Operating System

Homework 1

Ziya Kadir TOKLUOGLU

210104004228

Virtual Memory

A screen shot of a computer program

Description automatically generated

1. **Constructor**: Initializes the virtual memory setup, links the page table to the physical memory, and sets up mutexes for thread-safe operations.
2. **Memory Operations**:
   * **set()**: Writes a value to a specified virtual memory index, handling page faults and replacements.
   * **get()**: Retrieves a value from a specified index, also managing page faults.
   * **fill()**: Fills virtual memory with random values, ensuring thread safety.
3. **Sorting and Searching**:
   * **Merge Sort**: Recursively sorts the virtual memory using subarray instances of VirtualMemory, aggregating page fault and replacement statistics.
   * **Binary Search**: Searches for a value in the sorted virtual memory using efficient lookups.
4. **Synchronization**: Mutexes protect critical sections, preventing race conditions during concurrent memory access by multiple threads.
5. **Statistics Management**: During sorting, memory statistics like page faults and disk operations are tracked and aggregated.

Pyhsical MemoryA screen shot of a computer program

Description automatically generated

**1. Class Purpose**

The PhysicalMemory class stores and manages the actual pages in the system. It keeps track of which pages are loaded, handles page replacement using LRU and Clock algorithms, and interacts with the disk for swapping pages in and out.

**2. Constructor and Destructor**

* **Initialization:** Sets up the memory capacity, page size, and the name of the disk file where pages are stored when swapped out.
* **Destructor:** Cleans up resources when the class instance is destroyed.

**3. Page Replacement Algorithms**

* **off\_t LRU\_page\_remove()**: Implements the Least Recently Used (LRU) algorithm to identify the page to be removed when the physical memory is full. If the page is modified, it is written back to the disk.
* **off\_t CL\_page\_remove()**: Implements the Clock algorithm to select pages for removal. It checks the reference bit of each page and resets it during each pass. If no unreferenced pages are found, it picks any available page.

**4. Memory Management**

* **Page& operator[](off\_t\* address)**: Overloaded subscript operator manages fetching, loading, and swapping pages between physical memory and disk. It checks if a page is already loaded; if not, it handles fetching the page from the disk or creating a new page, managing page replacements when necessary.
* **bool is\_full() const**: Checks if the physical memory is at capacity.

**5. Synchronization**

Although not directly included in this class, the operations performed by this class would typically be synchronized by mutexes in a multi-threaded environment, ensuring that multiple threads do not access or modify shared resources concurrently.

**6. Disk Interaction**

The class interacts with disk storage to read and write pages:

* **Writing pages to disk** when removed from memory (if modified).
* **Loading pages from disk** when needed.

**7. Supporting Functions**

* **void clean\_R\_bit()**: Resets the reference bits for all pages, useful for page replacement algorithms like Clock.
* **void set(off\_t address, PageTableEntry\* page\_table\_entry)**: Maps a page table entry to a specific memory address.
* **PageTableEntry\* get(off\_t address)**: Retrieves the page table entry associated with a specific address.

Page Table Entry

A screen shot of a computer program

Description automatically generated

**1. Class Purpose**

The PageTableEntry class holds key information about each page in the virtual memory, including its physical address, access history, and status flags. It plays a critical role in page replacement algorithms and tracking page access patterns.

**2. Member Variables**

* **off\_t address**: Stores the physical address where the page is loaded.
* **chrono::steady\_clock::time\_point last\_accessed**: Records the last time the page was accessed, crucial for LRU page replacement.
* **int thread\_number**: Identifies the thread associated with this page.
* **bool referenced**: Indicates whether the page was recently accessed, used in the Clock algorithm.
* **bool modified**: Tracks whether the page has been modified since being loaded, ensuring it is written back to disk if needed.
* **PageTable\* page\_table**: A pointer to the page table this entry belongs to.

**3. Key Methods**

* **void setAddress(off\_t address)**: Sets the physical address for the page.
* **void setLastAccessed(chrono::steady\_clock::time\_point last\_accessed)**: Updates the last access time for LRU tracking.
* **chrono::milliseconds duration(chrono::steady\_clock::time\_point current\_time) const**: Calculates the time since the page was last accessed, used in LRU decisions.
* **bool isReferenced() const and void setReferenced(bool referenced)**: Get and set the referenced bit, key for the Clock algorithm.
* **bool isModified() const and void setModified(bool modified)**: Get and set the modified bit, determining if the page needs to be written back to disk.

**4. Utility Functions**

* **Overloaded << Operator**: Provides a simple way to print the page table entry's details, useful for debugging and logging.

**5. Constructor and Destructor**

* **PageTableEntry() and PageTableEntry(PageTable\* page\_table)**: Initialize the entry with default values or associate it with a specific page table.
* **Destructor**: Ensures proper cleanup of any resources, though no explicit dynamic memory management is involved here.

Page Table

A screen shot of a computer program

Description automatically generated

**1. Class Purpose**

The PageTable class maintains the page table entries for a particular thread, allowing translation from virtual memory addresses to physical memory addresses. It tracks page access patterns and handles operations like setting and getting values within pages, while managing page replacement and fault handling.

**2. Key Attributes**

* **PhysicalMemory\* physical\_memory**: Pointer to the physical memory that stores the actual data.
* **unordered\_map<int, PageTableEntry> table**: A map representing the page table, where each entry corresponds to a virtual page.
* **int page\_entry\_capacity and int page\_entry\_count**: Track the capacity and current count of page entries in the table.
* **Statics statics**: Holds statistics related to page faults, replacements, disk reads/writes, etc.

**3. Important Methods**

* **PageTableEntry& operator[](int page\_number)**: Provides direct access to a page table entry using the page number.
* **void set(int page\_number, int index, int value)**: Sets a value in the specified page and index. Handles page faults, page loading, and updating relevant statistics.
* **int get(int page\_number, int index)**: Retrieves a value from the specified page and index, managing page faults and updating access times and statistics.
* **bool is\_full() const**: Checks if the page table is full.
* **void clean\_R\_bit()**: Resets the reference bits of all pages, typically used in page replacement algorithms.

**4. Page Replacement Handling**

* The set and get methods incorporate logic to handle page faults, loading pages from disk, and updating reference and modified bits. They also interact with the PhysicalMemory class for page replacements, ensuring that the correct page is loaded when required.

**5. Statistics and Logging**

* The class tracks statistics related to memory operations, including page faults, replacements, disk reads, and writes. These are essential for evaluating the performance of different page replacement algorithms.

**6. Constructor and Destructor**

* **PageTable(PhysicalMemory\* physical\_memory, unsigned int Thread\_number, int page\_entry\_capacity, int page\_entry\_size)**: Initializes the page table with the specified capacity and associates it with the physical memory.
* **Destructor**: Manages clean-up, though it does not involve dynamic memory allocation directly.

**7. Utility Functions**

* **Overloaded << Operator**: Provides a formatted output of the page table entries, useful for debugging and visualization.
* **Copy and Move Constructors/Assignment Operators**: Ensure proper copying or moving of the page table, especially when dealing with deep copies or transferring ownership.

Page

A screen shot of a computer program

Description automatically generated

**1. Class Purpose**

The Page class models a memory page, storing data with a fixed capacity. It manages reading, writing, and copying page data, which are essential for managing virtual-to-physical memory mappings in your system.

**2. Key Attributes**

* **int capacity**: The maximum number of elements the page can hold.
* **int size**: The current number of elements in the page.
* **int\* data**: A dynamic array holding the page’s data.

**3. Important Methods**

* **int operator[](int index) const and int& operator[](int index)**: Overloaded subscript operators provide direct access to elements in the page for both reading and writing.
* **bool is\_full() const**: Checks if the page is full.
* **bool is\_empty() const**: Checks if the page is empty.

**4. Memory Management**

* The class handles dynamic memory allocation for the data array, with constructors, destructors, and assignment operators ensuring proper memory management.
* **Copy and Move Semantics**: The class supports both deep copy and move operations. The copy constructor and assignment operator create a new data array and copy values, while the move constructor and assignment operator transfer ownership without copying.

**5. Utility Functions**

* **Overloaded << Operator**: Enables easy printing of the page’s data, useful for debugging and visualization.

**6. Constructor and Destructor**

* **Default Constructor**: Initializes the page with zero capacity and size.
* **Parameterized Constructor**: Allocates the data array based on the given capacity.
* **Copy and Move Constructors**: Handle deep copying or transferring ownership of the data array.
* **Destructor**: Ensures proper cleanup by deallocating the data array.

**Acknowledgment of Assistance**

During the development of this virtual memory management system, I sought assistance from ChatGPT, particularly for understanding and handling the implementation of the page table, page replacement algorithms (LRU and Clock), and memory management logic. ChatGPT's guidance helped clarify complex concepts and provided insights that were crucial for properly synchronizing and structuring the interactions between virtual memory and physical memory.