Operating System (Fall 2014) Project 3
Virtual Memory

Jongwook Choi

Introduction

세 번째 pintos 프로젝트의 목표는 Virtual Memory를 구현하는 것이다. 크게 아래와 같은 세 가지 요구사항이 있다.

- **Paging**: 가상메모리로 user page와 frame page를 매핑하여, 메모리가 부족하더라도 frame eviction 및 swapping 을 통해 (swap이 허용하는 한) 더 많은 메모리를 사용할 수 있게끔 한다.
- **Stack Growth**: 스택 영역이 두개 이상의 page에 걸치게 되더라도, 메모리 에러가 나지 않고 자동으로 stack 영역에 할당된 page를 할당하여 stack이 자동으로 grow할 수 있도록 한다.
- **Memory-Mapped Files**: 파일을 가상 메모리에 매핑하여 file read/write 를 할 수 있게끔 memory-mapped file 스킴과 이에 관련된 시스템 콜 `mmap()`, `munmap()`을 구현한다.

이 보고서에서는 위의 3가지 요구사항을 중심으로, 각각 어떤 방식으로 설계하고 구현하였는지, 발생할 수 있는 문제점들은 무엇이며 이것들을 어떻게 해결하였는지를 간략하게 설명하기로 한다.

1. Paging

1.1. Data Structure

페이징과 가상메모리를 구현하기 위해, 몇 가지 자료구조의 구현이 필요하다. 구현한 것들은 (1) Frame Table, (2) Supplemental Page Table, (3) Swap Table 이다. 먼저 구현한 자료구조들과 연산을 간단히 설명한다.

(1) Frame Table

Frame이란 physical memory의 영역을 의미한다. 하나의 프레임은 `PGSIZE`의 크기를 갖고 모두 aligned 되어 있다. 페이징이 가능한 모든 메모리 영역은 frame table 을 통해 관리되어야 한다.

Frame table에 담기는 정보인 Frame table entry는 하나의 frame page에 대해 정확히 하나가 존재하며, 아래와 같은 정보들을 저장하고 있다.

```
/** Frame Table Entry */
struct frame_table_entry
  {
    void *kpage;               /* Kernel page, mapped to physical address */

    struct hash_elem helem;     /* see ::frame_map */
    struct list_elem lelem;     /* see ::frame_list */

    void *upage;               /* User (Virtual Memory) Address, pointer to page */
    struct thread *t;          /* The associated thread. */

    bool pinned;               /* Used to prevent a frame from being evicted, while it is acquiring some resources.
                                  If it is true, it is never evicted. */
  };
```

- `kpage` : Hash table에서의 키값이 되는 mapped frame의 kernel page 주소이다. PintOS에서는 physical address와 kernel page를 1:1 대응하므로 그냥 간단하게 `kpage`로 통일하여 physical address를 취급하기로 한다.
- `upage` : 해당 frame이 로드된 user page의 virtual memory 주소.
- `t` : 해당 frame을 로드한, owner 쓰레드를 지칭한다.
- `pinned` : Frame pinning 을 위해 필요하다. 아래에서 자세히 설명한다.
- `helem`, `lelem` : Frame hash table 및 linked list에 넣기 위해 필요한 element 객체.

Frame table은 당연히, global scope로서 하나만 존재한다. 지원하는 연산들은 아래와 같다.

- `void vm_frame_init()` : 초기화. 처음에 한번 불린다.
- `void* vm_frame_allocate(enum palloc_flags, void *upage)` : 유저 virtual address `upage` 에 대응되는 frame page를 하나 생성하여 page mapping을 수행하고, 생성된 page frame의 kernel address를 리턴한다.

- `void vm_frame_free(void*)` : page frame를 해제하는 역할을 한다. Frame table에서 해당 엔트리가 제거되고, 리소스(page)가 해제된다.
- `vm_frame_pin(), vm_frame_unpin()` : Pinning을 위해 필요하다. 아래에서 자세히 설명한다.

위 연산들의 구현을 위해서, 내부적으로 (`kpage` ↦ frame table entry)의 해쉬 테이블(`struct hash frame_map`)을 사용하였다. 이로써 키값인 `kpage`가 주어졌을 때 해당 frame 의 정보를 빠르게 lookup할 수 있다.

그 외에도 frame table 내부에 존재하는 모든 entry들을 linked list로 별도로 관리하는데(`struct list frame_list`), 이는 아래에서 설명할 frame eviction (clock) algorithm 을 위해서 필요하다. Frame table에 페이지가 추가되고 삭제되는 모든 연산에 대해 항상 데이터의 일관성을 보장할 수 있도록 구현하였다(straightforward). 또한 frame table에 접근하는 연산은 concurrency를 고려해야 하므로, frame table의 모든 연산은 lock을 잡아 critical section에서 수행될 수 있도록 하였다(`frame_lock`).

(2) (Supplemental) Page Table

Page table은 개념적으로는 user virtual address를 physical address로 매핑하는 역할을 한다. 유저 program 에서 virtual address를 통해 접근하면, 대응되는 frame의 메모리 영역을 접근하는 것과 같은 효과이다. PintOS 에서는 page table의 포맷 때문에, (user virtual) page에 관련된 모든 부가정보들을 저장하는 자료구조가 별도로 필요한데 이것이 바로 Supplemental Page Table (이하 SUPT) 이다.

SUPT는 쓰레드(process)마다 하나씩 생성되며, `uaddr` (user page) ↦ SPTE (supplemental page table entry)로의 매핑으로 이해하면 된다. SPTE는 아래와 같은 정보들을 저장한다.

```
enum page_status {
   ALL_ZERO,         // All zeros
   ON_FRAME,         // Actively in memory
   ON_SWAP,          // Swapped (on swap slot)
   FROM_FILESYS      // from filesystem (or executable)
};

struct supplemental_page_table_entry
  {
    void *upage;              /* Virtual address of the page (the key) */
    void *kpage;              /* Kernel page (frame) associated to it.
                                 Only effective when status == ON_FRAME.
                                 If the page is not on the frame, should be NULL. */
    struct hash_elem elem;

    enum page_status status;

    bool dirty;               /* Dirty bit. */

    // for ON_SWAP
    swap_index_t swap_index;  /* Stores the swap index if the page is swapped out.
                                 Only effective when status == ON_SWAP */

    // for FROM_FILESYS
    struct file *file;
    off_t file_offset;
```

```
    uint32_t read_bytes, zero_bytes;
    bool writable;
  };
```

- `upage` : 키값이 되는 것으로, user virtual address 를 저장한다.
- `status` : 해당 page의 상태를 나타낸다.
  - ON_FRAME : 프레임에 로드된 경우이다. 이 때 load된 frame의 주소는 `kpage`에 저장되어 있어야 하고, frame table에서는 `kpage`를 키로 하는 대응되는 frame table entry를 찾을 수 있다.
  - ON_SWAP : 현재 페이지가 frame에서 evict 되어 swap disk에 존재하는 경우이다. 이 때, swap disk의 어떤 곳에 저장되어 있는지를 `swap_index`에 저장하고, 이 값을 통해 추후 disk의 적당한 곳을 읽어들여 swap in을 할 수 있다.
  - ALL_ZERO : Page가 프레임에 로드되지 않았지만, 모든 내용이 0으로 채워진 상태를 의미한다.
  - FROM_FILESYS : Page가 역시 frame에 없지만, 그 내용이 파일시스템에서 로드되어야 하는 경우를 의미한다 (executable 또는 memory-mapped 등). 어떤 파일의 어디 offset부터인지는 `file`, `file_offset`, `read_bytes`, `zero_bytes` 등으로 알 수 있다. `writable`이 false인 경우에는 read-only page이다.
- `dirty` : page의 dirty bit. swapped out된 페이지의 dirty bit는 `pagedir_is_dirty()`를 통해 알아올 수 없기 때문에 dirty 여부를 별도로 저장하기 위해 필요하다. 자세한 것은 이것의 사용처인 memory mapped files 에서 설명한다.

SUPT의 활용에 대해서는 아래 Page Fault 처리 알고리즘에서 자세히 설명한다.

(3) Swap Table

Swap을 위한 연산들을 제공한다.

- `swap_index_t vm_swap_out(void *page)` : Swap-Out을 수행한다. `page`의 내용을 swap disk에 기록하고, 그 위치를 식별할 수 있는 `swap_index`를 반환한다.
- `void vm_swap_in(swap_index_t swap_index, void *page)` : Swap-In을 수행한다. `swap_index` 위치에서 single page를 읽어, `page`에 기록한다.
- `void vm_swap_free(swap_index_t swap_index)` : 해당 swap region을 그냥 버린다.

Swap disk의 전체 섹터를 PGSIZE로 나눈 개수가 가능한 swap slot의 수가 된다. Page size가 swap sector의 size보다 크기 때문에, 하나의 page를 swap에 저장하기 위해서는 `PGSIZE / BLOCK_SECTOR_SIZE` 개의 연속된 block이 필요하다. 어떤 swap slot이 available한지는 `bitmap` 자료구조를 이용하여 관리하며, 대응되는 block 과 page를 매핑시켜 swap in/out을 구현하는 것은 매우 straightforward 하므로 자세한 설명은 생략한다. (see commit b20fe2c9)

1.2. Swapping and Evicting

Frame allocation 시, page allocation에 실패한 경우는 메모리가 부족한 경우이므로 이 때에는 swapping을 해야 한다. 즉, 어떤 (frame) page를 swap disk로 swap out하고 그 빈 공간에 새로 페이지를 할당하는 것이다. 이 때 다음과 같은 일들이 일어난다.

- Evict 할 frame을 고른다. (second-chance clock algorithm)
- 해당 frame의 page mapping을 해제한다. 즉, 소유자 쓰레드의 pagedir 에서, upage 매핑을 제거한다.
- frame의 내용을 swap out 한다. (readonly일 때에는 filesystem 으로)
- 해당 frame을 frame table에서 제거한다 (page도 free된다)
- 그 후 page를 재할당하여, 새롭게 할당된 frame 을 테이블에 넣고 매핑을 처리한다.

Eviction될 프레임을 선택하기 위해서는, second-chance algorithm 을 사용하였다.

- victim pointer (`clock_ptr`)를 유지하여, frame table 의 원소들을 circular 하게 순회한다.
- reference bit 가 1 이면, reference bit를 0으로 설정하고 다음 entry로 넘어간다.
- reference bit 가 0 이면, eviction 대상으로 선택한다.

reference bit의 확인/설정을 위해서는, 간단히 해당 프레임에 매핑된 `upage`를 pagedir에서 확인하는 방법을 사용하였다. Reference bit 를 알기 위해서는 `pagedir_is_accessed()`를, reference bit를 0으로 설정하기 위해서는 `pagedir_set_accessed()` 를 사용하면 된다 (user program에서 page access시 내부적으로 reference bit가 1로 셋팅된다). 앞서 frame entry들을 linked list를 통해 관리하고 있으므로, 위 알고리즘을 효율적으로 구현할 수 있다.

```
/** Frame Eviction Strategy : The Clock Algorithm */
struct frame_table_entry* clock_frame_next(void);
struct frame_table_entry* pick_frame_to_evict( uint32_t *pagedir )
{
  size_t n = hash_size(&frame_map);
  if(n == 0) PANIC("Frame table is empty, can't happen - there is a leak somewhere");

  size_t it;
  for(it = 0; it <= n + n; ++ it) // prevent infinite loop. 2n iterations is enough
  {
    struct frame_table_entry *e = clock_frame_next();
    if(e->pinned) continue;
    else if( pagedir_is_accessed(pagedir, e->upage)) {
      // if referenced, give a second chance.
      pagedir_set_accessed(pagedir, e->upage, false);
      continue;
    }

    // OK, here is the victim : unreferenced since its last chance
    return e;
  }
  PANIC ("Can't evict any frame -- Not enough memory!\n");
}
```

1.3. Page loading: Page Fault Handler

로드되지 않은 페이지를 접근한 경우, 즉 `pagedir` 에 존재하지 않는 memory 영역에 접근하면 page fault가 발생한다. 이 때, SUPT에 page가 들어있었다면 잘못된 메모리 영역 접근이 아니고 swapped out 되어있거나 lazy-load 등으로 인해 frame가 로드되지 않은 것이므로 page를 다시 load하는 작업이 필요하다. 이 작업이 `vm_load_page()` 함수에 구현되어 있다.

1. SUPT entry를 확인한다. 없다면, 잘못된 메모리 접근이므로 fault후 kill된다.
2. 페이지의 내용을 저장할 frame page를 얻는다. 이 때 다른 프레임은 evict 될 수도 있다.
3. SUPT에 저장된 `status`를 보고, 알맞은 데이터를 메모리에 로드한다.
   - `ALL_ZERO` : 그냥 0으로 채운다(`memset`).
   - `ON_FRAME` : NO-OP. (can't happen?)[1]
   - `ON_SWAP` : swap in을 해야 한다. `vm_swap_in()`를 호출하여 데이터를 로드한다.
   - `FROM_FILESYS` : SPTE에 저장된 file 포인터 등의 정보를 통해 파일 시스템에서 내이터를 읽어온다. Offset `file_offset`에서부터 `read_bytes` 만큼 읽어오고, 남은 영역은 0으로 채울 수 있다.
4. `upage` 와, 2.에서 생성된 새로운 frame을 매핑한다 (`pagedir_set_page()`). 이제 이 frame의 상태는 다시 `ON_FRAME`이 된다.

이 핸들러를 통해 데이터를 채워넣어 page를 로드함으로써, Lazy load 로 표기되어 있는 page (e.g. `FROM_FILESYS`)를 resolving하는 것도 자연히 수행된다.

1.4. Process Loading 시 처리 (Lazy-Load)

위의 paging scheme 이 구현되었으므로, PintOS가 process를 로드할 때 모든 segment를 메모리에 load하는 대신 필요한 segment들만 lazy load하는 것이 가능해졌다. `load_segment()` (see `process.c:639`) 에서, 기존에는 할당된 page를 파일로부터 채우고 `install_page`를 바로 하던 반면, `vm_supt_install_filesys()`를 이용하여 lazy load 될 수 있도록 파일의 포인터(이 파일은 executable 이므로 프로세스가 종료되기 전까지는 그 포인터가 계속 유효하다)와 offset, size 정보를 SUPT에 기록해준다.

단, 최초의 user stack을 세팅할 때에는 (`setup_stack()`), lazy loading을 사용하지 않는다. 물론 최초의 stack 영역에 할당된 page 및 frame이 SUPT 및 frame table에 올바르게 적재될 수 있도록 적절한 처리가 필요했다.

[1] 아 물론, 이미 로드되어 있으므로 새로운 frame을 할당하지 않고 그냥 return.

1.5. Process Termination 시 처리

프로세스가 종료할 때, 이 프로세스가 점유하고 있던 pagedir가 destroy되는 과정에서 할당되었던 모든 page는 해제가 된다. 이와 더불어 SUPT 테이블도 같이 삭제를 해주어야 한다. SUPT 테이블은 per-thread scope 이므로 그냥 삭제를 하면 되는데, 만약 이 프로세스가 할당하고 있던 frame 이나 swap slot 이 있다면 이들도 적절한 해제를 해주어야 한다. 만약 해제를 해주지 않으면 (쓰레드가 free 됨), 나중에 evict할 frame을 골랐을 때 이미 pagedir 등 필요한 정보들이 다 소멸된 상태이므로 문제가 발생할 수 있다.

이 부분은 개념은 간단한데 구현이 다소 까다롭다. 자세한 것은 commit `eb9d8be0` 의 diff 및 message를 참조하자. 가장 중요한 포인트만 짚자면, 앞서 우리가 SPTE에 매핑된 프레임의 `kpage`를 저장해두었기 때문에 이를 key로 하여 frame table에서 더이상 유효하지 않은 프레임들을 삭제할 수 있다. 만약 page의 상태가 SWAPPED 였다면, 점유된 swap을 해제해버리면 된다. 추가로 memory mapped file이 있다면, 이에 대한 해제도 수행한다 (3. 에서 설명).

SUPT가 destroy 되는 과정이므로, hash table 의 모든 element 별로 수행되는 destructor 콜백을 이용하여 이 안에서 대응되는 frame entry를 제거 (`vm_frame_remove_entry()`) 하도록 처리하고 있다. 쓰레드 (프로세스) 가 종료되어 리소스가 해제될 때, `pagedir`에 연관된 모든 page들도 free 되는데 이 때 double-free가 일어나지 않도록 조심해야 한다.

1.6. Frame Pinning

User memory에 접근할 때, 커널이 가상메모리의 paging을 처리하고 있는데 또다른 page fault가 발생하면 문제가 생길 수 있다[2]. 예를 들어, write system call 을 수행하여 user memory를 읽어 file system에 write하는 도중 page fault가 발생하면 (user page가 swap된 상태) , swap in을 할 때 filesystem에 다른 access가 필요하다. 그런데 PintOS에서는 filesystem은 lock을 한 번만 잡고 들어갈 수 있으므로, double-lock에 의한 커널 패닉이 발생한다.

이를 해결하기 위해 frame pinning이라는 방법을 사용해야 한다. Pinned 된 프레임은 eviction 에서 제외하는 것이다. 구현 방법으로, frame table entry에 `pinned`라는 boolean 필드를 두어 pinned 여부를 기록하였고, `vm_frame_unpin()`, `vm_frame_pin()` 등의 함수로 특정 프레임의 pinned 비트를 설정할 수 있도록 구현하였다.

- 처음 frame을 할당할 때는 eviction이 일어나지 않게 하기 위해 pin 시키고, frame이 로드가 완료되면 (`vm_load_page()` 참고) pin을 해제한다.
- `read()` 및 `write()` 의 system call 을 수행하기 전에 앞서, 대상 buffer에 해당하는 모든 user page들을 pin한다. 이 동작은 `vm_pin_page()` 에 구현되었는데, page를 SUPT에서 lookup하여 만약 이것이 프레임에 로드되어 있다면 그 frame을 pin한다. SUPT에 없을 수도 있는데, lazy-load 를 사용하고 있기 때문에 아직 접근되지 않은 스택 영역이 있을 수도 있으므로 (stack growth) 무시한다.
- system call 수행이 완료된 이후에는, 앞서 pin했던 모든 프레임들의 pin을 해제한다. `vm_unpin_page()` 에 그 동작이 구현되어 있다.

[2] 'page-merge-mm', 'page-parallel' 과 같은 테스트 케이스 등.

2. Stack Growth

기존에는 user program의 스택이 하나의 page로만 이루어져 있었고, 이를 초과하면 page fault가 발생하였다. page가 허용하는 한도 내에서 stack이 grow할 수 있게 처리하는 것이 목표이다.

먼저, user program이 스택을 초과하면 page fault가 발생한다. 따라서 page fault가 발생했을 때, stack access인지를 판단하여 그럴 경우에는 page를 새로 할당한다. 물론, stack access가 아니고 잘못된 메모리를 access하는 경우도 있으므로 적절한 방법이 필요하다.

2.1. Stack Access 판별 방법

먼저 스택 포인터인 `esp`를 얻어온다. Page fault가 발생한 주소를 `fault_addr` 라고 할 때, 아래 두 가지가 모두 만족되면, 새로운 페이지를 할당하여 스택을 증가시켜야 한다.

- User memory의 스택 영역인지 : `PHYS_BASE - MAX_STACK_SIZE <= fault_addr && fault_addr < PHYS_BASE` (`MAX_STACK_SIZE`는 8MB로 정의하였다).
- 스택 프레임 위에 있는지 : `esp <= fault_addr` 또는 `fault_addr == f->esp - 4` 또는 `fault_addr == f->esp - 32`. 이는 매뉴얼에 따라, 80x86 PUSH/PUSHA operation 이 스택 포인터의 4바이트 밑 (PUSH) 또는 32바이트 밑 (PUSHA) 을 사용하기 때문이다.

그런데, `esp`를 얻어오는 것이 user mode와 kernel mode에 따라 약간 다르다. 만약 user mode라면 `intr_frame`에서 바로 얻어올 수 있지만 (`f->esp`), `read()` 혹은 `write()` 등의 시스템 콜 등을 수행하다가 stack 영역에서 page fault가 발생할 수도 있다. 앞서 Project 2에서, 잘못된 user memory 접근 여부를 page fault 에 기반한 방법으로 파악하였으므로 (Accessing User Memory 섹션 참고), 이 때에는 `f->esp`에 유저 프로그램의 stack pointer가 존재하지 않는다.

이 문제를 해결하기 위해, system call을 호출하는 시점에 유저의 `esp`를 `thread` 구조체에 저장한다. 그러면, page fault가 발생하여 interrupt handler로 들어오더라도 이를 바탕으로 `esp`를 알아낼 수 있다. 만약 (1) user mode면 `f->esp`를 그대로, (2) kernel mode면 쓰레드에 저장한 `curr->current_sep` 를 사용하면 된다.

```
@@ -118,4 +118,8 @@ struct thread
     struct list file_descriptors;       /* List of file_descriptors the thread contains */

     struct file *executing_file;        /* The executable file of associated process. */
+
+    uint8_t *current_esp;               /* The current value of the user program's stack pointer.
+                                           A page fault might occur in the kernel, so we might
+                                           need to store esp on transition to kernel mode. (4.3.3) */
 #endif
```

```
/* (4.3.3) Obtain the current value of the user program's stack pointer.
 * If the page fault is from user mode, we can obtain from intr_frame `f`,
 * but we cannot from kernel mode. We've stored the current esp
 * at the beginning of system call into the thread for this case. */
void* esp = user ? f->esp : curr->current_esp;
```

2.2. Stack 증가 시의 Paging 처리

앞서 판별한 조건이 만족되면, `fault_page` (`fault_addr`를 `PGSIZE`로 align한 page 시작 주소)를 알아내고 이곳에 페이지를 할당한다.

따라서 supplemental page table 에 `fault_page`를 user address로 하는 (새로운) entry를 추가한다. 추가되는 entry의 타입은 `ALL_ZERO` 로, 모든 data가 zero-filled 되어 있는 방식을 사용하였다 (see `vm_supt_install_zeropage()`). Linux System에서 초기화되지 않은 스택은 `0xcc` 등의 쓰레기값이 들어있지만, 여기서는 편의상 zero를 사용하였다.

이제 `vm_load_page()`를 통해 frame을 할당 후 페이지를 메모리에 로드하면, stack growth에 대한 인터럽트 핸들러의 처리는 끝이 난다.

```
// Stack Growth
bool on_stack_frame, is_stack_addr;
on_stack_frame = (esp <= fault_addr || fault_addr == f->esp - 4 || fault_addr == f->esp - 32);
is_stack_addr = (PHYS_BASE - MAX_STACK_SIZE <= fault_addr && fault_addr < PHYS_BASE);
if (on_stack_frame && is_stack_addr) {
  // OK. Do not die, and grow.
  // we need to add new page entry in the SUPT, if there was no page entry in the SUPT.
  // A promising choice is assign a new zero-page.
  if (vm_supt_has_entry(curr->supt, fault_page) == false)
    vm_supt_install_zeropage (curr->supt, fault_page);
}

if(! vm_load_page(curr->supt, curr->pagedir, fault_page) )
      goto PAGE_FAULT_VIOLATED_ACCESS;
```

자세한 구현은 commit 0703878를 참고하자.

3. Memory Mapped Files

Memory mapped files 기능을 위해서는 `mapid_t mmap(fd, addr)` 및 `munmap(mapid_t)` 시스템 콜이 구현되어야 한다. 이들의 구현은 굉장히 straightforward 하다.

먼저, `struct thread` 에는 해당 프로세스에 memory-mapped 된 descriptor 들의 정보를 저장한다.

```
@@ -127,6 +127,9 @@ struct thread
 #ifdef VM
     // Project 3: Supplemental page table.
     struct supplemental_page_table *supt;   /* Supplemental Page Table. */
+
+    // Project 3: Memory Mapped Files.
+    struct list mmap_list;              /* List of struct mmap_desc. */
 #endif

 /* Owned by thread.c. */
```

memory-map descriptor는 아래와 같은 정보를 저장해야 한다. Project 2의 file descriptor와 거의 판박이다. 파일과 그 크기, 매핑된 page 주소를 기억한다.

```
struct mmap_desc {
  mmapid_t id;
  struct list_elem elem;
  struct file* file;

  void *addr;   // where it is mapped to? store the user virtual address
  size_t size;  // file size
};
```

3.1. `mmap()`

`mmap()` system call의 구현은 아래와 같다. 역시 straightforward 하다.

1. argument를 체크한다. 매뉴얼의 specification에 따라, page address가 0인 경우, align되지 않은 경우, fd가 0 또는 1인 경우, 잘못된 메모리 영역인 경우, 기존의 다른 메모리 영역이 겹치는 경우 (SUPT에 페이지가 존재하는지로 검사) 등이 해당된다.
2. 파일을 열고, 크기를 검사한다. `file_reopen()`를 사용해야 file descriptor에서 관리하는 file 포인터와 별도로 파일을 관리할 수 있다. 여기서 열린 file은 당연히, `munmap()`에서 닫힌다.
3. `vm_supt_install_filesys()`를 이용하여, 페이지를 세팅한다(`FROM_FILESYS`). offset에 따라 어떤 영역을 읽어야 하는지를 `mmap_desc` 구조체에 저장해두면 된다. 이 때 마지막 페이지를 제외한 앞의 페이지들은 PGSIZE 바이트 만큼이 매핑되고, 마지막 페이지는 남은 bytes 들을 매핑해야 한다. 이제 그러면 해당 page는 file system으로부터 데이터를 lazy load할 수 있는 상태가 된다.

3.2. `munmap()`

`munmap()` system call의 구현은 아래와 같다. 역시 straightforward 하다.

1. 주어진 mapping id에 대응하는 `mmap_desc` 를 찾는다 (없으면 실패). 여기에 할당되어 있는 모든 page들을 순회하면서, `vm_supt_mm_unmap()` 를 호출하여 각 page별로 적절한 해제를 수행한다 (아래서 설명).
2. `vm_supt_mm_unmap()` 에서는 mapping된 각각의 page가 해제되면서, 수반되어야 하는 작업들이 수행된다.
   - pagedir 에서 user address와의 매핑을 해제한다 (later access는 모두 page fault).
   - page의 status에 따라 — frame table에서 제거하거나 (`ON_FRAME`), 할당된 swap slot을 버린다 (`ON_SWAP`).
   - 이 외에도, page가 dirty state 인 경우에 file에 page의 내용을 써주는 것이 필요하다. dirty state라면, 대응된 file 영역에 frame page의 내용 또는 swap의 내용을 기록해준다 (임시 page에 swap-in 한 뒤, 이것을 파일에 다시 기록).

Page별로 dirty 여부를 트래킹하고 있으므로 dirty state인지를 알아내기 위해 `pagedir_is_dirty()` 를 사용한다. dirty인지를 판단하는 조건은 (1) SPTE에 dirty로 마킹이 되었거나[3] (2) user page가 dirty 이거나 (3) kernel page (frame)가 dirty 이거나 (`pagedir_is_dirty()` 사용) 셋 중의 하나가 만족될 때이다 (user page는 frame page에 대한 일종의 aliasing 이기 때문에 이렇게 해야 한다).

프로세스가 종료될 때 (`process_exit()`), 열려진 파일을 모두 닫았던 것처럼 memory-mapped file이 있다면 이를 모두 해제하는 부분도 필요하다. 프로세스가 종료될 때, `munmap()` system call 핸들러를 호출함으로써 위의 과정 (특히, write back on the file)이 모두 일어날 수 있게 처리하였다.

Memory-mapped files에 관한 구현의 detail은 commit `754ce211` 및 `a6ebf11c` 의 diff 및 message를 참고하라.

[3] 물론 이를 위해서 frame이 evict 및 swap out될 때, dirty였는지 여부를 SPPE에 기록하고 있다. 기록하지 않으면 나중에 write back 되어야할지 여부를 pagedir의 정보만으로는 알 수 없기 때문이다. 추후 frame load가 되면 SPTE의 dirty flag는 리셋된다.

4. Test Results

Project 3의 요구사항인 Paging, Stack Growth, mmap 시스템 콜 등을 구현하였고, 109개의 모든 테스트 케이스를 통과하였다. 이들 테스트 케이스는 Project 2에서 완료하였던 user program, system call의 기본 기능과 더불어, stack growth / paging / mmap 의 기본 동작 및 예외 상황에서의 처리 여부, 그리고 여러 프로세스가 동시에 실행되는 복잡한 상황(parallel merge sort, 파일을 통해 communication) 등의 robustness를 검증한다.

```
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                    Pts Max  % Ttl  % Max
------------------------------------------- --- --- ------ ------
tests/vm/Rubric.functionality                55/ 55  50.0%/ 50.0%
tests/vm/Rubric.robustness                   28/ 28  15.0%/ 15.0%
tests/userprog/Rubric.functionality         108/108  10.0%/ 10.0%
tests/userprog/Rubric.robustness             88/ 88   5.0%/  5.0%
tests/filesys/base/Rubric                    30/ 30  20.0%/ 20.0%
------------------------------------------- --- --- ------ ------
Total                                               100.0%/100.0%
```

Conclusion

구현의 난이도가 상당한 편이었다. Paging / Frame 관련 연산이 다소 복잡하여, 커널 레벨의 API 를 깔끔하게 설계하는 것도 중요하고, race condition이 생기지 않도록 여러가지 경우를 고려하여 버그 없이 정확하게 구현하는 것도 필요하다. 다행히 여러 프로세스가 동시에 실행되는 등 여러가지 concurrency 상황에서도, 적절한 동기화와 lock 처리 등을 통해 virtual memory가 올바르게 동작하도록 구현할 수 있었다. 대부분은 어떤 것을 해야하는지 그 상세와 대략적인 방향만 정해지면 그 구현은 straightforward 한데, 작업 과정중에서 문서화(주석 및 commit message)를 잘 하였으므로 detail 한 내용들은 comment와 commit message를 참고하길 바란다.

일부는 약간 리팩토링이 필요한 부분들도 다소 있어 보이는데, 조금 더 robust하고 아름다운 설계를 처음부터 하고 구현을 시작했으면 하는 아쉬움도 남는다. 가상 메모리의 scheme과, 이를 올바르게 관리하기 위한 kernel의 동작 과정에 대해 이해할 수 있었고, 설계/구현/디버깅 등에 여러가지 어려움도 많았지만 즐겁고 유익한 시간이었다.