Lab2 Design Report

Member #1
Name: Suwon Yoon
Student ID: 20210527

Member #2
Name: Jiwon Lee
Student ID: 20210706

1. process execution 절차 분석

- init.c/main

  read_command_line 함수와 parse_options 함수를 사용하여 command line으로 입력된 arguments를 break 한다. 그리고 run_actions 함수를 실행시킨다. run_actions 함수의 구현을 살펴보면 과제 2에서는 run_task 함수가 실행된다.

- run_task(argv)

  이 함수에서 process_execute 함수와 process_wait 함수를 차례로 호출한다. process_wait 함수는 process execution 이라는 맥락에서는 크게 중요한 부분은 아니다. 따라서 다른 조금 뒤에 살펴볼 것이고 우선 process_execute 함수에 대해서 살펴보겠다.

- process_execute 함수

  이 함수의 인자로는 file_name이라는 변수명이 전달되는데 이 변수는 const char * 타입이다. pintos 코드의 흐름상 이 변수에 user command line이 저장되어 있다. 예시를 들어보면 “ls - l foo bar”과 같은 식이다.

  kernel virtual address에 single free page를 할당받고 그 시작 주소를 fn_copy로 한다. 그리고 single free page 할당에 문제가 생겼다면 TID_ERROR 를 리턴하며 함수가 종료된다. 문제가 생기지 않았다면 fn_copy에 file_name을 복사한다.

다음으로 매개변수로 들어온 file_name 이라는 이름으로 thread를 create한다. 이때 thread가 실행할 함수로 start_process를, 그 함수의 인자로 fn_copy를 지정한다. 지난 과제에서 설명하기는 했지만 중요한 부분이므로 다시 한 번 풀어서 설명하자면 TCB를 구성하고 fake stack frame을 구성하여 언젠가 스케줄링 될 때 start_process(fn_copy) 이 실행된다.

thread의 생성이 정상적으로 마무리 되었다면 tid를 리턴하며 process_execute 함수는 종료된다. 하지만 thread의 생성이 올바르게 이루어지지 않았다면, fn_copy가 차지하고 있는 메모리를 free 시키고 TIE_ERROR를 리턴한다.

한 가지 짚고 넘어갈 부분은 thread_create 함수로 생성한 thread는 process_execute 함수가 종료되기 이전에 실행될 수도 있으며 심지어는 종료될 수도 있다는 것이다.

- thread create 함수에서 thread의 생성이 올바르게 이루어지지 않는 경우는 어떤 경우인가?

  TCB를 위한 메모리를 할당받지 못한 경우를 의미한다. 메모리는 palloc_get_page 함수를 활용하여 할당받는데, 이때 PAL_ZERO를 매개변수로 넘겨주어 kernel_pool에 메모리를 할당하도록 한다. 따라서 thread_create 함수의 설명에서와 같이 방금 생성한 thread는 kernel thread라는 것이 확인되었다.

  system memory는 kernel pool과 user pool, 두개의 pool로 구성되어 있다. user pool은 user VM pages를 위한 것이고 kernel pool은 그 밖의 것이다. pool은 bitmap으로 관리된다. 핀토스에서는 user pool과 kernel pool이 RAM의 절반씩을 차지하게 된다.

- start_process 함수

  이 함수는 process_execute 함수에서 thread_create 함수로 만든 thread가 스케줄링되어 실행될 때, 실행되는 함수로 user process를 load하고 실행시키는 역할을 한다.

  우선 file_name 변수에 ‘process_execute 함수에서 할당한 페이지의 시작 주소’를 저장한다.(fn_copy를 활용하여 저장함.)

  그 후 interrupt frame을 user process의 실행에 어울리게 초기화하고 load 함수를 활용하여 executable을 load 한다. load 함수의 실행을 마치고 돌아오면, file_name은 더 이상 사용할 필요가 없으므로 file_name이 가리키고 있는 주소의 메모리를 free한다.

  만약 load가 실패하였었다면 thread_exit 함수를 호출하여 thread를 제거하는 절차를 밟는다. 이때 과제 2에서는 thread_exit 함수에서 process_exit 함수를 호출하는데 이는 page

dircetory와 같은 current process의 resources를 할당해제하고 page directory를 init_page_dir로 변경하기 위함이다.

load 함수가 실패하여 thread_exit 함수가 호출된 경우라면 thread_exit 함수의 마지막에 schedule 함수가 호출되므로 다른 thread가 실행되는 것으로 마무리 될 것이므로 start_process 함수에서는 살펴볼 것이 없다.

load 함수가 성공한 경우에는 ‘interrupt에서 return하는 것을 simulating’하는 것으로 user process가 실행된다. 이에 대해 자세히 알아보자.

- user process의 실행

  start_process 함수의 마지막 부분에서 실행하는 코드는 어셈블리어로 직접 코딩되어 있다. 어셈블리어를 해석해보면, %esp에 if_의 주소를 저장하고 intr_exit 함수로 jump한다는 것을 알 수 있다. 어떻게 이런 과정을 통해 user process를 실행하는 것이 되는지 살펴보기 위해서는 ‘interrupt frame의 구성’과 ‘intr_exit 함수’, 이 두 가지에 대한 이해가 필요하다.

  - intr_exit 함수

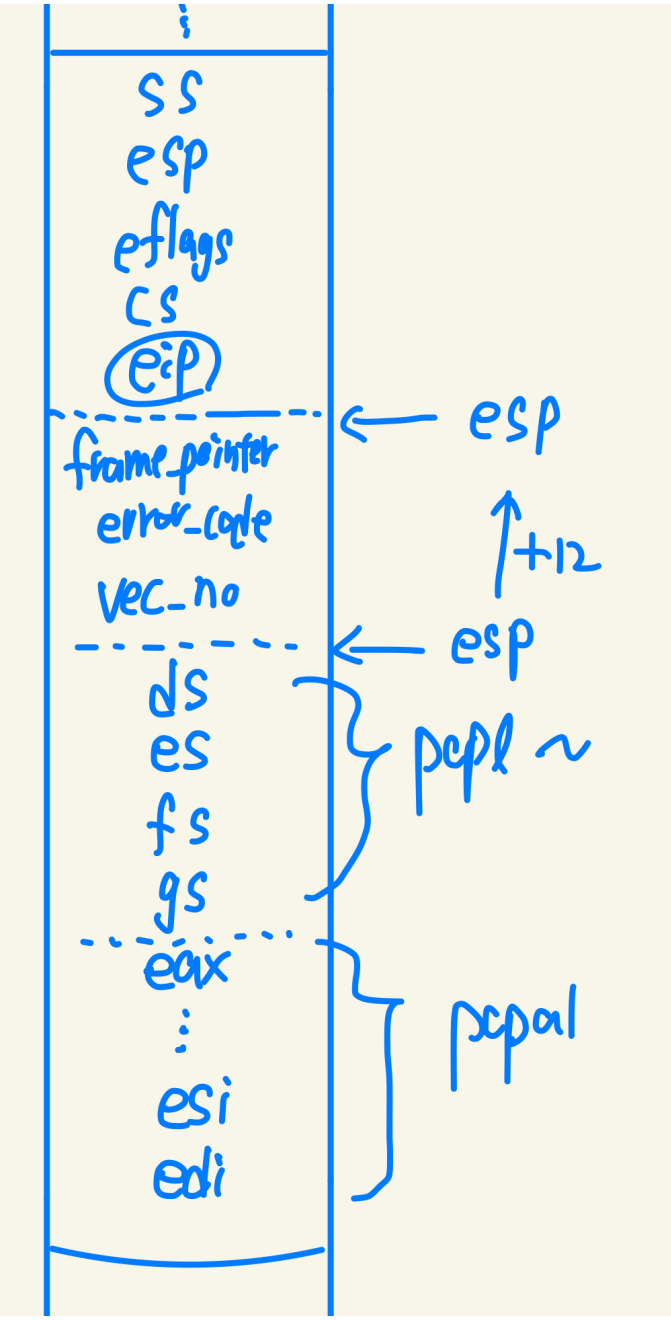
    interrupt frame이 어떤식으로 구성되는지 구조만 간략하게 살펴보자.

```
/* Interrupt stack frame. */
struct intr_frame
  {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no;            /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
       otherwise for consistency pushed as 0 by intrNN_stub.
       The CPU puts it just under `eip', but we move it here. */
    uint32_t error_code;        /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
       This frame pointer eases interpretation of backtraces. */
    void *frame_pointer;        /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
       These are the interrupted task's saved registers. */
    void (*eip) (void);         /* Next instruction to execute. */
    uint16_t cs, :16;           /* Code segment for eip. */
    uint32_t eflags;            /* Saved CPU flags. */
    void *esp;                  /* Saved stack pointer. */
    uint16_t ss, :16;           /* Data segment for esp. */
  };
```

intr_frame은 이와 같은 형태로 정의되어 있고 실제로 stack에 저장될 때는 edi가 stack의 상단에 있고 ss가 stack의 하단에 위치할 것이다.

핵심 내용을 살피기 위해 먼저 인터럽트 intr_exit 함수를 보겠다. 이 함수는 어셈블리어로 작성되어 있고 caller의 register를 restore하는 기능을 한다. 이러한 기능을 통해

user process를 처음 실행할 때는 restore를 가장하여 처음에 필요한 정보들을 register에 올려둘 것이다. 이를 위해서는 아래의 그림과 같이 stack에 정보가 들어있어야 한다.



popal 명령어를 통해 gnereal-purpose registers를 pop하고, 차례로 4개의 register를 추가로 pop한다. user program을 처음 실행할 때는 필요없는 정보가 intterupt frame에 들어있는데 이를 무시하고 iret 연산을 통해 파란 동그라미 표시를 해둔 eip에

적혀있는 주소로 jump 할 것이다. 그렇다면 eip에 user program의 첫 주소가 있어야 현재 우리의 목적인 ‘user program의 실행’을 달성할 수 있는 것이다. eip 값은 load 함수에서 설정 한다. load 함수를 살펴보자.

- load 함수: interrupt frame 구성

  load 함수는 const char * 타입으로 file_name, void (**) (void) 타입으로 eip, void ** 타입으로 esp를 받는다. 주요한 기능으로는 ELF executable 을 현재 thread로 load하는 것이 있다. 그리고 executable의 entry point를 *eip에 저장하고 executable의 초기 stack pointer를 *esp에 저장하기도 한다. 만약 모든 동작이 정상적으로 진행된다면 true를 리턴하고 아니라면 false를 리턴한다.

  load 함수를 호출한 thread(user program을 처음 실행시키는 현 상황에서 보면 이 thread는 process_execute 함수에서 thread_create 함수로 만든 file_name 이름의 thread임.)의 pagedir을 할당하고 활성화한다. pagedir의 ‘할당’과 ‘활성화’에 대해 더 알아보자.

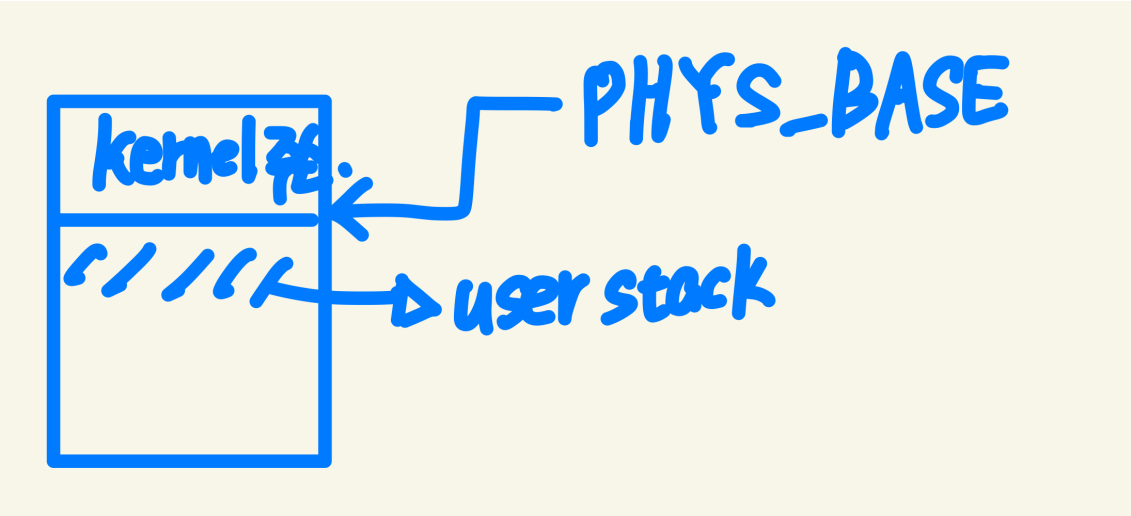
  할당을 위해서는 pagedir_create 함수가 사용되는데, 이 함수에서는 kernel virtual address의 mapping을 가진 새로운 page directory를 생성한다. 리턴할 값을 init_page_dir에서 memcpy하는 것을 통해 kernel virtual mapping을 가진 page directroy를 만들고 있음을 알 수 있다. user virtual address의 mapping은 없다.

  process_activate 함수에서 page directory를 활성화한다. page directory가 NULL이 아니라면 page dircetory의 주소를 %cr3 register에 저장한다. 이때 하드웨어상에서 page directory가 설정된 것이다. %cr3 register는 page directory base register를 저장하는 용도로 사용된다. 그 후에 tss_update 함수를 호출하여 tss→esp0을 현재 thread의 stack의 바닥으로 설정한다. 이로써 thread의 kernel stack을 set하였다. 이 kernel stack은 interrupt를 처리할 때 사용할 stack이다. process_activate 함수는 thread_schedule_tail 함수에서도 불리는 것을 보아 user process의 경우 스케줄링 할 때 page dircetory를 다시 설정해주고, kernel stack을 설정함을 알 수 있다.

  이후 filesys_open 함수를 통해 file_name 변수명과 일치하는 executable file을 찾고 executable header와 program header를 읽는다. 그 후 disk에서 segment를 읽고 virtual memory를 initialize한다.

  이제서야 유저 stack을 구성하는데, setup_stack 함수를 활용한다. 이 함수는 user VM의 top에 stack을 만든다. palloc_get_page 함수의 인자를 보면 user_pool을 사용하는 것을 알 수 있고 PHYS_BASE 에서 PGSIZE만큼을 뺀 주소를 인자로 하여 install_page 함수를 호출하는 것을 보아 아래 그림과 같이 user stack 공간을 할당해 둠을 알 수 있다. 이때 *esp 를 PHYS_BASE로 하여 stack을 초기 상태를 설정한다.

이 함수에서 설정하는 *esp는 현재 논의의 맥락에서 interrupt frame 안에 있는 esp이다.



마지막으로 *eip를 (void (*) (void)) 타입으로 케스팅한 ehdr.e_entry 값으로 설정하는 것을 통해 ELF executable의 header에 기록되어 있는 주소로 시작 주소를 설정하는 것을 볼 수 있다. 이 설정을 해 둠으로써 interrupt frame의 eip가 user program의 시작 주소로 설정되어 이후 user program이 잘 실행되는 것이다.

파일에서 필요한 정보를 모두 사용하였으므로 file_close를 통해 파일을 닫아준다.

- user process의 실행 정리
  1. ‘interrupt에서 return하는 것을 simulating’ 하는 것으로 user process가 실행시킴.
  2. 이 simulating을 위해 interrupt frame을 적절히 구성해야하고 이는 start_process 함수 내부에서 조금, load 함수에서 대부분을 함.
  3. 특히 load 함수에서는 eip와 esp 값을 설정하는 중요한 역할을 함.
  4. start_process 함수에서 intr_exit 함수를 호출하며 interrupt에서 return하는 것을 simulating하여 user process를 시작함.

- process_wait 함수

  Pintos 운영 체제의 process_wait() 함수는 부모 프로세스가 자식 프로세스를 어떻게 기다리고 어떻게 종료 상태를 검색하는지 관리한다.

process_wait() 함수의 주요 목표는 pid로 식별되는 자식 프로세스를 부모 프로세스가 기다리게 하고, 자식이 종료되면 종료 상태를 검색하는 것이다. pid를 가진 자식 프로세스가 아직 실행 중이면, 부모는 자식이 종료될 때까지 기다려야 한다. 종료되면, 함수는 자식 프로세스가 exit() 함수를 호출하여 전달한 상태를 반환해야 한다. 하지만 만약 자식 프로세스가 exit()을 호출하지 않고 커널에 의해 종료되었으면, process_wait() 함수는 1을 반환해야 한다. 만약 부모 프로세스가 이미 종료된 자식에 대해 wait()를 호출했다면, 커널은 여전히 부모 프로세스가 자식 프로세스의 종료 상태를 받거나 확인할 수 있도록 해야한다.

구체적으로 이에 대한 구현을 위해서는 세마포어를 활용할 것이다. 수업시간에 배운 내용을 활용하여 0으로 초기화된 세마포어를 부모에서 sema_down하고 이후 자식이 끝나면 sema_up을 하면 “자식이 끝난 후, 부모가 실행”되는 순서를 보장할 수 있다. (자식이 먼저 sema_up을 해도 동일한 동작이 된다.) 따라서 세마포어를 위한 변수를 struct thread에 추가해야겠다.

또한 부모와 자식 관계를 나타낼 수 있는 변수 또한 struct thread에 추가할 필요가 있다. 부모 프로세스는 여러 자식 프로세스를 만들 수 있기 때문에 struct list로 관리해야겠다. 자식 프로세스를 위한 구조체를 따로 만드는 것도 방법일 수도 있고 기존의 struct thread에 덮어서 사용하는 것도 방법이겠으나 이는 실제 구현에서 더 편리한 것으로 할 생각이다.

2. system call 절차 분석

우리는 첫 번째 프로젝트를 통해 user program으로부터 OS에게 control을 넘기는 경우인 external interrupt를 다루었다.

이번에 진행되는 두 번째 프로젝트에서는 user program이 OS에게 직접 특정한 서비스를 요청할 수 있는 수단인 system call을 다룬다.

Pintos에서 user program은 OS에게 제어를 넘기며 특정한 서비스를 요청하기 위해 각종 system call을 호출한다. system call 번호와 추가 인수는 stack에 push되고, 그런 다음 interrupt가 호출된다. syscall_handler()가 실행될 때 32비트 system call 번호는 caller의 stack pointer에 저장되며, 32비트 첫 번째 인수는 다음 높은 주소에 저장된다. syscall_handler()는 caller의 stack pointer에 esp register를 통해 접근할 수 있으며, 이는 struct intr_frame의 멤버로 전달된다. 반환 값은 eax register에 저장된다. 따라서 값을 반환하는 시스템 호출은 반환 값을 struct intr_frame의 eax 멤버에 저장함으로써 수행할 수 있다.

여기서 활용되는 system call 번호들은 lib/syscall-nr.h에 enum으로 정의되어 있다.

```
/* 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. */

    /* Project 3 and optionally project 4. */
    SYS_MMAP,                   /* Map a file into memory. */
    SYS_MUNMAP,                 /* Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,                  /* Change the current directory. */
    SYS_MKDIR,                  /* Create a directory. */
    SYS_READDIR,                /* Reads a directory entry. */
    SYS_ISDIR,                  /* Tests if a fd represents a directory. */
    SYS_INUMBER                 /* Returns the inode number for a fd. */
  };

#endif /* lib/syscall-nr.h */
```

그리고 이러한 syscall을 호출하기 위해서 사용되는 macro들은 lib/user/syscall.c에 정의되어 있다.

```
/* Invokes syscall NUMBER, passing no arguments, and returns the
   return value as an `int'. */
#define syscall0(NUMBER)                                        \
        ({                                                      \
          int retval;                                           \
          asm volatile                                          \
            ("pushl %[number]; int $0x30; addl $4, %%esp"       \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER)                          \
               : "memory");                                     \
          retval;                                               \
        })

/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                          \
        ({                                                              \
          int retval;                                                   \
          asm volatile                                                  \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
               : "=a" (retval)                                          \
               : [number] "i" (NUMBER),                                 \
                 [arg0] "g" (ARG0)                                      \
               : "memory");                                             \
          retval;                                                       \
        })

/* Invokes syscall NUMBER, passing arguments ARG0 and ARG1, and
   returns the return value as an `int'. */
#define syscall2(NUMBER, ARG0, ARG1)                            \
        ({                                                      \
          int retval;                                           \
          asm volatile                                          \
            ("pushl %[arg1]; pushl %[arg0]; "                   \
             "pushl %[number]; int $0x30; addl $12, %%esp"      \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER),                         \
                 [arg0] "r" (ARG0),                             \
                 [arg1] "r" (ARG1)                              \
               : "memory");                                     \
          retval;                                               \
        })
```

그리고 이제 아래에 서술될 흐름을 이해하기 위해 필요한 interrupt 관련 함수들을 설명해보도록 하겠다.

enum intr_level intr_get_level (void)

현재 인터럽트 상태를 반환한다.

enum intr_level intr_set_level (enum intr_level level)

level로 지정된 현재 인터럽트 상태를 설정한다.

enum intr_level intr_enable (void), enum intr_level intr_disable (void)

인터럽트를 활성화하고 비활성화하며 이전 인터럽트 상태를 반환한다.

void intr_init (void)

threads/init.c에 정의된 메인 함수에서 호출되며 interrupt 시스템을 초기화한다. interrupt controller와 Interrupt Descriptor Table을 초기화한다. 각 interrupt는 IDT에 정의된 고유한 핸들러를 가진다. intr_init의 나머지 코드는 IDT 레지스터를 로드하고 intr_names를 초기화한다.

static void register_handler (uint8_t vec_no, int dpl, enum intr_level level, intr_handler_func *handler, const char *name)

디스크립터 권한 레벨 dpl로 핸들러를 호출하여 interrupt vec_no를 등록한다. interrupt number는 0에서 0xFF 사이다.

void intr_register_ext (uint8_t vec_no, intr_handler_func *handler, const char *name)

register_handler를 호출하여 외부 인터럽트 vec_no를 등록하고, 디버깅 목적으로 name이라는 이름을 가진 핸들러를 호출한다. 핸들러는 인터럽트가 비활성화된 상태에서 실행된다. 즉 외부 인터럽트 핸들러가 실행되려면 인터럽트가 비활성화되어야 한다.

void intr_register_int (uint8_t vec_no, int dpl, enum intr_level level, intr_handler_func *handler, const char *name)

register_handler를 호출하여 내부 인터럽트 vec_no를 등록하고, 핸들러를 호출한다. 내부 핸들러는 인터럽트가 활성화된 상태에서 실행될 수 있다.

bool intr_context (void)

외부 인터럽트 동안 true를 반환하고 그렇지 않으면 false를 반환한다.

void intr_yield_on_return (void)

외부 인터럽트 중에 인터럽트 핸들러가 인터럽트에서 반환되기 직전에 cpu를 양보하도록 지시하여 새 프로세스가 실행될 수 있게 한다.

static void pic_init (void)

PIC, programmable interrupt controller를 초기화한다.

static void pic_end_of_interrupt (int irq)

주어진 IRQ, 인터럽트 요청에 대해 PIC에 인터럽트 종료 신호를 보낸다.

static uint64_t make_gate (void (*function) (void), int dpl, int type)

function을 호출하는 게이트를 만들고, 게이트는 디스크립터 권한 레벨 dpl을 가진다. 실제로, dpl=3은 사용자 모드가 게이트를 호출할 수 있음을 의미하고, dpl=0은 그러한 호출을 방지한다.

static uint64_t make_intr_gate (void (*function) (void), int dpl)

make_gate를 호출하여 dpl로 함수를 호출하는 인터럽트 게이트를 만든다.

static uint64_t make_trap_gate (void (*function) (void), int dpl)

make_gate를 호출하여 dpl로 함수를 호출하는 인터럽트 trap을 만든다.

void intr_handler (struct intr_frame *frame)

이 함수는 모든 인터럽트, 오류, 예외에 대한 핸들러이며 intr-stubs.S의 인터럽트 스텁에서 호출된다. 프레임은 인터럽트와 인터럽트가 발생한 스레드의 레지스터를 설명한다. 외부 인터럽트는 한 번에 하나씩 실행되어야 한다. handler (frame)를 호출하여 인터럽트의 핸들러를 호출한다. 핸들러가 등록되어 있지 않은 경우, unexpected_interrupt (frame)를 호출하여 인터럽트를 처리한다.
외부 인터럽트의 나머지 부분을 처리 완료한다.

static void unexpected_interrupt (const struct intr_frame *f)

intr_frame f에 있는 예상치 못한 인터럽트를 처리한다.

void intr_dump_frame(const struct intr_frame *f)

intr_frame f를 콘솔에 덤프한다.

const char *intr_name (uint8_t vec)

interrupt vec의 이름을 반환한다.

위에 언급된 assembly 파일인 intr-stubs.S에 정의된 함수들에 대해서도 설명하겠다.

intr_entry

내부 또는 외부 인터럽트는 intrNN_stub 루틴 중 하나에서 시작된다. 여기서 NN은 interrupt number를 16진수로 나타낸 것이다. 이 루틴은 스택에 struct intr_frame, error_code 및 vec_no members를 push한다. 그런 다음 이 함수로 점프한다. 이 함수는 processor가 이미 push하지 않은 모든 struct intr_frame members를 스택에 push한다. 마지막으로 kernel의 일부 register 값을 설정하고 intr_handler()를 호출하여 interrupt를 실제로 처리한다.

intr_exit

호출자의 registers를 복원하고 스택의 extra data를 버린 후 호출자에게 반환한다.

intr_stubs

이것은 256개의 code fragments를 정의한다. 각 조각은 intr00_stub부터 intrff_stub까지 명명되며, 해당 interrupt의 entry point로 사용된다. 이러한 functions의 주소를 intr_stubs라는 function pointers 배열의 올바른 위치에 넣는다.

우리가 통과해야하는 테스트를 예시로 전체적인 흐름을 정리해보도록 하겠다.

1) qemu

우리가 test를 돌리고자 하면 가장 먼저 실행되는 건 qemu이다. qemu는 에뮬레이터로, linux 환경 내에서 또 하나의 OS인 pintos를 사용할 수 있도록 가상화를 해준다.

2) int main()

qemu가 실행되면 바로 Pintos가 부팅 과정을 거치게 된다. Pintos 부팅의 시작은 init.c를 실행하는 것으로 시작한다.

```
/* Pintos main program. */
int
main (void)
{
  char **argv;

  /* Clear BSS. */
  bss_init ();

  /* Break command line into arguments and parse options. */
  argv = read_command_line ();
  argv = parse_options (argv);
```

int main()을 보면 argv = read_command_line(); 이라는 코드를 볼 수 있다. 이 **read_command_line()** 함수에서 test를 위해 입력된 command line을 읽어 들인다.

3) thread_init()

thread_init()은 thread를 초기화하는 함수로, 여기서 main thread가 실행된다.

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);

  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread ();
  init_thread (initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid ();
}
```

thread_init() 함수를 보면 init_thread()함수에 "main"이라는 이름을 인자로 넣어 initial_thread의 이름을 설정 해주는 것을 확인 할 수 있다.

```
/* Does basic initialization of T as a blocked thread named
   NAME. */
static void
init_thread (struct thread *t, const char *name, int priority)
{
  enum intr_level old_level;

  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);

  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;
  t->magic = THREAD_MAGIC;

  old_level = intr_disable ();
  list_push_back (&all_list, &t->allelem);
  intr_set_level (old_level);
}
```

init_thread()에서는 전달 받은 thread의 상태와 메모리를 세팅해준다. 그 다음엔 스택의 가장 아래 위치인 t를 사용해서 스택의 가장 끝 영역, kernel stack pointer를 계산한다. 이를 thread 내 멤버인 stack에 kernel stack pointer 위치를 저장한다.

4) tss_init()

여기서 USERPROG가 정의 되었을 때만 실행되는 과정이 있다. 바로 tss_init()이다.

```
  /* Segmentation. */
#ifdef USERPROG
  tss_init ();
  gdt_init ();
#endif
```

tss_init()은 task state segment를 초기화하는 함수이다. Task state segment는 hardware context switching을 위한 특별한 데이터 구조로, 프로세서가 현재 실행 중인 task의 status 정보들을 저장하는데 사용된다. 이 중에서 kernel stack address 또한 저장하는데, 이는 사용자 모드에서 커널 모드로 전환할 때 사용된다.

```
/* Initializes the kernel TSS. */
void
tss_init (void)
{
  /* Our TSS is never used in a call gate or task gate, so only a
     few fields of it are ever referenced, and those are the only
     ones we initialize. */
  tss = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  tss->ss0 = SEL_KDSEG;
  tss->bitmap = 0xdfff;
  tss_update ();
}
```

이게 필요한 이유는 간단하다. user process에서 system call을 호출했을 경우 interrupt handler가 실행된다. 이때 kernel stack에 지금까지 user process에서 완료한 task들을 쌓아야 하는데, 이때 kernel stack의 위치를 찾을 필요가 없도록 tss에 user process에 대응하는 kernel process에 대해 해당 kernel stack pointer의 끝을 가리키게 해놓는다.

```
/* Sets the ring 0 stack pointer in the TSS to point to the end
   of the thread stack. */
void
tss_update (void)
{
  ASSERT (tss != NULL);
  tss->esp0 = (uint8_t *) thread_current () + PGSIZE;
}
```

tss_init()에서 실행되는 함수인 tss_update()를 보면 tss의 멤버 esp0에다가 currernt thread의 주소 + PGSIZE를 더해서 저장하는 것을 볼 수 있다. 여기서 thread_current()는 부팅 작업을 하고 있는 kernel thread이기에, 여기다가 페이지 크기만큼 더하면 커널 페이지의 끝 부분을 가리키게 된다.

여기서 tss_update에 들어가는 thread는 thread_current()로 부팅 작업을 하고 있는 커널 스레드이다. 즉, 커널 스레드 주소의 시작 부분에 페이지 크기만큼 더하면 커널 페이지의 끝 부분을 가리킨다는 것을 알 수 있다.

5) exception_init, syscall_init

```
#ifdef USERPROG
  exception_init ();
  syscall_init ();
#endif
```

main 함수는 그 다음엔 exception_init()과 syscall_init()를 호출한다. syscall_init()는 intr_register_int(0x30, 3, INTR_ON, syscall_handler, “syscall”)를 호출해서 syscall handler를 등록한다. syscall descriptor privilege level (=dpl) 값은 3이기 때문에, 이 예외는 user mode에서 호출할 수 있으며, internal exception이기 때문에 INTR_ON인 경우에도 호출할 수 있다.

exception_init()는 user programs에 의해 발생하는 interrupts의 핸들러를 등록한다. 이 핸들러는 주로 프로세스를 종료하는 역할을 한다.

intr_register_int() 함수는 register_handler를 호출하여 internal handler를 등록하는 작업을 수행한다. 이 때, 예외가 INTR_ON으로 호출될 수 있다면, make_trap_gate를 사용하여 trap gate를 생성하는 과정이 진행되지만, 그렇지 않은 경우에는 make_intr_gate를 활용한다. 이러한 과정들을 거치면서 intr_entry는 스택에 이미 저장된 정보들, 예를 들면 interrupt number나 frame pointer 같은 정보들을 바탕으로 interrupt 처리를 위해 intr_handler를 호출한다. 이를 더 자세히 살펴보면, threads/interrupt.c 내에 위치한 intr_handler는 실제 interrupt를 처리하는 핸들러로 작동하며, 특정 코드의 도움으로 핸들러 함수를 실행하는 방식으로 동작한다.

6) run_actions(), run_task()

int main() 함수를 계속 보면 다른 설정들이 전부 완료되고 나면 부팅이 완료되었다는 표시를 하는 것을 확인할 수 있다. 그 다음에 실행되는 것이 run_actions()이다.

```
printf ("Boot complete.\n");

/* Run actions specified on kernel command line. */
run_actions (argv);
```

```
/* Executes all of the actions specified in ARGV[]
   up to the null pointer sentinel. */
static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, including action name. */
      void (*function) (char **argv);   /* Function to execute action. */
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"extract", 1, fsutil_extract},
      {"append", 2, fsutil_append},
#endif
      {NULL, 0, NULL},
    };

  while (*argv != NULL)
    {
      const struct action *a;
      int i;

      /* Find action name. */
      for (a = actions; ; a++)
        if (a->name == NULL)
          PANIC ("unknown action `%s' (use -h for help)", *argv);
        else if (!strcmp (*argv, a->name))
          break;

      /* Check for required arguments. */
      for (i = 1; i < a->argc; i++)
        if (argv[i] == NULL)
          PANIC ("action `%s' requires %d argument(s)", *argv, a->argc - 1);

      /* Invoke action and advance. */
      a->function (argv);
      argv += a->argc;
    }

}
```

run_actions에서는 Pintos가 실행해야될 각종 actions들을 확인하고 argument들에 문제가 없다면 입력값에 맞는 run_task()를 실행한다.

```
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

run_task() 함수는 argv 배열의 두 번째 원소에서 실행할 task의 이름을 추출한다. task를 실행한다는 메세지를 출력한 다음 USERPROG가 정의되었다면 user program을 시작하기 위해

process_execute() 함수를 호출한다. 프로세스가 완료될 때 까지 완료 메세지가 출력이 되지 않도록 process_wait()함수를 사용해 현재 thread를 대기시킨다.

7) process_execute()

process_execute() 함수는 주어진 file_name에서 user program을 로드하여 새 thread를 시작한다.

```
/* Starts a new thread running a user program loaded from
   FILENAME.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   thread id, or TID_ERROR if the thread cannot be created. */
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);

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

여기서 함수는 실행할 사용자 프로그램의 이름을 나타내는 문자열을 인수로 받는다. 그 다음엔 palloc_get_page()함수를 사용하여 새 페이지를 fn_copy에 할당해준다. 인수로 받은 문자열을 fn_copy에 복사본을 저장한다. 이는 race를 방지하기 위한 조치이다.
이러한 사전 조치들이 이루어지면, 기본 우선순위로 주어진 file_name을 실행하기 위해 새 thread를 셍성한다. 프로세스를 시작하기 위해 thread_create() 함수의 인자로 start_process 함수가 전달되며, 복사된 파일 이름이 인수로 전달된다.

8) run_actions(), run_task()

int main() 함수를 계속 보면 다른 설정들이 전부 완료되고 나면 부팅이 완료되었다는 표시를 하는 것을 확인할 수 있다. 그 다음에 실행되는 것이 run_actions()이다.

```
  printf ("Boot complete.\n");

  /* Run actions specified on kernel command line. */
  run_actions (argv);
```

```
/* Executes all of the actions specified in ARGV[]
   up to the null pointer sentinel. */
static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, including action name. */
      void (*function) (char **argv);   /* Function to execute action. */
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
      {"ls", 1, fsutil_ls},
      {"cat", 2, fsutil_cat},
      {"rm", 2, fsutil_rm},
      {"extract", 1, fsutil_extract},
      {"append", 2, fsutil_append},
#endif
      {NULL, 0, NULL},
    };

  while (*argv != NULL)
    {
      const struct action *a;
      int i;

      /* Find action name. */
      for (a = actions; ; a++)
        if (a->name == NULL)
          PANIC ("unknown action `%s' (use -h for help)", *argv);
        else if (!strcmp (*argv, a->name))
          break;

      /* Check for required arguments. */
      for (i = 1; i < a->argc; i++)
        if (argv[i] == NULL)
          PANIC ("action `%s' requires %d argument(s)", *argv, a->argc - 1);

      /* Invoke action and advance. */
      a->function (argv);
      argv += a->argc;
    }

}
```

run_actions에서는 Pintos가 실행해야될 각종 actions들을 확인하고 argument들에 문제가 없다면 입력값에 맞는 run_task()를 실행한다.

```c
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

run_task() 함수는 argv 배열의 두 번째 원소에서 실행할 task의 이름을 추출한다. task를 실행한다는 메세지를 출력한 다음 USERPROG가 정의되었다면 user program을 시작하기 위해 process_execute() 함수를 호출한다. 프로세스가 완료될 때 까지 완료 메세지가 출력이 되지 않도록 process_wait()함수를 사용해 현재 thread를 대기시킨다.

9) process_execute()

process_execute() 함수는 주어진 file_name에서 user program을 로드하여 새 thread를 시작한다.

```c
/* Starts a new thread running a user program loaded from
   FILENAME.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   thread id, or TID_ERROR if the thread cannot be created. */
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);

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

여기서 함수는 실행할 사용자 프로그램의 이름을 나타내는 문자열을 인수로 받는다. 그 다음엔 palloc_get_page()함수를 사용하여 새 페이지를 fn_copy에 할당해준다. 인수로 받은 문자열을 fn_copy에 복사본을 저장한다. 이는 race를 방지하기 위한 조치이다.
이러한 사전 조치들이 이루어지면, 기본 우선순위로 주어진 file_name을 실행하기 위해 새 thread를 셍성한다. 프로세스를 시작하기 위해 thread_create() 함수의 인자로 start_process 함수가 전달되며, 복사된 파일 이름이 인수로 전달된다.

```
/* A thread function that loads a user process and starts it
   running. */
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;

  /* Initialize interrupt frame and load executable. */
  memset (&if_, 0, sizeof if_);
  if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
  if_.cs = SEL_UCSEG;
  if_.eflags = FLAG_IF | FLAG_MBS;
  success = load (file_name, &if_.eip, &if_.esp);

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

  /* Start the user process by simulating a return from an
     interrupt, implemented by intr_exit (in
     threads/intr-stubs.S).  Because intr_exit takes all of its
     arguments on the stack in the form of a `struct intr_frame',
     we just point the stack pointer (%esp) to our stack frame
     and jump to it. */
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

start_process() 함수는 사용자 프로세스를 실행하기 위한 설정 작업을 수행한다.

함수는 file_name_이라는 user program의 이름을 인수로 받고, 초기에 memset을 사용해 인터럽트 프레임 if_를 설정해준다. 이 초기화 과정을 마침 다음에는 load() 함수를 사용하여 file_name으로 지정된 user_program을 메모리에 로드해준다. 성공적으로 로드되면 프로그램의 진입점과 스택 포인터가 설정된다. 프로그램 로드에 실패하면 할당된 페이지를 해제하고 thread를 종료한다. 마지막으로, inline assembly 코드를 사용해서 스택 포인터를 interrupt frame인 if_로 설정하고 intr_exit 함수로 jump한다. 이는 stack CPU context를 설정하여 intr_exit이 끝나고 나면 user program을 실행시키는 과정을 위해서 마련되었다.

```
intr_exit:
        /* Restore caller's registers. */
        popal
        popl %gs
        popl %fs
        popl %es
        popl %ds

        /* Discard `struct intr_frame' vec_no, error_code,
           frame_pointer members. */
        addl $12, %esp

        /* Return to caller. */
        iret
.endfunc
```

intr_exit은 호출자의 register를 복원하고, stack에 있는 추가 데이터를 버리고, 호출자에게 반환되는 함수이다.

10) test run - args.c

실행할 수 있는 다양한 test들 중에서 args.c를 사용하는 것을 예시로 흐름을 마저 서술해보겠다.

```
int
main (int argc, char *argv[])
{
  int i;

  test_name = "args";

  msg ("begin");
  msg ("argc = %d", argc);
  for (i = 0; i <= argc; i++)
    if (argv[i] != NULL)
      msg ("argv[%d] = '%s'", i, argv[i]);
    else
      msg ("argv[%d] = null", i);
  msg ("end");

  return 0;
}
```

여기서 msg()함수가 실행이 되면 실행 과정에서 vmsg()함수가 실행이 되게 된다.

```
static void
vmsg (const char *format, va_list args, const char *suffix)
{
  /* We go to some trouble to stuff the entire message into a
     single buffer and output it in a single system call, because
     that'll (typically) ensure that it gets sent to the console
     atomically.  Otherwise kernel messages like "foo: exit(0)"
     can end up being interleaved if we're unlucky. */
  static char buf[1024];

  snprintf (buf, sizeof buf, "(%s) ", test_name);
  vsnprintf (buf + strlen (buf), sizeof buf - strlen (buf), format, args);
  strlcpy (buf + strlen (buf), suffix, sizeof buf - strlen (buf));
  write (STDOUT_FILENO, buf, strlen (buf));
}
```

이 vsmg()함수의 마지막을 보면 system call의 한 종류인 write()를 호출하는 것을 볼 수 있다.
이 함수는 lib/user/syscall.c에 위치하여 user code에서 system call을 요청할 때 사용이 된다.

```
int
write (int fd, const void *buffer, unsigned size)
{
  return syscall3 (SYS_WRITE, fd, buffer, size);
}
```

```
/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
   ARG2, and returns the return value as an `int'. */
#define syscall3(NUMBER, ARG0, ARG1, ARG2)                      \
        ({                                                      \
          int retval;                                           \
          asm volatile                                          \
            ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "    \
             "pushl %[number]; int $0x30; addl $16, %%esp"      \
               : "=a" (retval)                                  \
               : [number] "i" (NUMBER),                         \
                 [arg0] "r" (ARG0),                             \
                 [arg1] "r" (ARG1),                             \
                 [arg2] "r" (ARG2)                              \
               : "memory");                                     \
          retval;                                               \
        })
```

아래 매크로에 들어있는 syscall3()를 통해 특정한 syscall을 argument들과 함께 호출하게 된다.

11) syscall interrupt handler

이러면 기존에 syscall_init()을 통해 등록된 syscall_handler로 가게된다.

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}
```

이렇게 호출된 syscall_handler()는 이제 앞으로 구현이 되어야 하는데, syscall의 종류에 따라 맞춰서 kernel 입장에서 수행해야할 작업을 실행할 함수들이 실행이 되도록 구현이 되어야 한다. 마찬가지로 그 함수들도 구현이 되어야 한다.

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  printf ("system call!\n");
  thread_exit ();
}
```

- 추가 서술

syscall_init 함수에서 intr_register_int 함수를 사용하여 int $0x30이 호출된 경우 syscall_handler 함수를 호출하도록 설정한다.(이와 비슷한 처리를 하는 것을 이전 프로젝트에서 다룬 내용인 timer_init 함수에서도 볼 수 있다. 물론 여기서는 intr_register_ext 을 호출하여 externel interrupt를 설정하는 것이기는 했지만 interrupt에 대한 번호를 예약한다는 동작의 목적은 동일하다. syscall_init → intr_register_int → register_handler → make_trap_gate → make_gate 순서로 함수를 호출한다.

이후에 어떤 방식으로 syscall_handler 함수가 호출되는지 알아보자. user program에서 #include <syscall.h>를 통해 src/lib/user/syscall.c 에 있는 함수를 호출할 수 있게 한다. user program에서 src/lib/user/syscall.c 에 있는 함수를 호출하면, 이 함수들은 syscall을 할 때 필요한 매개변수의 개수에 따라 syscall# 중 하나를 호출하게 된다. 어떤 함수를 호출하던지 공통

적으로 int $0x30을 호출하여 src/usrprog/syscall.c에 있는 syscall_handler를 호출한다. 참고로, syscall_handler를 호출하기 전에 syscall# 함수에서 어셈블리언어로 user stack(user program을 동작시키고 있기 때문에 호출한 시점에서 esp는 user stack의 stack pointer이기 때문)에 syscall number나 필요한 argument들을 다 넣어둔다. 이러한 상황에서 syscall_handler를 적절히 구현할 필요가 있겠다. 이에 대해서는 아래에서 언급하겠다.

위에서 언급한 syscall_init 함수에서 설정을 해두었기 때문에 이처럼 동작할 수 있는 것이라는 점을 명심해야한다.

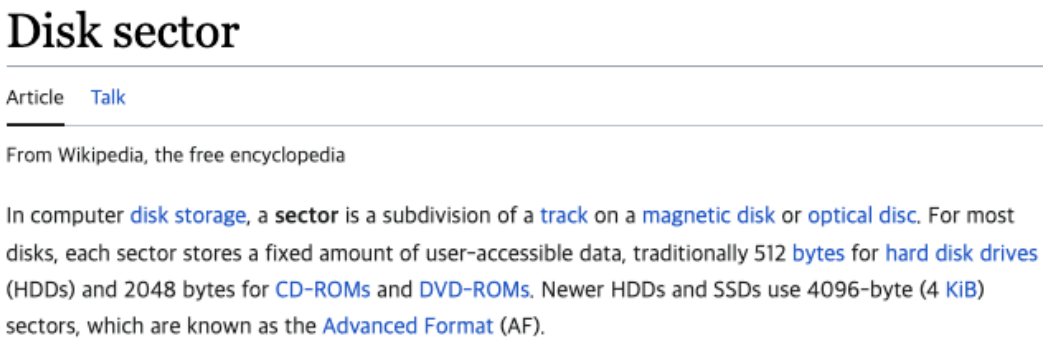
3. file system 분석하기

inode.c

disk에 있는 파일 정보의 layout을 나타내기 위한 기능을 하는 함수와 구조체가 정의되어있다.

struct inode_disk

파일과 관련된 정보중 disk와 관련된 정보를 나타내기 위한 구조체이다. 크기는 항상 512바이트가 되어야 하고, 그 이유는 디스크의 한 sector는 전통적으로 512바이트이기 때문이다.(아래의 위키피디아 정의 참고) 이를 위해 unused 변수를 두었다. 데이터가 들어가 있는 곳의 sector를 start 변수로, 파일의 길이를 length 변수로, inode임을 확인할 수 있는 magic 변수로 두었다. 문서에서 하나의 파일에 있는 disk 상에서 연속된 범위의 sector를 occupy해야만 한다라는 제약조건이 있었는데 이 제약조건 아래에서 구현한다고 생각해보면 충분한 변수가 있는 것이다.



Disk sector

Article Talk

From Wikipedia, the free encyclopedia

In computer disk storage, a **sector** is a subdivision of a track on a magnetic disk or optical disc. For most disks, each sector stores a fixed amount of user-accessible data, traditionally 512 bytes for hard disk drives (HDDs) and 2048 bytes for CD-ROMs and DVD-ROMs. Newer HDDs and SSDs use 4096-byte (4 KiB) sectors, which are known as the Advanced Format (AF).

bytes_to_sectors

size가 주어졌을 때, disk 상에 sector가 얼마나 필요한지를 리턴해주는 함수이다. 매개변수의 타입이 off_t라는 점을 보아 file size를 문제 없이 나타낼 수 있고, 출력의 타입이 size_t라는 점에서 개수를 문제 없이 나타낼 수 있다.

struct inode

파일과 관련된 정보 중 메모리 상에서 필요한 정보를 다루기 위한 구조체이다. struct inode_disk data 변수를 가지고 있어, disk관련 정보 하나를 보유하고 있음을 알 수 있다. sturct list_elem elem 변수가 있는 것을 보아 list로 관리되는 것을 알 수 있다. 그 list의 이름은 open_indoes이다. open된 inode는 이 list에서 관리된다. 그리고, 해당 inode를 몇 번 open 하였는지에 대한 정보가 open_cnt변수에 저장되고, deny_write_cnt는 파일에 write를 deny할 때 참고할 정보로 사용된다. 그리고 파일을 삭제하는 과정에서 사용할 removed 변수도 있다.

byte_to_sector

inode에 저장된 파일에 대해, disk 상에서 시작 지점부터 pos 바이트만큼의 offset을 더한 지점의 sector를 리턴한다. 만약 pos가 파일의 범위를 넘어가면 -1을 리턴한다.

open_inodes

open된 inode를 list로 다룬다. 하나의 inode를 두 번 open 하는 상황에서, 이 리스트에 동일한 inode가 두 개가 생기지 않는다. 한 번 연 inode의 open_cnt를 올리는 식으로 처리하고 이미 open된 inode를 리턴한다.

inode_init

open된 inode를 관리하는 struct list open_inodes 를 초기화할 때 사용되는 함수로, filesys_init 함수에서 호출된다. filesys_init 함수는 init.c의 main 함수에서 호출된다. 즉, open_inodes는 file system의 초기화 과정에서 호출된다.

inode_create

길이가 length인 inode를 create 하고, file system의 특정 sector에 새로운 inode를 write 한다.

free_map_allocate 함수를 활용해서 inode_disk * 타입을 가지는 disk_inode 변수의 start에 secotrs개 만큼의 연속된 sector의 시작 sector 위치를 할당한다. 그리고 block_write를 통해 할당된 sector의 내용을 0으로 초기화한다.

inode_open

매개변수로 들어온 sector에 있는 inode를 읽는다. 만약 read 할 inode가 이미 open 되어 있다면 inode의 open_cnt를 하나 올리고 동일한 struct inode *를 리턴한다. 만약 read 할 inode가 처음 open 되는 것이라면 open_inodes 에 push_front하고 관련 정보를 초기화한다.

block_read 함수를 통해 file system의 sector에 위치해 있는 값을 이 함수에서 만든 indoe의 data 멤버변수와 연결시킨다. 이를 위해서 struct inode에 struct inode_disk가 필요한 것이다. 다시 말하면, 메모리상에서 파일을 관리하기 위해 struct inode가 필요하고 실제 file system 상에 있는 파일 정보를 다루기 위해서 struct inode_disk가 필요한 것이다.

이렇게 구성된 inode를 리턴한다.

inode_reopen

inode의 open_cnt를 하나 올린다. inode가 NULL이라면 NULL을 리턴한다.

inode_get_inumber

inode가 file system에서 어떤 sector에 할당되어 있는지를 리턴해준다.

inode_close

open되어 있는 inode를 close한다. disk에 있는 특정 sector를 struct inode_disk 타입의 변수가 관리하고, struct inode 에 struct inode_disk 타입의 변수가 있다. 즉, struct inode 여러개가 struct inode_disk를 가리킬 수 있다는 것이다. 하지만 실제 구현은 inode_open 함수에서 이미 open된 inode는 다시 open 되는 것이 아니라 open_cnt를 올리는 식이다. 이러한 현재 구현을 생각해보았을 때 inode를 close 하는 것은 open_cnt를 하나 내리는 것으로 생각할 수 있다.

중요한 점은 open_cnt가 0이되는 시점에는 open_inode에서 inode를 제외시킨다. 또한 inode의 removed 가 참이라면 disk에서도 지워준다. inode 자체도 할당해제 하는 것을 잊으면 안된다.

inode_remove

struct inode의 removed 변수를 true로 만든다.

inode_read_at

inode가 가리키고 있는 inode_disk의 시작 sector로부터 offset만큼 떨어진 곳에 있는 데이터를 size 바이트 만큼 읽어온다. 읽어온 정보는 buffer에 저장된다. 리턴하는 값은 read 한 바이트 수이다.

inode_write_at

inode가 가리키고 있는 inode_disk의 시작 sector로부터 offset만큼 떨어진 곳에 있는 데이터로 size 바이트 만큼 write 한다. buffer에 있는 정보를 write한다. 리턴하는 값은 write 한 바이트 수이다. write 할 때 end of file을 만나면 그냥 거기까지 쓰고 만다.

inode_deny_write

struct inode 구조체의 deny_write_cnt 값을 증가시킴으로써 inode에 write 하는 것을 못하게 한다. inode owner 하나에 최대 한 번 씩 호출할 수 있다.

inode_allow_write

inode_deny_write 함수의 반대 연산으로, deny_write_cnt값을 감소시킨다.
inode_deny_write 함수를 호출한 indoe owner 에 의해서 호출되어야만 한다.

inode_length

inode의 data의 length를 리턴한다.

file.c

file read write를 disk sector의 read write로 translate하는데 사용되는 함수와 구조체가 정의되어 있다.

struct file

disk sector의 정보와 연결되는 struct inode * inode를 변수로 가지고 있으며 file 안에서 어떤 위치를 가리키는지를 기록하기 위한 off_t pos 변수가 있고, file에 대한 write를 막는데 사용하는 deny_write 변수가 있다.

file_open

주어진 inode에 대해서 file을 연다. 이는 struct file file을 할당함으로써 시작된다.

만약 할당하는 것에 실패하면 inode_close를 호출하고 할당한 file을 할당해제하고 NULL을 리턴한다.

정상적으로 file이 할당 되었다면 file의 inode 변수가 file_open 함수의 매개변수로 들어온 inode를 가리키게 한다. 그리고 pos를 0으로, deny_write를 false로 하는 초기화를 마치고 file을 리턴한다.

file_reopen

struct inode 타입의 동일한 인스턴스를 두 번 open 하면 새로운 Inode를 만드는 것이 아니라 open_cnt를 올려주었다. file 구조체에는 sturct inode * inode가 있다. 따라서 file을 다시 open 할 때는 inode_reopen 함수를 호출하고, file_open 함수를 호출하는 방식으로 구현하여 struct inode 에 있는 open_cnt를 올리는 처리를 한다.

file_close

file을 close 하므로 wirte를 allow한다. 그리고 file 구조체를 할당해제한다. 이 전에, file은 inode를 가리키므로, inode_close도 해주어야 한다.

file_get_inode

file의 inode 변수를 리턴한다.

file_read

file의 current position에서부터 size 바이트만큼을 buffer로 읽는다. 읽은 byte 수 만큼 file의 current position을 옮기고, 읽은 byte 수를 리턴한다. 위에서 설명한 inode_read_at 함수를 활용하여 구현하여 별 다른 추가 구현을 할 필요 없다.

file_read_at

file의 시작 지점부터 file_ofs 바이트 떨어진 위치에서부터 size 바이트만큼 buffer로 읽는다. 위에서 설명한 inode_read_at 함수를 활용하여 구현하여 별 다른 추가 구현을 할 필요 없고 file_read 함수와 다른 점은 file의 current position이 바뀌지 않는다는 것이다.

file_write

file의 current position에서부터 size 바이트만큼을 buffer로 write한다. write한 byte 수 만큼 file의 current position을 옮기고, write 한 byte 수를 리턴한다. 위에서 설명한 inode_write_at 함수를 활용하여 구현하여 별 다른 추가 구현을 할 필요 없다.

file_write_at

file의 시작 지점부터 file_ofs 바이트 떨어진 위치에서부터 size 바이트만큼 buffer로 write한다. 위에서 설명한 inode_write_at 함수를 활용하여 구현하여 별 다른 추가 구현을 할 필요 없고 file_write 함수와 다른 점은 file의 current position이 바뀌지 않는다는 것이다.

file_deny_write

file의 deny_write 변수가 거짓이라면, deny_write 값을 참으로 바꿔준다. file은 inode를 가리키는 식으로 구현되어 있기 때문에 inode에 대한 처리를 위해 file의 inode 변수를 인자로하여 inode_deny_write 함수 또한 호출해주어야한다.

file_deny_write 함수가 호출되는 때에 file의 deny_write 변수가 이미 참이라면 아무런 일을 하지 않는다.

file_allow_write

file의 deny_write 변수가 참이라면, deny_write 값을 거짓으로 바꿔준다. file은 inode를 가리키는 식으로 구현되어 있기 때문에 inode에 대한 처리를 위해 file의 inode 변수를 인자로하여 inode_allow_write 함수 또한 호출해주어야한다.

file_allow_write 함수가 호출되는 때에 file의 deny_write 변수가 이미 거짓이라면 아무런 일을 하지 않는다.

file_length

file은 inode를 가리키는 식으로 구현되어 있다. file이 가리키는 inode가 가리키는 inode_disk의 length를 리턴해준다. 이는 곧 file의 size를 바이트로 나타낸 것이다.

file_seek

file의 current position을 인자로 들어온 new_pos로 설정한다.

file_tell

file의 current position을 리턴한다.

filesys.c

file system의 top level 인터페이스이다. root directory를 만드는 기능을 할 수 있으며, 만들어진 root dircetory에 대해 file을 만들고, 열고, 닫을 수 있는 함수를 제공한다.

do_format

file system을 formatting한다. 이 함수는 filesys_init 함수에서 호출되며, file system을 초기화할 때 선택적으로 사용될 수 있다. init.c에 전역변수로 선언된 format_filesys가 참 일때 do_format 함수가 호출되는데, 이는 pintos 실행시 -f 옵션으로 format_filesys을 참으로 만들 수 있다.

free_map_create 함수를 통해 disk에 새로운 free map file을 생성하고 free map을 그 file에 write 한다. 그리고 dir_create 함수를 호출하는데, ROOT_DIR_SECTOR를 인자로 한다. 이 때, root directory가 생기는 것이다. 이 root directory는 filesys.c에 정의된 다른 함수들에 의해 사용된다.

그 후 free_map_close 함수를 통해 방금 만든 free_map_file을 닫는다. free_map_file은 free-map.c에 전역변수로 정의되어 있다.

filesys_init

이 함수는 inode_init 함수와 free_map_init 함수를 호출하여 file system을 위한 초기화를 한다. 그리고 free_map_open 함수를 통해 free_map_file을 open한다.

fs_device를 초기화 하는 것도 이 함수에서 한다.

filesys_done

free_map_close 함수를 호출하여 free_map_file 파일을 close 한다.

filesys_create

매개변수로 주어진 initial_size 바이트만큼으로 file을 만든다. file의 이름은 매개변수로 주어진 name이다. file의 이름이 이미 있거나 내부적으로 메모리 할당에 실패하면 false를 리턴한다.

특히 dir_open_root 함수를 통해서 root directory를 열고, 새로 만든 inode의 sector를 root directory에 add한다. root diectory를 close 하는 것도 잊지 말아야한다.

추가로 설명하자면 새로 만들어지는 inode는 free_map_allocate 함수로 free_map에 sector를 할당받았음을 기록하고, sector 번호를 받아온다. 그 후 indoe_create 함수에서 initial_size 바이트만큼으로 필요한 sector를 할당받아 만들어진다.

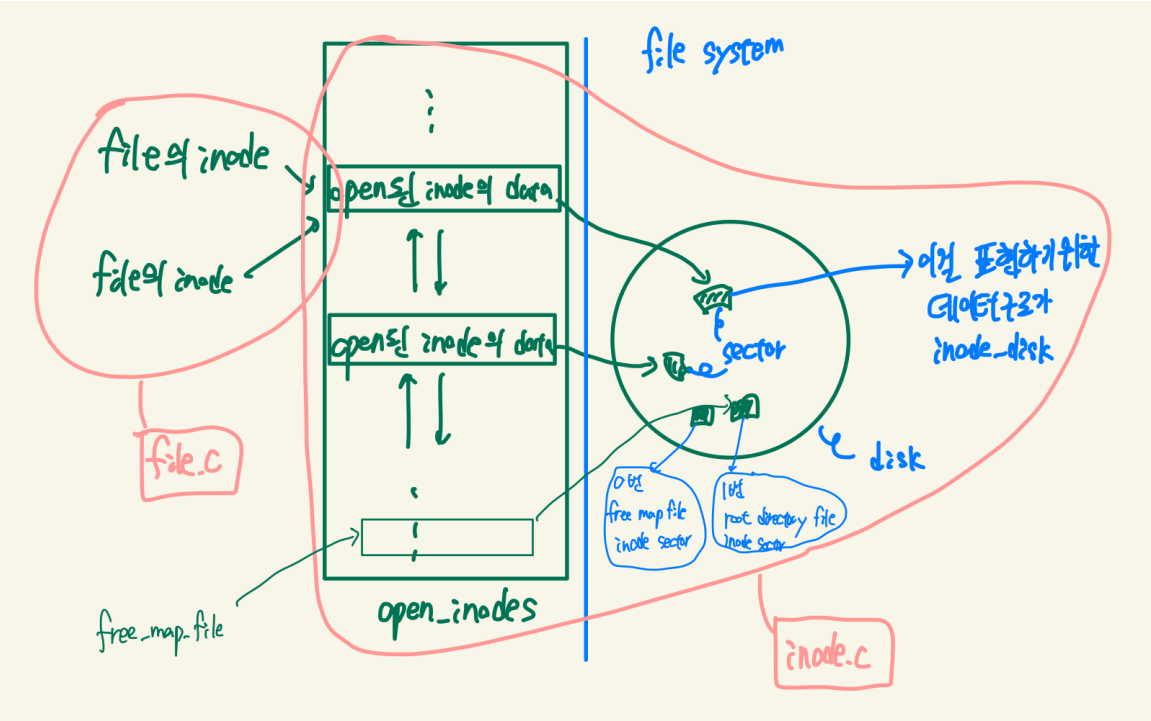
filesys_open

root directory를 열고 name이라는 이름의 파일이 있는지를 찾는다. 만약에 찾는 것을 성공하면 inode 변수에는 그 파일의 inode 주소가 들어가고 실패하면 inode 변수에는 NULL이 들어간다. inode를 매개변수로 file_open 함수를 활용하여 파일을 열고 이를 리턴한다. root diectory를 close 하는 것도 잊지 말아야한다.

filesys_remove

root directory를 열고 name이라는 이름의 파일을 삭제한다. 삭제할 파일이 없는 경우 아무일도 일어나지 않는다. root diectory를 close 하는 것도 잊지 말아야한다.

- file system 정리



4. Process Termination Messages Design Plan

user program을 위한 pintos의 c library에는 _start() 함수가 있다. 이 함수는 user program의 entry point가 된다. 따라서 정상적으로 종료되는 user process는 system call exit를 거쳐 종료되기 때문에 이 exit 함수 내에서 print를 해줄 예정이다. process의 이름과 exit code를 출력해야 하므로 TCB에 process의 이름과 exit code를 저장할 수 있는 멤버변수를 추가할 것이다.

user process가 아닌 kernel thread가 terminate 될 때나 system call halt가 호출되었을 때는 종료 메세지를 출력하면 안된다는 조건이 있다. halt는 pintos를 terminate 시키는 것이기 때문

에 자연스럽게 조건을 만족할 것이고 kernel thread는 exit 함수를 호출하는 방식으로 종료되지 않기 때문에 해당 조건도 만족할 것이다.

5. Argument Passing Design Plan

코드 분석 결과 load 함수에서 executable을 불러오고 있으며 이 함수의 설명을 보면 “Loads an ELF executable from FILE_NAME into the current thread.” 이라는 부분이 있기에 load 함수로 들어가기 이전에 argument parsing을 진행하여 executable파일 이름만 넘겨줄 것이다.

이는 process_execute 함수에서 호출하는 start_process 함수에서 load 함수를 호출하기 이전에 file_name_을 입력으로 받고, 쪼갠 결과를 출력으로 하는 함수를 정의함으로써 구현할 예정이다. 또는 process_execute 함수에서 parsing을 진행하여 구현해도 될 것 같다.

추가로, load 함수에서는 user stack의 공간 할당을 마치고 user stack의 stack pointer를 interrupt frame의 esp에 저장해두기 때문에 이 값(if_.esp)를 활용하여 user stack에 인자들을 주어진 convention에 맞게 넣을 것이다.

이는 load 함수 호출 이후에 &if_.esp와 위에서 정의한 함수의 결과를 입력으로 받아 user stack을 구성하는 함수를 정의함으로써 구현할 예정이다. 이때 주소를 4의 배수로 정렬해주는 기능과 fake return address를 넣는 등의 동작도 추가할 것이다. right to left 순서로 stack에 넣어야 한다는 것도 유의해야겠다.

- stack 주소를 감소 후 값 저장한다는 것은 주어진 문서에서도 강조하고 있는 내용이니 이를 주의하여 구현해야겠다.
- 주어진 문서의 추천대로 문자열 파싱 과정에는 strtok_r 함수를 사용할 것이다.

6. System call Design Plan

interrupt로부터 받아온 syscall num으로부터 어떤 system call 이 호출되었는지를 확인해야한다. system call number는 caller 즉, user의 stack 가장 상단에 위치해있고 argument는 차례로 윗 주소에 위치한다. stack pointer에 대한 정보는 user stack에 기록되어 있고, syscall_handler로 interrupt frame의 esp 멤버변수로 전달된다. 이때, kernel에서 user vertual memory에 대한 주소가 올바른지 확인해야한다. 만약 올바르지 않은 메모리 접근을 허용한다면 kernel에 영향을 줄 수 있다. 따라서 system call hander에서 user stack을 사용하여 접근하는 값에 대한 verify가 필요하다. 이를 위한 함수를 만들 예정이다. 이는 참고하려는 곳의 주소가 0x08048000 초과 PHYS_BASE 미만인지를 확인하는 것을 통해 할 것이다. 상황에 따

라 추가 조건을 넣을 수도 있다. 이하의 서술에서 user stack에서 어떤 값을 읽어올 때 항상 verify를 한다고 가정한다.

그리고 문서의 안내에 따라 하나의 process만 file system code를 사용할 수 있게 해야하기 때문에 filesys_lock을 추가하여 file system과 관련된 syscall에서는 lock을 얻고 종료 직전에 lock을 풀어주는 형식으로 구현할 것이다.

thread에서 파일을 다루기 위해 open 한 파일에 descriptor 번호를 부여해야하고 연결된 파일을 관리해야 하기 때문에, 번호 부여를 위한 변수, thread에서 연 파일을 다루기 위한 변수(여러 파일을 열 수 있으니 list로 다룰 것) 등을 추가로 선언할 것이다. file descriptor 번호와 실제 파일을 함께 묶어 다루기 위한 구조체를 만들면 위에서 언급한 list를 하나만 만들어도 되겠다.

file descriptor는 non negative 정수이고, 0과 1은 이미 예약되어 잇다. 그리고 각각의 프로세스는 독립적인 file descriptor를 가지고 있다. 이는 자식에게도 상속되지 않으며 파일은 여러 file descriptor들을 가질 수 있다는 특징이 있다. 이를 고려한 구현을 할 것이다.

syscall_handler

interrupt frame에서 user stack의 주소를 받아오고, syscall number를 찾는다. 그리고 그 번호(lib/syscall.h에 정의되어있음.)에 맞게 syscall 함수를 호출하여 적절한 기능을 수행하도록 할 것이다. 만약 올바르지 않은 주소를 참조해야하는 경우라면 해당 user process를 terminate 시킬 것이다.

void halt (void)

OS를 종료하기 위해 SYS_HALT syscall을 호출하는 함수다. 이는 devices/shutdown.c에 정의된 shutdown_power_off()를 실행시키는 방식으로 간단하게 구현할 예정이다.

void exit (int status)

현재 user program을 종료하고 status를 kernel로 반환하는 함수다. parent process가 current process를 기다리는 경우, parent process는 status 값을 받는다. 0을 반환 받았다는 것은 성공을, 다른 값은 오류를 뜻한다.

TCB에 exit status를 작성하고 현재 실행 중인 thread와 관련되어 할당된 메모리를 할당해제하기 위해 thread_exit 함수를 호출할 생각이다. thread_exit함수에서 process_exit 함수를 호출하므로 아래에서 언급할 process_exit 함수의 기능을 할 수 있을 것이다.

즉시 -1을 반환해야 하는 조건은 다음과 같다.

pid가 호출 프로세스의 직접적인 자식을 참조하지 않는 경우

프로세스가 자신의 자식의 자식, 혹은 그 이상의 자식 관계의 프로세스를 기다리려는 경우

프로세스가 pid에 대해 이미 wait()를 호출했던 경우

pid_t exec (const char *cmd_line)

cmd_line에서 지정된 실행 파일 이름을 실행하고, cmd_line에 주어진 모든 인수를 전달한다. 반환 값은 새 process의 pid이다. 프로그램을 로드하거나 실행할 수 없는 경우에는 유효하지 않은 pid를 의미하는 -1을 반환한다. exec 함수는 새 process의 성공적인 로딩과 실행을 확인할 때까지 반환할 수 없다.

이는 process_execute 함수를 통해 구현하면 되겠다. 함수에서 요구하는 것이 위에서 살펴본 user process의 실행과정과 동일하기 때문이다.

int wait (pid_t pid)

pid가 pid인 자식 프로세스를 기다리고, 자식 프로세스의 종료 상태를 받는 함수다. 이는 위에서 설명한 process_wait 함수의 호출로 구현할 수 있을 것이다.

bool create (const char *file, unsigned initial_size)

이름이 file이고 초기 크기가 initial_size 바이트인 새 파일을 생성한다. 생성이 성공적인 경우 반환 값은 true, 생성에 문제가 있었을 시 반환 값은 false다.

현재 thread에서 파일을 open하려고 한 것이므로 관련된 기능 중 가장 추상화 레벨이 높은 filesys_create 함수를 통해 파일을 만들 것이다.

bool remove (const char *file)

file이라는 이름의 파일을 삭제한다. 성공적인 경우 true를 반환한다.

현재 thread에서 파일을 open하려고 한 것이므로 관련된 기능 중 가장 추상화 레벨이 높은 filesys_remove 함수를 통해 파일을 지울 것이다.

int open (const char *file)

file이라는 이름의 파일을 연다. 파일을 열 수 없는 경우 -1 혹은 file descriptor를 반환한다.

현재 thread에서 파일을 open하려고 한 것이므로 관련된 기능 중 가장 추상화 레벨이 높은 filesys_open 함수를 통해 파일을 열 것이다. 그리고 연 파일에 대한 정보를 open syscall을 호출한 thread의 TCB에 저장해줄 것이다. 파일을 열었으니 file descriptor에 대한 정보를 할당하고 open 한 파일을 다루는 리스트에 넣는 식의 구현을 할 예정이다.

int filesize (int fd)

fd로 열린 파일의 크기를 바이트 단위로 반환한다.

이를 위해서는 thread가 open한 파일 중 fd에 해당하는 파일을 찾고 file_length 함수를 활용하여 구현하면 되겠다.

int read (int fd, void *buffer, unsigned size)

fd로 열린 파일에서 size 바이트만큼 buffer로 읽는다. 반환 값은 실제로 읽은 바이트 수다.

이를 위해서는 thread가 open한 파일 중 fd에 해당하는 파일을 찾고 file_read 함수를 통해 buffer에 있는 값을 읽으면 되겠다. 그리고 file_open 함수의 리턴 값을 리턴하면 되겠다.

read의 경우 file descriptor의 값이 0인 경우에 대한 처리를 다르게 하여 thread가 open 한 파일에서 찾는 것이 아니라 input_getc 함수를 이용하여 키보드에서 직접 읽어올 수 있게 하면 되겟다.

int write (int fd, const void *buffer, unsigned size)

buffer에서 fd로 열린 파일로 size 바이트를 쓴다. 실제로 쓴 바이트 수를 반환한다.

이를 위해서는 thread가 open한 파일 중 fd에 해당하는 파일을 찾고 file_write 함수를 통해 buffer에 있는 값을 쓰면 되겠다. 그리고 file_write 함수의 리턴 값을 리턴하면 되겠다.

write의 경우 file descriptor의 값이 1인 경우에 대한 처리를 다르게 하여 thread가 open 한 파일에서 찾는 것이 아니라 putbuf 함수를 통해 console에 출력하는 기능을 구현할 수 있겠다.

void seek (int fd, unsigned position)

열린 파일 fd에서 쓰거나 읽을 다음 바이트를 파일의 시작부터 바이트로 표현된 position으로 변경한다. position 0은 파일의 시작 지점이다.

이를 위해서는 thread가 open한 파일 중 fd에 해당하는 파일을 찾고, 존재한다면 file_seek 함수를 통해 할 일을 하면 되겠다.

unsigned tell (int fd)

열린 파일 fd에서 쓰거나 읽을 다음 바이트의 위치를 반환한다. 위치는 파일의 시작부터 바이트로 표현된다.

이를 위해서는 thread가 open한 파일 중 fd에 해당하는 파일을 찾고, 존재한다면 file_tell 값을 리턴하면 되겠다.

void close (int fd)

디스크립터가 fd인 파일을 닫는다.

이를 위해서는 thread가 open한 파일 중 fd에 해당하는 파일을 찾고, 존재한다면 file_close 함수를 통해 파일을 닫으면 되겠다. 만약 관련된 할당된 메모리가 있으면 제거도 할 필요가 있겠다.

7. Denying Writes to Executables Design Plan

executable 파일에 write를 deny하려면 executable 파일을 불러올 때 관련 처리를 해야한다. 즉, load 함수에서 filesys_open 함수를 사용하여 executable 파일을 부르므로 load 함수 내부에서 구현할 예정이다.

현상황과 구현할 내용에 대해 조금 자세히 살펴보자.

filesys_open 함수 내에서 dir_lookup 함수를 호출하고 이 함수는 inode_open 을 한다. inode_open 함수에서는 deny_write_cnt 를 0으로 설정하기 때문에 기존에는 open 된 file에 write를 할 수 있는 상태이다.

서로 다른 file이 동일한 inode를 가리킬 수 있기 때문에 file_deny_write 함수로 file의 inode에 대해 inode_deny_write 함수를 호출하는 것만으로도 executable 파일에 write를 deny하는 기능을 충분히 구현할 수 있을 것이다.

8. 추가 변경 내용

- process_exit 함수

  기본적으로 current process의 resource를 할당해제하는 것이 목적이다. thread에서 file을 열면 관련된 메모리가 할당되게 되는데 이를 할당해제 하는 기능을 추가해야할 것이다. 또한 thread에서 user process 실행을 위해 open한 파일을 write 하지 못하게 하는 기능을 추가하였으므로 다시 file에 write할 수 있도록 복구하는 기능도 여기서 해야할 것이다. 또한 user process 실행을 위해 load에서 filesys_open 함수로 연 executable 파일을 닫아줄 곳도 이 곳으로 할 생각이다. 이를 위해 TCB에 open 한 file의 이름을 기록해둘 필요가 있겠다. deny write 기능을 구현하게 되므로 file_allow_write 함수도 여기서 불러줘야겠다.

  또한 위에서 언급한 바와 같이 thread가 종료되었으면 자신을 기다리고 있는 부모에게 이를 알리는 sema_up 함수도 이 함수에서 호출해야겠다.