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

담당 교수 / 분반 : 박성용 교수님 / 1반

이름 / 학번 : 김채연 / 20181617

개발 기간 : 2020.10.31~2020.11.17

1. **개발 목표**

이전 프로젝트1 에서 구현한 프로그램에 추가로 파일 시스템에 대한 system call을 구현한다. file descriptor, create, remove, open, close, filesize, read, write, seek, tell 와 같은 system call 명령어, filesystem의 동기화 기능을 구현하는 것을 목표로 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

이전 프로젝트 1에서는 파일 시스템을 다루지 않는 기본적인 기능을 구현하였다. 따라서 이번에는 파일시스템을 다루는 시스템 콜에 대한 추가적인 구현을 한다.

1. File Descriptor

프로세스에서 특정 파일에 접근하기 위해서는 file descriptor를 이용해야 한다. 따라서 프로세스 별로 열고 있는 파일들을 관리할 수 있도록 file descriptor를 구현한다. 이 file descriptor를 이용하여 thread별로 file에 접근할 수 있다.

2. System Calls

System Call 명령어가 입력되었을 때 system call handler를 통해 argument를 stack에서 가져와서 file system 관련 system call 명령어에 맞게 처리된다.

3. Synchronization in File system

파일 시스템의 동기화는 파일을 read/write 할 때, 프로세스의 execute/exit 할 때 필요하다. 하나의 파일이 여러 프로세스에서 조작되는 것을 방지하고, 프로세스가 수행/종료될 때 부모와 자녀 프로세스의 상태를 확인하여 오류가 발생하는 것을 막는다.

* 1. **개발 내용**

1. File Descriptor

File Descriptor 구현은 index를 이용하여 접근이 쉬운 배열 자료구조를 이용하였다. 각 프로세스는 독립적인 file descriptor set을 가지고 있어야 하므로 thread 자료구조 내에 file구조체인 fd를 배열로 선언하였다. 핀토스 매뉴얼에서 프로세스당 open file의 수를 128로 정하였기 때문에 배열의 크기는 128로 하였다.

2. System Calls

아래는 이번 프로젝트에서 개발한 system call 명령어이다.

1. create : 주어진 file으로 불리는 새로운 file을 만든다. 성공적으로 만들어지면 true를 이외에는 false를 return한다.
2. remove : 주어진 file으로 불리는 파일을 삭제한다. 성공적으로 삭제가 된다면 true를, 이외에는 false를 return한다.
3. open : 주어진 file으로 불리는 file을 연다. 해당 file의 file descriptor를 return하거나 만약 여는 것에 실패한다면 -1을 return한다. 만약 single file이 한번 이상 열리게 된다면 새로운 file descriptor를 return하도록 한다.
4. close : 주어진 file descriptor fd를 닫는다.
5. filesize : 주어진 fd로 열리는 file의 size를 bytes로 return한다.
6. read : 주어진 fd로 열리는 file을 buffer로 read한다. 실제로 읽은 byte의 수를 return하고 만약 파일을 읽지 못하면 -1을 return한다. fd가 0이라면 keyboard를 통해 read를 진행한다.
7. write : open file fd에 buffer의 size bytes 만큼 write를 수행한다. 그리고 실제로 쓰인 byte의 수를 return 하고 만약 byte를 전혀 쓸 수 없다면 0을 return한다. fd가 1이라면 console창에 write를 진행한다.
8. seek : open file fd에서 다음 byte를 read나 write를 하는 위치로 바꾼다.
9. tell : open file fd에서 read나 write를 하는 위치를 return한다.

3. Synchronization in Filesystem

파일 시스템에서 사용되는 system call 처리에서 lock을 사용한다. 하나의 파일에 동시에 읽고 쓰고 지우는 경우가 발생하는 것을 막기위해 lock 구조체인 file\_lock을 새로 선언하여 이것을 lock\_acquire과 lock\_release 해주었다. 따라서 read, write, open 명령어에서 file에 접근하여 해당 명령어를 수행하기 전과 후에 lock을 이용하여 보호해주었다.

프로세스 간의 동기화 처리에서 semaphore를 사용한다. semaphore는 이전 프로젝트1에서 프로세스가 wait, exit될 때 동기화를 처리해주었는데 이번에는 프로세스가 execute 될 때, start 될 때 semaphore를 이용하여 동기화를 처리해주었다. thread 구조체에 semaphore를 위한 새로운 sema 구조체인 load\_sema를 선언하여 이것을 sema\_up, sema\_down해주며 처리하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2020.10.31~2020.11.17

10/31 ~ 11/7 file descriptor 및 system call 구현

11/8 ~ 11/14 synchronization 구현

11/15 ~ 11/17 오류 수정 및 보고서 작성

* 1. **개발 방법**

thread.h, thread.c에서 thread구조체 내에 file descriptor를 추가로 선언해준다. 그리고 child가 실행중인지를 관리하는 semaphore인 load\_sema와 parent를 가리키는 parent, 비정상적으로 종료된 child를 관리하는 flag, 현재 실행중인 file을 가리키는 cur\_file을 추가로 선언한다. thread.c의 init\_thread 함수 내에 file descriptor를 모두 NULL로 초기화하고 load\_sema는 sema\_init을 통해 초기화한다. flag는 0으로 parent는 running\_thread로, cur\_file은 NULL로 한다.

file.c, file.h에서 file 구조체를 다른 파일에서도 사용할 수 있도록 .c 에서 .h로 옮겨준다.

syscall.c, syscall.h에서 syscall\_handler 함수 내에서 switch 문에 새로운 명령어들을 추가한다. 각 명령어에 맞는 함수를 호출하여 system call을 수행한다. 이때 parameter는 esp를 통해 stack에 쌓여있는 argument를 넘겨준다. 각 명령어에 대한 함수는 sys\_명령어 함수로 처리하였다. read와 write의 경우에는 filesystem을 이용하는 코드를 추가한다. 명령어 처리 후 return value가 생긴다면 eax에 저장한다. syscall.c에서 각 명령어에 넘겨주는 값이 valid한지 user\_vaddr함수를 이용하여 검사를 한다. 또한 각 명령어 처리 함수 내에서도 file descriptor가 유효한지 검사를 한 뒤 NULL값을 가진다면 sys\_exit(-1)를 호출하여 종료하였다.

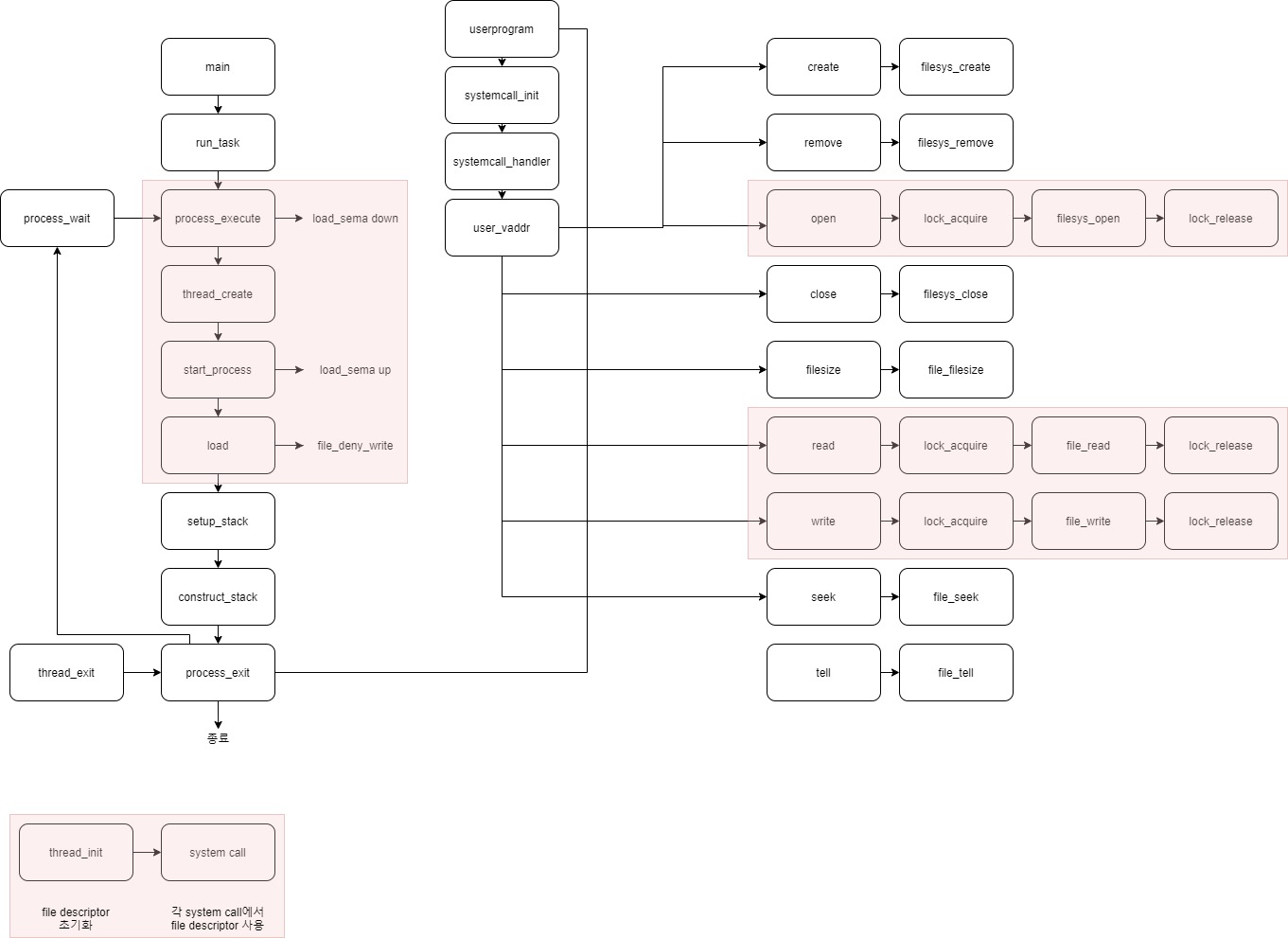
exception.c에서 page\_fault내에서 not\_present값을 검사하는 코드를 추가한다. 만약 not\_present가 0이라면 sys\_exit(-1)를 호출하여 종료하였다.

process.c에서 프로세스 동기화를 위해 semaphore 기능을 구현하는 코드를 추가한다. process\_execute와 process\_start에 load\_sema를 이용하여 child가 load 되기 전 parent가 끝나거나 child가 끝났을 때 이후에 메모리를 정리하도록 처리한다. 또한 비정상적으로 종료된 child에 대해서도 flag를 이용하여 처리는 코드를 추가한다.

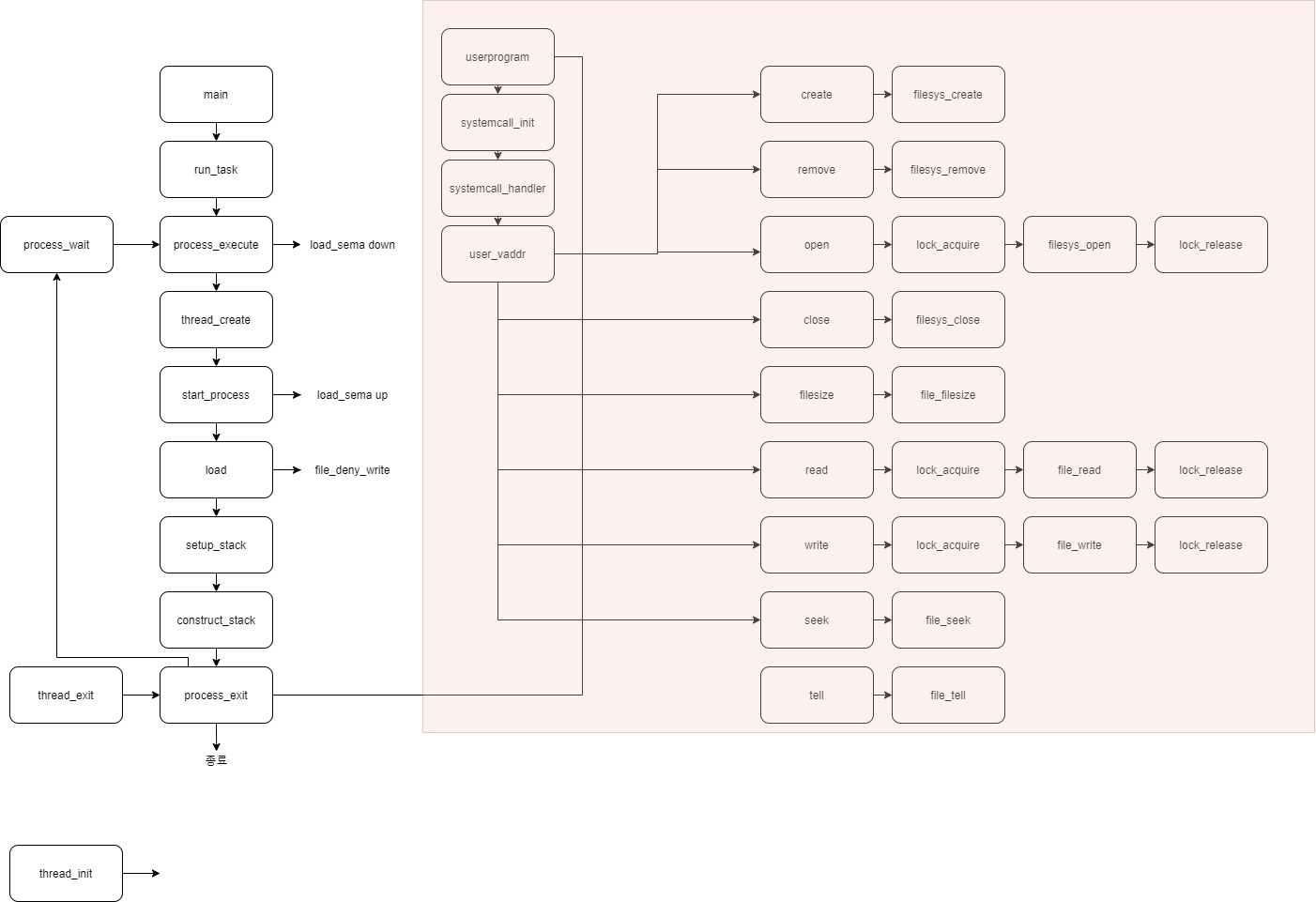
syscall.c에서 파일에 대한 동기화를 위해 lock 기능을 구현하는 코드를 추가한다. open, read, write에서 file\_lock을 이용하여 같은 파일이 동시에 처리되지 않도록 처리한다.

process.c에서 실행중인 thread에 write하는 것을 막기 위해 process\_load에서 정상적으로 load 되었을 때 해당 file을 cur\_file로 저장한 뒤 file\_deny\_write를 하여 처리하는 코드를 추가한다. 또한 syscall.c의 sys\_exit 함수 내에서 종료하기 전, 앞에서 저장한 thread의 cur\_file을 닫아주는 코드를 추가한다.

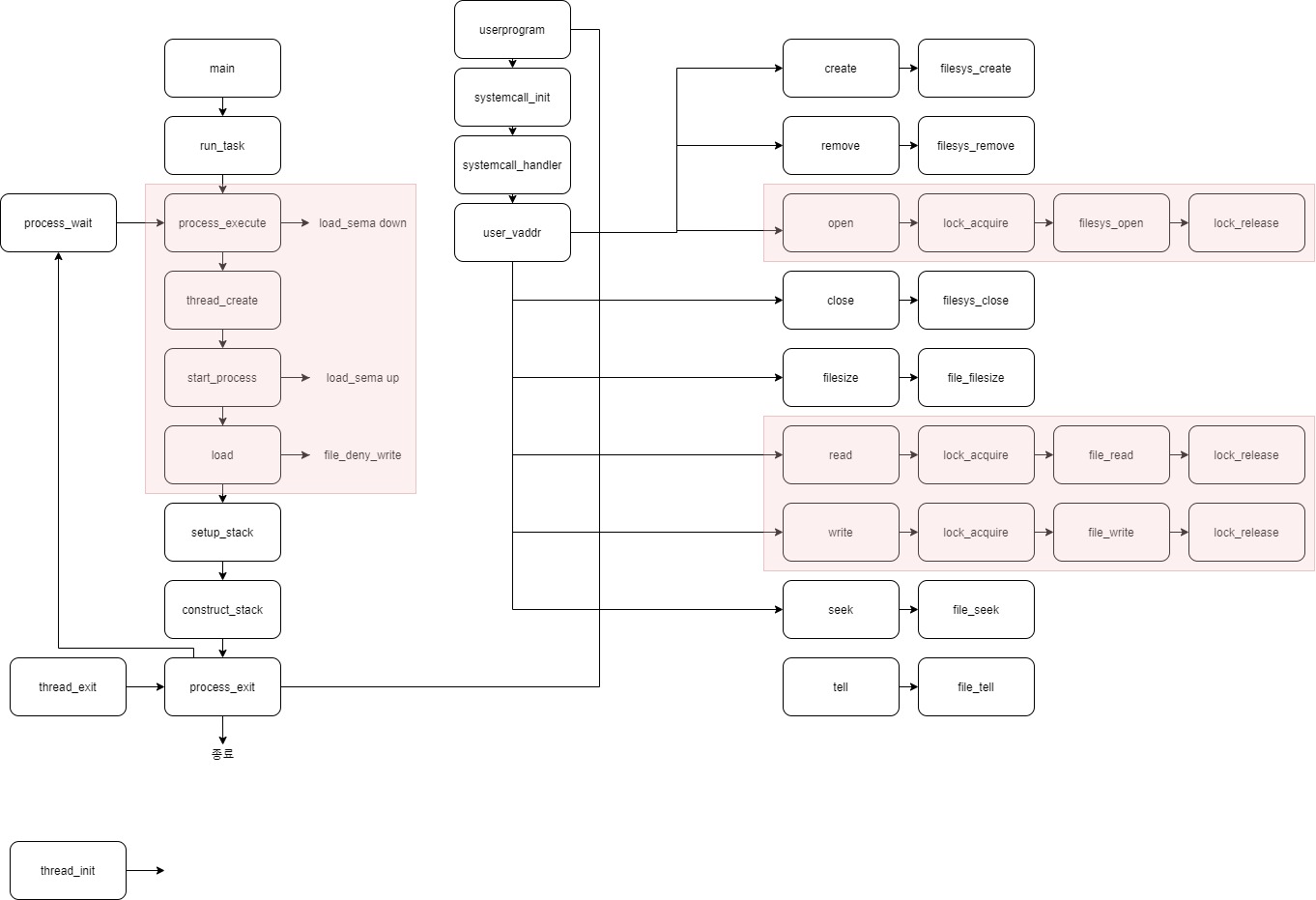
1. **연구 결과**
   1. **Flow Chart**
      1. File Descriptor



* + 1. Systemcalls (추가된 systemcall만 표현)



* + 1. Synchronization in Filesystem



* 1. **제작 내용**
     1. File Descriptor

1. thread/thread.h

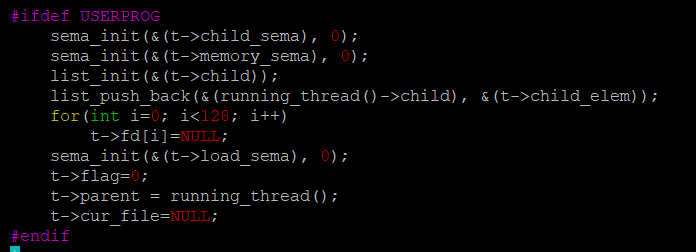
먼저 file descriptor 자료구조 선언에 대한 코드이다.

thread 구조체 내에 file구조체 배열인 fd를 선언하여 file descriptor를 처리하였다.



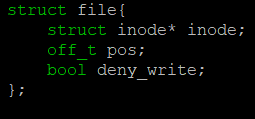
1. thread/thread.c

새로 선언된 fd에 대해서 초기화하는 코드를 추가하였다. init\_thread 함수에서 for문을 이용하여 128개의 배열을 모두 NULL로 초기화하였다.



1. filesys/file.c file.h

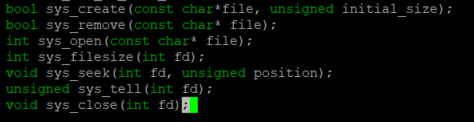
file구조체를 다른 .c 파일에서도 사용할 수 있도록 구조체 선언을 file.c에서 file.h로 옮겨주었다.



* + 1. System calls

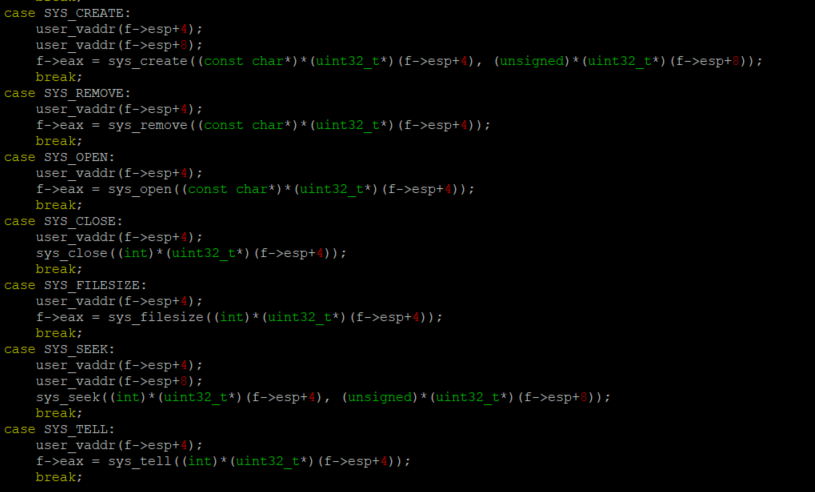
1. userprog/syscall.h

추가로 선언한 함수인 systemcall을 처리하는 함수들의 prototype을 추가했다.



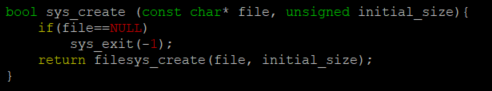
1. userprog/syscall.c

syscall\_handler내에서 switch문을 통해 명령어를 구분한다. 따라서 create, remove, open, filesize, seek, tell, close를 구분하는 case를 추가하여 각 명령어를 처리하는 함수를 호출하도록 하였다. 각 필요한 argument는 esp를 통해 stack에 접근하여 넘겨주었다.

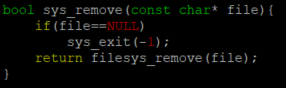


argument를 넘겨주기 이전에 project1에서 구현했던 user\_vaddr함수를 이용하여 memory 접근이 invalid 하지 않은지 검사를 했다.

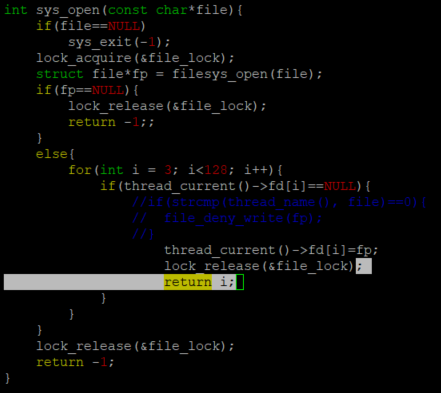
create는 filesys\_create함수를 호출하여 처리하였다. 파일이 NULL이 아닌지 먼저 체크를 하여 에러를 방지하였다. filesys\_create 함수는 filesys/filesys.c에서 기본적으로 제공된다. 이 함수는 주어진 이름의 파일을 주어진 사이즈만큼 만드는 함수이다.



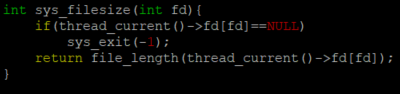
remove는 filesys\_remove함수를 호출하여 처리하였다. 파일이 NULL이 아닌지 먼저 체크를 하여 에러를 방지하였다. filesys\_remove함수는 filesys/filesys.c에서 기본적으로 제공된다. 이 함수는 주어진 이름의 파일을 제거하는 함수이다.



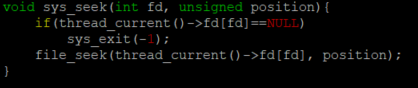
open은 filesys\_open함수를 호출하여 처리하였다. 먼저 파일이 NULL이 아닌지 체크를 하여 에러를 방지하였다. filesys\_open함수는 filesys/filesys.c에서 기본적으로 제공되는 함수인데 주어진 이름의 파일을 여는 함수이다. 만약 성공적이라면 새로운 파일을 리턴하고 아니라면 NULL을 리턴한다. 따라서 sys\_open에서도 filesys\_open의 결과가 NULL이라면 -1을 리턴하도록 하였다. NULL이 아니라면 for문을 통해 fd가 NULL인 것을 찾아 열린 file을 찾은 fd에 저장하고 해당 index를 return하였다.



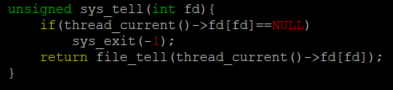
filesize는 file\_length함수를 호출하여 처리하였다. 먼저 fd가 NULL이 아닌지 체크를 하여 에러를 방지하였다. file\_length함수는 filesys/file.c에서 기본적으로 제공되는 함수인데 FILE의 바이트 사이즈를 리턴하는 함수이다.



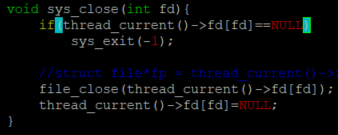
seek은 file\_seek 함수를 호출하여 처리하였다. 먼저 fd가 NULL이 아닌지 체크를 하여 에러를 방지하였다. file\_seek함수는 filesys/file.c에서 기본적으로 제공되는 함수인데 file의 시작점에서 새로운 위치로 현재 위치를 set해주는 함수이다.



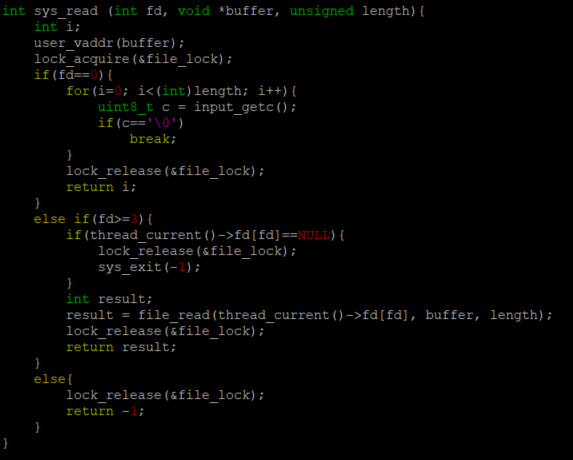
tell은 file\_tell함수를 호출하여 처리하였다. 먼저 fd가 NULL이 아닌지 체크를 하여 에러를 방지하였다. file\_tell 함수는 filesys/file.c에서 기본적으로 제공되는 함수인데 파일의 시작위치에서 현재 위치의 byte offset을 리턴하는 함수이다.



close는 file\_close함수를 호출하여 처리하였다. 먼저 fd가 NULL이 아닌지 체크를 하여 에러를 방지하였다. file\_close 함수는 주어진 파일을 닫는 함수이다. 파일을 닫은 뒤에는 fd를 다시 NULL로 초기화해주었다.

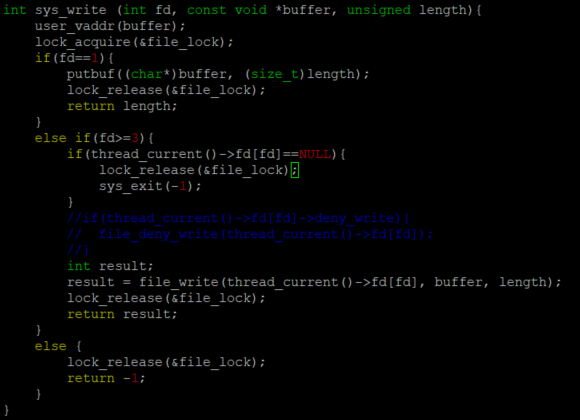


read함수에는 filesystem을 처리하는 코드를 추가해주었다.



먼저 buffer에 대해서 user\_vaddr함수를 통해 invalid한 memory이면 종료를 했다. fd가 3이상이라면 filesystem에 대한 처리를 하였다. fd가 NULL일 때의 에러처리를 한뒤 file\_read함수를 호출하였다. file\_read함수는 filesys/file.c에서 기본적으로 제공되는 함수인데 file에서 buffer로 주어진 size만큼을 읽는 함수이다.

write함수에도 filesystem을 처리하는 코드를 추가하였다.



먼저 buffer에 대해서 user\_vaddr함수를 통해 에러체크를 한 뒤 fd가 3이상이라면 filesystem에 대한 처리를 하였다. fd가 NULL일 때에도 에러처리를 했다. file\_write함수를 호출하였는데 이 함수는 filesys/file.c에서 기본적으로 제공되는 함수로 file로 buffer를 주어진 size만큼 쓰는 함수이다.

1. userprog.exception.c

매핑되지 않은 주소공간에 대한 접근을 하는 testcase (ex, bad-read, bad-write…)에 대해서 계속해서 faile이 떠서 page\_fault에 코드를 추가해주었다. not\_present의 값에 대해서 true라면 page가 not-present하다는 의미이므로 sys\_exit를 호출하여 에러처리를 해주었다. 이코드를 추가하니 해당 testcase에 대해 pass가 떴다.

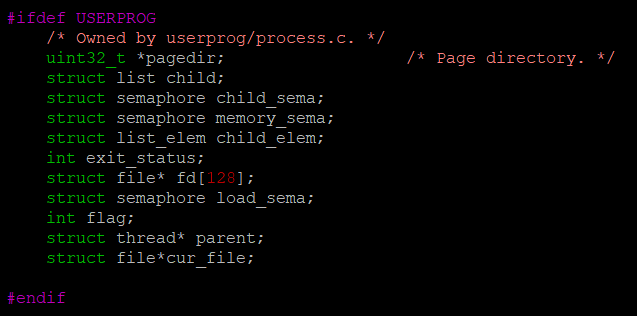
* + 1. Synchronization in Filesystem

1. threads/thread.h

먼저 file 동기화를 위한 file\_lock을 lock구조체로 선언해주었다.

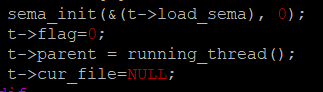


thread 구조체에 process 동기화를 위한 load\_sema와 flag, parent, 그리고 실행중인 파일에 write를 하지못하도록 할 때 이용되는 cur\_file을 추가해주었다.



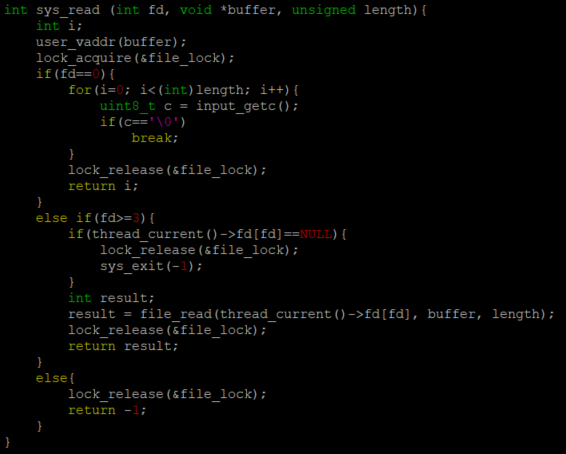
1. threads/thread.c

thread구조체에 추가된 것들에 대해 초기화하는 코드를 추가했다. load\_sema는 sema\_init함수를 통해 초기화를 했다. flag는 0으로, cur\_file은 NULL로 각각 설정하였고 parent는 running\_thread로 값을 할당해주었다.



1. userprog/syscall.c

먼저 여러 process가 하나의 파일에 접근하는 것을 막는 file을 동기화하는 코드를 추가한다. read, write, open 함수내에서 file\_lock에 대해서 lock\_acquire, lock\_release함수를 사용하여 lock을 구현하였다. lock\_acquire은 해당 lock의 semaphore를 sema\_down해주고 lock\_release는 sema\_up을 한다.

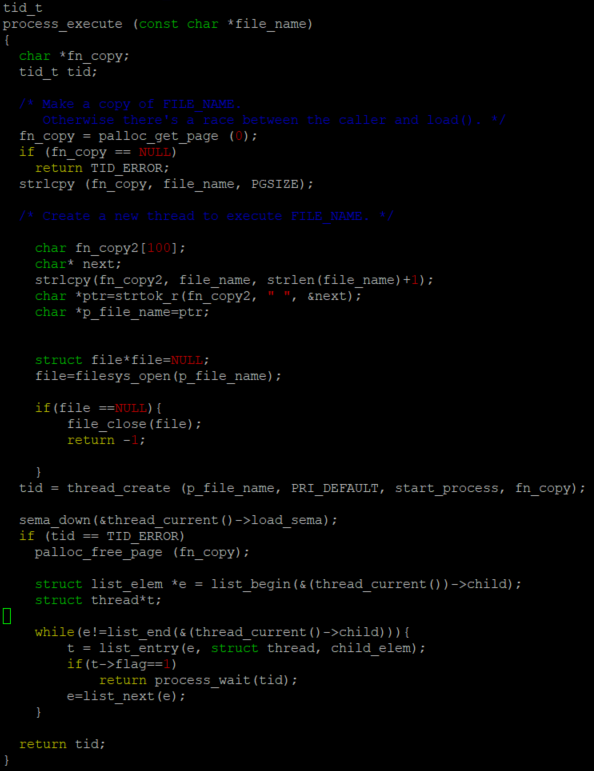
  

따라서 각 함수의 시작에서 file에 대한 접근이 이루어지기 전에 lock\_acquire를 호출하고 함수가 종료되기 전에 lock\_release 함수를 호출하여 파일에 접근을 하는 해당 명령어의 처리가 보호되도록 하였다.

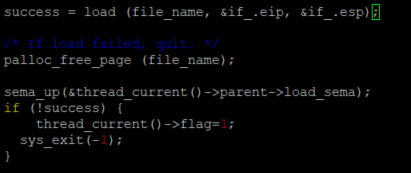
1. userprog/process.c

process 동기화를 구현하는 코드를 추가해주었다.

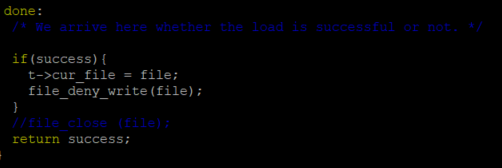
먼저 process\_execute에서 thread가 create된 뒤 현재 thread의 load\_sema를 sema\_down을 하였다. 이를 통해 parent가 child보다 먼저 죽는 경우를 막을 수 있다. 그리고 현재 thread의 child에 대해서 flag가 1이면 process\_wait를 호출하며 return했다.



start\_process에서는 load함수가 끝났을 때 해당 thread의 parent의 load\_sema를 sema\_up해주었다. 이를 통해 load가 정상적으로 종료되지 않은 경우를 보호할 수 있다. load가 정상적으로 종료되지 않았을 경우에는 flag를 1로 처리하여 parent에서 child의 상태를 볼 수 있도록 하였다. flag는 위의 process\_execute에서 사용하였다.

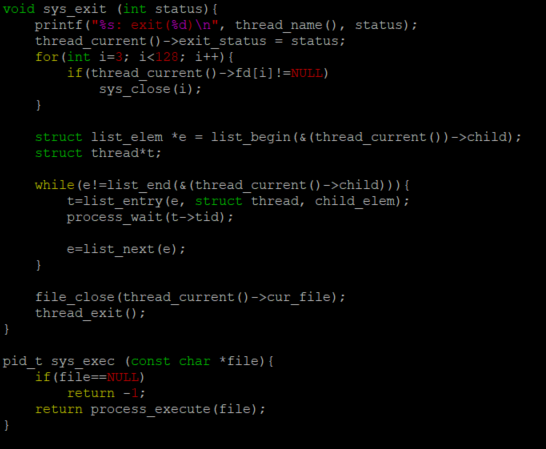


그리고 현재 실행중인 파일에 대해서 write를 막기 위하여 load함수에서 성공적일 때 cur\_file에 file을 저장하고 해당 file에 대해서 file\_deny\_write함수를 호출하여 write되는 것을 막았다. file\_deny\_write함수는 filesys/file.c에서 기본적으로 제공되는 함수로 file의 deny\_write값을 true로 바꾸어 주는 함수이다.



1. userprog.syscall.c

먼저 exit 함수 내에서 parent가 child보다 먼저 끝나지 않도록 while문을 통해 child에 대해서 process\_wait를 호출하여 기다리도록 하였다. child가 모두 실행이 종료된 상태라면 해당 thread를 exit하는데 그전에 cur\_file에 대해 file\_close를 하면서 어느 파일의 실행이 끝나면 close되도록 하였다. 이를 통해 실행중인 파일에 대한 write를 막을 수 있다.



* 1. **시험 및 평가 내용**

