+---------------------------+  
 | CS 451 |  
 | PROJECT 3: VIRTUAL MEMORY |  
 | DESIGN DOCUMENT |  
 +---------------------------+  
  
---- GROUP ----  
  
>> Fill in the names and email addresses of your group members.  
  
Hampton Terry hterry@uw.edu

Kevin Tang kevin.tang2@gmail.com

Matthew Dorsett madman2@cs.washington.edu  
  
---- PRELIMINARIES ----  
  
>> If you have any preliminary comments on your submission, notes for the  
>> TAs, or extra credit, please give them here.

Our swap table is implemented correctly, but the eviction algorithm seems to fail, resulting in many of the robustness tests failing. We have code that should be very close, but something is a little off and we simply did not have time to perfect it. Most of the work is done in frame.c, so take a look there and see how we handle evictions.  
  
>> 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.

**Information stored for each frame.**

struct frame

{

uint32\_t \*pte;

void \*uvaddr;

void \*frame;

tid\_t tid;

struct list\_elem frame\_elem;

};

struct list frame\_list; **List that stores all the frame structs**

struct lock frame\_lock; **Lock used to ensure mutual exclusion for accessing the frame table.**

**Types of supplemental page table pages**

enum suppl\_pte\_type{

SWAP = 001,

FILE = 002,

MMF = 004

};

**Information stored in the supplemental page table**

struct suppl\_pte{

uint8\_t \*vaddr; //user virtural address for page

enum suppl\_pte\_type type; //type of data stored in pte

struct hash\_elem elem; //to look up pte in suppl hash table

struct hash\_elem mmf\_elem; //to look up pte in mm\_files hash table

//data about PTE

struct file \*file;

off\_t file\_offset;

uint32\_t bytes\_read;

uint32\_t bytes\_zero;

size\_t swap\_index;

bool writable;

bool loaded;

//MMF stuff

int mm\_id;

void \*start\_addr;

int pg\_count;

};

**Used to store the hash elem for the supplemental page table**

in struct thread:

struct hash suppl\_page\_table; /\* Supplemental page table \*/

---- ALGORITHMS ----  
  
>> A2: In a few paragraphs, describe your code for locating the frame,  
>> if any, that contains the data of a given page.

The frame table stores frame in a list data structure. This is not used for efficient frame lookup, rather for implementing a clock eviction algorithm. Locating a frame is done by simply traversing the list and finding a the frame that has the frame that contains the the matching page address to what we want.   
  
>> 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?

When determining which frame to evict in the case where there is no space available for new frames, the clock algorithm checks the accessed and dirty bits of the page held by the frame first. If the accessed bit is false, then the frame is evicted. Furthermore, if the dirty bit is true, then in the case of memory-mapped files, the file is updated on disk before evicting, and in the case of a dirty file we put the page in swap.  
  
---- SYNCHRONIZATION ----  
  
>> A4: When two user processes both need a new frame at the same time,  
>> how are races avoided?

The frame table is a global list that is initialized when the system does initialization (init.c). New frames are added to the list using the built in list functions. Therefore the way we avoid race conditions is to lock the frame table as we do the adding and removing operations to the frame table. This ensures that no two threads are going to have races when trying to operate on the frame table.  
  
---- RATIONALE ----  
  
>> A5: Why did you choose the data structure(s) that you did for  
>> representing virtual-to-physical mappings?

We used a list for the frame table which seems sufficient to keep track of all frames being used. The reason why we chose a list rather than some other data structure is because of the eviction algorithm we intended to use. The linked list provided seems suitable for for the clock algorithm because you can simply walk through the list, update used bits, and choose the evicted page in that fashion.

The alternative design we thought of was to store the frames in an array. This would allow us to do an efficient random eviction algorithm that is not possible when using the list, however, there are other considerations such as having to grow the array when its capacity has been met. Because of this we felt like it was simpler just to implement the frame table as a list.  
  
 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.

/\* This is a bitmap that is used to find free pages on the swap disk \*/

struct bitmap \*swap\_map;

/\* This is the block that gives us access to the swap disk \*/

struct block \*swap\_device;  
  
---- 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 implemented a clock eviction algorithm for our frame list. There is a global “hand” that initially points to the first frame; when a frame needs to be evicted, it traverses the list, using pagedir\_is\_accessed() to test whether or not that data page held by the current frame has been accessed. If so, it uses pagedir\_set\_accessed() to set the accessed state of the page to false, and continues looping through the list until it finds a frame containing a page that has not been accessed since the last time the “hand” was there. This frame is evicted, and the hand increments to the next list element and waits for the eviction algorithm to be called again.  
  
>> 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?

When P obtains a frame previously used by another process, the swapping mechanism first updates the entry for that frame structure “held” by Q, indicating that it is in swap, and also storing where in swap that page of data is stored. If Q gets back control and wants to use that data, the page fault handler will call swap\_to\_mem to retrieve the page of data.  
  
>> 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.

There are three key ways to determine if something is a good candidate for being a stack access. Which can all be determined by examining the stack pointers passed into the program by the interrupt frame’s esp.

1. The stack pointer is below PHYS\_BASE.

This tells us that this is an access to user memory rather than kernel. The stack always grows downward from PHYS\_BASE therefore an access above this point could not be a valid stack access.

2. The pointer is above STACK\_LIMIT

STACK\_LIMIT represents the maximum amount the stack should be allowed to grow. Perhaps this can be set to some certain limit by some flag passed into pintos as it is starting, however, we did not do anything of the sort. We simply decided that we should impose an arbitrary 8MB limit to the stack.

3. The fault\_addr +32 >= esp.

The PUSHA instruction pushes 32 bytes to the stack and checks for valid access which is what is causing the page fault. Therefore access that are greater than 32 bytes should not be considered as candidates for page faults.  
  
---- 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.)

The frame table and swap table are both global, and the supplemental page tables are per-process. There is an internal lock that controls access to the frame table, ensuring that any modifications to the table are atomic. There is also a lock that makes the eviction process atomic, including all functions that modify the swap table. Both of these parts do not interact with each other in terms of lock acquiring, so there is no possible case of deadlock. We should also have some kind of lock on each supplemental page table, because those entries can be modified by other processes regardless of the fact that they are per-process.  
  
>> 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?

To prevent this race condition, we should have a lock for each supplemental page table. This way, the eviction mechanism would acquire the lock for the supplemental page table first, so that when the page fault occurred and the page fault handler looked up the supplemental page table entry, it would be forced to wait until the eviction completed.

>> 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?

If we were to implement this, we would have an “evictable” attribute in each frame table entry that would be set to false before every read from the file system or swap, and the eviction algorithm would check for this and honor it.  
  
>> 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?

Any paged-out page accessed by a system call is faulted in before the system call runs, just like with user programs. Accesses to invalid virtual addresses are pre-checked and rejected.  
  
---- 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.

If we only use a single lock for the entire VM system, it will be crippled by the lack of parallelism allowed. Likewise, too many locks result in far too many potential areas for deadlock, and the complexity become too much to manage. Our implemented solution (combined with proposed further additions) is a lock for the frame table, a lock for the swap table, and essentially a lock per process for the supplemental page tables. We initially did not feel it was necessary to lock the supplemental page tables, because each process should be able to manage its own, but the entries are modified by other processes when saving data during eviction, etc, so it became necessary.  
  
 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.

**page.h**

**/\*** enum to keep track of page type in a suppl\_pte \*/

**enum suppl\_pte\_type{**

**SWAP = 001,**

**FILE = 002,**

**MMF = 004**

**};**

**/\* Used to find all suppl\_pte for the memory mapped file Use the mm\_id field to look up this struct in the current threads mm\_files hash. The file starts at start\_addr, and ends at start\_addr + PGSIZE\*pg\_count. You can calculate the vaddr for each page, and use the suppl\_page\_table to look up the suppl\_pte for each vaddr \*/**

**struct mm\_file{**

**int mm\_id;**

**struct file \*file;**

**void \*start\_addr;**

**int pg\_count;**

**struct hash\_elem elem;**

**};**

**/\* Struct used to hold data for a supplement page table entry \*/**

**struct suppl\_pte{**

**uint8\_t \*vaddr; *//user virtural address for page***

**enum suppl\_pte\_type type; *//type of data stored in pte***

**struct hash\_elem elem; *//to look up pte in suppl hash table***

**struct hash\_elem mmf\_elem; *//to look up pte in mm\_files hash table***

***//data about PTE***

**struct file \*file;**

**off\_t file\_offset;**

**uint32\_t bytes\_read;**

**uint32\_t bytes\_zero;**

**size\_t swap\_index;**

**bool writable;**

**bool loaded;**

***//MMF stuff***

**int mm\_id;**

**void \*start\_addr;**

**int pg\_count;**

**};**

**thread.h**

**/\*** Added fields to thread struct \*/

**struct thread**

**{**

**/\*OTHER STUFF FROM PREVIOUS ASSIGNMENTS \*/**

**/\* NEW FIELDS: \*/**

**int next\_id; /\* counter for id for next mmap file \*/**

**struct hash suppl\_page\_table; */\* Supplemental page table \*/***

**struct hash mm\_files; */\* Stored active mem mapped files. mapps mmap\_id to***

***struct mm\_file \*/***

**};**

---- 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.

We store an ‘enum suppl\_pte\_type’ as one of the fields in our ‘struct suppl\_pte’. This type can be set as FILE, MMF, or SWAP. When the mmap system call is called, we add supplemental page table entries to the thread’s suppl\_page\_table, and set the type of the entry to ‘MMF’, as well as storing the rest of the needed information about the file in the suppl\_pte struct.

When a page\_fault occurs and we need to load the page, we check the thread’s supplemental page table and use the fault address to look up the suppl\_pte struct. If the type of the suppl\_pte is set to ‘MMF’, we know this page table entry is for a memory mapped file, and we can use the rest of the information stored in the suppl\_pte struct to load the page correctly.

In order to implement the munmap system call, we needed a way to map the mapid\_t to the set of suppl\_pte struct’s that reference the file. In order to do this, each thread has a hash table which maps mapid\_t to a mm\_file struct. This mm\_file struct stores a pointer to the file, the starting vaddr, and the number of pages. We can access all suppl\_pte for the memory mapped file by looking up the vaddrs in the thread’s suppl\_pte hash table. We can calculate the vaddr of each page with: (start\_vaddr + i \* PGSIZE) as you iterate i from 0 to pg\_count which is stored in the mm\_files struct. Once we have access to all the suppl\_pte structs, we can write any modifications to the file back to disk, and remove the suppl\_pte structs from the thread’s supplemental page table.

The eviction process differs slightly between swap pages and other types of pages, primarily in how it deals with the data in the page before evicting it. For a dirty, memory-mapped file page, we write the file back to disk and then place it in swap. For a dirty file page, we simply place it in swap. If the page is not dirty and is not a file, then it is part of the stack and needs to be placed in swap. If the page is a file and not dirty, then nothing needs to be done because that file can be re-loaded when needed from disk.

>> C3: Explain how you determine whether a new file mapping overlaps  
>> any existing segment.

In the mmap system call, we start at the user specified addr value and iterate through all the pages the file will need to make sure they are available. We do this by checking the following pages: (addr + i \* PGSIZE) with i starting at 0 and increasing until we’ve checked enough sequential pages to store the size of the file. (We can stop iterating when i\*PGSIZE > filesize).

For each page we check, we need to verify two things:

-There is not already an entry in the thread’s supplementary table for the page’s vaddr. We check this by looking up the vaddr in the threads supplementary page table, and make sure this lookup fails (meaning the entry does not exist yet).

-call pagedir\_exists(thread\_current()->pagedir, addr + i\*PGSIZE) to make sure there is not already an entry in the page’s pagedir.

If both these tests pass for the virtual address of all needed pages, we know there is enough room to memory map the file at the requested virtual address.

---- 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 use the same suppl\_pte struct to represent memory mapped pages, as well as pages from executables. This allows us to share code and re-use the similar fields, while being able to tell which kind of suppl\_pte struct you have by checking the enum type field.

Our implementation handles the two cases with mostly different code. We handle memory mapped files in the munmap system call. As we are unmapping each page in the memory mapped file, we check to see if any modifications were made, and write them back to the file on disk if needed.

Pages for executables are handled differently in our implementation. When a page for an executable needed to be swapped out, we use the bitmap to find a free space on the swap disk, and then copy the page to that region of the swap disk, and save the swap index in its suppl\_pte for the next time that page is accessed.

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?