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

담당 교수 / 분반 : 김영재 교수 / 2분반

이름 / 학번 : 고동헌 / 20191564

개발 기간 : 10-09 ~ 10-15

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술
* 이번 프로젝트에선 Project1에서 구현하지 않은 나머지 모든 system call을 구현하는 것이 목적이다. 남은 system call은 open, create, close 등 파일 관련 system call들이며, 여러 개의 thread가 concurrent하게 system call을 호출하였을 때 race condition 이 발생하지 않도록 synchronization을 잘 처리해줘야 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

2. (이번 프로젝트에서 구현해야 하는) System Calls

3. Synchronization in Filesystem

각 User Program 은 file을 생성할 수 있어야 하고, 생성된 file을 접근할 수 있어야 한다. Pintos에서 file에 대한 접근은 file struct를 통해서 할 수 있다. File struct는 내부적으로 디스크에 있는 해당 파일의 inode struct에 대한 포인터를 가지고 있고, 최종적인 file write, read 등은 inode를 통해서 이루어진다. 하지만 inode 관련 부분은 file system에서 다루는 내용이고, 이번 프로젝트에서는 이미 구현돼 있는 file system api 를 사용하면 된다.

위에서 설명하였듯, file 접근은 file struct를 통해 할 수 있다. 따라서 각 user process는 file struct구조체에 대한 배열을 가지고 있어야 한다. 이 배열을 통해, user가 특정 파일에 대한 read, write 등을 요청할 때 kernel은 user 가 넘겨준 정보를 받아 file struct에 접근하여 요청을 수행할 수 있다. 이 때 file struct 배열의 특정 원소를 접근하기 위해 필요한 것이 file descriptor이다. User가 file open 을 요청하면 file struct 배열 중 비어있는 원소에 해당 file 에 대한 struct를 만들어 mapping 시키고, 배열의 index를 return 해준다. Open 이후 user는 return 받은 file descriptor 값을 system call에 인자로 전달해 주어, 해당 파일에 대한 read/write 요청을 할 수 있다.

이번 프로젝트에서는 총 7개의 system call을 추가적으로 구현하여야 한다. 또한, 프로젝트 1에서 stdin, stdout 에 대한 처리만 가능하였던 read/write system call을 file read/write까지 가능하도록 확장하여야 한다. 각 system call 구현 시 기대되는 결과는 다음과 같다.

* + 1. **create**
* create system call의 구현을 통해, user program은 disk에 file을 생성할 수 있게 된다. 이 때 이 file의 이름 및 크기는 user가 전달한 parameter로 정해진다. 해당 file 이름이 이미 존재할 경우, false를 return 한다.
  + 1. **remove**
* remove system call의 구현을 통해, user program은 disk에 있는 file 을 삭제할 수 있게 된다. 만약 user가 전달한 file name을 갖는 file이 존재하지 않는 경우 false를 return 한다.
  + 1. **open**
* Open system call을 호출하면 kernel은 해당 file에 대한 file struct를 만들어 user program kernel thread 구조체의 file struct 배열에 mapping을 시키고, mapping 된 배열의 index를 return 해준다. 이를 통해, program은 disk에 있는 file에 대한 read, write, seek등의 연산을 수행할 수 있게 된다.
  + 1. **filesize**
* filesize system call을 통해 user는 open한 file의 size를 알 수 있게 된다. File의 size는 inode struct에 저장 돼있기 때문에 내부적으로 inode\_length 함수를 호출해서 user에게 return 한다.
  + 1. **Read**
* 프로젝트 1에선 read를 통해 stdin을 통한 입력만 가능하였다. 프로젝트 2에선, read를 통해 user가 open한 file에 대한 read가 가능해진다.
  + 1. **write**
* read 와 마찬가지로, 기존의 write는 stdout을 통한 출력만 가능하였다. 추가적인 구현을 통해 user는 open한 file에 대한 write가 가능해진다.
  + 1. **seek**
* user 가 open file에 대해 read/write를 할 때 다음으로 읽거나 쓸 byte 위치를 바꾸는 함수이다. 이 함수를 통해 user는 해당 파일을 중간까지 읽고 다시 처음부터 읽거나 하는 등의 작업을 수행할 수 있다.
  + 1. **tell**
* 해당 open file에 대해, 다음으로 읽거나 쓸 byte의 위치를 return 해주는 함수이다. seek 함수와 tell 함수 모두 file struct 내의 pos 멤버의 값을 이용한다.
  + 1. **close**
* user가 open한 file에 할당된 file struct를 free 시켜주고, inode의 reference count를 감소시켜준다. 이 때 해당 파일에 대해 마지막으로 close를 했을 때, 해당 파일이 제거됐으면 파일과 관련된 inode를 free 시켜준다.

프로젝트 2에서 구현해야 하는 system call들은 모두 내부적으로 file system에 구현된 api를 호출한다. 이 때 File system에서는 inode, file struct 등의 kernel 자료구조 및 disk의 file에 접근한다. 만약 여러 개의 thread가 동시에 file system을 사용하면서 concurrent하게 shared memory에 접근하는 상황이 발생하면 문제가 생길 수 있다. 이러한 race condition을 방지하기 위해, 특정 thread가 file system과 관련된 critical section에 들어갈 경우 synchronization을 통해 mutual exclusive access를 보장해줘야 한다. pintos에서는 lock을 통해 하나의 thread가 file 관련 작업이 완전히 끝날 때까지 다른 thread가 해당 critical section에 들어가지 못하도록 synchronization이 가능하다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

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)

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

User가 open file에 접근하기 위해선, file descriptor가 필요하다. kernel 이 user가 전달한 file descriptor를 통해 file struct에 O(1)로 접근하기 위해서, file struct 구조체에 대한 pointer를 배열로 선언하였다. File struct가 아닌 pointer들의 배열을 선언한 것은 kernel stack overflow를 방지하기 위한 것이다. 이 때 pintos manual에 따라 각 process당 가질 수 있는 file struct의 개수를 128로 설정하였다. 이번 프로젝트에서 수정 및 추가한 system call은 총 9개이다. 아래는 각 system call에 대한 설명이다.

* + 1. **create**
* user가 전달한 이름과 크기를 갖는 file을 생성해주는 함수이다. 내부적으로 filesys\_create 함수를 호출해 해당 file에 대한 inode를 만들어 disk에 할당하며, 성공적으로 생성됐을 경우 true를 return해준다.
  + 1. **remove**
* user가 전달한 이름을 갖는 file을 삭제해준다. 내부적으로 filesys\_remove 함수를 호출하여 해당 file에 관련된 inode의 remove flag 를 set 해주고, inode\_close함수를 호출한다. Remove를 호출한 process 외에 이 file을 open한 process가 없을 경우, disk에서 file 관련 구조를 free 시켜준다. 성공적으로 제거됐을 시 true를 return 한다.
  + 1. **open**
* user가 전달한 file에 대한 file struct를 생성해 해당 process의 kernel file descriptor table에 mapping 시켜주고, mapping된 index를 user에게 return 해준다. 성공적으로 open됐을 경우 3이상의 fd 값이 return돼고, 아닐 경우 -1이 return 된다. 내부적으로 filesys\_open 함수를 호출한다.
  + 1. **filesize**
* user가 open한 file에 대한 file 크기를 return 해주는 함수이다. 내부적으로 file\_length 함수를 호출하여 inode에 들어있는 file 의 크기를 받아온다. 이 때, user가 전달한 file descriptor는 반드시 open된 file에 대한 descriptor여야 한다. 아닐 경우 해당 process를 종료시킨다.
  + 1. **seek**
* User가 open한 file의 next to be read / next to write 의 offset을 변경해주는 함수이다. 내부적으로 file\_seek 함수를 호출하며, 이 함수에서 file struct를 통해 offset을 변경해준다. filesize 함수와 마찬가지로 user가 전달한 file descriptor는 반드시 open된 file이어야 한다.
  + 1. **tell**
* user가 open한 file의 next to be read / next to write를 가리키는 offset을 return 해주는 함수이다. 내부적으로 file\_tell 함수를 호출하며, file struct의 pos변수에 접근에 값을 알아낸다. 위와 마찬가지로 file descriptor는 반드시 유효한 값이어야 한다.
  + 1. **close**
* 내부적으로 file\_close 함수를 호출해 user가 open한 file에 mapping된 file struct를 free시켜준다. 위에서 fd를 전달하는 함수들과 마찬가지로 file descriptor는 반드시 open 된 file을 가리켜야 한다.
  + 1. **read**
* 프로젝트1 에서 확장시킨 system call 이다. Stdin을 통한 입력뿐만 아니라 open한 file을 통해서도 입력 받을 수 있도록 해준다. 내부적으로 file\_read 함수를 호출하여 읽어온 byte 수를 return 받는다.
  + 1. **Write**
* read 와 마찬가지로, 프로젝트 1에서 확장시킨 system call이다. 내부적으로 file\_write함수를 호출해 open된 file에 대한 write가 가능하다. 이 때 현재 실행중인 thread에 대해서 write를 수행하는 경우를 방지하여야 하는데, 이는 file\_write\_deny함수를 통해 가능하다. 아래 문항에서 자세히 설명하겠다.

file system을 사용할 때 race condition이 발생하지 않도록 하기 위해선, system call handler에서 file system api를 호출할 때마다 synchronization을 해주어야 한다. 여러 개의 process들이 하나의 inode에 대해 concurrent하게 접근하면서 값을 변경할 수 있기 때문이다. 따라서 critical section은 file system api를 호출하는 부분이다.

이 부분에 대한 protection은 pintos 내부 자료구조인 lock을 통해서 할 수 있다. Lock은 기본적인 semaphore에 더불어, 추가적인 제약사항이 있다. Semaphore의 경우 어떤 thread가 특정 semaphore에 down operation을 해 block 상태에 들어갔을 때, 해당 semaphore에 접근할 수 있는 어떤 thread던 sema up 연산을 통해 block된 thread를 release 시켜줄 수 있다. 하지만 lock 같은 경우, block 된 thread를 release해줄 수 있는 thread는 오직 해당 lock을 acquire한 thread뿐이다. File system에서는 하나의 thread가 완전히 작업을 끝날 때까지 다른 thread가 block된 thread를 release해주면 안되므로, semaphore가 아닌 lock을 활용하였다.

Lock을 이용해, file system api를 호출하는 부분 앞뒤로 lock\_acquire, lock\_release를 수행하면 하나의 thread가 file system 접근할 때 다른 thread와 interleaving 되지 않는 것이 보장된다. 이를 통해 file system의 synchronization을 수행할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

**(I) kernel panic error debugging : 10/09~10/11**

**(II) system call implementation : 10/11~10/14**

**(III) Report : 10/14 ~ 10/15**

* 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. **Kernel panic error debugging**

* 이 부분을 위해 수정한 변수 및 소스코드는 아래의 ‘개발 중 발생한 문제나 이슈’ 문항에서 자세히 설명하겠다.

1. **System call Implementation**

* **추가한 변수 및 함수**

1. **threads/thread.h/struct thread**

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

file system call 구현을 위해, thread.h의 struct thread에 file struct pointer들의 배열을 추가하였다. 정적 배열로 선언하였으며, pintos manual에 따라 크기는 128로 setting 하였다. 또한, load\_succeed\_flag 변수를 새롭게 추가하였다. 이 변수는 해당 thread의 load가 성공했는지의 여부를 저장하고 있는 변수이다. 프로젝트 1에서 exec\_lock 변수 및 child\_succeed\_flag 변수를 통해 parent process가 exec 함수를 호출해 child thread를 생성했을 때, child thread의 정상 load여부를 알 수 있을 때까지 process\_excute함수가 return 되지 않도록 synchronization을 해주었다. 하지만 load에 실패한 child의 reaping은 처리해주지 않았었는데, load\_succeed\_flag 변수는 이러한 child thread의 reaping을 위해 추가한 변수이다. process\_wait 함수에서 child process list를 살펴보면서, 이 flag 가 false 로 set 돼있는 child process를 reaping을 해준다.

1. **Userprog/syscall.c**

프로젝트 2에서 새롭게 구현해야 할 system call 함수들을 추가하였다. 추가된 함수들의 리스트는 다음과 같다.

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

맨 위의 create 함수 ~ close 함수가 새롭게 추가된 system call 함수들이다. 각 함수들은 intr\_frame을 parameter로 받아서, 함수 내부에서 각각 필요한 user parameter 들을 parsing해서 변수에 저장한다. file system call 들은 input/output 및 함수 이름을 전달하거나, file descriptor를 전달한다. 이 때 user가 전달한 모든 parameter들에 대해 유효한지 체크해줘야 한다. 예를 들어, close함수에 전달하는 file descriptor는 반드시 이 descriptor에 mapping된 file struct가 있어야 한다. 또한, file descriptor 값은 128보다 작아야 한다. checkFileValidation 함수는 검사하고자 하는 값을 void\*로 받고, 함께 전달된 flag에 따라 void\*를 해당 타입으로 변경시켜 parameter의 유효성을 검증한다. 만약 유효하지 않은 parameter일 경우, 해당 process를 종료시킨다. Flag는 매크로로 새롭게 선언하였으며, 다음과 같다.

폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

이 flag에 따라 void\*를 char\*, int 등으로 변환해 유효성을 검증한다.

프로젝트1 에서, argument들의 user memory access 검증을 위해 argNums 배열을 추가해 각 system call에 전달되는 parameter의 개수를 저장하고, validation을 하였었다. 새롭게 추가된 system call들을 위한 argNums 값들도 초기화 해주었다.

스크린샷, 텍스트이(가) 표시된 사진

자동 생성된 설명

위의 그림에서 lock\_init(&lock\_for\_file) 부분도 새롭게 추가됐다. lock\_for\_file 변수는 lock 구조체이며, syscall.c에 전역으로 선언된 변수이다. 이 변수는 각 thread 들 간의 synchronization을 위해 사용된다. 선언 부분은 다음과 같다.

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

* **수정된 함수**

1. **threads/thread.c/init\_thread()**

* thread 구조체에 새로운 멤버들이 추가됐으니, init\_thread 함수에서 이 변수들을 초기화 해주어야 한다. 초기화 부분은 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

load\_succeed\_flag 는 true로 초기화되며, 해당 thread가 load에 실패할 경우 false로 변경된다. fd\_table은 file descriptor table이다. Thread가 생성될 때는 아무런 file 도 open하지 않은 상태이므로, 모두 NULL로 초기화 해주었다.

1. **userprog/syscall.c/read, write, exit**

* 프로젝트2 에서 이 함수들은 file read/write를 할 수 있도록 확장됐다. 따라서 parameter로 넘어온 file descriptor가 STDIN, STDOUT 일 때 뿐만 아니라, 그 외의 유효한 file descriptor가 들어올 때도 file system api 를 이용해 입력 및 출력을 할 수 있도록 하였다. 코드 부분은 다음과 같다.

**// read**

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**// write**

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

특정 process가 exit system call을 호출하면, 해당 process가 open한 file struct들을 모두 free 시켜주어야 한다. 따라서, exit system call에 이러한 작업을 해주는 부분을 추가하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

3번 file descriptor부터 살펴보면서 file struct가 NULL이 아닐 경우, 즉 mapping된 file struct가 있을 경우 해당 file\_close 함수를 호출해 해당 file struct를 free 시켜준다. 0,1,2번 file descriptor는 각각 특별한 용도의 descriptor이므로, close시키지 않는다. pintos에선 보통의 unix 계열의 운영체제처럼 0,1번은 사용하지만, 2번까지 사용한다고 돼있지는 않다. 하지만 관습적으로 2번까지 사용한다고 생각하고 구현하였다.

**(iii) userprog/process.c/process\_execute, process\_wait, start\_process**

* 위의 추가된 변수에서, load\_succeed\_flag 변수는 load에 실패한 child process의 reaping을 위해 추가했다고 하였다. 프로젝트 1에선 process\_execute 함수에서 child process의 load여부가 결정나 sema\_down에서 풀려나고 child\_succeed\_flag 변수의 값이 false라면, 단순히 -1을 return하고 종료하였다. 프로젝트 2에선, 이에 더불어 process\_wait 함수를 호출하여 load에 실패한 child process들을 reaping할 수 있도록 하였다. 이때 wait 함수의 인자로 정상적으로 -1을 주는데, 이는 정상적으로 load된 child들은 wait하지 않기 위함이다. process\_wait 함수에선 child의 tid 값과 인자로 넘어온 값을 비교해서 reaping을 결정하므로, -1을 인자로 주면 정상적인 child process들은 reaping 되지 않을 것이다. -1이 넘어와 load 실패한 thread들만을 reaping하기 위해, process\_wait 함수의 reaping 조건에 load\_succeed\_flag 변수의 값을 체크하는 부분을 추가하였다. 각 함수의 수정된 부분은 다음과 같다.

**// process\_execute**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**// process\_wait**

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Process wait의 if 문을 보면, load\_succeed\_flag 값을 확인하고 false라면 reaping 하는 것을 확인할 수 있다.

각 process의 load 성공, 실패 여부는 start process 함수에서 load를 호출한 뒤에 결정된다. load\_succeed\_flag 는 true로 초기화해주었으므로, 실패했을 경우는 flag 를 false로 바꿔준다. 해당 부분은 다음과 같다.

**// start\_process**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
  + 1. **File descriptor table initialize**

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

* + 1. **System Call & Synchronization**

도표, 기술 도면, 평면도, 라인이(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) System call Implementation**

**(i) threads/thread.h/struct thread**

thread struct에는 file struct에 대한 포인터들의 배열인 file descriptor table, load에 실패한 child process의 reaping을 위한 load flag가 추가됐다. 변경된 thread 구조체는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**(ii) threads/thread.c/init\_thread()**

struct thread 부분에 추가한 변수들의 초기화를 위해, init\_thread함수를 수정하였다. File descriptor table은 모두 NULL로 초기화하였고, load\_succeed\_flag 의 초기값은 true 이다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **userprog/syscall.c**

syscall.c 에는 file system과 관련된 여러 가지 system call이 추가됐다. system call 설명에 앞서, 새롭게 추가된 system call에 적용되는 공통적인 사항 먼저 설명하겠다. 우선, 각 함수 모두 synchronization이 필요하다. 따라서, file system api를 호출하기 전, 후로 lock을 acquire하고, 끝나면 lock을 release 해주어야 한다. 이는 모든 함수에 공통적으로 적용됐으며, 한 가지 예시 코드는 다음과 같다.

스크린샷, 텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명

create system call 함수의 일부분이다. 코드를 보면, filesys\_create api를 호출하기 전후로 lock 을 통해 synchronization을 해주었다는 것을 확인할 수 있다.

Synchronization 뿐만 아니라, file 관련 system call에 전달되는 file descriptor 값 혹은 buffer pointer를 검사해주는 과정도 공통적으로 진행된다. 이는 checkFileValidation 함수에서 처리되며, 전달되는 flag에 따라 검사 과정이 다르다. 유효한 경우 true를 return하고, 아닌 경우 false를 return 한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Flag는 FILE\_NAME, FILE\_BUFFER, FILE\_DESC의 세 가지로 나뉜다. FILE\_NAME 같은 경우, null pointer를 전달하지 않았는지 확인해주어야 한다. 또한 file buffer 같은 경우, kernel address나 null pointer를 전달 여부를 확인해주어야 한다. File descriptor 같은 경우, 항상 open 된 file의 descriptor를 전달해야 하므로 호출한 thread의 descriptor table을 확인하고, 올바른 범위 내의 file descriptor 값을 전달했는지 확인해준다.

System call을 handle 하기 전, 모든 인자들에 대해 위 함수를 통해 validation을 하고 올바른 경우에만 해당 call을 처리해준다. 다음은 추가 및 수정된 함수들에 대한 설명이다.

**(1) create**

내부적으로 filesys\_create 함수를 호출해서 file을 생성한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**(2) remove**

내부적으로 filesys\_remove 함수를 호출해 해당 file을 제거한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **open**

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

open 함수는 해당 thread의 file descriptor table 의 3번 index부터 확인하면서, 특정index의 table 값이 NULL이라면 filesys\_open 함수를 통해 file struct를 return 받고, 이를 해당 file descriptor table에 mapping 해준다. 이때 open에 실패할 경우 NULL이 return 되므로, NULL이 아닐 경우에만 mapping 하도록 처리하였다.

open 같은 경우, 특히 신경 써 주어야 할 부분이 있다. 바로 실행중인 process의 executable file에 대해 write를 못하게 막아주는 것이다. 만약 write를 하게 되면, 예측하지 못하는 결과가 발생할 수도 있기 때문이다. 따라서 어떤 process가 현재 실행중인 thread의 실행 파일을 open 하고자 하면, 해당 file 에 write를 하지 못하도록 file\_deny\_write함수를 호출해주어야 한다. 이는 위의 코드의 if 문에 잘 나타나있다. thread\_name 함수를 통해 현재 실행중인 thread의 이름을 받아오고, 이를 open 하려는 file의 이름과 비교한다. 만약 같으면 file\_deny\_write함수를 호출해 write 를 방지한다.

정상적으로 file을 open 했을 경우, 해당 index 값을 return 해준다. 만약 아닐 경우, -1값을 return 해준다.

1. **filesize**

내부적으로 file\_length 함수를 호출해 해당 file의 크기를 알아온다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **seek**

내부적으로 file\_seek 함수를 호출해 해당 file의 offset을 변경해준다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **tell**

내부적으로 file\_tell 함수를 호출해 해당 file의 현재 offset을 반환해준다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **close**

내부적으로 file\_close함수를 호출하여 해당 file struct를 free 시켜준다. 이때, 호출한 thread의 해당 file descriptor table 원소를 NULL로 변경해주어야 한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **read**

프로젝트1 에서 발전하여, file을 통한 읽기가 가능해졌다. STDIN인 경우 여전히 input\_getc 함수를 호출해서 입력을 받고, file input인 경우 file\_read 함수를 호출해 입력받는다. 코드는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**(9) write**

프로젝트1 에서 발전하여, file에 쓰기가 가능해졌다. STDOUT인 경우 여전히 putbuf 함수를 이용해 출력하고, file output인 경우 file\_write 함수를 호출해 출력한다. 코드는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

exit 함수는 위에서 설명한대로 종료되는 thread의 open된 file struct들에 대해, 모두 close를 호출해 file struct를 free 시켜줘야 한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**(iv) userprog/process.c/process\_execute, process\_wait, start\_process**

위의 세 함수들은, load 에 실패한 child process의 memory leak을 방지하기 위해 수정됐다. 프로젝트 1 같은 경우, 해당 child process가 load에 실패했다는 것만 알고 reaping은 해주지 않았지만, 이러면 계속해서 system resource를 차지할 것이다. 이를 막기 위해 각 함수를 다음과 같이 수정하였다.

**// process\_execute**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

기존에는 child\_succeed\_flag == false인 경우 TID\_ERROR를 return 하고 종료됐지만, 현재는 reaping을 위해 process\_wait(-1) 를 호출하는 것을 확인할 수 있다. process\_wait에 -1이 전달된 경우, 일반적인 thread들은 reaping 하지 않는다. 오직 load\_succeed\_flag 가 false인 child thread들만 reaping 할 것이다. 이것이 가능하도록 process\_wait 함수를 수정하였고, 코드는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

함수를 확인해보면 if문에서 child->load\_succeed\_flag == false 인 경우 해당 child process에 대해 reaping 한다는 것을 알 수 있다. 전달된 tid 는 -1일 것이므로, tid가 양수인 일반 thread들은 reaping되지 않을 것이라는 것을 알 수 있다.

load의 실패를 확인하고, load\_succeed\_flag의 값을 변경해주는 부분은 start\_process이다. 이 함수에서 load 함수를 호출하고, return 값을 통해 성공 여부를 알 수 있기 때문이다. 코드는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

load의 초기값은 true이기 때문에, 오직 실패한 경우에만 flag 값을 변경해준다.

**(2) Kernel panic error debugging**

프로젝트 2를 진행하기 위해서, thread.h 의 thread struct에 file descriptor table 을 추가하고 테스트를 돌려보니, 프로젝트 1에선 정상적으로 pass 하였던 많은 테스트 케이스들에서 fail 이 떴다. Thread 구조체에 table을 추가한 것 말고는 아무것도 변경하지 않았는데 kernel panic이 뜨며 실패하니, 뭐가 문제인지 도통 알 수가 없었다. 며칠을 디버깅하고 고민해본 결과, kernel stack에서 overflow가 난 것이 문제였다. User program의 argument passing을 할 때, 기존의 코드는 아래와 같다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

이 코드를 보면, uint32\_t address\_array[MAXARGV] 를 local 변수로 선언하고, parsing 되는 argument 들의 주솟값을 저장하는 방식으로 함수를 구현하였다. 이것이 문제였다. MAXARGV는 256으로 define된 macro이다. 따라서 저 배열의 크기는 256 \* 4 byte = 1KB 이다. Thread에 할당된 한 개의 page에서, kernel stack은 최대 3KB까지 커질 수 있는데, 위와 같이 매우 큰 지역 변수를 선언해서 kernel stack에 overflow 가 발생한 것이었다. 프로젝트 1에선 file descriptor table와 같은 추가적인 변수를 선언하지 않아서 문제가 없었지만, 프로젝트 2에서 table과 더불어 다른 변수들도 추가적으로 선언하니 overflow가 난 것이다. 이를 해결하기 위해, argument passing 함수의 address\_array 배열을 정적으로 선언하지 않고, malloc을 통해 heap 영역에 할당하도록 변경하였다. 변경한 코드는 다음과 같다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Argument passing 함수뿐만 아니라, process\_execute 함수에서도 크기가 큰 정적 배열을 선언하였었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위의 코드는 프로젝트 1에서 작성한 코드로, commandline 변수도 굉장히 큰 정적 배열이다. Argument\_passing과, process\_execute 함수에 선언한 이러한 변수들 때문에 kernel stack overflow가 발생했던 것이다. 이 부분도 프로젝트 2에선 다음과 같이 변경하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

execution\_name 에 strlcpy로 데이터를 복사하지 않고, file\_name 변수를 보면서 바로 한 문자씩 저장하면서 시간과 메모리 사용을 개선하였다.

위와 같이 동적으로 데이터를 할당 받고, 크기가 큰 정적 배열을 제거하니 ASSERT(is\_thread(t)) 에 걸려서 kernel panic이 발생하던 모든 테스트 케이스들을 정상적으로 통과할 수 있었다.

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

