Advanced Operating Systems
Project 3: Virtual Memory
Design Document

The questions in this design document should reflect the design of the code you wrote for the project. **Your grade will reflect both the quality of your answer in this document and the quality of the design implementation in your code.** You may receive partial credit for answering questions for parts of the project that you did not get to implement, but you must indicate in your answer that there is no corresponding implementation, or you will not receive any credit.

For each question, **you should include both the name of the file(s), function name(s), and the line numbers where the relevant code may be found** – both the code that answers the question directly and any function that you refer to in your answer.

These design documents will be completed and submitted as a group. Please use this document as a guide for design and discuss the questions and their potential answers prior to beginning implementation. **All text in *italics* is intended to be replaced with your answer**. When you have completed your design document, submit it to the Canvas assignment Project 1 (Threads): Design.

Team Information Fill out the following information for each member of your team.

Name: *Yunseok Kang*
EID: *yk9784*
Email: *yunseok.kang@utexas.edu*
Unique Number: *51275*

Name: *Dohyun Kwon*
EID: *dk29493*
Email: *gundo0109@utexas.edu*
Unique Number: *51275*

Name: *Stefanus Adrian*
EID: *sja2673*
Email: *stefanus.adrian@utexas.edu*
Unique Number: *51275*

Preliminaries Please provide any preliminary comments and cite consulted sources.

If you have any preliminary comments on your submission or notes for the TAs, please give them here.

Your answer 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.

Your answer here.

PAGE TABLE MANAGEMENT (10 points)

The 10 total points are divided equally among the following sub-questions.

A1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration that was necessary for your page table implementation (frame table information should go in B1). Identify the purpose of each in 25 words or less.

- struct vm_entry in line 14 of page.h:
  - Purpose: Represents a single virtual page in the process's virtual address space, containing information necessary for managing this page.
- # define VM_EXEC 0 in line 9 of page.h:
  - Purpose: Load data from a executable file.
- # define VM_MAP 1 in line 10 of page.h:
  - Purpose: Load data from a mapped file.
- # define VM_SWAP 2 in line 11 of page.h:
  - Purpose: Load data from swap space.
- struct hash vm in struct thread in line 154 of thread.h:
  - Purpose: Hash table for storing vm_entry structures, facilitating fast lookup of virtual pages by their virtual address.

A2. Describe your code for locating the necessary data when a process page faults in the code segment.

When a process page faults in the code segment, our code first retrieves the faulting address from the CR2 register. It then checks if the address falls within the range of addresses mapped by the process's page table by consulting the supplemental page table (a hash table implemented in the vm member of struct thread). If the address is found, the corresponding **vm_entry** provides information on how to load or swap in the page. If the page needs to be loaded from disk or swap, the appropriate steps are taken to bring the page into memory and update the process's page table to reflect the newly loaded page.

Related functions and variables in page.h:

- vm_entry (line 14)

A3. How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that are aliases for the same frame, or alternatively how do you avoid the issue?

Accessed and dirty bits are managed by periodically scanning the frame table and the supplemental page table (SPT). When a frame is accessed or modified, the corresponding bits in the page directory and page tables are set by the hardware. Our code transfers these bits from the hardware-managed page tables to the SPT entries during operations like swapping out or eviction, ensuring the software representation is up-to-date. This process avoids issues with aliases by ensuring that whenever a

frame is shared, updates to its status are consistently propagated to all aliases' SPT entries

Related functions and variables in frame.h:

- struct list lruList (line 10)

A4. When two user processes both need a new frame at the same time, how are races avoided?

To avoid races when two user processes simultaneously request new frames, we use a global frame table lock. Before a process can allocate or free a frame, it must acquire this lock, which ensures only one process can modify the frame table at any given time. This mechanism prevents race conditions and ensures that frame allocation and deallocation operations are atomic, thus maintaining the integrity of the frame table.

Related functions and variables in frame.h:

- struct lock lruLock (line 11)

A5. Why did you choose the data structure(s) that you did for storing any supplemental page information? Please compare to an alternative data structure that is also a reasonable option.

We chose a hash table for storing supplemental page information due to its efficient O(1) average-case time complexity for insert, delete, and search operations. This efficiency is critical for page fault handling, where quick lookup of page information is necessary. An alternative data structure could be a balanced binary search tree (e.g., AVL tree or red-black tree), which offers O(log n) time complexity for these operations. While still efficient, the log n factor makes it slightly slower than a hash table for large numbers of entries. However, a binary search tree could provide ordered access to pages, which might be beneficial for certain page replacement algorithms.

Related functions and variables in thread.h:

- struct hash vm (line 154)

PAGING TO AND FROM DISK (10 points)

The 10 total points are divided equally among the following sub-questions.

B1. Copy here the declaration of each new or changed 'struct' or 'struct' member, global or static variable, 'typedef', '#define', or enumeration that was necessary for frame and swap implementation (do not include any declarations you included in A1). Identify the purpose of each in 25 words or less.

- struct list lruList in line 10 of frame.h:
  - Purpose: LRU list for managing frames
- struct lock lruLock in line 11 of frame.h:
  - Purpose: Lock for synchronizing access to the LRU list
- struct list_elem *lruIter in line 14 of frame.h:
  - Purpose: Pointer for iterating the LRU list with the clock algorithm
- struct bitmap *swap_bitmap in line 7 of swap.h:
  - Purpose: Tracks used/unused blocks in swap space
- struct vm_entry in line 14 of page.h:
  - Purpose: Describes a virtual memory page, including its type, virtual address, and whether it's loaded into physical memory.
- struct page in line 39 of page.h:
  - Purpose of void *kaddr: Physical address of the page
  - Purpose of struct vm_entry *vme; : Pointer to the vm_entry mapped to this physical page
  - Purpose of struct thread *thread; : Pointer to the thread using this physical page
  - Purpose of struct list_elem lru; : List element for LRU list linkage
- struct mmap_file in line 46 of page.h:
  - Purpose: Manages memory-mapped files, associating file objects with their virtual memory entries and a unique mapping ID.

B2. When a frame is required but none are free, some frame must be evicted. Describe your LRU page replacement implementation.

We use a global **lruList** to keep track of all allocated frames. Each frame is wrapped in a struct page that includes metadata like the frame's physical address, the owning thread, and a pointer to the corresponding virtual memory entry. The **lruIter** pointer is used to implement the clock algorithm over the LRU list. When a frame needs to be evicted, we iterate over the LRU list starting from **lruIter**, looking for an unpinned and unaccessed page to evict. If all pages are accessed or pinned, we loop until we find a suitable candidate. This mechanism ensures that the least recently used frame, which is not actively in use, is selected for eviction.

Related functions and variables in frame.h:

- struct list lruList (line 10)
- struct list_elem lruIter (line 14)

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 that Q no longer has the frame?

When a frame previously used by process Q is obtained by process P, we remove the mapping from Q's page table by clearing the appropriate page table entry, which is done in **pagedir_clear_page()**. We also remove the frame from Q's LRU list and update the frame's metadata to reflect its new association with process P and its virtual memory entry. This ensures that process Q cannot access the frame anymore, as its page table no longer contains a mapping to it.

Related functions and variables in pagedir.c:

- void pagedir_clear_page (line 138)

B4. Explain your heuristic for deciding whether or not page fault for an invalid virtual address should cause the stack to be extended.

We extend the stack when a page fault occurs for an address within a certain range below the stack pointer (esp) but above a minimum stack base address. Our heuristic checks if the fault address is within 32 bytes of the stack pointer, which allows for typical push operations, or if the address is part of a function call mechanism that might require additional space (e.g., for function arguments). This approach allows dynamic stack growth for scenarios like deep recursion and large arrays allocated on the stack.

Related functions and variables in exception.c:

- static void page_fault (line 120)

B5. Explain the basics of how you managed synchronization in your VM design. In particular, explain how it prevents deadlock.

Our design uses multiple locks to protect different parts of the VM system, including frame list and swap space management. We carefully order lock acquisitions and ensure that a thread holding a lock does not wait on another lock to prevent deadlocks. For operations that might require waiting (e.g., disk IO during swap in/out), we release locks before the operation and re-acquire them afterward, minimizing the time locks are held and allowing other threads to progress.

Related functions and variables in frame.h:

- struct list lruList (line 10)
- struct lock lruLock (line 11)

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?

When process P experiences a page fault necessitating either the loading of a page into memory or its retrieval from swap space, it may lead to the eviction of a frame owned by another process Q. To safeguard against concurrent access and modification by Q during this eviction process, the system employs a pinned attribute within the vm_entry structure. Marking a vm_entry as pinned effectively bars any other processes or threads from altering the page while it is in the midst of being swapped out or loaded, thus maintaining the integrity of the operation. To manage the swap space effectively, especially when reading from or writing to it, the system utilizes functions dedicated to swapping in and out, alongside a global bitmap that tracks which swap slots are available. This bitmap plays a critical role in synchronizing access to the swap space, ensuring that operations do not overlap in a way that could cause data corruption or loss. Crucial to the synchronization of these operations is a global lock, referred to as lru_list_lock, which orchestrates the timing of swaps and access to physical frames. By locking down critical sections of the code during swap operations, the system avoids race conditions between processes attempting to evict each other's frames and those trying to fault pages back into memory. This sophisticated approach, leveraging the pinned status of pages, a coherent management of swap space, and stringent synchronization mechanisms through a global lock, ensures that the system can handle page faults and memory management operations without resorting to disabling interrupts. It prevents unwanted access to pages undergoing swap operations, thereby preserving system stability and ensuring that memory management tasks are executed seamlessly.

Related functions and variables in frame.h:

- struct lock lruLock (line 11)

B7. Suppose a page fault in process P causes a page to be read from the filesystem or swap. How do you ensure that a second process Q cannot interfere by, for example, attempting to evict the frame while it is still being read in?

To prevent interference from another process while a page is being read into memory due to a page fault, the system employs a pinned attribute in the **vm_entry** structure. When a page fault occurs and a page read is initiated, the corresponding page is marked as pinned, indicating it's in use and cannot be evicted. This mechanism ensures exclusive access to the page during the read operation. A global lock, **lruLock**, further synchronizes memory management operations, preventing concurrent eviction attempts by other processes. Once the page is successfully loaded into physical memory, it is unpinned, allowing for normal operations to resume. This approach, combining the pinned attribute with global synchronization, safeguards against potential interference, ensuring stable and reliable memory access during page fault handling.

Related functions and variables in frame.h:

- struct lock lruLock (line 11)

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? If your method could result in potential deadlock, how do you prevent it?

In handling access to paged-out pages during system calls, the system uses a combination of page faults and a locking mechanism. Page faults trigger the loading of absent pages into physical memory, either from swap space or disk files, as needed by system calls. Additionally, a "locking" mechanism,

implemented through the pinned attribute in the **vm_entry** structure, temporarily secures critical frames in physical memory to prevent their eviction during pivotal operations. This dual approach ensures system calls proceed smoothly without disruption from paging activities. To avoid potential deadlocks, especially where multiple processes could simultaneously request several pages, the system employs strict ordering in lock acquisition and limits on pinning operations. These measures prevent scenarios where processes indefinitely wait on each other, thereby maintaining system stability and responsiveness during intensive memory operations.

Related functions and variables in page.h:

- vm_entry (line 14)

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.

In designing the virtual memory (VM) system, a balance between synchronization simplicity and parallelism is sought to minimize deadlock risks. The system employs a moderate number of locks, specifically targeting the LRU list for page management and potentially the frame table, though not explicitly detailed in provided snippets. This approach aims to:

Enhance Performance: Avoiding a single global lock eliminates bottlenecks, facilitating concurrent operations across threads. This is crucial for handling frequent page faults, allocations, and evictions efficiently.

Reduce Complexity and Deadlock Risk: While many fine-grained locks could maximize parallelism, they significantly complicate synchronization and increase deadlock potential. The chosen moderate locking strategy simplifies management and reduces deadlock scenarios by focusing on critical sections prone to concurrent access conflicts.

Achieve Practical Trade-offs: The system opts for practicality, striking a balance between operational parallelism and manageable synchronization complexity. This ensures reasonable system performance and stability without overly complicating the synchronization logic.

Ultimately, this design leverages strategic lock placement to optimize the VM system's efficiency and reliability, prioritizing a practical balance between simplicity, performance, and safety from deadlocks

Related functions and variables in frame.h:

- struct list lruList (line 10)

SURVEY QUESTIONS (optional) Answering these questions is optional, but it will help us improve the course in future semesters. 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 semester.

In your opinion, was this assignment, or any one of the two problems in it, too easy or too hard? Did it take too long or too little time?

Your answer here.

Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

Your answer here.

Is there some particular fact or hint we should give students in future semesters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

Your answer here.

Do you have any suggestions for the TAs to more effectively assist students, either for future semesters or the remaining projects?

Your answer here.

Any other comments?

Your answer here.