Operating Systems – Project 3: Virtual Memory

Final Report

Khau Lien Kiet
Team 10

2022/11/30

Department of Computer Science & Engineering
Pohang University of Science and Technology (POSTECH)

Table of Contents

I. Frame Table

a. Overall requirement/scheme

In the current PINTOS, there is no physical frame representation and thus it is required to implement a frame table to manage the physical frames effectively. With the view of frame table, there would contain frame table entries, which consists of a content of virtual address space and a pointer pointing toward each page (*spt_entry* type – ***detailed explanation later***). Indeed, if there is no free space in the frame table, it is required to implement a page replacement policy to select a page for evicting. Hence, the more explanation would be shown afterwards.
Indeed, it is necessary to create two files under the *vm* directory, which is called *"frame.c"* and *"frame.h"* to create/initialize/input a required data structure and functions inside it. Overall, the following procedure would initialize a data structure, create some functions to allocate and deallocate the frame, some functions to manipulate a lock for fulfilling the eviction policy, and initiate the idea of choosing a victim from corresponding frames while there is no free space ***(described more detailed in Swap Table section).***

b. Data Structure

- **Add vm/frame.h/struct frame{}:** Initialize the data structure as frame table entry in the frame table, which contains the *base* as the Kernel virtual address, **page* as the *spt_entry* type pointing to the corresponding page, and *lock* as the frame lock.

```
static struct frame
{
   void *base;              /* Kernel virtual address*/
   struct spt_entry *page;    /* Corresponding to page*/
   struct lock lock;         /* Frame lock*/
};
```

- **Add vm/frame.c/frame_table:** This data structure is itself a frame table where consists of *frame* data structure, which is represented as the frame table entry.

```
static struct frame *frame_table;
```

- **Add vm/frame.h/scan_lock:** This synchronization primitive is used for critical section in order to prevent multiple processes get accessed to the *frame_table*.

```
static struct lock scan_lock;
```

c. Functions & Algorithms

- **Initialization of frame_table and frames**
    - Add vm/frame.c/frame_init(): This function is to initialize various data structure for managing the frame table. For instance, it is required to initialize the *scan_lock* as the global lock to let one process access the frame table once at a time, initialize the *frame_table* within the *malloc()* function. Indeed, the

frame_table has a type of *frame* data structure. Therefore, we must allocate the frame as the entry inside the *frame_table* tracked by the counter, initialize the frame lock for the usage of selecting frames to evict ***(further explanation below)***, and assign the kernel virtual address to its frame.

```
void frame_init (void)
{
  void *base;
  lock_init (&scan_lock);

  frame_table = malloc (sizeof *frame_table * init_ram_pages);
  if (frame_table == NULL)
    PANIC ("out of memory allocating page frame_table");

  while ((base = palloc_get_page (PAL_USER)) != NULL)
    {
      struct frame *f = &frame_table[frame_cnt++];
      lock_init (&f->lock);
      f->base = base;
      f->page = NULL;
    }
}
```

- Modify threads/init.c/main(): Inside the function, we would add the function *frame_init()* in order to initialize the frame when the system is executed.

```
int
main (void)
{
  /…
  /* Project 3 - Frame Table: Initialize the Frame Table*/
  frame_init();
  /..
}
```

- **Frame Allocation**
  - Add vm/frame.c/try_frame_alloc_and_lock(): The main functionality of this function is to allocate a new frame a lock a frame for a page as *spt_entry* ***(explanation below).*** The procedure to allocate a frame is: Firstly, we prevent the

race condition by the usage of *scan_lock*. Then, we iterate to find the empty space inside the *frame_table*. If the *frame_table* has the empty frame to use for allocation, it would assign to the page and frame. Afterwards, it would release the *scan_lock* to provoke other process get executed.

```
struct frame *try_frame_alloc_and_lock (struct spt_entry *page) {
  int i = 0;
  lock_acquire(&scan_lock); //Prevent multiple process processes to access the frame table

  /* Iterate to find empty frame_table */
  for (i = 0; i < frame_cnt; i++)
  {
   struct frame *f = &frame_table[i];
   if (!lock_try_acquire(&f->lock))
     continue;

   if (f->page != NULL) {
     lock_release(&f->lock);
     continue;
   }
   // That empty frame would be used to allocate
   f->page = page;
   page->frame = f;
   lock_release(&scan_lock);
   return f;
  }

 lock_release(&scan_lock);
 return NULL;
}
```

- Add vm/frame.c/frame_alloc_and_lock(): This function has the same mechanism as *try_frame_alloc_and_lock()* but it yields that process a second trial to iterate through the *frame_table*. If there is no free frame in the *frame_table*, it would imply the procedure to select frames to evict, which is described clearly in the ***Swap Table section***.

```
struct frame *frame_alloc_and_lock (struct spt_entry *page) {
```

```
  struct frame *f;
  int i = 0;
  for (i = 0; i < 2; i++)
  {
    f = try_frame_alloc_and_lock(page);
    if (f != NULL)
      return f;
  }
  return NULL;
}
```

- **Frame Deallocation**
  - Add vm/frame.c/frame_deallocate(): This function is used to deallocate the corresponding the frame by releasing frame for the usage of another page from the *frame_table*. Within yielding to other pages, all the content of frame would be lost.

```
void frame_deallocate (struct frame *f) {
  if (!lock_held_by_current_thread(&f->lock))
     sys_exit(-1);
  f->page = NULL;
  lock_release(&f->lock);
  return;
}
```

- **Search frames to evict:** To fulfill the eviction policy in the later part, we utilize the *lock* variable inside the frame entry. It would determine which frame gets evicted based on the lock. Therefore, it is necessary to initialize two functions of *frame_lock()* and *frame_unlock()* for a further usage in the Swap Table section.
  - Add vm/frame.c/frame_lock(): If the frame is resided in the *frame_table*, it is necessary to lock the frame itself the *frame_table*. Thus, its purpose is to indicate the frame is holding the frame lock and that frame lives inside the *frame_table*. Until the page gets unlocked, the frame would be unlocked afterwards.

```
void frame_lock (struct spt_entry *p) {
  struct frame *f = p->frame;
  if (f!= NULL){
    ASSERT (!lock_held_by_current_thread (&f->lock));
    lock_acquire(&f->lock);
    if (f != p->frame)
```

```
    lock_release(&f->lock);
  }
  return;
}
```

- Add vm/frame.c/frame_unlock(): Whenever the function is provoked, it would unlock the frame, thus allowing this frame to be evicted and proceeding the eviction policy afterwards.

```
void frame_unlock (struct frame *f) {
  if (!lock_held_by_current_thread(&f->lock))
    sys_exit(-1);
  lock_release(&f->lock);
  return;
}
```

- **Setup Stack**
  - Modify userprog/process.c/setup_stack(): From the previous implementation, pages were directly allocated using the *palloc_get_page()*. However, since the memory is now managed by the physical frames that are represented by the *frame_table*, it is necessary to allocate the frames by *frame_alloc_and_lock()* and set up the stack areas which is implemented in the project 2. In addition, while finished setting up the stack and push all the argument in the user stack, it is necessary to provoke the *frame_unlock()* to let other process execute.

```
static bool
setup_stack (void **esp, char* cmdline)
{
  /…
    p->frame = frame_alloc_and_lock (p);
    kpage = p->frame->base;
    if (kpage != NULL)
      {

        /…
        frame_unlock (p->frame);
        return true;
      }
  }
```

```
    return false;
}
```

- **Eviction policy:** As mentioned previously, this section is strictly correlated to the later section to fulfill the eviction policy in that part. Therefore, it is required to initiate the function *evict_page()* to fulfill its policy. Thus, the detailed description would be explained in a ***Swap Table section***.

```
bool evict_page (struct spt_entry *p)
```

II. Lazy Loading

a. Overall requirement/scheme

Lazy Loading is a method that only a stack setup part is loaded during loading for memory allocation when a process starts. This means that the there are some critical parts of the process would be loaded in the memory to save the memory space. This is the procedure of page fault handler to handle the lazy loading instead. After handling the page fault, it needs to resume the process operation.
Indeed, it is required to create other two files in *vm* directory, which are "*page.c"* and *"page.h"*. Overall, the procedure to implement Lazy Loading is: Firstly, it is required to initialize a new data structure as a page, which is called the supplemental page table entry (*spt_entry*). Then, it is necessary to modify for the *load_segment()* in order to notice there are some critical information loaded in the memory. Finally, with the usage of page fault to implement the Lazy Loading, it is required to create some functions to fulfill loading a page to let the page fault handler solving this issue.

b. Data Structure

- **Add vm/page.h/struct spt_entry{}:** Under the structure of Supplemental Page Table entry, there would contain essential information for the page loading when the page faults are occurred. This data structure is the entry for the Supplemental Page Table, which variables are explained in the next section.

```
static struct spt_entry{};
```

c. Functions & Algorithms

- **Segment loading**
    - Modify userprog/process.c/load_segment(): In the previous implementation, all the page contents have been loaded into the memory which results in the waste of the resource. Therefore, it is implemented to load the required content in the *spt_entry*. Indeed, it would get a page of memory from the *page_allocate()* ***(detailed explanation in next section)*** but it is not allocated frames for the page instantly. However, through the page fault handler, it would take care of loading the remain content in the *page_fault()*.

```
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
```

```
            uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
    /…
    struct spt_entry * p = page_allocate(upage, !writable);
    /..
}
```

- **Page fault handler:** Under the page fault handler responsibility, it is required to implement Lazy Loading through *page_fault()* function.
    - Modify the userprog/exception.c/page_fault(): Within some various reasons for page fault, as the page fault could occur when referred to a page whose loading into memory has postponed, lazy loading could be implied and thus the process of page fault handler would be provoked through *load_page()* function, which mechanism is described afterwards.

```
static void
page_fault (struct intr_frame *f)
{
 /..
 if(not_present && user){
   if (!load_page(fault_addr))
       sys_exit(-1);
   return;
 }
}
```

- **Page loading:** When the page fault is occurred, the page fault handler provokes this function to handle. This means that this function is called from *page_fault()* and thus it required to execute the lazy loading section with the correspondence to the page type ***(as described in the design report – PAGE_FILE, PAGE_SWAP, PAGE_ZERO)***
    - Add vm/page.c/load_page(void *fault_addr): The argument passing in the function is the fault_addr. Firstly, it would look up the page within the given *fault_addr*. If there is none, it would terminate. Then, with the function *frame_lock()*, it would allocate the frame. If there is no frame, it would perform *do_page_in()* to swap in those page in the frame by checking the corresponding entry related to the page type (detailed explanation below). From *pagedir_set_page()*, it would know the process of memory allocation is successful or not. This function would return that boolean value related to the memory allocation. Afterwards, it is required to deallocate the frame.

```
bool load_page (void *fault_addr) {
```

```
  struct thread * t = thread_current();
  /…
  struct spt_entry* p = page_lookup(fault_addr); //Lookup a page with the given fault address
  if (p == NULL)
    return false;

  frame_lock(p); //Allocate the frame
  // If there is no frame, swap in.
  if (p->frame == NULL){
    // Checking the corresponding entry related to which type of PAGE
    if (!do_page_in(p))
      return false;
  }
  bool flag = pagedir_set_page(t->pagedir, p->addr, p->frame->base, !p->read_only);
  frame_unlock(p->frame); //Deallocate the frame
  return flag;
}
```

- Add vm/page.c/do_page_in(struct spt_entry *p): This sided function for *load_page()* would lock a frame for a page depending on the page type. Within the *spt_entry*, it would reference to the frame and perform the loading into the frame, particularly into the memory. If the corresponding entry is PAGE_SWAP by checking the sector, it would perform *swap_in()* ***(detailed explained in Swap Table)***. If the corresponding entry is *PAGE_FILE* when checking its *file* in the *spt_entry*, it would read the file and load it into the physical page. In addition, if the corresponding entry is *PAGE_ZERO*, it would refer to the physical address and set the page to 0. Based on the page type, it would notice exactly what to perform on the lazy loading section while the process is invoked by the page fault handler.

```
static bool do_page_in (struct spt_entry *p) {
  p->frame = frame_alloc_and_lock(p);
  if (p->frame == NULL)
    return false;
  /…
  //PAGE_SWAP
  if (p->sector != (block_sector_t)(-1)) //Ensure it has the swap slot
  {
    if (!swap_in(p))
```

```
    return false;
  }
  //PAGE_FILE
  else if (p->file != NULL){
    off_t read_amt = file_read_at(p->file,p->frame->base,p->file_size,p->file_ofs);
    off_t left_amt = PGSIZE - read_amt;
    memset(p->frame->base + read_amt, 0, left_amt);
  }
  else
    // PAGE_ZERO
    memset (p->frame->base,0,PGSIZE);
  return true;
}
```

III. Supplemental Page Table

a. Overall requirement/scheme

Within the Supplemental Page Table, it would contain more critical information than original page table. The main functionality is to decide to load the corresponding file in the accurate area inside the page table. If the corresponding entry is file-backed pages (*PAGE_ZERO* or *PAGE_FILE*), the supplemental page table would decide the page to load in the physical memory. In addition, if it is an anonymous page *(PAGE_SWAP*), the supplemental page table would decide to get a swap disk and its partition to store the page.
Indeed, the function would be added inside the two files *"page.c"* and *"page.h"*. Overall, to fulfill the requirement of the Supplemental Page Table (SPT), firstly, it would declare a supplemental page table and explain the parameters inside its entry. To integrate the usage of SPT, it is required to initialize the SPT, create a function to find the *spt_entry* according to its address, two main ideas for allocating SPT as well as deallocating its *spt_entry*.

b. Data Structure

- **Add vm/page.h/struct spt_entry{}:** In the structure of spt_entry{}, there are some fields such as:
    - *addr: The pointer stands for the user virtual address and it always starts with the page offset 0
    - *frame: The pointer is the *frame* data structure which contains the mapping frame
    - *thread: The pointer is the *thread* data structure which implies the current thread owns spt_entry data structure
    - read_only: The boolean parameter is to indicate read and/or write operation. If it is true, read only. If it is false, it could read and write.
    - *file: The pointer is the file data structure which implies the corresponding file to read

- file_ofs: The parameter stands for the file offset
- file_size: The parameter stands for the file size to read

```
static struct spt_entry
{
   void *addr;               /* User virtual address (always with page offset 0)*/
   struct frame *frame;      /* Mapping frame*/
   struct thread *thread;    /* Current thread owns the spt_entry*/
   bool read_only;           /* Indicator of read only or not (imply write)*/

   struct hash_elem hash_elem; /* Element field to be managed in Supplemental Page Table*/

   struct file *file;        /* Corresponding file to read*/
   off_t file_ofs;           /* File offset*/
   off_t file_size;          /* File size to read*/

   /…
};
```

- **Modify threads/thread.h/struct thread{}:** In the structure of thread{}, add the *supptable* under the hash data structure to manage the virtual pages as the virtual address space is allocated per process.

```
struct thread
 {
   /..
   struct hash *supptable;

 };
```

- **Modify threads/thread.c/init_thread():** At first, there is no content in the SPT and thus the supptable should be pointed to NULL.

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  /..
```

```
  t->supptable = NULL;
}
```

c. Functions & Algorithms

- **Initialization of Supplemental Page Table:** Since the Supplemental Page Table – *supptable* is hash structure, it is required to obtain two sided functions in order to satisfy the *hash_init()* for initializing the SPT.
    - Add vm/page.c/page_hash(): This function returns a hash value for the page that spt_entry refers to.

```
unsigned page_hash (const struct hash_elem *e, void *aux) {
  const struct spt_entry *p = hash_entry (e, struct spt_entry, hash_elem);
  return hash_bytes (&p->addr, sizeof p->addr);
}
```

    - Add vm/page.c/page_less(): This function is to compare the virtual address of two spt_entry in order to notice the order inside the SPT. If page A precedes page B in terms of the address, it would return true.

```
bool page_less (const struct hash_elem *a_, const struct hash_elem *b_, void *aux) {
  const struct spt_entry *a = hash_entry (a_, struct spt_entry, hash_elem);
  const struct spt_entry *b = hash_entry (b_, struct spt_entry, hash_elem);
  return a->addr < b->addr;
}
```

    - Modify userprog/process.c/load(): This modification is to initialize the SPT whether create a supptable using malloc() function. If there is a space for the supptable, it would provoke hash_init() by passing the supptable location as well as two sided functions created previously.

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{ /..
  t->supptable = malloc (sizeof *t->supptable);
  if (t->supptable == NULL)
    goto done;
  hash_init (t->supptable, page_hash, page_less, NULL);
  /..
}
```

- **Supplemental Page Table Entry Lookup**: While creating the SPT, there would have a function to seek for the *spt_entry* inside the SPT.
    - Add vm/page.c/page_lookup(void *addr): Within the given virtual address as the argument passing in the function, this function would return a page as *spt_entry* data structure through the *hash_entry()* function to find the correct *spt_entry*. Otherwise, if the page does not exist in the SPT, a null pointer would return. Indeed, this function would resolve the Stack Growth implementation that is explained more detailed in the next section.

```
static struct spt_entry *page_lookup (const void *address) {
  if (address >= PHYS_BASE) return NULL;
  struct spt_entry p;
  p.addr = pg_round_down(address);
  struct thread * t = thread_current();
  struct hash_elem * found = hash_find(t->supptable,&p.hash_elem);
/..
  return hash_entry(found, struct spt_entry, hash_elem);
  }
```

- **Supplemental Page Table Allocation and Address Space Initialization**
    - Add vm/page.c/page_allocate(): This function adds a mapping for the user virtual address to the supplemental page table. The main idea is to assign the required parameters for SPT, which are declared inside the *spt_entry*. While indicating essential information for those variables, it is necessary to insert the *spt_entry* into the SPT through *hash_insert()* regardless of the page type. If the virtual address is already mapped or its memory allocation fails, the process would fail and thus return NULL. Afterwards, this function would return the *spt_entry* that is inserted inside the SPT.

```
struct spt_entry * page_allocate (void *vaddr, bool read_only) {
  struct spt_entry *p = malloc (sizeof *p);
  if (p == NULL)
    return NULL;
  p->addr = pg_round_down (vaddr);
  p->frame = NULL;
  p->file = NULL;
  p->file_ofs = 0;
  p->file_size = 0;

  p->sector = (block_sector_t)(-1);
```

```
        p->swap_status = !read_only;

        p->read_only = read_only;

        struct thread * t = thread_current();
        p->thread = t;

        // Try to insert that spt_entry into the SPT
        if (hash_insert(t->supptable, &p->hash_elem) != NULL){
          free(p);
          return NULL;
        }
  return p;
}
```

- Recall userprog/process.c/load(): This modification is to serve as the initialization of the address space. Thus, it is remove the idea of loading the binary file directly to virtual address space (solved by Lazy Loading section) and replacing it with allocating the *spt_entry* by inserting into the SPT (solved by Supplemental Page Table section, especially *page_allocate()* function).

- **Stack Intialization**
  - Modify userprog/process.c/setup_stack(): This modification is to initialize the stack whether it could be expand larger through *page_allocate()*, which would be explained in the next section – ***Stack Growth***.

```
static bool
setup_stack (void **esp, char* cmdline)
{
  /…
struct spt_entry* p = page_allocate(userpage, false);
  if (p!= NULL)
  {
    /…
      }
  return false;
}
```

- **Supplemental Page Table Entry Deallocation**
    - Add vm/page.c/page_deallocate_spt_entry(): This function is to remove out the *spt_entry* out of the SPT with the corresponding virtual address. Firstly, it would seek the page to get out through *page_lookup()* function. If the target is pointed to the mapping frame, it is required to deallocate the frame in order not to result in deadlock ***(detailed explanation in On Process Termination)***. In addition, if the target is dirty file that is been modified during executing, it is necessary to write the page on the disk ***(detailed explanation in File Memory Mapping)***. Importantly, under the scope of this section, it is necessary to remove the target page out of the SPT through *hash_delete()*. Therefore, it could fulfill the purpose of deallocating *spt_entry* out of the SPT.

```
void page_deallocate_spt_entry (void *vaddr) {
  struct spt_entry *target = page_lookup(vaddr);

  if (target == NULL)
    sys_exit(-1);

  if (target->frame != NULL){
    frame_lock(target);

    struct frame *f = target->frame; // Simultaneously, deallocate the frame of target spt_entry
/…
  frame_deallocate(f);
  hash_delete(thread_current()->supptable, &target->hash_elem);
  free(target);

  return;
}
```

IV. Stack Growth

a. Overall requirement/scheme

The main requirement of this section is to make the stack expandable, which means make the stack available to increase the size of the stack.
Indeed, there is no file created to implement this section and it is necessary to input some codes inside the existed function in the *page.c*. Overall, to satisfy the above requirement, it is required to add a stack pointer to help for dynamically allocating. If a process accessed the address that is outside the stack, it is a page fault and thus page fault handler needs to solve for the expandable

stack. Therefore, we need to modify the function related to page fault handler in terms of the stack growth problem as well as set up the stack again for initializing the stack.

b. Data Structure

- **Modify threads/thread.h/struct thread{}:** To obtain the expandable stack, it is required to have a stack pointer *esp under the structure of thread. This pointer would record a dynamic allocation for a stack.

```
struct thread
 {
   /..
   void *esp;

 };
```

c. Functions & Algorithms

- **Page Fault Handler:** To recall, *load_page()* would be the function for handling the page fault. Therefore, inside the *load_page()*, with the given *fault_addr*, it would look up the corresponding page. Thus, it would lead to the function *page_lookup()* in order to solve the stack growth
    - Modify vm/page.c/page_lookup(): When it comes to the insufficient space for the stack area, a page fault would be occurred, thus letting the page fault handler yields toward the *load_page()* function and coming to the *page_lookup()* to resolve this problem. Indeed, if the stack needs to grow, the page fault handler would allow the stack to grow but its growth would still limit its size. If the *stack pointer (esp) – 32* is still below the *fault_addr* and its *fault_addr* is over the difference of *PHYS_BASE* and the *STACK_MAX*, it is allowed to allocate a free space for the stack and initialize the Supplemental Page Table.

```
static struct spt_entry *page_lookup (const void *address) {

 if (found == NULL &&  (t->esp - 32) <= address && PHYS_BASE - STACK_MAX <= address){
   struct spt_entry* new_page = page_allocate(address, false);
   return new_page;
 }
 else {
   return hash_entry(found, struct spt_entry, hash_elem);
 }
 return NULL;
 }
```

- **Setup Stack**
    - Recall userprog/process.c/setup_stack(): In the previous section, *page_allocate()* is added to initialize the *spt_entry*, and thus setting up the SPT. However, the idea is also overlapped for the Stack Growth section. Instead of directly allocating the page with *palloc_get_page()* in the previous implementation, it is modified to use *page_allocate()* to aware of the expandable stack scenario and yields to the page fault handler responsibility to solve the stack growth area.

V. File Memory Mapping

a. Overall requirement/scheme

Regarding the File Memory Mapping, it is necessary to implement the concept of opening an accessible file via direct memory access, which is to map them. If wanted to store data, it is necessary to write to the file on the disk. If wanted to read data, the procedure is to come to the file resided on the disk. Therefore, within its requirement, the objective of this section is to map the location within the virtual address space where the system wants the system to bind a given file on the disk. Indeed, storing on the memory address space would directly reflect the content of the file on the disk through the file memory mapping procedure.
Indeed, to implement this part, we would initialize a data structure for the memory-mapping file *(mmf_file)*, particularly for the file-backed pages and the list to keep track of those memory-mapping file. Importantly, it is required to implement two system calls (mmap and munmap) as the core idea for this section within the sided functions to integrate inside those system calls. Moreover, when it comes to the termination process, it is required to implicit unmap all the mappings of the process. Also, if the file-backed pages have been modified (consider through the dirty bit), it is required to write back to the file on the disk.

b. Data Structure

- **Add userprog/syscall.c/struct mmf_file{}:**The data structure of memory-mapping file (mmf_file) is a binding a mapping id between the region of the memory and the file. Its structure contains:
    - elem: The parameter is a *list_elem* type which consider the *mmf_file* as the element of the mmf_list (detailed explanation below)
    - handle: The parameter is the integer type which represents the mapping id of the file. Also, it is considered as the identifier given to recognize the memory-mapping file.
    - *file: The parameter is the pointer to the mapping file object for the purpose of managing the file with the correspondence to the *mmf*.
    - *base: The parameter is the starting point of memory mapping
    - page_cnt: The local parameter would indicate the number of pages get mapping

```
struct mmf_file
{
  struct list_elem elem;      /* List element. */
  int handle;                 /* Mapping id represents the number of MMFs allocated in the process */
```

```
  struct file *file;        /* Mapping file object*/
  uint8_t *base;            /* Start of memory mapping */
  size_t page_cnt;          /* Number of pages get mapping */
};
```

- **Modify threads/thread.h/struct thread{}:** Under the thread structure, it is required to add the list structure as *mmf_list* for containing the opened file as *mmf_file*.

```
struct thread
 {
   /..
   struct list mmf_list;


 };
```

c. Functions & Algorithms

- **Initialization of mmf_list**
  - Modify threads/thread.c/init_thread(): When the thread is initialized, it is required to initialize the data structure to manage the *mmf_file*, which is *mmf_list* through the *list_init()* function.

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  /..
  list_init(&t->mmf_list);
}
```

- **Function to get mmf_file**
  - Add userprog/syscall.c/get_mmf_file(): The function would search the mmf_file within the given identifier of mmf_file, which is the mapping id – handle. The main idea is to iteration through the mmf_list of current thread to seek for the mmf_file, which has the same mapping id – handle. If it is found, it would return the file descriptor associated with it. Otherwise, it would return the NULL pointer and thus terminating the process.

```
static struct mmf_file *get_mmf_file (int handle) {
  struct thread * t = thread_current();
  struct list_elem * elem;
  for (elem = list_begin(&t->mmf_list); elem != list_end(&t->mmf_list);
     elem = list_next(elem)){
```

```
        struct mmf_file *m = list_entry(elem, struct mmf_file, elem);

        if (m->handle == handle) {
          return m;
        }
      }
  sys_exit(-1);
  return NULL;
}
```

- **System Call Handler implementation for mapping and unmapping system call**
  - Add userprog/syscall.c/sys_mmap(): This function is to load the file data into memory and return the id of the memory-mapping file. Following the procedure of the mapping system call, it would check the condition whether it would be the appropriate file or not. If the virtual address is NULL or its page offset is 0, fails. Then, it would create a *mmf_file* through *malloc()* function in order to initialize the mapping procedure. If the file is NULL, fails. Also, it would look up the file descriptor whether NULL or not. Before the procedure of mapping, it is necessary to get the file length and push the *mmf_file* into the *mmf_list* after checking the required conditions. Thus, the procedure of mapping is to account the mapping to memory by taking an address as an input. Besides adding it to *mmf_list* previously, it is essential to initialize the *spt_entry* it self by assigning the important content to its entry inside the SPT, which is the process of initialize the *mmf_file*. With more detailed explanation, since the *mmf_file* is a file-backed page correspondence, it needs to create a spt_entry. After assigning the content of *spt_entry* appropriately, it would come to the next page for assigning other files. When the process of generating *spt_entry* and assigning the vital information are done, it would return the id of *mmf_file* where contains a unique id within a process to identify a mapped file.

```
static int sys_mmap (int handle, void *addr)
{
  if (addr == NULL || pg_ofs (addr) != 0) //The address is not NULL or its offset is 0, fails
    return -1;
  struct mmf_file *m = malloc (sizeof (struct mmf_file));
  if (m == NULL)  //Its file is NULL, fails.
    return -1;
  struct file_descriptor * fd = lookup_fd(handle);
  if (fd == NULL) //Its file_descriptor is NULL (infer STDIN or STDOUT), fails
    return -1;
```

```
struct thread * t = thread_current();
m-> handle = t->n_file;
t->n_file ++;

lock_acquire(&write_lock);
m->file = file_reopen(fd->file);
lock_release(&write_lock);

if (m->file == NULL){
  free(m);
  return -1;
}

lock_acquire(&write_lock);
int length = file_length(m->file);
lock_release(&write_lock);

int ofs = 0;

m->page_cnt = 0;
m->base = addr;
list_push_back(&t->mmf_list, &m->elem);
while(length > 0){
  struct spt_entry *p = page_allocate((uint8_t *) addr + ofs, false);
  if (p == NULL) {
    unmap(m);
    return -1;
  }
  p->file = m->file;
  p->file_ofs = ofs;
  if (length >= PGSIZE) p->file_size = PGSIZE;
  else p->file_size = length;
  p->swap_status = false;
  ofs += p ->file_size;

  m->page_cnt ++;
  length -= p->file_size;
```

```
  }
  return m->handle;
}
```

- Add userprog/syscall.c/unmap(): This function is to remove the mapping from the virtual address space. If there are some files contents are changing, it would be written back to the file on the disk through *page_deallocate_spt_entry(),* which is illustrated below). The procedure is simple that it is needed to deallocate the *spt_entry* form the mapping file since it did initialize the *spt_entry* when it is performed in the mapping system call. Finally, it is essential to close the mapping file and free all the *mmf_file* through the *free()* function.

```
static void unmap (struct mmf_file *m)
{
  list_remove(&m->elem);
  int i;

  for (i = 0; i < m->page_cnt; i++){
    page_deallocate_spt_entry((void *)(m->base) + PGSIZE*i);
  }
  file_close(m->file); //Close the file
  free(m); //Free all the mmf_file
  return;
}
```

- Add userprog/syscall.c/sys_munmap(): The system call is implemented the concept of the *unmap()* function above, which is to release the *mmf_file* inside the *mmf_list.* Firstly, it would invoke *get_mmf_file()* to look for the *mmf_file* which is going to unmap. Then, it would deallocate the connection and delete that file to the procedure of unmapping through the *unmap()* function. Finally, the last part is to close and free the *mmf_file*.

```
static int sys_munmap (int mmf_file)
{
  unmap (get_mmf_file (mmf_file));
  return 0;
}
```

- Modify userprog/syscall.c/syscall_handler(): As the mapping and unmapping are the system call function, it is needed to invoke in the system call handler.

```
case SYS_MMAP:
  value = sys_mmap(args[0],args[1]);
  break;
case SYS_MUNMAP:
  value = sys_munmap(args[0]);
  break;
```

- **Termination section**
  - Modify userprog/syscall.c/sys_exit(): As mentioned previously, when the process exits, all the mappings of the process should be implicitly unmapped afterwards through *unmap()* function. It would iterate through the *mmf_list* where contains all the memory-mapping file and thus it would unmap all the elements inside the *mmf_list.*

```
void sys_exit (int status){
  /..
  for (elem = list_begin(&t->mmf_list); elem != list_end(&t->mmf_list); elem = next_elem){
        struct mmf_file *m = list_entry(elem, struct mmf_file, elem);
        next_elem = list_next(elem);
        unmap(elem);
     }
  thread_exit();
}
```

  - Modify vm/page.c/page_deallocate_spt_entry() & evict_page(): Aforementioned, since the memory-mapping files is the file-backed pages (*PAGE_ZERO* and *PAGE_FILE* type), it is necessary to write the dirty page on page if the those files have been modified. To fulfill it, *evict_page()* has implemented that idea with the *file_write_at()* function if the file-backed pages are dirty pages. Essentially, more details in *evict_page()* function would be demonstrated in order to visualize the big picture of eviction policy in the ***Swap Table section.***

```
void page_deallocate_spt_entry (void *vaddr) {
/…
   if (!target->swap_status && target->file)
     evict_page(target);
   frame_deallocate(f);
```

```
 }
}

bool evict_page (struct spt_entry *p) {
 /..
 if (pagedir_is_dirty(p->thread->pagedir, p->addr)){
  if (p->file != NULL){
   file_write_at(p->file, p->frame->base, p->file_size, p->file_ofs);
   flag = true;
  }
  p->frame = NULL;
  pagedir_set_dirty(p->thread->pagedir, p->addr, false);
 }
 return flag;
}
```

VI. Swap Table

a. Overall requirement/scheme

In terms of the Swap Table, it is a following part for the Frame Table, where we would use some defined data structure in the previous section. Generally, when the physical frames represented as the frame_table are run out of the free frames, it is required to evict a page as a victim and put the copy of the page on the swap disk (if needed) to get the free frame. This section is to fulfill the *PAGE_SWAP* type. Indeed, when a page fault handler finds a page that is not resided in the memory but resided in the swap disk, it is essential to allocate a new frame and move it into the memory.

To implement this concept, we have to create two files *"swap.c"* and *"swap.h"*

b. Data Structure

- **Add vm/swap.h/swap_disk:** The parameter is a block data structure which its implementation is useful in the process of reading and writing the disk. Thus, it is required to store the corresponding block as the pointer to fulfill read or write operations.

```
static struct block *swap_disk;
```

- **Add vm/swap.h/swap_table:** The parameter is a bitmap data structure to manage the swapping area efficiently. The table consists of 0 and 1, where 1 means the corresponding block can be swapped out.

```
static struct bitmap *swap_table;
```

- **Add vm/swap.h/swap_lock:** The parameter is a lock data structure. Since swapping could happen simultaneously among multiple processes, its synchronization primitive is required to protect the swap table.

```
static struct lock swap_lock;
```

- **Modify vm/page.h/struct spt_entry{}:** Under the structure of *spt_entry*, it is required to add a *sector* field as the starting section in the swap partition in order to let the system write or not (-1 is indicated that page does not have a swapping slot). In addition, we add a *swap_status* as boolean variable to indicate the page type (true is swapping page (anonymous page), false is a file-backed pages).

```
static struct spt_entry
{
   /...
   block_sector_t sector;
   bool swap_status;
};
```

c. Functions & Algorithms

- **Initialization of the swap table**
    - Add vm/swap.c/swap_table_init(): This function is to initialize the swap table under the bitmap structure. Firstly, it obtains the block as *swap_disk* through the *block_get_role()* by passing in the role of *BLOCK_SWAP*. Then, it is necessary to check whether the *swap_disk* not to be NULL. If it is NULL, it is required to initialize the bitmap with 0. Otherwise, create a bitmap through *bitmap_create()* within its *block_size()* of the *swap_disk* divided by the *SECTOR_NUM*, which is the number of sectors per page. Furthermore, it is essential to initialize a lock to protect the *swap_table*, which is *swap_lock*.

```
void
swap_table_init (void)
{
 swap_disk = block_get_role (BLOCK_SWAP);
 if (swap_disk == NULL)
   {
    printf ("no swap device--swap disabled\n");
    swap_table = bitmap_create (0);
   }
```

```
  else
    swap_table = bitmap_create (block_size (swap_disk)/ SECTOR_NUM);
  if (swap_table == NULL)
    PANIC ("couldn't create swap table");
  lock_init (&swap_lock);
}
```

- Modify threads/init.c/main(): When the thread is executed, it is required to initialize the swap table through the newly created function *swap_table_init()*.

```
int
main (void)
{
  /..
  swap_table_init();
  /..
}
```

- **Add swap in function**
  - Add vm/swap.c/swap_in(): This function is to load a page from a disk, particularly swapping in an anonymous page which acquires the lock frame in the frame table. The argument passed in would be the *spt_entry*. If the current thread frame does not hold the frame lock, it would provoke *sys_exit()*. Then, in the process of swapping in a page, it would read the value of the corresponding slot to the given address (*base*) through *block_read()*. While updating the sector of *spt_entry* to indicate there is no mapped sector previously, it is required to set the *swap_table* to indicate that it could be swapped out through *bitmap_reset()*. All the task performed should mutually exclusive by the *swap_lock* protection.

```
bool swap_in (struct spt_entry *p)
{
  if(!lock_held_by_current_thread(&p->frame->lock))
     sys_exit(-1) ;

  int i;
  for (i = 0; i < SECTOR_NUM; i++)
    block_read(swap_disk, p->sector + i, p->frame->base + i * BLOCK_SECTOR_SIZE);

  lock_acquire(&swap_lock);
  bitmap_reset(swap_table, p->sector / SECTOR_NUM);
```

```
    /* Update to show this page has no mapped sector now */
    p->sector = (block_sector_t)(-1);
    lock_release(&swap_lock);

    return true;
}
```

- **Add swap out function**
  - Add vm/swap.c/swap_out(): This function is to swap out a page for anonymous page, which is loked in the frame data structure and thus saving those pages to the disk. The argument passed in the function is the *spt_entry*. Firstly, it is needed to check whether that page is holding a frame lock or not. If not, it indicates that page is not in the frame data structure and thus provoking *sys_exit()* afterwards. Within providing a mutual exclusion through *swap_lock*, to find appropriately empty space in the *swap_table* in an atomic function, it is necessary to use *bitmap_scan_and_flip()* in order to let the page reside in when it is swapped out. In addition, updating the starting sector of that page and releasing a *swap_lock* is the following task. According to that *destination_id*, the content of the swap-out page could be written in the sector through *block_write()*.

```
bool swap_out (struct spt_entry *p)
{
  if(!lock_held_by_current_thread(&p->frame->lock))
      sys_exit(-1) ;

  lock_acquire(&swap_lock);
  size_t destination_id = bitmap_scan_and_flip(swap_table, 0, 1, false);

  // If there is an error or no found of the swapping place, release a swap_lock and return.
  if (destination_id == BITMAP_ERROR) {
    lock_release(&swap_lock);
    return false;
  }
  p->sector = destination_id * SECTOR_NUM;
  lock_release(&swap_lock);

  // Write page content to sector in swapping area
  int i;
  for (i = 0; i < SECTOR_NUM; i++)
```

```
    block_write(swap_disk, p->sector + i, p->frame->base + i * BLOCK_SECTOR_SIZE);
  return true;
}
```

- **Function modification for loading an anonymous page (PAGE_SWAP)**
  - Modify vm/page.c/do_page_in(): As mentioned previously, *do_page_in()* is sided function for *load_page()* to support page fault handler to resolve the lazy loading section, When the corresponding spt_entry is related to the anonymous page *(PAGE_SWAP)*, it implies that the anonymous page should be swapped in instead of loading normally as *PAGE_FILE* or *PAGE_ZERO*. Therefore, to fulfill the process of lazy loading, *swap_in()* would be provoked by the page having a swap slot.

```
static bool do_page_in (struct spt_entry *p) {
  /..
  if (p->sector != (block_sector_t)(-1))
  {
    if (!swap_in(p))
      return false;
  }
}
```

- **Function modification for eviction policy for anonymous page (PAGE_SWAP)**
  - Add vm/page.c/evict_page(): This function is related to the page replacement policy for all the page type, where the page is held the frame lock. Through the function *frame_lock()* and *frame_unlock()* that are implemented in the *frame.c*, those functions are integrated the concept of choosing the victim to be evicted. Then, the following step is to clear the page directory through *pagedir_clear_page()*. There are two types of files, which are anonymous file and file-backed pages. If the file-backed pages are dirty which has been modified, it would be written back to the disk as implemented in the File-Memory Mapping section. On the other hand, if the file is the anonymous file, it should provoked *swap_out()* to swap out the *PAGE_SWAP*. Finally, set the frame of *spt_entry* to be NULL and set its dirty bit to be normal.

```
bool evict_page (struct spt_entry *p) {
/..
  if (pagedir_is_dirty(p->thread->pagedir, p->addr)){
    if (p->swap_status)
      flag = swap_out(p);
   }
```

- Recall vm/page.c/page_deallocate_spt_entry(): During the process of deallocating the spt_entry, when it is terminated, it is required to deallocate the space in the memory, thus performing the swap out if the corresponding file is the anonymous page.

VII. On Process Termination

a. Overall requirement/scheme

As regards to On Process Termination, it would deallocate all the using resources when a process is terminated. Particularly, it is necessary to destroy supplemental page table, deallocate the physical frames as well as each frame entry in the frame table, and free the swap area (if have) as well as the corresponding entries in the swap table. Finally, it is necessary to close all the files after writing the dirty memory-mapping pages from the memory back to the disk if the dirty bit is notified.
In addition, there would occur the deadlock issue. During the system call, the objective is to prevent evicting pages accessed and also define a flag in each physical page. Hence, it is required to find the virtual page and pin the associated physical frames.
Overall, there are no files created since it is required to add more functions related to the resource deallocation and solution of deadlock problem.

b. Data Structure

There is no data structure added in this section.

c. Functions & Algorithms

- **Resource deallocation:** In the previous section, we did implement several functions to deallocate the *spt_entry*, the physical frames, the swapping area. The final step is to initiate the function to deallocate the supplemental page table.
    - Add vm/process.c/destroy_page(): This function would destroy all the page table entries and also its corresponding frame connection through *frame_deallocate()* function. Afterwards, it would free all the *spt_entry* of the SPT. Indeed, it would use as the function argument for the *hash_destroy()* in the flowing function.

```
static void destroy_page (struct hash_elem *p_, void *aux)  {
  struct spt_entry * des_spt_entry = hash_entry(p_, struct spt_entry, hash_elem);
  frame_lock(des_spt_entry);
  if (des_spt_entry->frame != NULL)
     frame_deallocate(des_spt_entry->frame);
  free(des_spt_entry);
  return;
}
```

- Add vm/process.c/page_deallocate_supptable(): The function has the same objective as the above function, which is destroy the supplemental page table content using the *hash_destroy()* function.

```
void page_deallocate_supptable (void)  {
  struct hash *supptable = thread_current()->supptable;
  if (supptable != NULL)
    hash_destroy(supptable, destroy_page);
  return;
}
```

- Modify userprog/process.c/process_exit(): When the process is exited, it is required to deallocate all the spt_entry and thus removing all the content in the SPT through newly created function *page_deallocate_supptable()*.

```
void
process_exit (void)
{
  /..
  page_deallocate_supptable ();
   /..
}
```

- **Deadlock Prevention in the system call**
  - Add vm/page.c/page_lock(): To prevent the deadlock, it is required to create a new function in order to try to lock the page corresponding with the address into physical memory. If the writable is true, the page must be writable; otherwise it may be read_only. Firstly, it would check whether the range of conditions to let the page acquire the lock. The purpose of designing a meticulous lock is to avoid the deadlock instead. If not, they would acquire the lock so freely and thus leading to the deadlock easily. Under the code level, after acquiring the lock, it would check more conditions for the *do_page_in()* and *pagedir_set_page()*. In terms of the *do_page_in()*, it would not allow the swap-out page to acquire the lock or get spinning. Indeed, this function does a lock frame and release the frame lock if it fails. Regarding the *pagedir_set_page()*, it would check the procedure of memory allocation successful or not.

```
bool page_lock (const void *addr, bool writable) {
  struct spt_entry *p = page_lookup(addr);
  if (p == NULL)
     return false;
  if (p->read_only && writable)
```

```
    return false;


  frame_lock(p);
  if (p->frame == NULL)
  {
    if (!do_page_in(p))
      return false;
    if (!pagedir_set_page (thread_current()->pagedir, p->addr,p->frame->base, !p->read_only))
      return false;
  }
  return true;
}
```

- Add vm/page.c/page_unlock(): Opposite with the page_lock(), this function unlocks a page locked by the page_lock() within the function frame_unlock()

```
void page_unlock (const void *addr) {
  struct spt_entry *p = page_lookup(addr);
  if (p == NULL)
    sys_exit(-1);
  frame_unlock(p->frame);
  return;
}
```

- Modify userprog/syscall.c/copy_argument(), sys_open(), sys_create(), sys_read(), sys_exec(), sys_write(): To avoid the deadlock, within creating a file mutually exclusive, it is required to use *copy_argument_string()*, which is a function counted more conditions to emphasize which associated physical frames get executed during the *page_lock()* and *page_unlock()*. Therefore, it would pin a file from the temporary file and free it afterwards. Regarding the deadlock prevention, besides executing the system call in general, it is essential to let one process use the resources of the physical frame within *page_lock()* and *page_unlock()*. Thus, it could resolve the deadlock issue by pinning the files within two functions afterwards.

VIII. Discussion

Overall, from the project reflection, I have more understanding in terms of the current PINTOS system. It does support the multiprogramming and load the entire executable file (date, code, stack segments) into memory at once before getting executed a program. By this approach, it is inefficient since there are not all files executed and thus leading the recourse waste in the memory. Therefore, it is necessary to implement the concept of Lazy Loading according to the page type, particularly the

file-backed pages. Furthermore, the physical address of each page in the virtual address space are fixed at the beginning, thus resulting in the fixed size of stack on each process. Hence, Stack Growth is the idea to consider the expandable stack to resolve the fixed size of stack in the naïve PINTOS. In addition, when it comes to anonymous page, its performance is not the same as the file-backed pages. Thus, it is required to implement the concept of Swap Table, which is still resided in the disk but it is on the swap partition. Finally, to connect the file between the memory and disk, File Memory Mapping is required to binding a mapping file to fulfill its requirements.