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

담당 교수 / 분반 : 박성용 교수님

이름 / 학번 : 박종하 20182202

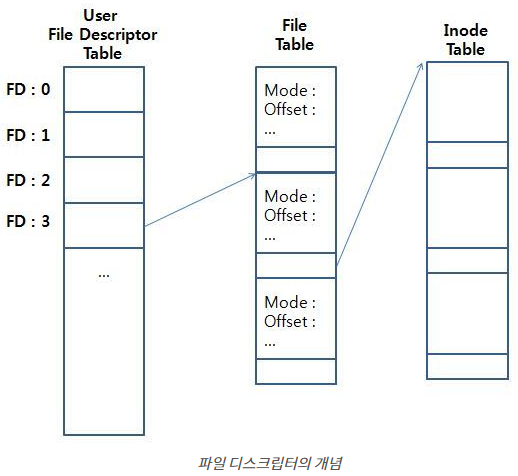
개발 기간 : 2022.10.06~2022.10.15

1. **개발 목표**

File descriptor를 구현함으로써, read, write system call function을 보완하고, file system과 관련된 open, create , remove, open, close, filesize, seek, tell 과 같은 system call 함수를 구현한다. 시스템 콜이 호출되어 커널 모드에서 파일 시스템에 접근할 때 동시에 프로세스가 접근하지 못하도록 synchronization 과정 또한 필수적으로 구현하여야 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



다음과 같이 각 프로세스는 자신의 file descriptor 테이블이 존재한다. 프로세스가 시스템 콜을 이용해 파일을 접근하는데, 파일을 접근하기 위해서 파일 디스크립터의 값을 통해 파일을 지칭한다. 따라서 파일 디스크립터를 구현하는 이유는 시스템 파일에 대해서 유저모드에서 직접 접근하는 것이 아닌, 시스템 콜을 통해 커널 모드에서 처리하도록 하기 위함이다. 따라서 유저모드와 커널모드를 분리시킴으로써 안정성의 강화한다.

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

이번 프로젝트에서는 파일 시스템과 관련한 system call 을 구현한다. 바로 open, read, write, close, filesize, seek, tell, remove과 같은 함수이다. 앞서, 파일 디스크립터를 구현하고 파일 시스템과 관련된 함수를 구현함으로써 파일 system I/O를 커널 모드에서 가능하게 한다.

3. Synchronization in Filesystem

파일 시스템에서 이루어지는 과정 중에 동시에 프로세스가 접근하지 못하는 critical section 에 대해 한 프로세스만 접근할 수 있도록 적절히 lock 을 걸어줘야 한다. 이를 통해 파일 I/O 동작 과정 중 발생할 수 있는 오류를 방지한다. 추가적으로 pintos 상에서는 메모리가 로드 된 이후에 지우는 행위가 금지 되어있기 때문에 file\_deny\_write 함수를 사용하여 막아야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술



핀토스 매뉴얼에 따르면, 최대 128개의 파일을 열수 있다고 했으므로, file 구조체를 담을 수 있는 128개 크기의 File \* 구조체 배열을 선언하였다. 정적 배열로 선언하기에 128개의 크기는 한정적이기 때문에 부담 스럽지 않을 뿐더러, 관리에 용이하기 때문이다. 또한 cur\_num\_fd를 선언했는데, 이는 file descriptor 에서 현재 접근하는 파일 디스크립터의 number를 의미한다.

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)

- open

파일을 파일 디스크립터에 연결해주는 함수이다. 파일 시스템 관련 동작을 하기 위해서는 선행되어야 하는 함수로, 해당 number 의 파일 디스크립터가 만약 null 이라면, 파일 구조체를 파일 디스크립터 배열에 할당한다.

- read

주어진 파일 디스크립터 넘버에 해당하는 파일 디스크립터에 접근하여 파일의 내용을 사이즈 만큼 buffer를 통해 읽는다. 0일 때는 표준 입력을, 그 외의 경우에는 파일 디스크립터를 사용한다.

- write

1일 때는 표준 출력을 그 외의 경우에는 파일 디스크립터를 이용한다. 파일 디스크립터 넘버에 해당하는 파일 디스크립터에 접근하여, 파일의 내용을 buffer를 통해 write 하는 함수이다.

- close

주어진 파일 디스크립터 넘버에 해당하는 파일 디스크립터에 접근하여 파일이 null이 아니라면 close 하고 null 처리를 해준다.

- filesize

주어진 파일 디스크립터 넘버에 해당하는 파일 디스크립터에 접근하여 파일의 사이즈를 반환하는 함수이다.

- remove

파일 이름과 일치하는 file 을 remove 하는 함수이다. 이미 open 된 파일 디스크립터의 파일을 접근하는 것과는 관계가 없다. remove하는 것이 close 하는 것과는 관계가 없다.

- seek

주어진 fd 에 해당하는 오픈 파일에 접근하여, position에 해당하는 다음 바이트를 바꾼다. 즉 offset을 바꾼다.

- tell

주어진 fd 에 해당하는 오픈 파일의 position 의 다음 바이트를 읽거나 write 하기 위해 position을 반환한다.

- create

file 이름을 갖는 새로운 파일을 initial size 만큼 생성한다. 파일을 생성하는 것은 open 함수와는 관련이 없다.

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

락과 세마포는 크게 두가지 차이점이 있다. Lock 은 binary semaphore 로, 0,1 로 이루어진 semaphore 의 special case 이다. 그래서 sema\_init(1) 을 하는 것과 같다. 또한 세마포는 어떤 thread 가 down 을 하면 다른 thread 에 의해 up 을 할 수 있지만, lock 은 acquire 를 건 thread 가 release를 해줘야 한다.

- read, write, open 함수에서 file\_load\_lock lock 이용.

Read, write, open 을 할 때에는 하나의 파일에 대해서 여러 프로세스가 접근하면 안된다. 즉, 한 프로세스가 특정 파일을 읽을 때 다른 프로세스가 특정 파일을 write 해서는 안된다. 따라서 이를 막기 위해 lock을 사용한다. 특정 프로세스가 read를 할 때 lock\_acquire 를 통해 세마포 값을 1 감소 시키고 음수라면, 대기한다. 하지만 1로 초기화를 했기 때문에 음수가 아니므로 해당 프로세스는 critical section 에 진입하게 된다. 해당 프로세스가 critical section 을 처리하고 lock\_release 를 하게 되면 세마포 값을 1 증가 시킨다. 이를 통해 critical section으로 지정한 open에서 filesys\_open을 호출한 부분, write에서 file\_write , read 에서 file\_read 를 호출한 부분이 다른 프로세스에 의해 침범 받지 않는다.

- parent process 가 child\_process 생성 및 로드를 기다리기 위한 is\_load\_lock semaphore 이용.

Parent process 는 child process 가. Create 될 때 까지 이를 확인하고 기다려 줘야 한다. 따라서 semaphore is\_load\_lock 을 만들어서 sema\_init(0) 을 해준다. 이는 Parent 가 child 를 create 하고 sema\_down 을 하게 되면 0에서 -1 로 감소하고 음수가 아닐 때까지 대기하게 된다. Sema\_up 을 하는 부분은 child 가 create 되는 부분, 즉 start\_process 에서 load 를 진행하고 parent의 세마포에 sema\_up 을 해주게 되면, 1이 증가하고 음수가 아닌 시점에 parent 프로세스를 깨울 수 있다. 만약 로드가 적절히 진행이 안되면 child process 의 load flag 를 설정하여 parent 프로세스가 깨워진 후, 자식 프로세스의 상태를 확인하며 process\_wait 시켜주며 자식 프로세스를 처리해 준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

**텍스트이(가) 표시된 사진

자동 생성된 설명**

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

기존에 표준 입력, 표준 출력만을 지원한 read, write system call function 을 파일에 대해서 접근 가능하도록, 파일 디스크립터를 설정하고 파일을 오픈하여 파일 디스크립터 넘버를 통해 각 파일을 접근할 수 있도록 한다. Read, write 함수는 작동될 때 동시에 프로세스가 접근하지 못하도록 lock을 걸어주고 각각 file.c 에 있는 file\_read, file\_write 함수를 호출한다.

* Exit 함수 수정

프로세스가 exit 할 때, 현재 사용하고 있는 파일 디스크립터를 닫아줘야 한다. 따라서 현재 파일디스크립터가 null 값이 아니라면 close 함수를 호출하고, null 값으로 초기화한다.

* Exception.c 의 page Fault 함수 수정

Physicall address 에 매핑 되지 않는 address 에 대해서 오류를 수정하기 위해 페이지에 실제 virtual memory 에 해당하는 실제 메모리가 존재하는지 의미하는 not\_present 변수를 이용하여 여부를 확인한다.

* Start\_process, process\_execute 함수 수정

부모 프로세스는 자식 프로세스의 생성이 올바르게 작동 안될 시 자식 프로세스를 기다리고, 이에 대한 처리를 해줘야 한다. 즉 자식 프로세스보다 일찍 죽어서는 안되며 자식 프로세스의 생성 여부를 확인하고 기다려 줘야 한다. 따라서 thread/thread.h 에 선언 된 is\_load\_lock 변수를 이용하여 parent process 가 process.c 함수에서 thread\_create 을 할 때 sema\_down을 진행하여, 자식 프로세스가 모두 로드될 때 까지 기다리고, 자식 프로세는 start\_process.c 함수에서 Load 를 하고서 sema\_up(&thread\_current() -> parent->is\_load\_lock) 을 통해 부모 프로세스를 깨워준다. 부모 프로세스는 자식 프로세스가 모두 로드 되면, child\_list 를 돌면서 자식 프로세스 중에서 로드가 정상적으로 되지 않은 프로세스를 확인하며, 자식 프로세스의 is\_success == false 라면, process\_wait 을 하여, 해당 자식 프로세스가 종료되는 것을 기다려주는 함수를 호출하고 에러 넘버를 반환한다.

* Open 함수 추가 수정

: 핀토스 상에서는 실행되고 있는 프로세스를 지워서는 안된다. 원래 프로세스에서는 메모리가 로드된 이후에 프로세스를 지울 수 있다. 하지만 핀토스는 이를 금지하기 때문에, file\_deny\_write 함수를 이용하여 막아 준다. 즉 파일을 open 하는 함수에서 thread\_current()->name 과 file 의 이름이 같으면 file\_deny\_write 함수를 호출하여 이를 막는다.

* + 수정하거나 추가해야 하는 자료구조
* Thread/thread.h

1. parent 명시 : 현재 돌고 있는 running\_thread() 를 자식 프로세스의 parent 로 설정한다.
2. is\_success : 각 프로세스가 정상적으로 load 되었는지 확인 하는 flag 이다. 처음에는 true 로 초기화 한다.
3. is\_load\_lock : 각 프로세스(thread) 마다 Is\_load\_lock 세마포 선언 ,부모 프로세스가 자식 프로세스가 모두 로드 될 때 까지 기다려주기 위해 설정한 lock 으로 부모프로세스가 자식 프로세스를 thread\_create 할때 semaphore 다운을, 그리고 자식이 load 를 다하면 부모의 세마포를 up 시켜주는 방식으로 구현.
4. File descriptor (File 구조체 배열) 선언, cur\_fd\_num 선언

* Thread/thread.c

1. Thread.h 에서 선언했던 Is\_success, parent , is\_load\_lock 초기화 해준다.
2. Thread.h 에서 선언했던 file\_descriptor fd\_list[128], cur\_fd\_num 을 초기화 해준다.

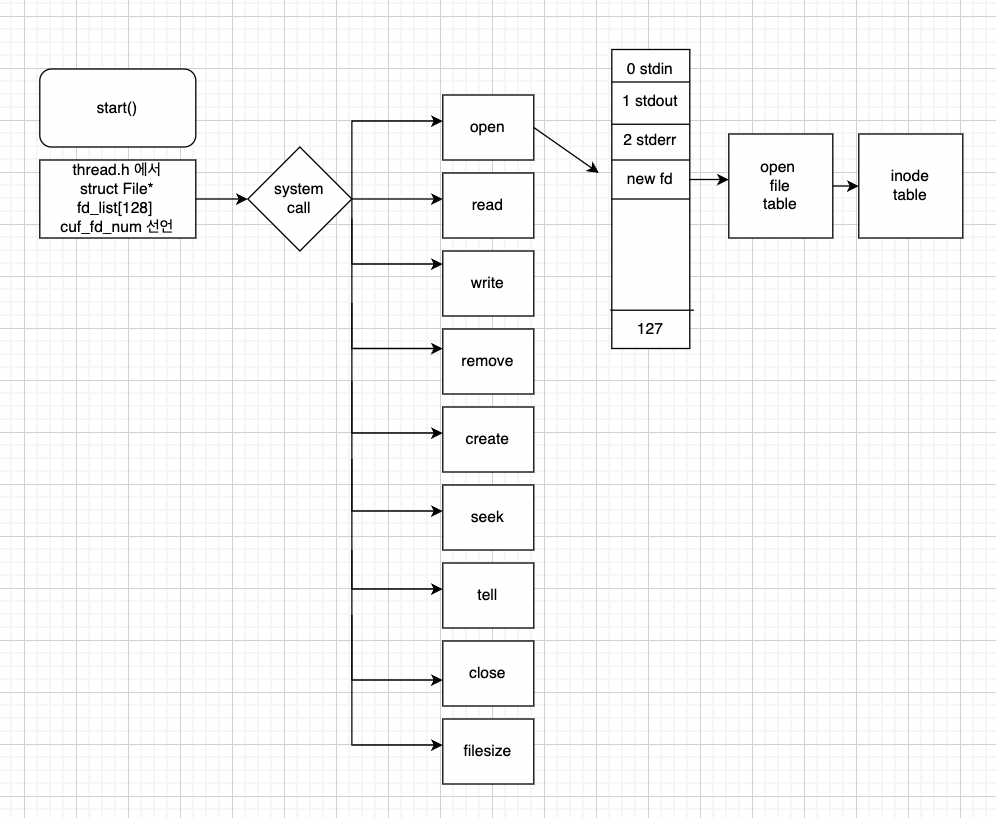
* Userprog/syscall.c

1. File\_load\_lock 선언 : 프로세스마다 read, write , open 을 할 때, 독립 적인 critical section을 위해 file\_load\_lock 을 선언하고, syscall\_init 에서 lock\_init 을 통해 초기화 해준다.
   * 수정하거나 추가해야 하는 함수

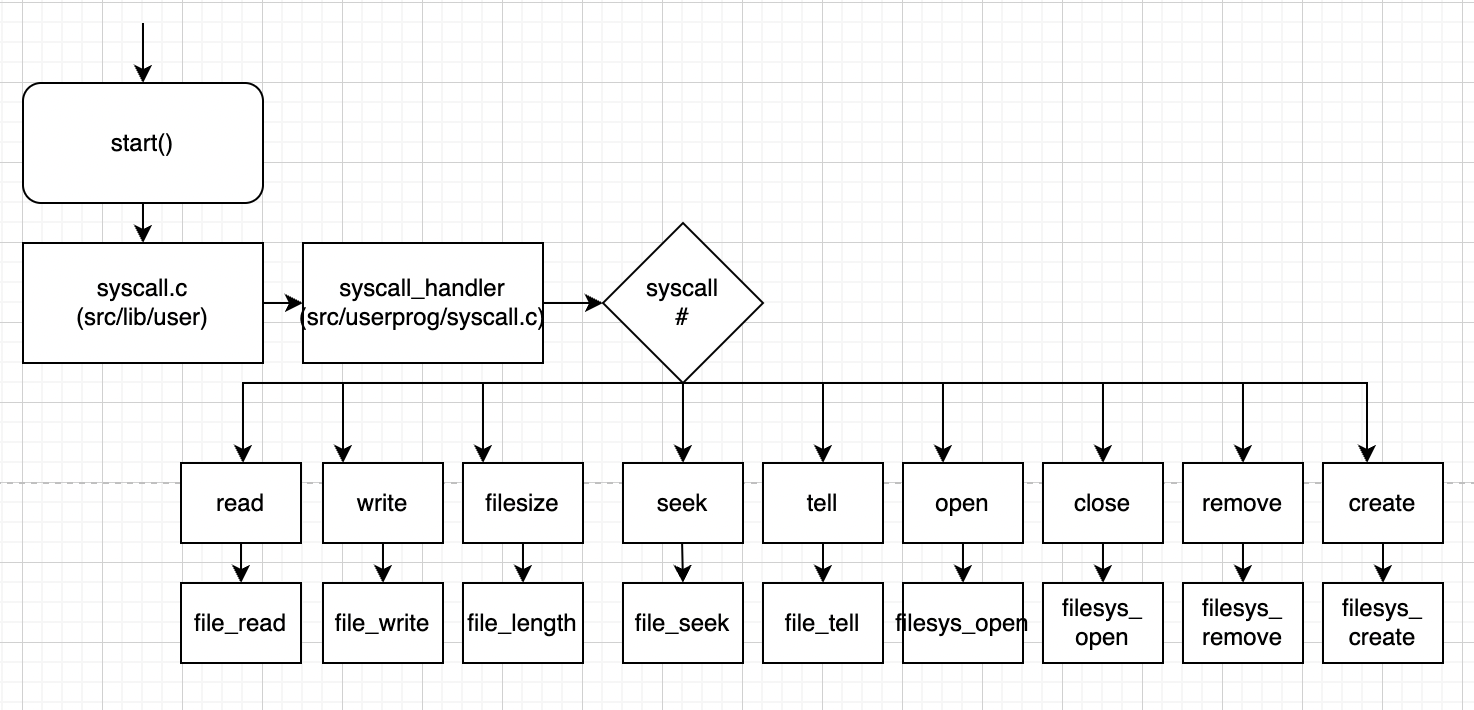
* Lib/user/syscall.c 에서 유저모드가 시스템 콜을 호출하면, userprog/syscall.c 에서 주어진 syscall number 에 맞게 system call function 을 호출하여야 한다. F->esp 함수를 통해 쌓여진 스택의 주소의 데이터를 system call function 의 인자로 넣어 호출한다. 또한 f->esp 가 커널 영역을 침범하는 지에 대한 오류 또한 확인하여야 한다.
* Open, close, seek , tell, remove , create, filesize 함수 추가
* Open, read, write에서 가장 중요한 부분은 synchronization 을 처리하는 Lock 부분이다.lock\_init 을 한 후, 이를 critical section 에 대해 적절히 lock\_acquire, lock\_release 를 한다.
* Remove, create 함수는 파일이 null 인지 확인하고 각각 filesys\_remove, filesys\_create 를 호출한다.
* Close, filesize, seek, tell은 주어진 인자 fd 에 해당하는 파일 디스크립터가 존재하는지 null 인지 확인하고, 각각 file\_close, file\_seek, file\_tell 함수를 호출한다.

1. **연구 결과**
   1. **Flow Chart**

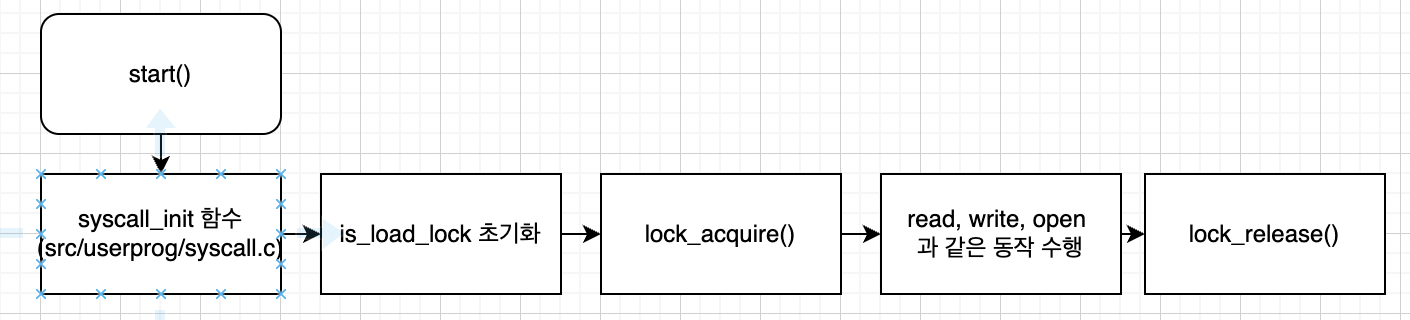
* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
* 1. File descriptor

****

**2. System Call**

****

**3. synchronization**

****

* 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
* File Descriptor

**텍스트이(가) 표시된 사진

자동 생성된 설명**

파일 디스크립터를 구현하기 위해 파일 구조체 배열 128 크기를 선언하였다.

각각의 프로세스는 각각 파일 디스크립터를 독립적으로 존재한다. 즉 child process 에게도 상속되지 않는다. Thread.c에서 파일 디스크립터를 null 로 초기화 하였다. 그리고 현재 파일 디스크립터 넘버를 의미하는 cur\_fd\_num 은 3으로 초기화 해준다. 왜냐하면, 0은 표준입력, 1은 표준출력, 2는 표준 에러이기 때문이다.

* System Call
  + - 1. Create

텍스트이(가) 표시된 사진

자동 생성된 설명

해당 파일이 존재하지 않으면 exit(-1) 로 나간다. 그 외의 경우에는 Kernel api 인 filesys\_create 함수를 호출하여 root directory 에 파일을 생성하도록 한다.

* + - 1. Remove



해당 파일이 존재하지 않으면 exit(-1) 로 나간다. 그 외의 경우에는 Kernel api 인 filesys\_remove 함수를 호출하여 root directory 에 파일을 지운다.

* + - 1. Open

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저 파일이 존재하지 않으면 exit(-1) 로 나간다. 그 후, 파일을 open 하는 kernel api 인 filesys\_open 함수를 호출할 때 lock을 걸어 준다. 파일 디스크립터 최대 크기인 128 까지 파일 디스크립터 넘버를 옮겨가며 Null 인 구간을 찾는다. 그리고 실행되고 있는 파일을 지우는 것을 방지하기 위해 strcmp 를 통해 file\_deny\_write 함수를 호출해주고 파일 디스크립터와 해당 오픈된 파일을 연결시켜주는 작업을 진행한다.

* + - 1. Read

텍스트이(가) 표시된 사진

자동 생성된 설명

Lock 을 통해 critical section 을 침해하지 않도록 한다. Fd가 1,2 일 때는 read 에서 처리하지 않으므로 Lock 을 release 하고 -1 을 반환한다. 만약 표준 입력일 때는 buffer 로 읽은 후, lock\_release 로 풀고, 읽은 바이트 수를 반환한다. Fd>=3 일때는 fd num 에 해당하는 fd 를 가져온 후, file\_read 함수 호출을 통해 size 만큼 읽어 온다. 그 후 lock \_release 하고 반환하는 방식이다.

* + - 1. Write

텍스트이(가) 표시된 사진

자동 생성된 설명

Lock 을 통해 critical section 을 침해하지 않도록 한다. Fd가 0, 2일 때는 write에서 처리하지 않으므로 Lock 을 release 하고 -1 을 반환한다. 만약 표준 출력일 때는 buffer 로 write한 후, lock\_release 로 풀고, write 은 바이트 수를 반환한다. Fd>=3 일때는 fd num 에 해당하는 fd 를 가져온 후, file\_write 함수 호출을 통해 size 만큼 write 온다. 그 후 lock \_release 하고 반환하는 방식이다.

* + - 1. Seek

텍스트이(가) 표시된 사진

자동 생성된 설명

파일 디스크립터 넘버에 해당하는 파일 디스크립터를 가져오고, null 일 때 오류를 처리한다. 그리고 file\_seek 함수를 호출하여, 현재 pos 를 position 으로 셋팅해준다.

* + - 1. Tell

텍스트이(가) 표시된 사진

자동 생성된 설명

파일 디스크립터 넘버에 해당하는 파일 디스크립터를 가져오고, null 일 때 오류를 처리한다. 그리고 file\_tell 함수를 호출하여, 현재 pos 에 대한 다음 바이트 값을 return 해준다.

* + - 1. close

텍스트이(가) 표시된 사진

자동 생성된 설명

파일 디스크립터 넘버에 해당하는 파일 디스크립터를 가져오고, null 일 때 오류를 처리한다. 그리고 file\_close 함수를 호출하여, 오픈되어있는 파일을 close 해주고, 클로즈 된 파일 디스크립터는null로 초기화한다.

* + - 1. filesize

텍스트이(가) 표시된 사진

자동 생성된 설명

파일 디스크립터 넘버에 해당하는 파일 디스크립터를 가져오고, null 일 때 오류를 처리한다. 그리고 file\_length 함수를 호출하여, 파일의 크기를 반환한다.

* Synchronization

텍스트이(가) 표시된 사진

자동 생성된 설명

Read, write, open 에서 synchronizaion을 위해 사용한 Lock 선언과 초기화 부분이다. 앞서 read, write, open 함수에서 lock 을 통해 synchronization 을 수행하는 부분을 기재하였다.

* 기타
  1. Bad-jump, bad-read, bad-write 의 오류

테스트 케이스를 통해 공통점을 확인해 본 결과, 매핑 되지 않은 주소에 대하여 읽으려고 하는 상황이 반복되고 있었다. 처음에는 시스템 콜에서 처리되지 않는 오류가 있나 read, write function 만 보완하려고 했지만, 매핑되지 않는 것은 페이지 폴트 핸들러를 통해 처리해야함을 깨닫고 exception.c 에서 not\_present 의 여부로 처리를 해주었다. 이를 통해 문제를 해결할 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + 1. Multi-ootm 에서의 오류

해당 테스트는 자식 프로세스가 실행하는 것을 fail 할 때까지 실행시켜서 parent 프로세스의 동작을 확인해보는 테스트임을 알았다. 하지만 해당 테스트에서 자식 프로세스가 exit 하지 않는 것을 보아하니, 부모 프로세스가 새로운 자식프로세스 생성할 때 성공여부에 대해서 기다리고 적절한 처리를 해주지 않음을 깨닫았다. 따라서 이를 위해 thread의 정의에 세마포(is\_load\_lock)를 정의 하고 0으로 초기화 시켜준다. is\_load\_lock 변수를 이용하여 parent process 가 process.c 함수에서 thread\_create 을 할 때 sema\_down을 진행하여, 자식 프로세스가 모두 로드될 때 까지 기다리고, 자식 프로세는 start\_process.c 함수에서 Load를 하고서 sema\_up(&thread\_current() -> parent->is\_load\_lock) 을 통해 부모 프로세스를 깨워준다. 부모 프로세스는 자식 프로세스가 모두 로드 되면, child\_list 를 돌면서 자식 프로세스 중에서 로드가 정상적으로 되지 않은 프로세스를 확인하며, 자식 프로세스의 is\_success == false 라면, process\_wait 을 하여, 해당 자식 프로세스가 종료되는 것을 기다려주는 함수를 호출하고 에러 넘버를 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명

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

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명
* 텍스트이(가) 표시된 사진

  자동 생성된 설명

주어진 테스트 케이스를 모두 만족하는 것을 볼 수 있다.

Cf) 정리 : <https://magical-authority-285.notion.site/Project2-3e54827f4d7f41e38cca845c920809a2>