OS Practice Assignment 2

---

1 Memory Virtualization and Segmentation

1.1 MMU Simulation with Base and Bounds Registers

Write a C program that simulates the address translation mechanism of a Memory Management Unit (MMU) using a single pair of base and bounds registers. This mechanism, known as dynamic relocation, protects processes from accessing memory outside their allocated address space.

Your program must implement a function *translate_address* that takes a virtual address, a base register value, and a bounds (or limit) register value.

- The function should first check if the virtual address is valid by comparing it against the bounds register. A virtual address VA is valid if $0 \leq \text{VA} < \text{bounds}$.
- If the address is valid, the function should calculate the physical address using the formula: $\text{PA} = \text{base} + \text{VA}$.
- If the address is invalid, it should indicate a segmentation fault.

Your main function should initialize a base and bounds register for a simulated process and then attempt to translate a given list of virtual addresses, printing the result for each.

int translate_address(int virtual_address, int base, int bounds, int physical_address);*

- **Returns:** 0 on successful translation, -1 on a fault.
- **physical_address:** An output parameter to store the result.

Given: base = 3000; bounds = 1000; Virtual Addresses to translate: 100, 999, 1000, -5

Expected Output:

```
Virtual Address 100 : Physical Address 3100
Virtual Address 999 : Physical Address 3999
Virtual Address 1000 : Segmentation Fault: Address out of bounds
Virtual Address -5 : Segmentation Fault: Address out of bounds
```

```c
#include <stdio.h>

int translate_address(int virtual_address, int base, int bounds,
int* physical_address) {
    // 1. Bounds Check: The virtual address must be within the legal range.
    // The range is [0, bounds-1].
    if (virtual_address >= 0 && virtual_address < bounds) {
        // 2. Translation: If valid, add the base address.
        *physical_address = base + virtual_address;
        return 0; // Success
    }
    // 3. Fault: If the address is outside the bounds, it's a fault.
    return -1; // Failure
}

int main() {
    int base_reg = 3000;
    int bounds_reg = 1000;
    int virtual_addresses[] = {100, 999, 1000, -5};
    int num_addresses = sizeof(virtual_addresses) /
    sizeof(virtual_addresses[0]);
    printf("Base Register: %d\nBounds Register: %d\n\n",
    base_reg, bounds_reg);

    for (int i = 0; i < num_addresses; i++) {
        int current_va = virtual_addresses[i];
        int pa_result; // To store the physical address

        if (translate_address(current_va, base_reg, bounds_reg,
        &pa_result) == 0) {
            printf("Virtual Address %d : Physical Address %d\n",
            current_va, pa_result);
        } else {
            printf("Virtual Address %d : Segmentation Fault: Address
            out of bounds\n", current_va);
        }
    }
    return 0;
}
```

1.2 MMU Simulation: Segmentation with Protection Flags

Write a C program that simulates address translation for a system using segmentation, where each segment also has access permissions. The MMU must not only check if an address is within a segment's bounds but also if the attempted memory access (e.g., read vs. write) is allowed.

Assume a 16-bit virtual address format where: The top 2 bits represent the segment number (0 to 3); The lower 14 bits represent the offset.
Your task is to process a series of memory access requests. Each request consists of an access type ('R' for Read, 'W' for Write) and a virtual address. Your simulation must:

1. Parse the virtual address to get the segment number and offset.

2. Look up the segment's base, limit, and permissions from a segment table.
3. Perform two checks: **Bounds Check:** Is the offset less than the segment's limit?; **Permission Check:** Is the access type (Read/Write) permitted for this segment?
4. If both checks pass, compute the physical address. Otherwise, report the specific error.

Data Structures:

```
#include <stdio.h>
#define READ_PERMISSION  (1 << 0) // Represents bit 0
#define WRITE_PERMISSION (1 << 1) // Represents bit 1
typedef struct {
    int base;
    int limit;
    char perms; // Bitmask for permissions
} SegmentEntry;
```

Given a Segment Table:
Segment 0 (Code): base = 2048, limit = 1024, perms = READ_PERMISSION
Segment 1 (Heap): base = 4096, limit = 2048, perms = READ_PERMISSION | WRITE_PERMISSION
Segment 2 (Stack): base = 8192, limit = 1024, perms = READ_PERMISSION | WRITE_PERMISSION
Segment 3: Unused/Invalid
And Memory Access Requests:
'R', VA = 500 (Read from Code Segment)
'W', VA = 600 (Write to Code Segment)
'W', VA = 17000 (Write to Heap Segment, Offset 616)
'R', VA = 20000 (Read from Heap Segment, Offset 3616)
Expected Output:

```
Access: READ, VA: 500 : Success! PA: 2548
Access: WRITE, VA: 600 : Protection Fault: WRITE permission denied for Segment 0
Access: WRITE, VA: 17000 : Success! PA: 4712
Access: READ, VA: 20000 : Segmentation Fault: Offset 3616 is out of bounds for Segment1
```

```
#include <stdio.h>
// Use bitmasks for permissions
#define READ_PERMISSION  (1 << 0) // Represents bit 0, value = 1
#define WRITE_PERMISSION (1 << 1) // Represents bit 1, value = 2
// Structure for a segment table entry
typedef struct {
    int base;
    int limit;
    char perms; // Bitmask for permissions (R/W)
} SegmentEntry;
// Structure to represent a memory access request
typedef struct {
    char type; // 'R' for Read, 'W' for Write
    int virtual_address;
} AccessRequest;
int main() {
    // Initialize the segment table for the process
    SegmentEntry segment_table[4] = {
        {2048, 1024, READ_PERMISSION},
        // Segment 0 (Code)
        {4096, 2048, READ_PERMISSION | WRITE_PERMISSION},
        // Segment 1 (Heap)
        {8192, 1024, READ_PERMISSION | WRITE_PERMISSION},
        // Segment 2 (Stack)
        {0, 0, 0}
        // Segment 3 (Unused)
    };
    // List of memory accesses to simulate
    AccessRequest requests[] = {
        {'R', 500},
        {'W', 600},
        {'W', 17000},
        {'R', 20000}
    };
    int num_requests = sizeof(requests) / sizeof(requests[0]);
    for (int i = 0; i < num_requests; i++) {
        char access_type = requests[i].type;
        int va = requests[i].virtual_address;
        // 1. Parse the 16-bit virtual address
        int segment_num = (va >> 14) & 0x03;
        int offset = va & 0x3FFF;
        printf("Access: %s, VA: %-5d -> ", (access_type == 'R' ? "READ"
        : "WRITE"), va);
        // Check if segment is valid
        if (segment_num > 3 || segment_table[segment_num].limit == 0) {
            printf("Segmentation Fault: Invalid segment
            number %d\n", segment_num);
            continue; }
        SegmentEntry segment = segment_table[segment_num];
        // 2. Bounds Check
        if (offset >= segment.limit) {
            printf("Segmentation Fault: Offset %d is out of bounds
            for Segment %d\n", offset, segment_num);
            continue; }
        // 3. Permission Check
        int has_permission = 0;
        if (access_type == 'R' && (segment.perms & READ_PERMISSION)) {
            has_permission = 1;
        } else if (access_type == 'W' && (segment.perms & WRITE_PERMISSION))
            has_permission = 1;
        }
        if (!has_permission) {
            printf("Protection Fault: %s permission denied for Segment
            %d\n", (access_type == 'R' ? "READ" : "WRITE"), segment_num);
            continue; }
        // 4. Translation
        int physical_address = segment.base + offset;
        printf("Success! PA: %d\n", physical_address); }
    return 0;  }
```

2 Paging

2.1 Single-Level Page Table Translation

Implement a C function unsigned int get_physical_addr(unsigned int virtual_addr) that translates a virtual address to a physical address using a single-level page table.
Specifications:

- A 16-bit virtual address space.
- A 4KB (2^{12} bytes) page size.
- The virtual address is split into a 4-bit Virtual Page Number (VPN) and a 12-bit offset.
- You are given a pre-defined global array int page_table[16] which maps a VPN to a Physical Frame Number (PFN). For example, page_table[5] gives the PFN for VPN 5.

Task: Write a function to:

- Extract the VPN and the offset from the virtual_addr using bitwise operations.
- Look up the PFN in the page_table using the VPN.
- Construct and return the final physical address by combining the PFN and the offset.

```
#include <stdio.h>
// System Specifications and pre-populated page table
#define VPN_SHIFT 12
#define PFN_SHIFT 12
#define OFFSET_MASK 0xFFF
int page_table[16] = {
    2, 4, 1, 7, 3, 5, 6, 8, // VPN 0-7
```

```c
    10, 11, 12, 13, 14, 9, 15, 0 // VPN 8-15
};
unsigned int get_physical_addr(unsigned int virtual_addr) {
    // 1. Extract the VPN (top 4 bits)
    unsigned int vpn = virtual_addr >> VPN_SHIFT;
    // 2. Extract the offset (bottom 12 bits)
    unsigned int offset = virtual_addr & OFFSET_MASK;
    // 3. Look up the PFN in the page table
    unsigned int pfn = page_table[vpn];
    // 4. Construct the physical address
    // Shift the PFN left to make room for the offset, then add the offset
    unsigned int physical_addr = (pfn << PFN_SHIFT) | offset;
    printf("VA: 0x%04X -> (VPN: %u, Offset: 0x%03X) -> PFN: %u -> PA:
    0x%05X\n",
           virtual_addr, vpn, offset, pfn, physical_addr);
    return physical_addr; }
int main() {
    // Example: Virtual Address 0x4ABC
    // VPN = 0x4 = 4
    // Offset = 0xABC
    // Physical Address = (3 << 12) | 0xABC = 0x3000 | 0xABC = 0x3ABC
    unsigned int va1 = 0x4ABC;
    get_physical_addr(va1);
    // Example 2: Virtual Address 0x0123
    // VPN = 0x0 = 0
    // Offset = 0x123
    // Physical Address = (2 << 12) | 0x123 = 0x2000 | 0x123 = 0x2123
    unsigned int va2 = 0x0123;
    get_physical_addr(va2);
    return 0; }
```

3 Swapping

3.1 Demand Paging and Swapping Mechanism

Create a C program to simulate a demand paging memory management system that handles page faults and swapping.

Specifications:

- The program should be initialized with a fixed number of pages in the logical address space and a smaller number of frames in physical memory.
- You must implement a page table as an array of structs. Each entry in the page table must contain at least: A frame number; A present bit (or valid-invalid bit) to indicate

if the page is currently in a physical memory frame.

- The program will process a sequence of page access requests. For each request: If the page's present bit is 1, it's a hit; If the present bit is 0, a page fault has occurred.
- On a page fault:
  - The system must find a free frame.
  - If memory is full (no free frames), you must implement the LRU policy to select a victim page to be swapped out. Swapping out involves updating the victim page's present bit to 0 in the page table.
  - The required page is then swapped into the newly freed frame. This involves updating the new page's frame number and setting its present bit to 1.
- Your program must count and report the total number of page faults that occur during the simulation and display the final state of the page table.

Example Input:
Total pages: 8
Total frames: 4
Access sequence: 0 1 2 3 0 1 4 0 1 2 3 4
Example Output:

```
Accessing 0 : Page Fault
Accessing 1 : Page Fault
Accessing 2 : Page Fault
Accessing 3 : Page Fault
Accessing 0 : Hit
Accessing 1 : Hit
Accessing 4 : Page Fault (Swapping out page 2)
Accessing 0 : Hit
Accessing 1 : Hit
Accessing 2 : Page Fault (Swapping out page 3)
Accessing 3 : Page Fault (Swapping out page 0)
Accessing 4 : Hit

Total Page Faults: 8
Final Page Table State:
Page 1: Frame 1
Page 2: Frame 3
Page 3: Frame 2
Page 4: Frame 0
```

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>

typedef struct {
    int frame_number;
    int present_bit;
} PageTableEntry;

void print_page_table(PageTableEntry page_table[], int total_pages) {
    printf("\n--- Final Page Table State ---\n");
    printf("Page | Frame | Present\n");
    printf("-----|-------|---------\n");
    for (int i = 0; i < total_pages; i++) {
        if (page_table[i].present_bit) {
            printf("%-4d | %-5d | %-7d\n", i,
            page_table[i].frame_number, page_table[i].present_bit);
        } else {
            printf("%-4d | %-5s | %-7d\n", i, "N/A",
            page_table[i].present_bit);
        }
    }
}

int main() {
    int total_pages, total_frames;
    char access_seq_input[256];
    int access_sequence[100];
    int n = 0;

    printf("Enter total number of pages: ");
    scanf("%d", &total_pages);
    printf("Enter total number of frames: ");
    scanf("%d", &total_frames);
    getchar();

    printf("Enter access sequence (space-separated integers):\n");
    fgets(access_seq_input, sizeof(access_seq_input), stdin);

    char *token = strtok(access_seq_input, " \n");
    while (token != NULL) {
        access_sequence[n++] = atoi(token);
        token = strtok(NULL, " \n");
    }

    if (n == 0) return 1;
```

```
PageTableEntry *page_table = (PageTableEntry*)malloc(total_pages *
sizeof(PageTableEntry));
int *frames = (int*)malloc(total_frames * sizeof(int)); // Stores
which page is in the frame
int *frame_timestamps = (int*)malloc(total_frames * sizeof(int));
// For LRU
int page_faults = 0;
int free_frame_ptr = 0;

for (int i = 0; i < total_pages; i++) {
    page_table[i].frame_number = -1;
    page_table[i].present_bit = 0;
}
for (int i = 0; i < total_frames; i++) {
    frames[i] = -1; // -1 indicates a free frame
}

for (int t = 0; t < n; t++) {
    int page_num = access_sequence[t];

    if (page_num >= total_pages || page_num < 0) {
        printf("Accessing invalid page %d. Skipping.\n", page_num);
        continue;
    }

    printf("Accessing page %d -> ", page_num);

    if (page_table[page_num].present_bit == 1) { // It's a HIT
        printf("Hit\n");
        frame_timestamps[page_table[page_num].frame_number] = t + 1;
    } else {
        page_faults++;
        if (free_frame_ptr < total_frames) {
            printf("Page Fault. Using free frame %d.\n", free_frame_ptr);
            frames[free_frame_ptr] = page_num;
            frame_timestamps[free_frame_ptr] = t + 1;
            page_table[page_num].frame_number = free_frame_ptr;
            page_table[page_num].present_bit = 1;
            free_frame_ptr++;
        } else {
            int lru_frame_index = 0;
            for (int i = 1; i < total_frames; i++) {
                if (frame_timestamps[i] <
                frame_timestamps[lru_frame_index]) {
                    lru_frame_index = i; } }
            int victim_page = frames[lru_frame_index];
            printf(”Page Fault. (Swapping out page %d from frame %d)\n”,
            page_table[victim_page].present_bit = 0;

            frames[lru_frame_index] = page_num;
            frame_timestamps[lru_frame_index] = t + 1;
            page_table[page_num].frame_number = lru_frame_index;
            page_table[page_num].present_bit = 1;
        } } }
printf(”\nTotal Page Faults: %d\n”, page_faults);
print_page_table(page_table, total_pages);
free(page_table);
free(frames);
free(frame_timestamps);
return 0; }
```