Project 2. Design Report

Team7 20210741 김소현 20210774 김주은

Virtual Memory Layout in Pintos

pintos pdf의 3.1.4의 내용을 기반으로 정리한 내용이다. 본 lab design report를 작성하기 이전에 가장 먼저 정리하고자 한다. Pintos에서 Virtual memory는 user/kernel의 두 가지 파트로 나뉜다. User virtual memory는 virtual address 0에서부터 PHYS_BASE까지 범위이고(‘threads/vaddr.h’에 정의됨), Kernel virtual memory는 virtual address space의 나머지인 PHYS_BASE부터 4 GB를 점유한다.

- **user virtual memory**

  process마다 별도로 할당된다. kernel은 process 간 전환 시 pagedir_activate() 함수를 통해 processor의 directory base register를 변경함으로써 user virtual memory를 전환한다. struct thread에는 process’s page table의 pointer가 포함되어 있다.

- **kernel virtual memory**

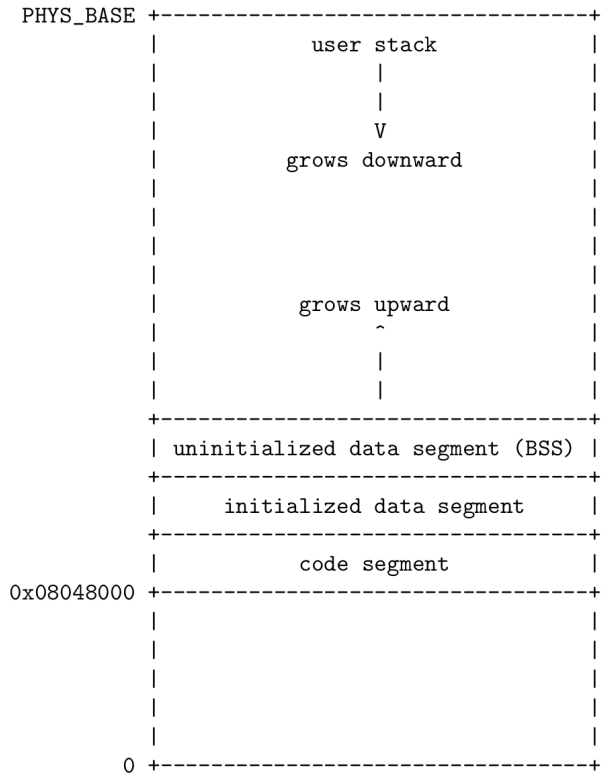
  global이다. 항상 동일한 방식으로 매핑되고, Pintos에서 PHYS_BASE부터 시작하여 physical memory를 1:1로 매핑한다.

- **access**

  user program은 자신의 user virtual memory에만 접근할 수 있다. 만약 kernel virtual memory에 접근 시도하면 page fault 일어나고, process terminated된다. 이러한 page_fault는 userprog/exception.c에 정의되어 있다.

  kernel thread는 kernel vm, user vm(user process 실행중이면) 둘 다 접근 가능하다. 하지만 unmapped user virtual address에는 접근하면 page fault가 일어난다.

- **structure**



project2에서는 user stack은 fixed size이다. 하지만 이후 project에서는 확장될 수 있다고 한다. Pintos의 코드 세그먼트는 사용자 가상 주소 0x08084000에서 시작하여 주소 공간의 하단에서 약 128MB 떨어진 곳에 위치한다.

- **Accessing User Memory**

  system call의 일환으로, kernel은 user program에 의해 제공되는 pointer로 메모리에 접근한다. 이때 kernel이나 다른 process에 피해를 미치지 않도록 주의해야 하고, 다음과 같은 방법이 있다.

  - 첫 번째로, 사용자가 제공한 포인터의 유효성을 확인한 다음에야 이를 역참조하는 것이다. 이 방법을 선택하면 'userprog/pagedir.c' 및 'threads/vaddr.h'의 함수를 이용하면 되고, 가장 간단한 방법이다.
  - 두 번째로, 사용자 포인터가 PHYS_BASE 아래를 가리키는지만 확인한 다음에야 이를 역참조하는 것이다. 잘못된 사용자 포인터는 "page fault"를 발생시킬 것이며 이를 'userprog/exception.c'의 page_fault() 코드를 수정하여 처리할 수 있다.

---

1. Analysis on process execution procedure

project2 구현에 앞서 process execution procedure와 관련된 현재 pintos code를 분석해 보고자 한다.

1) 함수 분석

threads/init.c

- **int main()**

  pintos main program을 실행하는 함수이다. `argv = read_command_line (); argv = parse_options (argv);` 에서 command line의 입력을 받아와서, parsing한다는 것을 알 수 있다. 현재는 구현이 완성되지 않은 상태다. `thread_start ();`에서 thread를 생성하고, `run_action (argv)` 에 인자를 넘기며 호출한다. 마지막에서는 `thread_exit ();`를 호출하며 종료한다.

- **static void run_actions (char **argv)**

  action name, 인자의 개수, 실행할 함수에 대한 정보를 가지는 `struct action`을 선언하여 사용한다. 여러가지 실행 중 user program (run)이라면 `run_task`를 호출한다.

- **static void run_task (char **argv)**

  argv[1]의 task 값을 process_execute 함수의 인자로 넘겨 실행한다. 이때 USERPROG이면 `process_wait (process_execute (task));`을, 이 외의 경우에는 `run_test(task);`를 구분하여 실행한다.

userprog/process.c

- **tid_t process_execute (const char *file_name);**

  file_name으로부터 로드된 user program을 실행하는 새로운 thread를 생성한다. 새로운 thread의 thread id인 tid 값을 return하고, 생성되지 못한 경우에는 TID_ERROR를 return한다. `tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);`라는 구문에서 thread를 생성하는 것을 확인할 수 있다. 또한, 이때 file_name을 인자로 넘긴다. 현재는 file_name에 올바른 처리를 하지 않고 인자로 넘기기 때문에 해당 부분에 추가 구현이 필요하다.

- **static void start_process (void *file_name_);**

  process를 말그대로 start하는 함수로 먼저 intr_frame 구조체를 초기화하고 `success = load (file_name, &if_.eip, &if_.esp);`을 실행하여 file name에 저장된 user process을 해당하는 memory에 옮긴다. load 결과 값인 success 값이 false이면 `thread_exit()`을 호출한다. 이후 user process를 시작하기 위해서는 intr_exit에서 구현된 interrupt에서 return하는 것처럼 simulating한다. 왜냐하면 intr_exit에서는 스택의 모든 인자를 struct

intr_frame의 형태로 가지기 때문에 stack frame을 %esp로 가리키고 jump하게 한다. 여기서도 제대로 parsing된 file_name이 아니기 때문에 문제 해결이 필요하다는 점을 확인 가능하다.

- **bool load (const char *file_name, void (**eip) (void), void **esp);**

  해당하는 user process에 대한 page table을 생성하는 함수다. file_name에서 현재 thread에 대해 ELF를 로드한다. 실행 가능한 entry point를 *EIP에 저장하고, initial stack pointer를 *ESP에 저장한다. 만약 성공했다면 true를 return한다.

  - `t->pagedir = pagedir_create ();`로 해당 process의 page table을 할당하고, `process_activate ();`을 호출하여 pdbr 값을 실행중인 thread의 page table의 주소로 변경한다.
  - `file = filesys_open (file_name);`을 통해 page table에 가져올 file을 open한다.
  - `file_read (file, &phdr, sizeof phdr)`을 통해 elf header을 읽고 header가 유효한지 검증한다.
  - 다음으로는 elf 실행 파일의 program header을 순차적으로 읽는다.
    - swtich 문을 통해서 program header 유형에 따라 다르게 처리하며 특정 segment들의 경우 작업이 중지되고 done으로 이동하게 된다.
    - 그리고 program header 유형이 PT_LOAD라면 실제로 memory에 load되어야 하는 segment이므로 먼저 `validate_segment (&phdr, file)`을 통해 segment의 유효성을 검사한다. 즉, validate_segment는 해당 phdr이 valid하고 loadable한지 return하는 함수다.
      - 또한 segment의 p_flags를 확인하여 memory내에서 write 권한을 결정하고 p_files를 바탕으로 disk에서 읽어야 할 byte와 0으로 초기화해야 할 byte를 결정한다.
      - 이후 `load_segment (file, file_page, (void *) mem_page, read_bytes, zero_bytes, writable)`의 값을 확인한다. 여기서 파일의 offset에 해당하는 위치에서 read_bytes만큼 load하고 UPAGE + read_bytes 위치의 값들은 zero_bytes만큼 0으로 세팅한다.
  - 마지막으로 stack을 set up하고, `eip = (void () (void)) ehdr.e_entry;`에서 코드의 start 위치를 eip 포인터가 가리키게 한다.
  - 위 모든 단계를 거치고 나면 return할 success 값을 true로 설정하고, 이는 해당 프로그램을 실행하기 위한 메모리를 준비하는 과정을 마쳤음을 뜻한다.

- **int process_wait (tid_t child_tid UNUSED);**

이 함수의 원래 동작은 인자로 넘겨 받은 tid를 가진 thread를 기다려서 die하면, 그 thread의 exit status를 리턴하는 것이다. 만약 kernel에 의해 종료되었다면, 즉 exception에 의해 kill되었다면 -1을 리턴한다. tid가 invalid하거나 호출된 process의 child가 아니거나, process_wait()가 이미 주어진 tid로 성공적으로 호출되었다면, 기다리지 않고 즉시 -1을 return한다. 현재 이 함수는 아무런 동작 없이 즉시 -1을 return하는 것으로 구현되어 있다. 따라서 정상적인 동작을 하지 못하기 때문에 올바른 구현이 필요하다.

- **void process_exit (void);**

  현재 process의 자원들을 free시킨다. page directory를 제거하고, kernel-only page directory로 switch back한다.

threads/thread.c

- **tid_t thread_create (const char *name, int priority, thread_func *function, void *aux);**

  process_execute 함수에서 file_name을 인자로 받아 실행되는 함수다. priority로는 PRI_DEFAULT 값인 31을, function은 start_process, 보조인자 aux로는 fn_copy를 받는다. 이러한 정보를 가지고 새로 kernel thread를 생성한다. 필요한 공간을 page allocate하고, tid를 allocate하고 kernel thread 등을 위한 stack frame을 할당받고, 해당 thread를 run queue에 추가한다. 그리고 tid를 process_execute의 변수 tid 값으로 return하게 된다. 여기서 priority에 PRI_DEFAULT로 넣으므로 priority scheduling이 아닌 다른 방식을 사용한다.

- **void thread_exit (void);**

  `start_progress` 함수에서 호출한 `load`함수가 실패했다면 호출되어 해당 thread를 종료시킨다. `#ifdef USERPROG`문을 통해 user program이었다면 `process_exit()`을 호출하여 user_thread를 종료시킨다. 아닌 경우에는, all thread list에서 해당 thread를 삭제하고, THREAD_DYING 상태로 만든 후에 schedule()을 호출하여 다른 thread로 context switch되도록 한다.

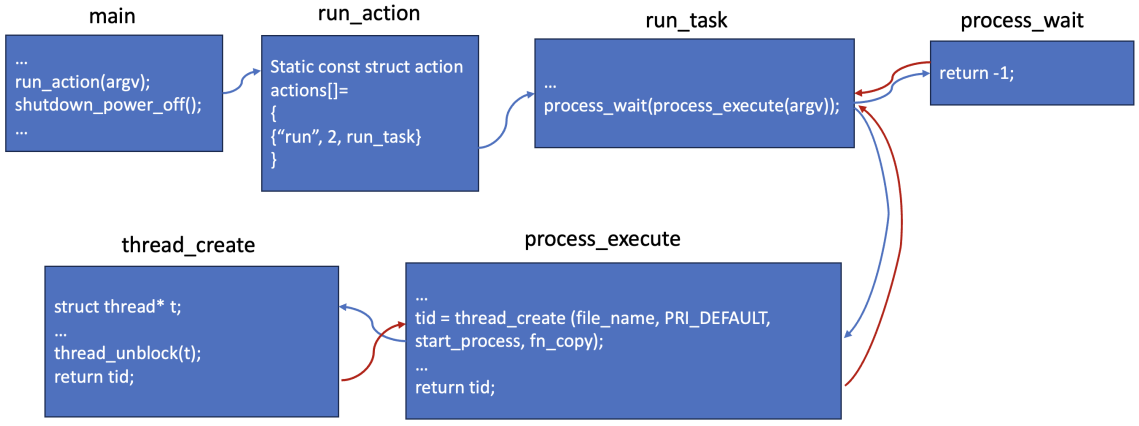
2) current pintos system에서의 process execution procedure 정리

1. init.c의 main(): pintos main program이 실행된다.

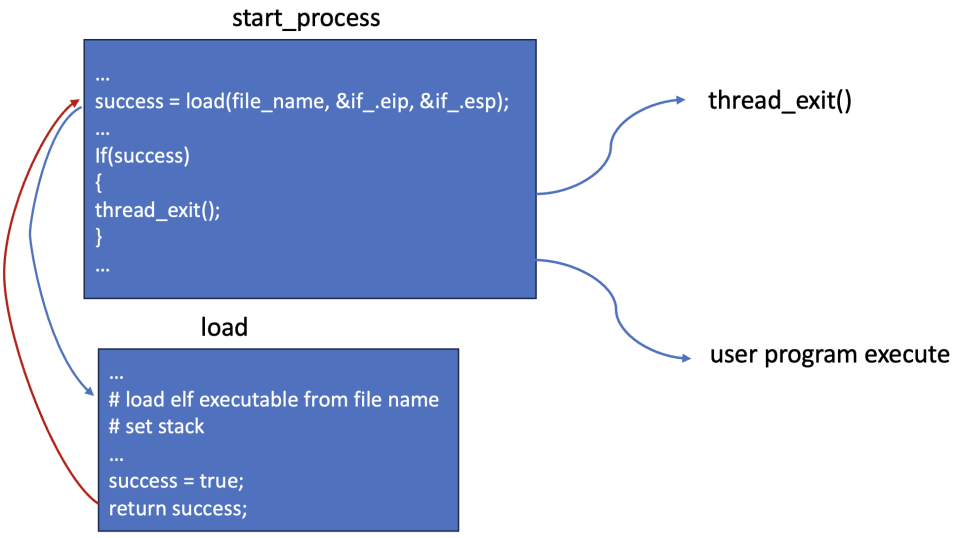
   command line을 읽고 인자를 받아 run_actions(argv)을 실행한다.

2. run_actions(argv): user program인 경우 rus_task()를 실행한다.

3. run_task(): 이에 대한 process_wait (process_execute (task))을 호출한다. 해당 task를 execute하고, process_wait 함수에서는 해당 process가 종료될 때까지 대기한다.
4. process_execute(): 해당 정보를 인자로 넘겨 thread_create()를 실행한다.
5. thread_create(): 이때 thread_create의 인자로는 (file_name, PRI_DEFAULT, start_process, fn_copy);이 전달된다.



6. 이후 thread가 schedule되면 start_process 함수를 실행하게 된다.
7. process_wait(): 해당 process가 종료될 때까지 대기하는 함수인데, 현재는 구현되어 있지 않고 -1을 return하도록 되어있다.
8. start_process(): process의 메모리를 확보하는 역할이다. load (file_name, &if_.eip, &if_.esp)으로 load 함수를 실행한다.
9. load(): file name의 해당 프로그램을 메모리에 탑재한다.
10. load() return값이 1(success)이면 user program이 실행된다.
11. main()에서 run_actions()이 종료되면, thread_exit()으로 종료된다.



Parent-Child relation in process

concurrent한 처리를 해주기 위해 OS는 여러 process가 실행되는 것을 지원해준다. 이 때 여러 Process가 생성되는 과정은 parent-child 개념을 따른다.

process를 생성할 때는 process 하나에서 fork() 함수를 호출하면 process의 자원을 복제하여 새로운 process를 만들게 된다. 이 때 새로운 process를 만든 process가 parent process이고, 만들어진 새로운 process가 child process다.

```
int child_pid = fork();
if (child_pid == 0) {            // I'm the child process
        // getpid is a system call to get the current process's ID
        printf("I am process #%d\n", getpid());
        return 0;
} else {                                  // I'm the parent
        printf("I am parent of process #%d\n", child_pid);
        return 0;
}
```

교과서 예제

이렇게 만들어진 parent process와 child process는 구현 방식에 따라 작동 방식이 달라질 수 있다.예를 들어 parent process와 child process 간에 자원을 공유하지 않는다고 하면 동시적인 실행이 가능하다. 즉, process를 새로 만든 직후에 parent process와 child process가 이후에 바로 실행될 수 있는 것을 말한다. 반면에 parent와 child 간에 자원을 공유하면 동시적인 실행이 제한되기 때문에 parent는 자신이 만든 child process가 모두 종료될 때까지 대기해야 한다. 이때 parent가 child를 기다리는 동작은 wait()함수를 통해 구현된다.

또한, parent process가 만든 child process에 parent process에서 돌아가는 program과 별개의 다른 program을 돌리고 싶으면 exec() 함수를 호출하면 된다. 즉, process의

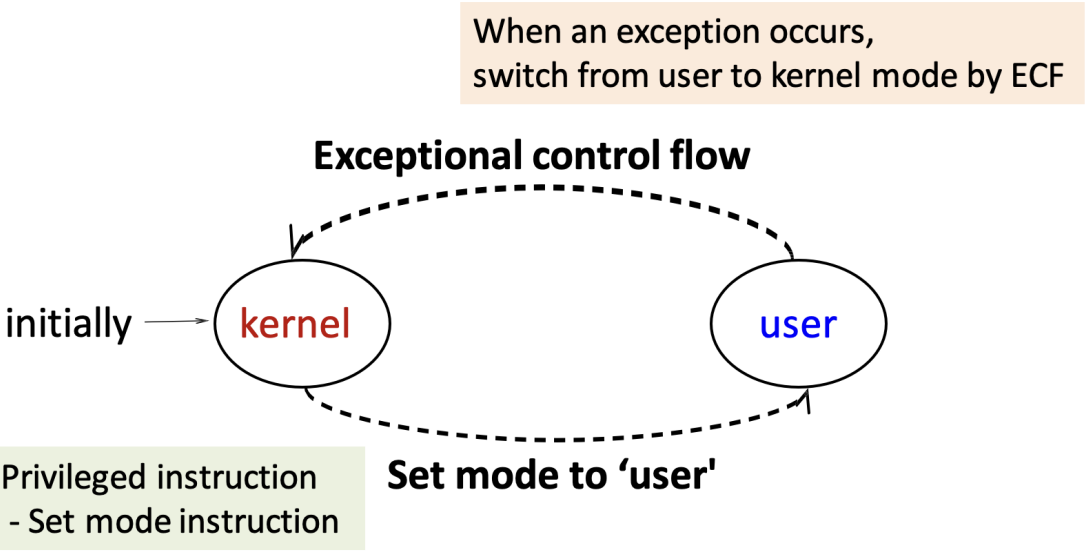
memory 공간을 새 프로그램으로 교체하는 것을 exec() 함수라고 할 수 있다.

2. Analysis on system call procedure

OS가 처리하는 interrupt는 internal interrupt와 external interrupt로 나뉘어져 있고, system call은 internal interrupt에 속한다. 하지만 현재 pintos는 system call handler가 구현되어 있지 않아서 system call이 pintos에서 처리되지 않는다. 이번 과제에서는 핀토스의 system call mechanism을 이해하고 system call handler를 구현해야 한다. 즉, system call (e.g. halt, exit, create, remove …)을 구현하고 system call handler을 호출하여 system call을 처리해야 한다.

- **System Call이란?**

먼저 system call의 작동 방식을 이해해보자. system call이란 OS가 제공하는 service에 대한 programming interface로 user mode program이 kernel 기능을 사용할 수 있도록 한다.



즉, system call은 kernel mode에서 실행되고 실행 후에는 user mode로 돌아간다. 즉, mode가 seperate됨으로써 system call을 처리할 시 실행 mode가 user mode에서 kernel mode로 바뀌는 것이 system call의 핵심 mechanism이다.

- **pintos의 system call 동작 방식 분석**

1. 먼저 **lib/user/syscall.c**을 살펴보면 아래의 예시와 같이 각 system call function들이 정의되어 있다.

```
void
halt (void)
{
  syscall0 (SYS_HALT);
  NOT_REACHED ();
}

void
exit (int status)
{
  syscall1 (SYS_EXIT, status);
  NOT_REACHED ();
}

pid_t
exec (const char *file)
{
  return (pid_t) syscall1 (SYS_EXEC, file);
}

int
wait (pid_t pid)
{
  return syscall1 (SYS_WAIT, pid);
}

bool
create (const char *file, unsigned initial_size)
{
  return syscall2 (SYS_CREATE, file, initial_size);
}

bool
remove (const char *file)
{
  return syscall1 (SYS_REMOVE, file);
}

int
open (const char *file)
{
  return syscall1 (SYS_OPEN, file);
}

int
filesize (int fd)
{
  return syscall1 (SYS_FILESIZE, fd);
}

int
read (int fd, void *buffer, unsigned size)
{
  return syscall3 (SYS_READ, fd, buffer, size);
}
```

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

void
seek (int fd, unsigned position)
{
  syscall2 (SYS_SEEK, fd, position);
}

unsigned
tell (int fd)
{
  return syscall1 (SYS_TELL, fd);
}

void
close (int fd)
{
  syscall1 (SYS_CLOSE, fd);
}

mapid_t
mmap (int fd, void *addr)
{
  return syscall2 (SYS_MMAP, fd, addr);
}

void
munmap (mapid_t mapid)
{
  syscall1 (SYS_MUNMAP, mapid);
}

bool
chdir (const char *dir)
{
  return syscall1 (SYS_CHDIR, dir);
}

bool
mkdir (const char *dir)
{
  return syscall1 (SYS_MKDIR, dir);
}

bool
readdir (int fd, char name[READDIR_MAX_LEN + 1])
{
  return syscall2 (SYS_READDIR, fd, name);
}

bool
```

```
isdir (int fd)
{
  return syscall1 (SYS_ISDIR, fd);
}

int
inumber (int fd)
{
  return syscall1 (SYS_INUMBER, fd);
}
```

이렇게 다양한 종류의 system call 함수의 동작은 syscall0, syscall1, syscall2, syscall3 이라는 매크로 함수로 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;                                                \
        })

/* 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;                                                \
        })
```

syscall0~syscall3 사이의 차이는 인자를 몇 개 Pass하느냐이다. 즉, syscall handler을 부를 때 stack에 push하는 인자 개수에 따라 함수가 결정된다. syscall0은 인자가 0개, syscall3은 인자를 3개 pass한다. 그리고 공통적인 부분은 stack에 syscall number을 push한다는 것이다. 해당 syscall number는 syscall-nr.h에 상수로 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. */
   ....
  };
```

인자와 number을 모두 stack에 push한 후에는 `int $0x30;`라는 명령어를 실행하는데, `int $0x30;`는 30번째 interrupt을 발생시키는 assembly 명령어다.

`int`라는 명령어는 interrupt 명령어를 인자로 사용하여 interrupt를 발생시키는데 이때 인자가 0x30이 되는 것이다. 즉, 이를 통해 CPU에게 Interrupt가 발생했음을 알리고, 해당 interrupt에 맞는 interrupt service routine이 실행된다. 여기서는 syscall에 대해 특정 interrupt service routine으로 syscall handler가 불려야 하고, 이러한 정보는 interrupt descriptor table이라고 하는 IDT에 저장되어 있다.

2. 그렇다면 어떻게 IDT에 각 interrupt에 대한 handler가 저장되어 있는지 먼저 알아보자.

   먼저 `syscall_init` 함수를 통해서 0x30번째 interrupt name을 “syscall”로, handler 함수를 syscall_handler로 설정하는 과정을 거친다.

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

```
void
intr_register_int (uint8_t vec_no, int dpl, enum intr_level level,
                   intr_handler_func *handler, const char *name)
{
  ASSERT (vec_no < 0x20 || vec_no > 0x2f);
  register_handler (vec_no, dpl, level, handler, name);
}
```

```
/* Interrupt handler functions for each interrupt. */
static intr_handler_func *intr_handlers[INTR_CNT];
/* Names for each interrupt, for debugging purposes. */
static const char *intr_names[INTR_CNT];

static void
register_handler (uint8_t vec_no, int dpl, enum intr_level level,
                  intr_handler_func *handler, const char *name)
{
  ASSERT (intr_handlers[vec_no] == NULL);
  if (level == INTR_ON)
    idt[vec_no] = make_trap_gate (intr_stubs[vec_no], dpl);
  else
    idt[vec_no] = make_intr_gate (intr_stubs[vec_no], dpl);
  intr_handlers[vec_no] = handler;
  intr_names[vec_no] = name;
}
```

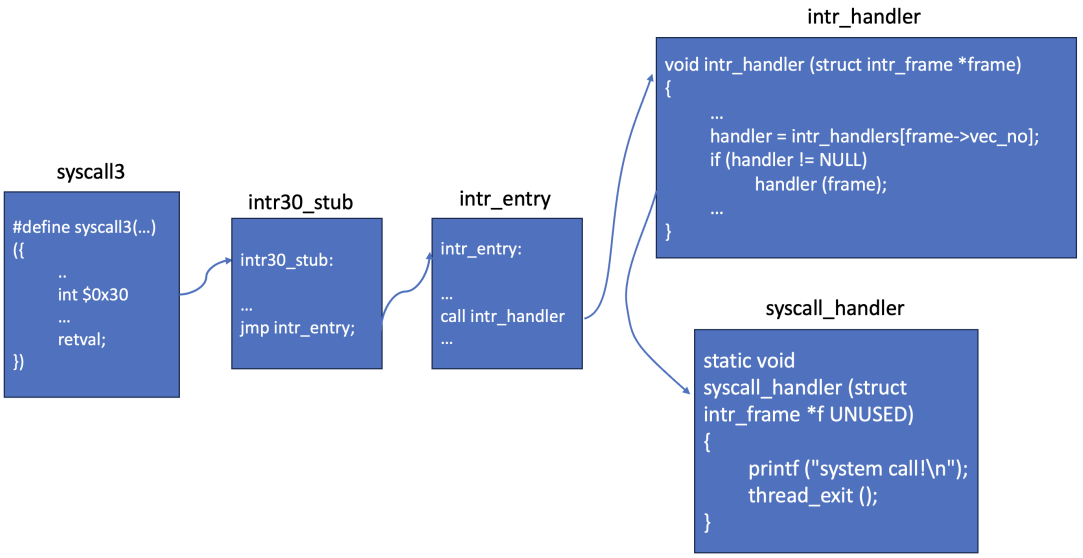
   0x30번째 interrupt name을 “syscall”로, handler 함수를 syscall_handler로 설정하는 과정을 위 `register_handler` 함수 동작을 통해 자세히 살펴보자.

   idt, intr_handlers, intr_names 각 배열에 syscall handling을 위한 정보를 저장하는 함수인데, idt 배열은 interrupt descriptor 배열로 `intr_sturbs[vec_no]` 라는 syscall에 해당하는 `interrupt stub` 함수 주소를 저장하는데, 함수는 각 interrupt에 대해 약간의 stack 조작을 수행한 다음 `intr_entry()` 로 jump하여 interrupt handling을 수행하게 한

다. 이후 intr_handlers 배열에는 handler 함수를 저장하고, intr_names 배열에는 interrupt 이름을 저장한다.

3. 이제 **intr-stubs.S** 파일과 **interrupt.c** 파일 분석을 통해 interrupt handling 함수들이 어떻게 호출되고 수행되는지를 자세히 살펴보자.

   위에서 설명했던 `int $0x30;`이 수행되면 `intr30_stub`가 먼저 수행되는데 각 intr(NUMBER)_stub함수는 intr-stubs.S 파일에 정의되어 있다. `intr30_stub`가 수행된 후 `intr_entry`로 jump하여 수행하고 그 후 `Interrupt handler` 함수를 호출한다. interrupt handler 함수에서는 interrupt vector 번호를 사용하여 intr_handler 배열에서 `해당 interrupt handler 함수`를 찾아 해당 함수를 호출한다.



위 그림과 같은 process로 각 함수들이 차례대로 호출되며 수행되는데 각 함수들의 동작 방식을 분석해보고자 한다.

---

interrupt stub 함수는 각 interrupt에 대한 초기 처리를 수행하는 역할을 한다. 즉, interrupt service routine의 시작 부분에서 handling을 위한 준비 과정을 거치는 함수다.

```
intr_stubs:

/* This implements steps 1 and 2, described above, in the common
   case where we just push a 0 error code. */
#define zero                                    \
  pushl %ebp;                                   \
  pushl $0
```

```
/* This implements steps 1 and 2, described above, in the case
   where the CPU already pushed an error code. */
#define REAL                                    \
        pushl (%esp);                           \
        movl %ebp, 4(%esp)

/* Emits a stub for interrupt vector NUMBER.
   TYPE is `zero', for the case where we push a 0 error code,
   or `REAL', if the CPU pushes an error code for us. */
#define STUB(NUMBER, TYPE)                      \
        .text;                                  \
.func intr##NUMBER##_stub;      \
intr##NUMBER##_stub:                            \
  TYPE;                                  \
  push $0x##NUMBER;                      \
        jmp intr_entry;                         \
.endfunc;         \
                                                \
        .data;                                  \
        .long intr##NUMBER##_stub;
```

`STUB` 매크로 함수는 특정 interrupt number에 대해 Interrupt stub을 생성한다. interrupt stub code를 생성하고 이 stub의 주소를 `intr_stubs` 배열에 저장한다. STUB는 인자로 주어진 TYPE에 따라 `zero` 혹은 `REAL` 매크로 함수를 실행하여 각 interrupt에 대한 초기 처리를 수행한다.

- `zero`는 stack에 %ebp 레지스터 값을 push하고 error code 0을 push하는데, 이는 interrupt에 대한 일반적인 처리 과정이다.
- 이에 반해 `REAL`은 stack의 현재 값인 error code를 복제하여 stack에 push하고 원래 error code의 위치에 %ebp register 값을 저장하는데, 이는 CPU가 error code를 stack에 자동으로 push하는 특정 interrupt에 대해 사용된다.
- 여기서 %ebp register 값을 저장하는 이유는 interrupt가 발생하면 현재 실행 중인 함수의 상태를 저장해야하며, 이는 interrupt handling이 완료된 후에 원래의 실행 흐름을 복구해야 하기 때문이다.
- 또한, 일부 interrupt는 interrupt 발생시 error code를 생성하는데, 이때 error code는 Interrupt의 원인이나 추가 상태 정보를 제공하여 interrupt handling 시에 이 정보를 사용하여 interrupt를 적절히 처리할 수 있다. 이러한 이유로 에러 코드가 제공되지 않은 Interrupt에 대해서는 `zero` 매크로가 사용되며 반대의 경우에는 `real` 매크로가 수행된다.

`zero` or `REAL` 매크로 함수를 실행하여 각 interrupt에 대한 초기 처리를 수행한 이후에는 특정 interrupt 번호를 stack에 push하고 `intr_entry`로 jump하여 실제 interrupt handling을 수행한다.

다음으로 `intr_stub`이 끝나며 jump되어 수행되는 `intr_entry`를 분석해보자.

```
.func intr_entry
intr_entry:
  /* Save caller's registers. */
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  /* Set up kernel environment. */
  cld     /* String instructions go upward. */
  mov $SEL_KDSEG, %eax  /* Initialize segment registers. */
  mov %eax, %ds
  mov %eax, %es
  leal 56(%esp), %ebp /* Set up frame pointer. */

  /* Call interrupt handler. */
  pushl %esp
.globl intr_handler
  call intr_handler
  addl $4, %esp
.endfunc
```

interrupt를 발생시킨 함수(caller)의 register을 모두 stack에 push하고, kernel environment을 set up한 후에 interrupt handler을 호출한다.

먼저 caller의 %ds, %es, %fs, %gs라는 segment register와 general purpose register을 모두 push(pushal)함으로써 원래의 system 상태를 저장하여 interrupt handler 실행이 완료된 후 원래 상태로 복구할 수 있도록 한다.

다음으로는 kernel 환경을 설정하는데 segment register들을 초기화하고, `struct intr_frame`을 저장할 stack frame을 설정한다.

이렇게 Kernel environment 설정까지 끝난 후에는 현재 stack pointer인 %esp를 push하고 `intr_handler` 함수를 호출한다.

여기서 `call intr_handler` 을 하게 되면 `intr_handler`라는 함수가 호출된다. 이 함수는 인자로 intr_frame 구조체를 받는데, 이를 먼저 분석해보자.

`intr_frame` 구조체는 interrupt 발생시에 processor의 상태를 저장하기 위한 구조체다. 즉, interrupt 처리 중에 변경된 register의 원래 값들을 restore하기 위해 필요하다.

```
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. */
  };
```

interrupt가 handling되기 전에 먼저 CPU를 통해 data segment, stack pointer, CPU flag, code segment, next instruction 정보가 먼저 push된다. 이 후에 intr_stubs.Sdml intrrupt stub를 통해 frame pointer, error code, vec_no가 push된다. 그 다음 intr_entry 함수를 통해 ds, es, fs, gs라는 segment register부터 eax, ecx, edx, ebx, esp_dummy, ebp, esi, edi까지 general purpose register가 push된다. 이로써 이렇게 구성된 `intr_frame struct`를 인자를 받아 `intr_handler`가 수행된다.

`void intr_handler (struct intr_frame *frame)`

```
void
intr_handler (struct intr_frame *frame)
{
  bool external;
  intr_handler_func *handler;

  external = frame->vec_no >= 0x20 && frame->vec_no < 0x30;
  if (external)
```

```
    {
      ASSERT (intr_get_level () == INTR_OFF);
      ASSERT (!intr_context ());

      in_external_intr = true;
      yield_on_return = false;
    }

  /* Invoke the interrupt's handler. */
  handler = intr_handlers[frame->vec_no];
  if (handler != NULL)
    handler (frame);
  else if (frame->vec_no == 0x27 || frame->vec_no == 0x2f)
    {
      /* There is no handler, but this interrupt can trigger
         spuriously due to a hardware fault or hardware race
         condition.  Ignore it. */
    }
  else
    unexpected_interrupt (frame);

  /* Complete the processing of an external interrupt. */
  if (external)
    {
      ASSERT (intr_get_level () == INTR_OFF);
      ASSERT (intr_context ());

      in_external_intr = false;
      pic_end_of_interrupt (frame->vec_no);

      if (yield_on_return)
        thread_yield ();
    }
}
```

intr_handler는 external interrupt인지 internal interrupt인지 먼저 확인함으로써 각 type에 맞게 handling을 진행한다.

- 먼저 external interrupt인 경우 interrupt가 비활성화되어 있는지 확인하고 intr_context() 함수를 호출하여 현재 Interrupt context에서 실행 중인지 확인한다. 이를 통해 중복 interrupt 처리를 방지한다. 다음으로 `in_external_intr` 변수를 true로 설정하고, `yield_on_return` 변수를 false로 설정함으로써 external interrupt 처리 중임을 system에게 알린다. 다음으로는 interrupt handler 배열에서 handler을 찾아서 호출하고 처리한다. 이후 interrupt 처리가 완료되면 다시 한번 interrupt가 비활성화되어 있는지 확인하고 현재 Interrupt context에서 실행 중인지 확인한다. 그리고 external interrupt 처리가 완료되었음을 알리고, 변수가 true로 되어 있으면 `thread_yield()` 함수를 호출하여 양보하며 다른 thread에게 실행 제어를 전달한다.
- 만약 internal interrupt인 경우 `intr_handlers[frame->vec_no]`을 통해 해당 interrupt 번호에 대한 handler 함수를 찾고 함수가 있으면 해당 handler 함수를 호출하여

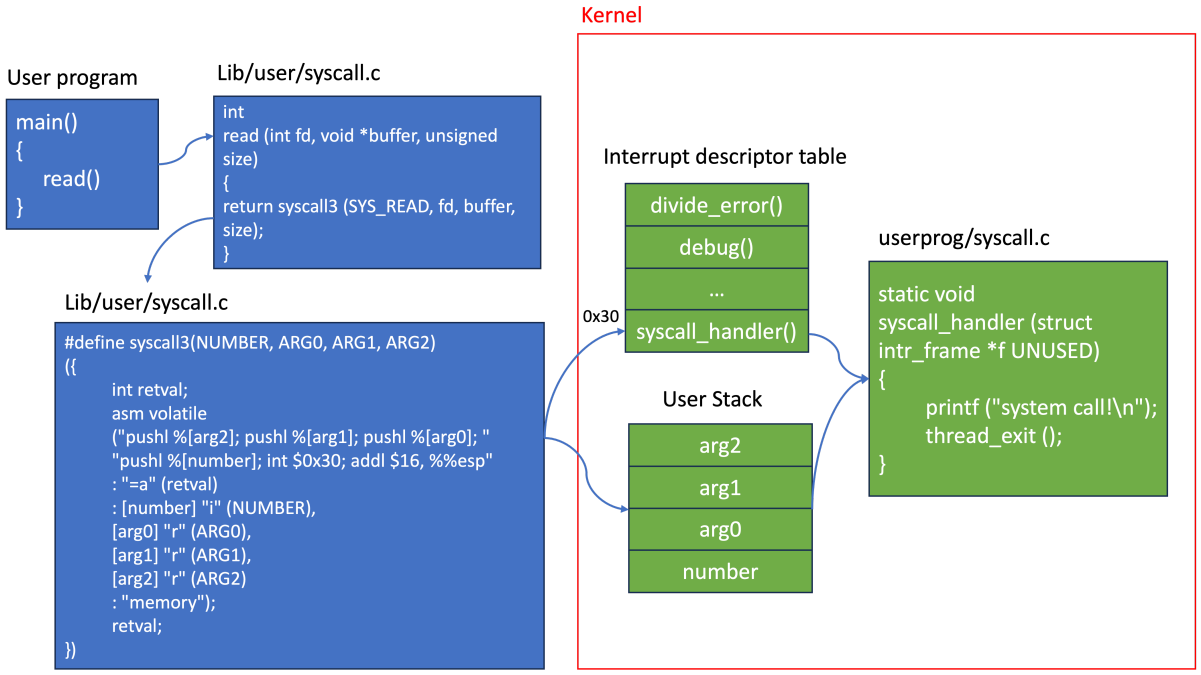
interrupt를 처리한다. 이외에 interrupt 번호가 0x27이거나 0x2f인 경우에는 잘못된 interrupt이므로 무시하며 예상치못한 interrupt가 발생했을 시 `unexpected_interrupt` 함수를 호출하여 처리한다.

다시 우리가 본 과제에서 다루어야 하는 system call에 집중해보면, 위에서 봤듯이 `intr_handlers[frame->vec_no]` 에는 `syscall_handler` 함수가 저장되어 있고 해당 함수는 현재 다음과 같이 구현되어 있다.

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

코드를 통해 알수 있듯 아직 system call을 처리하는 handler 함수의 동작이 구현되어 있지 않은 상태이기 때문에 이를 구현하는 것이 필요하다.

지금까지 전개한 전체적인 동작 방식을 그림으로 나타내면 다음과 같다.



3. Analysis on file system

1. **structure**

   Pintos의 file system을 분석하기 위해서는 먼저 file system에서 2가지 중요한 structure들을 먼저 짚고 넘어가야 한다.

- struct file

```
/* An open file. */
struct file
  {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;            /* Has file_deny_write() been called? */
  };
```

struct file은 file의 inode pointer, 현재 position, deny_write로 구성되어 있다. file 구조체는 파일로부터 read하거나 write할 때 생성되는 객체이나, 파일의 정보을 담고 있지 않고, inode 구조체가 파일의 정보를 담고 있다. file 구조체는 inode pointer을 가지고 있으며 file의 현재 위치와 파일에 대해 write 가능 여부 정보를 가진다.

- struct inode

```
struct inode
  {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    bool removed;                       /* True if deleted, false otherwise. */
    int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;             /* Inode content. */
  };

struct inode_disk
  {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
  };
```

inode는 index node의 줄임말로 file에 대한 전반적인 정보를 담고 있는 구조체다. inode 구조체를 이해하기 위해서는 file에 대한 index 할당 방식을 이해해야 한다.

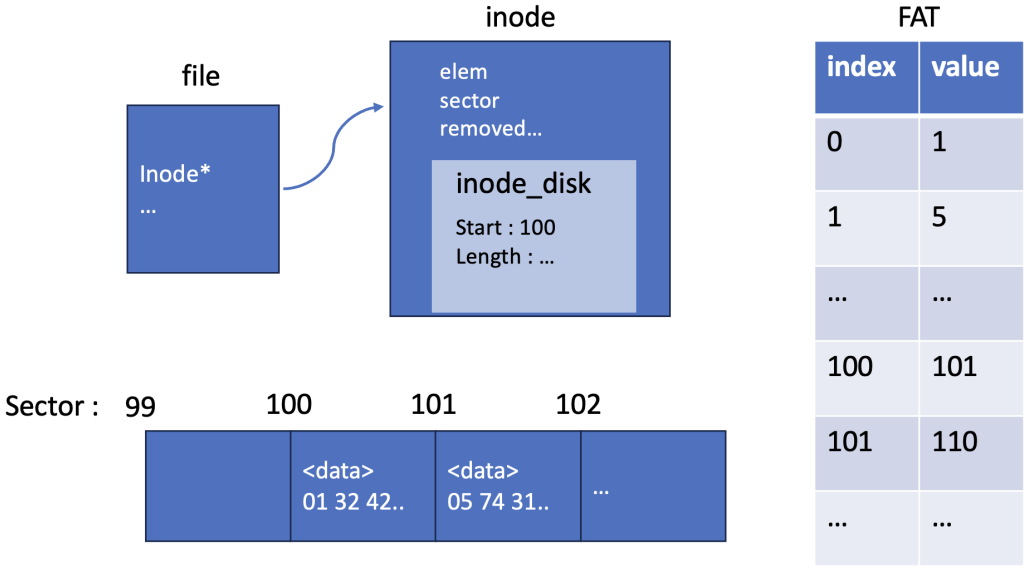
index 할당 방식과 다른 방식으로는 연속 할당 방식이 있는데 이는 하나의 파일이 disk에 연속적으로 저장되는 방식이다. 하지만 disk에 연속적으로 저장되게 되면 external fragmentation 문제가 발생한다. index 할당 방식은 해당 문제가 발생하지 않는데, FAT(file allocation table)을 활용하여 빈 sector들을 찾아 data를 저장하고 파일들이

연결되는 정보를 FAT에 기록하는 것이다. 즉, 파일이 기록된 섹터들의 정보를 저장할 수 있고 연속적으로 저장할 필요가 없기에 문제가 해결되는 것이다.

다시 돌아와서 inode는 결국 파일에 대한 metadata를 가지고 있는 고유 식별자이고, 파일은 각각 inode를 가리키는 inode 포인터를 가지고 있는 것이다. inode 구조체는 `inode_open()` 함수 실행 시 생성되는데 sector number라고 inode_disk가 저장되어 있는 disk의 sector 정보를 저장하고, 이 file을 open한 process와 thread 수, file의 삭제 여부, file의 write 가능 여부(0이면 write 가능), inode_disk를 저장하고 있다. 멤버 변수인 inode_disk 객체에 file의 meta data가 저장되는 것으로 disk에 저장된 file의 metadata를 physical memory에 올리는 개념이다. 즉 매번 disk에 참조할 수 없기 때문에 physical memory에 올려놓고 사용하여 `inode_close()`가 호출될 시 disk에 write back된다. 또한 inode는 list로 관리되기 때문에 inode list에 element로 삽입되기 위해 list_elem type의 elem을 멤버 변수로 갖는다.

inode_disk 구조체는 file의 실제 내용이 저장되어 있는 곳의 시작하는 sector number을 start에 저장하고, 저장된 공간의 길이를 length에 저장한다. 그리고 sector size가 512byte이기 때문에 이를 맞춰주기 위해 나머지 공간이 할당된다.

이를 그림으로 정리하면 다음과 같다.



FAT

index	value
0	1
1	5
...	...
100	101
101	110
...	...

2. Functions

구조체를 알아보았으니 함수를 분석해보자.

filesys/file.c

- **struct file * file_open (struct inode *inode)**

  주어진 inode에 대한 file을 열고 새로운 파일 구조체를 할당하여 반환하는 함수다. inode가 없거나 file 구조체 할당이 실패한 경우에는 NULL을 반환하고 아닌 경우에는 file 구조체를 반환한다. file의 초기 position은 0으로, deny_write는 false로 초기화한다.

- **struct file * file_reopen (struct file *file)**

  기존 파일과 동일한 inode에 대해 새 파일을 열고 반환하는 함수로 실패할 경우 NULL을 return한다. 성공하는 경우에는 file_open 함수의 return 값과 같이 file 구조체를 반환한다.

- **void file_close (struct file *file)**

  file을 close하는 함수로 file이 NULL이 아닌 경우 file에 대한 쓰기 작업을 허용하고 file에 대한 memory를 해제한다. 또한 inode_close 함수를 호출하여 inode를 close하고 disk에 write back한다.

- **struct inode * file_get_inode (struct file *file)**

  filed의 inode를 반환하는 함수다.

- **off_t file_read (struct file *file, void *buffer, off_t size)**

  주어진 file에서 주어진 buffer로 지정된 byte수만큼 읽고 읽은 byte 수를 반환한다. file의 current 위치를 읽은 byte 수만큼 앞으로 이동한다. 즉, pos 값을 증가시킨다.

- **off_t file_read_at (struct file *file, void *buffer, off_t size, off_t file_ofs)**

  주어진 file에서 지정된 offset에서 주어진 buffer로 지정된 byte수만큼 읽고 읽은 byte 수를 반환한다. offset이 지정되어 있기 때문에 file의 current position은 영향 받지 않는다.

- **off_t file_write (struct file *file, const void *buffer, off_t size)**

  주어진 buffer부터 file로 지정된 byte 수만큼 쓰고 실제로 쓴 byte 수를 반환하는 함수다. file의 current position을 Write한 byte수만큼 앞으로 이동한다.

- **off_t file_write_at (struct file *file, const void *buffer, off_t size, off_t file_ofs)**

  주어진 buffer로부터 지정된 offset에서 파일로 지정된 byte수를 쓰고 실제로 쓴 byte 수를 반환하는 함수다. 이로써 file의 current position은 영향을 받지 않는다.

- **void file_deny_write (struct file *file)**

file의 inode에 대한 write를 금지시킨다. file_allow_write()가 호출되거나 file이 close 될 때까지 금지된 것이 유지된다. deny_write 값을 true로 바꾸고 해당 file의 inode에 대해 inode_deny_write() 함수를 호출한다.

- **void file_allow_write (struct file *file)**

  file_deny_write와 반대 작업을 한다. 위에서 설명했듯 file의 deny_write 값을 false로 설정하여 inode에 대한 write의 금지를 풀고 다시 write 작업을 허용시킨다.

- **off_t file_length (struct file *file)**

  inode_length 함수를 사용하여 file의 byte단위 크기를 반환하는 함수다.

- **void file_seek (struct file *file, off_t new_pos)**

  file의 현재 위치를 주어진 새 position으로 설정하는 함수다. 즉 file의 cursor 위치를 변경하는 것이다.

- **off_t file_tell (struct file *file)**

  file의 시작부터 byte offset 단위로 현재 위치를 반환하는 함수다. 즉 file의 cursor 위치를 반환하는 것이다.

filesys/inode.c

- **static inline size_t bytes_to_sectors (off_t size)**

  file size를 인자로 받아 필요한 sector 수를 반환한다. DIV_ROUND_UP 매크로를 사용하여 크기를 BLOCK_SECTOR_SIZE로 나눔으로써 sector 수를 계산하게 된다.

- **static block_sector_t byte_to_sector (const struct inode *inode, off_t pos)**

  file의 특정 byte offset이 포함된 disk sector을 찾아 반환한다. 이 때 pos가 inode의 data length보다 작은 경우에만 sector을 계산하여 반환하고 그렇지 않은 경우에는 -1을 반환한다.

- **void inode_init (void)**

  inode 모듈을 초기화하는 것으로 open_inodes list를 list_init 함수를 통해 초기화하는 함수다.

- **bool inode_create (block_sector_t sector, off_t length)**

  새 inode를 생성하고 disk에 기록하는 함수로 memory 또는 disk 할당에 실패하면 false를 반환하고 성공하면 true를 반환한다. 먼저 free_map_allocate라는 함수를 호출하여 disk에 공간을 할당하고이때 sectors 개의 sector을 할당하고 성공하면 disk_inode→start를 할당된 공간의 시작 sector number로 업데이트한다. 이후에

block_write 함수를 호출하여 disk_inode data를 지정된 sector에 작성한다. 그리고 block write 함수를 호출하여 할당된 sector들에 0으로 초기화해준다.

- **struct inode * inode_open (block_sector_t sector)**

  지정된 sector에서 inode를 읽어서 새 struct inode 객체를 반환하는 함수다. inode가 이미 열려 있으면 기존 객체를 재사용한다. 먼저 open_inodes list에서 이미 열려 있는 Inodes를 찾는다. 이 list는 이전에 열었던 inode들을 저장하는데 각 inode들의 Sector 번호와 인자로 받은 sector을 비교함으로써 일치하면 Inode_reopen을 호출하여 해당 inode를 반환하게 된다. 이 경우가 아니라면 새로운 struct inode를 위해 memory를 할당한다. memory를 할당한 후 inode를 open_inodes list에 추가하고 Inode의 sector 번호를 설정한다. 그리고 inode의 다른 field들을 초기화하고 block_read를 사용하여 disk에서 data를 읽어와 inode→data에 저장한다. 이렇게 초기화된 inode를 반환하게 된다.

- **struct inode * inode_reopen (struct inode *inode)**

  이미 열려 있는 inode를 재사용하는 함수다. 이 때 open_cnt를 증가시킨다.

- **block_sector_t inode_get_inumber (const struct inode *inode)**

  inode의 sector 번호를 반환하는 함수다.

- **void inode_close (struct inode *inode)**

  inode를 close하고 disk에 기록하는 함수다. 마지막 참조인 경우에는 memory를 해제하고 inode가 삭제되면 해당되는 block도 해제한다. inode의 open_cnt를 감소시키고 그 결과가 0이면 inode에 대한 마지막 참조임을 확인할 수 있다. 그 경우 list_remove 함수를 통해 inode list로부터 inode를 제거한다.그리고 inode가 삭제되었는지 확인하고 삭제되었다면 free_map_release 함수를 호출하여 inode의 sector와 inode에 할당된 block들을 해제한다. 또한 memory가 해제되는데 이 모든 경우는 open_cnt가 0인 경우 즉, 마지막 참조인 경우에만 발생한다.

- **void inode_remove (struct inode *inode)**

  마지막에 close될 때 inode를 삭제하는 함수다.

- **off_t inode_read_at (struct inode *inode, void *buffer_, off_t size, off_t offset)**

  지정된 offset에서 inode에서 byte만큼을 읽어 buffer에 저장하고 실제로 읽은 byte 수를 반환하는 함수다. loop를 돌면서 어떤 disk sector에서 읽을 것인지, sector 내에서 어디서부터 읽을 것인지를 결정하게 되고 남은 byte 수가 0이하면 빠져나오게 된다.

- **off_t inode_write_at (struct inode *inode, const void *buffer_, off_t size, off_t offset)**

지정된 offset에서 inode에 byte를 쓴다. 실제로 write한 byte 수를 반환한다. write도 read와 같은 방식으로 loop를 돌면서 byte수만큼 write할 수있게 한다.

- **void inode_deny_write (struct inode *inode)**

  inode에 대한 write를 비활성화하여 금지시킨다.

- **void inode_allow_write (struct inode *inode)**

  inode에 대해 write가 금지되어 있던 것을 다시 활성화시킨다.

- **off_t inode_length (const struct inode *inode)**

  inode의 data length를 byte 단위로 반환한다.

filesys/filesys.c

- **void filesys_init (bool format)**

  file system module을 초기화하는 함수로 block_get_role 함수를 호출하여 file system block device를 가져온다. 그 후 inode_init과 free_map_init 함수를 호출하여 inode와 free map을 초기화한다. format이 true인 경우 do_format()을 호출한다. 마지막으로 free_map_open 함수를 호출하여 free map을 열어둔다.

  free map은 file system에서 사용 가능한 block을 추적하는데 사용되는 data 구조로 어떤 block이 사용가능하고 사용 중인지를 나타낸다. 이로써 file system은 새로운 data를 저장할 위치를 찾을 수 있다. free_map_open은 free map data 구조를 memory에 로드함으로써 free map을 사용하여 사용 가능한 block을 찾고 할당할 수 있다. free_map_close는 free map data 구조의 최신 상태를 disk에 쓰고 필요한 경우 memory에서 free map 데이터 구조를 해제하는 작업을 말하는 것이다.

  - **static void do_format (void)**

    file system을 format하는 함수로 free map을 생성하고 root directory를 생성한다. 마지막으로 free map을 닫아 format이 완료되었음을 알린다.

- **void filesys_done (void)**

  file system module을 종료하고 아직 disk에 쓰지 않은 data를 disk에 쓴다. free_map_close 함수를 호출하여 free map을 닫는다.

- **bool filesys_create (const char *name, off_t initial_size)**

  주어진 name과 Initial size로 file을 생성한다. root directory를 열고 free map에서 inode sector을 할당 받은 다음 해당 sector에 inode을 생성한다. 생성된 inode를 root directory에 추가한다. 실패할 경우 할당 받은 sector을 해제하고 root directory를 닫는다.

- **struct file * filesys_open (const char *name)**

  주어진 name의 file을 여는 함수다. root directory를 열고 그 directory에서 주어진 이름의 inode를 찾는다. 해당 inode를 여러 file을 반환한다.

- **bool filesys_remove (const char *name)**

  주어진 name의 file을 삭제하는 함수다. root directory를 열고 그 directory에서 주어진 이름의 file을 삭제한다.

---

4. How to achieve each requirement

A. Process Termination Messages

1) Problem : Current Implementation

user process가 terminate할 때에는 process name과 exit code를 출력해야한다. 왜냐하면 exit 호출이나 다른 이유에 의해 종료되었을 수 있기 때문이다. kernel thread가 종료될 때나 halt system call이 불렸을 때는 message를 출력하면 안된다. process가 load에 실패한 경우에는 선택적이다.

- 출력 형태는 다음과 같고, process name은 process_execute()를 통해 전달된 full name에서 cmd line 인자를 제외하고 사용한다.

```
printf ("%s: exit(%d)\n", ...);
```

현재 Pintos의 구현 상황은 다음과 같다. user process가 exit()에 의해 종료되고, syscall_handler()가 call되어 “system call!”이 출력되는 것을 확인할 수 있다. 따라서 문제에서 요구하는 서식대로 메세지를 출력할 수 있도록 구현이 필요하다.

2) Our Design

먼저 문제를 해결하기 위해 고려해야 할 것은, 1. 이 termination message를 언제 출력하는가 2. message 출력을 위해 필요한 정보는 무엇인가이다.

1. process termination message 출력 시점

   즉, user process가 언제 terminate하는지를 생각해보면 된다. 앞에서 process execution procedure을 분석했는데, 과정이 정상 작동했을 때 실패하는 경우는 다음의

두 가지 경우다.

- load() failed

  start_process()에서 부른 load() return 값이 false이면 thread_exit()을 호출된다. 이후 thread_exit()함수의 동작을 분석해보면, `#ifdef USERPROG`인 경우에 process_exit()을 호출한다.

- exception 등으로 exit system call이 발생한 경우

2. message 출력에 필요한 정보

   - **process name**

     process execution procedure 분석에서 process_execute()에서 thread_execute()에 인자를 넘겨서 해당 정보로 thread를 실행한다는 것을 확인했다. 인자 중 file_name에 대한 정보가 있는데, 이것을 struct thread의 name member variable로 저장한다. 따라서 thread→name의 값을 받으면 된다.

   - **exit code**

     thread의 exit status를 알아야 한다.

- Code Implementation Design

`void exit(int exit_code)` : termination message를 조건에 맞게 출력하는 기능을 추가한다.

3) Rationales

위의 조건들을 고려해보면, process termination 되어 호출되는 함수에서 process name, exit code의 정보를 받아 print할 수 있는 방법을 설계해야할 것이다. 우리는 thread_exit() 내의 process_exit() 호출 부분에서 출력 정보를 받아 print하는 것보다 C.system call에서 구현해야할 syscall.c의 void exit(int exit_code)에서 구현하는 것이 좋겠다고 생각했다.

B. Argument Passing

1) Problem : Current Implementation

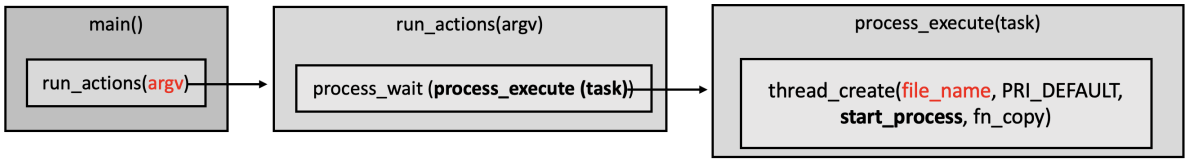
운영체제에서 새로운 process를 실행하기 위해서는 command line을 사용한다. 운영체제는 command line으로 들어온 입력 문자열을 process name, argument 등을 space로 분리하여 인식해야 한다. 현재의 pintos 구현에서는 이러한 기능이 구현되어 있지 않다. 즉,

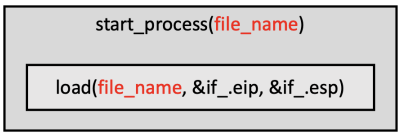
command line에 입력된 내용에서 space로 분리하여 **[process name] [first arg] [second arg]…**로 받는 동작을 구현해야 한다. (cmd line에서 single space는 multiple spaces와 동일하다.)

[program name] [first argument] [second argument] …

2) Our Design

현재 process_execute()은 새로운 process가 arg를 넘기는 것을 지원하지 않으므로 이를 구현해야 한다.





기존의 방식에서는 start_process()에서 호출한 load()에서 해당 process에 대한 memory layout을 구성하게 된다. 이 때 file_name을 인자로 넘기는데, command line에서 받은 입력 정보에 대한 처리 없이 그대로 넘기기 때문에 [process name] 외에 arguments들까지 포함된 값이 넘겨진다.

우리는 start_process()에서 load()가 실행된 이후에 esp는 PHYS_BASE를 가리키는 상태이며, (user stack의 top) 이 사실을 이용해서 argument stack을 구성해가는 것으로 설계했다. 또한 command line에서 [process name]만 따로 받아 file_name으로 파일을 열도록 할 것이다. 그리고 나머지 arguments들은 stack에 쌓는다.

argument를 stack에 set up하는 과정을 pintos.pdf의 3.5.1 Program Startup Details part를 참고하여 분석해보자. kernel은 user program이 실행되기 위한 argument들을 stack에 준비해야만 한다.

“/bin/ls -l foo bar”의 command를 처리하는 과정은 다음과 같다.
1) 가장 먼저 ‘/bin/ls’, ‘-l’, ‘foo’, ‘bar’로 분리한다.
2) 다음으로 이 단어들을 stack의 top에 넣는다. (pointer에 의해 참조될 것이므로 순서는 중요하지 않다.)
3) 각 string에 null pointer를 붙인 것의 address를 stack에 right-to-left order로 push한다.

즉, bar → foo → -1 → /bin/ls 순이다. 이들은 “argv”의 elements들이고, null pointer는 c 언어의 표준에 따라 argv[argc]가 null pointer인 경우를 위한 것이고, order는 argv[0]이 가장 lowest virtual address임을 보장한다.
4) word-aligned access가 빠르기 때문에 performance를 위해 stack pointer를 첫 push 이전의 4의 배수 값으로 round down시킨다.
5) argv와 argc를 차례대로 push한다.
6) 마지막으로 return address을 push한다.
최종적으로 user program 실행 전에는 아래 사진과 같이 stack과 register의 상태는 아래와 같다.
(PHYS_BASE는 0xc0000000로 가정함, stack pointer는 0xbfffffcc로 초기화되어야 함)

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 (*) ()

picture in pintos.pdf

위와 같은 과정을 구현하기 위해 이미 존재하는 process.c의 코드 중, process_execute(), start_process()의 함수를 수정하고, put_argv_stack()이라는 함수를 새로 구현할 것이다.

먼저 `process_execute()`에서는 argv로 넘겨받은 string을 parsing하여 첫번째 인자, 즉 process name 값을 file_name으로 저장한다. 이렇게 얻은 file_name을 process_execute()에서 thread_create()로의 인자로 전달하고, start_process()에서 load()를 실행하는데 인자로 전달할 것이다. string parsing을 위해 string.c의 strtok_r을 이용하고, 이외의 string 관련 함수들을 이용하여 argument를 parsing하고 값을 이용할 것이다. 이 함수를 구현하기 위해서 palloc_get_page()를 이용할 것이므로, 이후에 palloc_free_page()를 이용하여 memory issue가 발생하지 않도록 할 것이다. 이때 중요한 것은, strtok_r을 이용하여 parsing하면 원래 string을 변경하면서 동작하기 때문에 원래 값

을 보존하는 것이 필요하다는 점이다. 왜냐하면 process_execute()에서 start_process()를 호출할 때 parsing 전의 command line을 넘겨주어, 이용하게 하기 때문이다. 이를 위해서 strtok_r을 실행하기 전에 copy version의 변수를 만들어서 넘길 것이다.

`start_process()` 에서는, process_execute()에서 parsing하기 전의 기존 command line의 정보를 인자로 받아 처리할 것이다. load()를 호출할 때 이전에는 단순히 file name을 인자로 넣었는데, 이제는 parsing 결과의 첫번째 값인 실제 file name을 인자로 넣는다. load의 결과가 success일 때 put_argv_stack을 호출하는데에 사용할 것이다. process_execute()와 같은 방식으로 string.c의 함수를 이용해 구현할 것이다.

`put_argv_stack` 은 file_name과, &if_.esp를 인자로 받아 argument stack을 set up하는 함수다. 위에서 Pintos.pdf를 분석한 것과 동일하게 실행한다. parsing 결과 process name 이후의 arguments들을 passing 해야 하는데, 이를 위한 stack을 set up한다. 위에서 분석한 방식에 따라 구현한다.

3) Rationales

- stack pointer를 4의 배수로 round down하여 정렬하는 이유는?

memory alignment 규칙에 따라서, 이러한 식으로 alignment를 맞추는 것이 성능이 높기 때문이다. 이는 CPU에서 memory access하는 것과 관련있는데, data를 위해 memory access를 하는 cost를 줄이고 효과적으로 접근하기 위해 word size인 byte로 정렬한다.

C. System Calls

1) Problem : Current Implementation

현재는 system call handler는 아래와 같이 구현되어 있으며 동작 자체는 구현되어 있지 않은 상태다.

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

각 system call 종류에 맞게 적절한 system call handler가 실행되도록 구현해야 하며 아래 2개의 type에 맞는 system call handling 함수를 구현해야 한다.

- User Process Manipulation
- File Manipulation

2) Our Design

system call handler의 전체 동작은 switch문으로 stack에 저장된 syscall_number에 따라 각 syscall 종류에 대한 handling을 하는 것이다.

syscall_nr.h을 보면 우리가 handling 동작을 구현해야 하는 system call 종류는 halt, exit, exec, wait, create, remove, open, filesize, read, write, seek, tell, close가 있고 이를 모두 구현해야 한다.

- syscall_handler

  interrupt frame에서 Stack의 top 주소인 esp 값을 확인하여 해당 주소가 user 영역인지 확인하는 과정을 먼저 거친다.

  - `bool is_valid_addr(void *addr)`

    is_user_vaddr(addr)을 호출하여 true면 true를 return하고 false면 false를 Return한다.

  그 후에 stack Pointer 주소에 저장되어 있는 syscall number을 가져오고 이를 사용하여 switch문을 사용하여 각 syscall number에 맞게 위에서 구현한 함수들을 호출하게 한다.

  - `int syscall_num = *stack_ptr;`

  각 함수를 호출하기 전에 stack pointer 값을 사용하여 stack에 있는 argument들을 가져오고, 이때 argument를 각 syscall handling 함수가 요구하는 개수만큼 가져와야 한다.

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

    esp로 가리켜진 memory 주소로부터 지정된 개수(count)만큼 for문을 돌며 argument를 stack에서 읽어서 arg 배열에 저장하도록 구현한다. 이 때 각 인자들이 저장된 주소가 유효한지 확인하는 과정을 거치며 유효하지 않은 경우에는 exit(-1)을 호출하여 프로그램을 종료한다.

  이후 syscall handling 함수 실행 후에 결과값을 return하는 경우에는 이 값을 eax에 저장하도록 한다.

  전체 과정을 모두 반영한 syscall_handler 함수의 pseudo code는 다음과 같다.

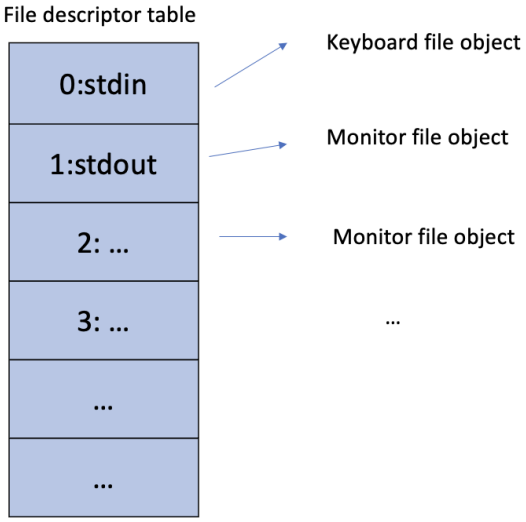
```
1. Retrieve the stack pointer and assign it to stack_ptr.

2. If the address of stack_ptr is not valid:
    Exit the program with a return code of -1.

3. Get the value at the location pointed by stack_ptr and assign it to syscall_nu
m.
Initialize an integer array argv[3].

4. According to the value of syscall_num,
    SYS_HALT:
        Call the halt() function.
    SYS_EXIT:
        Store 1 argument in argv.
        Call the exit function with value argv[0].
    SYS_EXEC:
        Store 1 argument in argv.
        Call the exec function with value argv[0] and assign the result to eax.
    SYS_WAIT:
        Store 1 argument in argv.
        Call the wait function with value argv[0] and assign the result to eax.
    ...
    Any other value:
        Exit the program with a return code of -1.
```

syscall_handler에서 각 syscall number에 대해 handling해주는 함수를 아래와 같이 구현하고자 한다. 구현하기전 각 함수의 type에 따라 분류하며 정리하고자 한다.

- user process 관련 system call
  - halt, exit, exec, wait
  - 먼저 현재 pintos는 process 간의 parent-child 관계를 명시하는 정보가 없어서 자식 process의 정보를 알 수 없기 때문에 부모 process가 자식의 시작 혹은 종료 전에 종료되어버리는 문제가 발생하고 있다. 즉 자식 process가 시작하지도 못하는 문제가 발생하는 것이다.
    - 먼저 thread struct에 parent, child 관련 정보를 담는 field를 추가한다. 자식 process가 성공적으로 생성되었는지를 나타내는 flag, process의 종료 유무 필드, 정상적으로 종료되는지 status 확인하는 필드, process wait를 위한 semaphore 필드, 자식 process list, 부모 Process를 가리키는 필드를 추가하고자 한다. init_thread 함수에 해당 field들을 초기화하는 코드를 추가한다.
    - 이를 관리하는 함수를 만들어야 하기 때문에 자식 process는 list로 구현하고, 자식 list에서 원하는 process를 검색하고 삭제하는 함수를 구현하고자 한다.

        - get_child_process() : 현재 process의 자식 list를 검색하여 해당 pid에 맞는 thread 구조체 포인터를 return하고 존재하지 않은 경우 null을 반환하도록 구현한다.
        - remove_child_process() : 자식 process의 thread 구조체를 자식 list에서 제거하고 memory도 해제한다.

- file system 관련 system call
    - create, remove, open, filesize, read, write, seek, tell, close
    - 먼저, 위에서 분석했듯 pintos의 file system과 관련 함수들은 이미 구현되어 있으며 이번 과제는 pintos의 file system 구조를 파악해서 이들을 적절히 사용하는 구현에 초점이 맞춰져있다.
    - 각 thread는 file들에 접근할 때 **file desriptor를** 사용해야 하는 것이 구현 조건이다.
        - file descriptor는 정수값을 가지고 thread가 어떤 file을 open하면 kernel은 현재 사용하지 않는 가장 작은 file descriptor 값부터 차례대로 할당하게 된다.
        - 0,1은 stdin, stdout을 가리키는 index이기 때문에 2부터 할당할 수 있다. 즉 이렇게 할당된 file descriptor 값을 저장한 것이 file descriptor table인데 각 thread마다 file descriptor table을 가지고 있으며 이를 통해 file에 접근할 수 있는 것이다.



    - 하지만 이러한 부분이 pintos에는 현재 구현되어 있지 않으므로 thread가 filed에 접근하는 기능을 가지고 있지 않다. 따라서 이를 구현해야 한다.

본격적으로 각 함수를 구현해보자.

- `void halt (void)`

  shutdown_power_off() 함수를 호출하여 pintos를 종료하게 한다.

- `void exit (int status)`

  현재 thread의 exit status를 설정해주고 thread가 종료됨을 print한 후 thread_exit() 함수를 불러 thread를 종료시킨다.

- `pid_t exec (const char * cmd_line)`

  자식 process를 생성하고 program을 실행시키는 system call이기 때문에 process를 생성하는 함수를 사용해야 한다. process 생성을 성공할시 pid 값을 반환하고 실패할 경우 -1을 반환한다. 이 때 부모 process는 자식 Process의 program이 memory에 다 load될 때까지 wait해야 하고 이를 semaphore을 사용해서 구현한다. 자식 process의 program이 memory에 다 load되면 user program을 실행하고 실패하면 종료한다.

- `int wait (pid_t pid)`

  자식 process가 모두 종료될 때까지 대기해야 하고, 올바르게 종료되었는지 확인도 해야 한다. 이를 위해 process_wait 함수를 호출한다. process wait 함수에서는 pid에 해당하는 child를 찾고 semaphore을 사용하여 Child가 종료될 때까지 기다린다. 이후 sema_down에 성공한 경우 자식 process의 exit status를 return한다.

- `bool create (const char * file , unsigned initial_size )`

  주어진 file을 initial size만큼 만드는 것이다. 이를 위해 filesys_create를 호출한다. success하면 ture를 fail하면 false를 return한다.

- `bool remove (const char * file )`

  주어진 file을 delete하는 것으로, 이를 위해 filesys_remove를 호출한다. success하면 true를 fail하면 false를 return한다.

- `int open (const char * file )`

  filesys_open을 호출하여 file을 open하고 file에 대해 file descriptor을 부여한다. 그리고 file descriptor 값을 반환하며 file이 존재하지 않은 경우 -1을 반환한다.

- `int filesize (int fd )`

  file descriptor을 이용하여 file 객체를 검색하고 file의 길이를 return한다. 존재하지 않을시 -1을 return한다.

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

  file에 동시에 read하는 것을 막아야하기 때문에 **lock mechanism을 사용한다.** 먼저 lock을 acquire하고 나머지 코드를 실행한 후 lock을 release한다. 먼저 file descriptor

가 0인 경우 key board 입력을 buffer에 저장하고 나머지의 경우 file_read 함수를 호출하여 size만큼 읽는다. 그리고 읽은 만큼 return한다.

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

  read와 마찬가지로 **lock을 acquire한 후** write하도록 한다. file descriptor가 1인 경우 buffer에 저장된 값을 화면에 출력하고 나머지의 경우 buffer에 저장된 값을 file에 기록하고 기록한 byte수를 return한다.

- `void seek (int fd, unsigned position)`

  file_seek를 호출하여 파일의 커서를 이동한다.

- `unsigned tell (int fd)`

  file_tell을 호출하여 파일의 현재 커서를 알려준다.

- `void close (int fd)`

  process_file_close를 호출하여 file을 닫는다.

3) Rationales

1. syscall_handler 함수

- stack에서 어떤 값을 읽기 전에 stack pointer가 가리키는 주소가 유효한지 is_valid_addr을 통해 검사한다. 이러한 과정은 memory에 대한 잘못된 접근을 방지하여 잘못된 동작으로 인해 예상치 못한 결과 혹은 오류가 나오는 것을 방지하고자 한 것이 근거라고 할 수 있다.
- syscall 함수 호출을 처리할 때 각 case별로 독립적이고 명확하게 동작하게 하기 위해 switch문을 구성하였고, 각 case마다 argument를 받아와야 하며 인자 개수가 다 다르기 때문에 case문 안에서 get_argument 함수를 호출하게 구현한다.

2. syscall 함수

- wait, exec, exit, open, write와 같은 syscall 함수 구현은 기존 pintos 문서와 requirement 문서에 따라 manually 구현하였다.

D. Denying Writes to Executables

1) Problem : Current Implementation

현재 pintos는 실행 중인 user process의 file에 다른 process가 data를 write하는 것을 막는 부분이 구현되어 있지 않다. 이렇게 되면 실행 중인 사용자 프로그램의 file이 다른 process에 의해 변경되면 disk에서 변경된 data를 읽어오기 때문에 의도한 방향과 다르게 program result를 도출할 수 있다.

2) Our Design

OS는 실행 중인 process의 program file에 대한 쓰기 접근을 막아야 하기 때문에 pintos도 이러한 역할을 하도록 구현을 하고자 한다. 위에서 분석했던 함수 file_deny_write()와 file_allow_write() 함수를 사용하여 file이 process에 의해 접근되고 있을 때 file이 변경되는 것을 막을 것이다.

- 먼저 thread 구조체에 새로운 필드를 추가하여, 현재 실행 중인 file을 가리키는 포인터를 저장하도록 한다.
- file이 open되는 시점에 load 함수가 호출되어 동작하기 때문에 load 함수를 수정하고자 한다. load 함수에서 file_deny_write()을 호출하여 write 권한을 막아버린다. 그리고 이 때 file 접근을 여러 thread에서 하지 못하게 하기 위해 lock mechanism을 추가로 구현하고자 한다. 즉 lock을 acquire한 후 file_deny_write()을 호출하게 하고 이 후 lock을 release한다.
- 또한 위에서 구현한 open 함수에서도 file을 open하기 때문에 마찬가지로 file_deny_write()을 호출하도록 한다.
- file이 close되는 시점에는 file_allow_write()을 호출해야하는데, 이는 이미 file_close 함수에 구현되어 있다.

3) Rationales

file이 multi process 환경에서 data를 consistent하게 보관하기 위해서는 write을 막는 기능을 구현하는 것이 필요하다. 즉, 특정 process가 접근하고 있는 상황에서 다른 process가 file을 수정하여 내용이 변경되면 안된다. process가 file을 접근하는 상황을 생각해보면, file을 open할 때 해당 file에 다른 process들의 write 권한을 막으면 한 process만이 file에 write할 수 있게 되므로 consistency가 보장된다.