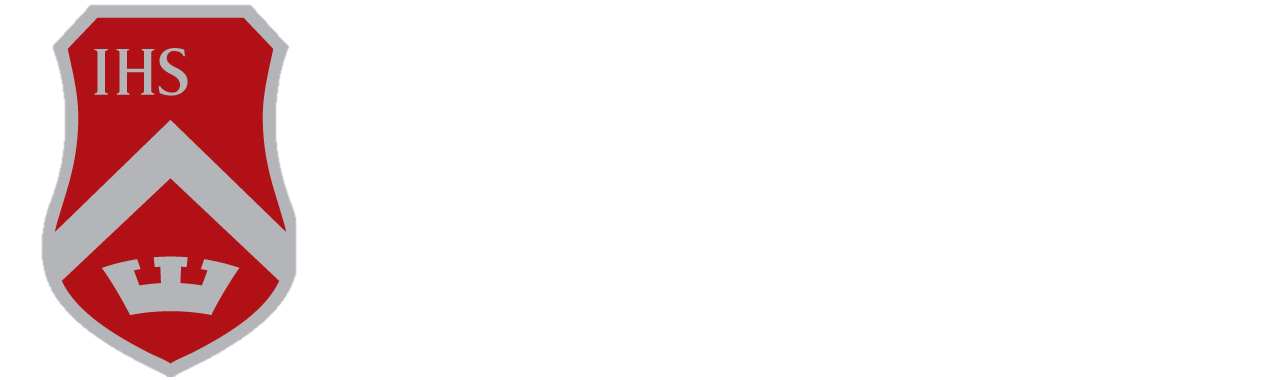
**2020 Fall- Operating System**

**Pintos Project 2**

**User Program (2)**

****

|  |  |
| --- | --- |
| 담당 교수 / 분반 : | 김영재 교수 / 2반 |
| 이름 / 학번 : | 김지수 / 20160366 |
| 개발 기간 : | 2020.11.4 ~ 11/19 |
|  |  |

목차

1. **개발 목표**
2. **개발 범위 및 내용**
   1. **개발 범위**
      1. File Descriptor
      2. System Calls
      3. Synchronization in Filesystem
   2. **개발 내용**
3. File Descriptor
4. System Calls
5. Synchronization in Filesystem
6. **추진 일정 및 개발 방법**
   1. **추진 일정**
   2. **개발 방법**
7. **연구 결과**
   1. **제작 내용**
      * 1. File Descriptor
        2. System Calls
        3. Synchronization in Filesystem
   2. **시험 및 평가 내용**

* make check 수행 결과

**1. 개발 목표**

현재 핀토스에는 User Program을 작동할 수 있도록 도와주는 기본적인 OS의 기능들이 구현되어 있지 않다. 그 중 이번 프로젝트2에서는 User Program이 file과 directory를 사용할 때 사용하는 커널의 기능을 위주로 구현한다. create, remove, open, close, filesize, read, write, seek, tell의 시스템콜을 사용할 수 있도록 함수와 인터페이스를 만들고, 그를 위해 필요한 file descriptor와 synchronization 기능도 함께 구현한다.

**2. 개발 범위 및 내용**

1. **개발 범위**
2. **File Descriptor**

커맨드 라인에서 프로세스 이름을 확인하여 그 이름으로 스레드를 만들고, 커맨드 라인을 Parsing하여 인자를 확인 하고, 인자를 스택에 push.

1. **System Calls**

user program이 요청한 virtual address가 valid한지 확인하고 invalid한 경우 프로세스 종료.

1. **Synchronization in File System**

0x30번으로 호출된 interrupt에 대해, interrupt handler가 system call number를 확인하여 알맞은 system call 함수를 호출하여 execute, exit, wait, read, write이 실행된다. 또한 추가적으로 Fibonacci, max\_of\_four\_int의 system call도 구현됨.

1. **개발 내용**
2. **File Descriptor**

프로세스가 open하는 파일 마다 file descriptor number가 주어지고, 한 프로세스는 여러 file을 관리할 수 있어야 하므로 한 프로세스가 가지는 file descriptor들은 리스트로 관리되어야 한다. 따라서 struct thread에 file descriptor들의 리스트를 가리키는 포인터 변수로 file descriptor table을 만든다. 또한 스레드가 새로운 파일을 open하게 되는 경우를 고려해 부여할 수 있는 next file descriptor number 또한 field로 가지고 있는다.

struct file\_descriptor\_entry를 구현하여 각 스레드의 file descriptor table의 entry를 정의한다. table에 연결시킬 list\_elem과, 해당 파일에 대한 file descriptor number, 해당 파일에 대한 포인터가 entry 구조체 안에 존재한다.

1. **System Calls**

* **static bool sys\_read(int fd, void \*buffer, unsigned size)**: fd가 0일 때 키보드의 데이터를 읽어 buffer에 저장하고, 다른 fd에 대해서는 파일에 있는 데이터를 읽어 buffer에 저장하는 시스템 콜. size는 읽을 데이터의 크기이며, 성공 시 읽은 바이트 수를 반환하지만 실패하면 -1을 반환한다.
* **static bool sys\_write(int fd, void \*buffer, unsigned size)**: fd가 1일 때 buffer에 저장된 데이터를 화면에 출력하고, 다른 fd에 대해서는 buffer의 데이터를 파일에 기록하는 시스템 콜. size는 출력할 데이터의 크기이며, 성공 시 출력한 바이트 수를 반환하지만 실패하면 -1을 반환한다.
* **static bool sys\_create(const char \*file, unsigned initial\_size)**: 초기 사이즈를 n으로 하고 file을 name으로 가지는 파일을 생성하는 시스템 콜.
* **static bool sys\_remove(const chart \*file)**: file을 이름으로 하는 파일을 제거하는 시스템 콜.
* **static bool sys\_open(const chart \*file)**: file을 이름으로 하는 파일을 open하는 시스템 콜.
* **static bool sys\_filesize(int fd)**: fd를 file descriptor number로 가지는 파일을 찾아 그 사이즈를 반환하는 시스템 콜.
* **static void sys\_seek(int fd, unsigned position)**: fd를 file descriptor number로 가지는 파일을 찾아, 다음 byte을 position으로 업데이트 하는 시스템 콜.
* **static unsigned sys\_tell(int fd)**: fd를 file descriptor number로 가지는 파일을 찾아, 다음 byte의 position을 반환하는 시스템 콜
* **void sys\_close(int fd)**: fd를 file descriptor number로 가지는 파일을 찾아, 존재한다면 파일을 닫고 entry를 table(리스트)에서 제거하고, 공간을 해제하는 시스템 콜.

1. **Synchronization in File System**

file system에서 어떤 file에 접근하는 것 또한 critical section에 진입하는 것이므로, 영역 진입에 제한을 둬야 한다. 한 파일에 동시에 두 개 이상의 프로세스가 읽는 경우는 문제가 없지만, 읽는 중에 다른 프로세스가 작성을 하거나 두 프로세스가 동시에 작성하게 되면 동기화가 되지 않아 문제가 생기기 때문에 영역에 들어가면서 다른 프로세스들은 해당 file에 접근할 수 없도록 막는 수단이 필요한데, 그 방법으로는 semaphore와 lock이 있다.

semaphore를 활용하게 되면, critical section에 들어가기 전에 sema\_down을 해주고 그 영역을 다 사용 후 빠져나오면서 sema\_up을 함으로써 다른 프로세스들의 접근을 가능하게 할 수 있다. lock을 사용하는 경우 또한 비슷하게, lock\_acquire를 통해 권한을 얻고 파일에 접근할 수 있으며 사용이 끝나고는 lock\_release를 통해 lock을 다른 프로세스가 취할 수 있게 풀어주어야 한다. 이번 프로젝트2에서는 lock을 활용한 동기화를 구현하였다.

**3. 추진 일정 및 개발 방법**

* + 1. **일정**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 11.04 | 11.05 | 11.06 | 11.07 | 11.08 | 11.09 | 11.10 |
| Pintos Manual 읽기 | | | File Descriptor 초안 작성 | | | System |
| 11.11 | 11.12 | 11.13 | 11.14 | 11.15 | 11.16~19 | 11.19~20 |
| Calls 초안 작성 | | Synchronization in Filesystem 초안 작성 | | | 디버깅 | 보고서 작성 |

* + 1. **개발 방법**

1. **File Descriptor**

(1) **src/userprog/process.h/struct file\_descriptor\_entry**을 구현하여 프로세스 마다 관리하는 file에 대한 정보를 구조체로 정의한다.

(2) **src/threads/thread.h/struct thread**에 해당 스레드가 관리하는 file descriptor entry에 대한 리스트 포인터와 다음 새롭게 열릴 file에 부여할 file descriptor number를 보유할 변수를 추가 선언한다.

(3) **src/threads/thread.c**에 **struct list\* thread\_get\_fdt(void)**를 구현하여 현재 스레드의 file descriptor table 리스트 포인터 fdt를 반환하도록 한다. 같은 디렉토리에 **int thread\_get\_next\_fdt(void)**도 구현하여 현재 스레드의 다음 file descriptor number를 반환하도록 한다.

**static void init\_thread(struct thread \*t, const char \*name, int priority)**에 새롭게 추가한 struct thread의 member를 초기화하는 코드를 추가한다.

(4) **src/userprog/process.c/void process\_exit(void)**에 스레드가 종료될 때 해당 스레드가 관리하는 열린 file들을 모두 종료하는 코드를 추가한다. 이 때 사용되는 sys\_close 시스템콜 함수는 뒷부분에서 구현에 대해 설명한다.

**struct file\_descriptor\_entry \*process\_get\_fd(int fd)**를 구현하여 현재 실행 중인 스레드의 file 중에 인자로 넘어오는 fd가 일치하는 file descriptor entry를 찾아 일치하는 것이 있다면 그 포인터를, 없다면 NULL을 반환한다.

1. **System Calls**

(1) **src/userprog/syscall.c/static bool sys\_read(int fd, void \*buffer, unsigned size)**에 fd가 0이 아닌 경우 해당 fd의 file의 데이터를 buffer에 읽을 수 있도록 코드를 추가한다.

**static bool sys\_write(int fd, void \*buffer, unsigned size)**에 fd가 1이 아닌 경우 buffer의 데이터를 fd와 일치하는 file에 쓸 수 있도록 코드를 추가한다.

**static bool sys\_create(const char \*file, unsigned initial\_size)**를 구현하여 주어진 크기로 file을 만드는 system call을 위한 함수를 만든다.

**static bool sys\_remove(const chart \*file)**을 구현하여 주어진 이름에 해당하는 파일을 만드는 system call을 위한 함수를 만든다.

**static bool sys\_open(const chart \*file)**을 통해 주어진 이름에 해당하는 파일을 여는 system call을 위한 함수를 만든다.

**static bool sys\_filesize(int fd)**을 통해 주어진 fd와 일치하는 file descriptor number를 가진 file entry를 탐색하여 찾은 경우에 그 파일의 크기를 byte단위로 반환한다.

**static void sys\_seek(int fd, unsigned position)**을 통해 해당 fd의 file을 찾아, 읽거나 쓸 다음 위치의 byte를 position으로 바꾸는 함수를 만든다.

**static unsigned sys\_tell(int fd)** 을 통해 해당 fd의 file을 찾아, 읽거나 쓸 다음 byte의 position을 반환하도록 한다.

**void sys\_close(int fd)**을 구현해 해당 fd의 file을 찾아 존재한다면 file을 닫고, 해당 entry를 file descriptor table에서 제거하고, 할당된 페이지를 해제하도록 함수를 만든다.

**static void syscall\_handler(struct intr\_frame \*f)**에 file에 관련된 system call을 위한 새롭게 추가한 함수들을 호출하는 case문들 추가한다.

1. **Synchronization in File System**

(1) **userprog/syscall.c/static struct lock filesys\_lock**를 새롭게 정의하여 syscall.c에서 system call 기능을 수행할 때와 프로세스의 로딩 시 deny writing을 할 때 사용할 수 있도록 한다.

(2) **threads/thread.h/struct thread**에 현재 실행 중인 file에 대한 정보 field를 추가 한다.

(3) **threads/thread.c**에 **void thread\_set\_running\_file(struct file \*new\_running\_file)**를 통해 현재 스레드의 running file filed를 인자로 넘어온 파일 포인터로 업데이트 한다.

**struct file \*thread\_get\_running\_file(void)**을통해 현재 스레드의 running file을 반환한다.

(4) **userprog/process.c/void process\_exit(void)**에, 프로세스가 종료되면서 실행 중이던 executable file을 close할 때 또한 lock을 얻은 후 진행하도록 코드를 추가한다.

**static bool load(const char \* file\_name, void (\*\*eip)(void), void \*\*esp)**에. 프로세스가 로딩되면서 현재 실행 중인 파일로 로딩할 executable file을 등록하고 로딩되자마자 write할 수 없도록 한다.

**4. 연구 결과**

1. **제작 내용**
2. **File Descriptor**

(1) src/userprog/process.h/struct file\_descriptor\_entry

|  |
| --- |
| **struct file\_descriptor\_entry**  **{**  **int fd;** /\* File descriptor. \*/  **struct file \*file;** /\* File. \*/  **struct list\_elem fdt\_elem;** /\* List elements for file descriptor table. \*/  **};** |

프로세스 마다 관리하는 file descriptor를 file descriptor entry 구조체로 정의한다. 이 구조체 안에는 file descriptor number와 file에 대한 포인터, thread에서 관리하는 file descriptor table 리스트에 연결되기 위한 list element가 포함된다.

(2) src/thread/thread.h/struct thread

|  |
| --- |
| **struct thread**  **{**  **︙**  #ifdef USERPROG  uint32\_t \* pagedir; /\* Page directory. \*/  struct process \*pcb; /\* Process Control Block. \*/  struct list children; /\* List of children process. \*/  **int next\_fdt;** /\* File descriptor number for next new file. \*/  **struct list \*fdt**; /\* File descriptor table. \*/  #endif  **︙**  **};** |

해당 스레드가 관리하는 file들을 관리하기 위해 file descriptor들을 리스트로 관리하는 list 포인터 fdt가 있으며, 다음 새롭게 열릴 파일에게 부여할 file descriptor number를 next\_fdt로 보유하고 있다. 이 번호는 표준 입출력에 부여된 0과 1을 제외한 2부터 시작하여 파일이 하나 열릴 때마다 1씩 증가하여 저장되었다가 새로운 파일이 열리면 해당 file descriptor number를 부여하고 또 1 증가된다.

(3-1) src/threads/thread.c/struct list\* thread\_get\_fdt(void)

|  |
| --- |
| **struct list \*thread\_get\_fdt(void)**  **{**  **return &thread\_current()->fdt;**  **};** |

현재 실행 중인 스레드의 file descriptor table포인터를 반환한다.

(3-2) src/threads/thread.c/int thread\_get\_next\_fdt(void)

|  |
| --- |
| **int thread\_get\_next\_fd(void)**  **{**  **int next\_fd = thread\_current()->next\_fd;**  **thread\_current()->next\_fd++;**  **return next\_fd;**  **};** |

현재 실행 중인 스레드가 다음 file에 부여할 next file descriptor number를 반환하고 스레드의 next\_fd는 1 증가 시킨다.

(3-3) src/threads/thread.c/static void init\_thread(struct thread \*t, const char \*name, int priority)

|  |
| --- |
| **struct void init\_thread(struct thread \*t, )**  **{**  **︙**  #ifdef USERPROG  t->pcb = NULL;  list\_init(&t->children);  **t->next\_fd = 2;**  **list\_init(&t->fdt);**  #endif  **︙**  **};** |

추가된 thread의 멤버를 초기화하는 코드를 init\_thread에 추가한다. 앞서 언급한 바와 같이, next\_fd는 표준 입출력에 할당된 0과 1을 제외해야 하므로 2로 초기화한다.

(4-1) src/userprog/process.c/void process\_exit(void)

|  |
| --- |
| **void process\_exit(void)**  **{**  struct thread \*cur = thread\_current();  struct process \*pcb = thread\_get\_pcb();  struct list \*children = thread\_get\_children();  struct list\_elem \*e;  uint32\_t \*pd;  **int max\_fd = thread\_get\_next\_fd(), i;**  pcb->is\_exited = true;  for (e = list\_begin(children); e != list\_end(children); e = list\_next(e))  process\_remove\_child(list\_entry(e, struct process , child\_elem));  **for (i=2; i < max\_fd; i++)**  **sys\_close(i);**  sema\_up(&pcb->sema\_exit);  pd = cur->pagedir;  if (pd != NULL)  {  cur->pagedir = NULL;  pagedir\_activate(NULL);  pagedir\_destroy(pd);  }  **}** |

process를 종료할 때 실행되는 process\_exit에 종료되는 해당 스레드가 가지고 있는 파일을 모두 종료하는 코드를 추가한다. max\_fd 변수에 현재 스레드의 next\_fd를 받으면 그것이 곧 열려 있는 파일의 개수가 되기 때문에 그만큼 for문을 돌려 하나하나 sys\_close(뒤에 구현 내용 나옴)으로 종료시킨다.

(4-2) src/userprog/process.c/struct file\_descriptor\_entry \* prcess\_get\_fb(int fd)

|  |
| --- |
| **struct file\_descriptor\_entry \* prcess\_get\_fb(int fd)**  **{**  **struct list \*fdt = thread\_get\_fdt();**  **struct list\_elem \*e;**  **for (e = list\_begin(fdt); e != list\_end(fdt); e = list\_next(e))**  **{**  **struct file\_descriptor\_entry \*fde = list\_entry(e, struct file\_descriptor\_entry, fdt\_elem);**    **if (fde->fd == fd)**  **return fde;**  **}**  **return NULL;**  **}** |

인자로 전달되는 fd와 일치하는 file descriptor number가 있는지, 현재 실행 중인 스레드의 file descriptor table을 탐색한다. 있다면 일치하는 file descriptor entry를, 없다면 NULL을 반환한다.

1. **System Calls**

(1-1) src/userprog/syscall.c/static int sys\_read(int fd, void \*buffer, unsigned size)

|  |
| --- |
| **static int sys\_read(int fd, void \*buffer, unsigned size)**  **{**  **struct file\_descriptor\_entry \*fde;**  **int bytes\_read,** i;  if (!check\_validity\_useraddr(buffer))  sys\_exit(-1);    for (i = 0; \*(char \*)(buffer + i) != NULL; i++)  if (!check\_validity\_useraddr(buffer + i + 1))  sys\_exit(-1);  if (fd == 0)  {  unsigned i;  for (i = 0; i < size; i++)  \*(uint8\_t \*)(buffer + i) = input\_getc();  return size;  }  **fde = process\_get\_fd(fd);**  **if(!fde)**  **return -1;**  **lock\_acquire(&filesys\_lock);**  **bytes\_read = (int)file\_read (fde->file, buffer, (off\_t)size);**  **lock\_release(&filesys\_lock);**  **return bytes\_read;**  **}** |

user program에 의해 system call 중에 read가 호출되면 system call handler가 sys\_read()을 실행시킨다. 기존에 표준 입력에 대해서만 처리하도록 구현했던 함수에 fd가 0이 아닌 경우에 대해 file을 읽을 수 있도록 코드를 추가했다. fd가 0이 아닌 경우, 새롭게 선언한 file\_descriptor\_entry에 현재 실행되는 스레드의 file descriptor table에서 인자로 넘어온 fd와 일치하는 entry를 찾아서 존재하지 않는다면 -1을 반환한다. 존재한다면, src/filesys의 file\_read를 호출하여 size만큼 파일로부터 데이터를 읽어 buffer로 옮긴 후, 실제로 읽은 byte size를 반환한다.

(1-2) src/userprog/syscall.c/static int sys\_write(int fd, void \*buffer, unsigned size)

|  |
| --- |
| **static int sys\_write(int fd, void \*buffer, unsigned size)**  **{**  **struct file\_descriptor\_entry \*fde;**  **int bytes\_written,** i;  if (!check\_validity\_useraddr(buffer))  sys\_exit(-1);    for (i = 0; \*(char \*)(buffer + i) != NULL; i++)  if (!check\_validity\_useraddr(buffer + i + 1))  sys\_exit(-1);  if (fd == 1)  {  putbuf((const char \*)buffer, (size\_t)size);  return size;  }  **fde = process\_get\_fd(fd);**  **if(!fde)**  **return -1;**  **lock\_acquire(&filesys\_lock);**  **bytes\_written= (int)file\_write (fde->file, buffer, (off\_t)size);**  **lock\_release(&filesys\_lock);**  **return bytes\_written;**  **}** |

user program에 의해 system call 중에 write가 호출되면 system call handler가 sys\_write()을 실행시킨다. 기존에 표준 출력에 대해서만 처리하도록 구현했던 함수에 fd가 1이 아닌 경우에 대해 file에 쓸 수 있도록 코드를 추가했다. fd가 1이 아닌 경우, 새롭게 선언한 file\_descriptor\_entry에 현재 실행되는 스레드의 file descriptor table에서 인자로 넘어온 fd와 일치하는 entry를 찾아서 존재하지 않는다면 -1을 반환한다. 존재한다면, src/filesys의 file\_write를 호출하여 size만큼 buffer로부터 데이터를 읽어 파일로 옮긴 후, 실제로 쓴 만큼 byte size를 반환한다.

(1-3) src/userprog/syscall.c/static bool sys\_create(const char \*file, unsigned initial\_size)

|  |
| --- |
| **static bool sys\_create(const char \*file, unsigned initial\_size)**  **{**  **bool ret;**  **int i;**  **if (!check\_validity\_useraddr(file))**  **sys\_exit(-1);**  **for (i = 0; \*(file + i) != NULL; i++)**  **if (!check\_validity\_useraddr(file + i + 1))**  **sys\_exit(-1);**  **lock\_acquire(&filesys\_lock);**  **ret = filesys\_create(file, (off\_t)initial\_size);**  **lock\_release(&filesys\_lock);**  **return ret;**  **}** |

인자로 전달된 file 이름과 크기를 가지는 file을 filesys\_create라는 기존에 src/filesys에 존재하는 함수를 통해 생성한다. sys\_create은 program에 의해 system call 중 create가 호출되면 system call handler가 실행시키는 함수로, user의 요청을 실행하기 전에 virtual address의 validity를 확인하여야 하고, 파일에 대해 접근하기 전에 lock을 얻고 실행 후 lock을 풀어주는 동기화 과정도 필요하다(synchronization 파트에서 더 설명할 것임).

(1-4) src/userprog/syscall.c/static bool sys\_remove(const char \*file)

|  |
| --- |
| **static bool sys\_remove(const char \*file)**  **{**  **bool ret;**  **int i;**  **if (!check\_validity\_useraddr(file))**  **sys\_exit(-1);**  **for (i = 0; \*(file + i) != NULL; i++)**  **if (!check\_validity\_useraddr(file + i + 1))**  **sys\_exit(-1);**  **lock\_acquire(&filesys\_lock);**  **ret = filesys\_remove(file, (off\_t)initial\_size);**  **lock\_release (&filesys\_lock);**  **return ret;**  **}** |

program에 의해 system call 중 remove가 호출되면 system call handler가 실행시키는 함수로, 인자로 전달된 file 이름의 file을 src/filesys의 filesys\_remove함수를 호출하여 제거한다. 이 함수는 user의 요청을 실행하기 전에 virtual address의 validity를 확인하여야 하고, 파일에 대해 접근하기 전에 lock을 얻고 실행 후 lock을 풀어주는 동기화 과정도 필요하다.

(1-5) src/userprog/syscall.c/static bool sys\_open(const char \*file)

|  |
| --- |
| **static bool sys\_open (const char \*file)**  **{**  **struct file\_descriptor\_entry \* fde;**  **struct file \* new\_file;**  **if (!check\_validity\_useraddr(file))**  **sys\_exit(-1);**  **for (i = 0; \*(file + i) != NULL; i++)**  **if (!check\_validity\_useraddr(file + i + 1))**  **sys\_exit(-1);**  **fde = palloc\_get\_page(0);**  **if(!fde)**  **return -1;**  **lock\_acquire(&filesys\_lock);**  **new\_file = filesys\_open(file);**  **if(!new\_file)**  **{**  **palloc\_free\_page(fde);**  **lock\_release(&filesys\_lock);**  **return -1;**  **}**  **fde->fd = thread\_get\_next\_fd();**  **fde->file = new\_file;**  **list\_push\_back(thread\_get\_fdt(), &fde->fdt\_elem);**  **lock\_release(&filesys\_lock);**  **return fde->fd;**  **}** |

program에 의해 system call 중 open이 호출되면 system call handler가 실행시키는 함수로, 인자로 전달된 file 이름의 file을 open한다. src/filesys의 filesys\_open를 호출하여 새로운 파일을 열게 되고, 실패한다면 할당된 페이지와 lock을 해제한 후 -1을 반환한다. 성공한다면 해당 스레드의 다음 fd값과 파일 포인터를 새로 만든 file\_descriptor\_entry에 초기화 시킨 후, 이 entry를 스레드의 file descriptor table에 push back하여 연결 시켜준다. 이 함수는 user의 요청을 실행하기 전에 virtual address의 validity를 확인하여야 하고, 파일에 대해 접근하기 전에 lock을 얻고 실행 후 lock을 풀어주는 동기화 과정도 필요하다.

(1-6) src/userprog/syscall.c/static int sys\_filesize(int fd)

|  |
| --- |
| **static int sys\_filesize(int fd)**  **{**  **struct file\_descriptor\_entry \* fde = process\_get\_fd(fd);**  **int filesize;**  **if(!fde)**  **return -1;**  **lock\_acquire(&filesys\_lock);**  **filesize = file\_length(fde->file);**  **lock\_release (&filesys\_lock);**  **return filesize;**  **}** |

program에 의해 system call 중 filesize가 호출되면 system call handler가 실행시키는 함수로, 인자로 전달된 file descriptor number에 해당하는 파일의 사이즈를 반환하는 함수이다. process\_get\_fd함수를 통해 해당 fd를 가진 file을 file descriptor table에서 찾은 후, 없다면 -1을 반환한다. 존재하는 경우에는 src/filesys의 file\_length함수를 호출하여 해당 파일의 사이즈를 byte단위로 받게 되고 그것을 반환한다. 파일에 대해 접근하기 전에 lock을 얻고 실행 후 lock을 풀어주는 동기화 과정이 필요하다.

(1-7) src/userprog/syscall.c/static void sys\_seek(int fd, unsigned position)

|  |
| --- |
| **static void sys\_seek(int fd, unsigned position)**  **{**  **struct file\_descriptor\_entry \* fde = process\_get\_fd(fd);**  **if(!fde)**  **return ;**  **lock\_acquire(&filesys\_lock);**  **file\_seek(fde->file, (off\_t)position);**  **lock\_release (&filesys\_lock);**  **}** |

program에 의해 system call 중 seek가 호출되면 system call handler가 실행시키는 함수로, 인자로 전달된 file descriptor number에 해당하는 파일의 table entry를 탐색하고 없다면 함수를 종료 시킨다. 해당 fd와 일치하는 entry를 찾은 경우, src/filesys의 file\_seek함수를 호출하여 읽거나 쓸 위치의 다음 byte를 position의 값으로 바꾼다. 파일에 대해 접근하기 전에 lock을 얻고 실행 후 lock을 풀어주는 동기화 과정이 필요하다.

(1-8) src/userprog/syscall.c/static unsigned sys\_tell(int fd)

|  |
| --- |
| **static unsigned sys\_tell(int fd)**  **{**  **struct file\_descriptor\_entry \* fde = process\_get\_fd(fd);**  **unsigned pos;**  **if(!fde)**  **return -1;**  **lock\_acquire(&filesys\_lock);**  **pos =(unsigned)file\_tell(fde->file);**  **lock\_release (&filesys\_lock);**  **return pos;**  **}** |

program에 의해 system call 중 tell이 호출되면 system call handler가 실행시키는 함수로, 인자로 전달된 file descriptor number에 해당하는 파일의 table entry를 탐색하고 없다면 -1을 반환한다. 해당 fd와 일치하는 entry를 찾은 경우, src/filesys의 file\_tell함수를 호출하여 읽거나 쓸 위치의 다음 byte의 position을 받아 반환한다. 파일에 대해 접근하기 전에 lock을 얻고 실행 후 lock을 풀어주는 동기화 과정이 필요하다.

(1-9) src/userprog/syscall.c/void sys\_close(int fd)

|  |
| --- |
| **void sys\_close(int fd)**  **{**  **struct file\_descriptor\_entry \* fde = process\_get\_fd(fd);**  **if(!fde)**  **return ;**  **lock\_acquire(&filesys\_lock);**  **file\_close(fde->file);**  **list\_remove(&fde->fdtelem);**  **palloc\_free\_page(fde);**  **lock\_release (&filesys\_lock);**  **}** |

program에 의해 system call 중 close가 호출되면 system call handler가 실행시키는 함수이다. 인자로 전달된 file descriptor number와 일치하는 table entry를 탐색하여 존재하지 않는다면 함수를 종료 시킨다. 해당 fd와 일치하는 entry를 찾은 경우, src/filesys의 file\_close함수를 호출하여 해당 file을 닫고, 그 file에 대한 entry를 file descriptor table에서 없애고 할당된 페이지도 해제한다. 파일에 대해 접근하기 전에 lock을 얻고 실행 후 lock을 풀어주는 동기화 과정이 필요하다

(1-10) src/userprog/syscall.c/static void syscall\_handler(struct intr\_frame \*f)

|  |
| --- |
| **static void syscall\_handler(struct intr\_frame \*f)**  **{**  void \*esp = f->esp;  if (!check\_validity\_useraddr(esp) || !check\_validity\_useraddr(esp + sizeof(uintpGtr\_t) - 1))  sys\_exit(-1);  int sys\_num = \*(int \*)esp;  **switch (sys\_num)**  **{**  **︙**  **case SYS\_CREATE:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 3­ \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **char \*file = \*(char \*\*)(esp + sizeof(uintptr\_t));**  **unsigned initial\_size = \*(unsigned \*)(esp + 2 \* sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_create(file, initial\_size);**  **break;**  **}**  **case SYS\_REMOVE:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 2­ \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **char \*file = \*(char \*\*)(esp + sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_remove(file);**  **break;**  **}**  **case SYS\_OPEN:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **char \*file = \*(char \*\*)(esp + sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_open(file);**  **break;**  **}**  **case SYS\_FILESIZE:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int fd = \*(int \*)(esp + sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_filesize(fd);**  **break;**  **}**  **case SYS\_SEEK:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 3 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int fd = \*(int \*)(esp + sizeof(uintptr\_t));**  **unsigned position = \*(unsigned \*)(esp + 2 \* sizeof(uintptr\_t));**  **sys\_seek(fd, position);**  **break;**  **}**  **case SYS\_TELL:**  **{**  **if (!check\_validity\_useraddr(esp + sizeof(uintptr\_t)) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int fd = \*(int \*)(esp + sizeof(uintptr\_t));**  **f->eax = (uint32\_t)sys\_tell(fd);**  **break;**  **}**  **case SYS\_CLOSE:**  **{**  **if (!check\_validity\_useraddr(esp) || !check\_validity\_useraddr(esp + 2 \* sizeof(uintptr\_t) - 1))**  **sys\_exit(-1);**  **int fd = \*(int \*)(esp + sizeof(uintptr\_t));**  **sys\_close(fd);**  **break;**  **}**  **default:**  **sys\_exit(-1);**  **}**  **}** |

user program에 의해 system call이 호출되면 system call handler는 적절한 system call 번호를 스택에서 pop하여 해당 switch 문을 실행한다. user address를 기준으로 스택에서 필요한 esp에서 pop하는 모든 경우에 대해 check\_validity를 수행하여 invalid한 경우 system call을 호출한 user program을 종료 시킨다. 프로젝트1에서 이미 구현한 case외에 새로 추가된 case들만 적절한 함수를 호출하여 system call기능을 수행하도록 작성하였다.

1. **Synchronization in File System**

(1) userprog/syscall.c/static struct lock filesys\_lock

|  |
| --- |
| **static struct lock filesys\_lock;** |

커널이 시스템 콜 호출이 되어 파일 시스템에 접근할 때, 파일이라는 critical 영역을 한번에 하나의 프로세스만이 접근하여 사용할 수 있도록 동기화하는 구조체이다. 이는 file을 열고 쓰고 종료하는 등의 시스템 콜 호출이나, 프로세스가 초기에 로딩 되고 종료될 때의 executable file에 대해서도 적용된다.

|  |
| --- |
| **struct thread**  **{**  **︙**  #ifdef USERPROG  uint32\_t \* pagedir; /\* Page directory. \*/  struct process \*pcb; /\* Process Control Block. \*/  struct list children; /\* List of children process. \*/  **int next\_fdt;** /\* File descriptor number for next new file. \*/  **struct list \*fdt**; /\* File descriptor table. \*/  **struct file \* running\_file**; /\* Currently running file. \*/  #endif  **︙**  **};** |

(2) threads/thread.h/struct thread

현재 프로세스가 실행 중인 executable file에 대한 정보 field를 추가한다.

(3-1) threads/thread.c/void thread\_set\_running\_file(struct file \*new\_running\_file)

|  |
| --- |
| **void thread\_set\_running\_file(struct file \*new\_running\_file)**  **{**  **thread\_current()->running\_file = new\_running\_file;**  **}** |

현재 실행 중인 스레드에 대해 추가된 thread의 field값을 set할 수 있는 함수를 구현한다.

(3-2) threads/thread.c/struct file \* thread\_get\_running\_file (void)

|  |
| --- |
| **struct file \* thread\_get\_running\_file (void)**  **{**  **return thread\_current()-> running\_file;**  **}** |

현재 실행 중인 스레드에 대해 추가된 thread의 field값을 get할 수 있는 함수를 구현한다.

(4-1) userprog/process.c/void process\_exit(void)

|  |
| --- |
| **void process\_exit(void)**  **{**  **︙**  **lock\_acquire(filesys\_lock);**  **file\_close(thread\_get\_running\_file());**  **lock\_release(filesys\_lock);**  **︙**  **}** |

현재 스레드가 종료될 때 running file에 저장된 executable file이 닫혀야 하므로, lock을 통해 다른 프로세스의 접근을 막은 후 해당 파일을 종료한다.

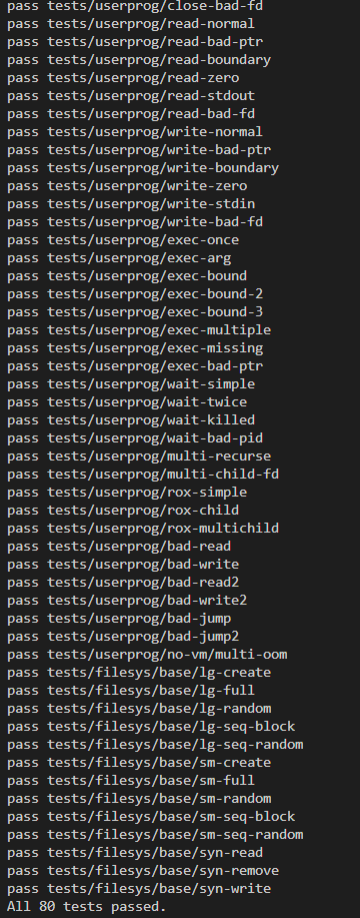
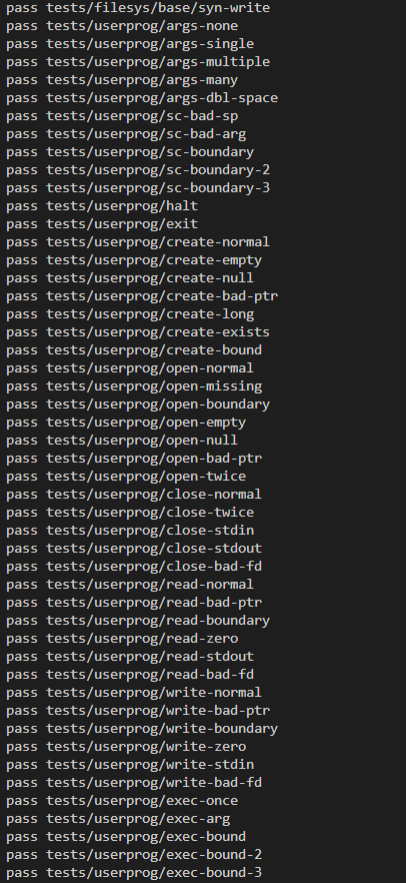
(4-2) userprog/process.c/static bool load(const char \* file\_name, void (\*\*eip)(void), void \*\*esp)

|  |
| --- |
| **static bool load(const char \* file\_name, void (\*\*eip)(void), void \*\*esp)**  **{**  **︙**  **lock\_acquire(filesys\_lock);**  **file = filesys\_open(argv[0]);**  **if(file==NULL)**  **{**  **printf(“load: %s: open failed\n”, file\_name);**  **goto done;**  **}**  **thread\_set\_running\_file(file);**  **file\_deny\_write(file);**  **︙**  **done:**  **lock\_release(filesys\_lock);**  **return success;**  **}** |

인자로 전달된 파일이름을 parsing하여 실제 파일 이름을 분리한 후 그것을 바탕으로 filesys\_open을 통해 파일을 열고, 성공했다면 이 실행파일을 thread\_set\_running\_file을 통해 현재 스레드의 running file 필드에 set한다. 또한 file\_deny\_write을 통해 실행이 종료될 때 까지 이 파일로의 write을 금지 시킨다. 그리고 모든 로딩 과정이 끝나고 done에 다다르면, file open에 사용했던 lock을 해제 한다.

1. **시험 및 평가 내용**

User program을 실행하는 데 필요한 모든 핀토스의 기능을 구현하였으며, userprog에서 make check을 한 결과 모든 테스트가 통과하는 것을 확인할 수 있었다.



(76개만 통과해도 되는지 확인 제대로 못하고 80개를 모두 통과하게 만들었습니다..)