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

담당 교수 / 분반 :박성용 교수님 / 1반

이름 / 학번 : 20161595 / 배성현

개발 기간 : 2020/11/02 ~ 2020/11/11

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

Create, Remove, Open, Close, Filesize, Read, Write, Seek, Tell의 파일 시스템과 관련된 시스템 콜을 구현하고, file에 대한 reader/writer문제와 같은 Critical Section 문제를 해결하기 위해 하나의 Process가 Critical section안에 있을 때, 다른 프로세스들은 Critical section에 못 들어가도록 하는 Synchronization을 구현한다. 또한 핀토스는 실행중인 프로그램의 executable file이 지워지지 않기를 원하기 때문에, 이러한 executable file에는 write을 할 수 없도록 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

프로세스가 Open을 통하여 파일을 연 이후에 열려 있는 해당 파일에 접근할 때에는 File Descriptor를 사용하여서 접근할 수 있기 때문에 File Descriptor를 구현해야 한다. 이러한 File Descriptor를 구현하게 되면 프로세스가 열려 있는 파일에 대한 닫기, 읽기, 쓰기 등의 필요한 작업을 수행할 수 있게 된다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

이번 프로젝트에서의 create, remove, open, close, filesize, read, write, seek, tell의 System call을 구현하게 되면 각각의 프로세스가 파일에 대한 작업인 파일 생성, 제거, 열기, 닫기, 읽기, 쓰기, 파일 사이즈 조회, 파일 offset의 이동, 파일 offset의 조회 등의 작업을 수행할 수 있게 된다. 즉 프로세스가 파일 시스템에 대한 전반적인 처리를 할 수 있게 된다.

3. Synchronization in Filesystem

파일 시스템에서 하나의 프로세스가 어떤 파일을 읽거나, 쓰고 있을 때 다른 프로세스가 동시에 해당 파일을 읽거나, 쓰게 되면 Critical Section 문제가 생겨 데이터의 일치를 보장할 수 없게 된다. 따라서 이러한 경우를 막기 위하여 File System에서의 Synchronization을 구현하여야 한다. Filesystem 에서의 Synchronization을 구현하게 되면, 하나의 프로세스가 쓰고 있을 때 다른 프로세스들은 읽거나 쓰는 작업에 접근하지 못하게 되어 이러한 Critical Section 문제를 해결할 수 있게 된다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

File Descriptor를 구현하기 위해서 프로세스에 의하여 열린 파일들을 관리할 수 있도록 배열을 선택하였다. 이렇게 배열을 선택한 이유로는 프로세스당 파일 open에 있어 128개의 제한이 있기 때문에 배열의 크기를 미리 정해 놓을 수 있고 file descriptor를 통하여 바로 indexing하여 파일에 접근할 수 있기 때문에 file descriptor에 대하여 계속 선형적인 탐색을 해야 하는 linked list에 비해 훨씬 효율적이라고 생각하였기 때문이다.

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: create system call이 들어오게 되면 파일 이름과, 파일의 initial size를 통해 새로운 파일을 생성하게 된다. 파일 생성에 성공하게 되면 true를 반환하고, 파일 생성에 실패하게 되면 false를 반환한다.

Remove: remove system call이 들어오게 되면 파일 이름을 통해 해당 파일의 열림, 닫힘 여부에 관계없이 파일을 삭제하게 된다. 파일 삭제에 성공하면 true를 반환하고, 파일 삭제에 실패하면 false를 반환한다.

Open: open system call이 들어오게 되면 파일 이름을 통해 해당 파일을 열게 된다. 파일 open에 성공하면 open한 파일에 대한 file descriptor를 지정하여 지정한 file descriptor를 반환하고, 실패시에는 -1을 반환한다.

Close: close system call이 들어오게 되면 file descriptor에 해당하는 파일을 닫게 된다.

Filesize: filesize system call이 들어오게 되면 file descriptor에 해당하는 파일의 size(byte size)를 반환한다.

Read: read system call이 들어오게 되면 file descriptor에 해당하는 파일을 size만큼 buffer로 읽어온다. Read에 성공하면 읽은 바이트 수를 반환하고, 실패하면 -1을 반환하며 file descriptor가 0인 경우에는 키보드로부터 읽어온다.

Write: write system call이 들어오게 되면 buffer로부터 size만큼을, file descriptor에 해당하는 파일에 쓰게 된다. 파일에 실제로 쓴 바이트 수를 반환하며 file descriptor가 1인 경우에는 console에 쓰게 된다.

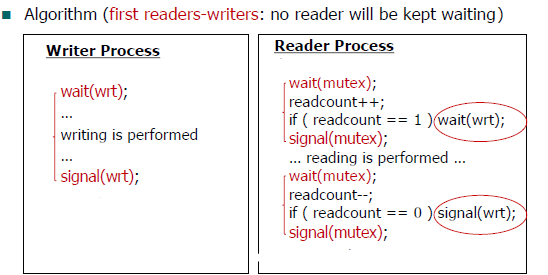
Seek: seek system call이 들어오게 되면 file descriptor에 해당하는 파일의 읽거나 쓸 다음 위치를 position으로 변경하게 된다.

Tell: tell system call이 들어오게 되면 file descriptor에 해당하는 파일의 읽거나 쓸 다음 위치를 반환하게 된다.

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

먼저 어떠한 프로세스가 파일에 대하여 open, read, write의 작업을 할 때에 해당 작업을 하기 전에 lock을 acquire해주고 작업이 끝났을 때 lock을 release 해주는 방법을 사용할 수 있다. 이 방법을 이용하게 되면 특정 프로세스가 open, read, write의 작업을 수행하고 있는 중에는 다른 프로세스들은 lock을 얻지 못하게 되므로 다른 프로세스들은 open, read, write의 작업을 수행할 수 없게 된다. 따라서 자연스럽게 Synchronization이 되게 된다. 이는 semaphore를 통해서도 똑같이 구현될 수 있는데, 처음 semaphore의 초기값을 1로 정해 주고, open, read, write의 작업을 시작하기 전에 sema\_down, 작업이 끝났을 때 sema\_up을 해주면 된다.

또한 수업시간에 배운 first readers/writers problem을 해결하는 방법을 이용할 수 있다. wrt와 mutex의 두 개의 semaphore를 이용하여 초기값을 1로 정해주고 read와 open(open의 경우에는 단순히 파일을 여는 것이므로 read와 같다고 생각할 수 있음)의 경우에는 아래의 reader와 같이 처리를 해주어 어떤 프로세스가 read, open을 하는 동안에는 다른 여러 프로세스들도 write만을 제외하고 read, open에 접근할 수 있도록 해주고, write의 경우에는 아래의 writer와 같이 처리해주어 어떤 프로세스가 write을 하는 경우에는 다른 어떠한 프로세스도 write, read, open을 할 수 없도록 해주면 된다.



1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

|  |  |
| --- | --- |
| 일정 | 내용 |
| 11/2 | 숙제 강의 자료 및 강의 확인 |
| 11/3 | 관련 Pintos Manual 확인 |
| 11/4 | File Descriptor 구현 |
| 11/5~11/6 | 파일 시스템의 System Call 구현 |
| 11/7~11/9 | 파일 시스템에 대한 Synchronization 구현 |
| 11/10~11/11 | Document 작성 |

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
* File Descriptor

File Descriptor를 구현하기 위하여 src/threads/thread.h, src/threads/thread.c, src/userprog/syscall.c의 소스코드들을 수정해줄 것이다. 가장 먼저 src/threads/thread.h의 thread구조체(자료구조)에 thread가 open한 file들의 pointer를 저장하는 배열 자료구조를 추가해 줄 것이고 Pintos manual에 따르면 파일 open에 있어 128개의 제한이 있기 때문에 해당 배열의 크기를 130(stdin, stdout의 0과 1을 제외한 128개를 저장하기 위해)으로 제한해줄 것이다. 또한 다음 file descriptor를 지칭하는 변수 int fd를 thread구조체에 추가해주어 새로운 file을 open할 때마다 fd를 통해 indexing하여 open된 파일의 포인터가 배열에 바로 추가될 수 있도록 할 것이다. 또한 이렇게 thread구조체에 추가한 file pointer 배열과 변수 fd를 초기화 해주기 위하여 src/threads/thread.c의 thread\_create()부분과 thread\_init()부분에 file pointer배열을 null로 초기화 하는 코드를 추가해줄 것이고, 또 fd 0과 1은 stdin, stdout을 나타내기 때문에 사용할 수 없으므로 변수 fd를 2로 초기화 해주는 코드를 추가할 것이다. 마지막으로 system call의 매개변수로 fd를 받았을 때에 해당 fd가 나타내는 파일 포인터가 null인지 아닌지를 체크하여 null인 경우에는 exit(-1)을 하는 함수를 src/userprog/syscall.c에 추가해 줄 것이다.

* System Calls

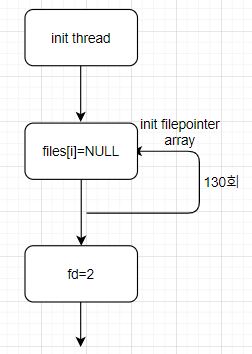
System Calls를 구현하기 위하여 src/userprog/syscall.c의 소스코드를 수정해줄 것이다. 자료구조의 경우 위의 File Descriptor항목에서 설명한 file들의 pointer를 저장하는 배열자료구조와, indexing하는 fd를 이용할 것이기 때문에 이외의 추가적인 자료구조를 필요로 하지 않는다. 먼저 create, remove, open, close, filesize, seek, tell에 대한 작동을 나타내는 함수들을 pintos에서 제공하는 filesys API를 이용하여 구현해 추가하고 syscall\_handler 함수도 수정해 각각의 system call number에 따라 해당하는 함수를 호출해 작동할 수 있도록 하는 코드를 추가해줄 것이다. 또한 open의 경우 해당 함수에서 fd를 이용하여 indexing해 file pointer 배열에 open한 파일의 pointer를 저장해 주는 코드를 추가할 것이고, executable of running thread와 같은 파일에 쓰는 것을 막기 위해 이를 체크하는 코드를 추가해줄 것이다. close의 경우에는 매개변수로 들어온 fd에 해당하는 배열의 공간을 null로 만들어주는 작업을 추가해 줄 것이다. 그리고 write과 read의 경우에는 저번 프로젝트때 이미 작성한 write함수와 read함수의 코드를 수정하여 fd가 stdout과 stdin이 아닌 경우에 file에 write, read를 할 수 있도록 하는 코드를 추가해줄 것이다.

* Synchronization in Filesystem

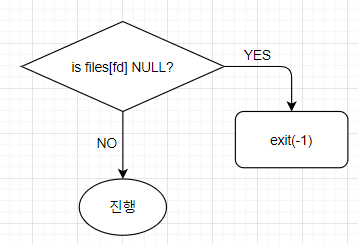
Synchronization in Filesystem을 구현하기 위하여 src/userprog/syscall.c의 소스코드를 수정해줄 것이다. 위의 II.B에서 설명한 first reader/writers problem을 해결하는 방법을 이용할 것이기 때문에 wrt와 mutex의 두 개의 semaphore(자료구조)를 이용할 것이다. 따라서 먼저 syscall\_init()함수에 wrt와 mutex의 두 개의 semaphore의 초기값을 1로 설정해주는 코드를 추가해 줄 것이다. 이후 read와 open의 작동을 나타내는 함수(read(), open())를 수정해 위의 II.B에서 언급한 reader process를 처리하는 방법과 같은 방법을 이용해 처리해줄 수 있도록 하는 코드를 추가하여 줄 것이고, write의 작동을 나타내는 함수(write())를 수정해 위의 II.B에서 언급한 writer process를 처리하는 방법과 같이 처리해주는 코드를 추가하여 줄 것이다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
* File Descriptor

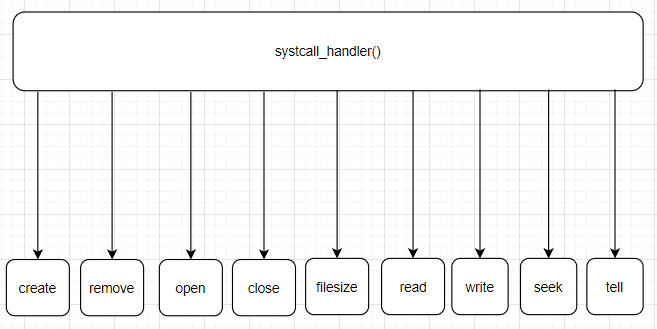


(thread init시에 해당 thread의 파일 포인터 배열과 fd 초기화)



(매개 변수로 들어온 fd가 유효한지에 대한 체크)

* System Calls



|  |  |
| --- | --- |
| create | remove |
| close | filesize |
| seek | tell |
| open | read |
| write |  |

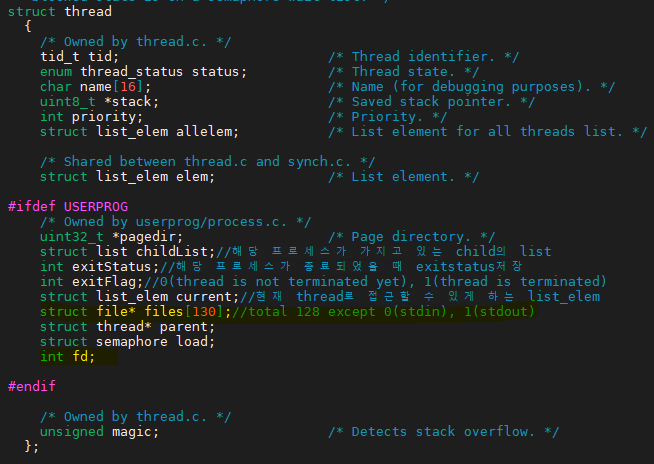
* Synchronization in Filesystem

|  |  |
| --- | --- |
| open | read |
| write |  |

* 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 설명

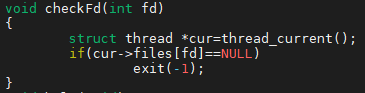
* File Descriptor



위와 같이 src/threads/thread.h에 정의되어 있는 thread 구조체에 thread가 open한 file들의 pointer를 저장할 배열(struct file\* files[130])을 추가해주었고 file descriptor를 지칭하는 변수 int fd를 thread구조체에 추가해주어 fd를 통하여 해당하는 file에 접근할 수 있도록 하였다.

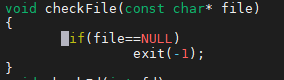
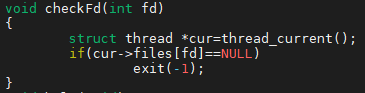


또한 위와 같이 src/threads/thread.c에 정의되어 있는 thread\_create()와 thread\_init()함수에 각각 위와 같은 코드를 추가해 주어 첫 thread나, 새로운 thread가 생성될 때마다 해당 thread의 file pointer 배열의 원소들을 null로, 변수 fd를 2로 초기화 할 수 있도록 하였다.



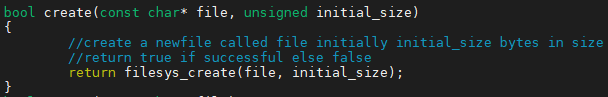
마지막으로 위와 같이 src/userprog/syscall.c에 fd가 지칭하는 파일 포인터가 null인지 아닌지를 체크하는 함수를 추가하여 fd를 사용하는 system call에 있어 해당 fd의 유효성을 체크해줄 수 있도록 하였다.

* System Calls

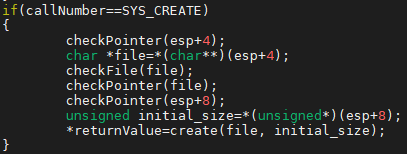
가장 먼저 파일과 관련된 system call에서는 file의 이름을 통해서, 또 fd를 통해서 작업을 수행하기 때문에 위와 같이 이들의 유효성을 체크하는 함수를 작성하여 src/userprog/syscall.c에 추가하여 주었다.

* Create



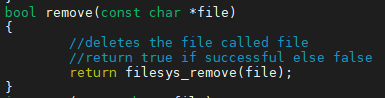
위와 같이 src/userprog/syscall.c에 파일을 생성시켜주는 핀토스 내장함수 filesys\_create()함수를 이용하여 create함수를 구현하였고, 이 create함수를 syscall\_handler()함수에서 호출하여 syscall create가 들어왔을 때 파일을 생성해줄 수 있도록 하였다.

filesys\_create(): file이름과, initial\_size를 받아 file을 만든다. 성공하면 true를 반환하고, 실패하면 false를 반환하게 된다. 만약 받은 file이름이 이미 존재하는 file이름이라면 파일 생성에 실패한다.



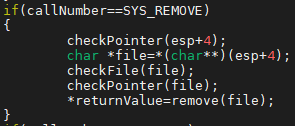
또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_CREATE이면 먼저 stack에 저장된 인자를 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자에 접근하고 이를 변수 file에 저장하도록 하였다. 또 해당 파일이름(file)에 대한 유효성을 체크하고 유효하다면 변수 file을 create()함수의 파라미터로 넘겨 create system call을 실행할 수 있도록 하였다.

* Remove



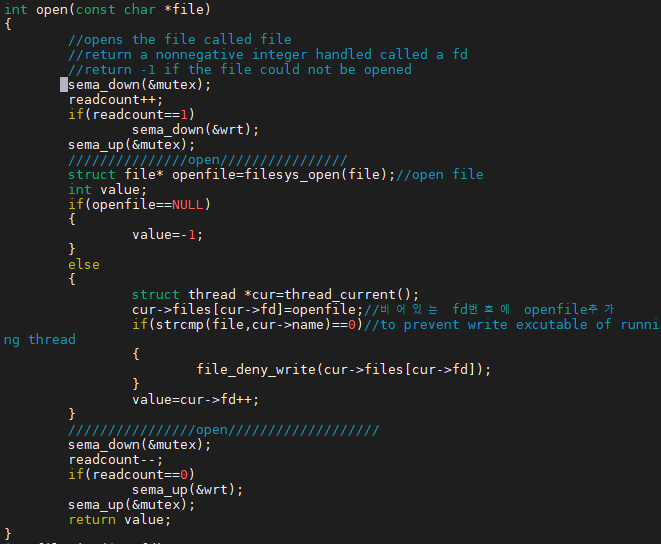
위와 같이 src/userprog/syscall.c에 파일을 삭제시켜주는 핀토스 내장함수 filesys\_remove()함수를 이용하여 remove함수를 구현하였고, 이 remove함수를 syscall\_handler()함수에서 호출하여 syscall remove가 들어왔을 때 파일을 삭제해줄 수 있도록 하였다.

filesys\_remove(): file이름을 받아 해당하는 파일을 삭제한다. 삭제에 성공하면 true를 반환하고, 실패하면 false를 반환하게 된다. 만약 받은 file이름이 존재하지 않는 file이름이라면 파일 삭제에 실패한다.



또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_REMOVE이면 먼저 stack에 저장된 인자를 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자에 접근하고 이를 변수 file에 저장하도록 하였다. 또 해당 파일이름(file)에 대한 유효성을 체크하고 유효하다면 변수 file을 remove()함수의 파라미터로 넘겨 remove system call을 실행할 수 있도록 하였다.

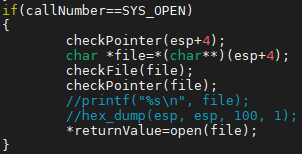
* Open



위와 같이 src/userprog/syscall.c에 파일을 open시켜주는 핀토스 내장함수 filesys\_open()함수를 이용하여 open함수를 구현하였다. 이때 file을 오픈하는데 실패한 경우에는 openfile이 null일 것이므로 이 경우에는 -1을 반환할 수 있도록 하였고, file을 오픈하는데 성공한 경우에는 현재 쓰레드의 fd(현재 쓰레드가 open한 file들을 나타내는 file pointer 배열[files]의 null로써 비어 있는 첫 인덱스를 나타냄)를 이용하여 openfile이 file pointer 배열인 files에 저장될 수 있도록 하였다. 또 executable of running thread와 같은 파일에 쓰는 것을 막기 위해 strcmp를 통해 file과 running thread의 name을 비교하여 일치하면 file\_deny\_write()함수를 통해 그러한 파일에 쓰는 것을 막아줄 수 있도록 하였다. 이후 최종적으로 file의 포인터를 저장하는데 사용하였던 fd를 반환하여 주도록 하였다. 또 이제 해당 thread의 원래의 fd에 해당하는 배열 공간에는 file pointer가 저장되었으므로 더 이상 null이 아니기 때문에 fd를 1 증가시켜주어 fd가 다시 null인 공간을 나타낼 수 있도록 하였다. 이 open함수를 syscall\_handler()함수에서 호출하여 syscall open이 들어왔을 때 파일을 open해줄 수 있도록 하였다.

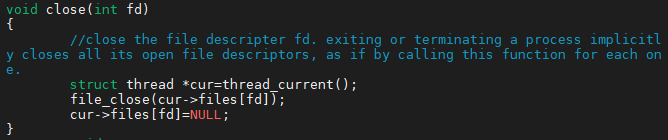
filesys\_open(): file이름을 받아 해당하는 파일을 open한다. open에 성공하면 파일 포인터를 반환하고, 실패하면 null을 반환하게 된다. 만약 받은 file이름이 존재하지 않는 file이름이라면 파일 open에 실패한다.

file\_deny\_write(): file 포인터를 받아, 해당하는 file에 write하려는 operation들을 막아주게 된다.



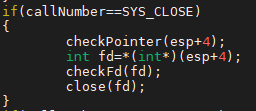
또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_OPEN이면 먼저 stack에 저장된 인자를 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자에 접근하고 이를 변수 file에 저장하도록 하였다. 또 해당 파일이름(file)에 대한 유효성을 체크하고 유효하다면 변수 file을 open()함수의 파라미터로 넘겨 open system call을 실행할 수 있도록 하였다.

* Close



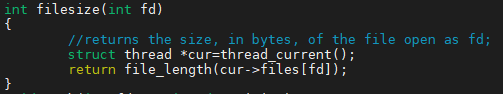
위와 같이 fd를 받아 thread가 연 파일들 중 fd에 해당하는 파일에 접근하였고 이를 핀토스 내장함수인 file\_close()함수를 이용해 닫아주도록 하는 방식으로 close함수를 src/userprog/syscall.c에 구현하였다. 그리고 닫은 이후에는 fd에 해당하는 file pointer 배열의 공간을 null로 바꾸어 주어 더 이상 그 파일에 접근할 수 없도록 하였다. 이 close()를 syscall\_handler()함수에서 호출하여 syscall close가 들어왔을 때 fd에 해당하는 파일을 닫아줄 수 있도록 하였다.

file\_close(): file포인터를 받아 해당하는 파일을 닫는다.



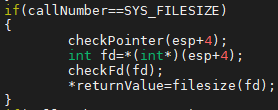
또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_CLOSE이면 먼저 stack에 저장된 인자를 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자에 접근하고 해당 인자를 fd변수로 저장하도록 하였다. 또 fd변수에 대한 유효성을 체크하고 유효하다면 fd변수를 close함수의 파라미터로 넘겨 close system call을 실행할 수 있도록 하였다.

* Filesize



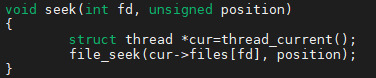
위와 같이 fd를 받아 thread가 연 파일들 중 fd에 해당하는 파일에 접근하고 이를 핀토스 내장함수인 file\_length()함수를 이용해 해당 파일의 사이즈를 반환하도록 하는 방식으로 filesize함수를 src/userprog/syscall.c에 구현하였다. 이 filesize함수를 syscall\_handler()함수에서 호출하여 syscall filesize가 들어왔을 때 fd에 해당하는 파일의 사이즈를 반환해 줄 수 있도록 하였다.

file\_length(): file포인터를 받아 해당하는 파일의 사이즈를 반환한다.



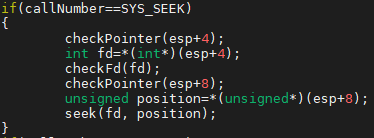
또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_FILESIZE이면 먼저 stack에 저장된 인자를 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자에 접근하고 해당 인자를 fd변수로 저장하도록 하였다. 또 fd변수에 대한 유효성을 체크하고 유효하다면 fd변수를 filesize함수의 파라미터로 넘겨 filesize system call을 실행할 수 있도록 하였다.

* Seek



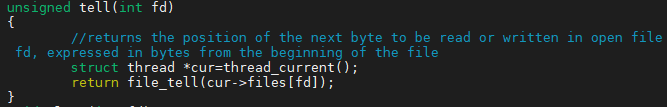
위와 같이 fd를 받아 thread가 연 파일들 중 fd에 해당하는 파일에 접근하고, 핀토스 내장함수인 file\_seek()함수를 이용해 fd에 해당하는 파일 안의 현재 위치를 파일의 시작지점으로부터 position만큼의 위치로 바꿔주도록 하는 seek함수를 src/userprog/syscall.c에 구현하였다. 이 seek함수를 syscall\_handler()함수에서 호출하여 syscall seek이 들어왔을 때 fd에 해당하는 파일의 current position을 바꾸어 줄 수 있도록 하였다.

file\_seek(): file포인터와 새 position을 받아 해당하는 파일 안의 현재 위치를 파일의 시작점으로부터 position만큼의 위치로 바꾸어 준다.



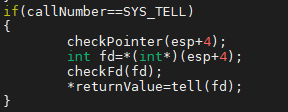
또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_SEEK이면 먼저 stack에 저장된 인자들을 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자들에 접근하고 해당 인자들을 변수(fd, position)에 저장하도록 하였다. 또 fd변수에 대한 유효성을 체크하고 유효하다면 fd, position 변수를 seek함수의 파라미터로 넘겨 seek system call을 실행할 수 있도록 하였다.

* Tell



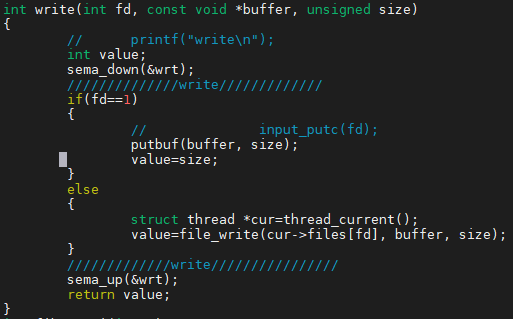
위와 같이 fd를 받아 thread가 연 파일들 중 fd에 해당하는 파일에 접근하고, 핀토스 내장함수인 file\_tell()함수를 이용해 fd에 해당하는 파일 안의 다음으로 읽거나 써질 현재 위치를 반환하여 주는 tell함수를 src/userprog/syscall.c에 구현하였다. 이 tell함수를 syscall\_handler()함수에서 호출하여 syscall tell이 들어왔을 때 fd에 해당하는 파일의 다음으로 읽거나 써질 현재위치를 반환해 줄 수 있도록 하였다.

file\_tell(): file포인터를 받아 해당하는 파일의 시작점으로부터 현재 위치를 byte offset으로 반환하여 준다.



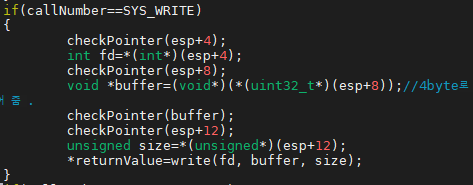
또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_TELL이면 먼저 stack에 저장된 인자를 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자에 접근하고 해당 인자를 fd변수로 저장하도록 하였다. 또 fd변수에 대한 유효성을 체크하고 유효하다면 fd변수를 tell함수의 파라미터로 넘겨 tell system call을 실행할 수 있도록 하였다.

* Write



위와 같이 fd가 1이 아닌 경우(stdout이 아닌 경우)에, 핀토스 내장함수인 file\_write()함수를 이용하여, buffer속의 내용을 size만큼 읽어와 fd에 해당하는 파일에 쓸 수 있도록 하는 코드를 이전 프로젝트에서 작성했던 src/userprog/syscall.c의 write함수에 추가하였다. 이 write함수를 syscall\_handler()함수에서 호출하여 syscall write이 들어왔을 때 buffer의 내용을 size만큼 읽어와 파일 또는 콘솔에 쓸 수 있도록 하였다.

file\_write(): file포인터, buffer, size를 받아 buffer의 내용을 size만큼 읽어 해당하는 파일의 current position으로부터 시작하여 쓰고, 실제로 써진 byte의 수를 반환한다.



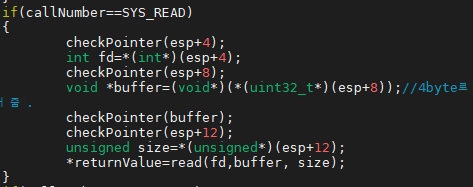
또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_WRITE이면 먼저 stack에 저장된 인자들을 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자들에 접근하고 해당 인자들을 변수(fd, buffer, size)에 저장하도록 하였다. 또한 포인터인 buffer에 대한 유효성을 체크하고 유효하다면 fd, buffer, size변수를 write함수의 파라미터로 넘겨 write system call을 실행할 수 있도록 하였다.

* Read



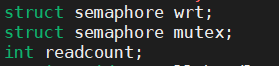
위와 같이 fd가 0이 아닌 경우(stdin이 아닌 경우)에, 핀토스 내장함수인 file\_read()함수를 이용하여, fd에 해당하는 파일 속의 내용을 size만큼 읽어와 buffer에 저장하도록 하는 코드를 이전 프로젝트에서 작성했던 src/userprog/syscall.c의 read함수에 추가하였다. 이 read함수를 syscall\_handler()함수에서 호출하여 syscall read가 들어왔을 때 파일로부터 또는 키보드로부터 size만큼 읽어와 buffer에 저장하도록 하였다.

file\_read(): file포인터, buffer, size를 받아 file의 내용을 current position으로부터 size만큼 읽어 buffer에 저장하고, 실제로 읽어진 byte의 수를 반환한다.

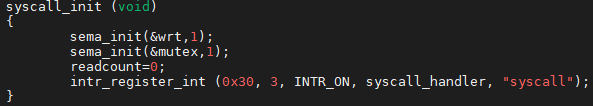


또한 syscall\_handler()함수에서는 위와 같이 system call#이 SYS\_READ이면 먼저 stack에 저장된 인자들을 가리키는 포인터가 유효한지를 checkPointer()함수를 통하여 확인하고, 유효하다면 포인터를 통해 인자들에 접근하고 해당 인자들을 변수(fd, buffer, size)에 저장하도록 하였다. 또한 포인터인 buffer에 대한 유효성을 체크하고 유효하다면 fd, buffer, size변수를 read함수의 파라미터로 넘겨 read system call을 실행할 수 있도록 하였다.

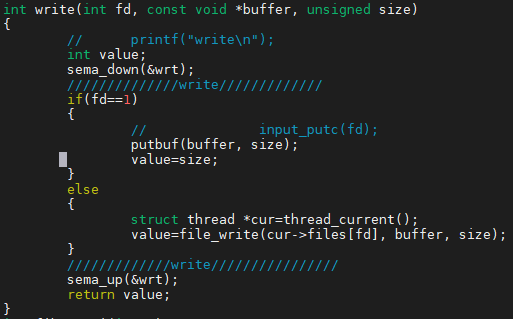
* Synchronization in Filesystem



Synchronization in Filesystem을 구현하기 위해서 first readers/writers problem을 해결하는 방법을 이용할 것이기 때문에 가장 먼저 src/userprog/syscall.c의 윗부분에 위와 같이 wrt, mutex의 두 개의 semaphore와 read와 open영역에 접근해 있는 프로세스 수를 나타내는 readcount를 전역변수로 선언하여 주었다.



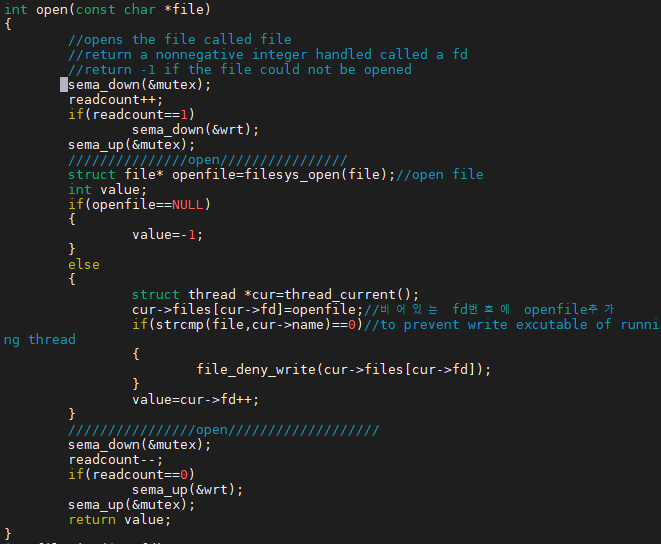
이후 wrt, mutex의 semaphore들과 readcount를 초기화 하기 위하여 위와 같이 src/userprog/syscall.c의 syscall\_init()함수에 wrt와 mutex의 두 개의 semaphore의 초기값을 1로 설정해주고, readcount를 0으로 초기화하는 코드를 추가해 주었다.



이후 위와 같이 write함수에서 실제로 write역할을 하는 부분을 sema\_down(&wrt)와 sema\_up(&wrt)으로 감싸주어 특정 프로세스에서 writing 작업을 하는 동안에는 이 wrt semaphore에 의하여 다른 프로세스에서 read, open, write의 작업에 접근하지 못하도록 하였다.



또한 위와 같이 read함수에서는 readcount와 mutex, wrt를 사용하여 실제 reading하는 부분을 감싸주어 어떤 프로세스가 write하고 있는 경우에 read 또는 open하고자하는 첫 프로세스는 sema\_down(&wrt)에서 기다리게 하고(open과 read 모두 readcount를 이용하기 때문에), 나머지 프로세스들은 sema\_down(&mutex)에 기다릴 수 있도록 하였다. 그리고 첫 프로세스가 sema\_down(&wrt)을 하고 들어오게 하여 write 작업을 하고자 하는 다른 프로세스들은 critical section안에 못 들어오도록 하였고, readcount가 1인 경우에만 sema\_down(&wrt)을 체크하도록 하여 이 경우에 다른 read하고자하는 프로세스들은 critical section에 들어올 수 있도록 하였다. 그리고 read 또는 open영역에 있는 남아있는 마지막 프로세스가 실제 read 또는 실제 open하는 부분을 나갈 때에 sema\_up(&wrt)을 하게하여 이후에는 write하고자 하는 프로세스가 critical section에 들어올 수 있도록 하였다.



read와 마찬가지로 open함수에서도 readcount와 mutex, wrt를 사용하여 실제 open하는 부분을 감싸주어 어떤 프로세스가 write하고 있는 경우에 open 또는 read하고자하는 첫 프로세스는 sema\_down(&wrt)에서 기다리게 하고(open과 read 모두 readcount를 이용하기 때문에), 나머지 프로세스들은 sema\_down(&mutex)에 기다릴 수 있도록 하였다. 그리고 첫 프로세스가 sema\_down(&wrt)을 하고 들어오게 하여 write 작업을 하고자 하는 다른 프로세스들은 critical section안에 못 들어오도록 하였고, readcount가 1인 경우에만 sema\_down(&wrt)을 체크하도록 하여 이 경우에 다른 open하고자하는 프로세스들은 critical section에 들어올 수 있도록 하였다. 그리고 open 또는 read영역에 있는 남아있는 마지막 프로세스가 실제 open 또는 실제 read하는 부분을 나갈 때에 sema\_up(&wrt)을 하게하여 이후에는 write하고자 하는 프로세스가 critical section에 들어올 수 있도록 하였다.

-multi-oom case에 대한 문제

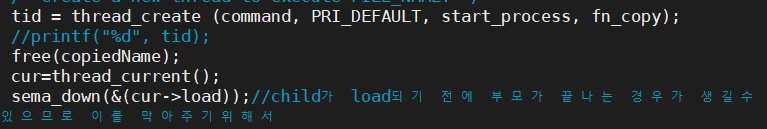
프로젝트의 여러 테스트케이스 중에 multi-oom이 계속 해서 문제가 생겼었다. multi-oom 은 메모리 누수가 있으면 fail이 발생하는 테스트 케이스이고 메모리 누수 문제는 child thread를 create하고나서 child thread가 메모리에 load되기 전에 부모가 먼저 죽어버린 경우나, 또 child thread가 load에 실패하였지만 parent의 child list에 그대로 남아있게 되는 경우에 생기게 된다. 따라서 이러한 경우들을 해결해 주기 위하여 아래와 같은 코드들을 추가하여 주었다.



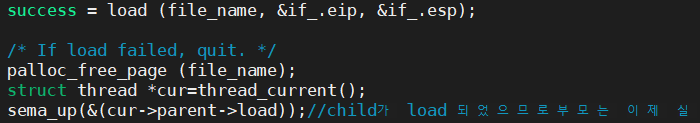
먼저 위와 같이 src/threads/thread.h의 thread 구조체에 자식이 부모thread를 식별할 수 있도록 parent라는 변수를 만들어 주었고, 또 자식이 load되기 전에 부모가 죽어버리는 것을 막기 위해서 load라는 semaphore를 추가해 주었다.



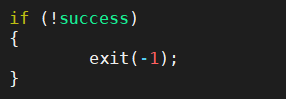
그리고 thread가 생성될 때마다 thread의 parent변수와 semaphore load를 src/threads/thread.c에서 위와 같이 초기화하여 주었다.



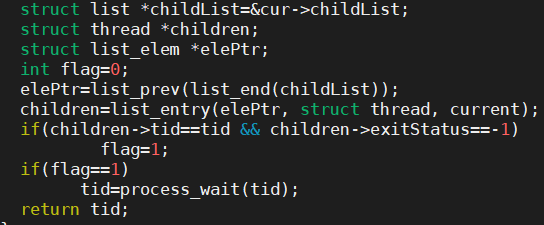
이어서 src/userprog/process.c의 process\_execute()함수에서 thread\_create()이후 부모 thread가 자신의 load semaphore에 대해서 sema\_down()을 하여 자식 thread가 부모의 load semaphore를 sema\_up()해줄 때까지 기다리도록 하였다.



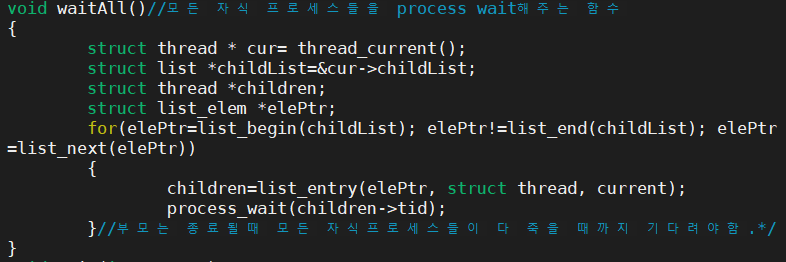
그리고 자식 thread는 parent변수를 통해 자신의 부모 thread에 접근할 수 있기 때문에 start\_process()에서 load가 끝난 이후 자식이 부모의 load semaphore를 sema\_up()해줄 수 있도록 하였고, 이를 통해 자식의 load 작업이 끝난 이후에만 부모 thread가 계속 진행할 수 있도록 하여, 자식 thread가 load되기 전에 부모가 죽어버리는 경우를 막아 주었다.

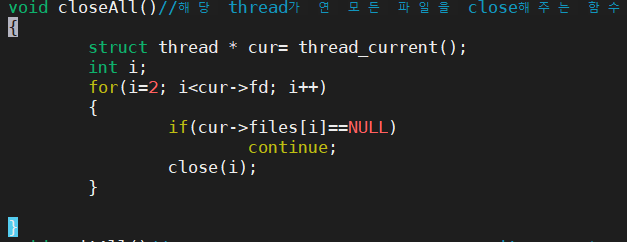


다음으로 위와 같이 자식 thread가 start\_process()에서 load가 실패한 경우에는 exit(-1)을 하면서 죽게 되기 때문에 자식 thread가 죽었지만 parent의 child list에 그대로 남아있게 되는 경우가 생기게 된다.

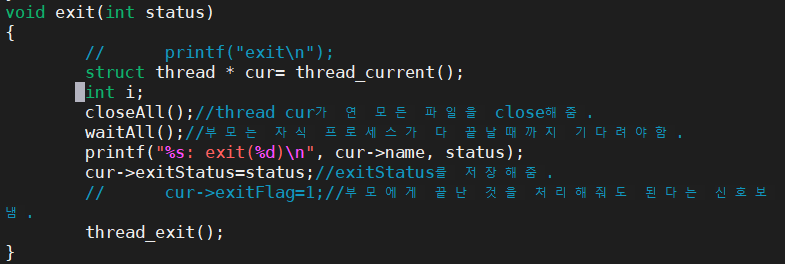


이러한 케이스를 막아 주기 위하여 src/userprog/process.c의 process\_execute()에 위와 같은 소스코드를 추가하였다. thread\_create를 통해 막 만들어진 자식 thread는 부모의 childList의 가장 뒤에 존재할 것이기 때문에 해당 자식 thread 즉 children이 exit(-1)을 통해 죽어버린 경우를 체크해주고 이 경우에 process\_wait()를 해주도록 하여 메모리 누수가 발생하지 않도록 하였다.





또한 src/userprog/syscall.c에 위와 같이 waitAll()과 closeAll()이라는 함수를 만들어 주었는데 waitAll()의 경우에는 부모가 childList를 돌아 자식 프로세스들이 다 죽을 때까지 wait하고 있도록 한 함수이고, closeAll()의 경우에는 자신이 연 파일들을 다 닫아 줄 수 있도록 한 함수이다.



따라서 이러한 closeAll()과 waitAll()을 src/userprog/syscall.c의 exit()에서 호출해 주도록 하였고, 최종적으로 multi-oom testcase를 통과할 수 있었다.

* 1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부

