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

담당 교수 / 분반 : 김영재 / 2

이름 / 학번 : 원종윤 / 20201604

개발 기간 : 2023.10.26 – 2023.10.29

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이전 프로젝트에서 구현하지 못한 system call을 마저 구현하고, thread 별 file descriptor 기능을 추가한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

File descriptor 구현을 통해, 한 thread가 STDIN/STDOUT 뿐만 아니라 다른 file에 입, 출력을 수행할 수 있다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

Write, read, create, remove, open, close, filesize, seek, tell을 구현해야 한다.

3. Synchronization in Filesystem

여러 process가 read, write에 접근하여도 data corruption이 발생하지 않도록 lock과 semaphore를 적절히 구현한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

Thread 자료구조 내부에 file descriptor를 위한 배열 fd를 선언한다. Fd[0], [1]은 stdin/out에 할당되고, fd[2] 부터 새로이 할당한다. 최대 개수가 정해져 있으므로 배열을 선택했다.

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)

Write: 기존 구현 부분에 STDOUT 뿐만 아니라 새로운 file descriptor에 출력할 수 있도록 구현한다. 이 때, predefined filesystem 함수를 사용한다.

Read: 기존 구현 부분에 STDIN 뿐만 아니라 새로운 file descriptor에 출력할 수 있도록 구현한다. predefined filesystem 함수를 사용한다.

Create: name과 size에 해당하는 file을 생성한다. 성공하면 true를 반환한다. 메모리할당 실패 혹은 중복 시 실패한다. predefined filesystem 함수를 사용한다.

Remove: 전달받은 name의 file을 제거한다. 성공하면 true를 반환한다. 존재하지 않는 파일에 시도하면 실패한다. predefined filesystem 함수를 사용한다.

Open: 전달받은 file을 predefined filesystem 함수를 사용하여 생성한다. 이후, 현재 thread의 fd에 빈 자리를 확인하여 빈 자리가 존재한다면 할당해준다. 이 때, thread\_current()의 name과 같다면 ROX를 위해 deny\_write을 호출한다.

Close: 전달받은 fd를 NULL로 초기화하고 연결된 file을 닫는다.

Filesize: 현재 file의 size를 반환한다. predefined filesystem 함수를 사용한다.

Seek: file의 현재 position을 NEW\_POS로 설정한다. predefined filesystem 함수를 사용한다.

Tell: 현재 position을 반환한다. predefined filesystem 함수를 사용한다.

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

Read, open, write, process에 대해서 semaphore와 lock을 사용해야 한다. 우선, read, open, write에 여러 process가 동시에 접근하지 못하게 syscall init에서 file system을 위한 sync\_lock을 시작하고, read, write, open의 시작과 끝에 lock acquire-release를 설정한다.

이후 process\_execute 함수에서 thread를 생성한 thread(부모)가 sema\_down을 호출하여 대기하도록 하고, start\_process에서 자식 thread가 parent의 sema를 up하여 다시 진행하도록 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10.26-10.27 system call 구현. File descriptor 구현

10.28-10.29 synchronization 구현

* 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: 구현을 위해 thread.h에 file descriptor array fd[128]을 추가한다. 이후 thread.c에서 thread 초기화 작업에 fd에 NULL을 저장하도록 추가한다.

Syscall: 우선, create, remove, filesize, seek, tell, close 구현을 위해 단순히 predefined 함수를 이용한다. 이후 read 구현을 위해 이전 project에서 구현한 부분을 if(fd==0){} block에 삽입하고, else if(fd>1) 부분을 새로 추가한다. 두번째 block에선 현재 thread의 file descriptor를 이용해 file\_read를 수행하고 이를 반환한다.

Write도 마찬가지다. If 조건문을 분기한 후 두번째 block에서 전달받은 fd에 file\_write를 수행하고 이를 반환한다.

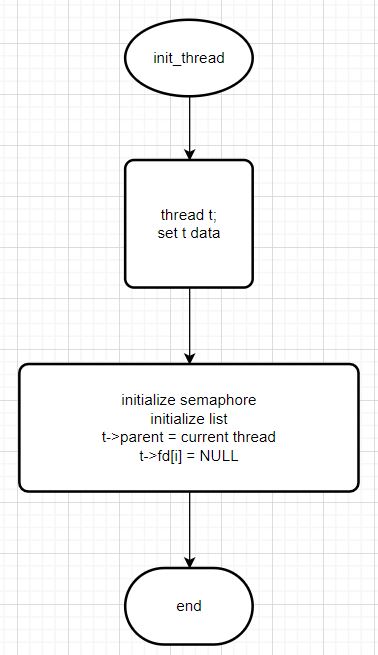
Open에서, 새로 open할 file을 위해 new\_file을 선언하고 filesys\_open을 이 곳에 저장한다. 이후, NULL이 아니라면 현재 thread의 fd를 순회하여 빈 자리를 찾는다. 이 때, 실행 중인 file과 새 file의 이름이 같다면 deny\_write을 호출한다. 이후 빈 자리에 새 file을 할당하고 그 결과를 return한다.

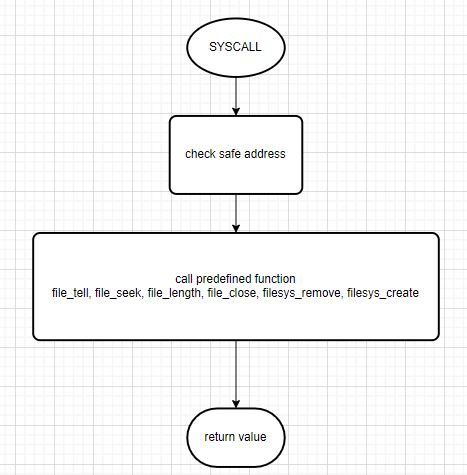
Synchronization: 우선, lock을 이용한 file system data corruption을 방지하기 위해 전역 변수 sync\_lock을 추가한다. 이후 이를 syscall\_init에서 lock\_init으로 초기화한다. 이후 read, write, open 함수의 시작과 끝 부분에 lock acquire-lock release를 추가하여 충돌을 방지한다.

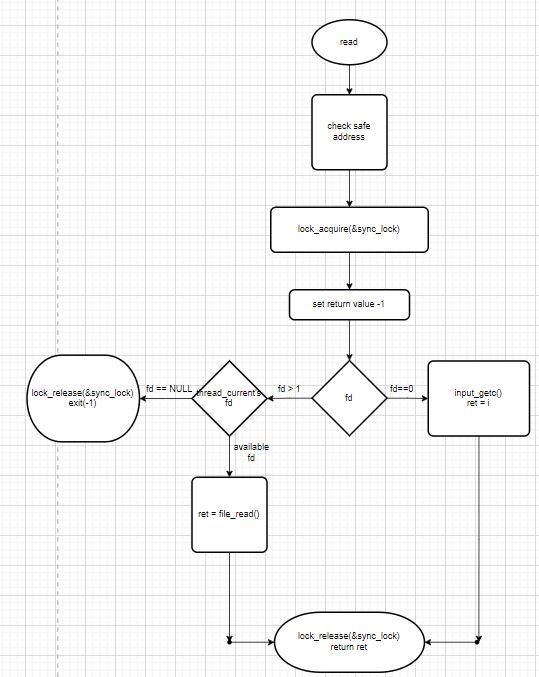
다음으로, semaphore를 이용해 parent thread가 먼저 죽는 것을 방지한다. process\_execute 함수에서 thread를 생성한 thread(부모)가 sema\_down을 호출하여 대기하도록 하고, start\_process에서 자식 thread가 parent의 sema를 up하여 다시 진행하도록 한다. 이를 위해 thread.h의 struct thread에 struct thread\* parent를 추가하고, 이를 thread.c의 thread initialize할 때 running\_thread()를 가리키도록 하여 구현한다. 또한 sema\_load를 추가하여 load 과정에서 parent thread가 기다렸다 child thread를 확인하고 죽을 수 있도록 구현한다.

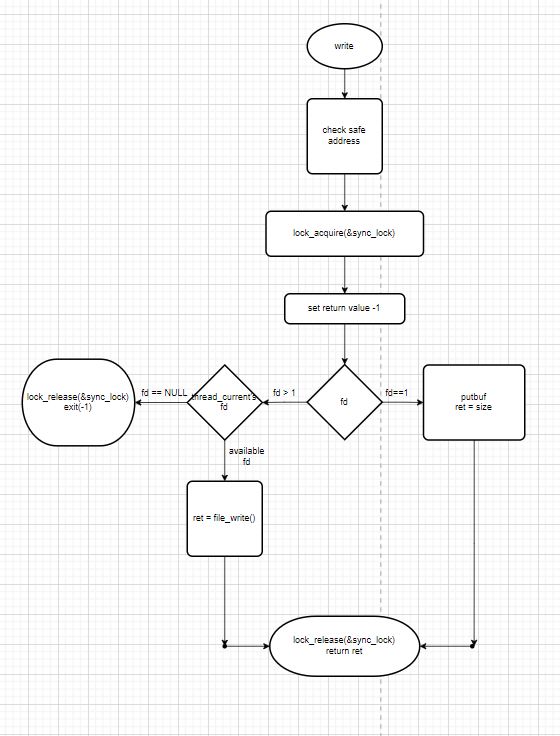
**연구 결과**

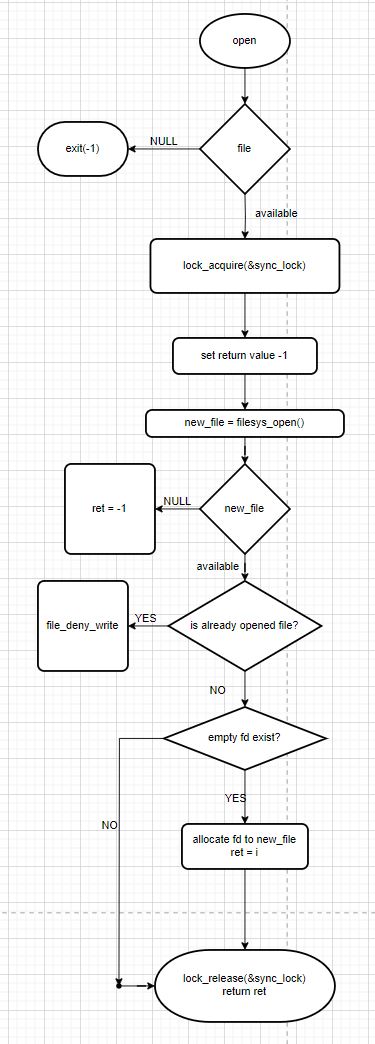
* 1. **Flow Chart**
* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성

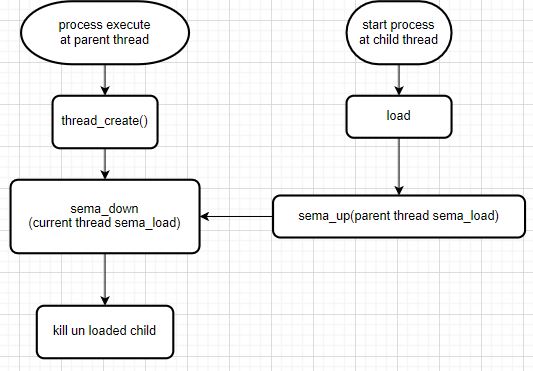












* 1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)

File descriptor의 구현은 단순하다. Pintos가 제공하는 struct file을 이용한 배열 fd[128]을 thread 내부에 심는다. 이후 이를 관리할 때, 0번과 1번은 STDIN-OUT에 해당한다는 것에 유의한다. 이들은 THREAD\_INIT에서 NULL로 initialize된다.

기존 exit에서, file descriptor 기능이 추가됨에 따라 2번부터 127번까지, file descriptor마다 close를 호출하도록 한다. 또한, 죽는 thread의 child list를 순회하며 wait되지 않은 thread가 있다면 wait으로 처리하도록 한 후, 종료한다.

기존 read에서, buffer가 안전한지 확인하도록 한다. 이후 sync\_lock acquire를 호출하여 lock을 설정하고 error return value ret = -1을 선언한다. 이후. STDIN이라면 input\_getc를 이용하여 처리하고, fd>1인 경우라면 현재 thread의 fd가 비정상이면 exit, 정상이라면 file\_read를 호출하고 그 결과값을 return한다. 이때, return 전에 lock\_release를 호출한다.

Write 역시 file\_write를 호출하는 점을 제외하면 read와 동일하다.

Open을 호출하면, file이 정상인지 확인하고 비정상이라면 exit(-1)로 종료한다. 이후 lock을 획득하고 return value를 설정하고 filesys\_open을 호출하여 새 file을 생성한다. 만약 new\_file이 NULL이라면 -1로 반환 값을 설정한다. 아니라면 2번부터 127번까지 순회하며 빈 자리가 있는지 확인하고, 빈 자리가 있다면 new\_file을 저장한다. 이 때, 이미 열린 파일이라면 deny\_write를 통해 rox를 처리한다.

기타 system call 함수는 predefined 함수를 사용하기에, 다음 단락에서 설명한다.

Synchronization 관점에서, filesystem 접근을 위한 lock은 system call에서 앞서 설명했다. parent process가 먼저 죽어 rox multi-child와 syn-read/write가 의도한대로 작동하지 않기에, child가 load되는 동안 parent를 살리기 위한 sema\_load를 thread 구조체에 새로 심는다. 이후, 자식 process를 생성한 부모 process를 잠그고, 자식 process의 정상 처리를 완료한 후 parent의 semaphore를 해제해주도록 한다. 이때, parent와의 관계 유지를 위해 thread 구조체에 parent 포인터를 심고, 마찬가지로 initialized에서 자식이 부모를 가리키도록 running\_thread를 호출하여 설정한다.

* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명

Lock\_init, lock\_acquire, lock\_release:

semaphore의 일종인 lock을 관리하는 함수다.

Lock\_init은 lock의 초기값을 설정한다.

Lock\_acquire는 lock을 획득하여 다른 process가 진입하지 않도록 한다.

Lock\_release는 lock을 해제하여 다른 process가 진입할 수 있도록 한다.

sema\_up, sema\_down:

sempahore를 관리하는 함수다. 이전 프로젝트에서 사용한 것과 동일하다. Lock과 마찬깆로, semaphore가 설정되면, 다른 process나 handler에 의해 해제되기 전까지 해당 process의 진행이 정지된다. 이를 통해 critical section에 진입하는 process의 수를 조절할 수 있다.

filesys\_open:

name을 전달받아 파일을 생성한다. 이 때, directory를 탐색하고 inode를 전달한다. 탐색에 실패하면 inode에 NULL이 저장된다. File\_open을 호출하고 그 결과를 반환한다.

File\_open:

file 구조체를 새로 생성하고, 메모리를 할당한다. 정상 생성되었고, filesys\_open도 정상적으로 진행되어 inode가 NULL이 아니라면, file 구조체에 값을 할당하고 이를 반환한다.

file\_read, file\_write:

file과 buffer, size를 전달받아 이를 읽거나 쓰고 그 크기를 반환한다. 이 때, file 구조체에 pos 변수를 읽거나 쓴 만큼 조절한다.

file\_deny\_write:

이미 열린 file이라면 쓰기를 거부할 수 있도록 하는 함수다. 이는 file 구조체에 deny\_write를 조절하여 구현한다.

filesys\_create:

file name과 initial size를 전달받아 이를 생성한다.

filesys\_remove:

file name을 전달받아 이를 삭제한다.

file\_close:

file을 닫는다. 이 때, file의 쓰기 가능여부를 다시 복구하고자 file\_allow\_write를 호출하고 inode를 닫고 메모리를 해제한다.

file\_lenght:

file 길이를 반환한다.

file\_seek:

file cursor position을 전달받은 new\_pos로 설정한다.

file\_Tell:

file cursor의 현재 position을 반환한다.

* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1. read, write, open 구조 개선.

Read, write, open의 경우 if, else 분기를 통해 return 값을 처리하고 각기 다른 상황의 return 값을 반환해야 하는데, return/exit point가 여러 condition에서 존재하여 각 상황마다 lock을 조절하기 복잡했다. 이를 바꾸기 위해 각 condition마다 return/exit을 수행하는 대신 공통 return value ret에 저장하도록 하고, return은 ㅎ함수의 말미에서 수행하도록 하여 lock을 두 곳만 삽입할 수 있게 하였다.

2. synchronize

Syn-read/write 구현의 초반부에, debug를 거친 결과를 살펴보니 synchronization 혼선으로 인한 wait값의 순서 뒤바뀜이 아닌, file open이 실패하고 있었다. 큰 시간을 들여 synchronization을 개선해보았으나, 변화가 없었다. 여타 테스트와 본 테스트를 살펴보니, child 생성에서만 오류가 발생하는 것을 발견하고, file name도 정상적으로 작동한 것을 보아 parent가 child open보다 먼저 죽는 것이 문제임을 깨닫고 semaphore를 삽입하여 해결했다.

3. bad test

Bad test에서 page fault 오류 메시지를 출력하고 종료되어 테스트를 통과하지 못하고 있었다. Result file을 뜯어보니, exit(-1)을 출력하고 종료하면 해결되는 것을 알아내고 이를 해결하기 위해 오랜 시간 고민하였다. 그러나, 생각하여 보니 page fault를 발생하는 상황이 본 테스트가 의도한 자연스러운 진행 상황이므로 page fault 진입을 방지할 것이 아니라 error message 출력 이전에 exit(-1)을 호출하면 해결되는 것을 깨닫고, 해결했다.

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

