REDUCING THE NUMBER OF PAGE FAULTS BY SEPERATING INSTRUCTIONS AND DATA

By

PAVAN KUMAR ATHOTA

Bachelor of Technology

Kakatiya University

Andhra Pradesh, INDIA

1997

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 2004

REDUCING THE NUMBER OF PAGE FAULTS BY

SEPERATING INSTRUCTIONS AND DATA

Thesis Approved:

Adviser

Dean of the Graduate College

PREFACE

In a typical paged memory management system, a high number of page faults generally decreases the performance of a computer system by increasing the average memory access time. The objective of this study was to increase the performance of a computer system by reducing the number of page faults through separation of instructions and data. In the adopted scheme, data and instructions are stored separately in the secondary memory. The page frames allocated to particular process in memory are divided between the data pages and the instruction pages. Each portion is managed in a different way to get optimum results in terms of a lower overall page fault rate.

A trace-driven simulation was implemented to evaluate the performance of the new design. Four standard page replacement algorithms, i.e., FIFO, LRU, LFU, and Second Chance, were used to evaluate the performance of the new design. The design was tested with both pre-generated input traces and random traces. In both cases, a significant improvement in the performance of the system, in terms of the number of page faults generated, was observed. It was observed that the number of page faults generated for the partitioned address space scheme, i.e., with data and instructions separated, was high if the allocation to the instruction pages and data pages in memory is uneven. But if the allocation is even, it was observed that the minimum number of page faults generated was less by an average of 3.5 percent than the minimum number of page faults generated by the standard algorithms, when applied to the same input.

ACKNOWLEDGEMENTS

My sincere gratitude is due to my adviser Dr. Mansur H. Samadzadeh for his inspiration, guidance, and continuous encouragement throughout my thesis work. The presented work is the result of his support, motivation, and valuable time.

I also thank Dr. Blayne Mayfield and Dr. Nohpill Park for serving on my graduate committee. Their comments and suggestions are greatly appreciated.

My heartfelt thanks go to my family members for their extreme love and continuing support. Finally, I would like to thank my friends for their encouragement and moral support.

TABLE OF CONTENTS

Chapter | Page

I INTRODUCTION 1

1.1 Problem Statement 1
1.2 Thesis Outline 2

II LITERATURE REVIEW 3

2.1 Memory Management 3
2.1.1 Paging 3
2.1.2 Segmentation 4
2.1.3 Paged Segmentation 5
2.2 Paging 5
2.2.1 Simple Algorithms 6
2.2.2 Enhanced Algorithms 7
2.2.2.1 Second Chance Algorithm 7
2.2.2.2 Page Fault Frequency Algorithm 8
2.2.2.3 SFIFO Algorithm 8
2.2.2.4 MLF Algorithm 9
2.2.2.5 EELRU Algorithm 9

III DESIGN AND IMPLEMENTATION ISSUES 11

3.1 Implementation Platform and Environment 11
3.2 Objective 11
3.3 Input Parameters 12
3.3.1 Input Traces 12
3.3.2 Page Frames 12
3.3.3 Page Size 13
3.3.4 Page Replacement Algorithm 13
3.3.5 Separate Consideration of Instruction Pages and Data Pages 13
3.4 Design of the Simulation 14
3.4.1 Random Trace Generation 14
3.4.2 Page Table 15
3.4.3 Clock 16
3.5 Implementation Detail 17

IV EVALUATION OF THE SIMULATION 19

4.1 Graphs 19
4.2 Observations 20

V SUMMARY AND FUTURE WORK 24

5.1 Summary 24
5.2 Future Work 25

REFERENCES 27

APPENDICES 29

APPENDIX A - GLOSSARY 30

APPENDIX B - TRADEMARK INFORMATION 32

APPENDIX C - EXPERIMENTAL RESULTS 33

APPENDIX D - PROGRAM LISTING 36

LIST OF FIGURES

Figure Page

1 Structure of a page table entry.. 15

2 Structure of each entry of the array holding additional information for memory-resident pages 16

3 Performance Graph 1 (Results produced for pre-generated traces with 30 memory page frames, 1024 words per page).. 21

4 Performance Graph 2 (Results produced for random trace with 30 memory page frames, 1024 words per page).. 23

LIST OF TABLES

Table Page

I Results produced using pre-generated traces with 30 memory page frames and 1024 words per page with input #4 33

II Results produced using random traces with 30 memory page frames and 1024 words per page.. 34

III The inputs taken from the public site at New Mexico State University.............. 35

CHAPTER I

INTRODUCTION

The memory of a computer system is made up of different levels. The memory at the highest level of the hierarchy consists of the registers in the CPU. Next comes the cache memory. Just below the cache memory there is the random access memory which is also referred to as main memory. Excluding external memory, at the bottom of the hierarchy there is the secondary memory or disk storage [Burger 96].

Typically, the CPU can only perform operations on the program instruction words and data words in the registers that have been loaded from main memory. So any word of instruction or data that is not in the registers will have to be fetched from the memory for the CPU to do its job. Since the number of program instruction words or data words that can be stored in the registers is very small, words of instruction and data keep moving in and out of the registers. When a required word of instruction or data cannot be found in the registers, the system fetches the instruction or data word from the next lower level of the memory hierarchy. And, if the word of instruction or data cannot be found in that level of the memory system, it is fetched from a lower level.

1.1 Problem Statement

Instruction word or data word transfers in the memory hierarchy can consume a significant amount of time, thereby decreasing the processing speed. Instruction or data

word transfers can take different amounts of time depending on the levels of memory hierarchy between which a transfer is occurring. Typically, it takes about 100 ns (nanoseconds) for a transfer between main memory and registers, and about 10 ms (milliseconds) for a transfer between secondary memory and main memory [Morris 99]. Typical memory clock cycle times are between 2.5 to 5.0 ns [Morris 99]. So accesses to main memory are fast enough and do not generally hamper the performance of the CPU. But, accesses to secondary memory take a lot more comparatively so that the CPU has to wait for a transfer before it can proceed with execution. A significant improvement in the performance of a system can be achieved if the number of accesses from secondary memory can be minimized.

1.2 Thesis Outline

In this thesis work, a new design was implemented to reduce the number of accesses to secondary memory and thus improving the performance of a system. In Chapter II, an introduction to various memory management schemes is given and the paging memory management system is discussed in detail. In Chapter III, the implementation issues of the new design are discussed. In Chapter IV, the performance of the simulation is evaluated and, finally, the summary and future work are discussed in Chapter V.

CHAPTER II

LITERATURE REVIEW

This chapter gives a brief description of various memory management schemes and then discusses the paging memory management system in detail.

2.1 Memory Management

Generally, every time data cannot be found in main memory, they have to be fetched from secondary memory (unless the system has an intervening layer of cache). It is common in such cases to transfer a block of data instead of just the requested piece of data.

Since the higher levels of the memory hierarchy can typically hold smaller amounts of data than the lower levels, one must make sure that, as much as possible, only the temporally and spacially relevant data reside in the higher levels of the memory hierarchy. For this purpose, a memory management system is needed to decide where each data item, or at least each block of data, is to be stored.

2.1.1 Paging

Paging is the memory management scheme that stores the instruction and data words of a process in fixed size blocks in the physical memory. In this scheme, the

physical memory is divided into blocks of fixed size called page frames. The logical memory is also broken into blocks of the same size called pages. In this scheme, all addresses generated by the CPU consist of two parts: a page number and an offset. A page table (a mapping mechanism between a process' pages and frames) converts each page number to the starting address of the corresponding page frame in the physical memory. A disadvantage of paging is that the physical memory could have internal fragmentation, which is the unused space in a page frame for all processes in the system.

2.1.2 Segmentation

Segmentation is the memory management scheme in which the program instruction words and data words as viewed by the user are stored in the physical memory, i.e., in terms of functions, subroutines, tables, etc. [Silberschatz et al. 03]. In this scheme, the physical memory consists of variable size blocks called segments. The addresses generated by the CPU consist of a segment name and an offset. A segment table (a mapping between a process' logical segments and physical segments) helps convert the addresses generated by the CPU into physical addresses. The main problem with segmentation is that it can cause external fragmentation where none of the empty slots in memory may be sufficient to accommodate an incoming process' segments due to inefficient handling of the lists of free memory segments or holes.

2.1.3 Paged Segmentation

Both paging and segmentation have some disadvantages. In order to overcome them, the features of both schemes are combined to obtain a memory management system called paged segmentation. In paged segmentation, each segment is divided into pages and has its own page map table. Paged segmentation incurs less external fragmentation than segmentation because the segments no longer need to be stored whole. But this scheme causes a significant overhead, i.e., maintaining more tables. In addition, two memory accesses are required for logical-to-physical address translation: one for the segment table and the other for the page table of that particular segment [Silberschatz et al. 03].

2.2 Paging

This section describes the memory management scheme of paging and then briefly discusses a number of popular page replacement algorithms.

In a typical computer system with a paged memory, a block of data (consisting of program instruction and/or program data words) is called a page, and a request for a page made to the disk by main memory is called a page fault. Whenever main memory is full at the time of a page fault, a memory resident page is removed from it and the requested page is loaded in its place from the secondary memory. This is called page replacement. The algorithm that selects the page to be removed is called a page replacement algorithm.

Handling page faults consumes a significant amount of time because of the time it takes to search the secondary memory for the desired page of instructions or data, and

also the time it takes to transfer the information between the levels of the memory hierarchy.

Since it takes time to transfer information between main memory and secondary memory, a computer system should use a page replacement algorithm that leads to a minimum number of page faults. A number of page replacement algorithms are discussed in the following subsections.

2.2.1 Simple Algorithms

In this section, three simple page replacement algorithms are discussed that require some hardware and software support for their implementation [Silberschatz et al. 03].

The First In First Out (FIFO) algorithm is the simplest of all page replacement algorithms conceptually. This algorithm states that when there is a page fault and memory is full, the page to be removed first is the page that has been in main memory for the longest period of time.

The Least Recently Used (LRU) algorithm generally yields the best results, among the page replacement algorithms of the early days, in terms of the fewest number of page faults generated. According to this algorithm, whenever there is a page fault and memory is full, the page to be replaced is the one that has not been used for the longest period of time. LRU-based algorithms are predominant in virtual memory management systems because of their efficiency and simplicity [Smaragdakis et al. 99].

According to the Least Frequently Used (LFU) algorithm, whenever there is a page fault and memory is full, the page that has to be replaced is the page that has the minimum number of hits in the time period it has resided in memory.

2.2.2 Enhanced Algorithms

The page replacement algorithms mentioned in Subsection 2.2.1 above are relatively simple in the sense that they require little hardware support and no dynamic calculations. In this section, a number of page replacement algorithms are discussed that require relatively more hardware support and/or dynamic calculations.

2.2.2.1 Second Chance Algorithm

This is an extension of the FIFO algorithm. In this algorithm, when a page fault occurs and memory is full, instead of replacing the selected page, the reference bit of that particular page is checked to see if it has been used recently. In such a case, that page is given a second chance and another page is selected to be removed from main memory. When a page gets a second chance, its reference bit is cleared [Silberschatz et al. 03]. This algorithm needs some hardware to store the bits. This algorithm generally reduces the number of page faults by preventing the removal of the recently used pages, which arguably are more likely to be used in the near future.

2.2.2.2 Page Fault Frequency Algorithm

The Page Fault Frequency (PFF) replacement algorithm attempts to dynamically control the rate of page faults produced by a program running in a paged virtual environment by varying the amount of memory allocated to the program [Sadeh 99]. According to this algorithm, when the page fault frequency rises above a critical level, some of the referenced pages that are not in main memory are brought into main memory. Similarly, when a number of pages in memory are not being used, some of the unused pages are removed from main memory [Chu and Opderbeck 76]. The PFF algorithm measures the inter page fault intervals during execution. At page fault times, the algorithm compares those intervals with a selected threshold T. If the interval exceeds T, all the pages that are not referenced during the interval are removed from memory. Otherwise, no page is dumped and the referenced page is brought into memory, thereby increasing the allocation of the respective process [Sadeh 75].

2.2.2.3 SFIFO Algorithm

The Segmented First In First Out (SFIFO) page replacement algorithm tries to decrease the number of page faults by dividing the memory into two segments, of which one is the primary buffer and the other is the secondary buffer. When a page is removed from the primary buffer, it is placed in the secondary buffer and remains in it until some other page replaces it [Turner and Levy 81]. This algorithm can be viewed as a combination of FIFO and LRU. As the percentage of the primary buffer increases, the

algorithm behaves more like FIFO; and if the percentage of the secondary buffer increases, the algorithm behaves like an LRU algorithm.

2.2.2.4 MLF Algorithm

The Marginal Loss Functions (MLF) page replacement algorithm is a three-level replacement policy, in which, the kernel captures the distribution among many competing processes [Ujaldon et al. 97]. It uses compile time information about an application's access patterns to the kernel. In the first step, the kernel chooses a process that must give up a page by determining the process with least MLF. In the second step, the kernel chooses a segment of the process from which a page has to be removed by determining the MLF of each segment. And in the third step, a page is chosen to be replaced. Compile time analysis is used to insert system calls that determine the MLF of each segment. MLF estimates the number of page faults that a segment would incur if a page is removed from the system. The MLFs of all active segments of a process are added to calculate the total MLF of the process. Then the segment with the least MLF is selected as the victim segment. A number of system calls are inserted to determine the active segments of a process. Finally a page is removed from the victim segment.

2.2.2.5 EELRU Algorithm

Early Eviction LRU (EELRU) is a page replacement algorithm which addresses the situation in which the repeating page fault sequences contain more pages than allocated main memory can hold. So, each page has to be removed from memory before

it is used again if the LRU algorithm is used, because main memory cannot hold that page long enough. But, the removed page has to be loaded into memory again possibly because of a loop or a data structure in the program. Thus all pages have to be loaded into memory each time they are used. When a repeating page fault sequence is detected, EELRU reduces the number of page faults by removing some pages shortly after they are used so as to allow other pages to stay in memory for a longer period of time [Smaragdakis et al. 99].

CHAPTER III

DESIGN AND IMPLEMENTATION ISSUES

3.1 Implementation Platform and Environment

The simulation was implemented on the OSU Computer Science Department's Sun Blade 150, which is a workstation-class computer. The system has 256 mega bytes of RAM. It also has 7.5 giga bytes of hard disk. It runs the Sun OS 5.9 operating system, which is a UNIX-based operating system.

3.2 Objective

The objective of this thesis was to reduce the number of the page faults generated in a paged memory system by separating how instruction pages and data pages are managed. Actual traces and a random trace consisting of virtual address references were used to evaluate the new implementation. The performance of the proposed scheme was evaluated by comparing it with standard algorithms such as FIFO, LRU, LFU, and Second Chance (see Chapter II for details). The performance of the implementation was evaluated on the basis of the number of the page faults generated.

3.3 Input Parameters

3.3.1 Input Traces

The input traces used to evaluate the algorithms in the simulation were obtained from pre-generated traces of virtual memory addresses from the public ftp site of New Mexico State University generated on a SPEC3000 benchmark [Tracebase 94]. Randomly generated traces using a synthetic trace generator [Thiebaut et al. 92] were also used to evaluate the algorithms in the simulation.

3.3.2 Page Frames

In a paged memory management system, a process is assigned a number of page frames in memory based on fixed or variable allocation. Too few page frames will result in a high number of page faults, which may result in thrashing or excessive page traffic with a process making no headway. On the other hand, allocating a large number of page frames to a process will probably result in a very small number of page faults generated, but it will cause low utilization of the memory system.

In the simulation environment used to evaluate the algorithm proposed in this thesis, the user is given an option to choose among 20, 30, and 40 page frames for a process. The total size of the memory can be ignored here because the focus of this research was static, fixed allocation of memory page frames to a program, and the evaluation was done by considering one program (i.e., one trace tape) at a time.

3.3.3 Page Size

The size of a page is the number of words that can be stored in each memory page frame. The typical range of a page size is from 512 words to 8192 words [Hennessy and Patterson 00]. In the simulation environment used to evaluate the design proposed in the thesis, the user is given an option to choose from among three different page sizes: 512, 1024, and 2048.

3.3.4 Page Replacement Algorithm

A page replacement algorithm has to be applied to select a victim page whenever a page fault is caused by the execution of the program and the memory page frames allocated to the program are already used up. In the simulation implemented for this thesis, a page replacement algorithm has to be chosen in three situations: one for data pages, one for instruction pages, and one for both instruction and data pages, when the simulation is run without considering the distinction between instructions and data.

In the simulation environment used to evaluate the design proposed in this thesis, the user is given a choice to choose from among four page replacement algorithms: FIFO, LRU, LFU, and second chance.

3.3.5 Separate Consideration of Instruction Pages and Data Pages

In the design proposed in the thesis, memory page frames allocated to a process are divided into data pages and instruction pages. This is because instructions and data might behave differently. So treating the instructions and the data differently might

produce better results in terms of the number of page faults generated, thus potentially improving the performance of the system. The results (see Chapter IV) show that when the percentage of page frames allocated to data is low, a high number of page faults are generated for data pages. Similarly, when the percentage of pages allocated to data pages is high, a high number of page faults are generated for instruction pages. So the percentage of data frames must be neither too high nor too low for optimum results.

In the simulation environment used to evaluate the design proposed in this thesis, the user can allocate a fixed percentage from the total page frames allotted to a process to data pages. The rest of memory page frames are allocated to instruction pages.

3.4 Design of the Simulation

The simulation was implemented in the C++ language on a Sun Microsystems machine running Sun OS 5.9 operating system. The application runs with traces of references to virtual memory addresses. The traces used were from two sources: 1) input files containing pre-generated traces of virtual memory addresses (generated on a SPEC3000 benchmark obtained from the public ftp site of New Mexico State University [Tracebase 94]), and 2) random traces produced by a synthetic trace generator (explained below).

3.4.1 Random Trace Generation

The random traces were generated by simulating a random walk in a finite address space with references governed by a hyperbolic probability law [Thiebaut et al.

92]. This algorithm works on the basis of locality of the reference and working set size. A random number generator was also used to determine the "type" of each reference, i.e., data or instruction. To produce the type of each reference randomly, a Mersenne twister random number generator was used [Ladd 01]. This algorithm is capable of producing long sequences of order 2^{19937} [Nishimura and Matsumoto 88]. The instruction references were generated with a probability of 0.3. This was determined by calculating the percentage of instruction references among all memory references in each of the five sample input trace files. The average of these values was observed to be approximately 30%. As for the randomly generated traces, different traces can be generated by using different seed values.

3.4.2 Page Table

The Page Table was implemented as a hash table to decrease the search time for a referenced page. The hash key is calculated based on the page number of a referenced page. The structure of each page table entry is shown in Figure 1.

```
struct pageEntry{
  long SAdr; //starting address on disk (32 bits)
  int InsData; //indicates whether a page is instruction or data
            page (1bit)
  int resBit; // residency bit of the page (1 bit)
  int FrameNum; //memory page frame number (8 bits)
};
```

Figure 1. Structure of a page table entry

The memory references in the input traces are 32 bits. So, with each page consisting of at least 512 (the options are: 512, 1024, and 2048) words, a maximum of 23 bits would be sufficient to identify each page uniquely. 32-bit time stamps were used to represent instances of time [Cleary et al. 97].

The entries of the page table implemented in this thesis consist of the following information about each page: time of entry, number of uses, most recent use, etc. This information is required only for the pages residing in main memory. So, a small 2-dimensional array was used to store this information for each page residing in main memory. The structure of an entry in this array is shown in Figure 2.

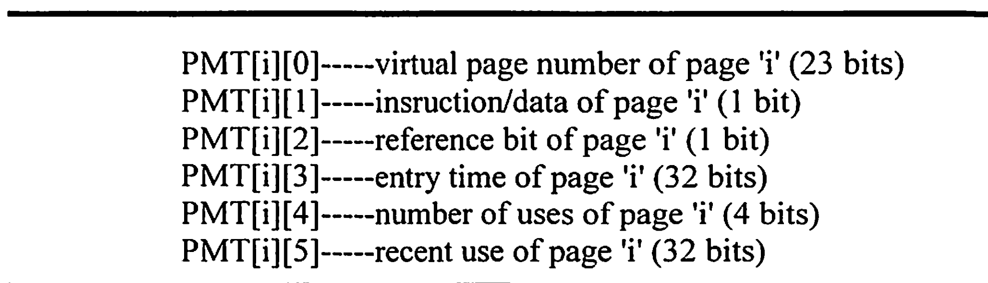

PMT[i][0]-----virtual page number of page 'i' (23 bits)
PMT[i][1]-----insruction/data of page 'i' (1 bit)
PMT[i][2]-----reference bit of page 'i' (1 bit)
PMT[i][3]-----entry time of page 'i' (32 bits)
PMT[i][4]-----number of uses of page 'i' (4 bits)
PMT[i][5]-----recent use of page 'i' (32 bits)

Figure 2. Structure of each entry of the array holding additional information for memory-resident pages

3.4.3 Clock

In the simulation, the clock is incremented after each reference is read from an input trace file or is generated by the random number generator. The clock values are recorded in the PMT (Page Map Table) at the corresponding times when necessary.

3.5 Implementation Detail

The simulation of the page replacement algorithms was done in C++ on the Oklahoma State University Computer Science Department's Sun Blade 150 machine running Sun OS 5.9 operating system. The input traces were taken from the public directory of the ftp site http://tracebase.nmsu.edu/ or randomly generated using random number generators.

The simulation was designed as a menu-based application. First, the user is asked to choose between a random trace and a pre-generated trace. If the user chooses a random trace, random number generators are used and a random trace is generated. In this case, both the referenced addresses and the type of the references (data or instruction) are generated randomly. If the user chooses a pre-generated trace, the user is asked to select a file name from the displayed list of input trace files which contain the reference strings. The lengths of the reference strings in the traces obtained from the public ftp site at New Mexico State University (http://tracebase.nmsu.edu/) are 1 million. So, in order to maintain consistency, the lengths of the reference strings generated randomly were also set to 1 million.

In the next step, the user is asked to choose between running the simulation with a standard algorithm or with separation between instructions and data. If the user chooses to run the simulation using a standard algorithm, the user is asked to choose from among four page replacement algorithms in the menu: FIFO, LRU, LFU, and Second Chance. Then the simulation is run using the selected algorithm. If the user chooses to run the simulation with data and instructions separated, the user is asked to enter a percentage for

the data pages in memory. And then the user is asked to choose between the four available page replacement algorithms for the data pages and for the instruction pages.

In the case of running the simulation with a standard algorithm, each time the page containing referenced address is not found in memory, a page fault is generated and the required page is loaded into memory. Each time a page is loaded into memory, the page table entry for that page is updated. If a page fault occurs and memory is full, a page is chosen to be removed from memory using the selected page replacement algorithm.

In the case of running the simulation with the new design, memory is allocated separately to data pages and instruction pages based on the percentage entered by the user. When a page fault occurs and there are unused page frames allocated to the corresponding segment (i.e., data or instructions), the requested page is loaded into memory into one of the available page frames. If there are no empty page frames allocated to the corresponding segment, a page allocated to that segment is removed from memory using the page replacement algorithm selected for that segment, and the requested page is loaded into memory.

Finally, the number of page faults generated is printed at the console and the user is given a choice to run the simulation again or to exit the system.

The results produced by the simulation were used to produce comparative performance display graphs. These graphs are discussed in Chapter IV.

CHAPTER IV

EVALUATION OF THE SIMULATION

This chapter discusses the performance of the new page replacement algorithm in terms of the number of page faults generated.

4.1 Graphs

A number of graphs were produced depicting the number of page faults generated for various input parameters (Figures 3 and 4). The performance of the new implementation can be evaluated by the graphs produced. The graphs were produced using Microsoft Excel by inserting the averages of the results obtained from the simulation into a spreadsheet. The x-axis of the graphs indicate the percentage of the program's main memory frames allocated to data pages. The y-axis of the graphs indicate the number of page faults generated. The number of page faults generated were measured for the increments of five percent of page frames allocated to data pages in main memory. The values used for the graphs are the averages of the outputs (number of page faults) generated by different input traces while keeping other parameters, i.e., the number of page frames allocated and the number of words per page, constant. The graphs contain the results obtained by the standard page replacement algorithms, i.e., FIFO, LRU, LFU, Second Chance, and also the results obtained by managing instruction pages and data pages separately, using each of the FIFO, LRU, LFU, and Second Chance page

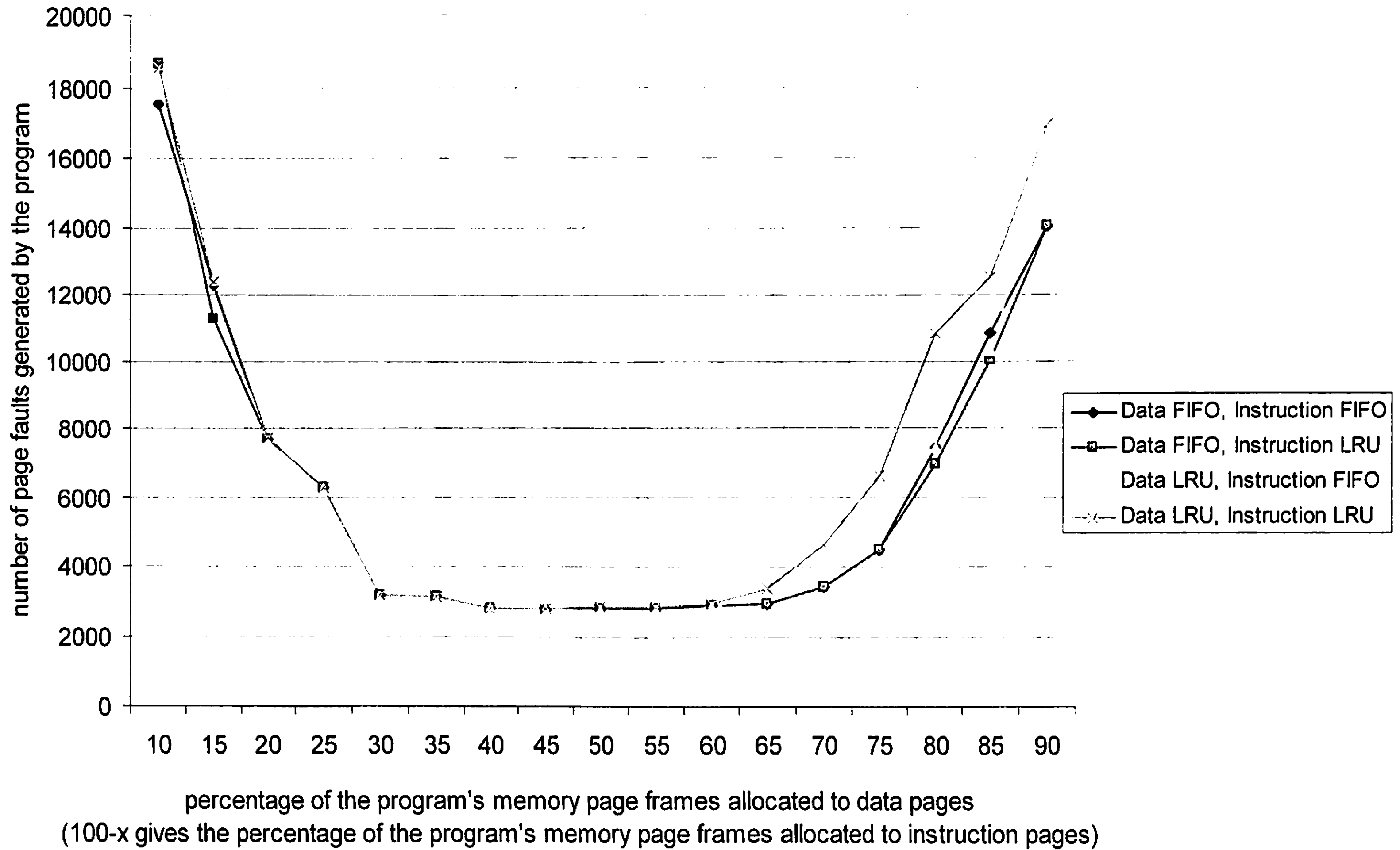
replacement algorithms for the instruction and data parts. Based on the graphs, we can investigate the optimum partition of memory allocated to a program between instructions and data.

4.2 Observations

From the graphs (Figures 3 and 4), it can be observed that better results are produced by the new design in terms of number of page faults generated, both with the random traces and the pre-generated traces by the benchmark programs. The new design generated the smallest number of page faults for a particular allocation of a program's memory frames to instruction pages and data pages. This number is on the average 3.5% less than the minimum value produced by the standard algorithms.

Figure 3 shows the results when the simulation was run with 30 memory page frames with each page containing 1024 words. The inputs used for this simulation are input#1, input#4, and input#7 (see Table III in Appendix C). Among the standard algorithms, the LRU algorithm produced the best results. But when the instructions and the data were managed separately, a decrease in the number of page faults generated was observed for a particular allocation to instruction pages and data pages. A high number of page faults was observed when the data pages were allocated less than ten percent of the total available memory page frames for the program. A decrease in the number of page faults generated was observed with the increase of the allocation until the minimum was reached, which was observed to occur when the data pages are allocated between forty percent and sixty percent of the program's memory page frames, depending on the algorithms and the input traces used. Then the number of page faults generated increased

Results generated by pre-generated traces with 30 memory page frames allocated to the program with 1024 words per page.



The number of page faults generated by standard algorithms for the same memory allocation and page size are: FIFO=2782,LRU=2816, LFU=85210, Second Chance=2782.

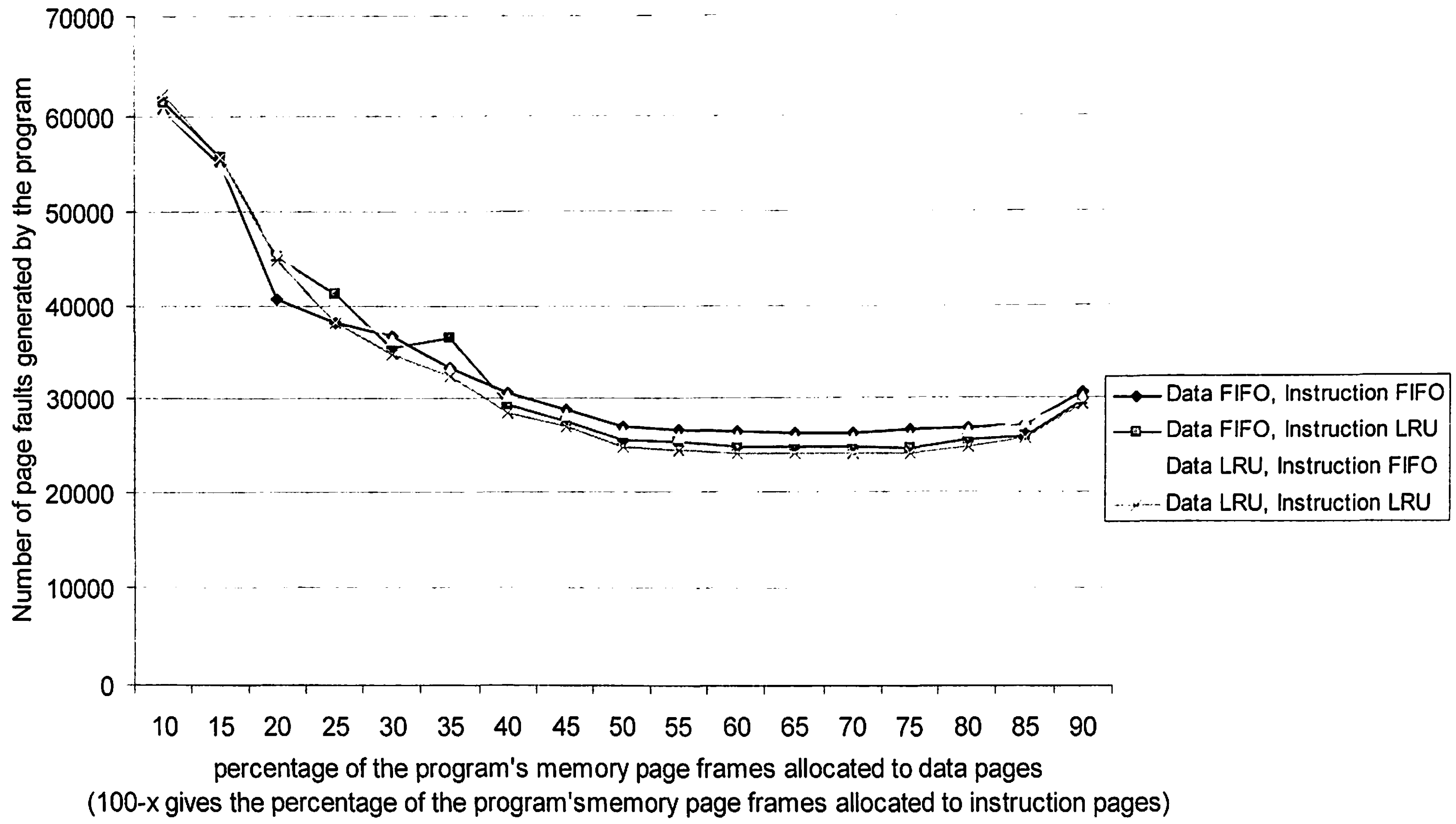
Figure 3. Program performance Graph 1

with the increase in allocation of the program's memory frames to data pages. The results are shown in figures 3 and 4 with FIFO and LRU applied to data parts and instruction parts. Not much difference was observed by changing algorithms for data pages and instruction pages, as all curves in the graphs exhibit identical behavior.

Figure 4 shows the results produced when the simulation was run with 30 memory page frames with each page containing 1024 words. Random traces were used as inputs for the simulation. Among the standard algorithms, the LRU algorithm generated the minimum number of page faults. The behavior was observed to be identical to the behavior depicted in figure 3. The minimum number of page faults generated by the new design was observed to be on the average 3.5% less than the minimum value produced by the standard algorithms.

It was observed that the new design generates fewer page faults both for random traces and pre-generated traces. The new design can improve the performance of a computer system in terms of the number of page faults generated if different page replacement algorithms are used for instruction pages and data pages.

Results generated by random traces with 30 memory page frames allocated to the program with 1024 words per page.



The number of page faults generated by standard algorithms for the same memory allocation and page size are: FIFO=26106, LRU=24803, LFU=38750, second chance=26106.

Figure 4. Program performance Graph 2

CHAPTER V

SUMMARY AND FUTURE WORK

This chapter gives a summary of this thesis report and also discusses the future work that can be done in this area.

5.1 Summary

In Chapter I, the memory scheme of a computer system was discussed. In Chapter II, various memory management schemes such as paging, segmentation, and paged segmentation were discussed. The scheme of paging, which is the main focus of this thesis, was discussed in detail. Chapter III discussed the design and implementation issues. It gave a detailed description of the design of the algorithm with its implementation details. Chapter IV evaluated the performance of the new page replacement algorithm in terms of page faults generated.

This thesis concerned the design and development of algorithms to reduce the number of the page faults generated by separating a program's memory space into instruction pages and data pages. The input parameters provided are the percentage of main memory page frames allocated to data pages, and the algorithms used to handle page replacement for instruction pages and for data pages. The simulation helps in determining the optimal allocation of memory pages to instruction pages and data pages.

The simulation was exercised by using pre-generated inputs (trace tapes) taken from the public ftp site at New Mexico State University (http://tracebase.nmsu.edu/) and by randomly generated traces. It was observed that the new design generated the smallest number of page faults for a particular partition of a program's memory frames to instruction pages and data pages. This minimum value was observed to be on the average 3.5% less than the minimum value produced by the standard algorithms. The new algorithm generates too many page faults when the data or instruction segments were allocated too few page frames, because of the possibility of excessive page traffic or even thrashing.

5.2 Future work

In the design implemented in this thesis, memory page frames are statically allocated to instruction pages and data pages of a program at the beginning of the execution of the program. The design can be modified to allocate memory page frames to data pages and instruction pages based on history data, i.e., the number of page faults generated for instruction pages and data pages in the recent past. It is conceivable that the performance of the system can be further improved if a program's memory resident pages are allocated dynamically based upon the page faults generated in the recent past, by increasing the number of page frames allocated to data pages if data pages cause more page faults than the instruction pages, and vice versa.

In this thesis work, it was assumed that no cache was present. So the performance of the algorithm can be analyzed in the presence of a cache.

The performance of the algorithm can also be analyzed for other memory management systems like paged segmentation, where the data is transferred in pages for each segment. In this case, different algorithms can be used to handle page faults in different segments.

Another area of future work is to investigate the overhead incurred in the process of handling instructions and data separately.

REFERENCES

[Burger 96] Doug Burger, "Memory Systems", *ACM Computing Surveys* , pp. 63-65, Vol. 28, No. 1, March 1996.

[Chu and Opderbeck 76] Wesley W. Chu and Holger Opderbeck, "Analysis of the PPF Replacement Algorithm via a Semi-Markov Model", *Communications of the ACM*, Vol. 19, No. 5, pp. 298-304, May 1976.

[Cleary et al. 97] John G. Cleary, J. A. David McWha, and Murray Pearson, "Timestamp Representations for Virtual Sequences", *Proceedings of the Eleventh Workshop on Parallel and Distributed Simulation,* pp. 98-105, Lockenhaus, Austria, June 1997.

[Hennessy and Patterson 00] John L. Hennessy and David A. Patterson, *Computer Architecture - A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2000.

[Ladd 01] Scott Robert Ladd, "libcoyote - A Library of C++ Tools", *http://www.coyotegulch.com/docs/libcoyote/a00033.html*, Creation date = 10/18/2001, Access date = 09/27/2003.

[Morris 99] John Morris, "The Memory Hierarchy in Modern Processors", *http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/mem_hierarchy.html*, Creation date = ??/??/1999, Access date = 10/03/2003.

[Nishimura and Matsumoto 88] Takuji Nishimura and Makoto Matsumoto, "Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", *ACM Transactions on Modeling and Computer Simulation (TOMACS),* Vol. 8, No. 1, pp. 3-30, January 1998.

[Sadeh 75] E. Sadeh, "An Analysis of the Performance of the Page Fault Frequency (PFF) Replacement Algorithm", *Proceedings of the Fifth Symposium on Operating Systems Principles*, pp. 6-13, Austin, TX, November 1975.

[Silberschatz et al. 03] Abraham Silbershatz, Peter Baer Galvin, and Greg Gagne, *Operating System Concepts*, Sixth Edition, John Wiley & Sons, Inc., New York, NY, 2003.

[Smaragdakis et al. 99] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson, "EELRU: Simple and Effective Adaptive Page Replacement", *Proceedings of the 1999*

ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 122-133, Atlanta, GA, May 1999.

[Thiebaut et al. 92] D. Thiebaut, J. L. Wolf, and H. S. Stone, "Synthetic Traces for Trace-Driven Simulation of Cache Memories", *IEEE Transactions on Computers*, Vol. 41, No. 4, pp. 388-410, April 1992.

[Tracebase 94] An International Trace Archive, *NMSU Tracebase,* New Mexico State University, Las Cruces, NM, 1994.

[Turner and Levy 81] Rollins Turner and Henry Levy, "Segmented FIFO Page Replacement", *Proceedings of the 1981 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 48-51, Las Vegas, NV, September 1981.

[Ujaldon et al. 97] Manuel Ujaldon, Shamik Das Sharma, and Joel Saltz, "Page Replacement Using Marginal Loss Functions", *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pp. 1-12, San Jose, CA, November 1997.

APPENDICES

APPENDIX A

GLOSSARY

Demand Paging: A method of paging in which a page is brought into main memory only when that page has been referenced.

EELRU: The Early Eviction LRU page replacement algorithm removes the pages early from memory if memory is not large enough to hold the entire sequence of recurring patterns. This can reduce the number of page faults by optimally removing some pages.

FIFO: When a page has to be removed from memory because of a page fault, First In First Out removes the page that has been in the memory for the longest time.

LRU: According to the Least Recently Used algorithm, a page that has not been used for the longest time will be removed from the memory when there is a page fault (when the set of frames allocated to the program under consideration have all been used up).

MLF: Marginal Loss Function calculates the number of page faults that a process could incur if a page is removed from memory.

Page: A block of logical memory that is of the same size as a frame in the physical memory.

Segment: A block of logical memory as viewed by a user, e.g., functions or subroutines.

Page Fault: A Page Fault occurs when a page, which a program is trying to access, is not present in memory.

Page Map Table: The data structure that stores the details about all the pages of a program residing in main memory. There are as many page tables as the number of active programs in a computer system.

PFF: Page Fault Frequency is the number of page faults occurring per unit time.

PRAD:	Page Replacement Algorithm for Data.
PRAI:	Page Replacement Algorithm for Instructions.
Trace:	The sequence of memory references generated by a program.
Victim Page:	The page that is selected to be removed from memory as a result of a page fault when the set of page frames allocated to a job are all used up.

APPENDIX B

TRADEMARK INFORMATION

Excel:	A registered trademark of Microsoft Corporation.
Sun Blade 150:	A registered trademark of Sun Microsystems Inc.
Sun OS 5.9:	A registered trademark of Sun Microsystems Inc.

APPENDIX C

EXPERIMENTAL RESULTS

This appendix contains the results generated by separating instructions and data. The number of page faults generated with increments of 5% in the allocation of the data pages is listed. The results produced by applying the FIFO and LRU algorithms to the instruction and data partitions are listed.

Percentage of memory page frames allocated to data pages	Number of page faults generated			
	Data=FIFO Instruction=FIFO	Data=LRU Instruction=FIFO	Data=FIFO Instruction=LRU	Data=LRU Instruction=LRU
10	36005	36005	35960	36040
15	23942	23942	23886	24145
20	14846	14846	14770	14986
25	12119	12119	12050	12134
30	6035	6035	5956	6034
35	5904	5908	5827	5955
40	5289	5289	5210	5288
45	5125	5125	5145	5239
50	5232	5232	5168	5374
55	5315	5315	5226	5403
60	5452	5452	5395	5603
65	5575	5575	6207	6472
70	6526	6522	7975	8938
75	8603	8603	11573	12833
80	14490	13490	14439	21271
85	21259	19678	20232	24641
90	27543	27550	31172	33406

TABLE I: RESULTS PRODUCED USING PRE-GENERATED TRACES WITH 30 MEMORY PAGE FRAMES AND 1024 WORDS PER PAGE WITH INPUT #4 (SEE TABLE 3)

Percentage of memory page frames allocated to data pages	Number of page faults generated			
	Data=FIFO Instruction=FIFO	Data=LRU Instruction=FIFO	Data=FIFO Instruction=LRU	Data=LRU Instruction=LRU
10	60234	60261	61342	62143
15	55102	54642	55710	55722
20	40784	45627	45305	44933
25	38136	39736	41361	38192
30	36707	36344	35531	34712
35	33205	33114	36549	32319
40	30526	30103	29200	28335
45	28744	28213	27482	26931
50	26856	26277	25484	24645
55	26532	25831	25229	24258
60	26244	25694	24703	23938
65	26145	25632	24660	23926
70	26147	25628	24655	23919
75	26457	25842	24506	23959
80	26718	26243	25374	24681
85	27129	27225	25817	25505
90	30450	29909	29589	29315

TABLE II: RESULTS PRODUCED USING RANDOM TRACES WITH 30 MEMORY PAGE FRAMES AND 1024 WORDS PER PAGE

Input #	Corresponding file in the ftp site
1	008.espresso.din.Z
2	013.spice2g6.din.Z
3	023.eqntott.din.Z
4	026.compress.din.Z
5	047.tomcatv.din.Z
6	052.alvinn.din.Z
7	072.sc.din.Z
8	078.swm256.din.Z
9	090.hydro2d.din.Z
10	094.fpppp.din.Z

TABLE III: THE INPUTS TAKEN FROM THE PUBLIC FTP SITE AT NEW MEXICO STATE UNIVERSITY

APPENDIX D

PROGRAM LISTING

This program demonstrates the behavior of the memory system when memory is divided into two parts: a data part and an instruction part. The standard algorithms, i.e., FIFO, LRU, LFU, and Second Chance are also implemented to compare the performance to the new design. The basis of comparison is the number of page faults generated.

```
/*******************************************************************
Student Name:       PAVAN KUMAR ATHOTA

Thesis Title:       REDUCING THE NUMBER OF PAGE FAULTS BY SEPARATING
                    INSTRUCTIONS AND DATA

Adviser:            MANSUR H.SAMADZADEH

Estimated Times:    DESIGNING: 40-50 hrs
                    IMPLEMENTATION: 75-80 hrs
                    TESTING: 150-200 hrs
*******************************************************************/
#include <iostream>
#include <fstream>
#include <string>
#include "randomc.h"
using namespace std;
/******************************************************************

******************************************************************/

////////////////////////////////////////////////////////////////////
//      THE LIST OF ALL STUCTURES USED IS DESCRIBED HERE          //
////////////////////////////////////////////////////////////////////

/******************************************************************
Represents the structure of page table entries.
******************************************************************/
struct pageEntry{
        long SAdr;
        int InsData;
        int resBit;
        int FrameNum;
};

/******************************************************************
This is used in the random trace generation, where all the different
possible memory locations are to be stored in a stack for the algorithm
to produce the random traces.
******************************************************************/
```

```
struct LRUNode{
        long Address;
        LRUNode* Next;
};

//////////////////////////////////////////////////////////////////////////////
//              THE LIST OF ALL GLOBAL VARIABLES USED IS DESCRIBED HERE     //
//////////////////////////////////////////////////////////////////////////////

int ActualDataFrames;                      /*the total memory page frames
                                             allocated to data pages*/

int ActualInstructionFrames;               /*the total memory page frames
                                             allocated to instruction pages*/

int AllocDataFrames;                       /*the total memory page frames
                                             occupied by data pages*/

int AllocInstructionFrames;                /*the total memory page frames
                                             occupied by instruction pages*/

int NumberOfFrames=768;                    /*the total number of page frames
                                             allocated to a process.*/

int NoOfPageFaults=0;                      /*the total number of page faults
                                             generated in a particular method*/

int Clock=0;                               /*the clock variable of the system*/


/************************************************************************
This is the 2-dimensional array representing the additional information
for the page table entries residing in the memory.
PMT[i][0]-----VIRTUAL PAGE NUMBER OF PAGE 'i'
PMT[i][1]-----INSRUCTION/DATA OF PAGE 'i'
PMT[i][2]-----DIRTY BIT OF PAGE 'i'
PMT[i][3]-----ENTRY TIME OF PAGE 'i'
PMT[i][4]-----NUMBER OF USES OF PAGE 'i'
PMT[i][5]-----RECENT USE OF PAGE 'i'
************************************************************************/
int PMT[768][8];

int blockSize=8;                           /*This represents the size of each
                                             page frame in the memory*/

int seedType=36;                           /*This is the seed used to produce
                                             the type of the reference, which
                                             is used in the random trace
                                             generation*/

int mainSeed=45;                           /*this is the seed used to produce
                                             the referenced address, which
                                             is used in the random trace
                                             generation*/

LRUNode* LRUStack;                         /*the stack which contains the list
                                             of all the unique addresses that can
                                             be generated in the random trace*/

/************************************************************************
These are the parameters to the random trace generator.
************************************************************************/
double Theta=2.5;
double A=3;
int memorySize=20000;                      /*the size of the program space to be
                                             used by the random trace generator*/


//////////////////////////////////////////////////////////////////////////////
//      ALL FUNCTIONS USED ARE DESCRIBED HERE                               //
//////////////////////////////////////////////////////////////////////////////
```

```
/**************************************************************
This function updates the LRU Stack after a reference is made. It puts
referenced address at the top of the stack and moves all the addresses
above that address, one location down.
**************************************************************/
LRUNode* UpdateLRUStack(double INDEX,LRUNode* LRUStack);

/**************************************************************
This function generates the next address randomly using a hyperbolic
distribution. This function uses the method mentioned in the thesis report
[illegible].
**************************************************************/
LRUNode* getNextAddress(LRUNode* LRUStack);

/**************************************************************
This is a function to convert the hexa-decimal reference addresses to
decimal addresses.
**************************************************************/
int getInt(const char* al);

/**************************************************************
This is a function to convert the hexa-decimal character to a decimal value.
**************************************************************/
int convert(char);

/**************************************************************
This function returns a page frame, into which the page entering the
main memory can be loaded, depending on the type of the reference and
the algorithm being used.
**************************************************************/
int getPage(char Type, int Algorithm);

/**************************************************************
Uses the First In First Out page replacement algorithm to determine the
victim page, when the memory is full.
**************************************************************/
int FIFO(char type);

/**************************************************************
Uses the Least Recently Used page replacement algorithm to determine the
victim page, when the memory is full.
**************************************************************/
int LRU(char type);

/**************************************************************
Uses the Least Frequently Used page replacement algorithm to determine the
victim page, when the memory is full.
**************************************************************/
int LFU(char type);

/**************************************************************
Uses the Second chance page replacement algorithm to determine the victim page,
when the memory is full.
**************************************************************/
int SecondChance(char type);

/**************************************************************
Finds the referenced page in the memory. If the page is not found, this
generates a page fault and brings the requested page into the memory.
**************************************************************/
int findPage(long PNo, char al, int algorithm);

pageEntry PageTable[100][10];
int bucketLen[100];

/**************************************************************
This is the main function of the simulation. It gives the user various
options to run the simulation and then displays the resulting number of
page faults.
**************************************************************/
```

```
int main(int argc,char* argv[])
{
	/*****This is the starting screen of the simulation*****/
	cout<<"***************************************************"<<endl;
	cout<<"                REDUCING PAGE FAULTS          "<<endl;
	cout<<"        BY SEPARATING INSTRUCTIONS AND DATA   "<<endl;
	cout<<"                                                   "<<endl;
	cout<<"                                                   "<<endl;
	cout<<"                         By                        "<<endl;
	cout<<"                 Pavan Kumar Athota                "<<endl;
	cout<<"                                                   "<<endl;
	cout<<"                                                   "<<endl;
	cout<<"               UNDER THE GUIDANCE OF               "<<endl;
	cout<<"              Dr. MANSUR H. SAMADZADEH       "<<endl;
	cout<<"***************************************************"<<endl;

	ifstream in;				/*The input file of the simulation*/


	char* a1=new char[25];			/*hexa-decimal reference string */

	for(int BuckIndex=0;BuckIndex<100;BuckIndex++)
		bucketLen[BuckIndex]=0;
	char* a1=new char[25];			/*hexa-decimal refernce string */
	char a2;

	char tempch;
	long b123=0;
	int runSim;
	int standard;

	/*****************************************************************
	The main menu which allows the user to run the simulation either with
	the pre-generated traces or with the random traces.
	*****************************************************************/
	cout<<"Choose from the following menu:"<<endl;
	cout<<"1. Run the simulation with pre-generated trace."<<endl;
	cout<<"2. Run the simulation with random trace."<<endl;
	cout<<"3. Exit the simulation."<<endl;
	cin>>runSim;

	switch(runSim)
	{
	case 1 :				/*case in which the user chooses
						  to run the simulation with
						  pre-generated input traces*/
		/*****************************************************************
		The menu which allows the user to run the simulation either with
		standard algorithm or the new design.
		*****************************************************************/
		cout<<"Choose from the following menu:"<<endl;
		cout<<"1. Run the simulation standard algorithm."<<endl;
		cout<<"2. Run the simulation with separation between instruction and
data."<<endl;
		cin>>standard;
		switch(standard)
		{
		case 1:				/*case in which the user chooses
						  to run the simulation with
						  standard algorithm*/
			/**********   The user chooses the algorithm here   *********/
			in.open("inp.txt",ios::in);
			in.get(a2);
			cout<<"Choose one of the standard algorithms"<<endl;
			cout<<"1. FIFO"<<endl;
			cout<<"2. LRU"<<endl;
			cout<<"3. LFU"<<endl;
			cout<<"4. Second Chance"<<endl;
			cin>>alg;
```

```
in.get(a2);
for(int j=0;j<NumberOfFrames;j++)      /*Reset the PMT*/
{
        PMT[j][4]=0;
        PMT[j][2]=0;
}
while(!in.eof())
{
        in.get(tempch);
        in.getline(a1,30,' '); /*get an address from the file*/
        int addr=getInt(a1);
        int PNo=addr/blockSize;
        int memPage=findPage(PNo,'3',alg);
        if(PMT[memPage][4]>0)
        {               /*if the generated address is
                          found in the PMT*/
                PMT[memPage][4]=PMT[memPage][4]+1;
                PMT[memPage][5]=Clock;
        }
        else
        {               /*if a page fault is generated*/
                PMT[memPage][0]=PNo;   /*update the PMT*/
                PMT[memPage][4]=1;
                PMT[memPage][2]=1;
                PMT[memPage][3]=Clock;
                PMT[memPage][5]=Clock;
                NoOfPageFaults++;
        }
        in.getline(a1,30);
        in.get(a2);
        Clock++;
}
Clock=0;
cout<<Clock<<endl;
cout<<"TOTAL PAGE FAULTS****  "<<alg<<"  *****  "<<"  ***
"<<NoOfPageFaults<<endl;
break;
case 2:                         /*case in which the user chooses
                                  to run the simulation with
                                  the new design*/
{

int ReferencedAddr,i,PMTEntryNumber,victim;
int percent,noOfPF;
char type;
in.open("inp.txt",ios::in);
in.get(type);
/****** The user chooses the algorithm for instruction part *****/
cout<<"Choose one of the algorithms for instruction part"<<endl;
cout<<"1. FIFO"<<endl;
cout<<"2. LRU"<<endl;
cout<<"3. LFU"<<endl;
cout<<"4. Second Chance"<<endl;
cons al1,al2;
cin>>InsAlg;

/*******   The user chooses the algorithm for data part   *********/
cout<<"Choose one of the standard algorithms"<<endl;
cout<<"1. FIFO"<<endl;
cout<<"2. LRU"<<endl;
cout<<"3. LFU"<<endl;
cout<<"4. Second Chance"<<endl;
cin>>DataAlg;
al1.val=InsAlg;
al2.val=DataAlg;


/*****************************************************************
user enters the percentage of the main memory page frames to be
allocated to data pages
```

```
*************************************************************/
                cout<<"Enter the percentage of the page frames allocated to data
pages"<<endl;
                for(int BuckIndex=0;BuckIndex<100;BuckIndex++)
                        bucketLen[BuckIndex]=0;
                Clock=0;

                /**************************************************************
                Caluculate the number of page frames allocated to data and
                instruction pages respectively.
                **************************************************************/
                AllocDataFrames=NumberOfFrames*percent/100;
                AllocInstructionFrames=NumberOfFrames-AllocDataFrames;
                ActualDataFrames=0;
                ActualInstructionFrames=0;        //make it so just to know the no
of page faults.
                noOfPF=0;

                for(i=0;i<NumberOfFrames;i++) /*reset the PMT*/
                {
                        PMT[i][4]=0;
                }

                while(!in.eof())
                {                               /*gets the referenced addresses
                                                  one by one and keeps track of
                                                  the page faults*/

                        in.get(tempch);
                        in.getline(a1,30,' ');
                        ReferencedAddr=getInt(a1);
                        long PNo=ReferencedAddr/blockSize;
                        if(type=='2')
                                PMTEntryNumber=findPage(PNo,type,a11.val);
                        else
                                PMTEntryNumber=findPage(PNo,type,a12.val);
                        if(PMT[PMTEntryNumber][4]>0)                    //if the
desired address is in the main memory
                        {               /*if the generated address is
                                          found in the PMT*/
                                PMT[PMTEntryNumber][4]=PMT[PMTEntryNumber][4]+1;
                                PMT[PMTEntryNumber][5]=Clock;
                        }
                        else
                        {               /*if a page fault is generated*/

                                PMT[PMTEntryNumber][0]=PNo;
                                PMT[PMTEntryNumber][2]=1;
                                PMT[PMTEntryNumber][3]=Clock;
                                PMT[PMTEntryNumber][4]=1;
                                PMT[PMTEntryNumber][5]=Clock;
                                noOfPF=noOfPF+1;        /*increment the number of the
                                                          page faults*/

                        }
                        in.getline(a1,30);
                        Clock++;
                        in.get(type);
                } //END FOR THE WHILE LOOP.
                cout<<"Percent "<<percent<<"---> "<<noOfPF<<endl;

                break;}
        default : {cout<<"Error input"<<endl;
                exit(0);}
        }
        in.close();
        break;

    case 2:                             /*case in which the user chooses
                                          to run the simulation random
```

```
                                        input traces*/
        {
                b123=0;
                LRUNode* TEMPNode;
                LRUNode* CreateNode;
                bool endOfFile=false;

                 *******   the random generator used to produce the type of the
 ............    ****

                TRandomMersenne trType=TRandomMersenne(seedType);
                int tempT;
                long addr;
                /***build the LRU stack***/
                LRUStack=new LRUNode();
                LRUStack->Next=NULL;
                LRUStack->Address=1;
                TEMPNode=LRUStack;
                for(int i=2;i<=memorySize;i++)
                {                               /*initializing the LRU stack*/
                        CreateNode=new LRUNode();
                        CreateNode->Next=NULL;
                        CreateNode->Address=i;
                        TEMPNode->Next=CreateNode;
                        TEMPNode=TEMPNode->Next;
                }
                srand(mainSeed);
                b123=0;
                endOfFile=false;


        /**************************************************************
                The menu which allows the user to run the simulation either with
                standard algorithm or the new design.

        **************************************************************/
                cout<<"Choose from the following menu:"<<endl;
                cout<<"1. Run the simulation standard algorithm."<<endl;
                cout<<"2. Run the simulation with separation between instruction
and data."<<endl;
                cin>>standard;
                switch (standard)
                {
                case 1:                         /*case in which the user chooses
                                                  to run the simulation with
                                                  standard algorithm*/
                        {
                        int alg;

                        /**********   The user chooses the algorithm here

**********/

                        cout<<"Choose one of the standard algorithms"<<endl;
                        cout<<"1. FIFO"<<endl;
                        cout<<"2. LRU"<<endl;
                        cout<<"3. LFU"<<endl;
                        cout<<"4. Second Chance"<<endl;
                        cin>>alg;
                        for(int j=0;j<NumberOfFrames;j++)
                        {                       /*reset the PMT*/
                                PMT[j][4]=0;
                                PMT[j][2]=0;
                        }
                        if(b123>100000)
                                endOfFile=true;
                        TEMPNode=LRUStack;
                        for(int i=1;i<=memorySize;i++)
                        {                       /*reset the LRU stack*/
                                TEMPNode->Address=i;
                                TEMPNode=TEMPNode->Next;
                        }
                        srand(mainSeed);
                        while(!endOfFile)
```

```
                {                       /*gets the referenced addresses
                                          one by one and keeps track of
                                          the page faults*/
                        b123++;
                        LRUStack=getNextAddress(LRUStack);
                        addr=LRUStack->Address;
                        int PNo=addr/blockSize;
                        int memPage=findPage(PNo,'3',alg);
                        if(PMT[memPage][4]>0)
                        {               /*if the generated address is
                                          found in the PMT*/
                                PMT[memPage][4]=PMT[memPage][4]+1;
                                PMT[memPage][5]=Clock;
                        }
                        else
                        {               /*if a page fault is generated*/
                                PMT[memPage][0]=PNo;
                                PMT[memPage][4]=1;
                                PMT[memPage][2]=1;
                                PMT[memPage][3]=Clock;
                                PMT[memPage][1]=1;
                                PMT[memPage][5]=Clock;
                                NoOfPageFaults++;
                        }
                        if(b123>100000)
                                endOfFile=true;
                        Clock++;
                }
                Clock=0;
                cout<<"TOTAL PAGE FAULTS****  "<<alg<<"  *****  "<<"  ***
"<<NoOfPageFaults<<endl;
                break;}
        case 2:                         /*case in which the user chooses
                                          to run the simulation with
                                          the new design*/
                {

/*************************************************************
        Make the data pages list and the instruction pages list from the
        available list of references.
*************************************************************/
                int DataAlg,InsAlg;
                int ReferencedAddr,i,PMTEntryNumber,victim;
                int percent,noOfPF;
                char type;
                b123=0;
                endOfFile=false;
                cons al1,al2;

        /*******   The user chooses the algorithm for instruction part  *********/
                cout<<"Choose one of the algorithms for instruction
part"<<endl;

                cout<<"1. FIFO"<<endl;
                cout<<"2. LRU"<<endl;
                cout<<"3. LFU"<<endl;
                cout<<"4. Second Chance"<<endl;
                cin>>InsAlg;

        /**********   The user chooses the algorithm for data part  *********/
                cout<<"Choose one of the standard algorithms"<<endl;
                cout<<"1. FIFO"<<endl;
                cout<<"2. LRU"<<endl;
                cout<<"3. LFU"<<endl;
                cout<<"4. Second Chance"<<endl;
                cin>>DataAlg;
                al1.val=InsAlg;
                al2.val=DataAlg;

/****************************************************************
        user enters the percentage of the main memory page frames to be
```

```
          allocated to data pages
*****************************************************************/
                    cout<<"Enter the percentage of the page frames allocated to
data pages"<<endl;
                    cin>>percent;
                    Clock=0;
                    b123=0;


    *************************************************************
          Calculate the number of page frames allocated to data and instruction
          pages respectively.
*****************************************************************/
                    AllocDataFrames=NumberOfFrames*percent/100;
                    AllocInstructionFrames=NumberOfFrames-AllocDataFrames;
                    ActualDataFrames=0;
                    ActualInstructionFrames=0;
                    noOfPF=0;

                    for(int i=0;i<NumberOfFrames;i++)     /*reset the PMT*/
                    {
                            PMT[i][4]=0;
                    }
                    if(b123>100000)
                            endOfFile=true;
                    TEMPNode=LRUStack;
                    for(int i=1;i<=memorySize;i++)
                    {
                            TEMPNode->Address=i;
                            TEMPNode=TEMPNode->Next;
                    }
                    srand(mainSeed);
                    while(!endOfFile)
                    {                       /*gets the referenced addresses
                                              one by one and keeps track of
                                              the page faults*/
                            b123++;
                            if(noOfPF==49)
                                    b123=b123;
                            LRUStack=getNextAddress(LRUStack);
                            ReferencedAddr=LRUStack->Address;
                            long PNo=ReferencedAddr/blockSize;
                            tempT=trType.IRandom(1,10);
                            if(tempT<8)     /*the data references are generated
                                              with a probabililty of 0.7*/
                                    type='1';
                            else
                                    type='2';
                            if(type=='2')
                                    PMTEntryNumber=findPage(PNo,type,al1.val);
                            else
                                    PMTEntryNumber=findPage(PNo,type,al2.val);
                            if(PMT[PMTEntryNumber][4]>0)
                            {               /*if the generated address is
                                              found in the PMT*/

PMT[PMTEntryNumber][4]=PMT[PMTEntryNumber][4]+1;
                                    PMT[PMTEntryNumber][5]=Clock;
                            }
                            else
                            {               /*if a page fault is generated*/
                                    PMT[PMTEntryNumber][0]=PNo;
                                    PMT[PMTEntryNumber][2]=0;
                                    PMT[PMTEntryNumber][3]=Clock;
                                    PMT[PMTEntryNumber][4]=1;
                                    PMT[PMTEntryNumber][5]=Clock;
                                    noOfPF=noOfPF+1; /*increment the number of
                                                       the page faults*/

                            }
                            Clock++;
```

```
                                        if(b123>100000)
                                                endOfFile=true;
                                        if(b123%1000==0)
                                                cout<<b123<<endl;
                                } //END FOR THE WHILE LOOP.
                                cout<<noOfPF<<endl;
                                int abcdef;
                                cin>>abcdef;
                                delete []LRUStack;
                                break;
                                  }
                        default: {cout<<"Invalid input. Exiting the Simulation...."<<endl;
                                exit(0);}
                        }
                }
        case 3:                                         /*case in which the user chooses
                                                          exit the simulation*/
                exit(0);
        default:                                        /*case when the user enters an
                                                          invalid input*/
                cout<<"Invalid input. Exiting the Simulation...."<<endl;
                exit(0);
        }
                }
        case 3:                                 /*case in which the user chooses
                                                  exit the simulation*/

                exit(0);
        default:                                /*case when the user enters an
                                                  invalid input*/

                cout<<"Invalid input. Exiting the Simulation...."<<endl;
                exit(0);
        }
}


/***************************************************************************
Finds the referenced page in the memory. If the page is not found, this
generates a page fault and brings the requested page into the memory.
***************************************************************************/
int findPage(long PNo, char type, int algorithm)
{
        int keyVal=PNo%100;
        int pos=bucketLen[keyVal];
        for(int i=0;i<bucketLen[keyVal];i++)
        {
                if(((PageTable[keyVal][i].SAdr)/blockSize)==PNo)
                {
                        if(type!='3')
                        {
                        if(PageTable[keyVal][i].InsData==1)
                                type='1';
                        else
                                type='2';
                        }
                        if(PageTable[keyVal][i].resBit==1)
                                return PageTable[keyVal][i].FrameNum;
                        else
                                break;
                }
        }

        int FNo=getPage(type, algorithm-1);
        if(i==bucketLen[keyVal])
        {
                PageTable[keyVal][pos].SAdr=PNo*blockSize;
                if(type=='2')
                        PageTable[keyVal][pos].InsData=2;
                else
```

```
                    PageTable[keyVal][pos].InsData=1;
            bucketLen[keyVal]=bucketLen[keyVal]+1;
            PageTable[keyVal][pos].FrameNum=FNo;
        }
        else
            PageTable[keyVal][i].FrameNum=FNo;
        PageTable[keyVal][i].resBit=1;
        if (PMT[FNo][4]!=0)
        {
            int al=PMT[FNo][0];
            keyVal=al%100;
            for(int i=0;i<bucketLen[keyVal];i++)
            {
                if(((PageTable[keyVal][i].SAdr)/blockSize)==al)
                {
                    PageTable[keyVal][i].resBit=0;
                    break;
                }
            }
        }
    PMT[FNo][0]=PNo;
        if(type=='2')
            PMT[FNo][1]=1;
        else
            PMT[FNo][1]=0;
        PMT[FNo][4]=0;
        return FNo;
}


/****************************************************************
This function generates the next address randomly using a hyperbolic
distribution. This function uses the method mentioned in the thesis report
[Thiebaut et al.92].
****************************************************************/
LRUNode* getNextAddress(LRUNode* LRUStack)
{
        double u;
        int TEMP;
        double INDEX;
        u=((double)rand() / (double)(RAND_MAX+1));
        if(u<(1/Theta))                         /*the case whent he next reference
                                                  is produced within the spatial
                                                  locality of the reference*/
        {
                double a=pow(A,Theta);
                double b=(u*Theta/a);
                double c=(1/(1-Theta));
                INDEX=pow(b,c);
        }
        else                                    /*the case when the reference
                                                  can be produced anywhere in
                                                  the program space*/
        {
                u=( (double)rand() / (double)(RAND_MAX+1) );
                INDEX=u*memorySize;
        }
        if(INDEX>=memorySize)                   /*make index to be in the limits
                                                  of the memory size */
                INDEX=memorySize-1;
        TEMP=INDEX;
        if(INDEX-TEMP<0.5)                      /*round the index to the nearest
                                                  integer*/
                INDEX=TEMP;
        else
                INDEX=TEMP+1;
        if(INDEX<1)
                INDEX=1;
                                                /*update the stack by bringing
                                                  the referenced memory location
                                                  to the top of the stack*/
```

```
        LRUStack UpdateLRUStack(INDEX,LRUStack);
        return LRUStack;
}


/*****************************************************************
This function updates the LRU Stack after a reference is made. It puts
referenced address at the top of the stack and moves all the addresses
at the top of stack, one location down.
*****************************************************************/
LRUNode* UpdateLRUStack(double INDEX,LRUNode* LRUStack)
{
        LRUNode* TEMP=LRUStack;
        LRUNode* TEMPHolder;
        if(INDEX>1)
        {
                for(int i=1;i<INDEX-1;i++)      /*get the address at the required
                                                    index*/
                        TEMP=TEMP->Next;

                /*****************************************************
        move the address at the required index to the top of the stack
        *************************************************************/
                TEMPHolder=TEMP->Next;
                TEMP->Next=TEMPHolder->Next;
                TEMPHolder->Next=LRUStack;
                LRUStack=TEMPHolder;
        }
        return LRUStack;
}

/*****************************************************************
This function returns a page frame, into which the page entering the
main memory can be loaded, depending on the type of the reference and
the algorithm being used.
*****************************************************************/
int getPage(char Type, int Algorithm)
{
        int VictimPage,i;
        if(Type=='3')                   /*this is the case in which there is
                                          no difference between instructions
                                          and data*/
        {
                for(i=0;i<NumberOfFrames;i++)
                {                               /*if a page frame is unused then
                                                  return the index of that page*/
                        if(PMT[i][4]==0 )
                                return i;
                }
        }
        else                                    /*case when there is a partition
                                                  between instructions and data*/
        {
                if(Type!='2' && ActualDataFrames<AllocDataFrames)
                {                               /*if there are free data frames*/
                        for(i=0;i<NumberOfFrames;i++)
                        {               /*find a free data frame and return it*/
                                if(PMT[i][4]==0)
                                {

                                        ActualDataFrames=ActualDataFrames+1;
                                        return i;
                                }
                        }
                }
                if(Type=='2' && ActualInstructionFrames<AllocInstructionFrames)
                {               /*if there are free instruction frames*/
                        for(i=0;i<NumberOfFrames;i++)
                        {                       /*find a free instruction frame and
                                                  return it*/
                                if(PMT[i][4]==0)
```

```
                            {
                                    ActualInstructionFrames=ActualInstructionFrames+1;
                                    return i;
                            }
                    }
            }
    }
/*****************************************************************
[illegible] frame available in the memory, remove a page from
[illegible] selected algorithm, and return the free frame.
*****************************************************************/
        switch(Algorithm)
        {
        case 0 : VictimPage=FIFO(Type);
                 break;
        case 1 : VictimPage=LRU(Type);
                 break;
        case 2 : VictimPage=LFU(Type);
                 break;
        case 3 : VictimPage=SecondChance(Type);
                 break;
        }
        return VictimPage;
}

/*****************************************************************
Use the Second Chance page replacement algorithm to determine the victim page when
memory is full.
*****************************************************************/
int SecondChance(char type)
{
        int min=Clock,victim=NumberOfFrames;
        bool found=false;
        int CheckedPages[768];
        int chekNum=0;
        bool Checked;
        while(!found)
        {                               /*loop until a victim page is found*/
                for(int i=0;i<NumberOfFrames;i++)
                {                       /*find the page that entered the
                                          memory first*/
                        Checked=false;
                        for(int j=0;j<chekNum;j++)
                        {               /*check if the page is already given
                                          a second chance*/
                                if(i==CheckedPages[j])
                                {       /*if the page is found in the checked
                                          list array then it is not considered
                                          in determined the next suitable victim
                                          page*/
                                        Checked=true;
                                        break;
                                }

                        }
                        if(PMT[i][4]!=0 && !Checked)
                        {
                                if(PMT[i][3]<min)
                                {       /*find the page which entered the memory
                                          the first*/
                                        victim=i;
                                        min=PMT[i][3];
                                }
                        }
                }
                if(PMT[victim][2]==0)/*if the reference bit is not set, the
                                          victim page is found*/
                        found=true;
                else
                {                       /*if the reference bit is set clear the
                                          reference bit and add the page to the
```

```
                                   CheckedPages list*/
                        CheckedPages[chekNum]=victim;
                        PMT[victim][2]=0;
                        chekNum++;
                }
                if(chekNum==NumberOfFrames)
                {                          /*if all the pages in the memory had
                                             their bit cleared, get the first page
                                             in the CheckedPages list*/
                        found=true;
                        return CheckedPages[0];
                }
        }
        return victim;
}

/****************************************************************
Uses the First In First Out page replacement algorithm to determine the
victim page when memory is full.
****************************************************************/
int FIFO(char type)
{
        int min=Clock;
        int victim=NumberOfFrames;
        int i;
        int tempChar;
        if(type=='3')                      /*this is the case in which there is
                                             no difference between instructions
                                             and data*/
        {
                for(i=0;i<NumberOfFrames;i++)
                {
                        if(PMT[i][4]!=0)
                        {
                                if(PMT[i][3]<min)
                                {       /*find the page which entered first
                                             into the memory*/
                                        victim=i;
                                        min=PMT[i][3];
                                }
                        }
                }
        }
        else                               /*case when there is a partition
                                             between instructions and data*/
        {
                if(type!='2')
                {                          /*find and return a data page*/
                        if(ActualDataFrames<AllocDataFrames)
                        {                  /*if there are unassigned data page
                                             frames, return an empty page frame*/
                                for(i=0;i<NumberOfFrames;i++)
                                        if(PMT[i][4]==0)
                                        {       /*return the first available empty
                                                     page*/
                                                victim=i;
                                                return victim;
                                        }
                        }
                }
                else                       /*find and return an instruction frame
                                             and return it*/
                        if(ActualInstructionFrames<AllocInstructionFrames)
                        {             /*if there are unassigned instruction page
                                           frames, return an empty page frame*/
                                for(i=0;i<NumberOfFrames;i++)
                                        if(PMT[i][4]==0)
                                        {      /*return the first available empty
                                                    page*/
                                                victim=i;
                                                return victim;
```

```
                                    }
                        }
                        if(type=='2')
                                tempChar=1;
                        else
                                tempChar=0;
                        for(i=0;i<NumberOfFrames;i++)
                        {               /*in case a page has to be removed*/
                                if(tempChar==PMT[i][1] && PMT[i][4]!=0)
                                {       /*find the page that entered the
                                          memory before all other pages*/
                                        if(PMT[i][3]<min)
                                        {
                                                victim=i;
                                                min=PMT[i][3];
                                        }
                                }
                        }
        }
        return victim;          /*return the victim page*/
}

/**************************************************************
Uses the Least Recently Used page replacement algorithm to determine the
victim page when memory is full.
**************************************************************/
int LRU(char type)
{
        int min=Clock;
        int victim=NumberOfFrames;
        int i;
        int tempChar;
        if(type=='3')                   /*this is the case in which there is
                                          no difference between instructions
                                          and data*/
        {
                for(i=0;i<NumberOfFrames;i++)
                {
                        if(PMT[i][4]!=0)
                        {
                                if(PMT[i][5]<min)
                                {       /*find the page which was used least
                                          recently*/
                                        victim=i;
                                        min=PMT[i][5];
                                }
                        }
                }
        }
        else                            /*case when there is a partition
                                          between instructions and data*/
        {
                if(type!='2')
                {                       /*find and return a data page*/
                        if(ActualDataFrames<AllocDataFrames)
                        {               /*if there are unassigned data page
                                          frames, return an empty page frame*/
                                for(i=0;i<NumberOfFrames;i++)
                                        if(PMT[i][4]==0)
                                        {/*return the first available empty
                                          page*/
                                                victim=i;
                                                return victim;
                                        }
                        }
                }
                else                    /*find and return an instruction frame
                                          and return it*/
                {
                        if(ActualInstructionFrames<AllocInstructionFrames)
```

```
                {               /*if there are unassigned instruction page
                                  frames, return an empty page frame*/
                        for(i=0;i<NumberOfFrames;i++)
                                if(PMT[i][4]==0)
                                {   /*return the first available empty
                                        page*/
                                        victim=i;
                                        return victim;
                                }
                }
        }
        if(type=='2')
                tempChar=1;
        else
                tempChar=0;
        for(i=0;i<NumberOfFrames;i++)
        {
                if(tempChar==PMT[i][1] && PMT[i][4]!=0)
                {               /*in case a page has to be removed*/
                        if(PMT[i][5]<min)
                        {
                                victim=i;
                                min=PMT[i][5];
                        }
                }
        }
    }
    return victim;          /*return the victim page*/
}

/**************************************************************
Uses the Least Frequently Used page replacement algorithm to determine the
victim page when memory is full.
**************************************************************/
int LFU(char type)
{
    int min=Clock;
    int min2=Clock;
    int victim=NumberOfFrames;
    int tempChar;
    int i;
    if(type=='3')                   /*this is the case in which there is
                                      no difference between instructions
                                      and data*/
    {
        for(i=0;i<NumberOfFrames;i++)
        {
                if(PMT[i][4]!=0)
                {
                        if(PMT[i][4]<=min && PMT[i][5]<min2)
                        {       /*find the page which is used least
                                  frequently. use LRU to resolve any
                                  conflict*/
                                victim=i;
                                min=PMT[i][4];
                                min2=PMT[i][5];
                        }
                }
        }
    }
    else                            /*case when there is a partition
                                      between instructions and data*/
    {
        if(type!='2')
        {                           /*find and return a data page*/
                if(ActualDataFrames<AllocDataFrames)
                {                   /*if there are unassigned data page
                                      frames, return an empty page frame*/
                        for(i=0;i<NumberOfFrames;i++)
                                if(PMT[i][4]==0)
                                {/*return the first available empty
```

```
                                        page*/
                                                victim=i;
                                                return victim;
                                        }
                        }
                }
                else                    /*find and return an instruction frame
                                          and return it*/
                {
                        if(ActualInstructionFrames<AllocInstructionFrames)
                        {               /*if there are unassigned instruction page
                                          frames, return an empty page frame*/
                                for(i=0;i<NumberOfFrames;i++)
                                        if(PMT[i][4]==0)
                                        {       /*return the first available empty
                                                  page*/
                                                victim=i;
                                                return victim;
                                        }
                        }
                }

                if(type=='2')
                        tempChar=1;
                else
                        tempChar=0;
                for(i=0;i<NumberOfFrames;i++)
                {
                        if(tempChar==PMT[i][1] && PMT[i][4]!=0)
                        {               /*in case a page has to be removed*/
                                if(PMT[i][4]<=min && PMT[i][5]<min2)
                                {       /*LRU is used to resolve the conflict*/

                                        min2=PMT[i][5];
                                        victim=i;
                                        min=PMT[i][4];
                                }
                        }
                }

        }
        return victim;          /*return the victim page*/
}

/*****************************************************************
This is a function to convert the hexa-decimal reference addresses to a
decimal address.
*****************************************************************/
int getInt(const char* a1)
{
        char c;
        int length=(int)strlen(a1);

        int retVal=0;
        for(int i=0;i<length;i++)
                retVal=retVal*16+convert(a1[i]);
        return retVal;
}

/*****************************************************************
This is a function to convert the hexa-decimal character to a decimal
value.
*****************************************************************/
int convert(char c)
{
        int retVal;
        switch(c)
        {
        case '0' : retVal=0;break;
        case '1' : retVal=1;break;
        case '2' : retVal=2;break;
```

```
        case '3' : retVal=3;break;
        case '4' : retVal=4;break;
        case '5' : retVal=5;break;
        case '6' : retVal=6;break;
        case '7' : retVal=7;break;
        case '8' : retVal=8;break;
        case '9' : retVal=9;break;
        case 'a' : retVal=10;break;
        case 'b' : retVal=11;break;
        case 'c' : retVal=12;break;
        case 'd' : retVal=13;break;
        case 'e' : retVal=14;break;
        case 'f' : retVal=15;break;
        default  : retVal=0;break;
        }
        return retVal;
}


/************************** MERSENNE.CPP ****************** AgF 2001-10-18 *
*  Random Number generator 'Mersenne Twister'                              *
*                                                                          *
*  This random number generator is described in the article by             *
*  M. Matsumoto & T. Nishimura, in:                                        *
*  ACM Transactions on Modeling and Computer Simulation,                   *
*  Vol. 8, No. 1, pp. 3-30, 1998.                                          *
*                                                                          *
*  Experts consider this an excellent random number generator.             *
*                                                                          *
*    [illegible] A. Fog. GNU General Public License www.gnu.org/copyleft/gpl.html *
****************************************************************************/

#include "randomc.h"


void TRandomMersenne::RandomInit(long int seed) {
  // re-seed generator
  unsigned long s = (unsigned long)seed;
  for (mti = 0; mti < MERS_N; mti++) {
    s = s * 29943829 - 1;
    mt[mti] = s;}
  // detect computer architecture
  union {double f; unsigned long i[2];} convert;
  convert.f = 1.0;
  if (convert.i[1] == 0x3FF00000) Architecture = LITTLE_ENDIAN;
  else if (convert.i[0] == 0x3FF00000) Architecture = BIG_ENDIAN;
  else Architecture = NON_IEEE;}


unsigned long TRandomMersenne::BRandom() {
  // generate 32 random bits
  unsigned long y;

  if (mti >= MERS_N) {
    // generate MERS_N words at one time
    const unsigned long LOWER_MASK = (1LU << MERS_R) - 1; // lower MERS_R bits
    const unsigned long UPPER_MASK = -1L  << MERS_R;      // upper (32 - MERS_R) bits
    int kk, km;
    for (kk=0, km=MERS_M; kk < MERS_N-1; kk++) {
      y = (mt[kk] & UPPER_MASK) | (mt[kk+1] & LOWER_MASK);
      mt[kk] = mt[km] ^ (y >> 1) ^ (-(signed long)(y & 1) & MERS_A);
      if (++km >= MERS_N) km = 0;}

    y = (mt[MERS_N-1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
    mt[MERS_N-1] = mt[MERS_M-1] ^ (y >> 1) ^ (-(signed long)(y & 1) & MERS_A);
    mti = 0;}

  y = mt[mti++];

  // Tempering (May be omitted):
```

```
  y ^=  y >> MERS_U;
  y ^= (y << MERS_S) & MERS_B;
  y ^= (y << MERS_T) & MERS_C;
  y ^=  y >> MERS_L;

  return y;}


double TRandomMersenne::Random() {
  // output random float number in the interval 0 <= x < 1
  union {double f; unsigned long i[2];} convert;
  unsigned long r = BRandom(); // get 32 random bits
  // The fastest way to convert random bits to floating point is as follows:
  // Set the binary exponent of a floating point number to 1+bias and set
  // the mantissa to random bits. This will give a random number in the
  // interval [1,2). Then subtract 1.0 to get a random number in the interval
  // [0,1). This procedure requires that we know how floating point numbers
  // are stored. The storing method is tested in function RandomInit and saved
  // in the variable Architecture. (A PC running Windows or Linux uses
  // LITTLE_ENDIAN architecture):
  switch (Architecture) {
  case LITTLE_ENDIAN:
    convert.i[0] =  r << 20;
    convert.i[1] = (r >> 12) | 0x3FF00000;
    return convert.f - 1.0;
  case BIG_ENDIAN:
    convert.i[1] =  r << 20;
    convert.i[0] = (r >> 12) | 0x3FF00000;
    return convert.f - 1.0;
  case NON_IEEE: default:
  ;}
  // This somewhat slower method works for all architectures, including
  // non-IEEE floating point representation:
  return (double)r * (1./((double)(unsigned long)(-1L)+1.));}


long TRandomMersenne::IRandom(long min, long max) {
  // output random integer in the interval min <= x <= max
  long r;
  r = long((max - min + 1) * Random()) + min; // multiply interval with random and
truncate
  if (r > max) r = max;
  if (max < min) return 0x80000000;
  return r;}


/***************************** RANDOMC.H ********************** 2001-10-24 AF *
* This file contains class declarations Mersenne Twister random number generators.
*
* Member functions (methods):
* ===========================
* Constructor(long int seed):
* The seed can be any integer. Usually the time is used as seed.
* Executing a program twice with the same seed will give the same sequence of
* random numbers. A different seed will give a different sequence.
*
* double Random();
* Gives a floating point random number in the interval 0 <= x < 1.
* The resolution is 32 bits in TRanrotBGenerator, TRandomMotherOfAll and
* TRandomMersenne. 52 or 63 bits in TRanrotWGenerator. 63 bits in
* TRandomMotRot.
*
* int IRandom(int min, int max);
* Gives an integer random number in the interval min <= x <= max. (max-min < MAXINT).
* The resolution is the same as for Random().
*
* unsigned long BRandom();
* Gives 32 random bits.
* Only available in the classes TRanrotWGenerator and TRandomMersenne.
*
* Further documentation:
```

```
*
* The file randomc.htm contains further documentation on these random number
* generators.
*     © Agner Fog. GNU General Public License www.gnu.org/copyleft/gpl.html
*****************************************************************************/

#ifndef RANDOMC_H
#define RANDOMC_H

#include <math.h>
#include <assert.h>
#include <stdio.h>

class TRandomMersenne {                    // encapsulate random number generator
  #if 1
    // define constants for MT11213A:
    // (long constants cannot be defined as enum in 16-bit compilers)
    #define MERS_N   351
    #define MERS_M   175
    #define MERS_R   19
    #define MERS_U   11
    #define MERS_S   7
    #define MERS_T   15
    #define MERS_L   17
    #define MERS_A   0xE4BD75F5
    #define MERS_B   0x655E5280
    #define MERS_C   0xFFD58000
  #else
    // or constants for MT19937:
    #define MERS_N   624
    #define MERS_M   397
    #define MERS_R   31
    #define MERS_U   11
    #define MERS_S   7
    #define MERS_T   15
    #define MERS_L   18
    #define MERS_A   0x9908B0DF
    #define MERS_B   0x9D2C5680
    #define MERS_C   0xEFC60000
  #endif
  public:
  TRandomMersenne(long int seed) {       // constructor
    RandomInit(seed);}
  void RandomInit(long int seed);        // re-seed
  long IRandom(long min, long max);      // output random integer
  double Random();                       // output random float
  unsigned long BRandom();               // output random bits
  private:
  unsigned long mt[MERS_N];              // state vector
  int mti;                               // index into mt
  enum TArch {LITTLE_ENDIAN, BIG_ENDIAN, NON_IEEE};
  TArch Architecture;                    // conversion to float depends on computer
architecture
  };
#endif
```

VITA (1)

Pavan Kumar Athota

Candidate for the Degree of

Master of Science

Thesis: REDUCING THE NUMBER OF PAGE FAULTS BY SEPARATING INSTRUCTIONS AND DATA

Major Field: Computer Science

Biographical:

Personal Data: Born in Seetharampuram, INDIA, August 27, 1980, the son of Mr. Ramakrishna Athota and Mrs. Prabhavati Athota.

Education: Received the degree of Bachelor of Technology in Computer Science and Engineering from Kakatiya University, Warangal, India, in May 2001; completed the requirements for the Master of Science degree at the Computer Science Department at Oklahoma State University, Stillwater, Oklahoma, in May 2004.

Experience: Web Designer for Star Schools Project under the Earth Science team at Oklahoma State University, from October 2001 to August 2003.