운영 체제 과제 보고서

project #3

과목명: 운영 체제

담당 교수: 원유집 교수님

담당 조교: 오준택 조교님

제출 일자: 2018/ 06/ 01

2014004411 김시완

2016025714 이동윤

Github address:

https://github.com/gimsiwan/Operating_system.git

목차

1. Virtual Memory

과제 목표:

Swap 사용과 Demand paging 사용을 위해 Virtual Memory 구현

과제 설명:

실행 파일이 탑재될 때 전체 프로세스 주소 공간 할당이 아닌, 가상 주소의 각 페이지에 vm_entry 를 할당한다. 페이지 폴트 발생시, 가상 주소에 해당하는 vm_entry 를 탐색한 다음 물리 페이지를 할당.

a. thread 구조체/ vm_entry 구조체

```
    struct file **fdt;            /* pointer of file descripter table */
    int next_fd;                  /* number of next empty space in file descripter table */
    //this extra value on structure is needed on project5
    struct file *executing_file;      // executing file on the thread

    struct hash vm;    //VM_ENTRY 저장한 해시테이블

    struct list mmap_list;    //해당 스레드가 관리하는 MMAP_FILE LIST
    int mapid;    //MMAP 성공시 MAPPID++(총 MMAP된 파일 개수)
};
```

Thread 구조체에 vm_entry 들을 관리해 줄 해시 테이블 vm 을 추가 시켜준다.

```
struct vm_entry
{
    uint8_t type; /* VM_BIN, VM_FILE, VM_ANON의타입 */
    void * vaddr; /* vm_entry의가상페이지번호 */
    bool writable; /* True일경우해당주소에write 가능
                      False일경우해당주소에write 불가능 */
    bool is_loaded; /* 물리메모리의탑재여부를알려주는플래그 */
    struct file * file; /* 가상주소와맵핑된파일 */

    /* Memory Mapped File 에서다룰예정 */
    struct list_elem mmap_elem; /* mmap리스트element */

    size_t offset; /* 읽어야할파일오프셋 */
    size_t read_bytes; /* 가상페이지에쓰여져있는데이터크기 */
    size_t zero_bytes; /* 0으로채울남은페이지의바이트 */

    /* Swapping 과제에서다룰예정 */
    size_t swap_slot; /* 스왑슬롯 */

    /* 'vm_entry들을위한자료구조' 부분에서다룰예정 */
    struct hash_elem elem; /* 해시테이블Element */
};
```

각 페이지마다 할당 시켜줄 vm_entry 구조체를 생성한다. vm_entry 는 물리 주소 mapping 에 필요한 정보를 가지고 있으며, 가상 주소와 1 대 1 로 매칭된다.

Vm_Entry 의 type 으로는

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

로 세가지가 있으며

VM_FILE 은 MMAP 에서, VM_ANON 은 SWAP 에서 사용된다.

b. vm_init 함수 / vm_hash_func 함수 / vm_less_func 함수

```
void vm_init(struct hash * vm){    //vm_entry를 저장하는 해시 테이블 초기화
    hash_init(vm, &vm_hash_func, &vm_less_func, NULL);
}

static unsigned vm_hash_func (const struct hash_elem *e, void * aux UNUSED){
    //해시테이블에 vm_entry 삽입 시 어느 위치에 넣을지 계산
    struct vm_entry *vme = hash_entry(e, struct vm_entry, elem);

    return hash_int((int)vme->vaddr);
}

static bool vm_less_func(const struct hash_elem *a, const struct hash_elem *b, void * aux UNUSED){
    //hash_elem의 주소 값 비교시 사용
    struct vm_entry *va = hash_entry(a, struct vm_entry, elem);
    struct vm_entry *vb = hash_entry(b, struct vm_entry, elem);

    return (va->vaddr) < (vb->vaddr);
}
```

vm_init 함수는 vm_entry 들을 관리하기 위한 해시 테이블을 초기화 해주는 함수로서, hash_init 함수를 이용한다. 두 번째 인자로는 해시 테이블에 자료를 저장하는 위치를 결정할 함수를 넣어주며, 세 번째 인자로는 해시 테이블에 저장되는 자료의 비교를 위한 함수를 넣어준다.

```
  for(token=strtok_r(file_name," ", &save_ptr); token!=NULL; token=strtok_r(NULL," ",&save_ptr)){
    parse[count] = palloc_get_page(0);
    strlcpy(parse[count],token,strlen(token)+1);
    count += 1;
  }
  // parsing the file_name with " " delimiter and store it on the variable named 'parse'
  // parse[0] is the real file_name, and from parse[1] is the argument of the function

  vm_init(&thread_current()->vm);

  memset (&if_, 0, sizeof if_);
```

위의 코드와 같이 start_process 에서 해당 함수를 호출함으로써 스레드가 실행될 때 vm 해시 테이블을 초기화 시켜준다.

vm_hash_func 함수는 vm_entry 가 해시 테이블에서 어느 위치에 저장될 것인지 결정해주는 함수로써 hash_int 함수를 사용하여 만들어 주었다.

vm_less_func 는 해시 테이블 요소의 크기 비교를 위해 사용되는 함수로서, 여기서는 vm_entry 의 주소 값을 비교하는데 사용되었다.

c. Insert_vme 함수 / delete_vme 함수

```
bool insert_vme(struct hash * vm, struct vm_entry * vme){
    if(!hash_insert(vm, &vme->elem))//해시 테이블에 vm_entry 추가
        return true;
    return false;
}

bool delete_vme(struct hash *vm, struct vm_entry *vme){
    if(hash_delete(vm,&vme->elem)){    //해시 테이블에서 vm_entry 제거 및 메모리 해제
        free(vme);
        return true;
    }
    return false;
}
```

Insert_vme 함수는 해시 테이블에 vm_entry 를 저장하기 위한 함수로서, hash_insert 함수를 이용하여 구현하였으며, delete_vme 함수는 vm_entry 를 제거하는 함수로서 hash_delete 함수를 이용하여 해시 테이블에서 제거하였으며, free 함수를 통해 메모리를 해제 시켜 주었다.

d. find_vme 함수

```
struct vm_entry * find_vme(void * vaddr){        //현재 프로세스의 주소공간에서 vaddr에 해당하는 vm_entry 검색
    struct vm_entry vme;

    void * vaddr_ = pg_round_down(vaddr);
    vme.vaddr = vaddr_;

    struct hash_elem * elem = hash_find(&thread_current()->vm,&vme.elem);
    if(!elem) return NULL;

    return hash_entry(elem, struct vm_entry, elem);
}
```

find_vme 함수는 해시 테이블에 해당 주소에 해당하는 vm_entry 가 존재하는지 확인하는 함수로서 인자로 받은 주소의 가상 번호와 vm_entry 구조체가 가지고 있는 가상 번호가 일치하는지 확인을 통해 vm_entry 를 확인한다.

e. vm_destroy 함수 / vm_destroy_func 함수

```
void vm_destroy(struct hash * vm){
    hash_destroy(vm, &vm_destroy_func);      //해시 테이블 제거
}

void vm_destroy_func(struct hash_elem *e, void * aux UNUSED){      //vm_entry 제거
    struct vm_entry * vme = hash_entry(e, struct vm_entry, elem);

    if(vme->is_loaded){
        free_page(pagedir_get_page(thread_current()->pagedir,vme->vaddr));      //vm_entry에 할당된 페이지 제거
        pagedir_clear_page(thread_current()->pagedir,vme->vaddr);      //페이지 테이블에서 해당 주소의 엔트리 제거
    }
    else{
        if(vme->swap_slot != -1)
            bitmap_set_multiple (swap_bitmap, vme->swap_slot ,1 , false);      //vme의 type이 ANON일 경우 swap_slot초기화
    }
    delete_vme(&thread_current()->vm, vme);      //vm_entry 제거
}
```

vm_destroy 함수는 해시 테이블을 제거하는 함수로서 hash_destroy 함수를 이용하여 해시 테이블을 제거해준다. 해당 함수의 두 번째 인자로 주어진 함수는 해시 버켓의 entry 를 제거해주는 함수로서, 해당 인자를 통해 해시 테이블이 제거될 때 vm_entry 도 함께 제거 시켜준다.

vm_destroy_func 함수는 vm_destroy 함수에 두 번째 인자로 사용되는 함수로서 vm_entry 를 제거 시켜주는 함수이다. 만약 vm_entry 에 해당하는 주소의 물리페이지가 할당이 되어 있다면 해당 물리페이지의 메모리 해제와 함께 페이지 테이블에서 해당 주소의 엔트리를 제거 시켜준다.

```
vm_destroy(&thread_current()->vm);      //vm 해시 테이블 제거
pd = cur->pagedir;
if (pd != NULL)
{
```

위의 코드와 같이 process_exit 함수에서 vm_destroy 함수를 호출해줌으로서 프로세스가 종료될 때 해시테이블도 함께 제거 시켜준다.

f. load_segment 함수

```
//물리메모리를 바로 할당해주지 않고 vm_entry를 생성한다음 vm테이블에 추가
    struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
    if (vme == NULL)
      return false;

    vme->type = VM_BIN;
    vme->vaddr = upage;
    vme->writable = writable;
    vme->is_loaded = false;
    vme->file = file;
    vme->read_bytes = page_read_bytes;
    vme->zero_bytes = page_zero_bytes;
    vme->offset = ofs;
    vme->swap_slot = -1;
    if(insert_vme(&thread_current()->vm, vme) == false){
      free(vme);
      return false;
    }


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

load_segment 함수를 통해 주소 공간을 초기화 시켜줄 때, 기존 물리메모리를 탑재하는 함수들을 모두 지워주고, 메모리 크기에 맞추어 vm_entry 를 할당 시켜준다. vm_entry 구조체의 멤버들을 주어진 값들을 통해 설정 시켜주고, vm_entry 를 관리하는 해시테이블에 insert_vme 함수를 통해 생성한 vm_entry 를 넣어준다. 여기서, 아직 물리 메모리가 할당되지는 않았으므로, vme 의 is_loaded 구조체 멤버는 false 로 설정 시켜준다.

g. setup_stack 함수

```
static bool
setup_stack (void **esp)
{
  struct page *kpage;
  bool success = false;
  // stack은 바로 사용되므로 물리메모리 미리 할당(vm_entry로 같이 생성)
  struct vm_entry *vme = (struct vm_entry *)malloc(sizeof(struct vm_entry));
  if(vme == NULL)
    return false;

  vme->type = VM_BIN;
  vme->vaddr = ((uint8_t *)PHYS_BASE) - PGSIZE;
  vme->writable = true;
  vme->is_loaded = false;
  vme->swap_slot = -1;
  insert_vme(&thread_current()->vm, vme);//vm table에 vm_entry 추가

  kpage = alloc_page (PAL_USER | PAL_ZERO);  //메모리 할당
  if (kpage != NULL){
    success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage->kaddr, true);
    if (success){
      *esp = PHYS_BASE;
      vme->is_loaded =true;
      kpage->vme = vme;
    }
    else{  //물리페이지와 가상페이지 매핑 실패시 페이지 할당 해제
      del_page_from_lru_list(kpage);
      palloc_free_page(kpage->kaddr);
      free(kpage);
    }
  }
  return success;
}
```

stack 을 할당해주는 함수로서 load_segment 함수와 마찬가지로 메모리에 대응하여 vm_entry 를 할당 시켜준다. 다만, stack 같은 경우는 프로세스에서 바로 사용되는 메모리 공간이니, 이 경우 물리 메모리의 할당과 매핑을 함께 해주는 대신 vm_entry 구조체의 멤버인 is_loaded 를 true 로 바꾸어 준다.

h. check_address 함수 / check_valid_buffer 함수/ check_valid_string 함수

```
struct vm_entry * check_address(void *addr, void * esp UNUSED){

  if((void*)0x08048000 > addr || addr >= (void*)0xc0000000){
    exit(-1);
  }//check the address is user space

  struct vm_entry * vme = find_vme(addr);  //해당 주소의 vm_entry를 찾은 다음 리턴
  return vme;
}
```

```
void check_valid_buffer(void * buffer, unsigned size, void * esp, bool to_write){
    int i = 0;
    void* buffer_ = (char *)buffer;
    struct vm_entry * vme;
//buffer의 유효성 검사
    while(i<(int)size){
        vme = check_address(buffer_,esp);//buffer의 주소가 유효한 가상주소인지 검사
        if(vme == NULL) exit(-1);
        if(to_write && !vme->writable) exit(-1);        //write가 불가능한 주소 영역에 write를 할경우 비정상 종료
        i += PGSIZE;
        buffer_ += PGSIZE;
    }
}

void check_valid_string(const void * str, void * esp){
    char *str_ = (char*)str;
    struct vm_entry *vme;
//문자열의 주소값이 유효한 가상주소인지 검사
    while(*str_ != 0){
        vme = check_address((void*)str_, esp);
        if(vme == NULL) exit(-1);
        str_++;
    }
}
```

기존 check_address 함수에서 해당 주소에 대한 vme_entry 가 존재하는지 확인하는 코드를 추가하였다. 또한, 해당하는 vm_entry 를 리턴 값으로 변경시켜주었다.

check_valid_buffer 함수는 write, read syscall 에서 사용되는 함수로서 해당 버퍼의 주소가 유효한 가상 주소인지 검사하는 함수이다. 버퍼의 주소에 해당하는 모든 vm_entry 에 대해 존재 유무를 체크한다.

check_valid_string 함수는 인자로 주어진 문자열의 주소 값이 유효한 가상 주소인지 검사하는 함수로서 마찬가지로 해당 주소에 vm_entry 가 존재하는지 확인을 통해 체크한다.

i. syscall_handler 함수

```
case SYS_OPEN:
    get_argument(sp,arg,1);
    check_valid_string((const void*)arg[0],sp);        // check the address if the argument is on user space
    f->eax = open((const char*)arg[0]);        // store the return value on eax value in intr_frame structure
    break;
case SYS_FILESIZE:
    get_argument(sp,arg,1);
    f->eax = filesize(arg[0]);
    break;
case SYS_READ:
    get_argument(sp,arg,3);
    check_valid_buffer((void*)arg[1],(unsigned)arg[2], sp, true);
    f->eax = read (arg[0],(void*)arg[1],(unsigned)arg[2]);
    break;
case SYS_WRITE:
    get_argument(sp,arg,3);
    check_valid_buffer((void*)arg[1],(unsigned)arg[2], sp, false);
    f->eax = write(arg[0],(const void*)arg[1],(unsigned)arg[2]);
    break;
```

위에서 설명한 check_valid_buffer 함수를 통해 read, write syscall 을 할 때 주어진 buffer 에 대해 올바른 주소인지 확인을 하고, check_valid_string 을 통해 파일 이름 등이 주어졌을 때 해당 string 이 올바른 주소에 위치해 있는지를 확인하도록 syscall_handler 를 변경시켜주었다.

j. page_fault 함수 / handle_mm_fault 함수

```
  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){
    bool pg_load = false;
    struct vm_entry *vme = check_address(fault_addr, f->esp); //해당 주소의 vme_entry 접근
  if(vme == NULL){
      exit(-1);
    }
    if(!(vme->writable) && write) exit(-1);   //write하지 못하는 주소에 write 실행시 강제 종료
    pg_load = handle_mm_fault(vme);   //물리 페이지 할당
    if(!pg_load)
      exit(-1);
  }else      //read only page에 대한 접근일 경우
    exit(-1);
```

물리페이지 공간이 매핑 되어 있지 않아(vm_entry 만 존재하여) page_fault 가 발생할 경우 위와 같이 handle_mm_fault 를 실행 시켜주었다. Read only page 에

대한 접근이거나 write 하지 못하는 주소에 write 를 하려는 경우, 올바른 주소가 아닐 경우에는 모두 에러 처리를 해주었다.

```
bool handle_mm_fault(struct vm_entry * vme){
  bool load_pg=false;
  struct page * page=alloc_page(PAL_USER);    //메모리 할당
  if(page==NULL)return false;
  page->vme=vme;
  switch(vme->type)
  {
    case VM_BIN:
    case VM_FILE:
      load_pg=load_file(page->kaddr, vme);    //해당 페이지를 물리 메모리에 로드
      if(load_pg)
        load_pg=install_page(vme->vaddr, page->kaddr, vme->writable);   //물리페이지와 가상페이지 맵핑
      break;
    case VM_ANON:    //swap 페이지
      swap_in(vme->swap_slot, page->kaddr);    //메모리로 해당 페이지 swap
      vme->swap_slot=-1;    //swap_slot 초기화
      load_pg=install_page(vme->vaddr, page->kaddr, vme->writable);   //물리페이지와 가상페이지 맵핑
      break;
  }
  if(!load_pg){
    __free_page(page);
  }else
    vme->is_loaded=true;    //물리 메모리에 로드 되었음을 표시
  return load_pg;
}
```

vme 에 대응되는 주소에 물리 페이지를 할당해주고 매핑 시켜주는 함수로서, alloc_page 함수(swap 에서 구현)를 통해 페이지를 할당 시켜주고, load_file 함수를 통해 생성한 페이지를 물리 메모리에 로드, install_page 함수를 통해 물리 페이지와 가상 페이지를 매핑 시켜주었다. vme 의 type 이 VM_FILE 경우에는, MMAP 에서 사용되며, VM_ANON 같은 경우는 SWAP 에서 사용된다. 만약 로드나 매핑이 정상적으로 이루어지지 않으면 페이지를 해제 시켜 주고, 정상적으로 이루어졌을 경우에는 vme 의 로드 상태를 TRUE 로 바꾸어 준다.

k. load_file 함수

```
bool load_file(void *kaddr, struct vm_entry * vme){    //disk에 존재하는 page를 물리 메모리로 로드
  off_t r_bytes=file_read_at(vme->file, kaddr, vme->read_bytes, vme->offset);
  if((size_t)r_bytes != vme->read_bytes)
    return false;
  memset(kaddr+r_bytes,0,vme->zero_bytes);    //4KB 페이지의 빈 공간은 0으로 채움

  return true;
}
```

file_read_at 함수를 통해 Disk 에 존재하는 파일을 주어진 물리 메모리에 로드하는 함수로서 페이지의 남는 메모리 공간은 0 으로 채워준다.

2. Memory Mapped File

과제 목표:

mmap() 함수와 munmap() 함수의 구현

과제 설명:

Read 와 write 시스템 콜은 커널 stack 으로 context switch 가 발생하기 때문에 overhead 가 존재하므로, 대신 메모리 접근을 통해 파일을 접근할 수 있는 mmap 과 munmap 함수를 따로 구현한다.

a. mmap_file 구조체

```
struct mmap_file{
    int mapid;      //MMAP 성공시 리턴되는 MAPPING_ID
    struct file* file;      //매핑하는 파일
    struct list_elem elem;
    struct list vme_list;       //mmap_file에 해당하는 모든 vm_entry들의 리스트
};
```

매핑된 파일의 정보를 저장하는 구조체로 스레드의 mmap file 을 해당 구조체를 통해 관리한다. Mapid 는 mmap 성공시 리턴되는 mapping id, file 은 매핑하는 파일의 파일 오브젝트, vme_list 는 mmap_file 에 해당하는 모든 vm_entry 의 elem 이 저장된 리스트이다.

b. mmap 함수

```
int mmap(int fd, void * addr){//해당 주소에 파일 매핑
  int rt = -1;
  struct file * f = NULL;
  struct mmap_file *mf;

  if(fd != 0 && fd != 1)
    f = process_get_file(fd);

  if((f != NULL) && (file_length(f) != 0) && ((void *)0x08048000 <= addr || addr < (void *)0xc0000000) && (((uint32_t)addr % PGSIZE) == 0) && (addr != NULL)){
    //파일이 존재하지 않거나 길이가 0, 잘못된 주소 혹은 PGSIZE단위의 주소가 아닐 경우 실행 X
    struct file *rf = file_reopen(f);//user에서 open한 file을 close할수 있기 때문에 안전을 위해 reopen
    if(rf == NULL){
      return -1;
    }
    mf = (struct mmap_file*)malloc(sizeof(struct mmap_file));//mmap_file 구조체 생성 및 초기화
    if(mf == NULL){
      return -1;
    }
    mf -> mapid = thread_current()->mapid++;
    mf -> file = rf;
    list_init(&mf->vme_list);

    uint32_t read_bytes = file_length(rf);
    off_t offset = 0;
    while(read_bytes>0){////파일 다 읽을때까지 해당 page단위 주소마다 vme생성

      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));//해당 주소의 vm_entry 생성 및 초기화
      if (vme == NULL){
        return -1;
      }
```

```
      vme->type = VM_FILE;
      vme->vaddr = addr;
      vme->writable = true;
      vme->is_loaded = true;
      vme->file = rf;
      vme->read_bytes = page_read_bytes;
      vme->zero_bytes = page_zero_bytes;
      vme->offset = offset;

      list_push_back(&mf->vme_list, &vme->mmap_elem);//vme_list에 생성한 vme 추가
      if(!insert_vme(&thread_current()->vm, vme)){//vm 해시테이블에 생성한 vme 추가
        return -1;
      }

      read_bytes -= page_read_bytes;
      offset += page_read_bytes;
      addr += PGSIZE;
    }
    list_push_back(&thread_current()->mmap_list, &mf->elem);//mmap_list에 생성한 mmap_file 구조체 추가
    rt = mf->mapid;
  }
  return rt;
}
```

인자로 주어진 주소에 파일을 매핑하는 함수로서 주어진 주소가 유저 영역이 아니거나, PGSIZE 의 단위의 주소가 아니거나, 주어진 파일이 존재하지 않는 파일이거나 길이가 0 이면 예외 처리를 해준다. 해당 조건을 만족 시킨다면, 기존 열려 있는 파일이 USER 에서 close 될 수 있기 때문에 안전을 위해 reopen 해준 다음, mmap_file 구조체와 PGSIZE 메모리 크기당 vm_entry 구조체를 생성한 다음(해당 파일을 모두 다 읽을 때 까지), 각 파일 정보를 넣어준다. vm_entry

경우 insert_vme 함수를 통해 해시테이블에 넣어주고, mmap_file 구조체의 vme_list 에 넣어주며, mmap_file 구조체의 경우 스레드의 mmap_list 에 넣어주어 관리 해준다.

c. munmap 함수 / do_munmap 함수

```
void munmap(int mapping){////mmap_list 내에서 mapping에 해당하는 mapid를 가진 vm_entry모두 해제
    struct list_elem * elem = list_begin(&thread_current()->mmap_list);
    for(;elem != list_end(&thread_current()->mmap_list);){
        struct mmap_file * mf = list_entry(elem,struct mmap_file, elem);
        if(mapping == CLOSE_ALL || mf->mapid == mapping){////mapping값이 CLOSE_ALL일 경우 모든 파일 매핑 제거
            do_munmap(mf);//////해당 mmap_file의 vme_list에 연결된 vm_entry 모두 제거
            file_close(mf->file);
            elem = list_remove(elem);
            free(mf);
        }
        else elem = list_next(elem);
    }
}

void do_munmap(struct mmap_file * map_file){
    struct list_elem * elem = list_begin(&map_file->vme_list);
    for(;elem != list_end(&map_file->vme_list);){
        struct vm_entry * vme = list_entry(elem, struct vm_entry, mmap_elem);

        if(vme->is_loaded){
            if(pagedir_is_dirty(thread_current()->pagedir, vme->vaddr)){////해당 페이지가 dirty할 경우 디스크에 메모리 내용 기록
                lock_acquire(&filesys_lock);
                file_write_at(vme->file, vme->vaddr, vme->read_bytes, vme->offset);
                lock_release(&filesys_lock);
            }
            free_page(pagedir_get_page(thread_current()->pagedir,vme->vaddr));//해당 page 할당 제거
            pagedir_clear_page(thread_current()->pagedir,vme->vaddr);//////page mapping 해제
        }
        elem = list_remove(elem);//////vme_list에서 vme제거
        delete_vme(&thread_current()->vm,vme);//////////vme 메모리 해제
    }
}
```

Munmap 함수는 mmap_list 내에서 mapping 에 해당하는 mapid 를 가진 mmap_file 을 모두 제거 하는 함수로서 mmap_list 를 순차적으로 확인하여 mapid 를 비교하여 찾아낸다. Mmap_file 의 vme_list 에 연결된 vme_list 를 제거하는 데에는 do_munmap 함수를 이용하였으며, 만약 주어진 mapid 가 CLOSE_ALL(-1)일 경우 모든 mmap_file 을 제거 시켜 주었다.

do_munmap 함수는 mmap_file 내 vme_list 에 들어 있는 vme_entry 를 모두 제거하는 함수로서, 만약 해당 vme_entry 가 load 된 상태라면 해당 페이지의 내용을 disk 에 기록해주고(dirty 일 경우) 할당된 페이지도 함께 해제 시켜준다.

d. handle_mm_fault 함수 / process_exit 함수

```
····case·VM_FILE:
·····load_pg·=·load_file(page->kaddr,·vme);····//해당·페이지를·물리·메모리에·로드
·····if(load_pg)
·····load_pg·=·install_page(vme->vaddr,·page->kaddr,·vme->writable);···//물리페이지와·가상페이지·맵핑
·····break;
```

handle_mm_fault 함수에서 물리 페이지를 할당시켜주는데 있어 vme_entry 의 type 이 VM_FILE (MMAP_FILE 을 통해 생성된 vme_entry)인 경우에 대한 handling 을 추가 시켜준다. (VM_BIN 과 동일)

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

··munmap(CLOSE_ALL);····//mmap_list·내에·모든·vm_entry제거
```

프로세스가 종료될 때 mmap_list 내에 있는 모든 mmap_file 과 vme_entry 를 제거 시켜 주기 위해 process_exit 함수에 munmap(CLOSE_ALL) 명령어를 호출 시켜준다.

3. Swapping

과제 목표:

현재 제공 되고 있지 않은 swapping 지원 구현

과제 설명:

물리 페이지가 부족할 경우 victim 페이지를 선정하여 메모리의 내용을 디스크로 복사 한 다음 해당 메모리를 해제하여 여유 메모리 공간 할당. Victim 페이지를 선정하는데 있어서는 LRU 알고리즘(CLOCK 알고리즘)을 사용하여 구현.

a. page 구조체

```
struct page{
    void *kaddr;        //페이지의 물리주소
    struct vm_entry *vme;       //물리 페이지가 매핑된 가상 주소의 vm_entry
    struct thread *thread;      //해당 물리 페이지를 사용중인 스레드
    struct list_elem lru;
};
```

유저에게 할당된 각 물리 페이지를 표현하는 구조체로 페이지의 물리주소, 물리 페이지가 매핑된 가상 주소의 vm_entry, 해당 물리 페이지를 사용중인 스레드 등을 구조체 멤버로 가지고 있다.

b. lru_list_init 함수 / add_page_to_lru_list 함수 / del_page_from_lru_list 함수

```
void lru_list_init(void){      //lru_list 초기화
    list_init(&lru_list);
    lock_init(&lru_lock);
    lru_clock = NULL;
}
void add_page_to_lru_list(struct page *page){      //lru_list에 page elem 추가
    lock_acquire(&lru_lock);
    list_push_back(&lru_list,&page->lru);
    lock_release(&lru_lock);
}

void del_page_from_lru_list(struct page *page){      //lru_list에서 page elem 제거
    lock_acquire(&lru_lock);
    if(lru_clock == &page->lru)
        lru_clock = list_remove(&page->lru);    //해당 페이지 elem이 lru_clock이었을 경우 lru_clock을 다음 elem으로 설정
    else
        list_remove(&page->lru);
    lock_release(&lru_lock);
}
```

lru_list_init 함수는 전역으로 주어진 lru_list(사용자 프로세스에게 할당된 물리 페이지들의 리스트)와, lru_list 에 접근할 때 사용되는 lru_lock, victim page 를 선정할 때 사용되는 clock elem 을 모두 초기화 시켜주는 함수이다.

해당 함수는 main 함수에서 호출 하여 처음 프로그램이 실행될 때 lru_list 에 대한 영역을 초기화 시켜준다.

add_page_to_lru_list 함수는 전역으로 선언된 lru_list 에 page elem 을 추가시켜주는 함수로서 alloc_page 를 통해 page 가 할당될 경우 해당 함수를 통해 list 에 추가 시켜주어 관리한다.

del_page_from_lru_list 함수는 lru_list 에서 해당 page 의 elem 을 제거하는 함수로서, add_page_to_lru_list 와 함께 lru_list 의 관리에 대한 함수이다. 이 두 함수 모두 lru_list 에 접근하므로 lru_lock 을 통해 임계 영역에 대한 상호배제 처리를 해준다.

c. alloc_page 함수

```
struct page *alloc_page(enum palloc_flags flags){//페이지 할당 함수
    struct page * page = (struct page *)malloc(sizeof(struct page));
    if(page == NULL)return NULL;
    page->thread = thread_current();
    page->kaddr = palloc_get_page(flags);
    while(page->kaddr == NULL){
        page->kaddr = try_to_free_pages(flags);//페이지 할당 공간이 없을 경우 기존 페이지를 swap한다음 페이지 공간 생성 후 다시 할당
        if(page->kaddr != NULL)break;
    }
    add_page_to_lru_list(page);//lru list에 해당 page elem 추가
    return page;//page 구조체 리턴
}
```

페이지를 할당 해주는 함수로서 기존 palloc_get_page 와 유사한 기능을 제공한다. 다만, 할당하고자 하는 페이지가 더 이상 없는 경우 try_to_free_page 로 victim page 를 선정하여 swap 시켜 준 다음 물리 메모리를 제공하며 제공된 페이지는 add_page_to_lru_list 함수를 통해 lru_list 에 넣어주어 관리한다.

d. free_page 함수 / __free_page 함수

```
void free_page(void * kaddr){//물리 주소 kaddr에 해당하는 page 구조체를 lru_list에서 검색 후 __free_page 함수 호출
    if(kaddr == NULL) return;
    struct page * pg = NULL;
    struct list_elem * elem = list_begin(&lru_list);
    for(;elem != list_end(&lru_list);elem = list_next(elem)){
        pg = list_entry(elem, struct page, lru);
        if(pg->kaddr == kaddr) break;
    }
    if(elem != list_end(&lru_list))
        __free_page(pg);
}

void __free_page(struct page * page){
    del_page_from_lru_list(page);//lru 리스트에서 page elem 제거
    page->vme->is_loaded = false;
    palloc_free_page(page->kaddr);//해당 페이지의 물리주소 메모리와 페이지 구조체 해제
    free(page);
}
```

condition variable 을 통해 lock 을 잡아주는 경우 waiters 리스트에 스레드를 넣어주는데 있어 기존 list_push_back 함수를 통해 우선 순위와 상관없이 넣어준 것에 반해 list_insert_ordered 함수를 통해 높은 우선순위 순서로 넣어준다.

e. get_next_lru_clock 함수

```
static struct list_elem * get_next_lru_clock(void){//lru_list에서 다음 page elem 리턴

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

Lru_list 에서 다음 page 의 elem 을 lru_clock 변수에 할당해 주는 함수로서 만약 lru_clock 이 list 의 마지막 elem 이 되었을 경우 다시 list 의 첫 번째 elem 으로 변경 시켜준다.

f. try_to_free_pages 함수

```
void* try_to_free_pages(enum palloc_flags flags){////희생 페이지 선택 lru방식 채택
  lock_acquire(&lru_lock);
  struct list_elem * elem = get_next_lru_clock();
  struct page * page = list_entry(elem, struct page, lru);
  while(1){
    if(!pagedir_is_accessed(page->thread->pagedir,page->vme->vaddr))//최근에 해당 페이지를 접근한 경우 없을시 채택
      break;
    pagedir_set_accessed(page->thread->pagedir,page->vme->vaddr,false);//해당 페이지 accessed bit false로 설정
    elem = get_next_lru_clock();
    page = list_entry(elem, struct page, lru);
  }
  bool is_dirty = pagedir_is_dirty(page->thread->pagedir, page->vme->vaddr);//해당 페이지 dirty(페이지 내용 변경이 있는지) 체크
  switch(page->vme->type){
    case VM_BIN:
      if(is_dirty){
        page->vme->swap_slot = swap_out(page->kaddr);//디스크로 해당 페이지 swap
        page->vme->type =VM_ANON;//type 변경(다음번 swap_in을 위해)
      }
      break;
    case VM_FILE:
      if(is_dirty)
        file_write_at (page->vme->file, page->vme->vaddr, page->vme->read_bytes, page->vme->offset);//파일에 변경 내용 저장
      break;
    case VM_ANON:
      page->vme->swap_slot = swap_out(page->kaddr);//디스크로 해당 페이지 swap
    break;
  }
  lock_release(&lru_lock);
  pagedir_clear_page(page->thread->pagedir,page->vme->vaddr);//해당 페이지의 물리 메모리 할당 제거 및 페이지 구조체 해제
  __free_page(page);
  return palloc_get_page(flags);//새로운 페이지 할당
}
```

물리 페이지를 할당할 공간이 가득 찬 경우 victim 페이지를 선택해 주는 함수로서 기본적으로 lru 방식을 사용한다. get_next_lru_clock 함수를 통해 list 를 순회 하면서 해당 페이지에 대해 accessed bit 가 false 일 경우 해당 페이지를 선출 해주며, true 일 경우에는 false 로 변경 시켜주고 다음 elem 의 페이지를 확인한다. 이와 같이 victim page 를 선택하고 나서, 만약 해당 page 가 dirty bit 가 true 일 경우 해당 페이지를 disk 에 써주고 VM_BIN, VM_ANON type 일 경우 swap_out 을 통해 디스크에 적어준다. 이 후 할당된 메모리를 해제 시켜주어 공간을 확보한다음, 해당 페이지의 vme 의 load 상태는 false 로 변경시켜 준다.

마지막으로 할당 된 공간에 다시 page 를 할당 시켜 주어 리턴 해 준다.

g. swap_init 함수

```
void swap_init(void){//스왑 영역 초기화
  swap_block = block_get_role(BLOCK_SWAP);
  if(swap_block == NULL) exit(-1);
  size_t size = (size_t)block_size(swap_block) * BLOCK_SECTOR_SIZE/PGSIZE;
  swap_bitmap = bitmap_create(size);
}
```

Swap 영역에 대한 block 할당과 함께, swap slot 의 사용 가능 여부를 표시하기 위해 bitmap 을 생성 시켜준다. Bitmap 이 true 일 경우 해당 공간이 swap_in 되어 있는 상태이고, false 일 경우 out 되어 있는 상황이다.

해당 함수는 main 함수에서 호출 하여 처음 프로그램이 실행될 때 swap 영역에 대한 초기화를 시켜준다.

h. swap_in 함수 / swap_out 함수

```
void swap_in(size_t used_index, void *kaddr){//disk->memory
    //used_index에 저장된 데이터를 kaddr주소(메모리)로 복사
    lock_acquire(&filesys_lock);
    if(BITMAP_ERROR == bitmap_scan_and_flip(swap_bitmap,used_index,(size_t)1,true)){
        //swap_bitmap false로
        lock_release(&filesys_lock);
        exit(-1);
    }
    size_t i = 0;
    for(i=0;i<8;i++)
        block_read(swap_block, used_index*8+i, (char*)kaddr + BLOCK_SECTOR_SIZE * i);
    lock_release(&filesys_lock);
}

size_t swap_out(void* kaddr){//memory -> disk
    //kaddr 주소가 가리키는 페이지를 스왑 파티션에 기록
    lock_acquire(&filesys_lock);
    size_t index = bitmap_scan_and_flip(swap_bitmap, 0, 1, false);  //swap_bitmap true로
    if(BITMAP_ERROR == index){
        lock_release(&filesys_lock);
        exit(-1);
    }
    size_t i = 0;
    for(i=0;i<8;i++)
        block_write(swap_block,index*8+i, kaddr + BLOCK_SECTOR_SIZE * i);
    lock_release(&filesys_lock);
    return index;       //페이지를 기록한 swap slot 번호 리턴
}
```

Swap_in 함수는 used_index 의 swap 공간을 해당 메모리 주소에 복사 해주는 함수로서, swap-out 된 페이지를 다시 메모리로 탑재 시켜주는 함수이다. 이때 해당 index 의 swap 공간은 더 이상 사용하지 않으므로 bitmap 을 false 로 바꿔준다.

Swap_out 함수는 해당 메모리 주소의 페이지를 스왑 파티션에 기록해주는 함수로서 bitmap_scan_and_flip 함수를 통해 false 인 index 를 찾아 true 로 변환 시켜준다. 해당 함수를 통해 찾은 index 에 메모리 주소의 페이지 내용을 기록시켜준다.

i. handle_mm_fault 함수 / setup_stack 함수 / do_munmap 함수

```
bool handle_mm_fault(struct vm_entry * vme){
  bool load_pg =false;
  struct page * page = alloc_page(PAL_USER);    //메모리 할당
  if(page==NULL) return false;
  page->vme = vme;
  switch(vme->type)
  {
    case VM_BIN:
    case VM_FILE:
      load_pg = load_file(page->kaddr, vme);    //해당 페이지를 물리 메모리에 로드
      if(load_pg)
        load_pg = install_page(vme->vaddr, page->kaddr, vme->writable);    //물리페이지와 가상페이지 맵핑
      break;
    case VM_ANON:    //swap 페이지
      swap_in(vme->swap_slot, page->kaddr);    //메모리로 해당 페이지 swap
      vme->swap_slot = -1;    //swap_slot 초기화
      load_pg = install_page(vme->vaddr, page->kaddr, vme->writable);    //물리페이지와 가상페이지 맵핑
      break;
  }
  if(!load_pg){
    __free_page(page);
  }else
    vme->is_loaded= true;    //물리 메모리에 로드 되었음을 표시
  return load_pg;
}
```

기존 handle_mm_fault 에서 palloc_get_page 를 통해 page 를 할당해 주는 부분을 alloc_page 함수로 바꿔주고, 만약 vme type 이 VM_ANON 인 경우 즉, swap-out 된 페이지인 경우, swap_in 을 통해 다시 메모리에 복사 해주는 경우를 추가 시켜준다.

```
  kpage = alloc_page (PAL_USER | PAL_ZERO);    //메모리 할당
  if (kpage != NULL){
    success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage->kaddr, true);
    if (success){
      *esp = PHYS_BASE;
      vme->is_loaded =true;
      kpage->vme = vme;
    }
    else{    //물리페이지와 가상페이지 매핑 실패시 페이지 할당 해제
      del_page_from_lru_list(kpage);
      palloc_free_page(kpage->kaddr);
      free(kpage);
    }
  }
  return success;
}
```

Setup_stack 함수 같은 경우도 메모리 할당에 대한 함수를 alloc_page 함수로 변경 시켜준다.

```
void do_munmap(struct mmap_file * map_file){
    struct list_elem * elem = list_begin(&map_file->vme_list);
    for(;elem != list_end(&map_file->vme_list);){
        struct vm_entry * vme = list_entry(elem, struct vm_entry, mmap_elem);

        if(vme->is_loaded){
            if(pagedir_is_dirty(thread_current()->pagedir, vme->vaddr)){    //해당 페이지가 dirty할 경우 디스크에 메모리 내용 기록
                lock_acquire(&filesys_lock);
                file_write_at(vme->file, vme->vaddr, vme->read_bytes, vme->offset);
                lock_release(&filesys_lock);
            }
            free_page(pagedir_get_page(thread_current()->pagedir,vme->vaddr));  //해당 page 할당 제거
            pagedir_clear_page(thread_current()->pagedir,vme->vaddr);       //page mapping 해제
        }
        elem = list_remove(elem);       //vme_list에서 vme제거
        delete_vme(&thread_current()->vm,vme);          //vme 메모리 해제
    }
}
```

do_munmap 함수에서는 메모리 해제에 대한 부분을 free_page 함수로 변경시켜준다.

참고 사항

Gcc 4.5 버전에서 실행시 pt-write-code.c 에 대한 test 가 문제가 발생하여 gcc 4.4 버전에서 해당 프로젝트를 진행하였습니다.

실행 결과

```
pass tests/filesys/base/syn-write
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
```

```
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/vm/pt-grow-stack
FAIL tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
FAIL tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
FAIL tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
FAIL tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
pass tests/vm/page-shuffle
pass tests/vm/mmap-read
pass tests/vm/mmap-close
pass tests/vm/mmap-unmap
pass tests/vm/mmap-overlap
pass tests/vm/mmap-twice
pass tests/vm/mmap-write
pass tests/vm/mmap-exit
pass tests/vm/mmap-shuffle
pass tests/vm/mmap-bad-fd
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
6 of 109 tests failed.
```

Make check 결과 총 109 개의 test 에서 6 개 영역에서 fail 이 발생하였다. 이중 5 개는 stack 과 관련된 내용이며, 결과적으로 1 개의 test 에 대해 해결 하지 못하였는데, lock 충돌과 관련하여 오류가 발생하였다.