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

담당 교수 / 분반 : 박성용 / 1반

이름 / 학번 : 오세정 / 20181655

개발 기간 : 2022/10/25~2022/10/26

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

**지난 프로젝트에 이어서 이번 프로젝트에서는 System Call을 추가로 구현한다. 이때 사용되는 System call 은 create, remove, open, close, filesize, read, write, seek, tell등이다.**

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

**->** file descriptor를 구현해야 하는 이유는 이번 system call에서는 file system과 관련한 system call 들을 구현해야 하기 때문에 디스크에 저장되어 있는 file에 접근하거나 새로운 파일을 생성하는 등에 작업을 위하여 file descriptor를 구현해야 한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

**->** Create, Remove, open, close, filesize, read, write, seek, tell 등을 구현해야 한다. Create call은 file을 생성하고, remove는 파일을 삭제하며, open은 해당 파일을 열고, close는 해당 파일을 닫는다. Filesize는 해당하는 파일의 크기를 구하며, read는 해당 파일에서 (혹은 STDIN)에서 값을 읽어오고, write는 해당 파일에서 (혹은 STDOUT에서) 쓰기 작업을 수행한다. Seek 은 open 된 file에서 다음에 읽거나 쓸 곳의 포인터 위치를 변경한다. Tell은 open된 파일에서 다음에 읽거나 쓸 곳의 포인터 위치를 반환한다.

3. Synchronization in Filesystem

**->** 동시에 read나 write가 일어나고 혹은 다른 thread에서 해당 파일에 write 중인데 다시 write을 하려고 하는 등의 concurrency와 관련된 문제가 발생한다. 따라서 이를 해결하기 위해서 File System에서 Synchronization이 중요하다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

**->** file descriptor 구현에는 여러 자료구조를 사용하여 구현할 수 있는데, 이번 프로젝트에서는 thread의 자료구조 안에 fd라는 file descirptor를 담을 배열을 선언하여 구현하였다. 이때 배열은 구조체포인터 배열로 설정하였으며, file에 대한 정보를 담고 있는 file 구조체의 포인터 배열로 구현하였다. 해당 pintos 프로젝트는 multi-thread 환경에서 동작하므로, 여러 thread들이 동시에 파일에 관한 system call등을 호출할 것이다. 이때 각각 thread들이 참조하고 있는 fd 배열의 값을 각 thread의 local stack에 가지고 있어야 구현이 용이하기 때문에 해당 방식을 채택하였다.

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)

**->** 구현해야 할 모든 system call은 아래와 같다.

**- create :** 인자로 들어오는 initial\_size byte만큼의 크기를 가지는 file을 하나 생성한다. 만약 성공할 시 true를 리턴하고 실패시 false를 리턴한다. 실제로 open 하는 것은 아니다.

**- remove :** 인자로 들어오는 file 이름을 가진 file을 delete한다. 만약 성공한다면 true를 리턴하고, 실패한다면 false를 리턴한다. 파일은 open되어 있거나 close 되어 있는 것과 상관 없이 remove 된다.

**- open :** 인자로 들어오는 file 이름을 가진 file을 open 한다. 음이 아닌 정수, 곧 file descriptor를 리턴한다. 만약 open 할 수 없다면 -1을 리턴한다.

**- close :** 인자로 들어오는 fd를 close한다. 프로세스가 종료되거나 exit 될 때 해당하는 프로세스의 열려있는 모든 fd를 close한다.

**- filesize :** 인자로 들어오는 fd에 대해 해당 fd가 가리키고 있는 file의 size를 byte 단위로 리턴한다.

**- read :** 인자로 들어오는 fd가 가리키는 file에서 size만큼 값을 읽어 buffer에 저장한다. 실제로 얼마나 읽었는지 byte 단위로 값을 리턴한다.

**- write :** 인자로 들어오는 fd가 가리키는 file에 size만큼 buffer에서 write 연산을 수행한다. 실제로 write 된 bytes를 리턴한다. 실제로 write된 byte는 size보다 작을 수 있다.

**- seek :** 인자로 들어온 fd가 가리키는 file의 position을 인자로 들어온 position 값으로 바꾼다. File의 시작부터 몇 byte 떨어진 곳인지를 확인한다.

**- tell :** 인자로 들어온 fd가 가리키는 file의 position을 리턴한다. Seek과 마찬가지로 file의 시작부터 몇 byte 떨어졌는지를 기준으로 리턴한다.

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

Lock\_init, Lock\_release, Lock\_acquire 등의 함수를 활용하여 Lock과 Semaphore을 적절히 이용하여 Synchronization을 구현할 수 있다. 먼저 file이 open, read, write 될 때, 시작과 끝에 lock\_acquire와 lock\_release 함수를 추가하여 해당 system call이 다른 thread의 영향을 받지 않도록 구현한다. 이때 중간에 NULL을 참조하는 등의 이유로 함수가 return될 때에도 반드시 lock을 release 해주어야 한다. 추가로 semaphore는 child process가 실행되는 동안 부모 process가 먼저 죽지 않도록 자식 프로세스가 실행될 때 sema\_up을 통해 부모가 죽지 않도록 하고, 자식 프로세스가 종료되면 sema\_down을 통해 부모 프로세스가 다시 동작할 수 있도록 구현한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

총 개발 기간 : 2022/10/25 ~ 2022/10/26

개발 일정 :

* + 1. 2022/10/25 : 전체적인 system call 들의 기능을 작성 및 file descriptor 구현 (Sync 제외)
    2. 2022/10/26 : Synchronization 구현
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정하거나 추가해야 하는 소스코드 / 자료구조 / 함수
    1. File descriptor
       - 먼저 사전에 정의된 struct file 구조체를 활용하여 각 thread 내부에 file descriptor array를 추가해야 한다.
       - 각 system call 별로 필요한 fd가 적절하게 들어가도록 isvalid\_fd() 함수를 통하여서 valid 한 fd인지를 검사한다.
       - Thread 구조체에 새롭게 추가된 내용은 init\_thread() 함수에서 빼놓지 않고 initialization 해준다.
       - 초기화 시 모든 fd (0, 1, 2를 제외하고)를 NULL로 초기화한다.
       - 아래 코드는 현재까지 구현된 thread 구조체이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + 1. System call
       - Syscall\_handler()에서 새롭게 추가되는 system call 들에 대해서 case를 분류해주고, 각 인자들에 대한 주소를 확인해 address\_protect()를 진행한다.
       - Read와 write system call의 경우 프로젝트1에서 stdin, stdout 에 대한 부분은 구현이 되어 있으므로 fd의 값을 기준으로 if 문으로 분기하여 file read와 write를 구현한다.
       - 마찬가지로 return 값이 존재하는 system call의 경우에는 f->eax에 해당하는 리턴값을 저장해준다.
       - Exec() 내부에서 filesys\_open 을 검사해 오픈할 수 없는 경우 -1을 리턴하도록 한다.
       - 텍스트이(가) 표시된 사진

         자동 생성된 설명
       - 위 사진은 syscall\_handler에서 case가 추가된 코드이다.
       - 각 system call 별 세부 구현은 아래와 같다.

**- create : file NULL 확인 / filesys\_create() 호출 / 예외처리 exit(-1)**

**- remove : file NULL 확인 / filesys\_remove() 호출 / 예외처리 exit(-1)**

**- open : file NULL 확인 / file\_open return NULL 확인 / 함수를 lock\_acquire()과 lock\_release()로 감싸기 / thread name과 file 비교 후 file\_deny\_write()**

**- close : current thread의 fd NULL 확인 / fd NULL 할당 / file\_close() 호출**

**- filesize : isvalid\_fd()로 fd 확인 / file\_length() 호출**

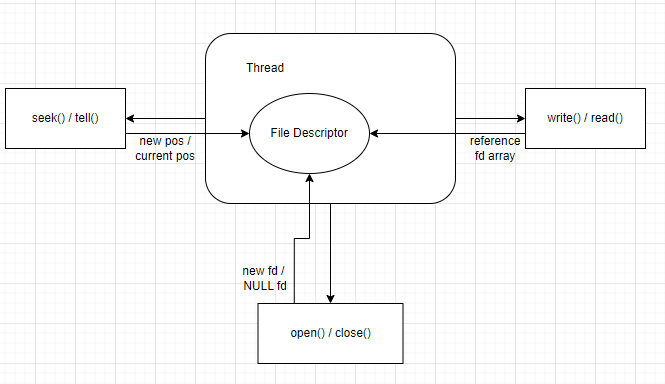
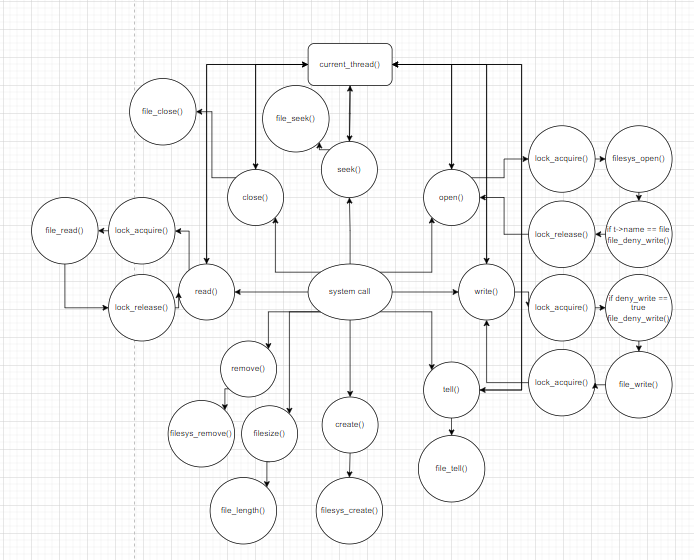
**- read : current\_thread fd NULL 확인, 예외처리 exit(-1) / 함수를 lock\_acquire()과 lock\_release()로 감싸기 / file\_read() 호출**

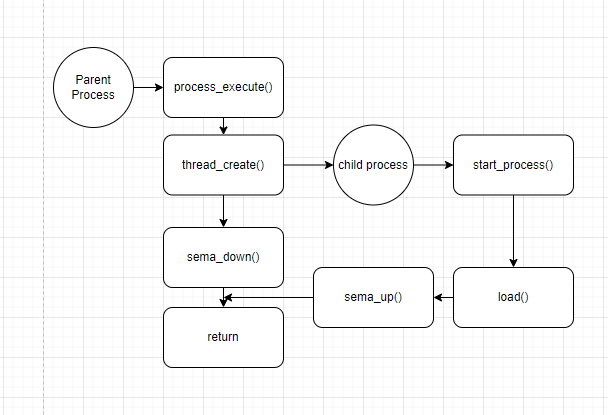
**- write : current thread의 fd NULL 확인, 예외처리 exit(-1) / 함수를 lock\_acquire()과 lock\_release()로 감싸기 / file\_write() 호출**

**- tell, seek : current\_thread 정보 가져오기 / isvalid\_fd()로 fd 확인 / file\_tell, file\_close 호출**

* + 1. Synchronization (lock, file\_deny\_write 부분은 system call에서 설명)
       - 먼저 parent process가 child process보다 먼저 terminate 되어 orphan process가 생기는 것을 막기 위해 parent process에서 child process 생성 후 semaphore로 lock을 건다. 그 후 child process 내부에서 load 함수가 종료되면 다시 parent의 semaphore를 unlock하여 orphan process가 생기지 않도록 한다.
       - 추가로 exit() 함수에서 해당 thread의 모든 child 들을 확인하면서 해당 child를 process\_wait() 함수로 기다린다. 이 방법을 통해 synchronization을 구현한다.
       - 위에 1번에서 서술한 thread 구조체 내부에 이를 위한 struct semaphore execution 이나, parent와 같은 자료형을 추가한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
* 1. File Descriptor
* ****
* 2. System call
* ****
* 3. Synchronization (lock 부분은 system call에서 표현)

****

* 1. **제작 내용**
     1. File descriptor
* 먼저 file descriptor를 구현하기 위해서는 threads/threads.h에 정의되어 있는 구조체에 fd에 관한 정보를 추가해 주어야 한다. Struct file\* fd[128]을 통해서 각 thread 들은 file descriptor 배열을 관리하게 된다. 이때 각 fd 값이 NULL을 가리키도록 initialize 함수에서 추가로 수정해준다.
* 텍스트이(가) 표시된 사진

  자동 생성된 설명
* 해당 코드는 fd가 valid 한지를 체크하는 isvalid\_fd() 함수이다.
* 
* 해당 코드는 fd의 값들을 NULL로 초기화 하는 부분이다.
* 
* Open() 에서 fd를 순회할 때, STDIN, STDOUT, STDERR를 제외하고 진행해야 하므로 i의 값을 3부터 시작한다.
  + 1. System call

각 시스템 콜의 구현은 아래와 같다. 각 system call 별 구현 이전에 syscall\_handler에서 switch-case 문에 적절한 case를 추가해주고 (ex SYS\_READ, SYS\_SEEK 등), 각 주소를 전부 프로젝트 1에서 사용했었던 address\_protect 함수로 감싸준다. 추가로 isvalid\_fd() 함수를 구현하였다. Isvalid\_fd() 함수는 넘어온 fd가 현재 thread의 fd array에 NULL인지 아닌지를 검사한다.

**- create :** file의 값이 NULL인지를 먼저 확인하고, filesys\_create라는 kernel 함수를 호출한다. 이때 만약 file이 NULL이라면 exit(-1)을 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- remove :** file의 값이 NULL인지를 먼저 확인하고, filesys\_remove라는 kernel 함수를 호출한다. 이때 만약 file이 NULL이라면 exit(-1)을 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- open :** 마찬가지로 file의 값이 NULL인지를 먼저 확인하고, open 프로세스가 수행 중간에 다른 thread에 의해 침범되는 것을 막기 위해서 lock\_acquire를 호출한다. 그 후, filesys\_open을 진행하고, 이때 만약 file\_open이 NULL이라면 lock\_release 후 -1을 리턴한다. 그리고 fd 배열에서 3번부터 128번까지 순회하며 빈 fd를 찾는다. 이때, 만약 해당 thread의 이름과 현재 실행하려고 하는 file의 이름이 같다면 file\_deny\_write을 통해 해당 file이 write 되는 것을 막는다. 그 후 해당 fd에 filesys\_open의 결과를 저장하고, lock\_release 한 이후에 system call을 종료한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- close :** 현재 thread를 thread\_current()를 통해서 받아오고, 해당 thread에 인자로 넘어온 fd 값을 활용해 해당 file descriptor의 값을 NULL로 만들어준 뒤, file\_close 함수를 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- filesize :** isvalid\_fd() 함수를 통해 인자로 넘어온 fd가 valid한 file descriptor인지를 확인하고, 만약 valid 하다면 file\_length() 함수를 호출하여 file의 length를 구하고 length를 리턴한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- read :** thread\_current() 함수를 호출하여 현재 thread를 받아오고, 마찬가지로 lock\_acquire() 함수를 통해 다른 thread들의 간섭을 막는다. 그 후, file\_read() 함수를 통해서 read system call을 수행하고, 만약 해당 file descriptor가 NULL 이라면 lock\_release를 수행한 뒤에 exit(-1)을 리턴한다. 일반적인 경우 해당 함수의 리턴값은 file\_read()의 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- write :** read() system call과 전체적인 flow가 비슷하다. 현재 thread의 정보를 받아오고, file\_write() kernel 함수를 호출하여 write system call을 수행한다. 리턴값은 file\_write()의 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- tell, seek :** 현재 thread에 대한 정보를 받아오고, isvalid\_fd()를 통해서 valid 한 fd인지 확인한 뒤에, file\_tell() / file\_seek() kernel 함수를 호출하여 tell / seek system call을 수행한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + 1. Synchronization

Synchronization은 크게 lock과 semaphore로 구현된다. Lock의 경우는 syscall\_init() 함수에서 lock\_init()을 통해 먼저 initialization 되고, open, read, write 3개 함수에서 lock\_acquire() <-> lock\_release()로 해당 system call이 다른 thread에 의해 간섭받지 않도록 구현한다. 아래 두 사진은 syscall\_init과 read 함수의 구현부로, lock\_init과 lock\_acquire() <-> lock\_release() 로 블록이 쌓여져 있는 것을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

Semaphore의 경우는 부모 프로세스와 자식 프로세스간 orphan process가 발생하지 않도록 구현하는데 사용한다. 해당 정보를 init\_thread()에서도 추가로 초기화 하는 과정을 거친다. 이때 자식 프로세스를 만드는 process\_execute 내부의 thread\_create() 함수에서 thread\_create() 이후에 sema\_down을 통해 해당 thread에 semaphore lock을 건다. 그 후 thread가 실행하는 start\_process 함수에서 load 이후에 sema\_up을 추가하여 부모 프로세스의 semaphore lock을 해제한다. 아래는 process\_execute 함수와 start\_process 함수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**

텍스트이(가) 표시된 사진

자동 생성된 설명