# Project 3. Virtual Memory

## Score

Total: 35 points

## Requirements

※ Implement on empty vm directory. But files in other directories need to be modified also.
※ You will build this assignment on top of the last one. Test programs from project 2 should also work with project 3. You should take care to fix any bugs in your project 2 submission before you start work on project 3, because those bugs will most likely cause the same problems in project 3. (We recommend including "alarm clock" in project 1 because it is helpful for project 3 (not essential)).
※ The purpose of this project is focused on how many user programs are executed simultaneously in efficient way. Therefore, you should design data structures of each table and managing functions carefully.
※ For each table, you need to decide whether it is declared globally or locally (for each thread).

### 1. Frame Table (5 points)

: Implement a frame table to manage frames effectively.

- Current implementation: The size of page and frame is 4096 bytes. The management of page tables is implemented in "userprog/pagedir.c", and the allocation/deallocation of page tables are implemented in "threads/palloc.c".
- **New implementation (5 points):** Create a frame table to manage information of each frame.
- Expected entry: frame number, thread id, possibility of allocation
- Main functions
  - ✓ Allocate/deallocate frames
  - ✓ Choose a victim which returns occupying frames when free frame doesn't exist.
  - ✓ Search frames used by user process (thread)
  - Frames used by user pages must be allocated by "user pool".
  - Modify process loading (the loop in "load_segment()") in "userprog/process.c" for managing a frame table.

### 2. Lazy Loading (5 points)

: Implement "lazy loading" for loading page to memory.

- Current implementation: Executable codes needed for process start are directly loaded in memory. When page fault occurs, a program execution always stop by considering this situation as an invalid access error.

- **New implementation (5 points):** Only a stack setup part is loaded during loading procedure for memory allocating when a process starts. (Other parts are not loaded in the memory. Just pages are allocated.) When a page fault is occurred from an allocated page, this page is loaded on memory. A page fault handler should resume a process operation when this procedure is ended.
- Page fault handler modifies the page_fault() in "threads/exception.c".
- For page fault handler, when a situation which needs I/O (lazy loading) and a situation which doesn't need I/O (wrong memory access) occur simultaneously, the later situation must be handled first without waiting the earlier situation.

## 3. Supplemental Page Table (5 points)

: Implement S-page table which have more functions than the previous page table.

- **S-page table (5 points):** Implement S-page table and a managing function for S-page table. Lazy loading, file memory mapping, swap table must work normally.
- Expected entry: page number, possibility of frame allocation, frame number, possibility of swap out, ...
- Main functions
    - ✓ Allocate frames for each page by lazy loading.
    - ✓ Store the modified data in a frame into a file or a swap disk.
    - ✓ When page is deleted, the related entries should also be deleted by finding them in frame table, swap table, and etc.
- We recommend using a Hash table.
- A page fault handler in "threads/exception.c" should refer an S-page table.

## 4. Stack Growth (5 points)

: Make it available to increase a size of a stack.

- Current implementation: The size is fixed as 1 page.
- **New implementation (5 points):** First, check whether a page fault handler needs a stack growth or not. If it needs, get a stack pointer from an interrupt frame (Calculation of the number of necessary pages is needed). Load pages, and modify the information of a frame table and S-page table.
- Decide early the maximum size of a stack by considering stack/heap collision.

## 5. File Memory Mapping (5 points)

: Make it available for files to be mapped with memories.

- **File memory mapping (5 points)**: Create a file mapping table which manages a relationship between files and pages, and implement mapid_t mmap(int fd, void *addr) and void munmap(mapid_t mapping) system call.
- Implement mmap by lazy loading. When it is dirty (write), conduct "write back" on a disk to prevent a lack of memory.
- When the size of a file and a page size are not same in mmap, fill the remaining bit as 0, and in write

back case, ignore this part.

- A call to mmap may fail if the file open as fd has a length of zero bytes. It must fail if addr is not page-aligned or if the range of pages mapped overlaps any existing set of mapped pages, including the stack or pages mapped at executable load time. It must also fail if addr is 0, because some Pintos code assumes virtual page 0 is not mapped. Finally, file descriptors 0 and 1, representing console input and output, are not mappable.

## 6. Swap Table (5 points)

: Implement a swap table to handle efficiently the case of when free frame doesn't exist.

- **Swap table (5 points):** Create a swap disk, and implement a swap table to manage the swap disk.
- Swap out: Copying a selected evicting page into swap disk to get free frames.
- Swap in: Allocating a page in swap disk into a new frame when page fault occurs.
- Use read/write functions based on a sector implemented in the "devices/block.h, .c" file.
- We recommend using a Bitmap structure. ("lib/kernel/bitmap.h, .c").
- Swap disk need to be created additionally. When "pintos-mkdisk swap.disk n" is executed in "vm/build", a partition named "swap.disk" with n MB is created.

## 7. On Process Termination (5 points)

: Make it available to deallocate all using resources when a process is terminated.

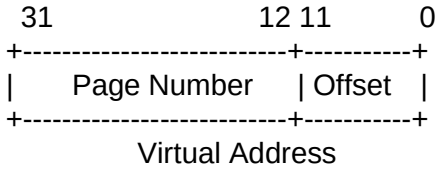
- **On process termination (5 points):** When a process is terminated, delete related contents in S-page table, frame table, and swap table. Close all related files (Dirty page needs to be written back)

# Background

## 1. Memory Terminology

**Pages**

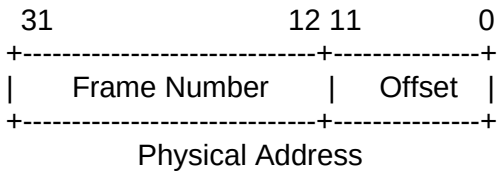
Page or virtual page is a continuous region of virtual memory 4,096 bytes (the page size) in length. A page must start on a virtual address evenly divisible by the page size. A 32-bit virtual address can be divided into a 20-bit page number and a 12-bit page offset (a location in page, 2^12 = 4,096).




Each process has an independent set of user (virtual) pages, which are those pages below virtual address PHYS_BASE (0xc00000000, 3GB). The set of kernel pages is global regardless of what thread or process is active. The kernel may access both user and kernel pages, but a user process may access only its own user pages.

**Frames**

A frame, a physical frame or a page frame is a continuous region of physical memory. Like pages, frames must be page-size and page-aligned. Thus, a 32-bit physical address can be divided into a 20-bit frame number and a 12-bit frame offset.




To directly access memory at a physical address, Pintos works around this by mapping kernel virtual memory directly to physical memory (different from 80x86). The first page of kernel virtual memory is mapped to the first frame of physical memory, the second page to the second frame, and so on. Thus, frames can be accessed through kernel virtual memory.

**Page Tables**

A page table is a data structure that the CPU uses to translate a virtual address to a physical address, that is, from a page to a frame. The page table format is dictated by the 80x86 architecture. Pintos provides page table management code in 'pagedir.c'. The diagram below illustrates the relationship between pages and frames. The virtual address, on the left, consists of a page number and an offset. The page table translates the page number into a frame number, which is combined with the unmodified offset to obtain the physical address, on the right.

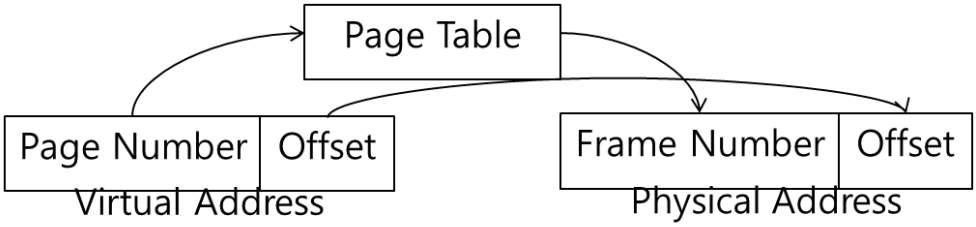



**Figure 1. A relationship among Virtual address, physical address, page table**

**Swap slots**

A swap slot is a continuous, page-size region of disk space in the swap partition. Although hardware limitations dictating the placement of slots are looser than for pages and frames, swap slots should be page-aligned because there is no downside in doing so.

## 2. Resource Management

**Managing the Supplemental Page Table**

The supplemental page table supplements the page table with additional data about each page. The supplemental page table is used for at least two purposes. Most importantly, on a page fault, the kernel looks up the virtual page that faulted in the supplemental page table to find out what data should be there. Second, the kernel consults the supplemental page table when a process terminates, to decide what resources to free.

The most important user of the supplemental page table is the page fault handler. In project 2, a page fault always indicated a bug in the kernel or a user program. In project 3, this is no longer true. Now, a page fault might only indicate that the page must be brought in from a file or swap. You will have to implement a more sophisticated page fault handler to handle these cases. Your page fault handler, which you should implement by modifying page_fault() in 'userprog/exception.c', needs to do roughly the following:

- Locate the page that faulted in the supplemental page table. If the memory reference is valid, use the supplemental page table entry to locate the data that goes in the page, which might be in the file system, or in a swap slot, or it might simply be an all-zero page. If you implement sharing, the page's data might even already be in a page frame, but not in the page table. If the supplemental page table indicates that the user process should not expect any data at the address it was trying to access, or if the page lies within kernel virtual memory, or if the access is an attempt to write to a read-only page, then the access is invalid. Any invalid access terminates the process and thereby frees all of its resources.
- Obtain a frame to store the page. If you implement sharing, the data you need may already be in a frame, in which case you must be able to locate that frame.
- Fetch the data into the frame, by reading it from the file system or swap, zeroing it, etc. If you implement sharing, the page you need may already be in a frame, in which case no action is necessary in this step.

- Point the page table entry for the faulting virtual address. (“userprog/pagedir.c”)

**Managing the Frame Table**

The frame table contains one entry for each frame that contains a user page. The frames used for user pages should be obtained from the “user pool,” by calling palloc_get_page(PAL_USER). You must use PAL_USER to avoid allocating from the “kernel pool,” which could cause some test cases to fail unexpectedly. If you modify ‘palloc.c’ as part of your frame table implementation, be sure to retain the distinction between the two pools.

The most important operation on the frame table is obtaining an unused frame. This is easy when a frame is free. When none is free, a frame must be made free by evicting some page from its frame.

If no frame can be evicted without allocating a swap slot, but swap is full, panic the kernel. Real OSes apply a wide range of policies to recover from or prevent such situations, but these policies are beyond the scope of this project.

The process of eviction comprises roughly the following steps:

- Choose a frame to evict, using your page replacement algorithm. The “accessed” and “dirty” bits in the page table, described below, will come in handy.
- Remove references to the frame from any page table that refers to it. Unless you have implemented sharing, only a single page should refer to a frame at any given time.
- If necessary, write the page to the file system or to swap.

**Managing the Swap Table**

The swap table tracks in-use and free swap slots. It should allow picking an unused swap slot for evicting a page from its frame to the swap partition. It should allow freeing a swap slot when its page is read back or the process whose page was swapped is terminated.

You may use the BLOCK_SWAP block device for swapping, obtaining the struct block that represents it by calling block_get_role(). From the ‘vm/build’ directory, use the command pintos-mkdisk swap.dsk --swap-size=n to create a disk named ‘swap.dsk’ that contains a n-MB swap partition. Afterward, ‘swap.dsk’ will automatically be attached as an extra disk when you run pintos. Alternatively, you can tell pintos to use a temporary n-MB swap disk for a single run with ‘--swap-size=n’. Swap slots should be allocated lazily, that is, only when they are actually required by eviction. Reading data pages from the executable and writing them to swap immediately at process startup is not lazy.

**Managing Memory Mapped Files**

```
#include <stdio.h>
#include <syscall.h>
int main (int argc UNUSED, char *argv[])
{
  void *data = (void *) 0x10000000;        /* Address at which to map. */

  int fd = open (argv[1]);                 /* Open file. */
```

```
    mapid_t map = mmap (fd, data);              /* Map file. */
    write (1, data, filesize (fd));           /* Write file to console. */
    munmap (map);                                   /* Unmap file (optional). */
    return 0;
}
```

Suppose file ‘foo’ is 0x1000 bytes (4 kB, or one page) long. If ‘foo’ is mapped into memory starting at address 0x5000, then any memory accesses to locations 0x5000. . .0x5fff will access the corresponding bytes of ‘foo’. Above is a program that uses mmap to print a file to the console. It opens the file specified on the command line, maps it at virtual address 0x10000000, writes the mapped data to the console (fd 1), and unmaps the file.