Operating Systems – Project 3: Virtual Memory

Design Report

Khau Lien Kiet

Team 10

2022/10/14

Department of Computer Science & Engineering

Pohang University of Science and Technology (POSTECH)

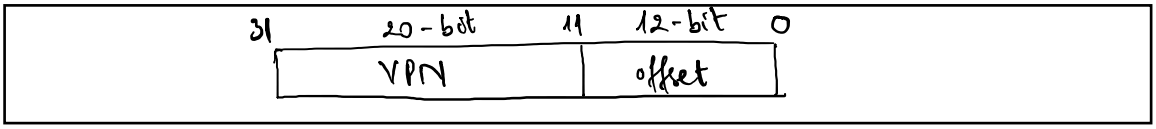
Table of Contents

I. Analysis of current system

a. Background and Terminology

- **Pages:** It is also called as virtual pages that contains the continuous region of virtual memory with the size of 4,096 bytes. A page must be page-aligned that it should start on virtual address evenly divisible by the page size. For instance, a 32-bit virtual address is separated as 20-bit for VPN (Virtual Page Number) and 12-bit offset. ($2^{20} = 4{,}096$ virtual pages).

  Interpretation for 20-bit VPN and 12-bit offset:



  One process has its individual virtual/user pages due to the virtualization of OS where virtual RAM provides the illusion of private memory. Its user space region is under the *PHYS_BASE (0xc0000000)*, while the kernel space, above the *PHYS_BASE*, is represented as global area that is shared for many processes' usage. In terms of the responsibility, the kernel can access both kernel and user pages, whereas the user can only access on their own user space.

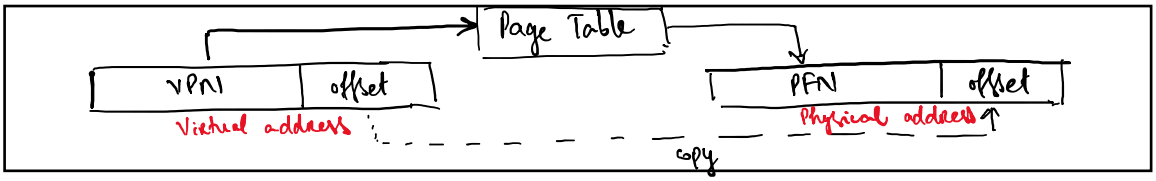
- **Frames:** It is also called as physical frame that contains the continuous region of physical memory. Same as pages, frames must be aligned as well. Given 32-bit physical address, we could divide 20-bit for PFN (Physical Frame Number) and 12-bit for offset.

  Interpretation for 20-bit PFN and 12-bit offset:

  Within the address translation, PINTOS could map the kernel virtual memory to physical memory for directly accessing memory at physical address. By its mechanism, the first page of kernel virtual memory is to map with the first frame of physical memory and so on.

  In addition, there is some provided functions in PINTOS to perform the conversion between virtual address and physical address (i.e. *threads/vaddr.h/ptov()*, *threads/vaddr.h/vtop()*)

- **Page Tables:** It is a data structure that CPU provides for address translation that is to translate the virtual address to physical address (VPN to PFN, or page to frame). The diagram below illustrates the translation among virtual address to physical address.

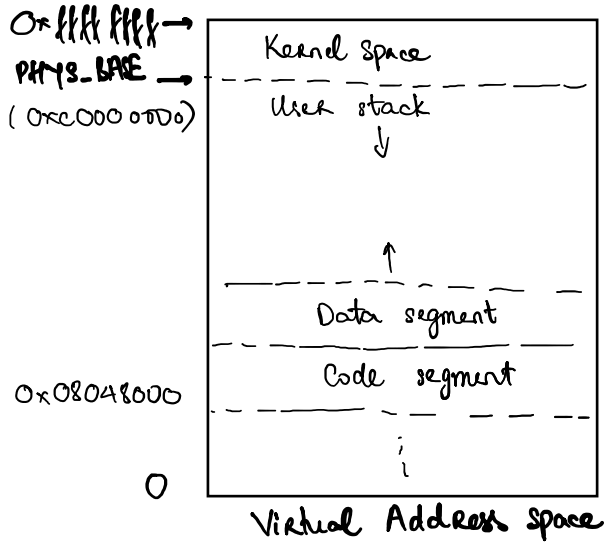


  Some functions for page table management are provided in *pagedir.c* ***(detailed explanation in I.d)***

- **Swap slots:** are page-aligned and imply for the page-size region (array of page-size block) of swap partition that resides in the disk. The swap area would be used in the Swap Table's implementation.

b. Stack Structure

- **Stack Layout:** PINTOS virtual memory is divided the kernel space and user space, which is ranging from *0xc0000000* to *0x08048000*.



- **Initialization of stack through setup_stack(void *esp):** Before loading, the stack of the process is created through *setup_stack()*. Originally, the process is mapped the page to the memory after it has been allocated through *palloc_get_page()* and *install_page()*. However, the current implementation is not considered the expandable stack since the stack only resides in 4KB size. Thus, if it exceeds that amount, it would occur a page fault that terminates the process directly. Therefore, to resolve this issue, Stack Growth would define and explain more clearly.

c. Page Allocation

Before going to implementation section, it is necessary to understand some functions provided from PINTOS for a future usage. When the process starts to execute, page allocation should be performed to initially set up its own address space.

- **process.c/install_page(void *upage, void *kpage, bool writable):** This function is to map the physical page (*kpage*) and virtual page (*upage*) or matches the virtual page to the physical page. Indeed, it would connect the page to the page directory of the current executed thread, verify that the directory of the current thread is empty, and set the page to the directory through *pagedir_set_page()*. In terms of *writable* variable, while 1 is represented as *writable*, 0 stands for *read-only*.

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

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

- **palloc.c/palloc_get_page(enum palloc_flags flags):** This function is to allocate 4KB page and return physical address of page. There are three types of flags such as *PAL_USER* (allocate pages from the user memory pool), *PAL_KERNEL* (allocate pages in kernel memory pool), *PAL_ZERO* (initialize pages to 0). Indeed, it would provoke *palloc_get_multiple()* ***(further explanation below)***

```
void *
palloc_get_page (enum palloc_flags flags)
{
  return palloc_get_multiple (flags, 1);
}
```

- **palloc.c/palloc_get_multiple(enum palloc_flags flags, size_t page_cnt**): Within this function, it would take the number of allocated pages within the type of flag and return the corresponding address. Simultaneously, it selects a pool from the user or kernel pool based on the assigned flags and thus finding a new location for a new page through *bitmap_scan_ and_flip()*. Thus, if the allocation succeeds, it would return the address. Otherwise, it is required to result in Kernel panic.

```
void *
palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)
{
  struct pool *pool = flags & PAL_USER ? &user_pool : &kernel_pool;
  void *pages;
  size_t page_idx;

  if (page_cnt == 0)
    return NULL;

  lock_acquire (&pool->lock);
  page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);
  lock_release (&pool->lock);

  if (page_idx != BITMAP_ERROR)
    pages = pool->base + PGSIZE * page_idx;
```

```
  else
    pages = NULL;

  if (pages != NULL)
    {
      if (flags & PAL_ZERO)
        memset (pages, 0, PGSIZE * page_cnt);
    }
  else
    {
      if (flags & PAL_ASSERT)
        PANIC ("palloc_get: out of pages");
    }

  return pages;
}
```

- **palloc.c/palloc_free_page(void *page**): This function is to use physical address of page as an argument and thus freeing a given page by putting page back in the free memory pool. Indeed, it would provoke *palloc_free_multiple()* ***(further explanation below).***

```
void
palloc_free_page (void *page)
{
  palloc_free_multiple (page, 1);
}
```

- **palloc.c/palloc_free_multiple(void *pages, size_t page_cnt):** With the same idea as the above function, instead of freeing a single page, this function is to deallocate multiple pages. Through *bitmap_set_multiple(),* it is deallocated from the bool and thus updated the pool usage map after verifying its page as the user or kernel pool.

```
void
palloc_free_multiple (void *pages, size_t page_cnt)
{
  struct pool *pool;
  size_t page_idx;

  ASSERT (pg_ofs (pages) == 0);
  if (pages == NULL || page_cnt == 0)
```

```
    return;

  if (page_from_pool (&kernel_pool, pages))
    pool = &kernel_pool;
  else if (page_from_pool (&user_pool, pages))
    pool = &user_pool;
  else
    NOT_REACHED ();

  page_idx = pg_no (pages) - pg_no (pool->base);

#ifndef NDEBUG
  memset (pages, 0xcc, PGSIZE * page_cnt);
#endif

  ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));
  bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
}
```

- **process.c/load_segment():** When the process gets executed, this function is to initially set up the segment. In the naïve PINTOS, the main functionality is to allocate page, load the data from the file to the page, and set up the page table. Indeed, the kernel and user page to store the executable file are mapped together after it is loading in memory at one time. However, it results in the downside when loading at one since there is a waste whether some files are not used to execute in a program. Therefore, to resolve this issue, Lazy Loading would be implemented in the current PINTOS ***(further explanation in II.b)***

```
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;
```

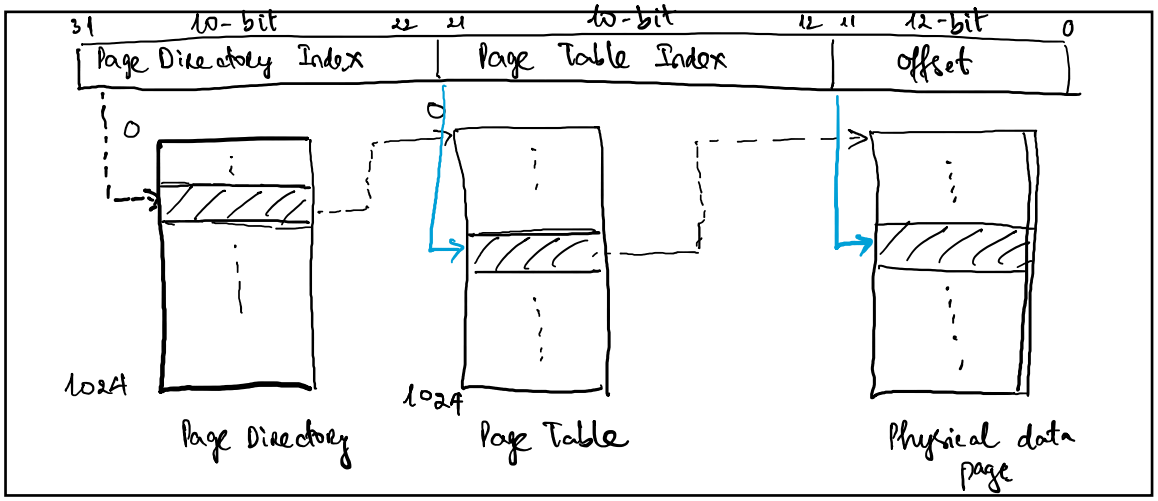
```
      /* Get a page of memory. */
      uint8_t *kpage = palloc_get_page (PAL_USER);
      if (kpage == NULL)
        return false;

      /* 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;
}
```

d. Page Table Structure

To recall the usage of page table, it is a data structure resided in the memory to provide for address translation that performs mapping from page number to frame number. In the PINTOS implementation, it provides with two-level page table which contains the first level page table as the page directory and the second level page table as page table. In terms of the page directory, it is an abstract interface to manage the page table.

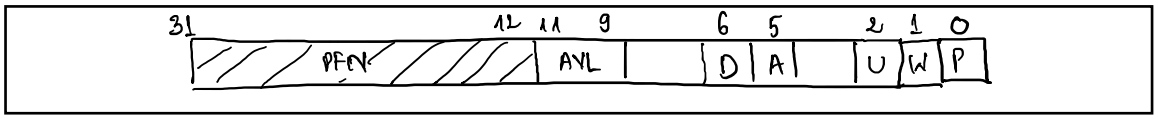
- **Page Structure:** In the two-level page table, it consists of Page Direction (PD) is an array of 1,024 Page Directory Entries. Within each entry of PD, it would point to another page level which is called Page Table (PT) that includes 1,024 page table entries (PTE). After integrating the page offset, we would result in the physical address.

The diagram illustrates the detailed conversion from virtual address to physical address in the two-level page table performance:



In general, the virtual address is divided into three sections (Page Directory Index, Page Table Index, Offset). Firstly, the top-most significant bit *(bit 22nd – 31st)* is represented as the index of Page Directory. If there is a corresponding page table from Page Directory Index, it would point to the beginning of that page table. Secondly, the next bit *(bit 12th – 21st)* is the index of page table. Inside each page table entry, it would consist of the information of physical frame number (PFN) and several additional information ***(described in the Page Table Entry Format).*** Finally, the least significant bit *(bit 0th – 11th)* is represented as the offset. After successfully seeking the frame, it would search the accurate region by the offset. Therefore, physical address would be returned.

- Page Table Entry Format



Page Table is page-aligned so that the least significant 12 bit is used to represent as additional information for the PTE. Initially, it would be initialized to 0.

  - **AVL (bit 9th – 11th):** These bits indicate whether OS is available or not
  - **D (bit 6th):** This bit is represented as the *"dirty" bit*. When writing to the memory space, the dirty bit is set to 1 by hardware. When the page contains a *dirty* bit of 1 as a victim due to the eviction policy ***(explained in II),*** the updated page must store on the disk.
  - **A (bit 5th):** This bit is represented as the *"accessed" bit*. Each time the page is referenced, the *accessed* bit is set to 1 by hardware.
  - **U (bit 2nd):** This bit is represented as the "user or supervision" bit. If 1, the user or kernel can access the page. If 0, only kernel gets accessed to the page.
  - **W (bit 1st):** This bit is represented as allowing to "read and/or write" on the page. If 1, read or write could be performed. If 0, read-only.
  - **P (bit 0th):** This bit is represented as the *"present"* bit. If 1, the entry is valid. If 0, it occurs the page fault when accessing that entry.

- In *pagedir.c*, current PINTOS has some fundamental functions which is to manage and validate or update the page table:

- **pagedir_create(void):** This function is to create and return a new page table, which is in Kernel virtual page. If there is no page allocation in the memory, it would return the *NULL*.

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

- **pagedir_destroy(pd):** This function is to free or destroy all the resource reference by page directory *(pd)*, which is not only the resource itself but also the corresponding frame.

```
void
pagedir_destroy (uint32_t *pd)
{
  uint32_t *pde;

  if (pd == NULL)
    return;

  ASSERT (pd != init_page_dir);
  for (pde = pd; pde < pd + pd_no (PHYS_BASE); pde++)
    if (*pde & PTE_P)
      {
        uint32_t *pt = pde_get_pt (*pde);
        uint32_t *pte;

        for (pte = pt; pte < pt + PGSIZE / sizeof *pte; pte++)
          if (*pte & PTE_P)
            palloc_free_page (pte_get_page (*pte));
        palloc_free_page (pt);
      }
  palloc_free_page (pd);
}
```

- **pagedir_activate(pd):** This function is to solve the memory reference by page directory *(pd)*.

```
void
```

```
pagedir_activate (uint32_t *pd)
{
  if (pd == NULL)
    pd = init_page_dir;
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}
```

- **pagedir_set_page(pd, *upage, *kpage, writable):** When this function is provoked, it would add kernel virtual address from the user page (*upage*) to the kernel page (*kpage*). At the time of calling this function, *upage* must be a page that is not mapped with any *kpage* and thus it would allow to execute a page setting. Afterwards, the function would return the boolean as true if the mapping is successful.

```
bool
pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)
{
  uint32_t *pte;

  ASSERT (pg_ofs (upage) == 0);
  ASSERT (pg_ofs (kpage) == 0);
  ASSERT (is_user_vaddr (upage));
  ASSERT (vtop (kpage) >> PTSHIFT < init_ram_pages);
  ASSERT (pd != init_page_dir);

  pte = lookup_page (pd, upage, true);

  if (pte != NULL)
    {
      ASSERT ((*pte & PTE_P) == 0);
      *pte = pte_create_user (kpage, writable);
      return true;
    }
  else
    return false;
}
```

- **pagedir_get_page(pd, *uaddr):** When the function is called, user would find a frame with virtual memory address to be mapped. If the mapping frame is existed, it would return the kernel virtual memory address. Otherwise, *null pointer* would return.

```
void *
```

```
pagedir_get_page (uint32_t *pd, const void *uaddr)
{
  uint32_t *pte;

  ASSERT (is_user_vaddr (uaddr));

  pte = lookup_page (pd, uaddr, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    return pte_get_page (*pte) + pg_ofs (uaddr);
  else
    return NULL;
}
```

- **pagedir_clear_page(pd, *upage):** Within this function, *pd* of *upage* is represented by the subsequent *upage* access. If *upage* is not mapped, it would return in fault that no action is taken measured.

```
void
pagedir_clear_page (uint32_t *pd, void *upage)
{
  uint32_t *pte;

  ASSERT (pg_ofs (upage) == 0);
  ASSERT (is_user_vaddr (upage));

  pte = lookup_page (pd, upage, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    {
      *pte &= ~PTE_P;
      invalidate_pagedir (pd);
    }
}
```

- As previously mentioned about eviction policy, these functions below could use accessed or dirty bit to fulfill the **page replacement policy**:
- **pagedir.c/pagedir_is_dirty(*pd, *vpage):** This function is to return the *dirty bit* of PTE for virtual page (*vpage*) in page directory (*pd*)

```
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;
}
```

- **pagedir.c/pagedir_set_dirty(*pd, *vpage, dirty):** This function is to set the dirty bit to *dirty* variable in the PTE for *vpage* in page directory (*pd)*

```
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);
        }
    }
}
```

- **pagedir.c/pagedir_is_accessed(*pd, *vpage):** This function is to return *access* bit of PTE for *vpage* in page directory (*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;
}
```

- **pagedir.c/pagedir_set_accessed(*pd, *vpage, accessed):** This function is to set the access bit to *accessed* variable in the PTE for *vpage* in page directory *(pd).*

```
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);
        }
    }
}
```

e. Page Fault

- In the original PINTOS implementation, when a given page is not existed, this function is provoked that let that process terminate directly. In the previous project, when the page fault is occurred, it is acceptable to kill the process; however, it is not accurate toward the project 3. Therefore, to successfully implement, *page_fault()* function should be modified to search on the disk, allocate the page frame, and load the page from disk to that allocated page frame (resided in the memory). When there is no free space in memory, it is necessary to replace the current page. This is the procedure of ***Page Fault Handler***, which is described later ***section II***.

```
static void
page_fault (struct intr_frame *f)
{
  /…

  /* Project 2: System Call - This implementation is to prevent the memory leakage.
  syscall_exit(-1);
  /* 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);
}
```

II. Solution for requirements

Before diving into the details of each requirement, it is necessary to recall the overall picture of the current PINTOS implementation ***(before project 3).*** *It does support the multiprogramming and load the entire executable file (data, code, stack segments) into memory at once before getting executed a program. By this approach, it is inefficient since there are not all the files getting executed. In addition, physical address of each page in the address space are fixed at the beginning, thus resulting in fixed size of stack on each process. Indeed, it is still a restricted design for the improvement of PINTOS. Therefore, it is required to implement "Virtual Address" to support these seven requirements below, especially Lazy Loading, Stack Growth, File Memory Mapping, and Swap Table.*

a. Frame Table

- **Requirement idea:** In the current PINTOS, there is no physical frame representation and thus it is essential to implement a Frame Table for effectively managing physical frames. Indeed, a frame table includes a data structure as a physical frame to contain a user page. Each entry or each frame in the frame table has the pointer pointing toward to the page. For instances, if there are many pages appeared but there are no free space to allocate, it is necessary to obtain the eviction policy to select a page to evict.

- **Implementation of original PINTOS & Problem Identification:** As mentioned previously, if there is no free space to allocate the frame in the frame table, with the current implementation, the existing PINTOS cannot perform as expected. Thus, to resolve this issue, it is required to add new data structure to support on the task of selecting physical frame for replacement. Within those function in previous section in *palloc.c* and *pagedir.c* file, we could implement for the management of the page table and the procedure of allocation/deallocation of page table, respectively.

- **Necessity for requirement:**

- Data Structure: Add a new data structure for frame table
    - **struct fte{ }:** A data structure to represent the physical page that contains a user page, which includes physical address of page (PFN), reference to the virtual page that maps to the physical page (*upage* and *kpage*), reference to the thread structure to which it belongs (*thread *t*), list element for the frame table (*list_elem*). Therefore, Frame Table Entry (*fte*) manages the frame table through kernel virtual page (*kpage)* and user virtual page (*upage).*
    - **static struct list frame_table:** The frame table is a list structure as a global variable to manage the frame table entry (*fte*) as its list element.
    - **static struct lock frame_lock:** This primitive is used for controlling the synchronization of frame table. If there are many processes trying to access the frame table simultaneously, the lock would solve the issue to increase the reliability of the content.
- Algorithms or Functions:
    - **frame_init():** This function is used for initialization of *frame_table* and *frame_lock*.
    - **Allocate/Deallocate frames:**
        - **falloc_get_frame():** This function is used for allocating a frame and return the page address. Within the function, it is required to obtain a free space when memory cannot be allocated through *palloc_get_poage()* within the page allocation function. If it could be allocated, add the frame into the *frame_table* and return its address. Otherwise, if there are no free space to allocate, it would determine the victim frame to evict and thus swap-out victim frame ***(detailed explanation afterwards).***

        - **falloc_free_frame():** This function is used to deallocate the corresponding the frame table entry *(fte).* Its main functionality is to find a frame in the *frame_table* and delete it from the *frame_table*. In addition, *palloc_free_page()* should be provoke and thus freeing the frame.
        - For an easier implementation, it is vital to have the function that *kpage* returns a corresponding frame as *fte* through *get_fte()*
    - **Choose a victim which returns the corresponding frames while there is no free space in the *frame_table***
        - Create a function to perform the eviction policy (i.e. clock algorithm). Thus, the Swap Table section would demonstrate thoroughly.
        - **Modify setup_stack():** It is necessary to allocate frames from the *frame_table* when the page fault occurs. After getting evicted, the memory allocation is used to allocate frames (*falloc*) while growing a stack.
    - **Search frames used by user process:** detailed implementation in Swap Table

b. Lazy Loading

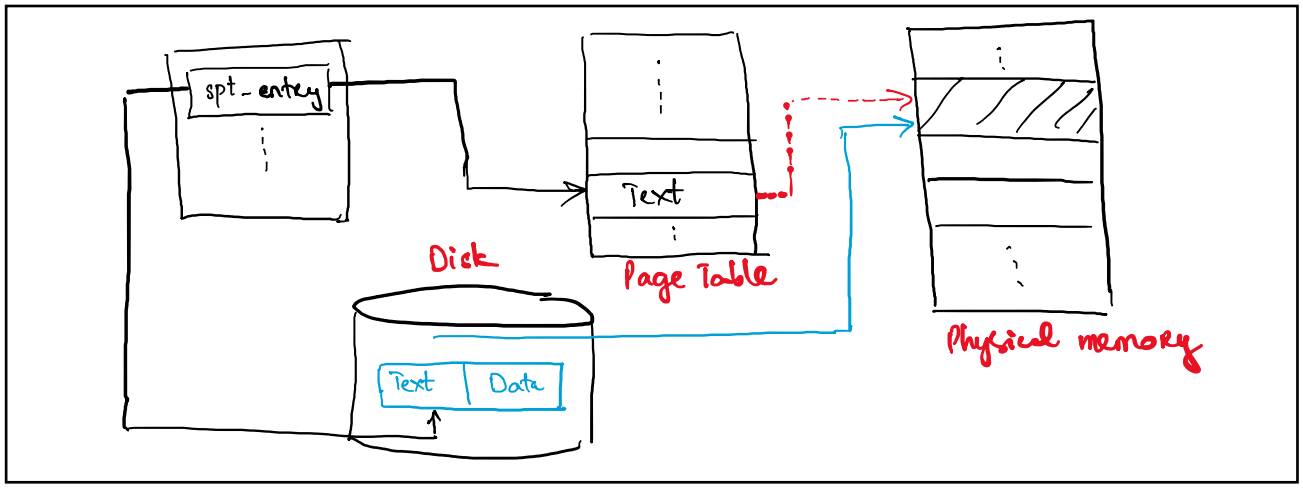
- **Requirement idea:** Lazy loading is a method that only a stack setup part is loaded during loading for memory allocation when a process starts. Other parts are not loaded in the memory whereas pages are allocated. If the page faults are occurred from allocated page, it is necessary to load the page into memory. After handling the page fault, it needs to resume process operation when the above procedure is finished. Indeed, important data should be loaded into the memory to save the memory space.

- **Implementation of original PINTOS & Problem Identification:** In the current PINTOS implementation, when the process starts, all executable codes are loaded into memory such as code, data, stack segment. Within this approach, the memory can be wasted as not all data would be used. Furthermore, when the page fault occurs, it terminates those process immediately or results in Kernel panic, which means that the naive PINTOS is not to handle those faults or resume its operation. Thus, Lazy Loading is to solve the issue.

- **Necessity for requirement:**

- Data Structure: Creation of Supplementary Page Table Entry (*spt_entry*) ***(detailed explanation in Supplemental Page Table section)***
    - **struct spt_entry {}:** Under the structure of Supplemental Page Table Entry, there would have several variables to track the information of an executable file to load the file if needed. To avoid all the executable file loaded into memory, it is required to have these variables as **file* (file to read), *offset* (file offset), *read_bytes* (bytes to read), *zero_bytes* (bytes setting to 0), *writable* (indicating whether the page is written or not). A further variable initialization would show up in the ***Supplemental Page Table section***.
- Algorithm:
    - **init_spte():** This function is to create and return Supplemental Page Table Entry with given file information.
    - **Modify load_segment():** Within the improvement, it is required to integrate *init_spte()* in the function in order to create *spte* and insert to Supplemental Page Table (*spt* – ***detailed explanation below)*** instead of loading a full memory page.
    - **Demand Paging Design**: When application accesses to the page, it concerns the page table and if the page table is not present in memory, it would find on the disk and load from the disk. This demand paging is to handle the page fault of the current PINTOS implementation.

The diagram is the big picture of demand paging:



Firstly, it would access to the memory and finds that it generates the page fault. If the *page_fault()* happens, in the page fault, it would scan through the variable in the *spt_entry* (the set of entries in supplemental page table). In addition, they would check if a given memory access can be serviced by one of the objects in the *spt_entry*. If the OS cannot satisfy the page fault, it starts to read the disk and load it from the disk to memory. Finally, it would update the page table.

However, under the scope of the lazy loading, it is required to modify the *page_fault()* as the main part and add two functions such as handling memory allocation (*handle_mm_fault()*) and loading the disk to the memory (*load_file()*). Thus, we would provide the system with page fault handler.

- **Page Fault Handling:** This stage is to modify the *page_fault()*. To avoid terminating the process, it is required to delete the *kill(-1)*. Then, check whether the given address (*fault_addr*) is valid of not. In addition, we could define a new page fault handler through *handle_mm_fault()* function as mentioned previously.
- **Implementation of new page fault handler:** In terms of this section, it is vital to provide a more detail requirements for *handle_mm_fault()*. When the page fault occurs, this function is provoked to allocate the physical memory, load the file from disk to memory by *load_file()* ***(mentioned later),*** and update the associated page table entry in page table after successfully loading into physical memory. Indeed, the file loaded into physical memory is executable file (ELF File) and there are two more types of files which are introduced in the next part.
- **Loading the File to Physical Memory:** When it comes to this section, *load_file()* is needed to add since it would support the loading ELF File. They would receive two types of arguments which are file address and *spt_entry* – pointer to the virtual page object. When loading the file, it would search the location of the desired block from the disk using *file* structure and its *offset* through *file_read_at()* function to identify the physical location of data needed to read on the disk. Then, in the disk, it would load the block from disk to memory by the page allocation algorithm.

c. Supplemental Page Table

- **Requirement idea:** Within the Supplemental Page Table, it would contain more critical information than the original page table. The main functionality is to decide to load the

corresponding file in the accurate area. For instance, if it is an executable file (file-backed page), the supplemental page table would decide which corresponding page to load. In addition, if it is an anonymous page, the supplemental page table would decide to get swap disk and which partition of swap disk that stores the corresponding page.

- **Implementation of original PINTOS & Problem Identification:** In the current page table entry, the information contains AVL, dirty bit, accessed bit, present bit, writeable bit, user/supervisor bit. However, within that information, they could not completely handle the page fault since it is needed to decide which frame should be loaded in the corresponding area and refer to supplemental page table as a new data structure.

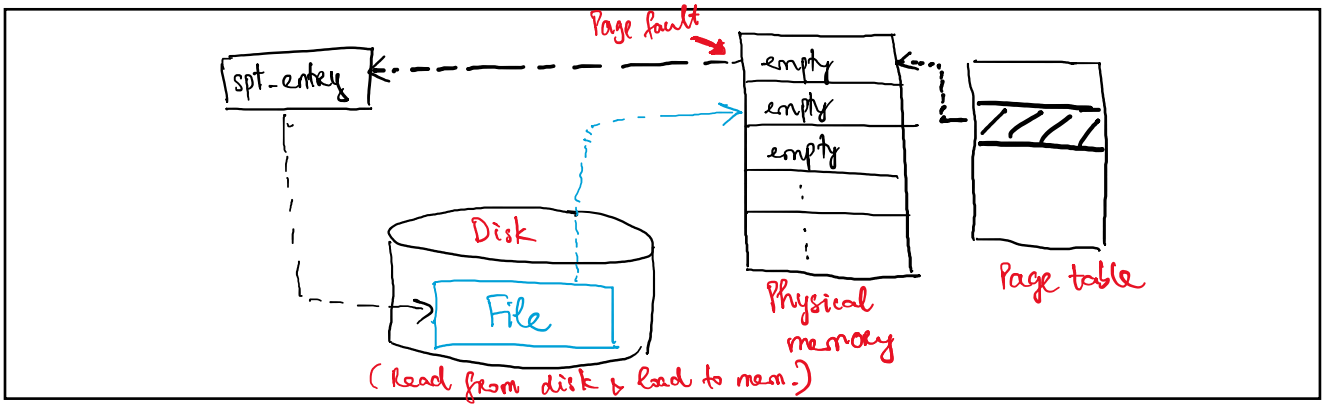
- **Necessity for requirement:**

- Data Structure: Under the structure of supplemental page table entry *(spt_entry)*, we should add more field to store virtual address a field to manage the hash element since the Supplemental Page Table is formed as the hash table. In general, inside its entry, it needs to contain the VPN, read/write permission, the status of virtual page, reference to the file object, offset, amount of data in the page, location in the swap area. Therefore, it is necessary to add those fields in *struct spt_entry{}*.
    - In addition, there are three pages' status that we should consider: a page of ELF executable file, a page of general file, a page of swap area. Thus, it needs to define those three states as *PAGE_FILE* as 2, *PAGE_FRAME* as 1, *PAGE_ZERO* as 0, respectively.
    - **Modify struct thread {}:** Add a supplemental page table field under the thread structure by using hash table as implemented in the current PINTOS. As the virtual address space is allocated per process, it is useful to define the hash table to manage virtual pages.

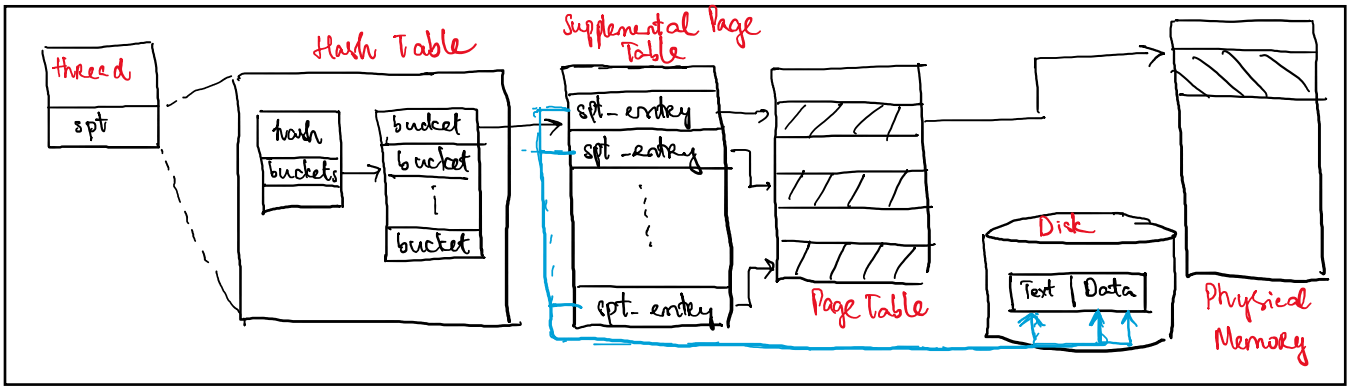
- Algorithm

  To fully complete the implementation of page fault handler, the objective of this section is to modify page_fault(). To fulfill its necessity, the main idea of those should be done as following. If the memory reference is verified, it should locate the content that needs to go into the virtual memory and decide which accurate location based on the status of the page (from the file, from the swap-area, or from the zero-page). Indeed, when it comes to shared page, the page is already the frame. Otherwise, if the access is invalid, it is necessary to kill the process. For example, there is no valid user address, or in the kernel address, or the occurrence of permission error. Thus, it is required to kill the process. Afterwards, the later task is done by Lazy Loading. Particularly, it should scan through all frames and select one page frame to allocate it. Then, load the data from the disk to the frame and update the page table. Hence, it yields to the completion of page fault handler.

  The diagram below is to illustrate the big picture when implementing spt_entry and also briefly describe the picture of page fault handler:



If a given page is not in memory, page fault occurs. Then, it would allocate free frame from the array of frame entries as physical frames. Next, it would allocate appropriate disk block for loading the page in memory to the allocated frame, after getting allocated, it would read the block from disk to memory. In here, supplemental page table is contained *spt_entry* that is to manage virtual page content of the process that would link to page table entry inside the page table. The diagram below shows the implementation of supplemental page table as described:



- **Supplemental Page Table Initialization:** Add a function called *init_spt()* to initialize the hash table with *hash_init()* and modify the *start_process()* function with initializing the set of *spt_entry*.
- **Supplemental Page Table Entry Deallocation:** Modify *process_exit()* with added function to deallocate *spt_entry* and its hash table.
- **Address Space Initialization:** From the original PINTOS, it allocates the physical memory by reading ELF header through read data and code segment from *load_segment()* and allocate the physical page of stack by *setup_stack()*. However, to improve, it is necessary to allocate the page table and allocate the *spt_entry* for each page instead of allocating of physical memory. Importantly, it is essential to modify the *load_segment()*. In terms of the implementation of that function, we should remove the idea of loading the binary file directly to virtual address space and replace with allocating the *spt_entry* structure by *malloc()*, initializing *spt_entry* file through setting *spt_entry* members/offset/size of the file to read/*zero_byte* to pad, and adding those *spt_entry* into the hash table. To fulfill those necessities, it is necessary to add some functions related to initialize the supplemental page table according to the status of the page frame.
- **Stack Initialization:** As mentioned previously, in the original PINTOS, it would allocate a single page, set a page table and it stack pointer (*esp*). However, if the process is run out of space for stack, the process is killed. Thus, it is necessary to add some ideas to modify the *setup_stack()* function. Indeed, it is required to create a *spt_entry* as 4KB stack, initialize the *spt_entry* members, and insert the *spt_entry* in the hash table.
- Those main idea above cannot be implemented without those sub-functions are needed to consider as*: init_spt():* hash table initialization, *delete_spt():* hash table deallocation, *find_spt_entry():* searching *spt_entry* accordance with the virtual address in the address space of the current process, *insert_spt_entry():* insertion of *spt_entry* in the hash table, *delete_spt_entry():* deletion of the *spt_entry* from hash table, *spt_hash_function():* method of hashing to put the *spt_entry* into the hash table, *spt_less_function():* function to compare the address value from two hash element, *spt_delete_function():* function to remove *spt_entry* from memory.

d. Stack Growth

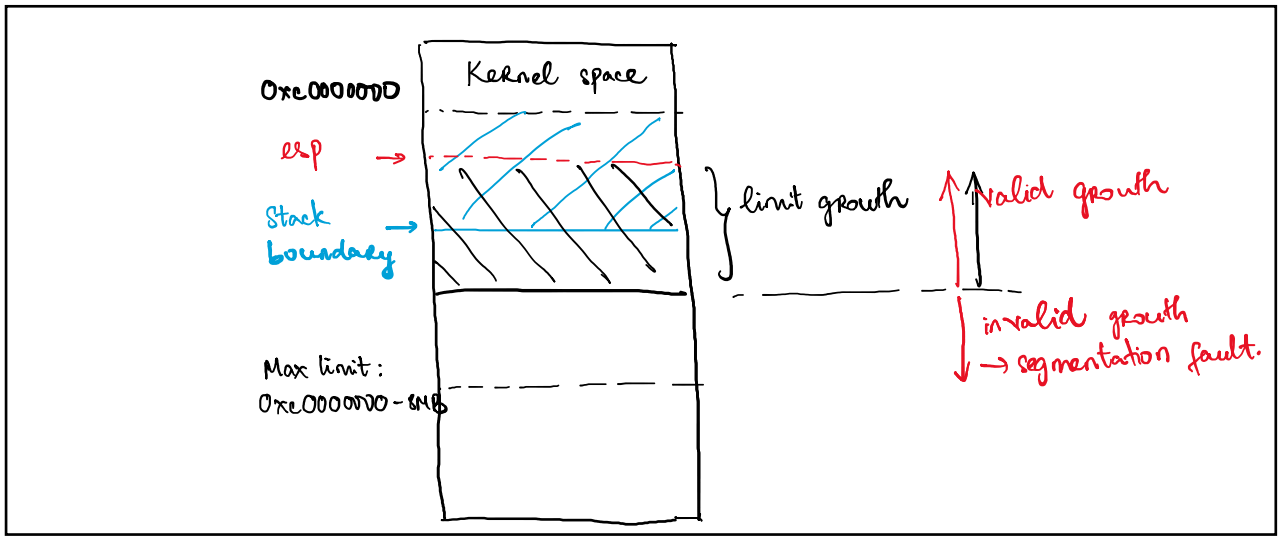
- **Requirement idea:** The main functionality is to make the stack expandable, which means make it available to increase the size of the stack.

- **Implementation of original PINTOS & Problem Identification:** In the current PINTOS, user stack is fixed with size of 1 page (4KB). Thus, when the process exceeds that boundary (grows larger than 4KB), a page fault would occur. Indeed, if the process needs more stack space, Operating System should allow it to expand the stack space; however, its concept is not implemented in the original PINTOS. Hence, after implementing, user stack could allow additional pages if the process needs.

- **Necessity for requirement:** The necessity is to let the stack expandable but still limit the space growth. If a process accesses the address that is outside the stack and handles by the page fault handler, it is available to expand the stack. Particularly, the access address should be smaller this bound (stack pointer (*esp*) – 32). Indeed, the maximum size of stack is only 8MB for not colliding with the heap segment.

- Data Structure: No data structure is added during this part
- Algorithm:
  In addition, this implementation is to support page fault handler on the edgy case that some user stack exceed the initial bound of 4KB. When it comes to this scenario, page fault would be occurred as usual, but we could resolve by implementing page fault handler to expand the stack. Therefore, when a stack is expanded, the memory access is within 32 byte of stack top. To be more specific, a legitimate stack access is that an application generates a stack access within 32 byte of stack top. If it grows over that boundary, it would be considered as the illegal access and thus generating a page fault.

  The diagram would illustrate the above interpretation:



  - Modify page_fault(): If the stack pointer (esp) is close to the supplemental page table (spt), we could initialize zero page and load inside the memory. Before performing it, it is vital to check through some certain conditions related to esp, PHYS_BASE, and the maximum size of stack. Thus, in page fault handler, the stack could grow afterwards.

e. File Memory Mapping

- **Requirement idea:** The requirement of this section is to make the open files accessible via direct memory access, which is to map them. If wanted to store data, it is necessary to write to the file. In addition, if wanted to read data, the procedure is to come to the file resided on the disk. Therefore, after implementing memory mapped file, the objective is to map the location within the virtual address space where application wants the system to bind a given file. In addition, storing something in the memory address space would directly reflect to the file content through file memory mapping procedure.

- **Implementation of original PINTOS & Problem Identification:** In the current PINTOS, memory pages are linked to stack or heap segment but there are some cases the link should be data segment in files. Since it is an executable file, the code segment must be linked to the file. Indeed, the anonymous pages, the page not resided on the disk, must be stored in the disk area through swap partition. However, when it comes to the file-backed pages, it is unnecessary to perform the procedure as the anonymous page since its file information is located on the disk. Hence, in the naïve PINTOS, it does not have such these implementations. Thus, it is essential to have system call "*mmap*" and "*munmap*" to resolve the above issue for helping a system manage a file loaded to virtual address region.

- **Necessity for requirement:**
  The necessity for implementation is: Firstly, to keep track of the open files of a process, we could initialize a data structure under struct file. The main section of this implementation is to design two system call *mmap()* and *munmap(),* where *mmap()* would populate the supplemental page table. Importantly, we would have to initialize a data structure to keep track those mappings. In addition, if there are some cases such as lazy loading, write-back if dirty bit, eviction if physical memory has no free space, it is required to have mmap's ed page.

- Data Structure:
  - **struct file* mmap_file{}:** The data structure is used to keep track the open file during the memory mapped file procedure, containing information from mapped files. Under this structure, we would create some *mapping id*, mapping *file* object, *mmap_file* as the list element, *page_list* as a list structure associated with given mapping.
  - **Modify struct thread{}:** To manage memory mapped file, we initialize a list structure to contain the opened files as *mmap_file* structure inside *mmf_list.*
- Algorithm
  When it comes to the requirement for two system calls (mmap, munmap), there are some notices that: When the process exits, all the mappings of the process are implicitly unmapped afterwards. When a mapping is unmapped, the pages should be written back to the file, During the munmap() system call, that pages should be removed through the process's virtual page. Regardless of the file is closed are deleted, mapping is still valid (when the mapping is created) until it gets unmapped. If there are more than one process mapped to the same file, it is unnecessary to let them view consistently.
  - **Add system call mmap():** This function is to load the file data into memory by demand paging. Indeed, when page fault occurs, it loads associated data block from disk to memory. In addition, *mmap()*'s ed page should be swapped out to its original location in the file but not to swap partition. When it comes to fragmented page, it is allowed to initialize a zero page. To implement as the code level, when receiving a request to map a file, it is vital that check the file within an appropriate file. Next, it would create an *id* under the structure of *mmap_file* and thus adding the *mmf_list* in the thread structure. Afterwards, this function would return the *id* where contains a unique *id* within a process to identify a mapped file. Otherwise, this function would fail if the file size is 0, address

passed in is not a page-aligned or in use or 0, also *STDIN* and *STDOUT* could not be mapped.
    - **Add system call munmap():** This function is to unmap the mapping inside *mmf_list*. When receiving the unmapping request, *mmf_list* would search for the corresponding file under the *mmf_file* structure. Then, it would deallocate that connection and delete them to the process of memory unmapping. Finally, it would close the file if the process is finished.
    - **Modify process_exit():** Release the *spt_entry* at the end of mapping procedure.
    - **Modify handle_mm_fault():** If the status of the page is *PAGE_FILE*, it performs the same as *PAGE_FRAME*, which is load the data from the disk to memory.

f. Swap Table

- **Requirement idea:** This section is the continuous part from the Frame Table, where we would use some defined data structure at that section. When implementing Swap Table, generally, when the physical frames (*frame_table*) is run out of the free frames, it is required to evict a page as a victim and put the copy on the swap disk (if needed) to get the free frame. In addition, when a page fault handler finds a page that is not in the memory yet resided in the swap disk, it is essential to allocate a new frame and move it into the memory.

- **Implementation of original PINTOS & Problem Identification:** In the current PINTOS, there is no implementation to evict the physical frames, thus provoking swapping is not useable. In addition, if there is no free frame, page allocation would return error. Indeed, there are two types of pages such as *anonymous page* and *file-backed page*. Regarding the anonymous page, as there is no file corresponded with it on the disk, creating a swap area and save the data toward it are the essential improvement when swapping out. Therefore, the swap area in the disk is used to save the file. In terms of the file-backed page, its file characteristic is in the file. Therefore, when performed to swap out, it is necessary to modify the changing data directly to corresponding file.

- **Necessity for requirement:**
  To recall, the swapping mechanism should be as followed. When the pages belong to the data or stack segment, it is required to store the victim pages in the swap space. In addition, those swap-out pages should be reloaded into memory by demand paging when coming to the memory.

- Data Structure:
    - **struct bitmap *swap_table:** Initialize the method to keep track whether a page has been swapped or not and the location that a page has been stored in swap disk. Therefore, this data structure as a *bitmap* is used to manage the swapping area.
    - **Modify spt_entry{}:** Under the structure of *spt_entry*, to perform swapping more efficiently, it is essential to add a field for *swap_table* as it would decide which page to be swapped out or in while iterating through the frames.
- Algorithm
    - **Add init_swap_table() function:** This function is to initialize the *swap_table* under the *bitmap* structure. To briefly draft the idea, firstly, it is required to obtain the swap block size via *block_size()* and divide the size by the number of pages per sector to obtain the total number of pages that could be inside the swap block. Simultaneously, using the *block_get_role()* is to return the block device fulling the given role. Finally, we would create *bitmap* through *bitmap_create()* to initialize the number of pages in the *swap_table*.
    - **Function to select victim page and swap-out page:** This function provokes the eviction policy by using the previous data structure in the *frame_table*. The main purpose of using *frame_table* is to find a victim frame for perform swapping by using the eviction policy.

Based on that policy, it is reasonable to choose the suitable page replacement policy, especially clock algorithm. Then, we could select a frame to swap out from the frame table and send the frame to swap disk to prevent some changes to frame. Afterwards, update the page table if necessary.

- **Add swap_out() function:** To recall the procedure of the swapping out, it is the following section after selecting a victim frame. If the page is dirty, we must write to the disk *(if needed)*. Indeed, when writing to the disk OS has to select the location on the disk where the page has to go through. For instance, if it is the anonymous page, it should go inside the swapping area. Otherwise, the file-backed pages are going to the file on disk. Indeed, it is implemented through the procedure of finding an appropriate empty space in the *bitmap* and writing the data to the corresponding space.
- **Add swap_in() function:** This function is to check the index of the page to be read from the *spt_entry* and load the swap data into physical memory. Indeed, the main function is to serve for the anonymous page. When it comes to the file-backed page, the process of finding *swap_table* is not used as the original idea of *swap_in()* function. Instead, it would perform to read the file, which means *swap_in()* is not used to serve the file-backed page.
- **Modify handle_mm_fault():** If the status of the frame as *PAGE_ZERO* (anonymous page), it is required to insert the *swap_in()* function in this scenario.

g. On Process Termination

- **Requirement idea:** The idea of this section is to deallocate all the using resources when a process is terminated. Particularly, it is necessary to destroy the 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 files after writing the dirty memory mapped pages from the memory back to disk if the dirty bit is notified.

- **Implementation of original PINTOS & Problem Identification:** In the current PINTOS, the process of reading data block from the disk to memory is that: the data would initially bring from the kernel section and then it would copy the data block from the kernel to the user space. However, while loading the disk to memory, OS sometimes swaps out associated user buffer and thus the process from copying data from kernel to user space is not occurred. To resolve it, it should generate the page fault and does the swap-out procedure. Thus, it might have to allocate a new frame from a free page list.
  However, before generated a page fault to prevent memory leaks, it did grab the driver lock. In addition, while handling page fault, it does acquire the driver lock once again after selecting a victim page and write a victim page to swap space. Therefore, it would occur the *deadlock* due to double locking on driver lock in the naïve PINTOS. Therefore, to resolve this issue, if there are any holding lock at the termination time, it is essential to release them to avoid deadlock occurrence.

- **Necessity for requirement:**

- Data Structure: Add a list of lock to manage the locks holding by one process.
  - **Modify struct thread {}:** Under the thread structure, create a list of lock as *lock_list*
  - **struct lock {}:** Add a *list_elem* for *lock_list* for management purpose.
- Algorithm
  To prevent the deadlock, the objective is to prevent evicting pages accessed during the system call and also define a pinning flag in each physical page. Regarding each system call implementation, it is required to find the virtual page and pin the associated physical frame. Hence, after it

returns or its system call handlers return, the page could be unpinned. In addition, it is necessary to aware that victim page should not be chosen to pin while swapping.

- **Modify process_exit():** As mentioned previously, when a process is terminated, it is required to free all element in the mapping list (*mmf_list*), deallocate all the entry of supplemental page table and *frame_table*, and remove all swap files of anonymous pages. If there is a notice of dirty page existence, it is required to write those pages back into the file in the disk. Finally, to avoid the deadlock, it is required to release all the locks from *lock_list*.