Pintos Project: 2. User Program

Design Report – team 30

20210054 정하우

20210716 최대현

Context

1. **Introduce**
    - **1-1. Analysis on process execution procedure**
    - **1-2. Analysis on system call procedure**
    - **1-3. Analysis on file system**

2. **Analysis of the current implementation**
    - **2-1. Process termination messages**
    - **2-2. Argument Parsing**
    - **2-3. System Call**
    - **2-4. Denying Writes to Executables**

3. **Design Plan**
    - **3-1. Process termination messages**
    - **3-2. Argument Parsing**
    - **3-3. System Call**
    - **3-4. Denying Writes to Executables**

1. Introduce & Structure

1-1. Analysis on process execution procedure

- 전반적인 process execution의 과정을 분석해본다.
- 우선 Pintos의 main program은 init.c/main()를 통해 시작된다. 해당 함수에서 우리가 입력한 명령어를 parsing하여 이를 argument와 option으로 나누고, 넣어준 argv 값에 대해 run_action 함수를 호출하고 있다.

```
int
main (void)
{
  char **argv;

  /* Clear BSS. */
  bss_init ();

  /* Break command line into arguments and parse options. */
  argv = read_command_line ();
  argv = parse_options (argv);

...
  printf ("Boot complete.\n");

  /* Run actions specified on kernel command line. */
  run_actions (argv);

  /* Finish up. */
  shutdown ();
  thread_exit ();
}
```

- Pintos의 main program이다. read_command_line() 함수로 command line에 입력한 전체 인자를 받아오고, 이를 parse_options()라는 함수로 argument와 option으로 나눠준 뒤 arguments를 argv에 저장한다. 그 뒤 kernel command line에서 run_actions 함수에 argv 인자를 넘겨 호출한다.
-

```
static void

run_actions (char **argv)
{
  /* An action. */
  struct action
```

```
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, including action name. */
      void (*function) (char **argv);   /* Function to execute action. */
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"extract", 1, fsutil_extract},
      {"append", 2, fsutil_append},
#endif
      {NULL, 0, NULL},
    };

...
}
```

- run_actions() 함수는 argv라는 변수명을 가진 argument를 넘겨받아 action을 취하는 함수이다. 이 안에는 action이라는 구조체를 정의하여 우리가 취할 action에 대한 정보를 struct 형태로 저장한다. Action의 name인 char *name, args의 수인 argc, execute action을 취할 함수의 이름인 void *function char **argv를 멤버 변수로 갖고 있다. 예를 들어 run User program이 수행되면 argc는 2가 되고 run_task를 호출한다.

```
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

- run_task 함수는 task를 수행하는 함수이다. run_task에서는 run을 통해 argv라는 인자가 넘어온 상태이고, 이의 1번 index (string일 것이다.)에 접근하여 task의 이름을 parsing한다. 그 뒤 이 task에 대해서 후에 구현할 process_execute 함수와 process_wait 함수를 호출한다. process_wait 함수의 경우 다른 부분의 구현에서 자세히 다룰 예정이므로 현재 설명은 생략하고, process_execute 함수의 호출로 넘어간다.

```
tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;

  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

- process_wait 함수는 현재는 구현되어 있지 않지만, 본래의 기능은 child process가 종료될 때까지 대기하는 함수이다.
- process_execute 함수는 process execution을 수행하는 함수로, 이 함수에서는 char * str을 argument로 받는다. 예를 들어, process_execute("echo")와 같이 입력 argument를 'echo'라는 문자열로 넣어주면, 이 안에서 thread_create 함수를 호출하여 thread를 만들어내고 함수의 return 값을 변수 tid에 할당해준다. 그 뒤, tid를 return한다.
- 현재의 구현으로서 process_execute 함수에서는 file_name을 name argument로 넘기고 start_process 함수를 인자로 넘겨서 thread를 create한다.

```
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;

  /* Initialize interrupt frame and load executable. */
  ...
  success = load (file_name, &if_.eip, &if_.esp);

  /* If load failed, quit. */
  palloc_free_page (file_name);
  if (!success)
    thread_exit ();

  /* 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 ();
}
```

- start_process 함수는 process를 시작하는 함수이다. 눈여겨볼 부분은 process의 start가 문제없이 진행됐는지를 판단하는 Boolean type 변수 success를 선언하고 이를 load()함수의 반환값을 대입해준다는 점이다. 이후 success가 false라면, 즉 process start를 실패했다면 thread_exit() 함수를 실행한다.
- 아직은 load 함수가 무엇인지 모르니 load 함수를 분석해본다.

```
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 ();

  /* Open executable file. */
  file = filesys_open (file_name);
  if (file == NULL)
    {
      printf ("load: %s: open failed\n", file_name);
      goto done;
    }

...
  /* Set up stack. */
  if (!setup_stack (esp))
    goto done;

  /* 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;
}
```

- load() 함수는 pagedir_create() 함수를 호출하여 start할 user process의 page table을 생성하고, filesys_open이라는 method를 실행하여 excute하려는 program의 이름으로 executable한 파일을 open한다. 그 뒤 ELF header의 정보를 읽어오고 stack pointer인 esp와 eip를 setup_stack 함수와 ehdr.e_entry 값 대입을 통해 세팅한다. 마지막으로 file을 close하고 success 변수를 true로 설정한 뒤 반환한다.
- 이로써 전반적인 thread execution 과정을 모두 분석해보았다.

1-2. Analysis on System Call

- 전반적인 System Call의 procedure을 분석해본다.
- Pintos Project2에서 다뤄야할 System Call은 다음과 같다.

```
/* System call numbers. */

enum
  {
    /* Projects 2 and later. */
    SYS_HALT,                   /* Halt the operating system. */
    SYS_EXIT,                   /* Terminate this process. */
    SYS_EXEC,                   /* Start another process. */
    SYS_WAIT,                   /* Wait for a child process to die. */
    SYS_CREATE,                 /* Create a file. */
    SYS_REMOVE,                 /* Delete a file. */
    SYS_OPEN,                   /* Open a file. */
    SYS_FILESIZE,               /* Obtain a file's size. */
    SYS_READ,                   /* Read from a file. */
    SYS_WRITE,                  /* Write to a file. */
    SYS_SEEK,                   /* Change position in a file. */
    SYS_TELL,                   /* Report current position in a file. */
    SYS_CLOSE,                  /* Close a file. */
        …
  };
```

- system call number가 enumeration 형태로 선언되어 있고, syscall_handler함수가 위 case에 해당하는 system call을 호출하도록 구현해야 한다.
- syscall_handler 함수는 user program이 system call을 할 때 아래와 같은 적절한 어셈블리 코드 매크로가 작동하면서 실행된다.

```
 /* Invokes syscall NUMBER, passing argument ARG0, and returns the
    return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                           \
        ({                                                               \
          int retval;                                                    \
          asm volatile                                                   \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
               : "=a" (retval)                                           \
               : [number] "i" (NUMBER),                                  \
                 [arg0] "g" (ARG0)                                       \
               : "memory");                                              \
          retval;                                                        \
        })
```

- 위 코드는 argument 개수가 1개일 때 불리는 syscall1 매크로이다. Argument 개수에 따라 다른 매크로가 불린다. 그리고 0x30은 kernel space의 interrupt vector table에서 syscall_handler 함수의 주소를 의미하므로, interrupt 시 syscall_handler 함수가 호출된다고 해석이 가능하다. 이제 우리는 기능이 구현되어 있지 않은 syscall_handler 함수를 구현해야 한다.

-

1-3. Analysis on file system

- 위에서 설명했듯이, 총 13가지의 각기 다른 system call을 처리해야 한다. 그중 user process를 다루는 halt, exit, exec, wait 4개를 제외한 나머지 9가지 경우는 file system이 필요한 경우이다.
- 다음은 pintos에 구현되어 있는 file system 함수이다.

bool filesys_create (const char *name, off_t initial_size)

```
/* Creates a file named NAME with the given INITIAL_SIZE.
   Returns true if successful, false otherwise.
   Fails if a file named NAME already exists,
   or if internal memory allocation fails. */
bool
filesys_create (const char *name, off_t initial_size)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = dir_open_root ();
  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
                  && inode_create (inode_sector, initial_size)
                  && dir_add (dir, name, inode_sector));
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);

  return success;
}
```

struct file * filesys_open (const char *name)

```
/* Opens the file with the given NAME.
   Returns the new file if successful or a null pointer
   otherwise.
   Fails if no file named NAME exists,
```

```
   or if an internal memory allocation fails. */
struct file *
filesys_open (const char *name)
{
  struct dir *dir = dir_open_root ();
  struct inode *inode = NULL;

  if (dir != NULL)
    dir_lookup (dir, name, &inode);
  dir_close (dir);

  return file_open (inode);
}
```

bool filesys_remove (const char *name)

```
/* Deletes the file named NAME.
   Returns true if successful, false on failure.
   Fails if no file named NAME exists,
   or if an internal memory allocation fails. */
bool
filesys_remove (const char *name)
{
  struct dir *dir = dir_open_root ();
  bool success = dir != NULL && dir_remove (dir, name);
  dir_close (dir);

  return success;
}
```

- 이번 project에서는 file system을 수정할 필요는 없이, file descriptior(fd)라는 개념을 통해 thread가 file을 접근하는 방식을 구현해야 한다.
- File Descriptor (fd)는 스레드가 파일에 접근하고 제어하는 데 사용하는 추상적인 값이다. 스레드가 파일을 열면, 커널은 사용 가능한 가장 작은 fd 값을 할당한다. File Descriptor를 통해 스레드는 여러 파일을 추상적으로 관리하고 동시에 여러 파일에 접근할 수 있다.
- Pintos는 아직 위 사항에 대해 구현이 안 되어 있어 halt, exit, exec, wait 4개를 제외한 나머지 9가지 경우를 fd를 통해 구현해야 한다.

2. Analysis of the current implementation

2-1. Process termination messages

Problem Description

- 현재는 출력되고 있지 않는 process termination message 출력을 구현해야 한다. User Program이 종료되었을 때 종료된 process의 name과 어떤 system call에 의해 종료됐는지에 대한 정보인 exit code를 다음과 같은 형식으로 출력해야 한다.
- **printf("%s: exit(%d₩n)",variable_1, variable_2)**
- %s에 해당하는 variable_1은 process name(char *)이 들어가게 될 것이고, variable 2는 exit code이다. 위 형식에서 variable_1은 Process의 Name이고, variable_2는 exit code(int)이다.
-

관련 함수 설명

- 현재는 위의 정보를 출력하는 기능이 구현되어 있지 않으나, 관련하여 system call의 여러 형태 중 exit하는 과정에서 출력하는 함수를 구현하면 될 것으로 판단된다.
- 다른 함수에서 종료된 process의 name을 받아오고, exit code는 곧 status이기 때문에 이를 그대로 variable1, variable2에 넣어준 뒤 출력하는 식으로 구현하면 된다.
- process의 name을 받아오는 방법을 고려하던 중, process_execute()함수를 호출할 때 argument에 file_name을 인자로 넣어준다는 것을 파악했다. 또한, 미리 정의된 thread structure에는 char * type의 name이 있다. 따라서 해당 함수와 구조체를 들여다보았다.

- **1.src/userprog/process.c/tid_t process_execute (const char *file_name)**

```
tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;

  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);
```

```
  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

- Process를 execute하는 과정을 수행하는 함수이다. 여기에는 메모리 할당, thread create 등이 해당될 수 있다.
- Char * type, 즉 C-style string인 file_name 변수를 argument로 받는다. 그 뒤 이 이름을 가지고 memory를 allocate하고 (user program에서 다루지 않음.) 이를 thread_create함수의 'name' argument로 가져가면서 넣어준다. 즉, process termination message를 원활하게 출력하기 위해서는 process의 name을 얻어야 하고 이 process의 name을 얻으려면 thread_create로 만들어준 thread에서 name을 얻어오는 별도의 함수를 구현하면 해결할 수 있을 것이다.

- **2. src/userprog/thread.h/struct thread**

```
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.
*/

...
  };
```

-
- 길이가 16인 char 정적 배열 name을 member 변수로 갖는다는 것을 알 수 있다. 즉 struct thread type으로 정의된 각 thread들의 멤버 변수로서 접근이 가능하다.
- 추가적으로, Pintos document에 따르면 현재로써는 setup_stack 함수를 실행했을 때 Page fault가 발생한다. *esp = PHYS_BASE을 *esp = PHYS_BASE-12로 수정해야 한다.

2-2. Argument Passing

Problem Description

- 일반적으로 shell에 명령어를 입력하는 경우를 생각해보았을 때, shell은 공백 단위로 우리의 (단어 sequence 형태로 전달된) 명령어를 프로그램과 인자로 분류하여 그에 맞는 program을 수행해준다. 그러나, 현재의 Pintos는 프로그램과 인자를 구분하지 못한다. 예를 들어, 'ls -a'라는 argument를 command line에 입력하면 현재의 Pintos는 'ls -a'를 하나의 프로그램 이름으로 인식하고 있다. 따라서 해당 구현해서는 input으로 들어가는 argument를 프로그램 이름과 인자로 구분하여 Pintos의 stack에 저장하고, argument를 프로그램에 올바르게 전달하여 의도에 맞는 process execution을 할 수 있도록 하는 것을 목표로 한다.

관련 함수 설명

- **1. init.c/int main(void)**

```
int
main (void)
{
  char **argv;

  /* Clear BSS. */
  bss_init ();

  /* Break command line into arguments and parse options. */
  argv = read_command_line ();
  argv = parse_options (argv);

...
  printf ("Boot complete.\n");

  /* Run actions specified on kernel command line. */
  run_actions (argv);

  /* Finish up. */
  shutdown ();
  thread_exit ();
}
```

  - Pintos의 main program이다. read_command_line() 함수로 command line에 입력한 전체 인자를 받아오고, 이를 parse_options()라는 함수로 argument와 option으로 나

줘준 뒤 arguments를 argv에 저장한다. 그 뒤 kernel command line에서 run_actions 함수에 argv 인자를 넘겨 호출한다.

- **2. init.c/static void run_actions (char **argv)**

```
 static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, including action name. */
      void (*function) (char **argv);   /* Function to execute action. */
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"extract", 1, fsutil_extract},
      {"append", 2, fsutil_append},
#endif
      {NULL, 0, NULL},
    };

...
}
```

- run_actions() 함수는 argv라는 변수명을 가진 argument를 넘겨받아 action을 취하는 함수이다. 이 안에는 action이라는 구조체를 정의하여 우리가 취할 action에 대한 정보를 struct 형태로 저장한다. Action의 name인 char *name, args의 수인 argc, execute action을 취할 함수의 이름인 void *function char **argv를 멤버 변수로 갖고 있다. 예를 들어 run User program이 수행되면 argc는 2가 되고 run_task를 호출한다.

- **3. init.c/static void run_task (char **argv)**

```
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

- task를 수행하는 함수이다. run_task에서는 run을 통해 argv라는 인자가 넘어온 상태이고, 이의 1번 index (string일 것이다.)에 접근하여 task의 이름을 parsing한다. 그 뒤 이 task에 대해서 후에 구현할 process_execute 함수와 process_wait 함수를 호출한다. process_wait 함수의 경우 다른 부분의 구현에서 자세히 다룰 예정이므로 현재 설명은 생략한다.

- **4. tid_t process_execute (const char *file_name)**

```
tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;

  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

- **Analysis on process execution procedure**

- Process를 execute하는 과정을 수행하는 함수이다. 여기에는 메모리 할당, thread create, tid 반환 등이 해당될 수 있다.

- argument passing을 구현하는 데 있어 현재의 가장 큰 문제점은, input으로 들어오는 file_name을 그대로 thread_create 함수의 "name" argument로 넘겨준다는 것이다. thread_create() 함수를 잠시 들여다보면 함수 내에서 init_thread 함수를 name argument와 함께 호출한다. 즉, name argument 그대로의 이름을 가진 thread를 만들게 되는데, 예를 들어 command line에 명령어를 'pintos -q run alarm-single'라고 입력한다면 현재의 방식으로는 'pintos -q run alarm-single'라는 이름을 가진 thread와 이를 갖고 있는 process가 만들어질 것이라는 것이다. 이것 때문에 현재의 Pintos system에서는 명령어를 제대로 인식하지 못한다. 따라서 input argument로 들어오는 file_name을 공백 단위로 구분하여 thread_create 함수에 적절히 넣어주는 작업이 필요하다.

- **5.process.c/static void start_process (void *file_name_)**

```
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;

  /* Initialize interrupt frame and load executable. */
  ...
  success = load (file_name, &if_.eip, &if_.esp);

  /* If load failed, quit. */
  palloc_free_page (file_name);
  if (!success)
    thread_exit ();

  /* 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 ();
}
```

- Process를 시작하는 함수이다. 눈여겨볼 부분은 process의 start가 문제없이 진행됐는지를 판단하는 Boolean type 변수 success를 선언하고 이를 load()함수의 반환값을 대입해준다는 점이다. 이후 success가 false라면, 즉 process start를 실패했다면 thread_exit() 함수를 실행한다.

- **5. process.c/ load (const char *file_name, void (**eip) (void), void **esp)**

```
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 ();

  /* Open executable file. */
  file = filesys_open (file_name);
  if (file == NULL)
    {
      printf ("load: %s: open failed\n", file_name);
      goto done;
    }

...
  /* Set up stack. */
  if (!setup_stack (esp))
    goto done;

  /* 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;
}
```

- pagedir_create() 함수를 호출하여 start할 user process의 page table을 생성하고, filesys_open이라는 method를 실행하여 excute하려는 program의 이름으로 executable한 파일을 open한다. 그 뒤 ELF header의 정보를 읽어오고 stack pointer인 esp와 eip를 setup_stack 함수와 ehdr.e_entry 값 대입을 통해 세팅한다. 마지막으로 file을 close하고 success 변수를 true로 설정한 뒤 반환한다.

- esp는 stack의 가장 아래 (마지막)을 가리키는 pointer이며, setup_stack에서 PHYS_BASE로 정의된 기본 값으로 설정된다. 이 stack에 argument를 순서대로 넣어서 후에 각 함수들이 이 값들을 불러와서 필요한 인자를 사용하게 구현할 수 있을 것이다. argument를 넣는 rule은 pintos document와 과제 material을 참고하여 다음과 같이 구현할 예정이다.

Address	Name	Data	Type	
0xbffffffc	argv[3][...]	bar\0	char[4]	
0xbffffff8	argv[2][...]	foo\0	char[4]	
0xbffffff5	argv[1][...]	-l\0	char[3]	
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]	
0xbfffffec	word-align	0	uint8_t	
0xbfffffe8	argv[4]	0	char *	bffffffc0 bffffffd0 bffffffe0 bfffffff0
0xbfffffe4	argv[3]	0xbffffffc	char *	
0xbfffffe0	argv[2]	0xbffffff8	char *	
0xbfffffdc	argv[1]	0xbffffff5	char *	
0xbfffffd8	argv[0]	0xbfffffed	char *	
0xbfffffd4	argv	0xbfffffd8	char **	
0xbfffffd0	argc	4	int	
0xbfffffcc	return address	0	void (*) ()	

- 이를 위해 argument_stack() 함수를 추가 구현할 예정이다.

2-3. System Call

Problem Description

- System call analysis에서 설명했듯이 구현되어 있지 않은 handler 함수를 구현해야 한다. 이를 위해서 위 매크로에서 user stack에 push한 argument를 다시 pop하여 정보를 가져오는 함수 등을 추가적으로 구현해야 한다.

관련 함수 설명

- **1. src/userprog/syscall.c/void syscall_init(void)**

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}
```

- 0x30을 argument vec_no로, intr_register_int 함수를 호출한다.

- **2. src/threads/interrupts.c/void intr_register_int (uint8_t vec_no, int dpl, enum intr_level level, intr_handler_func *handler, const char *name)**

```
 void
intr_register_int (uint8_t vec_no, int dpl, enum intr_level level,
                   intr_handler_func *handler, const char *name)
{
  ASSERT (vec_no < 0x20 || vec_no > 0x2f);
  register_handler (vec_no, dpl, level, handler, name);
}
```

- 0x30을 vec_no argument로 register_handler 함수로 넘겨준다.

- **3. src/threads/interrupts.c/static void register_handler (uint8_t vec_no, int dpl, enum intr_level level, intr_handler_func *handler, const char *name)**

```
 static void
register_handler (uint8_t vec_no, int dpl, enum intr_level level,
                  intr_handler_func *handler, const char *name)
{
```

```
  ASSERT (intr_handlers[vec_no] == NULL);
  if (level == INTR_ON)
    idt[vec_no] = make_trap_gate (intr_stubs[vec_no], dpl);
  else
    idt[vec_no] = make_intr_gate (intr_stubs[vec_no], dpl);
  intr_handlers[vec_no] = handler;
  intr_names[vec_no] = name;
}
```

- intr_handlers 배열에 vec_no에 해당하는 exception handler을 초기화한다. 본 과정에서는 system call에 대한 handling을 다루는 syscall_handler 를 intr_handler 배열에서 index가 vec_no, 즉 0x30에 systemcall handler를 넣어준다. 이 과정은 intr_handler를 초기화하는 과정으로 각 상황별로 어떤 handler를 호출해야할지 정하는 과정이다. 본 내용에서는 system call에 관한 내용이므로 아래에 있는 13가지 case를 다루는 메크로에서 system call을 처리하는 syscall_handler가 불릴 것이다

- **4. src/lib/user/syscall.c/syscall0, syscall1, syscall2, syscall3**
- 위 analysis에서 설명한 매크로. syscall_handler 함수가 호출되며, argument 개수에 따라 구현되어 있다.

- **5. src/lib/user/syscall.c/void halt (void), void exit (int status), pid_t exec (const char *file), int wait (pid_t pid), bool create (const char *file, unsigned initial_size), bool remove (const char *file), int open (const char *file), int filesize (int fd), int read (int fd, void *buffer, unsigned size), int write (int fd, const void *buffer, unsigned size), void seek (int fd, unsigned position), unsigned tell (int fd), void close (int fd)**

- 위에서 말했듯, 우리는 13가지 case에 대한 system call을 handler 처리 해야한다. 각 case마다 handler에 전달해야 하는 argument의 개수가 다르기 때문에 argument 개수에 따른 매크로를 이용해 user stack에 push하고, handler에서 pop해 argument를 다시 가져오는 과정을 통해 argument를 handler에 전달한다.

2-4. Denying Writes to Executables

Problem Description

- OS에서는 현재 execute 중인 user program의 file data가 변경되지 않도록 막아야 한다. 만약 execute 중인 user program의 file data가 변경된다면 program이 원래 예상했던 데이터와 다르게 데이터를 읽어올 가능성이 있기 때문이다. 이런 현상이 발생하면 running program이 올바른 연산 결과를 도출할 수 없기 때문에, OS 차원에서 실행 중인 user program의 write를 막아야 하고 이를 적절히 구현해야 한다.
- 이를 위해 분석해야 하는 관련 함수를 설명한다.

File Structure & Function

(Analysis on file system)

- **1. file.c/struct file**

```
/* An open file. */
struct file
  {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;            /* Has file_deny_write() been called? */
  };
```

- File 구조체의 구조를 파악해보면, 총 3개의 멤버 변수를 가진다. 첫 번째로 inode를 가지는데, inode는 file system에 서 file의 정보를 저장하는 data structure이다. 두 번째는 pos로, 해당 file에서 read 또는 write하는 cursor의 current position을 의미하는 변수이다. 마지막으로 boolean type의 deny_write라는 변수를 가지는데, 이 변수는 file_deny_write() 함수가 call되었는지를 나타내는 변수이다. 즉, 바꿔 말하면, 현재 이 파일에 write가 가능한지 불가능한지를 나타내는 변수이다. 이 값이 true라면 write가 deny되어 불가능한 상태이고, false라면 write가 가능한 상황이다.

- **2. file.c/struct inode**

```
/* In-memory inode. */
struct inode
  {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
```

```
    bool removed;                       /* True if deleted, false otherwise. */
    int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;             /* Inode content. */
  };
```

- inode는 file의 정보를 저장하는 구조체로, inode list의 element, sector number, opener의 개수, removed되었는지를 표시하는 removed, write가 deny되었는지를 표시하는 deny_write_cnt, 마지막으로 inode의 content를 표현하는 inode_disk struct datatype의 data를 멤버 변수로 가지고 있다.

- **3. file.c/struct inode_disk**

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
  };
```

- inode의 data를 저장하는 구조체이다. Inode_disk에는 저장한 data의 first data sector index 값을 담는 start와, file의 크기를 byte 단위로 저장한 length를 저장하고 있다.

Functions about file system

- 현재로서는 file system 내의 함수를 수정하여 구현할 필요는 없지만, system call을 구현할 때 관련 함수들을 이용하여 구현하고, file system을 계속해서 다뤄야 하므로 file system과 관련한 함수를 분석해보는 것으로 한다.

- **1. filesys.c/void filesys_init (bool format)**

```
void
filesys_init (bool format)
{
  fs_device = block_get_role (BLOCK_FILESYS);
  if (fs_device == NULL)
```

```
    PANIC ("No file system device found, can't initialize file system.");

  inode_init ();
  free_map_init ();

  if (format)
    do_format ();

  free_map_open ();
}
```

- file system module을 initialize하는 함수이다. File system device를 찾고, inode와 free map을 initialize하고, free map을 open한다.

- **2. filesys.c/bool filesys_create (const char *name, off_t initial_size)**

```
bool
filesys_create (const char *name, off_t initial_size)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = dir_open_root ();
  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
                  && inode_create (inode_sector, initial_size)
                  && dir_add (dir, name, inode_sector));
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);

  return success;
}
```

- 파일 이름 'name'과 파일 사이즈를 담은 off_t 변수 'initial_size'를 인자 값으로 받아 파일을 생성하는 함수이다.

- **3. filesys.c/struct file * filesys_open (const char *name)**

```
. /* Opens the file with the given NAME.

   Returns the new file if successful or a null pointer
```

```
   otherwise.
   Fails if no file named NAME exists,
   or if an internal memory allocation fails. */
struct file *
filesys_open (const char *name)
{
  struct dir *dir = dir_open_root ();
  struct inode *inode = NULL;

  if (dir != NULL)
    dir_lookup (dir, name, &inode);
  dir_close (dir);

  return file_open (inode);
}
```

- 파일의 이름을 담는 string 변수 'name'을 argument로 받아 해당 이름을 가진 함수를 open하는 함수이다.

- **4. filesys.c/bool filesys_remove (const char *name)**

```
/* Deletes the file named NAME.
   Returns true if successful, false on failure.
   Fails if no file named NAME exists,
   or if an internal memory allocation fails. */
bool
filesys_remove (const char *name)
{
  struct dir *dir = dir_open_root ();
  bool success = dir != NULL && dir_remove (dir, name);
  dir_close (dir);

  return success;
}
```

- File name을 담는 string 변수 'name'을 argument로 받아 해당 이름을 가진 file을 delete하는 함수이다. directory를 열고, directory와 file name을 연결해서 dir_close 함수를 호출하여 해당 파일을 닫는다. 그리고 해당 함수 return 값을 가져와서 반환해준다.

- **Struct file**

File Structure & Function

1. **bool load (const char *file_name, void (**eip) (void), void **esp)**

- file을 open하고 stack을 세팅해주는 함수이다.
- load 함수는 pagedir_create() 함수를 호출하여 start할 user process의 page table을 생성하고, filesys_open이라는 method를 실행하여 excute하려는 program의 이름으로 executable한 파일을 open한다. 그 뒤 ELF header의 정보를 읽어오고 stack pointer인 esp와 eip를 setup_stack 함수와 ehdr.e_entry 값 대입을 통해 세팅한다. 마지막으로 file을 close하고 success 변수를 true로 설정한 뒤 반환한다.
- load 함수에서 file을 open하는데, 현재로서는 write에 대한 아무런 대응도 하고 있지 않다. 따라서 이를 적절히 수정해야 한다.

2. **void file_deny_write (struct file *file)**

```
void
file_deny_write (struct file *file)
{
  ASSERT (file != NULL);
  if (!file->deny_write)
    {
      file->deny_write = true;
      inode_deny_write (file->inode);
    }
}
```

- input argument로 들어오는 file struct의 deny_write 멤버 변수에 접근해서 해당 값을 true로 바꿔주는 함수이다. 이를 file을 열기 직전 적절히 호출해주어야 한다.

3. **void file_deny_write (struct file *file)**

```
void
file_allow_write (struct file *file)
{
  ASSERT (file != NULL);
  if (file->deny_write)
    {
      file->deny_write = false;
```

```
      inode_allow_write (file->inode);
    }
}
```

- file_deny_write 함수의 반대 역할을 해 주는 함수이다. input argument로 들어오는 file struct의 deny_write 멤버 변수에 접근해서 해당 값을 false로 바꿔준다.

4. **void file_close (struct file *file)**

```
void
file_close (struct file *file)
{
  if (file != NULL)
    {
      file_allow_write (file);
      inode_close (file->inode);
      free (file);
    }
}
```

- file_allow_write 함수를 호출해줘야 하는 부분이다. Pintos에서 제공하는 baseline에 이미 구현이 되어 있어서 해당 부분은 추가적인 구현이 필요 없다.

3. Design

3-1. Process terminate messages

Algorithm, Rationale

- System call 중 exit에 대한 처리이다. 따라서 exit 함수를 수정하는 방향으로 구현된다. 해당 함수가 호출될 때 printf("%s: exit(%d₩n)", variable_1, variable_2) 꼴로 exit 되는 process의 정보를 출력하고 이에 맞는 대응을 해주어야 한다. 이 때, variable_1에 해당되는 name은 command line argument를 제외하고 process_execute()함수를 통해 전달되는 이름의 전체여야 하고, variable_2에 해당되는 exit code는 정수이다. (in system call) 이 때 variable_1에 해당하는 thread_name과 exit code를 각각 thread_name() 함수와 input argument인 status에서 값을 가져와서 넣어주면 된다.

- User thread가 아닌 kernel thread를 실행할 때나 halt system call이 invoke되는 경우에는 이 메시지를 출력하지 않아야 하는 예외를 처리해야 한다.

새 함수 구현

- **1. src/userprog/syscall.c/void sys_exit (int status)**
- termination message를 출력하는 부분을 추가한다. 이 때 thread.c/thread_name() 함수로 running thread의 name을 가져올 수 있다.

```
const char *
thread_name (void)
{
  return thread_current ()->name;
}
```

- thread_name() 함수로 가져온 thread의 이름을 variable1로, argument로 받은 status를 variable2로 하여 printf를 통해 process name과 exit code를 출력한다. 그 뒤, thread_current() 함수로 접근하여 멤버 변수인 exit_status를 argument인 status로 바꿔준다. 마지막으로, thread_exit()함수를 수행하여 thread를 exit하기 위한 과정들을 수행한다.

3-2. Argument Passing

Algorithm, Rationale

- 현재로서는 process_execute() 함수가 새 process에 argument를 넘겨주는 부분이 구현되어 있지 않다.
- 원래의 의도대로 구현되어 있지 않은 process_execute 함수의 구현을 수정하여 command line을 공백 단위로 program name과 argument들로 나누는 작업을 수행할 수 있도록 구현하여야 한다. 예를 들어, process_execute("grep foo bar")이 호출되었다면, 이 함수는 grep이라는 이름을 가진 program을 foo와 bar을 각각 arg1과 arg2로 나눈 argument 형태로 넘겨주며 실행해야 한다.
- process_exectue() 함수를 실행한 이후, 그리고 그 안에서 thread_create() 함수를 실행하기 전에 command line에서 process name을 받아온다.

- start_process() 함수를 호출하고 난 이후, 그리고 load() 함수를 실행하여 success 값을 받아오기 전에 command line에서 argument를 parsing하여 넣어줘야 하는 인자를 분리한다. 이 때, lib/string.c에 구현되어 있는 strtok_r() 함수를 활용하여 문자열을 자르는 데 활용할 수 있다.
- argument_stack() 함수를 load 함수 호출 이후 호출하여 앞서 parsing한 인자들을 stack에 알맞게 삽입한다.

기존 함수 수정

- **1. process.c/tid_t process_execute (const char *file_name)**
- 기존에 argument인 file_name 전체를 thread_create() 함수의 argument, 즉 thread의 name으로 넘겨주던 문제점을 해결하기 위해 Pintos에서 제공된 strtok_r 함수를 이용하여 공백을 기준으로 앞부분을 잘라서 thread name을 가져올 수 있다. 그 뒤 이 thread name을 argument로 넣어 thread_create() 함수를 호출한다.

- **2. process.c/ static void start_process (void *file_name_)**
- 위의 수정과 비슷한 방식으로, 현재의 구현은 load 함수를 호출할 때 file_name_ 전체를 argument로 넣어주고 있다. 'echo x'를 예시로 든다면 우리는 load function에 'echo'만을 넣어줘야 한다. 따라서 같은 방식으로 strtok_r 함수를 통해 'echo'에 해당하는 command_name 변수를 정의해준 후 대입해주고, load가 완료되었다면 뒤에서 정의할 stack_argument() 함수를 호출하여 stack에 argument를 쌓고 esp 값을 업데이트해준다.

새 함수 구현

- **1. process.c/void stack_arguments(char* file_name, void **esp)**
- 앞서 언급한 start_process 함수에서 호출해서 stack에 argument들을 쌓고 esp 값을 올바르게 업데이트하기 위한 함수이다. file_name과 esp를 인자로 받는다.
- stack이 위에서 아래로 점점 확장되므로 esp값을 계속 빼면서 esp를 업데이트하는 방식을 고려할 수 있다.
- 함수의 전반적인 흐름은 다음과 같다.
  - 1. argc와 argv 등 data structure을 정의한다.

- 2. File_name을 공백 단위로 자르면서 argc(argument의 개수)를 업데이트한다.
- 3. argc만큼 반복문을 순회하면서 file_name을 공백 단위로 자르고, 이를 argv에 넣어준다.
- 4. argc만큼 반복문을 역으로 순회하면서 argv에 맞게 fake return address를 포함하여 stack과 esp를 update해준다.

3-3. System Call

Algorithm, Rationale

- argument를 user stack에서 가져오기 위해 pop해주는 함수를 구현해야 한다. 이를 위해 user stack에서 가져오기 전에 가져올 address가 user 영역에 있는지 확인하는 함수 역시 구현해야 한다. 또한 file descriptor를 구현하기 위해 pid로 child thread를 search하는 함수도 구현해야 한다.

기존 함수 수정

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

  File descriptor 구현을 위해 file table 값을 저장하는 fd_table, thread를 create할때 할 때 thread를 저장하는 parent, 자식 process를 담아두는 child_list와 이를 관리하기위한 child_elem, system call에서 exec와 wait 시 사용되는 semaphore을 나타내는 sema_exec, sema_wait, prgram이 load를 성공했는지, process가 종료했는지 여부를 저장하는 isLoad, isExit 를 추가한다.

- **2. src/userprog/syscall.c/static void syscall_handler (struct intr_frame *f UNUSED)**

  이제 위에서 말한 사항을 handler에 구현해야 한다. esp 값이 user 영역에 있는지 확인하고, 각 case 마다 구현된 system call 함수를 경우에 따라 호출이 되도록 구현한다.

- 3**. src/userprog/syscall.c**

  File을 다룰 때 다른 thread가 접근하지 못하도록 하는 lock (struct lock_file) 를 추

가한다. 이를 가지고 acquire & release 연산을 통해 관리한다.

-

새 함수 구현

1. void sys_halt (void)

- shutdown_power_off() 함수를 호출하여 Pintos를 종료한다.

2. void sys_exit (int status)

- 현재 실행중인 user program을 종료하고, 해당 status를 kernel에 return한다. 이 때 process termination message를 출력한다.

3. pid_t sys_exec (const char *file)

- command line으로 받은 arguments들을 passing해 해당하는 program을 실행한다. Program 실행 성공 시 새로운 pid를 리턴하고, 실패하면 -1을 return한다. Synchronization을 보장하기 위해 child process에 대해 semaphore을 생성한다.

4. int sys_wait (pid_t pid)

- parameter로 받은 pid와 동일한 program id를 가진 process를 기다리고, 해당 process의 exit status를 리턴한다.
- pid 값을 가진 child process를 찾고, 이 process가 종료될 때까지 기다린다. wait에 성공하였을 경우 child_list에서 해당 thread를 삭제하고, exit status를 return하고, 실패한 경우 -1을 return한다.
- 실패한 경우는 다음과 같다.
    a. pid는 호출한 프로세스의 직접적인 자식을 가리키지 않는다. pid가 호출한 프로세스의 직접적인 자식인 경우는 exec 호출로부터 pid를 성공적으로 반환받았을 때뿐이다.
    b. 자식 프로세스는 상속되지 않는다. 즉, A가 자식 B를 생성하고 B가 자식 프로세스 C를 생성하더라도, A는 C를 기다릴 수 없다. A가 프로세스 C를 기다리는 호출은 실패해야 한다. 마찬가지로 부모 프로세스가 종료되기 전에 자식 프로세스가 종료되면 새로운 부모에게 할당되지 않는다.
    c. wait를 호출한 프로세스는 이미 pid에 대해 한 번 이상 wait를 호출했다. 즉, 한 프로세스는 특정 자식에 대해 최대 한 번만 wait를 할 수 있다.

5. **bool sys_create (const char *file, unsigned initial_size)**

- 이름이 file이고 사이즈가 initial_size인 file을 생성한다. 성공시 true를 return 하고 실패시 false를 return 한다.

6. **bool sys_remove (const char *file)**

- 이름이 file인 file을 제거한다. 성공시 true를 return 하고 실패시 false를 return 한다.

7. **int sys_open (const char *file)**

- 이름이 file인 file을 open하고, file descriptor가 file을 가리키게 한다. 성공시 해당 file descriptor를 return 하고 실패시 -1를 return한다.

- file discriptor 0과 1은 console에 예약되어 있다. fd 0 (STDIN_FILENO)는 표준 입력을 나타내며, fd 1 (STDOUT_FILENO)는 표준 출력을 나타낸다. 이러한 file discriptor는 시스템 호출 인수로만 명시적으로 설명된 경우에만 유효하며, open 시스템 호출은 이러한 파일 디스크립터 중 하나를 반환하지 않는다.

- 각 process는 독립적인 file discriptor 세트를 가지며, file discriptor 자식 process에게 상속되지 않는다.

- 동일한 파일을 한 번 이상 여는 경우, 동일한 process 또는 다른 process에 의해 여는 경우 모든 open 호출은 새로운 file discriptor 를 반환한다. 동일한 파일에 대한 다른 file discriptor 는 각각 별도의 close 호출에서 독립적으로 닫히며 파일 위치를 공유하지 않는다.

8. **int sys_filesize (int fd)**

- file descriptor로 open 된 file의 size를 return한다.

9. **int sys_read (int fd, void *buffer, unsigned size)**

- file descriptor로 open 된 파일의 size만큼을 읽고 buffer에 저장한다. fd값이 0일 경우 이는 stdin을 의미하므로 키보드로 직접 input를 입력받고, 나머지 경우에는 기존에 구현이 되어있는 file_read함수를 통해 read한다. 성공 시 read한 size를 return하고 실패 시 -1을 return 한다.

10. **int sys_write (int fd, const void *buffer, unsigned size)**

- file descriptor로 open 된 파일의 size만큼을 write한다. fd값이 1일 경우 이는 stdout을 의미하므로 console을 통해 출력하고, 나머지 경우에는 기존에 구현이 되

어있는 file_write함수를 통해 write한다. 파일 끝을 넘어서 쓰기를 시도하는 경우 요청한 size만큼 byte를 쓰지 못하는 경우가 생긴다. 이렇게 항상 요청한 size만큼 return 되지는 않는다. 아무것도 쓰지 않았을 경우에는 0을 return한다.

11. void sys_seek (int fd, unsigned position)

- open된 file의 file descriptor에서 read하거나 write 할 다음 byte를 position만큼 이동한다.

12. unsigned sys_tell (int fd)

- open된 file의 file descriptor에서 read하거나 write 할 다음 byte의 position을 return 한다.

13. void sys_close (int fd)

- File Descriptor를 통해 file을 close 한다.

14. bool isAddressValid (void *addr)

- address가 user 영역에서 사용하는 address 값인지, 즉 kernal 영역의 address가 아닌지 확인해야 할 필요가 있기 때문에 이를 검사하는 함수를 구현함. 유효한 address일 시 true를, 유효하지 않을 시 false를 return 한다.

15. void get_argument (int *esp, int *arg, int count)

- user stack에 있는 argument를 pop 해 가져오는 함수. user stack에 argument가 있는 esp + 4에서부터 count만큼의 argument를 pop한다. pop 할때 마다 address가 user 영역인지 항상 확인해야한다.

16. struct thread *get_child_process (pid_t pid)

- pid가 동일한 child process를 찾고 해당 process의 descriptor를 return 한다. child_list에서 pid가 동일한 process가 있는지 찾으며, 없을시에는 null을 return 한다.

3-4. Denying Writes to Executables

Algorithm, Rationale

- execution하고 있는 파일에 write를 거부할 수 있는 (deny write) 코드를 추가하는 것이 목적이다. 이를 해결하기 위해 앞서 언급했던 file_deny_write와 file_allow_write 함수를 이용할 수 있다.

- load 함수가 호출될 때 file_read() 함수 호출을 통해 file이 open되므로 그 직전에 file_deny_write()를 호출한다. 그 뒤 file_close() 함수를 호출하며 file이 close될 때 file_allow_write()를 호출한다.

기존 함수 수정

- 이 함수들은 Pintos에 내장되어 있으므로 추가적으로 함수를 구현할 필요는 없다.

1. **bool load (const char *file_name, void (**eip) (void), void **esp)**

```
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;

...

// (file_deny_write 함수 호출 )

  /* Read and verify executable header. */
  if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
      || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
      || ehdr.e_type != 2
      || ehdr.e_machine != 3
      || ehdr.e_version != 1
      || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
      || ehdr.e_phnum > 1024)
    {
      printf ("load: %s: error loading executable\n", file_name);
      goto done;
    }

...

 done:
  /* We arrive here whether the load is successful or not. */
  file_close (file);
  return success;
}
```

- file_read 함수를 호출하기 전 내장되어있는 함수인 file_deny_write(file)을 호출한다.

2. **void file_close(struct file *file)**

```
void
file_close (struct file *file)
{
  if (file != NULL)
    {
      file_allow_write (file);
      inode_close (file->inode);
      free (file);
    }
}
```

- 앞서 언급했듯 baseline에 file_allow_write(file)을 호출하고 있어 별도로 수정해줄 부분은 없다.