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

담당 교수 / 분반 : 김영재 / 2분반

이름 / 학번 : 박민준 / 20212020

개발 기간 : 10/28~10/31

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

프로젝트 2의 개발 목표는 프로젝트 1 작업에 이어 Pintos 파일 시스템과 사용자 프로그램의 기능을 확장하기 위해 다양한 시스템 호출을 구현하는 것이다.

1) File System Call Implementation

- create, remove, open, close, filesize, seek, tell 등 사용자 프로그램에서 파일 시스템 관련작업을 수행하기 위한 시스템 호출을 구현해야 한다. 이 때, 프로젝트 1에서 구현하였던 read, write 함수 또한 수정해야 한다.

2) Denying Writes to Executable files

- 실행 중인 프로그램 파일에 대한 쓰기를 거부하여 프로그램 실행 도중에 해당 파일이 변경되지 않도록 보호해야 한다.

3) Protect Critical Section

- 여러 프로세스 또는 thread가 파일 시스템과 같은 공유 자원에 동시에 접근할 때 발생할 수 있는 데이터 충돌 및 무결성 문제를 방지해야 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

- File Descriptor는 운영체제에서 파일을 참조하는 정수형 handler이다. 프로세스는 File Descriptor를 사용하여 파일을 열고, 닫고, 읽고, 쓰는 등 파일과 상호작용한다. File Descriptor는 프로세스가 여러 파일에 접근할 수 있게 하고, 각 파일에 대한 고유 식별자를 부여하여 프로세스 간 파일 관리 충돌을 방지하는 중요한 개념이다.

- File Descriptor 구현 시 각 프로세스는 고유한 File Descriptor Table을 가지므로 파일을 독립적으로 관리할 수 있다. 또한, File Descriptor를 통해 동시에 여러 파일에 접근하고 작업할 수 있으며, 유효하지 않은 File Descriptor 사용을 감지하여 예외를 처리함으로써 파일 접근 오류로 인한 시스템 오류를 예방하고 안정성을 높일 수 있다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

- System Call은 사용자 프로그램이 운영체제 커널의 기능을 호출하여 파일 작업, 프로세스 생성, 메모리 관리 등 하드웨어와 가까운 자원에 접근할 수 있도록 하는 Interface이다. 해당 프로젝트에서는 파일과 관련된 System Call을 구현함으로써 사용자 프로그램에서 파일 작업이 가능하도록 한다.

- 파일을 생성하고, 삭제하고, 열고 닫고, 읽고 쓰는 등의 System Call 구현을 통해 파일 시스템을 사용자 프로그램에서 효율적이고 안전하게 제어할 수 있으며, 파일 작업의 안정성과 확장성을 높일 수 있다.

3. Synchronization in Filesystem

- 운영체제는 다중 프로세스 또는 다중 thread 환경에서 동시성 문제를 해결해야 한다. 특히 여러 프로세스가 파일 시스템에 접근할 경우, 데이터 무결성을 유지하고 자원 충돌을 방지하는 동기화 메커니즘이 필수적이다.

- 여러 프로세스가 동시에 파일을 읽고 쓰려고 할 때, lock과 같은 동기화 도구를 사용하여 데이터가 손상되지 않도록 보장한다. 또한, 임계 구역에서 동시 접근을 제한하여 프로그램이 파일을 열거나 닫는 동안 다른 프로세스가 해당 파일에 접근하는 문제를 방지한다. 파일 시스템 작업이 동기화되지 않을 경우, 비정상적인 프로그램 동작이나 kernel panic 등의 시스템 오류가 발생할 수 있다. 동기화 구현은 이러한 오류를 방지하여 시스템의 안정성을 높일 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

- File Descriptor Table을 구현하기 위해 배열을 사용한다. 각 프로세스는 고유의 File Descriptor 배열을 가지고 있으며, 각 인덱스는 File Descriptor를 나타낸다.

- 배열을 사용하면 각 File Descriptor에 상수 시간 복잡도로 접근할 수 있어 효율적이다. 또한 File Descriptor의 고유성을 보장하기 위해 인덱스를 할당하여 파일이 올바르게 열리고 닫히는지 관리하기 매우 용이하다.

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()**: file name과 initial\_size를 parameter로 받아 filesys\_create() 함수에 인자로 넣어 호출하여 파일을 생성한다. root directory에서 file name을 가진 파일을 생성한다. 파일 생성에 성공하면 true, 실패하면 false를 반환한다.

2) **remove()**: file name을 parameter로 받아 filesys\_remove() 함수 호출을 통해 해당 이름을 가진 파일을 삭제한다. 삭제 성공 여부에 따라 true 또는 false를 반환한다.

3) **open()**: file name을 parameter로 받아 filesys\_open() 호출을 통해 파일을 연다. 파일이 성공적으로 열리면 File Descriptor를 반환하고, 실패 시 -1을 반환하여 오류를 나타낸다.

4) **close()**: fd를 parameter로 받아 File Descriptor를 닫고, file\_close() 함수를 통해 해당 파일과의 연결을 해제한다.

5) **filesize()**: fd를 parameter로 받아 file\_length() 함수에 전달하여 파일의 크기를 반환한다. 크기는 byte 단위로 반환된다.

6) **read()**: fd, buffer, size를 parameter로 받아 File Descriptor에서 데이터를 읽어 buffer에 저장한다. fd가 0일 경우 표준 입력에서 읽는다.

7) **write()**: read()와 마찬가지로 fd, biffer, size를 parameter로 받아 File Descriptor에 데이터를 쓴다. fd가 1일 경우 표준 출력으로 데이터를 출력한다.

8) **seek()**: fd와 position을 parameter로 받아 File Descriptor에서 열려 있는 파일의 현재 위치를 position으로 이동시킨다. 위치는 byte 단위로 설정된다.

9) **tell()**: fd를 parameter로 받아 file\_tell() 함수의 호출을 통해 파일의 현재 위치를 반환한다. 위치는 byte 단위로 반환된다.

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

1) Lock: 파일 시스템의 임계 구역에서 lock을 사용하여 프로세스가 파일을 독점적으로 사용할 수 있도록 한다. 특히, 파일을 열거나 닫는 작업, File Descriptor Table의 수정과 같은 작업에 lock을 걸어 동시 접근으로 인한 데이터 손상을 방지하고 thread가 자원을 순서대로 이용하도록 한다.

[src/threads/synch.c]

- **lock\_init()**: Lock을 초기화하는 역할이다. sema\_init() 함수를 사용해 semaphore의 값을 1로 초기화하여 Lock을 한 번에 하나의 thread만 가질 수 있도록 설정한다.

- **lock\_acquire()**: sema\_down() 함수를 호출하여 Lock을 획득하는 데 사용된다. Lock이 이미 다른 thread에 의해 사용 중이면 현재 thread는 대기 상태로 전환된다. Lock이 확보되면 현재 thread를 Lock의 소유자로 설정한다.

- **lock\_release()**: Lock을 해제하는 데 사용된다. 현재 thread가 Lock을 소유하고 있어야만 호출할 수 있으며, Lock의 소유자를 NULL로 설정하고 sema\_up() 함수를 호출하여 대기 중인 다른 thread 중 하나를 깨워 Lock을 확보할 수 있도록 한다.

2) Semaphore: 여러 프로세스가 동일한 자원에 접근하는 상황에서 동기화를 위한 semaphore를 사용하여 상호 배제를 유지한다. 파일 입출력 작업의 대기열을 관리하고, 파일 시스템의 효율성을 높이는 데 활용할 수 있다.

[src/threads/synch.c]

- **sema\_down()**: semaphore의 값을 감소시키며, 값이 0이 되면 자원이 모두 사용 중이라는 의미로 thread를 대기 목록에 추가한 후 대기 상태로 전환한다.

- **sema\_up()**: sema\_down() 함수와 반대로 semaphore의 값을 증가시키고, 대기 중인 thread가 있으면 그 중 하나를 대기 목록에서 제거하고 실행 준비 상태로 전환하여 자원을 사용할 수 있도록 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10/28 ~ 10/29: File Descriptor 자료구조 정의 및 데이터 구조 구현. create, remove, open, close, filesize, seek, tell에 대한 System Calls 구현.

10/30: 파일 시스템 접근에 대한 Lock과 Sempaphore를 이용해 동기화 작업 구현. 모든 System Call과 동기화 구현이 정상 작동하는지 테스트 및 디버깅

10/31: 해당 프로젝트에 대한 최종 보고서 작성.

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

[src/userprog/process.c]

- **process\_execute()**: child thread가 load 과정을 완료할 때까지 parent thread가 대기하도록 sema\_down() 함수를 호출해 load가 정상적으로 완료될 때까지 현재 thread가 기다리도록 수정하였다. 즉, child thread의 종료를 parent thread가 기다릴 수 있도록 child thread의 list를 검색하며, load 실패 시 process\_wait() 함수 호출을 통해 child thread의 종료를 대기할 수 있도록 코드를 수정하였다.

- **start\_process()**: 사용자 프로세스가 시작할 때 file load가 성공적으로 완료되었는지 확인하기 위해 load가 끝나면 부모 thread의 sema\_up() 함수를 호출하여 대기 중이던 parent thread에 신호를 보낸다. Load 실패 시 exit(-1)을 호출해 child thread가 종료되도록 변경하였다.

[src/userprog/syscall.c]

- **read()**: fd가 3 이상일 때, 즉 표준 입력 이외의 파일에 대해서는 file\_lock을 이용해 다중 thread 동기화 문제를 방지하였으며, 파일이 열려 있지 않으면 exit(-1)을 호출하도록 수정하였다.

- **write()**: fd가 3 이상일 때, 즉 표준 출력 이외의 파일에 대해서는 file\_lock을 통해 동기화를 유지하며, read() 함수와 마찬가지로 fd가 유효하지 않거나 파일이 열려 있지 않은 경우 exit(-1)을 호출하여 오류를 처리하도록 수정하였다.

- **syscall\_handler()**: create, remove, open, close 등 파일 시스템 관련 system call 함수에 대해 스택의 esp 값을 확인하여 적절한 system call을 호출하도록 코드를 구성하였다.

- **syscall\_init()**: syscall.c 파일의 전역 변수로 선언한 lock 구조체 file\_lock을 초기화하여 파일 시스템 접근에 대한 동기화를 관리하도록 하였다.

[src/threads/thread.c]

- **init\_thread()**: thread 구조체의 멤버들을 초기화한다. USERPROG가 정의되어 있는 경우에 한해 사용자 프로그램을 위한 동기화 및 부모-자식 thread 관리 기능을 추가하였다. sema\_init() 함수를 이용하여 semaphore child\_lock, mem\_lock, load\_lock을 초기화하여 자식 thread 생성 시의 순서 및 종료 동작에서의 동기화를 지원하도록 코드를 수정하였다. 또한, thread 생성 시 현재 thread를 부모로 설정하기 위해 t->parent = running\_thread()를 설정하였다.

* + 수정하거나 추가해야 하는 자료구조

[src/threads/thread.h/struct thread]

- **struct thread \*parent**: 현재 thread의 부모 thread를 가리키는 포인터이다. 이를 통해 부모와 자식 thread 간에 관계를 유지하고, 부모가 자식의 종료 상태를 확인할 수 있다.

- **struct semaphore load\_lock**: 자식 thread가 load 작업을 완료할 때까지 부모 thread가 대기할 수 있도록 하는 semaphore이다. 프로세스가 정상적으로 load 되었는지를 확인하는 데 사용된다.

- **struct file \*fd[128]**: File Descriptor 배열로, 현재 thread가 열고 있는 파일들을 관리한다. 각 파일은 인덱스를 통해 참조되며, 최대 128개의 파일을 열 수 있도록 설정하였다. 배열의 각 요소는 파일에 접근할 수 있또록 file pointer로 관리된다.

* + 수정하거나 추가해야 하는 함수

[src/userprog/syscall.c]

- **create()**: 인자로 파일 이름과 초기 크기를 받아 filesys\_create()를 호출하여 해당 이름과 크기의 파일을 생성한다.

- **remove()**: 파일 이름을 인자로 받아 해당 파일을 삭제하도록 filesys\_remove()를 호출한다.

- **open()**: 인자로 받은 파일 이름을 사용하여 filesys\_open()을 호출한다. 성공적으로 열렸다면 thread의 File Descriptor Table에 파일을 등록한다.

- **close()**: 인자로 fd를 받아 해당 파일을 닫고 File Descriptor Table에서 제거한다.

- **filesize()**: 인자로 fd를 받아 file\_length()를 호출하여 파일 크기를 가져온다.

- **seek()**: fd와 이동할 위치를 인자로 받아 해당 파일의 file pointer를 조정한다.

- **tell()**: fd를 인자로 받아 해당 파일의 file pointer 위치를 확인하고 반환한다.

1. **연구 결과**
   1. **Flow Chart**

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

1. **File Descriptor**

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

**2. System Calls (해당 프로젝트에서 수정하거나 추가된 system call만 표현)**

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

**3. Synchronization in FileSystem**

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실질적으로 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**

[src/threads/thread.h]

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- struct file \*fd[128]는 File Descriptor Table로, 최대 128개의 열린 파일을 관리하는 포인터 배열이다. 배열의 각 인덱스는 특정 파일을 나타내며, 해당 인덱스가 File Descriptor 값으로 사용된다. 예를 들어, 인덱스가 3인 경우 fd[3]에 저장된 파일 포인터는 File Descriptor 3을 나타내며, 이 파일을 열고 사용하는 데 사용된다.

[src/threads/thread.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- init\_thread() 함수에서 fd 배열을 초기화해준다. thread가 생성될 때마다 각 배열 요소를 NULL로 설정하여 모든 File Descriptor 슬롯이 비어있음을 나타낸다. for loop는 fd[3]부터 fd[127]까지를 NULL로 설정한다. 배열 인덱스 3부터 시작하는 이유는 fd[0], fd[1], fd[2]가 각각 표준 입력, 표준 출력, 표준 오류로 이미 예약되어 있기 때문이다.

[src/userprog/syscall.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- open() 함수는 주어진 파일 이름을 열고, 해당 파일에 대한 File Descriptor 번호를 반환한다. 이 때, thread의 fd 배열이 사용된다. for loop를 사용해 fd[3]부터 fd[127]까지 빈 슬롯(NULL 값)을 찾는다. 빈 슬롯이 발견되면, 열려 있는 파일 포인터를 해당 인덱스에 저장하고, 그 인덱스를 File Descriptor로 반환한다. 만약 fd 배열에 빈 슬롯이 없다면 파일을 열 수 없으므로 -1을 반환한다.

**2. System Calls**

[src/userprog/syscall.h]

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

- syscall.h 파일에 파일 시스템 관련 시스템 호출을 위한 함수들을 추가로 정의하였다.

[src/userprog/syscall.c]

**텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

- syscall\_handler() 함수의 switch-case 구문에 사용자 프로그램이 요청하는 파일 관련 작업을 직접적으로 처리하는 부분을 추가로 구현하였다. 호출된 시스템 호출 번호에 따라 각각의 파일 작업을 위한 함수를 호출하게 된다.

- 각 시스템 호출에서 인수로 전달된 포인터 주소가 유효한지 check\_user\_vaddr() 함수를 사용해 확인한다. 이를 통해 커널이 잘못된 메모리 접근으로 인해 충돌하지 않도록 보호하며, 유효하지 않은 경우 즉시 종료하여 안전성을 보장한다.

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

- create() 함수는 파일을 생성하는 기능을 갖는다. 인자로 파일 이름과 초기 크기를 받는다. 만약 파일 이름이 NULL일 경우 exit(-1)을 호출하여 잘못된 입력으로 프로그램을 종료한다. 이후 filesys\_create() 함수를 호출하여 파일을 생성하며, 생성 성공 여부를 반환한다.

텍스트, 폰트, 스크린샷, 친필이(가) 표시된 사진

자동 생성된 설명

- remove() 함수는 지정된 이름을 가진 파일을 삭제한다. filesys\_remove() 함수를 호출하여 파일을 삭제하며, 삭제 성공 여부를 반환한다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- open() 함수는 파일을 열고, File Descriptor를 반환한다. file이 NULL일 경우, 오류를 의미하는 -1을 반환한다. file\_lock을 사용하여 동기화하고, filesys\_open() 함수를 호출하여 파일을 연다. file\_lock 변수는 3번에서 설명할 예정이다.

- 파일 포인터 f가 NULL이 아니라면 현재 thread의 fd 배열에서 비어 있는 첫 번째 위치를 찾아 파일 포인터를 저장하고, 그 인덱스를 idx에 저장한다.

- 모든 File Descriptor가 사용 중이라면, 파일을 닫고 -1을 반환한다.

- 만약 thread 이름과 파일 이름이 일치하면 쓰기 작업을 방지하기 위해 file\_deny\_write() 함수를 호출하고 deny\_write\_file flag를 true로 설정해준다.

- open 함수의 작업이 완료되면 lock\_release() 함수를 통해 file\_lock을 해제함으로써 현재 thread가 사용 중인 파일 시스템에 대한 잠금을 풀어준다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- filesize() 함수는 fd가 가리키는 파일의 크기를 반환한다. 현재 thread의 fd 배열에서 파일 포인터를 가져와 file\_length() 함수를 호출하여 파일의 크기를 반환해준다.

텍스트, 폰트, 스크린샷, 친필이(가) 표시된 사진

자동 생성된 설명

- seek() 함수는 fd로 열린 파일의 읽기 또는 쓰기 위치를 position으로 설정한다. fd 배열에서 파일 포인터를 가져와 file\_seek() 함수를 호출하여 위치를 설정해준다.

텍스트, 폰트, 스크린샷, 친필이(가) 표시된 사진

자동 생성된 설명

- tell() 함수는 fd의 현재 위치를 반환한다. fd 배열에서 파일 포인터를 가져와 file\_tell() 함수를 호출하여 위치를 반환해준다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- close() 함수는 fd에 연결된 파일을 닫고, File Descriptor 배열에서 제거해준다. fd가 유효하지 않거나 NULL인 경우 exit(-1)로 오류를 처리해준다.

- file\_lock을 통해 동기화하고, 파일에 쓰기 제한이 설정되어 있다면 file\_allow\_write() 함수를 호출하여 쓰기를 허용한 뒤 deny\_write\_file을 false로 설정하여 flag를 해제해준다.

- file\_close() 함수로 파일을 닫고 fd 배열에서 NULL로 설정하여 파일이 닫혔음을 나타낸다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**-** read() 함수는 fd를 통해 지정된 파일로부터 size 바이트의 데이터를 buffer로 읽어온다.

- 먼저 check\_user\_vaddr(buffer)를 통해 buffer가 사용자 주소 영역 내에 있는지 확인한다.

- fd가 0보다 작거나 1이거나 128 이상일 경우 프로그램을 바로 종료한다.

- file\_lock을 획득하여 다른 thread가 동시에 파일 시스템에 접근하지 못하도록 보호한다.

- fd가 0인 경우, 표준 입력으로부터 키보드 입력을 읽어 buffer에 저장한다. fd가 일반 파일인 경우, file\_read() 함수를 호출하여 해당 파일에서 데이터를 읽어 buffer에 저장한다.

- 읽기가 완료되면 lock\_release(&file\_lock);을 통해 잠금을 해제하여 다른 thread가 파일 시스템에 접근할 수 있도록 한다.

**텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명**

- write() 함수는 fd를 통해 지정된 파일에 size 바이트의 buffer 데이터를 쓴다.

- read() 함수와 동일하게 먼저 buffer가 사용자 영역의 주소인지 확인한다.

- fd가 0보다 작거나 128 이상일 경우 프로그램을 바로 종료한다.

- file\_lock을 획득하여 다른 thread의 동시 접근을 방지한다.

- fd가 1인 경우, 표준 출력으로 buffer의 내용을 출력한다. fd가 일반 파일인 경우, file\_write() 함수를 통해 해당 파일에 데이터를 쓴다.

- 쓰기 작업이 완료되면 lock\_release(&file\_lock);을 호출하여 다른 thread가 파일 시스템에 접근할 수 있도록 잠금을 해제한다.

[src/userprog/exception.c]

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- make check 실행 결과 multi-oom과 bad-read, bad-write, bad-read2, bad-write2, bad-jump 부분이 계속해서 실패하였다. 이에 메모리 접근 관리, 특히 페이지 폴트 처리에 관련된 오류일 가능성을 의심하였고, 페이지 폴트가 발생하였을 때 단순히 종료하지 않고 접근한 주소가 유효한지를 검사하는 코드를 추가하여 문제를 해결하였다.

**3. Synchronization in FileSystem**

[src/threads/thread.h]

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- struct thread \*parent: parent 포인터는 현재 thread의 부모 thread를 가리킨다. thread가 생성될 때 parent 필드를 통해 생성한 thread와 부모 thread 간의 관계가 설정된다. 부모-자식 관계를 통해 부모 thread는 자식 thread의 상태를 추적하거나, 종료 시 자식이 남긴 정보를 가져올 수 있다.

- struct semaphore load\_lock: thread가 특정 작업이 완료될 때까지 대기하는 데 사용하는 semaphore이다. 새 process or thread를 생성할 때 부모 thread는 자식 thread가 파일을 성공적으로 load할 때까지 대기해야 한다. 이 때 load\_lock이 자식 thread의 load 작업이 끝날 때까지 부모 thread를 대기 상태로 유지한다. 자식 thread에서 load 작업이 완료되면 sema\_up()을 호출하여 부모 thread를 깨워주게 된다.

- bool deny\_write\_file: 현재 실행 중인 thread가 자기 자신의 실행 파일에 대한 쓰기 접근을 방지하는 데 사용되는 flag이다. 실행 중인 파일에 대한 쓰기를 막음으로써 파일 손상을 방지하고 일관성을 유지한다. 만약 현재 실행 중인 파일이 다시 열리면 file\_deny\_write() 함수가 호출되어 쓰기 접근이 거부된다.

[src/threads/thread.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- t->parent = running\_thread(): parent를 현재 실행 중인 thread로 설정한다.

- sema\_init(&(t->load\_lock), 0): semaphore를 0으로 초기화하여 기본적으로 자식 thread가 load될 때까지 잠겨 있는 상태로 만든다.

- t->deny\_write\_file = false: 기본적으로 쓰기 거부 flag를 false로 설정한다.

[src/userprog/process.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 부모 thread는 process\_execute() 함수에서 thread\_creat() 함수를 통해 자식 thread를 생성한다.

- load\_lock sempahore는 부모 thread가 자식의 load 상태를 기다리는 동안 사용하는 동기화 도구이다.

- 부모 thread는 sema\_down()을 호출하여 semaphore를 잠근다. 이 때 자식 thread가 load 완료될 때까지 부모 thread는 대기하게 된다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 자식 thread는 start\_process() 함수에서 load가 성공적으로 완료되면 sema\_up(&thread\_current()->parent->load\_lock);을 호출하여 부모 thread를 깨운다. 이를 통해 자식 thread가 laod를 성공적으로 마쳤음을 부모 thread에게 알리고, 부모 thread는 semaphore가 해제되면서 대기 상태에서 깨어나게 된다.

[src/userprog/syscall.c]

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

- syscall.h 파일에 struct lock type의 file\_lock을 전역 변수로 설정한다. 해당 변수는 다중 thread 환경에서 파일에 대한 동시 접근을 안전하게 관리하기 위해 사용되는 변수이다.

- 파일 시스템에서 접근하는 여러 시스템 호출 함수에서 사용된다. 이를 통해 한 thread가 파일 시스템을 수정하거나 읽는 동안 다른 thread가 동시에 접근하지 못하게 하여 데이터의 무결성과 일관성을 유지할 수 있다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

- file\_lock은 syscall\_init() 함수에서 lock\_init(&file\_lock);을 통해 초기화된다. 이를 통해 file\_lock은 사용 준비 상태가 된다.

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

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- cspro에서 make check 수행 시 출력 밀림 현상이 자주 발생하여 정확한 수행 결과를 확인할 수 없었다. 따라서 vscode를 이용해 local 환경에서 작업한 후 make check 수행 결과를 확인하였고, 이를 캡쳐하여 첨부하였다.