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

담당 교수 / 분반 : 김영재 / 02반

이름 / 학번 :백상욱 / 20190388

개발 기간 : 2023.10.09 ~ 2023.10.29

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술
* 핀토스는 현재 기본적인 base file system을 제공하고 있다. 하지만 User program이 실제로 파일을 사용하기 위해서 필요한 파일 관련 시스템 콜들은 구현되어있지 않은 상황이다. 이번 프로젝트에는 그러한 파일 관련 시스템 콜, 구체적으로 create, remove, open, cloase, filesize, read, write, seek, tell을 구현한다.
* 추가적으로 위의 내용을 구현할 때 발생할 수 있는 다양한 동기화 문제를 해결해야한다. 파일을 열 때, 쓸 때, 읽을 때, 닫을 때에 대해서 다양한 동기화 이슈가 존재하는데 이를 해결한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

기본적으로 프로세스가 파일에 쓰고, 읽기 위해서는 파일 객체만 들어지고 그 객체로 연결을 해주는 File descriptor가 필요하다. 즉 File descriptor는 프로세스 혹은 스레드가 커널에 존재하는 파일 객체로 접근할 수 있도록 해주는 창구이기 때문에 필수적이다. 하지만 현재 핀토스에서는 이것이 구현되어있지 않기때문에 이를 구현해줘야 한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

file과 관련된 system call들, create, remove, open, close, filesize, read, write, seek, tell을 구현해야 한다. 앞서 말했듯 기본적으로 모든 파일 객체는 커널 영역에 존재한다. 그러므로 일반 유저 프로세스가 유저모드로는 접근하거나 사용할 수 없다.그렇기 때문에 파일 객체를 만들고 없애고 파일을 열고 닫고, 읽고 쓰는 다양한 기능들을 유저가 사용하기 위해서 이러한 시스템콜들을 구현해야한다.

3. Synchronization in Filesystem

파일 객체는 단일로 존재한다. 이를 여러 프로세스가 접근할 때는 서로 다른 파일 디스크립터를 통해 접근하는 것이기 때문에 결과적으로 파일 객체는 하나가 존재하게 된다. 이로 인해서 동기화 문제가 발생할 수 있다. 가령 내가 읽고자 하는 파일을 읽고 있을 때, 누군가 쓰게 된다면 원치않는 결과가 나올 수 있다. 또 서로 다른 프로세스가 같이 쓰기를 시도한다면 이또한 동기화 문제가 발생할 수 있다. 추가로 allow/deny write문제라는게 존재하는데, 이는 실행중인 파일에 대해서는 쓰기를 막아야하는 이슈와 관련되어있다. 이러한 다양한 동기화 문제를 해결하지 않으면 얘기치못한 결과를 맞거나 일관성에 문제가 생길 수 있기 때문에 이러한 동기화 문제를 해결해야한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

file 구조체 포인터 배열을 사용한다. 참조하는 파일 구조체가 있고 그를 참고할 수 있는 포인터 배열을 선언하는 것이다. 이때 정적으로 128개 사이즈로 배열을 Thread 구조체 안에 선언한다. 이는 핀토스 메뉴얼에 128개 정도로 지정하는 것이 괜찮다고 했기에 선정한 숫자이다. 그리고 이 배열에 file 포인터를 어디에 넣을지 결정하기 위한 Reference 가 필요하다. 가령 처음 초기화된 프로세스는 pintos에서는 0,1번이 차있고 2번부터 넣어줘야하기 때문에 이 값이 2로 선정되어있다. 이를 int fd로 선언하여 Thread 구조체 안에 넣어둔다.

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: 파일이 올바른 파일이라면 해당하는 파일 구조체를 생성한다. size를 파라미터로 받고 그만큼의 고정된 크기 파일으 만들어낸다. 이때 생성에 성공했다면 True를 실패했다면 False를 반환한다.

remove : create로 만든 파일을 삭제한다. 파일명이 올바르게 들어와서 삭제해냈다면 True를 반환하고 그렇지 않다면 False를 반환한다.

open : 파일을 연다. 파일이 올바른 파일이라면 file descriptor에 연결해준다. 그리고 fd 숫자를 반환해준다. 열리지 않았다면 Fd가 아닌 -1을 반환한다.

filesize : 파라미터로 들어온 Fd와 연결되어있는 파일의 사이즈를 반환한다.

read : 파라미터로 드러온 fd와 연결된 파일에서 파라미터로 들어온 사이즈만큼 읽고 이 내용을 버퍼에 저장한다. 그리고 읽은 만큼의 사이즈를 반환한다.

write : 파라미터로 들어온 fd와 연결된 파일에 대해서 내용을 쓴다. 이때 쓰는 내용은 버퍼에 담겨져서 온다. 그리고 쓴 만큼의 사이즈를 반환한다.

seek : 파라미터로 들어온 fd와 연결된 파일의 offset을 파라미터로 받은 Position으로 바꾼다

tell : 파라미터로 들어온 Fd와 연결된 파일의 offset을 반환한다.

clsose : 파라미터로 들어온 Fd와 연결된 파일을 닫는다.

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

synchronization의 범위는 우선 크게 open, read, write에 대한 문제 그리고 실행 파일에 대한 allow/deny write 문제로 나눌 수 있다. read, write의 경우는 lock 구조체를 이용한다. 파일을 읽고 쓰는 경우에 대해서 서로 상충될 수 있기 때문에 읽기 시작했거나 쓰기 시작한 경우 Lock\_acquire를 통해 lock을 걸어주고 이것이 끝났다면 Lock\_release를 걸어 락을 해제한다. 그리고 allow/deny write의 경우는 파일이 실행되기 시작하는 부분이라고 볼 수 있는 load함수에서 filesys\_open을 한 후, deny\_write을 걸어주고, 이것이 끝나는 지점에 대해서 allow\_write을 걸어서 쓸 수 있도록 해준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성10.09 ~ 10.15 : 기본적인 syscall 내용 작성  
  10.16 ~ 10.23 : 예외처리, 동기화 이슈 해결
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드  
    syscall.c : 프로젝트 1에서 작성한 시스템콜에 더해서 swtich문 안에 syscall number에 따라서 SYS\_CREATE, SYS\_REMOVE, SYS\_OPEN, SYS\_SEEK, SYS\_TELL, SYS\_ FILESIZE, ,SYS\_CLOSE의 경우를 추가해준다. SYS\_READ, SYS\_WRITE의 경우는 1번 프로젝트에서는 stdin, stdout에 대해서만 구현했지만 이번 프로젝트에서는 그 외의 경우에 대해서도 추가로 구현한다. 이렇게 케이스를 만들어낸 후에는 그리고 각각에 create, remove, open, read, write, seek, tell, filesizeclose 함수를 호출하도록 한다. 그리고 이 각각의 함수들을 구현하는 것이 이번프로젝트의 주된 내용이다.  
    process.c : 동기화 이슈를 해결하기 위해서 추가로 수정해줘야하는 코드들이 존재한다. 특히 Load함수 안에서 파일을 열고 닫을 때의 동기화를 위해서 코드를 추가 및 수정해야한다. 그리고 process 중 Load안에서 파일을 열고 파일을 실행하는 부분이 추가되기 때문에 여기서 deny\_write을 걸어주고 allow\_write으로 풀어줘야한다.  
    이외에도 부모가 child를 만들고 Child가 죽는 과정에서 파일에 대해 발생할 수 있는 동기화 문제들을 다양하게 해결하기 위해서 코드를 수정할 필요가 존재한다,
  + 수정하거나 추가해야 하는 자료구조  
    struct thread : A에서 설명했듯 file descriptor를 추가로 구현한다는 것은 결국 프로세스, 특히 핀토스에는 스레드에 각각 File descriptor를 만들어야한다. 이는 thread 구조체 안에 가질 수 있도록 해야하기 때문에 file struct \*fd\_table[128]을추고한다. 그리고 어떤 숫자에 file descriptor를 추가할 수 있는지 확인하기 위해서 int fd;를 추가한다. 만약 fd가2라면 2번 fd에는 file descriptor를 할당할 수 있는 것이다.   
    그리고 struct lock file\_lock;을 syscall.h안에서 추가한다. 이는 read, write, open등 다양한 파일에 대한 synchronization을 해결하기 위해서 추가하는 변수이다. 이 변수에 lock을 acquire하거나 release함으로서 동기화 문제를 해결한다.
  + 수정하거나 추가해야 하는 함수  
    pintos/src/userprog/syscall.c 의 syscall\_handler()를 전반적으로 수정해야한다. 앞서 언급했듯 switch의 경우를 각 syscall\_number에 맞게 확장해준다. 그리고 create, remove, open,close, filesize, seek, tell함수를 추가로 구현한다. 이때 create는 filesys\_create 함수를 , remove함수는 filesys\_remove 함수를, open은 filesys\_open함수를, filesize는 file\_length함수를, tell은 file\_tell함수를, close는file\_close함수를 이용하여 구현한다. 이들은 이미 pintos안에서 자체적으로 만들어진 함수이기 때문에 별도로 만들 필요가 없다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성  
  **텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

  자동 생성된 설명**  
  **텍스트, 스크린샷, 번호, 평행이(가) 표시된 사진

  자동 생성된 설명**  
  **스크린샷, 도표, 라인, 텍스트이(가) 표시된 사진

  자동 생성된 설명**
  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 설명이번 프로젝트에서 주된 내용은 syscall.c 파일의 syscall\_handler의 각 경우에 대해서 코드를 작성하는 것이기 때문에 이를 중점적으로 서술하고 이후에 발생한 다양한 예외 상황, 그리고 동기화 문제를 해결하는데 추가된 코드를 서술한다.  
  SYS\_CREATE : create 함수를 부른다.  
  create : 인자로 const. char file와 initial\_size를 받는다. 이 두개의 파라미터를 filesys\_create에 바로 넣어준다. 이는 pintos내부에 구현된 함수로 첫 번째 입력으로 들어온 이름을 가진 파일을 두 번째 인자로 받은 사이즈의 파일 객체로 만든다. 그리고 이것이 성공했다면 True를 실패했다면 false를 반환한다.SYS\_REMOVE : remove함수를 호출한다  
  remove : 인자로 const char \*file을 받는다. 그리고 이 파일명을 바로 filesys\_remove의 파라미터로 넣어 호출한다. filesys\_remove는 파일명으로 받은 파일을 없앤다. 그리고 성공했다면 True를 아니라면 false를 반환한다.  
  SYS\_OPEN : open함수를 호출한다. open : const char \*file을 인자로 받아서 해당하는 파일을 열어야한다. 이를 위해서는 filesys\_open을 호출하고 이에 앞서 인자로 받은 파일 명을 인자로 넣어준다. filesys\_open은 파라미터로 들어온 파일을 열어서 해당 파일명에 해당하는 파일 객체를 반환한다. 만약 파일이 열리지 않았다면 이 반환값을 NULL이 된다. 그리고 만약 NULL이 반환되었다면 이때 -1값을 return해야한다. open함수의 반환값은 만약 파일을 열어서 file descriptor를 할당했다면 해당 디스크립터 번호를 반환해야하는데 만약 파일을 열지 못했다면 -1을 반환해야한다. 그리고 만약 NULL이 반환되지 않았다면, 즉 파일을 잘 열어서 그 파일 객체를 얻었다면 그것을 thread 구조체의 fd\_table에 연결해줘야한다. 이때 어떤 번호에 연결해야할지는 fd에 저장되어이다. 결과적으로 다음과 같이 할당한다. file\_ptr = filesys\_open(file); cur\_thread->fd\_table[cur\_thread->fd] = file\_ptr;   
  그리고 이렇게 했을 때, thread의 fd 값을 다음으로 조정해줘야한다. 그래서 fd+1을 하고 fd-1값을 반환하도록 했으나 이렇게 했을 때 문제가 있을 수 있다고 생각했다. 원래 OS를 생각했을 때, 비어있는 가장 낮은 fd에 할당해야하는데 이런 원칙이 지켜지지 않을 수 있기도하고 가령 3번이 할당되어있지만 2번 fd는 비어있어서 2번에 할당한 경우 3번에 할당하면 안되고 한번 더 넘어가 4번에 할당해야한다. 이를 위해서 fd를 새롭게 정할 때 현재 descriptor에 fd가 없는 경우에만 할당하도록 했다. 이를 위해서 while문을 돌면서 fd\_table[fd]이 null인 경우까지 fd를 1씩 증가시키면서 찾아나갔다. 그리고 해당 fd가 null이면 멈춘다. 그리고 이전에 할당해놓은 fd값을 따로 저장해서 반환한다.  
  **cur\_thread->fd\_table[cur\_thread->fd] = file\_ptr;   
  cur\_thread->fd\_bitmap[cur\_thread->fd] = 1;   
  return\_fd = cur\_thread->fd;   
  while(cur\_thread->fd\_table[cur\_thread->fd]){   
  cur\_thread->fd += 1;}   
  return return\_fd;**  
  코드는 위와 같이 짜게 된다. SYS\_FILESIZE : filesize함수를 호출한다. filesize: 파라미터로 fd번호를 받는데, 이 번호를 다시 파라미터로 주어서 file\_length를 호출한다. File\_length는 인자로 받은 파일의 사이즈를 반환한다. 하지만 이때 예외처리를 해야한다. 만약 fd 인덱스에 해당하는 file 이 file descriptor table에 없다면 올바르지 못한 경우이기 때문에 exit(-1)을 호출한다. 이후 나오는 시스템콜들 중 인자로 fd번호를 받아오는 경우라면 모두 이번 경우와 같이 file\_descriptor가 있는지 확인해준다.  
  **cur\_fd = cur\_thread->fd\_table[fd];   
  if (!(cur\_fd))  
  { exit(-1); }   
  else {   
  return file\_length(cur\_fd);}**  
  SYS\_READ : 이전에는 fd==0인 경우 즉 stdin에 대해서만 구현했다. 하지만 이번 프로젝트에서는 Fd가 0이 아닌 경우에도 구현한다. fd==0인 경우에는 input\_getc함수를 이용해서 구현하지만 fd가 0이 아닌 경우는 file\_read함수를 이용해서 구현한다. file\_read는 파라미터로 Fd, buffer, size를 파라미터로 주어서 최대 size만큼 읽어서 buffer에 저장한다. 그리고 읽은 bytes만큼을 반환한다. 그리고 이 값을 read함수의 반환값으로 반환한다. 이때 또 앞서 filesize를 구현할 때와 같이 입력으로 받은 fd를 기반으로 file descriptor를 참조했을 때, pointer가 없고 null이라면 exit(-1)을 호출해준다.   
  SYS\_WRITE : 이전에는 fd==1인 경우, 즉 stdout에 대해서만 구현했다. 하지만 이번 프로젝트에서는 fd가 1이 아닌 경우에도 구현한다. fd가 0인 경우는 pubuf함수를 이용해 구현했다. 하지만 이 외의 경우에는 file\_write함수를 이용해서 구현한다. file\_write함수는 fd, buffer, size를 파라미터로 받고 buffer의 내용을 size만큼 Fd에 해당하는 파일에 작성해준다. 그리고 쓴 bytes의 양을 반환하는데, 이 값을 write의 반환값으로 주면된다. 이때 또 앞서 filesize를 구현할 때와 같이 입력으로 받은 fd를 기반으로 file descriptor를 참조했을 때, pointer가 없고 null이라면 exit(-1)을 호출해준다.   
  SYS\_SEEK : seek함수를 호출해준다. seek : 인자로 fd, position을 받는데, 이 fd와 poisition을 file\_seek의 인자로 주어서 해당 파일의 offset을 position으로 바꿔준다. 따로 return값은 존재하지 않는다. 이때 또 앞서 filesize를 구현할 때와 같이 입력으로 받은 fd를 기반으로 file descriptor를 참조했을 때, pointer가 없고 null이라면 exit(-1)을 호출해준다.   
  SYS\_TELL : tell 함수를 호출해준다. tell : 인자로 fd를 받는데, fd를 file\_tell에 입력으로 넣어준다. file\_tell은 해당 filedescriptor의 offset을 반환해준다. 이 반환값을 tell함수의 반환값으로 사용한다. 이때 또 앞서 filesize를 구현할 때와 같이 입력으로 받은 fd를 기반으로 file descriptor를 참조했을 때, pointer가 없고 null이라면 exit(-1)을 호출해준다.   
  SYS\_CLOSE : close함수를 호출한다. close : fd를 입력으로 받아서 해당 index에 해당하는 file\_descriptor를 참고하여서 그 파일 디스크립터를 file\_close 함수의 입력으로 준다. 이 과정을 하면 해당 file은 닫힌다. 별도의 반환값은 없다. 이때 닫아주고 난뒤에 우선 file\_descriptor의 fd에 해당하는 값을 null로 할당해 비워준다. 그리고 스레드 구조체의 fd번호를 바꿔줘야한다. 해당 Fd가 더 작은값이라면 그 번호부터 fd를 할당할 수 있도록 fd 번호를 바꿔준다. 그리고 또 앞서 filesize를 구현할 때와 같이 입력으로 받은 fd를 기반으로 file descriptor를 참조했을 때, pointer가 없고 null이라면 exit(-1)을 호출해준다.   
  예외처리 : 앞서서 fd 번호를 입력으로 받는 syscall의 경우는 fd에 해당하는 File\_descriptor가 null인지 확인했다. 이 외에도 fd번호가 0보다 작거나 128이상이라면 이 또한 잘못된 참조이기 때문에 Exit할 수 있도록 해준다.  
  그리고 추가로 file\_name을 입력으로 받아서 호출하는 syscall(create, remove, open)의 경우는 file\_name이 잘못된 값이 들어오는 경우가 있기 때문에 이 경우를 예외처리한다. check\_file\_null이라는 함수를 추가한다. 이 함수는 const char \*file을 파라미터로 줘서 그 값이 NULL이라면 exit(-1)을 호출한다. 반면 NULL이 아니라 포인터 값이 들어오고 그 주소가 굉장히 이상한 주소가 들어오는 경우가 존재한다. 이러한 경우 또한 예외처리를 해야하는데, 이는 이전 프로젝트 1에서 구현했던 check\_addr\_valid를 사용해서 예외처리를 할 수 있다. 이는 포인터로 들어온 값을 저장하고 있는 주소를 가지고 역참조하여 해당 주소가 올바른지 확인하는 함수이다.   
  file\_pointer = (char\*)(f->esp + 8);   
  check\_addr\_valid(pagedir,\*(void \*\*)file\_pointer);  
  코드는 위와 같이 작성한다. 그리고 이러한 코드를 각각 create, remove, open. read, write에서 불러준다.   
  동기화 : 기본적으로 동기화를 하는 방법은 lock\_acquire(&file\_lock)을 걸어주고 lock\_release(&file\_lock)으로 풀어주는 방식이다. 그리고 동기화가 필요한 부분은 syscall 중 open, read, write이다. open의 경우 파일을 여는 중 다른데가 읽거나 쓰는 것이 발생할 때 내용이 바뀌는 등의 이슈가 생길 수 있기 때문에 동기화를 한다. 그리고 filesys\_open을 하기전에 lock\_acquire를 걸어준다. 그리고 Return이 발생하기 이전마다 모두 lock\_relase를 해줘서 return하기 전에 lock이 풀릴 수 있게 해준다. 그리고 read, write의 경우도 읽고 쓰는 과정에서 얘기치 못하게 동기하문제가 발생할 수 있다. 원래 read, read인 경우에는 별도의 동기화가 필요없지만 이번 프로젝트에서는 별도의 자료구조를 두지 않고 read, read에 대해서도 동기화를 수행한다. read함수에서 fd에 따라 조건을 나누고 input\_getc혹은 file\_read로 파일을 읽기전에 file\_lock을 건다. 그리고 return이 발생하는 곳 마다 lock\_release로 락을 풀어준다. write에 대해서도 동일하다 fd에 따라 조건을 나누기 전, 그리고 각각 pubuf, file\_write으로 쓰기 전에 Lock을 걸어주고 return전에 lock을 release해준다. 다음으로는 allow\_write, deny\_write문제를 해결해줘야한다.   
  이는 syscall.c가 아닌 process.c에서 특히 load함수에서 해결해준다. load함수에서는 filesys\_open을 하고 이 filesys\_open내용이 NULL이 아닌 경우 이를 기반으로 파일을 실행한다. 그렇기 때문에 이 부분에서파일을 열고 실행하게 되므로 이 직전에 file\_deny\_write을 걸어준다. 그리고 file\_allow\_write은 별도로 진행할 필요가 없다. 소스코드를 조금 더 찾아봤을 때, file\_close에서 file\_allow\_write을 모두 해주기 때문에 실행이 끝나고 파일을 닫아주는 과정이 들어가있기 때문에 굳이 별도의 file\_allow\_write을 써줄 필요가 없다.   
  위의 내용을 구현했을 때 테스트 케이스 중 통과가 안되는 테스트케이스가 존재했고, 이는 ROX 계열 케이스, sem read, write, multi -oom이었다. 이에 대해서 child가 만들어지고 죽는 과정에서 제대로 wait이 안되었거나 동기화가 잘 해결되지 않은 문제라고 판단하여 추가로 이를 찾아내어 수정했다.   
  우선 load함수에서 파일을 열고 닫는 경우 또한 파일을 열어주는 경우로 이에 대한 동기화가 필요하다고 생각하여 앞서 syscall.c에서 선언한 lock을 extern으로 선언해서 load함수에서도 사용할 수 있도록 했다. 그리고 load 함수에서 filesys\_open을 하기전에 lock\_acquire을 진행하고, 처리해야할 내용을 모두 수행한 뒤, lock\_release를 풀어준다. 이렇게 구현했을 경우, lock\_release가 발생하는 경우도 있지만 파일을 열어주는데 실패했을 경우 바로 done으로 오게 되는데, 이때는 lock을 이부분에서 처리해줘야한다. 이를 따로 처리하기 위해서 lock\_held\_by\_cuurent\_thread라는 함수를 사용해서 이를 조건문에 넣어서 만약 Lock이 있다면 lock\_release를 하도록 바꿔준다. 하지만 이렇게 해도 child가 wait이 안되는 경우 혹은 동기화가 잘안되는 부분이 존재한다. 우선 exit이 되는 경우를 본다. 이 경우는 file descriptor를 닫아주지 않고 죽는다. 이것이 stack에 문제를 일으킬 수 있다고 생각하여서 exit에서 모든 fd를 닫아주도록 수정한다. 그리고 현재 실행하려는 파일을 여는 경우는 file descriptor에 추가되지 않는다. 하지만 이 또한 exit될때 닫아줘야한다. 이를 위해서 struct 구조체에 self\_fd\_table을 추가하고 self\_fd숫자를 추가한다.그리고 load에서 파일을 열때 self\_fd\_table에 추가해주고 Exit이 되는 경우에 이를 다 지워준다. 다음으로는 wait을 추가로 수정한다. thread를 만들어내었지만 자식이 load하기 전에 부모가 먼저 죽는 이슈가 존재할 수 있다. 이를 보완해줘야한다. 즉 부모가 너무 빨리 죽지 않게 일단 child가 load될때까지는 기다리도록 멈추게 해야한다. 이때 이 semaphore는 부모의 것을 참조해야하므로 struct thread \* parent를 thread 구조체에 추가한다. init thread에서 t->parent = running\_thread()로 만들어진 경우. 부모와 연결한다. 그리고 start\_process를 하는 경우 thread\_current->parent->oom\_sem을 sema\_up 해준다. 이때부터 부모가 비정상 종료하지 못하게 한다. 그리고 filesys\_open을 다해주고 thread\_create를 한뒤에 sema\_down(&thread\_currnet()->oom\_sem)을 시켜서 semaphore를 마지막에 풀어준다. 하지만 이렇게 할 경우 프로세스가 아예 lock이 걸리는문제가 발생한다. 이는 앞서 load에서 걸었던 Lock\_release가 이 sema\_down이후에 있으면 lock이 서로 충돌해서 데드락이 걸리기 때문이다. 따라서 lock\_release위치를 옮겨준다. 다 해결되었지만 multi-oom이 여전히 해결되지 못했다. 이에 대해서 load를 했을 때, filesys\_open이 되지 않으면 나는 스레드를 만들지 않고 retur -1을 한다. 이경우에는 child가 만들어지지않았다고 봐야한다. 하지만 코드 입장에서는 이걸 child를 만들어졌다고보기 때문에 wait이 되어야한다. 이에 대해서 별도로 wait을 해주기 위해서 thread 구조체에 flag라는 변수를 추가해서 만약 잘 load되지 않았다면 flag를 1로 만든다. 그리고 process\_execute에서 flag가 1인 process list를 찾아서 해당 process를 process\_wait에 넣어준다.  
  이로서 모든 동기화와 wait이슈를 해결했다.
  1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부  
    
  텍스트, 스크린샷이(가) 표시된 사진

  자동 생성된 설명