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

담당 교수 / 분반 : 김영재 / 2분반

이름 / 학번 :

개발 기간 : 2024-10-28 ~ 2024-10-28

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

1. File System Call Implementation  
   Project1에서 구현한 system call handling 함수들에 이어서 이번에는 파일 입출력과 관련된 system call 호출에 대한 작업을 처리하는 함수들을 구현한다. 파일 생성, 파일 제거, 파일 읽기, 파일 쓰기, 파일 크기 얻기 등등 여러 파일 입출력 관련 동작을 구현한다. 특히 read(), write()의 경우 기존에는 표준 입출력에 대해서만 구현하였기 때문에 이번 프로젝트에서 추가적인 구현이 필요하다.
2. Denying Writes to Executable Files  
   실행 중인 파일은 이미 메모리에 올라가 있는 상태이기 때문에 사실 쓰기 및 삭제를 해도 큰 영향이 없다. 하지만 pintos에서 이를 원치 않으므로 실행 중인 파일에 대한 쓰기 작업을 막을 것이다. Pintos에서 제공하는 file\_deny\_write() 함수와 file\_allow\_write() 함수를 이용한다.
3. Protect Critical Section  
   여러 프로세스가 공유된 데이터에 동시에 접근하면 race condition 발생과 같은 synchronization 문제가 발생할 수 있다. 이렇게 공유된 데이터에 접근하는 code segment를 critical section이라 하는데, 하나의 프로세스가 critical section에 진입하면 다른 프로세스는 critical section에 진입하지 못하게 막아 synchronization 문제를 해결할 것이다.
4. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor  
File descriptor를 구현하여 파일 입출력 관리와 자원 관리를 효율적으로 할 수 있게 된다. 프로세스가 파일을 open하여 파일에 대한 descriptor 번호를 얻게 되고 해당 번호를 기반으로 write, read와 같은 파일과 관련된 여러 동작을 수행할 수 있다. 그리고 각 프로세스마다 고유의 file descriptor table을 가지므로 프로세스 간의 자원 보호가 가능하게 된다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

파일 입출력에 대한 system calls를 구현하여 파일을 생성(create)할 수 있다. 그리고 파일을 열어(open) 파일에 대한 여러 operation을 수행할 수 있게 된다. (write, read, seek, tell, …). 또한 open한 파일을 닫아(close) 프로세스에서 자원을 효율적으로 관리할 수 있게 된다.

3. Synchronization in Filesystem

프로세스에서 공유된 데이터에 접근하게 되는 code segment를 critical section이라고 한다. 같은 데이터에 대해 서로 다른 프로세스가 동시에 접근하는 상황을 방지하기 위해 한 프로세스가 critical section에 진입하면 다른 프로세스는 진입하지 못하게 막아서 file system에 대한 synchronization을 보장할 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술  
File 구조체의 포인터형에 대한 배열을 선언하였고, pintos manual을 참조하여 크기는 131로 하였다. Read, write 등의 system call에 대한 인자로 fd 번호를 받아오면 배열의 fd번째 index가 이에 해당되는 file 구조체의 위치가 된다. Linked list를 이용할 수 있지만 system call이 호출될 때마다 해당 fd에 대한 file 구조체를 탐색해야 하므로 비효율적이라 판단하여 fixed size의 배열을 이용하였다.

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)

(1) create : 주어진 이름을 가진 새로운 파일이 root directory에 생성되어 공간을 할당한다.

(2) remove : 해당 이름을 가진 파일을 삭제한다.

(3) open : 주어진 파일에 대한 file descriptor가 생성되어 프로세스에게 번호를 반환한다. 이 번호를 통해 프로세스가 파일에 대한 입출력을 수행할 수 있게 된다.

(4) filesize : 주어진 file descriptor 번호가 가리키는 파일의 크기를 반환한다.

(5) read : 주어진 file descriptor 번호가 가리키는 파일에서 주어진 size만큼 데이터를 읽어와 buffer에 저장한다. 만약 번호가 0이라면 표준 입력으로부터 읽어오고, 2 이상이라면 파일로부터 읽어온다.

(6) write : 주어진 file descriptor 번호가 가리키는 파일에 주어진 buffer에 저장된 내용을 쓴다. 만약 번호가 1이라면 표준 출력에 쓰고, 2 이상이라면 파일에 쓴다.

(7) seek : 주어진 file descriptor가 가리키고 있는 파일의 위치를 주어진 position으로 변경한다.

(8) tell : 주어진 file descriptor가 가리키고 있는 파일의 위치를 반환한다.

(9) close : 주어진 file descriptor와 file 구조체에 대한 자원을 반환하고 file descriptor 번호를 사용 가능한 상태로 만든다.

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

(1) Lock을 사용하면 특정 프로세스가 특정 자원에 대한 exclusive access를 얻고, 나머지 프로세스는 access 할 수 없게 되면서 race contidion을 피할 수 있게 된다. 이번 프로젝트에서는 각각의 파일에 대한 lock이 아니라 파일 입출력 작업에 대한 lock을 설정할 수 있고, 이 또한 데이터 무결성이 보장되는 방법이다.

(2) Semaphore를 사용하면 특정 프로세스 하나만 exclusive access를 얻는 것이 아니라 여러 프로세스가 동시에 파일에 접근하는 것을 허용할 수 있다. 이는 semaphore가 counter mechanism을 사용하기 때문이다. 예를 들어 semaphore counter를 특정 수 n으로 초기화 하여 프로세스가 자원에 읽기 접근할 때 counter를 1 감소시키는 방식을 통해 n개의 프로세스가 동시에 파일 읽기를 수행할 수 있게 된다. 하지만 쓰기 작업의 경우에는 exclusive access가 필요하므로 counter를 0으로 설정하여 작업이 끝난 후 되돌린다.

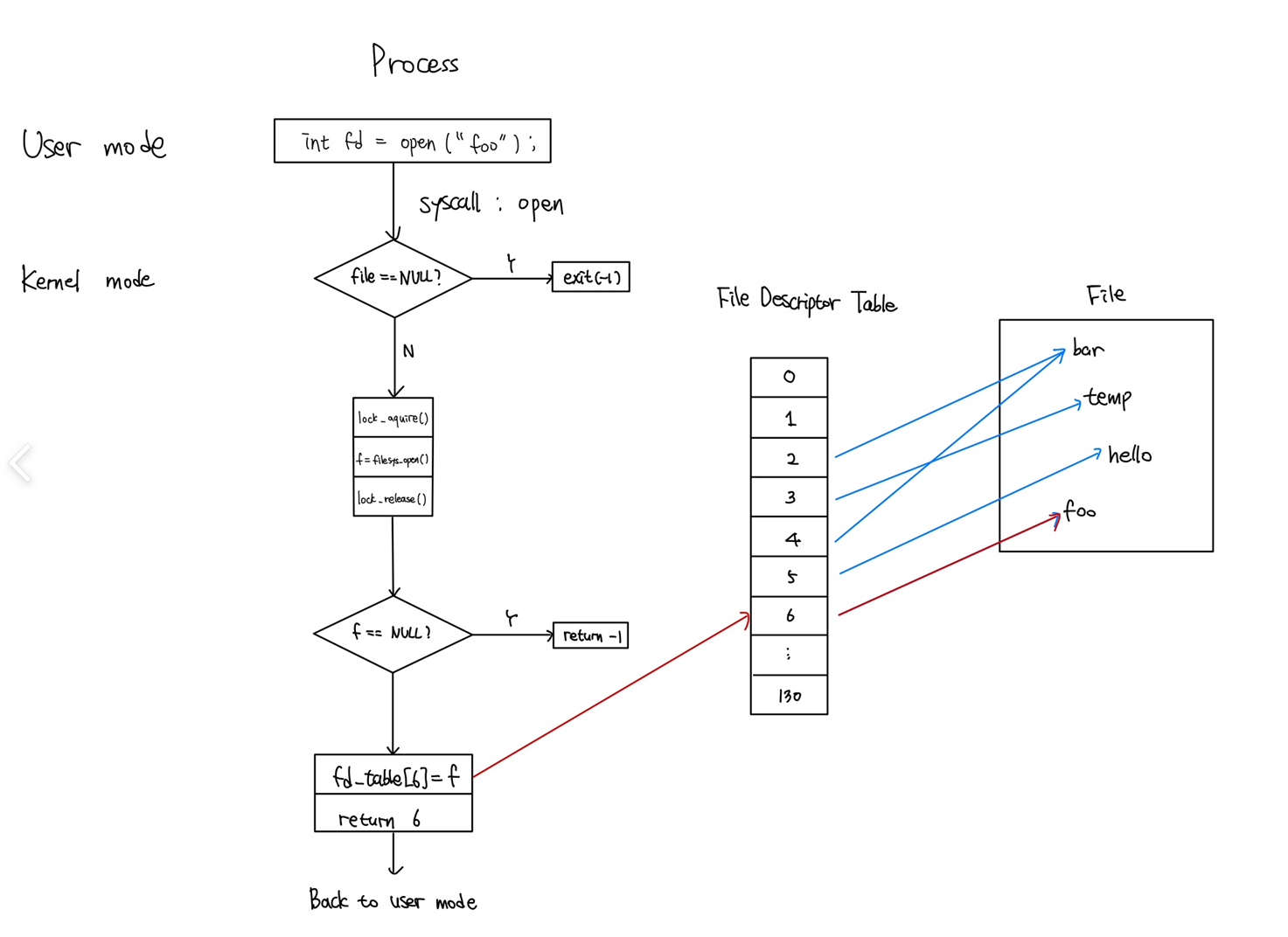
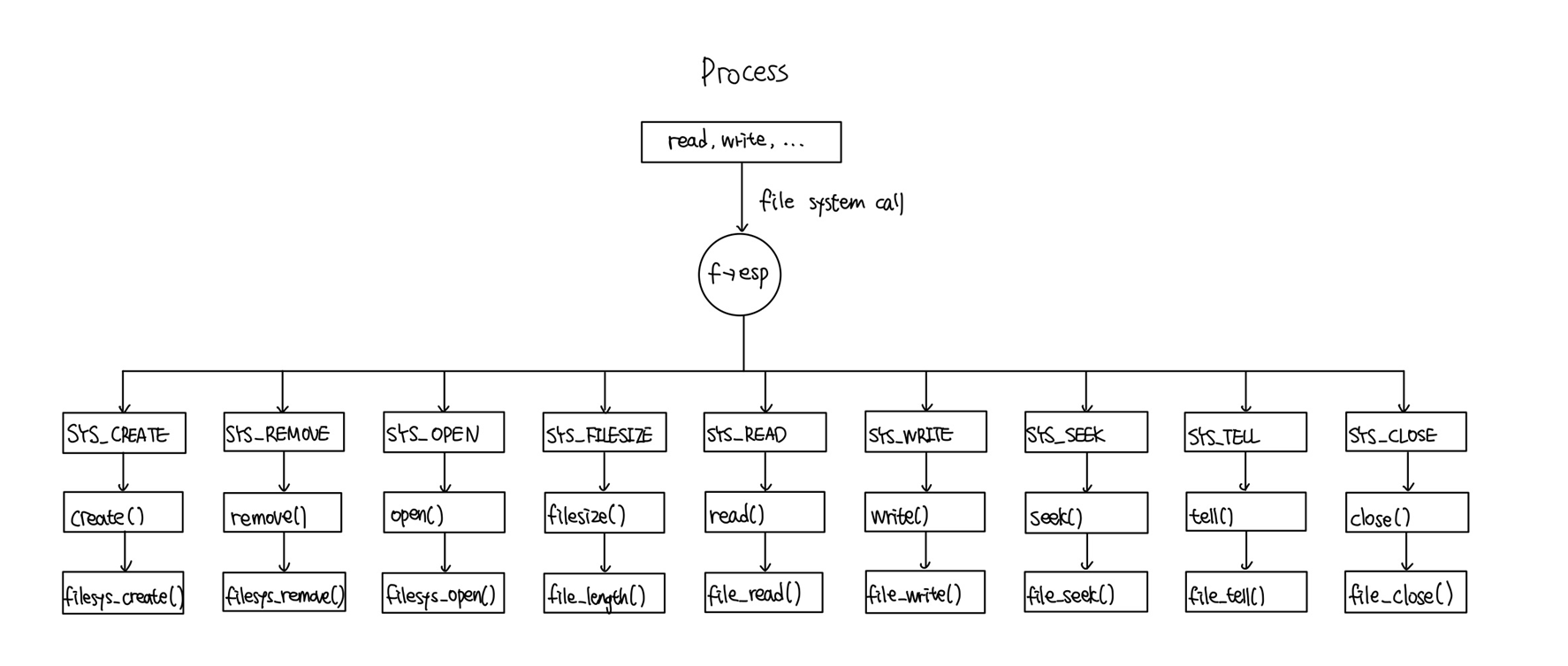
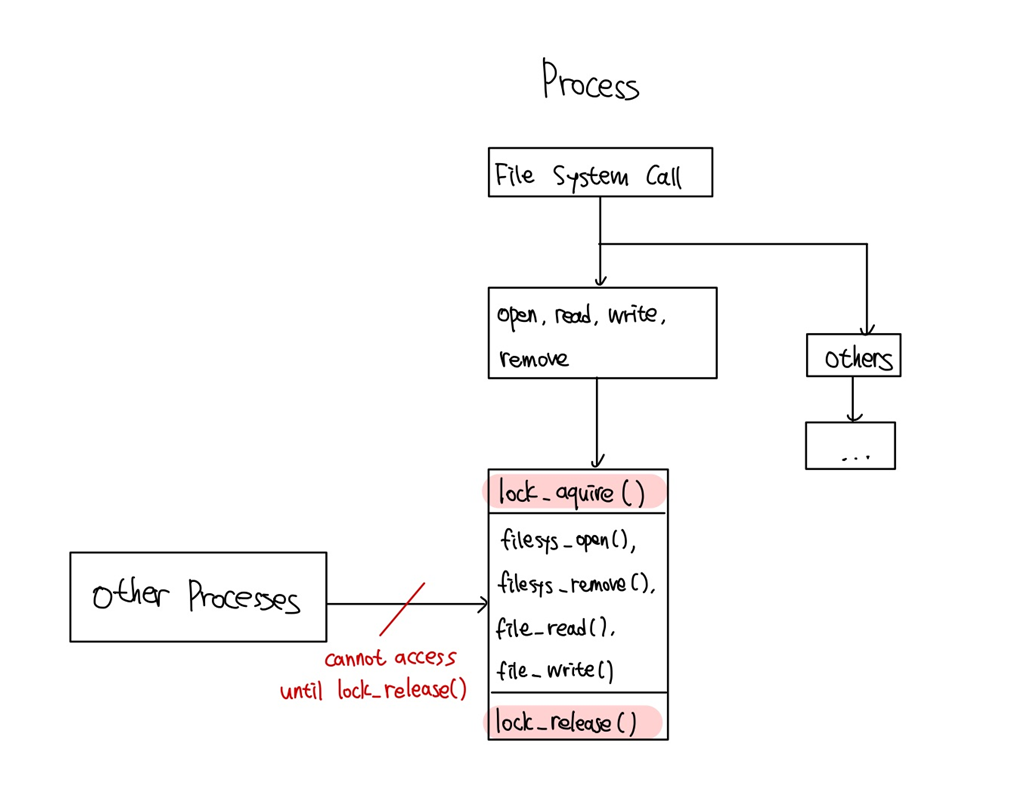
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

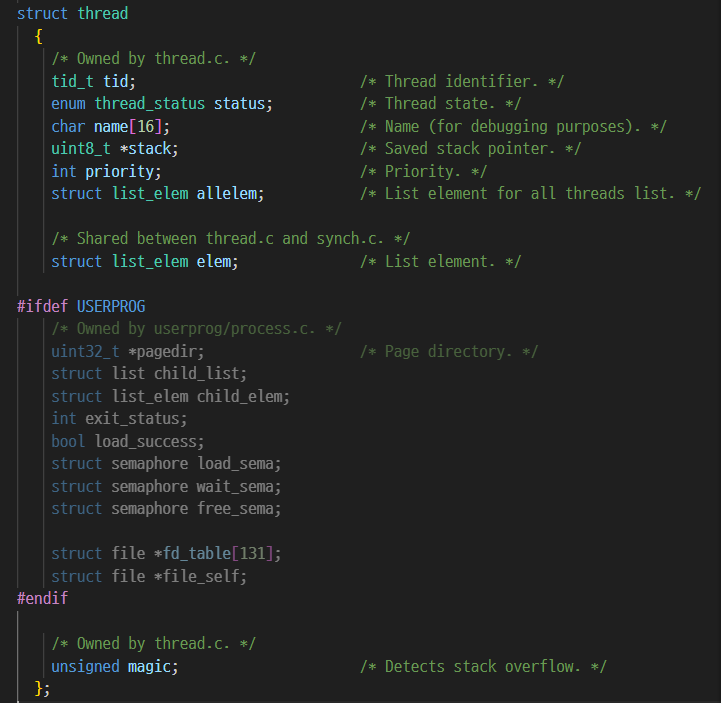
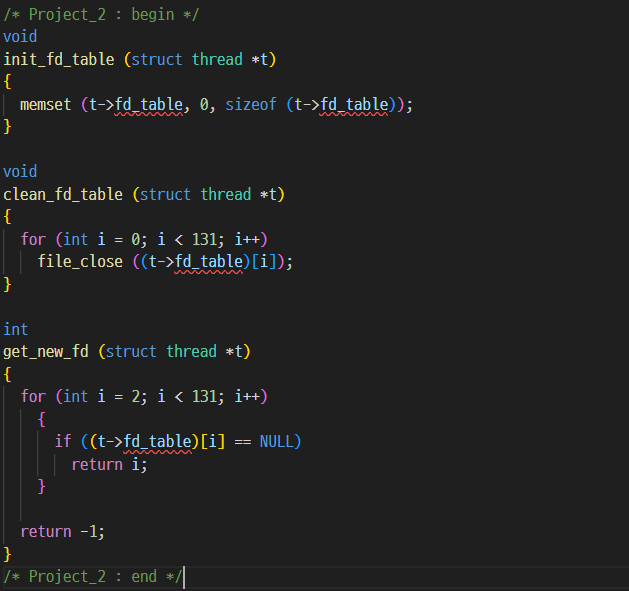
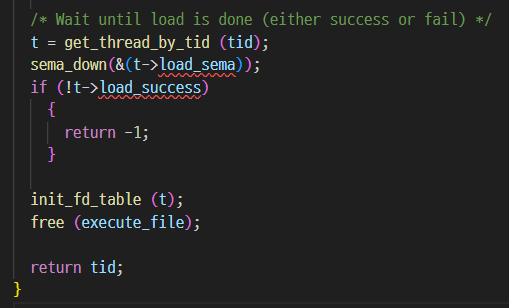
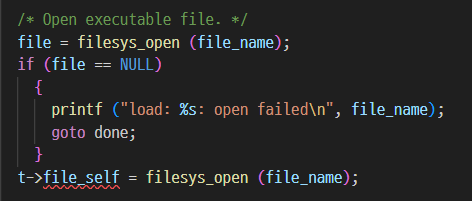
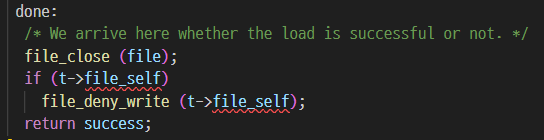
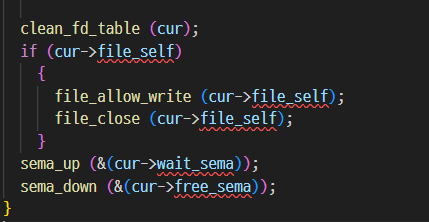
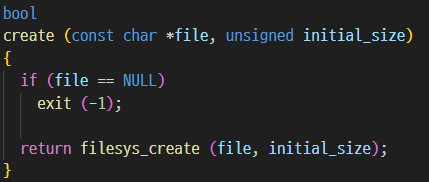
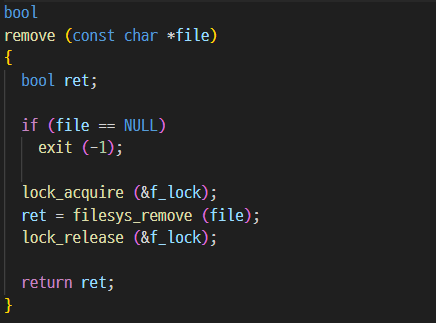
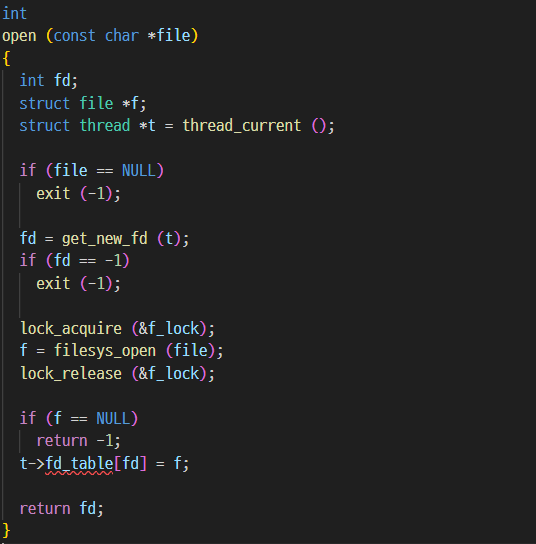
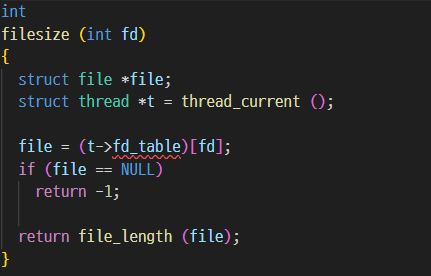
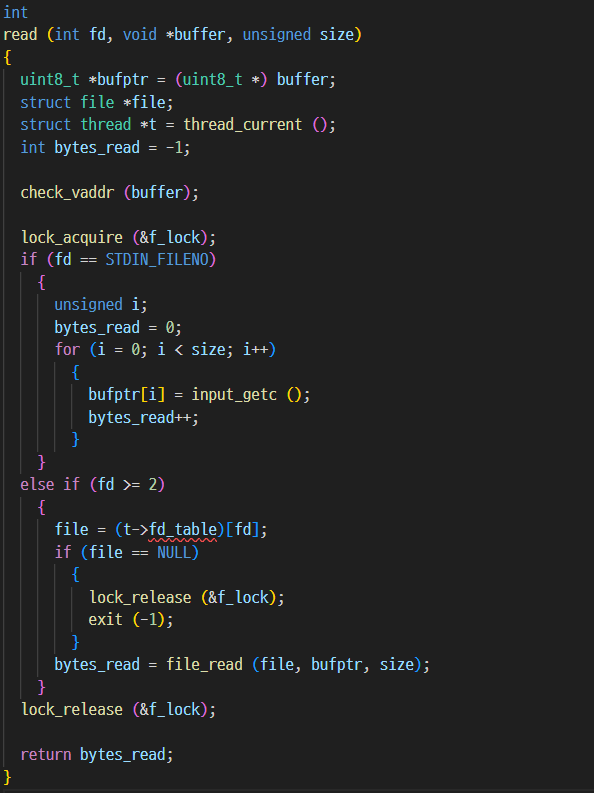
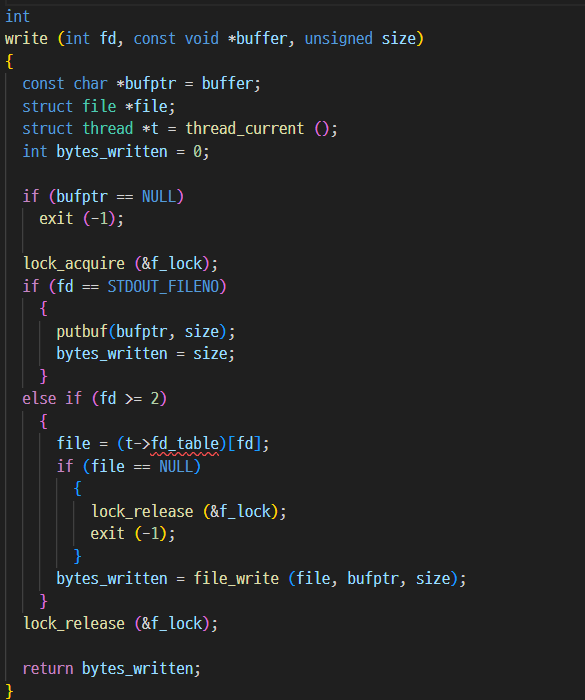
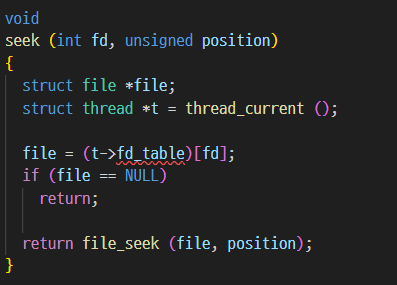
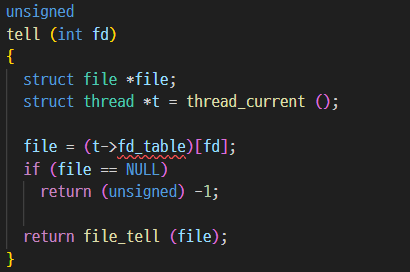
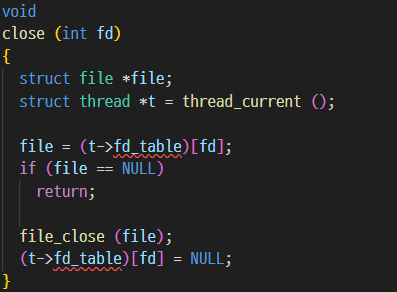
* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10/28 : File system call 구현, synchronization 구현

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드  
    - src/userprog/syscall.c  
    - src/userprog/process.c  
    - src/threads/thread.h
  + 수정하거나 추가해야 하는 자료구조  
    1. src/thread/thread.h : struct thread  
    Thread 구조체 내부에 file descriptor 구현 및 실행 중인 파일에 대한 denying write 구현을 위한 변수를 추가한다.
  + 수정하거나 추가해야 하는 함수  
    1. src/userprog/syscall.c : system call handling functions  
    - bool create()  
    - bool remove()  
    - int open()  
    - int filesize()  
    - int read() => 기존 표준입력에 대한 구현과 더불어 파일 입력에 대한 구현 추가  
    - int write() => 기존 표준출력에 대한 구현과 더불어 파일 출력에 대한 구현 추가  
    - void seek()  
    - unsigned tell()  
    - void close()  
    2. src/userprog/process.c : file descriptor managing funcitons  
    - void init\_fd() => 새로 생성된 프로세스에 대한 file descriptor table 초기화  
    - void clean\_fd() => 종료되는 프로세스에 대한 file descriptor table에 남아있는 자원 회수  
    - int get\_new\_fd() => 새로운 파일을 open할 때 부여받는 file descriptor 번호를 반환  
    - bool load() => 실행 파일 load 후 struct thread의 file\_self 변수에 자기 자신에 대한 file 구조체 할당, 쓰기 금지 설정 추가  
    - tid\_t process\_execute() => 프로세스 정상 실행 후 init\_fd() 호출 추가  
    - void process\_exit() => 프로세스 종료될 때 deinit\_fd() 호출, 자기 자신에 대한 쓰기 금지 해제 추가

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성  
  1. File descriptor  
  ****  
    
  2. System Calls  
  ****  
    
  3. Synchronization in File System  
  ****
  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

1. src/threads/thread.h  
Thread 구조체에 file descriptor table과 실행 파일에 대한 file 구조체 변수를 추가하였다.  
****  
- struct file \*fd\_table[131] : file 구조체 포인터 배열, file descriptor table  
- struct file \*file\_self : 프로세스의 실행 파일에 정보를 담는 file 구조체  
  
2. src/userprog/process.c  
각 thread의 file descriptor에 대한 동작을 수행하는 함수를 정의하였다.  
  
- void init\_fd\_table() => thread 구조체의 fd table을 0으로 초기화한다.  
- void clean\_fd\_table() => thread 구조체의 fd table에 아직 열려있는 file descriptor들에 대한 자원을 모두 회수한다.  
- int get\_new\_fd() => thread 구조체의 fd table에서 가장 낮고 닫혀있는 번호를 찾아 반환한다. Table이 꽉차있다면 -1를 반환한다.  
  
  
- tid\_t process\_execute() => load가 완료되면 init\_fd\_table 호출.  
  
  
  
- bool load() => thread 구조체의 file\_self에 실행 파일에 대한 내용 할당. Load 완료되면 실행 파일에 대해 file\_deny\_write 호출.  
  
  
- void process\_exit() => 프로세스 종료될 때 clean\_fd\_table 호출. 실행 파일에 대한 file\_allow\_write 호출 및 file\_close.  
  
3. src/userprog/syscall.c  
File system call implementation에 대한 함수 구현 내용이다.  
****  
- bool create() => filesys\_create 호출, file이 NULL인 경우에 대한 예외 처리  
****  
- bool remove() => filesys\_remove 호출, 파일 입출력에 대한 lock 사용, file이 NULL인 경우에 대한 예외 처리  
****  
- int open() => get\_new\_fd()를 통해 fd 할당받고 filesys\_open을 통해 얻은 file 구조체를 fd table에 저장. 파일 입출력에 대한 lock 사용, 각종 예외처리  
****  
- int filesize() => fd값을 통해 fd table에서 file 구조체 찾고 file\_length() 호출하며 반환. File이 NULL인 경우에 대한 예외처리  
****  
- int read() => 파일 입력에 대한 구현 추가. File\_read 호출하여 buffer의 내용 입력. Lock 사용. File이 NULL인 경우에 대한 예외처리  
****  
- int write() => 파일 출력에 대한 구현 추가. File\_write 호출하여 buffer에 내용 저장. Lock 사용. File이 NULL인 경우에 대한 예외처리.  
****  
- void seek() => file\_seek 호출. File이 NULL인 경우에 대한 예외처리.  
****  
- unsigned tell() => file\_tell 호출. File이 NULL인 경우에 대한 예외처리.  
****  
- void close() => file\_close 호출. Fd table에 해당 자리를 NULL로 설정. File이 NULL인 경우에 대한 예외처리.

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

처음 file descriptor를 구현하는 과정에서 한 번 사용하고 버려진 descriptor 번호를 가져오는 것에 대해 큐를 사용하는 방법을 채택하였다. 구현 결과 의도한 대로 동작이 되었지만 test 과정에서 multi-oom에 대한 결과가 FAIL이 나왔다. 왜 FAIL인지 분석하다가 fd table과 더불어 큐까지 사용하는 것이 많은 프로세스를 생성하는 과정에서 메모리 부족 현상을 일으키는 것이라 추측하였다. 그래서 큐를 사용하는 것 대신 fd를 할당할 때 반복문을 통해 적절한 fd를 찾는 방식으로 다시 구현하였더니 test를 정상적으로 통과하였다.

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