CSE311 OPERATING SYSTEMS

Pintos Project 3 Design Document

```
            +---------------------------+
            |          CS 212           |
            | PROJECT 3: VIRTUAL MEMORY |
            |      DESIGN DOCUMENT      |
            +---------------------------+
```

```
---- GROUP ----

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

Jeonghoon Park <hoonably@unist.ac.kr>
Deokhyeon Kim <ejrgus1404@unist.ac.kr>

---- 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 threads/thread.h:
        struct hash page_table; => Supplemental page table: maps each user vaddr to its VM metadata for page faults.

    In vm/frame.h:
        struct frame {
            void   *kpage;         => Kernel virtual address of the frame.
            void   *upage;         => Corresponding user virtual address.
            struct thread *thread; => Owner thread of this frame.
            struct list_elem elem; => Element for insertion into global frame_list.
        };

    In vm/frame.c:
        static struct list frame_list; => Global list of all frames for eviction.
        static struct lock frame_lock; => Synchronizes access to frame_list and frame allocation.

    In vm/page.h:
        struct page {
            uint8_t *vaddr;        => User virtual page start.
            bool     writable;     => Is page writable?
            bool     is_loaded;    => Has a frame been allocated?
            enum page_type type;   => Backing store type.
            struct file *file;     => Backing file (for binary/mmap).
            off_t    offset;       => Byte offset in file.
            size_t   read_bytes;   => Bytes to read from file.
            size_t   zero_bytes;   => Bytes to zero fill.
            struct hash_elem elem; => Hash table element (keyed by vaddr).
            int      swap_slot;    => Index of swap slot if paged out.
        };

---- ALGORITHMS ----

>> A2: In a few paragraphs, describe your code for accessing the data
>> stored in the SPT about a given page.

    In our implementation, each thread's supplemental page table is a struct hash
    stored in thread->page_table, keyed by page-aligned user virtual addresses.
    To look up the metadata for a faulting address:

    1. Round the faulting address down to its page boundary using
    pg_round_down(fault_addr).
```

2. Construct a temporary struct page on the stack with only its vaddr field
set to that rounded address.
3. Call hash_find(&thread_current()->page_table, &tmp.elem).
- Our hash callbacks, page_hash(e) and page_less(a,b), hash and compare
on ((struct page*) e)->vaddr.
4. If hash_find returns non-NULL, convert the returned struct hash_elem*
to struct page* via:
hash_entry(elem, struct page, elem).
5. If it returns NULL, no supplemental entry exists. The access is invalid,
or (for valid stack growth) a new page must be created.

Once we have struct page* p, its fields (p->type, p->file, p->offset,
p->read_bytes, p->zero_bytes, p->swap_slot, p->writable, etc.) provide all
information needed by the page-fault handler to:
• allocate a frame,
• fetch or zero data,
• install the PTE, and
• mark p->is_loaded = true.

This cleanly separates lookup logic (O(1) expected per lookup) from
fault-handling logic.

>> 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?

A3: Coordinating Accessed and Dirty Bits

In our implementation, we always operate on the user PTE via pagedir_* on
the user virtual address (upage), avoiding the kernel alias entirely.

1. Accessed Bit
• We call pagedir_is_accessed(thread->pagedir, upage) to test if the
page was referenced.
• After sampling (in frame_evict_and_reuse), we call
pagedir_set_accessed(thread->pagedir, upage, false) to clear it.

2. Dirty Bit
• Unlike a full optimization, we do NOT call pagedir_is_dirty or
pagedir_set_dirty.
• Instead, on eviction we always call swap_out(kpage, &slot) if a
supplemental entry exists, regardless of whether the page was dirty.
• This simplifies alias handling—no need to track or clear dirty bits—
at the cost of extra swap I/O for clean pages.

Because all bit checks and clearances use the user vaddr and we never touch
the kernel virtual alias's PTE, there is no inconsistency between aliases.

---- SYNCHRONIZATION ----

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

In our implementation, frame_alloc(flags, upage) and
frame_evict_and_reuse() serialize all frame-table updates and eviction
under a single global lock, frame_lock:

1. frame_alloc(flags, upage):
- Call palloc_get_page(flags) to try allocating a new page.
- If that fails, call frame_evict_and_reuse() to reclaim one.
- Acquire frame_lock.
- Allocate and initialize a struct frame (kpage, upage, thread).
- Insert it into frame_list via list_push_back().
- Release frame_lock and return kpage.

2. frame_evict_and_reuse():
- Acquire frame_lock on entry.
- Loop over frame_list to find a victim with accessed bit == false.
- For the chosen victim:
  - pagedir_clear_page() to unmap from its page table.
  - swap_out() under swap_lock to write back if dirty.
  - Update the victim's struct page (swap_slot, type, is_loaded).
  - Remove the victim's frame struct (list_remove + free).
- Release frame_lock and return the reclaimed kpage.

Because both allocation (insertion) and eviction (removal) hold
frame_lock throughout their critical sections, no two threads can
manipulate frame_list or reclaim the same frame concurrently. This

prevents races in frame allocation and eviction.

---- RATIONALE ----

>> A5: Why did you choose the data structure(s) that you did for
>> representing virtual-to-physical mappings?

We use a per-process hash table (struct hash) for the supplemental page table, mapping each user virtual page (vaddr) to its struct page entry.

- Sparse address space: User pages are sparse and dynamic; a hash table avoids large arrays and wasted memory compared to a frame-indexed array.
- Fast lookup: hash_find provides average O(1) lookup on page faults, minimizing page-fault handling overhead.
- Dynamic updates: Insertion and deletion of entries (load, eviction, unmap) are efficient and require no table resizing.

The global frame_list uses a doubly linked list because:
- Eviction scan: We iterate over resident frames in clock order.
- Easy removal: list_remove() is O(1) when freeing or evicting a frame.
- Simple insertion: list_push_back() for newly allocated frames.

These structures balance performance, memory usage, and implementation simplicity, fitting Pintos's educational goals and our VM design.

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 vm/swap.c:
```
static struct lock    swap_lock;   => Synchronizes swap table operations.
static struct bitmap *swap_bitmap; => Bitmap tracking free/used swap slots.
static struct block  *swap_block;  => Block device handle for the swap partition.
```

---- 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.

When frame_alloc() finds no free page, it calls frame_evict_and_reuse(), which:

1. Acquires the global frame_lock to serialize eviction.
2. Enters an infinite loop ("clock" sweep):
    a. Iterates over each struct frame in frame_list in insertion order.
    b. For each candidate 'victim':
        - Calls pagedir_is_accessed(victim->thread->pagedir, victim->upage).
        - If accessed==true: clears it via
          pagedir_set_accessed(..., victim->upage, false)
          and continues the scan ("second chance").
        - If accessed==false: selects this frame as the victim.
3. For the chosen victim:
    a. Unmaps the page via pagedir_clear_page(victim->thread->pagedir,
       victim->upage).
    b. Looks up its supplemental entry vme = find_page_entry(...).
    c. If vme exists:
        - Calls swap_out(victim->kpage, &slot) under swap_lock.
        - Sets vme->swap_slot = slot, vme->type = PAGE_SWAP,
          vme->is_loaded = false.
    d. Removes the struct frame from frame_list and frees it.
4. Releases frame_lock and returns the reclaimed kpage pointer.

This is a second-chance (clock) algorithm: frames with accessed==true get their bit cleared once, then are evicted on the next sweep if not accessed again, approximating LRU with O(n) worst-case scan but low overhead in steady state.

>> 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 a frame kpage is reclaimed from process Q and assigned to process P, we:

1. In frame_evict_and_reuse() for Q:

```
        a. Call pagedir_clear_page(Q->pagedir, Q_upage) to unmap the old PTE.
        b. Lookup Q's supplemental entry:
                vme = find_page_entry(&Q->page_table, Q_upage);
        c. If vme exists:
                - Under swap_lock, call swap_out(kpage, &slot)
                - vme->swap_slot = slot
                - vme->type      = PAGE_SWAP
                - vme->is_loaded = false
        d. Remove and free Q's frame struct:
                list_remove(&victim->elem)
                free(victim)

    2. Back in frame_alloc(flags, P_upage) for P:
        a. Acquire frame_lock, malloc a new struct frame f
        b. Initialize f->kpage = kpage, f->upage = P_upage,
                f->thread = thread_current()
        c. Insert f into frame_list via list_push_back()
        d. Release frame_lock

    3. Finally install P's new mapping:
        pagedir_set_page(P->pagedir, P_upage, kpage, writable);

    This sequence ensures Q's PTE and supplemental data reflect the eviction,
    and P's page table and frame_list are updated for the new owner.

>> 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.

    When page_fault() sees a fault at user address fault_addr with no SPT entry:

    1. Check user-mode stack bounds:
        • fault_addr < PHYS_BASE (0xC0000000)
        • fault_addr >= (intr_frame->esp - 32)
        • (PHYS_BASE - fault_addr) ≤ MAX_STACK_BYTES (e.g., 8 MB)

    2. If all true, treat as valid stack growth:
        a. Create a new struct page with:
            - vaddr = pg_round_down(fault_addr)
            - type = PAGE_STACK
            - read_bytes = 0, zero_bytes = PGSIZE
            - writable = true, is_loaded = false
        b. Insert into thread->page_table via hash_insert().
        c. In fault handler, allocate frame, zero it, install PTE,
            set is_loaded = true.

    3. Otherwise, fault is invalid; terminate the process.

    This ensures only accesses just below the current ESP (within 32 bytes)
    and within the maximum stack limit trigger lazy stack allocation.

---- 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.)

    We employ two primary locks for VM structures:

    1. frame_lock (vm/frame.c)
        - Protects the global frame_list and all frame allocation/eviction.
    2. swap_lock (vm/swap.c)
        - Protects the swap_bitmap and all swap_in/swap_out operations.

    Per-thread supplemental page tables (thread->page_table) and mmap_list are
    owned and accessed only by their thread during page faults or munmap, so they
    require no additional locking.

    Deadlock Prevention:
        • Lock Ordering:
            — Always acquire frame_lock before calling swap_out/in (which acquires
                swap_lock). No code ever acquires swap_lock first, avoiding circular
                wait.
        • No Nested VM Locks:
            — Per-thread structures are updated without holding VM locks.
        • Short Critical Sections:
            — Locks guard only list/hash/bitmap updates, minimizing hold time.
```

By eliminating circular lock dependencies and minimizing lock scope, we prevent deadlock while allowing parallel page faults and swap I/O to proceed safely.

>> 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 P’s page fault triggers eviction of Q’s frame, we:

1. Acquire frame_lock in frame_evict_and_reuse(), blocking any new faults that allocate or evict frames (including Q’s fault handler).

2. Call pagedir_clear_page(Q->pagedir, Q_upage) immediately, removing Q’s mapping. At this point, any user access by Q causes a new page fault, not a memory access to the old frame.

3. Under swap_lock, write the frame contents to swap, update Q’s struct page (swap_slot, type=PAGE_SWAP, is_loaded=false).

4. Remove and free Q’s struct frame, then release frame_lock.

Effects:

- **No access during eviction**: Clearing Q’s PTE prevents Q from reading or writing the frame while it’s being evicted.
- **No race on reload**: If Q faults on that address during eviction, its handler calls frame_alloc(), which blocks on frame_lock until eviction finishes. After eviction, Q’s page_lookup sees type=PAGE_SWAP and uses the recorded swap_slot to reload the page correctly.

This lock ordering and immediate PTE invalidation prevent Q from accessing or racing with P’s eviction of its frame.

>> 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?

In our page_fault() handler, we prevent concurrent eviction during I/O by holding `frame_lock` across the entire data-load and PTE install:

1. Allocate or reclaim a frame:
    void *kpage = frame_alloc(flags, upage);
2. Protect I/O and mapping with frame_lock:
    lock_acquire(&frame_lock);
        if (p->type == PAGE_SWAP) {
        lock_acquire(&swap_lock);
        swap_in(kpage, p->swap_slot);
        lock_release(&swap_lock);
        } else {
        file_seek(p->file, p->offset);
        file_read(p->file, kpage, p->read_bytes);
        memset(kpage + p->read_bytes, 0, p->zero_bytes);
        }
        pagedir_set_page(thread->pagedir, upage, kpage, p->writable);
        p->is_loaded = true;
    lock_release(&frame_lock);

Because both eviction (in frame_evict_and_reuse) and this load/install section require `frame_lock`, no other thread Q can evict the frame while P is still reading it in or installing its PTE.

>> 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?

We handle user buffers in syscalls by validating and preloading pages before holding any file-system or swap locks, relying on the normal page-fault handler to bring in pages, and killing the process on invalid addresses:

1. User Pointer Validation & Preload
    - Each syscall that reads or writes a user buffer calls
        is_valid_buffer(buffer, size, writable), which:
        — Rounds addresses down to page boundaries.

- For each page:
  * If vaddr ≥ PHYS_BASE or not is_user_vaddr ⇒ exit(-1).
  * Calls find_page_entry(&thread->page_table, vaddr).
    · If NULL: allows lazy loading (page_fault will allocate).
    · If found but !is_loaded: calls load_page(p) to
    allocate a frame, load from swap or file, install PTE.
    · Ensures p->is_loaded = true before proceeding.
- All frame and swap operations occur with frame_lock (and swap_lock) held, but no file-system lock is held, avoiding circular waits.

2. File-System I/O on Kernel Pages
- After preload, syscalls use pagedir_get_page or a kernel buffer to perform file_system_read/file_system_write (file_read/file_write) without touching user virtual addresses.
- Since pages are already loaded, no page faults occur during file-system calls.

3. Invalid Access Handling
- If is_valid_buffer finds an unmapped or invalid vaddr, we immediately terminate the process via exit(-1).

This design uses page faults to load pages lazily but ensures they occur before any file-system or swap locks are held—no explicit "pinning" is needed, and invalid accesses are handled cleanly.

---- 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 employ two global locks:
- frame_lock: protects the global frame_list and all frame allocation, eviction, loading, and PTE installation.
- swap_lock: protects the swap_bitmap and swap_in/swap_out operations.

Locking Strategy:
1. Consistent Lock Order
   - Always acquire frame_lock before swap_lock.
   - Never acquire swap_lock without first holding frame_lock.
2. No Circular Wait
   - Per-thread supplemental page tables and mmap_list have no locks.
   - File system locks are never held while holding swap_lock or frame_lock.
3. Short Critical Sections
   - frame_lock is held only around list/hash updates and frame I/O setup.
   - swap_lock is held only during the actual swap bitmap update and disk I/O.
4. Minimal Locks
   - No per-page or per-thread locks beyond the two globals.
   - Simplifies reasoning and avoids lock-order complexity.

Trade-Offs:
- Coarse-grained locking simplifies implementation and ensures deadlock freedom.
- Parallelism is limited: threads contend on frame_lock for any frame operations.
- Per-thread operations that don't touch frames (e.g., hash_insert on SPT) proceed without locking, preserving some concurrency.

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 threads/thread.h:
```
struct list mmap_list;  => List of this thread's active memory-mapped files (struct mmap_file).
mapid_t mmap_idx;       => Next mmap() mapping ID, unique per thread.
```

In vm/page.h:
```
enum page_type { => Type/backing of a page:
    PAGE_BINARY,   => loaded from executable
    PAGE_MMAP,     => memory-mapped file
    PAGE_STACK,    => dynamically grown stack
    PAGE_SWAP      => swap-backed page
```

```
        };

        struct mmap_file {
            mapid_t      idx;      => Unique mapping ID returned by mmap().
            struct file *file;     => File being mapped.
            off_t        offset;   => Offset in file where mapping begins.
            size_t       length;   => Total bytes mapped.
            void        *upage;    => Start of the user-virtual address region.
            struct list_elem elem; => Element for thread's mmap_list.
        };

---- 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.

    1. Lazy-allocation on mmap:
        • In syscall_mmap(), reopen the file and record it in a struct mmap_file.
        • For each PGSIZE chunk of the file:
            - Allocate struct page *p.
            - Set p->vaddr   = addr + offset.
            - Set p->type    = PAGE_MMAP.
            - Set p->file    = reopened file.
            - Set p->offset  = offset.
            - Set p->read_bytes = min(file_length - offset, PGSIZE).
            - Set p->zero_bytes = PGSIZE - p->read_bytes.
            - Set p->writable   = true, p->is_loaded = false.
            - Call insert_page_entry(&thread_current()->page_table, p).

    2. Page-fault handling:
        • In exception.c, page_fault() finds p via find_page_entry().
        • Calls load_page(p):
            a. frame_alloc(PAL_USER, p->vaddr) to get kpage.
            b. Since p->type == PAGE_MMAP:
                    file_read_at(p->file, kpage, p->read_bytes, p->offset);
                    memset(kpage + p->read_bytes, 0, p->zero_bytes);
            c. pagedir_set_page(thread->pagedir, p->vaddr,
                                kpage, p->writable);
            d. p->is_loaded = true.

    3. Eviction (frame_evict_and_reuse):
        • Regardless of p->type, for a victim frame:
            - pagedir_clear_page(victim->thread->pagedir,
                                 victim->upage);
            - swap_out(victim->kpage, &slot);
            - vme->swap_slot = slot; vme->type = PAGE_SWAP;
                vme->is_loaded = false;
            - Remove and free the struct frame.

    4. Unmap cleanup (syscall_munmap):
        • For each mapped page p:
            if (p->is_loaded && pagedir_is_dirty(...)):
                file_write_at(p->file, p->vaddr,
                              p->read_bytes, p->offset);
            pagedir_clear_page(thread->pagedir, p->vaddr);
            if (kpage = pagedir_get_page(...)) frame_free(kpage);
            hash_delete(&thread->page_table, &p->elem);
        • Close the file, remove mmap_file from list, free it.

    Thus, mmap pages share the same lazy-load and eviction code as binary
    pages, but munmap specially writes back dirty pages to the original file.

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

    In syscall_mmap(fd, addr, esp), we detect overlap as follows:

    1. Preliminary Checks:
        - fd > 1, addr ≠ NULL, and pg_ofs(addr) == 0.
        - length = file_length(file) > 0.
        - Prevent stack collision: if addr ≥ esp or
            addr ≥ PHYS_BASE - STACK_MAX_SIZE, return -1.

    2. mmap_file Registration:
        - Reopen file, allocate struct mmap_file, set mf->addr = addr,
            mf->length = length, push to t->mmap_list.
```

    3. Supplemental Page Table Insertion:
        For each PGSIZE chunk at page_addr = addr + offset:
            a. Allocate struct page *p with p->vaddr = page_addr, type =
                PAGE_MMAP, file, offset, read_bytes, zero_bytes, is_loaded = false.
            b. Call insert_page_entry(&t->page_table, p).
                - insert_page_entry wraps hash_insert().
                - hash_insert returns non-NULL if a page with same vaddr
                already exists.
            c. If insert_page_entry returns false, syscall_mmap aborts with -1.

    By relying on hash_insert→insert_page_entry to fail on duplicate vaddr,
    we detect any overlap (with existing mmap regions, executable segments,
    stack pages, or prior mappings) at page granularity.

---- 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 share nearly all page-fault and loading code between executable-backed
    (PAGE_BINARY) and mmap pages (PAGE_MMAP):

    1. load_page(p):
        - frame_alloc(PAL_USER, p->vaddr) for any p->type.
        - If p->type == PAGE_SWAP: swap_in() + swap_free().
        - Else if p->type == PAGE_BINARY or PAGE_MMAP:
            * file_read_at(p->file, kpage, p->read_bytes, p->offset)
            * memset(kpage + p->read_bytes, 0, p->zero_bytes)
        - Else if p->type == PAGE_STACK: zero page.
        - pagedir_set_page(...), p->is_loaded = true.

    2. Eviction (frame_evict_and_reuse):
        - Regardless of p->type, always swap_out(kpage, &slot).
        - p->type = PAGE_SWAP, p->swap_slot = slot, p->is_loaded = false.

    3. munmap cleanup writes back mmap pages:
        - In syscall_munmap, for each p of type PAGE_MMAP:
            if pagedir_is_dirty(...):
                file_write_at(p->file, p->vaddr, p->read_bytes, p->offset)
        - Then clear PTE, free frame, delete p from hash.

    By unifying page_fault load and eviction for all p->type, we reuse code.
    We specialize only in munmap to write mmap pages back to their file
    instead of leaving them in 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?

==== Peer Evaluation ====

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

| Name | Contribution |

Jeonghoon Park	50%, Implemented frame, page table, mmap
Deokhyeon Kim	50%, Implemented eviction, swap, munmap