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

담당 교수 / 분반 : 김영재 / 01 분반

이름 / 학번 : 김효림 / 20221549

개발 기간 : 5일(10.28~11.02)

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

File system 구현을 위해, system call 중 create, remove, open, close, filesize, seek, tell에 대한 system call handler를 구현한다. Proj 1에서 작성한 read, wrtie를 stdin, stdout 말고도 다른 fd 값이 들어왔을 때의 행동 양식을 추가해 주어야 한다. 이때 여러 개의 thread를 이용할 때 발생할 수 있는 synchronization 문제를 해결해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

프로세스 마다 file descriptor를 만들어, STDIN, STDOUT을 제외하고도 파일을 식별하여 읽고 쓸 수 있는 환경을 만들 수 있다. 만들어지는 프로세스마다 0번에 STDIN, 1번에 STDOUT, 2번에 STDERR을 할 수 있도록 하고, 그 이후 fd에 다른 프로세스를 할당하여 프로세스의 입출력을 가능하게 한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

새로 Create, Remove, Open, Close, Filesize, Seek를 구현해야 한다. 그리고 프로젝트 1에서 구현했던 Read, Write를 수정하여 STDIN, STDOUT을 제외한 입출력도 가능할 수 있도록 한다. 이 시스템 콜을 구현하여 file system에 필요한 동작들을 실행할 수 있다.

3. Synchronization in Filesystem

하나의 파일에 여러 프로세스가 동시에 접근할 경우 synchronization 문제가 발생할 수 있다. 예를 들어, ProcessA가 F에 읽기를 하고, ProcessB가 F에 쓰기를 한다고 하자. 이때 만약 ProcessA가 읽는 파일에 ProcessB가 write를 하게 되면, 작성자가 원하는 대로 결과가 나오지 않고 이상한 결과가 도출될 수 있다. 따라서 파일에 들어가고 행동을 수행할 때까지 다른 프로세스의 접근을 막기 위해 lock을 통해 문제를 해결해 주어야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

filesys.h에 미리 선언되어 있는 struct file\* fd[128]을 선언해 File Descriptor를 선언한다. Fd의 크기를 128로 설정한 이유는 핀토스 매뉴얼에 따르면 한 프로세스 당 open file의 최대 개수가 128개이기 때문이다. Struct file\* 자료형의 배열을 만들어 fd 값을 부여해 원소에 접근할 수 있게 된다.

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) bool create(const char\* file, unsigned initial\_size)

: file(파일 이름)과 initial\_size(초기 크기)를 인자로 받아 새로운 파일을 생성한다. 파일 생성에 성공하면 true를 반환하고, 실패하면 false를 반환한다.

2) bool remove(const char\* file)

: file(파일 이름)을 인자로 받아 해당 이름에 해당하는 파일을 삭제한다. 파일 삭제에 성공하면 true, 실패하면 false를 반환한다.

3) int open(const char\* file)

: file(파일 이름)을 인자로 받아 해당 이름에 해당하는 파일을 연다. 파일을 여는데 성공했다면 해당 파일의 fd 값을 반환하고, 실패했다면 -1을 반환한다.

4) int filesize(int fd)

: 주어진 file descriptor 값에 해당하는 열린 파일을 찾아서, 해당 파일의 크기를 반환한다. 파일을 찾을 수 없는 경우 exit(-1)로 예외 처리를 해 준다.

5) int read(int fd, void\* buffer, unsigned size)

: 주어진 file descriptor 값에 해당하는 열린 파일을 찾아서, buffer에 내용을 저장한다. 읽은 바이트 수(size)를 return 값으로 가지며, 실패한 경우 -1을 반환한다. fd 값이 0이면 STDIN을 진행한다.

6) int write(int fd, const void\* buffer, unsigned size)

: 주어진 filed descriptor 값에 해당하는 열린 파일을 찾아서, buffer로부터 size 만큼 내용을 작성한다. 작성한 byte size 만큼의 값을 return하며, 실패한 경우 -1을 반환한다. fd값이 1이면 STDOUT을 진행한다.

7) void seek(int fd, unsigned, position)

: 인자로 받은 position 만큼 fd에 해당하는 파일의 포인터 위치를 변경한다.

8) unsigned tell (int fd)

: fd에 해당하는 open된 파일의 현재 포인터 위치를 반환한다.

9) void close (int fd)

: 주어진 fd에 해당하는 열린 파일을 찾아 close한다.

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

우선 lock을 사용하는 경우, 동일한 파일에 여러 프로세스가 동시에 접근하게 되면 발생할 수 있는 문제를 해결하기 위해 사용한다. 따라서 파일에 접근할 때 lock\_acquire을 통해 lock을 걸고, read나 write의 행동이 끝나면 lock\_release를 통해 lock을 풀어 준다.

Semaphore는 프로세스 간 synchronization에 활용한다. 부모 thread가 자식 thread보다 먼저 죽는 것을 막기 위해 사용한다. 핀토스 프로젝트 내에 구현된 sema\_down()을 통해 임계 구역에 들어가고, sema\_up()을 호출하여 세마포어를 해제한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10/28 : file descriptor를 thread에 추가적으로 선언했다. 그 후, 구현해야 하는 System call들(create, remove, open, close, filesize, seek, tell, read, write)를 선언 및 수정하였다.

10/29 : lock을 이용해 file에 접근할 경우 발생할 수 있는 synchronization 문제를 해결하였다.

11/1 : syn-read, syn-write를 해결하기 위해 child thread를 찾아 synchronization 문제를 해결하였다.

11/2 : multi-oom 문제를 해결하기 위해 thread\_exit을 삭제하고 exit(-1)을 추가하였다. 그 후 보고서를 작성하였다.

* 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) ../src/threads/thread.h에 존재하는 thread 구조체에, file descriptor를 관리하기 위한 배열 struct \*file fd[128]을 선언하였다. 또한 추후 synchronization 문제를 해결하기 위한 load\_sema를 추가적으로 선언하였다. 이후 thread.c 파일의 init\_thread 내부에 fd를 초기화하는 부분, load\_sema를 sema\_init을 통해 초기화하는 부분을 추가하였다.

2) ../src/userprog/syscall.c에서 create, remove, open, filesize, read, write, seek, tell에 해당하는 system call을 구현하였다. 또한 switch 문에서 각 추가적인 handling case에 대한 부분을 추가하고, 각각의 함수가 가지는 인자들이 유효한지 valid\_addr을 통해 검사하였다. 이때 struct file 구조체를 활용하기 위해 filesys.h에 존재하는 struct file 구조체를 syscall.c에 똑같이 선언해 주었다.

또한 외에도 파일 접근에 대한 synchronization을 위해 syscall.c 함수 내에 struct lock file\_lock을 선언하고, syscall\_init 함수 내에서 이 함수를 초기화해 주었다.

3) kernel panic을 예방하고, page fault를 강화하기 위해 ../userprog/exception에 not\_present 조건을 추가하였다. 페이지가 메모리에 로드되지 않았을 때 발생하는 예외를 처리하기 위해 해당 조건을 추가했다.

4) ../src/userprog/process.c의 process\_execute 함수 내에서 부모 스레드가 먼저 죽는 synchronization 문제를 해결하기 위해 child 스레드를 찾아 load\_sema를 이용해 sema\_down으로 임계 구역을 설정해 주었다. 이를 통해 자식 스레드가 load된 후에 부모 스레드가 실행될 수 있도록 하여 synchronization 문제를 해결할 수 있다.

또한 start\_process 함수에서 load에실패할 경우 thread\_exit()이 아닌 exit(-1)을 통해 예외를 처리할 수 있도록 수정하였다.

1. **연구 결과**
   1. **Flow Chart**

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

1. File Descriptor

텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명

2. System calls

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

자동 생성된 설명

3. Synchronization in FileSystem

텍스트, 라인, 폰트, 스크린샷이(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**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- (../threads/thread.h)에 존재하는 thread structure에, file descriptor를 관리할 struct fild\*fd[129]를 선언하였다. 핀토스 매뉴얼에 따르면 프로세스 당 최대 128개의 파일을 오픈할 수 있기 대문에, 129개의 원소를 가질 수 있는 배열을 만들었다. 이때 사용한 strcut file 구조체는 (../filesys/file.c)에 구현되어 있다.

static void

init\_thread (struct thread \*t, const char \*name, int priority)

{

  enum intr\_level old\_level;

  ASSERT (t != NULL);

  ASSERT (PRI\_MIN <= priority && priority <= PRI\_MAX);

  ASSERT (name != NULL);

  memset (t, 0, sizeof \*t);

  t->status = THREAD\_BLOCKED;

  strlcpy (t->name, name, sizeof t->name);

  t->stack = (uint8\_t \*) t + PGSIZE;

  t->priority = priority;

  t->magic = THREAD\_MAGIC;

  old\_level = intr\_disable ();

  list\_push\_back (&all\_list, &t->allelem);

  intr\_set\_level (old\_level);

  #ifdef USERPROG

    sema\_init(&(t->wait\_sema), 0);

    sema\_init(&(t->exit\_sema), 0);

    sema\_init(&(t->load\_sema), 0);

    list\_init(&(t->child\_threads));

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

    for(int i=0; i<128; i++){

      t->fd[i] = NULL;

    }

  #endif

}

- (../threads/thread.c)에 존재하는 init\_thread 함수에서 0번부터127번까지의 128개의 fd에 NULL 값을 부여하여 초기화를 진행하였다.

**2. System Calls**

텍스트, 폰트, 스크린샷, 타이포그래피이(가) 표시된 사진

자동 생성된 설명

정의되어 있던 file 구조체를 syscall 내부에도 따로 선언해 주었다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

각 함수에서 사용한 주소 유효성 검사 함수이다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**-** (../userprog/syscall.c)에 우선 syscall\_handler에 각 API에 대한 행동처리 양식을 작성해 주었다. Proj1에서 구현했던 additional 부분에 대한 캡쳐는 생략하였다. 이전에 구현했던 부분이 아닌 create, remove, open, close, filesize, seek, tell에 대한 경우를 추가하였다. 이때 사용하는 함수가 인자를 가지는 경우 valid\_addr 함수를 통해 memory가 접근 가능한, 유효한 메모리인지 확인하였다. 그 후 return 되는 값들은 eax register에 값을 넣어주었다. 아래로는 각 함수에 대한 구현이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

파일의 이름과 초기 크기를 인자로 받아서 지정된 파일을 생성한다. 이때 file이 접근에 있어 유효한지 판단한다. 이미 구현된 filesys\_create()함수를 이용했다. 이 함수는 파일 이름과 파일 크기를 인자로 받아 디스크에 파일을 생성한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

파일의 이름을 인자로 받아 유효성을 검사하고, 구현되어 있는 filesys\_remove()로 파일을 삭제한다. 이 함수는 파일의 이름을 인수로 받아 해당 파일을 삭제하며, 할당된 디스크블록을 해제한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

파일의 이름을 인자로 받아 filesys\_open()으로 파일을 연다. 파일 이름이 유효한지 판단하기 위해 valid\_addr을 통해 검사한다. Filesys\_open 함수는 열고자 하는 파일의 이름을 인수로 받아 파일이 존재하면 해당 파일에 대한 포인터를 반환한다. (Synchronization에 대한 설명은 3번 부분에서 따로 설명하도록 한다.)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

fd 값을 인자로 받아, 해당 파일의 크기를 반환한다. 만약 현재 thread에서 해당하는 fd값이 NULL인 경우, 즉 할당이 되지 않은 경우 exit(-1)로 예외를 처리한다. 이때 핀토스 내부에 구현되어 이는 file\_length()를 이용한다. 이 함수는 file descriptor를 인자로 받아 해당 파일의 크기를 바이트 단위로 반환한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

fd 값을 인자로 받아, 해당 파일에서 입력을 받는다. fd가 0이라면 STDIN을 통해 입력을 받는다. fd 값이 2 이상인 경우 file\_read()를 이용해 파일에서 읽기를 진행한다. 이 함수는 지정된 크기만큼 데이터를 읽어와 버퍼에 저장한다. 그 후 오프셋을 업데이트하여 다음 읽기 위치를 유지한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

fd값을 인자로 받아, 해당 파일에 buffer의 내용을 출력한다. fd가 1인 경우 STDOUT에 해당 내용을 출력한다. fd값이 2보다 큰 경우 file\_write()를 이용한다. 이 함수는 buffer로부터 size 만큼의 데이터를 파일에 작성하고, 오프셋을 업데이트하여 이후 쓰기 작업이 이어질 수 있도록 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Seek 함수는 fd와 position을 인자로 받아, fd 값에 맞는 파일 포인터의 위치를 position으로 바꾸어 준다. 이때 file\_seek() 함수를 사용한다. 이 함수는 주어진 파일의 오프셋을 변경하는 함수이다. 파일의 현재 읽기/쓰기 위치를 position으로 변경한다.

Tell 함수는 fd를 인자로 받아, fd 값에 대해 file\_tell()을 이용하여 그 파일의 offset 값을 반환한다. File\_tell() 함수는 파일의 현재 위치를 나타내는 오프셋을 반환한다.

Close 함수는 fd 값을 인자로 받아, fd에 해당하는 파일을 file\_close()를 통해 닫는다. 그 후 닫은 파일에 대해 fd 값을 NULL로 초기화 한다. File\_close()함수는 파일을 닫고, 파일 시스템에서 해당파일에 대한 메모리를 해제한다.

**3. Synchronization**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

앞서 첨부한 ../threads/thread.h의 thread 구조체에 struct semaphore load\_sema를 선언하였다. 부모 스레드가 자식 스레드를 회수하기도 전에 먼저 죽어버리는 경우를 막기 위해 선언하였다. 이 load\_sema는 sema\_init을 통해 thread.c에 존재하는 init\_thread함수에서 초기화된다.

아래는 ../userprog/process.c 내부의 process\_execute 함수의 수정된 코드 부분만 따로 캡쳐한 사진이다. 자식 스레드 목록(child\_thread)를 돌며 tid가 새로 생성된 tid와 일치하는 자식 스레드를 찾아 해당 스레드에 sema\_down을 통해 현재 스레드(부모 스레드)를 대기시킨다. 자식 스레드가 비정상적으로 종료된 경우(exit status가 -1인 경우) process\_wait(tid)를 호출하여 자식 스레드가 종료될 때까지 기다릴 수 있도록 한다. 그 후 start\_process 함수에서 sema\_up을 통해 부모의 대기를 해제시킨다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

다음으로 ../userprog/syscall.c 내부에서는 파일 입출력에 대한 lock을 통해 synchronization 문제를 해결하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

우선 struct lock file\_lock을 선언하였고, syscall\_init 함수 내에서 lock\_init을 이용해 해당 함수를 초기화하였다. (아래 사진은 syscall.c에 선언된 open, read, write에 대한 함수 사진으로, 위에 언급된 것과 같다.)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

코드를 보면 알 수 있듯이, 여러 프로세스가 동시에 한 파일에 접근하는 경우 synchronization 문제가 발생할 수 있다. 따라서 파일에 접근할 때는 strcut lock file\_lock을 이용해, lock\_acquire()함수로 접근 권한을 얻는다. 그 후 모든 작업이 끝날 때는 lock\_release()를 통해 lock을 해제할 수 있다. Open 함수에서는 추가적으로, 이미 실행중인 프로세스 자체에 쓰기 작업을 진행할 때 file\_deny\_wrtie() 함수를 이용해 오류를 방지한다. 이 함수는 쓰기 권한을 거부할 파일을 인자로 받아, 해당 파일이 열려 있는 동안 해당 파일의 쓰기 권한을 차단한다.

* 1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부



