Practicum 1 - CS5600 - Spring
Shorena K. Anzhilov
Memory Hierarchy Simulation

This project is a simulation of hierarchical memory within a message-oriented data store. Due to the high volume of messages, not all messages can fit into main memory. Therefore, they must be stored in secondary memory (disk) and retrieved when needed. This simulation demonstrates the principles of virtual memory by implementing message creation, storage to disk, retrieval from disk, caching in memory, and page replacement strategies. The entire project is implemented in C and organized into separate modules for code clarity, modularity, and maintainability. I created two header files, three C source files, and a Makefile to compile the program efficiently.

In the 1th step: Memory Hierarchy Simulation (Baseline System).

I have designed and implemented a basic system to simulate hierarchical memory using disk storage. I began defining a Message structure in **message.h** that included important fields: a unique identifier, a timestamp (time_sent), sander and receiver names, content, and a delivery status flag. I then implemented related functions in the **message.c** to create messages dynamically (**`create_msg()`**), print them (**`print_msg()`**), and free allocated memory (**`free_msg()`**).
In **storage.h** and **storage.c**, I implemented the function to handle persistent storage. The **store_msg()** function writes messages to a file **(messages.dat)**, and the **retrieve_msg()** function reads messages from the file by searching for a specific ID. The main program, **`main.c`**, serves as a test driver by creating messages, storing them on disk, and retrieving them back to demonstrate that the system works correctly. The **`Makefile`** was created to automate compilation and cleanup, allowing for efficient development and testing.

In the 2nd Step: Adding a Cache!

In Part 1, messages were stored only on disk, requiring disk access every time a message was retrieved. Since disk access is slow, we are now implementing a cache that keeps recently accessed messages in fast memory (RAM). So we focused on improving performance by adding a cache in memory to reduce disk. I implemented a fixed-size in-memory cache that holds up to 16 messages, with each message sized to 1024 bytes. This cache is stored as an array and follows a paged structure similar to how operating systems cache pages in memory. Then to have efficient access, I implemented a **`cache_lookup()`** function that checks if a message is already in the cache before accessing the disk. If found in cache → return immediately (cache hit), If not found in cache in memory → load from disk using **`retrieve_msg()`**and insert into the cache with **`cache_insert()`** (cache miss). So in this step, we have a Cache Replacement Strategy (LRU) mechanism, when the cache is full, we remove the least recently used (LRU) message to make space for new ones. The process of LRU is the following: when a message is accessed, it moves to the front of the cache. The oldest message gets removed if the cache is full.

The caching mechanism improves performance by reducing the number of file reads when messages are accessed multiple times.

The 3rd Step is implementing Page Replacement Algorithms!

In the third step, I implemented two-page message replacement strategies to manage the cache when it becomes full. These strategies were implemented in **chache_insert() in storage.c.** The first strategy is Least Recently Used **(LRU)**, which evicts the message that has not been accessed for a longest time as already explained in the previous step. This is done by shifting the cache array and maintaining recently accessed messages at the front. The second strategy is **Random Replacement**, where a message is removed at random when a new one needs to be added. I added a global variable **`cache_replacement_policy`** that can be toggled between **`LRU_REPLACEMENT`** and **`RANDOM_REPLACEMENT`**, to have flexible testing of both strategies.

In the 4th Step: Evaluating Cache Performance!

The final step involved testing and evaluating the cache under different access patterns. I wrote a testing loop that performs 1000 random message accesses to simulate real-world usage. During each access, the program tracks whether the message is found in the cache (a cache hit) or must be loaded from disk (a cache miss) message is not in the cache. From these values, I calculated the cache hit ratio —-> Cache Hit Ratio (hits / total accesses). I ran the test for both LRU and Random Replacement strategies to compare their effectiveness. Results LRU provides a higher cache hit ratio when message access patterns are predictable. Random Replacement is less efficient but simpler to implement. This test confirms that LRU performs better for real-world workloads.

In Summary:

This project simulates the concept of hierarchical memory using a message-oriented data store. It demonstrates how messages are moved between disk and memory, how caching improves performance, and how different page replacement strategies affect system behavior. The project is modular, well-structured, and expandable for future work. It reflects real-world memory management concepts used in operating systems and computer architecture.