[CSED312] OS project 3 Design Report

- 20180673 하재현
- 20180501 최진수

Analysis on current Pintos system

현재 pintos system 상에서 virtual memory 자체는 하나도 구현되어 있지 않다.

따라서 전체적인 메모리 시스템에 대해서 설명하겠다.

Memory Structure

핀토스는 virutal memory를 user memory와 kernel memory, 이 두개의 pool로 나눈다.

User memory의 경우 0x0에서 PHYS_BASE까지이다.

Kernel memory의 경우 PHYS_BASE에서 메모리의 끝 까지이다.

이때, Kernel address와 physical address의 mapping은 physical address = kernel virtual address - PHYS_BASE 으로 이루어진다.

그리고 제공되는 스택은 fixed size stack (1 page)이다.

Address Translation

핀토스의 주소 번역은 두 단계로 이루어진다.

이때, 상위 20bit는 번역에 쓰이고, 하위 12bit는 offset에 쓰인다. (Page size가 4KB이므로)

첫 번째 단계는 상위 10bit를 page directory index로 사용하여 page directory를 참조 해, page table을 얻어낸다.

두 번째 단계는 그 다음 10bit를 page table index로 사용하여 첫 번째 단계에서 구한 page table을 참조해, page를 얻어낸다.

만약 이 lookup이 실패하면 page fault가 발생한다.

```
static void
page_fault (struct intr_frame *f)
{
  ...

  /* 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);
}
```

현재로써는 page fault가 발생할 시, 곧바로 process가 종료된다.

Dirty & Accessed Bit

Page Table Entry는 dirty bit와 accessed bit라는 추가 정보를 포함하고 있다.

Dirty bit는 수정이 일어났음을 표기하고, accessed bit는 접근이 일어났음을 표기한다.

아래는 각 bit를 수정하고, 얻는 함수이다.

```
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;
}

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);
        }
    }
}

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;
}

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);
          }
      }
  }
```

Loading Program

현재 핀토스는 실행에 필요한 데이터를 모두 메모리에 적재한 후 실행하고 있다.

이로인해 메모리가 효율적으로 사용되지 못하고 있다.

```
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;
}

static bool
setup_stack (void **esp, char* file_name)
{
  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);
    }

  argument_passing(esp, file_name);

  return success;
}
```

`load_segment`는 메모리 적재를 담당하는 함수이다.

`setup_stack`은 스택 적재를 담당하는 함수이다.

Access File

메모리 매핑이 구현되어있지 않다.

Solutions for each requirement

1. Frame Table

1. Data Structure

```
// in frame.h
struct frame
{
  struct lock lock;        // Lock for synchronization
  void *kpage;             // kernel virtual address
  struct page* page;       // corresponding page
```

```
  bool is_loaded;          // is this frame loaded?
  struct list_elem elem;   // list elem
}

// in frame.c
struct list frames;
```

2. Create

전부 다 `vm/frame.c`안에 구현

```
void frame_init(void)
{
  /*
  Frames list를 초기화한다.
  각 frame의 kpage를 palloc_get_page를 통해 얻는다.
  이때 user pool에서 얻어야 하므로 PAL_USER falg를 준다.
  로딩이 안된 상태이므로 padge = null, is_loaded = false이다.
  더 이상 페이지를 얻을 수 없을 때까지 반복한다.

  이 함수는 핀토스 초기화 과정 때 호출된다 (init.c)
  */
}
```

```
void frame_lock_aquire(struct page* p)
{
  /*
  주어진 page의 frame의 lock을 acquire한다.
  Frame에 동시접근하는 것을 막기 위해서이다.
  */
}
```

```
void frame_lock_release(struct page* p)
{
  /*
  주어진 page의 frame의 lock을 release한다.
  Frame에 동시접근하는 것을 막기 위해서이다.
  */
}
```

```
struct frame* frame_allocate(void)
{
  /*
  Frame을 할당한다.
```

```
  우선 Free frame이 있는지 탐색한다.
  있다면, 해당 free frame을 반환한다.
  없다면, eviction을 해야한다.
  frame_eviction()을 호출하여 eviction을 한다.
  Evict되어 free해진 frame을 반환한다.

  여러 동작 전후로 lock을 걸어 synchronization을 한다.
  */
}
```

```
struct frame* frame_eviction()
{
  /*
  Algorithm에 따라 선택된 frame을 evict 한다.
  Evict를 위해 page = null로 초기화하고, page_out 함수를 통해 page out을 처리한다.
  Eviction algorithm은 clock algorithm을 목표로 한다.

  Evict된 frame을 반환한다.
  */
}
```

```
void frame_deallocate(struct frame*)
{
  /*
  frame->page = null을 통해 등록된 page를 제거한다.
  */
}
```

3. Algorithm

1. Allocate / deallocate frames

   Allocation은 `frame_allocate`을 통해 할 수 있다.

   Deallocation은 `frame_deallocate`를 통해 할 수 있다.

2. Choose a victim which returns occupying frames when free frame doesn't exist

   `frame_allocate`에서 eviction 여부를 판단해서, `frame_eviction`에서 eviction을 수행하게 된다.

   이때 알고리즘은 clock algorithm의 구현을 목표로 한다.

3. Search frames used by user process

   해당 frame의 소유 thread는 `page->thread`를 통해 얻을 수 있다.

Frame table은 global하게 thread들이 공유한다.

Frame table의 초기화는 핀토스의 초기화 과정과 동시에 이루어진다.

초기화를 통해 user pool의 크기만큼의 크기를 할당받고, kpage를 부여받게 된다.

이후 page fault가 났을 때, page는 `frame_allocate`를 통해 frame을 할당받아 page와 연결하게 된다.

동시에 frame은 is_loaded flag가 올라가며, 자신과 연결된 page를 등록한다.

PDF에서 요구한 Modify process loading은 아래에서 설명한다.

2. Lazy Loading

1. Data Structure

```
struct page
{
  /*
  See Data structure of 3.Supplemental Page Table
  */
}
```

```
struct frame
{
  /*
  See Data structure of 1.Frame Table
  */
}
```

2. Modify

- In `userprog/process.c`

```
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)
    {
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      /* Remove loading part. Replace with page initation for lazy loading
*/
      /*
      uint8_t *kpage = palloc_get_page (PAL_USER);
```

```
        if (kpage == NULL)
          return false;

        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);

        if (!install_page (upage, kpage, writable))
          {
            palloc_free_page (kpage);
            return false;
          }
        */

        /* Replaced page allocation */
        page_allocate_with_file(upage, file, ofs, read_bytes, zero_bytes,
writable);

        read_bytes -= page_read_bytes;
        zero_bytes -= page_zero_bytes;
        upage += PGSIZE;
      }
    return true;
}
```

- In `userprog/exception.c`

```
static void
page_fault (struct intr_frame *f)
{
  ...

  if valid:
    call page_load(fault_addr)
    if success, return;

  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);
}
```

3. Algorithm

기존의 핀토스는 `load()`시 모든 데이터를 메모리에 올렸다.

Lazy loading을 위해서는 메모리에 올리는 대신, page에 loading에 관한 정보만 넣어둔다.

이후 `page_fault()`가 발생했을 때, lazy loading인지 판단하여 맞다면 page에 등록된 정보를 가지고 loading을 수행한다.

Lazy loading인지 판단하는 방법은 addr이 valid 하며, 해당 addr와 대응되는 page(해쉬를 통해 얻는다)에 file이 등록되어있는지 판단하면 된다.

3. Supplemental Page Table

1. Data Structure

```
// in vm/page.h
struct page
{
  struct hash_elem hash_elem;   // Hash 테이블을 위한 element
  struct thread *thread;        // Owner
  struct frame* frame;          // Mapped frame
  void *upage;                  // User virtual address
  struct file *file;            // file to read from
  bool writable;                // Writable access mode?
  uint32_t read_bytes;          // Bytes to read
  uint32_t zero_bytes;          // Bytes to fill with zero
  off_t ofs;                    // offset
}

// in threads/thread.h
struct thread
{
  ...
  struct hash* pages;
  ...
}
```

2. Create

전부 다 `vm/page.c`안에 구현

Memory mapping 관련 함수들은 5번에서 설명한다.

```
bool page_allocate_without_file(void *upage)
{
  /*
  Page를 새로 생성한다.
  upage와 thread를 등록하고, 나머지는 모두 null로 만든다.
  Page를 page table에 등록한다 (hash)
  성공 여부를 반환한다.
  */
}
```

```
bool page_allocate_with_file(void *upage, struct file *file, off_t ofs, uint32_t
read_bytes, uint32_t zero_bytes ,bool writable)
{
  /*
  Page를 새로 생성한다.
  인자들과 thread를 등록한다. 이때, frame은 null 상태이다 (다른 함수에서 할 것임)
  Page를 page table에 등록한다
  성공 여부를 반환한다.
  */
}
```

```
bool page_load (void *fault_addr)
{
  /*
  frame_allocate()를 통해 frame을 얻고, page에 등록한다.
  page를 load하는 경우는 총 3가지로 나뉜다.
  각각의 조건을 판단하고, 적절한 처리를 해준다.

  1. swap in
  Swap out을 한 적이 있어 sector가 등록되어있는 경우이다.
  swap in을 해준다. (구체적인 구현은 6번 참고)
  2. file (lazy loading)
  file이 등록되어있는 경우이다.
  load_segment()에서 하던 file reading을 해준다.
  Reading한 데이터를 memset을 통해 frame에 세팅해준다.
  3. zero (stack)
  이는 4번 Stack Growth의 구현이다.
  stack grow를 해준다 (구체적인 구현은 4번 참고)

  성공 여부를 반환한다.
  */
}
```

```
bool page_deallocate(void *upage)
{
  /*
  Hash table에서 page를 찾는다.
  Frame과의 mapping을 푼다 (frame_deallocate 호출)
  page에 할당된 것들을 모두 풀어준다.
  이때, 여러가지 동작 수행이 필요하다.
  삭제하고자 하는 페이지가 프로세스의 데이터 영역 혹은 스택에 포함 될 때, 이를 swap out
해준다.
  이때, dirty 상태라면 file에 작성도 해준다.
  작업 완료 후, Hash table에서 삭제한다.
  */
}
```

```
bool page_destory(void *upage)
{
  /*
  Hash table에서 page를 찾는다.
  Swapping 등의 처리 없이, 그냥 지워버린다.
  성공 여부를 반환한다.
  */
}
```

3. Algorithm

Supplemental Page Table은 lazy loading, mmap, swap 등이 동작하기 위한 여러가지 자료구조와 기능들을 정의하고 있다.

Page Table은 hash table로 구성되어 있으며, 각 thread마다 하나씩 가진다.

Page table의 page는 `page_allocate_without_file` 또는 `page_allocate_with_file`을 통해 할당된다.

이후 page fault가 일어났을 때, `page_load`를 통해 frame을 할당받고, 적절한 값이 들어가게 된다.

나중에 eviction 때문에 page 가 지워져야 할때에는 `page_deallocate`를 통해, frame과의 관계를 끊고 경우에 따라 write back과 swap out을 수행해준 후 할당 해제된다.

혹은 진짜로 그냥 page가 지워져야 하는 경우 (프로세스 종료 등의 이유로 인해) WB/swap out없이 할당해제한다. 이는 `page_destory`에 구현되어있다.

4. Stack Growth

1. Data Structure

```
struct page
{
  /*
  See Data structure of 3.Supplemental Page Table
  */
}
```

```
struct frame
{
  /*
  See Data structure of 1.Frame Table
  */
}
```

2. Modify

- In `userprog/exception.c`

```
static void
page_fault (struct intr_frame *f)
{
  ...

  if valid:
    call page_load(fault_addr)
    if success, return;

  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);
}
```

- In `vm/page.c`

```
bool page_load (void *fault_addr)
{
  /*
  ...
  3. zero (stack)
  pg_round_down으로 얻어낸 페이지 주소가 스택 영역 안에 있고, fault 주소가 스택
포인트 -32 보다 크다면 이를 수행한다.
  page_allocate_without_file로 page를 할당한다.
  이후 frame을 할당받아 zeroing을 해주고 등록한다.
  만들어진 page를 page table에 등록한다.
  만들어진 page를 page dir에 등록한다.

  성공 여부를 반환한다.
  */
}
```

3. Algorithm

Page fault가 일어났는데 해당 접근이 stack 영역 내라서 stack 확장이 필요할 경우에 동작한다.

Stack 확장은 우선 file 없이 page를 할당해준 후, frame을 할당받아 zeroing을 해주고 이를 page에 등록하면 된다.

5. File Memory Mapping

1. Data Structure

- `vm/page.h`

```
struct file_mapping{
  int mapid;
  struct file* file;
  struct list_elem elem;
  struct list frame_list;
};
```

file mapping이 이루어진 후 mapping된 page들을 관리할 자료구조가 필요하다.

`struct file_mapping` mapping이 이루어진 frame들을 리스트로 만들어 관리한다. 그리고 map id와 mapping을 한 file을 가리키고 있는 포인터도 함께 기억한다.

- `threads/thread.h`

```
struct thread{
  /***/
  struct list file_mapping_table;
  /***/
}
```

각 thread는 여러개의 파일을 mapping할 수 있다. `struct file_mapping`은 하나의 파일에 대해서 매핑된 frame들을 리스트로 엮어 관리하므로 `struct thread`는 이것들을 또 리스트로 묶어 관리해야 한다.

2. Create

- `userprog/syscall.c`

```
mapid_t mmap(int fd, void *addr)
{
  /*
  fd와 mapping을 시작할 주소인 addr를 인자로 받는다.
  fd가 가리키고 있는 파일에 대해 file_reopen()을 실행해 그 정보를 저장하고, mapid를 할당한다.
  mapping이 이루어진 frame(page)들을 모아서 file mapping entry를 만들고 file_mapping_table에 insert한다.

  만약 page table에서 virtual address에 해당하는 주소가 dirty하다면 write back을 실행해준다.
  */
}

void munmap(mapid_t mapping)
{
  /*
  특정 mapid가 가리키고 있는 file mapping entry를 삭제한다.
  (그 전에 file mapping entry가 관리하고 있는 frame들도 모두 삭제한다.)
  이후 파일을 닫는다.
```

```
  */
}
```

3. Modify

- `userprog/exception.c`

```
static void
page_fault (struct intr_frame *f)
{
  /*
  if file is mapped -> load data from the file
  */
}
```

`page_fault()` 함수는 page fault를 핸들링하는 역할을 맡는다. 하지만 이전의 구현에서는 executing file에 대한 demand paging만이 고려되었고 mapping된 파일에 대해서 demand paging이 일어나지 않고 있으므로 만약 file이 mapping되어 있다면 file data를 memory로 loading하는 과정이 추가되어야 한다.

- `userporg/process.c`

```
void
process_exit (void)
{
  /*
  ...
    mapping된 file들에 의해 할당된 frame들을 모두 할당 해제시켜주는 과정이 추가되어야 한다.
  ...
  */
}
```

4. Algorithm

- `mmap()`

1. 인자로 전해지는 fd와 addr가 valid한지 체크한다.
2. file_reopen을 이용해 file pointer을 반환하고 그것을 file mapping entry에 저장한다.
3. mapid를 할당한다.
4. file의 length만큼 page를 allocate한다. allocate된 page들은 file mapping entry 내의 list 안에 묶어서 저장한다.
5. mapid를 return한다.

- `munmap()`

1. mapid와 맞는 file mapping entry를 file mapping table 내에서 찾는다.
2. 해당하는 file mapping entry의 list 속의 frame들을 할당 해제한다.

3. file을 close한다.
4. file mapping entry를 file mapping table에서 제거한다.

6. Swap Table

1. Data Structure

별도로 table을 유지하지 않고, bitmap과 block을 통해 해결한다.

- In `vm/page.c`

```
struct page
{
  block_sector_t sector     // default -1
}
```

- In `vm/swap.c`

```
static struct block *swap_block_device;

static struct bitmap *swap_bitmap;

static struct lock swap_lock;
```

2. Create

전부 `vm/swap.c`에 구현한다.

```
void swap_init()
{
  /*
  Get block (block_get_role(BLOCK_SWAP))
  Init bitmap
  Init lock
  */
}
```

```
void swap_in (struct page*)
{
  /*
  Device으로부터 데이터를 읽어들여 frame에 저장한다.
  4096 / 512 block 만큼 읽어들인다.
  Bitmap에서 해당 page가 차지하던 영역을 초기화한다.
  */
}
```

```
void swap_out (struct page*)
{
  /*
  해당 page가 차지한 swap slot을 구한다
  Device에다가 frame의 데이터를 저장한다.
  bitmap에서 해당 swap slot을 할당됨 표시한다.
  */
}
```

3. Modify

Swap 관련해서 modify 할 내용은 3번의 `page_load`와 `page_deallocate`에 이미 적혀있다.

4. Algorithm

Page가 evict 될 때, swap을 해야 하는 경우라면 swap_out을 호출하여 swap slot을 차지하며 frame의 내용을 device에 저장해준다.

이때, 저장된 slot을 thread->slot에 저장해준다.

이후 `page_load`에 의해 다시 복귀할 때, thread->slot을 통해 저장된 slot을 알아내서 swap_in을 해준다. 이후 해당 slot은 다시 빈 상태로 바꾼다.

7. On Process Termination

1. Algorithm

Thread에 정의된 hash table을 통해 모든 s-page를 조회할 수 있다.

이 page를 조회하면서, 전부 destory 시켜준다.

이를 통해 page에 할당된 frame들이 free 될 수 있다.

마지막으로 s-page table을 날려준다.