Virtual Memory 과제 보고서

운영체제 (ELE-3021 12781)

교수명 : 원유집

조교명 : 오준택

학생명 :

서창범 2014004648, 서문은지 2016025487

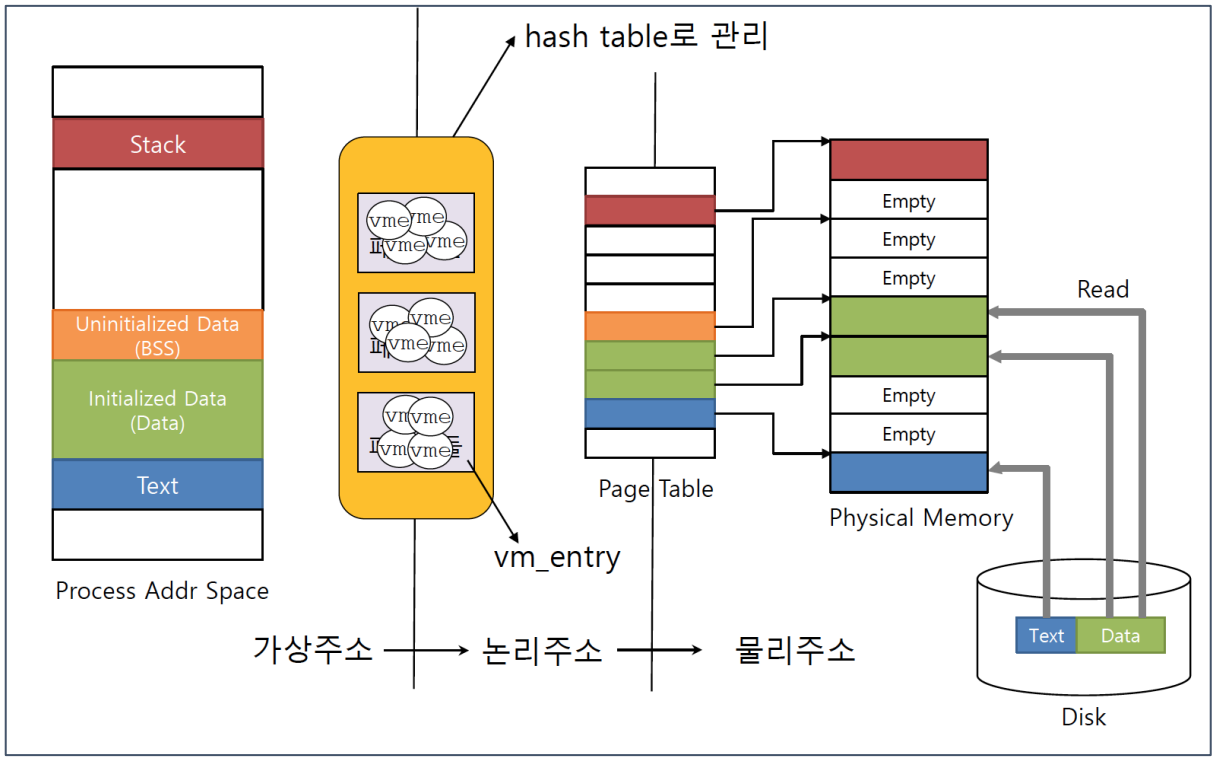
제출일자 : 2018 년 6 월 1 일

Virtual Memory

- virtual Memory

< vm_entry >

(1) 과제 설명



현재, 핀토스에서는 2단계 페이징, 가상 메모리를 사용하고 있는데, 실행 파일이 메모리에 탑재될 때 프로세스 전체의 모든 세그먼트에 대해 물리페이지가 할당된다. 주소공간과 각 페이지의 물리적 주소가 모두 매핑된다. 프로세스 실행에 필요한 가상주소 공간이 모두 물리 페이지에 매핑되어 있으므로 현재 페이지 폴트는 비정상적인 메모리 접근에 대해 예외 처리를 해주는 역할만 하고 있다(세그멘테이션 오류에 대한 처리).

위와 같은 경우, 프로세스가 자신이 사용하지 않을지도 모르는 메모리까지 사용하기 때문에 메모리 낭비가 극심하다.

그래서, 모든 페이지를 한번에 다 할당 해주지 않고, 요청한 페이지에 대해서만 물리페이지를 할당한다. 그렇기 때문에, 페이지 폴트 발생시 프로그램을 강제 종료를 하는 것이 아니라, 해당 vm_entry의 존재 유무를 확인하여 vm_entry의 가상주소에 해당하는 물리페이지를 할당하고, vm_entry의 정보를 참조하여, 디스크에 저장되어 있는 실제 데이터를 로드한다.

vm_entry는 가상 주소 공간에서의 페이지를 위한 자료구조이다.

1. 해당 페이지에 로드할 파일 포인터, 오프셋, 크기를 저장한다.
2. 프로그램 실행 시, 가상 주소 공간의 각 페이지에 vm_entry를 할당한다.
3. 페이지 폴트 발생 시, 가상 주소에 해당하는 vm_entry를 탐색한다.

(vm_entry가 없는 경우, segmentation fault를 발생시키며 종료하고, vm_entry가 존재할 경우, vm_entry에 있는 파일포인터, 읽기 시작할 오프셋, 읽어야 할 크기 등을 참조해서 물리페이지를 할당하고 물리메모리에 로드 한 후, 물리 주소와 mapping시킨다.)

이 과제를 수행하고 나면, 가상주소 공간에 매번 물리메모리를 할당하는 대신, 가상 페이지마다 vm_entry만 할당 해놓는다. 그리고 메모리 접근이 일어나 페이지 폴트가 발생하게 되면 그 때 물리 메모리를 할당한다. vm_entry는 스레드의 멤버인 vm hash테이블을 이용하여 관리한다.

예를 들어, 0x0804bec0 가상주소를 접근하려고 한다면 0x0804b000번 페이지가 요청된다. 하지만, 해당 VPN에 해당하는 물리 페이지는 할당이 되어있지 않기 때문에(매핑 또한 되어있지 않음) 페이지 테이블 엔트리의 present bit는 0이다. 그래서 페이지 폴트가 발생하게 된다.

그러므로, page fault handler에 의해 물리페이지를 할당해준 후에, 물리 메모리에 File 내용을 탑재 하여야 한다. 그러기 위해서는 파일에 대한 포인터와, offset, 읽어야 할 데이터의 크기 등 정보를 vm_entry에 저장할 필요가 있다.

또한, 가상 주소 페이지를 3가지 타입으로 분류해 놓았기 때문에, vm_entry의 type 필드에는 실행파일인지, 일반 파일인지, anonymous(매핑된 파일이 없는 경우)인지를 지정해주어야 한다. (VM_BIN, VM_FILE, VM_ANON)

vm/page.h

```
#ifndef VM_PAGE_H
#define VM_PAGE_H

#include <hash.h>
#include <list.h>
#include "threads/thread.h"
#include "filesys/file.h"
#include "threads/palloc.h"

#define VM_BIN  0
#define VM_FILE 1
#define VM_ANON 2
```

<vm_entry type 매크로 상수 정의>

```
21 struct vm_entry {
22   uint8_t type;              /*  VM_BIN, VM_FILE, VM_ANON의 타입 */
23   void *vaddr;               /*  vm_entry의 가상페이지 번호 */
24   bool writable;             /*  True일 경우 해당 주소에 write 가능
25                                  False일 경우 해당 주소에 write 불가능 */
26   bool is_loaded;            /*  물리메모리의 탑재 여부를 알려주는 플래그 */
27   struct file *file;         /*  가상주소와 맵핑된 파일 */
28
29   /*  Memory Mapped File 에서 다룰 예정 */
30   struct list_elem mmap_elem;   /*  mmap 리스트 element */
31
32   size_t offset;                /*  읽어야 할 파일 오프셋 */
33   size_t read_bytes;            /*  가상페이지에 쓰여져 있는 데이터 크기 */
34   size_t zero_bytes;            /*  0으로 채울 남은 페이지의 바이트 */
35
36   /*  Swapping 과제에서 다룰 예정 */
37   size_t swap_slot;             /*  스왑 슬롯 */
38
39   /*  'vm_entry들을 위한 자료구조' 부분에서 다룰 예정 */
40   struct hash_elem elem;        /*  해시 테이블 Element */
41 };
42
```

<vm_entry 구조체 정의>

< vm_entry 관리를 위한 자료구조 >

1. 과제설명

핀토스는 vm_entry들의 관리를 위한 체이닝 해시 테이블을 제공한다. 해시 값이 충돌할 경우, 충돌한 해시 값의 element들을 각 버킷에 가상주소 기준으로 정렬된 리스트로 관리한다.

- 프로세스 생성시, 해시 테이블을 초기화 하고, 프로세스의 가상페이지들에 대한 vm_entry들을 생성한다. 생성한 vm_entry들을 해시 테이블에 추가한다.
- 프로세스 실행 중, 페이지 폴트가 발생했을 때, 페이지 폴트가 발생한 주소에 해당하는 vm_entry를 해시 테이블에서 탐색한다.
- 프로세스 종료 시, 해시 테이블의 버킷리스트와 vm_entry를 위한 자료구조를 해제한다.

2. 함수 구현 및 수정

(1) thread 구조체에 해시 테이블 자료구조 추가

프로세스마다 가상주소 공간이 할당되므로, 가상 페이지들을 관리할 수 있는 자료구조인 해시테이블을 struct thread 안에 정의하여야 한다.

threads/thread.h

```
/* 스레드가 가진 가상 주소 공간을 관리하는 해시테이블 */
struct hash vm;

/* mmap으로 물리 메모리에 매핑 시킨 파일 목록을 관리하는 리스트 */
struct list mmap_list;
int next_mapid;
```

(2) 해시 테이블 초기화 및 제거 및 해시 함수 구현

vm/page.c

```
27 void vm_init (struct hash *vm) {
28
29   /* hash_init() 로 해시테이블 초기화 */
30   /* 인자로 해시 테이블과 vm_hash_func과 vm_less_func 사용 */
31   hash_init (vm, vm_hash_func, vm_less_func, NULL);
32 }
```

각 스레드의 가상주소 공간 정보(vm_entry)를 저장할 해시테이블을 스레드 별로 따로 관리할 필요가 있다. vm 해시테이블은 thread 구조체에 멤버로 추가되어 있으며, 스레드 초기화시 vm_init() 함수도 호출되어 해시테이블도 초기화 된다.

userprog/process.c

```
201 start_process (void *file_name_)
202 {
203   char *file_name = file_name_;
204   char **argv;
205   int argc;
206   struct intr_frame if_;
207   bool success;
208
209   // 인자의 개수를 셈.
210   argc = _get_argc(file_name);
211
212   /* file_name에 프로그램 이름과 인자들을 각각 구분하여 잘라
213      문자열 포인터 배열을 반환함.
214      내부적으로 malloc을 사용하여 메모리 동적 할당을 하기 때문에
215      사용을 마치고 나면 반드시 리턴 값에 대해 free를 해줘야함.  */
216   argv = _get_argv(file_name);
217
218   /* vm_init() 함수를 이용하여 해시테이블 초기화 */
219   vm_init (&thread_current ()->vm);
220
221   /* mmap_list 초기화 */
222   list_init (&thread_current ()->mmap_list);
223
224   /* Initialize interrupt frame and load executable. */
225   memset (&if_, 0, sizeof if_);
226   if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
227   if_.cs = SEL_UCSEG;
228   if_.eflags = FLAG_IF | FLAG_MBS;
229   success = load (file_name, &if_.eip, &if_.esp);
230
```

<start_process에서 vm_init()으로 해시테이블을 초기화 한다>

vm/page.c

```
34 static unsigned vm_hash_func (const struct hash_elem *e, void *aux) {
35
36   /* hash_entry()로 element에 대한 vm_entry 구조체 검색*/
37   struct vm_entry *ve = hash_entry (e, struct vm_entry, elem);
38
39   /* hash_int()를 이용하여 vm_entry의 멤버 vaddr에 대한 해시값을 구하고 반환 */
40   return hash_int (ve->vaddr);
41 }
42
44 static bool vm_less_func (const struct hash_elem *a, const struct hash_elem *b) {
45
46   /* hash_entry()로 각각의 element에 대한 vm_entry 구조체를 얻은 후 vaddr값 비교.
47      b가 크다면 true 반환.*/
48   struct vm_entry *ve_a = hash_entry (a, struct vm_entry, elem);
49   struct vm_entry *ve_b = hash_entry (b, struct vm_entry, elem);
50
51   return ve_a->vaddr < ve_b->vaddr;
52 }
```

각 해쉬테이블에는 해쉬 연산을 할 해쉬 함수와 해쉬 값이 같은 element 들에 대해 가상주소 기준으로 각 버킷에 정렬하여 저장하기 위한 정렬함수가 필요하다. vm_hash_func()함수는 hash_int()함수를 이용하여 가상주소 4바이트(int)에 대한 해쉬 연산을 수행하고, vm_less_func() 함수는 각 vm_entry를 연결하는 hash_elem으로 각각의 해당하는 vm_entry의 vaddr멤버를 비교하여 bool 값으로 리턴한다.

이 두 함수는 해쉬테이블이 초기화 될 때 인자로서 함수 포인터가 넘겨진다.

(4) 해시 테이블에 element를 삽입 및 제거 함수 구현

vm/page.c

```
54 bool insert_vme (struct hash *vm, struct vm_entry *vme) {
55   /* hash_insert() 함수 이용하여 vm_entry를 해시 테이블에 삽입 */
56   return !hash_insert (vm, &vme->elem);
57 }
```

insert_vme()는 인자로 받은 해쉬테이블에 vm_entry를 삽입하는 함수이다.

```
59 bool delete_vme (struct hash *vm, struct vm_entry *vme) {
60   /* hash_delete() 함수를 이용하여 vm_entry를 해시 테이블에서 제거 */
61   return !!hash_delete (vm, &vme->elem);
62 }
```

delete_vme() 함수는 insert_vme()함수와는 반대 작업인 해당 vm_entry를 해시 테이블에서 찾아 제거해주는 함수이다.

(5) 해시 테이블 내 vm_entry 검색 함수 구현

vm/page.c

```
64 struct vm_entry *find_vme (void *vaddr) {
65   struct hash *vm = &thread_current ()->vm;
66   struct vm_entry vme;
67
68   /*  pg_round_down()으로 vaddr의 페이지 번호를 얻음 */
69   vme.vaddr = pg_round_down (vaddr);
70
71   /*  hash_find() 함수를 사용해서 hash_elem 구조체 얻음 */
72   struct hash_elem *elem = hash_find (vm, &vme.elem);
73
74   /*  만약 존재하지 않는다면 NULL 리턴 */
75   if (!elem) {
76     return NULL;
77   }
78
79   /* 존재 하면  hash_entry()로 해당 hash_elem의 vm_entry 구조체 리턴 */
80   return hash_entry (elem, struct vm_entry, elem);
81 }
82
```

find_vme() 함수는 가장주소를 인자로 받아 해당 스레드의 가상주소에 해당하는 vme_entry를 해쉬테이블에서 찾아서 반환해주는 함수이다. 인자로 넘겨진 가상주소를 page 크기 단위로 round_down하여 해쉬 테이블에 같은 가상페이지 넘버(VPN)의 vm_entry가 있는지 찾고 해당 vm_entry의 hash_elem을 찾으면 vm_entry를 반환하고, 못 찾으면 해당 가상 페이지 넘버의 vm_entry가 없는 것이므로 NULL을 반환한다.

(6) 해시 테이블 제거 함수 구현

vm/page.c

```
83 void vm_destroy (struct hash *vm) {
84   /* hash_destroy()로 해시테이블의 버킷리스트와 vm_entry들을 제거 */
85   hash_destroy (vm, vm_destroy_func);
86 }
87
```

프로세스 종료 시 malloc()으로 메모리 동적할당 해준 vm_entry들을 해쉬 테이블 내에서 모두 제거하고 메모리 해제를 해주어야 한다. vm_destroy()는 이런 과정을 함수 한번의 호출로 수행할 수 있게 해주는 함수이다. 각 hash_elem에 해당하는 vm_entry를 해제하고, 해당 VPN에 할당된 물리 페이지가 있으면 물리페이지도 해제하는 vm_destroy_func() 함수의 포인터를 인자로 받는다.

vm/page.c

```
88 void vm_destroy_func (struct hash_elem *e, void *aux) {
89
90   /*  Get hash element (hash_entry() 사용) */
91   struct vm_entry *vme = hash_entry (e, struct vm_entry, elem);
92
93   /*  load가 되어 있는 page의 vm_entry인 경우
94    page의 할당 해제 및 page mapping 해제 (palloc_free_page()와
95    pagedir_clear_page() 사용) */
96   if (vme->is_loaded) {
97     void *kaddr = pagedir_get_page (thread_current ()->pagedir, vme->vaddr);
98     palloc_free_page (kaddr);
99     pagedir_clear_page (thread_current ()->pagedir, vme->vaddr);
100  }
```

vm_entry의 vme->is_loaded를 검사한다. 이는 해당 vm_entry에 물리페이지가 할당되어 디스크로부터 데이터가 load되었는지를 flag로서 나타낸다. 물리페이지가 할당된 경우 해당 물리페이지를 해제하고 present bit를 0으로 set하여 해당 page table entry를 무효화한다.

(7) process_exit() 함수 수정

userprog/process.c

```
451 /* Free the current process's resources. */
452 void
453 process_exit (void)
454 {
455   struct thread *cur = thread_current ();
456   struct list_elem *elem = NULL;
457   uint32_t *pd;
458   int i;
459
460   if (!(cur->tid == 1 || cur->tid == 2)) {
461
462     for (i = 2; i < FILE_MAX; i++) {
463       process_close_file (i);
464     }
465     for (elem = list_begin (&cur->mmap_list);
466          elem != list_end (&cur->mmap_list); ) {
467       struct mmap_file *mm_f = list_entry (elem, struct mmap_file, elem);
468       elem = list_next (elem);
469       do_munmap (mm_f);
470     }
471      palloc_free_page(cur->FDT);
472   }
473   file_close (cur->run_file);
474
475   /* vm_entry들을 제거하는 함수 추가 */
476   vm_destroy (&cur->vm);
477   /* Destroy the current process's page directory and switch back
478      to the kernel-only page directory. */
479   pd = cur->pagedir;
```

프로세스 종료 시 해쉬 테이블에서 vm_entry들을 안전하게 제거할 수 있도록 process_exit 함수에서 vm_destroy() 함수를 호출한다.

<가상 주소 공간 초기화>

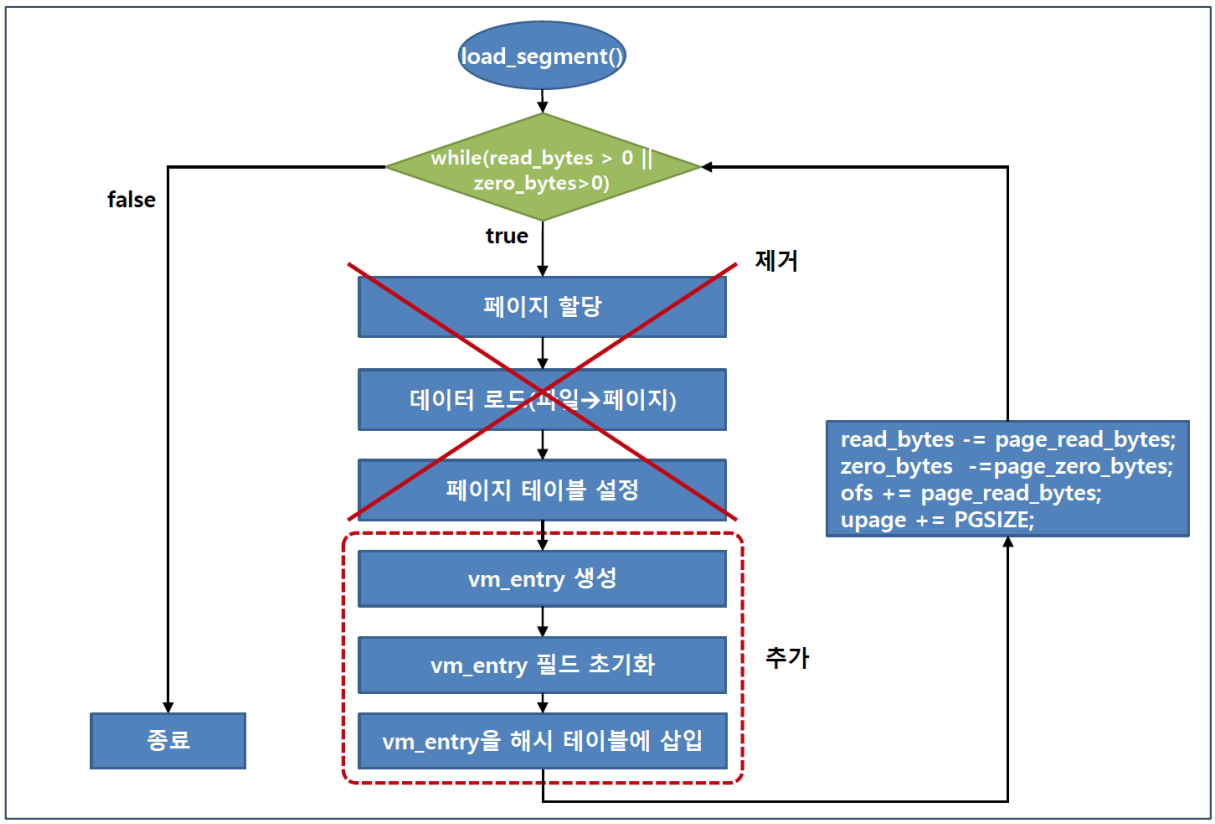
1. 함수 수정

(1) 주소 공간 초기화 관련 함수 수정 (ELF 세그먼트)

load_segment()는 ELF 포맷 파일의 세그먼트를 프로세스 가상 주소 공간에 탑재하는 함수이다.

이 함수에 프로세스 가상 메모리 관련 자료구조를 초기화하는 기능을 추가한다.

-> 프로세스 가상주소 공간에 바로 물리페이지를 할당하는 부분을 제거하고, vm_entry 구조체의 할당, 필드값 초기화, 해시 테이블 삽입 코드를 추가한다.



userprog/process.c

```
781 load_segment (struct file *file, off_t ofs, uint8_t *upage,
782               uint32_t read_bytes, uint32_t zero_bytes, bool writable)
783 {
784   ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
785   ASSERT (pg_ofs (upage) == 0);
786   ASSERT (ofs % PGSIZE == 0);
787
788   file_seek (file, ofs);
789   while (read_bytes > 0 || zero_bytes > 0)
790     {
791
792       size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
793       size_t page_zero_bytes = PGSIZE - page_read_bytes;
794
795       /*  vm_entry 생성 (malloc 사용) */
796       struct vm_entry *vme = (struct vm_entry*)malloc (sizeof (struct vm_entry));
797       if (vme == NULL)
798         return false;
799
800       /*  vm_entry 멤버들 설정, 가상페이지가 요구될 때 읽어야할 파일의 오프
801        셋과 사이즈, 마지막에 패딩할 제로 바이트 등등 */
802       memset (vme, 0x00, sizeof (struct vm_entry));
803       vme->vaddr = upage;
804       vme->type = VM_BIN;
805       vme->writable = writable;
806       vme->is_loaded = false;
807       vme->file = file;
808       vme->offset = ofs;
809       vme->read_bytes = page_read_bytes;
810       vme->zero_bytes = page_zero_bytes;
811
812
813       /*  insert_vme() 함수를 사용해서 생성한 vm_entry를 해시테이블에 추가 */
814       insert_vme (&thread_current ()->vm, vme);
815
816       /* Advance. */
817       ofs += page_read_bytes;
818       read_bytes -= page_read_bytes;
819       zero_bytes -= page_zero_bytes;
820       upage += PGSIZE;
821     }
822   return true;
823 }
```

load_segment()는 load()함수에서 호출되며, load()함수에 의해 ELF 파일이 해석되어 각 세그먼트별로 페이지 크기 단위로 잘라서 메모리에 로드될 수 있도록 한다. 기존에는 여기서 바로 물리페이지가 할당되었다. 기존 코드를 제거하고 vm_entry를 생성하여 스레드의 해시테이블에 추가한다.

```
4 static bool
5 load_segment (struct file *file, off_t ofs, uint8_t *upage,
6               uint32_t read_bytes, uint32_t zero_bytes, bool writable)
7 {
```

load() 함수에서 elf 포맷을 해석하고 세그먼트별로 load_segment()를 호출하는데, file은 실행파일의 파일 포인터를 의미하고, ofs는 파일에서 해당 세그먼트의 시작 오프셋을 의미한다. upage는 ELF 파일 포맷에 어떤 세그먼트가 어떤 가상주소 공간에 로드되는지 명시된 시작 가상주소를 의미한다. (핀토스에서는 segment의 동적 재배치가 일어나지 않는다.) read_bytes는 해당 오프셋에서 읽어들일 byte 수를 의미하고, zero_bytes는 upage으로부터 offset까지 파일을 읽어오고 마지막 주소가 page 크기 단위로 alignment 될 수 있도록 얼마나 zero로 패딩할 지 그 사이즈를 나타낸다. writeable은 해당 세그먼트에 대해 write를 할 수 있는지를 나타내며, 만약 할 수 없다면 PTE를 세팅할 때 flag가 read only로 설정되어 해당 페이지에 write 접근 시 세그멘테이션 폴트가 발생할 수 있도록 한다.

```
   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;
```

page_read_bytes는 해당 페이지에 데이터를 얼마나 읽어올지 결정한다. 예를 들어 해당 세그먼트에서 총 읽어야 하는 read_bytes가 5000byte라면, 반복문 첫번째에서 page_read_bytes는 일단 페이지 크기인 4096byte로 설정이 된다.

page_zero_bytes는 page_read_bytes가 페이지 크기라면 0으로 세팅되며, 세그먼트의 마지막 페이지를 로드할 때 얼마나 zero로 패딩할 지 그 사이즈 정보를 담는다.

```
/*  vm_entry 생성 (malloc 사용) */
struct vm_entry *vme = (struct vm_entry*)malloc (sizeof (struct vm_entry));
if (vme == NULL)
  return false;

/*  vm_entry 멤버들 설정, 가상페이지가 요구될 때 읽어야할 파일의 오프
 셋과 사이즈, 마지막에 패딩할 제로 바이트 등등 */
memset (vme, 0x00, sizeof (struct vm_entry));
vme->vaddr = upage;
vme->type = VM_BIN;
vme->writable = writable;
vme->is_loaded = false;
vme->file = file;
vme->offset = ofs;
vme->read_bytes = page_read_bytes;
vme->zero_bytes = page_zero_bytes;


/*  insert_vme() 함수를 사용해서 생성한 vm_entry를 해시테이블에 추가 */
insert_vme (&thread_current ()->vm, vme);
```

vme를 생성해서 스레드의 해쉬 테이블에 추가한다. load_segment()에 넘겨진 인자들과 page_read_bytes, page_zero_bytes로 vm_entry의 멤버들을 초기화 한다. EFL 파일을 로드하는 것이므로 type은 VM_BIN으로 지정해줘야한다.

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

다음 반복문을 돌 때 세그먼트의 다음 페이지 정보를 vm_entry로 만들 수 있도록 변수들을 계산한다.

(2) setup_stack() 함수 수정

userproc/process.c

```
861 setup_stack (void **esp)
862 {
863   struct page *page = NULL;
864   bool success = false;
865   struct vm_entry *vme = NULL;
866   void *upage = ((uint8_t *) PHYS_BASE) - PGSIZE;
867
868   page = alloc_page (PAL_USER | PAL_ZERO);
869   if (page != NULL)
870   {
871     success = install_page (upage, page->kaddr, true);
872     if (success)
873       *esp = PHYS_BASE;
874     else {
875
876       lock_acquire (&lru_lock);
877       __free_page (page);
878       lock_release (&lru_lock);
879     }
880   }
881   /* vm_entry 생성 */
882   vme = (struct vm_entry*)malloc (sizeof (struct vm_entry));
883   if (!vme)
884     return false;
885   memset (vme, 0x00, sizeof (struct vm_entry));
886
887   /* vm_entry 멤버들 설정 */
888   vme->type = VM_ANON;
889   vme->writable = true;
890   vme->vaddr = upage;
891   vme->is_loaded = true;
892   page->vme = vme;
893
894   /* insert_vme ()로 해시테이블 추가 */
895   insert_vme (&thread_current ()->vm, vme);
896   add_page_to_lru_list (page);
897
898   return success;
899 }
900
```

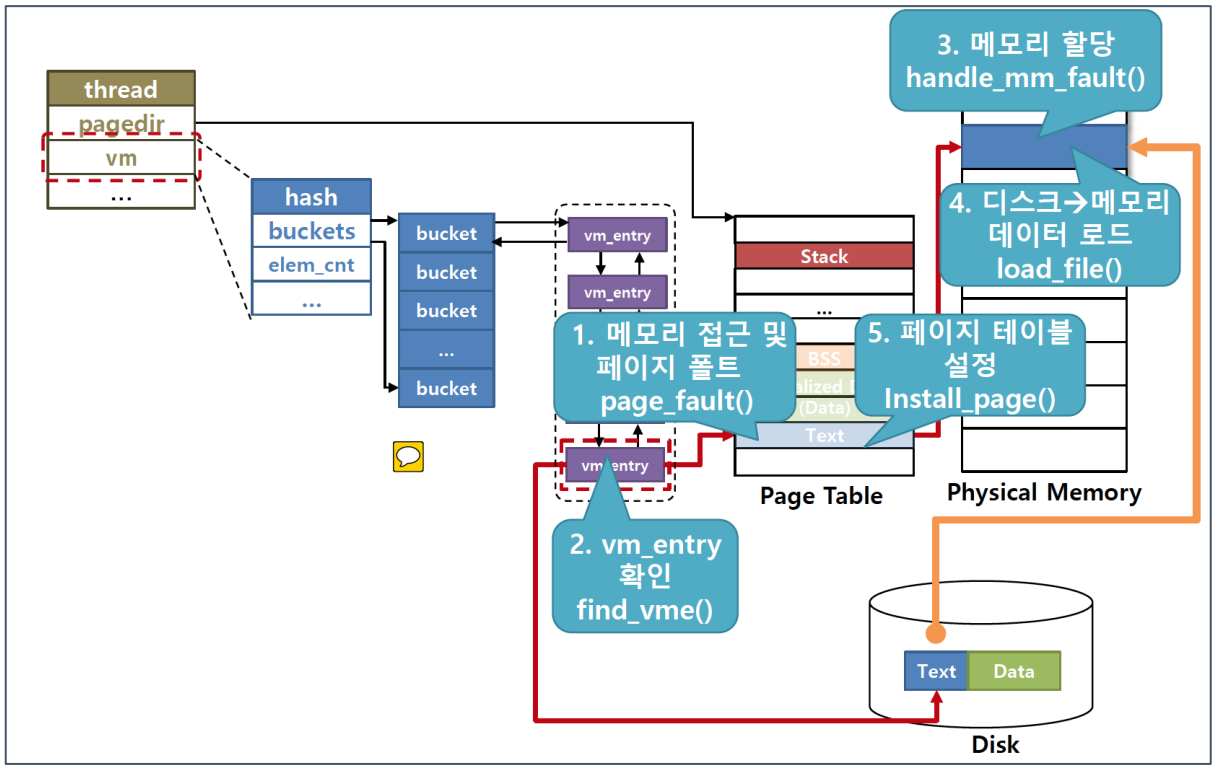
스택은 프로그램 실행 시 실행파일 따위에서 로드되는 것이 아니라 매핑된 파일 없이 물리 페이지가 할당 되어야 한다. 매핑된 파일이 없는 가상주소 공간은 anonymuous인데(VM_ANON), 최초에는 페이지 폴트가 발생해도 load할 파일이 따로 없으므로 페이지 폴트가 발생하지 않도록 바로 물리페이지를 할당해 주어야 한다.

스택의 시작주소는 유저 가상 메모리 공간의 가장 마지막 페이지의 주소가 된다. alloc_page() 함수로 물리페이지를 할당하고 install_page로 페이지 디렉토리에 해당 가상주소의 PTE에 물리페이지가 매핑될 수 있도록 한다.

그러고 나서 vm_entry를 초기화 하고 스레드의 해시테이블에 추가한다.

< demand paging >

1. 과제 설명



페이지 폴트는 아래와 같은 경우에 발생한다.

- 가상 주소가 나타내는 페이지 테이블 엔트리의 present bit이 0인데 접근이 발생할 경우
- 가상 주소가 나타내는 페이지 테이블 엔트리의 r/w 권한과 맞지 않는 접근이 발생할 경우
- privilege 권한에 맞지않는 페이지 테이블 엔트리에 접근한 경우 (유저모드에서 커널 페이지에 접근한 경우)

두번째, 세번째 경우에는 page fault handler에서 해당 스레드를 exit(-1)로 죽여야한다. 유저 코드에서 비정상적인 작동이 발생한 경우이기 때문이다.

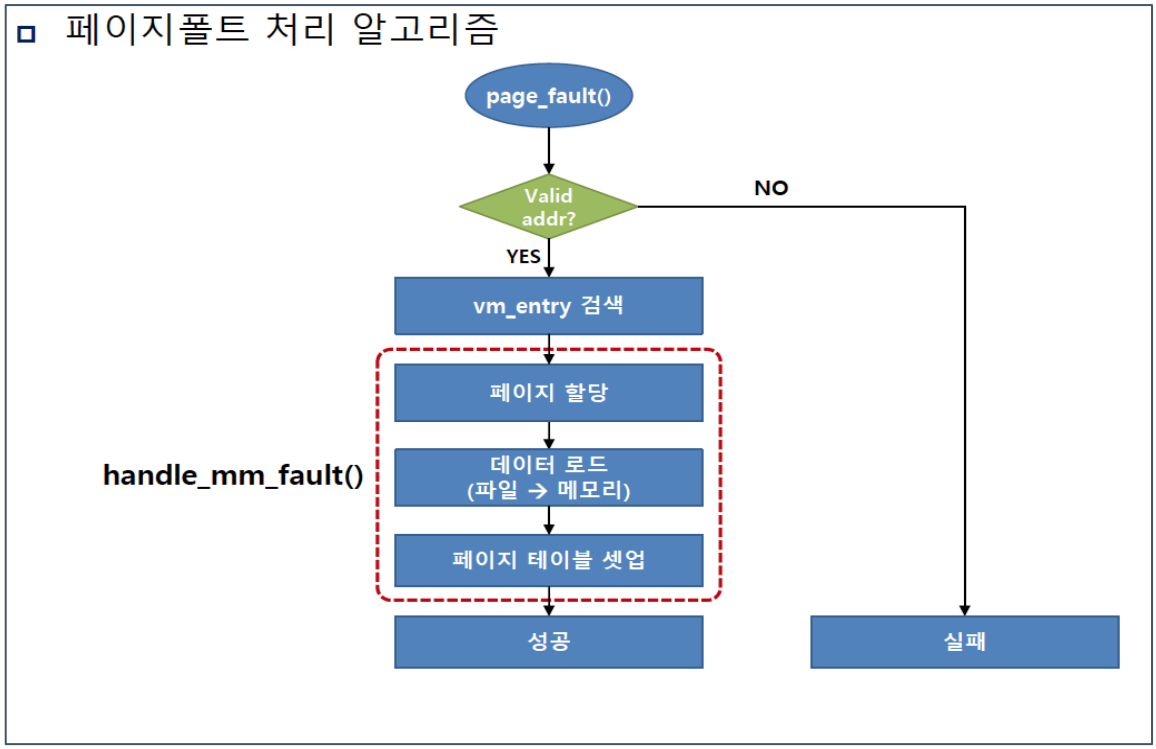
첫번째 경우에는 기존 구현에서는 스레드를 무조건 종료시켰지만, 이제는 vm_entry만 할당되고 물리 페이지가 할당되지 않은 경우가 있으므로 vm_entry 정보를 읽어 물리페이지를 할당하고, 디스크로부터 데이터를 읽어와야 하는 작업을 해주어야 한다.

첫번째 경우에 해당 가상 주소의 가상 페이지 넘버에 해당하는 vm_entry가 있는지 없는지 검사를 하여 유효한 가상주소인지 검사를 하고, 유효하지 않다면 해당 프로세스를 종료한다.

2. 함수 구현 및 수정

(1) 페이지 폴트 처리

물리 페이지의 부재로 페이지 폴트가 발생하게 된다. 그럼, vm_entry를 검색한 후 페이지를 할당하도록 한다.



userprog/exception.c::page_fault()

```
154   /* read only 페이지에 대한 접근이 아닐 경우 (not_present 참조)*/
155   if (not_present) {
156     /* 페이지 폴트가 일어난 주소에 대한 vm_entry 구조체 탐색 */
157     vme = find_vme (fault_addr);
158     /* bad address (비정상적인 가상 주소 접근 시) 프로세스 종료 */
159     if (vme == NULL)
160       exit (-1);
161
162     /* vm_entry를 인자로 넘겨주며 handle_mm_fault() 호출 */
163     /* 제대로 파일이 물리 메모리에 로드 되고 맵핑 됐는지 검사 */
164     /* file을 읽어오지 못하거나, 페이지 pool이 가득 차 물리페이지에 맵핑을 못한경우 */
165     if (handle_mm_fault (vme) == false)
166       exit (-1);
167   } else {
168     /* present인 페이지를 접근하다가 page_fault가 난 경우는 모두 죽여버려야함.
169        privilege 권한 위반인경우, r/w 권한 위반인 경우 등*/
170     exit (-1);
171   }
```

page_fault() : 페이지 폴트 발생시 intr_handler에 의해 호출되어지는 핸들러 함수이다. demand paging이 필요한 경우 아니면 모두 비정상적인 메모리 접근에 의한 페이지 폴트 발생이므로 프로세스를 종료하는 예외 처리를 해주어야 한다.

demand paging이 필요한 경우라 함은 fault_address의 가상 페이지 넘버에 해당하는 vm_entry가 해당 스레드의 vm 해쉬 테이블에 존재하는 경우를 말한다. find_vme() 함수에서 vm_entry가 존재하는지 찾아보고 vm_entry가 존재하면 handle_mm_fault에 의해 물리 페이지가 할당되고 매핑된 파일이 로드 될 수 있도록 처리한다.

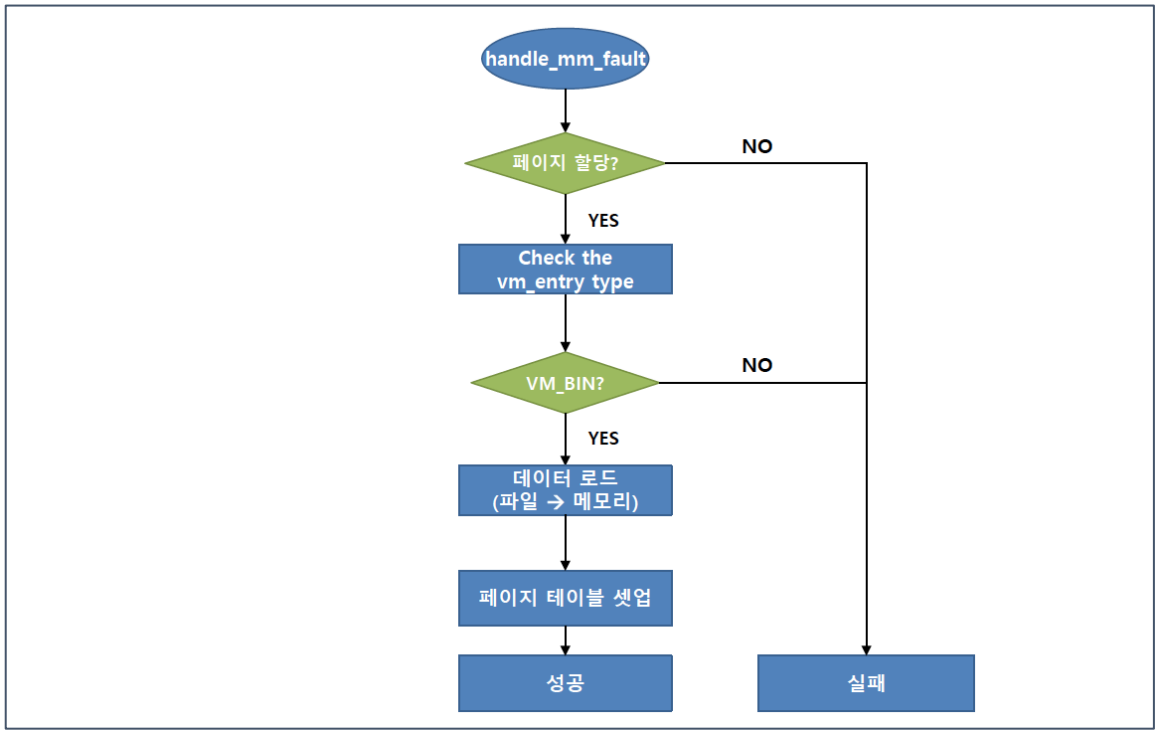
vm_entry를 찾아보고 없는 경우에는 프로세스를 종료시켜야한다. 이 경우는 비정상적인 가상

주소를 참조할 때(vm_entry가 할당되지 않은 공간) 발생한다.

present 플래그가 set 되어있는 물리페이지를 참조하다가 페이지 폴트가 발생한 경우에도 무조건 프로세스를 종료시켜야 한다. 이러한 경우는 privilege 권한 문제 혹은 r/w 권한 문제의 경우에 발생한다.

(2) 페이지 폴트 핸들러 구현

handle_mm_fault는 demand paging이 필요한 경우의 page fault의 handling을 수행하는 함수이다.



userprog/process.c

```
79 bool handle_mm_fault (struct vm_entry *vme) {
80   /* palloc_get_page()를 이용해서 물리메모리 할당 */
81   /* page는 user pool에서 가져와야 한다. */
82   struct page *page = NULL;
83   do {
84     page = alloc_page (PAL_USER|PAL_ZERO);
85   } while (page->kaddr == NULL);
86
87   ASSERT (vme != NULL);
88   /* switch문으로 vm_entry의 타입별 처리 (VM_BIN외의 나머지 타입은 mmf
89      와 swapping에서 다룸*/
90   page->vme = vme;
```

alloc_page()함수를 이용하여 물리페이지를 할당한다. alloc_page() 함수에 대해서는 swap 과제에서 다룬다.

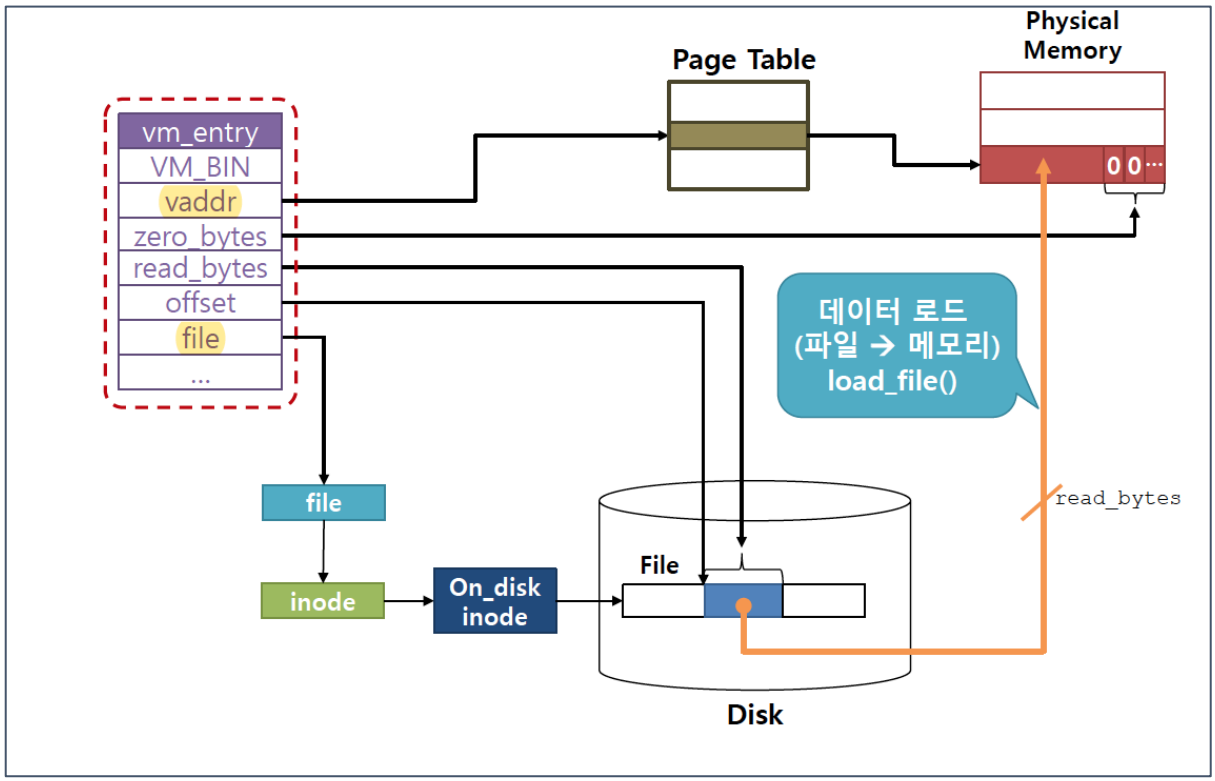
```
92   switch (vme->type) {
93     /* VM_BIN일 경우 load_file()함수를 이용해서 물리메모리에 로드 */
94     /* install_page를 이용해서 물리페이지와 가상페이지 맵핑 */
95     case VM_BIN :
96     case VM_FILE :
97       /* load_file(), install_page() 수행 중 false 반환 되는 경우 예외처리 */
98       if (!(load_file (page->kaddr, vme) &&
99             install_page (vme->vaddr, page->kaddr, vme->writable)))  {
100        __free_page (page);
101        return false;
102      }
103      vme->is_loaded = true;
104      add_page_to_lru_list (page);
105      break;
106    case VM_ANON :
107      swap_in (vme->swap_slot, page->kaddr);
108      if (!install_page (vme->vaddr, page->kaddr, vme->writable)) {
109        __free_page (page);
110        return false;
111      }
112      vme->is_loaded = true;
113      add_page_to_lru_list (page);
114      break;
115  }
116  /* 로드 성공 여부 반환 */
117  return true;
118 }
```

vm_entry의 type이 실행파일이거나 일반 파일인 경우면 load_file 함수로 디스크에 저장된 데이터를 물리 페이지로 읽어온다. 그러고 나서 install_page() 함수로 page directory에 virtual page number에 해당하는 PTE에 할당된 물리 페이지의 PFN가 매핑될 수 있도록 한다.

vm_entry의 type이 VM_ANON인 경우는 swap 과제에서 다룬다.

(3) 물리메모리에 파일 쓰기

물리 메모리 할당 완료 후 실제 디스크의 파일을 물리페이지로 load한다.



```
12 bool load_file (void *kaddr, struct vm_entry *vme) {
13   /* Using file_read_at()*/
14   /* file_read_at으로 물리페이지에 read_bytes만큼 데이터를 씀*/
15   /* file_read_at 여부 반환 */
16
17   if (file_read_at (vme->file, kaddr, vme->read_bytes, vme->offset) != vme->read_bytes)
18     return false;
19
20   /* zero_bytes만큼 남는 부분을'0'으로 패딩 */
21   memset (kaddr + vme->read_bytes, 0, vme->zero_bytes);
22
23   /*정상적으로 file을 메모리에 loading 하면 true 리턴*/
24   return true;
25 }
26
```

vm_entry에 어떤 파일의 어디 오프셋부터 얼마나 읽고 0으로 채워야 할 페이지의 남은 공간의 크기가 얼마나 되는지 등을 참고하여 디스크로부터 데이터를 읽어온다.

Virtual Memory

- Memory Mapped File

(1) 과제 설명

운영체제의 시스템 콜에는 mmap()이라는 함수가 있다. 이 함수는 파일을 메모리에 매핑시켜서 해당 파일을 디스크가 아닌 메모리 접근하듯이 사용할 수 있게 해준다.

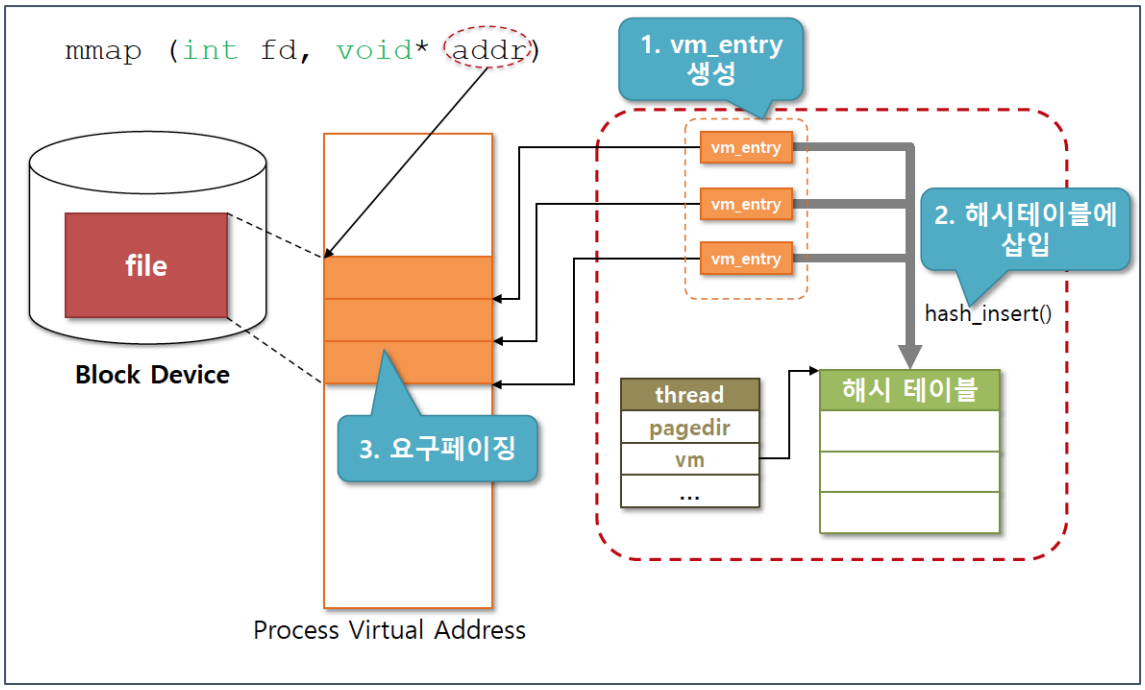
디스크가 아닌 메모리에 매핑 시켜서 파일에 접근하면 여러가지 이점이 있다. 첫번째로는 기존 write(), read() 시스템 콜을 이용해 파일에 접근을 하게 되는 경우 매번 시스템 콜을 호출하고 disk I/O를 기다리는데 발생하는 오버헤드가 크다. 파일에 10000번 쓰기를 수행해야 하는 경우, write() 시스템 콜을 하기 위해 privilege 모드 스위칭 10000번, disk io요청 10000번이 발생하게 된다. 하지만 mmap을 이용하여 메모리에 파일을 한번만 매핑 시키고 나면 메모리 접근하듯이 파일에 접근하면 되는 것이므로 10000번의 쓰기 작업을 해야 해도 privilege 모드 스위칭이나 disk io를 기다려야 하는 오버헤드가 발생하지 않는다.

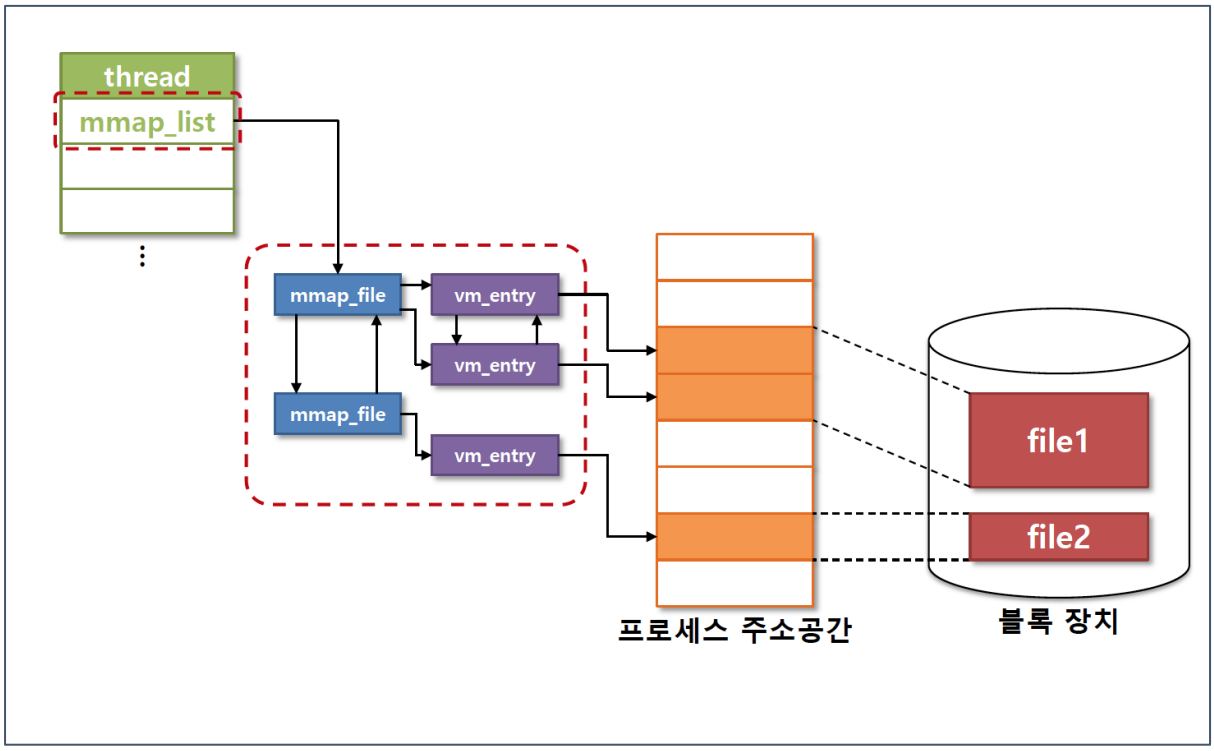
또한 핀토스에서는 구현하진 않지만, anonymous 페이징을 할당 받는 데도 사용할 수 있다. file 인자를 NULL로 넘겨주면 파일이 매핑되지 않은 물리 페이지를 할당받고 해당 mmap_object를 다른 프로세스와 공유하여 프로세스간 메모리 공유를 하는데도 활용할 수 있다.

(2) 과제 목표

현재의 핀토스는 mmap()과 munmap()함수가 구현되어 있지 않아 본 과제에서 이 두 함수를 구현한다.

(3) 함수 구현





threads/thread.h :: struct thread

```
/* 스레드가 가진 가상 주소 공간을 관리하는 해시테이블 */
struct hash vm;

/* mmap으로 물리 메모리에 매핑 시킨 파일 목록을 관리하는 리스트 */
struct list mmap_list;
int next_mapid;
```

thread 구조체에 해당 스레드가 mmap() 시스템콜로 만든 mmap_file 구조체를 관리하는 리스트 mmap_list와 mmap_file의 mapid를 할당하는데 사용되어지는 next_mapid 멤버 변수를 추가한다.

vm/page.h

```
14 struct mmap_file {
15   int mapid;
16   struct file* file;
17   struct list_elem elem;
18   struct list vme_list;
19 };
20
```

mmap() 시스템콜에서 vme_entry와 file의 매핑 정보를 저장하는 mmap_file 구조체다. mapid는 mmap_file의 고유 id를 의미하고, file은 매핑된 파일의 파일 객체 포인터를 저장하며, elem은 thread에서 mmap_file 리스트를 관리하기 위한 list_elem 구조체다. vme_list는 한 파일의 크기가 페이지 크기보다 크면 매핑된 vme_entry가 여러 개이므로 이를 관리하기 위해 list로 관리한다.

vm/page.h

```
21 struct vm_entry {
22   uint8_t type;              /*  VM_BIN, VM_FILE, VM_ANON의 타입 */
23   void *vaddr;               /*  vm_entry의 가상페이지 번호 */
24   bool writable;             /*  True일 경우 해당 주소에 write 가능
25                                  False일 경우 해당 주소에 write 불가능 */
26   bool is_loaded;            /*  물리메모리의 탑재 여부를 알려주는 플래그 */
27   struct file *file;         /*  가상주소와 맵핑된 파일 */
28
29   /*  Memory Mapped File 에서 다룰 예정 */
30   struct list_elem mmap_elem;   /*  mmap 리스트 element */
31
32   size_t offset;                /*  읽어야 할 파일 오프셋 */
33   size_t read_bytes;            /*  가상페이지에 쓰여져 있는 데이터 크기 */
34   size_t zero_bytes;            /*  0으로 채울 남은 페이지의 바이트 */
35
36   /*  Swapping 과제에서 다룰 예정 */
37   size_t swap_slot;             /*  스왑 슬롯 */
38
39   /*  'vm_entry들을 위한 자료구조' 부분에서 다룰 예정 */
40   struct hash_elem elem;        /*  해시 테이블 Element */
41 };
42
```

<vm_entry 구조체에 추가된 mmap_elem list_elem>

mmap()

userprog/syscall.c

```
64 mapid_t mmap (int fd, void *addr) {
65   struct mmap_file *mmap_file = NULL;
66   struct file *file = NULL;
67   off_t offset = 0;
68   uint8_t *upage = addr;
69   uint32_t read_bytes = 0;
70
71   /* file object pointer를 가져옴 */
72   file = process_get_file (fd);
73   if (file == NULL) {
74     return -1;
75   }
76
77   /* addr 은 page 크기 단위여야함. 그리고 유저 메모리 영역이어야함*/
78   if (!(addr != NULL &&
79         pg_ofs (addr) == 0 &&
80         is_user_vaddr (addr) &&
81         check_address (addr) == NULL)) {
82     return -1;
83   }
84
85   /* mmap_file 를 생성하기 위해 메모리 할당 */
86   mmap_file = (struct mmap_file *)malloc (sizeof (struct mmap_file));
87   if (mmap_file == NULL) {
88     return -1;
89   }
90
```

mmap을 호출하기 위해선 먼저 open() system call로 파일을 열어야 한다. 핀토스에선 anonymous mmap()을 구현하지 않으므로 입력으로 받은 file descriptor에 해당하는 파일이 존재하지 않으면 함수를 종료한다.

mmap() 함수 내에서 인자로 받은 유저 가상 주소인 addr에 대한 유효성 검사를 하고 예외처리를 수행한다. 유효성 검사는 인자로 받은 addr이 NULL이 아닌지, VPN를 나타내기 위해 페이지 크기 단위로 round down 된 값인지, 유저 공간의 주소인지, 해당 가상 주소의 vm_entry가 존재하는지를 검사하고, 조건에 맞지 않으면 mmap() 함수를 종료한다.

유효성 체크를 하고 나서는 mmap_file 구조체에 동적 메모리를 할당한다.

```
91   /* mmap_file 멤버 초기화 */
92   memset (mmap_file, 0x00, sizeof (struct mmap_file));
93   /* fd + 최대 파일 디스크립터 개수로 mapid 결정*/
94   mmap_file->mapid = thread_current ()->next_mapid++;
95   /* 파일이 나중에 close되어도 mmap() 유효성 유지 */
96   mmap_file->file = file_reopen (file);
97   list_init (&mmap_file->vme_list);
98   list_push_back (&thread_current ()->mmap_list, &mmap_file->elem);
99
100  read_bytes = file_length (mmap_file->file);
101
102  while (read_bytes > 0) {
103      /* Calculate how to fill this page.
104         We will read PAGE_READ_BYTES bytes from FILE
105         and zero the final PAGE_ZERO_BYTES bytes. */
106      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
107      size_t page_zero_bytes = PGSIZE - page_read_bytes;
108
109      /*  vm_entry 생성 (malloc 사용) */
110      struct vm_entry *vme = (struct vm_entry*)malloc (sizeof (struct vm_entry));
111      if (vme == NULL)
112        return false;
113
```

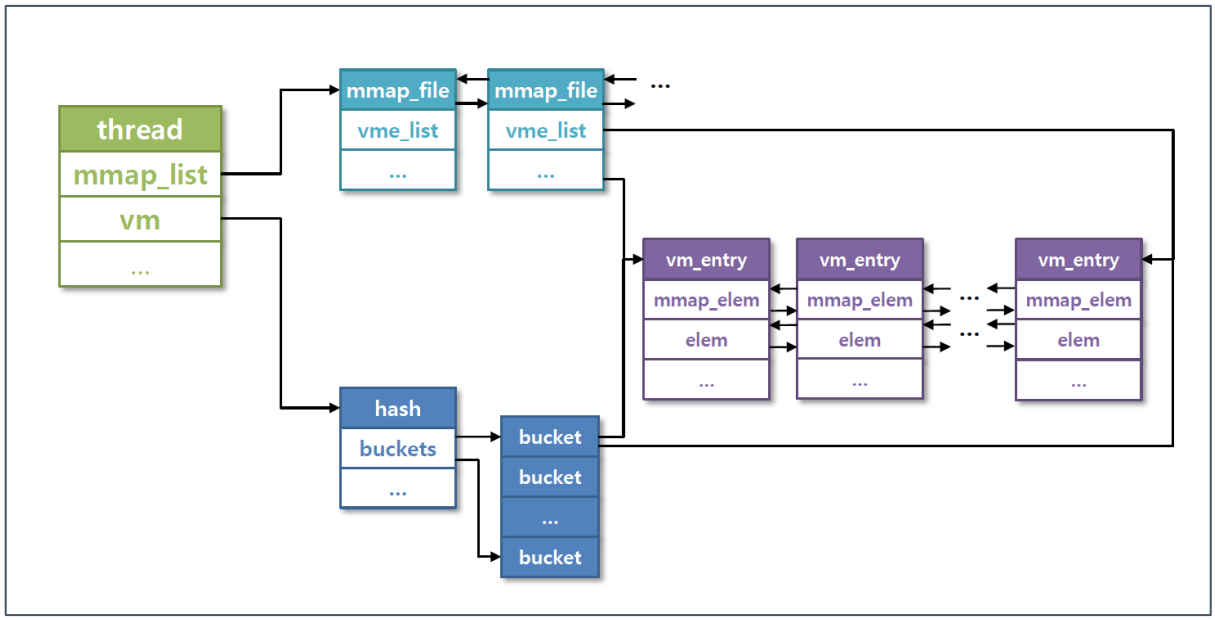
mmap_file 구조체에 mapid를 할당하고 file_reopen() 함수를 이용해 기존 file 객체를 따로 다시 열어서 file이 닫힌 후에도 mmap에 의한 파일 매핑이 유효하도록 한다. 새로 생성한 file object에 대한 포인터를 mmap_file->file에 저장한다.

mmap_file에 대한 초기화가 끝나면 thread 구조체에서 관리하고 있는 mmap_list에 추가한다.

```
113
114      /*  vm_entry 멤버들 설정, 가상페이지가 요구될 때 읽어야할 파일의 오프
115       셋과 사이즈, 마지막에 패딩할 제로 바이트 등등 */
116      memset (vme, 0x00, sizeof (struct vm_entry));
117      vme->vaddr = upage;
118      vme->type = VM_FILE;
119      vme->writable = true;
120      vme->is_loaded = false;
121      vme->file = mmap_file->file;
122      vme->offset = offset;
123      vme->read_bytes = page_read_bytes;
124      vme->zero_bytes = page_zero_bytes;
125
126      /*  insert_vme() 함수를 사용해서 생성한 vm_entry를 해시테이블에 추가 */
127      list_push_back (&mmap_file->vme_list, &vme->mmap_elem);
128      insert_vme (&thread_current ()->vm, vme);
129
130      /* Advance. */
131      offset += page_read_bytes;
132      read_bytes -= page_read_bytes;
133      upage += PGSIZE;
134  }
135  return mmap_file->mapid;
136 }
137
```

mmap_list를 추가하고 파일을 페이지 크기 단위로 잘라서 각각 vm_entry에 할당한다. 이는 load_segment()에서 세그먼트에 해당하는 부분을 페이지 단위로 잘라 각각 vm_entry를 할당하는 루틴과 같은 작업을 한다.

munmap()



userprog/syscall.c

```
43 void munmap (int mapping) {
44   struct list_elem *elem = NULL;
45   struct thread *cur = thread_current ();
46   struct mmap_file *mmap_file = NULL;
47
48   /* mapping에 해당하는 mmap_file을 thread 구조체의 mmap_list를 순회하여 찾는다 */
49   for (elem = list_begin (&cur->mmap_list);
50        elem != list_end (&cur->mmap_list); elem = list_next (elem)) {
51     struct mmap_file *mm_f = list_entry (elem, struct mmap_file, elem);
52     if (mm_f->mapid == mapping) {
53       mmap_file = mm_f;
54       break;
55     }
56   }
57   /* mapping 에 해당하는 mmap_file을 찾지 못한 경우 종료 */
58   if (mmap_file == NULL)
59     return;
60   else
61     do_munmap (mmap_file);
62 }
```

mmap()으로 할당받은 mmap_file을 해제하는 시스템 콜이다. thread 구조체의 mmap_list를 순회하여 인자로 받은 mapid인 mapping과 mmap_file.mapid를 비교하여 맞는 mmap_file을 찾는다. 리스트 전체를 순회해도 해당하는 mmap_file이 없으면 함수를 종료한다. mmap_file 객체를 찾으면 do_munmap() 함수를 호출하여 mmap_file에 대한 해제를 한다.

userprog/process.c

```
42 void do_munmap (struct mmap_file *mmap_file) {
43   struct thread *cur = thread_current ();
44   struct list_elem *vm_elem = NULL;
45
46   /* mmap_list를 순회하여 vme를 해제함 */
47   vm_elem = list_begin (&mmap_file->vme_list);
48   while (vm_elem != list_end (&mmap_file->vme_list)) {
49     struct list_elem *next_elem = list_next (vm_elem);
50
51     /* vme가 가리키는 가상주소에 대한 물리 페이지가 존재하고, dirty하면 write-back을 함 */
52     struct vm_entry *vme = list_entry (vm_elem, struct vm_entry, mmap_elem);
53     if (vme->is_loaded) {
54       if (pagedir_is_dirty (cur->pagedir, vme->vaddr)) {
55         lock_acquire (&rw_lock);
56         file_write_at (vme->file, vme->vaddr, vme->read_bytes, vme->offset);
57         lock_release (&rw_lock);
58       }
59       /* load 된 경우에는 해당 페이지 해제 */
60       free_page (pagedir_get_page (cur->pagedir, vme->vaddr));
61     }
62
63     /* vm_elem 을 mmap_list에서 제거한다 */
64     list_remove (vm_elem);
65     /* hash table에서 vme를 제거한다 */
66     delete_vme (&cur->vm, vme);
67     /* vme에 할당된 동적 메모리를 해제한다 */
68     free (vme);
69
70     vm_elem = next_elem;
71   }
72
73   /* mmap_list에서 mmap_file을 제거한다 */
74   list_remove (&mmap_file->elem);
75   /* mmap_file에 할당된 동적 메모리를 해제한다 */
76   free (mmap_file);
77 }
```

do_munmap() 함수는 인자로 받은 mmap_file 객체를 해제하는 함수다. mmap_file의 vm_list를 순회하여 mapping된 vm_entry에 할당된 페이지가 dirty할 시 disk에 변경내역이 반영될 수 있도록 write-back 한다. write-back이 완료된 이후에 vm_list에서 해당 vm_entry를 제거하고, 스레드의 해쉬테이블에서 해당 vm_entry를 제거한다. 그리고 vm_entry 생성 시 할당해줬던 메모리를 해제한다.

vm_entry에 대한 해제가 모두 완료되면, thread의 mmap_list에서 mmap_file을 제거하고 mmap_file에 할당된 메모리를 해제한다.

userprog/syscall.c :: syscall_handler

```
243       case SYS_MMAP :
244          get_argument (esp, arg, 2);
245          f->eax = mmap ((int)arg[0], (void*)arg[1]);
246          break;
247
248       case SYS_MUNMAP :
249          get_argument (esp, arg, 1);
250          munmap ((int)arg[0]);
251          break;
```

mmap()과 munmap() 시스템 콜을 추가한다.

```
451 /* Free the current process's resources. */
452 void
453 process_exit (void)
454 {
455   struct thread *cur = thread_current ();
456   struct list_elem *elem = NULL;
457   uint32_t *pd;
458   int i;
459
460   if (!(cur->tid == 1 || cur->tid == 2)) {
461
462     for (i = 2; i < FILE_MAX; i++) {
463       process_close_file (i);
464     }
465     for (elem = list_begin (&cur->mmap_list);
466          elem != list_end (&cur->mmap_list); ) {
467       struct mmap_file *mm_f = list_entry (elem, struct mmap_file, elem);
468       elem = list_next (elem);
469       do_munmap (mm_f);
470     }
471      palloc_free_page(cur->FDT);
472   }
473   file_close (cur->run_file);
474
475   /* vm_entry들을 제거하는 함수 추가 */
476   vm_destroy (&cur->vm);
477   /* Destroy the current process's page directory and switch back
478      to the kernel-only page directory. */
479   pd = cur->pagedir;
```

thread 종료 시 mmap_file 들이 모두 해제될 수 있도록 process_exit() 함수에서 thread의 mmap_list를 순회하며 do_munmap()함수를 호출하여 mmap_file을 해제한다.

userprog/syscall.c

```
427 // 인자로 받아온 포인터 주소가 kernel영역 메모리 주소면 해당 프로세스를 종료시킨다.
428 struct vm_entry *check_address (void *addr)
429 {
430   struct vm_entry *vme;
431   if (!( (void*)USER_START <= addr && addr < (void*)KERNEL_START ))
432   {
433     //printf ("invailed parameter.\n");
434     exit (-1);
435   }
436
437   /* addr이 vm_entry에 존재하면 vm_entry를 반환하도록 코드 작성 */
438   /* find_vme() 사용*/
439   vme = find_vme (addr);
440   if (!vme)
441     return NULL;
442
443   return vme;
444
445 }
446
```

기존에 구현한 check_address() 함수에서 인자로 받은 가상 주소에 해당하는 vm_entry가 존재하는지 검사할 수 있도록 코드를 수정한다. fine_vme() 함수에 의해 찾아 지지 않았다면 NULL을 반환한다.

vm/page.c

```
106 void check_valid_buffer (void *buffer, unsigned size,
107                          void *esp, bool to_write) {
108   void *from = NULL, *to = NULL;
109   void *vaddr = NULL;
110   struct vm_entry *vme = NULL;
111   unsigned number_of_page, i;
112
113   /*  인자로 받은 buffer부터 buffer + size까지의 크기가 한 페이지의
114       크기를 넘을 수도 있음 */
115   /* 검사해야할 페이지 개수를 구함 */
116   from = (unsigned)pg_round_down (buffer);
117   to = (unsigned)pg_round_down ((unsigned)buffer + size - 1);
118   number_of_page = ((to - from) >> PGBITS) + 1;
119
120   for (i = 0; i < number_of_page; i++) {
121     /*  check_address를 이용해서 주소의 유저영역 여부를 검사함과 동시
122         에 vm_entry 구조체를 얻음 */
123     if (i != number_of_page - 1) {
124       vaddr = (unsigned)buffer + (PGSIZE * i);
125     } else {
126       vaddr = (unsigned)buffer + size - 1;
127     }
128     vme = check_address (vaddr);
129
130     /*  해당 주소에 대한 vm_entry 존재여부와 vm_entry의 writable 멤
131         버가 true인지 검사 */
132     // only if to_write is true, check writable
133     if (vme == NULL ||
134         (to_write == true && vme->writable == false)) {
135       exit (-1);
136     }
137   }
138   /*  위 내용을 buffer 부터 buffer + size까지의 주소에 포함되는
139       vm_entry들에 대해 적용 */
140 }
141
```

가상 주소 공간을 시작 버퍼 주소로부터 size만큼의 범위를 검사할 수 있도록 check_vaild_buffer() 함수를 만들었다. to_write frag가 설정되어 있으면 해당 vm_entry가 writeable 한지 검사할 수 있도록 하였다.

```
113   /*  인자로 받은 buffer부터 buffer + size까지의 크기가 한 페이지의
114       크기를 넘을 수도 있음 */
115   /* 검사해야할 페이지 개수를 구함 */
116   from = (unsigned)pg_round_down (buffer);
117   to = (unsigned)pg_round_down ((unsigned)buffer + size - 1);
118   number_of_page = ((to - from) >> PGBITS) + 1;
119
```

number_of_page 변수에 총 검사해야 하는 VPN 개수가 몇 개인지를 계산하여 저장한다.

```
120   for (i = 0; i < number_of_page; i++) {
121     /*  check_address를 이용해서 주소의 유저영역 여부를 검사함과 동시
122         에 vm_entry 구조체를 얻음 */
123     if (i != number_of_page - 1) {
124       vaddr = (unsigned)buffer + (PGSIZE * i);
125     } else {
126       vaddr = (unsigned)buffer + size - 1;
127     }
128     vme = check_address (vaddr);
129
130     /*  해당 주소에 대한 vm_entry 존재여부와 vm_entry의 writable 멤
131         버가 true인지 검사 */
132     // only if to_write is true, check writable
133     if (vme == NULL ||
134         (to_write == true && vme->writable == false)) {
135       exit (-1);
136     }
137   }
138   /*  위 내용을 buffer 부터 buffer + size까지의 주소에 포함되는
139       vm_entry들에 대해 적용 */
140 }
```

입력 받은 buffer 인자와 size 인자에 해당하는 가상 주소 범위에 걸쳐 있는 vm_entry를 모두 검사한다. vm_entry가 없거나, to_write frag가 설정되어 writeable 한지도 검사할 경우, vm_entry의 writeable이 false인 경우에 프로세스를 종료하게 된다.

vm/page.c

```
142 void check_valid_string (const void *str, void *esp) {
143   /* check virtual address without checking writeability */
144   check_valid_buffer (str, strlen (str) + 1, esp, false);
145 }
146
```

입력받은 string이 유효한지 검사한다. writeable한지 검사하지 않기 위해 to_write 인자에 false를 넘겼다.

Virtual Memory

- Swapping

1. 과제 설명

물리 메모리의 공간은 제한 되어있다. 지금까지 구현한 핀토스에서는 한번에 실행할 수 있는 thread가 제한될 수밖에 없다. 요구 페이징에 의해 물리 페이징을 모두 할당하고 나면 더 이상 메모리를 할당할 수가 없어서 프로그램 실행에 문제가 생긴다. swapping 기법을 통해 물리 페이지 할당 시도 중 페이지 풀이 가득 차 할당할 수 있는 페이지가 남아있지 않다면, 기존 할당한 페이지 중 victim page를 선정해 disk로 swap_out 시키고 해당 물리 페이지를 해제한다음 해당 물리 페이지를 다른 필요한 thread에게 할당해주는 swapping을 구현할 것이다.

swap-out은 메모리의 내용을 디스크의 swap 영역으로 방출하는 것을 말하고, swap-in은 swap-out된 페이지를 다시 메모리에 탑재하는 것을 말한다.

핀토스는 swap 파티션을 swap space로 제공한다. make check test에서 기본적으로 4MB의 swap partition을 사용한다.

우리가 스와핑을 하기위해서는 다음 4단계를 따라

1. 물리메모리의 크기를 알아야한다.

물리메모리의 크기를 정확히 알 필요는 없다. palloc_get_page()함수가 NULL을 리턴하면 할당해줄 페이지가 없는 것이기 때문에, NULL을 리턴할 때 스와핑을 해주도록 하면 된다.

2. 디스크의 swap partition에 접근 할 수 있어야 한다. partion하나는 block device라고 한다. 핀토스의 block_get_role() 함수를 사용하여 swap disk를 block 단위로 접근할 수 있는 block 구조체의 포인터를 갖고 올 수 있다.

3. swap partition을 관리할 수 있어야한다. swap-out을 하기위해서는 swap partition에서 어떤 offset에 저장을 해야 하는지에 대한 정보를 알아야 한다. 즉, 비트맵을 사용하여, swap partition의 offset 들을 페이지 하나마다 관리한다.

4. victim page 선정 알고리즘이 필요하다. 우리는 LRU방식을 근사한 clock알고리즘을 사용하였다. clock알고리즘은 포인터를 시계 방향으로 이동시키면서 교체될 페이지를 선정한다.
현재 포인터가 가리키고 있는 페이지의 참조 비트를 검사하며, '0'이면 해당 페이지를 victim으로 선정하고, '1'이면 참조 비트를 0으로 재설정한다.

victim으로 선정된 페이지가 프로세스의 데이터 영역 혹은 스택에 포함될 때 이를 스왑 영역에 저장한다. swap-out 된 페이지는 요구 페이징에 의해 다시 메모리에 로드한다.

페이지 테이블의 dirty bit은 해당 메모리 영역에 쓰기시 하드웨어에 의하여 "1"로 설정된다. dirty bit이 "1"인 페이지가 victim으로 선정되었을 때, 변경 내용을 항상 디스크에 반영해줘야 한다.

(2) 함수 구현 및 수정

vm/swap.c

```
 6 #include "userprog/syscall.h"
 7
 8 struct lock swap_lock;
 9 struct bitmap *swap_bitmap;
10 struct block *swap_block;
11
12 #define SECTORS_PER_PAGE (PGSIZE / BLOCK_SECTOR_SIZE)
```

swap disk에 접근할 때 race condition을 막고자 lock을 사용해야한다. swap_lock을 전역변수로 선언하였다.

bitmap으로 어떤 swap disk의 block이 swap-out된 페이지가 저장되어 있는지 bit array로 관리하는 구조체의 포인터를 선언한다. swap_out(), swap_in()에서 모두 참조해야 하므로 전역변수로 선언하였다.

swap_block은 실질적으로 swap partition에 block 단위로 접근이 가능하게 해주는 구조체이다. 자세한 원리는 file system에서 다룬다.

vm/swap.c

```
14 void swap_init (void) {
15   lock_init (&swap_lock);
16   swap_bitmap = bitmap_create (SWAP_SIZE / PGSIZE);
17   swap_block = block_get_role (BLOCK_SWAP);
18 }
```

해당 전역 변수들을 핀토스 부팅 시 초기화 할 수 있도록 swap_init() 함수를 만들어 주었다. 이 함수는 init.c의 mail() 함수에서 filesystem 초기화 이후에 호출되어진다.

vm/swap.c

```
20 size_t swap_out (void *kaddr) {
21   size_t swap_index = 0, sector_index = 0;
22   int i = 0;
23   lock_acquire (&swap_lock);
24   swap_index = bitmap_scan_and_flip (swap_bitmap, 0, 1, false);
25   sector_index = swap_index << 3;
26   lock_acquire (&rw_lock);
27   for (i = 0; i < SECTORS_PER_PAGE; i++) {
28     block_write (swap_block, sector_index + i, kaddr + BLOCK_SECTOR_SIZE * i);
29   }
30   lock_release (&rw_lock);
31   lock_release (&swap_lock);
32   return swap_index;
33 }
34
```

swap_out 함수는 물리 페이지에 저장되어있는 데이터를 swap data에 저장하게 해주는 함수이다. swap_bitmap에서 bitmap_scan_and_flip() 함수로 linear search 수행 후 first-fit으로 비어있는 block 인덱스를 찾아 반환한다. 여기서의 인덱스는 8개의 block을 하나로 간주하여 해석되는 index이다. 왜냐하면 block의 크기는 512byte이고 page의 크기는 4KB인데 하나의 페이지를 swap partition에 저장하기 위해서는 8개의 block에 나눠서 저장해야 하기 때문이다.

해당 인덱스를 좌로 3 shift 하여 8을 곱한 후, 해당 block index로부터 8개 block에 나눠서

page에 저장된 내용을 해당 block에 저장한다. 그러고 나서 8개 block을 하나의 인덱스로 나타내는 swap_index 변수를 반환한다.

vm/swap.c

```
35 void swap_in (size_t used_index, void *kaddr) {
36   int i = 0;
37   size_t sector_index = used_index << 3;
38   lock_acquire (&swap_lock);
39   lock_acquire (&rw_lock);
40   for (i = 0; i < SECTORS_PER_PAGE; i++)  {
41     block_read (swap_block, sector_index + i, kaddr + BLOCK_SECTOR_SIZE * i);
42   }
43   bitmap_set (swap_bitmap, used_index, false);
44   lock_release (&rw_lock);
45   lock_release (&swap_lock);
46 }
```

swap_in()는 swap out된 page를 swap partition에서 가져와 물리 페이지에 읽어오는 함수다. used_index에 해당하는 block에서 8 block을 읽어와 물리 페이지에 로드하고, bitmap의 bit array에서 used_index의 bit를 false로 set한다.

vm/page.h

```
43 struct page  {
44   void *kaddr;
45   struct vm_entry *vme;
46   struct thread *thread;
47   struct list_elem lru;
48 };
49
```

물리 페이지를 할당을 하고 해당 물리페이지가 어떤 스레드의 어떤 가상주소에 할당 되어있는지 정보를 유지할 필요가 있어서 page 구조체를 정의하였다.

또한 swapping 구현을 위해 victim page를 선정하기 위해 전역으로 선언된 lru_list에 page 구조체를 list로 유지하게 된다. page 구조체의 lru 멤버는 lru_list의 element가 된다.

vm/frame.h

```
1 #ifndef VM_FRAME_H
2 #define VM_FRAME_H
3
4
5 struct list lru_list;
6 struct lock lru_lock;
7
```

물리 페이지가 할당된 page 구조체를 관리하는 lru_list와 lru_list를 다룰 때 레이스 컨디션을 방지하고자 lock을 이용하기 위해 lru_lock을 사용한다.

vm/frame.c

```
 9 struct list_elem *lru_clock;
10
11 void lru_list_init (void) {
12   list_init (&lru_list);
13   lock_init (&lru_lock);
14   lru_clock = NULL;
15 }
```

lru_list와 lru_lock을 초기화 하고 victim 페이지 선정 알고리즘인 clock에 의해 lru_list에서 가리켜지는 element를 저장하는 lru_clock을 NULL로 초기화한다.

threads/init.c::main()

```
123 #ifdef FILESYS
124   /* Initialize file system. */
125   ide_init ();
126   locate_block_devices ();
127   filesys_init (format_filesys);
128   swap_init ();
129   lru_list_init ();
130 #endif
```

swap_init() 함수와 lru_list_init() 함수를 init.c main() 함수의 파일시스템 초기화 이후에 수행될 수 있도록 추가한다. 핀토스 부팅 시 실행되어 초기화를 수행한다.

vm/frame.c

```
17 void add_page_to_lru_list (struct page *page) {
18   lock_acquire (&lru_lock);
19   list_push_back (&lru_list, &page->lru);
20   lock_release (&lru_lock);
21 }
```

page 구조체를 lru_list에 추가하는 함수이다. list에 insert하는 동안 race condition 방지하기 위해 lock으로 보호한다.

vm/frame.c

```
23 void del_page_from_lru_list (struct page *page) {
24   if (lru_clock == &page->lru) {
25     lru_clock = list_next (&page->lru);
26   }
27   list_remove (&page->lru);
28 }
```

del_page_from_lru_list() 함수는 lru_list에서 제거해주는 함수이다. lru_clock이 현재 제거하려는 page 구조체의 lru이라면 lru_clock을 다음 lru으로 바꾸어주고 현재 lru를 list에서 제거한다.

vm/frame.c

```
30 static struct list_elem *get_next_lru_clock (void) {
31   //lock_acquire (&lru_lock);
32   if (lru_clock == NULL) {
33     if (list_empty (&lru_list))
34       return NULL;
35     else
36       lru_clock = list_begin (&lru_list);
37   } else {
38     do {
39       if (lru_clock == list_end (&lru_list))
40         lru_clock = list_begin (&lru_list);
41       else
42         lru_clock = list_next (lru_clock);
43     } while (lru_clock == list_end (&lru_list));
44   }
45   //lock_release (&lru_lock);
46   return lru_clock;
47 }
```

clock 알고리즘의 시곗바늘을 옮겨주는 역할을 하는 함수이다. 현재 clock이 나타내는 lru list_elem이 NULL로 초기화 되어있는 경우 lru_list가 비어있지 않을 때 list의 맨 앞의 list_elem을 저장한다. lru_clock을 옮기면서 lru_clock이 list의 마지막(tail)을 가리키지 않도록 while 문 내에서 처리한다.

vm/frame.c

```
162 void free_page (void *kaddr) {
163   struct list_elem *e = NULL;
164   struct page *page = NULL;
165   lock_acquire (&lru_lock);
166   for (e = list_begin (&lru_list); e != list_end (&lru_list); e = list_next (e)) {
167     page = list_entry (e, struct page, lru);
168     if (page->kaddr == kaddr)
169       break;
170   }
171
172   if (page != NULL) {
173     __free_page (page);
174   }
175   lock_release (&lru_lock);
176 }
```

물리 페이지를 해제하는 함수다. demand paging에 의해 물리 페이지가 할당되었다면 lru_list에 page 구조체가 있으므로 lru_list를 순회하여 물리 페이지 주소에 맞는 page 구조체를 찾는다. page를 찾으면 __free_page() 함수에서 page 구조체에 대한 해제가 수행된다.

vm/frame.c

```
178 void __free_page (struct page* page) {
179   pagedir_clear_page (page->thread->pagedir, page->vme->vaddr);
180   del_page_from_lru_list (page);
181   palloc_free_page (page->kaddr);
182   free (page);
183 }
```

page 구조체를 해제하기 위해 물리 페이지의 present bit를 0으로 set하여 무효화하기 위해 pagedir_clear_page() 함수를 호출한다. 다음에 lru_list에서 해당 page 구조체를 제거하고 할당된 물리 페이지를 해제한다. 그 다음 page 구조체에 할당된 동적 메모리를 해제한다.

vm/page.c

```
147 struct page *alloc_page (enum palloc_flags flags) {
148   struct page *page = NULL;
149   page = (struct page*)malloc (sizeof (struct page));
150   if (page == NULL) {
151     return NULL;
152   }
153   memset (page, 0x00, sizeof (struct page));
154   page->thread = thread_current ();
155   page->kaddr = palloc_get_page (flags);
156   if (page->kaddr == NULL) {
157     page->kaddr = try_to_free_pages (flags);
158   }
159   return page;
160 }
```

page 구조체를 생성하고 kaddr 멤버에 새로 할당된 물리 페이지 PFN를 저장하고 현재 thread 정보를 thread 멤버에 저장하여 page 구조체 포인터를 반환한다. palloc_get_page() 함수에 의해 물리 페이지를 할당 받는데 실패한 경우 page pool이 가득 찬 것이므로 try_to_free_pages() 함수로 swapping을 통해 물리 페이지 공간을 확보하여 물리 페이지를 할당 받을 수 있게 한다.

vm/page.c

```
49 void *try_to_free_pages (enum palloc_flags flags) {
50   struct page *victim_page = NULL;
51   lock_acquire (&lru_lock);
52   while (1) {
53     struct list_elem *elem = get_next_lru_clock ();
54     victim_page = (struct page*)list_entry (elem, struct page, lru);
55     if (pagedir_is_accessed (victim_page->thread->pagedir, victim_page->vme->vaddr)) {
56       pagedir_set_accessed (victim_page->thread->pagedir, victim_page->vme->vaddr, false);
57     } else if (pagedir_get_page (victim_page->thread->pagedir, victim_page->vme->vaddr) != NULL) {
58       /* victim_page found */
59       break;
60     }
61   }
62
```

페이지 풀이 가득 찼을 경우 호출되며, victim page를 선정하고 해당 page를 swap-out 시켜서 물리 페이지 공간을 확보하고 확보된 페이지를 새로 할당하여 반환한다.

victim page는 lru_clock이 lru_list를 순회하면서 lru list_elem에 해당하는 page 구조체의 저장된 물리 페이지 PFN를 해당 page에 저장된 thread 구조체 포인터를 참조하여 pagedir로부터 PTE를 찾아 access bit를 검사한다. access bit 가 set 되어있으면 최근에 사용되어져 사용률이 많은 페이지라 간주하고 access bit가 0인 page가 찾아질 때까지 lru_list를 순회한다. 순회하면서 access bit는 0으로 set 한다.

```
64   if (victim_page->vme->type == VM_ANON) {
65     victim_page->vme->swap_slot = swap_out (victim_page->kaddr);
66   } else if (pagedir_is_dirty (victim_page->thread->pagedir, victim_page->vme->vaddr)) {
67     switch (victim_page->vme->type) {
68       case VM_FILE :
69         lock_acquire (&rw_lock);
70         file_write_at (victim_page->vme->file, victim_page->vme->vaddr,
71                        victim_page->vme->read_bytes, victim_page->vme->offset);
72         lock_release (&rw_lock);
73         break;
74       case VM_BIN :
75         victim_page->vme->swap_slot = swap_out (victim_page->kaddr);
76         victim_page->vme->type = VM_ANON;
77         break;
78     }
79   }
80   victim_page->vme->is_loaded = false;
81   __free_page (victim_page);
82   lock_release (&lru_lock);
83
84   return palloc_get_page (flags);
85 }
```

해당 victim page의 vm_entry type이 VM_ANON이면 해당 페이지는 swap partition으로 무조건 swap out 한다.

그 이외에 type이 VM_FILE 인 경우 swap partition으로 swap-out할 필요 없이 원래 파일에 write-back을 한다. 이 과정은 해당 페이지에 변경사항이 있을 때만 수행한다.

vm_entry의 type이 VM_BIN 인 경우 실행파일은 프로세스 실행 중인 동안 write를 하지 못하게 되어있다. 그래서 전역변수나 static 변수가 저장되는 data 세그먼트는 변경내역이 있으면 vm_entry의 type을 VM_ANON으로 전환하고 swap partition에 swap out 될 수 있도록 한다.

userprog/process.c :: handle_mm_fault()

```
 92   switch (vme->type) {
 93     /* VM_BIN일 경우 load_file()함수를 이용해서 물리메모리에 로드 */
 94     /* install_page를 이용해서 물리페이지와 가상페이지 맵핑 */
 95     case VM_BIN :
 96     case VM_FILE :
 97       /* load_file(), install_page() 수행 중 false 반환 되는 경우 예외처리 */
 98       if (!(load_file (page->kaddr, vme) &&
 99             install_page (vme->vaddr, page->kaddr, vme->writable)))  {
100         __free_page (page);
101         return false;
102       }
103       vme->is_loaded = true;
104       add_page_to_lru_list (page);
105       break;
106     case VM_ANON :
107       swap_in (vme->swap_slot, page->kaddr);
108       if (!install_page (vme->vaddr, page->kaddr, vme->writable)) {
109         __free_page (page);
110         return false;
111       }
112       vme->is_loaded = true;
113       add_page_to_lru_list (page);
114       break;
115   }
116   /* 로드 성공 여부 반환 */
117   return true;
118 }
```

swap out 된 페이지에 다시 메모리 참조가 발생하면 swap partition에서 다시 swap in이 될 수 있도록 handle_mm_fault() 함수를 수정하였다. 새로 할당 받은 물리 페이지에 swap partition 으로부터 page out 된 데이터를 읽어 들여오는 swap in을 수행하고 install_page 함수를 통해 page table에 해당 VPN에 해당하는 PTE에 물리 페이지 주소가 매핑 될 수 있도록 한다. 이 과정을 수행하고 마지막으로 새로 생성된 page 구조체를 lru list에 추가하기 위해 add_page_to_lru_list() 함수를 호출한다.

make check 결과

```
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
pass tests/filesys/base/syn-write
9 of 109 tests failed.
make[1]: *** [check] 오류 1
make[1]: Leaving directory `/home/ckdqja0225/project/src_changbeom/vm/build'
make: *** [check] 오류 2
ckdqja0225@ubuntu:~/project/src_changbeom/vm$ make check | grep FAIL
FAIL tests/vm/pt-grow-stack
FAIL tests/vm/pt-grow-pusha
FAIL tests/vm/pt-big-stk-obj
FAIL tests/vm/pt-grow-stk-sc
FAIL tests/vm/page-parallel
FAIL tests/vm/page-merge-seq
FAIL tests/vm/page-merge-par
FAIL tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
make[1]: *** [check] 오류 1
make: *** [check] 오류 2
ckdqja0225@ubuntu:~/project/src_changbeom/vm$
```

bitbucket 주소

git clone

https://seomooneunji@bitbucket.org/pintos_ejcb/project.git

에서 **" src_changbeom 폴더 "** 입니다.