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

담당 교수 / 분반 : 박성용 교수님/운영체제 3반

이름 / 학번 : 김동빈/20161211

개발 기간 : 2021.11.05 ~ 2021.11.14

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

Project 1에서 read와 write system call을 stdin, stdout에 대해서만 구현을 했는데, 이를 file input, output 전체에 작동할 수 있도록 수정해주고 이외에도 create, remove, open, filesize, seek, tell, close system call을 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

File descriptor란 일반적으로 특정 파일에 접근하기 위해 사용되는 추상적인 key를 의미하는데 먼저 open system call의 경우 어떤 file을 open하고 그 file descriptor가 return value가 되며 filesize, read, write, seek, tell, close의 경우 file descriptor를 parameter 전달하기 때문에 file descriptor를 구현해 주어야한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

Project 1에서 stdin에 대해서만 구현했던 read system call과 stdout에 대해서만 구현했던 write system call의 경우에는 stdin, stdout이 아닌 다른 file에 대해서도 읽고 쓰는 것이 가능하도록 수정을 해주어야 하고 이외에도 create, remove, open, filesize, seek, tell, close system call을 추가로 구현해 주어야 한다.

3. Synchronization in Filesystem

File을 읽거나 쓰는 과정에서 다른 thread가 어떤 file을 읽고 있는 중에 다른 thread가 그 file에 write를 하려고 하거나 같은 file에 서로 다른 thread가 write하려는 상황을 방지해야 하므로 이를 막아 주기 위해 synchronization이 필요하다. 뿐만 아니라 현재 실행중인 파일에는 다른 thread가 write를 진행할 수 없도록 file.h의 file\_dent\_write 함수를 이용한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

앞서 project 1에서 진행했듯이 file descriptor가 0이면 stdin을 의미하고 1이면 stdout을 의미하므로 우리가 실제 사용할 수 있는 file descriptor는 2부터이다. 그리고 pintos manual에 따르면 각각의 thread는 각각의 file에 대해 file descriptor를 가지고 최대 128개까지 file을 open할 수 있다고 되어있으므로 이를 참고해서 각 thread에 크기 130인 file 배열을 선언해주었다. 또한 각 file descriptor를 thread내에서 현재 어떤 다른 file이 사용하고 있는지 여부를 의미하는 bool type의 fd\_used 배열도 생성해서 fd의 사용여부를 판단해주었다.

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(): filesys\_create 함수를 호출하여 file이라는 이름을 가진 initial\_size크기의 새로운 file을 생성한다.

remove(): filesys\_remove 함수를 이용해 file이라는 이름을 가진 file을 삭제한다.

open(): filesys\_open 함수를 호출하여 file을 open하고 현재 thread에서 사용 가능한 file descriptor를 찾아 그 file descriptor를 반환한다. 사용할 수 있는 file descriptor가 없는 경우 -1을 반환한다.

filesize(): 인자로 받은 fd를 file descriptor로 하는 file을 찾아 file\_length 함수를 호출해 그 크기를 반환한다.

read(): 인자로 받은 fd를 file descriptor로 하는 file을 찾아 file\_read 함수를 호출해 size만큼 buffer에 read하고 읽은 크기를 반환한다.

write(): 인자로 받은 fd를 file descriptor로 하는 file을 찾아 file\_write 함수를 호출해 size만큼 buffer의 내용을 write하고 작성한 크기를 반환한다.

seek(): 인자로 받은 fd를 file descriptor로 하는 file을 찾고 file\_seek 함수를 호출해 file 내의 포인터를 position의 위치로 이동시킨다.

tell(): 인자로 받은 fd를 file descriptor로 하는 file을 찾고 file\_tell 함수를 호출해 file 내의 현재 포인터의 위치를 반환한다.

close(): 인자로 받은 fd를 file descriptor로 하는 file을 찾고 file\_close 함수를 호출해 해당 file을 닫는다.

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

위의 함수 중 synchronization이 필요한 함수들을 생각하면 open, read, write가 있는데 각각이 동작하는 동안 write가 일어나면 안 되기 때문에 각 함수 안에서 filesys\_open 함수를 호출하기 전에 semaphore나 lock을 걸어주고 각각의 함수에서 open, read, write의 동작이 끝나면 다시 semaphore나 lock을 풀어주어 다시 해당 file에 접근할 수 있도록 해주는 것이다. 본 프로젝트에서는 모두 semaphore를 이용하여 구현해주었는데, lock을 이용해도 될 것이다.

또한 현재 실행중인 파일은 그 파일에 대해서 write가 진행되면 안 된다. 따라서 현재 실행 파일이 무엇인지 알아야 하고 write가 진행될 때 현재 실행파일일 경우 file\_deny\_write 함수를 이용해 실행 중인 파일에 write되는 것을 막아줄 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2021.11.05 ~ 2021.11.06: system call 기초 구현

2021.11.07 ~ 2021.11.09: open system call 구현, exit, exec system call 수정

2021.11.10 ~ 2021.11.11: read, write system call 수정 및 synchronization 작업 수행

2021.11.12 ~ 2021.11.14: test 후 오류 수정 및 보고서 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

1. file descriptor

File descriptor를 구현하기 위해 가장 먼저 해주어야 할 것은 현재 thread가 open하는 file들을 저장하는 새로운 struct file\* type의 배열 files[130]을 struct thread안에 추가해 주어야한다. 또한 2 ~ 129의 사용가능한 file descriptor중에서 사용되고 있는 file descriptor가 무엇인지를 나타내는 bool type 배열 fd\_used[130]의 자료구조가 필요하다. 또한 files[i]의 값의 update와 fd\_used[i]값의 update는 동시에 이루어져야 한다고 생각해서 files[i]에 새로운 값이 할당되거나 제거되는 경우 동시에 fd\_used[i]가 update되어야하고 이를 위해서는 synchronization작업이 추가로 필요하다고 판단해서 fd\_wait라는 semaphore를 추가로 구현한다. 따라서 thread.c에서 thread\_create과 thread\_init함수에서 sema\_init을 이용해 fd\_wait을 초기화 해주어야 한다.

2. system call

새로 구현해 주어야할 system call인 create, remove, open, filesize, seek, tell, close는 추가적으로 구현해야하고 각 함수안에서 호출되는 filesys\_create, filesys\_remove, filesys\_open 함수는 filesys.h에 정의되어 있기 때문에 이를 이용하면 되고 나머지 file\_length, file\_seek, file\_tell, file\_close함수의 경우 file.h에 정의되어 있기 때문에 이를 이용해주면 된다.

그리고 read와 write의 경우 각각 fd가 0과 1인 경우에 대해서만 구현이 되어 있는데 fd가 2이상 130미만인 경우에 새롭게 file system을 이용해서 새로운 구현을 해주면 된다.

마지막으로 exit system call에도 수정이 필요한데, 어떤 thread가 종료될 때 open file이 존재하면 안 된다고 생각하여 thread\_exit를 호출하기 전에 fd\_used배열을 탐색하면서 true인 경우 파일을 닫고 fd\_used도 false로 만들어주도록 수정해야한다. 또힌 pintos manual에 따르면 exec는 load함수가 수행을 완료할 때까지 exit를 하면 안 된다고 했으므로 이를 위해 exec system call에도 수정이 필요할 것이다.

3. synchronization

Open, read, write에서의 synchronization은 syscall.c의 전역변수로 struct semaphore mutex를 선언하고 syscall\_init함수 안에서 sema\_init함수를 이용해 초기화 하고 각각의 system call 함수 안에서 open, read, write가 각각 일어나기 전에 sema\_down함수를 호출하고 수행이 완료되면 sema\_up함수를 호출한다.

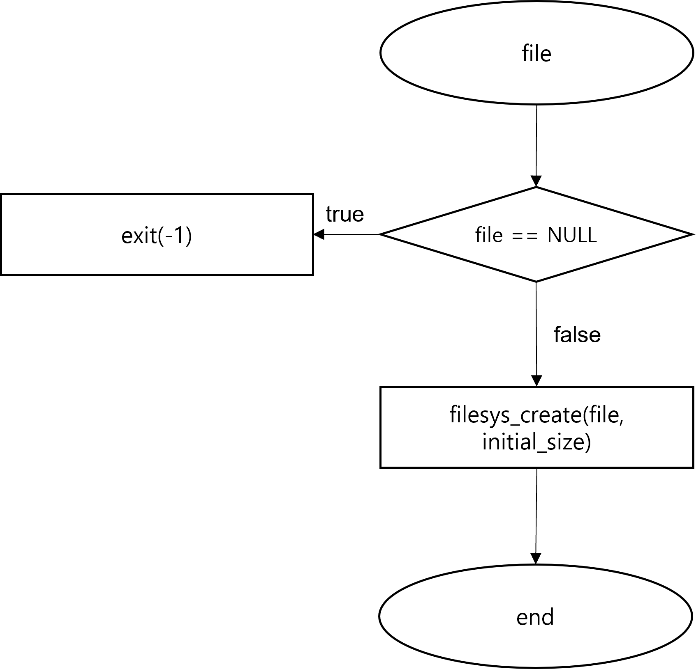
실행파일에 write를 못하게 막는 것은 write을 할 때 현재 write하고자하는 파일이 실행파일인지를 확인하고 그 경우에는 file\_deny\_write함수를 호출해 write가 일어나지 않도록 해줄 것이다.

마지막으로 Pintos manual에 따르면 exec system call이 process의 load가 종료되기 전에 먼저 종료되면 안 된다고 되어있다. 이는 만약 load의 실패하는 경우 이를 알리기 위해 -1을 반환하기 위함인데 이를 구현하기 위해 struct thread내부에 bool type의 load\_finish와 child\_load\_success 두 변수를 추가해준다. 이를 이용해서 process.c의 process\_execute가 종료되기 전에 busy waiting을 통해 load가 끝나기를 기다리는 것을 구현할 것이다.

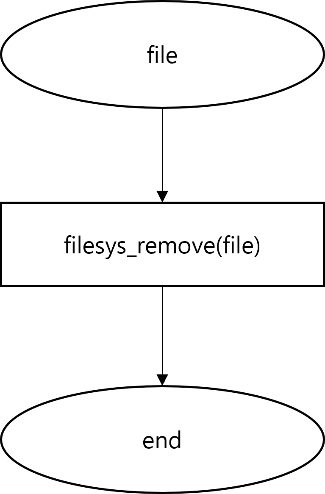
1. **연구 결과**
   1. **Flow Chart**

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

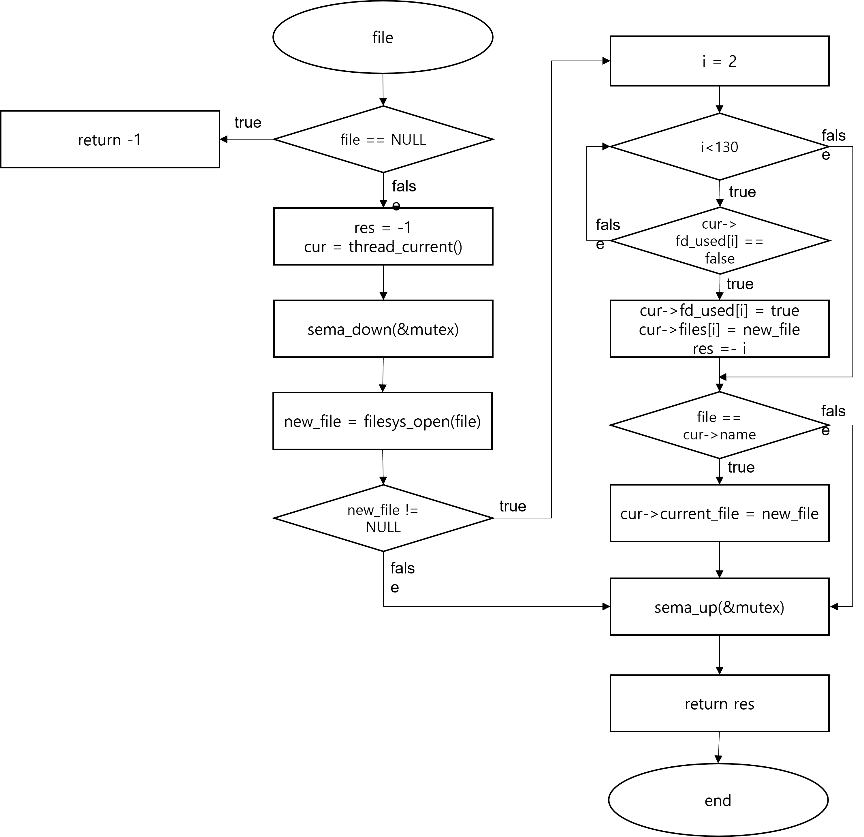
create의 flow chart는 아래와 같다.



remove의 flow chart는 아래와 같다.



open의 flow chart는 아래와 같다. mutex를 통해 다른 thread에서 write하는 것을 방지했고 fd\_wait을 통해 fd\_used와 files의 update를 동시에 진행했으며 filer과 cur->name을 비교해 실행파일임 경우 current\_file에 저장했다.

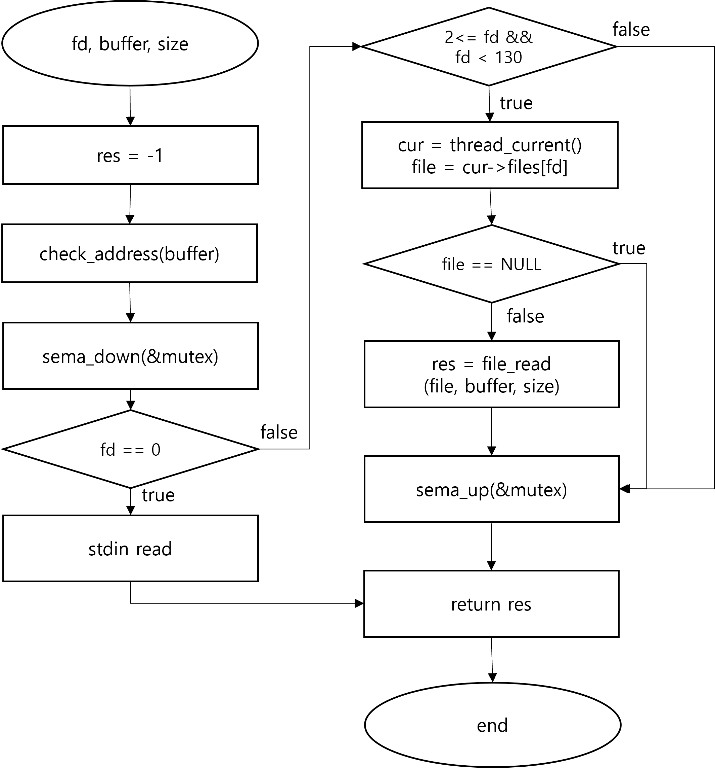


filesize의 flow chart는 아래와 같다.

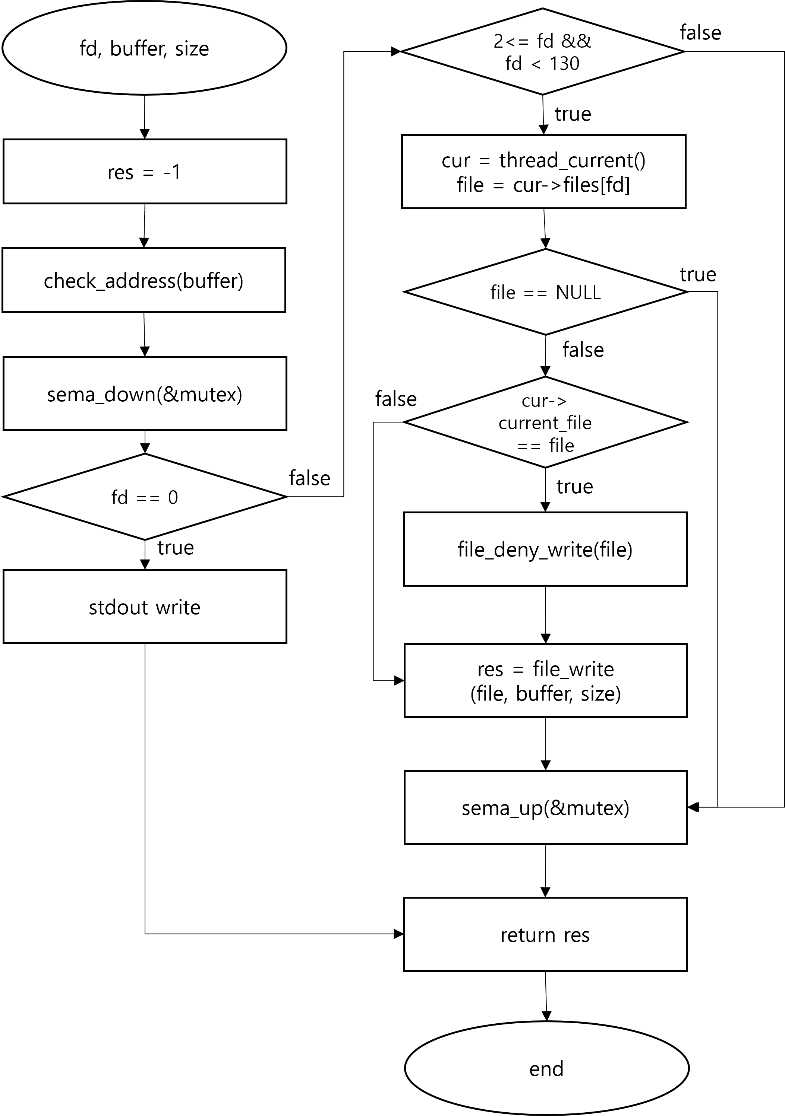
텍스트이(가) 표시된 사진

자동 생성된 설명

read의 flow chart는 아래와 같다. mutex를 통해 다른 thread에서 write하는 것을 방지했다.



write의 flow chart는 아래와 같다. mutex를 통해 다른 thread에서 write하는 것을 방지했다.



seek의 flow chart는 아래와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

tell의 flow chart는 아래와 같다.

텍스트이(가) 표시된 사진

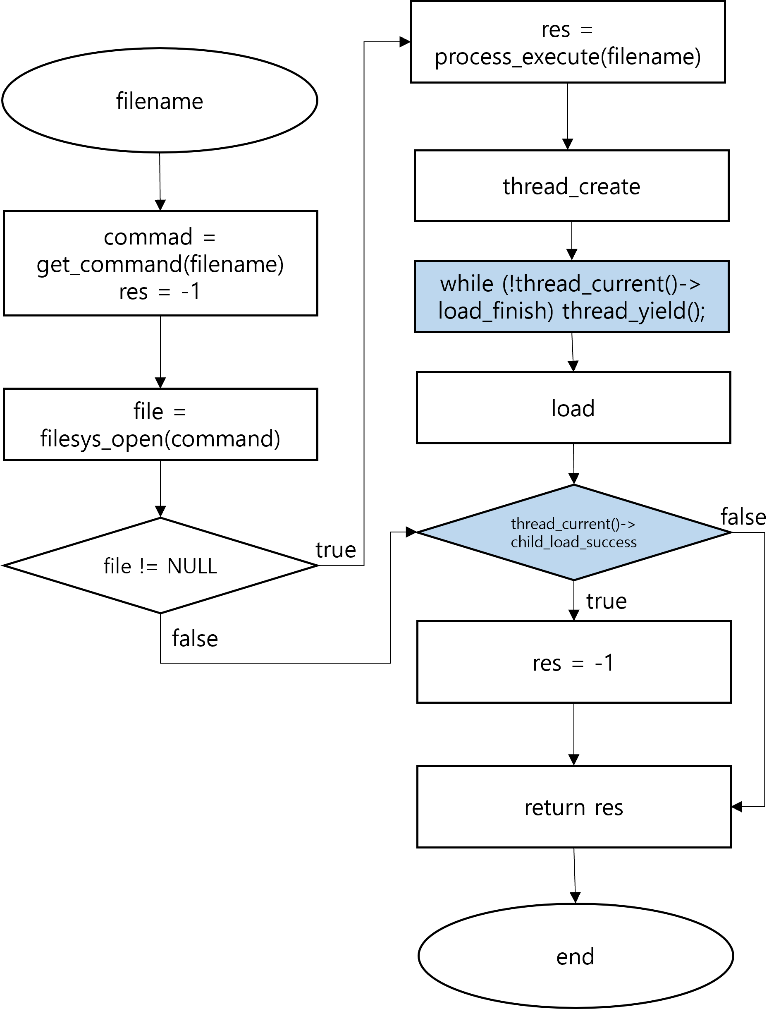
자동 생성된 설명

close의 flow chart는 아래와 같다. fd\_wait을 통해 fd\_used와 files의 update를 동시에 진행했다.

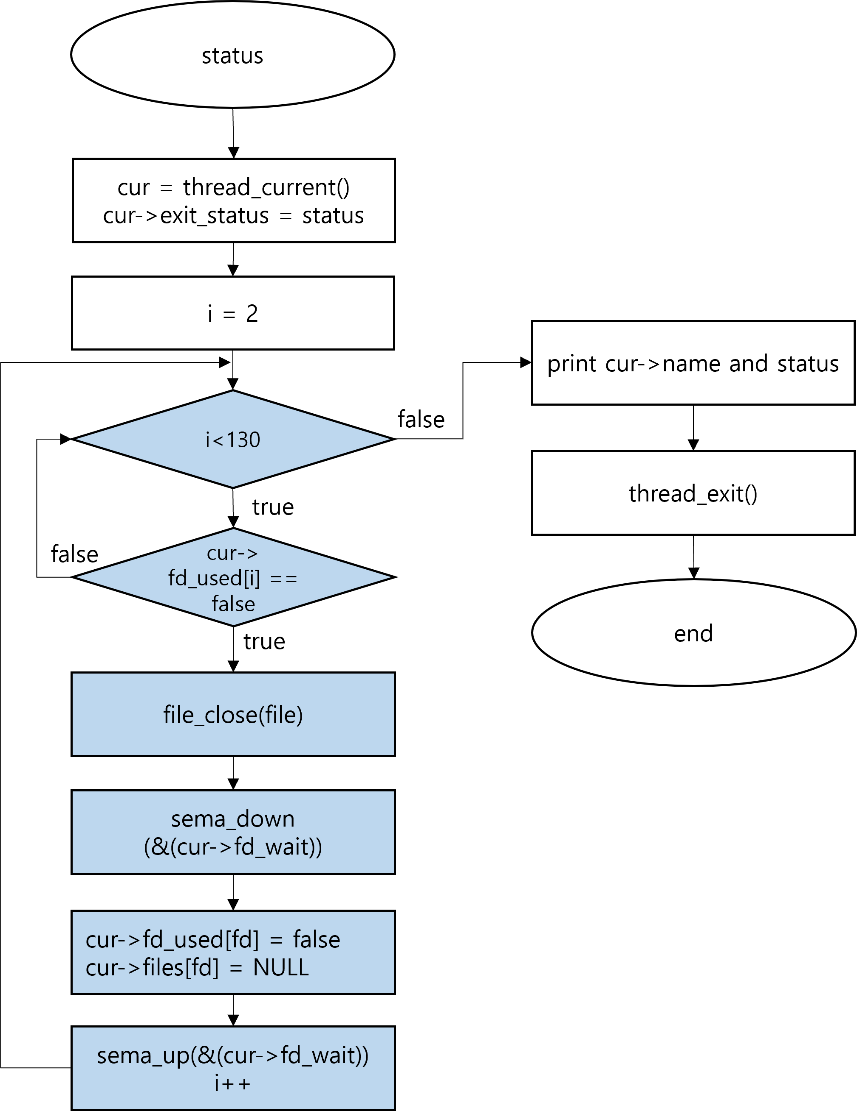
테이블이(가) 표시된 사진

자동 생성된 설명

exec의 flow chart는 아래와 같다. Project1에서 추가된 부분을 파란색으로 나타냈고 exec이 load가 끝나기를 기다리는 busy waiting과정과 load가 올바르게 완료되었는지 여부를 확인하는 부분에 해당한다.



exit의 flow chart는 아래와 같다. Project1에서 추가된 부분을 파란색으로 나타냈고 thread가 종료될 때 open 되어 있는 파일을 모두 닫는 과정에 해당한다.



* 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

1. file descriptor

Struct thread의 구조체 안에 struct file\* 타입 배열 files[130], bool 타입 배열 fd\_used[130], struct semaphore fd\_wait, bool 타입 변수 load\_finish와 child\_load\_success를 추가한다. 여기서 file descriptor를 위해 추가된 것은 files배열, fd\_used배열, semaphore fd\_wait이다.

먼저 open system call에서 새로운 파일을 open하면 이 new\_file에게 file descriptor를 지정해 주어야한다. 이를 위해 사용한 방법은 fd\_used를 sequential search를 이용해서 가장 작은 사용되지 않고 있는 file descriptor를 찾아준다. 만약 찾았다면 files[i]에 new\_file을 저장하고 fd\_used[i]는 true가 된다. 이 때 앞서 언급했듯이 files[i]와 fd\_used[i]의 update는 하나의 동작으로 이루어져야 하므로 두 instruction전에 sema\_down(&(cur->fd\_wait)), 후에 sema\_up(&(cur->fd\_wait))를 이용해 동기화 해준다.

마찬가지로 close 함수에서도 file을 close하면 더 이상 그 file descriptor를 사용하지 않으므로 다시 fd\_used[fd]는 false로 바꿔주고 files[i]는 NULL을 넣어준다. 위 과정에서도 마찬가지로 두 instruction전에 sema\_down(&(cur->fd\_wait)), 후에 sema\_up(&(cur->fd\_wait))를 이용해 동기화 해준다.

마지막으로 exit system call 내부에서 어떤 thread가 종료될 때 open되어 있는 file이 존재하면 안 된다고 생각해 thread\_exit함수를 호출하기 전에 fd\_used 배열을 sequential search를 이용해 fd\_used[i]가 true인 경우 file을 닫고 fd\_used[i]는 false로, files[i]는 NULL로 만들어 주어야 하는데 마찬가지로 두 instruction전에 sema\_down(&(cur->fd\_wait)), 후에 sema\_up(&(cur->fd\_wait))를 이용해 동기화 해준다.

2. system call

create, remove, filesize, seek, tell system call의 경우 복잡한 구현 없이 filesys.h 또는 file.h에 저장된 함수를 이용하면 되는데 하나씩 살펴보면 아래와 같다.

create(): 먼저 인자로 들어오는 파일의 이름을 의미하는 file이 NULL인 경우는 exit(-1)을 이용해 바로 종료해주고 그 외의 경우에는 filesys\_create() 함수를 호출해 해당 file이라는 이름을 가진 initial\_size 크기의 파일을 생성한다. filesys\_create() 함수는 파일 생성에 성공하면 true를 실패하면 false를 반환할 것인데 이를 그대로 반환한다.

remove(): filesys\_remove()함수를 호출해 file이라는 이름을 가진 파일에 성공하면 true를 반환, 실패하면 false를 반환한다.

filesize(): 인자로 전달된 fd를 이용하여 현재 thread에 files[fd]에 접근해 해당 파일을 찾은 다음 file\_length()함수를 호출해 해당 파일의 크기를 반환한다.

seek(): 인자로 전달된 fd를 이용하여 현재 thread에 files[fd]에 접근해 해당 파일을 찾은 다음 file\_seek()함수를 호출해 해당 파일의 포인터를 position의 위치로 바꾸어준다. 특별한 반환값은 없다.

tell(): 인자로 전달된 fd를 이용하여 현재 thread에 files[fd]에 접근해 해당 파일을 찾은 다음 file\_tell()함수를 호출해 해당 파일의 현재 포인터 위치를 반환한다.

open system call같은 경우는 먼저 파일 이름에 해당하는 file이 NULL인 경우는 먼저 -1을 반환하여 종료하고 아닌 경우에는 filesys\_open함수를 이용해 file이라는 이름을 가진 파일을 열어준다. 그리고 해당 파일의 file descriptor를 지정해주기 위해 현재 thread의 fd\_used 배열을 참조해 사용가능한 file descriptor를 찾아준다. 그리고 또 중요한 부분은 현재 실행 파일에는 write이 불가능하도록 하려면 현재 실행 파일임을 어떻게 지정해 주어야 하는가 하는 문제가 있었는데 thread 구조체 안에 current\_file이라고 현재 실행하는file을 저장할 변수를 추가했다. 처음에는 process.c의 load함수 내에서 file을 open하고 해당 파일은 current\_file에다 저장해주는 방식으로 이를 구현하여 했으나 작동하지 않았고 이는 load가 종료될 때 file\_close를 해서라고 생각했다. 이를 해결하기 위해 open system call에서 open하고자 하는 파일의 이름인 file과 현재 수행중인 thread의 이름이 같다면 현재 실행파일이라는 생각에 이름이 같은 경우 현재 thread의 current\_file변수에 위에서 open한 file을 저장해주었다.

close system call의 경우 인자로 전달된 fd로부터 현재 thread에서 file을 찾고 file\_close함수를 이용해 file을 닫고, fd\_used[fd]는 false로, files[fd]는 NULL로 업데이트 해준다.

read system call의 경우에는 먼저 buffer가 올바른 주소를 가리키는지 판단해 주어야한다. 그러지 못한 경우 read\_bad\_ptr을 통과하지 못한다. 다음 fd가 0인 경우는 project 1에서 구현한 대로 input\_getc()함수를 이용해 stdin으로부터 read를 하고, fd가 2이상 130미만인 경우에는 fd를 이용해서 현재 thread에서 files[fd]에 해당하는 파일을 찾아 파일이 NULL이 아닌 경우에만 file\_read() 함수를 이용해 file의 내용을 읽어준다.

write system call의 경우도 비슷하게 먼저 buffer가 올바른 주소를 가리키는지 판단해 주어야한다. 다음 fd가 1인 경우는 project 1에서 put\_buf()함수를 이용해 stdout에 write한 것을 그대로 사용해주었고, fd가 2이상 130미만인 경우에는 fd를 이용해서 현재 thread에서 files[fd]에 해당하는 파일을 찾아 파일이 NULL이 아닌 경우에 위의 open에서 현재 실행 파일은 thread의 current\_file이 저장하고 있으므로 현재 내가 write할 파일이 thread의 current\_file과 같으면 file\_deny\_write() 함수를 호출해 write가 일어나지 않도록 해주어야하고 file\_wirte() 함수를 호출해 file에 write를 진행한다.

다음으로 exit함수에서도 수정을 해줬는데 thread\_exit를 해주기 전에 fd\_used를 탐색하여 사용 중인 file이 있다면 모두 file\_close를 이용해 닫아주고, fd\_used와 files값도 조정해준다. 이때 write에서 thread의 file\_deny\_write() 함수를 통해 current\_file에 해당하는 파일에 write가 안 되도록 해주어서 이를 exit에서 file\_allow\_write() 함수를 호출해 write가 가능하도록 해주어야 한다고 생각했는데 제대로 작동하지 않았고 그 이유는 file\_close함수 안에서 file\_allow\_write()함수를 호출해주기 때문이라고 생각하여 굳이 따로 호출을 하지 않았다.

3. synchronization

먼저 open, read, write system call에서 각각의 open, read, write가 수행되는 과정에는 다른 thread의 방해가 있어서는 안 된다고 판단하여 syscall.c의 전역변수로 semaphore mutex를 하나 선언하고 이를 syscall\_init에서 초기화 해주었다. 그리고 각 system call에서 filesys\_open을 호출하기 전에 sema\_down(&mutex)를 통해 동기화를 진행해주었고 open에서는 file desciptor를 찾고 현재 실행 파일인지를 확인해서 current\_file에 저장한 이후에, read의 경우에는 stdin 혹은 file로부터의 read를 완료한 이후에, write의 경우에는 stdout 혹은 file로의 write를 완료한 이후에 sema\_up(&mutex)를 수행하여 각각의 수행이 다른 thread로부터 방해받지 않도록 구현해주었다.

그리고 exec 함수가 load의 수행이 완료될 때까지 기다려 주어야하는 부분은 thread 구조체에 bool 타입 변수 load\_finish와 child\_load\_success를 추가해줌으로써 구현해주었는데 load\_finish는 child thread의 load과정이 성공했든 실패했든 끝났는지 여부를 의미하고 process.c의 process\_execute() 함수 내에서 thread\_create()함수를 호출한 이후에 while문과 thread\_yield를 이용하여 busy waiting을 이용하여 load가 끝날 때까지 기다려주는 것을 구현했고 이는 start\_process()함수에서 load의 수행이 종료된 시점 이후에 child thread의 parent의 load\_finish를 true로 바꿔줌으로써 load가 완료되었음을 알린다. child\_load\_success는 결과적으로 load가 성공했는지 여부를 나타내고 성공했다면 원래 exec대로 child thread의 tid를 반환하고 실패한 경우에는 -1을 반환해야 하므로 procesp.c의 load함수 제일 마지막에 success를 반환하기 전에 success가 false인 경우 현재 thread의 parent의 child\_load\_success를 false로 만들어 준다. 그리고 exec system call에서 process\_execute가 수행이 완료된 이후에 child\_load\_success를 확인해서 false인 경우에는 반환값을 1로 설정함으로써 이를 구현해준다.

위의 방식대로 구현을 했을 때 bad\_read, bad\_write test를 통과하지 못하는 경우가 발생했는데 오류 내용을 살펴보니 page fault가 일어났었다. 그래서 여기 저기 살펴보다가 project 1에서 execption.c에서 user, not\_present, write중 하나라도 false이면 exit(-1)을 호출해 종료하도록 해줬는데 다시 살펴보니 not\_present는 false일때가 아니라 true일 때 exit(-1)해줘야 하고 이를 통해 bad\_read는 해결했고 bad\_write는 is\_user\_vaddr함수를 호출해 fault\_addr user memory가 아닌 경우 exit(-1)를 호출하도록 해주었더니 해결되었다.

위의 구현대로 구현은 수행했으나 syn\_read, syn\_write multi\_oom test를 통과하지 못했는데 syn\_read test에서 process.c의 load함수에서 child\_syn\_read 파일을 open을 하지 못하는 문제가 발생했다. 해당 문제는 open system call을 호출하지는 않아서 filesys\_open함수를 뜯어보니 filesys/directory.c에 구현된 dir\_lookup함수가 디렉토리에 해당 파일이 있다면 true를 반환해야 하고 없는 경우에는 false를 반환해야 하는데 child\_syn\_read가 있음에도 불구하고 false를 반환하는 것을 알 수 있었고 이 문제에 대해 며칠을 뜯어보고 고쳐보고 했지만 결국 해결하지 못했다.

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

make check 수행 결과는 아래와 같다.

