# Pintos Project 2: User Program (2)

# I. 개발 목표

File system을 위한 system call을 구현한다. create, remove, open, close, filesize, seek, tell system call의 경우 새롭게 구현하고, project 1에서 구현한 read, write system call은 project 1에서 구현한 write(stdout), read(stdin)을 보강하여 구현한다.

# II. 개발 범위 및 내용

## A. 개발 범위

1. File Descriptor

Process에서 파일에 접근하기 위해서는 file descriptor가 필요하다. 각 process가 가진 독립된 file descriptor를 구현하여 이를 가능하도록 한다.

2. System Calls

File system을 위한 system call(create, remove, open, close, filesize, seek, tell)을 새로 구현해 파일에 대한 각 함수의 기능이 작동하도록 한다. 또한 project 1에서 standard input, standard output에만 구현되었던 read, write system call을 보강해 파일에 대해서도 read, write가 가능하도록 함수 기능을 확장한다.

3. Synchronization in Filesystem

File system code를 critical section으로 처리하여 같은 파일에 대해 여러 개의 process가 동시에 접근하는 경우를 막는다. 특히 read와 write를 동시에 수행하는 경우를 막는다.

## B. 개발 내용

1. File Descriptor

File descriptor의 구현에 이용한 자료구조는 배열이다. Pintos manual p.35를 참고해 배열의 크기는 128로 설정하였다. 필요할 때마다 동적 할당을 하는 방법도 있으나, 코드의 단순함을 위해 정적 구조를 사용했다.

2. System Calls

- create(): 파라미터로 받은 이름과 파일 크기로 새 파일을 생성한다.
- remove(): 파라미터로 받은 이름의 파일을 삭제한다.
- open(): 파라미터로 받은 이름의 파일을 연다.
- filesize(): 파라미터로 받은 file descriptor에 해당하는 파일의 byte 크기를 리턴한다.
- seek(): 파라미터로 받은 file descriptor에 해당하는 파일에서 읽거나 쓸 다음 위치를 파라미터로 받은 위치로 변경한다.
- tell(): 파라미터로 받은 file descriptor에 해당하는 파일에서 읽거나 쓸 다음 위치를 리턴한다.
- close(): 파라미터로 받은 file descriptor에 해당하는 파일을 닫는다.
- read(): 파일에서 정해진 크기 만큼 읽어들인다. file descriptor가 0일 경우 standard input에서 읽어들인다.
- write(): 파일에서 정해진 크기 만큼 출력한다. file descriptor가 1일 경우 standard output에서 출력한다.

3. Synchronization in Filesystem

- Lock

  src/threads/synch.h

```c
/* Lock. */
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };
```

  lock 구조체 변수를 이용해 critical section 앞에서 lock_acquire()를 호출하고, critical section 뒤에서 lock_release()를 호출해 여러 개의 process가 파일 시스템 코드를 동시에 호출하지 못 하도록 할 수 있다.

- Semaphore

src/threads/synch.h

```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

semaphore를 이용하면 readers-writers problem을 해결할 수 있다. Semaphore로 mutex와 wrt를 사용하고 int 타입의 readcount를 사용한다. Reading 수행 구간 앞에서 sema_down(mutex)를 호출하고 readcount를 증가시킨 뒤, sema_up(mutex)를 호출한다. readcount가 1인 경우 read를 수행하고 있는 process가 있으므로 sema_down(wrt)를 호출해야 한다. Reading 수행 후 sema_down(mutex)를 호출하고 readcount를 감소시킨 뒤 readcount가 0이라면 더 이상 read를 수행하고 있는 process가 없으므로 sema_up(wrt)를 호출해 write가 가능하도록 한 뒤 sema_up(mutex)를 호출한다. 또한 Writing 수행 구간 앞에 sema_down(wrt), 뒤에 sema_up(wrt)를 호출해준다.

한편 child thread의 load가 성공할 때까지 parent thread가 기다려야 하므로 thread 구조체에 load 관련 semaphore를 추가한 뒤 process_execute()에서 sema_down()을 호출하고 start_process()에서 sema_up()을 호출한다.

# III. 추진 일정 및 개발 방법

## A. 추진 일정

11/1-11/7: 프로젝트 명세서와 pintos manual을 읽으며 목표를 파악한다.

11/8-11/13: File descriptor와 system call을 구현하고 synchronization을 구현한다.

11/13-11/14: 보고서 작성

## B. 개발 방법

1. File Descriptor

src/threads/thread.*

: thread 구조체에 file descriptor를 구현하기 위한 배열을 추가하고, init_thread()에 file descriptor를 초기화하는 코드를 추가한다.

2. System Calls

src/userprog/syscall.*

: Switch문에 create, remove, open, close, filesize, seek, tell을 호출할 수 있도록 코드를 추가하고, 각 기능을 수행할 수 있도록 함수를 구현한다. read, write 함수는 file에 대해 기능을 수행할 수 있도록 코드를 추가 및 수정한다. 또 파일명이 NULL인 경우를 확인하도록 코드를 추가한다.

src/userprog/exception.c

: page_fault()에 조건을 추가해 NULL 주소에 접근하는 경우를 해결한다.

src/userprog/process.c

: process_exit() 함수에서 thread의 file을 모두 닫아주도록 코드를 추가한다.

3. Synchronization in Filesystem

src/userprog/syscall.*

: read(), write(), open() 함수 내에서 기능 수행 전 lock_acquire(), 기능 수행 후 lock_release()를 호출해 여러 개의 process가 파일 시스템 코드를 동시에 호출하지 못 하도록 한다.

src/threads/thread.*,

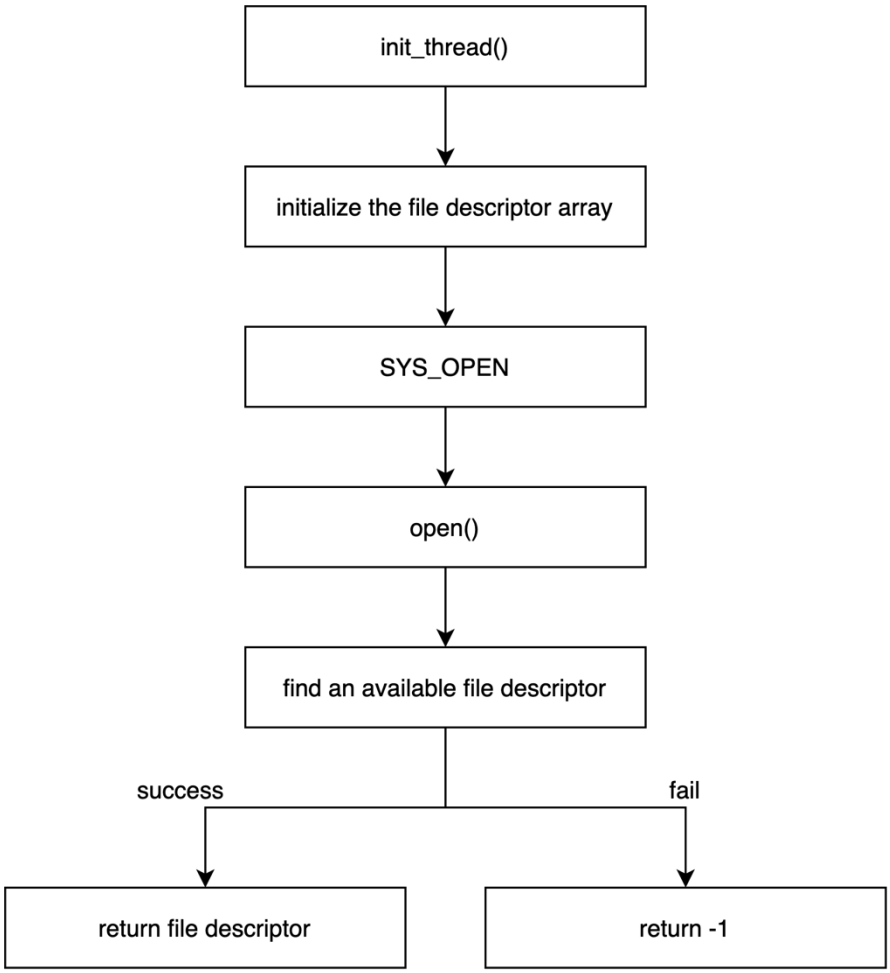
src/userprog/process.c

: child process가 성공적으로 load되는 것을 parent process가 기다려야 하기 때문에 이를 확인하기 위해 load를 확인할 수 있는 semaphore를 struct thread에 추가해주고 init_thread()에서 이를 초기화해준다. 또한 process_execute()에서 sema_down()을 호출하고 start_process()에서 sema_up()을 호출한다.

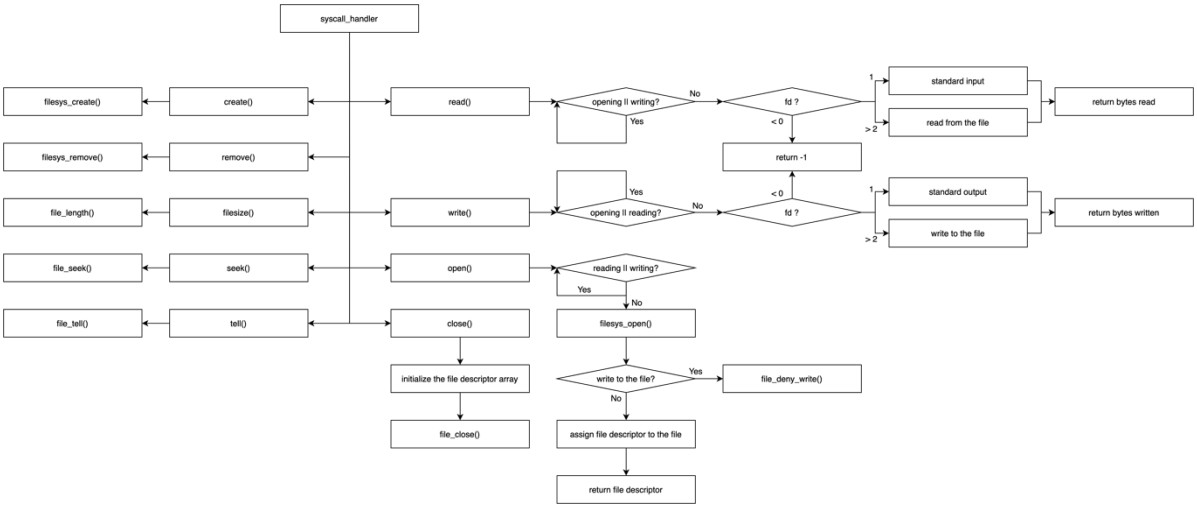
# IV. 연구 결과

## A. Flow Chart

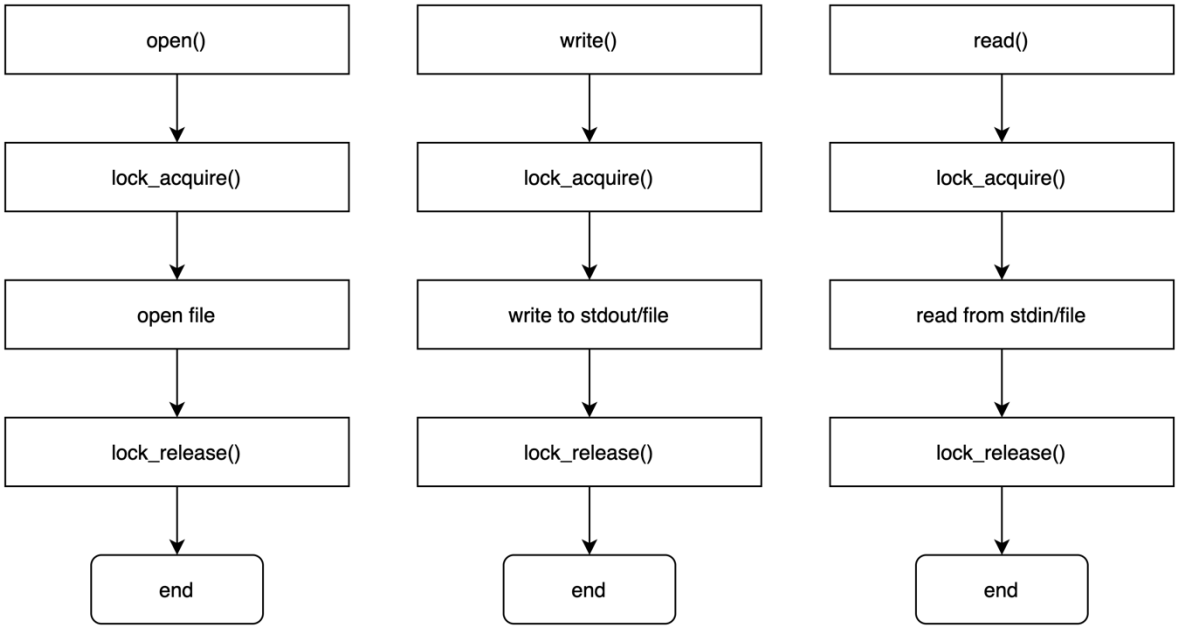
1. File Descriptor



## 2. System Calls




## 3. Synchronization in Filesystem



## B. 제작 내용

1. File Descriptor

src/threads/thread.*

```
struct thread
  {
    . . .
    /* ============================ */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                /* Page directory. */
    struct semaphore child_sema;
    struct semaphore mem_sema;
    struct semaphore load_sema;
    struct list child;
    struct list_elem child_elem;
    struct file* file_d[128];
    int exit_status;
    bool success;
#endif
    /* ============================ */
    . . .
  };
```

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  . . .
  /* ============modified in prj2================ */
#ifdef USERPROG
  for (int i = 0; i < 128; i++)
    t->file_d[i] = NULL;

  sema_init(&(t->child_sema), 0);
  sema_init(&(t->mem_sema), 0);
  sema_init(&(t->load_sema), 0);

  list_init(&(t->child));
  list_push_back(&(running_thread()->child), &(t->child_elem));
#endif
  /* ============================ */
}
```

thread 구조체에 file descriptor 배열을 선언하는 코드를 추가하고, init_thread() 함수에 이를 NULL로 초기화하는 코드를 추가했다.

2. System Calls

src/userprog/syscall.c

```
syscall_handler (struct intr_frame *f UNUSED)
{
   /* Project 1: system call impelementation */
   switch (*(uint32_t*)(f->esp)) {
   . . .
   /* Project 2: system call impelementation */
   case SYS_CREATE:
      check_valid(f->esp + 4);
      check_valid(f->esp + 8);
      f->eax = create((const char *)*(uint32_t*)(f->esp + 4), (unsigned)*(uint32_t*)(f->esp + 8));
      break;

   case SYS_REMOVE:
      check_valid(f->esp + 4);
      f->eax = remove((const char *)*(uint32_t*)(f->esp + 4));
      break;

   case SYS_OPEN:
      check_valid(f->esp + 4);
      f->eax = open((const char *)*(uint32_t*)(f->esp + 4));
      break;

   case SYS_FILESIZE:
      check_valid(f->esp + 4);
      f->eax = filesize((int)*(uint32_t*)(f->esp + 4));
      break;

   case SYS_SEEK:
      check_valid(f->esp + 4);
      check_valid(f->esp + 8);
      seek((int)*(uint32_t*)(f->esp + 4), (unsigned)*(uint32_t*)(f->esp + 8));
      break;

   case SYS_TELL:
      check_valid(f->esp + 4);
      f->eax = tell((int)*(uint32_t*)(f->esp + 4));
```

```
        break;

    case SYS_CLOSE:
        check_valid(f->esp + 4);
        close((int)*(uint32_t*)(f->esp + 4));
        break;
    . . .
}
```

switch문에 SYS_CREATE, SYS_REMOVE, SYS_OPEN, SYS_FILESIZE, SYS_SEEK, SYS_TELL, SYS_CLOSE의 case를 추가하여 pointer가 valid한지 체크하고 각 함수를 호출하도록 했다. case는 src/lib/syscall-nr.h에 선언되어있는 코드를 참고했다.

src/userprog/syscall.c

```
void check_valid(const void *vaddr) {
    if (!is_user_vaddr(vaddr))
        exit(-1);
    if (vaddr == NULL)
        exit(-1);
}
```

check_valid() 함수를 수정해 filesystem 관련 system call 함수에서 파일 이름이 NULL인지 확인하는 과정을 거칠 수 있도록 사용했다.

src/userprog/syscall.c

```
bool create(const char *file, unsigned initial_size) {
    check_valid(file);
    return filesys_create(file, (off_t)initial_size);
}
```

create() 함수에서는 수정한 check_valid() 함수를 이용해 파일명이 NULL인지 확인했고, filesys_create()를 호출해 파일을 생성했다. create() 함수는 생성에 성공한 경우 true, 실패한 경우 false를 리턴한다. filesys_create() 함수는 src/filesys/filesys.*에 구현되어있는 것을 참조했다.

src/userprog/syscall.c

```
bool remove(const char *file) {
    check_valid(file);
    return filesys_remove(file);
}
```

remove() 함수에서는 파일명이 NULL인지 확인한 뒤 filesys_remove()를 호출해 파일을 삭제했다. remove() 함수는 삭제에 성공한 경우 true, 실패한 경우 false를 리턴한다. filesys_remove() 함수는 src/filesys/filesys.*에 구현되어있는 것을 참조했다.

src/userprog/syscall.c

```
int filesize(int fd) {
    return file_length(thread_current()->file_d[fd]);
}
```

filesize() 함수는 현재 thread의 file descriptor 배열에서 파라미터로 전달받은 file descriptor에 해당하는 파일의 크기(byte 단위)를 file_length()를 호출하여 반환한다. file_length() 함수는 src/filesys/file.*에 구현되어있는 것을 참조했다.

src/userprog/syscall.c

```
void seek(int fd, unsigned position) {
    file_seek(thread_current()->file_d[fd], position);
}
```

seek() 함수는 현재 thread의 file descriptor 배열에서 파라미터로 전달받은 file descriptor에 해당하는 파일에서 읽거나 쓸 다음 위치를 *position*(시작점으로부터 몇 byte 떨어져있는가)으로 변경한다. file_seek() 함수는 src/filesys/file.*에 구현되어있는 것을 참조했다.

src/userprog/syscall.c

```
unsigned tell(int fd) {
    return file_tell(thread_current()->file_d[fd]);
}
```

tell() 함수는 현재 thread의 file descriptor 배열에서 파라미터로 전달받은 file descriptor에 해당하는 파일에서 읽거나 쓸 다음 위치(시작점으로부터 몇 byte 떨어져있는가)를 리턴한다.

file_tell() 함수는 src/filesys/file.*에 구현되어있는 것을 참조했다.

src/userprog/syscall.c

```
int open(const char *file) {
    struct file* fp;
    int fd;

    check_valid(file);
    lock_acquire(&file_lock); //lock

    fp = filesys_open(file);

    if (fp == NULL) {
        lock_release(&file_lock); //lock
        return -1;
    }

    if (strcmp(thread_name(), file) == 0)
        file_deny_write(fp);

    for (fd = 3; fd < 128; fd++) {
        if (thread_current()->file_d[fd] == NULL) {
            thread_current()->file_d[fd] = fp;
            lock_release(&file_lock); //lock
            return fd;
        }
    }

    lock_release(&file_lock); //lock
    return -1;
}
```

open() 함수에서는 먼저 파일명이 NULL인지 확인한 뒤 filesys_open() 함수를 통해 파일을 연다. filesys_open() 함수는 src/filesys/filesys.*에 구현되어있는 것을 참조했는데, 파일 오픈에 실패한 경우 NULL을 반환하므로 이 경우 -1을 리턴하며 함수를 종료한다. 파일을 성공적으로 오픈한 경우, 수행 중인 파일에 write하는 것을 막아야 하기 때문에 thread와 파일명을 비교해 일치하는 경우 file_deny_write() 함수를 호출했다. file_deny_write() 함수는 src/filesys/file.*을 참조했다. 이후 stdin(0), stdout(1), stderr(2)를 제외하고 3부터 for문을 이용해 file descriptor 배열을 확인하여 NULL인 것을 찾아 파일을 할당해준다. 파일 할당에 성공한 경우 해당 file descriptor(nonnegative integer) 값을 반환하고, 실패한 경우 -1을 반환한다.

src/userprog/syscall.c

```
void close(int fd) {
    struct file* fp;
    fp = thread_current()->file_d[fd];
    if (fp == NULL)
        exit(-1);
    thread_current()->file_d[fd] = NULL;
    file_close(fp);
}
```

파라미터로 전달받은 fd에 해당하는 파일을 닫는다. file descriptor 배열에서 해당 위치를 NULL로 초기화한 뒤, file_close() 함수를 호출했다. 단, file descriptor 배열에서 이미 NULL인 경우에는 함수를 종료했다. file_close() 함수는 src/filesys/file.*를 참조했다.

src/userprog/syscall.c

```
int read(int fd, const void *buffer, unsigned size) {
    /* =========modified in prj2========== */
    int i = -1;
    struct thread* cur = thread_current();

    check_valid(buffer);
    lock_acquire(&file_lock);

    if (fd == 0) { //read from the keyboard
        for (int i = 0; i < (int)size && ((char*)buffer)[i] != '\0'; i++)
            buffer = input_getc();
    }

    else if (2 < fd && fd < 128) {
        if(cur->file_d[fd] == NULL) {
            lock_release(&file_lock);
            return -1;
        }
        i = (int)file_read(cur->file_d[fd], (void*)buffer, (off_t)size);
    }

    lock_release(&file_lock);
    return i;
    /* ================== */
}
```

Project 1에서 read() 함수가 stdin(0)인 경우에 작동하도록 작성했다. 이번 project 2에서 이를 확장하여 file descriptor 배열에서 3부터 127까지의 경우 파일을 읽어들일 수 있도록 했다. 파일을 읽는 데는 file_read() 함수를 이용했고, 만약 배열에서 해당 file descriptor에 대한 값이 NULL이라면 함수를 종료하도록 했다. 파라미터로 전달된 file descriptor 값이 범위를 벗어나는 경우 -1을 리턴한다. file_read() 함수는 src/filesys/file.*를 참조했다.

src/userprog/syscall.c

```
int write(int fd, const void *buffer, unsigned size) {
   /* =========prj2========== */
   struct thread* cur = thread_current();
   int i = -1;

   check_valid(buffer);
   lock_acquire(&file_lock);

   if (fd == 1) { //write to the console
      putbuf((char*)buffer, (size_t)size);
      i = (int) size;
   }

   else if (2 < fd && fd < 128) {
      if (cur->file_d[fd] == NULL) {
         lock_release(&file_lock);
         exit(-1);
      }
      i = (int)file_write(cur->file_d[fd], buffer, (off_t)size);
   }

   lock_release(&file_lock);
   return i;
   /* ================== */
}
```

Project 1에서 write() 함수가 stdout(1)인 경우에 작동하도록 작성했다. 이번 project 2에서 이를 확장하여 file descriptor 배열에서 3부터 127까지의 경우 file_write() 함수를 이용해 입력이 가능하도록 했다. 만약 배열에서 해당 file descriptor에 대한 값이 NULL이라면 함수를 종료하도록 했다. 파라미터로 전달된 file descriptor 값이 범위를 벗어나는 경우 -1을 리턴한다. file_write() 함수는 src/filesys/file.*를 참조했다.

src/userprog/exception.c

```
static void
page_fault (struct intr_frame *f)
{
  . . .
  if (not_present)
     exit(-1);
  . . .
}
```

page_fault() 함수에서 not_present가 true인 경우 종료하도록 하여 NULL 주소에 접근하는 경우를 해결한다.

3. Synchronization in Filesystem

src/userprog/syscall.c

```
/* prj2 ==================== */
struct lock file_lock;
/* ======================== */

void
syscall_init (void)
{
  /* prj2 ==================== */
  lock_init(&file_lock);
  /* ======================== */
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}
```

src/threads/synch.*의 lock 구조체를 활용해 *file_lock*을 선언하고, syscall_init() 함수에서 이를 초기화했다. 그리고 2. System Calls에서 open, read, write 함수 내부에서 기능을 수행하기 전 lock_acquire()를 호출하고, 함수 종료 전 lock_release()를 호출하도록 해 다른 thread와 동시에 수행되는 것을 막았다.

src/userprog/process.c

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

```
  /* ===========prj2========== */
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);

  struct list_elem *e = list_begin(&(thread_current())->child);
  struct thread *t;

  while (e != list_end(&(thread_current()->child))) {
    t = list_entry(e, struct thread, child_elem);
    if (t->tid == tid)
      sema_down(&(t->load_sema)); //load sema down
    e = list_next(e);
  }

  if (!(t->success && tid != -1))
     return -1;
  /* ========================== */

  return tid;
}
```

```
static void
start_process (void *file_name_)
{
  . . .
  /* ===========prj2========== */
  thread_current()->success = success;
  /* ========================== */
  . . .
  /* ===========prj2========== */
  sema_up(&thread_current()->load_sema); //load sema up
  if (!success) {
    thread_exit ();
  }
  /* ========================== */
  . . .
}
```

Parent process에서는 load semaphore에 대해 sema_down()을 호출하고 child process에서는 start_process()를 수행할 때 load semaphore에 대해 sema_up()을 호출하여서, parent가 child의 load를 기다리도록 한다. 또 thread 구조체에 추가된 *success*를 활용해 load 성공/실패를 저장한다.

## C. 시험 및 평가 내용

```
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/filesys/base/syn-write -a syn-write -p tests/filesys/base/child-syn-wrt -a
 child-syn-wrt -- -q  -f run syn-write < /dev/null 2> tests/filesys/base/syn-write.errors > tests/filesys/base/syn-write.output
perl -I../.. ../../tests/filesys/base/syn-write.ck tests/filesys/base/syn-write tests/filesys/base/syn-write.result
pass tests/filesys/base/syn-write
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
```

```
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
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.
make[1]: Leaving directory '/sogang/under/cse20181464/pintos/src/userprog/build'
cse20181464@cspro9:~/pintos/src/userprog$ exit
```