Gueho Choi

CSCI 350 – PintOS Project 3

December 7, 2014

Design Document

I. What happens during a page-fault?
   - During the page fault, I first check if it is a user virtual address using the function in vaddr.h along with checking that if the virtual address is not present. If the condition is not satisfied, It will generate page fault exception and kill the process. If the condition is satisfied, it will search into supplemental page table for the fault address, if there is supplement page table entry associated with that fault address, then it will load the page from the supplemental page table structure. If entry not present, it will check if the fault address is asking for a stack growth, then grow the stack if appropriate.
   - Struct spage_entry
     - Void *user_va: user virtual address the entry is holding
     - Int spage_type: the page type dictates in which functionality to run
     - Bool Is_pinned: pin the page to not evict
     - Bool is_loaded: the page is loaded or not
     - Bool writable: if the page is writable or not
     - Struct file *file: lazy file loading
     - Size_t offset: lazy file loading
     - Size_t read_bytes: lazy file loading, precalculated from load_segment
     - Size_t zero_bytes: lazy file loading, pre-calculated from load_segment
     - Sizet swap_index: index to be used to identify swap
     - Struct hash_elem elem: hashable

II. How did you implement eviction & pinning?
   - frame_evict will be called when it cannot retrieve page using palloc. Then the eviction will be repeatedly check for the swappable frame in the frame table. That is, if there is frame table entry that page was not pinned, it will check if the page was accessed, and if it is accessed before, it will set it so that next time coming back, it can be accessed. Then it will check page's dirtiness if not accessed before. If the page is dirty, it will swap out. Then return the freed page.
   - Frame_table_entry
     - Void *frame: the pointer to frame
     - Struct spge_entry: supplemental page table entry that frame is associated with
     - Struct thread *t: thread that frame was being used
     - Struct list_elem: listable
   - Swap is done by frame table entry and page_dir.
   - Swap.h contains struct bitmap swap_bitmap to keep track of free swap slots
   - Swap.h contains struct blok *swap as suggested in the project slides

III. How do you implement memory mapped files?
   a. System call handler code

- On mmap system call, it takes file descriptor, and looks for the file in the file descriptor table in struct thread. Then the old_file is transferred empty slot in mmaptable by creating new struct spage_entry, which serves as supplemental page that contains all info about file (struct file, readbytes, zerobytes, offset, address, and mmapfd) readbytes and zerobytes are calculated in the same way as it was calculated in load_segment() in process.c

b. Data structures to keep track of them

- Mmaptable in struct thread in thread.h keeps track of all the mmap files mapped, just like file descriptor tables.
- Struct spage_entry *mmaptable[128]: supplemental page table uses mmap just like regular files, but it also contains type to identify if it is mmap file or not.
- Struct spage_entry
  - struct file
  - readbytes
  - zerobytes
  - offset
  - address
  - mmapfd

IV. How did you implement page sharing

- I did not implement page sharing functionality