Assignment-1: Paging and Memory Sharing - Simulating Shared Libraries

Objective: To understand and simulate how paging allows different processes to share common code or data (like a shared library) in physical memory, thus saving space and improving efficiency.

Problem:

Write a program that simulates a memory management unit (MMU) for a system with multiple processes that share a common code segment. The program will have descriptions of several processes, including the size of their private data and an indication that they use a shared library. Construct the page tables for each process and manage the physical frame table to correctly map the shared library pages to the same physical frames for all processes that use it.

Implementation Guidelines:

1. **Data Structures:**
    - **Page Table Entry (PTE):** A structure containing at least a FrameNumber and a ValidBit.
    - **Page Table:** An array of PTEs for each process.
    - **Frame Table:** A global array representing physical memory frames. Each entry should track if the frame is free or, if allocated, which process and page number occupy it.
2. **Input:**
    - The total number of physical memory frames.
    - The size of the shared library (in pages).
    - A list of processes, each with a size for its private data segment (in pages).
3. **Logic:**
    - First, allocate contiguous physical frames for the entire shared library. Mark these frames as occupied in your frame table. This only happens once.
    - For each process:
        - Create a new page table for it.
        - Allocate physical frames for its private data pages from the remaining free frames.
        - Fill the page table of each process

- Entries for the private data pages should map to the newly allocated unique frames.
- Entries for the shared library pages should map to the common frames allocated in the first step.

4. **Address Translation:** Include a function that, given a process ID and a logical address, can translate it to the corresponding physical address by looking up the correct page table.

Output:

1. A printout of the final **Page Table** for each process.
2. A printout of the final state of the **Physical Memory Frame Table**.
3. The output must clearly demonstrate that the logical pages corresponding to the shared library in different processes all map to the *exact same physical frame numbers*. For example:

   Page Table for Process P1

   Page 0 (Data) -> Frame 7

   Page 1 (Lib) -> Frame 2

   Page 2 (Lib) -> Frame 3

   Page Table for Process P2

   Page 0 (Data) -> Frame 9

   Page 1 (Data) -> Frame 11

   Page 2 (Lib) -> Frame 2

   Page 3 (Lib) -> Frame 3

Assignment-2: Memory Protection using Page Tables

Objective: To simulate how the hardware uses protection bits (e.g., Read, Write, Execute) in a page table to enforce memory access rights and generate a protection fault on invalid access attempts.

Problem Statement:

Write a program that simulates logical-to-physical address translation while enforcing memory protection. Input is an initial page table for a process, where each page is assigned specific permissions (Read, Write, Execute). Your program then process a sequence of

memory access requests. For each request, it must determine if the access is permitted. If it is, translate the address; otherwise, report a "Protection Fault".

Implementation Guidelines:

1. **Data Structures:**
    - Modify your **Page Table Entry (PTE)** structure to include three boolean flags: CanRead, CanWrite, CanExecute.
2. **Input:**
    - The page size (e.g., 4KB or 4096 bytes).
    - The initial page table configuration, mapping logical pages to physical frames and setting their protection bits. (e.g., Page 0 is R-X, Page 1 is R-W).
    - A sequence of memory access requests, where each request specifies an access type (READ, WRITE, EXECUTE) and a logical address.
3. **Logic:** For each memory access request:
    - Parse the request to get the access type and logical address.
    - Calculate the logical page number (p) and offset(d) from the address.
    - Use the page number to look up the corresponding PTE in the page table.
    - Perform the check:
        - If the access type is READ, check if the CanRead bit is set.
        - If the access type is WRITE, check if the CanWrite bit is set.
        - If the access type is EXECUTE, check if the CanExecute bit is set.
    - If the check passes, calculate the physical address (Frame Number * Page Size + Offset) and print it.
    - If the check fails, print a clear "PROTECTION FAULT" error message describing the violation (e.g., "Attempted to write to a read-only page").

Output:

For each request in the input sequence, your program should produce one line of output.

- If access is valid: READ at 0x1A0F -> Physical Address 0x3A0F
- If access is invalid: WRITE at 0x03FF -> VIOLATION: PROTECTION FAULT (Attempt to write to a non-writable page)