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

담당 교수 / 분반 : 박성용 교수님 / 1분반

이름 / 학번 : 강상원 / 20191559

개발 기간 : 2023. 10. 09. ~ 2023. 10. 15.

1. **개발 목표**

Proj #1에서 개발한 프로젝트에 더해 파일 시스템에 필요한 System Call을 개발한다(create, remove, open, close, filesize, read, write, seek, tell). 더불어 File System을 다룰 때 발생할 수 있는 synchronization 문제를 해결해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
      1. File Descriptor

Proj #1까지의 구현 내용으로는 PintOS에서 파일을 읽은 후, 이를 구분, 식별할 수 있는 방법이 존재하지 않았다. 따라서 File Descriptor를 개발해서 각 프로세스마다 입, 출력 정보를 다루는(여는 파일을 저장하는) 구조가 필요하다. 0은 STDIN, 1은 STDOUT, 2는 STDERR, 이후 다른 입출력에 할당하여 프로세스마다 입출력을 관리할 수 있다.

* + 1. (이번 프로젝트에서 구현해야 하는) System Calls

기존까지는 STDIN, STDOUT에 해당하는 입출력만 고려하였다면, 이제는 read, write 할 때 다른 입출력의 경우에도 동작하게 해야 한다. create, remove, open, close, filesize, read, write, seek, tell의 system call들로 하여금 받아온 argument들을 이용해 file system을 다룰 수 있게 해야 한다.

* + 1. Synchronization in Filesystem

동일한 파일에 여러 프로세스가 동시에 접속하려 하는 등의 상황이 발생하면, 파일 입출력에서 사용자가 예측할 수 없는 결과를 초래할 수 있다. 또한 프로세스가 수행되기 시작할 때, 종료될 때에도 적절한 syncronization이 필요하다. 이를 위해 semaphore를 선언해 lock, unlock 해야 한다.

* 1. **개발 내용**
     1. File Descriptor

struct file\* FD[128];를 thread.h에 선언해서 각 thread마다 길이가 128인 File Descriptor를 가지게 한다. (Pintos Manual에 의거, 프로세스당 open file 개수 최대 128개) file 구조체의 배열로 만들어서 순번대로 쉽게 각 원소에 접근할 수 있게 하였다.

* + 1. System Calls
       - CREATE

file 이름을 인자로 받아 이에 해당하는 새로운 file을 만든다. boolean 값으로 주어진 file 생성 실패 유뮤를 반환한다.

* + - * REMOVE

file 이름을 인자로 받아 이에 해당하는 file을 삭제한다. boolean 값으로 주어진 file 삭제 실패 유무를 반환한다.

* + - * OPEN

file 이름을 인자로 받아 이에 해당하는 file을 open한다. 같은 이름의 파일을 두 번 이상 열지 못하게 예외 처리하며, 여는 데 성공한 file의 file descriptor 값을 반환하거나 오류 -1을 반환한다.

* + - * CLOSE

주어진 file descriptor 값에 해당하는 열린 file을 찾아 닫는다. 열려 있지 않은 file을 닫으려 시도한 경우에 대해 예외 처리를 한다.

* + - * FILESIZE

주어진 file descriptor 값에 해당하는 열린 file을 찾아 file size를 반환한다. 파일을 찾지 못하거나 읽지 못하는 경우에 대한 예외 처리를 한다.

* + - * READ

주어진 file descriptor 값에 해당하는 열린 file을 찾아 buffer에 저장한다. 읽은 file byte를 반환하고 오류일 경우 -1을 반환한다.

* + - * WRITE

주어진 file descriptor 값에 해당하는 file에 인자로 받은 size만큼 write한다. 작성한 file byte를 반환하고 오류일 경우 -1을 반환한다.

* + - * SEEK

인자로 받은 pos만큼 현재 file의 포인터 위치를 변경한다.

* + - * TELL

현재 open된 file의 현재 포인터 위치를 반환한다. 즉, READ/WRITE 하고 있는 위치를 반환한다.

* + 1. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명 (다른 방법을 서술해도 되지만 lock과 semaphore는 반드시 포함해야 함)

동일한 파일에 여러 프로세스가 동시에 접근해 READ, WRITE를 시도하는 경우 사용자가 예상치 못한 결과값을 도출할 수 있으므로 이를 방지하기 위해 lock을 사용한다. System call initialization 단계에서 lock\_init(), file 접근 전 lock\_acquire, 이후 lock\_release()를 하여 수행을 보호한다.

Semaphore는 프로세스 간 syncronization을 위해 이용한다. 부모 thread가 wait로 각 자식을 거둬들이기 전에 종료되는 경우를 막기 위해, sema\_up(), sema\_down()을 사용해 자식 thread가 load된 후 부모 실행하도록 설정하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

총 개발 기간: 2023. 10. 09. ~ 2023. 10. 15

* + - * 10/9 ~ 10/11: File Descriptor 구현, 이에 대응하는 System Call (create, remove, open, close, filesize, read, write, seek, tell) 구현
      * 10/12 ~ 10/14: Semaphore, Lock 구현 (sys\_write, sys\_read 해결)
      * 10/15: 보고서 작성
  1. **개발 방법**
     1. File Descriptor

File Descriptor 관련해서 struct file\* FD[128];를 thread.h에 선언해서 각 thread마다 길이가 128인 File Descriptor를 가지게 한다. (Pintos Manual에 의거, 프로세스당 open file 개수 최대 128개) file 구조체의 배열로 만들어서 순번대로 쉽게 각 원소에 접근할 수 있게 하였다.

* + 1. System Calls

[userprog/syscall.c]에서 create, remove, open, close, filesize, read, write, seek, tell에 해당하는 system call을 구현하였다. 이에 대한 구현 사항은 II.B에 상술하였다. [userprog/exception.c]에서 page fault 검색 조건에 not\_present를 주가하였다.

* + 1. Synchronization in Filesystem

[userprog/syscall.c]에 lock\_file이라는 lock을 추가하고, 이를 통해 파일 접근 시 critical section에 접근하는 것을 막는다. [threads/thread.h]에 struct semaphore load\_lock을 추가해서, 부모 thread가 wait로 각 자식을 거둬들이기 전에 종료되는 경우를 막기 위해, sema\_up(), sema\_down()을 사용해 자식 thread가 load된 후 부모 실행하도록 설정하였다.

1. **연구 결과**
   1. **Flow Chart**
      1. File Descriptor

**스크린샷, 도표, 라인, 디자인이(가) 표시된 사진

자동 생성된 설명**

* + 1. System Calls

**디자인, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명**

* + 1. Synchronization in Filesystem

**스크린샷, 도표, 디자인, 직사각형이(가) 표시된 사진

자동 생성된 설명**

* 1. **제작 내용**
     1. **File Descriptor**
        + [threads/thread.h]

struct file\* FD[128];를 thread.h에 선언해서 각 thread마다 길이가 128인 File Descriptor를 가지게 한다. (Pintos Manual에 의거, 프로세스당 open file 개수 최대 128개) file 구조체의 배열로 만들어서 순번대로 쉽게 각 원소에 접근할 수 있게 하였다.

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. \*/

/\* Added in #Proj 1 \*/

int exit\_status; /\* Exit status of process \*/

struct list child\_list; /\* List of child processes \*/

struct list\_elem child\_elem; /\* Element of child\_list \*/

// Semaphores

struct semaphore mem\_lock; /\* Lock of memory access \*/

struct semaphore child\_lock; /\* Lock of child process \*/

struct semaphore load\_lock; /\* Lock of load \*/

struct file\* FD[128]; /\* File Descriptor \*/

#endif

/\* Owned by thread.c. \*/

unsigned magic; /\* Detects stack overflow. \*/

};

* + - * [userprog/syscall.c]

struct *file* {

struct *inode* \*inode;

*off\_t* pos;

bool deny\_write;

};

filesys/file.c에 있는 파일 구조체 선언을 syscall.c에서도 쓸 수. ㅣㅆ도록 추가 선언해 주었다.

* + - * [threads/thread.c] > init\_thread()

#ifdef USERPROG

// Init File Descriptor

FOR(i, 128) t->FD[i] = NULL;

sema\_init(&(t->child\_lock), 0);

sema\_init(&(t->mem\_lock), 0);

sema\_init(&(t->load\_lock), 0);

list\_init(&t->child\_list);

list\_push\_back(&(running\_thread()->child\_list), &(t->child\_elem));

#endif

Thread initialization을 할 때 NULL 값으로 초기화해준다.

* + 1. **System Calls**
       - [userprog/syscall.c]

static void

syscall\_handler (struct *intr\_frame* \**f* *UNUSED*)

{

switch (\*(*uint32\_t* \*)(*f*->esp)) {

...

case SYS\_CREATE:

VERIFY\_ADDR(*f*->esp + 4);

*f*->eax = CREATE(\*(*uint32\_t* \*)(*f*->esp + 4), \*(*uint32\_t* \*)(*f*->esp + 8)); //

break;

case SYS\_REMOVE:

VERIFY\_ADDR(*f*->esp + 4);

*f*->eax = REMOVE(\*(*uint32\_t* \*)(*f*->esp + 4)); //

break;

case SYS\_OPEN:

VERIFY\_ADDR(*f*->esp + 4);

*f*->eax = OPEN(\*(*uint32\_t* \*)(*f*->esp + 4)); //

break;

case SYS\_FILESIZE:

VERIFY\_ADDR(*f*->esp + 4);

*f*->eax = FILESIZE(\*(*uint32\_t* \*)(*f*->esp + 4)); //

break;

case SYS\_READ:

VERIFY\_ADDR(*f*->esp + 4);

*f*->eax = READ(\*(*uint32\_t* \*)(*f*->esp + 4), \*(*uint32\_t* \*)(*f*->esp + 8), \*(*uint32\_t* \*)(*f*->esp + 12)); //

break;

case SYS\_WRITE:

*f*->eax = WRITE(\*(*uint32\_t* \*)(*f*->esp + 4), \*(*uint32\_t* \*)(*f*->esp + 8), \*(*uint32\_t* \*)(*f*->esp + 12));

break;

case SYS\_SEEK:

VERIFY\_ADDR(*f*->esp + 4);

SEEK(\*(*uint32\_t* \*)(*f*->esp + 4), \*(*uint32\_t* \*)(*f*->esp + 8)); //

break;

case SYS\_TELL:

VERIFY\_ADDR(*f*->esp + 4);

*f*->eax = TELL(\*(*uint32\_t* \*)(*f*->esp + 4)); //

break;

case SYS\_CLOSE:

VERIFY\_ADDR(*f*->esp + 4);

CLOSE(\*(*uint32\_t* \*)(*f*->esp + 4)); //

break;

...

}

}

syscall\_hadler() 함수의 Syscall switch 문 안에 이번 구현 System Call인 create, remove, open, close, filesize, read, write, seek, tell 경우를 추가한다. VERIFY\_ADDR() 매크로 함수를 이용하여 memory 접근 validation을 체크한다.

bool CREATE (const char \**file*, unsigned *size*) {

if(!*file*) EXIT(-1);

return filesys\_create(*file*, *size*);

}

파일의 이름과 크기를 인자로 받아 지정된 파일을 생성한다. (filesys\_create() 이용)

bool REMOVE (const char \**file*) {

if(!*file*) EXIT(-1);

return filesys\_remove(*file*);

}

파일의 이름을 인자로 받아 filesys\_remove()로 파일을 삭제한다. 파일 이름 validation을 체크해 EXIT(-1)로 예외 처리 하였다.

int OPEN (const char \**file*) {

// Check file validation

if(!*file*) EXIT(-1);

VERIFY\_ADDR(*file*);

lock\_acquire(&lock\_file);

struct *file* \*f = filesys\_open(*file*);

int res = -1;

if (!f) {

lock\_release(&lock\_file);

return res;

}

// Find empty FD and OPEN

FOR\_RANGE(i, 3, 128) {

if (!thread\_current()->FD[i]) {

if(!strcmp(thread\_current()->name, *file*)) file\_deny\_write(f);

thread\_current()->FD[i] = f;

lock\_release(&lock\_file);

return res = i;

}

}

lock\_release(&lock\_file);

return res;

}

파일 이름을 인자로 받아 filesys\_open()으로 파일을 연다. 파일을 열기 전 파일 이름 validation check을 해 준다.

int FILESIZE (int *fd*) {

if(!thread\_current()->FD[*fd*]) EXIT(-1);

// Check FD validation

if (!*fd*) return -1;

return file\_length(thread\_current()->FD[*fd*]);

}

File Descriptor를 인자로 받아, 해당 파일의 크기를 반환한다. File descriptor의 validation을 체크해 주고, file\_length()로 반환한다.

int READ (int *fd*, void \**buffer*, unsigned *size*) {

if(!*buffer*) EXIT(-1);

if(!thread\_current()->FD[*fd*]) EXIT(-1);

VERIFY\_ADDR(*buffer*);

if (!*fd*) {

lock\_acquire(&lock\_file);

FOR(i, *size*) \*((*uint8\_t* \*)*buffer* + i) = input\_getc();

lock\_release(&lock\_file);

return *size*;

} else if (*fd* > 2) {

lock\_acquire(&lock\_file);

int res = file\_read(thread\_current()->FD[*fd*], *buffer*, *size*);

lock\_release(&lock\_file);

return res;

} else return -1;

}

File Descriptor를 인자로 받아, 해당 파일에서 입력을 받는다. fd가 0인 경우, STDIN으로 입력을 처리하고, 나머지 경우 file\_read()로 수행한다.

int WRITE (int *fd*, const void \**buffer*, unsigned *size*) {

if(!*buffer*) EXIT(-1);

VERIFY\_ADDR(*buffer*);

if (*fd* == 1) {

lock\_acquire(&lock\_file);

putbuf(*buffer*, *size*);

lock\_release(&lock\_file);

return *size*;

} else if (*fd* > 2) {

if(!thread\_current()->FD[*fd*]) EXIT(-1);

lock\_acquire(&lock\_file);

int res = file\_write(thread\_current()->FD[*fd*], *buffer*, *size*);

lock\_release(&lock\_file);

return res;

} else return -1;

}

File Descriptor 값을 인자로 받아, 해당 파일에 buffer의 내용을 출력한다. FD가 0, 1 인 경우는 STDIN, STDOU, STDERR이므로 따로 처리하고, 2보다 큰 경우에 file\_write()를 수행한다.

void SEEK (int *fd*, unsigned *pos*) {

if(!thread\_current()->FD[*fd*]) EXIT(-1);

file\_seek(thread\_current()->FD[*fd*], *pos*);

}

file\_seek()을 통해 인자로 받은 File Descriptor 값에 맞는 파일 포인터의 위치를 pos로 바꾼다.

unsigned TELL (int *fd*) {

if(!thread\_current()->FD[*fd*]) EXIT(-1);

return file\_tell(thread\_current()->FD[*fd*]);

}

인자로 받은 File Descriptor 값에 대해 file\_tell()을 통해 그 파일의 offset 값을 unsigned 값으로 반환한다.

void CLOSE (int *fd*) {

if(!thread\_current()->FD[*fd*]) EXIT(-1);

int res = file\_close(thread\_current()->FD[*fd*]);

thread\_current()->FD[*fd*] = NULL;

return res;

}

인자로 받은 File Descriptor 값에 대한 파일을 file\_close()를 톨해 닫는다. FD[]의 값도 NULL 값으로 초기화 해준다.

* + 1. **Synchronizaton in Filesystem**
       - [threads/thread.h]

#ifdef USERPROG

/\* Owned by userprog/process.c. \*/

uint32\_t \*pagedir; /\* Page directory. \*/

/\* Added in #Proj 1 \*/

int exit\_status; /\* Exit status of process \*/

struct list child\_list; /\* List of child processes \*/

struct list\_elem child\_elem; /\* Element of child\_list \*/

// Semaphores

struct semaphore mem\_lock; /\* Lock of memory access \*/

struct semaphore child\_lock; /\* Lock of child process \*/

struct semaphore load\_lock; /\* Lock of load \*/

struct file\* FD[128]; /\* File Descriptor \*/

#endif

make check 기준, 다른 구현 사항은 비교적 평이하게 구현하였으나 multi-oom, syn-write, syn-read가 마지막까지 통과되지 못하였다. 이 원인을 분석하기 위해 [syn-read.c] 파일을 직접 찾아 문제가 되는 경우를 파악해 보았다.

void

test\_main (void)

{

pid\_t children[CHILD\_CNT];

int fd;

CHECK (create (file\_name, sizeof buf), "create \"%s\"", file\_name);

CHECK ((fd = open (file\_name)) > 1, "open \"%s\"", file\_name);

random\_bytes (buf, sizeof buf);

CHECK (write (fd, buf, sizeof buf) > 0, "write \"%s\"", file\_name);

msg ("close \"%s\"", file\_name);

close (fd);

exec\_children ("child-syn-read", children, CHILD\_CNT);

wait\_children (children, CHILD\_CNT);

}

결과 여러 thread가 동시에 자원에 접근할 때, 그 접근이 제어되지 않으면 예측하지 못한 결과나 오류를 일으키는(race condition) 문제 때문에 pass가 뜨지 못함을 알 수 있었다. 따라서 위 코드에 보이다시피 thread.h에 struct semaphore load\_lock을 추가하여 동기화 메커니즘이 작동되도록 구현하였다.

struct semaphore load\_lock을 추가해서, 부모 thread가 wait로 각 자식을 거둬들이기 전에 종료되는 경우를 막기 위해, sema\_up(), sema\_down()을 사용해 자식 thread가 load된 후 부모 실행하도록 설정하였다.

* + - * [userprog/syscall.c]

int OPEN (const char \**file*) {

// Check file validation

if(!*file*) EXIT(-1);

VERIFY\_ADDR(*file*);

lock\_acquire(&lock\_file);

struct *file* \*f = filesys\_open(*file*);

int res = -1;

if (!f) {

lock\_release(&lock\_file);

return res;

}

// Find empty FD and OPEN

FOR\_RANGE(i, 3, 128) {

if (!thread\_current()->FD[i]) {

if(!strcmp(thread\_current()->name, *file*)) file\_deny\_write(f);

thread\_current()->FD[i] = f;

lock\_release(&lock\_file);

return res = i;

}

}

lock\_release(&lock\_file);

return res;

}

여러 프로세스가 동시에 한 파일에 작성하는 것을 막기 위한 lock 처리를 한다. 또한 이미 실행중인 프로세스 자체에 쓰기 작업을 할 경우 오류를 일으키므로 이 경우에 대해서도 file\_deny\_write()로 처리해 준다.

int READ (int *fd*, void \**buffer*, unsigned *size*) {

if(!*buffer*) EXIT(-1);

if(!thread\_current()->FD[*fd*]) EXIT(-1);

VERIFY\_ADDR(*buffer*);

if (!*fd*) {

lock\_acquire(&lock\_file);

FOR(i, *size*) \*((*uint8\_t* \*)*buffer* + i) = input\_getc();

lock\_release(&lock\_file);

return *size*;

} else if (*fd* > 2) {

lock\_acquire(&lock\_file);

int res = file\_read(thread\_current()->FD[*fd*], *buffer*, *size*);

lock\_release(&lock\_file);

return res;

} else return -1;

}

여러 프로세스가 동시에 한 파일을 읽는 것을 막기 위한 lock 처리를 한다. 각 file\_read() 또는 input\_getc() 앞 뒤로 lock\_acquire(), lock\_release()를 한다.

* 1. **시험 및 평가 내용**

텍스트, 스크린샷, 멀티미디어 소프트웨어, 소프트웨어이(가) 표시된 사진

자동 생성된 설명80가지 Test Case를 모두 통과하였다.