OS Project3 Final Report

letsgo_seuljongbin
20180038 박형규, 20180480 성창환

1. Introduction

OS Project3을 통해 Frame Table, Lazy Loading, Supplemental Page Table, Stack Growth, File Memory Mapping, Swap Table, On Process Termination을 구현하였다. Naïve Pintos에 Virtual Memory와 관련된 것이 아무것도 구현되어 있지 않아서 VM과 관련된 모든 내용을 src/vm의 directory에 page.c/h, frame.c/h, swap.c/h를 통해 구현하였다.

2. Frame Table

2.1. Flow Chart

2.2. Added or modified Function & Data Structure

- Added Function
  Frame_table_init(), frame_get_next(), frame_find_free_and_lock(), frame_free_and_unlock(), page_make_new()
- Modified Function
  syscall_handler(), load_segment()
- Added Data Structure
  Struct frame

2.3. Implementation

아래는 frame table을 구성하는 구조체이다.

```
struct frame{
    void *base_addr;
    struct page *page;
    struct lock lock;
    struct list_elem elem;
};
```

Base_addr은 frame table의 base address를 나타내고, page는 frame이 할당된 page를 pointing 한다. Lock은 frame의 synchronization을 위한 lock이다.

먼저 우리가 implement하려고 했던 frame table을 initialized 하는 function인 frame_table_init() 이다.

```
/* initialize frame table */
void frame_table_init(void){
    void *base_ptr;
    struct frame *temp;

    list_init(&frame_list);
    lock_init(&frame_sys_lock);

    while ((base_ptr = palloc_get_page(PAL_USER)) != NULL){
        temp = malloc(sizeof(struct frame));
        temp->base_addr = base_ptr;
        temp->page = NULL;
        lock_init(&temp->lock);
        list_push_back(&frame_list, &temp->elem);
    }
    frame_pos = list_front(&frame_list);
}
```

List_init()과 lock_init()을 통해 frame들의 list를 관리하기 위한 전역변수인 frame_list와 frame 전체 system을 관리하기 위한 frame_sys_lock을 initialized해준다. 이후 palloc_get_page() 를 통해 single free page를 할당받고 return된 virtual address를 base_ptr에 저장한 뒤, 차례로 frame을 할당해주고 frame을 이루는 성분들을 initialize해준다. 그 후 clock algorithm을 위한 frame_pos를 frame_list의 front로 설정해준다.

Frame_find_free_and_lock() 함수는 free page를 찾고 lock을 걸어주는 함수이다. 먼저 아래와 같이 for문을 통해 free_page를 찾는 작업을 진행한다.

```c
/* find or make free page and lock */
/* just provide free frame not install_page */
struct frame *frame_find_free_and_lock(struct page *page){
    ASSERT(page != NULL);

    struct list_elem *e;
    struct frame *f;
    bool result;

    lock_acquire(&frame_sys_lock);

    // find free page
    for (e = list_begin(&frame_list); e != list_end(&frame_list); e = list_next(e)){
        f = list_entry(e, struct frame, elem);
        if (lock_held_by_current_thread(&f->lock)) continue;
        if (!lock_try_acquire(&f->lock)) continue;
        // if find free page, allocate and return
        if (f->page == NULL){
            // page에서 page_number를 구해서 temp의 page_number에 저장한다.
            f->page = page;
            ASSERT(lock_held_by_current_thread(&f->lock));
            lock_release(&frame_sys_lock);
            return f;
        }
        lock_release(&f->lock);
    }
```

반복문을 통해 free page를 찾고 만약 free page를 찾게 된다면 그 page는 NULL일 것이다. 따라서 이 조건을 만족하면 page를 할당하고 return한다. 만약 for문이 종료될 때 까지 free page를 찾지 못한다면 현재 할당되어있는 page 중에서 clock algorithm을 통해 victim을 선정하고 swap out을 통해 free frame을 확보하고 그 free frame에 page를 할당한다. 아래는 이 기능이 구현되어있는 코드이다.

```c
    // if fail to find free frame, then choose victim
    // 희생자 선정하고, swap out시켜서 free frame을 확보하고, 그리고 거기에 page allocation.
    // not yet
    f = frame_get_next();
    pagedir_set_accessed(thread_current()->pagedir, f->base_addr, false);

    while (true) {
        if (!pagedir_is_accessed(thread_current()->pagedir, f->base_addr)){

            result = swap_out(f->page);
            if (!result) return NULL;

            lock_release(&f->lock); // lock이 이미 다른 process에 들려있었으므로 풀어주고
            pagedir_clear_page(f->page->t->pagedir, f->page->page_addr);
            f->page->frame = NULL; // page - frame 매칭 해제

            f->page = page; // new match
            lock_acquire(&f->lock);
            ASSERT(lock_held_by_current_thread(&f->lock));
            lock_release(&frame_sys_lock);
            return f;
        }
        f = frame_get_next();
    }
}
```

아래는 clock algorithm을 구현할 때 next frame을 받아오는 함수인 frame_get_next() 함수이다.

```
/* for clock algorithm */
static struct frame *frame_get_next(void){
    struct list_elem *next;
    if (frame_pos == list_back(&frame_list)) next = list_front(&frame_list);
    else next = list_next(frame_pos);
    frame_pos = next;
    return list_entry(frame_pos, struct frame, elem);
}
```

마지막으로 frame을 free시키고 unlock하는 함수인 frame_free_and_unlock() 함수이다.

```
/* clear and unlock */
void frame_free_and_unlock(struct frame *f){
    ASSERT(lock_held_by_current_thread(&f->lock));

    lock_acquire(&frame_sys_lock);

    lock_release(&f->lock);
    pagedir_clear_page(f->page->t->pagedir, f->page->page_addr);
    f->page->frame = NULL;
    f->page = NULL;

    lock_release(&frame_sys_lock);
}
```

다음으로 userprog/process.c에서 load_segment()함수를 수정하였다. Segment를 로딩할 때 우리는 이제 pintos를 page frame기반 구현으로 수정하였으므로 이에 알맞게 segment가 로딩되도록 하였다. 아래는 이를 구현한 코드이다.

```
// pj3
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);

    while (read_bytes > 0 || zero_bytes > 0)
    {
        size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
        size_t page_zero_bytes = PGSIZE - page_read_bytes;

        struct page *p = page_make_new(upage, !writable);
        if (p == NULL) return false;

        if (page_read_bytes > 0){
            p->file = file;
            p->file_offset = ofs;
            p->file_length = page_read_bytes;
        }

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

Page가 읽을 bytes를 page_read_bytes variable에 저장해두고, 나머지 0으로 채울 부분을 page_zero_bytes 부분에 저장해 두었다. 이후, page_make_new 함수를 통해 새로운 page를 생성하였다. Page_make_new의 구현은 아래와 같다.

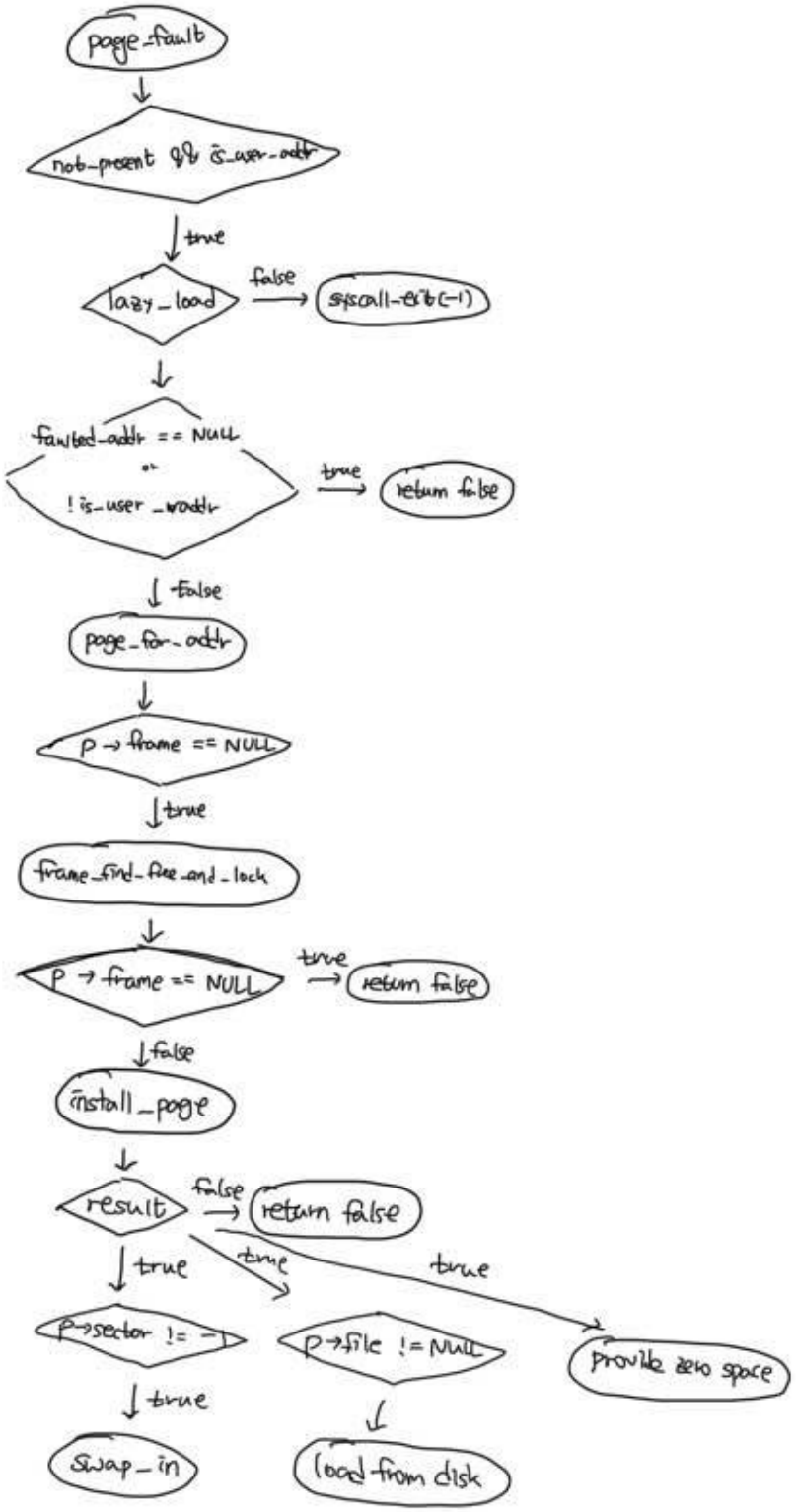
```c
/*
    새로운 page 생성
    성공하면 page의 address return
    실패하면 NULL return
*/
struct page *page_make_new(void *vaddr, bool read_only){
    ASSERT(pg_ofs(vaddr) == 0);

    struct thread *t = thread_current();
    struct page *p = (struct page *)malloc(sizeof(struct page));
    if (p != NULL){
        p->page_addr = vaddr;
        p->frame = NULL;
        p->t = t;
        p->read_only = read_only;
        p->sector = (block_sector_t)(-1);
        p->file = NULL;

        // hash에 삽입
        if (hash_insert(t->page_table, &p->h_elem) != NULL){
            // already inserted
            free(p);
            p = NULL;
        }
    }
    return p;
}
```

Page_make_new()에서 hash_insert() 함수를 사용하는데, 이는 우리가 page table을 관리할 때 hash table을 이용하여 관리하므로 hash table에 insert 해 준 것이다. 이에 관하여서는 이후 supplemental table 부분에서 자세히 다루도록 할 것이다.

이후 page_read_bytes > 0이라면 page의 정보를 할당해주고 read_bytes, zero_bytes, ofs, upage를 업데이트 해주었다. 이 과정을 read_bytes나 zero_bytes가 0이하가 될 때 까지 반복해 주며 segment를 loading 하였다.

3. Lazy Loading, Supplemental Page Table

3.1. Flow Chart



3.2. Added or modified Function & Data Structure

- Added Function

  Page_for_addr(), lazy_load(), page_hash(), page_less()
- Modified Function

  Page_fault(), setup_stack(), load(), init_thread()
- Modified Data Structure

  Struct thread

3.3. Implementation

먼저 loading procedure이 시작될 때 stack을 setup하는 함수인 setup_stack() 함수에서도 마찬가지로 page frame구조에 알맞게 stack이 setup되도록 update해주었다.

```
/* Create a minimal stack by mapping a zeroed page at the top of
   user virtual memory. */
static bool
setup_stack(void **esp)
{
    struct page *p = page_make_new(((uint8_t *) PHYS_BASE) - PGSIZE, false);
    if (p == NULL) return false;

    p->frame = frame_find_free_and_lock(p);
    if (p->frame == NULL) return false;

    bool success = install_page(p->page_addr, p->frame->base_addr, !p->read_only);

    if (success)
        *esp = PHYS_BASE;
    else
        syscall_exit(-1);

    return success;
}
// pj3
```

이 함수에서는 각각 page와 frame을 하나 할당하여 install시키는 과정을 통해 stack을 setup 하였다.

```
    // pj3
    /* If page fault occurs in user mode, terminates the current
     process. */
    if (not_present && is_user_vaddr(fault_addr)) {
        if (!lazy_load(fault_addr)) {
            syscall_exit(-1);
        }
        else return;
    }
    // pj3
```

위 코드는 userprog/exception.c에서 page_fault()함수에서 수정한 부분이다. 이전에는 page fault가 발생하면 바로 syscall_exit(-1)을 호출했지만, 이제는 lazy loading을 통해 올바른 access라면 메모리에 로드되지 않은 page를 메모리로 로딩하게 만들어 주었다.

```
/*
    lazy_loading
    if page's frame is NULL, find one free frame.
    if the page was swapped out, then swap in
    elif the page is one part of file, then load it
    else (== stack growth), then provide 0 space
*/
bool lazy_load(void *faulted_addr){
    if (faulted_addr == NULL || !is_user_vaddr(faulted_addr)){
        return false;
    }

    bool result = false;
    struct page *p = page_for_addr(faulted_addr);
    if (p==NULL) return false;

    if (p->frame == NULL){ // 맨 처음 or swap out 된 상태
        // 빈 frame 하나 받고
        p->frame = frame_find_free_and_lock(p);
        if (p->frame == NULL) return false;

        // install page
        result = install_page(p->page_addr, p->frame->base_addr, !p->read_only);
        if (result == false) return false;

        // lazy load
        if (p->sector != (block_sector_t) -1) {
            // swap out 된 page swap in
            // printf("lazy load\n");
            swap_in(p);
        }
        else if(p->file != NULL) {
            // load from disk
            // printf("load from disk\n");
            off_t read_bytes = file_read_at (p->file, p->frame->base_addr, p->file_length, p->file_offset);
            memset(p->frame->base_addr + read_bytes, 0, PGSIZE - read_bytes);
            if (read_bytes != p->file_length) PANIC("file read failed!");
        }
        else {
            // provide zero space
            // printf("zero space\n");
            memset(p->frame->base_addr, 0, PGSIZE);
        }
    }
    return result;
}
```

먼저 인자로 들어온 faulted_addr이 valid address인지 확인한다. 이후 page_for_addr() 함수를 통해 faulted_addr에 맞는 page를 찾는다. Page_for_addr() 함수에서 page를 찾는 과정은 아래와 같다.

```
/*
    address에 맞는 page를 찾아서 찾으면 그 page return
    만약 address가 stack growth라면 (== stack growth 이면 page가 없음), valid한지 확인하고
    page가 없다면 NULL return
*/
struct page *page_for_addr(void *addr){
    ASSERT(addr != NULL);

    struct page p;
    struct hash_elem *e;
    p.page_addr = (void *)pg_round_down(addr);

    // find page
    e = hash_find(thread_current()->page_table, &p.h_elem);
    if (e!=NULL) return hash_entry(e, struct page, h_elem);

    // not find
    // stack growth
    if ((addr > PHYS_BASE - STACK_MAX) && addr >= (thread_current()->saved_esp - 32)){
        // valid한 stack growth
        return page_make_new(p.page_addr, false);
    }

    // not find, not stack growth
    return NULL;
}
```

Page를 찾는 과정은 page를 hash table에서 관리하므로 hash_find를 통해 찾는다. 만약 page hash table에서 찾지 못한다면 stack growth를 통해 새로운 page를 만들어 주게 된다.

이 과정을 통해 page를 찾아 온 뒤, 해당 page의 frame이 NULL이라면 page가 처음 로딩되거나 swap out 된 상태이므로 먼저 page를 install 해주고, page가 swap out된 경우라면 page를 swap in해주고, page가 file의 일부분이라면 이를 load해주고, 마지막으로 stack growth를 통해 page가 만들어 진 것이라면 해당 page에 zero space를 할당해준다.

다음으로 우리는 page table을 좀 더 효율적으로 관리하기 위해 hash table을 사용 할 것이다. 따라서 먼저, 각 thread마다 page table을 가지고 있게 될 것이므로 struct thread에 hash* 형의 page_table을 아래와 같이 선언해준다.

```
// pj3
struct hash *page_table;
void *saved_esp;
struct list map_list;
mapid_t next_mapid;
// pj3
```

다음으로 load() 함수에서 page table을 할당해주고 hash_init()을 통해 initialize해준다. 이 과정은 아래와 같다.

```
// pj3
/* make hash table */
t->page_table = malloc (sizeof(struct hash));
if (t->page_table == NULL) goto done;
hash_init(t->page_table, page_hash, page_less, NULL);
// pj3
```

Hash_init() 함수에서 인자로 사용하는 page_hash() 함수와 page_less() 함수는 각각 page를 해싱하는 함수와 비교함수이다. 구현은 아래와 같다.

```
/*
    해싱 함수
*/
unsigned page_hash (const struct hash_elem *e, void *aux UNUSED) {
    const struct page *p = hash_entry (e, struct page, h_elem);
    return ((uintptr_t)p->page_addr) >> PGBITS;
}

/*
    해싱 함수
*/
bool page_less (const struct hash_elem *a_, const struct hash_elem *b_, void *aux UNUSED) {
    const struct page *a = hash_entry (a_, struct page, h_elem);
    const struct page *b = hash_entry (b_, struct page, h_elem);
    return a->page_addr < b->page_addr;
}
```

4. Stack Growth

4.1. Flow Chart

4.2. Added or modified Function & Data Structure

- Modified Data Structure
  struct thread
- Added Function
  Page_for_addr()
- Modified Function
  Intr_handler()

4.3. Implementation

먼저 stack growth 의 조건을 check 하기 위해 현재 thread 의 esp 가 필요하다. 따라서 thread 구조체에 해당 thread 의 esp 를 저장하는 variable 인 saved_esp 를 아래와 같이 선언하였다.

```
// pj3
struct hash *page_table;
void *saved_esp;
struct list map_list;
mapid_t next_mapid;
// pj3
```

이는 intr_handler() 함수에서 아래와 같이 할당해주었다.

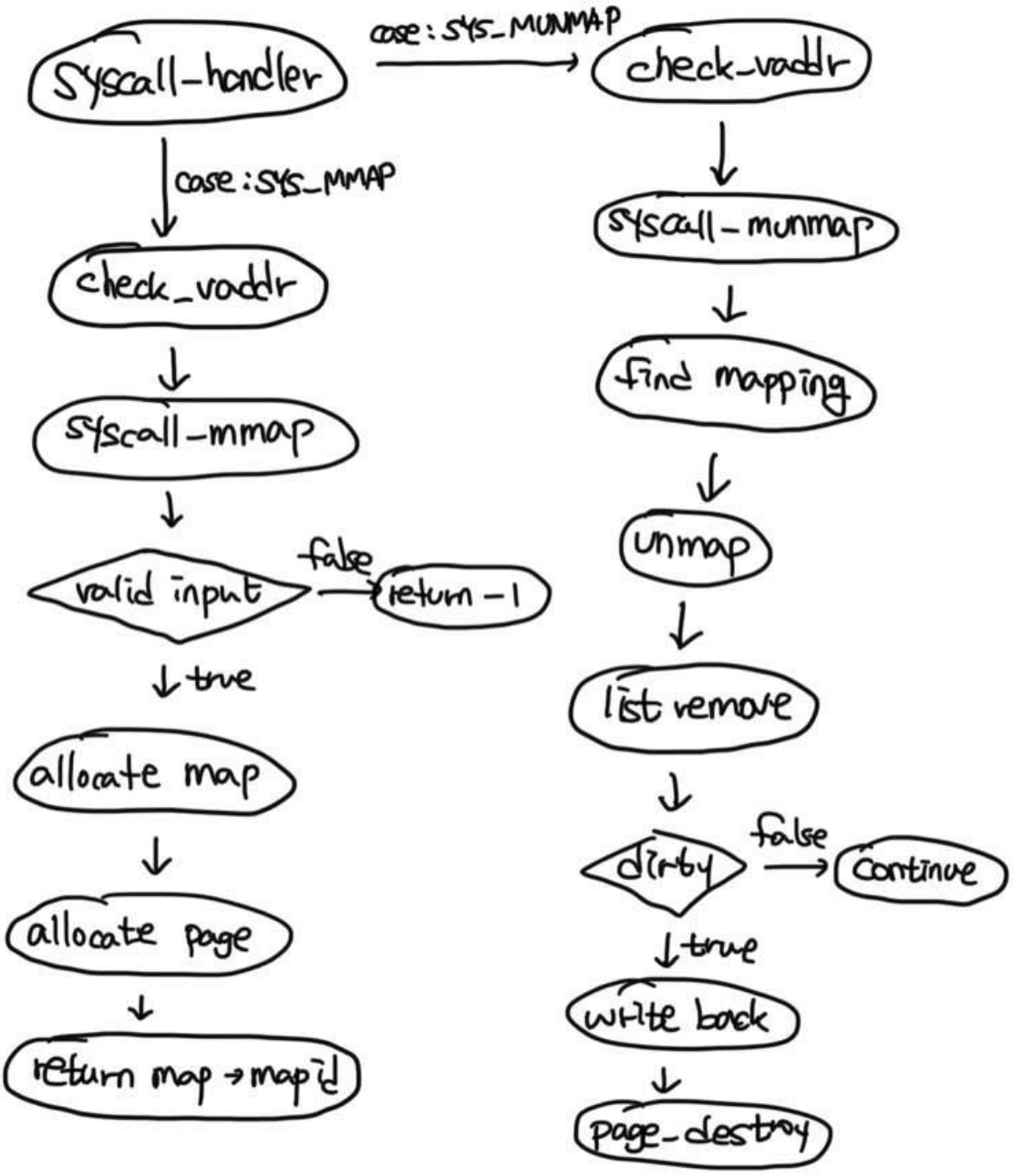
```
// pj3
else thread_current()->saved_esp = frame->esp;
// pj3
```

이후 page_for_addr() 함수에서 address 에 맞는 page 를 찾지 못한다면, stack growth 가 필요한 경우이므로 이 경우에 valid 한 stack growth 가 일어날 조건인지를 확인하고, valid 한 stack growth 가 일어날 조건이라면 page_make_new()를 통해 새로운 page 를 할당해 주었다. 최대 stack 의 크기(STACK_MAX)는 8MB 로 설정하였다. 코드는 아래와 같다.

```
/*
    address에 맞는 page를 찾아서 찾으면 그 page return
    만약 address가 stack growth라면 (== stack growth 이면 page가 없음), valid한지 확인하고 stack growth
    page가 없다면 NULL return
*/
struct page *page_for_addr(void *addr){
    ASSERT(addr != NULL);

    struct page p;
    struct hash_elem *e;
    p.page_addr = (void *)pg_round_down(addr);

    // find page
    e = hash_find(thread_current()->page_table, &p.h_elem);
    if (e!=NULL) return hash_entry(e, struct page, h_elem);

    // not find
    // stack growth
    if ((addr > PHYS_BASE - STACK_MAX) && addr >= (thread_current()->saved_esp - 32)){
        // valid한 stack growth
        return page_make_new(p.page_addr, false);
    }

    // not find, not stack growth
    return NULL;
}
```

5. File Memory Mapping

5.1. Flow Chart



5.2. Added or modified Function & Data Structure

- Modified Data Structure
  struct thread
- Added Data Structure
  struct mapping
- Added Function
  unmap(), syscall_mmap(), syscall_munmap(), is_in_page()

- Modified Function
  Syscall_handler(), check_vaddr(),

5.3. Implementation

먼저 syscall_handler()에서 case 가 SYS_MMAP 이면 vaddr 이 valid 한지 check 해주고, syscall_mmap 을 호출하였고, case 가 SYS_MUNMAP 이면 vaddr 이 valid 한지 check 해주고, syscall_munmap 을 호출하였다. 구현은 아래와 같다.

```
// pj3
case SYS_MMAP:
{
    int fd;
    void *addr;

    check_vaddr(esp + sizeof(uintptr_t));
    check_vaddr(esp + 3 * sizeof(uintptr_t) - 1);
    fd = *(int *)(esp + sizeof(uintptr_t));
    addr = *(void **)(esp + 2 * sizeof(uintptr_t));

    f->eax = (uint32_t)syscall_mmap(fd, addr);
    break;
}
case SYS_MUNMAP:
{
    mapid_t map;

    check_vaddr(esp + sizeof(uintptr_t));
    check_vaddr(esp + 2 * sizeof(uintptr_t) - 1);
    map = *(mapid_t *)(esp + sizeof(uintptr_t));

    syscall_munmap(map);
    break;
}
// pj3
```

Address 가 valid 한지 check 해주는 check_vaddr() 함수에서는 vaddr 이 NULL 이거나, vaddr 이 user address 의 영역에 존재하지 않거나, is_in_page() 함수를 사용하여 vaddr 에 해당하는 page 가 존재하지 않는 경우에는 valid 한 vaddr 이 아니므로 syscall_exit(-1)을 호출해주었다. 구현은 아래와 같다.

```
// pj3
/*
    Checks user-provided virtual address. If it is invalid, terminates the current process.
*/
static void check_vaddr(const void *vaddr)
{
    if (!vaddr || !is_user_vaddr(vaddr) || is_in_page(vaddr) == NULL){
        syscall_exit(-1);
    }
}
```

다음으로 syscall_mmap() 함수에서는 먼저 주어진 input 들이 valid 한 input 인지를 검사하고 valid 하지 않다면 -1 을 return 한다. 모든 input 이 valid input 이라면 새로운 mapping 을 생성하여 알맞은 값들을 할당하고, map_list 에 넣어준다. 그 후, 파일의 크기에 알맞도록 page 들을 생성하여 mapping 에 해당하는 내용을 집어넣어준다.

다음으로 syscall_munmap() 함수에서는 mapid 를 이용하여 해당하는 mapping 을 찾고, unmap 함수를 이용하여 해당되는 mapping 을 삭제한다. 구현은 아래와 같다.

```
/*
    syscall_unmap 함수
    mapid를 이용해 mapping을 찾고. unmap 함수를 호출해서 mapping을 삭제한다.
*/
static void syscall_munmap (mapid_t mapping) {
    struct thread *t = thread_current();
    struct list_elem *e;
    struct mapping *map;
    bool find = false;
    // mampping 찾고
    for (e = list_begin(&t->map_list); e != list_end(&t->map_list); e = list_next(e)){
        map = list_entry(e, struct mapping, elem);
        if (map->mapid == mapping){
            find = true;
            break;
        }
    }
    // 이거 일어나면 오류
    if (!find) PANIC("NO EXIST MAPPING!\n");
    // unmap
    unmap(map);
}
// pj3
```

Unmap() 함수에서는 인자로 받은 map 에 해당되는 mapping 을 제거해야 하므로 먼저 map_list 에서 해당되는 mapping 을 제거하고, 만약 그 mapping 의 dirty bit 이 setting 되어 있으면 file_write_at() 함수를

통해 write back 시켜준다. 그 후 마지막에 page_destroy() 함수를 통해 page 를 deallocate 시켜준다. 구현은 아래와 같다.

```
// pj3
/*
    unmap 함수
    인자로 주어진 mapping을 이용해서 이 mapping을 없앤다.
    dirty페이지는 write back 한다.
    그리고 할당된 page들도 삭제한다.
*/
static void unmap(struct mapping *map){
    struct page *p;
    struct thread *t = thread_current();
    // map list에서 제거
    list_remove(&map->elem);
    // dirty인지 확인하고 dirty이면 write back
    for (int i = 0; i < map->page_cnt; i++) {
        p = page_for_addr((map->page_addr) + (i * PGSIZE));
        if (p == NULL) PANIC("unmap error\n"); // because, umap이므로 page가 없을수가 없음
        if (pagedir_is_dirty(t->pagedir, (map->page_addr) + (i * PGSIZE))){
            lock_acquire (&filesys_lock);
            file_write_at(p->file, p->page_addr, p->file_length, p->file_offset); //마지막은 length만큼만 write back
            lock_release (&filesys_lock);
        }
    }
    // page deallocate
    for (int i = 0; i < map->page_cnt; i++) {
        page_destroy(page_for_addr((map->page_addr) + (i * PGSIZE)));
    }
}
```

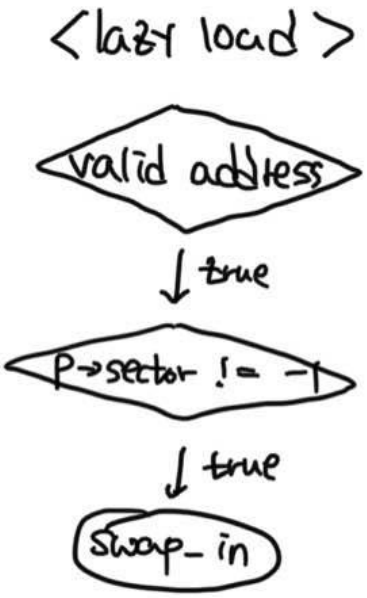
Page 를 deallocate 할 때 사용하는 함수인 Page_destroy() 함수는 다음과 같다.

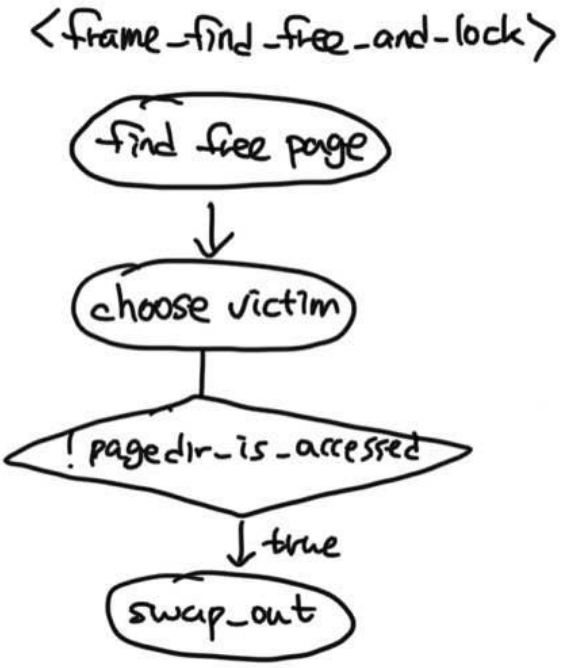
```
/*
    page - frame page에 할당된 frame을 해제하고, page 삭제
*/
void page_destroy(struct page *p){
    frame_free_and_unlock(p->frame);
    if (hash_delete(thread_current()->page_table, &p->h_elem) == NULL) PANIC("hash_delete_fail!\n");
    free (p);
}
```

Frame_free_and_unlock() 함수를 통해 해당되는 page 의 frame 을 free 시키고 unlock 시켜준다.

6. Swap Table

6.1. Flow Chart





6.2. Added or modified Function & Data Structure

- Added Function

  Swap_init(), swap_in(), swap_out()

6.3. Implementation

먼저 swap_init() 함수에서는 swap disk 와 swap bitmap 을 initialize 해준다. 구현은 아래와 같다.

```
/*
    swap initial 함수
*/
void swap_init (void){
    swap_device = block_get_role(BLOCK_SWAP);
    if (swap_device != NULL) swap_bitmap = bitmap_create(block_size(swap_device) / SECTORS);
    if (swap_device == NULL || swap_bitmap == NULL) PANIC("swap initial fail\n");
    lock_init (&swap_lock);
}
```

다음으로 swap_in() 함수에서는 swap disk 에서 데이터를 찾아 할당된 frame 으로 불러오는 작업을 진행한다. 구현은 아래와 같다.

```
/*
    swap in 함수
    swap 디스크에서 데이터를 찾아서, 할당된 frame으로 불러온다.
*/
void swap_in (struct page *p) {
    uint32_t i;
    lock_acquire (&swap_lock);
    for (i = 0; i < SECTORS; i++) block_read (swap_device, p->sector + i, p->frame->base_addr + i * BLOCK_SECTOR_SIZE);
    bitmap_reset (swap_bitmap, p->sector / SECTORS);
    p->sector = (block_sector_t)(-1);
    lock_release (&swap_lock);
}
```

for 문을 통해 block_read() 함수를 반복하여 불러서 swap_disk 에서 data 를 찾아 frame 에 넘겨주는 작업을 하고, 이후 bitmap_reset() 함수를 통해 bitmap 을 다시 세팅해준다. 그리고 p->sector 을 -1 로 설정해 주어서 해당 영역이 memory 에 할당되어 있다는 것을 나타낸다.

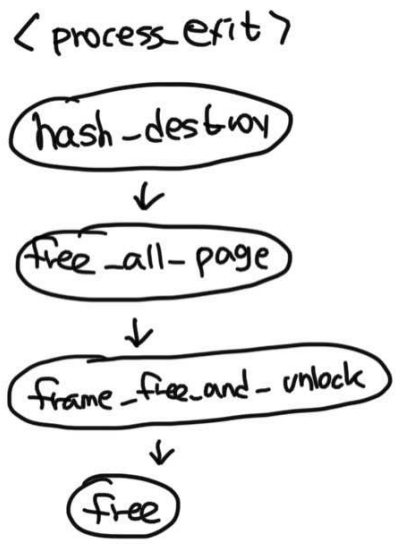
다음으로 swap_out() 함수는 swap disk 의 빈공간에 page 의 frame 의 데이터를 다시 넣어주는 함수이다. 구현은 아래와 같다.

```
/*
    swap out 함수
    swap 디스크에서 빈공간을 찾아서 page의 frame의 데이터를 block에 쓴다.
*/
bool swap_out (struct page *p) {
    ASSERT (p->frame != NULL);
    ASSERT (lock_held_by_current_thread (&p->frame->lock));

    size_t slot;
    size_t i;

    lock_acquire (&swap_lock);

    slot = bitmap_scan_and_flip (swap_bitmap, 0, 1, false);
    if (slot == BITMAP_ERROR) {
        lock_release (&swap_lock);
        return false;
    }

    p->sector = slot * SECTORS;

    for (i = 0; i < SECTORS; i++) {
        block_write (swap_device, p->sector + i, p->frame->base_addr + i * BLOCK_SECTOR_SIZE);
    }

    lock_release (&swap_lock);
    return true;
}
```

Bitmap_scan_and_flip() 함수를 통해 bitmap 을 scan 하여 처음 bit 가 0 인 것을 찾아 1 로 flip 해준다. 그 뒤, 찾은 swap disk 의 영역에 block_write() 함수를 통해 data 를 넣어준다.

7. On process Termination

7.1. Flow Chart



7.2. Added or modified Function & Data Structure

- Modified Function

  Process_exit()
- Added Function

  Free_all_page()

7.3. Implementation

Process 가 termination 될 때, 메모리가 누수되지 않도록 process_exit() 함수에서 hash_destroy() 함수와 free() 를 호출하여 메모리가 정상적으로 종료되도록 하였다. 구현은 아래와 같다.

```
// pj3
hash_destroy(thread_current()->page_table, free_all_page);
free(thread_current()->page_table);
// pj3
```

Hash_destroy() 함수의 인자로 free_all_page() 함수가 사용되었다. 이 함수에서는 frame_free_and_unlock() 함수를 호출하여 frame 을 free 시켜주고 unlock 시켜주었다. 구현은 아래와 같다.

```c
/*
    해싱 함수
*/
void free_all_page(struct hash_elem *p_, void *aux UNUSED){
    struct page *p = hash_entry (p_, struct page, h_elem);
    if (p->frame) frame_free_and_unlock(p->frame);
    free (p);
}
```

8. Discussion

이번 과제를 모두 성공적으로 구현하였다. 이는 아래 첨부한 All Pass의 결과를 보고 알 수 있다.

```
pass tests/filesys/base/syn-write
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-bound
```

```
pass tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
pass tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
pass tests/vm/page-merge-stk
pass tests/vm/page-merge-mm
pass tests/vm/page-shuffle
pass tests/vm/mmap-read
pass tests/vm/mmap-close
pass tests/vm/mmap-unmap
pass tests/vm/mmap-overlap
pass tests/vm/mmap-twice
pass tests/vm/mmap-write
pass tests/vm/mmap-exit
pass tests/vm/mmap-shuffle
pass tests/vm/mmap-bad-fd
pass tests/vm/mmap-clean
pass tests/vm/mmap-inherit
pass tests/vm/mmap-misalign
pass tests/vm/mmap-null
pass tests/vm/mmap-over-code
```

이번 과제를 통해 OS에서 virtual address의 구조가 어떻게 이루어지는지, frame table의 구조가 어떻게 이루어졌는지, page swapping이 어떻게 이루어지는지, memory mapping이 어떤 방식으로 이루어지는지, lazy loading이 어떻게 이루어 지는지 자세히 알게 되었다. Project 3을 구현하면서 Design Report에서 작성했던 것과 다르게 구현된 부분들이 존재한다. 먼저 Frame Table 부분에서 원래 LRU 알고리즘을 사용하려고 하였는데, pintos document에 approximation LRU algorithm을 사용하라고 되어 있는 것을 보고 Clock algorithm을 사용하였다. 다음으로 stack growth 부분에서 stack_growth 함수를 따로 만들어서 page_fault 함수 안에서 호출되도록 구현할 것이라고 하였는데, address에 해당되는 page를 찾는 과정에서 stack growth가 필요한 조건을 찾을 수 있다는 점에 착안하여 page_for_addr() 함수 안에서 stack_growth를 구현하였다.