Lab 12: Memory Management

1 Objective

To build a C-based virtual memory simulator that manages a contiguous block of memory allocated from the host OS. The simulator's own data structures, such as page tables, must reside *within* this simulated physical memory, creating a more realistic and challenging memory management environment.

2 Conceptual Overview

In this lab, you will request a single, large chunk of memory using `malloc`. This chunk will serve as your entire "physical memory" or "RAM". All operations—allocating frames, storing page tables, and storing application data—must happen within this block.

2.1 Address Spaces

- **Host Virtual Address:** The actual pointer returned by `malloc`. We only use this to access the start of our simulated RAM.
- **Physical Address (`physaddr_t`):** An integer offset from the beginning of our simulated RAM. For a 64KB RAM, a physical address is a value from 0 to 65535. To access data at physical address `p_addr`, you would use `(void*)((char*)physical_memory_base + p_addr)`.
- **Virtual Address (`virtual_addr_t`):** A 32-bit integer that your simulated process uses. It has no direct relation to physical addresses until it is translated by your MMU.

2.2 Simulated Hardware

Our simulated machine will have the following characteristics:

- **Physical Memory Size:** 64 KB (65536 bytes)
- **Virtual Address Space Size:** 4 MB (4194304 bytes)
- **Page Size:** 4 KB (4096 bytes)
- **Physical Frames:** 64 KB / 4 KB = 16 frames (numbered 0 to 15)
- **Virtual Pages:** 4 MB / 4 KB = 1024 pages

2.3 Two-Level Page Table

To manage the 1024 virtual pages, we will use a two-level page table. A page number (10 bits) is split into:

- **Page Directory Index (PDI):** 5 bits (32 entries in the Page Directory)
- **Page Table Index (PTI):** 5 bits (32 entries in each Page Table)

A **Page Directory** contains 32 Page Directory Entries (PDEs). Each PDE points to a Page Table. A **Page Table** contains 32 Page Table Entries (PTEs). Each PTE points to a 4KB frame of application data. Both the Page Directory and Page Tables are data structures that must be stored in physical frames themselves!

2.4 The "CR3 Register"

In a real CPU, the `CR3` register holds the physical base address of the top-level Page Directory. We will simulate this with a global variable: `physaddr_t page_directory_base_reg;`. This is the single most important pointer in our MMU.

2.5 Free Frame Management

We need to know which of our 16 physical frames are free. A simple bitmap is perfect for this. For simplicity, we will keep the free frame bitmap outside of the simulated physical memory.

3 Your Task

You will implement a set of functions in `mmu.c`. A header file `mmu.h` and a testing file `main.c` are provided.

3.1 Part 1: Initialization & Frame Management

- **Implement `void mmu_init()`:**
  1. Use `malloc` to allocate the `PHYS_MEM_SIZE` block of memory.
  2. Initialize the `free_frame_bitmap`, marking all 16 frames as free.
  3. Allocate a frame for the Page Directory using `alloc_frame()`. This is the first, most crucial allocation.
  4. Store the physical address of this frame in `page_directory_base_reg`.
  5. Access the newly allocated Page Directory frame and initialize all its 32 entries to be invalid.
- **Implement `physaddr_t alloc_frame()`:**
  1. Search the `free_frame_bitmap` for the first available frame.
  2. If none is found, return `(physaddr_t)-1`.
  3. If found, mark it as used in the bitmap.
  4. Calculate and return the physical address of the start of that frame (i.e., `frame_number * PAGE_SIZE`).
- **Implement `void free_frame(physaddr_t p_addr)`:**
  1. Calculate the frame number from the given physical address.
  2. Mark that frame as free in the bitmap.

3.2 Part 2: The Core MMU Logic

- **Implement `int mmu_map_page(virtual_addr_t v_addr, physaddr_t p_addr)`:**
  1. Parse `v_addr` to get the Page Directory Index (PDI) and Page Table Index (PTI).
  2. Find the Page Directory in memory using `page_directory_base_reg`.

3. Read the Page Directory Entry (PDE) at index PDI.
4. If the PDE is not valid, it means a Page Table for this address range doesn't exist yet. You must:
   - Call `alloc_frame()` to get a new frame for the Page Table.
   - If allocation fails, return -1.
   - Initialize this new Page Table frame (set all its PTEs to invalid).
   - Update the PDE in the Page Directory: mark it as valid and set its address to the physical address of the new Page Table frame.
5. Get the physical address of the Page Table from the now-valid PDE.
6. Access the Page Table and find the Page Table Entry (PTE) at index PTI.
7. Update the PTE: mark it as valid and set its address to `p_addr`.
8. Return 0 on success.

- **Implement `physaddr_t mmu_translate_address(virtual_addr_t v_addr)`:**

1. Parse `v_addr` to get the PDI, PTI, and offset.
2. Read the PDE at index PDI from the Page Directory. If invalid, return `(physaddr_t)-1`.
3. Get the Page Table's physical address from the PDE.
4. Read the PTE at index PTI from the Page Table. If invalid, return `(physaddr_t)-1`.
5. Get the data frame's physical base address from the PTE.
6. Calculate and return the final physical address by adding the `offset`.

3.3 Part 3: Memory Access

- **Implement `void mmu_write(virtual_addr_t v_addr, uint8_t value)`:**

1. Call `mmu_translate_address()` to get the physical address.
2. If translation fails, print an error.
3. If it succeeds, write the `value` to the correct location within your `malloc`'d memory block.

- **Implement `uint8_t mmu_read(virtual_addr_t v_addr)`:**

1. Call `mmu_translate_address()` to get the physical address.
2. If translation fails, print an error and return 0.
3. If it succeeds, read and return the byte from that location.

4 Getting Started: Code Template

4.1 mmu.h

```c
#ifndef MMU_H
#define MMU_H

#include <stdint.h>
#include <stddef.h>

#define PHYS_MEM_SIZE (64 * 1024)
#define VIRT_ADDR_SPACE_SIZE (4 * 1024 * 1024)
#define PAGE_SIZE 4096
```

```c

#define NUM_FRAMES (PHYS_MEM_SIZE / PAGE_SIZE)
#define NUM_PAGES (VIRT_ADDR_SPACE_SIZE / PAGE_SIZE)

#define PDI_BITS 5
#define PTI_BITS 5
#define OFFSET_BITS 12

#define PDI_SHIFT (PTI_BITS + OFFSET_BITS)
#define PTI_SHIFT (OFFSET_BITS)

#define PDI_MASK ((1 << PDI_BITS) - 1)
#define PTI_MASK ((1 << PTI_BITS) - 1)
#define OFFSET_MASK ((1 << OFFSET_BITS) - 1)

typedef uintptr_t physaddr_t;
typedef uint32_t virtual_addr_t;

typedef struct {
    physaddr_t frame_base_addr;
    int present;
} pte_t;

void mmu_init();
void mmu_shutdown();

physaddr_t alloc_frame();
void free_frame(physaddr_t p_addr);

int mmu_map_page(virtual_addr_t v_addr, physaddr_t p_addr);
physaddr_t mmu_translate_address(virtual_addr_t v_addr);

void mmu_write(virtual_addr_t v_addr, uint8_t value);
uint8_t mmu_read(virtual_addr_t v_addr);

#endif
```

Listing 1: Header file: mmu.h

4.2 mmu.c (Your Implementation File)

```c
#include "mmu.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define static variables for MMU state here.

void mmu_init() {

}

void mmu_shutdown() {

}

physaddr_t alloc_frame() {
```

```
17
18  }
19
20  void free_frame(physaddr_t p_addr) {
21
22  }
23
24  int mmu_map_page(virtual_addr_t v_addr, physaddr_t p_addr) {
25
26  }
27
28  physaddr_t mmu_translate_address(virtual_addr_t v_addr) {
29
30  }
31
32  void mmu_write(virtual_addr_t v_addr, uint8_t value) {
33
34  }
35
36  uint8_t mmu_read(virtual_addr_t v_addr) {
37
38  }
```

Listing 2: Implementation file: mmu.c

4.3 main.c (Testing File)

```
1  #include "mmu.h"
2  #include <stdio.h>
3  #include <assert.h>
4
5  // Test case for mapping a single page and performing read/write operations
       .
6  void test_simple_mapping() {
7      printf("\n--- Running Test: Simple Mapping ---\n");
8      // your implementation
9      printf("Test Simple Mapping: To be implemented.\n");
10 }
11
12 // Test case for ensuring that new page tables are created on demand.
13 void test_new_page_table_creation() {
14     printf("\n--- Running Test: New Page Table Creation ---\n");
15     //your implementation
16     printf("Test New Page Table Creation: To be implemented.\n");
17 }
18
19 int main() {
20     test_simple_mapping();
21     test_new_page_table_creation();
22
23     printf("\nLab finished. Implement tests to verify correctness.\n");
24
25     return 0;
26 }
```

Listing 3: Testing file: main.c

5 Problem: Page Replacement

When `alloc_frame()` fails because memory is full, instead of returning an error, implement a page replacement algorithm that swaps out the least recently used page from the memory.