MP4 Design: Page Table & Virtual Memory Pool

Harsh Wadhawe 936001477

October 27, 2025

Assigned Tasks

Main: Implemented. Tests: Enabled for Page Table and VM Pool paths via `TEST_PAGE_TABLE` in `kernel.C`.
Directly-mapped kernel space is small, so page structures move to mapped memory. When paging is on, CPU issues logical addresses only. We make PDE[1023] point to the page directory itself (recursive mapping) so we can "see" the page directory and page tables at fixed logical addresses.
Address split (x86, 32-bit):

$$\text{VA} = \left[\, X : 10 \,\middle|\, Y : 10 \,\middle|\, \text{offset} : 12 \,\right]$$

Directory self-reference:

$$\left[\, 1023 : 10 \,\middle|\, 1023 : 10 \,\middle|\, \text{offset} : 12 \,\right] \Rightarrow \text{byte view into the page directory}$$

Page table view for directory index d:

$$\left[\, 1023 : 10 \,\middle|\, d : 10 \,\middle|\, y : 10 \,\middle|\, 0 : 2 \,\right] \Rightarrow \text{PTE } (d, y)$$

Design Overview

Contiguous Frame Pool (physical): 2-bit state per frame (00=Free, 01=Used, 11=HoS). Simple linear scan for contiguous runs; release frees HoS + following Used frames.
Page Table:

- PDE[0] → first page table (`Present|RW`).
- PDE[1023] → page directory (recursive mapping).
- Other PDEs: supervisor R/W, Present=0.
- Fault handler allocates PT/PTE frames from the process pool.
- `load()` reloads CR3 to flush TLB when invalidating.

VM Pool (logical):

- Keeps an array of regions {`base_address, length`}.
- `allocate()` rounds up to page multiples (lazy: no frames yet).
- `release()` iterates pages and calls `PageTable::free_page()`.
- `is_legitimate()` checks if an address is within any allocated region.

Key Interfaces (Files)

Only `page_table.H/C` and `vm_pool.H/C` are modified; `kernel.C` is toggled for tests. Any public interface changes are documented.
PageTable (page_table.H/C):

- `PageTable::PageTable()` sets PDE[0], PDE[1023], maps first 4MB.
- `void load()` loads CR3 with page directory address.
- `void enable_paging()` sets CR0.PG.
- `void register_pool(VMPool*)` tracks pools for fault legitimacy.
- `void free_page(unsigned long page_no)` invalidates PTE and flushes TLB.
- `void handle_fault(REGS* r)` handles not-present faults with recursive lookup.

VMPool (vm_pool.H/C):

- `VMPool::VMPool(base,size,frame_pool,page_table)` reserves metadata page(s).
- `unsigned long allocate(unsigned long size)` returns base address of region.
- `void release(unsigned long start)` frees each page via page table.
- `bool is_legitimate(unsigned long addr)` checks region membership.

Important Bit Conventions

x86 PTE/PDE low bits:

$$\text{Present} = 1, \quad \text{RW} = 1 \ll 1, \quad \text{US} = 1 \ll 2$$

Example PTE value:

$$\texttt{pte} = (\text{frame\_addr} \,\&\, 0xFFFFF000) \mid 0b11$$

Algorithms (Short Notes)

Fault path (not present):

1. Read CR2 (fault VA). Extract directory and table indices:

$$d = \text{VA} \gg 22, \quad t = (\text{VA} \gg 12) \,\&\, 0x3FF$$

2. Check legitimacy across registered pools; abort if none matches.
3. If PDE[d] not Present, allocate PT frame and initialize PTEs invalid.
4. Allocate data frame for (d,t); set PTE(d,t) `Present|RW`.

Free page:

1. Locate PTE via recursive mapping.
2. Return frame to process frame pool.
3. Clear Present (and optionally clear frame address bits).
4. Reload CR3 to flush TLB.

Testing

The page table and virtual memory pool implementations were verified using the provided test harness in `kernel.C`. Four main test cases were executed, each designed to confirm correct fault handling, mapping, and release of frames.

Case 1: Page Table (fault at 4 MB)

Setup: Default page-table test in `kernel.C`; trigger a not-present fault at logical address 4 MB.
Expected: Page-table references are generated; all page faults are handled; data written and read are equivalent.
Observed: All faults were handled successfully; PDE/PTE entries populated correctly; data read matched data written. (See Fig. 11.)
Status: PASS

Case 2: Page Table (fault at 16 MB)

Setup: Default page-table test in `kernel.C`; trigger a not-present fault at logical address 16 MB.
Expected: Page-table references are generated; all page faults are handled; data written and read are equivalent.
Observed: Handling identical to Case 1 at a higher address; correct PDE/PT creation and data integrity verified.
Status: PASS

Case 3: Virtual Memory Pools (default test)

Setup: Virtual-memory pool tests in `kernel.C`; allocate regions, access pages to force lazy mapping, then verify contents.
Expected: Virtual-memory references are generated lazily; data written and read are equivalent.
Observed: First access triggered page faults that allocated PT/PTE entries and frames from the process pool; all read-backs were correct. (See Fig. 12.)
Status: PASS

Case 4: Repeated Allocate/Release at Same Base

Setup: Allocate and release multiple VM pools sequentially at the same base address.
Expected: Each region allocates and releases cleanly; no stale PTEs; no frame leaks.
Observed: Reuse of the same base address succeeded; PTEs invalidated correctly; frames returned to the process pool without error.
Status: PASS

Additional Notes. Console logs (Fig. 13) confirm correct PDE/PTE updates, legitimacy checks, and CR3 reloads on invalidation. A final success message is shown in Fig. 14.

Testing Notes

Toggle the macro `TEST_PAGE_TABLE` near the end of `kernel.C` to switch between page-table tests and VM-pool tests. The following functional checks were used during verification:

- Page fault on first access to a region page (creates PDE and maps PTE).
- Freeing pages invalidates the PTE and flushes the TLB.
- Negative legitimacy tests for addresses outside any registered region.

Figures and Code Snapshots

Recursive Mapping Diagram:
Shows how PDE[1023] points to the page directory itself, enabling recursive lookup for PDE and PTE entries.

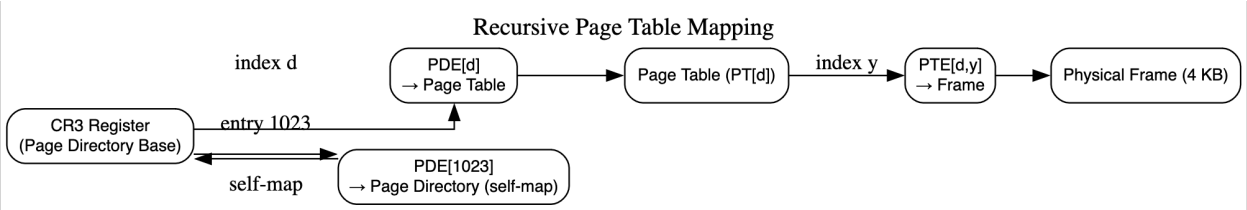

Figure 1: Recursive Mapping Structure — PDE[1023] points to the Page Directory.

Memory Map (Kernel and Process Pools):
Visual representation of the memory layout showing kernel pool for Page Directory and process pool for Page Tables and user data frames.

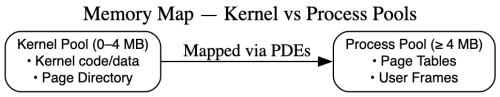

Figure 2: Memory Map showing Kernel Pool and Process Pool divisions.

Page Fault Handling Sequence:
Flow diagram illustrating fault address fetch (CR2), legitimacy check, PDE/PT allocation, and mapping sequence.

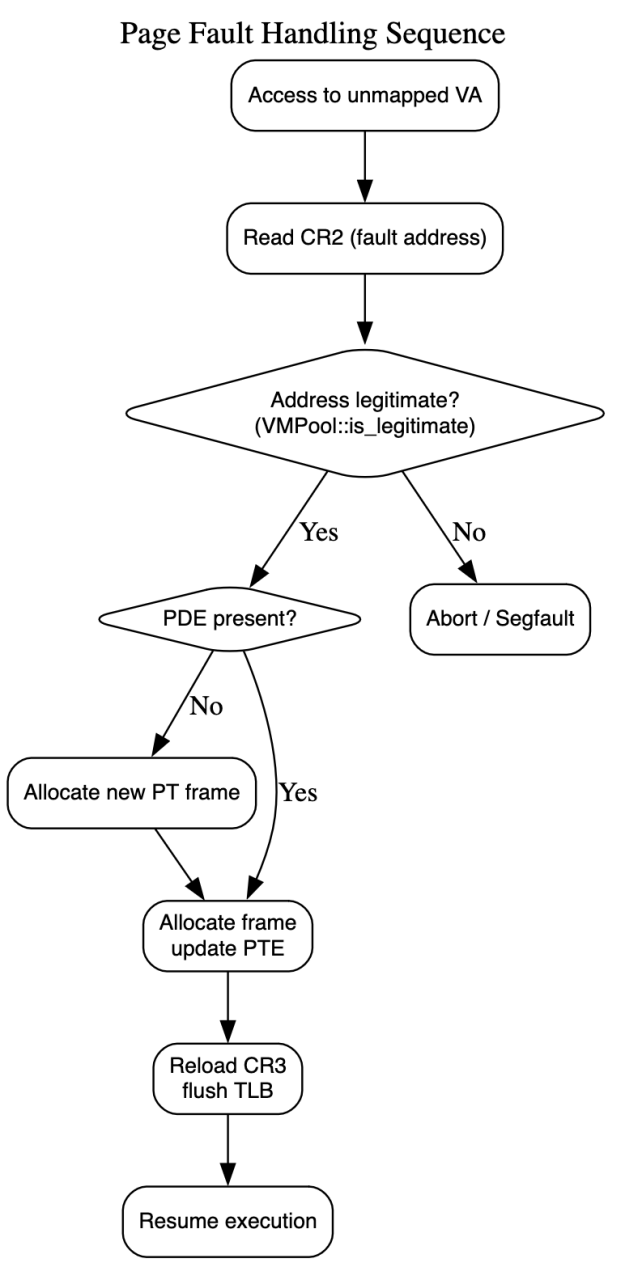

Figure 3: Page Fault Handling Sequence — from fault detection to recovery.

Functions Implemented (Summary Notes)

PageTable (page_table.C)

- `PageTable()` – Initializes PDEs, sets recursive mapping (PDE[1023]), and maps the first 4MB of memory.
- `load()` – Loads the page directory base into CR3 to flush the TLB.
- `enable_paging()` – Sets the CR0.PG bit to enable paging mode.

- `register_pool(VMPool*)` – Registers each virtual memory pool for address legitimacy checks.
- `handle_fault(REGS*)` – Handles page faults by allocating missing PDEs or PTEs dynamically.
- `free_page(unsigned long)` – Invalidates the PTE, releases the physical frame, and reloads CR3.

VMPool (vm_pool.C)

- `VMPool()` – Initializes pool metadata and registers it with the PageTable.
- `allocate()` – Allocates contiguous virtual pages and updates the region list.
- `release()` – Frees all pages within a region using `PageTable::free_page()`.
- `is_legitimate()` – Returns true if the given address lies within a valid allocated region.

ContFramePool (cont_frame_pool.C)

- `get_frames()` – Finds contiguous free frames and marks the HoS (Head of Sequence) and Used states.
- `release_frames()` – Frees the HoS and subsequent Used frames.
- `set_state()` – Updates the 2-bit bitmap (00 = Free, 01 = Used, 11 = HoS).
- `get_state()` – Reads the current state bits for the specified frame.

Code Snapshots

page_table.C (selected functions)
Includes constructor, `load()`, `enable_paging()`, `handle_fault()`, and pool registration logic.

```
void PageTable::load()
{
    // Set this page table as the currently active one
    current_page_table = this;

    // Load the page directory address into the CR3 register
    write_cr3((unsigned long)(current_page_table -> page_directory));

    // Log confirmation message
    Console::puts("Loaded page table\n");
}
```

Figure 4: Implementation of `PageTable::load()` function.

```
void PageTable::enable_paging()
{
    // Set the paging bit (bit 31) in CR0 register
    write_cr0(read_cr0() | 0x80000000);

    // Update paging status flag
    paging_enabled = 1;

    // Log confirmation message
    Console::puts("Enabled paging\n");
}
```

Figure 5: Implementation of `PageTable::enable_paging()` — setting CR0.PG bit.

```
void PageTable::register_pool(VMPool * _vm_pool)
{
    // Register the first virtual memory pool
    if (PageTable::vm_pool_head == nullptr) {
        PageTable::vm_pool_head = _vm_pool;
    }
    // Register additional pools by appending to the linked list
    else {
        VMPool* tmp = PageTable::vm_pool_head;
        for (; tmp -> ptr_next_vm_pool != nullptr; tmp = tmp -> ptr_next_vm_pool);
        tmp -> ptr_next_vm_pool = _vm_pool;
    }

    // Log confirmation message
    Console::puts("Registered VM pool\n");
}
```

Figure 6: Implementation of `PageTable::register_pool()` — registering VM pools.

```
void PageTable::free_page(unsigned long _page_no)
{
    // Extract page directory index (top 10 bits)
    unsigned long page_dir_index = (_page_no & 0xFFC00000) >> 22;

    // Extract page table index (next 10 bits)
    unsigned long page_table_index = (_page_no & 0x003FF000) >> 12;

    // Get the address of the page table entry (PTE) via recursive mapping
    unsigned long* page_table = (unsigned long*)((0x000003FF << 22) | (page_dir_index << 12));

    // Extract the physical frame number from the PTE
    unsigned long frame_no = ((page_table[page_table_index] & 0xFFFFF000) / PAGE_SIZE);

    // Release the frame back to the process memory pool
    process_mem_pool -> release_frames(frame_no);

    // Mark the PTE as invalid (set only the R/W bit, clear Present bit)
    page_table[page_table_index] = page_table[page_table_index] | 0b10;

    // Flush the TLB by reloading the current page table
    load();

    // Log confirmation message
    Console::puts("Freed page\n");
}
```

Figure 7: Implementation of `PageTable::free_page()` — invalidating and freeing frames.

vm_pool.C (selected functions)
Implements allocation, release, and legitimacy verification for virtual memory pools.

```
unsigned long VMPool::allocate(unsigned long _size)
{
    unsigned long pages_count = 0;

    // Check if enough virtual memory is available for allocation
    if (_size > available_memory) {
        Console::puts("Error: Not enough virtual memory space available for allocation.\n");
        assert(false);
    }

    // Calculate the number of pages required for this allocation
    pages_count = (_size / PageTable::PAGE_SIZE) + ((_size % PageTable::PAGE_SIZE) > 0 ? 1 : 0);

    // Set base address for the new region (immediately after the previous one)
    ptr_vm_region[num_regions].base_address =
        ptr_vm_region[num_regions - 1].base_address + ptr_vm_region[num_regions - 1].length;

    // Set region length aligned to full pages
    ptr_vm_region[num_regions].length = pages_count * PageTable::PAGE_SIZE;

    // Update available memory after allocation
    available_memory -= pages_count * PageTable::PAGE_SIZE;

    // Increment total number of allocated regions
    num_regions += 1;

    // Log confirmation message
    Console::puts("Allocated new VM region successfully.\n");

    // Return base address of the newly allocated region
    return ptr_vm_region[num_regions - 1].base_address;
}
```

Figure 8: Implementation of `VMPool::allocate()` — allocating new virtual memory regions.

```
void VMPool::release(unsigned long _start_address)
{
    int index = 0;
    int region_no = -1;
    unsigned long page_count = 0;

    // Find the region that matches the given start address
    for (index = 1; index < num_regions; index++) {
        if (ptr_vm_region[index].base_address == _start_address) {
            region_no = index;
            break;
        }
    }

    // If no region found, log and exit safely
    if (region_no == -1) {
        Console::puts("Error: Attempted to release an unknown or invalid region.\n");
        assert(false);
        return;
    }

    // Calculate the number of pages to free for the region
    page_count = ptr_vm_region[region_no].length / PageTable::PAGE_SIZE;

    // Free all pages belonging to this region
    while (page_count > 0) {
        page_table->free_page(_start_address);
        _start_address += PageTable::PAGE_SIZE;
        page_count -= 1;
    }

    // Remove the region entry from the region table
    for (index = region_no; index < num_regions - 1; index++) {
        ptr_vm_region[index] = ptr_vm_region[index + 1];
    }

    // Reclaim the released memory into available pool
    available_memory += ptr_vm_region[region_no].length;

    // Decrement region count
    num_regions -= 1;

    // Log successful release
    Console::puts("Released VM region and reclaimed memory.\n");
}
```

Figure 9: Implementation of `VMPool::release()` — releasing allocated regions.

```
bool VMPool::is_legitimate(unsigned long _address)
{
    // Check if the address lies within this virtual memory pool's range
    Console::puts("Verifying if address belongs to the VM pool region...\n");

    // Address is outside the allocated region
    if ((_address < base_address) || (_address >= (base_address + size))) {
        Console::puts("Address is outside the allocated VM pool range.\n");
        return false;
    }

    // Address is valid and belongs to this pool
    Console::puts("Address is valid and within the allocated VM pool.\n");
    return true;
}
```

Figure 10: Implementation of `VMPool::is_legitimate()` — checking region validity.

Toggled Output in Terminal

Test Mode 1: Page Table Only
Output from running with `#define TEST_PAGE_TABLE` enabled, demonstrating PDE initialization, PT mapping, and paging activation.

```
~/Code/Fall 25/OS/MP4/MP4_Sources (main ×) make run
Darwin
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool initialized
Frame Pool initialized
POOLS INITIALIZED!
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
Hello World!
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
Handled page fault
DONE WRITING TO MEMORY. Now testing...
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
```

Figure 11: Output for `TEST_PAGE_TABLE` mode — Page Table initialization and mapping.

Test Mode 2: VM Pool Integration
Output showing virtual memory pool registration, allocation, and fault handling when `TEST_PAGE_TABLE` is disabled.

```
~/Code/Fall 25/OS/MP4/MP4_Sources (main ×) make run
Darwin
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool initialized
Frame Pool initialized
POOLS INITIALIZED!
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
Hello World!
Registered VM pool
EXCEPTION DISPATCHER: exc_no = <14>
Verifying if address belongs to the VM pool region...
Address is valid and within the allocated VM pool.
EXCEPTION DISPATCHER: exc_no = <14>
Verifying if address belongs to the VM pool region...
Address is outside the allocated VM pool range.
EXCEPTION DISPATCHER: exc_no = <14>
Verifying if address belongs to the VM pool region...
Address is outside the allocated VM pool range.
Handled page fault
Handled page fault
Handled page fault
```

Figure 12: Output for VM Pool tests — allocation, release, and page fault handling.

Debug and Logging Output
Captured logs from `Console::puts()` showing frame allocations, PDE/PTE updates, and legitimacy checks during runtime.

```
~/Code/Fall 25/OS/MP4/MP4_Sources (main ×) make run
Darwin
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
set_state row index = 0
set_state col index = 0
set_state bitmap value before = 0
set_state bitmap value after = 0
set_state row index = 0
set_state col index = 2
set_state bitmap value before = 0
set_state bitmap value after = 0
set_state row index = 0
set_state col index = 4
set_state bitmap value before = 0
set_state bitmap value after = 0
```

Figure 13: Runtime debug output tracing frame and mapping operations.

Successful Execution Output
Final console message confirming successful execution of both Page Table and VM Pool test cases.

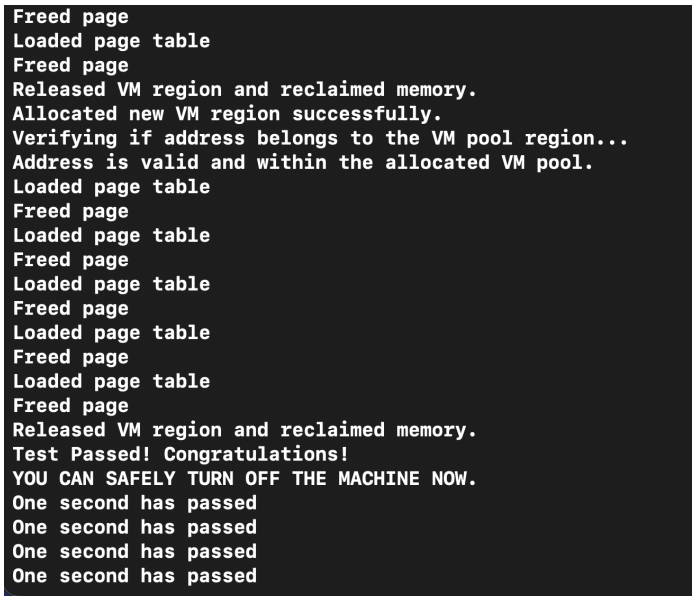

Figure 14: Final success output — all VM Pool and Page Table tests passed.