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

| CS 140 |

| PROJECT 3: VIRTUAL MEMORY |

| DESIGN DOCUMENT |

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

---- GROUP ----

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

[haoweizh@buffalo.edu](mailto:haoweizh@buffalo.edu)

zwang75@buffalo.edu

htang6@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 thread{

struct list sup\_page\_table; /\* supplemental page table. \*/

struct lock sup\_page\_table\_lock; /\* supplemental page table lock. \*/

}

In page.h

struct sup\_page\_entry{

uint8\_t \*user\_page; /\* Address of user page. \*/

struct file \*file; /\* Opened file. \*/

off\_t ofs; /\* Offset in a page. \*/

bool writable; /\* Is file writable. \*/

uint32\_t read\_bytes; /\* Read bytes in a page. \*/

uint32\_t zero\_bytes; /\* Left bytes in a page. \*/

struct list\_elem elem;/\* List elem. \*/

};

---- ALGORITHMS ----

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

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

-First, using *page\_get\_page(uint32\_t \*pd, const void \*uaddr)* to check whether user address is mapped. If yes, load the page according to page table. If not, the page might be unloaded or not exist. To check whether this page exist, we use supplementary page table, which stores the information of this page according to correspond virtual address. If supplementary page table’s information shows that this page doesn’t exist, then process this error and terminate this process. If this page haven’t loaded, load it (lazy load).

>> 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?

-Processes can access data only through user virtual address, we cannot access data through kernel virtual address, and this design can avoid the issue.

---- SYNCHRONIZATION ----

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

>> how are races avoided?

-Frames are managed by a bitmap, 1 represents valid and 0 represents invalid. So once we call *frame\_get (enum palloc\_flags flags)*, kernel will access bitmap, when kernel wants to access bitmap, it should acquire a lock first, after kernel get the valid frames, the kernel release the lock. This design avoid races between processes.

---- RATIONALE ----

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

>> representing virtual-to-physical mappings?

-Lazy swapper will never swaps a page into memory unless that page will be needed. Original pintOS load all the pages before process being executed, now we have to load the pages which are needed. So we need many other information to find out what pages should be in the memory. So we design a supplemental page table. Base on basic page table, supplemental page table contains other information, include dirty bit, whether loaded, lock for accessing.

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.

struct list vm\_frames; //list of global virtual memory frames

struct vm\_frame {

void \*frame; //point to physical frame

tid\_t tid; //current thread id

uint32\_t \*pte; //point to page table

struct list\_elem elem;

bool cannot\_swap; //whether this frame can be swapped

};

---- 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.

-We want to choose clock algorithm to select a frame to evict. We just add a bool variable in vm\_frame called cannot\_swap. When first load this page, cannot\_swap is set to 1, iterating the whole vm\_frame list and swap the first page we encountered which cannot\_swap is 0, if cannot\_swap is 1, change the value to 0. That’s a simplest clock algorithm.

>> 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?

-By calling function *pagedir\_clear\_page (uint32\_t \*pd, void \*upage)* in process Q and we can remove this unused page from supplemental page table in Q.

>> 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.

-Generally speaking, OS will cause page fault if virtual address below the stack pointer. However, the 80x86 PUSH instruction checks access permissions before it adjusts the stack pointer, so even if we access esp – 32, it is also valid. So what we have to do is check whether virtual address maps address below esp – 32. If yes, then page fault.

---- 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.)

-Each process has a supplemental page table, it is used by the process which owns it and other processes which during eviction. So in order to avoid many processes modifying the same supplemental page table, we should add a lock in each thread. When a process wants to modify supplemental page table, this process acquires a lock. However, deadlock won’t happen because a process cannot modify many supplemental page table at the same time, a process just wants to find a page which can be swapped, so it won’t access many table, deadlock couldn’t happen.

>> 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?

-Each process has a lock, so when P during eviction, P must get its lock, so Q cannot modify or access supplemental page table during the eviction process because Q doesn’t have lock. So by using a lock, only one process can modify or access supplemental page table, that’s how we avoid a race between P and Q.

>> 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?

-Like question B2 said, I add a variable cannot\_swap in vm\_frame, when P is reading in a page, this variable should be set to true before operation. And if Q wants to evict this frame, it should check this page’s cannot\_swap first, if it is true, we can block process Q until P finish reading in the page.

>> 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?

-I use clock algorithm and each page has a variable cannot\_swap and it is initialized to 1. Every time kernel wants to evict a page, check cannot\_swap first, the kernel will page-out the first page it encountered whose cannot\_swap is false. If a virtual addresses is invalid, just terminate the user processes.

---- 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.

-I don’t want to use a global lock to limit parallelism, but because I don’t know how many processes we could have, so it is unrealistic by using many global locks. Perhaps the best way is every thread has a lock. By this way, only one process can modify one supplemental page table but it will not prevent other processes modifying other supplemental page table. So the whole system can keep appropriate parallelism.

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 mmfile.h

- struct mmfile{

Unsigned mapid; //The ID of this map

Int fd; //The ID of the file

Void \*head\_addr; //The starting address

Unsigned pg\_num; //Number of page allocated to that file

List\_elem elem; //form a list

};

In thread.h

Int mid; //which is use to allocate the mapid

List mmfile; // the lsit of mmfile

}

---- 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.

- Each process have a list of mmfile which will be initialized at starting time, The mmap function map the file to virtual memory while the mumap check whether the file page is dirty, if so write back to disk, and then delete the mmfile structure and free corresponding virtual pages. We evict swap page to specific swap slot while write back to file for mmfile page. And when we need page back, we need to look at swap slot for swap page while original file for mmfile.

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

>> any existing segment.

- Before any allocation, we first check weather demanding pages has been allocated, By using the head address and the size of file to calculate what chunks of pages is need, then we check both the original page table and our added page table to find these pages, if any of page is used then return -1.

---- 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.

- We acutually did plan to share part of the code between this to mapping since while allocating pages, they both need to find free page and which need to check page of mmpfile and swap page. While evicted, they both need to be write back to disk despite the different location. So the basic write and load function can share between them.

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?