Assn2 Design Report

Pintos Assn2 Design Report

Team 37
20200229 김경민, 20200423 성치호

Pintos 구현 분석

TSS

TSS(Task-State Segment)란 80x86 architectural task switching에서 사용되는 segment로, 프로세서에 내장된 멀티태스킹 지원인 'Task'를 정의하는 구조체이다. 핀토스에서는 User mode에서 interrupt가 발생하여 stack을 전환할 때 TSS를 사용한다.

user mode에서 interrupt 발생시, 프로세서는 현재 TSS의 `ss0`과 `esp0`을 참조하여 어떤 stack에서 interrupt를 핸들링할지 결정한다. 이를 위해 TSS를 생성하고 `ss0`, `esp0`을 초기화해야한다.

아래는 이번 프로젝트에서 변경하지 않아도 되지만 가지고 와서 활용할 TSS에 관련된 함수들이다.

`struct tss`

```
struct tss
  {
    uint16_t back_link, :16;
    void *esp0;                         /* Ring 0 stack virtual address. */
    uint16_t ss0, :16;                  /* Ring 0 stack segment selector. */
    void *esp1;
    uint16_t ss1, :16;
    void *esp2;
    uint16_t ss2, :16;
    uint32_t cr3;
    void (*eip) (void);
    uint32_t eflags;
    uint32_t eax, ecx, edx, ebx;
    uint32_t esp, ebp, esi, edi;
    uint16_t es, :16;
    uint16_t cs, :16;
    uint16_t ss, :16;
    uint16_t ds, :16;
    uint16_t fs, :16;
    uint16_t gs, :16;
    uint16_t ldt, :16;
    uint16_t trace, bitmap;
  };
```

TSS(Task-State Segment)를 표현하는 struct이다. 주로 `ss0`, `esp0`만 사용된다.

- `ss0`은 Ring 0 stack의 virtual address이다. 즉 kernel stack의 virtual address이다.
- `esp0`은 Ring 0 stack의 segment selector이다. 즉 kernel stack의 segment selector이다.

`static tss`

```
static struct tss *tss;
```

`userprog/tss.c`의 `tss`는 kernel tss로 static variable이다.

`tss_init (void)`

```
void
tss_init (void)
{
  tss = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  tss->ss0 = SEL_KDSEG;
  tss->bitmap = 0xdfff;
  tss_update ();
}
```

> 전역의 kernel tss인 `tss`를 초기화하는 함수

TSS는 OS 내에서 상대적으로 한정적인 곳에서만 사용할 것이기에 `tss`의 일부 필드만 초기화해준다.
`palloc_get_page`를 통해 `tss`에 0으로 채워진 페이지를 할당한다. 그리고 `tss`의 `ss0`을 `SEL_KDSEG`로 초기화하고 `bitmap`도 `0xdfff`로 초기화해준다.
이후 `tss_update`를 통해 현재 스레드의 stack 끝으로 `esp0`를 초기화한다.

`tss_get(void)`

```
struct tss *
tss_get (void)
{
  ASSERT (tss != NULL);
  return tss;
}
```

> kernel tss를 반환하는 함수

`tss`의 초기화가 이루어졌는지 확인하고 `tss`를 반환한다.

`tss_update(void)`

```
void
tss_update (void)
{
  ASSERT (tss != NULL);
  tss->esp0 = (uint8_t *) thread_current () + PGSIZE;
}
```

kernel tss `tss`의 stack pointer `esp0`가 현재 스레드 stack의 끝을 가리키게 변경하는 함수이다.

Global Descriptor Table (GDT)

80x86은 segmented architecture로 GDT가 존재한다. Global Descriptor Table (GDT)는 권한에 따라 모든 프로세스에서 사용할 가능성이 있는 segment들을 정의하는 테이블이다. GDT의 각 entry는 세그먼트를 나타낸다. 중요히 생각해야할 segment는 code, data, TSS descriptor 세가지 유형의 세그먼트뿐이다.

아래는 이런 GDT를 셋업하기 위한 변수 및 함수로 이번 프로젝트에서는 변경할 필요는 없으며 호출해 사용할 가능성은 있다.

```
#define SEL_NULL        0x00    /* Null selector. */
#define SEL_KCSEG       0x08    /* Kernel code selector. */
#define SEL_KDSEG       0x10    /* Kernel data selector. */
```

from `threads/loader.h`

```
#define SEL_UCSEG       0x1B    /* User code selector. */
#define SEL_UDSEG       0x23    /* User data selector. */
#define SEL_TSS         0x28    /* Task-state segment. */
#define SEL_CNT         6       /* Number of segments. */
```

GDT는 위의 6가지 유형의 segment만 관리한다.

```
static uint64_t gdt[SEL_CNT];
```

GDT를 나타내는 전역 변수 `gdt`로 6가지 segment 정보를 담을 것을 나타내고 있다.

`gdt_init(void)`

```
void
gdt_init (void)
{
  uint64_t gdtr_operand;

  /* Initialize GDT. */
  gdt[SEL_NULL / sizeof *gdt] = 0;
  gdt[SEL_KCSEG / sizeof *gdt] = make_code_desc (0);
  gdt[SEL_KDSEG / sizeof *gdt] = make_data_desc (0);
  gdt[SEL_UCSEG / sizeof *gdt] = make_code_desc (3);
  gdt[SEL_UDSEG / sizeof *gdt] = make_data_desc (3);
  gdt[SEL_TSS / sizeof *gdt] = make_tss_desc (tss_get ());
```

```
  /* Load GDTR, TR.  See [IA32-v3a] 2.4.1 "Global Descriptor
     Table Register (GDTR)", 2.4.4 "Task Register (TR)", and
     6.2.4 "Task Register".  */
  gdtr_operand = make_gdtr_operand (sizeof gdt - 1, gdt);
  asm volatile ("lgdt %0" : : "m" (gdtr_operand));
  asm volatile ("ltr %w0" : : "q" (SEL_TSS));
}
```

> userprog 관련 segment를 포함해 `gdt`를 초기화하는 함수

`make_*_desc` 함수들을 통해 생성한 segment descriptor를 `gdt` 내 적절한 위치에 넣는다.
이후 `make_gdtr_operand`를 통해 `lgdt`에 넘겨줄 operand를 만든다. 해당 operand는 `gdt`의 주소와 크기를 담고 있다. 위에서 생성한 operand를 이용해 `lgdt`를 호출해 `gdtr`에 새로 만든 `gdt` 정보를 저장하게 한다. 이로써 userprog 관련 segment descriptor들이 GDT에 추가된 것이다.

`enum seg_class`

```
enum seg_class
  {
    CLS_SYSTEM = 0,             /* System segment. */
    CLS_CODE_DATA = 1           /* Code or data segment. */
  };
```

system에 관한 segment인지, code나 data에 관련된 segment인지 구별하는 enum

`enum seg_granularity`

```
enum seg_granularity
  {
    GRAN_BYTE = 0,              /* Limit has 1-byte granularity. */
    GRAN_PAGE = 1               /* Limit has 4 kB granularity. */
  };
```

seg가 어떤 크기로 세분되는지 나타내는 enum으로 `GRAN_BYTE`(1byte), `GRAN_PAGE`(4kB)로 구분할 수 있다.

`make_seg_desc`

```
static uint64_t
make_seg_desc (uint32_t base,
               uint32_t limit,
               enum seg_class class,
               int type,
               int dpl,
               enum seg_granularity granularity)
{
  uint32_t e0, e1;

  ASSERT (limit <= 0xfffff);
  ASSERT (class == CLS_SYSTEM || class == CLS_CODE_DATA);
  ASSERT (type >= 0 && type <= 15);
  ASSERT (dpl >= 0 && dpl <= 3);
  ASSERT (granularity == GRAN_BYTE || granularity == GRAN_PAGE);

  e0 = ((limit & 0xffff)             /* Limit 15:0. */
        | (base << 16));             /* Base 15:0. */

  e1 = (((base >> 16) & 0xff)        /* Base 23:16. */
        | (type << 8)                /* Segment type. */
        | (class << 12)              /* 0=system, 1=code/data. */
        | (dpl << 13)                /* Descriptor privilege. */
        | (1 << 15)                  /* Present. */
        | (limit & 0xf0000)          /* Limit 16:19. */
        | (1 << 22)                  /* 32-bit segment. */
        | (granularity << 23)        /* Byte/page granularity. */
        | (base & 0xff000000));      /* Base 31:24. */

  return e0 | ((uint64_t) e1 << 32);
}
```

> 주어진 정보(base, 세그먼트 class, granularity, type, dpl)를 바탕으로 64비트 segment descriptor를 생성해 반환하는 함수

`make_code_desc`, `make_data_desc`, `make_tss_desc`에서 호출하여 각 세그먼트 descriptor를 생성할 때 사용한다.
입력한 `base`, `class`, `type`, `dpl`, `granularity`를 형식에 맞춰 64비트 내 적절한 위치에 배치하고 `limit`을 통해 `base` 등 값 노출 여부를 조절하여 segment descriptor를 제작해 반환한다.

`make_code_desc(int dpl)`

```
static uint64_t
make_code_desc (int dpl)
{
  return make_seg_desc (0, 0xfffff, CLS_CODE_DATA, 10, dpl, GRAN_PAGE);
}
```

> 주어진 `dpl`을 가지는 code segment descriptor를 생성해 반환하는 함수

`make_data_desc(int dpl)`

```
static uint64_t
make_data_desc (int dpl)
{
  return make_seg_desc (0, 0xfffff, CLS_CODE_DATA, 2, dpl, GRAN_PAGE);
}
```

> 주어진 `dpl`을 가지는 data segment descriptor를 생성해 반환하는 함수

`make_tss_desc(void *laddr)`

```
static uint64_t
make_tss_desc (void *laddr)
{
  return make_seg_desc ((uint32_t) laddr, 0x67, CLS_SYSTEM, 9, 0, GRAN_BYTE);
}
```

> 주어진 `dpl`을 가지는 tss segment descriptor를 생성해 반환하는 함수

`make_gdtr_operand(uint16_t limit, void *base)`

```
static uint64_t
make_gdtr_operand (uint16_t limit, void *base)
{
  return limit | ((uint64_t) (uint32_t) base << 16);
}
```

> `lgdt`를 수행할 때 넘길 operand를 생성하는 함수로 gdt의 주소 및 크기를 담고 있다.

ELF

ELF specification에 명시된 것을 동일하게 구현한 것이다.

```
typedef uint32_t Elf32_Word, Elf32_Addr, Elf32_Off;
typedef uint16_t Elf32_Half;
```

ELF 내에서 헤더를 표현할 때 사용되는 data type이다.

```
struct Elf32_Ehdr
  {
    unsigned char e_ident[16];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
```

```
  Elf32_Half    e_shstrndx;
};
```

ELF binary의 매우 앞 부분에 등장하는 정보에 대한 struct로 Exectable에 대한 정보를 담는 **Executable header**를 표현하는 struct이다.

```
struct Elf32_Phdr
{
  Elf32_Word p_type;
  Elf32_Off  p_offset;
  Elf32_Addr p_vaddr;
  Elf32_Addr p_paddr;
  Elf32_Word p_filesz;
  Elf32_Word p_memsz;
  Elf32_Word p_flags;
  Elf32_Word p_align;
};
```

ELF binary 내 정보에 대한 struct로 **Program header**를 표현하는 struct이다.
바이너리 내 `e_phoff`에서 시작하여 `e_phnum`개의 entries를 가지고 있다.

```
#define PT_NULL    0            /* Ignore. */
#define PT_LOAD    1            /* Loadable segment. */
#define PT_DYNAMIC 2            /* Dynamic linking info. */
#define PT_INTERP  3            /* Name of dynamic loader. */
#define PT_NOTE    4            /* Auxiliary info. */
#define PT_SHLIB   5            /* Reserved. */
#define PT_PHDR    6            /* Program header table. */
#define PT_STACK   0x6474e551   /* Stack segment. */
```

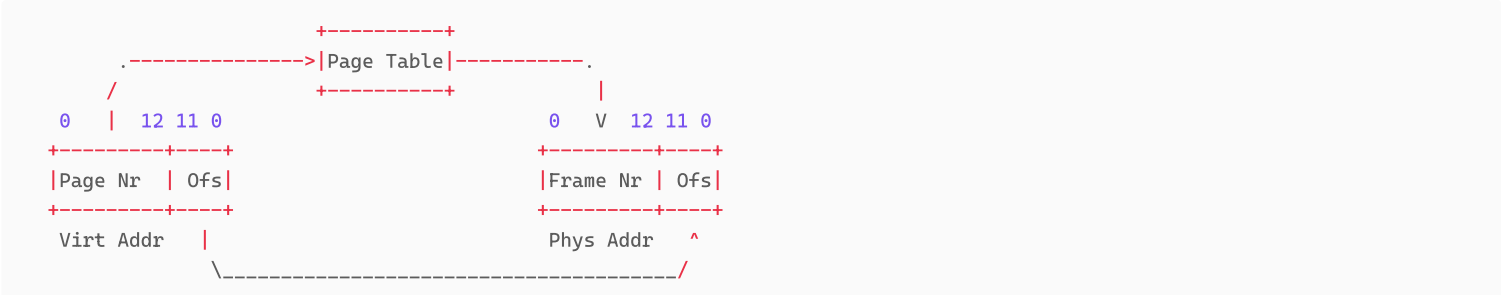
`p_type`은 세그먼트의 타입을 표현하는 값으로 위 값들은 `p_type`에 사용되는 값들의 목록이다.

```
#define PF_X 1          /* Executable. */
#define PF_W 2          /* Writable. */
#define PF_R 4          /* Readable. */
```

`p_flags`에 사용되는 값들의 목록으로 각 segment에 의존적인 값들이다.

Page Directory

Page table이란 CPU가 virtual address를 physical address로 변환할 때 사용하는 데이터 구조로 page -> frame으로 변환한다.
아래 그림에서 알 수 있듯이 virtual address는 page number와 offset으로 이루어진다. page table을 이용하여 page number를 frame number로 변환 뒤 offset과 결합한다.



아래는 80x86 hardware page table에 대한 간단한 관리자 page directory에 관한 함수, 변수들이다. 이번 프로젝트에서 변경할 필요는 없을 것으로 예상되며 아래 함수들을 호출해 사용할 예정이다.

`pagedir_create(void)`

```
uint32_t *
pagedir_create (void)
{
  uint32_t *pd = palloc_get_page (0);
  if (pd != NULL)
    memcpy (pd, init_page_dir, PGSIZE);
  return pd;
}
```

> 새로운 page directory를 생성하고 반환하는 함수

`palloc_get_page`를 통해 page directory에 페이지를 할당하고 `init_page_dir`을 복사하여 kernel virtual address에 대한 mapping을 추가해준다. 그리고 해당 page directory를 반환한다.

`pagedir_destroy(uint32_t *pd)`

```
void
pagedir_destroy (uint32_t *pd)
{
  uint32_t *pde;

  if (pd == NULL)
    return;

  ASSERT (pd != init_page_dir);
  for (pde = pd; pde < pd + pd_no (PHYS_BASE); pde++)
    if (*pde & PTE_P)
      {
        uint32_t *pt = pde_get_pt (*pde);
        uint32_t *pte;

        for (pte = pt; pte < pt + PGSIZE / sizeof *pte; pte++)
          if (*pte & PTE_P)
            palloc_free_page (pte_get_page (*pte));
        palloc_free_page (pt);
      }
  palloc_free_page (pd);
}
```

> 입력 받은 `pd` 페이지 디렉토리를 삭제하고 해당 페이지가 참조하는 모든 페이지를 해제(free)하는 함수

입력 받은 `pd`의 페이지 디렉토리 엔트리를 순회하며 `pde_get_pt`를 통해 페이지 디렉토리 엔트리에 해당하는 페이지 테이블 `pt`를 찾고, `pt`의 엔트리를 돌며 이에 해당되는 페이지를 `palloc_free_page`를 통해 모두 해제한다. 이후 페이지 테이블 `pt`, 페이지 디렉토리 `pd` 또한 해제한다.

`loopup_page(uint32_t *pd, const void *vaddr, bool create)`

```
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)];
}
```

> 입력한 `pd` 페이지 디렉토리에서 `vaddr` virtual address에 대한 페이지 엔트리 주소를 반환하는 함수. `create` 인수에 따라 `vaddr`에 대한 페이지 테이블이 없으면 생성하고 그 주소를 반환하거나 `null` 포인터를 반환한다.

`pd + pd_no (vaddr)`을 통해 `vaddr`의 주소가 포함되는 page number을 얻고, 이를 페이지 디렉토리 주소 `pd`로부터 더해 page directory entry 얻은 뒤 존재하는지 조회한다.

만약 없다면 `create`가 참이면 `pde_crate`를 통해 생성하고 거짓이면 `null`을 반환한다.
이후 `pde_get_pt`를 통해 디렉토리 엔트리 `pde`에 대한 page table `pt`를 얻고 `&pt[pt_no(vaddr)]`을 통해 `vaddr`에 해당하는 페이지 엔트리 주소를 반환한다.

`pagedir_set_page`

```
bool
pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)
{
  uint32_t *pte;

  ASSERT (pg_ofs (upage) == 0);
  ASSERT (pg_ofs (kpage) == 0);
  ASSERT (is_user_vaddr (upage));
  ASSERT (vtop (kpage) >> PTSHIFT < init_ram_pages);
  ASSERT (pd != init_page_dir);

  pte = lookup_page (pd, upage, true);

  if (pte != NULL)
    {
      ASSERT ((*pte & PTE_P) == 0);
      *pte = pte_create_user (kpage, writable);
      return true;
    }
  else
    return false;
}
```

> 페이지 디렉토리 `pd`에 user virtual page `upage` -> kernel virtual address인 physical 프레임 `kpage`로의 매핑을 추가하는 함수

`loopup_page`를 통해 `pd` 페이지 디렉토리에 `upage`에 해당되는 페이지 테이블 엔트리 주소를 찾는다. 없으면 생성한다. 찾지 못하거나 생성하지 못하였다면 false를 반환한다. 이후 찾은 페이지 테이블 엔트리 `pte`에 `pte_create_user`를 사용해 `kpage`를 향하는 page entry를 넣는다. 이 때 writable 여부가 포함된다.

`pagedir_get_page(uint32_t *pd, const void *uaddr)`

```
void *
pagedir_get_page (uint32_t *pd, const void *uaddr)
{
  uint32_t *pte;

  ASSERT (is_user_vaddr (uaddr));

  pte = lookup_page (pd, uaddr, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    return pte_get_page (*pte) + pg_ofs (uaddr);
  else
    return NULL;
}
```

> `pd` page directory에서 `uaddr` user virtual address에 대응되는 physical address를 찾고 이에 대응되는 kernel virtual address를 반환하는 함수.

`loopup_page`를 통해 `pd` 페이지 디렉토리에서 `uaddr`에 해당되는 페이지 테이블 엔트리 주소를 얻는다. `pte_get_page`를 통해 해당 page table entry가 가리키는 페이지 포인터를 얻고 이에 `pg_ofs(uaddr)` 오프셋을 더해 반환한다. 만약 `lookup_page`에서 `uaddr`에 대응되는 페이지 테이블 엔트리를 찾지 못하였다면 `uaddr`에 대응되는 주소가 없는 것이므로 null pointer를 반환한다.

`pagedir_clear_page(uint32_t *pd, void *upage)`

```
void
pagedir_clear_page (uint32_t *pd, void *upage)
{
  uint32_t *pte;

  ASSERT (pg_ofs (upage) == 0);
  ASSERT (is_user_vaddr (upage));

  pte = lookup_page (pd, upage, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    {
      *pte &= ~PTE_P;
      invalidate_pagedir (pd);
```

```
    }
}
```

> page directory `pd` 에서의 user virtual page `upage` 를 존재하지 않게 보이게 만드는 함수

`lookup_page` 를 통해 `pd` 에서 `upage` 에 해당되는 페이지 테이블 엔트리를 찾고 해당 엔트리를 초기화한다. 이후 `invalidate_pagedir` 을 통해 `pd` 와 관련된 TLB를 invalidate한다.

`pagedir_is_dirty(uint32_t *pd, const void *vpage)`

```
bool
pagedir_is_dirty (uint32_t *pd, const void *vpage)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  return pte != NULL && (*pte & PTE_D) != 0;
}
```

> `pd` 에서 `vpage` virtual page에 대한 PTE가 dirty한 여부를 반환하는 함수

만약 `pd` 에서 `vpage` 에 대한 page table entry를 찾을 수 없거나 `*pte & PTE_D` 결과 dirty하지 않으면 false를 반환, 두 경우가 아니라면 true를 반환한다.

`pagedir_set_dirty(uint32_t *pd, const void *vpage, bool dirty)`

```
void
pagedir_set_dirty (uint32_t *pd, const void *vpage, bool dirty)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  if (pte != NULL)
    {
      if (dirty)
        *pte |= PTE_D;
      else
        {
          *pte &= ~(uint32_t) PTE_D;
          invalidate_pagedir (pd);
        }
    }
}
```

> `pd` page directory의 `vpage` virtual page에 해당하는 page table entry의 dirty 값을 변경하는 함수

Process

`process_execute(const char *file_name)`

```
tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;

  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE);

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

> `file_name` 에서 로드한 user program을 실행하는 새로운 thread를 생성하는 함수

`palloc_get_page` 를 통해 page를 할당 받고 그 페이지의 주소를 `fn_copy` 에 저장한다. 이후 `fn_copy` 가 가리키는 위치에 `file_name` 을 복사한다. 이후 `thread_create` 를 통해 `file_name` 을 이름으로 가지고 `fn_copy` argument와 함께 `start_process` 를 실행하는 스레드를 생성한다. 성공적으로 생성했다면 이 스레드의 thread id를 반환한다,

이때 주의할 점은 `process_execute()` 반환되기 전에 이로 인해 생성된 스레드가 종료되거나 스케줄링될 수 있다는 것이다.

`start_process(void *file_name_)`

```
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 ();

  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

> `file_name_` 로부터 ELF executable을 로드하고 이를 실행하는 함수

`process_execute` 에 의해 생성되는 스레드가 실행할 함수이다.
`intr_frame` 인 `if_` 를 0으로 초기화한 뒤 현재 세그먼트가 유저 코드 세그먼트라는 것을 포함해 `if_` 의 `gs`, `cs`, `eflags` 값을 초기화해준다.
이후 `load` 를 통해 `file_name` 으로부터 ELF를 로드하고 `if_.eip` 에 entry point를, `if_.esp` 에 초기 stack pointer를 저장한다. 만약 이를 실패시 `thread_exit()` 를 통해 현재 스레드를 삭제한다.
또한 `file_name_` 를 넘기느라 할당했던 page를 free해준다.

```
asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
```

위 코드를 통해 `intr_exit` 을 통해 interrupt에서 돌아오는 것처럼 user process를 실행한다.

- `intr_exit` 은 스택에서 `intr_frame` 형태로 값을 가져온다. 즉 위에서 설정한 `if_` 을 사용할 것이다.
  - `if_` 의 return address, 즉 eip가 user process임.
- `esp` 가 위에서 설정한 `if_` 를 가르키게 한 뒤 `intr_exit` 으로 점프해 `intr_exit` 이 실행되도록 한다.

`process_wait(tid_t child_tid)`

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

우리가 구현해야 할 함수로 현재는 항상 -1을 리턴한다.
이상적인 작동은 `child_tid` 의 스레드가 종료될 때까지 기다리다가 그 스레드의 exit status를 반환하고 kernel에 의해 종료되었거나 잘못된 요청일 경우 -1을 리턴하는 것이다.

`process_exit(void)`

```
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      cur->pagedir = NULL;
      pagedir_activate (NULL);
```

```
      pagedir_destroy (pd);
    }
}
```

> 현재 실행 중인 process에게 할당된 자원을 모두 해제하는 함수.

해당 함수 내 실행 순서는 매우 중요하다. 먼저 현재 실행 중인 스레드의 `pagedir`을 `null`로 변경하여 해당 page directory로 다시 switch되지 않도록 해야한다.
`pagedir_activate(NULL)`을 통해 base page directory를 활성화한 뒤 현재 실행되는 있는 프로세스(스레드)의 `pagedir` page directory를 `pagedir_destroy`를 통해 삭제하고 그것이 참조하는 모든 페이지를 모두 free한다.

- 이 때 삭제 전 반드시 base page directory를 먼저 활성화 해야 한다.

`process_activate(void)`

```
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 ();
}
```

> context switch마다 실행되어 현재 스레드에서 user code가 실행될 수 있도록 cpu를 설정하는 함수

`pagedir_activate`를 통해 현재 스레드의 `pagedir`, page table를 활성화한다.
`tss_update`를 통해 현재 스레드 스택 끝을 tss의 stack pointer가 가리키게 한다. 이 이유는 interrupt 발생시 어떤 프로세스에 의해 발생하였는지(user에 의해서인지, kernel에 의해서인지, `exception.c`의 `kill` 참고) 알기 위함이다.

`load(const char *file_name, void (**eip) (void), void **esp)`

> 주어진 `file_name`으로부터 ELF executable을 현재 스레드에 로드하는 user program에서 가장 핵심적인 함수이다.

`start_process`에서 호출해 사용하며 주어진 `file_name`으로부터 ELF executable을 현재 스레드에 로드하고 executable의 entry point를 EIP에, 초기 stack pointer를 esp에 저장하는 함수이다. 이 때 성공시 true를, 실패시 false를 반환한다.

```
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 ();
```

`pagedir_create()`를 통해 현재 스레드에 page directory를 할당하고 이를 스레드의 `pagedir`에 저장한다. 성공시 `process_activate()`를 통해 해당 page directory를 활성화하고 tss를 업데이트해 해당 스레드의 스택 끝을 esp에 저장한다.

```
  /* 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;
    }
```

루트 디렉토리에 있는 파일 중 이름이 `file_name` 인 파일을 찾아 연다. `file_read` 를 통해 `file` 을 Executable header의 크기만큼 읽어 들여 `ehdr` 에 저장한 뒤 올바른 Executable header를 가지고 있는지 검증한다. Executable header은 ELF binary의 매우 앞 부분에 등장하는 헤더로 executable에 대한 정보를 담고 있다.

```
  /* 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 phdr)
        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 the rest. */
                  read_bytes = page_offset + phdr.p_filesz;
                  zero_bytes = (ROUND_UP (page_offset + phdr.p_memsz, PGSIZE)
                                - read_bytes);
                }
              else
                {
                  /* Entirely zero.
                     Don't read anything from disk. */
                  read_bytes = 0;
                  zero_bytes = ROUND_UP (page_offset + phdr.p_memsz, PGSIZE);
                }
              if (!load_segment (file, file_page, (void *) mem_page,
                                 read_bytes, zero_bytes, writable))
                goto done;
            }
          else
            goto done;
          break;
        }
    }
```

`file_ofs`에 `ehdr.e_phoff`으로부터 얻은 program header의 위치(offset)을 저장한다.
`file_ofs`부터 `file_read`을 통해 program header 크기만큼 `ehdr.e_phnum`(program header 개수)번 읽어들이며 매 헤더에 대해 아래를 수행한다.

- 읽어들인 program header는 `phdr`에 저장된다. `phdr.p_type`, segment 타입을 보고 `PT_LOAD`인 경우 (loadable segment인 경우) 다음을 수행한다.
  - 해당 `phdr` 프로그램 헤더가 올바른 segment를 나타내는지 `validate_segment`를 통해 검증한다.
  - `phdr`로부터 해당 세그먼트에 대한 각종 정보(writable, 파일 내 위치 등)를 뽑아내고 `phdr.p_filesz`가 0 초과라면(파일 내 세그먼트가 차지하는 크기가 0 초과) 일반적인 segment이고 그렇지 않으면 zero로 이루어진 segment일 뿐이다.
  - 일반적인 세그먼트라면 `load_segment`를 통해 세그먼트를 로드한다.

```
  /* 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;
}
```

`setup_stack`을 통해 stack을 초기화하고 매개변수의 `eip`에 executable 시작 위치를 저장한다.
모두 완료되면 `file_close`를 통해 파일을 닫고 성공 여부를 반환한다.

`validate_segment(const struct Elf32_Phdr *phdr, struct file *file)`

```
static bool
validate_segment (const struct Elf32_Phdr *phdr, struct file *file)
{
  /* p_offset and p_vaddr must have the same page offset. */
  if ((phdr->p_offset & PGMASK) != (phdr->p_vaddr & PGMASK))
    return false;

  /* p_offset must point within FILE. */
  if (phdr->p_offset > (Elf32_Off) file_length (file))
    return false;

  /* p_memsz must be at least as big as p_filesz. */
  if (phdr->p_memsz < phdr->p_filesz)
    return false;

  /* The segment must not be empty. */
  if (phdr->p_memsz == 0)
    return false;

  /* The virtual memory region must both start and end within the
     user address space range. */
  if (!is_user_vaddr ((void *) phdr->p_vaddr))
    return false;
  if (!is_user_vaddr ((void *) (phdr->p_vaddr + phdr->p_memsz)))
    return false;

  /* The region cannot "wrap around" across the kernel virtual
     address space. */
  if (phdr->p_vaddr + phdr->p_memsz < phdr->p_vaddr)
    return false;

  if (phdr->p_vaddr < PGSIZE)
    return false;

  /* It's okay. */
  return true;
}
```

> 주어진 `Elf32_Phdr`인 `phdr` 즉 ELF의 program header가 `file` 내에서 유효한지, load할 수 있는 segment에 대한 것인지 체크하고 그 결과를 반환하는 함수

아래 중 하나라도 만족하지 못하면 false 반환, 모두 만족하면 true 반환환
체크 항목

- `phdr`의 `p_offset`과 `p_vaddr`이 같은 페이지 offset을 가지는 확인
- `phdr`의 `p_offset`이 `file`의 메모리 범위 내에 위치하는지 확인
- `phdr`의 `p_memsz`가 `p_filesz`보다 크거나 같은지 확인
  - 메모리 상 크기가 파일 내 크기보다 큰가?
- `phdr`의 `p_memsz`가 아닌지 확인 ==> segment가 비어있지 않은지 확인
- segment의 시작, 끝 위치가 모두 user address space 범위 내인지 확인
  - `p_vaddr`(처음), `p_vaddr+p_memsz`(끝) 모두 `is_user_vaddr()`이 참이여야 함.
- `p_vaddr` + `p_memsz` >= `p_vaddr` 이여야 함.
- segment에 할당된 페이지0이 아닌지 확인
- `p_vaddr` >= `PGSIZE`

`load_segment`

```
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool 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_bytes : 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;
}
```

> `file` 내의 `ofs`으로부터 시작하는 segment를 `upage`에 로드하고 매핑을 page table에 추가하는 함수

`file_seek`을 통해 파일 `file`이 가리키는 오프셋을 `ofs`로 변경한다.
우선 read해 load 또는 0으로 채우는 것은 한 페이지 단위로 진행한다. `palloc_get_page`를 통해 user page를 할당 받고 `file_read`를 통해 읽어와 할당 받은 page에 저장하고 해당 페이지 끝에서부터 `page_zero_byte`(PGSIZE - 이번에 read한 바이트)만큼 0으로 초기화 한다. `install_page`를 통해 `upage`와 `kpage` 매핑을 page table에 추가한다. 이 때 writable 여부는 입력 받은 인수에 따른다. 이를 `read_bytes`만큼 모두 로드할 때까지 페이지 단위로 반복한다.
성공하면 true를 반환하고 중간에 실패했으면 false를 반환한다.

`setup_stack(void **esp)`

```
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) - PGSIZE, kpage, true);
      if (success)
        *esp = PHYS_BASE;
      else
        palloc_free_page (kpage);
    }
  return success;
}
```

> user virtual memory 최상단에 0으로 초기화된 페이지를 매핑하여 stack을 초기화하는 함수

`load`에서 ELF 로드 완료 이후 stack을 초기화할 때 사용한다.
`palloc_get_page`를 통해 0으로 초기화된 user page를 할당하고 `install_page`를 통해 해당 페이지-kernel virtual address 매핑을 page table에 추가한다.

`install_page(void *upage, void *kpage, bool writable)`

```
static bool
install_page (void *upage, void *kpage, bool writable)
{
  struct thread *t = thread_current ();

  /* Verify that there's not already a page at that virtual
     address, then map our page there. */
  return (pagedir_get_page (t->pagedir, upage) == NULL
          && pagedir_set_page (t->pagedir, upage, kpage, writable));
}
```

> user virtual address -> kernel virtual address 매핑을 page table에 추가하는 함수

주어진 user virtual address인 `upage` -> kernel virtual address인 `kpage` 매핑을 page table에 추가하는 함수로 입력받은 `writable` 값에 따라 read-only, writeable을 결정해 page table에 추가에 함께 적용한다.

Exception

user process가 privileged(권한이 필요한) 또는 prohibited(금지된) operation을 실행하려고 하면, 커널에 exception이나 fault로 trap된다. 실제 Unix-like OS에서는 대부분의 interrupt는 user process에게 signal의 형태로 전달되는데 핀토스 현재 구현상 모든 exception은 단순히 메세지 출력 후 프로세스 종료한다. 현재는 Page fault 또한 다른 exception과 동일하게 다루지만 이를 변경하는 것이 이번 assn 목표이다.

```
#define PF_P 0x1    /* 0: not-present page. 1: access rights violation. */
#define PF_W 0x2    /* 0: read, 1: write. */
#define PF_U 0x4    /* 0: kernel, 1: user process. */
```

Page fault error code는 뒤에서부터 비트 단위로 오류의 이유를 설명한다.
맨 뒷자리

- 0: 존재하지 않는 페이지, 1: 접근 권한 없음.
  뒤에서 두번째
- 0: read할 때, write할 때
  뒤에서 세번째
- 0: kernel, 1: user process

```
static long long page_fault_cnt;
```

처리된 page fault 횟수를 나타내는 전역 변수

`exception_init(void)`

```
void
exception_init (void)
```

```
{
  intr_register_int (3, 3, INTR_ON, kill, "#BP Breakpoint Exception");
  intr_register_int (4, 3, INTR_ON, kill, "#OF Overflow Exception");
  intr_register_int (5, 3, INTR_ON, kill,
                     "#BR BOUND Range Exceeded Exception");

  intr_register_int (0, 0, INTR_ON, kill, "#DE Divide Error");
  intr_register_int (1, 0, INTR_ON, kill, "#DB Debug Exception");
  intr_register_int (6, 0, INTR_ON, kill, "#UD Invalid Opcode Exception");
  intr_register_int (7, 0, INTR_ON, kill,
                     "#NM Device Not Available Exception");
  intr_register_int (11, 0, INTR_ON, kill, "#NP Segment Not Present");
  intr_register_int (12, 0, INTR_ON, kill, "#SS Stack Fault Exception");
  intr_register_int (13, 0, INTR_ON, kill, "#GP General Protection Exception");
  intr_register_int (16, 0, INTR_ON, kill, "#MF x87 FPU Floating-Point Error");
  intr_register_int (19, 0, INTR_ON, kill,
                     "#XF SIMD Floating-Point Exception");

  intr_register_int (14, 0, INTR_OFF, page_fault, "#PF Page-Fault Exception");
}
```

> page fault를 포함해 각종 exception에 대한 핸들러 함수를 등록하는 함수이다.

해당 함수는 `syscall_init()` 과 함께 `threads/init.c` 의 `main()` 에서 커널을 초기화할 때 호출되어 `exception` 에 대한 핸들러를 등록한다.
(`syscall_init` 은 interrupt 중 system call에 대한 핸들러 등록)
`intr_register_int` 를 통해 `BP, OF, BR` exception에 대한 interrupt handler 함수로 `kill` 을 등록한다. 이 exception은 dpl이 3으로 user program에 의해서 명시적으로 발생할 수 도 있다.
`intr_register_int` 를 통해 `DE, DB, UD, NP, SS, NM, GP, MF ,XF` exception에 대한 interrupt handler 함수로 `kill` 을 등록한다. 이 exception은 dpl이 0으로 user process에 의해 명시적으로 발생할 수 없으나 조건에 따라 간접적으로 발생할 수 있다.
위 exception에 대해서는 항상 핸들러 함수인 `kill` 을 호출해 해당 user process를 죽인다.
마지막으로 page fault에 대한 핸들러 함수로 `page_fault` 를 등록하였고 이는 dpl=0으로 user process가 직접적으로 발생시킬 수 없으며 `INTR_OFF` 로 interrupt가 off된 상태로 핸들링한다. 이는 fault address가 CR2에 저장되고 보존되어야 하기 때문이다.

`exception_print_stats(void)`

```
void
exception_print_stats (void)
{
  printf ("Exception: %lld page faults\n", page_fault_cnt);
}
```

> page fault 발생 횟수를 출력하는 함수이다.

`kill(struct intr_frame *f)`

```
static void
kill (struct intr_frame *f)
{
  switch (f->cs)
    {
    case SEL_UCSEG:
      printf ("%s: dying due to interrupt %#04x (%s).\n",
              thread_name (), f->vec_no, intr_name (f->vec_no));
      intr_dump_frame (f);
      thread_exit ();

    case SEL_KCSEG:
      intr_dump_frame (f);
      PANIC ("Kernel bug - unexpected interrupt in kernel");

    default:
      printf ("Interrupt %#04x (%s) in unknown segment %04x\n",
             f->vec_no, intr_name (f->vec_no), f->cs);
      thread_exit ();
    }
}
```

> page fault를 제외한 exception에 대한 handler 함수이다.

page fault를 제외한 exception에 대한 handler 함수로 사용된다.(page fault 핸들러 함수 내부에서 해당 함수를 호출함.)
핸들러 함수인 해당 함수를 호출할 때 인수로 들어온 `intr_frame` 의 `cs` , 즉 interrupt frame의 eip의 code segment(interrupt 발생 위치)를 보고 user와

kernel 중 어디에서 exception이 발생하였는지 구별한다. `SEL_UCSEG`, user code segment라면 user exception이므로 관련 에러 메세지를 출력하고 디버깅을 위해 interrupt frame을 덤프시킨다. 이후 `thread_exit`을 통해 해당 스레드를 종료시킨다. 즉 해당 user process는 죽는다.
만약 `cs`가 `SEL_KCSEG`라면 kernel에서 발생한 exception이므로 interrupt frame을 덤프시키고 패닉한다.
이 외의 경우는 발생할 수 없지만 패닉한다.

`page_fault(struct intr_Frame *f)`

```
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

> page fault exception에 대한 handler 함수이다.

page fault가 일어나게 한 가상 주소는 CR2 레지스터에 저장되어 있으며 `"movl %%cr2, %0" : "=r"`을 통해 얻어 `fault_addr`에 저장한다.
`fault_addr`을 올바르게 얻었으므로 `intr_enable`을 통해 interrupt를 다시 활성화 한다. page fault가 발생했기에 `page_fault_cnt`를 1 증가시켜준다.
핸들러 함수의 인수로 받은 interrupt frame의 `error_code`는 에러 코드를 포함하고 있고 page fault의 에러코드는 위에서 명시한 대로 오류의 이유에 대해 설명한다. 이를 이용해 세가지 기준에 대해 오류 원인을 추출하고 이에 관해 메세지로 출력한다.
마지막으로 `kill`을 호출해 user process를 죽인다.
page fault의 exception 핸들링과 관련해 가상 메모리를 올바르게 구현을 완료하기 위해서는 해당 함수를 수정해야 할 필요가 있다.

Interrupt

`idt`

```
static uint64_t idt[INTR_CNT];
```

IDT(Interrupt Descriptor Table)을 저장하는 리스트 변수, `intr-stubs.S`에 있는 256개의 x86 interrupt stub과 연결하는 Gate를 저장한다. `intr_init`에서 초기화된다.

`intr_handlers`

```
static intr_handler_func *intr_handlers[INTR_CNT];
```

`idt`에 있는 각 interrupt에 대한 intr handler function을 담은 리스트

`intr_level`

```
enum intr_level
  {
    INTR_OFF,             /* Interrupts disabled. */
    INTR_ON               /* Interrupts enabled. */
  };
```

Interrupt 활성화, 비활성화 여부를 표현하는 enum

`intr_get_level`

```
enum intr_level
intr_get_level (void)
{
  uint32_t flags;

  /* Push the flags register on the processor stack, then pop the
     value off the stack into `flags'.  See [IA32-v2b] "PUSHF"
     and "POP" and [IA32-v3a] 5.8.1 "Masking Maskable Hardware
     Interrupts". */
  asm volatile ("pushfl; popl %0" : "=g" (flags));

  return flags & FLAG_IF ? INTR_ON : INTR_OFF;
}
```

processor stack에 flag register를 넣고 pop시켜 flag에 저장하여 flag 값을 확인하여 현재 Interrupt 활성화 여부를 `intr_level`로 반환하는 함수

`intr_set_level`

```
enum intr_level
intr_set_level (enum intr_level level)
{
  return level == INTR_ON ? intr_enable () : intr_disable ();
}
```

입력 받은 `intr_level`에 따라 `intr_enable` 또는 `intr_disable`을 통해 interrupt 활성화 여부를 조절하는 함수

`intr_enable`

```
enum intr_level
intr_enable (void)
{
  enum intr_level old_level = intr_get_level ();
  ASSERT (!intr_context ());

  /* Enable interrupts by setting the interrupt flag.

     See [IA32-v2b] "STI" and [IA32-v3a] 5.8.1 "Masking Maskable
     Hardware Interrupts". */
  asm volatile ("sti");

  return old_level;
}
```

> interrupt 를 enable하고 변경 전 원래의 interrupt level을 `intr_level` 꼴로 반환하는 함수

external interrupt를 처리하고 있지 않을 때 실행되어야 한다.
`sti` 어셈블리를 통해 interrupt flag를 설정함으로써 interrupt를 허용하고 변경 전 원래의 interrupt level을 반환한다.

`intr_disable`

```
enum intr_level
intr_disable (void)
{
  enum intr_level old_level = intr_get_level ();

  /* Disable interrupts by clearing the interrupt flag.
     See [IA32-v2b] "CLI" and [IA32-v3a] 5.8.1 "Masking Maskable
     Hardware Interrupts". */
  asm volatile ("cli" : : : "memory");

  return old_level;
}
```

> interrupt 를 disable하고 변경 전 원래의 interrupt level을 `intr_level` 꼴로 반환하는 함수

`cli` 어셈블리를 통해 interrupt flag를 지움으로써 interrupt를 비활성화하고 변경 전 원래의 interrupt level을 반환한다.

`intr_init`

```
void
intr_init (void)
{
  uint64_t idtr_operand;
  int i;

  /* Initialize interrupt controller. */
  pic_init ();

  /* Initialize IDT. */
  for (i = 0; i < INTR_CNT; i++)
    idt[i] = make_intr_gate (intr_stubs[i], 0);

  /* Load IDT register.
     See [IA32-v2a] "LIDT" and [IA32-v3a] 5.10 "Interrupt
     Descriptor Table (IDT)". */
  idtr_operand = make_idtr_operand (sizeof idt - 1, idt);
  asm volatile ("lidt %0" : : "m" (idtr_operand));

  /* Initialize intr_names. */
  for (i = 0; i < INTR_CNT; i++)
    intr_names[i] = "unknown";
  intr_names[0] = "#DE Divide Error";
  intr_names[1] = "#DB Debug Exception";
  intr_names[2] = "NMI Interrupt";
  intr_names[3] = "#BP Breakpoint Exception";
  intr_names[4] = "#OF Overflow Exception";
  intr_names[5] = "#BR BOUND Range Exceeded Exception";
  intr_names[6] = "#UD Invalid Opcode Exception";
  intr_names[7] = "#NM Device Not Available Exception";
  intr_names[8] = "#DF Double Fault Exception";
  intr_names[9] = "Coprocessor Segment Overrun";
  intr_names[10] = "#TS Invalid TSS Exception";
  intr_names[11] = "#NP Segment Not Present";
  intr_names[12] = "#SS Stack Fault Exception";
  intr_names[13] = "#GP General Protection Exception";
  intr_names[14] = "#PF Page-Fault Exception";
  intr_names[16] = "#MF x87 FPU Floating-Point Error";
  intr_names[17] = "#AC Alignment Check Exception";
  intr_names[18] = "#MC Machine-Check Exception";
  intr_names[19] = "#XF SIMD Floating-Point Exception";
}
```

> Interrupt system을 초기화하는 함수

`pic_init()` 을 통해 interrupt controller를 초기화하고 `intr_stubs` 를 순회하며 x86 interrupts에 대한 `make_intr_gate` 를 통해 커널 레벨에서 `intr_stubs[i]` 을 호출하는 interrupt gate를 만든다. 또한 `lidt` 를 통해 IDT(Interrupt Descriptor Table) register를 로드한다. `intr_names` 에 interrupt들의 이름을 집어 넣는다.

`intr_handler`

> 모든 interrupt, exception, fault 등을 핸들링하는 핸들러로 `intr-stubs.S` 에 의해 호출되는 함수이다

```
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;
    }
```

`frame` 은 `intr_frame` 으로 interrupt 및 interrupt 발생 스레드에 대한 정보를 담고 있다.
`frame->vect_no` 즉 발생한 interrupt vector가 0x20(타이머 인터럽트)이상이면 external interrupt로 취급하여 `yield_on_return` (interrupt에서 돌아올 때 yield 여부)을 false, `in_external_intr` 을 true로 한다.

```
  /* 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)
    {
    }
  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_handlers[frame->vec_no]` 를 통해 interrupt에 해당되는 등록된 interrupt handler를 `frame` 을 패싱해 호출한다. external interrupt 처리를 완료 후 리턴되면 `in_external_intr` 을 false로 복구하고 `yield_on_return` 값에 따라 thread를 yield한다.

- `intr_register_int` , `register_handler` , `intr_register_ext` 를 통해 `intr_handlers` 에 interrupt vector에 해당되는 핸들러 함수가 들어가게 된다.

intr-stubs

```
.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
```

`intrNN_stub` 에 의해 internal 또는 external interrupt가 시작되며, `intr_frame` 에 `frame_pointer` , `error_code` , `vec_no` 를 모두 push한 뒤 `intr_entry` 로 점프하게 된다.
`intr_entry` 에서는 interrupt가 발생한 스레드의 register 값들을 stack 넣어 저장한다. 이후 `mov $SEL_KDSEG, %eax` 를 통해 커널 환경으로 변경한 이후 frame pointer를 설정한다. 마지막으로 `esp` 를 넣은 다음 `intr_handler` 를 호출하고 stack point를 증가시킨다.

```
.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`은 interrupt 처리를 마무리할 때 호출되는 함수로 `intr_exit`에서는 넣어 두었던 레지스터 값들을 pop해 복구하고 esp에 12를 더해 `intr_frame`의 `vec_no`, `error_code`, `frame_pointer members`를 무시한다. 그리고 return 된다.

```
        .data
.globl intr_stubs
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;
```

User System call

`lib/user/syscall.c`에는 유저가 프로그램에 사용할 수 있는 system call 발생 함수들이 존재한다. user는 다음 함수들을 직접 호출하여 0x30 interrupt 즉 system call을 발생시키고 `intr_frame`이 인수로 넘어간 `syscall_handler`가 호출된다.

`syscall0~3`

```
#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;                                           \
        })
```

> argument를 패싱하고 syscall `NUMBER`을 발생하고 리턴 값을 int로 리턴하는 함수

현재 스레드의 스택에 뒤의 argument부터 순차적으로 집어넣고 마지막에는 syscall number인 `NUMBER`를 집어넣는다. 이후 syscall로부터 리턴된 값을 int로 리턴한다.

- 스택에 집어넣은 값들은 `syscall_handler`에서 `intr_frame`의 `esp`(현재 값을 집어넣은 스택의 끝 주소)로부터 역순으로 꺼내 어떤 syscall인지 구분거나 argument를 얻을 수 있다. 하지만 현재 구현 상에는 단지 프로세스를 종료시킨다.
- syscall handler로부터 return되는 값은 handler에서 리턴값을 eax에 넣어 eax 레지스터에 담겨 리턴되는 것이다.
  passing하는 argument의 개수는 함수 뒤의 숫자이다. 아래의 각 syscall을 발생하는 함수에서 syscall 별 필요한 argument의 개수에 따라 syscall0~3 중 적절한 함수를 호출한다.

아래는 유저가 직접 호출하여 직접 원하는 system call을 발생시키는 함수들이다. 이에는 이번에 제공해야하는 syscall에 대한 함수도 있고 assn3에서 구현해야하는 syscall도 존재한다.

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

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

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

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

위는 많은 syscall에 대한 함수 중 일부이며 syscall에 따라 argument의 개수와 return value 유무, 타입 등이 다르다.

System call

유저 프로세스가 kernel functionality에 접근하고 싶으면 system call을 invoke(호출)하면 된다. 현재 pintos에 구현된 것은 skeleton system call handler로 system call을 invoke할 시 handler에 의해 `"system call!"` 메세지를 출력하고 user process terminate한다. 이번 assn에서 이를 requirement에 맞게 변경하여 구현하면 된다.

System Call은 다음 순서로 호출된다.
`lib/user/syscall.c`의 `syscall` 함수 **-->** `intrNN_stub` **-->** `intr_entry` **-->** `intr_handler` **-->** `syscall_handler` **-->** 각`system call`의 대응 함수

`syscall_handler(struct intr_frame *)`

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

> system call이 발생시 호출되는 handler 함수이다.

현재는 skeleton 함수로 단지 `"system call!"`을 출력하고 `thread_exit`을 통해 스레드를 종료시킴으로써 user process를 terminate한다.

`syscall_init(void)`

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

> system call interrupt에 대한 handler를 등록하여 system call 시스템을 세팅하는 함수

`threads/init.c`의 `main`에서 커널을 초기화할 때, 호출되어 system call 시스템을 세팅한다.
`intr_register_int`를 통해 0x30 interrupt vector에 반응하고 user mode에서도 호출 가능하게 dpl을 3으로 설정하여 `syscall_handler`를 interrupt handler 함수로 등록한다.

File System

Inode

Inode (Index node)란 UNIX 계통의 파일 시스템에서 주로 사용하는 자료구조이며, 운영체제 상의 각 파일은 하나의 Inode와 대응된다. 각 Inode는 데이터의 시작 부분이 담겨있는 Block의 인덱스, 데이터의 길이, 현재 파일의 상태 등의 메타데이터를 담고 있다. Pintos 역시 Inode 자료구조를 사용하며 `filesys/inode.c` 에 해당 구현체가 정의되어 있지만 실제 UNIX계열 운영체제에서 사용하는 방식보다는 간략화되어있다.

디스크에 존재하는 모든 데이터는 일정한 크기의 Block 단위로 저장되어 구분되는데, 해당 크기인 `BLOCK_SECTOR_SIZE` 는 `devices/block.h` 에 512B로 전처리 되어있다.

```
/* In-memory 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. */
  };
```

Inode의 구조체. 현재 열려있는 `inode` 들의 리스트인 `open_inodes` 와 상호작용하기 위한 `elem` , 디스크에 저장되어 있는 inode block을 가리키기 위한 인덱스 `sector` 와 해당 block 데이터인 `data` , 그리고 현재 inode의 오픈 상태를 나타내는 메타데이터인 `open_cnt` , `removed` , `deny_write_cnt` 로 이루어져 있다.

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
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. */
  };
```

실제 디스크에 저장되는 Inode block을 나타내는 구조체. 파일의 데이터를 직접 담고 있지 않고 해당 데이터가 담겨있는 첫 번째 블록의 인덱스를 `start` 에 저장한다. Inode block을 나타내는 데 필요한 메타데이터가 현재로선 많지 않음에도 불구하고 `BLOCK_SECTOR_SIZE` 를 채우기 위해 `unused` 를 선언하여 더미 데이터로 512B를 채운 모습을 볼 수 있다.

해당 구현체를 보면서 `inode` 와 `inode_disk` 가 굳이 분리되어 선언되어있는 이유에 대해 의문점이 생겼는데, `inode_disk` 의 데이터는 파일 자체의 정적인 정보를 나타내는 데 반해 `inode` 의 데이터는 런타임 중의 동적인 정보를 나타내기 때문이라고 추측한다.

```
/* Identifies an inode. */
#define INODE_MAGIC 0x494e4f44
```

멤버 변수 중 `magic` 에는 전처리된 값인 `INODE_MAGIC (0x494e4f44 = "INOD")` 이 들어가는데, 해당 블록 영역이 Inode block임을 인식하는 데 사용된다.

```
/* Returns the number of sectors to allocate for an inode SIZE
   bytes long. */
static inline size_t
bytes_to_sectors (off_t size)
{
  return DIV_ROUND_UP (size, BLOCK_SECTOR_SIZE);
}
```

`size` 만큼의 바이트를 할당하는 데 필요한 블럭 영역의 수를 계산한다.

```
/* Returns the block device sector that contains byte offset POS
   within INODE.
   Returns -1 if INODE does not contain data for a byte at offset
   POS. */
static block_sector_t
byte_to_sector (const struct inode *inode, off_t pos)
{
  ASSERT (inode != NULL);
  if (pos < inode->data.length)
    return inode->data.start + pos / BLOCK_SECTOR_SIZE;
  else
    return -1;
}
```

주어진 `inode` 위에 있는 `pos` 번째 바이트가 어떤 블럭 위에 있는지를 계산한다. 위의 계산식은 단순히 첫 인덱스인 `inode->data.start` 부터 `pos` 바이트만큼의 블럭 수를 더해주므로, 한 데이터를 이루는 블럭들은 모두 메모리 상의 연속된 공간에 배치되어있다는 가정이 수반되어야 한다. 만약 `pos` 가 데이터 길이를 넘어설 경우 `-1` 을 반환한다.

```
/* List of open inodes, so that opening a single inode twice
   returns the same `struct inode'. */
static struct list open_inodes;

/* Initializes the inode module. */
void
inode_init (void)
{
  list_init (&open_inodes);
}
```

현재 오픈된 Inode들의 리스트인 상술한 `open_inodes` 와 이를 초기화하는 함수이다. 본 코드에서 전역적으로 선언되었고 초기화가 필요한 변수는 `open_inodes` 밖에 없기 때문에 시작 시 이를 초기화해주는 것으로 충분하다.

```
/* Initializes an inode with LENGTH bytes of data and
   writes the new inode to sector SECTOR on the file system
   device.
   Returns true if successful.
   Returns false if memory or disk allocation fails. */
bool
inode_create (block_sector_t sector, off_t length)
```

메모리 위의 블럭들 중 `sector` 인덱스의 블럭에 `length` 바이트의 데이터를 가리킬 Inode 블럭을 생성한다.

- 임시로 `inode_disk` 데이터를 담아둘 공간을 동적할당한 뒤 필요한 메타데이터를 채워준다.
- 디스크 위에 `length` 바이트의 데이터가 들어갈 만큼의 블럭들을 `free_map_allocate` 함수를 통해 확보해준다.
- 확보에 성공할 시 임시로 만들어놨던 `inode_disk` 블럭의 내용을 `block_write` 함수를 통해 `sector` 위치의 블럭에다 복사해준 뒤, 앞서 확보한 데이터 블럭들을 `0` 으로 초기화해준다.
- 마지막으로 임시로 할당한 블럭을 `free` 해주고 성공 여부를 반환한다.

> `free_map` 자료구조는 디스크 상에 블럭 단위로 쪼개진 영역들을 총괄하여 관리한다. 각 블럭이 현재 할당된 상태인지 아닌지를 메모리 상의 매 bit 마다 0과 1로 나타내는 `bitmap` 을 할당하여 `FREE_MAP_SECTOR(0)` 번 블럭에 저장하며, 이를 참조하여 각 블럭들을 할당 및 해제한다.

```
/* Reads an inode from SECTOR
   and returns a `struct inode' that contains it.
   Returns a null pointer if memory allocation fails. */
struct inode *
inode_open (block_sector_t sector)
```

`sector` 위치의 Inode 블럭을 가리키는 Inode를 찾거나 생성하여 주솟값을 반환한다.

- `open_inodes` 리스트를 순회하며 해당 Inode가 이미 open되어있는지를 확인한다. 만약 존재할 경우 `inode_reopen` 을 호출한 뒤 해당 Inode를 반환한다.
- 리스트에서 찾지 못했을 경우 새로운 Inode를 동적할당하여 멤버 변수들을 설정해준 뒤 반환한다.

```
/* Reopens and returns INODE. */
struct inode *
inode_reopen (struct inode *inode)
{
  if (inode != NULL)
    inode->open_cnt++;
  return inode;
}
```

앞선 `inode_open` 의 경우와 같이 이미 열려있는 Inode를 다시 열려고 할 때 호출되는 함수. 해당 Inode를 연 횟수인 `inode->open_cnt` 를 증가시킨다.

```
/* Closes INODE and writes it to disk.
   If this was the last reference to INODE, frees its memory.
   If INODE was also a removed inode, frees its blocks. */
void
inode_close (struct inode *inode)
```

`inode->open_cnt` 를 감소시켜 열려있던 Inode를 닫는다.

- 만약 `inode->open_cnt` 를 감소시켜 `0` 이 되었을 경우, 더이상 해당 Inode에 대한 opener가 없다는 뜻이므로 할당했던 Inode를 해제해준다.

- 이때 만약 `inode_remove` 함수에 의해 `removed` 플래그가 설정되었던 상태라면 해당 Inode 블럭과 데이터 블럭들을 `free_map_release` 함수를 통해 모두 해제해준다.

```
/* Reads SIZE bytes from INODE into BUFFER, starting at position OFFSET.
   Returns the number of bytes actually read, which may be less
   than SIZE if an error occurs or end of file is reached. */
off_t
inode_read_at (struct inode *inode, void *buffer_, off_t size, off_t offset)
```

`inode`의 데이터 중 `offset` 위치부터의 데이터를 최대 `size` 바이트만큼 `buffer_` 공간에 복사한다. 한 번에 최대 `BLOCK_SECTOR_SIZE`만큼의 데이터를 복사하며 총 `size`만큼의 바이트를 복사했거나 파일 데이터의 끝에 도달한 경우 종료한다.

```
/* Writes SIZE bytes from BUFFER into INODE, starting at OFFSET.
   Returns the number of bytes actually written, which may be
   less than SIZE if end of file is reached or an error occurs.
   (Normally a write at end of file would extend the inode, but
   growth is not yet implemented.) */
off_t
inode_write_at (struct inode *inode, const void *buffer_, off_t size,
                off_t offset)
```

위의 `inode_read_at`과 유사하게 `inode`의 데이터 블럭 중 `offset` 바이트의 위치에 `buffer_` 공간에 들어있는 데이터를 최대 `size` 바이트만큼 복사한다. 만약 해당 Inode에 `deny_write_cnt` 플래그가 설정되어있을 경우 write를 수행하지 않고 종료한다.

```
/* Disables writes to INODE.
   May be called at most once per inode opener. */
void
inode_deny_write (struct inode *inode)
{
  inode->deny_write_cnt++;
  ASSERT (inode->deny_write_cnt <= inode->open_cnt);
}

/* Re-enables writes to INODE.
   Must be called once by each inode opener who has called
   inode_deny_write() on the inode, before closing the inode. */
void
inode_allow_write (struct inode *inode)
{
  ASSERT (inode->deny_write_cnt > 0);
  ASSERT (inode->deny_write_cnt <= inode->open_cnt);
  inode->deny_write_cnt--;
}
```

Inode의 `deny_write_cnt` 플래그를 설정하고 해제할 때 호출하는 함수들. 각 opener가 `deny_write_cnt`를 최대 한 번 증가시켰을 것을 ASSERT문을 통해 약하게 검증한다. 완전히 검증하려면 `deny_write_cnt`를 설정한 opener를 리스트화시켜 관리하는 식으로 보완할 수 있을 것이라 생각한다.

```
/* Returns INODE's inode number. */
block_sector_t
inode_get_inumber (const struct inode *inode)
{
  return inode->sector;
}

/* Returns the length, in bytes, of INODE's data. */
off_t
inode_length (const struct inode *inode)
{
  return inode->data.length;
}
```

이외에 편의성을 위해 정의된 getter 함수들.

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? */
  };
```

`file` 구조체는 OS에서 관리하는 파일들의 표현 형식으로 `filesys/file.c` 에 구현되어있다. 기본적으로 `inode` 의 wrapping class 형식을 따르고, 추가적으로 현재까지 읽었던 파일의 오프셋인 `pos` 와 수정 가능 여부에 대한 플래그인 `deny_write` 가 멤버 변수로 존재한다.

`inode` 의 `deny_write_cnt` 는 `int` 형으로 0 이상의 정수값을 가질 수 있도록 로직이 짜여진 반면 `file` 의 `deny_write` 가 `bool` 형식인 이유는 `inode` 는 여러 파일들이 동시에 가리킬 수 있기 때문에 이에 대한 중복 처리가 필요하지만 각 `file` 은 독립적으로 존재하기 때문이다.

```
/* Opens a file for the given INODE, of which it takes ownership,
   and returns the new file.  Returns a null pointer if an
   allocation fails or if INODE is null. */
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;
    }
}
```

새로운 `file` 구조체를 동적할당하여 주어진 `inode` 를 가리키도록 만들고 멤버 변수를 초기화해준다. 파라미터로 주어진 `inode` 에 대해 함수 내에서 따로 `inode_open` 이나 `inode_reopen` 을 호출해주지 않는 이유가 궁금했는데, Pintos 프로젝트 내에서 `file_open` 이 사용된 경우들을 확인하니 모두 함수 호출 이전에 `inode` 를 미리 열어주는 식으로 구현이 되어있었다.

```
/* Opens and returns a new file for the same inode as FILE.
   Returns a null pointer if unsuccessful. */
struct file *
file_reopen (struct file *file)
{
  return file_open (inode_reopen (file->inode));
}
```

해당 파일이 가리키는 Inode를 가리키는 새로운 `file` 을 생성한다. 기존 Inode는 `inode_reopen` 을 통해 opener 카운트를 올려줘야 한다.

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

열린 파일을 닫는다(삭제한다). 후술할 `file_allow_write` 함수에서 `deny_write` 플래그가 설정되었을 경우에 대한 로직을 수행한 뒤, 가리키고 있던 `inode` 를 닫고 `file` 구조체의 동적할당을 해제한다.

```
/* Reads SIZE bytes from FILE into BUFFER,
   starting at the file's current position.
   Returns the number of bytes actually read,
   which may be less than SIZE if end of file is reached.
   Advances FILE's position by the number of bytes read. */
off_t
file_read (struct file *file, void *buffer, off_t size)
{
  off_t bytes_read = inode_read_at (file->inode, buffer, size, file->pos);
  file->pos += bytes_read;
```

```
  return bytes_read;
}
```

현재까지 읽은 파일의 오프셋인 `file->pos` 부터 최대 `size` 바이트만큼의 정보를 `inode_read_at` 함수를 통해 읽어와 `buffer` 에 저장한다. 이후 `file->pos` 를 읽어온 바이트 수만큼 증가시켜준다.

```
/* Reads SIZE bytes from FILE into BUFFER,
   starting at offset FILE_OFS in the file.
   Returns the number of bytes actually read,
   which may be less than SIZE if end of file is reached.
   The file's current position is unaffected. */
off_t
file_read_at (struct file *file, void *buffer, off_t size, off_t file_ofs)
{
  return inode_read_at (file->inode, buffer, size, file_ofs);
}
```

위의 `file_read` 와 흡사하지만, 파일의 오프셋을 `file_ofs` 파라미터를 통해 임의로 지정 가능하다. `file->pos` 는 변화시키지 않는다.

```
/* Writes SIZE bytes from BUFFER into FILE,
   starting at the file's current position.
   Returns the number of bytes actually written,
   which may be less than SIZE if end of file is reached.
   (Normally we'd grow the file in that case, but file growth is
   not yet implemented.)
   Advances FILE's position by the number of bytes read. */
off_t
file_write (struct file *file, const void *buffer, off_t size)
{
  off_t bytes_written = inode_write_at (file->inode, buffer, size, file->pos);
  file->pos += bytes_written;
  return bytes_written;
}

/* Writes SIZE bytes from BUFFER into FILE,
   starting at offset FILE_OFS in the file.
   Returns the number of bytes actually written,
   which may be less than SIZE if end of file is reached.
   (Normally we'd grow the file in that case, but file growth is
   not yet implemented.)
   The file's current position is unaffected. */
off_t
file_write_at (struct file *file, const void *buffer, off_t size,
               off_t file_ofs)
{
  return inode_write_at (file->inode, buffer, size, file_ofs);
}
```

`file_read` , `file_read_at` 과 동일하게 각각 현재 파일의 오프셋과 명시된 오프셋 `file_ofs` 에서 시작하여, `buffer` 에 있는 내용을 최대 `size` 바이트만큼 파일에 작성한다.

```
/* Prevents write operations on FILE's underlying inode
   until file_allow_write() is called or FILE is closed. */
void
file_deny_write (struct file *file)
{
  ASSERT (file != NULL);
  if (!file->deny_write)
    {
      file->deny_write = true;
      inode_deny_write (file->inode);
    }
}
```

현재 파일에 이미 `deny_write` 플래그가 설정되어있지 않다면 해당 플래그를 설정하고, `inode_deny_write` 함수를 호출하여 inode의 `deny_write` 카운트 역시 올려준다.

```
/* Re-enables write operations on FILE's underlying inode.
   (Writes might still be denied by some other file that has the
   same inode open.) */
void
```

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

위의 `file_deny_write`와 정확히 반대의 동작을 수행한다.

```
/* Sets the current position in FILE to NEW_POS bytes from the
   start of the file. */
void
file_seek (struct file *file, off_t new_pos)
{
  ASSERT (file != NULL);
  ASSERT (new_pos >= 0);
  file->pos = new_pos;
}
```

현재 파일이 가리키고 있는 오프셋인 `file->pos`를 명시적으로 변경한다.

```
/* Returns the size of FILE in bytes. */
off_t
file_length (struct file *file)

/* Returns the current position in FILE as a byte offset from the
   start of the file. */
off_t
file_tell (struct file *file)
```

이외에 편의성을 위해 정의된 getter 함수들.

Filesys

```
/* Partition that contains the file system. */
struct block *fs_device;
```

전역으로 사용되는 파일 시스템 구조체의 포인터. `BLOCK_FILESYS` 속성을 가진 하나의 블럭으로 정의된다.

```
/* Initializes the file system module.
   If FORMAT is true, reformats the file system. */
void
filesys_init (bool format)
{
  fs_device = block_get_role (BLOCK_FILESYS);
  if (fs_device == NULL)
    PANIC ("No file system device found, can't initialize file system.");

  inode_init ();
  free_map_init ();

  if (format)
    do_format ();

  free_map_open ();
}
```

`BLOCK_FILESYS` 속성을 가지는 블럭의 주솟값을 받아온 뒤, inode와 free map을 초기화시켜주는 등의 초기화 작업을 진행한다.

```
/* Creates a file named NAME with the given INITIAL_SIZE.
   Returns true if successful, false otherwise.
   Fails if a file named NAME already exists,
   or if internal memory allocation fails. */
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_size)
                  && dir_add (dir, name, inode_sector));
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);

  return success;
}
```

파라미터로 주어진 이름과 크기의 파일을 루트 디렉토리 아래에 생성한다.

- 새로운 Inode를 생성할 블럭 하나를 Free map에서 할당받은 뒤 `inode_create`를 통해 해당 블럭에서 Inode를 생성한다.
- `dir_add`를 호출하여 루트 디렉토리의 Inode 데이터에 엔트리를 추가한다.

> ```
> /* A directory. */
> struct dir
>   {
>     struct inode *inode;                /* Backing store. */
>     off_t pos;                          /* Current position. */
>   };
> ```
>
> `directory`는 운영체제에서 파일 디렉토리를 나타내는 자료형이고, `file`과 유사하게 정보를 담고 있는 Inode와 현재까지 읽은 데이터의 오프셋으로 구성된다.
>
> 해당 Inode에는 현재 디렉토리 내부에 들어있는 파일들의 Inode 블럭 인덱스, 이름 문자열, 현재 사용중 여부가 `dir_entry` 구조체에 묶여 순차적으로 저장되고, 이를 탐색하며 디렉토리 내 파일의 존재 여부 등을 해석한다.
>
> Project 2에서는 subdirectory 없이 모든 파일이 루트 디렉토리 바로 아래에 존재한다고 가정하여 별다른 수정이 필요 없지만, 추후 파일명을 파싱하여 대응되는 subdirectory로 이동하는 방식으로 개선이 필요할 것으로 보인다.

```
/* Opens the file with the given NAME.
   Returns the new file if successful or a null pointer
   otherwise.
   Fails if no file named NAME exists,
   or if an internal memory allocation fails. */
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);
}
```

루트 디렉토리에 있는 파일 중 이름이 `name`인 파일을 찾아 연다.

```
/* Deletes the file named NAME.
   Returns true if successful, false on failure.
   Fails if no file named NAME exists,
   or if an internal memory allocation fails. */
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;
}
```

루트 디렉토리 아래에 있는 파일 중 이름이 `name`인 파일을 삭제한다. 메모리 영역을 초기화하는 대신 해당 `dir_entry`의 `in_use` 플래그를 해제하고 가리키던 파일의 Inode를 삭제한다.

```
/* Formats the file system. */
static void
do_format (void)
{
  printf ("Formatting file system...");
  free_map_create ();
  if (!dir_create (ROOT_DIR_SECTOR, 16))
    PANIC ("root directory creation failed");
  free_map_close ();
  printf ("done.\n");
}
```

앞서 설명한 `filesys_init`에서 `format` 플래그가 설정되어있을 시 실행되는 함수. 새로운 free map을 생성한 뒤 1번 블럭에 16개의 파일 정보를 담을 수 있는 루트 디렉토리를 생성한다.

Init

`paging_init(void)`

```
static void
paging_init (void)
{
  uint32_t *pd, *pt;
  size_t page;
  extern char _start, _end_kernel_text;

  pd = init_page_dir = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  pt = NULL;
  for (page = 0; page < init_ram_pages; page++)
    {
      uintptr_t paddr = page * PGSIZE;
      char *vaddr = ptov (paddr);
      size_t pde_idx = pd_no (vaddr);
      size_t pte_idx = pt_no (vaddr);
      bool in_kernel_text = &_start <= vaddr && vaddr < &_end_kernel_text;

      if (pd[pde_idx] == 0)
        {
          pt = palloc_get_page (PAL_ASSERT | PAL_ZERO);
          pd[pde_idx] = pde_create (pt);
        }

      pt[pte_idx] = pte_create_kernel (vaddr, !in_kernel_text);
    }

  /* Store the physical address of the page directory into CR3
     aka PDBR (page directory base register).  This activates our
     new page tables immediately.  See [IA32-v2a] "MOV--Move
     to/from Control Registers" and [IA32-v3a] 3.7.5 "Base Address
     of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (init_page_dir)));
}
```

> 페이지 디렉토리, 페이지 테이블을 가상 메모리와 매핑하고 Page Directory와 Page Table에 free한 페이지를 allocate해 초기화하는 함수

`palloc_get_page`를 이용해 `pd`에 자유로운 페이지를 allocate한다. 해당 함수는 kernel의 init 중 일부로 매우 중요한 작업이기에 page 할당 중 실패하였을 때 패닉한다.
page의 개수만큼 순회하며 각 페이지의 물리적 주소에 대응되는 virtual address를 구하고 이에 해당하는 페이지 디렉토리 엔트리 인덱스(`pde_idx`), 페이지 테이블 엔트리 인덱스(`pte_idx`)을 구한다. 또한 페이지 테이블을 위한 페이지를 할당해주기도 한다.
마지막으로 CR3로 알려진 pbdr 레지스터에 page diretory의 물리 주소를 저장한다.

`read_command_line(void)`

```
static char **
read_command_line (void)
{
  static char *argv[LOADER_ARGS_LEN / 2 + 1];
  char *p, *end;
  int argc;
  int i;

  argc = *(uint32_t *) ptov (LOADER_ARG_CNT);
```

```
  p = ptov (LOADER_ARGS);
  end = p + LOADER_ARGS_LEN;
```

> command arguments를 단어 단위로 잘라 그 문자열의 주소들을 리스트를 반환하는 함수

`argc`에 physical 주소 `LOADER_ARG_CNT`에 저장되어 있는 argument 개수를 저장하고 `p`에는 argument들이 저장되어 있는 physical 주소 `LOADER_ARGS`의 가상 주소를 저장한다.

```
  for (i = 0; i < argc; i++)
    {
      if (p >= end)
        PANIC ("command line arguments overflow");

      argv[i] = p;
      p += strnlen (p, end - p) + 1;
    }
  argv[argc] = NULL;

  /* Print kernel command line. */
  printf ("Kernel command line:");
  for (i = 0; i < argc; i++)
    if (strchr (argv[i], ' ') == NULL)
      printf (" %s", argv[i]);
    else
      printf (" '%s'", argv[i]);
  printf ("\n");

  return argv;
}
```

`p`에서부터 시작해 `\0`를 기준으로 문자열을 나누어 생각해 `0`로 구분된 각 문자열의 시작 주소를 `argv`(list 처럼 작동)에 차례대로 저장한다. 조회 중인 주소가 최대 수치의 주소(`end`)를 넘어서면 패닉을 일으킨다.
`argv[argc]`(끝 부분)에 `NULL`을 집어넣어 argument list의 끝을 표시한다.
그리고 커널에 집어넣은 커맨드를 출력하고 args가 단어 단위로 저장된 `argv`를 반환한다.

`parse_options(char **argv)`

```
static char **
parse_options (char **argv)
{
  for (; *argv != NULL && **argv == '-'; argv++)
    {
      char *save_ptr;
      char *name = strtok_r (*argv, "=", &save_ptr);
      char *value = strtok_r (NULL, "", &save_ptr);

      if (!strcmp (name, "-h"))
        usage ();
      else if (!strcmp (name, "-q"))
        shutdown_configure (SHUTDOWN_POWER_OFF);
      else if (!strcmp (name, "-r"))
        shutdown_configure (SHUTDOWN_REBOOT);
#ifdef FILESYS
      else if (!strcmp (name, "-f"))
        format_filesys = true;
      else if (!strcmp (name, "-filesys"))
        filesys_bdev_name = value;
      else if (!strcmp (name, "-scratch"))
        scratch_bdev_name = value;
#ifdef VM
      else if (!strcmp (name, "-swap"))
        swap_bdev_name = value;
#endif
#endif
      else if (!strcmp (name, "-rs"))
        random_init (atoi (value));
      else if (!strcmp (name, "-mlfqs"))
        thread_mlfqs = true;
#ifdef USERPROG
      else if (!strcmp (name, "-ul"))
        user_page_limit = atoi (value);
#endif
      else
```

```
      PANIC ("unknown option `%s' (use -h for help)", name);
    }

  /* Initialize the random number generator based on the system
     time.  This has no effect if an "-rs" option was specified.

     When running under Bochs, this is not enough by itself to
     get a good seed value, because the pintos script sets the
     initial time to a predictable value, not to the local time,
     for reproducibility.  To fix this, give the "-r" option to
     the pintos script to request real-time execution. */
  random_init (rtc_get_time ());

  return argv;
}
```

> argument의 주소들이 담긴 주소를 받아 각 option에 맞는 행위를 수행한 뒤 option이 아닌 첫번째 argument를 반환하는 함수

매개변수로 입력 받은 argument의 주소들이 담긴 list를 주소 값이 null이 아니거나 '-' 값으로 시작하는 동안 순회하며 확인한다. 순회한 주소의 문자열을 확인하여 '-h', '-q', '-r' 등에 일치하는지 확인하고 그에 맞는 작업을 수행하여 kernel의 설정을 변경한다. 이에는 파일시스템 초기화, user pool의 페이지 limit 등에 대한 옵션이 포함되어 있다. 그리고 '-'값으로 시작하지 않는 문자열의 주소가 담긴 주소(입력 매개변수 list element 주소 중 하나)를 반환한다. 해당 주소는 kernel이 실행할 명령어를 담고 있다.

`run_task(char **argv)`

```
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);
}
```

> 매개변수로 받은 `argv`의 `argv[1]`가 가리키는 명령어를 `run_test` 또는 `process_wait`를 통해 수행하는 함수

매개변수로 받은 `char **argv`의 `argv[1]`가 가리키는 명령어를 `process_wait`를 통해 수행한다. `process_execute`를 통해 `task`라는 이름의 파일을 로드하여 user program을 실행하는 스레드를 생성한다. `process_wait`을 통해 해당 스레드가 끝날 때까지 기다리게 구현해야하지만 현재 구현상 바로 -1을 반환하고 리턴한다. 즉 정상적으로 user program을 수행할 수 없다.

`run_actions(char **argv)`

```
static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, including action name. */
      void (*function) (char **argv);   /* Function to execute action. */
    };

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

> 매개변수로 입력 받은 argv를 순회하며 argv 내 명시된 action들을 순차적으로 수행하는 함수

`action` 구조체는 수행 가능한 액션의 이름, 필요한 argument 개수, 해당 액션을 수행하기 위한 function을 저장한다.
`actions`은 수행 가능한 action에 대응대는 `action` list로 현재로서는 `run`과 예외 처리를 위한 값인 `NULL`이 있다.

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

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

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

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

}
```

매개변수로 입력받은 `argv`를 순회하며 주소가 가리키는 문자열과 `actions`의 아이템과 이름이 일치하는 것이 있는지 확인 후 있다면 해당 `action`에 명시된 argument 개수만큼 제공하였는지 확인한다. 개수가 동일하다면 함수에 해당 action에 해당되는 문자열 주소를 가리키는 argv 내 값의 주소를 해당 `action`에 해당되는 함수 `action->function`의 매개변수로 두어 해당 함수를 호출해 action을 수행한다. `argv`를 순회하며 NULL 값을 만날 때까지 이를 반복한다.
만약 `actions`에 없는 action을 요구하거나 `action`에 대한 args가 부족할 때는 패닉을 일으킨다.

초기설정 in `main in threads/init.c`

많은 초기 설정이 있지만 이는 Assn1에서 깊이 다루었기에 이번 과제(User program)와 연관된 초기화에 대한 설명만 첨부하도록 하겠다.

```
int
main (void)
{
  char **argv;
  ...
  /* Initialize memory system. */
  palloc_init (user_page_limit);
  malloc_init ();
  paging_init ();

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

`user_page_limit`은 pintos command에서 `-ul`과 함께 입력된 수치 또는 기본 값 `SIZE_MAX`로 user pool의 page 개수 한계이다. `palloc_init`을 호출할 때 넘겨주어 user pool에 최대로 가질 수 있는 page 개수를 정해준다.
`paging_init()`을 통해 cpu에 새 page directory를 세팅하고 Page Directory와 Page Table을 초기화하고 pbdr 레지스터도 초기화해준다.

`threads/init.c`의 `main()`에서 kernel을 초기화할 때 `tss_init()`을 호출해 Kernel TSS, 전역변수 `tss`를 초기화하여 User mode에서 interrupt가 발생하여 stack을 전환할 때 TSS를 사용할 수 있도록 한다.
`gdt_init()` 또한 호출하여 핵심적인 6가지 segment에 대한 descriptor를 포함하는 GDT(Global Descriptor Table)을 초기화한다.

```
  intr_init ();
```

`intr_init`을 호출하여 `interrupt` 시스템을 초기화한다. 이는 exception 발생에 따라 handler를 호출하게 한다.

```
  exception_init ();
  syscall_init ();
```

이후에는 `exception_init()` 을 호출해 page fault를 포함해 각종 exception에 대한 핸들러 함수를 등록하여 exception에 대한 처리를 할 수 있도록 한다. 또한 `syscall_init()` 을 호출하여 system call interrupt에 대한 핸들러 함수를 등록해 system call에 대한 처리를 할 수 있도록 한다.

```
#ifdef FILESYS
  /* Initialize file system. */
  ide_init ();
  locate_block_devices ();
  filesys_init (format_filesys);
```

`format_filesys` 는 pintos command 에 `-f` 포함될 시 true가 되는 값으로 file system을 format 여부를 담는다. `filesys_init` 에 함께 넘겨줘 호출하여 file system을 초기화한다.

```
  run_actions (argv);
```

이렇게 초기화가 완료되면 `run_actions` 를 통해 `run_task` 를 호출해 `process_execute` 를 통해 userprogram을 로드하고 수행하는 스레드를 생성하고 이를 `process_wait` 으로 기다리길 기대하지만 현재 구현상 `process_wait` 을 바로 리턴하여 userprogram을 정상적으로 수행되지 않는다.

Design

Process Termination Messages

후술할 시스템 콜 관련 구현에서 `exit` 이 호출되거나 다른 시스템 콜 처리 중 예외 상황이 발생했을 때 스레드를 종료시킬 수 있는 함수를 만들어 정상 종료와 예외 상황을 일괄적으로 처리할 수 있게 구현한다. 프로세스 종료 메세지은 해당 함수의 실행 도중 출력하면 되는데, 프로세스 이름은 `thread_current()->name` 으로 불러올 수 있고 exit code는 함수 인자로 전달될 예정이다.

Argument Passing

현재 user program을 실행하는 구현은 다음과 같다.

- command line에서 parsing해 얻은 `argv[1]` 을 `file_name` 으로 하여 `process_execute(file_name)` 에서 `start_process(file_name_)` 을 수행하는 스레드를 생성한다.
- `start_process` 에서는 전해 받은 `file_name_` 을 이름으로 하는 실행 파일을 load하고 해당 프로그램의 시작 위치로 가 프로그램을 실행한다.
- `process_execute` 에 실행해야 하는 파일 명과 argument를 포함한 `file_name` 을 넘긴다.
  - 다만 초기 구현에선 `process_execute` 와 `start_process` 에서 파일 명과 argument가 모두 포함된 파싱되지 않은 `file_name` 을 가지고 처리하고 있고, 또한 load한 프로그램에게 argument를 passing해주는 로직도 포함되어 있지 않다.

그렇기에 `process_execute` 또는 `start_process` 에서 `file_name` 을 parsing하여 파일 이름과 argument로 분리하는 로직을 추가해야 한다. 또한 파일 이름을 가지고 executable을 로드 완료한 이후에 parsing해 얻은 argument 및 관련 정보를 executable을 로드한 스레드의 스택에 컨벤션에 맞추어 집어 넣어 user program이 해당 argument를 사용할 수 있도록 해야 한다.

자세한 구현 계획은 아래와 같다.

- `process_execute(file_name)` 에서 `strtok_r` 을 이용하여 `file_name` 을 `' '` 을 기준으로 나누어 맨 앞을 `thread_create` 의 스레드 이름으로 사용한다.
  - argument와 파일 이름을 모두 포함한 스레드 이름을 가지는 것이 아닌 파일 이름(실행 프로그램 이름)을 스레드 이름으로 가지게 하기 위함이다.
- `start_process` 에서 매개변수로 받은 `file_name_` 을 `strtok_r` 을 이용하여 `' '` 을 기준으로 나누어 맨 앞을 `load` 함수 호출 시 파일이름으로 사용한다.
- `start_process` 에서 load한 직후, parsing한 `file_name_` 의 뒷 부분을 차례로 순회하며 `if_.esp` 로부터 시작해 문자열을 복사해 집어 넣는다. (Ex. `bar -> f00 -> -l -> /bin/ls\0` 순)
  - 각 argument의 끝은 `\0` 존재. 즉 생각하는 길이보다 1 더 길다.
- 앞서 argument로 다양한 문자열을 집어넣었기에 스택에 패딩을 집어넣어 주소가 4-byte allign 될 수 있도록 한다.
- 앞서 집어넣은 argument 문자열의 시작 주소를 집어 넣은 순서대로 차례로 stack에 넣는다.
- stack에 넣은 argument의 개수(`argc`)를 스택에 넣은 뒤 마지막 return address에 대응되는 NULL 값을 스택에 넣은 뒤 esp를 올바르게 설정한다.

아래의 표는 `/bin/ls -l foo bar` 커맨드를 실행하였을 때 argument를 올바르게 패싱한 스레드 스택의 모습이다.

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 *`

Address	Name	Data	Type
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 (*) ()`

- 위 구현에 문자열을 보관, 유지하기 위해 page를 allocation해야 할 필요가 있을 수 있다.

System Call

현재 구현에서 system call 즉 0x30 interrupt에 대한 handler function은 `syscall_handler`로 어떤 system call인지 구분하지 않고 메세지를 출력하고 `thread_exit`을 통해 스레드를 종료한다.

이를 system call 에 따라 각기 다른 기능을 수행하도록 변경해 구현해야 한다.

System Call Handler

- `syscall_handler`에서 인수로 받은 `intr_frame *f`의 `esp`를 조회하여 32비트 system call number를 얻는다.
  - 해당 esp가 올바른 주소인지 검증하기 위해
  - 또한 system call에 대한 argument는 system call number 다음 주소에 32비트 간격으로 존재한다.
- 위에서 얻은 system call number를 바탕으로 여러 system call 함수 중 어떤 함수를 실행할지 결정하고 호출해 실행한다.
  - 이 때 함수 호출시 현재 esp 다음 주소를 매개변수로 넘겨준다. 이는 각 system call 함수에서 사용해야 할 argument를 포함할 것이다.
  - 주소로부터 정해진 갯수(각 함수의 argument 개수)의 32비트를 얻을 수 있는 util 함수를 추가한다.
- 함수 호출의 결과를 `intr_frame`의 `eax`에 대입하여 system call의 결과 값을 리턴할 수 있도록 한다.

Process Control Block(PCB)

process에 대한 정보를 담은 Process Control Block struct인 `struct pcb`를 `userprog/process.h`에 선언한다.
`struct pcb`

- process identifier인 `pid`
- exitcode `exit_code`
- `wait`, `exit`에서 사용하는 exit code에 대한 semaphore `exit_code_sema`
- 해당 프로세스가 가진 file descriptor의 table인 `fdt`
- 자식 프로세스 list인 `children_list`
- 위 list의 자식 프로세스 list elem인 `child_list_elem`
- exec 완료를 관리하는 semaphore `load_sema`

userprog를 실행하는 스레드들은 자신을 포함하는 process를 가지게 되고 이를 표현하기 위해 `thread` 구조체에 멤버 변수에 `pcb`에 대한 포인터를 추가해준다.

`create_tid`와 유사하게 unique한 pid를 생성하는 `create_pid` 함수를 추가해주어야 한다.
`pcb` 구조체 정보를 초기화하는 `pcb_init` 함수를 추가한다.

- `wait`에 의해 스레드보다 `pcb`가 더 오랫동안 살아있을 수 있기에 `palloc_get_page` 등을 이용해 `pcb`를 위한 공간을 별도로 할당 받아야 한다.
- `create_pid`를 통한 pid 설정, `fdt`, `children_list`, 등의 초기화를 진행한다.
- `load_sema`의 값은 0으로 하여 초기화한다.
  - `start_process`에서 `load` 직후 `load_sema`를 up해준다. 이는 load 완료를 나타내기 위함이다.

User Process Manipulation

`void halt(void)`

- pintos를 종료하는 함수이다.
- `shutdown_power_off()`를 호출하여 종료한다.

`pid_t exec(const char *cmd_line)`

주어진 cmd를 수행하는 자식 프로세스를 생성하는 시스템 콜 함수이다.
`load_sema`를 이용해 자식 프로세스의 `load` 완료시점까지 parent process가 syscall을 리턴하지 않게 한다.

- `pcb_init`을 호출하여 새로운 `pcb`를 만들어 초기화한 뒤 `children_list`에 해당 pcb를 추가한다.
- `cmd_line`를 인수로 하여 `process_execute`를 호출한다.
- `pcb`의 `load_sema`를 `down`한다.

    - child process가 로드 완료되면 `up` 되며 통과할 수 있게 된다.
- `pcb` 의 `pid` 를 반환한다.

`void exit(int status)`

- `pcb` 의 `exit_code` 에 `status` 를 넣는다. `exit_code_sema` 를 `up` 한다.
- `process_exit` 을 이용해 process 자원을 모두 반납한다.

`int wait(pid_t pid)`

`pid` 프로세스가 exit할 때 까지 기다리는 시스템 콜 함수이다.
`exit_code_sema` 를 통해 자식 프로세스가 exit할 때까지 wait를 리턴하지 않고 기다리다 exit 하면 해당 프로세스(이미 해제됨)의 `pcb` 를 할당 해제하고 `exit_code` 를 리턴한다.

- 자식 프로세스 중, 즉 `children_list` 를 순회하며 `pcb.pid` 가 `pid` 인 `pcb` 를 찾는다.
- 찾은 `pcb` 의 `exit_code_sema` 를 `down` 한다.
    - 해당 프로세스가 exit()하면 `up` 되어 진행할 수 있다.
- `pcb.exit_code` 를 `exit_code` 임시 변수에 저장한다.
- `pcb` 의 페이지를 해제하고 `exit_code` 를 반환한다.

File Descriptor (FD)

UNIX 기반 운영체제에서 각 프로세스는 현재 open한 파일들에 접근하기 위해 각 파일마다 특정한 정수 키(key)를 대응시키는데, 이를 File Descriptor(FD)라고 한다. 일반적으로 시스템상 사전 정의된 일부 FD들을 제외하면 나머지 FD들은 프로세스마다 독립적으로 관리되기 때문에, PCB 위의 고정된 길이의 배열 (File Descriptor Table) 형태로 선언되어 관리한다. File Descriptor Table의 각 엔트리는 다음과 같이 구상해볼 수 있다.

```
struct fdt_entry
  {
    struct file *file;
    bool in_use;
  };
```

`*file` 은 해당 인덱스의 엔트리가 가리키는 파일, `in_use` 는 해당 fd가 현재 할당된 상태인지를 나타낸다.

이를 이용하여 File Descriptor Table은 다음과 같이 `fdt_entry` 의 정적 배열로 나타낼 수 있다.

```
struct fdt_entry fd_table[OPEN_MAX];
```

FD를 해당 테이블의 인덱스로 사용하여 접근함으로써 대응하는 파일에 접근할 수 있다.

- `OPEN_MAX` 는 한 프로세스가 동시에 열 수 있는 최대 파일 개수로, 운영체제 내에서 전처리된 상수이다.

create

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

명시된 이름(`file`)과 크기(`initial_size`)를 가진 파일을 생성한 뒤 성공 여부를 반환하는 시스템 콜.

- `filesys_create` 와 동일한 동작이므로 이를 호출한다.

remove

```
bool remove (const char *file)
```

명시된 이름(`file`)을 가진 파일을 찾아 삭제한 뒤 성공 여부를 반환하는 시스템 콜.

- `filesys_remove` 와 동일한 동작이므로 이를 호출한다.

open

```
int open (const char *file)
```

명시된 이름(`file`)을 가진 파일을 찾아 오픈한 뒤 해당 파일에 부여된 FD를 반환하는 시스템 콜.

- `filesys_open` 을 호출하여 해당하는 파일을 찾는다.
- 현재 프로세스의 File Descriptor Table를 인덱스 0부터 순회한다.
- 사용 가능한 (`in_use == false`) 첫 엔트리를 발견할 시 엔트리에 파일 관련 정보를 할당한 후 해당 인덱스를 FD로써 반환한다.
- 한 프로세스 내에서 동일한 파일이 여러번 열리더라도 매번 새로운 FD를 할당하여 반환해야 한다.

- 실패 시 `-1`을 반환한다.

filesize

```
int filesize (int fd)
```

주어진 FD에 해당하는 파일의 크기를 바이트 단위로 반환하는 시스템 콜.

- FD Table에서 파일 포인터를 참조한 후 `file_length` 함수의 반환값을 반환한다.
- 실패 시 에러 코드와 함께 프로세스를 종료한다.

read

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

주어진 FD에 해당하는 파일에서 최대 `size` 바이트 만큼의 데이터를 읽어 `buffer`에 복사하는 시스템 콜. 실제로 파일에서 복사된 바이트 수를 반환한다.

- FD Table에서 파일 포인터를 참조한 후 `file_read`를 호출한다.
- 실패 시 `-1`을 반환한다.
- `fd == 0`일 경우 사용자 입력에서 정보를 읽어온다는 뜻이므로 `input_getc`를 통해 키보드 입력을 받는다.
- 악의적인 호출에 대비하여 `buffer`의 주소가 안전한지 검사한다.

write

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

`buffer` 위에 존재하는 데이터를 주어진 FD에 해당하는 파일에 최대 `size` 바이트 만큼 복사하는 시스템 콜. 실제로 파일에 복사된 바이트 수를 반환한다.

- 현재 프로젝트에선 각 파일의 크기가 고정되어있기 때문에, 복사할 데이터의 크기가 파일의 끝을 넘어설 경우 파일의 끝까지만 복사한 뒤 복사된 바이트 수를 반환한다.
- FD Table에서 파일 포인터를 참조한 후 `file_write`를 호출한다.
- 실패 시 에러 코드와 함께 프로세스를 종료한다.
- `fd == 1`의 경우 `putbuf` 함수를 이용해 콘솔에다 `buffer`의 데이터를 출력한다.
- 악의적인 호출에 대비하여 `buffer`의 주소가 안전한지 검사한다.

seek

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

주어진 FD에 해당하는 파일이 현재까지 읽은 데이터의 오프셋을 `position`으로 변경하는 시스템 콜.

- FD Table에서 파일 포인터를 참조한 후 `file_seek`를 호출한다.
- 새 `position`이 기존 파일 길이를 넘어서는 경우에도 정상 동작으로 간주한다.
- 실패 시 에러 코드와 함께 프로세스를 종료한다.

tell

```
unsigned tell (int fd)
```

주어진 FD에 해당하는 파일이 현재까지 읽은 데이터의 오프셋을 반환하는 시스템 콜.

- FD Table에서 파일 포인터를 참조한 후 `file_tell`을 호출한다.
- 실패 시 에러 코드와 함께 프로세스를 종료한다.

close

```
void close (int fd)
```

주어진 FD에 해당하는 파일을 찾아 닫아주는 시스템 콜.

- FD Table에서 파일 포인터를 참조한 후 `file_close`를 호출한 뒤, 해당 테이블 엔트리의 `in_use` 플래그를 `false`로 바꾼다.
- 실패 시 에러 코드와 함께 프로세스를 종료한다.

위의 File Manipulation 시스템 콜들은 동시에 여러 프로세스가 동시에 실행할 시 Race Condition이 발생할 우려가 있으므로 Lock 등의 Synchronization 수단을 통해 동시성을 확보해줘야 한다.

시스템 콜 도중 예외 상황 시 동작은 우선 Pintos 문서에 명시되어있는 동작을 따르되, 별다른 명시가 없을 경우 프로세스를 종료하는 방식을 따른다.

Denying Writes to Executables

디스크 상에 존재하는 유저 프로그램 역시 하나의 파일이기 때문에 동시에 여러 프로세스가 접근하여 Read, Write, Execute하는 것이 가능하다. 하지만 어떤 프로세스가 유저 프로그램을 실행하는 중 다른 프로세스가 해당 프로그램 파일에 Write 하는 등의 수정이 일어날 경우 정상적인 동작이 불가능할 것이다. 때문에 어떤 프로세스가 파일을 실행하고 있을 경우 앞서 살펴본 `file_deny_write`를 호출하여 다른 프로세스가 파일을 변경하는 것을 막고, 실행을 마칠 시 `file_allow_write`를 호출하여 제한을 해제해주는 과정이 수반되어야 한다.

프로세스가 실행 가능한 ELF 파일을 읽어와 스택에 올리는 과정은 `userprog/process.c`의 `load` 함수에서 일어난다. 이때 대상 파일은 `filesys_open(file_name)`으로 명시가 되어있으므로, 파일을 성공적으로 불러온 후 해당 파일에 대해 `file_deny_write`를 호출함으로써 변경 제한을 설정하는 기능은 쉽게 구현이 가능하다.

하지만 프로세스가 종료될 때인 `process_exit`에서는 해당 프로세스가 현재 실행중이었던 파일이 무엇이었는지 알 수 없어 `file_allow_write`를 호출할 수 없다. 이를 위해 `thread` 구조체에 현재 실행중인 파일에 대한 정보를 담고 있는 멤버 변수 `file_exec`을 추가하여 `load` 중 현재 프로세스가 해당 파일을 실행 중임을 명시해주고, `process_exit`에서 어떤 파일을 실행중이었는지를 `file_exec`을 참조하여 알아낸 뒤 `file_allow_write`를 호출해줌으로써 변경 제한을 해제하는 기능을 구현할 예정이다.