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

담당 교수 / 분반 : 김영재 교수님 / 2분반

이름 / 학번 : 이민석 / 20201614

개발 기간 : 2024.10.27 ~ 2024.10.31

1. **개발 목표**

* 지난 project1에서 구현한 pintos의 system call, system call handler, argument passing 등을 바탕으로 file system 관련 system call(“create”, “remove”, “read”, “write”, “open”, “close”, “filesize”, “seek”, “tell”)을 추가적으로 구현해야 한다. “filesys.h”, “file.h”의 API를 통해 file system을 구현할 수 있으며, 각 thread마다 file descriptor set을 관리하고 사용할 수 있도록 해야 한다. 또한 Multithread 환경에서 발생할 수 있는 critical section문제를 회피하기 위해 lock과 semaphore 변수를 이용한 synchronization이 필요하다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

- File Descriptor는 열려 있는 파일을 추적하는데 사용되는 Integer ID이다. 이를 통해 user program은 file을 읽고 쓸 수 있게 된다. 또한 각 thread는 각각 자신의 file descriptor table(이후 코드에서 ‘fd\_set’으로 표현)을 가지며, 각 entry는 특정 파일에 대한 정보를 가리키며, 이를 통해 서로 다른 thread와 독립적으로 파일을 접근할 수 있게 된다. 이번 프로젝트에선 최대 128개의 file descriptor를 가질 수 있게 구현하였으며, “open()”을 통해 file을 열고, fd\_set의 빈 entry에 file 포인터를 저장한다. 또한 이외의 file system 관련 system call 함수들도 file descriptor를 통해 파일을 읽고 쓰며, 파일에 대한 정보를 응용할 수 있게 된다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

- 이번 프로젝트에서 새로 구현해야 할 System call function은 총 7개이다. ‘create()’, ‘remove()’함수를 통해 새로운 파일을 생성하거나 삭제한다. ‘open()’, ‘close()’함수를 통해 파일을 열거나 닫으며, 파일이 열린 file descriptor를 반환하거나 닫는다. ‘seek()’함수를 통해 인자로 넘어오는 file descriptor의 read/write 위치를 지정한 위치로 이동한다. ‘tell()’은 인자로 넘어오는 file descriptor의 다음 read/write 위치를 반환한다. “filesize()’는 인자로 넘어오는 file descriptor의 파일 크기를 반환한다. ‘read()’, ’write()’는 STDIN, STDOUT을 제외한 file descriptor의 파일에도 read/write를 할 수 있도록 추가로 구현해야 한다.

3. Synchronization in Filesystem

- Multi thread 환경에서 여러 thread가 동시에 한 파일에 접근할 때, critical section문제가 발생할 수 있다. 특히 여러 개의 thread가 “open()”, “read()”, “write()”와 같은 system call 함수를 실행한다면 동기화 문제가 발생할 수 있다. 이를 방지하기 위해 전역 변수 lock이 필요하다. 또한 “open()”함수에서 현재 실행중인 thread와 동일한 파일이 open 될 경우, 파일이 수정되지 않도록, file system API인 ‘file\_deny\_write()’를 호출하여 write를 방지해야 한다. 또한 위 과정에서 lock / unlock을 적절히 하여, Deadlock이 발생하지 않도록 해야 한다.

* 1. **개발 내용**

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

- File Descriptor Table을 관리하기 위해 각 thread마다 각자의 fd\_set array를 가지게 된다. 배열을 통해 fd\_set을 관리하여, ‘int fd’를 통해 빠르게 해당 인덱스에 접근하여 파일에 접근할 수 있게 된다. 또한 “Pintos” 문서에 따르면 필요하다면, 각 프로세스 당 최대 128개의 file을 open하기에 총 128 크기의 file pointer array를 설정하였다. 이를 통해 메모리도 효율적으로 관리할 수 있다.

2. System Calls: 구현할 각 system call에 대해 간략히 서술

- int read(int fd, void \*buffer, unsigned size): 현재 thread의 fd\_set의 index “fd”가 가리키는 파일에서 최대 “size”만큼 읽어들여 “buffer”에 저장한다. 읽은 byte 수를 return 한다.

- int write(int fd, const void \*buffer, unsigned size): 현재 thread의 fd\_set의 index “fd”가 가리키는 파일에 최대 “size”만큼 “buffer”의 내용을 기록한다. 읽은 byte 수를 return 한다.

- bool create(const char \*file, unsigned initial\_size): “file”이름을 가진 파일을 생성하고 파일 크기를 “initial\_size”로 한다. 파일의 생성 성공 여부를 true / false로 반환한다.

- bool remove(const char \*file): “file”이름을 가진 파일을 삭제한다. 파일의 삭제 성공 여부를 true / false로 반환한다.

- int open (const char \*file): “file” 이름으로 새로운 파일을 열게 되며, 이 파일을 가리키는 fd\_set의 index인 file descriptor를 반환한다. 파일이 열리지 않으면 -1을 반환하며, STDIN(0)과 STDOUT(1)은 새로운 파일의 fd로 설정하지 않는다.

- int filesize (int fd): 파일을 가리키는 fd\_set의 index인 “fd”를 통해, 열려 있는 파일의 크기를 byte 단위로 return 한다.

- void seek (int fd, unsigned position): 파일을 가리키는 fd\_set의 index인 “fd”를 통해, 파일의 read/write하는 위치를 “position”만큼 이동시킨다.

- unsigned tell (int fd): 파일을 가리키는 fd\_set의 index인 “fd”를 통해, 파일의 현재 read/write하는 위치를 byte 단위로 return한다.

- void close (int fd): 파일을 가리키는 fd\_set의 index인 “fd”를 통해, 해당 파일을 닫게 된다.

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

- 동시에 여러 thread가 file system에 접근할 경우 synchronization 문제가 발생할 수 있다. 따라서 전역 lock 변수를 사용하여 critical section 문제를 방지해야 한다. 따라서 ‘syscall\_init()’ 함수에서 이 변수를 초기화를 한 후, file system 관련 system call function들을 사용할 때, “lock\_aquire()”, “lock\_release()”를 적절히 사용하여 파일 작업이 진행 중일 때, 다른 thread가 접근하지 못하도록 막아야 한다. 또한 현재 실행중인 thread에 대해서 ELF file이 변경되는 것을 방지하기 위해, “open”에서 “file\_deny\_write()”를 사용하여 이를 방지한다.

- 또한 한 thread에서 자식 thread가 생성되었을 때, 자식 thread는 실행을 위해 필요한 file들을 load해야 한다. 이 때, 부모 thread는 자식 thread의 load 작업이 끝날 때까지 기다려야 하며, 자식 thread의 load 작업이 끝나기 전에 종료되면 안 된다. 따라서 자식 thread가 성공적으로 load 작업을 완료했는지 확인할 수 있는 변수와, 자식 thread가 load 작업을 할 동안 부모 thread가 기다릴 수 있도록 하는 semaphore 변수가 필요하다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**
   * 10.27 ~ 10.29: File system API의 전반적 이해, file system 관련 system call functions 구현.
   * 10.29 ~ 10.31: synchronization을 고려한 추가 요소 구현
   * 10.31 ~ 11.1: 보고서 작성
   1. **개발 방법**

1. File Descriptor:

- “threads/thread.h”에 구현되어 있는 struct thread 내부에 “struct file \* fd\_set[128]”, “int next\_fd”을 선언한다. 초기 fd\_set array는 0~127 file descriptor에 대해서 NULL값을 가지도록 “threads/thread.c”의 “init\_thread()”에서 초기화 한다. Pintos에선, STDIN(0), STDOUT(1)만을 고정 file descriptor로 정하고, 다음 file open시에 2번 인덱스부터 시작하므로, “next\_fd = 2”를 마찬가지로 “thread\_create()”에서 초기화 한다. 추가로 “userprog/syscall.c”의 “exit()”에서 thread가 종료 시, fd\_set의 모든 인덱스 file descriptor에 대해서 열려있는 파일에 대해 모두 “file\_close()”를 통해 닫아주도록 한다.

2. System Calls:

- 이전 프로젝트 1에서 구현한 system call과 마찬가지로, 새롭게 구현해야 하는 “create”, “remove”, “open”, “close”, “filesize”, “seek”, “tell”에 대해서 “userprog/syscall.h”에 선언한 후, “userprog/syscall.c”에 이를 구현한다. 또한 이전에 구현했던 “read()”는 STDIN(0)에 대해서만, “write()”는 STDOUT(1)에 대해서만 구현이 되어있으므로, 이를 모든 파일에 대해서 read/write를 수행할 수 있도록 수정해야 한다. 함수의 구현은 2-B-2를 참고하여, 각 함수가 적힌 내용을 적절히 수행할 수 있도록 코드를 구현해야 한다. 추가로, 구현하는 file system 관련 system call의 인자가 Invalid 한 Memory 영역에 접근하는지 확인하는 기능을 추가하기 위해 “userprog/exception.c”의 “page\_fault()”함수에 이를 구현한다.

3. Synchronization in Filesystem:

- thread가 ELF 파일을 load 하는 동안, 부모 thread가 먼저 종료하지 않고 기다려야 한다. 이러한 lock을 걸어주는 데에 필요한 “struct semaphore load\_lock”을 “threads/thread.h”에 구현되어 있는 struct thread 내부에 선언해야 한다. 또한 자식 thread가 성공적으로 ELF 파일을 load하였는지 확인할 수 있는 변수인 “bool load\_success” 변수를 선언한다. 또한 부모 thread를 가리킬 수 있도록 “struct thread \*parent”를 선언한다. 이 세 변수 모두 “threads/thread.c”의 “thread\_create()”에서 초기화 해준다. “userprog/process.c”의 “process\_execute()”에서 부모 thread는 생성된 자식 thread의 load작업이 잘 수행되었는지 “load\_success”변수를 통해 알 수 있고, 만약 성공하지 못했다면, -1을 return 한다. 자식 thread는 “start\_process()”에서 load를 성공적으로 마친 후, “load\_lock”변수를 “sema\_down()”을 통해 락을 해제하여, 다시 부모 thread가 실행될 수 있도록 한다.

- “userprog/syscall.c”의 전역 변수로 “struct lock file\_lock”을 선언하여 동시에 여러 process가 file system에 접근하는 것을 방지하는 변수를 생성한다. 사용하기 전에, “syscall\_init()”에서 “file\_lock”변수를 초기화 한다. 이를 “open()”, “read()”, “write()” 함수에서 함수 시작과 끝에 적절히 lock / unlock을 걸어 파일을 보호한다. 추가로 현재 실행중인 thread의 ELF 파일과 동일한 이름의 파일이 open 된다면, 이를 write하지 못하도록 “file\_deny\_write()”함수를 통해 구현한다. “file\_close()”함수 내에서 “file\_allow\_write()”가 실행되므로, “file\_deny\_write()”함수만 사용하였다.

1. **연구 결과**
   1. **Flow Chart**

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명1. File Descriptor:

텍스트, 도표, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명2. System Calls:

**텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 도표, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**3. Synchronization in Filesystem:

* 1. **제작 내용**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명1. File Descriptor:

struct thread 구조체 내부에 file descriptor를 관리하는 “struct file \* fd\_set[128]”과, file open시에 인덱스를 빠르게 찾기 위해 사용하는 “int next\_fd”를 선언한다. 이를 통해 thread는 file open시, 새롭게 저장되는 file descriptor 인덱스를 탐색을 통해 찾지 않아도 된다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명“threads/thread.c”의 “thread\_create”의 일부이며, ‘next\_fd = 2’를 통해 초기에 STDIN, STDOUT 다음 인덱스인 2 값을 가지도록 하였다. ‘t’는 현재 실행중인 thread를 가리키는 포인터이다.

“threads/thread.c”의 “init\_thread()”함수의 코드이며, 마지막 부분에 초기 fd\_set의모든 인덱스에 대하여 NULL로 초기화하는 부분이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명“userprog/syscall.c”의 “exit()”함수의 코드이다. thread가 system call function인 “exit()”을 호출하게 되면, STDIN, STDOUT을 제외한 나머지 file descriptor에 대하여 모두 “file\_close()”함수를 통해 파일을 닫게 된다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명2. System Calls:

이번 프로젝트2에서 새롭게 구현하는 file system 관련 system call function들을 “userprog/syscall.c”의 system call handler에 추가하는 코드이다. 각 함수의 인자들이 valid한 address에 위치한지 check한 후, type casting을 거친 후, 함수의 인자로 넘기며 system call을 호출한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명인자로텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명 넘어온 “file”의 이름으로 파일을 생성하게 되며, 초기 파일 크기는 “initial\_size”로 설정한다. “filesys/file.c”에 구현된 “filesys\_create()”함수를 이용하여 파일을 생성하고, 성공적으로 생성하였다면 true를, 그렇지 않다면 false를 return 한다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명인자로 넘어온 “file”의 이름으로 파일을 삭제한다. “filesys/file.c”에 구현된 “filesys\_remove()”함수를 이용하여 파일을 지우고, 성공적으로 삭제하였다면 true를, 그렇지 않다면 false를 return 한다.

인자로 넘어온 “fd”값이 0(STDIN)이 아닌 경우, 현재 실행중인 thread의 fd\_set에 해당 fd 인덱스가 가리키는 파일이 존재할 경우, 그 파일로부터 read하게 된다. “filesys/file.c”에 구현된 “filesys\_read()”함수를 이용하여 이를 수행한다. 읽어들인 값을 buffer에 저장하게 되며, 읽은 byte수를 return하게 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명인자로 넘어온 “fd”값이 1(STDOUT)이 아닌 경우, 현재 실행중인 thread의 fd\_set에 해당 fd 인덱스가 가리키는 파일이 존재할 경우, 그 파일에 buffer의 값을 write하게 된다. “filesys/file.c”에 구현된 “filesys\_write()”함수를 이용하여 이를 수행한다. write한 byte수를 반환한다.

우선 인자로 넘어온 “fd”값이 valid한 인덱스인지 또는 현재 실행중인 thread의 fd\_set에서 “fd” 인덱스가 pointing 하고 있는 파일이 존재하는지 확인한다. 만약 valid한 인덱스라면, “filesys/file.c”에 구현된 “file\_length()”함수를 이용하여 파일의 크기를 byte단위로 반환한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명우선 인자로 넘어온 “fd”값이 valid한 인덱스인지 또는 현재 실행중인 thread의 fd\_set에서 “fd” 인덱스가 pointing 하고 있는 파일이 존재하는지 확인한다. 만약 valid한 인덱스라면, “filesys/file.c”에 구현된 “file\_seek()”함수를 이용하여 파일의 read/write하는 위치를 “position”만큼 이동시킨다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명우선 인자로 넘어온 “fd”값이 valid한 인덱스인지 또는 현재 실행중인 thread의 fd\_set에서 “fd” 인덱스가 pointing 하고 있는 파일이 존재하는지 확인한다. 만약 valid한 인덱스라면, “filesys/file.c”에 구현된 “file\_tell()”함수를 이용하여 파일의 read/write하는 위치를 byte단위로 return한다.

“userprog/exception.c”의 “page\_fault()”의 일부이다. User program이 invalid한 address에 접근할 경우 page fault가 발생하는데, “kernel panic”을 띄우지 않고, exit(-1)을 통해 바로 종료하도록 코드를 추가하였다. 이를 위해 “threads/vaddr.h” 헤더파일을 추가한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명“file”이름에 해당되는 file을 open한다. “filesys/filesys.c”에 구현된 “filesys\_open()”함수를 이용하여 파일을 open하게 되며, 만약 새로운 file이 생성되지 않았다면 “-1”을 return 한다. 또한 현재 실행중인 file에 대해서 file이 새롭게 open된다면, 이를 방지하기 위해, “file\_deny\_write()”함수를 호출한다. 현재 실행중인 thread가 가지고 있는 next\_fd의 값을 return하기 위해 ‘int fd’라는 변수에 저장하고, fd\_set[next\_fd]에 새롭게 열게 된 파일의 포인터를 저장한다. 다음 file open을 위해 next\_fd++를 수행하는 코드가 마지막 부분에 적혀있으며, open 함수를 통해 열게된 file의 fd를 반환하게 된다.

현재 실행중인 thread의 fd\_set에서 “fd” 인덱스에 해당하는 file을 닫는다. 우선 인자로 넘어온 “fd”값이 valid한 인덱스인지 또는 현재 실행중인 thread의 fd\_set에서 “fd” 인덱스가 pointing 하고 있는 파일이 존재하는지 확인한다. 그 후 “filesys/file.c”에 구현된 “file\_close()”함수를 이용하여 파일을 닫고, 해당 fd 인덱스가 NULL값을 가지도록 하며 마무리한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명3. Synchronization in Filesystem:

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명- thread가 ELF 파일을 load 하는 동안, 부모 thread가 먼저 종료하지 않고 기다려야 한다. 이러한 lock을 걸어주는 데에 필요한 “struct semaphore load\_lock”을 선언하였다. 또한 자식 thread가 성공적으로 ELF 파일을 load하였는지 확인할 수 있는 변수인 “bool load\_success” 변수를 선언하였다.

“threads/thread.c”의 “thread\_create”의 일부이며, struct thread에서 선언한 semaphore변수와 load\_success를 초기화 하고 있다. 또한 실행중인 thread의 부모 thread를 가리키도록 설정하였다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

“userprog/process.c”의 “process\_execute()”함수의 일부이며, 생성된 자식 thread의 ELF file load 작업이 완료될 때까지 기다리기 위해 우선 “tid”에 해당하는 자식 thread를 찾는다. 자식 thread의 load semaphore 변수인 “load\_lock”을 “sema\_down()”함수를 이용하여 기다린 후, 자식 thread의 load 성공 여부를 “load\_success”를 통해 확인한다. 만약 성공적으로 load하지 못하였다면, -1을 return 한다. 또한 자식 thread가 비정상적으로 종료하였을 경우 process\_wait을 호출하여 해당 thread를 reaping하는 과정을 거친다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

“userprog/process.c”의 “start\_process()”함수의 일부이며, 아래 두 줄의 코드가 synchronization을 위한 코드이다. 자식 thread가 ELF file 로드를 완료한 후, 성공여부를 현재 “success”변수에 저장하고 있다. 이를 현재 실행 중인 thread(자식 thread)의 “load\_success”에 저장하여 부모 thread가 “process\_execute()”함수에서 알 수 있도록 하고, 자신의 “load\_lock”변수를 “sema\_up()”을 통해 부모 thread가 대기 상태에서 다시 실행될 수 있게끔 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명“userprog/syscall.c”의 “read()”함수 코드이다. 파일 시스템 API를 이용하여 file을 read하기 전, 함수 시작 부분에 전역 변수 “file\_lock”을 “lock\_aquire()”하며 lock을 걸어주고, read 작업이 끝난 후, “lock\_release()”를 통해 lock을 해제하게 된다.

“userprog/syscall.c”의 “write()”함수 코드이다. 파일 시스템 API를 이용하여 file에 write하기 전, 함수 시작 부분에 전역 변수 “file\_lock”을 “lock\_aquire()”하며 lock을 텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명걸어주고, write 작업이 끝난 후, “lock\_release()”를 통해 lock을 해제하게 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명“userprog/syscall.c”의 “open()”함수 코드이다. 파일 시스템 API를 이용하여 file을 open하기 전, 함수 시작 부분에 전역 변수 “file\_lock”을 “lock\_aquire()”하며 lock을 걸어주고, file open작업이 모두 끝난 후, “lock\_release()”를 통해 lock을 해제하게 된다. 또한 open하려는 file 이름이 현재 실행 중인 thread의 실행 파일과 동일할 경우, “file\_deny\_write()”를 통해 쓰기를 금지한다.

“userprog/syscall.c”의 “close()”함수 코드이다. 파일 시스템 API를 이용하여 file을 close하기 전, 함수 시작 부분에 전역 변수 “file\_lock”을 “lock\_aquire()”하며 lock을 걸어주고, file close작업이 모두 끝난 후, “lock\_release()”를 통해 lock을 해제하게 된다.

* 1. **시험 및 평가 내용**
* 텍스트, 스크린샷, 폰트이(가) 표시된 사진

  자동 생성된 설명make check 수행 결과를 캡처하여 첨부