Pintos Project 1: User Program (1)

1. 개발 목표

Pintos에서 Argument Passing, User Memory Access, System Call을 구현하여 user program이 정상적으로 작동하도록 한다. 이번 프로젝트1에서는 system call handler와 halt(), exit(), exec(), wait(), write(stdout), read(stdin)와 추가적인 system call로 fibonacci(), max_of_four_int()를 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Argument Passing

입력받은 argument를 parsing하여 80x86 calling convention에 따라 esp를 사용해 stack에 push하여 메모리를 할당한다.

2. User Memory Access

User가 범위 밖의 memory에 접근하려고 하면 프로그램을 종료한다.

3. System Calls

halt(), exit(), exec(), wait(), write(stdout), read(stdin)와 추가적인 system call로 fibonacci(), max_of_four_int()의 system call을 구현해 전달받은 argument에 대해 각자 해당하는 작업을 하도록 한다.

B. 개발 내용

- Argument Passing

  ✓ 커널 내 스택에 argument를 쌓는 과정

  먼저 argument를 띄어쓰기 단위로 parsing해준 뒤, 한 개씩 stack에 쌓아준다. stack에 쌓아줄 때는 esp를 사용하고 esp를 감소시켜가며 쌓아준다. 80x86 calling convention에 따라서 argv의 내용을 push하고 word alignment를 시켜준 뒤 null pointer sentinel을 push하고, argv의 string들의 주소를 push, argv(argv[0]의 주소)를 push, argc를 push한다. 마지막으로 return address를 push한다.

- User Memory Access

  ✓ Pintos 상에서의 invalid memory access 개념

  ✓ Invalid memory access를 막는 방법

  Pintos 상에서 invalid한 memory를 access하는 경우에는 NULL pointer를 통해 메모리에 접근하려고 하거나, unmapped virtual memory에 접근하려고 하거나, kernel virtual address space에 접근하려고 하는 경우가 있다. 이러한 invalid memory accesss는 exception.c의 page_fault()를 수정해 위 세 가지에 대한 예외처리를 해주면 막을 수 있다.

- System Calls

  ✓ 시스템 콜의 필요성

  ✓ 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명

  ✓ 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소

  user program이 작동할 때 memory나 disk에 접근할 수 없는데, system call을 통해 kernel mode로 진입할 수 있으며 시스템 안정성, 보안을 지킬 수 있다. 또한 user는 시스템 내부 동작에 대해 걱정하지 않고 사용할 수 있다. User level에서 system call을 호출하면 interrupt handler를 통해 system call handler로 전달되어 kernel API로 넘어갈 수 있고, return 값을 레지스터에 저장해서 다시 user level로 돌아가게 된다.

  - halt(): shutdown_power_off()를 호출해 Pintos를 종료한다.

  - exit(): 현재 실행하고 있는 user program을 종료하고 상태를 kernel로 반환한다.

  - exec(): process_execute()를 호출해 process를 실행한다.

  - wait(): child thread ID가 유효한 경우 process_wait()를 호출해 child process가 끝날 때까지 기다린다.

  - write(stdout): stdout에서 정해진 크기 만큼 출력한다.

  - read(stdin): stdin에서 정해진 크기 만큼 읽어들인다.

  - fibonacci(): 인자로 n을 받으면 n번째 피보나치 수를 구한다.

- max_of_four_int(): 정수 4개 중 가장 큰 수를 찾는다.

3. 추진 일정 및 개발 방법

A. 추진 일정

10/10-10/26: 프로젝트의 목표와 기존 코드의 흐름을 파악한다.

10/27: argument passing 구현

10/28: user memory access, system call 구현

10/29: system call 구현

10/30: 오류 검토 및 코드 정돈

10/31: 보고서 작성

B. 개발 방법

a) argument passing

src/userprog/process.c의 load() 함수에 argument를 parsing하는 코드를 구현하고, stack을 쌓는 construct_stack() 함수를 구현하고 호출한다.

b) user memory access

src/userprog/exception.c의 page_fault() 함수에 메모리 범위를 확인하는 조건문을 추가해 범위 밖으로 접근하려고 한다면 종료한다.

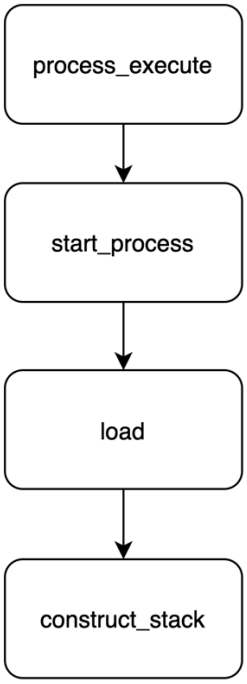
c) system call

src/userprog/syscall.c에 syscall_handler() 함수를 구현해 switch 문으로 각 system call을 처리해준다. sys_halt(), sys_exit(), sys_exec(), sys_wait(), sys_write(), sys_read(), sys_fibonacci(), sys_max_of_four_int()를 구현하여 case에 따라 호출하고, 각 함수의 prototype을 src/userprog/syscall.h에 선언한다. src/lib/syscall-nr.h의 enum 부분에 FIBONACCI, MAX_OF_FOUR_INT를 추가해준다. fibonacci(), max_of_four_int()의 경우 src/lib/user/syscall.h, src/lib/user/syscall.c에서 코드를 적절하게 추가해주고, argument가 4개인 경우도 처리할 수 있도록 syscall4도 작성한다.

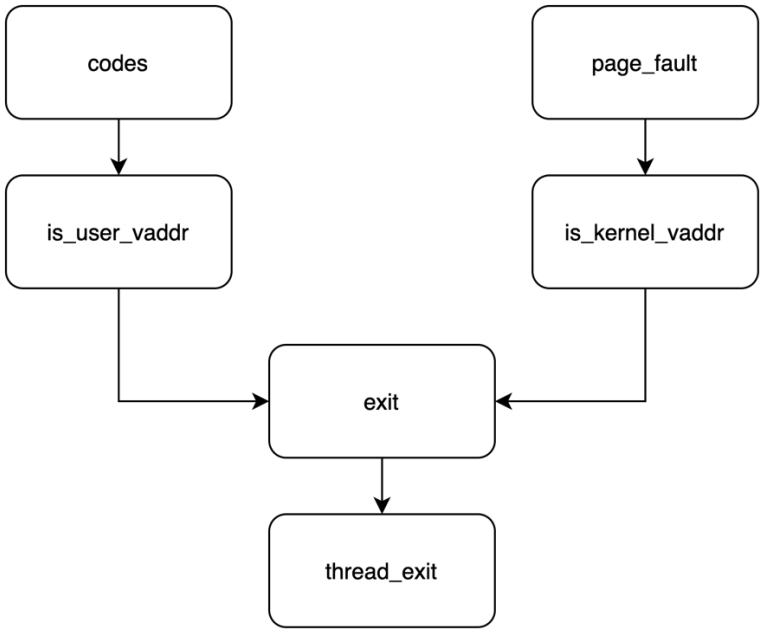
4. 연구 결과

A. Flow Chart

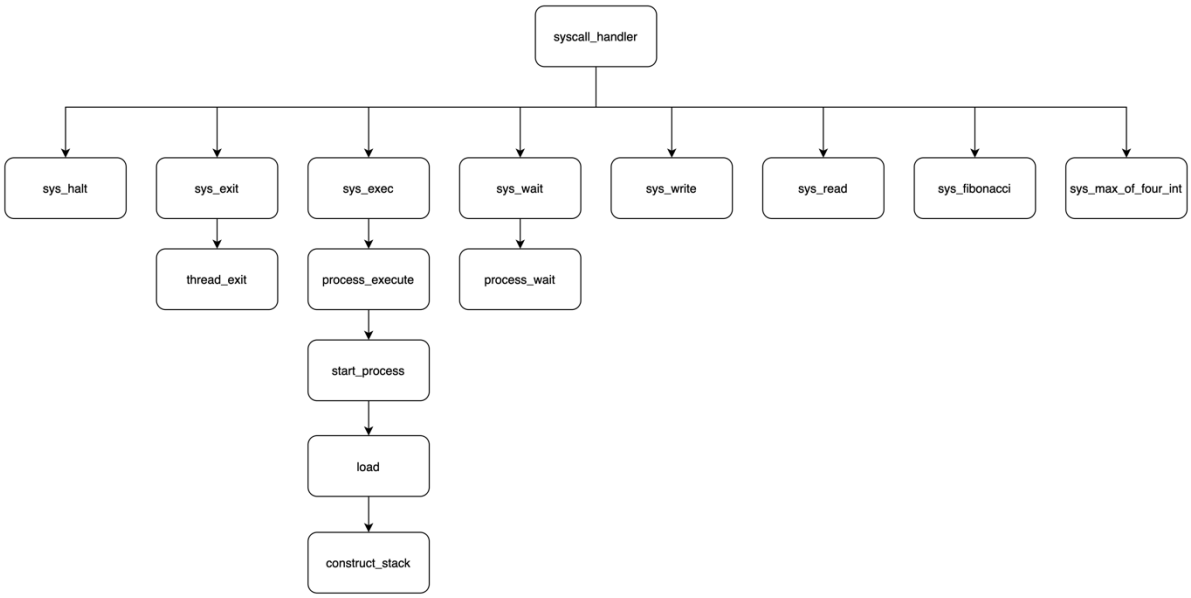
1. Argument Passing



2. User Memory Access



3. System Calls



B. 제작 내용

1. Argument Passing

src/userprog/process.c의 load() 함수에 argument를 parsing하는 코드를 구현하고, stack을 쌓는 construct_stack() 함수를 구현하고 호출했다. parsing은 strtok_r() 함수를 활용해 argv[]에 저장해주었고, 그에 맞춰 argc도 늘려주었다. construct_stack()에서는 esp를 감소시켜가며 80x86 calling convention에 맞게 stack에 push해주었다.

src/userprog/process.c

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{
  . . .
  /* ---------------------------- */
  //argument parsing
  char fn_copy[129];
  strlcpy(fn_copy, file_name, 129);

  char *argv[50];
  char *nxt_ptr;
  int argc = 0;

  char* ptr = strtok_r(fn_copy, " ", &nxt_ptr);
  char* pure_file_name = ptr;
  while (ptr != NULL){
```

```
    argv[argc] = ptr;
    ptr = strtok_r(NULL, " ", &nxt_ptr);
    argc++;
  }
  /* ============================ */

  . . .

  /* ============== */
  /* Construct stack. */
  construct_stack(argc, argv, esp);
  /* ============== */

  . . .
}
```

```
void construct_stack(int argc, char** argv, void **esp) {
  int i;
  int total_size = 0;
  int size;
  char* argv_addr[50];

  //argument
  for (i = argc - 1; i >= 0; i--) {
    size = strlen(argv[i]) + 1;
    *esp = *esp - size;
    total_size = total_size + size;
    strlcpy(*esp, argv[i], size);
    argv_addr[i] = *esp;
  }

  //word alignment
  if (total_size % 4 != 0) {
    *esp = *esp - (4 - total_size % 4);
  }

  //NULL : 80x86 calling convention
  *esp = *esp - 4;
  **(uint32_t**)esp = 0;

  //addresses of arguments
  for(i = argc - 1; i >= 0; i--) {
    *esp = *esp - 4;
    **(uint32_t**)esp = (uint32_t)argv_addr[i];
  }

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

  //argc
  *esp = *esp - 4;
```

```
  **(uint32_t**)esp = argc;

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

hex_dump()를 사용해 argument passing이 잘 되는지 확인할 수 있었다. 이후 구현 과정에서도 stack의 내용 파악을 위해 hex_dump()를 용이하게 사용했다.

2. User Memory Access

src/userprog/exception.c에서 is_kernel_vaddr() 함수를 이용해 invalid한 memory access를 막아주는 조건문을 추가했다. src/userprog/syscall.c에는 is_user_vaddr() 함수를 이용해 invalid한 memory access를 막아주고, valid한 경우에만 작동할 수 있도록 했다. is_kernel_vaddr(), is_user_vaddr()는 threads/vaddr.h에 정의된 함수로, 전자는 kernel virtual address일 경우 true, 후자는 user virtual address일 경우 true를 반환하는 점을 이용했다.

src/userprog/exception.c

```
static void
page_fault (struct intr_frame *f)
{
  . . .

  /* ================================================== */
  /* Project 1: Protect user memory accesses from system calls.
*/
  if (!(user && is_kernel_vaddr(fault_addr)))
     sys_exit(-1);
  /* ================================================== */

  . . .
}
```

3. System Calls

비어있던 syscall_handler() 내부에 코드를 작성했다.

src/userprog/syscall.c

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
    /* Project 1: system call impelementation */
    switch (*(uint32_t*)(f->esp)) {
    case SYS_HALT:
        sys_halt();
        break;

    case SYS_EXIT:
        check_valid(f->esp + 4);
        sys_exit(*(uint32_t *)(f->esp + 4));
        break;

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

    case SYS_WAIT:
        check_valid(f->esp + 4);
        f->eax = sys_wait(*(uint32_t *)(f->esp + 4));
        break;

    case SYS_WRITE:
        check_valid(f->esp + 20);
        check_valid(f->esp + 24);
        check_valid(f->esp + 28);
        f->eax = sys_write(*(int*)(f->esp + 20), *(void**)(f-
>esp + 24), *(unsigned*)(f->esp + 28));
        break;

    case SYS_READ:
        check_valid(f->esp + 20);
        check_valid(f->esp + 24);
        check_valid(f->esp + 28);
        f->eax = sys_read(*(int*)(f->esp + 20), *(void**)(f->esp
+ 24), *(unsigned*)(f->esp + 28));
        break;

    case SYS_FIBONACCI:
        f->eax = sys_fibonacci(*(int*)(f->esp + 4));
        break;

    case SYS_MAX_OF_FOUR_INT:
```

```
        f->eax = sys_max_of_four_int(*(int*)(f->esp + 4),
*(int*)(f->esp + 8), *(int*)(f->esp + 12), *(int*)(f->esp +
16));
        break;

    default:
        break;
    }

}
```

switch문의 case는 아래와 같이 syscall-nr.h에 선언되어있는 enum을 참고해 작성했다. 각 case에서는 pointer가 valid한지 체크하고 각각 수행할 함수를 호출했다. 스택에 접근할 때는 esp를 사용했고 return 값은 eax로 넘겨줄 수 있도록 했다.

src/lib/syscall-nr.h

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

    . . .
  };
```

다음은 is_user_vaddr()를 이용해 invalid memory access를 검사하는 함수이다.

src/userprog/syscall.c

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

- halt

halt system call은 shutdown_power_off()를 호출하도록 했다.

src/userprog/syscall.c

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

- exit

exit system call은 현재 thread의 이름, 상태를 출력하고 thread_exit()을 호출해 종료하도록 했다.

src/userprog/syscall.c

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

- exec

exec system call은 process_execute()을 실시하도록 했다.

src/userprog/syscall.c

```
tid_t sys_exec(const char *arg) {
    return process_execute(arg);
}
```

process_execute()은 src/userprog/process.c에 있다.

src/userprog/process.c의 process_execute() 함수 내에 다음과 같이 코드를 추가해 filesys_open()의 인자를 strtok_r()로 구했다.

src/userprog/process.c

```
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);

  /* ==========추가한 부분============ */
  char command[256];
  char *ptr;
  strlcpy(command, file_name, strlen(file_name) + 1);
  char *thread_name = strtok_r(command, " ", &ptr);
  /* ==========추가한 부분============ */

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

  return tid;
}
```

이후 start_process()가 실행되면 load()가 수행되어 1. argument passing에서 구현한 대로 esp를 이용해 stack에 push하게 된다.

- wait

process_wait()를 호출한다.

src/userprog/syscall.c

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

process_wait()는 src/userprog/process.c에 있다. process_wait()는 thread TID가 종결할 때까지 기다렸다가 그 exit status를 반환한다.커널에 의해 종료된 경우 -1를 반환한다.

src/userprog/process.c

```
int
process_wait (tid_t child_tid UNUSED)
{
    /* ====================== */
    struct list_elem* child_e;
    struct thread* child_t;
    int exit_status;

    child_e = list_begin(&(thread_current()->child));
    while (child_e != list_end(&(thread_current()->child))) {
        child_t = list_entry(child_e, struct thread,
child_elem);

        if (child_t->tid == child_tid) {
            sema_down(&(child_t->child_sema));
            list_remove(&(child_t->child_elem));
            sema_up(&(child_t->mem_sema));
            exit_status = child_t->exit_status;
            return exit_status;
        }

        child_e = list_next(child_e);
    }

    return -1;
    /* ====================== */
}
```

위와 같이 기다리게 하기 위해서 threads/thread.h의 thread 구조체에 다음과 같이 semaphore, list, list_elem, exit_status 변수를 추가해줬다.

src/threads/thread.h

```
struct thread
  {
    . . .

    /* ============================ */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct semaphore child_sema;
    struct semaphore mem_sema;
    struct list child;
```

```
    struct list_elem child_elem;
    int exit_status;
#endif
    /* ============================ */

    . . .
  };
```

- write

write system call은 stdout 즉 file descriptor가 1인 경우에 대해서만 구현했고, pintos/src/lib/kernel/console.c의 putbuf() 함수를 호출해 size만큼 출력했다.

src/userprog/syscall.c

```
int sys_write(int fd, const void *buffer, unsigned size) {
    if (fd == 1) {
        putbuf((char*)buffer, (size_t)size);
        return (int)size;
    }

    return -1;
}
```

- read

read system call은 stdin 즉 file descriptor가 0인 경우에 대해서만 구현했고, pintos/src/devices/input.c의 input_getc() 함수를 호출해 size만큼 입력을 받았다.

src/userprog/syscall.c

```
int sys_read(int fd, const void *buffer, unsigned size) {
    if (fd == 0) {
        for (int i = 0; i < (int)size && buffer != '\0'; i++)
            buffer = input_getc();

        return (int)size;
    }

    return -1;
}
```

4. Additional System calls

src/lib/syscall-nr.h의 enum 부분에 FIBONACCI, MAX_OF_FOUR_INT를 추가해주었다.

src/lib/syscall-nr.h

```
/* System call numbers. */
enum
  {
    /* Projects 2 and later. */
    . . .
    SYS_FIBONACCI,
    SYS_MAX_OF_FOUR_INT,
    . . .
  };
```

src/lib/user/syscall.h, src/lib/user/syscall.c에 코드를 작성해주고, argument가 4개인 경우를 처리할 수 있도록 syscall4도 작성했다. syscall4는 syscall1, syscall2, syscall3을 참고해 작성하였다.

src/lib/user/syscall.h

```
/* Projects 2 and later. */
. . .
int fibonacci(int n);
int max_of_four_int(const int a, const int b, const int c,
const int d);
. . .
```

src/lib/user/syscall.c

```
/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, ARG2
and
   ARG3, and returns the return value as an `int'. */
#define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3)
\
       ({                                              \
         int retval;                                    \
         asm volatile                                   \
           ("pushl %[arg3]; pushl %[arg2]; pushl %[arg1];
pushl %[arg0]; "    \
            "pushl %[number]; int $0x30; addl $20, %%esp"     \
             : "=a" (retval)                            \
             : [number] "i" (NUMBER),                     \
               [arg0] "r" (ARG0),                       \
               [arg1] "r" (ARG1),                       \
               [arg2] "r" (ARG2),                       \
               [arg3] "r" (ARG3)                        \
```

```
            : "memory");                                    \
  retval;                                                   \
})
```

```
int fibonacci(int n) {
    return syscall1(SYS_FIBONACCI, n);
}

int max_of_four_int(const int a, const int b, const int c,
const int d) {
    return syscall4(SYS_MAX_OF_FOUR_INT, a, b, c, d);
}
```

- fibonacci

0, 1, 1, 2, 3, 5, ...의 피보나치 수열에서 변수 2개를 이용해 n번째 수를 구했다.

src/lib/user/syscall.c

```
int sys_fibonacci(int n) {
    int curr = 0, next = 1;
    int temp;

    for (int i = 0; i < n; i++) {
        temp = next;
        next = curr + next;
        curr = temp;
    }

    return curr;
}
```

- max_of_four_int

파라미터 4개 중 가장 큰 정수를 찾아 리턴해줬다.

src/lib/user/syscall.c

```
int sys_max_of_four_int(const int a, const int b, const int c,
const int d) {
    int max1 = a > b ? a : b;
    int max2 = c > d ? c : d;

    int res = max1 > max2 ? max1 : max2;
    return res;
```

```
}
```

additional.c는 argument가 5개가 아니라면 usage를 출력하고 종료했으며 argument가 5개라면 fibonacci(), max_of_four_int()를 호출해 첫 번째 수(N)에 대해 N번째 fibonacci 수를 출력하고 네 개의 숫자 중 가장 큰 수를 출력했다.

src/examples/additional.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>

int main(int argc, char **argv) {
    if (argc != 5) {
        printf("Usage : additional [num 1] [num 2] [num 3] [num
4]\n");
        return EXIT_FAILURE;
    }

    int fibo_result = fibonacci(atoi(argv[1]));
    int max_result = max_of_four_int(atoi(argv[1]),
atoi(argv[2]), atoi(argv[3]), atoi(argv[4]));

    printf("%d %d\n", fibo_result, max_result);

    return EXIT_SUCCESS;
}
```

또한 Makefile의 양식을 참고해 다음과 같이 additional도 추가해주었다.

src/examples/Makefile

```
. . .
PROGS = cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm
shell \
    bubsort lineup matmult recursor additional

# Should work from project 2 onward.
cat_SRC = cat.c
cmp_SRC = cmp.c
cp_SRC = cp.c
echo_SRC = echo.c
halt_SRC = halt.c
hex-dump_SRC = hex-dump.c
lineup_SRC = lineup.c
ls_SRC = ls.c
recursor_SRC = recursor.c
rm_SRC = rm.c
additional_SRC = additional.c

. . .
```

additional system call을 구현한 뒤에 `pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run 'additional 10 20 62 40'` 명령어로 테스트하기 위해서 make를 실행해 볼 때 additional.c에서 fibonacci(), max_of_four_int() 함수를 찾지 못 하여 make에 실패를 겪었는데, 위에서 구현한 다른 system call과 달리 src/lib/user/syscall.h, src/lib/user/syscall.c에 적절하게 함수를 선언해주어야 했다.

C. 시험 및 평가 내용

- fibonacci 및 max_of_four_int 시스템 콜 수행 결과를 캡처하여 첨부.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)
Booting from Hard Disk...
PPiiLLoo  hhddaa1
1
LLooaaddiinngg.....................
Kernel command line: -f -q extract run 'additional 10 20 62 40'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  416,563,200 loops/s.
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 197 sectors (98 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 118 sectors (59 kB), Pintos scratch (22)
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'additional' into the file system...
Erasing ustar archive...
Executing 'additional 10 20 62 40':
55 62
additional: exit(0)
Execution of 'additional 10 20 62 40' complete.
Timer: 60 ticks
Thread: 3 idle ticks, 58 kernel ticks, 0 user ticks
hda2 (filesys): 63 reads, 240 writes
hda3 (scratch): 117 reads, 2 writes
Console: 900 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
cse20181464@cspro9:~/pintos/src/userprog$
```