+--------------------------+
| CS 330 |
| PROJECT 3: VIRTUAL MEMORY |
| DESIGN DOCUMENT |
+--------------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

20170773 김건우<kakaloto85@kaist.ac.kr> 55%
20180350 신정윤<jugipalace@kaist.ac.kr> 45%

"토큰 1개 사용합니다!"

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.

PAGE TABLE MANAGEMENT

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

vm/page.c, vm/page.h, vm/frame.c, vm/frame.c 추가.
vm/frame.h

```
struct lock frame_table_lock;
static struct list frame_table;
static struct list_elem* next_fte_elem;
struct fte {
 void *frame;
 struct spte *spte;
 struct thread *thread;
 struct list_elem elem;
};
```

우리는 struct list frame_table 을 만들어 Physical Memory를 관리하는 table을 만들었고, 각 frame 은 frame table entry (fte)로 일대일 대응을 시켜준다.

struct fte 는 frame의 주소, spte(supplementary page table entry 이후 설명) frame과 연결된 process(thread), 그리고 frame table에 넣어주기 위해 list_elem elem을 갖고 있다.

그 외, struct lock frame_table_lock 을 이용하여, frame_alloc, free_frame에서의 race condition을 회피하였다.

vm/page.h

```
enum status {
  SWAP_DISK, MEMORY, EXEC_FILE, MM_FILE
};
struct spte {
    struct hash_elem elem;
    struct list_elem mmf_elem;
    enum status state; // SWAP_DISK, MEMORY
    void *upage; // virtual address of an user page
    bool writable; // check whether the frame is writable or not
    block_sector_t swap_location; //index for swap in/out
    //for lazy loading
    struct file *file;
    struct mmap_file *mmap_file;
    int32_t offset;
    uint32_t read_bytes;
    uint32_t zero_bytes;
    //
};
```

우리는 user virtual memory page (이하 page) 를 관리하여, swap_in / out 의 수행을 비롯하여 mmap file까지, virtual memory를 용이하게 관리하기 위해 모든 process에 추가적으로 supplementary page table을 struct hash의 형태로 만들고, 각 page를 spte에 대응시켜 mapping 하고자 하였다.

status는 user virtual memory page의 정보가 저장된 장소를 나타낸다.
struct spte는 hash로 만들어진 spt 안에 있는 원소로, supplementary page table entry의 의미이다. 가장 기본적으로, user virtual memory page → upage를 포함하고 있다.
또한 이후 설명할 mmap 에서 각 page단위를 연결하기 위해, mm_f elem 이라는 list_elem 을 필드로 갖고 있다.
page를 file, 혹은 frame 과 mapping 할 때, writable 여부를 전달해준다.swap_location은 이후, 설명할 swap_disk에서 upage의 정보가 기록될 때, 기록이 시작되는 위치를 알려준다.
struct file *file 그리고 struct mmap_file *mmap_file 은 lazy loading에서, file을 physical memory에 load 할 때, 현재 spte에 기록된 정보가 어떠한 file의 정보였는지를 저장해놓기 위한 용도로 쓰였다.
offset , read_bytes, zero_bytes 는 file 혹은 mmap_file에서 어떤 구역을 읽어야하고, 0으로 채워야하는지를 알려주기 위한 지표로, 이후 unmap을 하거나, load를 할 때 이용된다.

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for locating the frame,
>> if any, that contains the data of a given page.

우리 코드는 kpage(frame) -> fte -> spte -> upage 의 구조로 연결되어 있고,
fte와 spte는 각각 struct list frame_table과 struct hash spt로 연결되어 있다.

kpage(frame)을 할당 받을 때, 기존의 코드는 load_segment에서 palloc_get_page()만을 이용하였다. 하지만 이 경우, 메모리와 디스크간의 swap을 구현할 수 없어, 메모리 효율이 떨어진다는 문제점이 있다.
따라서 우리는, create_spte를 통해 해당 upage에 대한 spte를 할당 받고 frame_alloc을 통해 fte를 할당받아, 해당 fte에 kpage와 upage(spte)에 대한 정보를 저장할 수 있게 하였다.

vm/frame.c - frame_alloc

```
void*
frame_alloc (enum palloc_flags flags,struct spte *spte){
  if ((flags & PAL_USER) == 0 )
    return NULL;
    lock_acquire(&frame_table_lock);

  void *frame = palloc_get_page(flags);
      lock_release(&frame_table_lock);

  if (!frame){
    lock_acquire(&frame_table_lock);
    struct fte *fte = find_victim();
    frame = fte->frame;
    swap_out(fte);
    lock_release(&frame_table_lock);

    // swap_out된 frame의 spte, pte 업데이트
    frame_table_update(fte, spte, thread_current());
    }
  else {
    create_fte(frame,spte);
  }
  return frame;
}
```

frame_alloc 에서는 두 가지 인자 flags, spte를 받는다. flags는 physical memory에서 어느 pool인지를 알려주는 인자이고, spte는 frame과 mapping될 upage를 알려주는 struct spte* 인자다.

frame은 우선, palloc_get_page를 통해, physical memory에 PGSIZE 만큼 빈 영역이 있는지를 알려준다. 만약, 빈 frame영역을 찾으면, 그 frame 영역을 담당할 fte를 create_fte를 통해 새로만들어주고, 인자로 받은 spte와 연결한다.
이때, 만약 메모리에 공간이 없다면 find_victim 함수를 통해 swap_disk에 쓰일 frame 영역을 찾고, swap_out 함수를 통해, victim frame의 정보를 swap_disk에 기록한다. 이후, victim frame과 연결되어있던 spte를 frame_table_update를 통해 인자로 받은 spte로 교체해준다.

이때, frame과 spte에 mapping 및 data loading이 완료되면, spte->status 를 MEMORY 로 지정해준다.

vm/page.c → create_spte

```
struct spte*
create_spte(void * upage, enum status state) {
    //언제 free해줘야하나
    struct spte *spte = (struct spte*)malloc(sizeof(struct spte));
    spte->upage=pg_round_down(upage);
    spte->state=state;
    spte->swap_location = -1;
    spte->writable=false;
    return spte;
}
```

vm/page.c → create_spte_from_exec

```
bool
create_spte_from_exec(struct file *file, int32_t ofs, uint8_t *upage,
uint32_t read_bytes, uint32_t zero_bytes,bool writable)
{
  struct spte *spte = malloc(sizeof(struct spte));
  if (!spte)
    return false;

  spte->upage = pg_round_down(upage);
  spte->state = EXEC_FILE;

  spte->file = file;
  spte->offset = ofs;
  spte->read_bytes = read_bytes;
  spte->zero_bytes = zero_bytes;
  spte->writable = writable;
  return (hash_insert(&thread_current()->spt, &spte->elem) == NULL);
}
```

이후, 설명할 exec file에서 미리 spte를 만드는 경우이다.

>> A3: How does your code coordinate accessed and dirty bits between
>> kernel and user virtual addresses that alias a single frame, or
>> alternatively how do you avoid the issue?

우리는 user virtual address(Upage)를 통해서만 frame에 접근 할 수 있게 만들어서 해당 문제를 해결했다.

---- SYNCHRONIZATION ----

>> A4: When two user processes both need a new frame at the same time,
>> how are races avoided?

모든 프로세스는 frame을 요청할 때, frame_alloc을 이용해서만 요청하게끔 디자인하였고, frame_alloc 함수에는 frame_table_lock 이라는 lock을 acquire / release 하여, 한 프로세스에서 frame을 요청이 끝날 때 lock_release(&frame_table_lock)을 하여, 다른 process에서 같은 frame을 요청하더라도 race condition이 발생하지 않도록하였다. 또한, swap이 필요하여 victim frame이 필요할 때도, find_victim 전후로 lock을 잠갔다가, 풀어주어 race condition을 회피하였다.

---- RATIONALE ----

>> A5: Why did you choose the data structure(s) that you did for
>> representing virtual-to-physical mappings?

virtual-to-physical mappings을 위해서는 여러 구현 방법이 있을 것이고, 대표적으로는 한 data structure(ex)vm-entry)에다가 frame(kpage)과 page(upage)의 모든 정보를 저장하는 경우와, 우리의 구현방식처럼 frame과 page의 각각의 entry를 만드는 경우가 있을것이다. 우리의 방식 처럼 kpage(frame) -> fte -> spte -> upage 로 mapping을 구현하는 것이, 문제가 발생했을 때 frame table에서 문제가 발생한 것인지 supplementary page table에서 문제가 발생했을 때 구분해서 해결할 수 있다. 또한 각각에 lock을 걸어서 synchronization를 좀 더 섬세하게 할 수 있다는 장점도 있다.

PAGING TO AND FROM DISK

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

우리는 vm/swap.c vm/swap.h 를 추가하여 disk 에 대한 paging을 수행하고자 하였다. swap 과정을 위해 추가한 struct 는 아래와 같다.
static struct block *swap_block;
먼저, 우리는 devices/block.c 와 block.h 에 정의되어 있는 block structure를 이용하였다. block은 disk에 접근하기 위한 interface로 본 프로젝트에서 우리는 BLOCK_SWAP block을 이용하여 block의 각 sector에 page의 정보를 쓰는 과정을 진행하였다.

static struct bitmap *swap_table;
swap_table 은 각 page가 block에서 기록되어있는 location을 표시하여, 기록할 수 있는 location을 알려주는 역할을 하는 중간체 역할의 structure이다. location 표시를 위해 쉽게 하기 위해 bitmap structure를 사용하였다.

struct lock swap_lock;
swap_block에서 page 내용을 읽거나, 쓸 때 다른 process로 인한 변화가 없도록 lock을 걸어 synchronization을 수행하고자 추가한 struct이다.

---- ALGORITHMS ----

>> B2: When a frame is required but none is free, some frame must be
>> evicted. Describe your code for choosing a frame to evict.

우리는 find_victim() 함수를 이용해서, evict해야할 frame (victim frame)을 고른다. 우리는 find_victim 에서는 frame table의 elem을 이동하면서, 조건을 충족시키는 fte를 찾아낸다. 해당 함수를 위해, 우리는 static struct list_elem* next_fte_elem 을 추가하였는데, 이는 frame_table에서 fte의 위치를 나타내는 커서 역할을 한다.

```
struct fte*
find_victim (void){
  struct fte* evict;
  struct list_elem* e;
  if (next_fte_elem == NULL)
    e = list_begin(&frame_table);
  else
    e = next_fte_elem;
  while (true) {

    evict = list_entry(e,struct fte, elem);
    struct spte* evict_spte = evict->spte;
    if (!pagedir_is_accessed evict->thread->pagedir,
evict->spte->upage)){

      if (list_next(e) == list_tail(&frame_table))
        next_fte_elem = list_begin(&frame_table);
      else
        next_fte_elem = list_next(e);
      break;
    }
    else {
      pagedir_set_accessed (evict->thread->pagedir,
evict->spte->upage,0);
      continue;
```

```
    }
    if (list_next(e) == list_tail(&frame_table))
      e = list_begin(&frame_table);
    else
      e = list_next(e);
  }
  return evict;
}
```

처음 find_victim을 시작할 때는 그 초기값이 NULL로 되어있기 때문에, frame_table 의 시작값을 커서로 두고, 그 외의 상황에서는 next_fte_elem으로부터 시작하여 while 문을 돌며, 조건을 만족하는 frame을 찾는다.

우리는, LRU 알고리즘 중에서도 second chance 알고리즘을 구현하고자 하였다. 그래서 mapping 되어있는 thread의 pagedir의 user virtual address를 확인하여, accessed bit을 확인한다. 이 accessed bit은 access 되어 있지 않으면(accessed bit가 0일때), 현재 커서 next_fte_elem이 위치한 frame을 victim frame으로 하고, break 해준다. 만약 accessed 되어 있으면, accessed_bit을 0으로 바꿔주고 넘어가서, 다음 elem을 확인한다. 이때 list의 끝에 도달하면, next_fte_elem 을 list_begin(&frame_table)로 지정하여, list을 순환하며 확인할 수 있도록 하였다.

>> B3: When a process P obtains a frame that was previously used by a
>> process Q, how do you adjust the page table (and any other data
>> structures) to reflect the frame Q no longer has?

각 Process가 frame_alloc을 호출해야만 frame을 얻을 수 있다. 여기서, 다른 Process가 사용하고 있던 frame을 사용하는 경우는 find_victim을 통해 다른 process가 소유하던 frame을 얻었을 때 말고는 뿐이다. 이때, 우리는 기존의 frame(process Q의 frame)을 swap disk로 swap_out해주고 frame_table_update(fte, spte, thread_current()) 를 통해, 새로운 process(Q)의 spte와 fte를 해당 frame에 mapping을 시켜줌으로써 B3의 문제를 해결하였다.

>> B4: Explain your heuristic for deciding whether a page fault for an
>> invalid virtual address should cause the stack to be extended into
>> the page that faulted.

우리는 stack_growth 함수에 else if (fault_addr >= f->esp - 32) 를 추가하였다.
그 이유는 PUSHA가 발생했을 때, 32byte 만큼 push 하기 때문에, fault_addr는 최대 user program의 스택 포인터 아래 32 byte에서 발생할 수 있다. 그렇기 때문에 우리는 heuristic을 32로 정했다.

---- SYNCHRONIZATION ----

>> B5: Explain the basics of your VM synchronization design. In
>> particular, explain how it prevents deadlock. (Refer to the
>> textbook for an explanation of the necessary conditions for
>> deadlock.)

우리의 vm 구현에서는 fte를 위한 frame_table_lock, swap을 위한 swap_lock, file을 위한 file_lock 이 있다. 우리는 해당 lock들을 사용하는 모든 함수, load_from_exec, load_from_swap 등등에서 frame_table_lock이 acquire되고 나서 swap lock, file lock을 사용할 수 있게 하였고, swap_lock과 file_lock이 release되고 난뒤 frame_table_lock을 release하게 하였다. 그렇기 때문에, 우리의 코드 상 VM에서는 process가 순환대기를 하지 않는다. 즉, P1과 P2가 있을 때, P2이 P1를 기다리고, P1이 P2를 기다리는 상황을 만들지 않았다. 예를 들어, 한 프로세스가 frame_table_lock을 acquire 하고 있는 상황에서, 다른 process 가 file_lock 혹은 swap_lock을 hold 하고 있는 상황은 있을 수 없다. 이런 이유로, deadlock 상황을 회피할 수 있었다.

>> B6: A page fault in process P can cause another process Q's frame
>> to be evicted. How do you ensure that Q cannot access or modify
>> the page during the eviction process? How do you avoid a race
>> between P evicting Q's frame and Q faulting the page back in?

frame 에 접근하는 방법은 frame_alloc, find_victim, free_frame 총 3가지인데 3가지 함수에서 모두 frame_table_lock acquire / release를 frame 접근 전후에 걸었다. 이 경우, 문제의 해당하는 find_victim의 상황에서는 find_victim 과정에서 frame_table_lock 을 걸어, 우선 victim frame 을 찾을 때까지는 다른 process에서 frame_alloc, free_frame을 통해 접근하지 못하도록 하였다.

>> B7: Suppose a page fault in process P causes a page to be read from
>> the file system or swap. How do you ensure that a second process Q
>> cannot interfere by e.g. attempting to evict the frame while it is
>> still being read in?

우리는 load_from_exec으로 file로 부터 데이터를 읽고, load_from_swap으로 swap disk 에서 데이터를 읽었다. 이때 두 함수 모두에서, 가장 먼저 frame_alloc으로 새로운 frame을 할당받는데, 새로운 frame을 할당 받은 직후에 frame_table_lock을 acquire하고, 각각의 함수(load_from_exec,load_from_swap)가 종료될때 release함으로써 해당 문제를 해결하였다.

>> B8: Explain how you handle access to paged-out pages that occur
>> during system calls. Do you use page faults to bring in pages (as
>> in user programs), or do you have a mechanism for "locking" frames
>> into physical memory, or do you use some other design? How do you
>> gracefully handle attempted accesses to invalid virtual addresses?

우리는 page fault를 불러서 fault addr의 upage 에 해당하는 spte를 찾아서 없으면, stack_growth 에 해당하는 상황인지를 fault_addr 와 esp를 비교해서 확인하여 stack_growth를 위한 새로운 spte를 만들어준다. 만약, 해당하는 spte가 thread_current에 있다면, spte->status를 통해 page의 정보가 저장된 곳을 찾는다. 그 status에 따라서, demand loading을 어디서(swap_disk, exec_file) 에서 해야하는지를 찾아서 loading해준다.

---- RATIONALE ----

>> B9: A single lock for the whole VM system would make
>> synchronization easy, but limit parallelism. On the other hand,

>> using many locks complicates synchronization and raises the
>> possibility for deadlock but allows for high parallelism. Explain
>> where your design falls along this continuum and why you chose to
>> design it this way.
→ 우리는 많은 lock을 사용하여 parallelism이 가능한 디자인을 하였다. 여러 개의 lock을 사용하면 deadlock 이 발생할 상황은 늘지만, 전체 VM 시스템에 lock을 하나만 이용하면, 하나의 process 가 load / physical memory 접근 등 많은 과정을 전부 수행하고 나서야, 다음 process가 수행할 수 있는데, 이런 디자인의 경우에는 OS의 execution speed가 급격하게 느려지게 된다. 따라서 여러 process을 dynamic하게 수행시키기 위해 여러개의 lock을 사용하는 방법을 선택하였다.

MEMORY MAPPED FILES

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

```
struct spte {
    ...
    struct mmap_file *mmap_file;
    ...
    //
};
```

우리는 memory mapped file을 위해spte에 한 가지 새로운 field를 추가하였다. struct mmap_file *mmap_file 은 lazy loading에서, file을 physical memory에 load 할 때, 현재 spte에 기록된 정보가 어떠한 file의 정보였는지를 저장해놓기 위한 용도로 쓰였다.

threads/ thread.h

```
...
struct list mmap_list;
...
```

thread 에도 새로운 field를 추가하였는데, mmap_list는 process가 갖고 있는 memory mapped file 이 여러개 일때, 이를 모아놓은 list 구조다.

```
struct mmap_file {
  int map_id;
  struct file* file;
  struct list_elem elem;
  struct list spte_list;
};
```

mmap_file은 file의 memory mapping management를 위해 새롭게 만든 struct 다. map_id는 fd를 대체하여, mmap_file 를 구분하는 역할을 한다. 이때, 우리는 map_id 값을 fd 값을 넣어서

함께 이용한다. struct file* file은 mapping 할 file이다. struct list_elem은 앞서 언급한 thread(process)의 mmap_list에 넣기 위한 요소이며, struct list spte_list는 해당 file 과 mapping 된 user_virtual_memory의 page들을 모아 놓기 위해, spte를 이용하여 list 형태로 저장하였다.

---- ALGORITHMS ----

>> C2: Describe how memory mapped files integrate into your virtual
>> memory subsystem. Explain how the page fault and eviction
>> processes differ between swap pages and other pages.

→ load_from_exec 에서 MM_FILE 부분
→ mmap // munmap signal
우리는 mmap syscall을 아래 코드와 같이 짰다.

```
int mmap (int fd, void* upage) {
  if(get_spte(&thread_current()->spt,upage))
    return -1;
  if(pg_round_down(upage)!=upage)
    return -1;
  if(!is_user_vaddr(upage)||upage<0x08048000)
    return -1;
  ASSERT(fd!=0&&fd!=1);
  if (fd>130)
    exit(-1);
  struct thread *cur =thread_current();
  struct file* file = cur->fd_table[fd];
  int filesize=file_length(file);
  int read_bytes=filesize;
  if(filesize<1)
    return -1;
  int ofs=0;
  struct mmap_file* mmap_file = (struct mmap_file*)
malloc(sizeof(struct mmap_file));
  list_init(&mmap_file->spte_list);
  list_push_back(&cur->mmap_list,&mmap_file->elem);
  mmap_file->map_id=fd;
  while (ofs<filesize)
    {
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes :
PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      if(!get_spte(&cur->spt,upage)){
```

```
        bool result=  create_spte_from_mmf(file, ofs,upage,
page_read_bytes, page_zero_bytes,true,mmap_file);
        if(!result){
          printf("spte error at load_segment");
        }
        ofs+= PGSIZE;
        /* Advance. */
        read_bytes -= page_read_bytes;
        upage += PGSIZE;
      }
      else
        return -1;
    }
  sema_down(&file_lock);
  struct file* new_file = file_reopen(file);
  mmap_file->file = new_file;
  sema_up(&file_lock);
  if(new_file==NULL){
    return -1;
  }
  return fd;
}
```

mmap_file은 file의 size 만큼 page 들을 할당하여 create_spte_from_mmf로 필요한만큼 spte를 만들어, mmap_file->spte_list에 spte들을 저장한다. 이후, file_reopen(file)을 통해, original file의 복사본을 만든다. 이후, munmap 할 때, process에 의해서 write 된 page들은 disk에 기록된 file에 적어주고, close해준다.

우리는 mmap된 file의 spte에는 MM_FILE이라는 flag를 달아주었다. 이를 이용하여 lazy loading을 위해 page fault가 일어나고 load_from_exec함수가 수행될 때, spte의 file이 아닌 spte의 mmap_file struct에 저장된 file을 통해 loading을 진행하였다.

```
bool load_from_exec (struct spte *spte)
{

   uint8_t *frame = frame_alloc(PAL_USER, spte);
   if (!frame) {
     return false;
   }

   if (spte->state==MM_FILE){
     if (spte->read_bytes > 0){
       sema_down(&file_lock);
       if ((int) spte->read_bytes != file_read_at(spte->mmap_file->file, frame,
spte->read_bytes, spte->offset)){
         sema_up(&file_lock);
```

```
        free_frame(frame);
        return false;
      }
      sema_up(&file_lock);
      memset(frame + spte->read_bytes, 0, spte->zero_bytes);
    }
    else{
      memset (frame, 0, PGSIZE);
    }

  }
  else{
    if (spte->read_bytes > 0){
      sema_down(&file_lock);
      if ((int) spte->read_bytes != file_read_at(spte->file, frame,
spte->read_bytes, spte->offset)){
        sema_up(&file_lock);
        free_frame(frame);
        return false;
      }
      sema_up(&file_lock);
      memset(frame + spte->read_bytes, 0, spte->zero_bytes);
    }
    else{
      memset (frame, 0, PGSIZE);
    }
  }
  if (!install_page(spte->upage, frame, spte->writable)) {
      free_frame(frame);
      return false;
  }
  spte->state = MEMORY;
  return true;
}
```

>> C3: Explain how you determine whether a new file mapping overlaps
>> any existing segment.

→ mmap(int fd, void *upage) syscall에서 upage 영역에 할당된 spte가 있는지를 get_spte 함수를 통해 확인한다. 이때 이미 할당된 spte가 있다면, 이는 load_segment, 혹은 setup_stack을 통해 만들어진 것을 의미하므로, return -1을 하며, mmap syscall을 종료한다. 만약, upage 영역에 spte가 없다면 mmap 과정을 진행하며 해당 영역에 새로운 spte를 이후, create_spte_from_mmf을 통해 만들어준다.

---- RATIONALE ----

>> C4: Mappings created with "mmap" have similar semantics to those of
>> data demand-paged from executables, except that "mmap" mappings are
>> written back to their original files, not to swap. This implies
>> that much of their implementation can be shared. Explain why your
>> implementation either does or does not share much of the code for
>> the two situations.

질문에서 알 수 있듯이, data demand-paged from executables와 Mappings created with "mmap" 은 비슷한 원리를 가지고 있기 때문에, 우리의 핀토스 구현에서는 demand loading을 위한 load_from_exec을 통해 두 기능을 **함께 구현**하였다. C2의 답변에서 알 수 있듯이, spte의 flag로 MM_FILE, EXEC_FILE flag를 달아서 MM_FILE일 경우 Mappings created with "mmap", EXEC_FILE일 경우 data demand-paged from executables를 구현하도록 했다.

SURVEY QUESTIONS

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?