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

담당 교수 / 분반 : 김영재 교수님 / 2분반

이름 / 학번 : 김동현 / 20190345

개발 기간 : 2023.10.09 - 2023.10.28

1. **개발 목표**

파일 디스크립터 자료구조를 사용하여 파일 작업을 위한 정보를 저장한다. 또한 파일 관련 시스템 콜을 구현하여 파일 관련 작업을 수행한다. 이 과정에서 프로세스가 동시에 동일한 파일에 접근하는 것을 막기 위해 동기화 과정을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

프로세스는 프로그램을 실행하면서 파일에 접근하여 데이터를 읽고 써야한다. 이렇게 접근해야할 파일을 관리하는 자료구조가 File Descriptor Table이다. 유저 프로세스가 직접 파일에 접근하게 된다면, 권한이 없는 파일에 접근하거나 다른 프로세스와의 동기화 문제가 발생할 수 있으므로 시스템 콜을 사용하여 OS에게 파일에 관한 정보를 요청하게 된다. 이를 관리하기 위해 File Descriptor가 존재하며, 시스템 콜을 통해 받은 파일 정보와 시스템 콜에 요청한 파일 정보를 저장하고 관리한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

파일 시스템과 관련된 시스템콜을 구현한다. 파일을 생성하는 create( ), 제거하는 remove( ), 파일을 여는 open( ), 파일을 닫는 close( ), 파일 사이즈를 얻는 filesize( ), 파일의 오프셋을 이동하는 seek( ), 파일의 offset을 찾는 tell( )을 구현하고, 파일 디스크립터를 통해 읽고 쓰는 작업을 수행하기 위해 read( ), write( ) 시스템 콜을 수정한다.

3. Synchronization in Filesystem

하나의 파일은 하나의 공유 자원으로서 여러 프로세스에서 동시에 접근할 수 있다. 하나의 프로세스에서 파일 내용을 사용하고 있을 때에 다른 프로세스에서 파일 내용을 수정하거나 메모리에 파일이 올라와 있는데 파일의 내용이 수정되게 된다면, 오류가 발생할 수 있다. 이를 막고자 Lock과 같은 자료구조를 통해 동기화를 해주어야 한다.

* 1. **개발 내용**

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

파일 구조체를 가리키는 포인터를 원소로 하는 리스트 자료구조를 사용하여 file descriptor table을 구현한다. 핀토스 메뉴얼에서 하나의 프로세스에서 열 수 있는 파일의 최대 개수는 128개이므로 128개의 파일 구조체 포인터를 원소로 가지는 리스트를 정적으로 선언한다. 또한 해당 원소에 접근하기 위해서는 최대 128번의 순회로 해당 원소에 접근할 수 있기에 연결리스트보다는 linear list를 사용하는 것이 효율적이라 생각하였다.

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)

- create( ) : 인자로 들어온 문자열을 파일 이름으로 하는 파일을 생성한다. 이때 파일의 크기는 인자로 들어온 정수형 크기로 설정한다. 파일 이름이 NULL이라면 exit( ) 시스템 콜을 호출하고, 그렇지 않다면 파일을 생성하고, 파일 생성 성공 여부를 반환한다.

- remove( ) : 인자로 들어온 문자열을 이름으로 가진 파일을 제거한다. 문자열이 NULL이라면 exit( ) 시스템 콜을 호출하고, 그렇지 않다면 파일을 제거하고, 파일 제거 성공 여부를 반환한다.

- open( ) : 인자로 들어온 문자열을 이름으로 가진 파일을 오픈한다. 해당 파일을 열지 못하거나 해당 프로세스의 파일 디스크립터가 가득차 더이상 저장하지 못한다면 -1을 리턴하고, 그렇지 않다면 해당 파일 구조체를 프로세스의 파일 디스크립터에 저장하고 해당 fd의 인덱스를 반환한다.

- close( ) : 인자로 들어온 file descriptor 인덱스에 있는 파일을 닫는다. 해당 파일일 없을 경우 exit( ) 시스템 콜을 호출하고, 있다면 해당 파일을 닫고 해당 원소를 NULL로 초기화한다.

- filesize( ) : 인자로 들어온 file descriptor 인덱스에 있는 파일의 크기를 반환한다.

- seek( ) : 인자로 들어온 file descriptor 인덱스에 있는 파일의 offset을 인자로 들어온 정수형으로 바꾼다. 만약 파일이 없다면 exit( ) 시스템 콜을 호출하고, 그렇지 않다면 offset을 변경한다.

- tell( ) : 인자로 들어온 file descriptor 인덱스에 있는 파일의 offset을 반환한다. 만약 파일이 없다면 exit( ) 시스템 콜을 호출하고, 그렇지 않다면 offset을 반환한다.

- read( ) : 기존 read 시스템 콜에서 파일로의 입력 기능을 추가한다. 해당 디스크립터 인덱스에 파일이 있을 경우 해당 파일로부터 데이터를 읽어오고, 없을 경우 exit( )시스템 콜을 호출한다.

- write( ) : 기존 write 시스템 콜에서 파일로의 출력 기능을 추가한다. 해당 디스크립터 인덱스에 파일이 있을 경우 해당 파일로 데이터를 출력하고, 없을 경우 exit( )시스템 콜을 호출한다.

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

파일 시스템과 관련된 시스템 콜에서 read( ), write( ), open( )을 할 경우 lock을 이용하여 여러 프로세스가 해당 시스템 콜을 동시에 실행하지 못하도록 한다. 해당 함수에서 작업을 수행하기 전 lock\_aquire( )을 통해 다른 프로세스가 접근하지 못하도록 한 뒤, 작업이 끝나게 되면 lock\_release( )을 통해 다른 프로세스의 접근을 허용한다. 이때 같은 프로세스에서 lock과 unlock을 수행하기 때문에 lock 자료구조를 사용하다.

부모 프로세스가 자식 프로세스를 생성하는 과정에서 자식 프로세스의 생성을 부모 프로세스가 기다리도록 하기 위해 semaphore을 이용한다. 부모는 sema\_down( )을 통해 자식 프로세스의 생성을 기다리고, 자식은 스레드를 초기화하고, 파일을 적재하는 과정 등을 모두 수행한 뒤, 프로세스 생성이 완료된 뒤, 부모 프로세스의 semaphore을 sema\_up()하여 부모 프로세스에게 프로세스 생성이 완료되었음을 알려준다. 이를 통해 자식 프로세스가 메모리 적제에 실패 했을 경우 process\_wait( )를 통해 reaping할 수 있도록 하여 메모리 leaking을 예방한다. 이러한 과정은 부모 프로세스와 자식 프로세스 사이에서 sema\_up( ), sema\_down( )을 수행하기 때문에 semaphore 자료구조를 사용한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **2023.10.09 ~ 2023.10.11 : pintos manual 및 ppt 숙지**
* **2023.10.12 ~ 2023.10.12 : file descriptor table 구현**
* **2023.10.13 ~ 2023.10.15 : system call 구현**
* **2023.10.15 ~ 2023.10.20 : synchronization 구현**
* **2023.10.21 ~ 2023.10.26 : debugging**
* **2023.10.26 ~ 2023.10.28 : document 작성**
  1. **개발 방법**
  + 수정해야하는 소스코드  
    1. userprog/syscall.c sys\_read( ) : 기존 standard input 만이 아닌 파일 디스크립터에 있는 파일을 통해 데이터를 읽어온다. 인자로 들어온 fd를 통해 표준 입출력 0,1번을 제외한 2번부터 127번 인덱스의 파일 디스크립터 테이블 원소를 확인한다. 파일이 존재한다면, file\_read( ) 함수를 호출하여 파일로부터 데이터를 읽어온다. 이후, 읽어온 데이터의 offset을 반환한다. 이때 파일 간 동기화를 위해 함수가 호출되자마자 lock을 건 뒤, 리턴하여 함수를 종료하기 전에 unlock한다.  
      
    2. userprog/syscall.c sys\_write( ) : 기존 stardard ouput 만이 아닌 파일 디스크립터에 있는 파일에 데이터를 저장한다. 인자로 들어온 fd를 통해 표준 입출력 0,1번을 제외한 2번부터 127번 인덱스의 파일 디스크립터 테이블 원소를 확인한다. 파일이 존재한다면 file\_write( ) 함수를 호출하여 파일에 데이터를 작성한다. 이후 작성한 데이터의 크기를 반환한다. 이때 파일 간 동기화를 위해 함수가 호출되자마자 lock을 건 뒤, 리턴하여 함수를 종료하기 전에 unlock한다.  
      
    3. userprog/syscall.c syscall\_init( ) : 시스템 콜 중 파일 동기화를 위해 사용할 lock 자료조를 lock\_init( ) 함수를 통해 초기화한다.  
      
    4. userprog/syscall.c syscall\_handler( ) : lib/syscall-nr.h에 선언되어 있는 시스템콜 고유 번호에 맞는 파일 시스템콜 함수를 호출하도록 한다. SYS\_CREATE, SYS\_REMOVE, SYS\_OPEN, SYS\_CLOSE, SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL을 스위치 문에 추가시켜 각각 알맞은 시스템 콜 함수를 호출하도록 한다. 이때 해당 시스템 콜이 인자를 가지고 있다면, check\_user\_address( ) 함수를 통해 올바른 주소 접근인지 확인한다. 해당 시스템 콜이 반환값을 가질 경우 eax 레지스터에 값을 저장하도록 한다.  
      
    5. userprog/syscall.c sys\_exit( ) : 프로세스가 exit 할 때, 해당 프로세스가 가지고 있는 파일 디스크립터 테이블의 원소 중 열려 있는 파일을 모두 닫아준다. 이를 통해 메모리 leaking을 막을 수 있다.  
      
    6. userprog/process.c start\_process( ) : 자식 프로세스의 관점에서 프로세스가 시작되면 호출되는 함수로, load( ) 함수를 통해 스택에 인자들을 적재한다. 이후 load( )가 끝난다면, 메모리에 프로세스 적재가 끝났음을 의미하므로, 부모 프로세스의 load\_sema에 접근하여 sema\_up( ) 함수를 통해 메모리 적재가 끝났음을 알려주어 부모가 load 성공 여부에 따라 이에 맞는 작업을 수행하도록 순서를 강제해준다.  
      
    7. userprog/process.c process\_execute( ) : 부모 프로세스의 관점에서 thread\_create( )를 통해 자식 프로세스를 통해 새로운 프로세스를 실행한다. 이때 자식 프로세스의 메모리 적재를 기다리기 위해 load\_sema를 사용한다. 이 세마포어는 자식 프로세스의 메모리 적재를 기다리기 위한 자료구조로 sema\_down( )을 통해 자식 프로세스를 기다린다. 이후 findLoadFailedEntry( ) 함수를 통해 thread\_create( )를 통해 만들어진 자식 프로세스의 load( )가 정상적으로 이루어졌는지 확인한다. 만약 해당 함수가 true라면 load가 이루어지지 못한 함수이므로 바로 process\_wait( ) 함수를 호출하여 해당 자식 프로세스를 reaping 해준 뒤, 종료 번호를 반환한다. 그렇지 않을 경우에는 해당 자식 프로세스의 아이디를 리턴해준다.  
      
    8. userprog/exception.c page\_fault( ) : 가상 주소에 의해 mapping 되는 물리 주소가 실제 메모리에 존재하는지를 나타내는 not\_present 변수를 확인한다. 만약 존재하지 않는 페이지라면, exit( ) 시스템 콜을 호출한다.  
      
    9. threads/thread.c init\_thread( ) : 새로운 프로세스를 만들기 위한 함수로 USERPROG영역에서 사용하는 sturct thread 자료구조를 초기화한다. 파일 디스크립터 테이블의 모든 원소를 NULL로 초기화 해준다. 이외에도 is\_loaded 함수를 true로 초기화하여 로드에 실패한 경우에만 false를 저장할 수 있도록 한다. 부모 프로세스 정보를 저장하기 위해 running\_thread( ) 함수를 통해 부모 프로세스의 struct thread의 포인터를 저장한다.
  + 수정하거나 추가해야 하는 자료구조  
    1. userprog/syscall.h struct lock load\_file\_lock : 파일에 여러 프로세스가 동시에 접근하지 못하도록 하는 lock 자료구조이다. userprog/syscall.c 에 있는 시스템 콜 함수에서만 해당 자료구조를 사용하므로, syscall.h에서 전역 변수로 설정한다.   
      
    2. threads/thread.h struct file \* fd\_list[128] : 파일 디스크립터 테이블로 파일 포인터를 원소로 한다. 프로세스 별로 파일 디스크립터 테이블을 가지고 있어야 하기 때문에 thread 구조체에 선언한다.  
       
    3. threads/thread.h struct semaphore load\_sema : 부모 프로세스가 자식 프로세스의 순서를 강제하기 위한 세마포어이다. 자식 프로세스에서 메모리 적재 함수인 load( ) 함수를 수행한 이후, 부모 프로세스가 적재 성공 여부에 맞게 대처하는 순서를 맞춰주기 위함이다.  
      
    4. threads/thread.h bool is\_loaded : 해당 프로세스의 메모리 적재 여부를 나타내는 자료구조이다. true이면 메모리 적재에 성공한 경우이고, false이면 메모리 적재에 실패한 경우이므로 부모 프로세스에서 이에 맞는 reaping을 해주어야 한다.  
      
    5. threads/thread.h thread \* parent : 해당 프로세스를 호출한 부모 프로세스를 저장하는 함수로, 자식 프로세스에서 부모 프로세스의 load\_sema를 접근하기 위해 선언한다.
  + 수정하거나 추가해야 하는 함수  
    1. userprog/syscall.c sys\_create( ) : 인자로 들어오는 문자열을 이름으로 하는 파일을 생성하는 시스템 콜이다. 인자로 들어온 문자열이 NULL이라면 exit( ) 시스템 콜을 호출한다. 그렇지 않다면, filesys\_create( ) 함수를 호출하여 파일을 생성하고, 파일 생성 성공 여부를 반환한다.  
      
    2. userprog/syscall.c sys\_remove( ) : 인자로 들어오는 문자열을 이름으로 하는 파일을 제거하는 시스템 콜이다. 인자로 들어온 문자열이 NULL이라면 exit( ) 시스템 콜을 호출한다. 그렇지 않다면, filesys\_remove( ) 함수를 호출하여 파일을 제거하고, 파일 제거 성공 여부를 반환한다.  
      
    3. userprog/syscall.c sys\_open( ) : 인자로 들어오는 문자열을 이름으로 하는 파일을 생성하는 시스템 콜이다. 인자로 들어온 문자열이 NULL이라면 exit( ) 시스템 콜을 호출한다. 그렇지 않다면, filesys\_open( ) 함수를 호출하여 파일을 오픈한다. 이때 파일을 여는 과정에서의 동기화를 위해 lock 자료구조를 사용한다. 파일 오픈에 성공했다면, 해당 프로세스의 파일 디스크립터 테이블의 남는 원소 중 가장 작은 인덱스에 해당 파일 포인터를 저장한다. 이때 파일을 열지 못했거나 파일 디스크립터 테이블이 가득 차 저장하지 못한 경우에는 -1을 그렇지 않다면 해당 파일 포인터를 저장한 디스크립터 테이블 원소 인덱스를 반환한다. 또한 파일을 실행하는 중 파일을 변경하는 것을 막기 위해 해당 파일을 file\_deny\_write( )를 호출하여 수정을 막는다.  
      
    4. userprog/syscall.c sys\_close( ) : 현재 프로세스의 파일 디스크립터 테이블에서 원소로 들어온 인자에 있는 파일 포인터를 찾는다. 해당 파일 포인터가 NULL이라면 exit( ) 시스템 콜을 호출하고, 그렇지 않을 경우 file\_close( ) 함수를 호출하여 해당 파일을 닫고, 파일 디스크립터 테이블의 해당 원소도 NULL로 초기화해준다.  
      
    5. userprog/syscall.c sys\_filesize( ) : 현재 프로세스의 파일 디스크립터 테이블에서 인자로 들어온 번호의 파일 포인터를 찾는다. 해당 파일 포인터가 NULL이라면 exit( ) 시스템 콜을 호출하고, 그렇지 않을 경우 file\_length( ) 함수의 반환값을 반환한다.   
      
    6. userprog/syscall.c sys\_seek( ) : 현재 프로세스의 파일 디스크립터 테이블에서 인자로 들어온 번호의 파일 포인터를 찾는다. 해당 파일 포인터가 NULL이라면 exit( ) 시스템 콜을 호출하고, 그렇지 않을 경우 file\_seek( ) 함수를 호출하여 해당 파일의 offset을 변경한다.  
      
    7. userprog/syscall.c sys\_tell( ) : 현재 프로세스의 파일 디스크립터 테이블에서 인자로 들어온 번호의 파일 포인터를 찾는다. 해당 파일 포인터가 NULL이라면 exit( ) 시스템 콜을 호출하고, 그렇지 않을 경우 file\_tell( ) 함수의 반환값을 반환한다.  
      
    8. userprog/process.c findLoadFailedEntry( ) : 부모 프로세스가 저장하고 있는 자식 프로세스 리스트에서 파일 적제를 실패한 프로세스가 있는지 여부를 확인한다. 구조체 내에 저장된 is\_loaded 인자가 false일 경우 자식 프로세스가 적제를 실패한 것이므로 true를 반환하고 그렇지 않을 경우 false를 반환한다.

1. **연구 결과**
   1. **Flow Chart**

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

파일 디스크립터 테이블을 초기화 하는 과정을 나타낸 flow chart이다.

스크린샷, 텍스트, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

파일과 관련된 시스템콜의 호출을 나타낸 flow chart이다.

텍스트, 스크린샷, 폰트, 로고이(가) 표시된 사진

자동 생성된 설명

시스템 콜 내부에서 open, write, read 시스템 콜에 대한 동기화를 나타낸 flow chart이다.

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

프로세스를 생성하는 과정에서 부모프로세스와 자식 프로세스의 순서를 강제하기 위한 동기화 과정을 나타낸 flow chart 이다.

* 1. **제작 내용**  
     - File Descriptorthreads/thread.h에 선언된 struct thread 자료구조에 struct file \*fd\_list[128]; 을 선언한다. 해당 자료구조는 프로세스마다 가지고 있으며, 파일을 관리하는 자료구조이다. threads/thread.c에서 프로세스가 처음 생성될 때, init\_thread( )함수를 통해 초기화된다. 반복문을 통해 128개의 원소를 모두 NULL로 초기화하여 아무런 파일이 열리지 않은 상태가 된다. 이러한 파일 디스크립터 테이블은 시스템 콜에 의해 관리된다. 또한 프로세스가 종료될 때, 열려있는 파일을 모두 닫아야 한다. 이를 위해 userprog/syscall.c의 sys\_exit( )에서 파일 디스크립터 테이블을 반복문을 통해 순회하여 NULL이 아닌 값, 즉 파일이 열려있는 인자가 있다면 이를 모두 file\_close( ) 함수를 통해 닫아준다. 이를 통해 프로세스가 종료되었음에도 열려 있는 파일이 메모리 공간을 차지하여 발생할 수 있는 메모리 누수를 예방한다.  
       
     - File System Call  
     userprog/syscall.c에서 시스템 콜 관련 함수들을 구현한다. syscall\_handler( )의 스위치 문에서 파일 관련 시스템 콜을 호출한다. lib/syscall-nr.h에 정의된 시스템 콜 번호를 확인하여, SYS\_CREATE, SYS\_REMOVE, SYS\_OPEN, SYS\_CLOSE, SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL case를 추가한다. 각 시스템 콜의 인자가 유효한 주소에 저장되어 있는지를 확인한 뒤, 해당 시스템 콜을 호출한다. 시스템 콜의 반환값이 존재할 경우 f->eax에 저장한다.  
     sys\_read는 기존 표준 입력에 대해서만 구현한 상태이다. 파일 입력을 수행하도록 다음과 같이 추가 구현하였다. 우선 인자로 들어온 파일 디스크립터 번호를 확인하여 2부터 127까지의 값에 대해 현재 프로세스의 fd\_list에서 해당 인덱스에 저장된 파일 포인터 값을 확인한다. 만약 NULL이라면 할당되지 않은 파일에 접근하고자 한 것이므로 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면 file\_read( ) 를 호출하여 인자로 들어온 버퍼에 인자로 들어온 크기만큼의 데이터를 읽어온다. 이후 읽어온 값을 반환한다. 이때 사용한 file\_read( )는 파일 구조체에 저장된 inode를 따라 인자로 들어온 길이만큼 파일에서 읽어 버퍼에 저장하는 함수이다.  
     sys\_write 역시 표준 출력에 대해서만 구현한 상태이다. 파일 출력을 수행하도록 다음과 같이 추가 구현하였다. 우선 인자로 들어온 파일 디스크립터 번호를 확인하여 2부터 127까지의 값에 대해 현재 프로세스의 fd\_list에서 해당 인덱스에 저장된 파일 포인터 값을 확인한다. 만약 NULL이라면 할당되지 않은 파일에 접근하고자 한 것이므로 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면 file\_write( ) 를 호출하여 인자로 들어온 버퍼에 인자로 들어온 크기만큼의 데이터를 출력한다. 이후 출력한 크기를 반환한다. 이때 사용한 file\_write( )는 파일 구조체에 저장된 inode를 따라 인자로 들어온만큼 버퍼에 저장된 값을 파일로 출력하는 함수이다.  
     sys\_create는 파일을 생성하는 시스템 콜이다. 인자로 들어온 문자열을 이름으로 하는 파일을 인자로 들어온 크기로 생성한다. 만약 문자열이 NULL이라면 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면, filesys\_create( ) 함수를 호출하고, 파일 생성 성공 여부를 반환한다. filesys\_create( )는 인자로 들어온 조건에 따른 파일을 생성하는데, 이름이 만약 존재하거나 내부 메모리 할당에 실패한 경우에도 false를 반환한다.  
     sys\_remove는 파일을 삭제하는 시스템 콜이다. 인자로 들어온 문자열을 이름으로 하는 파일을 삭제한다. 만약 문자열이 NULL이라면 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면, filesys\_remove( ) 함수를 호출하고, 파일 생성 성공 여부를 반환한다. filesys\_remove( ) 인자로 들어온 조건에 따라 파일을 삭제하는데, 이름이 존재하지 않거나 내부 메모리 할당에 실패할 경우 false를 반환한다.  
     sys\_open은 파일은 여는 시스템 콜이다. 인자로 들어온 문자열을 이름으로 하는 파일을 오픈한다. 만약 문자열이 NULL이라면 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면, filsys\_open( ) 함수를 호출하여 파일 포인터를 반환받는다. 파일 포인터가 null이라면 해당 파일을 열지 못하는 것이므로 -1을 리턴한다. 이후 해당 프로세스의 fd\_list를 순회하며 비어있는 원소에 해당 파일 포인터를 저장하고, 해당 fd 인덱스를 반환한다. 만약 리스트를 모두 순회하는 동안 빈 곳을 찾지 못한다면 -1을 리턴한다. 또한 파일이 실행되는 동안 현재 프로세스에 대한 파일이 수정되는 것을 막기 위해, file\_deny\_write 함수를 통해 파일의 변경을 막는다. filesys\_open은 주어진 이름을 가진 파일을 여는 함수로, 해당 이름을 가진 파일이 없거나 메모리 할당에 실패할 경우 null을 반환한다. file\_deny\_write는 파일 구조체에 선언된 deny\_write 변수를 true로 할당하여 해당 파일에 대한 수정을 막는다.  
     sys\_close는 파일을 닫는 시스템 콜이다. 인자로 들어온 파일 디스크립터 번호를 현재 프로세스의 fd\_list의 원소로 접근한다. 만약 해당 인덱스의 원소가 NULL이라면 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면, file\_close( ) 함수를 호출하여 파일을 닫는다. 이후 해당 원소를 NULL로 초기화한다. **file\_close( ) 함수는 할당된 파일에 대해 file\_allow\_write( ) 함수를 호출하여 다시 수정 가능한 상태로 바꾼 뒤, 해당 메모리를 반환하여 파일을 닫는다.  
     sys\_fliesize는 파일의 크기를 알아내는 시스템 콜이다.** 인자로 들어온 파일 디스크립터 번호를 현재 프로세스의 fd\_list의 원소로 접근한다. 만약 해당 인덱스의 원소가 NULL이라면 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면, file\_length( ) 함수를 호출하여 파일의 크기를 얻어와 반환한다. file\_length( )는 파일 구조체에 저장되어 있는 파일의 크기르 반환하는 함수이다.  
     sys\_seek은 파일의 오프셋을 바꾸는 시스템 콜이다. 인자로 들어온 파일 디스크립터 번호를 현재 프로세스의 fd\_list의 원소로 접근한다. 만약 해당 인덱스의 원소가 NULL이라면 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면 file\_seek 함수를 통해 인자로 들어온 position으로 파일의 offset을 이동한다. file\_seek는 파일에 저장된 pos를 인자로 들어온 position으로 바꾼다.  
     sys\_tell은 파일의 오프셋을 알아내는 시스템 콜이다. 인자로 들어온 파일 디스크립터 번호를 현재 프로세스의 fd\_list의 원소로 접근한다. 만약 해당 인덱스의 원소가 NULL이라면 exit(-1) 시스템 콜을 호출한다. 그렇지 않다면, file\_tell 함수를 통해 파일의 현재 오프셋을 알아낸다. file\_tell( ) 함수는 파일 구조체에 저장되어 있는 pos 값을 반환하여 Offset을 알 수 있도록 한다.  
       
     - File Synchronization  
     userprog/syscall.h 에 load\_file\_lock 전역 변수를 선언하였다. 이후 userprog/syscall.c의 syscall\_init( ) 함수에서 lock\_init( )을 통해 락을 초기화한다. 이후 sys\_read와 sys\_write 함수에 진입하자마자 lock\_acquire( )을 통해 락을 걸어준다. 이후 해당 함수 실행이 끝나면, lock\_release( )를 통해 락을 풀어주어 다른 프로세스 혹은 함수에서 해당 락에 접근할 수 있도록 한다. 이를 통해 Read/write problem을 예방할 수 있다. 또한 sys\_open에서 filesys\_open을 실행하기 위해 락을 걸어주어야 한다. 이는 해당 파일을 write하고 있을 경우 file을 열게 되면 동기화 문제가 발생할 수 있기 때문이다. 이를 깨닫고, filesys\_open앞뒤로 락을 걸어주고 풀어주었다. lock\_init( ) 함수는 세마포어를 1로 설정해준다. lock\_acquire( )은 sema\_down( )을 통해 세마포어를 소유하도록 한다. 이로 인해 다른 함수가 락에 접근하지 못하고 대기하도록 한다. lock\_release( )는 sema\_up( )을 통해 세마포어를 놓아준다. 이로 인해 다른 함수가 락에 접근할 수 있도록 한다.  
       
     파일 동기화에 관련해서 위와 같은 동기화 과정만 필요하다고 생각하였지만, 모든 테스트를 통과하지 못하였다. 이러한 원인이 다른 곳에서 동기화 문제가 발생하는 것이라 생각하였다. 특히 multi-oom 테스트가 통과되지 못하였는데, 이는 파일을 올바르게 로드하지 못하는 자식 프로세스가 부모 프로세스에게 reaping되지 못하고, 좀비 상태 혹은 고아 상태로 남아있기 때문이라고 생각했다. 이를 해결하기 위해 semaphore를 사용하였다. 부모 프로세스의 threads/thread.h 의 struct thread에 load\_sema를 선언하였다. 또한 자식 프로세스의 적재 성공 유무를 나타내기 위해 threads/thread.h의 struct thread에 is\_loaded를 추가했다. 자식이 부모의 세마포어에 접근할 수 있도록 부모 프로세스를 저장하는 struct thread \* parent 변수도 추가한다. 이후 init\_thread( )에서 위의 3개의 변수를 초기화 한다. userprog/process.c 에서 부모와 자식 프로세스 간의 적재 성공 여부를 저장하고, 실행 순서를 강제한다. 부모 프로세스의 관점에서 process\_execute( ) 함수에서 thread\_create를 통해 자식 프로세스를 생성하고 난 뒤, sema\_down( )을 통해 자식 프로세스의 적재를 기다린다. 자식 프로세스의 적재 과정이 완료된다면, sema\_down이 리턴하게 되고, findLoadFailedEntry( ) 함수를 실행한다. 해당 함수는 현재(부모) 프로세스의 자식 프로세스를 저장한 연결리스트를 순회하며, is\_loaded 변수를 확인한다. 만약 해당 값이 false일 경우 자식 프로세스가 적재에 실패한 것이므로 true를 반환하고, 그렇지 않은 경우 false를 반환한다. 만약 해당 함수에서 true를 반환하였다면, process\_wait( ) 함수를 통해 적재에 실패한 자식 프로세스를 reaping해준다. 자식 프로세스의 관점에서 start\_process( ) 함수에서 load( ) 함수를 통해 해당 프로세스의 메모리를 구성한다. 만약 실패한다면 success 값이 false가 될 것이다. load( ) 함수 이후 sema\_up( )을 통해 구조체에 저장된 부모 프로세스의 load\_sema에 접근하여 세마포어를 1 증가 시켜 부모 프로세스가 실행을 이어나갈 수 있도록 한다. 이후 자식 프로세스가 적재에 실패하였다면, is\_loaded를 false로 할당하고, sys\_exit(-1)을 호출한 뒤, 부모 프로세스의 reaping을 기다리며 프로세스를 종료한다.
  2. **시험 및 평가 내용**
* 텍스트, 스크린샷, 멀티미디어 소프트웨어, 소프트웨어이(가) 표시된 사진

  자동 생성된 설명텍스트, 스크린샷, 패턴, 디자인이(가) 표시된 사진

  자동 생성된 설명
* make check 결과 80개의 테스트가 모두 pass된 것을 확인할 수 있다.