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

담당 교수 / 분반 : 박성용 교수님 / 1반

이름 / 학번 : 김민식 / 20171609

개발 기간 : 2020. 11. 01 ~ 2020. 11. 17

1. **개발 목표**

지난 project1에서는 기본 기능을 구현하였다. 이번 project2에서는 project1의 내용을 바탕으로 system call functions(create, remove, open, close, read, write, seek, tell, filesize)를 추가하여 개발한다. 또한 file system function을 이용하여 denying writes to executable files를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
      1. **File system call Functions**

< pintos/src/userprog/syscall.c > **:** user program에서 file system을사용할 수 있도록 system call에 file system function(create, remove, open, close, read, write, seek, tell, filesize) 을 추가한다. 또한 동기화가 필요한 부분을 알아보고, 기존의 API를 바타으로 동기화를 올바르게 구현한다.

* + 1. **denying writes to executable files**

< pintos/src/userprog/syscall.c > **:** syscall.c의 open(), write() 함수를 수정하여 executable file이 수정되지 않도록 한다.

* + 1. **synchronization**Semaphore 방법을 통해 동기화를 구현한다.

< pintos/src/userprog/syscall.c > **:** syscall.c의 read, write, open함수를 수정하여 critical section을 보호해준다. 이를 위해 file을 열거나 읽고 쓸 때 synchronization 하게 해준다.  
< pintos/src/userprog/process.c > **:** process.c의 process\_wait, process\_execute, process\_exit, start\_process 함수를 수정하여 parent thread와 child thread가 정상적으로 load, exit 되도록 synchronization 하도록 한다.

* 1. **개발 내용**
     1. **Semaphore**

동기화 기법중 하나로, process의 순차적인 처리를 위해 wait(sema down),signal(sema up) 함수를 호출하면서 여러 프로세스를 의도대로 수행한다. Process 끼리 원활한 자원의 공유를 가능하게 하며 기존에 사용했던 busy waiting보다 더욱 빠르고 효율적이다.

* + 1. **File system call Functions** 각 함수별로 filesys.c의 API를 이용하여 system call을 구현한다. 또한 file descriptor의 상황에 맞게 file read/write를 구현하고, 예외 처리 또한 구현한다.

아래는 이번 project에서 구현한 file system call functions 이다.

* + - * create() : filesys\_create()을 이용하여 구현.
      * remove() : filesys\_remove()을 이용하여 구현.
      * close() : file\_close()을 이용하여 구현.
      * open() : file\_open()을 이용하여 구현.
      * filesize() : file\_length()를 이용하여 구현.
      * read() : file\_read()를 이용하여 구현.
      * write() : file\_write()를 이용하여 구현.
      * seek() : file\_seek()을 이용하여 구현.
      * tell() : file\_tell()을 이용하여 구현.
    1. **denying writes to executables**

실행중인 프로그램이 executable file이 수정되지 않도록 하기 위해 file\_deny\_write()를 사용한다. Process가 write를 하려는데, 만약 process가 실행중인 상태면 file\_deny\_write를 통해 control section에 해당하는 상황을 적용하고, lock\_acquire, lock\_release를 이용하여 file read, file writer 문제에 해당하는 mutex, progress, bounded waiting에 대한 상황을 해결한다.

* + 1. **Synchronization**

Process 끼리 통신할 때 자원을 동시에 접근하는 경우 문제가 생길 수 있다. 이를 위해 critical section을 이용한다. 만약 critical section에서 어떤 process가 execute하고 있다면 다른 process는 그 process가 critical section을 빠져나올 때까지 기다려야한다. 이를 관리하기 위한 기법이 Synchronization(동기화)이다.

동기화 방법으로 semaphor를 사용한다.

process가 제대로 load될 수 있도록 child thread들이 올바르게 종료되고 부모가 exit status를 제대로 반환할 수 있도록 부모가 자식이 완전히 끝날때까지 종료되지 않도록 구현한다.

* + - Thread/thread.h에 필요한 semaphore 변수들을 추가하고 초기화하는 내용을 추가한다. 그 후 userprog/process.c에서 semaphore 방식으로 올바른 동기화를 구현한다.
    - filesys \의 filesys.c / file.c의 다양한 API 함수들을 적절하게 사용하여 userprog/syscall.c 안에 file system call 함수들을 구현한다.
    - 동기화가 필요한 file system call ( open/close/create/remove/read/write) 함수들에 대해서는 lock\_acquire/release 함수를 통해 동기화 시켜준다. 이를 위해 syscall.c에 struct lock 변수를 추가해준다. Lock\_acquire/release를 이용하여 file read/writer 문제에 대항하는 mutex, progress, bounded waiting에 대한 상황을 해결한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2020. 11. 1. ~ 2020. 11. 4. : 매뉴얼 분석, 개선 및 구현이 필요한 부분 정리

2020. 11. 5. ~ 2020. 11. 8. : 자료구조 및 알고리즘 설계

2020. 11. 9 ~ 2020. 11. 12 : file system, denying writes to executable files 구현

2020. 11. 13. ~ 2020. 11. 15. : 테스트, 예외 처리 및 버그 수정

2020. 11. 16. ~ 2020. 11. 17. :보고서 작성

* 1. **개발 방법**
     1. **System call**

**<threads/thread.h>**

필요한 자료구조를 추가한다.

**<userprog/syscall.c>**

필요한 함수를 추가한다.

System call number에 따라 필요한 함수를 호출해 주는 부분을 추가하였으며 파일 시스템을 사용할 수 있도록 file system function들(create(), open(), close(), read(), write(), seek(), tell(), filesize(), remove())을 각각의 function에 해당하는 pintos의 filesys API들을 이용하여 구현한다.



* + 1. **Synchronization**

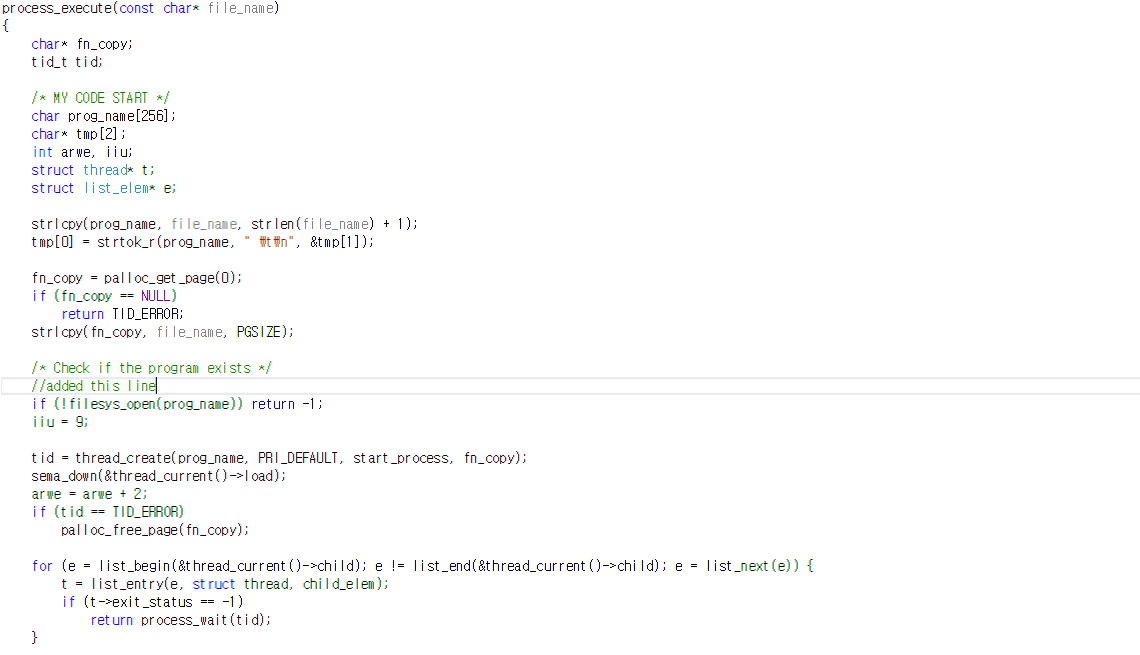
**<threads/thread.h>**

Struct semaphore 변수인 load, exit, fin을 추가하였다.

**<userprog/process.c>**

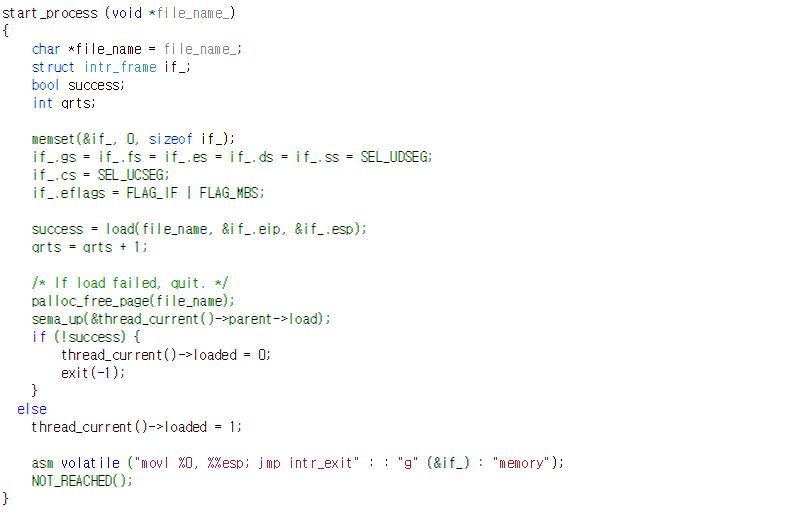
**- process\_execute()**

Thread\_create로 child thread를 생성하면 child thread의 load가 끝날때까지 sema\_down한다. 이후 child thread가 start\_process에서 load를 호출하고 load가 종료되면 sema up하여 parent thread를 깨운다. 이후 child 리스트 중에 zombie가 true인 thread가 존재한다면, 이는 제대로 exit status를 수거하지 못한 zombie process임을 의미한다. 이 thread의 exit 변수를 sema up하여 종료할 수 있게 해주고, 이 thread exit status를 반환해준다.



**- process\_start()**

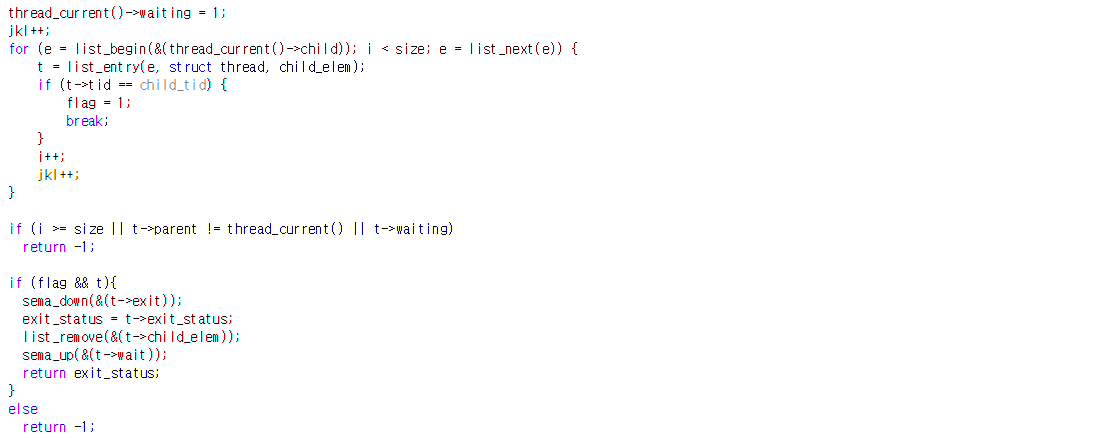
load함수 수행이 끝나고 나면, 자신의 부모의 load를 sema up 해준다. 만약 load의 결과가 false이면 자신의 zombie flag를 세팅시키고, exit(-1)을 해준다.



**- process\_wait()**

Thread의 child 중 child\_tid와 맞는 thread가 있으면 그 child thread가 종료될 때까지 현재 thread를 blocking한다.

Child thread의 exit 변수를 sema up 하며 child가 process\_exit()를 실행할 수 있게 하고, 자식의 fin을 sema up 하여 완전히 종료되고, 자신의 fin을 sema up할때까지 잠들어 있는다.



**- process\_exit()**

모든 file을 닫는다. 자식의 exit status를 저장하는 변수를 하나 생성하고, 이를 부모 child status 리스트에 추가한다.

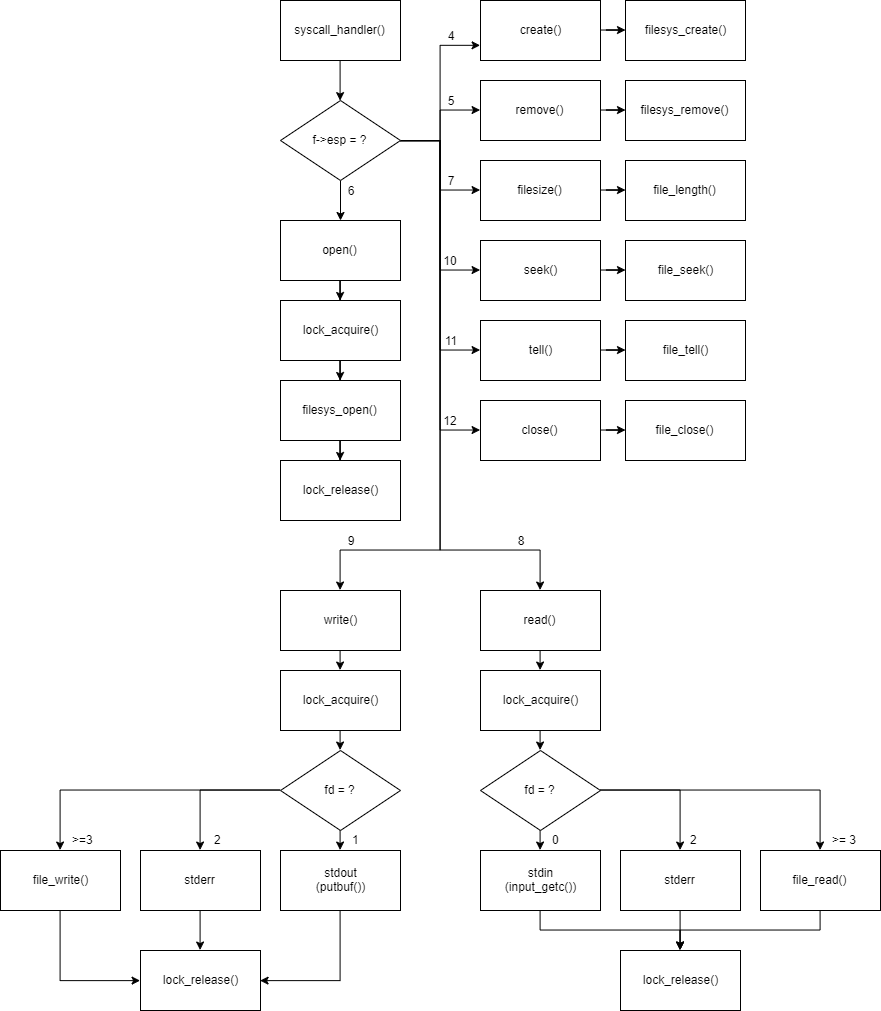
자신의 exit 변수를 sema down하여 부모가 자신을 기다려줄때까지 blocking 시킨다.



**<userprog/syscall.c>**

File의 critical section problem을 해결하기 위해 open, read, write에 lock\_acquire()과 lock\_release()를 이용하여 동기화 해준다.

1. **연구 결과**
   1. **Flow Chart**

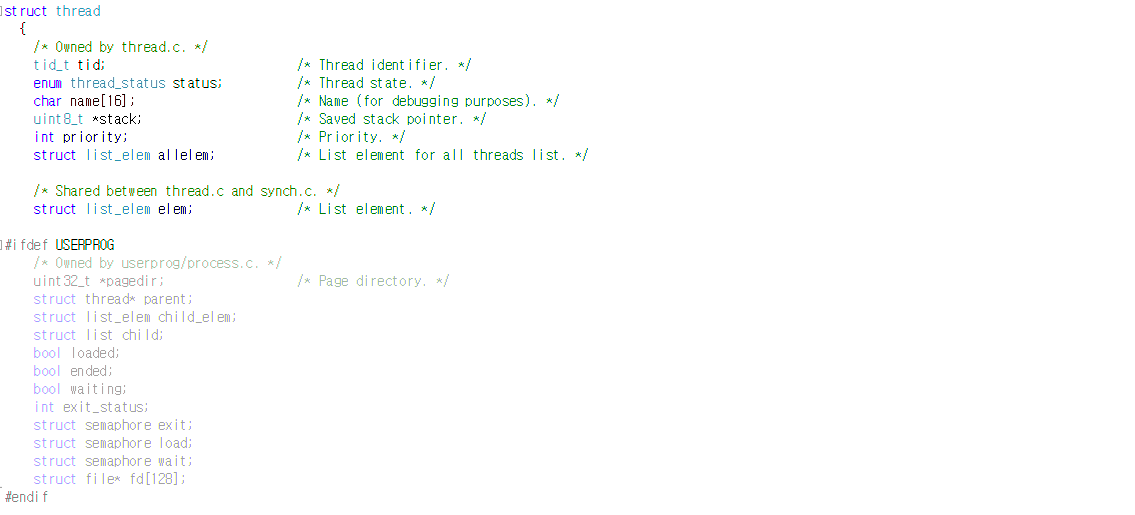
****

* 1. **제작 내용**
     1. **File system call**

**<thread/thread.h>**

Thread 구조체에 필요한 변수를 추가한다. Thread가 오픈하는 파일들의 정보를 저장하기 위한 변수를 추가해주었다.

또한 semaphore 변수도 설정해주었다. 기존에 있던 semaphore 외에 wait 과정에서 child의 메모리를 보호할 semaphore sema wait 변수도 추가해주었다.



**<thread/thread.c>**

Thread.h에서 추가한 변수들을 초기화 하는 코드를 추가해주었다.



**<userprog/syscall.c>**

File system call function 함수들은, 해당 작업을 수행하고 있는 중에, 다른 file 작업이 중간에 끼어들게 되면 결과가 의도한대로 나오지 않을 수도 있다.

따라서 lock\_acquire/release 함수를 사용하여 동기화 기법을 추가해주었다. 이를 위해 struct lock 변수인 file\_lock을 추가하였다.

또한 system call 함수들의 prototype을 바탕으로, 각각 함수마다 적절한 개수의 적절한 type의 parameter를 넘겨줄 수 있도록 했다.

* + - * Bool create (const char \*, unsigned )

입력받은 file을 입력받은 size만큼 할당하여 생성해준다. Filesys\_create API를 이용하여 구현 가능하다. 파일이름이 NULL인 경우에는 예외 처리를 해준다.



* + - * Bool remove (const char\* file)

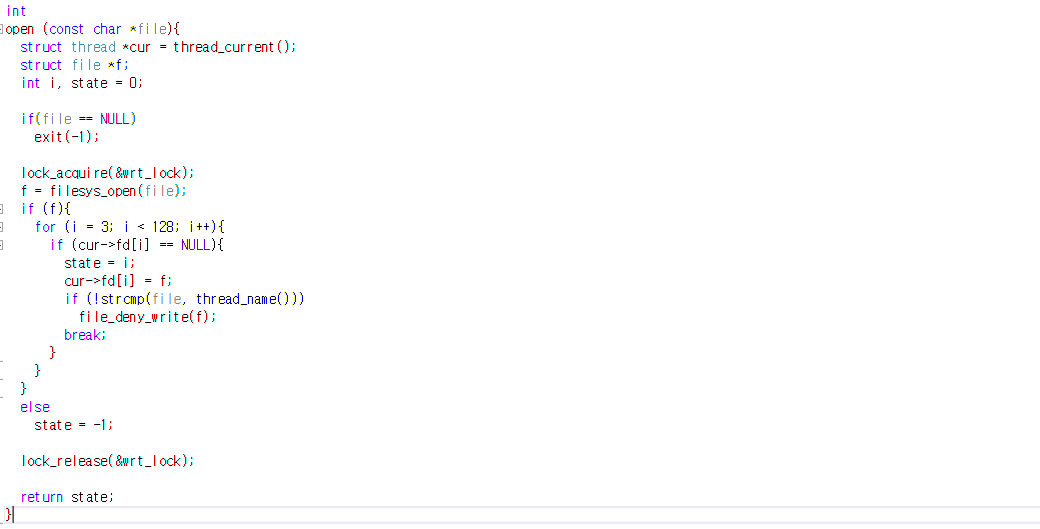
입력받은 이름을 가진 file을 삭제해준다. 그 후 삭제 결과를 return한다. File\_remove API 함수를 통해 구현하였다.



* + - * Int open(const char\* file)

File을 open 하는 system call 함수이다. Reader/writer problem을 해결하기 위해 먼저 lock을 걸어준다. 그 후 받은 파일명을 가진 file의 유효성을 확인한다.

File dexcriptor 배열에서 비어있는 곳에 open한 file의 포인터를 넘겨준다. 만약 실행중인 파일이라면 file\_deny\_wirte 함수를 통해 권한을 제한단다. 리턴할때는 락을 해제하고 return해주어 다른 thread가 critical section에 진입할 수 있게 해준다.



* + - * Void close(int fd)

File을 닫는 system call 함수이다. file descriptor를 가진 file 포인터를 close해준다. File\_clost API함수를 통해 구현하였다.

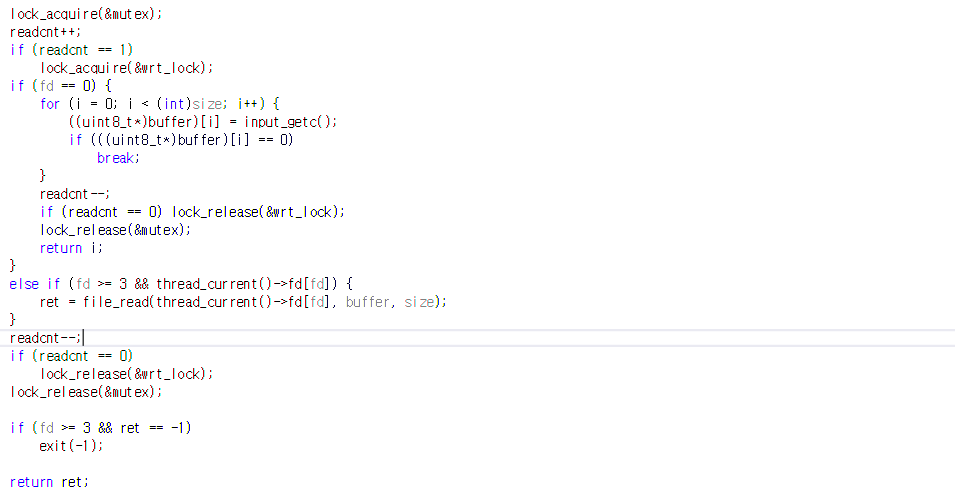


* + - * Int read(int fd, void\* buffer, unsinged size)

Fd가 0인 경우, 즉 stdout인 경우에는 키보드에서 문자를 입력받아 읽어들인다. 이는 input\_getc함수를 통해 구현 가능하다. 그 후 size를 return한다.

Fd>3인 경우, 즉 file을 읽는 경우 buffer의 주소를 체크하고 만약 잘못된 영역에 있는 경우 예외처리를 해준다. 올바른 파일의 경우 file read함수를 통해 size 만큼 파일에서 읽어온다. 그 후 size를 return한다.

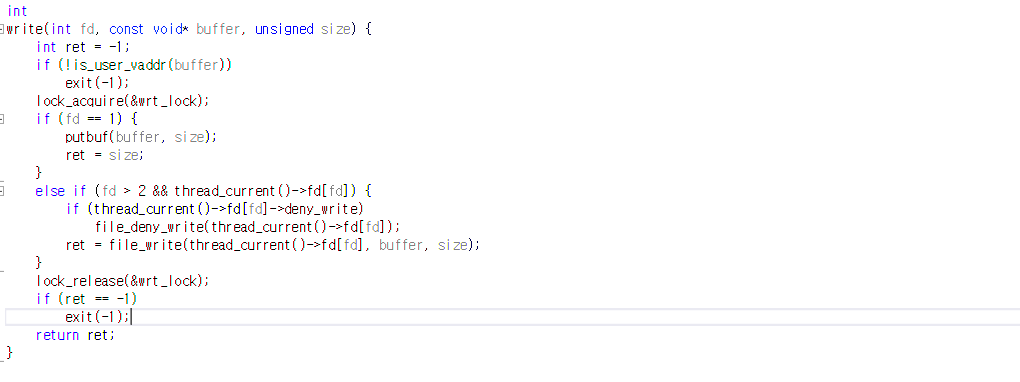
파일을 읽는 경우에도 동기화가 필요하기 때문에 lock 함수들을 이용하여 동기화를 구현하였다.



* + - * Int write(int fd, const void\* buffer, unsigned size)

size만큼 buffer에서 내용을 읽어와서 입력 받은 fd를 가지는 파일에 write하는 함수이다. stdin (fd==1)인 경우는 우선 buffer가 NULL이면 exit(-1)을 하여 예외처리를 해준다. 그렇지 않은 경우는 putbuf 함수를 통해 buffer에 있는 내용을 size만큼 읽어서 콘솔창에다 write 해준다. 그리고 size를 반환한다.

file에 write 하는 경우는, 해당 fd를 가진 파일이 없는 경우 예외처리를 해주고, 그렇지 않은 경우 file\_write 함수를 통해 파일에 올바르게 write한다. 그리고 size를 반환한다.



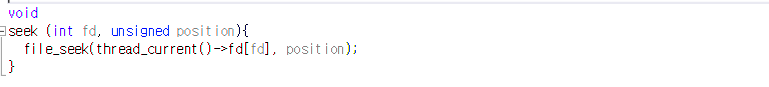
* + - * Int filesize(int)

Fd에 해당하는 파일의 size를 return 해준다.



* + - * Void seek(int fd, unsigned position)

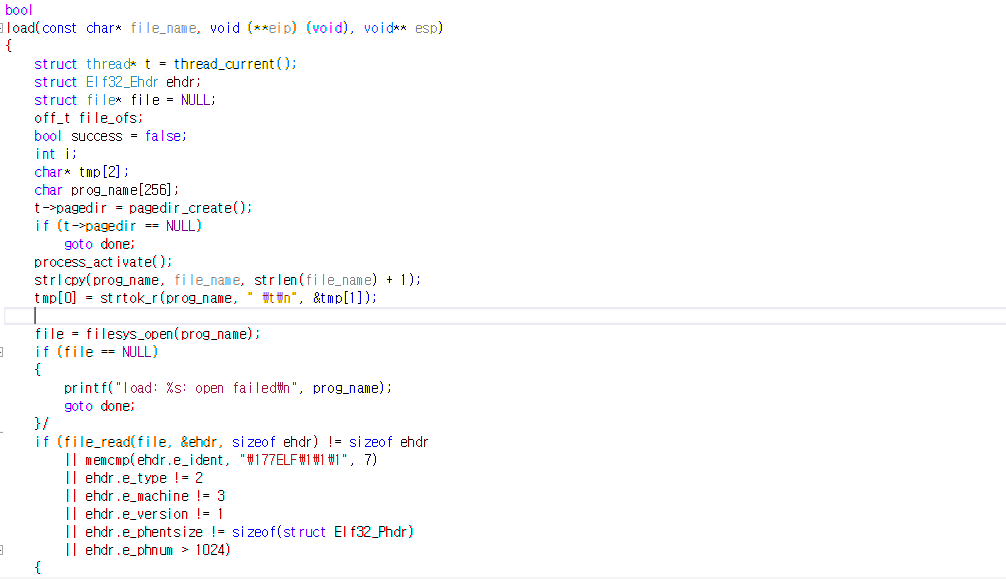
Fd에 해당하는 파일의 pos를 입력받은 position으로 설정해준다.



**<userprog/process.c>**

* + - * Bood load()

execute file을 open할 때, lock 함수들을 사용하여 동기화 처리를 해준다. 우리가 syscall.c에서 만든 open 함수가 아니라 filesys\_open() 함수를 사용한 것이므로 해당 함수 안에는 동기화 내용이 들어있지 않다. 따라서 안전하게 파일을 열기위해 위와 같은 부분을 추가해주었다.



**<userprog/exeption.c>**

page fault 조건에, “ not\_present “ 조건을 추가하여 잘못된 파일을 읽은 경우에 대해서 에러를 일으켜 테스트케이스가 의도한 결과가 나오도록 해주었다.



* + 1. **Synchronization**

**<thread/thread.h>**

필요한 semaphore 변수를 추가한다.



- load : child의 로드가 끝나기 전에 부모의 process\_execute()가 끝나지 않도록 하기 위한 변수이다. Child를 생성할 때 perent가 sema\_down을 하고, child의 로드가 다 끝나면 sema\_up을 한다.

- exit : 부모가 자신을 wait하여 exit을 기다려줄 때까지 자식의 exit을 미루기 위해 추가한 변수이다. 자식의 process\_exit()에서 자신의 exit을 sema\_down하고, 부모가 wait을 호출하면 sema\_up을 하여 자식의 exit을 진행시킨다.

- fin : process\_exit()까지 모두 수행되고, 프로세스가 완전히 끝난 상태가 되면 sema\_up이 되는 변수이다. 부모는 자식의 fin 변수가 sema\_up 될 때까지 기다린다.

**<userprog/process.c>**

* + - * Tid\_t process\_execute(const char\* file\_name)

생성된 child thread는 완전히 load될 때 까지 parent thread를 재운다.

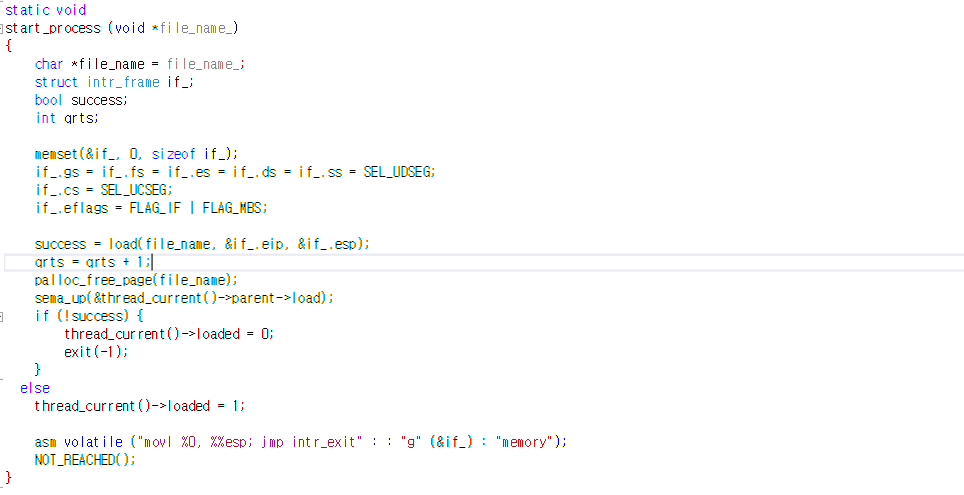
Parent의 semaphore load 변수를 sema down해준다. 이후 child의 start\_process 함수 수행 중 load가 완료되면 parent의 load는 sema up 한다.

그 후 child list 중 zombie인 thread가 있으면 그 thread의 exit를 sema up 하여 끝까지 종료될 수 있도록 한다. Zombie thread의 경우 parent가 별도로 wait 함수를 호출하기도 전에 종료되려고 하기 때문에 따로 처리해야 한다.



* + - * Static void start\_process(void \* file\_name)

load 함수 수행 후, 자신의 부모의 load를 sema\_up하여 부모가 process\_execute 함수의 나머지 부분을 수행할 수 있게 해준다. 만약 로드가 실패한 경우, 현재 thread의 zombie 변수를 true로 세팅해주고, exit(-1)을 해주어 thread가 비정상적으로 종료되었음을 알린다. 이후 해당 thread는 process\_exit에서 부모가 자신의 exit 변수를 sema\_up할 때까지 blocking 되었다가 부모가 이를 sema\_up 하면 완전히 종료된다.



* + - * Int process\_wait(tid\_t chid\_tid)

입력 받은 tid에 해당하는 child가 종료될때까지 기다려주는 함수이다.

현재 thread의 자식 리스트 중, child\_tid와 일치하는 tid의 thread가 있으면, 이의 exit 변수를 sema\_up하여 해당 thread가 종료 과정을 진행할 수 있도록 해주고, thread가 완전히 종료되었음을 알리는 변수인 fin을 sema\_down하여, child tid의 thread가 완전히 끝날 때까지 잠들어 있는다. 이후 해당 변수가 sema\_up되면 이의 exit status를 get\_child\_exit\_status() 함수를 통해 return 하고, 해당 entry를 exit\_status 리스트에서도 제거한다.

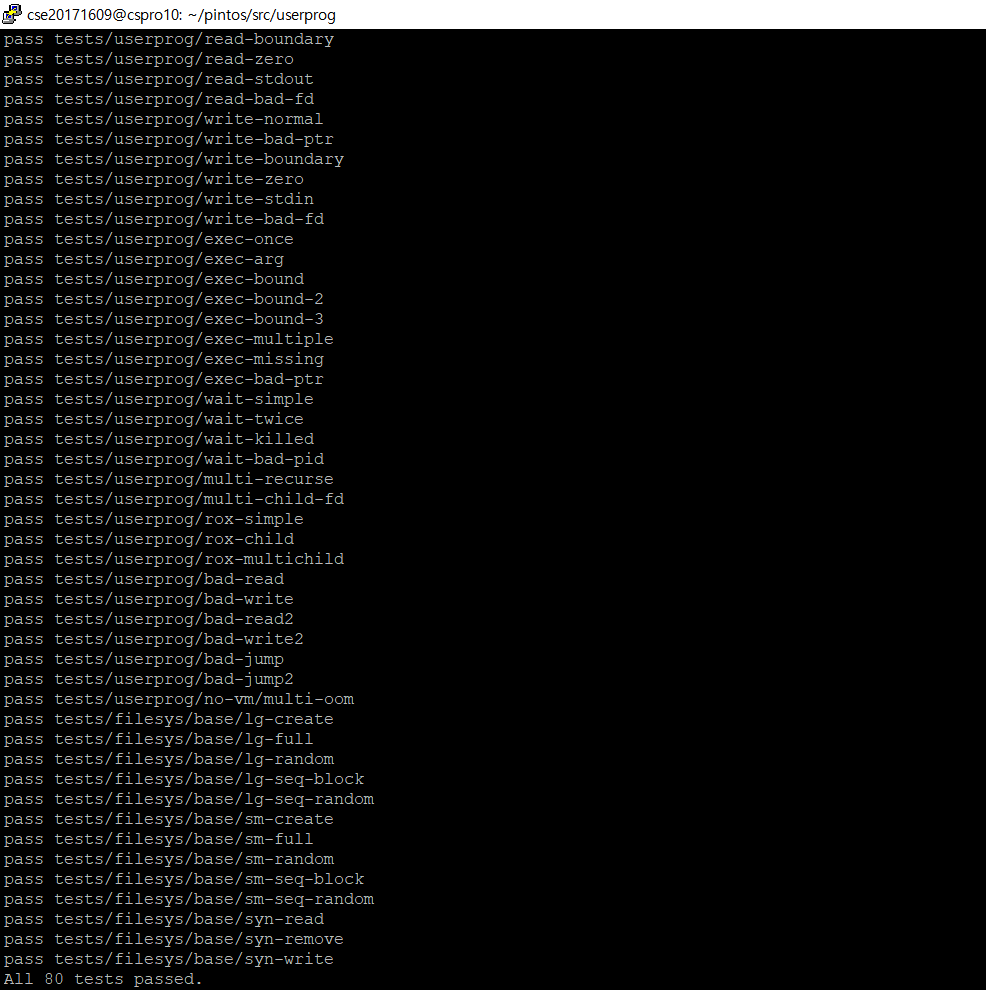


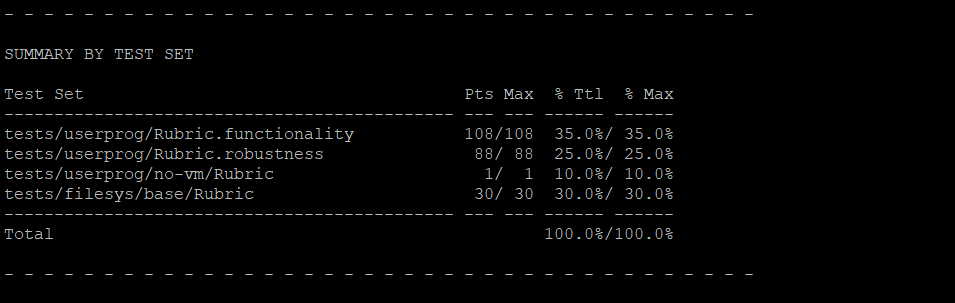
* + - * Void process\_exit()

Thread를 종료 시켜주는 함수이다. 부모가 자신을 기다려줄 때까지는 완전히 종료되지 않게 하기위해, 자신의 exit을 sema down하여 나머지 진행을 blocking한다. 이후 부모가 해당 thread를 기다리게 되면 현재 thread의 exit이 sema up되어 나머지 부분의 수행이 가능해진다



* 1. **시험 및 평가 내용**





Make check와 make grade를 한 결과 위와 같은 결과를 얻을 수 있었다.