Virtual Memory Manager

Operational Manual and Specification

Mustafa Fawaz (103184737)
Ashraf Taifour (104262768)

COMP-3300 - Operating Systems
Dr. Alioune Ngom
University of Windsor
April 4, 2020

Mustafa Fawaz - 103184737
Ashraf Taifour - 104262768

Disclaimer:

I confirm that I will keep the content of this project confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this project. I acknowledge that a mark of 0 may be assigned for copied/plagiarized work."

- Ashraf Taifour 104262768, Mustafa Fawaz 103184737.

Back To Algorithm

Back To Top

Mustafa Fawaz - 103184737
Ashraf Taifour - 104262768

TABLE OF CONTENTS

1.0 OVERVIEW

2.0 PROGRAM STRUCTURE AND FUNCTIONALITY
- 2.1 CONSTANTS AND MACROS
- 2.2 DATA STRUCTURES
- 2.3 PROTOTYPES AND THEIR FUNCTIONS
- 2.4 VARIABLES

3.0 ALGORITHM

4.0 PROGRAM OUTPUTS AND STATISTICS
- 4.1 COMPILING
- 4.2 PROGRAM OUTPUTS AND STATISTICS

5.0 CONCLUSION

**Please visit section 4.1 Compiling for details on how to run the program.*

Back To Algorithm

Back To Top

Mustafa Fawaz - 103184737
Ashraf Taifour - 104262768

1.0 OVERVIEW

This project gives us the opportunity to create a Virtual Memory Manager that utilizes the concepts of demand paging learned in the COMP-3300 Operating Systems course at the University of Windsor. After taking the time to understand the project requirements, we were finally able to implement the required algorithms and processes to complete the operation. As it is understood, our file output matches the one provided by Dr. Alioune Ngom (correct.txt) ensuring that our use of the required files was correct.

Ultimately this program will present how Virtual memory works in Operating systems and how using virtual memory can solve many issues such as:

1) Not having enough memory.
2) Having any holes in our address space
3) Keeping programs secure by preventing any program from overwriting another.

This program will simulate how virtual memory works by using arrays as storage units (to represent virtual and physical memory) and the page fault handling method that is used is **demand paging** - this is when pages are loaded to memory only when in demand and **NOT** in advance.

The remainder of this report will provide a detailed breakdown of the program structure, functionality, variables, functions and outputs.

Back To Algorithm Back To Top

Mustafa Fawaz - 103184737
Ashraf Taifour - 104262768

2.0 PROGRAM STRUCTURE AND FUNCTIONALITY

The Virtual Memory Manager is program structure is broken down into three key files:

- main.c
- memfunc.h
- Memfunc.c

Main *(main.c)*

- This file drives the main functionality of the program utilizing the header file and function definition file to complete operations.
- Main communicates with all the necessary data structures, functions and variables.
- Main consists of all I/O, file descriptors and virtual memory for the backing store.

Header File *(memfunc.h)*

- This file consists of all macro definitions (defined in next section) and all the necessary function prototypes needed.
- The header file was created to ensure that we were able to track all functions created in the manager and no duplicate names were added during collaboration.

Function File *(memfunc.c)*

- This file provides all of the algorithms required for each function including their return values.
- This file also includes all global variables used in main. Most are required for tracking purposes such as counters and globally accessed file descriptors.
- Detailed breakdowns of each function are provided in section 2.3 (Prototypes and Their Functions).

Back To Algorithm Back To Top

2.1 CONSTANTS AND MACROS

All constants used in this program are defined in the memfunc.h file. They are all described in detail below.

#define PAGE_SIZE 256 : Page size used is 2_8 bytes.
*#define PM_SIZE (256 * 256)* : Physical Memory size used is 65,536 bytes.
#define TLB_SIZE 16 : Translation Look-Aside Buffer will have 16 entries

2.2 DATA STRUCTURES

Throughout the virtual memory manager, the following data structures are used to access, lookup and store page numbers and frames:

char physicalMem[PM_SIZE]

- This array stores all physical memory addresses after conversion.
- It is of size 256*256 = 65,536 bytes. Each entry consists of 256 bytes and our physical memory store is capable of storing 256 addresses.
    - **NOTE: This was done in accordance to Dr. Ngom's email on Mar 9 2020 where he outlines the following: "You may also revert to a page-table of size 256 entries (2^8) [not 128 entries] and proceed with the normal implementation of a page-table"*
- This array is also used for the program to retrieve the *value* output - that is, the signed byte value stored at the translated physical address in the array.

int pageTable[PAGE_SIZE]

- This array stores the page numbers (as indices of the array 0 - 255) and their corresponding frames.
- It is paged frequently throughout the operation of the program.

int tlBuffer[TLB_SIZE][2]

- This array houses the Translation Look-Aside Buffer.
- It is a 16 x 2 2D array. The first column houses the page numbers and the second column houses the corresponding frames.
- It is used for quick access when paging.
- **NOTE: Since we are using **Demand Paging**, the TLB will be empty to start causing TLB misses.*

Back To Algorithm Back To Top

char backStoreMap

- This data structure is a pseudo-data structure in memory.
- Instead of accessing the backing_store randomly when needed, we have mapped the contents of the backing_store to memory using the c-function mmap() (from <sys/mann.h>).
- We are given access to this area in memory using the pointer *backStoreMap*.
- When a page fault occurs, we access this area in memory using this variable to retrieve the respective page and corresponding frame number. This information is then added to the page table.

2.3 PROTOTYPES AND THEIR FUNCTIONS

This section will detail the function prototypes and their algorithms. All prototypes for these functions are in the *memfunc.h* file, with their function bodies located in *memfunc.c*. This was done to prevent duplicate naming conventions for functions when working on code simultaneously.

void createPageTable()

- This function creates the page table which is used throughout the program.
- The array *int pageTable* is initialized with the value -1 at each index (256 pages).
    - -1 is used to track faults.
- Each index + 1 corresponds to the page number, e.g. index 0 = page number 1, or 0 = 0, etc.

void createTLB()

- This function creates the Translation Look-Aside Buffer
- The array *int tlBuffer* is initialized with the value -1 at each index in the 2D array.
- Each page number (16 total) and each corresponding frame (16 total) is initialized to -1.
    - -1 is used to track “empty” positions in the buffer.

void createBackStore()

- This function creates the backing store in memory by taking the *BACKING_STORE.bin* binary file and mapping it to local memory.
- The *backStoreFD* is a file descriptor which opens the *BACKING_STORE.bin* file as a *READ-ONLY* file. The binary file MUST be in the local directory for the program to continue, otherwise the function will throw an error and exit.
- Once the binary file is opened, the mmap() function is called which begins mapping the file to memory. mmap() takes the following parameters:
    - “0” - The “starting-point” in local memory. This is more of a hint for mmap() and the actual address of where this is mapped is returned by the function.
    - “256” - Each frame in memory is to be 256 bytes in length, corresponding to our page size.

Back To Algorithm Back To Top

    - "PROT_READ" - Pages may only be read from memory. This is similar to opening a file as READ-ONLY.
    - "MAP_SHARED" - Shares this memory location with processes that map the object. Required for our program to access this backStore.
    - "backStoreFD" - The file descriptor containing the data to be mapped, opened one step prior.
    - "0" - The offset. Since no offset is required, this is set to 0.
- mmap() returns a pointer to this section in memory which is stored in the variable *backStoreMap*.
- If the return value is equal to "MAP_FAILED", the backStoreFD is closed, an error is presented to the console and the program exits.

int checkTLB(int pageNumber)

- This function checks the tlBuffer array using the pageNumber being passed to see if there is a TLB hit (page number/corresponding frame number found).
- For each of the 16 entries, if the pageNumber matches the corresponding value in the first column, the TLBCounter is incremented (signifying a hit) and the frame number is returned at the corresponding page back to main.
- If no match is found for the pageNumber in the tlBuffer, -1 is returned to main indicating a TLB miss.

int checkPageTable(int pageNumber)

- This function checks the pageTable array for a match using the pageNumber being passed.
- The pageNumber is used as the index for the pageTable array and the value at that index in the array is stored in a temp variable.
- If the value in *temp* is -1, this signifies a fault and the *pageFaultCounter* is incremented.
- The *temp* variable is returned back to main regardless of the result. This is because:
    - The *temp* may be a hit and the corresponding frame number is found.
    - The *temp* may be a fault and the corresponding frame number is not found.

void addToTLB(int frameNum, int pageNumber)

- This function will update the frame number and page number to TLB using FIFO.
- It initially checks if TLB is empty by checking the *tlbPointer* variable. And if it is the variable '*tlbPointer'* will be set to 0.
- If the TLB isn't empty then the tlbPointer will be incremented while also using modulus '%' to the TLB_SIZE so we don't go over the tlBuffer size - this is important for when the tlb is incremented to a value above the TLB_SIZE (above 16) as it means we always stay at values from the range of 0-15.
- The actual values of the 2D array '*tlBuffer*' in the first column indicates the PageNumber and the second column is the 'frameNum'.

Back To Algorithm Back To Top

- Each row will be filled by the variable ‘tlbPointer’ and by doing this we can fill each value of the tlBuffer 2D array with address information using the parameters that are passed into it ‘frameNum’ and ‘pageNumber’.

2.4 VARIABLES

This section provides a breakdown and use of each variable. Due to the vast nature of naming conventions for variables, these explanations will be beneficial in understanding the overall algorithm of our memory manager.

The variables used in this memory manager were split into two categories: global and local. This was done to ensure that all functions outside of main were able to access variables with ease while also being given the ability to update them without restrictions. Since this is not a parallel program, the use of global variables was not a concern. All local variables are nested within *main.c* and they are strictly used within that scope.

Global *(in memfunc.c, fig 2.1)*

- **int TLBCounter**
  - TLB hit counter. Is incremented when a TLB hit occurs.
- **int pageFaultCounter**
  - Page table fault counter. Is incremented when a page fault occurs.
- **int addressCount**
  - Input address counter. Is incremented when a new address is read from file.
- **char *backStoreMap**
  - Pointer to the *BACKING_STORE.bin* file mapped to local memory.
- **int backStoreFD**
  - File descriptor assigned to opening the *BACKING_STORE.bin* file
- **int tlbPointer**
  - Index pointer for the TLB array. Used for FIFO policy insertions.
- **int physMemPointer**
  - Index pointer for the physicalMem array. Used to access frames stored in physical memory.

Local *(in main.c, fig 2.2)*

- **FILE *addressFile**
  - File descriptor assigned to the *addresses.txt* input file from the first and only command line argument accepted by the program.
- **FILE *outputFile**
  - File descriptor assigned to the output.txt output file produced by the program.

Back To Algorithm Back To Top

- **char address[7]**
  - Temp buffer to store the current virtual address retrieved from the *addressFile*.
- **int currentAddress**
  - Stores the current virtual address as an integer after retrieving it from the buffer.
- **int physicalAddress**
  - Stores the current converted physical address.
- **int pageNumber**
  - Stores the extracted 8-bit page number from the current virtual address
- **int offset**
  - Stores the extracted 8-bit offset from the current virtual address
- **int frameNum**
  - Stores the current corresponding frame number returned from the TLB or page table.
- **int byteValue**
  - Stores the current final physical memory value in bytes.

Fig 2.1 - Global Variables

```
//=========================
//ALL GLOBAL VARIABLES
//=========================
int TLBCounter = 0, pageFaultCounter = 0, addressCount = 0; //Counters
char *backStoreMap; //Map to backing store in memory
int backStoreFD; //Used as the file descriptor for the backStore
int tlbPointer = -1; //Pointer for TLB when using FIFO policy
int physMemPointer = 0; //used an index for physical memory
```

Fig 2.2 - Local Variables

```
FILE *addressFile; //file descriptor for the input file
FILE *outputFile; //file descriptor for the output file
char address[7]; //input buffer for virtual addresses from file
int currentAddress, physicalAddress; //address storage
int pageNumber, offset, frameNum, byteValue, tempVal;
clock_t start, finish, total_time; //used for timing
```

Back To Algorithm Back To Top

3.0 ALGORITHM

Throughout this section, a detailed breakdown of the implementation and the associated algorithms will be presented. Each function and variable are described in their respective sections (*2.3 Prototypes and Their Functions & 2.4 Variables)*. Please click on each for a detailed definition if required.

This program starts by calling the following functions from the memfun.c header file:

1) createPageTable()
2) createTLB()
3) createBackStore()

The program then proceeds to open an input file that contains the addresses which will be the first argument that is passed into the console after our executable example:

>> *"./a.out addresses.txt*"

addresses.txt is the text file that stores all of the given addresses that will be used for the virtual addressing and the file descriptor will be stored in the variable '*addressFile*'.

The program will then open (create/overwrite) *'output.txt'* . This is the file that we will write our output to. The file descriptor for this file is stored in the variable '*outputFile*'.

**NOTE: error handling is done for both file opening (I/O) processes. This is to inform the user if any unexpected behavior occurs.*

Before the loop starts we use the *time.h* function *clock();* this method will store the number of clock ticks elapsed since the program was launched at that point in the code. This will be stored in the '*start*' variable of data type '*clock_t*'.

The program then enters a while loop. This while loop's end condition is when the EOF of "*addresses.txt*" is reached.

Start of loop:

Each iteration of the while loop starts by using the function '*fgets*', which has three parameters. The first parameter is '*address*', followed by 7 (this will take 7 characters including the newline character) and finally '*addressFile*'. fgets will go line by line through the address file and store it in the address buffer.

Back To Algorithm Back To Top

sscanf is then used to write the contents of the buffer to an integer. It takes three parameters: '*address*' is the buffer, '*%d*' is the convert-to-type (int) and the third is '*currentAddress*', which is where the current address will be stored.

The program then determines the offset and the page number from the current address. This is done by using bitwise operations. *currentAddress* is bitwise &'d by 0xff and is initialized to the offset variable. This will & the value of the address that we have with 11111111 which will preserve the value of the first 8 bits.Then, the *currentAddress* is bit-shifted by 8-bits to the right and the result is stored in the *pageNumber* variable. This isolates the bits that were at position 8-15 which represents the page number.

'*frameNum*' is set then to equal the return value of the function '*checkTLB(pageNumber)*'. The TLB has to be checked first as it allows for quick access if the address was recently used. Note that with demand paging, the first series of pages will fault because the TLB will be empty. This will change overtime as the TLB fills.

'*frameNum*' is checked if it is NOT equal to -1. If this statement is true then there is a TLB hit.

TLB HIT STEPS:

1) Set '*physicalAddress*' equal to '*frameNum*' + '*offset*'
2) Set '*byteValue*' equal to '*physicalMem[physicalAddress]*'. This passes the '*physicalAddress*' into our physical memory array to obtain the value in that address and store it in '*byteValue*'.

TLB MISS STEPS:

1) Set '*frameNum*' equal to the return value of '*checkPageTable(pageNumber)*'

   NO PAGE FAULT:
   If '*frameNum*' is not equal to -1 then it means no page fault has occurred.
   In that case three lines of code are executed:
   1) Set '*physicalAddress*' equal to '*frameNum*' + '*offset*'
   2) Call function '*addToTLB(frameNum, pageNumber)*'. This will add frameNum to the first column, pageNumber to the second column of the data structure '*tlBuffer*'.
   3) Set '*byteValue*' equal to '*physicalMem[physicalAddress]*'. This passes the '*physicalAddress*' into our physical memory array to obtain the value in that address and stores it in '*byteValue*'.

Back To Algorithm Back To Top

PAGE FAULT OCCURRED:

- If '*frameNum*' is equal to -1, a page fault has occurred. In this case we first set a variable of type int '***pageFront***'. This is initialized to equal '*pageNumber*' * '*PAGE_SIZE*'. This variable is used to acquire the front of the pageTable. This is used as an index along with '*backStoreMap*' to find a free frame by consulting our BACKING_STORE.bin file.
- '*physMemPointer*' is checked. If it is NOT equal to -1 then the following lines of code will be executed:
    a. '*memcpy(physicalMem + physMemPointer, backStoreMap + pageFront, PAGE_SIZE);*. This will copy data from the local mapped backing store in memory (i.e. the BACKING_STORE.bin) into the physicalMem array. The size of the entry will be equal to '*PAGE_SIZE*' (256 bytes). This will ultimately retrieve info from BACKING_STORE.BIN and place it into the free frame.
    b. '*frameNum*' will be set equal to '*physMemPointer*' meaning it will be pointing to its dedicated memory.
    c. Set '*physicalAddress*' equal to '*frameNum*' + '*offset*'
    d. Set '*byteValue*' equal to '*physicalMem[physicalAddress]*." This passes the '*physicalAddress*' into our physical memory array to obtain the value in that address and stores it in '*byteValue*'.
    e. '*pageTable[pageNumber] = physMemPointer;* This will insert '*frameNum*' into the '*pageTable*'.
    f. '*addToTLB(frameNum, pageNumber);*' this will insert '*frameNum*' into the '*tlBuffer*'.
    g. Finally, we check if '*physMemPointer*' is less than '*PM_SIZE*' - 256. If it is, we increment by 256 else we set '*physMemPointer*' equal to -1. This process works by tracking the memory index pointer for each addition to determine if there exists space within the physical memory array. This is done by adding 256 bytes to the pointer from it's existing position to the next entry. If physical memory is full, assign the pointer to -1 indiciating no free frames.

Finally, the program writes the following to the output.txt file by using fprintf:

- Virtual address:
    - *fprintf(outputFile, "Virtual Address: %d ", currentAddress);*
- Physical address:
    - *fprintf(outputFile, "Physical Address: %d ", physicalAddress);*
- Value by:
    - *fprintf(outputFile, "Value: %d\n", byteValue);*

Back To Algorithm Back To Top

The program then increments '*addressCount*' to keep track of the amount of addresses we have accessed. The while loop proceeds for its next iteration.

Once the EOF has been met and the while loop is broken, we use the *time.h* header function *clock().* This method will store the number of clock ticks elapsed since the program was launched at that point in the code. This will be stored in the '*finish*' variable of data type '*clock_t*'.

The program then subtracts '*finish*' from '*start*' and stores the result in '*loop_time*'. This will store the number of clock ticks that occured exclusively in the loop. '*loop_time*' is then printed to the console.

'*Total_time*' will store the difference between finish and start divided by CLOCKS_PER_SEC (this is 1000000 in 32-bit machines) to get the number of seconds used by the CPU. **NOTE:** The answer will now likely be 0 as the number of clock ticks is around 2000 for my machine meaning the loop is executed very quickly.

The program will then write the following statistics to the output file:

- Number of translated addresses:
  - *fprintf(outputFile, "Number of Translated Addresses: %d\n", addressCount);*
- Page faults:
  - *fprintf(outputFile, "Page Faults = %d\n", pageFaultCounter);*
- Page fault rate
  - *fprintf(outputFile, "Page Fault Rate = %.3f\n", (double)pageFaultCounter/addressCount);*
- TLB hits
  - *fprintf(outputFile, "TLB Hits = %d\n", TLBCounter)*;
- TLB hit rate
  - *fprintf(outputFile, "TLB Hit Rate = %.3f\n", (double)TLBCounter/addressCount);*
- TIME PRINTOUT
  - *fprintf(outputFile, "Average Page-Replacement Time = %.9f", (total_time)/1000.0);*
  - Note: we divide by 1000 as it is the total number of addresses so we are trying to get the time taken for every single page-replacement.
  - This number will be zero in our results as the loop is executed rapidly but the time taken will be close to nanoseconds per replacement.

Back To Algorithm Back To Top

4.0 PROGRAM OUTPUTS AND STATISTICS

This section includes details on how to compile and run the Virtual Memory Manager along with an interpretation of the outputs. Details regarding the statistics are provided.

4.1 COMPILING

To compile the Virtual Memory Manager, ensure that the following files are present in the **same** local folder:

- main.c
- memfunc.h
- memfunc.c
- BACKING_STORE.bin
- Addresses.txt

Once confirmed, open a local terminal/shell OR navigate to the relevant directory with the present program files. In the command line, run the following:

>> gcc main.c
>> ./a.out addresses.txt

```
moose@Moose:/mnt/c/Users/Moose/Desktop/Winter 2020/COMP-3300 - OS/cs330/Project$ gcc main.c
moose@Moose:/mnt/c/Users/Moose/Desktop/Winter 2020/COMP-3300 - OS/cs330/Project$ ./a.out addresses.txt
```

On successful compilation and run, an output file called output.txt will be produced with the outputs from the Virtual Memory Manager.

The following errors may occur if there are incorrect files in the directory or files are missing:

Missing BACKING_STORE.bin

```
moose@Moose:/mnt/c/Users/Moose/Desktop/Winter 2020/COMP-3300 - OS/cs330/Project$ ./a.out addresses.txt
BACKING STORE file not found. Please add to directory.
```

Missing addresses.txt

```
moose@Moose:/mnt/c/Users/Moose/Desktop/Winter 2020/COMP-3300 - OS/cs330/Project$ ./a.out addresses.txt
addresses.txt file not found. Please add to directory.
```

[Back To Algorithm](#) [Back To Top](#)

Note: Since the input file is specified in the command line, it can be renamed to whatever the user prefers; however, it must be the structure of the input file provided by Dr. Ngom. The backing store MUST be titled 'BACKING_STORE.bin'

4.3 PROGRAM OUTPUTS AND STATISTICS

The Virtual Memory Manager outputs all converted virtual addresses, their values and some additional statistics related to:

- Page Faults
- Page Fault Rate
- TLB Hits
- TLB Hit Rate
- Average Replacement Time

Page Faults & Page Fault Rate

>> Page Faults = 244
>> Page Fault Rate = 0.244 (24%)

Based on the 1000 translated addresses, our memory manager encounters 244 page faults utilizing the BACKING_STORE.bin and **demand paging**. This is the expected amount based on the results provided by Dr. Ngom. These faults occur when the page number in the page table does not have a corresponding frame number. This results in accessing the backing store to retrieve the frame.

The rate at which this occurs is simply the number of faults divided by the total number of addresses converted, which is approximately 0.244, or 24%. As previously mentioned, demand paging results is a reactive algorithm and does not have frames mapped prior to execution. Therefore, on program launch, the page table will be empty so faults will occur.

Translation Look-Aside Buffer Hits / Hit Rate

>> TLB Hits = 54
>> TLB Hit Rate = 0.054 (5%)

As the Memory Manager runs, the TLB is filled with page numbers and their respective frames. Once again, this occurs because of the Demand Paging algorithm. As we know, the TLB is used for quick access of frames in memory; however, when the requested frame is not in the TLB, it is considered a miss. Should the requested page/frame be in the TLB, this is considered a hit and the required information is accessed faster. Based on the addresses and the provided

backing_store, our program encounters 54 TLB hits at a rate of 5%, which is the expected result based on those provided by Dr. Ngom.

Average Replacement Time

>> loop time = 2130 clock ticks
>> Average Page-Replacement Time = 0.000000000

Note that Avg page-replacement time is calculated by dividing loop time by '*CLOCKS_PER_SEC*' (this is 1000000 in 32-bit machines). This division operation is stored in the variable '*total_time*'. This variable is then divided by 1000 (which is the number of addresses) that we have so we can know the time for page-replacement for each address. This explains why the answer is '0.000000000' as the number is far too small to be captured since the operation is very fast.

Back To Algorithm Back To Top

5.0 CONCLUSION

In conclusion, we found this project very rewarding on both the coding and knowledge fronts. Having learned the concepts of demand paging in the course, the application of it is very eye-opening. The use of Operating System GUI's overshadow all of the back-end work that occurs between the CPU, memory and so forth all within nano-seconds. Even more remarkable is the speed at which our (and all) Virtual Memory Managers produce the required output files considering all of the moving parts.

This project was challenging at times; however, it is key to note that the implementation of such concepts and algorithms is not easy. To design a large (but small) aspect of back-end OS functionality from scratch takes time and patience. Overall, we hope that you enjoyed testing our program and reading through our documentation.

Thanks.

Mustafa Fawaz (103184737)
Ashraf Taifour (104262768)

Back To Algorithm Back To Top