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

담당 교수 / 분반 : 김영재

이름 / 학번 : 20161614 원성현

개발 기간 : 21.10.04~10.15

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트의 목표는 파일시스템과 관련된 system call ( rw, open, close 등)을 이식하고, 나아가 reader-writer problem 등을 synchronization을 통해 해결하는 것이다.

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 Descriptor는 pintOS 뿐만 아니라 여러 OS에서 파일시스템을 이루는 가장 기본요소이다. FD의 0과 1은 stdin, stdout으로 나와있고, 2는 stderr로 pre-reserved되어있으므로, 우리는 FD의 array를 만들고, 3번 fd부터는 우리가 사용할 수 있는 일반 파일에 대한 정보를 담게 된다.

2. (이번 프로젝트에서 구현해야 하는) System Calls  
이번 프로젝트에서는 첫째로 프로젝트1에서 구현했던 read/write를 확장해야 한다. 프로젝트 첫번째에서는 stdin,stdout에 대해서만 처리했지만 본격적인 file system에서는 fd>2인 경우에 대해 파일을 읽고 쓰는 것을 지원해야 한다.  
그리고 명세서에서 요구하는 다음 System Call들이 존재한다.  
Seek : 원하는 position으로 파일을 탐색한다. Create : 파일을 만든다. Open : 파일을 연다. Close : 파일을 닫는다. Remove : 파일을 삭제한다. FileSize : 파일의 크기를 계산한다. Tell : 파일의 시작으로부터 현재 offset의 값을 계산한다.

3. Synchronization in Filesystem  
여러 개의 thread가 실행된다면, OS/SP 강의에서 이야기했던 것 처럼 synchronization 문제가 발생한다. 대표적으로 Read & Write Problem인데, 어떤 스레드가 파일을 읽고 있을때 다른 스레드가 파일을 쓰는 것과 같은 행위이다. 따라서 우리는 스레드 간 파일접근의 엄격한 분리를 해주어야 한다. 이것은 process 실행시 semaphore을 이용하고, 또 한 process 안에서 추가로 read/write 등의 함수에서 semaphore을 이용하는 등의 방법으로 해결할 수 있다. 이러한 방식으로 파일 시스템에서 synchronization 문제가 해결된다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술  
  
File Descriptor을 구현하기 위해 file.h에서 제공하는 struct file\*의 배열을 사용했다.  
여기서 이중포인터를 이용한 동적할당이 아니라 배열을 이용한 정적할당을 한 이유는, 핀토스 매뉴얼에서 최대 128까지의 max값을 가진 fd를 만들수 있다고 나와있기 때문이고, 둘째로는 동적할당을 하면 메모리의 관리가 까다롭기 때문이다.

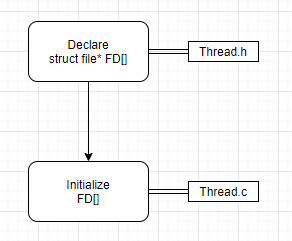
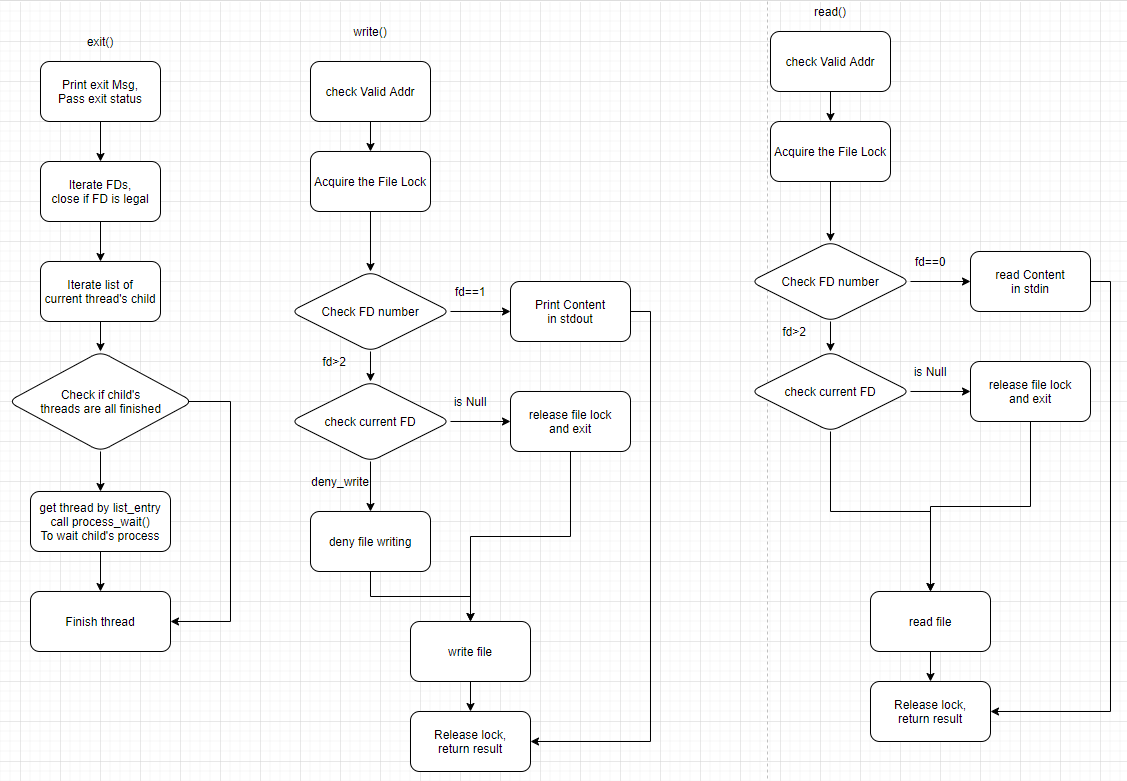
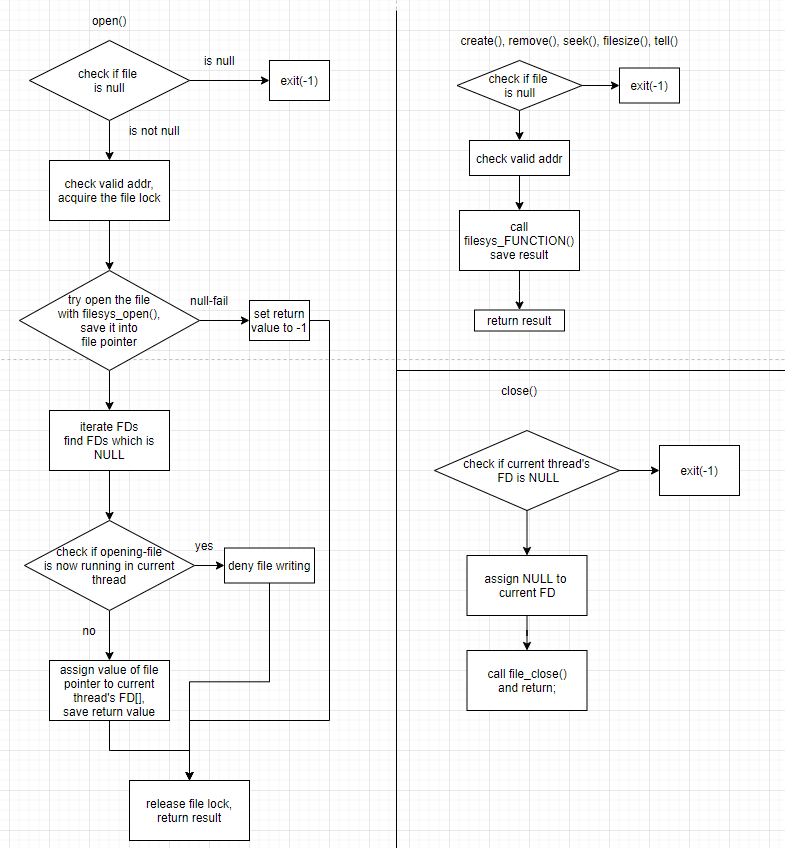
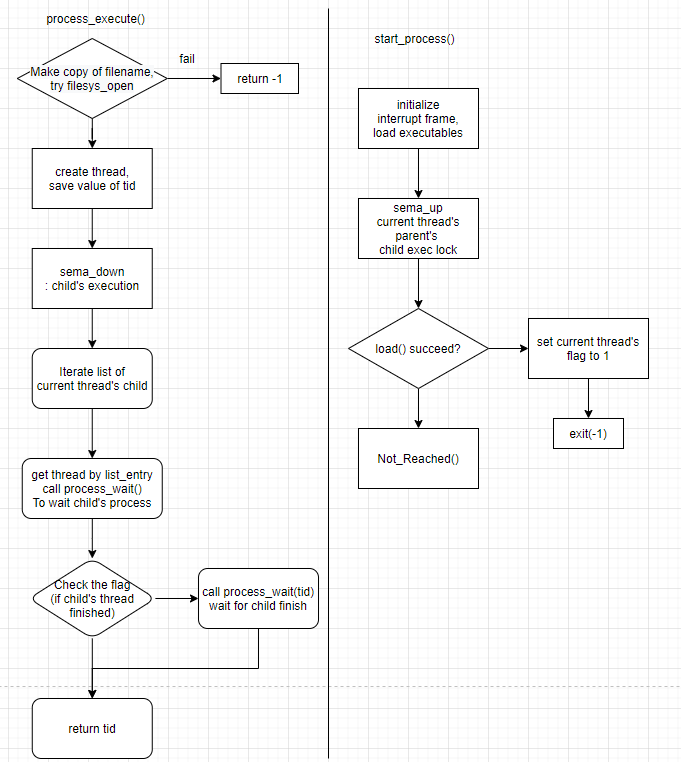
2. System Calls: 구현할 각 system call에 대해 간략히 서술 (하나의 system call 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)  
  
Write: fd>=3인 경우에 대해, FD가 Null인 경우, deny\_write인 경우에 대해 예외처리를 해주고, 예외처리에 걸리지 않는다면 file\_write를 이용해 파일에 쓴다.   
Read: fd>=3인 경우에 대해, FD가 Null인 경우에 대해 예외처리를 해주고, safe한 경우에는 file\_read를 호출해서 파일에 쓴다.  
Create: 현재 쓰레드에 대해 filesys\_create를 호출하고 그 결과를 반환한다.  
Remove: 현재 쓰레드에 대해 filesys\_remove를 호출하고 그 결과를 반환한다.  
Open : 최대 128개의 fd를 돌면서, fd가 null이며 현재 파일이 사용중이라면 쓰기를 deny한다. 만약 사용중이 아니라면 fd에 현재 파일포인터를 대입함으로서 파일을 open하게 된다.  
Filesize : 현재 쓰레드에 대해 file\_length를 호출하고 그 결과를 반환한다.  
Seek : 현재 쓰레드에 대해 file\_seek를 호출하고 그 결과를 반환한다.   
Tell : 현재 쓰레드에 대해 file\_tell를 호출하고 그 결과를 반환한다.  
Close : 현재 fd를 참조하고 NULL을 대입해서 초기화 후 file\_close를 호출한다.

3. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명 (다른 방법을 서술해도 되지만 lock과 semaphore는 반드시 포함해야 함)  
  
첫째로 file이 read와 write가 동시에 되면 file corruption이 일어나기 때문에 우리가 파일을 읽고 쓰거나 조작할 때 이것을 lock을 이용해 방지해야 한다. 따라서 이번 프로젝트에서는 syscall.c에서 struct lock형 변수인 FILELOCK을 선언하고, write(), read(), open() 의 함수를 호출할 때, fd를 참조하기 전 locking을 하고, 필요한 작업 후 함수 종료 전에 lock을 해제하는 방식으로 파일에서의 synchronization을 구현했다.  
  
lock이 아니라 semaphore의 방식을 사용하고 싶다면, struct semaphore형 변수 여러 개가 필요하다. 저번학기 SP에서 이것을 다루었는데 우리가 reader-writers problem을 해결할 때 read에게 우선권을 준다고 하자. Read() 함수를 구현하기 위해 mutex와 write를 선언하고, readCnt를 선언한다. Mutex는 1, Write는 1, Readcnt는 1로 초기화를 해주고 sema\_down(write)를 통해 read에게 우선권을 준다. 그리고 sema\_up(mutex)로 더 이상 read하지 않을 때, 즉 readCnt가 0일 때에만 write를 할 수 있게 write를 sema\_up해준다. 이 두가지가 safe하게 이루어질 수 있게 두 블록을 sema\_down(mutex), sema\_up(mutex)로 감싸준다.   
Write()함수의 경우에는 간단하다. Sema\_down(write)와 sema\_up(write)로 시작과 끝을 감싸서 safe area를 만들어주면 된다.  
  
마지막으로, semaphore는 프로세스 단위에서 sync 문제를 해결하기 위해서도 쓰인다. 기존에는 filesys가 단순했지만 이제 본격적으로 open/close등의 함수가 불리기 때문에, process간 file의 execution/load의 순서를 정하고 sync를 하기 위해서 thread에 semaphore 변수를 추가로 선언하고, start\_process(), process\_execute()의 함수에서 sema\_up(), sema\_down()을 호출해주어야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성  
  21.10.04 ~ 10.10 : 시스템 콜 구현  
  21.10.10 ~ 10.15 : 파일 synchronization 해결
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드  
    syscall.c, process.c, exception.c, thread.c, thread.h
  + 수정하거나 추가해야 하는 자료구조  
    syscall.c : 앞서 설명한 대로, 파일에 대한 lock이 필요하기 때문에 struct lock형 변수인 FILELOCK을 선언했다. 또한 코드 안에서 파일 포인터를 다루어야 하므로 struct file\* 구조체를 선언하였다.   
    thread.h : File descriptor을 구현하기 위해 thread구조체 안에 struct file\* FD[]를 선언했다. 추가로 sync 문제를 해결하기 위해 현재 스레드의 부모를 파악하고, 종료되었는지 확인하기 위해 struct thread\* parent와 int Flag를 추가로 선언했다. 마지막으로 파일을 load/exec할때 lock의 문제를 해결하기 위해 struct semaphore child\_exec\_lock을 선언했다.
  + 수정하거나 추가해야 하는 함수  
    exit() : 파일을 다룰 수 있게 수정해야 한다. 기존에는 단순히 exit status를 알려주고 종료했지만, 이제는 현재 쓰레드에서 file descriptor를 순회하며 열린 descriptor에 대해 close해준다. 그리고 스레드를 종료하기 전에, 자식보다 일찍 종료해 고아가 생기는 것을 막기 위해 현재 스레드의 자식들을 순회한다. 자식이 종료된 이후에 스레드를 종료하게 해야 한다.  
    Exec() : 현재 실행하는 cmd가 가리키는 파일이 null인 경우에 예외처리를 해주어야 한다.  
    Write() : fd>=3인 경우에 대해, FD가 Null인 경우, deny\_write인 경우에 대해 예외처리를 해주고, 예외처리에 걸리지 않는다면 file\_write를 이용해 파일에 쓴다.   
    Read() : fd>=3인 경우에 대해, FD가 Null인 경우에 대해 예외처리를 해주고, safe한 경우에는 file\_read를 호출해서 파일에 쓴다.  
    Create() : 현재 쓰레드에 대해 filesys\_create를 호출하고 그 결과를 반환한다.  
    Remove() : 현재 쓰레드에 대해 filesys\_remove를 호출하고 그 결과를 반환한다.  
    Open() : 최대 128개의 fd를 돌면서, fd가 null이며 현재 파일이 사용중이라면 쓰기를 deny한다. 만약 사용중이 아니라면 fd에 현재 파일포인터를 대입함으로서 파일을 open하게 된다.  
    Filesize() : 현재 쓰레드에 대해 file\_length를 호출하고 그 결과를 반환한다.  
    Seek() : 현재 쓰레드에 대해 file\_seek를 호출하고 그 결과를 반환한다.   
    Tell() : 현재 쓰레드에 대해 file\_tell를 호출하고 그 결과를 반환한다.  
    Close() : 현재 fd를 참조하고 NULL을 대입해서 초기화 후 file\_close를 호출한다.  
    init\_thread() : struct file\*의 array를 초기화 하고, 새로 추가한 자료구조들을 초기화 할 수 있게 수정한다.  
    process\_execute() : 쓰레드의 exec lock을 다룰 수 있게 sema\_down을 해준다.   
    그리고 exit()함수 처럼 고아가 일어나지 않게 처리를 해준다.  
    start\_process() : 쓰레드의 exec\_lock을 sema\_up해주고, 파일 로딩에 성공하지 못했다면 스레드를 종료하는 대신, 현재 스레드의 로딩이 실패했음을 알려주는 int flag를 on해주고 exit()를 호출한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성  
    
    
    
  FD: declaration, initialization  
    
    
    
    
    
    
    
    
    
  System Call: System Call functions, Revised functions  
    
  
* Synchronization: process.c  
  
  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

**FD:**File Descriptor을 구현하기 위해서 thread.h에서 struct file\* FD[128]을 선언했다.  
그리고 thread.c의 init\_thread()함수에서 fd의 array를 돌면서 NULL로 초기화해주었다. **System Call:**명세서에서 요구하는 여러가지 함수들을 구현/변형했다.  
  
  
우선 file을 lock하기 위한 struct lock FILELOCK을 선언한다.  
그리고 struct file 구조체를 선언해서 아래 함수에 file 연산을 할 수 있게 한다.  
매 함수마다 valid한 memory인지 확인하기 위해, check\_address()를 호출해서 예외처리를 하도록 한다. 추가로 이런 함수들을 구현할 때는 FILELOCK을 이용해 sync문제, 즉 reader-writer problem을 해결할 수 있게 한다.  
  
exit()의 경우 for문으로 128까지 돌면서 thread\_current()->FD[]가 null이 아니라면 close()를 호출해서 파일을 닫는다.  
그리고 for문으로 list\_begin~list\_end까지 list\_next를 이용해 순회한다.  
그리고 Thread를 list\_entry()를 호출해 가져온다. 그리고 process\_wait()을 호출해서 child의 종료를 기다린다.  
이것이 끝난 뒤에 thread\_exit()을 호출한다.  
  
write()의 경우에는 시작할 때 FILELOCK을 걸어주고, fd>=3일 때 FD[]==NULL이라면 lock을 해제하고 종료한다.   
만약 FD[]->denywrite가 true라면 file\_deny\_write()를 호출한다.  
결과를 res라는 변수에 저장하고, lock을 해제한다.  
  
read()의 경우도 write랑 흡사하다. FILELOCK을 걸어주고 fd>=3일 때 FD[]==NULL이라면 lock을 해제하고 종료한다.  
그리고 결과를 res에 저장하고 lock을 해제한다.  
  
open()은 시작할 때 lock을 걸어주고, file을 이용해 struct file\* 파일포인터를 연다. 결과값이 null이라면 -1을 리턴한다. 아니라면 for문을 이용해 모든 fd를 돌면서 현재 스레드의 이름과 파일이 동일하고(strcmp==0) 현재 fd가 null인 경우는 열지 않으므로 file\_deny\_write()를 호출한다. 현재 fd가 null이라면 fd에 파일포인터를 assign해주고 break한다. 마지막으로 lock을 해제하고 결과값을 리턴한다.  
  
close()는 현재 FD가 null이라면, 닫을 필요가 없으므로 exit(-1)합니다.  
그리고 struct file\* 파일포인터를 열어서 현재 thread의 fd를 assign한다.  
그리고 파일포인터에 fd에 null을 assign하고 파일포인터를 file\_close()을 호출해서 close한다.  
  
create, remove, seek, tell, filesize 의 경우에는 내용이 동일하다.  
file\_FUNCTION() api를 호출하고 결과를 리턴해주기만 하면된다. **Synchronization**Synchronization문제는 process.c와 thread.h/c를 수정해야 한다. 파일rw 등의 sync 문제는 앞에서 같이 서술하였다.  
Thread.h에 child가 제대로 load되었는지 확인하기 위한 struct semaphore child\_exec\_lock, parent에 대한 정보를 담고있는 struct thread\* parent, 그리고 스레드가 load되었는지 확인하기 위한 int Flag를 선언한다.  
Thread.c에서는 init\_thread()에서 sema\_init으로 child\_exec\_lock 을 초기화하고, parent는 running\_thread()로 초기화 한다. Flag역시 0으로 초기화해준다.

이제 process.c에서 sync문제를 해결해야 한다.  
Start\_process()에서는 load()를 호출한 다음, 제대로 끝났다면 부모 proces를 sema\_up한다. 즉 후술할 execute()함수에서 lock된 부모 process를 다시 실행하게 해준다. Load가 제대로 되지 않았다면(!success) flag를 1로 set하고 exit한다.  
그리고 이 상황을 process\_execute() 함수에서 처리하게 된다.   
process\_execute()함수에서 thread\_current()의 child\_exec\_lock을 sema\_down의 인자로 호출해준다. 이렇게 parent가 child보다 먼저 죽지 않게 waiting을 시켜주고, struct list\_elem\* 변수인 E와 struct thread\* 변수인 T를 이용해서 thread\_current()->child의 list를 list\_begin(), list\_end(), list\_next()등의 함수를 이용해서 순회한다. 그리고 T->Flag가 1이라면 앞서 start\_process()에서 비정상적으로 load된 child process이므로 이것을 process\_wait를 이용해 처리한다.

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