2020학년도 2학기

Operating System

Project #3.

Final Report

담당교수 : 박찬익
학번 : 20180579 / 20180180
학과 : 컴퓨터공학과 / 컴퓨터공학과
이름 : 정혜일 / 최검기
POVIS ID : hyelie / cgg7777

1. code flow

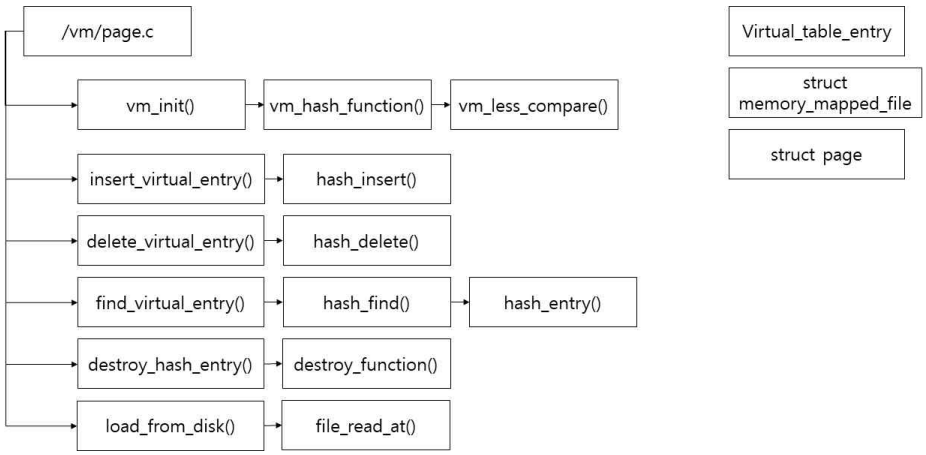

그림 2. /vm/page.c의 code flow

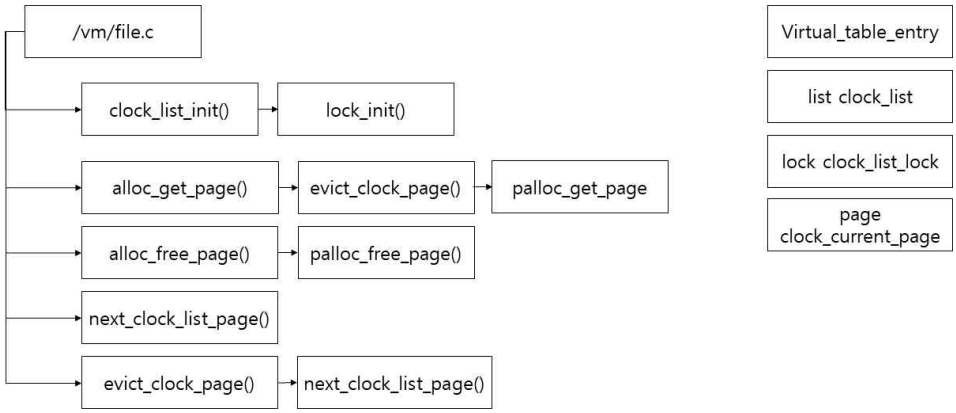

그림 3. /vm/file.c의 code flow

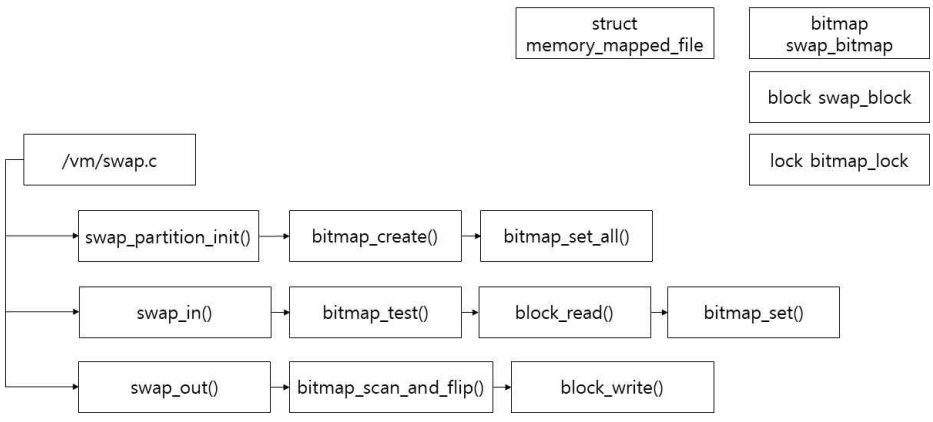

그림 4. /vm/swap.c의 code flow

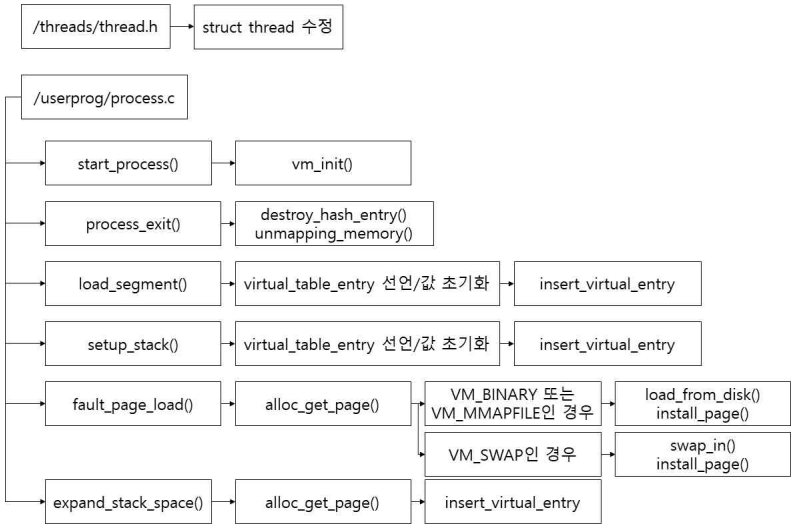

그림 5. /threads/thread.h의 code flow()

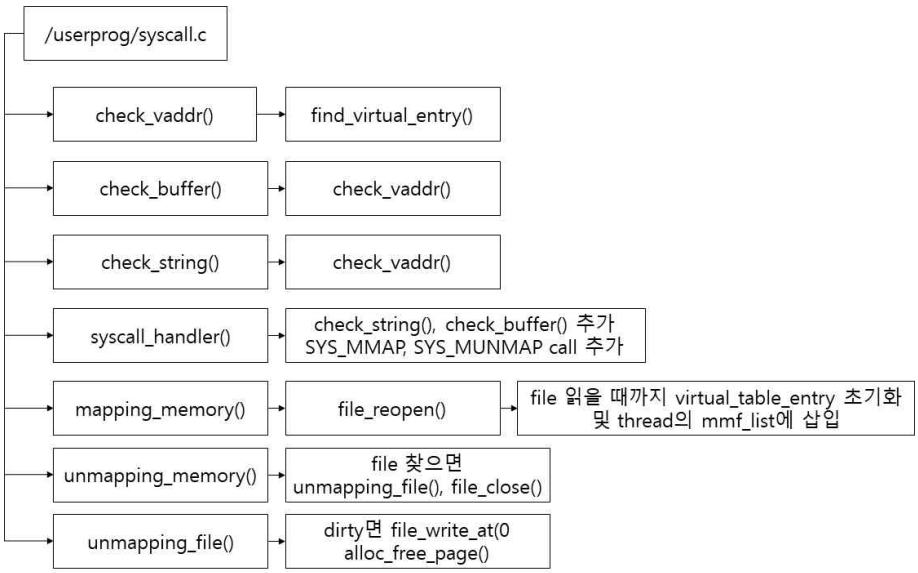

그림 6. /userprog/syscall.c의 code flow

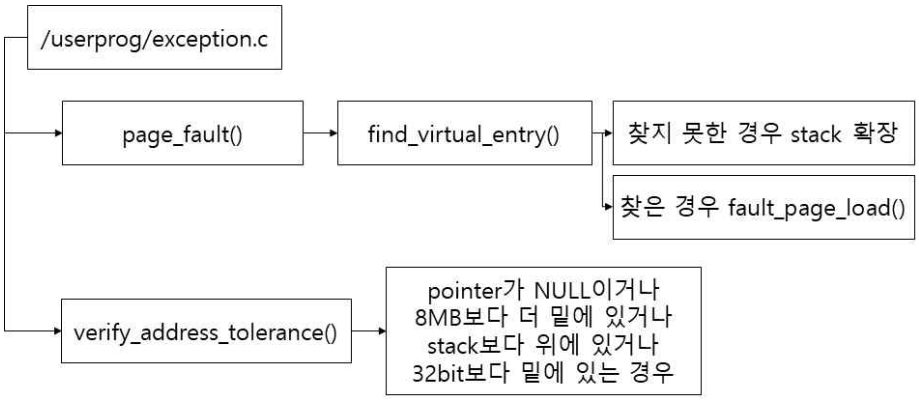

그림 7. /userprog/exception.c의 code flow

2. 코드 분석 및 이유 설명

naive pintos에는 virtual memory가 구현되어 있지 않다. 그래서 이에 대한 구현을 하고자 한다. 구현 방식은 demand paging 방식으로 구현했다. pintos document에 언급되었듯 paging 방식으로 해결해야 한다. 먼저 process.c에서 frame table을 할당할 수 있게 하고, supplemental page table을 수정하고자 한다.

demand paging은 instruction이 실행되면 -> virtual address로부터 virtual page number 얻어냄 -> page table 읽고 -> page table에 physical page가 없으면 page fault -> page frame을 할당하고 page table 갱신 -> disk에서 page를 읽어와서 page frame에 할당의 과정을 거친다.

이 때, document에 언급되었듯 모든 page는 lazy하게 load되어야 한다. 이 lazy loading은, 필요할 때 memory가 load되어야 한다는 것이므로 demand paging과 동일한 것으로 볼 수 있다.

2.1. frame table

PGSIZE는 4096 byte이며, naive pintos는 page frame에 대한 관리를 해야 하기 때문에 먼저 frame table에 대한 구현을 보겠다. demand paging 과정에서, file 전체를 읽어오는 것은 memory 낭비이기 때문에 각 page당 1개의 data structure를 추가해 load할 data만 관리하는 것이 목적이다. 이를 위해 추가하는 data structure는 file pointer, offset, 읽을 data size가 필요하고, 실제 file로부터 읽어온 physical memory를 mapping하는 virtual address에 대한 정보가 필요하다. 이 자료구조는 아래와 같다. source_type은 page가 어디서 로드되었는지에 대한 정보(binary file, file, swap file - 추후에 다룸), virtual_address는 각 각 virtual_table_entry가 가지고 있는 virtual page address, offset과 page_byte, align_byte는 file에서 읽은 offset, 읽을 data sized고 isWritable과 isLoaded는 이 frame에 write 가능한지, load 되었는지에 대한 정보이다. swap_slot은 swapping을 위해 추후 다룰 것이고, vte_hashelem은 virtual table entry의 관리를 위해 hash table에 넣을 것인데, 이에 대한 entry이다. vte_mmfelem은 mapping memory에서 다루고, file은 이 virtual table entry가 여는 file이다.

```
10 struct virtual_table_entry{
11   int source_type;
12   void *virtual_address;
13
14   size_t offset;
15   size_t page_bytes;
16   size_t align_bytes;
17
18   bool isWriteable;
19   bool isLoaded;
20   size_t swap_slot;
21
22   struct hash_elem vte_hashelem;
23   struct list_elem vte_mmfelem;
24   struct file* file;
25 };
```

그림 8. vm/page.h - struct virtual_table_entry

이렇게 virtual_table_entry 자료구조를 각 page마다 하나씩 두어서 page fault가 일어날 때마다 주어진 virtual_address에 해당하는 virtual_table_entry를 참조해 page를 load해야 할 것이다. 그래서 document에 언급되어 있듯 빠른 탐색을 위해 hashing을 이용한다.

각 process에는 1개의 thread가 있고(pintos의 경우) 각 thread가 관리하는 file과 virtual_table_entry에서 hash table이 관리되기 때문에, struct thread 내에 hash table이 추가되어야 하고, process 생성(또는 thread 생성) 시 hash table을 초기화 해야 할 것이다. process가 종료될 때는 hash table을 잘 해제해 주어야 하고, process 실행 중 page fault가 발생하면 hash table에서 virtual table entry를 적당히 탐색해, file에서 값을 읽어와야 할 것이다.

위에서 설명한 바와 같이 thread struct에 hash table을 추가하고, hash table의 초기화, hash 함수를 만들기 위해 virtual address 값으로 hashing하는 함수, 그리고 hash 값의 초기화에 넣기 위해 virtual address를 비교하는 함수를 vm/page.c에 만든다. 비교하는 함수는 virtual_address가 더 작으면 true를 리턴, 아닌 경우 false를 리턴하며, 다른 hashing의 경우 lib/kernel/hash.c에 있는 hashing 함수를 이용했다.

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;                  /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16];              /* Name (for debugging purposes). */
    uint8_t *stack;             /* Saved stack pointer. */
    int priority;               /* Priority. */
    struct list_elem allelem;  /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem; /* List element. */

    /* Owned by devices/timer.c. */
    int64_t wake_ticks; /* Ticks to wake up. */

    /* Shared between thread.c and synch.c. */
    int original_priority;   /* Original priority before donation. */
    struct list donators;    /* List of donators. */
    struct list_elem doelem; /* List element for donators list. */
    struct thread *donee;    /* Thread that is given priority. */

    /* Owned by thread.c. */
    int nice;       /* Figure that indicates how nice to others. */
    int recent_cpu; /* Weighted average amount of received CPU time. */

#ifdef USERPROG
    /* Shared between userprog/process.c and userprog/syscall.c. */
    uint32_t *pagedir;          /* Page directory. */
    struct process *pcb;        /* Process control block. */
    struct list children;       /* List of children processes. */
    struct list fdt;            /* List of file descriptor entries. */
    int next_fd;                /* File descriptor for next file. */
    struct file *running_file; /* Currently running file. */
#endif

    /* Owned by thread.c. */
    unsigned magic; /* Detects stack overflow. */
    struct list mmf_list;
    int next_mmfid;


    // this thread's virtual address management hash table
    struct hash thread_vm_hash_table;
};
```

그림 9. threads/thread.h – struct thread 수정

```
void vm_init(struct hash *thread_vm_hash){
  hash_init(thread_vm_hash, vm_hash_function, vm_less_compare, NULL);
}


static unsigned vm_hash_function(const struct hash_elem *e, void *aux){
  struct virtual_table_entry *temp;
  temp = hash_entry(e, struct virtual_table_entry, vte_hashelem);
  return hash_int(temp->virtual_address);
}


static bool vm_less_compare(const struct hash_elem *a, const struct hash_elem *b){
  struct virtual_table_entry *aentry, *bentry;
  aentry = hash_entry(a, struct virtual_table_entry, vte_hashelem);
  bentry = hash_entry(b, struct virtual_table_entry, vte_hashelem);

  if(aentry->virtual_address < bentry->virtual_address){
    return true;
  }
  else{
    return false;
  }
}
```

그림 10. /vm/page.c - vm_init(), vm_hash_function(), vm_less_compare() 추가

이후 hash 함수에 값을 추가하는 함수 insert_virtual_entry(), 삭제하는 함수 delete_virtual_entry(), 원하는 값을 hashing 결과를 이용해 탐색하는 함수 find_virtual_entry()를 추가한다. insert_virtual_entry()는 insert 성공하면 true를, 아니면 false를 리턴하고 delete_virtual_entry 함수는 삭제에 성공하면 false를, 실패하면 true를 리턴한다.

```
bool insert_virtual_entry(struct hash *thread_vm_hash, struct virtual_table_entry *vte){
  //printf(" **** /vm/page.c : insert_virtual_entry - vte->vitrual_address : %x\n", vte->virtual_address);
  bool result = hash_insert(thread_vm_hash, &(vte->vte_hashelem));
  if(result == NULL)
    return true;
  else
    return false;

}

bool delete_virtual_entry(struct hash *thread_vm_hash, struct virtual_table_entry *vte){
  bool result = hash_delete(thread_vm_hash, &(vte->vte_hashelem));
  if(result == NULL){
    return false;
  }
  else{
    return true;
  }
}

struct virtual_table_entry* find_virtual_entry(void *virtual_address){
  struct virtual_table_entry temp;
  struct hash_elem *temp_hash;


  temp.virtual_address = pg_round_down(virtual_address);
  temp_hash = hash_find(&(thread_current()->thread_vm_hash_table), &(temp.vte_hashelem));
  if(temp_hash == NULL){
    // if not found, return null
    //printf(" ****** /vm/page.c : find_virtual_entry - not found\n");
    return NULL;
  }
  else{
    //printf(" ****** /vm/page.c : find_virtual_entry - find results : %x\n", hash_entry(temp_hash, struct virtual_table_entry, vte_hashelem)->virtual_address);
    return hash_entry(temp_hash, struct virtual_table_entry, vte_hashelem);
  }
}
```

그림 11 /vm/page.c - insert_virtual_entry(), delete_virtual_entry(), find_virtual_entry()

마지막으로 hash data를 해제하는 함수인 destroy_hash_entry(), destroy_function()을 추가한다. destroy_function()의 경우 destroy_hash_entry() 내부에서 사용되며, memory에 load된 hash entry들에 대해서만 이 physical address에 해당하는 page frame을 해제하고, page table에서 이 page를 삭제하고, 동적할당 받은 virtual table entry를 삭제해 준다. 처음에는 인자로 찾은 virtual_table_entry 값만 free했는데, memory에 load된 entry를 삭제하지 않아 디버깅에 오랜 시간이 걸렸다.

```c
void destroy_hash_entry(struct hash *thread_vm_hash){
  hash_destroy(thread_vm_hash, destroy_function);
}

void destroy_function(struct hash_elem *e, void *aux UNUSED){
  struct virtual_table_entry *vte = hash_entry(e, struct virtual_table_entry, vte_hashelem);
  if(vte->isLoaded == true){
    void *physical_address = pagedir_get_page(thread_current()->pagedir, vte->virtual_address);
    alloc_free_page(physical_address);
    pagedir_clear_page(thread_current()->pagedir, vte->virtual_address);
  }
  free(vte);
}
```

그림 12. /vm/page.c - destroy_hash_entry(), destroy_function()

이렇게 만든 hash함수를 초기화하는 vm_init()함수를 process가 시작할 때 hash를 초기화할 수 있게 해야 한다. process.c의 start_process() 함수에 이 내용을 추가하고, 또한 process가 종료되면 동적할당한 모든 hash entry와 hashing table을 삭제해야 하기 때문에 process_exit()에 destroy hashing 함수를 추가한다.

```c
static void
start_process(void *pcb_)
{
    struct process *pcb = pcb_;
    struct intr_frame if_;
    bool success;
    int argc = 0;
    char *argv[MAX_ARGS];
    char *file_name = pcb->file_name;

    /* Set the current process's pcb to PCB. */
    thread_set_pcb(pcb);

    vm_init(&(thread_current()->thread_vm_hash_table));
```

그림 13. /userprog/process.c - start_process()

```c
/* Free the current process's resources. */
void process_exit(void)
{
    struct thread *cur = thread_current();
    struct process *pcb = thread_get_pcb();
    struct list *children = thread_get_children();
    struct list_elem *e;
    struct lock *filesys_lock = syscall_get_filesys_lock();
    uint32_t *pd;
    int max_fd = thread_get_next_fd(), i;

    /* Set exit flag, remove all of the current process's exit
     close all of its files, and notify its parent of its term
     Finally, free its page if it is orphaned. */
    pcb->is_exited = true;
    for (e = list_begin(children); e != list_end(children); e
        process_remove_child(list_entry(e, struct process, chi
    for (i = 2; i < max_fd; i++)
        syscall_close(i);
    sema_up(&pcb->exit_sema);
    if (pcb && !pcb->parent)
        palloc_free_page(pcb);

    unmapping_memory(-1);
    /* Close the running file. */

    //if(filesys_lock->holder != NULL)
    //  printf(" **** /process.c : process_exit - thread tid :
thread_current()->tid, filesys_lock->holder->tid);

    lock_acquire(filesys_lock);
    file_close(thread_get_running_file());
    lock_release(filesys_lock);

    destroy_hash_entry(&(cur->thread_vm_hash_table));
```

그림 14. /userprog/process.c - process_exit()

- 추가 file

/vm/page.h

/vm/page.c

- 추가한 구조체 / 함수

/vm/page.h - struct page_table_entry

/vm/page.c - vm_init()

/vm/page.c - vm_hash_function()

/vm/page.c - vm_less_compare()

/vm/page.c - insert_virtual_entry()
/vm/page.c - delete_virtual_entry()
/vm/page.c - find_virtual_entry()
/vm/page.c - destroy_hash_entry()
/vm/page.c - destroy_function()

- 수정한 구조체 / 함수
/threads/thread.h - struct thread
/userprog/process.c - start_process()
/userprog/process.c - process_exit()

다음으로, virtual address에 접근할 때, 만약 page table에 해당 virtual address에 해당하는 virtual_table_entry가 없다면 virtual_table_entry에 있는 file, offset, page_byte 등을 이용해 disk에서 file을 읽고 physical frame에 load해야 한다. 기존 load_segment에는 page를 할당하고, file에서 page를 load한 후 page table을 설정했지만 이제는 virtual page에 대한 paging을 구현하기 위해 virtual_table_entry를 만들고, 이 virtual_table_entry에 대해 인자로 받은 file, ofs, upage, read_byte, zero_byte, writeable 등을 virtual_table_entry의 값으로 초기화하고, 이렇게 만든 virtual_table_entry를 hash table에 삽입할 수 있게 바꾸어야 한다. 이 때 read_byte가 0이 될 때 까지 file을 읽어야 하고,(읽는 단위는 4KB) 그렇기 때문에 while문을 이용하고 read_byte를 수정해 준다.

```
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;

        struct virtual_table_entry *vte = (struct virtual_table_entry *)malloc(sizeof(struct
virtual_table_entry));
        if(vte == NULL) return false;

        vte->source_type = VM_BINARY;
        vte->virtual_address = upage;
        vte->offset = ofs;
        //printf(" **** src/userprog/process.c : load_segment - vte->offset : %d\n", vte-
>offset);
        vte->page_bytes = page_read_bytes;
        vte->align_bytes = page_zero_bytes;
        vte->isLoaded = false;
        vte->isWriteable = writable;
        vte->swap_slot = 0;
        vte->file = file;

        if(insert_virtual_entry(&(thread_current()->thread_vm_hash_table), vte) == false){
          return false;
        }


        /* Advance. */
        read_bytes -= page_read_bytes;
        zero_bytes -= page_zero_bytes;
        ofs += page_read_bytes;
        upage += PGSIZE;
    }
    return true;
}
```

그림 15. /userprog/process.c - load_segment()

이후 setup_stack을 수정해야 한다. 기존 setup_stack()은 page를 할당하고 stack pointer를 설정하는 것에서 끝났지만, 이제는 virtual_table_entry를 생성하고 내부 자료를 초기화 한 후 hash table에 삽입해야 한다. virtual_table_entry를 만드는 것은 load_segment와 동일하게 진행되었으며 여기서는 offset 등에 넣을 값이 없기 때문에 0으로만 초기화해 주었다.

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

    kpage = alloc_get_page(PAL_USER | PAL_ZERO);
    if (kpage != NULL)
    {
        success = install_page(((uint8_t *)PHYS_BASE) - PGSIZE, kpage->physical_address,
true);
        if (success)
            *esp = PHYS_BASE;
        else
            alloc_free_page(kpage->physical_address);
    }

    struct virtual_table_entry *vte = (struct virtual_table_entry *)malloc(sizeof(struct
virtual_table_entry));
    if(vte == NULL) return false;

    vte->source_type = VM_SWAP;
    vte->virtual_address = ((uint8_t *)PHYS_BASE) - PGSIZE;
    vte->page_bytes = 0;
    vte->align_bytes = 0;
    vte->isLoaded = true;
    vte->isWriteable = true;
    vte->swap_slot = 0;
    vte->file = NULL;
    kpage->vte = vte;

    success = insert_virtual_entry(&(thread_current()->thread_vm_hash_table), vte);

    return success;
}
```

그림 16. /userprog/process.c - setup_stack()

- 추가 file

없음

- 추가한 구조체 / 함수

없음

- 수정한 구조체 / 함수

/userprog/process.c - load_segment()

/userprog/process.c - setup_stack()

이후 address가 주어졌을 때, 각 주소가 주어졌을 때 virtual_table_entry가 존재하는지 검사해야 한다. hash table을 순회하며 값을 찾는 /vm/page.c - find_virtual_entry() 함수를 이용해 hash table에서 virtual address에 해당하는 값을 찾는다. 원래는 boolean이 return type이었지만 이제 hash element를 리턴해야 하기 때문에 리턴 자료형이 바뀌었다. 또한, syscall에서 인자로 받는 모든 자료형들에 대해 그 buffer나 string이 valid한지 검사하기 위해 check_buffer, check_string 함수를 추가했다. check_buffer 함수는 buffer에 있는 모든 data에 대해 address가 valid한지 check_vaddr 함수를 이용해 검사했고, string 또한 \0이 나올 때 까지 모든 address에 대해 valid한지 check_vaddr 함수를 이용해 검사했다.

```c
/* Checks user-provided virtual address. If it is
   invalid, terminates the current process. */
static struct virtual_table_entry* check_vaddr(const void *vaddr)
{
    if (!vaddr || is_user_vaddr(vaddr) == false || is_kernel_vaddr(vaddr) == true)
        syscall_exit(-1);

    return find_virtual_entry(vaddr);
}
```

그림 17. /userprog/syscall.c - check_vaddr()

```c
static void check_buffer(void *buffer, unsigned size, bool isWriteable){
  struct virtual_table_entry *vte;
  void *buffer_address = buffer;
  int i;

  for(i = 0; i<size; i++, buffer_address++){
    vte = check_vaddr(buffer_address);
    if(vte == NULL){
      syscall_exit(-1);
    }
    else{
      if(isWriteable == true && vte->isWriteable == false){
        syscall_exit(-1);
      }
    }
  }
}

static void check_string(const void *str){
  struct virtual_table_entry *vte;
  char *str_address = str;

  while(*str_address != '\0'){
    vte = check_vaddr(str_address);
    if(vte == NULL){
      syscall_exit(-1);
    }
    else{
      str_address++;
    }
  }
}
```

그림 18. /userprog/syscall.c - check_buffer(), check_string()

이후, syscall_handler에서 string을 사용하는 함수는 check_string()함수를 이용하고, buffer를 사용하는 함수는 check_buffer()함수를 이용해 address가 valid한지, 그리고 virtual entry가 있는지 확인한다. 대표적으로 sys_open은 string을 사용하기 때문에 check_string()을, sys_read는 buffer를 사용하기 때문에 check_buffer를 해 준다.

```c
case SYS_OPEN:
{
    char *file;

    check_vaddr(esp + sizeof(uintptr_t));
    check_vaddr(esp + 2 * sizeof(uintptr_t) - 1);
    file = *(char **)(esp + sizeof(uintptr_t));

    check_string(file);

    f->eax = (uint32_t)syscall_open(file);
    break;
}
```

그림 19. /userprog/syscall.c - syscall_handler() : open

```c
case SYS_READ:
{
    int fd;
    void *buffer;
    unsigned size;

    check_vaddr(esp + sizeof(uintptr_t));
    check_vaddr(esp + 1 * sizeof(uintptr_t));
    check_vaddr(esp + 2 * sizeof(uintptr_t));
    check_vaddr(esp + 3 * sizeof(uintptr_t));
    fd = *(int *)(esp + sizeof(uintptr_t));
    buffer = *(void **)(esp + 2 * sizeof(uintptr_t));
    size = *(unsigned *)(esp + 3 * sizeof(uintptr_t));

    // TODO : why isWriteable is true?
    check_buffer(buffer, size, true);

    f->eax = (uint32_t)syscall_read(fd, buffer, size);
    break;
}
case SYS_WRITE:
```

그림 20. /userprog/syscall.c - syscall_handler() : read

- 추가 file

없음

- 추가한 구조체 / 함수

/userprog/syscall.c - syscall_handler()

/userprog/syscall.c - check_buffer()

/userprog/syscall.c - check_string()

- 수정한 구조체 / 함수

/userprog/syscall.c - check_vaddr()

이렇게 하면 frame table을 모두 구현했다.

2.2. lazy loading(page demanding으로 구현함)

이제 demand paging을 구현할 모든 준비가 되었다. 앞에서 언급했듯 page fault가 나면 virtual table entry를 탐색하고, 만약 존재한다면 disk에서 memory를 load하고 그렇지 않으면 exit한다. memory에 load한 이후에는 page table에 install해 준다. 이 과정을 차근차근 보려 한다.

먼저, page를 할당하고 data를 file에서 memory로 load하고, install_page를 이용해 page table에 값을 넣어주는 함수 fault_page_load이다. source type이 VM_BINARY(code)인 경우만 보자면(MMAPFILE과 SWAP은 후술한다), page를 file에서 읽어오고,이 결과가 false라면 할당했던 physical address를 free, 아니라면 true를 리턴한다. 내부에 있는 load_from_disk() 함수는 file_read_at과 virtual_table_entry에 있는 offset, page_byte, file 등의 정보를 이용해 file을 읽어오고, 성공했다면, 4KB를 전부 읽지 못한 경우에는 나머지는 0으로 채워주는 과정을 진행한다. 이 load_from_disk() 함수를 이용해 file을 physical memory에 load하고, install_page를 이용해 physical page와 virtual page를 mapping한다.

```
bool load_from_disk(void *new_page_address, struct virtual_table_entry *vte){
  bool success;
  //printf(" **** /vm/page.c : load_from_disk - vte->offset : %d\n", vte->offset);
  off_t readbyte = file_read_at(vte->file, new_page_address, vte->page_bytes, vte->offset);
  if(readbyte != vte->page_bytes){
    success = false;
  }
  else{
    success = true;
    memset(new_page_address + vte->page_bytes, 0, vte->align_bytes);
  }
  //printf(" **** /vm/page.c : load_from_disk - sucesss : %d\n", success);
  //printf(" **** /vm/page.c : load_from_disk - vte->offset : %d\n", vte->offset);
  return success;
}
```

그림 21. /vm/page.c - load_from_disk()

```
bool fault_page_load(struct virtual_table_entry *vte){
  struct page *new_page_address = alloc_get_page(PAL_USER);
  new_page_address->vte = vte;
  bool load_success;

  //printf(" **** /userprog/process.c : fault_page_load - vte->offset : %d\n", vte->offset);

  if(vte->source_type == VM_BINARY){
    load_success = load_from_disk(new_page_address->physical_address, vte);
    if(load_success == false){
      alloc_free_page(new_page_address->physical_address);
    }
    else{
      load_success = install_page(vte->virtual_address, new_page_address->physical_address, vte->isWriteable);
      vte->isLoaded = load_success;
    }
  }
  else if(vte->source_type == VM_MMAPFILE){
    load_success = load_from_disk(new_page_address->physical_address, vte);
    if(load_success == false){
      alloc_free_page(new_page_address->physical_address);
    }
    else{
        //printf(" **** inff \n");

      load_success = install_page(vte->virtual_address, new_page_address->physical_address, vte->isWriteable);
          //printf(" **** outff \n");
      vte->isLoaded = load_success;
    }
  }
  else if(vte->source_type == VM_SWAP){
    swap_in(vte->swap_slot, new_page_address->physical_address);
    load_success = install_page(vte->virtual_address, new_page_address->physical_address, vte->isWriteable);
    vte->isLoaded = load_success;
  }

  if(load_success == false){
    alloc_free_page(new_page_address->physical_address);
  }

  return load_success;
}
```

그림 22. /userprog/process.c - fault_page_load()

이렇게 추가한 함수들을 이용해 page fault가 일어났을 때 검사문을 돌린다. page fault가 일어났을 때, 해당 address에 해당하는 virtual table entry가 존재한다면 fault_page_load 함수를 이용해 page를 할당하고, file로부터 data를 읽어오고, mapping하는 과정을 거친다. 그렇지 않다면 exit(-1)을 한다. 아래 코드는 완성본 코드로, stack growth까지 구현된 코드이기에 내용이 추가되어 있다.

```
vte = find_virtual_entry(fault_addr);
if(vte != NULL){
  if(fault_page_load(vte) == false){
    //printf(" **** /userprog/exception.c : page_fault - nothing work\n");
    syscall_exit(-1);
  }
}
else{
  //printf(" **** /userprog/exception.c : page_fault - tolerlance\n");
  if(verify_address_tolerlance(f, fault_addr) == true){
    //printf(" **** /userprog/exception.c : page_fault - tolerlance true\n");
    bool a = expand_stack_space(fault_addr);
    //printf(" **** /userprog/exception.c : page_fault - expand result : %d\n", a);
    //bool b = fault_page_load(vte);
    //printf(" **** /userprog/exception.c : page_fault - load result : %d\n", b);
    //printf(" **** /userprog/exception.c : page_fault - a : %d, b : %d\n", a, b);
    if(a == false){
      //printf("**** expand false exit\n");
      syscall_exit(-1);
    }
  }
  else{
    //printf("**** verify false exit\n");
    syscall_exit(-1);
  }
}
```

그림 23. /userprog/exception.c - page_fault()

마지막으로 추가한 file들을 포함하기 위해 makefile에 경로를 추가해 준다.

```
64 # No virtual memory code yet.
65 vm_SRC = vm/page.c
66 vm_SRC += vm/file.c                    # Some file.
67 vm_SRC += vm/swap.c
```

그림 24. pintos/src/Makefile.build

- 추가 file

없음

- 추가한 구조체 / 함수

/userprog/process.c - fault_page_load()

/vm/page.c - load_from_disk()

- 수정한 구조체 / 함수

/userprog/exception.c - page_fault()

2.3. supplemental page table

supplemental page table의 목적은 page fault가 일어난 virtual page에 대해, 어떤 data가 있어야 하는지를 찾아주는 것이 목적이고, terminate가 일어날 때 어떤 resource를 free해야하는지에 대한 정보 때문에 필요하다. 그러나 이 부분은 frame table에서 추가했다.(hash를 이용해 virtual_table_entry들을 관리하고, destroy 시 어떤 정보를 없앨지 관리했음)

위에서 언급한 모든 것들이 정상작동 한다면 page table이 잘 돌아가는 것이고, page table에 대한 요구를 잘 해결한 것이므로 이에 대한 내용은 생략한다. 이 table은 '2.1. frame table'에서 frame table을 관리하면서 해결했고, 각 table entry는 demand paging으로 allocate된다. deallocate는 후술되는 swap table에서 설명된다.

2.4. stack growth

design report에서 작성한 대로, page fault handler가 발생하더라도 의도한 stack size 범위를 만족시킬 경우 stack을 expand 할 수 있도록 구현하였다.

우선, 제한된 stack size내에 존재하는지 체크하는 함수를 구현하였다.

첫 번째 조건은 address 자체가 null인지를 체크하며, 두 번째 조건은 stack expand의 제한 size은 8MB를 초과하는지 체크한다. 마지막 조건은 fault address의 참조를 체크한다. stack pointer보다 4byte 아래에 있는지 확인하여 error 발생을 막는다.

```
bool verify_address_tolerlance(struct intr_frame *f, void *access_address){
  return access_address != NULL && (PHYS_BASE - 0x4000000 <= (uint32_t)access_address) && (f->esp >= (uint32_t)access_address) && (f->esp - 32 <= (uint32_t)access_address);
}
```

그림 25. /userprog/exception.c - verify_address_tolerlance()

아래 함수는 stack growth를 위한 operation을 진행하는 함수이다. stack을 확장하기 위해 새로운 page를 할당하고, 그에 맞는 virtual table entry를 할당한다.

```
bool expand_stack_space(void *access_address){
  void *aligned_address = pg_round_down(access_address);
  struct page *expand_page = alloc_get_page(PAL_USER | PAL_ZERO);
  if(expand_page == NULL){
    return false;
  }
  struct virtual_table_entry *vte = (struct virtual_table_entry *)malloc(sizeof(struct virtual_table_entry));
  if(vte == NULL){
    alloc_free_page(expand_page);
    return false;
  }

  vte->source_type = VM_SWAP;
  vte->virtual_address = aligned_address;
  vte->offset = 0;
  vte->page_bytes = 0;
  vte->align_bytes = 0;
  vte->isWriteable = true;
  vte->isLoaded = true;
  vte->swap_slot = 0;
  vte->file = NULL;

  // map first address to second argument page
  if(install_page(vte->virtual_address, expand_page->physical_address, true) == false){
    alloc_free_page(expand_page->physical_address);
    free(vte);
    return false;
  }
  expand_page->vte = vte;

  if(insert_virtual_entry(&(thread_current()->thread_vm_hash_table), vte) == false){
    return false;
  }

  return true;
}
```

그림 26. /userprog/process.c - expand_stack_space()

page fault 함수의 마지막 부분을 아래와 같이 수정하였다. 앞서 구현한 두 함수를 중심적으로 작동한다. verify_address_tolerlance 함수를 사용하여, expand가 가능한 address가 주어졌는지 체크하고, 만약 expand 가능할 경우 expand_stack_space 함수를 호출하여 stack growth를 구현하였다.

```
vte = find_virtual_entry(fault_addr);
if(vte != NULL){
  if(fault_page_load(vte) == false){
    syscall_exit(-1);
  }
}
else{
  if(verify_address_tolerlance(f, fault_addr) == true){

    bool a = expand_stack_space(fault_addr);

    if(a == false){
      syscall_exit(-1);
    }
  }
  else{
    syscall_exit(-1);
  }
}
```

그림 27. /userprog/exception.c - page_fault()

- 추가한 구조체 / 함수
/userprog/process.c - expand_stack_space()
/userprog/process.c - verify_address_tolerlance()

- 수정한 구조체 / 함수
/userprog/exception.c - page_fault()

2.5. file memory mapping

기존의 pintos에 구현되어 있지 않은 mmap과 munmap 함수를 구현하는 것이 가장 중요한 과제이다. Memory mapping을 구현하기 위해 다음과 같이 구조체를 선언하였다. 아래 구조체는 담고 있는 file에 대한 정보, file을 저장하기 위해 사용된 vte와 mmf elemt들의 list를 member variable로 가지고 있다.

```
struct memory_mapped_file{
  int id;              // identifier
  struct file *file;
  struct list vte_list;
  struct list_elem mmf_elem;
};
```

그림 28. /src/vm/page.h

아래 함수는 mmap을 구현한 함수이다. file descriptor number, VA를 인자로 받아 operation을 진행하며 address의 조건을 체크 한 후 file을 mmf_entry로 encode를 진행한다. PGSIZE 단위로 data를 읽어오며, mmf_entry의 가 가진 list member에 entry를 추가한다. 모든 load가 종료되면, 현재의 mmf_entry를 thread의 mmf_list에 추가하며 추가된 entry의 id를 return한다.

```
int mapping_memory(int fd, void *virtual_address){
  if(!virtual_address || is_user_vaddr(virtual_address) == false || ((uint32_t)(virtual_address))%PGSIZE != 0){
    return -1;
  }
  struct file_descriptor_entry* file_des_ent = process_get_fde(fd);
  struct file *reopened_file = file_reopen(file_des_ent->file);
  if(reopened_file == NULL){
    return -1;
  }
  struct memory_mapped_file *mmf_entry = malloc(sizeof(struct memory_mapped_file));
  if(mmf_entry == NULL){
    return -1;
  }
  mmf_entry->id = thread_current()->next_mmfid;
  mmf_entry->file = reopened_file;
  list_init(&(mmf_entry->vte_list));
  thread_current()->next_mmfid = thread_current()->next_mmfid + 1;
  int read_bytes = file_length(mmf_entry->file);
  int32_t ofs = 0;
  while (read_bytes > 0){
        size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
        size_t page_zero_bytes = PGSIZE - page_read_bytes;
        struct virtual_table_entry *vte = (struct virtual_table_entry *)malloc(sizeof(struct virtual_table_entry));
        if(vte == NULL) {
          return -1;
        }
        vte->source_type = VM_MMAPFILE;
        vte->virtual_address = virtual_address;
        vte->offset = ofs;
        vte->page_bytes = page_read_bytes;
        vte->align_bytes = page_zero_bytes;
        vte->isLoaded = false;
        vte->isWriteable = true;
        vte->swap_slot = 0;
        vte->file = reopened_file;

        if(insert_virtual_entry(&(thread_current()->thread_vm_hash_table), vte) == false){
          return -1;
        }
        read_bytes -= page_read_bytes;
        ofs += page_read_bytes;
        virtual_address += PGSIZE;
        list_push_back(&(mmf_entry->vte_list), &(vte->vte_mmfelem));
  }
  list_push_back(&(thread_current()->mmf_list), &(mmf_entry->mmf_elem));
  return mmf_entry->id;
}
```

그림 29. /src/userprog/syscall.c - mapping_memory()

아래는 munmap 함수의 기능을 위해 구현한 함수이다. 해제하고자 하는 memory_mapped_file의 id number를 입력받으면 mmf_list를 순회하여 일치하는 entry를 찾고, 해당 entry를 해제한다.

```
void unmapping_memory(int mmfid){
  struct list_elem *iter;

  for(iter = list_begin(&(thread_current()->mmf_list)); iter != list_end(&(thread_current()->mmf_list));){
    struct memory_mapped_file *mmf = list_entry(iter, struct memory_mapped_file, mmf_elem);

    if(mmfid < 0 || mmf->id == mmfid){
      unmapping_file(mmf);
      lock_acquire(&filesys_lock);
      file_close(mmf->file);
      lock_release(&filesys_lock);
      iter = list_remove(iter);
      free(mmf);
    }
    else{
      iter = list_next(iter);
    }
  }
}
```

그림 30. /src/userprog/syscall.c - unmapping_memory()

아래는 unmapping_memory 함수 내부에서 사용한 함수이다. memory_mapped_file을 deallocation 하는 operation을 진행하는 함수로, 입력받은 mmf의 vte list를 순회하여 vte를 해제하며, dirty page인 경우 write back을 진행하도록 구현하였다.

```
void unmapping_file(struct memory_mapped_file *mmf){
  struct list_elem *iter;

  for(iter = list_begin(&(mmf->vte_list)); iter != list_end(&(mmf->vte_list));){

    struct virtual_table_entry *vte = list_entry(iter, struct virtual_table_entry, vte_mmfelem);

    if(vte->isLoaded == true){
      if(pagedir_is_dirty(thread_current()->pagedir, vte->virtual_address) == true){
        lock_acquire(&filesys_lock);
        file_write_at(vte->file, vte->virtual_address, vte->page_bytes, vte->offset);
        lock_release(&filesys_lock);
      }
      void *page_physical_address = pagedir_get_page(thread_current()->pagedir, vte->virtual_address);

      pagedir_clear_page(thread_current()->pagedir, vte->virtual_address);

      alloc_free_page(page_physical_address);

      iter = list_remove(iter);
      delete_virtual_entry(&(thread_current()->thread_vm_hash_table), vte);
    }
    else{
      iter = list_next(iter);
    }
  }
}
```

그림 31. /src/userprog/syscall.c - unmapping_file()

이후, 구현된 함수를 이용하여 syscall_handler에 mmap, munmap의 syscall 기능을 추가하여 기능을 완성시킬 수 있다.

```
case SYS_MMAP:
{
    int fd;
    void *buffer;

    check_vaddr(esp + sizeof(uintptr_t));
    check_vaddr(esp + 2 * sizeof(uintptr_t));
    check_vaddr(esp + 3 * sizeof(uintptr_t) - 1);

    fd = *(int *)(esp + sizeof(uintptr_t));
    buffer = *(void **)(esp + 2 * sizeof(uintptr_t));

    f->eax = mapping_memory(fd, buffer);

    break;
}
case SYS_MUNMAP:
{
    int fd;

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

    fd = *(int *)(esp + sizeof(uintptr_t));
    unmapping_memory(fd);

    break;
}
default:
    syscall_exit(-1);
}
```

그림 32. /src/userprog/syscall.c - syscall_handler()

- 추가한 구조체 / 함수

/vm/page.h - struct memory_mapped_file
/userprog/syacall.c - mapping_memory()
/userprog/syacall.c - unmapping_memory()
/userprog/syacall.c - unmapping_file()

- 수정한 구조체 / 함수

/userprog/syacall.c - syscall_handler()

2.6. swap table

swap 기능 구현을 위해 swap table, swap in, swap out기능을 구현하였으며 document의 내용에 따라 sector 기반 read/write operation 수행을 위하여 bitmap의 기능들을 사용하였다 swap_partition_init function은 swap 기능 구현에 필요한 swap block과 bitmap structure를 setting 하는 기능을 수행한다.

```
void swap_partition_init(void){
  lock_init(&(bitmap_lock));
  swap_block = block_get_role(BLOCK_SWAP);
  if(swap_block == NULL){
    return;
  }
  // page per block. just one bit need per one page.
  // swap partition is page unit. so bit number is page per block
  // sector/block / sector/page -> page/block
  swap_bitmap = bitmap_create(block_size(swap_block) * PGSIZE / BLOCK_SECTOR_SIZE);
  if(swap_bitmap == NULL){
    return;
  }
  bitmap_set_all(swap_bitmap, 0);
}
```

그림 33. /src/vm/swap.c - swap_partition_init()

swap_in function은 page fault발생 시, swap disk로부터 page frame으로 allocating을 진행하는 함수이다. page 할당에 필요한 sector 숫자만큼 loop를 돌며 block_read 함수를 이용하여 sector의 정보를 physical memory로 load한다. 이후 할당된 page frame에 해당하는 bitmap을 0으로 변경한다.

```
void swap_in(size_t swap_idx, void *physical_address){

  //if(bitmap_lock.holder != NULL)
  //  printf(" **** /page.c : swap_in - thread tid : %d, lock holder tid : %d\n", thread_current()->tid, bitmap_lock.holder->tid);


  lock_acquire(&(bitmap_lock));
  if(bitmap_test(swap_bitmap, swap_idx) == 0){
    lock_release(&(bitmap_lock));
    return;
  }
  // must loop for 'sector per page'
  int i, sector_per_page = PGSIZE / BLOCK_SECTOR_SIZE;
  block_sector_t sector_number = sector_per_page * swap_idx;
  for(i = 0; i<sector_per_page; i++, sector_number++){
    block_read(swap_block, sector_number, physical_address);
    physical_address += BLOCK_SECTOR_SIZE;
  }
  //block_read(swap_block, ???, physical_address);
  // if use, then 1
  bitmap_set(swap_bitmap, swap_idx, 0);

  lock_release(&(bitmap_lock));
}
```

그림 34. /src/vm/swap.c - swap_in()

swap_out 함수는 evicted된 page를 swap disk로 가져와 free frame을 만드는 operation을 수행한다. physical address를 입력받아 page에 해당하는 sector 숫자 만큼 loop를 돌며 block에 page frame의 data를 write한다.

```
size_t swap_out(void *physical_address){


  //if(bitmap_lock.holder != NULL)
  //  printf(" **** /page.c : swap_out - thread tid : %d, lock holder tid : %d\n", thread_current()->tid, bitmap_lock.holder->tid);

  lock_acquire(&(bitmap_lock));

  // find bitmap index and change, and write to file
  size_t zero_swap_idx = bitmap_scan_and_flip(swap_bitmap, 0, 1, 0);

  if(zero_swap_idx == BITMAP_ERROR){
    return BITMAP_ERROR;
  }
  int i, sector_per_page = PGSIZE / BLOCK_SECTOR_SIZE;
  block_sector_t sector_number = sector_per_page * zero_swap_idx;
  for(i = 0; i<sector_per_page; i++, sector_number++){
    block_write(swap_block, sector_number, physical_address);
    physical_address += BLOCK_SECTOR_SIZE;
  }

  lock_release(&(bitmap_lock));

  return zero_swap_idx;
}
```

그림 35. /src/vm/swap.c - swap_out()

Approximate LRU를 구현하기 위해, clock_algorithm을 이용하여 swapping이 일어나도록 한다. clock algorithm은 모든 entry를 순차적으로 순회하면서 access bit를 체크하여 evict 될 page를 선택하는 알고리즘이다. evict_clock_page함수는 현재의 entry의 access bit를 체크하고, evict 유무를 판단하며, 만약 evict가 일어나는 경우 dirty bit를 체크하여 wrtie back을 수행할지 결정하는 operation을 수행한다. 순차적으로 현재의 entry를 변화시키는 함수는 아래와 같이 next_clock_list_page 함수를 사용하여 구현하였다.

```
void evict_clock_page(enum palloc_flags flags){
  struct page *victim_page;


  if(list_empty(&(clock_list)) == true){
    return;
  }
  while(1){
    next_clock_list_page();
    if(clock_current_page == NULL){
      return;
    }
    if(pagedir_is_accessed(clock_current_page->t->pagedir, clock_current_page->vte->virtual_address) == true){
      pagedir_set_accessed(clock_current_page->t->pagedir, clock_current_page->vte->virtual_address, 0);
    }
    else{
      lock_acquire(&(clock_list_lock));
      victim_page = clock_current_page;
      break;
    }
  }
  if(victim_page->vte->source_type == VM_BINARY){
    if(pagedir_is_dirty(clock_current_page->t->pagedir, clock_current_page->vte->virtual_address) == true){
      victim_page->vte->swap_slot = swap_out(victim_page->physical_address);
      victim_page->vte->source_type = VM_SWAP;
    }
  }
  else if(victim_page->vte->source_type == VM_MMAPFILE){
    if(pagedir_is_dirty(clock_current_page->t->pagedir, clock_current_page->vte->virtual_address) == true){
      lock_acquire(syscall_get_filesys_lock());
      file_write_at(clock_current_page->vte->file, clock_current_page->vte->virtual_address, clock_current_page->vte->page_bytes, clock_current_page->vte->offset);
      lock_release(syscall_get_filesys_lock());
    }
  }
  else if(victim_page->vte->source_type == VM_SWAP){
    victim_page->vte->swap_slot = swap_out(victim_page->physical_address);
  }
  victim_page->vte->isLoaded = false;
  pagedir_clear_page(victim_page->t->pagedir, victim_page->vte->virtual_address);
  palloc_free_page(victim_page->physical_address);
  list_remove(&(victim_page->clock_elem));
  free(clock_current_page);

  lock_release(&(clock_list_lock));
}
```

그림 36. /src/vm/file.c - evict_clock_page

```
static struct list_elem *next_clock_list_page(void){
  // if don't see any page, then need to allocate clock list's begin
  if(clock_current_page == NULL){
    // if clock list is empty, then nothing set
    if(list_begin(&(clock_list)) == list_end(&(clock_list))){
      return NULL;
    }
    // else clock list have any page, then allocate list begin.
    else{
      clock_current_page = list_entry(list_begin(&(clock_list)), struct page, clock_elem);
      return list_begin(&(clock_list));
    }
  }
  // if see some page, then need to see next clock page.
  else{
    // if next page is list end, then need to see first page
    if(list_next(&(clock_current_page->clock_elem)) == list_end(&(clock_list))){
      clock_current_page = list_entry(list_begin(&(clock_list)), struct page, clock_elem);
      return list_begin(&(clock_list));
    }
    // if next page is not list end, then need to see next page
    else{
      clock_current_page = list_entry(list_next(&(clock_current_page->clock_elem)), struct page, clock_elem);
      return list_next(&(clock_current_page->clock_elem));
    }
  }
}
```

그림 37. /src/vm/file.c - next_clock_list_page

2.7. on process termination

이번 project를 통해 새롭게 선언한 다양한 structure가 process의 수행과정에서 생성된다. 정상적인, 그리고 효율적인 resource 사용을 위하여 process의 종료 시 할당된 다양한 구조체를 해제해 줄 필요가 있다. 따라서 process_exit 함수를 다음과 같이 수정한다.

```
/* Free the current process's resources. */
void process_exit(void)
{
    struct thread *cur = thread_current();
    struct process *pcb = thread_get_pcb();
    struct list *children = thread_get_children();
    struct list_elem *e;
    struct lock *filesys_lock = syscall_get_filesys_lock();
    uint32_t *pd;
    int max_fd = thread_get_next_fd(), i;

    /* Set exit flag, remove all of the current process's exited children,
     close all of its files, and notify its parent of its termination.
     Finally, free its page if it is orphaned. */
    pcb->is_exited = true;
    for (e = list_begin(children); e != list_end(children); e = list_next(e))
        process_remove_child(list_entry(e, struct process, childelem));
    for (i = 2; i < max_fd; i++)
        syscall_close(i);
    sema_up(&pcb->exit_sema);
    if (pcb && !pcb->parent)
        palloc_free_page(pcb);

    unmapping_memory(-1);
    /* Close the running file. */

    lock_acquire(filesys_lock);
    file_close(thread_get_running_file());
    lock_release(filesys_lock);

    destroy_hash_entry(&(cur->thread_vm_hash_table));
```

그림 38. /src/userprog/process.c - process_exit()

unmapping_memory 함수를 사용하여 thread에 할당된 모든 mmf entry를 해제하고, destroy_hash_entry 함수를 사용하여 현재 thread의 hash table의 할당을 해제한다.

3. 검증 과정 및 스크린샷

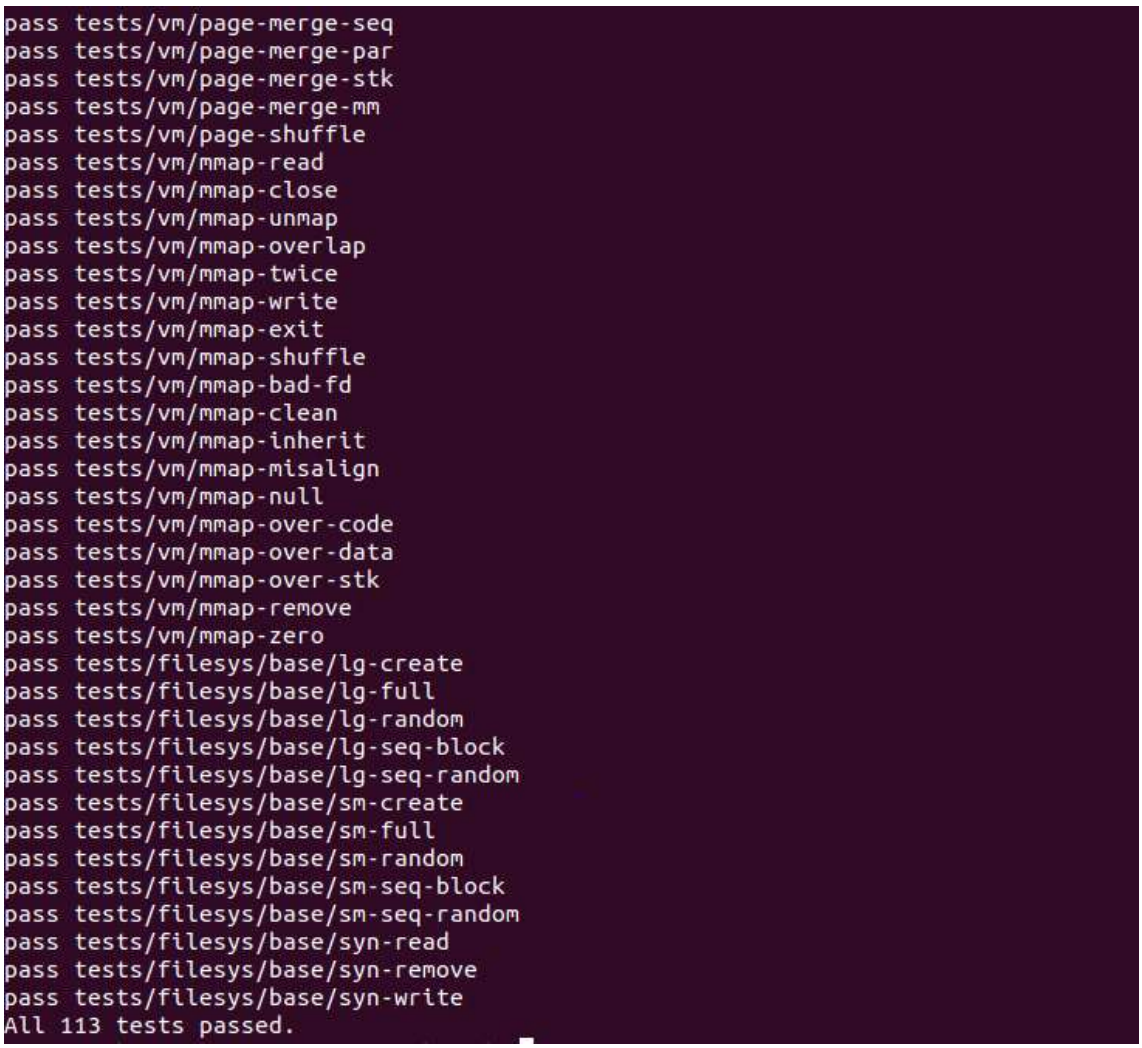

그림 39. test result

위와 같이, 모든 test case를 통과하여 이번 project를 완수하였음을 확인할 수 있었다.

4. discussion

이번 프로젝트를 수행하며 가장 힘들었던 점은 make check를 이용한 test 결과를 확인하는 과정에서 각 part의 interaction이 매우 컸다는 점에 있었다. 특히 destroy_function() 함수를 다루는 과정에서 큰 어려움이 있었다. 처음에 demanding page 작업을 완성시키기 위해 destroy_function() 함수를 단순히 vte를 deallocation하는 기능만을 가지게 하여 만들었었는데, 이후 mmap part를 구현한 이후에는 해당 함수가 page structure의 member들을 모두 할당 해제할 수 있게 구현해주어야 했었다. 그러나 많은 양의 다른 함수들을 작성하는 과정에서 이 점을 누락하였고, 오랜 시간의 코드 검토를 거친 후에야 해당 문제를 찾아 고칠 수 있었다. 또한 lock과 관련하여서도 많은 sync 문제가 발생하였으나, 오랜 디버깅 끝에 thread간 sync를 맞추어 성공적으로 test를 통과시킬 수 있었다.

이번 project는 디버깅에 굉장히 많은 시간을 소요하였지만, 그만큼 demanding page, mmap, swapping등 resource를 효율적으로 수행하기 위한 OS의 기본적인 mechanism에 대한 깊은 이해를 할 수 있는 기회가 된 것 같다.