Assn2 Final Report

Pintos Assn2 Final Report

Team 37
20200229 김경민, 20200423 성치호

구현

Argument Passing

디자인 레포트에서 제안한 구현 방식에서 벗어나지 않으나 디자인 레포트에서는 세부적인 사항들을 포함하지 않아 이를 바탕으로 구현하는 과정에서 디자인 레포트에서는 언급하지 않은 세부적인 사항이 많이 추가되었다. 아래 사항들이 있다.

- argument string, argument 길이 array들을 저장하기 위해 page allocation을 사용.
- `process_execute`, `start_process` 에서만 구현 사항을 추가하는 기존 계획 --> 예상보다 구현의 복잡도와 높아 Argument Passing 기능을 여러 함수에 걸쳐 수행할 수 있게 함. `process_execute`, `start_process`, `parse_args`, `setup_args_stack` 에 걸쳐서 작동.
  - `parse_args`, `setup_args_stack` 함수 추가
- 기존 계획: `process_execute` 에서 받은 `file_name` 을 parsing하여 앞 부분을 스레드 생성 이름으로, 뒷 부분을 실행 함수의 argument로 `thread_create` 에 넘겨주기
  --> `process_execute` 에서 받은 `file_name` 을 두 개로 복사한 뒤 하나를 파싱하여 앞부분을 스레드 생성 이름으로 사용하고 다른 복사본을 argument로 통째로 넘겨준다.
  - 기존 계획에서는 `load` 에서 파일 이름을 사용하는 것을 고려하지 못하여서 계획을 수정함.

`parse_args` in `userprog/process`

```
static void
parse_args (char *cmd_line_str, char **argv, size_t *argv_len, uint32_t *argc)
{
  char *arg, *save_ptr;

  for (arg = strtok_r (cmd_line_str, " ", &save_ptr); arg != NULL;
    arg = strtok_r (NULL, " ", &save_ptr))
  {
    argv_len[*argc] = strlen (arg) + 1;
    argv[*argc] = arg;
    (*argc)++;
  }
}
```

> `cmd_line_str` 을 Space를 기준으로 파싱하여 각 문자열의 시작 주소를 `argv` 배열에 집어넣고 각 문자열의 길이를 `argv_len` 배열에, 문자열의 개수를 `argc` 에 저장해 전달하는 함수.

`cmd_line_str` 을 Space를 기준으로 끊어가면서 끝에 도달할 때까지 순회하며 다음을 수행한다.

- `argv_len[*argc]` 에 현재 끊긴 문자열의 길이(`strlen` 으로 측정) + 1을 추가한다.
  - 각 문자열 끝의 `NULL` 을 고려한 수치이다.
- `argv[*argc]` 에는 현재 끊긴 문자열의 시작 주소를 집어 넣는다.
- `*argc` 를 1 증가시킨다.
  - `argc` 는 순회 중에는 배열의 index로서 사용되며 순회 완료시에는 각 배열의 길이, 문자열의 총 개수를 의미하게 된다.

인수로 받는 `argv`, `argv_len` 등은 올바르게 공간(페이지)가 할당되어 사용할 수 있음을 기대한다.
해당 함수는 인수의 `cmd_line_str` 은 변형하며 맨 앞 단어만을 담은 문자열처럼 작동하게 된다.

- 실행할 파일 이름만을 의미하게 된다.

- 단어 끝 `\0`에 의해 뒤 문자들은 읽지 않고 멈춤

`setup_args_stack` in `userprog/process`

> Argument Passing의 핵심 함수로 80x86 Calling Convention에 맞추어 주어진 argument 정보들을 이용해 stack에 argument 정보를 쌓는 함수

80x86 Calling Convention 예시

`/bin/ls -l foo bar` 명령일 시 해당 함수 리턴시 stack이 다음 꼴을 나타내도록 해야 함.

```
Example cmd line: "/bin/ls -l foo bar"
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 *
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 (*) ()   <- esp
```

```
static void
setup_args_stack (char **argv, size_t *argv_len, uint32_t argc,
  void **esp)
{
```

함수 입력 매개변수
`argv` : file name을 포함한 argument string들의 시작 주소를 담은 배열
`argv_len` : file name을 포함한 argument string들의 길이를 담은 배열
`argc` : parsing 후 file name을 포함한 argument string의 개수
`esp` : argument 정보들을 쌓을 스택 point의 주소를 가리키는 포인터

```
  /* argv string */
  char *ptr_argv = (char*) *esp;
  for (int i = argc - 1; i >= 0; i--)
  {
    ptr_argv = ptr_argv - (char*) (argv_len[i]);
    strlcpy (ptr_argv, (const char*) argv[i], (size_t) (argv_len[i]));
  }
```

`ptr_argv`는 실시간 stack point로 `*esp`로 설정해 둔다. `char *` 자료형이므로 1바이트씩 증가한다.
`argv`와 `argv_len`를 뒤에서부터 앞으로 순회하며 다음을 수행한다. (두 배열의 길이는 `argc`로 같다)

- `ptr_argv`에서 `argv_len[i]` 만큼 빼주어 `argv[i]` 문자열이 들어갈 수 있는 공간을 확보한다.
- [`ptr_argv`, `ptr_argv + argv_len[i]`)에 `argv[i]` 문자열을 deep copy해준다.
  이로써 right->left로 argument 문자열을 스택에 모두 집어넣었다. (Example 기준 `0xbffffffc` ~ `0xbfffffed`)

```
  /* Word Align */
  ptr_argv = (char *)(((uint32_t) ptr_argv) - ((uint32_t) ptr_argv) % 4);
```

`char`는 한 바이트이기에 위에서 집어넣은 Argument 문자열의 길이는 모두 제각각이다. 주소는 보통 4바이트 기준으로 읽어들이고 쓰므로 이를 위해 4바이트 단위로 나누어 떨어지도록 패딩을 추가해주어야 한다.

이를 위해 `ptr_argv` 스택 포인트를 4로 나눈 나머지를 빼주어 4로 나누어 떨어지도록 해준다. (Example 기준 `0xbffffffec`)

```
ptr_argv -= 4;
char** ptr_argv_addr = (char **) ptr_argv;

*((uint32_t *) ptr_argv_addr) = 0;
ptr_argv_addr--;

/* argv string pointer */
char* argv_addr_iter_ptr = (char*) *esp;
for(int i = argc-1; i >= 0; i--)
{
  argv_addr_iter_ptr -= argv_len[i];
  *ptr_argv_addr = (char *) argv_addr_iter_ptr;
  ptr_argv_addr--;
}
```

다음 4바이트를 0으로 비워준다. 이는 마지막 argument 다음에 대한 주소이다. 즉 argument 주소들의 끝을 나타내기 위해 존재하는 공백이라 생각하면 된다. (Example 기준 `0xbffffffe8`)
`char*` 자료형이던 `ptr_argv` 를 `char **` 로 형변환하여 `ptr_argv_addr` 에 저장한다.

- `ptr_argv_addr` 은 주소를 가리키는 포인터이기에 4바이트 단위로 증감한다.
  `argv_len` 을 역순으로 순회하며 기존 스택 포인트에서 문자열 길이만큼 빼가며 각 argument 문자열의 시작 주소를 얻어 스택에 right -> left 순서로 집어 넣는다. (Example 기준 `0xbffffffe4` ~ `0xbffffffd8`)

```
/* argv address */
*ptr_argv_addr = (char**) (ptr_argv_addr + 1);
ptr_argv_addr--;
```

argument 문자열 주소들이 스택 어디부터 시작하는지를 스택에 집어 넣는다. 즉 현재 스택 point의 바로 이전 주소를 집어넣으면 된다. (Example 기준 `0xbffffffd4`)

```
/* argument count */
*(uint32_t*) ptr_argv_addr = argc;
ptr_argv_addr--;
```

스택에 `argc`, 즉 argument들의 개수를 집어넣는다. (Example 기준 `0xbffffffd0`)

```
*ptr_argv_addr = 0;

void* ori_if_esp = *esp;
*esp = (void*) ptr_argv_addr;
```

스택에 마지막으로 0으로 집어 넣고 현재 스택 포인터 주소를 담고 있는 `ptr_argv_addr` 가 담고 있는 주소를 `esp` 에 넣는다. 즉 위에서 쌓은 스택의 stack point가 인수의 `esp` 가 가리키는 값이 된다.

`start_process` in `userprog/process` 변경점

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

  /* argv: process arguments str point array */
  /* argv_len: process arguments str length array */
  /* argc: process arguments count */
  char **argv;
```

```
  size_t *argv_len;
  uint32_t argc = 0;
  success = true;
  t = thread_current();

  ...

  argv = palloc_get_page (0);
  if (argv == NULL)
    success = false;

  argv_len = palloc_get_page (0);
  if (argv_len == NULL)
    success = false;

  /* file_name stops at the null character only at the end of the pure file name */
  parse_args (file_name, argv, argv_len, &argc);
  success = load (file_name, &if_.eip, &if_.esp) && success;
  if (success)
    setup_args_stack (argv, argv_len, argc, &if_.esp);
  else
    t->process_ptr->pid = PID_ERROR;

  palloc_free_page (file_name);
  palloc_free_page (argv);
  palloc_free_page (argv_len);

  ...

  if (!success)
    thread_exit ();

  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

`start_process`는 유저 프로그램 프로세스 실행을 위해 `process_execute`에서 스레드 생성시 넘겨주는 함수로 스레드가 실행하는 함수이다. 실행해야 할 명령어 문자열 전체를 매개변수 `file_name_`로 받는다.
argument 문자열들의 시작 주소를 담을 `argv`, 그 문자열들의 각 길이를 담을 `argv_len`를 선언하고 해당 배열들의 공간을 위해 `palloc_get_page`를 통해 각각 페이지를 할당해준다.
만약 할당 실패시 `success`를 `false`로 변경한다.
`parse_args`를 통해 `file_name`을 파싱하여 `argv`, `argv_len`, `argc`에 각각 argument 문자열 시작 주소들, 각 argument 길이, argument 개수를 집어 넣는다.

- `parse_args`에 의해 `file_name`은 파일 명만을 담은 문자열처럼 작동하게 된다.
  이후 `file_name`을 로드 이후 성공 여부를 `success`에 저장한다.
  만약 그동안 모든 작업을 성공했다면(`success`가 true) `setup_args_stack`을 통해 `if_.esp`를 stack point로 생각하여 80x86 Calling Convention에 맞추어 argument 정보를 스택에 쌓는다.
  만약 그동안 한 번이라도 실패한 작업이 있다면(`success`가 false) 해당 프로세스의 `pid`를 `PID_ERROR`로 설정한다.
- 해당 내용은 뒤의 SystemCall Handler에서 자세히 설명하겠다.

로드 작업, 프로그램 실행시 사용할 스택 설정이 완료되었으므로 `file_name`, `argv`, `argv_len`을 모두 할당 해제해준다.

`process_execute` in `userprog/process` 변경점

`process_execute` 내 변경 사항 중 Argument Passing 구현을 위해 변경한 내용만을 포함하여 설명하겠다.

```
tid_t
process_execute (const char *file_name)
{
  /* Command: File Name + Arguments */
  char *full_cmd_line_copy;
```

```
  /* Just File Name */
  char *file_name_copy;
  char *save_ptr;

  ...

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

  strlcpy (full_cmd_line_copy, file_name, PGSIZE);
  strlcpy (file_name_copy, file_name, PGSIZE);

  strtok_r (file_name_copy, " ", &save_ptr);

  ...
```

`full_cmd_line_copy`은 입력 받은 `file_name`를 통째로 복사해 저장할 예정이며, `file_name_copy`는 `file_name` 중 파일 이름만을(파싱 후 맨 앞 단어) 저장할 예정이다.

문자열을 저장할 수 있게 `palloc_get_page`를 이용해 `full_cmd_line_copy`, `file_name_copy`에 각각 페이지를 할당한다.

만약 페이지 할당 실패시 `TID_ERROR`를 리턴해 프로세스 실행에 실패했음을 나타낸다.

이후 `strlcpy`를 이용해 `full_cmd_line_copy`, `file_name_copy`에 각각 `file_name`을 deep copy한다.

이후 `strtok_r`을 이용해 `file_name_copy`를 첫 공백을 기준으로 나눈다. 이 때 공백은 `\0`로 대체되어 `file_name_copy`는 맨 앞 단어만으로 이루어진 문자열처럼 작동하게 된다.
즉 맨 앞 단어인 파일명이다.

```
  ...

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create_with_pcb (file_name_copy, PRI_DEFAULT, p, start_process,
    full_cmd_line_copy);
  palloc_free_page (file_name_copy);
  if (tid == TID_ERROR)
  {
    palloc_free_page (full_cmd_line_copy);
    palloc_free_page (p);
    return TID_ERROR;
  }

  ...

  return tid;
}
```

`thread_create_with_pcb`는 `thread_create`의 확장된 형태로 뒤에서 자세히 후술하겠다.
`thread_create_with_pcb`에 스레드 이름으로 `file_name_copy`를 넘겨주어 실행하는 파일 명으로 스레드 명을 가지게 한다. 또한 `start_process`의 인수로써 `full_cmd_line_copy` 즉 `file_name` 통 복사본을 넘겨준다.

`thread_create_with_pcb`가 완료되면 스레드 생성은 완료되었으므로 스레드 이름으로 사용된 `file_name_copy`의 페이지를 할당 해제한다.

만약 `thread_crate_with_pcb`가 실패하였다면 `start_process`의 인수인 `full_cmd_line_copy` 또한 사용될 일이 없으므로 할당 해제해준다.

위의 작업들을 통해 Argument Passing을 구현하여 올바르게 파일을 오픈하여 유저 프로그램을 수행하고 유저 프로그램 실행시 명령어의 argument들을 올바르게 넘겨줄 수 있게 되었다.

System Call - Basement

기존 디자인 레포트의 Process Control Block에 대한 struct의 명세에서 각 프로세스의 스레드 id를 나타내는 `tid`, 현재 실행하고 있는 프로그램 파일을 나타내는 `file_exec`이 추가되었다.
또한 기존 디자인 레포트에서는 PCB 초기화를 위해 `pcb_init` 함수 하나만을 계획하였으나 Process/System Call 시스템 전반에 사용되는 전역 변수(`pid_lock`, `file_lock`)을 초기화하고 Main Program의 스레드 생성, 초기화 과정이 다른 여타 스레드와 다르기에 별도의 프로세스 초기화 함수가 필요하여 이를 위한 함수 `procoess_init`을 추가하였다.
마지막으로 기존 계획에는 기존의 스레드 생성 함수인 `thread_create`에 프로세스 관련 내용을 추가하기로 하였으나 이렇게 되면 커널 스레드, 유저 스레드를 구분할 수 없고 기존 `thread_create`를 사용 중인 모든 곳을 변경해야하기에 유저 프로세스의 스레드를 생성할 때만 사용하는 별도의 함수 `thread_create_with_pcb`를 추가하였다.
이외에 계획과 동일한 기능을 하되 사소하게 변수 명 및 함수 명이 변경되었다.

`struct process` in `userprog/process`

`process` 구조체는 Process Control Block을 표현하는 구조체로 스레드의 정보를 표현하고 스레드를 관리하는데 사용하는 `struct thread`와 동일한 역할을 프로세스 관리 및 시스템 콜에서 수행한다.
`process`는 `exec`, `wait`, `exit`과 같은 프로세스에 직접 연관 있는 시스템 콜 및 프로세스 간에 관여하는 시스템 콜을 쉽게 통제하고 프로세스별 file system call 사용 현황 관리 및 관련 synchronization 문제들을 쉽게 해결하기 위해 존재한다.

`process`는 프로세스 하나에 일대일 대응되는 구조체이다. 또한 Pintos에서는 현재 프로세스는 스레드 하나만을 가진다.

```
// in userprog/process.h
typedef int pid_t;
```

`tid_t`와 유사하게 프로세스의 identifier 자료형이다.

```
struct process
{
  pid_t pid;
  tid_t tid;
  int exit_code;
  struct semaphore exit_code_sema;
  struct list children;
  struct list_elem elem;
  struct semaphore exec_load_sema;
  struct file *file_exec;
  struct fd_table_entry fd_table[OPEN_MAX];
};
```

`struct process`의 각 멤버 변수는 다음과 같다.

멤버 변수 이름	자료형	설명
`pid`	`pid_t`	프로세스를 구별하는 Identifier
`tid`	`tid_t`	해당 프로세스를 실행 중인 스레드의 id
`exit_code`	`int`	프로세스 exit시 exit_code
`exit_code_sema`	`struct semaphore`	부모 프로세스의 wait을 위해 사용되는 semaphore
`children`	`struct list`	자식 프로세스 리스트
`elem`	`stuct list_elem`	자식 프로세스 리스트를 구성하기 위한 `list_elem`
`exec_load_sema`	`struct semaphore`	부모 프로세스에게 로드 여부를 알려주기 위한 semaphore
`file_exec`	`struct file*`	프로세스가 현재 실행중인 프로그램 파일
`fd_table`	`struct fd_table_entry[OPEN_MAX]`	프로세스마다 독립적인 file descriptor의 배열

`exit_code_sema` 값이 의미하는 바

- 0: 아직 해당 프로세스 exit 안함. (+ 부모가 exit 확인시 해당 상태 직후 바로 `process` 는 할당 해제됨.)
- 1: 해당 프로세스 exit됨. 하지만 부모는 아직 알지 못함.

`exec_load_sema` 값이 의미하는 바

- 0: 아직 해당 프로세스 load 완료 안 됨. load 완료 후 부모가 확인함.
- 1: 해당 프로세스 load 완료 후 부모가 알기 전의 상태

`process` 를 위한 `struct thread` 변화

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */

    ...

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct process* process_ptr;
#endif

    ...
  };
```

스레드 정보를 나타내는 `struct thread` 멤버에 `struct process* process_ptr` 를 추가한다.
이는 해당 스레드의 프로세스가 무엇인지 표현하는 변수로 스레드의 프로세스의 `struct process` 를 가리키는 포인터이다.
구조체 자체를 넣는 것이 아닌 포인터를 사용한 이유는 `struct thread` 와 `struct process` 의 lifecycle이 다르기 때문이다.

- exit code 전달을 위해 `wait` , `exit` 등에서 `thread` 는 free지만 `process` 는 존재하는 상황들 발생 가능.

`allocate_pid` in `userprog/process.c`

```
static struct lock pid_lock;
```

프로세스 identifier인 pid가 unique하게 보장하기 위해 프로세스 id 생성 함수인 `allocate_pid` 에서 사용하는 lock

```
static pid_t
allocate_pid (void)
{
  static pid_t next_pid = 1;
  pid_t pid;

  lock_acquire (&pid_lock);
  pid = next_pid++;
  lock_release (&pid_lock);

  return pid;
}
```

> 각 프로세스에게 unique한 id를 생성해 반환하는 함수

원리는 `allocate_tid` 와 완전히 동일하다. `next_pid` 를 반환하며 1부터 1씩 증가시킨다.
해당 함수 실행시 항상 `next_pid` 는 1씩 증가하며 `next_pid` 변경 및 pid 할당은 `pid_lock` 에 의해 한 스레드만 수행할 수 있으므로 id가 unique함이 보장된다.

`init_process` in `userprog/process.c`

```
void
init_process (struct process *p)
{
  memset (p, 0, sizeof *p);
  p->pid = allocate_pid ();
  sema_init (&p->exit_code_sema, 0);
  sema_init (&p->exec_load_sema, 0);
  list_init (&p->children);

  /* Initialize fd table */
  for (size_t i = 0; i < OPEN_MAX; i++)
  {
    p->fd_table[i].file = NULL;
    p->fd_table[i].in_use = false;
    p->fd_table[i].type = FILETYPE_FILE;
  }
  p->fd_table[0].in_use = true;
  p->fd_table[0].type = FILETYPE_STDIN;
  p->fd_table[1].in_use = true;
  p->fd_table[1].type = FILETYPE_STDOUT;
}
```

> 주어진 `process` 구조체 `p`를 초기화하는 함수

`allocate_pid`를 통해 해당 `process`에 대한 id를 받아 `p->pid`에 넣는다. `sema_init`을 통해 `exit_code_sema`, `exec_load_sema`를 0으로 초기화한다. 또한 `children`을 `list_init`으로 초기화한다. 그 후 후술할 file descriptor table의 0번과 1번 인덱스를 사전 정의된 대로 채워 초기화해준다.

해당 함수는 주어진 `p`가 올바르게 페이지를 할당 받았을 것이라고 가정하고 작동한다.

`process_init` in `userprog/process.c`

```
void
process_init(void)
{
  struct process *p;
  struct thread *t;

  // Main Thread
  t = thread_current();

  lock_init(&pid_lock);
  lock_init(&file_lock);

  p = palloc_get_page (PAL_ZERO);
  if(p == NULL)
  {
    palloc_free_page(p);
    PANIC("Main Process Init Fail");
  }

  init_process(p);
  t->process_ptr = p;
  p->tid = t->tid;
}
```

> `process` 시스템 초기화 및 Main Process의 `process`를 초기화하는 함수

`thread_current()` 결과가 메인 프로세스이기를 기대한다.
`lock_init`을 이용하여 Process id 할당에 사용되는 `pid_lock`, 각종 파일 시스템 콜 critical section에 사용되는 `file_lock`을 초기화한다.
현재 스레드(프로세스)(Main 스레드/프로세스)의 `process` 구조체를 위한 공간을 `palloc_get_page`로 할당 받는다. 만약 할

당 실패시 패닉한다. 메인 프로세스의 프로세스 정보를 초기화하지 못하면 다른 작업들을 정상적으로 수행할 수 없기 때문이다.

할당에 성공하면 `init_process`를 통해 `process` 구조체를 초기화한다. 또한 현재 스레드 `thread`의 `process_ptr`이 새로 생성한 `process` 구조체를 가르키도록 한다. 새로 생성한 `process` `p`의 `tid`를 현재 스레드의 `tid`로 설정하여 스레드, 프로세스간 매핑 관계를 설정한다.

`thread_create_with_pcb` in `threads/thread`

```
tid_t
thread_create_with_pcb (const char *name, int priority, struct process* p_ptr,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

  t->process_ptr = p_ptr;
  t->process_ptr->tid=tid;

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;

  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame (t, sizeof *sf);
  sf->eip = switch_entry;
  sf->ebp = 0;

  /* Add to run queue. */
  thread_unblock (t);

  return tid;
}
```

> `process` 연관 정보 설정 기능을 추가한 `thread_create` 확장 함수로 user process thread 생성시 사용한다.

기존 스레드 생성 함수인 `thread_create`에 다음 기능만을 추가한 함수이다.

- 매개변수로 `struct process* p_ptr`을 추가로 받는다. `p_ptr`은 해당 스레드의 프로세스의 `process` 구조체 주소이다.
  - 이를 통해 해당 스레드의 프로세스의 `process` 구조체의 변수들을 보거나 변경할 수 있다.

- `t->process_ptr = p_ptr;` 를 통해 새로 생성한 `thread` 의 `process_ptr` 을 넘겨준 `p_ptr` 로 설정하게 한다. 즉 `p_ptr` 의 `process` 와 새로 생성한 스레드가 관계를 형성하는 것이다.
- `t->process_ptr->tid=tid;` 를 통해 해당 스레드와 연결된 `process` 의 `tid` 를 현재 스레드의 `tid` 로 설정하여 양방향 관계를 형성한다.

System Call - System Call Handler

기존 디자인에서는 시스템 콜에 유저가 넘겨준 포인터가 유효한지 검증하는 로직에 대한 계획이 없었다. 하지만 유저는 신뢰하면 안되기에 유저가 시스템 콜에 넘겨준 주소를 검증하는 로직들을 추가하였다.
또한 기존 디자인에서는 시스템 콜의 argument 처리에 관한 계획이 부족하여 본 구현에서는 interrupt frame의 esp이후에 4바이트씩 끊어 읽어 Argument를 얻는 `get_arg` 함수를 추가하였다.

Virtual Memory User Space

System call 함수들을 구현할 때 가상 메모리 위의 User Space에 있는 주소에만 읽고 쓰는 것이 가능하도록 핸들링을 해줘야 했다. Pintos 문서에는 이를 구현하기 위한 두 가지 방법이 제시되어 있는데, 이번 프로젝트에선 해당 위치가 User space인지 Kernel space인지만 검사한 뒤 User space 내에서의 잘못된 참조는 Page fault handler가 관리하도록 구현하였다.

```
/* Only checks whether its in the user space */
bool
check_ptr_in_user_space (const void *ptr)
{
  return ptr < PHYS_BASE;
}
```

`exception.c` 의 Page fault handler에서도 기존과 다른 동작을 하도록 수정해줘야 했다. 유저가 page fault를 일으켰을 경우 `exit(-1)` 로 프로세스를 종료하고, 커널이 일으켰을 경우 `-1` 을 반환한다.

```
static void
page_fault (struct intr_frame *f)
{
  ...
  /* Kernel caused page fault by accessing user memory */
  if(!user && check_ptr_in_user_space(fault_addr))
  {
   f->eip = (void *)f->eax;
   f->eax = -1;
   return;
  }
  /* User caused page fault */
  else sys_exit(-1);
  ...
}
```

해당 기능을 편리하게 구현하기 위해 User space에서 한 바이트씩 읽거나 쓰는 함수를 Pintos 문서에서 제시해주었다.

```
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
```

```
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
  int error_code;
  asm ("movl $1f, %0; movb %b2, %1; 1:"
       : "=&a" (error_code), "=m" (*udst) : "q" (byte));
  return error_code != -1;
}
```

다만 위 함수들로는 한 바이트씩밖에 읽고 쓸 수 없으므로 위의 함수들을 이용하여 임의 길이의 데이터를 읽을 수 있는 함수를 구현했다.

```
/* Reads NUM bytes at user address SRC, stores at DEST.
   Note that DEST is not a vmem address.
   Returns true if every byte copies are successful. */
static bool
get_user_bytes (void *dest, const void *src, size_t num)
{
  uint8_t *_dest = dest;
  const uint8_t *_src = src;
  for (size_t i = 0; i < num; i++)
  {
    if (!check_ptr_in_user_space (_src)) return false;
    int res = get_user (_src);
    if (res == -1) return false;
    *_dest = (uint8_t) res;
    _dest++;
    _src++;
  }
  return true;
}
```

임의 길이 write 함수는 이번 프로젝트에서 사용처가 없어 구현하지 않았다.

System Call Arguments

System call이 호출되었을 때 추가로 전달되는 인자들은 `Stack pointer + 4` 위치부터 4바이트 단위로 순서대로 배치되어 있다. 이를 가져오기 위해 원하는 만큼의 인자를 읽어 가져오는 `get_args` 함수를 구현했다.

```
/* Parse NUM arguments from sp + 4 to dest. */
static void
get_args (int *sp, int *dest, size_t num)
{
  for(size_t i = 0; i < num; i++)
  {
    int *src = sp + i + 1;
    if (!check_ptr_in_user_space (src)) sys_exit (-1);
    if (!get_user_bytes (dest + i, src, 4)) sys_exit (-1);
  }
}
```

전달되는 인자의 수는 호출되는 System call마다 다르므로, System call 핸들러가 호출되고 어떤 함수가 호출되었는지 알아낸 뒤 위의 함수를 통해 인자를 읽어와 실행해준다.

`syscall_handler` in `userprog/syscall`

```
static void
syscall_handler (struct intr_frame *f)
{
  int arg[4];
```

```
  if (!check_ptr_in_user_space (f->esp))
    sys_exit (-1);
  switch(*(uint32_t *) (f->esp))
  {
    case SYS_HALT:
      sys_halt ();
      break;
    case SYS_EXIT:
      get_args (f->esp, arg, 1);
      sys_exit (arg[0]);
      break;
    case SYS_EXEC:
      get_args (f->esp, arg, 1);
      f->eax = sys_exec ((const char *) arg[0]);
      break;
    case SYS_WAIT:
      get_args (f->esp, arg, 1);
      f->eax = sys_wait ((pid_t) arg[0]);
      break;
    case SYS_CREATE:
      get_args (f->esp, arg, 2);
      f->eax = sys_create ((const char *) arg[0], (unsigned) arg[1]);
      break;
    case SYS_REMOVE:
      get_args (f->esp, arg, 1);
      f->eax = sys_remove ((const char *) arg[0]);
      break;
    case SYS_OPEN:
      get_args (f->esp, arg, 1);
      f->eax = sys_open ((const char *) arg[0]);
      break;
    ...
    case SYS_FILESIZE:
      get_args (f->esp, arg, 1);
      f->eax = sys_filesize (arg[0]);
      break;
    case SYS_READ:
      get_args (f->esp, arg, 3);
      f->eax = sys_read (arg[0], (void *) arg[1], (unsigned) arg[2]);
      break;
    case SYS_WRITE:
      get_args (f->esp, arg, 3);
      f->eax = sys_write(arg[0], (const void *) arg[1], (unsigned) arg[2]);
      break;
    case SYS_SEEK:
      get_args (f->esp, arg, 2);
      sys_seek (arg[0], (unsigned) arg[1]);
      break;
    case SYS_TELL:
      get_args (f->esp, arg, 1);
      f->eax = sys_tell (arg[0]);
      break;
    case SYS_CLOSE:
      get_args (f->esp, arg, 1);
      sys_close (arg[0]);
      break;
    default:
      sys_exit (-1);
  }
}
```

> System call에 대한 interrupt handler 함수로 등록된 함수로 어떤 system call인지 판별 후 각 system call 맞는 함수를 호출 후 리턴 값을 설정하는 함수이다.

이번 프로젝트에 구현해야 할 시스템 콜은 모두 다음과 같다.

- `SYS_HALT`, `SYS_EXIT`, `SYS_EXEC`, `SYS_WAIT`, `SYS_CREATE`, `SYS_REMOVE`, `SYS_OPEN`, `SYS_FILESIZE`, `SYS_READ`, `SYS_WRITE`, `SYS_SEEK`, `SYS_TELL`, `SYS_CLOSE`
  시스템 콜의 argument 주소를 저장할 길이 4의 배열 `int arg[4]`를 선언한다.
- 다음 시스템 콜들은 호출 함수의 argument 개수가 4개를 넘지 않는다.
  인수로 받은 interrupt frame `f`의 `esp`가 올바른 주소인지(user space의 주소인지) `check_ptr_in_user_space`를 통해 검증 후 올바르지 않다면 `sys_exit (-1)`을 통해 해당 프로세스를 종료한다.
- OS는 유저를 믿으면 안되고 유저가 보낸 값들은 우선 의심해야 한다.
  `f->esp`에 담긴 값은 system call number로 해당 값을 기준으로 호출할 함수를 결정한다.

`get_args`를 통해 `f->esp`로부터 각 시스템 콜이 필요로 하는 개수의 argument들을 얻어 `arg`에 저장한다.
이후 각 argument 순서대로 각 시스템 콜 함수의 인수로 넣어 호출한다.

- argument들의 기본 자료형이 int로 되어 있으므로 각 함수 매개변수 자료형으로 형변환해주어야만 한다.
  만약 해당 시스템 콜이 리턴 값을 필요로 한다면 시스템 콜 함수의 반환 값을 `f->eax`에 집어 넣어 시스템 콜의 리턴 값을 설정해준다.

System Call - User Process Manipulation

`sys_halt` in `userprog/syscall`

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

> SYS_HALT 시스템 콜에 대응되는 함수. Pintos 자체를 종료시킨다.

`shutdown_power_off`를 호출하여 Pintos 자체를 종료시킨다.

`sys_exec` in `userprog/syscall`

```
static pid_t
sys_exec (const char *cmd_line)
{
  struct list_elem *elem;
  struct process *p;
  tid_t tid = process_execute (cmd_line);
  if (tid == TID_ERROR)
    return PID_ERROR;
  /* Last added child */
  elem = list_back (&thread_current ()->process_ptr->children);
  p = list_entry (elem, struct process, elem);
  return p->pid;
}
```

> SYS_EXEC 시스템 콜에 대응되는 함수

main thread에서 user program을 실행하듯이 `process_execute`를 통해 `cmd_line`을 실행하기 위한 스레드를 생성한다. 만일 스레드 생성 실패시 PID_ERROR(-1)을 반환한다.
생성 성공시 해당 시스템 콜을 호출한 스레드의 `process_ptr->children`의 맨 뒤의 `process`를 참조한다. 이는 `cmd_line`을 수행하기 위해 이번에 생성한 프로세스이다. 그리고 해당 프로세스의 id를 리턴한다.

```
tid_t
process_execute (const char *file_name)
{
  ...
```

```
  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create_with_pcb (file_name_copy, PRI_DEFAULT, p, start_process,
    full_cmd_line_copy);

  ...

  /* System call Exec Load Sync */
  sema_down(&(p->exec_load_sema));

  ...

  return tid;
}
```

유저 프로그램을 수행하는 스레드를 생성하는 `process_execute` 에서 `thread_create_with_pcb` 를 통해 `start_process` 를 수행하는 스레드를 생성한 이후 새로 생성된 스레드의 `start_process` 에서 `load` 가 정상적으로 완료되길 기다리기 위해 `sema_down(&(p->exec_load_sema));` 을 추가하였다. `start_process` 에서 load 완료시 뒤를 넘어가 리턴할 수 있게 된다.

```
static void
start_process (void *file_name_)
{

  ...

  parse_args (file_name, argv, argv_len, &argc);
  success = load (file_name, &if_.eip, &if_.esp) && success;

  ...

  /* System call Exec Load Sync */
  sema_up (&t->process_ptr->exec_load_sema);

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

  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

유저 프로그램을 수행하기 위해 유저 프로세스의 스레드가 수행하는 함수로 `start_process` 는 유저 프로그램을 로드 후 해당 프로그램으로 넘어가게 된다. 이 때 자식 프로세스의 `load` 이후 부모의 `sys_exec` 시스템콜에 대해 반환하기 위해 `load` 이후 `sema_up (&t->process_ptr->exec_load_sema);` 을 추가해주었다.

`sys_exit` in `userprog/syscall`

이번 프로젝트의 목표 중 하나인 **Process Termination Messages**에 대한 구현도 포함되어 있다.

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

  file_close (cur->process_ptr->file_exec);
  for (size_t i = 2; i < OPEN_MAX; i++)
  {
    if(cur->process_ptr->fd_table[i].in_use)
    {
      file_close (cur->process_ptr->fd_table[i].file);
```

```
      remove_fd (cur->process_ptr, i);
    }
  }
  sema_up (&(cur->process_ptr->exit_code_sema));
  thread_exit ();
  NOT_REACHED ();
}
```

> SYS_EXIT 시스템 콜에 대응되는 함수.

현재 `thread`의 이름, 인수로 받은 `status`를 이용해 형식에 맞추어 Process Termination Message를 출력한다.

- 시스템 콜 처리 중 각종 오류 발생시 `sys_exit(-1)`로 프로그램을 종료한다.
  입력 받은 `status`를 현재 스레드의 `process->exit_code`에 집어 넣어 이후 부모 프로세스가 볼 수 있도록 한다.
  `file_close`를 통해 현재 실행 중이었던 프로그램 파일을 close하고 `cur->process_ptr->fd_table`을 순회하며 해당 프로세스가 사용 중이였던 파일들도 닫은 뒤 그 descriptor도 할당 해제해준다.
  `cur->process_ptr->exit_code_sema`를 `sema_up`해주어 자식 프로세스가 exit하기를 wait하는 부모 프로세스가 이후 wait 절차를 진행할 수 있게 한다.
- 해당 자식의 `exit_code`를 얻을 수 있다.
  이후 `thread_exit`을 통해 해당 스레드에게 할당된 자원들을 해제하게 된다.
- `process`에 대한 자원(`process` 할당 페이지)은 부모의 `wait`에서 수행된다.

`sys_wait` in `userprog/syscall`

```
static int
sys_wait (pid_t pid)
{
  struct thread *cur = thread_current ();
  struct list* children = &cur->process_ptr->children;

  /* find pid process in children */
  for (struct list_elem *e = list_begin (children); e != list_end (children);
    e = list_next (e))
  {
    struct process *p = list_entry (e, struct process, elem);
    if (p->pid == pid)
    {
      /* Wait for child exit */
      sema_down (&p->exit_code_sema);
      list_remove (e);
      int exit_code = p->exit_code;
      palloc_free_page (p);
      return exit_code;
    }
  }
  return -1;
}
```

> SYS_WAIT 시스템 콜에 대응되는 함수.

현재 스레드의 `process_ptr->children`, 해당 스레드의 프로세스의 자식 프로세스를 순회하며 인수의 `pid`를 `process.pid`로 가지는 `process`인 `p`를 찾는다.
`sema_down (&p->exit_code_sema)`을 통해 `p` 프로세스가 `sys_exit`을 호출하여 (종료할 때까지) `sema_up`할 때까지 기다린다. 이후 자식 프로세스의 `sys_exit`을 통해 `sema_up`되면 `children`에서 해당 자식 프로세스의 `list_elem`을 제거하고 `p->exit_code`를 통해 마지막에 exit_code를 얻어 반환한다. 그 전에 exit된 해당 자식 프로세스의 `process`에 할당된 페이지를 해제해 준다.
만약 `process.pid`가 `pid`인 프로세스를 찾지 못하였다면 -1을 반환한다.

- 해당 프로세스는 이미 wait했거나 현재 프로세스의 자식 프로세스가 아니다.

System Call - File Manipulation

File manipulation과 관련된 System call의 대부분이 File descriptor 시스템을 필요로 한다.

File Descriptor in `userprog/process`

File descriptor는 프로세스마다 독립적이므로 `process.h` 에 모두 구현하였다.

```
struct fd_table_entry
{
  struct file *file;
  bool in_use;
  int type;
};

#define FILETYPE_FILE 0
#define FILETYPE_STDIN 1
#define FILETYPE_STDOUT 2
```

각 File descriptor 구성요소를 나타내는 구조체. 각 fd가 어떤 파일을 가리키는지 나타내는 `file` 포인터, 현재 점유중인지를 나타내는 `in_use` 플래그, 그리고 design에서 추가된 사항으로 해당 fd가 가리키고 있는 대상의 타입을 나타내는 `type` 로 이루어져 있다.

위와 같이 `type` 을 추가한 이유는 Linux상에서 실제 System call을 통해 테스트한 동작 방식을 구현하기 위함인데, 테스트 결과 Linux에서는 사전 정의되어있는 `0~2` 번 fd도 닫을 수 있었고, 해당 fd에 새로운 파일을 열 경우 fd가 `0~2` 임에도 불구하고 일반적인 파일과 같이 동작하였다. 때문에 본 프로젝트에서도 fd `0~1` 을 예외처리해주는 대신 현재 파일 타입을 보고 조건처리를 해주는 방식으로 구현해보았다.

```
#define OPEN_MAX 128

struct process
{
  ...
  struct fd_table_entry fd_table[OPEN_MAX];
};
```

File descriptor table은 고정된 길이의 배열 형식으로 `process` 구조체, 즉 process control block 위에 선언하였다. 현재는 최대 128개의 fd만 필요하고 후술할 메모리 누수를 관리하기 쉽게 하기 위해서 고정 배열 형식으로 선언했는데, 사용할 fd가 더욱 많아질 경우 `list` 형태로 선언해 fd를 동적으로 할당 및 해제하는 방식으로 재구현해야 할 것이다.

```
void
init_process (struct process *p)
{
  memset (p, 0, sizeof *p);
  ...

  p->fd_table[0].in_use = true;
  p->fd_table[0].type = FILETYPE_STDIN;
  p->fd_table[1].in_use = true;
  p->fd_table[1].type = FILETYPE_STDOUT;
}
```

fd table을 위와 같이 초기화해준다. fd 0과 1을 제외한 모든 엔트리는 0 `{NULL, false, FILETYPE_FILE}` 으로 초기화되고 fd 0과 1만 사전 정의된 대로 초기화해준다.

```
int
get_available_fd (struct process *p)
{
  for (size_t i = 0; i < OPEN_MAX; i++)
  {
    if (!p->fd_table[i].in_use)
      return i;
```

```
  }
  /* No more available fd in the table */
  return -1;
}
```

새로 fd를 할당하기 전 어디에 할당할 수 있는지를 반환하는 함수. 기본적으로 사용 가능한 fd 중 가장 낮은 번호를 반환한다. 고정 배열이므로 단순히 가장 낮은 인덱스부터 순차적으로 확인하는 방법으로 구현이 가능하다. 만약 모든 fd가 점유 중일 경우 `-1` 을 반환한다.

```
bool
set_fd (struct process *p, int fd, struct file *_file)
{
  if (!(0 <= fd && fd < OPEN_MAX)) return false;
  if (p->fd_table[fd].in_use) return false;
  p->fd_table[fd].file = _file;
  p->fd_table[fd].in_use = true;
  /* Currently there's no way to open STDIN or STDOUT
     Unless there's dup syscall or something */
  p->fd_table[fd].type = FILETYPE_FILE;
  return true;
}
```

프로세스의 fd 엔트리를 실제로 할당해주는 함수. 올바르지 않은 fd 번호가 들어오거나 이미 점유 중인 fd를 할당 시도할 경우 (`open-twice` 테스트로 확인) 실패를 반환한다. 정상적인 시도일 경우 `p->fd_table[fd]` 에 새로운 파일 포인터를 저장해준다.

```
void
remove_fd (struct process *p, int fd)
{
  if(!(2 <= fd && fd < OPEN_MAX)) return;
  /* Intended not to check the validity */
  p->fd_table[fd].in_use = false;
}
```

할당했던 fd를 해제하는 함수. 엄밀히 말하자면 할당되지 않은 fd를 해제해도 아무 일도 일어나지 않은 채 정상적으로 종료된다 (Linux 상에서 코드 테스트로 확인). 여기서는 fd 0과 1을 따로 예외처리 해주었는데, Linux 동작에서는 fd 0과 1도 `close` 가 가능했지만 본 프로젝트에선 `close-stdin` 과 `close-stdout` 테스트에 따라 해당 시도가 실패해야 하기 때문이다.

File lock in `userprog/process`

```
static struct lock file_lock;

/* Wrapper function for file_lock acquire */
void
file_lock_acquire (void)
{
  lock_acquire (&file_lock);
}

/* Wrapper function for file_lock release */
void
file_lock_release (void)
{
  lock_release (&file_lock);
}
```

파일에 대한 Concurrency 확보를 위해 파일 관련 함수 실행 전후로 `file_lock` 을 확보 및 해제해준다. 위 기능이 없을 시 `open-twice` 등의 테스트가 실패할 수 있다.

`sys_create` in `userprog/syscall`

```
static bool
sys_create (const char *file, unsigned initial_size)
{
  if (file == NULL || !check_ptr_in_user_space (file))
    sys_exit (-1);
  file_lock_acquire ();
  bool res = filesys_create (file, initial_size);
  file_lock_release ();
  return res;
}
```

올바른 파일 포인터인지 확인한 후 `filesys_create`의 결과를 반환해준다.

`sys_remove` in `userprog/syscall`

```
static bool
sys_remove(const char *file)
{
  if(file == NULL || !check_ptr_in_user_space(file))
    sys_exit(-1);
  file_lock_acquire();
  bool res = filesys_remove(file);
  file_lock_release();
  return res;
}
```

올바른 파일 포인터인지 확인한 후 `filesys_remove`의 결과를 반환해준다.

`sys_open` in `userprog/syscall`

```
static int
sys_open(const char *file)
{
  if(file == NULL || !check_ptr_in_user_space(file))
    sys_exit(-1);

  /* Whole section is critical section due to open-twice test */
  file_lock_acquire();
  struct process *cur = thread_current()->process_ptr;

  int fd = get_available_fd(cur);
  if(fd == -1)
  {
    file_lock_release();
    return -1;
  }

  struct file *target_file = filesys_open(file);
  if(target_file == NULL)
  {
    file_lock_release();
    return -1;
  }

  /* Should verify the return value but seems okay now */
  set_fd(cur, fd, target_file);

  file_lock_release();
  return fd;
}
```

올바른 파일 포인터인지 확인한 후, `filesys_open(file)`을 통해 해당 이름의 파일을 열고 상술한 `get_available_fd`로 사용 가능한 fd를 받아 해당 fd에 파일을 할당한다. 해당 로직 전체를 `file_lock`으로 묶어 critical section으로 만들어줘야 동시에 두 파일을 open 시도해 같은 fd값에 파일을 쓰는 오류가 일어나지 않는다.

`sys_filesize` in `userprog/syscall`

```
static int
sys_filesize(int fd)
{
  if(!(0 <= fd && fd < OPEN_MAX))
    return -1;

  struct process *cur = thread_current()->process_ptr;

  struct fd_table_entry *fd_entry = &(cur->fd_table[fd]);
  if(!(fd_entry->in_use &&
       fd_entry->type == FILETYPE_FILE &&
       fd_entry->file != NULL))
       return -1;

  file_lock_acquire();
  int res = file_length(fd_entry->file);
  file_lock_release();

  return res;
}
```

주어진 fd에 해당하는 파일의 크기를 반환한다. 현재 프로세스의 fd table은 `thread_current()->process_ptr->fd_table`로 접근 가능하다. 현재 가리키고 있는 파일이 할당 중이고, `stdin`이나 `stdout`이 아닌 실제 파일일 경우 `file_length`를 통해 파일 길이를 반환한다. 올바르지 않은 호출일 경우 `-1`을 반환한다.

`sys_read` in `userprog/syscall`

```
static int
sys_read(int fd, void *buffer, unsigned size)
{
  if(!check_ptr_in_user_space(buffer))
    sys_exit(-1);
  if(!(0 <= fd && fd < OPEN_MAX))
    return -1;

  struct process *cur = thread_current()->process_ptr;

  if(!cur->fd_table[fd].in_use)
    return -1;

  int file_type = cur->fd_table[fd].type;
  if(file_type == FILETYPE_STDIN)
  {
    void *cur_pos = buffer;
    unsigned write_count = 0;
    while(write_count < size)
    {
      if(!check_ptr_in_user_space(cur_pos))
        sys_exit(-1);
      uint8_t c = input_getc();
      if(!put_user((uint8_t *)cur_pos, c))
        sys_exit(-1);
      write_count++;
      cur_pos++;
    }
    return write_count;
```

```
  }
  else if(file_type == FILETYPE_STDOUT)
  {
    /* Actually it also works same as STDIN in LINUX */
    sys_exit(-1);
  }
  else
  {
    file_lock_acquire();
    int res = file_read(cur->fd_table[fd].file, buffer, size);
    file_lock_release();
    return res;
  }
}
```

`buffer`와 fd가 올바른지를 확인한 후, 일반적인 파일일 시 `file_read`를 통해 최대 `size` 바이트만큼을 파일에서 읽어 `buffer`에 복사한다. 만약 fd가 0이었을 경우 `input_getc`와 `put_user` 함수들을 이용해 키보드 입력에서 한 바이트씩 읽어 `buffer`에 복사한다. fd가 1일 경우 Linux에서는 stdin과 똑같이 키보드 입력에서 데이터를 복사해오지만 본 프로젝트에선 `exit(-1)`을 실행하도록 하였다.

`sys_write` in `userprog/syscall`

```
static int
sys_write(int fd, const void *buffer, unsigned size)
{
  if(!check_ptr_in_user_space(buffer))
    sys_exit(-1);
  if(!(0 <= fd && fd < OPEN_MAX))
    return -1;

  struct process *cur = thread_current()->process_ptr;

  if(!cur->fd_table[fd].in_use)
    return -1;

  int file_type = cur->fd_table[fd].type;
  if(file_type == FILETYPE_STDIN)
  {
    /* Actually it also works same as STDOUT in LINUX */
    sys_exit(-1);
  }
  else if(file_type == FILETYPE_STDOUT)
  {
    putbuf(buffer, size);
    return size;
  }
  else
  {
    file_lock_acquire();
    int res = file_write(cur->fd_table[fd].file, buffer, size);
    file_lock_release();
    return res;
  }
}
```

`sys_read`와 유사한 로직으로 구현하였고, `file_write`를 통해 최대 `size` 바이트만큼을 `buffer`에서 읽어 파일에 복사한다. fd가 1일 때는 `putbuf`를 호출하여 해당 내용을 콘솔에 출력한다. fd가 0일 경우 Linux에서는 stdout과 동일하게 콘솔에다 출력하지만 본 프로젝트에선 `exit(-1)`을 실행하도록 하였다.

`sys_seek` in `userprog/syscall`

```
static void
sys_seek(int fd, unsigned position)
{
  if(!(0 <= fd && fd < OPEN_MAX))
    return;

  struct process *cur = thread_current()->process_ptr;

  struct fd_table_entry *fd_entry = &(cur->fd_table[fd]);
  if(!(fd_entry->in_use &&
       fd_entry->type == FILETYPE_FILE &&
       fd_entry->file != NULL))
       return;

  file_lock_acquire();
  file_seek(fd_entry->file, position);
  file_lock_release();
}
```

fd와 파일 검증을 마친 후 `file_seek`를 통해 현재 파일이 보고 있는 위치를 `position`으로 바꿔준다. 파일 크기보다 더 큰 위치로 이동하더라도 정상 동작으로 간주한다.

`sys_tell` in `userprog/syscall`

```
static unsigned
sys_tell(int fd)
{
  if(!(0 <= fd && fd < OPEN_MAX))
    return -1;

  struct process *cur = thread_current()->process_ptr;

  struct fd_table_entry *fd_entry = &(cur->fd_table[fd]);
  if(!(fd_entry->in_use &&
       fd_entry->type == FILETYPE_FILE &&
       fd_entry->file != NULL))
       return -1;

  file_lock_acquire();
  unsigned res = file_tell(fd_entry->file);
  file_lock_release();

  return res;
}
```

fd와 파일 검증을 마친 후 `file_tell`을 통해 현재 파일이 보고 있는 위치를 반환한다.

`sys_close` in `userprog/syscall`

```
static void
sys_close(int fd)
{
  if(!(0 <= fd && fd < OPEN_MAX))
    return;

  struct process *cur = thread_current()->process_ptr;

  struct fd_table_entry *fd_entry = &(cur->fd_table[fd]);
  if(fd_entry->in_use &&
     fd_entry->type == FILETYPE_FILE &&
     fd_entry->file != NULL)
  {
```

```
    file_lock_acquire();
    file_close(fd_entry->file);
    file_lock_release();
  }

  remove_fd(cur, fd);
}
```

fd와 파일 검증을 마친 후 `file_close`를 통해 파일을 닫아준다.

Denying Writes to Executables

```
struct process
{
  ...
  struct file *file_exec;
  ...
};
```

PCB에 현재 실행 중인 프로그램 파일을 나타내는 포인터 변수를 추가해준다.

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

  /* Open executable file. */
  file_lock_acquire();
  file = filesys_open (file_name);

  ...

  success = true;
  file_deny_write(file);
  t->process_ptr->file_exec = file;

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

프로세스에 실행 파일을 load할 때, 모든 작업이 성공하였을 경우 `file_deny_write`를 호출하여 해당 프로그램 파일이 수정되는 것을 막는다. 또한 기본 구현에서 load가 끝날 때 프로그램 파일을 바로 `close`해버리는 동작을 수정하여 계속 열어두도록 고쳤다. 해당 로직을 `file_lock`을 통해 묶어주는 과정 역시 필요하다.

```
void
sys_exit (int status)
{
  ...
  file_close (cur->process_ptr->file_exec);
  for (size_t i = 2; i < OPEN_MAX; i++)
  {
    if(cur->process_ptr->fd_table[i].in_use)
    {
      file_close (cur->process_ptr->fd_table[i].file);
      remove_fd (cur->process_ptr, i);
    }
  }
```

```
    ...
}
```

Deny 해제는 프로세스가 종료되고 `exit` 을 호출하는 도중 `file_close` 를 통해 현재 실행중이었던 프로그램 파일을 close해주면 완료된다. (내부적으로 `file_allow_write` 호출)

발전한 점

이번 프로젝트를 통해 시스템 콜과 파일 관련한 동작을 구현하여 OS가 실제 파일과 유저 프로그램을 읽어와 실행 가능하도록 구현하는 데 성공했다. 또한 파일을 읽고 쓰는 중 유저에 의해 일어날 수 있는 여러 예외상황들을 처리하고, 프로세스 할당 및 해제가 일어날 때 발생하는 메모리 누수의 원인을 분석한 뒤 해결하였다.

먼저 기존에는 유저 프로그램을 정상적으로 실행조차 할 수 없는 상황이였다.
왜냐하면 커맨드라인으로 입력 받은 실행하고자 하는 유저 프로그램의 명칭을 정확히 파싱하지 않아 유저 프로그램 로드에 실패하였기 때문이다. 또한 모든 시스템 콜 호출시 스레드를 단순히 종료시키고 각 시스템 콜에 대한 올바른 작동을 하지 않았다.

Process Terminiation Messages

Process Termination Message를 통해 유저 프로그램이 종료되었을 때, 또는 유저 프로그램이 생성한 다른 프로세스 들이 종료되었을 때 어떤 프로세스가 종료되었는지, 왜 종료되었는지, 오류로 인한 것인지 등을 알 수 있게 되었다.

Argument Passing

유저가 핀토스에게 실행하라고 넘겨준 명령어를 올바르게 파싱해서 해석할 수 있게 되었다. 실행할 파일명의 스레드 명을 가지는 유저 프로세스에 대한 스레드를 만들 수 있다. 또한 이전에는 실행할 프로그램과 프로그램에 넘겨줄 arguments를 구분하지 못하여 "프로그램명+arguments"를 load하려고 하여 올바른 유저 프로그램을 불러와 실행할 수 없었는데 이제 이 둘을 구분하여 올바른 유저 프로그램을 실행할 수 있다. 또한 이제 80x86 Calling Convention을 준수하여 Argument Passing을 통해 원하는 argument, 함수 인자를 유저 프로그램에 넘겨 실행할 수 있다.

System Call

유저 프로그램에서 유용하게 사용할 수 있는 여러 시스템 콜 기능을 사용할 수 있게 되었다. 시스템 콜이 발생할 시 스택에 있는 데이터를 참고하여 어떤 시스템 콜이 호출되었는지와 인자로 어떤 값들이 전달되었는지를 알아낸 후 대응되는 로직을 실행한다.

File system

`open` , `read` , `write` 등 파일을 열고 수정하는 시스템 콜을 통해 파일과 관련된 동작을 수행할 수 있게 되었다. file 및 file system 관련된 인터페이스는 사전 정의되어 있는 함수들을 이용했고, 추가적으로 여러 프로세스가 동시에 파일 시스템에 접근할 때 일어날 수 있는 동시성 문제를 lock을 통해 해결하였다.

한계

파일 및 디렉토리 구조

현재 구현에선 디렉토리 관련 구현을 하지 않았기 때문에 모든 유저 파일이 루트 디렉토리 아래에 위치해있다고 가정하고 실행을 한다. 이를 구현하기 위해선 커맨드나 함수 인자로 주어지는 파일명을 디렉토리 별로 다시 파싱하여 관리하는 등의 추가적인 구현 사항이 있을 것으로 예상된다.

정적 File descriptor Table

또한 앞서 언급했듯 현재 file descriptor table의 구현은 고정 길이의 배열을 pcb에 정적으로 할당하는 방식인데, 사용 가능한 fd의 크기가 더욱 늘어날 경우 (~65535) 위의 방법 대신 리스트를 이용해 동적으로 할당해주는 방식으로 수정이 필요할 것이다. 다만 동적으로 할당하는 경우 메모리 누수에 더욱 주의해야 한다.

부모, 자식 프로세스 life cycle 한계

현재 구현은 부모, 자식의 프로세스 life cycle에서 다소 이상적인 다음 가정을 필요로 한다. "부모 프로세스는 자식 프로세스가 종료하기 전까지 유지된다."

현재 구현상 만일 부모 프로세스가 자식 프로세스보다 먼저 종료된다면 부모 프로세스가 자식 프로세스를 wait할 수 없으므로 자식 프로세스(스레드에 대한 자원을 할당 해제됨.)의 프로세스 자원(Process Control Block(`process`)에 할당된 페이지 등)을 할당 해제 할 수 없다. 즉 일종의 좀비 프로세스, 고립된 프로세스가 될 수 있다. 이를 위해 자식 프로세스를 아직 종료되지 않은 다른 프로세스에게 자식으로 넘겨주거나 부모 프로세스가 종료될 때 자신의 자식 프로세스들이 이후 PCB를 할당 해제할 것을 보장하는 구현을 추가해야할 것이다.

프로세스 - 스레드 1대1 매핑

현재 구현상 한 프로세스는 한 스레드를 가진다. `process` 가 하나의 `tid` 를 가지고 `thread` 도 하나의 process 포인터 `process_ptr` 을 가지기 때문이다. 하지만 실제로는 멀티 스레드 프로그램 등은 한 프로세스가 여러 스레드를 실행하고자 하기도 한다. 이를 위해 현재 `process` 가 자식 프로세스들 `children` 을 관리하는 것과 유사하게 리스트로 여러 `tid` 또는 `thread` 포인터 등을 가질 수 있게 해야 한다.

배운 점

유저 프로그램 실행시의 Argument Passing 및 System Call Handler의 Argument Passing을 구현할 때 `hex_dump` 등을 사용하여 디버깅하여 메모리 상에서의 디버깅 방법을 알 수 있게 되었다.
평소에 C언어로 짠 프로그램에 argument, argument 개수 등이 어떻게 들어가는지 모르고 그저 사용하였는데 80x86 Convention에 맞추어 Argument Passing을 구현함으로써 어떠한 원리/과정으로 C언어 main함수에 값들이 넘어가는지 이해하게 되었다.
스택 값 설정하기, `process` 와 `thread` 간 관계 생성, `palloc` 을 이용한 페이지 할당 등을 사용하는 과정 속에서 다소 불분명하게 이해하고 있던 C 포인터 개념에 대해 완벽히 이해할 수 있었다.
물리적 메모리 대신 가상 메모리 공간을 만들고 매핑하여 사용하는 이유와 User space, Kernel space로 분리된 이유를 배웠고, 두 영역 간에 잘못된 접근을 처리하는 방법 또한 알게 되었다.
`multi-oom` 테스트를 통과하기 위해 디버깅하는 과정에서 메모리 누수 없이 자원을 할당하고 해제하는 것이 얼마나 어려운 일인지, 좀비 프로세스가 왜 어쩔수 없이 생길 수 밖에 없는지, 왜 C로 짠 프로그램들이 메모리 관련 버그가 많을 수 밖에 없는지, Garbage Collecting을 지원하는 언어들은 왜 나오는지를 깊이 이해하게 되었다. Rust의 소유권 개념이 왜 유용한지 잘 와닿지 않았는데 이번 과제를 통해 Rust 소유권이 왜 대단하고 C를 Rust로 대체하기 위해 노력하는지 이해하였다.
Physical memory 위의 데이터가 일정한 크기의 블럭 단위로 어떻게 관리되는지를 `inode` 와 파일 시스템 분석을 통해 알 수 있었고, 파일 관련 기능들을 다룰 때 동시성을 확보하지 않으면 많은 문제가 생길 수 있음을 깨달았다.