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

담당 교수 / 분반 : 김영재 / 2반

이름 / 학번 : 정동혁 / 20161640

개발 기간 : 2020/11/03~2020/11/14

1. **개발 목표**

이번 프로젝트의 개발 목표로는 크게 3가지가 존재한다.  
첫번째로 이번 프로젝트에서는 kernel영역에서 수행되는 File system call을 구현하고 syscall\_handler function을 완성해줘야 한다. File system call에서 구현해야 할 기능은 file system create, file system remove, file open, file close, filesize, file read, file write, file seek, file tell등이 존재한다. 이러한 file system call을 위해서는 ppt에 명시된대로 thread에 file decripter[]라는 추가적인 자료구조가 요구된다.  
두번째로 file\_deny\_write function을 적절히 사용하여 실행중인 프로그램의 executable file이 지워지지 않도록 해줘야 한다.  
마지막으로 강의중 학습하였던대로 critical section이 보호될 수 있도록 syscall.c및 process.c를 수정해줘야 한다. 예를들자면 위에서 언급하였던 file system의 read와 write가 동시에 수행되면 안되므로 critical section으로 간주되고, lock을 통해 이 두 과정(read, write)을 각각 보호해줘야 한다. 이러한 critical section problem은 ppt에 명시된대로 lock과 semaphore를 이용하여 구현할 수 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

**1. File Descriptor**  
각 process가 여러개의 파일들을 구분하기 위해 필요한 자료구조이다. 3이상의 값을 지니며 0의 값을 지니면 stdin, 1의 값을 지니면 stdout, 2의 값을 지니면 stderr를 의미한다. File Descriptor를 구현해야만 모든 file system call을 추가적으로 구현할 수 있다.

**2. (이번 프로젝트에서 구현해야 하는) System Calls**  
이번 프로젝트의 주된 목표중 하나가 file system을 구축하는 것인데, 지난 프로젝트1에서 구현하지 못했던 file system call case들을 syscall\_handler에서 추가적으로 구현해줌으로써 file management(대표적으로 file write, read등)가 가능해진다.

**3. Synchronization in Filesystem**file system에는 critical section problem이 존재한다. 대표적으로 한 thread가 file을 wirte하고 있을때 만약 다른 어떠한 thread가 같은 file을 read하게 된다면 file system의 sychronization에 문제가 생긴다. 그러므로, 주어진 lock과 semaphore관련 function들을 이용하여 이를 막아줘야 한다. 이를 통해 file system의 sysnchronization을 해줄 수 있다.

* 1. **개발 내용**

**1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술**  
File Descriptor의 구현을 위해서 filesys/file.c에 존재하는 struct file을 userprog/syscall.h에 그대로 옮겨준뒤에 배열형태로 사용할 것이다. 왜냐하면 위에서 언급하였듯이 file descriptor란 각각의 파일들을 구분하기 위해 필요한 자료구조인데 이번 프로젝트에서 file에서 요구되는 정보인 file의 inode, position, deny\_write호출여부 등을 모두 포함한 file access용 구조체가 이미 filesys디렉토리의 file.c파일 안에 존재하기 때문이다. 또한 메뉴얼상으로 아래 사진과 같이 process당 128개의 file을 open할 수 있기 때문에 struct file을 배열형태(struct file\* file\_des[128])로 이용함으로써 각 thread당 128개의 file을 구분하고 관리할 수 있게 된다.  
[]

**2. System Calls: 구현할 각 system call에 대해 간략히 서술**   
1. SYS\_CREATE : file name과 file size를 입력받아 파일을 create해주는 system call.  
2. SYS\_REMOVE : 입력받은 file name을 갖는 file을 remove해주는 system call.  
3. SYS\_OPEN : 입력받은 file name을 갖는 file을 open해주는 system call.  
4. SYS\_FILESIZE : 입력받은 file descriptor의 file size를 return해주는 system call.  
5. SYS\_READ : file descriptor == 0이면 STDIN을 수행하고 (file descriptor >= 3)이면 file read system call을 수행한다. (pintos OS의 경우 STDERR는 고려하지 않는다)  
6. SYS\_WRITE : file descriptor == 1이면 STDOUT을 수행하고 (file descriptor >= 3)이면 file read system call을 수행한다. (pintos OS의 경우 STDERR는 고려하지 않는다)  
7. SYS\_SEEK : 입력받은 첫 번째 parameter를 file descriptor로 갖는 file의 position을 두 번째 parameter를 file descriptor로 갖는 file의 position으로 바꿔주는 system call.  
8. SYS\_TELL : 입력받은 file descriptor의 file의 position을 return해주는 system call.  
9. SYS\_CLOSE : 입력받은 file descriptor의 file을 close해주는 system call.

**3. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명**위에서 언급하였듯이 file system call에는 synchronization issue가 존재한다. 예를들어 한 thread가 file을 wirte하고 있을때 만약 다른 어떠한 thread가 같은 file을 read하게되는 case가 존재한다. 이러한 critical section problem을 해결하는 방법으로 가장 간단한 방법으로는 lock을 사용하는 방법이 있다. lock방식은 한 thread가 critical section을 점유중일 때 다른 thread의 접근을 막음으로써 synchronization issue를 해결해준다.  
이 userprog/syscall.h내부에 struct lock thread\_lock을 만들어 준뒤에 file을 open할때, file을 read할때, file에 write할때에 주어진 lock\_acquire(&thread\_lock)를 통해 lock을 해줌으로써 하나의 file system call만을 수행하게 해주고 file open, read, write가 종료되기 이전(return 이전)에 lock\_release(&thread\_lock)을 통해 lock을 풀어줌으로써 filesystem의 synchronization issue를 해결할 수 있다.   
또한 이번 프로젝트에서의 test-case중 multi-oom test case의 경우 메모리의 누수를 직접적으로 확인하는 test case인데, 여기서 말하는 메모리 누수란 부모 process가 자식 process보다 먼저 종료된 경우를 의미한다. 이러한 parent, child process synchronization issue를 해결하는 방법으로는 semaphore방식이 존재한다. semaphore방식 또한 이전에 언급하였던 lock방식과 상당히 유사한 모습을 보인다. 둘간의 가장 큰 차이점은 lock은 현재 thread간의 critical section issue를 해결해줄 수 있고, semaphore는 현재 process간의 critical section issue를 해결해줄 수 있다.  
즉, 위의 parent, child issue의 경우 ppt에 명시된 sema\_down function을 통해 부모process를 잠가줌으로서 자식 process가 종료되기 이전에 비정상적으로 종료되지 않도록 해주고, 이후 자식 process가 종료되고 나면 다시 sema\_up을 통해 부모 process가 종료될 수 있게 만들어준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| **날짜** | **구현 내용** |
| **2020/11/03** | **struct thread에 file descriptor추가 및 SYS\_READ, SYS\_WRITE를 제외한 나머지 file system call와 syscall\_handler function완성** |
| **2020/11/05** | **file descriptor, file\_deny\_write function, lock functions를 이용하여 SYS\_READ, SYS\_WRTE 구현** |
| **2020/11/10** | **bad-read, bad-write, bad-jump test case의 코드분석 및 exception.c 수정** |
| **2020/11/12** | **lock, semaphore를 이용하여 synchronization preoblem해결** |
| **2020/11/13** | **semaphore관련 function을 통해 parent, child process sync issue해결 및 모든 test case PASS** |
| **2020/11/14** | **보고서 작성** |

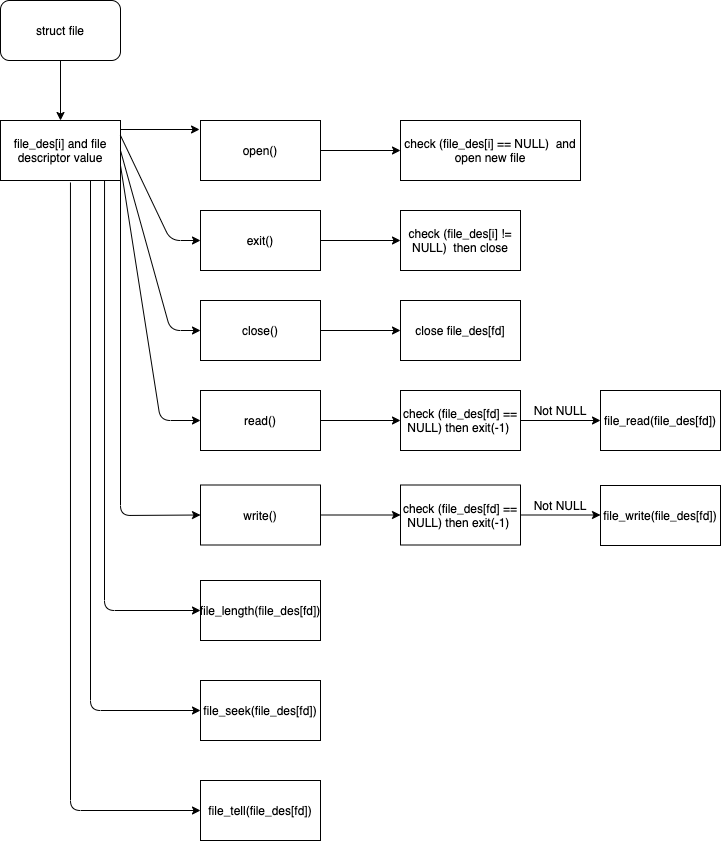
* 1. **개발 방법**

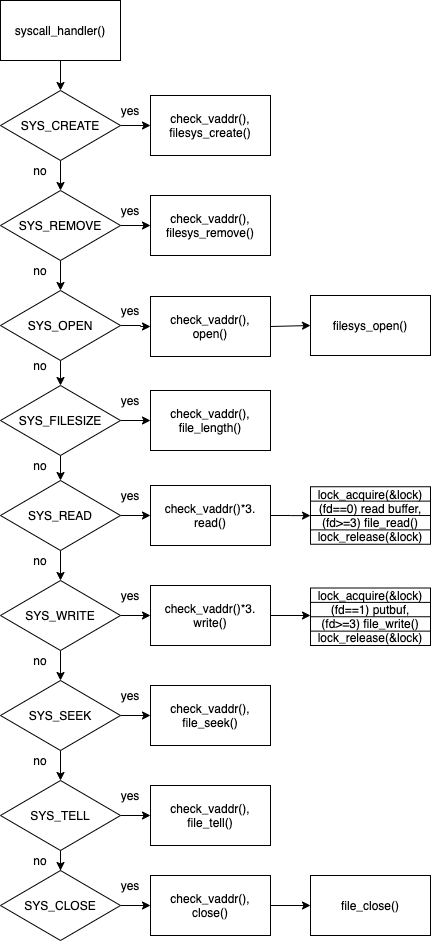
**1. File Descriptor**  
File Descriptor를 생성하기 위해서 threads/thread.h에 존재하는 struct thread를 수정해야한다. struct thread의 element로 위에서 언급한대로 struct file\* file\_des[128]을 추가해준다. 이후 추가된 element를 initial하기 위해 thread.c에 존재하는 init\_thread function에서 for반복문을 통해 모든 file descriptor를 NULL값으로 초기화 해준다.

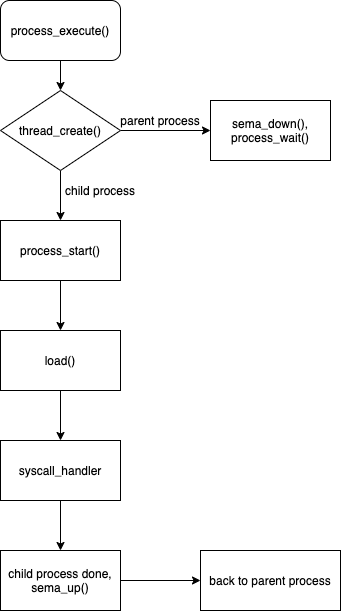
**2. System calls**  
File system call을 전부 구현해주기 위해서는 지난 프로젝트에서 구현해주었던 userprog/syscall.c의 syscall\_handler function을 수정해줘야 한다. 다른 system call과 마찬가지로 각 SYS\_case마다 입력받은 argument의 address가 valid한지 검사를 해주고, 이미 filesys/file.c에 구현되어있는 file\_tell, file\_seek, ..와 같은 file관련 함수들을 각 case에 맞게 호출함으로써 file system call들을 구현한다. 또한 지난 시간에는 STDIN, STDOUT만 구현했던 SYS\_READ, SYS\_WRITE에 대해서 추가적으로 file descriptor가 3보다 같거나 큰 경우에 대해 file\_read, file\_write function을 호출함으로써 모든 file system call을 구현한다.   
마지막으로 process가 memory에 load된 후 실행되더라도 실행 중인 프로그램의 executable file이 지워지는것을 방지하기 위해 file을 open해주는 open function에서 현재 thread의 name과 현재 file의 name이 일치하면 file\_deny\_write function을 통해 write를 방지해준뒤에, write function에서 struct file의 deny\_write element를 통해 확인해준다.

**3. Synchronization in Filesystem and Thread**위에서 언급한대로 Filesystem의 synchronization을 위해 ppt에 명시된 struct lock을 이용한다. ppt에 명시된 struct lock을 통해 lock변수 thread\_lock을 userprog/syscall.h내부에 하나 생성해준뒤에, (fd >= 3)인 경우에 대해서 file\_open, file\_read, file\_write function호출 이전에 lock\_acquire(&thread\_lock)을 통해 filesystem에 lock을 걸어주고 file\_open, file\_read, file\_write function호출이 마무리 되면 lock\_release(&thread\_lock)을 통해 lock을 풀어줌으로서 두 개의 thread가 동시에 같은 filesystem에 접근하는것을 막아준다.  
또한 부모process가 자식process 이전에 비정상적으로 종료되는 case를 막아주기 위해서는 ppt에 명시된 struct semaphore를 이용한다. 우선적으로 userprog/process.c에 존재하는 process\_execute function에서 현재 부모process를 sema\_down function을 통해 잠가준뒤에, strart\_process function에서 자식 process의 load function 호출 이후에 다시 sema\_up을 해줌으로서 부모, 자식 process간의 synchronization issue를 해결해 줄 수 있다.

1. **연구 결과**
   1. **Flow Chart**

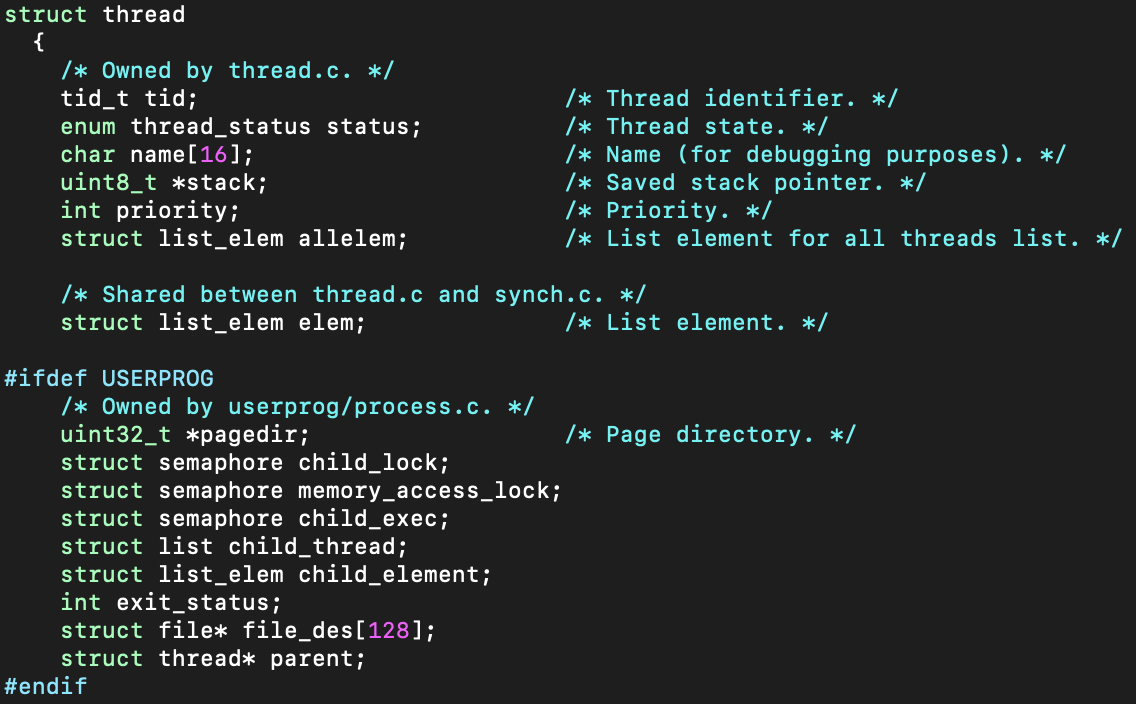
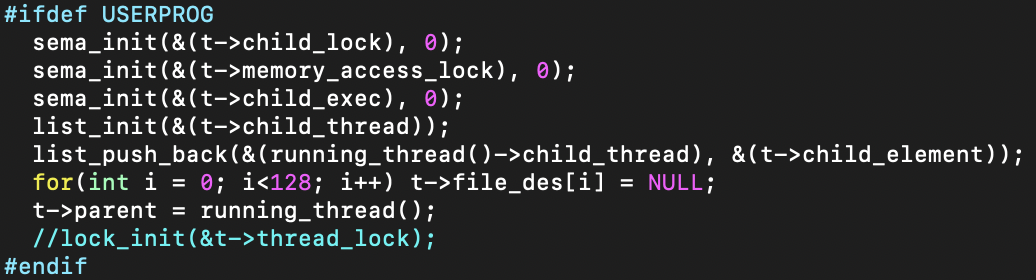
**1. File descriptor**  
****

**2. System calls  
**

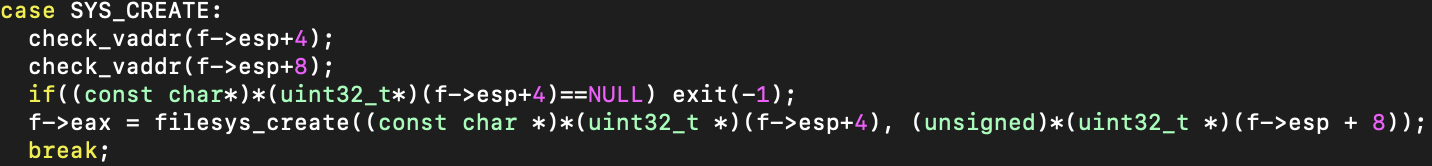
**3. Synchronization  
**

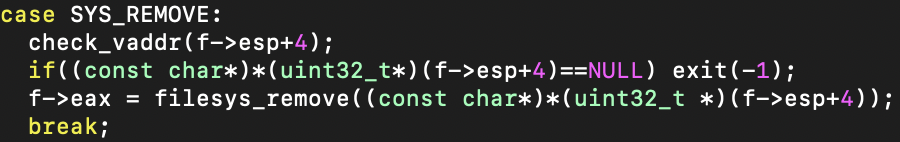
* 1. **제작 내용**

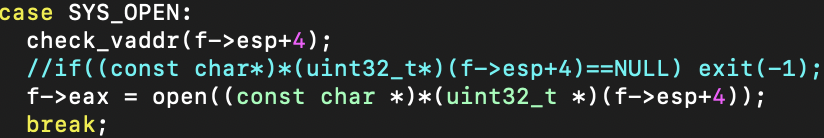
**1. File descriptor**

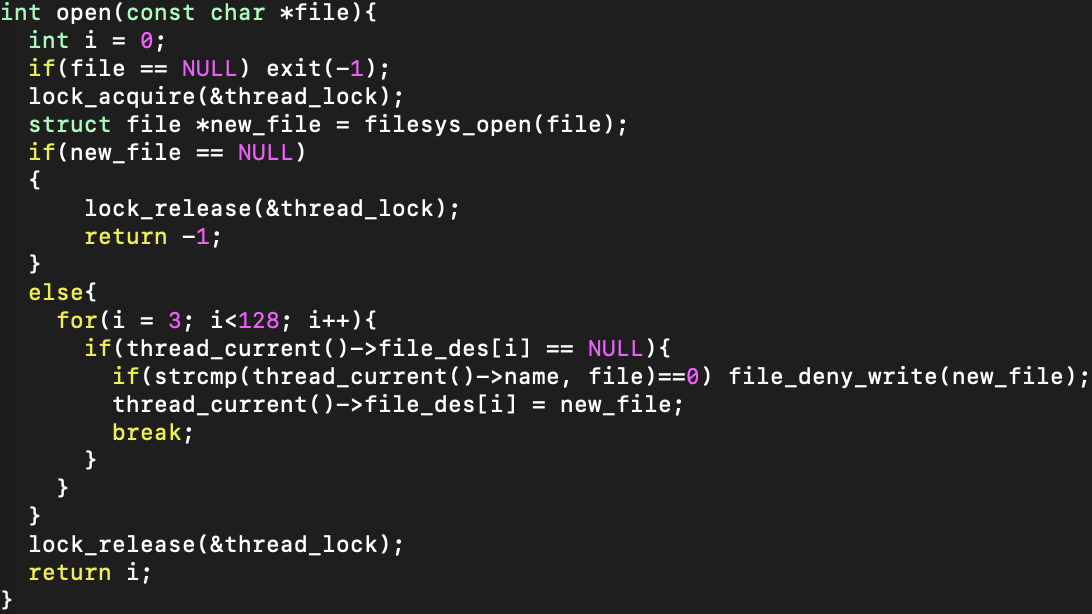
위에서 기술한대로 다음과 같이 threads/thread.h에 존재하는 struct thread내부에 element로서 각 file을 구분할 수 있게 해주는 구분자인 file\_des[128]을 추가해주었다. (메뉴얼에서 언급된대로 128개의 file descriptor를 만들어주었다)  
  
또한 위의 file descriptor들을 initial해주기 위해서 threads/thread.c에 존재하는 init\_thread function에서 다음과 같이 전부 NULL값을 넣어주는 initial작업을 진행하였다.  


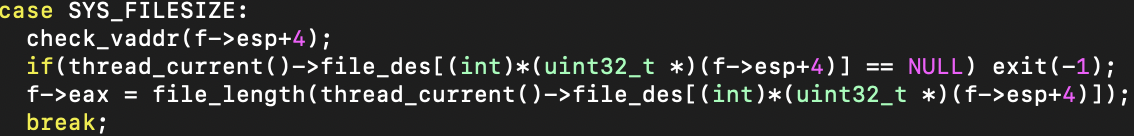
**2. System calls**

**2-1). SYS\_CREATE**  
  
SYS\_CREATE의 경우 argument의 주소검사를 해준뒤에 filesys\_create function을 호출하였다. filesys\_create는 첫 번째 argument로 file name을 설정하고, 두 번째 argument로 size를 결정하여 file system을 만들어주는 function이다.

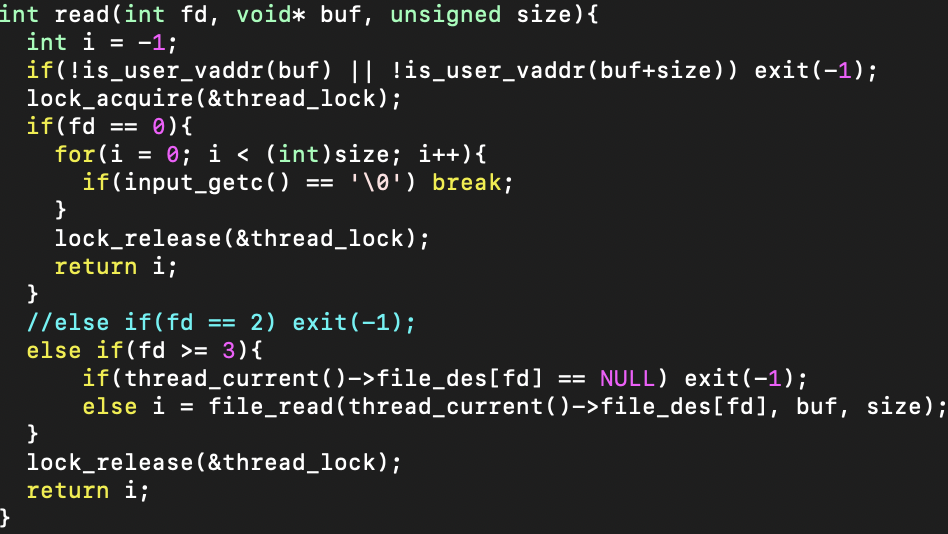
**2-2). SYS\_REMOVE**  
  
SYS\_REMOVE의 경우 argument의 주소검사를 해준뒤에 filesys\_remove function을 호출하였다. filesys\_remove는 첫 번째 argument의 이름을 가지는 file system을 제거해주는 function이다.

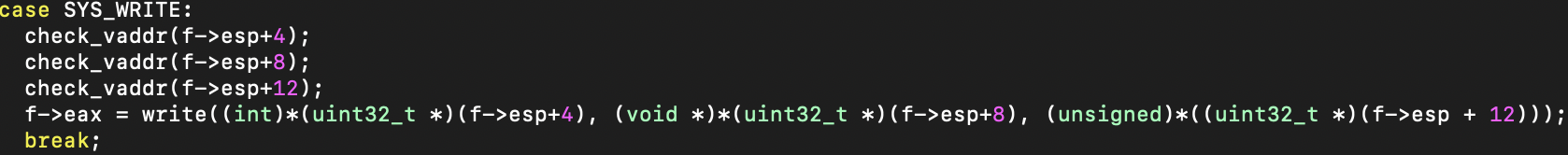
**2-3). SYS\_OPEN**  
  
SYS\_OPEN의 경우 argument의 주소검사를 해준뒤에 user defined function인 open function을 호출해준다. open function은 다음과 같다.

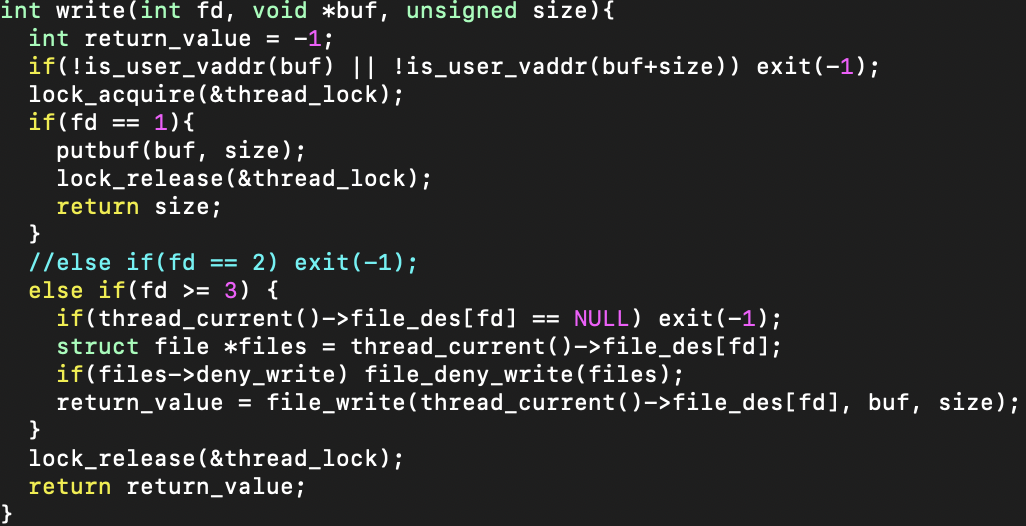
 open function  
open function의 경우 우선적으로 file name이 NULL이라면 정상적으로 파일이름을 전달받지 못한 경우이므로 exit(-1)을 통해 비정상적 종료를 전달한다. 만약 정상적으로 파일이름을 받아왔다면 file open에도 synchronization, critical section problem이 존재하기 때문에 lock\_acquire(&thread\_lock)을 통해 lock을 걸어준다. 이후 주어진 filesys\_open을 통해 전달받은 argument의 이름을 가지는 file을 open해준다. filesys\_open function으로 파일을 제대로 가져오지 못한 경우 lock을 풀어주고 return -1을 통해 에러를 반환해준다.  
파일을 제대로 open한 경우, for반복문을 통해 비어있는 file\_descriptor의 위치를 찾고 실행중인 프로그램의 excutable file이 지워지는 것을 방지하기 위해 현재 thread의 name과 file의 이름이 같은경우 file\_deny\_write()를 호출해준다. 이후 현재 thread의 비어있는 file descriptor자리에 새로운 file을 넣어줌으로써 file open을 최종적으로 완료한다. 이후 file open작업이 모두 끝마쳤으니 lock\_release(&thread\_lock)을 통해 lock을 풀어준다.

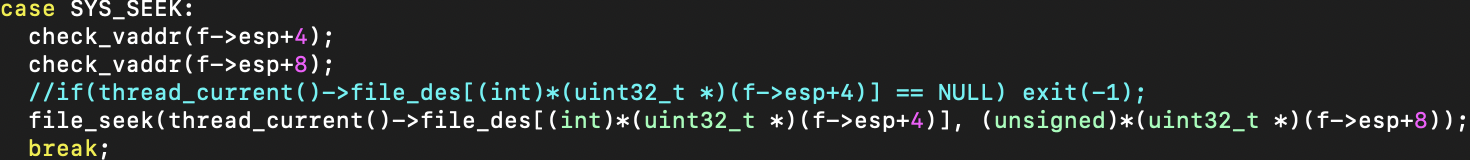
**2-4). SYS\_FILESIZE**  
  
SYS\_FILESIZE의 경우 argument의 주소검사를 해준뒤에 file\_length function을 호출하였다. file\_length function은 전달받은 argument를 file descriptor의 값으로 가지는 file의 file size를 return해주는 function이다.

**2-5). SYS\_READ**  

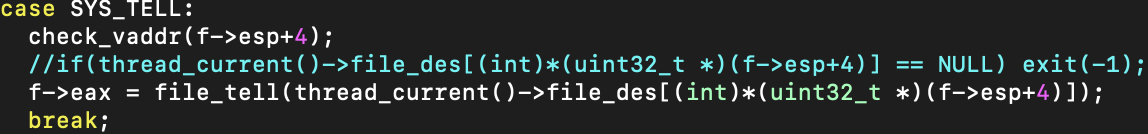

SYS\_READ의 경우 argument의 주소검사를 해준뒤에 user defined function인 read function을 호출한다.  
read function은 다음과 같다.  
 read function  
read function의 경우 우선적으로 주어진 시작주소와 size를 통해 주소값이 유효한지 검사해준다. (유효하지 않으면 exit(-1)을 통해 비정상적 종료를 알린다)  
file read과정 또한 synchronization, critical section problem이 존재하기 때문에 open과 마찬가지로 lock\_acquire(&thread\_lock)을 통해 lock을 걸어준다.  
이후 fd가 0인 경우 STDIN을 의미하므로 ppt에서 명시된대로 input\_getc() function을 통해 콘솔로 입력을 받은뒤에 lock\_release(&thread\_lock)을 통해 lock을 풀어주고 입력받은 크기를 반환해준다.  
만약 fd가 3보다 크거나 같은 경우 file을 read하겠다는 것을 의미하므로 전달받은 fd값을 통해 file을 찾아본다. 만약 file이 NULL일 경우 exit(-1)을 통해 파일을 제대로 읽어오지 못했다는 것을 알리고, NULL이 아닐 경우 file\_read function을 통해 파일을 읽어온다. 여기서 file\_read function은 전달받은 첫 번째 값을 file descriptor값으로 갖는 file에서 세 번째 argment size byte만큼을 읽어온뒤에 두 번째 argument인 buffer에 넣어주는 역할을 수행하고 읽은 byte만큼을 반환해준다.  
이후 file read작업이 모두 끝마쳤으니 lock\_release(&thread\_lock)을 통해 lock을 풀어주고 i값을 반환해줌으로서 읽어온 byte수를 반환한다.

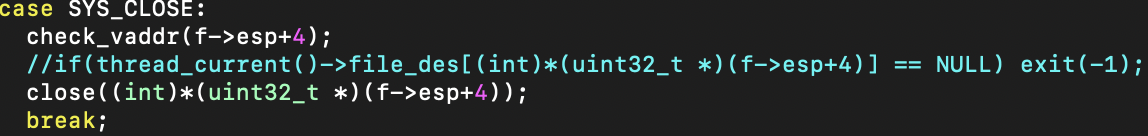
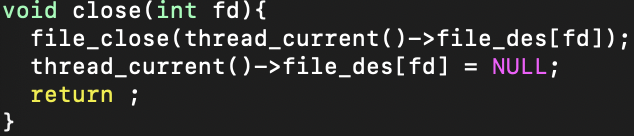
**2-6). SYS\_WRITE**   


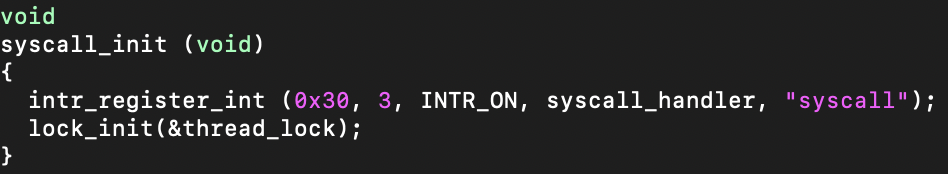
SYS\_WRITE의 경우 argument의 주소검사를 해준뒤에 user defined function인 write function을 호출한다.  
write function은 다음과 같다.  
  write function  
write function의 경우 우선적으로 주어진 시작주소와 size를 통해 주소값이 유효한지 검사해준다. (유효하지 않으면 exit(-1)을 통해 비정상적 종료를 알린다)  
file write과정 또한 synchronization, critical section problem이 존재하기 때문에 open, read와 마찬가지로 lock\_acquire(&thread\_lock)을 통해 lock을 걸어준다.  
이후 fd가 1인 경우 STDOUT을 의미하므로 ppt에서 명시된대로 putbuf(buf, size) function을 통해 buf에서 size만큼을 콘솔로 출력을 해준다. 이후 lock\_release(&thread\_lock)을 통해 lock을 풀어주고 출력해준 크기를 반환해준다.  
만약 fd가 3보다 크거나 같은 경우 file을 write하겠다는 것을 의미하므로 전달받은 fd값을 통해 file을 찾아본다. 만약 file이 NULL일 경우 exit(-1)을 통해 파일을 제대로 읽어오지 못했다는 것을 알린다. 또한 읽어오려는 파일의 deny\_write element가 참일 경우 file\_deny\_write()를 호출해줌으로서 executable file이 delete되는 것을 막아준다. 이후 file이 NULL이 아닐 경우 file\_write function을 통해 파일에 데이터를 써준다. 여기서 file\_write function은 전달받은 두 번째 argument인 buffer에서 세 번째 argment size byte만큼을 읽어온뒤에 첫 번째 argument값을 file descriptor값으로 갖는 file에 써주고, 써준 size를 반환해준다.  
이후 file write작업이 모두 끝마쳤으니 lock\_release(&thread\_lock)을 통해 lock을 풀어주고 i값을 반환해줌으로서 써준 byte수를 반환한다.

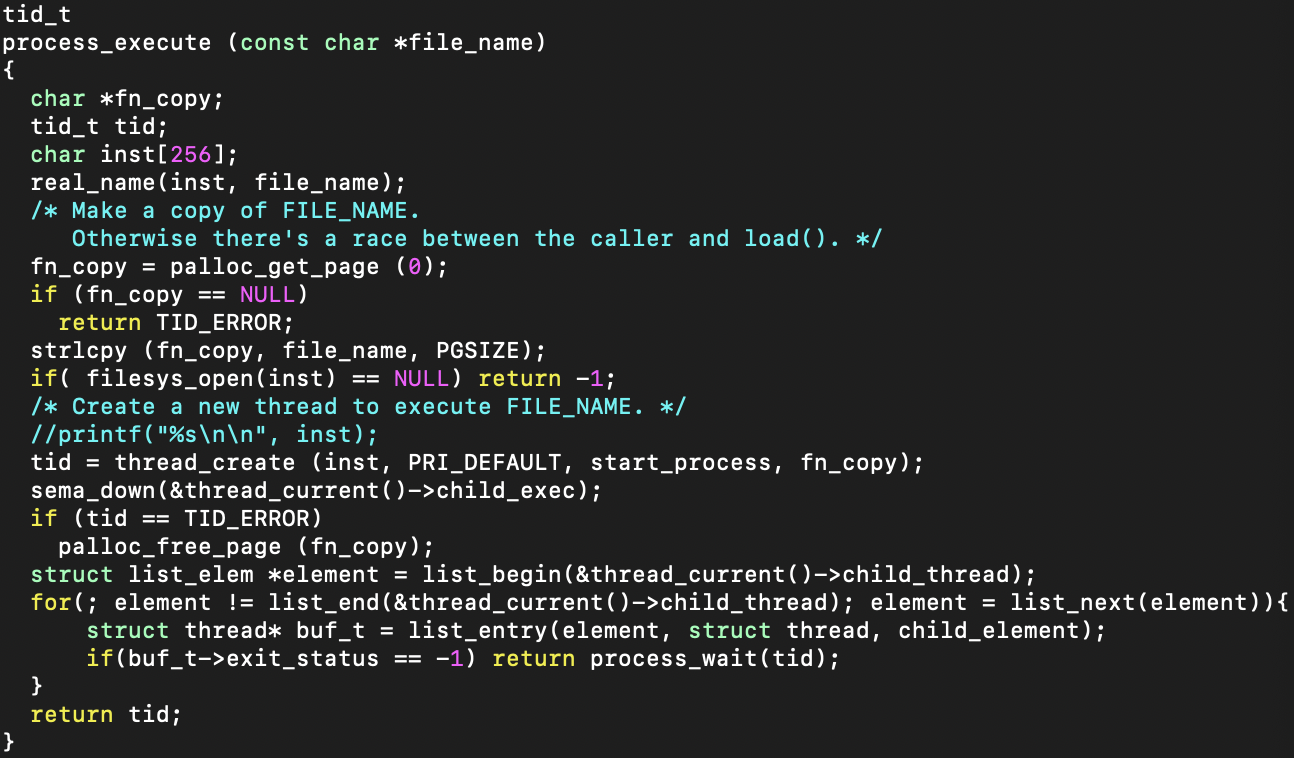
**2-7). SYS\_SEEK**  
  
SYS\_SEEK의 경우 argument의 주소검사를 해준뒤에 file\_seek function을 호출하였다. file\_seek function은 전달받은 첫 번째 argument를 file descriptor의 값으로 가지는 file의 position을 전달받은 두 번째 argument를 file descriptor의 값으로 가지는 file의 position으로 옮겨준다.

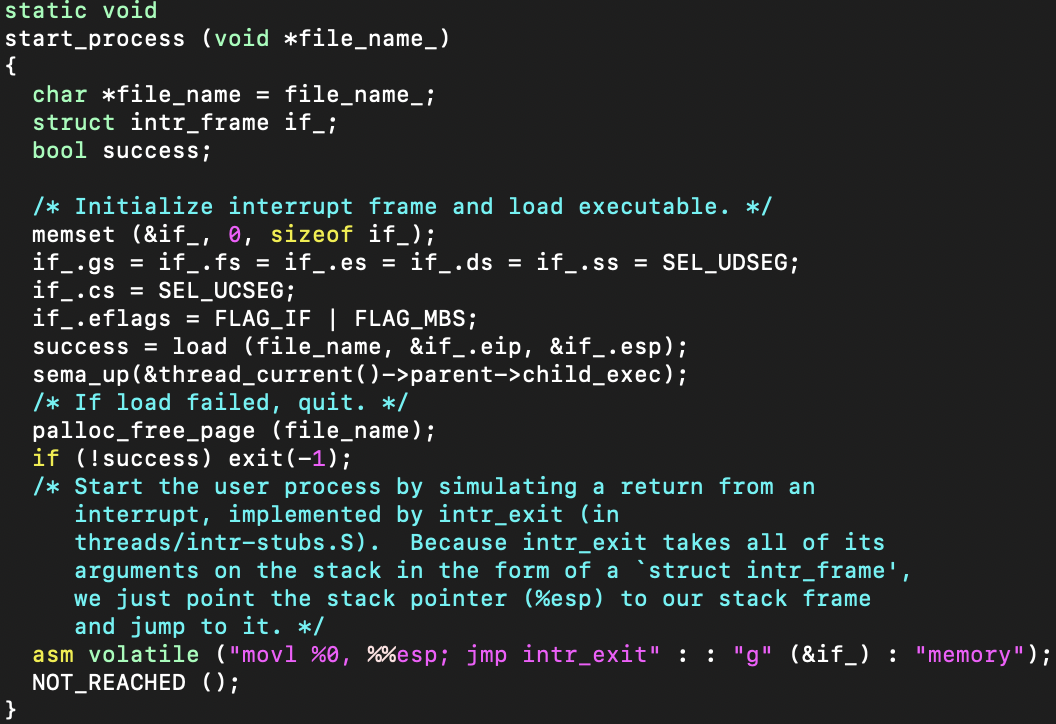
**2-8). SYS\_TELL**

  
SYS\_TELL의 경우 argument의 주소검사를 해준뒤에 file\_tell function을 호출하였다. file\_tell function은 전달받은 argument를 file descriptor의 값으로 가지는 file의 position을 반환해주는 함수이다.

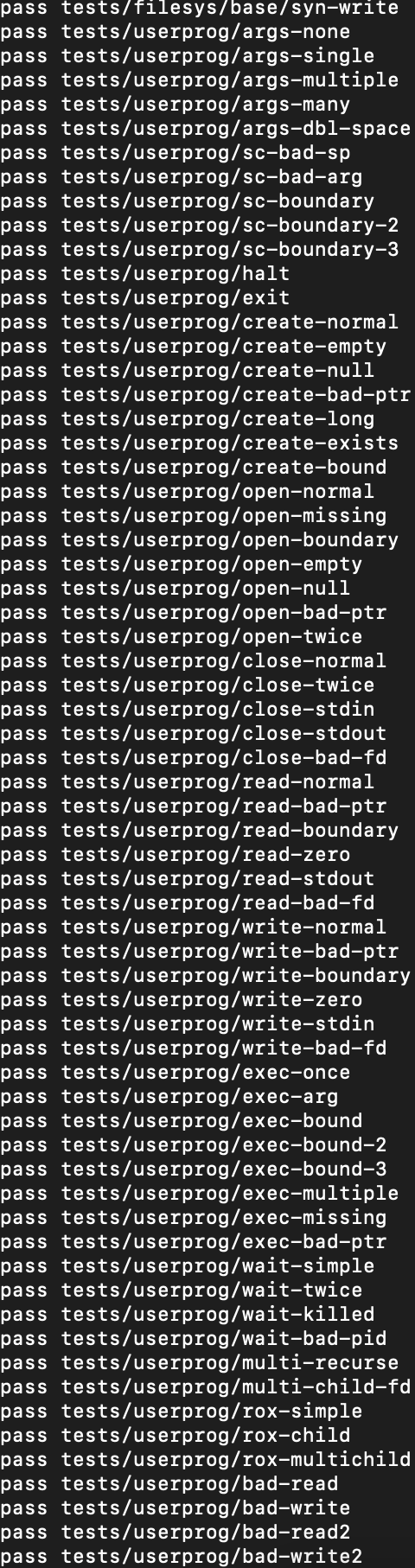
**2-9). SYS\_CLOSE**  
  
SYS\_CLOSE의 경우 argument의 주소검사를 해준뒤에 user defined function인 close function을 호출해주었다.  
close function은 다음과 같다.  
  
close function의 경우 file\_close function을 호출한다. 여기서 file\_close function은 전달받은 argument file을 close해준다.  
이후 해당 file이 존재하였던 file\_des[fd]를 NULL로 설정해줌으로써 file descriptor를 비워준다.

**3. Synchronization(Filesystem and Process)  
3-1). File system**  
  
위에서 open, read, write에서 syscall.h에 선언해준 thread\_lock변수를 사용하기 위해서 다음과 같이 syscall\_init function에서 lock\_init(&thread\_lock)을 호출해줌으로서 lock변수를 통해 file system synchronization issue를 해결할 수 있도록 하였다. 여기서 lock\_init function은 parameter로 lock type의 변수를 입력받은뒤에 그러한 lock변수를 critical section의 lock기능을 수행할 수 있도록 initial을 해주는 역할을 수행한다.

**3-2). Process  
3-2-1). process\_execute  
**process\_execute에서는 자식 process를 만들기 이전에 sema\_down(&thread\_current()->child\_exec)를 통해 부모 process를 semaphore child\_exec 변수로 잠가주는 역할을 수행한다. 이로 인해 부모 process가 자식 process이전에 종료되는 issue를 해결할 수 있다. 여기서 sema\_down function은 value가 0이 되면 그 value가 0이 아닌 값을 부여받기 이전까지 thread를 block하고 해당 semaphore의 waiter list에 현재 thread를 추가해주는 함수이다.

**3-2-2). start\_process  
**start\_process function에서는 자식 process의 load가 완료된다면 sema\_up(&thread\_currrent()->parent->child\_exec)를 통해 부모 process를 unblock헤주는 역할을 수행한다. 이로인해 자식 process가 종료된 후 부모 process가 종료될 수 있다는 것을 보장한다.  
여기서 sema\_up function은 위에서 언급한 waiter list에 member가 하나라도 존재한다면 list중 하나의 member를 pop해준뒤 unblock처리를 해주고 semaphore value를 증가시켜주는 함수이다.

* 1. **시험 및 평가 내용**



80개의 test case에 대해 전부 pass하였다.