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

담당 교수 / 분반 : 박성용 교수님 / 1반

이름 / 학번 : 장서우 / 20181679

개발 기간 : 2020.11.02~2020.11.15

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

file system이란 file과 directory를 관리하는 kernel의 모듈이다. 이런 file system과 관련된 system call (create, remove, open, close, filesize, read, write, seek, tell)을 구현한다. 이를 구현하기 위해 Base File System을 이해한다. 또한, 프로세스, 파일 간의 동기화를 위해 semaphore와 filelock을 사용한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

open, create의 return 값으로, 프로세스들이 파일에 부여해주는 구분자로, pintos에서 각 thread는 각각의 file descriptor를 얻고 관리한다. pintos에서 제공하는 file system 관련 함수들을 정상적으로 이용하기 위해서는 file descriptor가 필요하다. fd는 기본적으로 0은 STDIN, 1은 STDOUT, 2는 STDERR를 의미한다. 우리는 3 이상 양의 정수에 해당하는 fd를 사용할 수 있으며 파일이 열릴 때마다 부여하기 때문에 동일한 파일이 열리더라도 다른 fd 값이 부여된다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

user program에서 file system을 사용할 수 있도록 system call에 file system function들을 syscall.c에 추가한다. (create(파일 생성), remove(파일 삭제), open(파일 열기), close(파일 닫기), filesize(파일 크기 얻어오기), read(파일 읽기), write(파일 쓰기), seek(파일 탐색), tell(현재 파일 위치))

3. Synchronization in Filesystem

multi-child 간의 synchronization을 위하여 semaphore를 사용한다. file synchronization을 위하여 filelock을 사용한다. 여러 프로세스가 동시에 수행되고 있을 때, 한 프로세스가 읽거나, 쓰는 파일에 대하여, 다른 프로세스가 읽거나, 쓰게 되면, 데이터에 이상이 발생할 수 있기 때문에, synchronization을 통해 critical section을 보호하여 이를 해결한다. 또한, running program의 executable file이 수정되지 않도록 file deny write를 활용한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

struct file pointer을 사용하여 struct thread에 struct file \*fd[128];을 추가해주었다. pintos에서 kernel이 파일에 접근할 때는 struct file이라는 구조체를 사용한다. fd값을 통하여 struct file을 찾을 수 있도록 구현하였다. 또한 하나의 프로세스에서 최대 128개의 fd를 가질 수 있도록 하고 있다.

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)

* + bool sys\_create(const char \*file, unsigned initial\_size): file을 생성해주는 함수이다. filesys\_create를 사용하여 구현한다.
  + int sys\_remove(const char \*file): file을 삭제한다. filesys\_remove를 사용하여 구현한다.
  + int sys\_open(const char \*file): filesys\_open을 사용하여 특정 file을 연다. 성공하면 file descriptor를 return한다.
  + int syscall\_read(int fd, void \*buffer, unsigned size): file descriptor이 0인 경우(STDIN), Input\_getc() 함수를 이용하여 한 글자씩 읽는다. file descriptor가 3이상일 경우 file\_read를 사용하여 구현하고, filelock을 사용하여 synchronization을 한다.
  + int syscall\_write(int fd, void \*buffer, unsigned size): file descriptor이 1인 경우(STDOUT), putbuf() 함수를 이용하여 출력한다. file descriptor가 3이상일 경우 file\_write를 사용하여 구현하고, filelock을 사용하여 synchronization을 한다.
  + unsigned sys\_filesize(int fd): file\_length를 사용하여 fd에 해당하는 특정 file의 크기를 return한다.
  + void sys\_close(int fd): file\_close를 사용하여 fd에 해당하는 특정 file을 닫는다.
  + void sys\_seek(int fd, unsigned pos): file\_seek를 사용하여 fd에 해당하는 특정 file의 위치를 position으로 바꿔주는 역할을 하는 함수이다.
  + unsigned sys\_tell(int fd): file\_tell을 사용하여 fd에 해당하는 특정 file의 위치를 return한다.

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

pintos에서는 running program의 executable file이 수정되지 않도록 해야 하기 때문에, sys\_open에서 file\_deny\_write를 사용하여 제한시켰다. parent와 child thread에 대한 동기화는 semaphore, file 접근과 관련해서는 lock을 사용하여 구현하였다. parent가 각각의 child에게 wait 명령을 하기 전까지 죽지 못하도록 해야 하고, child thread가 load되기 전에 parent thread의 execute가 끝나면 안되기 때문에 child의 load후 parent thread를 execute하도록 적절히 sema\_down과 sema\_up을 활용한다. 또한, 파일을 처리할 때에는 동시적인 처리를 하면 안되는 부분을 critical section으로 보호해주어야한다. 이를 lock을 사용하여 static struct lock filelock을 선언하여 lock\_acquire과 lock\_release를 사용하여 구현하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
* 11. 02 ~ 11. 08: 매뉴얼 분석 및 pintos 코드 이해
* 11. 09 ~ 11. 14: system call 구현 (synchronization 포함)
* 11. 15: 마무리 작업, 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

* threads/thread.h에서 struct thread에 struct file \*fd[128]을 추가하고 threads/thread.c의 init\_thread()에서 초기화를 해준다.
* userprog/syscall.c에서 각 file system function에 맞게 적절하게 사용한다.

2. System Calls

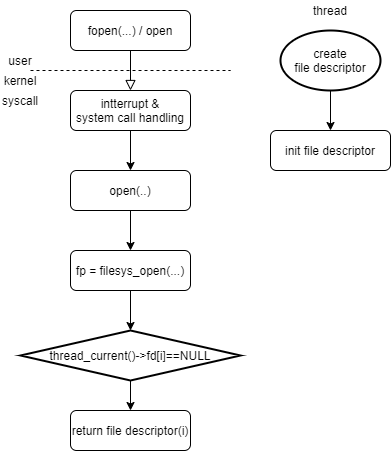
* userprog/syscall.c에서 syscall\_handler()의 switch문안에 새로운 file system을 적절히 추가해준다. 이는 project1의 내용을 참고한다.
* userprog/syscall.c에 file system과 관련한 system call function을 구현한다: sys\_create(), sys\_remove(), sys\_open(), syscall\_read(), syscall\_write(), sys\_filesize(), sys\_close(), sys\_seek(), sys\_tell()
* 이에 맞게 userprog/syscall.h를 수정한다.
* userprog/exception.c의 page\_fault() 일부분을 오류에 맞게 수정해주었다.

3. Synchronization in Filesystem

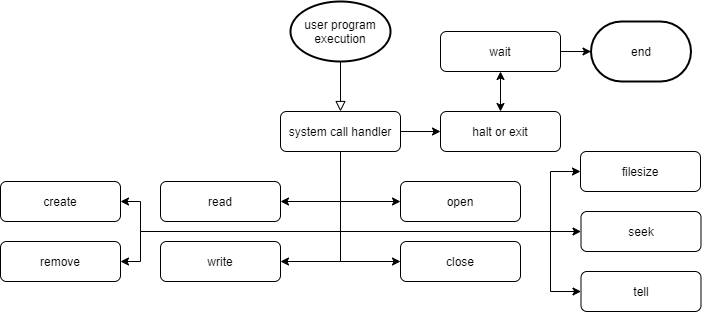
* threads/thread.h에서 struct thread에 load, exit, wait에 관한 semaphore를 만들고 threads/thread.c의 init\_thread()에서 sema\_init()을 통해 추가한 변수를 초기화 해준다.
* userprog/process.c에서 parent, child thread를 생성하고 종료할 때 zombie process와 orphan process가 발생하지 않도록 process\_exit()와 process\_wait() 함수에서 exit, wait semaphore를 사용하고, start\_process(), process\_execute() 함수에서 load semaphore를 사용하여 동기화를 시켜준다.
* userprog/syscall.c에서 static struct lock filelock을 선언하고 syscall\_init()에서 lock\_init()을 해주고, syscall\_read()와 syscall\_write()에서 lock\_acquire()과 lock\_release()를 사용하여 동기화를 시켜준다.

1. **연구 결과**
   1. **Flow Chart**

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

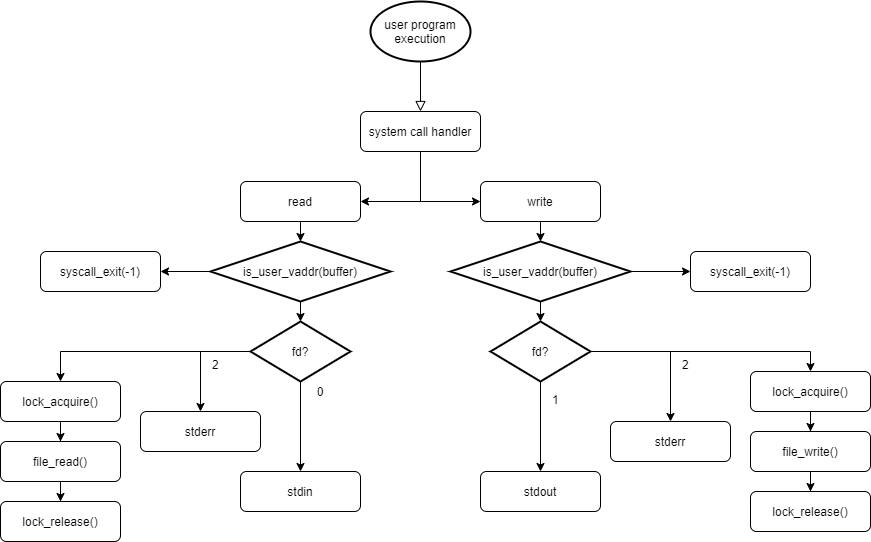
1. File Descriptor  


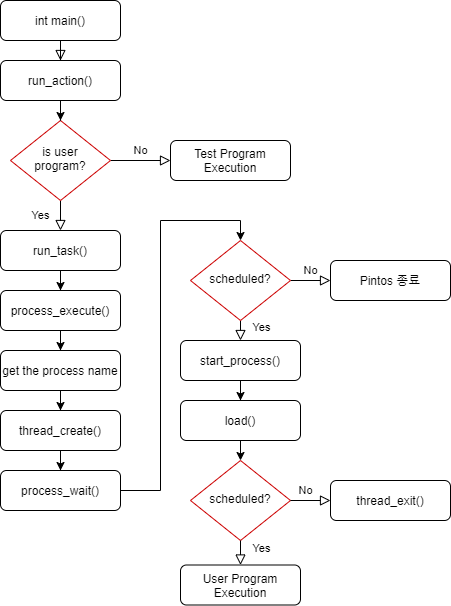
2. System Calls



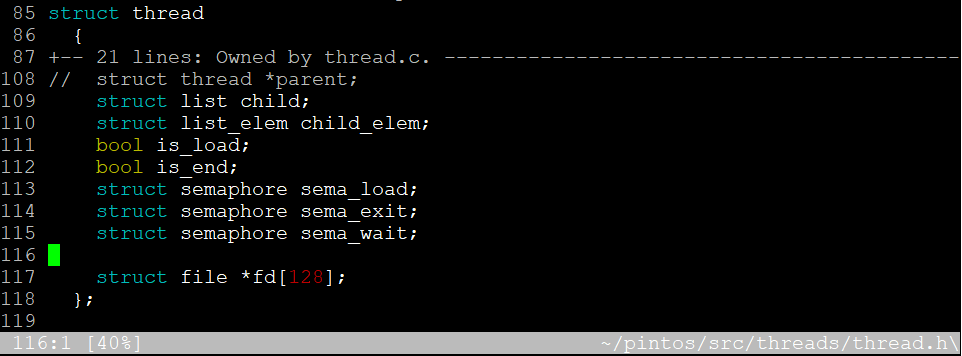
3. Synchronization in Filesystem

* read/write

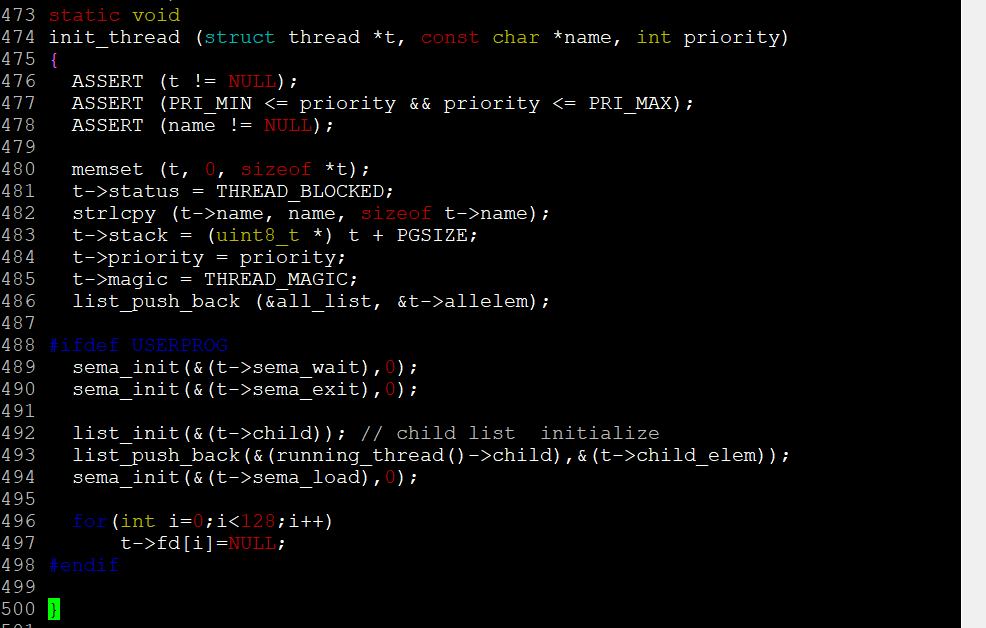


* process는 다음과 같은 flowchart를 따른다.  
  
  1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

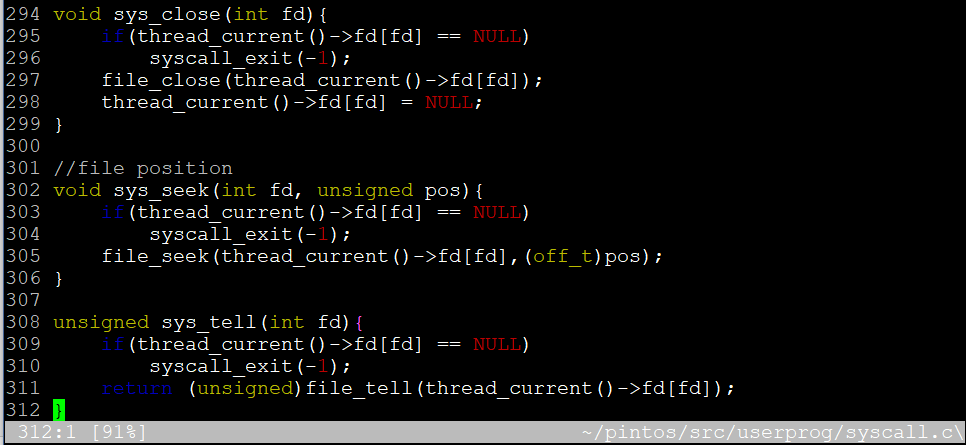
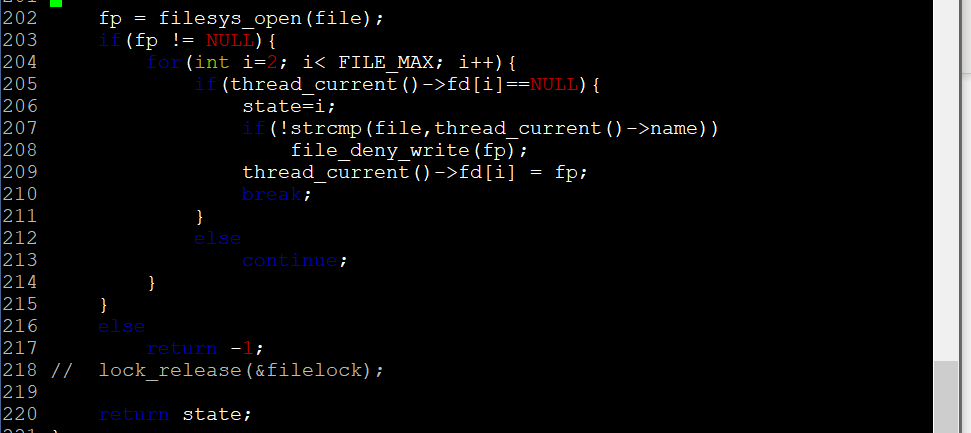
**1. File Descriptor**



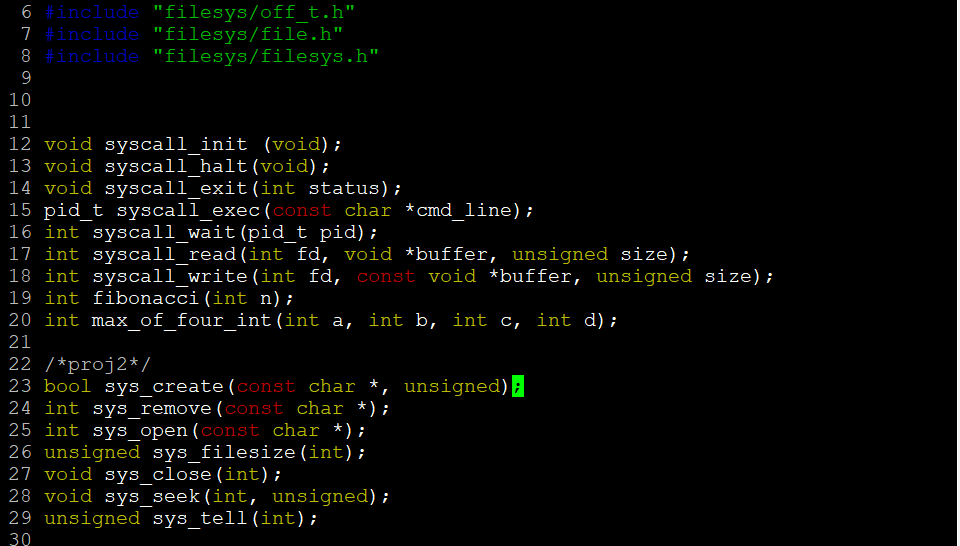
* thread.h의 struct thread에 다음과 같이 file descriptor를 추가해준다.

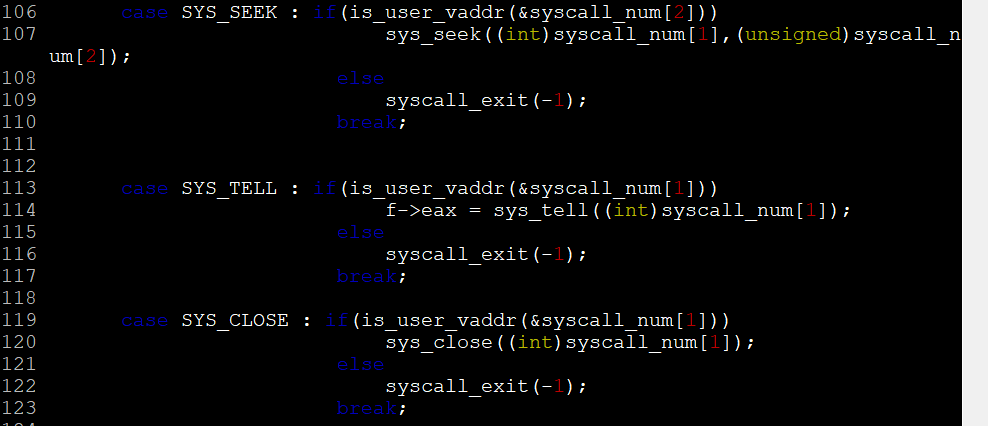
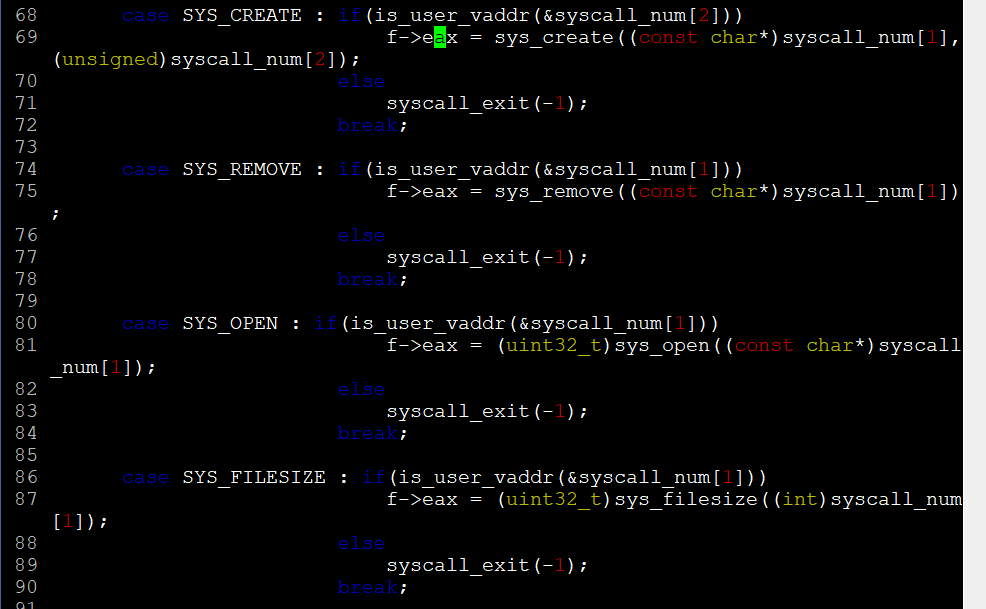


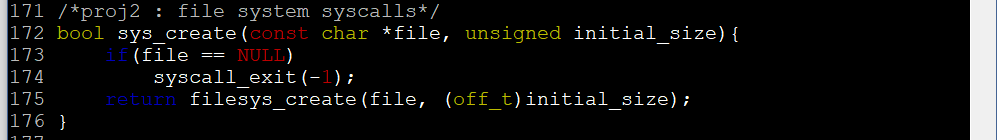
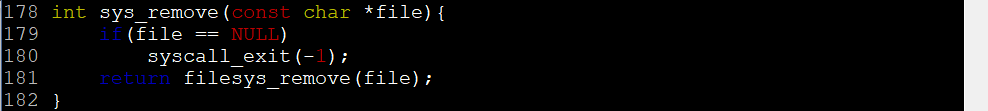
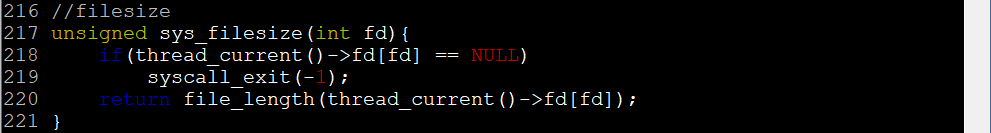
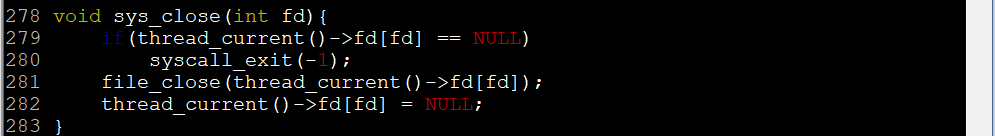
* thread.c의 init\_thread()에서 다음과 같이 초기화를 해준다.
* 최대 길이는 FILE\_MAX 128로 지정해준다.
* userprog/syscall.c에서 sys\_open()을 통해 file descriptor를 반환해주고, 이를 다양한 syscall에서 사용한다.

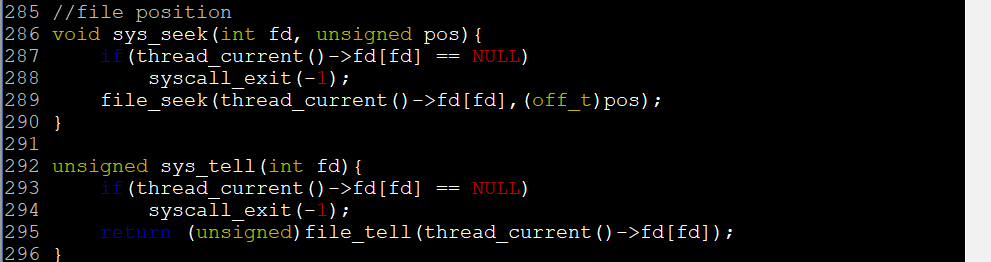


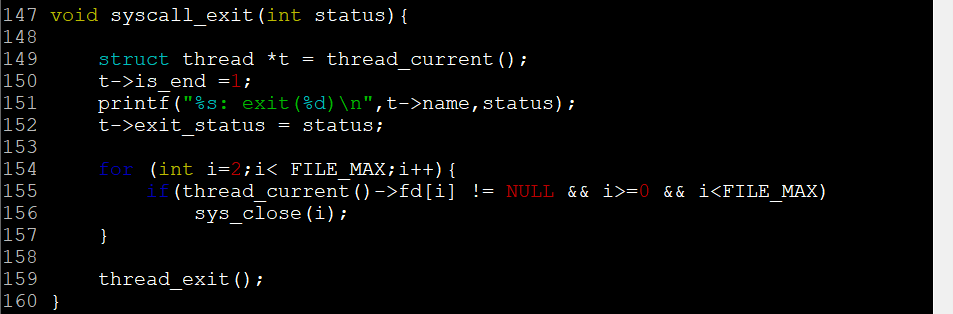
**2. System Calls**

* userprog/syscall.h에 file system과 관련한 함수들이 다음과 같이 추가되었다.   
  
* userprog/syscall.c에서 다음과 같은 코드들이 추가되었다.
* system call handler에서 sys\_create, sys\_remove, sys\_open, sys\_filesize, sys\_seek, sys\_tell, sys\_close가 추가되었다.



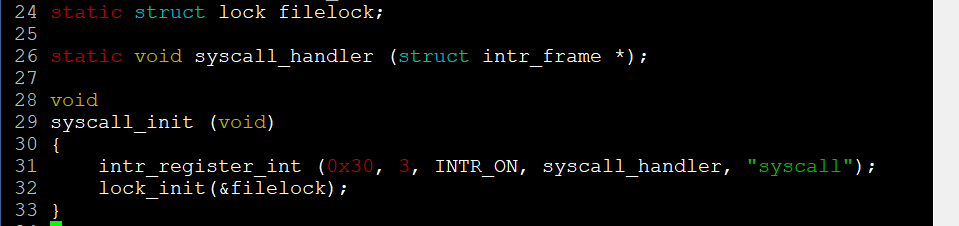
* sys\_create(): filesys\_create를 사용하여 file을 생성하는 함수를 구현한다. 파일명과 사이즈를 인자로 받고 파일명이 NULL이면 syscall\_exit(-1)을 수행한다. 아닐 경우, filesys\_create()를 return한다.  
  
* sys\_remove(): filesys\_remove를 사용하여 file을 삭제하는 함수를 구현한다. 파일명을 인자로 받고 파일명이 NULL이면 syscall\_exit(-1)을 수행한다. 아닐 경우, filesys\_remove()를 return한다.  
  
* sys\_open(): filesys\_open을 사용하여 특정 file을 연다. 파일명을 인자로 받고 파일명이 NULL이면 syscall\_exit(-1)을 수행한다. file 주소 범위 유효성을 체크한다. 그 이후, 인자로 받은 파일명을 가진 파일의 유효성을 체크한다. 그 후, file descriptor 배열에서 비어 있는 곳을 찾아 open한 file의 포인터를 넘겨주고 state에 해당 fd를 저장한다. 파일 이름과 현재 실행중인 파일 이름이 같다면 file\_deny\_write()를 통해 file open을 제한한다. state를 return한다.  
  
* sys\_filesize(): file descriptor를 받아 file\_length()를 사용하여 file size를 return한다.  
  
* sys\_close(): file\_close()를 사용하여 파일을 닫아준다.  
  
* sys\_seek()/sys\_tell(): file의 position과 관련된 함수로 sys\_seek는 file\_seek()를 호출하고, sys\_tell()는 file\_tell()를 적절히 인자를 받아 return한다.

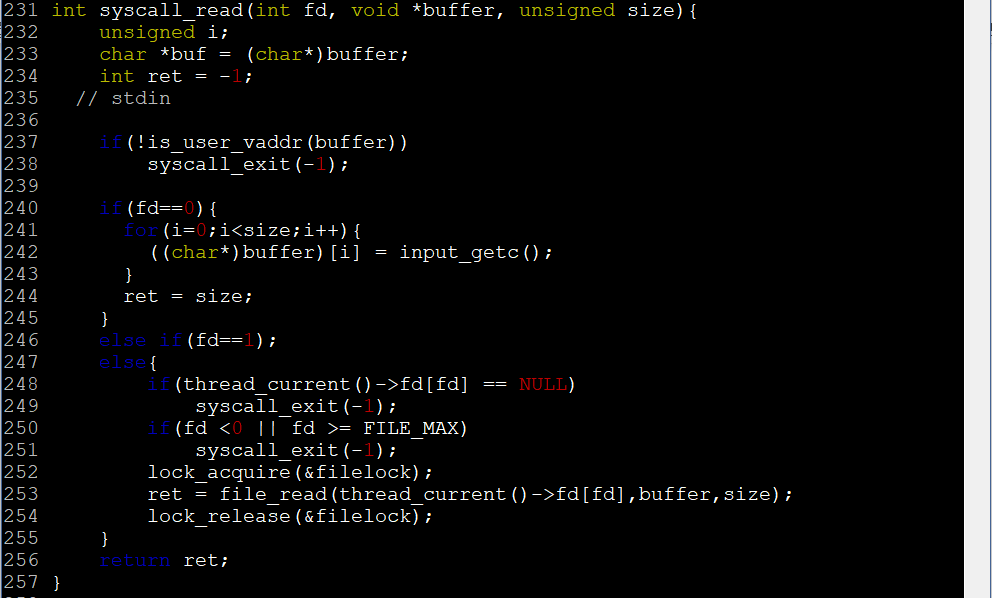
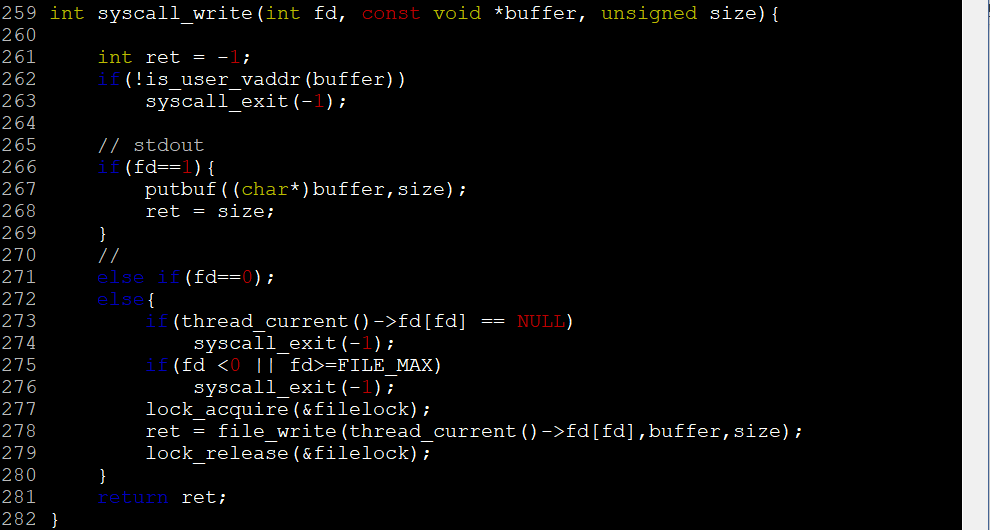


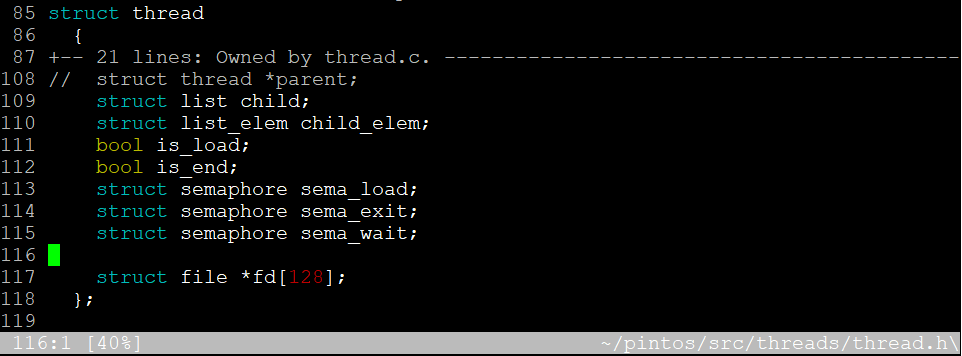
* syscall\_exit() : 기존의 syscall\_exit()을 약간 수정하여 status를 조정해주었고, 조건에 맞는 file의 sys\_close()를 실행해주었다.  
  
* syscall\_read()와 write()는 3번에 이어서 설명이 나온다.

**3. Synchronization in Filesystem**

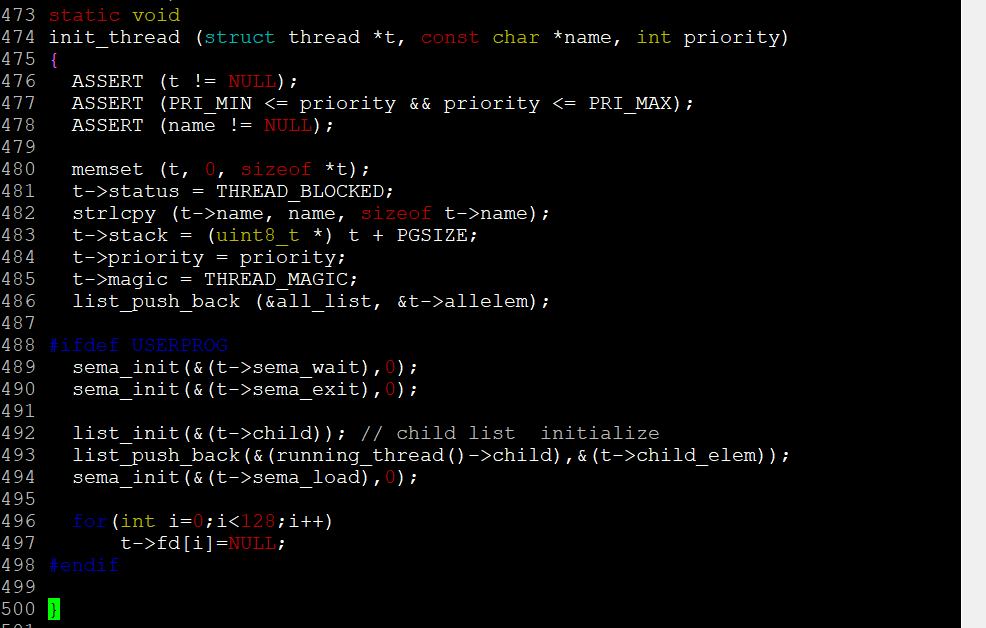
* **lock**
* 아래 그림과 같이 lock을 선언하고 syscall\_init에서 lock\_init을 해준다.



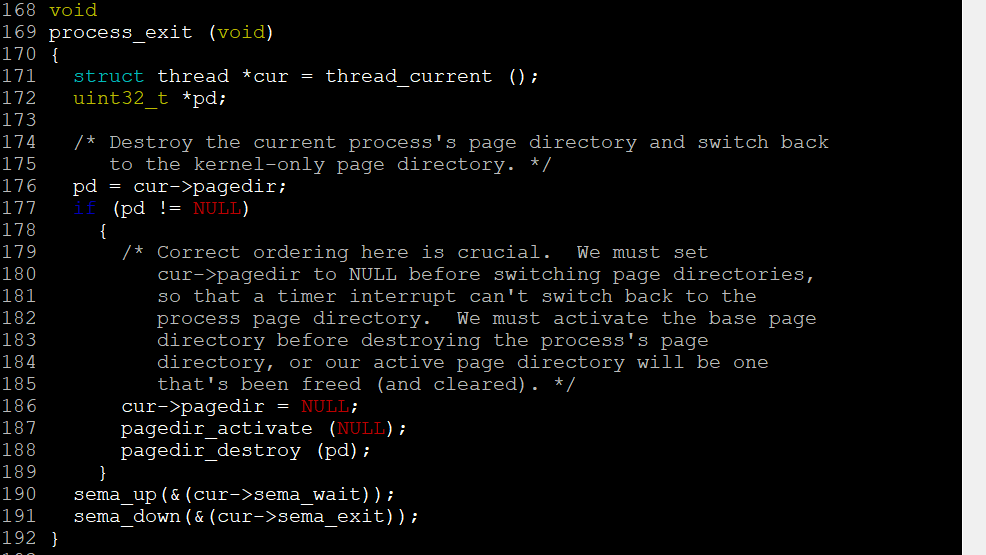
* syscall\_read()/syscall\_write(): 이전에 stdin, stdout에 대해서만 구현한 syscall\_read()와 syscall\_write()를 3이상의 file descriptor(2는 stderr)에 대하여 추가적으로 구현해준다. 이는 filelock을 사용하여 file system의 synchronization을 해준다. 또한, 여기서 buffer의 유효성을 체크해주어 read-bad-ptr 문제를 해결할 수 있었다.  
  
* 
* **semaphore**

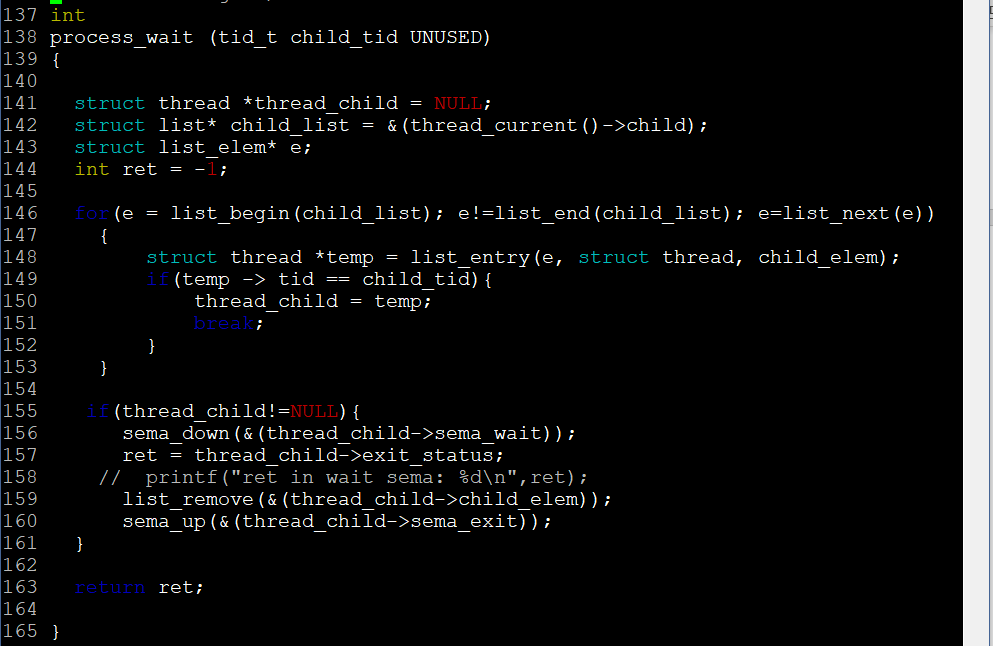


* thread.h의 struct thread에 다음과 같이 semaphore를 추가해주었다.

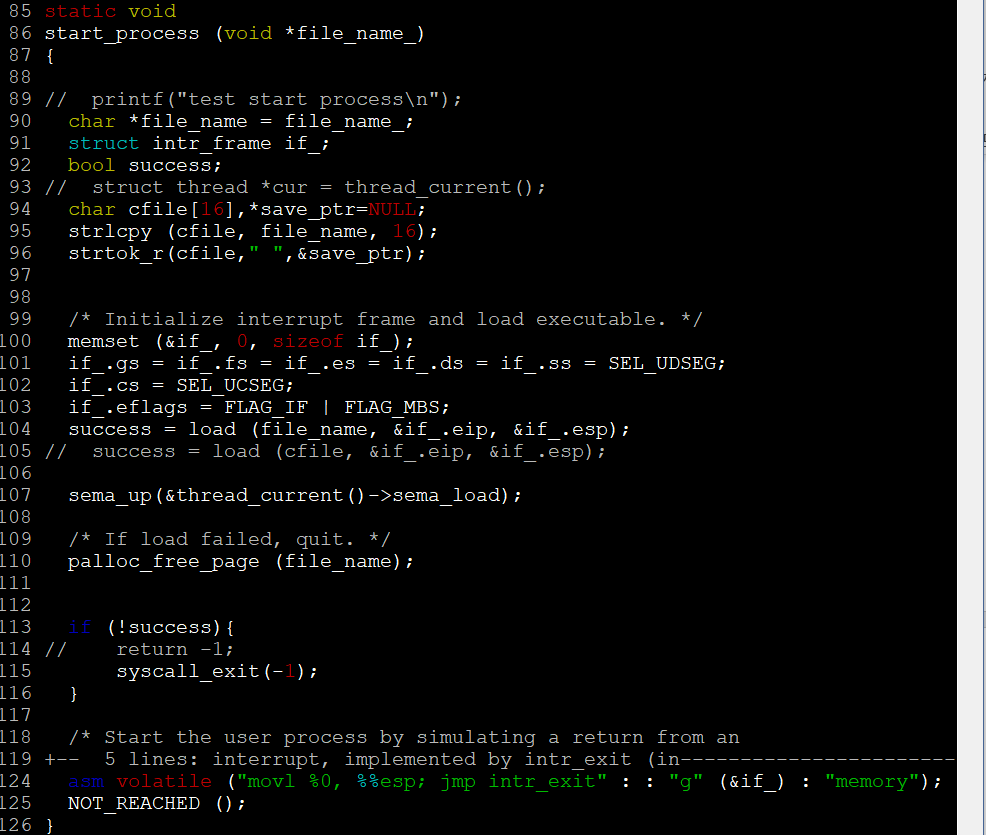
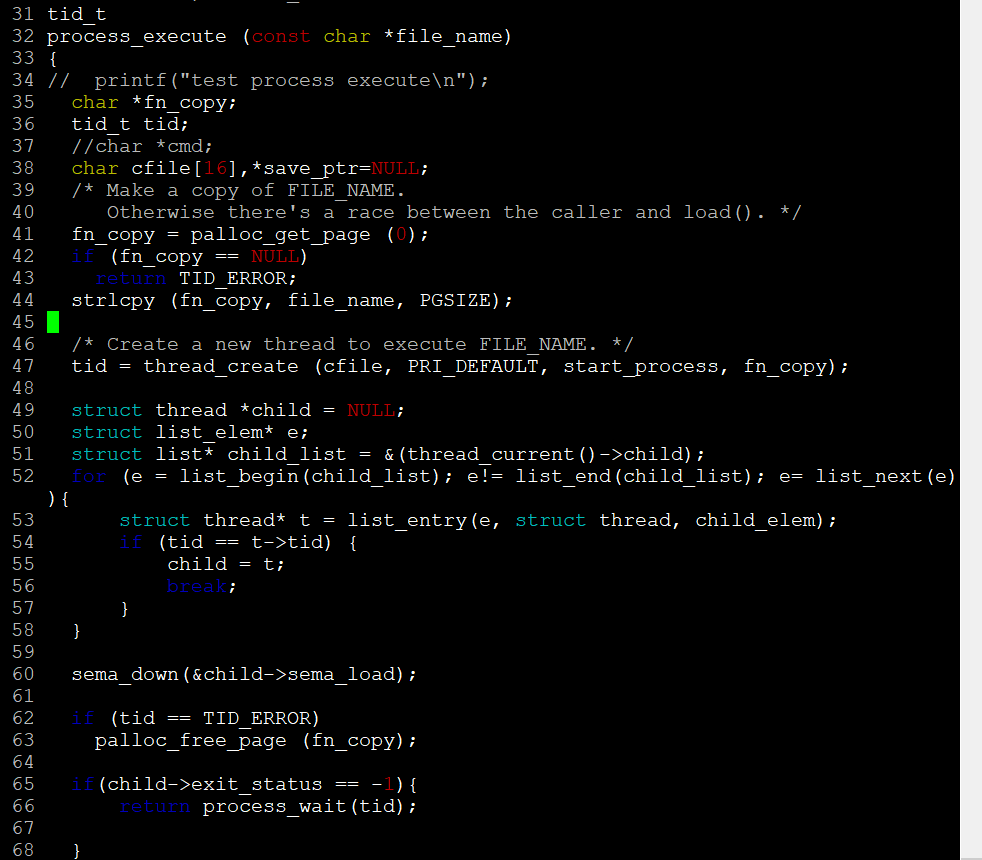


* thread.c의 init\_thread()에서 다음과 같이 초기화를 해준다.
* userprog/process.c에서는 project 1 에서의 semaphore를 강화하여 구현하였다. 이로써 syn\_read와 syn\_write까지 완전히 작동시킬 수 있었다. 다음과 같이 process\_wait()와 process\_exit()에서 exit semaphore와 wait semaphore를 사용하였다. process\_wait()에 존재하는 sema\_down, sema\_up과 process\_exit()에 존재하는 sema\_up, sema\_down이 연결되어 있는 형식이다. 이를 통하여 자식 thread가 죽을 때까지 기다리는 형식의 synchronization이 가능하였다.

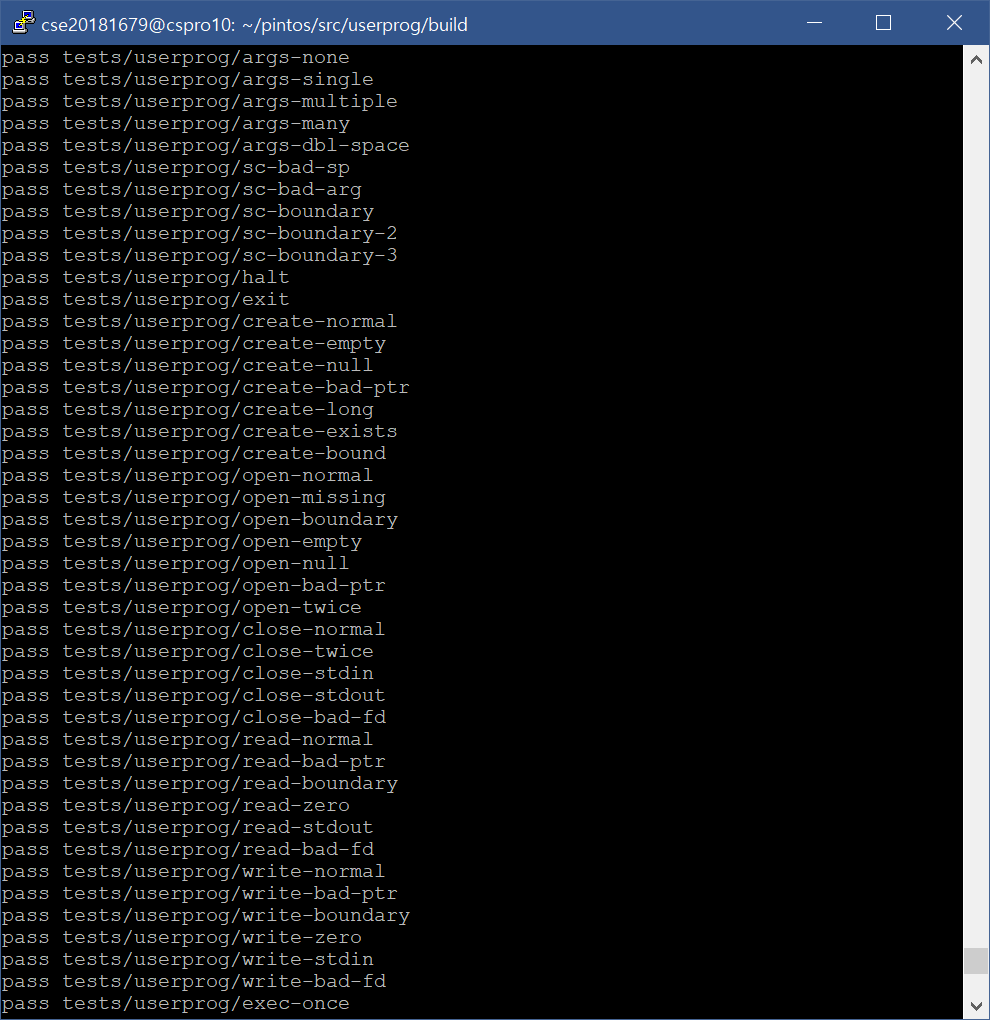


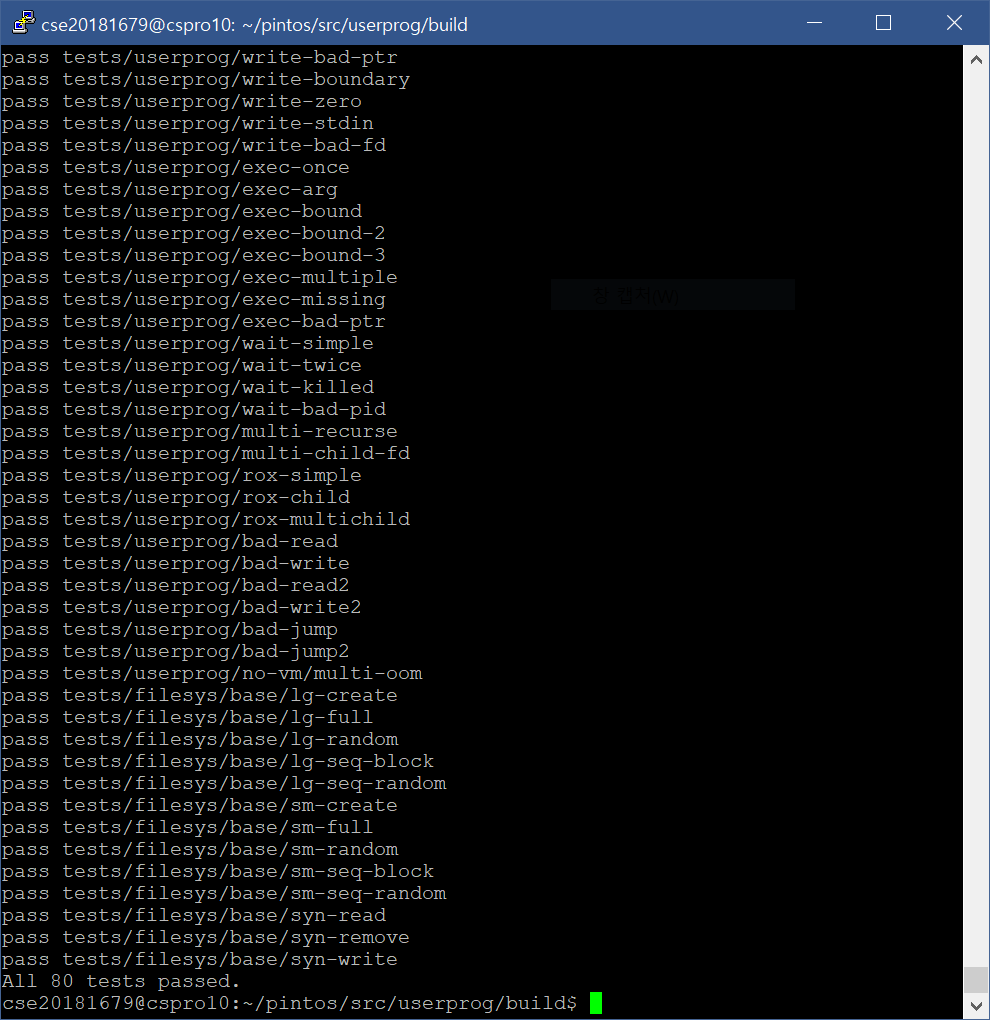


* 또, process\_execute()의 sema\_down과 start\_process()의 sema\_up이 연결되는 형식이다. child thread가 생성된 후 start\_process()가 수행되는데 그 전에 parent thread의 process\_execute()가 끝나면 parent가 child의 load 성공 유무를 알 수 없다. 즉, parent thread는 child thread의 load가 끝날 때까지 기다려야한다. 이를 구현하기 위해 process\_execute()에서 child thread를 생성한 후 child가 load될 때까지 parent를 sema\_down 시켜준다. child thread의 load가 완료되면 parent thread가 진행될 수 있도록 start\_process()에서 parent thread를 sema\_up 시켜준다.



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





* make grade

