Virtual Memory Manager Design Document

Overview

MemoryManager is a simple implementation of a virtual memory system. It features demand paging and page fault handling with a simulated disk storage.

Personal Background

I chose to implement a virtual memory simulation because while I did generally understand the lecture material, I felt like I could gain a better understanding of how virtual memory worked by implementing it myself. I used C++ as I like it's flexibility and built-in libraries over C.

Core Architecture

Memory Hierarchy

- **Virtual Memory Space**: Divided into fixed-size pages
- **Physical Memory**: Divided into frames of same size as pages
- **Disk Storage**: Backing store for pages not currently in physical memory

Key Components

Page Table Entry Structure

```
struct pageTableEntry {
    bool validBit = false;      // Whether the page is allocated/valid
    bool presentBit = false;    // Whether page is in physical memory
    bool referenceBit = false;  // Used for page replacement algorithm
    bool modifyBit = false;     // Whether page has been modified (dirty bit)
    int pageFrameNum = -1;      // Physical frame number if present
};
```

Memory Manager Class Members

- `pageTable`: Vector of page table entries mapping virtual to physical pages
- `physicalMemory`: Byte-array representing physical RAM
- `freeFrames`: Boolean vector tracking available physical frames
- `diskStorage`: Simulated disk storage for swapped-out pages
- Configuration constants: `PAGE_SIZE`, `PAGE_COUNT`, `PHYSICAL_SIZE`
- `clockPointer`: For CLOCK page replacement algorithm

Configuration

Default Configuration

- Page Size: 4,096 bytes (4KB)
- Page Count: 1,024 entries
- Physical Memory: 1,024 frames (4MB total)
- Virtual Address Space: 1,024 pages (4MB total)

Custom Configuration

Supports custom page sizes, page table sizes, and physical memory frames through constructor parameters.

Core Algorithms

Virtual to Physical Address Translation

Process:

1. Extract virtual page number and offset from virtual address
2. Validate virtual page number bounds
3. Check page table entry validity
4. Handle page fault if page not present in memory
5. Calculate physical address using frame number and offset
6. Update reference and modify bits

Formula:

```
offset = virtualAddress & (PAGE_SIZE - 1)
virtualPageNumber = virtualAddress / PAGE_SIZE
physicalAddress = (frameNumber * PAGE_SIZE) + offset
```

Page Fault Handling

Trigger: Access to valid but not-present page

Handling Process:

1. Find free physical frame
2. If no free frames, utilize page replacement algorithm
3. Load requested page from disk into free frame
4. Update page table entry
5. Mark frame as no longer free

Page Replacement (CLOCK Algorithm)

Used when: No free physical frames available

Algorithm Steps:

1. Iterate through page table entries using clock pointer
2. Skip invalid or not-present pages
3. If reference bit is set, clear it and continue
4. If reference bit is clear, select this page for replacement
5. If page was modified, write it back to disk
6. Clear the frame and return it for reuse

Public Methods

Memory Allocation

- `allocateAnyPage()`: Finds a free page table entry and physical frame, creates page table entry, and finally returns a virtual address. Utilizes page replacement if there are no free physical frames

Memory Access

- `writeVirtualMemory()`: Write data to virtual address
- `readVirtualMemory()`: Read data from virtual address

Memory Management

- `deletePageTableEntry()`: Deallocates a page and frees associated resources
- `printPageTableEntry()`: Prints basic debug information about a specific page

Internal/Private Methods

Memory Initialization

- Resizes all vectors based on configuration
- Marks all frames as free
- Initializes disk storage space

Frame Management

- `_wipeMemoryFrame()`: Zeros out physical frame contents
- `_allocatePage()`: Maps virtual page to physical frame

Disk Operations

- `_writePageToDisk()`: Copies page from memory to disk
- `_readPageFromDisk()`: Copies page from disk to memory
- `_deletePageFromDisk()`: Clears contents of a disk-stored page

Error Handling

The system throws exceptions for:

- Out-of-bounds memory accesses
- Invalid page accesses (segmentation fault simulation)
- Allocation failures
- Invalid operations on memory regions

Usage Flow

1. **Initialization**: Configure `MemoryManager()` with default or desired parameters
2. **Allocation**: Request virtual pages via `allocateAnyPage()`
3. **Access**: Read/write to virtual addresses (triggers page faults as needed)
4. **Management**: Delete pages when no longer needed