Pintos Project: 3. Virtual Memory

Final Report – team 30

20210054 정하우

20210716 최대현

Context

1. **Implementation**

   1-1. Frame Table

   - Algorithm

   - Data Structure

   - Function Implementation

   1-2. Lazy Loading

   - Algorithm

   - Function Implementation

   1-3. Supplemental Page Table

   - Algorithm

   - Data Structure

   - Function Implementation

   1-4. Stack Growth

   - Algorithm

   - Function Implementation

1. Implementation

1-1. Frame Table

Algorithm

- Pintos에서 현재는 모든 데이터를 PM에 load하고 있는데, 이는 매우 비효율적인 구현이므로 Lazy loading을 구현하여 필요한 데이터만을 PM에 load를 해야 한다. Lazy loading을 구현하기 위해서는 PM가 가득 찬 상황에서 page를 load할 경우 기존에 PM에 load되어 있는 page 중 하나를 선택하여 제거하는 기능이 필요하다.
- 2. Lazy loading 파트에서 Lazy loading을 위해 PM에 필요한 데이터만 load하는 기능을 추가할 예정이다. 이를 위해 page 정보를 리스트 형태로 관리하고, 리스트에 page를 추가/삭제하는 함수를 구현한다.
- 또 page 할당 및 해제 함수를 새롭게 구현하여 단순히 page를 할당/해제 만 하는 것이 아닌 해당 리스트에서 page 정보를 추가하거나 제거하는 함수를 구현하여 기존 page 할당/해제 함수를 대체한다.
- 또한, Eviction을 위한 정보를 저장하기 위해 전역 변수로 LRU 리스트를 구현하고, 이 리스트 정보를 업데이트할 때 다른 프로세스의 접근을 막기 위한 Lock도 추가한다. 그리고 어떤 page가 다음으로 제거될지 결정하는 데 사용되는 page 포인터도 추가한다.

Data Structure

1. **src/vm/frame.h/struct page**

```
struct page {
    void *kaddr;
    struct vm_entry *vme;
    struct thread *thread;
    struct list_elem lru_elem;
};
```

- physical memory에 load되어있는 page에 대한 정보를 담은 struct. 실제 physical memory에서 어디에 위치하는지를 나타내는 kaddr을 가지고 있으며, 이번 프로젝트에서 virtual memory를 사용하기 때문에 해당 physical memory에 매핑된 virtual memory 엔트리 정보인 vme을 가지고 있다. (3. Supplemental Page Table에서 자세히 다룰 예정) 또한, 현재 이 프레임을 사용 중인 thread에 대한 정보와 이 프레임 엔트리를 관리하기 위한 리스트를 포함하기 위해 list_elem을 가지고 있다.

2. **src/vm/frame.c/lru_list**

```
struct list lru_list;
```

- page 정보를 관리하는 list.

3. **src/vm/frame.c/lru_lock**

```
struct lock lru_lock;
```

- lru_list는 actomic하게 추가/삭제 되어야하므로 해당 list 관리를 위한 lock을 추가해야 한다.

4. **src/vm/frame.c/lru_clock**

```
struct list_elem *lru_clock;
```

- 다음으로 evict 될 page를 선택하는데 사용되는 iterator pointer.

Function Implementation

1. **src/vm/frame.c/void lru_list_init(void)**

```
void lru_list_init(void)

{
  list_init(&lru_list);
  lock_init(&lru_lock);
  lru_clock = NULL;
}
```

- 위에서 언급한 세 struct를 초기화 하는 함수.

2. **src/vm/frame.c/void add_page_to_lru_list(struct page *page)**

```
void add_page_to_lru_list(struct page *page)

{
  list_push_back(&lru_list, &page->lru_elem);
}
```

- lru_list에 page를 추가하는 함수.

3. **src/vm/frame.c/void del_page_from_lru_list(struct page *page)**

```
void del_page_from_lru_list(struct page *page)

{
  if (lru_clock == &page->lru_elem)
  {
    lru_clock = list_remove(lru_clock);
  }
  else
  {
    list_remove(&page->lru_elem);
  }
}
```

- lru_list에 page를 삭제하는 함수. lru_clock과 동일한 page를 list에서 삭제하는 경우 lru_clock 값을 업데이트 해 준다.

4. src/vm/page.c/static struct list_elem *get_next_lru_clock()

```
static struct list_elem *get_next_lru_clock()

{
    if (list_empty(&lru_list))
    {
        return NULL;
    }

    if (lru_clock && lru_clock != list_end(&lru_list))
    {
        lru_clock = list_next(lru_clock);
    }

    if (!lru_clock || lru_clock == list_end(&lru_list))
    {
        return (lru_clock = list_begin(&lru_list));
    }
    else
    {
        return lru_clock;
    }
}
```

- 다음 clock을 return하는 함수.
- List가 비어있으면 null을 return한다. lru_clock의 다음 원소로 lru_clock를 업데이트 하

고, 만약 list의 마지막 원소였으면 다시 첫 원소로 업데이트하고, 최종적으로 잘 업데이트 된 lru_clock를 return한다.

5. **src/vm/page.c/void* try_to_free_pages(enum palloc_flags flags)**

```
void* try_to_free_pages(enum palloc_flags flags)

{
    struct page *page;
    struct list_elem *element;
    while (true)
    {
        element = get_next_lru_clock();
        page = list_entry(element, struct page, lru_elem);
        if(page->vme->_pin)
        {
            continue;
        }
        if(pagedir_is_accessed(page->thread->pagedir, page->vme->vaddr))
        {
            pagedir_set_accessed(page->thread->pagedir, page->vme->vaddr,
false);
            continue;
        }
        else
        {
            break;
        }
    }

    bool dirty = pagedir_is_dirty(page->thread->pagedir, page->vme->vaddr);

    if (page->vme->type == VM_FILE&&dirty)
    {
        file_write_at(page->vme->file, page->kaddr, page->vme->read_bytes,
page->vme->offset);
    }
    else if (page->vme->type == VM_ANON||(page->vme->type == VM_BIN&&dirty))
    {
        page->vme->swap_slot = swap_out(page->kaddr);
        page->vme->type = VM_ANON;
    }

    page->vme->is_loaded = false;
    pagedir_clear_page(page->thread->pagedir, page->vme->vaddr);
    del_page_from_lru_list(page);
```

```
    palloc_free_page(page->kaddr);
    free(page);

    return palloc_get_page(flags);
}
```

- Physical memory가 가득 차 있는 경우 page를 eviction하는 함수. 위에서 구혀한 get_next_lru_clock함수를 통해 evict할 page를 찾는다. 해당 page가 pinning 여부를 확인한다. 만약 pinning이 되어 있지 않으면 pagedir_is_accessed() 함수를 통해 accessed bit을 확인한다. 만약 accessed bit가 1이라면 pagedir_set_accessed()함수를 통해 해당 bit를 0으로 set해준다. accessed bit가 0이라면 해당 page를 evict하면 된다.
- 그런데 Lazy loading에서 중요한 점은 Stack 영역의 page는 Eviction되면 안 된다는 것이다. 따라서 이 page가 dirty한지와 type을 확인한다.
- 만약 page가 VM_FILE 타입이라면, 해당 page가 dirty인지 확인한 후 파일이 위치한 메모리 공간을 덮어쓰면서 파일을 덮어씌운다.
- 하지만 VM_ANON 타입이거나 VM_BIN이고 해당 page가 dirty인 상태인 경우 다시 physical memory에 올라올 수 있어야 한다. 이를 위해 스왑 디스크를 사용하여 해당 정보를 저장한다.
- 그리고 physical memory에서 해당 page를해제한 뒤 lru_list에서 page 정보를 제거한다. 이렇게 하면 physical memory가 비게 되어 새로운 page할당이 가능해진다.

6. src/vm/page.c/struct page *alloc_page(enum palloc_flags flags)

```
struct page *alloc_page(enum palloc_flags flags)

{
    if((flags & PAL_USER) == 0)
    {
        return NULL;
    }

    struct page *page;
    void *page_kaddr;

    page_kaddr=palloc_get_page(flags);
    while (!page_kaddr)
    {
        page_kaddr=try_to_free_pages(flags);
```

```
    }

    page = (struct page *)malloc(sizeof(struct page));
    if (!page)
    {
        palloc_free_page(page_kaddr);
        return NULL;
    }

    memset(page, 0, sizeof(struct page));
    page->thread = thread_current();
    page->kaddr = page_kaddr;

    add_page_to_lru_list(page);
    return page;
}
```

- physical memory에 할당할 때 사용하는 함수. Lazy loading을 구현하기 위해서는 기존 palloc_get_page 함수를 사용할 때 lru_list에 해당 page의 정보를 담긴 page를 할당한 후 추가하는 로직이 추가로 필요하다. 이 외에도 physical memory가 가득 찬 경우 기존에 load되어있는 page를 evict하는 로직도 추가되어야 한다. 이 기능들을 하나로 묶은 alloc_page함수를 구현해 palloc_get_page 함수를 대신해 사용하였다.

7. src/vm/page.c/void free_page(void *kaddr)

```
void free_page(void *kaddr)

{
    struct page *lru_page =NULL;
    struct list_elem *element;
    for (element = list_begin(&lru_list); element != list_end(&lru_list);
element = list_next(element))
    {
        lru_page = list_entry(element, struct page, lru_elem);
        if (lru_page->kaddr == kaddr)
        {
            if (lru_page != NULL)
            {
                pagedir_clear_page(lru_page->thread->pagedir, lru_page->vme-
>vaddr);
                del_page_from_lru_list(lru_page);
                palloc_free_page(lru_page->kaddr);
                free(lru_page);
            }
```

```
            break;
        }
    }
}
```

- page를 할당 해제할 때도 마찬가지로 할당 해제뿐만 아니라 lru_list에 해당 page를 삭제하고 page를 할당 해제를 하는 로직이 추가적으로 필요하다. 이 기능을 하나로 묶은 free_page 함수를 구현해 palloc_free_page 함수를 대신해 사용하였다.

1-2. Lazy Loading

Algorithm

- 기존 Pintos는 특정 프로세스를 실행할 때 해당 프로세스의 모든 값을 physical memory에 미리 load한다. 이는 비효율적이며, physical memory보다 큰 메모리를 요구하는 프로세스를 실행할 수 없는 단점이 있어 Lazy loading 방식을 도입하여 필요한 값만 physical memory에 load하여 더 큰 메모리를 요구하는 프로세스도 실행될 수 있도록 변경한다.
- 또, 특정 page가 physical memory에 load되지 않으면 page fault를 발생시키며, 이 경우 프로세스를 강제로 종료한다. Lazy loading 방식에서는 physical memory에 load되지 않았더라도 virtual memory에 값이 존재하는 경우가 있을 수 있다. 이를 고려하여 page fault 함수를 다시 구현해야 한다.
- 기존 Pintos는 프로세스가 실행될 때 모든 값을 physical memory로 미리 load한다. 코드를 살펴보면, process_exec() 함수 내에서 load() 함수가 실행되면서 load_segment() 함수가 호출되어 모든 값을 physical memory에 load하는 것을 확인할 수 있다. 이에 따라 load_segment() 함수를 physical memory로 load하는 것이 아니라 virtual memory로 load하도록 변경해야 한다.
- 또한, 특정 page가 physical memory에 load 되어 있지 않는 경우 바로 page fault를 발생시켜 프로세스를 즉시 종료하는 대신, virtual memory에 값이 있는 경우 load를 시도하도록 구현을 변경해야 한다. page fault 시에 virtual memory에 값이 있으면 이 값을 physical memory로 load해야 한다. 이를 위해 virtual memory가 어떤 정보를 담고 있는지 확인해야 한다. 만약 virtual memory에 값이 이미 physical memory에 존재한다면 load할 필요가 없으며, load되지 않은 경우에만 physical memory에 load하는 추가 구현이 필요하다. 위 기능을 handle_mm_fault 함수로 담아 page_fault함수 내에서 virtual memory를 확인하는 로직을 추가한다.

Function Implementation

1. **src/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)**

```
/* Loads a segment starting at offset OFS in FILE at address
   UPAGE.  In total, READ_BYTES + ZERO_BYTES bytes of virtual
   memory are initialized, as follows:

        - READ_BYTES bytes at UPAGE must be read from FILE
          starting at offset OFS.

        - ZERO_BYTES bytes at UPAGE + READ_BYTES must be zeroed.

   The pages initialized by this function must be writable by the
   user process if WRITABLE is true, read-only otherwise.

   Return true if successful, false if a memory allocation error
   or disk read error occurs. */


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;

      struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct
vm_entry));
      if (!vme)
      {
        return false;
      }

      memset(vme, 0, sizeof(struct vm_entry));
```

```
      vme->type = VM_BIN;
      vme->vaddr = upage;
      vme->writable = writable;
      vme->is_loaded = false;
      vme->_pin=false;

      vme->file = file;
      vme->offset = ofs;
      vme->read_bytes = page_read_bytes;
      vme->zero_bytes = page_zero_bytes;


      insert_vme (&thread_current ()->vm, vme);

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

- load()함수에서 physical memory에 load를 진행하는 함수.
- 기존 pintos와는 다르게 segment들을 physical memory가 아닌 virtual memory에만 load를 한다. Load될 segment의 정보를 virtual memory entry에 저장하고, 현재 thread의 supplemental page table에 추가한다. palloc_get_page 함수는 physical memory에 load되므로 삭제했다.

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

```
/* Create a minimal stack by mapping a zeroed page at the top of
   user virtual memory. */
static bool
setup_stack (void **esp)
{
  lock_acquire(&lru_lock);
  struct page *kpage;

  kpage = alloc_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      if (install_page(((uint8_t *)PHYS_BASE) - PGSIZE, kpage->kaddr, true))
```

```
        *esp = PHYS_BASE;
      else
      {
        free_page(kpage->kaddr);
        lock_release(&lru_lock);
        return false;
      }
    }
    else
    {
      lock_release(&lru_lock);
      return false;
    }

    struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
    if (!vme)
    {
      lock_release(&lru_lock);
      return false;
    }

    memset(vme, 0, sizeof(struct vm_entry));
    vme->type = VM_ANON;
    vme->vaddr = ((uint8_t *)PHYS_BASE) - PGSIZE;
    vme->writable = true;
    vme->is_loaded = true;
    vme->_pin=true;

    vme->file = NULL;
    vme->offset = NULL;
    vme->read_bytes = 0;
    vme->zero_bytes = 0;

    kpage->vme =vme;

    lock_release(&lru_lock);
    return insert_vme(&thread_current()->vm, kpage->vme);
}
```

- process의 stack를 초기화 하는 함수.
- stack는 physical memory에서 빠지면 안되므로 physical memory에 load하는 부분을 삭제하면 안된다. 대신 palloc_get_page 함수 대신 alloc_page함수를 사용해 1.에서 구현한 frame table을 관리를 해 주어야한다.
- Load될 segment의 정보를 virtual memory entry에 저장하고, 현재 thread의

supplemental page table과 physical memory에 load된 page의 virtual memory entry의 정보를 저장해준다.

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

```
/* Page fault handler.  This is a skeleton that must be filled in
   to implement virtual memory.  Some solutions to project 2 may
   also require modifying this code.

   At entry, the address that faulted is in CR2 (Control Register
   2) and information about the fault, formatted as described in
   the PF_* macros in exception.h, is in F's error_code member.  The
   example code here shows how to parse that information.  You
   can find more information about both of these in the
   description of "Interrupt 14--Page Fault Exception (#PF)" in
   [IA32-v3a] section 5.15 "Exception and Interrupt Reference". */
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(!not_present)
```

```
  {
    sys_exit(-1);
  }

   bool load = false;
  struct vm_entry *vme = find_vme (fault_addr);
   if (!vme)
   {
      if (!verify_stack(fault_addr, f->esp))
      {
         sys_exit(-1);
      }
      if (!expand_stack(fault_addr))
      {
         sys_exit(-1);
      }
      return;
   }
   else
   {
      load = handle_mm_fault(vme);
      if(vme->is_loaded == true)
      {
         load = true;
      }
   }


  /* 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. */
     if(!load)
     {
         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);
     }
}
```

- 위에서 말했듯, physical memory에 해당 page가 없을 때 반드시 page fault를 띄우는 기존 pintos를 수정해야 한다.
- 이를 위해서 page_fault 함수 내에서 바로 process를 kill하지 않도록 구현을 해야 하는

데, 이를 위해서 일단 verify_stack() 함수를 통해 fault address가 user address인지를 판단해야한다. 그리고 expand_stack() 함수를 통해 stack영역을 확장한다. (verify_stack 함수와 expand_stack 함수의 경우 추후 4.Stack Growth에서 설명)

- 이 과정을 다 거친 뒤, 아래에서 소개할 handle_mm_fault() 함수를 통해 physical memory로 load를 시도한다. 만약 load가 실패했을 경우, page fault를 발생시켜 process를 kill한다.

4. src/userprog/process.c/bool handle_mm_fault(struct vm_entry *vme)

```
bool handle_mm_fault(struct vm_entry *vme)

{
  lock_acquire(&lru_lock);
  bool success = false;

  struct page *kpage;
  kpage = alloc_page(PAL_USER);
  kpage->vme = vme;

  if(vme->type==VM_BIN)
  {
    if (!load_file(kpage->kaddr, vme))
    {
      free_page(kpage->kaddr);
      lock_release(&lru_lock);
      return false;
    }
  }
  else if(vme->type==VM_FILE)
  {
    if (!load_file(kpage->kaddr, vme))
    {
      free_page(kpage->kaddr);
      lock_release(&lru_lock);
      return false;
    }
  }
  else if(vme->type==VM_ANON)
  {
    swap_in(vme->swap_slot, kpage->kaddr);
  }
  vme->is_loaded = install_page(vme->vaddr, kpage->kaddr, vme->writable);
  if (vme->is_loaded)
  {
```

```
    lock_release(&lru_lock);
    return true;
  }
  else
  {
    free_page(kpage->kaddr);
    lock_release(&lru_lock);
    return false;
  }
}
```

- 원하는 virtual memory를 physical memory로 load하는 함수.
- 먼저, 원하는 virtual memory 엔트리를 가져온 후 해당 부분에 대해 physical memory를 할당한다.
- physical memory가 할당되면, 로딩하려는 virtual memory의 타입을 확인한다. VM_BIN 또는 VM_FILE인 경우, virtual memory에서 physical memory로 로딩이 아직 이뤄지지 않은 상태이므로 load_file() 함수를 사용하여 파일에서 로딩을 시도하고, 성공할 경우 install_page() 함수를 통해 physical memory 로딩을 진행한다. 그리고 virtual memory의 load 값을 변경한다.
- VM_ANON 즉, 스택 영역의 경우 현재 값은 physical memory에서 빠질 수 있지만 보관되어야 하므로 스왑 디스크에 저장되어 있다. 이 경우, swap_in() 함수를 사용하여 스왑 디스크에서 physical memory로 로딩해야 한다. 이 과정이 성공적으로 이루어지면 true를 return한다. 만약 실패시 다시 할당된 page를 모두 제거하고 free_page() 함수를 사용하여 할당해제를 해주고 false를 return 한다.

5. src/vm/page.c/bool load_file(void *kaddr, struct vm_entry *vme)

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

{
    int read_byte = file_read_at(vme->file, kaddr, vme->read_bytes, vme-
>offset);

    if (read_byte != (int)vme->read_bytes)
    {
        return false;
    }
    memset(kaddr + vme->read_bytes, 0, vme->zero_bytes);

    return true;
```

```
}
```

- 디스크의 파일을 physical memory로 load하는 함수. file_read() 함수를 사용하여 디스크의 데이터를 물리 page에 load하고, 남은 부분은 0으로 채운다. 이 과정을 통해 디스크에 있는 파일의 내용을 physical memory로 가져와 메모리에 load한다.

1-3. Supplemental Page Table

Algorithm

- Lazy loading 방식을 구현하기 위해서는 page fault가 발생할 때 virtual memory에 값이 있는지 확인하고, 필요한 경우 load해야 한다. 이를 위해 각 thread는 physical memory에 load해야 하는 page, 즉 virtual memory의 정보를 모두 가지고 있어야 한다.
- 기존pintos에는 모든 프로세스의 값이 physical memory에 load되는 방식이기 때문에 모든 값이 physical memory에 load되어 있어 virtual memory의 개념이 없어 virtual memory 정보를 저장하는 방법 자체가 존재하지 않았다. 이에 따라 virtual memory 엔트리의 정보를 저장하는 vm_entry라는 struct를 만들어 virtual memory 정보를 저장한다.
- 이러한 vm_entry는 각 thread에서 hash table 형태로 저장된다. 이를 supplementel page table이라고 한다. vm_entry를 해시로 관리하는 이유는 빠른 검색을 위해서이다.
- Pintos에서 제공하는 해시 interface를 활용하여 vm_entry를 hash table로 관리하는 함수들을 구현할 것이다. 그리고 virtual memory를 load하는 load_segment(), setup_stack() 함수들에서 각 thread가 virtual memory entry 정보를 저장할 수 있도록 구현한다.

Data Structure

1. **src/vm/frame.h/struct vm_entry**

```
struct vm_entry {

    uint8_t type;
    void *vaddr;
    bool writable;
    bool is_loaded;
    bool _pin;
```

```
    struct file* file;
    size_t offset;
    size_t read_bytes;
    size_t zero_bytes;

    struct hash_elem elem;
    struct list_elem mmap_elem;
    size_t swap_slot;
};
```

- 이 struct는 supplemental page entry를 나타내며, virtual memory와 관련된 정보를 담고 있다.
- type field는 엔트리의 종류를 나타내는데, VM_BIN, VM_FILE, VM_ANON 등의 타입을 지정한다.
- vaddr은 가상 page 번호를 나타내며, 해당 주소에 쓰기가 가능한지 여부를 나타내는 writable field와 physical memory로 load되었는지 여부를 나타내는 is_loaded flag와 page의 pinned여부를 나타내는 _pin flag가 있다.
- file은 mapping된 파일을 가리키며, offset은 파일에서 읽을 오프셋을, read_bytes는 가상 page에 쓰여진 데이터의 바이트 수를, zero_bytes는 0으로 채울 남은 page의 바이트 수를 나타낸다.
- 또한, hash_elem은 hash table의 요소를 나타내어 해당 supplementel page entry들은 각 thread마다 hash table을 통해 관리된다.

2. src/threads/thread.h/struct thread

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

- 각 thread마다 page을 관리하는 table을 hash type으로 추가한다.

Function Implementation

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

```
/* A thread function that loads a user process and starts it
   running. */
static void
start_process (void *file_name_)
{
…
  vm_init(&thread_current()->vm);
…
}
```

- process가 시작할 때, vm_init()를 호출하여 thread의 vm을 초기화한다.

2. **src/vm/page.c/void vm_init(struct hash *vm)**

```
void vm_init(struct hash *vm)
{
    hash_init(vm, vm_hash_func, vm_less_func, NULL);
}
```

- vm_init()함수는 hash_init()함수를 이용해 hash table을 초기화하는 함수이다.
- 기존 pintos에 구현되어 있는 hash function을 이용하여 hash table을 관리하는 함수들을 구현한다.

3. **src/vm/page.c/static unsigned vm_hash_func(const struct hash_elem *e, void *aux UNUSED)**

```
static unsigned
vm_hash_func(const struct hash_elem *e, void *aux UNUSED)
{
    struct vm_entry *vme = hash_entry(e, struct vm_entry, elem);
    return
    {
        hash_int((int)vme->vaddr);
    }
}
```

- element에 대한 hash key를 return하는 함수를 추가한다. vm_init()함수에 사용된다.

4. **src/vm/page.c/static bool vm_less_func(const struct hash_elem *a, const struct hash_elem *b, void *aux UNUSED)**

```
static bool

vm_less_func(const struct hash_elem *a, const struct hash_elem *b, void *aux
UNUSED)
{
    void *vaddr_a = hash_entry(a, struct vm_entry, elem)->vaddr;
    void *vaddr_b = hash_entry(b, struct vm_entry, elem)->vaddr;
    if(vaddr_a < vaddr_b)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

- argument로 받은 두 element에 대한 vm_entry의 vaddr값을 비교하는 함수. 첫 argument가 크면 true, 두번째 argument 가 크면 false를 return 한다. vm_init()함수에 사용된다.

5. **src/vm/page.c/bool insert_vme(struct hash *vm, struct vm_entry *vme)**

```
bool insert_vme(struct hash *vm, struct vm_entry *vme)

{
    if (!hash_insert(vm, &vme->elem))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

- hash table에 vm_entry를 추가하는 함수.

- hash_insert()함수를 이용해 hash table에 해당 원소를 추가한다. hash_insert()는 성공시 null을, 실패시(추가하려는 원소가 이미 hash table에 있는 경우) 추가하려는 원소를 return하므로 위와 같이 구현해 성공 시 true를, 실패 시 false를 return하도록 구현한다.

6. src/vm/page.c/bool delete_vme(struct hash *vm, struct vm_entry *vme)

```
bool delete_vme(struct hash *vm, struct vm_entry *vme)

{
    if (!hash_delete(vm, &vme->elem))
    {
        return false;
    }
    else
    {
        lock_acquire(&lru_lock);
        free_page(pagedir_get_page(thread_current()->pagedir, vme->vaddr));
        lock_release(&lru_lock);
        swap_free(vme->swap_slot);
        free(vme);
        return true;
    }
}
```

- hash table에서 해당 vm_entry를 삭제하는 함수.
- hash_delete 함수를 사용하며, 삭제 성공 시 frame table 관리를 위해 free_page를 호출하여 physical memory를 비우고 page를 할당 해제한다.

7. src/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)

```
/* Loads a segment starting at offset OFS in FILE at address
   UPAGE.  In total, READ_BYTES + ZERO_BYTES bytes of virtual
   memory are initialized, as follows:

        - READ_BYTES bytes at UPAGE must be read from FILE
          starting at offset OFS.

        - ZERO_BYTES bytes at UPAGE + READ_BYTES must be zeroed.
```

```
   The pages initialized by this function must be writable by the
   user process if WRITABLE is true, read-only otherwise.

   Return true if successful, false if a memory allocation error
   or disk read error occurs. */


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;

      struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct
vm_entry));
      if (!vme)
      {
        return false;
      }

      memset(vme, 0, sizeof(struct vm_entry));
      vme->type = VM_BIN;
      vme->vaddr = upage;
      vme->writable = writable;
      vme->is_loaded = false;
      vme->_pin=false;

      vme->file = file;
      vme->offset = ofs;
      vme->read_bytes = page_read_bytes;
      vme->zero_bytes = page_zero_bytes;


      insert_vme (&thread_current ()->vm, vme);

      /* Advance. */
```

```
      read_bytes -= page_read_bytes;
      zero_bytes -= page_zero_bytes;
      ofs += page_read_bytes;
      upage += PGSIZE;
    }
  return true;
}
```

- 위에서 설명했듯, load_segment에서 virtual memory를 load할 때 segment의 정보를 담은 vm_entry를 현재 thread의 hash table에 추가를 해주어야 한다.

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

```
/* Create a minimal stack by mapping a zeroed page at the top of
   user virtual memory. */
static bool
setup_stack (void **esp)
{
  lock_acquire(&lru_lock);
  struct page *kpage;

  kpage = alloc_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      if (install_page(((uint8_t *)PHYS_BASE) - PGSIZE, kpage->kaddr, true))
        *esp = PHYS_BASE;
      else
      {
        free_page(kpage->kaddr);
        lock_release(&lru_lock);
        return false;
      }
    }
    else
    {
      lock_release(&lru_lock);
      return false;
    }

    struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
    if (!vme)
    {
      lock_release(&lru_lock);
      return false;
```

```
    }

    memset(vme, 0, sizeof(struct vm_entry));
    vme->type = VM_ANON;
    vme->vaddr = ((uint8_t *)PHYS_BASE) - PGSIZE;
    vme->writable = true;
    vme->is_loaded = true;
    vme->_pin=true;

    vme->file = NULL;
    vme->offset = NULL;
    vme->read_bytes = 0;
    vme->zero_bytes = 0;

    kpage->vme =vme;

    lock_release(&lru_lock);
    return insert_vme(&thread_current()->vm, kpage->vme);
}
```

- 마찬가지로 위에서 말한 것처럼 segment의 정보를 담은 vm_entry를 만들고 install_page()함수를 통해 physical memory에 load한 후, 현재 thread의 hash_table에 vm_entry를 넣어준다.

1-4. Stack Growth

Algorithm

- 기존의 Pintos에서는 stack 공간이 4KB로 고정되어 있는데, 프로그램을 실행하다 보면 고정된 stack 영역보다 더 큰 영역이 필요할 수 있다. 이러한 상황에서 stack 영역을 확장해주는 알고리즘이 필요하다.
- 이를 위해 기존의 stack 공간을 초과하는 영역에 대한 접근 요청이 왔을 때 stack의 영역을 확장하도록 한다. 이 때 Pintos document에 근거하여 stack은 최대 8MB까지 확장할 수 있도록 하고, 8MB보다 더 큰 영역에 대한 접근을 요구하는 경우 segmentation fault를 발생시킨다.

Function Implementation

1. **src/userprog/process.c/expand_stack(void *addr)**

```
bool verify_stack(uint32_t addr, void *esp)
```

```
{
  uint32_t stack_start = 0xC0000000;
  uint32_t stack_limit = 0x8000000;

  if (!is_user_vaddr(addr))
    return false;
  if (addr < stack_start - stack_limit)
    return false;
  if (addr < esp - 32)
    return false;

  return true;
}
```

- stack을 확장하는 expand_stack 함수를 호출하기 전에 유효한 공간을 확장하려 하는지 검토하는 함수이다.
- Input argument로 들어오는 addr는 stack을 확장하여 stack 범위 내에 포함하려는 대상 주소이고, esp는 stack pointer이다. 이를 input argument로 받아 user의 virtual address가 아닐 경우, addr이 stack_start-stack_limit보다 작은 경우 false를 return한다.
- 추가적으로 인자로 받은 주소가 stack pointer-32보다 크거나 같은 경우에만 stack 영역을 확장해줘야 하므로 이에 대한 예외 처리를 진행하고 나머지 경우 true를 return하도록 구현한다.

2. src/userprog/process.c/expand_stack(void *addr)

```
bool expand_stack(void *addr)
{
  lock_acquire(&lru_lock);
  struct page *kpage;
  void *upage = pg_round_down(addr);

  kpage = alloc_page(PAL_USER | PAL_ZERO);
  if (kpage != NULL)
  {
    if (!install_page(upage, kpage->kaddr, true))
    {
      free_page(kpage->kaddr);
      lock_release(&lru_lock);
      return false;
    }
  }
  else
```

```
  {
    lock_release(&lru_lock);
    return false;
  }

    struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
    if (!vme)
    {
      lock_release(&lru_lock);
      return false;
    }

    memset(vme, 0, sizeof(struct vm_entry));
    vme->type = VM_ANON;
    vme->vaddr = upage;
    vme->writable = true;
    vme->is_loaded = true;
    vme->_pin=true;

    vme->file = NULL;
    vme->offset = NULL;
    vme->read_bytes = 0;
    vme->zero_bytes = 0;

    kpage->vme =vme;

    lock_release(&lru_lock);
    return insert_vme(&thread_current()->vm, kpage->vme);
}
```

-

3. **src/userprog/syscall.c/static void isAddressValid(void *addr, void *esp)**

```
static void isAddressValid(void *addr, void *esp)
{
  // if(addr >= (void *)0x08048000 && addr < (void *)0xc0000000)
  // {
  //   return true;
  // }
  // else
  // {
  //   return false;
  // }
  if(addr < (void *)0x08048000 || addr >= (void *)0xc0000000) sys_exit(-1);
  if(!find_vme(addr))
  {
```

```
    if(!verify_stack((int32_t) addr, esp)) sys_exit(-1);
    if(!expand_stack(addr)) sys_exit(-1);
  }
}
```

- input argument로 들어온 주소 변수 addr가 유효한 user 공간인지를 확인하고, 아니라면 system call을 통해 exit 처리한다.
- 그 뒤, 앞서 구현한 verify_stack과 expand_stack을 순서대로 호출하고, 실패할 경우 system call을 통해 exit 처리한다.

1-5. File Memory Mapping

Algorithm

- file memory mapping이란, disk block과 memory에 존재하는 page를 mapping하는 것을 말한다. 이를 통해 demand paging 방식으로 file을 읽게 되고 file read, file write가 일반적인 memory access로 간주된다.
- 이는 sys_read, sys_write과 같이 기존에 구현한 system call이 아니라 memory를 통해 file I/O를 처리할 수 있게 되고, file access를 더 단순화하여 빠르게 처리할 수 있다는 장점이 있다. 더불어 여러 process가 동일한 file을 mapping해 memory page를 공유할 수 있게 된다.
- 현재 Pintos에는 Project 2를 거치며 다양한 system call 함수들이 구현되어 있지만, file memory mapping과 연관되는 sys_mmap() 함수와 sys_munmap() 함수가 구현되어 있지 않아서 file memory mapping이 불가능하다. 즉, 현재로서는 file을 읽거 나 수정할 때 기존에 구현한 sys_read나 sys_write 함수를 호출하는 등의 방식으로 처리해야 한다.
  - File memory mapping을 가능하기 위해 별도의 자료 구조(memory mapped file)을 구현하고, memory map (sys_mmap)과 memory unmap (sys_munmap) 함수를 구현하는 것이 목적이다.
- 이를 위해 file memory mapping과 이를 통해 할당한 메모리의 unmapping을 지원하는 sys_mmap과 sys_munmap 함수를 구현하는 것이 목적이다.
- File mapping이 진행된 후 mapping된 page를 관리할 data structure을 만들어야 한다. 이를 위해 'struct mmap_file'을 새로이 구현한다.
- 마지막으로 process의 exit 시 file 수정 사항을 모두 disk에 저장할 수 있도록 앞서 구현한 sys_munmap 함수를 호출하고 exit하도록 수정한다.

Data Structure

- **src/vm/page.h**

  struct mmap_file

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

- Memory mapping된 file을 관리할 data structure이다. file마다 고유한 mapping id를 가져야 하기 때문에 이를 저장하기 위해 mapid 변수를 만들고, mapping하는 대상의 file object의 pointer, mmap_file을 list로 관리하기 위해 list element, mmap_file structure 내에서 vm_entry들의 list를 관리하는 변수인 vme_list가 포함된다.

- **src/thread/thread.h/struct thread**

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
 ...

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    …
    struct file *running_file;
    struct list mmap_list;
    int mmap_nxt;
#endif

...
  };
```

- Thread 내에서 memory 내에 load한 file을 저장하고 있어야 하므로 이를 mmap_list로

관리하기 위해 선언해주었고, memory mapping된 file의 개수를 저장하기 위해 mmap_nxt 변수를 만들어주었다.

Function Implementation

1. **src/userprog/syscal.c/sys_mmap(int fd, void *addr)**

```
mapid_t
sys_mmap(int fd, void *addr)
{
  if (pg_ofs (addr) != 0 || !addr) return -1;
  if (!is_user_vaddr (addr)) return -1;

  struct mmap_file *mfe;
  size_t ofs = 0;

  //mmap_file 생성 및 초기화
  mfe = (struct mmap_file *)malloc(sizeof(struct mmap_file));
  if (!mfe) return -1;

  memset (mfe, 0, sizeof(struct mmap_file));
  mfe->mapid = thread_current()->mmap_nxt++;
  lock_acquire(&lock_file);
  mfe->file = file_reopen(process_get_file(fd));
  lock_release(&lock_file);

  list_init(&mfe->vme_list);
  list_push_back(&thread_current()->mmap_list, &mfe->elem);

  //vm_entry 생성 및 초기화
  int file_len = file_length(mfe->file);
  while (file_len > 0)
    {
      if (find_vme (addr)) return -1;

      size_t page_read_bytes = file_len < PGSIZE ? file_len : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct
vm_entry));
      if (!vme)
      {
      return false;
      }

      memset(vme, 0, sizeof(struct vm_entry));
```

```
      vme->type = VM_FILE;
      vme->vaddr = addr;
      vme->writable = true;
      vme->is_loaded = false;
      vme->_pin=false;
      vme->file = mfe->file;
      vme->offset = ofs;
      vme->read_bytes = page_read_bytes;
      vme->zero_bytes = page_zero_bytes;

      list_push_back(&mfe->vme_list, &vme->mmap_elem);
      insert_vme(&thread_current()->vm, vme);
      addr += PGSIZE;
      ofs += PGSIZE;
      file_len -= PGSIZE;
    }

  return mfe->mapid;
}
```

- file memory mapping system call을 구현한 함수이다.
- 앞서 새롭게 구현했던 data structure인 mmap_file을 malloc을 통해 생성하고 초기화한다. 해당 entry의 mapping id는 앞서 thread에 추가해주었던 mmap_nxt를 대입해준다.
- Memory mapping을 할 때 열려있어야 하므로 file_reopen 함수를 호출하여 열어주고, file 길이만큼 virtual memory entry를 생성하고 초기화해준다.
- 이 때 해당 entry의 type은 VM_FILE로 설정한다.

2. src/vm/syscall.c/void sys_munmap(mapid_t mapid)

```
void sys_munmap(mapid_t mapid)
{
  struct mmap_file *mfe = NULL;
  struct list_elem *ele;
  for (ele = list_begin(&thread_current()->mmap_list); ele !=
list_end(&thread_current()->mmap_list); ele = list_next (ele))
  {
    mfe = list_entry (ele, struct mmap_file, elem);
    if (mfe->mapid == mapid) break;
  }

  if (!mfe) return;
```

```
  for (ele = list_begin(&mfe->vme_list); ele != list_end(&mfe->vme_list);)
  {
    struct vm_entry *vme = list_entry(ele, struct vm_entry, mmap_elem);
    if (vme->is_loaded && pagedir_is_dirty(thread_current()->pagedir, vme-
>vaddr))
    {
      lock_acquire(&lock_file);
      if (file_write_at(vme->file, vme->vaddr, vme->read_bytes, vme-
>offset) != (int)vme->read_bytes) NOT_REACHED();
      lock_release(&lock_file);

      lock_acquire(&lru_lock);
      free_page(pagedir_get_page(thread_current()->pagedir,vme->vaddr));
      lock_release(&lru_lock);
    }
    vme->is_loaded = false;
    ele = list_remove(ele);
    delete_vme(&thread_current()->vm, vme);
  }
  list_remove(&mfe->elem);
  free(mfe);
}
```

- 앞서 구현한 sys_mmap 함수를 통해 memory mapping한 file을 unmap하기 위한 system call 함수이다. 이를 위해 file mapping을 delete한다.
- Input argument로 mapping id를 넘겨받고, 이를 통해 해당 id를 갖고 있는 memory mapped file을 찾는다.
- current thread에 접근한 뒤, 앞서 새롭게 정의한 member variable인 mmap_list를 순회하면서, argument로 입력받은 mapid를 mapping id로 가지는 mmap_file을 찾는다.
- mmap_file entry을 찾은 뒤 해당 mmap_file entry의 vme_list를 순회하면서 모든 virtual memory entry에 대해 list_remove() 함수를 호출하여 vme_list에서 제거해주고, virtual memory entry를 제거하는 delete_vme 함수를 호출하여 thread_current()의 반환 값인 running thread의 virtual memory 상에서 해당 virtual memory entry를 제거해준다.
- for문으로 vme_list를 모두 순회해준 이후에는 mmap file 상에서 방금 unmap한 mmap file entry를 list_remove를 이용하여 지워준다. 마지막으로 해당 memory file entry에 대해 free해준다.

1-6. Swap Table

Algorithm

- 먼저, page swap을 위한 swap disk와 swap table을 구현한다
- 그 뒤 page swap을 해야 하는 상황이 발생했을 때 swap out할 page를 결정하는 policy를 선택해야 한다. LRU, Clock algorithm 등을 예시로 학습했는데, 구현 상 유리하고 동작이 직관적인 Clock algorithm을 사용하여 구현했다.
- 이 때 Page swap 과정은 synchronize하게 진행되어야 하기 때문에 관련 lock 변수를 만들어 활용해야 하고, 관련 lock 변수 swap_lock을 구현하여 활용했다.
- 또한 victim으로 선정된 page가 process의 data 영역 혹은 stack에 포함될 때 이를 swap 영역에 저장해야 한다. stack의 data를 저장할 수 있는 저장 공간과 이를 관리할 data structure을 추가해야 했고, 이를 각각 block과 bitmap으로 구현했다.
- 현재 Pintos의 swap partition은 4MB이며, 이를 4KB씩 나누어 관리한다. 각 partition들의 frame을 연결해야 하기 때문에 swap table은 list로 구현한다. 또한, 사용 가능한 swap partition을 관리하기 위해 상태를 나타내는 bit가 필요하고, swap table list의 entry로는 bitmap을 사용한다. 각 partition들이 free한, 즉 사용 가능한 상태라면 0으로 값을 설정한다. 해당 partition을 사용 중이라면 1로 값을 설정한다.

Data Structure

1. **src/vm/swap.h/struct lock lock_swap**

- swap 과정을 synchronize하게 수행하기 위한 lock structure이다.

2. **src/vm/swap.h/struct bitmap *bitmap_swap**

- 사용 가능한 swap partition을 관리하기 위해 각 partition을 0또는 1로 표현하고, 이를 위해 구현된 bitmap이다.

3. **src/vm/swap.h/struct block *block_swap**

- victim으로 선정된 page가 process의 data 영역, 또는 stack에 포함될 때 swap 영역에 저장하는데, 이 때 stack의 data를 저장할 수 있는 저장 공간이다.

Function Implementation

1. **src/vm/swap.c/void swap_init()**

```
void swap_init()
```

```
{
    lock_init(&lock_swap);
    bitmap_swap = bitmap_create(1024*8);
    if (!bitmap_swap) return;
    block_swap = block_get_role(BLOCK_SWAP);
    if (!block_swap) return;
}
```

- 위에서 구현한, swap과 관련된 data structure들을 초기화하는 함수이다.
- 기존에 구현된 lock_init 함수를 통해 lock_swap을 초기화하고, block의 크기만큼 bitmap을 create한다. 그 뒤 block을 swap을 위한 용도로 get한다.

2. src/vm/swap.c/void swap_in(size_t used_index, void* kaddr)

```
void swap_in(size_t used_index, void *kaddr)
{
    block_swap = block_get_role(BLOCK_SWAP);
    if (used_index-- == 0)
        NOT_REACHED();

    lock_acquire(&lock_swap);

    int i = 0;
    for (i = 0; i < 8; i++) block_read(block_swap, used_index * 8 + i, kaddr +
BLOCK_SECTOR_SIZE * i);

    bitmap_set_multiple(bitmap_swap, used_index, 1, false);

    lock_release(&lock_swap);
}
```

- swap in 동작을 구현한 함수이다. 먼저 block을 swap을 위한 용도로 get하고, synchronization을 위해 lock_swap을 acquire한다.
- 그 뒤 8개로 나눠진 swap partition을 모두 순회하면서 device로부터 데이터를 읽어 frame에 저장한다. 그 뒤 해당 block은 free하므로 bitmap을 false(0)으로 setting해준다.
- Swap in 과정이 끝났으므로 lock_swap을 release한다.

3. src/vm/swap.c/void swap_out(size_t used_index, void* kaddr)

```
size_t swap_out(void *kaddr)
{
    block_swap = block_get_role(BLOCK_SWAP);
    lock_acquire(&lock_swap);
    size_t index_swap = bitmap_scan_and_flip(bitmap_swap, 0, 1, false);

    if (BITMAP_ERROR == index_swap)
    {
        NOT_REACHED();
        return BITMAP_ERROR;
    }

    int i;
    for (i = 0; i < 8; i++) block_write(block_swap, index_swap * 8 + i, kaddr
+ BLOCK_SECTOR_SIZE * i);

    lock_release(&lock_swap);

    index_swap++;
    return index_swap;
}
```

- swap out 과정을 구현한 함수이다.
- block을 swap을 위한 용도로 get한 뒤 lock_swap을 acquire한다.
- 그 뒤 bitmap_scan_and_flip 함수를 호출하여 swap block 중 free한 곳을 찾고, 해당 영역의 bitmap을 1로 설정한다. (flip)
- 그 뒤 8개로 나눠진 swap partition을 순회하면서 block_write 함수를 호출하여 해당 block에 frame의 데이터를 저장한다.

4. src/vm/swap.c/swap_free(size_t used_index)

```
void swap_free(size_t used_index)
{
    if (used_index-- == 0) return;

    lock_acquire(&lock_swap);
    bitmap_set_multiple(bitmap_swap, used_index, 1, false);
    lock_release(&lock_swap);
}
```

- swapping에서 이용된 bitmap_swap structure을 0으로 초기화하는 함수이다.
- swap에 사용하였던 해당 disk sector의 bitmap을 0으로 초기화한다.

5. src/threads/init.c/int main (void)

```
int main (void) NO_RETURN;

/* Pintos main program. */
int
main (void)
{
  char **argv;

…
#endif
  swap_init(); // Project 3
  lru_list_init(); // Project 3

…

  printf ("Boot complete.\n");

  /* Run actions specified on kernel command line. */
  run_actions (argv);

  /* Finish up. */
  shutdown ();
  thread_exit ();
}
```

- Pintos system이 booting될 때 앞서 구현한 swap_init()과 lru_list_init() 함수를 호출하며 page swap을 사용할 준비를 마친다.

6. src/vm/page.c/static void vm_destroy_func(struct hash_elem *e, void *aux UNUSED)

```
tatic void
vm_destroy_func(struct hash_elem *e, void *aux UNUSED)
{
    struct vm_entry *vme = hash_entry(e, struct vm_entry, elem);
    if(vme->is_loaded)
    {
        lock_acquire(&lru_lock);
```

```
        free_page(pagedir_get_page(thread_current()->pagedir, vme->vaddr));
        lock_release(&lru_lock);
    }
    else
    {
        if(vme->_pin)
        {
            swap_free(vme->swap_slot);
        }
    }
    free(vme);
}
```

- vm_destroy 함수에서 hash_destroy 함수를 호출할 때 인자로 들어가는 함수이다.
- Hash_elem을 가지고 있는 virtual memory entry를 가져오고, 만약 is_loaded 변수가 true라면 lru_lock을 활용하여 synchronization이 보장되도록 free_page 함수를 호출하고, load되지 않았었다면 _pin 변수가 true일 때 해당 vme의 swap_slot을 free해준다.

1-7. On Process Termination

Algorithm

- Process가 종료될 때 지금까지 사용했던 자원을 반납하여 해당 자원들에 대한 누수를 방지해야 한다. 자원을 반납한다는 측면에서 고려해야 할 것은 다음 3가지이다.
- 1) supplemental page table의 제거
- 2) frame table entry의 제거
- 3) swap disk 공간의 제거
- 이를 위해 기존에 구현되어 있던 process.c/process_exit() 함수를 수정하는 방향으로 구현을 진행했다. Project 2의 구현까지는 process를 종료할 때 process가 사용중인 file_descriptor만 close를 해주면 되었기에 close_file 함수를 차례대로 호출하면 됐다.
- 하지만 지금까지의 구현을 통해 thread가 memory mapping을 이용해 mapping한 file memory가 있을 것이고, supplemental page table도 존재할 것이고, disk에 swap되어 있는 stack 값들도 존재할 것이므로 이 값들에 대한 할당 해제를 구현했다.
- 할당 해제 이외에 추가적으로 고려해야 할 것은, process가 종료될 때 file에 대한 수정사항을 모두 disk에 저장할 수 있도록 sys_munmap을 call해야 한다. 이를 위해 sys_munmap 함수에서도 dirty bit로 disk에 write해야하는지 여부를 체크하고 필요 시

disk에 write하도록 구현을 수정한다.

Function Implementation

1. **src/userprog/process.c/void process_exit()**

```
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  int i;
  for (i = 1; i < cur->mmap_nxt+1; i++) sys_munmap(i);


  for(i = 2; i < cur->fd_count; i++)
  {
     process_close_file(i);
  }


  file_close(cur->running_file);
  vm_destroy(&cur->vm);

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      cur->pagedir = NULL;
      pagedir_activate (NULL);
      pagedir_destroy (pd);
    }
}
```

- thread_current() 함수를 호출하고 반환값을 받아 현재 실행되고 있는 current의 pointer로 받아온다.
- 앞서 언급했듯 sys_munmap 함수를 현재 thread 내에 mapping되어있는 file들을 모두 순회하며 호출해주어야 한다. 그 뒤에는 모든 file descriptor을 순회하며 사전에 구현한 process_close_file 함수를 호출하여 file close를 수행한다. 마지막으로 current thread의 vm에 대해 vm_destroy 함수를 호출하여 virtual memory를 반납한다.
-

2. src/userprog/syscall.c/void sys_munmap(mapid_t mapid)

```
void sys_munmap(mapid_t mapid)
{
  struct mmap_file *mfe = NULL;
  struct list_elem *ele;
  for (ele = list_begin(&thread_current()->mmap_list); ele !=
list_end(&thread_current()->mmap_list); ele = list_next (ele))
  {
    mfe = list_entry (ele, struct mmap_file, elem);
    if (mfe->mapid == mapid) break;
  }

  if (!mfe) return;

  for (ele = list_begin(&mfe->vme_list); ele != list_end(&mfe->vme_list);)
  {
    struct vm_entry *vme = list_entry(ele, struct vm_entry, mmap_elem);
// newly implement in 7. On Process Termination (dirty bit check and write)
    if (vme->is_loaded && pagedir_is_dirty(thread_current()->pagedir, vme-
>vaddr))
    {
      lock_acquire(&lock_file);
      if (file_write_at(vme->file, vme->vaddr, vme->read_bytes, vme-
>offset) != (int)vme->read_bytes) NOT_REACHED();
      lock_release(&lock_file);
// newly implement in 7. On Process Termination (dirty bit check and write)

      lock_acquire(&lru_lock);
      free_page(pagedir_get_page(thread_current()->pagedir,vme->vaddr));
      lock_release(&lru_lock);
    }
    vme->is_loaded = false;
    ele = list_remove(ele);
```

```
    delete_vme(&thread_current()->vm, vme);
  }
  list_remove(&mfe->elem);
  free(mfe);
}
```

- 앞서 구현한 file system call sys_munmap 함수에 copy on write를 추가한다. pagedir.c에 구현되어 있는 pagedir_is_dirty()를 호출하여 dirty bit를 판단하고, dirty bit이 ture라면 free_page를 호출하기 전 disk에 write back하고, false라면 바로 해제할 수 있도록 구현했다.

2. Discussion

2-1. page-parallel test

- "tests/vm/page-parallel"는 Pintos의 virtual memory 시스템이 여러 프로세스의 메모리 요청을 동시에 처리하고, 각 프로세스의 virtual memory 공간을 올바르게 관리할 수 있는지 확인하는 테스트이다.
- 해당 test가 채점서버에서 통과가 되지 않아 해당 test를 통과하기 위해 많은 노력을 하였다. 해당 테스트를 통과하지 못하는 이유가 여러 process의 요청을 제대로 처리하지 못하기 때문이라고 생각했기 때문에, lru_list를 actomic하게 관리하기 위한 lru_lock을 유심히 보기로 하였다. 기존 코드에서는 lru_list에 page를 추가하거나 삭제하는 함수(alloc_page, free_page)에서 lock를 걸어 주었다.
- 하지만 이번에 구현한 부분에서 alloc_page함수를 통해 page를 할당하는 함수들은 조건에 맞지 않으면 다시 free_page함수를 통해 할당해제하는 과정이 포함되어 있다. page-parallel 테스트를 통과하기 위해서는 alloc_page, free_page각각에 lock을 거는것이 아니라 이 과정 전체를 atomic하게 관리되어야 한다고 생각했다. 그래서 기존 lru_lock을 걸어둔 부분을 다 삭제하고 alloc_page와 free_page가 모두 사용되는, 위와 같은 로직이 포함된 함수전체를 lock을 걸어줬다. 그리고 단순히 page를 할당해제시키는 로직만 있는, 즉 free_page만 사용하는 함수는 해당 함수에서 free_page가 사용되는 부분만 lock을 걸어줬다.
- 위와 같이 lru_lock에 관한 부분을 수정하니, 채점서버에서도 "tests/vm/page-parallel" 테스트를 통과할 수 있었다.

2-2. Pinning with page-merge test

- "tests/vm/page-merge" test는 page fault가 발생할 때, page table entry가 이미 존재하는 frame을 가리키고 있는지 확인하는 test이다. page fault가 발생하면, Pintos는 physical memory가 부족할 경우 swap 영역으로 page를 이동시키거나, 필요하지 않은 page를 제거함으로써 메모리를 확보한다.
- 그러나 해당 test에서는 여러 process가 동일한 page에 동시에 access하므로, 한 process가 page를 사용하는 동안 다른 process에 의해 해당 page가 변경되거나 제거되면 안된다. 따라서 위와 같이 동시에 memory에 access하는 여러 thread 간의 동기화 문제를 방지하기 위해 stack영역에 대해 page pinning 기능을 구현해야한다.

```
static bool
setup_stack (void **esp)
{
…
    struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
…
    vme->_pin=true;
…
}
```

```
bool expand_stack(void *addr)
{
…
    struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
…
    vme->_pin=true;
…
}
```

- 따라서 stack 영역을 다루는 setup_stack(), expand_stack()함수에서 page를 할당할때 page를 pinning해 주었다.

```
void* try_to_free_pages(enum palloc_flags flags)
{
    struct page *page;
    struct list_elem *element;
    while (true)
    {
        element = get_next_lru_clock();
        page = list_entry(element, struct page, lru_elem);
        if(page->vme->_pin)
        {
            continue;
        }
```

```
...
}
```

- page_fault()함수에서 physical memory로 load를 시도하는 handel_mm_fault()함수에서 evict할 page를 search할때 pinning되어 있지 않은 page를 찾는 로직을 추가해주었다.
- page pinning을 구현하니, 해당 test를 통과할 수 있었다.