Project 3: Virtual Memory

To complete this task you must

- understand memory & virtual memory (from CS 341 text book)
- refer to PintDoc Chapter 4 that provides instructions and expectations for this project.
- understand project specification (including Appendix A.6, A.7 and A.8)
- understand the important parts of source code (`process.c: load_segment()`, `exception.c: page_fault()`)

Through this project you will be adding virtual memory implementation to Pintos. Until now, Pintos loads the entire data, code and stack segments into memory before executing a program (refer to `load()` function in `process.c`).

VM01: Implement the frame table, supplemental page table, and page fault handler (in this task you need not implement swapping. if you run out of frames, fail the allocator or panic the kernel).

A frame table keeps track of all the frames of physical memory used by the user processes. You can implement either by modifying the current frame allocator `palloc_get_page(PAL_USER)` or you can introduce your own frame allocator on top of `palloc_get_page(PAL_USER)` without modifying it (see `init.c` and `palloc.c` to understand how they work). It is not necessary to use hash table. You can use any of the data structures, arrays, lists, bitmaps, and hash tables that are supported by Pintos. You have to implement lazy loading of pages. In the beginning nothing is loaded, only the stack is setup. During process execution, a page fault occurs and page fault handler will check where the expected page is, and loads the corresponding page after looking at the supplemental page table. It is recommended to use hash table for supplemental page table.

After this implementation, the kernel should pass all the test cases of project 2, but only some of the robustness tests.

VM02: Implement stack growth and memory-mapped files.

So far Pintos allowed user stack to be of fixed size of 1 page, i.e. 4KB. After completion of this task the user stack can be allocated additional pages as required. If the user program exceeds the currently allocated stack size then a page fault will occur. In page fault handler, you need to determine whether the page fault is for lazy load or stack growth. You also need to impose an absolute limit on stack size, `STACK_SIZE`.

For memory-mapped files, you can use `struct file*` to keep track of the open files of a process. You need to design two new system calls: `mapid_t mmap(fd,addr)` and `void munmap(mapid_t)`. The `mmap()` system call also populates the supplemental page table. You will also need to design a data structure to keep track of these mappings. The memory mapped pages are required to be loaded lazily, written back only if dirty, subject to eviction if physical memory is deficient.

VM03: Implement swap table and frame eviction, and page reclamation on process exit.

When free frames are not available then a page is required to be evicted from its frame and a copy is put into swap disk, if necessary, to get a free frame i.e. swapped out. When the page fault handler finds a page is not in the memory but in swap disk, it allocates a new frame and move it to memory i.e. swaps

in. You need to implement a method to keep track of whether a page has been swapped and in which part of swap disk a page has been stored if so. You can select your own data structure to implement the swap table.

Implement an eviction (page replacement) algorithm to select a frame to swap out from frame table and to send to swap disk. Update the page tables (supplemental page table and hardware page table) via `pagedir_*` function as required. Finally on process termination, remove the supplemental page table, free the frames and corresponding entries in the frame table, free the swap slots and delete the corresponding entries in the swap table, close all files and for memory mapped files write the dirty pages from memory to the file disk.

While implementing, take care of synchronization issues. Page fault handling from multiple processes must be allowed in parallel.