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

담당 교수 / 분반 : 김영재

이름 / 학번 : 최재원 20171700

개발 기간 : 2021/10/1 ~ 2021/10/10

1. **개발 목표**

프로젝트1에서 구현한 pintos를 토대로 나머지 Pintos의 system call 구현을 완성시킴을 목표로 한다. 이미 system call 에 대한 function API는 구현이 되어 있으므로, systemcall handler에서 이를 연결시켜주는 작업을 한다. 여러 system call 중에 특히 read, write 등 file system과 관련된 system call이 이번 프로젝트의 중점이 된다. 이를 구현하는 과정에서 readers-writers problem등의 문제를 해결한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 1. File Descriptor

Pintos에서 각 파일은 file descriptor로 관리된다. File descriptor table을 사용하여 file table 그리고 inode table을 관리한다. 표준 입력(Stdin)은 keyboard file object를, 표준 출력(stdout)은 monitor file object를, 표준 에러(stderr)는 마찬가지로 monitor file object를, 그리고 I/O file object에 대한 File descriptor를 구현해야 한다. File table은 현재 쓰고 있는 포지션, 해당 파일에 대한 쓰기 권한, 그리고 inode table에 대한 포인터값을 가지고 있고, indode는 해당 파일에 대한 메타 데이터를 저장하고 있다. 결과적으로 file descriptor를 구현함으로써, 시스템 파일을 참조할 수 있고, 이를 사용하여 open, read, write와 같은 시스템 콜을 구현할 수 있을 것이다. 또한 process가 해당 file descriptor table의 정보를 직접 고칠 수 없도록 하고 반드시 kernel을 통해서 수정을 할 수 있도록 함으로써, 여러 process가 file I/O를 안전하게함과 동시에 readers-writers problem과 같은 synchronization 문제등을 오류 없이 해결할 수 있을것이다.

2. System Calls (in Project2)

현재 Pintos는 stdin에 대한 read, stdout에 대한 write만을 수행할 수 있는 상태이다. 따라서 다른 file object에 대한 I/O을 가능하게 하기 위해서 read와 write에 대한 시스템 콜을 수정해야 하며, 마찬가지로 정상적인 file I/O를 구현하기 위해 create, open, close, tell, remove, filesize, seek에 대한 시스템 콜 처리를 system call handler에서 작성한다. 이 과정에서 file descriptor를 사용할 수 있을 것이고, 또한 위에서 언급한 동기화 문제또한 이를 구현하는 과정에서 해결해야 할 것이다.

3. Synchronization in Filesystem

우선 pintos에서는 현재 running program에 대한 executable file을 지우는 것을 허용하지 않는다. 따라서 file open을 할 때 동일 file descriptor에 대한 덮어쓰기를 금지해줘야 한다. ( file descriptor가 close에 의해 initialize되기 전까지 동일 file descriptor 사용 불가.) 이 때 우리는 명세서에 언급된 file\_deny\_write 등을 사용한다. 다음으로 readers writers problem이 있다. 동일한 file에 대해 서로 다른 thread가 각각 read와 write를 수행하려는 경우 또는 둘다 write를 수행하려는 상황에 대비하여 synchronization이 필요하다. 이 과정에서 pintos에서 제공하는 semaphore 또는 lock을 사용 가능하다. 최종적으로 filesystem에 대한 synchronization을 구현함으로써 안전하게 여러 thread에 대한 File I/O이 수행될 것이다.

* 1. **개발 내용**

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

File descriptor를 구현하기 위해서 linked list와 배열 등을 택할수 있을 것이다. Pintos 상에서 정의한 최대 file number가 128인 것과 메모리 누수의 위험을 방지하기 위한 이유 등으로 나는 구조체 배열로 file descriptor를 구현하였다. 정확히는 구조체 포인터 배열이며, 128개의 index를 가진 배열의 포인터는 3번부터 file object를 할당 받을수 있다. Linked list를 사용한다면 보다 효율적으로 메모리를 활용할 수 있겠지만, file descriptor number가 128개로, 배열로 구현하여도 큰 부담이 없다는 점과 메모리 효율대비 메모리 누수의 위험성을 모두 고려하였다. File system을 구현할때 file close와 open에 있어서 동기화를 하는 과정에서 메모리 누수가 발생할 위험이 있다고 판단하였다. 구조체 포인터 배열로 구현한 해당 file descriptor는 init\_thread에서 initialize해주며, system call handler에서 open을 구현할 때 각 file obejct와 연결시켜준다.

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)

1) create

Argument로 넘겨받은 initial\_size 크기의 새로운 file을 생성한다. 만약 성공한다면 true를 return 하고, 실패한다면 false를 return한다. Create는 file을 open하지는 않으며, open을 수행하는 동작은 이와 구분되어 open system call에서 해준다.

2) remove

Argument로 넘겨받은 이름의 file을 삭제한다. 마찬가지로 삭제를 성공하면 true를 return하고, 실패하면 false를 return한다. Remove system calld에서는 file이 open되어 있건 close되어 있건 상관없이 삭제해주며, 삭제를 할 때 close해주진 않는다.

3) open

넘겨받은 이름의 파일을 open 한다. 각 process는 독립적인 file descriptor를 가지고 있고, file descriptor는 child process에 상속되지 않는다. Open에 성공하면 할당된 file descriptor index가 return된다. 할당된 descriptor는 close에서 initialize된다.

4) filesize

Argument로 받은 fd index에 해당하는 file descriptor가 가르키는 file의 크기를 return한다.

5) seek

해당 fd index가 가르키고 있는 파일에서 읽거나 써야할 다음 byte를 바꾼다. 즉 파일의 위치(offset)을 이동시키는 함수이다.

6) tell

해당 파일의 위치를 알려주는 함수이다. Fd index가 가르키는 파일의 현재 offset을 return한다. 해당 offset은 file의 시작으로부터 byte 수로 표현된다.

7) close

Open 했던 file을 close해주는 함수이다. Exit하거나 terminate 하는 프로세스는 open 했던 모든 file descriptor를 close 해준다. 열었던 file을 close해주지 않을시 pintos의 multi-oom test를 통과할 수 없다.

8) read

Project1에서 구현하였던 stdin에 대한 read 뿐만 아니라 모든 file object에 대한 read를 구현한다. Readers writers problem을 해결해야 하며, 이 때 synchronization으로 semaphore 또는 lock이 사용된다.

9) write

Read와 마찬가지로, readers writer problem을 해결해야 하며, stdout에 대한 write 뿐만 아니라 다른 file object에 대한 write도 가능하도록 구현한다.

3. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명

Pintos에서는 synchromization의 tool로써 seamphore와 lock에 대한 함수를 제공한다. Project1에서는 process wati에 대한 동기화 구현을 위하여 semaphore를 사용했던바 있다. 이번 filesystem에 대한 synchronization을 구현할 때는 lock을 사용하려 한다. Lock을 구현하는 함수들은 모두 semaphore를 사용하므로, lock은 semaphore를 포함한다고 할 수 있다. Lock과 semaphore의 차이점은 lock에 한가지 제한사항이 추가됐다는 점이다. Lock에서는 해당 lock을 얻은 thread만이 (lock’s owner) 해당 lock을 release할 수 있다. Lock을 readers writers problem에서 first readers-writer problem으로, reader를 우선시 하도록 synchronization할 계획이다. Starvation이 일어날 수 있지만, read 동작에 대해서는 overhead가 생기지 않으므로 해당 방법을 택하였다. 결과적으로 read, write, open의 system call에서 lock을 사용하게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |  |
| --- | --- | --- |
| Date | Works | Details |
| 10.1 | Code preview | Pintos manual& ppt 활용 |
| 10.2-10.3 | System call coding | Create,seek,tell,filesize,close  Remove 구현 |
| 10.4 | Open, read, write구현 |
| 10.5-10.9 | Make chek & revise code | Test에 따른 코드 수정 |
| 10.10 | Reporter | 보고서 작성 |

* 1. **개발 방법**

1. File Descriptor

Threads/thread.h에 file descriptor를 구현하기 위하여 file 구조체 포인터 배열 Fd를 작성한다. 해당 file 구조체는 filesys/file.c 에 위치한다. Header file에서 선언한 Fd 배열은 thread.c의 init\_thread에서 모두 NULL로 그 값을 초기화 해준다. 생성되는 thread마다 file descriptor를 가지고 있으므로, 각 thread의 fd는 독립적이다. 새로운 file object에 대한 할당이나 open 했던 file descriptor에 대한 initialize는 userprog/syscall의 systemcall handler와 각 system call에서 호출하는 해당 함수에서 구현해 준다. File table을 가르키는 file descriptor pointer는 stack영역에 존재하고, File descriptor를 static하게 할당받았기 때문에, 따로 file descriptor 메모리에 대한 동적 해제를 해줄 필요 없다.

1. System Calls

우선, 추가되는 create, remove, open, filesize, seek, tell, close 에 대한 함수를 userprog/syscall.c에 추가해준다. 추가되는 시스템 콜 함수들은 모두 filesystem 관련된 함수로써, 명세서의 Useful APIs에 명시된 함수를 사용한다. 해당 API들은 filesys/file.h에 구현되어 있다. Create 함수는 filesys\_create 함수를 사용한다. 해당 함수의 자료형은 bool type이고 그대로 return에 사용될 수 있다. Remove에서는 filesys\_remove 함수를 사용한다. Create, remove, open 함수들은 argument로 file이름을 넘겨받는데, 이 때 해당 값이 NULL일 때 error가 발생할 수 있으므로 예외처리를 해주어야 한다. Oepn 함수는 filesys\_open 함수를 사용하며, 현재 thread의 file descriptor중 NULL을 가르키고 있는, 즉 file table을 가르키고 있지 않은 index에 file object를 할당하고, 해당 index를 return한다. 또한 read와 write에 대해서 synchronization을 해줘야 하므로 Lock을 사용한다.

다음으로 Filesize, seek, tell, close 함수에서는 모두 argumet로 넘겨받은 fd index에 대하여 해당 index의 file descriptor가 file table을 가르키고 있는지, 아니면 NULL 포인터를 가르키고 있는지 확인하여 예외처리를 해주어야 한다. 그리고 차례대로 file\_length 함수, file\_seek 함수, 그리고 file\_close함수를 사용한다. 이 때 file\_close함수에서는 따로 우리가 선언한 file descriptor pointer에 대한 initialize를 해주지 않으므로, close를 해주는 fd index에 대하여 매번 NULL 포인터로 initiallize 해준다.

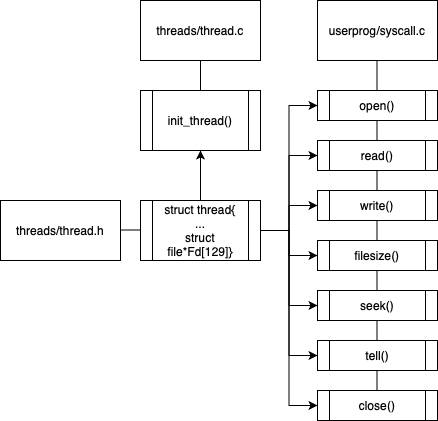
마지막으로 read, write 함수에서는 fd가 각각 0(stdin), 1(stdout)인 경우 외에 각각 File\_read, file\_write 함수를 사용하여 파일을 읽고 써준다. 그리고 synchronization이 필요하므로 lock을 써주는데, readers favor이므로 정수형 변수 readcount와 lock 구조체 변수를 2개 선언하여 활용한다.

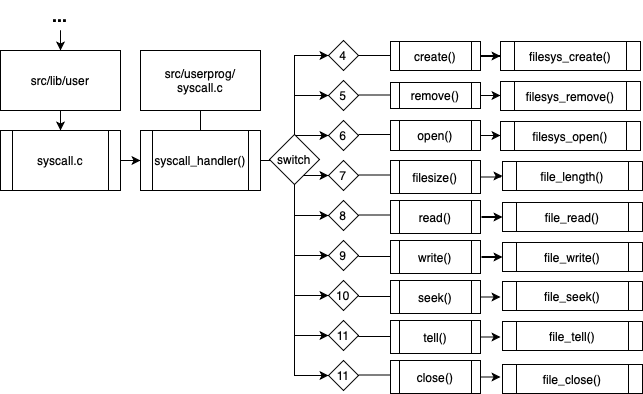
3. Synchronization in Filesystem

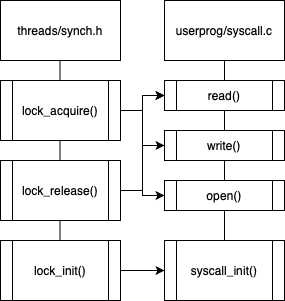
위에서 언급한 readers-writers problem을 해결하는데에 lock을 사용한다. Lock structure는 threads/synch.h에 위치한다. Readers favor로 해당 synchronization을 구현하므로, 정수형 변수 read count와 lock 구조체 변수 2개를 선언하여 userprog/syscall.c의 syscall\_init 함수에서 프로그램 초반에 시스템 콜을 초기화 해줄때 같이 lock 변수와 read count를 initialize 해준다. 이 때 앞에 언급한 변수들은 모두 global variable로 선언해 주는데, 이는 file에 대해 read, write, 또는 open system call로 접근하는 threads들에 대해서 각각 독립적으로 관리하는 것이 아닌, 공통적으로 lock을 해주어야 하기 때문이다. Lock을 사용할 때는 lock\_acquire와 lock\_release를 사용하며, 이 함수들 또한 threads/sync.h에 구현되어 있다.

1. **연구 결과**
   1. **Flow Chart**

* File Descriptor



* System Calls
* Synchronization in Filesystem

****

* 1. **제작 내용**
     1. 텍스트이(가) 표시된 사진

        자동 생성된 설명File Descriptor

텍스트이(가) 표시된 사진

자동 생성된 설명우선, thread.h에 왼쪽과 같이 file structd인 Fd를 선언한다. 총 128개의 index를 사용하며, 해당 file descriptor를 순회하다가 segment fault가 발생할 위험에 대비하여 크기를 129로 하였다.

다음으로 왼쪽 while문은 thread.c의 init\_thread 함수에서 해당 thread의 file descriptor인 Fd 포인터 배열을 NULL 포인터로 초기화 해주는 코드이다. 해당 file desciptor는 userprog/syscall.c의 시스템 콜 함수에서 사용된다.

* + 1. 텍스트이(가) 표시된 사진

       자동 생성된 설명System Calls

왼쪽은 syscall.c의 system handler 코드중 일부이다. 앞서 설명한 create, remove, open, filesize 등의 시스템 콜에대한 case 처리가 추가되었다. 특히 remove, open, create의 system call에서는 해당 함수를 호출할때 file name를 argument로 넘기는데, 이 때 NULL값을 가지고 있으면 오류가 발생할 수 있으므로 해당 값을 가지고 있는 p[0]를 사용하여 예외처리를 해주었다. P 배열은 project1에서 설명했듯이 user stack에 있는 값들을 argument로 시스템 콜 함수에 넘겨주기 전에 esp 스택 포인터를 사용하여 저장해 놓은 배열이다. 또한 다른 시스템 콜 함수에서 해줬었던 바와 같이 protect\_user\_memory 함수를 사용하여 사용하고 있는 주소가 user memory 영역인지 확인하고, kernel memory 영역등의 허용된 영역이 아닐 경우 exit(-1)을 해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위는 create와 remove에 대한 system call 함수이다. 정확하게는 해당 API와 연결시켜주기 위하여 system call handler에서 해당 system call에 따라서 호출한 함수이다. 이번 프로젝트에서 구현한 system call 함수를 매게하는 방법에 있어서 사용한 함수들은 공통적으로 filesys/filesys.h와 filesys/file.h에 있는 함수(API)들을 사용하였다. Create, open, remove 구현에서는 filesys.h에서 정의된 함수들을 사용한다.

먼저 create와 remove는 위 그림에서 볼 수 있듯 각각의 헤더파일로부터 filesys\_create() 함수와 filesys\_remove() 함수를 호출한다. 이 때 함수를 return 하는데, 이는 해당 함수들이 bool type이고, 그 결과인 반환값을 바로 위 함수에서 return하는 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명filesize, seek, tell, 그리고 close 함수는 각각의 시스템 이 발생한 userprogram과 kernel을 매게하는 역할을 하는 함수이고, 공통적으로 int fd를 argument로 지니고 있는데, 이는 file descriptor index로써 만약 해당 index를 가지는 file descriptor가 NULL 포인터를 가르키고 있을시에는 file object가 할당된것이 아니므로, 전부 exit(-1)을 해줌으로써 예외처리를 해주었다. 앞에서부터 차례대로 file\_length()함수, file\_seek()함수, file\_tell()함수, file\_close()함수를 사용하며, 모두 해당 코드 오른쪽에 적혀있는 헤더파일에 구현되어 있으므로 단순히 호출해 줌으로써 해당 기능들을 구현하였다.

특히 close 함수에서는 thread\_current() 함수(thread.c에 구현되어 있다.)를 사용하여 Fd[fd]값을 file\_close() 함수의 argument로 넘겨주고도, NULL값으로 초기화 해주는것을 볼 수 있는데, 이는 위에서도 언급했듯이 file\_close() 함수에서 해당 thread의 file descriptor를 초기화 해주지 않기 때문이다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명왼쪽은 open 함수를 구현한 것이다. Synchronization에 관한것은 아래에서 설명하기로 한다. Open 함수에서는 filesys 디렉토리의 filesys.h에 정의된 filesys\_open()함수를 사용한다.해당 함수의 반환값이 NULL일 경우 return -1을 해줌으로써 예외처리를 해주었다. 새로운 file에 대한 fd를 할당해주는 구현을 while문을 사용하였다. Fd배열의 index를 증가시키면서 할당되지 않은 포인터를 찾아서 새로운 file을 가르키도록 하였다. 이 때 모든 file descriptor가 할당되어 있을 경우 return -1로 예외처리 해주었다. 다음으로 strcmp를 사용하여 running program에 대한 file open을 해주려고 할 때, 즉 현재 실행파일에 대해 덮어쓰려고 하는 경우에 대한 예외처리를 해준 것을 확인할 수 있다. 이는 이미 load된 memory를 remove하는 것은 문제가 없을 수 있지만 pinto 상에서는 이를 금지하기 때문이다. 이 때 filesys/file.c에 있는file\_deny\_write()함수를 사용하였다.

왼쪽은 project1에서 구현한 exit 시스템 콜에 대한 매개 함수이다. Thread가 exit 시스템 콜을 호출하여 죽기전에, 할당하였던 모든 file descriptor를 close 해줌으로써 메모리 누수를 방지하였다. 이런 수행을 exit함수에서 하기 때문에, process.c의 start\_process 함수에서 load를 실패하였을때 thread\_exit()을 수행해주었던 것을 exit(-1)로 바꿔줌으로써 thread 종료 전에 filedescriptor 포인터에 대한 close를 진행시켜 주었다.

다음으로 read와 write 함수에서는 각각 fd index값에 따라서 다른 수행을 해주는데, 먼저 read에서는 fd == 0인경우를 project1에서 구현하였고, fd >2인 경우에 대해서 새로 구현해주었다. Fd == 2인 경우는 stderr로 가정하였고, fd >2인 경웅 file\_read 함수를 사용하였다. 해당 file read 함수로부터 읽어들인 bytes를 rbytes 변수에 저장하여 반환해 주었다.

마지막으로 write 함수에서는 fd == 1인 경우를 project1에서 구현하였고, fd>2인 경우에 file\_write()함수를 호출하여 쓴 bytes 결과값을 wbytes 변수에 저장하여 최종적으로 write 함수에서 이를 반환해주었다.

3. Synchronization in Filesystem

**텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

위는 read와 write의 system call 처리에 관한 함수이다. 시스템 콜과 관련된 구현 내용은 앞서 2번에서 설명하였다. 먼저 lock 구조체 변수 mut과 w를 선언하였고, 이를 syscall\_init 함수에서 lock\_init 함수를 사용하여 초기화해주었다. 해당 lock과 관련된 함수는 threads/synch.c에 구현되어 있다. 그리고 readcnt를 전역변수로 설정하여 syscall\_init 함수에서 그 값을 0으로 초기화 해주었다. 이 변수들은 readers writers problem에 대하여 readers favor로 synchronization을 해주기 위한 변수이다. Read 함수로 돌아와서 코드를 살펴보면 readcnt를 증가시켜주고, read 시스템 콜을 처음 호출한 thread일 경우 w를 사용하여 lock을 해준것을 볼 수 있고, 다시 readcnt == 0, 즉 read에 대한 수행이 없을 경우에 다시 lock을 해제시켜준 것을 볼 수 있다. 이로써 read 시스템 콜에 대한 수행은 병목현상 없이 이루어지지만, write 시스템 콜에 대해서는 먼저 read 시스템 콜에 대한 수행이 모두 끝난뒤, 심지어 그 뒤로 호출된 read 시스템 콜에 대한 처리까지 끝난뒤에 수행되는 것을 볼 수 있다. 하지만, 완벽한 readers favor라고 할 수는 없는 점이 open과의 동기화에 있다. Open에서도 read, write에서와 마찬간지로 w를 사용하여 Lock을 해주기 때문에, 동일 파일에 대해서 open을 또 수행하여 읽으려는 경우에는 병목현상이 발생한다. Read, write, open 함수에서 lock을 해준 critical section을 최소화 하기 위해서 각 예외처리를 해주고 난 후의 범위만을 critical section에 포함시켜, 수행 속도를 고려하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명다음은 make check를 통해서 test를 하던make중 multi-oom등에 대한 test를 통과하지 못하여 디버깅을 하던중 고친 부분에 대한 process.c의 코드이다. 우선 User program에서 thread create을 하고 다른 thread들이 종료되는 것은 wait을 함으로써 동기화되지만, load되기전에 종료되는 것에 대한 예외처리는 해주지 않았던 것을 발견하고 thread 구조체에 wait\_load 라는 변수의 semaphore 구조체를 추가하여(앞 서 thread 구조체 그림에서 확인할 수 있다.) process\_execute 함수에서 thread\_create을 해준 다음 sema\_down을 통해 wait을 한 다음, start\_process 함수에서 load수행이 끝나고 sem\_up을 통해 wait 수행을 끝내준다(아래 그림). 그리고 만약 load를 실패하였다면 실패한 thread들에 대한 수거 처리도 해주어야 한다. 따라서 위쪽 그림에서 처럼 load가 실패한 함수들에 대해서 process\_wait()함수를 호출해주었고, load를 실패한 thread들은 -1을 return할 것이다.

* 1. **텍스트이(가) 표시된 사진

     자동 생성된 설명**텍스트, 명판이(가) 표시된 사진

     자동 생성된 설명**시험 및 평가 내용**

Make check와 Make grade를 한 결과, 성공적으로 모든 test들에 대해서 통과한 것을 확인할 수 있었다.