1>



<make check 실행 결과2>

위 결과를 통해 80개의 모든 test을 통과하는 pintos을 구현함을 파악할 수 있다.