 # CS 140 PROJECT 3: VIRTUAL MEMORY DESIGN DOCUMENT

---- GROUP ----

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

Chuyi Zhao zhaochy1@shanghaitech.edu.cn

Bowen Xu xubw@shanghaitech.edu.cn

---- 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.

* **Pintos Guide** by Stephen Tsung-Han Sher

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 `page.h`:

```
/* spt entry with information about page */

struct sup_page_table_entry {

  /* for hash */

  struct hash_elem hash_ele;   /* hash element */

  uint32_t* user_vaddr;      /* virtual address, as a key in hash
table */

  struct lock spt_lock;      /* lock to provide page operation e.g.
hash */

  struct frame_table_entry* frame;  /* frame */

};
```

in `frame.h`:

```
struct frame_table_entry {

  struct list_elem ele;          /* for list */

  uint32_t* frame;            /* ptr to page that currently
occupies it */

  struct thread* owner;          /* the thread who owns it */

  struct sup_page_table_entry* page;   /* ptr to supplemental page
entry */

// Maybe store information for memory mapped files here too?

};
```

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for accessing the data

>> stored in the SPT about a given page.

Here the concept of `page` is not really clear. So possible situations are listed below.

The `userprog` is initialized with one page. And we keep the so-called `supplemental page table` for the additional pages as the stack grows.

If `page` means a virtual address, then we may face page fault.(stack growth):

If the page is not in page directory, we can need to do the stack growth. I allocate a page of memory in `user_pool`, construct a frame table entry `struct frame_table_entry` and assign it to the page. Then use `page_install` to connect the two address.

If the the page is already in page directory, we can get `struct sup_page_table_entry*` by calling `spt_hash_lookup()`. This way we can find through the SPT hash table, getting the SPT entry of the given page address.

If `page` means a `spt_entry` or `struct sup_page_table_entry*`. In the SPT entry, all the data of the given page is accessible.

>> 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 kernel should only access user data through the user virtual address.

And we do not use kernel virtual address.

---- SYNCHRONIZATION ----

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

>> how are races avoided?

Only one process can enter page fault exception at a time.

The frame will be given first come first serve.

And some frame operations will be locked to avoid trouble.

---- RATIONALE ----

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

>> representing virtual-to-physical mappings?

SPT maps user virtual memory to kernel virtual memory (or swap, or memory mapped file) while Frame tables maps the kernel virtual memory to physical memory.

The struct of each entry contains their basic attributes.

When passing the pointer to the structs instead of the address,

more relative data and attributes could be easily accessed.

(Otherwise we will have to use a hash table or a list to find the structure)

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 `page.h`:

```
/* where the data of a page in \*/*

enum page_type{

  FRAME,

  SWAP,
```

```
  FILE

};



/* spt entry with information about page */

struct sup_page_table_entry {

  /* information for swap */

  enum page_type status;       /* where the data are in */

  struct frame_table_entry* frame;  /* frame */

  size_t swap_id;           /* swap index in bitmap table */

};
```

in `frame.h`:

```
struct frame_table_entry {

  struct list_elem ele;          /* for list */

  uint32_t* frame;            /* ptr to page that currently
occupies it */

  struct thread* owner;          /* the thread who owns it */

  struct sup_page_table_entry* page;   /* ptr to supplemental page
enty */

};
```

---- 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.

A random frame is selected and evicted as long as it is not occupied by `mmap` critical functions.

>> 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?

As described in ***pintos-Guide***,

> *• You will be evicting the frame, therefore you the page associated with the frame you have selected needs to be unlinked. Then you want to remove this frame from your frame tableafter you have freed the frame with* `pagedir_clear_page`*.*
>
> *• You do not want to delete the supplementary page table entry associated with the selected frame. The process that was using the frame should still have the illusion that they still have this page allocated to them. If you delete this page table entry, you will not be able to reclaim the data from disk when needed.*
>
> *• Find a free block to write your data to. Since the blocks are just numbered contiguously, you just need an index that is free. Now this index is going to be needed to reclaim the data of the page, therefore it would be best to keep this index of where the data is in some member variable in the supplemental page table entry.*
>
> *• You'll also want to keep track of which pages are evicted and which are not for quick checking.*

In `struct frame_table_entry`, `owner` marks which process owns this frame and `page` marks which page is now occupying this frame. We will not delete the page, but the page and the frame is no longer linked.

This way, after eviction, the `frame` pointer of Q is set to `NULL` and its `status` is set to `SWAP`, meaning it is now linked with no frame and the data is written to the swap slots on the disk.

Therefore it is garanteed that the frame is associated with page P instead of 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.

As long as the pointer is:

\1. not NULL

\2. not a kernel address, (Below PHYS_BASE)

\3. not below the bottom of user stack

\4. is a reasonable stack access, (`esp-4` or `esp-32`)

The stack will be extended into the page.

---- 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.)

At each time a swap happens, a system-wise lock should be acquired

before the actual manipulation begins and released as soon as the

operation id finished.

Therefore all operations about `file`, `frame`, `swap` are atomic and

there will be no data race or deadlock.

>> 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?

The eviction process happens during page fault, which is an interruption.

Process Q will not be scheduled until the eviction is done.

The frame to be evited is chosen at random. Such race won't happen unless there remains only one available physical page.

>> 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?

The eviction process happens during page fault, which is an interruption. Process Q will not be scheduled until the eviction is done.

>> 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?

All the access to paged-out pages is handled by the page fault exception. If the page fault occurs during system calls or other kernel processes, the stack pointer used to handled the page fault is accordingly chosen. Attempts to access invalid virtual addresses will cause the misconducting process to exit with status code `-1`.

---- 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 use multiple single locks since pintos itself is a single-threading system supporting limited parallelism.

For example, most operations in virtual memory are locked independently. They are accomplished by multiple locks but the locks will not be comflicted.

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.

```
struct mmap_descriptor{

 struct list_elem elem; // for list operation



 mapid_t md;       // md number (start from 1)

 struct file *file;   // the opened file

 void *addr;       // mmap first page

 size_t size;      // file size

};
```

---- 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.

Memory mapped files has its own `mmap` descriptor, acting as a file descriptor for a normal file. In this descriptor, the memory pages mapping the file is recorded. The corresponding page entries store the actual file handler, the size and the offset.

During page fault handling, swapped pages will be reclaimed while frame or file pages will cause the process to exit unless a stack growth operation is needed.

During the eviction process, the (dirty) data in the memory mapped file pages will not be stored into swap space but back into the origin disk file.

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

>> any existing segment.

By calling `spt_hash_lookup` on all contiguous pages covering the size of the file to be mapped, we can avoid the new file pages overlapping existing pages.

---- 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 process of writing or reading data to or from the disk file is shared between `mmap` and `swap`.

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?