Assn3 Design Report

Pintos Assn3 Design Report

Team 37
20200229 김경민, 20200423 성치호

1. Frame Table

Basics

Pintos는 Virtual Memory를 효율적으로 관리/구현하기 위해 Page와 이를 관리하기 위한 Page Directory, Page Table 등을 구현해두었다.

Page

pintos에서 주로 메모리를 관리할 때 사용하는 단위이다. Virtual Memory 위의 연속된 메모리 공간 **4KB**를 의미하고, 모든 Virtual Memory는 Page 단위로 나누어져 할당 받거나 해제된다. Virtual Memory 상 4KB를 페이지 하나로 생각하기 때문에 Virtual Address는 다음과 같이 해석된다.

```
 31                12 11         0
+--------------------+------------+
|    Page Number     |   Offset   |
+--------------------+------------+
                  Virtual Address
```

virtual address의 MSB부터 20비트는 Page Number로, 나머지 12비트는 offset으로 취급한다. 왜냐하면 Page는 Page-Aligned되어 있으며 Page가 4KB = 2^{12} Byte이기 때문이기에 주소의 하위 12비트가 표현하는 주소들은 모두 같은 페이지 내에 있기 때문이다. 또한 page number는 총 20비트로 1024 * 1024개의 page를 표현할 수 있으며 이는 각각 page directory, page table로 구성된다. 이에 대해 자세히 후술하겠다.

Page Directory, Page Table, Page Table Entry

Pintos에서 Virtual Memory는 Page Directory, Page Table, Page Table Entry를 통해 구현된다.

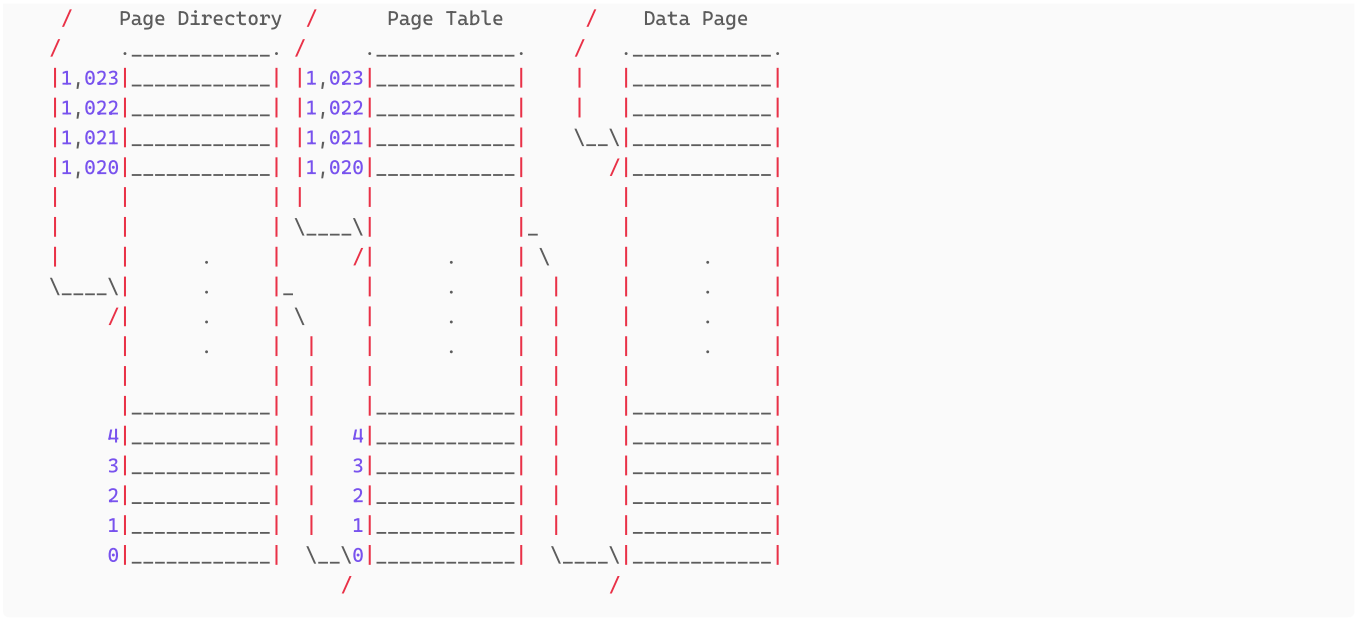
```
struct thread
  {
    ...
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
        ...
  };
```

모든 프로세스(스레드)는 각자의 Page Directory를 가지고 있으며 독립적으로 관리하게 된다. 위 `pagedir`은 Page Directory로 사용되게 할당 받은 페이지의 시작 주소(kernel virtual Address임)로, Page Directory의 시작 위치이다. 추후 이 pagedir은 `pagedir_activate` in `userprog/pagedir.c`의 `asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");`를 통해 활성화되어 virtual address -> physical address의 변환 및 매핑을 설정하게 된다.

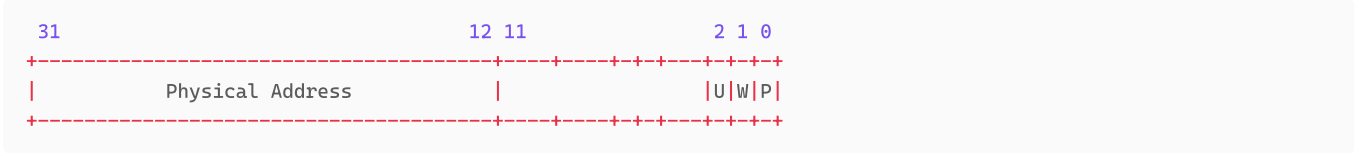
Pintos의 Virtual Memory는 아래 구조처럼 구성되며 32bit Virtual Address는 다음 구조를 가지고 있다.

Virtual Address는 Page Directory Index 10비트, Page Table Index 10비트, Page offset이 12비트로 이루어진다. 위에서 말한 것과 같이 Page Number가 총 20비트로 1024 * 1024 개의 page를 표현할 수 있는데 이를 Page Directory(1024 index), Page Table(1024)로 2단계로 구분하여 이와 같은 구조를 띄게 된 것이다.

```
 31                 22 21                  12 11                   0
+---------------------+----------------------+----------------------+
| Page Directory Index |   Page Table Index   |    Page Offset       |
+---------------------+----------------------+----------------------+
             |                    |                     |
      _______/             _______/              _____/
     /                    /                     /
```



pintos는 각 스레드마다 각기 다른 page directory를 가지고 있고 독립적으로 관리한다. 이런 page directory는 1024개의 page directory entry를 가지며 각 entry는 32bit로 이루어진다.



엔트리의 앞선 31~12bit는 각각 다른 page table의 시작 physical address의 31~12bit 부분을 담고 있다. 이 때 뒤의 12bit를 포함하지 않아도 되는 이유는 page table의 시작 위치가 4KB 단위로 allign되어 하위 12비트는 모두 0일 것이 보장되기 때문이다. 이는 추후 나올 Page Table Entry에서도 동일하다. 하위 11~0 bit에는 page directory entry에 대한 flag들이 포함된다.

```
static inline uint32_t pde_create (uint32_t *pt) {
  ASSERT (pg_ofs (pt) == 0);
  return vtop (pt) | PTE_U | PTE_P | PTE_W;
}
```

Flag	없을 때	있을 때
`PTE_U`	kernel만 접근 가능	kernel, user 모두 접근 가능
`PTE_P`	PDE 존재X, 다른 flag 모두 의미 없어짐.	PDE 존재O, 유효
`PTE_W`	read-only	read/write 둘 다 가능

`pde_create` 는 주어진 page table을 가르키는 page directory entry를 생성하는 함수로 base page directory를 초기화하는 `paging_init` 에서 kernel virtual memory에 대한 page를 초기화할 때 또는 `lookup_page` 에서 virtual address에 대한 page table entry가 없을 때, 생성하는 도중 사용한다.

`pagedir_create` in `userprog/pagedir.c`

```
uint32_t *
pagedir_create (void)
{
  uint32_t *pd = palloc_get_page (0);
  if (pd != NULL)
    memcpy (pd, init_page_dir, PGSIZE);
  return pd;
}
```

page directory를 생성하는 함수로 page directory를 위한 page를 할당 받고 여기에 `init_page_dir` 을 복사해 넣는다. `init_page_dir` 은 base page directory로 kernel virtual memory - physical memory mapping을 포함하고 있으며 모든 page directory가 생성시 해당 pd를 복제함으로써 해당 매핑을 모든 page directory가 가질 수 있게 한다.
해당 함수는 userprogram load시 해당 프로세스의 스레드의 독립적인 page directory를 구축할 때 사용한다.

`pagedir_destroy`

```
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);
}
```

user virtual memory에 대응되는 page directory entry가 가르키는 page table과 해당 page table의 entry와 대응되는 할당된 page(`palloc_get_page`에 의해서)를 할당 해제해준다. 마지막으로 page directory에 할당된 page도 할당 해제한다.
이로써 해당 프로세스의 page directory 자원이 할당 해제되고 프로세스 자원 해제`process_exit`에서 사용된다.

각 Page Table Entry가 가르키는 Page Table은 1024개의 Page Table Entry로 구성되어 있다. 각 Page Table Entry는 아래 같은 구조를 가진다.

```
 31                                  12 11 9      6 5     2 1 0
+-------------------------------------+----+----+-+-+---+-+-+-+
|           Physical Address            | AVL|    |D|A|   |U|W|P|
+-------------------------------------+----+----+-+-+---+-+-+-+
```

상위 31~12 비트는 해당 Page와 매핑된 Frame(Physical Memory 단위)의 상위 20비트이다. Frame은 Page와 유사하기 4KB이며 4kb로 aligned되어 있어 Frame의 시작 주소는 하위 12개 비트가 0임이 보장된다. PTE의 하위 12비트에는 Page Table Entry에 대한 Flag가 포함되어 있다.

```
static inline uint32_t pte_create_kernel (void *page, bool writable) {
  ASSERT (pg_ofs (page) == 0);
  return vtop (page) | PTE_P | (writable ? PTE_W : 0);
}
```

Flag	없을 때	있을 때
`PTE_P`	PTE 존재X, 다른 flag 모두 의미 없어짐.	PTE 존재O, 유효
`PTE_W`	read-only	read/write 둘 다 가능

위의 구조와 달리 실제 kernel virtual page에 대한 page table entry 생성은 `PTE_P`, `PTE_W` flag만을 포함한다.

```
static inline uint32_t pte_create_user (void *page, bool writable) {
  return pte_create_kernel (page, writable) | PTE_U;
}
```

Flag	없을 때	있을 때
`PTE_U`	kernel virtual memory	user virtual memory

user virtual page에 대한 page table entry 생성은 `PTE_P`, `PTE_U`, `PTE_W` flag를 포함하게 된다.

Frame

pintos에서 **Physical Memory**를 관리할 때 사용하는 단위이다. 연속된 공간의 Physical Memory로, page와 동일하게 **4KB**이다. pintos에서 page는 관리하기 위해 page directory, page table 등 을 구현하고, 함수들의 반환 값으로 사용하는 등 빈번하게 사용되는 반면, frame은 `pagedir_set_page` 와 `install_page` 등에서 간접적으로 언급되는 것을 제외하고는 직접적으로 언급되지 않는다. 그대신 `kernel page와 user page의 매핑` 이라는 용어를 통해 사용된다.

```
 31                12 11         0
+--------------------+------------+
|    Frame Number    |   Offset   |
+--------------------+------------+
                        Physical Address
```

Physical Address는 앞의 20비트(31~12)는 Frame Number를, 나머지 뒤의 12비트(11 ~ 0)은 Frame 내 offset을 의미한다. 이는 virtual address & page와 유사하게 physical memory 상에서 frame이 frame-aligned 되어있으며 frame이 4KB이기 때문이다.

80x86 프로세서는 단순히 Physical Address를 통해 메모리에 접근하는 방법을 제공하지 않는다. Pintos에서는 이를 kernel virtual memory와 physical memory를 direct mapping하여 간접적으로 방법을 제공한다. 즉 kernel virtual memory의 첫 page는 physical memory의 첫 frame과 매칭된다. kernel virtual memory는 Virtual Memory 상에서 `PHYS_BASE(0xc0000000)` 부터 시작하기에 Kernel Virtual Memory `0xc0000000` 은 physcial Address `0x00000000` 에 대응된다고 할 수 있다.

```
// In threads/vaddr.h
static inline void *
ptov (uintptr_t paddr)
{
  ASSERT ((void *) paddr < PHYS_BASE);

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

static inline uintptr_t
vtop (const void *vaddr)
{
  ASSERT (is_kernel_vaddr (vaddr));

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

위의 원리로 Pintos에서는 `ptov` 에서는 physical address에 `PHYS_BASE` 를 더해서 virtual address를 반환하고, `vtop` 에서는 virtual address에 `PHYS_BASE` 를 뺌으로써 physcial address를 구해 반환한다. 당연히 이 때 virtual address는 user virtual address가 아닌 (physical memory와 매핑되어 있는) kernel virtual address여야만 한다.

위 같은 작동이 가능하도록 pintos kernel 초기화(`main` in `init.c`)에서 `paging_init` 함수를 호출하여 kernel virtual memory와 physical memory간 Mapping을 생성한다.

```
// threads/init.c
static void
paging_init (void)
{
  uint32_t *pd, *pt;
  size_t page;
  extern char _start, _end_kernel_text;

  pd = init_page_dir = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  pt = NULL;
  for (page = 0; page < init_ram_pages; page++)
    {
      uintptr_t paddr = page * PGSIZE;
      char *vaddr = ptov (paddr);
      size_t pde_idx = pd_no (vaddr);
      size_t pte_idx = pt_no (vaddr);
      bool in_kernel_text = &_start <= vaddr && vaddr < &_end_kernel_text;

      if (pd[pde_idx] == 0)
        {
          pt = palloc_get_page (PAL_ASSERT | PAL_ZERO);
```

```
          pd[pde_idx] = pde_create (pt);
        }

      pt[pte_idx] = pte_create_kernel (vaddr, !in_kernel_text);
    }

  asm volatile ("movl %0, %%cr3" : : "r" (vtop (init_page_dir)));
}
```

base page directory `init_page_dir`에 page를 할당한다. 물리주소 0부터 page의 크기(4kB)만큼 주소를 늘려가며 해당 Physical Address에 대응되는 kernel virtual address(0xc0000000 이상)에 대한 Page Table Entry를 `pte_create_kernel`을 통해 생성한다. 중간에 Page Directory, Page Table에 대한 공간 할당이 필요하다면 `palloc_get_page`를 통해 공간을 할당한다.
위 과정을 `init_ram_pages`(Physical Memory / 4kB, 가능한 페이지/프레임 개수)만큼 반복한다.
마지막으로 cr3 레지스터가 `init_page_dir`의 물리 주소를 가리키게 한다.
이로써 Kernel Virtual Memory(존재하는 Physical Memory만큼)와 Physical Memory는 대응되게 된다. **즉 Kernel Virtual Address의 page를 physical frame처럼 취급할 수 있게 된다.** 모든 process의 page directory는 virtual-physical mapping이 포함된 `init_page_dir`을 복사하여 생성되므로 동일한 virtual-physical mapping을 가지고 있게 된다.

위의 내용들은 Pintos에서 Kernel Virtual Memory를 통해 간접적으로 원하는 Physical Address의 Physical Memory의 Frame에 접근할 수 있도록 한다.
아래 구현은 Pintos에서 User Virtual Address/Page에 Frame을 연결하는 방법이다.

```
static bool
install_page (void *upage, void *kpage, bool writable)
{
  struct thread *t = thread_current ();

  /* Verify that there's not already a page at that virtual
     address, then map our page there. */
  return (pagedir_get_page (t->pagedir, upage) == NULL
          && pagedir_set_page (t->pagedir, upage, kpage, writable));
}
```

해당 함수는 `upage`에 `kpage`를 install하는, 다른 말로는 mapping, 할당하는 함수이다.
`kpage`는 이미 `palloc_get_page`를 통해 할당받은 Physical Frame과 매핑된 Kernel Virtual Page이다.
`upage`는 physical memory를 할당 받고 싶은 user virtual page이다. `install_page`에서 `pagedir_get_page`의 값이 `NULL`인 것을 통해 `upge`에 대응되는 page table entry가 없음을, 즉 할당된 physical frame이 없음을 확인한다. 이후 `pagedir_set_page`를 호출한다.

```
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) >> PTSHIFT < 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;
}
```

`lookup_page`를 통해 `upage`에 대응되는 page table entry 주소를 얻는다. 또한 중간에 없는 page table, page directory entry 등을 생성한다.
`pte_create_user`를 통해 `kpage`의 page table entry에 `PTE_U`를 추가하여 `upage`의 page table entry로 집어 넣는다. 즉, `palloc_get_page`를 통해 얻은 kernel virtual page(physical frame과 대응되는)와 물리 공간을 할당 받기 원하는 user virtual page간

mapping을 생성하여 간접적으로 user page에 공간을 할당한 것이다.
이것이 user page가 간접적으로 frame을 할당 받는 방법이다.

```
static inline uint32_t pte_create_user (void *page, bool writable) {
  return pte_create_kernel (page, writable) | PTE_U;
}
```

Page Allocator

`palloc_get_page`, `palloc_free_page`로 "Page" Allocator처럼 작동하지만 실상은 frame allocator에 가깝다.

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

Memory pool을 나타내는 struct로 `bitmap`을 통해 pool 내에서 free한 page들을 저장하고 `base`는 해당 memory pool의 시작 주소를 나타낸다. `lock`은 `pool`로부터 페이지를 할당 받을 때 사용한다.

```
static struct pool kernel_pool, user_pool;
```

메모리 풀은 `user_pool`, `kernel_pool`로 2 개 존재한다.

```
void
palloc_init (size_t user_page_limit)
{
  /* Free memory starts at 1 MB and runs to the end of RAM. */
  uint8_t *free_start = ptov (1024 * 1024);
  uint8_t *free_end = ptov (init_ram_pages * PGSIZE);
  size_t free_pages = (free_end - free_start) / PGSIZE;
  size_t user_pages = free_pages / 2;
  size_t kernel_pages;
  if (user_pages > user_page_limit)
    user_pages = user_page_limit;
  kernel_pages = free_pages - user_pages;

  /* Give half of memory to kernel, half to user. */
  init_pool (&kernel_pool, free_start, kernel_pages, "kernel pool");
  init_pool (&user_pool, free_start + kernel_pages * PGSIZE,
             user_pages, "user pool");
}
```

메모리 풀을 초기화하는 함수이다. kernel pool과 user pool은 각각 메모리 전체의 반씩을 가지게 된다.

- 메모리 풀의 범위는 `ptov`를 통해 virtual address로 나타나지만 두 pool에 할당될 총 page 개수, 가상 메모리 범위는 physical memory의 크기이다.
- `init_ram_pages`는 physical memory의 크기를 page 크기로 나눈 것이다.
- kernel pool은 physical memory에 대응되는 kernel virtual memory 앞의 절반을, user pool은 뒤의 절반을 차지하게 된다.
  pintos kernel 초기화 과정인 `main() in init.c`에서 호출하여 page allocator를 초기화한다.

```
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;
}
```

`PAL_USER` flag에 따라 올바른 pool을 선택하여 `bitmap_scan_and_flip`을 통해 free인 page index를 얻고 bitmap 상에 free가 아닌 상태로 변경한다. 동시에 bitmap을 조작하는 일이 없도록 lock을 사용한다.
`pool->base`를 더하고 page index에 `PGSIZE` 만큼 더하여 해당 page의 virtual address를 얻는다. 이후 flag에 따라 page를 0으로 초기화하거나 page를 얻지 못했을 때 panic한다.
`palloc_get_page`는 내부적으로 `page_cnt=1`로 설정하여 해당 함수를 호출한다.

```
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);
}
```

`palloc_get_multiple`의 반대작용으로 `page`의 `pool`을 판별한 뒤 해당 pool에서 `page`에 해당하는 bit를 false(free)로 설정한다.
`palloc_free_page`는 내부적으로 `page_cnt=1`로 설정하여 해당 함수를 호출한다.

User, Kernel Memory pool은 핀토스 내 코드에서 `palloc_get_page`를 통해 페이지를 할당할 때 사용한다. 이렇게 얻는 page들은 결국 각각 page 크기만큼의 서로 다른 physical memory 공간에 대응되며, pool의 bitmap을 통해 관리되므로 페이지가 free되지 않는 이상 같은 page, 같은 kernel virtual page를 절대로 할당/리턴 받을 수 없다. 그렇기에 `palloc`은 이름으로만 page allocator이고 실제로는 frame allocator와 유사한 의미로 작동하게 된다. 그렇기에 우리는 아래 디자인에서 `palloc`을 다른 allocator로 대체하여 frame 할당/해제를 제어하고 frame table을 제어하는 것이다.

Limitations and Necessity

현재 Pintos에는 `kernel virtual page - physical memory 매핑`을 통해 frame 접근방식을 제공하고 `user virtual page`를 `kernel virtual page` 매핑(user virtual page table entry는 kernel virtual page table entry의 복사본 + user flag)하여 user virtual page가 간접적으로 frame을 할당 받을 수 있도록 하였다.

이것이 frame과 관련된 구현의 전부로 frame(kernel virtual page)과 user virtual page 간의 매핑을 별도로 관리하지 않아 frame이 부족할 때 evict할 (user page - frame의 매핑을 끊을) page를 정하는데 어려움을 겪는다. 이를 개선하기 위해 어떤 Frame(kernel virtual page)이 어떤 Page(user virtual page)와 매핑되어 있는지를 관리하는 Frame Table이 필요하다.

Blueprint

아래 코드들은 c와 유사한 문법을 작성한 대략적인 구조, 알고리즘을 나타낸 pseudo 코드이다.
우리는 Frame Table을 `list` 자료구조를 이용해 설계하기로 결정하였다.

- 이와 같이 결정한 이유 중 하나는 `pintos`에서 비슷한 예시로 `inode`를 이미 `list`를 이용해 관리하고 있기 때문이다. 또한 `list`를 이용한 구현이 간단하며 실제로 사용하고 있는 frame만 저장하기에 효율적이고 이후 clock 알고리즘을 evict policy로 사용할 시 구현이 상대적으로 직관적인 이점이 있다.

Frame Table

```
static struct list frame_table;

struct frame_table_entry
{
        tid_t tid;
        void *upage;
        void *kpage;
        bool use_flag;
        struct list_elem elem;
}
```

`frame_table`은 Frame Table로 전역에 하나만 존재한다.

- kpage는 frame으로 전역에서 각각 유일하며(kernel virtual page-physical memory mapping은 모든 page directory에서 공유됨.), user page는 어떤 스레드(`tid`)의 user page인지로 구별할 수 있다.

멤버	자료형	설명
`tid`	`tid_t`	해당 frame을 점유하고 있는 thread의 id, `upage`가 어떤 스레드의 user virtual page인지.
`upage`	`void *`	`kpage`와 매핑될 user virtual page
`kpage`	`void *`	physical frame과 매칭되는 kernel virtual page
`use_flag`	`bool`	clock 알고리즘에서 사용할 use flag
`elem`	`list_elem`	`frame_table`를 `list`로 구성하기 위한 `list_elem`

```
void
frame_table_init()
{
        list_init(&frame_table);
}
```

frame table을 초기화하는 함수로 kernel 초기화 과정 중 `paging_init` 직후 호출한다.

`palloc_get_page`를 비롯한 page allocator(실제로는 physical memory와 매핑된 kernel virtual page만을 반환하므로 frame allocator 역할을 수행) `palloc`을 대체하기 위한 `falloc` (Frame Allocator)를 추가한다. `falloc`은 기존 `palloc` 역할에 더해 `frame_table`을 함께 변경시킨다.

```
enum falloc_flags
{
        FAL_ASSERT = 001,
        FAL_ZERO = 002,
        FAL_USER = 004,
}
```

`vmalloc_get_page`를 위한 flag enum이다. 기존의 `enum palloc_flags`와 동일한 역할과 구성이다.

Flag	없을 때	있을 때
`FAL_ASSERT`	allocation 실패시 null 반환	allocation 실패시 panic

Flag	없을 때	있을 때
FAL_ZERO		page 0으로 초기화. PAL_ZERO 에 대응.
FAL_USER	page를 kernel pool에서 가져옴	page를 user pool에서 가져옴. PAL_USER 에 대응

falloc_get_page_w_page 는 모든 상황에서 FAL_USER flag가 함께하길 기대한다. 이번 프로젝트의 대부분 작업이 user virtual memory를 다루는 일이기 때문이다.

```
void *
falloc_get_frame_w_upage (enum falloc_flags, void* upage)
{
        void *kpage = palloc_get_page(falloc_flags except FAL_ASSERT);
        if(kpage == null)
        {
                //TODO: evict policy
                evict_policy();
                *kpage = palloc_get_page(falloc_flags);
                if(kpage == null)
                        return null;
        }
        struct frame_table_entry *fte = malloc(sizeof *fte);
        if(fte == null)
        {
                palloc_free_page(kpage);
                if(falloc_flags & FAL_ASSERT)
                        PANIC
                return null;
        }
                tid_t tid;
        void *upage;
        void *kpage;
        bool use_flag;
        fte->tid = thread_current()->tid;
        fte->upage = upage;
        fte->kpage = kpage;
        fte->use_flag = false;
        list_push_back(&frame_table, &fte->elem);
}
```

주어진 upage user virtual page에 frame (kernel virtual page)를 매핑하고 frame table에 해당 매핑을 등록하는 함수의 pseudo코드이다. 만약 palloc_get_page 가 실패한다면, 즉 할당 가능한 남은 frame이 없다면 후술할 evict policy에 근거하여 특정한 user virtual page의 frame 매핑을 제거하여 frame을 확보한다. 이후 다시 palloc_get_page 를 통해 frame 할당을 시도한다. 올바르게 frame을 얻었다면 해당 frame에 대한 frame_table_entry 값을 초기화해준 뒤 frame_table 에 추가한다. frame_table_entry 를 위한 공간을 할당하기 위해서는 malloc 을 이용해 frame_table_entry 만큼의 공간만큼만 할당할 것이다.

```
*struct frame_table_entry
find_frame_table_entry_from_frame(void *frame)
{
        struct list_elem *e
        for(e = list_begin(&frame_table); e != list_end(&frame_table); e = list_next(e))
        {
                struct frame_table_entry *fte = list_entry(e,struct frame_table_entry, elem);
                if(fte->kpage == frame)
                        return fte;
        }
        return null;
}
```

frame 을 입력할 시 해당 frame (kernel virtual page)와 매핑된 frame_table_entry 를 반환하는 함수이다. frame_table 을 순회하며 입력된 frmae 과 엔트리의 kpage 가 동일하면 해당 엔트리 주소를 반환한다. 만약 frame table에없는 frame이라면 null을 반환한다. falloc_free_frame 에서 frame에 대응되는 entry를 찾을 때 사용한다.

```
struct frame_table_entry *
find_frame_table_entry_from_upage(void *upage)
{
        struct list_elem *e
```

```
        for(e = list_begin(&frame_table); e != list_end(&frame_table); e = list_next(e))
        {
                struct frame_table_entry *fte = list_entry(e,struct frame_table_entry, elem);
                if(fte->upage == upage)
                        return fte;
        }
        return null;
}
```

`upage`을 입력할 시 해당 `upage`(user virtual page)와 매핑된 `frame_table_entry`를 반환하는 함수이다. `frame_table`을 순회하며 입력된 `upage`과 엔트리의 `upage`가 동일하면 해당 엔트리 주소를 반환한다. 만약 `upage` user page와 매핑된 `frame`이 존재하지 않다면 null을 반환한다. 즉 해당 user virtual page는 `frame`을 할당받지 못한 상태이다.(swap out 또는 lazy loading)

```
void
falloc_free_frame (void *frame)
{
        struct frame_table_entry *fte = find_frame_table_entry_from_frame(frame);
        if(fte == null)
        {
                // ERROR!
                return;
        }
        list_remove(&fte->elem);
        palloc_free_page(frame);
        free(&fte);
}
```

입력 받은 `frame`에 대응되는 `frame_table_entry`를 찾는다. (사용되고 있는 올바른 frame인가?) 이후 entry를 frame table에서 삭제한 후, 해당 frame을 `palloc_free_page`로 할당 해제하고 frame table entry로 할당 해제한다.

이렇게 완성된 falloc interface는 기존에 user virtual memory에 대해서 사용되던 `palloc_get_page`, `palloc_free_page` 등을 대체하여 사용한다.

- 프로젝트 2의 `load_segment`, `setup_stack` 등의 `palloc`을 대체한다.

2. Lazy Loading

Basics

```
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
  ASSERT (pg_ofs (upage) == 0);
  ASSERT (ofs % PGSIZE == 0);

  file_seek (file, ofs);
  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* Calculate how to fill this page.
         We will read PAGE_READ_BYTES bytes from FILE
         and zero the final PAGE_ZERO_BYTES bytes. */
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      /* Get a page of memory. */
      uint8_t *kpage = palloc_get_page (PAL_USER);
      if (kpage == NULL)
        return false;

      /* Load this page. */
      if (file_read (file, kpage, page_read_bytes) != (int) page_read_bytes)
        {
          palloc_free_page (kpage);
          return false;
        }
```

```
      memset (kpage + page_read_bytes, 0, page_zero_bytes);

      /* Add the page to the process's address space. */
      if (!install_page (upage, kpage, writable))
        {
          palloc_free_page (kpage);
          return false;
        }

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

`load_segment` 는 `file` 의 `ofs` 에서 시작하는 세그먼트를 `upage` user virtual page에 로드하는 함수이다. 이 때 `uint8_t *kpage = palloc_get_page (PAL_USER);` 를 통해 세그먼트를 저장할 kernel virtual page(user pool에서 얻은 frame)를 할당 받고 `file_read` 를 통해 `kpage` 에 값을 읽어 들인다. `install_page (upage, kpage, writable)` 을 통해 `upage` 와 segment를 담은 `kpage` 와 매핑을 생성한다.

- `kpage` 는 kernel virtual page로 physical frame과 대응되어 간접적으로 `frame` 을 의미한다. user virtual page `upage` 와 `kpage` 의 매핑을 생성함으로써 `upage` 가 해당 프레임을 할당 받은 것과 같게 된다.

```
static bool
install_page (void *upage, void *kpage, bool writable)
{
  struct thread *t = thread_current ();

  /* Verify that there's not already a page at that virtual
     address, then map our page there. */
  return (pagedir_get_page (t->pagedir, upage) == NULL
          && pagedir_set_page (t->pagedir, upage, kpage, writable));
}
```

- `install_page` 는 `upage` 의 page entry를 `kpage` 의 page entry(virtual address - PHYS_BASE 물리 주소와 매핑된)의 복사본에 user flag를 더한 것으로 설정함으로써 매핑을 설정한다.
  이를 upage와 읽어들이는 file의 위치를 page size만큼 증가시키며 segment를 끝까지 user virtual page에 저장할 때까지 반복한다.
  현재 `load_segment` 는 userprogram/file을 로드할 때, 이처럼 user virtual page에 반드시 frame을 할당받으며 전체를 저장하게 된다.

```
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));
  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;

  /* Kernel caused page fault by accessing user memory */
  if(!user && check_ptr_in_user_space(fault_addr))
  {
   f->eip = (void *)f->eax;
   f->eax = -1;
   return;
```

```
  }
  /* User caused page fault */
  else sys_exit(-1);

  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);
}
```

다음은 pintos2 구현 완료 후의 page fault를 처리하는 핸들러 함수인 `page_fault` 코드이다.
page fault exception 발생시 해당 함수가 트리거된다.

- 현재는 page fault 발생시 kernel process에서 발생된 것이라면 interrupt frame의 eax를 -1로 설정하고 return하고 user process에서 발생된 것이라면 해당 user process를 exit code -1와 함께 종료한다.
- 이처럼 page fault는 현재 항상 오류로 취급되어 process를 종료시킨다.

Limitations and Necessity

user process가 실행할 user program(file)을 로드하는 `load` 에서 segment를 로드하는 함수인 `load_segment` 는 항상 kernel virtual page(frame)와 매핑된 user virtual page들에 세그먼트의 전체를 저장하게 된다. 즉 user program 전체가 physical memory에 올라오는 것이며 만약 현재 사용 가능한 physical memory 공간(남은 user pool)이 user program보다 작을 경우 해당 user program을 로드할 수 없고 실행할 수 없다. 또한 큰 프로그램일 경우 로드하는데 오랜 시간이 걸린다. 프로그램을 모두 로드할 때 까지 프로그램을 실행하지 않으므로 프로그램 실행까지 latency가 길어진다. 만약 매우 큰 userprogrm를 모드 로드한 뒤 유저 프로세스를 실행하였으나 실행 직후 해당 유저 프로세스가 오류나 코드 내 조건에 의해 종료된다면 유저 프로그램 전체를 로드한 것이 낭비가 될 것이다. 또한 유저 프로세스 실행시 로드된 유저 프로그램의 모든 부분을 사용하는 것이 아니라 그 중 일부만을 차근차근 사용하게 된다. 즉 현재 load 구조는 physical memory를 심각하게 낭비하고 있다.
이를 개선하기 위해 lazy loading을 도입해야 한다. lazy loading이란 정말 필요할 때가 되서야 physical memory에 데이터를 조금씩(한 page/frame 씩) load하는 것이다. 그 전 까지는 단순히 page만 할당하여(frame은 할당되지 않은 user virtual page) 해당 데이터가 로드된 것처럼 속임으로써 초기 load latency를 획기적으로 줄일 수 있다. 또한 실제 물리 메모리에, `frame` 을 할당하여 데이터를 저장하지 않으므로 `frame` 을 낭비하지 않고 다른 곳에서 효율적으로 사용할 수 있다.

Blueprint

아래는 모두 c언어 문법과 유사하게 작성된 pseudo code이다.
Lazy Loading을 구현하기 위해서는 먼저 3. Supplemental Page Table이 구현되어 있어야 한다. (3번 항목을 먼저 참고)

```
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
  ASSERT (pg_ofs (upage) == 0);
  ASSERT (ofs % PGSIZE == 0);

  file_seek (file, ofs);
+ int i = -1;
  while (read_bytes > 0 || zero_bytes > 0)
    {
+     i += 1;
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      /* Get a page of memory. */
      //uint8_t *kpage = palloc_get_page (PAL_USER);
      //if (kpage == NULL)
      //  return false;
+     spte_create(true, file, ofs + i * PGSIZE, writable, upage, null, page_read_bytes, page_zero_bytes,
FAL_USER)

      /* Load this page. */
      //if (file_read (file, kpage, page_read_bytes) != (int) page_read_bytes)
      //  {
      //    palloc_free_page (kpage);
```

```
      //    return false;
      //  }
      //memset (kpage + page_read_bytes, 0, page_zero_bytes);

      /* Add the page to the process's address space. */
      //if (!install_page (upage, kpage, writable))
      //  {
      //    palloc_free_page (kpage);
      //    return false;
      //  }

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

기존 `load_segment` 함수를 다음과 같이 변경한다.

- 기존 대비 삭제한 코드는 주석 처리, 추가한 코드는 `+` 표시를 해두었다.
- 기존에는 file_read를 통해 file 내 보고 있는 위치가 자동으로 변경되었으나 변경 후에는 lazy loading을 적용하여 당장 `file_read` 를 사용하지 않기에 `i` 를 이용해 while 문을 돌 때마다 읽어야 할 file 내 위치를 조작해준다.
- 또한 `palloc_get_page`, `palloc_free_page`, `install_page` 등의 frame/page 할당, user virtual page <-> kernel virtual page(frame) 매핑 추가 등의 코드를 모두 삭제한다.
  - 추후 page fault 발생시 실제로 load할 때 해당 코드를 사용하게 된다.
  - 대신 `spte_create` 를 통해 segment가 추후 실제로 로드될 user virtual page(frame 매핑이 없음.)를 할당한다.

```
bool
is_valid_page(void* uvaddr)
{
        void *upage = uvaddr & 0xffff000;
        s_page_table_entry* spte = find_s_page_table_entry_from_upage(upage);
        if(spte == null)
        {
                return false;
        }
        return spte->present;
}
```

입력한 user virtual address인 `uvaddr` 이 valid한 page에 포함되는 주소인지 판별하는 함수이다.

- 입력한 user virtual address인 `uvaddr` 와 `0xffff000` (상위 20비트만 값 있음) and 연산해 page aligned한 주소로 변경한다.(해당 주소가 속한 page)
- 이 때 valid하다는 것은 supplemental page table 현재 스레드의 `s_page_table` 에 엔트리가 존재하는가 이다. 만약 valid하지 않다면 false를 반환하고 존재한다면 `spte->present` 를 반환한다. `present` 는 해당 entry가 유효한지 여부이다.

```
bool
load_frame_mapped_page(void* uvaddr)
{
        void *_upage = uvaddr & 0xffff000;
        s_page_table_entry* spte = find_s_page_table_entry_from_upage(_upage);
        if(_page == null)
        {
                return false;
        }
        if(spte->in_swap)
        {
                // swap in
                // include spte->in_swap = false;
                return true/false;
        }
        if(spte->is_lazy && (spte->has_loaded == false))
        {
                spte->kpage = falloc_get_page_w_upage(spte->falloc_flags, spte->upage);
```

```
                if(spte->kpage == null)
                {
                        return false;
                }
                if(spte->file != null)
                {
                        file_lock_acquire();
                        file_seek (spte->file, spte->file_ofs);
                        if (file_read (spte->file, spte->kpage, spte->file_read_bytes) != (int) spte-
>file_read_bytes)
                {
                  falloc_free_page (spte->kpage);
                  file_lock_release();
                  return false;
                }
                    memset (spte->kpage + spte->file_read_bytes, 0, spte->file_zero_bytes);


                if (!install_page (spte->upage, spte->kpage, spte->writable))
                {
                  falloc_free_page (spte->kpage);
                  file_lock_release();
                  return false;
                }
                        file_lock_release();

                }
                spte->has_loaded = true;
        }

        return true;
}
```

특정 user virtual address에서 page fault 발생하였을 때, page fault가 발생한 주소의 page가 valid할 때 supplemental page table을 이용해 해당 page에 올바른 값의 frame을 할당하는 함수이다.

- 입력 받는 `uvaddr` user virtual address는 `is_valid_page` 를 통해 사전에 검사해 `valid` 함을 가정한다.
- `uvaddr` 이 포함된 `page` 의 `s_page_table_entry` `spte` 를 구한다.
- `spte->in_swap` , 즉 swap table에 위치하였다면 해당 page을 swap in 하여 frame을 할당해준다.
- `spte->is_lazy` 이나 `spte->has_loaded` 가 아닌 경우는 lazy loading page로 아직 load하지 않은 페이지이다.
  - `file` 정보가 없다면 단순히 엔트리에 저장된 `falloc_flags` 를 이용해 frame을 할당해준다.
  - `file` 정보가 있다면 기존에 `load_segment` 에 위치했던 segment를 메모리에 저장하는 로직을 수행한다.
  - 성공적으로 완료되면 `spte->has_loaded` 를 true로 변경하여 로딩이 완료되었음을 표시해준다.
- 이처럼 page fault의 원인을 해결하였다면 true를, frame 할당 등에 실패하여 아직도 해결할 수 없다면 false를 반환한다.

```
static void
page_fault (struct intr_frame *f)
{
  ...
  if(!user && check_ptr_in_user_space(fault_addr))
  {
   f->eip = (void *)f->eax;
   f->eax = -1;
   return;
  }
  /* User caused page fault */
+ if(is_valid_page(fault_addr) && load_frame_mapped_page(fault_addr))
+ {
+   pass
+ }
  else sys_exit(-1);
  ...
}
```

`page_fault` 핸들러 함수에 `+` 내용을 추가한다. 먼저 page fault의 원인이 되는 `fault_addr` 을 `is_valid_page` 를 통해 검사하고 valid하다면 `load_frame_mapped_page` 를 통해 swap in, lazy loading 등을 수행하여 page fault를 해결하고자 한다. 만약 frame 부족 등의 이

유로 실패시 기존 처리 대로 `sys_exit` 한다.

3. Supplemental Page Table

Basics

Page Table과 Page Table Entry에 대한 자세한 설명은 1. Frame Table에서 다루었기에 핵심이 되는 기존 Pintos의 Page Table Entry 구조에 대해서만 간단히 작성하도록 하겠다. (1. Frame Table 참고)

```
 31                                  12 11 9      6 5     2 1 0
+--------------------------------------+----+----+-+-+---+-+-+-+
|           Physical Address            | AVL|    |D|A|   |U|W|P|
+--------------------------------------+----+----+-+-+---+-+-+-+
```

```
static inline uint32_t pte_create_kernel (void *page, bool writable) {
  ASSERT (pg_ofs (page) == 0);
  return vtop (page) | PTE_P | (writable ? PTE_W : 0);
}

static inline uint32_t pte_create_user (void *page, bool writable) {
  return pte_create_kernel (page, writable) | PTE_U;
}
```

Flag	없을 때	있을 때
`PTE_P`	PTE 존재X, 다른 flag 모두 의미 없어짐.	PTE 존재O, 유효
`PTE_W`	read-only	read/write 둘 다 가능
`PTE_U`	kernel virtual page	user virtual page

user virtual page에 대한 page table entry는 매핑된 frame physical address, writable, page table entry의 유효 여부, user/kernel virtual page 여부, dirty bit, accessed bit만 주로 관리하게 된다.

```
static bool
install_page (void *upage, void *kpage, bool writable)
{
  struct thread *t = thread_current ();

  /* Verify that there's not already a page at that virtual
     address, then map our page there. */
  return (pagedir_get_page (t->pagedir, upage) == NULL
          && pagedir_set_page (t->pagedir, upage, kpage, writable));
}
```

`install_page` 에서 `pagedir_set_page` 를 통해 user page table entry에 kpage table entry의 복사본+ user flag를 넣음으로써 유효한 user page table entry를 생성하기에 유효한 user page table entry는 반드시 frame을 할당 받은 상태이게 된다.

Limitations and Necessity

기존의 Page Table과 Page Table Entry는 Frame을 할당 받은 Page에 대한 Page Table Entry만 Page Table에 유의미하게 존재하였다 (present bit이 0일시는 아무런 의미를 가지지 않는다). 즉 기존의 Page Table은 `Page를 할당 받는다 = Frame을 할당 받았다` 였다. 그렇기에 Virtual Memory에 존재하려면 반드시 Physical Memory에도 올라와 있어야 했다. 이로 인해 Physical Memory 보다 큰 크기의 파일을 읽거나 프로그램을 로드할 수 없어 실행할 수 없었다. virtual memory는 physical memory와 별개의 넓은 address space를 사용할 수 있을 것으로 기대하였으나 결국에는 남은 physical memory의 크기에 바운드되어 사용 가능한 virtual memory의 크기는 한정되었다.
만약 frame과 page의 연결성을 끊어준다면 frame을 할당 받지 않은 page를 생성할 수 있게 되며 보다 넓게 virtual memory를 사용할 수 있으며 physical memory 자원인 frame을 때에 맞게 page에 할당해줄 수 있게 된다. 기존 page table entry는 이처럼 frame과 page(user virtual page)의 연결성을 끊는 상황을 고려할 수 없다. 또한 기존 page table entry는 먼저 virtual page를 할당한 이후에 frame을 할당할 때/실제로 physical memory를 할당 받을 때 어떤 옵션으로 frame을 할당할지, 파일을 저장한 virtual page라면 파일의 어떤 부분을 로드해야 하는지를 포함할 수 없다.
이러한 문제를 개선하여 page와 frame을 연결성을 끊고 page를 frame과 무관하게 관리하게 위해 새로운 형식의 page table entry, 즉 보충된 정보들을 갖는 supplemental page table entry와 이들을 담는 supplemental page table이 필요해졌다.

Blueprint

우리는 Supplemental Page Table을 Pintos에서 제공하는 자료구조 `hash`, hash table을 이용해 구현하기로 결정하였다.

```
struct thread {
        ...
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
    struct hash s_page_table;
    struct process* process_ptr;
#endif
        ...
}
```

다음처럼 기존에 page directory에 대한 포인터를 저장하던 `thread->pagedir` 처럼 `thread` 에 supplemental page table `hash s_page_table` 을 추가한다. 이 때 `hash` 는 `list` 와 유사하게 ...
supplemental page table은 기존 page directory처럼 각 프로세스(핀토스는 프로세스-스레드 1대1)마다 각각 관리하게 된다.

```
struct s_page_table_entry
{
        bool present;
        bool in_swap;
        bool has_loaded;
        bool writable;
        bool is_dirty;
        bool is_accessed;
        bool is_lazy;
        struct file* file;
        off_t file_ofs;
        uint32_t file_read_bytes;
        uint32_t file_zero_bytes;
        size_t swap_idx;
        void *upage;
        void *kpage;
        enum falloc_flags;
        struct hash_elem elem;
}
```

`s_page_table_entry` 는 `s_page_table` 을 구성하는 page table entry이다.

- 기존 page table entry의 성분, lazy loading을 위한 성분, swap in/out을 위한 성분이 포함되어 있다.

멤버	자료형	설명
`present`	`bool`	현재 유효한 값을 가진 page table entry인지 여부 swap이든, lazy loading 중과 무관하게, frame 할당 여부와 무관하게 해당 page table entry의 정보가 유효한지 여부 해당 값이 false라면 아래 값들은 모두 무시되며, 다른 값들로 임의로 초기화될 수 있다.
`in_swap`	`bool`	현재 swap disk에 있는지 여부 swap disk에 있다면 true,
`has_loaded`	`bool`	lazy loading에서 사용되며 실제로 frame을 할당받았었는지 여부. lazy loading에서 load 전까지는 false
`writable`	`bool`	기존 pte의 writable bit
`is_dirty`	`bool`	기존 pte의 dirty bit
`is_accessed`	`bool`	기존 pte의 accessed bit
`is_lazy`	`bool`	lazy loading의 대상인지 여부
`file`	`struct file*`	어떤 파일을 loading해야하는지에 대한 변수 lazy loading에서 사용한다. `is_lazy` 가 참일 때 유효한 값.
`file_ofs`	`off_t`	`file` 의 어디부터 담은 page인지, lazy loading시 활용하기 위해 page만 미리 할당받을 때 해당 page가 file 중 어느 부분에 대한 것인지 정할 때 사용. `is_lazy` 가 참일 때 유효한 값.
`file_read_bytes`	`uint32_t`	기존 pintos 구현상 `load_segment` 에서 `page` 에 segment 저장시 사용하는 값. lazy loading에서만 사용되므로 `is_lazy` 가 참일 때만 유효함.
`file_zero_bytes`	`uint32_t`	기존 pintos 구현상 `load_segment` 에서 `page` 에 segment 저장시 사용하는 값. lazy loading에서만 사용되므로 `is_lazy` 가 참일 때만 유효함.

멤버	자료형	설명
`swap_idx`	`size_t`	swap table의 index `in_swap` 이 `true` 일 때만 유효한 값.
`upage`	`void *`	`user virtual page`
`kpage`	`void *`	`in_swap` 이 거짓이고 `has_loaded` 이 참일 때, 해당 user page `upage` 에 매핑된 frame(kernel virtual page) swap out, lazy loading에 의해 frame이 할당되지 않아 유효한 값이 아닐 수도 있다.
`falloc_flags`	`enum`	(lazy loading시) frame할당시 `falloc_get_page` 에서 사용할 옵션
`elem`	`hash_elem`	`s_page_table` 를 `hash` 로 구성하기 위한 `hash_elem`

```
void
init_s_page_table(hash* s_page_table)
{
        hash_init(s_page_table, s_page_table_hash_func, s_page_table_hash_less_func);
}
```

프로세스별 Supplemental page table인 `s_page_table` 을 `hash_init` 을 통해 초기화하는 함수이다. 기존에 `thread->pagedir` 을 초기화해주던 `load` (in `userprog/process.c`)에 해당 함수 호출을 추가한다.

```
unsigned
s_page_table_hash_func(const struct hash_elem *e, void *aux)
{
        struct s_page_table_entry *spte = hash_entry(e, struct hash_table_entry, elem);
        return hash_bytes(&spte->upage,32);
}
```

`hash_init` 에서 `s_page_table` hash를 초기화할 때 사용하는 hash function으로 `hash_hash_func` 형식을 따르는 함수이다.
`hash_bytes` 를 이용해 `s_page_table_entry` 의 `upage` 에 기반하여 해시한다.

- 기존 page table도 virtual page를 table의 index로 사용하였기 때문이며, 프로세스마다 `s_page_table` 이 존재하기에 `upage` 는 프로세스 내에서 유일한 값으로써 hash 값으로 사용하기 적절하다.

```
bool
s_page_table_hash_less_func(const struct hash_elem *a, const struct hash_elem *b, void *aux)
{
        struct s_page_table_entry *_a = hash_entry(a, struct hash_table_entry, elem);
        struct s_page_table_entry *_b = hash_entry(b, struct hash_table_entry, elem);
        return a->upage < b->upage;
}
```

`hash_init` 에서 `s_page_table` hash를 초기화할 때 사용하는 hash 대소 비교 function으로 `hash_less_func` 형식을 따르는 함수이다.
hash 값으로 사용하는 `upage` 를 유사하게 대소 비교 기준으로 사용하였다.

```
s_page_table_entry*
spte_create(bool is_lazy, struct file* file, off_t file_ofs, bool writable, void *upage, void *kpage, uint32_t
file_read_bytes, uint32_t file_zero_bytes, enum falloc_flags)
{
        struct s_page_table_entry *spte = malloc(sizeof *spte)
        spte->present = true;
        spte->upage = upage;
        spte->kpage = kpage;
        spte->writeable = writeable
        spte->falloc_flags = falloc_flags
        spte->is_lazy = is_lazy;
        spte->has_loaded = !is_lazy;
        spte->in_swap = false;
        spte->is_dirty = false;
        spte->is_accessed = false;
        spte->file = file;
        spte->file_ofs = file_ofs;
        spte->file_read_bytes = file_read_bytes;
        spte->file_zero_bytes = file_zero_bytes;
        hash_insert(&thread_current()->s_page_table,&spte->elem);
```

```
        return spte;
}
```

`s_page_table_entry`를 현재 스레드의 `s_page_table`에 추가하는 함수이다.

- 새로운 `s_page_table_entry`를 위한 공간을 할당 받고 `in_swap`, `in_dirty`, `in_accessed`를 false로 초기화해주고 입력 받은 값을 차례로 넣어준다. 이 때 `is_lazy`가 false라면 lazy loading이 아니므로 `upage`-`kpage` 매핑, 즉 frame을 할당 받은 상황이므로 `has_loaded`는 true로 설정해준다.
- 이처럼 생성 후 초기화한 `spte`를 현재 스레드(즉 현재 프로세스)의 `page table` 에 `hash_insert`를 이용해 추가한다.

```
s_page_table_entry*
find_s_page_table_entry_from_upage(void* upage)
{
        struct s_page_table_entry temp;
        temp->upage = upage;
        struct hash_elem *finded_elem= hash_find(&thread_current()->s_page_table,temp->elem)
        if(finded_elem == null)
                return null;
        return hash_entry(finded_elem, struct s_page_table_entry, elem);
}
```

입력 받은 user virtual page `upage`를 `s_page_table_entry`의 `upage`로 가지는 `s_page_table_entry`와 동일한 hash 값을 가지는 hash_elem이 현재 스레드의 `s_page_table`에 존재하는지 찾는다. 찾지 못하였다면 null을 리턴하고 찾았다면 해당 hash_elem을 가지는 entry 즉 `upage`를 `upage`로 가지는 `s_page_table_entry`의 주소를 반환한다.

- `s_page_table_hash_func`에서 `upage`를 hash 값 생성시 사용하였기에 `upage`만 집어 넣은 `s_page_table_entry`를 `hash_find`에서 이용하여 `upage`를 `upage`로 가지는 `s_page_table_entry`를 찾을 수 있다.
- `upage` user virtual page를 할당 해제하며 `spte_delete`를 호출할 때 함께 사용될 수 있다.

```
void
spte_delete(s_page_table_entry* spte)
{
        spte->present = false;
        hash_delete(&thread_current()->s_page_table, &spte->elem);
        free(spte);
}
```

`s_page_table`에서 `spte` `s_page_table_entry`를 제거하는 함수이다.
`present`를 false로 변경하여 더이상 유효하지 않음을 나타내고 `hash_delete`를 통해 현재 스레드의 `s_page_table`에서 제거한다. 마지막으로 해당 `s_page_table_entry`에 할당된 공간을 할당 해제한다.

- `spte`의 `upage`가 할당 해제되었을 때 해당 함수가 호출될 수 있다.
  - 이 때 user virtual page의 할당 해제란 단순한 frame 할당 해제(swap out)와는 다르며 해당 user virtual page 자체가 유효하지 않은 (virtual, physical memory에서 존재하지 않는) page가 되었음을 나타낸다.

4. Stack Growth

Basics

```
static bool
setup_stack (void **esp)
{
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
        *esp = PHYS_BASE;
      else
        palloc_free_page (kpage);
    }
```

```
  return success;
}
```

user program을 load하는 `load` 에서 `setup_stack` 을 호출하여 user virtual memory의 가장 top(`PHYS_BASE`)에서 주소가 감소하는 방향으로 user stack을 위한 공간 page 하나를 할당한다.

- `palloc_get_page (PAL_USER | PAL_ZERO)` 를 통해 0으로 초기화된 user pool로부터 얻은 page를 할당 받는다. 이를 `install_page` 를 통해 `PHYS_BASE-PGSIZE` user virtual address와 매핑한다.
  이후 `esp` 에 stack의 top (`PHYS_BASE`)을 저장한다. 즉 user stack은 `PHYS_BASE` 로부터 주소가 감소하는 방향으로 신장한다.
  하지만 user stack을 위한 공간은 `setup_stack` 에서 정해진 뒤 변경되지 않는다.

Limitations and Necessity

user stack을 위한 공간은 처음 할당 받은 1 페이지에서 더 커지지 않는다. 그렇기에 만약 user stack이 길어져 처음 할당 받은 1페이지(4KB)을 넘어선다면 page fault를 발생하게 되고 현재 구현상 `sys_exit(-1)` 을 통해 유저 프로세스를 종료하게 된다. 하지만 이는 page에 할당될 frame이 부족하여 생기는 문제가 아닌 단순히 user stack을 위한 공간이 1페이지로 한정되어 생기는 문제이다. 해당 문제는 user stack에 할당된 공간을 넘은 접근으로 인해 page fault가 발생시 user stack을 위한 공간을 이어서 더 할당함으로써 해결할 수 있다.

Blueprint

user stack의 공간 부족으로 인해 page fault 발생시 user stack을 위한 공간을 확장하기 위해서는 page fault 발생시 해당 page fault가 user stack의 공간 부족으로 인한 것임을 판별할 수 있어야 한다. 이를 위해 page fault handler `page_fault` 에 다음을 추가한다.

```
static void
page_fault (struct intr_frame *f)
{
  ...
  /* User caused page fault */
  if(is_valid_page(fault_addr) && load_frame_mapped_page(fault_addr))
  {
    pass
  }

+ if(fault_addr<PHYS_BASE && f->esp - 32 < fault_addr )
+ {
+   struct thread* t= thread_current();
+   void *kpage = falloc_get_page_w_upage(FAL_USER|FAL_ZERO, t->last_stack_page-PGSIZE);
+   if(!install_page(t->last_stack_page-PGSIZE,kpage,true))
+   {
+     falloc_free_page(kpage);
+   }
+   spte_create(..., t->last_stack_page-PGSIZE, kpage, ...);
+   t->last_stack_page = t->last_stack_page-PGSIZE;
+ }
+ else if(fault_addr<PHYS_BASE && t->esp - 32 < fault_addr)
+ {
+   //same to upward
+ }
  else sys_exit(-1);
```

page fault를 일으킨 주소가 user virtual address에 있고 그 상황에 esp(`f->esp`)-32보다 fault_addr이 클 때 user stack 공간 부족으로 인한 page fault로 판별한다.

- `80x86` 에서 stack에 값을 집어넣는 명령어는 `push` , `pusha` 가 있는데 이는 각각 4바이트, 32바이트를 stack에 집어넣는다. 그렇기에 esp보다 최대 32까지 작을 수 있다.
- 이 경우 `fallow_get_page_w_upage` 를 통해 현재 스레드의 `last_stack_page` - `PGSIZE` (user virtual page)에 frame을 할당한다.
  - `last_stack_page` 는 해당 스레드의 user stack 용도로 마지막에 할당된 page의 시작 주소이다.
- 이후 `install_page` 를 통해 매핑한 뒤 Supplemental page table에 해당 page-frame에 대한 entry를 추가한다.
- 마지막으로 `last_stack_page` 를 현재 할당한 페이지 시작 주소로 변경한다.
  `f->esp` 가 정의되지 않은 경우(처음으로 user->kernel mode 전환시)을 고려하기 위해 `t->esp` (현제 스레드의 `esp` 멤버)를 esp로 생각하여 동일한 과정으로 처리한다.
- user->kernel mode 로 전환될 때 에러가 발생할 때만 `f->esp` 값을 설정한다.

```
struct thread
{
```

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    ...
+   void* last_stack_page;
+   void* esp;
    ...
#endif
}
```

위의 `page_fault` 에서는 `thread` 의 새로운 멤버 `last_stack_page` 와 `esp` 를 사용한다.
`thread->esp` 은 timer interrupt의 tick마다 갱신해준다.
`last_stack_page` 는 처음 user process의 초기화 중 `setup_stack` 에서 초기화해주며, 이후 `page_fault` 에서 새로운 stack page를 할당할 때마다 업데이트한다.

5. File Memory Mapping

Basics

File Memory Mapping이란 파일을 가상 메모리 위의 연속적인 공간에 맵핑한 뒤, 일반적인 가상 메모리 접근과 동일하게 데이터에 접근하는 기법을 뜻한다.

```
mapid_t
mmap (int fd, void *addr)
{
  return syscall2 (SYS_MMAP, fd, addr);
}

void
munmap (mapid_t mapid)
{
  syscall1 (SYS_MUNMAP, mapid);
}
```

Pintos 프로젝트에서 이를 구현하기 위해서는 프로젝트 2에서 구현한 시스템 콜 핸들러에 추가로 `mmap` 과 `munmap` 시스템 콜을 구현해야 한다.

Limitations and Necessity

- 파일의 특정 위치의 데이터를 읽어야 하는데 페이지 테이블 위에 정보가 없을 때만 page fault handler를 통해 해당 페이지 정보를 읽어오는 Lazy loading이 일어나기 때문에 파일을 한 번에 읽어오는 방법보다 오버헤드가 적고, page hit일 경우 기존 데이터를 재사용 가능하다는 장점이 있다.
- 또한 파일에 write 연산을 수행할 경우 디스크에 직접 연산이 일어나는 것이 아니라 메모리 상에만 수정사항이 반영되고, 추후에 페이지가 evict될 때 수정사항을 디스크에 한 번에 반영하여 write할 때도 오버헤드가 줄어드는 효과가 있다.

File Memory Mapping과 디스크 직접 접근에 대한 장단점을 정리하자면 다음과 같다.

	장점	단점
File Memory Mapping	Lazy Loading을 통한 오버헤드 감소, Page hit일 때 데이터 재사용 가능	파일 크기 변경 불가, 메모리 부족으로 인한 파일 맵핑 크기 제한
디스크 직접 접근	메모리 부족 X	잦은 접근에 대한 오버헤드 높음

현재 Pintos 프로젝트의 구현의 경우 File Memory Mapping이 구현되어있지 않기 때문에 대용량의 파일에 접근해야 할 경우 시스템 콜을 통해 디스크에 직접 접근해야 하며, 실행 파일 역시 데이터 전체를 메모리 위로 복사해야 하기 때문에 오버헤드가 크다.

Blueprint

각 프로세스는 여러 개의 파일에 접근해 `mmap` 을 호출할 수 있고, `mmap` 으로 생성된 각 맵핑은 메모리 상의 여러 페이지와 대응된다. 따라서 이를 일관적으로 관리하기 위해서는 파일과 페이지의 연결 관계를 2차원 연결 리스트 형태로 나타내어 프로세스 별로 관리하는 것이 옳다고 생각되어 다음과 같이 구현 계획을 세웠다.

```
struct fmm_data
{
  mapid_t id;
```

```
  struct file *file;

  struct list page_list;
  struct list_elem fmm_data_list_elem;
}
```

우선 `mmap`으로 맵핑된 각 파일에 대한 페이지들을 관리하기 위해 위와 같은 구조체를 선언한다. `id`는 각 맵핑마다 할당되는 고유한 넘버링이고, `*file`은 어떤 파일이 맵핑되었는지를 나타낸다. 해당 파일에 대한 접근으로 생성되는 페이지는 `page_list` 리스트에서 일괄적으로 관리한다.

```
struct process
{
  ...
  struct list fmm_data_list;
  ...
}
```

그리고 프로세스 구조체 아래에 `fmm_data` 구조체들을 관리할 리스트인 `fmm_data_list`를 추가한다. `fmm_data` 구조체의 `fmm_data_list_elem`는 위 리스트의 원소로 사용되기 위해 필요하여 추가하였다. 파일에 대한 접근 및 맵핑 정보는 프로세스 별로 독립적이고, 한 프로세스 아래의 스레드들은 모두 동일한 가상 메모리 공간을 공유하므로 파일들에 대한 맵핑 정보인 `fmm_data_list`는 스레드가 아닌 프로세스에서 관리하는 것이 옳다고 판단하였다.

```
mapid_t
sys_mmap (int fd, void *addr)
{
  if(
    /* empty file */ ||
    /* addr is not page alligned */ ||
    /* mapped page already exists in the range */ ||
    addr == NULL ||
    fd < 2
    )
      return MAP_FAILED;

  /* Create new struct fmm_data, initialize it and push into the list */
  /* allocate new mapid for new fmm */
  return mapid;
}
```

`mmap` 시스템 콜은 위와 같이 validity 확인을 해준 뒤 새 `fmm_data` 구조체를 동적할당받아 맵핑 정보를 설정해주는 식으로 구현할 예정이다.

```
void
sys_munmap (mapid_t mapid)
{
  if(/* not a valid mapid */) return;

  struct fmm_data *fmm;
  for(/* i in fmm_data_list */)
  {
    if(i->id == mapid) fmm = i, break;
  }

  for(/* p in fmm->page_list */)
  {
    if(/* p is dirty */)
      /* update file on disk */
    /* delete p */
  }

  /* free fmm */
}
```

`munmap` 시스템 콜은 위와 같이 인자로 주어진 `mapid`에 대응하는 맵핑을 찾은 후, 해당 맵핑에서 생성한 페이지들을 모두 삭제해주도록 구현한다. 이때 페이지에 write 연산이 일어난 적이 있어 dirty bit가 설정되어 있을 경우 변경 사항을 디스크 위의 실제 파일에 반영해줘야 한다.

6. Swap Table

Basics & Limitations and Necessity

대량의 페이지를 할당해 사용할 경우 물리적 메모리가 부족해지는 상황이 생길 수 있는데, 현재 Pintos 구현에선 이러한 상황에서 추가로 페이지 할당을 시도할 시 별다른 처리 없이 실패하게 된다. 따라서 물리적 메모리가 모두 할당된 상황에서도 운영체제가 정상적으로 동작할 수 있도록 하기 위해선 자주 쓰이지 않는 페이지를 골라 할당 해제하거나 외부로 옮겨 여유 메모리를 확보해야 한다.

이때 외부로 옮겨질 페이지들을 저장해놓을 공간을 사전에 디스크에 할당하여 사용할 수 있다. 페이지를 디스크 메모리에 저장하거나 (Swap out) 필요할 때 다시 메모리로 불러오는 (Swap in) 동작을 수행함으로써 물리적 메모리가 모자란 상황에서도 유연하게 메모리 할당 및 접근을 가능하게 한다. 이렇게 디스크 위에 할당한 저장공간을 **Swap Block**이라 하며, 이를 관리하는 테이블을 **Swap Table**이라고 한다.

다만 이러한 동작은 모두 디스크 읽기 및 쓰기 동작을 수반하고, Swap이 끝나기 전까지는 해당 프로세스의 메모리 접근이 막히므로 빈번하게 일어날 시 큰 성능 저하로 이어질 수 있다. 따라서 어떤 페이지를 할당 해제할 지 선택하는 규칙인 **Page Replacement Policy**가 미래에 사용될 가능성이 가장 낮은 페이지를 우선으로 선택하도록 잘 구성해줘야 한다.

Blueprint

Swap Block의 크기는 부팅 시 결정되기 때문에 변경되지 않고, 특정 위치의 페이지가 현재 할당중이었는지를 빠르게 알아내는 것이 중요하므로 `bitmap` 자료구조를 사용하는 것이 적합하다고 판단했다. Swap Block에 저장 가능한 페이지의 수는 `(Swap Block의 크기) / (페이지 크기)`로 계산 가능하므로, 해당 크기만큼의 비트를 가지는 비트맵을 할당하여 특정 비트의 값이 `0`인지 `1`인지로 해당 위치의 페이지 존재 여부를 알아낼 수 있다.

- Swap block에 새로운 페이지를 저장하고자 할 때 저장 가능한 위치를 반환해줄 수 있어야 한다. 이는 현재 최대 페이지 수와 할당 해제된 인덱스를 관리하는 리스트를 이용하여 빠르게 반환하거나 비트맵 자료구조의 `bitmap_scan` 함수를 이용하여 처음으로 `0`이 등장하는 인덱스를 찾는 방법으로 구현 가능한데, 구현의 편의성을 위해 이번 프로젝트에선 후자를 선택할 예정이다.
- 디스크 공간은 모든 프로세스에서 공유하기 때문에 Swap block은 운영체제 상에서 전역으로 선언되어 접근 가능해야 한다.

현재 Evict하고자 하는 페이지가 일반적인 메모리 페이지인지, 혹은 앞서 살펴본 `mmap`에 의해 할당되어 특정 파일을 가리키고 있는 페이지인지에 따라서 Swap in과 Swap out의 동작이 달라진다. 현재 페이지가 메모리 페이지인지 파일 맵핑 페이지인지는 위에서 선언한 Supplemental Page Table Entry를 참조하여 구분할 수 있다.

Swap out

```
bool
swap_out(struct s_page_table_entry *page)
{
  /* Validity check */

  if(page->is_lazy)
  {
    /* File mapped page */
    if(page->is_dirty)
    {
      /* Update data on disk file */
    }
    /* Free the page, no need to swap */
  }
  else
  {
    /* Memory page */
    int idx = /* Get available swap table entry */
    /* Copy page data to swap block */
  }
}
```

Swap out 동작은 위와 같이 구현할 수 있다.

- 만약 `page`가 파일에 맵핑되어있어 `is_lazy` 플래그가 설정되어 있는 페이지라면 Swap block에 따로 데이터를 옮기지 않고 페이지를 해제해준다. 다만 `is_dirty` 플래그가 설정되어있어 파일에 수정사항이 있음을 나타낼 시 해당 데이터를 Swap block이 아닌 실제 파일에 적용시켜준다.
- 일반적인 메모리 페이지라면 Swap table로부터 할당 가능한 인덱스를 하나 받아와 해당 위치에 페이지 정보를 복사한 뒤 페이지를 해제해준다.

Swap in

```
bool
swap_in(struct s_page_table_entry *page, void *paddr)
{
  /* Validity check for paddr, page->swap_idx */

  /* Memory page (File mapped page would not go here) */
  /* Copy page data from swap block */
  /* Unset swap table entry */
}
```

Swap in 동작은 위와 같이 구현 가능하다. 파일 맵핑된 페이지는 Swap out되지 않으므로 메모리 페이지에 대해서만 구현을 해주면 되는데, 대상 Swap table entry와 페이지를 Swap in 하고자 하는 물리적 메모리 주소인 `paddr` 에 대한 validity를 체크해준 후 디스크로부터 페이지 데이터를 복사해 메모리에 작성한다. 마지막으로 할당되어있던 Swap table entry의 비트를 `0` 으로 바꿔줌으로써 할당을 해제한다.

Page Replacement Policy

물리적 메모리에 할당되어있는 frame entry 중 적절한 페이지를 찾아 Swap이 최대한 덜 일어나도록 페이지를 evict해야 하는데, frame table의 `use_flag` 를 이용해 clock 알고리즘을 구현하여 효과적인 Page Replacement Policy를 구현할 계획이다.

7. On Process Termination

Basics & Limitations and Necessity

프로세스가 종료된 뒤 동적할당했던 메모리 및 자원들을 제대로 반환해주지 않을 시 메모리 누수로 이어질 수 있기 때문에 빠짐없이 해제해줘야 한다. Pintos Project 2 종료 시점 기준으로 프로세스 종료 시 메모리 누수를 검사하는 테스트케이스가 있었고 이를 통과했으므로 현재는 메모리 누수가 없는 상황이지만, 이번 Project 3에서 추가로 할당한 메모리들에 대해서는 추가적으로 할당 해제해주는 과정이 필요하다.

Blueprint

```
void
sys_exit (int status)
{
  ...
  /* Call sys_munmap for all mapped file */
  /* Free SPT and following entries */
  /* Free Swap table and pages */
  /* Write to the disk when dirty bit is set */
  ...
}
```

프로세스가 종료될 때의 동작은 Project 2와 마찬가지로 `sys_exit` 에서 구현한다. `mmap` 으로 맵핑한 모든 파일을 `munmap` 으로 다시 해제해줘야 하고, SPT와 그 엔트리들, Swap table까지 모두 할당 해제해줘야 한다. 페이지를 해제하는 중 dirty bit가 설정되어있을 시 해제하기 전 데이터의 변경 사항을 디스크에 반영해줘야 한다.