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

담당 교수 / 분반 : 문의현 교수님 / 02반

이름 / 학번 : 심은서 / 20191112

개발 기간 : 2022.11.09~2022.11.14

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

지난 project 1에서 user program을 실행하기 위한 기본 뼈대를 만들었다면 이번 프로젝트에서는 file system에 대한 system call들, 즉 create, remove, open, close, filesize, write, seek, tell을 마저 구현함으로써 user가 file을 열고, 읽고, 쓸 수 있도록 한다. 이 프로젝트를 통해 pintos의 base file system을 이해하고 file을 처리 할 때 생길 수 있는 race condition에 대한 문제를 처리한다.

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 descriptor란 시스템에서 실행되는 file들을 관리하기 위하여 할당되는 것으로, file descriptor table로 관리되며 각 프로세스는 각자의 file descriptor table을 가지고 있다. File descriptor table에서 index 0은 STDIN, 1은 STDOUT, 2는 STDERR로 할당이 되어있으므로 유저가 실행하는 파일은 index 3번부터 추가된다. File descriptor를 통해 파일에 접근할 수 있는데 현재 pintOS에서는 file descriptor가 구현되어있지 않으므로 파일 입출력을 위하여 반드시 구현해야한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

File system을 구현하기 위해서는 다음과 같은 system call들이 필요하다. close(파일을 닫음), read(파일의 데이터를 읽음), write(파일에 데이터를 기록함), seek(파일에서 원하는 위치로 이동함), tell(파일의 위치를 알려줌), remove(파일을 삭제함), create(파일을 생성함), filesize(파일의 크기를 알려줌) 따라서 이 system call들을 project 1에서 했던 것과 같이 syscall.c에 추가하여 적절히 실행되도록 한다.

3. Synchronization in Filesystem

동시에 같은 파일에 접근하는 등 race condition이 발생할 수 있다. 이를 방지하기 위하여 lock을 잠금으로써 현재 사용 중인 파일에 다른 프로세스가 접근하지 못하도록 하고, 또 사용이 끝나면 적절한 때에 lock을 풀어주어야 한다. 현재 실행 중인 파일에 다른 프로세스가 쓰지 못하도록 file\_deny\_write()을 활용하여 특정 파일이 현재 쓸 수 있는 상태인지 아닌지에 대한 flag를 활용할 수 있다. 또한 read, write 과정에서 synchronization 문제가 발생하지 않도록 semaphore를 추가하여 parent가 child의 load를 기다리도록 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

File descriptor들을 저장하는 배열인 struct file\* fdt[128]을 선언하여 사용하였다. 처음엔 동적 할당을 하려고 했으나 메모리 누수를 철저하게 관리해야하기 때문에 위험부담이 덜한 리스트 자료구조를 선택하게 되었다. 또한 pintos manual에서 최대로 open하는 file의 수를 128개로 하기를 추천하고 있으므로 크기가 128인 리스트를 만들었다.

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)

- bool create (const char \*file, unsigned initial\_size) : initial\_size를 크기로 하는 새로운 파일을 생성하며 생성에 성공했다면 true, 실패했다면 false를 반환한다.

- int open (const char \*file) : file을 열고 열린 file의 file descriptor를 반환한다. 열리지 않았다면 -1을 반환한다.

- Int filesize (int fd) : fd에 해당하는 file의 size를 byte단위로 반환한다.

- int write (int fd, const void \*buffer, unsigned size) : fd가 1일경우 file에 putbuf()로 write하고 write된 byte 수를 리턴한다. 아무것도 쓰지 못했을 경우 0을 리턴하고 파일을 열지 못하는 등 정상적인 처리가 되지 않았을 경우 -1을 리턴한다.

- int read (int fd, void \*buffer, unsigned size) : fd가 0일 경우 file을 input\_getc()로 읽어들이고 읽은 byte 수를 리턴한다. 읽지 못했을 경우 -1을 리턴한다.

- void seek (int fd, unsigned position) : 읽거나 쓸 byte의 위치를 position으로 이동시킨다.

- unsigned tell (int fd) : 읽거나 쓰여질 다음 byte의 위치를 반환한다.

- void close (int fd) : file descriptor fd를 닫는다. 프로세스가 끝나면 해당 프로세스에 열려있는 모든 file을 닫아야한다.

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

먼저 lock은 read, write시의 간섭을 방지하기 위하여 이용할 수 있다. 한 process가 어떤 file을 read하는 동안 다른 process가 같은 file을 write해서는 안된다. 이런 상황을 막기 위하여 lock을 선언하고, syscall.c의 read()와 write() 각각에서 file에 접근하기 직전에는 lock\_acquire을 통해 다른 프로세스의 접근을 막은 뒤, 파일에 대한 read또는 write 작업을 모두 마친 후에는 lock\_release를 통해 lock을 풀어 이제 다른 프로세스가 이 파일을 읽거나 쓸 수 있도록한다. Semaphore의 경우 마찬가지로 여러 process가 동일한 file을 가지고 작업을 하지 않도록 동기화 문제를 해결하기 위해 사용한다. Parent thread가 child thread가 끝나기 전에 먼저 끝나버리면 child thread가 정상적으로 load되지 않았는데도 이를 처리할 수 없게 되므로, 이런 상황을 방지하기 위하여 parent thread가 child thread의 load가 끝날 때 까지 기다리도록 해야한다. 이를 위해 적절한 위치에 sema\_down과 sema\_up을 활용하여 parent와 child간 동기화가 해결되도록 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

11.09 ~ 11.10 : 프로젝트 내용과 코드 이해

11.11 : file descriptor, system call 구현

11.12 ~ 11.13 : synchronization, lock 관련 문제 해결

11.14 : 마무리 및 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

생성되는 각 thread별로 File descriptor들을 저장하는 리스트가 있어야 하므로 thread.h의 struct 구조체에 struct file \*fdt[128]를 추가하며 모두 null로 초기화한다.

1. System Calls

Syscall.c에서 syscall\_handler를 받아와서 해당하는 system call을 처리하는 부분에서 아직 구현되지 않은 system call인 SYS\_CREATE, SYS\_REMOVE, SYS\_OPEN, SYS\_FILESIZE, SYS\_SEEK, SYS\_TELL, SYS\_CLOSE에 대한 case를 추가한 뒤 각 system call을 처리할 함수를 만들고 적절한 인자를 넘겨준다. 대부분의 system call들이 내장 함수를 통해 해결될 수 있으나 open()에서는 현재 thread의 file descriptor table을 탐색하면서 비어있는 곳에 현재 file을 할당해주는 작업을 수행해야한다.

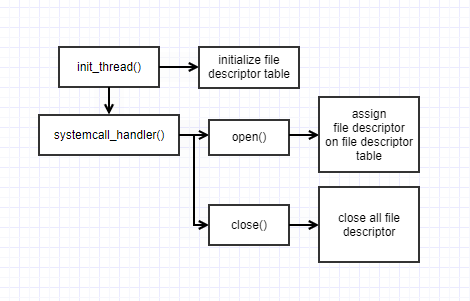
1. Synchronization in Filesystem

Syscall.h에 struct lock filesys\_lock을 선언하고 syscall.c의 syscall\_init에서 lock\_init으로 lock을 초기화해준다. syscall.c의 open(), write(), read() 함수에서는 critical section을 보호하기 위하여 파일에 접근하기 전에는 lock\_acquire을 통해 다른 프로세스에서 같은 파일에 접근하지 못하도록 하고, 파일 사용을 마친 후에는 lock\_release를 통해 다시 다른 프로세스에서 이 파일에 접근할 수 있도록 lock을 풀어주어야 한다. 또한 open()에서 현재 열고자 하는 file과 현재 thread의 이름이 같다면 동시에 접근하는 것을 막아야 하므로 file\_deny\_write을 통해 다른 프로세스가 이 파일에 write하지 말 것을 알린다. 이를 write()에서 확인하여 file의 deny\_write이 1이라면 쓰기를 수행하지 않도록 해야한다.

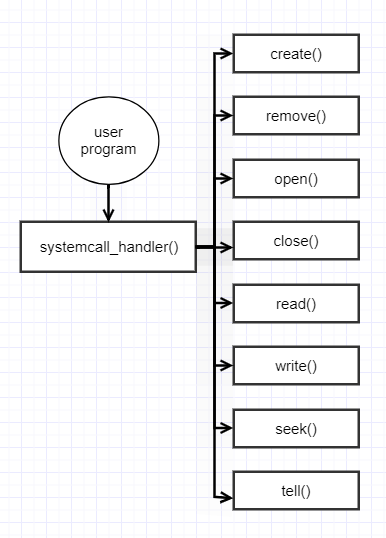
Thread.h의 thread 구조체에 새로운 semaphore child\_load\_lock을 선언하고 thread.c의 init\_thread()에서 sema\_init으로 초기화해준다. Process.c에서는 parent가 child의 load를 기다리도록 하기 위하여, child process를 생성하는 thread\_create()직후에 sema\_down을 실행하여 parent가 child를 기다리도록 한다. 그리고 start\_process()에서 child가 load를 마친 시점, 즉 load 함수가 불러진 후에 이 process의 parent에 대하여 sema\_up을 실행하여 synchronization 문제를 해결한다.

1. **연구 결과**
   1. **Flow Chart**

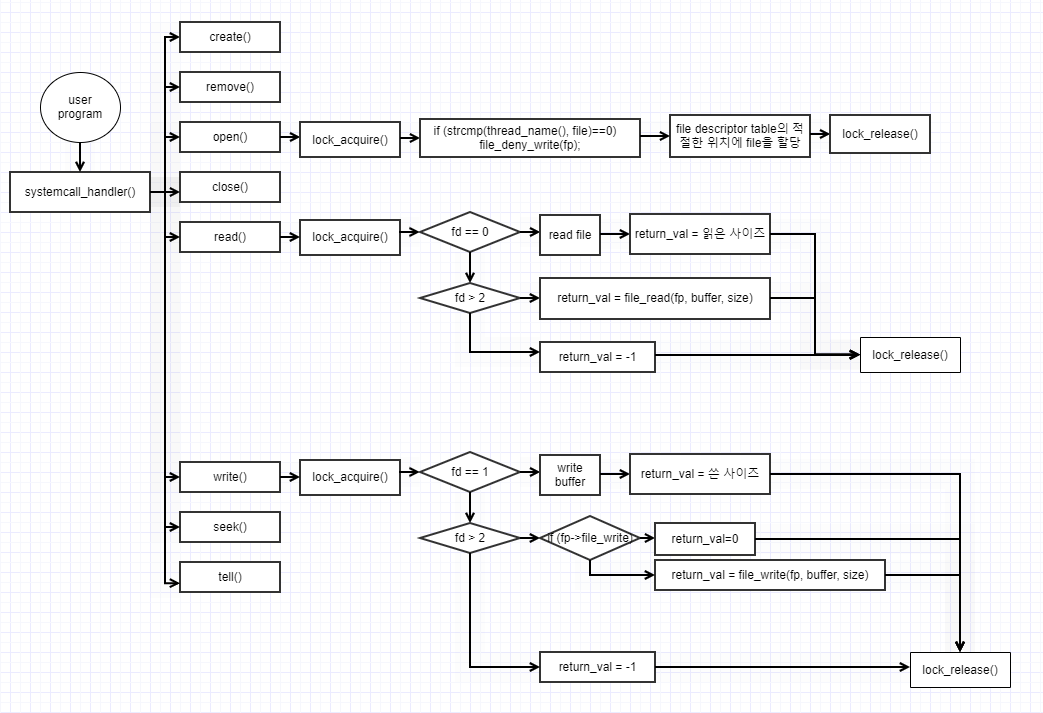
* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
  + 1. File descriptor

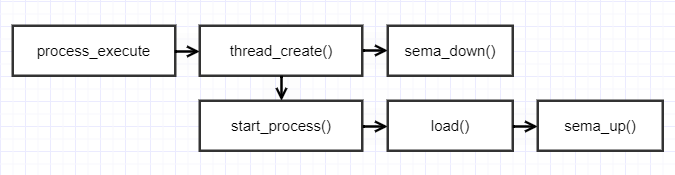


* + 1. System calls



* + 1. Synchronization in Filesystem





* 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 \*/

#define FD\_LIMIT 128 // maximum number of file descriptor

struct thread {

…

struct file \*fdt[FD\_LIMIT];//file descriptor table

…

}

/\* thread.c \*/

for(int i = 0; i < FD\_LIMIT; i++)

    t->fdt[i] = NULL;

thread.c의 thread 구조체 안에 file descriptor들을 저장할 file descriptor table 리스트를 선언한다. Pintos manual에 따라서 file descriptor의 최대 수를 128로 제한하였다. Thread.c의 init\_thread()에서는 선언된 fdt의 모든 file descriptor를 NULL로 초기화해준다.

**2. System Calls**

case SYS\_CREATE:

    {

      if (!is\_user\_vaddr(f->esp+4)) exit(-1);

      if (!is\_user\_vaddr(f->esp+8)) exit(-1);

      f->eax = create((const char \*)\*(uint32\_t \*)(f->esp + 4), (unsigned)\*(uint32\_t \*)(f->esp + 8));

      break;

    }

    case SYS\_REMOVE:

    {

      if (!is\_user\_vaddr(f->esp+4)) exit(-1);

      f->eax = remove((const char\*)\*(uint32\_t \*)(f->esp + 4));

      break;

    }

    case SYS\_OPEN:

    {

      if (!is\_user\_vaddr(f->esp+4)) exit(-1);

      f->eax = open((const char\*)\*(uint32\_t\*)(f->esp+4));

      break;

    }

    case SYS\_FILESIZE:

    {

      if (!is\_user\_vaddr(f->esp+4)) exit(-1);

      f->eax = filesize((int)\*(uint32\_t\*)(f->esp+4));

      break;

    }

    case SYS\_SEEK:

    {

      if (!is\_user\_vaddr(f->esp+4)) exit(-1);

      if (!is\_user\_vaddr(f->esp+8)) exit(-1);

      seek((int)\*(uint32\_t\*)(f->esp+4), (unsigned)\*(uint32\_t \*)(f->esp+8));

      break;

    }

    case SYS\_TELL:

    {

      if (!is\_user\_vaddr(f->esp+4)) exit(-1);

      f->eax = tell((int)\*(uint32\_t \*)(f->esp + 4));

      break;

    }

    case SYS\_CLOSE:

    {

      if (!is\_user\_vaddr(f->esp+4)) exit(-1);

      close((int)\*(uint32\_t \*)(f->esp + 4));

      break;

    }

Syscall.c의 syscall\_handler()에서 file system에서 필요한 system call들에 대하여 case를 만들어 처리되도록 하였다. 각 system call들에서 잘못된 주소에 접근하는 것을 막기 위하여 is\_user\_vaddr로 주소를 체크해주고, 올바른 주소라면 각 system call에 맞는 함수를 실행한다.

bool

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

  if (file == NULL) exit(-1);

  return filesys\_create(file, initial\_size);

}

bool

remove(const char \*file) {

  if (file == NULL) exit(-1);

  return filesys\_remove(file);

}

Create()과 remove()에서는 file NULL인지 아닌지 체크한 후 filesys\_create와 filesys\_remove라는 내장함수를 통해 파일 생성과 삭제를 각각 수행한다.

int

open(const char \*file) {

  if (!is\_user\_vaddr(file)) exit(-1);

  if (file == NULL) return -1;

  lock\_acquire(&filesys\_lock);

  int return\_val = -1;

  struct thread \*cur\_thread = thread\_current();

  struct file\* fp = filesys\_open(file);

  if (fp == NULL) {

    lock\_release(&filesys\_lock);

    return -1;

  }

  if (strcmp(thread\_name(), file)==0){

    file\_deny\_write(fp);

  }

  for(int fd = 3; fd<FD\_LIMIT; fd ++){

    if(cur\_thread->fdt[fd]==NULL){

      cur\_thread->fdt[fd]=fp;

      return\_val = fd;

      cur\_thread->fd\_index = fd+1;

      break;

    }

  }

  lock\_release(&filesys\_lock);

  return return\_val;

}

Open()은 file descriptor를 관리하고 동기화 문제를 해결함에 있어서 굉장히 중요한 부분이다. 기본적으로 인자로 받은 file을 filesys\_open을 통해 open한 뒤 정상적으로 열리지 않았다면, 즉 NULL이 return되었다면 -1을 return하고 끝낸다. 그렇지 않다면 file descriptor table를 탐색하며 새로 열린 file이 할당 될 빈 자리를 찾고 찾았다면 해당 위치에 열린 file descriptor를 저장한다. 또한 이 file descriptor를 return한다.

Open()에서 해야할 임무 중 중요한 것은 현재 실행중인 파일이 다른 프로세스에 의해 수정되는 것을 막는 것이다. 이를 위해 현재 thread\_name()과 새로 open하는 file의 이름을 비교하고 이 둘이 같다면, 이미 실행 중인 file을 open하려는 것이므로, file\_deny\_write()을 통해 해당 file에 대한 deny\_write값을 1로 만들어줌으로써 이 file은 현재 write할 수 없음을 알린다.

int

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

{

  if (!is\_user\_vaddr(buffer)) exit(-1);

  struct thread \*cur\_thread = thread\_current();

  struct file\* fp = cur\_thread->fdt[fd];

  int return\_val = -1;

  lock\_acquire(&filesys\_lock);

  if (fd == 0){

    for (int i=0; i<size; i++){

      if(input\_getc() == '\0'){

        return\_val = i;

        break;

      }

    }

  } else if (fd>2) {

    if (fp == NULL) {

      return\_val = -1;

    }

    else {

      return\_val = file\_read(fp, buffer, size);

    }

  }

  else {

    return\_val = -1;

  }

  lock\_release(&filesys\_lock);

  return return\_val;

}

int

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

  if (!is\_user\_vaddr(buffer)) exit(-1);

  lock\_acquire(&filesys\_lock);

  int return\_val = -1;

  if (fd == 1) {

    putbuf(buffer, size);

    return\_val = size;

  }

  else if (fd > 2){

    if (thread\_current()->fdt[fd] == NULL){

      return\_val = -1;

    }

    else {

      struct thread \*cur\_thread = thread\_current();

      struct file\* fp = cur\_thread->fdt[fd];

      if (fp->deny\_write){

        return\_val = 0;

      } else{

        return\_val = file\_write(fp, buffer, size);

      }

    }

  }

  else {

    return\_val = -1;

  }

  lock\_release(&filesys\_lock);

  return return\_val;

}

Read()와 write()함수는 project 1에서 구현했던 함수지만 file system을 위해서 추가적인 구현이 필요하다. Project 1에서는 STDIN, STDOUT에 대해서만 구현해놓았기 때문에, STDIN과 STDOUT에 해당하는 file descriptor인 0,1과 STDERR에 해당하는 file descriptor인 2를 제외한 나머지 file descriptor에 대해 처리를 해주기 위하여 else if (fd>2)라는 조건문을 추가하였다.

Read()에서는 fd가 2보다 클 경우, 해당 file이 NULL이라면 return값을 -1로 하는 예외처리를 해준 뒤 file\_read()를 실행하여 파일을 읽고 읽은 byte 수를 return한다. Write()에서 또한 fd가 2보다 클 경우, file\_write()을 실행하여 파일에 쓰고 쓴 byte 수를 return하는데, read()와 다른 점은 file의 deny\_write flag를 확인한다는 것이다. File을 open()할 때 현재 실행중이므로 다른 프로세스가 쓸 수 없는 file에 대해서는 deny\_write flag를 세워놓았었다. 따라서 write()에서 이를 체크하여 deny\_write상태인 file이라면 write하지 않고, 아무것도 쓰지 않았다는 의미로 0을 return한다.

Open(), read(), write() 모두에서 중요한 것은 바로 lock이다. 한 process가 특정 file에 접근하는 동안 다른 process는 접근하지 못하도록 해야한다. 라서 함수의 초반에 lock\_acquire()로 lock을 걸어주고 모든 작업을 마친 후에 lock\_release()로 lock을 풀어준다. 이때 lock\_acquire()과 lock\_release()의 적절한 위치를 찾는데 많은 어려움을 겪었다. Sys\_read와 sys\_write test에서 FAIL이 난 것은 lock이 제대로 풀리지 않았기 때문이었다. 여러 조건문이 있다보니 lock\_release()가 부적절한 위치에 들어가면 어떤 조건에서는 lock이 풀리지 않는 문제가 발생하였다. Lock이 풀리지 않은채로 return해버리는 상황을 막기 위해 모든 조건문에서 return하는 것이아니라 return\_val이라는 변수를 따로 만들어 이 변수에 return할 값을 저장한 뒤, 모든 조건문에 대한 체크가 끝난 후 마지막에 lock\_release()를 하고 return\_val를 return하는 방식으로 구현함으로써 해결할 수 있었다.

int

filesize (int fd) {

  struct thread \*cur\_thread = thread\_current();

  struct file\* fp = cur\_thread->fdt[fd];

  if (fp == NULL) return -1;

  return file\_length(fp);

}

void

seek (int fd, unsigned position) {

  struct thread \*cur\_thread = thread\_current();

  struct file\* fp = cur\_thread->fdt[fd];

  if (fp == NULL) exit(-1);

  file\_seek(fp, position);

}

unsigned

tell (int fd) {

  struct thread \*cur\_thread = thread\_current();

  struct file\* fp = cur\_thread->fdt[fd];

  if (fp == NULL) return -1;

  return file\_tell(fd);

}

void

close (int fd) {

  struct thread \*cur\_thread = thread\_current();

  struct file\* fp = cur\_thread->fdt[fd];

  if (fp == NULL){

    exit(-1);

  }

  fp = NULL;

  file\_close(fp);

}

Filesize(), seek(), tell(), close()에 관해서는 모두 내장 함수를 이용하였다. 네 함수 모두 fd를 인자로 받으므로 해당 fd에 대한 file이 NULL인지 체크하고 NULL이 아니라면 해당하는 내장함수를 실행하였다.

**3. Synchronization in Filesystem**

Read()와 write()에 관한 동기화 문제에 관해서는 위의 2. System call에서 설명하였다. 해당 함수들에 사용된 lock은 다음과 같이 syscall.h에서 선언하고 syscall.c의 syscall\_init()에서 초기화해주었다.

struct lock filesys\_lock;

void

syscall\_init (void)

{

  intr\_register\_int (0x30, 3, INTR\_ON, syscall\_handler, "syscall");

  lock\_init(&filesys\_lock);

}

process간의 synchronization 문제를 해결하기 위하여 다음과 같이 thread.h의 thread 구조체에 새로운 semaphore를 선언하였다. 또한 동기화 과정에서 process의 parent에 접근해야하는 상황이 생기기 때문에 parent 변수도 새로 선언하였다.

struct semaphore child\_load\_lock;

struct thread\* parent;

project1에서 해결되지 않은 동기화문제는 바로 parent의 실행과 child의 load에 관한 것이다. Process를 실행하면 parent에서 thread\_create()을 통해 child를 생성하게 되고, 이 child가 load()를 통해 thread를 만들어 수행하게 된다. 이 때 child의 상태를 제대로 알기 위하여 parent가 child보다 먼저 죽지 않도록 해야하는데, 즉 child의 load가 모두 끝날 때까지 parent가 죽어서는 안된다. 이를 위하여 child가 만들어지는 시점인 thread\_create()의 바로 다음에 sema\_down을 통해 parent가 죽지 않도록 한다.

  /\* process.c의 process\_execute \*/

…

tid = thread\_create(tmp, PRI\_DEFAULT, start\_process, fn\_copy);

sema\_down(&thread\_current()->child\_load\_lock);

…

그리고 child의 load가 끝나고나면 그 child의 부모에 대해 sema\_up을 해주어야 한다. 따라서 load가 끝나는 시점에 sema\_up을 실행한다. 이 때 현재 process에 대해 sema\_up을 하는 것이 아니라 아까 sema\_down을 해주었던 부모에 대해 sema\_up을 해주어야 하므로 thread\_current()->parent->child\_load\_lock에 sema\_up을 실행한다.

  /\* process.c의 start\_process \*/

…

success = load (file\_name, &if\_.eip, &if\_.esp);

sema\_up(&thread\_current()->parent->child\_load\_lock);

…

이때 parent에 접근하기 위해서는 parent의 값이 할당이 되어야 하므로, thread.c의 init\_thread()에서 다음과 같이 parent를 초기화헤주었다.

t->parent = running\_thread();

추가로, syscall.c의 exit()에서 모든 file이 닫힌 상태로 안정적으로 process가 종료되도록 하기 위하여 file descriptor table을 돌며 NULL이 아닌 fd가 있다면 close해주는 코드를 추가하였다.

  /\* syscall.c의 exit() \*/

…

for (int i=3; i<FD\_LIMIT; i++){

    if(thread\_current()->fdt[i]!=NULL){

      close(i);

    }

}

…

프로젝트 진행 중 해결에 가장 많은 시간이 걸린 것은 rox-child, rox-multichild test에 관한 것이었다. 해당 test들은 현재 실행중인 파일이 child에 의해 접근되는 것을 막아야 해결이 되는데, file\_deny\_write 처리를 해도 해당 test들이 해결되지 않았다. 여러 디버깅 과정을 통해 해당 test들에서 file의 deny\_write이 여전히 0으로 되어있는 것을 확인했고, open()에서 printf()를 통해 디버깅을 해본 결과 다음과 같은 결과로 file\_deny\_write처리가 되지 않는 것을 확인했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이를 통해 file의 name을 처리하는데 무엇인가 문제가 있음을 추측해보았고, process.c의 process\_execute에서 file name을 parsing하는 과정에서 thread\_create()에 넘기는 thread의 이름 값이 잘못 넘어가고 있다는 것을 발견했고 이를 수정함으로써 해결할 수 있었다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명