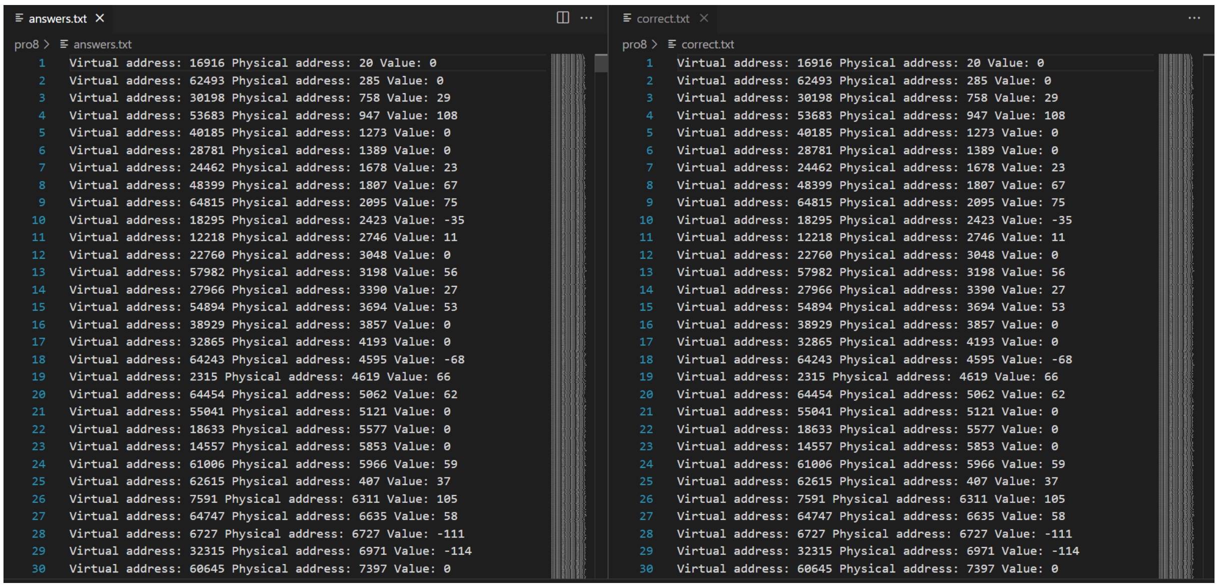
Project 8

Virtual Memory Manager

Introduction

In this project, I implemented a virtual memory manager that provides read and write access to a virtual address space. The main program uses the virtual memory manager to translate virtual addresses in file `addresses.txt` to physical addresses and read raw data in file `BACKING_STORE.bin`. The program then writes the virtual address, translated physical address, and the value stored at the physical address to file `answers.txt`.

The virtual memory manager:
- Use page table to translate virtual addresses to physical addresses.
- Use TLB to accelerate the translation.
- Use frame table to manage the physical memory.
- Use LRU algorithm to replace TLB entries when TLB miss occurs.
- Use LRU algorithm to replace frames when page fault occurs.
- Use dirty bit to track the modification of frames.
- Use an on-stack buffer to simulate the physical memory.



```
Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
Virtual address: 64815 Physical address: 2095 Value: 75
Virtual address: 18295 Physical address: 2423 Value: -35
Virtual address: 12218 Physical address: 2746 Value: 11
Virtual address: 22760 Physical address: 3048 Value: 0
Virtual address: 57982 Physical address: 3198 Value: 56
Virtual address: 27966 Physical address: 3390 Value: 27
Virtual address: 54894 Physical address: 3694 Value: 53
Virtual address: 38929 Physical address: 3857 Value: 0
Virtual address: 32865 Physical address: 4193 Value: 0
Virtual address: 64243 Physical address: 4595 Value: -68
Virtual address: 2315 Physical address: 4619 Value: 66
Virtual address: 64454 Physical address: 5062 Value: 62
Virtual address: 55041 Physical address: 5121 Value: 0
Virtual address: 18633 Physical address: 5577 Value: 0
Virtual address: 14557 Physical address: 5853 Value: 0
Virtual address: 61006 Physical address: 5966 Value: 59
Virtual address: 62615 Physical address: 407 Value: 37
Virtual address: 7591 Physical address: 6311 Value: 105
Virtual address: 64747 Physical address: 6635 Value: 58
Virtual address: 6727 Physical address: 6727 Value: -111
Virtual address: 32315 Physical address: 6971 Value: -114
Virtual address: 60645 Physical address: 7397 Value: 0
```

```
Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
Virtual address: 64815 Physical address: 2095 Value: 75
Virtual address: 18295 Physical address: 2423 Value: -35
Virtual address: 12218 Physical address: 2746 Value: 11
Virtual address: 22760 Physical address: 3048 Value: 0
Virtual address: 57982 Physical address: 3198 Value: 56
Virtual address: 27966 Physical address: 3390 Value: 27
Virtual address: 54894 Physical address: 3694 Value: 53
Virtual address: 38929 Physical address: 3857 Value: 0
Virtual address: 32865 Physical address: 4193 Value: 0
Virtual address: 64243 Physical address: 4595 Value: -68
Virtual address: 2315 Physical address: 4619 Value: 66
Virtual address: 64454 Physical address: 5062 Value: 62
Virtual address: 55041 Physical address: 5121 Value: 0
Virtual address: 18633 Physical address: 5577 Value: 0
Virtual address: 14557 Physical address: 5853 Value: 0
Virtual address: 61006 Physical address: 5966 Value: 59
Virtual address: 62615 Physical address: 407 Value: 37
Virtual address: 7591 Physical address: 6311 Value: 105
Virtual address: 64747 Physical address: 6635 Value: 58
Virtual address: 6727 Physical address: 6727 Value: -111
Virtual address: 32315 Physical address: 6971 Value: -114
Virtual address: 60645 Physical address: 7397 Value: 0
```

Figure 1: Correctness of the virtual memory manager

Overview

The virtual memory manager consists of the following components:

- LRU cache: a generic LRU cache implemented using a doubly linked list.
- TLB: a cache that stores the most recent translations from virtual addresses to physical addresses.
- Physical memory: a memory manager that provides block read and write operations.
- Virtual memory: a memory manager that implements the page table and integrates the TLB and physical memory.

The structure of the virtual memory manager is shown below:

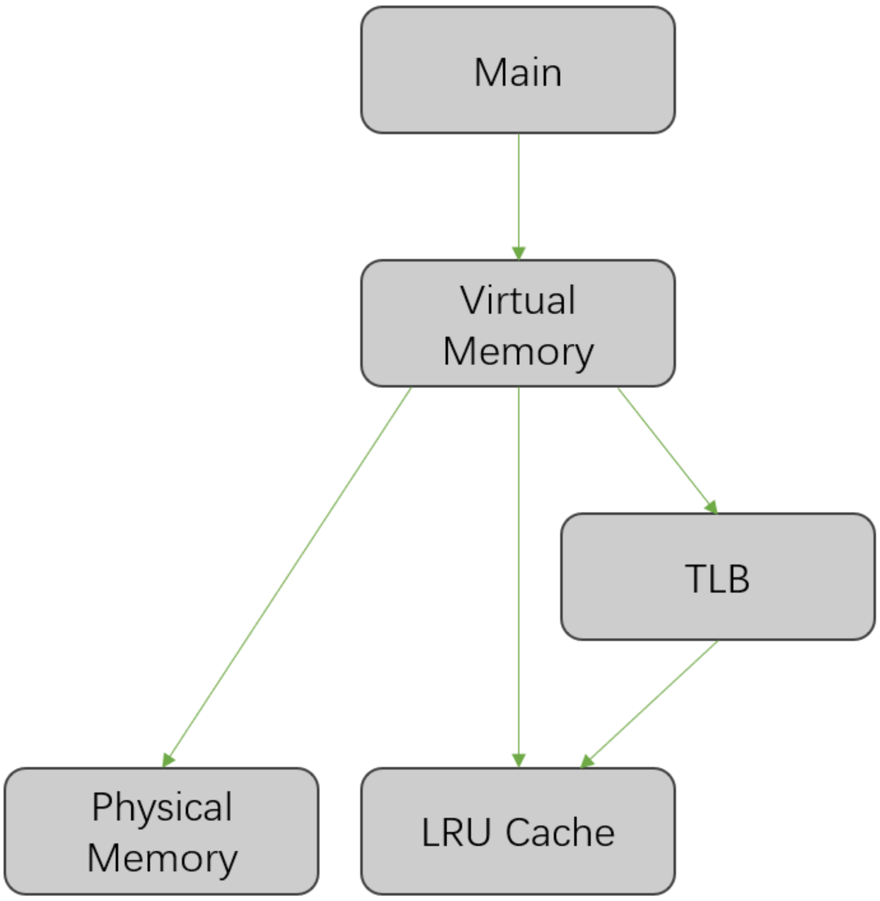


Figure 2: Structure of the project

Implementation

LRU Cache

The LRU cache is implemented using a circular doubly linked list. The node closest to the head is the least recently used, and the node closest to the tail is the most recently used. In order to support different types of data, the LRU cache is implemented as a generic class using `void *` to represent data and requires a comparison function to compare two data items.

The data structure of the LRU cache is:

```c
typedef void LRU_data;
typedef bool (*equal_t)(LRU_data *a, LRU_data *b);

/// @brief LRU Cache node
typedef struct LRU_cache_node
{
    LRU_data              *data;
    struct LRU_cache_node *prev;
    struct LRU_cache_node *next;
} LRU_cache_node_t;

/// @brief LRU Cache
typedef struct LRU_cache
{
    LRU_cache_node_t *head;
    size_t            size;
    size_t            capacity;
    equal_t           data_equal;
} LRU_cache_t;
```

The LRU cache provides the following main operations:

```c
/// @brief Query the cache for the cache data that matches the key
/// @return The cache data if found, NULL otherwise
LRU_data *LRU_cache_query(LRU_cache_t *LRU, LRU_data *key);
/// @brief Insert the data into the LRU cache
///        If there exists equivalent data, it will be updated
/// @return Removed data if the LRU cache is full, NULL otherwise
LRU_data *LRU_cache_insert(LRU_cache_t *LRU, LRU_data *data);
/// @brief Get the LRU data, will not update the LRU cache
/// @return The LRU if the LRU cache is not empty, NULL otherwise
LRU_data *LRU_cache_least_recently_used(LRU_cache_t *LRU);
/// @brief Remove the data from the LRU cache
/// @return The removed data if found, NULL otherwise
LRU_data *LRU_cache_remove(LRU_cache_t *LRU, LRU_data *key);
```

The query function is implemented by traversing the list and using the comparison function to compare the data. If the data is found, it is moved to the tail of the list to indicate that it is the most recently used.

The insert function first queries the cache for the data. If the data is found, it is updated and moved to the tail of the list. The function then checks if the cache is full. If so, the least recently used data is removed from the head of the list. The new data is then inserted at the tail of the list.

The least recently used function returns the data at the head of the list without updating the list.

The remove function is similar to the query function but removes the data from the list if found.

TLB

The TLB is implemented as a cache that stores the most recent translations from virtual addresses to physical addresses. The TLB is implemented using the LRU cache. The data structure of the TLB is:

```
/// @brief TLB entry structure, the "LRU_data" in LRU cache
typedef struct TLB_entry
{
    size_t page_number;
    size_t frame_number;
    bool   valid;
} TLB_entry_t;

/// @brief TLB structure
typedef struct TLB
{
    TLB_entry_t *entries;
    LRU_cache_t  cache;
    size_t       hit_count;
    size_t       miss_count;
} TLB_t;
```

The TLB provides the following main operations:

```
/// @brief Query for the frame number of a page number in the TLB
/// @return The frame number if TLB hits, -1 otherwise
int TLB_query(TLB_t *TLB, size_t page_number);
```

```
/// @brief Insert an entry into the TLB
void TLB_insert(TLB_t *TLB, size_t page_number, size_t
frame_number);

/// @brief Remove an entry from the TLB
/// @return True if the page number was removed
///         False if such page number was not in the TLB
bool TLB_remove(TLB_t *TLB, size_t page_number);

/// @brief The equal function for TLB entries
static bool TLB_entry_equal(LRU_data *entry_1, LRU_data *entry_2)
{
    TLB_entry_t *a = (TLB_entry_t *)entry_1;
    TLB_entry_t *b = (TLB_entry_t *)entry_2;
    return a->page_number == b->page_number;
}
```

The query function constructs a key as `{page_number, 0, false}` and queries the LRU cache. If the LRU cache hits, the frame number is returned. Otherwise, -1 is returned.

The insert function first judges if the LRU cache is full. If so, the least recently used entry is removed from the LRU cache and marked as “entry to insert”, otherwise it traverses the entries to find an invalid entry to be “entry to insert”. Then the “entry to insert” is initialized with parameters and inserted into the LRU cache.

The remove function constructs a key as `{page_number, 0, false}` and calls the remove function of the LRU cache.

Physical Memory

The physical memory is implemented as a memory manager that provides read and write operations of frames. The physical memory is implemented using an on-stack buffer given as a parameter. The data structure of the physical memory is:

```
/// @brief Memory frame structure
typedef struct physical_memory_frame
{
    int       frame_number;
    ptrdiff_t address;
    bool      valid;
    bool      dirty;
```

```c
} physical_memory_frame_t;

/// @brief Physical memory structure
typedef struct physical_memory
{
    void                    *memory;
    size_t                   memory_size;
    physical_memory_frame_t *frames;
    size_t                   frame_size;
} physical_memory_t;
```

The physical memory provides the following main operations:

```c
/// @brief Read from a memory frame, do not check address range and
validity
void physical_memory_read(physical_memory_t *memory, size_t
frame_number, ptrdiff_t offset, void *buffer, size_t byte_size);

/// @brief Write to a memory frame, do not check address range and
validity
void physical_memory_write(physical_memory_t *memory, size_t
frame_number, ptrdiff_t offset, const void *buffer, size_t
byte_size);
```

The read function reads data from the memory buffer at the specified frame number and offset.

The write function writes data to the memory buffer at the specified frame number and offset. The dirty bit of the frame is set to true after writing.

Virtual Memory

The virtual memory is the core component of the virtual memory manager. It integrates the LRU cache, TLB, and physical memory to provide read and write operations of virtual addresses. It holds a page table managed by LRU cache. In order to write answers, the virtual memory manager also holds a file pointer to the `answers.txt` file. The data structure of the virtual memory is:

```c
/// @brief The virtual memory page table entry
typedef struct virtual_memory_page_table_entry
{
    size_t page_number;
    size_t frame_number;
    bool   valid;
} virtual_memory_page_table_entry_t;
```

```c
/// @brief The virtual memory
typedef struct virtual_memory
{
    size_t                             page_size;
    virtual_memory_page_table_entry_t *page_table;
    size_t                             page_table_capacity;

    LRU_cache_t        page_cache;
    TLB_t              TLB;
    physical_memory_t physical_memory;
    FILE              *backing_store;

    size_t page_faults;
    size_t total_accesses;

    bool  log_enabled;
    FILE *log_file;
} virtual_memory_t;
```

The virtual memory provides the following main operations:

```c
/// @brief Read a byte from the virtual memory
int8_t virtual_memory_read(virtual_memory_t *virtual_memory,
ptrdiff_t address);

/// @brief Write a byte to the virtual memory, do nothing if the
logical address is invalid
void virtual_memory_write(virtual_memory_t *virtual_memory,
ptrdiff_t address, int8_t value);

/// @brief Get physical address from virtual address
ptrdiff_t virtual_memory_get_physical_address(virtual_memory_t
*virtual_memory, ptrdiff_t address);
```

The read function calls the `virtual_memory_get_physical_address` function to get the physical address and calls the `physical_memory_read` function to read the data from the physical memory. If log is enabled, the function writes the log to the log file.

The write function calls the `virtual_memory_get_physical_address` function to get the physical address and calls the `physical_memory_write` function to write the data to the physical memory. If log is enabled, the function writes the log to the log file.

The get physical address function is the core function of the virtual memory manager. The detailed implementation is as follows:

```
ptrdiff_t virtual_memory_get_physical_address(virtual_memory_t
*virtual_memory, ptrdiff_t address)
{
  // Extract page number and offset according to the requirement
  size_t page_number = (address & 0xFF00) >> 8;
  size_t offset      = address & 0x00FF;
  ++virtual_memory->total_accesses;

  // Query TLB for the frame number
  // If hit, return the physical address
  int TLB_query_frame =
    TLB_query(&virtual_memory->TLB, page_number);
  if (TLB_query_frame != -1)
    return TLB_query_frame * virtual_memory->page_size + offset;

  // Query page table for the frame number
  // If no page fault occurs, insert the entry into TLB
  // Then return the physical address
  virtual_memory_page_table_entry_t key = {page_number, 0, false};
  virtual_memory_page_table_entry_t *page_table_query_frame =
    LRU_cache_query(&virtual_memory->page_cache, &key);
  if (page_table_query_frame != NULL)
  {
    TLB_insert(&virtual_memory->TLB, page_number,
               page_table_query_frame->frame_number);
    return page_table_query_frame->frame_number *
           virtual_memory->page_size + offset;
  }

  // TLB and page table miss, page fault occurs
  // Evict the least recently used frame
  // If the frame is dirty, write back to the backing store
  // Then read the page from the backing store
  // Update the page table and TLB
  // Return the physical address
  ++virtual_memory->page_faults;

  // Find the frame to insert
  // If the LRU cache is full, evict the least recently used frame
  // Otherwise, find an invalid frame
  physical_memory_frame_t *frame_to_insert = NULL;
```

```
// If the LRU cache is full, evict the least recently used frame
if (LRU_cache_full(&virtual_memory->page_cache))
{
  // Find the least recently used frame
  virtual_memory_page_table_entry_t *least_recently_used_frame =
    LRU_cache_least_recently_used(&virtual_memory->page_cache);
  physical_memory_frame_t *frame_to_evict =
    physical_memory_get_frame(&virtual_memory->physical_memory,
                              least_recently_used_frame->frame_number);

  // Write back to the backing store if the frame is dirty
  if (frame_to_evict->dirty)
  {
    fseek(virtual_memory->backing_store,
          least_recently_used_frame->page_number *
          virtual_memory->page_size, SEEK_SET);
    fwrite(physical_memory_get_frame_address(
             &virtual_memory->physical_memory,
             frame_to_evict->frame_number),
           virtual_memory->page_size, 1,
           virtual_memory->backing_store);
    frame_to_evict->dirty = false;
  }

  // Evict the least recently used frame and reset its value
  LRU_cache_remove(&virtual_memory->page_cache,
                   least_recently_used_frame);
  TLB_remove(&virtual_memory->TLB,
             least_recently_used_frame->page_number);
  least_recently_used_frame->frame_number = (size_t)-1;
  least_recently_used_frame->valid        = false;
  frame_to_insert                         = frame_to_evict;
  }
// Otherwise, the LRU cache is not full
else
{
  // Traverse the physical memory to find an invalid frame
  for (size_t i = 0;
       i < virtual_memory->physical_memory.memory_size /
       virtual_memory->page_size; ++i)
  {
    physical_memory_frame_t *frame =
      physical_memory_get_frame(
          &virtual_memory->physical_memory, i);
      if (!frame->valid)
      {
        frame_to_insert = frame;
        break;
      }
    }
  }

  // The frame to insert is found
  // Read the page from the backing store
  // Then update the page table
  fseek(virtual_memory->backing_store,
        page_number * virtual_memory->page_size, SEEK_SET);
  fread(physical_memory_get_frame_address(
          &virtual_memory->physical_memory,
          frame_to_insert->frame_number),
        virtual_memory->page_size, 1,
        virtual_memory->backing_store);
  virtual_memory->page_table[page_number].frame_number =
    frame_to_insert->frame_number;
  virtual_memory->page_table[page_number].valid = true;

  // Insert the entry into the LRU cache and TLB
  frame_to_insert->valid = true;
  frame_to_insert->dirty = false;
  LRU_cache_insert(&virtual_memory->page_cache,
                   &virtual_memory->page_table[page_number]);
  TLB_insert(&virtual_memory->TLB, page_number,
             frame_to_insert->frame_number);

  // Return the physical address
  return frame_to_insert->frame_number *
         virtual_memory->page_size + offset;
}
```

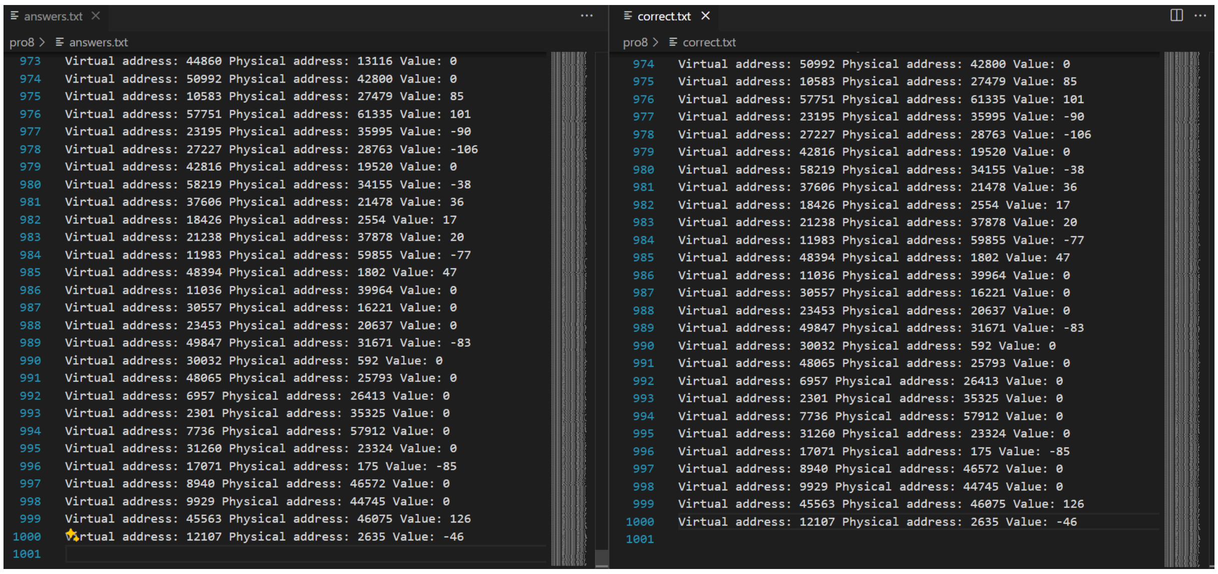
Main Program

The main program reads the virtual addresses from the `addresses.txt` file and calls the `virtual_memory_read` function to read the data from the virtual memory. The main program then writes the virtual address, translated physical address, and the value stored at the physical address to the `answers.txt` file by setting the log file of the virtual memory manager. The main program also prints the page fault rate and TLB hit rate to the console. The program finally checks the correctness of the virtual memory manager by comparing `answers.txt` with `correct.txt`.

Correctness

The correctness of the virtual memory manager is tested by comparing the output of the main program with the correct output. The correctness test is performed by running the main program with the provided `BACKING_STORE.bin`, `addresses.txt`, and `correct.txt`. The correctness test is passed if the output of the main program matches the correct output.

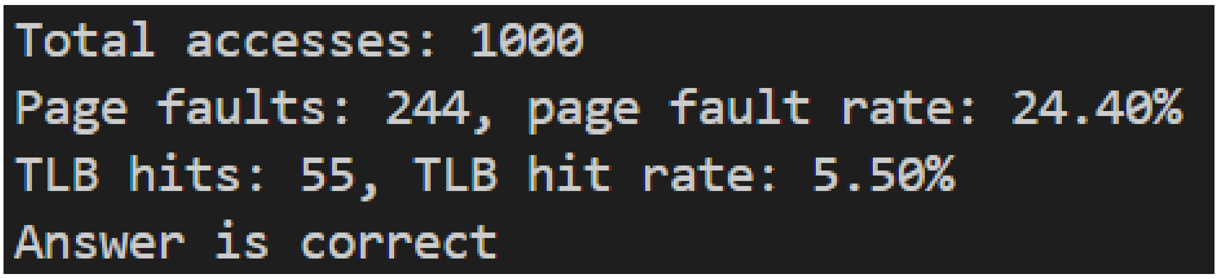
The correctness of the virtual memory manager is shown in the figure below. Only the last few lines of the output are shown for brevity. The correctness test is passed, as the output of the virtual memory manager matches the correct output.



```
Virtual address: 44860 Physical address: 13116 Value: 0
Virtual address: 50992 Physical address: 42800 Value: 0
Virtual address: 10583 Physical address: 27479 Value: 85
Virtual address: 57751 Physical address: 61335 Value: 101
Virtual address: 23195 Physical address: 35995 Value: -90
Virtual address: 27227 Physical address: 28763 Value: -106
Virtual address: 42816 Physical address: 19520 Value: 0
Virtual address: 58219 Physical address: 34155 Value: -38
Virtual address: 37606 Physical address: 21478 Value: 36
Virtual address: 18426 Physical address: 2554 Value: 17
Virtual address: 21238 Physical address: 37878 Value: 20
Virtual address: 11983 Physical address: 59855 Value: -77
Virtual address: 48394 Physical address: 1802 Value: 47
Virtual address: 11036 Physical address: 39964 Value: 0
Virtual address: 30557 Physical address: 16221 Value: 0
Virtual address: 23453 Physical address: 20637 Value: 0
Virtual address: 49847 Physical address: 31671 Value: -83
Virtual address: 30032 Physical address: 592 Value: 0
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
```

```
Virtual address: 50992 Physical address: 42800 Value: 0
Virtual address: 10583 Physical address: 27479 Value: 85
Virtual address: 57751 Physical address: 61335 Value: 101
Virtual address: 23195 Physical address: 35995 Value: -90
Virtual address: 27227 Physical address: 28763 Value: -106
Virtual address: 42816 Physical address: 19520 Value: 0
Virtual address: 58219 Physical address: 34155 Value: -38
Virtual address: 37606 Physical address: 21478 Value: 36
Virtual address: 18426 Physical address: 2554 Value: 17
Virtual address: 21238 Physical address: 37878 Value: 20
Virtual address: 11983 Physical address: 59855 Value: -77
Virtual address: 48394 Physical address: 1802 Value: 47
Virtual address: 11036 Physical address: 39964 Value: 0
Virtual address: 30557 Physical address: 16221 Value: 0
Virtual address: 23453 Physical address: 20637 Value: 0
Virtual address: 49847 Physical address: 31671 Value: -83
Virtual address: 30032 Physical address: 592 Value: 0
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
```

Figure 3: The output of the virtual memory manager
(page number: 256, frame number: 256)



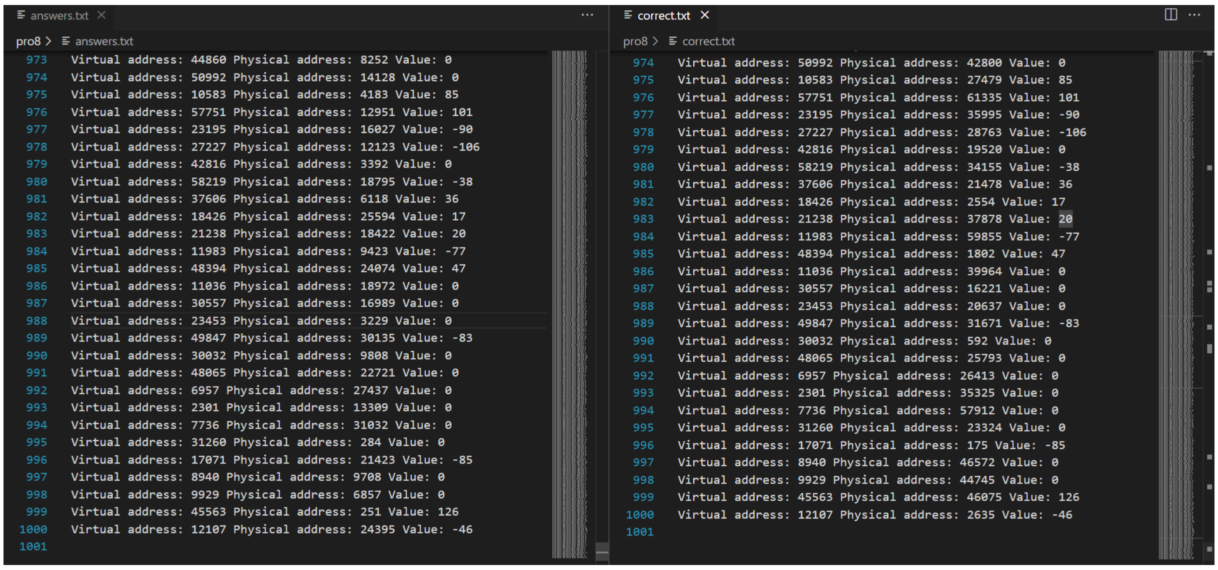
```
Total accesses: 1000
Page faults: 244, page fault rate: 24.40%
TLB hits: 55, TLB hit rate: 5.50%
Answer is correct
```

Figure 4: The TLB hit rate and page fault rate
(page number: 256, frame number: 256)

In the bonus test, we change the number of frames from 256 to 128.

- The "Physical address" is different from the correct answer because there will be page faults and the frames are different.

- The "Value" remains the same because the data in the backing store is the same.
- The page fault rate is significantly higher because the number of frames is reduced.
- The TLB hit rate is the same because the TLB is not affected by the number of frames as long as the capacity of TLB is smaller than the number of frames.
- The correctness test is not passed at line 172, where the 129th page is accessed and the manager must evict the least recently used frame.


answers.txt

pro8 > answers.txt

```
Virtual address: 44860 Physical address: 8252 Value: 0
Virtual address: 50992 Physical address: 14128 Value: 0
Virtual address: 10583 Physical address: 4183 Value: 85
Virtual address: 57751 Physical address: 12951 Value: 101
Virtual address: 23195 Physical address: 16027 Value: -90
Virtual address: 27227 Physical address: 12123 Value: -106
Virtual address: 42816 Physical address: 3392 Value: 0
Virtual address: 58219 Physical address: 18795 Value: -38
Virtual address: 37606 Physical address: 6118 Value: 36
Virtual address: 18426 Physical address: 25594 Value: 17
Virtual address: 21238 Physical address: 18422 Value: 20
Virtual address: 11983 Physical address: 9423 Value: -77
Virtual address: 48394 Physical address: 24074 Value: 47
Virtual address: 11036 Physical address: 18972 Value: 0
Virtual address: 30557 Physical address: 16989 Value: 0
Virtual address: 23453 Physical address: 3229 Value: 0
Virtual address: 49847 Physical address: 30135 Value: -83
Virtual address: 30032 Physical address: 9808 Value: 0
Virtual address: 48065 Physical address: 22721 Value: 0
Virtual address: 6957 Physical address: 27437 Value: 0
Virtual address: 2301 Physical address: 13309 Value: 0
Virtual address: 7736 Physical address: 31032 Value: 0
Virtual address: 31260 Physical address: 284 Value: 0
Virtual address: 17071 Physical address: 21423 Value: -85
Virtual address: 8940 Physical address: 9708 Value: 0
Virtual address: 9929 Physical address: 6857 Value: 0
Virtual address: 45563 Physical address: 251 Value: 126
Virtual address: 12107 Physical address: 24395 Value: -46
```

correct.txt

pro8 > correct.txt

```
Virtual address: 50992 Physical address: 42800 Value: 0
Virtual address: 10583 Physical address: 27479 Value: 85
Virtual address: 57751 Physical address: 61335 Value: 101
Virtual address: 23195 Physical address: 35995 Value: -90
Virtual address: 27227 Physical address: 28763 Value: -106
Virtual address: 42816 Physical address: 19520 Value: 0
Virtual address: 58219 Physical address: 34155 Value: -38
Virtual address: 37606 Physical address: 21478 Value: 36
Virtual address: 18426 Physical address: 2554 Value: 17
Virtual address: 21238 Physical address: 37878 Value: 20
Virtual address: 11983 Physical address: 59855 Value: -77
Virtual address: 48394 Physical address: 1802 Value: 47
Virtual address: 11036 Physical address: 39964 Value: 0
Virtual address: 30557 Physical address: 16221 Value: 0
Virtual address: 23453 Physical address: 20637 Value: 0
Virtual address: 49847 Physical address: 31671 Value: -83
Virtual address: 30032 Physical address: 592 Value: 0
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
```

Figure 5: The output of the virtual memory manager
(page number: 256, frame number: 128)

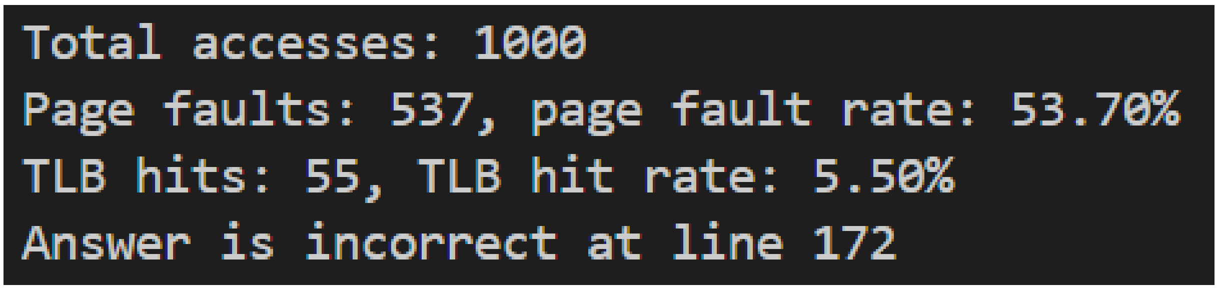

```
Total accesses: 1000
Page faults: 537, page fault rate: 53.70%
TLB hits: 55, TLB hit rate: 5.50%
Answer is incorrect at line 172
```

Figure 6: The TLB hit rate and page fault rate
(page number: 256, frame number: 128)

Conclusion

In this project, I implemented a virtual memory manager that provides read and write access to a virtual address space.

The virtual memory manager implements the page table, TLB, physical memory, and virtual memory, and it uses the LRU cache to manage the page table and TLB.

The virtual memory manager is tested for correctness by comparing the output of the main program with the correct output. The correctness test is passed, as the output of the virtual memory manager matches the correct output.

In the bonus test, we change the number of frames from 256 to 128. The same value is read from the backing store, but the physical address is different due to page faults. The page fault rate is significantly higher, and the correctness test is not passed, as is expected.

The virtual memory manager is a fundamental component of modern operating systems and provides a convenient and efficient way to translate virtual addresses to physical addresses. The virtual memory manager plays a crucial role in providing a stable and secure environment for running applications.

In this project, I learned how to implement a virtual memory manager and gained a deeper understanding of the memory management system in modern operating systems. I also learned how to implement template-like data structures in C and got familiar with the LRU algorithm and its applications in cache management.

The project may be further improved by splitting the page table into a independent structure to provide more flexibility and scalability, especially when the page table may use different structures like two-level page table or inverted page table. The project may also be improved by adding more detailed logs to help debug and analyze the performance of the virtual memory manager.