Pintos Project 3. Pintos Virtual Memory

(설계 프로젝트 수행 결과)

과목명: [CSE4070-01] 운영체제
담당교수: 서강대학교 컴퓨터공학과 이혁준
조원: 54조 고창영

개발기간: 2014. 12. 07. - 2014. 12. 25.

최 종 보 고 서

프로젝트 제목: Pintos 프로젝트 3. Pintos Virtual Memory
제출일: 2014. 12. 26.
참여조원: 54조 고창영

1 개발목표

최소한의 기능을 갖는 교육용 OS Pintos를 만들어나가는 과정에서 OS의 Page 관리 방법과 Intel x86의 Paged Segment 메모리 관리 방식, 그리고 이를 Frame과 연계해서 다양한 Page Replacement 방법들의 구현 및 Memory Mapped File의 구현을 해봄으로써 VM 전반 및 Error 핸들링, User모드와 Kernel모드에 대한 개념들을 직접 실습함으로써 이들에 대한 이해도를 높인다.

2 개발 범위 및 내용

2.1 개발 범위

1. Stack Growth
2. Supplemental Page Table
3. Pure Demand Paging
4. Frame Allocator with Swap

2.2 개발 내용

2.2.1 Stack Growth

`Process`당 `Stack`의 크기는 기본적으로 `4kB`로 고정되어있으나, 이러한 제한을 `8MB`까지 늘린다.

2.2.2 Supplemental Page Table

원래 `Memory`에 할당되었으나 현재 `Memory`상에 존재하지 않는 `Page`들의 현재 위치를 적어놓는 용도로 사용한다 (예를 들어 `Swapped out`된 `Page`나, 아직 `Load`되지 않은 `Code/Data Segment`들이 각 `File`및 `Offset`과 함께 저장된다). 이는 추후 해당 `Memory` 영역에 `Access`가 일어났을 때 해당 영역이 속하는 `Page`에 원래 있어야 할 내용을 복구하여 정상적인 수행을 가능하게 한다.

2.2.3 Pure Demand Paging

Memory의 각 Page는 그 Page가 필요해졌을 때 할당한다. 즉, 실제 Process가 사용하기 전까지는 해당 영역을 할당하지 않고 뒀다가, 해당 영역에 접근해서 Page Fault가 발생하면 비로소 할당한다. 이는 각 Process의 Code Segment와 Data Segment에도 해당하는데, 이러한 경우에 모든 내용을 Memory에 바로 올리지 않고 Supplemental Page Table에 각 영역이 어떤 File의 어떤 Offset에 존재하는지만 적어둔 다음, 해당 부분이 Access될 때 로드한다.

2.2.4 Frame Allocator with Swap

User Process가 직접 Page를 할당하지 않고, Frame Allocator를 통해 Page를 할당받도록 한다. 이렇게 함으로써 각 Process가 어떤 Page를 가지고 있는지, 그리고 어떤 Page를 Access하는지 알 수 있고, 이를 통해서 각 Process가 현재 필요로 하지 않는 Page들은 Swap Disk로 Swap out시킨 다음 Supplemental Page Table에 적어놓고 해당 Frame을 다른 Page에 Mapping함으로써 실제 Physical Memory Size보다 더 큰 Virtual Memory를 사용할 수 있도록 한다.

2.2.5 Memory Mapped File

File의 접근을 일반적인 memory 접근처럼 편하게 할 수 있도록 구현한다.

3 추진 일정 및 개발 방법

3.1 추진 일정

- 12.07 - 12.07: Pintos 매뉴얼 정독 및 threads/palloc.c, userprog/pagedir.c, 그리고 devices/block.c정독 후 요구사항을 최대한 간단하게 Reduce.
- 12.07 - 12.08: 실제 코드의 작성(Stack Growth, Supp.Page Table, Pure Demand Paging, Frame Allocator).
- 12.09 - 12.11: 디버깅(page-merge-stk, page-parallel test와 같은 까다로운 test).
- 12.24 - 12.24: Memory Mapped File의 구현.
- 12.25 - 12.25: 제출 전 마지막 디버깅(free의 timing 문제).

3.2 개발 방법

개발은 다음과 같은 원칙을 바탕으로 진행했다:

1. Pintos Manual의 내용을 숙지한다.
2. 이미 작성되어있는 Pintos 내부의 코드(pagedir.c 등)를 숙지한다.

3. 구현해야할 각 기능의 목표와 작동, `interrupt disable/enable`의 필요성을 숙지한다.
4. 구현시 한 함수를 작성하면 해당 함수의 작동을 완벽히 분석한 다음 다른 함수를 작성한다.
5. 각 기능 구현의 기한을 정하고, 이에 맞춰 작성할 수 있도록 시간을 잘 안배한다.

또한, 다음과 같이 도표를 그려서 각 함수의 연결과 그 흐름을 확인했다.

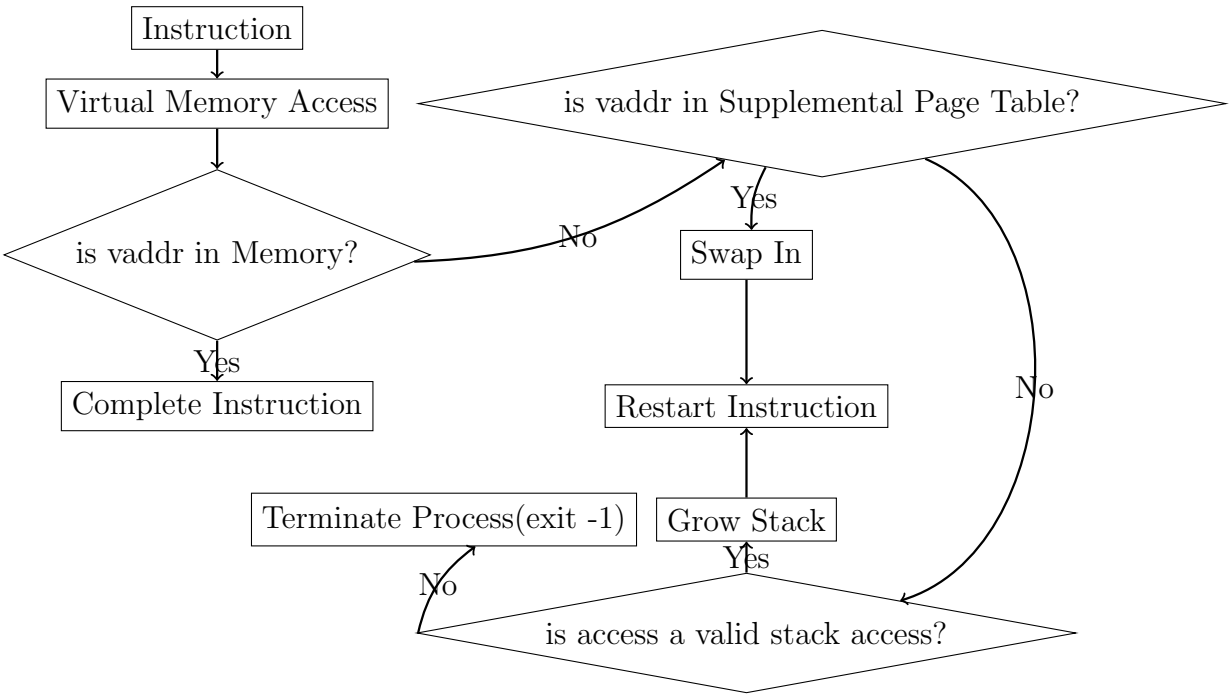

Figure 1: Page Fault Handler 구현 흐름도

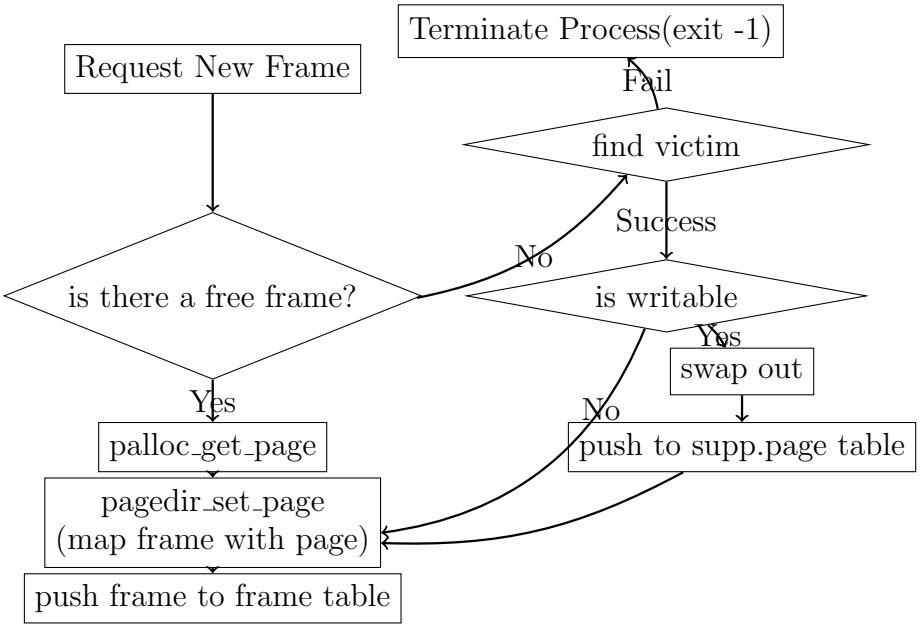

Figure 2: Frame Allocator 구현 흐름도

3.3 연구원 역할 분담

혼자서 하는 프로젝트였으므로 역할을 분담하기보다는 위에서 적은 것과 같이 날짜를 기준으로 작업을 분배하였다.

4 연구 결과

결과적으로 6개를 제외한 모든 test에 대해 pass를 받을 수 있었다(*:**채점항목**):

```
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/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-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
```

```
FAIL tests/vm/mmap-close
FAIL 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
FAIL tests/vm/mmap-clean
pass tests/vm/mmap-inherit
pass tests/vm/mmap-misalign
FAIL tests/vm/mmap-null
pass tests/vm/mmap-over-code
pass tests/vm/mmap-over-data
pass tests/vm/mmap-over-stk
FAIL tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
6 of 109 tests failed.
```

4.1 합성 내용

4.1.1 Stack Growth

<Figure 1>에서 볼 수 있듯이, Valid Stack Access일 경우 Page Fault Handler에서 Stack의 크기를 늘려주었다.

Valid Stack Access 인지 확인 User Mode에서 Page Fault가 발생한 경우, ESP와 Fault Address가 모두 구간[PHYS_BASE$-8\times1024\times1024$,PHYS_BASE)안에 속해야 한다. 또한, NotPresent Error인데 Read Access인건 의미가 없으므로 Write인 것만 체크한다.

Stack의 크기를 늘림 Fault가 일어난 가장 낮은 Address까지 Stack Page를 반복문을 수행하면서 한 개씩 계속 늘려 나간다. Stack이 자라는 방향은 높은 Address에서 낮은 Address쪽이다.

4.1.2 Supplemental Page Table

빠른 성능을 위해 Per-Process Hash Table을 이용, 각 entry를 구조체{Swap File, Swap Offset, Virtual Address, Is Writable?}로 저장했다.

4.1.3 Pure Demand Paging

<Figure 1>에서 볼 수 있듯이, Page Fault가 일어나면 Supplemental Page Table을 검색하여 해결하도록 했다.

4.1.4 Frame Allocator with Swap

<Figure 2>와 같이 Frame을 할당하고, 이를 Page Table에 연결시키는 작업과, Frame이 부족할 경우 희생 Frame을 찾고, Swap에 쓰는 작업을 수행하도록 작성했다. 각 Frame에 대한 정보 역시 Per-Process Hash Table로 저장한다.
희생 Frame을 찾는 것은 Second Chance(CLOCK) Algorithm을 이용했다.

4.1.5 Memory Mapped File

거의 Demand loading of segments와 같이 구현했다.
Supplemental Page Table에 file의 offset과 Page의 user virtual address를 저장한 다음, 이를 처음 요청할 때 file에서 지정된 offset을 이용, 값을 읽어왔다. 이후 이렇게 로드된 값이 변경될 경우 file을 unmap할 때 다시 file로 값을 쓰도록 구현했다. 또한, processk 종료될 때에는 implicit하게 모든 mapped memory region을 unmap하도록 구현했다.

4.2 제작 내용

4.2.1 Stack Growth

<Figure 1>에서 볼 수 있듯이, Valid Stack Access일 경우 Page Fault Handler에서 Stack의 크기를 늘려주었다.

Valid Stack Access 인지 확인 User Mode에서 Page Fault가 발생한 경우, ESP와 Fault Address가 모두 구간[PHYS_BASE$-8 \times 1024 \times 1024$,PHYS_BASE)안에 속해야 한다. 또한, NotPresent Error인데 Read Access인건 의미가 없으므로 Write인 것만 체크한다.
구현은 아래와 같다:

```
/* Stack Growth */
if (user && not_present && write &&
  fault_addr >= STACK_LIMIT && f->esp >= STACK_LIMIT &&
  fault_addr <= PHYS_BASE && f->esp <= PHYS_BASE) {
    ...
}
```

Stack의 크기를 늘림 Fault가 일어난 가장 낮은 Address까지 Stack Page를 반복문을 수행하면서 한 개씩 계속 늘려 나간다. Stack이 자라는 방향은 높은 Address에서 낮은 Address쪽이다.
구현은 아래와 같다:

```
/* Stack Growth */
if (/*...*/) {
  uint8_t *stack_ptr1= (uint8_t*)(((unsigned)(fault_addr)) / PGSIZE * PGSIZE)
    + PGSIZE;
  uint8_t *stack_ptr2= (uint8_t*)(((unsigned)(f->esp)) / PGSIZE * PGSIZE)
    + PGSIZE;
  uint8_t *stack_ptr = stack_ptr1 > stack_ptr2 ? stack_ptr1 : stack_ptr2;

  while (stack_ptr > f->esp || stack_ptr > fault_addr) {
  stack_ptr -= PGSIZE;
  if (frame_find_upage(&thread_current() -> frame_table, stack_ptr))
  continue;
  if (frame_get_page(&thread_current() -> frame_table, stack_ptr,
 FRM_WRITABLE | FRM_ZERO) == NULL)
  break;
  }
  if (stack_ptr <= f->esp && stack_ptr <= fault_addr)
  return;
}
```

4.2.2 Supplemental Page Table

빠른 성능을 위해 Per-Process Hash Table을 이용, 각 entry를 구조체{Swap File, Swap Offset, Virtual Address, Is Writable?}로 저장했다.
구현은 아래와 같다:

```
/* ... */
struct supp_page_entry*
```

```
supp_page_find(struct hash *supp_page_table, uint8_t *uaddr)
{
    acquire_page_lock();
    struct hash_elem *e = NULL;
    struct supp_page_entry *item =
(e = find_supp_page_entry(supp_page_table, uaddr)) ?
 hash_entry(e, struct supp_page_entry, all_elem) : NULL;
    release_page_lock();
    return item;
}

void
supp_page_insert(struct hash *supp_page_table, uint8_t *upage,
 struct file *swap_file, size_t swap_offset,
 size_t length, bool is_segment, bool is_writable)
{
    acquire_page_lock();
    struct supp_page_entry *item = &pool[pool_ptr++];
    ASSERT(pool_ptr <= PAGE_POOL_SIZE);

    item -> user_vaddr = upage;
    item -> swap_file = swap_file;
    item -> swap_offset = swap_offset;
    item -> length = length;
    item -> is_segment = is_segment;
    item -> is_writable = is_writable;

    hash_insert(supp_page_table, &item -> all_elem);

    release_page_lock();
}
/* ... */
```

4.2.3 Pure Demand Paging

<Figure 1>에서 볼 수 있듯이, Page Fault가 일어나면 Supplemental Page Table을 검색하여 해결하도록 했다.
구현은 아래와 같다:

```
if (user && not_present) {
  uint8_t *fault_page = (uint8_t*)((unsigned)fault_addr / PGSIZE * PGSIZE);
  struct supp_page_entry *e = supp_page_find(&thread_current() -> supp_page_table,
     fault_page);
```

```
  /* Lazy loading of segments. */
  if (e && e -> is_segment) {
  uint8_t *frm = frame_get_page(&thread_current() -> frame_table, fault_page,
(e -> is_writable ? FRM_WRITABLE : 0) | FRM_ZERO);
  if (frm) {
  file_seek(e -> swap_file, e -> swap_offset);
  if (file_read (e -> swap_file, frm, e -> length) == (int) e -> length)
  return;
  frame_free_page(&thread_current()  -> frame_table, fault_page);
  }
  } else if (e) {
  /* Swapped out pages. */
  uint8_t * frm = frame_get_page(&thread_current() -> frame_table, fault_page,
(e -> is_writable ? FRM_WRITABLE : 0) | FRM_ZERO);
  if (frm) {
  swap_read(frm, e -> swap_offset);
  supp_page_remove(&thread_current()->supp_page_table, fault_page);
  return;
  }
  }
}
```

4.2.4 Frame Allocator with Swap

<Figure 2>와 같이 Frame을 할당하고, 이를 Page Table에 연결시키는 작업과, Frame이 부족할 경우 희생 Frame을 찾고, Swap에 쓰는 작업을 수행하도록 작성했다. 각 Frame에 대한 정보 역시 Per-Process Hash Table로 저장한다.
희생 Frame을 찾는 것은 Second Chance(CLOCK) Algorithm을 이용했다.
구현은 아래와 같다:

```
/* ... */
/* CLOCK page replacement algorithm implementation.
   It returns kernel page of evicted frame. */
static uint8_t*
find_victim(void)
{
    acquire_frame_lock();

    int loop_cnt = list_size(&frame_list) << 1;



    return NUL;
    /* Circulate whole list to find victim. */
```

```
 while (loop_cnt --> 0) {
     if (clock_hand == NULL || clock_hand == list_end(&frame_list)) {
         clock_hand = list_begin(&frame_list);
     }

     /* Find pte for this frame, then investigate accessed bit(&PTE_A). */
     struct frame_entry *e = list_entry(clock_hand, struct frame_entry,
vict_elem);

     if (e -> pinned) { /* pinned frame, do NOT evict at this time. */
         if (clock_hand != list_end(&frame_list))
             clock_hand = list_next(clock_hand);
         else
             clock_hand = NULL;
         continue;
     }

     lock_acquire(&e->holder->thread_page_lock);
     if (pagedir_is_accessed(e -> pagedir, e -> user_vaddr)) {
         /* Give second chance. */
         pagedir_set_accessed(e -> pagedir, e -> user_vaddr, false);
         lock_release(&e->holder->thread_page_lock);
         if (clock_hand != list_end(&frame_list))
             clock_hand = list_next(clock_hand);
         else
             clock_hand = NULL;
     } else {
         lock_release(&e->holder->thread_page_lock);
         /* Evict this. */
         unsigned offset = BITMAP_ERROR;
         if (clock_hand != list_end(&frame_list))
             clock_hand = list_next(clock_hand);
         else
             clock_hand = NULL;
         if (e -> writable)
             offset = swap_write(e -> kernel_vaddr);
         if (offset != BITMAP_ERROR || !e -> writable) {
             lock_acquire(&e->holder->thread_page_lock);
             list_remove(&e -> vict_elem);
             hash_delete(&e -> holder -> frame_table, &e -> all_elem);
             pagedir_clear_page(e -> pagedir, e -> user_vaddr);
             lock_release(&e->holder->thread_page_lock);
             uint8_t *kpage = e -> kernel_vaddr;
             if (e -> writable) {
```

```
                    if (supp_page_find(&e -> holder -> supp_page_table,
e -> user_vaddr) != NULL)
                        supp_page_remove(&e -> holder -> supp_page_table,
  e -> user_vaddr);
                    supp_page_insert(&e -> holder -> supp_page_table,
  e -> user_vaddr, NULL, offset,
  PGSIZE, false, true);
                }
                release_frame_lock();
                return kpage;
            } else {
                release_frame_lock();
                return NULL;
            }

        }

    }

    release_frame_lock();
    return NULL;
}
/* Allocate new frame(or evict other frames), and return. */
uint8_t*
frame_get_page(struct hash *frame_table, uint8_t* upage, enum frame_flags flags)
{
    acquire_frame_lock();

    enum palloc_flags pal_flags = PAL_USER;

    if (flags & FRM_ZERO) {
        pal_flags |= PAL_ZERO;
    }

    if (flags & FRM_ASSERT) {
        pal_flags |= PAL_ASSERT;
    }

    uint8_t* frame = (uint8_t*)palloc_get_page(pal_flags);

    struct frame_entry *item = NULL;
    if (frame == NULL) {
        /* Failed to allocate new frame.
           Use swap to resolve this situation! */
```

```
        frame = find_victim();

        if (frame == NULL) {
            release_frame_lock();
            return NULL;
        }

        if (flags & FRM_ZERO)
            memset(frame, 0, PGSIZE);

    }

    item = &pool[pool_ptr++];
    ASSERT(pool_ptr <= FRM_POOL_SIZE);
    ASSERT (item != NULL);
    lock_acquire(&thread_current()->thread_page_lock);
    item -> pagedir = thread_current() -> pagedir;
    item -> user_vaddr = upage;
    item -> kernel_vaddr = frame;
    item -> holder = thread_current();
    item -> writable = true && (flags & FRM_WRITABLE);
    item -> pinned = false;
    hash_insert(frame_table, &item->all_elem);
    list_push_back(&frame_list, &item->vict_elem);
    pagedir_set_page(item->pagedir, item->user_vaddr,
 item->kernel_vaddr, item->writable);
    lock_release(&thread_current()->thread_page_lock);

    release_frame_lock();
    return frame;
}
/* ... */
```

4.2.5 Memory Mapped File

거의 Demand loading of segments와 같이 구현했다.
Supplemental Page Table에 file의 offset과 Page의 user virtual address를 저장한 다음, 이를 처음 요청할 때 file에서 지정된 offset을 이용, 값을 읽어왔다. 이후 이렇게 로드된 값이 변경될 경우 file을 unmap할 때 다시 file로 값을 쓰도록 구현했다. 또한, processk 종료될 때에는 implicit하게 모든 mapped memory region을 unmap하도록 구현했다.
자세한 구현은 다음과 같다:

```
static int
mmap_handler (int fd, uint8_t *upage)
{
    lock_acquire(&mmap_access_lock);
    /* You can't map stdin/out, you can map only opened files. */
    if (fd < 2 || fd >= thread_current() -> last_fd) {
        lock_release(&mmap_access_lock);
        return -1;
    }

    int mmap_length = filesize_handler(fd),
        mmap_npages = (mmap_length + PGSIZE - 1) / PGSIZE;

    /* It must be page-aligned, and fit into the virtual address region. */
    if ((unsigned)upage != (unsigned)upage / PGSIZE * PGSIZE || upage + mmap_npages *
        lock_release(&mmap_access_lock);
        return -1;
    }

    /* It can't be mapped on stack region. */
    if (STACK_LIMIT <= upage && upage + mmap_npages * PGSIZE < PHYS_BASE) {
        return -1;
    }

    int left_length = mmap_length,
        offset = 0, backup_offset = 0;
    uint8_t *upage_ptr = upage;
    struct list_elem *e = find_file_from_thread(fd);
    struct fdesc* fdesc = NULL;

    /* zero-sized file, or non-existing file. */
    if (e == NULL || mmap_npages < 1 || mmap_length < 1) {
        lock_release(&mmap_access_lock);
        return -1;
    }
```

```
    fdesc = list_entry(e, struct fdesc, elem);
    backup_offset = file_tell(fdesc -> file);

    /* Check whether the memory region is assigned to another page or not. */
    int i;
    for (i = 0; i < mmap_npages; i++) {
        if (supp_page_find(&thread_current() -> supp_page_table, upage + PGSIZE * i))
            lock_release(&mmap_access_lock);
            return -1;
        }
    }

    /* Reached here because the memory regions are seems to be legal. */
    while (left_length > 0) {
        /* Push entries to the supplemental page table, one by one. */
        supp_page_insert(&thread_current() -> supp_page_table, upage_ptr, fdesc -> fi
                left_length >= PGSIZE ? PGSIZE : left_length, false, true);
        left_length -= PGSIZE;
        upage_ptr += PGSIZE;
        offset += PGSIZE;
    }
    file_seek(fdesc -> file, backup_offset);

    /* Push this record to the list. */
    struct mdesc *new_md = malloc(sizeof *new_md);
    new_md -> md = thread_current() -> last_md++;
    new_md -> file = fdesc -> file;
    new_md -> upage = upage;
    new_md -> length = mmap_length;
    list_push_back(&thread_current() -> maps, &new_md -> elem);

    lock_release(&mmap_access_lock);
    return new_md -> md;
}
void
munmap_handler (int md)
{
    lock_acquire(&mmap_access_lock);

    struct list_elem *e = find_map_from_thread(md);
    if (e == NULL) {
        lock_release(&mmap_access_lock);
        return;
    }
```

```
    struct mdesc *mdesc = list_entry(e, struct mdesc, elem);
    list_remove(e);

    int mmap_npages = (mdesc -> length + PGSIZE - 1) / PGSIZE,
        offset = 0, backup_offset = file_tell(mdesc -> file),
        left_size = mdesc -> length;
    uint8_t *upage_ptr = mdesc -> upage;

    /* Write contents back to the file. */
    lock_acquire(&file_access_lock);
    while (left_size > 0) {
        is_valid_user_addr(upage_ptr, true);
        file_write_at(mdesc -> file, upage_ptr, left_size < PGSIZE ? left_size : PGSI
        left_size -= PGSIZE;
        upage_ptr += PGSIZE;
        offset += PGSIZE;
    }
    file_seek(mdesc -> file, backup_offset);
    lock_release(&file_access_lock);

    free(mdesc);
    lock_release(&mmap_access_lock);
}
```

4.3 시험 및 평가내용

평가는 기본적으로 make check의 pass / fail로 평가했으며, 특정 문제를 해결하기 힘든 경우에는 특정 프로그램을 특정 option와 함께 시험해보도록 다음과 같은 커맨드를 이용했다.

```
$ make clean && make && cd build && pintos -v -k -T 600 --bochs --filesys-size=2
-p tests/vm/page-parallel -a page-parallel -p tests/vm/child-linear -a child-linear
--swap-size=4 -- -q -f run page-parallel
```

이번 채점에 들어가는 test들에 대한 pass / fail 결과는 다음과 같다. 이 평가 내용들은 각각 개발 목표의 네 부분, 그리고 생산성과 내구성으로 나눌 수 있다.

```
1a. Stack Growth - 생산성: 기능이 정상 작동하는가.
pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-stk-sc
pass tests/vm/pt-grow-pusha
pass tests/vm/pt-big-stk-obj

1b. Stack Growth - 내구성: 잘못된 접근에 대해 적절한 예외처리가 가능한가.
pass tests/vm/pt-grow-bad
```

2a. Frame Allocation with Swap + Supplemental Page Table - 생산성: 기능이 정상 작동하는가.
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-shuffle

1a + 2a. Frame Allocation with Swap + Supplemental Page Table + Stack Growth - 생산성: 기능이 정상 작동하는가.
pass tests/vm/page-merge-stk

1a + 3a. Frame Allocation with Swap + Supplemental Page Table + Memory Mapped File - 산성: 기능이 정상 작동하는가.
pass tests/vm/page-merge-mm

2b. Frame Allocation with Swap + Supplemental Page Table - 내구성: 잘못된 접근에 대해 적절한 예외처리가 가능한가.
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2

4.3.1 Comparison Between LRU Approximation Algorithms

어떤 LRU 근사 Algorithm이 더 좋은 성능을 보여주는가 궁금해서 강의자료에 나오는 두 가지 방법을 실험해보았다.
실험은 모두 Page-Parallel Test로 수행했다(사실 더 많이 수행했으나 코드와 결과가 유실되었다).

Second Chance(CLOCK) Algorithm Page-Parallel에서 3107 Page Fault

16 Reference Bit Algorithm Page-Parallel에서 3114 Page Fault

결론 적어도 Pintos OS의 Test들에서는 Second-Chance(CLOCK) Algorithm이 더 좋은 성능을 보여주었다.
따라서 16 Reference Bit Algorithm을 사용하던 코드를 Second Chance를 이용하도록 바꿈으로써 성능향상에 성공했다.

4.3.2 보건 및 안정

언제나 속도와 안정성을 가장 우선시해서 작성했다. 실제 Copy-On-Write와 Frame Sharing, 그리고 16 Reference-Bit Algorithm 등 수많은 최적화를 강의자료에서 보고 구현했으나, 모두 이론과는 다르게 실제 성능이 떨어지기만 해서 다시 제거한 다음, 최대한 단순한

설계를 통해 수행하는 Instruction 자체를 줄이기 위해 노력했다. 이러한 과정에서 몇 가지 Error가 발생했지만 해결했고, 이를 방지하기 위해 Lock을 도입한 다음, Dynamic Memory Allocation을 줄이기 위해 Frame Table Entry와 Supplemental Page Table Entry는 Pre-Allocate 해놓고 Pool에서 필요할 때마다 가져다 사용하게 해서 안정성과 속도 모두를 극대화했다.

5 기타

5.1 Page Directory

80386 Architecture에서는 Multi-Level Page Table을 사용한다(사실 Two-Level이라고 봐도 무방하다).
이중 가장 윗 단계의 Page Table은 Page Directory라고 불리며, 이러한 Page Directory의 각 Entry(PDE)는 다시 Page Table을 가리킨다. 그리고 다시 Page Table의 Entry(PTE)는 Offset과 함께 실제 Physical Address를 가리키게 된다.
여기서 딱 하나 단순한 Two-Level Page Table과 다른 점은, 언제나 동시에 단 한 개의 Page Directory만 Active해야 한단 것이다.

5.2 연구 조원 기여도

고창영 : 설계 및 코드 작성, 그리고 보고서(100%).

5.3 느낀점

`Virtual Memory`의 원리와 이론들, 그리고 `Memory Management Policy`들을 직접 구현하면서 실습할 수 있는 기화가 되어 유익했다.
특히 `LRU Approximation Algorithm`의 구현에서 최적화 방안과 더욱 효율적인 Algorithm을 찾는 등 많은 것을 배울 수 있었고 `Page Directory`와 같은 새로운 개념도 생각해볼 수 있어서 좋은 기회였다.
하지만 `test`가 획일적이고 적은 것과, `page-parallel`을 구현할 때 `cspro2`가 너무 느려서 `pass`하기 힘들었던 점이 아쉽다.
다음부터는 `Virtual Machine` 내부의 타이머를 이용해서 컴퓨터 성능에 따라 `Pass/Fail` 여부가 갈리지 않았으면 좋겠다.