

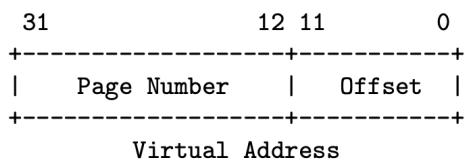
Project 3. Virtual Memory

컴퓨터공학과 20210741 김소현 20210774 김주은

The Goal of Project3 : focusing on how many user programs are executed simultaneously in efficient way

Terminology

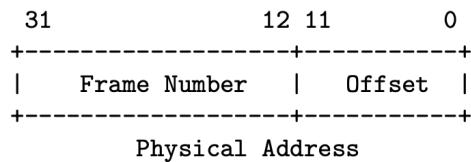
Pages (= Virtual Page)



Page는 virtual memory의 4096 bytes만큼의 continuous region을 말한다. page는 page-aligned되어 있는데, 즉 page size로 divisible한 virtual address로부터 시작한다. 그렇기 때문에 위 그림과 같이 32-bit의 virtual address는 20 bit의 page number와 12 bit의 page offset으로 나뉜다.

Each process는 independent한 user (virtual) page들의 set을 가지고, 이에 반해 kernel (virtual) page들은 global하다. 또한, kernel은 user와 kernel page 둘 다에 access할 수 있지만, user process는 only 해당 process의 user page들에만 접근할 수 있다.

Frames (= Physical Frame, Page frame)

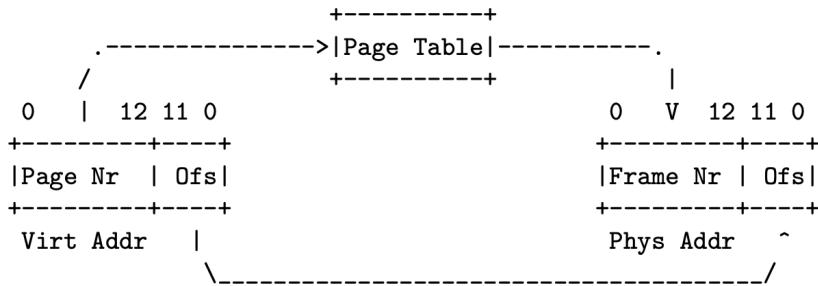


frame은 physical memory의 continuous region이다. page와 마찬가지로 frame들은 page-size, page-aligned다. 그렇기 때문에 32-bit physical address는 20-bit frame number와 12-bit frame offset으로 나뉜다.

80x86 physical address에 directly memory access하는 방법을 제공하지 않는다. pintos 는 physical memory에 kernel virtual memory를 direct하게 mapping하여 사용한다. 결국 frame들은 kernel virtual memory를 통해서 access할 수 있는 것이다.

Page Table

pintos에서는 page table이 CPU가 virtual address를 physical address로 translate할 때 사용하는 data structure이다. 즉, 위에서 다른 개념을 적용하면 page로부터 frame으로 translate하는 것이다. page table format은 80x86 architecture에 맞춰져있으며 pintos에서는 page table management code를 pagedir.c에서 제공하고 있다.



위 digram은 page와 frame 간의 relationship을 나타내고 있다. virtual address에서 page number을 page table에서 frame number로 translate해준다. 이에 반해 offset은 unmodified된 상태로 유지된다.

Swap Slots

swap slot은 swap partition에서 disk space의 continuous한 page-size region을 말한다. swap slot은 page-aligned이어야 한다.

TO DO in project 3

project3에서는 다음 4가지를 설계해야 한다.

- **Supplemental page table**
 - Enable page fault handling
- **Frame table**
 - Allows efficient implementation of eviction policy
- **Swap table**
 - Tracks usage of swap slots
- **Table of file mappings**
 - Processes may map files into their virtual memory space

- 각 파일이 어떤 페이지에 매핑되어 있는지 track할 table이 필요하다

이를 위해 pintos에서 지원하는 structure들이 있다.

1) bitmap data structure → lib/kernel(bitmap.c)

bit들의 array로, true or false가 될 수 있다. set of resources들의 사용을 관리하기 위해 사용된다. 만약 resource n이 사용되면, bit n of the bitmap은 true이다.

2) hash table data structure

insertion, deletion을 효율적으로 지원한다.

Analysis about Current Pintos System (code)

현재 pintos code를 분석하여 page table, memory system 등의 구현을 정리하고자 한다.

1. Page

virtual memory는 page 단위로 구성된다. virtual address는 20bit의 Page Number와 12bit의 Page Offset으로 표현된다.

```
/* Page offset (bits 0:12). */
#define PGSHIFT 0                      /* Index of first offset bit. */
#define PGBITS 12                      /* Number of offset bits. */
#define PGSIZE (1 << PGBITS)          /* Bytes in a page. */
#define PGMASK BITMASK(PGSHIFT, PGBITS) /* Page offset bits (0:12). */
```

thread/vaddr.h의 code

- PGSHIFT를 0으로 정의하여, virtual address에서 offset part의 index를 0으로 설정하고 있다.
- PGBITS는 Offset size가 12임을 의미한다.
- PGSIZE는 해당 bit수를 byte 단위로 나타내어 4096byte 크기를 나타낸다.
- PGMASK는 Offset만큼을 masking 하기 위해 정의한다.

```

/* Offset within a page. */
static inline unsigned pg_ofs (const void *va) {
    return (uintptr_t) va & PGMASK;
}

/* Virtual page number. */
static inline uintptr_t pg_no (const void *va) {
    return (uintptr_t) va >> PGBITS;
}

/* Round up to nearest page boundary. */
static inline void *pg_round_up (const void *va) {
    return (void *) (((uintptr_t) va + PGSIZE - 1) & ~PGMASK);
}

/* Round down to nearest page boundary. */
static inline void *pg_round_down (const void *va) {
    return (void *) ((uintptr_t) va & ~PGMASK);
}

```

- pg_ofs() : virtual address va로부터 masking하여 구한 page offset 값을 return한다.
- pg_no() : virtual address va로부터 shift하여 구한 page number 값을 return한다.
- pg_round_up() : virtual address va의 nearest page boundary 값을 반올림하여 return한다.
- pg_round_down() : virtual address va의 nearest page boundary 값을 반내림하여 return한다.

```
#define PHYS_BASE ((void *) LOADER_PHYS_BASE)
```

thread/vaddr.h

```

/* Kernel virtual address at which all physical memory is mapped.
   Must be aligned on a 4 MB boundary. */
#define LOADER_PHYS_BASE 0xc0000000      /* 3 GB. */

```

thread/loader.h

pintos pdf의 내용과 동일하게 PHYS_BASE를 0xc0000000로 설정하고 있다.

```

/* Returns true if VADDR is a user virtual address. */
static inline bool
is_user_vaddr (const void *vaddr)
{
    return vaddr < PHYS_BASE;
}

/* Returns true if VADDR is a kernel virtual address. */
static inline bool
is_kernel_vaddr (const void *vaddr)
{
    return vaddr >= PHYS_BASE;
}

```

PHYS_BASE와의 주소값의 대소비교를 통하여, user virtual address인지, kernel virtual address인지 확인하여 return하는 함수이다. project2에서 user-provided pointer의 validity를 확인하기 위하여 해당 함수를 사용할 수 있다.

```

/* Returns kernel virtual address at which physical address PADDR
   | | is mapped. */
static inline void *
ptov (uintptr_t paddr)
{
    ASSERT ((void *) paddr < PHYS_BASE);

    return (void *) (paddr + PHYS_BASE);
}

/* Returns physical address at which kernel virtual address VADDR
   | | is mapped. */
static inline uintptr_t
vtop (const void *vaddr)
{
    ASSERT (is_kernel_vaddr (vaddr));

    return (uintptr_t) vaddr - (uintptr_t) PHYS_BASE;
}

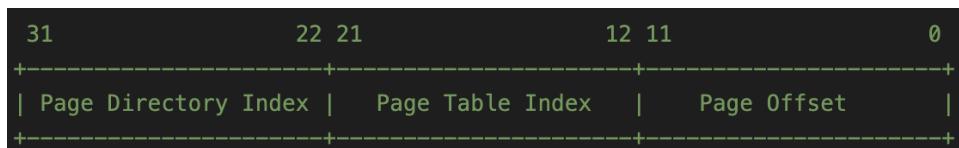
```

ptov() : physical address PADDR이 매핑되어 있는 kernel virtual address를 return

vtop() : kernel virtual address VADDR이 매핑되어 있는 physical address를 return

2. Page Table

page table entry (thread/pte.h)



thread/pte.h

```

/* Page table index (bits 12:21). */
#define PTSHIFT PGBITS           /* First page table bit. */
#define PTBITS 10                 /* Number of page table bits. */
#define PTSPAN (1 << PTBITS << PGBITS)    /* Bytes covered by a page table. */
#define PTMASK BITMASK(PTSHIFT, PTBITS) /* Page table bits (12:21). */

/* Page directory index (bits 22:31). */
#define PDSHIFT (PTSHIFT + PTBITS)      /* First page directory bit. */
#define PDBITS 10                      /* Number of page dir bits. */
#define PDMASK BITMASK(PDSHIFT, PDBITS) /* Page directory bits (22:31). */

```

위와 같이 pte.h에서 page table entry를 관리하기 위한 값을 지정하고 있다.

현재 page table은 userprog/pagedir.c에서 관리한다.

userprog/pagedir.c

pagedir.c에 있는 code는 80x86 hardware page table의 abstract interface이며 이 page table은 page directory로도 불린다.

- pagedir_create

```

/* Creates a new page directory that has mappings for kernel
   virtual addresses, but none for user virtual addresses.
   Returns the new page directory, or a null pointer if memory
   allocation fails. */
uint32_t *
pagedir_create (void)
{
    uint32_t *pd = palloc_get_page (0);
    if (pd != NULL)
        memcpy (pd, init_page_dir, PGSIZE);
    return pd;
}

```

새로운 page table을 생성하는 함수이다. 새로운 page table은 pintos의 normal kernel virtual page mapping을 포함하며 user virtual mapping을 포함하지 않는다. 만약 allocation에 실패하면 null을 리턴한다.

- pagedir_destory

```

void
pagedir_destroy (uint32_t *pd)
{
    uint32_t *pde;

    if (pd == NULL)
        return;

    ASSERT (pd != init_page_dir);
    for (pde = pd; pde < pd + pd_no (PHYS_BASE); pde++)
        if (*pde & PTE_P)
        {
            uint32_t *pt = pde_get_pt (*pde);
            uint32_t *pte;

            for (pte = pt; pte < pt + PGSIZE / sizeof *pte; pte++)
                if (*pte & PTE_P)
                    palloc_free_page (pte_get_page (*pte));
                palloc_free_page (pt);
        }
    palloc_free_page (pd);
}

```

page directory 값 pd를 제거하는 함수이다. 즉 page table 그 자체와 그것이 Mapping 된 모든 frame들을 palloc_free_page 함수를 이용하여 할당 해제 시킨다.

- pagedir_activate

```

void
pagedir_activate (uint32_t *pd)
{
    if (pd == NULL)
        pd = init_page_dir;

    /* Store the physical address of the page directory into CR3
     * aka PDBR (page directory base register). This activates our
     * new page tables immediately. See [IA32-v2a] "MOV—Move
     * to/from Control Registers" and [IA32-v3a] 3.7.5 "Base
     * Address of the Page Directory". */
    asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}

```

page directory pd를 activate 시키는 함수로, cpu의 page directory base register로 load한다.

아래 함수들은 page에서 frame으로의 mapping을 관리하는 함수다.

- pagedir_set_page

```

bool
pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)
{
    uint32_t *pte;

    ASSERT (pg_ofs (upage) == 0);
    ASSERT (pg_ofs (kpage) == 0);
    ASSERT (is_user_vaddr (upage));
    ASSERT (vtop (kpage) >> PTSIFT < init_ram_pages);
    ASSERT (pd != init_page_dir);

    pte = lookup_page (pd, upage, true);

    if (pte != NULL)
    {
        ASSERT ((*pte & PTE_P) == 0);
        *pte = pte_create_user (kpage, writable);
        return true;
    }
    else
        return false;
}

```

user virtual page UPAGE를 kernel virtual address KPAGE에 의해 명시된 physical frame으로 page directory pd에 매핑시키는 것을 추가한다. UPAGE는 반드시 이미 mapped되지 않아야 한다. KPAGE는 palloc_get_page(PAL_USER)로 user pool에서 얻은 page여야 한다. 만약 writeable이 true라면, 새로운 page는 read/write이고, 아니라면 무조건 read-only이다. 성공적이라면 return true, memory allocation fail하면 false를 return한다.

- pagedir_get_page

```

void *
pagedir_get_page (uint32_t *pd, const void *uaddr)
{
    uint32_t *pte;

    ASSERT (is_user_vaddr (uaddr));

    pte = lookup_page (pd, uaddr, false);
    if (pte != NULL && (*pte & PTE_P) != 0)
        return pte_get_page (*pte) + pg_ofs (uaddr);
    else
        return NULL;
}

```

pd의 user virtual address UADDR에 mapping된 frame을 찾아, 해당 frame에 대응되는 kernel virtual address 값을 리턴한다. 만약 UADDR이 unmapped라면 null pointer를 리턴한다.

- pagedir_clear_page

```

void
pagedir_clear_page (uint32_t *pd, void *upage)
{
    uint32_t *pte;

    ASSERT (pg_ofs (upage) == 0);
    ASSERT (is_user_vaddr (upage));

    pte = lookup_page (pd, upage, false);
    if (pte != NULL && (*pte & PTE_P) != 0)
    {
        *pte &= ~PTE_P;
        invalidate_pagedir (pd);
    }
}

```

page directory pd에서 현재 존재하지 않는 user virtual page UPAGE를 마크한다. 만약 해당 page에 나중에 access하면 fault가 발생하며, page table entry의 다른 bit는 보존된다. mapping되어 있지 않은 page에 대해서는 아무 효과가 없다.

pintos에서 page replacement algorithm을 위해서 page table entry에 이에 필요한 정보를 저장하는 bit가 존재한다. accessed/dirty bit가 그것이고, 만약 pte가 가리키는 page를 read or write 하는 경우에 해당 bit가 설정된다. 이와 관련된 함수들은 다음과 같다.

- pagedir_is_dirty

```

bool
pagedir_is_dirty (uint32_t *pd, const void *vpage)
{
    uint32_t *pte = lookup_page (pd, vpage, false);
    return pte != NULL && (*pte & PTE_D) != 0;
}

```

pd의 vpage가 dirty이면 true를 return한다. 이는 pte가 설정된 이후에 해당 page가 수정된 적이 있다는 뜻이다. 그리고 pd에 vpage에 대한 pte가 없는 경우에도 null을 return한다.

- pagedir_set_dirty

```

void
pagedir_set_dirty (uint32_t *pd, const void *vpage, bool dirty)
{
    uint32_t *pte = lookup_page (pd, vpage, false);
    if (pte != NULL)
    {
        if (dirty)
            *pte |= PTE_D;
        else
        {
            *pte &= ~(uint32_t) PTE_D;
            invalidate_pagedir (pd);
        }
    }
}

```

pd의 vpage를 위한 pte의 dirty bit을 설정하는 함수다.

- pagedir_is_accessed

```

bool
pagedir_is_accessed (uint32_t *pd, const void *vpage)
{
    uint32_t *pte = lookup_page (pd, vpage, false);
    return pte != NULL && (*pte & PTE_A) != 0;
}

```

pd의 vpage에 대한 pte에서 최근에 accessed 한 적 있으면 true를 return한다. 즉, pte가 설정되고 pte가 마지막으로 cleared된 시점 사이에 accessed 된 기간을 의미한다.
만약 vpage를 위한 pte를 pd가 포함하지 않는다면 fail을 return한다.

- pagedir_set_accessed

```

void
pagedir_set_accessed (uint32_t *pd, const void *vpage, bool accessed)
{
    uint32_t *pte = lookup_page (pd, vpage, false);
    if (pte != NULL)
    {
        if (accessed)
            *pte |= PTE_A;
        else
        {
            *pte &= ~(uint32_t) PTE_A;
            invalidate_pagedir (pd);
        }
    }
}

```

pd의 virtual page vpage를 위한 pte의 accessed bit를 설정한다.

3. Hash

pintos에서 hash라는 data structure를 지원한다. 그리고 이는 pintos/src/lib/kernel/hash.h, hash.c의 파일로 구현되어 있다.

```
pintos > src > lib > kernel > C hash.h
1  #ifndef __LIB_KERNEL_HASH_H
2  #define __LIB_KERNEL_HASH_H
3
4  /* Hash table.
5
6  | This data structure is thoroughly documented in the Tour of
7  | Pintos for Project 3.
8
```

- struct hash이다.

```
/* Hash table. */
struct hash
{
    size_t elem_cnt;           /* Number of elements in table. */
    size_t bucket_cnt;         /* Number of buckets, a power of 2. */
    struct list *buckets;      /* Array of `bucket_cnt' lists. */
    hash_hash_func *hash;       /* Hash function. */
    hash_less_func *less;      /* Comparison function. */
    void *aux;                 /* Auxiliary data for `hash' and `less'. */
};
```

lib/kernel.h

- hash table iterator을 위한 struct이다.

```
/* A hash table iterator. */
struct hash_iterator
{
    struct hash *hash;          /* The hash table. */
    struct list *bucket;        /* Current bucket. */
    struct hash_elem *elem;     /* Current hash element in current bucket. */
};
```

```
/* Hash element. */
struct hash_elem
{
    struct list_elem list_elem;
};
```

struct list_elem list_elem을 사용한다.

```
#define hash_entry(HASH_ELEM, STRUCT, MEMBER) \
| | | | ((STRUCT *) ((uint8_t *) &(HASH_ELEM)->list_elem \
| | | | | | | | - offsetof (STRUCT, MEMBER.list_elem)))
```

```
/* Basic life cycle. */
bool hash_init (struct hash *, hash_hash_func *, hash_less_func *, void *aux);
void hash_clear (struct hash *, hash_action_func *);
void hash_destroy (struct hash *, hash_action_func *);

/* Search, insertion, deletion. */
struct hash_elem *hash_insert (struct hash *, struct hash_elem *);
struct hash_elem *hash_replace (struct hash *, struct hash_elem *);
struct hash_elem *hash_find (struct hash *, struct hash_elem *);
struct hash_elem *hash_delete (struct hash *, struct hash_elem *);

/* Iteration. */
void hash_apply (struct hash *, hash_action_func *);
void hash_first (struct hash_iterator *, struct hash *);
struct hash_elem *hash_next (struct hash_iterator *);
struct hash_elem *hash_cur (struct hash_iterator *);

/* Information. */
size_t hash_size (struct hash *);
bool hash_empty (struct hash *);

/* Sample hash functions. */
unsigned hash_bytes (const void *, size_t);
unsigned hash_string (const char *);
unsigned hash_int (int);
```

- **hash_insert**

새로운 hash elem new를 hash table h에 삽입하고, table에 같은 요소가 없었다면 null pointer를 리턴한다. 만약 equal element가 이미 table에 있었다면, new를 삽입하지 않고 그것을 리턴한다.

```
struct hash_elem *
hash_insert (struct hash *h, struct hash_elem *new)
{
    struct list *bucket = find_bucket (h, new);
    struct hash_elem *old = find_elem (h, bucket, new);

    if (old == NULL)
        insert_elem (h, bucket, new);

    rehash (h);

    return old;
}
```

- **hash_find**

hash table에서 e와 동일한 element를 찾아 리턴한다. 만약 동일한 element가 table에 없다면 null pointer를 리턴한다.

```

struct hash_elem *
hash_find (struct hash *h, struct hash_elem *e)
{
| return find_elem (h, find_bucket (h, e), e);
}

```

- **find_elem**

*h*의 bucket에서 *e*와 동일한 hash element를 찾는다. 만약 찾는다면 리턴하고, 아니라면 null pointer를 리턴하는 함수이다. hash_find에서 호출되어 사용된다.

```

/* Searches BUCKET in H for a hash element equal to E. Returns
| | it if found or a null pointer otherwise. */
static struct hash_elem *
find_elem (struct hash *h, struct list *bucket, struct hash_elem *e)
{
    struct list_elem *i;

    for (i = list_begin (bucket); i != list_end (bucket); i = list_next (i))
    {
        struct hash_elem *hi = list_elem_to_hash_elem (i);
        if (!h->less (hi, e, h->aux) && !h->less (e, hi, h->aux))
        | return hi;
    }
    return NULL;
}

```

- **hash_init**

hash table *H*를 초기화한다. 주어진 auxiliary data *aux*를 사용하여, *hash*를 이용하여 hash value를 계산하고, *less*를 이용하여 hash elements를 비교한다.

```

bool
hash_init (struct hash *h,
| | | | | hash_hash_func *hash, hash_less_func *less, void *aux)
{
    h->elem_cnt = 0;
    h->bucket_cnt = 4;
    h->buckets = malloc (sizeof *h->buckets * h->bucket_cnt);
    h->hash = hash;
    h->less = less;
    h->aux = aux;

    if (h->buckets != NULL)
    {
        hash_clear (h, NULL);
        return true;
    }
    else
        return false;
}

```

- **hash_delete**

hash table *h*에서 *e*와 동일한 element를 찾아 지우고, 리턴하는 함수이다. 만약 동일한 element가 table에 없으면 null pointer를 리턴한다. 만약 hash table의 element가 동적 할당되거나 own resources이면 caller가 그들을 deallocate할 의무가 있다.

```

struct hash_elem *
hash_delete (struct hash *h, struct hash_elem *e)
{
    struct hash_elem *found = find_elem (h, find_bucket (h, e), e);
    if (found != NULL)
    {
        remove_elem (h, found);
        rehash (h);
    }
    return found;
}

```

- **hash_int**

int i의 hash를 리턴한다.b

```

unsigned
hash_int (int i)
{
    return hash_bytes (&i, sizeof i);
}

```

이때 hash_bytes 함수는 다음과 같다.

```

unsigned
hash_bytes (const void *buf_, size_t size)
{
    /* Fowler-Noll-Vo 32-bit hash, for bytes. */
    const unsigned char *buf = buf_;
    unsigned hash;

    ASSERT (buf != NULL);

    hash = FNV_32_BASIS;
    while (size-- > 0)
        hash = (hash * FNV_32_PRIME) ^ *buf++;

    return hash;
}

```

buffer의 size byte만큼의 hash를 리턴하는 함수다.

- **hash_destroy**

hash table을 destroy하는 함수이다. 만약 destructor가 비어있지 않으면, hash table의 각 요소에 대해 destructor를 호출한다. elements의 메모리를 해제하는 등의 작업을 수행 하지만, hash_clear 함수가 실행 중일 때 hash table을 변경하면 원하지 않은 동작이 발생 할 수도 있다. 즉, hash_clear(), hash_destroy(), hash_insert(), hash_replace(), 또는 hash_delete() 함수를 destroy 함수 내부나 다른 곳에서 호출하면 예측할 수 없는 결과가 나올 수도 있다.

```

void
hash_destroy (struct hash *h, hash_action_func *destructor)
{
    if (destructor != NULL)
        hash_clear (h, destructor);
    free (h->buckets);
}

```

pintos에서는 위와 같은 기본 hash function들을 제공한다. hash.c에서 구체적인 함수 정의를 확인할 수 있다.

```

/* Computes and returns the hash value for hash element E, given
   auxiliary data AUX. */
typedef unsigned hash_hash_func (const struct hash_elem *e, void *aux);

/* Compares the value of two hash elements A and B, given
   auxiliary data AUX. Returns true if A is less than B, or
   false if A is greater than or equal to B. */
typedef bool hash_less_func (const struct hash_elem *a,
                            const struct hash_elem *b,
                            void *aux);

/* Performs some operation on hash element E, given auxiliary
   data AUX. */
typedef void hash_action_func (struct hash_elem *e, void *aux);

```

- hash_hash_func

aux를 가지고 hash element e를 위한 hash value를 계산하고 리턴한다.

- hash_less_func

aux를 가지고 두 hash elem A, B의 값을 비교하여, A가 B보다 작으면 true를 return, 만약 A가 크거나 같으면 false를 return한다.

1. Frame table

1) Basics

- the definition or concept

Frame table이란, frame을 효과적으로 관리하기 위한 table이다. 현재 page, frame은 4096 bytes로 구성되어 있다. frame table은 각 frame 당 하나의 entry를 포함하며, 각 entry는 page로의 포인터를 포함한다. 가장 중요한 기능은 **unused frame을 얻는 것이다**. free한 frame이 없을 때 evict할 page를 고름으로써 eviction policy를 효과적으로 구현하는

것을 가능하게 해준다. user page를 위한 frame들은 palloc_get_page(PAL_USER)를 호출함으로써 “user pool”에서 얻을 수 있다.

만약 allocating swap slot 없이 evict될 수 있는 frame이 없고, swap이 full이라면 panic the kernel을 한다. 실제 os는 이러한 상황을 예방하거나 복구하는 정책들을 가지고 있다. 하지만 pintos에서는 따로 요구하지 않는다. process eviction은 다음과 같은 단계를 따른다.

- 1) replacement algorithm을 사용하여 evict할 frame을 고른다. 그리고 accessed와 dirty bit를 본다.
- 2) 그것을 refer하는 any page table의 reference를 삭제한다. 만약 sharing을 구현하지 않았다면, single page만 그 frame을 refer하게 된다.
- 3) 만약 필요하다면, file system이나 swap에 page를 write 한다.
- 4) evicted frame은 다른 page를 저장하는데 사용된다.

- **Implementations in original pintos**

page의 allocation/deallocation은 threads/palloc.c에 구현되어 있다.

threads/palloc.c

page allocator는 threads/palloc.h에 정의되어 있고, page 단위로 memory를 allocate한다. 한 번에 하나의 page를 allocate하는 용도로 많이 사용되지만, 여러 개의 연속된 pages를 한 번에 allocate할 때도 사용된다.

page allocator는 memory를 2개의 pool로 나누어 할당한다. → kernel pool and user pool

- struct pool

```
/* A memory pool. */
struct pool
{
    struct lock lock;           /* Mutual exclusion. */
    struct bitmap *used_map;   /* Bitmap of free pages. */
    uint8_t *base;              /* Base of pool. */
};

/* Two pools: one for kernel data, one for user pages. */
static struct pool kernel_pool, user_pool;
```

palloc.c

user pool은 user process를 위해 memory를 할당하고, 이 외는 kernel pool을 이용한다. 각 pool의 사용은 bitmap을 이용하여 트래킹되며, pool의 각 page마다 한 bit가 이

용된다. n page의 할당을 위한 요청이 들어오면, bitmap에서 n개의 연속된 bit가 false(free)로 설정되어 있는지 확인하고, 해당 부분을 찾으면 true로 바꾸어서 사용되는 것으로 설정한다. 이는 가장 처음에 match하는 위치에 할당하는 “first fit” allocation strategy이다.

page allocator는 fragmentation에 대해서, n개 이상의 pages가 free여도 분리되어 있어 연속된 n개의 pages를 할당하지 못할 수도 있다. 또한 pages는 interrupt context에서 allocate 될 수 없지만, free될 수는 있다. 만약 page가 freed되면 모든 bytes는 0xcc로 초기화되고, 이는 디버깅에 쓰인다.

- void palloc_init (size_t user_page_limit)

kernel page allocator를 set up한다. 최대 USER_PAGE_LIMIT의 page가 user pool에 들어간다.

다음의 page 관련 flag들이 있다.

```
/* How to allocate pages. */
enum palloc_flags
{
    PAL_ASSERT = 001,           /* Panic on failure. */
    PAL_ZERO = 002,             /* Zero page contents. */
    PAL_USER = 004              /* User page. */
};
```

palloc.h

PAL_ASSERT (001)	만약 page가 allocate되지 않았으면, panic the kernel. 이는 kernel initialization 동안에 쓰이고, user process들은 절대 panic the kernel이 허용되지 않아야 한다.
PAL_ZERO (002)	return하기 전에 할당된 페이지들의 모든 bytes를 0으로 만든다. 만약 이렇게 set 되지 않으면 새로 할당되는 페이지들의 내용은 예측 불가하다.
PAL_USER (004)	use pool에서 page를 얻는다. 만약 이 flag가 설정되지 않으면 page를 kernel pool에서 할당해오게 된다.

- palloc_get_multiple (enum palloc_flags flags, size t page_cnt)

```

void *
palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)
{
    struct pool *pool = flags & PAL_USER ? &user_pool : &kernel_pool;
    void *pages;
    size_t page_idx;

    if (page_cnt == 0)
        return NULL;

    lock_acquire (&pool->lock);
    page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);
    lock_release (&pool->lock);

    if (page_idx != BITMAP_ERROR)
        pages = pool->base + PGSIZE * page_idx;
    else
        pages = NULL;

    if (pages != NULL)
    {
        if (flags & PAL_ZERO)
            memset (pages, 0, PGSIZE * page_cnt);
    }
    else
    {
        if (flags & PAL_ASSERT)
            PANIC ("palloc_get: out of pages");
    }

    return pages;
}

```

page_cnt의 contiguous free pages를 얻고 리턴한다. 만약 할당되지 않으면 null pointer를 리턴한다. 만약 PAL_USER flag가 설정되어 있으면, user pool에서 얻어오고 아닌 경우 kernel pool에서 가져오는 것을 알 수 있다. 만약 PAL_ZERO가 설정되어 있으면, page들은 모두 zero로 설정된다. 만약 너무 적은 개수의 page가 가능한 상황에서는, null pointer를 리턴한다. (PAL_ASSERT가 세팅되어 있으면 kernel panic을 일으키게 된다.)

- palloc_get_page (enum palloc flags flags) → 단순히 palloc_get_multiple 함수에서 cnt를 1로 설정한다.
- palloc_free_multiple (void *pages, size t page_cnt)

```

void
palloc_free_multiple (void *pages, size_t page_cnt)
{
    struct pool *pool;
    size_t page_idx;

    ASSERT (pg_ofs (pages) == 0);
    if (pages == NULL || page_cnt == 0)
        return;

    if (page_from_pool (&kernel_pool, pages))
        pool = &kernel_pool;
    else if (page_from_pool (&user_pool, pages))
        pool = &user_pool;
    else
        NOT_REACHED ();

    page_idx = pg_no (pages) - pg_no (pool->base);

    #ifndef NDEBUG
    memset (pages, 0xcc, PGSIZE * page_cnt);
    #endif

    ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));
    bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
}

```

pages에서 시작하는 page_cnt의 contiguous page를 할당해제 시킨다. 이러한 page는 반드시 palloc_get_page or palloc_get_multiple 함수를 이용하여 얻어진 페이지들이어야만 한다.

- palloc_free_page (void *page) → 마찬가지로 palloc_free_multiple 함수에서 인자로 1을 넣는다.

2) Limitations and Necessity

- **the problem of original pintos**

virtual memory system을 구현하기에 정보가 불충분하다. 본 프로젝트를 통해서 최종적으로 virtual memory system을 구현해야하는데, 현재는 program의 숫자와 사이즈는 machine의 main memory size에 제한되어 있다. 이러한 제한을 극복하고 page table에서의 frame을 allocate, deallocate할 때 replacement policy을 올바르게 적용하기 위해서는 frame table과 같은 data structure가 필요하다.

- **the benefit of implementation**

이러한 frame table을 도입해야 하는 이유는 page replacement algorithm을 효과적으로 구현하기 위해서이다. 제한된 메모리 영역 내에서 새로운 page를 할당하려고 하면, 이미 모든 공간에 page가 할당되어 있는 경우에는 특정 page를 evict 시켜야 한다. 이러한 evict를 위한 page replacement policy는 여러 가지가 있는데, 우리는 이번 랩 과제에서 LRU approximate하는 clock algorithm을 구현하기 위해 필요한 정보를 담는 데에 frame table을 사용하고자 한다.

3) Blueprint: how to implement it

detailed data structures and pseudo codes

각 frame에 대한 정보를 관리하는 frame table을 생성해야 한다.
project3_requirements.pdf에 따르면,

- expected entry : **frame number, thread id, possibility of allocation**
 - main function
- 1) allocate/deallocate frames
 - 2) free frame이 존재하지 않을 때 occupying frame을 return할 victim을 선정하는 것
 - 3) user process(thread)에 의해 사용되는 frames을 찾는 것
- user pages에 의해 사용되는 frame은 “user pool”에 의해 할당된다.
 - userprog/process.c의 process loading을 수정하여(load_segment의 loop부분)
frame table을 관리해야 한다.

다음과 같은 함수들을 **pintos/src/vm/frame.c, h** 파일을 생성하여 정의할 것이다.

```
struct frame
{
    void *page_addr; // 할당되어있지 않으면 null pointer의 값이다.
    struct vm_entry *vme;
    struct thread *thread;
    struct list_elem ft_elem; // frame table의 element를 관리하기 위한 변수이다.
};
```

frame table에서는 physical page를 관리하기 위해 위와 같은 정보들이 필요하다.

page_addr	실제 physical frame addr를 가리키는 포인터 → allocation 여부를 알 수 있다.
vm_entry	이 frame과 매핑되어 있는 page를 의미
thread	이 frame을 가지고 있는 thread
ft_elem	frame table의 element를 관리하기 위한 변수

```
struct list frame_table; // frame table의 list 선언
struct lock frame_lock; // frame table에서 synchronization을 위한 lock variable 선언
struct list_elem *frame_clock; // page evict policy를 위한 변수이다, 이후 swap에서 고려
```

- **initialization**

```
void frame_table_init(void)
{
    list_init(&frame_table);
    lock_init(&frame_lock);
    frame_clock = NULL;
}
```

이후 이 frame_table_init 함수를 threads의 init.c의 main() 함수에서 호출해줄 것이다.

```
int main(void) {
    ...
    frame_table_init();
    ...
}
```

- **frame_insert**

frame table에 새로운 frame을 추가하는 함수이다.

```
void frame_insert(struct frame *frame)
{
    lock_acquire(&frame_lock);
    list_push_back(&frame_table, &frame->ft_elem);
    lock_release(&frame_lock);
}
```

- **frame_delete**

```
void frame_delete(struct frame *frame)
{
    // frame을 list로 관리할 것이므로 list_remove를 이용하여
    // list_remove(&frame->ft_elem);을 호출하여 삭제할 것이다.
    // 이때 이후의 pswap table에서 evict policy에 맞게 처리해 주어야 할 것이다.
}
```

- **alloc_frame**

frame을 할당하는 함수이다. 기존에는 palloc_get_page()만으로 page allocation이 구현되어 있었던 것을 해당 함수로 변경해준다. (setup_stack(), expand_stack(), handle_fault())

```

struct frame *alloc_frame(enum palloc_flags flags)
{
    // 1. frame 구조체 선언 및 malloc으로 할당
    struct frame *frame;
    frame = (struct frame *)malloc(sizeof(struct frame));
    if (!frame) return NULL;
    memset(frame, 0, sizeof(struct frame));

    // 2. 현재 thread의 정보 저장
    frame->thread = thread_current();
    // 3. flag에 따라서 page를 할당하려고 시도함
    frame->page_addr = palloc_get_page(flags);

    // 4. 만약 page가 제대로 할당되지 않은 경우에 처리
    while (!frame->page_addr)
    {
        // page를 할당 해제하고 다시 할당하려고 시도한다
        // 이를 위해 evict policy에 맞게 할당을 해제하는 함수를 만들어서 호출할 예정이다.
        // 해당 함수의 정확한 구현은 이후에 하겠다.
        frame->page_addr = palloc_get_page(flags);
    }

    // 5. 할당된 frame을 frame table에 insert하고 return
    frame_insert(frame);
    return frame;
}

```

- **free_frame**

할당된 page를 free시키는 함수이다. address를 받아서 frame table에서 frame을 찾는다. 찾은 frame을 frame table에서 제거하고, struct frame에서 할당했던 memory를 free 시켜 준다. 마찬가지로 기존 코드에서 palloc_free_page()만으로 free가 구현되어있던 부분을 free_frame로 변경한다.

```

void free_frame(void *addr)
{
    lock_acquire(&frame_lock);
    // 1. address를 받아서 frame table에서 frame을 찾는다.
    // 2. frame이 null이 아닌 경우에
    //     frame table에서 삭제, 할당했던 메모리를 free시킨다.
    lock_release(&frame_lock);
}

```

- **rationale**

- hash table을 사용하는 이유

hash table의 data structure은 find operation의 time complexity가 $O(n)$ 이다. 만약 frame table을 관리할 때 page fault가 발생하면 해당 address를 갖는 supplementary page table entry를 찾아야 하기 때문에 hash table은 이러한 부분에서 benefit이 있다.

- frame lock을 사용하는 이유

두 개 이상의 process에서 새로운 frame을 할당하려고 할 때의 synchronization을 보장하기 위함이다. race condition이 일어날 수 있기 때문에 한 번에 한 process만 frame을 allocation할 수 있다.

2. Lazy loading

1) Basics

- the definition or concept

page를 memory로 로딩하는 **lazy loading**을 구현해야 한다. 사용자가 사용할 부분만 load하고, 당장 사용하지 않는 부분은 load하지 않고 표시만 해두는 개념이다.

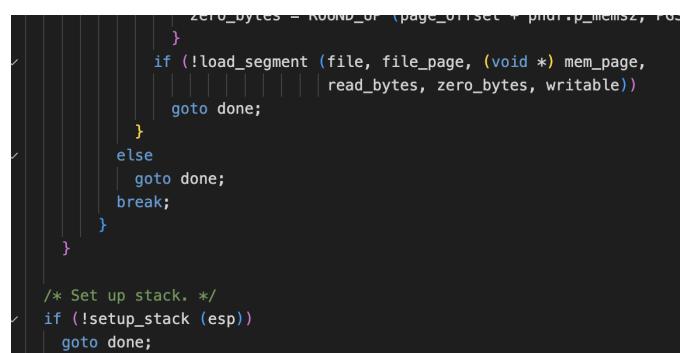
- Implementations in original pintos

process start를 위한 executable codes들은 memory에서 directly load되어야 한다. 만약 page fault가 일어나면, program execution은 이 상황을 invalid access error라고 판단하여 실행을 멈춘다.

현재 pintos에서 process가 시작될 때 memory loading 방식을 분석해보자. 앞의 project2에서 process execution sequence를 분석했었다. 이때 memory load의 단계로는

`process_exec() → load() → load_segment() → setup_stack()`

여기서 load 함수에서 다음 사진과 같이 load_segment()와 setup_stack()을 호출한다.



userprog/process.c/load function

- static bool **load_segment** (struct file *file, off_t ofs, uint8_t *upage, uint32_t read_bytes, uint32_t zero_bytes, bool writable)

FILE의 offset OFS에서 시작하는 segment를 address UPAGE에 load한다. read_bytes + zero_bytes의 virtual memory가 초기화 된다. upage에서의 read_bytes는 반드시 offset ofs에서 시작하는 file에서 read되어야 한다. upage+read_bytes의 zero_bytes는 반드시 zeroed되어야 한다.

이 함수에 의해 초기화되는 pages는 만약 writable이 true이면 반드시 writable이어야 하고, 아닌 경우에는 read-only이다. 성공적으로 실행되면 return true, mem allocation error 혹은 disk read error 발생 경우에는 false를 return한다.

- static bool **setup_stack** (void **esp)

user virtual memory의 top의 zeroed page에 minimal stack을 생성한다. 이 함수에서도 palloc_get_page, install_page와 같은 page에 관한 함수를 호출하여 사용한다.

```
case PT_LOAD:
    if (validate_segment (&phdr, file))
    {
        bool writable = (phdr.p_flags & PF_W) != 0;
        uint32_t file_page = phdr.p_offset & ~PGMASK;
        uint32_t mem_page = phdr.p_vaddr & ~PGMASK;
        uint32_t page_offset = phdr.p_vaddr & PGMASK;
        uint32_t read_bytes, zero_bytes;
        if (phdr.p_filesz > 0)
        {
            /* Normal segment.
             | | Read initial part from disk and zero the rest. */
            read_bytes = page_offset + phdr.p_filesz;
            zero_bytes = (ROUND_UP (page_offset + phdr.p_memsz, PGSIZE)
                         - read_bytes);
        }
        else
        {
            /* Entirely zero.
             | | Don't read anything from disk. */
            read_bytes = 0;
            zero_bytes = ROUND_UP (page_offset + phdr.p_memsz, PGSIZE);
        }
        if (!load_segment (file, file_page, (void *) mem_page,
                           read_bytes, zero_bytes, writable))
            goto done;
    }
```

load function

load 함수에서 program loading을 수행하는 코드의 부분이다. 여기서는 파일의 모든 segment를 순회하면서 data를 memory에 load한다. if (phdr.p_filesz > 0)에서 파일의 필요한 부분인지 판단하는 logic은 현재 구현되어 있지 않다.

2) Limitations and Necessity

- the problem of original pintos

현재 pintos에서는 실행에 필요한 모든 data를 physical memory에 load하기 때문에 비효율적이다.

- the benefit of implementation

이를 위한 해결 방안으로 Lazy Load를 구현하려 한다. Lazy Load는 필요한 부분만 load 함으로써 불필요한 메모리 사용을 줄일 수 있다.

Demand Paging

- Bring a page into memory only when it is needed.
 - Need less I/O and less memory
 - Faster response
 - More users (e.g. more processes)
- Page fault trap : Whenever CPU tries to access a page that is not swapped in, a page fault occurs.

Page fault handler needs to differentiate two cases :

- (1) illegal reference ? → abort
- (2) legal reference, but not-in-memory ? → bring to memory from disk
 - page table has “valid” bit (or “present” bit)
 - initially, invalid on all entries (demand paging)

- Works fine due to **locality of reference**

Lec9. Memory Management pdf

이는 수업시간에 배운 Demand Paging의 개념으로 설명할 수 있다. limited physical memory에서 각 process마다 virtual address를 infinite memory로 illusion 하는 방법 중 Demand Paging이 있다.

memory가 reference될 때, 필요할 때에만 page를 memory로 가져오는 것이다. 이전의 project2까지는 page fault가 error로 취급되었지만, project3에서는 page fault는 demand page를 의미한다. CPU가 page에 access할 때 page fault trap이 일어나고, 이때 page fault에서 legal reference인지 확인되면, 해당 memory를 Load 해온다.

따라서 다음과 같은 Benefit이 있다.

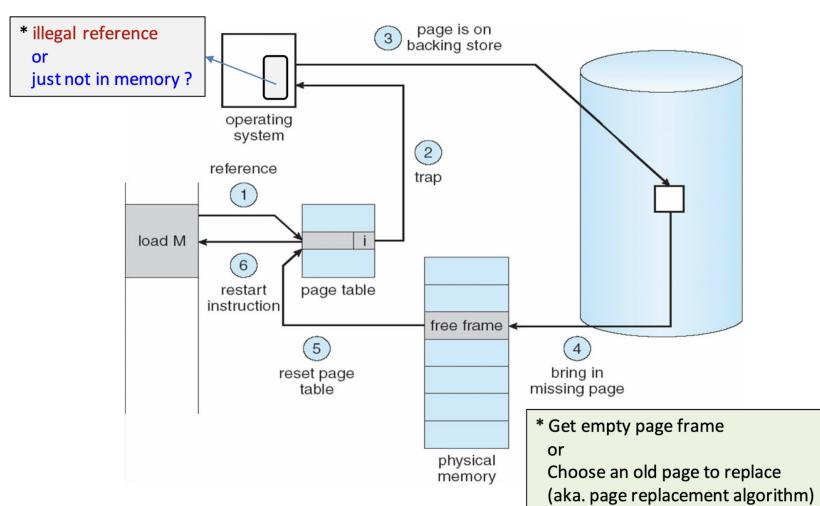
- 1) Need less I/O and less memory
- 2) Faster Response 가능
- 3) More processes가 동작 가능

3) Blueprint: how to implement it

detailed data structures and pseudo codes

page fault가 발생해도 process execution이 멈추지 않게 수정하는 것이 필요하다.
process start할 때 memory allocating을 위한 loading procedure 동안 stack setup part
만이 load 되도록 한다. (다른 part는 memory에 load되지 않는다. page들만 allocate된다.)
만약 page fault가 allocated page에서 발생하면, 이 page는 memory에 로드된다. page
fault handler는 이 procedure가 끝나면 process operation을 다시 시작해야한다.
page fault handler에서, I/O(lazy loading)과 I/O가 필요하지 않은 상황(wrong memory
access)이 동시에 일어나면, 후자의 상황이 전자를 기다리는 것보다 더 먼저 실행되어야 한
다.

Steps Handling a Page Fault Trap



Lec9. Memory Management

위의 demand paging 방법에 따라 우리 코드에서 구현할 내용을 단계별로 정리하면 다음과 같다.

- 1) 처음 process를 생성할 때에는 physical memory에 load하지 않는다. 이후 필요한 상황에만 로드하기 위해서이다.
- 2) 이후 process execute 함수에서 virtual address에 접근할 때 physical page가 매핑되어 있는지 확인한다. 만약 없다면 page table에서 page fault(trap)이 발생한다.
- 3) 이때 위의 사진처럼 page fault는 illegal reference 혹은 not in memory에 의해 발생 하므로 어떠한 상황인지 확인한다.

Page fault trap : Whenever CPU tries to access a page that is not swapped in, a page fault occurs.

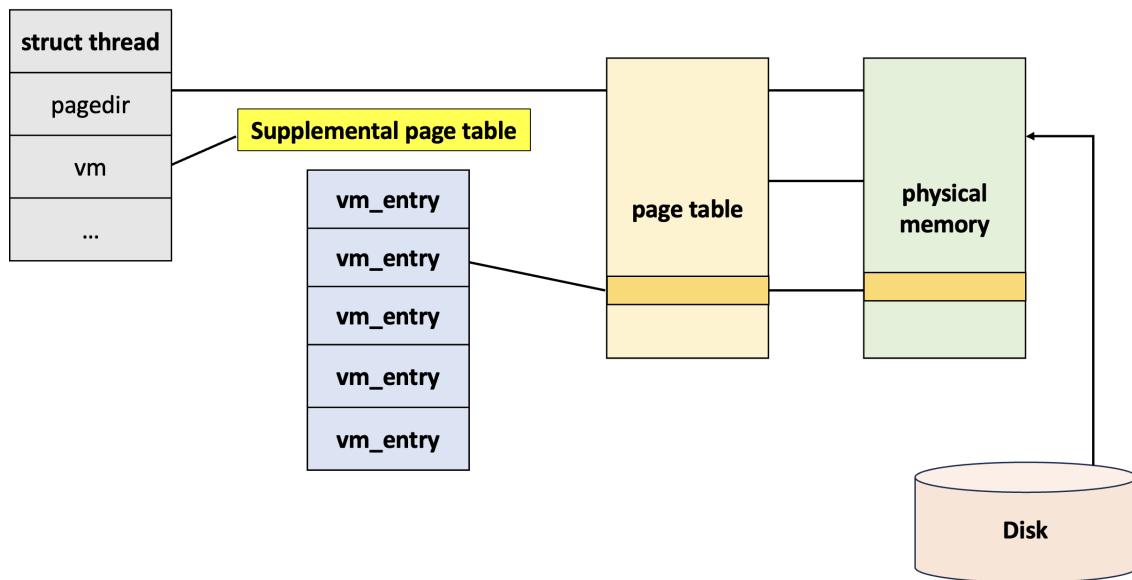
Page fault handler needs to differentiate two cases :

- (1) illegal reference ? → abort
- (2) legal reference, but not-in-memory ? → bring to memory from disk
page table has “valid” bit (or “present” bit)
initially, invalid on all entries (demand paging)

lec9 pdf

만약 illegal reference인 경우이면 lazy loading이 아니라 다른 처리를 해야할 것이다.

- 4) page fault handler를 통해서 접근했던 vaddr에 대한 vm_entry를 찾는다. 이때 frame table에서 vaddr에 대한 frame table entry를 찾아 정보를 얻는 함수를 생성할 것이다.
- 5) 해당 entry에 저장된 정보를 통해 data를 읽고 disk에서 physical frame에 load 한다.
- 6) fault 났던 page table에 physical frame으로의 mapping을 할당하기 위해 install_page 함수를 사용한다.



이러한 implementation을 위해서 page_fault() 함수의 동작을 분석해보자.

```

static void
page_fault (struct intr_frame *f)
{
    bool not_present; /* True: not-present page, false: writing r/o page. */
    bool write;        /* True: access was write, false: access was read. */
    bool user;         /* True: access by user, false: access by kernel. */
    void *fault_addr; /* Fault address. */

    asm ("movl %%cr2, %0" : "=r" (fault_addr));

    /* Turn interrupts back on (they were only off so that we could
     | be assured of reading CR2 before it changed). */
    intr_enable ();

    /* Count page faults. */
    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(!user) {
        f->error_code = 0;
        f->eip = (void (*) (void)) f->eax;
        f->eax = -1;
        return;
    }

    /* 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);
}

```

fault가 일어나면 CR2(Control Register 2)에 해당 주소를 저장하고, 이에 대한 정보들은 exception.h의 PF~ 매크로로 저장된다. 이때 project2까지는, page fault가 일어나면 에러를 확인한 후 무조건 kill 하는 동작을 하였다. 하지만 project3에서부터는 page fault가 에러를 뜻하는 것이 아니라, memory 관리 차원에서의 의도적인 동작이 일어나게 되므로 해당 부분을 수정해야 할 것이다.

- **page_fault 함수 수정**

```

static void
page_fault (struct intr_frame *f)
{
    ...
    void *fault_addr; /* Fault address. */
    asm ("movl %%cr2, %0" : "=r" (fault_addr));
    ...

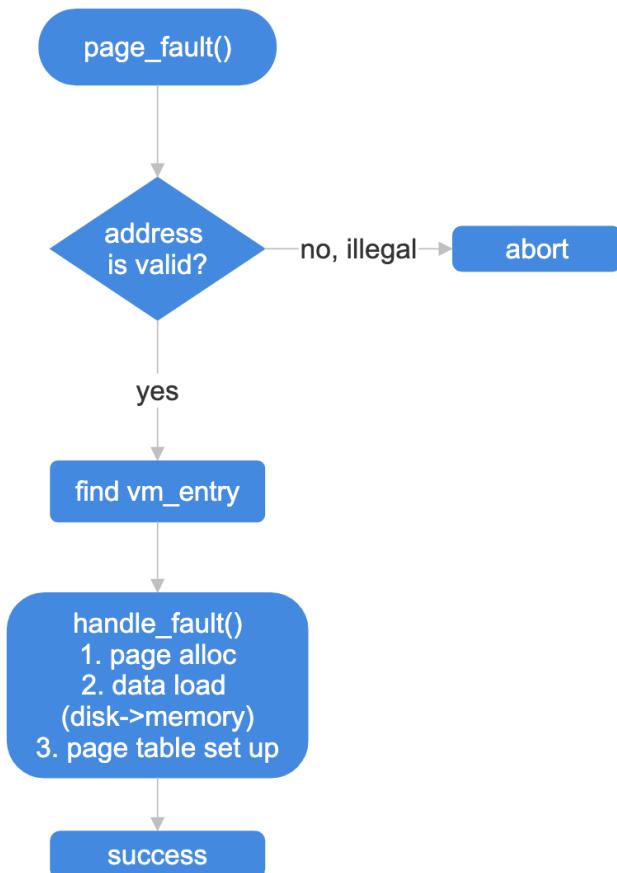
    //수정할 파트
    if(!not_present) exit(-1); // 1. read only 페이지에 대한 접근이 아니면 처리
    struct vm_entry *vme = vme_find(fault_addr); // 2. frame table에서의 정보 찾기
    // 3. 해당 entry의 유효성을 확인

```

```

    // 4. 새로 생성할 handle_fault()함수를 호출하여 이후의 과정 수행
    // 여기서 kill 하는 부분을 삭제
}

```



- **handle_fault** 함수 생성

physical page를 할당하고, 메모리를 로드하고, page table을 set up하는 동작을 처리하는 함수를 생성한다. vm entry에 따라 해당 page를 phys memory에 할당하고 mapping하는 과정에서 page load가 성공적으로 수행되면 true를 리턴한다.

```

bool handle_fault(struct vm_entry *vme)
{
    // 1. 물리 메모리 페이지를 할당 받음
    // 이때 PAL_USER flag를 사용해서 user pool에 할당

    // 2. 할당받은 page에 현재의 vme를 연결

    // 3. vme의 type에 따라 switch문으로 알맞게 처리한다.
    // VM_BIN에 대해서만 구현하고, 다른 type에 정확한 동작은 mmap file, swap에서 추후 고려한다.
    // VM_BIN일 경우 load_file 함수를 호출하여 physical mem에 load한다.
}

```

```

// 4. phys-virtual mem의 mapping 설정
// 이때 install_page를 이용
// mapping fail하면 page free, return false

// 5. 다 끝나면 vme의 is_loaded를 true로 만들고 return true
}

```

- **load_file** 함수 생성

Disk에 존재하는 page를 physical memory로 load하는 함수를 생성한다.

```

bool load_file (void* kaddr, struct vm_entry *vme)
{
    // file_read_at() 함수를 이용한다.
    // 1. file_read_at으로 physical page에 read_bytes만큼 데이터를 쓴다.
    // 2. file_read_at 여부 return한다.
    // 3. zero_bytes만큼 남는 부분을 '0'으로 채운다.
    // 4. 정상적으로 file을 loading 하면 return true
}

```

```

/* Reads SIZE bytes from FILE into BUFFER,
   starting at offset FILE_OFS in the file.
   Returns the number of bytes actually read,
   which may be less than SIZE if end of file is reached.
   The file's current position is unaffected. */

off_t
file_read_at (struct file *file, void *buffer, off_t size, off_t file_ofs)
{
    return inode_read_at (file->inode, buffer, size, file_ofs);
}

```

filesys/file.c/file_read_at

- **vme_find** 함수 생성

illegal reference or not in memory 확인하기 위해 vm에서 해당 주소를 찾는 함수를 생성 한다.

```

struct vm_entry *vme_find (void *vaddr)
{
    // 인자로 받은 vaddr에 해당하는 vm_entry를 찾아서 return한다.
    // (pg_round_down())을 이용하여 vaddr에 해당하는 page number을 추출한다.
    // hash_find() 함수를 이용하여 vm_entry를 찾아 리턴한다.
}

```

- **load_segment** 함수 수정

기존의 함수는 위에서 분석했듯이, page 할당, file을 page에 load하고 page를 process의 address space에 저장한다. 이를 vm_entry를 할당하고 초기화, vm table에 넣도록 수정한다.

```
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
    ...
    while (read_bytes > 0 || zero_bytes > 0)
    {
        size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
        size_t page_zero_bytes = PGSIZE - page_read_bytes;

        // 기존의 코드에서 phys mem 할당하고 mapping하는 파트 삭제
        // 1. vm entry를 생성 (malloc)
        // 2. vm_entry struct의 member variable 설정
        // 3. vme_insert()로 생성한 vm_entry를 추가
        read_bytes -= page_read_bytes;
        zero_bytes -= page_zero_bytes;
        upage += PGSIZE;
    }
    return true;
}
```

• setup_stack 함수 수정

기존에는 page 하나를 할당하고, install_page를 이용해 UPAGE에서 KPAGE로의 mapping을 page table에 추가하고 esp를 설정한다. 이것을 vm_entry를 생성하고 초기화, insert하도록 수정한다.

```
static bool
setup_stack (void **esp)
{
    ...
    if (kpage != NULL)
    {
        ...
    }
    // 1. vm_entry 생성
    // 2. vm_entry member variable 설정
    // 3. vme_insert()로 vm에 추가
    return success;
}
```

3. Supplemental page table

1) Basics

- the definition or concept

현재의 page table보다 더 많은 기능을 가지고 있는 Supplemental Page Table(S-page table)을 구현해야 한다. S-page table에서는 본 project3에서 구현하는 lazy loading, file memory mapping, swap table을 정상적으로 작동시킬 수 있어야 한다. Lazy load를 구현하기 위해서는 **어디에서 가져와야 하는지, 어떠한 데이터인지 등에 대한 page에 대한 정보를 관리하는 data structure**이 필요하다. 또한, kernel이 process가 종결될 때 supplemental page table을 보고 어떠한 resource가 free 될지 정한다. page fault가 발생했을 때 메모리가 필요하다는 것을 알고, 해당 정보를 찾아서 memory에 load해야 한다. 이를 supplemental page table에 저장하는 것이다.

supplemental page table은 terms of segments 또는 in terms of pages로 구현될 수 있다. page table을 supplemental page table의 요소를 위한 index로 사용할 수도 있다. 이를 위해 pagedir.c의 page table implementation을 수정할 수 있다. 하지만 Pintos pdf에서 이는 advanced students only로 권장하였기 때문에 기본 기능으로 구현 완료 되면 추후 고려해보려고 한다.

- page fault handler

project2에서, page fault는 kernel이나 user program의 bug로 인식하였다. 하지만 project3에서는 다르다. page fault는 file이나 swap에서 page를 가져와야 하는 것으로 인식된다.

→ exception.c의 page fault handler를 다음과 같이 수정하도록 하자.

1) supplemental page table에서 faulted page를 위치시킨다. 만약 mem ref가 valid하면 supplemental page table entry를 사용하여 page로 가야하는 data를 위치시킨다. (file system, swap slot, or all zero page여야 하는 것) 만약 sharing을 구현한다면, page 데이터는 page table이 아니라, 미리 page table에 있어야 할 것이다.

- 만약 이때 supplemental page table이 user process가 액세스하는 주소에서 아무 데이터를 기대하지 않거나, page가 kernel virtual memory에 놓여 있거나, read-only page에 액세스하여 write하려고 할 때 access는 invalid하다. invalid access에서는 process를 종결시키고, 자원들을 모두 할당 해제 시켜야 한다.

2) page를 저장하기 위한 frame을 얻어야 한다. 만약 sharing을 구현한다면, data는 이미 frame에 있어야 할 것이다.

3) file system이나 swap에서 읽거나 zeroing함으로써, frame으로 data를 fetch한다. data sharing을 구현하면, page는 이미 frame에 있어야 한다.

4) page fault가 발생한 virtual address에 대한 page table entry를 physical address에 point하도록 한다. → pagedir.c의 함수를 사용한다.

- **Implementations in original pintos**

현재의 Pintos에서는 이러한 기능을 하는 structure가 정의되어 있지 않다. 기존의 pintos code에서는 lazy loading이 아니라 필요한 모든 파일을 메모리에 로드하는 비효율적인 방식으로 구현되어 있다.

2) Limitations and Necessity

- **the problem of original pintos**

supplemental page table이 구현되어 있지 않기 때문에 page fault가 발생한 경우에도 어떠한 페이지에서 발생했는지, 해당 페이지에 대한 추가적인 정보를 알 수 없다. 그리고 process terminate 과정에서도 관련된 어떤 데이터를 free 시켜야하는지 알 수 없다.

- **the benefit of implementation**

supplemental page table은 다음과 같은 이점을 가진다. 정리하자면, lazy loading을 위해서 필요한 정보를 담기 때문에 supplemental page table을 통하여 lazy loading을 구현하고, 효율적인 메모리 관리를 할 수 있다. supplemental page table은 최소 2가지의 목적으로 사용된다.

- 1) page fault가 발생한 경우에 해당 page fault가 발생한 page를 찾을 수 있다. 또한 찾은 page의 여러 data에 대한 정보를 알 수 있다.
- 2) kernel이 process를 terminate시킬 때 exit하는 과정에서 해당 supplemental page table의 정보로부터, 어떠한 data들을 free 시켜야하는지 알 수 있다.

3) Blueprint: how to implement it

detailed data structures and pseudo codes

- expected entry : **page number, possibility of frame allocation, frame number, possibility of swap out...**
- main function

- 1) lazy loading에 의한 각 page에 대해 frame을 할당해야 한다.
- 2) frame의 modified data들은 file이나 swap disk에 저장한다.
- 3) page가 삭제될 때, 관련된 entries도 frame table, swap table 등에서 찾아서 삭제해야 한다.

우리는 **Hash table**을 이용하여 구현할 것이다. 보고서 윗 상단에서 pintos/src/lib/kernel/hash.h, hash.c에 대한 기존의 코드를 분석했는데, pintos에서 지원하는 해당 data structure를 이용할 것이다. 또한 requirement에 따르면 threads/exception.c의 page fault handler는 S-page table을 참조해야 한다.

또한 supplemental page table을 가장 중요하게 사용하는 것은 page fault handler이다. project2에서는 항상 page fault는 항상 bug를 의미했지만, project3에서는 이제 아니다. page fault는 page가 file이나 swap에서 가져와야함을 뜻할 수도 있다. 따라서 이러한 page fault handler를 구현해야 한다고 한다.

우리 프로젝트에서는 pintos pdf와 다르게 7 requirements대로 문제를 구분하여, 본 design report에서는 lazy loading에서 page_fault() 함수 수정을 이미 작성하였다. 3. supplemental page table에서는 spt의 구조를 구현하는 내용을 작성할 것이다.

supplemental page table의 각 entry에서 user program page 하나에 대응하는 정보를 담을 것이다. pintos pdf의 expected entry에 따라 Pintos에서 지원하는 hash table을 사용하여 구현할 것이다. 구체적인 구현 계획은 다음과 같다.

process 생성	vm을 초기화한다. process의 virtual page에 대한 vm entry를 생성하여 table에 추가한다.
process 실행 중	page fault가 발생한 경우에 (lazy loading 관련) 해당 주소에 해당하는 entry를 vm에서 찾아야 한다.
process 종료	vm에서 해당 entry를 위한 데이터를 삭제한다.

- `struct thread`: thread마다 virtual space를 관리할 것이므로 struct thread에 vm 변수를 추가한다.
 - hash table로서 supplemental page table을 구현하고자 하여, 이를 앞으로 vm이라고 지칭하겠다.
 - `struct hash vm;`

```
struct vm_entry
{
    uint8_t type; //VM_BIN, VM_FILE, VM_ANON의 타입
    void vaddr; // virtual page number /
    bool writable; // write permission
    bool is_loaded; // physical memory의 load 여부 flag
    struct file file; // mapping된 파일
    size_t offset; // read file offset
    size_t read_bytes; // virtual page에 쓰여져 있는 byte 수
    size_t zero_bytes; // 0으로 채울 남은 페이지의 byte 수
    struct hash_elem elem; // Hash Table element
    struct list_elem mmap_elem; // mmap list element
```

```
    size_t swap_slot;  
}
```

- **vm_init**

```
void vm_init (struct hash *vm)  
{  
    // hash_init() 함수를 사용하여 vm entry를 initialization한다.  
}
```

- **vm_hash**

```
static unsigned vm_hash (const struct hash_elem *e, void *aux)  
{  
    //hash_entry()로 elem에 대한 vm_entry 구조체를 검색  
    // → hash_int()를 이용해서 vm_entry의 vaddr에 대한 해시 값을 구해 return 한다.  
}
```

- **vm_less**

```
static bool vm_less  
(const struct hash_elem *a, const struct hash_elem *b, void *aux)  
{  
    // 입력된 두 hash_elem의 vaddr 비교하여 a의 vaddr이 b보다 작을 시 true를,  
    // a의 vaddr이 b보다 클 시 false를 return한다.  
}
```

hash table에서 elem을 sorting하기 위해서 필요하다.

- **start_process()**

pintos/src/userprog/process.c의 start_process()에서 vm_init()을 호출하여 vm을 초기화 해야한다.

그리고 이러한 vm entry를 추가하고 삭제하는 함수가 필요하다.

- **vme_insert**

```
bool vme_insert (struct hash *vm, struct vm_entry *vme)  
{  
    // hash_insert() 함수로 인자로 넘겨 받은 vme를 vm entry에 삽입하고 성공하면 true를 리턴한다.  
}
```

- **vme_delete**

```

bool vme_delete (struct hash *vm, struct vm_entry *vme)
{
    // hash_delete() 함수로 인자로 받은 vme를 vm_entry에서 삭제하고 성공하면 true를 리턴한다.
}

```

- **vm_destroy**

```

void vm_destroy (struct hash *vm)
{
    // hash_destroy() 함수를 사용하여 해시 테이블의 버킷리스트와 vm_entry들을 삭제하는 함수다.
    // 만약 load가 되어 있는 page의 vm_entry인 경우 page의 free와,
    // page mapping을 palloc_free_page()와
    // pagedir_clear_page() 사용해서 해제해준다. 그리고 vm_entry의 할당도 해제한다.
}

```

4. Stack growth

1) Basics

- **the definition or concept**

stack이란?

process에서 function call과 local variable 저장에 사용되는 memory 영역이다.
project2에서는 stack이 single page로 제한되어 있었고 program은 그 stack 이상의 영역에 대해서는 제한되었다. 지금부터는 stack은 현재 사이즈에서 추가적으로 page를 할당함으로써 grow할 수 있다. 다만 stack access로 보이는 경우에만 추가적인 페이지를 할당해야 한다.

How to get value of stack pointer

User program의 stack pointer의 현재 값을 얻어야 할 것이다. user program에 의해 발생되는 page fault or system call에서, syscall_handler() or page_fault()의 인자로 struct intr_frame이 넘어오게 된다. 이 intr_frame 구조체의 member esp의 값으로서 stack pointer 값을 받을 수 있다.

User-provided pointer의 Valid 여부 확인

Project2에서 user-provided pointer의 valid 여부를 확인하는 두 가지 방식에 따라서 접근이 달라진다.

1) 만약 user pointer로 접근하기 전에 valid 여부를 판단하면, 이 경우만 handle하면 될 것이다.

2) 만약 invalid memory access를 page fault에서 detect하도록 설계한 경우 page fault가 kernel에서 발생하는 경우에 대해서도 handle해야 한다. exception이 user → kernel mode로 switch할 때 processor는 stack pointer만을 저장하기 때문에, page_fault()에서 전달된 struct intr_frame의 esp 값을 읽으면 user stack pointer가 아닌 정의되지 않은 값이 될 것이다. 따라서 user에서 kernel mode로 switch될 때 esp를 struct thread에 저장하는 방식과 같이 다른 방식으로 해결해야 할 것이다.

- **Implementations in original pintos**

기존에 Page fault는 다음과 같이 구현되어 있다.

```
static void
page_fault (struct intr_frame *f)
{
    bool not_present; /* True: not-present page, false: writing r/o page. */
    bool write;        /* True: access was write, false: access was read. */
    bool user;         /* True: access by user, false: access by kernel. */
    void *fault_addr; /* Fault address. */

    asm ("movl %%cr2, %0" : "=r" (fault_addr));

    /* Turn interrupts back on (they were only off so that we could
     | be assured of reading CR2 before it changed). */
    intr_enable ();

    /* Count page faults. */
    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(!user) {
        f->error_code = 0;
        f->eip = (void (*)(void)) f->eax;
        f->eax = -1;
        return;
    }

    /* 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);
}
```

fault가 일어나면 CR2(Control Register 2)에 해당 주소를 저장하고, 이에 대한 정보들은 exception.h의 PF~ 매크로로 저장된다. 이후 여러 원인을 확인하고 kill하는 동작을 하였다. 우리는 여기서 stack 범위를 넘어가는 access의 경우 stack 확장을 하는 코드를 추가하여야 한다.

2) Limitations and Necessity

- the problem of original pintos

현재는 stack의 size가 4KB(1 page)로 fix되어 있다. 만약 할당된 stack의 범위를 벗어나게 되면, 즉 stack의 크기를 초과하는 주소에 접근하게 되면 pintos가 종료되게 된다. 그렇기 때문에 현재 구현은 program이 stack 사이즈에 대해 제한적이라는 문제가 존재한다.

- the benefit of implementation

이번 project3은 stack의 범위를 벗어나게 될 때 stack의 크기를 늘릴 수 있도록 수정한다. 이로써 program이 stack 사이즈에 대해 제한적이었던 문제를 해결할 수 있다.

3) Blueprint: how to implement it

detailed data structures and pseudo codes

실제 구현에서는 현재 stack size를 초과하는 주소에 접근할 때 page fault handler에서 stack 확장이 필요한지 결정해야 한다. stack 확장은 최대 8MB까지 확장할 수 있기 때문에, interrupt frame에서 esp값을 가져와 현재 stack pointer로부터 limit 이내에 포함되는 access는 유효한 stack 접근으로 간주하여 stack을 확장한다. 반면에 limit을 넘어가는 access의 경우 segment fault를 발생시킨다.

먼저 `page_fault()` 를 수정하여 page fault handler에서 stack 범위를 넘어가는 access에 대해 stack 확장을 하게 하는 코드를 작성한다.

```
static void page_fault (struct intr_frame *f) {
    ...
    struct vm_entry *vme = vme_find (fault_addr);
    if (!fte)
    {
        if (!check_stack_boundary(fault_addr, f->esp)) exit(-1);
        if (!expand_stack(fault_addr)) exit(-1);
        return;
    }
    ...
}
```

vmd_find()로 찾은 vme가 NULL일 경우 `check_stack_boundary()` 을 통해서 address가 stack 영역에 포함되어 있는지 확인하고, stack 영역에 있지 않으면 exit(-1)을 호출하게 한다. stack 영역에 있는 address인 것이 확인되면 `expand_stack()`을 호출하여 stack을 확장 한다. stack 확장에 실패할 경우 exit(-1)을 호출하게 한다.

다음으로는 `check_stack_boundary()` 함수를 구현하여 주어진 address가 valid한 stack 영역에 있는지 판별하게 한다.

```
bool check_stack_boundary(uint32_t addr, void *esp)
{
    uint32_t base = 0xC0000000;
    uint32_t limit = 0x80000000;
    uint32_t lowest_stack_addr = base-limit;

    if (!is_user_vaddr(addr) | addr < lowest_stack_addr | addr < esp-32)
        return false;
    else
        return true;
}
```

주어진 address가 user address 영역인지, stack의 가능한 가장 낮은 주소를 넘어가는 address인지, esp-32보다 더 작은 address인지 확인한다.

esp-32보다 더 작은 address인지 확인하는 이유는 stack에 항상 최소한의 버퍼 공간을 유지하기 위함이다.

- 신호나 interrupt가 발생하면 이들의 handler는 stack에 data를 push하는데, 이때 user program이 현재 stack pointer 아래의 stack에 쓰는 것은 버그로 간주된다. 즉 이 논리대로면 `addr < esp` 를 검사하는 것이 맞다.
- 그러나 80x86 `PUSH` instruction은 stack pointer를 감소시킨 후 데이터를 쓰며, stack pointer를 조정하기 전에 접근 권한을 확인한다. 이는 PUSH가 stack을 새 page로 확장할 때 page fault가 발생할 수 있음을 의미하며 stack pointer 아래 4byte에서 발생할 수 있다.
- 마찬가지로 `PUSHA` instruction은 32byte를 한 번에 푸시할 수 있으므로, stack pointer 아래 32바이트에서 페이지 폴트가 발생할 수 있다.
- 따라서 `addr < esp - 32` 를 확인하는 것은 스택에 항상 최소한 32바이트의 buffer가 있는지를 보장하여 `PUSHA` 와 같은 명령어가 오류 없이 제대로 작동할 수 있게 한다.

마지막으로 stack을 확장하는 `expand_stack()` 함수를 구현한다.

```
bool expand_stack(void *addr)
{
    struct frame *kpage;
    void *upage = pg_round_down(addr);
    bool success = false;

    // 1. user mode용 page 할당하고 0으로 초기화
```

```

kpage = alloc_page(PAL_USER | PAL_ZERO);
if (kpage == NULL) return false;

// 2. 할당된 kernel page를 upage에 mapping
success = install_page(upage, kpage->kaddr, true);
if (!success)
{
    free_page(kpage->kaddr); // page 할당 해제
    return false;
}

// 3. 해당하는 vm entry 생성
// vm entry를 생성 (malloc)
// vm_entry struct의 member variable 설정

// 4. 해당 vm_entry를 insert

return true;
}

```

5. File memory mapping

1) Basics

- **the definition or concept**

file을 memory mapping하는 것은 process address space에 file을 mapping하는 것을 말한다. 기존에 file에 read하거나 write하기 위해서는 read(), write()라는 시스템콜을 사용하였는데 file memory mapping을 하게 되면 시스템콜 대신 메모리 접근을 통해 파일을 접근할 수 있다.

mmap system call로 file을 virtual pages로 map하도록 구현한다. 따라서 program은 memory instruction을 file data에 directly 사용할 수 있게 한다.

다음은 mmap를 사용하여 console에 file을 출력하는 program이다. command line에 specified된 file을 open하고 이를 virtual address에 mapping한다. 이후에 mapped data를 console에 write하고 file을 unmap한다.

```

#include <stdio.h>
#include <syscall.h>
int main (int argc UNUSED, char *argv[])
{
    void *data = (void *) 0x10000000;      /* Address at which to map. */

    int fd = open (argv[1]);                /* Open file. */
    mapid_t map = mmap (fd, data);         /* Map file. */
    write (1, data, filesize (fd));        /* Write file to console. */
    munmap (map);                         /* Unmap file (optional). */
    return 0;
}

```

memory mapping은 mmap()와 munmap() system call을 통해서 구현되며, file들의 page들은 lazy loading 방식으로 load된다. mapping된 이후에 mapped page가 dirty bit이 setting되면 이는 load된 이후에 write된 적이 있으므로 file에 write back을 한다. memory mapped file에 의해 사용된 memory를 추적할 수 있어야 한다. 그리고 mapped region에서의 page fault를 적절히 관리하고, mapped files이 다른 프로세스의 segment와 겹치지 않도록 주의해야 한다.

- **Implementations in original pintos**

기존에 file에 read, write를 하기 위해서는 read(), write() system call을 호출해야 한다.

```

int
read (int fd, void *buffer, unsigned size)
{
    return syscall3 (SYS_READ, fd, buffer, size);
}

int
write (int fd, const void *buffer, unsigned size)
{
    return syscall3 (SYS_WRITE, fd, buffer, size);
}

/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
   ARG2, and returns the return value as an `int'. */
#define syscall3(NUMBER, ARG0, ARG1, ARG2) \
({ \
    int retval; \
    asm volatile \
        ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; " \
         "pushl %[number]; int $0x30; addl $16, %%esp" \
         : "=a" (retval) \
         : [number] "i" (NUMBER), \
           [arg0] "r" (ARG0), \
           [arg1] "r" (ARG1), \
           [arg2] "r" (ARG2) \
        )
}

```

```

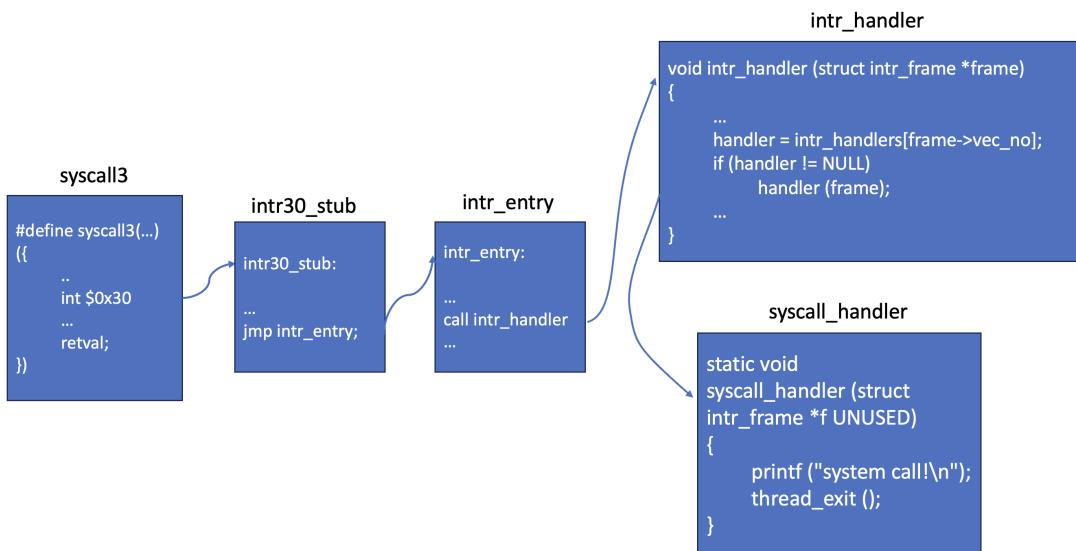
        : "memory");
    retval;
})

```

매크로함수 `syscall3`는 인자와 `number`을 모두 stack에 push한 후에는 `int $0x30;` 라는 명령어를 실행하는데, `int $0x30;` 는 30번째 interrupt를 발생시키는 assembly 명령어다.

`int`라는 명령어는 interrupt 명령어를 인자로 사용하여 interrupt를 발생시키는데 이때 인자가 `0x30`이 되는 것이다. 즉, 이를 통해 CPU에게 Interrupt가 발생했음을 알리고, 해당 interrupt에 맞는 interrupt service routine이 실행된다. 여기서는 `syscall`에 대해 특정 interrupt service routine으로 `syscall handler`가 불려야 하고, 이러한 정보는 interrupt descriptor table이라고 하는 IDT에 저장되어 있다.

위에서 설명했던 `int $0x30;` 이 수행되면 `intr30_stub` 가 먼저 수행되는데 각 `intr(NUMBER)_stub`함수는 `intr-stubs.S` 파일에 정의되어 있다. `intr30_stub` 가 수행된 후 `intr_entry`로 jump하여 수행하고 그 후 `Interrupt handler` 함수를 호출한다. `interrupt handler` 함수에서는 interrupt vector 번호를 사용하여 `intr_handler` 배열에서 해당 `interrupt handler` 함수를 찾아 해당 함수를 호출한다.



interrupt stub 함수는 각 interrupt에 대한 초기 처리를 수행하는 역할을 한다. 즉, interrupt service routine의 시작 부분에서 handling을 위한 준비 과정을 거치는 함수다. `STUB` 매크로 함수는 특정 interrupt number에 대해 Interrupt stub을 생성한다. interrupt stub code를 생성하고 이 stub의 주소를 `intr_stubs` 배열에 저장한다. STUB

는 인자로 주어진 TYPE에 따라 `zero` 혹은 `REAL` 매크로 함수를 실행하여 각 interrupt에 대한 초기 처리를 수행한다.

다음으로 `intr_stub` 이 끝나며 `intr_entry` 가 수행된다. interrupt를 발생시킨 함수(caller)의 register를 모두 stack에 push하고, kernel environment를 set up한 후에 interrupt handler를 호출한다.

먼저 caller의 %ds, %es, %fs, %gs라는 segment register와 general purpose register을 모두 push(pushal)함으로써 원래의 system 상태를 저장하여 interrupt handler 실행이 완료된 후 원래 상태로 복구할 수 있도록 한다. 다음으로는 kernel 환경을 설정하는데 segment register들을 초기화하고, `struct intr_frame` 을 저장할 stack frame을 설정한다. Kernel environment 설정까지 끝난 후에는 현재 stack pointer인 %esp를 push하고 `intr_handler` 함수를 호출한다.

`intr_handler` 는 external interrupt인지 internal interrupt인지 먼저 확인함으로써 각 type에 맞게 handling을 진행한다.

- 먼저 external interrupt인 경우 interrupt가 비활성화되어 있는지 확인하고 `intr_context()` 함수를 호출하여 현재 Interrupt context에서 실행 중인지 확인한다. 이를 통해 중복 interrupt 처리를 방지한다. 다음으로 `in_external_intr` 변수를 true로 설정하고, `yield_on_return` 변수를 false로 설정함으로써 external interrupt 처리 중임을 system에게 알린다. 다음으로는 interrupt handler 배열에서 handler를 찾아서 호출하고 처리한다. 이후 interrupt 처리가 완료되면 다시 한번 interrupt가 비활성화되어 있는지 확인하고 현재 Interrupt context에서 실행 중인지 확인한다. 그리고 external interrupt 처리가 완료되었음을 알리고, 변수가 true로 되어 있으면 `thread_yield()` 함수를 호출하여 양보하며 다른 thread에게 실행 제어를 전달한다.
- 만약 internal interrupt인 경우 `intr_handlers[frame->vec_no]` 을 통해 해당 interrupt 번호에 대한 handler 함수를 찾고 함수가 있으면 해당 handler 함수를 호출하여 interrupt를 처리한다. 이외에 interrupt 번호가 0x27이거나 0x2f인 경우에는 잘못된 interrupt이므로 무시하며 예상치못한 interrupt가 발생했을 시 `unexpected_interrupt` 함수를 호출하여 처리한다.

system call handler의 경우 `intr_handlers[frame->vec_no]` 에는 `syscall_handler` 함수가 저장되어 있기에 해당 함수가 불리면 system call handling이 수행된다. system call handling 동작은 project2에서 구현한 바에 따라 동작한다.

2) Limitations and Necessity

- the problem of original pintos

지금은 file memory mapping이 구현되어 있지 않다. 즉 mmap(), munmap() system call이 없다. 따라서 file에 read, write를 하기 위해서는 무조건 read, write system call

을 호출해야 한다. 이러한 접근 방식은 효율성 면에서 제한적이며, 특히 대용량 파일 작업이나 빈번한 파일 액세스가 필요한 애플리케이션에서 성능 저하를 초래할 수 있다. 또한, 이는 프로그램이 파일 데이터를 자신의 주소 공간에 직접 맵핑할 수 없다는 것을 의미하여, 메모리 사용의 최적화 및 직관적인 파일 처리에 제약을 가합니다.

- **the benefit of implementation**

mmap(), munmap() 함수를 구현함으로써 file memory mapping을 가능하게 할 것이다. 이를 통해 file 내용을 process의 virtual memory address space에 직접 맵핑할 수 있게 되어, 파일 데이터에 대한 접근이 더욱 효율적으로 이루어질 수 있다. file memory mapping은 program이 file을 마치 memory의 일부인 것처럼 다룰 수 있게 해주어, 복잡한 file 입출력 작업을 단순화하고 성능도 향상된다. 특히나 대용량 file을 처리할 때나 file을 자주 access하는 경우 더욱 유용할 것이다.

3) Blueprint: how to implement it

detailed data structures and pseudo codes

모든 mapping들은 process exit할 때 implicitly unmapped된다. 그리고 이때 process에서 write된 모든 페이지들은 file에 written back되어야 한다. 그 이후에 process의 virtual page list에서 page들을 제거해야 한다.

closing이나 removing file은 mapping을 unmap 하지 않는다. 한 번 create된 이후에는 mapping은 munmap이 호출되거나 process exit이 되기 전까지는 항상 valid하다. 그리고 각 mapping에 대해 file에 대해 분리, 독립적으로 참조하는 함수를 위해 `file_reopen`을 사용해야 한다.

- `struct file * file_reopen (struct file *file)`

기존 파일과 동일한 inode에 대해 새 파일을 열고 반환하는 함수로 실패할 경우 NULL을 return한다. 성공하는 경우에는 file_open 함수의 return 값과 같이 file 구조체를 반환한다.

만약 같은 file에 두 개 이상의 process를 맵핑하면, 무조건 consistent data를 봐야할 필요는 없다. unix에서는 2개 이상의 맵핑이 같은 physical page를 공유하도록 만들지만, mmap system call에서는 client가 그 페이지가 shared인지 private인지를 (즉, copy-on-write) 특정지을 수 있게 하는 argument를 가진다.

- `mapid_t mmap (int fd, void *addr)`

- fd에 해당하는 open된 file을 process의 virtual address space에 mapping하는 함수다. 전체 file은 주어진 addr부터 시작하는 연속적인 virtual page들에 mapping된다.

- virtual memory system은 mmap 영역의 페이지를 **lazy loading**을 통해 구현해야 하며 mapping을 위한 backup store은 mmap된 file 자체다. 즉 mmap에 의해 mapping된 page가 삭제될 때는 mapping된 file에 다시 작성된다.
- 파일 길이가 `PGSIZE`의 배수가 아닌 경우, 최종 mapping된 page의 일부 바이트는 파일 끝을 넘어간다. 이러한 byte들은 page가 파일 시스템에서 fault 처리 때 0으로 설정되어야 하며, page가 disk에 다시 쓰여질 때 버려져야 한다.
- file mapping을 성공할 경우, 이 함수는 프로세스 내에서 맵핑을 고유하게 식별하는 `mapping ID`를 반환한다. 실패할 경우, 유효하지 않은 mapping ID로 간주되는 -1을 반환해야 하며, 프로세스의 mapping은 변경되지 않아야 한다.
- file mapping에 실패하는 경우
 - fd로 열린 파일의 길이가 0바이트인 경우
 - `addr`이 page 정렬되지 않은 경우
 - mapping된 page 범위가 기존에 mapping된 page 세트, stack 또는 실행 시에 mapping된 page를 포함하여 겹치는 경우
 - `addr`이 0인 경우
 - 콘솔 입력 및 출력을 나타내는 파일 디스크립터 0과 1은 맵핑할 수 없다.
- `void munmap (mapid_t mapping)`
 - `mmap`을 통해 same process에 의해 return된 mapping ID로 지정된 mapping을 취소한다. 이 때 해당 mapping은 아직 취소되지 않은 mapping이어야 한다.
- 위 mmap와 munmap을 구현하기 위해서는 mapping된 file들의 정보를 관리할 수 있는 data structure을 선언할 것이다.

```
struct mmap_file {
    mapid_t mapid;
    struct file* file;
    struct list_elem elem;
    struct list vme_list;
};
```

- mapid : file mapping에 대한 식별자 역할을 하는 id로, mmap() success시 return 하는 값이다.
- file : mapping하는 file의 file Object.
- elem : mmap_file structure을 list로 관리하기 위해 list_elem을 선언한다.

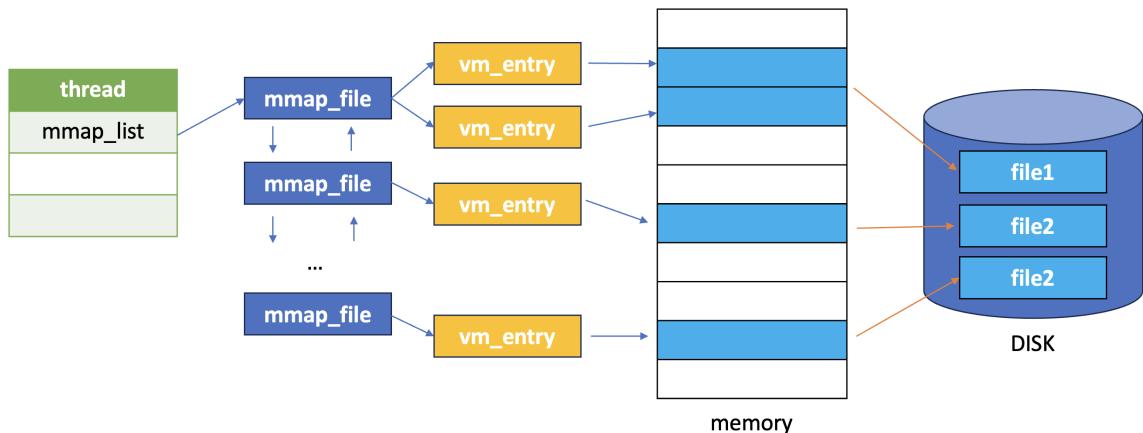
- struct thread에서 memory에 Load한 file을 관리해야 하기 때문에 mmap_list를 선언하여 관리하게 할 것이다.

```

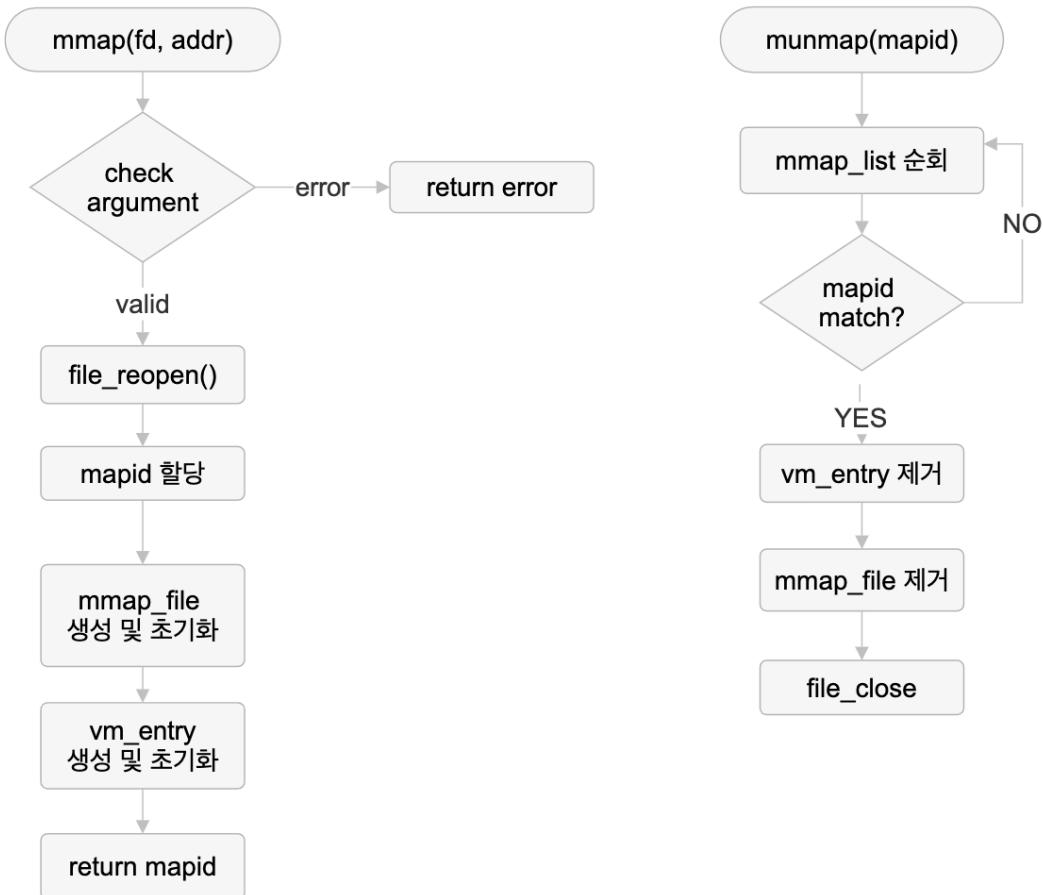
    struct thread
    {
    ...
    struct list mmap_list;
    int mmap_next;
    ...
    };
  
```

- vme_list : 이 structure에 대응되는 모든 vm_entry의 list.

위의 구현 방식을 그림으로 나타내면 다음과 같다.



- 마지막으로 **mmap()**와 **munmap()** 함수의 수도코드를 작성하였다.



- `mapid_t mmap (int fd, void *addr)`

```

mapid_t mmap (int fd, void *addr)
{
    // 1. mmap_file 구조체 생성 및 메모리 할당
    struct mmap_file *mfe = (struct mmap_file *)malloc(sizeof(struct mmap_file));
    if (!mfe) return -1;

    // fd에 해당하는 file reopen

    list_init(&mfe->vme_list);
    while()// file 다 읽을 때 까지 반복
    {
        // 1. vm entry 할당
        // 2. vme_list에 mmap_elem과 연결된 vm entry 추가
        // 3. current thread에 대해 vme insert

        // 4. file addr, offset 업데이트 (page size만큼)
        // 5. file에 남은 길이 업데이트 (page size만큼)
    }
    mfe->mapid = thread_current()->mmap_next++;
    slist_push_back(&thread_current()->mmap_list, &mfe->elem);
}

```

```
    return mfe->mapid;
}
```

- `void munmap (mapid t mapid)`

```
void munmap (mapid t mapid)
{
    // 1. thread의 mmap_list에서 mapid에 해당하는 mfe 찾기
    struct mmap_file *mfe = NULL;
    struct list_elem *e;
    for (e = list_begin(&thread_current()->mmap_list);
        e != list_end(&thread_current()->mmap_list); e = list_next (e))
    {
        mfe = list_entry (e, struct mmap_file, e);
        if (mfe->mapid == mapid) break;
    }

    // 2. mfe 없으면 종료

    // 3. 해당 mfe의 vme_list를 돌면서 vme를 지우기
    for (e = list_begin(&mfe->vme_list); e != list_end(&mfe->vme_list);)
    {
        struct vm_entry *vme = list_entry(e, struct vm_entry, mmap_elem);
        vme->is_loaded = false;
        ele = list_remove(ele);
        vme_delete(&thread_current()->vm, vme);
    }

    // 4. mfe를 mmap_list에서 제거
    list_remove(&mfe->elem);
    // 5. mfe 구조체 자체를 free
    free(mfe);
}
```

6. Swap table

1) Basics

- **the definition or concept**

Swapping이란?

Physical memory가 가득찼을 때 더이상 새로운 frame을 allocate할 수 없다. 이러한 상황에서 기존에 있던 frame들 중 하나를 evict하는 것을 swapping mechanism이라고

한다. 이렇게 기존에 physical memory의 content를 disk에서 memory로 load하거나 반대로 memory에서 disk로 evict하는 기법을 말한다.

- swap-out : memory 상의 content를 disk의 swap 영역으로 방출
- swap-in : swap-out된 page를 다시 memory로 load

Swap Partition

Pintos는 swap partition을 swap space로 제공한다. swap partition은 4MB이며 4KB로 나누어 관리한다. 우리는 이 partition의 frame들을 연결 해주기 위해 list type으로 swap table을 관리하고자 한다. 또한, 사용 가능한 swap partition을 관리하기 위해 bitmap data structure을 사용하고자 한다.

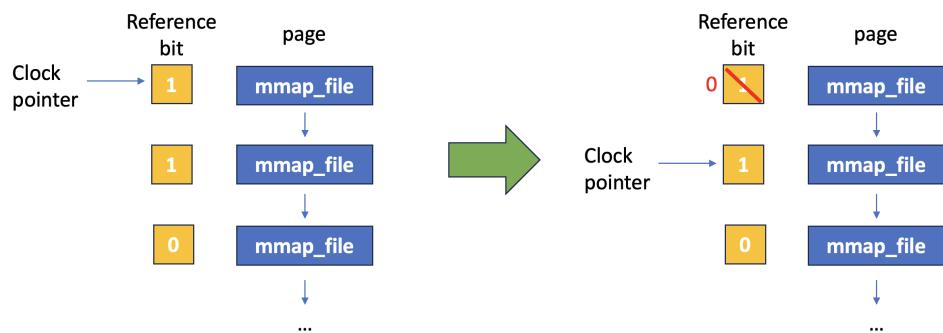
Replacement policy

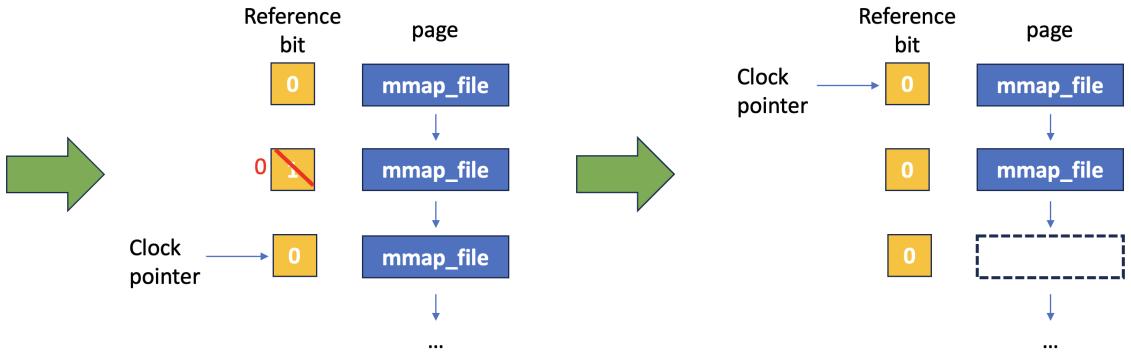
Physical memory에 load되어 있는 frame들 중 하나를 evict할 때 어떤 frame을 evict 할 것인가에 대한 policy다. FIFO, LRU 등 다양한 알고리즘이 존재한다. 본 과제에서는 clock algorithm을 사용하고자 한다.

Clock algorithm

Clock algorithm은 pointer를 시계방향으로 이동시키면서 교체될 frame을 선정하는 것 이 main idea다. 이때 frame을 순회하면서 현재 pointer가 가리키고 있는 page의 access bit를 확인하여,

- access bit이 1이면 access bit을 0으로 재설정하고 그 다음을 가리키고,
- access bit이 0이면 해당 frame을 victim으로 선정한다.





How to manage accessed and dirty bits

80x86 hardware는 각 page의 page table entry에 있는 2 bit을 통해 page replacement algorithm 구현을 지원한다. page에 대한 read or write가 발생할 때마다 CPU는 page의 page table entry의 accessed bit은 1로 설정되고, write가 발생할 때마다 dirty bit가 1로 설정된다. HW는 accessed bit나 dirty bit을 다시 0으로 reset할 수 없다. 즉, 0으로 reset하는 것은 결국 OS의 역할이 되는 것이다.

또한, 2개 이상의 page들이 같은 frame을 참조하는 **aliases**에 대해서도 고려해줘야 한다. aliased frame이 access될 때 accessed, dirty bits는 only one page table entry에서만 업데이트 되고, 다른 aliases들에 대해서는 업데이트되지 않는다.

Pintos에서는 모든 user virtual page이 kernel virtual page에 aliased 되어 있기 때문에 이러한 경우를 무조건 manage해야 한다. 즉 both address에 대해 accessed, dirty bit을 모두 check하고 update해야 한다. 또는 kernel이 user virtual address를 통해서만 user data를 accessing하는 방법을 사용해도 해당 문제를 해결할 수 있다.

- **Implementations in original pintos**

본 조는 swapping을 구현하기 위해 pintos에서 제공하는 bitmap과 block을 사용하고자 한다. 따라서 디자인에 앞서 bitmap과 block이 어떻게 구현되어 있는지 살펴보자.

1. bitmap

- **data structure**

```
struct bitmap
{
    size_t bit_cnt;      /* Number of bits. */
    elem_type *bits;    /* Elements that represent bits. */
};
```

bitmap은 0, 1 bit를 사용해서 데이터를 표현하는 방법으로 bitmap의 각 bit는 일반적으로 켜짐/꺼짐, 사용됨/ 사용되지 않음과 같은 상태와 속성을 나타낼 때

사용된다.

bitmap structure의 구조를 살펴보면

`bit_cnt` : bitmap의 전체 bit 수를 저장한다.

`bits` : 실제 bitmap의 bit가 저장되는 배열을 가리키는 pointer.

```
typedef unsigned long elem_type;  
  
/* Number of bits in an element. */  
#define ELEM_BITS (sizeof (elem_type) * CHAR_BIT)
```

여기서 `elem_type`이 `unsigned long type`이기 때문에 32비트 정수라면 `bits` 배열의 각 요소는 32비트를 저장할 수 있다.

`ELEM_BITS` 는 `elem_type`이 포함할 수 있는 bit수를 말한다.

- **functions**

- `elem_idx`

해당 bit가 몇번째 `elem_type`에 위치하는지를 반환한다.

- `bit_mask`

`bit_idx`에 해당하는 단일 비트만 켜진 `elem_type`값을 반환한다.

- `elem_cnt`

주어진 bit 수를 저장하기 위해 필요한 `elem_type` 개수를 반환한다.

- `byte_cnt`

주어진 bit수를 저장하기 위해 필요한 byte 수를 반환한다.

- `last_mask`

bitmap b의 마지막 `elem_type`에서 실제 사용되는 bit를 나타내는 mask를 반환한다.

- `bitmap_create`

주어진 비트 수만큼의 공간을 가진 새로운 비트맵을 생성하고, 그에 대한 포인터를 반환한다. `bitmap_set_all`을 호출하여 모든 bit를 false로 초기화 한다. 메모리 할당에 실패하면 null 포인터를 반환한다.

- `bitmap_create_in_buf`

이미 할당된 memory block내에서 `bit_cnt` bit를 가진 bitmap을 생성한다. block size는 `bitmap_needed_bytes(BIT_CNT)`보다 커야한다.

- bitmap_buf_size
 - bit_cnt bit를 저장할 수 있는 bitmap에 필요한 byte 수를 반환한다. 즉 bitmap 구조체 자체의 크기와 bit_cnt라는 비트맵이 저장할 비트 총 수를 더해서 총 byte수를 반환한다.
- bitmap_destroy
 - bitmap_create함수로 생성된 bitmap의 메모리를 해제한다. bitmap의 bits 배열과 bitmap 자체를 해제한다.
- bitmap_size
 - bitmap의 bit_cnt를 반환한다.
- bitmap_set
 - bitmap b의 idx위치에 있는 bit를 value로 설정한다. value가 true면 bitmap_mark를 호출하여 비트를 true로 설정하고 value가 false면 bitmap_reset을 호출하여 비트를 false로 설정한다.
- bitmap_mark
 - 비트맵 b의 bit_idx 위치에 있는 비트를 true로 설정한다. 인라인 어셈블리를 사용하여 해당 비트를 true로 설정한다.
- bitmap_reset
 - 비트맵 b의 bit_idx 위치에 있는 비트를 false로 설정한다. 인라인 어셈블리를 사용하여 해당 비트를 false로 설정한다.
- bitmap_flip
 - 비트맵 b의 bit_idx 위치에 있는 비트를 반전시킨다. 인라인 어셈블리를 사용하여 해당 비트의 값을 반전시킨다.
- bitmap_test
 - 비트맵 b의 bit_idx 위치에 있는 비트를 반환한다. 해당 bit의 위치를 계산하고, 비트 마스크를 사용하여 해당 bit의 값을 테스트한다. 비트가 설정되어 있으면 true, 아니면 false를 반환한다.
- bitmap_set_all
 - 비트맵 b의 모든 bit를 value로 설정한다. bitmap_set_multiple 함수를 호출하여 전체 bitmap 범위에 대해 value를 설정한다.
- bitmap_set_multiple

비트맵 b에서 시작 위치 start부터 cnt개의 비트를 value로 설정한다. 주어진 범위에 대해 반복하여 bitmap_set을 호출하며 각 bit를 value로 설정한다.

- bitmap_count

지정된 범위(start부터 start+cnt까지)에서 특정 value로 설정된 bit 수를 세어 반환한다. 주어진 범위를 순회하면서, 각 비트가 value와 일치하는지 확인하고, 일치하는 비트의 수를 세어 반환한다.

- bitmap_contains

지정된 범위 내에 최소 하나의 비트가 value 값으로 설정되어 있는지를 확인한다. 주어진 범위를 순회하면서 value 값으로 설정된 bit가 있는지 확인하고 발견되면 true를 반환한다. 발견되지 않으면 false를 반환한다.

- bitmap_any

지정된 범위 내에 최소 하나의 bit가 true로 설정되어 있는지 확인한다.

`bitmap_contains` 함수를 호출하여 true 값으로 설정된 비트가 있는지 확인한다.

- bitmap_none

지정된 범위 내에 모든 bit가 false로 설정되어 있는지 확인한다.

`bitmap_contains` 함수를 호출하여 true 값으로 설정된 비트가 없는지 확인한다.

- bitmap_all

지정된 범위 내에 모든 bit가 true로 설정되어 있는지 확인한다.

`bitmap_contains` 함수를 호출하여 false 값으로 설정된 비트가 없는지 확인한다.

- bitmap_scan

지정된 범위 내에서 연속적으로 value 값으로 설정된 cnt 개의 bit를 찾아 그 시작 index를 반환한다. 주어진 범위를 순회하면서 연속적으로 value 값으로 설정된 bit 그룹을 찾는다. 찾으면 그 시작 인덱스를 반환하고, 찾지 못하면 BITMAP_ERROR를 반환한다.

- bitmap_scan_and_flip

지정된 범위 내에서 연속적으로 value 값으로 설정된 cnt 개의 비트를 찾아, 그 비트들을 !value로 반전시키고, 시작 인덱스를 반환한다. 즉, `bitmap_scan` 함수를 사용하여 연속적인 비트 그룹을 찾는다. 찾으면 해당

비트들을 반전시키고 시작 인덱스를 반환한다. 찾지 못하면 BITMAP_ERROR를 반환한다.

- bitmap_file_size

bitmap b를 file에 저장하는 데 필요한 byte 수를 반환한다.

- bitmap_read

파일로부터 비트맵 b를 읽어들인다. 비트맵에 저장된 bit 수가 0보다 큰 경우 필요한 byte 수를 계산한다. file_read_at 함수를 이용하여 file에서 bitmap data를 읽어온다.

- bitmap_write

파일에 bitmap b를 쓴다. 필요한 byte수를 계산한다. file_write_at 함수를 사용하여 bitmap data를 file에 쓴다. 쓰기 작업의 성공 여부를 반환한다.

- bitmap_dump

bitmap b의 내용을 16진수 형태로 콘솔에 출력한다. hex_dump함수를 사용하여 bitmap의 bit 배열을 16진수 형태로 출력한다.

2. block

struct block은 pintos에서 block device를 나타내는 구조체다. block device는 컴퓨터에서 data를 고정된 크기의 block이나 sector 단위로 저장하고 접근하는 저장 장치의 일종이다. 그리고 이러한 block은 일반적으로 수백에서 수천 바이트의 크기를 가진다.

- data structure

```
/* A block device. */
struct block
{
    struct list_elem list_elem;           /* Element in all_blocks. */

    char name[16];                      /* Block device name. */
    enum block_type type;               /* Type of block device. */
    block_sector_t size;                /* Size in sectors. */

    const struct block_operations *ops;   /* Driver operations. */
    void *aux;                          /* Extra data owned by driver. */

    unsigned long long read_cnt;        /* Number of sectors read. */
    unsigned long long write_cnt;       /* Number of sectors written. */
};
```

`struct list_elem list_elem`: struct block 인스턴스들을 리스트로 관리하기 위해 사용된다. 즉, 블록 디바이스들을 전역적으로 관리하는 연결 리스트에 이 구조체를 추가하거나 제거하는 데 사용된다.

`char name[16]`: 블록 디바이스의 이름을 저장하고, 이 이름은 디바이스를 식별하는 데 사용된다. 블록 디바이스에 고유한 이름을 할당하며, 시스템 내에서 디바이스를 구분하는 데 사용된다.

`enum block_type type`: 블록 디바이스의 타입을 나타낸다. `enum block_type`은 디바이스의 종류를 나타내는 열거형이다. 시스템이 디바이스의 종류에 따라 다르게 동작해야 할 때 이 정보를 사용한다.

`block_sector_t size`: 블록 디바이스의 전체 크기를 섹터 단위로 저장한다. 디바이스의 총 용량을 계산하거나 특정 섹터에 접근할 때 사용된다.

`const struct block_operations *ops`: 디바이스 드라이버의 연산을 정의하는 `block_operations` 구조체에 대한 포인터다. 이 구조체는 디바이스에 대한 기본 연산들(read, write 등)을 정의한다. 시스템이 블록 디바이스를 조작할 때 이 포인터를 통해 해당 연산을 수행한다.

`void *aux`: 드라이버나 시스템이 사용할 수 있는 추가 데이터를 저장하기 위한 포인터다. 이 포인터는 여러 용도로 사용될 수 있으며, 구체적인 사용 방법은 드라이버나 시스템에 따라 다르다. 드라이버나 시스템이 필요에 따라 추가적인 데이터를 저장하거나 접근할 때 사용된다.

`unsigned long long read_cnt`, `unsigned long long write_cnt`: 각각 디바이스에서 수행된 섹터 읽기 및 쓰기 작업의 횟수를 저장한다. 시스템이 디바이스의 I/O 성능을 모니터링하거나 분석할 때 이 데이터를 사용한다.

- block sector size

```
#define BLOCK_SECTOR_SIZE 512
```

block device의 sector 크기가 512byte이다. 즉 한 sector에 최대 512 byte의 data를 저장할 수 있다는 것을 의미한다. sector는 data를 읽고 쓰는 최소 단위로 각 sector는 독립적으로 주소가 지정되어 저장 장치는 이 주소를 사용하여 sector 단위로 데이터를 읽거나 쓴다.

- type of block device

```

/* Type of a block device. */
enum block_type
{
    /* Block device types that play a role in Pintos. */
    BLOCK_KERNEL,           /* Pintos OS kernel. */
    BLOCK_FILESYS,          /* File system. */
    BLOCK_SCRATCH,          /* Scratch. */
    BLOCK_SWAP,              /* Swap. */
    BLOCK_ROLE_CNT,         

    /* Other kinds of block devices that Pintos may see but does
     * | | not interact with. */
    BLOCK_RAW = BLOCK_ROLE_CNT, /* "Raw" device with unidentified contents. */
    BLOCK_FOREIGN,           /* Owned by non-Pintos operating system. */
    BLOCK_CNT                 /* Number of Pintos block types. */
};

```

- **functions**

- `block_type_name`

```

const char *
block_type_name (enum block_type type)
{
    static const char *block_type_names[BLOCK_CNT] =
    {
        "kernel",
        "filesystem",
        "scratch",
        "swap",
        "raw",
        "foreign",
    };

    ASSERT (type < BLOCK_CNT);
    return block_type_names[type];
}

```

주어진 block device type에 대해 human-readable name을 return한다.

- `block_get_role`

주어진 role에 해당하는 block device를 반환한다. `block_by_role` 배열에서 해당 role에 할당된 block device를 찾아 반환하고 해당 role에 device가 할당되지 않았다면 NULL을 반환한다.

- `block_set_role`

특정 role에 block device를 할당한다. `block_by_role` 배열에 해당 role에 대한 block device를 설정한다.

- `block_first`

첫번째 block device를 반환한다. `all_blocks` 리스트의 첫번째 요소를 찾아 해당 block device를 반환한다. 등록된 block device가 없는 경우 NULL을 return한다.

- `block_next`

주어진 block 다음에 오는 block device를 반환한다. block의 `list_elem`을 사용하여 `all_blocks` 리스트에서 다음 요소를 찾아 반환한다. block이 리스트의 마지막 요소인 경우 NULL을 반환한다.

- `block_get_by_name`

주어진 name에 해당하는 block device를 반환한다. `all_blocks` 리스트를 순회하면서 각 block device의 이름을 입력된 이름과 비교한다. 일치하는 이름을 가진 block device를 찾으면 해당 device를 반환한다. 일치하는 이름을 가진 device가 없는 경우 NULL을 반환한다.

- `check_sector`

주어진 sector가 block의 유효한 범위 내에 있는지 확인한다. sector 값이 block의 크기보다 크거나 같으면 잘못된 접근이므로 시스템에 panic을 발생시킨다.

- `block_read`

block의 sector에서 data를 읽어 buffer에 저장한다. 먼저 `check_sector`을 호출하여 요청한 sector가 유효한지 확인하고 해당 block의 읽기 연산을 호출하여 data를 buffer로 읽어온다. 후에 `read_count`를 증가시킨다.

- `block_write`

buffer의 data를 block의 sector에 쓴다. `check_sector`을 호출하여 요청된 sector가 유효한지 확인한다. 해당 block의 write 연산을 호출하여 data를 sector에 쓴다. `write_cnt`를 증가시킨다.

- `block_size`

block의 size를 반환한다.

- `block_name`

block의 name을 반환한다.

- `block_type`

block의 type을 반환한다.

- `block_print_stats`

모든 block device의 read write statistics를 출력한다. `block_by_role` 배열을 순회하며 각 block device의 statistics(name, type, read/write 횟수)를 출력한다.

- `block_register`

새로운 block device를 시스템에 등록한다. 새로운 block 구조체를 할당하고 초기화하고, block의 이름, 탑입, 크기, 연산, 추가 데이터를 설정한다. all_blocks 리스트에 추가한다.

- list_elem_to_block

list_elem을 struct block으로 변환한다. list_elem이 all_blocks 리스트의 끝이 아니라면 해당 요소를 struct block으로 변환하여 반환한다.

3. How to manage accessed and dirty bits

아래는 pintos에서 각 bit를 수정하고 얻는 함수다.

- pagedir_is_dirty

pd의 vpage가 dirty이면 true를 return한다. 이는 pte가 설정된 이후에 해당 page가 수정된 적이 있다는 뜻이다. 그리고 pd에 vpage에 대한 pte가 없는 경우에도 null을 return한다.

- pagedir_set_dirty

pd의 vpage를 위한 pte의 dirty bit을 설정하는 함수다.

- pagedir_is_accessed

pd의 vpage에 대한 pte에서 최근에 accessed 한 적 있으면 true를 return한다. 즉, pte가 설정되고 pte가 마지막으로 cleared된 시점 사이에 accessed 된 기간을 의미한다. 만약 vpage를 위한 pte를 pd가 포함하지 않는다면 fail을 return한다.

- pagedir_set_accessed

pd의 virtual page vpage를 위한 pte의 accessed bit를 설정한다.

2) Limitations and Necessity

- the problem of original pintos

현재 swap 기능이 존재하지 않는다. 따라서 새로운 frame을 할당하려고 해도 physical memory에 free space가 없으면 할당할 수 없는 문제가 존재한다. 결국 physical memory가 제한되어 있기 때문에 시스템의 효율성이 떨어지고, 메모리 사용의 최적화가 어렵다. 특히나 메모리 요구가 높은 process가 실행될 때 시스템의 성능이 심각하게 저하될 수 있다.

- the benefit of implementation

따라서 우리는 swapping을 구현함으로써 physical memory에서 frame을 clock algorithm에 따라 evict하여 새로운 free space를 만들고 새로운 frame을 allocate할 수 있게 한다. 이를 통해 새로운 프레임을 할당할 수 있는 여지가 생긴다. clock

algorithm을 사용하는 swap mechanism은 덜 자주 사용되는 데이터를 disk와 같은 보조 저장 장치로 이동시켜 main memory를 효과적으로 관리한다. 이로 인해 시스템의 전반적인 메모리 관리가 개선되며, 더 많은 프로세스와 user program을 동시에 실행할 수 있는 능력이 향상될 수 있다.

3) Blueprint: how to implement it

detailed data structures and pseudo codes

swapping을 위해서는 swap disk와 swap table이 필요하다. swap table은 in-use, free swap slot을 track하게 해야 한다. 또한 frame에서 swap partition으로 evict함으로써 unused swap slot을 고를 수 있게 한다. 이 때 evict할 frame을 정하는 replacement policy를 설계해야 한다. replacement policy 중에는 LRU algorithm, clock algorithm 등이 있는데, 본 과제에서는 clock algorithm을 사용하고자 한다. 그리고 swap된 page를 가지는 process가 종료되거나 page가 read back될 때 swap slot을 free시켜야 할 것이다.

우리는 swapping을 위해 BLOCK_SWAP type의 block device를 사용하고, block_get_role()을 호출함으로써 struct block을 얻을 것이다.

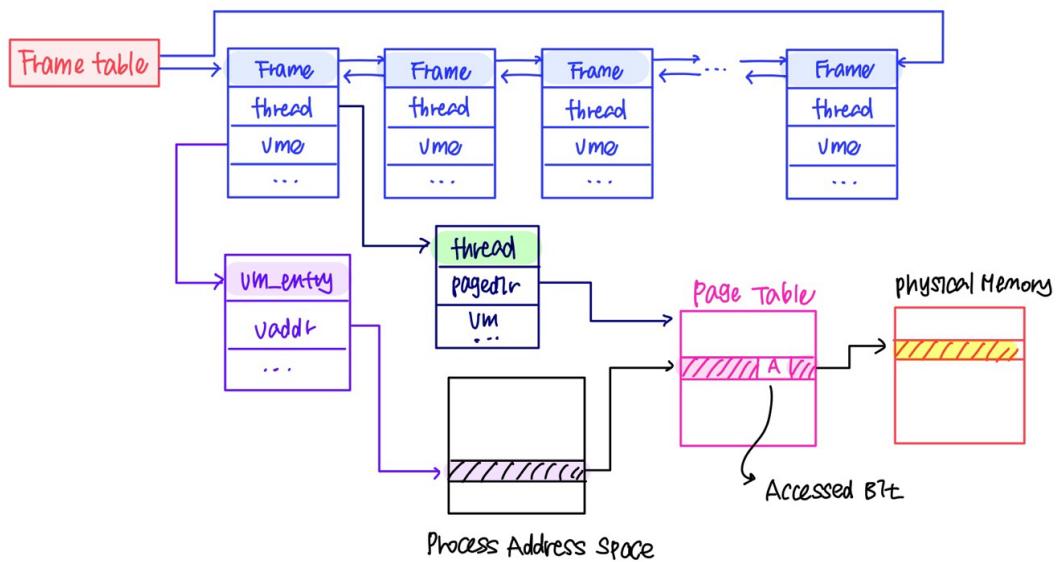
또한, 우리는 swap partition의 frame들을 연결 해주기 위해 list type으로 swap table을 관리하고자 한다. 또한, 사용 가능한 swap partition을 관리하기 위해 bitmap data structure를 사용하고자 한다.

Data structure

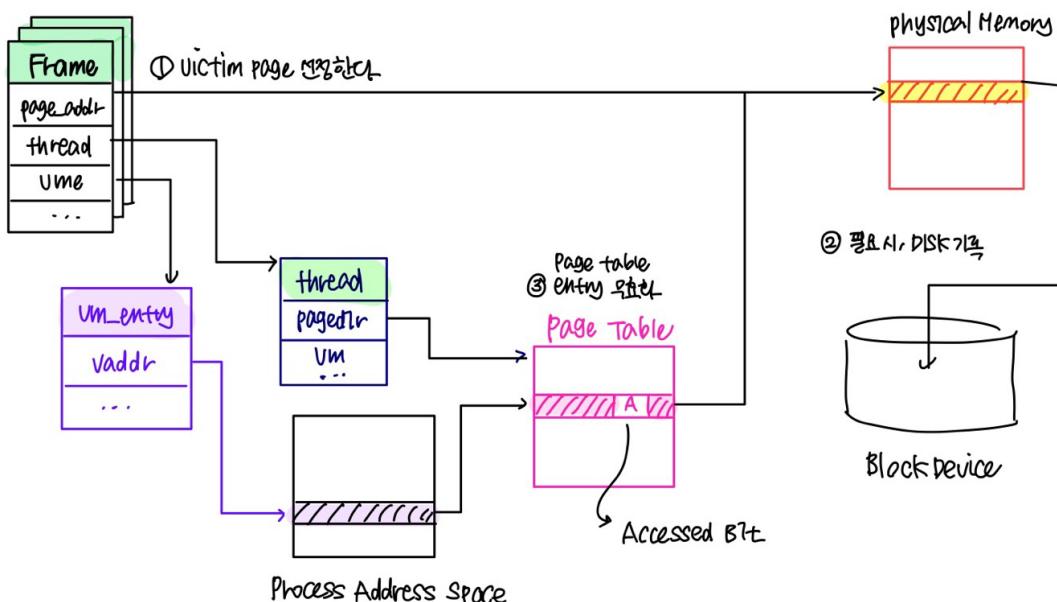
```
struct frame
{
    void *page_addr;
    struct vm_entry *vme;
    struct thread *thread;
    struct list_elem ft_elem;
};
```

```
struct list frame_table; // frame table의 list 선언
struct lock frame_lock; // synchronization을 위한 lock variable 선언
struct list_elem *frame_clock; // page evict policy를 위한 변수
```

우리가 사용하고자 하는 data structure은 frame table이다. frame table은 아래 그림과 같이 frame 구조체들을 list로 관리한다. 위에서 frame table과 vm_entry를 정의한 것을 아래 그림에서 함께 정리하였다.



따라서 swapping mechanism을 다음과 같이 구현하고자 한다.



Initialization

```
void frame_table_init(void)
{
    list_init(&frame_table);
    lock_init(&frame_lock);
```

```
    frame_clock = NULL;  
}
```

frame_table, frame_lock, frame_clock을 초기화하는 함수다.

```
void swap_init()  
{  
    // 1. lock initialization  
    // 2. bitmap initialization  
    // 3. block initialization
```

swapping은 pintos에서 제공하는 bitmap과 block을 사용하여 구현할 것이다.
swapping을 위해 swap_init은 bitmap과 block을 초기화해준다.

```
int main(void) {  
    ...  
    swap_init(); // swap table initializatoin  
    frame_table_init(); // frame table initialization  
    ...  
}
```

위 frame_table_init과 swap_init 함수를 threads의 init.c의 main() 함수에서 호출해줄 것이다.

Managing bitmap_swap, block_swap

swapping에서의 synchronization을 위해 lock variable lock_swap을 선언하여 사용할 것이다.

- **swap_in**

```
void swap_in(size_t slot_index, void *kaddr)  
{  
    lock_acquire(&lock_swap); // lock acquire  
    // 1. block read를 통해 swap slot에서 data 가져오기  
    // 2. slot_index 부분에 false라고 세팅  
    lock_release(&lock_swap); // lock release  
}
```

`block_read (struct block *block, block_sector_t sector, void *buffer)` 을 호출하여 swap slot에서 data를 가져오게 하는 것이다. data를 가져온 후에는 slot에 false로 즉 free space로 setting해주어야 한다.

- **swap_out**

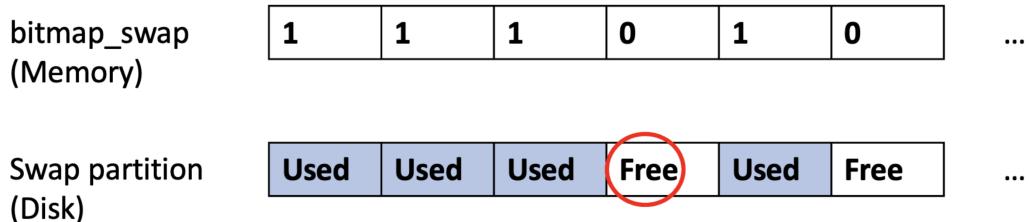
```

size_t swap_out(void *kaddr)
{
    lock_acquire(&lock_swap);
    // 1. 사용 가능한 swap slot의 index를 찾고 해당 slot을 true로 표시(사용 중으로 표시)
    // 2. block write 사용해서 memory에서 swap slot으로 데이터 쓰기
    lock_release(&lock_swap);
    // 3. swap한 index 반환
    return index;
}

```

사용 가능한 swap slot의 index를 `bitmap_scan_and_flip` 함수를 사용하여 찾는다. 즉, 인자에 따라 첫번째로 사용 가능한 즉 false인 slot을 찾고 그 bit를 true로 바꿈으로써 사용 중인 곳으로 바꾼다. 다음으로 해당 swap slot index에 memory에 있는 data를 쓴다. 마지막으로 write이 끝나면 swap slot의 index를 반환한다.

우리는 여기서 first fit 알고리즘을 사용하여 탐색하고자 한다.



Managing frame_table

- **frame_insert**

frame table에 새로운 frame을 추가하는 함수이다.

```

void frame_insert(struct frame *frame)
{
    lock_acquire(&frame_lock);
    list_push_back(&frame_table, &frame->ft_elem);
    lock_release(&frame_lock);
}

```

- **frame_delete**

frame table에서 frame을 delete하는 함수다. 이 때 elem을 삭제할 때는 `list_remove`을 쓰며 frame이 `frame_clock`에 해당하는 경우 해당 `frame_clock`을 다음 element로 업데이트 해주는 과정이 필요하다.

```
void frame_delete(struct frame *frame)
{
    if (frame_clock == &frame->ft_elem)
        frame_clock = list_remove(frame_clock);
    else
        list_remove(&page->ft_elem);
}
```

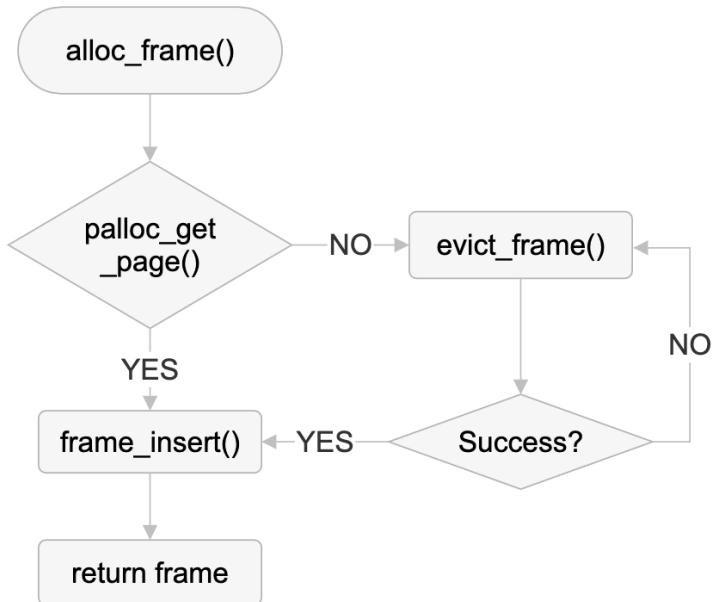
- **frame_find**

frame table에서 addr에 해당하는 frame을 찾아서 return하는 함수다.

```
struct frame *frame_find(void *addr)
{
    struct list_elem *e;
    for (e = list_begin(&frame_table); e != list_end(&frame_table); e = list_next(e))
    {
        struct frame *frame = list_entry(e, struct frame, ft_elem);
        if (frame->page_addr = addr)
            return frame;
    }
    return NULL
}
```

Replacement Policy Implementation

- **alloc_frame**



frame을 할당하는 함수이다. palloc_get_page를 통해 page를 할당하게 하고 성공할 때 까지 while문을 통해 할당한다. 실패할 때마다 evict_frame을 호출하여 evict하는 과정을 거친다. 이후에 frame 할당에 성공한 경우 frame을 frame table에 insert하고 frame을 return한다.

```

struct frame *alloc_frame(enum palloc_flags flags)
{
    // 1. frame 구조체 선언 및 malloc으로 할당
    struct frame *frame;
    frame = (struct frame *)malloc(sizeof(struct frame));
    if (!frame) return NULL;
    memset(frame, 0, sizeof(struct frame));

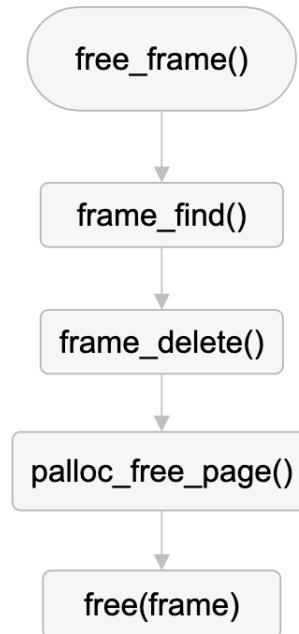
    // 2. 현재 thread의 정보 저장

    // 3. flag에 따라서 page를 할당
    frame->page_addr = palloc_get_page(flags);
    // 만약 page가 제대로 할당되지 않은 경우에 처리
    while (!frame->page_addr)
    {
        // evict 해주고 다시 할당 시도
        evict_frame();
        frame->page_addr = palloc_get_page(flags);
    }

    // 4. 할당된 frame을 frame table에 insert하고 return
    frame_insert(frame);
    return frame;
}

```

- **free_frame**



할당된 page를 free시키는 함수이다. `addr`에 해당하는 frame을 frame table에서 찾는다. 찾은 frame을 frame table에서 제거하고, struct frame에서 할당했던 memory를 free 시켜준다. 마찬가지로 기존 코드에서 `palloc_free_page()`만으로 `free`가 구현되어있던 부분을 `free_frame`로 변경한다.

```

void free_frame(void *addr)
{
    lock_acquire(&frame_lock);

    struct frame *frame = frame_find(addr);
    if (frame != NULL)
    {
        pagedir_clear_page(frame->thread->pagedir, frame->vme->vaddr);
        frame_delete(frame);
        palloc_free_page(frame->page_addr);
        free(frame);
    }

    lock_release(&frame_lock);
}
  
```

- **evict_frame**

evict할 victim frame을 clock algorithm을 통해 찾아서 swap out 시킴으로써 memory에 free 공간을 만드는 것이다. 이 때, victim frame이 방출 될 때 vm_entry의 type을 고려해야 한다.

- VM_BIN : Dirty bit가 1이면 swap partition에 기록 후 frame을 free하고, demand paging을 위해 type을 VM_ANON으로 변경한다.
- VM_FILE : Dirty bit가 1이면 file에 변경 내용을 저장 후 frame을 free하고 Dirty bit가 0이면 바로 frame을 free한다.
- VM_ANON : 항상 swap partition에 기록한다.

```
void evict_frame()
{
    lock_acquire(&frame_lock);

    // 1. victim frame 찾기
    struct frame *frame = find_victim();
    // 2. 해당 frame의 dirty bit 확인
    bool frame = pagedir_is_dirty(frame->thread->pagedir, frame->vme->vaddr);

    // 3. frame을 evict할 때 vm_entry의 type을 고려
    vme_type = frame->vme->type
    if (vme_type == VM_FILE) // VM_FILE일 경우
    {
        // dirty일 경우 write back
    }
    else if (vme_type == VM_BIN) // VM_BIN의 경우
    {
        // dirty bit이 1인 경우 swap partition에 기록
        // demand paging을 위해 type을 VM_ANON으로 변경
    }
    else if (vme_type == VM_ANON)
    {
        // swap partition에 기록
    }

    // 4. free frame

    lock_release(&frame_lock);
}
```

- **find_victim**

```
static struct list_elem* find_victim()
{
    struct list_elem *frame_clock = NULL;
    struct list_elem *e;
    struct frame *frame;
```

```

while (true)
{
    // clock algorithm에 따라 frame clock 이동
    // 초기화 or clock이 맨 끝을 가리키는 경우
    if (!frame_clock || frame_clock == list_end(&frame_table))
    {
        if (!list_empty(&frame_table))
            frame_clock = list_begin(&frame_table);
        e = list_begin(&frame_table)
        else // frame table이 비어있는 경우
            return NULL;
    }
    else // next로 이동
    {
        frame_clock = list_next(frame_clock);
        if (frame_clock == list_end(&frame_table))
            continue;
        e = frame_clock;
    }

    frame = list_entry(e, struct frame, elem);

    // access bit 확인 -> 0이면 바로 Return
    if (!pagedir_is_accessed(frame->thread->pagedir, frame->vme->vaddr))
    {
        return e;
    }
    else
    {
        // access bit 1이면 0으로 바꾸고 그 다음으로 clock이동
        pagedir_set_accessed(frame->thread->pagedir, frame->vme->vaddr, false);
    }
}
}

```

7. On process termination

1) Basics

- **the definition or concept**

Process Termination은 process가 실행을 완료하고 시스템에서 해당 process의 모든 자원을 해제하는 과정을 의미한다. process termination은 일반적으로 프로그램이 정상적으로 완료되었거나 오류로 인해 비정상적으로 종료될 때 발생한다.

Process가 terminate할 때 수행되는 과정은 다음과 같다.

Exit Request : process는 실행이 완료되거나 특정 조건(예: 오류)에 도달했을 때 종료를 요청한다. 이는 `exit` 시스템 콜을 통해 발생할 수 있다.

Resource Deallocation : process 종료 과정에서 해당 process가 사용했던 모든 자원(메모리, 열린 파일, 네트워크 연결 등)이 시스템에 반환된다. 이를 통해 다른 process 가 이 자원들을 재사용할 수 있다.

Process State Update : process의 상태를 update한다.

Interaction with Parent Process: 자식 process가 종료되면 부모 process에게 통지된다. 부모 process는 `wait` 시스템 콜을 통해 자식 process의 종료 상태를 얻을 수 있다.

Cleanup of Page Directory: process에 할당된 가상 메모리 페이지 딕토리가 정리된다.

Exit Code: process는 종료 시 Exit Code를 반환한다. 이 코드는 process가 성공적으로 완료되었는지, 오류로 종료되었는지를 나타내는 데 사용된다.

- **Implementations in original pintos**

process가 terminate될 때 호출되는 `process_exit` 함수는 현재 다음과 같이 구현되어 있다. (sample 코드 분석)

```
/* Free the current process's resources. */
void
process_exit (void)
{
    // 1
    struct thread *cur = thread_current ();
    uint32_t *pd;
    struct signal *sig;
    struct fd_elem *f_e;
    struct list_elem *e;

    // 2
    if(cur->parent) list_remove(&cur->p_elem);

    // 3
    for (e = list_begin (&cur->signal_list); e != list_end (&cur->signal_list); )
    {
        sig = list_entry (e, struct signal, elem);
        e = list_remove(e);
        free(sig);
    }
    // 4
    if (cur->current_file != NULL) {
        file_allow_write(cur->current_file);
        file_close(cur->current_file);
    }
}
```

```

    }
    // 5
    for (e = list_begin (&cur->fd_table); e != list_end (&cur->fd_table); )
    {
        f_e = list_entry (e, struct fd_elem, elem);
        e = list_remove(e);
        file_close(f_e->file_ptr);
        free(f_e);
    }
    // 6
    /* Destroy the current process's page directory and switch back
       to the kernel-only page directory. */
    pd = cur->pagedir;
    if (pd != NULL)
    {
        /* Correct ordering here is crucial. We must set
           cur->pagedir to NULL before switching page directories,
           so that a timer interrupt can't switch back to the
           process page directory. We must activate the base page
           directory before destroying the process's page
           directory, or our active page directory will be one
           that's been freed (and cleared). */
        cur->pagedir = NULL;
        pagedir_activate (NULL);
        pagedir_destroy (pd);
    }
}

```

1. 먼저 현재 thread를 가져온다.
2. 현재 thread의 parent가 있으면 parent thread의 자식 list에서 현재 thread를 제거 한다.
3. signal_list를 돌면서 모든 element를 free 시킨다.
4. 현재 thread에서 연 file을 가지고 있으면 그 file의 write 권한을 allow하고 file을 close한다.
5. fd_table을 돌면서 모든 file descriptor을 해제한다.
6. 현재 process의 page directory를 가져와 NULL이 아닌 경우 timer interrupt가 process page directory로 다시 전환되는 것을 방지하고 kernel 전용 page directory로 전환하며 process의 page directory를 destroy한다.

2) Limitations and Necessity

- the problem of original pintos

현재 process_exit은 vm project를 구현하기 전에 맞게 구현되어 있다. 즉, process에 할당된 모든 자원들을 할당 해제하고 종료하도록 구현되어 있지만 우리가 vm project를 진행하면서 추가한 필드들을 해제하는 코드는 따로 해제하고 있지 않다.

- **the benefit of implementation**

따라서 우리는 우리가 추가한 필드들에 대해 할당 해제하는 코드를 추가함으로써 메모리가 누수되는 것을 방지해야 한다. 이렇게 함으로써 pintos의 메모리 효율성을 높일 수 있으며 시스템의 안정성 또한 보장할 수 있다.

3) Blueprint: how to implement it

detailed data structures and pseudo codes

Process가 terminate되면 supplemental page table, frame table, swap table 등 우리가 이번 project에서 추가한 data structure들과 resource들을 모두 삭제해주어야 한다.

1~6을 design하면서 우리가 thread struct에 추가한 필드는 다음과 같다.

```
struct thread
{
    struct list mmap_list;
    int mmap_next;
    struct hash vm;
}
```

- **process_exit(void)**

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

process가 종료될 때 호출하는 process_exit에서 munmap을 호출하여 모든 mapping들을 unmap할 수 있도록 한다. 즉, mmap_list에서 mapping에 해당하는 mapid를 갖는 모든 vm_entry를 해제하도록 한다. 또한, thread에 할당되어 있던 vm_entry도 모두 제거하는 코드를 추가한다.

최종적으로, 이번 추가 implementation을 위해 새로 생성한 vm directory의 파일을 사용하기 위해 pintos/Makefile.build 파일에서 코드를 추가해주어야 한다.