Lab2 Desgin report

20210389 최준호, 20210509 정윤재

개요

- 본 과제는 user process와 kernel간의 발생하는 문제를 다루고자한다. user process는 privilege mode에 대한 접근 권한이 없다. 따라서 system call을 사용해서 kernel의 도움을 받아 privilege instruction들을 처리해줘야한다.
- 본 과제는 argument passing과 system call 두 가지 부분으로 나눠진다. 첫번째 argument passing의 경우 user program을 실행하기 전에 kernel은 function의 argument를 저장해줘야한다. `process_execute` 함수는 user가 instruction을 수행할 수 있도록 process를 memory에 load하는 기능을 수행한다. 그러나 현재의 구현은 `process_execute`가 새로운 process에 대한 argument passing 기능을 제공하지 않는다. 따라서, 이 기능을 구현해야 한다. 두번째로 system call이 발생한 경우 handler에서 system call number에 따른 올바른 처리를 해주면 된다.

Analysis on process execution procedure

현재 pintos system에서 process execution이 어떻게 일어나는지를 알아보기 위해 `threads/init.c`와 `userprog/process.c` 파일의 주요함수들을 살펴보았다.

```
int main (void) {
  char **argv;

  bss_init ();

  argv = read_command_line ();
  argv = parse_options (argv);

  thread_init ();
  console_init ();

  printf ("Pintos booting with %'"PRIu32" kB RAM...\n",
          init_ram_pages * PGSIZE / 1024);

  palloc_init (user_page_limit);
```

```
  malloc_init ();
  paging_init ();

#ifdef USERPROG
  tss_init ();
  gdt_init ();
#endif

  intr_init ();
  timer_init ();
  kbd_init ();
  input_init ();
#ifdef USERPROG
  exception_init ();
  syscall_init ();
#endif

  thread_start ();
  serial_init_queue ();
  timer_calibrate ();

#ifdef FILESYS
  ide_init ();
  locate_block_devices ();
  filesys_init (format_filesys);
#endif

  printf ("Boot complete.\n");

  run_actions (argv);

  /* Finish up. */
  shutdown ();
  thread_exit ();
}
```

- kerenl은 main 함수로 시작하게 되는데, `bss_init`으로 kernel의 bss를 clear하고, `read_comand_line`로 command line을 argument로 나눠준 뒤, `parse_option`으로

command line을 parse option으로 나눠준다.

- `thread_init`, `palloc_init`, `malloc_init`, `paging_init`을 차례대로 호출하여 thread와 memory system을 초기화해준다. 그리고 userprogram인 경우, `tss_init`과 `gdt_init`을 추가적으로 호출한다. 여기서 TSS, Task State Segment는 pintos에서 user process가 interrupt handler로 진입할 때, stack을 switch 하기 위해서 사용한다. 그리고 GDT, Global Descriptor Table는 각 segement들을 describe하는 table이다.
- 추가적으로 `intr_init`, `timer_init`, `kbd_init`, `input_init`를 실행함으로써 interrupt handler를 초기화한다. 그리고 userprog인 경우, `exception_init`과 `syscall_init` 으로 interrupt handling을 세팅한다.
- 이후, `thread_start`로 thread scheduler를 시작하고 interrupt를 enable한다. 그리고 file system이 컴파일되었다면 `ide_init`으로 IDE disk를 초기화하고, `filesys_init`으로 file system을 초기화한다.
- 마지막으로 저장해둔 `argv`를 `run_actions`의 argument로 넘겨준다.

```
static void run_actions (char **argv) {
  /* An action. */
  struct action
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, inclu
ding action name. */
      void (*function) (char **argv);   /* Function to exec
ute 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)", *ar
gv, a->argc - 1);

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

}
```

- action structure를 살펴보면, `name`, `argc`, `func(argv)`로 구성되는 것을 볼 수 있다. 예를 들어, {"run", 2, run_task}와 같이 action의 action object가 설정되어있는 것을 볼 수 있다. 그리고 이후의 과정을 살펴보면, argument로 받은 `argv[0]`와 일치하는 name을 가진 action object를 찾은 뒤, `argv[0]`이후의 argv에 필요한 argument들이 `argc`만큼 있는지 확인한 뒤, func(argv)를 호출한다.

```
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_aciton` 함수에서 "run"이라는 action을 읽게되면, 등록된 `run_task` 함수가 실행된다. 그리고 함수를 보면 userprog인 경우, process_execute 함수를 통해 argument로 받은 `argv[1]` 에 대한 user process 를 만들고 `process_wait` 함수를 실행하는 것을 볼 수 있다.

```
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_proces
s, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

- caller와 load의 race를 막기 위해서 page를 새로 할당 받아 fn_copy에 file_name을 복사한다. 그리고 `thread_create`를 이용해 입력 받은 `file_name`을 이름으로 하고 `PRI_DEFAULT`를 priority로 갖는 thread를 생성한다. `thread_create`는 생성한 thread

의 tid 를 return 한다. 이후, thread는 `fn_copy`를 argument로 `start_process` 를 실행한다.

```
int process_wait (tid_t child_tid UNUSED) {
  return -1;
}
```

- thread가 종료될 때까지 wait하다가 exit status를 return하는 함수이다. kernel에 의해 exception이 발생해서 종료되는 경우 -1을 return한다. 현재는 그냥 -1을 return하고 있다.

```
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_f
rame',
     we just point the stack pointer (%esp) to our stack fr
ame
     and jump to it. */
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&i
```

```
f_) : "memory");
  NOT_REACHED ();
}
```

- `start_process`는 `thread_create`로 생성된 thread가 실행하는 함수로, file_name을 memory에 load 하는 함수이다.
- 가장 먼저 struct intr_frame 변수 `if_`를 초기화하는데, 이 변수는 process가 실행될 때 필요한 정보를 담고 있다.
- 이후, `load` 함수를 호출되면서 `file_name`의 process에 해당하는 executable file을 memory에 load한다. 그리고 실행의 시작 위치 (`%eip`)와 stack pointer (`%esp`)를 설정해준다. 이 과정을 성공하면 succes가 true가 되고, 실패 시 `thread_exit`을 호출해 process의 실행을 중단한다.
- executable file이 성공적으로 memory에 적재되면, user process가 시작된다. `intr_exit`함수는 stack에 저장된 저장된 정보를 struct intr_frame의 형태로 가져간다. 그리고 `intr_exit`을 호출하기 전에 `%esp`를 `if_` 가 위치한 주소로 설정해주면, `intr_exit`은 `if_`에 저장된 상태로 프로그램을 시작할 수 있다. 이는 CPU가 interrupt에서 복귀하는 것처럼 동작하게 한다.

```
bool load (const char *file_name, void (**eip) (void), void
**esp) {
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();

  /* Open executable file. */
  file = filesys_open (file_name);
  if (file == NULL)
    {
```

```
      printf ("load: %s: open failed\n", file_name);
      goto done;
    }

  /* Read and verify executable header. */
  if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
      || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
      || ehdr.e_type != 2
      || ehdr.e_machine != 3
      || ehdr.e_version != 1
      || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
      || ehdr.e_phnum > 1024)
    {
      printf ("load: %s: error loading executable\n", file_
name);
      goto done;
    }

  /* Read program headers. */
  file_ofs = ehdr.e_phoff;
  for (i = 0; i < ehdr.e_phnum; i++)
    {
      struct Elf32_Phdr phdr;

      if (file_ofs < 0 || file_ofs > file_length (file))
        goto done;
      file_seek (file, file_ofs);

      if (file_read (file, &phdr, sizeof phdr) != sizeof ph
dr)
        goto done;
      file_ofs += sizeof phdr;
      switch (phdr.p_type)
        {
        case PT_NULL:
        case PT_NOTE:
        case PT_PHDR:
        case PT_STACK:
```

```
        default:
          /* Ignore this segment. */
          break;
        case PT_DYNAMIC:
        case PT_INTERP:
        case PT_SHLIB:
          goto done;
        case PT_LOAD:
          if (validate_segment (&phdr, file))
            {
              bool writable = (phdr.p_flags & PF_W) != 0;
              uint32_t file_page = phdr.p_offset & ~PGMASK;
              uint32_t mem_page = phdr.p_vaddr & ~PGMASK;
              uint32_t page_offset = phdr.p_vaddr & PGMASK;
              uint32_t read_bytes, zero_bytes;
              if (phdr.p_filesz > 0)
                {
                  /* Normal segment.
                     Read initial part from disk and zero t
he rest. */
                  read_bytes = page_offset + phdr.p_filesz;
                  zero_bytes = (ROUND_UP (page_offset + phd
r.p_memsz, PGSIZE)
                                - read_bytes);
                }
              else
                {
                  /* Entirely zero.
                     Don't read anything from disk. */
                  read_bytes = 0;
                  zero_bytes = ROUND_UP (page_offset + phd
r.p_memsz, PGSIZE);
                }
              if (!load_segment (file, file_page, (void *)
mem_page,
                                 read_bytes, zero_bytes, wr
itable))
                goto done;
```

```
            }
          else
            goto done;
          break;
        }
    }

  /* Set up stack. */
  if (!setup_stack (esp))
    goto done;

  /* Start address. */
  *eip = (void (*) (void)) ehdr.e_entry;

  success = true;

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

```
void process_activate (void) {
  struct thread *t = thread_current ();

  /* Activate thread's page tables. */
  pagedir_activate (t->pagedir);

  /* Set thread's kernel stack for use in processing
     interrupts. */
  tss_update ();
}
```

```
static bool load_segment (struct file *file, off_t ofs, uin
t8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, boo
```

```
l writable) {
  ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
  ASSERT (pg_ofs (upage) == 0);
  ASSERT (ofs % PGSIZE == 0);

  file_seek (file, ofs);
  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* Calculate how to fill this page.
         We will read PAGE_READ_BYTES bytes from FILE
         and zero the final PAGE_ZERO_BYTES bytes. */
      size_t page_read_bytes = read_bytes < PGSIZE ? read_b
ytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      /* Get a page of memory. */
      uint8_t *kpage = palloc_get_page (PAL_USER);
      if (kpage == NULL)
        return false;

      /* Load this page. */
      if (file_read (file, kpage, page_read_bytes) != (int)
page_read_bytes)
        {
          palloc_free_page (kpage);
          return false;
        }
      memset (kpage + page_read_bytes, 0, page_zero_bytes);

      /* Add the page to the process's address space. */
      if (!install_page (upage, kpage, writable))
        {
          palloc_free_page (kpage);
          return false;
        }

      /* Advance. */
      read_bytes -= page_read_bytes;
```

```
      zero_bytes -= page_zero_bytes;
      upage += PGSIZE;
    }
  return true;
}
```

- `load`는 `pagedir_create` 함수를 호출하고 user process의 page table을 생성한다. 그리고 현재 thread t의 `t → pagedir`에 저장하고, 그 값이 NULL이라면 done으로 이동하여 `file_close` 함수를 호출한다. 그리고 page table이 정상적으로 생성되었다면, `process_activate` 함수를 호출하는 것을 볼 수 있다. `process_activate` 는 `t → pagedir`를 argument로 `pagedir_activate`를 호출해 page table을 활성화하고, tss를 update 해준다.
- 이제, `filesys_open`을 호출하여 file_name을 이름으로 갖는 file을 open 하여 filed에 저장한다. 그리고 file이 NULL이라면, done으로 이동한다.
- 이후, `file_read` 함수를 호출하여 file을 읽고 `file_seek` 함수를 통해 file의 배치 정보를 확인한다. 그리고 이를 바탕으로 `load_segment` 함수를 실행하여 file을 memory에 불러온다. 마지막으로 `setup_stack` 함수를 호출하여 stack을 초기화한다.

  → 즉, `load`는 binary file을 disk에서 memory로 load하고, user stack set을 initialize 하는 함수이다.
- 이 외에도 load는 ELF file을 load해 run하기 위한 data, text segment를 초기화한다.

```
static bool setup_stack (void **esp) {
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGS
IZE, kpage, true);
      if (success)
        *esp = PHYS_BASE;
      else
        palloc_free_page (kpage);
    }
```

```
  return success;
}
```

- `palloc_get_page`를 사용해 사용자 모드에서 사용할 페이지를 할당하고, 0으로 초기화한다. 실패하면 `NULL`을 return한다. `kpage`를 user stack의 가장 높은 address(`PHYS_BASE - PGSIZE`)에 mapping한다. 그리고 `install_page` 함수가 성공하면 `success`를 `true`로 설정한다. 이제 mapping이 성공하면 `%esp`를 user virtual space의 최상위 주소 `PHYS_BASE`로 설정한다.

80x86 Calling Convention (funciton call in UNIX 32bit 80x86 implementation) (pintos 문서)

- caller가 `push`를 이용하여 function argument들을 stack에 하나씩 넣어준다. (오→왼의 순서)
- caller는 next instruction address를 stack에 저장해 함수가 종료된 뒤, 다시 원래 위치로 돌아갈 수 있게 한다. `call`은 return address를 stack에 저장한 뒤, callee의 첫 번째 instruction의 address로 jump하게 한다.
- callee가 실행되면, `%esp`는 return address를 가리키게 된다. 이때, stack은 호출된 함수의 argument와 return address를 포함하고 있습니다. callee는 이 stack을 이용할 수 있다.
- callee가 return 값을 가질 경우, 그 값을 `%eax`에 저장한다. 이를 통해 caller는 return 값을 알 수 있다.
- callee가 끝나면, `%ret`를 사용하여 return address를 pop하고 그 주소로 jump하여 caller의 다음 instruction을 실행하게 된다.
- caller는 `pop`를 사용하여 stack을 정리하거나, stack pointer를 조정한다.

Analysis on system call procedure

Virtual memory Layout

- Pintos에서 구현하는 virtual memory는 0 ~ `PHYS_BASE` (3GB)의 user virtual memory와 `PHYS_BASE` ~ 4GB의 kernel virtual memory 두 영역으로 나뉘어진다.
- **user virtual memory**는 process 별로 고유하게 설정된다. 따라서, kernel이 다른 process에서 다른 process로 switch할 때, user virtual memory도 함께 switch되어야한다. 이 과정은 processor의 page directory base register를 변경함으로써 진행된다.

  → `userprog/pagedir.c`의 `pagedir_activate()`

    - 실제 physical address와의 mapping은 page table에 기록되어있는데, 각 process는 `struct thread`에 page_table pointer를 포함하고 있어 해당 process의 page table을 참조할 수 있다.

- **kernel virtual memory**는 global로 설정된다. user process가 동작하든 kernel thread가 동작하든 항상 같은 방식으로 mapping된다. kernel virtual memory는 `PHYS_BASE` 에서 부터 memory의 `0x0` 주소에 1:1로 매핑된다. 예를 들어, `PHYS_BASE + 0x1234`라는 virtual address는 physical address `0x1234`를 참조한다.
- **user program**은 자신의 user virtual meory에만 접근 가능하다. 만약, kernel virtual memory에 접근하려하는 경우, page fault를 일으킨다.
- **kernel thread**는 kernel virtual memory에 접근가능하다. 그리고 process가 running state라면, user virtual memory에도 접근 가능한다. 그러나, unmapped user virtual address에 접근하는 경우 page fault가 일어난다.

Accessing User memory

- System call을 호출하는 과정에서 kernel은 user program이 제공하는 pointer로 memory에 접근해야하는 경우가 존재한다. 그리고 user program은 user memory stack에 system call number와 필요한 argument들을 저장한 뒤, interrupt를 발생시킨다. 이 과정에서 아래의 이유들로 발생하는 invalid pointer가 문제가 될 수 있다.
    - null pointer
    - pointer on unmapped virtual memory
    - pointer on kernel virtual address (above `PHYS_BASE`)

→ 이렇게 invalid pointer가 발생하는 경우, 해당 process를 종료하고 관련 resource를 해제함으로써 kernel이나 실행중인 다른 process에 영향을 미치지 않도록 헤야한다.

- 따라서, user memory에 접근할 때는 pointer의 유효성을 확인해줘야한다. Pintos.pdf에 따르면 아래의 두가지 방법을 사용한다.
    - user가 제공한 pointer의 유효성을 전부 검증한 뒤, dereference하는 방법
        - thread/mmu.c (in include/threads/vaddr.h)
    - user pointer가 `PHYS_BASE` 아래의 pointer인지만 확인한 뒤, dereference하는 방법
        - invalid pointer는 page fault를 일으킨다. (userporg/exceptions.c)
        - 나머지 처리는 page_fault() 함수 내부에서 처리하는데, 이러한 방법이 processor의 mmu를 이용하기 때문에 더 빠르다고 한다. (실제 kernel에서 이용)

두 방법을 사용할 때, 모두 resource leak가 발생하지 않도록 주의해야한다. 예를 들어 system call이 lock을 획득하거나 malloc()을 통해 memory를 할당받은 경우, invalid pointer를 받으면 lock을 헤제하거나 page memory를 free 해야한다.

또한, 두번째 방법을 사용하는 경우 invalid pointer가 page_fault()를 발생기키고, memory 접근하는데, 이때 error code를 받을 수 있는 방법이 따로 없다. 따라서 아래의 두 함수를 사용한다.

- `static int get_user (const uint8_t *uaddr)`
  - user virtual memory의 `UADDR`에서 byte를 읽었을 때, `PHYS_BASE` 이하에 있어야 한다. 유효하면 byte 값을, segmentation fault가 발생하면, -1을 return한다.

```
static int get_user (const uint8_t *uaddr) {
    int result;
    asm ("movl $1f, %0; movzbl %1, %0; 1:"
         : "=&a" (result) : "m" (*uaddr));
    return result;
}
```

- `static bool put_user (uint8_t *udst, uint8_t byte)`
  - user virtual memory의 `UDST`에 `BYTE`를 기록하며, 반드시 `PHYS_BASE` 이하에 있어야 한다. 성공 시 `true`, segmentation fault 발생 시 `false`를 return한다.

```
static bool put_user (uint8_t *udst, uint8_t byte) {
    int error_code;
    asm ("movl $1f, %0; movb %b2, %1; 1:"
         : "=&a" (error_code), "=m" (*udst) : "q" (by
te));
    return error_code != -1;
}
```

- 위의 함수들은 user pointer가 `PHYS_BASE` 이하임을 이미 확인한 후 사용해야 한다. 또한 `page_fault()`가 수정되어 커널의 페이지 폴트가 `eax`를 `0xffffffff`로 설정하고 이전 값을 `eip`에 복사하도록 해야 한다.

Internal Interrupt Handling

- Internal interrupt는 CPU instruction에 의해 직접적으로 일어나는 interrupt로 system call, page fault, divide by zero 등의 이유들로 instruction 실행 중에 발생한다.
- 만약, interrupt가 일어나면 CPU는 현재의 state를 stack에 저장한 뒤, interrupt handler를 호출한다. `intr_handler`는 발생한 interrupt의 종류에 따라 각각의 interrupt를 handling하는 함수를 호출한다.
  - 이 과정은 아래의 `list_entry` 함수와 `intr_handler` 함수에 대해서 설명할 때 더 자세하게 다룬다.
- handler가 return 될 때, 이전에 저장한 state들을 restore하고, CPU로 이동한다. (internal interrupt handler는 intr_frame에 전달된 정보를 보거나 수정할 수 있다. 내용이 바뀌면 프로세스 상태가 바뀌어서 예를 들어 시스템 호출의 return값을 EAX 레지스터에 저장하여 user program으로 전달할 수 있다.
- 일반적으로 **내부 인터럽트 핸들러는 인터럽트를 활성화한 상태**로 실행되며, 다른 커널 스레드가 이를 **선점할 수 있는 상태**입니다. 따라서 **공유 데이터**나 자원에 접근할 때는 동기화가 필요합니다.
- Internal interrupt handler는 nested하게 호출될 수 있다. 즉, systerm call handler가 page fault를 일으키는 것이 가능하다. 그러나, deep recursion은 제한된 kernel stack을 overflow할 수 있다는 위험이 존재한다.

System call Details

- OS는 external interrupt뿐만 아니라 software exception도 처리한다. software exception은 프로그램 코드에서 생기고 page fault, divide by zero 등이 있다. 예외는 시스템 호출을 통해 사용자 프로그램이 OS에 서비스 요청을 전달하는 수단으로도 사용됩니다.
- 80×86 architecture에서는 `int` 명령어가 시스템 호출을 수행하는 데 가장 일반적으로 사용된다. Pintos에서는 사용자 프로그램이 `int $0x30` 명령어를 호출하여 시스템 호출을 수행해주고 호출 전 system call number와 추가 argument들을 stack에 push해야 합니다. 그 이후 pintos 문서에 따라서 처리되는 것 같다.
- `syscall_handler`는 인터럽트를 처리하는 동안 제어를 획득하고, 호출자의 스택 포인터(ESP)를 통해 system call number와 argument를 확인한다. 이때, system call number는 caller stack pointer가 가리키는 32bit word에 있고 첫 번째 argument는 그 다음 32bit word에 있고, 이후 argument들은 순차적으로 저장된다. `syscall_handler`는 `struct intr_frame`의 `esp` 멤버를 통해 stack pointer를 참조하여 argument들에 접근한다. (`struct intr_frame`은 kernel stack에 있다)

- return value를 처리할 때는 80x86 에 따라 EAX 레지스터에 값을 저장한다. value를 return하는 system call은 `struct intr_frame`의 `eax` 멤버를 수정하여 return value를 설정한다. handler가 종료된 후 사용자 프로그램이 EAX 레지스터에서 return value를 확인할 수 있게 해준다.
- 중복 코드 최소화: 모든 시스템 호출 인수는 스택에 4바이트씩 저장되므로, 반복적인 코드 없이 stack에서 argument들을 쉽게 가져오는 방식을 활용 가능하다.

syscall_handler() calling process

System call의 작동을 이해하기 위해 `userprog/syscall.h`와 `userprog/syscall.c`파일의 함수들을 살펴볼 것이다.

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

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

- `syscall_init` 은 system call procedure 를 초기화할 때 호출되는 함수이다. USERPROG 가 포함되면 init.c의 main 함수에서 호출되어 system call을 수행할 있도록 한다.

```
void intr_register_ext (uint8_t vec_no, intr_handler_func *
handler,
                        const char *name) {
  ASSERT (vec_no >= 0x20 && vec_no <= 0x2f);
  register_handler (vec_no, 0, INTR_OFF, handler, name);
}
```

- external interrupt가 발생하면 이를 `vec_no`에 등록한다. vec_no는 내부의 interrupt number를 말하는데, 디버싱을 위해서 name이라는 이름의 handler를 호출한다.

```
static void register_handler (uint8_t vec_no, int dpl, enum
intr_level level, intr_handler_func *handler, const char *n
ame) {
```

```
  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;
}
```

- register_handler는 pintos의 interrupt vector table (IDT)에 interrupt handler를 등록하는 역할을 수행하는 함수이다. 코드의 구현을 보면 아래와 같다.
  - `ASSERT(intr_handlers[vec_no] == NULL);`을 통해 지정된 `vec_no`에 이미 핸들러가 등록되어 있지 않은지 확인한다.
  - 이후에는 level의 값에 따라서 `make_trap_gate`를 호출하거나 `make_intr_gate` 를 호출한다.
  - Intr_on일 경우, `make_trap_gate`를 이용해 trap gate를 설치하고, 그렇지 않다면 `make_intr_gate`를 이용해 interrupt gate를 생성한다. 두 gate의 차이는 CPU가 수행할 동작을 결정할 때 드러나느데, trap gate의 경우, interrupt 발생 후 if flag를 변경하지 않지만, interrupt gate는 Interrupt를 비활성화 한다. 그리고 두 경우 모두 생성된 gate는 IDT의 `vec_no`에 저장되도록 한다.

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

- 현재의 구현은, 단순히 “system call!”을 출력하고 `thread_exit`을 호출하고 있다.

이제 pintos에서 사용중인 system call 함수들과 그 구현을 살펴볼 것이다. system call 함수 목록은 아래와 같다.

```
void halt (void) NO_RETURN;
void exit (int status) NO_RETURN;
pid_t exec (const char *file);
int wait (pid_t);
bool create (const char *file, unsigned initial_size);
```

```
bool remove (const char *file);
int open (const char *file);
int filesize (int fd);
int read (int fd, void *buffer, unsigned length);
int write (int fd, const void *buffer, unsigned length);
void seek (int fd, unsigned position);
unsigned tell (int fd);
void close (int fd);
```

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

```
#define syscall0(NUMBER)
\
        ({
\
          int retval;
\
          asm volatile
\
            ("pushl %[number]; int $0x30; addl $4, %%esp"
\
               : "=a" (retval)
\
               : [number] "i" (NUMBER)
\
               : "memory");
\
          retval;
\
        })

#define syscall1(NUMBER, ARG0)
```

```
\
        ({
\
          int retval;
\
          asm volatile
\
            ("pushl %[arg0]; pushl %[number]; int $0x30; ad
dl $8, %%esp" \
               : "=a" (retval)
\
               : [number] "i" (NUMBER),
\
                 [arg0] "g" (ARG0)
\
               : "memory");
\
          retval;
\
        })

#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, an
d
   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, syscall1, syscall2, syscall3이 정의되어있는 것을 확인할 수 있다. 그리고 argument로 받는 것에 따라 구분된다. 예를 들어 argument로 number 받는 경우, syscall0가 사용된다. pintos의 syscall 함수들은 위의 정의를 이용하여 작성되었다.

```
.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 register
s. */
    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
```

- `intr_entry`는 pintos에서 interrupt가 발생했을 때 가장 먼저 호출되는 함수고, main interrupt의 진입점 역할을 한다. intterupt가 발생하면, CPU의 현재 state를 저장하고 register와 kernel environment를 설정하고 `interrupt_handler`를 호출한다.

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

```
  /* External interrupts are special.
     We only handle one at a time (so interrupts must be of
f)
     and they need to be acknowledged on the PIC (see belo
w).
     An external interrupt handler cannot sleep. */
  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 trigge
r
         spuriously due to a hardware fault or hardware rac
e
         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_handler`는 pintos의 모든 intterupt, ,fault exception을 처리하는 함수이다. intr_entry에 의해서 호출되고, interrupt가 발생한 상황에 대한 세부 정보를 `struct intr_frame`으로 전달 받아 interrupt를 처리한다.
- external이 발생한 경우, interrupt 처리 중, 다른 interrupt가 호출되지 않도록 assert를 사용해서 예외처리를 해준다. 그리고 `in_external_intr`를 true로 바꿔 external 처리 중임을 나타내고 `yield_on_return`를 false로 바꿔 interrupt 처리 직후 scheduling을 중단하도록 한다.
- IDT에 저장되어있는 intr_handler에 저장되어있는 handler 함수를 가져온다. 만약, handler가 존재하지 않을 경우, HW의 race로 인해 spurious interrupt가 발생한 것이므로 무시해준다.
- interupt를 처리 완료한 후에도 assert를 이용해서 예외처리를 해주고, `in_external_intr` 를 false로 바꾸고, `pic_end_of_interrupt`에 interrupt 처리가 완료된다. 그리고 `yield_on_return`이 true라면, `thread_yield`를 호출해서 다른 thread로 전환되도록 한다.

```
.globl intr_exit
.func intr_exit
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_exit` 함수는 pintos에서 interrupy를 처리한 뒤, 원래 상태로 복귀할 수 있게 한다. 따라서, interrupt가 발생했을 때 stack에 저장된 레지스터와 CPU state를 restore하고, 원래 프로그램의 실행 흐름으로 돌아갑니다. 또한 새로운 user_process를 시작할 때도 `intr_exit`이 호출된다.

Analysis on File System

Using file system

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

- file structure를 살펴보면, `inode` (index node)는 간단한 file의 정보를 담는 structure이다. 그리고 `pos`는 현재의 position을 나타내고, `deny_write`는 `file_deny_wite` 함수가 호출되었는지에 대한 정보를 나타낸다.

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

```
*/
  };
```

- inode structure를 살펴보면, inode list를 관리해보기 위한 `elem`이 있다. 그리고 `sector`는 inode가 저장되어있는 disk의 sector number를 저장한다. `open_cnt`는 inode가 열려있을 때 할당을 해제하지 않기 위해서 사용하는 변수로, inode가 얼마나 open 되어있는지를 알려주는 변수이다. 그리고 `removed`는 inode를 remove 해도 되는지에 대한 여부를 파악할때 사용한다. 그리고 `deny_write_cnt`는 inode를 수정해도 되는지 여부를 파악할 때 사용한다. `removed`는 삭제되면 true를 저장하고, `deny_write_cnt`가 0이면 수정 가능하다는 것을 의미한다. 마지막으로 data는 inode_disk의 structure 형태로 저장되어있다.

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

- inode_disk는 inode의 data를 저장하는 사용되는 structure이다. `start` 는 첫번째 data sector이고, `length`는 file size 를 저장하고 있다. 그리고 `magic`은 magic number 를 저장하고 있다. 그리고 한 sector는 512 byte이기 때문에 unused[125]를 통해 크기를 맞춰주고 있다.

struct file* filesys_open(const char *name)

```
struct file * filesys_open (const char *name) {
  struct dir *dir = dir_open_root ();
  struct inode *inode = NULL;

  if (dir != NULL)
    dir_lookup (dir, name, &inode);
  dir_close (dir);
```

```
  return file_open (inode);
}
```

- `filesys_open`는 file을 open 하는데 사용하는 함수이다. 코드를 살펴보면, dit 변수에 root directory를 open하고 `dir_lookup` 함수로 name과 같은 file이 존재하는지 확인한다. 그리고 다른 문제가 없다면, `file_open` 함수를 호출햇 file을 open한다.

```
bool dir_lookup (const struct dir *dir, const char *name, s
truct inode **inode) {
  struct dir_entry e;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);

  if (lookup (dir, name, &e, NULL))
    *inode = inode_open (e.inode_sector);
  else
    *inode = NULL;

  return *inode != NULL;
}
```

- `dir_lookup` 함수를 보면 `look_up` 함수를 이용해 directory에 name과 같은 file이 있는지 확인하는 것을 볼 수 있다. 그리고 같은 name의 file이 있다면, `inode_open`을 호출한다.

```
struct inode * inode_open (block_sector_t sector) {
  struct list_elem *e;
  struct inode *inode;

  /* Check whether this inode is already open. */
  for (e = list_begin (&open_inodes); e != list_end (&open_
inodes);
       e = list_next (e))
    {
      inode = list_entry (e, struct inode, elem);
      if (inode->sector == sector)
        {
```

```
          inode_reopen (inode);
          return inode;
        }
    }

  /* Allocate memory. */
  inode = malloc (sizeof *inode);
  if (inode == NULL)
    return NULL;

  /* Initialize. */
  list_push_front (&open_inodes, &inode->elem);
  inode->sector = sector;
  inode->open_cnt = 1;
  inode->deny_write_cnt = 0;
  inode->removed = false;
  block_read (fs_device, inode->sector, &inode->data);
  return inode;
}
```

```
struct inode * inode_reopen (struct inode *inode) {
  if (inode != NULL)
    inode->open_cnt++;
  return inode;
}
```

- `inode_open` 함수를 보면, 가장 먼저 이미 open 했던 file인지 아닌지 확인한다. 이전에 open 했었던 file이라면 이미 file의 inode가 존재하는 경우이기 때문에, `inode_reopen` 함수를 이용해 inode의 `open_cnt` 를 1 증가시켜준다. 그리고 처음 open하는 file이라면 inode를 allocate 받고, 초기화 해준다. 그리고 그 inode를 return한다.

```
struct file * file_open (struct inode *inode) {
  struct file *file = calloc (1, sizeof *file);
  if (inode != NULL && file != NULL)
    {
      file->inode = inode;
      file->pos = 0;
      file->deny_write = false;
```

```
      return file;
    }
  else
    {
      inode_close (inode);
      free (file);
      return NULL;
    }
}
```

- 최종적으로 `file_open` 함수를 이용해서 file을 open 해주게 되는데, `file_open`은 `dir_lookup`에서 할당 받은 inode를 argument로 받는다. 그리고 file의 inode를 argument로 받은 inode로 설정해주고, 다른 변수들을 초기화해준다. 그리고 inode가 없거나 file이 없는 경우, inode를 받고 file을 할당 해제해준다.

```
void file_close (struct file *file) {
  if (file != NULL)
    {
      file_allow_write (file);
      inode_close (file->inode);
      free (file);
    }
}
```

- 이제 file_close를 보면 file을 close할 때, file_allow_write, inode_close 함수가 호출되고, free 함수를 통해 할당을 해제해주는 것을 볼 수 있다.

```
void file_allow_write (struct file *file) {
  ASSERT (file != NULL);
  if (file->deny_write)
    {
      file->deny_write = false;
      inode_allow_write (file->inode);
    }
}
```

- `file_allow_write` 함수는 file의 inode에 대한 수정을 다시 허용한다.

```
void inode_close (struct inode *inode) {
  /* Ignore null pointer. */
  if (inode == NULL)
    return;

  /* Release resources if this was the last opener. */
  if (--inode->open_cnt == 0)
    {
      /* Remove from inode list and release lock. */
      list_remove (&inode->elem);

      /* Deallocate blocks if removed. */
      if (inode->removed)
        {
          free_map_release (inode->sector, 1);
          free_map_release (inode->data.start,
                            bytes_to_sectors (inode->data.l
ength));
        }

      free (inode);
    }
}
```

- `inode_close`는 open_cnt를 1 줄이고, open_cnt가 0인 경우, 해당 Inode는 open된 inode가 아닌 것이기 때문에 indoe도 free 해주게 된다.

bool filesys_create(const char *name, off_t initial_size)

```
bool filesys_create (const char *name, off_t initial_size)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = dir_open_root ();
  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
                  && inode_create (inode_sector, initial_si
ze)
                  && dir_add (dir, name, inode_sector));
```

```
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);

  return success;
}
```

- `filesys_create`는 file을 create하는데 사용되는 함수이다. 그러나 코드를 보면 `filed_open` 함수는 실행되지 않는 것을 볼 수 있는데, 이를 통해 `file_create`와 `file_open` 이 동시에 실행되지 않아 inode를 file에 연결해주는 과정이 별도로 필요하지 않다는 것을 볼 수 있다.

```
bool inode_create (block_sector_t sector, off_t length) {
  struct inode_disk *disk_inode = NULL;
  bool success = false;

  ASSERT (length >= 0);

  /* If this assertion fails, the inode structure is not ex
actly
     one sector in size, and you should fix that. */
  ASSERT (sizeof *disk_inode == BLOCK_SECTOR_SIZE);

  disk_inode = calloc (1, sizeof *disk_inode);
  if (disk_inode != NULL)
    {
      size_t sectors = bytes_to_sectors (length);
      disk_inode->length = length;
      disk_inode->magic = INODE_MAGIC;
      if (free_map_allocate (sectors, &disk_inode->start))
        {
          block_write (fs_device, sector, disk_inode);
          if (sectors > 0)
            {
              static char zeros[BLOCK_SECTOR_SIZE];
              size_t i;

              for (i = 0; i < sectors; i++)
```

```
                block_write (fs_device, disk_inode->start +
i, zeros);
            }
          success = true;
        }
      free (disk_inode);
    }
  return success;
}
```

- `inode_create` 함수를 보면 특정 sector에 size 만큼의 공간을 할당한다. 따라서, `filesys_create`는 `inode_create`를 이용해 disk의 특정 sector에 해당 size만큼의 공간을 할당 받게된다. 그리고dir_add() 함수를 통해 특정 name을 가진 파일을 directory에 포함시킨다.

bool filesys_remove (const char *name)

```
bool filesys_remove (const char *name) {
  struct dir *dir = dir_open_root ();
  bool success = dir != NULL && dir_remove (dir, name);
  dir_close (dir);

  return success;
}
```

filesys_remove() 함수는 file_close와는 다르게 단순히 dir에서 지우기만 . 즉 remove 했다고 해서 아직 열려있는 파일을 닫는 것은 아니다.

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

```
off_t file_read (struct file *file, void *buffer, off_t siz
e) {
  off_t bytes_read = inode_read_at (file->inode, buffer, si
ze, file->pos);
  file->pos += bytes_read;
```

```
  return bytes_read;
}
```

- `file_read`는 실제로 읽어온 `bytes_read`를 return한다. `inode_read_at`을 이용해 argument로 받은 size bytes를 buffer로 읽어오고, 그 값을 `bytes_read`에 저장한다. 그리고 `file→pos`를 읽은 만큼 업데이트 해준다.

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

```
off_t file_write (struct file *file, const void *buffer, of
f_t size) {
  off_t bytes_written = inode_write_at (file->inode, buffe
r, size, file->pos);
  file->pos += bytes_written;
  return bytes_written;
}
```

- `file_read`와 거의 유사하지만, `inode_read_at` 대신 `inode_write_at`를 사용하여 file의 pos 부터 size 만큼의 data를 buffer에서 inode의 data로 옮긴다.

void file_deny_write (struct file *file)

```
void file_deny_write (struct file *file) {
  ASSERT (file != NULL);
  if (!file->deny_write)
    {
      file->deny_write = true;
      inode_deny_write (file->inode);
    }
}
```

- file의 inode에 대한 wrtie 작업이 허용될 때까지 write를 막아둔다. 이는 file이 running 되는 와중에 수정 되는것을 방지하기 위함이다. 그리고 file이 close 될 때, `file_allow_write`를 사용해서 다시 write가 가능하도록 한다. 코드를 보면, `file→deny_write`를 true로 바꿔주고 `inode_deny_write`를 호출해서 inode도 수정할 수 없게 한다.

off_t file_length (struct file *file)

```
off_t file_length (struct file *file) {
  ASSERT (file != NULL);
  return inode_length (file->inode);
}
```

- `file_length`는 file 의 size 를 bytes로 반환한다. file이 처음 create 될 때, size가 정해져 있으므로 `inode_length`를 호출해서 inode 내부에 저장되어있는 length를 반환하면 된다.

void file_seek (struct file *file, off_t new_pos)

```
void file_seek (struct file *file, off_t new_pos) {
  ASSERT (file != NULL);
  ASSERT (new_pos >= 0);
  file->pos = new_pos;
}
```

- file→pos 값을 new_pos 로 변경해준다

off_t file_tell (struct file *file)

```
off_t file_tell (struct file *file) {
  ASSERT (file != NULL);
  return file->pos;
}
```

- file→pos 값을 byte offset 으로 return한다
- file_seek과 file_tell은 inode를 이용하지 않고 file structure의 member만을 사용한다.

Design plan

Process Termination Messages

- user process가 exit을 호출하거나 terminate 될 때, process의 이름과 exit code를 출력해줘야한다. 그리고 이 경우 `thread_exit`을 호출하기 때문에 이 함수 안에서 message를 출력하도록 한다.

- kernel thread가 user process가 아니거나 halt system call이 발생하는 경우에는 message를 출력하지 않아도 된다. 또한, process가 load를 실패했을 때의 message 출력은 선택 사항이다.
- 출력 형식을 코드를 보면 proecess name, exit code가 필요한데 process name은 thread_create 함수의 argument로 file name을 넣어주고 thread_create함수에서 init_thread를 호출해준다. 그 다음에 stricpy 함수가 실행된다. 현재 thread를 가져와서 해당 thread의 name에 접근하면 된다. 다시 정리하면 file name을 thread 구조체에 저장해두고 thread_exit에서 출력하면 될 것 같다.
- 이를 바탕으로 exit 함수를 구성한다. 함수는 현재 thread를 이용해서 thread name을 바탕으로 status 변수(thread 구조체에 추가로 선언한 변수)에 인자로 받아온 status를 저장하고 pintos 문서를 바탕으로 process name과 status를 출력해주고 thread_exit을 호출해주는 식으로 함수를 구현해주면 될 것 같다. 이렇게 되면 process가 exit, terminate 되었을 때 exit함수가 호출 된다.

Argument Passing

- process_execute의 현재 implementation은 새 process에 argument를 전달하지 않는다. 이 부분을 개선해주려면 file name를 비롯해서 공백으로 단어를 나누어서 handling해줘야 한다.
- 자세히 설명하면 process_Execute함수는 parameter로 받은 file_name을 thread_create 함수에게 전달해주고 start_process가 호출되면서 strcpy를 통하여 copy본을 load 함수에 넘겨주게 된다. 이후 load에서 user memory stack을 관리한다.
- 공백으로 단어를 나눌 때 첫번째는 name, 두번째는 첫번째 argument 등등 아래의 pintos 문서 그림을 참고하여 구현을 해주면 된다. process_execute에서 file name이 parsing되고 thread create, start_process, load를 거쳐 parsing된 값이 esp reg에 저장된다.

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 (*) ()

- 위에서의 과정을 처음부터 해보면 가장 먼저 process_Execute 함수가 받아온 file name을 추출해야 한다. 이때 이미 string library에 선언되어있는 strtok_r 함수를 이용하면 될 것 같다. 이 함수의 parameter들을 조절해서 file name만 따로 분해해줄 수 있을 것 같다. 이후 parsing 된 file name을 thread_create에 전달해주면 된다.
- thread_create이 file name이랑 fn_copy를 넘겨받고 start_process를 실행할 차례이다. 이때 인자로 file_name을 받아오게 되고 load에 넣어줄 때에도 strtok_r을 이용해서 ㅏ장 앞 단어만 넣어줘야 한다. 따라서 위에서 했던 것과 비슷하게 strtok_r함수를 이용해서 parsing 해주면 될 것 같다. 이때는 parsing을 for문을 이용해서 구현해주면 된다.
- load가 끝난 뒤 stack에 parsing한 내용을 save해야 한다. argv값들을 넣어주어야 하는데 이때 stack의 구조상 역으로 넣어줘야 한다. argv의 마지막은 NULL로 해줘야 한다. 앞서 어셈블리어 코드를 봤듯이 argv 값만큼 esp에서 빼주고 값을 넣어주는 것을 반복하면 된다.
- 4byte aligning을 해주면 된다. 4의 배수로 esp값을 맞춰서 stack을 관리해준다.
- 이후 argv를 모두 넣어주면 argv, argc, return address를 넣어서 완성해준다.

System Calls

1. User memory access

- system call이 발생해서 kerenel이 user progam을 수정해야하는 경우, pointer의 validity를 확인해야한다. validity를 확인하는데는 accessing user memory 부분에 자세하게 정리 되어있다. 두 방법 중 PHYS_BASE 아래에 위치하는지 확인하는 것이 더 효율적인 방법이다.

```
static inline bool is_kernel_vaddr (const void *vaddr) {
  return vaddr >= PHYS_BASE;
}
```

  - `is_kernel_vaddr`은 argument로 받은 vaddrdl PHYS_BASE 보다 큰지 작은지 비교하는 함수이다. 따라서, `is_kernel_vaddr` 가 호출되고 접근하려하는 주소가 kernel virtual address인지 확인한다.

```
static uint32_t * lookup_page (uint32_t *pd, const void
*vaddr, bool create) {
  uint32_t *pt, *pde;

  ASSERT (pd != NULL);

  /* Shouldn't create new kernel virtual mappings. */
  ASSERT (!create || is_user_vaddr (vaddr));

  /* Check for a page table for VADDR.
     If one is missing, create one if requested. */
  pde = pd + pd_no (vaddr);
  if (*pde == 0)
    {
      if (create)
        {
          pt = palloc_get_page (PAL_ZERO);
          if (pt == NULL)
            return NULL;

          *pde = pde_create (pt);
        }
      else
        return NULL;
    }
```

```
  /* Return the page table entry. */
  pt = pde_get_pt (*pde);
  return &pt[pt_no (vaddr)];
}
```

- 이후에 page table entry를 확인해야하는데, 이는 lookup_page 함수를 호출해서 확인할 수 있다. page table entry가 NULL이 아니고, page가 할당되어있고, user process가 page에 접근 가능하다면 true를 return하고 아니라면 false를 return하도록 한다. 그러면 최종적으로 user memory에 access 가능한지 여부를 판단할 수 있게 된다.

2. System Call Functions

- system call functions들의 모든 argument로 args가 전달된다. args는 정확히 말하면 user memory에서 system call number 다음으로 첫번째 argument를 가리키는 address이다. args를 통해서 각 function들은 자신이 원하는 argument에 맞춰서 user memory에서 읽어와 준비하는 과정이 필요하다. 추가로 각 functino별로 가져온 argument들을 비롯한 정보들의 validation을 하는 과정이 있다. 각 자료에 대한 유효성을 확인해야 한다.
- Halt()
  - shut_down_power_off를 호출하면 된다.

- Exec()
  - process_execute함수를 호출해주고 연쇄적으로 thread_create가 호출되어 새로운 child thread가 생성된다. parent thread와 child thread간의 관계를 담고 있는 list를 thread 구조체에 추가해주어서 관리하면 될 것 같다. (donation list 같은 느낌) 초기화 작업을 해주고 child thread를 parent thread의 child list에 넣어주고 unblock 해준다. 추가로 해당 list를 자세히 생각해보면 parent가 child 생성했을 때 추가되고, parent가 child를 wait하고 return할 때 삭제될 것이다.
  - 이후 child가 context switch를 통하여 처음 execute할 때 start_process → load함수로 file load가 된다. 이때 parent child 사이의 synchronization을 잘 정의해서 관리해줘야 한다. (parent 입장에서는 child가 완전히 load될 때까지 process_execute return되면 안된다. 따라서 schedule constraint semaphore를 이용해서 해당 semaphore를 0으로 초기화 해주어서

sema_down, sema_up을 이용해서 synchronization을 구현하자(child thread가 start_process → load return 되면 sema_up)

- wait()
  - process_wait함수를 호출해준다. pintos 문서에 따르면 child_tid를 함께 argument로 갖게된다. parent의 child list를 접근하여 tid와 일치하는 것이 있는지 체크하고 만약 없으면 -1 return
  - parent가 child를 wait할 때 child가 살아 있는 상태이면 parente는 child terminate 등 끝날때까지 wait해야 한다. 근데 이미 child가 죽은 상태이면 wait할 필요가 없어진다.
- Exit()
  - 현재 user program을 종료해주고 status를 kernel에 반환한다. process의 parent가 기다리고 있으면 이 부분이 return 되야하는 status가 되고 status가 0 이면 성공 아니면 오류를 나타낸다.
  - process_exit함수를 호출한다. process의 child list에 있는 structure들을 free 해준다.
  - 현재 process의 parent가 살아있다면 parent의 child list에 접근하여 변수의 값들을 상황에 맞추어서 바꿔줘야 한다. parent가 child를 wait할 때 child가 죽은 상황에도 exit status를 통해 정보를 얻을 수 있다.
- create()
  - `sys_create`는 file을 create하는 함수이다. file create은 file을 open하는 작업이 아니므로 따로 fd를
    setting 해주지 않는다.
  - `args_parse`를 통해서 받아온 file 과 `initial_size`를 `filesys_create`에 argument로 넘겨주면 file이라는 이름의 `initial_size`를 크기로 갖는 파일을 생성할 수 있다.
  - 이후에 `args_parse`로 받은 file의 주소가 유효한지 확인할 때는 `access_user_mem`를 이용하면 된다.
- remove()
  - `sys_remove`는 file이라는 이름을 가진 file을 삭제한다. file을 close하는 것이 아니므로 fd를 바꾸지 않는다.
  - `args_parse`를 이용해서 `sys_remove`에 필요한 file을 받아 `filesys_remove`에 argument로 넘겨주게 된다.

    - 이후에 `args_parse`로 받은 file의 주소가 유효한지 확인할 때는 `access_user_mem`를 이용하면 된다.
- open()
    - `file_open`은 file을 open하면서 이에 해당하는 fd를 return 해주는 함수이다. 그리고 open 함수를 call한 함수의 `l_fd`를 수정해줘야한다. 그리고, process가 file을 처음 open 하는 경우에는 `l_fd`를 새롭게 할당해줘야한다.
    - 코드를 보면, `args_parse`로 file의 이름을 받아 `filesys_open`를 통해 file을 open한다.
    - 이후에 `args_parse`로 받은 file의 주소가 유효한지 확인할 때는 `access_user_mem`를 이용하면 된다.
    - 그리고 open하기 전에 lock을 걸어 open 과정을 atomic하게 설정해준다.
- filesize()
    - `sys_filesize`는 fd 값을 필요로 한다. fd 값은 user memory로 부터 읽어오는데, 이를 이용해 file descriptor가 fd인 file 의 크기를 byte 단위로 return 한다.
    - 만약, file이 없다면 -1을 return 한다.
- read()
    - `sys_read`에 필요한 file, buffer의 주소, size를 args_parse를 사용해서 받아온다.
    - 이후에 `args_parse`로 받은 file의 주소가 유효한지 확인할 때는 `access_user_mem`를 이용하면 된다.
    - buffer에 size bytes 만큼 file descriptor가 fd인 file로부터 내용을 읽어온다.
    - 그리고 읽은 byte를 return 하는데, file을 끝까지 다 읽었다면 0을 return하고, EOF를 제외한 다른 이유로
      file을 읽지 못한 경우, 1을 return 한다. 그리고 fd가 0인 경우에는
      `input_getc`를 사용해 console에서 값을 읽어온다.
- write()
    - `sys_write`는 buffer 로부터 size bytes 만큼 읽어와서 file descriptor가 fd인 file에 write 해주는 함수이다. 그리고 작성한 byte를 return한다.
    - EOF까지 파일을 작성한 뒤, 일반적으로는 file이 확장되어야 하지만, file growth가 구현되지 않았으므로 EOF 까지 가능한 한 많이 write 한 뒤 write한 byte만큼 return한다.
- seek()

    - `sys_seek`는 fd를 argument로 받아 가리키는 file의 position을 바꾸주는 함수로 file_seek를 사용해 구현한다.
    - 또한, open한 file을 접근하는 작업이므로 file_seek 앞뒤로 lock을 걸어 atomic하게 수행해줘야한다.
    - 만약 open한 file이 null인 경우 -1을 return한다.
- tell()
    - `sys_tell`은 fd가 가리키는 파일이 어디를 읽을 준비를 하고 있는지를 알려주는 함수이다. 즉, 다음으로 읽거나 쓸 byte의 position을 return한다.
    - `file_tell`로 쉽게 구현할 수 있는데, 이 과정도 file을 open해서 다루기 때문에 lock을 이용해 atomic하게 구현해준다.
- close()
    - file을 close할 때는 해당 file을 open한 thread의 `l_fd` 에서 해당 fd를 빼주어야 한다.
    - 그리고 `file_from_l_fd`의 argument로 fd와 true를 넣어준다. 만약, flie이 null이라면 -1을 return 하고 아니라면, file_close를 사용하여 파일을 닫는다.
    - 이 과정 역시, lock을 사용해서 atomic하게 진행한다

Denying writes to executables

- executable file들이 running 상태일 때 write되는 것을 막아줘야 한다. file_deny_write함수를 이용하면 된다. file 구조체는 deny_write라는 boolean 변수를 private하게 가지고 있다. 이것을 이용해서 만약 file이 open되면 file이 executable일 것이고 이때 deny_write값을 바꿔주면 된다. 그리고 deny_write값을 바꿔주고 current thread의 이름과 file name이 같으면 file_deny_write함수를 수행하면 된다. 이렇게 되면 file이 close되기 전까지 write를 할 수 없다. file_close에 file_allow_write이 있어서 따로 해줄 필요는 없다.