CSED312 OS Lab 3 - Virtual Memory

Final Report

20180085 송수민 20180373 김현지

I. Frame Table

Brief Algorithm

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

Implementation

```
/* vm/page.h */
struct vm_entry{
  uint8_t type; /* VM_BIN, VM_FILE, VM_ANON의 타입 */
  void *vaddr; /* virtual page number */
  bool writable; /* 해당 주소에 write 가능 여부 */
  bool is_loaded; /* physical memory의 load 여부를 알려주는 flag */
  struct file* file; /* mapping된 파일 */
  struct hash_elem elem; /* hash table element */
  size_t offset; /* read 할 파일 offset */
  size_t read_bytes; /* virtual page에 쓰여져 있는 데이터 byte 수 */
  size_t zero_bytes; /* 0으로 채울 남은 페이지의 byte 수 */
}
```

> virtual memory의 frame table entry를 위와 같이 정의한다.

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

> 각 thread마다 virtual address space의 frame table을 hash type으로 추가한다.

```
/* vm/page.c */
void vm_init(struct hash *vm) /* hash table 초기화 */
{
  hash_init(vm, vm_hash_func, vm_less_func, NULL); /* hash_init()으로 hash table
초기화 */
}
```

> hash_init()를 사용하여 vm을 initialize하는 함수를 추가한다.

```
/* vm/page.c */
void vm_destroy(struct hash *vm) /* hash table 제거 */
{
  hash_destroy(vm, vm_destroy_func); /* hash_destroy()으로 hash table의 버킷리스트
와 vm_entry들을 제거 */
}
```

> hash_destroy()를 사용하여 vm을 destroy하는 함수를 추가한다.

```
/* vm/page.c */
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);
}
```

> vm_init 구현에 필요한, element에 대한 hash key를 반환하는 함수를 추가한다.

```
/* vm/page.c */
static bool vm_less_func(const struct hash_elem *a, const struct hash_elem *b,
void *aux UNUSED)
{
    return hash_entry(a, struct vm_entry, elem)->vaddr < hash_entry(b, struct
vm_entry, elem)->vaddr;
}
```

> vm_init 구현에 필요한, 두 element에 대한 vm_entry의 vaddr값을 비교하는 함수를 추가한다. (a가 작으면 true, 크면 false return)

```
/* vm/page.c */
static void
vm_destroy_func(struct hash_elem *e, void *aux UNUSED)
{
  struct vm_entry *vme = hash_entry(e, struct vm_entry, elem);
  free(vme);
}
```

> vm_destroy 구현에 필요한, element에 대한 vm_entry를 free시키는 함수를 추가한다. hash_entry를 통해 vm_entry를 찾고 이를 free시킨다.

```
/* vm/page.c */
bool insert_vme(struct hash *vm, struct vm_entry *vme) {
  if (!hash_insert(vm, &vme->elem)) return false;
  else return true;
}
```

> hash table에 vm_entry를 삽입하는 함수를 추가한다. hash_insert()를 이용하여 vm에 vm_entry element를 추가하고 그 성공여부를 return한다.

```
/* vm/page.c */
bool delete_vme(struct hash *vm, struct vm_entry *vme) {
  if (!hash_delete(vm, &vme->elem)) return false;
  else {
    free_page(pagedir_get_page(thread_current()->pagedir, vme->vaddr));
    swap_free(vme->swap_slot);
    free(vme);
    return true;
  }
}
```

> hash table에서 vm_entry를 삭제하는 함수를 추가한다. hash_delete()를 이용하여 vm에서 vm_entry element를 삭제하고 vm_entry를 free시킨 후 그 성공여부를 return한다.

```
/* vm/page.c */
struct vm_entry *find_vme (void *vaddr) {
  struct vm_entry vme;
  struct hash *vm = &thread_current()->vm;
  struct hash_elem *elem;
  vme.vaddr = pg_round_down(vaddr); // pg_round_down() 함수를 이용하여 vaddr의 page
number를 구한다.
  if ((elem = hash_find(vm, &vme.elem))) // hash_find() 함수를 이용해서 hash_elem을
찾고,
```

```
    return hash_entry(elem, struct vm_entry, elem); // 있으면 해당 elem에 대한
vm_entry를 hash_entry()를 이용하여 구한 후 return
  else return NULL; //elem이 없으면 NULL을 return한다.
}
```

> vaddr에 해당하는 hash_elem의 vm_entry를 찾아주는 함수를 추가한다. hash_find()를 이용하여 vm에서 virtual address에 해당하는 vm의 entry를 찾고 이를 return한다.

```
/* vm/page.c */
struct vm_entry *make_vme( uint8_t type, void *vaddr, bool writable, bool
is_loaded, struct file* file,
                           size_t offset, size_t read_bytes, size_t zero_bytes)
{
  /* vm_entry 생성 (malloc 사용) */
  struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
  if (!vme) return NULL;

  /* vm_entry 멤버들 설정, virtual page가 요구될 때 읽어야할 file의 offset과 size,
zerobyte 등등 */
  memset(vme, 0, sizeof(struct vm_entry));
  vme->type = type;
  vme->vaddr = vaddr;
  vme->writable = writable;
  vme->is_loaded = is_loaded;

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

  return vme;
}
```

> vm_entry를 생성하고, parameter로부터 넘겨받은 내용을 통해 vm_entry를 설정하는 함수를 추가한다.

```
/* userprog/process.c */
static void start_process (void *file_name_){
  ...
  vm_init(&thread_current()->vm);
  ...
}
```

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

```
/* userprog/process.c */
void process_exit (void) {
  ...
  vm_destroy(&thread_current()->vm);
  ...
}
```

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

II.Lazy Loading

Analysis

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

Brief Algorithm

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

Implementation

```
/* 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)
{
  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;

      /* vm_entry 생성 */
      struct vm_entry *vme = make_vme(VM_BIN, upage, writable, false, file, ofs,
page_read_bytes, page_zero_bytes);
      if(!vme) return false;

      /* insert_vme() 함수를 사용해서 생성한 vm_entry를 hash table에 추가 */
      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_segment는 segment를 process의 virtual address space에 load하는 함수이다. 원래는 load_segment가 실행되면 page를 할당하고, file을 page에 load하고 process's address space에 page를 저장하도록 구현되어있다. 하지만 이 부분은 삭제하고, 이를 vm을 이용하도록 바꾸어야한다. vm_entry를 할당하고 초기화하며, vm (hash table)에 insert하도록 수정하고, upage를 updage시켜준다.

```
/* userprog/process.c */
static bool setup_stack(void **esp)
{
  struct page *kpage;
  bool success = false;

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

  /* vm_entry 생성 */
  kpage->vme = make_vme(VM_ANON, ((uint8_t *)PHYS_BASE) - PGSIZE, true, true,
NULL, NULL, 0, 0);
  if (!kpage->vme)  return false;
  add_page_to_lru_list(kpage);
  /* insert_vme() 함수로 hash table에 추가 */
  insert_vme(&thread_current()->vm, kpage->vme);
```

```
  return success;
}
```

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

III. Supplemental Page Table

Brief Algorithm

현재 pintos는 Lazy loading, file memory mapping, swap table이 정상적으로 작동하도록 하는 함수 구현이 전혀 되어있지 않다. Lazy loading, file memory mapping, swap table이 모두 잘 작동할 수 있도록 구현이 수정되어야한다. pintos는 page fault 발생시 처리를 위해 page_fault()라는 함수가 존재한다. 원래는 무조건 “segmentation fault”를 발생시키고 kill(-1)을 하여 강제종료하도록 구현되어있다. 이를 vm_entry type에 맞게 처리되도록 수정한다.

Implementation

```
static void page_fault (struct intr_frame *f) {
   /* 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) exit(-1);
   struct vm_entry *vme = find_vme (fault_addr);
   if (!handle_mm_fault(vme)) {
      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);
}
```

> 원래는 무조건 “segmentation fault”를 발생시키고 kill(-1)을 하여 강제종료하도록 구현되어있는 부분을 삭제하고, find_vme를 이용하여 vm_entry를 찾은 후 해당 entry에 대해 type별로 page fault를 handle하

> 는 함수를 호출한다.

```
bool handle_mm_fault(struct vm_entry *vme)
{
  bool success = false;
  struct page *kpage;
  kpage = alloc_page(PAL_USER);
  kpage->vme = vme;

  switch (vme->type)
  {
  case VM_BIN:
    success = load_file(kpage->kaddr, vme);
    if (!success)
    {
      free_page(kpage->kaddr);
      return false;
    }
  case VM_FILE: break;
  case VM_ANON: break;
  default:
    NOT_REACHED ();
  }

  // install_page를 이용해서 physical page와 virtual page 맵핑
  if (!install_page(vme->vaddr, kpage->kaddr, vme->writable))
  {
    free_page(kpage->kaddr);
    return false;
  }

  // 로드 성공 여부 반환
  vme->is_loaded = true;
  add_page_to_lru_list(kpage);
  return true;
}
```

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

```
bool load_file(void *kaddr, struct vm_entry *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;
}
```

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

IV. Stack Growth

Brief Algorithm

Current Pintos System에는 1 page(4KB)로 fixed되어, Stack 확장이 불가능하게 구현되어있다. 따라서, Stack 확장이 가능하도록 구현하여야한다. stack pointer esp가 유효한 stack의 영역을 벗어나면 segmentation fault가 발생한다. 먼저, 현재 stack size를 초과하는 주소에 접근이 발생했을 때, page fault handler에서 stack확장이 필요한지 판별하게 한다. 필요한 경우로 최대 limit이내의 접근일 경우 valid한 stack 접근으로 간주하며, interupt frame에서 esp를 가져온다. 이때 필요한 page 수를 계산하는 과정이 필요하다. 또한 pintos document에 명시된 대로, 최대 8MB까지 stack을 확장할 수 있도록 수정한다.

Implementation

```
/* userprog/process.c */
bool expand_stack(void *addr)
{
  struct page *kpage;
  void *upage = pg_round_down(addr);
  bool success = false;

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

  /* vm_entry 생성 */
  kpage->vme = make_vme(VM_ANON, upage, true, true, NULL, NULL, 0, 0);
  if (!kpage->vme)  return false;
  add_page_to_lru_list(kpage);
  /* insert_vme() 함수로 hash table에 추가 */
  insert_vme(&thread_current()->vm, kpage->vme);
```

```
  return success;
}
```

> addr을 포함하도록 stack을 확장시키는 함수를 추가한다.

```
/* userprog/process.c */
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; //address < stack_start
  if (addr < stack_start - stack_limit)
    return false;
  if (addr < esp - 32)
    return false;

  return true;
}
```

> addr이 stack 범위에 포함되어 있는지 확인하는 함수를 추가한다.

```
/* userprog/syscall.c */
static void valid_address(void *addr, void *esp)
{
  if(addr < (void *)0x08048000 || addr >= (void *)0xc0000000) exit(-1);
  if(!find_vme(addr))
  {
    if(!verify_stack((int32_t) addr, esp)) exit(-1);
    if(!expand_stack(addr)) exit(-1);
  }
}
```

> address가 valid한지 확인하는 함수에서 verify stack()을 이용하여 address가 stack 영역에 포함되어 있는지 확인한 후, expand_stack()을 이용하여 stack을 확장하도록 수정하였다.

```
/* userprog/exception.c */
static void page_fault (struct intr_frame *f) {
  ...
   struct vm_entry *vme = find_vme (fault_addr);
   if (!vme)    /*Added*/
   {
      if (!verify_stack(fault_addr, f->esp)) exit(-1);
```

```
        if (!expand_stack(fault_addr)) exit(-1);
        return;
    }
   ...
}
```

> page_fault에서도 find_vme()로 찾은 vme가 NULL일 경우, verify stack()을 이용하여 address가 stack 영역에 포함되어 있는지 확인한 후, expand_stack()을 이용하여 stack을 확장하도록 수정하였다.

V. File Memory Mapping

Analysis

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

- mmap() : Lazy Loading에 의한 File Data를 memory에 load.
- munmap() : mmap으로 형성 된 file mapping을 delete.

추가로, file memory mapping을 완전히 구현하기 위해서 file을 VA로 관리한다. VA를 관리 하기 위해 Virtual Address entry를 이용하여 hash_table로 관리한다.

Implementation

```
/* vm/page.h */
struct mmap_file {
  mapid_t mapid;          //mmap() success시 mapping id를 return하는데, 이를 담을 variable.
  struct file* file;      //mapping하는 file의 file Object.
  struct list_elem elem; //이 data structure을 관리할 list가 필요하다. 이를 위해 리스트 연결을 할 구조체.
  struct list vme_list;  //이 data에 해당하는 모든 vm_entry의 list.
};
```

> mmap System call을 구현하기 위해 새로 선언한 data structure이다. Mapid는 identifier 역할을 하고, file은 mmap으로 load되어진 file pointer를 담는다. 이 mmap_file을 관리하는 list_elem을 추가하여 thread와 같은 관리 방법을 구현함으로써 일관된 구현을 유지하려 하였다. 추가로, 이 structure에 대항하는 vme를 파악하기 위해 list를 선언해주었다.

```
struct thread
{
  ...
```

```
  struct list mmap_list;  /*Added*/
  int mmap_nxt; /*Added*/
  struct file *file_run;           /*Added*/
  ..
};
struct vm_entry {
  ...
  struct list_elem mmap_elem; /* mmap_list element */ /*Added*/
  ...
};
```

> Thread에서 메모리에 load한 file을 파악할 필요가 있다. 따라서 list를 선언하여 이를 구현하였고, 몇개의 file이 mmap되어 있는지 파악하기 위해 mmap_nxt를 선언하였다. file_run은 file_reopen을 위한 file pointer이다.

```
mapid_t 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 = make_vme(VM_FILE, addr, true, false, mfe->file, ofs,
page_read_bytes, page_zero_bytes);
    if(!vme) return false;
    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;
}
```

> System call : mmap을 구현한 것이다. File descriptor와 vaddr를 parameter로 요구하며, 위에서 선언한 mmap_file을 할당하여 mmap_file의 member variable들을 초기화 해준다. mapid는 thread에 선언한 mmap_nxt를 이용하여 넣어주고, 이후 increment를 진행한다. File이 close되어도 mmap시에는 열려있어야 하므로 file_reopen을 해준다. 이후, file의 길이 만큼 vm_entry를 생성해준다. 이때, vme의 종류는 VM_FILE로 설정한다.

```
void 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);
    vme->is_loaded = false;
    ele = list_remove(ele);
    delete_vme(&thread_current()->vm, vme);
  }
  list_remove(&mfe->elem);
  free(mfe);
}
```

> 넘겨 받은 mapid를 통하여 해당하는 mmap_file을 찾아준다. 이때, 찾는 주체는 thread에 있는 mmap_list를 순회하면서 탐색한다. 해당 mmap_file을 찾으면 찾은 structure에 있는 vme_list를 순회하여 이 파일을 load한 vme를 찾는다. Vme에 load여부를 파악하는 boolean variable을 false로 바꾸주고, 해당 vme를 지워준다. 이후, mmap_file에서도 해당 vme를 지우고, mfe를 free하여 준다.

```
bool handle_mm_fault(struct vm_entry *vme)
{
  ...
```

```
  switch (vme->type)
  {
  case VM_BIN:
  case VM_FILE: /*Added*/
    success = load_file(kpage->kaddr, vme);
    if (!success)
    {
      free_page(kpage->kaddr);
      return false;
    }
    break;
  case VM_ANON: break;
  default:
    NOT_REACHED ();
  }
  ...
}
```

> Page fault가 발생하면 이 handler에서 physical page를 하나 할당하여 해당 주소와 page fault가 발생한 vme로 file을 load하여 memory에 올려주는 절차를 추가한다. 만약, load가 실패하였을 경우에는 미리 할당하였던 page를 해제하고 page_fault method에 false를 넘겨준다.

VI. Swap Table

Analysis

Frame을 새로 allocation 할 때 메모리가 부족하여 할당이 실패하는 경우가 존재한다. 이 경우에 Swapping이 필요한데, Swapping을 하기 위해 Swap disk와 Swap table이 필요하다. 원리는 간단하다. 메모리에 공간이 부족하다면 Swap table에 frame을 넣고 그 공간을 사용하는 것이다. 그렇다면 어떤 frame을 swap table에 넣어야 하는지는 Policy가 필요하다. Policy는 LRU Algorithm, Clock Algorithm 등이 있는데, LRU Algorithm은 Least Recently Used의 약자로 말 그대로 가장 오래전에 사용된 항목을 선택하는 것이고, Clock Algorithm은 frame을 순회하면서 어떠한 bit가 1이면 해당 frame을 swapping하는 것이다. 본 과제에서는 Clock Algorithm이 bit를 사용하여 구현하기 전에는 좀 더 구현이 용이할 것으로 보이기도 하고, 가시적이어서 해당 algorithm을 사용하려고 한다. 현재 Pintos의 swap partition은 4MB이며, 4KB로 나누어 관리를 한다. 이 Partition의 frame들을 연결해 줄 필요가 있기 때문에, Swap table은 list type으로 구현한다. 추가로, 사용가능한 swap partition을 관리하기 위해 bitmap을 사용한다. 사용가능한 상태, free상태라면 0으로 나타내고, 사용중이라면 1로 나타낸다. 0인 섹터를 찾는 알고리즘은 First-fit을 사용하고자 한다. Bitmap은 pintos에 내제되어 있는 것을 사용한다.

Implementation

```
/* vm/frame.h*/
struct list lru_list;
struct lock lru_lock;
struct list_elem *lru_clock;
```

> page replacement LRU policy 구현에 필요한 변수들을 추가한다.

```
/* vm/frame.c*/
void lru_list_init(void)
{
  list_init(&lru_list);
  lock_init(&lru_lock);
  lru_clock = NULL;
}
```

> 위에서 생성한 변수들을 초기화하는 함수이다. lru_list는 list_init(), lru_lock은 lock_init()을 통해 초기화하고, lru_clock은 NULL로 초기화한다.

```
void add_page_to_lru_list(struct page *page)
{
  lock_acquire(&lru_lock);
  list_push_back(&lru_list, &page->lru_elem);
  lock_release(&lru_lock);
}
```

> lru_list의 끝에 page를 add하는 함수를 추가한다.

```
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_list에서 page를 delete하는 함수를 추가한다. 이때 page가 lru_clock에 해당할 경우, 해당 elem을 삭제하고 lru_clock을 list의 다음 element로 설정해준다.

```
struct page *find_page_in_lru_list(void *kaddr)
{
  struct list_elem *ele;
  for (ele = list_begin(&lru_list); ele != list_end(&lru_list); ele =
list_next(ele))
  {
    struct page *page = list_entry(ele, struct page, lru_elem);
    if (page->kaddr == kaddr)
      return page;
  }
```

```
  return NULL;
}
```

> lru_list에서 physical address(kaddr)에 해당하는 page를 찾아 return하는 함수를 구현한다.

```
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;
}
```

> Swapping은 bitmap structure에 의해 관리된다. bitmap과 block은 이미 pintos에 구현되어 있는 것을 사용했다. 해당 변수와 lock은 전역 변수로 선언하였다. swap_init은 swapping을 하기 위해 사용 할 structure을 초기화하는데 쓰인다. 이는, init.c의 main에서 호출한다.

```
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;
  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);
}
```

> used_index의 swap slot에 저장된 data를 kaddr로 넣어준다. 이는 frame을 다시 메모리에 적재하는 역할을 할 것이다.

```
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);
  return index_swap + 1;
}
```

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

```
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 disk의 bitmap을 초기화는 method이다. Parameter로 vme에 선언되어 있는 swap_slot을 쓴다. Swap에 사용하였던 해당 disk sector의 bitmap을 0으로 초기화한다.

```
static struct list_elem *get_next_lru_clock()
{
  if (!lru_clock || lru_clock == list_end(&lru_list))
  {
    if (!list_empty(&lru_list)) return (lru_clock = list_begin(&lru_list));
    else return NULL;
  }
  else
  {
    lru_clock = list_next(lru_clock);
    if (lru_clock == list_end(&lru_list)) return get_next_lru_clock();
    else return lru_clock;
  }
}
```

> LRU Algorithm의 일종이 Clock alogrithm을 통해 clock 정보를 얻는 것이다. Clock algorithm 구현은 수업시간 강의노트 LECTURE 10에 나온 내용을 토대로 구현하였다. lru_clock이 0이면 lru_list를 확인하여

> lru_clock을 return 한다. 1이라면 다음 lru_clock을 찾아 recursive를 구현한다.

```
/* vm/page.h */
struct page {
  void *kaddr;
  struct vm_entry *vme;
  struct thread *thread;
  struct list_elem lru_elem;
};
```

> struct page는 physical page를 표현하는 structure이다.
>
> - kaddr : page의 physical address를 가리키는 포인터
> - vme : physical page가 mapping된 virtual address의 vm_entry 포인터
> - thread : page를 사용중인 thread의 포인터
> - elem : page replacement 구현에 필요한 lru list elem

```
struct page *alloc_page(enum palloc_flags flags)
{
  struct page *page;
  page = (struct page *)malloc(sizeof(struct page));  //page 구조체를 할당, 초기화
  if (!page) return NULL;

  memset(page, 0, sizeof(struct page));
  page->thread = thread_current();
  page->kaddr = palloc_get_page(flags);   // palloc_get_page()를 통해 페이지 할당
  while (!page->kaddr)
  {
    try_to_free_pages();
    page->kaddr = palloc_get_page(flags);
  }
  add_page_to_lru_list(page);
  return page;
}
```

> page를 할당하는 함수이다. 먼저 page 구조체를 할당 및 초기화하고, page의 정보를 설정한다. 이때, palloc_get_page()와 flag를 통해 page의 physical address를 할당하고, 할당이 되지 않았을 경우, 반복문을 통해 아래에서 후술할 try_to_free_pages()를 호출하면서 할당을 시도한다. 할당이 되었다면, 해당 page를 add_page_to_lru_list()를 이용하여 lru_list에 추가하고, return한다.
>
> 기존에 palloc_get_page()만으로 page allocation이 구현되어있었던 setup_stack(), expand_stack(), handle_mm_fault()에서 이를 alloc_page로 변경한다.

```
void free_page(void *kaddr)
{
  lock_acquire(&lru_lock);
  struct page *page = find_page_in_lru_list(kaddr);
  if (page != NULL)
  {
    pagedir_clear_page(page->thread->pagedir, page->vme->vaddr);
    del_page_from_lru_list(page);
    palloc_free_page(page->kaddr);
    free(page);
  }
  lock_release(&lru_lock);
}
```

> 할당된 page를 free시키는 함수이다. 먼저, find_page_in_lru_list()와 kaddr를 통해 page를 찾는다. page를 찾았다면, del_page_from_lru_list()를 이용하여 lru_list에서 제거하고, page 구조체에 할당받은 메모리 공간을 해제한다.
>
> free_page()도 마찬가지로 기존에 palloc_free_page()만으로 page free가 구현되어있던 부분을 free_page로 변경한다.

```
struct page *victim_page()
{
  struct list_elem *ele = get_next_lru_clock();
  struct page *page = list_entry(ele, struct page, elem);
  while (pagedir_is_accessed(page->thread->pagedir, page->vme->vaddr))
  {
    pagedir_set_accessed(page->thread->pagedir, page->vme->vaddr, false);
    ele = get_next_lru_clock();
    page = list_entry(ele, struct page, elem);
  }
  return page;
}
```

> clock Algorithm을 이용하여 accessed bit이 0인 page를 찾는다. while loop을 돌면서 pagedir_is_accessed()함수를 통해 page table의 accessed bit이 0인 것을 찾는다. 만약 1일 경우 이를 0으로 재설정해주고, 0일 경우 loop를 벗어나며 해당 page를 victim으로 선정하여 return한다.

```
void try_to_free_pages()
{
  lock_acquire(&lru_lock);

  struct page *page = victim_page();
  bool dirty = pagedir_is_dirty(page->thread->pagedir, page->vme->vaddr);
```

```
  if (page->vme->type == VM_FILE)
  {
    if(dirty) file_write_at(page->vme->file, page->kaddr, page->vme->read_bytes,
page->vme->offset);
  }
  else if (!(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);
  lock_release(&lru_lock);
}
```

> victim page를 선정된 page를 free시켜 여유 메모리 공간을 확보하는 함수이다. 먼저 victim page가 dirty인지 pagedir_is_dirty()를 통해 확인한다. victim의 방출은 page의 vm_entry type에 따라 구현이 달라지는데 VM_FILE일 경우에는 dirty한지 확인해보고, dirty한 경우 파일에 변경 내용을 저장한 후 page를 해제해야한다. VM_BIN일 경우에는 dirty한지 확인해보고, dirty한 경우에는 swap_out을 이용하여 swap partition에 기록후, type을 ANON으로 변경하고 page를 해제해야한다. VM_ANON일 경우에는 무조건 swap partition에 기록후 page를 해제하여야한다.

```
/* Pintos main program. */
int
main (void)
{
  ...
  swap_init();
  lru_list_init();
  ...
}
```

> pintos main program이 시작할때, swap_init과 lru_list_init을 호출하여 initialize해준다.

```
/* vm/page.c */
static void
vm_destroy_func(struct hash_elem *e, void *aux UNUSED)
{
  struct vm_entry *vme = hash_entry(e, struct vm_entry, elem);
  free_page(pagedir_get_page(thread_current()->pagedir, vme->vaddr)); /*Added*/
  swap_free(vme->swap_slot);                                          /*Added*/
  free(vme);
}
```

> vm hash variable을 삭제 할 때, mapping되어 있는 pagedir과 swapping에 사용한 swap_slot을 넘겨 swapping table에 bitmap 또한 초기화 해준다.

```
bool delete_vme(struct hash *vm, struct vm_entry *vme)
{
  if (!hash_delete(vm, &vme->elem))
    return false;
  else
  {
    free_page(pagedir_get_page(thread_current()->pagedir, vme->vaddr));
    swap_free(vme->swap_slot);
    free(vme);
    return true;
  }
}
```

> 직접 vme을 삭제하는 method에서도 mapping 되어 있는 pagedir을 free해주고, bitmap 또한 초기화 한다.

```
bool handle_mm_fault(struct vm_entry *vme)
{
  ...
  switch (vme->type)
  {
  case VM_BIN:
  case VM_FILE:
    success = load_file(kpage->kaddr, vme);
    if (!success)
    {
      free_page(kpage->kaddr);
      return false;
    }
    break;
  case VM_ANON:
    swap_in(vme->swap_slot, kpage->kaddr); /*Added*/
    break;
  default:
    NOT_REACHED ();
  }
  ...
}
```

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

VII. On Process Termination

Brief Algorithm

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

Implementation

```
void process_exit (void){
  ...
  for (i = 1; i < cur->mmap_nxt; i++)
    munmap(i);
  ...
}
```

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

```
void munmap(mapid_t mapid)
{
  ...
  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))
/*Added*/
    {
      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);

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

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

Discussion

1. Lazy Loading

> Lazy Loading을 구현하고 test를 진행하였는데, 한 개의 test도 통과하지 못하였었다. 디버깅을 한 결과, file이 load 되기 전에 어디선가 file이 close된다는 점을 알아냈다. file_close()는 여러곳에서 사용되고 있어, 어느 것이 잘못 되었는지, 어떠한 연쇄적으로 이루어지는 일로 발생하는 것은 아닌지로 인해 파악하는 것이 어려웠다. 다행히, load 되기 전에 file이 close된다는 점을 알아내 가능성이 가장 높은 load 말미에 있던 file_close를 없애보았다.

```
/* userprog/process.c */
bool
load (const char *file_name, void (**eip) (void), void **esp){
  ...
  /* Set up stack. */
  if (!setup_stack (esp))
    goto done;

  /* Start address. */
  *eip = (void (*) (void)) ehdr.e_entry;

  success = true;

 done:
  /* We arrive here whether the load is successful or not. */
  // file_close (file); /* Deleted */
  return success;
}
```

> 위와 같이 마지막에 file_close (file) 부분을 삭제하니, lazy loading이 이루어지는 것을 확인하였다.

2. Sync-read & write Test

> Project2에서 잘 통과되던 test가 Project 3에서 돌아가지 않아, 이전 과제에서 구현하였던 부분 중에 잘못 구현한 부분이 있는지를 확인해보았다.

```
tid_t
process_execute (const char *file_name)
{
  ...
  /* Make a copy of FILE_NAME.
    Otherwise there's a race between the caller and load(). */
```

```
  fn_copy_1 = palloc_get_page(PAL_ZERO);
  fn_copy_2 = palloc_get_page(PAL_ZERO);
  if (fn_copy_1 == NULL)
    return TID_ERROR;
  strlcpy(fn_copy_1,file_name,PGSIZE);
  strlcpy(fn_copy_2,file_name,PGSIZE);

  name = strtok_r(fn_copy_2," ",&remain);.
  // 아래의 부분을 제거
  // if (filesys_open(name) == NULL) {
  //   return -1;
  // }

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (name, PRI_DEFAULT, start_process, fn_copy_1);
  ...
```

> Project 2에서는 어떠한 close 과정이 있어 잘 닫히고 처리되었던 filesys_open이 virtual memory & lazy loading으로 인해 mechanism이 변화가 있었고, 해당 부분의 open을 close해주지 못한 것 같다. 따라서, 위 부분을 지웠더니, read / write에 관한 sync 문제가 해결되었다.

3. Page test

> tests/vm/page-linear, parallel, merge-seq, merge-par, merge-stk, merge-mm의 테스트는 이번 과제에서 통과하기 위해 가장 긴 시간을 투자한 부분이다. 위에서 기술한 구현사항을 모두 완료하였음에도 6개의 test가 번갈아가며 pass가 되지않는 모습을 보였다. 대부분 "run: open buf"의 에러가 발생하며, file자체를 열지 못하고 실패하였는데 Debugging하여도 그 원인을 쉽게 파악할 수 없었다. 이 부분 또한 file open과 관련된 부분이기때문에 project2의 구현이 완벽하지 못하다 판단하여, 조교님께서 올려주신 code를 통해 어떤 부분이 부족하였는지 비교하였다. 그러던 중 syscall.c에 구현되어있는 open, read와 write함수를 제외한 file관련 함수들에 lock으로 file을 관리하지 않는것이 다르다는 점을 깨달았다.

```
bool create(const char *file, unsigned initial_size)
{
  if (!file) exit(-1);
  lock_acquire(&lock_file);   /*Added*/
  bool success = filesys_create(file, initial_size);
  lock_release(&lock_file);   /*Added*/
  return success;
}
```

> 그래서 위와 같이 file system과 관련된 함수에 lock을 걸어주었더니, file을 안전하게 open하였고 test에 통과할 수 있었다.

4. What We have Learned

> 위의 1,2,3 문제들에서 특히 어려움을 겪었는데 그 이유를 다음과 같이 생각한다. Project 3는 Project 2와 다르게 이전 Project를 기반으로 하여 구현을 진행하여야 한다. 이는 Project 2에서 어떠한 구현법을 채택하였느냐에 따라 Project 3에 영향을 끼칠 수 있다. 위의 문제들은 File에 관한 문제라는 것이 공통점이다.

> 구현을 마무리 할 때쯤, file synchronization을 구현하였던 방법이 기억났는데, 그때 당시 원래 Copy-on-write를 구현하고자 하였으나, 구현이 어려워 한 프로세스가 file을 열어 점유하고 있으면 해당 file은 접근을 못하게 구현하였다. 본 조의 생각으로는, 이렇게 구현하면 read / write는 의도한대로 구현이 이루어지고, open은 영향을 미치지 않을 것이라 생각하였다. 하지만, code를 구현하다보니 추가로 file을 열어야 할 경우가 발생하거나, 여러 child가 거의 동시에 file을 여는 test들이 존재하였다. OS 수업 복습을 하면서 Copy-on-write의 중요성을 파악하게 되었고, Project 2에서 구현의 어려움으로 인해 구현을 미숙하게 완료했던 부분이 Project 3에서 발목을 잡았다는 생각이 들었다. 실제로 구현은 대공사가 될 것 같아 해보지는 않았지만, child가 open을 여러번 하는 과정에서는 copy-on-write가 유용하게 이루어졌을 것이라는 점을 예상해보았다.

Result

서버에서 make check를 입력하여 나온 모든 test에 대한 결과값이다.

```
pass tests/vm/mmap-clean
pass tests/vm/mmap-inherit
pass tests/vm/mmap-misalign
pass tests/vm/mmap-null
pass tests/vm/mmap-over-code
pass tests/vm/mmap-over-data
pass tests/vm/mmap-over-stk
pass tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 113 tests passed.
```

위와 같이 이번 PintOS Project 3의 모든 test를 pass한 모습을 볼 수 있다.