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

담당 교수 / 분반 : 김영재 교수님

이름 / 학번 : 최호진 / 20201654

개발 기간 : 2024. 10. 25. ~ 2024. 10 31.

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술
* 프로젝트 1에서 개발하였던 것에 더해서 유저 프로그램을 위해 필요한 시스템 콜을 추가적으로 개발한다. 특히, 파일과 관련해 필요한 파일 디스크립터와 시스템콜 (open, close, read, write, seek 등)을 추가하고, 파일 시스템의 Synchronization을 구현해 올바르게 작동할 수 있도록 구현하는 것을 목표로 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**

프로젝트 1번에서 구현한 내용으로는 파일과 관련한 작업을 진행할 수 없는 상태였다. 특히 프로세스에서 파일을 관리하기 위해 존제하는 파일 디스크립터가 구현이 되지 않은 상태였는데, file descriptor table 구조체를 선언해주고, 각 프로세스에서 파일을 다룰 수 있도록 개발한다.

* + 1. **(이번 프로젝트에서 구현해야 하는) System Calls**

앞서 설명했듯이, 기존에는 입출력을 STDIN, STDOUT을 이용해서만 가능하고, 파일을 이용한 입출력이 불가능했는데 이를 가능하게 하도록, 1번에서 구현한 파일 디스크립터를 활용하여 Read, Write System Call을 Update하고 그 외에 프로젝트 1에서는 구현하지 않았던 시스템 콜(create, remove, open, close, filesize, read, write, seek, tell)을 구현해 파일 시스템을 이용할 수 있도록 한다.

* + 1. **Synchronization in Filesystem**

멀티 프로세스(쓰레드) 환경에서 동일한 파일에 여러 사용자가 접근하여 Read, Write를 하려고 할 때 적절한 Synchronization을 통해 기대치 않은 결과가 발생하지 않도록 해야한다. 이를 위해 적절한 곳에 Lock을 구현한다.

* 1. **개발 내용**
     1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

Thread 구조체에 file 타입의 배열을 선언하여 하나의 Thread에 최대 128개의 File Descriptor를 가질 수 있도록 한다. (Pintos Manual 참고) 배열을 이용해 구현함으로써, int 타입의 파일 디스크립터를 index로 활용해 원하는 파일에 접근하고 사용, 또한 반복문을 이용해 원활하게 파일 디스크립터들을 활용할 수 있으므로 배열을 이용한다.

* + 1. System Calls: 구현할 각 system call에 대해 간략히 서술 (하나의 system call 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
    2. CREATE

파일의 이름을 인자로 받아 이에 해당하는 파일을 새로 만든다. 파일 생성 성공 또는 실패 여부를 Boolean 값으로 반환한다.

* + 1. REMOVE

파일의 이름을 인자로 받아 이에 해당하는 파일을 삭제한다. 파일 삭제 성공 또는 실패 여부를 Boolean 값으로 반환한다.

* + 1. OPEN

파일의 이름을 인자로 받아 이에 해당하는 파일을 open한다. 같은 이름의 파일이 두 번 열리지는 않는지 확인하여 예외처리 하고, 파일을 여는데 성공했다면 해당 파일의 파일 디스크립터 값, 실패하면 -1을 반환한다.

* + 1. FILESIZE

파일 디스크립터를 인자로 전달 받고, 해당 파일의 크기를 반환한다. 유효한 파일 디스크립터인지 확인하고 예외처리 한다.

* + 1. READ

파일 디스크립터와 버퍼, 크기를 인자로 전달 받고 파일에서 입력을 받아 버퍼에 저장한다. 읽은 byte의 크기를 반환하고 유효하지 않은 파일 디스크립터일 경우 -1을 반환한다.

* + 1. WRITE

파일 디스크립터와 버퍼, 크기를 인자로 전달 받고 입력받은 크기 만큼 버퍼의 내용을 파일에 작성한다. 작성한 byte의 크기를 반환하고 유효하지 않은 파일 디스크립터일 경우 -1을 반환한다.

* + 1. SEEK

파일 디스크립터와 pos를 인자로 전달 받아 pos만큼 현재 파일 포인터 위치를 변경한다.

* + 1. TELL

인자로 전달 받은 파일 디스크립터의 현재 파일 포인터 위치를 반환한다.

유효하지 않은 파일 디스크립터일 경우 -1을 리턴한다.

* + 1. CLOSE

파일 디스크립터를 인자로 전달 받고 해당하는 값에 해당하는 열린 file을 찾아 닫는다. 유효한 파일 디스크립터인지 확인하고 예외처리 한다.

* + 1. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명 (다른 방법을 서술해도 되지만 lock과 semaphore는 반드시 포함해야 함)

앞서 개발 범위에서 설명했듯이, 멀티 프로세스(쓰레드) 환경에서 동일한 파일에 동시에 여러 프로세스가 접근하여 Read, Write를 할 때 원치 않은 결과가 발생하는 것을 막기 위해 Lock을 도입한다. 해당 프로젝트에서는 System call initialization 단계에서 lock을 initialization (lock\_init())해주고, File에 접근하는 System call (Open, Read, Write)에서 파일에 접근하기 전에 lock을 걸어주고 (lock\_acquire()), 사용이 끝나면 lock을 풀어준다(lock\_release()). 이러한 방식으로 구현하여 synchronization한다.

추가로, 프로세스 간의 synchronization을 위해 프로젝트 1에서는 두 semaphore를 이용해 wait, exit 과정에서 동기화 했는데, 이번 프로젝트에서는 추가로 하나의 semaphore를 도입하여 start\_process와, process\_execute 시 자식 프로세스가 로드 되는 걸 기다린 후 작업이 이어질 수 있도록 구현한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
  + 10/25 ~ 10/26: File Descriptor 구현 및 관련 System Call 구현
  + 10/27 ~ 10/29: File System의 Synchronization 구현
  + 10/30 ~ 10/31: 보고서 작성
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 1. File Descriptor

threads/threads.h 파일에 선언된 thread 구조체에 struct file\* 타입의 배열인 struct file\* fd\_table[128]을 선언한다. 이렇게 함으로써 쓰레드 당 128개의 file descriptor를 가질 수 있고, 순서대로 파일 디스크립터를 관리하여 더욱 효율적이다. 또한, File Descriptor table은 init\_thread 함수에서 Null로 초기화하고 각 System call에서 필요에 맞게 이용한다.

* + 1. System Calls

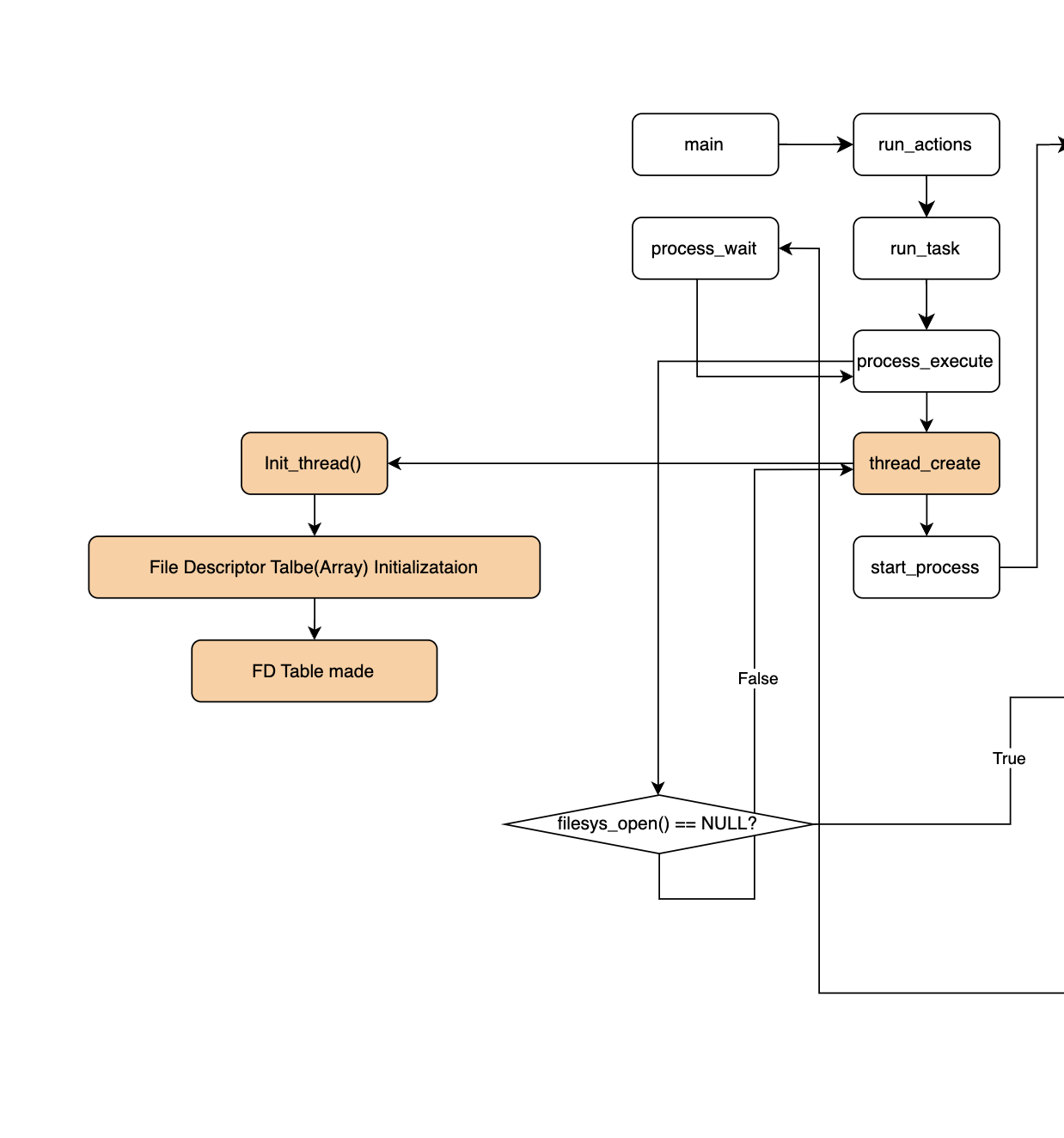
userprog/syscall.c에서 create, remove, open, filesize, read, write, seek, tell, close system call을 다루기 위한 함수들을 선언하고 구현하였다. 시스템 콜에 대한 자세한 설명은 앞선 II.B, 구현된 코드는 IV.B에서 설명한다. 또한, userprog/exception.c에서 page fault 함수 내에 Exit 조건에 not\_present도 추가하였다.

* + 1. Synchronization in File System

앞서 설명했듯이 여러 프로세스에서 하나의 파일에 Read, Write를 할 경우에 원치 않은 결과를 방지하기 위해 userprog/syscall.c에 lock 구조체 타입의 filesys\_lock을 추가한다. 이 lock을 이용해 open, read, write 함수에서 파일을 직접 사용하기 전에 lock\_acquire()로 lock을 걸어주고, 함수가 끝날 때 lock\_release()로 lock을 풀어준다.

추가적으로, thread 구조체에 semaphore type의 load\_lock을 추가하여, 부모 – 자식 관계에서 부모 쓰레드가 wait으로 자식을 reaping하기 전에 종료되는 것을 막기 위해 sema\_up, sema\_down 함수를 이용해서 자식 쓰레드의 load 함수가 실행된 후 부모가 실행될 수 있도록 하였다. 또한, thread 구조체 내에 flag 변수를 추가해 load가 성공했는지 실패했는지 알 수 있도록 하였다.

1. **연구 결과**
   1. **Flow Chart**
      1. File Descriptor

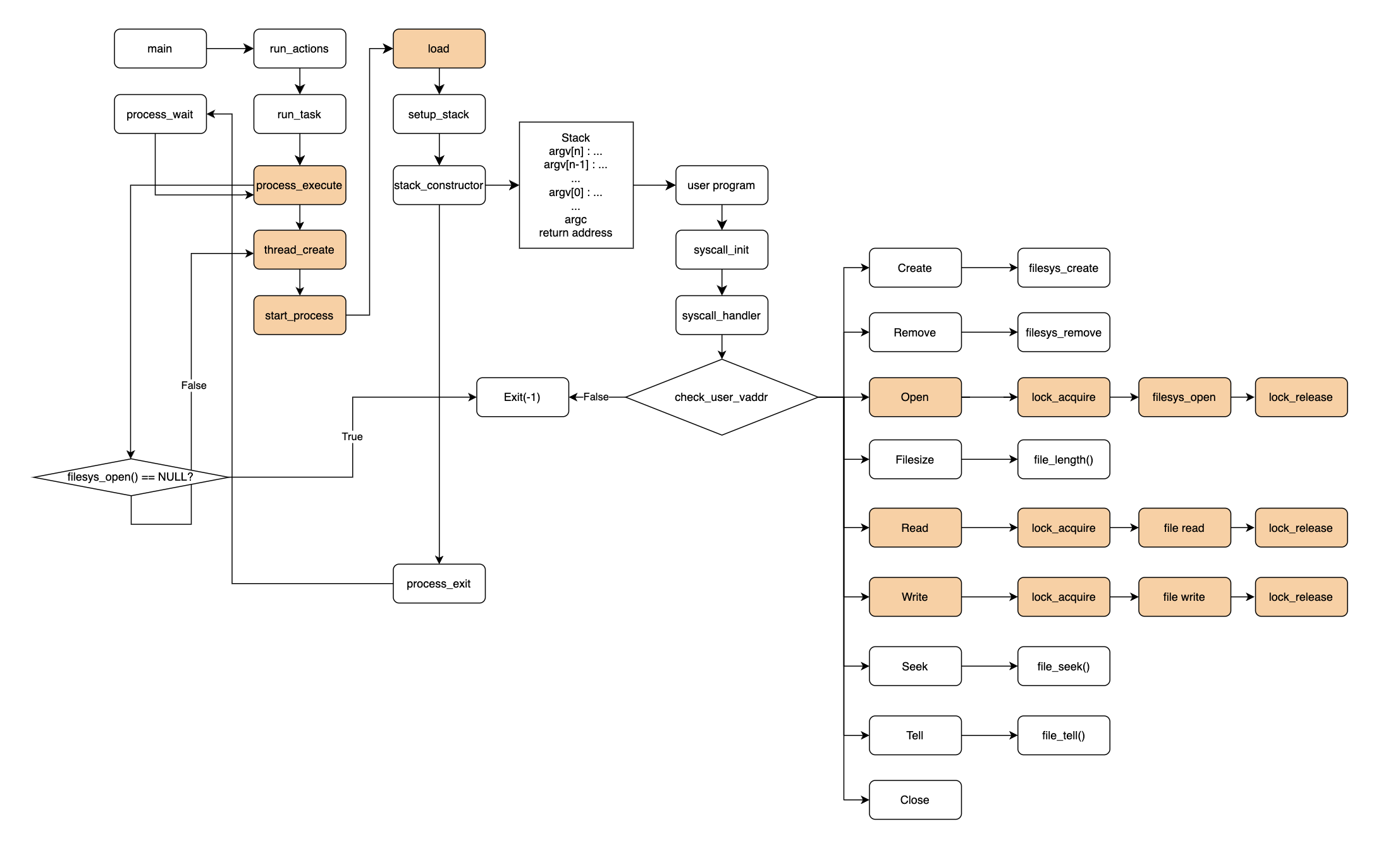


* + 1. System Calls

A diagram of a flowchart

Description automatically generated

* + 1. Synchroization in File System

****

* 1. **제작 내용**
     1. **File Descriptor**
     + **threads/thread.h**

A screenshot of a computer program

Description automatically generated

Struct file\* 타입의 배열인 fd\_table을 선언해 핀토스 매뉴얼에 따라 각 쓰레드마다 128개의 File Descriptor를 갖게 한다. 배열로 구현한 이유는 앞서 설명했듯이 int type의 fd를 테이블의 index로 활용하여 원하는 파일에 접근하기 유용하고 반복문을 이용해 빈 file descriptor를 찾는다던가 할 때 편리하므로 배열을 이용한다.

* + - **threads/thread.c**

파일 디스크립터 테이블을 초기화한다.

void init\_userprog(struct thread \*t) {

t->exit\_status = -1;

list\_init(&(t->children));

t->parent = running\_thread();

t->flag = 0;

// initialize file descriptor table

for (int i = 0; i < 128; i++) t->fd\_table[i] = NULL;

// initialize locks

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

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

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

list\_push\_back(&(running\_thread()->children), &(t->child));

}

* + 1. **System Calls**
    - **Userprog/syscall.c**

File 구조체를 이용하기 위해 filesys.h와 file.h 라이브러리를 이용한다.

#include "filesys/filesys.h"

#include "filesys/file.h"

그 다음 파일 시스템을 위한 시스템콜을 차례대로 구현하고, 이를 syscall\_handler 함수의 switch case문에 추가해준다.

**<syscall\_handler (struct intr\_fram \*f UNUSED)>**

static void

syscall\_handler (struct intr\_frame \*f UNUSED)

{

// f->esp points to # of syscall

// printf("syscall\_number: %d\n", \*(uint32\_t \*)(f->esp));

switch (\*(uint32\_t \*)(f->esp)) {

case SYS\_HALT:

Halt();

break;

case SYS\_EXIT:

check\_user\_vaddr(f->esp + 4);

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

break;

case SYS\_EXEC:

check\_user\_vaddr(f->esp + 4);

f->eax = Exec((const char \*)\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_WAIT:

check\_user\_vaddr(f->esp + 4);

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

break;

case SYS\_CREATE:

check\_user\_vaddr(f->esp + 16);

check\_user\_vaddr(f->esp + 20);

f->eax = Create((const char \*)\*(uint32\_t \*)(f->esp + 16), (unsigned)\*(uint32\_t \*)(f->esp + 20));

break;

case SYS\_REMOVE:

check\_user\_vaddr(f->esp + 4);

f->eax = Remove((const char \*)\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_OPEN:

check\_user\_vaddr(f->esp + 4);

f->eax = Open((const char \*)\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_FILESIZE:

check\_user\_vaddr(f->esp + 4);

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

break;

case SYS\_READ:

check\_user\_vaddr(f->esp + 4);

check\_user\_vaddr(f->esp + 8);

check\_user\_vaddr(f->esp + 12);

f->eax = Read((int)\*(uint32\_t\*)(f->esp + 4), (void\*)\*(uint32\_t\*)(f->esp + 8), (unsigned)\*(uint32\_t\*)(f->esp + 12));

break;

case SYS\_WRITE:

check\_user\_vaddr(f->esp + 4);

check\_user\_vaddr(f->esp + 8);

check\_user\_vaddr(f->esp + 12);

f->eax = Write((int)\*(uint32\_t \*)(f->esp + 4),(const void \*)\*(uint32\_t \*)(f->esp + 8), (unsigned)\*(uint32\_t \*)(f->esp + 12));

break;

case SYS\_SEEK:

check\_user\_vaddr(f->esp + 16);

check\_user\_vaddr(f->esp + 20);

Seek(\*(uint32\_t \*)(f->esp + 16), \*(uint32\_t \*)(f->esp + 20));

break;

case SYS\_TELL:

check\_user\_vaddr(f->esp + 4);

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

break;

case SYS\_CLOSE:

check\_user\_vaddr(f->esp + 4);

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

break;

case SYS\_FIBONACCI:

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

break;

case SYS\_MAX\_OF\_FOUR\_INT:

f->eax = Max\_of\_four\_int(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12), \*(uint32\_t \*)(f->esp + 16));

break;

default:

break;

}

}

**<Create>**

파일의 이름(file)과 크기(nitial\_size)를 인자로 받아 initial\_size bytes의 파일을 새로 만든다. File = NULL일 때 Exit(-1)로 종료한다. 미리 선언된 함수인 filesys\_create()를 이용한다.

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

if(file == NULL) Exit(-1);

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

}

**<Remove>**

파일의 이름(file)을 인자로 받아 이를 제거하고, Create과 마찬가지로 file = NULL일 때 Exit(-1)로 종료한다. 미리 선언된 함수인 filesys\_remove()를 이용한다.

bool Remove (const char \*file){

if(file == NULL) Exit(-1);

return filesys\_remove(file);

}

**<Open>**

파일의 이름을 인자로 받아 filesys\_open으로 파일을 연다. 이때 파일을 열기 전 인자로 받은 file이 NULL인지 Check하고, 파일 오픈 이후 정상적으로 열렸는지 한 번 더 확인하여 예외처리를 해준다. 만약 정상적으로 열렸다면, file descriptor 3부터 (STDIN = 0, STDOUT = 1, STDERR = 2) 마지막인 128을 순서대로 확인하여 비어있는 곳에 새 파일을 추가한다.

int Open (const char \*file) {

if(file == NULL) Exit(-1);

lock\_acquire(&filesys\_lock);

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

if(f == NULL){

lock\_release(&filesys\_lock);

return TID\_ERROR;

}

else {

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

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

if(strcmp(thread\_current()->name, file) == NULL) file\_deny\_write(f);

thread\_current()->fd\_table[i] = f;

lock\_release(&filesys\_lock);

return i;

}

}

}

lock\_release(&filesys\_lock);

return TID\_ERROR;

}

**<Filesize>**

파일 디스크립터를 인자로 받아, 해당 파일의 크기를 미리 선언된 file\_length 함수를 이용해 확인하고 이를 반환한다. 마찬가지로 valid한 file descriptor인지 확인하고 예외 처리를 해주었다.

int Filesize (int fd) {

if(!is\_valid\_file\_descrpitor(fd)) return TID\_ERROR;

if (fd == 0) return TID\_ERROR;

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

}

**<Read>**

프로젝트1에서 구현할 때에는 fd = 0 (STDIN)의 상황만 고려했다면, STDIN이 아닌 파일을 읽는 상황을 고려해 코드를 업데이트 해주었다. File descriptor를 인자로 받고, 해당 파일에서 입력을 받는데, fd = 0 일때에는 프로젝트1에서 사용한 코드를 그대로 사용해주고, 그 외에 파일에서 입력을 받을 때에는 미리 선언된 함수인 file\_read()를 이용해 파일에서 읽어준다. 또한, 유효하지 않은 파일 디스크립터에 대해선 Exit(-1)로 예외 처리를 해주었다.

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

lock\_acquire(&filesys\_lock);

check\_user\_vaddr(buffer);

unsigned i;

//fd = 0: reads from keyboard using input\_getc()

if (fd == 0) {

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

((uint8\_t \*)buffer)[i] = input\_getc();

if (((char \*)buffer)[i] == '\0') break;

}

lock\_release(&filesys\_lock);

return i;

}

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

if(!is\_valid\_file\_descrpitor(fd)){

lock\_release(&filesys\_lock);

Exit(-1);

}

int r = file\_read(thread\_current()->fd\_table[fd], buffer, size);

lock\_release(&filesys\_lock);

return r;

}

lock\_release(&filesys\_lock);

Exit(-1);

}

**<Write>**

Read와 마찬가지로 STDOUT이 아닌 파일에 Write하는 상황을 고려해 프로젝트 1의 코드에서 업데이트 해주었다. Fd = 1일 때는 기존에 프로젝트 1에서 구현한 코드를 사용하고, 이외의 Descriptor에 대해서는 file\_write 함수를 이용해 파일에서 읽어준다. 또한, Read와 마찬가지로 유효하지 않은 파일 디스크립터에 대해서 Exit(-1)로 예외처리를 해주었다.

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

lock\_acquire(&filesys\_lock);

check\_user\_vaddr(buffer);

//fd = 1: writes to the console

if (fd == 1) {

putbuf(buffer, size);

lock\_release(&filesys\_lock);

return size;

}

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

if(!is\_valid\_file\_descrpitor(fd)){

lock\_release(&filesys\_lock);

Exit(-1);

}

int r = file\_write(thread\_current()->fd\_table[fd], buffer, size);

lock\_release(&filesys\_lock);

return r;

}

else {

lock\_release(&filesys\_lock);

Exit(-1);

}

}

**<Seek>**

파일 디스크립터와 이동하고 싶은 위치(pos)를 인자로 받아 파일 포인터의 위치를 pos로 바꿔준다. 이때 미리 선언횐 file\_seek 함수를 이용하고, 마찬가지로 file descriptor가 유효한지 확인하고 예외처리를 해주었다.

void Seek (int fd, unsigned pos) {

if (!is\_valid\_file\_descrpitor(fd)) Exit(-1);

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

}

**<Tell>**

파일 디스크립터를 인자로 받고 file\_tell() 함수를 이용해 해당 파일의 포인터 위치(offset)을 반환한다. 마찬가지로 유효한 파일 디스크립터인지 확인하여 예외처리 해주었다.

unsigned Tell (int fd) {

if (!is\_valid\_file\_descrpitor(fd)) Exit(-1);

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

}

**<Close>**

인자로 받은 파일 디스크립터에 해당하는 파일을 file\_close()를 이용해 닫느다. 마찬가지로, 유효한 file\_descriptor인지 확인하여 예외처리 해주었다.

void Close (int fd){

if(!is\_valid\_file\_descrpitor(fd)) Exit(-1);

file\_close(thread\_current()->fd\_table[fd]);

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

}

* + - **Userprog/exception.c**

프로젝트 1의 조건에 not\_present일 때 exit(-1) 되도록 예외 조건을 추가하였다.

if(!user || !is\_user\_vaddr(fault\_addr) || not\_present) Exit(-1);

* + 1. **Synchronization in File System**
    - **threads/thread.h**

프로젝트 1에서 구현한 mem\_lock과 child\_lock에 추가해서 semaphore type의 load\_lock을 추가한다. 이름에서 알 수 있듯이 userprog/process.c에서 load 함수가 마무리 되는 것을 기다리기 위한 lock이다. 해당 lock을 추가한 이유는 테스트 케이스 중 multi-oom을 통과하기 위해 추가한 lock이다. 이외의 테스트 케이스는 시스템 콜과 예외처리를 이용해 잘 통과했지만, multi-oom에 대해서는 통과가 되지 않아 이를 위해 구현된 Lock이다. Multi-oom은 운영체제에서 메모리 누수가 있는지 확인하는 테스트 케이스인데, lock을 걸지 않으면 부모 쓰레드가 자식을 reaping 하기 전에 종료되어 메모리 누수가 발생하게 되는 경우가 발생한다. 따라서, 이를 방지하기 위해 자식이 로드 되기 전까지 부모는 lock을 걸어 기다리고 있도록 하여 이를 해결하였다. 또한, flag를 추가해 load 함수의 성공 여부를 쓰레드 내에 저장한다.

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. \*/

/\* Shared between thread.c and synch.c. \*/

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

#ifdef USERPROG

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

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

/\* Exit Status \*/

int exit\_status;

/\* File Descriptor Table\*/

struct file\* fd\_table[128];

/\*child\*/

struct list children;

struct list\_elem child;

struct semaphore mem\_lock;

struct semaphore child\_lock;

struct semaphore load\_lock;

struct thread \*parent;

int flag;

#endif

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

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

};

* + - **threads/thread.c**

새로 선언한 flag와 load\_lock을 초기화한다.

void init\_userprog(struct thread \*t) {

t->exit\_status = -1;

list\_init(&(t->children));

t->parent = running\_thread();

t->flag = 0;

// initialize file descriptor table

for (int i = 0; i < 128; i++) t->fd\_table[i] = NULL;

// initialize locks

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

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

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

list\_push\_back(&(running\_thread()->children), &(t->child));

}

* + - **userprog/process.c**

위에서 선언한 lock을 설명했듯이, start\_process 함수에서 load()가 끝나기 전에 부모가 종료되면 안되기 때문에 child thread의 load가 끝났을 때 lock을 풀어준다. 이때 load가 실패하면 flag = 0 으로 하고 쓰레드를 종료한다. 만약 성공했다면 flag = 1로 업데이트한다.

static void

start\_process (void \*file\_name\_)

{

char \*file\_name = file\_name\_;

struct intr\_frame if\_;

bool success;

/\* Initialize interrupt frame and load executable. \*/

memset (&if\_, 0, sizeof if\_);

if\_.gs = if\_.fs = if\_.es = if\_.ds = if\_.ss = SEL\_UDSEG;

if\_.cs = SEL\_UCSEG;

if\_.eflags = FLAG\_IF | FLAG\_MBS;

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

//Wait for finish loading

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

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

palloc\_free\_page (file\_name);

// If load failed, exit

if (!success) {

thread\_current()->flag = 0;

Exit(-1);

}

// If load success, set flag to 1

else thread\_current()->flag = 1;

process execute에서는 thread\_create에서 만들어진 thread의 tid와 동일한 값을 가진 쓰레드를 쓰레드의 child list에서 찾아 해당 쓰레드의 load가 끝날 때까지 wait할 수 있도록 lock을 걸어준다. 만약 child의 load가 실패했다면 (flag = 0 이라면) TID\_ERROR를 리턴한다.

tid\_t

process\_execute (const char \*file\_name)

{

char \*fn\_copy;

tid\_t tid;

/\* Make a copy of FILE\_NAME.

Otherwise there's a race between the caller and load(). \*/

fn\_copy = palloc\_get\_page (0);

if (fn\_copy == NULL)

return TID\_ERROR;

strlcpy (fn\_copy, file\_name, PGSIZE);

// Split file\_name and cmd\_name

char cmd[128];

strlcpy(cmd, file\_name, strlen(file\_name) + 1);

char \*save\_ptr;

char\* temp = strtok\_r(cmd, " ", &save\_ptr);

if(filesys\_open(cmd) == NULL) return TID\_ERROR;

/\* Create a new thread to execute FILE\_NAME. \*/

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

struct thread \*t;

struct list\_elem \*e;

struct thread \*child\_t = NULL;

// Find the child thread and wait for loading

for(e = list\_begin(&thread\_current()->children); e != list\_end(&thread\_current()->children); e = list\_next(e)) {

t = list\_entry(e, struct thread, child);

if(t->tid == tid) {

child\_t = t;

break;

}

}

sema\_down(&child\_t->load\_lock);

if (child\_t->flag == 0) return TID\_ERROR;

if (tid == TID\_ERROR) palloc\_free\_page (fn\_copy);

return tid;

}

* + - **userprog/syscall.c**

파일과 관련된 시스템 콜에서, 여러 쓰레드가 동시에 하나의 파일에 접근할 때 예기치 않게 발생하는 문제를 막기 위해서는 적절한 lock이 필요하다. 따라서, lock type 구조체의 filesys\_lock을 선언하고, syscall\_init에서 lock\_init 함수를 이용해 초기화하였다.

struct lock filesys\_lock;

void syscall\_init (void)

{

lock\_init(&filesys\_lock);

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

}

이렇게 초기화된 lock은 file을 직접적으로 파일을 수정하거나 읽는 시스템 콜(Open, Read, Write)에서 함수 실행할 때 락을 걸고(lock\_acquire), 작업이 끝나고 return 하기 전에 lock을 풀어주었다(lock\_release). 이렇게 함으로써 멀티 쓰레드 환경에서 하나의 파일에 여러 쓰레드가 동시에 접근할 때 발생하는 문제를 막아 주었다. 또한 Open 함수에서는 file\_deny\_write() 함수를 이용해 현재 오픈된 파일에 write 하는 것을 방지한다.

int Open (const char \*file) {

if(file == NULL) Exit(-1);

lock\_acquire(&filesys\_lock);

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

if(f == NULL){

lock\_release(&filesys\_lock);

return TID\_ERROR;

}

else {

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

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

if(strcmp(thread\_current()->name, file) == NULL) file\_deny\_write(f);

thread\_current()->fd\_table[i] = f;

lock\_release(&filesys\_lock);

return i;

}

}

}

lock\_release(&filesys\_lock);

return TID\_ERROR;

}

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

lock\_acquire(&filesys\_lock);

check\_user\_vaddr(buffer);

unsigned i;

//fd = 0: reads from keyboard using input\_getc()

if (fd == 0) {

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

((uint8\_t \*)buffer)[i] = input\_getc();

if (((char \*)buffer)[i] == '\0') break;

}

lock\_release(&filesys\_lock);

return i;

}

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

if(!is\_valid\_file\_descrpitor(fd)){

lock\_release(&filesys\_lock);

Exit(-1);

}

int r = file\_read(thread\_current()->fd\_table[fd], buffer, size);

lock\_release(&filesys\_lock);

return r;

}

lock\_release(&filesys\_lock);

Exit(-1);

}

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

lock\_acquire(&filesys\_lock);

check\_user\_vaddr(buffer);

//fd = 1: writes to the console

if (fd == 1) {

putbuf(buffer, size);

lock\_release(&filesys\_lock);

return size;

}

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

if(!is\_valid\_file\_descrpitor(fd)){

lock\_release(&filesys\_lock);

Exit(-1);

}

int r = file\_write(thread\_current()->fd\_table[fd], buffer, size);

lock\_release(&filesys\_lock);

return r;

}

else {

lock\_release(&filesys\_lock);

Exit(-1);

}

}

* + - **userprog/process.c**
  1. **시험 및 평가 내용**

모든 테스트 케이스를 통과할 수 있었다.

