CSED312 OS Lab 3 - Virtual Memory

Design Report

20180085 송수민 20180373 김현지

Introduction

이번 프로젝트의 목표는 Virtual Memory를 구현하는 것이다. 아래 사항들을 구현함으로써 목표를 달성 할 수 있을 것이다.

1. Frame Table
2. Lazy Loading
3. Supplemental Page Table
4. Stack Growth
5. File Memory Mapping
6. Swap Table
7. On Process Termination

I. Anaylsis on Current Pintos system

현재 Pintos의 상황은 Project 2에서 보았듯이 load와 load_segment에서 program의 모든 부분을 Physical Memory에 적재한다. 이는 매우 비효율적으로 모든 부분의 데이터를 올리는 대신, Progress 중에 어떠한 데이터가 필요하다면 해당 데이터를 Physical Memory에 올리는 것이 효율적일 것이다. Project 1의 Alarm clock과 비슷한 맥락이다. 이 방법을 Lazy Loading이라 한다. 이를 구현하기 위해서는 Virtual Memory의 구현이 필요하다.

Analysis

각 Process는 위와 같은 Address Space를 가지는데, 내부에 Stack, BSS, Data, Code의 영역을 가진다. 현재 Pintos의 Memory Layout은 아래와 같다.

```
31                 12 11       0
+------------------+--------+
| Page Number | Offset |
+-----------------+--------+
```

위를 Virtual Address라 하는데, 현재 구현 되어있는 PTE(Page Table Entry)가 VA를 Physical Address로 변환하여 가리켜준다. PA의 형태는 아래와 같다.

```
31                 12 11       0
+------------------+--------+
| Frame Number | Offset |
+-----------------+--------+
```

Current Pintos System은 process_exec() -> load()-> load_segment() -> setup_stack()을 거쳐 Physical Memory에 data를 적재하였다. load_segment()를 살펴보자.

```
static bool load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
  ASSERT (pg_ofs (upage) == 0);
  ASSERT (ofs % PGSIZE == 0);

  file_seek (file, ofs);
  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* Calculate how to fill this page.
         We will read PAGE_READ_BYTES bytes from FILE
         and zero the final PAGE_ZERO_BYTES bytes. */
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      /* Get a page of memory. */
      uint8_t *kpage = palloc_get_page (PAL_USER);
      if (kpage == NULL)
        return false;

      /* Load this page. */
      if (file_read (file, kpage, page_read_bytes) != (int) page_read_bytes)
        {
          palloc_free_page (kpage);
          return false;
        }
      memset (kpage + page_read_bytes, 0, page_zero_bytes);

      /* Add the page to the process's address space. */
      if (!install_page (upage, kpage, writable))
        {
          palloc_free_page (kpage);
          return false;
        }

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

위에서 볼 수 있듯이, Program 전체를 memory에 load하고 있다. 이를 해결하기 위해 Lazy loading을 구현 할 것이다. Lazy loading은 사용해야 할 부분만 load하고, 당장 사용하지 않는 부분은 일종의 표시만 해두는 것이다. Size가 작은 Program이라면 문제의 영향이 작을 수 있겠지만, Size가 큰 프로그램이라면 Lazy Loading을 통하여 모두 load하지 않고, 필요 시에 memory에 올리면 되므로 효율적이다. 이를 구현하기 위해 우리는 Page의 개념

을 적용하려 한다. 만약 100이란 size의 memory가 주어졌고, 날 것으로 활용한다면, 중간중간 빈 공간이 발생하고 이는 메모리의 낭비를 발생시킬 것이다. 이 메모리에 들어갈 수 있는 크기를 10으로 나누어 놓는다면 딱 맞는 size의 메모리에 넣거나 할 수 있는데 이 또한 효과적이지 못해 frame 10개를 만들고 각 프로세스가 가지는 메모리의 크기도 size 10의 page로 나누어 두어 효율성을 높일 수 있다. Lazy Loading에서 필요한 page만 메모리에 올리고 필요 할 때 마다 disk에서 page를 올리면 해결될 것이다. 결국, Project 3에서는 Memory Management를 구현 하는 것인데, 사용되고 있는 Virtual / Physical Memory 영역에 대해 추적을 한다는 것과 같은 말일 것이다. 또한, Page Fault에 대해서 알아보자. 현재 Page Fault는 아래와 같다.

```
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  if (!user || is_kernel_vaddr(fault_addr) || not_present ) exit(-1);

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

Current Pintos는 모든 segment를 Physical Page에 할당하므로 page fault 시 process를 kill하여 강제 종료 시킨다. 이를 바람직하지 못하며 구현하고자 하는 방향으로 생각해보면, disk에 있는지 아니면 구현 할 다른 data structure에 사용되지 않았다고 표시를 남긴채 존재하는지를 검사하여, 이를 memory에 올려서 계속 progress를 이어나갈 수 있도록 해야한다. 위 사항들을 구현하기 위해, 현재 page address가 어떠한 방식으로 mapping 되는지 알아보자.

```
static bool install_page (void *upage, void *kpage, bool writable)
{
  struct thread *t = thread_current ();

  /* Verify that there's not already a page at that virtual
     address, then map our page there. */
  return (pagedir_get_page (t->pagedir, upage) == NULL
          && pagedir_set_page (t->pagedir, upage, kpage, writable));
}
```

현재 pintos의 page table에 PA와 VA를 Mapping 시켜주는 함수이다. 위 함수에서 kpage가 PA를 가리키고, upage가 VA를 가리킨다. Writable은 true이면 쓰기 가능이고, false이면 읽기 전용인 page이다. 또한, pagedir이라는 것이 존재하는데, 이는 모든 VA Page에 대하여 entry를 사용한다면 접근이 비효율적이므로, 한 단계 상위 개념인 page directory를 사용하는 것이다. Project 1의 thread elem과 elem_list와 비슷한 맥락이라고 보여진다. Page directory는 Page table의 address를 가지고 있는 table이다. Page Table은 VA -> PA인 PA를 가지고 있는 Entry들의 모임이다. 또한, 현재 PA를 할당 및 해제하는 방법은 palloc_get_page(), palloc_free_page()를 이용한다.

```
void * palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)
{
  struct pool *pool = flags & PAL_USER ? &user_pool : &kernel_pool;
  void *pages;
  size_t page_idx;

  if (page_cnt == 0)
    return NULL;

  lock_acquire (&pool->lock);
  page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);
  lock_release (&pool->lock);

  if (page_idx != BITMAP_ERROR)
    pages = pool->base + PGSIZE * page_idx;
  else
    pages = NULL;

  if (pages != NULL)
  {
    if (flags & PAL_ZERO)
      memset (pages, 0, PGSIZE * page_cnt);
  }
  else
  {
```

```
    if (flags & PAL_ASSERT)
      PANIC ("palloc_get: out of pages");
  }

  return pages;
}
/*palloc_get_page는 palloc_get_multiple을 호출하는 하나의 line으로 이루어져 있다.*/
```

palloc_get_page는 4KB의 Page를 할당하고, 이 page의 PA를 return 한다. Prototype에 flag를 받는데, PAL_USER, PAL_KERNEL, PAL_ZERO가 있다. 각각의 설명은 아래와 같다.

> - PAL_USER : User Memory (0~PHYS_BASE(3GB))에 Page 할당.
> - PAL_KERNEL : Kernel Memory (>PHYS_BASE)에 Page 할당.
> - PAL_ZERO : Page를 0으로 initialization. Palloc_free_page()는 page의 PA를 Parameter로 사용하며, page를 재사용 할 수 있는 영역에 할당한다.

```
/* Frees the PAGE_CNT pages starting at PAGES. */
void palloc_free_multiple (void *pages, size_t page_cnt)
{
  struct pool *pool;
  size_t page_idx;

  ASSERT (pg_ofs (pages) == 0);
  if (pages == NULL || page_cnt == 0)
    return;

  if (page_from_pool (&kernel_pool, pages))
    pool = &kernel_pool;
  else if (page_from_pool (&user_pool, pages))
    pool = &user_pool;
  else
    NOT_REACHED ();

  page_idx = pg_no (pages) - pg_no (pool->base);

#ifndef NDEBUG
  memset (pages, 0xcc, PGSIZE * page_cnt);
#endif

  ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));
  bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
}
/*palloc_free_page는 palloc_free_multiple을 호출하는 하나의 line으로 이루어져 있다.*/
```

현재 Pintos의 stack 크기는 4KB로 고정되어 있다. 이 영역의 크기를 벗어나면 현재는 Segmentation fault를 발생 시킨다. 이 Stack의 크기가 일정 조건을 충족한다면 확장하는 방향으로 구현하고자 한다.

> -> Stack의 size를 초과하는 address의 접근이 발생하였을 때

> - Valid stack access or Segmentation Fault 판별 기준 생성
>   Valid stack access -> Stack size expansion (Limit max = 8MB)

이제 아래 영역에서는 각 요소의 구현 방법을 나타낼 것이다.

II. Frame Table

Analysis

현재 pintos는 frame table에 대해 구현된 사항이 없다.

Solution

Frame을 효율적으로 관리하기 위해 필요한 virtual memory frame table을 구현해야한다. table의 각 entry는 user program의 page하나에 대응되고, 각각의 thread마다 frame table을 가지고 있어야한다. 각 table은 탐색이 빠른 hash로 구현하며, vaddr로 hash값을 추출한다. hash와 관련된 code는 src/lib/kernel/hash.*에 정의되어있어, 이를 사용하면 된다.

To be Added & Modified

```
/* vm/page.h */
struct vm_entry{
  uint8_t type; /* VM_BIN, VM_FILE, VM_ANON의 타입 */
  void *vaddr; /* virtual page number */
  bool writable; /* 해당 주소에 write 가능 여부 */
  bool is_loaded; /* physical memory의 load 여부를 알려주는 flag */
  struct file* file; /* mapping된 파일 */
  struct hash_elem elem; /* hash table element */

  size_t offset; /* read 할 파일 offset */
  size_t read_bytes; /* virtual page에 쓰여져 있는 데이터 byte 수 */
  size_t zero_bytes; /* 0으로 채울 남은 페이지의 byte 수 */
}
```

> frame table의 entry를 위와 같이 정의한다.

```
/* threads/thread.h*/
struct thread {
  ...
  #ifdef USERPROG
    ...
    struct hash vm;     /* thread가 가진 virtual address space를 관리하는 hash
table */
  #endif
```

```
  ...
}
```

> 각 thread마다 frame을 관리하는 table을 hash type으로 추가한다.

```
/* vm/page.c */
void vm_init (struct hash *vm) {
  //hash_init() 함수를 이용하여 vm (hash table)을 init
}

//참고 /* hash.c */
bool hash_init (struct hash *h, hash_hash_func *hash, hash_less_func *less, void
*aux)
```

> frame table을 초기화하는 함수를 추가한다

```
/* vm/page.c */
void vm_destroy (struct hash *vm) {
  // hash_destroy() 함수를 이용하여 vm (hash table)을 destroy
}

//참고 /* hash.c */
void hash_destroy (struct hash *h, hash_action_func *destructor)
```

> frame table을 destroy하는 함수를 추가한다

```
/* vm/page.c */
static unsigned vm_hash_func (const struct hash_elem *e, void *aux) {
  // element에 대한 vm_entry를 hash_entry() 함수를 이용하여 찾고,
  // 해당 entry의 vaddr에 대한 hash key를 hash_int() 함수를 이용하여 구한 후 return
}
```

> element에 대한 hash key를 반환하는 함수를 추가한다
> vm_init에 필요하다

```
/* vm/page.c */
static bool vm_less_func (const struct hash_elem *a, const struct hash_elem *b,
void *aux) {
  // 두 element에 대한 vm_entry를 hash_entry() 함수를 이용하여 찾고, 각 entry의 vaddr
을 비교
}
```

> 두 element에 대한 vm_entry의 vaddr값을 비교하는 함수를 추가한다 (a가 작으면 true, 크면 false return)
> vm_init에 필요하다

```
/* vm/page.c */
static void
vm_destroy_func(struct hash_elem *e, void *aux UNUSED)
{
  // element에 대한 hash_entry() 함수를 이용하여 찾고, vm_entry를 해당 entry를 free
}
```

> element에 대한 vm_entry를 free시키는 함수를 추가한다
> vm_destroy에 필요하다

```
/* vm/page.c */
bool insert_vme(struct hash *vm, struct vm_entry *vme) {
  // hash_insert() 함수를 이용하여 vm (hash table)에 vme (entry)를 insert
}
```

> hash table에 vm_entry를 삽입하는 함수를 추가한다

```
/* vm/page.c */
bool delete_vme(struct hash *vm, struct vm_entry *vme) {
  // hash_delete() 함수를 이용하여 vm (hash table)에서 vme (entry)를 delete
}
```

> hash table에서 vm_entry를 삭제하는 함수를 추가한다

```
/* vm/page.c */
struct vm_entry *find_vme (void *vaddr) {
  // pg_round_down() 함수를 이용하여 vaddr의 page number를 구하고,
  // hash_elem을 hash_find() 함수를 이용해서 찾는다.
  // 해당 hash_elem에 대한 vm_entry 를 hash_entry()를 이용하여 구한 후 return */
}
```

> vaddr에 해당하는 hash_elem의 vm_entry를 찾아주는 함수를 추가한다

```
/* userprog/process.c */
static void start_process (void *file_name_)
```

> process가 시작할 때, vm_init()를 호출하는 부분을 추가한다.

```
/* userprog/process.c */
void process_exit (void)
```

> process가 종료될 때, vm_destroy()를 호출하는 부분을 추가한다.

III.Lazy Loading

Analysis

현재 pintos는 process execute에 필요한 Disk의 file에 대해 바로 physical memory에 page를 할당하여 load하고, page fault가 발생하면 error로 간주하여 program 실행을 중지한다. 사용하지 않는 data가 load될 경우, physical memeory를 차지하여 메모리가 낭비된다. 따라서 virtual memory의 page를 이용하여 필요한 data만 load되도록 구현하고, page fault handler도 수정해야한다.

Solution

처음에는 physical memory에 어떤 것도 load하지 않고, 각 file의 pointer 및 offset, size 등의 정보는 vm_entry에 저장한다. process execute를 통해 특정 virtual address에 접근할 때 physical page가 mapping되어 있지 않다면, page fault가 발생한다. page fault handler를 통해 접근이 시도된 vm_entry를 탐색한 후, vm_entry에 저장된 정보를 통해 data를 읽어 physical frame에 load하도록 구현한다. 또한, page fault handler에서 Lazy Loading(I/O가 필요)과 Wrong memory access(I/O가 필요X) 각각의 상황이 동시에 발생할 경우, 후자의 상황에 대해 먼저 처리하고 전자의 상황을 처리해야한다.

To be Added & Modified

```
/* userprog/process.c */
static bool load_segment (struct file *file, off_t ofs, uint8_t *upage, uint32_t
read_bytes, uint32_t zero_bytes, bool writable)
```

> load_segment는 segment를 process의 virtual address space에 load하는 함수이다.
> 현재는 load_segment가 실행되면 page를 할당하고, file을 page에 load하고 process's address space에 page를 저장하도록 구현되어있다.
> 이를 vm_entry를 할당하고 초기화 하며, vm (hash table)에 insert하도록 수정한다.

```
/* userprog/process.c */
static bool setup_stack (void **esp)
```

> setup_stack은 stack을 초기화하는 함수이다.
> 현재는 page를 할당하고, install page함수를 통해 user virtual address UPAGE에서 kernel virtual address KPAGE로의 mapping을 page table에 추가한 후, esp를 설정하는 것까지 구현되어있다.
> 이후에 vm_entry를 생성 및 초기화하고, vm (hash table)에 insert하는 부분을 추가한다.

IV. Supplemental Page Table

Analysis

현재 pintos는 Lazy loading, file memory mapping, swap table이 정상적으로 작동하도록 하는 함수 구현이 전혀 되어있지 않다. Lazy loading, file memory mapping, swap table이 모두 잘 작동할 수 있도록 구현이 수정되어야한다.

Solution

현재 pintos는 page fault 발생시 처리를 위해 page_fault()라는 함수가 존재한다. 이는 I의 Analysis에서 명시하였듯, 무조건 "segmentation fault"를 발생시키고 kill(-1)을 하여 강제종료하도록 구현되어있다. 이를 vm_entry type에 맞게 처리되도록 수정한다.

To be Added & Modified

```
static void page_fault (struct intr_frame *f)
```

> find_vme를 이용하여 vm_entry를 찾은 후 해당 entry에 대해 page fault를 handle하는 함수를 호출한다.

```
bool handle_mm_fault(struct vm_entry *vmentry)
```

> handle_mm_fault는 page fault시 이를 handle하기 위해 필요한 함수이다. page fault 발생시 palloc_get_page() 함수를 이용하여 physical page를 할당하고, switch문으로 vmentry type별로 처리한다. 이 단계에서는 일단 먼저 VM_BIN에 대해서만 구현하고, 다른 type에 대해서는 아래의 단계에서 추가로 구현한다. VM_BIN일 경우 load_file()함수를 이용해서 physical page에 Disk에 있는 file을 load한다. 각 타입에 대해 load가 완료되었으면, install_page() 함수를 이용하여 page table로 virtual address와 physical address를 mapping한 후 그 성공 여부를 반환한다.

```
bool load_file (void* kaddr, struct vm_entry *vmentry)
```

> disk의 file을 physical memory로 load하는 함수이다. file_read()함수를 이용하여 physical page에 data를 load하고 남은 부분은 0으로 채운다.

V. Stack Growth

Analysis

Current Pintos System에는 1 page(4KB)로 fixed되어, Stack 확장이 불가능하게 구현되어있다. 따라서, Stack 확장이 가능하도록 구현하여야한다.

Solution

stack pointer esp가 유효한 stack의 영역을 벗어나면 segmentation fault가 발생한다. 먼저, 현재 stack size를 초과하는 주소에 접근이 발생했을 때, page fault handler에서 stack확장이 필요한지 판별하게 한다. 필요한 경우로 최대 limit이내의 접근일 경우 valid한 stack 접근으로 간주하며, interupt frame에서 esp를 가져온다. 이때 필요한 page 수를 계산하는 과정이 필요하다. 또한 pintos document에 명시된대로, 최대 8MB까지 stack을 확장할 수 있도록 수정한다.

To be Added & Modified

```
/* userprog/process.c */
bool expand_stack(void *addr)
```

> addr을 포함하도록 stack을 확장시키는 함수를 추가한다.

```
/* userprog/process.c */
bool verify_stack(void *sp)
```

> sp(address)가 포함되어 있는지 확인하는 함수를 추가한다.

```
/* userprog/exception.c */
static void page_fault (struct intr_frame *f)
```

> verify stack()을 이용하여 address가 stack 영역에 포함되어 있는지 확인한 후, expand_stack()을 이용하여 stack을 확장시킨다.

VI. File Memory Mapping

Analysis

Current Pintos System에는 System Call : mmap(), mummap()이 없다. 따라서, file memory mapping이 불가능하다. 또한, Pintos document에 의거하여 read & write system call을 사용하지 말고, mmap을 이용하여 vm과 mapping한다고 한다. 이를 통해, 동일한 파일을 여러 process들의 vm에 mapping하여 접근 가능한 것이다. 이를 이루기 위해 위의 system call을 구현한다. mmap()과 mummap()의 역할은 아래와 같다.

- mmap() : Lazy Loading에 의한 File Data를 memory에 load.
- munmap() : mmap으로 형성 된 file mapping을 delete. 이를 위해서는 mapping된 file들의 정보를 담을 수 있는 structure가 필요하다. 이 structure가 담아야 할 data는 아래와 같다.
- mapid : mmap() success시 mapping id를 return하는데, 이를 담을 variable.
- file : mapping하는 file의 file Object.
- elem : 이 data structure을 관리할 list가 필요하다. 이를 위해 리스트 연결을 할 구조체.
- vme_list : 이 data에 해당하는 모든 vm_entry의 list. 추가로, file memory mapping을 완전히 구현하기 위해서 file을 VA로 관리한다. VA를 관리 하기 위해 Virtual Address entry를 이용하여 hash_table로 관리한다.

To be Added & Modified

```
int mmap(int fd, void *addr)
```

> Lazy loading에 의해 file data를 memory에 load한다. fd는 process의 VA Space에 mapping할 file descriptor이고, addr은 mapping을 시작 할 주소이다. 이는 성공 시 mapping id를 return 하고, 실패 시 error -1을 return한다. 또한, mmap으로 mapping이 된 file은 mummap, process의 terminate 이전에는 접근이 가능해야 한다. 예로, munmap() 이전에 close() system call이 호출 되더라도 file mapping은 유효하여야 한다는 것이다. 이를 위해, filsys/file.c에 있는 method를 사용한다.

```
struct file *file_reopen (struct file *file)  {
  return file_open (inode_reopen (file->inode));
}
```

> File object를 copy하여 copy된 object의 주소를 return한다. 끝으로, process의 file descriptor를 탐색하기 위해 project 2에서 구현한 method를 사용하고자 한다.

```
int munmap(mapid_t mapid)
```

> 위에서 언급한 structure 내에서 mapid에 해당 되는 vm_entry를 해제하는 system call이다. 이때, 모든 mapid value가 close_all인 경우 모든 file mapping을 제거한다.

```
bool handle_mm_fault(struct vm_entry *vmentry)
```

> Supplemental Page Table에서 구현하였던 handle_mm_fault() 함수의 Switch-case 문에서 VM_FILE에 해당하는 case를 추가적으로 구현한다. vmentry type이 VM_FILE인 경우 data를 load할 수 있도록 한다. 추가적으로, process_exit시에 munmap이 호출되지 않음에 따라, delete되지 않아 메모리 누수를 일으킬 있는 것들을 제거한다. Project 2에서 orphan process와 비슷한 맥락의 경우이다.

VII. Swap Table

Analysis

Frame을 새로 allocation 할 때 메모리가 부족하여 할당이 실패하는 경우가 존재한다. 이 경우에 Swapping이 필요한데, Swapping을 하기 위해 Swap disk와 Swap table이 필요하다. 원리는 간단하다. 메모리에 공간이 부족하다면 Swap table에 frame을 넣고 그 공간을 사용하는 것이다. 그렇다면 어떤 frame을 swap table에 넣어야 하는지는 Policy가 필요하다. Policy는 LRU Algorithm, Clock Algorithm 등이 있는데, LRU Algorithm은 Least Recently Used의 약자로 말 그대로 가장 오래전에 사용된 항목을 선택하는 것이고, Clock Algorithm은 frame을 순회하면서 어떠한 bit가 1이면 해당 frame을 swapping하는 것이다. 본 과제에서는 Clock Algorithm이 bit를 사용하여 구현하기 전에는 좀 더 구현이 용이할 것으로 보이기도 하고, 가시적이어서 해당 algorithm을 사용하려고 한다. 현재 Pintos의 swap partition은 4MB이며, 4KB로 나누어 관리를 한다. 이 Partition의 frame들을 연결해 줄 필요가 있기 때문에, Swap table은 list type으로 구현한다.

Solution

크게 3가지의 method가 필요할 것으로 생각한다.

1. swap_init()
    - Swapping을 다룰 부분을 initialization 하는 method이다.
2. swap_in()
    - 메모리가 부족하여 swap table로 빼두었던 frame을 다시 메모리에 올리는 method이다.
3. swap_out()
    - 메모리가 부족하여 swap table로 frame을 빼는 method이다.

To be Added & Modified

```
- void swap_init()
```

> Swapping을 다룰 영역을 initialization한다.

```
- void swap_in(size_t used_index, void *kaddr)
```

> used_index의 swap table 공간에 있는 data를 kaddr로 넣어준다. 이는 frame을 다시 메모리에 적재하는 역할을 할 것이다.

```
- size_t swap_out(void *kaddr)
```

> 사용 가능한 memory가 존재하지 않을 때, Clock algorithm에 의해 선정된 victim frame을 swap partition으로 넣어준다. Dirty bit를 확인하여 true라면 write back을 하여 disk에 기록한다.

```
- static struct list_elem* next_victim_frame()
```

> 다음 Swapping 시 행해질 victim frame을 탐색 및 선정하는 method이다.

```
- bool handle_mm_fault(struct vm_entry *vmentry)
```

> Supplemental Page Table에서 구현하였던 handle_mm_fault() 함수의 Switch-case 문에서 VM_ANON에 해당하는 case를 추가적으로 구현한다. 이 type은 swap partition으로 부터 data를 load하기 때문에 이 handler에서 swapping in을 해주어야 한다.

VIII. On Process Termination

Solution

On process termination에서 요구하는 것은 Process가 종료될 때 할당한 모든 resourece들, 예로 frame table과 같은 것들을 delete 해주라는 것이다. 단, Copy on Write가 필요한 page는 dirty bit를 판단 기준으로 삼아 disk에 써준다. 이를 위해서 System call : munmap을 구현하여 사용하고자 한다.

To be Added & Modified

```
void process_exit (void)
```

> process가 종료되는 시점에서 해당 thread의 자원을 모두 해제해야 한다. 따라서, process_exit을 수정하여 구현하고자 하고, munmap을 사용하여 current thread의 resource들을 반복문을 통해 순회하며 해제한다.

```
void munmap (mapid_t);
```

> munmap에 Copy on write를 추가한다. pagedir.c에 있는 pagedir_is_dirty()를 사용하여 dirty bit를 판단하고, dirty == true라면 disk에 write back, false라면 바로 해제할 수 있도록 구현한다.