Design Report

20160116 김승환, 20160859 김요한

Backgrounds

1. Memory Terminology

현재 핀토스의 메모르는 stack, Initialized data, Uninitialized data, code의 4개의 세그먼트로 구성되어 있다.

현재 pintos의 Memory의 한계는 swap과 demand paging을 사용할 수 없고 virtual memory가 규현되어 있지 않다는 점이 있다.

- Pages

page는 데이터를 저장하고 검색하는 메모리 관리 기법으로 가상기억장치를 모두 같은 크기의 블록으로 편성하여 운용하는 기법이다. 이때 일정한 크기를 갖는 블록을 Page라고 부른다.

```
31                  11         0     Virtual address
+-------------------+----------+
|    page number    | offset   |
+-------------------+----------+
```

page의 크기는 프로세서 아키텍처에 의해 결정되는데, continuous region of virtual memory 4,096 bytes를 갖는다. page offset 는 virtual address의 마지막 12bit이다.

```
struct thread
  {
    ...

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

...
  };
```

`uint32_t *pagedir` 를 사용하여 구현해야한다.

- page fault

프로세스가 virtual address에 접근하려고 시도했을 때 대응되는 physical memory가 load 되지 않으면 page fault이다. pintos project 2에서는 page fault가 발생하면 bug로 간주했으나 project 3를 완성하고 나면 vm 시스템에서는 발생할 수 있게 된다.

- Frames

frame은 Physical memory를 일정한 크기로 나눈 블럭이며 frame과 page의 크기는 같다. 페이지가 하나의 프레임을 할당받으면, physical memory에 정보가 위치한다. 프레임을 할당받지 못한 페이지들은 외부 저장장치에 저장된다.

```
31                 11         0     physical address
+------------------+----------+
|    page number    | offset   |
+------------------+----------+
```

- Page Tables

페이지 테이블은 페이징 기법에서 사용되는 자료구조이다. 프로세스의 페이지 정보를 저장하는 테이블로 하나의 프로세스는 하나의 페이지 테이블을 지니며 Virtual page number를 Physical page number로 mapping하는 table이다.

- Swap slots

swap partition에 있는 disk space로 page-aligned 되어있다. (4KB)

2. Resource Management

Managing the Supplemental Page Table

모든 데이터를 물리메모리에 올려버리는 것은 비효율적이므로, 그 대신 실행 중에 해당 데이터가 필요한 순간이 오면 그 때 물리메모리에 올리는 방식을 택한다. 이를 구현하기 위해서는 페이지에 대한 데이터를 어딘가에 저장해두고 Page fault가 발생했을 때, 해당하는 데이터를 찾아서 메모리에 load해야한다. 이 공간을 Supplemental page table라고 한다.

Supplemental page table은 각 페이지의 추가적인 데이터로 기존의 page table을 보충하는 역할을 한다. 이것은 page fault가 발생했을 때, kernel이 오류가 발생한 virtual page를 찾아내어 어떤 데이터가 있는지 확인하기 위해 필요하다. 또한 kernel은 process가 종료 될 때, supplemental page를 참조하여 어떠한 resource를 확보할 지 결정한다.

Supplemental page table은 page fault handler와 많이 연결되어 작동한다. Page fault는 page가 file이나 swap에서 page를 가져올 때 발생하는데, 이를 처리하기 위해서는 page fault handler를 수정할 필요가 있다. 이를 수정하여 더 정교한 page fault handler를 만들어야 한다.

우선 Supplemental page table에서 page fault가 발생하는 page를 찾아서, 메모리 참조가 vaild한 경우 Supplemental page table을 이용하여 페이지에 있는 데이터를 찾는다. 여기서 이 페이지는 file system 안에 있거나, swap slot 또는 all-zero page일 수 있다. 만약, sharing을 구현할 경우, 페이지의 데이터는 이미 page frame에 구현되어 있을 수 있으나, page table에는 없다. Supplemental page table이 사용자 프로세스가 access하기 원하는 주소에 없는 데이터를 예상하거나 페이지가 kernel virtual memory에 있거나, access 권한이 읽기 전용 페이지에 write하는 경우에 해당 access는 invalid하다. Invalid access는 프로세스를 종료시키고 프로세스의 모든 resource가 해제된다.

페이지를 저장할 frame이 필요하다. sharing을 구현할 경우, 필요한 데이터가 frame에 이미 있을 수 있고, 그 frame을 찾는 것을 구현해야 한다. 파일시스템에서 읽거나, swap또는 zeroing을 통해 데이터를 frame으로 가져온다. sharing을 구현할 경우, 필요한 데이터가 이미 frame에 있을 수 있기 때문에 이 과정을 수행하지 않는다. Supplemental page table은 또한 faulting virtual address에 대한 page table entry를 가르킨다.

Managing the Frame Table

Frame table은 eviction policy의 효과적인 구현을 위해 필요하고 그것의 가장 중요한 역할은 unused frame을 얻는 것이다. 즉, 메모리 상에서 어떤 부분이 비어있으며 어디가 채워져 있는지를 관리하는 역할이 Frame table이다. 메모리가 꽉 차있는 경우 어떤 것을 evict 시킬지 결정하는 것도 Frame table에서 이루어진다.

Frame table은 user 페이지를 가지고 있는 각 frame마다 하나의 entry를 가진다. Entry는 페이지의 포인터를 저장하고 있으며, 데이터가 있으면 그 데이터를 점유할 수도 있다. Eviction policy는 Frame table에 의해 효과적으로 수행되며, 이것은 모든 frame이 자유롭지 않을 경우 해당 페이지를 선택함으로서 가능하다. user 페이지를 위해 쓰이는 frame들은 반드시 user pool에서 가져와야하고

palloc_get_page(PAL_USER)을 통해 불러온다. PAL_USER을 사용하여 kernel pool로부터 frame을 배정받는 것을 피하는 것이 중요하다.

Frame table은 unused frame을 획득하는데, 하나 이상의 frame이 free할 경우 이 과정은 쉽게 이루어진다. 다만, 모든 frame이 free하지 않으면 원래 frame을 evicting으로 free하게 만들고 나서 frame을 획득해야한다.

Managing the Swap Table

물리메모리가 가득 찬 경우, 페이지들 중 하나를 빼내고 새로운 페이지를 넣어야한다. 이 때 꺼내지는 페이지는 Swap out된다고 부르고, 새로 물리메모리에 load되는 페이지를 Swap in된다고 부른다.

페이지의 종류는 Uninit, Anon, File 이렇게 3가지 종류가 있다. Uninit은 아직 초기화가 안된 페이지를 의미하며 페이지 정보는 생성되었지만 user가 아직 그 페이지에 접근하지 않은 것이다. 물리메모리의 페이지는 uninit일 수 없다. File은 mmap된 페이지로 Swap in되는 경우 disk에서 직접 읽어와야하고, Swap out되는 경우에는 직접 disk에 다시 write해야한다. Anon은 stack과 text를 포함하며, Swap out되는 경우 디스크로 다시 돌아갈 수 없고, 제거하면 다음에 다시 접근할 수 없기 때문에 물리 disk에 swap disk란 공간을 따로 할당하여 관리해준다.

Swap table의 역할은 swap disk의 정보를 포함하는 것으로 bitmap을 사용하여 구현할 수 있다. Swap table은 in-use와 free swap slot을 track한다. Frame에서 swap 파티션으로 페이지를 제거하기 위해 사용하지 않는 swap slot을 선택할 수 있어야 한다. 페이지를 다시 읽거나 스왑된 프로세스가 종료 되는 경우 swap slot을 종료시킨다.

Managing Memory Mapped Files

Memory mapped file은 파일을 다루기 위해 존재하는데, 이것을 통해 process의 virtual memory address space에 파일을 mapping하고 virtual memory address에 직접 접근해서 file read, write를 대신한다. 또한 운영 체제 내에서 파일을 다루는 모든 곳에서 사용이 가능하다.

Memory mapped file은 메모리에 mapping된 파일을 통해 메모리를 찾을 수 있어야 한다. mapping된 곳의 page fault를 정확히 처리하고 mapping 된 파일은 process의 다른 segment들에 중복되지 않아야 하기 때문이다.

Memory mapped file은 file_backed mapping이며, page fault가 일어났을 경우 phy frame이 즉시 할당되고 content는 파일에서 메모리로 옮겨진다. mmp가 unmap되거나 swap out되는 경우 해당 변경사항은 파일에 반영되어야 한다.

Requirements

1. Frame Table

현재로는 frame table이 존재하지 않기 때문에 kernel virtual 메모리와 물리 메모리에 매핑되어 있지 않다. 따라서 frame table과 frame을 구현하여 물리 프레임이 가상 메모리 페이지를 통해 접근할 수 있게 한다. Frame table은 각 프레임 당 하나의 entry를 가진다. Frame table의 entry는 페이지를 가르키는 포인터를 포함하고, 프레임이 부족할 때는 제거할 페이지를 결정하여 효율적으로 동작하도록 구현한다.

구현 :

구조체 frame을 추가한다.

```
//vm/frame.h
struct frame {
    void *kva;
    struct page *page;
    struct list_elem frame_elem;
};
```

frame_table 변수를 선언해주고, frame을 관리할 함수들을 추가한다. user pool로부터 페이지를 받아와서 프레임을 할당해주는 함수인 get_frame(), 할당되어 있는 frame을 제거해주는 함수인 evict_frame(), 가상 주소와 물리 주소를 mapping시켜주는 함수인 claim_page()를 추가해준다.

```
//vm/frame.c
struct list frame_table;

...

static struct frame * get_frame (void) {
    /*palloc_get_page()에 PAL_USER을 넣어서 사용*/
    /*user pool로부터 페이지를 못 받았을 경우, 메모리 공간을 얻기 위해 할당되어 있는
frame 제거*/
    /*list_push_back을 이용하여 frame_table에 element 넣어주기*/
}

static struct frame *evict_frame (void) {
    /* for문을 이용해 frame_table을 돌면서 해당하는 frame 제거 */
}

bool claim_page (void *va UNUSED) {
    /*user page가 이미 mapping 되었거나 메모리 할당 실패시 false return*/
}
```

기존의 방식인 물리페이지를 할당하고 mapping하는 것을 하지 않기 위해 load_segment()를 수정

```
static bool load_segment(struct file *file, off_t ofs, uint8_t *upage, uint32_t
read_bytes, uint32_t zero_bytes, bool writable)
{
 ...
    /*물리페이지를 할당하고 mapping하는 부분 삭제*/
 ...
}
```

2. Lazy Loading

프로그램을 실행하기 위해서는, 실행에 필요한 데이터를 물리메모리에 올려야 한다. 지금은 load(), load_segment() 함수에서 실행에 필요한 모든 데이터를 물리메모리에 올리고 있다. 그리고 page fault가 발생하면 이 상황을 잘못된 접근 오류로 간주하여 프로그램 실행이 중지된다.

구현 :

이러한 방식은 매우 비효율적이기 때문에 실행 중에 필요한 데이터가 있을 경우에 그 때 물리메모리에 올리는 방식으로 바꿔주어야 한다. 그래서 일단 프로세스가 시작될 때 메모리를 할당하기 위해 모든 데이터를 올리는 것이 아닌 스택 부분만 올려준다. 또한 할당된 페이지에서 page fault가 발생하면 해당하는 페이지를 메모리에 로드해준다. Page fault handler는 이 과정이 종료되면 작업이 재개되도록 구현한다.

이를 구현하기 위해 새로운 자료구조인 vm_entry를 정의

```
//vm/page.h
struct vm_entry{
    uint8_t type;
    void *vaddr;
    bool writable;

    bool is_loaded;
    struct file* file;

    size_t offset;
    size_t read_bytes;
    size_t zero_bytes;


    struct hash_elem elem;
}
```

load_segment()를 수정하여 가상메모리 자료구조를 초기화시키는 코드를 추가하고, vm_entry를 활용하도록 구현한다.

```
static bool load_segment(struct file *file, off_t ofs, uint8_t *upage, uint32_t
read_bytes, uint32_t zero_bytes, bool writable)
{
 ...
    /*malloc을 이용하여 vm_entry 생성 및 맴버들 설정*/
    /*vm_entry를 hash table에 추가*/
 ...
}
```

3. Supplemental Page Table

현재 Supplemental page table은 현재 구현되어 있지 않고, 기존의 page table만 존재한다. 그래서 디스크로부터 모든 세그먼트들을 물리메모리로 읽어들이기 때문에 비효율성과 낭비를 유발하고 있는 상태이다.

구현 :

위의 Lazy loading을 구현하기 위해서는, 각 page에 대한 정보를 어딘가에 저장할 필요가 있다. 그래서 page fault가 발생했을 때, 이 정보를 찾아서 메모리에 load를 해주어야한다. 이 page의 정보를 저장하는 것이 Supplemental page table의 역할이다. Supplemental page table는 page에 대한 정보를 찾는 시간을 최소화하기 위해서 hash table을 사용하여 구현한다.

Process마다 가상 주소 공간이 할당되므로 그것을 관리하기 위한 hash table을 정의한다.

```
struct thread{
...
struct hash vm;
}
```

이 구조체를 관리할 수 있는 함수들을 추가해줘야한다. hash table을 초기화시키는 함수 vm_init, hash 요소들의 vaddr을 비교하는 함수 vm_less_func, hash요소의 vaddr값을 이용하여 hash 값을 반환하는 함수 vm_hash_func을 추가해준다.

```
//vm/page.c
void vm_init(struct hash *vm)
{
    /* hash_init()을 이용하여 hash table 초기화*/
}
static unsigned vm_hash_func(const struct hash_elem *e, void* aux)
{
    /* hash_entry()를 이용하여 구조체 얻음. */
    /* hash_int()를 이용하여 vaddr에 대한 해시값을 return.*/
}
static bool vm_less_func(const struct hash_elem *a, const struct hash_elem *b)
{
    /* hash_entry()를 이용하여 각 element에 대한 구조체를 얻은 후 vaddr 비교 */
}
```

hash table에 element를 삽입, 제거하는 함수도 추가

```
//vm/page.c
bool insert_element(struct hash *vm, struct vm_entry *vme)
{
    /* hash_insert()함수를 사용하여 element를 삽입*/
}
bool delete_element(struct hash *vm, struct vm_entry *vme)
{
    /* hash_delete()함수를 사용하여 element를 제거*/
}
```

기존의 page_fault는 주소가 valid한지 검사 후 오류가 발생했을 때 segmentation fault를 발생시키고 종료되는 방식이다. 그래서 page_fault를 수정하여 vm_entry를 검색하고 해당하는 페이지를 할당하도록 구현한다.

```
static void page_fault(struct intr_frame *f)
{
    /*기존 방식에서 필요한 부분 삭제*/
    /*page_fault가 일어난 주소를 이용하여 vm_entry 구조체 탐색*/
    /*얻어낸 vm_entry를 이용하여 물리메모리 할당*/
    /*물리페이지와 가상페이지 mapping*/
}
```

4. Stack Growth (5 points)

현재 pintos의 스택 크기는 4KB로 고정되어 있어서 현재 스택의 크기를 초과하는 주소에 접근이 발생하면 잦은 버그가 발생한다. 이를 해결하기 위해서 유효한 접근을 판별하기 위해 유저 프로그램의 스택 포인터가 가리키는 주소값을 알아내어야 하도록 코드를 수정해야한다. stack growth 를 해결하기 위해서는 page fault를 잘 다루어야 한다.

- **New implementation (5 points):**

```
//in process.c
bool expand_stack(void *addr){
    //parameter로 받은 addr 주소를 포함하도록 스택을 확장한다
    //최대 8MB까지 확장이 가능하다
}
```

```
//in process.c
bool handle_mm_fault(struct vm_entry *vme){
    //vme가 이미 load이면 false를 return 한다.
    //vme의 type을 보고 vme type이 스택 확장이면 스택을 확장한다.
}
```

```
static void page_fault (struct intr_frame *f)
{
    ...
    //fault_addr를 확인해서 stack 영역을 확인하고 expand_stack 호출
}
```

```
bool expand_stack(void* addr){
    //stack 이 full인지 검사하고 full이면 return false한다.
    //alloc_page()를 통해 메모리 할당 뒤 vm_entry를 할당하고 초기화한다.
    //install_page()를 호출하여 페이지 테이블을 설정한다.
}
```

```
bool verify_stack(void *sp){
    //sp 주소가 존재할 시 handle_mm_fault() 함수 호출한다.
}
```

5. File Memory Mapping (5 points)

현재 핀토스는 mmp가 구현되어 있지 않는다. mmp를 구현하는 것이 이 과제의 목적이며 mmp는 page fault가 발생할 때 frame을 즉시 할당하고 그 내용은 file에서 memory로 보낸다. mmp가 unmap되거나 swap out되면 변경 사항은 file에 반영해야한다.

```
mapid_t mmap (int fd, void* addr)
void munmap (mapid_t mapping)
```

위 두 개의 함수를 구현하는 것이 목적이다.

· **File memory mapping (5 points)**:

```
struct mmap_file {
    int mapid;
    struct file* file;
    struct list_elem elem;
    struct list vme_list;
};
```

struct of `mmap_file`

```
mapid_t mmap (int fd, void* addr){
    //fd를 addr에 mapping을 성공하면 mapping id를 return 하고 실패하면 -1을 리턴한다.
    //file data를 메모리에 로드한다.
}
```

```
void munmap (mapid_t mapping){
    //mmap_list내에서 mapping의 id를 갖는 모든 virtual memory entry을 해제
    //인자로 넘겨진 mapping 값이 CLOSE_ALL인, 경우 모든 파일 매핑을 제거
    //페이지 테이블에서 엔트리 제거
}
```

```
void do_munmap(struct mmap_file* mmap_file){
   //mmap_file의 vme_list에 연결된 모든 virtual memory entry들을 제거
    //페이지 테이블 엔트리 제거
    //virtual memory entry가리키는 가상 주소에 대한 물리 페이지가 존재하고,
    //dirty이면 디스크에 메모리 내용을 기록한다.
}
```

6. Swap Table (5 points)

victim으로 선정된 page가 process의 데이터 영역이나 stack에 포함될 때 swap 영역에 저장한다. swap-out된 페이지는 다시 메모리에 load해야한다.

· **Swap table (5 points):**

```
struct vm_entry{
...
    size_t swap_slot; /* swap slot 추가*/
...
}
```

```
void swap_init(size_t used_index, void* kaddr){
    //swap 영역을 초기화한다.
}
```

```
void swap_in(size_t used_index, void* kaddr){
    //used_index의 swap slot에 저장된 데이터를 kaddr로 복사한다.
}
```

```
size_t swap_out(void *kaddr){
    //kaddr의 주소에 있는 페이지를 swap partition에 넣어준다.
    //page를 기록한 swap slot의 number를 retrun 한다.
}
```

7. On Process Termination (5 points)

프로세스가 종료될 때 리소스를 사용하는 모든 할당을 종료시켜주어야 한다.

· **On process termination (5 points):**

```
void process_exit (void){
    struct thread *cur = thread_current();
    uint32_t *pd;
...
    /* 리소스를 사용하는 모든 할당을 제거한다.*/
    vm_entry(&cur->vm);
    pd = cur->pagedir;
    ...
}
```