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

담당 교수 / 분반 : 김영재 교수님 / 2분반

이름 / 학번 : 박정원 / 20200183

개발 기간 : 2023/10/11 ~ 2023/10/13

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술
* **파일 디스크립터, 파일 시스템에 대한 시스템 콜 구현 ( create, remove, open, close, filesize, read, write, seek, tell) 및 공유 자원에 대한 동시 접근을 막기 위한 동기화 적용**

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor

: 파일 디스크립터를 구현해야 하는 이유는 파일을 open 하거나 create 할 시, 즉 파일에 접근할 때, 정수값을 갖는 파일 디스크립터를 통해 파일을 접근할 수 있기 때문이다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

1) create: 파일을 생성해주는 시스템 콜이다.

2) remove: 파일을 지워주는 시스템 콜이다.

3) open: 파일을 열어주는 시스템 콜이다.

4) close: 열려있는 파일을 닫아주는 시스템 콜이다.

5) filesize: 열려있는 파일의 크기를 알려주는 시스템 콜이다.

6) read: 인자로 넘겨진 사이즈 바이트만큼 파일을 읽어들여서 버퍼에 쓴다.

7) write: 인자로 넘겨진 사이즈 바이트만큼 버퍼에서 읽어들여 파일에 쓴다.

3. Synchronization in Filesystem

: 만약 두 개의 쓰레드가 같은 파일을 접근하여 read나 write 시스템 콜을 수행한다고 할 때, 시스템 콜은 적절히 동기화를 통해서 이 critical section에 대한 접근을 한 프로세스에 대해서만 제한해야 한다. 그러기 위해서는 동기화가 구현되어야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

핀토스에 미리 정의되어 있는 struct file 구조체의 포인터 배열을 선언하여 file descriptor를 구현할 수 있고, 커널에서의 파일 접근은 이 file 구조체를 통해서 가능하고, 파일 디스크립터를 이 구조체의 포인터 배열로 선언하면, 각각의 fd 정수값에 대해서 파일 접근을 할 수 있다.

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) create: create() 시스템 콜은 두 개의 인자를 받는데, 첫 째는 생성할 파일의 이름, 둘 째는 파일의 사이즈이다. 이 두 개의 인자를 시스템 콜 핸들러에서 filesys.h의 filesys\_create() 함수에 그대로 넘겨주며 구현할 수 있다.

2) remove: remove() 시스템 콜은 지울 파일의 이름을 인자로 받는데, 이를 시스템 콜 핸들러에서 filesys.h의 filesys\_remove() 함수에 넘겨주며 구현할 수 있다.

3) open: open() 시스템 콜은 인자로 받은 파일의 이름을 가진 파일을 여는데, filesys.h의 filesys\_open() 함수에 받은 인자를 그대로 넘겨줌으로써 구현할 수 있다.

4) close: close() 시스템 콜은 닫을 파일의 fd 정수 값을 인자로 받는데, 이를 받아서 해당 파일 디스크립터를 가진 파일을 닫아주는 역할을 한다. 이는 file.h의 file\_close에 앞서 선언한 file descriptor 배열의 fd 인덱스의 포인터를 넘겨주면 된다.

5) filesize: filesize() 시스템 콜은 인자로 받은 fd 정수 값을 가진 파일의 크기를 반환하는데, 이는 file\_length() 함수에 앞서 선언한 file descriptor 배열의 fd 인덱스의 포인터를 넘겨주면 된다.

6) read: read() 시스템 콜은 프로젝트 1번에서도 구현하였지만, 이번에는 인자로 온 fd 값을 가진 파일을 주어진 사이즈 바이트만큼 읽어 버퍼에 써야 한다. 즉, 앞서 언급한 struct file \* 배열의 fd 인덱스를 file\_read에 넘겨주면 된다. 나머지 버퍼와 fd 값은 동일하게 넘겨준다.

7) write: write() 시스템 콜도 프로젝트 1번에서도 구현하였지만, 이번에는 인자로 온 fd 값을 가진 파일에다가 버퍼에서 사이즈만큼 읽어들여 써야 한다. 즉, 앞서 언급한 struct file \* 배열의 fd 인덱스를 file\_write에 넘겨주면 된다. 나머지 버퍼와 fd 값은 동일하게 넘겨준다.

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

Critical section에 두 개 이상의 프로세스가 접근하는 것을 방지하기 위해서 동기화 기법을 사용해야 하는데, 이 기법들 중에는 lock과 semaphore가 있다. 우선, 기본적으로 critical section을 사용하려고 한 프로세스가 실행되면, 다른 프로세스는 해당 critical section에 대해서 접근하지 못하게 해야 한다.

Lock 같은 경우에는 lock\_acquire을 한 프로세스가 직접 lock\_release를 해주어야 한다는 특징이 있다. 따라서, 한 프로세스가 공유 자원에 접근하기 전에 lock\_acquire을 하고, 공유 자원 사용이 끝났다면 lock\_release를 통해서 해당 공유 자원을 다른 프로세스가 접근할 수 있도록 해준다.

Semaphore 같은 경우에는 위와 같은 상황에 사용하기 위해서는 처음 세마포어의 값을 1로 초기화 하고, 공유 자원을 사용하기 전에 sema\_down을 해서 0으로 만들어주고, 사용이 끝나면 sema\_up을 통해 다시 사용 가능한 자원의 개수를 1로 만들어준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2023-10-11 : 개발 시작 / 프로젝트 1번에서의 exit() 시스템 콜을 따로 함수로 만들고, 적용하기 / Switch() 문에 파일 시스템 시스템 콜에 대한 구문 작성하기 / 각 시스템 콜에 대한 매뉴얼 재확인 및 코드 점검

2023-10-12 : 각 시스템 콜에 대한 세부 구현 / critical section에 대한 접근 동기화 적용 / 세마포어, 락에 대한 학습 / 구현을 바탕으로 중간 점검 및 코드 오류 찾기

2023-10-13 : 개발 마무리 / 통과 못 하는 테스트 케이스에 대한 원인 분석 및 해결 / 최종 점검

* 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
* File descriptor 구현을 위해서 thread.h의 struct thread에 struct file \* fd[128] 을 선언해주었다. 각 프로세스 마다 파일 디스크립터를 가지고 있어야 하고, 매뉴얼에 이는 128가 최대가 되도록 하라고 했으므로 128개 선언해준다. 또한, thread.c의 init\_thread에서 선언한 128개의 file 포인터를 NULL로 초기화해준다.
* syscall.c에서 새로 추가할 시스템 콜들에 대한 시스템 콜 넘버를 switch 문에 추가한다. 새로 추가한 시스템 콜에 대해서도 user\_valid\_addr를 통해서 메모리 접근에 대한 validity 확인을 한다. 이번에 추가하는 시스템 콜들은 파일 시스템에 관한 시스템 콜들이기에 앞서 선언해주었던 파일 디스크립터 포인터에 대한 validity도 user\_valid\_addr를 통해서 체크를 해주고, 찾고자 하는 파일의 파일 디스크립터가 NULL로 초기화되어 있다면, exit(-1)을 시켜주도록 예외 처리를 잘 해주어야 한다. 기존에 stdin과 stdout으로만 구현되어 있는 read랑 write에도 file descriptor 부분을 추가해준다. 이번 프로젝트 2번을 하면서, exit() 시스템 콜과 close() 시스템 콜은 각각 sys\_exit, sys\_close 라는 함수로 따로 빼서 만들어주었다. sys\_exit 함수의 구현 부분을 살짝 수정했는데, 스레드가 Exit 되기 전에 file descriptor 테이블을 모두 null 로 초기화시켜주었다. 또한, 프로세스가 종료되기 전에 자식 프로세스를 전부 종료한 뒤에 종료시켰다. (좀비 프로세스, 고아 프로세스 방지)
* open(), read(), write()에 대해서 synchronization 기법을 적용하여 critical section 에 대한 동시 접근을 방지하였는데, 이를 위해서 syscall.c 에 1로 초기화되는 세마포어를 선언해주었다. 세마포어의 초기화는 syscall\_init() 함수에 추가해주었다. 그리고 위 세 가지의 시스템 콜을 수행하기 전에 sema\_down을 해주고, 해당 작업이 끝났으면 다시 sema\_up을 시켜주는 방식으로 구현하였다.

1. **연구 결과**
   1. **Flow Chart**

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

1. **File Descriptor**

**친필, 텍스트, 폰트, 스케치이(가) 표시된 사진

자동 생성된 설명**

1. **System Calls**

**텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명**

read(), write(), open() 시스템 콜은 바로 밑 동기화 파트에 포함시켰습니다.

1. **Synchronization in File System**

**스케치, 텍스트, 폰트, 화이트이(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

1. **File Descriptor**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* 먼저 Pintos에서 구현되어 있는 file 구조체이다. 커널에서의 파일 접근은 이 file 구조체를 통해서 가능하다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* thread.h 에서 thread 구조체 선언부에 struct file \* fd[128]을 추가하였다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

* thread.c 에서 init\_thread() 함수에 각 fd[i] file 구조체 포인터를 NULL로 초기화해주었다.

이제 각 쓰레드별로 파일 디스크립터를 통한 파일 접근이 가능하다.

1. **System Calls**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

우선, create() 시스템 콜이 받아오는 인자는 두 개이고, 각각 파일의 이름과 사이즈이다. user\_valid\_addr를 통해서 받아온 파일의 이름의 주소 값 validity를 체크해준다. 만약 NULL 값이라면 sys\_exit(-1)를 통해 강제 종료시킨다. 그리고 그대로 받아온 인자 두개를 filesys\_create 함수의 인자로 넘겨주어 시스템 콜을 수행한다. filesys\_create() 함수는 filesys.c에서 제공되는 내장 함수로서, create() 시스템 콜과 같이 받아온 파일의 이름을 해당 사이즈 바이트만큼 생성하는 함수이다. 반환 값은 성공 여부에 따른 bool 값이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

remove() 시스템 콜에서 받아오는 인자는 파일 이름 char 배열이다. 따라서, 이 주소에 대한 validity를 user\_valid\_addr를 통해 확인하고, NULL 주소이면 sys\_exit(-1)를 통해 강제 종료시킨다. filesys\_remove()에 그대로 인자를 넘겨주어 시스템 콜을 수행한다. Filesys\_remove() 함수는 filesys.c에 정의되어 있는 내장 함수로서, remove() 시스템 콜의 요청대로 해당 이름의 파일을 지워주는 함수이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

close() 시스템 콜은 하나의 인자를 받아오는데 이는 닫을 파일의 fd 정수 값이다. 고로, sys\_close()에 해당 인자 값을 넘겨주면 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

sys\_close는 여러 함수에서 자주 사용되는 부분이라 함수로 따로 만들어주었다.

우선 받아온 fd 정수 값, 즉 닫으려고 하는 파일의 인덱스 값에 대해서 해당 인덱스의 파일 포인터가 NULL인지 확인하고, 맞으면 sys\_exit(-1)를 통해 강제 종료 시킨다. 그 이유는 닫으려는 파일의 포인터가 NULL인 것은 있을 수 없는 일이기 때문이다. 해당 조건문에 해당 하지 않는다면, 닫으려는 파일의 포인터 값을 새로운

struct file \* 에 옮겨주고, NULL로 바꾸어준다. 그리고 file\_close() 시스템 콜을 호출한다. file\_close() 함수는 file.c에 정의되어 있는 내장 함수로, close() 시스템 콜의 요청대로 해당 파일을 닫아주는 함수이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

filesize() 시스템 콜은 인자로 받아온 fd 정수 값을 인덱스로 가지는 파일 디스크립터의 파일 사이즈를 반환한다. 이 때, 해당 인덱스를 가지는 파일 포인터가 NULL 이면 앞서와 마찬가지로 sys\_exit(-1)로 강제종료 시킨다. 그리고 아니면, file\_length() 함수를 통해서 사이즈 바이트를 받아온다. file\_length() 함수는 해당 파일의 크기를 바이트 기준으로 반환해주는 file.c에 정의되어 있는 내장 함수이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

seek() 시스템 콜은 열린 파일의 read하거나 write할 때의 Offset을 인자로 받아온 position으로 옮겨주는 시스템 콜이다. 첫 번째 인자로는 Fd 값이, 두 번째 인자로는 position 정수 값이 들어온다. 앞의 시스템 콜과 동일하게 해당 파일의 포인터 NULL 체크를 해주고, file\_seek() 시스템 콜을 호출한다. file\_seek() 함수는 file.c에 정의되어 있는 내장 함수로서, 해당 파일의 offset을 position 값으로 옮겨준다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

tell() 시스템 콜은 fd 값을 가지는 파일의 다음 읽거나 쓸 바이트의 위치를 반환해준다. 앞의 시스템 콜과 동일하게 fd 값을 가지는 파일의 포인터 null 체크를 한 후에 file\_tell() 함수에 인자를 동일하게 넘겨주며 호출한다. file\_tell() 함수는 file.c에 정의되어 있는 내장 함수로서, 해당 파일의 다음 읽거나 쓸 바이트의 위치를 반환한다.

read(), write(), open() 시스템 콜은 아래 3번 파트에서 파일 시스템 동기화와 함께 설명하겠습니다.

1. Synchronization in File System

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

자동 생성된 설명

먼저, open() 시스템 콜은 따로 세마포어를 이용한 동기화 처리가 필요했다. 그 이유는 read()와 write() 시스템 콜의 critical section에 대한 동시 접근을 막는 것이 주된 동기화의 목적인데, 일단 read()와 write()를 하기 위해서는 파일을 open 하는 과정이 반드시 수반되기 때문이다. 우선, 열려고 하는 파일의 char 포인터에 대한 validity를 체크하고, 문제 없다면 sema\_down(&IO)를 통해 공유 자원에 접근 가능한 프로세스의 수를 나타내는 세마포어의 값을 1 내려주고 open 작업을 수행한다. 이때, IO 세마포어는 syscall\_init() 함수에서 1로 초기화된 세마포어 값이다. 현재 실행중인 프로세스가 sema\_down을 하게 되면, 다른 프로세스가 실행될 때, sema\_down을 만나면 실행하지 못하고 sema\_up이 될 때까지 기다리게되므로, Critical section에 대한 동시 접근을 막을 수 있다. 그 후 filesys.c에 정의되어 있는 filesys\_open 함수를 통해서 해당 파일을 연다. filesys\_open 함수의 반환 값은 open 한 new file에 대한 포인터다. 여기서 주의할 점은 만약 현재 프로세스가 open 하려고 한 파일의 이름이 현재 프로세스의 이름과 같다면, 해당 파일에 대한 write 를 Deny 해야 한다. PPT에도 나와있듯이, 현재 프로그램의 executable을 modify 하거나 delete하는 것을 막아야 하기 때문에 해당 처리를 해준 것이다. 이제 filesys\_open을 통해 새로 open 한 파일의 포인터 값을 fd 값 중에서 NULL인 인덱스 중 가장 빠른 인덱스에 연결해주고 break 해준다. 그리고 마지막에 sema\_up을 통해 다른 프로세스의 접근을 허용해주면 된다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

write() 시스템 콜도 open() 함수와 동일한 이유와 동일한 방법으로 동기화를 적용해주었다. 먼저, 첫번째 인자, 즉 fd 값이 1인 경우 stdout이라는 뜻이므로, putbuf 내장 함수를 통해 화면에 버퍼를 출력해준다. 이는 프로젝트 1번에서 설명했으므로 간략히 넘어가고, 이번 프로젝트에서 추가한 부분은 fd 값이 1이 아닌 경우이다. 우선, 인자로 받아온 fd 값이 3이상인 경우에 대해서 file\_write() 함수를 호출해주었는데, 이는 fd 0, 1, 2는 각각 stdin, stdout, stderr로 고정되어 있기 때문이다. 쓰려고 하는 파일의 포인터가 NULL이면, sema\_up을 시켜준 후 sys\_exit(-1)을 한다. 아닌 경우에만 file\_write 내장 함수를 통해서 버퍼의 내용을 해당 fd 값을 가진 파일에 인자로 주어진 바이트만큼 쓴다. 그 후 sema\_up을 해준다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

read() 시스템 콜은 write() 시스템 콜의 구현과 매우 유사하다. 인자들이 저장되어 있는 포인터들에 대한 validity를 체크 하고, sema\_down(&IO)을 통해서 다른 프로세스의 공유 자원 접근을 막는다. Fd 값이 0인 경우 stdin인 경우이므로 프로젝트 1번에서 설명했듯이 input\_getc() 내장 함수를 통해서 사용자로부터 입력을 받은 것을 버퍼에 저장한다. 나머지 1, 2 fd 값에 대한 경우는 각각 stdout과 stderr의 값이므로 예외 처리하고, fd 값이 3이상인 경우에 대해서는 똑같이 읽으려고 하는 파일의 포인터가 NULL인지 체크하고, 맞으면 sema\_up을 하고, sys\_exit(-1)로 강제종료 시킨다. 아닌 경우 file\_read 내장 함수를 통해서 해당 요청을 그대로 처리하면 된다. file\_read 함수는 read() 시스템 콜과 동일하게 파일로부터 입력을 읽어들여 버퍼에 저장한다. 반환 값은 읽어들인 바이트의 수이다.

* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
* 추가적으로, 프로젝트 1번에서 두 개의 세마포어를 가지고, process\_wait()을 구현하였는데, 이번 프로젝트를 하면서 process\_execute() 부분에 세마포어를 추가할 필요성이 생겼다. 그 이유는 테스트 케이스에서 exec() 시스템 콜을 실행하는 테스트 케이스에 대해서 통과되지 않아 exec() 매뉴얼을 찾아보았는데, 자식 프로세스가 “로드”가 끝날 때까지 부모 프로세스가 return을 할 수 없다고 써져있고, 이를 위해서 동기화가 필요하다고 써져있다. 따라서, thread.h에 struct thread 구조체에 struct semaphore ld를 추가해준다. thread\_execute()에서 thread\_create()를 호출한 다음에 sema\_down을 해주고, 자식 프로세스가 load가 끝나면 자신의 부모 프로세스의 ld 세마포어를 sema\_up 해주어 process\_execute()로 부터 return 할 수 있도록 해준다. 이를 위해선 자식 프로세스도 부모 프로세스를 알고 있어야 하고, 이를 위해서 thread.h에 있는 thread 구조체에 struct thread \* par 도 추가해준다. 추가적으로, 프로세스가 로드에 실패했는지를 확인하는 int load\_fail\_flag를 struct thread에 추가해준다. ld 세마포어에 대한 초기화는 thread.c의 init\_thread()에 추가해준다.

**텍스트, 친필, 폰트, 서예이(가) 표시된 사진

자동 생성된 설명**

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

**위에 설명 이해를 위해 필요한 thread 구조체의 선언부와 초기화 부분이다.**

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

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명