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

담당 교수 / 분반 : 김영재 / 2반

이름 / 학번 : 이예은 / 20171666

개발 기간 : 2020.11.12 ~ 2020.11.16

1. **개발 목표**

프로젝트1에서 구현하지 않은 나머지 system call을 구현한다. Create, remove, open, close, filsize, seek, tell을 새로 구현하고, 이전에 구현했던 read, write를 stdin, out에서 벗어나 file을 읽고 쓸 수 있도록 확장시킨다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

File descriptor는 프로세스에서 파일에 접근할 때 사용하는 추상적인 값으로, file descriptor를 통해 파일 포인터를 직접 넘기지 않아도 파일에 간단히 접근할 수 있다. pintos에서도 file descriptor를 사용하며, 각 thread가 자신의 독립적인 file descriptor를 가지게 구현한다.

2. System Calls

Create, remove, open, close, filesize, seek, tell, read, write의 system call을 구현하여 file system을 완성한다. 결과적으로 파일에 접근해 원하는 작업을 할 수 있게 된다.

3. Synchronization in Filesystem

만약 서로 다른 thread가 같은 파일에 동시 접근하여 동시에 read와 write를 할 경우 race condition이 발생해 정확한 값을 읽지 못하게 된다. 그러므로 read를 하고 있을 때에는 write를 막아야 하고, write 중일 때는 read를 막아야 한다. 이를 이번 프로젝트에 구현하여 각 thread가 문제없이 read, write를 할 수 있도록 만든다.

* 1. **개발 내용**

1. File Descriptor

File descriptor는 128 크기의 struct file\* 배열로 구현한다. 해당 배열에 open한 파일 구조체를 넣고 파일에 접근할 때 해당 파일이 저장된 배열의 인덱스를 주고받게 구현한다. 이를 통해 file\*를 파라미터로 주는 대신 int형 인덱스 값으로도 충분히 파일에 접근할 수 있다. 또한 해당 인덱스 값으로 배열에서 바로 원하는 file\*를 얻을 수 있다.

2. System Calls

Create: file 이름과 size를 받아 size 크기의 새로운 파일을 만든다.

Remove: file 이름을 받아 해당 이름의 파일을 삭제한다. 파일이 열려 있는지는 상관하지 않으며 열려 있는 파일을 닫지는 않는다.

Open: file 이름을 받아 해당 파일을 연다. 적절한 file descriptor를 반환한다.

Filesize: file descriptor를 받아 해당 파일의 크기를 반환한다.

Read: file descriptor, buffer, size를 받아 file descriptor에 맞게 파일 또는 stdin 입력을 읽는다.

Write: file descriptor, buffer, size를 받아 file descriptor에 맞게 파일 또는 stdout으로 쓴다.

Seek: file descriptor, position을 받아 해당 파일에서 다음에 읽거나 써야할 byte의 위치를 position으로 바꿔준다.

Tell: file descriptor를 받아 해당 파일에서 다음에 읽거나 써야할 byte의 위치를 반환한다.

Close: file descriptor를 받아 해당 파일을 닫는다.

3. Synchronization in Filesystem

Read와 write에서 synchronize가 필요하므로 해당 함수에서 read와 write를 시작할 때 lock\_acquire를 사용해 lock을 걸고, read와 write가 끝날 때 lock\_release를 사용해 lock을 해제한다. 또한 pintos는 running thread의 executable file이 삭제되거나 write되는 것을 막아야 하기 때문에 write를 하기 위해 반드시 거쳐야하는 open 함수에 lock을 걸고 file\_deny\_write를 설정해 잘못된 write 접근을 막는다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11.12: 프로젝트 파악

11.13: create, remove, open, filesize, seek, tell, close의 system calls 구현

11.14: 기존 코드에서 read, write 수정 및 synchronization 추가

11.15: open에 lock 추가

11.16: process exec, wait에 예외처리 추가 및 보고서 작성

* 1. **개발 방법**

1. File descriptor

이를 구현하기 위해 thread.h의 thread 구조체에 struct file\* file\_desp[128] 배열을 추가한다. 또한 thread.c의 init\_thread 함수에서 적절하게 초기화하는 코드를 추가한다.

1. System call

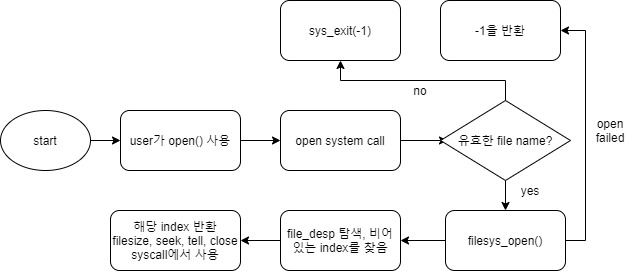
Syscall.c의 syscall\_handler에 각 system call에 맞는 함수를 부르도록 switch 문에서 case를 추가한다. 그리고 각 system call에 대해 이를 처리하는 함수를 syscall.c에 구현한다. 또한 file system으로 확장하기 위해 이전 sys\_read, sys\_write 함수를 수정한다. Sys\_exit 함수에도 thread가 끝났을 때, 해당 thread의 모든 file을 닫도록 코드를 추가한다.

1. Synchronization in file system

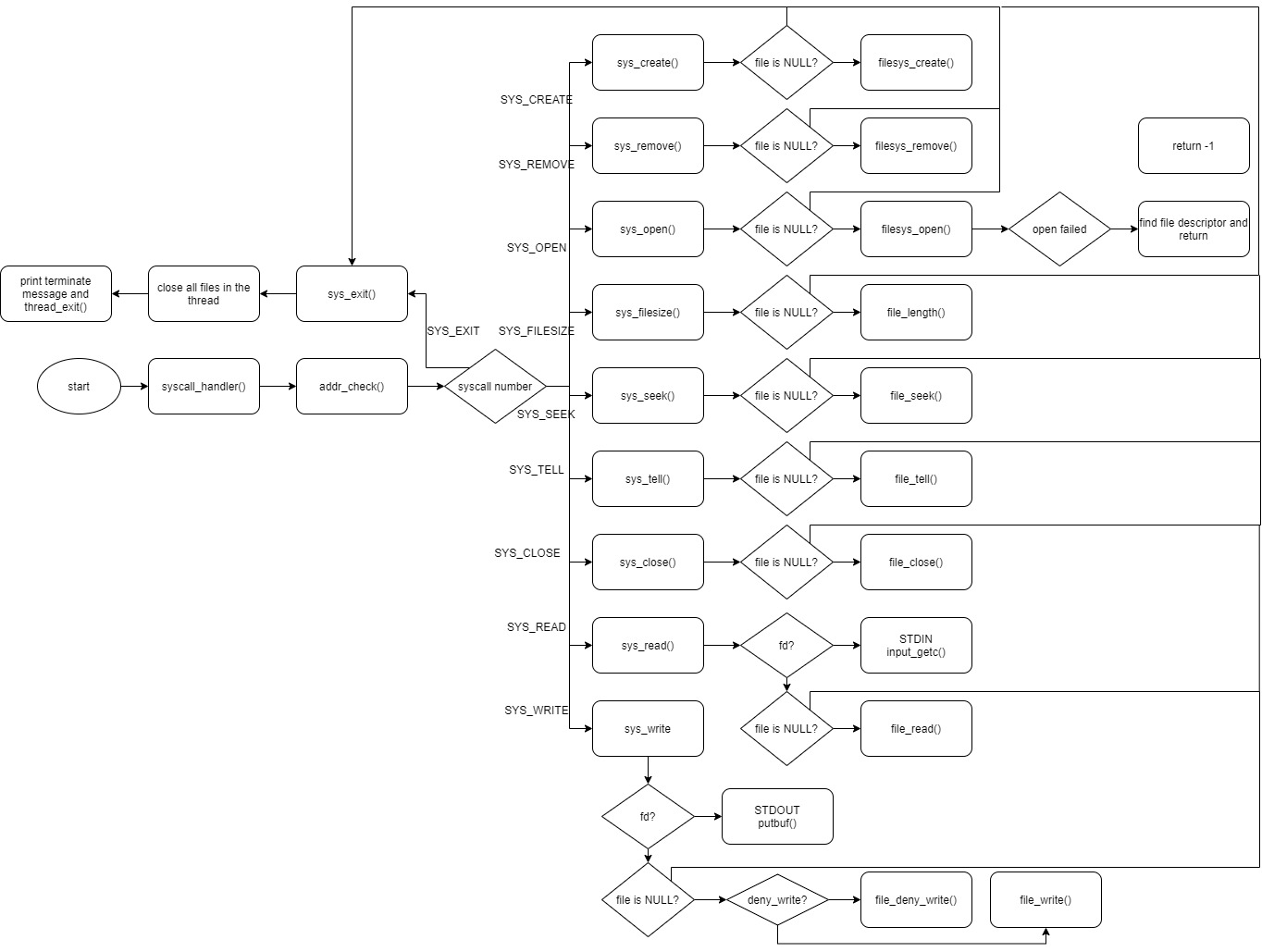
Sys\_read, sys\_write 함수를 수정할 때, 함수가 시작할 때 적절히 lock을 걸고 끝날 때 적절히 lock을 해제하는 코드를 추가한다. 또한 sys\_open에도 lock을 추가한다.

1. **연구 결과**
   1. **Flow Chart**

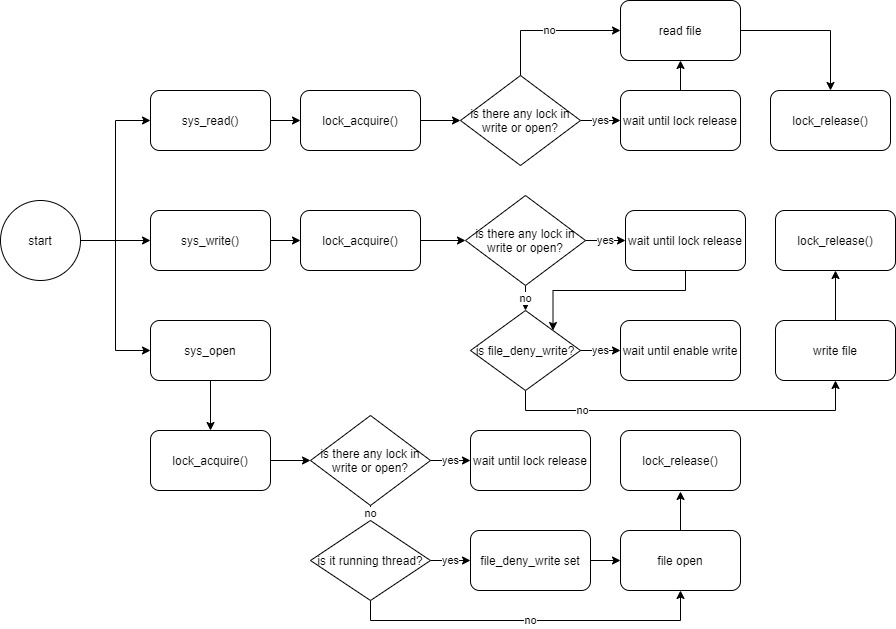
1. File descriptor



2. System call

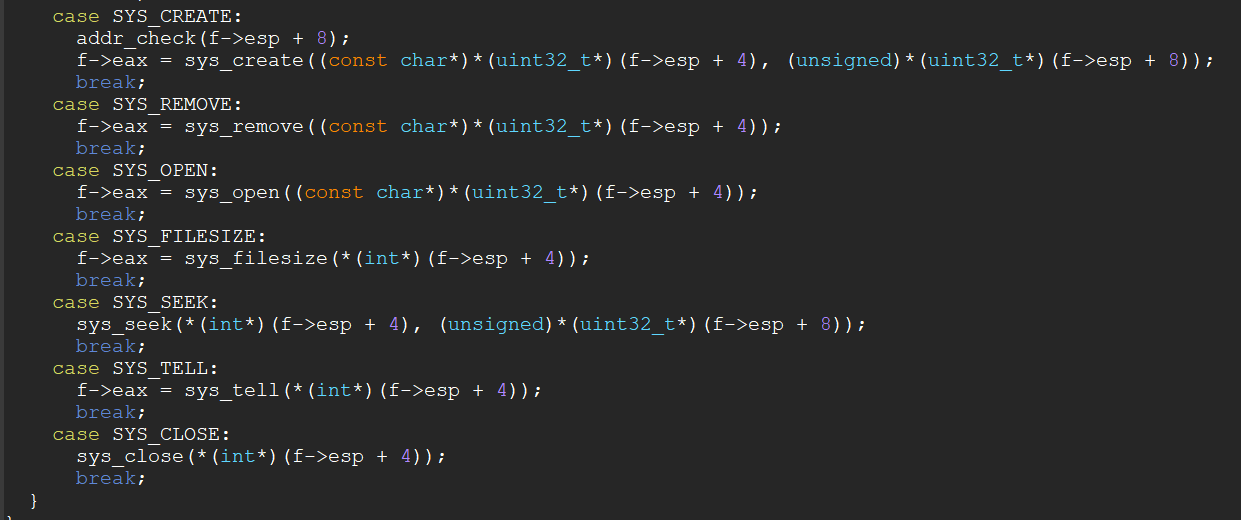


3. Synchronization in file system



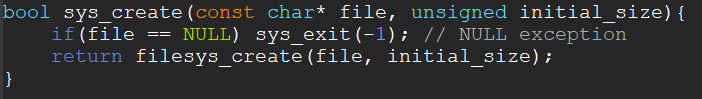
* 1. **제작 내용**

Syscall\_handler()



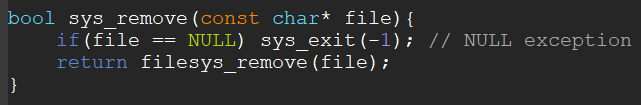
Syscall\_handler 함수에서 저번 1번 프로젝트의 system call 이후에 다음과 같이 create, remove, open, filesize, seek, tell, close case를 추가하고 각자 알맞은 syscall 함수를 불러준다. 파라미터가 여러 개일 경우, addr\_check를 통해 유효한 주소 값인지 검사 후 넘겨준다.

Sys\_create()



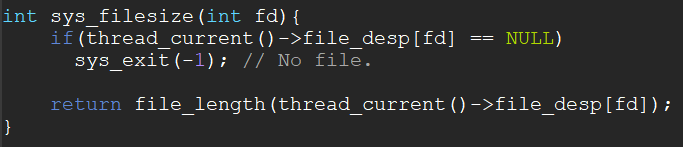
먼저 file 이름이 NULL이 아닌지 검사한다. NULL이라면 잘못된 파일 이름이므로 sys\_exit 함수를 불러 종료한다. 올바른 파일 이름이라면 filesys\_create 함수를 불러 새로운 파일을 생성하고 이에 따른 결과를 반환한다.

Sys\_remove()



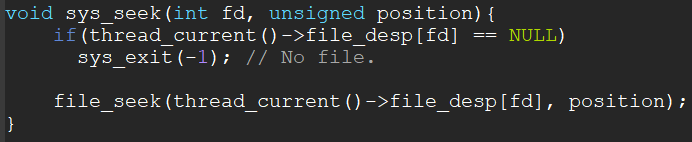
먼저 file 이름이 NULL이 아닌지 검사한 후 잘못된 파일 이름이라면 sys\_exit을 함수를 불러 종료한다. 올바른 파일 이름이라면 filesys\_remove 함수를 불러 파일을 삭제하고 이에 따른 결과를 반환한다.

Sys\_filesyze()



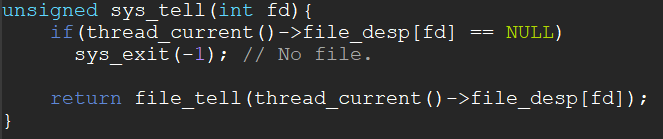
File descriptor인 fd를 통해 현재 thread에서 해당 fd를 인덱스로 가지는 파일을 찾는다. 이 파일이 NULL이라면 잘못된 접근이므로 sys\_exit으로 종료하고 아니라면 file\_length 함수를 불러 해당 파일의 크기를 반환한다.

Sys\_seek()



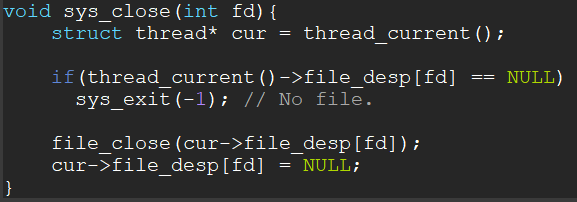
File descriptor인 fd를 통해 현재 thread에서 해당 fd를 인덱스로 가지는 파일을 찾는다. 이 파일이 NULL이라면 잘못된 접근이므로 sys\_exit으로 종료하고 아니라면 file\_seek 함수를 부른다.

Sys\_tell()



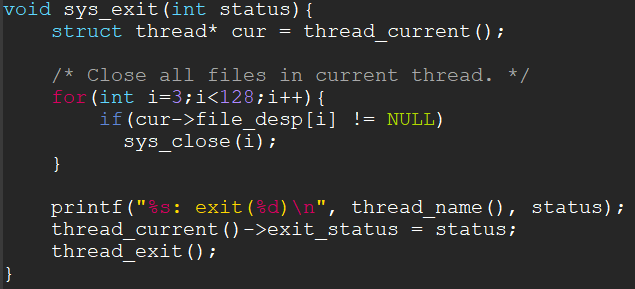
File descriptor인 fd를 통해 현재 thread에서 해당 fd를 인덱스로 가지는 파일을 찾는다. 이 파일이 NULL이라면 잘못된 접근이므로 sys\_exit으로 종료하고 아니라면 file\_tell 함수를 부른다.

Sys\_close()



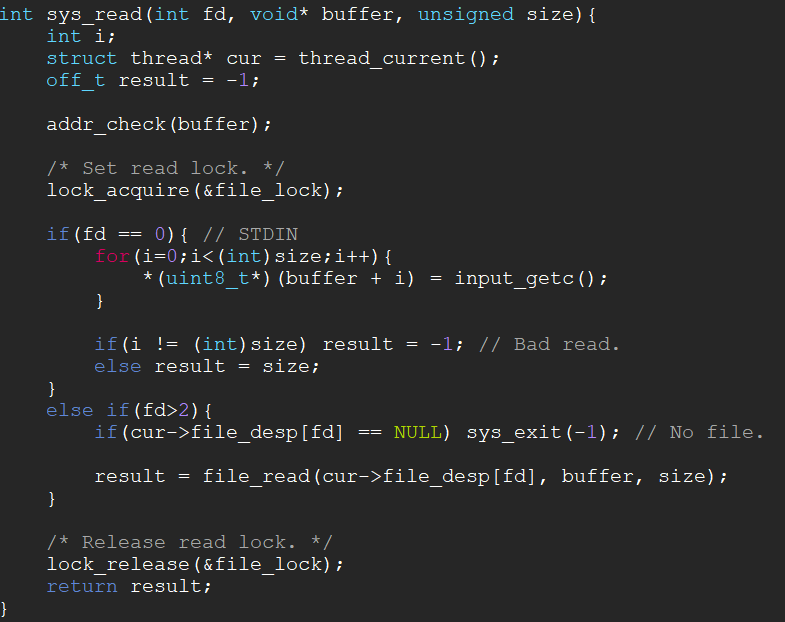
File descriptor인 fd를 통해 현재 thread에서 해당 fd를 인덱스로 가지는 파일을 찾는다. 이 파일이 NULL이라면 잘못된 접근이므로 sys\_exit으로 종료하고 아니라면 file\_close 함수를 불러 해당 파일을 닫고, thread의 file descriptor에 NULL을 넣어 갱신한다.

Sys\_exit()



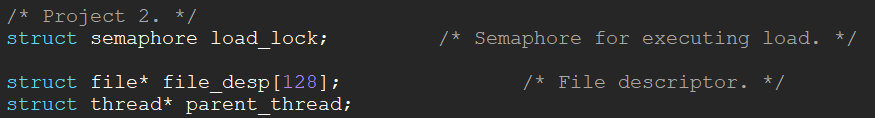
Sys\_exit을 할 때, 해당 thread의 열려 있는 file들을 모두 닫아야 한다. 이를 위해 file\_desp 배열을 검사해 NULL이 아니라면 아직 열려 있는 파일이므로 sys\_close를 불러 파일을 닫는다.

Sys\_read()



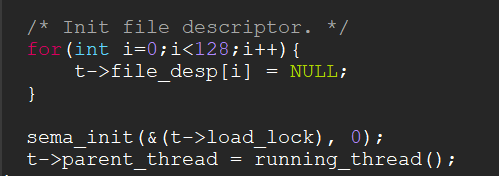
Sync를 맞추기 위해 read의 가장 처음에 lock\_acquire를 불러 lock을 세팅한다. Fd가 0일일 때 stdin으로 이전 프로젝트 1과 같아 그대로 둔다. Fd가 3부터 파일을 읽는 것으로 먼저 해당 file descriptor의 파일이 유효한지 확인한다. NULL이라면 잘못된 파일이므로 sys\_exit으로 종료한다. 아니라면 file\_read 함수를 불러 파일을 읽고 모든 read가 끝나면 lock\_release를 불러 lock을 풀고 읽은 결과를 반환하며 종료한다.

Thread.h



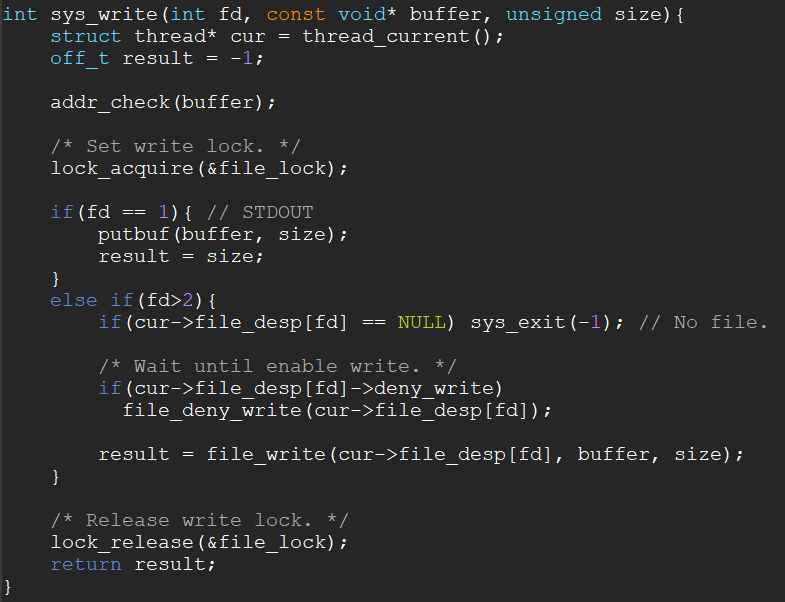
추가적으로 load를 위해 필요한 semaphore인 load\_lock과 file descriptor 배열, 부모 thread를 저장할 변수를 thread 구조체에 추가한다.

Init\_thread()



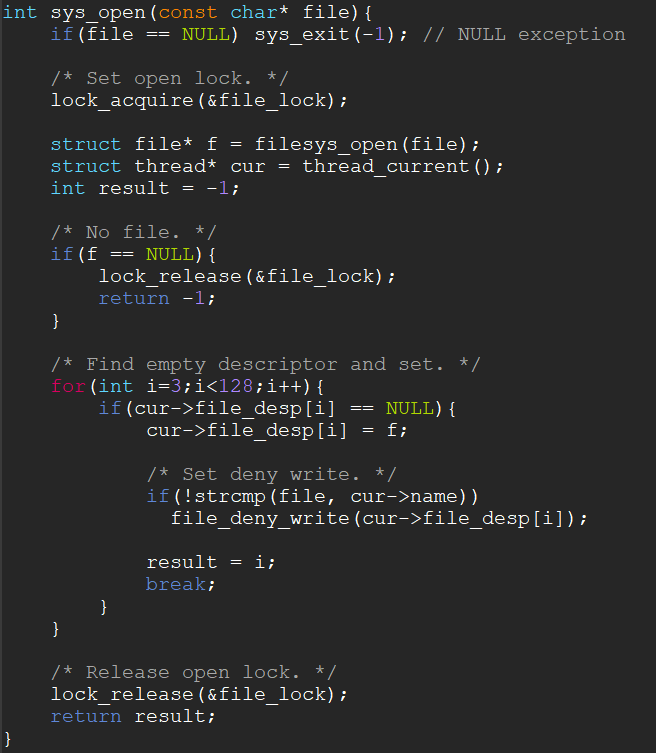
추가한 변수에 대해 모두 초기화한다.

Sys\_write()



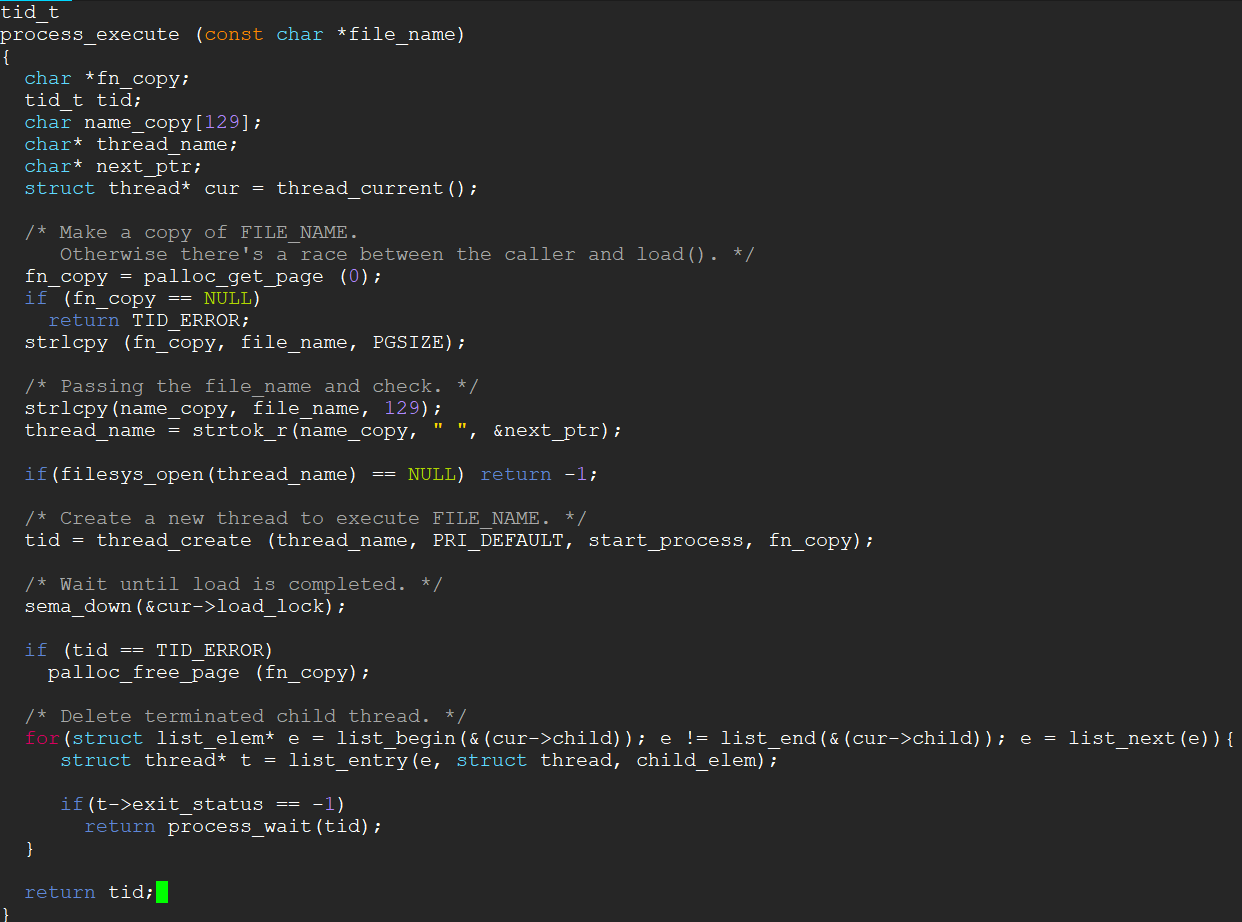
Sync를 맞추기 위해 가장 처음에 lock\_acquire를 불러 lock을 세팅한다. Fd가 1이라면 stdout으로 프로젝트 1과 같아 그대로 둔다. Fd가 3부터 파일 쓰기로 먼저 해당 fd의 파일이 유효한지 검사한다. NULL이라면 잘못된 파일이므로 sys\_exit을 한다. 그리고 지금 쓰기가 가능한지 검사하고 기다린다. 만약 file의 deny\_write가 1이라면 현재 쓸 수 없으므로 file\_deny\_write를 불러 enable 할 수 있을 때까지 기다린다. 이후 file\_write를 불러 write하고 lock\_release를 불러 lock을 해제한다. 이후 결과를 반환하고 종료한다.

Sys\_open()



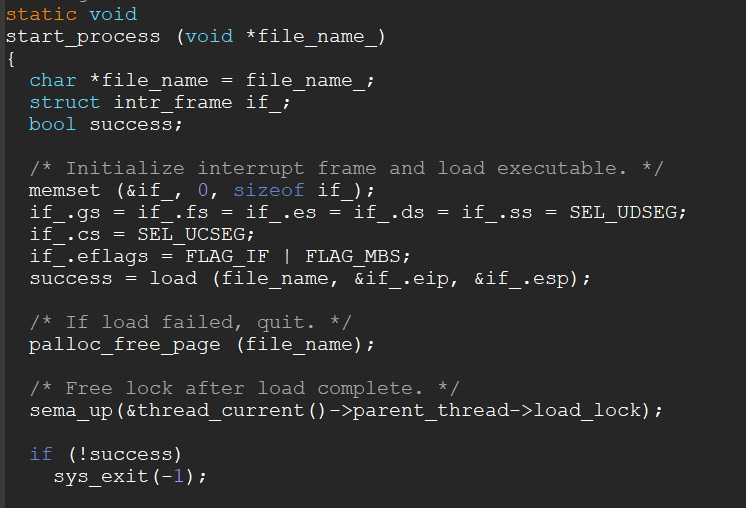
Open도 sync를 맞추기 위해 lock\_acquire를 불러 lock을 세팅한다. 먼저 file 이름이 유효한지 검사하고 NULL이라면 잘못된 이름으로 sys\_exit을 부른다. Filesys\_open으로 해당 이름의 파일을 열고 성공적으로 열었는지 검사한다. 반환값이 NULL로 잘못 열었다면 lock을 해제하고 -1을 반환한다. 성공적으로 열었다면, 해당 thread의 file\_desp 배열을 탐색하며 비어있는 가장 작은 인덱스의 자리를 찾는다. 찾았다면 해당 배열에 연 파일을 넣어준다. 그리고 만약 연 파일 이름과 현재 돌아가고 있는 thread의 이름이 같다면, running thread에 새로 write하려 접근하는 것일 수 있으므로 file\_deny\_write 함수를 불러 write가 enable 할 때까지 기다리도록 한다. 이후 lock을 해제하고 결과를 반환한다.

Process\_execute()



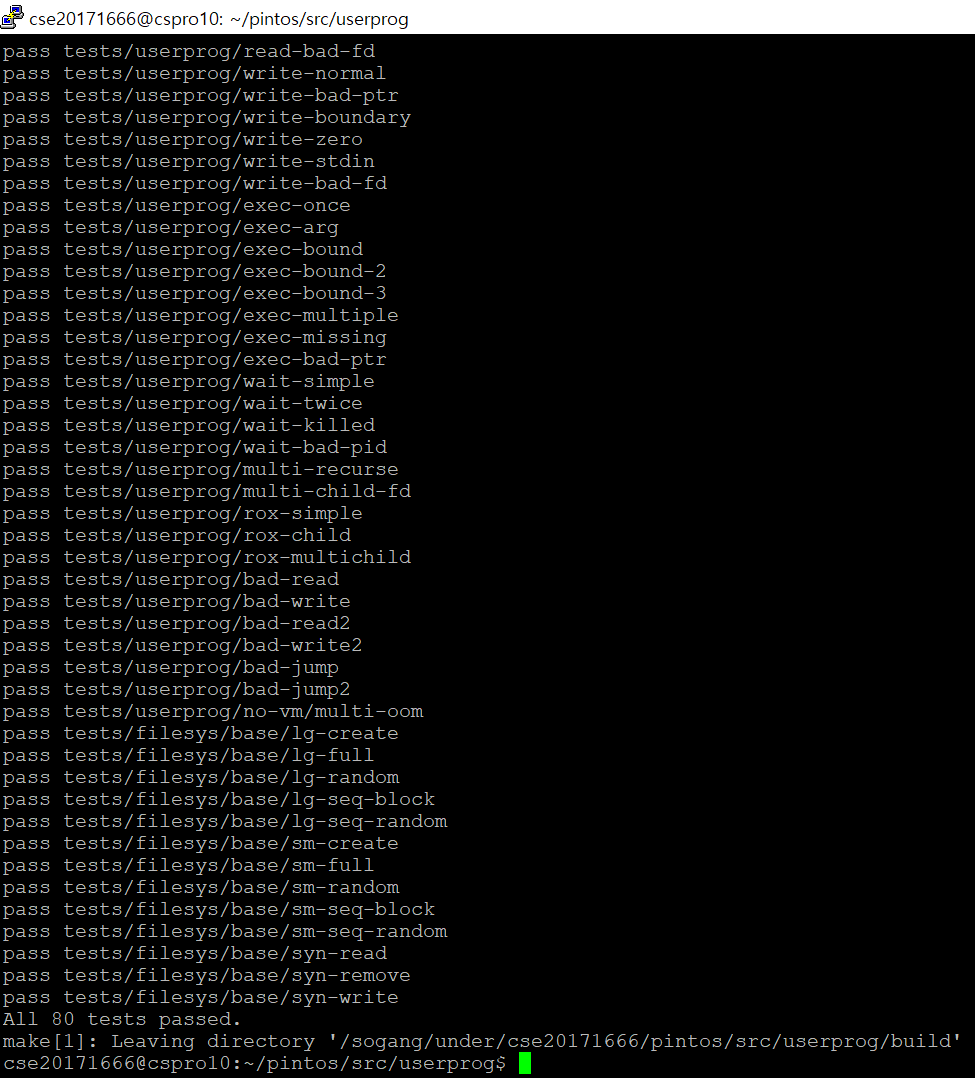
Multi-oom을 해결하기 위해 다음과 같은 코드를 process\_execute에 추가한다. 이는 새로 만들어진 child thread가 load되기 전에 parent가 죽거나 끝나버리는 경우를 처리하기 위해서이다. 이를 위해 thread를 create하고 load되기 전까지 sema\_down을 통해 기다리도록 한다. 또한 만약 child thread가 비정상적으로 종료되었을 때 이를 처리해주어야 하므로 현재 thread의 전체 child를 돌며 exit\_status가 -1인 경우를 찾아 process\_wait를 불러 정리해주도록 구현한다.

Start\_process()



해당 함수에서 load함수가 끝나면 sema\_up을 하여 기다리고 있는 parent의 load\_lock을 해제한다.

* 1. **시험 및 평가 내용**

****