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

담당 교수 / 분반 : 김영재 / 수금

이름 / 학번 : 임정호 / 20171682

개발 기간 : ’21.10.07 ~ ’21.10.10

1. **개발 목표**

file system과 관련된 system call을 구현하는 것이다. 구현해야 할 System call에는 create, remove, open, close, filesize, read, write, seek, tell이 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

우선 thread가 파일에 접근하고 특정 작업을 수행할 때, 필요한 파일 포인터 변수가 필요하다. 그렇기 때문에 thread는 파일 포인터 변수를 사용할 수 있어야 하고, 같은 파일을 접근하더라도 각 thread마다 fd는 다른 값을 가질 수 있어야 한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

우선 프로젝트1에서 STDIN과 STDOUT만 고려해서 read와 write를 구현했는데, 프로젝트2에서는 모든 fd에 대해서 가능하게끔 해야 한다. create, remove, open, close, filesize, read, write, seek, tell의 구현이 모두 끝나면 pintos에서 make check을 했을 때 수행하는 80개의 테스트를 통과할 수 있다.

3. Synchronization in Filesystem

pintos는 internal synchronization이 되지 않는 상태이다. 동일한 file에 대한 concurrent한 접근은 file을 망가뜨릴 수 있고, 불완전한 수행으로 이어지기 때문에, 하나의 프로세스만 파일을 접근할 수 있게끔 synchronization을 구현해야 한다. 프로젝트 1에서 부모 thread와 자식 thread 사이에서 synchronization을 위해 semaphore을 사용했고, 프로젝트 2에서는 lock과 semaphore를 이용할 것이다.

* 1. **개발 내용**

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

우선 struct file\* fd[]의 포인터 배열로 thread가 관리할 수 있도록 하였다. 동일한 file을 접근하더라도, thread마다 fd가 다를 수 있기 때문에 thread의 구조체 변수로 file을 접근할 수 있게 하였다. pintos manual의 FAQ에 thread는 최대 128개의 fd를 가질 수 있다고 기술되어 있어서 그 크기는 128로 하였다. 동적 할당을 하게 되면 혹시 모를 memory deallocation의 문제가 생길 것 같아서 정적으로 길이가 128인 포인터 배열을 선언하였다.

1. 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: 새로운 file을 매개변수로 받은 size bytes만큼의 크기로 생성한다. 성공하면 true, 실패 하면 false를 리턴한다.

remove: 매개변수로 받은 이름을 가진 file을 삭제한다. 성공시 true, 실패시 false를 리턴한다.

open: 매개변수로 받은 이름의 file을 open한다. open에 성공하면 해당 file descriptor를 리턴하고, 실패하면 -1을 리턴한다.

close: 매개변수로 받은 fd의 file을 close한다.

filesize: 현재 thread에서 관리하는 fd 중에서 매개변수로 받은 fd의 filesize를 리턴한다.

read: 매개변수로 받은 fd의 file을 size만큼 buffer에 넣어준다.

write: 매개변수로 받은 fd의 file에 buffer에 있는 값을 size만큼 출력한다.

seek: 파일을 읽을 때 사용하는 포인터를 position의 위치로 옮긴다.

tell: 파일을 읽을 때 사용하는 포인터의 위치를 리턴한다.

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

syscall.c에 전역변수로 file\_lock이라는 struct lock 구조체 변수를 선언한다. 어떠한 thread가 filesystem에 접근하는 경우(open, write, read) 오직 하나의 thread만 사용할 수 있게끔 조치를 해야 한다. 이 때 사용하는 것은 project1에서 사용했던 semaphore가 아닌 lock이다. lock과 semaphore의 다른 점은 semaphore는 다른 thread가 sema\_up()을 한 semaphore를 sema\_donw()을 할 수 있다는 점이다. lock은 lock\_acquire()를 한 thread만 lock\_release()를 할 수 있다. semaphore보다 훨씬 엄격한 방법이다. 물론 struct lock의 변수에 semaphore가 있다. lock을 사용해서 file에 대한 접근을 막고, 동일한 thread가 open한 파일을 write하는 경우를 막기 위해 file 구조체의 deny\_write 변수를 사용한다. 마지막으로 부모 thread는 자식 thread가 load()가 끝날 때까지 기다려야 한다. pintos manual exec 함수 설명에 보면 부모 프로세스는 exec에서 child가 load()를 성공했는지 알기 전까지는 종료할 수 없다고 나와있다. 이를 위한 semaphore변수를 thread 구조체에 선언하여 활용한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

’21.10.07 : create, remove, close, filesize, seek, tell의 syscall 구현

’21.10.08 : open, close, read, write의 전반적인 구현

’21.10.08 ~ ’21.10.10 : make check 중 fail에 대한 디버깅(모든 synchronization 문제는 여기서 해결)

* 1. **개발 방법**
  + 수정해야 하는 소스코드
* 일단 수정해야 하는 소스코드와 함수의 구분이 조금 헷갈리는데, .c파일 기준으로 설명하겠다. 우선 syscall.c와 process.c를 수정해야 한다. 이유는 file system과 관련된 system call의 추가와 file과 관련된 synchronization 구현이다. 또한 file descriptor를 초기화하기 위해 thread.c를 수정해야 한다.
  + 수정하거나 추가해야 하는 자료구조

우선 thread 구조체를 수정해야 한다. thread.h.에 2가지 변수만 추가하면 된다. struct file\* fd[128]과 struct semaphore for\_parent이다. file system에 접근하기 위한 포인터 배열이고, load()가 잘 끝났는지 확인하기 위한 semaphore 변수이다. syscall.c에 전역변수로 struct lock 변수를 선언한다. file system에 접근하는 thread를 오직 한 개의 thread로 제한하기 위함이다.

* + 수정하거나 추가해야 하는 함수

수정해야하는 함수: syscall\_handler(), syscall\_init(), read(), write(),

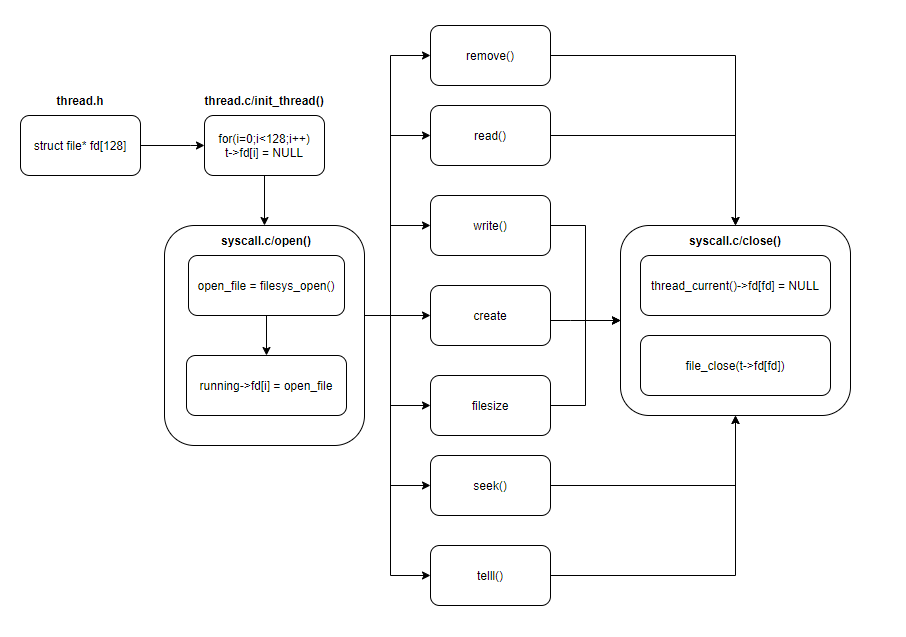
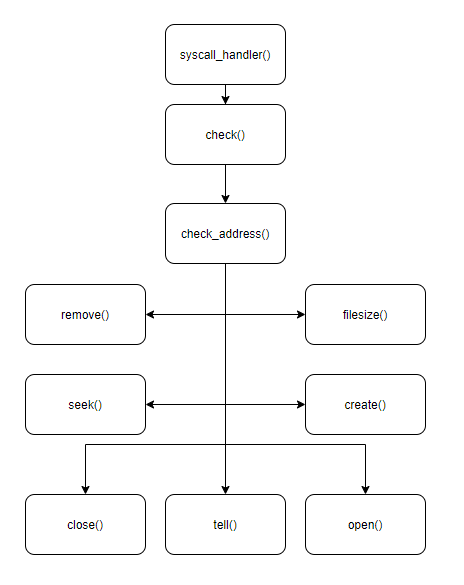
,init\_thread(), process\_execute(), start\_process(), process\_exit(), load()

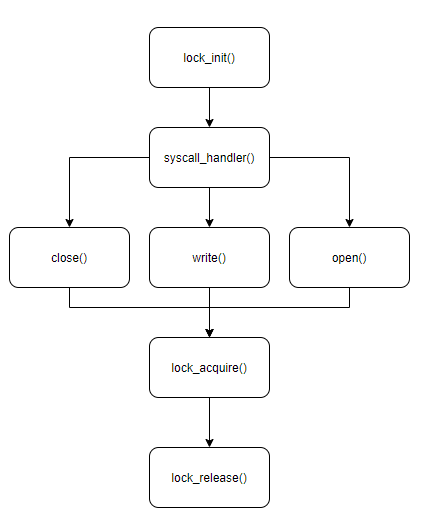
우선 부모 프로세스가 자식 프로세스 load() 제대로 수행하는지 기다리기 위해 세마포를 thread\_create() 호출 후에 sema\_down()으로 기다리는데, 어떤 자식을 기다리는지 알아야 하기 때문에, thread\_create()이 리턴한 tid를 project1에서 구현한 자식 thread list에서 해당 tid를 기다리게 한다. process\_wait()에서 기다렸던 방식과 동일하다. 자식 thread느 start\_process()에서 load()를 호출하고 나서 본인의 세마포 변수를 sema\_up을 해서 부모 에게 알려준다. syscall\_handler()에서 file에 영향을 줄 수 있는 작업을 수행할 때 사용할 세마포 변수를 syscall\_init()에서 lock\_init()으로 초기화한다. 이후 open(), read(), write()

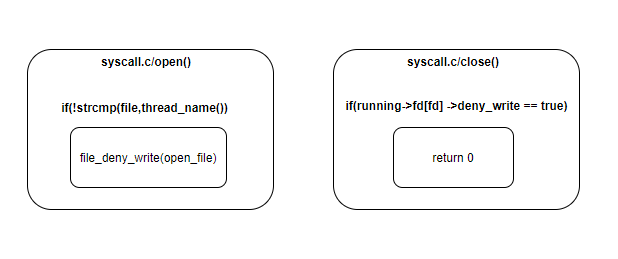
함수의 시작과 끝에 lock변수를 잠그고, 해제한다. 이런 방식으로 file system의 concurrent한 접근을 막을 수 있다. 이제 concurrent한 접근을 막았으면, 실행되고 있는 파일에 write를 하지 못 하게 막아야 한다. 이 때 사용하는 방법이 file 구조체의 변수 deny\_write를 사용하는 것이다. open할 때, thread의 이름과 접근하는 파일의 이름을 비교하는데, 둘의 이름이 같으면 file\_deny\_write()를 호출하여, 해당 파일을 write하지 못 하게 하는데, write()에서 파일의 deny\_write 변수를 확인하기 때문이다.

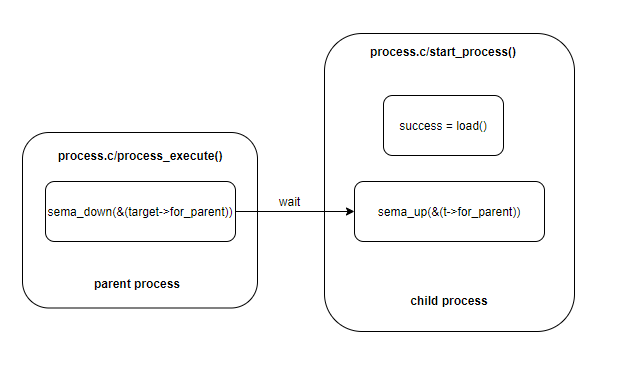
추가해야 하는 함수: tell(), seek(), open(), close(), filesize(), remove(), create()

프로젝트 2를 구현하면서 작성한 코드가 프로젝트 1때 작성했던 코드와 중복되는 상황이발생했다. 예를 들면, check\_address(), 자식 쓰레드 리스트에서 tid의 자식 찾는 부분 등. 이런 중복되는 부분을 함수로 구현하여 가독성과 재사용성을 높였다.

1. **연구 결과**
   1. **Flow Chart**
2. **File descriptor**
3. **System calls**
4. **Synchronization in Filesystem**







* 1. **제작 내용**
* **1. File descriptor**

우선 thread.h에 struct file\* fd[128]을 선언해주었다. 초기화는 thread.c에서 init\_thread()에서 한다. struct fiel을 포인터로 선언하고, 동적으로 필요할 때 메모리 할당을 하려고 했지만, memory deallocation 이슈가 있을 듯 하여 안전하게 정적으로 길이 128의 포인터 배열로 선언하였다.





* **2. System calls**

synchronization이 필요하지 않은 syscall 함수들은 명세서에 나와있던 kernel api를 이용하였다. 내가 크게 구현할 부분은 없었고, filesys/filesys.h와 filesys/file.h에 정의되어 있는 kernel 함수들과 연결만 하면 웬만한 test가 통과되었다. 그래서 synchronization을 크게 고민하지 않아도 되는 함수들, create(), remove(), close(), filesize(), seek(), tell()은 kernel에 있는 함수들과 연결하니 별 문제없이 끝났다.

**create**: filesys.c에 구현되어 있는 filesys\_create()을 호출하였다. file이 null인 경우는 exit(-1)로 처리하였는데, create-null 테스트를 통과하기 위함이었다.

**remove**: filesys.c에 구현되어 있는 filesys\_remove()를 호출하였다.

**filesize:** 매개변수로 받은 fd를 이용하여 현재 쓰레드에서 관리하는 file descriptor 배열 중 fd에 해당하는 파일 포인터를 file.c에 구현되어 있는 file\_length()의 매개변수로 넘겨 호출했다.

**seek:** 매개변수로 받은 fd를 이용하여 현재 쓰레드에서 관리하는 file descriptor 배열 중 fd에 해당하는 파일 포인터를 file.c에 구현되어 있는 file\_seek()의 매개변수로 넘겨 호출했다.

**tell**: 매개변수로 받은 fd를 이용하여 현재 쓰레드에서 관리하는 file descriptor 배열 중 fd에 해당하는 파일 포인터를 file.c에 구현되어 있는 file\_tell()의 매개변수로 넘겨 호출했다.

**close**: 매개변수로 받은 fd를 이용하여 현재 쓰레드에서 관리하는 file descriptor 배열 중 fd에 해당하는 파일 포인터를 NULL로 초기화하고, file.c에 구현되어 있는 file\_close()에 매개변수로 파일 포인터를 넘겨 호출했다.

**open**부터 **write**는 synchronization이 필요하다. 이 부분에 대한 설명은 3번 Synchronization in File system에서 설명하겠다.

**open**: 매개변수로 받은 file이 null인지 검사한다. null인 경우 exit(-1)을 호출하고, 아닌 경우 file을 filesys\_open()으로 open을 하는데, 이 때 해당 파일이 존재하지 않으면, -1를 리턴하고 종료한다. 이 정도만 에러를 처리해주면 make check 테스트 중에서 open 시리즈는 대부분 해결된다. 그리고 현재 쓰레드에서 관리하는 file descriptor를 3부터 찾는다. (0,1,2는 STDIN, STDOUT, STDERR로 예약되어 있기 때문) for loop을 이용해 포인터 변수 fd[i]가 NULL인 경우를 찾아 붙여준다. 그리고 붙여준 fd의 인덱스를 리턴한다. 실패하면 -1을 리턴한다.

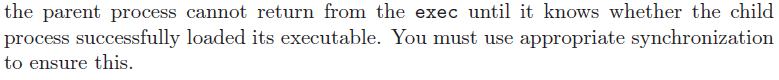
**read:** 프로젝트1에서 STDIN인 경우만 처리했는데, 이제 모든 fd에 대해 처리해야 한다. 현재 쓰레드가 관리하는 file descriptor배열에서 매개변수로 받은 fd에 해당하는 포인터를 찾아 file.c에 구현되어 있는 file\_write()를 호출하고, 그 반환된 값을 리턴한다. read시리즈의 테스트는 read-bad-ptr에서 시간이 조금 걸렸는데, test code를 뜯어보니, buffer는 kernel address는 아니지만, buffer가 가리키는 곳에 있는 주소가 kernel address였던 것이다. 그래서 read() 내부에 check\_address()를 호출하여 해당 문제를 해결할 수 있었다.

**write:** 프로젝트1에서 fd가 STDOUT인 경우만 처리했는데, 이제 모든 fd에 대해 처리해야 한다. 현재 쓰레드가 관리하는 file descriptor에서 매개변수로 받은 fd에 해당하는 포인터를 찾아 file.c에 구현되어 있는 file\_read()를 호출하여 그 반환된 값을 리턴한다.

* **3. Synchronization in File system**

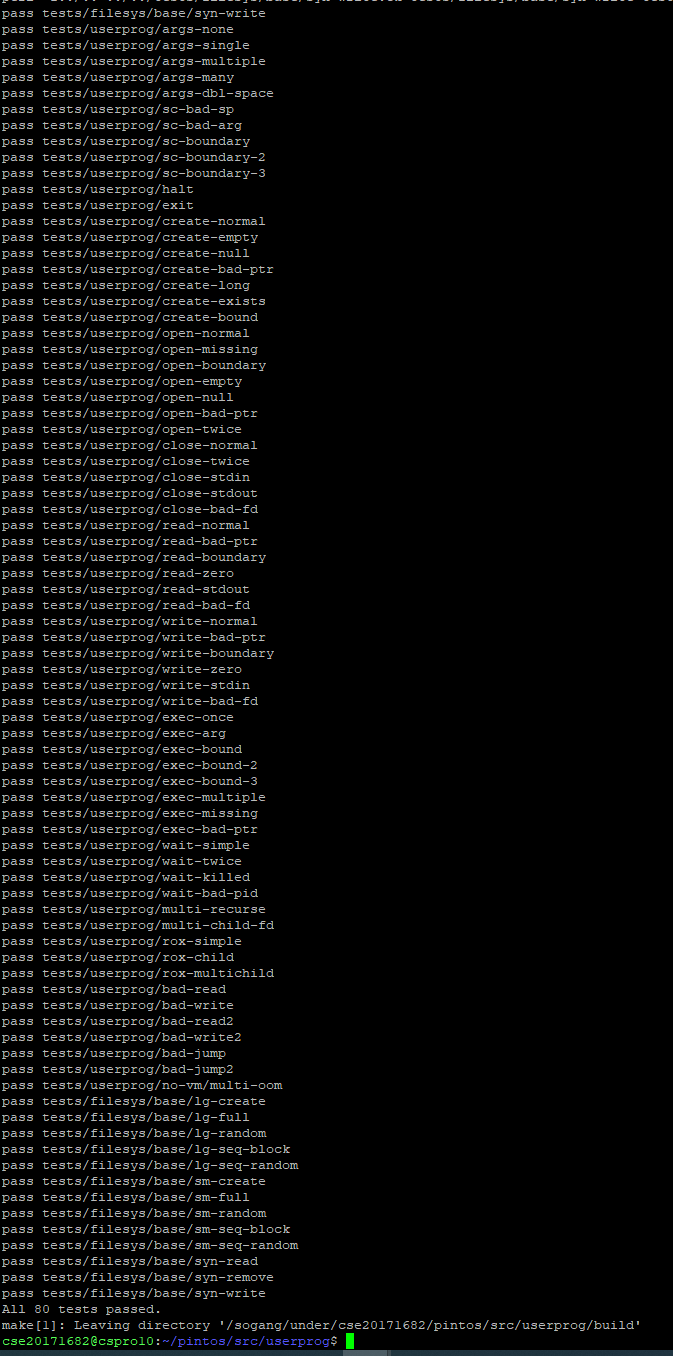
우선 어떠한 쓰레드가 file에 직접적으로 영향을 주거나, 간접적으로 영향을 받을 수 있는 작업들에 대해 synchronization이 필요하다. 예를 들면 open()를 하고 있는데, write()을 한다거나, read()를 하고 있는데 write()하면 안 된다. 이 문제를 해결하기 위해 file 작업에 대해 lock을 걸어야 한다. 처음에는 file 구조체에 lock변수나 세마포 변수를 두어 file system 전체보다는 각각의 file에 대해 처리하고 싶었지만, filesys디렉토리는 건들 필요가 없다고 전체 강의와 pintos manual에서 설명했기에 어쩔 수 없이 전역 변수로 struct lock 변수 file\_lock을 선언하여 사용하였다. open(), read(), write()의 시작과 끝( 종료 되기 전)에 lock\_acquire(&file\_lock)과 lock\_release(&file\_lock)을 호출하여 다른 쓰레드의 concurrent한 접근을 방지했다.

또한 파일이 실행되고 있는 동안에 해당 파일에 대한 write를 막아야 했다. 이 문제는 전체 강의에서 설명했던 것처럼 struct file의 구조체 변수 deny\_write와 file.c에 정의되어 있던 file\_deny\_write()를 이용했다. 우선 open()에서 매개변수로 받은 file의 이름과 현재 쓰레드의 이름을 비교한다. 현재 쓰레드의 이름은 테스트 코드 상 실행파일의 이름을 가진다. 예를 들면 echo를 실행하면 echo가 thread의 이름이 되는 것이다. 그래서 현재 쓰레드의 이름이 open하려는 file의 이름과 같은 경우라면, 해당 파일에 대한 쓰기 접근을 막아야 한다. 그래서 file\_deny\_write()를 호출하여 deny\_write를 true로 설정한다. deny\_write가 true로 되면 이제 이 변수는 write()에서 사용된다. write()에서 file\_write()를 호출하는데, 이 kerenl 함수에서 deny\_write를 확인하여 true이면 알아서 0으로 리턴한다.

여기까지 처리를 하면 rox의 문제도 해결되고 대부분의 문제가 해결 된다. 하지만 syn-read와 syn-write그리고 multi-oom 문제가 해결되지 않는다. syn-read와 syn-write, 그리고 multi-oom은 이번 프로젝트의 3대장이였다. 디버깅만 총 2일이 걸렸다. 내가 프로젝트 1번에서 놓쳤던 부분이 있었다. pintos manual의 exec()에 대한 설명을 쭉 읽다보면 아래와 같은 설명이 있다.

읽어보면 부모 프로세스는 자식 프로세스가 load를 성공적으로 수행했는지 알 때까지는 리턴을 하면 안된다고 한다. manual을 읽다보니 이 부분이 눈에 걸렸고, 이 부분을 해결하기 위해 thread 구조체에 세마포 변수를 하나 추가했다. for\_parent. 이는 자식 프로세스가 load를 잘 수행하면 sema\_up을 해주고, 부모 프로세스는 process\_execute()에서 thread\_create()을 호출하고 나서 sema\_down()을 하며 기다린다. 일단 부모 프로세스는 어떤 자식의 load를 기다려야 하는지 알아야 하기 때문에 thread\_create()에서 리턴한 tid를 단서로 평소 관리하던 자식 thread 리스트에서 해당 tid를 가진 thread를 찾는다. 이 부분이 프로젝트 1번에서 process\_wait()부분에 구현했던 코드와 거의 중복되어서 이 부분을 get\_child(tid)로 함수화 하였고, 이 함수는 tid를 가진 쓰레드를 리턴하고, 탐색에 실패한 경우 NULL을 리턴한다. 리턴 받은 쓰레드의 for\_parent 세마포 변수를 sema\_down()으로 기다리면 된다. 이러면 syn-read와 syn-write의 문제가 해결되었던 걸로 기억이 난다.

마지막 문제는 multi-oom이다. 이 문제가 제일 까다로웠다. oom은 out of memory의 약자로 메모리 누수가 발생하는지를 테스트하는 것이다. 이는 자식 프로세스를 제대로 부모 프로세스가 회수를 하는지, 동적 할당을 한 메모리를 제대로 해제를 하는지를 신경 써야 하는 것이다. 동적 할당의 메모리 해제 문제는 프로젝트1에서 반은 해주고, 반은 안 해준 것을 확인했고, malloc()을 사용한 변수를 찾아 free()해주었다. 그러고 나서 다시 multi-oom을 테스트해봤지만, 여전히 안 된다. 이제 남은 문제는 제대로 자식 프로세스를 회수하는 것이다. 그래서 확인해보았더니 비정상적으로 죽는 자식 프로세스를 부모가 모르는 경우가 발생했다. 자식 프로세스가 언제 비정상적으로 죽는지 print()를 해보며 추적한 결과 start\_process()에서 load()가 fail될 때임을 찾았다. 그래서 원래 주어진 코드에서는 success가 false인 경우 thread\_exit()을 호출하는 코드이었는데, 이 쓰레드가 비정상적으로 종료되어, 부모 프로세스가 알아야한다고 체크하기 위해, 해당 thread의 exit\_status를 -1로 바꿔주고, thread\_exit()을 부르게 하였다. 그리고 나서 이 자식 프로세스를 어떻게든 부모 프로세스가 알아차리고 회수를 해야 좀비 프로세스 문제를 해결할 수 있기 때문에 process\_execute()에서 자식의 load가 무사히 끝냈는지 기다리는 부분에서 부모가 자식 쓰레드를 확인하면서 자신도 모르는 사이에 비정상적으로 종료된 자식 프로세스는 없는지 검사한다. 이 부분의 코드를 retrieve\_zombies()로 함수화하였다. 이 함수는 비정상적으로 종료된 자식이 있으면 그 자식을 쓰레드 리스트에서 제거하고 1을 리턴하게된다. 1을 리턴하면 부모 프로세스는 자식이 비정상적으로 죽었기에 -1 리턴으로 응답하며 종료하게 된다. 여기까지 진행을 하고 다시 multi-oom을 테스트해보았다. 여전히 되지 않는다. 이 문제는 비유를 하자면 자식 프로세스가 죽는걸 부모가 알아도 자식 프로세스가 사용하던 유품까지 정리는 못 했던 점에서 비롯된다. 즉, 자식 프로세스에서 사용했던 file들을 file\_close()해야 한다. pintos 내에서 생성한 모든 프로세스는 결국 thread\_exit()을 통해 process\_exit()으로 오게 된다. 따라서 이 함수에서 현재 쓰레드가 관리하던 file descriptor를 for loop으로 확인하면서 null이 아닌 경우에 대해 file\_close()를 해주는 식으로 메모리를 회수하였다. 이러고 다시 multi-oom을 테스트했더니 pass를 받을 수 있었다.

* 1. **시험 및 평가 내용**