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

담당 교수 / 분반 : 박성용 / 01

이름 / 학번 : 김태곤 / 20191583

개발 기간 : 10.26 ~ 10.29

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

Project1에 이어 미구현된 file system에 관련된 system call을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목을 구현해야 하는 이유, 혹은 구현 시 기대되는 결과를 간략히 서술

1. File Descriptor

File Descriptor는 파일을 참조할 때 사용하는 개념으로, 주로 0, 1, 2 등의 정수로 표현된다. 이들 번호는 표준 입력(stdin), 표준 출력(stdout), 표준 오류(stderr)를 참조한다. 나머지 번호는 열린 파일이나 I/O 장치를 참조하며, 프로그램이 해당 파일이나 장치와 상호 작용할 때 사용된다. File Descriptor 구현을 통해 파일을 열고, 파일 내용을 읽거나 쓸 수 있으며, 네트워크 통신을 수행할 수 있다. 또한 여러 input과 output을 다양하게 활용할 수 있다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

이번 프로젝트에서는 create, remove, open, close, filesize, read, write, seek, tell을 구현해야 한다. 위 System Call들을 통해 파일 및 데이터 관리, 입출력 작업, 데이터 검색, 수정 및 삭제와 같은 다양한 응용 프로그램에서 파일 시스템 및 데이터 관리를 효과적으로 제어할 수 있다.

3. Synchronization in Filesystem

여러 thread가 동시에 작업할 때 Synchronization 문제가 발생한다. 한 thread가 open을 통해 작업을 수행할 때, 다른 thread에서 open/ read/ write 작업을 수행하면 데이터가 원치 않게 변경될 수 있고, 이상한 값을 참조할 수 있다. 따라서 Synchronization 문제를 lock 등을 통해 다른 thread의 수행을 막아 해결할 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

File Descriptor는 간단하게 Array를 사용하였다. 전체 FD의 수가 128개로 정해져 있으므로 복잡하게 동적 할당할 필요 없이 정해진 크기로 선언하였다.

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)

Create(const char \*file, unsigned initial\_size) : 파일 시스템에 새로운 파일을 하는 함수로, 문자열 형식의 file 인자는 생성할 파일의 이름을 나타내고, initial\_size는 파일의 초기 크기를 나타낸다. 파일 생성에 성공하면 true를 반환하고, 실패하면 false를 반환한다.

Remove(const char \*file) : 파일 시스템에서 지정된 파일을 삭제하는 함수로, 문자열 형식의 file 인자는 삭제할 파일의 이름을 나타낸다. 파일 삭제에 성공하면 true를 반환하고, 실패하면 false를 반환한다.

Open(const char \*file) : 파일을 열고 해당 파일에 대한 파일 디스크립터를 반환하는 함수로, 문자열 형식의 file 인자는 열 파일의 이름을 나타낸다. 열린 파일에 대한 파일 디스크립터를 반환하며, 실패한 경우 -1을 반환한다.

Close(int fd) : 지정된 파일 디스크립터를 닫는 함수로, fd 인자는 닫을 파일의 파일 디스크립터를 나타낸다. 파일을 닫으면 해당 파일 디스크립터를 해제하고 파일과의 연결을 끊는다.

Filesize(int fd) : 열린 파일의 크기를 반환하는 함수로, fd 인자는 파일 디스크립터를 나타낸다.

Read(int fd, void \*buffer, unsigned size) : 파일 데이터를 읽는 함수로, fd는 읽을 파일의 파일 디스크립터를, buffer는 읽은 데이터를 저장할 버퍼를, size는 읽을 데이터의 크기를 나타낸다. 읽은 데이터의 크기를 반환하며, 실패한 경우 -1을 반환한다.

Write(int fd, const void \*buffer, unsigned size) : 파일 데이터를 쓰는 함수로, fd는 쓸 파일의 파일 디스크립터를, buffer는 쓸 데이터를 담고 있는 버퍼를, size는 쓸 데이터의 크기를 나타낸다. 쓴 데이터의 크기를 반환하며, 실패한 경우 -1을 반환한다.

Seek(int fd, unsigned position) : 파일 디스크립터의 읽기/쓰기 위치를 설정하는 함수로, fd는 대상 파일의 파일 디스크립터를, position은 설정하고자 하는 위치를 나타낸다.

Tell(int fd) : 파일 디스크립터의 현재 읽기/쓰기 위치를 반환하는 함수로, fd는 대상 파일의 파일 디스크립터를 나타낸다.

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

파일 시스템은 여러 thread나 process가 동시에 파일을 열고 읽거나 쓰는 등의 작업을 수행할 수 있다. 이로 인해 동시성 문제가 발생할 수 있으며, 이를 제어하기 위해 Lock이 필요하다. Lock\_acquire 호출을 통해 Lock을 획득한 후, 파일을 읽거나 쓰는 작업을 수행한다. 이때, Lock을 획득한 스레드는 해당 작업을 독점적으로 수행할 수 있으며, 다른 스레드는 Lock을 획득하기 위해 대기한다. 작업이 완료되면 lock\_release를 호출하여 Lock을 해제하고, 다른 thread가 해당 자원에 접근할 수 있도록 한다. 이를 통해 파일을 안전하게 열고, 파일에 데이터를 읽거나 쓰는 동안 다른 스레드가 간섭하지 않도록 할 수 있다.

또한 process excute 부분에서, thread를 생성하고 자식 thread가 load 되기 전 부모가 끝나버리는 경우도 생각해야 한다. 이때 Semaphore를 이용하여 이러한 상황을 막을 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10.26 : 프로젝트 이해

10.27 : FD 구현, System call 구현

10.28 : Synchronization 구현. 추가 오류 해결

10.29 : 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수

1) File Descriptor

File Descriptor를 구현하기 위해서 thread 부분을 조금 수정해야 한다. 먼저 Array 형태의 FD를 thread.h의 thread 구조체에 추가한다. 이때 Fd의 크기는 128로 고정되어 있다. 이후 thread.c에서 Fd array를 초기화 하는 작업을 추가한다.

2) System Call

Project1에서 구현한 System call 이외의 System call 함수들은 구현해야 한다. Syscall.c 파일에서 create, remove, open, close, filesize, read, write, seek, tell에 해당하는 함수를 구현한다. 추가로 exit 함수를 호출할 때 Fd를 초기화 하는 과정이 필요하므로 추가해준다.

3) Synchronization in File System

Synchronization을 위해 syscall.c에 lock struct를 추가한다. 이를 이용하여 read와 write, open 작업 전에 호출하여 다른 thread의 접근을 막고, 해당 작업이 종료될 때 다시 호출하여 Synchronization 작업을 수행한다.

thread를 생성하고 자식 thread가 load 되기 전 부모가 끝나버리는 경우를 해결하기 위해서 먼저 thread.h에서 load를 막는 semaphore를 thread 구조체에 추가한다. 이를 thread.c에서 초기화를 하고, process.c에서 해당 semaphore를 사용한다. Start\_process 과정에서 현재 thread의 load Semaphore를 up 시키고, Process\_excute 과정에서 현재 thread의 load semaphore를 down 시켜 Synchronization을 구현할 수 있다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성

1) File Descriptor

**텍스트, 라인, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

2) System Call

텍스트, 도표, 평행, 평면도이(가) 표시된 사진

자동 생성된 설명

3) Synchronization in File System

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1) File Descriptor

Thread.h 파일에 있는 thread 구조체에 struct file \*fd[128]를 추가한다. Thread.c 파일의 init\_thread에서 이전에 선언한 fd array를 초기화시킨다.

2) System Call

각 System call 함수들이 호출되기 전, syscall\_handler() 함수에서 넘겨받은 인자를 바탕으로 변수 주소의 유효성 is\_user\_vaddr을 통해 확인하고, 함수를 호출한다.

Create : parameter로 받은 file이 비어 있을 경우 exit(-1)을 반환한다. File이 유효한 경우 filesys\_create 함수를 호출한다.

Remove : parameter로 받은 file이 비어 있을 경우 exit(-1)을 반환한다. File이 유효한 경우 filesys\_remove 함수를 호출한다.

Open : 먼저 인자로 넘겨받은 file이 Null이거나, file의 주소가 유효하지 않을 경우 exit(-1)을 반환한다. 이후 lock\_acquire(&rw)을 통해 다른 thread의 추가 작업을 막아준다. 이후 filesys\_open을 통해 파일을 열어준다. 열린 file이 Null인 경우 lock\_release(&rw)를 통해 lock을 풀어주고, -1을 반환한다. 파일이 제대로 열린 경우, fd의 3번째 부터 탐색하면서 fd의 값이 존재를 파악한다. Fd의 값이 있는 경우, 현재 thread의 이름과 인자로 받은 file의 이름을 비교해 file\_deny\_write 작업을 해준다. 이는 write를 하기 위해서는 파일이 열려 있어야 한다를 구현하기 위함이다. 이 과정을 거친 후 Open 함수에서 현재 thread의 fd가 가리키는 것을 열린 파일로 지정한 후 lock\_release를 통해 lock을 풀어준다.

Close : 넘겨받은 fd 번호에 해당하는 fd가 비어 있을 경우 exit(-1)을 반환한다. Fd에 값이 있는 경우 Null값으로 초기화 하고, file\_close 함수를 호출한다.

Filesize : 넘겨받은 fd 번호에 해당하는 fd가 비어 있을 경우 exit(-1)을 반환한다. Fd에 값이 있는 경우 file\_length 함수를 호출한다.

Read : 먼저 인자로 넘겨 받은 buffer의 유효성을 검사하고, 유효하지 않을 경우 exit(-1)을 반환한다. 이후 lock\_acquire(&rw)를 통해 다른 thread의 작업을 막아준다. Fd가 0인 경우 stdinput이므로 input\_getc를 통해 입력을 받고, lock\_release(&rw)를 통해 lock 풀고 받은 값을 반환한다. Fd가 2보다 큰 경우, 현재 thread의 해당 fd가 비어있는 경우 lock을 풀고 exit(-1)을 해준다. 해당 fd가 존재하는 경우, File\_read를 통해 파일을 읽어준다. 이후 lock\_release를 통해 lock을 풀어주고 읽은 정보를 반환한다. 만약 fd가 0보다 작거나, 1인 경우는 Read에 해당하지 않으므로 lock을 바로 풀어주고 -1을 반환한다.

Write : 먼저 인자로 넘겨 받은 buffer의 유효성을 검사하고, 유효하지 않을 경우 exit(-1)을 반환한다. 이후 lock\_acquire(&rw)를 통해 다른 thread의 작업을 막아준다. Fd가 1인 경우 stdout이므로 putbuf를 통해 쓰는 작업을 하고, lock\_release(&rw)를 통해 lock을 풀고 write한 size를 반환한다. Fd가 2보다 큰 경우, 먼저 fd의 유효성을 검사한다. Fd가 Null인 경우 바로 lock을 풀고 -1을 반환한다. Fd가 존재하는 경우, file\_write 함수를 호출하고, lock을 풀고 그 크기를 반환한다. 그 외의 경우 write에 해당하지 않으므로 lock을 바로 풀어주고 -1을 반환한다.

Seek : 넘겨받은 fd 번호에 해당하는 fd가 비어 있을 경우 exit(-1)을 반환한다. Fd에 값이 있는 경우 file\_seek 함수를 호출한다.

Tell : 넘겨받은 fd 번호에 해당하는 fd가 비어 있을 경우 exit(-1)을 반환한다. Fd에 값이 있는 경우 file\_tell 함수를 호출한다.

3) Synchronization in File System

File System에서 Synchronization은 lock\_acquire(&rw)와 lock\_release(&rw)를 통해 구현하였다. 해당 lock은 syscall.c 파일에서 read, write, open에 적용되었다. 이를 통해 open, read, write 작업을 하는 경우, 다른 thread와의 충돌을 예방할 수 있었다. Lock의 적용은 4-B-2의 Syscall 함수 구현 설명에 자세히 나와있다.

위의 작업을 완료하면 multi-oom과 syn-read, syn-write만 fail하는 것을 확인할 수 있다. Syn-read와 write의 경우 process.c 파일에서 semaphore를 사용하여 해결하였다. thread를 생성하고 자식 thread가 load 되기 전 부모가 끝나버리는 경우를 해결하기 위해서 먼저 thread.h에서 load를 막는 semaphore를 thread 구조체에 추가한다. 이를 thread.c에서 초기화를 하고, process.c에서 해당 semaphore를 사용한다. Start\_process 과정에서 현재 thread의 load Semaphore를 up 시키고, Process\_excute 과정에서 현재 thread의 load semaphore를 down 시켜 Synchronization을 구현하였다.

위의 과정을 통해 syn-read와 syn-write는 해결하였지만, 아직 multi-oom이 해결되지 않았다. 메모리 누수를 다 막은 것 같지만 어느 부분에서 추가 문제가 발생하였는지 찾지 못하였다.

* 1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부
* 텍스트, 스크린샷, 흑백, 패턴이(가) 표시된 사진

  자동 생성된 설명

스크린샷, 텍스트, 디자인이(가) 표시된 사진

자동 생성된 설명

Make check를 통한 채점 결과 1개의 test case를 제외하고는 모두 통과하였다. Multi-oom을 통과하지 못하였는데, 어느 부분에서 메모리 누수가 발생했는지 끝내 찾지 못하였다.