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

담당 교수 / 분반 : 김영재

이름 / 학번 : 20161598 손동현

개발 기간 : 2020.11.2 ~ 2020.11.17

1. **개발 목표**

Pr1에서 구현하지 않은 file system 관련 system call(ex. read, write, create, remove, open, close, filesize, seek, tell)들이 전부 오류 없이 돌아가게 만들고, pr1에서 구현한 동기화 방법을 semaphore를 이용한 방법으로 변경하여 동기화 기법을 효율적으로 개선한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

**File Descriptor :**

* File descriptor란 프로세스에서 특정 파일에 접근할 때 사용하는 추상적인 값이다. File descriptor는 file system 관련 system call들을 수행하는데, file descriptor가 사용된다. File descriptor를 구현하게 되면, 프로세스가 이미 존재하는 파일을 open() 함수를 통해 열거나 create() 함수를 통해 새로운 파일을 생성해달라고 커널에 요청한 경우, 커널은 필요한 동작을 수행하고 file descriptor 값을 return하게 된다. 그리고 이러한 file descriptor는 프로세스 read(), write() 함수를 수행할 때, 인자로 사용되며, 어느 파일에 read(), write() 요청을 수행할지 구분하는 값으로 사용된다.

**System Calls :**

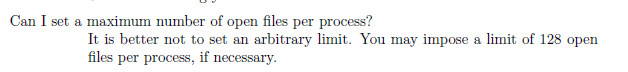
* pintos에서 요구하는 file system 관련 system call은 pr1에서 구현하지 않았다. 따라서 file system 관련 system call(ex. read, write, create, remove, open, close, filesize, seek, tell)을 구현하여, pintos에서 요구하는 file system 관련 system call을 실행할 수 있게 한다.

**Synchronization in Filesystem :**

* Multithreading 환경에서 서로 다른 thread가 critical section에 동시에 접근을 하게 되면 문제가 생기게 된다. 이를 해결하기 위해, 동기화가 필요하다. 동기화를 구현하게 되면, 위에 언급한 서로 다른 thread의 critical section 동시 접근 문제를 막을 수 있다.
  1. **개발 내용**

**File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술**

* File Descriptor 구현에는 struct file\* fd[128]; 의 struct file\* 자료형의 배열을 선택하였다. Pintos 명세서에 아래와 같이 process가 최대 128개의 file을 열 수 있다고 하였으므로, 해당 file들을 index로 관리하기 쉬운 크기 128의 정적 배열을 자료 구조로 선택하였다.



**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() :

file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys\_create 함수를 호출하여 create 한다.

open() :

file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys\_open 함수를 호출하여 open 한다. 그리고 현재 thread의 fd 배열에 filesys\_open에서 반환되는 file descriptor 값을 삽입한다.

remove() :

file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys\_remove 함수를 호출하여 remove 한다.

filesize() :

현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 size를 file\_length 함수를 호출하여 구한다.

close() :

현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file을 file\_close 함수를 호출하여 닫아준다.

seek() :

현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 파일 포인터의 커서 위치를 position 위치로 file\_seek 함수를 호출하여 옮겨준다.

tell() :

현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 파일 포인터의 커서 위치를 file\_tell 함수를 호출하여 반환한다.

read() :

파라미터로 입력 받은 fd(File Descriptor) 값을 확인하여 0이면 STDIN이므로 표준 입력에서 데이터를 읽는다. 만일 fd가 3 이상이면 현재 thread의 fd번째 file의 파일을 file\_read 함수를 호출하여 읽는다. 그리고 읽은 바이트의 수를 반환한다.

write() :

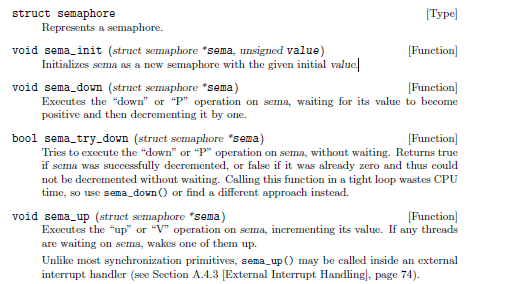
파라미터로 입력 받은 fd(File Descriptor) 값을 확인하여 1이면 STDOUT이므로 표준 출력장치에 데이터를 출력한다. 만일 fd가 3 이상이면 file\_write 함수를 호출하여 thread의 fd번째 file의 파일에 데이터를 쓴다. 그리고 쓴 바이트의 수를 반환한다.

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

프로세스가 통신을 하기 위해선 프로세스 간에 메시지를 전송하는 방법을 이용하거나, 프로세스끼리 공유하는 메모리를 통해 특정 데이터를 공유하는 방법을 이용할 수 있다. 이 때, 공유하는 메모리에 동시에 접근 하는 경우, 문제가 발생하게 된다. 특히 우리가 하는 multi-threading 환경의 경우, 여러 개의 threads 들이 공유하는 메모리는 하나의 thread만 접근할 수 있도록 제한해야 하므로 이를 위해 semaphore와 lock을 사용한다.

* Semaphore :

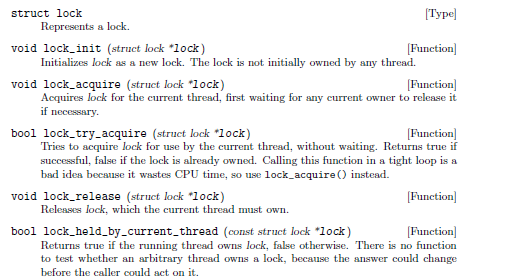
따라서 아래와 같은 semaphore 구조체와, 이에 관련된 함수들을 사용한다.



본 프로젝트에서는 sema\_init 함수를 활용하여, 처음에 thread 내부에 정의된 semaphore들을 초기화하여 준다. 그리고 sema\_down, sema\_up 함수를 process\_wait(), process\_exit(), process\_execute 함수에 적절하게 사용하여 wait()로부터 return 전에 process가 종료되는 것, child thread가 load 되기 전에 parent가 terminate 된 경우를 막는다. 자세히 어떻게 사용하는지는 아래 제작 내용에서 코드를 보고 설명한다.

* Lock :

Lock은 아래와 같은 lock 구조체와 이에 관련된 함수들을 사용한다.



Lock은 Readers-Writers Problem을 해결하기 위해 사용한다. File system에 관련된 system call 중 우선 syscall\_int 함수에서 lock을 lock\_init 함수를 이용하여 초기화 시켜준 뒤, open, read, write 함수에 lock\_acquire, lock\_release 함수를 적절하게 배치하여 Readers-Writers Problem을 해결한다. 자세히 어떻게 사용하는지는 아래 제작 내용에서 코드를 보고 설명한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 11/2 ~ 11/4 – proj2 동영상 시청 및 명세서 분석   
  11/5 ~ 11/12 – File Descriptor, System Call 구현   
  11/13 ~ 11/15 – Synchronization 문제 해결   
  11/16 ~ 11/17 – 보고서 작성
  1. **개발 방법**

**File Descriptor:**

threads/thread.h의 struct thread 구조체 : File Descriptor 구현에는 struct file\* fd[128]; 을 thread.h의 thread 구조체 내에 선언하여 각 thread가 최대 128개의 file descriptor를 갖을 수 있게 구현한다.

threads/thread.c의 init\_thread() : memset 함수를 통하여 thread 내부의 fd 배열을 전부 NULL로 초기화할 수 있게 구현한다.

**System Calls:**

userprog/syscall.c의 syscall\_handler() : Project2 요구 사항이 create, remove, open, filesize, read, write, seek, tell, close 이렇게 이므로 pr1에 추가로 해당 사항에 관한 코드들을 구현한다. 여기서 주의할 점은 accessingUserMemory 함수를 사용하여 각각의 if/else if문에 들어오게 된 경우, stack pointer가 invalid pointer가 아닌지 check한 후, 만일 invalid pointer이면 accessingUserMemory 함수 내부에서 exit(-1)이 되고, 그게 아니라면 해당 내용에 맞는 system call 함수를 실행할 수 있게 구현한다.

userprog/syscall.c의 create() : file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys\_create 함수를 호출하여 create 할 수 있게 구현한다.

userprog/syscall.c의 open() : file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys/filesys.h에 있는 filesys\_open 함수를 호출하여 open 할 수 있게 구현한다. Filesys\_open 함수의 return 값이 만일 NULL인 경우 예외처리를 해주고, 그게 아니라면, 현재 NULL로 비어 있는 File Descriptor 배열에 현재 open한 file의 포인터를 삽입할 수 있게 구현한다. 그리고 이미 실행 중인 thread에 write하는 경우를 막기 위해, filesys/file.h에 있는 file\_deny\_write 함수를 사용한다. Write 함수를 실행하기 위해서는 반드시 파일을 open 함수를 통해 열어야 하므로, 현재 thread의 이름과 file이 일치하면 file\_deny\_write를 통해 write하는 것을 방지할 수 있게 하여 준다.

userprog/syscall.c의 remove() : file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys/filesys.h에 있는 filesys\_remove 함수를 호출하여 remove 할 수 있게 구현한다.

userprog/syscall.c의 filesize() : 현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 size를 filesys/file.h에 있는 file\_length 함수를 호출하여 구할 수 있게 구현한다.

userprog/syscall.c의 close() : 현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 닫을 필요가 없으므로 exit(-1)을 한다. 그리고 현재 thread의 fd번째 file 포인터 값을 임시 file 포인터 변수에 저장한 후, 현재 thread의 fd번째 file 포인터는 NULL로 초기화한 후, 임시 file 포인터 변수를 filesys/file.h에 있는 file\_close의 매개변수로 넘겨주어 닫아줄 수 있게 구현한다.

userprog/syscall.c의 exit() :

현재 thread의 file descriptor 배열이 NULL이 아닌 경우 해당 index를 close해주어 남은 파일을 모두 close할 수 있게 해준 후, thread가 exit할 수 있게 구현하였다.

userprog/syscall.c의 seek() : 현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 파일 포인터의 커서 위치를 position 위치로 filesys/file.h에 있는 file\_seek 함수를 호출하여 옮겨줄 수 있게 구현한다.

userprog/syscall.c의 tell() : 현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 파일 포인터의 커서 위치를 filesys/file.h에 있는 file\_tell 함수를 호출하여 반환할 수 있게 구현한다.

userprog/syscall.c의 read() : 파라미터로 입력 받은 fd(File Descriptor) 값을 확인하여 0이면 STDIN이므로 표준 입력에서 데이터를 읽는다. 만일 fd가 3 이상이면 현재 thread의 fd번째 file의 파일을 체크하여 NULL인 경우를 예외처리 하여 주고, 그게 아니라면 filesys/file.h에 있는 file\_read 함수를 호출하여 읽고 읽은 바이트의 수를 반환할 수 있게 구현한다.

userprog/syscall.c의 write() : 파라미터로 입력 받은 fd(File Descriptor) 값을 확인하여 1이면 STDOUT이므로 표준 출력장치에 데이터를 출력한다. 만일 fd가 3 이상이면 현재 thread의 fd번째 file의 파일을 체크하여 NULL인 경우를 예외처리 하여 주고, 그게 아니라면 filesys/file.h에 있는 file\_write 함수를 호출하여 thread의 fd번째 file의 파일에 데이터를 쓴다. 그리고 쓴 바이트의 수를 반환한다. 단, 여기서 주의할 점은 앞서 open 함수에서 언급한 부분이다. 이미 실행 중인 thread에 write하는 경우를 막기 위해, filesys/file.h에 있는 file\_deny\_write 함수를 사용한다. Write 함수를 실행하기 위해서는 반드시 파일을 open 함수를 통해 열어야 하므로, open 함수에서 현재 thread의 이름과 file이 일치하면 file\_deny\_write를 통해 file 구조체 내의 deny\_write bool 타입 변수가 true가 된다. 따라서 현재 thread의 fd번째 file의 파일의 deny\_write 변수가 true인 경우, file\_deny\_write 함수를 호출하여, 해당 문제를 방지할 수 있게 구현한다.

**Synchronization in Filesystem:**

* Semaphore :

threads/thread.h의 struct thread 구조체 : parent thread가 child thread를 가리키기 위해 사용되는 변수와 list 구조에서 list안의 구성 요소인 list\_elem가 필요하다. 또한 parent process가 child process가 실행될 때 잠시 멈추고 있는 지시를 받기 위한 semaphore, child process가 종료되었을 때, 이를 parent process로 알리기 위한 semaphore, 그리고 parent가 child를 생성하고, 이 thread가 load되기 전에 parent가 terminate되는 경우를 막아주기 위한 locking을 위해 필요한 semaphore가 필요하다. 마지막으로 child process가 종료된 것인지 아닌지를 나타내는 status를 알려주는 변수가 필요하다.

threads/thread.c의 init\_thread() : threads/synch.h에 정의되어 있는 sema\_init 함수를 사용하여 thread 내부의 앞서 언급한 3개의 semaphore를 초기화하고 child 리스트를 초기화한 후, 현재 실행되고 있는 parent thread의 child\_thread를 list\_push\_back하여 준다.

userprog/process.c의 process\_wait() : Thread TID가 죽기를 기다리고, 그것의 exit status를 return 하게 한다. 만일 이것이 kernel에 의해 종료되었다면 -1을 return하게 한다. 반복문을 통해 현재 실행중인 current를 thread\_current()함수를 통해 찾은 뒤, 해당 thread의 child들을 하나씩 확인하고, parent thread의 child thread를 비교하며 해당 thread의 tid가 파라미터로 입력 받은 child\_tid와 같다면, 이것은 현재 실행할 child thread와 동일하다는 것을 의미하므로 앞서 thread.h에 선언한 semaphore를 이용하여 현재 thread의 parent thread를 sema\_down을 활용해 잠시 정지시킨다. 그 후, 프로그램이 진행되다가 system call인 SYS\_EXIT이 실행되면 exit\_status의 값이 -1이 들어오게 될 것이고, 그 후 현재 thread에서 사용된 list를 list\_remove 함수를 통해 삭제한다. 그리고, parent thread에게 child thread가 종료되었다는 것을 알리기 위해 sema\_up을 이용하여 알린 뒤, exit\_status(-1) 값을 return할 수 있게 구현한다.

userprog/process.c의 process\_exit() : process\_wait() 함수에서 sema\_down()을 호출하게 되면 parent process가 잠시 정지되게 된다. 그리고 child process가 진행되고 난 후, 종료될 때, process\_exit() 함수가 호출되게 되는데, 이것이 호출되기 위해서는 system call인 SYS\_EXIT가 호출될 것이다. 이 때, exit() 함수가 실행되면서 status 값을 -1로 호출하고 process\_exit 함수 내부에서는 sema\_up을 통해 child thread가 종료된 것을 알린다. 그리고 list를 삭제하기 위해 잠시 sema\_down을 하여 parent process를 정지시키고, process\_wait() 함수에서 child\_thread의 list가 제거되면 다시 sema\_up을 호출하여 완전히 종료되었음을 parent thread에게 알릴 수 있게 구현한다.

userprog/process.c의 process\_execute() : parent가 child를 생성하고, 이 thread가 load되기 전에 parent가 terminate되는 경우를 막아줘야 한다. 따라서 이를 해결하기 위해 load에 대한 locking을 thread\_create()를 하고 난 후, sema\_down을 통해 child의 수행권을 먼저 열어주게 하는 방식으로 구현한다. 또한 강제 종료된 child list가 있는지 exit\_status가 -1인지를 통해 판단하여 있는지 검사하고, 만일 있다면, 이에 process\_wait 함수를 재호출하여 올바르게 종료할 때까지 기다려준 뒤, 해당 리스트에서 제거할 수 있도록 구현한다.

userprog/process.c의 start\_process() : 앞서 언급한 것처럼, multi-oom과 같은 문제를 해결하기 위해 parent가 child를 생성하고, 이 thread가 load되기 전에 parent가 terminate되는 경우를 막아줘야 한다. 이를 해결하기 위해 앞서 process\_execute에서 sema\_down을 하였고, 현재 start\_process에서는 이에 대해 현재 thread의 parent의 해당 semaphore 변수를 sema\_up 시켜준다. 그리고, load가 실패한 경우, exit(-1)을 통해 예외처리를 하여 준다.

* Lock :

userprog/syscall.h : userprog/syscall.h에 사용할 lock 변수를 선언하여 준다.

userprog/syscall.h의 syscall\_init() : userprog/syscall.h에서 선언한 lock 변수를 threads/synch.h에 있는 lock\_init 함수를 이용하여 초기화 시켜준다.

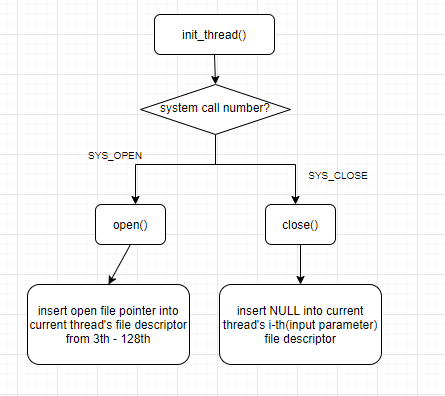
userprog/syscall.c의 open() : 파라미터로 입력 받은 file이 NULL인 예외가 아닌 경우, syn\_lock을 threads/synch.h에 있는 lock\_acquire 함수를 이용하여 현재 open을 시도하는 thread의 lock을 획득시켜준다. 정상적으로 해당 함수의 코드들이 수행되면, threads/synch.h에 있는 lock\_release 함수를 이용해 lock 해제하여 다른 thread의 critical section 진입을 허가한다.

userprog/syscall.c의 read() : 파라미터로 입력 받은 buffer가 NULL인 예외가 아닌 경우, syn\_lock을 threads/synch.h에 있는 lock\_acquire 함수를 이용하여 현재 read를 시도하는 thread의 lock을 획득시켜준다. 정상적으로 해당 함수의 코드들이 수행되거나, 읽으려는 파일 포인터가 NULL인 경우 threads/synch.h에 있는 lock\_release 함수를 이용해 lock 해제하여 다른 thread의 critical section 진입을 허가한다.

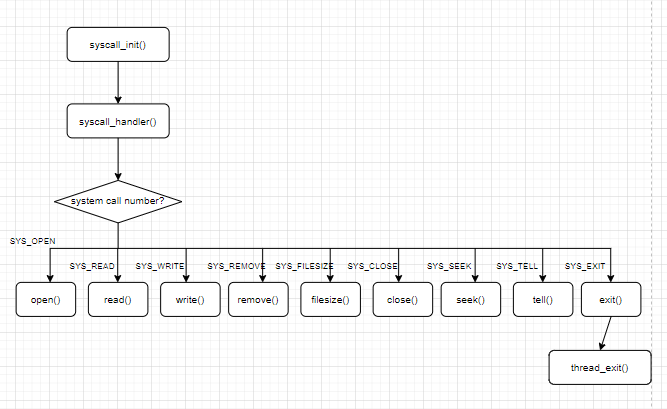
userprog/syscall.c의 write() : 파라미터로 입력 받은 buffer가 NULL인 예외가 아닌 경우, syn\_lock을 threads/synch.h에 있는 lock\_acquire 함수를 이용하여 현재 write를 시도하는 thread의 lock을 획득시켜준다. 정상적으로 해당 함수의 코드들이 수행되거나, 쓰려는 파일 포인터가 NULL인 경우 threads/synch.h에 있는 lock\_release 함수를 이용해 lock 해제하여 다른 thread의 critical section 진입을 허가한다.

1. **연구 결과**
   1. **Flow Chart**

**File Descriptor :**

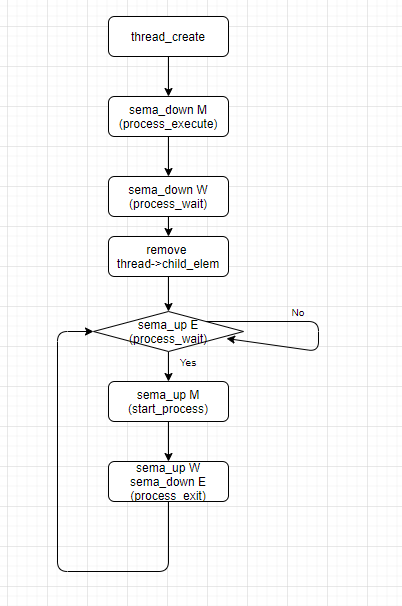


**System Calls :**

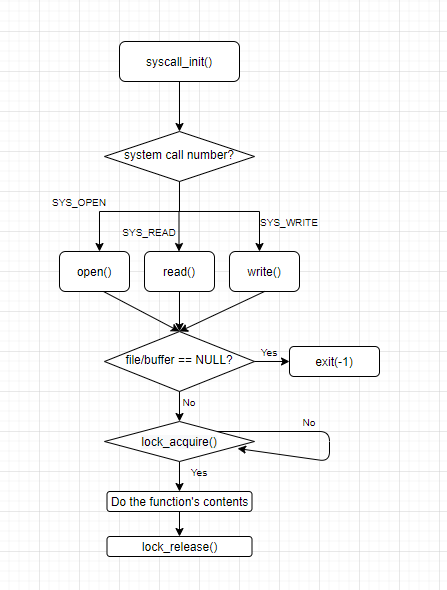


**Synchronization in Filesystem :**

* **Semaphore**



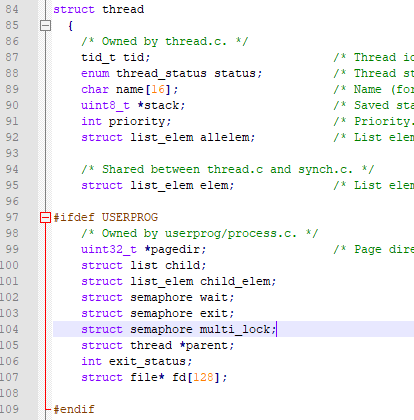
* **Lock**



* 1. **제작 내용**

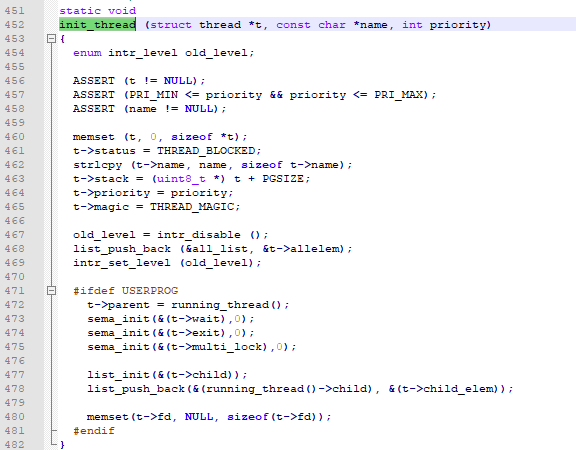
**File Descriptor:**

threads/thread.h



File Descriptor 구현에는 struct file\* fd[128]; 을 thread.h의 thread 구조체 내에 선언하여 각 thread가 최대 128개의 file descriptor를 갖을 수 있게 구현하였다.

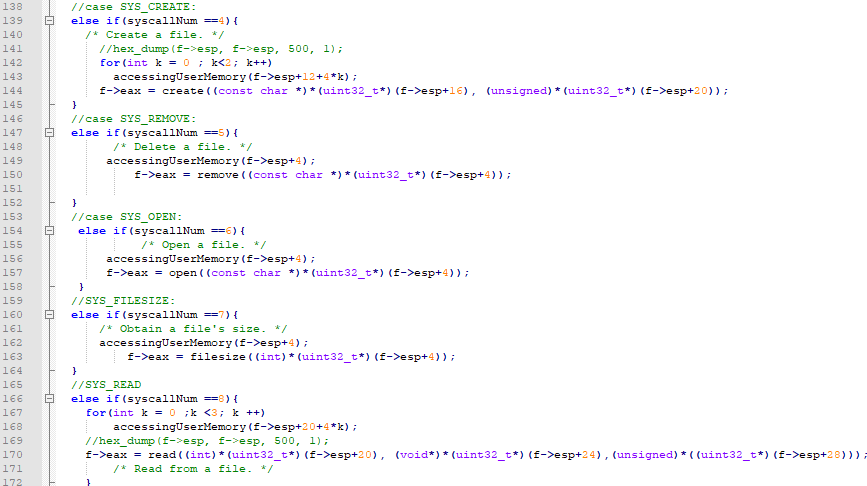
threads/thread.c

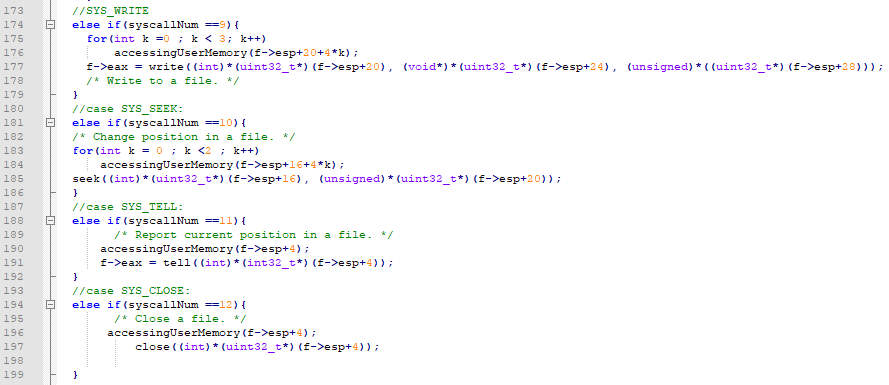


Thread를 초기화하는 init\_thread 함수에서 memset 함수를 통하여 thread 내부의 fd 배열을 전부 NULL로 초기화한다.

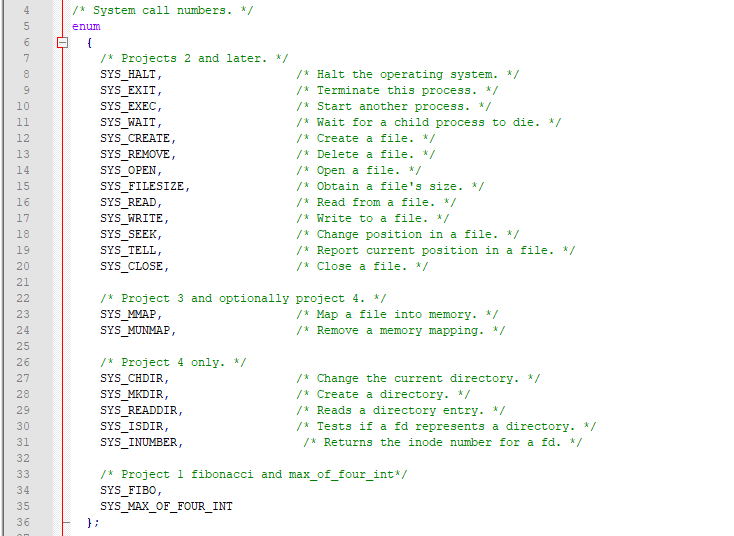
**System Calls:**

syscall\_handler() : userprog/syscall.c



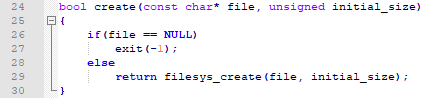


Syscall\_handler는 파라미터로 struct intr\_frame 타입 포인터 f를 받는다. 해당 파라미터의 구성요소인 esp가 lib/syscall-nr.h에 정의된 아래와 같은 syscall에 대한 int 값을 갖고 있게 된다.



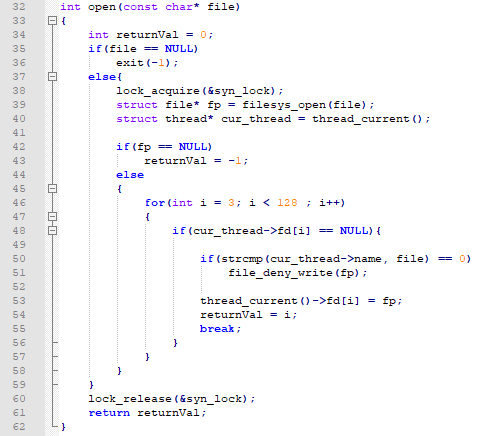
Syscall-nr.h에 정의된 system call numbers에 따라 해당 system call에 맞는 함수를 호출할 수 있도록 한다. Project2 요구 사항이 create, remove, open, filesize, read, write, seek, tell, close 이렇게 이므로 pr1에 추가로 해당 사항에 관한 코드들을 구현하였다. 여기서 주의할 점은, access User Memory를 위해 구현한 accessingUserMemory 함수를 사용하여 각각의 if/else if문에 들어오게 된 경우, stack pointer가 invalid pointer가 아닌지 check한 후, 만일 invalid pointer이면 accessingUserMemory 함수 내부에서 exit(-1)이 되고, 그게 아니라면 해당 내용에 맞는 system call 함수를 실행하게 된다.

create() : userprog/syscall.c



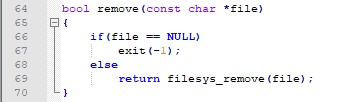
file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys\_create 함수를 호출하여 create 한다.

open() : userprog/syscall.c



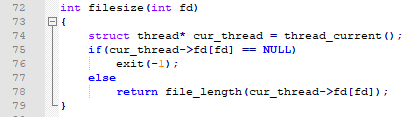
file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys/filesys.h에 있는 filesys\_open 함수를 호출하여 open 한다. Filesys\_open 함수의 return 값이 만일 NULL인 경우 예외처리를 해주고, 그게 아니라면, 현재 NULL로 비어 있는 File Descriptor 배열에 현재 open한 file의 포인터를 삽입한다. 그리고 이미 실행 중인 thread에 write하는 경우를 막기 위해, filesys/file.h에 있는 file\_deny\_write 함수를 사용한다. Write 함수를 실행하기 위해서는 반드시 파일을 open 함수를 통해 열어야 하므로, 현재 thread의 이름과 file이 일치하면 file\_deny\_write를 통해 write하는 것을 방지할 수 있게 하여 준다. Lock 관련 함수에 대한 설명은 아래 synchronization 부분 구현에서 설명하겠다.

remove() : userprog/syscall.c



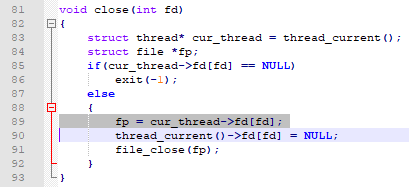
file이 NULL이면 exit(-1)을 하고, 그게 아니라면 파라미터로 입력 받은 file 이름에 해당하는 file을 filesys/filesys.h에 있는 filesys\_remove 함수를 호출하여 remove 한다.

filesize() : userprog/syscall.c



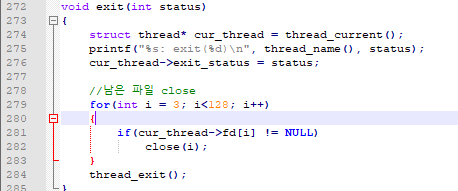
현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 size를 filesys/file.h에 있는 file\_length 함수를 호출하여 구한다.

close() : userprog/syscall.c



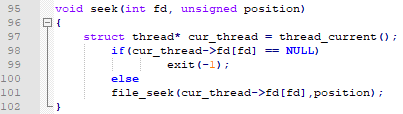
현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 닫을 필요가 없으므로 exit(-1)을 한다. 그리고 현재 thread의 fd번째 file 포인터 값을 임시 file 포인터 변수에 저장한 후, 현재 thread의 fd번째 file 포인터는 NULL로 초기화한 후, 임시 file 포인터 변수를 filesys/file.h에 있는 file\_close의 매개변수로 넘겨주어 닫아주게 한다.

exit() : userprog/syscall.c



pr1에 추가적으로 구현한 부분은 아래 반복문으로, 현재 thread의 file descriptor 배열이 NULL이 아닌 경우 해당 index를 close해주어 남은 파일을 모두 close할 수 있게 해준 후, thread가 exit할 수 있게 구현하였다.

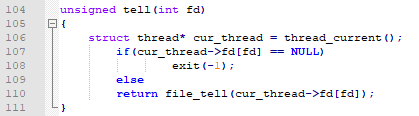
seek() : userprog/syscall.c



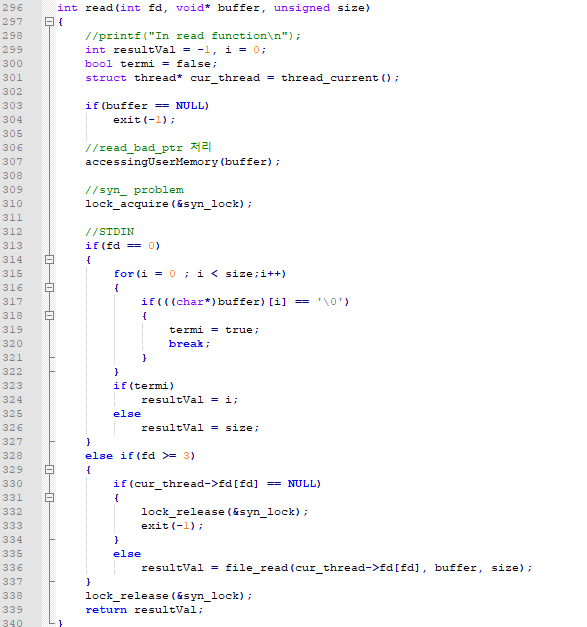
현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 파일 포인터의 커서 위치를 position 위치로 filesys/file.h에 있는 file\_seek 함수를 호출하여 옮겨준다.

tell() : userprog/syscall.c

현재 thread의 파라미터로 입력 받은 fd번째 file 포인터가 NULL이면 exit(-1)을 하고, 그게 아니라면 현재 thread의 fd번째 file의 파일 포인터의 커서 위치를 filesys/file.h에 있는 file\_tell 함수를 호출하여 반환한다.



read() : userprog/syscall.c



파라미터로 입력 받은 fd(File Descriptor) 값을 확인하여 0이면 STDIN이므로 표준 입력에서 데이터를 읽는다. 만일 fd가 3 이상이면 현재 thread의 fd번째 file의 파일을 체크하여 NULL인 경우를 예외처리 하여 주고, 그게 아니라면 filesys/file.h에 있는 file\_read 함수를 호출하여 읽는다. 그리고 읽은 바이트의 수를 반환한다. 여기서 3 이상으로 보는 이유는 0은 STDIN, 1은 STDOUT, 2는 STDERR이기 때문에 3부터 본다. Lock 관련 함수에 대한 설명은 아래 synchronization 부분 구현에서 설명하겠다.

write() : userprog/syscall.c



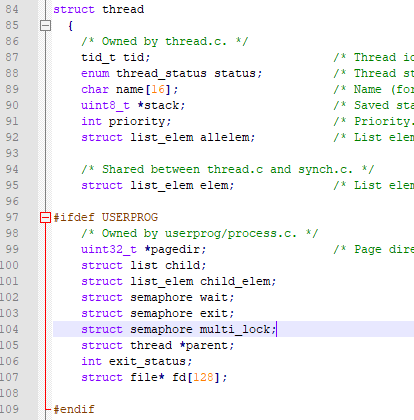
파라미터로 입력 받은 fd(File Descriptor) 값을 확인하여 1이면 STDOUT이므로 표준 출력장치에 데이터를 출력한다. 만일 fd가 3 이상이면 현재 thread의 fd번째 file의 파일을 체크하여 NULL인 경우를 예외처리 하여 주고, 그게 아니라면 filesys/file.h에 있는 file\_write 함수를 호출하여 thread의 fd번째 file의 파일에 데이터를 쓴다. 그리고 쓴 바이트의 수를 반환한다. 단, 여기서 주의할 점은 앞서 open 함수에서 언급한 부분이다. 이미 실행 중인 thread에 write하는 경우를 막기 위해, filesys/file.h에 있는 file\_deny\_write 함수를 사용한다. Write 함수를 실행하기 위해서는 반드시 파일을 open 함수를 통해 열어야 하므로, open 함수에서 현재 thread의 이름과 file이 일치하면 file\_deny\_write를 통해 file 구조체 내의 deny\_write bool 타입 변수가 true가 된다. 따라서 현재 thread의 fd번째 file의 파일의 deny\_write 변수가 true인 경우, file\_deny\_write 함수를 호출하여, 해당 문제를 방지한다. 여기서 3 이상으로 보는 이유는 read와 동일하게 0은 STDIN, 1은 STDOUT, 2는 STDERR이기 때문에 3부터 본다. Lock 관련 함수에 대한 설명은 아래 synchronization 부분 구현에서 설명하겠다.

**Synchronization in Filesystem:**

* Semaphore :

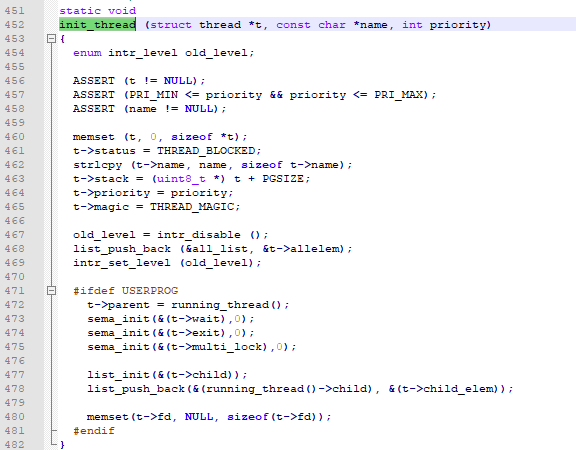
wait()로부터 return 전에 process가 종료되는 것을 막기 위해 semaphore를 사용하였다. 따라서 이에 대한 semaphore를 추가하기 위해 threads/thread.h의 thread 구조체 내부 #ifdef USERPROG #endif 사이에 아래와 같은 코드를 추가해주었다.

threads/thread.h



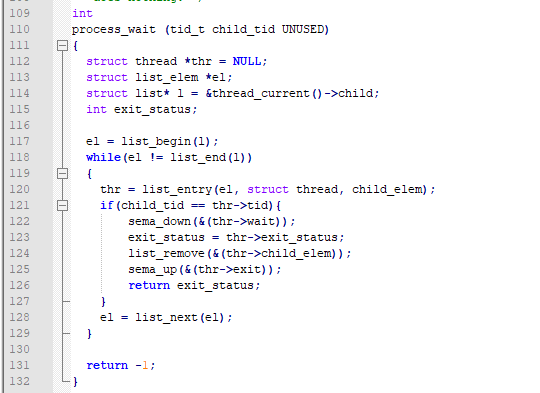
struct list child는 parent thread가 child thread를 가리키기 위해 사용되는 변수이고, project0-2에서와 마찬가지로 list 구조에서 list안의 구성 요소인 list\_elem 또한 선언해주었다. 그 후, semaphore wait와 exit을 선언하여 주었는데, semaphore wait는 parent process가 child process가 실행될 때 잠시 멈추고 있는 지시를 받기 위한 것이고, semaphore exit는 child process가 종료되었을 때, 이를 parent process로 알리기 위한 것이다. 그리고 semaphore multi\_lock은 multi-oom과 같은 문제가 생기는 것을 해결해주기 위해 추가한 변수이다. Multi-oom과 같은 test case는 fork()를 지속적으로 호출하여, child를 계속 생성하여 제한 개수를 초과할 때, 이들을 강제종료 시켜서 비정상적으로 종료된 thread의 메모리 회수가 제대로 되었는지 확인하는 test case이다. 이를 해결하기 위해 parent가 child를 생성하고, 이 thread가 load되기 전에 parent가 terminate되는 경우를 막아줘야 한다. 따라서 이를 해결하기 위해 locking을 수행할 때, 필요한 변수가 multi\_lock이다. 마지막으로 exit\_status는 child process가 종료된 것인지 아닌지를 나타내는 status를 알려주는 변수이다.

threads/thread.c



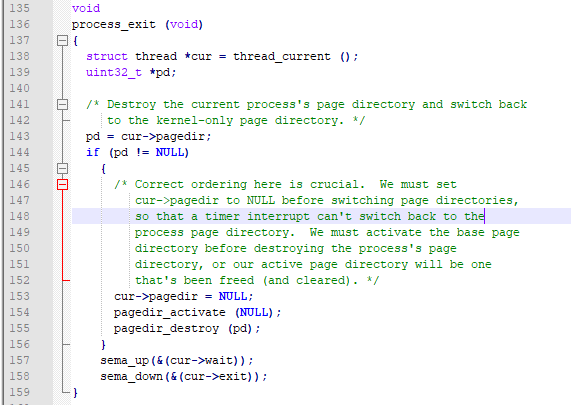
thread를 초기화하는 init\_thread 함수에서 #ifdef USERPROG #endif 부분을 추가해주었는데, 해당 부분에서는 threads/synch.h에 정의되어 있는 sema\_init 함수를 사용하여 thread 내부의 semaphore wait, exit, multi\_lock을 초기화하여 준다. 또한 child 리스트를 초기화한 후, 현재 실행되고 있는 parent thread의 child\_thread를 list\_push\_back하여 주었다.

process\_wait() : userprog/process.c



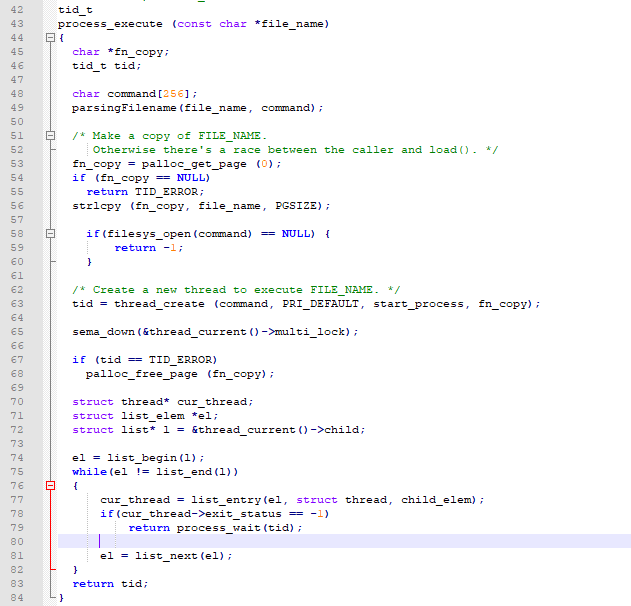
Thread TID가 죽기를 기다리고, 그것의 exit status를 return 하는 함수이다. 만일 이것이 kernel에 의해 종료되었다면 -1을 return하게 된다. 위의 while 반복문은 현재 실행중인 current를 thread\_current()함수를 통해 찾은 뒤, 해당 thread의 child들을 하나씩 확인하기 위한 반복문이다. 해당 반복문을 통해 parent thread의 child thread를 비교하며 해당 thread의 tid가 파라미터로 입력 받은 child\_tid와 같다면, 이것은 현재 실행할 child thread와 동일하다는 것을 의미하고, 따라서 앞서 thread.h에 선언한 semaphore를 이용하여 현재 thread의 parent thread를 sema\_down을 활용해 잠시 정지시킨다. 그 후, 프로그램이 진행되다가 system call인 SYS\_EXIT이 실행되면 exit\_status의 값이 -1이 들어오게 될 것이고, 그 후 현재 thread에서 사용된 list를 list\_remove 함수를 통해 삭제한다. 그리고, parent thread에게 child thread가 종료되었다는 것을 알리기 위해 sema\_up을 이용하여 알린 뒤, exit\_status(-1) 값을 return하게 된다.

process\_exit() : userprog/process.c



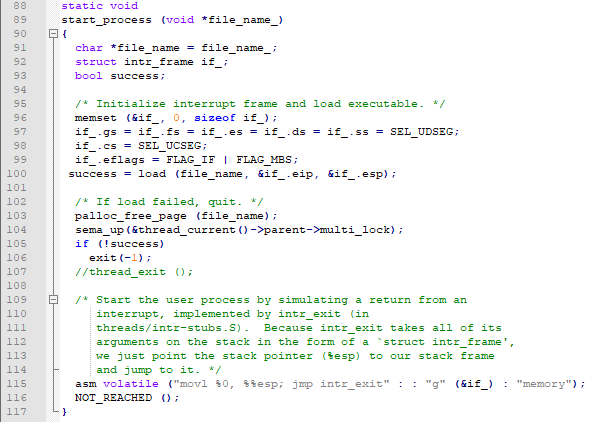
process\_wait() 함수에서 sema\_down()을 호출하게 되면 parent process가 잠시 정지되게 된다. 그리고 child process가 진행되고 난 후, 종료될 때, process\_exit() 함수가 호출되게 되는데, 이것이 호출되기 위해서는 system call인 SYS\_EXIT가 호출될 것이다. 이 때, exit() 함수가 실행되면서 status 값을 -1로 호출하고 process\_exit 함수 내부에서는 sema\_up을 통해 child thread가 종료된 것을 알린다. 그리고 list를 삭제하기 위해 잠시 sema\_down을 하여 parent process를 정지시키고, process\_wait() 함수에서 child\_thread의 list가 제거되면 다시 sema\_up을 호출하여 완전히 종료되었음을 parent thread에게 알린다.

process\_execute() : userprog/process.c



pr1에서 추가적으로 구현한 부분은 65번째 줄의 thread 구조체 내부의 multi\_lock semaphore에 관한 함수와, 70~82번째 줄까지의 반복문이다. 해당 부분은 앞서 언급한 것처럼, multi-oom과 같은 문제가 생기는 것을 해결해주기 위해 추가한 것이다. 이를 해결하기 위해 parent가 child를 생성하고, 이 thread가 load되기 전에 parent가 terminate되는 경우를 막아줘야 한다. 따라서 이를 해결하기 위해 load에 대한 locking을 thread\_create()를 하고 난 후, sema\_down을 통해 child의 수행권을 먼저 열어주게 하는 방식으로 구현하였다. 또한 강제 종료된 child list가 있는지 exit\_status가 -1인지를 통해 판단하여 있는지 검사하고, 만일 있다면, 이에 process\_wait 함수를 재호출하여 올바르게 종료할 때까지 기다려준 뒤, 해당 리스트에서 제거할 수 있도록 구현하였다.

start\_process() : userprog/process.c

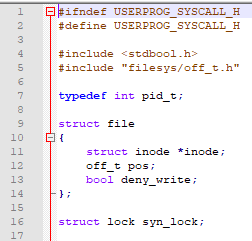


pr1에서 추가적으로 구현한 부분은 104번째 줄의 thread 구조체 내부의 multi\_lock semaphore에 관한 함수와, 106번째 줄의 exit(-1)이다. 해당 부분은 앞서 언급한 것처럼, multi-oom과 같은 문제가 생기는 것을 해결해주기 위해 추가한 것이다. 이를 해결하기 위해 parent가 child를 생성하고, 이 thread가 load되기 전에 parent가 terminate되는 경우를 막아줘야 한다. 이를 해결하기 위해 앞서 process\_execute에서 sema\_down을 하였고, 현재 start\_process에서는 이에 대해 현재 thread의 parent의 multi\_lock 변수를 sema\_up 시켜준다. 그리고, load가 실패한 경우, exit(-1)을 통해 예외처리를 하여준다.

* Lock :

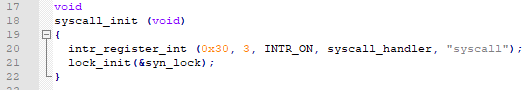
Lock은 Readers-Writers Problem을 해결하기 위해 사용한다. Readers-Writers problem이란 여러 명의 독자와 저자들이 하나의 저장 공간(버퍼)를 공유하며 이를 접근할 때 발생하는 문제로, 여러 명의 독자가 동시에 데이터를 읽어 오는 것은 가능하지만, 한 저자가 공유 공간에 데이터를 쓰고 있는 동안에는 그 저자만 접근이 가능하며, 다른 독자와 저자들은 접근할 수 없어야 한다. 현재는 한 번에 한 저자 또는 한 독자만 접근 가능하게 구현하였다.

userprog/syscall.h



Lock을 userprog/syscall.c에 있는 open, read, write 함수에서 사용할 것이기 때문에 userprog/syscall.h에 사용할 lock 변수를 선언하여 준다.

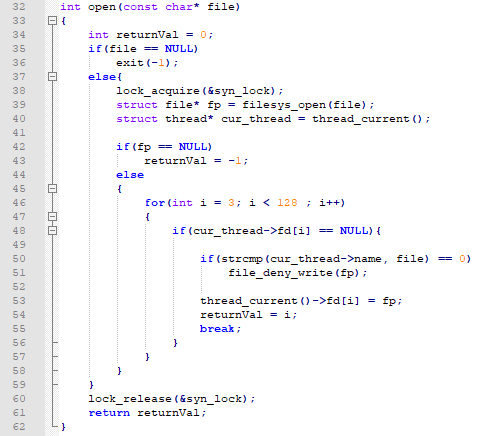
userprog/syscall.h



File system에 관련된 system call 중 syscall\_int 함수에서 userprog/syscall.h에서 선언한 lock 변수

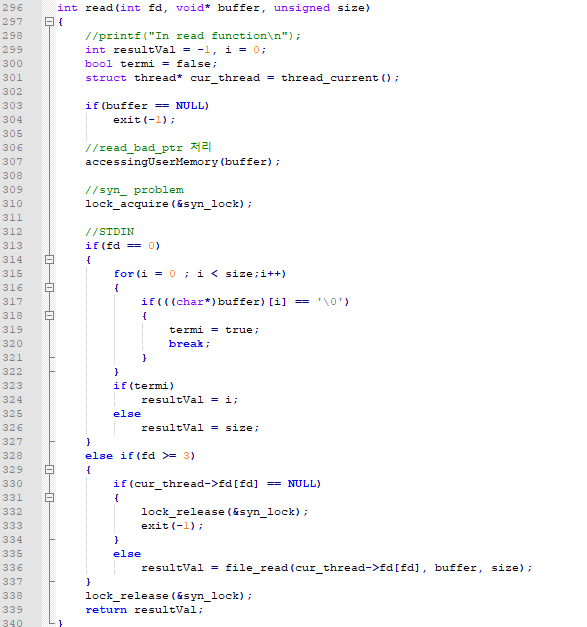
syn\_lock을 threads/synch.h에 있는 lock\_init 함수를 이용하여 초기화 시켜준다.

open() : userprog/syscall.c



파라미터로 입력 받은 file이 NULL인 예외가 아닌 경우, syn\_lock을 threads/synch.h에 있는 lock\_acquire 함수를 이용하여 현재 open을 시도하는 thread의 lock을 획득시켜준다. 정상적으로 해당 함수의 코드들이 수행되면, threads/synch.h에 있는 lock\_release 함수를 이용해 lock 해제하여 다른 thread의 critical section 진입을 허가한다.

read() : userprog/syscall.c



파라미터로 입력 받은 buffer가 NULL인 예외가 아닌 경우, syn\_lock을 threads/synch.h에 있는 lock\_acquire 함수를 이용하여 현재 read를 시도하는 thread의 lock을 획득시켜준다. 정상적으로 해당 함수의 코드들이 수행되거나, 읽으려는 파일 포인터가 NULL인 경우 threads/synch.h에 있는 lock\_release 함수를 이용해 lock 해제하여 다른 thread의 critical section 진입을 허가한다.

write() : userprog/syscall.c

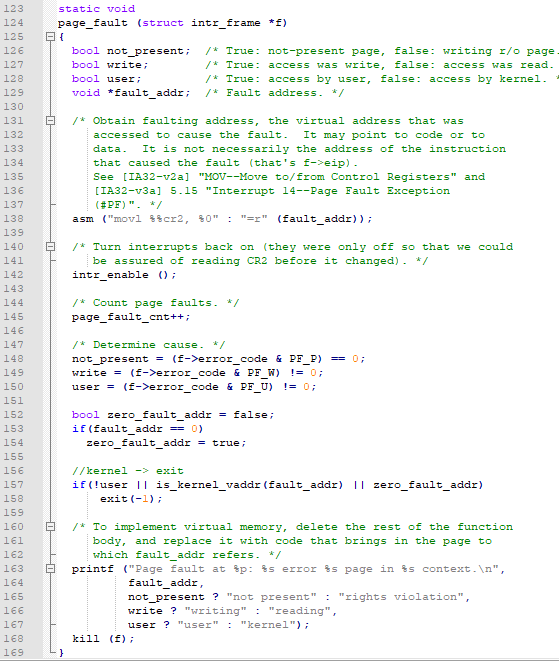


파라미터로 입력 받은 buffer가 NULL인 예외가 아닌 경우, syn\_lock을 threads/synch.h에 있는 lock\_acquire 함수를 이용하여 현재 write를 시도하는 thread의 lock을 획득시켜준다. 정상적으로 해당 함수의 코드들이 수행되거나, 쓰려는 파일 포인터가 NULL인 경우 threads/synch.h에 있는 lock\_release 함수를 이용해 lock 해제하여 다른 thread의 critical section 진입을 허가한다.

**발생한 문제 :**

**bad-read / bad-write test case 문제**

page\_fault() : userprog/exception.c



해당 문제는 기본 주소로 NULL이 들어와서 페이지 접근 에러를 유발하는 행위에 대한 것이었다. 따라서 이를 152번째 줄처럼 zero\_fault\_addr 라는 변수를 추가하여 만일 fault\_address가 0인 경우, 이를 해당 변수를 참으로 만든 후, kernel이 exit 되는 조건에 해당 zero\_fault\_addr 변수도 참이 되면 강제 종료하게 구현하였다.

* 1. **시험 및 평가 내용**
* make check 수행 결과

