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

담당 교수 / 분반 : 김영재 교수님 / 2반

이름 / 학번 : 정성엽 / 20161641

개발 기간 : 2020.11.5 ~ 2020.11.14

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트에서는 지난 프로젝트 Proj1에서 구현한 Pintos 환경에 추가적으로 파일 시스템을 구현하고 파일 시스템상에서 발생하는 Critical Section 문제를 Synchronization을 통해 해결하는 것, 그리고 File system에 해당하는 System Call (read, write, create, open, close, remove, filesize, seek, tell)을 구현하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

2. (이번 프로젝트에서 구현해야 하는) System Calls

3. Synchronization in Filesystem

1. File Descriptor

: File System을 구현하기 위해서는 사용하는 File에 대한 정보를 저장할 File Descriptor가 필수적이다. File Descriptor는 프로세스에서 특정 파일에 접근할 때 사용하는 추상적인 값인데, 각 프로세스는 독립된 File Descriptor를 가진다. 이때 0번은 STDIN, 1번은 STDOUT, 2번은 STDERR로 사전에 지정된 Descriptor이다. File Descriptor를 Process(Thread)가 저장할 수 있도록 구현한다. 다음과 같은 File Descriptor의 구현을 통해 우리가 이번 프로젝트에서 구현하고자 하는 File System의 기본이 되는 파일의 접근이 가능하다.

2. System Calls (이번 프로젝트에서 구현해야 하는 System Call)

: File System의 구현을 위해 프로세스에서 요청가능한 File관련 System Call의 구현이 필요하다. 구현해야 하는 System Call은 아래와 같다.

1) read

: File Descriptor를 확인하여 만약 STDIN에 해당한다면 Buffer에 저장된 Char을 읽고, 만약 2 초과라면, 해당 Descriptor에 해당하는 File의 내용을 읽는다. 이 함수의 구현을 통해 File의 내용을 읽을 수 있다.

2) write

: File Descriptor를 확인하여 만약 STDOUT에 해당한다면 Buffer에 저장된 내용을 출력하고, 만약 2 초과라면, 해당 Descriptor에 해당하는 File에 내용을 적는다. 이 함수의 구현을 통해 File에 내용을 저장할 수 있다.

3) create

: File System의 create를 구현한다. initial\_size에 해당하는 공간을 할당하여 file의 name에 해당하는 Filed을 생성한다. 만약 생성에 실패하거나 이전에 동일한 이름이 존재한다면 False를 return 한다. 함수의 구현을 통해 File name에 해당하는 File을 생성할 수 있다.

4) open

: File name에 해당하는 File을 open하고 File Descriptor를 저장한다. 이 함수의 구현을 통해 File Descriptor를 저장하고, 이후에도 open된 File에 File Descriptor를 통해 지속적으로 접근 가능하다.

5) close

: File Descriptor에 해당하는 File을 Close한다. 이 함수의 구현을 통해 더 이상 접근하지 않을 File을 Close 할 수 있다.

6) remove

: File System의 remove를 구현한다. file name에 해당하는 File을 삭제한다. 만약 동작에 실패하거나 해당하는 File이 없다면 False를 Return 한다. 함수의 구현을 통해 File name에 해당하는 File을 삭제할 수 있다.

7) filesize

: 현재 File의 크기를 Byte 단위로 return 한다. 이 함수를 통해 프로세스는 원하는 File의 크기를 확인 가능하다.

8) seek

: 현재 File의 위치를 새로운 NEW\_POS에 해당하는 위치로 이동시킨다. 이 함수의 구현을 통해 파일에 새로운 작성 위치를 설정 가능하다.

9) tell

: 현재 File의 작성 위치를 return 한다. 이 함수의 구현을 통해 현재 File의 작성 위치를 확인 가능하다.

3. Synchronization in Filesystem

: File System 상에서 각 프로세스(Thread)는 동일한 파일에 접근할 수 있다. 이때 동시에 한 파일에 Read와 Write가 동작한다면, 정상적으로 동작하지 않기 때문에 이를 방지하기 위해 Lock이나 Synchronization을 사용해 Synchronization을 해주어야 한다. Synchronization을 성공적으로 구현한다면 File system에서 발생가능한 Critical Section 문제를 방지할 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

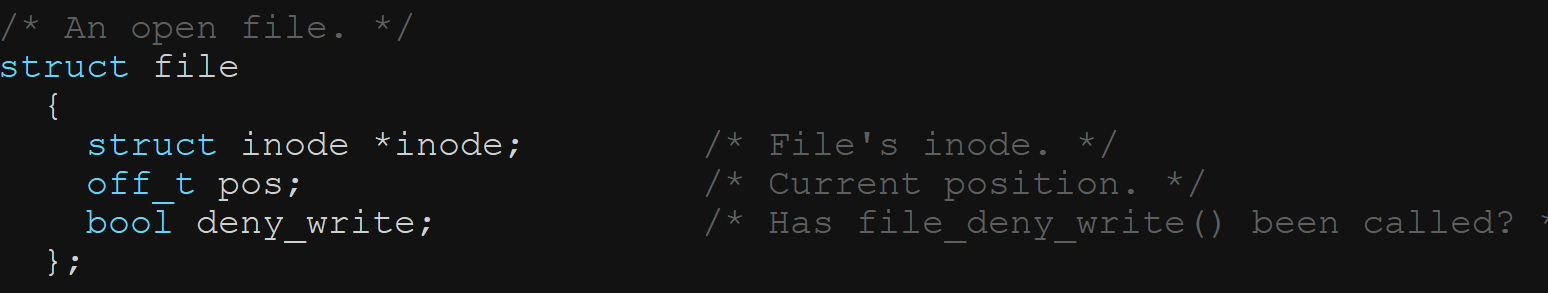
1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

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)

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

1. File Descriptor

: File System을 구현하기 위해서는 사용하는 File에 대한 정보를 저장할 File Descriptor가 필수적이다. File Descriptor는 프로세스에서 특정 파일에 접근할 때 사용하는 추상적인 값인데, 각 프로세스는 독립된 File Descriptor를 가진다. 이때 0번은 STDIN, 1번은 STDOUT, 2번은 STDERR로 사전에 지정된 Descriptor이다. File Descriptor를 Process(Thread)가 저장할 수 있도록 구현한다. File Descriptor의 저장을 위해 file\* (파일 포인터)를 원소로 갖는 array자료구조를 사용하였다. 이러한 자료구조의 선택 이유는 지속적으로 저장 및 사용해야하는 File Descriptor의 저장을 위해 안정적이고 안전한 자료구조를 사용해야 하고, 구현상의 편리함을 추가적으로 고려하였을 때 array가 가장 적합하다고 고려되었기 때문이다. file 구조체는 아래와 같은 구성을 갖는다.



이러한 file 구조체를 저장하기 위해 file 구조체를 원소로 갖는 array를 사용하는 것 보다, 필요할 때 마다 file 구조체를 할당 가능한 file pointer를 원소로 갖는 array를 선택하였다.

2. System Calls (이번 프로젝트에서 구현해야 하는 System Call)

: File System의 구현을 위해 프로세스에서 요청가능한 File관련 System Call의 구현이 필요하다. 구현해야 하는 System Call은 아래와 같다.

1) read

: File Descriptor를 확인하여 만약 STDIN에 해당한다면 Buffer에 저장된 Char을 읽고, 만약 2 초과라면, file\_read() 함수를 사용해 해당 Descriptor에 해당하는 File의 내용을 읽는다.

2) write

: File Descriptor를 확인하여 만약 STDOUT에 해당한다면 putbuf() 함수를 사용해 Buffer에 저장된 내용을 출력하고, 만약 2 초과라면, file\_write() 함수를 사용해 해당 Descriptor에 해당하는 File에 내용을 적는다.

3) create

: filesys\_create() 함수를 사용해, initial\_size에 해당하는 공간을 할당하여 file의 name에 해당하는 Filed을 생성한다. 만약 생성에 실패하거나 이전에 동일한 이름이 존재한다면 False를 return 한다.

4) open

: filesys\_open() 함수를 사용해, File name에 해당하는 File을 open하고 therad의 file descriptor를 저장할 수 있는 비어있는 array를 검사하여, File Descriptor를 저장한다.

5) close

: filesys\_close() 함수를 사용해, File Descriptor에 해당하는 File을 Close한다.

6) remove

: filesys\_remove() 함수를 사용해, file name에 해당하는 File을 삭제한다. 만약 동작에 실패하거나 해당하는 File이 없다면 False를 Return 한다.

7) filesize

: files\_length() 함수를 사용해, 현재 File의 크기를 Byte 단위로 return 한다.

8) seek

: file\_seek() 함수를 사용해, 현재 File의 편집 위치를 새로운 position에 해당하는 위치로 이동시킨다.

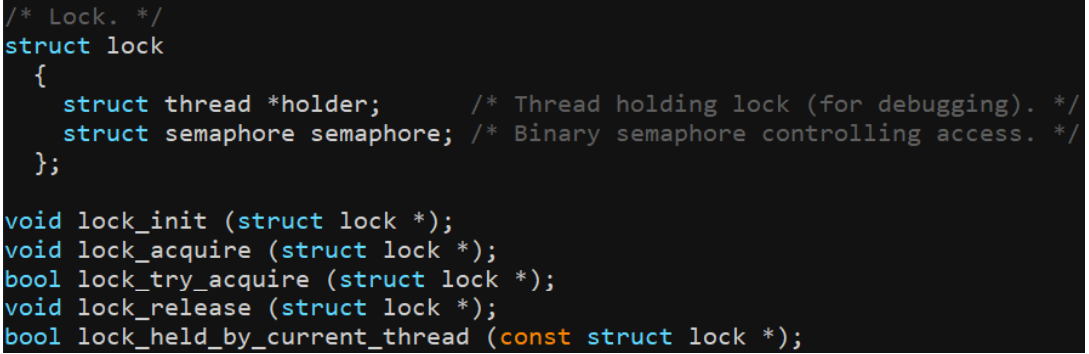
9) tell

: file\_tell() 함수를 사용해, 현재 File의 작성 위치를 return 한다.

3. Synchronization in Filesystem

: File System 상에서 각 프로세스(Thread)는 동일한 파일에 접근할 수 있다. 이때 동시에 한 파일에 Read와 Write가 동작한다면, 정상적으로 동작하지 않기 때문에 이를 방지하기 위해 Lock이나 Synchronization을 사용해 Synchronization을 해주어야 한다.

Lock을 사용하는 경우, 아래와 같은 함수를 사용하여 lock을 사용할 수 있다.



read와 write에서 발생하는 readers and writers problem을 해결하기위해 lock을 사용하는 경우 system call 상에서만 lock을 사용할 것 임으로 file system에 사용할 lock을 추가하고, syscall\_init 호출시 lock\_init 함수를 사용하여 lock을 초기화 한다. 이후 read와 write시 lock\_acquire 함수를 사용해 지금 critical section에 들어가 있는 다른 프로세스(thread)가 있는지 확인하고 없다면 critical section에 진입한다. 1개의 프로세스(thread)가 critical section을 사용 중이라면 다른 프로세스(thread)는 진입하지 못한다. critical section에서의 동작이 완료된 후, lock\_release 함수를 통해 다른 프로세스(thread)의 진입이 가능하게 만든다.

Semaphore을 사용하는 경우, 아래와 같은 함수를 사용하여 Semaphore를 사용할 수 있다.



Semaphore을 사용하는 경우 system call 상에서만 Semaphore을 사용할 것 임으로 file system에 사용할 Semaphore을 추가하고, syscall\_init 호출시 sema\_init 함수를 사용하여 lock을 초기화 한다. 이후 read와 write시 sema\_down 함수를 사용해 지금 critical section에 들어가 있는 다른 프로세스(thread)가 있는지 확인하고 없다면 critical section에 진입한다. 1개의 프로세스(thread)가 critical section을 사용 중이라면 다른 프로세스(thread)는 진입하지 못한다. critical section에서의 동작이 완료된 후, sema\_up 함수를 통해 다른 프로세스(thread)의 진입이 가능하게 만든다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

1) 2020/11/1 : Pintos 프로젝트 이해 및 코드 분석, 문재 해결방법 구상

2) 2020/11/4 : File Descriptor, File system System Call일부 구현

3) 2020/10/5 : File system System Call 구현 및 디버깅

4) 2020/10/7 : File read write Synchronization 구현 및 디버깅

5) 2020/10/8 : multi.oom 테스트케이스 해결 및 디버깅

6) 2020/10/13 : 디버깅 및 보고서 작성

7) 2020/10/14 : flowchart 작성 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 System을 구현하기 위해서 File Descriptor를 저장할 자료구조를 선언하여야 한다. src/thread/thread.h 코드에 선언된 thread 구조체에 struct file\* fd[128];와 같이 file pointer를 원소로 하는 배열을 추가한다.

이후 src/thread/thread.c 코드에 선언된 init\_thread() 함수에서 File Descriptor 배열의 초기화를 수행한다. 이후 각 system call에서 File Descriptor에 대한 처리가 필요한데, 이는 아래에 기술되었다.

2. System Calls (이번 프로젝트에서 구현해야 하는 System Call)

: src/userprog/syscall.c 코드에 추가 및 수정되는 System Call 함수는 아래와 같다.

기존 선언된 함수를 수정하는 경우만 (수정)을 표시하였고, 다른 함수들은 이번에 추가하는 기능을 위해 새롭게 추가한 함수들이다.

1) read (수정)

: File Descriptor를 확인하여 만약 STDIN에 해당한다면 Buffer에 저장된 Char을 읽고(Proj1 구현내용), 만약 2 초과라면, file\_read() 함수를 사용해 해당 Descriptor에 해당하는 File의 내용을 읽을 수 있도록 기능을 추가한다.

2) write (수정)

: File Descriptor를 확인하여 만약 STDOUT에 해당한다면 putbuf() 함수를 사용해 Buffer에 저장된 내용을 출력하고(Proj1 구현내용), 만약 2 초과라면, file\_write() 함수를 사용해 해당 Descriptor에 해당하는 File에 내용을 적도록 기능을 추가한다.

3) create

: filesys\_create() 함수를 사용해, initial\_size에 해당하는 공간을 할당하여 file의 name에 해당하는 Filed을 생성한다. 만약 생성에 실패하거나 이전에 동일한 이름이 존재한다면 False를 return 한다.

4) open

: filesys\_open() 함수를 사용해, File name에 해당하는 File을 open하고 therad의 file descriptor를 저장할 수 있는 비어있는 array를 검사하여, File Descriptor를 저장한다.

5) close

: filesys\_close() 함수를 사용해, File Descriptor에 해당하는 File을 Close한다.

6) remove

: filesys\_remove() 함수를 사용해, file name에 해당하는 File을 삭제한다. 만약 동작에 실패하거나 해당하는 File이 없다면 False를 Return 한다.

7) filesize

: files\_length() 함수를 사용해, 현재 File의 크기를 Byte 단위로 return 한다.

8) seek

: file\_seek() 함수를 사용해, 현재 File의 편집 위치를 새로운 position에 해당하는 위치로 이동시킨다.

9) tell

: file\_tell() 함수를 사용해, 현재 File의 작성 위치를 return 한다.

10) exit(수정)

: thread\_exit()을 호출하기 전 현재 thread의 File Descriptor들 중 닫히지 않은 것이 있다면 확인하고 닫아주는 기능을 추가한다.

위와 같은 각 System Call을 수행할 수 있게 하기 위해 syscall\_handler() 함수에 각 syscall 번호에 따라 parameter가 user\_vaddr인지 검사하고, 해당하는 함수를 호출하도록 추가한다.

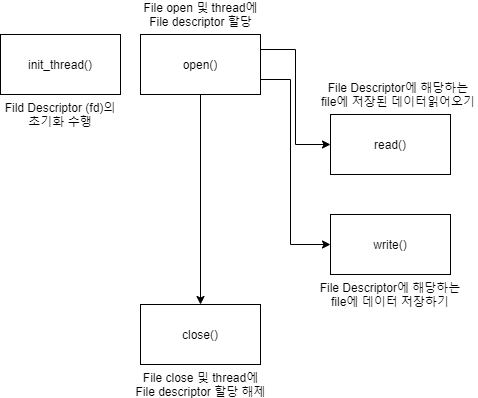
3. Synchronization in Filesystem

: 이번 프로젝트의 Filesystem Synchronization 구현을 위해 Lock와 Semaphore를 사용할 수 있는데, 명세서상의 추가적인 제약사항이 없음으로 구현이 비교적 더 쉬운 Lock을 사용해 구현하였다. src/userprog/syscall.c 코드에 전역변수로 lock 변수를 선언하고, syscall\_init 함수에서 lock을 초기화 한다. 그리고 read함수, open 함수, write 함수에서 함수의 Critical Section 진입시 lock\_acquire 함수를 통해 현재 critical section에 진입해 있는 프로세스(thread)가 있는지 검사하고 없다면 진입하도록 한다. critical section의 동작이 완료되면 lock\_release 함수를 사용해 다른 프로세스(thread)가 critical section에 진입할 수 있도록 한다.

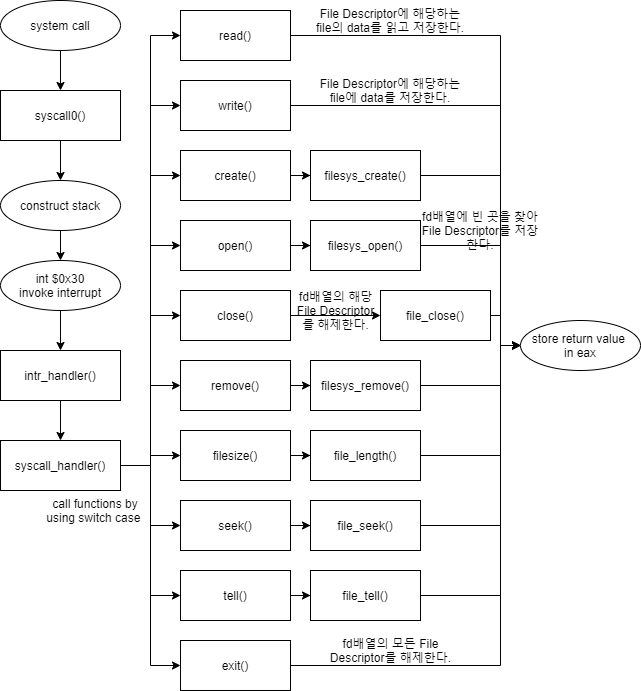
1. **연구 결과**
   1. **Flow Chart**

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

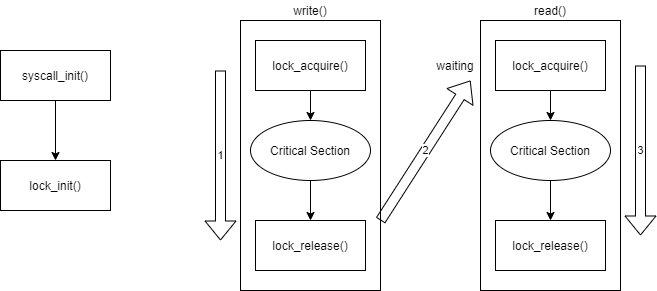
1. File Descriptor



2. System Calls (이번 프로젝트에서 구현해야 하는 System Call)



3. Synchronization in Filesystem

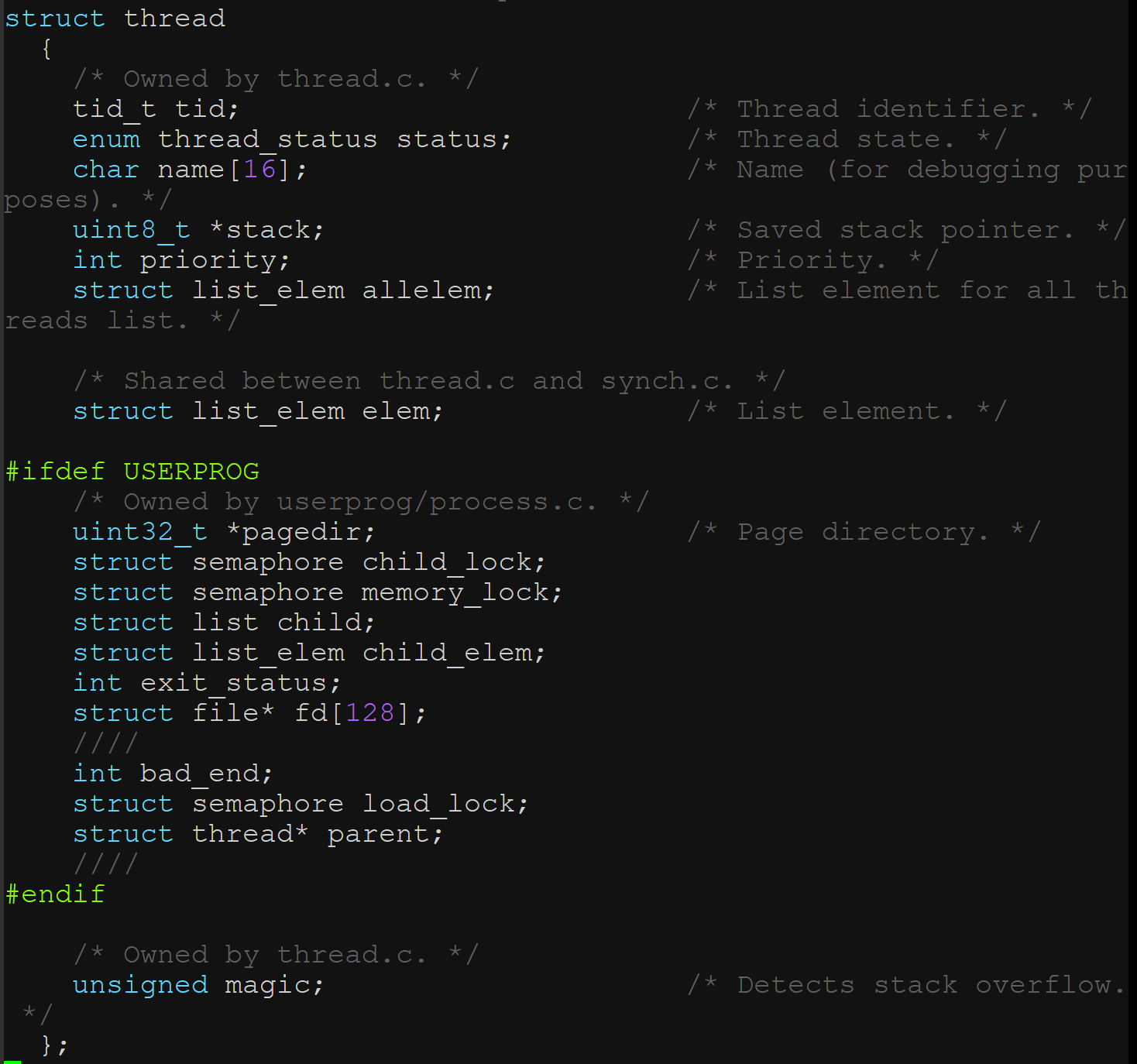
****

위 Flowchart에서는 writer와 reader만을 표현하였지만 구현상에서는 writer, writer간의 관계, reader, reader의 관계, reader, writer의 관계 모두 동일한 lock이 적용된다.

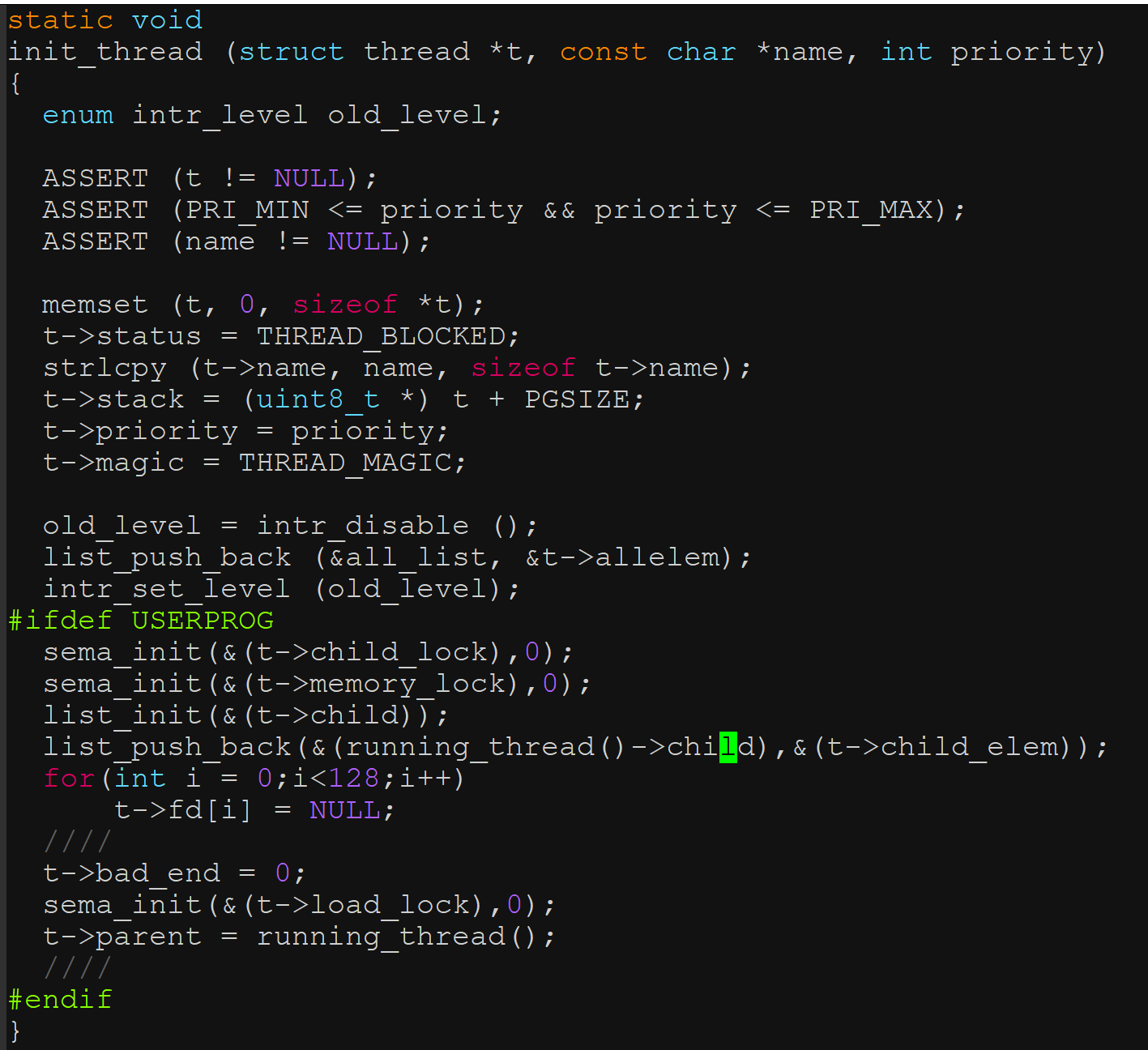
* 1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1. File Descriptor

: File System을 구현하기 위해서 File Descriptor를 저장할 자료구조를 선언하여야 한다. src/thread/thread.h 코드에 선언된 thread 구조체에 struct file\* fd[128];와 같이 file pointer를 원소로 하는 배열을 추가한다.



이후 src/thread/thread.c 코드에 선언된 init\_thread() 함수에서 File Descriptor 배열의 초기화를 수행한다.



이후 각 system call에서 File Descriptor에 대한 처리가 필요한데, 이는 아래에 기술되었다.

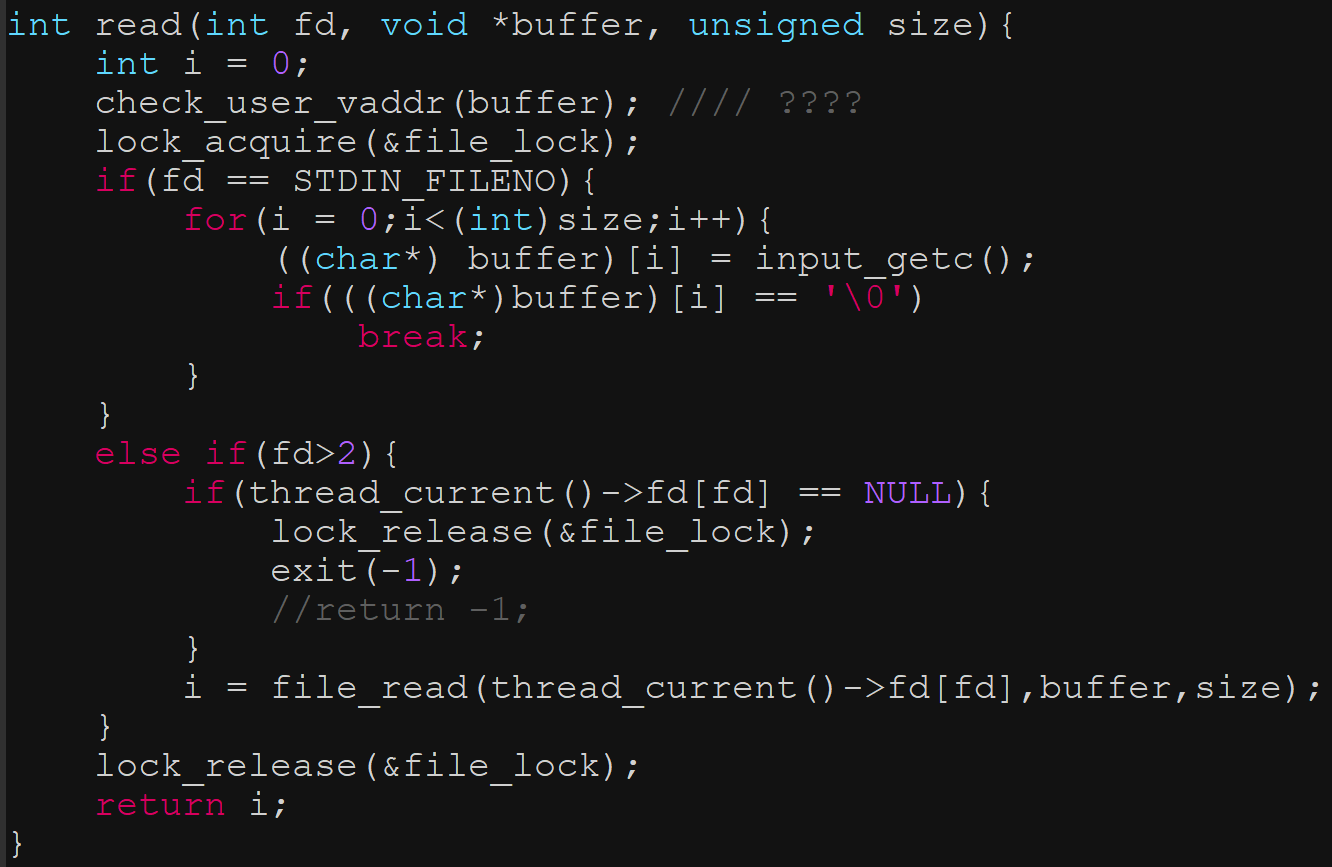
2. System Calls (이번 프로젝트에서 구현해야 하는 System Call)

: src/userprog/syscall.c 코드에 추가 및 수정되는 System Call 함수는 아래와 같다.

기존 선언된 함수를 수정하는 경우만 (수정)을 표시하였고, 다른 함수들은 이번에 추가하는 기능을 위해 새롭게 추가한 함수들이다.

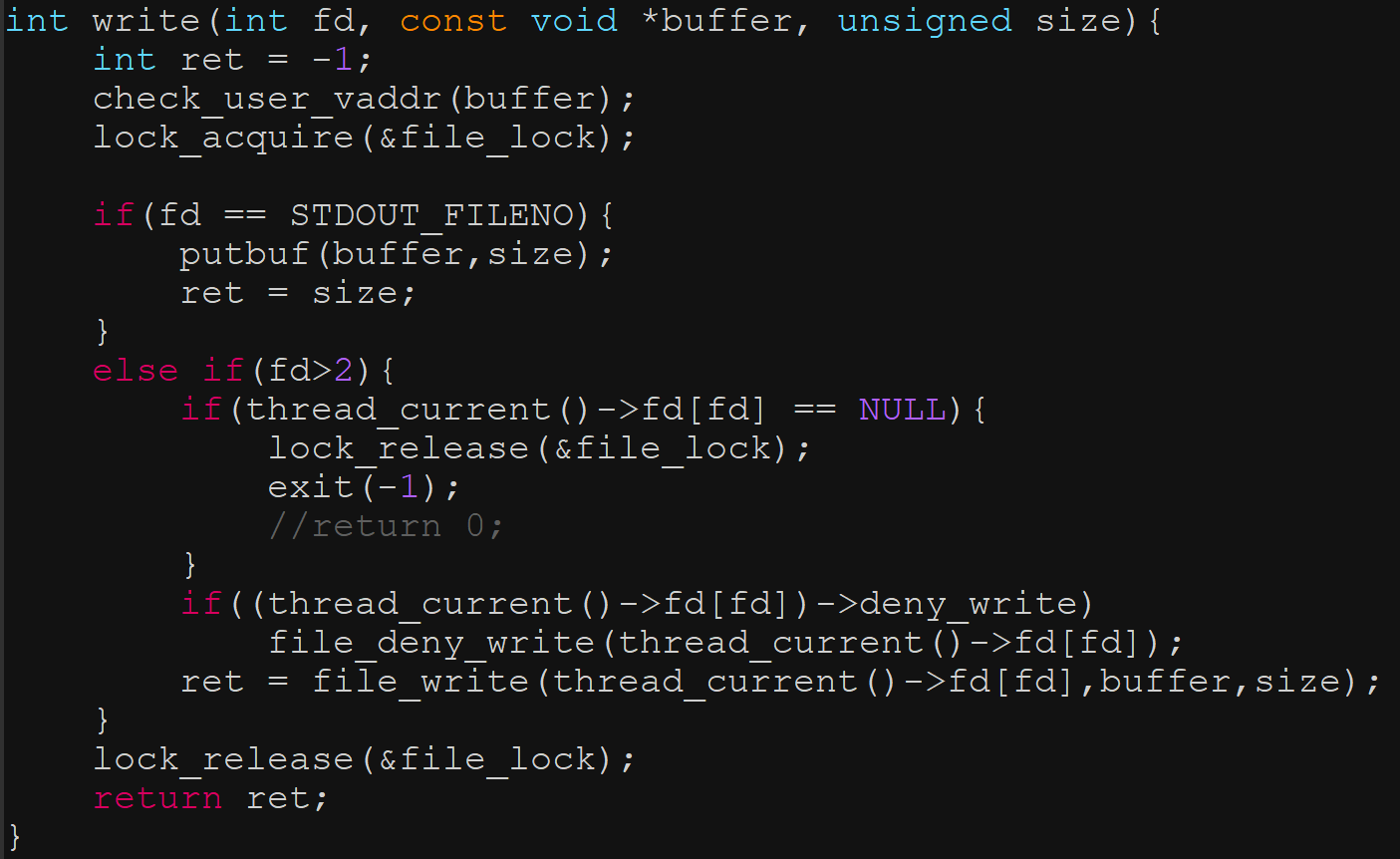
1) read (수정)

: File Descriptor를 확인하여 만약 STDIN에 해당한다면 Buffer에 저장된 Char을 읽고(Proj1 구현내용), 만약 2 초과라면, file\_read() 함수를 사용해 해당 Descriptor에 해당하는 File의 내용을 읽을 수 있도록 기능을 추가한다.



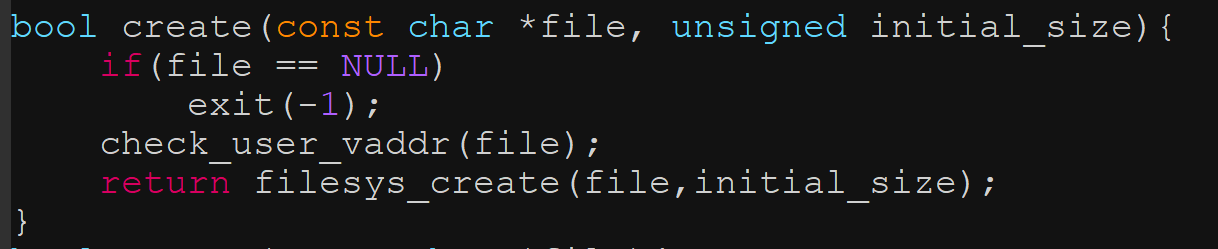
2) write (수정)

: File Descriptor를 확인하여 만약 STDOUT에 해당한다면 putbuf() 함수를 사용해 Buffer에 저장된 내용을 출력하고(Proj1 구현내용), 만약 2 초과라면, file\_write() 함수를 사용해 해당 Descriptor에 해당하는 File에 내용을 적도록 기능을 추가한다.



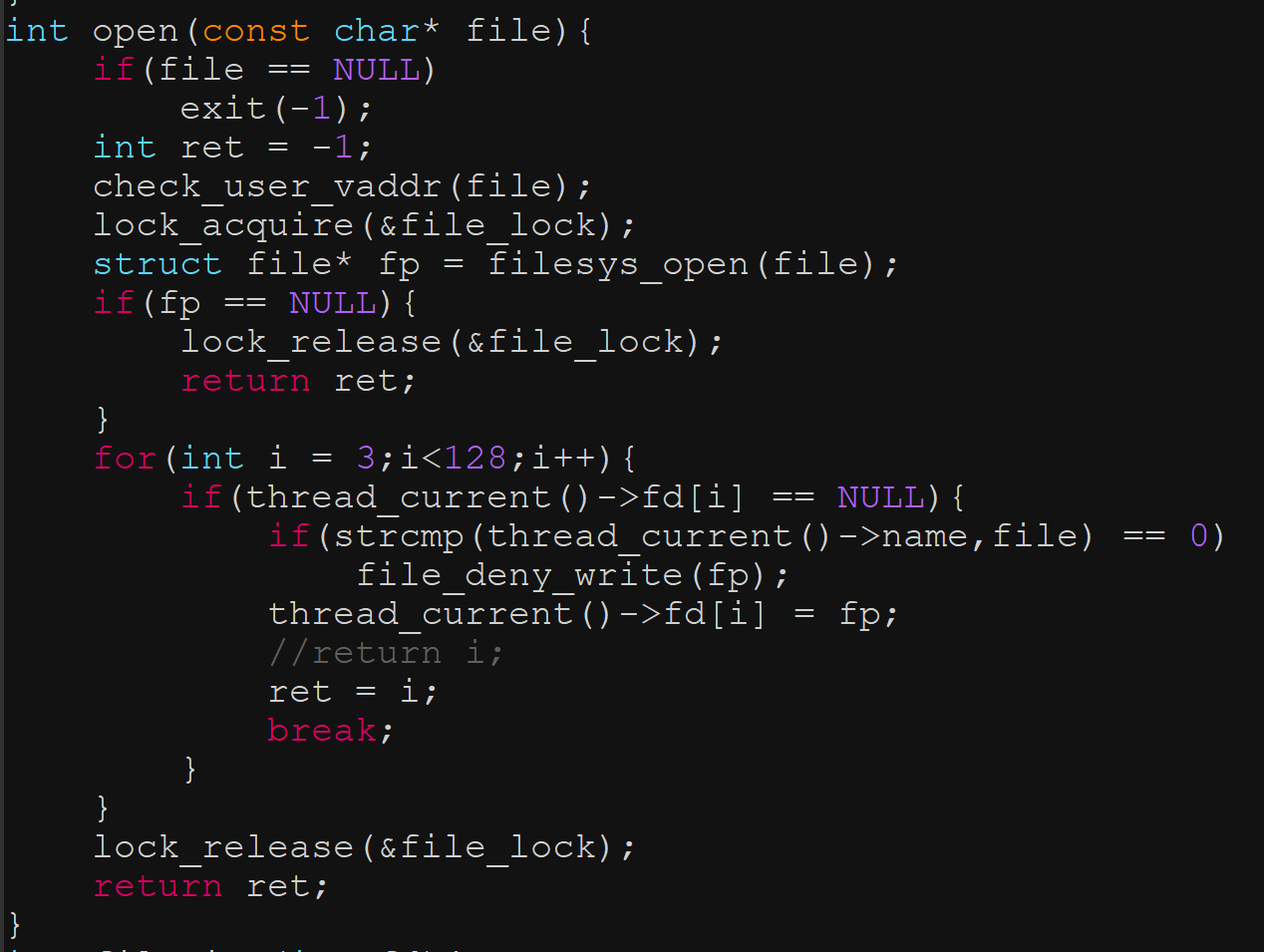
3) create

: filesys\_create() 함수를 사용해, initial\_size에 해당하는 공간을 할당하여 file의 name에 해당하는 Filed을 생성한다. 만약 생성에 실패하거나 이전에 동일한 이름이 존재한다면 False를 return 한다.



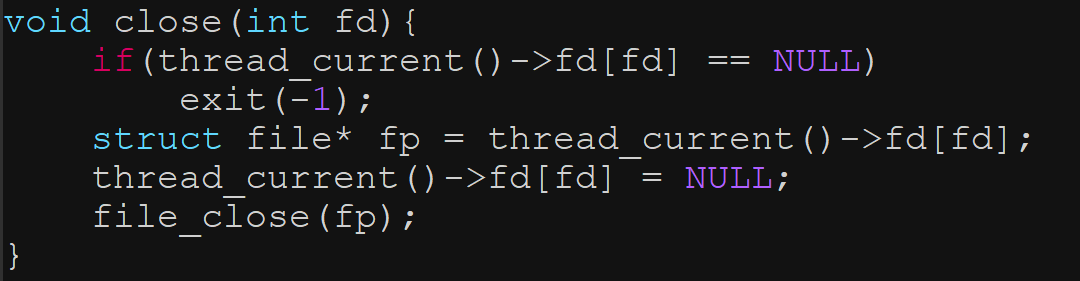
4) open

: filesys\_open() 함수를 사용해, File name에 해당하는 File을 open하고 therad의 file descriptor를 저장할 수 있는 비어있는 array를 검사하여, File Descriptor를 저장한다.



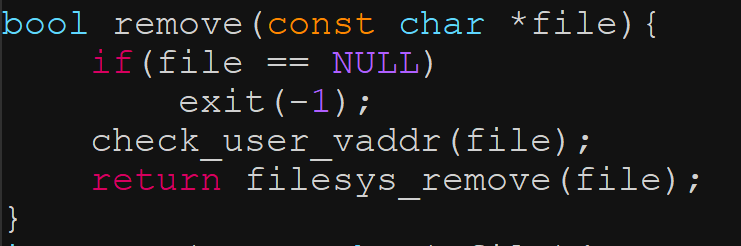
5) close

: filesys\_close() 함수를 사용해, File Descriptor에 해당하는 File을 Close한다.



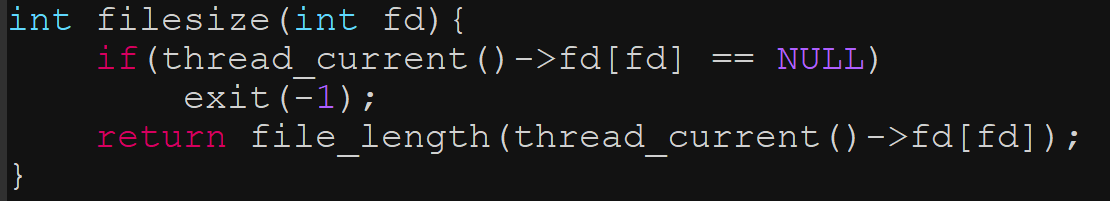
6) remove

: filesys\_remove() 함수를 사용해, file name에 해당하는 File을 삭제한다. 만약 동작에 실패하거나 해당하는 File이 없다면 False를 Return 한다.



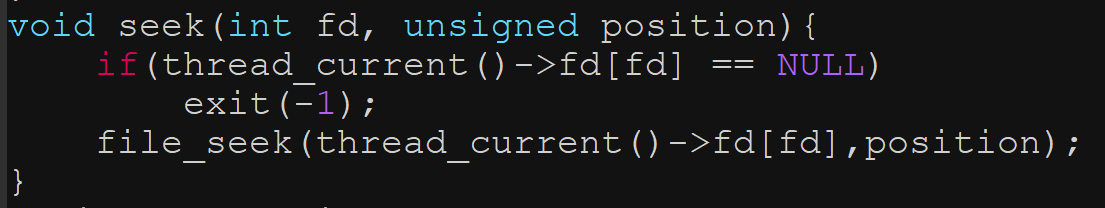
7) filesize

: files\_length() 함수를 사용해, 현재 File의 크기를 Byte 단위로 return 한다.

ㄴ

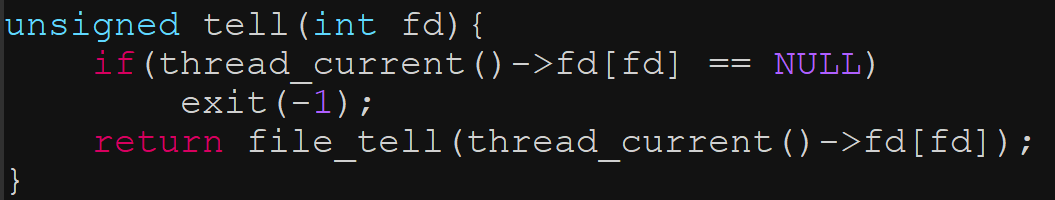
8) seek

: file\_seek() 함수를 사용해, 현재 File의 편집 위치를 새로운 position에 해당하는 위치로 이동시킨다.



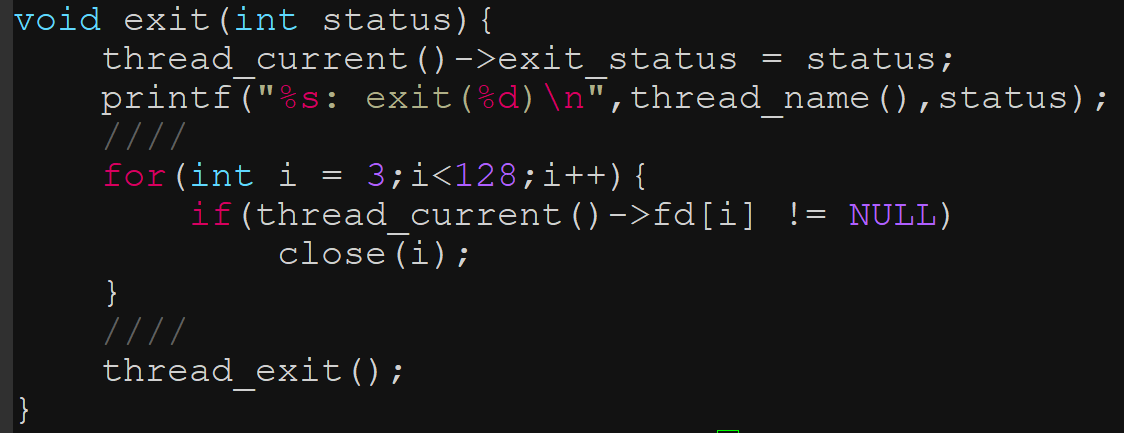
9) tell

: file\_tell() 함수를 사용해, 현재 File의 작성 위치를 return 한다.



10) exit(수정)

: thread\_exit()을 호출하기 전 현재 thread의 File Descriptor들 중 닫히지 않은 것이 있다면 확인하고 닫아주는 기능을 추가한다.



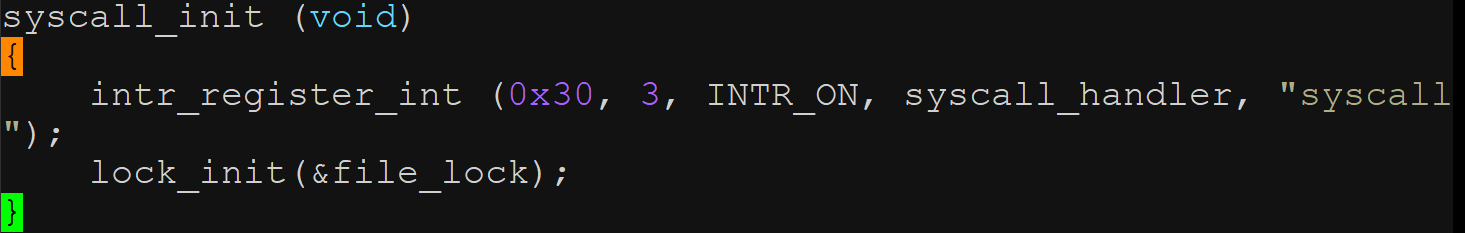
위와 같은 각 System Call을 수행할 수 있게 하기 위해 syscall\_handler() 함수에 각 syscall 번호에 따라 parameter가 user\_vaddr인지 검사하고, 해당하는 함수를 호출하도록 추가한다.

3. Synchronization in Filesystem

: 이번 프로젝트의 Filesystem Synchronization 구현을 위해 Lock와 Semaphore를 사용할 수 있는데, 명세서상의 추가적인 제약사항이 없음으로 구현이 비교적 더 쉬운 Lock을 사용해 구현하였다. src/userprog/syscall.c 코드에 전역변수로 lock 변수를 선언하고,



syscall\_init 함수에서 lock을 초기화 한다.



그리고 read함수, open 함수, write 함수에서 함수의 Critical Section 진입시 lock\_acquire 함수를 통해 현재 critical section에 진입해 있는 프로세스(thread)가 있는지 검사하고 없다면 진입하도록 한다.



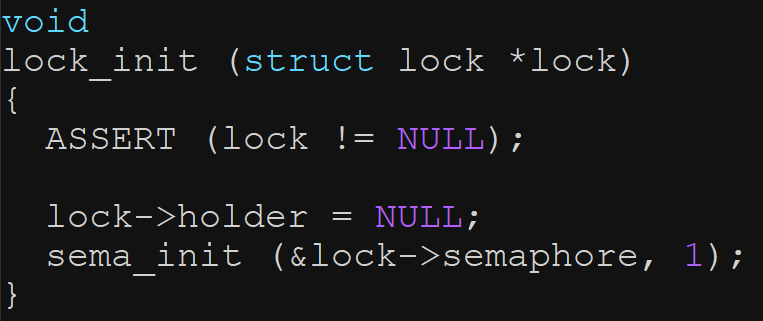
critical section의 동작이 완료되면 lock\_release 함수를 사용해 다른 프로세스(thread)가 critical section에 진입할 수 있도록 한다.



해당 lock의 구현을 위해 내장된 lock\_init(), lock\_acquire(), lock\_release() 함수를 사용한다. 아래에 사용한 각 함수의 설명을 기술하였다.

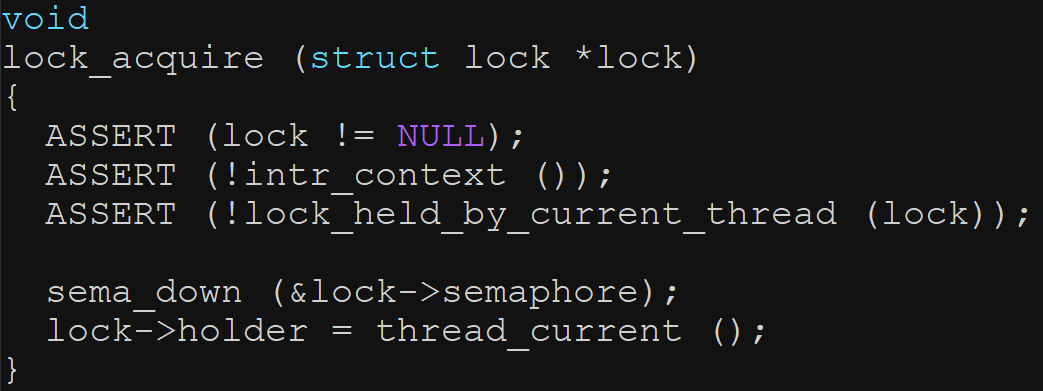
1) lock\_init

: lock이 NULL이 아닌지 검사하고, lock을 초기화 한다.

****

2) lock\_acquire

: lock이 NULL이 아닌지 검사하고, 현재 context를 검사한다. 현재 thread에 의해서 lock이 걸린 상태인지 검사하고, sema\_down 함수를 사용하여 waiting을 구현하였다.



3) lock\_release

: lock이 NULL이 아닌지 검사하고, 현재 thread에 의해서 lock이 걸린 상태인지 검사하고, sema\_up 함수를 사용하여 wake를 구현하였다.

위와 같은 lock 함수들을 사용하여 Synchronization 문제를 해결하였다.

추가적으로 Test Case 중 multi.oom을 해결하는데 어려움이 있었는데, multi.oom은 child thread를 여러 개 생성하는 과정에서 child 보다 parent가 먼저 종료하는 경우 발생하는 Orphan Process 문제를 다룬다. 일반적인 운영체제의 해결책으로는 parent process가 먼저 종료하면 parent child 관계에서 child process들을 모두 root process (init process)의 child로 바꾸어 줌으로써 해결하는데, pitos 상에서는 init process에게 Orphan process를 입양시키는 과정의 구현이 어려웠다. 때문에 이를 구현하는데 실패했고, 다른 방법으로 child process가 모두 종료할 때까지 parent process가 기다리며, child process의 정상종료, 비정상종료 여부를 parent process에서 handling 하는 방법으로 해결하였다.

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

