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

담당 교수 / 분반 : 이혁준 / 2반

이름 / 학번 : 고성빈 / 20161563

개발 기간 : 2020.10.31 ~ 2020.11.17

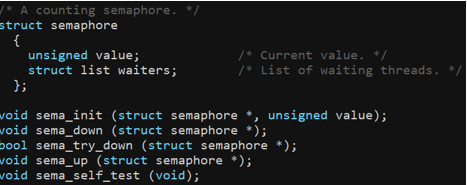
1. **개발 목표**

* File Descriptor를 구현하고 관리하며 file open/read/write를 수행하도록 한다.
* system call의 file system에 대한 내용을 보완한다.
* File system 구현 시 synchronization을 만족하도록 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. File Descriptor
   * File Descriptor를 구현 시 총 128개를 생성해 0은 stdin, 1은 stdout, 2는 stderr으로 설정하고, 3~127까지는 각 파일에 대한 입출력을 처리할 수 있도록 한다. 이를 위해 file open, close시의 처리는 물론 file이 열려있는 채로 exit되는 경우 등에 대해서도 처리해주어야 한다.
3. System Calls
   * Project 1에서 구현해주지 않은 추가적인 system call을 구현해야 한다. create, remove, open, filesize, seek, tell, close를 추가로 구현해주어야 하고, 지난 프로젝트에서 일부만 구현한 read, write에 대해 구현을 마쳐주어야 한다. 그리고 기존에 구현한 함수에서도 추가적인 수정이 필요한 경우(exit 등)에는 일부 코드를 수정해주어야 한다.
4. Synchronazation in Filesystem.

* Filesystem 구현 시 파일의 정보는 critical section이기 때문에 read/write시 문제가 발생할 수 있다. 동시에 write하는 경우나 read와 write를 동시에 하는 경우에는 문제가 생길 수 있으므로 이에 대한 synchronization 처리를 해주어야 한다.
  1. **개발 내용**
* File Desriptor
  + File Descriptor는 포인터로 간단하게 정보를 저장할 수 있으므로, 각 thread마다 struct file\* fd[128]을 선언해 하나의 쓰레드가 최대 128개까지의 포인터를 저장할 수 있도록 해주었다. (stdio 와 stderr로 인해 실제 저장 가능한 포인터는 최대 125개이다)
* System Calls
  + create/remove는 filesys\_create/filesys\_remove 함수를 호출해 파일을 생성하거나 제거한다.
  + open은 filesys\_open을 통해 파일을 연 뒤, 그 file pointer를 저장해준다. 저장을 위해 thread의 128개의 file descriptor중 3~127번째의 공간을 확인한 뒤 null이 저장되어 있는 index에 file pointer를 저장해 준다.
  + filesize는 index를 입력으로 받아, file descriptor중 index 번째의 파일에 대해 file\_length 함수를 통해 파일의 길이를 출력해준다.
  + seek는 index와 position을 인자로 받아 file descriptor중 index 번째의 파일에 대해 pointer가 가리키는 위치를 position으로 바꾸어준다.
  + tell은 file\_tell 함수를 호출해 index 번째 file pointer가 가리키는 위치(읽거나 쓰는 위치)를 반환해준다.
  + close는 index 번째 파일이 열려있을 경우 닫아준다.
  + read(수정)는 기존에는 index가 0일 경우에 stdin을 통해 입력을 받아주었는데, index가 3 이상인 경우에 대해서는 추가적으로 해당 index 번째의 파일로부터 데이터를 읽어주어야 한다. 이를 위해 file\_read 함수를 사용한다.
  + write(수정)는 기존에는 index가 1일 경우에 stdout을 통해 출력해주었는데, index가 3 이상인 경우에 대해서는 추가적으로 해당 index 번째의 파일에 데이터를 출력해주어야 한다. 이를 위해 file\_write 함수를 사용한다.
* Synchronization in Filesystem
  + file system 구현 시 lock이나 semaphore를 사용해 critical section에 동시에 접근하는 문제 상황을 막아주어야 한다. 현재 상황에서는 key가 최대 1개이므로 lock을 사용하든 semaphore를 사용하든 구현상에 문제는 발생하지 않는다. cost efficiency를 위해서 구현이 편하고, 구현하며 에러 발생이 적은 방식인 lock을 사용해 구현했다.





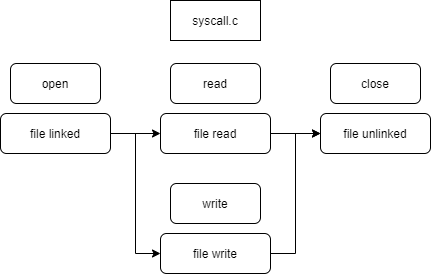
* + filesys\_lock을 사용해 write, 첫번째 open/read시에 lock을 acquire한다. 이미 open/read 중인 상황에서는 추가적으로 lock acquire 없이 파일에 접근할 수 있도록 해준다.
  + 이를 위해서는 현재 파일을 읽고 있는 thread의 갯수가 필요하고 이는 전역변수 readCount에 저장이 된다. 이 readCount 역시 critical section이므로 이에 접근하기 전에 mutex\_lock을 acquire해 두 thread가 readCount에서 충돌하지 않도록 해준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

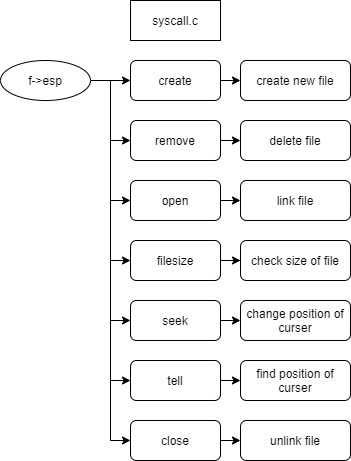
* 이번 프로젝트는 지난 프로젝트와는 다르게 추가적으로 수업시간에 내용을 배우지 않아도 모두 구현할 수 있기 때문에 빠르게 구현을 시작했다.
* 1차적으로 11/4~11/5일에 프로젝트 영상을 보며 구현을 시작했고, 기본 file descriptor 구현과 system call 추가 구현을 했다. 다만 인자로 받는 각 데이터가 null인지 확인을 해주지 않아 대부분의 bad 테스트를 통과하지 못했다.
* 11/7~11/8일 주말동안 구현을 하려 했으나 서버가 8일 오전까지 닫혀있어 8일에만 구현을 할 수 있었다. 다시 시작해보며 실패한 테스트 케이스를 확인해 null처리를 추가적으로 해주었고, synchronization 처리를 마쳤다. 이후 11/14일에 보고서 작성을 마쳤다.
  1. **개발 방법**
* 수정이 필요한 부분
  + thread.h의 struct thread에 file descriptor 배열을 추가해주어야 한다.
  + 기존에 구현한 read, write를 stdio 외에도 file input/output이 가능하도록 수정해주어야 한다.
  + exit시에 모든 file descriptor을 확인해 close 해주도록 해주어야 한다.
* 추가해야 하는 부분
  + thread 생성 시 file descriptor를 초기화해주어야 한다.
  + system call에 create, remove, open, filesize, seek, tell, close에 대해서 추가해주어야 한다.
  + open, read, write간의 synchronization을 위한 lock 또는 semaphore를 추가해주어야 한다.

1. **연구 결과**
   1. **Flow Chart**

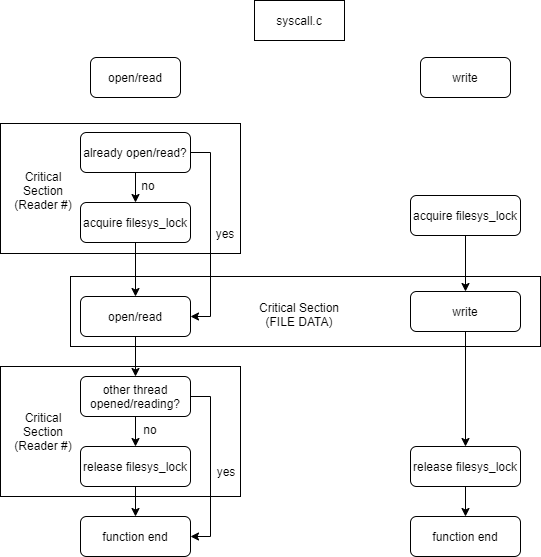
* About File Descriptor

****

* About System Calls

****

* About Synchonization



Reader Number의 critical section 접근 관리를 위해서 mutex\_lock을 사용해 Reader Number에 최대 하나의 thread만 접근할 수 있도록 해주었다.

* 1. **제작 내용**

1. File Descriptor
   * + thread.h 내부의 struct thread에 인자를 추가해주었다. struct file\* fd[128]을 선언해 file descriptor를 저장하고 있는 pointer를 128개까지 저장할 수 있도록 해 주었다. 실제로는 3~127까지 125개만 저장이 된다.
     + thread.c의 init\_thread 함수의 호출 시에 struct thread의 file descriptor 128개를 모두 null로 초기화해주었다.
2. System Calls
   * + syscall.c 내부에 추가 구현 함수를 추가해주었다.
     + create에서는 인자로 받은 pointer parameter가 null이 아닐 경우에 filesys\_create 함수를 사용해주었다.
     + remove에서는 인자로 받은 pointer parameter가 null이 아닐 경우에 filesys\_remove 함수를 사용해주었다.
     + open에서는 filesys\_open 함수를 호출해 파일을 제대로 열었는지 확인한 뒤, thread의 file descriptor를 확인해 file이 저장되어 있지 않은(data가 null인) 구간이 있으면 해당 칸에 연 파일의 포인터를 저장해 준다.
     + filesize에서는 해당 index 번째에 정상적인 file pointer가 저장되어 있는지 확인한 다음, file\_length 함수를 호출해 파일의 길이를 반환해준다.
     + read에서는 index가 3~127일 경우 file\_read 함수를 호출해 해당 index번째 file에서 data를 읽어온다.
     + write에서는 index가 3~127일 경우 file\_write 함수를 호출해 해당 index번째 file에 data를 작성한다.
     + seek/tell에서는 file\_seek/file\_tell 함수를 통해 file pointer가 가리키는 위치를 확인한 뒤 해당 위치를 수정/그대로 반환해준다.
     + close에서는 file\_close 함수를 통해 file descriptor가 가리키는 file을 닫아준 뒤, 해당 file descriptor 값을 null로 수정해준다.
     + 해당 함수를 처음 구현 시 null 체크를 해주지 않아서 일부 bad 테스트 케이스를 통과하지 못했으나 출력 결과와 테스트 코드를 읽은 뒤 체크하는 부분을 추가해주었다.
3. Synchronization
   * + filesys\_lock은 한개 이상의 open/read나 하나의 write만 critical section에 접근할 수 있도록 관리해주는 lock이다. 이를 위해 모든 write와 첫 open/read(open/read 중인 thread가 없는 상황에서의 open/read)는 filesys\_lock을 acquire해주어야 한다. 또한 모든 write와 마지막 open/read(open/read를 종료할 시 더 이상의 open/read가 없을 경우)는 filesys\_lock을 release해주어야 한다.
     + 이를 위해 현재 몇개의 thread가 open/read중인지 확인하는 shared data readCount가 필요하다. readCount 역시 critical section이며 전역변수로 구현해주었다. 이를 위해 open/read에서 mutex\_lock을 걸어주고, readCount를 바꾼 다음, readCount로 filesys\_lock을 acquire/release할지 정한다. 그 다음 mutex\_lock을 풀어주는 식으로 synchronization을 관리해주었다.
   1. **시험 및 평가 내용**

