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

담당 교수 / 분반 : 김영재

이름 / 학번 : 박태윤 / 20171646

개발 기간 : 10/01 ~ 10/12

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

project1에서 make check fail이 뜬 80개 중 나머지 59개의 test가 통과할 수 있도록 구현한다. 이를 위해선 project1에서 구현되지 않은 open, seek등의 시스템 콜을 구현해야 하며, read, write에서 동기화 문제를 해결해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

2. (이번 프로젝트에서 구현해야 하는) System Calls

3. Synchronization in Filesystem

1. pintos, 리눅스 체제에서는 file system이 존재하는데 이 파일들에 접근하기 위해서 file descriptor가 존재한다. 즉 시스템 콜을 이용하여 어떠한 file을 열거나 그 file에 쓰거나 할 때 file descriptor를 통해 접근하여 동작을 수행시킬 수 있다. 각 프로세스(쓰레드)들이 독립적인 file descriptor table을 가지게 함으로써 open, write등 시스템 콜이 동작하는데 이를 사용할 수 있을 것이다. 예를 들어 어떤 파일 디스크립터 값이 NULL인 경우를 열려고 하는 경우 이에 대한 처리를 할 수 있을 것이며 같은 파일을 읽거나 쓰려고 하는 경우에서 나타나는 동기화 문제를 해결할 수 있을 것이다.

2. project1에서 구현되지 않은 SYS\_CREATE, SYS\_REMOVE, SYS\_OPEN, SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL, SYS\_CLOSE와 STDIN, STDOUT에서만 동작하는 SYS\_READ, SYS\_WRITE에 대한 시스템 콜 함수를 구현해야 한다. load에서 memory로 passing한 command를 가지고 esp를 통해 접근하여 시스템 콜 함수를 동작시키는 방식은 project1에서와 동일하며, 구현이 필요한 시스템 콜 함수들을 적절하게 구현하면 project1에서 pass하지 못한 rox-simple, multi-oom등을 통과할 수 있을 것이다.

3. SYS\_READ, SYS\_WRITE등에 대한 시스템 콜 함수를 모두 구현한 뒤 make check를 돌려보면 테스트 중 sys\_write, sys\_read가 돌아가지 않는 경우가 생긴다. 이는 read와 write에 대한 동기화 문제를 해결하지 못했기 때문인데, 예를 들어 어떠한 특정 파일을 read하고 있는 경우 동시에 write를 해주면 write한 내용이 read하고 있는 프로세스에서 반영이 되지 않는 등의 동기화 적인 문제가 발생할 수 있다. 이를 해결하기 위해서 semaphore, lock을 사용하여야 하며 해당 api의 적절한 사용을 통해 동기화 문제를 해결한다면 sys-write, sys\_read테스트를 통과할 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

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)

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

1. src/thread의 thread.h에 있는 thread구조체에 struct file\* file\_des[128]이라는 구조체 배열을 추가하였다. file구조체는 src/filesys의 file.c에 선언이 되어 있었고, 이 구조체를 다른 파일에서도 import하여 사용할 수 있도록 file.h로 옮겼다. 쓰레드가 생성이 되면 해당 쓰레드에서 des[128]은 모두 NULL로 초기화가 된다. 이 배열을 통해 파일들에 접근을 할 수 있는데, 예를 들어 open시스템 콜에서 어떤 특정 파일을 오픈하고자 한다면 filesys\_open api를 이용해 파일을 열고 그 결과를 file descriptor table에 저장함으로써 다른 동작에서도 해당 쓰레드에서 open을 한 파일에 접근할 수 있도록 구현하였다.

2.

- SYS\_CREATE : 파일을 create하는 시스템 콜이다. 시스템 콜 함수에서 파일의 이름과 생성하고자 하는 파일의 크기를 인자로 받아 filesys\_create api를 통해 파일을 생성하며 생성에 성공하면 true, 실패하면 false를 반환한다.

- SYS\_REMOVE : 특정 파일을 remove하는 시스템 콜이다. 삭제하고자 하는 파일의 이름을 인자로 받아 filesys\_remove api를 이용하여 파일을 삭제하며 삭제에 성공하면 true, 실패하면 false를 반환한다.

- SYS\_OPEN : 파일을 open하는 시스템 콜이다. 열고자 하는 파일의 이름을 시스템 콜 함수에서 인자로 받아 filesys\_open api를 통해 파일을 열고 파일을 open 하는 것이 성고앟면 해당 file descriptor를 반환하고 실패하면 -1를 반환한다.

- SYS\_FILESIZE : 해당 파일의 크기를 알려주는 시스템 콜이다. 특정 파일 디스크립터를 인자로 시스템 콜 함수에서 인자로 받아 현재 쓰레드에서 해당 디스크립터에서 가리키는 파일의 크기를 file\_length api를 이용하여 반환한다.

- SYS\_WRITE : open이 된 파일에 내용을 write하는 시스템 콜이다. 시스템 콜 함수에서 fd, buffer, n(size)를 인자로 받아 buffer에 해당하는 내용을 fd가 1이면 putbuf를 통해 화면에 출력하고 2보다 크면 현재 쓰레드의 fd에 해당하는 파일에 내용을 file\_write한다.

- SYS\_READ : open이 된 파일의 내용을 read하는 시스템 콜이다. 시스템 콜 함수에서 fd, buffer, n을 인자로 받아 fd가 0이면 키보드의 입력을 받고 2보다 크면 file\_read api를 이용하여 현재 쓰레드의 fd에 해당하는 파일의 내용을 buffer에 저장한다.

-SYS\_SEEK : open이 된 파일의 위치를 변경하는 시스템 콜이다. 시스템 콜 함수에서 fd,와 position을 인자로 받아 현재 쓰레드의 fd에 해당하는 파일의 위치를 position만큼 file\_seek api를 이용하여 이동시킨다.

- SYS\_TELL : open한 파일의 위치를 알려주는 시스템 콜이다. 시스템 콜 함수에서 fd를 인자로 받아 현재 쓰레드의 fd에 해당하는 파일의 위치를 file\_tell api를 이용하여 반환한다.

- SYS\_CLOSE : 열린 파일을 close하는 시스템 콜이다. 시스템 콜 함수에서 fd를 인자로 받아 현재 쓰레드의 fd에 해당하는 파일을 file\_close api를 이용하여 close한다. close를 하면서 파일 디스크립터 테이블에서도 제거를 해주어야 한다.

3.

같은 파일에 동시에 접근하는 부분에서 문제가 발생할 수 있기 때문에 이를 해결을 해줘야 한다. 이를 위해 src/userprog의 syscall.c에서 semaphore를 사용하였다. SYS\_READ, SYS\_WRITE, SYS\_OPEN 시스템 콜에서 semaphore를 적용시켰는데, syscall\_handler에서 받은 시스템 콜 번호가 SYS\_READ, SYS\_WRITE, SYS\_OPEN에 해당한다면 시스템 콜 함수를 호출하기 전에 sema\_down을 시켜주었다. 세마포어 구조체는 wro라는 이름으로 전역변수로 선언하였으며 syscall\_init에서 1로 sema\_init 시켜주었다. 만일, 각 시스템 콜 함수 안에서 kernel메모리 영역에 접근하는 등의 오류를 범해서 exit를 시켜주어야 하는 경우 그 전에 sema\_up으로 lock을 해제를 시켜주었고, 그렇지 않고 시스템 콜 함수 동작이 완전히 끝난 경우 시스템 콜 함수를 호출한 부분 그 다음에서 sema\_up으로 lock을 해제하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

**10/1 ~ 10/3 : 시스템 콜 함수 구현**

**10/9 ~ 10/10 : 동기화 문제 처리**

**10/11 ~ 10/12 : multi – oom 테스트 처리**

* 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.h의 thread구조체에 file형태의 배열 file\_des[128]배열을 추가하여 각 프로세스에서 고유한 파일 디스크립터 테이블을 가질 수 있도록 하였다. 이를 통해 open한 파일을 저장하고 해당 테이블을 이용해 read하거나 write할 수 있도록 이를 추가하였다. 또한 이번 project2에서 구현해야하는 시스템 콜에 따른 시스템 콜 함수들을 추가하였다. 이는 다음과 같다.

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

bool remove (const char \*file);

int open (const char \*file);

int filesize (int fd);

int read (int fd, void \*buffer, unsigned size); (STDIN이외의 동작 추가 구현)

int write (int fd, const void \*buffer, unsigned size); (STDOUT이외의 동작 추가 구현)

void seek (int fd, unsigned position);

unsigned tell (int fd);

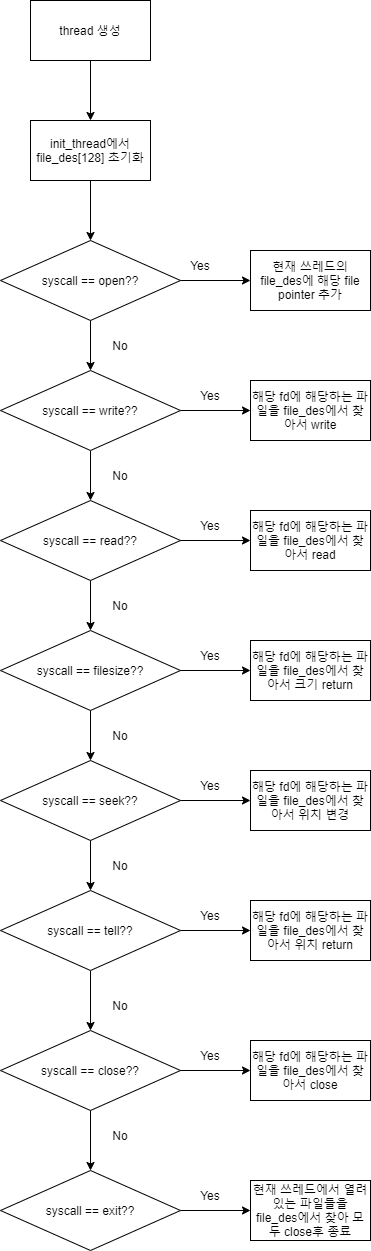
void close (int fd);

동기화 문제를 해결하기 위해 lock을 걸어주기 위한 semaphore인 struct semaphore wro;를 syscall.c에 선언하였다. syscall\_init에서 sema\_init(&wro, 1)로 초기화를 해주었으며, SYS\_READ, SYS\_WRITE, SYS\_OPEN에 해당하는 동작에서 sema\_down(&wro), sema\_up(&wro)를 활용하여 같은 파일에 동시에 접근하려 하는 동기화 문제를 해결하였다.

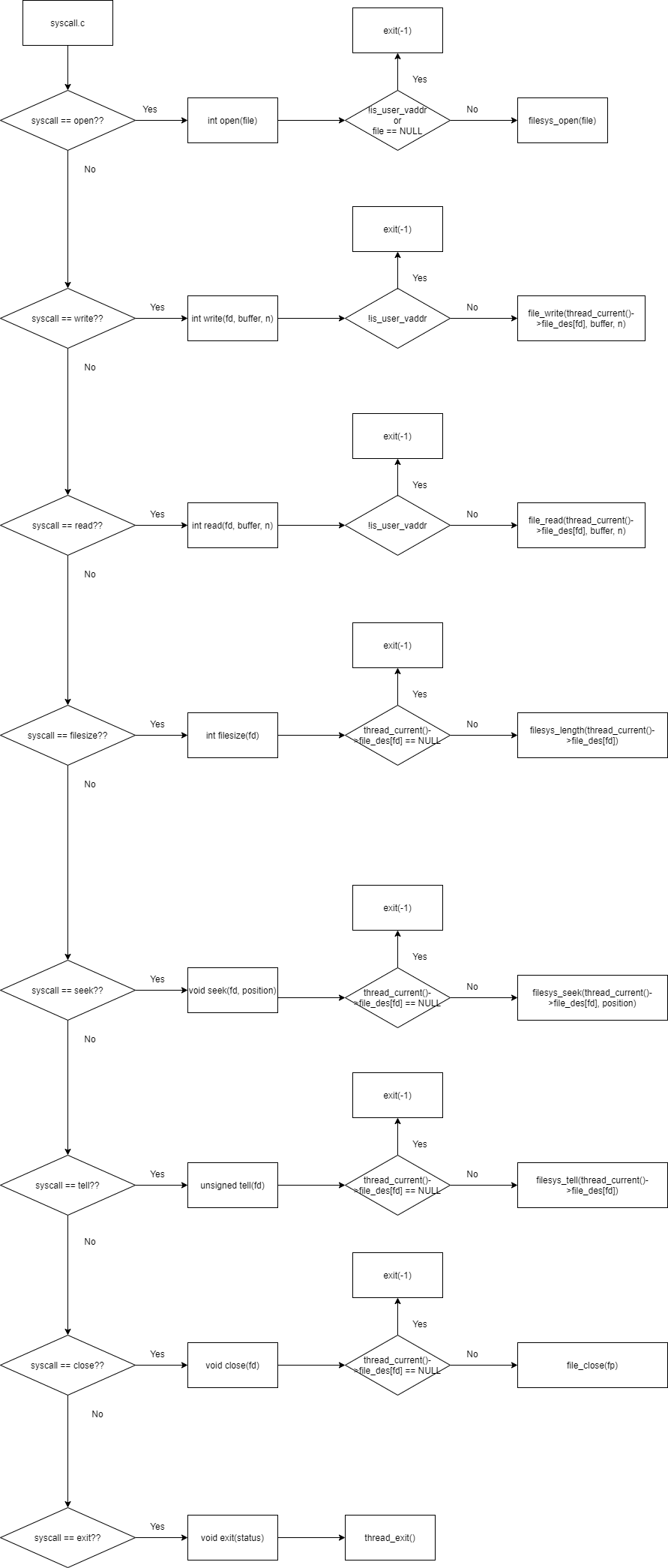
1. **연구 결과**
   1. **Flow Chart**

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

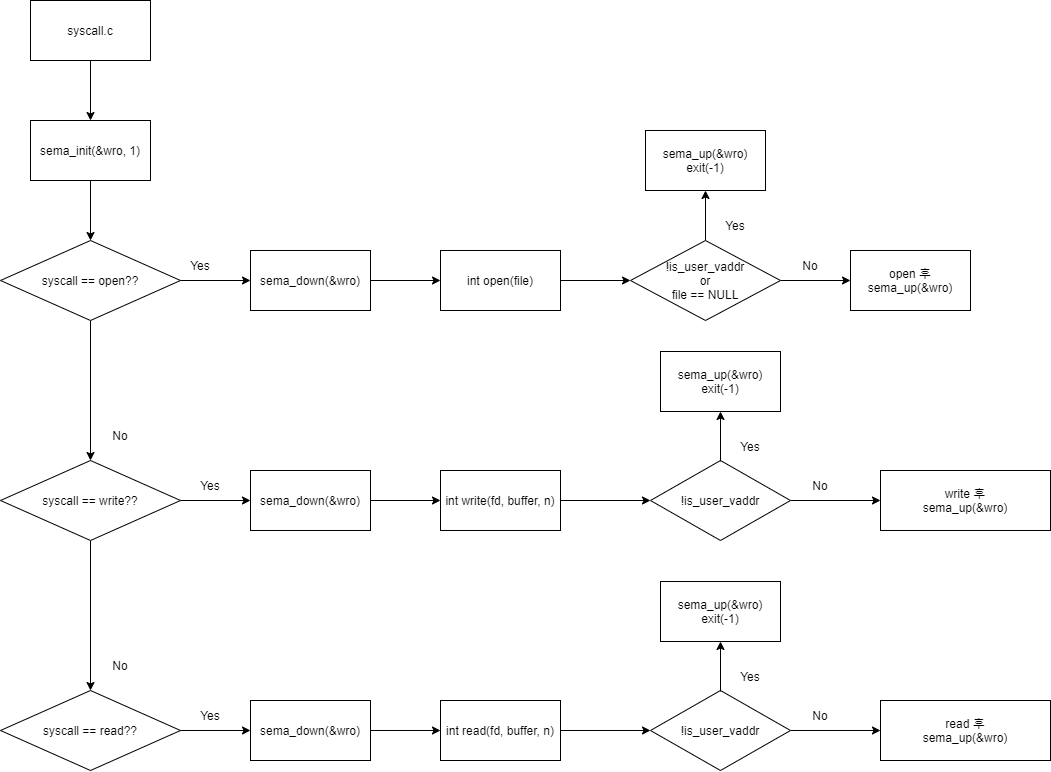
**(1)**

****

**(2)**

****

**(3)**

****

* 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
* struct thread
* {
* /\* Owned by thread.c. \*/
* tid\_t tid; /\* Thread identifier. \*/
* enum thread\_status status; /\* Thread state. \*/
* char name[16]; /\* Name (for debugging purposes). \*/
* uint8\_t \*stack; /\* Saved stack pointer. \*/
* int priority; /\* Priority. \*/
* struct list\_elem allelem; /\* List element for all threads list. \*/
* **struct file\* file\_des[128];** //new
* /\* Shared between thread.c and synch.c. \*/
* struct list\_elem elem; /\* List element. \*/
* #ifdef USERPROG
* /\* Owned by userprog/process.c. \*/
* bool flag;
* uint32\_t \*pagedir;
* struct list child;
* struct list\_elem child\_elem;
* int exit\_status;
* struct thread\* parent;
* #endif
* /\* Owned by thread.c. \*/
* unsigned magic; /\* Detects stack overflow. \*/
* };

thread.c의 thread구조체에 다음과 같이 파일 디스크립터 테이블인 file\_des[128]을 선언하였다. 이는

static void

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

{

enum intr\_level old\_level;

ASSERT (t != NULL);

ASSERT (PRI\_MIN <= priority && priority <= PRI\_MAX);

ASSERT (name != NULL);

//new

for(int i = 0; i < 200; i++)

t->file\_des[i] = NULL;

와 같이 init\_thread에서 NULL로 초기화를 해주었다.

//project 2

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

if(file==NULL || !is\_user\_vaddr(file))

exit(-1);

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

}

bool remove(const char \*file){

if(file == NULL || !is\_user\_vaddr(file))

exit(-1);

return filesys\_remove(file);

}

int open(const char \*file){

int i;

if(!is\_user\_vaddr(file) || file == NULL){

sema\_up(&wro);

exit(-1);

}

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

if(fp == NULL)

return -1;

else{

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

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

if(strcmp(thread\_current()->name, file) == 0)

file\_deny\_write(fp);

else

file\_allow\_write(fp);

thread\_current()->file\_des[i] = fp;

return i;

}

}

}

return -1;

}

int filesize(int fd){

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

exit(-1);

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

}

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

int res = -1;

if(!is\_user\_vaddr(buffer)){

sema\_up(&wro);

exit(-1);

}

if(fd == 1){

putbuf(buffer, n);

return n;

}

else if(fd > 2){

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

sema\_up(&wro);

exit(-1);

}

if(thread\_current()->file\_des[fd]->deny\_write)

file\_deny\_write(thread\_current()->file\_des[fd]);

else

file\_allow\_write(thread\_current()->file\_des[fd]);

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

}

else

return -1;

}

int read(int fd, void \*buffer, unsigned n){

int i;

int ret=-1;

if(!is\_user\_vaddr(buffer)){

sema\_up(&wro);

exit(-1);

}

if(fd == 0){

for(i = 0; i < n; i++){

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

break;

}

return i;

}

else if(fd > 2){

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

sema\_up(&wro);

exit(-1);

}

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

}

else

return -1;

}

void seek(int fd, unsigned position){

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

exit(-1);

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

}

unsigned tell(int fd){

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

exit(-1);

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

}

void close(int fd){

struct file\* fp;

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

exit(-1);

fp = thread\_current()->file\_des[fd];

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

return file\_close(fp);

}

**project2에서 추가적으로 구현해야 하는 시스템 콜 함수들은 syscall.c에서 다음과 같이 구현하였다. pintos에 내장되어 있는 api를 몇 가지 사용하였는데, create함수에서는 해당 file이라는 이름으로 initial\_size로 파일을 생성하는 filesys\_create(file, initial\_size), remove함수에서는 해당 파일을 삭제하는 filesys\_remov(file)함수를, open함수에서는 해당하는 파일을 여는 filesys\_open(file)함수를, filesize함수에서는 현재 쓰레드의 디스크립터 테이블에서 해당하는 디스크립터가 가리키는 파일의 길이를 반환해주는 file\_length(thread\_current()->file\_des[fd])를, write함수에서는 현재 쓰레드에서 해당 fd가 가리키는 파일에 buffer의 내용을 쓰는 file\_write(thread\_current()->file\_des[fd], buffer, n)함수를, read함수에서는 현재 쓰레드에서 fd가 가리키는 파일의 내용을 buffer에다 쓰는 file\_read(thread\_current()->file\_des[fd], buffer, n)을, seek 함수에서는 현재 쓰레드가 fd로 가리키는 파일을 position만큼 위치를 옮기는 file\_seek(thread\_current()->file\_des[fd], position)함수를, tell함수에서는 현재 쓰레드의 fd가 가리키는 파일의 위치를 알려주는 file\_tell(thread\_current()->file\_des[fd])를, close함수에서는 해당하는 파일을 닫는 file\_close(fp)함수를 사용하였다. exit에서는 현재 쓰레드에 대해 file descriptor table에서 나타내는 파일 중 열려있는 파일이 있다면 모두 close를 해주었다.**

**또한 동기화 문제를 해결하기 위해** struct semaphore wro; **라고 세마포어를 하나 선언하였다. 이를 syscall\_init에서 sema\_init(&wro, 1)을 통해 초기화를 시켜주었으며 SYS\_READ, SYS\_WRITE, SYS\_OPEN에 해당하는 시스템 콜 함수를 호출하는 부분과 시스템 콜 함수 안에서 sema\_down(&wro), sema\_(&up)을 이용해 lock을 걸어주고 풀어주는 방식으로 구현하였다.**

**구현을 하는 동안 자식 쓰레드가 로드가 되기도 전에 부모 쓰레드가 죽거나 로드가 실패했을 시 자식 쓰레드를 처리해야 하는 문제를 발견하였다. 이는 process.c에서 해결하였다. 우선 쓰레드의 load상태를 저장하기 위해 project1에서 따로 선언한 thread\_list구조체에**

typedef struct Thread\_list{

tid\_t tid;

tid\_t parent\_tid;

int flag;

int exitStatus;

int load\_flag;

}thread\_list;

**와 같이 load\_flag를 선언하였다. 쓰레드가 생성이 되면 이 load\_flag는 1로 초기화가 된다. 이렇게 초기화가 된 load\_flag는 process\_execute에서**

int cur\_idx = search\_wait\_list(thread\_current()->tid);

wait\_list[cur\_idx].load\_flag = 0;

**와 같이 0으로 세팅이 된다. 이 다음에**

while(1){

if(wait\_list[cur\_idx].load\_flag != 0)

break;

thread\_yield();

}

**로 load\_flag가 0에서 다른 값으로 바뀔 때 까지 계속 yield를 사용해 동작 진행을 멈춘다. load\_flag가 바뀌는 순간은 총 2가지 경우가 있는데 하나는 자식 프로세스가 로드가 완료가 될 시 부모의 load\_flag를 1로 바꾸는 경우이다.**

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

/\* If load failed, quit. \*/

palloc\_free\_page (file\_name);

int p\_idx = search\_wait\_list(wait\_list[cur\_idx].parent\_tid);

wait\_list[p\_idx].load\_flag = 1;

**start\_process에서 다음과 같이 load가 끝나면 부모 쓰레드의 load\_flag를 1로 바꾸는 것을 알 수 있다. 두 번째 경우는 load가 실패했을 시, 즉 load의 리턴값을 받는 success변수가 false를 나타낼 시 현재 쓰레드의 load\_flag를 -1로 바꾸는 경우이다.**

if (!success){

wait\_list[cur\_idx].load\_flag = -1;

exit(-1);

}

**이와 같이 표현하였으며 load\_flag가 -1로 변경이 된 쓰레드는 로드가 정상적으로 진행이 되지 않아 종료시켜줘야 하는 쓰레드 이므로 해당 쓰레드에서 열려있는 모든 파일들을 exit(-1)을 동작시키면서 닫아주었다. 이러한 방식으로 load쪽에서 thread\_yield를 이용해 lock을 거는 방식으로 자식 쓰레드가 로드가 완료되기 전에 부모 쓰레드가 종료가 되는 문제를 해결하였다. 추가적으로 load가 비정상적으로 종료되어 load\_flag값이 -1을 나타내는 쓰레드에 대해서는 부모쓰레드에서 process\_excute에서**

for(int i = 0; i < 60; i++){

if(wait\_list[i].load\_flag == -1 && wait\_list[i].parent\_tid == thread\_current()->tid){

return process\_wait(tid);

}

}

**와 같이 회수를 해주었다.**

**실행 중인 파일에 write작업을 수행할 수 있는 문제를 해결하기 위해 file\_deny\_write(), file\_allow\_write()를 사용하였다. 시스템 콜 open함수에서 open한 파일을 수정하려고 하면 오류가 발생할 수 있기 때문에 thread\_current()->name이 인자로 받은 file과 같다면 이를 호출하였으며 그렇지 않다면 file\_allow\_write를 호출하여 파일에 write를 하는 것을 허락하였다. write에서도 현재 쓰레드의 보고 있는 파일 구조체에서 file\_deny\_write가 불렸다면 이를 처리를 해주었는데, 파일 구조체 안에 있는 bool형 변수 deny\_write를 통해 file\_deny\_write 호출 여부를 알 수 있었다.**

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

|  |  |
| --- | --- |
| 텍스트이(가) 표시된 사진  자동 생성된 설명 | 텍스트이(가) 표시된 사진  자동 생성된 설명 |

텍스트이(가) 표시된 사진

자동 생성된 설명