Pintos Project: 3. Virtual Memory

Design Report – team 30

20210054 정하우

20210716 최대현

Context

3-7. On Process Termination

1. Introduce & Structure

1-1. Current Pintos system's load progress

- 현재 pintos는 program 을 execute할 때 필요한 모든 값들을 physical memory에 load한다.
- 코드를 통해 보면, pintos에서 process를 execute는 process_exec() -> load()-> load_segment() -> setup_stack() 함수 순으로 실행되어 physical memory에 load가 됨을 알 수 있다. 다음은 load_segment 함수이다.

```
/* 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;

      /* 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;
}
```

load_segment 함수를 보면 program 전체를 physical memory에 load를 함을 볼 수 있다. 이는 매우 비효율적이며, 이번 project를 통해 필요한 부분만 memory에 load하는 lazy loading을 구현할 것이다. 이를 위해서 모든 virtual memory를 관리해 physical memory에 load할 때 어떤 값이 올라가야 하는지 결정하는 memory management를 구현해야한다.

- 그리고 현재 pintos는 lazy loading이 구현되어 있지 않기 때문에 page_fault가 발생할 시 program을 바로 kill한다. 다음은 현재 pintos의 page_fault함수이다.

```
/* 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)//지금은 처리하지 않음
  {
    sys_exit(-1);
  }
  if (!user || is_kernel_vaddr(fault_addr)) {
    sys_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);
}
```

Lazy loading이 구현이 되면 physical memory에 필요한 값이 load가 되어 있지 않

더라도 바로 process를 kill하는 것이 아니라 virtual memory에 필요한 값이 있는지 확인하는 과정과, 값이 있을시 physical memory에 해당 값을 load하는 기능을 추가적으로 구현해야한다.

1-2. Page fault

- Process가 virtual address에 접근을 시도했지만, 실제로 이에 대응되는 page가 physical memory 상에 load되지 않은 상황을 의미한다. Page fault handler을 통해 해당 상황을 핸들링한다.
- 만약 physical memory 상에 page를 load할 공간이 충분하다면, disk에서 virtual address에 대응되는 page를 찾은 뒤 physical memory 상에 load하는 것으로 충분하겠지만, 만약 모든 physical memory가 다른 process에 의해 점유되고 있어서 load할 physical memory가 충분하지 않다면, 다른 process에 의해 점유되고 있는 physical memory의 일부를 다시 disk로 내린 뒤 해당 공간을 활용해야 할 것이다. 이를 위해서 어떤 page를 evict할 것인지에 대한 policy가 필요하다. 이를 page replacement algorithm 또는 page replacement policy라고 한다.
- 또한, algorithm을 적절히 구현하여 evict할 page를 선정했을 때, 이를 단순히 버리는 것이 아니라 데이터가 업데이트되어 disk와 내용이 다를 수 있다. 이 경우 disk에 업데이트된 내용을 기록해줘야 한다. 이를 detect하기 위해서 dirty bit을 두고 이 값을 설정함으로서 수정해야 하는 여부를 확인할 수 있다.
- 현재 userprog/exception.c의 page_fault() 함수가 구현되어 있고, Pintos system에서는 page fault가 발생하는 경우 해당 함수를 호출하여 page fault handler를 호출하고 처리해야 한다. 지금까지는 lazy loading이 구현되어 있지 않고 모든 program이 physical memory 상에 올라와 있고, page fault가 kernel이나 user program의 bug의 결과로만 나타났기 때문에 현재의 구현으로서는 page_fault() 함수를 호출할 경우 segmentation fault를 발생시키고 kill(-1)을 하여 강제 종료하도록 구현되어 있다. 하지만, 앞서 언급한 대로 lazy loading과 virtual memory를 구현하면서 paging이 필요한 상황들이 발생할 수 있기 때문에 이를 무조건적으로 강제 종료하는 것이 아니라 이를 virtual memory entry type에 맞게 처리하도록 수정하는 것이 필요하다.

2. Analysis of the current implementation

2-1. Frame Table

Problem Description

- 현재 Pintos는 모든 부분을 Physical Memory에 load한다. 이 방식은 매우 비효율적이므로 본 project에서 필요한 data만 Physical Memory에 load 하는 lazy loading을 구현할 것이다.
- Lazy loading을 구현하기 위해서는 Physical Memory가 가득 찬 상태에서 page를 load하는 경우 선택한 policy대로 (여기서는 Clock Algorithm, 추후에 Page Replacement Algorithm) 방식으로 기존에 Physical Memory에 load되어있는 page 중 한 page를 evict 하는 기능을 구현해야 한다.
- 위 기능을 구현하기 위해서는 list 형식으로 할당되어 있는 page의 정보를 가진 구조와 해당 list에 add, delete 하는 함수와, 이 함수들을 이용해 page를 할당 또는 할당해제 시 list에서 해당 page의 정보를 add, delete 할 수 있도록 page 할당/할당해제 함수를 추가적으로 구현하고, 기존에 page를 palloc_get_page, palloc_free_page 함수를 대체하면 frame table의 관리를 위한 구현이 완료된다.

관련 함수 설명

- Lazy loading을 구현할 때 Page replacement algorithm을 Clock Algorithm로 구현하기 위해서는 page directory가 최근에 접근했는지 확인하는 함수와, 최근에 접근했는지 여부를 set하는 함수를 사용해야한다.

- **1. src/userprog/pagedir.c/bool pagedir_is_accessed (uint32_t *pd, const void *vpage)**

```
/* Returns true if the PTE for virtual page VPAGE in PD has been
   accessed recently, that is, between the time the PTE was
   installed and the last time it was cleared.  Returns false if
   PD contains no PTE for VPAGE. */
bool
pagedir_is_accessed (uint32_t *pd, const void *vpage)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  return pte != NULL && (*pte & PTE_A) != 0;
}
```

- 주어진 virtual page의 page table entry이 최근에 access 되었는지 여부를 확인하는 함수. lookup_page함수를 통해 virtual page에 대한 table entry를 가리키는 포인터를 가져와 해당 table entry에서 PTE_A bit가 설정되어 있는지, 즉 해당 page가 access 된 적이

있는지 여부를 return 한다.

- **2. src/userprog/pagedir.c/ void pagedir_set_accessed (uint32_t *pd, const void *vpage, bool accessed)**

```
/* Sets the accessed bit to ACCESSED in the PTE for virtual page
   VPAGE in PD. */
void
pagedir_set_accessed (uint32_t *pd, const void *vpage, bool accessed)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  if (pte != NULL)
    {
      if (accessed)
        *pte |= PTE_A;
      else
        {
          *pte &= ~(uint32_t) PTE_A;
          invalidate_pagedir (pd);
        }
    }
}
```

- 주어진 Virtual Page의 Page table entry에서 accessed bit를 설정하는 함수. Access 가 true 인 경우 or 연산을 이용해 access bit를 설정하고 Accessed가 false인 경우 PTE_A 의 비트 반전을 이용해 access bit를 해제한다. 그리고 invalidate_pagedir함수를 이용해 TLB를 무효화하여 page table의 변경내용을 반영한다.

3. src/userprog/pagedir.c/ bool pagedir_is_dirty (uint32_t *pd, const void *vpage)

```
/* Returns true if the PTE for virtual page VPAGE in PD is dirty,
   that is, if the page has been modified since the PTE was
   installed.
   Returns false if PD contains no PTE for VPAGE. */
bool
pagedir_is_dirty (uint32_t *pd, const void *vpage)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  return pte != NULL && (*pte & PTE_D) != 0;
}
```

- 주어진 Virtual Page의 Page table entry가 가진 ditry bit를 return 하는 함수. 해당 항목이 존재하고 dirty bit가 설정되어 있으면 true를, 해당 항목이 없거나 dirty bit가 설정되지 않았으면 false를 return 한다.

2-2. Lazy Loading

Problem Description

- 위에서 설명했듯, 현재 pintos는 특정 process를 실행할 때 모든 값들을 physical memory에 load한다. 이는 매우 비효율적인 방식이며, physical memory보다 큰 값을 요구하는 process는 실행을 시킬 수 없는 단점이 존재한다. lazy loading 방식을 통해 필요한 값만 physical memory에 load하도록 구현하여 기존 memory의 용량보다 큰 memory를 요구하는 process도 실행시킬 수 있도록 구현할 것이다.
- 현재 pintos는 특정 page가 physical memory에 load되어 있지 않으면 page fault를 발생시킨다. 문제는 page fault가 발생한 경우 무조건 process를 kill한다는 것이다. Lazy loading 방식에서는 physical memory에 load되어 있지 않아도 virtual memory에 값이 존재하는 경우가 있다. 이렇게 page_fault 함수에서 처리가 가능한 경우를 고려하여 다시 구현해야 한다.

관련 함수 설명

- 기존 pintos는 process가 실행될 때 모든 값들이 다 physical memory로 load된다. 코드를 통해 보면, process_exec()함수를 통해 process가 실행되면 load()함수 안에 load_segment()함수가 실행돼 모든 값들이 다 physical memory에 load가 됨을 확인할 수 있다. 따라서 load_segment()함수를 실행했을 때 physical memory로 load하는 것이 아니라 virtual memory로 load하도록 새로 구현해 줘야한다.
- 그리고 page_fault 역시 process를 바로 kill하지 않고, 위에서 설명한 대로 virtual memory에 값이 있을 시에는 load를 시도하도록 구현을 변경해야 한다.

- **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;

      /* Get a page of memory. */
      /* physical memory 에 load 하는 부분. 이 부분을 virtual memroy 에
load 하도록 구현을 변경해야한다. */

      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;
}
```

- 메모리를 초기화하고, 해당 메모리를 프로세스의 주소 공간에 추가하는 함수. 위 introduce에서 설명했듯 현재는 physical memory에 load가 되고 있다. 위에서 설명한 대로, physical memory가 아니라 virtual memory에만 load를 하도록 구현해야 한다. 따라서 physical memory에 page를 할당하는 palloc_get_page 함수는 모두 제거를 해 주어야 한다.
- Load 된 virtual memory 정보는 thread에서 가지고 있어야 한다. 이는 3. Supplemental Page Table에서 자세히 다룰 것이다.

- **2. 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;
  //page_fault 시 바로 process를 kill한다.
  //이제 virtual memory에 값이 있는지 확인하고, 있을시에는 load를 시도하도록
구현해야한다.
  if(not_present)
  {
    sys_exit(-1);
  }
  if (!user || is_kernel_vaddr(fault_addr)) {
    sys_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);
}
```

- page_fault가 발생했을 때 호출되는 함수. 위 introduce에서 말했듯, virtual memory에 값이 존재하고, physical memory에 정상적으로 loading이 된다면 process가 계속해서 실행될 수 있도록 처리를 해주어야 한다. 이 말은 fault가 난 address가 user address인지 확인을 해 주어야 하며 user address일 경우 virtual memory에 해당 값이 존재하는지 확인을 해 주어야 한다. 그리고 후에 언급하고 구현할 handle_mm_fault 함수를 호

출하여 physical memory로 해당 값을 load해준다. Load를 성공할 시 code가 계속 진행되지만, load가 실행되지 않으면 exit(-1)으로 process를 kill해 준다.

- **3. 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)
{
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
        *esp = PHYS_BASE;
      else
        palloc_free_page (kpage);
    }

  return success;
}
```

- Process의 stack를 초기화 하는 함수. Data나 code를 읽는 load_segment 함수와는 달리 stack은 program을 실행하기 위해 반드시 필요한 요소이므로 lazy loading을 적용할 수 없다. 따라서 기존의 구현대로 physical address에 load를 해 주어야 한다.
- 또한, 이 경우 virtual memory를 관리하기 위해 할당된 physical memory가 어떤 virtual memory에 mapping되었는지를 저장하기 위해 supplemental page table에 해당 virtual memory의 정보가 담긴 entry를 추가해주어야 한다. 이 내용은 3. Supplemental page table에서 자세히 다룰 것이다.
- page를 할당할 때 palloc_get_page가 아니라 1. Frame table에서 구현한 alloc_page 함수를 이용해 page를 할당한다.
- 그 뒤 install_page 함수를 통해 physical memory와 virtual memory를 mapping하고, 성공 여부를 반환받은 뒤 성공했다면 esp 값을 PHYS_BASE로 setting한다.

- Virtual memory의 정보는 thread에서 가지고 있어야 한다. physical memory와 mapping 된 virtual memory를 해당 thread에 저장한다. 이는 3. Supplemental Page Table에서 자세히 다룰 것이다.

- **4. src/userprog/process.c/static bool install_page (void *upage, void *kpage, bool writable)**

```
/* Adds a mapping from user virtual address UPAGE to kernel
   virtual address KPAGE to the page table.
   If WRITABLE is true, the user process may modify the page;
   otherwise, it is read-only.
   UPAGE must not already be mapped.
   KPAGE should probably be a page obtained from the user pool
   with palloc_get_page().
   Returns true on success, false if UPAGE is already mapped or
   if memory allocation fails. */
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));
}
```

- Page table에 physical address 와 virtual address를 mapping하는 함수. Writeable를 받아 write가 가능한지에 대한 정보도 저장해준다.

2-3. Supplemental Page Table

Problem Description

- 위에서 말했듯 lazy loading 방식을 구현하기 위해서는 page fault가 발생하면 virtual memory에 값이 있는지 판단하고 load를 시도해야 한다. 이 말은 각 thread는 physical memory에 load해야 하는 page, 즉 virtual memory의 정보를 모두 가지고 있어야 함을 의미한다.
- 기존에는 모든 process의 값을 physical memory에 load하는 방식이기 때문에 모든

값들이 physical memory에 load되어있어 physical memory에 load해야 하는 page, 즉 virtual memory의 개념이 존재하지 않아 virtual memory의 정보를 저장하는 방식 자체가 존재하지 않는다. 따라서 우리는 virtual memory entry의 정보를 저장하는 struct인 vm_entry를 만들어 virtual memory의 정보를 저장할 것이다.

- virtual memory의 정보를 저장한 vm_entry는 해당 thread에서 hash table 형태로 저장한다. 이를 supplemental page table이라고 한다.
- vm_entry 를 hash로 관리하는 이유는 search가 빠르기 때문이다. pintos에서 제공하는 hash interface를 이용해 vm_entry를 hash table 형태로 구현한 supplemental page table 하에 관리할 것이다.

관련 함수 설명

- virtual memory의 정보를 저장하는 vm_entry struct를 추가적으로 구현해 준다.
- vm_entry를 hash table로 관리하기 위해 pintos에서 제공하는 hash interface 기능을 이용해야 한다.

1. src/thread/thread.h

```
struct thread

  {
    /* virtual memory entry 정보를 저장하는 hash table 추가 */
  };
```

- Thread struct에서 virtual memory entry 정보를 저장하는 hash table을 추가한다. virtual memory에 값이 있는지 찾기 위해 search 할때 시간을 단축하고자 Hash table을 이용한다. pintos에서 제공하는 hash table을 이용해 해당 값들을 다룰 것이다.

- **2. 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;

      /* Get a page of memory. */
      /* physical memory 에 load 하는 부분. 이 부분을 virtual memroy 에
load 하도록 구현을 변경해야한다. */
      /* supplemental page table 에 해당 virtual memory 를 추가한다*/


      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 설명한 load_segment 함수이다. Page를 allocation하고 install_page를 통해 physical memory와 virtual memory를 mapping 한다.
- 그리고 추가적으로 구현할 vm_entry struct를 만들어 load한 virtual memory의 정보를 넣고, vm_entry struct를 thread의 supplemental page table에 추가한다. load_segment함수에서는 physical function에 load 되지 않았으므로 vm_entry에는 load되지 않았다는 bit 정보를 저장해야 한다.

- **3. 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)
{
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
        *esp = PHYS_BASE;
      else
        palloc_free_page (kpage);
    }
/* load 된 physical memory 가 virtual memory 에 mapping 되는 기능 추가 */
```

```
  return success;
}
```

- 위에서 설명한 setup_stack 함수이다. 역시 install_page함수를 이용해 physical address에 load를 해주어야 하므로 load 된 physical memory가 virtual memory를 mapping 한다.
- 그리고 추가적으로 구현할 vm_entry struct를 만들어 load한 virtual memory의 정보를 넣고, vm_entry struct를 thread의 supplemental page table에 추가한다. setup_stack 함수에서는 physical function에 load되었으므로 vm_entry에는 load되었다는 bit 정보를 저장해야 한다.

- **4. src/lib/hash.c/bool hash_init (struct hash *h, hash_hash_func *hash, hash_less_func *less, void *aux)**

```
/* Initializes hash table H to compute hash values using HASH and
   compare hash elements using LESS, given auxiliary data AUX. */
bool
hash_init (struct hash *h,
           hash_hash_func *hash, hash_less_func *less, void *aux)
{
  h->elem_cnt = 0;
  h->bucket_cnt = 4;
  h->buckets = malloc (sizeof *h->buckets * h->bucket_cnt);
  h->hash = hash;
  h->less = less;
  h->aux = aux;

  if (h->buckets != NULL)
    {
      hash_clear (h, NULL);
      return true;
    }
  else
    return false;
}
```

- Hash table을 초기화 하는 함수. 초기화 하는 과정에서 필요한 memory를 할당하고 hash table에 필요한 정보들을 설정한다. 초기화 성공 시 true를, 실패 시 false를 return 한다.

- **5. src/lib/hash.c/void hash_destroy (struct hash *h, hash_action_func *destructor)**

```
/* Destroys hash table H.

   If DESTRUCTOR is non-null, then it is first called for each
   element in the hash.  DESTRUCTOR may, if appropriate,
   deallocate the memory used by the hash element.  However,
   modifying hash table H while hash_clear() is running, using
   any of the functions hash_clear(), hash_destroy(),
   hash_insert(), hash_replace(), or hash_delete(), yields
   undefined behavior, whether done in DESTRUCTOR or
   elsewhere. */
void
hash_destroy (struct hash *h, hash_action_func *destructor)
{
  if (destructor != NULL)
    hash_clear (h, destructor);
  free (h->buckets);
}
```

- Hash table을 제거하는 함수. Hash table에 있는 모든 요소들을 제거한 후 memory를 해제하여 자원을 반납한다.

- **6. src/lib/hash.c/ struct hash_elem * hash_insert (struct hash *h, struct hash_elem *new)**

```
/* Inserts NEW into hash table H and returns a null pointer, if
   no equal element is already in the table.
   If an equal element is already in the table, returns it
   without inserting NEW. */
struct hash_elem *
hash_insert (struct hash *h, struct hash_elem *new)
{
  struct list *bucket = find_bucket (h, new);
  struct hash_elem *old = find_elem (h, bucket, new);

  if (old == NULL)
    insert_elem (h, bucket, new);

  rehash (h);

  return old;
```

```
}
```

- Hash table에 새로운 요소를 추가하는 함수. Hash 구조에 따라 중복 삽입을 방지하여 데이터 일관성을 유지한다. 동일한 요소가 있는 경우 해당 요소를 return하고 아닌 경우 null을 return 한다.

- **7. src/lib/hash.c/ struct hash_elem * hash_delete (struct hash *h, struct hash_elem *e)**

```
/* Finds, removes, and returns an element equal to E in hash
   table H.  Returns a null pointer if no equal element existed
   in the table.

   If the elements of the hash table are dynamically allocated,
   or own resources that are, then it is the caller's
   responsibility to deallocate them. */
struct hash_elem *
hash_delete (struct hash *h, struct hash_elem *e)
{
  struct hash_elem *found = find_elem (h, find_bucket (h, e), e);
  if (found != NULL)
    {
      remove_elem (h, found);
      rehash (h);
    }
  return found;
}
```

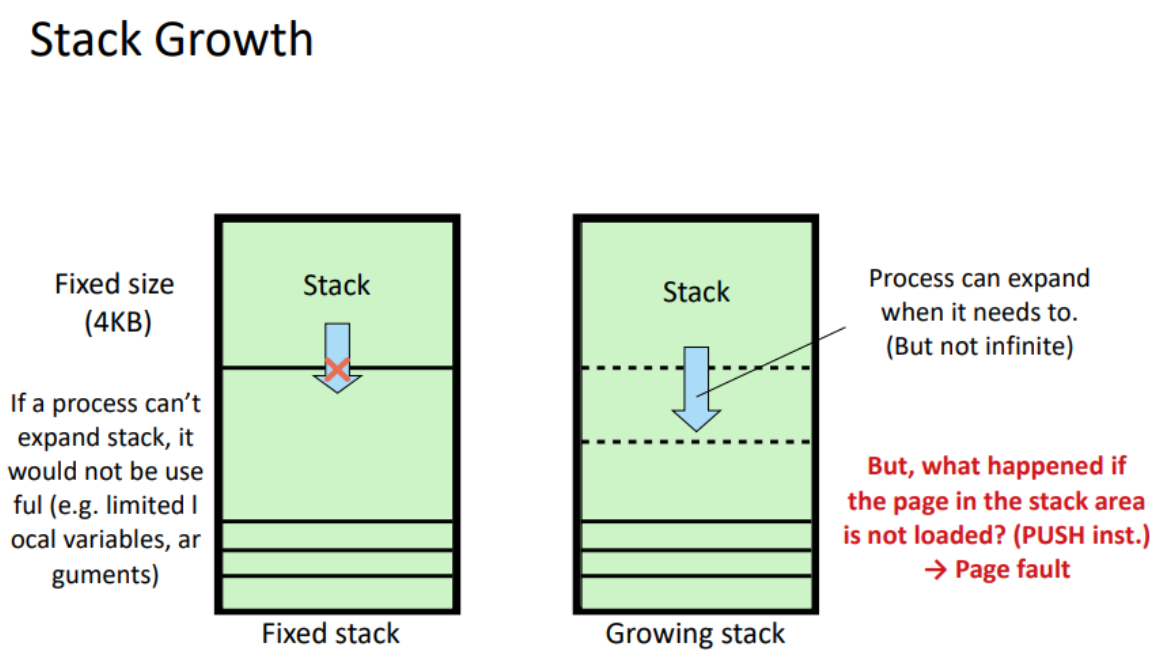
- 특정요소를 hash table에서 제거하는 함수. 성공적으로 제거하였으면 제거한 요소를 return하고 특정 요소가 존재하지 않으면 null을 return한다.

- **8. src/lib/hash.c/ struct hash_elem * hash_find (struct hash *h, struct hash_elem *e)**

```
/* Finds and returns an element equal to E in hash table H, or a
   null pointer if no equal element exists in the table. */
struct hash_elem *
hash_find (struct hash *h, struct hash_elem *e)
{
  return find_elem (h, find_bucket (h, e), e);
}
```

- 특정요소와 동일한 요소를 찾아 return 하는 함수. 특정요소가 없으면 null을 return한다.

2-4. Stack Growth



Supporting stack growth using page fault handler will be completed in the project

Problem Description

- 현재 Pintos 구현 상의 stack memory는 4KB 크기로 고정되어 있는 Fixed Stack이다. 하지만 프로그램을 실행하다 보면 더 큰 stack 영역을 필요로 할 때가 있다. P 이러한 상황에서 stack 영역을 확장하는 기능이 필요하다. 이 때 stack을 확장하기 위해서 stack이 확장하려는 영역이 유효한지에 대한 확인 과정도 필요하다.
- 현재 Pintos 구현 상으로는 stack의 영역을 4KB로 제한해두고 있고, 크기를 늘릴 수 없다. 또한 stack의 범위 외의 주소에 접근하면 segmentation fault를 발생하도록 구현되어 있다. 이를 stack을 늘릴 수 있는, 즉 유효한 stack 영역에 대한 접근까지는 허용하고 stack을 늘리는 방식으로 변경 구현이 필요하다.

2-5. File Memory Mapping

Problem Description

- 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 읽거나 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.

2-6. Swap Table

- Page swap을 위한 swap disk를 만들고, swap disk를 관리하기 위한 swap table과 policy를 구현하는 것을 목표로 한다.

  Page swap

- 컴퓨터의 Physical Memory는 한정된 용량을 가지고 있을 수 있지만, 프로그램이 실행되는 동안 필요한 memory 공간은 그 제한을 초과할 수 있다. 이러한 상황에서 OS는 virtual memory를 사용하여 memory 부족 문제를 해결하려 한다.
- 이 때, Physical Memory에 올라가 있는 process의 일부는 disk의 특정 영역으로 **swap out**되고, 다시 필요할 때 demand paging 과정을 통해서 해당 process의 일부를 disk에서 다시 **swap in** 한다. 이 과정을 page swap이라고 한다.
- **swap in:** page fault가 발생했을 때 swap disk에 존재하던 page를 memory 내의 새로운 frame으로 할당하는 것이다.
- **swap out:** memory에 존재하는 page 중 victim(disk로 넘어갈 page)를 선정한 뒤 이를 swap disk에 넣음으로서 free frame을 확보하는 것이다. 이를 구현하기 위해서 Swap out 과정에서 victim을 선정하는 데 policy가 필요하다.

- 현재 Pintos의 구현 상으로는 page swap 기능이 구현되어 있지 않다. 하지만 page table의 accessed bit은 page가 참조될 때마다 1로 setting되고 있다. 또한 stack의 정보가 적히는 공간인 swap partition을 swap space로 제공하며, 기본적으로 제공하는 크기는 4MB이다.

-

2-7. On Process Termination

- 앞서 구현했던 Process termination에 더해서, 지금까지 구현하면서 Process 상에서 자원들을 할당했고 (e.g. frame table) 이를 Process가 종료될 때는 모두 할당 해제함으로써 메모리 누수를 방지해야 한다.
- 자원을 할당 해제할 때 추가적으로 고려해야할 점들이 있다: Process가 terminate되기 전, 해당 process에 있는 page 중 dirty bit로 copy on write가 필요한 page를 판단하여 disk에 데이터를 업데이트한 뒤 terminate하도록 한다.
- 더해서, Process가 lock을 hold한 채로 종료될 시 deadlock이 발생할 수 있기에 보유중인 lock을 모두 release하고 마친다.

3. Design

3-1. Frame Table

Algorithm, Rationale

- Clock Algorithm을 page replacement policy로 구현하기 위해서는 page를 evict할 때 판단 근거가 되는 정보를 저장해야 하고, 이를 위해 전역변수로 lru_list를 구현해야 한다. 그리고 해당 list 정보를 update할 때 다른 process의 접근을 막기 위한 lru_lock와 다음으로 evict 될 page를 결정하는 정보를 담고 있는 page pointer 인 lru_clock도 같이 추가해준다.
- 할당된 page의 정보를 담는 list가 변경되는 경우는 page가 할당되거나 할당 해제됐을 경우이다. 위에서 말했듯, page를 할당하거나 할당 해제될 때 바로 list에 add 또는 delete 하는 기능을 가진 alloc_page 함수와 free_page 함수를 구현하고, 이 함수들이 palloc_get_page, palloc_free_page 함수를 대체하면 frame table 관리를 위한 구현이 완료된다.

Data structure of Clock Algorithm

vm/frame.h

- **다음과 같은 data structure들을 정의해준다.**

1. **struct list lru_clock_list:** clock algorithm을 구현하기 위해 필요한 list로, 현재 memory 상에 있는 page들을 모아놓은 list이다. 후에 언급될 get_next_lru_clock() 함수에서 해당 list를 순회하면서 swap out할 victim page를 결정하는 데 활용된다.
2. **struct list_elem *lru_clock:** clock_list를 iteration하기 위한 용도의 pointer이다.
3. **struct lock lru_lock:** lru_clock_list를 수정할 때 다른 process의 접근을 막기 위한 lock 이다.

새 함수 구현

- **void lru_list_init(void)**

  lru_list, lru_lock, lru_clock를 초기화 하는 함수.

- **void add_page_to_lru_list(struct page *page)**

  lru_list에 page를 추가하는 함수. lru_list 를 다루는 경우가 많기 때문에 해당 list에 page를 add하거나 delete하는 함수를 추가로 구현해 준다. list에 page를 추가하는 경우 lock를 걸어 다른 process가 접근하지 못하도록 구현해야 한다.

- **void del_page_from_lru_list(struct page *page)**

  lru_list에 해당 page를 삭제하는 함수. 위에서 언급했듯 lru_list 를 다루는 경우가 많기 때문에 해당 list에 page를 add하거나 delete하는 함수를 추가로 구현해 준다. list에 해당 page를 삭제하는 경우 lock를 걸어 다른 process가 접근하지 못하도록 구현해야 한다.

- **static struct list_elem *get_next_lru_clock()**

  lru_clock의 다음 값을 넣어주는 함수. lru_list를 순회할 때 사용된다.

- **void try_to_free_pages()**

  alloc_page 함수를 통해 page allocation시 physical memory 빈 공간이 없을 시 가장 적절한 page를 evict 한 후 빈공간이 생긴 physical memory에 새로운 page 를 allocation 한다. Lru 알고리즘을 통해 Page를 evict 하도록 구현해야 하므로, pagedir_is_accessed 함수를 통해 해당 page가 담겨있는 page dirctiory가 최근에

access 되었는지 판단한다. 만약 최근에 access되었다면 pagedir_set_accessed 함수를 이용하여 해당 page directory의 accessed bit를 0으로 설정하고 다음 page directory를 조사한다. 만약 최근에 access된 page directory라면 해당 page를 할당해제를 시켜줘야 한다. 할당해제를 하기 전에 해당 page의 type이 file인 경우 page_is_dirty함수를 통해 dirty 상태인지 파악하고, file type인 경우 overwrite 한 후 할당해제를 해 준다. Stack type인 경우 physical memory에 올릴 수 있는 가능성을 남겨두기 위해 swap disk를 구현해 정보를 저장하고, 해당 page는 할당 해제시켜준다.

- **struct frame *alloc_page(enum palloc_flags flags)**

  physical memory를 allocation해주는 함수. palloc_get_page 함수를 대신해 사용한다. palloc_get_page 함수를 통해 page를 할당시키는데, 이때 physical memory가 가득 차 있는지 확인하고, 가득차 있는 경우 try_to_free_pages함수를 통해 physical memory에 공간을 만든다. 단순히 page를 할당만 하지 않고 위에서 구현한 add_page_to_lru_list 함수를 통해 해당 page의 정보를 담은 page struct를 하나 생성하고 lru_list에 추가 해 준다.

- **void free_page(void *kaddr)**

  해당 physical address의 page를 할당해제하는 함수, 위 함수를 통해 page를 할당을 할 수 있게 되었으므로, 할당해제하는 함수도 따로 구현해야한다. 단순히 page를 할당해제만 하지 않고 위에서 구현한 del_page_from_lru_list 함수를 통해 page의 정보를 담고 있는 lru_list에 해당 page를 삭제하는 기능까지 구현하였다.

3-2. Lazy Loading

Algorithm, Rationale

- page fault시 virtual memory에 값이 존재하면 physical memory로 load를 시도해야 한다. 이를 위해 Virtual memory가 어떤 정보를 담고 있는지 확인해야한다. Virtual memory에 있는 값이 이미 physical memory에 존재한다면 load를 할 필요가 없다. Load가 되어 있지 않은 경우에는 physical memory에 load 를 하도록 추가적으로 구현을 해야 한다.

새 함수 구현

1. bool handle_mm_fault(struct vm_entry *vme)

- page fault시 virtual memory에 값이 존재하면 physical memory로 load를 시도하는 함수.
- 이를 위해 Virtual memory가 어떤 정보를 담고 있는지 확인해야한다. Virtual memory 에 있는 값이 이미 physical memory에 존재한다면 load를 할 필요가 없다.
- Load가 되어 있지 않은 경우에는 physical memory에 load 를 하도록 구현한다.
- page의 type이 VM_BIN이거나 VM_FILE인 경우 load_file함수를 통해 disk에 존재하는 file을 physical memory로 load한다. Page type이 VM_ANON일 경우는 disk가 아닌 swap 영역으로부터 정보를 가져와 physical memory로 load한다.
- Load 성공 시 install_page함수를 이용해 virtual address와 physical address를 mapping시켜준다.
- 위 모든 과정이 성공하면 true를 반환한다. 하지만 위 과정이 실패한 경우 위에서 구현한 free_page함수를 통해 page를 다시 할당해제를 해 준다.

2. bool load_file(void *kaddr, struct vm_entry *vme)

Disk에 존재하는 file을 physical memory에 load하는 함수. Physical memory에 load하고 남은 data는 0으로 채운다.

3-3. Supplemental Page Table

Algorithm, Rationale

- virtual memory의 정보를 담는 vm_entry struct를 구현해야 한다. 그리고 각 thread에서 vm_entry를 pintos에서 제공하는 hash table interface를 이용해 hash table로 관리할 것이다. 이는 virtual memory에 값이 존재하는지 search하는 시간을 줄이기 위함이다.

Data structure

- **struct vm_entry**

```
struct vm_entry{

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

```
  struct file* file;
  struct hash_elem elem;

  size_t offset;
  size_t read_bytes;
  size_t zero_bytes;
}
```

- virtual memory의 정보를 담는 struct.
  a. Type : VM_BIN(binary file로부터 data를 load), VM_FILE(mapping 된 file로부터 data를 load, VM_ANON(swap 영역으로부터 data를 load)의 type을 저장하는 value.
  b. Vaddr : virtual page number
  c. writeable : 해당 address에 write 가능 여부를 저장하는 value
  d. is_loaded : physical memory에 load 여부를 저장하는 value
  e. file : mapping 된 file 을 저장하는 value
  f. elem : hash table element
  g. offset : read할 file 의 offset를 저장하는 value
  h. read_bytes : virtual page에 write 되어있는 data bytes를 저장하는 value
  i. zero_bytes : 남은 page의 byte를 저장하는 value

새 함수 구현

1. **static unsigned vm_hash_func(const struct hash_elem *, void *UNUSED)**

해당 elem에 대한 hash key를 return 하는 함수. 기존 pintos에 구현되어있는 hash table interface를 이용해 구현한다. vm_init함수를 구현할 때 사용된다.

2. **static bool vm_less_func(const struct hash_elem *a, const struct hash_elem *b, void *aux UNUSED)**

두 elem의 vaddr을 비교하는 함수. 기존 pintos에 구현되어있는 hash table interface를 이용해 구현한다. vm_init함수를 구현할 때 사용된다.

3. **void vm_init(struct hash *vm)**

- 해당 hash table을 나타내는 vm을 init하는 함수. hash_init함수를 이용해 hash table을 초기화한다.

4. bool insert_vme(struct hash *vm, struct vm_entry *vme)

- supplemental page table에 vm_entry를 insert하는 함수. Insert 성공시 true, 실패시 false를 return 한다. hash_insert함수를 이용해 hash table에 요소를 추가한다.

5. bool delete_vme(struct hash *vm, struct vm_entry *vme)

- supplemental page table에 해당 vm_entry를 delete하는 함수. delete 성공시 true, 실패시 false를 return 한다. hash_delete 함수를 이용해 hash table에 해당 요소를 제거한다.

3-4. Stack Growth

Algorithm, Rationale

- 확장하기 전 stack의 (고정된) 용량인 4KB를 초과하는 영역에 대한 접근 요청이 왔을 때, 즉 현재 stack의 크기를 초과하는 주소에 접근이 발생했을 때 이것이 유효한 stack 접근이지, 혹은 segmentation fault인지 판별한 뒤 stack의 영역을 확장한다. 이때, stack이 최대로 확장가능한 크기는 pintos document에 제시되어 있듯이 **8MB**로 제한하도록 한다. 만약, 8MB보다 더 큰 영역에 대한 접근 요청이 들어오는 경우 segmentation fault가 발생하도록 exception handling한다.

기존 함수 수정

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

- verify_stack() 함수를 호출하여 address가 stack 영역에 포함되어 있는지 확인한 후 expand_stack()을 이용하여 stack을 확장한다.
- 현재 구현되어 있는 함수는 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)) {
    sys_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);
}
```

- 수정 사항은 다음과 같다.
- find_vme(fault_addr)를 호출하여 virtual memory entry를 찾는다.
- virtual memory entry가 찾아졌다면, fault_addr과 argument로 들어온 interrupt frame f의 esp로 접근하여 verify_stack() 함수를 호출함으로써 stack 내에 있는지 확인한다.
- verify_stack 함수의 return 값이 false인데 page fault가 발생했다는 것은 비정상적인 상황이므로 sys_exit(-1) 함수를 호출한다. 만약 true 값이 들어왔다면 stack을 확장할 수 있는 상황이므로 expand_stack(fault_addr) 함수를 호출한다.

- virtual memory entry를 찾지 못한 경우 또는 expand_stack 과정에서 error가 발생한 경우는 exit(-1)로 처리한다.
- 마지막으로 printf() 이후 부분들을 지워준다. 구현이 완료된 함수의 대략적인 흐름은 다음과 같다.

```
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);
  }
 // 수정할 부분

  struct vm_entry *vme = find_vme (fault_addr);
   if (!vme)
   {
      verify = verify_stack(fault_addr, f->esp));
     if(!verify) sys_exit(-1);
     expand = expand_stack(fault_addr);
      if(!expand) sys_exit(-1);
```

```
      return;
    }
    if (!handle_mm_fault(vme)) {
      sys_exit(-1);
    }
}
 // 이후 삭제
/*
  /* 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);
  */
}
```

새 함수 구현

2. userprog/process.c/bool verify_stack(int addr, void *esp)

- argument로 입력되는 address가 stack 공간 내에 포함되어 있는지 확인하는 함수이다.
- Argument로 입력되는 address가 esp-32보다 크거나 같은 경우에 true를 반환한다.

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

- argument로 들어오는 addr을 포함하도록 stack을 확장시키는 함수이다.
- alloc_page 함수를 호출하여 user 영역의 페이지를 할당하고, 페이지를 0으로 초기화한다. 그 뒤 페이지 할당이 성공했는지 확인하고, 성공하지 않았다면 함수를 종료하고 실패를 반환한다.
- install_page 함수를 호출하여 가상 주소(upage)와 물리 주소(kpage->kaddr)를 연결한다. 성공 여부는 success에 저장된다. install_page가 실패한 경우, 할당된 페이지를 해제하고 실패를 반환한다.
- 가상 메모리 엔트리(vm_entry)를 생성한다. 이 엔트리는 익명 페이지(VM_ANON)를 나타내며, 해당 페이지의 여러 속성을 지정한다. make_vme 함수가 실패한 경우 함수를 종료하고 실패를 반환한다. 생성된 가상 메모리 엔트리를 현재 스레드의 가상 메모리 테이블에 추가하고, 함수가 정상적으로 실행되었다면 성공을 반환한다.

3-5. File Memory Mapping

Algorithm, Rationale (Requirement)

- File memory mapping을 구현하기 위해 mmap_file이라는 새로운 data structure을 정의해준다.
- Memory mapping을 처리할 수 있는 system call이 현재 Pintos 시스템 상으로는 존재하지 않으므로, 이를 처리할 수 있는 아래 2가지 system call function을 구현한다.
- **sys_mmap():** File data를 memory에 load한다. (file mapping을 만든다.)
- **sys_munmap():** mmap으로 만들어진 file mapping을 delete한다.

Data structure of Memory Mapped files (Requirement)

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

- **userprog/vm/page.h/struct mmap_file:** memory mapping의 대상이 되는 file을 정의하기 위한 data structure이다. Mapid, file, vme_list, list_elem을 member로 갖는다.
  - **mapid_t mapid:** sys_mmap()이 성공적으로 수행될 시 return되는 mapping id를 저장할 member variable
  - **struct file* file:** mmap_file 대상의 file 객체
  - **struct list_elem elem:** mmap_file data structure을 관리할 list를 만들고, 각 mmap_file들을 list로 연결하기 위한 data structure
  - **struct list vme_list:** 이 data에 해당하는 모든 virtual memory entry를 담은 list.

새 함수 구현

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

- File data를 memory에 load하는 system call 함수이다. 이 때 load 과정은 lazy loading을 통해 수행된다. Process의 virtual address space에 mapping할 file descriptor와 memory mapping을 시작할 주소를 argument로 입력받는다.
- Address가 NULL 값이거나 page offset 값이 0이 아닌 경우, 그리고 user 영역의 address가 아닌 경우를 is_user_vaddr 함수와 pg_ofs 함수를 통해 예외처리한다.
- 먼저 앞서 만든 mmap_file의 크기만큼 malloc()을 통해 메모리를 할당하고, 이를 mmap file pointer 변수에 대입한다. 이를 실패할 경우 -1을 return한다. 그 뒤 memset 함수를 pointer 변수를 argument로 넣어 호출한다. 이 때 memory를 setting해줄 size는 mmap_file 구조체의 크기만큼, 값은 0으로 initialize해준다. 그 뒤 file descriptor로 파일을 가져온 뒤 file_reopen()함수를 호출해주고, 이것의 반환 값을 mmap_file pointer 변수가 가리키는 구조체의 'file' member 변수에 넣는다.
- 새롭게 만든 mmap_file structure의 member variable인 vme_list를 linitialize해주고, thread_current()에 접근한 뒤 mapid를 1 늘린다.
- 마지막으로, memory mapping을 한 file의 정보를 가진 page를 만든 후, hash table과 해당 mmap_file의 vme_list에 push_back 하면서 저장한다. 그리고 insert_끋 함수를 활용하여 thread_current()가 반환하는 running thread의 virtual memory에 virtual memory entry를 넣는다. 이 과정을 file_length로 initialize한 전체 파일 길이로부터 Page size씩 file length를 감소시키고, 해당 작업을 수행할 address를 page size씩 증가시키면서 file의 전체 내용을 mmap할 때까지 반복한다.

2. userprog/syscall.c/int sys_munmap(mapid_t mapid)

- mmap으로 만들어진 file mapping을 delete하는 함수이다.
- current thread의 새롭게 정의한 member variable인 mmap_list를 순회하면서, argument로 입력받은 mapid를 mapping id로 가지는 mmap_file을 찾는다. (for문을 순회하면서 현재 순회하고 있는 mmap file entry의 mapid가 찾는 mapid와 같다면 break하는 식으로 구현 가능하다.)
- 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.

- 추가적으로, 7.On Process Termination을 구현하기 위해 할당 해제하기 전 page directory가 dirty한지 check하여 dirty하다면 (즉, memory 상의 값이 수정되어 disk와 다른 경우) file.c/file_write_at 함수를 호출하여 disk에 접근하여 데이터를 수정해준 뒤 delete_vme 함수를 호출하여 running thread의 virtual memory 상에서 해당 virtual memory entry를 지워준다. (해당 부분은 3-7. On Process Termination 부분에도 언급되어 있다.)

3-6. Swap Table

Algorithm, Rationale

- 우선, 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 유리한 Clock algorithm 사용하여 구현하기로 결정했고, 그렇기 때문에 해당 policy를 구현해야 할 것이다. (해당 data structure은 1. Frame table에서 구현을 완료했다.)
- Page swap 과정은 synchronize하게 진행되어야 하기 때문에 관련 lock 변수를 만들어 활용해야 한다.
- 또한 victim으로 선정된 page가 process의 data 영역 혹은 stack에 포함될 때 이를 swap 영역에 저장해야 한다. stack의 data를 저장할 수 있는 swap block과 이를 관리할 swap bitmap을 추가해야 한다.
- 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.
- 추가적으로, 0인 partition을 찾는 알고리즘은 first-fit을 사용할 예정이다. (즉, 처음으로 0이 나오는 partition을 찾는 즉시 이를 victim으로 활용한다.)

새 함수 구현

Data structure of swap table

vm/swap.h

1. vm/swap.h (관련 data structure들 추가)

struct lock swap_lock

- swap 과정에 대한 synchronization을 위해 acquire, release하는 lock이다.

struct bitmap *swap_table

- swap block을 관리할 bitmap 형태의 data structure이다.

struct block *swap_block

- stack의 data를 저장할 수 있는 block 형태의 data structure이다.

2. vm/swap.c/void swap_init()

- swapping을 다룰 영역을 initialization하는 함수이다. Swap block을 block.c/block_get_role 함수를 호출하여 swap을 위한 용도로 가져오고, 해당 block의 크기만큼 bitmap을 생성한 뒤 모두 0으로 initialize한다.
- 그 뒤 swap_lock을 lock_init 함수를 통해 initialize해준다.

3. vm/swap.c/void swap_in()

- 먼저 swap 과정에 대한 synchronization을 위해 swap_lock을 acquire해준다.
- 정의되어 있는 enumerator 상수 BLOCK_SWAP를 argument로 block_get_role 함수를 호출하여 swap 용도의 block을 불러온다. 그 뒤 읽고자 하는 block이 free한 상태인지를 bitmap_test 함수를 통해 확인한다. 그 후 block_read 함수를 통해 device로부터 data를 읽어 frame에 저장한다. 그 후 읽어온 block은 free되어야 하므로 bitmap을 0으로 setting한다.
- 마지막으로 swap_lock을 release해준다.

4. vm/swap.c/size_t swap_out(void *kaddr)

- 먼저 swap 과정에 대한 synchronization을 위해 swap_lock을 acquire해준다.
- 정의되어 있는 enumerator 상수 BLOCK_SWAP를 argument로 block_get_role 함수를 호출하여 swap 용도의 block을 불러온다. 그 뒤 bitmap_scan_and_flip 함수를 통해 swap block 중 free한 곳을 찾는다. 또한, 해당 영역의 bitmap을 1로 설정한다. 그 후, block_write을 통해 해당 block에 frame의 데이터를 저장한다.
- 마지막으로 swap_lock을 release해준다.

5. **userprog/swap.c/static struct list_elem *get_next_clock()**

- swap out할 clock algorithm을 구현한 함수이다. 앞서 정의한 data structure인 clock_list를 순회하면서 reference bit 값이 0인 것을 찾으면 해당 clock을 return한다. 1이라면 list_next 함수를 호출하여 다음 clock을 체크하는 방식으로, 재귀적으로 구현한다.

6. **bool handle_mm_fault(struct vm_entry *vmentry)**

- 앞서 3. Supplemental Page Table에서 구현하였던 handle_mm_fault() 함수의 switch-case의 case: VM_ANON를 처리한다. 해당 type은 swap partition으로부터 data를 load하기 때문에, swap in을 해야 한다. 즉, 해당 case에 대해 swap in함수를 호출하도록 수정한다.

3-7. On process termination

Algorithm, Rationale

- 자원을 반납한다는 측면에서 고려해야 할 것은 다음 3가지이다. 1) supplemental page table의 제거, 2) frame table entry의 제거, 3) swap disk 공간의 제거를 고려해줘야 한다. 이를 위해 기존에 구현되어 있던 process.c/process_exit() 함수를 수정하는 방향으로 구현이 진행된다.
- 추가적으로 고려해야 할 것은, process가 종료될 때 file에 대한 수정 사항을 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.

기존 함수 수정

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

```
/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  int i;
  for(i = 2; i < cur->fd_count; i++)
```

```
  {
    sys_close(i);
  }

  palloc_free_page(cur->fd_table);
  file_close(cur->running_file);

  /* 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);
    }
}
```

- 수정되기 전 코드는 위와 같다.
- File_close를 하는 부분 직후에 sys_munmap(CLOSE_ALL)을 호출하여 모든 physical memory상에 mapping된 file들에 대해 free해준다.
- 후에 서술할 vm_destroy() 함수를 호출하여 남아 있는 virtual memory들에 대한 entry를 제거해준다. 이 때 argument는 앞선 구현에서 thread_current() 함수를 호출하여 이것을 thread pointer cur에 넣어주므로 이를 활용하여 thread structure의 supplement table에 접근 가능하다.

```
/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  int i;
```

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

  palloc_free_page(cur->fd_table);
  file_close(cur->running_file);

  // 수정된 부분
  sys_munmap(CLOSE_ALL);
  vm_destroy(&cur->sup_table);
  // 수정된 부분

  /* 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);
    }
}
```

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

- 앞서 언급했듯 virtual memory 상에서 page가 dirty하기 때문에 unmapping을 할 시 disk에 데이터를 덮어써야 하는 상황에 대한 처리를 추가적으로 구현해준다. 원하는 mapid를 가진 memory map file entry를 찾은 뒤, 해당 entry의 virtual memory entry list를 순하며 각 virtual memory entry에 접근할 때마다 pagedir_is_dirty 함수를 호출하여 해당 entry가 dirty한지 판단한다.
- dirty하다면 (즉, memory 상의 값이 수정되어 disk와 다른 경우) file.c/file_write_at 함수를 호출하여 disk에 접근하여 데이터를 수정해준다.
- 그 뒤 free_page() 함수와 pagedir_get_page() 함수를 호출하여 해당 entry 상에 올라와있는 page를 할당 해제해준다.

3. vm/page.c/vm_destroy (struct hash *vm)

- hash_destroy(vm, vm_destroy_func)를 호출한다. 즉, argument로 들어오는 virtual memory를 hash로 넘겨주고, 후에 서술할 vm_destroy_func을 hash action function으로 넘겨주어 hash destroy를 수행한다. 즉 해당 hash table의 bucketlist와 vm entry들을 제거한다.

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

- 이미 구현되어 있는 hash_entry 함수를 호출하여 virtual memory entry를 반환받고 이를 vm_entry pointer 변수 vme에 저장한다.
- 해당 virtual memory entry를 free한다.