Project3 Design Report

20210084 김지민, 20210216 양준영

Analysis of the current implementation

1. Frame table

   A. Basics (개념, 현재 구현)

      i. Frame table: 유저 페이지 하나를 포함하는 각 프레임에 대해 하나의 엔트리를 가진다. 유저 페이지에 의해 사용되는 프레임은 palloc_get_page(PAL_USER)을 호출함으로써 user pool로부터 얻어져야 한다. 프레임이 비어있다면 간단하지만, 비어있는 프레임이 없다면 특정 프레임을 골라 evict 시켜야 한다.

      ii. 현재 프레임과 페이지의 크기는 4096 bytes이며, 페이지 테이블의 관리와 페이지 테이블 할당/해제는 각각 "userprog/pagedir.c"와 "threads/palloc.c"에 구현되어있다.

      iii. userprog/pagedir.c

         1. pagedir_create (void): 커널 가상 주소들에 매핑되는 새로운 페이지 디렉토리를 생성한다. (페이지 디렉토리는 페이지 테이블의 첫 주소들을 모아둔 것. 즉, 각 엔트리는 페이지 테이블의 첫 주소를 가짐)

         2. pagedir_destroy (uint32_t *pd): 해당 pd를 가진 페이지 디렉토리를 없앤다. 이 디렉토리가 가리키는 모든 페이지 테이블의 엔트리에 있는 페이지들을 모두 free 시킨다.

         3. lookup_page (uint32_t *pd, const void *vaddr, bool create): pd 페이지 디렉토리에서 vaddr 가상주소를 가진 페이지 테이블 엔트리의 주소를 반환한다.

         4. pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable): pd 디렉토리에서 가상주소 upage를, 가상주소 kpage가 나타내는 물리적 프레임에 매핑한다.

         5. pagedir_get_page (uint32_t *pd, const void *uaddr): 유저 가상주소 uaddr에 대응되는 물리 주소가 pd 안에 있는지 조사한다. 있다면

대응되는 주소를 반환.

6. pagedir_clear_page (uint32_t *pd, void *upage): upage에 해당하는 페이지를 not present로 마크. (나중에 여기에 접근하면 page fault 발생)
7. pagedir_is_dirty (uint32_t *pd, const void *vpage): 해당 pd 속 vpage 페이지에 대한 PTE가 dirty면 true 반환.
8. pagedir_set_dirty (uint32_t *pd, const void *vpage, bool dirty): 해당 vpage에 대한 PTE를 dirty bit를 DIRTY로 set.
9. pagedir_is_accessed (uint32_t *pd, const void *vpage): vpage에 대한 PTE를 최근에 접근한 적 있다면 true.
10. pagedir_set_accessed (uint32_t *pd, const void *vpage, bool accessed): 해당 페이지에 대한 PTE의 accessed bit을 ACCESSED로 set.
11. pagedir_activate (uint32_t *pd): 해당 pd 페이지 디렉토리를 CPU의 페이지 디렉토리 베이스 레지스터 (CR3)로 로드(디렉토리의 물리 주소를 저장).
12. active_pd (void): 현재 CR3에 로드된 페이지 디렉토리 반환.
13. invalidate_pagedir (uint32_t *pd): pd 디렉토리를 활성화시킴으로써 TLB를 invalidate시키는 함수다.

iv. threads/palloc.c

1. palloc_init (size_t user_page_limit): 메모리를 초기화하고 user pool과 kernel pool로 나눠 초기화한다. 이때 user pool에 들어가는 페이지 수는 user_page_limit을 넘지 않도록 한다.
2. palloc_get_multiple (enum palloc_flags flags, size_t page_cnt): page_cnt만큼의 연속된 free page들을 얻고 그 그룹의 시작 주소를 반환한다. 만약 flag가 PAL_USER라면 user pool으로부터 페이지가 얻어진다. PAL_ZERO가 세워진다면 0으로 채워진 page들이 얻어진다.

3. palloc_get_page (enum palloc_flags flags): 하나의 free page를 얻는다.
4. palloc_free_multiple (void *pages, size_t page_cnt): pages 주소에서부터 page_cnt만큼의 페이지들을 free시킨다.
5. palloc_free_page (void *page): 하나만 free시킨다.
6. init_pool (struct pool *p, void *base, size_t page_cnt, const char *name): base부터 시작하여 page_cnt만큼의 페이지가 할당된, name이라는 이름의 pool을 초기화한다.
7. page_from_pool (const struct pool *pool, void *page): 해당 페이지가 해당 pool로부터 할당되었는지 검사한다. 맞다면 true 반환.

B. Limitations and Necessity (현재 구현의 문제점, 새 구현의 필요성)

i. 현재 핀토스에는 프레임 테이블이 구현되어 있지 않다.

ii. free한 frame이 없을 때 victim을 골라 evict시켜야 하는데, 이때 각 프레임에 대한 정보를 관리하는 이 테이블이 있다면 훨씬 효율적으로 eviction 알고리즘을 구현할 수 있다.

C. Blueprint (새 구현을 어떻게 만들 것인지 / 필요한 구조체 및 간단한 코드)

i. Frame table list에 들어갈 요소 "frame"구조체를 선언한다.

1. void * kvaddr: 그 프레임에 매핑되는 커널 가상주소
2. void * upage: 해당 프레임에 매핑되는 페이지 가상주소.
3. list_elem frame_list: 프레임 테이블 리스트에 들어갈 요소
4. tid: 이 프레임을 할당한 thread를 저장
5. not_evict: 이게 true면 이 프레임은 evict 대상에서 제외. 등등이 필요할 것이다.

ii. vm/frame.c를 새로 생성한다.

iii. 여기에 frame_table이라는 이름의 list와, frame 구조체를 선언한다.

iv. 주요 함수

1. Allocate_frame: frame을 할당하는 함수. 페이지는 User pool로부터 할당되어야 한다고 했으므로 "palloc_get_page(PAL_USER)"를 호출하여 frame 구조체의 kvaddr에 반환값을 저장한다. 만약 이 함수의 반환값이 NULL이라면 남는 frame이 없다는 것이므로 "evict frame" 함수를 호출하여 evict할 frame을 반환 받고, 해당 frame의 upage를 pagedir_clear_page 함수에 넣어 호출함으로써 그 페이지를 not present로 마크한다. 이후 다시 "palloc_get_page(PAL_USER)"을 호출하여 페이지를 할당 받는다. 이렇게 frame 구조체의 kvaddr에 페이지를 할당하고 난 뒤, frame_table list에 이 프레임을 추가하고 해당 프레임을 반환한다. (아마 이 페이지가 축출될 때 필요하면 디스크에 write하는 함수까지 추가되어야 할 것이다.)
2. Evict_frame: 이 함수는 frame_table list를 돌면서 victim을 찾아 그 프레임을 반환한다. 다양한 알고리즘 중 일단 clock 알고리즘을 기준으로 한다. 우선 최근에 리스트에서 축출되었던 요소의 바로 다음 요소부터 시작하여 해당 프레임의 페이지가 access된 적 있는지 검사한다. 이때 pagedir_is_accessed 함수를 이용하면 된다. 만약 true를 반환한다면 pagedir_set_accessed 함수를 호출하여 해당 페이지의 access bit를 0으로 설정하고 다음 요소로 넘어간다. 이런 식으로 반복하다가 만약 pagedir_is_accessed 함수가 0을 반환했다면 그 프레임을 반환한다.
3. Deallocate_frame: 인수로 프레임에 매핑되는 kvaddr를 받으면, 거기에 해당하는 프레임을 찾아 list에서 삭제한다. 만약 필요하다면 조건에 따라 palloc_free_page를 호출하여 페이지 해제까지 진행한다. 이후 해당 프레임을 free시킨다.
4. Search_frame: 유저 프로세스 혹은 쓰레드가 의해 사용되는 프레임을 찾는다. 이는 프레임에 tid가 저장되어 있으므로, frame_table list를 순회하면서 해당 쓰레드의 tid와 일치하는 frame을 반환하면 될 것 같다.

2. Lazy loading

A. Basics

i. Lazy loading: Demanding Page을 하는 것으로, 프로세스가 시작할 때 모

든 페이지들이 로드되는 게 아니라 실제 필요할 때만 로드되는 방식이다.

ii. Page fault: 프로세스가 가상주소에 접근할 때, 거기에 해당하는 페이지가 실제 물리적 메모리에 없는 경우를 말한다.

iii. static void page_fault (struct intr_frame *f): "threads/exception.c"에 있다. 현재는 다른 exception들과 마찬가지로 다루게끔 되어있다. not_present (page가 존재하는지), user (유저에 의한 접근인지), fault_addr (fault address) 변수를 이용하여 만약 "not_present || !user || is_kernel_vaddr(fault_addr)"를 만족한다면 exit(-1)를 호출하도록 되어있고, 아니라면 나중에 kill(f)하고 끝난다.

B. Limitations and Necessity

i. Load_segment 함수에서 현재는 세그먼트를 바로 메모리에 로드하는 형식이다. 이를 실제로 필요할 때만 로드하여 메모리를 효율적으로 사용하도록 한다.

ii. 현재 page_fault 함수 구현에 의하면 페이지 폴트가 발생하면 exit(-1) 혹은 kill(f)를 호출함으로써 항상 stop하도록 되어있다.

iii. 따라서 만약 할당된 페이지에 대해 페이지 폴트가 발생한다면(lazy loading에 의한 page fault라면) 이 페이지를 메모리에 로드한 후, procedure가 끝난 시점부터 다시 process operation을 재개할 수 있도록 구현한다. 즉, invalid access error가 아닌 경우에는 페이지를 로드하고 다시 재개할 수 있도록 수정해야 한다.

C. Blueprint

i. exception.c의 page_fault 함수에서 wrong memory access인 경우를 먼저 처리해주어야 한다.

1. is_kernel_vaddr(fault_addr) 값을 먼저 조사하여 true라면 잘못된 메모리 접근이므로 kill 한다.

ii. 만약 not present인 경우라면 이후 나올 Load_page 함수를 호출하여 fault_addr에 해당하는 페이지를 frame에 올린다. 이상이 없다면 kill이 아니라 return 하여 다시 재개될 수 있도록 한다.

iii. process.c의 load_segment를 수정함으로써 lazy load를 구현한다.

1. While 문 내에서 페이지를 할당받고, 파일을 읽고 하는 과정 대신, Thread_current -> supplemental page table, upage, file 등을 인수로 하여 (뒤에서 설명될) supp_insert_page 함수를 호출한다. 즉, 해당 페이지의 정보만 supp 페이지 테이블에 저장하고 실제로 frame을 할당하지는 않는다. 이렇게 하면 나중에 이 페이지에 접근할 때 page fault가 일어날 것이고, 핸들러에서는 supp 페이지 테이블의 정보를 기준으로 load_page (뒤에 나옴)을 호출함으로써 필요할 때만 페이지를 로드할 수 있다.

3. Supplemental Page Table

A. Basics

i. Supplemental page table: page table을 보완하며, 각각의 페이지에 대한 추가 데이터를 가진다. Page fault시 커널은 이 테이블을 lookup해야 하며, 프로세스를 종료시킬 때도 이 테이블을 lookup함으로써 어떤 리소스를 free시킬지 결정한다. Hash table로 구현한다.

ii. Hash 구조체 (lib/kernel/hash.h, hash.c): key에 해시 함수를 적용하여 고유 인덱스를 생성하고 그 위치(buckets이라 불리는 list)에 value를 저장하는 구조. 주요 함수 몇 개만 알아본다.

1. Struct hash: 해시 테이블의 요소 개수, buckets (값이 저장되는 리스트), 해시 함수 등으로 구성된다.

2. hash_init (struct hash *h, hash_hash_func *hash, hash_less_func *less, void *aux): 해시 테이블 h가 주어진 "hash"를 해시 함수로 갖도록 하고, buckets 크기도 할당하는 등 해시테이블을 초기화한다.

3. hash_find (struct hash *h, struct hash_elem *e): 해시 테이블 h에서 e와 같은 element를 반환한다.

B. Limitations and Necessity

i. 페이지 테이블은 가상주소를 물리 주소로 변환해주는 역할을 한다. 그러나 페이지 폴트가 발생했을 때 커널은 그 페이지를 찾아 데이터가 어디 있는지 알아내야 하고 프로세스가 종료될 때 어떤 리소스를 해제시

켜야 하는지 결정해야 하는데, 페이지 테이블만으로는 이 기능들을 구현하기 어렵다. 따라서 페이지에 대한 추가 정보를 담은 이 테이블이 필요하다.

ii. 각 페이지에 대해 데이터가 어디 존재하는지, 이 페이지가 대응되는 커널 가상주소, 이 페이지가 현재 dirty인지 등등을 저장한다.

C. Blueprint

i. 우선 supplemental_page_table 구조체를 만든다.

1. struct hash supp: 해시 테이블을 이용하므로 해시 구조체를 멤버로 갖는다.

ii. 그리고 thread.h에 이 구조체를 멤버로 추가한다.

iii. 그리고 supplemental_page_table의 요소가 될 엔트리 구조체인 supp_entry 구조체도 만든다.

1. Void *upage: 이 페이지의 가상주소
2. Void * kvaddr: 페이지가 할당된 프레임의 가상주소. 즉, 만약 upage가 프레임에 할당되어 있지 않다면, 이 값은 NULL!
3. hash_elem ha: supplemental_page_table이 해시 테이블로 구현된다. hash.c의 많은 함수들이 hash_elem을 사용하므로 함수 사용을 위해 이 멤버를 추가한다.
4. Int status: 페이지의 데이터가 현재 어디 있는지 상태를 나타낸다. 예를 들어, 이 값이 1이면 현재 메모리 프레임에, 2면 swap slot에 있다는 의미가 될 것이다.
5. Bool dirty: 수정되었는지 여부.
6. swap_index: 뒤에 나올 swap table에서의 인덱스.
7. Struct file* file: 만약 페이지가 파일로부터 로드되었다면 해당 파일을 저장.
8. Off_t offset: 파일의 오프셋. 등등이 필요할 것이다.

iv. 또한 hash_init 함수에서 쓰일 hash 함수와 less 함수를 구현해야 한다.

1. Supp_hash: 우리는 주어진 페이지 주소를 통해 해시 테이블에서 엔트리를 찾아야 하므로 hash 함수가 페이지 주소 즉, upage를 기준으로 해시할 수 있도록 구현해야 한다. 암호화하는 hash_int 함수에 upage 값을 넣어주고 호출한다.
2. Less: 이 함수 역시, 두 element의 upage값을 비교하여 그 결과를 반환한다.

v. vm/page.c를 새로 생성한다.

vi. 우선 새로운 페이지를 이 테이블에 추가하는 함수가 필요할 것이다.

1. supp_insert_page: 우선 엔트리인 supp_entry를 malloc으로 엔트리 크기만큼 할당받은 뒤, 해당 페이지의 주소 upage를 이 엔트리의 멤버인 upage에 저장한다. 이후 supp 테이블의 해시 구조체에 이 엔트리를 삽입한다(hash_insert 호출). 이때, 이 페이지의 데이터가 현재 어디있냐에 따라 함수의 매개변수가 달라질 것이다. 만약 파일에 존재한다면 해당 file과 오프셋 등을 추가로 받아야 할 것이다. 따라서 이 함수를 위치 각각에 대해 따로 선언해주어야 할 것이다.

vii. 페이지 폴트 시, 커널은 이 테이블을 참조한다고 했으므로 관련 기능을 구현한다.

1. lookup: 페이지 폴트가 발생한 페이지 주소를 받으면, supp 테이블을 lookup함으로써 해당 페이지 주소를 가진 엔트리를 반환한다. 해시 테이블이므로 hash_find 함수를 이용함으로써 쉽게 찾을 수 있을 것이다.

viii. 페이지를 메모리 프레임에 할당하는 함수도 필요할 것이다.

1. Load_page: 우선 위에서 구현한 lookup 함수를 호출하여 해당 페이지가 supp 테이블에 있는지 조사한다. 즉, 유효한 페이지 주소를 가리키고 있는지 검사한다. 이후 frame.c에서 구현한 allocate_frame 함수를 호출하여 프레임을 할당받은 후, 이 프레임에 해당 페이지의 데이터를 올린다. 이때, swap disc에 있다면 이후 나올 swap in 함수를 호출하고, 파일에 있다면 load_from_file 함수를 호출한다. 마지막으로 pagedir_set_page 함수를 호출하여 해당 주소와 프레임을 연결한다.

2. Load_from_file: 해당 supp_entry의 file값 등을 이용하여 file_read 함수를 호출한다.

ix. 페이지가 쫓겨날 때, 만약 수정되었다면 파일에 이를 반영해야 한다.

1. 해당 supp의 엔트리의 dirty가 true라면 이를 파일에 써야 하므로 file_write_at 함수를 호출함으로써 해당 페이지의 값을 파일에 쓴다. 이후 프레임을 free시키고, pagedir_clear_page 함수를 호출하여 해당 페이지를 not present로 마크한다.

x. 페이지가 쫓겨날 때, 해당 엔트리는 frame 테이블, swap 테이블에서도 삭제되어야 한다.

1. 페이지의 status에 따라, 만약 frame에 있다면 deallocate_frame을 호출하여 해당 페이지의 프레임을 프레임 테이블에서 삭제한다.

2. 만약 swap 테이블에 있다면 swap_index에 해당하는 엔트리를 그 테이블에서 삭제한다.

xi. 2번 lazy loading에서 봤듯이, Exception.c의 page_fault 함수에서 page fault가 일어났을 때 not_present인 경우 load_page를 호출함으로써 페이지를 프레임에 올린 후 return한다.

4. Stack Growth

A. Basics: 지금까지는 1 page 크기로 size가 고정되어 있었다. 이제는 가변적으로 만든다.

B. Limitations and Necessity: 1 page밖에 없으니 공간이 부족할 수 있다. 이 경우 페이지를 load하지 못할 수 있다. 따라서 스택 크기를 유동적으로 키워 유용하게 사용하도록 한다. Page fault가 발생했을 때 스택 크기에 관한 거라면 스택 크기를 키우도록 한다.

C. Blueprint

i. 우선 thread 구조체에 esp 멤버를 추가한다.

1. 이 값은 user에서 커널모드로 초기에 변할 때 저장해야 한다. 따라서 syscall.c의 syscall_handler 함수에서 현재 쓰레드의 esp 값을 f-

>esp 값으로 저장한다.

ii. Thread 구조체에 bottom을 추가하여 스택의 bottom을 가리키도록 한다.

1. Setup_stack 함수에서 현재 쓰레드의 bottom을 PHYS_BASE – PGSIZE로 저장한다.

iii. 이후 page_fault 핸들러에서 만약 user 값이 true라면 f->esp 값을 그대로 쓰고, 만약 아니라면 현재 쓰레드 구조체에 있는 esp 값을 사용한다. 이 값을 새로운 변수 curr_esp에 저장한다.

iv. fault_addr 값이 만약 PHYS_BASE - MAX_STACK_SIZE 보다 크거나 같고, PHYS_BASE보다는 작거나 같고, curr_esp – 4 (푸시를 할 때 한 번에 32 비트를 감소시키므로) 보다 크거나 같으면, stack_growth 함수를 호출한다.

v. Stack_growth 함수: 'thread_current() -> bottom - PGSIZE'값을 인수로 받는다. 1개의 페이지를 만든 후 이 주소와 매핑한다. 이후 thread_current-> bottom을 PGSIZE 만큼 감소시킨다.

5. File memory mapping

A. Basic

i. Definitions:

1. Mmap:

A. offset byte에서 시작하여 length byte를 addr에 있는 프로세스의 가상 주소 공간에 매핑한다. 전체 파일은 addr에서 시작해 연속적인 virtual page에 매핑된다.

B. 만약 시작점이 page-align 되지 않았을 때, page 시작 주소가 page-align 되지 않았을 때, 매핑하려는 페이지가 이미 넷에 존재할 때, 페이지를 만들 주소가 NULL이거나 파일 길이가 null일 때, STDIN, STDOUT일 때 실패하며, -1을 반환한다.

C. 정상적으로 동작했을 경우, 매핑한 유효 주소를 반환한다.

D. Page fault가 났을 때, 바이트를 0으로 하고, disk로 write-back할 때 버린다.

E. 이 외에는 mmap 영역에서 lazy load하며, dirty시 write back가 되도록 구현한다. 파일 길이가 pgsize의 배수가 아니면, 매핑된 최종 페이지의 일부 바이트가 파일 끝을 나올 수 있다.

2. Munmap:

A. 지정된 주소 범위에 대한 매핑을 해제한다. 이 주소는 아직 매핑 해제되지 않은 동일 프로세스에서 mmap의 이전 호출에 의해 반환된 virtual address여야 한다. 이 페이지는 mmap으로 할당한 유저 프로세스와 동일한 프로세스여야 한다

B. Process가 exit할 경우, munmap되어야 한다. Munmap될 경우, 해당 프로세스에 의해 기록된 모든 페이지는 파일에 다시 기록된다. 기록되지 않은 페이지는 기록되지 않으며, 페이지는 프로세스의 가장 페이지 목록에서 제거된다.

C. 파일을 닫거나 제거해도 매핑은 해제되지 않는다. 파일에 대한 독립적인 참조를 얻으려면 file_reopen 함수를 사용해야 한다.

ii. Implemented: Pintos에서 구현되어 있지 않다.

B. Limitations and Necessity:

i. Necessity: Mmap과 munmap이 syscall에서 구현되어 있지 않으므로, handler 번호에 맞게 해당 함수들을 실행할 필요가 있다. 따라서, 해당 함수들을 syscall.c에 추가하여 실행되도록 구현하여야 한다.

ii. Benefits: mmap을 통해 메모리에 파일을 mapping할 수 있게 된다. Munmap을 통해 매핑을 해제할 수 있다.

C. Blueprint

i. Data structure: Hash table을 사용할 수 있을 것으로 생각된다.

ii. Pseudo Code:

1. Mmap의 경우

A. 해당 주소의 시작점이 page align되어 있는지, fd 값이 정상적인지, 파일의 길이가 0이하인지, 파일에 충분한 공간이 있는지를 확인하여 그렇지 않다면 -1을 반환한다.

B. 정상적인 경우에는 0으로 채워야할 공간과 아닌 공간의 길이를 파악해서 해당 공간에 데이터를 채우도록 구현하면 될 것으로 생각된다.

C. mmap에서 열었던 파일을 reopen해주는데, 원본파일을 그대로 유지하고, 유저가 해당 fd로 close를 하더라도 munmap하지 않았다면 매핑이 유지될 수 있도록 하기 위함.

D. 이후, while 문을 돌면서 파일을 페이지 단위로 나누어 파일 정보를 저장한다. 이 과정에서 해시 테이블을 사용할 수 있을 것을 생각된다.

2. Munmap의 경우

A. Hash table에 저장한 파일을 제거하도록 한다. 이 때, 페이지가 NULL이라면 NULL을 반환하도록 한다.

B. 페이지가 수정되었다면 파일에 업데이트 하고(write back) dirty bit을 0으로 만든다. 이후, present bit를 0으로 만든다. 페이지를 virtual page의 프로세스 리스트에서 제거한다. close되거나 remove된 파일은 unmap하지 않는다.

6. Swap table

A. Basic

i. Definitions: Free Frame이 존재하지 않는 경우를 효율적으로 다루기 위한 것이 swap table이다. Swap table에서 사용가능한 영역과 사용 중인 영역을 구분하기 위한 테이블이다. 물리 페이지가 부족할 때, Victim page를 선정하여 디스크로 swap out해 여유 메모리를 확보할 수 있도록 한다.

1. Swap in: 새로 물리 메모리에 page를 load하는 것
2. Swap out: 기존에 메모리에 있던 페이지를 꺼내는 것을 swap out이라고 한다.

ii. Implemented:

1. devices/block.c에 구현된 섹터를 기반으로 읽기/쓰기 기능을 사용할 수 있다.
2. 리소스 집합에서 사용을 추적하는데 사용될 수 있는 비트맵 구조가 bitmap.c에 구현되어 있다.

B. Limitations and Necessity

i. Limits: block.c나 bitmap.c로 기능 구현에 도움을 받을 수 있지만, swap table을 구현한 부분은 없어서 새로 구현해야 할 것으로 보인다.

ii. Necessity: 물리페이지가 부족할 경우, 지금 사용할 페이지를 불러와야되며, 사용가능한 영역과 사용 중인 영역을 구분하고 효과적인 victim page를 선정해 제거하기 위해 필요하다.

iii. Benefits: 한정되어 있는 물리 메모리의 용량에 필요한 page를 load하고 LRU와 같은 알고리즘을 통해 victim page를 선택하여 swap out 하여 물리 메모리를 최대로 이용할 수 있도록 만들 수 있다.

C. Blueprint

i. Data structure: block.c에 있는 block structure(BLOCK_SWAP)를 사용하여 섹터를 블록 단위로 다룰 예정이다. 이 과정에서 block.c의 block_get_role()을 사용할 것이다. 또한, bitmap을 이용하여 해당 페이지가 swap out되어 디스크의 스왑 공간에 저장되었는지를 확인할 수 있도록 사용할 예정이다.

ii. Pseudo Code:

1. Swap table을 초기화하는 함수

A. Swap할 table을 block_get_role(BLOCK_SWAP)을 통해 받아온다.

B. 받아온 table을 바탕으로 swap_device를 페이지 크기로 나눈 만큼 bitmap을 만든다.

C. 이후 bitmap의 값을 모두 참으로 설정한다.

2. Swap in함수

A. 반복문을 통해 swap table로부터 block_read()를 통해 블록을 가져오고 bitmap_flip함수를 사용하여 가져온 블록들을 바꿔준다.

3. Swap out 함수

A. swap할 때, bitmap_swap_and_flip을 사용하여 비트맵을 바꾼다.

B. 반복문을 통해 swap slot에 페이지를 write하고 swap한 위치를 반환한다.

7. On process termination

A. Basic

i. Definitions: 프로세스를 종료할 경우, 기존에 생성했던 자료들을

전부 할당 해제해 줄 필요가 있다. S페이지 테이블, 프레임 테이블, 스왑 테이블에서 관련 내용 삭제 후 모든 관련 파일 닫기 (Dirty page는 다시 써야 함)

ii. Implemented: Process.c의 process_exit()이나, syscall.c의 exit()에서 열린 파일들을 닫고 종료한다. 하지만 새로 추가된 테이블들은 여기서 다뤄지지 않으며, 모든 파일들이 닫힌다.

B. Limitations and Necessity

i. Limits: 현재 구현한 내용에 따르면 새로 추가된 테이블들은 여기서 다뤄지지 않으며, 모든 파일들이 닫힌다.

ii. Necessity: 낭비되는 자원이나 제대로 할당 해제되지 않은 자원이 문제를 일으킬 수 있으므로 종료되는 과정에서 모든 자료를 정리해야하며, write back 방식이므로 해당 페이지의 값을 메인 메모리에 업데이트 해야한다.

iii. Benefits: 정상적인 종료를 통해 메모리를 낭비하지 않고 사용할 수 있으며, 다른 프로세스들이 정상적으로 작동할 수 있도록 한다.

C. Blueprint

i. Data structure: 특별한 데이터 구조를 사용하진 않을 것 같다.

ii. Pseudo Code:

1. process.c의 process_exit()에서 현재 프로세스의 page directory를 destroy하고 초기화한다.

2. Malloc과 같이 동적할당한 부분이 있으면 동적할당을 해제하고, 열린 파일들을 닫아준다. 이 때, write back할 것이 있으면 write back을 할 수 있을 것으로 보인다.