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

담당 교수 / 분반 : 김영재 교수님 / 2반

이름 / 학번 : 정현정 / 20210428

개발 기간 : 10/27 ~ 10/28

1. **개발 목표**

이번 프로젝트에서는 파일 시스템과 관련된 시스템 콜을 구현한다. 기본적인 파일 시스템 동작을 위한 시스템 콜을 구현하고 데이터를 읽거나 쓸 때 동기화 문제를 해결하는 것이 이번 프로젝트의 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

Pintos에서는 기본적으로 File Descriptor 및 이와 관련된 기능을 제공하지 않는다. 즉, 특정 파일에 대해 이를 식별하는 값을 저장하지 않고 있다. Pintos manual에 따르면, 한 프로세스는 최대 128개의 open file을 가질 수 있다. 이를 저장할 File Descriptor table이 필요하다. STDIN, STDOUT이 0번과 1번 File Descriptor를 이미 사용하고 있으므로 2번부터 사용한다.

2. System Calls

이번 프로젝트에서는 파일 시스템과 관련된 시스템 콜인 create, remove, open, close, filesize, seek, tell, read, write를 구현한다. 이전 프로젝트에서 구현하였던 read, write는 STDIN과 STDOUT만 다루고 있기 때문에, 다른 파일들을 대상으로 한 입출력도 가능하도록 수정해야 한다.

3. Synchronization in Filesystem

파일 시스템 관련 시스템 콜을 구현할 때 동기화 문제를 해결해주어야 한다. 여러 프로세스가 같은 파일에 접근하는 경우 동기화 문제가 발생한다. 어떤 프로세스가 특정 파일을 읽는 도중 데이터 업데이트가 발생하는 경우, 실행 중인 사용자 프로그램에 데이터를 쓰는 경우 등이 그 예이다. 이를 해결하기 위해 lock 혹은 semaphore를 적절하게 사용한다.

* 1. **개발 내용**

1. File Descriptor

thread 구조체에 struct file \*fd\_table[130]; 필드를 추가하고 NULL값으로 init\_thread() 함수에서 초기화를 수행한다. Linked List에 비해 메모리 사용량 측면에서 더 좋은 성능을 보이지만, 삽입과 삭제 연산에 인덱스를 사용할 수 없어 임의적인 접근이 불가능하다는 단점이 있다. 따라서 직관적으로 file descriptor table에 접근하기 위해 인덱싱이 가능한 정적 배열을 사용했다. 총 130개의 파일 디스크립터를 저장할 수 있도록 배열을 선언하였고, 이는 STDIN, STDOUT과 최대 128개의 open file을 가질 수 있다는 pintos manual을 반영한 것이다.

2. System Calls

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

이름이 file인 새 파일을 생성한다. 파일 생성에 성공하면 true, 실패하면 false를 반환한다. 생성 시점에 initial\_size 바이트만큼의 초기 크기를 갖는다.

- bool remove(const char \*file)

이름이 file인 파일을 삭제한다. 파일 삭제에 성공하면 true, 실패하면 false를 반환한다.

- int open(const char \*file)

file이라는 이름을 가진 파일을 연다. 성공 시 파일에 대한 file descriptor를 반환하고, 실패 시 -1을 반환한다. 파일을 여러 번 열면 같은 파일이라고 하더라도 별도의 file descriptor를 할당한다.

- void close(int fd)

인자로 전달받은 fd에 해당하는 파일을 닫는다. 프로세스가 종료되면 모든 열린 fd는 자동으로 닫혀야 한다.

- int filesize(int fd)

fd가 가리키는 열려있는 파일의 크기를 바이트 단위로 반환한다.

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

열려있는 fd에서 최대 size 바이트를 buffer로 읽어오고, 실제로 읽어온 바이트 수를 반환한다. 파일 끝에 도달하면 0을 반환하고, 읽기에 실패하면 -1을 반환한다. fd가 0인 경우 키보드 입력에서 읽어온다.

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

buffer에 있는 size 바이트의 데이터를 열려있는 fd에 기록하고, 실제로 기록된 바이트 수를 반환한다. 만약 파일 끝을 넘어 쓰기가 불가능 하면 가능한 만큼만 쓴다. 아무것도 기록하지 못하면 0을 반환하며, fd가 1인 경우 콘솔에 기록한다.

- void seek(int fd, unsigned position)

fd로 열린 파일의 다음 읽기나 쓰기 위치를 position 바이트 위치로 변경한다. 파일의 끝을 넘어가도 오류가 발생하지 않는다.

- unsigned tell(int fd)

fd에서 다음에 읽거나 쓸 위치를 바이트 단위로 반환한다.

3. Synchronization in Filesystem

여러 프로세스에서 동시에 같은 파일에 접근하는 경우 동기화 문제가 발생한다. 따라서 파일에 접근하는 시스템 콜에 대해 작업 수행 전후로 lock을 이용하여 동기화 문제를 해결한다. 이때 사용되는 lock은 syscall.c에 전역 변수로 선언한 것을 사용한다.

또한 실행 중인 사용자 프로그램에 데이터를 쓰는 것을 방지해야 한다. 즉, 프로그램이 종료된 상태일 때만 변경이 가능하도록 해야 한다. 이를 위해 load 함수에서 실행할 파일을 열 때, 실행 중인 파일에 대해 file\_deny\_write()를 호출해야 한다. 프로그램이 종료된 후 다시 쓰기가 가능해야 하므로 process\_exit() 함수를 수정하여 현재 실행 중인 파일에 대해 file\_allow\_write()를 호출한다. 이를 구현하기 위해 thread 구조체에 현재 실행 중인 파일을 저장하는 새로운 필드가 필요하다. 더불어 자식 프로세스가 load 되는 도중 부모 프로세스가 종료되는 것을 막기 위해 semaphore를 사용해야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

- 10/27 : File Descriptor, System Call 구현

- 10/28 : Synchronization in Filesystem 구현 및 보고서 작성

* 1. **개발 방법**

1. File Descriptor

파일 디스크립터 테이블을 구현하기 위해 thread.h에 있는 thread 구조체에 struct file \*fd\_table[130]; 배열을 선언한다. 그리고 init\_thread() 함수 안에서 반복문을 돌며 NULL로 초기화를 한다. 이후 파일 디스크립터를 사용하는 시스템 콜에서 해당 배열에 접근하여 파일 디스크립터를 사용하도록 한다.

2. System Calls

write와 read의 경우 이전 프로젝트에서 사용하였던 시스템 콜을 수정한다. 나머지 시스템 콜들은 syscall.c에 새롭게 함수를 작성한다. 필요한 경우 thread.h에 선언한 파일 디스크립터 테이블을 사용한다.

3. Synchronization in Filesystem

각 프로세스가 파일에 접근할 때 발생할 수 있는 동기화 문제를 해결하기 위해서 syscall.c에 전역 변수로 lock을 선언하고 이를 syscall\_init() 함수에서 초기화한다. 이 변수를 통해 동기화가 필요한 시스템 콜에서 작업 수행 전후로 lock을 걸고 해제해준다.

또한 실행 중인 사용자 프로그램에 데이터를 쓰는 경우를 방지하기 위해 process.c의 load 함수를 수정하여 file\_deny\_wrtie() 함수를 사용하는 과정을 추가한다. file\_allow\_write()의 경우, file\_close()에서 수행되고 있으므로 process.c의 process\_exit() 함수에 현재 실행 중인 파일을 닫도록 한다. 또 자식 프로세스가 load 되는 도중 부모 프로세스가 종료되는 것을 막기 위해 기존의 exec 시스템 콜에 semaphore를 추가한다.

1. **연구 결과**
   1. **Flow Chart**

**도표, 텍스트, 평면도, 디자인이(가) 표시된 사진

자동 생성된 설명**

* 1. **제작 내용**

**1. File Descriptor**

src/threads/thread.h

struct thread

  {

    /\* Owned by thread.c. \*/

    tid\_t tid;                          /\* Thread identifier. \*/

    enum thread\_status status;          /\* Thread state. \*/

    char name[16];                      /\* Name (for debugging purposes). \*/

    uint8\_t \*stack;                     /\* Saved stack pointer. \*/

    int priority;                       /\* Priority. \*/

    struct list\_elem allelem;           /\* List element for all threads list. \*/

    /\* Shared between thread.c and synch.c. \*/

    struct list\_elem elem;              /\* List element. \*/

#ifdef USERPROG

    /\* Owned by userprog/process.c. \*/

    uint32\_t \*pagedir;                  /\* Page directory. \*/

    struct semaphore wait\_exit; //자식 프로세스 종료 대기

    struct semaphore wait\_free; //메모리 해제 대기

    struct semaphore wait\_load; //자식 프로세스 load 대기

    bool exec\_flag; //자식 프로세스가 정상적으로 exec 되었는지 표시하는 flag : true = 정상 실행

    int exit\_status; //종료 상태

    struct list child\_threads; //자식 프로세스 모아두는 링크드리스트

    struct list\_elem child\_elem; //내가 자식 프로세스일 때 부모 프로세스의 child\_threads에서의 포인터 정보

    struct file \*fd\_table[130]; //fd 테이블 (0, 1 + 128개)

    struct file \*execute\_file; //현재 실행 중인 file

#endif

    /\* Owned by thread.c. \*/

    unsigned magic;                     /\* Detects stack overflow. \*/

  };

위와 같이 thread 구조체에 fd\_table 배열을 선언하여 프로세스마다 파일 디스크립터 테이블을 갖도록 한다. 이렇게 선언한 배열을 아래 init\_thread 함수를 통해 NULL로 초기화 한다.

src/threads/thread.c

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

  list\_init(&(t->child\_threads));

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

  sema\_init(&(t->wait\_exit), 0);

  sema\_init(&(t->wait\_free), 0);

  sema\_init(&(t->wait\_load), 0);

  t->exec\_flag = true;

  for(int i = 0; i < 130; i++){

    t->fd\_table[i] = NULL;

  }

#endif

}

**2. System Calls**

**1) lock 선언 및 초기화**

파일과 관련된 시스템 콜을 구현하기 위해서는 우선 파일에 대한 동시 접근을 막을 lock을 선언하고 초기화 해주어야 한다. syscall.c에 전역 변수로 lock을 선언하고 syscall\_init 함수를 통해 이를 초기화하도록 하였다.

syscall.c

struct lock file\_access\_lock;

void

syscall\_init (void)

{

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

  lock\_init(&file\_access\_lock);

}

lock은 세마포어의 특수한 버전이다. 내부적으로 세마포어를 사용하여 동시성 문제를 해결함과 동시에 lock을 걸은 thread에 한해서만 lock을 풀 수 있도록 threads/synch.c에 구현되어 있다. synch.c의 코드를 살펴보면 lock\_init이 내부적으로 sema\_init 함수를 사용하여 1로 초기화를 수행하고 있음을 알 수 있다.

**2) syscall\_handler 수정 및 유효성 검사 함수 선언**

새롭게 추가할 system call들에 대해서 syscall\_handler가 올바르게 동작할 수 있도록 수정해준다. 각각의 system call마다 필요한 인자의 개수를 추가하고, 조건문을 추가하여 system call number에 맞는 system call 함수들을 호출한다.

syscall.c

static void

syscall\_handler (struct intr\_frame \*f)

{

  //syscall number마다 필요한 argc 저장하는 배열

  int arg\_cnt[22];

  //prj1

  arg\_cnt[SYS\_HALT] = 0;

  arg\_cnt[SYS\_EXIT] = 1;

  arg\_cnt[SYS\_EXEC] = 1;

  arg\_cnt[SYS\_WAIT] = 1;

  arg\_cnt[SYS\_READ] = 3;

  arg\_cnt[SYS\_WRITE] = 3;

  arg\_cnt[MAX\_OF\_FOUR\_INT] = 4;

  arg\_cnt[FIBONACCI] = 1;

  //prj2

  arg\_cnt[SYS\_CREATE] = 2;

  arg\_cnt[SYS\_REMOVE] = 1;

  arg\_cnt[SYS\_OPEN] = 1;

  arg\_cnt[SYS\_CLOSE] = 1;

  arg\_cnt[SYS\_FILESIZE] = 1;

  arg\_cnt[SYS\_SEEK] = 2;

  arg\_cnt[SYS\_TELL] = 1;

  uint32\_t\* esp = (uint32\_t\*)f->esp;

  check\_valid\_addr(esp, arg\_cnt[\*esp]);

  //prj1

  if (\*esp == SYS\_HALT) {

    halt();

  }

  else if (\*esp == SYS\_EXIT) {

    int status = (int)\*(esp + 1);

    exit(status);

  }

  else if (\*esp == SYS\_EXEC) {

    char \*cmd\_line = (char\*)\*(esp + 1);

    f->eax = exec(cmd\_line);

  }

  else if (\*esp == SYS\_WAIT) {

    tid\_t pid = (tid\_t)\*(esp + 1);

    f->eax = wait(pid);

  }

  else if (\*esp == SYS\_READ) {

    int fd = (int)\*(esp + 1);

    void\* buffer = (void\*)\*(esp + 2);

    unsigned size = (unsigned)\*(esp + 3);

    f->eax = read(fd, buffer, size);

  }

  else if (\*esp == SYS\_WRITE) {

    int fd = (int)\*(esp + 1);

    void\* buffer = (void\*)\*(esp + 2);

    unsigned size = (unsigned)\*(esp + 3);

    f->eax = write(fd, buffer, size);

  }

  else if(\*esp == MAX\_OF\_FOUR\_INT){

    int a = (int)\*(esp + 1);

    int b = (int)\*(esp + 2);

    int c = (int)\*(esp + 3);

    int d = (int)\*(esp + 4);

    f->eax = max\_of\_four\_int(a, b, c, d);

  }

  else if(\*esp == FIBONACCI){

    int n = (int)\*(esp + 1);

    f->eax = fibonacci(n);

  }

  //prj2

  else if(\*esp == SYS\_CREATE){

    char \*file = (char\*)\*(esp + 1);

    unsigned initial\_size = (unsigned)\*(esp + 2);

    f->eax = create(file, initial\_size);

  }

  else if(\*esp == SYS\_REMOVE){

    char \*file = (char\*)\*(esp + 1);

    f->eax = remove(file);

  }

  else if(\*esp == SYS\_OPEN){

    char \*file = (char\*)\*(esp + 1);

    f->eax = open(file);

  }

  else if(\*esp == SYS\_CLOSE){

    int fd = (int)\*(esp + 1);

    close(fd);

  }

  else if(\*esp == SYS\_FILESIZE){

    int fd = (int)\*(esp + 1);

    f->eax = filesize(fd);

  }

  else if(\*esp == SYS\_SEEK){

    int fd = (int)\*(esp + 1);

    unsigned position = (unsigned)\*(esp + 2);

    seek(fd, position);

  }

  else if(\*esp == SYS\_TELL){

    int fd = (int)\*(esp + 1);

    f->eax = tell(fd);

  }

  else {

    exit(-1);

  }

}

또 각각의 system call 함수들을 구현할 때 필요한 유효성 검사 함수도 새롭게 선언한다. 검사해야 하는 경우는 filnename이 NULL인 경우, 파일 디스크립터 테이블에 저장된 file pointer가 NULL인 경우, fd값이 3 이하이거나 130 이상인 경우로 총 3가지이다. 파일 시스템 콜을 구현함에 있어서 해당 사항을 체크해주어야 하는 경우가 많으므로, 편의를 위해 함수를 새롭게 선언하였다. 이 함수들은 각각의 경우를 체크하고, 때에 따라 조건을 위반하면 exit(-1)을 호출한다.

syscall.c

//filename이 NULL인 경우 체크

void check\_valid\_filename(const char \*file){

  if(file == NULL) exit(-1);

}

//file ptr이 NULL인 경우 체크

void check\_valid\_file(struct file \*file\_ptr){

  if(file\_ptr == NULL) exit(-1);

}

void check\_valid\_fd(int fd){

  if(fd < 3 || fd >= 130) exit(-1);

}

**3) create**

syscall.c

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

  check\_valid\_filename(file);

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

}

create는 파일을 새롭게 생성하는 시스템 콜 함수이다. 우선 check\_valid\_filename 함수를 통해 인자로 받은 file(실질적으로는 파일명을 가리키는 포인터)의 NULL 여부를 확인한다. 이후 filesys\_create 함수를 사용한다. filesys\_create는 filesys/filesys.c에 선언되어 있는 함수로 새롭게 파일을 생성하고 성공 여부를 bool type으로 반환한다.

**4) remove**

syscall.c

bool remove(const char \*file){

  check\_valid\_filename(file);

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

  return filesys\_remove(file);

}

remove 함수도 create와 유사한 패턴을 갖는다. check\_valid\_filename 함수로 인자의 NULL 여부를 확인하고, 유효한 파일명이라면 filesys\_remove를 호출한다. filesys\_remove는 filesys/filesys.c에 선언되어 있는 함수로 파일을 삭제하고 성공 여부를 bool type으로 반환한다. 커널 영역에 있는 파일 삭제를 시도하는 경우를 방지하기 위해 유저 영역에 있는 포인터인지 확인하는 과정을 추가했다.

**5) open**

syscall.c

int open(const char \*file){

  check\_valid\_filename(file);

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

  //file 열 때, 다른 곳에서 file 수정하는 것 방지

  lock\_acquire(&file\_access\_lock);

  struct file\* open\_file = filesys\_open(file);

  //file 열기에 실패한 경우

  if(open\_file == NULL){

    lock\_release(&file\_access\_lock);

    return -1;

  }

  //file 열기에 성공했으면 fd\_table에 넣어주어야 함

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

  int i;

  for(i = 3; i < 130; i++){

    if(current\_thread->fd\_table[i] == NULL){

      current\_thread->fd\_table[i] = open\_file;

      break;

    }

  }

  //fd\_table이 모두 찬 경우

  if(i == 130) {

    file\_close(open\_file);

    lock\_release(&file\_access\_lock);

    return -1;

  }

  lock\_release(&file\_access\_lock);

  return i;

}

open 시스템 콜은 파일을 열고, 그 파일의 파일 디스크립터, 즉 fd를 반환한다. open 시스템 콜의 동작 수행 전, 파일명의 NULL 여부와 해당 파일명을 가리키는 포인터가 user space인지 확인하는 과정이 필요하다. 이는 커널 영역에 있는 파일을 읽는 것을 방지하기 위함이다.

이후 파일을 열 때 lock\_acquire 함수를 통해 lock을 걸고, 파일을 열고 fd를 할당하는 작업이 종료된 후 lock\_release 함수를 통해 lock을 풀어야 한다. lock\_acquire과 lock\_release 함수는 threads/synch.c에서 확인할 수 있으며, 특히 lock\_release 함수의 경우에는 내부적으로 현재 실행 중인 thread에 의해 발생하는 lock 해제인지를 체크하는 과정이 포함되어 있다.

open 시스템 콜에서는 크게 두 가지 경우로 나누어서 생각해야 한다. 먼저 파일 열기에 실패한 경우 lock\_release를 수행하고 -1을 반환한다. 파일 열기에 성공했다면 반복문을 돌며 현재 thread의 fd table에서 NULL인 가장 작은 fd값을 찾는다. 만약 fd table이 모두 다 찼다면 fd를 할당하고자 했던 파일을 다시 닫고 lock을 해제한 후 -1을 반환한다. 이외의 경우는 정상적으로 fd 할당이 이루어졌으므로 fd값을 반환한다.

**6) close**

syscall.c

void close(int fd){

  check\_valid\_fd(fd);

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

  struct file\* target\_file = current\_thread->fd\_table[fd];

  check\_valid\_file(target\_file);

  file\_close(target\_file);

  current\_thread->fd\_table[fd] = NULL;

}

close 함수는 기본적으로 인자로 받은 fd에 해당하는 파일을 닫는 작업을 수행한다. check\_valid\_fd 함수를 통해 fd가 유효한 범위 내에 있는지 확인한다. 이후 현재 실행 중인 thread의 fd table에서 해당 fd 번호에 해당하는 파일 포인터를 가져와 유효한 파일 포인터인지 확인한다. 만약 NULL이 아닌 파일 포인터라면 file\_close 함수를 사용하여 해당 파일을 닫고, thread의 fd table에서 인자로 전달받은 fd에 해당하는 칸을 NULL로 설정해준다. 이때 file\_close 함수는 filesys/file.c 내부에 선언되어 있는 함수로, 내부적으로 file\_allow\_write 함수를 호출하여 쓰기 권한을 부여하고 있다.

pintos manual에 따르면 특정 프로세스가 종료될 때, 해당 프로세스에 열려 있는 open file들이 모두 닫혀야 한다. 이를 구현하기 위해 아래와 같이 process.c의 process\_exit 함수를 수정한다. 코드에 file\_close(cur->execute\_file)의 경우에는 synchronization과 관련된 부분이므로 본 보고서의 뒷 부분에서 설명하도록 한다.

process.c

void

process\_exit (void)

{

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

  uint32\_t \*pd;

  //현재 실행 중인 파일 닫기 = file\_allow\_write 수행

  file\_close(cur->execute\_file);

  //프로세스 종료할 때, 갖고 있는 모든 open file 닫기

  for(int i = 3; i < 130; i++){

    if(cur->fd\_table[i] != NULL){

      file\_close(cur->fd\_table[i]);

      cur->fd\_table[i] = NULL;

    }

  }

  /\* Destroy the current process's page directory and switch back

     to the kernel-only page directory. \*/

  pd = cur->pagedir;

  if (pd != NULL)

    {

      /\* Correct ordering here is crucial.  We must set

         cur->pagedir to NULL before switching page directories,

         so that a timer interrupt can't switch back to the

         process page directory.  We must activate the base page

         directory before destroying the process's page

         directory, or our active page directory will be one

         that's been freed (and cleared). \*/

      cur->pagedir = NULL;

      pagedir\_activate (NULL);

      pagedir\_destroy (pd);

    }

  sema\_up(&(cur->wait\_exit)); //up하는 순간 부모 프로세스에서 동작 시작

  sema\_down(&(cur->wait\_free)); //부모 프로세스에서 삭제한 뒤 메모리가 풀리도록 대기

}

**7) filesize**

syscall.c

int filesize(int fd){

  check\_valid\_fd(fd);

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

  struct file\* target\_file = current\_thread->fd\_table[fd];

  check\_valid\_file(target\_file);

  return file\_length(target\_file);

}

filesize 함수는 fd에 해당하는 파일의 크기를 반환한다. close 함수와 동일하게 fd의 범위와 fd에 해당하는 파일 포인터에 대해 유효성 검사를 수행한다. 유효성 검사를 통과하면 file\_length 함수를 통해 파일의 크기를 반환한다. file\_length 함수는 filesys/file.c에 구현되어 있는 함수로 inode 테이블에 접근해 파일의 크기를 반환한다.

**8) seek**

syscall.c

void seek(int fd, unsigned position){

  check\_valid\_fd(fd);

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

  struct file\* target\_file = current\_thread->fd\_table[fd];

  check\_valid\_file(target\_file);

  return file\_seek(target\_file, position);

}

seek 함수는 fd에 해당하는 파일에 대해 읽기나 쓰기 위치를 position으로 변경한다. fd의 범위와 fd에 해당하는 파일 포인터에 대해 유효성 검사를 수행하고 file\_seek 함수를 호출한다. file\_seek 함수는 filesys/file.c에 구현되어 있는 함수로 file 구조체의 pos 필드를 새로운 position으로 바꿔준다.

**9) tell**

syscall.c

unsigned tell(int fd){

  check\_valid\_fd(fd);

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

  struct file\* target\_file = current\_thread->fd\_table[fd];

  check\_valid\_file(target\_file);

  return file\_tell(target\_file);

}

tell 함수는 fd에 해당하는 파일에서 다음에 읽거나 쓸 위치를 반환한다. fd의 범위와 fd에 해당하는 파일 포인터에 대해 유효성 검사를 수행하고 file\_tell 함수를 호출한다. file\_tell 함수는 filesys/file.c에 구현되어 있는 함수로 file 구조체의 pos 필드값을 반환한다.

**10) read**

syscall.c

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

  if(fd < 0 || fd == 1 || fd >= 130) exit(-1);

  //buffer 주소 체크

  check\_valid\_addr(buffer, 1);

  lock\_acquire(&file\_access\_lock);

  //stdin인 경우, 키보드 입력을 읽음

  if(fd == 0){

    int i;

    uint8\_t temp;

    for(i = 0; (i < size) && (temp = input\_getc()); i++){

      \*(uint8\_t\*)(buffer + i) = temp;

    }

    lock\_release(&file\_access\_lock);

    return i;

  }

  //그 이외의 입력

  else{

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

    struct file\* target\_file = current\_thread->fd\_table[fd];

    if(target\_file == NULL || fd == 2){

      lock\_release(&file\_access\_lock);

      exit(-1);

    }

    int byte = file\_read(target\_file, buffer, size);

    lock\_release(&file\_access\_lock);

    return byte;

  }

}

read 시스템 콜은 기존의 시스템 콜을 수정하여 다른 파일에 대한 입력 또한 가능하도록 구현하였다. read의 동작을 수행하기에 앞서서 fd의 범위에 대한 확인이 필요하다. fd가 음수인 경우, fd가 1인 경우(STDOUT), fd가 최대 범위를 넘어가는 경우에 대해서는 exit(-1)을 호출하여 예외처리를 해주었다. 또 읽어 들이는 buffer의 주소 유효성 검사도 수행해야 한다. 이때 이전 프로젝트에서 사용하였던 check\_valid\_addr 함수를 사용하였다.

read는 파일 데이터 변경과 관련된 시스템 콜이기 때문에 read 로직 수행 전에 lock을 걸고, 수행 후 lock을 해제하는 과정이 필요하다. read는 크게 두 가지 경우로 나뉜다. 먼저 fd가 0인 경우이다. fd가 0인 경우, 즉 STDIN인 경우 키보드로부터 입력을 받는다. 이전 프로젝트에서 구현하였던 코드를 if 조건문 안으로 넣어주었다. else 부분이 그 외의 입력으로 이번 프로젝트에서 새롭게 추가된 부분이다. 유효하지 않은 파일 포인터인 경우와 fd가 2인 경우(2번 fd는 STDERR로 예약되어 있다고 가정)에는 lock을 해제라고 exit(-1)을 호출하였다. 그 외의 경우에는 유효한 범위 내의 STDIN이 아닌 fd이므로 file\_read 함수를 통해 읽어들인 바이트 수를 반환 받은 다음 lock을 해제한다. 이때 file\_read 함수는 filesys/file.c 내부에 선언되어 있는 함수로 file->pos를 읽은 만큼의 위치로 업데이트 하고 읽은 바이트 수를 반환한다.

**11) write**

syscall.c

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

  if(fd < 0 || fd == 0 || fd >= 130) exit(-1);

  //buffer 주소 체크

  check\_valid\_addr(buffer, 1);

  lock\_acquire(&file\_access\_lock);

  //stdout인 경우, 콘솔 출력

  if(fd == 1){

    putbuf(buffer, size);

    lock\_release(&file\_access\_lock);

    return size;

  }

  //그 이외의 출력

  else{

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

    struct file\* target\_file = current\_thread->fd\_table[fd];

    if(target\_file == NULL || fd == 2){

      lock\_release(&file\_access\_lock);

      exit(-1);

    }

    int byte = file\_write(target\_file, buffer, size);

    lock\_release(&file\_access\_lock);

    return byte;

  }

}

write 함수도 read 함수와 동일한 패턴을 갖는다. 먼저 fd의 범위를 확인하는 과정을 거친다. fd가 0이거나 음수인 경우, 최대 범위를 넘어간 경우에는 exit(-1)을 호출하여 예외처리를 하였다. 또 읽어들이는 buffer에 대해 주소 유효성 검사를 check\_valid\_addr 함수를 통해 진행하였다.

write 함수는 파일의 데이터 변경과 관련된 시스템 콜이기 때문에, 로직 수행 전후로 lock을 걸었다가 해제하는 과정이 필요하다. write 함수의 동작은 크게 두 가지로 나뉜다. 먼저 fd가 1인 경우 STDOUT이므로 콘솔에 출력이 이루어져야 한다. 이전 프로젝트에서 작성하였던 부분을 if문 안에 넣어 구현하였다. 두 번째로 else에 해당하는 부분이다. 이들은 콘솔 출력 이외의 출력으로 다른 파일에 대해 쓰는 작업이다. 유효하지 않은 파일 포인터인 경우와 fd가 2인 경우에는 lock을 해제하고 exit(-1)로 예외처리를 하였다. 그 외의 경우에는 정상적인 fd이므로 file\_write 함수를 통해 쓰기 작업을 수행하고 lock을 해제하였다. 이때 file\_write 함수는 filesys/file.c 내부에 선언되어 있는 함수로 쓴 바이트 수를 반환한다.

**3. Synchronization in Filesystem**

만약 실행 중인 사용자 프로그램에 데이터 쓰기가 발생한다면 이는 동기화 문제를 야기한다. 따라서 실행 중인 사용자 프로그램에 대해 데이터 변경을 막아야 하며, 프로그램이 종료된 상태일 때만 변경이 가능해야 한다. 이를 구현하기 위해 load 함수를 수정했다. load 함수에서 실행할 파일을 열 때, 실행 중인 파일에 대해 file\_deny\_write 함수를 호출하여 현재 프로세스를 제외한 다른 프로세스가 해당 파일에 접근하지 못하도록 한다. file\_deny\_write는 filesys/file.c에 있는 함수로 file 구조체의 deny\_write 필드에 true값을 할당한다. 이 함수를 호출하고 나면 파일 변경이 불가능하다.

process.c

bool

load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp)

{

  struct thread \*t = thread\_current ();

  struct Elf32\_Ehdr ehdr;

  struct file \*file = NULL;

  off\_t file\_ofs;

  bool success = false;

  int i;

  /\* Allocate and activate page directory. \*/

  t->pagedir = pagedir\_create ();

  if (t->pagedir == NULL)

    goto done;

  process\_activate ();

  //file\_name 파싱

  int argc = 0;

  char\* argv[64];

  char\* next\_ptr;

  char tmp[128];

  strlcpy(tmp, file\_name, strlen(file\_name) + 1);

  char\* ret\_ptr = strtok\_r(tmp, " ", &next\_ptr);

  while(ret\_ptr){

    argv[argc++] = ret\_ptr;

    ret\_ptr = strtok\_r(NULL, " ", &next\_ptr);

  }

  argv[argc] = NULL;

  file\_name = argv[0];

  /\* Open executable file. \*/

  file = filesys\_open (file\_name);

  if (file == NULL)

    {

      printf ("load: %s: open failed\n", file\_name);

      goto done;

    }

  t->execute\_file = file;

  file\_deny\_write(file);

  /\*중간 생략\*/

  /\* Set up stack. \*/

  //4KB의 stack page 할당하는 함수

  //void \*\*esp

  //\*esp = PHYS\_BASE;

  if (!setup\_stack (esp))

    goto done;

  //스택 쌓기

  push\_stack(argv, argc, esp);

  /\* Start address. \*/

  \*eip = (void (\*) (void)) ehdr.e\_entry;

  success = true;

 done:

  /\* We arrive here whether the load is successful or not. \*/

  //file\_close (file);

  return success;

}

이때 load 함수의 마지막 부분에 있는 done 부분의 file\_close 함수를 주석처리 해주어야 한다. 해당 done 부분은 load 성공 여부와 관계없이 무조건 도달하게 되는데, file\_close는 내부적으로 file\_allow\_write 함수를 호출하여 다시 해당 파일에 데이터 변경이 가능하도록 한다. 따라서 주석처리를 해주지 않으면 file\_deny\_write를 수행한 의미가 없어지게 된다. 이를 주석처리 하고 load가 성공했을 때 다시 파일 변경이 가능하도록 process\_exit 함수를 수정한다. 즉, 프로세스가 종료되는 시점에 쓰기 권한을 다시 부여하는 것이다.

process.c

void

process\_exit (void)

{

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

  uint32\_t \*pd;

  //현재 실행 중인 파일 닫기 = file\_allow\_write 수행

  file\_close(cur->execute\_file);

  //프로세스 종료할 때, 갖고 있는 모든 open file 닫기

  for(int i = 3; i < 130; i++){

    if(cur->fd\_table[i] != NULL){

      file\_close(cur->fd\_table[i]);

      cur->fd\_table[i] = NULL;

    }

}

/\*중간 생략\*/

}

process.c에서 현재 실행 중인 파일을 알기 위해 thread 구조체에 현재 실행 중인 파일을 담는 truct file \*execute\_file; 필드를 새롭게 추가해야 한다.

또, 중요한 것은 부모 프로세스와 자식 프로세스 간의 동기화 문제이다. 자식 프로세스가 메모리에 load 되는 동안 부모 프로세스는 진행되지 않아야 한다. 즉, 자식 프로세스가 load 되는 동안 부모 프로세스의 종료를 막아야 한다. 이를 구현하기 위해 thread 구조체에 wait\_load라는 명칭의 세마포어를 새롭게 추가해야 하고 exec 시스템 콜 함수를 수정해야 한다. 수정 사항을 반영한 코드는 다음과 같다. wait\_load 세마포어는 init\_thread 함수에서 0으로 초기화를 수행한다.

thread.h

struct thread

  {

    /\* Owned by thread.c. \*/

    tid\_t tid;                          /\* Thread identifier. \*/

    enum thread\_status status;          /\* Thread state. \*/

    char name[16];                      /\* Name (for debugging purposes). \*/

    uint8\_t \*stack;                     /\* Saved stack pointer. \*/

    int priority;                       /\* Priority. \*/

    struct list\_elem allelem;           /\* List element for all threads list. \*/

    /\* Shared between thread.c and synch.c. \*/

    struct list\_elem elem;              /\* List element. \*/

#ifdef USERPROG

    /\* Owned by userprog/process.c. \*/

    uint32\_t \*pagedir;                  /\* Page directory. \*/

    struct semaphore wait\_exit; //자식 프로세스 종료 대기

    struct semaphore wait\_free; //메모리 해제 대기

    struct semaphore wait\_load; //자식 프로세스 load 대기

    bool exec\_flag; //자식 프로세스가 정상적으로 exec 되었는지 표시하는 flag : true = 정상 실행

    int exit\_status; //종료 상태

    struct list child\_threads; //자식 프로세스 모아두는 링크드리스트

    struct list\_elem child\_elem; //내가 자식 프로세스일 때 부모 프로세스의 child\_threads에서의 포인터 정보

    struct file \*fd\_table[130]; //fd 테이블 (0, 1 + 128개)

    struct file \*execute\_file; //현재 실행 중인 file

#endif

    /\* Owned by thread.c. \*/

    unsigned magic;                     /\* Detects stack overflow. \*/

  };

syscall.c

int exec(char \*cmd\_line){

  tid\_t child\_tid = process\_execute(cmd\_line);

  if(child\_tid != -1){

    struct list\_elem\* elem;

    struct list\* child\_list = &(thread\_current()->child\_threads);

    struct thread\* child;

    for(elem = list\_begin(child\_list); elem != list\_end(child\_list); elem = list\_next(elem)){

      child = list\_entry(elem, struct thread, child\_elem);

      if(child->tid == child\_tid){

        //자식이 메모리에 load되는 동안 부모 동작 제한

        //이 코드를 실행시키는 주인이 부모이므로 sema\_down을 사용해서

        //여기서 부모 프로세스가 대기하도록 구현

        sema\_down(&(child->wait\_load));

      }

    }

    if(child->exec\_flag == false) return -1; //자식 프로세스 load에 실패한 경우

  }

  return child\_tid;

}

sema\_down으로 부모 프로세스가 대기하고 있으므로 sema\_up은 자식 프로세스가 load 동작을 수행한 후에 이루어져야 한다. 여기서 주의해야할 점은 sema\_up이 자식 프로세스의 load 작업이 종료되었음을 의미할 뿐, 자식 프로세스의 load가 성공했음을 의미하지는 않는다는 것이다. 즉, sema\_up 이후에 자식 프로세스의 load 성공 여부를 확인해서 exec의 반환값을 조정해야 한다. 이를 start\_process 함수에 구현하였다. 자식 프로세스의 load 성공 여부를 확인하기 위해 thread 구조체에 새롭게 exec\_flag 필드를 추가하였다. exec\_flag 필드가 false인 경우 자식 프로세스 load의 실패를 의미한다. exec\_flag는 init\_thread 함수에서 true로 초기화하였다. 또 자식 프로세스의 load 실패 시 thread\_exit 함수가 아닌 exit(-1) 함수를 호출하여 종료 상태 코드가 저장되도록 하였다.

process.c

static void

start\_process (void \*file\_name\_)

{

  char \*file\_name = file\_name\_;

  struct intr\_frame if\_;

  bool success;

  /\* Initialize interrupt frame and load executable. \*/

  memset (&if\_, 0, sizeof if\_);

  if\_.gs = if\_.fs = if\_.es = if\_.ds = if\_.ss = SEL\_UDSEG;

  if\_.cs = SEL\_UCSEG;

  if\_.eflags = FLAG\_IF | FLAG\_MBS;

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

  //자식 load 후 sema\_up

  sema\_up(&(thread\_current()->wait\_load));

  /\* If load failed, quit. \*/

  palloc\_free\_page (file\_name);

  if (!success) {

    thread\_current()->exec\_flag = false;

    //thread\_exit ();

    exit(-1);

  }

  /\* Start the user process by simulating a return from an

     interrupt, implemented by intr\_exit (in

     threads/intr-stubs.S).  Because intr\_exit takes all of its

     arguments on the stack in the form of a `struct intr\_frame',

     we just point the stack pointer (%esp) to our stack frame

     and jump to it. \*/

  asm volatile ("movl %0, %%esp; jmp intr\_exit" : : "g" (&if\_) : "memory");

  NOT\_REACHED ();

}

**4. 개발 중 발생한 문제나 이슈 : multi-oom**

동기화 문제를 모두 해결하였다고 생각하고 테스트를 진행하던 도중 지속적으로 multi-oom 테스트를 통과하지 못했다. 그래서 multi-oom.c 파일을 살펴본 결과, 여러 개의 자식 프로세스가 실행이 되고 load에 실패하였을 때 메모리 공간을 얼마나 효율적으로 관리하는지를 검사한다는 것을 알 수 있었다. 즉, 메모리가 부족해 더 이상 프로세스를 생성할 수 없는 상황에서 프로세스 생성 실패 시 exec 함수가 -1을 반환하는지 확인하는 것이다. 이를 위해서 부모 프로세스에서 exec 함수 호출 시 child\_thread에 있는 모든 자식 프로세스에 대해 exec\_flag를 검사하고, exec\_flag가 false인 경우에 process\_wait을 호출하여 리소스를 회수할 수 있도록 하였다.

process.c

tid\_t

process\_execute (const char \*file\_name)

{

  /\*중간 생략\*/

  /\* Create a new thread to execute FILE\_NAME. \*/

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

  if (tid == TID\_ERROR)

    palloc\_free\_page (fn\_copy);

  struct list\_elem\* elem;

  struct list\* child\_list = &(thread\_current()->child\_threads);

  for(elem = list\_begin(child\_list); elem != list\_end(child\_list); elem = list\_next(elem)){

    struct thread\* child = list\_entry(elem, struct thread, child\_elem);

    if(child->exec\_flag == false) return process\_wait(tid);

  }

  return tid;

}

* 1. **시험 및 평가 내용**

스크린샷, 패턴, 텍스트, 패브릭이(가) 표시된 사진

자동 생성된 설명

스크린샷, 패턴, 텍스트, 패브릭이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 그래픽 소프트웨어이(가) 표시된 사진

자동 생성된 설명