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

담당 교수/분반: 박성용 교수님, 3분반

이름/학번: 김나현, 20191286

개발 기간: 2021/11/06~2021/11/14

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

본 프로젝트는 file system에 관한 다양한 system call을 구현함으로써 프로세스마다 생성하고 연 파일들에 접근하여 읽고, 쓰며 파일의 내용을 변경할 수 있고 파일에 대한 정보를 읽을 수 있게 됩니다. 이때, file descriptor를 사용하여 파일의 이름을 반복적으로 사용하지 않고서도 편리하게 관리할 수 있게 하고 동일한 파일에 동시에 접근하여 쓰고 읽으면 안 되기 때문에 semaphore와 lock을 이용한 synchronization 과정을 통해 이러한 문제를 방지해줍니다.

1. **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 구현함으로써 thread 내에서 여러 개의 file을 다룰 때, 쉽고 빠르게 특정 파일에 접근할 수 있습니다. 모든 process에서 0은 stdin, 1은 stdout, 2는 stderr를 의미하므로 open file들을 관리할 때는 3 이상의 file descriptor를 사용해야 합니다. File descriptor를 사용하면 open file에 대해 파일의 이름을 반복적으로 사용하지 않고도 편리하게 특정 file에 접근이 가능하고 각 process마다 가지고 있는 여러 file들에 대해 관리가 가능합니다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

프로젝트 1에서 구현하지 않았던 file system과 관련된 system call인 create, remove, filesize, seek, tell, open, close을 추가적으로 구현해야 하고 프로젝트 1에서 구현했던 read와 write가 stdin과 stdout을 이외의 다른 file system에 대해서도 동작할 수 있도록 수정해줘야 합니다. 따라서, userprog/syscall.c의 system call handler에서 매개 변수로 입력 받은 f를 이용해 \*(unsigned\*)(f->esp)가 각 system call number, 즉 SYS\_CREATE, SYS\_REMOVE, SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL, SYS\_OPEN, SYS\_CLOSE일 때 알맞은 kernel API를 호출할 수 있도록 하고, create, remove, filesize, seek, tell, open, close에 대한 kernel API 함수를 선언하고 정의해줍니다. 또한, 프로젝트 1에서 이미 구현해놓은 read와 write kernel API 함수를 수정하여 stdin, stdout 이외에도 해당 process의 open file에 대해서도 사용할 수 있도록 수정해줍니다.

3. Synchronization in File system

동일한 파일을 한 쪽에서는 읽는 와중에, 다른 쪽에서는 쓰면서 수정을 하면 원하는 값을 읽지 못하는 등의 문제가 발생하므로 반드시 file system을 접근할 때에는 synchronization을 해주어야 합니다. 따라서, lock을 이용하여 동시에 여러 개의 file에 접근할 수 없게 하여 하나의 thread만이 critical section을 수행할 수 있도록 해야 합니다. 또한 부모 process에서 여러 자식 process를 만들었을 때, 자식 process가 죽기 전에 부모 process가 죽는 것을 방지하기 위해 semaphore를 이용하여 orphan process나 zombie process가 생기지 않게 합니다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

Thread 구조체 안에 file 구조체 포인터를 저장하기 위한, 크기가 128인 배열을 만들고, file이 저장된 마지막 인덱스를 알기 위해 lastfileindex라는 int형 변수만을 사용하게 되면 특정 file이 제거되면 file 구조체 포인터 배열에서 해당 file을 가리켰던 원소가 NULL이 되어 file 구조체 포인터 배열에 있는 실제 file의 개수를 알 수 없기 때문에 실제 file의 개수를 저장하기 위해 realfilenum이라는 int형 변수를 선언할 것입니다. thread 구조체 안에 file 구조체 포인터 배열을 작성하는 이유는 각 process는 하나의 thread를 갖으며 각 process마다 최대 128개의 open file을 가질 수 있다고 매뉴얼에 적혀 있으므로 linked list를 이용하지 않고 단순하게 128개의 file 구조체 포인터를 저장할 수 있는 배열을 선언하여 thread의 lastfileindex 변수를 이용해 (lastfileindex+1)번째 원소부터 채워가는 것입니다. 만약 lastfileindex가 127이 되면 더 이상 배열을 채울 수 없으므로 뒤에서부터 빈 index가 있는지, 즉 remove된 file의 포인터를 저장했던 공간이 있는지를 확인하며 저장합니다.

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)

sys\_open은 *file*이라고 불리는 file을 열고, standard input을 의미하는 0과 standard output을 의미하는 1을 제외한 양의 정수 file descriptor를 반환하며 한 개의 파일이 여러 번 open되면 다른 file descriptor를 생성하여 반환합니다. 같은 파일에 대한 다른 file descriptor는 close라는 시스템 콜에 의해 독립적으로 닫힙니다. 이때, src/filesys/filesys.c에 이미 filesys\_open라는 함수가 구현되어 있으므로 sys\_open에서는 filesys\_open 함수를 호출하여 사용하면 됩니다.

sys\_close는 file descriptor를 변수로 입력 받아 open되어 있는 file descriptor를 닫는 시스템 콜로, 프로세스를 exit하는 것이나 terminate하면 그 프로세스의 모든 open file descriptor에 대해 sys\_close 함수를 호출하여 하나씩 닫습니다. sys\_close 시스템 콜은 아무것도 반환하지 않습니다. 이때, src/filesys/file.c에 이미 file\_close라는 함수가 구현되어 있으므로 sys\_close에서는 file\_close함수를 호출하여 사용하면 됩니다.

sys\_create는 초기에 *initial\_size*의 바이트를 갖는 *file*이라는 불리는 file을 생성하여 성공적으로 생성이 되었을 때는 true를 반환하고, 그렇지 않았을 경우에는 false를 반환합니다. sys\_create 시스템 콜에서는 file을 open하지 않습니다. 이때, src/filesys/filesys.c에 이미 filesys\_create라는 함수가 구현되어 있으므로 sys\_create에서는 filesys\_create 함수를 호출하여 사용하면 됩니다.

sys\_remove는 *file*이라는 불리는 file을 제거하여 성공적으로 제거가 되었을 때는 true를 반환하고, 그렇지 않았을 경우에는 false를 반환합니다. 파일은 열려 있는지, 닫혀 있는지와 상관없이 제거되고 열려 있는 file을 제거하는 것은 sys\_close 함수를 호출하여 file을 close를 하는 과정을 포함하지 않습니다. 이때, src/filesys/filesys.c에 이미 filesys\_remove라는 함수가 구현되어 있으므로 sys\_remove에서는 filesys\_remove 함수를 호출하여 사용하면 됩니다.

sys\_filesize는 fd라는 file descriptor를 변수로 입력 받아, 해당 파일의 사이즈가 몇 바이트인지를 반환합니다. 이때, src/filesys/file.c에 이미 file\_length라는 함수가 구현되어 있으므로 sys\_filesize에서는 file\_length 함수를 호출하여 사용하면 됩니다.

sys\_seek은 fd라는 file descriptor를 변수로 입력 받아, open file fd가 읽거나 써야할 다음 위치를 파일의 시작 지점으로부터 *position* byte만큼 뒤로 바꿔줍니다. 이때, src/filesys/file.c에 이미 file\_seek라는 함수가 구현되어 있으므로 sys\_seek에서는 file\_seek 함수를 호출하여 사용하면 됩니다.

sys\_tell은 fd라는 file descriptor를 변수로 입력 받아, open file fd가 읽거나 써야할 다음 위치를, 파일의 시작 지점으로부터 byte offset으로써 반환합니다. 이때, src/filesys/file.c에 이미 file\_tell라는 함수가 구현되어 있으므로 sys\_tell에서는 file\_tell 함수를 호출하여 사용하면 됩니다.

sys\_read를 수정하여 fd가 0이 아니고, file descriptor가 가리키는 file 구조체 포인터가 NULL이 아니면 src/filesys/file.c에 이미 구현되어 있는 file\_read 함수를 호출하여 파일로부터 실제로 읽은 byte의 수를 반환합니다.

sys\_write를 수정하여 fd가 1이 아니고, file descriptor가 가리키는 file 구조체 포인터가 NULL이 아니면 src/filesys/file.c에 이미 구현되어 있는 file\_write 함수를 호출하여 파일에 실제로 쓴 byte의 수를 반환합니다.

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

struct lock형 전역 변수를 만들어 한 번에 단 하나의 file만을 access하여 쓰거나 읽을 수 있도록 제한하고, parent process가 child process를 생성하고 먼저 죽거나 종료하지 않도록 thread 구조체에 struct semaphore형 변수를 만들어 이 변수를 이용하여 child process가 load되기 전에 부모 process가 죽지 않고 child process가 종료되기 전까지 부모 process가 종료되지 않도록 synchroni zation 합니다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

thread 구조체 내부에 file 구조체 포인터를 저장하기 위한 배열을 만들고, 배열을 사용하기 위해 필요한 int형 변수들을 선언합니다. thread가 만들어질 때, 즉 thread\_create 함수 내부에서 thread 구조체에 추가한 int형 변수들을 초기화해줍니다.

Synchronization 문제를 해결하기 위해 thread.h에서 thread 구조체 내부에 semaphore를 위한 변수들을 선언해주고 filesys.h에서 전역 변수로, struct lock형의 변수를 선언해줍니다.

System call number가 create, remove, filesize, seek, tell, open, close system call을 의미할 때 호출할 kernel API 함수들을 src/userprog/syscall.c에 구현해주고 system call handler에 추가해줍니다.

이미 구현되어 있는 write, read, exit system call 함수를 수정하여 file system에 대해서도 사용이 가능하게 해줍니다.

디버깅을 하며 오류를 찾고, 코드를 수정해줍니다.

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야 하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수

System call handler에 system call number가 SYS\_CREATE, SYS\_REMOVE, SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL, SYS\_OPEN, SYS\_CLOSE일 때 각각 알맞은 kernel API를 호출할 수 있도록 switch문에 추가해줘야 하고, sys\_create, sys\_remove, sys\_filesize, sys\_seek, sys\_tell, sys\_open, sys\_close kernel API를 구현해야 합니다. 뿐만 아니라, 프로젝트 1에서 구현한 sys\_read와 sys\_write을 수정하는 과정도 필요한데 sys\_read와 sys\_write는 각각 stdin과 stdout이 아닌 다른 file descriptor에 대해서도 읽거나 쓸 수 있게 코드를 수정해야 합니다.

sys\_open을 구현할 때에는 실행 중인 파일에 쓰는 것을 방지하기 위해서 thread\_current()->name과 변수로 입력 받은 file이 같을 때(open하려는 파일이 실행 파일일 때) file\_deny\_write을 이용하여 해당 파일에 쓰지 못하게 해야 합니다.

sys\_close를 구현할 때에는 file descriptor를 매개 변수로 입력 받아 file\_close 함수를 이용해 해당 file을 닫아주고 thread\_current()->filelist[fd]=NULL;이라는 코드를 추가하여 현재 돌고 있는 thread의 struct file 포인터 배열의 file descriptor index를 NULL로 바꿔주어야 합니다.

프로젝트 1에서 구현하였던 sys\_exit은 단순하게 thread의 status를 출력하고 thread\_exit이라는 함수를 이용해 종료하는 시스템 콜이지만 본 프로젝트에서는 thread마다 지금까지 열었던 file들이 존재하므로 하나하나 sys\_close 시스템 콜을 이용하여 종료해줘야 합니다. 따라서, sys\_exit 함수에서는 thread 구조체에서 선언한 struct file 포인터 배열을 돌면서 struct file 포인터가 NULL이 아니면 sys\_close 함수를 호출하도록 for loop를 이용한 코드를 추가합니다.

또한, file descriptor을 이용하기 위해 src/threads/thread.h에 구현된 struct thread 선언 부분에 struct file\* filelist[128]; int lastfileindex; int realfilenum;을 추가하여 각 thread마다 file에 file descriptor을 할당하여 file descriptor를 이용해 배열의 index에 random하게 접근할 수 있게 해줍니다. struct thread에서 선언한 lastfileindex와 realfilenum을 초기화하기 위해 src/thread/thread.c의 thread\_create 함수에서 생성한 struct thread 포인터 t의 lastfileindex를 2로, realfilenum을 3으로 설정하는, t->lastfileindex=2; t->realfilenum=3;이라는 코드를 추가합니다. 이때, stdin이 0, stdout이 1, stderr가 2의 index를 가지고 있으므로 마지막 struct file 포인터가 들어있는 배열의 인덱스, stderr의 인덱스인 2이므로 lastfileindex를 2로 설정하는 것이고, 실제로 배열에 저장된 파일의 개수는 stdin, stdout, stderr, 3개로 realfilenum을 3이라고 설정합니다. realfilenum은 단순하게 lastfileindex에 1을 더한 것이라고 생각할 수 있지만 sys\_close를 호출하게 되면 해당 file을 가리키는 struct file 포인터 배열이 NULL로 바뀌므로 배열의 해당 file descriptor 인덱스가 비게 되므로 lastfileindex를 통해 실제로 open된 file의 개수를 알 수 없기 때문에 이러한 변수를 만들어야 합니다.

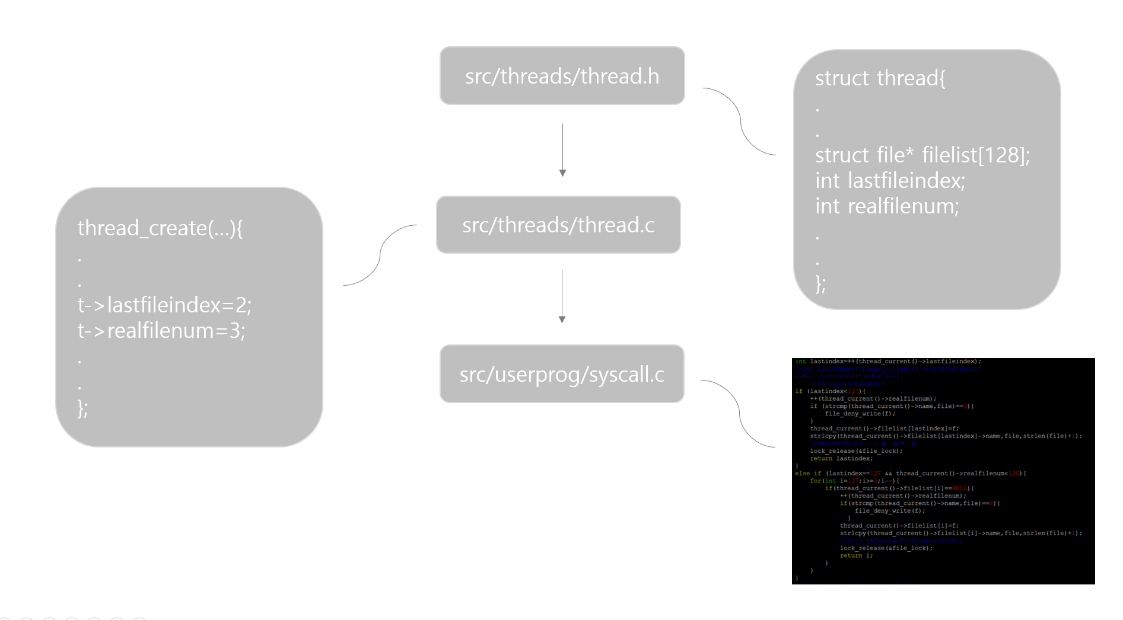
뿐만 아니라 src/threads/thread.h에 구현된 struct thread 선언 부분에 struct semaphore child\_load\_flag;와 struct thread \*parent\_thread;라는 코드를 추가하여 child\_load\_flag라는 세마포어와 부모 thread 포인터를 이용하여 부모 process에서 child process를 생성하였을 때, child process가 load를 하기 전까지 부모 process는 process\_execute라는 함수에서 대기하도록 하여야 합니다.

src/filesys/filesys.h에 struct lock file\_lock;이라는 전역 변수 선언을 통해 src/userprog/syscall.c에서 lock을 통한 synchronization이 가능하게 합니다. 이때, 이미 구현되어 있는 src/filesys/filesys.c의 filesys\_init 함수에 lock\_init(&file\_lock); 코드를 추가하여 file\_lock을 초기화할 수 있도록 해야 합니다. System call 함수 중 sys\_write과 sys\_read 함수에서는 이러한 file\_lock 변수와 lock\_acquire, lock\_release 함수를 사용하여 모든 프로세스가 딱 하나의 file에만 접근할 수 있도록 합니다.

1. **연구 결과**
   1. **Flow Chart**

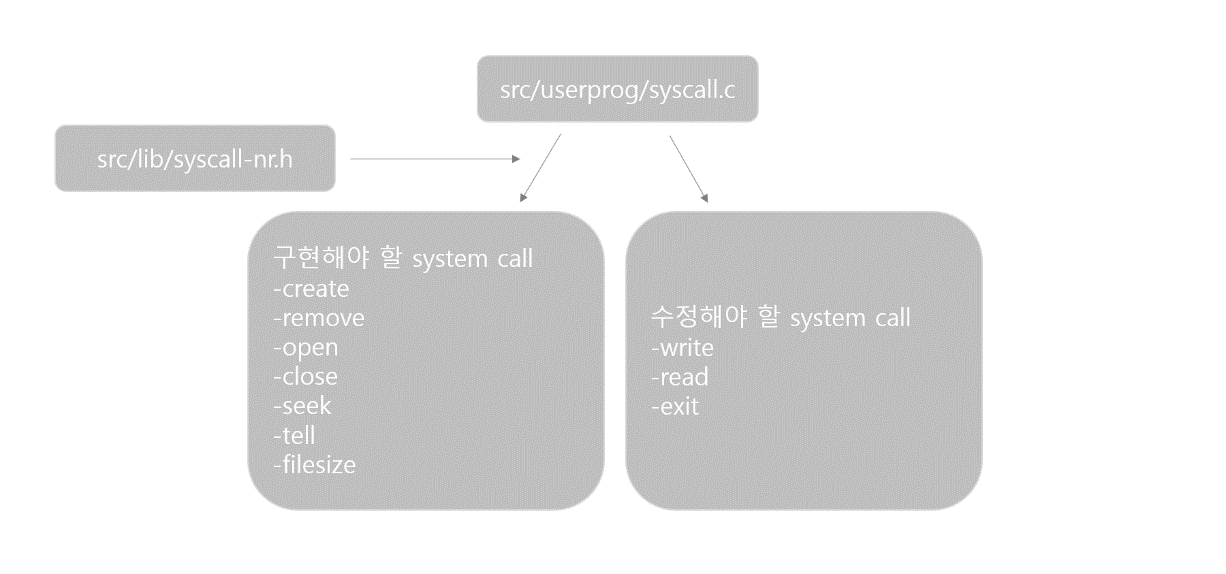
* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성

1. File Descriptor



위의 그림은 file descriptor을 구현하기 위한 flow chart로, thread.h의 thread 구조체 선언 부분에 file 구조체 포인터 배열, filelist을 선언하고, lastfileindex와 realfilenum이라는 정수형 변수를 선언합니다. 그 다음에, thread.c의 thread\_create 함수에서 struct thread\* t=palloc\_get\_page(PAL\_ZERO)일 때, t->lastfileindex=2; t->realfilenum=3;으로 초기화 해줍니다. 시스템 콜 open이 호출되었을 때, lastfileindex와 realfilenum을 보고 배열에 빈 부분을 찾아 파일 포인터를 넣고, 그 index를 반환합니다. 이 외에도 file system과 관련된 시스템 콜에서는 thread\_current()->realfilenum 등을 이용하여 코드를 작성하게 됩니다.

1. System calls



위의 그림은 file system과 관련된 system call을 구현하기 위한 flow chart로, src/userprog/syscall.c에 있는 system call handler의 수정과 system call 함수의 생성 및 수정으로 구성됩니다. 우선, 새로 구현되는 system call들은 system call handler에 작성이 되어 있지 않으므로 프로젝트 1에서 구현했던 다른 system call과 마찬가지로 system call handler에 코드를 추가하여 주면 됩니다. Create, remove, open, close, filesize, seek, tell의 새로운 system call을 구현해야 하고, 이미 구현되어 있는 write, read, exit의 system call의 수정을 통해 system call을 구현할 수 있습니다. 이때, src/lib/syscall-nr.h에 있는 system call number를 참고하여 system call handler를 작성할 수 있습니다.

1. Synchronization

텍스트, 명함, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Lock과 semaphore, 두 가지 synchronization 방법을 통해 모든 process 중 딱 하나의 process만이 file을 read하거나 write하도록 하는 lock 변수를 선언하고 초기화하고, semaphore 변수를 선언하고, 이용하여 하나의 process로부터 여러 개의 process가 생성되었을 때 자식 process가 종료되기 전에 부모 process가 종료되지 않게 구현하였던 프로젝트 1과 유사하게 자식 process가 load되기 전에 부모 process가 process\_execute 함수를 종료하지 않도록 구현하여야 합니다.

* 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 descriptor을 이용하기 위해 src/threads/thread.h에 구현된 struct thread 선언 부분에 struct file\* filelist[128]; int lastfileindex; int realfilenum;을 추가하여 각 thread마다 file에 file descriptor을 할당하여 file descriptor를 이용해 file의 포인터가 있는 배열의 index에 random하게 접근할 수 있게 해줍니다.

random access를 가능하게 하기 위해 file descriptor를 list 형식으로 만들었는데 close 시스템 콜을 호출하면 배열 내부에서 file 포인터를 NULL로 바꿔서 실제 file 포인터가 저장된 배열의 index를 이용하여 배열의 빈 index를 접근할 수 없어지기 때문에 lastfileindex가 127이 되면 배열의 뒤에서부터 index를 하나씩 줄여가며 배열의 빈 부분을 찾아 새로운 open file의 포인터를 저장합니다. 만약 realfilenum이라는 변수가 없고 lastfileindex가 127일 때, 배열의 127번째 index부터 3번째 index까지 모든 배열의 원소를 확인하며 빈 부분을 찾아야 하기 때문에 그러한 시간 낭비를 줄이기 위해 realfilenum을 사용하게 되었습니다.

struct thread에서 바로 lastfileindex와 realfilenum을 각각 2와 3으로 설정하였더니 오류가 발생하여 struct thread에서는 lastfileindex와 realfilenum을 선언만 하고, 따로 각 변수들을 초기화하기 위해 src/thread/thread.c의 thread\_create 함수에서 생성한 struct thread 포인터 t의 lastfileindex를 2로, realfilenum을 3으로 설정하는, t->lastfileindex=2; t->realfilenum=3;이라는 코드를 추가하였더니 문제가 해결되었습니다.

sys\_read에서는 lock\_acquire(&file\_lock);이라는 entry section을 만들어 file system 접근이 available할 때, critical section을 수행할 수 있게 구현하였습니다. sys\_read 함수에서 critical section은 file\_read 함수를 호출하는 부분으로, file\_read 함수가 반환하는 정수를 변수에 저장한 후, lock\_relese(&file\_lock);이라는 exit section을 거친 다음 그 변수를 반환하게 하였습니다. sys\_write에서도 마찬가지로 f->deny\_write이 1일 때도 return 0을 하여 sys\_write을 종료하도록 하였습니다. 그렇지 않으면 lock\_acquire(&file\_lock);이라는 entry section을 만들어 file system 접근이 available할 때, file\_write 함수를 호출하는 critical section을 수행할 수 있게 구현하였습니다. file\_write 함수가 반환하는 정수를 변수에 저장한 후, lock\_relese(&file\_lock);이라는 exit section을 거친 다음 그 변수를 반환하게 하였습니다.

프로젝트 1에서 구현하였던 sys\_exit를 수정하여 thread마다 존재하는 지금까지 열었던 file들을 하나하나 sys\_close 시스템 콜을 호출함으로써 닫아줘야 합니다. 따라서, sys\_exit 함수에서는 thread 구조체에서 선언한 struct file 포인터 배열의 0, 1, 2번째 index를 제외한 모든 index를 돌면서 struct file 포인터가 NULL이 아니면 sys\_close 함수를 호출하도록 for loop를 이용한 코드를 추가하였습니다.

src/filesys/filesys.h에 struct lock file\_lock;라는 코드를 추가하여 file\_lock을 전역 변수로 선언한 후, 초기화 해주지 않아서 write-normal이나 read-normal 등 file system에 대해 읽고 쓰는 기본적인 test가 통과되지 않았는데 src/filesys/filesys.c의 filesys\_init 함수에서 lock\_init 함수를 사용하여 file\_lock이라는 lock 변수를 초기화해 줌으로써 이러한 문제를 해결할 수 있었습니다.

프로젝트 1에서 src/userprog/process.c의 process\_wait과 process\_exit 함수에서 semaphore를이용한 synchronization을 한 것과 같이, 본 프로젝트에서는 struct thread의 변수인 struct semaphore형 변수, child\_load\_flag와 struct thread 포인터 변수, parent\_thread을 이용하여 start\_process와 process\_execute 함수에서 자식 process를 load할 때까지 부모 process에서 process\_execute 함수를 더 이상 수행하지 않고 기다리도록 수정하였습니다.

src/userprog/exception.c에 있는 page\_fault 함수를 수정하여 if(!user || is\_kernel\_vaddr(fault\_addr) || not\_present) sys\_exit(-1);라는 코드를 추가하여 user에 의한 접근이 아니거나 fault address가 커널 가상 주소에 있거나 존재하지 않은 page일 경우, sys\_exit(-1)을 수행하도록 하였습니다.

또한, 다른 system call을 호출할 때와 마찬가지로 is\_user\_vaddr 함수를 이용해 f->esp+n(n은 4의 배수)가 가리키는 주소가 유효한지 체크함으로써 valid한 user 가상 메모리에 대해서만 system call을 호출할 수 있도록 하였습니다.

* 1. **시험 및 평가 내용**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명make check 수행 결과를 캡처하여 첨부