**Pintos Project 4: Virtual Memory**

담당 교수 : 박성용

이름 / 학번 : 이은지 / 20170101

개발 기간 : 2020.12.10 ~ 2020.12.22

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

가상 메모리를 구현하여 page fault를 핸들링.

스택에 한 페이지 이상을 할당가능케 하여(stack growth) 메모리 접근 에러를 조정.

(주요 개발 목표는 아니나 일부 테스트케이스 통과를 위해 파일을 가상메모리에 매핑하는 것 또한 필요함)

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Page Table & Page Fault Handler

기존 페이지 테이블에 필요한 정보를 추가하되, 특히 페이지를 swap하는 데 필요한 정보를 추가한다. 이를 바탕으로 page fault를 구현하고 physical memory로 매핑할 수 있게 한다.

2. Disk Swap

디스크에서 페이지를 swap in하거나 디스크로 페이지를 swap out하여 페이지테이블에 페이지가 없어 페이지 폴트를 일어나는 것을 방지한다.

3. Stack Growth

만약 적절한 이유로 인해 user 프로그램이 스택을 초과하여 page fault가 발생하면 page fault 발생 주소를 통하여 page를 새로 할당한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술
  1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

페이지 테이블에 없는 페이지에 접근을 시도한 경우, 그리고 사용할 수 있는 영역을 벗어난 메모리에 접근을 시도하면 page fault가 발생한다. 이 때 supplementary table을 통하여 적절한 접근 유무를 판단하고, 만약 잘못된 영역 접근이 아니고 그저 swapped out된 경우나 지연 로딩되어 디스크에 있으면 해당 페이지를 찾아내 reload한다.

* 1. Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술

frame list를 관리할 때 유사 LRU 알고리즘인 second chance 알고리즘을 활용하였다. 해당 리스트를 순회하며 swap out 될 frame을 victim\_ptr을 통하여 선정한다. 만약 이미 reference 되었다면 (reference = 1) 이를 0으로 세팅하며 기회를 재제공하고, 다음 엔트리로 포인터를 넘긴다.

* 1. Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술

유저 메모리의 스택의 접근에 의해서만 유저의 접근이라고 판단하기 때문에 유저 메모리 스택 포인터 esp를 확인한다. 하지만 이는 유저 모드에 국한된 경우이고, 만약 커널모드의 read나 write 시스템 콜의 신호를 받을 경우에서 page fault가 일어날 수도 있다. 이를 대비해서 thread에 current\_esp라는 포인터를 통해 시스템 콜을 불러오기 직전의 스택 포인터를 저장한 뒤 활용한다.

이외에도 핀토스 매뉴얼 p.46에 따르면 PUSH/PUSHA와 같은 명령어는 미리 스택 포인터를 확인하여 page fault를 일으키므로 각각 명령어의 크기인 4바이트/ 32바이트까지도 확인해야한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

12/10 – 12/12 : vm 폴더 내 (frame, page,swap) 각각의 함수 고안

(12/13 – 12/18 : 기말시험)

12/19 – 12/22: process 내 적용 및 page fault 개선

12/22 : 최종 확인

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드

- vm/page.c, vm/page.h 생성 : supplementary table 관련함수 생성. 페이지폴트 시의 handle\_mm\_fault와 expand\_stack도 여기서 해결.

- vm/frame.c, vm/frame.h 생성 : virtual memory와 대응하여 frame에 메모리를 할당한다. 그리고 frame 교체 시에 필요한 추가 함수 또한 이 파일에 선언되어있다.

- vm/swap.c, vm/swap.h 생성 : 페이지 테이블과 디스크 사이 swap을 도와주는 swap in, swap out 등이 선언되어있다.

- userprog/exception.c : 페이지 폴트에 stack growth와 page\_fault handling 포함

- userprog/process.h, userprog/process.c 수정 :virtual memory의 경우를 반영하여 virtual memory에 대응하여 frame 관리.

- userprog/init.c 수정 : page\_fault handling에 필요한 리스트 초기화

수정하거나 추가해야 하는 자료구조

Bitmap – 디스크와 테이블 간에 간편한 swap을 위해 도입된 자료구조이다. swap 시에 사용 가능한 swap slot을 알아내기 위해 사용된다. 크기는 전체 block 크기 / 페이지 크기이다.

Linked List – frame entry들을 리스트로 관리한다. 뿐만 아니라 memory mapped files들도 리스트로 관리된다.

Stack – 이는 특정한 요소가 아니라 전반적인 구현에 있어 반드시 이해되어야 하고 페이지를 통한 할당이 필요하다.

Hash Table – Supplementary Table의 페이지들은 hash 함수를 통해 매핑된다. 그 중에서 kpage를 key로 삼아 접근되고 변경된다.

* + 수정하거나 추가해야 하는 함수

vm/frame.c

/\* Initialize \*/

void vm\_frame\_init(void);

/\* Create a frame page corresponding to user virtual address upage.

After the page mapping, return the kernel address of created page frame. \*/

void\* vm\_frame\_allocate(enum palloc\_flags flag, void \*upage);

/\* Free the page frame.

Remove the entry in the frame table, free the memory resource. \*/

void vm\_frame\_free(void\* kpage);

/\* Just remove the entry from table, do not palloc free \*/

void vm\_frame\_remove\_entry(void \*kpage);

void vm\_frame\_do\_free (void \*kpage, bool free\_page);

/\*For pinning\*/

void vm\_frame\_pin(void\*kpage);

void vm\_frame\_unpin(void \*kapge);

vm/page.c

/\* Functions for manipulating supplemental page table. \*/

struct supplemental\_page\_table \*vm\_supt\_create(void);

void vm\_supt\_destroy(struct supplemental\_page\_table \*supt);

bool vm\_supt\_install\_frame(struct supplemental\_page\_table \*supt, void \*upage, void \*kpage);

bool expand\_stack(struct supplemental\_page\_table \*supt, void \*);

bool vm\_supt\_install\_filesys(struct supplemental\_page\_table \*supt, void \*page,

struct file \*file, off\_t offset, uint32\_t read\_bytes, uint32\_t zero\_bytes,bool writable);

bool vm\_supt\_set\_swap(struct supplemental\_page\_table \*supt, void \*, swap\_index\_t);

struct supplemental\_page\_table\_entry \*vm\_supt\_look\_up (struct supplemental\_page\_table \*supt, void \*);

bool vm\_supt\_has\_entry(struct supplemental\_page\_table \*supt, void \*page);

bool vm\_supt\_set\_dirty(struct supplemental\_page\_table \*supt, void \*, bool);

bool handle\_mm\_fault(struct supplemental\_page\_table \*supt, uint32\_t \*pagedir, void \*upage);

bool vm\_supt\_mm\_unmap(struct supplemental\_page\_table \*supt, uint32\_t \*pagedir,

void \*page, struct file \*file, off\_t offset, size\_t bytes);

void vm\_pin\_page(struct supplemental\_page\_table \*supt, void \*page);

void vm\_unpin\_page(struct supplemental\_page\_table \*supt, void \*page);

void preload\_and\_pin\_pages(const void \*, size\_t);

void unpin\_preloaded\_pages(const void \*, size\_t);

vm/swap.c

/\* Functions for swap table. \*/

/\* Initialize the swap table. \*/

void vm\_swap\_init(void);

/\* swap out : write the content of page into the swap disk

and return the index of swap region. \*/

swap\_index\_t vm\_swap\_out(void \*page);

/\* swap in : read the content through the index from the mapped block

and store into page. \*/

void vm\_swap\_in(swap\_index\_t swap\_index, void \*page);

/\* Free the swap region. \*/

void vm\_swap\_free(swap\_index\_t swap\_index);

userprog/process.c

// Lazy load. . .

struct thread \*curr = thread\_current ();

ASSERT (pagedir\_get\_page(curr->pagedir, upage) == NULL); // no virtual page yet?

if (! vm\_supt\_install\_filesys(curr->supt, upage,

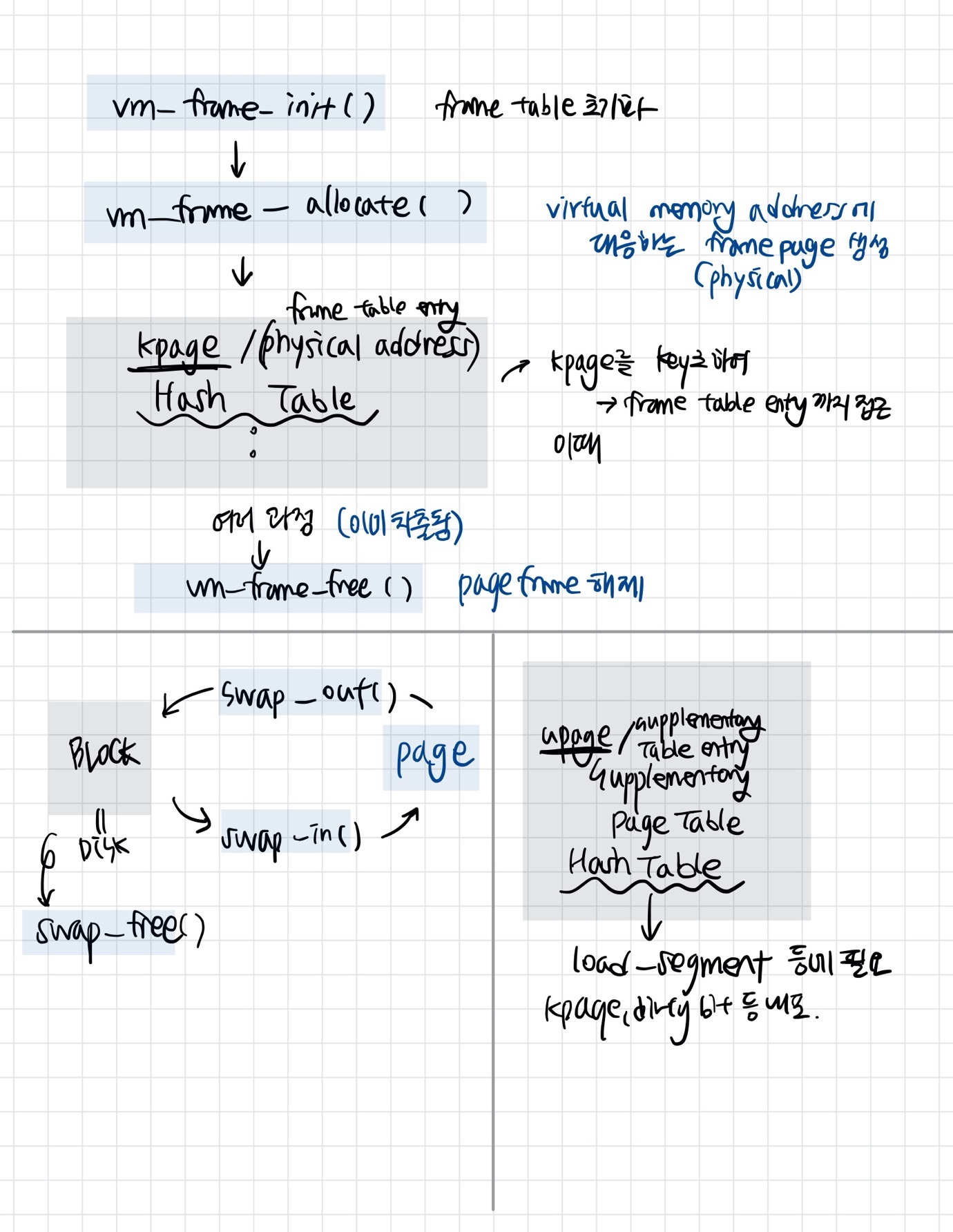
file, ofs, page\_read\_bytes, page\_zero\_bytes, writable) ) {

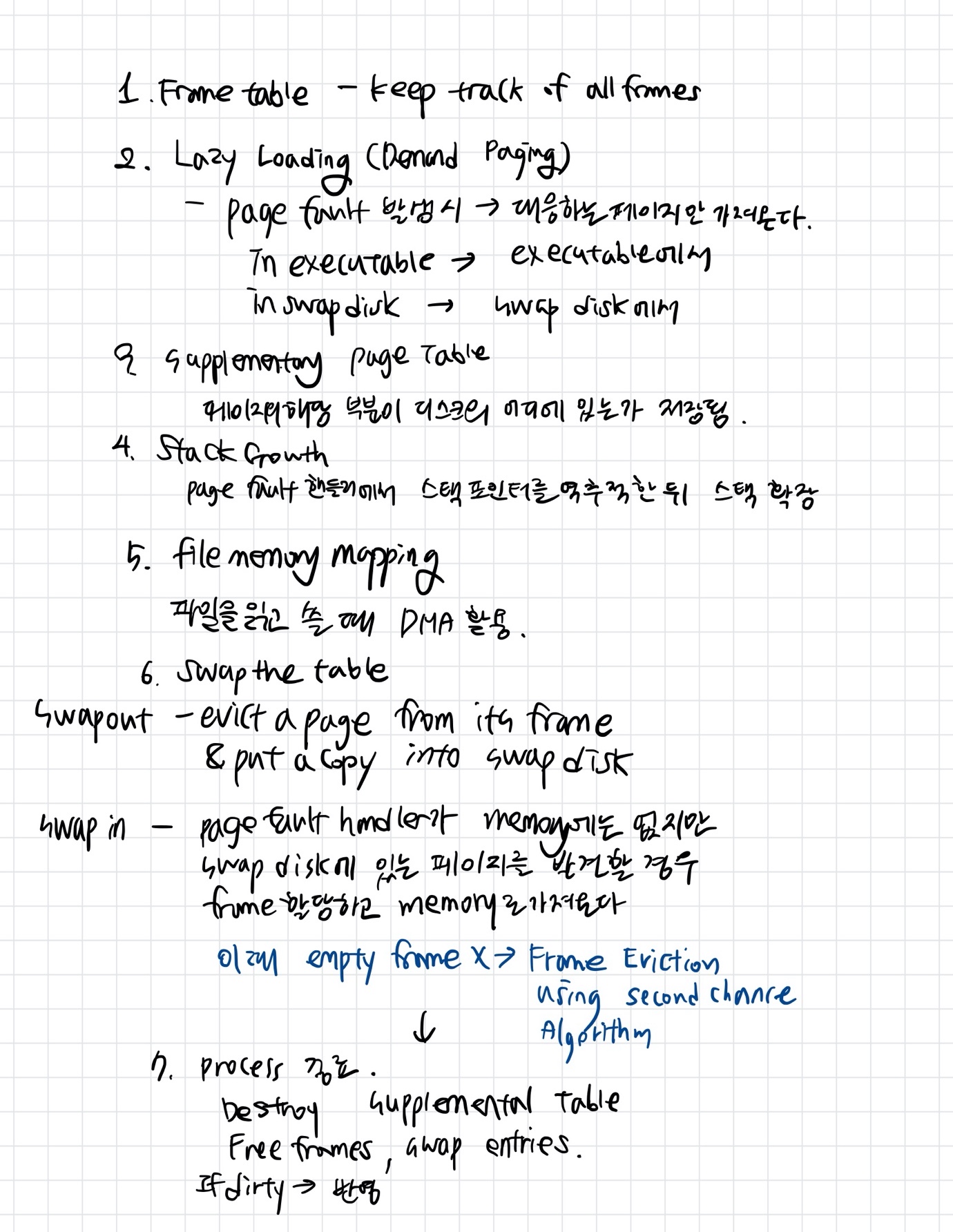
return false;

}

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성





* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
  1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술
* bool handle\_mm\_fault(struct supplemental\_page\_table \*supt, uint32\_t \*pagedir, void \*upage){
* //1. Check the validity of memory reference.
* struct supplemental\_page\_table\_entry \*supte;
* supte = vm\_supt\_look\_up(supt, upage);
* if(supte == NULL)
* return false;
* //If already loaded
* if(supte->status == ON\_FRAME)
* return true;
* //2. Obtain a frame to store the page.
* void \*frame\_page = vm\_frame\_allocate(PAL\_USER, upage);
* if(frame\_page == NULL){
* return false;
* }
* //3. Fetch the data into the frame.
* bool writable = true;
* switch(supte->status){
* case ALL\_ZERO:
* memset(frame\_page, 0 , PGSIZE);
* break;
* case ON\_FRAME:
* break;
* case ON\_SWAP:
* //Swap in (swap disc -> data)
* vm\_swap\_in(supte->swap\_index, frame\_page);
* break;
* case FROM\_FILESYS:
* if(vm\_load\_page\_from\_filesys(supte, frame\_page) == false){
* vm\_frame\_free(frame\_page);
* return false;
* }
* writable = supte->writable;
* break;
* default:
* PANIC("Exception");
* }
* //4. Find the page table entry that faults virtual address to physical page.
* if(!pagedir\_set\_page(pagedir, upage, frame\_page, writable)){
* vm\_frame\_free(frame\_page);
* return false;
* }
* supte->kpage = frame\_page;
* supte->status = ON\_FRAME;
* pagedir\_set\_dirty(pagedir, frame\_page, false);
* vm\_frame\_unpin(frame\_page);
* return true;
* }

memory에 로드되지 않은 페이지를 접근한 경우인데 memory에 접근하면 page fault가 발생한다. 그런데 이는 잘못된 메모리 영역 접근이 아니라 swapped out이나 lazy loaded로 판단하여 page를 disk에서 가져와서 해결하는 함수이다.

순서는 다음과 같다.

1. 엔트리를 확인하여 만약 잘못된 메모리 접근이면 false를 리턴하고 exception.c의 page fault함수에서 시스템 종료된다. 정상적인 메모리 접근은 다음으로 진행.

2. 페이지의 내용을 저장할 frame page를 찾는데, 이 과정에서 empty fame이 없다면 second chance algorithm에 의하여 evicted frame이 발생할 수도 있다.

3. 엔트리의 status에 따라 데이터를 알맞게 채운다.

- ALL ZERO : 0으로 채운다.

- ON\_FRAME : 이미 프레임화되어있다는 것이므로 불가능한 경우이다. 그래도 혹시 모르니 넣어줌.

- ON\_SWAP : swap\_in 실행

- FROM\_FILESYS : 파일 포인터를 통해 파일을 읽고 데이터 축적.

4. user mode에서의 주소와 새로운 frame을 대응시킨다. 그리고 이는 frame 테이블에 올라온 것이므로 status를 ON\_FRAME으로 바꾼다.

2. Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술

/\* Second Chance Algorithm \*/

struct frame\_table\_entry\* next\_frame(void);

struct frame\_table\_entry\* second\_chance( uint32\_t \*pagedir ) {

size\_t n = hash\_size(&frame\_map);

if(n == 0)

PANIC("Frame table is empty.");

size\_t it;

for(it = 0; it <= n + n; ++ it) // prevent infinite loop.

{

struct frame\_table\_entry \*e = next\_frame();

// if pinned, continue

if(e->pinned) continue;

// if referenced, give a second chance.

else if( pagedir\_is\_accessed(pagedir, e->upage)) {

pagedir\_set\_accessed(pagedir, e->upage, false);

continue;

}

// victim

return e;

}

PANIC ("Can't evict any frame. \n");

}

struct frame\_table\_entry\* next\_frame(void)

{

if (list\_empty(&frame\_list))

PANIC("Frame table is empty, can't happen - there is a leak somewhere");

if (victim\_ptr == NULL || victim\_ptr == list\_end(&frame\_list))

victim\_ptr = list\_begin (&frame\_list);

else

victim\_ptr = list\_next (victim\_ptr);

struct frame\_table\_entry \*e = list\_entry(victim\_ptr, struct frame\_table\_entry, lelem);

return e;

}

- frame table을 돌면서 victim ptr을 통해 evicted될 frame을 고른다.

1) reference\_bit = 1 이면 reference bit을 0으로 설정하고 다음 entry로 넘어간다.

2) reference\_bit = 0 이면 해당 frame을 evict한다.

- 해당 frame의 page mapping을 제거하고 swap out한다.

- 또한 frame table에서 아예 제거한다. (이 때 자료구조는 linked list임을 명심하였다.)

- 그 후 page를 재할당하여 새롭게 할당된 frame을 테이블에 넣고 매핑한다.

이 때 만약 pinned 된 frame의 경우에는 swap out 되지 않도록한다.

void

vm\_frame\_pin (void\* kpage) {

vm\_frame\_set\_pinned (kpage, true);

}

void

vm\_frame\_unpin (void\* kpage) {

vm\_frame\_set\_pinned (kpage, false);

}

static void

vm\_frame\_set\_pinned (void \*kpage, bool new\_value)

{

lock\_acquire (&frame\_lock);

// hash lookup : a temporary entry

struct frame\_table\_entry f\_tmp;

f\_tmp.kpage = kpage;

struct hash\_elem \*h = hash\_find (&frame\_map, &(f\_tmp.helem));

if (h == NULL) {

PANIC ("The frame to be pinned/unpinned does not exist");

}

struct frame\_table\_entry \*f;

f = hash\_entry(h, struct frame\_table\_entry, helem);

f->pinned = new\_value;

lock\_release (&frame\_lock);

}

User memory에 접근할 때, 커널이 가상 메모리의 paging을 처리하고 있는데 또 다른 page fault를 다루다보면 문제가 발생한다. 왜냐하면 lock을 하나만 사용하기 때문이다.

이를 해결하기 위해 frame pinning을 활용하고, pinned frame은 evicted 되지 않는다. 전반적으로 시스템 콜 수행 이전에 pin시키고, 이후에 Pin을 푸는 방식으로 진행된다.

3. Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술

bool expand\_stack (struct supplemental\_page\_table \*supt, void \*upage){

struct supplemental\_page\_table\_entry \*supte =

(struct supplemental\_page\_table\_entry \*)malloc(sizeof(struct supplemental\_page\_table\_entry));

supte -> upage = upage;

supte -> kpage = NULL;

supte -> status = ALL\_ZERO;

supte -> dirty = false;

struct hash\_elem \*prev = hash\_insert(&supt->page\_map, &supte->elem);

if(prev == NULL)

return true;

else {

PANIC ("Duplicated Supplementary Page Table Entry for zero page");

return false;

}

}

user 모드에서의 시작 주소를 가리키는 upage를 통해 스택에 추가적으로 페이지를 할당해준다. 이 때 user mode와 kernel mode를 구분하여 정상적인 page fault인지

판별 가능하다. 우선 만약 user mode라면 intr\_frame에서 바로 esp를 얻어올 수 있지만.

kernel mode에서의 경우 시스템 콜 중간에 page fault가 일어날 수 있음을 감안하여 current\_esp라는 멤버를 통해 알아낼 수 있다.

struct thread

{

/\* Owned by thread.c. \*/

tid\_t tid; /\* Thread identifier. \*/

enum thread\_status status; /\* Thread state. \*/

char name[16]; /\* Name (for debugging purposes). \*/

uint8\_t \*stack; /\* Saved stack pointer. \*/

int priority; /\* Priority. \*/

//int original\_priority;

struct list\_elem allelem; /\* List element for all threads list. \*/

struct list\_elem waitelem; /\* List element stored in the wait queue \*/

int64\_t sleep\_endtick;

uint8\_t \*current\_esp;

A page fault might occur in the kernel, so we might need to store esp. \*/

#endif

이렇게 각각의 경우에 알맞게 esp를 알아낸 후, 새로운 페이지를 할당하기 이전에

1) 스택 프레임 전반에 해당하는지 판별

2) User memory의 스택 영역인지 판별

이 두 가지 조건을 통과하여야만 할당하여 스택이 증가한다.

void\* esp = user ? f->esp : curr->current\_esp;

// Stack Growth

bool is\_user\_stack, avaialbe\_push;

is\_user\_stack = (PHYS\_BASE - MAX\_STACK\_SIZE <= fault\_addr && fault\_addr < PHYS\_BASE);

//printf("vm2\n");

avaialbe\_push = (esp <= fault\_addr || fault\_addr == f->esp - 4 || fault\_addr == f->esp - 32);

if (is\_user\_stack && avaialbe\_push) {

// we need to add new page entry in the SUPT, if there was no page entry in the SUPT.

// A promising choice is assign a new zero-page.

//printf("vm3\n");

if (vm\_supt\_has\_entry(curr->supt, fault\_page) == false)

expand\_stack(curr->supt, fault\_page);

}

if(! handle\_mm\_fault(curr->supt, curr->pagedir, fault\_page) ) {

//printf("vm4\n");

goto PAGE\_FAULT\_VIOLATED\_ACCESS;

}

// success

return;

1) 의 경우에는 is\_stack\_addr로 표현되었다. 페이지 폴트가 일어난 영역이 유저 스택으로 할당된 부분인지를 확인한다. 즉 PHYS\_BASE부터 최대 스택 크기인 8MB를 고려한다.

2) 의 경우에는 PUSH/PUSHA instruction이 미리 가능한지 예측한다. 그래서push의 양에 따라 모든 경우를 고려해준다. push의 경우에는 4MB, pusha의 경우에는 32MB까지 사용가능하면 이를 통과시켜준다.

* 1. **시험 및 평가 내용**
* (채점 대상 테스트 케이스에 해당하는) make check 수행 결과를 캡처하여 첨부
* 