Pintos Project: 2. User program

Final Report – team 30

20210054 정하우

20210716 최대현

Context

1. **Implementation**

   1-1. Argument passing

   - Algorithm

   - Data Structure

   - Function Implementation

   1-2. Process terminatation messages

   - Algorithm

   - Function Implementation

   1-3. System call

   - Algorithm

   - Data Structure

   - Function Implementation

   1-4. Denying writes to executables

   - Algorithm

   - Data Structure

   - Function Implementation

2. **Discussion**

   2-1 multi-oom test

2-2 rox-test

1-1. Argument passing

Algorithm

1. process_execute() 함수가 호출되면 argument인 char * file_name을 내장 함수인 strtok_r 함수로 자른다.
2. 공백 이전의 첫 번째 토큰을 새롭게 정의한 char *name 변수에 저장하고, name을 thread_create의 첫 번째 인자 (char *name)로 넘겨 thread를 새롭게 생성한다.
3. Thread_create에서 start_process를 실행할 함수로 넘겨주고, start_process 함수에서 char **argv와 int argc를 변수로 선언한 뒤 fn_to_argument() 함수를 호출하여 값을 설정해준다.
4. argv[0]를 인자로 넘기면서 load 함수를 호출하여 메모리를 load하고 성공 여부를 반환받는다.
5. 메모리 load가 성공했다면 argument_stack() 함수를 argc, argv와 함께 호출하여 stack에 argument, argment의 주소, return address를 넣어준다.

Data Structure

- **argc, argv:** Pintos에서 user program을 실행할 때 argument를 인자로 받는데, argc는 전달되는 인자의 개수이고, argv는 문자열 형태의 인자들의 주소를 저장하는 포인터 배열이다. 포인터 배열이기 때문에 argv[0], argv[1]... 등 배열의 각 원소가 가리키는 값을 참조함으로써 argument의 내용도 얻을 수 있다.
- Pintos에서 argc ,argv는 명시적인 data structure로 구현되어 사용되기보다는 후에 구현할 함수들에서 필요 시 file name을 잘라 argc와 argv로 만들어주는 형태로 임시적으로 사용된다.

Function Implementation

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;
  char *fn_copy_copy;
  char *name; //new
  char *parsed; // new, remainder of file_name
  /* 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);

  fn_copy_copy = palloc_get_page(0);
  strlcpy(fn_copy_copy, file_name, PGSIZE);
  name = strtok_r(fn_copy_copy," ", &parsed);

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

  .
  .
  .

  return tid;
}
```

- 기존에 선언되어 있었던 fn_copy 이외에도 file_name의 앞 부분을 잘라서 name 변수에 할당하기 위한 fn_copy_copy 변수를 새롭게 정의한다. 이 때, fn_copy는 thread_create 함수를 호출하는 데 필요하기 때문에 fn_copy_copy를 별도로 선언하여 사용한다.
- fn_copy_copy 변수를 palloc_get_page 함수를 통해 동적 할당해주고, file_name 값을 strlcpy를 통해 대입해준다.
- strtok_r 함수에 넣어 " ", 즉 공백 단위로 자른 뒤 앞 부분을 name으로 반환해준다.
- name 변수를 인자로 thread_create 함수를 호출하여 thread를 생성해준 뒤, 해당 함수를 반환하면 palloc_free_page()를 통해 동적 할당했던 메모리를 반환한다.

2. src/userprog/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;

  char* fn_copy = palloc_get_page(0);
  int argc;
  strlcpy(fn_copy,file_name,PGSIZE);
  char** argv;


.
.
.
  argv = palloc_get_page(0); // new
  argc = fn_to_argument(argv, fn_copy); // new
  success = load (argv[0], &if_.eip, &if_.esp);

  thread_current()->isLoad = success;
  if(success){ //new
    argument_stack(argv, argc, &if_.esp);
  }
  sema_up(&thread_current()->sema_exec);

  //printf("Checking Memory\n"); // for debugging
  //hex_dump(if_.esp, if_.esp, PHYS_BASE - if_.esp, true); // for debugging
  palloc_free_page (argv); //new
  palloc_free_page (fn_copy);

  /* If load failed, quit. */
.
.
.

}
```

- fn_copy 변수를 동적 할당한 뒤 file_name 변수를 복사해준다.
- argv와 argc 변수를 선언하여 사용하고, argv의 경우 동적 할당이 필요하여 할당 뒤 fn_to_argument() 함수를 호출하여 fn_copy 값을 argv에 잘라 넣어주고 argc (argument 의 개수)를 반환한다.
- load() 함수에 file_name을 넣어주던 것을 명령어에 해당하는 부분인 argv[0]을 넣는 것으로 바꿔주고, 메모리 로드를 성공했는지 여부를 success 변수로 반환한다.

- success 값이 true라면 argument_stack 함수를 호출하여 stack 공간에 argument, argument의 주소, return address를 차례대로 push해준다.

3. src/userprog/process.c/int fn_to_argument(char **argv, char *file_name)

```
int fn_to_argument(char **argv, char *file_name){

  char *token;
  char *parsed;

  int argc = 0;

  for (token = strtok_r (file_name, " ", &parsed); token != NULL;
  token = strtok_r (NULL, " ", &parsed), argc++){

    argv[argc] = token;

  }

  return argc;

}
```

- 새롭게 구현한 함수이다. file_name을 인자로 받아 적절하게 argv와 argc를 설정해준다.
- for문을 순회하며 초기조건으로는 file_name을 넣어주면서 strtok_r을 호출하고, 그 뒤에는 strtok_r의 첫 번째 인자를 NULL로 설정하면서 가장 최근에 잘리고 남은 문자열을 기억하고, 해당 문자열에서 계속 공백 단위로 token을 parsing한다.
- argc는 처음에 0으로 초기화하고, token의 개수가 늘어날 때마다 1씩 증가하여 최종 값을 반환한다.

4. src/userprog/process.c/void argument_stack(char **argv, int argc, void **esp)

```
// newly impelmented
void argument_stack(char **argv, int argc, void **esp){

  int i = 0;
  int len = 0;

  //printf("%s", argv); //for debugging
  //printf("%d", argc); //for debugging
```

```
  // push argv[i]
  for (i = argc - 1; i >= 0; i--) {
    len = strlen(argv[i]);
    *esp -= len + 1;
    strlcpy(*esp, argv[i], len + 1);
    argv[i] = *esp;
  }

  // align stack
  if (((uint32_t)*esp) % 4 !=0) *esp -= ((uint32_t)*esp) % 4;

  // push null
  *esp -= 4;
  **(uint32_t **)esp = 0;

  // push argv[i]
  for (i = argc - 1; i >= 0; i--) {
    *esp -= 4;
    **(uint32_t **)esp = argv[i];
  }

  // push argv
  *esp -= 4;
  **(uint32_t **)esp = *esp + 4;

  // push argc
  *esp -= 4;
  **(uint32_t **)esp = argc;

  // push return address
  *esp -= 4;
  **(uint32_t **)esp = 0;

}
```

- 메모리에 argument, argument의 주소, return address 등을 넣어주는 함수이다.
- argument로 들어오는 esp 포인터 값을 조작하는 방식으로 구현되었다.
- 스택 메모리에 넣어주는 순서는 pintos document를 참고하여 그대로 구현하였고. 순서는 *argv[i] -> stack word align -> null -> argv[i] -> argv -> argc -> return address 순으로 stack에 push해주었다. 이 때 stack word align의 경우 *esp의 4로 나눈 나머지를 체크하여 나머지만큼을 더 빼 줌으로써 stack 공간을 4의 배수로 맞춰줄 수 있었다.

Testing argument pass

argument pass를 구현하고 start_process 함수 내에서 argument_stack 함수를 호출하여 argument를 스택에 넣어주는 과정을 거치고 나서, hex_dump 함수를 호출한 뒤 pintos에서 user program을 실행하여 argument parse, stack에 push 등 pintos에서 요구하는 사항대로 argument passing이 원활히 진행됐는지 확인할 수 있다. 예를 들어,

```
static void
start_process (void *file_name_)
{
.
.
.
  argv = palloc_get_page(0); // new
  argc = fn_to_argument(argv, fn_copy); // new
  success = load (argv[0], &if_.eip, &if_.esp);

  thread_current()->isLoad = success;
  if(success){ //new
    argument_stack(argv, argc, &if_.esp);
  }

  printf("Checking Memory\n"); // for debugging
  hex_dump(if_.esp, if_.esp, PHYS_BASE - if_.esp, true); // for debugging
  palloc_free_page (argv); //new
  palloc_free_page (fn_copy);
```

위와 같이 hex_dump 함수를 호출하도록 만들고 pintos –v -- run 'echo x' 커맨드를 입력하면, 다음과 같이 stack에 값이 잘 담기고 있음을 확인할 수 있다.

```
Executing 'echo x':
Execution of 'echo x' complete.
success: 0
Checking Memory
bfffffe0  00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |................|
bffffff0  fe ff ff bf 00 00 00 00-00 65 63 68 6f 00 78 00 |.........echo.x.|
system call!
```

1-2. Process termination messages

Algorithm

- process가 종료될 때, termination message를 출력한다. 즉 해당 부분을 추후 system call에서 구현하는 sys_exit에서 처리해줄 수 있다.

Function Implementation

1. **src/userprog/syscall.c/ void sys_exit(int status)**

```
void sys_exit(int status)
{
  thread_current()->exit_status = status;
  printf("%s: exit(%d)\n", thread_name(), status);
  thread_exit();
}
```

- 현재 process를 종료하는 함수이다. 해당 thread에 현재 상태인 status를 저장한 후, 현재 thread를 종료하기 전 process termination message를 출력한다. process termination message에는 thread의 이름과 status를 포함된다.

1-3. System call

Algorithm

- Design report에서 언급한 내용처럼, 현재 pintos에는 system call이 발생시 syscall.c에 있는 syscall_handler 함수가 실행된다. 하지만 thread_exit 함수만 실행될 뿐, 아무런 구현이 되어있지 않다. 따라서 각 System call case에 대해 알맞은 system call을 수행하도록 구현하는 것이 목표이다.
- 이번 project에서 다루는 system call은 총 13가지다. **(halt, exit, exec, wait, create, remove, open, filesize, read, seek, tell, close)** 해당 system call이 발생될 때 syscall_handler 함수에서 호출되는 13가지 함수를 구현하고, syscall_handler에서 switch-case 문으로 각 case 별로 알맞은 함수가 호출이 되도록 구현해야 한다.

Data Structure

1. **src/thread/thread.c/struct thread**

```
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;
    bool isLoad;
    bool isExit;
    struct semaphore sema_wait;
    struct semaphore sema_exec;
    int exit_status;

    struct file **fd_table;
    int fd_count;

    struct file *file_run;
#endif

…

  };
```

```
src/thread/thread.c

tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
…
  #ifdef USERPROG
    t->parent = thread_current();
    sema_init(&(t->sema_wait), 0);
    sema_init(&(t->sema_exec), 0);
    t->isExit = false;
    t->isLoad = false;
    list_push_back(&(thread_current()->child_list), &(t->child_elem));

    t->fd_count = 2;
    t->fd_table = palloc_get_page(PAL_ZERO);
    if(t->fd_table == NULL) return TID_ERROR;
  #endif

  …

}
```

1. **thread.h/struct thread/struct thread *parent**

create 된 process의 parent thread를 담는 member variable. thread create 시 현재 thread가 parent thead가 된다.

2. **thread.h/struct thread/struct list_elem child_elem**

   child thread 들을 관리하기 위한 member variable.

3. **thread.h/struct thread/struct list child_list**

   child thread를 담고 있는 list 형식의 member variable.

4. **thread.h/struct thread/bool isLoad**

   process의 program 이 memory에 load 되었는지 여부를 담고 있는 member variable. thread_create함수에서 false으로 초기화된다.

5. **thread.h/struct thread/bool isExit**

   process의 exit 여부를 확인하는 member variable. thread_create함수에서 false으로 초기화된다.

6. **thread.h/struct thread/struct semaphore sema_wait**

   wait 과정에서 process의 synchronization을 위한 semaphore member variable. thread_create함수에서 0으로 초기화된다.

7. **thread.h/struct thread/struct semaphore sema_exec**

   exec 과정에서 process의 synchronization을 위한 semaphore member variable. thread_create함수에서 0으로 초기화된다.

8. **thread.h/struct thread/int exit_status**

   child process가 종료된 방식의 정보를 담는 member variable.

9. **thread.h/struct thread/struct file** fd_table**

   file descriptor table 정보를 담는 member variable.

10. **thread.h/struct thread/int fd_count**

    현재 file descriptor table의 최대 index 정보를 담는 member variable. 이 값을 통해 다른 함수에서 file descriptor table에 값을 가져올 때 유효한 index인지 판단할 때 사용된다. stdin과 stdout은 이미 존재하므로 thread_create에서 2로 초기화된다.

11. **thread.h/struct thread/struct file* file_run**

    현재 실행되고 있는 file의 정보를 담는 member variable.

Function Implementation

1. **src/userprog/syscall.c/static void isAddressValid(void *addr)**

```
bool isAddressValid(void *addr)

{
  if(addr >= (void *)0x08048000 && addr < (void *)0xc0000000)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

- parameter로 받은 주소가 user 영역에 속하는지 확인하는 함수이다. user 영역이 아닐 시 false를 return 한다.

2. **src/userprog/syscall.c/void get_argument(void *esp, int *arg, int count)**

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

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

- user stack에 있는 값을 pop해서 kernel에 저장하는 함수이다.
- Pop할 값의 주소가 user 영역인지 확인한 후 arg로 pop한다. User 영역이 아닌 주소의 값을 pop 하려고 하면 sys_exit(-1)을 호출한다.

3. **src/userprog/syscall.c/void sys_halt(void)**

```
void sys_halt(void)

{
  shutdown_power_off();
}
```

- 내장 구현되어 있는 shutdown_power_off 함수를 호출하여 pintos를 종료하는 함수이다.

4. **src/userprog/syscall.c/ void sys_exit(int status)**

```
void sys_exit(int status)

{
  thread_current()->exit_status = status;
  printf("%s: exit(%d)\n", thread_name(), status);
  thread_exit();
}
```

- 현재 process를 종료하는 함수이다.
- 해당 thread에 현재 상태인 status를 저장한 후, 1-2. process termination message에서 언급했듯, process termination message를 출력한 뒤 thread_exit 함수를 호출해 현재 thread를 종료한다.

5. **src/userprog/syscall.c/ pid_t sys_exec(const char *file)**

```
pid_t sys_exec(const char *file)

{
  struct thread *child;
  if(!isAddressValid(file))
  {
    sys_exit(-1);
  }
  pid_t pid = process_execute(file);
  if (pid == -1)
  {
    return -1;
```

```
  }
  child = get_child_process(pid);
  sema_down(&(child->sema_exec));
  if (!child->isLoad)
  {
    return -1;
  }
  return pid;
}
```

- process.c에 구현되어 있는 process_execute를 통해 child process를 생성하고 program을 실행시키는 함수이다.
- File 주소가 user 영역인지 확인한다. User 영역이 아닐 시 sys_exti(-1)을 호출한다.
- semaphore을 이용해 child process의 program이 load 될 때까지 wait한 후 isLoad 값을 통해 child process의 load 성공여부를 판단한다.
- Load 성공 시 child process의 pid를, 실패 시 -1를 return한다.
- 이때 child process가 load를 성공했을 시 isLoad를 true로 설정하는 start_process 함수에서 sema_up를 하여 parent process의 wait를 해제한다.

6. src/userprog/syscall.c/ int sys_wait(pid_t pid)

```
int sys_wait(pid_t pid)

{
  return process_wait(pid);
}
```

- 후에 언급할 process_wait 함수를 호출해 child process가 종료할 때까지 wait하는 함수이다.

7. src/userprog/process.c/ int process_wait (tid_t child_tid)

```
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->sema_wait);
  status = child->exit_status;
  list_remove(&(child->child_elem));
  palloc_free_page(child);

  return status;
}
```

- 현재 process의 child process의 종료를 기다리는 함수. get_child_process 함수를 통해 child process를 가져온다. 이때 child process가 없다면 -1을 return한다. Child process가 있는 경우, child list에서 해당 child를 순회하여 찾는다. Child_list에서 child process를 찾으면, semaphore를 down시켜 해당 process가 종료될때까지 wait한다. Child process가 종료되면 어떤 방식으로 종료되었는지에 대한 정보를 담는 exit_status 를 status에 저장한 후, child_list에서 해당 process를 제거한 후 free를 한다.

8. src/userprog/syscall.c/ bool sys_create(const char *file, unsigned initial_size)

```
bool sys_create(const char *file, unsigned initial_size)

{
  if(!isAddressValid(file)||file==NULL)
  {
    sys_exit(-1);
  }
  return filesys_create(file, initial_size);
}
```

- File 주소가 user 영역인지 확인한다. User 영역이 아닐 시 sys_exit(-1)을 호출한다.
- filesys_create 함수를 호출하여 file을 create 하는 함수이다.

9. src/userprog/syscall.c/ bool sys_remove(const char *file)

```
bool sys_remove(const char *file)
```

```
{
  if(!isAddressValid(file))
  {
    sys_exit(-1);
  }
  return filesys_remove(file);
}
```

- File 주소가 user 영역인지 확인한다. User 영역이 아닐 시 sys_exit(-1)을 호출한다.
- fillesys_remove 함수를 통해 file을 remove하는 함수이다.

10. src/userprog/syscall.c/ int sys_open(const char *file)

```
int sys_open(const char *file)

{
  if(!isAddressValid(file))
  {
    sys_exit(-1);
  }
  struct file *f;

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

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

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

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

- File 주소가 user 영역인지 확인한다. User 영역이 아닐 시 sys_exit(-1)을 호출한다.
- file을 open하는 system call을 처리하는 함수이다.

- lock_file에 대한 lock acquire를 통해 다른 process의 file 접근을 막는다.
- 만약 file이 null일 경우 lock을 release 한 후 -1을 return 한다.
- 현재 thread와 file의 이름을 비교해 해당 file의 실행 여부를 판단하고, 만약 해당 file이 실행중이라면 file_deny_write함수를 통해 해당 파일에 write를 방지한다**(1-4. Denying writes to executables).**
- 위 case가 아닐 경우, file descriptor table에 추가한다. 아직 fd_table에 없는 index인 fd_count에 file을 추가하고, fd_count 값을 1 추가한다.

11. src/userprog/syscall.c/ int sys_filesize(int fd)

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

  if(fd < thread_current()->fd_count)
  {
    f = thread_current()->fd_table[fd];
  }
  else
  {
    f=NULL;
  }


  if (f==NULL)
  {
    return -1;
  }
  else
  {
    return file_length(f);
  }
}
```

- 해당 fd 값에 해당하는 file의 filesize를 return 하는 함수이다.
- 구현되어 있는 file_length함수를 호출하여 filesize를 return한다.
- 만약 fd 값이 fd_count보다 크거나 같으면 -1을 return하여 예외처리한다.

12. src/userprog/syscall.c/ int sys_read(int fd, void *buffer, unsigned size)

```
int sys_read(int fd, void *buffer, unsigned size)

{
  int i;
  for(int i=0;i<size;i++)
  {
    if(!isAddressValid(buffer))
    {
      sys_exit(-1);
    }
  }
  int read_size = 0;
  struct file *f;
  int current_fd=thread_current()->fd_count;

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

  lock_acquire(&lock_file);

  if (fd == 0)
  {
    unsigned int i;
    for (i = 0; i < size; i++)
    {
      if (((char *)buffer)[i] == '\0')
        break;
    }
    read_size = i;
  }
  else
  {
    f = thread_current()->fd_table[fd];

    if (f==NULL)
    {
      sys_exit(-1);
    }
    read_size = file_read(f, buffer, size);
  }

  lock_release(&lock_file);

  return read_size;
```

```
}
```

- file의 data를 size만큼 read하는 system call 함수이다.
- 먼저 size 만큼 read 할 buffer가 user 영역인지 확인한다. 그리고 fd값이 0보다 작거나 fd_count보다 크면 file descriptor table에는 없는 file이므로 exit한다. 위 case가 아닌 경우 read가 가능한 경우이므로, file read를 진행한다. 먼저 다른 process의 접근을 막기 위해 lock을 acquire 한다. 그리고 fd 값을 확인한다. 만약 fd 값이 0이라면 표준 입력인 경우이므로 키보드 입력을 '₩0' 이 들어오기 전까지 buffer에 저장한다. 표준입력이 아닐 경우, file_read 함수를 통해 file을 read한다. 그리고 read를 성공한 size만큼 return 한다.

13. src/userprog/syscall.c/ int sys_write(int fd, const void *buffer, unsigned size)

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

{
  int i;
  for(int i=0;i<size;i++)
  {
    if(!isAddressValid(buffer))
    {
      sys_exit(-1);
    }
  }
  int write_size = 0;
  struct file *f;

  int current_fd=thread_current()->fd_count;

  if (fd < 1 || fd > current_fd)
  {
    sys_exit (-1);
  }

  lock_acquire(&lock_file);

  if (fd == 1)
  {
    putbuf(buffer, size);
    write_size = 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(&lock_file);

  return write_size;
}
```

- file의 data를 size만큼 write하는 system call 함수이다.
- sys_read 함수와 동일하게 size 만큼 read 할 buffer 가 user 영역인지 확인한다. 그리고 write 함수일 경우에는 fd가 음수이거나 descriptor table에 해당 file이 없는 경우뿐만 아니라 write할 수 없는 표준 입력인 경우도 배제해야 한다. 따라서 fd값이 1보다 작거나 fd_count보다 큰 경우 exit 한다.
- 위 case가 아닌 경우, write가 가능한 경우이므로 write를 진행한다. 역시 다른 process의 접근을 막기위해 lock을 acquire한다.
- 그리고 fd가 1인 경우 표준 출력이므로 화면에 buffer값을 출력한다. 표준 출력이 아닌 경우 fd_table에서 file을 가져와 fille_write 함수를 통해 write한다. write가 완료된 경우 write 한 size를 return한다.

14. src/userprog/syscall.c/ void sys_seek(int fd, unsigned position)

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

  if(fd < thread_current()->fd_count)
  {
    f = thread_current()->fd_table[fd];
  }
  else
  {
    f=NULL;
  }

  if (f != NULL)
  {
```

```
    file_seek(f, position);
  }
}
```

- open 된 file의 위치를 이동하는 system call 함수이다.
- fd값이 file description table에 존재하는 index인지 판단한 후 file_seek 함수를 통해 해당 file의 위치를 이동한다.

15. src/userprog/syscall.c/ unsigned sys_tell(int fd)

```
unsigned sys_tell(int fd)

{
  struct file *f;

  if(fd < thread_current()->fd_count)
  {
    f = thread_current()->fd_table[fd];
  }
  else
  {
    f=NULL;
  }

  if (f != NULL)
  {
    return file_tell(f);
  }
  return 0;
}
```

- fd값에 해당하는 file의 위치를 return하는 system call 함수이다.
- fd값이 file description table에 존재하는 index인지 판단한 후 file_tell 함수를 통해 해당 file의 위치를 return한다.

16. src/userprog/syscall.c/ void sys_close(int fd)

```
void sys_close(int fd)

{
  struct file *f;
```

```
  if(fd < thread_current()->fd_count)
  {
    f = thread_current()->fd_table[fd];
  }
  else
  {
    f=NULL;
  }

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

- file을 close하는 system call 함수이다.
- fd값이 file description table에 존재하는 index인지 판단한 후 file_close 함수를 호출하여 file을 close한다. 이 때 sys_open에서 write 기능을 deny했으므로, file_close 함수 안에서 write 기능을 다시 allow한다 **(1-4. Denying writes to executables).**
- 마지막으로 file descriptor table에서 해당 file을 제거한다.

17. src/userprog/syscall.c/ static void syscall_handler(struct intr_frame *f)

```
static void

syscall_handler(struct intr_frame *f)
{
  if(!isAddressValid(f->esp))
  {
    sys_exit(-1);
  }

  int argv[3];
  switch (*(uint32_t *)(f->esp))
  {
  case SYS_HALT:
    sys_halt();
    break;
  case SYS_EXIT:
    get_argument(f->esp + 4, &argv[0], 1);
    sys_exit((int)argv[0]);
```

```
    break;
  case SYS_EXEC:
    get_argument(f->esp + 4, &argv[0], 1);
    f->eax = sys_exec((const char *)argv[0]);
    break;
  case SYS_WAIT:
    get_argument(f->esp + 4, &argv[0], 1);
    f->eax = sys_wait((pid_t)argv[0]);
    break;
  case SYS_CREATE:
    get_argument(f->esp + 4, &argv[0], 2);
    f->eax = sys_create((const char *)argv[0], (unsigned)argv[1]);
    break;
  case SYS_REMOVE:
    get_argument(f->esp + 4, &argv[0], 1);
    f->eax = sys_remove((const char *)argv[0]);
    break;
  case SYS_OPEN:
    get_argument(f->esp + 4, &argv[0], 1);
    f->eax = sys_open((const char *)argv[0]);
    break;
  case SYS_FILESIZE:
    get_argument(f->esp + 4, &argv[0], 1);
    f->eax = sys_filesize(argv[0]);
    break;
  case SYS_READ:
    get_argument(f->esp + 4, &argv[0], 3);
    f->eax = sys_read((int)argv[0], (void *)argv[1], (unsigned)argv[2]);
    break;
  case SYS_WRITE:
    get_argument(f->esp + 4, &argv[0], 3);
    f->eax = sys_write((int)argv[0], (const void *)argv[1],
(unsigned)argv[2]);
    break;
  case SYS_SEEK:
    get_argument(f->esp + 4, &argv[0], 2);
    sys_seek(argv[0], (unsigned)argv[1]);
    break;
  case SYS_TELL:
    get_argument(f->esp + 4, &argv[0], 1);
    f->eax = sys_tell(argv[0]);
    break;
  case SYS_CLOSE:
    get_argument(f->esp + 4, &argv[0], 1);
    sys_close(argv[0]);
    break;
  default:
    sys_exit(-1);
```

```
  }
}
```

- stack pointer가 가리키는 주소가 user 영역인지 확인한다. User 영역이 아닐 시 sys_exti(-1)을 호출한다.
- esp에 담겨있는 값을 확인한 뒤 이를 swtich-case문으로 위에서 구현한 함수들을 호출하는 형태의 syscall_handler 함수를 구현하였다.
- 각 system call에서 호출해야 함수가 호출되고, 주어진 경우가 아닌 경우 sys_exit(-1)을 호출하여 예외처리를 하도록 switch case 구문을 구현하였다.
- 그리고 각 case에서 필요한 argument를 argv[0]로부터 get_argument 함수를 통해 얻을 수 있도록 하였다. 필요한 argument의 개수를 지정해줄 수 있다.

1-4. Denying writes to executables

Algorithm

- 실행 중인 file에서 writer가 file 내 데이터를 변경한다면 의도치 않은 결과가 유도될 수 있으므로, 실행 중인 file에 write하는 것을 방지한다.
- 이를 위해 구현되어 있는 file_deny_write 함수와 file_allow_write 함수를 사용할 수 있다.
- File을 실행하기에 앞서 file_deny_write 함수를 호출해주고, file을 close할 때 file_allow_write 함수를 호출해주는 방식으로 간단히 구현 가능하다.

1. **src/filesys/file.c/file_allow_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);
    }
}
```

- input argument로 들어오는 file struct의 deny_write (write 권한에 대한 변수)에 접근하여 이 값이 true라면 false로 바꿔주고, inode 수준의 함수인 inode_allow_write() 함수를 호출함으로써 해당 file에 write를 허용해준다.

2. **src/filesys/file.c/ 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 (write 권한에 대한 변수)에 접근하여 이 값이 false라면 true로 바꿔주고, inode 수준의 함수인 inode_deny_write() 함수를 호출함으로써 해당 file에 write를 허용하지 않게 만든다.

Function Implementation

1. **src/userprog/process.c/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 = filesys_open (file_name);
```

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

  t->running_file = file;
  file_deny_write(file);

.
.
.
}
```

- 베이스라인 코드에서 load 함수 내에서 input argument인 file_name을 가지고 filesys_open 함수를 호출해주고 정상적으로 열었는지를 체크해주는데, 이를 체크하고 나서 file_deny_write(file) 함수를 호출하여 쓰기 권한을 해제해준다.
- file_close 함수에는 file_allow_write 함수를 호출하게끔 디자인되어 있어서 별도의 구현이 필요하지 않다.

2. src/userprog/syscall.c/int sys_open(const char *file)

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

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

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

  if (!strcmp(thread_current()->name, file))
  {
```

```
    file_deny_write(f);
  }

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

- rox test에서 발견한 일종의 예외처리인데, 앞서 system call에서 구현했던 sys_open 함수에서 file과 현재 실행되고 있는 thread의 name이 같다면 현재 해당 file을 실행중이라는 것을 의미한다. 따라서 해당 file에 대해 file_deny_write 함수를 호출하여 write 권한을 해제해주었다.

2. Discussion

2-1. 'multi-oom' test

- 해당 test는 여러 process가 동시에 메모리를 요청할 때 시스템이 어떻게 반응하는지 확인하는test이다. 이러한 상황에서 올바르게 메모리를 할당하고 관리하는지를 확인한다.
- 따라서 메모리 누수를 먼저 확인해 야한다. 기본적으로 메모리를 할당한 경우 반드시 user program이 종료되기 전 할당을 해제해줘야 한다. palloc_get_page를 사용한 경우 반드시 palloc_free_page 함수를 호출해 메모리를 할당해제 해 주었다.

```
void
process_exit (void)
{
…
  int i;
  for(i = 2; i < cur->fd_count; i++)
  {
    sys_close(i);
  }

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

- 그리고 process가 exit하기 전에 모든 fd를 sys_close 함수를 통해 할당 해제하고, file descriptor table도 할당 해제를 한 후 exit하여 메모리 누수를 방지하였다.

- 해당 test는 여러 process가 동시에 메모리를 요청하므로 적절한 할당 해제와 더불어 Concurrency Issue를 해결하기 위해 Synchronization 기능이 완벽하게 구현되어 있어야 한다. 이 test를 통과하기 전에는 syscall.h와 process.h 각각 lock를 하나씩 선언하여 사용하였다. Synchronization 기능을 향상하기 위해 process.h에 syscall.h를 include 하여 하나의 lock으로 sync를 맞추었다.

```
tid_t
process_execute (const char *file_name)
{
…
  struct list_elem *e;
  struct thread* child;
  // 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 process_wait(tid);
  }
…
}
```

- 또한 child process가 parent process 보다 먼저 dying 하는 경우도 처리를 해 주어야 한다. 이 경우 child list에 있는 모든 child process 의 exit status를 확인해 child process가 종료되지 않았을 경우에는 process_wait를 호출해 child process가 종료될 때까지 wait을 해 주어야한다.(?)

2-2. ‘rox'test

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{

…

  t->running_file = file;
  file_deny_write(file);

…

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

- rox test는 running file에 write가 되지 않아야 통과를 할 수 있다. 따라서 load 함수에서 running file에 대해 file_deny_write 함수로 write 권한을 해제하였다. 그리고 load 종료시 file_close함수를 부르면 다시 write가 allow된다. Load 함수가 종료될 때 가 아니라 해당 process가 exit될때 running file의 write를 allow 해야하므로

```
void
process_exit (void)
{

…

  file_close(cur->running_file);

…

}
```

Process_exit 함수에서 running file에 대해 file_close 함수를 불러 write를 allow한다.

- 그리고 우리가 구현한 sys_open 함수에서도 running file에 write를 할 수 있다.

```
int sys_open(const char *file)
{

…

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

…

}
```

- open하려는 file running 상태라면, write를 허용하면 안되므로 thread_current의 name과 file을 비교해 일치할 경우 해당 file은 write 권한을 해제해 주었다.