CSE 421/521
PROJECT 3: VIRTUAL MEMORY
DESIGN DOCUMENT

---- GROUP ----

Aditya Subramanian Muralidaran (adityasu@buffalo.edu)
Pratibha Arjun Barsale (pbarsale@buffalo.edu)
Venkata Krishnan Anantha Raman (va34@buffalo.edu)

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.

```
struct suppl_page_table_info{

        void *uv_addr;
        struct hash_elem hash_table_elem;
        struct file *load_file;
        size_t page_offset;
        size_t zero_bytes;
        size_t read_bytes;
        bool is_writable;
        bool page_isloaded;
        bool is_page_accessed;
};
```

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for locating the frame,
>> if any, that contains the data of a given page.
Using the given page, we check if the user virtual address is present in the page table. If it is present, we return the corresponding frame, i.e. kernel virtual address. If it is not present then we look in the supplemental page table and find if the page is from a file or swap and the details about whether the page is loaded into frame.
When the user will execute the instruction again, the user's virtual address should be map correctly now.

>> 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?
We access dirty and accessed bits only through user virtual address. While using pointers, we are checking its legality first. The page table is checked against the user address rather than the kernel address.

---- SYNCHRONIZATION ----

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

>> how are races avoided?
Our implementation is such that at any given time only one process can access and get the frame. This is implemented using locks so that if one process holds the lock then the other processes must wait for it to be released. Using this method, we avoid race conditions. Since the frame table has a lock, it will make sure that the frames are added without conflicts.

---- RATIONALE ----

>> A5: Why did you choose the data structure(s) that you did for
>> representing virtual-to-physical mappings?
Until now all the pages of one process are all mapped in the per-process page table. But now we will be dealing with swap and lazily loaded pages. So, we decided to implement new data structures.

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 frame_table_entry{

        struct thread *thread;
        struct list_elem frame_entry_elem;
        void *frame;
        struct suppl_page_table_info *spte;
};

struct list frame_list;
```

---- 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.
For finding a frame, first all the frames will be scanned one by one. For each frame we have set the attributes to indicate the availability of it. If the accessibility attribute is false, the frame will be evicted. Also while swapping a page we are planning to check the dirty bit. If the dirty bit is set to 1, the page is written back to memory before overwriting.

>> 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 that used by Q, we get the thread structure of Q and free it’s page. This will remove Q’s reference.

>> 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.

Any user memory access between PHYS_BASE and esp - 32 is valid. Any access beyond this is considered invalid. This will also cause the stack size to grow beyond the maximum limit of 8mb.

---- 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.)
Whenever a user virtual address is accessed, an attribute in the corresponding frame will be set true indicating it is in use and cannot be evicted. The frames set true are ignored while running the algorithm for frame eviction.

>> 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?
Our implementation requires lock before entering the page table. So, only one process can access a page at a time and this avoid race conditions. When Q is accessing a frame, it will set the bit to true. When P will scan through the frames, it will ignore the frames with the bit true. So the stated condition is avoided using bit.

>> 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?
This also uses the same concept stated in the above question. Mutual exclusion is achieved by setting a bit to true when a process is using a frame. This frame won’t be evicted by another process thus avoiding the condition of evicting the frame used by another process.

>> 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?
If a page is paged-out during a system call it will be brought back. An attribute in frame table entry will be set to true for the corresponding frames and this will prevent it from being considered for eviction process. Also, the memory pointers are being checked for validation before use. If virtual memory address is found to be invalid, the user process is killed and the pages held by it are freed. The bit is again set to false in the frame table entry.

---- 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.
Currently, we decided to go with a single lock as it will be easy to detect deadlock cases and the design is also simple to understand without complicating

the locking mechanisms. Single bit is being used to take care of conflicts during frame eviction.

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 list memory_map_list;
Int id;

---- 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 maintain a hash table of memory mapped files which will be initialized when the process starts. By using this it will be much simpler to free the memory mapped files when the process exits. Also, while swapping the pages, we are planning to check the dirty bit and write the page table back to the system before swapping.

>> C3: Explain how you determine whether a new file mapping overlaps
>> any existing segment.
The file is mapped page by page and the corresponding supplemental page table pointer is added to the thread’s hash table. If a duplicate entry is found all the previously mappings will be removed.

---- 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.
Our implementation uses shared data structure and this reduces redundancy in code. The dirty pages are written back to their corresponding file. The non-dirty files are planned to swap as it is.

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?