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

담당 교수 / 분반 : 김영재 / 1반

이름 / 학번 : 문상영 / 20161581

개발 기간 : 10/1 ~ 10/14

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

project1에서 구현한 system call handler 함수를 추가로 개발한다. project2에서는 file system 에 해당하는 system call들 ( create, remove, open, close, filesilze, read, write, seek, tell)을 implement한다. 이중 read 와 write는 project1 에서 구현한 stdin, stdout에 대한 읽기와 쓰기에 추가로 file 에대한 read, write 를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

linux 운영체제는 시스템을 file의 형태로 추상화해서 관리한다. 이 때, 여러 프로세스나 thread 들이 file system에 접근하기 위해서 자신에게 할당된 file descriptor를 이용한다. file descriptor는 struct file\* 를 저장하고 이를 이용해 file에 접근해 file system 작업을 kernel 에 요청해 원하는 작업을 수행한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

create : 파일을 생성.

remove : 파일을 삭제.

open : 파일을 open 하고 file descriptor 할당

close : open 한 파일을 close하고 file descriptor 해제.

filesize : file의 크기를 반환

read : open 된 파일을 size 만큼 읽는다.

write : open 된 파일을 size 만큼 쓴다.

seek : 파일에서 read 나 write 할 위치를 바꾼다.

tell : 파일에서 다음 read나 write가 수행될 위치를 반환한다.

read write 의 경우 project1에서 stdin, stdout 을 구현한 것 이외에 file 에 대한 read write 도 추가 구현해야 한다.

3. Synchronization in Filesystem

동일한 파일에 대해서 read가 수행되는 동안 write가 concurrent 하게 수행되게 되면 원래 read 하려고 했던 data가 훼손되고, 또한 shared buffer에 대해서 여러 thread의 read가 동시에 이루어지거나 했을 때에도 critical section 이 atomic 하게 수행되지 못해서 문제가 발생할 수 있다. 따라서 lock이나 semaphore를 이용해서 critical section을 atoimc 하게 실행 시키도록 control 한다.

또한 executable file 에 대해서 write를 방지하기 위해서 file\_deny\_write와 file\_allow\_write를 이용해서 쓰기권한을 통제한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

우선 file descriptor table를 구현하기 위해 사용한 자료구조는 struct file\* 형 array 이다. file descriptor의 경우 process 마다 최대 128개의 file descriptor 가 있을 수 있는데 기본적으로 0,1,2 는 각 각 stdin ,stdout, stderr 로 reserved 되어 있다. file descriptor number는 0 이상의 정수이므로, 해당 인덱스에 open된 file 구조체의 주소값을 저장한다.

file descriptor number를 index로 file descriptor table 에 접근하면 해당 인덱스의 struct file\* 를 이용해서 파일에 접근할 수 있기 때문에, 구현 상 편의, 빠르게 특정 file에 접근할 수 있다는 장점 때문에 struct file\* 형 array 로 구현했다.

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)

create : initial\_size byte 크기의 새로운 파일을 생성하는 system call. 성공 실패 여부를 반환한다.

remove : 인자로 받은 file name의 파일을 삭제하는 system call. 성공 실패 여부를 반환한다.

open : 인자로 받은 file name의 파일을 open 하는 system call. 할당되지 않은 가장 작은file descriptor number를 반환한다.

close : 인자로 받은 file name의 파일을 close 하고 할당받은 file descriptor number를 해제하는 system call.

filesize : 인자로 받은 file descriptor number의 file의 크기를 반환한다.

read : 인자로 받은 file descriptor number의 file을 size 만큼 buffer로 읽어들이는 system call. size를 return 한다.

write : buffer에서 인자로 받은 file descriptor number의 파일에 size bytes 만큼 작성하는 system call.

seek : 파일에서 read 나 write 할 위치를 position으로 바꾸는 system call

tell : file descriptor number이 파일에서 다음 read나 write가 수행될 위치를 반환하는 system call.

**read write 의 경우 project1에서 stdin, stdout 을 구현한 것 이외에 file 에 대한 read write 도 추가 구현해야 한다.**

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

semaphore는 sema\_up, sema\_down 을 이용해서 context flow를 통제 할 수 있다. sema\_down 은 semaphore를 1 감소 시킨다. 그런데 만약 값이 0이라면 대기한다. 그러다 어떤 thread 에서 sema\_up을 하면 값이 1로 증가하고 대기하고 있던 thread를 wake해 다시 sema\_down을 수행하도록 한다.

lock의 경우도 semaphore 와 비슷하다. concurrent 하게 thread가 실행되는 상황에서 특정 작업에 대해 exclusive하게 thread가 접근하도록 통제한다. 그러나 semaphore와 다른 점은 semaphore의 경우 다른 thread에서도 sema\_up을 이용해 sema\_down으로 대기중인 다른 thread를 깨울 수 있지만 lock의 경우는 lock을 acquire한 thread에서만 release 할 수 있다는 점이 다르다.

동일한 파일에 대해서 read 와 write가 concurrent 하게 일어난다면 원본파일의 data가 훼손될 수 있기 때문에 동일한 파일에 대해서 read가 일어나는 도중에는 write가 일어나면 안된다. 또한 여러 thread에서 동시에 read를 할 때 critical section이 atomic 하게 실행되지 않으면 마찬가지로 shared buffer 에 의해 data가 훼손될 수 있다.

이를 예방하기 위해 lock이나 binary semaphore를 이용해서 shared data나 buffer에 대해서 exclusive 하게 thread의 접근을 통제 한다. 다시 말해 lock\_acquire나 sema\_down을 이용해서 특정 작업을 실행하는 thread가 한번에 하나씩만 접근하도록 통제한다.

추가로 executable file 에 대해서 write를 방지하기 위해서 file\_deny\_write와 file\_allow\_ write를 이용해서 쓰기권한을 통제할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10/1 :자료구조 및 알고리즘 설계

10/2~10/3: file descriptor , thread 구조체 modify

10/4~10/5 :create, remove, open, close 구현

10/6~10/7 :filesize, read, write 구현

10/8~10/10: seek, tell 구현, Denying Writes to Executable files 구현

10/11~10/13: make check 로 test, Synchronization in filesystem 구현

10/14: 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드

page fault 에서 read only 등의 write rights에 대한 handling 과정을 추가

* + 수정하거나 추가해야 하는 자료구조

thread 구조체에 file descriptor table 를 추가한다.

Synchronization in filesystem 을 구현하기 위해서 read, write를 수행할 때 필요한 lock을 선언한다.

* + 수정하거나 추가해야 하는 함수

thread 구조체에 file descriptor table 을 추가했으므로 이를 initialize 하는 과정을init\_thread 에 추가한다.

SYS\_CREATE , SYS\_REMOVE, SYS\_OPEN , SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL, SYS\_CLOSE를 handling 할 system call number로 handler에 추가하고 함수를 작성한다. read, write 에 대해서 fd가 STDIN, STDOUT 이외의 경우도 handling 한다.

1. File Descriptor

thread 구조체에 file descriptor table을 추가한다. table을 struct file\* array 로 선언한다. 0, 1, 2 index는 각각 stdin, stdout, stderr로 reserved 되어 있기 때문에 이에 따라서 초기화할 수 있도록 한다.

file descriptor table을 thread 구조체에 추가했으므로 init\_thread 에서 이를 초기화 해준다.

open system call 이 일어날 때, file descriptor table에서 비어있는 index 중 가장 작은 값을 반환하고 해당 index에 open 하는 파일의 파일 구조체 포인터를 저장한다. 이를 이용해 file descriptor table에 file descriptor number를 indexing해 접근하면 해당 file descriptor number에 해당하는 파일의 구조체에 접근할 수 있다.

close system call 이 일어날 때, file descriptor table 에서 전달받은 file descriptor number에 해당하는 index에 NULL 포인터를 저장해 초기화 한다.

2. System Calls:

project1 에서 구현한 system call handler 에서 user stack 에 있는 system call number에 따른 handler 작업 수행에 SYS\_CREATE , SYS\_REMOVE, SYS\_OPEN , SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL, SYS\_CLOSE 에 대한 내용을 추가한다. 그리고 이미 구현한 read, write 에 대해서도 fd가 STDIN, STDOUT 이외의 경우를 handle 하도록 modify 한다.

**bool create( const char\* file, unsigned initial\_size ) :** bool filesys\_create(const char \*name, off\_t initial\_size) 함수를 이용해서 구현한다. initial\_size byte 크기의 새로운 파일을 생성하고 성공 여부를 반환한다.

**bool remove( const char \*file ) :** bool filesys\_remove( const char \*name )을 이용해서 구현한다. 인자로 받은 file name의 파일을 삭제하고. 성공 여부를 반환한다.

**int open ( const char \*file ):** struct file\* filesys\_open( const char\* name ) 을 사용해서 구현한다. 인자로 받은 file name의 파일을 open 하고 file descriptor number를 반환한다.

**void close(int fd) :** void file\_close를 이용해서 구현한다. 인자로 받은 file name의 파일을 close 하고 할당받은 file descriptor number를 해제한다.

**int filesize(int fd) :** off\_t file\_length( struct file\* fp )를 이용해서 구현한다. 인자로 받은 file descriptor number의 file의 크기를 반환한다.

**int read( int fd, void \*buffer, unsigned size ) :** off\_t file\_read( struct file\* fp , void \* buffer, off\_t size )를 이용해서 파일에 대한 인자로 받은 file descriptor number의 file을 size 만큼 buffer로 읽는다.

**int write (int fd, const void \*buffer, unsigned size ):** off\_t file\_write( struct file\* fp, const void\* buffer, off\_t size)을 이용해 구현. buffer에서 인자로 받은 file descriptor number의 파일에 size bytes 만큼 쓴다.

**void seek( int fd, unsigned position ) :** void file\_seek( struct file\* fp, off\_t size)을 사용해서 구현. 파일에서 read 나 write 할 위치를 position으로 바꾼다.

**unsigned tell( int fd ) :** off\_t file\_tell( struct file \*file ) 을 사용해서 구현. file descriptor number이 파일에서 다음 read나 write가 수행될 위치를 반환한다.

3. Synchronization in Filesystem

filesystem에서 synchronization을 구현하기 위해서 syscall handler를 initialize 할 때 lock도 같이 initialize를 한다. shared memory / data 에 접근할 때 critical section을 보호하고 data가 손상되는 것을 막기위해서 exclusive 하게 하나의 thread 만 read 나 write를 수행해야한다.

이를 위해서는 critical section을 atomic 하게 실행하도록 해야한다. 그렇게 하면 critical section이 interleaving 되는 것을 막을 수 있고, 동일한 file에 대한 read 와 write를 Synchronize 시킬 수 있다.

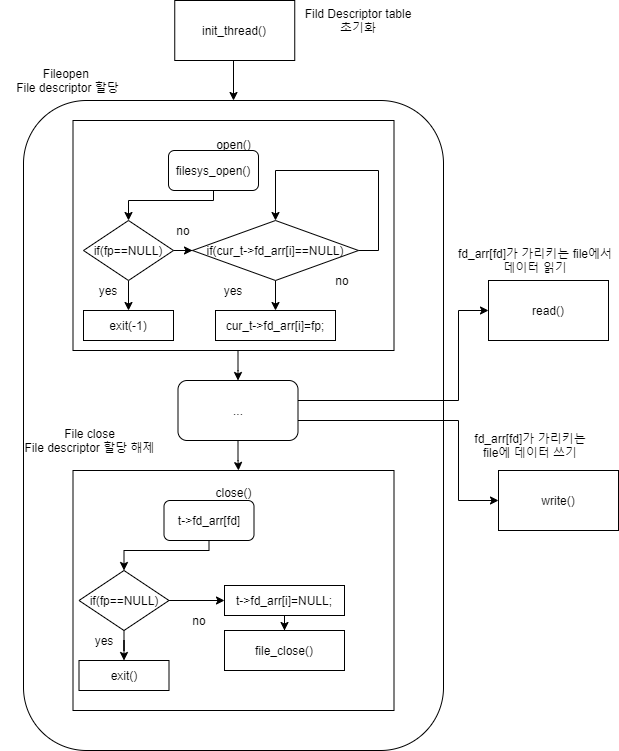
개발방법은 우선 syscall\_init 함수에서 lock\_init 함수를 호출해서 초기화한다. read와 write의 내부에서 critical section이 시작되는 부분에 lock\_acquire( &filesync\_lock ) 를 호출해 lock을 획득하고 critical section이 종료 되면 lock\_release(&filesync\_lock)를 이용해서 lock을 해제한다.

또한 executable files 에 대해서 write를 방지하려면 void file\_deny\_write( struct file \*)와 void file\_allow\_write( strruct file \* )를 이용해서 executable files 에 대한 쓰기권한을 통제해야 한다. 이 때 권한으로 인한 pagefault 문제를 해결하기위해 src/userprog/ exception.c 의 pagefault()를 not\_present에 대해서도 조건을 추가한다.

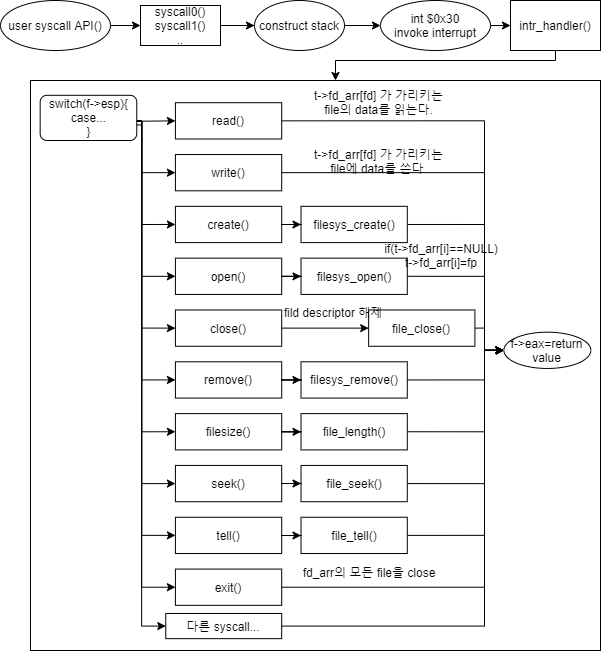
1. **연구 결과**
   1. **Flow Chart**

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

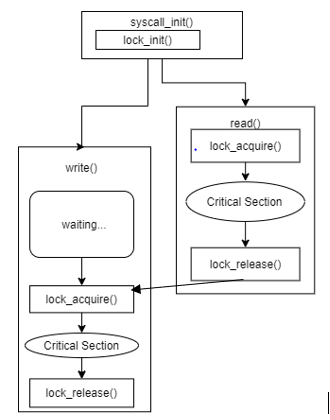
1.File Descriptor

****

2.System Call

****

3. Synchronization in Filesystem



* 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

system call 함수를 구현하면서 kernel 에 system call instruction을 요청하는 systemcall kernel API 사용 (아래에 설명), critical section을 보호하기 위해서 lock\_init, lock\_acquire와 lock\_release를 사용, 유저 메모리 영역인지 여부를 확인하기 위해 is\_user\_vaddr 사용 등

* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

multi-oom test에서 memory leak 에 대한 검사를 수행한다. 이를 처리하기 위해서 process를 execute 가 끝날 때, 현재 thread의 child thread 를 확인해서 종료되었는데도 할당 해제 되지 않은 resource 들을 해제하도록 처리했다. (아래에 설명)

1. File Descriptor

thread 구조체에 file descriptor table을 추가하고 table을 struct file\* array 로 선언한다. modify된 thread 구조체는 다음과 같다.

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. \*/

struct thread\* parent\_thread;

struct semaphore load\_lock;

struct semaphore wait\_lock;

struct semaphore mem\_reap\_lock;

struct list t\_child;

struct list\_elem t\_child\_elem;

int exit\_status;

struct file\* fd\_arr[FD\_MAX];

int is\_zombie;

#endif

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

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

};

file descriptor table 을 strcut file\* 형 array 로 구현했고 그 크기를 FD\_MAX로 선언했다.

struct file\* fd\_arr[FD\_MAX];

여기서 FD\_MAX 는 128로 linux에서 한 프로세스에 최대로 할당할 수 있는 file descriptor 의 개수이다.

이렇게 추가된 file descriptor table은 src/thread/thread.c 파일의 init\_thread() 에서 thread가 초기화 될 때 모두 NULL 값으로 초기화 되고 0, 1, 2는 각각 STDIN, STDOUT, STDERR 를 가리키도록 한다.

이렇게 할당된 kernel 메모리 영역에 있는 file descriptor table 을 이용해서 open한 파일에 대한 접근이 가능하고 이를 이용해 여러 filesystem instruction 을 요청한다.

예를 들어 open은 file descriptor table을 반복문으로 탐색하면서 NULL값이 저장되어있는(비어있는) index 중 가장 작은 index에 filesys\_open 으로 반환받은 파일 구조체 포인터를 저장하고 그 index를 return 한다.

또 close 에서는 parameter로 전달받은 file descriptor number로 file descriptor table 에 indexing한다. 해당 index에 비어있다는 의미로 NULL 값을 저장해 초기화 하고 file\_close를 호출한다.

2. System Calls:

system call handler 에서 user stack 에 있는 system call number를 확인한다. 그 값에 따라서 switch 문을 사용해 handler가 수행할 작업을 분류했다.

project2 에서 추가구현할 system call에 대해 handling을 수행하기 위해, SYS\_CREATE , SYS\_REMOVE, SYS\_OPEN , SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL, SYS\_CLOSE 을 case 에 추가했다.

read 의 경우 fd가 0 ( STDIN ) write의 경우 fd가 1( STDOUT ) 인 경우 만 handling 했는데 project 2에서는 fd가 다른 경우에 대해서 file descriptor table에 fd로 인덱싱해서 파일 포인터를 가져온다. 그리고 그 포인터를 이용해 파일에 대한 read, write를 수행했다.

추가로 각 system call 을 구현하면서 사용한 kernel API와 수정, 구현한 방식은 다음과 같다.

**bool create( const char\* file, unsigned initial\_size ) :** bool filesys\_create(const char \*name, off\_t initial\_size) 함수를 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 filesys\_create 함수를 호출해 반환했다.

**bool remove( const char \*file ) :** bool filesys\_remove( const char \*name )을 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 filesys\_remove 함수를 호출해 반환했다.

**int open ( const char \*file ) :** struct file\* filesys\_open( const char\* name ) 을 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 lock\_acquire를 하고 다음을 수행했다.

filesys\_open을 통해 반환된 struct file\*가 null이 아니라면 실행중인 thread의 이름을 확인해 같다면 executeable file 에 대한 write를 방지하기위해 file\_deny\_write(fp) 를 실행했다.그리고 반복문으로 current thread 의 fd\_arr 배열을 순회해 NULL이 저장된 가장 작은 인덱스에 fp를 저장하고 그 인덱스를 ret에 저장한다. 만약 file pointer 가 NULL 이라면 -1을 ret에 저장한다. 마지막으로 lock\_release를 호출한뒤, ret를 반환한다. 따라서 open에 성공하면 file descriptor number를, 실패하면 -1을 반환한다.

**void close(int fd) :** void file\_close( struct file\* fp )를 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 thread\_current() 함수를 이용해 현재 thread 구조체에 접근한뒤 fd\_arr[fd]의 값을 가져온다. 만약 fp==NULL 이면 할당받지 않은 file descriptor 이므로 exit(-1)을 호출하고 아니라면 fd\_arr[fd]에 NULL 값을 대입한다.

t->fd\_arr[fd]=NULL;

이후 file\_close(fp)를 호출해 파일을 닫는다.

**int filesize(int fd) :** off\_t file\_length( struct file\* fp )를 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 file\_length 함수를 호출해 반환한다.

**int read( int fd, void \*buffer, unsigned size ):** off\_t file\_read( struct file\* fp , void \* buffer, off\_t size )를 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 lock\_acquire를 호출해 critical section을 보호했다. 기존의 fd==0(STDIN) case 에 대해서는 return 직전 lock\_release를 호출하는 부분만 추가했고, fd가 0이 아닌 경우에 대해서는 다음 코드를 추가했다.

if(fd==0){

...

}else{

struct file\* fp=thread\_current()->fd\_arr[fd];

if(fp==NULL){

lock\_release(&filesync\_lock);

exit(-1);

}

ret=file\_read(fp,buffer,size);

}

해당 fd가 가리키는 주소가 NULL 인지 확인하고 아니라면 file\_read를 호출해 파일을 읽었고 NULL이라면 lock\_release를 호출하고 exit(-1)을 했다.

if-else문을 빠져나오면 lock\_release를 이용해서 lock을 해제하고 ret를 반환했다.

**int write (int fd, const void \*buffer, unsigned size ) :** off\_t file\_write( struct file\* fp, const void\* buffer, off\_t size)을 사용. fd==1인 경우 ret에 size를 저장한다. 아닌 경우 critical section을 보호하기 위해 lock\_acquire 한뒤, thread\_current() 함수로 fd\_arr[fd]에 저장된struct file\* 값을 가져온다. 포인터가 가리키는 주소가 NULL 이라면 lock\_release를 호출하고 exit(-1)로 종료한다. NULL이 아니면 file\_write를 호출해 파일에 쓰기를 진행하고 lock\_release 한뒤 ret값을 return 한다.

**void seek( int fd, unsigned position ) :** void file\_seek( struct file\* fp, off\_t size)을 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 file\_seek 함수를 호출했다.

**unsigned tell( int fd ) :** off\_t file\_tell( struct file \*file ) 을 사용. parameter로 전달받은 const char\* 가 NULL 이나 user memory 영역이 아닌곳을 참조하는지 is\_user\_vaddr 을 사용해 확인하고 맞으면 exit(-1) 아니면 file\_tell 함수를 호출했다.

3. Synchronization in Filesystem

test에서 요구하는 Synchronization은 shared data / memory에 대한 exclusive한 실행이 중요하므로 lock 을 acquire한 thread 가 작업이 종료되고 lock을 release하는 방식으로 구현할 수 있다. 따라서 lock 을 이용해서 synchronization을 구현했다.

file에 대한 작업을 수행하기 위해서는 system call 을 이용해서 kernel 에 요청을 해야하기 때문에 syscall\_handler에서 Synchronization을 수행하면 된다. 따라서 synchronization in filesystem을 구현하기 위해 src/userprog/syscall.c 에 struct lock filesync\_lock을 선언했다. handler는 pintos 가 booting 될때 등록되므로 void syscall\_init (void) 함수에서 lock\_init ( &filesync\_lock )을 호출했다.

앞서 system call 부분에서 설명했듯, open, read, write가 실행 될 때, lock을 이용해서 critical section을 감쌌다. 이렇게 되면 lock\_acquire가 호출되고 lock\_release되기 까지의 구간은 exclusive 하게 하나의 thread 에서만 실행되고 shared data / memory 의 corruption을 예방한다.

또한 executable files 에 대해서 write를 방지하기 위해 void file\_deny\_write( struct file \*)와 void file\_allow\_write( strruct file \* )으로 쓰기 권한을 통제했다. filename과 thread\_name을 비교해 둘이 같다면 file\_deny\_write를 호출했고, process\_exit에서 file이 unload 될 때, file\_allow\_write를 호출했다.

* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

multi-oom test에서 메모리 leak 에 대한 검사가 있어서 비정상적으로 종료된 child thread 에 대해 resource를 해제해야 했다. 이를 위해서 thread 구조체에 int is\_zombie 라는 flag를 추가했고, process\_execute가 tid를 return 하기 전에 thread 구조체의 struct list t\_child 를 반복문으로 순회하면서 flag를 확인했다. 만약 flag가 set되었다면 process\_wait를 이용해 resource 를 해제해 해결했다.

이 때 반복문을 순회하기 위해서 list\_begin, list\_end, list\_next, list\_entry 를 이용했다. struct list\_elem\* cur\_e을 우선 list\_begin을 이용해 list의 head로 초기화 했고, 종료 조건은 list\_end를 이용해 확인했다. list\_item에 접근하기 위해 list\_entry를 사용했고 다음 list로 iterate 하기 위해 list\_next를 사용했다.

또 한가지는 exit 로 thread 가 종료될 때 파일을 스스로 close 해주지 않기 때문에 file descriptor table 에서 NULL 값이 아닌, 다시말해 open 된 파일이 있으면 close 를 호출해서 파일을 닫아준뒤 thread\_exit()를 호출해 해결했다.

* 1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부
* 텍스트이(가) 표시된 사진

  자동 생성된 설명