Lab3 Design Report_

Member #1
Name: Suwon Yoon
Student ID: 20210527

Member #2
Name: Jiwon Lee
Student ID: 20210706

1. Frame Table

1.1 Basics the definition or concept, implementations in original pintos (if yes)

Page

virtual page라고도 불리며, 연속된 4096 바이트 크기의 가상메모리 영역이다. 페이지는 페이지 크기로 나눠지는 시작 주소를 가지도록 설정되어 있다. 이를 page-aligned 되었다고 한다. 페이지의 크기는 4096 바이트이고 4096은 2^12 이기 때문에 12 비트만 주어진다면, 페이지에 있는 모든 주소를 1바이트 단위로 가리킬 수 있다. 우리가 Pintos에서 사용하는 아키텍쳐 상에서는 주소는 32 비트이고 상위 20 비트의 page number와 하위 12 비트의 page offest으로 주소를 나타내는 비트를 쪼개어 해석할 수 있다. 상위 20 비트는 수 많은 페이지들 중에서 하나를 가르키는 page number의 역할을 하고, 하위 12 비트는 페이지 안에서의 위치를 나타내는데 사용된다.

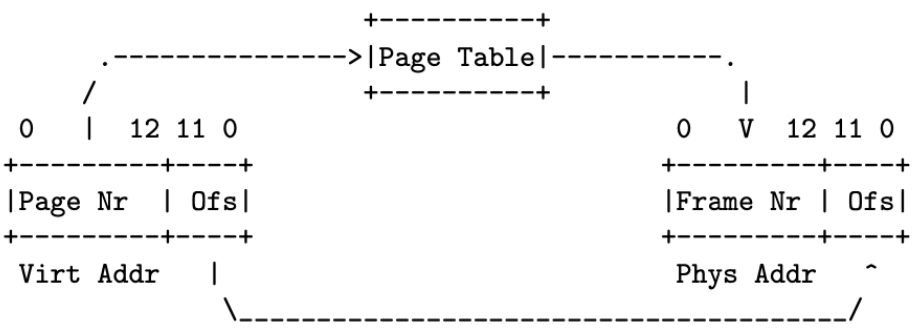
모든 프로세스는 독립적인 user page의 집합을 가지고 있다. 이 user page들은 전부 PHYS_BASE (0xc0000000) 보다 작은 virtual address를 가진다.

Frame

physical frame, page frame이라고도 불리며, 연속된 4096 바이트 크기의 physical memory 영역이다. 페이지와 마찬가지로 page-aligned이기 때문에, 20 비트의 frame number, 12 비트의 frame offest으로 주소의 역할을 구분하여 나눌 수 있다.

Page Table

Pintos에서 page table은 CPU가 가상 주소를 물리 주소로, 즉 page에서 frame으로 변환할 때 사용하는 데이터 구조다. 이 페이지 테이블에 대한 코드는 pagedir.c에서 확인할 수 있다.



위 그림은 페이지와 프레임 간의 관계를 보여준다. 왼쪽의 가상 주소는 page number와 offset으로 구성되고, 페이지 테이블은 이 page number를 frame number로 변환한 다음 수정되지 않은 오프셋과 결합하여 오른쪽의 물리적 주소를 얻는다.

Frame Table

Frame Table은 각 frame에 대한 정보를 entry로 저장하는 표이다. Frame Table이 필요한 이유는, page load 상황에서 다른 process에 의해 사용되고 있지 않는 frame을 찾아 사용하기 위해서다.

Pintos의 현재 구현은 threads/palloc.c에 정의된 palloc_get_page()를 사용하여 프레임을 할당한다.

Physical memory는 kernel pool과 user pool이라는 두 개의 pool로 나뉜다. user pool로부터 frame을 할당할때는 palloc_get_paged(PAL_USER)를 사용한다. kernel pool은 user pool과 독립되어 있어서 kernel data와 kernel code는 physical memory에서 거의 언제나 존재하는데에 반해 user pool은 계속 추가되면 physical memory에 사용가능한 frame이 부족할 수 있다. physical memory에 사용 가능한 frame이 없다면 palloc_get_page()는 null pointer를 반환한다.
frame table은 user page가 들어있는 frame에 대한 정보를 entry로 가져야한다. frame table의 각 entry에는 현재 해당 frame을 차지하고 있는 page에 대한 포인터를 담을 것이다. 이러한 frame table은 Pintos가 free frame이 없을 때 evict할 page를 선택하는 eviction 과정의 구현을 위해서 필요하다.

우리는 eviction 과정을 구현하는데 있어서 필요한 replacement algorithm에 clock algorithm을 사용하기로 했다.

Clock Algorithm

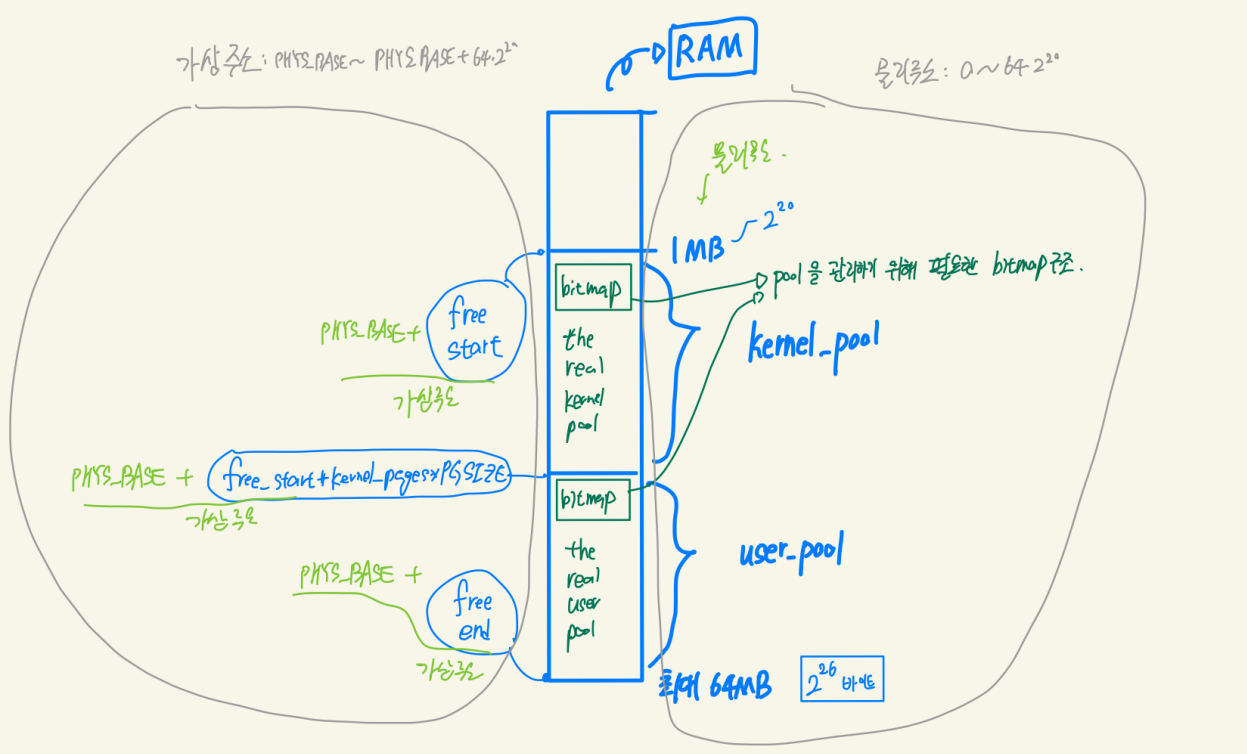
Page들을 가르키는 entry들을 circular queue에 저장하여 최근에 사용되지 않은 page를 evict 하는 알고리즘으로, LRU를 완벽하지는 않지만 어느 정도 실현하는 알고리즘이다. 수행과정을 요약하자면, queue를 순회하면서 access bit를 확인한다. 1이면 0으로 바꿔주고 0이면 queue를 한바퀴 순회하는 동안 한번도 프로세스가 사용하지 않은 page인 것이기에 queue에서 제거하고 evict한다. 이 circular queue를 계속 순회하면서 evict할 page가 나올때까지 돌아가는 알고리즘이다.

Current Implementation

Virtual Memory를 지원하고 있지 않기 때문에 frame table은 구현되어 있지 않다. 현재 구현에서 기본이 되는 palloc.c와 pagedir.c에서 중요 부분을 분석 해보자.

메모리 공간은 user pool과 kernel pool로 나누어져 있으며 system의 RAM을 절반씩 차지한다. struct pool 을 통해 이를 관리한다. page 단위의 관리를 위해서 struct bitmap * 를 멤버로 포함한다.

palloc_init 함수에서는 free page의 개수를 정하고 user pool과 kernel pool이 free page 개수를 절반씩 차지하게 한다. init_pool 함수가 각각의 pool에 대해서 사용되어 아래 그림과 같은 상태가 될 것이다. 각각의 pool에 pool을 다루기 위해 필요한 자료구조인 bitmap을 저장하고 있음을 알 수 있다. ptov 함수를 활용해서 가상주소와 물리주소의 간단한 매핑관계가 있음을 알 수 있다. 여기서 kernel의 가상주소와 물리주소는 PHYS_BASE 만큼 차이난다는 것을 엿볼 수 있다.



palloc_get_multiple 함수는 user pool과 kernel pool을 '선택'하고 할당받은 페이지 수 만큼을 bitmap에 '표시'한다. bitmap으로 페이지를 다루기 때문에 이 표시가 곧 할당되었음을 나타낸다. 그 후에 memset 함수를 통해 할당한 영역을 0으로 채운다. pool의 base에 저장되는 주소는 RAM 상의 물리주소에 PHYS_BASE 만큼을 더한 가상주소(kernel virtual address)이다. 이 가상주소에 방금 정한 '사용할 페이지의 인덱스(번호)'*4KB를 하면 할당된 공간의 시작 주소(kernel virtual address임)를 알 수 있다. palloc_get_page 함수는 palloc_get_multiple 함수의 특수 케이스이다.

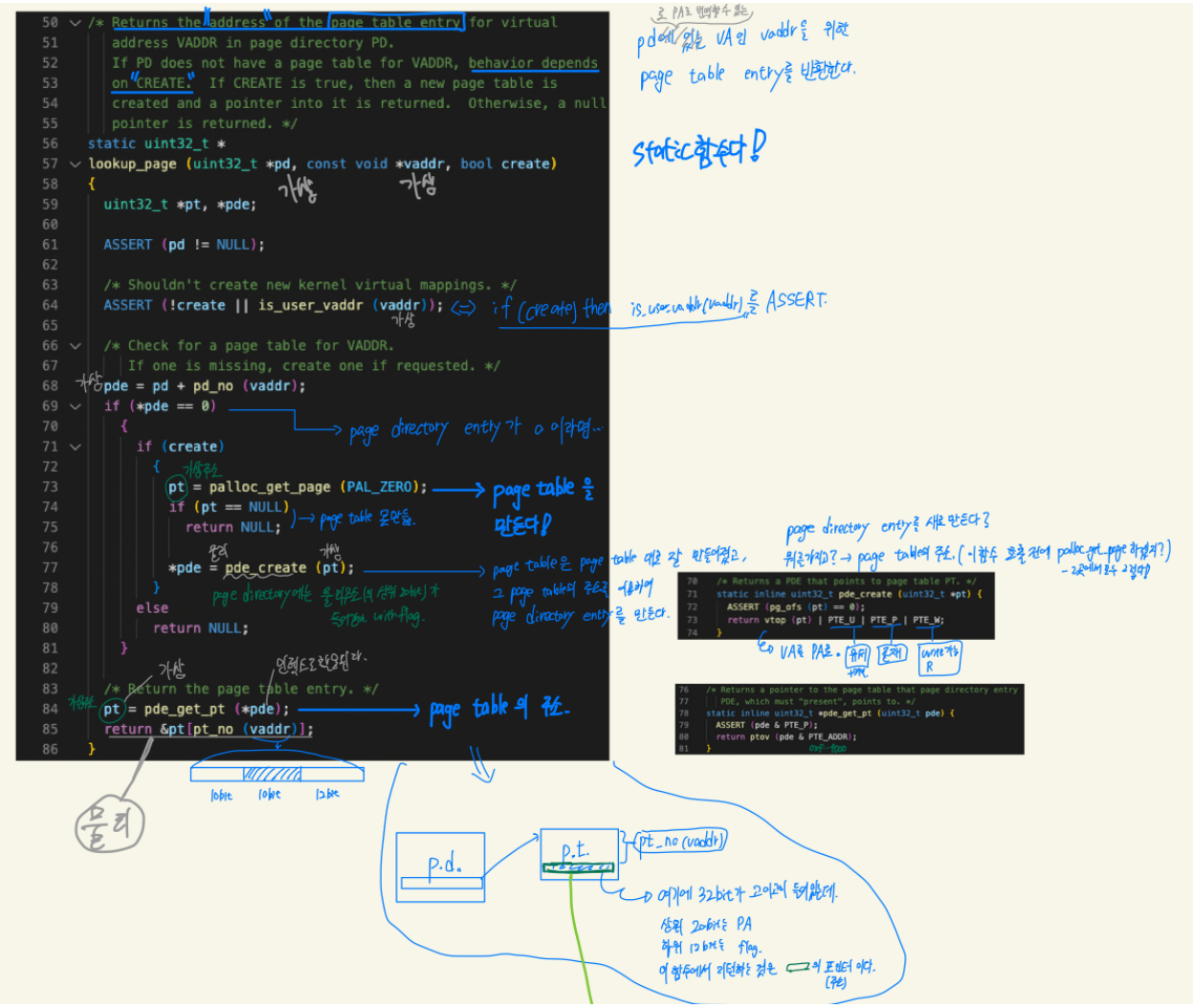
palloc_free_multiple 함수 역시 user pool과 kernel pool을 선택하고 할당해제(사실은 표시)할 index를 찾는다. 그리고 memset으로 0xcc로 값을 채움으로써 초기화한다. 초기화했다고 bitmap에 표시도 해준다.

pagedir_create 함수에서는 kernel pool에 page를 할당하고 (이 page는 page directory가 저장됨)에 init_page_dir에 있는 값을 복사한다. 즉 새로 만드는 page directory는 init_page_dir과 같은 값이다.

init_page_dir은 paging_init 함수에서 만들어진다. 이 함수에서는 init_page_dir을 위한 페이지를 kernel pool에 할당하고 할당한 영역을 0으로 초기화한다. 구체적인 동작은 아래의 필기를 확인하면 된다.

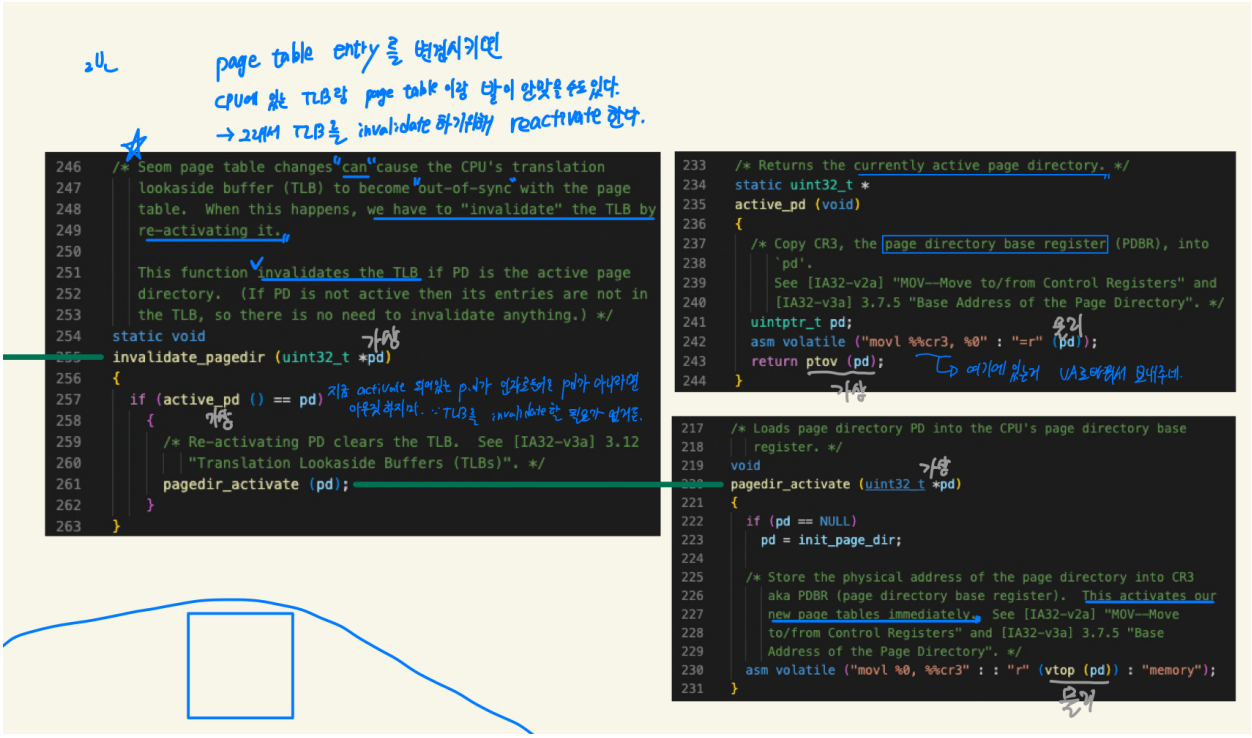
page directory나 page table의 entry는 모두 물리주소 변환하여 값을 저장한다. 그래야 cpu의 MMU가 정상 작동한다.

다음으로는 page directory관련 기능을 구현할 때 사용하는 lookup_page 함수이다. lookup_page는 page table entry의 주소를 반환한다. 추후에 다른 함수에서 해당 주소를 활용하여 pte를 변경하기도 하고 역참조하여 사용하기도 한다. lookup_page에 대한 자세한 설명은 아래의 필기를 확인하면 된다.

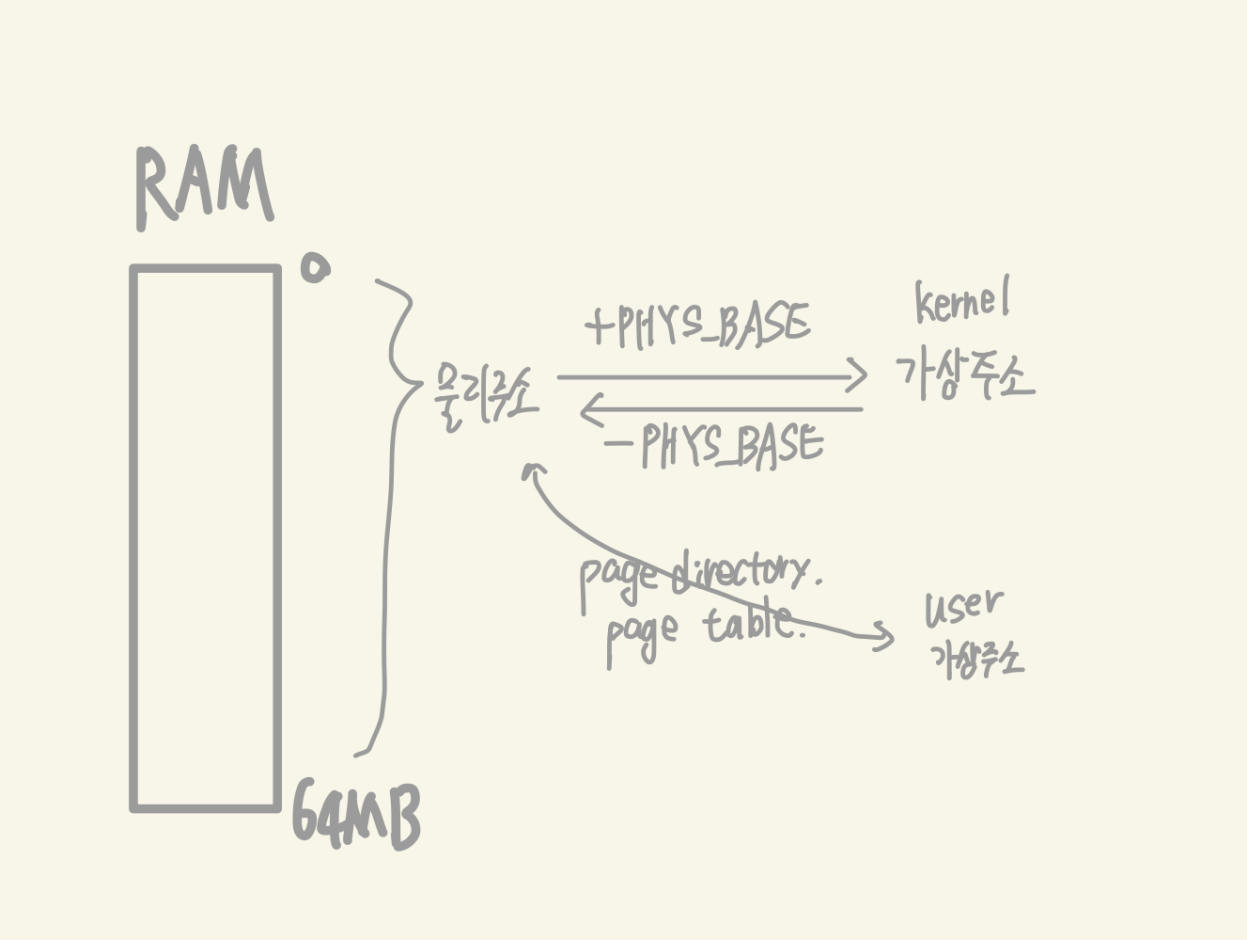


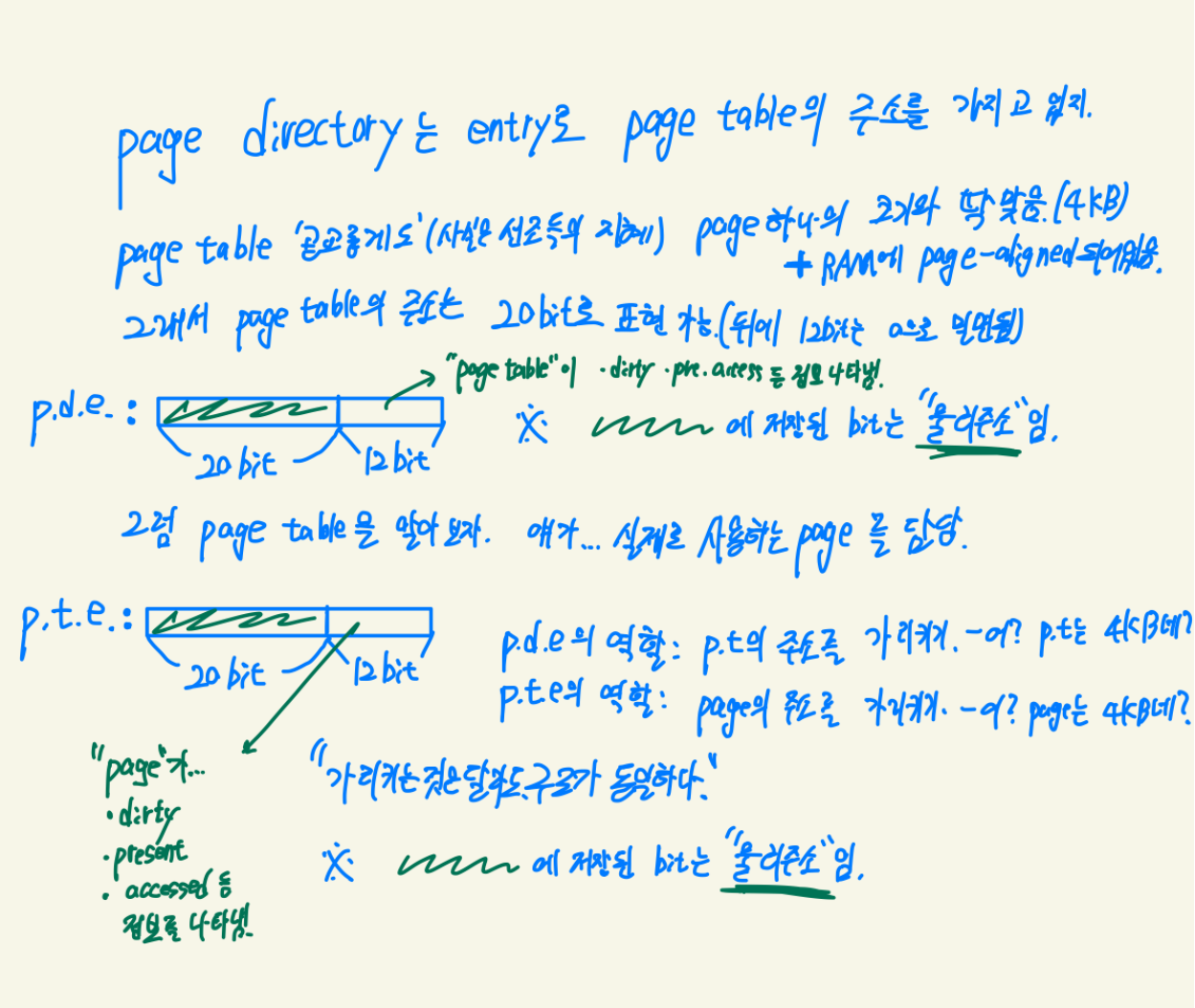
lookup_page 함수를 사용하는 하나의 예시로 pagedir_set_page 함수가 있다. 이 함수에서는 lookup_page 함수로 받아온 page table entry 주소를 활용하여 pte를 새로 할당한 physical frame의 주소(인자로 들어올 때는 kernel virtual address이고 pte에 저장할 때는 kernel physical address로 변환)로 설정한다.

pagedir_clear_page 함수도 살펴볼 필요가 있다. 이 함수는 user virtual page를 “not present”로 바꾼다. 그래서 해당 페이지로의 다음번 접근은 page fault이다. 이 함수의 올바른 동작을 위해서 아래의 함수가 사용된다.



아래는 palloc.c와 pagedir.c 를 보면서 정리한 개념이다.


RAM
0
64MB
물리주소
+PHYS_BASE
-PHYS_BASE
kernel
가상주소
page directory.
page table.
user
가상주소



1.2 Limitations and Necessity : the problem of original pintos, the benefit of implementation

프로세스별로 virtual memory를 physical memory로 매핑하게 하여 여러 프로세스가 실행될 때 각각의 프로세스가 마치 전체 메모리 영역을 가지고 있는듯한 환상을 줄 것이다. 현재는 이러한 구현이 아니라 물리 주소를 거의 그대로 사용하고 있는 것과 같다. 이런 방식은 여러 프로세스를 실행하는데 걸림돌이 될 수 있다. frame table을 사용하여 frame을 효과적으로 관리할 것이다.

1.3 Blueprint : how to implement it (detailed data structures and pseudo codes)

일단 frame의 정보를 저장하는 자료구조가 필요하다. 이는 그냥 각종 변수들의 모음이기 때문에 struct로 충분하다. 이를 struct frame_entry로 일단 이름 짓자. 각 frame_entry는 frame의

physical address(frame number), frame의 page를 가리키는 virtual address(thread 포인터 정보가 담겨있으면 필요 page dircetory에서 직접 보면되니까 없어도 될수도 있다.), frame을 사용하고 있는 중인 thread를 가리키는 포인터가 필요하다. 상황에 따라 추가될 수 있다.

그 다음에 이 frame_entry들을 모아서 관리할 자료구조가 필요하다. 지금까지 써왔던 list 자료구조를 사용하여 frame table역할을 할 frame_table을 만들 것이다. frame_table의 entry로 frame_entry를 사용하기 위해서는 list_elem 변수를 struct frame entry의 멤버로 추가해야한다. 이 frame_table은 프로세스 별로 할당되는 것이 아니기 때문에 전역으로 정의되어야하며, init.c의 초기화 과정에서 초기화할 것이다. 그리고 frame table에 대한 mutual exclusion을 위해서 lock을 정의해줄 것이다.

frame table과 관련된 기본적인 자료구조의 정의를 마쳤으니 기능적으로는 page에 대응되는 frame을 할당하는 함수를 만들어야한다. 이 함수를 alloc_frame으로 이름 짓도록 하자. alloc_frame은 palloc_get_page함수를 사용해서 frame을 할당할 것이다. (frame을 user pool에서 고를지, kernel pool에서 고를지 판단도 이 함수가 함)만약 physical memory의 공간이 부족하다면, palloc_get_page 함수는 할당에 실패할 것이고 null을 반환할 것이다. null이 반환되었다는 것은 free frame이 부족하다는 것이니 eviction을 진행해야되고, null이 아닌 것이 반환되었다면 frame을 잘 만든 것이니 별다른 처리를 할 것이 없을 것이다. eviction을 할 때는 해당 page를 clear하고 free 하는 등의 처리를 해줄 것이다. frame이 할당 되었다면, 그냥 바로 palloc_get_page 함수의 반환값을 return하는 것으로 할당된 frame의 주소를 이 함수의 caller로 알릴 것이다. caller는 frame_entry를 구성하고 frame table에 frame_entry를 넣을 의무가 있다. 정리하자면 alloc_frame 함수는 eviction 기능이 추가된 palloc_get_page 함수로 사용될 것이다. frame을 할당하는 함수가 있으니 해제하는 함수도 만들어야한다.

그 전에 위에서 언급한 eviction에서 필요한 Clock Algorithm 관련한 부분을 구현하자. Clock Algorithm은 일단 circular queue가 필요하다. 우리의 구현에서는 앞서 말한 list형태의 frame_table을 사용하는데, 실제 circular queue 자료구조를 만드는 것보다 이미 구현되어있는 list 자료구조를 사용하여 circular queue와 똑같이 기능할 수 있게 만들 것이다. circular queue의 기능을 구현하기 위해서는 그냥 순회할 때 사용하는 포인터가 리스트의 끝에 도달했다면, list_begin함수를 사용해 다시 리스트의 처음으로 포인터를 보내면 간단하게 구현이 가능할 것이다. 그 다음에는 이제 clock algorithm의 핵심이 되는 accessed bit 확인 과정을 구현해야한다. 이는 userprog/pagedir.c에 위치한 pagedir_is_accessed 함수를 사용하여 구현할 것이다. pagedir_is_accessed 함수는 page directory number를 받는데 이를 사용해서 accessed 여부를 반환해준다. 그렇기에 우리는 이 함수를 사용해서 frame_entry들의 accessed bit여부를 확인할 것이다. bit의 값이 1이면 0으로 flip을 시켜줄 것이고, 0이면 그 frame을 evict할 것이다. evict할 frame이 선정될 때까지 circular queue를 순회할 것이고, 선정이 되면 바로 순회를 멈추도록 구현할 것이다. 위에서처럼 eviction을 구현하고 나면, 새로운 frame을 할당할 수 있게된

다. 문서에서는 이부분을 마지막에 구현하라고 하였지만 동시에 구현해도 문제 없을 것 같아 이렇게 진행하기로 하였다.

마지막으로 frame table의 구현을 완성하려면 frame이 할당 해제되는 경우를 구현해야한다. Thread가 어떤 사유로든 종료가 되었다면, thread가 사용하고 있었던 모든 frame과 frame_entry를 할당 해제해야한다. 이는 while문을 사용하여 종료된 thread의 포인터를 멤버로 가지는 entry들을 전부 찾아내어 할당 해제하면 될 것이다. 이 때 가장 먼저 frame_table에서 entry를 제거한 다음 pagedir_clear_page 함수로 page table entry를 invalidate하여 모든 사후 접근을 fault가 일어나게 하고, palloc_free_page 함수를 사용하여 frame을 할당해제한 다음, free 함수로 frame_entry를 할당해제하는 과정을 거칠 것이다. 필요한 경우에는 page를 file system 또는 swap disk로 write 하는 기능도 만들 것 같다.

추가하자면 user pool과 kernel pool에 대한 구별에 신경 써야할 것이다. 위에 그려둔 그림을 보면 user pool과 kernel pool은 각각 RAM에 실제하는 영역이다. 여기다가 frame을 할당시킬 것이다. user process가 사용하는 frame은 user pool에, 그 밖의 것들은 kernel pool에 할당하는 식이다. 관련해서 헷갈릴 수 있는 명제가 있다. user virtual address와 kernel virtual address관련된 내용인데 “kernel virtual address로 user pool에 접근할 수 있다.”는 것이다. user page는 user pool에 할당되는 frame 을 사용할 것이다. 따라서 user virtual address로 user pool에 있는 frame에 접근할 수 있다. kernel virtual address는 paging_init에서 만드는 init_page_dir을 사용하여 physical address로 번역할 수 있는데, 동작을 보면 user pool과 kernel pool 모두에 대한 주소 번역을 가능하게 한다. 따라서 kernel virtual address로 user pool에 접근할 수 있다. 관련된 이야기가 문서 4.1.5.1에 나온다. 모든 user virtual page는 kernel virtual page와 aliase 관계를 가진다고 표현되어 있는데 위의 이유에서이다. CPU는 page table entry의 access bit이나 dirty bit을 1로 만든다. 이는 OS가 하는 것이 아니라 CPU가 동작할 때, OS입장에서 자동적으로 일어나는 일이다. OS는 이런 bit들을 0으로 만들어야한다. aliase 관련된 처리도 해야할 수 있음을 인지해둬야겠다.

2. Lazy Loading

2.1 Basics the definition or concept, implementations in original pintos (if yes)

Lazy Loading

Lazy Loading은 memory management 및 전반적인 효율을 높이기 위한 기술이다. 이 개념의 핵심은 process가 실제로 해당 메모리를 필요로 할 때까지 loading을 지연시키는 것이다. Lazy Loading은 특히 돌아가는 프로그램들에 비해 메모리가 부족한 시스템이나 multiprocess

environment일때 유용하다. 시스템의 전반적인 반응 시간과 성능을 향상시키는 데 중요한 역할을 한다.

Current Implementation

load 함수

프로젝트 2에서 load 함수의 호출과 관련된 전후 상황에 대해 살펴보았다. 간략하게 되짚어보자면 process_execute 함수에서 thread_create 함수로 start_process 함수가 실행될 수 있게 한다. start_process 함수에서는 load 함수로 executable file을 읽어온다. 지난 프로젝트에서는 이 부분에 대한 디테일은 필요없었으나 이번 프로젝트의 lazy loading을 구현하기 위해서는 이 부분을 수정해야한다. 따라서 load 함수를 살펴보겠다.

load 함수는 프로세스의 초기과정을 담당하는 함수이다. 그 초기과정은 pagedir_create 함수로 page directory를 생성하고 process_activate 함수로 방금 생성한 page directory를 활성화 시키는 것으로 시작된다.

- 이 과정에서 tss_update 함수가 호출되는데 이는 thread_create 함수에서 thread를 위한 page를 하나 할당하며 생기는 kernel stack을 interrupt 가 발생하였을 때 사용할 수 있도록 설정한다. user code가 실행되며 interrupt가 발생할 때 user stack에서 처리하면 안되기 때문이다.

이후 filesys_open 함수를 통해 파일을 열고 executable 파일에 적힌 header 정보를 struct Elf32_Ehdr 타입을 가진 ehdr 변수로 읽어낸다. 이때 header가 약속된 형태가 아닌 경우 load 함수는 실패한다. header에 대한 검증을 마친 이후에는 program headers를 읽을 차례이다.

load 함수에서는 아래의 과정을 edhr에 있는 정보인 e_phnum만큼 반복한다.

ehdr의 e_phoff 값을 file_ofs로 설정하고 file_ofs부터 유용한 정보를 읽는다. 그렇게 읽은 값은 struct Elf32_Phdr 타입을 가지는 phdr 변수에 저장되는데 그 변수의 멤버변수인 p_type에 대한 분류를 switch_case 문으로 한다. 그 중 PT_LOAD인 경우에는 validate_segment 함수를 통해 file_read로 읽어온 정보(phdr)가 올바른지 검증한다.

검증을 통과한 경우에는 disk에서 memory로 해당 파일의 일부를 읽어온다. 이때 사용되는 함수가 load_segment 함수이다. load_segment 함수를 실행하기에 앞서 PGMASK(상위 20bit는 0 하위 12bit는 1)를 활용하여 load_segment 함수의 인자로 사용될 값들을 설정한다.

load_segment 함수는 아래의 과정을 읽어와야할 모든 바이트를 읽을때 까지 반복한다.

1. palloc_get_page로 user pool에 page(frame)를 할당
2. file_read 함수로 할당받은 공간에 file에 있는 값을 읽어온다. 그리고 page의 남는 부분은 0으로 설정

3. install_page 함수를 통해 방금 읽어온 segment에 대한 address mapping을 추가
    - install_page 함수는 page directory에 user virtual address 정보(phdr의 p_vaddr 값이 전달됨.)를 추가한다. 이 함수 덕분에 user virtual address를 translate 하면 physical address가 나오게 된다. 중간 과정을 더 설명해보자면, 곧바로 physical address가 page table entry로 저장되는 것이 아니다. install_page 함수의 추상화 레벨에서는 user virtual address와 kernel virtual address의 매핑을 만드는 것이며, 한 단계 내려가보면 pagedir_set_page 함수에서 호출되는 gte_create_user 함수에서 vtop 함수를 활용하여 page table entry에 physical address로 저장한다.

추가로 file_ofs가 file의 길이보다 긴 경우이거나 file_read 함수에서 읽은 phdr 값의 크기가 예상한 값과 다른 경우에는 load 함수는 실패한다. 이는 validate_segment 함수가 false를 리턴했을 때도 마찬가지이다.

page_fault 함수

page fault가 일어나면 항상 잘못된 것으로 간주하고 kill 함수를 통해 thread_exit 함수를 호출하여 page_fault를 호출시 실행되는 thread를 제거한다.

2.2 Limitations and Necessity : the problem of original pintos, the benefit of implementation

현재 구현에서는 load 함수에서 ehdr.e_phnum 번을 반복하며 load_segment 함수를 호출한다. 이때 load_segment 함수는 file_read 함수를 활용하여 disk에서 memory로 executable file의 segment를 모두 load한다. file_read 함수는 inode_read_at 함수를 통해 구현되는데 inode_read_at 함수는 size 바이트만큼을 모두 읽을때까지 반복하는 while문으로 구성되어 있다. 즉 lazy loading이 구현되어 있지 않고 process가 시작할 때 모든 영역을 메모리에 load하여 시간이 오래 걸린다. lazy loading을 구현하면 load_segment 함수의 동작이 빨라질것이다. 그리고 process가 처음부터끝까지 실행되지 않고 중간에 중단되는 경우에는 불필요한 disk I/O를 줄일 수도 있다.

또는 이렇게도 생각해볼 수 있다. 맨 처음부터 process의 모든 segment들을 physical memory에 loading하게 구현되어있는데, process의 모든 segment들을 바로 physical memory에 올려도 ‘바로’ 사용되는 영역은 많지 않다. 당장 쓰지도 않을 page에 대해 frame을 physical memory 영역을 할당하면 동시에 돌릴 수 있는 프로세스의 수가 제약받게된다. 그렇기에 lazy loading을 구현하면 process에서 당장 필요한 page들에 대해서만 frame을 할당해주게 되어 memory efficiency가 오르게 된다.

2.3 Blueprint : how to implement it (detailed data structures and pseudo codes)

lazy loading을 구현하는 방법은 여러가지 방법이 있을 것 같다. load_segment 함수에서 page를 먼저 allocate(physical memory에 공간을 할당받는다.) 해두고 나중에 해당 페이지로 값을 load하는 방법이 있을 수 있고, page를 allocate 하지 않고 관련된 정보만 기록해둔 이후, 나중에 allocate까지 하는 방법이 있을 수 있다. 후자의 방식을 택할 경우 physical memory가 뒤늦게 할당되는 효과를 볼 수 있을 것이다. 전자의 방법과 후자의 방법 중 어떤 방법이 더 좋을지는 직접 해봐야 알 수 있을 것 같고 현재로써는 후자의 방식이 더 좋아보이므로 이에 대해 설명하겠다.

가장 먼저 수정해야할 함수는 load_segment 함수이다. 기존에 이 함수는 palloc_get_page 함수를 통해 page를 할당하고, file_read 함수를 통해 할당된 페이지에 disk에 있는 executable file 정보를 memory로 올린다. 그리고 install_page 함수를 통해 할당된 페이지에 대한 메모리 매핑을 만들어냈다. 이러한 동작을 나중에 할 수 있도록 관련 정보만 기록해두는 식으로 구현을 바꿀 것이다. 이후 supplemental page table의 설명에서 이 부분을 조금 더 자세히 알아보겠다. 간단하게만 생각해보면 supplemental page table의 entry를 위한 공간을 할당한 다음, 필요한 정보들을 저장한다. 그리고 supplemental page table에 삽입할 것이다.

loading을 할 곳에서 loading을 하지 않고 기록만 해두었으니, 실제로 loading을 하는 부분을 구현해야한다. 지금까지의 상황을 보면 process가 실행 중일때 memory에 있어야 하는 executable file의 정보가 없는 예외적인 상황이 발생하게된다. loading을 할 곳에서 loading을 하지 않고 기록만 해두었기 때문이다. 따라서 page fault가 발생할 것이다.

page_fault 함수에서 loading을 하는 기능을 추가해줄 것이다. (page_fault 함수는 exception_init 함수에서 intr_register_int 로 등록되므로 page fault 발생시 호출될 것이다.) 하지만 load_segment 함수의 구현대로 palloc_get_frame 함수를 활용하지 않고 frame table 파트에서 작성하기로 한 frame을 할당받는 함수인 alloc_frame 함수를 활용할 것이다. 그리고 메모리 매핑을 위해서는 동일하게 install_page 함수를 활용할 것이다. page_fault 함수는 alloc_frame 함수의 caller이기 때문에 여기서 frame_entry를 구성하고 frame table에 frame_entry를 넣어야한다. 추가로, lazy loading을 해야하는 경우에 발생하는 page fault는 page_fault 함수의 not_present 값이 true일 것이다. page_fault 함수는 다른 경우에 다른 일도 할 것이기 때문에 해당 값을 활용하여 위에서 묘사한 작업을 하도록 한다. 이후의 설명에서 page_fault함수는 더 다루어진다.

swap에 대해서 추가적으로 이야기해보자면, dirty bit이 set된 page들만 swap disk로 쓰여져야하고 변경되지 않은 페이지, 즉 dirty bit이 0인 page들은 swap disk로 쓰여져서는 안된다. 왜냐하면 file로 부터 다시 읽으면 되기 때문이다. parallelism을 지원해야하는데, 이는 I/O를 필요로 하는 page_fault 함수 실행이 되는 동안에, 다른 프로세스가 일으키는 I/O가 필요 없는 page_fault 함수 실행이 가능해야한다는 점이다. page_fault 함수 내에서 I/O를 하는 경우에 따른 분기를 잘 하고 mutal execulsion을 사용하여 구현하면 되겠다.

load_segment 함수의 현재 구현을 보면 page_read_bytes와 page_zero_bytes가 있는데, page_read_bytes 변수의 의미는 executable file로 부터 read한 바이트의 수이고, page_zero_bytes는 할당된 페이지의 남은 부분의 바이트수를 의미한다. 이는 0으로 채워지는 부분의 바이트 수와 같다. 관련 정보를 supplemental page table에 적어줘야겠다.

3. Supplemental Page Table

3.1 Basics the definition or concept, implementations in original pintos (if yes)

Supplemental Page Table

Supplemental Page Table은 process의 메모리 관리를 돕기 위해 고안된 자료구조다. 이 테이블은 이름 그대로 기존의 페이지 테이블를 대체하는 것이 아닌 보조하여 사용되며, 운영 체제가 프로세스의 가상 메모리를 더 효율적으로 관리하는 데 도움을 준다. Supplemental Page Table은 기본 페이지 테이블에서 관리하지 않는 추가 정보를 저장한다. 예를 들어, 각 가상 페이지의 상태(예: 디스크에 스왑된, 메모리에 로드된, 아직 로드되지 않은 등)와 관련 데이터를 포함할 수 있다. 이러한 추가 정보를 사용을 해서 lazy loading을 할 때 필요한 정보를 지원한다. 또한, 아직 메모리에 로드되지 않은 페이지에 접근하여 page fault가 발생했을때, kernel이 page 정보를 쉽게 얻을 수 있게 보조한다. 또한 이후에 나올 swap out 과정에서 필요한 정보 (어떤 page가 swap되었는지, page의 disk에서 위치)등을 제공한다.

Hash Table

Pintos에서의 Hash Table은 key와 value 사이의 mapping을 효율적으로 제공하는 데이터 구조다. Pintos 내에서 이 기능은 두 가지 주요 구조를 사용하여 구현된다. 첫 번째는 struct hash로, hash table 그 자체를 나타낸다. 이 구조는 사용자가 직접 내부 구조에 접근하지 않도록 불투명하게 설계되었으며, Pintos 라이브러리에서 제공하는 특정 함수와 매크로를 통해서만 상호작용한다. 두 번째는 struct hash_elem으로, hash table에 저장될 요소를 위한 구조다. hash table에 포함시키고자 하는 각 요소는 struct hash_elem을 내장하고 있어야 한다. 이 또한 불투명한 구조다. hash table에 요소를 추가할 때 해당 키를 사용하고, 요소를 검색할 때는 이 키를 사용하여 요소를 빠르게 찾는다. 여기서 hash table을 사용하려면 사용자가 정의해야 하는 함수들이 몇 가지가 있다. 그 중에는 hash function이 있다. Hash funtion은 key에서 table index를 계산하는 데 사용되며, 이 함수는 테이블에서 키를 균일하게 분포시켜 효율적인 성능을 보장해야 한다. Hash table은 이상적인 조건에서 조회, 삽입, 삭제 작업에 대해 평균적으로 O(1)의 시간 복잡도를 제공하기 때문에 Supplemental page table처럼 많은 access를 하는 경우에는 매우 효율적인 데이터 구조다.

3.2 Limitations and Necessity : the problem of original pintos, the benefit of implementation

Supplemental page table이 구현되어 있지 않은 경우에는 page table만 사용을 하게 된다. 이때 page table은 virtual address와 physical address 사이의 mapping을 기록한다. 이는 추가적인 기능을 도입할 때 필요한 각 page의 위치(메모리 내, 디스크 상, 아직 로드되지 않음) 및 상태와 같은 자세한 정보가 부족한 상태다. 이러한 자세한 정보 없이 swapping이나 lazy loading을 도입할 경우, 메모리의 효율적인 관리가 불가할 수 있다. 또한, page fault의 처리를 할때 충분한 정보가 주어지지 않아 세분화된 fault handling을 진행하기 어려울 수 있다.
Supplemental page table을 구현하게되면 위에 언급된 문제들을 전부 해결할 수 있다.
Supplemental page table은 page table을 보조하여 더 효율적인 메모리 관리를 가능하게 한다.

Supplemental Page Table에 페이지와 관련된 정보를 기록해둠으로써 page fault handling의 기능을 다양하게 할 수 있다.

3.3 Blueprint : how to implement it (detailed data structures and pseudo codes)

문서에서 시키는대로 supplemental page table을 hash table로 supplemental page table을 구현할 것이다. Virtual Address가 주어졌을 때 해당 page에 대응되는 entry에 바로 접근하기 용이하기 때문이다. kernel/hash.c에 있는 struct hash 구조체를 사용하여 supplemental page table 역할을 할 supp_page_table을 만들 것이다. 이렇게 만들어진 supp_page_table은 모든 thread들이 하나씩 가지고 있어야하기 때문에 struct thread의 멤버 변수로써 쓰일 것이다. 그러므로 초기화는 start_process 함수와 같이 process 생성 초기에 해줄 예정이다.

supplemental page table은 hash table로 구현할 것이고 그 entry는 여러 정보들을 저장할 것이다. 그 중에는 우선 page를 physical memory로 load 시킬 때 필요한 정보들이 있겠다. user virtual address, file pointer와 file offset, file에서부터 page로 읽어올 값의 길이인 read_bytes와 page 크기에서 read_bytes를 제외한 값인 zero_bytes, writable 등 기존 load_segment 함수에서 사용하는 정보를 저장할 수 있게 entry를 구성해야한다. 또한 변경된 데이터를 file 또는 swap disk로 저장하게 해야한다는 요구사항이 있는데 어디로 저장할지에 대한 정보 또한 supplemental page table에 기록해둘 필요가 있겠다.

지금까지의 흐름에서 supplemental page table의 사용처에 대해서 짚어보자. page fault가 일어날 때 supplemental page table에 기록해둔 정보를 바탕으로 lazy loading을 구현할 것이므로 page_fault 함수에서 supplemental page table의 entry를 활용할 것이다. 이 밖에도 supplemental page table은 이후의 구현에서 사용될 것이고 상황에 따라 추가 정보를 저장하도록 구현될 수 있다.

page_fault 함수에서 supplemental page table을 주로 사용하게 될 것이므로 여기서 page_fault 함수에 대한 구현계획을 더 쓸 필요가 있겠다. invalid한 page fault는 다음과 같은 상황을 포함한다. 우선 user process가 page fault가 생긴 주소를 원하지 않는 경우이다. 그리고 페이지가 kernel virtual address에 있는 경우, 접근이 read-only인 페이지인 경우이다. 이 세 경우는 invalid 한 접근이고 process를 종료시킬 것이다.

4. Stack Growth

4.1 Basics the definition or concept, implementations in original pintos (if yes)

Stack Growth

Stack Growth는 프로그램의 stack 영역이 동적으로 확장되는 과정을 의미한다. 이는 프로그램이 실행되는 동안 stack이 추가 메모리를 필요로 할 때마다 추가로 할당하는 방식으로 작동한다. 더 자세히 말하자면, 프로그램 실행 중 지역변수나 함수 호출 등의 사유로 stack 영역이 더 많은 메모리를 필요로 할 때, kernel은 stack 영역을 동적으로 확장시켜 추가 메모리를 제공하는 기작이다. 현재 핀토스에서는 해당 기능이 구현되어 있지 않다.

4.2 Limitations and Necessity : the problem of original pintos, the benefit of implementation

기존 Pintos에서의 구현에서는 physical memory에 할당되는 stack segment를 4KB로 고정되어 있다. 이는 제한된 크기의 stack 영역만을 사용한다는 것으로, 기존에 할당된 stack 영역을 넘어설만큼 복잡한 프로그램을 수행이 불가능하다는 의미이다. 그래서 Stack Growth를 구현하여 stack segment를 필요하다면 확장을 할 수가 있어야 한다. stack 확장이 가능하게 되면 기존에 제한된 크기의 stack으로 동작시킬 수 없었던 프로그램을 동작시킬 수 있게 된다.

4.3 Blueprint : how to implement it (detailed data structures and pseudo codes)

stack growth 기능을 하는 함수를 하나 만들 예정이다. 이 함수에서는 stack growth가 필요한 주소로의 접근인지 판단하는 메커니즘을 구현해야한다. Pintos에서 Stack 영역은 한 번 새로운 데이터를 push할 때 최대 32 바이트만큼 push할 수 있다. 그렇다면 스택 포인터로부터 최대 32 바이트 떨어진 주소로의 접근은 stack growth가 필요한 상황이라고 생각하는 것이 합리적이다. 간단한 if 검사문을 사용하여 stack growth가 필요한 상황인지 아닌지 파악할 수 있다. 또한 Pintos 문서상 정해져있는 제한도 구현해야 한다. 공식 문서에 따르면, stack이 설령 stack growth를 통해 크기가 변할 수는 있더라도, 8MB가 최대라고 나와있다. 고로 stack이 할당될 수

있는 영역은 PHYS_BASE로부터 8MB를 뺀 주소까지일 것이다. Stack growth는 한번 진행될 때마다 4KB만큼 stack의 크기를 증가시키기 때문에, page_fault 함수가 받아오는 intr_frame의 멤버인 stack pointer esp를 사용해서 stack의 크기가 할당될 영역을 벗어나는지 (PHYS_BASE - stack pointer가 8MB 보다 작은지)만 확인해주면 공식문서에 부합하는 제한을 구현할 수 있을 것이다. stack을 다루는 page들 또한 evict될 수 있다. evict되는 page들은 swap disk에 쓰여야한다.

stack growth를 수행하는 기능은 1번에서 구현해둔 alloc_frame 함수를 사용해서 새로운 stack frame을 할당해야하는 식으로 구현할 것이다. 이번에는 stack growth를 수행하는 함수가 alloc_frame 함수의 caller 이기 때문에 stack growth를 수행하는 함수에서 frame table에 해당 entry를 넣어줘야한다. 또한 install_page함수를 호출하여 virtual address로부터 page table로의 mapping을 추가해서 더 이상 해당 영역을 접근할 때 page_fault가 발생하지 않도록 할 것이다.

한편 stack growth를 구현하더라도 stack이 처음 생성될 때의 page는 lazy 하지 않다는 점을 기억해야한다.

5. File Memory Mapping

5.1 Basics the definition or concept, implementations in original pintos (if yes)

File Memory Mapping

File Memory Mapping은 파일의 내용을 process의 virtual memory address space에 직접 mapping하는 기술이다. 이를 통해 process는 일반 메모리를 접근할 때와 동일한 방식을 사용하여 파일 데이터에 접근하고 수정할 수 있게 된다.

부연하자면, 프로그램이 파일을 메모리에 매핑할 때, kernel은 파일의 내용을 프로세스의 가상 주소 공간의 특정 부분에 매핑한다. 이렇게 하면, 파일을 읽고 쓰는 것이 메모리에 데이터를 읽고 쓰는 것처럼 간단해진다. Process는 일반적인 파일 I/O syscall (read, write 등)을 사용하지 않고도 파일 내용에 직접 접근할 수 있게 되어 효율성이 많이 올라간다. 특히 대용량 파일의 경우 syscall 사용시에는 메모리와 디스크 사이에서 반복적으로 복사를 하게 되는데, lazy loading을 추가로 사용해주면 파일을 메모리에 필요한 만큼 mapping하여 낭비를 막게 된다. 이러한 mapping이 이루어진다면 파일 데이터를 수정할 때에도 메모리에서 변경을 한 다음에, 나중에 mapping을 할당 해제할 때 dirty 여부를 판단하고 disk에 write back을 해주어 해당 변경 사항이 실제 파일에도 저장되도록 한다.

5.2 Limitations and Necessity : the problem of original pintos, the benefit of implementation

기존의 Pintos 구현에서는 file을 memory에 mapping하는 것에 대해 아무런 언급이 없었다. 애초에 파일을 메모리에 load하지 않고 read나 write같은 system call을 사용해 file을 다루도록 구현되어 있기 때문이다. 이러한 구현은 file을 읽을 때마다 disk reading을 해야한다는 단점이 있다. 또한, read 같은 system call을 사용해 file을 다루게 되면 kernel이 file daya를 user space에 계속 복사를 해주어야 하는데, 이는 공간적, 시간적으로도 낭비일 뿐만 아니라 cache의 효율 또한 떨어트린다는 단점이 있다. 이러한 단점들을 해결하는게 이번에 구현할 File memory mapping이다. File의 data를 disk에서 계속 읽어오는게 아니라, memory에 load를 해두고 필요할 때 disk를 안거치고 읽어올 수 있도록 하겠다는 것이다. 이렇게 하면 disk reading이라는 상당히 시간적으로 waste가 심한 과정을 반복해서 거칠 필요가 없어지고, data 복사라는 전반적인 효율을 떨어트리는 과정 또한 줄어들게 된다. 이는 프로그램은 메모리 연산을 통해 파일에 대한 읽기, 쓰기를 직접 수행할 수 있게 되기 때문이다.

5.3 Blueprint : how to implement it (detailed data structures and pseudo codes)

문서에서 File Memory Mapping을 위해 주어진 조건은 mmap, munmap 함수들을 구현하는 것이다. mmap 함수는 어떤 process의 file descriptor에 있는 파일을, 인자로 전달 받은 주소 (virtual address) 위치로 mapping 시키는 함수다. 성공 시 map된 index를 반환하도록 구현하고 실패시 -1을 리턴하도록 구현할 것이다. munmap 함수는 인자로 주어진 mapid_t mapping을 메모리로부터 제거하는 기능을 한다. 두 함수 모두 system call function을 구현하기 위해 만들 것이다.

위 함수들의 구현을 위해서는 thread마다 file과 memory space 간의 관계를 저장하는 table역할을 할 자료구조, 즉 file mapping table이 필요하다. 이 file mapping table 역할은 thread별로 있는 mmap_list가 하게 될 것이다. 이 mmap_list의 mmap_entry들은 하나의 file을 가리킬 것이며, 그 file이 해당 thread 기준으로 virtual memory의 어떤 부분을 가리키고 있는지를 저장하도록 할 것이다. 이 정보가 있어야 mmap 관련 page fault를 적절히 처리할 수 있기 때문이다. 그리고 over lap에 대한 검사를 할 때도 이 정보가 필요하다. 한편 하나의 file은 page하나 이상을 사용할 수 있기 때문에, 여러 개의 supplementary page table entry를 차지할 것이다. 해당 정보를 모두 (thread별로 있는) supplementary page table에 저장해야한다. mmap_entry에는 파일이 사용하는 페이지 수로 넣을 수 있겠다.

위 구현들이 끝나면 이제 본격적으로 mmap함수와 munmap함수를 구현해볼 수 있다. mmap 함수부터 구현 계획을 세워보자. 일단 가장 먼저 파일 크기를 커버할 만큼의 supp_entry들을 생성해야한다. 이 supp_entry들을 supp_page_table에 hash_insert 함수로 insert하는 것으로

lazy loading을 구현할 수 있다. 이때 검사를 몇 가지를 할 것인데, file의 크기가 0바이트인지, addr가 page aligned (4KB로 나누어떨어지는가)인지, 이미 mapping한 page와 겹치는지, fd가 0이나 1인지 등을 검사하고 만역 하나라도 해당한다면 지금까지 거친 과정을 되돌리고 -1을 반환하도록 구현해야한다. 이 검사들을 통과하면 성공인 것이다. 이때 우리는 thread가 가지고 있는 mmap_entry의 개수를 return하게 하여 이를 map id로써 사용할 생각이다.

page들을 lazy하게 load하는 것으로 구현해야하므로 이전에 executable file을 loading할 때와 유사하게, mmap 함수(system call을 구현하기 위한 함수)에서는 supplemental page table에 entry를 추가하는 식의 구현을 할 것이다.(위에서 언급함) 그리고 page_fault 함수에서 해당 경우에 대한 처리를 할 수 있도록 할 것이다. 할당된 page가 파일에 비해 커서 사용하지 않는 부분이 생길 수 있는데 (4KB - file크기%4KB 의 공간) 이 부분은 0으로 설정해야하는 부분도 기억해야한다.

munmap함수의 구현 계획을 세워보자. 일단 인자로 받는 mapid_t mapping에 해당하는 mmap_entry를 지워 page와 file 간의 관계를 삭제해야한다. 이는 memory mapped file이 사용하고 있는 frame을 할당해제하고 매모리 매핑을 없애주는 것으로 구현될 수 있다. 물론 write back은 필수적으로 해줘야한다. 그렇지 않으면 정보가 손실될 것이다. 이를 파일이 차지하고 있던 모든 page(frame)에 대해서 해줘야한다. 이를 구현하기 위해서는 일단 위에서 언급했던 mmap_list에서 mapping에 대응되는 mmap_entry를 찾아야한다. supplemental page table에 저장해두었던 값을 모두 보면서 write back 하는 구현을 할 것이다. 그리고 해당 영역에 대한 mapping을 지워준다. 이는 현재 thread의 page directory를 pagedir_clear_page 함수의 인자로 넣어줌으로써 할 수 있을 것이다.

추가로, 문서에서 요구하길 파일을 닫거나 지울 때는 unmap 되지 않아야한다. 따라서 file_reopen함수를 활용하여 mmap 함수를 구현할 것이다. page가 evict될 때 해당 영역에 있는 값을 file에 write back해주는 구현 또한 할 것이다.

6. Swap Table

6.1 Basics the definition or concept, implementations in original pintos (if yes)

Swap Slot

Swap slot은 swap partition에 있는 디스크 공간의 연속적인 page와 같은 크기의 구역이다. slot의 배치를 결정하는 하드웨어 제한은 page나 frame에 비해 느슨한 편이지만, page-aligned이지 않을 이유가 없기 때문에 page-aligned이도록 만든다.

Swap Table

Swap Table은 메모리 부족 상황을 처리하기 위해 사용하는 데이터 구조다. Swap Table은 시스템의 virtual memory management에 사용되며, 사용하지 않는 메모리 page를 disk의 swap slot으로 이동시키고 관리하는 데 사용된다. Swap Table은 virtual memory page가 disk의 어느 부분에 swap되었는지 기록한다. 이는 메모리가 부족한 상황에서 kernel이 어떤 페이지를 디스크로 이동시켜야 하는지, 그리고 나중에 해당 페이지를 메모리로 다시 로드할 필요가 있을 때 어디서 찾아야 하는지를 결정하는 데 사용한다. Swap Table은 스왑 공간을 효율적으로 할당 및 할당 해제하는 것을 가능하게 해주기 때문에 page가 더 이상 필요하지 않을 때 해당 공간을 다른 page에 재할당할 수 있다. 이는 Pintos는 메모리 사용을 최적화하고, 시스템의 전반적인 성능을 개선할 수 있다.

Swap In, Swap Out

메모리가 부족할 때, kernel은 가장 적게 사용될 것으로 추정되는 페이지를 디스크의 swap space로 이동시킨다. 이 과정을 swap out이라 한다. 이와 반대로, swap된 page가 필요할 때는 다시 메모리로 로드하는 과정은 swap in이라 한다.

6.2 Limitations and Necessity : the problem of original pintos, the benefit of implementation

Swap table이 구현되어 있지 않는 Pintos는 swapping의 구현에 있어서 효율성을 추구하기가 힘들다. 특히, swap하는 공간에 어떤 페이지가 있는지와 그 위치를 추적하는 조직적인 방법이 부족하다. swap table 없이 swap된 page를 찾고 식별하는 것은 더 복잡하고 시간이 많이 걸리게 된다. Swap table을 구현하게 되면 swap out된 page를 추적하기 훨씬 수월해지며, 이는 swapping의 구현을 더 간결하고 빠르게 만든다.

6.3 Blueprint : how to implement it (detailed data structures and pseudo codes)

swap table의 역할은 swap slot이 사용중에 있는지, free한지 상태를 추적하기 위해 사용된다. evicting되는 page를 swap disk로 보내야하니까 unused swap slot을 찾을 수 있도록 해야한다. 그리고 page가 read back 또는 process가 terminated될 때 관련된 swap slot을 free 해줘야한다. swap slot은 특정 page를 담도록 reserved 되어서는 안된다. 문서에서 제시한대로 swap disk를 만들어주는 것도 잊어서는 안된다. 구현상 주의할 점은 swap slot은 lazy 하게 allocate되어야 한다는 점이다. 즉, page가 eviction 될 때만 사실상 swap disk가 사용될 것이라는 것이다.

우리는 일단 swap space를 커널이 다룰 수 있는 형태로 초기화하는 함수를 구현할 것이다. 문서에서 제시한 대로 devices/block.c 파일을 활용할 것이다. 우선 전역 변수로 block* swap_block을 가져야한다. 이 변수는 block_get_role(BLOCK_SWAP)가 반환하는 swap

ROLE을 부여받은 block device를 저장하는 용이다. 또 다른 전역 변수 bitmap* swap_bitmap은 swap block을 관리할 swap table의 역할을 할 것이다. swap disk도 page단위로 관리하기 때문에 이런 구현을 할 수 있는 것이다. 초기화 함수에서는 이렇게 만든 전역변수들을 초기화해줄 것이다. swap_block에는 위에서 언급했듯이 devices/block.c에 선언되어 있는 block_get_role 함수를 사용해서 swap ROLE을 부여받은 block device를 담게해줄 것이고, swap_bitmap은 kernel/bitmap.c에 저장되어 있는 bitmap_create 함수를 통해서 생성한 bitmap을 담게 할 것이다. 이때 0으로 bitmap의 모든 bit를 초기화해줄것이다.

그 다음에는 swap in을 수행하는 함수를 구현해야한다. swap_in이라고 하자. 이 함수는 인자로 bitmap에서 flip할 index를 전달 받아 bitmap_flip 함수를 사용해 bit 값을 flip을 시켜줘야한다. swap disk의 특정 영역(4KB)을 앞으로 사용할 수 있다는 표시를 해두기 위해서이다. 그 다음에는 block_read 함수를 통해 swap_block에서 해당 섹터의 값을 읽어서 저장할 것이다.

swap out을 수행하는 함수도 구현해야한다. swap_out이라고 하자. 이 함수는 frame을 swap space 중에서 mapping할 위치(index number)를 반환하는 함수이다. bitmap_scan_and_flip 함수를 통해 아직 비어있는 swap space의 bit를 flip 한 다음 block_write 함수를 호출해서 이를 swap disk에 저장할 생각이다. 만약 bitmap_scan_and_flip이 BITMAP_ERROR 를 리턴하면 swap disk에 공간이 없다는 것이므로 문서에서 안내한대로 kernel panic을 유발할 생각이다. 이는 frame table을 구현할 때도 함께 생각해야겠다.
swap_in 함수는 read back 될 때, page_fault 함수에서 사용할 생각이고, frame table에서 page를 evict 해야할 때 swap_out을 하도록 사용할 것이다.

7. On Process Termination

7.1 Basics the definition or concept, implementations in original pintos (if yes)

On Process Termination

On Process Termination은 process termination 과정에서 virtual memory관련 구현을 추가하면서 추가되어야할 일련의 작업과 처리 과정을 의미한다.프로세스가 종료될 때, kernel은 해당 프로세스에 할당된 모든 자원(메모리, 파일 핸들, 장치 등)을 해제하고 시스템에 반환해야한다. 이는 메모리 누수를 방지하고, 다른 프로세스가 이 자원을 사용할 수 있도록 해준다. 프로세스가 사용하던 swap space, 즉 swap out된 page가 저장된 공간 또한 해제를 해야한다. 이러한 올바른 termination process를 도입하여 시스템 리소스를 적절히 회수하고, 후속 프로세스가 원활하게 실행될 수 있게하는게 이번 과제에서의 On Process Termination이다.

7.2 Limitations and Necessity : the problem of original pintos, the benefit of implementation

이전 프로젝트에서 child 정보를 담고 있는 메모리 영역을 할당해제 해주는 것과 같이, 기존에도 메모리를 할당해제하는 것은 중요했다. 하지만 이번 프로젝트에서는 process와 관련된 더 많은 정보가 필요하여 많은 메모리 할당을 해주었다. 따라서 Process를 terminate할 때 메모리 누수가 발생하지 않도록 process가 사용하던 모든 자원을 할당 해제를 특별히 신경써서 해주어야한다. On proccess termination을 구현하여 frame, supp_page_table 등 process가 사용하던 각종 resource를 할당 해제해줌으로써 우리는 더 이상 쓰이지 않을 자원을 메모리 공간에서 제거하고 새로운 자원을 할당해줄 공간을 마련할 것이다.

7.3 Blueprint : how to implement it (detailed data structures and pseudo codes)

프로세스가 종료되는 예외적인 경우(메모리 할당을 받지 못했다거나 하는 등)는 그때그때 자원을 할당하는 시점에서 처리해주고 기본적으로 process_exit 함수에서 이를 처리하면 될 것 같다.

일단 frame_table을 할당 해제하는 함수를 구현해보자. Process가 종료될 때 우리는 process가 사용하고 있는 모든 frame과 frame_entry를 할당 해제 해야한다. 이는 thread pointer 변수를 인자로 받아, 각 frame entry를 frame_table에서 제거하고, thread가 사용하고 있는 모든 frame을 free시키고, 각 frame에 대응되는 frame_entry를 free하는 함수를 만들면 된다. frame_table이 list로 구현되어 있기 때문에 이 과정은 순회용 반복문을 사용하면 쉽게 구현 가능하다. 이 때 실행해야될 함수는 list_remove, palloc_free_page, free 정도가 될 것이다.

그 다음에는 supp_page_table을 할당 해제하는 함수를 구현해보자. Process가 종료될 때에는 supp_page_table과 그 안에 있는 모든 entry들은 할당 해제가 되어야 한다. 이 함수는 일단 thread마다 멤버로서 가지고 있는 struct hash인 supp_page_table을 할당해제해야하는데, 이 때 사용하기 좋은 함수가 kernel/hash.c에 정의되어있는 hash_destroy함수다. 이 함수는 supp_page_table의 모든 entry들을 할당해주기에 따로 순회를 할 필요가 없다. 하지만 이 함수를 사용하기 위해서는 추가적인 구현이 필요한데, 바로 destructor 함수다. destructor함수는 할당해제할 hash_elem을 인자로 받아 할당 해제할 supp_entry의 포인터를 구하고, 이를 free시키는 역할을 할 것이다. 이 함수를 hash_destroy의 인자로 넣어주면 모든 supp_entry들을 한번에 할당 해제시킬 수 있다.

mmap_list를 할당 해제하는 함수를 구현해보자. 이는 해당 process의 모든 mmap_entry들에 munmap 함수를 사용해서 unmapping 해주는 해주면 되는 문제이다. 그냥 while문을 사용해서 list를 순회하며 munmap 함수를 호출해주면 구현이 끝이 난다.

마지막으로 swap table 구현에서 나오는 bitmap에 관련한 함수를 구현해야한다. 이는 process가 종료될 때 swap_bitmap에서 process와 관련된 swap_slot에 대응되는 bit들을 찾아 리셋시켜주어야한다.