Pintos 프로젝트 2_2. Pintos User Program

(설계 프로젝트 수행 결과)

과목명 : [CSE4070-01] 운영체제
담당교수 : 서강대학교 컴퓨터공학과 박성용

조원 : 83조 권명준, 전해성
개발기간 : 2017. 10. 28. - 2017. 11. 10.

최 종 보 고 서

프로젝트 제목: Pintos 프로젝트 2_2. User Program Basic
제출일: 2017. 11. 10.
참여조원: 83조 권명준, 전해성

I. 개발 목표

지난 프로젝트 2-1에서는 standard input/output에 대해서만 입/출력이 가능하도록 file system과 관련된 system call을 구현했었다. 이번 프로젝트 2-2에서는 일반적인 입/출력이 가능하도록 기존의 system call 2개를 수정하고 새로 9개를 구현한다.

II. 개발 범위 및 내용

가. 개발 범위

File system과 관련된 system call: create, remove, open, close, filesize, read, write, seek, tell을 구현한다. read와 write는 프로젝트 2-1에서 standard io만 가능했던 것을 일반적인 입/출력이 가능하도록 수정하는 것이고, 나머지는 새로 작성하는 것이다. 이때 동기화 문제를 적절히 해결하며 실행되고 있는 파일에 쓰기 작업을 제한하도록 제약을 준다.

나. 개발 내용

상기한 system call을 pintos manual에서 요구하는 기능대로 구현한다. 이때 몇 가지 가정을 한다. 우리는 파일을 만드는 시점에 파일 이름뿐만 아니라 미리 파일 크기를 정해놓는다. 또한 우리는 subdirectory를 취급하지 않기로 한다.

bool **create** (const char *file, unsigned initial_size) [System Call]
파일 이름과 파일 크기를 인수로 받아 파일을 생성하고 성공 여부를 반환한다. 파일을 만들기만 할 뿐 열지는 않는다.

bool **remove** (const char *file) [System Call]
파일 이름을 인수로 받아 해당 파일을 제거한다. 성공 여부를 반환한다. 파일은 열려있든 닫혀있는 무관하게 삭제되고 삭제된다고 해서 닫는 것도 아니다.

int **open** (const char *file) [System Call]
파일 이름을 인수로 받아 해당 파일을 연다. 음이 아닌 정수 값을 갖는 'file descriptor'를 반환한다. 만약 열기에 실패했다면 -1을 반환한다. 이때 file descriptor 0은 standard input, 1은 standard output으로 미리 정해져있다. 이 system call은 0 또는

1을 절대 반환하지 않는다. 모든 프로세스는 독립적인 file descriptor를 갖고 있다. 만약 하나의 파일이 여러 번 열리면 다른 file descriptor를 갖게 된다.

int **filesize** (int fd) [System Call]
file descriptor가 fd인 파일의 크기를 byte 단위로 반환한다. file descriptor가 할당되어 있다는 것은 열려있음을 함의한다.

int **read** (int fd, void *buffer, unsigned size) [System Call]
buffer에서 size bytes만큼을 열려있는 파일 fd에 읽어 들인다. 읽어 들인 bytes 수를 반환한다. EOF라서 읽을 수 없었다면 0을 반환하고 다른 이유로 읽을 수 없었다면 -1을 반환한다. fd==0일 경우 standard input인 keyboard로부터 읽어 들이며 이때 input_getc()를 사용하여 처리한다.

int **write** (int fd, const void *buffer, unsigned size) [System Call]
buffer에서 size bytes만큼을 열려 있는 파일 fd에 쓴다. 쓰기에 성공한 bytes 수를 반환한다. 만약 EOF를 넘어서 쓰려고 하면 이를 넘지 못하게 한다. fd==1은 standard output이므로 console에 출력한다.

void **seek** (int fd, unsigned position) [System Call]
fd의 파일 위치 지정자(읽고 쓰는 위치로, 커서와 비슷한 것)를 position으로 변경한다. 다시 말해서 fd의 위치 지정자는 파일 처음에서부터 position bytes만큼 떨어진 곳이 된다. position==0이면 파일 처음을 가리키는 것이다.

unsigned **tell** (int fd) [System Call]
seek과 반대되는 system call로, fd의 현재 위치 지정자의 위치를 반환한다. seek과 마찬가지로 파일 처음에서부터 얼마나 떨어져 있는지를 bytes단위로 나타내는 것이다.

void **close** (int fd) [System Call]
file descriptor가 fd인 파일을 닫는다. 어떤 프로세스가 종료되면 내부적으로 그 프로세스와 관련된 모든 file을 닫는데 이때 이 함수를 각각 호출하는 셈이다.

또한 이 system call들의 critical section problem을 적절히 해결해 동기화 이슈를 처리한다. 그리고 실행되는 파일에 쓰기 작업을 제한하도록 구현한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

- 10월 28일 토요일 : 이번 프로젝트에 대한 강의를 듣고 내용을 파악한다.
- 10월 31일 ~ 11월 1일 : 동기화를 배제하고 기존적인 코드의 틀을 작성한다.
- 11월 2일 ~ 11월 6일 : 동기화 문제를 해결한다.
- 11월 7일 ~ 11월 8일 : 실행되고 있는 파일에 쓰기 작업을 막는 기능을 구현한다.
- 10월 9일 ~ 10월 10일 : 보고서를 작성하고 마무리한다.

나. 개발 방법

이미 지난 프로젝트 2-1에서 argument passing과 기초 system call등 핵심적인 부분이 구현되어 있기 때문에 이번 프로젝트는 특별히 더 공부하지 않고 바로 작업에 착수해도 좋다고 판단하였다. pintos manual이 요구하는 대로 먼저 system call 코드를 작성한다. 그리고 test 파일을 돌려봄으로써 동기화 문제가 제대로 처리되었는지를 확인하고 이를 적절히 처리한다. 이때 lock을 사용하는데, lock의 사용법을 찾아봐야 한다. 마지막으로, 프로세스로 존재하는 실행되는 파일에 쓰기 작업을 하지 못하도록 제약을 걸어준다.

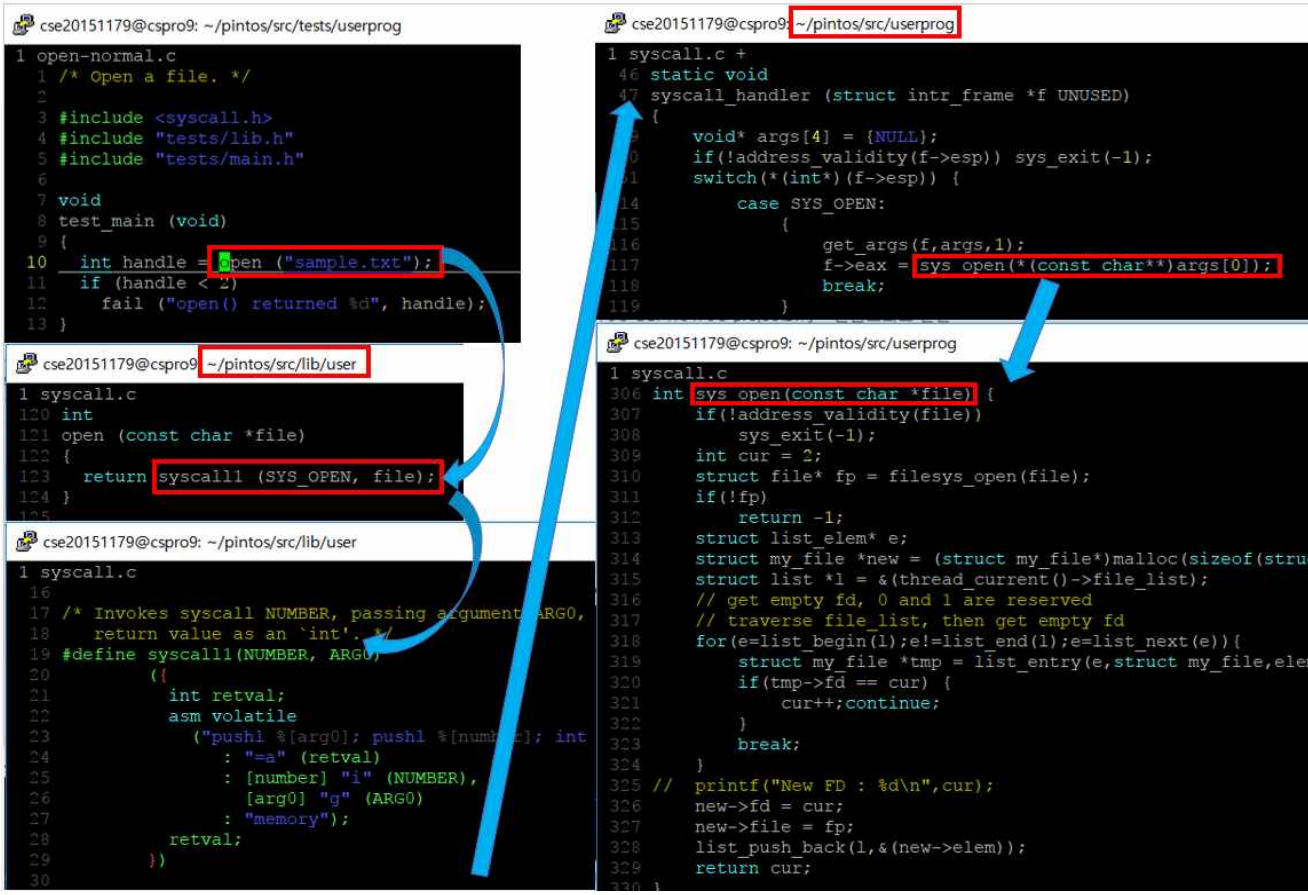
다. 연구원 역할 분담

- 전해성 : system call의 기능을 구현, 동기화와 실행되고 있는 파일에 쓰기 제한 구현 등 모든 개발에 기여
- 권명준 : 동기화와 실행되고 있는 파일에 쓰기 제한 구현, 보고서 작성

Ⅳ. 연구 결과

1. 합성 내용

1-1. open() 함수 호출부터 sys_open()까지의 호출 경로



#include <syscall.h>를 하면 pintos/src/lib/user/syscall.h가 포함된다. 여기에 있는 open()을 호출하면 pintos/src/lib/user/syscall.c에서 SYS_OPEN이라는 번호(번호는 pintos/src/lib/syscall-nr.h에 enum의 형태로 정의되어있다.)의 system call을 호출한다. 어셈블리 코드가 실행되어 pintos/src/userprog/syscall.c의 syscall_handler에서 정보를 받을 수 있고 이는 마침내 sys_open()을 호출한다. 따라서 sys_open에서 구현한 코드가 system call로서 동작할 수 있다. 이는 다른 system call 또한 마찬가지의 경로를 거쳐서 동작한다. 따라서 #include <syscall.h>를 하고 여기 있는 함수를 사용하면, 함수가 interrupt를 걸고 system call로서 동작하게 되고, 코드는 pintos/src/userprog/syscall.c에 작성한다.

1-2. Denying writes to executables의 구현 원리

thread.h

```
struct thread{
...
        file * cur_file;
...
}
```

process.c

```
static void start_process(void *file_name_) {
...
85  +  cur->cur_file = filesys_open(file_name);
86  +  if(cur->cur_file) {
87  +          file_deny_write(cur->cur_file);
88  +  }
89  +
...
}
```

syscall.c

```
void sys_exit(int status) {
...
207 +      printf("%s: exit(%d)\n",cur->name,status);
208 +      if(cur->cur_file){
209 +              file_close(cur->cur_file);
210 +      }
211        thread_exit();
}
```

struct thread에 file* 타입의 cur_file 변수를 추가해준다. 그리고 프로세스가 시작될 때, 프로세스 이름과 동일한 파일을 열고 cur_file에 저장한다. 이는 filesys_open()을 통해 구현한다. 이에 대한 이론적 근거는 다음과 같다:

Thus, to deny writes to a process's executable, you must keep it open as long as the process is still running. (33p, pintos manual)

열림 상태로 만든 뒤에 file_deny_write()를 사용해 '현재 만들고 있는 thread'를 실행시킨 파일을 쓰기 불가능 상태로 만든다.

이제 sys_exit()의 마지막 부분에 file_close()를 추가하여 앞에서 열었던 파일을 닫아준다. 이렇게 함으로써 프로세스로 실행되는 파일에 쓰기 기능을 막아 놓을 수 있다.

2. 제작 내용

2-1. 자료구조

struct thread (thread.h)

```
struct thread
 {
/* Owned by thread.c. */
tid_t tid; /* Thread identifier. */
enum thread_status status; /* Thread state. */
char name[16]; /* Name (for debugging purposes). */
uint8_t *stack; /* Saved stack pointer. */
int priority; /* Priority. */
struct list_elem allelem; /* List element for all threads list. */

/* Shared between thread.c and synch.c. */
struct list_elem elem; /* List element. */

#ifdef USERPROG
/* Owned by userprog/process.c. */
uint32_t *pagedir; /* Page directory. */
#endif

/* Owned by thread.c. */
/* proj 2-1 */
struct thread* parent; /* identifier of parent */
tid_t cur_child; /* tid of current waiting child */
bool child_load_successful;
int child_status; /* exit status of child */
struct semaphore sema;
struct list child_list;

/* proj 2-2 */
struct list file_list; /* current thread open file list */
struct file* cur_file; //KMJ
unsigned magic; /* Detects stack overflow. MUST BE AT THE BOTTOM -
KMJ */
 };
```

주석에서 확인할 수 있듯 이 프로젝트를 위해서 struct thread에 2개의 변수를 추가하였다. struct list file_list는 어떤 thread가 연 파일의 목록을 struct list로 보관한다. struct file* cur_file은 현재 thread를 실행하는 데에 사용한 파일을 가리키는 포인터로, 실행되는 파일에 쓰기 제한(denying writes to executables)를 위해 만들었다.

struct file (file.h)

```
struct file
 {
struct inode *inode; /* File's inode. */
off_t pos; /* Current position. */
bool deny_write; /* Has file_deny_write() been
called? */
 };
```

이 구조체는 수정하지 않고 사용했다. 대신 아래 구조체를 새로 만들었다.

struct my_file (thread.h)

```
struct my_file{
struct list_elem elem;
int fd;
struct file *file;
};
```

my_file은 file*에 더하여 리스트에 추가할 수 있는 struct list_elem, file descriptor를 포함한다.

2-2. 알고리즘

serachFileList() 정의 (pintos/src/threads/thread.c)

```
struct my_file* searchFileList(struct list *file_list, int fd) {
  struct my_file *tmp;
  struct list_elem *e;
  for(e=list_begin(file_list);e != list_end(file_list);
   e = list_next(e)) {
   tmp = list_entry(e,struct my_file,elem);
  if(tmp->fd == fd) {
  return tmp;
   }
   }
  return NULL;
}
```

이 함수는 syscall.c에서 파일을 열고 닫고 처리할 때 file descriptor을 가지고 파일 포인터를 찾아내기 위해 많이 호출된다. 이 함수를 구현함으로써 쉽게 그런 기능을 수행한다. 원리는 thread 구조체의 file_list의 모든 연결 리스트 노드를 탐색하여 이름을 비교하는 것이다.

CREATE, REMOVE의 구현 (pintos/src/userprog/syscall.c)

```
bool sys_create(constchar *file,unsigned initial_size) {
if(!address_validity(file))
sys_exit(-1);
returnfilesys_create(file,initial_size);
}
```

대표적인 예로 sys_create()만을 적어놓았다. 주소가 적절한지를 판단하고 filesys_create() API를 호출하는 간단한 원리이다. REMOVE의 경우도 호출하는 API만 다르고 구조가 같다.

OPEN의 구현 (pintos/src/userprog/syscall.c)

```
int sys_open(constchar *file) {
if(!address_validity(file))
sys_exit(-1);
int cur = 2;
struct file* fp = filesys_open(file);
if(!fp)
return -1;
struct list_elem* e;
struct my_file *new = (struct my_file*)malloc(sizeof(struct my_file));
struct list *l = &(thread_current()->file_list);
// get empty fd, 0 and 1 are reserved
// traverse file_list, then get empty fd
for(e=list_begin(l);e!=list_end(l);e=list_next(e)){
struct my_file *tmp = list_entry(e,struct my_file,elem);
if(tmp->fd == cur) {
 cur++;continue;
 }
break;
 }
// printf("New FD : %d₩n",cur);
 new->fd = cur;
 new->file = fp;
list_push_back(l,&(new->elem));
return cur;
}
```

먼저 file 포인터의 적절성을 검토한 뒤에 시작한다. filesys_open() API를 사용하여 파일을 연다. 이를 thread 구조체의 file_list에 추가해준다. 그리고 file descriptor를 할당하고 이를 반환한다.

SEEK, TELL, FILESIZE의 구현 (pintos/src/userprog/syscall.c)

```
voidsys_seek(int fd, unsigned pos) {
struct my_file *tmp = searchFileList(&(thread_current()->file_list),fd);
if(!tmp)
return ;
file_seek(tmp->file,pos);

}
```

대표적인 예로 SEEK을 상기하였다. file descriptor 정보를 사용해 파일을 찾는다. SEEK은 file 구조체의 pos 정보(위치 지정자)를 file_seek() API를 사용해 수정하고 TELL은 file_tell() API를 사용해 이를 반환한다. FILESIZE는 file_length() API를 사용해 받은 값을 그대로 반환한다.

READ, WRITE의 구현 (pintos/src/userprog/syscall.c)

```
int sys_read(int fd, void *buffer, unsigned size) {
int i;
char input;
if(!address_validity(buffer)) sys_exit(-1);
if(fd == 0) {
for(i=0;i<size;i++) {
 input = input_getc();
memset((char*)buffer+i,input,1);
 }
return i;
 }
elseif(fd == 1)
return -1;
else {
struct my_file *tmp = searchFileList(&(thread_current()->file_list),fd);
if(!tmp)
return -1;
returnfile_read(tmp->file,buffer,size);
 }
}
```

대표적인 예로 READ만 표시하였다. fd==0이면 standard io를 사용하고 그 외의 경우 파일을 찾아 존재한다면 file_read() 또는 file_write() API를 호출한다.

CLOSE의 구현 (pintos/src/userprog/syscall.c)

```
voidsys_close(int fd) {
struct my_file *tmp = searchFileList(&(thread_current()->file_list),fd);
if(!tmp)
return ;
file_close(tmp->file);
list_remove(&(tmp->elem));
free(tmp);
}
```

file descriptor를 사용해 파일을 찾고 file_close() API를 사용해 닫는다. 또한 리스트에서도 제거해주고 동적 할당한 메모리도 풀어준다.

2-3. 동기화

먼저 pintos/src/filesys/filesys.c에 struct lock *filesys_lock;을 선언해주었다. 그리고 filesys.c에 있는 filesys_create(), filesys_remove(), filesys_open()에 lock_acquire()과 lock_release()를 추가했다. 아래에는 대표적으로 filesys_create()만을 예시로 실었다.

```
49  bool
50  filesys_create (const char *name, off_t initial_size)
51  {
52  +       lock_acquire(filesys_lock);
53    block_sector_t inode_sector = 0;
54    struct dir *dir = dir_open_root ();
55    bool success = (dir != NULL
56                    && free_map_allocate (1, &inode_sector)
57                    && inode_create (inode_sector, initial_size)
58                    && dir_add (dir, name, inode_sector));
59    if (!success && inode_sector != 0)
60      free_map_release (inode_sector, 1);
61    dir_close (dir);
62  + lock_release(filesys_lock);
63
64    return success;
65  }
66
```

3. 시험 및 평가 내용:

3-1. tests

tests 폴더에 있는 userprog의 모든 test를 통과하였다.

```
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                        Pts Max  % Ttl  % Max
----------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality             108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness                 88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                       1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                        30/ 30  30.0%/ 30.0%
----------------------------------------------- --- --- ------ ------
Total                                                   100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

3-2. 보건 및 안정성

파일 포인터에 NULL이 오거나 적절하지 않은 포인터 값이 온다고 해도 이런 예외를 적절히 처리하였다. 또한 0바이트를 읽는 등의 틀수한 경우도 pintos manual이 요구한 대로 동작할 수 있도록 만들었다.

예를 들어 `open ((char *) 0x20101234));`, `open (NULL);`등도 예외 처리가 되고 `byte_cnt = read (handle, &buf, 0);`는 문제없이 0을 반환한다.

상기한 대로 lock을 사용하여 critical section을 보호하였기에 동기화 문제도 적절히 처리되었다. Denying writes to executables도 상기한대로 프로세스가 시작할 때 해당 파일을 열고 쓰기를 제한하며 프로세스 exit()할 때 파일을 close()함으로써 이 기능을 구현하였다. 예를 들어 rox-child 프로세스에서 이를 실행시킨 파일에 쓰고자 한다면 쓰기가 제한된다.

3-3. 생산성 및 내구성

file 구조체를 수정하는 대신 my_file 구조체를 만들어서 사용하였다. 이를 통해 file descriptor 정보를 저장하고 list에 포함될 수 있게 함으로써 코드가 더 간결해질 수 있었다. 프로세스를 실행시킨 파일에 쓰기 작업을 제한하는 것을 구현할 때, 처음에 생각한 방법은 파일을 열 때 (또는 쓰기 작업을 할 때) 프로세스 전체를 all_list를 모두 탐색해서 이름이 같은 프로세스가 있다면 해당 파일을 쓰기 불가 상태로 만들려고 했다. 그러나 이 경우 파일을 열 때마다 모든 프로세스를 탐색해야하는 비효율적인 측면이 있었다. 그래서 나온 대안은 thread를 만들 때, thread이름과 같은 파일을 열어 file descriptor를 만들어 이를 쓰기 불가 상태로 만드는 것이다. 이는 모든 리스트를 탐색할 필요가 없어서 훨씬 효율적으로 동작할 수 있다.

V. 기타

1. 연구 조원 기여도: 권명준(40%) 전해성(60%)

2. 기타 본 설계 프로젝트를 수행하면서 느낀 점을 요약하여 기술하라. 내용은 어떤 것이든 상관이 없으며, 본 프로젝트에 대한 문제점 제시 및 제안을 포함하여 자유롭게 기술할 것.
(느낀점)
- 권명준: 이번 프로젝트는 지난번 프로젝트에서 구현한 것에 system call을 몇 개 더 추가하는 것뿐이어서 훨씬 이해하기도 쉽고 난감함도 없었다. 지난 프로젝트는 도대체 뭘 하라는 건지 알 수가 없어서 울고 싶었는데 이번 것은 바로 이해가 되어서 수월했다. 좀 난이도 균형이 맞지 않는 거 같다. 지난 프로젝트를 좀 줄이고 이 프로젝트를 늘리는 것도 괜찮을 것 같다. 이번 프로젝트는 마음에 여유가 있는 만큼 매뉴얼도 더 꼼꼼히 읽고 최적화를 더 시킬 수 있었다. 또한 github를 사용하니 무엇을 언제 바꿨는지 로그가 남아 좋았다. 이를 다른 학생들에게도 권고해주면 좋을 것 같다.

- 전해성 :
(수업시간에 배운 process의 수행 시나리오와 동기화 과정을 본 프로젝트를 통해 구현해봄으로써 교과목에 대한 이해가 더 깊어지는 것을 알 수 있었다. 프로젝트를 처음 시작할 때, 정말 뭐부터 손대야 할지 감도 안 와서 상당히 고통스러웠다. 차분하게 소스코드들을 분석하면서 pintos의 실행과정을 알고 나니 프로젝트의 진행이 가능했다. 본 실습은 대부분의 수강생이 학교 측에서 제공하는 서버 위에서 진행하게 되는데 분석할 때, ctags만으론 조금 부족한 느낌이 있었다. 그래서 제공받은 서버가 아닌 개인 서버를 이용해서 추가적으로 cscope를 추가로 사용하였다. 서버에 cscope도 깔아주면 좋겠다. 그리고 이번 학기는 연휴 때문에 늦어져서인지 동기화에 대한 수업이 프로젝트 마감일이 있는 주에야 시작되었다. 이 점 또한 아쉬웠다.)