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
```

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:

```c
#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 512)
'R', VA = 20000 (Read from Heap Segment, Offset 3608)
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: 4608
Access: READ, VA: 20000 : Segmentation Fault: Offset 3608 is out of bounds for Segment1
```

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.

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: 7
Final Page Table State:
Page 0: Not Present
Page 1: Frame X
Page 2: Frame Y
Page 3: Frame Z
Page 4: Frame W
... and so on
```