Lab2 Final report

20210389 최준호, 20210509 정윤재

Implementation

1. Argument Passing

- 입력받은 command에서 적절히 문자열로 바꿔서 parsing해주고 pintos에 나오는 stack 구성에 맞게 stack에 argument들과 추가 정보들을 push 하는 과정이다.
- run user program에 대해서 `run_task`가 호출되면, `process_wait` 함수 안에서 `process_execute` 함수가 호출된다. 이어서 `thread_create` 함수가 호출되고 새롭게 생성되는 threadsms `start_process` 함수를 호출한다. 그리고 그 안에서 `load`함수가 호출되고 `setup_stack`에서 stack을 초기화해주게 된다.
  - run_task → process_execute → thread_create → start_process → load → setup_stack
  - 그런데, 현재는 `process_execue`에서 file_name을 입력받아 그 정보를 stack에 넣어주는 과정이 구현되어있지 않다.
- 먼저, `process_execute` 함수에서 file name을 받아오고 filesys_open을 통해 file을 open 해주는데, 우리 는 `strtok_r`을 이용해서 file_name에서 실행되는 함수 이름을 parsing해서 thread를 생성해주었다.
  - 이때, `name_copy`를 사용한 이유는 `strtok_r` 에 의해 file_name의 구성이 바뀌어 load에서 argv, argc를 저장할때 문제가 발생할 수 있기 때문이다. (마지막에 `palloc_free_page`로 free 해줘야한다.)
  - 해당 thread는 `start_process(file name)`을 호출하게 된다.

```
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);

  /* Argument passing */
  char *name_copy;
  char *name;
  char *name_ptr;
  name_copy = palloc_get_page(0);
  strlcpy(name_copy, file_name, PGSIZE);
  name = strtok_r(name_copy," ", &name_ptr);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (name, PRI_DEFAULT, start_process, fn_copy);
  palloc_free_page(name_copy); // new

  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);

  return tid;
}
```

- `start_process`에서는 intr_frame을 초기화 해주고, file name을 `load`에 전달한다.

```
static void start_process (void *file_name_) {
    ...
  /* Argument passing */
  success = load (file_name, &if_.eip, &if_.esp);

  /* System Call */
  thread_current()->is_load = success;
  sema_up(&thread_current()->exec_semaphore);

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

- `load`는 file_name을 공백 기준으로 parsing해서 나온 argument들을 argv에 저장해주고 갯수를 argc에 저장한 뒤, `setup_stack`을 호출하게 된다.

```
bool load (const char *file_name, void (**eip) (void), void **esp) {
    ...
  /* Argument Passing */
  int argc = 0 ;
  char** argv = palloc_get_page(0);
  char* token;
  char* save_ptr;
  for (token = strtok_r (file_name, " ", &save_ptr); token != NULL; token = strtok_r
  {
    argv[argc] = token;
    argc++;
  }
  ////////////////////

  /* Set up stack. */
  if (!setup_stack (esp, argv, argc))
    goto done;
  palloc_free_page(argv);   // new
    ...
}
```

- `setup_stack`은 `install_page`에 성공하게 되면, `argument_stack`을 호출하여 argv, argc값들을 stack에 push한다.
  - `setup_stack`이 esp 이외에도 argv, argc 값에 접근할 수 있도록 argument를 수정해주었다.

```
static bool setup_stack (void **esp, char** argv, int argc) {
  uint8_t *kpage;
  bool success = false;
  struct thread *t = thread_current ();

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success) {
        *esp = PHYS_BASE;
```

```
        argument_stack(esp, argv, argc);
      }
      else
        palloc_free_page (kpage);
    }

  return success;
}
```

- argument_stack은 아래와 같이 구현해주었다.

```
void argument_stack(void **esp, char **argv, int argc) {
    char *addr[argc];
    int len_null = 1;
    int len = 0;
    // 1. 문자열 복사
    for (int i = argc - 1; i >= 0; i--) {
        len = strlen(argv[i]) + len_null;
        *esp -= len;
        memcpy(*esp, argv[i], len);
        addr[i] = *esp;
    }

    // 2. 패딩
    while ((uintptr_t)(*esp) % 4 != 0) {
        *esp -= 1;
        *(uint8_t *)(*esp) = 0;
    }

    // 3. argv[i] 주소를 스택에 저장
    for (int i = argc; i >= 0; i--) {
        *esp -= sizeof(char *);
        if (i == argc)
        {
          *(void **)(*esp) = NULL;
        }
        else
        {
          *(void **)(*esp) = addr[i];
        }
    }

    // 4. argv의 주소 저장
    void *argv_ptr = *esp;
    *esp -= sizeof(void *);
    *(void **)(*esp) = argv_ptr;

    // 5. argc 값 저장
    *esp -= sizeof(int);
    *(int *)(*esp) = argc;

    // 6. fake return address
    *esp -= sizeof(void *);
```

```
    *(void **)(*esp) = NULL;
}
```

- stack에 data를 넣는 과정은 pintos 문서에 상세하게 나와있고 이를 똑같이 구현해주면 된다.
- `argument_stack` 함수는 esp 주소, argv, argc 를 argument로 전달받는다. 이 값들을 stack에 push해주는 방식으로 구현하였다.
  - esp 주소값을 이용하여 넣어주고 이를 조절하면서 stack에 새로운 값들을 push해주었다.
- step 1:
  - argv를 구성하는 문자열을 복사해준다. for문을 통해 argc만큼 반복하면서 argv의 element들에 접근하여 그 값들을 memcpy 함수를 이용해서 저장해준다. 이때 esp는 argv의 각 element 문자열의 크기에 맞춰서 빼주면 된다. ('\0'을 고려해 +1 해준다)
  - 그리고 argv[i]가 저장된 주소도 stack에 넣어줘야하기 때문에 addr에 순서대로 저장해준다.
    - 이때 주의할 점은 stack에는 argument 순서가 거꾸로되어서 들어간다는 것이다. 따라서 for문을 보면 argc-1 index부터 넣어주게된다.
- step 2:
  - 이후 padding하는 과정이 필요하다 앞선 과정에서 align을 고려하지 않고 숫자를 넣어줬기 때문에 4의 배수에 맞춰주기 위하여 while문을 통하여 esp값을 빼준다. 그리고 그곳에 0 값을 넣어준다. 4의 배수가 될때까지 이 과정을 while 문으로 반복한다.
- step 3:
  - 이후 아까 말한 address값을 stack에 push해준다. 마찬가지로 역순으로 넣어줘야 하기 때문에 for문을 이용해서 i=argc로 시작해서 넣어주게 된다. 이때 i==argc일 때는 null값을 넣어주고 그 이후부터는 아까 만들어놓은 addr list에서 가져와서 esp에 순서대로 넣어주게 된다. 이때는 넣어주는 값이 데이터가 아니라 주소기 때문에 sizeof(char*) 만큼 esp를 빼줘서 주소들을 넣어주게 된다.
- step 4:
  - argv와 argc의 주소를 저장해준다. 이때 data size를 생각해서 esp를 각각 sizeof(void*), sizeof(int)만큼 esp값들을 빼주게 된다.
- step 5:
  - 마지막으로 fake return address를 NULL로 채워서 마무리 해준다.
- pintos 문서에 따르면 hex_dump함수를 이용해서 stack에 제대로 값들이 push되었는지 확인 할 수 있다.

2. Process Termination Messages

- process가 종료될 때, termination message를 출력해야한다. 따라서, process 종료를 실행시키는 함수인 sys_exit 함수에서 thread의 이름과 exit code를 process termination message를 통해 출력하도록 한다.

```
void sys_exit(int status) {
  struct thread *t  = thread_current();
  t->exit_code = status;
  printf("%s: exit(%d)\n", t->name, status);
  thread_exit();
}
```

3. System call

- system call이 발생하는 경우, `syscall_handler` 가 실행되어 syscall_number에 맞는 sys 함수를 호출해야한다. 기존에는 `thread_exit` 만 호출하고 있었기에, pintos 문서에서 설명하는 13개의 syscall에 대한 sys 함수를 구현하고, switch 함수를 이용해서 system handler 함수에서 호출되도록 코드를 작성해주었다.

3-0. Basic Setting

- 가장 먼저 syscall을 구현하기 위해 thread structure에 아래와 같이 새로운 변수들을 선언해주었다.

```
struct thread {
...
     #ifdef USERPROG
   /* Owned by userprog/process.c. */
   uint32_t *pagedir;                  /* Page directory. */

   struct thread *parent;
   struct list_elem child_elem;
   struct list child_list;
   struct semaphore wait_semaphore;
   struct semaphore exec_semaphore;
   bool is_load;
   int exit_code;

   struct file **fd_table;
   struct file *running_file;
   int fd_cnt;
   #endif
...
};
```

- **struct thread *parent**
  - 현재 process의 parent process에 대한 정보를 저장한다.
  - thread_create 호출 시, 생성된 process의 current process가 parent가 된다.
- **struct list_elem child_elem, struct list child_list**
  - parent process가 create한 여러 child process들을 관리하기 위해 사용한다.
- **struct semaphore wait_semaphore**
  - child process가 끝날때까지 wait할 때, process의 synchronization을 위해 사용
- **struct semaphore exec_semaphore**
  - exec 과정에서 process의 synchronization을 위해 사용
- **bool is_load**
  - process의 현재 상태에 대한 정보를 전달하기 위해 사용
- **int exit_code**
  - process가 종료될 때의 exit code를 저장후, parent process에 전달
- **struct file **fd_table**
  - fd를 Index로 갖는 file들을 저장하는 table이다.
- **struct file *running_file**
  - running_file을 저장하는 변수로 running file의 write를 deny하는데 사용된다.
- **int fd_cnt**
  - open된 file의 수로 예외처리를 위해 사용하였다.

- **valid_access(void *addr)**

```
bool valid_access(void *addr) {
  if(addr >= (void *)0x08048000 && addr < (void *)0xc0000000) {
    // 해당 주소가 page table에 mapping 되어있는지 확인
    return pagedir_get_page(thread_current()->pagedir, addr) != NULL;
  }
```

```
  else {
    return false;
  }
}
```

- syscall_handler에서 sys 함수를 호출할 때, argument로 받는 주소가 user의 영역에 존재하는 것인지 확인해주는 함수이다.
- valid한 영역에 존재한다면, page_table에 mapping 되었는지 확인한 뒤, return한다.

- **get_argument(void *esp, int *arg, int count)**

```
void get_argument(void *esp, void **arg, int count) {
  for (int i = 0; i < count; i++) {
    void *check_address = esp + 4 * i;
    if (!valid_access(check_address)) {
      sys_exit(-1);
    }
    arg[i] = *(void **)(check_address);
  }
}
```

- argument passing을 통해 cmd에 입력된 argument들을 stack에 넣어주었다. 이후, syscall을 수행하기 위해서는 입력 받은 argument들을 sys 함수들에게 전달해주어야한다. 이를 위해 get_argument를 구현하였다. 이는, sys 함수에 필요한 argument의 수 만큼 stack에 저장된 값을 읽어 list에 저장한다. 그리고 해당 list는 sys 함수의 argument로 사용된다.

3-1. syscall_handler(struct intr_frame *f)

```
static void syscall_handler(struct intr_frame *f) {
  if (!valid_access(f->esp)) {
    sys_exit(-1);
  }

  void *args[3];
  switch (*(uint32_t *)(f->esp)) {
    case SYS_HALT:
      sys_halt();
      break;

    case SYS_EXIT:
      get_argument(f->esp + 4, args, 1);
      sys_exit((int)args[0]);
      break;

    case SYS_EXEC:
      get_argument(f->esp + 4, args, 1);
      f->eax = sys_exec((const char *)args[0]);
      break;

    case SYS_WAIT:
      get_argument(f->esp + 4, args, 1);
      f->eax = sys_wait((pid_t)args[0]);
      break;

    case SYS_CREATE:
      get_argument(f->esp + 4, args, 2);
```

```
      f->eax = sys_create((const char *)args[0], (unsigned)args[1]);
      break;

    case SYS_REMOVE:
      get_argument(f->esp + 4, args, 1);
      f->eax = sys_remove((const char *)args[0]);
      break;

    case SYS_OPEN:
      get_argument(f->esp + 4, args, 1);
      f->eax = sys_open((const char *)args[0]);
      break;

    case SYS_FILESIZE:
      get_argument(f->esp + 4, args, 1);
      f->eax = sys_filesize((int)args[0]);
      break;

    case SYS_READ:
      get_argument(f->esp + 4, args, 3);
      f->eax = sys_read((int)args[0], (void *)args[1], (unsigned)args[2]);
      break;

    case SYS_WRITE:
      get_argument(f->esp + 4, args, 3);
      f->eax = sys_write((int)args[0], (const void *)args[1], (unsigned)args[2]);
      break;

    case SYS_SEEK:
      get_argument(f->esp + 4, args, 2);
      sys_seek((int)args[0], (unsigned)args[1]);
      break;

    case SYS_TELL:
      get_argument(f->esp + 4, args, 1);
      f->eax = sys_tell((int)args[0]);
      break;

    case SYS_CLOSE:
      get_argument(f->esp + 4, args, 1);
      sys_close((int)args[0]);
      break;

    default:
      sys_exit(-1);
  }
}
```

- esp가 user 영역에 존재하는지 확인한다. (아닐 경우 sys_exit(-1)을 호출)
- esp에 저장된 값을 통해 어떤 system call에 대한 요청인지 확인한 뒤, switch-case를 이용해서 위의 구현된 함수들을 호출해준다. 그리고 주어진 경우가 아닌 다른 값이 esp에 저장되어있는 경우 `sys_exit(-1`)을 호출해준다.
- 각 case에 필요한 argument의 수는 argv[0]에서 확인할 수 있다.

3-2. sys_halt (void)

```
void sys_halt(void) {
  shutdown_power_off();
}
```

- 이미 구현되어 있는 `shutdown_power_off`를 호출하여 pintos를 종료한다.

3-3. sys_exit (int status)

```
void sys_exit(int status) {
  struct thread *t  = thread_current();
  t->exit_code = status;
  printf("%s: exit(%d)\n", t->name, status);
  thread_exit();
}
```

- 현재 process를 종료하는 함수이다.
- argument로 입력 받은 status를 current thread의 exit_code로 설정해준 뒤, `thread_exit`을 호출해서 thread를 종료한다.

3- 3. sys_exec(const char *cmd_line)

```
pid_t sys_exec(const char *cmd_line) {
  struct thread *child;
  if(!valid_access(cmd_line)) {
    sys_exit(-1);
  }

  // 새로운 child process 생성
  pid_t pid = process_execute(cmd_line);

  // process 생성에 실패한 경우
  if (pid == -1) return -1;

  // child에 current thread의 child process를 저장 (pid가 일치하는지 확인)
  child = get_child_process(pid);

  // process execute에서 start_process 실행 시 sema_up 실행 (child process가 load 될때까지 대기)
  sema_down(&(child->exec_semaphore));

  if (!child->is_load) {
    return -1;
  }

  return pid;
}
```

- argument로 전달받은 cmd_line이 valid 하다면 `process_execute` 함수를 실행해 child process를 생성하고 valid 하지 않다면 sys_exit(-1)이 호출되도록 한다.
- return된 pid가 유효하지 않은 경우, -1이 return되도록 하였다.
- `get_child_process`를 이용해 생성된 process를 child list에서 찾아 가져온다.

  ```
  // pintos/src/userporg/process.c
  struct thread* get_child_process (pid_t pid) {
    struct thread *parent = thread_current();
  ```

```
  struct list_elem *e;
  struct list *child_list = &parent->child_list;

  for (e = list_begin (child_list); e != list_end (child_list); e = list_next (e))
{
    struct thread *t = list_entry(e, struct thread, child_elem);
    if(t->tid == pid)
      return t;
  }
  return NULL;
}
```

- parent process는 child process가 load 되기 전까지는 대기해줘야한다. 따라서, exit_semaphore를 sema_down해서 wait하게 한다. (exit_semaphore는 `start_process`에서 `load`를 완료한 뒤, `sema_up`된다.)
- 제대로 child process가 제대로 load가 되었는지 확인하기 위해 `is_load = success`를 통해 load 성공 유무를 저장 후, 실패시 -1을 return한다.

```
// pintos/src/userprog/process.c
static void start_process (void *file_name_) {
    ...
    /* Argument passing */
  success = load (file_name, &if_.eip, &if_.esp);

  /* System Call */
  thread_current()->is_load = success;
  sema_up(&thread_current()->exec_semaphore);
    ...
}
```

3-4. sys_wait(tid_t child_tid)

```
int sys_wait(pid_t pid) {
  return process_wait(pid);
}
```

- `process_wait`을 호출함으로써 child process가 종료될때까지 parent process가 wait하도록 해준다.
- `process_wait`은 argument로 전달받은 tid의 thread가 존재하는 경우, wait_semaphore를 sema_down 해서 child thread가 종료될때까지 parent thread의 실행이 정지되도록하는 기능을 수행한다. 그리고 child thread가 종료될 때의 exit_code를 저장하여, return하고 종료된 child thread는 child list에서 삭제한다.

```
// pintos/src/userprog/process.c
int process_wait (tid_t child_tid) {
  struct thread *parent = thread_current();
  struct thread *child = get_child_process(child_tid);

  int status;
  struct list_elem *e;
  if (child==NULL) {
    return -1;
  }
  sema_down(&child->wait_semaphore);
  status = child->exit_code;
  list_remove(&(child->child_elem));
  palloc_free_page(child);
```

```
  return status;
}
```

- `thread_exit`에 의해 thread가 exit될 때, wait_semaphore를 sema_up 해주어, parent thread가 다시 실행될 수 있도록 해주었다.

```
void thread_exit (void) {
    ...
    list_remove (&thread_current()->allelem);

  if(thread_current() != initial_thread){
      sema_up(&(thread_current()->sema_wait)); // new
  }
    ...
}
```

- child process가 kill에 의해 exit되는 경우 exit_code가 -1이 되는데, for 문을 통해서 child의 exit_code가 -1인 경우, child list에서 제거하고 -1을 return하도록 해줘야한다. 그리고 이를 `process_execute` 함수에 구현해 처리해주었다. (???? 필요 없는 것 같음)

```
tid_t process_execute (const char *file_name) {
    ...
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);

  /* System call */
  struct list_elem *e;
  struct thread* child;
  for(e = list_begin(&thread_current()->child_list);e!= list_end(&thread_current()->child_list);e=list_next(e)) {
    child = list_entry(e, struct thread, child_elem);
    if(child->exit_status == -1)
      return -1;
  }
  return tid;
}
```

3-5. sys_create(const char *file, unsigned initial_size)

```
bool sys_create(const char *file, unsigned initial_size) {
  if (file == NULL) {
    sys_exit(-1);
  }
  if (!valid_access(file)) {
    sys_exit(-1);
  }

  return filesys_create(file, initial_size);
}
```

- file의 주소가 user 영역인지 확인한 뒤, valid한 영역에 존재한다면, `filesys_create`함수를 호출하여 file을 생성한다.

3-6. sys_remove(const char *file)

```
bool sys_remove(const char *file) {
  if(!valid_access(file)) {
    sys_exit(-1);
  }
  return filesys_remove(file);
}
```

- file의 주소가 user 영역인지 확인한 뒤, valid한 영역에 존재한다면, `filesys_remove` 함수를 호출하여 file을 삭제한다.

3-7. sys_open(const char *file)

```
int sys_open(const char *file) {
  if(!valid_access(file)){
    sys_exit(-1);
  }

  struct file *f;

  lock_acquire(&file_lock);
  f = filesys_open(file);

  if (f == NULL) {
    lock_release(&file_lock);
    return -1;
  }

  if (!strcmp(thread_current()->name, file)) {
    file_deny_write(f);
  }

  int fd = thread_current()->fd_cnt;
  thread_current()->fd_table[fd] = f;
  thread_current() -> fd_cnt++;
  lock_release(&file_lock);
  return fd;
}
```

- `sys_open`은 user로부터 file open을 요청받아, file object를 생성하고, fd를 return하는 함수이다.
- file의 주소가 user 영역인지 확인한 뒤, 아니라면 `sys_exit(-1)` 을 호출한다.
- file object를 생성하기 전, `lock_acquire` 를 통해 file system에 대한 접근을 lock을 획득한다. 그리고 `filesys_open` 을 이용해 'file'이라는 이름을 가진 file을 open한다. 추가적으로 file이 존재하지 않는 경우에 대한 예외처리를 해주었다.
- 현재 열려있는 thread의 이름이 open한 file과 같은 이름을 갖는다면, `file_deny_write(f)` 를 통해 file에 대한 write를 거부한다. 이는 실행 파일이 open된 상태에서 수정되는 것을 예방하기 위함이다.
- file descriptor에 필요한 정보를 저장한 뒤, lock을 해제하고 return한다.

3-8. sys_filesize(int fd)

```
int sys_filesize(int fd) {
  struct file *f;

  if(fd < thread_current()->fd_cnt) {
      f = thread_current()->fd_table[fd];
    return file_length(f);
   }
```

```
  else {
    f = NULL;
    return -1;
  }
}
```

- argument로 받은 fd에 해당하는 file의 filesize를 return하는 함수이다.
- fd_table에 저장된 file object를 불러온 뒤, `file_length`를 이용해 file size를 찾고 return한다.

3-9. sys_read(int fd, void *buffer, unsigned size)

```
int sys_read(int fd, void *buffer, unsigned size) {
  for(int i=0;i<size;i++) {
    if(!valid_access(buffer+i)) {
      sys_exit(-1);
    }
  }
  int read_size = 0;
  struct file *f;

  if (fd < 0) {
    sys_exit (-1);
  }

  if(thread_current()->fd_cnt < fd) {
    sys_exit (-1);
  }

  lock_acquire(&file_lock);

  if (fd == 0) {
    unsigned int i;
    for (i = 0; i < size; i++) {
      if (((char *)buffer)[i] == '\0')
        break;
    }
    return i;
  }

  f = thread_current()->fd_table[fd];
  if (f == NULL) {
    sys_exit(-1);
  }

  read_size = file_read(f, buffer, size);

  lock_release(&file_lock);

  return read_size;
}
```

- file에서 size만큼의 data를 read하는 함수이다.
- 먼전, buffer에 size만큼 read한 내용이 추가되었을때, 여전히 유효한 주소를 갖는지 확인해준다. 그리고 fd에 대한 예외 case들에 대해서 `sys_exit(-1)`을 호출한다.

- read를 진행하기 전, `lock_acquire`를 이용해 다른 process들의 접근을 막아준다.
- fd 값을 확인하고, 그 값이 0이라면 표준 입력인 경우이므로 '\0'이 읽히기 전까지 buffer에 저장해준다. 그 외의 경우에는 `file_read`를 호출해서 file의 내용을 읽어준다. 두 경우 모두 읽은 size의 크기를 return한다.

3-10. sys_write(int fd, const void *buffer, unsigned size)

```
int sys_write(int fd, const void *buffer, unsigned size) {
  for(int i=0;i<size;i++) {
    if(!valid_access(buffer+i)) {
      sys_exit(-1);
    }
  }

  int write_size = 0;
  struct file *f;

  if (fd < 1) {
    sys_exit (-1);
  }

  if (fd > thread_current()->fd_cnt) {
    sys_exit(-1);
  }

  lock_acquire(&file_lock);

  if (fd == 1) {
      // buffer의 data를 size 만큼 출력
    putbuf(buffer, size);
    lock_release(&file_lock);
    return size;
  }
  else {
    f = thread_current()->fd_table[fd];

    if (f == NULL) {
      sys_exit(-1);
    }
    write_size = file_write(f, (const void *)buffer, size);
    lock_release(&file_lock);
    return write_size;
  }
}
```

- file의 data를 size 만큼 write하는 함수이다.
- 먼전, buffer에 size만큼 wirte한 내용이 추가되었을때, 여전히 유효한 주소를 갖는지 확인해준다. 그리고 fd에 대한 예외 case들에 대해서 `sys_exit(-1)`을 호출한다.
- write을 진행하기 전, `lock_acquire`를 이용해 다른 process들의 접근을 막아준다.
- fd가 1인 경우는 표준 출력을 의미하므로 화면에 buffer 값을 출력해준다. 그리고 그 외의 경우 `file_write`를 호출해서 size만큼 write해준다. 그리고 write이 완료되면 lock을 해제하고, write한 size의 크기를 return한다.

3-11. sys_seek(int fd, unsigned position)

```
void sys_seek(int fd, unsigned position) {
  struct file *f;
```

```
  if(fd < thread_current()->fd_cnt) {
    f = thread_current()->fd_table[fd];
    file_seek(f, position);
  }
  else {
    f = NULL;
  }

}
```

- open된 file의 위치를 이동하는데 사용되는 함수이다.
- fd의 index를 갖는 file이 fd_table에 존재할 경우, `file_seek`를 호출해서 position으로 file f의 위치를 옮겨준다.

3-12. sys_tell(int fd)

```
unsigned sys_tell(int fd) {
  struct file *f;

  if(fd < thread_current()->fd_cnt) {
    f = thread_current()->fd_table[fd];
    return file_tell(f);
   }
  else {
    f = NULL;
    return 0;
  }
}
```

- fd의 index를 갖는 file의 위치를 return하는 함수이다.
- fd의 index를 갖는 file이 fd_table에 존재할 경우, `file_tell`을 호출해서 file의 위치를 return한다.

3-13. sys_close(int fd)

```
void sys_close(int fd) {
  struct file *f;
  if(fd < thread_current()->fd_cnt) {
      f = thread_current()->fd_table[fd];
   }
  else {
    f = NULL;
    return;
  }

  file_close(f);
  thread_current()->fd_table[fd] = NULL;
}
```

- fd 값에 위치하는 file을 close하는 함수이다.
- fd의 index를 갖는 file이 fd_table에 존재할 경우, file_close를 호출한다.
- fd_table에서 해당 file을 제거한다.

4. Denying Writes to Executable

- file이 실행되는 중에 data가 변경된다면, 의도와 다른 결과가 발생할 수 있다. 따라서, 실행중인 file에 write되는 것을 예방해줘야한다. `file_deny_write`을 이용해 이를 구현할 수 있다. 코드는 아래와 같다.

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

  - argument로 받아오는 file의 deny_write (write 권한 변수)가 false라면 true로 바꿔준 뒤, `inode_deny_write`를 호출함으로써 file이 write되지 않도록 한다.

- 이후 `file_close`가 호출되면 `file_allow_write`를 호출해서 deny_wrtie을 해제해주는 것을 볼 수 있다.

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

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

- load 함수에서 filesys_open을 통해 file을 open하고 정상적으로 file이 open되었자면, `running_file`을 file로 설정해주고, `file_deny_write`를 호출해서 실행중인 file이 write되는 것을 예방한다.

```
bool load (const char *file_name, void (**eip) (void), void **esp) {
  ...
  /* Open executable file. */
  lock_acquire(&file_lock); // new
  file = filesys_open (argv[0]);

  if (file == NULL)
    {
      lock_release(&file_lock); // new
      printf ("load: %s: open failed\n", file_name);
      goto done;
    }

  /* Denying */
  t->running_file = file;   // new
  file_deny_write(file);    // new
```

```
  lock_release(&file_lock); // new
  ...
}
```

- 추가적으로 `sys_open` 함수에서 file이 open 되었을때, file_deny_write을 호출해서 running file이 write되지 않도록 해주었다.
- process_exit에서 process가 종료되기 전에 할당되어있던 file들과 fd_table 들을 모두 할당 해제해준다. 또한, `file_close` 를 호출해서 running_file을 close 함으로서 deny_write를 해제해준다.

```
void process_exit (void) {
  struct thread *cur = thread_current ();
  uint32_t *pd;

  /* Denying */
  for(int i = 2; i < cur->fd_cnt; i++) {
    sys_close(i);
  }

  palloc_free_page(cur->fd_table);
  file_close(cur->running_file);
    ...
}
```

5. Discussion

- 위의 구현까지 완료하였을때, 일부 bad-~ test case에 실패하는것을 확인하였다.
- 이를 해결하기 위해 page_fault가 발생하거나 잘못된 memory 접근이 발생하는 경우에 대해서 sys_exit(-1)이 호출되도록 아래와 같이 추가적으로 구현해서 80 pass를 확인할 수 있었다.

```
// userprog/exception.c
static void page_fault (struct intr_frame *f) {
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */
    ...
  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  if(not_present)
  {
    sys_exit(-1);
  }
  if (!user || is_kernel_vaddr(fault_addr)) {
    sys_exit(-1);
  }
    ...
}
```

- 최종적으로 80 pass를 확인할 수 있었다.


pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 80 tests passed.