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

담당 교수 / 분반 :

이름 / 학번 : 이상윤 20171664

개발 기간 : 3주

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트2 에서는 프로젝트1 에서 구현한 프로세스의 동기적 실행을 더욱 정교하게 실행될 수 있도록 하고, 루트 디렉토리의 한정되지만 파일 시스템을 완성한다. 세마포어와 락을 사용하여 여러개의 프로세스 환경에서도 적절히 동작하게 한다. create, remove, open, close, filesize, read, write, seek, tell 의 시스템 콜을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

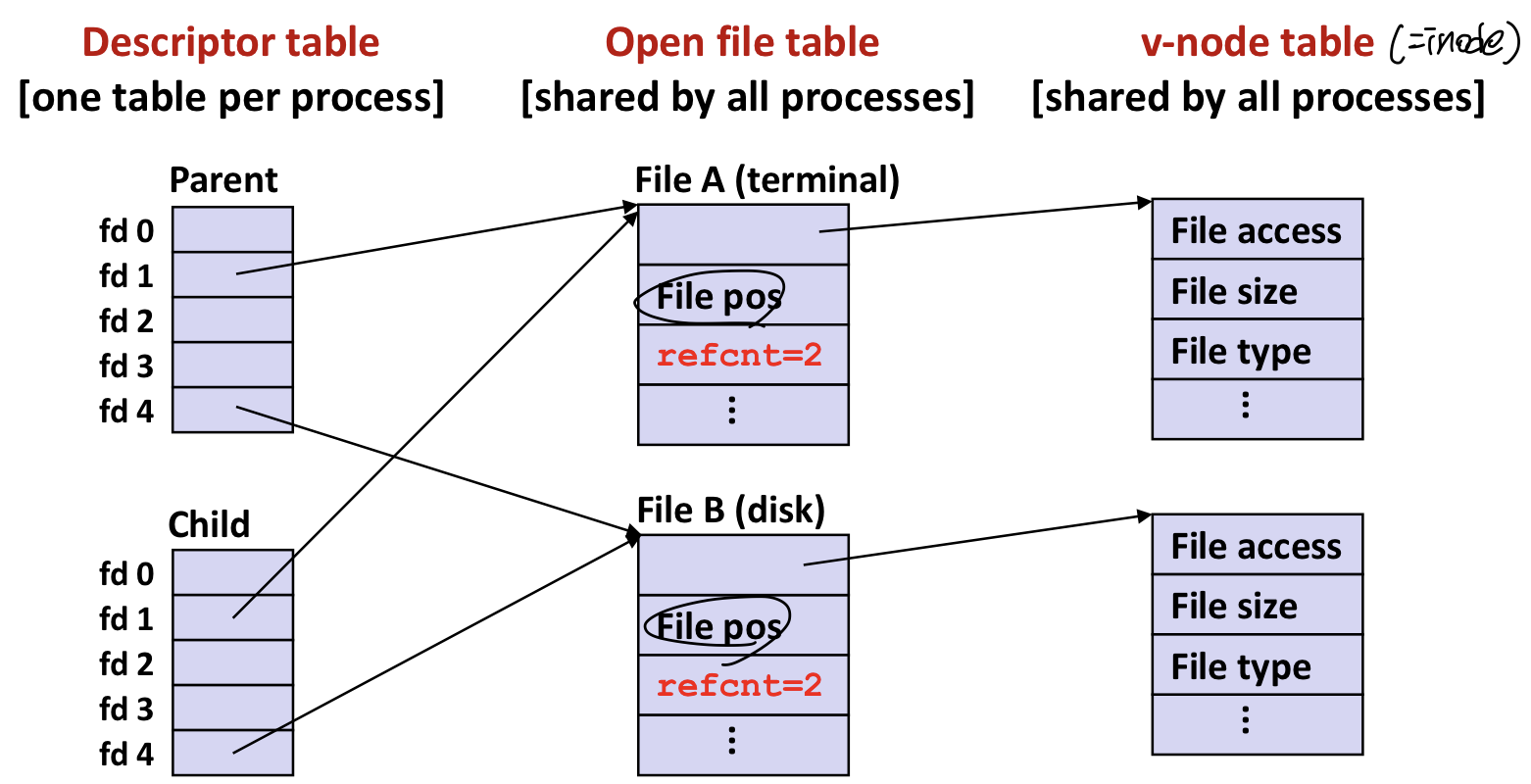
1. File Descriptor

파일 디스크립터(fd) 테이블은 프로세스마다 가지고 있으며, 이를 이용하여 특정 파일에 접근이 가능하다. 즉, 각각의 프로세스가 파일의 inode에 접근하여 sector로 부터 데이터를 읽고 쓰기 위해서는 이 fd를 통해서 가능하다. PCB라고 할 수 있는 thread.h의 thread 구조체에 정의한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

이번 프로젝트에서 구현하는 시스템 콜에는 create, remove, open, close, filesize, read, write, seek, tell 이 있다. 파일 시스템이 파일을 만들거나 삭제하거나, 열거나 닫거나, 파일의 크기를 알거나, 파일을 읽고 쓰거나, 파일에서 읽고 있는 위치를 바꾸거나 알려주는 역할을 수행할 수 있도록 한다.

3. Synchronization in Filesystem

여러 프로세스가

이와 같이 fd 로 file table을 참조하고, 그 file table은 inode를 참조하는 식으로 파일 시스템은 이루어 져있다. 이 때 위 그림과는 다르지만, File B table 도 첫번째 inode 를 참조한다고 가정해보자. Parent process 가 inode를 read 하는 와중에 context-switch 가 발생하여 Child process 가 inode를 write 한다고 생각해보자. 이 때, synchronization 이 적절히 이루어져있지 않다면 커널은 사용자의 요청에 의해 예측할 수 없는 상황에 처하게 될 것이다. 위 예시는 대표적인 readers-writers problem 이다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

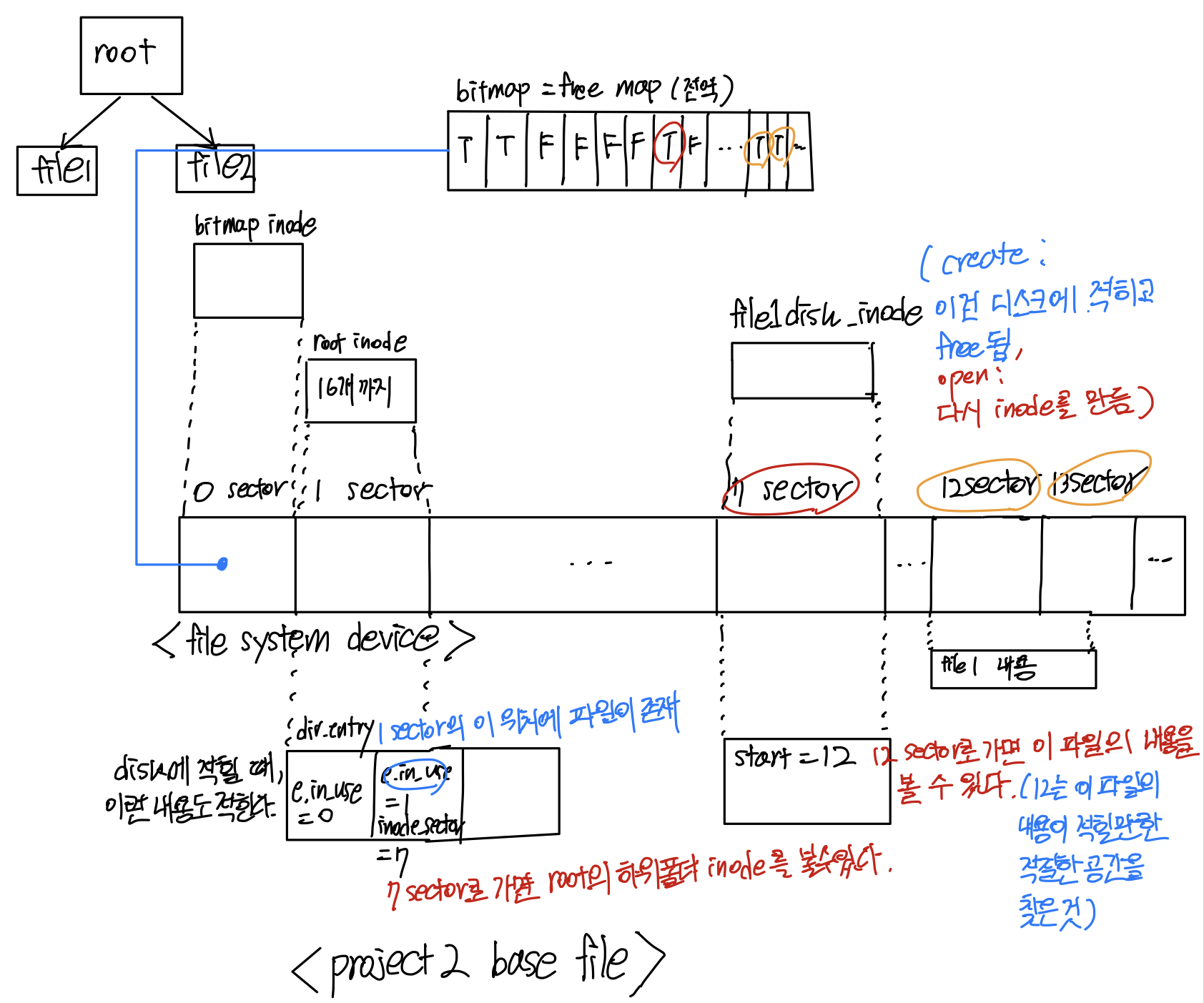
1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

fd table 은 프로세스 마다 PCB에 구현되어 있어야 하므로, thread.h 의 thread 구조체에 배열로

struct file\* fd[128];

과 같이 구현하였다. file 구조체는 file table 을 의미하고, 그 file table 을 가르키는 배열인 fd 를 만든다. 길이는 핀토스 매뉴얼에서 프로세스 당 128 개의 파일을 열 수 있다고 명시된 대로 128로 정하였다.

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. bool create(const char\* file, unsigned initial\_size) : file 이라는 이름의 파일을 만든다. 처음의 크기는 initail\_size 이다. 이 작업이 open으로 이어지지는 않는다. 정상적으로 만들어졌을 경우에는 true, 아닌 경우에는 false 를 반환한다.
2. bool remove(const char\* file) : file 은 파일이 open 된 상황이든 아니든 간에 지워진다. 만약 file 이 open된 상황이라고 해서 이 함수를 호출한다고 close 함수가 자동으로 호출되진 않는다. 그리고 이 함수를 호출한다고 해서 저장공간 상의 file 이 바로 지워지지도 않는다.
3. int open(const char\* file) : file 이란 이름의 파일을 연다. 열리지 않았다면 -1, 열렸다면 fd 를 반환한다. 0, 1, 2는 표준 입출력, 에러로서 open의 반환값이 될 수 없다. 하나의 파일이 두번 이상 열렸다면 두개의 서로 다른 fd 를 가진다. 그리고 이 서로 다른 fd 는 독립적인 file table 을 가지지만, 서로 같은 파일의 inode 를 참조한다.
4. void close (int fd) : fd 를 닫는다. 프로세스를 종료하면 모두 이 함수를 호출하는 것 처럼 닫힌다. 그리고 fd 가 가르키는 파일이 remove 함수의 호출로 인해 지워져도 되는 조건을 충족 하였을 때 파일의 실질적인 삭제가 이루어 진다.
5. int read(int fd, void\* buffer, unsigned size) : fd 로 open된 파일에서 buffer로 size 만큼을 읽어들인다. 파일을 읽을 수 없는 경우에는 -1, 그 외의 경우에는 실제로 읽은 bytes의 수를 반환한다. Fd 가 0 인 경우는 input\_getc()를 이용하여 키보드로 부터 읽어들인다.
6. int write (int fd, const void \*buffer, unsigned size) : fd 가 가르키는 파일에 buffer을 쓴다. buffer 는 size bytes 의 크기이다. 파일에 쓸 수 없는 경우에는 -1, 그 외의 경우에는 실제로 쓴 bytes의 수를 반환한다. fd가 1이라면 콘솔창에서 putbuf()를 이용하여 입력을 진행한다.
7. int filesize(int fd) : fd 로 여는 파일의 크기를 bytes 로 반환한다.
8. void seek (int fd, unsigned position) : open 된 파일을 가르키는 fd 에서 다음 byte를 읽고 쓸 위치를 position 으로 바꾼다.
9. unsigned tell (int fd) : open 된 파일을 가르키는 fd 에서 다음 byte를 읽고 쓸 위치를 반환한다.

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

우선 lock 은 semaphore 와 다르게 1 이상의 값을 가질 수 없다. 그리고 한번에 하나의 프로세스에 의해서만 소유되어 진다. 또한 lock 을 거는 lock\_acquire() 과 lock 을 푸는 lock\_release() 는 전부 한 프로세스에 의해서만 이루어 져야 한다.

그리고 현재 base file system 은 위의 그림과 같이 나타낼 수 있다. 일단 bitmap 을 통해서 file system device 에서 공간을 할당되어 있는지의 여부와 같은 사실을 알 수 있다. 즉, bitmap 조작만은 항상 lock이 걸려져야 한다. free-map.c 의 free\_map\_allocate(), free\_map\_release() 함수는 free-map.h 에 선언되어 있는 bitmap\_lock 을 통해 시작과 끝에 lock을 잡고 풀어주었다.

또한 inode 는 동기화되서 만들어져야한다. 똑같은 파일이나 디렉토리를 나타내는 inode 는 하나만 생성되어야 하는데 여러 inode 가 생성된다면 메모리 누수가 일어날 가능성이 있다.

static struct lock\*\* inode\_lock;

을 inode.h 에 선언하여, 동일한 inode 에 대한 open과 close는 동기화되어서 이루어지도록 한다. inode\_lock[sector] 와 같은 방법으로 sector 를 참조하는 inode 에 대한 lock 을 알 수 있다.

dir\_add(), dir\_remove() 와 같은 함수는 디렉토리를 나타내는 inode 를 참조해 sector 의 데이터를 수정하는 함수이다. 그리고 dir\_lookup() 은 dir 을 조회하는 함수이다. 또 inode\_deny\_write(), inode\_allow\_write(), file\_write() 는 inode 를 수정하는 함수이고, file\_read() 는 inode 를 조회하는 함수이다. 위 예시 이외에도 수많은 reader-writer problem 을 해결하기 위해, inode 구조체에도

int read\_cnt;

struct lock w;

struct lock inode\_readcnt\_mutex;

이와 같이 lock 구조체를 선언한다.

이외에도 open\_inodes 리스트가 제대로 삭제되기 위한 락이 있다. 위에서 선언한 lock 들을 이용하여synchronization 을 file system 에서 구현하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2022-10-10~2022-10-10 : 프로젝트1의 multi-processing 코드 수정, file descriptor table 구현

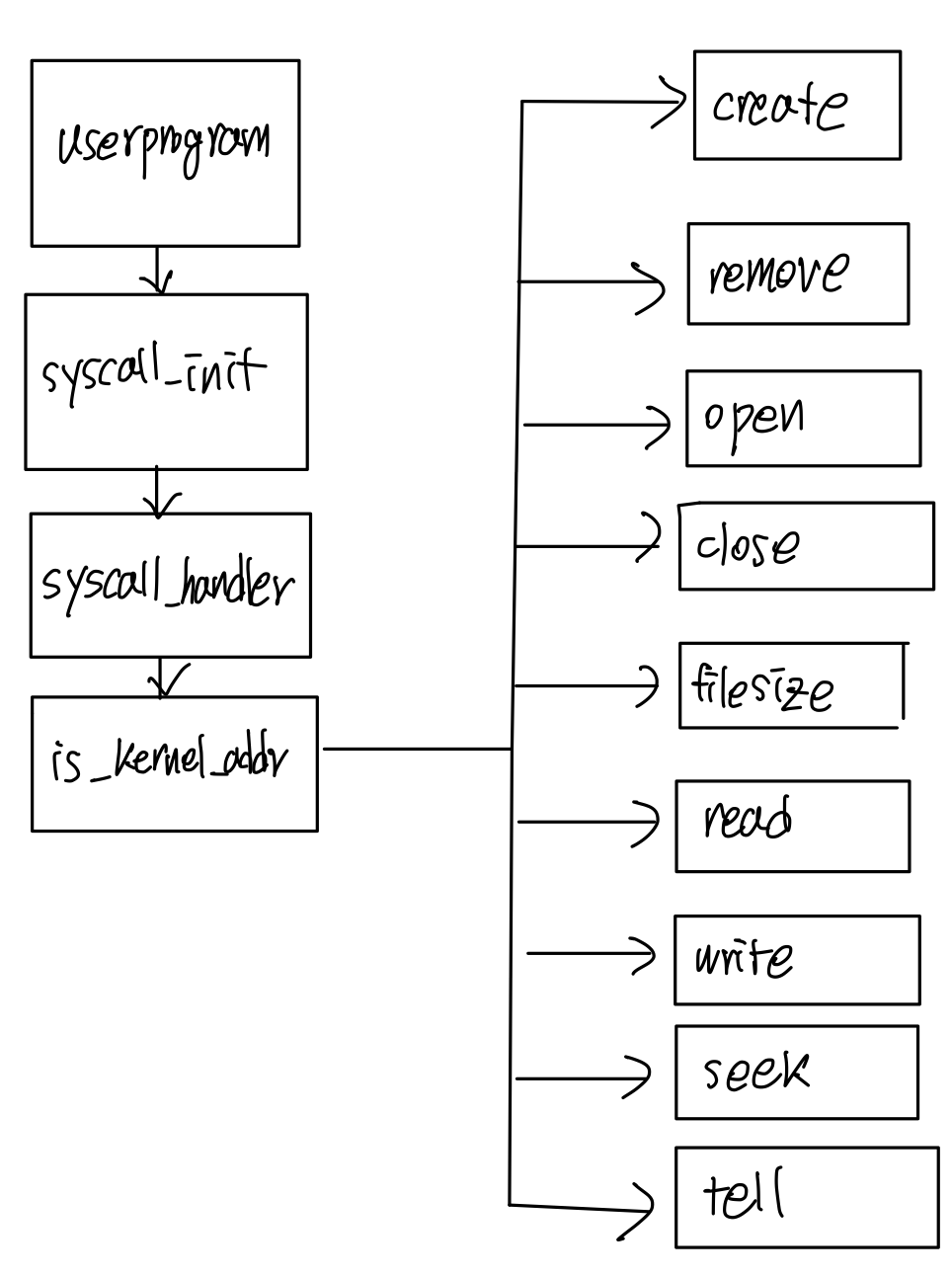
2022-10-11~2022-10-24 : semaphore, lock으로 filesystem 수정(synchronization in filesystem)

2022-10-25~2022-10-26 : system call 구현, 오류 수정, 보고서 작성

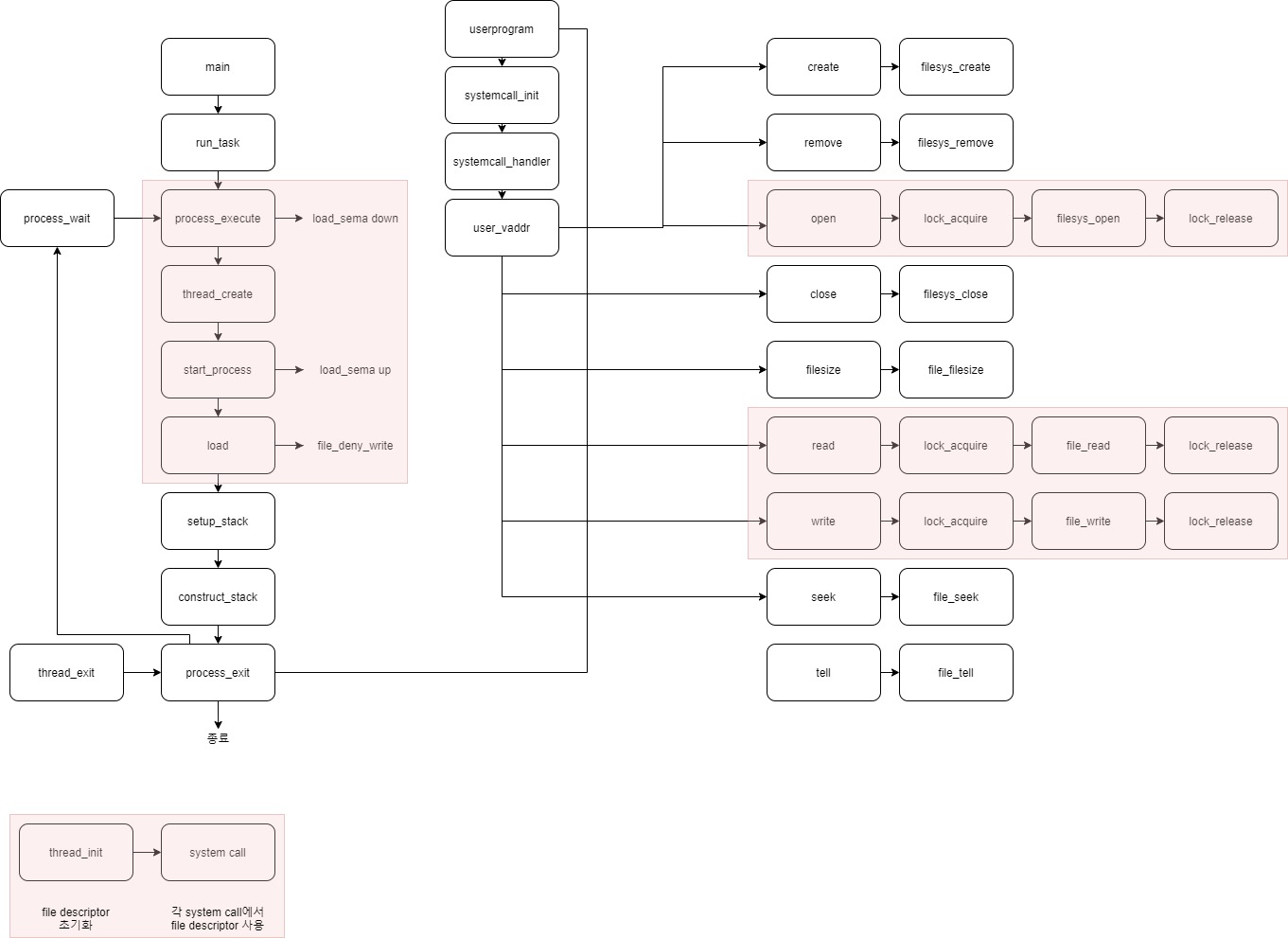
* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명

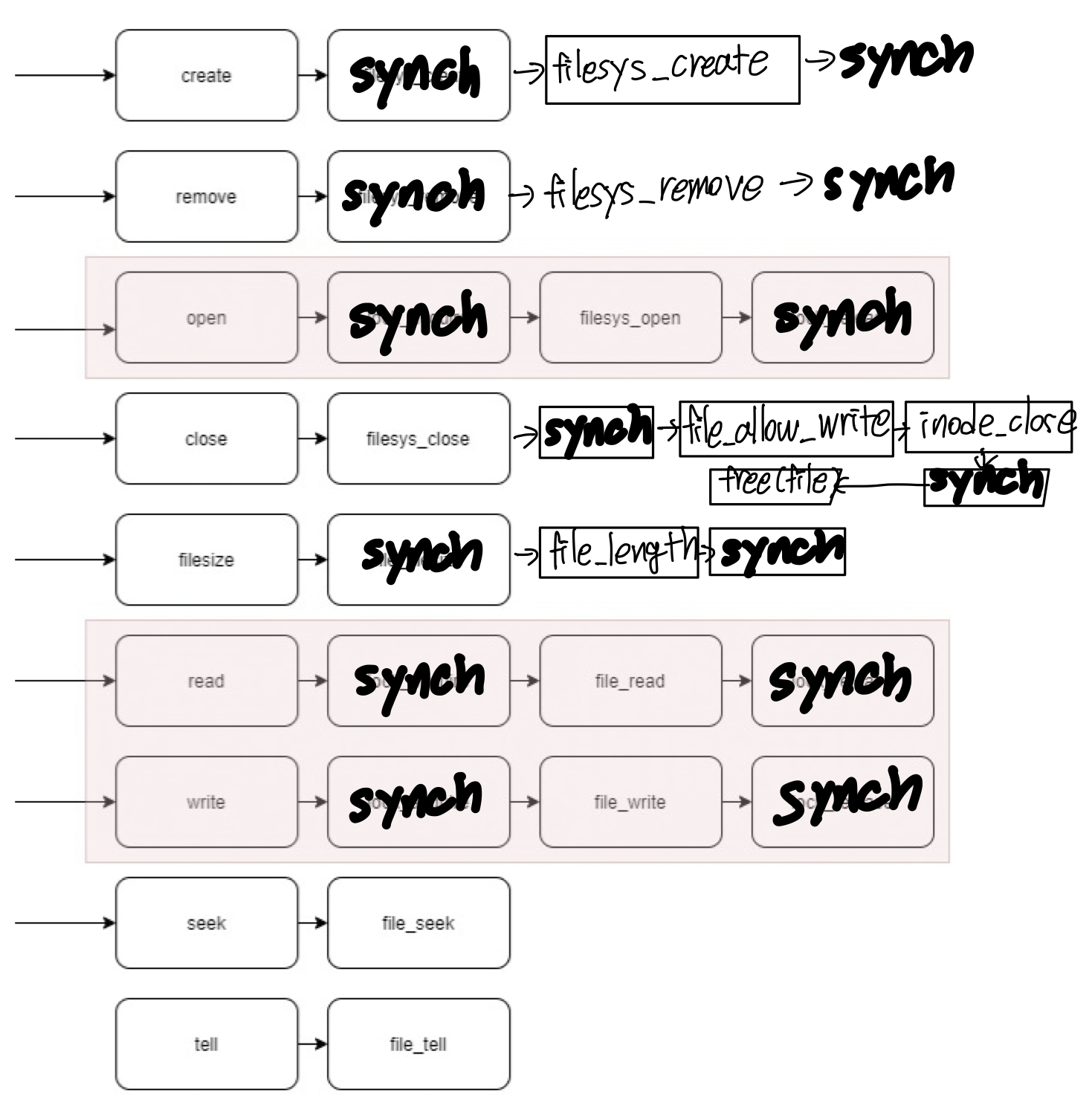
우선 PCB 에서 file descriptor table 을 추가하기 위해 thread.h 의 thread 구조체에 fd 를 선언한다. 이는 init\_thread() 함수에서 NULL 로 전부 초기화된다. open system call 에서는 이를 초기화하고, close, exit system call 에서는 이를 다시 NULL 로 되돌릴 것이다.

그리고 위에서 서술한 lock 구조체와 변수들을 추가한다.

syscall.c 의 syscall\_handler() 에서 switch-case 문에 구현할 system call 이 들어온 경우를 추가한다. 이 때 syscall\_handler() 에서 호출할 함수의 원형은 핀토스 메뉴얼을 참고하였다. exit() 함수는 모든 fd 에 대해 close() 을 호출하도록 수정한다. open() 함수에서 핀토스는 실행중인 파일의 수정을 원하지 않으므로 현재 프로세스를 연 경우에는 이 파일의 수정을 file\_deny\_write() 를 통하여 막아둔다. close() 함수는 file\_close() 함수를 호출하는데, 이때 file\_allow\_write() 를 통해 다시 수정 가능하게끔 하도록 한다. read() 함수는 file\_read() 함수를 호출하는데, 여기서 inode 의 구조체에서 선언한 lock 들을 사용하여 reader-writer problem 을 해결한다. write() 함수는 file\_write() 함수를 호출하는데, 여기서 마찬가지로 lock 을 사용하여 reader-writer problem 을 해결한다. create() 함수는 filesys\_create() 함수를 호출하는데, 이 함수가 호출되는 동안 inode\_open(), free\_map\_allocate(), inode\_create(), dir\_add(), free\_map\_release(), inode\_close() 등등 file system device 과 bitmap 을 수정하는 특히나 더 많은 함수들이 호출된다. 이 함수들이 멀티 프로세스 환경에서 정상적으로 작동할 수 있도록 lock 을 적절히 pintos/src/filesys 에 위치한 파일들에 추가한다. 마찬가지로 inode 를 조회하는 filesize(), remove() 함수는 lock 을 통해 정상적으로 동작하지 않을 가능성을 없앤다.

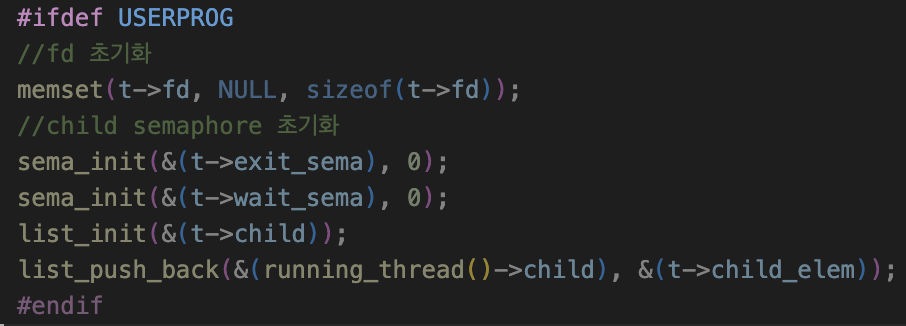
1. **연구 결과**
   1. **Flow Chart**

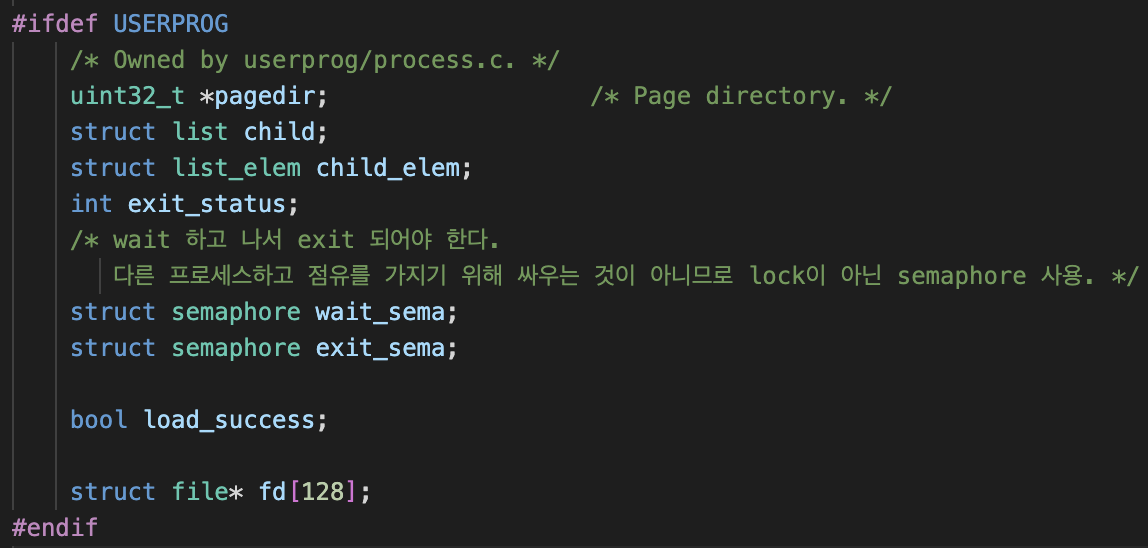
* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성

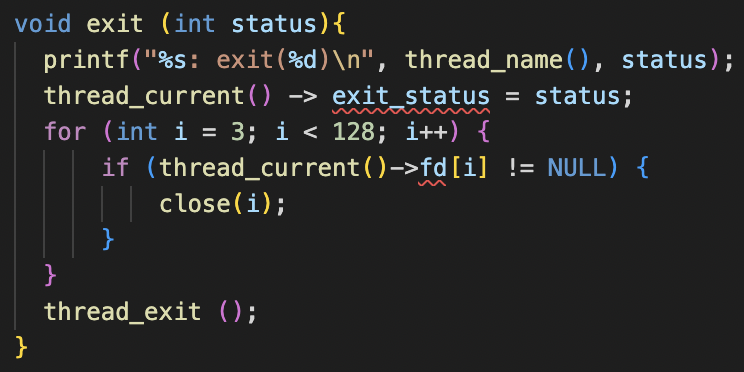
(synch는 해당 함수 호출에 대해 synchronization 이 잘 되어 있음을 의미한다)

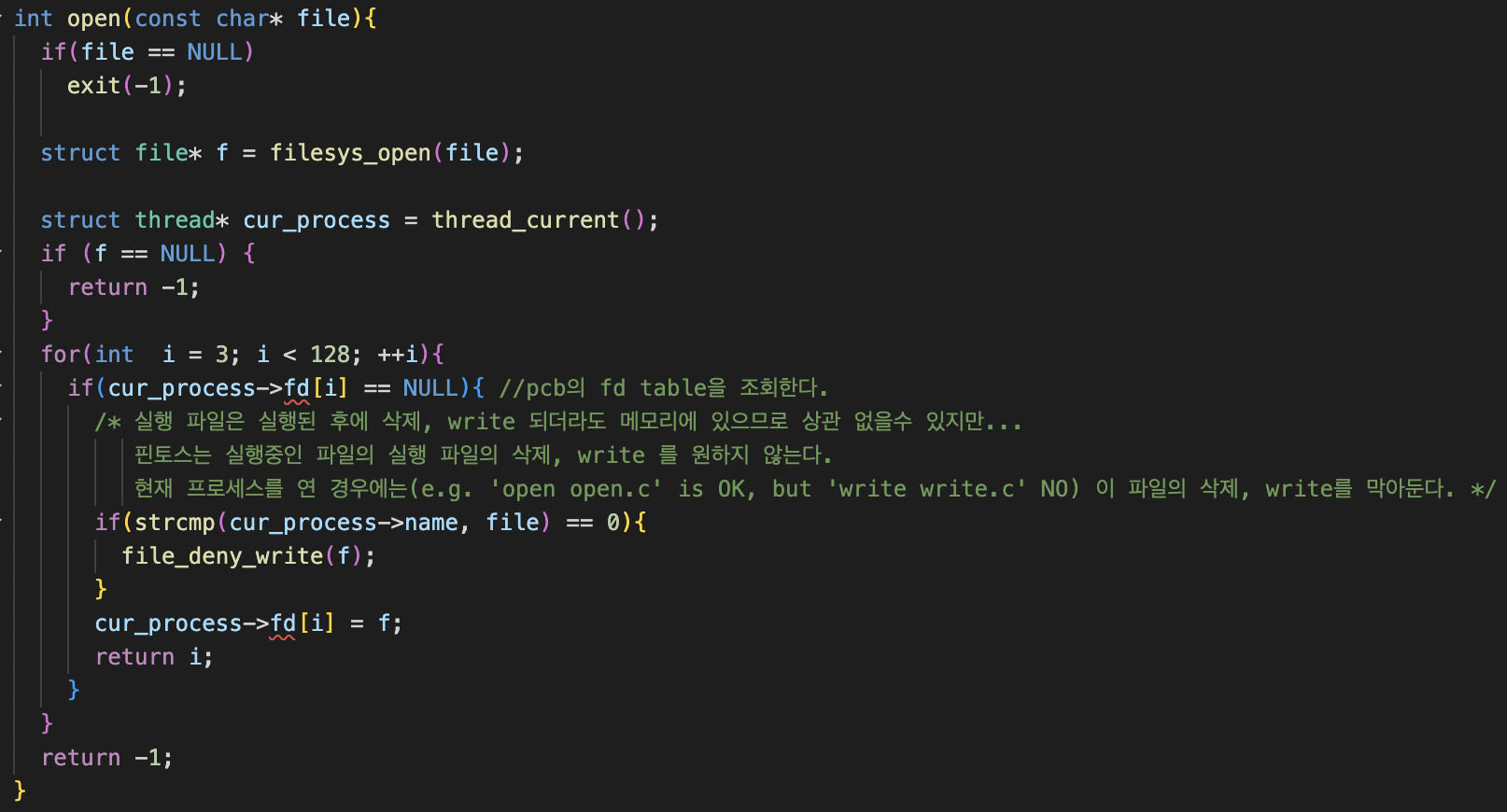
* 1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

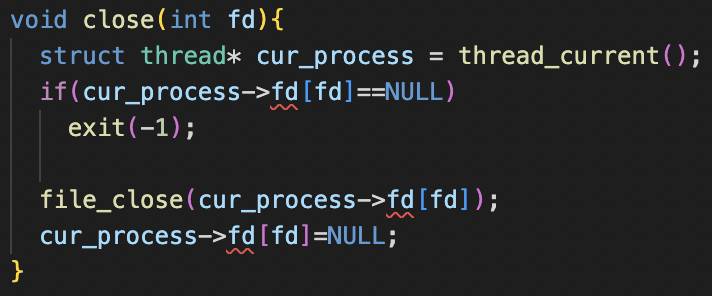
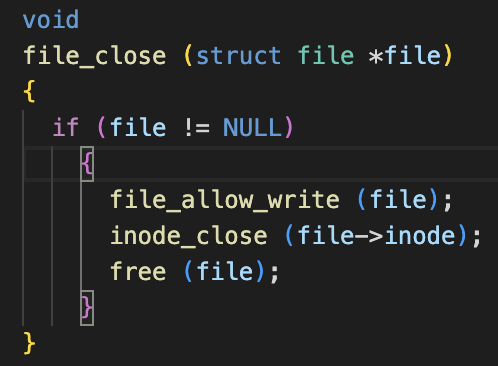
1. File descriptor

thread.h 의 thread 구조체에 fd 를 추가하였다.

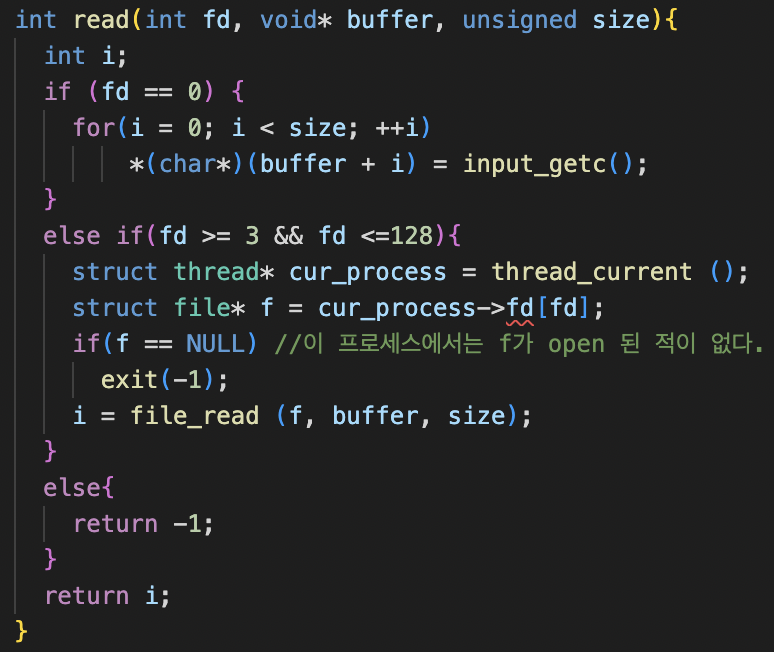
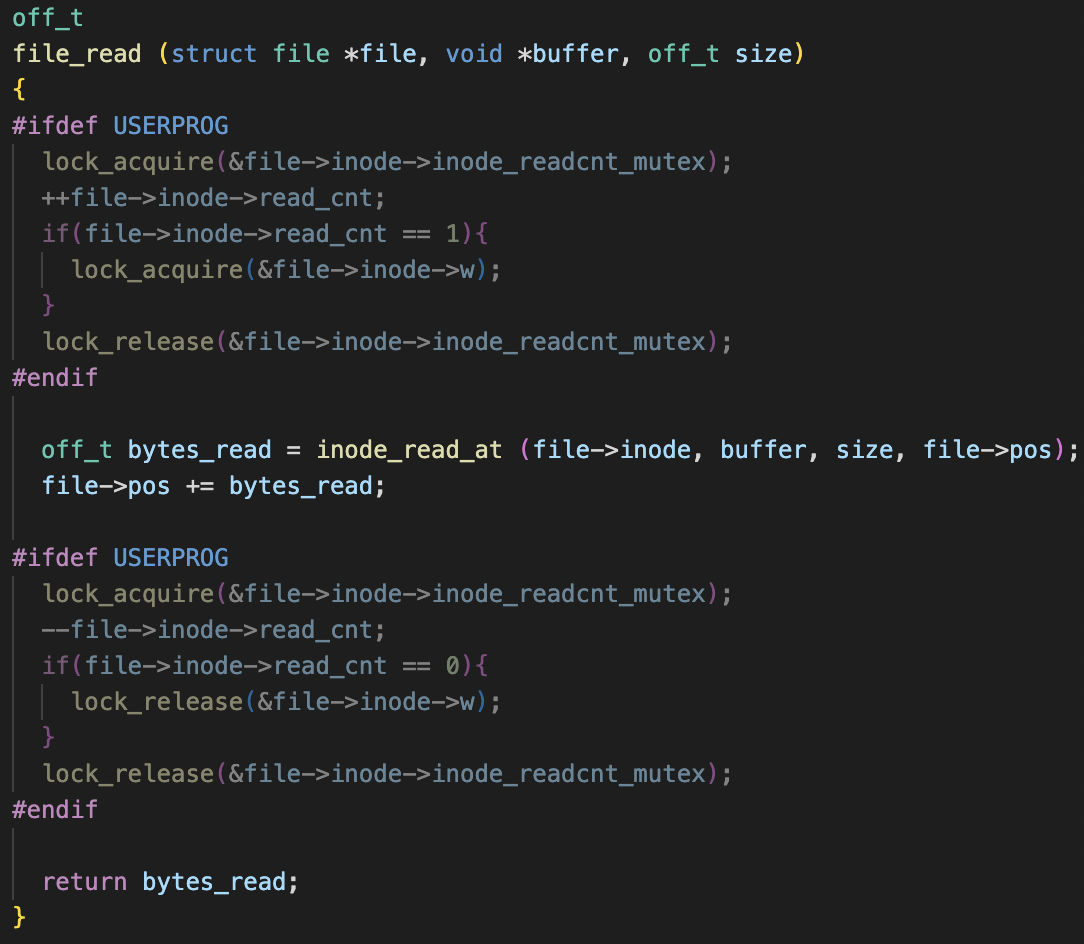
thread.c 의 init\_thread() 함수에서 이를 초기화 하였다.

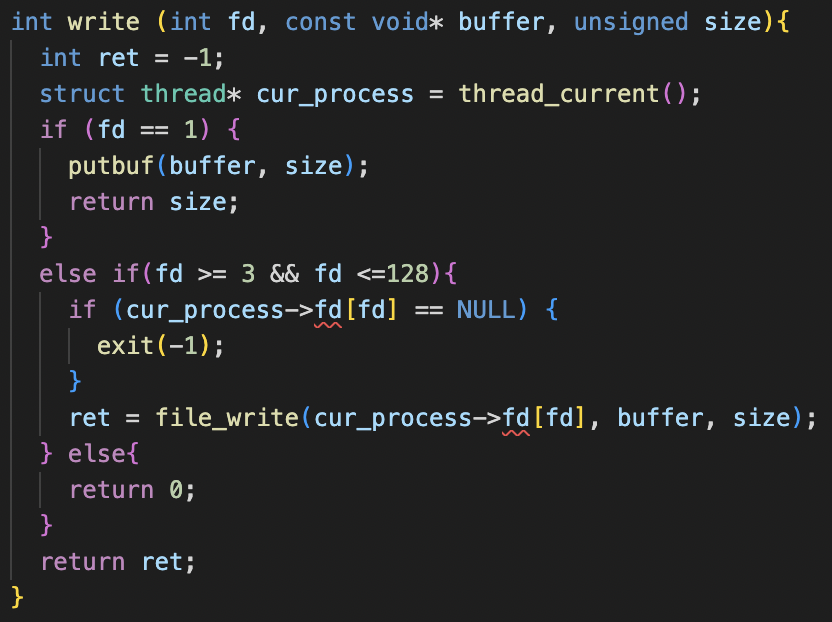
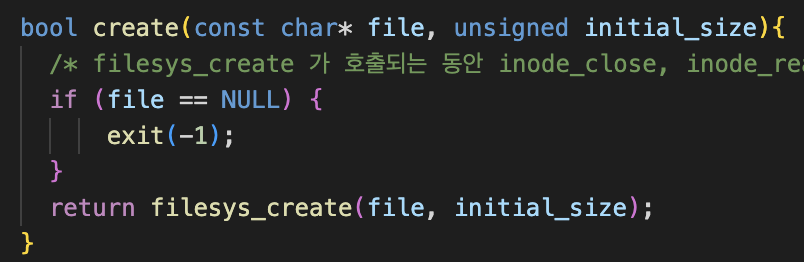
1. System call, Synchronization

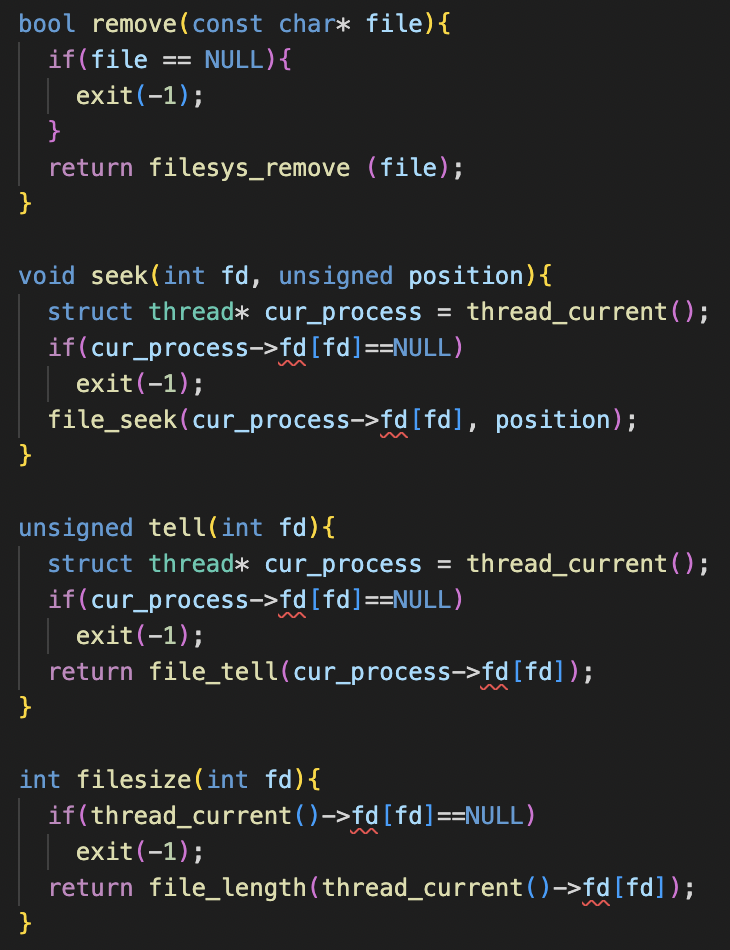
exit() 함수는 현재 프로세스의 fd를 모두 close 하고 종료하도록 수정한다.

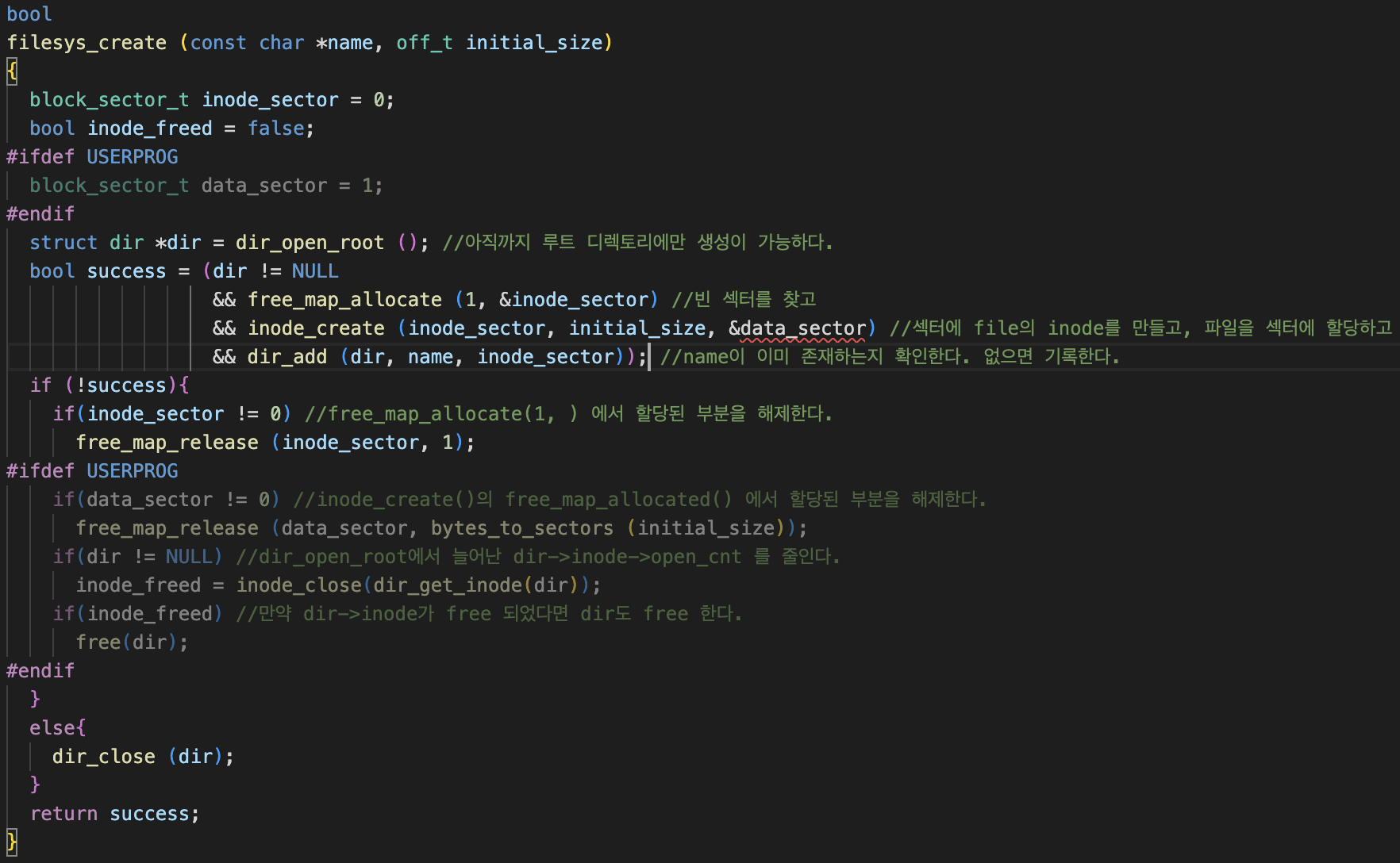
open() 함수에서 file\_deny\_write() 함수를 호출하는 이유는 주석과 같다. 그리고 기본적으로 파일이름이 NULL 이라면, fd table 에 적절한 공간이 없다면 -1 을 반환한다. file\_deny\_write() 는 inode\_deny \_write() 를 호출하고, inode\_deny\_write() 함수는 inode를 수정하는 writers 로서 lock 을 적절히 잡아준다.

close() 함수에서 fd[fd] 가 NULL 이라면 exit(-1) 으로 종료되게 하였다. file\_close() 함수의 file\_allow\_write() 는 inode\_allow\_write() 를 호출하고 이는 writers 로서 lock 을 적절히 잡아준다. 그리고 inode\_close() 는 동일 sector 에서 일어나는 inode\_open() 과 동시에 일어날 수 없으므로 inode->open\_cnt 는 보호된다. 그리고 inode\_close() 에서 호출하는 free\_map\_release() 는 bitmap 조작 함수로서 모든 비트맵 조작함수와는 lock 이 걸려져 있다.

read() 함수는 fd 가 3 이상, 128 이하인 경우는 file\_read() 를 호출하여 fd 에서 파일을 읽어들여서 buffer 에 저장하도록 하였다. file\_read() 는 inode 를 조회하는 readers 로서 lock 이 걸려져 있다.

write() 함수 또한 fd 가 3 이상, 128 이하인 경우만 file\_write() 를 호출하여 buffer 의 내용을 쓰도록 하였다. file\_write() 또한 비슷하게 inode 를 수정하는 writers 로서 lock 이 걸려져 있다.

create() 는 파일 이름이 없다면 exit(-1) 을 호출하고, filesys\_create() 를 호출한다. filesys\_create() 의 내용은 주석과 같다. Bitmap 조작 함수들은 모두 서로 락이 걸려져 있고, dir\_add() 는 writers 로서 락이 걸려져 있다.

remove(), seek(), tell(), filesize() 의 함수는 위와 같다. 모두 NULL 일때의 예외가 걸려져 있다. 이 중, filesys\_remove(), file\_length() 는 inode 를 조회하거나 수정하는 함수이므로 writers, reader lock 을 적절히 잡아준다. 

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

Image

하나의 실패는 출력이 밀린 것으로 확인된다.