+---------------------------+

| CS 521 |

| PROJECT 3: VIRTUAL MEMORY |

| DESIGN DOCUMENT |

+---------------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Sai Nikhila Muthyala <sainikhi@buffalo.edu>

Harika Reddy Kumbum <harikare@buffalo.edu >

Tejaswini Reddy Boda <tboda@buffalo.edu>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while

>> preparing your submission, other than the Pintos documentation, course

>> text, lecture notes, and course staff.

PAGE TABLE MANAGEMENT

=====================

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

In thread.h

struct hash suppl\_pt; /\* Supplemental page table for the thread \*/

In page.h

struct suppl\_pt\_entry {

struct hash\_elem elem;

uint32\_t entry\_type; /\* To store if it is a file/swap/mmap \*/

void\* user\_virtual\_address; /\* User virtual address of physical memory

page\*/

bool loaded; /\* To store if the page table entry is loaded into physical memory or successfully swapped out or written to a file \*/

bool accessed; /\* To check if kernel is currently accessing page table entry \*/

struct file \*file\_src; /\* The file from which the page is read \*/

size\_t offset; /\* The offset from which page is read \*/

size\_t swap\_partition\_index; /\* The destination swap partition index if the entry\_type == “swap” \*/

};

In frame.h

struct lock ft\_lock; /\* Lock for the frame table \*/

struct list frame\_table;

struct ft\_entry {

struct suppl\_pt\_entry \*sptentry; /\* Entry using the physical frame \*/

struct thread \*t; /\* Thread using the physical frame \*/

void \*frame; /\* Pointer to physical memory frame \*/

struct list\_elem elem;

};

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for locating the frame,

>> if any, that contains the data of a given page.

If a page fault occurs when user program tries to access a page to which no frame is mapped, we look up the supplemental page table (which is a hash table) entry using user virtual address as the key. Now a new frame is allocated to this entry if it is available, else one of the frames is swapped out and then we swap in the new frame. Then the virtual address is mapped to physical frame address.

If the user virtual address is a swap slot address, then the data is directly read from the swap partition. On the other hand, if it is a file access, the data is read as a byte array (memory mapped i/o).

We set the flag ’loaded’ in page table entry.

>> A3: How does your code coordinate accessed and dirty bits between

>> kernel and user virtual addresses that alias a single frame, or

>> alternatively how do you avoid the issue?

The issue is avoided by using user virtual address to access user stack and to look up the supplemental page table.

This way there is no need for the kernel to use the kernel virtual address.

---- SYNCHRONIZATION ----

>> A4: When two user processes both need a new frame at the same time,

>> how are races avoided?

The races are avoided by holding a lock for adding or evicting frames to/from a frame table.

---- RATIONALE ----

>> A5: Why did you choose the data structure(s) that you did for

>> representing virtual-to-physical mappings?

struct suppl\_pt\_entry \*sptentry is included in struct ft\_entry as there is a need to unset the ‘loaded’ flag and invalidate the user virtual address for the respective frame’s thread for every frame eviction.

void\* user\_virtual\_address is included in struct suppl\_pt\_entry to prevent the problem of aliasing by allowing the kernel to access data using only user virtual address.

All other data structures declared support every attribute of supplementary page table entry and frame table entry.

PAGING TO AND FROM DISK

=======================

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

In swap.h

struct lock swap\_lock; /\* Lock for swap \*/

struct block swap\_block; /\* Block to store pages \*/

---- ALGORITHMS ----

>> B2: When a frame is required but none is free, some frame must be

>> evicted. Describe your code for choosing a frame to evict.

Using LRU clock algorithm i.e. traversing through the frames till we find the first frame with 0 access bit. We clear the access bit of each frame (whose access bit is 1) as we move.

>> B3: When a process P obtains a frame that was previously used by a

>> process Q, how do you adjust the page table (and any other data

>> structures) to reflect the frame Q no longer has?

To remove Q’s reference to the frame which was previously used by it, we free the page from Q’s thread structure using pagedir\_clear\_page().

>> B4: Explain your heuristic for deciding whether a page fault for an

>> invalid virtual address should cause the stack to be extended into

>> the page that faulted.

If the present user virtual address causing page fault falls between PHY\_BASE and esp-32 (to handle assembly push instruction that decrements stack pointer by 32 bytes at once), then the access is considered valid and a new page is added to the stack and registered in the page table entry. Any access beyond this is considered invalid and kills user program.

---- SYNCHRONIZATION ----

>> B5: Explain the basics of your VM synchronization design. In

>> particular, explain how it prevents deadlock. (Refer to the

>> textbook for an explanation of the necessary conditions for

>> deadlock.)

When a page fault occurs that accesses user virtual memory, we set the ‘accessed’ flag which is an indication that it cannot be evicted by an eviction algorithm. When the required user data is accessed by the kernel, ‘accessed’ flag is cleared.

>> B6: A page fault in process P can cause another process Q's frame

>> to be evicted. How do you ensure that Q cannot access or modify

>> the page during the eviction process? How do you avoid a race

>> between P evicting Q's frame and Q faulting the page back in?

Locks are used for each supplemental page table entry. When Q tries to access a page during eviction, it acquires the lock. Since the lock is held by P, Q will get page fault and it will look up the supplemental page table entry and it waits till eviction is finished and lock is released.

>> B7: Suppose a page fault in process P causes a page to be read from

>> the file system or swap. How do you ensure that a second process Q

>> cannot interfere by e.g. attempting to evict the frame while it is

>> still being read in?

Each page table entry has an attribute which, when set to false when reading from file system or swap, will make the frame unevictable.

>> B8: Explain how you handle access to paged-out pages that occur

>> during system calls. Do you use page faults to bring in pages (as

>> in user programs), or do you have a mechanism for "locking" frames

>> into physical memory, or do you use some other design? How do you

>> gracefully handle attempted accesses to invalid virtual addresses?

When the pages are loaded, we lock the pages using ‘accessed’ flag. Since the eviction can still happen, we allow kernel to page fault in user pages as well.

When an attempt to access invalid virtual address is made, we kill the process using exit().

---- RATIONALE ----

>> B9: A single lock for the whole VM system would make

>> synchronization easy, but limit parallelism. On the other hand,

>> using many locks complicates synchronization and raises the

>> possibility for deadlock but allows for high parallelism. Explain

>> where your design falls along this continuum and why you chose to

>> design it this way.

We limit the use of locks to frame table addition and eviction and the rest of the accessing is handled by flags thus simplifying parallelism and preventing deadlocks.

MEMORY MAPPED FILES

===================

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

In thread.h :

struct list mmap\_list; // stores list if mmaped files for this thread.

In process.h :

struct mmap\_file { // individual entry in the mmap\_list of thread

struct suppl\_pt\_entry \*sptentry; /\* mmaped page in this thread \*/

struct list\_elem elem;

};

---- ALGORITHMS ----

>> C2: Describe how memory mapped files integrate into your virtual

>> memory subsystem. Explain how the page fault and eviction

>> processes differ between swap pages and other pages.

A hash table for each process is maintained which consists of memory mapped files. mmap loads the file into memory and munmap frees memory and checks if corresponding pages are dirty. The contents of dirty pages are written into the file else the pages are freed. Once the process exits, all the files are freed.

>> C3: Explain how you determine whether a new file mapping overlaps

>> any existing segment.

We determine overlaps by checking if the thread’s hash table has duplicate entries (the entries are the supplemental page table entries).

---- RATIONALE ----

>> C4: Mappings created with "mmap" have similar semantics to those of

>> data demand-paged from executables, except that "mmap" mappings are

>> written back to their original files, not to swap. This implies

>> that much of their implementation can be shared. Explain why your

>> implementation either does or does not share much of the code for

>> the two situations.

The implementation for executables and mmapped files is similar in loading the files. But in case of mmapped files, the dirty pages have to be written to their corresponding files.

SURVEY QUESTIONS

================

Answering these questions is optional, but it will help us improve the

course in future quarters. Feel free to tell us anything you

want--these questions are just to spur your thoughts. You may also

choose to respond anonymously in the course evaluations at the end of

the quarter.

>> In your opinion, was this assignment, or any one of the three problems

>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in

>> future quarters to help them solve the problems? Conversely, did you

>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist

>> students, either for future quarters or the remaining projects?

>> Any other comments?