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

담당 교수 / 분반 : 박성용 교수님 / 1반

이름 / 학번 :이나연 / 20171662

개발 기간 :11/1~11/17

1. **개발 목표**

여태까지의 pintos의 user program은 아주 간단한 exec, wait, 등 기본 기능만을 제공하였고 file system 관련 기능은 아예 구현이 되어 있지 않았다. 이번 프로젝트에서는 system call에서 file system API를 적절히 사용하여 file system 관련 function(create, remove, open, close, read, write, filesize, seek, tell)을 구현할 것이다. 또한 그와 관련된 critical section problem을 해결하며 read와 write를 할 때에 synchronization이 잘 진행되도록 lock과 semaphore를 적절히 활용할 것이다. 추가적으로 denying writes to executable files의 기능을 수행하기 위해 file system function을 활용할 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

Pintos에선 각 쓰레드는 각자 독립적인 file descriptor를 갖게 된다. 이 때 file descriptor table을 통해 각 쓰레드는 열린 파일의 목록을 관리한다. 만약 해당 부분의 구현이 되어있지 않다면, 각각의 쓰레드들은 STDIN, STDOUT, STDERR에 해당하는 file object에는 물론 I/O 관련 file object에 대한 관리가 전혀 되지 않을 것이고, 접근도 하지 못할 것이다. 따라서 해당 프로젝트에서는 기존의 thread structure에 file descriptor table의 기능을 하는 member를 추가하였다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

이번 프로젝트에서 구현하는 system call은 create, remove, open, close, filesize, read, write, seek, tell로 Pintos의 file system과 관련된 기능들이다. base file system의 경우 내부적으로 synchronization이 구현되지 않거나, file size가 creation time에 고정되어 있거나, file data가 disk에서 연속적인 range of sectors에 할당되는 등의 한계점이 존재한다. 이러한 문제를 system call의 구현을 통해 해결할 수 있다. user program에서 file system을 사용할 수 있도록 system call에 file system function들을 추가한다.

3. Synchronization in Filesystem

process 간에 통신을 할 때 공유메모리를 통해 특정 데이터를 공유하게 되는데 이런 경우에는 자원을 동시에 접근할 경우 문제가 발생할 수도 있다. multi-threaded 프로그램에서는 공유하는 자원에 대해 하나의 데이터에 하나의 프로세스만 접근할 수 있도록 제한해 두어야 하는데 이것이 critical section이다. 만약 critical section에서 어떤 process가 execute하고 있다면 다른 process는 그 process가 critical section을 빠져나올 때까지 기다려야한다. 이를 관리하기 위한 기법이 Synchronization(동기화)이다.

file을 읽고 쓸 때에 synchronization을 고려하지 않는다면, 이미 실행되고 있는 파일에 write가 수행될 수도 있고, 이미 실행되고 있는데 삭제될 수도 있고, 읽고 있는 파일에 write할 수 있는 경우 등 매우 다양한 위험한 경우의 수가 존재한다. 따라서 critical section을 보호하기 위한 synchronization 장치가 마련되어야 한다. 맞게 구현을 한다면, 각각의 파일에 대하여 그 안의 critical section이 두 개 이상의 프로세스에 의해 access되지 않게 되어 critical section problem이 발생하지 않게 될 것이다.

* 1. **개발 내용**

1. File Descriptor

file descriptor를 저장하기 위해 struct thread내부에 struct file\*형의 배열의 자료형을 가지는 멤버 변수를 사용하였다.

struct file\* fd[128];

로 구현을 하였는데 이는 pintos manual내에서 한 프로세스 당 128개의 file만이 open된다고 가정하여도 된다고 하였기 때문에 수행 시간면에서 효율적이도록 미리 128개의 file pointer 공간을 마련하였다. 또한 앞선 3개의 0번째, 1번째, 2번째는 각각 STDIN. STDOUT, STDERR로 이미 쓰임이 정해졌기 때문에 3번째 인덱스부터 I/O file descriptor를 저장하는 것으로 하였다. 각 배열의 인덱스를 file descriptor로 사용하였다.

2. System Calls

- create: 인자로 전달받은 파일명과 파일 사이즈로 이루어진 새로운 파일을 create한다. 새로운 파일의 file descriptor를 반환한다.

- remove: 인자로 전달받은 파일명을 가지는 파일을 삭제한다.

- open: 인자로 전달받은 파일명을 가지는 파일을 open한다. 그 후 음이 아닌 정수로 된 이 파일의 file descriptor를 반환한다.

- filesize: 인자로 전달받은 file descriptor를 가지는 파일의 size를 반환한다.

- read: 인자로 전달받은 file descriptor를 가지는 파일에서 size만큼의 byte를 읽어 들여와 buffer에 저장한다. 실제로 읽어 들인 byte수를 반환한다. fd가 0이면 input\_getc()를 통해 키보드에서 읽어 들인다.

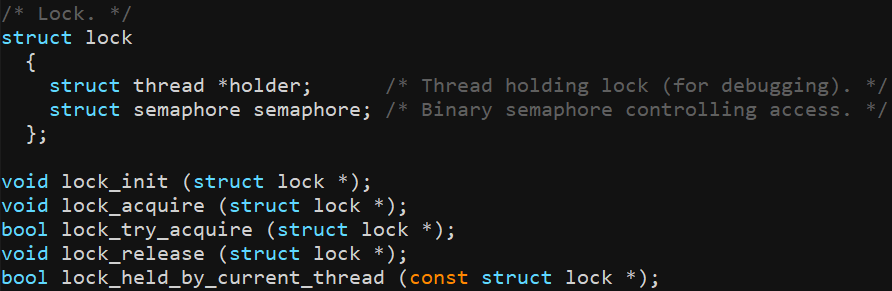
- write: 인자로 전달받은 file descriptor를 가지는 파일에서 size만큼의 byte를 buffer에 write한다. 실제로 write한 byte수를 반환한다. fd가 1이라면 console에 putbuf()를 통해 write를 수행한다.

- seek: 인자로 전달받은 file descriptor를 가지는 파일에서 다음으로 read되거나 write될 byte를 position으로 바꾼다. byte수는 파일의 시작을 기준으로 한다.

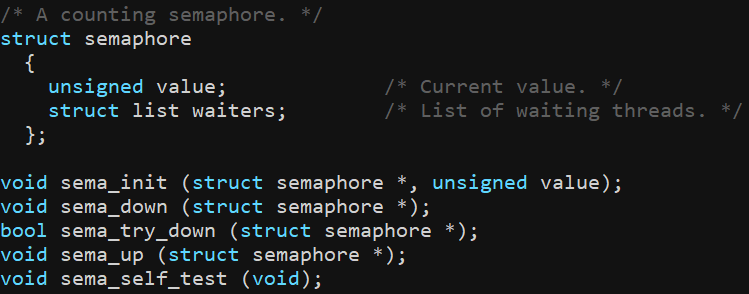
- tell: 인자로 전달받은 file descriptor를 가지는 파일에서 다음으로 read되거나 write될 byte를 반환한다. byte수는 파일의 시작을 기준으로 한다.

- close: 인자로 전달받은 file descriptor를 가지는 파일을 close한다. exit하거나 terminate되는 process는 그에 해당하는 file descriptor들을 모두 implicit하게 close한다.

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



lock: file에 접근할 때 하나의 file을 동시에 여러 process가 접근하지 않도록 하게 하는 synchronization을 구현할 때에 사용된다. 대표적으로 read()의 경우, 두 가지의 lock을 사용하였는데, 하나는 read하는 file에 write를 하지 못하게 막는 lock을 설정하였고 나머지 하나는 read를 수행할 때에 read count를 세고 read count가 충분한지, read중의 write를 막는 등의 내용이 담긴 critical section을 위한 wrt lock을 설정하였다. 그 외의 open과 write에는 하나의 lock을 사용하여 critical section에 두 개 이상의 프로세스가 접근하지 못하도록 하였다. 특히 write에서는 write를 수행하기 전에 read에서 사용한 wrt lock을 사용하여 release된 wrt lock 이 없으면 write를 수행하지 않는다.



semaphore: counting semaphore와 binary semaphore가 존재하는데, binary semaphore는 위의 lock과 비슷하게 가질 수 있는 경우가 2가지이다. 따라서 위와 비슷하게 동작할 수 있는 반면, counting semaphore는 조금 더 손쉽게 접근할 수 있다. 위에선 read count로 또 다른 변수를 사용하여 read가 가능한지 check를 하였고, 이를 semaphore로 구현한다면 두 개의 semaphore empty와 full을 사용하여 buffer의 empty, full을 check할 수 있게 된다. 또한 내부의 critical section의 경우는 binary semaphore를 사용하여 다른 process의 접근을 막을 수 있게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11/1~11/9: 매뉴얼 분석, 자료구조 및 알고리즘 설계, file system, system call 구현

11/10~11/17: synchronization 구현, 테스트 및 코드 수정, 보고서 작성

* 1. **개발 방법**

<userprog/process.c>

project1에서 구현하였던 semaphore를 이용한 parent-child process 관리에 대한 부분을 수정해야 함. 특히 parent가 child가 load되기도 전에 부모가 끝나는 상황을 고려하지 않았었기 때문에 이 부분에 대한 수정이 필요하다. 또한 load에 실패했을 때는 해당 thread의 parent의 child list에서 빼주어야 한다. 강제 종료된 child list가 있는지를 확인하고 있다면 process\_wait로 실패한 프로세스를 회수하는 코드도 구현해야 한다.

수정해야 하는 함수: process\_execute(), start\_process()

<userprog/syscall.c, syscall.h>

create, remove, open, close, filesize, read, write, seek, tell의 system call을 구현해야 한다. pintos manual에 나와 있는 설명대로 각 system call을 구현하는데, open, read, write와 같은 critical section이 존재하는 system call의 경우 lock을 사용하여 해당 부분을 protect해야 한다. 해당 lock은 syscall\_init에서 initialize된다. 또한 위의 system call들을 위하여 syscall\_handler()에서도 들어온 syscall number에 대한 적절한 system call 함수를 실행시킬 수 있도록 해야 한다. 메모리 누수에 대한 문제도 추가적으로 해결하기 위해 exit() system call에서 close되지 않은 open된 file들을 모두 닫아주는 부분도 구현해야 한다.

수정(생성)해야 하는 함수: create(), remove(), open(), close(), filesize(), read(), write(), seek(), tell(), syscall\_init(), syscall\_handler(), exit()

<threads/thread.c. thread.h>

project1에서 수정하였던 struct thread의 멤버에 더 추가해야 하는 멤버들이 생긴다. 우선 load fail시 처리할 수 있게끔 도와주는 load관련 semaphore, 현재 process의 parent process를 가리키는 thread형 포인터, 그리고 file descriptor를 저장할 수 있는 struct file\* fd[128]까지 추가되어야 한다. 또한 child process가 load에 성공했는지, 끝난 상태인지를 더 쉽게 확인하기 위해 bool형 변수 2개를 추가한다.

수정해야 하는 구조체: struct thread

수정해야 하는 함수: init\_thread()

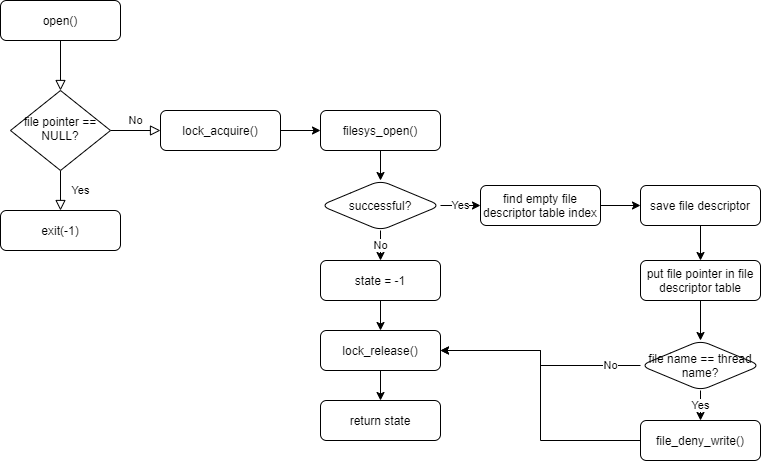
<userprog/exception.c>

kernel panic을 방지하기 위해 exception.c에서 exit(-1)을 할 수 있게끔 조건을 강화해야 한다.

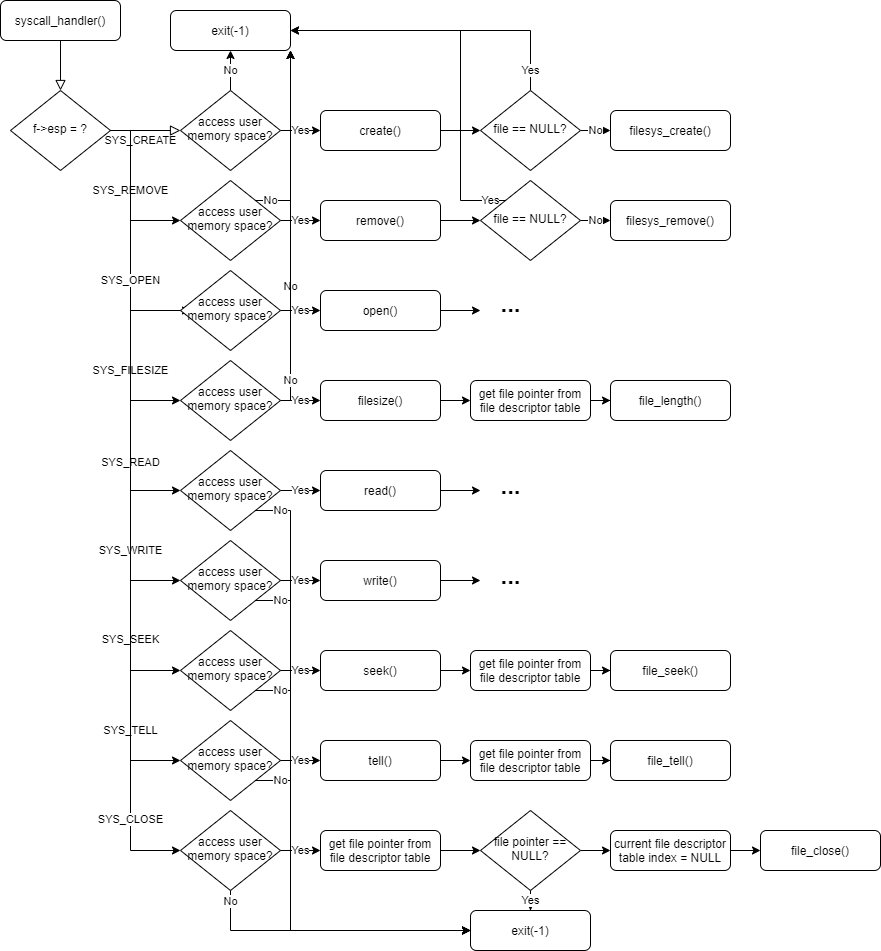
수정해야 하는 함수: page\_fault()

1. **연구 결과**
   1. **Flow Chart**

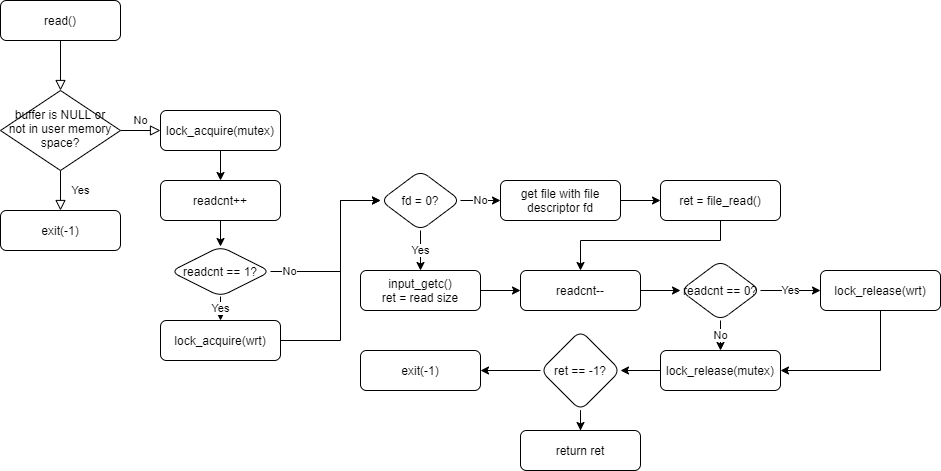
1. File Descriptor

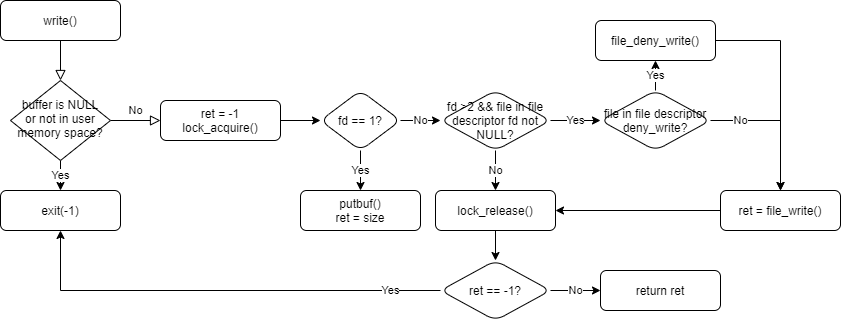


2. System Calls



3. Synchronization in File System

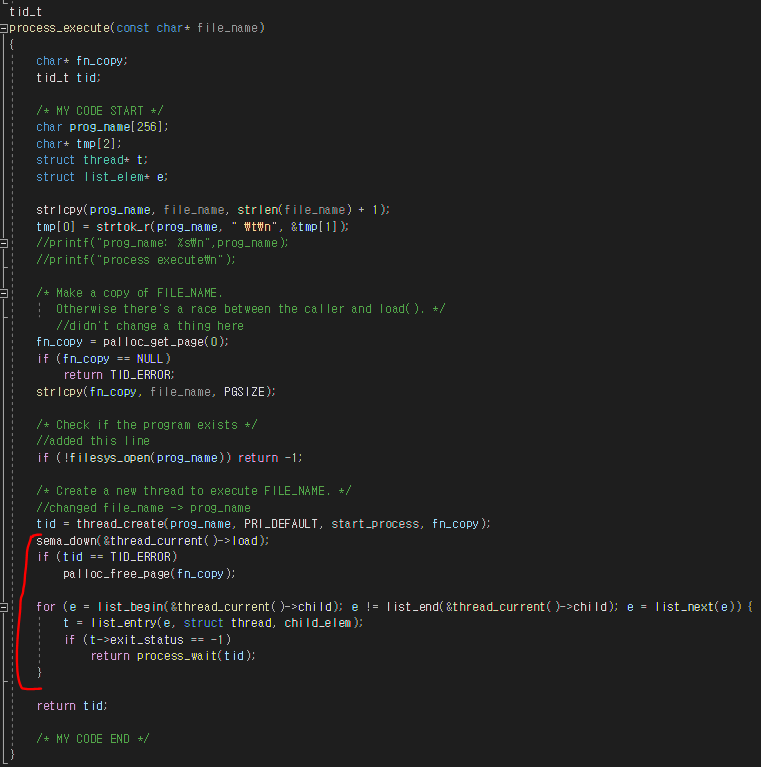




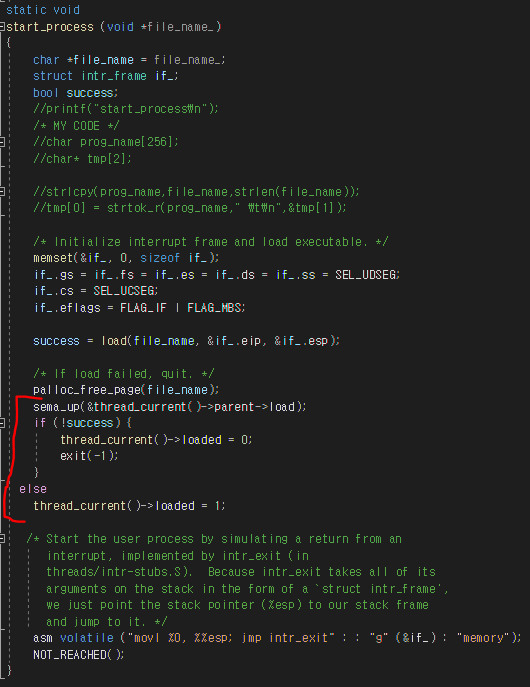
* 1. **제작 내용**

<userprog/process.c>

file system에 완전 직접적인 부분은 아니지만, file system 관련 기능을 구현하다 syn-read, syn-write가 계속 pass가 되지 않는 이슈를 발견하였다. 추후에 설명할 syscall.c에서 lock을 사용하여 read와 write, open의 synchronization은 완벽히 고려한 것 같았는데도 해당 test case는 fail이었다. output file을 살펴본 결과, load에서 에러가 남을 확인하였다. 이 부분에서 load에서도 문제가 생길 가능성이 있다는 것을 깨달은 후 아래 부분을 구현하였다.

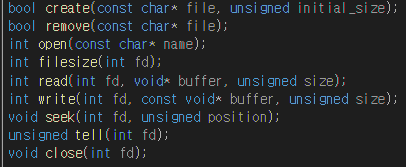


빨간색으로 표시한 부분을 기존 코드에 추가하였다. process에서 parent가 child가 load되기도 전에 끝나는 부분을 고려하려 해당 부분을 추가하였다. child thread가 생성된 후 start\_process가 수행되는데 그 전에 parent thread의 process\_execute가 끝나면 parent가 child의 load 성공 유무를 알 수 없다. 즉, parent thread는 child thread의 load가 끝날 때까지 기다려야한다. 이를 구현하기 위해 process\_execute에서 child thread를 생성한 후 child가 load될 때 까지 parent를 sema\_down 시켜준다. 또한 현재 thread가 끝나기 전에 child process들 중 강제로 종료된 프로세스가 존재하는지 for문을 통해서 확인하였고, 만약 있다면 process\_wait()로 실패한 프로세스를 회수하였다. 성공적으로 회수되면 return tid를 하게끔 해주었다.



child thread의 load가 완료되면 parent thread가 진행될 수 있도록 start\_process에서 parent thread를 sema\_up 시켜줘야한다. 그런 다음 load의 결과를 살핀다. 만약 해당 쓰레드의 load가 실패하였다면, 현재 thread의 loaded라는 bool형 변수에 0을 넣는다. 이는 해당 쓰레드가 load 성공했는지에 관한 flag의 역할을 한다. 그런 다음 exit(-1)을 하여 load가 fail되었음을 알린다. 성공했다면 loaded에 1을 넣는다. 해당 부분을 구현하여 syn-read/syn-write를 성공적으로 구현해낼 수 있었다.

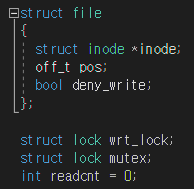
<userprog/syscall.h>

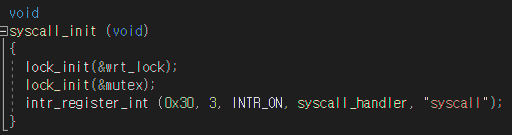


file system 관련 system call 함수들을 정의해주었다.

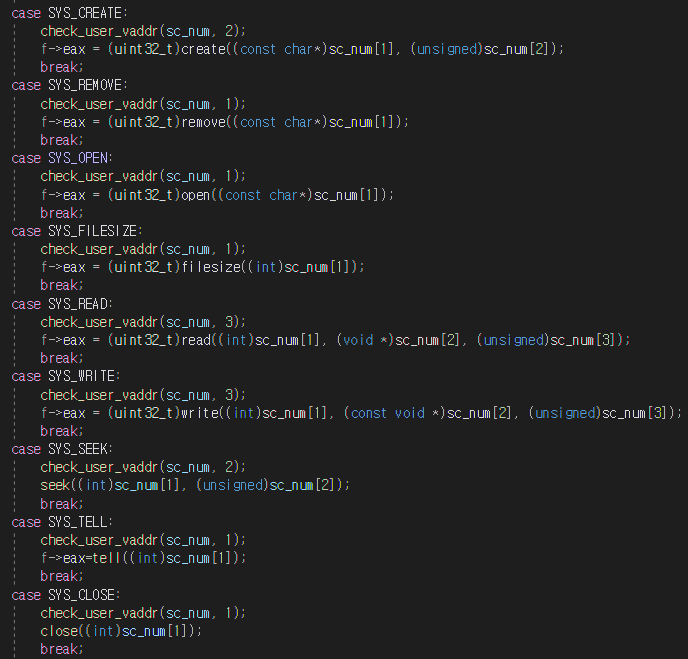
<userprog/syscall.c>

해당 부분에서 file system 관련 system call을 구현해주었다. 우선 synchronization이 정상적으로 수행되기 위해서는 read, write, open에서의 critical section을 보호해주어야 한다. 한 번에 두 개 이상의 프로세스가 critical section에 들어가면 안되기 때문이다. 또한 read의 경우 first readers-writers 알고리즘을 사용하기 위해 두 개의 lock이 필요하므로 두 개의 lock을 전역변수로 선언해주었다. 또한 file descriptor을 통한 file 정보 access를 해야 하는데, struct file에 대한 정보가 헤더 파일에 없기 때문에 해당 구조체의 구조를 그대로 가져왔다.



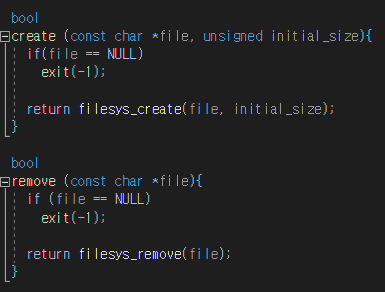


위에서 선언한 lock들을 initialize해주어야 하는데, 이는 syscall\_init()에서 구하였다.

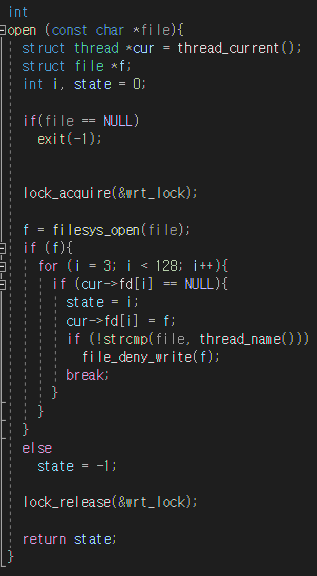


f->esp에 저장되어 있던 syscall number를 sc\_num(위치)에 저장한 후 각 system call number에 해당하는 system call 함수들을 넣었다. project 1에서는 raw하게 address를 f->esp+4 이런 식으로 표현하였는데, 해당 부분이 가독성이 떨어지는 것 같아서 int\*인 sc\_num을 활용하여 위와 같이 argument를 깔끔하게 넘겨주었다. project 1에서와 같이 각 system call이 사용하는 메모리 공간에 대하여 해당 부분이 user program memory space내부에 있는지를 항상 확인해주었다.

이제부터 file system의 system call의 구현이 어떻게 되었는지를 설명한다.

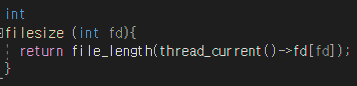


file system의 system call중 create와 remove이다. 둘 다 file system에서 제공하는 API를 통해서 구현이 된다. create의 경우 기본적으로 filesys\_create()을 통해 파일명과 사이즈를 이용하여 새로운 파일을 생성하는 기능이 구현되는데, 들어온 file명 자리에 NULL이 들어온 경우 exit(-1)로 예외처리를 해주었다. remove 역시 기본적으로 filesys\_remove()를 통해 주어진 파일명으로 된 파일을 제거하는 기능이 구현이 되나 역시 file에 NULL이 들어온 경우 exit(-1)로 예외처리를 해주었다.

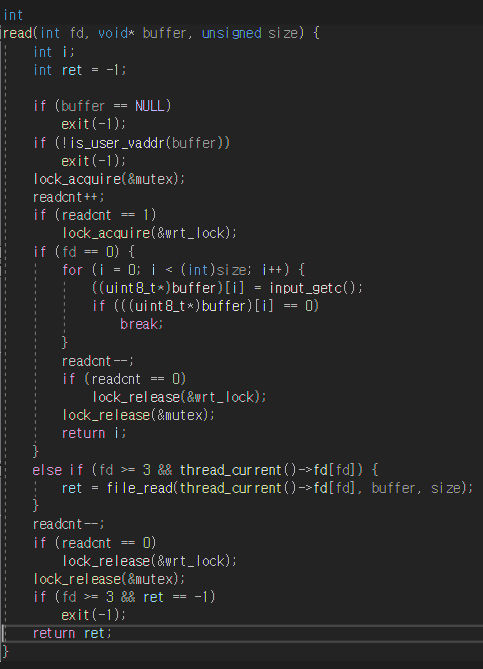


file system의 system call중 open이다. 역시 file system에서 제공하는 API인 filesys\_open()으로 file의 이름을 가지는 file이 open되고 file 포인터를 반환하는 기능이 구현되지만, 이 외에 추가적으로 file descriptor table내에 open하는 file을 저장하도록 해주어야 한다. 우선 file자리에 NULL이 들어온 경우 exit(-1)로 예외처리를 해주었고, 그런 경우가 아니면 open하는 과정은 critical section에 해당하기 때문에 lock\_acquire()를 실행하였다. 그런 다음 filesys\_open()으로 file pointer를 받아온 후 현재 thread내의 file descriptor table역할을 하는 fd멤버에서 앞에서부터 비어있는 칸을 찾은 후 해당 칸 안에 file pointer 정보를 넘겨준다. 이후 해당 file descriptor (음이 아닌 정수)를 반환해주어야 하기 때문에 배열의 인덱스를 file descriptor로 삼아 i를 state라는 변수에 저장해둔다. 또한 여기서 현재 thread명과 파일명이 같다면 file\_deny\_write()로 이미 실행중인 파일을 열려고 하는 것을 막는다. 이 역시 file system에서 제공하는 API이다.

만약 filesys\_open()에서 null이 들어왔다면 파일 open이 실패했다는 것이므로 -1을 state에 저장한다. return하기 전에는 lock\_release()를 하여 lock을 해제하고 state를 반환하며 file descriptor를 반환하고, 다른 thread가 critical section에 진입할 수 있게끔 해준다.



file system의 system call중 filesize이다. 주어진 fd를 file descriptor로 가지는 파일을 찾아야 하는데, 앞서 설명하였듯 인덱스를 file descriptor로 설정하였으므로 thread\_current()에서 fd 파일 포인터 배열의 fd번째에 저장되어 있는 file 포인터를 file system API file\_length()에 인자로 넘겨 파일의 크기를 반환하는 기능을 구현한다.

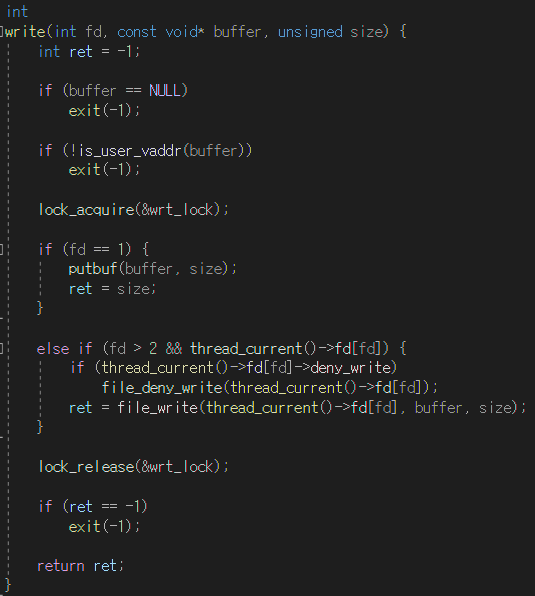


해당 부분은 file system에서의 synchronization이 반드시 진행되어야 하는 read부분이다. 강의자료에 나와있는 first readers-writers 알고리즘을 사용해보았다.

우선 buffer자리에 NULL pointer가 들어왔거나 buffer가 user program memory space에 있지 않다면 exit(-1)로 예외처리를 해주었다. 그런 다음 mutex lock을 사용해서 critical section을 보호하였다. readcount를 readcnt변수를 사용하여 read가 얼마나 실행되고 있는지를 파악하였고, wrt\_lock을 사용하여 read중의 write가 실행되는 것을 막았다. 우선 read가 시작되면 mutex lock을 걸어 주었고 read가 실행되기 때문에 readcnt를 1 증가시켰다. 그런 다음 만약 readcnt가 1이라면 read가 수행되고 있다는 뜻이므로 wrt\_lock을 걸어주었다.

그런 다음 만약 file descriptor가 0이라면 STDIN이라는 뜻으로 input\_getc()를 이용하여 최대 size만큼의 byte를 keyboard로부터 읽어 들이고, 다 수행되었다면 다시 readcnt를 감소시켜주었다. 그런 다음 만약 readcnt가 0이라면 더 이상 read가 수행되고 있지 않다는 뜻이므로 wrt\_lock을 해제해주었다. 그런 다음 critical section이 끝이 났으므로 mutex lock도 풀어준 다음 실제로 입력 받은 byte값을 반환해준다.

만약 file descriptor가 3 이상이라면 I/O file에 대한 read를 수행해야 한다는 뜻이므로 file system API인 file\_read를 통해 fd file descriptor를 가지는 file에 대한 read를 수행하게끔 하였고, 반환 값을 저장하였다. read가 다 수행되고 나면 다시 readcnt를 감소시키고 readcnt가 0이라면 더 이상 read가 수행되고 있지 않다는 뜻이므로 wrt\_lock을 해제해주었다. 그런 다음 critical section이 끝이 났으므로 mutex lock도 풀어준 다음 file\_read()의 반환값을 반환하였다. 이 때 반환값이 -1이 나왔다면 read가 잘되지 않았다는 뜻이므로 exit(-1)로 예외처리를 해주었다.

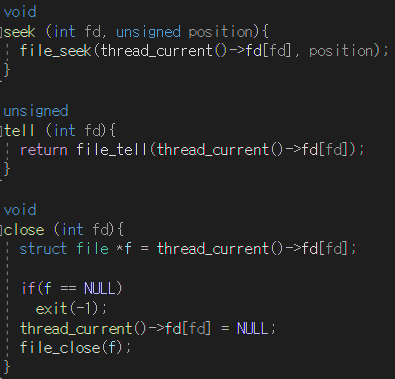


write도 마찬가지로 first readers-writers 알고리즘을 사용하였다. 우선 위와 마찬가지로 buffer가 NULL이거나 buffer의 위치가 user program memory space안에 들어있지 않다면 exit(-1)로 예외처리를 해주었다. 그런 다음 wrt\_lock을 해주어 critical section을 보호함과 동시에 read에서 wrt\_lock이 release되기전까진 read가 실행되는 동안 실행되지 않도록 하였다. 이렇게 되면 반대로 write가 되는 도중에 read가 되지 않게도 된다.

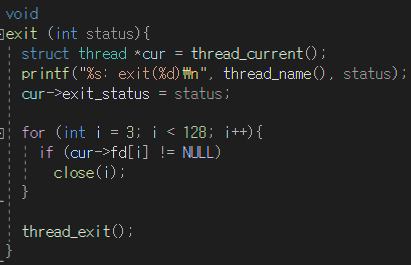
만약 file descriptor가 1이라면 STDOUT이므로 putbuf를 사용하여 console에 buffer로부터 size만큼의 byte를 출력하도록 하였다. 그런 다음 반환될 값인 size를 ret에 저장하였다.

만약 file descriptor가 3 이상이라면 I/O file에 대한 write를 수행하는 것이다. 우선 주어진 file descriptor를 가지는 file에 대한 정보를 file descriptor table에서 찾아서 해당 file의 deny\_write가 true로 되어 있다면 file system API인 file\_deny\_write()를 실행한다. 그런 다음 write의 기능을 수행하는 file system API인 file\_write를 해당 파일에 대해 실행하여 buffer자리에 size만큼의 byte를 write하게 하였고, 해당 함수의 반환값을 ret에 저장하였다.

이렇게 write가 수행이 완료되면, critical section이 끝이 났으므로 wrt\_lock을 풀어준 다음 만약 ret에 -1이 들어 있다면 write가 제대로 수행이 되지 않았다는 뜻이므로 exit(-1)을 해준다. 그렇지 않다면 ret를 반환한다.

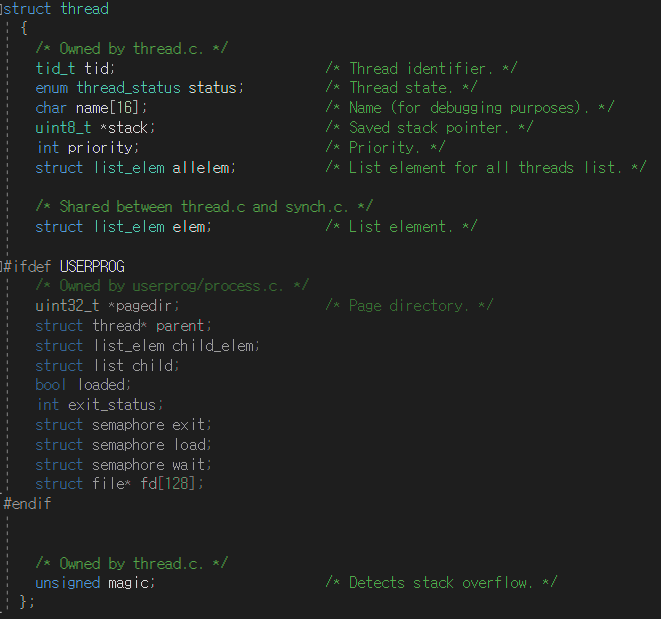


나머지 file system관련 system call이다. seek은 전달받은 file descriptor를 가지는 file을 file descriptor table에서 찾아 file\_seek()이라는 file system API로 다음으로 read되거나 write될 byte를 position으로 바꾸는 기능을 구현한다. tell은 전달받은 file descriptor를 가지는 file descriptor table에서 찾아 file\_tell()이라는 file system API로 다음으로 read되거나 write될 byte를 반환하는 기능을 구현한다. close역시 전달받은 file descriptor를 가지는 file을 file descriptor table에서 찾아 NULL이라면 exit(-1)을 하고, 아니라면 해당 file descriptor table부분을 NULL로 바꾸어 주고 file\_close라는 file system API를 통해 해당 파일을 닫는 기능을 구현한다.

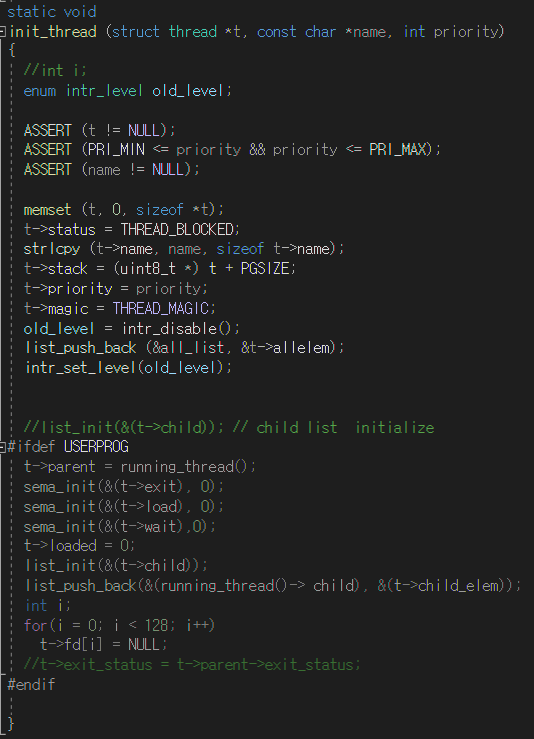


기존의 exit system call의 구현도 수정하였다. 기존의 exit\_status를 저장하는 부분은 똑같지만, 그 후 file descriptor table에 해당 thread에 미처 닫지 못한 file들이 존재하면서 thread\_exit()을 진행하게 되면 메모리 누수가 발생할 것이다. 따라서 이러한 file들을 일일이 체크하여 close해주는 부분을 추가적으로 구현하였다.

<threads/thread.c, thread.h>

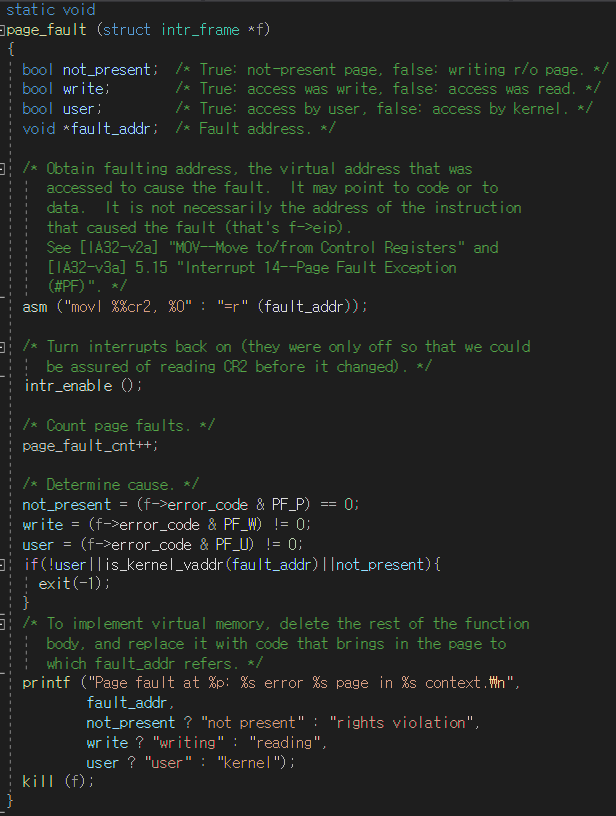


parent와 child process간의 메모리 반환 등의 메커니즘을 더욱 확실하게 구현하고자 추가한 부분이 bool loaded, struct semaphore load이다. 또한 file system관련 system call을 구현함에 있어서 file descriptor table이 필요하였기에 file descriptor table의 역할을 하는 file\* array fd를 추가적으로 멤버로 삽입하였다.



그런 다음 thread.c 내부의 init\_thread부분에서 새로 추가한 semaphore와 bool변수, 그리고 file descriptor table부분을 모두 초기화해주었다.

<userprog/exception.c>



rox- test case를 구현하던 중 kernel panic이 뜨며 실패하는 것을 보았다. 따라서 이를 보완하기 위해 exception.c에서 page\_fault()함수에서의 exit(-1)조건을 강화하였다. not\_present일 때에도 exit(-1)을 하게끔 설정하였다.

* 1. **시험 및 평가 내용**

