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

담당 교수 / 분반 : 김영재 교수님 / 2반

이름 / 학번 : 장준영 / 20211584

개발 기간 : 2023-10-16 ~ 2023-10-28

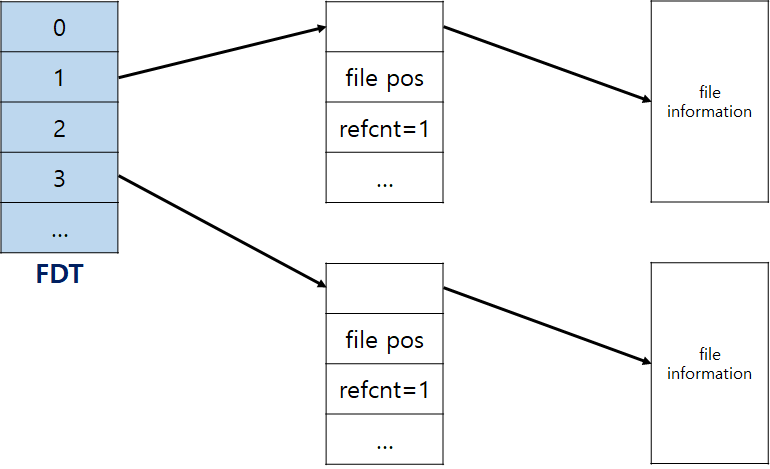
1. **개발 목표**

이번 프로젝트에서는 프로젝트 1에서 구현하지 못한 system call을 마저 구현한다. 구현하거나 수정해야 하는 system call은 create, remove, open, close, filesize, read, write, seek, tell로, OS를 통해 file에 접근하는 기능을 갖고 있다. 해당 system call로 인해 열린 file을 관리하기 위해 추가적인 자료구조를 선언하거나, 파일의 무결성을 보장하기 위해 access synchronization도 구현하여야 한다. 또, 이전 프로젝트에서는 요구하지 않던 복잡한 케이스 테스트를 pass하기 위해 exit, exec과 같은 system call도 그에 맞게 수정되어야 한다. (해당 프로젝트에선 file system에 대한 깊은 이해를 요구하지 않고, pintOS에서 제공하는 file system API를 이용해 후처리를 위탁한다.)

1. **개발 범위 및 내용**
   1. **개발 범위**

**1. File Descriptor**

이번 프로젝트에선 file과 관련된 system call을 구현해야 하기 때문에, 프로세스마다 관련된 file의 data를 관리해주어야 한다. file 자체는 pintOS 내의 file 구조체와 이를 이용하는 file system API를 통해 관리하고, 우리는 그 각각의 파일을 지칭하는 file descriptor를 구현해주어야 한다. file descriptor는 open한 파일에 대한 고유 identifier로, 프로세스에서 사용하는 file을 구별할 수 있게 해준다. 이 file descriptor는 모든 프로세스마다 고유하게 가지고 있는 file descriptor table을 이용해 관리하고, 프로세스는 이를 통해 다양한 file operation을 할 수 있다.



시스템프로그래밍 시간에 배운 file table의 추상화이다. 프로세스 내에서 file descriptor라는 identifier를 통해 FILE \*를 저정하면, OS 내의 file system이 그에 맞는 파일의 data를 제공한다. 이를 통해 이번 프로젝트에서 구현해야 하는 system call이 올바르게 파일을 인식할 수 있다.

**2. System Calls**

UNIX 계열 운영체제에서는 모든 정보를 file로 관리한다. 디렉토리, 소켓, 파이프, 디바이스 등 여러 기능이 file로 관리되기 때문에, 프로세스가 file에 접근하고 제어할 수 있는 기능이 필요하다. 이는 OS를 거쳐가는 system call로 구현되어야 하는데, 유저 프로세스가 접근 불가한 파일에 접근하거나 OS 코드를 수정하는 등 보안과 시스템 안전 상에 큰 문제를 일으킬 수 있는 동작을 막기 위해서이다. 이번 프로젝트에서는 가장 기본적인 기능인 create, remove, open, close, filesize, read, write, seek, tell을 구현하여 프로세스가 file에 접근할 수 있도록 한다.

**3. Synchronization in Filesystem**

앞서 file descriptor의 설명에서도 볼 수 있듯, 복수의 프로세스가 하나의 file data에 접근할 수 있기 때문에 synchronization이 필요하다. 시스템프로그래밍 수업 시간에 배웠던 readers-writers problem 같은 것이 대표적인 예시이다. 구현해야 하는 모든 system call에 synchronization이 필요하진 않고, 값의 변조가 있는 open, read, write에서 지켜주어야 한다. pintOS의 synch.\*에서 제공하는 synchronization 기능으로 lock과 semaphore가 있는데, semaphore는 counting semaphore로 초기값을 1보다 큰 수로 잡을 수 있지만, lock은 binary semaphore로 0과 1 사이를 진동하기만 한다는 차이가 있다. 두 방식 모두 critical section을 보호하는 기능을 수행할 수 있고, 이번 프로젝트에서의 경우 binary semaphore, 즉 lock을 사용하는 것이 수월했다. 왜냐하면, pintOS에서 lock은 해당 lock을 설정한 프로세스(쓰레드)만이 그것을 해제할 수 있고, semaphore는 그렇지 않기 때문이다. pintOS의 매뉴얼을 자세히 읽어보면, file system과 관련된 critical section에 어떤 프로세스가 진입했을 때 다른 그 어떤 프로세스도 그 영역에 진입할 수 없게 해야한다는 조건 존재하기도 한다. 사실 구현의 차이이겠지만, 해당 조건과 더 잘 부합하는 기능이 lock이라고 판단하였다. 이 외에도 이미 실행되고 있는 프로세스의 executable file에 write하는 것을 막기 위해 file\_deny\_write / file\_allow\_write 등의 함수를 통한 synchronization도 필요하다.

* 1. **개발 내용**

**1. File Descriptor**

File descriptor table은 단순히 file pointer array를 프로세스(쓰레드)마다 하나씩 두는 것으로 구현할 수 있다. 그렇게 하면, 해당 array의 index는 file descriptor, 그 값인 file pointer는 open file(struct file)에 대한 pointer가 될 것이기 때문이다. Linear array가 시간적 손해를 가진다고 생각할 수 있지만, pintOS 상의 FDT는 128이 최대 크기이기 때문에 무시할 수 있다. 또 구현이 아주 간단해 여러 routine에서 FDT를 편리하게 참조할 수 있다는 장점도 있다.

**2. System Calls**

1) create : file\_name을 이름으로 갖는 file을 size의 크기로 생성하고, 성공적으로 생성했다면 true를, 아니면 false를 return한다. create의 경우 filesys/filesys.\*에 존재하는 filesys\_create 함수에 모든 기능을 위탁한다.

2) remove : file\_name을 이름으로 갖는 file을 시스템 내에서 삭제한다. 이 경우도 create와 마찬가지로 filesys\_remove 함수에 모든 기능을 위탁한다.

3) open : file\_name을 이름으로 갖는 file을 filesystem에서 찾아 해당 프로세스 내에서 open해준다. filesys\_open을 통해 file을 열면 해당 file의 open file table 값(file pointer)가 생기는데, 이를 해당 프로세스의 FDT에 저장해주면 된다.

4) close : open을 통해 연 file을 닫아준다. file\_close라는 함수에 모든 기능을 위탁하지만, 여기에는 open으로 인해 추가된 FDT의 값을 NULL로 다시 바꿔주는 기능이 없기에 이 부분은 추가적으로 구현해야 한다.

5) filesize : 말 그대로 filesize를 return한다. file\_length 함수에 모든 기능을 위탁한다.

6) read : file descriptor가 가리키는 open file의 정보를 offset(position)에서 size byte만큼 읽어 지정된 buffer에 저장한다. 이전에 구현한 read는 stdin에 대한 콘솔 입력만 읽었지만 이번엔 file에서 읽어와야 하므로 수정된다.

7) write : file descriptor가 가리키는 open file에 offset(position)에서 size byte만큼 작성한다. read와 마찬가지로 stdout에 대한 콘솔 출력 외에 file에 대한 출력이 되도록 해야한다.

8) seek : file descriptor가 가리키는 open file의 offset(position)을 지정한 값으로 바꿔준다. file\_seek 함수에 모든 기능을 위탁한다.

9) tell : file descriptor가 가리키는 open file의 offset(position)을 알려준다. file\_tell 함수에 모든 기능을 위탁한다.

**3. Synchronization in Filesystem**

앞서 **A.3.**에서 설명하였듯 lock과 semaphore 중 lock을 이용해 synchronization을 진행할 것이다. (왜냐하면, pintOS에서 lock은 해당 lock을 설정한 프로세스(쓰레드)만이 그것을 해제할 수 있고, semaphore는 그렇지 않기 때문이다. pintOS의 매뉴얼을 자세히 읽어보면, file system과 관련된 critical section에 어떤 프로세스가 진입했을 때 다른 그 어떤 프로세스도 그 영역에 진입할 수 없게 해야한다는 조건 존재하기도 한다. 사실 구현의 차이이겠지만, 해당 조건과 더 잘 부합하는 기능이 lock이라고 판단하였다. 서로 다른 프로세스 간의 interleaving이 가능한 semaphore를 이용한 synchronization은 filesystem에 관련된 것이 아니고, parent-child process의 memory leakage protection에서 사용하였다.) 여러 차례 학습했듯 shared data의 변조에는 항상 synchronization이 필요하므로, read, write, open system call을 구현 및 수정하면서 critical section을 lock으로 잠가주도록 할 것이다. 자세한 내용은 이후 서술하겠다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

이번 프로젝트는 지난 프로젝트 1과 달리 system call 구현을 맨 처음부터 시작하지도 않았고, file system에 관련해서는 pintOS의 API만을 사용했기 때문에 구현이 수월하였다. 우선 이번 프로젝트에서 새로이 쓰게 된 filesys/file.\*, filesys.\*에 대한 이해가 필요하다고 판단해 이를 전부 읽어보았고, 이를 통해 지원하는 구조체나 함수가 어느 system call에서 쓰일지 미리 판단하였다. 이후 구현을 시작하였는데, 표면적으로 프로젝트 2에서 요구하는 내용인 system call about filesystem을 전부 작동하도록 구현하는 데는 이틀 정도, 그리 오랜 시간이 걸리지 않았다. 이를 통해 기초적인 test는 모두 pass를 받을 수 있었다. 하지만, 도무지 pass가 되지 않는 복잡한 case의 test(multi-oom, rox-child, rox-multichild, …)가 있었기 때문에, userprog 내의 test file을 모두 읽어보면서 오류를 디버깅하였다. 이 과정에서 memory leakage protect, reaping every child 등을 위한 복잡한 구현이 필요했기 때문에, 5일 정도 정보를 수집하고 코드를 수정하였다. 이후 이해하고 수행한 내용을 바탕으로 보고서를 작성하였다.

* 1. **개발 방법**

**1. File Descriptor**

프로세스마다 하나의 FDT를 가지면 되기 때문에, thread.h의 struct thread에 file pointer array를 하나 선언한다. 이는 thread.c의 init\_thread 함수에서 모두 NULL로 초기화된다. 이후 많은 routine에서 thread\_current()->fd[i]로 참조된다.

**2. System Calls**

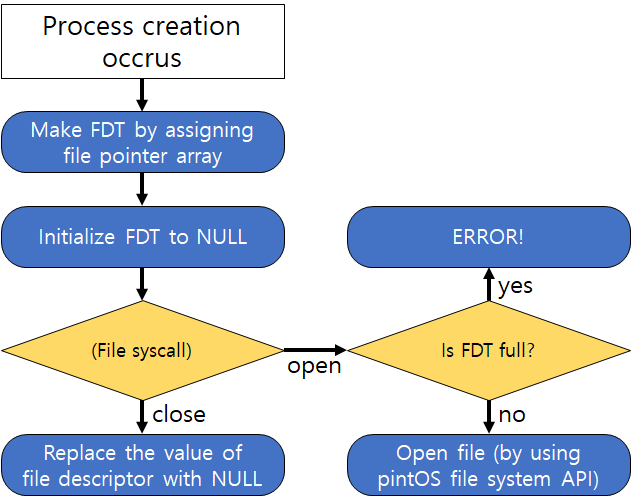
이전에 system call handler와 syscall table routine을 모두 작성해두었으므로, 거기에 추가적으로 filesystem과 관련된 system call을 작성하기만 하면 된다. 이는 process.\*에서 이루어진다. syscall table routine(switch 구문)에 새로 구현할 system call에 대해 user memory validation을 진행하고, 알맞은 동작을 하도록 만들어주면 된다. system call의 동작 부분을 위해 open, close 등의 함수를 새로 작성하였고 이는 process.h에도 추가되었다. 프로젝트 1에서 모두 설명한 내용이므로 이만 마치겠다.

**3. Synchronization in Filesystem**

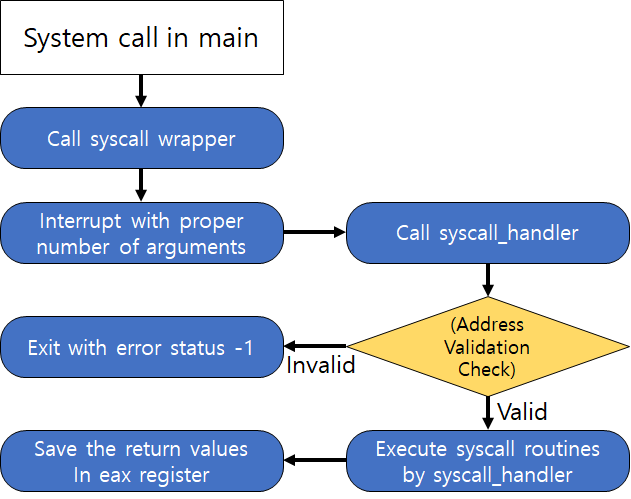
Synchronization을 위해 lock이라는 새로운 기능을 도입했기 때문에, 이를 제대로 이용해야 한다. 또, filesystem에 대해서도 이해해야 하기 때문에 file.\*와 filesys.\* 내의 struct file로 표현된 open file table에 대해서 알아야 한다. syscall.c 내의 system call handler에서 file에 대한 직접적인 access를 하기 때문에, 여기서 lock을 이용한다. lock은 선언과 초기화가 필요한 변수이기 때문에 syscall.h에서 미리 선언하고, syscall.c의 syscall\_init routine에서 trap instruction이 초기화될 때 lock도 동시에 초기화한다. 이후 synchronization이 필요한 system call인 open, read, write의 routine 내에서 lock을 적절히 이용하면 된다. 또, running executable file에 대한 수정을 막기 위해 filesys.\*의 file\_deny\_write 함수를 사용한다. 이는 파일 실행 루틴인 open, 작성 루틴인 write 내에서 적용된다.

1. **연구 결과**
   1. **Flow Chart**

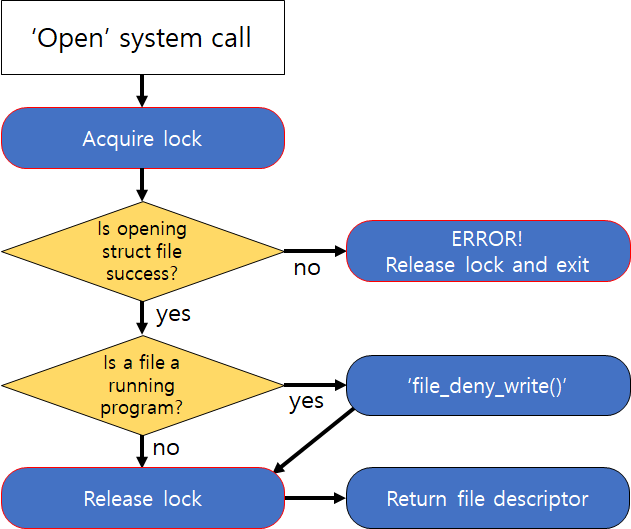
**1. File Descriptor**



**2. System Calls**



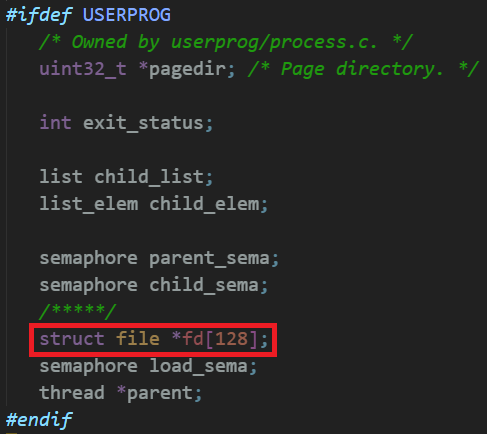
**3. Synchronizations in File System**

****

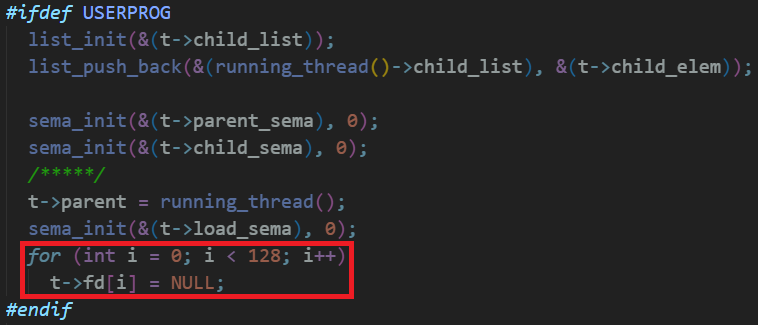
* 1. **제작 내용**

**1. File Descriptor**

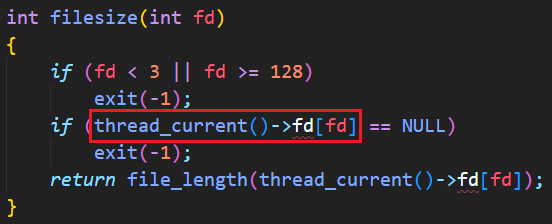
앞서 계속 설명한 내용이다. thread마다 하나의 FDT를 갖도록 구현해주면 된다.



thread.h의 struct thread 부분이다. open file table인 struct file \* 타입의 array를 만들어준다. pintOS 상에서 최대 open file 개수는 128개이므로, 크기를 128로 만들어준다. Local declaration으로 garbage value가 들어있으므로, 프로세스 초기화 시에 FDT도 초기화해주어야 한다.



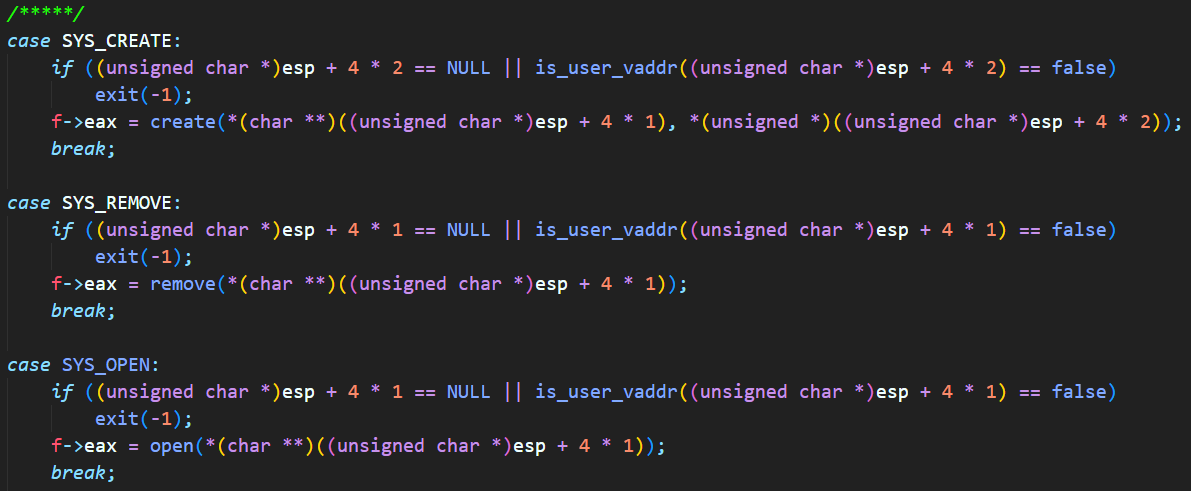
thread.c의 init\_thread 함수 내부이다. 현재 thread의 fd를 모두 NULL로 만들어준다. 이렇게 하면 FDT를 사용할 준비를 모두 마치게 된다. 이를 사용하는 간단한 예시를 보자.



위 사진과 같이, 현재 thread의 FDT를 참조하고 싶으면 thread\_current()->fd를 호출하면 된다.

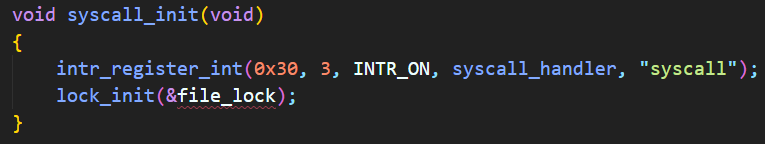
**2. System Calls / 3. Synchronization in Filesystem**

System call은 이전에 구현해 둔 syscall.\*에 추가만 해주면 된다.



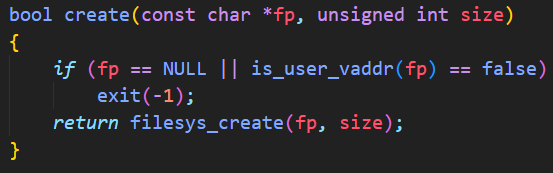
syscall.c의 syscall\_handler switch 구문 내부이다. 이전과 마찬가지로, user pointer의 validity를 체크하고 inturrupt frame의 eax register에 routine의 결과값을 넣어준다. 각각의 routine은 이후 설명하겠다.





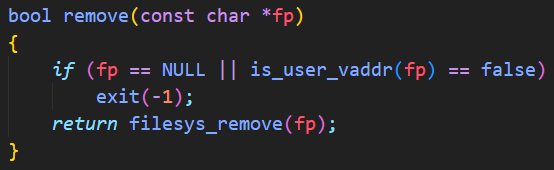
routine을 설명하기 앞서, 우리는 이번 system call을 구현하기 위해 synchronization 방법인 lock을 사용할 것이다. syscall.h에서 file system call에 사용될 lock인 file\_lock을 선언하고, syscall\_init 내에서 lock\_init 한다. 이렇게 하면 이후 file\_lock을 acquire, release하며 사용할 수 있게 된다.

a) create



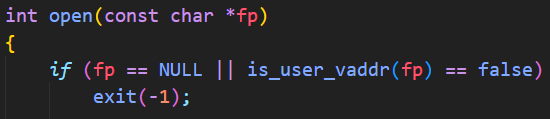
filesys.\* 내에 있는 함수인 filesys\_create를 이용하면 pintOS의 filesystem 내에 파일이 생성된다. switch 구문에서 user pointer validation을 진행했다고 하더라도, 가리키고 있는 fp가 NULL이거나 user address 영역이 아닐 수도 있기 때문에 추가적인 체크를 진행한다.

b) remove

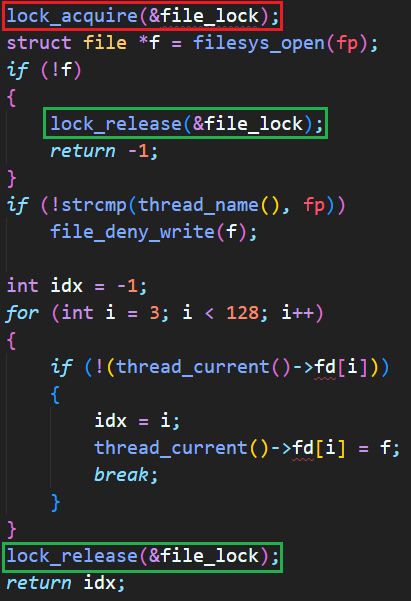


filesys.\* 내에 있는 함수인 filesys\_remove를 이용하면 pintOS의 filesystem 내에 파일이 제거된다. switch 구문에서 user pointer validation을 진행했다고 하더라도, 가리키고 있는 fp가 NULL이거나 user address 영역이 아닐 수도 있기 때문에 추가적인 체크를 진행한다.

c) open

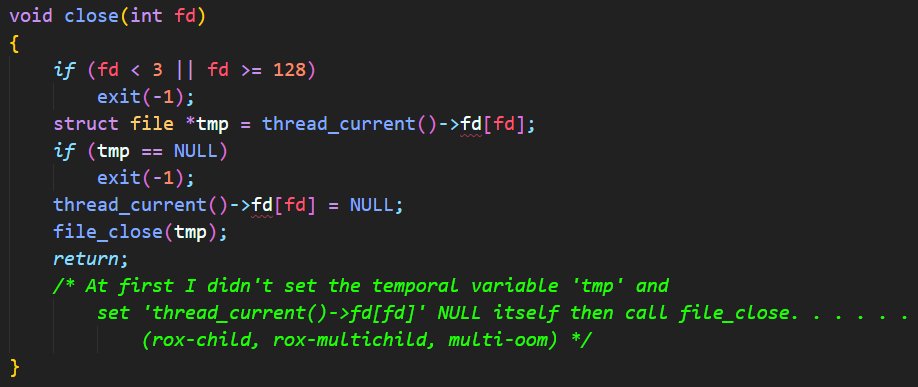


우선 create, remove와 동일한 check routine을 우선 거친다. 이 다음부터가 실질적인 open routine으로, open 자체는 filesys\_open 함수를 사용하여 마칠 수 있지만 synchronization 과정은 critical section을 보호하기 위한 lock과 file\_deny\_write 함수의 활용이 중요하다.



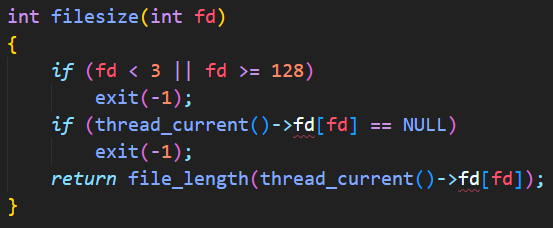
lock\_acquire부터 lock\_release까지가 lock으로 보호되고 있는 critical section이다. filesys\_open으로 file을 열기 전 lock\_acquire로 다른 프로세스의 접근을 막는다. 실행이 실패하면 lock\_release를 해주고 -1 error code를 return 해준다. 또, 현재 수행중인 프로그램의 파일 명과 open한 file이 동일하다면 file\_deny\_write로 file의 변조를 막는다. 이후, thread의 FDT 내에 새로 open한 file을 넣어주기 위해 순차 탐색으로 비어있는 곳을 찾는다. 찾았다면 해당 인덱스에 open file pointer를 삽입한다. 이 과정이 끝나면 lock\_release로 lock을 해제하고, FDT 내의 idx를 return 한다(삽입이 실패했다면 초기값인 -1이 return 된다). 이렇게 하면 유저 프로세스의 FDT에 실행된 파일 정보를 넣을 수 있다.

d) close



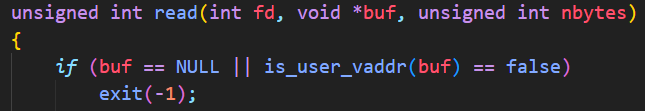
close 하고자 하는 fd가 3보다 작거나(stdin, stdout, stderr or negative), 128보다 크다면 FDT 내에서 제거할 수 없다. 이후 제거하고자 하는 file pointer를 tmp라는 임시 변수에 저장하고, fd index의 값을 NULL로 바꿔준다. 그리고 tmp에 저장한 open file을 file\_close 해준다. 맨 처음에 구현했을 땐 tmp로 file pointer를 임시 저장하지 않고, fd를 NULL로 바꾼 후 또 그 값을 file\_close 했었다. 당연히 오류가 났고 (memory leakage로 인해) rox-child, rox-multichild, multi-oom 등의 test에서 pass를 받을 수 없었다. 기초적인 실수이지만 구현이 간단하다보니 쉽게 찾지 못해서 거의 이틀 내내 디버깅했다.

e) filesize

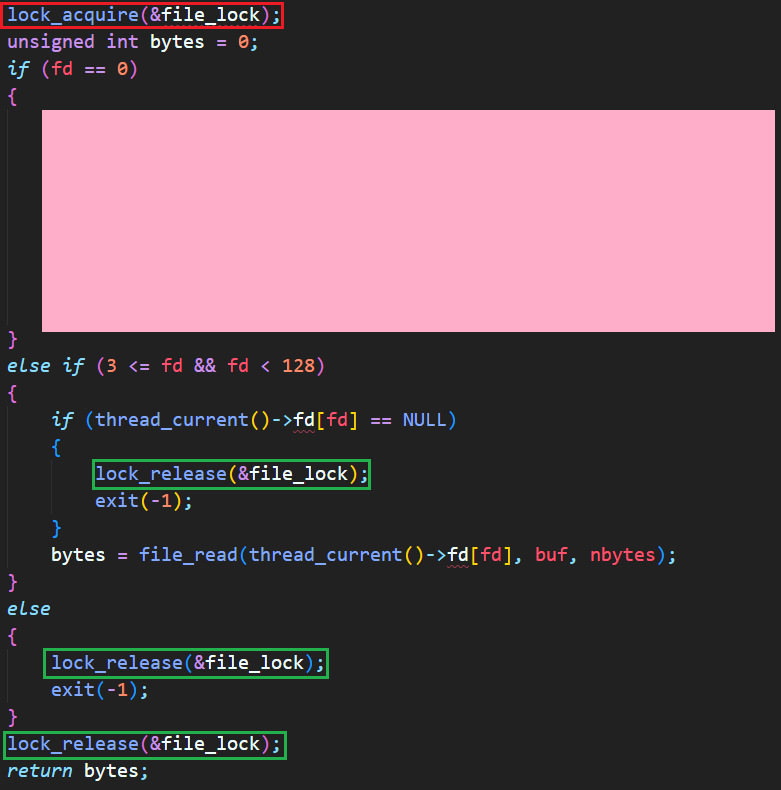


filesize는 open되어 FDT에 저장되어 있는 user file에 대해서만 실행이 가능하다. fd가 그 영역에 없거나 FDT에 삽입되어있지 않다면 exit(-1) 한다. 이후 과정은 file\_length에 모든 기능을 위탁한다.

f) read

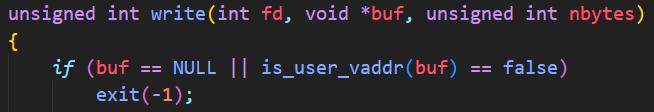


read 역시 user pointer에 대한 기본적인 check routine을 거친다. 이후부터는 synchronization 영역이다.

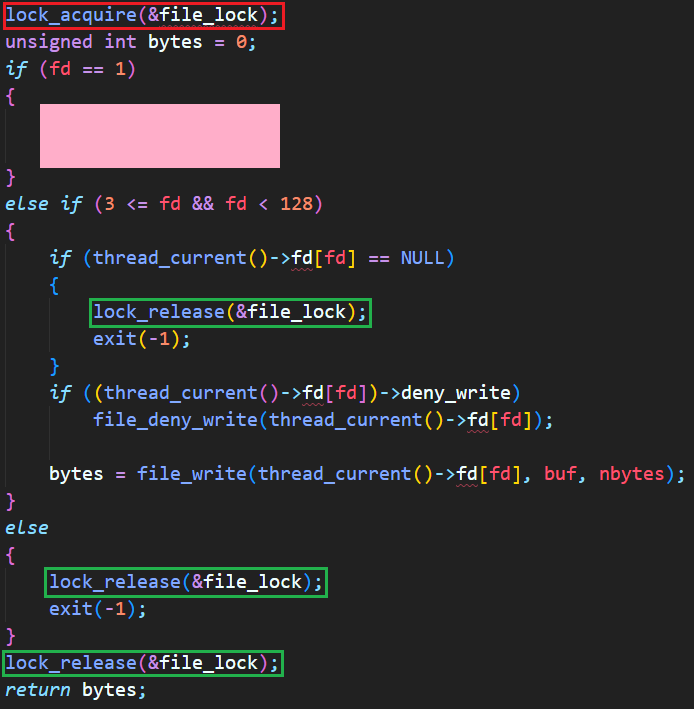


우선 file을 읽는 과정이므로 다른 프로세스가 file에 접근할 수 없도록 막는다(다소 overhead가 클 수 있는 synchronization 방법이지만 안정성이 보장된다). fd가 0인 상황은 지난 프로젝트에서 설명한 stdin에 대해 읽는 과정이므로 생략하고, 3과 128 사이의 user file에 대한 read 과정을 보자. FDT에 값이 없는, 실행되지 않은 file에 대한 read라면 lock을 해제하고 exit(-1) 한다. 그렇지 않다면, file\_read에 기능을 위탁하고 읽어온 만큼 bytes에 저장한다. fd의 값이 정상적이지 않다면 lock을 해제하고 exit(-1) 한다. 모든 read 과정이 끝나면 lock을 해제하고 읽은 bytes를 return 한다.

g) write

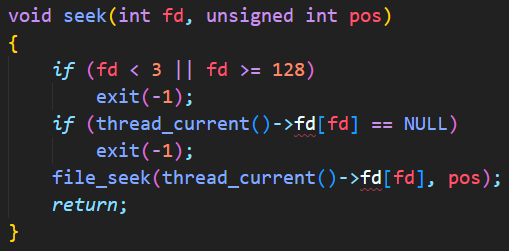


write도 read와 마찬가지로 user pointer에 대한 validation 이후 synchronization이 필요한 구간을 거친다.



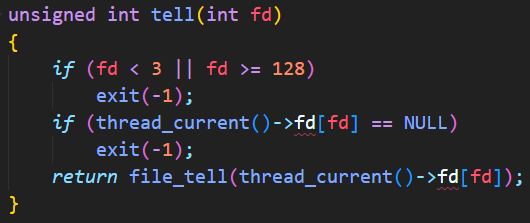
우선 file에 쓰는 과정이므로 다른 프로세스가 file에 접근할 수 없도록 막는다. fd가 1인 상황은 지난 프로젝트에서 설명한 stdout에 쓰는 과정이므로 생략하고, 3과 128 사이의 user file에 대한 write 과정을 보자. FDT에 값이 없는, 실행되지 않은 file에 대한 write라면 lock을 해제하고 exit(-1) 한다. 또, 이전의 file\_deny\_write로 인해 open file table 내의 deny\_write 값이 true인 경우 그 파일에 쓸 수 없도록 file\_deny\_write 해준다. 이후 file\_write에 기능을 위탁하고 작성한 만큼 bytes에 저장한다. fd의 값이 정상적이지 않다면 lock을 해제하고 exit(-1) 한다. 모든 write 과정이 끝나면 lock을 해제하고 읽은 bytes를 return 한다.

h) seek



seek은 open되어 FDT에 저장되어 있는 user file에 대해서만 실행이 가능하다. fd가 그 영역에 없거나 FDT에 삽입되어있지 않다면 exit(-1) 한다. 이후 과정은 file\_seek에 모든 기능을 위탁한다.

i) tell



tell은 open되어 FDT에 저장되어 있는 user file에 대해서만 실행이 가능하다. fd가 그 영역에 없거나 FDT에 삽입되어있지 않다면 exit(-1) 한다. 이후 과정은 file\_tell에 모든 기능을 위탁한다.

**4. Additional Memory Protect & Reaping**

userprog의 다양한 test를 통과하기 위해선, 이전에 구현한 부분에 대한 세밀한 조정이 필요했다. 이 과정에 대한 내용을 설명한다.

a) exception.c의 error check routine 수정



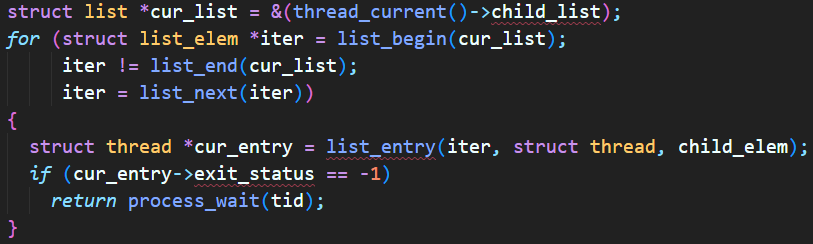
이전에 있었던 조건 외에 not\_present 조건이 추가되었다. 원활한 reaping을 위해 오류로 종료되는 모든 프로세스를 -1의 exit status를 가지게 해야하기 때문에, load 과정에서 오류가 발생해 not\_present option이 true가 되어도 exit(-1) 하게 만들어준다.

b) process.c의 start\_process 내 error check routine 수정



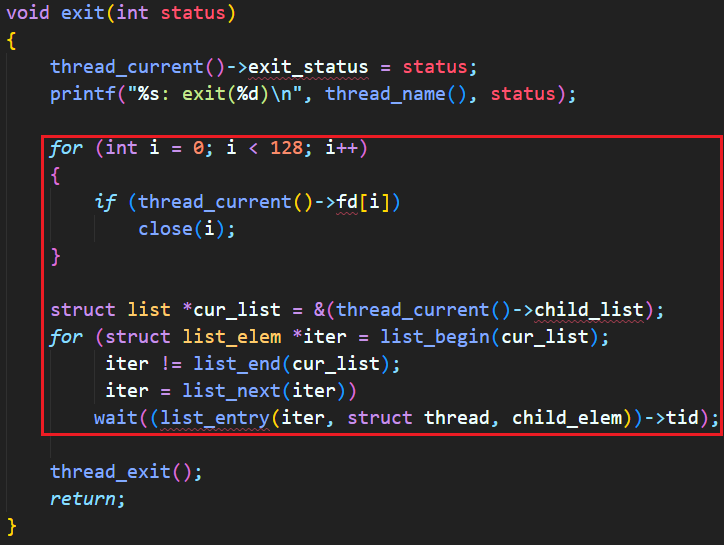
a)에서 설명한 이유와 마찬가지로 start\_process가 실패하면 -1의 exit status를 가지고 종료될 수 있도록 수정한다.

c) process.c의 process\_execute 내의 parent process busy waiting



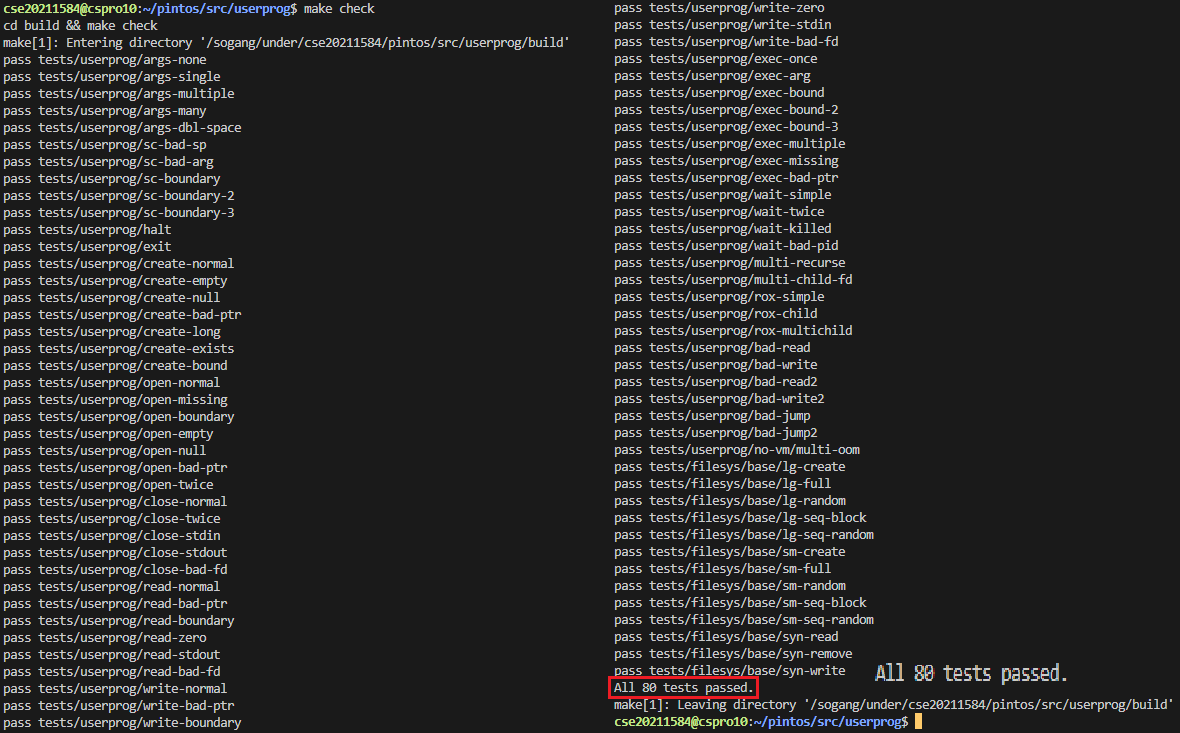
process\_execute에선 fork를 진행하는데, 이때 자신이 생성시킨 모든 child process가 비정상 종료된 상황에 대비해 busy waiting을 실시한다. 이렇게 하면 exec system call이 불릴 때마다 반복문을 통해 자신의 child가 비정상 종료되어도 reaping 할 수 있다.

d) syscall.c의 SYS\_EXIT routine 수정



multi-oom test가 도무지 pass되지 않아서 자료를 조사하던 도중 찾게 된 방법이다. parent process에 대한 busy waiting만 한다고 모든 memory leakage를 막을 수 있는 것이 아니다. exit하는 프로세스 자신의 필요 없는 data를 비워주어야 한다. orphaned process를 만들지 않기 위해, exit 처리가 되기 이전에 모든 child를 reaping 해준다. 또, 종료되는 프로세스의 FDT는 더 이상 필요가 없으므로 열린 file을 모두 close 해준다(이는 필요 없이 open file table로 남아있는 file을 없애줄 수 있다). 이렇게 하면 쓸모없이 메모리 공간을 차지하는 값들을 없애주고 exit을 마칠 수 있다.

* 1. **시험 및 평가 내용**



모든 항목에서 pass를 받았다. **All 80 tests passed.**