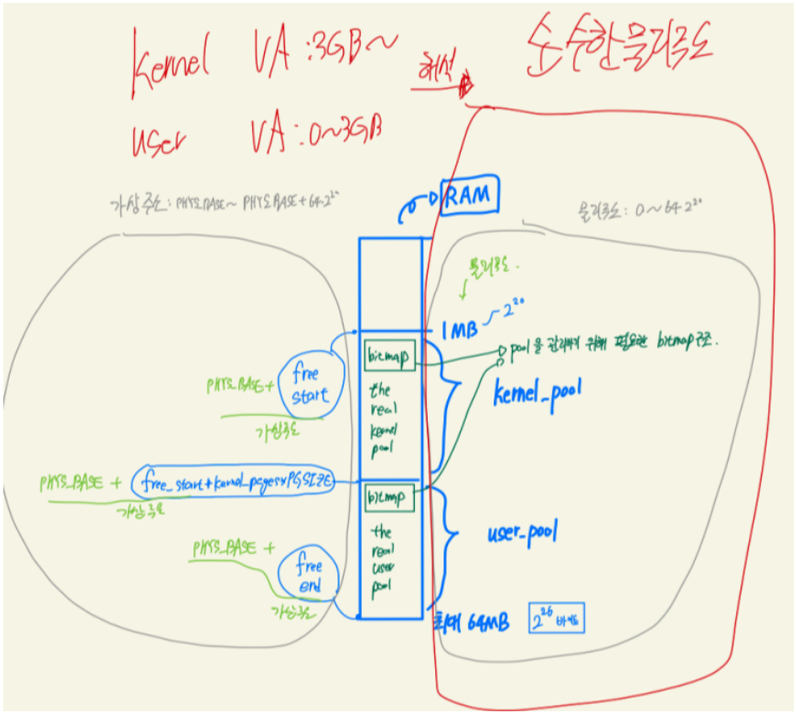
Lab 3 Final Report

Member #1
Name: Suwon Yoon
Student ID: 20210527

Member #2
Name: Jiwon Lee
Student ID: 20210706

Physical Address / Kernel Virtual Address / User Virtual Address



OS에서 사용하는 주소는 모두 가상주소이다. HW가 사용하는 page directory에 저장하는 주소랑 page table에 저장하는 주소가 순수한 물리주소(==물리주소)임.

OS에서 사용하는 주소는 모두 가상주소라고 했는데, 이는…

- kernel virtual address
  - 얘는 순수한 물리주소+PHYS_BASE으로 계산되어서 PHYS_BASE만 빼면 순수한 물리주소 나옴.
  - kernel virtual address의 메모리 주소가… 3GB~
- user virtual address
  - 얘는 page directory랑 page table 타고 들어가서 pte봐야 순수한 물리주소 나옴.
  - user virtual address 메모리주소가… 0~3GB

kernel virtual address도 user virtual address와 동일한 원리로 cpu에서 page directory와 page table을 참고하여 순수한 물리주소로 해석된다. 인간이 봤을때는 kernel virtual address

에 - PHYS_BASE를 해서 순수한 물리주소를 얻어내면 될 것 같지만 하드웨어(cpu)님께서 page directory를 통해 순수한 물리주소를 계산하도록 디자인되셨기에 kernel virtual address도 user virtual address와 같은 방법으로 해석되게 한다.(커널가상주소인지 유저가상주소인지 판별하여 각각의 경우 다른 '순수한 물리주소 찾기' 방법을 적용하는 하드웨어를 만들라면 만들 수도 있겠지. 근데 번거롭잖아.) kernel virtual address를 위한 page directory와 page table은 paging_init 함수에서 생성된다. 생성되는 page directory의 이름은 init_page_dir이다.

kernel virtual address는 커널에서 주소 연산을 할 때 사용된다. OS에서 우리가 사용하는 포인터는 kernel virtual address인 것이다.

커널 코드(우리가 작성하는 c코드)에 있는 kernel virtual address를 'cpu'가 해석할 때는 cr3레지스터에 들어있는(얘는 init_page_dir이 들어있는 곳의 순수한 물리주소를 담고있어야함) page directory를 참고한다. cpu는 순수한 물리 주소 밖에 모르는 순수한 아이이다.(Cr3으로 넣어줄 때 아마 vtop함, 반대로 page table entry를 가지고 올때는 ptov를 함.)

커널 코드(우리가 작성하는 c코드)에 있는 kernel virtual address를 'kernel'이 해석할 때는, 이 경우는 page table entry를 구성하거나 레지스터에서 직접 값을 넣거나 받아올때가 유일한듯, 굳이굳이 init_page_dir을 안 써도 된다. 그냥 더하기 빼기 PHYS_BASE 하면 된다. 커널이 굳이굳이 cpu의 주소 해석 방법을 따라할 필요는 없으니까.

위 두 방법 중 어떤 방법으로 해석해든 kernel virtual address → 순수한 물리주소 의 결과는 같다.

물리주소와 PHYS_BASE차이인 kernel virtual address는 왜 있는 것인가? 이에 대한 추측은 세 가지가 있음.

1. user virtual address랑 겹치지 않게 하기 위해
2. 지금은 핀토스라는 단순한 교육용오에스라서 kernel virtual address랑 순수한 물리주소랑 이런 간단한 관계를 가지지만 실제 오에스에서는 kernel virtual address도 user virtual addree와 마찬가지로 순수한 물리주소와 복잡한 관계를 가질 수 있음. 그냥 커널 가상주소 흉내내고 싶었나봐.
3. 커널에서도 가상주소가 필요하다! 커널 코드를 만들어야하니까.

Frame Table

struct list frame_table

struct lock frame_table_lock

struct list_elem clock_ptr - frame_table의 entry를 순회하는 용도

- **frame table의 entry**
  - fte_kernel_VA_for_frame: frame의 주소
  - fte_VA_for_page: page의 주소
  - spt entry의 포인터: eviction시 추가 정보를 볼 수 있음.
  - fte_thread: thread의 포인터. frame에 들어가 있는 page의 주인인 thread를 알아야 pagedir을 볼 수 있음.
  - evict되어도 문제없는지 정보.
- **frame을 allocate 하는 함수**: paging을 위함. evict할 frame을 고르고 evict할 때 적절한 동작 하기

1. 기본적으로는 palloc_get_page와 동일한 기능
2. palloc_get_page을 못하면 eviction 한다.
    - 일단 evict할 frame찾는다. 못찾으면 kernel panic
        - clock_ptr 초기화 필요. evict가 필요한 첫시점에 한다.
        - clock algo를 구현한다.
            - page가 최근에 접근된적있는지 확인하고 최근에 접근되으면 access bit를 0으로.
    - evict 진행
        - 기본적인 경우(exefile) + mmap인 경우 → dirty bit이 set 이면 파일로 쓴다.
            - stack, exe file은 swap disk로 쓰게 함.
            - mmap 은 다시 file로 쓰게 함.
        - 공통적으로는 frame table에서 빼내기 / pagedir의 페이지를 못쓰게 만들기 / 각 pool에 bit를 1→0하기 / frame table entry를 할당해제하기
    - evict한 후에 다시 palloc_get_page. 또 못하면 evict 찾는 과정부터 반복.

    frame_table_lock을 걸어둔 상태에서 무한루프 돌 수 있지만… 그냥 palloc_free_page가 공간 부족을 해소해줄 수 있을 수 있으니까 괜찮을듯. 가정: 모~든 kernel,user pool을 page table이 관리하지는 않는다. 만약 이 가정이 틀리면 while (kernel_VA_for_new_frame == NULL) 필요없음. 일단 while로 더 견고한 구현을 한것임.

    evict할 page 강제로 하나 고르게 해서 종료를 보장하는 방법 추가도 괜찮을 것 같으나 구현하지 않음.

- **frame을 free하는 함수**: 개별 페이지를 free하는 함수. allocate 하는 함수의 역연산으로 사용. 그냥 혼자 잘 있는 페이지를 아무 이유없이 지우는 것으로는 사용못함. allocate했는데 물리고싶을때 사용. 해당 frame을 다른 page directory를 통해서 찾아올 수 있으면 그 연결을 끊는 것은 이 함수의 caller 몫임.
    1. ASSERT frame 주소인지
    2. page table을 모두 본다.
        a. frame table에서 제거하고
        b. 물리 메모리에서 frame을 할당해제한다.
        c. page table entry를 할당해제 한다.
        - clock_ptr이 해제당하는 frame을 가리키고 있으면 다른 frame을 가리키게한다.
- **process 종료시 관련 frame을 모두 free하는 함수**: process가 종료될때 해당 process가 사용 중이던 frame을 제거하기 위함.

    1. thread를 주면 관련된 frame을 ‘모두’ 할당해제한다.
        - frame table을 모두 훑어야함.
        - 어떤 frame이 인자로 들어온 thread가 사용중인 거라면…
            1. frame table에서 제거하고

2. 해제하는 pagedir을 invalidate 시켜주자. 그를 위해 pagedir_clear_page 함수 사용
3. 물리 메모리에서… frame을 할당해제한다.
4. page table entry를 할당해제 한다.
   - clock_ptr이 해제당하는 frame을 가리키고 있으면 다른 frame을 가리키게한다.

반대로… thread에서 사용가능한 가상주소를 모두 훑고 그 가상주소가 담당하는 frame을 spt로부터 얻고 frame을 free해도 될 것 같은데 이건 복잡해보임. thread → frame 의 탐색이 어려워보여서 frame table → thread 탐색으로 한다.

page_fault함수에서 frame table에 원소를 추가한다. evict는 allocate 할 때 하도록 구현해뒀으니까. page_fault함수레벨에서 볼 때는 frame table에 원소추가하는 것이 끝이다.

stack은 evict의 대상이니까 palloc_get_page가 쓰이고 있던 setup_stack 함수에서 palloc_get_page 함수를 allocate_frame 함수로 대체한다.

Supporting Page Table

동기1: frame table에서 evict될 때 write back하는 것에서 관련 정보가 필요하다.

동기2: lazy loading구현할 때 필요하다.

page table을 돕는 아이니까 가상주소를 입력으로 하는게 맞다. 우리 오에스님께서 할 일이 많으셔서 필요한 정보를 기록해둬야한다. page table은 mmu(HW)가 주소 번역 용으로 잘 쓸 것이고, OS(SW)는 복잡한 일을 해줘야해서 spt가 필요하다. spt는 page table(page dir에 주렁주렁있는것)의 보조자니까 역시 thread별로 있겠다.

spt의 entry가 아닌 경우 process가 해당 가상주소를 원하지 않는 것임. (p.42)

lazy loading 이랑 wrong memory access 가 occur in simultaneously 가 의미하는게… lazy loading하는 와중에 cpu가 wrong memory access 하는 것을 처리하라는 것인듯 (조교문서 2번. page fault)

- **sup page table의 entry 정의**
  - hash_elem 넣어줘야함.
  - lazy loading을 할 때 필요한 정보.
    - file, offset, page 주소, read bytes, zero bytes, writable
  - page가 ‘지금’ 어디에 들어있는지: 1. physical memory 2. file 3. swap disk
  - evict될때 write back에서 어떤 처리를 해줘야하는지 1.파일로.(mmap경우) 2.swap disk로.
- **hash table을 쓰기 위한 기초**
  - hash_func 함수 구현: 가상 주소를 key로 해야함. pte는 process마다 하나씩 있으니까.
  - less 함수 구현: hash의 element가 들어있는 구조체의 비교를 가능하게함.
  - destructor 함수 구현: sturct hash에서 struct hash_elem을 빼주는 거랑 할당해제하기랑은 별개다. 둘 다 하자.

- **sup page table 사용**
  1. sup page table은 page table이 activate 된 직후에 초기화한다.(초기화는 load 함수에서 load_segment 하기 이전.)
  2. load_segment 함수에서 실제로 일하는 부분을 spt에 관련 정보를 기록하는 것으로 대체.(이게 조교문서에서 'just pages are allocated'라는 표현의 의미인듯) hash_insert 해준다. 더 효율적 구현을 위해 아직 물리 메모리를 할당하지 않음.
  3. 실제로 메모리에 로딩되는 부분은 page_fault 함수에서 구현된다.
     - frame을 allocate 할 때마다(frame을 할당한다는게 그에 맞는 가상주소가 생긴다는 의미니까.) frame table entry 뿐만아니라 spt entry도 만들어줘야한다. hash_insert 해준다.
- **page_fault 함수**

  sup page table은 프로세스마다 존재하고, 프로세스가 '아~ 이런 페이지 있어야하지!' 하는 소망들을 저장해둔다. 근데 그게 실제로 메모리에 frame이 할당되어서 page가 '존재'할 수도 있고 frame이 할당되지 않아서 spt에 '프로세스의 소망'만 남아있을 수도 있다. 프로세스의 소망은 page fault를 통해서 이루어진다. page fault 함수는 MMU의 자극에 의해 호출된다. OS를 구현하는 우리가 할 일은 page_fault 함수를 구현하는 것이다. 이를 통해 프로세스의 소망인 '물리메모리에 frame 잡아주기'를 한다.

  page_fault 함수가 호출되기 '이전에' page fault시 필요한 동작을 기록해둬야한다. 이러한 기록을 할 때 사용되는 것이 sup page table이다. 특정 가상주소에 대해 특정 동작을 하도록 sup page table에 정보를 기록해둔다. page_fault함수는 그에 맞게 동작하게끔 만들어야한다.

  - page_fault 함수를 호출한 가상주소를 찾는다.
  - 가상주소로부터 spt의 entry를 찾는다.
  - spt에 없는 주소라면 exit을 하는 것이 맞겠으나 stack을 늘려야하는 상황일 수도 있기 때문에 stack growth를 수행해야하는지 확인해보고 가능하다면 유효한 page fault로 판단한다. 만약 stack growth가 불가능하다면 exit(-1)을 한다.
  - spt에 있는 정보(지금 이 메모리로 올릴 page가 어디에 존재하는가.)를 가지고…
    1. 메모리에 있으면? → 일어나서는 안되는데… 일어날 수도 있나? 아무일을 하지 않도록 한다.
    2. file에 있으면?(lazy를 끝내고 loading에서 부분.) → frame을 할당하고, frame 테이블에 정보 기록하고, 파일을 읽자.
    3. 파일의 특정 부분을 읽어와야하므로 파일을 읽을때는 file_seek하고 file_read한다.
  - swap disk에 있으면? → frame을 할당하고, frame 테이블에 정보 기록하고, swap disk에서 메모리로 가지고오자.
  - spt에 페이지 위치 정보를 바꿔주기. + 메모리로 올린 페이지에 대한 주소 해석위해 pagedir_set_page

43.p의 3번을 해결하기 위해 page.c 에서 allocate_frame 함수를 보자 → 그러려면 write back에 대한 정보가 있어야하는데, 이는 page별로 있다. (evict는 페이지가 물리 메모리에서 사라지는 거니까.)→ spte에 write back 정보를 추가하고, fte에서 spte를 가리키게 하자. ⇒ 그러면… allocate_frame이 완성된다. 연결해서 남는게… → swap disk해야함.

sup page table은 per thread이고 pintos는 single thread processd이니까 process가 죽을때 spt처리함수 만들기

Stack Growth

Project 2에서는 physical memory에 할당 가능한 stack segment의 크기를 4KB의 크기로 고정하였다. 4KB 보다 큰 stack을 사용하기 위해 stack growth를 다음과 같이 구현하였다.

page_fault 함수에서 내부 fault_addr에 해당하는 sup_page_table_entry가 현재 thread의 sup_page_table에 존재하지 않는 경우에 fault_addr이 valid한 범위에 해당하는 경우 stack 영역을 늘려주도록 구현했다. extended stack 주소에 대한 접근이 page fault가 아닌 system call 안에서도 일어날 수 있기 때문에 이에 대한 처리도 해주었다. system call의 check_address, check_writable, check_readable 함수들에서도 필요한 경우 stack 영역을 늘려주도록 구현했다.
stack growth 기능을 구현하는 함수는 decide_stack_growth_and_do_if_needed 함수로 syscall.c에 구현해두었다. decide_stack_growth_and_do_if_needed 함수를 설명하자면, 인자로 esp와 addr를 받고 stack growth가 잘 마무리되면 true, 잘 안되면 false를 반환한다. esp를 인자로 받아올 필요가 있는 이유는 esp를 사용해서 해당 address를 stack growth로 처리할지 아니면 잘못된 접근으로 생각할 지 정해주어야 하기 때문이다. 그래서 decide_stack_growth_and_do_if_needed 함수는 가장 먼저 addr < esp – 32 인지 확인하도록 되어있다. 만약 addr이 esp - 32 보다 메모리 공간 상에서 더 떨어진 곳에 존재한다면 잘못된 접근으로 간주하고 false를 반환하도록 했다. 다음으로는 PHYS_BASE에서 addr를 뺀 값의 크기가 8MB 보다 큰 지 확인하도록 했다. Pintos 공식 문서에 따르면 stack은 커질 수 있는 최대 크기가 8MB이기 때문에 이보다 크면 false를 반환하도록 했다. 이후 새로운 sup_page_table_entry, frame_table_entry를 할당하고 각 table에 추가한 뒤, pagedir_set_page를 통해 이후 접근에는 page fault가 발생하지 않고 접근할 수 있도록 했다. 만약 alloc_page_frame을 통한 frame 할당이나, page_dir_set_page가 잘 되지 않는다면 역시 false를 반환하도록 했다.

vm/syscall.c : decide_stack_growth_and_do_if_needed(void *_esp_, void *_addr_)

stack growth 기능을 수행하는 함수다. 스택 성장 필요성 판단을 함수는 주어진 주소 (addr)가 esp보다 32 바이트 이상 아래에 있는지 확인해서 한다. 그 다음에는 주소가 유효한지를 주어진 주소가 물리적 주소 공간의 한계를 넘지 않는지 확인해서 검증한다. 스택 성장이 필요한 경우, 새로운 스택 페이지를 위한 sup_page_table_entry가 생성된다. 그 다음에는 새 stack page를 위한 physical memory frame이 할당되고, 할당된 frame에 대한 정보를 저장하는 frame_table_entry가 생성된다. 마지막으로 install_page 함수를 사용하여 새로 할당된 프레임을 user space의 가상 주소에 매핑한다.

vm/syscall.c : allocate_frame_for_syscall_if_needed(struct *_hash_elem *to_find_*, void *_virtual_page_addr_)

이 함수는 시스템 호출 중에 필요한 경우 프레임을 할당하는 데 사용된다. 주요 목적은 가상 페이지 주소에 대응하는 물리적 메모리 프레임을 할당하고, 해당 페이지를 메모리로 로드하는 것이다. sup_page_table을 검색해서 해당하는 hash 요소를 가진 entry를 찾는다. 현재 thread의 페이지 디렉토리에서 주어진 가상 페이지 주소에 대응하는 페이지가 없는지 pagedir_get_page 함수를 사용하여 확인한다. 페이지가 메모리에 없는 경우 새로운 프레임을 할당하고, frame_table_entry를 생성한다. 페이지가 파일 시스템에 있는 경우에는, 파일 시스템에서 해당 페이지를 읽어 할당된 프레임에 로드한다. 페이지가 스왑 디스크에 있는 경우 스왑 영역에서 해당 페이지를 읽어 할당된 프레임에 로드한다. 할당된 frame에 대한 정보를 frame

table에 추가한다. 마지막으로는 할당된 frame을 current thread의 page directory에 설정한다. 이 과정에서 실패하면 할당된 자원을 해제하고 false를 반환하고, 모든 작업이 성공적으로 완료되면 true를 반환한다.

지난 프로젝트에서 구현한 check_address 함수는 주소가 page directory에 존재하지 않으면 항상 비정상적인 접근으로 취급하였다. 하지만 stack growth의 경우와 lazy loading의 경우 때문에 page directory에 가상 주소가 없다고 하더라도 비정상적인 접근이 아니게 될 수 있다. 이에 대한 처리를 위해 allocate_frame_for_syscall_if_needed 함수와 decide_stack_growth_and_do_if_needed 함수를 만들었다. mmu는 page fault를 통하여 OS로부터 서비스를 받는데, OS 자신이 OS에게 제공하는 서비스로써 이 두 함수를 해석할 수도 있다.

Swap Table

Frame table에서 말했듯이, physical memory에서 더 이상 available한 frame이 없는 경우, 일부 page를 swap 공간으로 빼내고 새로운 page를 넣도록 해야한다. 이때 이러한 Swap 공간을 효율적으로 관리하기 위해 사용하는 것이 swap table인데, 우리는 이를 bitmap 형태로 관리할 수 있도록 bitmap swap_disk_bitmap을 만들어 이용하였다. Physical memory에서 page를 제거하고 swap disk에 넣는 swap_out, swap disk의 page를 new frame에 넣는 swap_in을 구현했다.

swap_in이 호출되는 곳은 exception.c의 page_fault 함수에서 current_page_location이 InSwapDisk인 경우와, syscall.c의 allocate_frame_for_syscall_if_needed에서 current_page_location이 InSwapDisk인 경우가 전부이다. swap_out 함수는 frame.c의 handle_frame_eviction에서 수행되는 eviction 과정에서 호출되는 것이 전부이다.

페이지를 단위로 swap disk를 다루기 때문에 512바이트를 단위로 다루어지는 disk를 다루기 쉽게 하기 위해 4096바이트를 512바이트로 나눈 값인 8을 상수로 정의하여 구현하였다. 그리고 swap disk는 kernel pool과 user pool과 같이 bitmap으로 다루어진다.

vm/swap.h : void swap_disk_init(void)

swap_disk_bitmap을 초기화하는 함수다. bitmap_create 함수를 사용하여 전역 변수인 swap_disk_bitmap가 bitmap을 가르키도록 초기화한다. 그 다음에는 bitmap_set_all 함수를 사용해서 모든 항목을 false로 설정하도록 했다. false인 경우 swap disk에 공간이 있는 것을 의미하도록 설정했기 때문이다.

vm/swap.h : void swap_in(void *, size_t)

swap disk의 page를 new frame에 넣는 함수다. swap_disk_bitmap에서 frame에 해당하는 idx의 bit를 bitmap_flip 함수를 사용해서 false로 표시하도록 구현했다. 그 다음에는 block_read를 호출하여 실제로 swap disk의 page를 메모리로 옮기는 작업을 수행한다.

vm/swap.h : size_t swap_out(void *)

Physical memory에서 page를 제거하고 swap disk에 넣는 함수다. bitmap_scan_and_flip 함수를 사용해서 swap_disk_bitmap에서 frame에 해당하는 idx의 bit를 true로 표시하도록 구현했다. 그 다음에는 block_write를 호출하여 physical memory에서 page를 제거하고 swap disk에 넣는 과정을 수행시켰다.

File Memory Mapping

thread별로 mmap파일을 기록해두는 list와 매핑 번호를 위한 정수 하나가 필요하다.

- **memory mapped file을 위한 구조체**
  - mapping id
  - mapping되는 page 주소
  - 사용되는 page 수
- **mmap 함수**
  - mmap매핑되는 주소와 파일포인터에 대한 검사를 한다.
    - page aligned 된 주소에 mapping을 하는지, 주소 0이 아닌지, file descriptor가 0과 1이 아닌지, 파일의 크기가 0이 아닌지
  - file_reopen 함수를 사용하여 파일을 읽고, load_segment 함수를 수정한 것과 같은 논리로 sup page table에 원소를 추가하여 lazy 하게 loading 되게 한다.
  - 만약에 lazy loading 중 파일이 사용하는 가상주소가 기존에 사용하던 가상주소라면 지금까지 mapping할 때 사용한 sup page table entry를 모두 없애고 -1 리턴.(lazy loading이니까 frame에는 아직 안올라갔을것. 그래서 spt에서만 빼면됨. - syscall은 중간에 멈추지 않음.)
- **munmap 함수**
  - thread 별로 있는 mmap_file_list를 순회하여 인자로 들어온 mapping과 같은 mapping_id를 가진 struct mmap_file을 찾는다.
  - 해당 구조체에는 파일이 사용하고 있는 가상 주소 정보가 담겨있는데, 이를 활용하여 sup page table의 원소를 찾고 해당 page가 사용하고 있는 frame에 대응되는 kernel virtual address를 찾는다.
    - frame에 대한 소유를 현재 thread가 하고 있지 않거나, 위에서 찾은 주소가 frame table에 기록된 ‘frame과 관련된 page 주소’와 다르면 넘어가는 식으로 frame table을 순회한다.
  - pagedir_is_dirty 함수를 활용하여 page에 write 연산이 있었음을 확인하고 그렇다면 정보의 손실을 막기 위해 다시 파일로 써준다.
  - 이 과정을 파일이 사용하고 있는 page 모두에 대해서 진행한다.
  - 이 함수에서 sup page table entry를 할당해제하는 역할을 한다.

추가

- 지난번 프로젝트의 system call 디버깅 과정에서 create, remove, open, read, write system call에 인자로 포인터가 들어가기 때문에 추가적인 주소 검증이 필요함을 깨달았다. 이번 프로젝트에도 이와 유사한 검증이 필요했다. paging을 구현하였기 때문에 read와 write system call을 할 때 인자로 들어오는 buffer를 위한 검증이 필요하다.
- page가 사용하는 virtual address에 대응하는 frame의 주소를 찾을 때, frame table을 순회하며 찾자.
- hash table의 원소를 찾을 때 동적할당을 한 원소를 통해 찾도록 한다.
- 특정 데이터를 다루기 위한 할당을 했으면, 할당 해제도 해줘야한다.

- process가 exit할 때 memory mapped된 파일들 처리, prcoess가 사용중이던 frame 할당 해제 처리를 한다. 그리고 process마다 하나씩 있는 데이터 중 하나인 sup page table에 대한 할당해제도 진행한다.