LRU PAGE REPLACEMENT ALGORITHM:
A NEW APPROXIMATION IMPLEMENTATION

By

EUNJAE JUNG

Bachelor of Science

Myong Ji University

Seoul, Korea

1991

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
July 1996

LRU PAGE REPLACEMENT ALGORITHM:

A NEW APPROXIMATION IMPLEMENTATION

Thesis Approved:

Thesis Advisor

Dean of the Graduate College

PREFACE

The purpose of this thesis work was to develop a trace-driven simulation to investigate the viability of applying a splay tree to a page replacement algorithm (new implementation). The basic idea of the splay tree is that frequently accessed items are placed near the root of the tree. This notion is compatible with the basic idea of the LRU page replacement algorithm. Reference strings consisting of virtual addresses were used as input for this simulation. To assess the performance of the splay tree, as applied to the implementation of the LRU page replacement algorithm, it was compared with other implementations of LRU approximations such as the clock algorithm and the additional-reference-bits algorithm. The performance parameters were page fault rate, time and space complexities, and memory utilization.

Four methods (leftmost, rightmost, highest, and LRU leaf) were used to select a victim page in the new implementation. Although the algorithm overhead (i.e., the time and space complexity) was lower in the leftmost and rightmost methods, the number of page faults and the memory utilization were not as good. The highest and LRU leaf methods generated the better results in terms of the number of page faults and memory utilization when compared with the clock and additional-reference-bits algorithms. The LRU leaf method had the demerit that its overhead was high. The highest leaf method, which did not need any hardware support, had the most reasonable result over all performance factors considered. Therefore, the highest leaf method of selecting a replacement victim in the new implementation using a splay tree could be recommended as a page replacement algorithm.

ACKNOWLEDGMENTS

I would like to express special appreciation to my advisor Dr. Mansur H. Samadzadeh. He provided essential guidance and inspiration throughout my thesis work. Dr. Samadzadeh continued to spend endless hours reviewing my work and offering suggestions for further refinement.

I would like to thank my other committee members, Drs. G. E. Hedrick and Blayne E. Mayfield. Their time and effort are greatly appreciated.

Finally, I would like to express my sincere thanks to my family for their continued support. They helped me throughout my MS program. I couldn't have done it without their continued love and support.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 5

2.1 Paging 5
2.2 Page Replacement Algorithms 6
2.2.1 Optimal Algorithm 6
2.2.2 FIFO Algorithm 7
2.2.3 LRU Algorithm 7
2.2.3.1 Second-Chance Algorithm 7
2.2.3.2 Additional-Reference-Bits Algorithm 9
2.3 Splay Tree 10
2.3.1 Splaying 10
2.3.2 Update Operations on Splay Tree 12

III. DESIGN AND IMPLEMENTATION ISSUES 14

3.1 Implementation Platform and Environment 14
3.2 Objectives 14
3.3 Input Parameters 15
3.3.1 Input Traces 15
3.3.2 Process Number 15
3.3.3 Memory Size 15
3.3.4 Page Size 16
3.3.5 Page Fault Handling Time 17
3.3.6 Page Replacement Algorithms 17
3.4 Design of the Simulation 18
3.4.1 New Implementation 18
3.4.2 Clock Algorithm 19
3.4.3 Additional-Reference-Bits Algorithm 20
3.4.4 Scheduling 21
3.5 Implementation Details 22

IV. EVALUATION 27

4.1 Testing 28
4.1.1 Test Traces 28

Chapter Page

4.1.2 Memory Size ... 29
4.1.3 Time Interval ... 33
4.1.4 Result of the Test ... 37
4.2 Analysis ... 38
4.2.1 Graphs ... 38
4.2.2 Observations ... 39
4.2.3 Time and Space Complexities ... 46
4.2.3.1 Space Complexity ... 46
4.2.3.2 Time Complexity ... 47
4.2.3.2.1 Searching ... 48
4.2.3.2.2 Selecting a Victim Page ... 48
4.2.3.2.3 Rebuilding ... 49

V. SUMMARY AND FUTURE WORK ... 51

5.1 Summary ... 51
5.2 Future Work ... 52

REFERENCES ... 53

APPENDICES ... 55

APPENDIX A: Glossary ... 56

APPENDIX B: Trademark Information ... 58

APPENDIX C: Experimental Results ... 59

APPENDIX D: Program Listing ... 72

LIST OF TABLES

Table Page

I Five sampled traces used for the simulation 28

II Minimum number of page faults for different page sizes 29

III The start points of memory sizes yielding minimum page fault numbers in each algorithm 30

IV The number of page faults according to memory size (in the highest leaf method in the new implementation) 31

V The number of page faults according to memory size (in the clock algorithm with interval 28,000) 32

VI The number of page faults according to memory size (in the additional-reference-bits algorithm with interval 140,000) 32

VII The number of page faults and memory utilization when each process has 163 frames (5 * 512 * 163 = 417,280 bytes) 37

VIII The number of page faults and memory utilization when each process has 178 frames (5 * 512 * 178 = 455,680 bytes) 38

IX Space complexity of each algorithm in the worst case 47

X Time complexity of each algorithm 49

LIST OF FIGURES

Figure | Page

1 Basic address mapping mechanism with paging ... 6

2 Clock algorithm ... 8

3.a Zig step ... 10

3.b Zig-zag step ... 11

3.c Zig-zig step ... 11

4.a Insertion of 8 ... 12

4.b Join of the left and right subtrees of node I and deletion of 5 ... 13

5 Data structure of splay tree ... 19

6 Data structure of linked list used to contain leaves ... 19

7 Data structure of circular queue and Hand pointer ... 20

8 Data structure used for 8-bit shift register ... 21

9 Data structure used for page table in additional-reference-bits algorithm ... 21

10 Data structure used for blocked queue ... 22

11 The main menu of the simulation ... 23

12 The number of page faults generated vs. the allocated memory size ... 27

13 The number of page faults generated as affected by the change of regular time intervals in the clock algorithm ... 34

14 Expansion of Figure 13 from 10 to 100,000 ... 34

15 The number of page faults generated as affected by the change of regular time intervals in the additional-reference-bits algorithm ... 36

Figure | Page

16 Expansion of Figure 15 from interval 10 to 1,000,000 36

17 Comparison of page fault numbers for three different algorithms for a page size of 512 and memory allocation of 417,280 bytes 40

18 Comparison of memory occupancy in the three different algorithms for a page size of 512 and memory allocation of 417,280 bytes 41

19 Comparison of page fault numbers in the four different methods used in the new implementation for a page size of 512 and for memory allocation of 417,280 bytes 41

20 Comparison of memory occupancy in the four different methods used in the new implementation for a page size of 512 and for memory allocation of 417,280 bytes 42

21 Comparison of page fault numbers for three different intervals used in the clock algorithm with a page size of 512 and memory allocation of 417,280 bytes 44

22 Comparison of memory occupancy for three different intervals used in the clock algorithm with a page size of 512 and memory allocation of 417,280 bytes 44

23 Comparison of page fault numbers for three different intervals used in the additional-reference-bits algorithm for a page size of 512 and memory allocation of 417,280 bytes 45

24 Comparison of memory occupancy for three different intervals used in the additional-reference-bits algorithm for a page size of 512 and memory allocation of 417,280 bytes 45

25 A typical page table entry 46

CHAPTER I

INTRODUCTION

The memory management of a computer system has a significant effect upon its operating system design [Belady et al. 81] [Deitel 90]. To execute a process, its instructions and data must be stored in main memory. Because of the restricted size of main memory, due to the fact that it is expensive relative to secondary memory, the execution of a process whose address space (i.e., instructions plus data) is larger than main memory is difficult. Also, as multiprogramming has been used to improve the utilization of CPU, a single memory (i.e., only main memory) is not large enough to hold several processes [Silberschatz and Galvin 94]. These problems may be solved by using virtual memory [Belady 66] [Denning 70].

Silberschatz and Galvin state that "virtual memory is a technique that allows the execution of processes that may not be completely in memory" [Silberschatz and Galvin 94]. In this scenario, programs, each of which can be larger than main memory, can be executed. So the programmer does not have to worry about the size of programs. The operating system keeps parts of the programs and data that are currently in use in main memory, and those parts that are not expected to be required soon are kept in secondary memory [Tanenbaum 92]. Virtual memory is specially relevant to multiprogramming environments. Tanenbaum describes that "while a program is waiting for part of itself to be swapped in, it is waiting for I/O and cannot run, so the CPU can be given to another process". In multiprogramming/time sharing systems, each user has the illusion that (s)he has a larger and individual memory of her/his own through the virtual memory scheme [Belady 66] [Denning 70].

The basic idea of the virtual memory concept is separating the virtual addresses referenced in a running process from the real physical addresses in main memory [Deitel 90]. That is, the virtual address space and the real address space are separated. A programmer conceptualizes a program in the virtual address space and the operating system links the program to the real address space locations. Actually, to execute a process, the virtual addresses of a process must be translated to real addresses dynamically [Lister and Eager 93].

Demand paging is frequently used to implement the fetching component of virtual memory management [Silberschatz and Galvin 94]. In paged memory management scheme, the program and the data for each process are partitioned into equal-sized blocks called pages and stored in secondary memory. Main memory is also divided into fixed-sized blocks called frames. The pages and the frames are always the same size [Carr 84] [Silberschatz and Galvin 94] [Tanenbaum 92]. When a process is executing, a page that is immediately needed is swapped into main memory and, unless there is a free frame available, a page deemed not to be needed for a while is swapped out. Thus, if there is no room in main memory for the page that has to be brought in, the operating system must choose a page to be removed from main memory, and replace it with the required page using a page replacement algorithm.

Since Belady's research on page replacement algorithms [Belady 66], many algorithms have been introduced (e.g., see [Deitel 90] and [Silberschatz and Galvin 94]). The LRU (least recently used) replacement algorithm is considered to be close to the optimal algorithm (see Section 2.2 for a detailed discussion of replacement algorithms). The implementation of LRU requires special hardware support, which many systems do not provide, so various LRU approximations are usually used.

Splay tree, which is a self-adjusting binary search tree based on splaying (moving a referenced node to the root of a tree through a sequence of rotations), was developed by Sleator and Tarjan [Sleator and Tarjan 85]. As they claim, "splay tree approximately halves the depth of all nodes along the original path from the accessed node to the root". A splay tree does not require the maintenance of height or balance information. Thus it saves space and is simpler than a balanced tree [Weiss 92]. A splay tree has an amortized bound of O(log n) per operation [Tarjan 83]. It is at least as efficient as a balanced tree and especially good in the case of a long sequence of accesses [Sleator and Tarjan 85] [Udi 89], because a node is likely to be accessed soon again when it is accessed once. Splay tree is practically useful in many applications [Weiss 92].

Trace-driven simulation is one of the methods that can be used to evaluate the performance of a system [Poursepanj 94]. This method uses a dynamic sequence of addresses, which has been compiled during an actual execution, as input instead of actually executing instructions or generating results. Because designers do not have to be concerned about producing correct results or other overhead, they can focus on the performance of the designed system. The trace-driven model is thus frequently used to evaluate the performance of a proposed system.

The main goal of this thesis was to develop a trace-driven simulation to apply a splay tree to a page replacement algorithm. To execute the simulation, reference strings consisting of virtual addresses were used as input. This new implementation was compared to traditional LRU approximation implementations.

The rest of this thesis is organized as follows. Chapter II provides a review of literature related to virtual memory management and splay tree. Chapter III contains the design and implementation issues. Chapter IV discusses evaluation. Finally, Chapter V gives the summary and future work.

CHAPTER II

LITERATURE REVIEW

2.1 Paging

Paging is one of the two common methods of implementing virtual memory (the other being segmentation). The paging method has two roles [Lister and Eager 93]. One is to carry out the address mapping procedure and the other is to transfer pages between main memory and secondary memory. Figure 1 depicts the basic address mapping with paging. CPU sends virtual addresses to MMU (memory management unit) and MMU sends physical addresses to main memory after performing address mapping by means of a page table. The index into the page table is a page number, and the page table has the location of the page frame 'p'' which corresponds to page 'p'. Combining the base address of 'p'' and the page offset 'd' yields the physical address in main memory.

The address translation mechanism can be represented theoretically as a function $f: V \rightarrow P \cup \phi$, where V is the set of page numbers in the logical address space of a process, and P is the set of memory-resident frame numbers for that process [Denning 70]. If $x \in V$ is at location $x' \in P$, $f(x) = x'$, else $f(x) = \phi$, which means that a page fault has occurred. In such a case, the processing of the program is interrupted until $x \in V$ is loaded to yield some $x' \in P$, and $f(x) = x'$. Page replacement algorithms are needed when P is full, and a page fault has occurred.

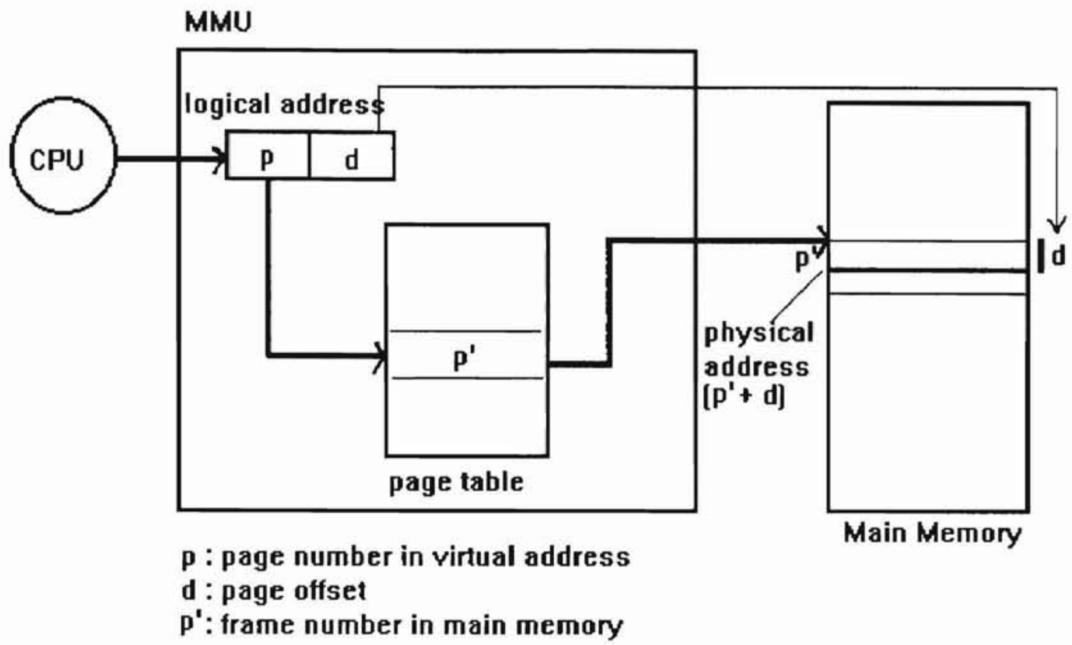

Figure 1. Basic address mapping mechanism with paging

2.2 Page Replacement Algorithms

There are many page replacement algorithms. Usually, generating the lowest page fault rate is considered as the main performance criterion when a replacement algorithm is chosen for an operating system. The following subsections briefly discuss three major page replacement algorithms.

2.2.1 Optimal Algorithm

The optimal algorithm, which is usually referred to as OPT or MIN, replaces the page that is least likely to be used again [Belady 66] [Denning 70]. OPT always has the lowest page fault rate. Since OPT is an ideal algorithm (it requires future knowledge), it cannot be actually implemented. It is used to gauge the performance of other replacement algorithms.

2.2.2 FIFO Algorithm

The FIFO algorithm can be implemented with a FIFO queue that would keep track of all pages in memory [Silberschatz and Galvin 94]. When memory is full while handling a page fault, the page at the head of the queue is removed and the new page is added to the tail of the list. Although the overhead of this algorithm is low, Belady's anomaly [Belady et al. 69] can occur. Belady's anomaly involves a counter-intuitive increase in the page fault rate as a result of increasing the memory size for a program.

2.2.3 LRU Algorithm

The LRU algorithm replaces the page that has not been referenced for the longest time [Tanenbaum 92]. It is based on the idea that the page which has been frequently referenced will probably be called on again in the next few instructions. Although LRU is considered a good approximation to OPT, the implementation is not easy. It can be implemented by adding a counter to the addresses generated by the CPU or by keeping a stack of the page numbers. Both methods have high overheads with or without hardware support. Operating system designers use LRU approximation algorithms (as discussed in the following subsections) that are less expensive in terms of software and hardware overhead.

2.2.3.1 Second-Chance Algorithm

The second-chance algorithm can be considered as a variation of the FIFO algorithm [Deitel 90]. In the FIFO algorithm, although the oldest page (i.e., the page that is at the head of the FIFO queue) is heavily used, it must be replaced unconditionally. The second-chance algorithm can prevent this kind of weakness of the FIFO algorithm. It investigates the reference bit (X bit) of the oldest page. If the bit is 0, the page is replaced. If it is 1, the page gets a second chance. When a page gets a second chance, the page is moved to the end of the FIFO queue and the X bit of the page changes to 0. The load time of the page also changes to the current time. These steps are repeated until the oldest page whose X bit is 0 is found. The second-chance algorithm searches for the oldest page which has not been referenced in the previous time interval (e.g., every 20 or 100 milliseconds).

The approach that uses a circular queue instead of a FIFO queue to implement the second-chance algorithm is called the clock algorithm. As shown in Figure 2, a circular queue shaped like a clock holds the pages of a particular process that reside in main memory, and a hand indicates the oldest page. If the X bit of the oldest page is 1, the bit changes to 0 and the hand goes to the next page. These steps are repeated until the page whose X bit is 0 is found. When such a page is found, the new page is inserted at that position and the hand goes to the next page.

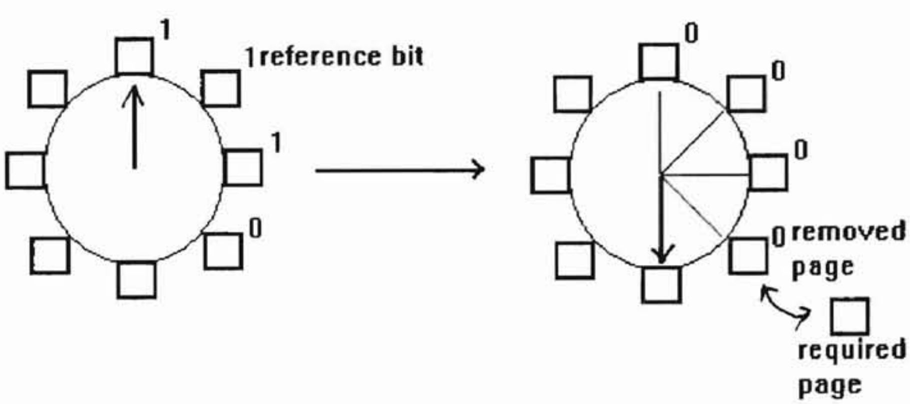

Figure 2. Clock algorithm

In the worst case, if all pages in the FIFO queue or the circular queue have been referenced in the previous time interval, each of them gets a second chance. Therefore, the above algorithms (one using a FIFO queue and the other using a circular queue) repeat until all elements have an X bit of 0; in this case those algorithms emulate the FIFO algorithm.

2.2.3.2 Additional-Reference-Bits Algorithm

Another approximation to LRU is the additional-reference-bits algorithm. The ordering information of references for each page can be partially captured by keeping an 8-bit shift register for each page in a page table [Silberschatz and Galvin 94]. The shift register records the X bits (reference bits) for each page at each time interval as follows. At each clock interrupt, the shift register is shifted right 1 bit and the current X bit is inserted as the leftmost bit. If there is a need for a victim selection for replacement, the page with the lowest value in the shift registers, which roughly indicates whether it has or has not been used recently, is replaced. If a number of pages have the same lowest value for their respective shift registers, this algorithm either chooses one of them or replaces all of them.

The additional-reference-bits algorithm has two main problems that distinguish it from a true LRU algorithm. One problem is that the order of pages referenced during a single time interval cannot be determined because only one bit is recorded per time interval. If some pages were referenced towards the end of a certain interval and one of them must be replaced, the page with the lowest value will be replaced even though it might not be the earliest one referenced in that interval. The other problem is that it cannot distinguish between the pages referenced shortly before 8 time intervals ago (e.g., 9 or 10 time intervals ago) and the pages referenced long time ago (e.g., 100 or 1,000 time intervals ago), because the ordering information is limited to eight instances with an eight-bit shift register.

2.3 Splay tree

When a sequence of access operations is carried out on a binary search tree, if the frequently accessed items can be placed near the root of the tree, the total access time can be reduced. On the assumption that the accessed items are likely to be accessed again soon, Sleator and Tarjan devised a method of restructuring the tree after each access that moves the accessed item to the root [Sleator and Tarjan 85]. They also developed an implementation of splaying. The following two subsections describe splaying and its implementation [Sleator and Tarjan 85] [Weiss 92].

2.3.1 Splaying

To splay a tree at item I (Figure 3), the following steps are repeated bottom-up along the access path until I is the root of the tree. In Figures 3.a, 3.b, and 3.c, the circles indicate single nodes and the triangles indicate subtrees.

1. Zig: If the parent of I, P, is the root of the tree, rotate the edge joining I and the root. This is the last rotation along the access path.

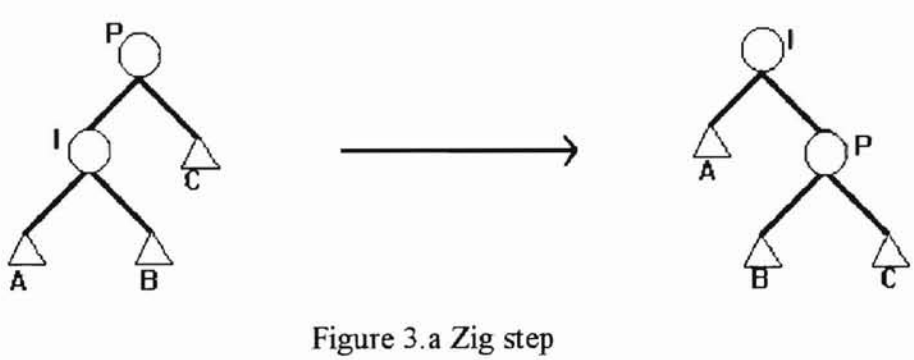

Figure 3.a Zig step

2. Zig-Zag: If the grandparent of I, G, exists and I is right child of P and P is left child of G (or vice versa), rotate the edge joining I and P and then rotate the edge joining I and G.

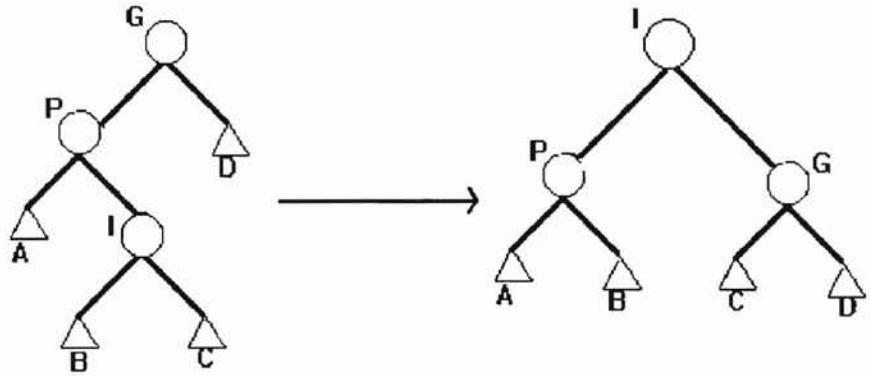

Figure 3.b Zig-zag step

3. Zig-Zig: If the grandparent of I, G, exists and I and P are either both left children of G or both right children of G, rotate the edge joining P and G and then rotate the edge joining I and P.

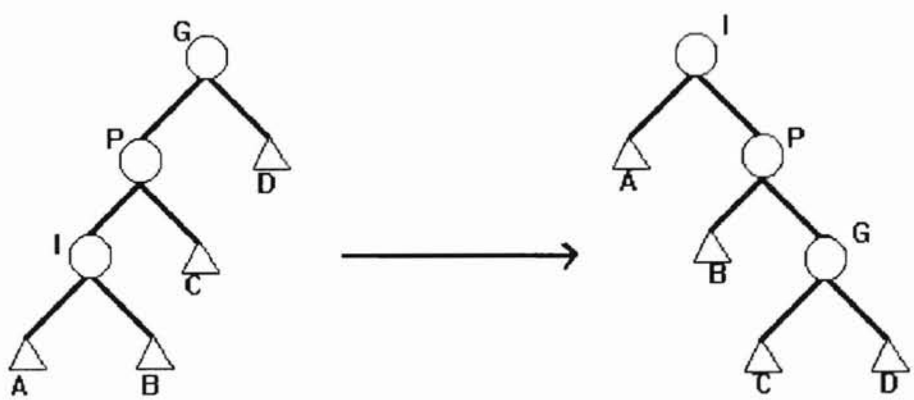

Figure 3.c Zig-zig step

2.3.2 Update Operations on Splay Tree

The standard update operations on a binary search tree can be implemented using splaying as outlined below [Sleator and Tarjan 85].

Insert(x,t): To insert item x in tree t, search t for x and then replace the null pointer reached during the search by a pointer to a new node containing x, and finally splay the tree at the inserted node. Figure 4.a depicts an insertion.

Join(t1,t2): Let's assume that all items in tree t2 are greater than all those in tree t1. To combine t1 and t2 into a single tree, search for the largest item x in t1 and make the root of t1 contain x. Then make t2 the right subtree of the root. Figure 4.b shows how the right and left subtrees of node I are joined.

Delete(x,t): To delete item x from tree t, search t for the node I containing x and then replace I with the root, R, of the subtree that will result if the right and left subtrees of I are joined. Finally, splay the tree at the parent of R. Figure 4.b depicts a deletion.

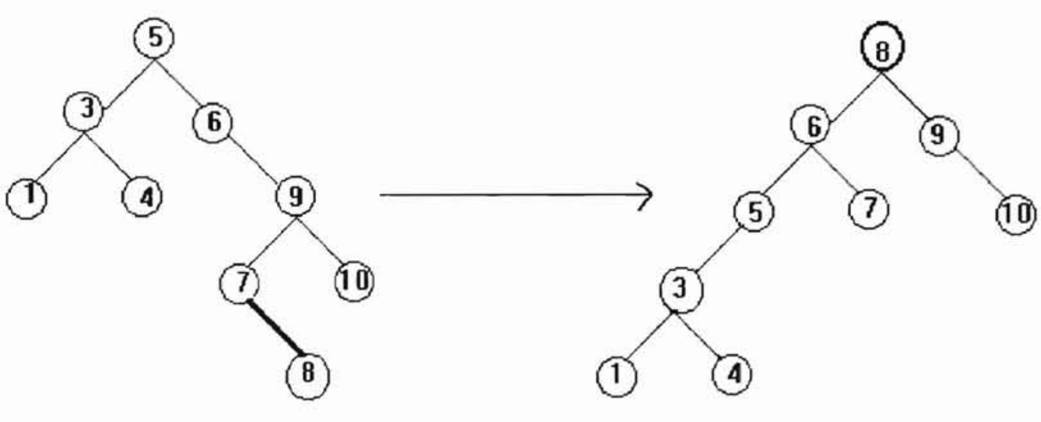

Figure 4.a Insertion of 8

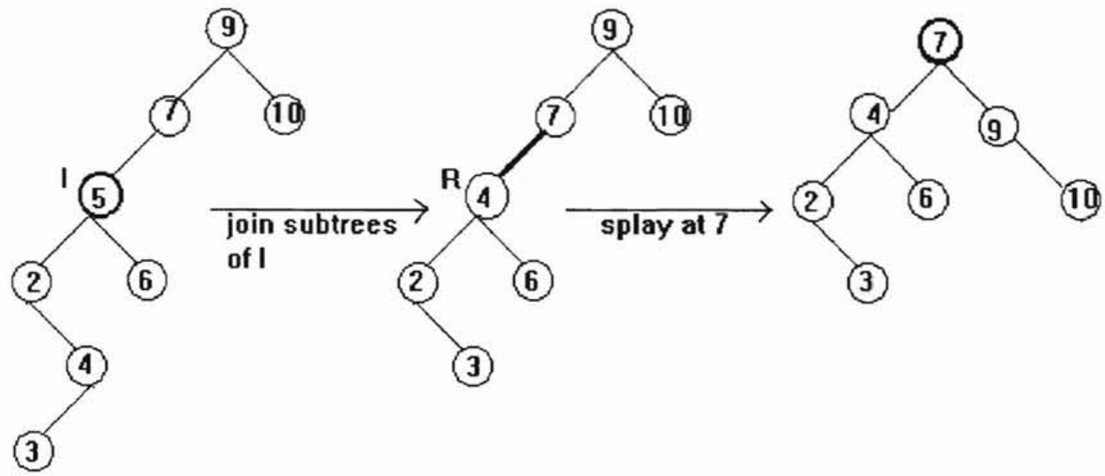

Figure 4.b Join of the left and right subtrees of node I and deletion of 5

CHAPTER III

DESIGN AND IMPLEMENTATION ISSUES

3.1 Implementation Platform and Environment

The simulation program was implemented on a Sequent Symmetry S/81 in C. The Symmetry S/81 is a mainframe-class multiprocessor system which has a parallel architecture using multiple industry-standard microprocessors [Sequent 90]. In its present configuration, this system has twenty four 80386-20MHZ processors. It also has 104 mega bytes of RAM and 5 giga bytes of total hard disk storage. Each process contains 64K of cache memory. It runs the DYNIX/ptx or DYNIX V3.0 operating system that has been engineered to incorporate parallel processing features. DYNIX V3.0 supports both UNIX System V command sets and the Berkeley UNIX, however DYNIX/ptx is compatible with AT&T System V3.2 only.

3.2 Objective

The main goal of the thesis was to develop a trace-driven simulation to apply a splay tree as a data structure to implement an LRU approximation page replacement algorithm. Reference strings consisting of virtual addresses were used as input to this simulation. The performance of this new implementation was evaluated by comparing it with two popular LRU approximation algorithms, namely the clock algorithm and the additional-reference-bits algorithm. The performance factors for the evaluation were number of page faults, memory utilization, and time and space complexities.

3.3 Input Parameters

3.3.1 Input Traces

The traces used as input to the simulation were developed at the Parallel Architecture Research Laboratory of New Mexico State University. They were available in the public directory of the `ftp` site `tracebase@nmsu.edu`.

3.3.2 Process Number

The number of processes is limited to ten (i.e., the maximum degree of multiprogramming is ten). Each process handles one file, which consists of a different reference string. A user can select the number of processes through a standard input.

3.3.3 Memory Size

A critical parameter in the simulation is the memory size. Excessively large memory results in no page faults and excessively small memory results in thrashing (the typical range for the miss rate is from 0.00001% to 0.001% [Hennessy and Patterson 90]). The degree of multiprogramming is constrained as a consequence of the availability of a limited number of traces. The memory size too indirectly depends on the traces.

In the absence of historical data, the same traces that were used to drive the simulation, were used in a pre-processing step to determine a plausible memory size. The necessary memory size for running each process was obtained by gradually increasing the memory size and considering the start point for each process at which the number of page faults generated becomes stable (i.e., would not decrease) in the face of further increasing the memory size. The memory size can also be selected through a standard input.

Having determined the memory size necessary for each process, two methods were used to arrive at the overall memory size for the simulation. The first method consisted of three steps. The first step was to take the median values among the start points of all processes obtained by using each approach of each different algorithm. The second step was to calculate the average value of these median values. The final step was to decide the memory size based on the above two steps. The memory size was *average value * number of processes * page size* because the memory was equally partitioned to each process for the simulation.

The second method had also consisted of three steps. The only a difference was in the first step compared to the first method. The average value of each process was taken instead of the median value (with the minimum and maximum values excluded as possible outliers).

3.3.4 Page Size

The typical range of a page size is from 512 bytes to 8192 bytes [Hennessy and Patterson 90]. Four different page sizes (i.e., 512, 1024, 4096, or 8192) can be selected by a user.

3.3.5 Page Fault Handling Time

The service time required to handle a page fault is the page fault handling time. When a page fault occurs, the relevant page must be read from secondary memory and the desired position of the page must be accessed [Silberschatz and Galvin 94]. There are three primary services that need to be performed during a page fault. A service for the page fault interrupt, a service for reading in the page, and finally a service for restarting the process. The second service time is much more than the other two service times. The typical range of memory access times and page fault handling times are from 1 to 10 and from 100,000 to 600,000 clock cycles, respectively [Hennessy and Patterson 90]. Although the page fault handling time for the simulation was fixed, it can be given by a user differently through a standard input. For the simulation, the default memory access time and page fault handling time are 1 and 10,000 clock cycles, respectively.

3.3.6 Page Replacement Algorithms

To investigate the performance of the new LRU approximation implementation comparatively, two LRU approximation algorithms (i.e., clock and additional-reference-bits) were also implemented. A user can select any of the three algorithms and observe its performance by comparing it with the performance of the other two algorithms.

There are four different methods which a user can select to implement the new page replacement algorithm. First, the leftmost leaf page in a splay tree is replaced when a page replacement is needed. Second, the rightmost leaf page is replaced. Third, the highest leaf page in a splay tree is replaced. Fourth, the LRU page among the leaves is replaced.

The other two algorithms also have different methods by changing the time intervals. Therefore, each implemented version of each algorithm also can be compared with the other versions of the same algorithm. The best implemented version of each algorithm was compared with the best one of the other algorithms when checking the performance of three algorithms. Regular time intervals were assigned when clock and additional-reference-bits algorithms were executed.

3.4 Design of the Simulation

The simulation was implemented as a trace-driven model on the Sequent Symmetry S/81 machine running the DYNIX/ptx operating system using the C programming language.

3.4.1 New Implementation

Splay tree was used as a data structure to implement the new LRU page replacement algorithm. Each node of the splay tree represents a page which is in main memory. The page table size is thus variable. Since there are no actual address spaces, there is no a priori information about the page table sizes such as the total number of pages for each program, as a result there is no simulated disk.

Each process has its own page table which is linked in the form of a splay tree. Figure 5 gives the data structure used in simulating a page table. A parent pointer was needed to do a bottom-up pass over an access path when splaying. The original splay tree does not have to have a height field to compute the height of each node. But when the method which determines the victim page as the highest leaf was used, the height field was needed to compare the height of leaves. To implement the method, which determines LRU leaf as the victim page, all leaves in a splay tree must be linked and implemented as a queue. Figure 6 gives the data structure used for the list which links all leaves to determine the LRU leaf among the leaves.

```
struct stree {
        int         page_num;
        int         height;
        struct stree     *right;
        struct stree     *left;
        struct stree     *parent;
};
typedef struct stree PAGE_TABLE1;
```

Figure 5. Data structure of splay tree

```
struct leaf_list {
        struct stree        *leaf;
        int                 e_flag;
        struct leaf_list    *next;
};
typedef struct leaf_list LEAF_L;
```

Figure 6. Data structure of linked list used to contain leaves

3.4.2 <u>Clock Algorithm</u>

A circular queue was used to implement the clock algorithm. Each node of the queue represents a page which is in main memory. Figure 7 depicts the data structure used in simulating a circular queue. The rbit field which denotes the reference bit of each page is set when a page is inserted into main memory or referenced, and is cleared after the regular time interval. The Hand pointer [Tanenbaum 92] points to the oldest page among the pages in the circular queue (see Subsection 2.2.3.1 for further explanation).

```
struct circular_que {
        int       rbit;
        int       page_num;
        struct circular_que      *next;
};
typedef struct circular_que PAGE_TABLE2;
PAGE_TABLE2         *Hand[MAX_PROCESS];
```

Figure 7. Data structure of circular queue and Hand pointer

3.4.3 Additional-Reference-Bits Algorithm

A linked list was used to simulate the page table of each process for the additional-reference-bits algorithm. Figure 8 is the data structure used to simulate the 8-bit shift register. The bit field was needed to shift 1 bit and to change the leftmost bit. The victim page is the page that has the lowest value for its shift register. When several pages have the same lowest value, the page which had been inserted first in the linked list among the pages is replaced. To do this, a new page is inserted at the tail of the linked list. Figure 9 gives the data structure used to simulate a page table. The 8-bit shift register is kept as one field of the page table to get the ordering of page references. When a page is referenced, the leftmost bit of the shift register is set. The shift register is shifted right 1 bit at each time interval. The value of the shift register indicates the ordering of page references. To get this value, union is used.

```
struct s_reg {
        unsigned int unused:7;
        unsigned int first:1;
};
typedef struct s_reg   SHIFT_REGISTER
```

Figure 8. Data structure used for 8-bit shift register

```
struct add_ref {
        int       page_num;
        union shift {
                unsigned int    value:8;
                SHIFT_REGISTER   reg;
        } shift_reg;
        struct add_ref *next;
};
typedef struct add_ref   PAGE_TABLE3
```

Figure 9. Data structure used for page table in additional-reference-bits algorithm

3.4.4 Scheduling

Two types of random number generators were used for the scheduling of the processes. One was used to select a process number and the other was used to select a length of the reference string according to which the selected process would make progress. If a page fault occurs during process execution, the process is blocked and another process is selected and executed as much as dictated by the random amount generated for headway. At this time, the process which has finished handling a page fault in the blocked queue has priority to be the next process to be run. If no process is finished with its page fault handling, the next process is selected at random from among the processes that are not blocked. There are two situations when a page fault occurs. One is when the page, which will be referenced next, does not exist in the memory even though the memory is not full. The other situation is when the memory is full and the page which will be referenced next is not memory resident. Figure 10 depicts the data structure used to implement the blocked queue.

```
struct  blocked_que {
        int       process_id;
        int       enter_time;
        struct blocked_que *next;
};
typedef struct blocked_que BLOCK_Q;
```

Figure 10. Data structure used for blocked queue

3.5 Implementation Details

The simulation is menu driven. Figure 11 gives the main menu of this simulation. Input traces of the simulation consist of virtual page numbers. These pages were obtained by converting virtual addresses to page numbers before the simulation was performed. The files containing the virtual addresses in dinero+ format were obtained by using anonymous `ftp`. This dinero+ format is a common format used for capturing and representing traces defined at `/pub/tracebase4/r3000/README` of the `ftp` site `tracebase@nmsu.edu` as follows: "in addition to the usual type and address fields, a third field is present that lists the instruction word for instruction fetches". These files had been compressed using the "compress" command. They were decompressed by using the "uncompress" or "gunzip" command. The virtual pages were obtained by dividing virtual addresses by a certain page size. The names and the lengths of the converted files were stored in the "traces.dat" file.

```
MENU

1. Convert virtual addresses to virtual pages.
2. Perform the simulation.
3. Generate graph for page faults.
4. Generate graph for memory utilization.
5. Exit the simulation.
```

Figure 11. The main menu of the simulation

Several input parameters (i.e., number of processes, memory size, method of memory allocation, page fault handling time, page size, and page replacement algorithm) are given by a user to perform each simulation. The process (i.e., process trace) to be executed and the length of the corresponding reference string to be progressed are obtained by calls to random number generators. At each clock (i.e., virtual memory access time), a page is referenced and the number of page faults is computed. Every 500 virtual clock cycles, the memory occupancy of each process is considered for memory utilization. To determine whether a page is in main memory, an examination of the page table is required. If a page fault occurs, the running process gets blocked. The next unblocked process to be executed is randomly selected if no process that is in the blocked queue consumes its page fault handling time. If no page fault occurs, the running process proceeds until finishing the trace length previously obtained randomly.

Each process has its own work space. These work spaces are the same because the total memory is equally partitioned among the active processes. Each page replacement algorithm has its own page table implemented by a different data structure. Each page table is updated in a different way and has its own page replacement algorithm.

In the case of the new implementation, when no page fault occurs (i.e., when the page that is referenced is in the splay tree), the splay tree is reconstructed using splaying at the node containing the page. Otherwise, the tree is reconstructed after the page is inserted as a leaf node.

In the case of the clock algorithm, when no page fault occurs (i.e., when the page that is referenced is in the circular queue), the reference bit of the page is set. Otherwise, the page is inserted into the circular queue. Finally, in the case of the additional-reference-bits algorithm, when no page fault occurs, the leftmost bit of the page is set. Otherwise, the page is inserted into the linked list while setting the leftmost bit.

When a page fault occurs and there is no more memory available, a victim page is determined followed by a page replacement algorithm. In the case of the new implementation, four different methods were used to determine a victim page. The victim page is one of the leaves in the splay tree in all the four cases. The method of choosing a victim page as a leftmost leaf or a rightmost leaf is straightforward and the algorithm overhead is lower than the two other methods. The height field of each node is not necessary in the leftmost, rightmost, or the LRU leaf choosing method. The method of choosing the farthest leaf from the root needs the height field to compare the distance of each node from the root. The height of a node is the distance from the root to the node. The root of a splay tree is an MRU (most recently used) page and the frequently accessed pages are placed near the root of the tree. Therefore, the leaf that has the highest height can be approximately considered an LRU page. The heights of all the nodes of a splay tree should be computed to know the height of each leaf node. To compute the height of a certain node, the height of the parent node is needed. So the preorder tree traversal strategy is used. When several leaves have the same highest height, the leftmost leaf is selected among those leaves. The last method attempts to get the leaf which is an LRU page from among the current leaves. A queue linking all the leaves is used. To link all the leaves, this method also traverses the entire tree. If a current leaf was not a leaf at the previous state, the leaf is inserted at the tail of the queue. If the page which was a leaf at previous state is not a leaf currently, then the page is removed from the queue. The head of the queue (i.e., the page that has stayed the longest as a leaf among the pages in the queue) is considered as the LRU leaf.

The graphs, which express the memory utilization and the number of page fault for each process, were generated by using BLT routines. `Tcl` library and `blt-wish` installed in the `/contrib/bin` directory are needed to use BLT on the Oklahoma State University Computer Science Department's Sequent Symmetry S/81 running DYNIX/ptx. This path and the environment variable must be set in `.login` file (for `/bin/csh` users) or `.profile` file (for `/bin/sh` or `/bin/ksh` users) [Ousterhout 94]. The following commands are for `csh` users.

```
set path=(.... /contrib/bin .......... .)
setenv TCL_LIBRARY /contrib/lib/tcl
setenv TK_LIBRARY /contrib/lib/tk
```

The X co-ordinates of the graphs represent process numbers, time intervals, or the number of frames allocated as the domain of each graph. The Y co-ordinates of the graphs show the number of page faults or the memory occupancy which are given as the results of the simulation. A graph is shown on the screen (Figure 12) after giving the values of X and Y co-ordinates. This sample graph shows the change of the number of page faults affected by the change of memory size in the new implementation using the highest leaf method of victim page selection. There are two buttons (i.e., Print and Quit) in the graph.

If the Print button is pushed, a postscript format file is generated to print the graph. The name of a postscript file is provided by the user. If the Quit button is selected, the system terminates displaying the graph returns to the main menu of the simulation (Figure 11).

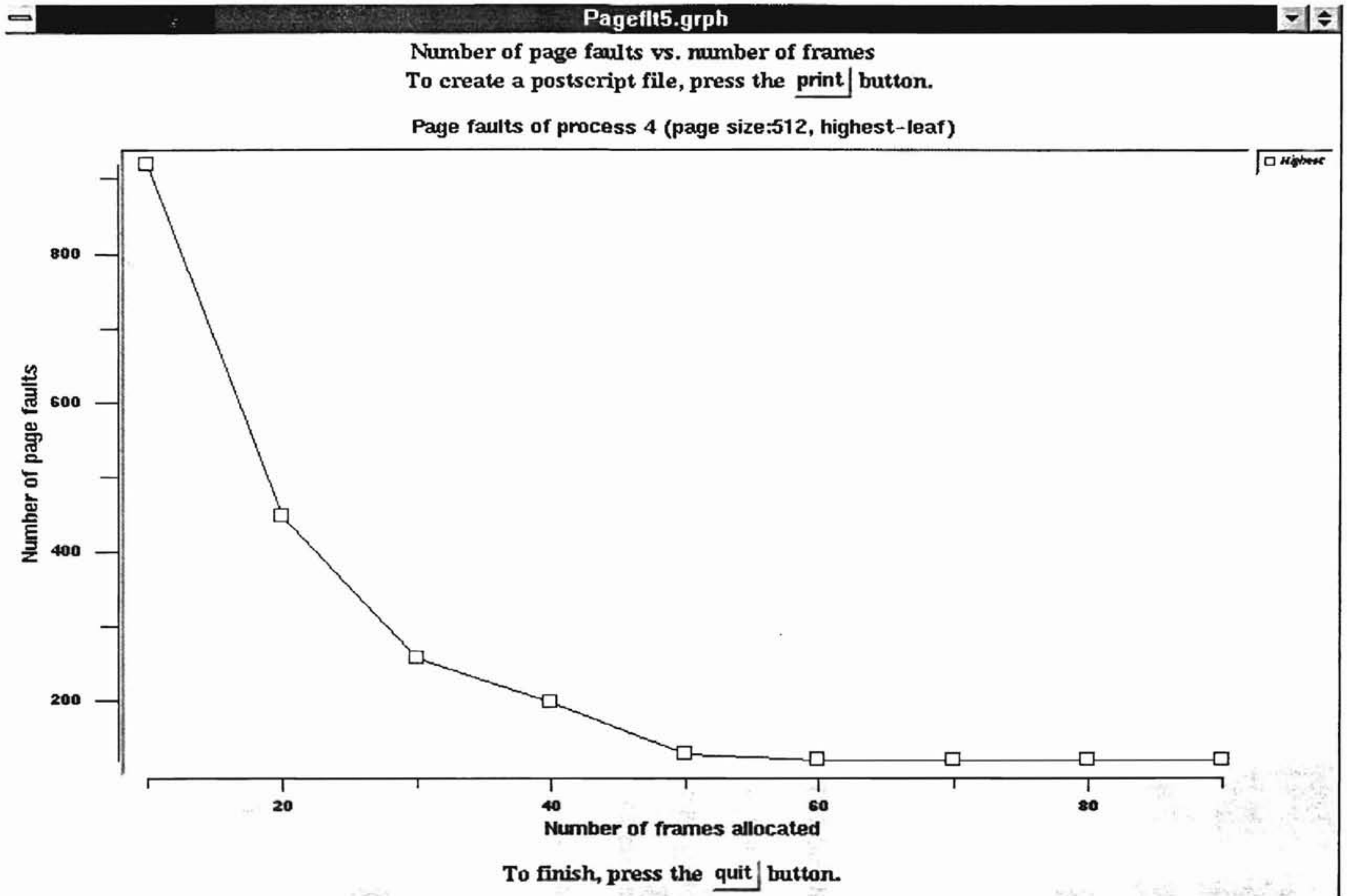

Figure 12. The number of page faults generated vs. the allocated memory size

CHAPTER IV

EVALUATION

4.1 Testing

4.1.1 Test Traces

The five traces used as input to the simulation were traces of SPEC92 benchmarks running on a MIPS R3000 simulator. These traces are available in the directory `/pub/tracebase4/r3000/din/` of the `ftp` site `tracebase@nmsu.edu`. The `din` directory contains traces in dinero+ format. The five sampled traces were selected randomly from among the twenty traces that were in that directory (in compressed format). After converting the files which consist of virtual pages, four sampled traces except "072.sc.din" were truncated to the length of 1,000,000 references. Table I shows the names of five sampled files and their lengths.

TABLE I. FIVE SAMPLED TRACES USED FOR THE SIMULATION

Process ID	File name	Length of reference string
1	039.wave5.din	1,000,000
2	056.ear.din	1,000,000
3	072.sc.din	999,996
4	078.swm256.din	1,000,000
5	093.nasa7.din	1,000,000

4.1.2 Memory Size

Table II shows the number of unique pages (minimum number of page faults) of each process for five different page sizes.

TABLE II. MINIMUM NUMBER OF PAGE FAULTS FOR DIFFERENT PAGE SIZES

Process ID	Page Sizes				
	512	1024	2048	4096	8192
1	163	98	62	38	25
2	297	162	88	50	27
3	221	137	82	50	33
4	121	74	47	32	19
5	432	228	124	71	39

However these numbers were not the "start points" of the frame numbers (i.e., the number of allocated frames) to get the minimum number of page faults (see Subsection 3.3.3 for an explanation of the start point). The start point was considered for each process as having an adequate memory size. The start points were different for different algorithms and different methods. We can find a start point through graphs similar to Figure 12. This graph shows the number of page faults according to different memory sizes for process 4 with 512 bytes as a page. The highest leaf method of the new implementation algorithm was used to generate the graph. We can observe that the number of page faults is stable at 121 after the number of frames allocated reaches 60. This number became a start point and did not agree with the number of unique pages (i.e., 121) for input trace or process 4.

Table III shows the start point of each process with 512 bytes as a page. Let us consider process 4 in Table III. The start point of the highest leaf and the LRU leaf methods in the new implementation was 60, and the start points of the additional-reference-bits algorithm for different intervals were 60 and 65. These working set sizes correspond to about half the number of unique pages that process or trace 4 has. On the other hand, the start points of the leftmost leaf and the rightmost leaf methods in new implementation were 120 and 110, which were almost similar to the number of unique pages in trace 4. This means that the start points over the sampled traces of each algorithm is different for each page replacement algorithm.

TABLE III. THE START POINTS OF MEMORY SIZES YIELDING MINIMUM PAGE FAULT NUMBERS IN EACH ALGORITHM

Process ID (# of unique pages)	New implementation			
	Leftmost	Rightmost	Highest	LRU leaf
1 (163)	160	163	160	140
2 (297)	297	297	290	280
3 (221)	220	220	220	210
4 (121)	120	110	60	60
5 (432)	430	380	130	130

Process ID (# of unique pages)	Clock interval			Additional-reference-bits interval		
	16800	28000	39200	70000	140000	210000
1 (163)	150	135	135	155	155	155
2 (297)	290	290	290	290	290	290
3 (221)	220	220	220	210	190	200
4 (121)	110	110	110	60	60	65
5 (432)	120	120	125	150	150	150

The two memory sizes, which were calculated by the two methods described in Subsection 3.3.3 over the start points of ten different methods in Table III, were 417,280 bytes (i.e., 163 frames * 5 processes * 512 bytes for a page) from method 1 and 455,680 bytes (i.e., 178 * 5 * 512) from method 2. The page size for the simulation was fixed as 512 bytes. The three Tables below (i.e., IV, V, and VI) show the number of page faults according to different memory sizes in the three algorithms. The tables for the other methods of each algorithm appear in Appendix C.

TABLE IV. THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZE (IN THE HIGHEST LEAF METHOD IN THE NEW IMPLEMENTATION)

Process	Number of frames allocated to each process						
ID	40	50	60	120	130	150	160
1	322	267	238	174	170	165	**163**
2	586	372	328	302	301	-	-
3	1661	1347	1076	393	342	278	261
4	198	129	**121**	-	-	-	-
5	578	539	499	433	432	-	-
	170	210	220	230	280	290	300
1	-	-	-	-	-	-	-
2	-	-	-	-	-	**297**	-
3	253	222	**221**	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

TABLE V. THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZE (IN THE CLOCK ALGORITHM WITH INTERVAL 28,000)

Process	Number of frames allocated to each process						
ID	105	110	115	120	130	135	140
1	173	-	174	173	164	**163**	-
2	325	324	325	324	-	-	323
3	386	327	338	327	317	303	313
4	122	**121**	-	-	-	-	-
5	434	-	-	**432**	-	-	-
	210	215	220	230	285	290	300
1	-	-	-	-	-	-	-
2	314	313	-	311	300	**297**	-
3	222	223	**221**	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

TABLE VI. THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZE (IN THE ADDITIONAL-REFERENCE-BITS ALGORITHM WITH INTERVAL 140,000)

Process	Number of frames allocated to each process						
ID	55	60	130	135	140	145	150
1	289	281	178	-	169	164	-
2	388	384	318	-	317	-	318
3	1295	1170	259	258	263	261	248
4	123	**121**	-	-	-	-	-
5	547	511	433	-	-	-	**432**
	155	185	190	195	200	285	290
1	**163**	-	-	-	-	-	-
2	316	313	-	-	312	298	**297**
3	247	222	**221**	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

4.1.3 Time interval

For the simulation of the clock and additional-reference-bits algorithms, the best time interval range which produces the minimum page fault number was chosen by running a number of experiments for each algorithm. The two memory sizes, which were obtained from the two methods described in Subsection 3.3.3 over the start points of the four different methods for the new implementation in Table III, were used to decide the best time interval ranges for two algorithms. These were 473,600 bytes (i.e., 185 frames * 5 processes * 512 bytes for a page) from method 1, and 504,320 bytes (i.e., 197 * 5 * 512) from method 2. The best range for the time interval for each algorithm was different based on the memory allocated. However, only one best range per algorithm was selected for the simulation because there was no big gap in the two resulting memory sizes. Two distinct ranges per each algorithm according to two memory sizes did not produced.

In the following discussion, process 3 is considered when the best range of time intervals in the two algorithms were fixed. The reason being that the two given frame numbers (i.e., 185 and 197) allocated to each process, were too large to enable one to observe the changing of the page fault numbers in the case of other processes. Figure 13 depicts the change of page fault numbers according to different time intervals using the clock algorithm with 473,600 bytes as the memory size. Figure 14 is the enlarged graph of Figure 13 showing the interval from 10 to 100,000 for finding the best range for the time intervals. The number of page faults were worst at time intervals 10 and after 1,500,000 as shown in Figure 13. Through Figure 14, the range from 28,000 to 42,000 was roughly found as the best interval range. Time interval 28,000 was taken as the best interval for the clock algorithm in the simulation.

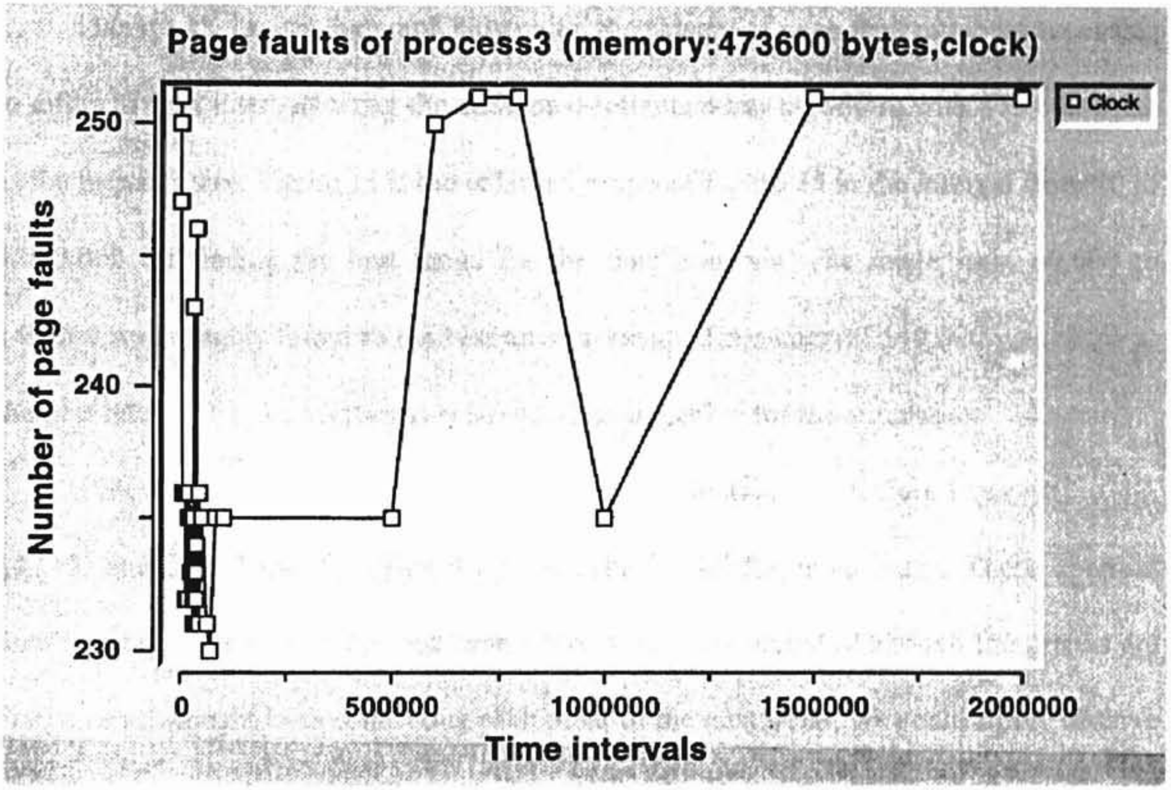

Figure 13. The number of page faults generated as affected by the change of regular time intervals in the clock algorithm

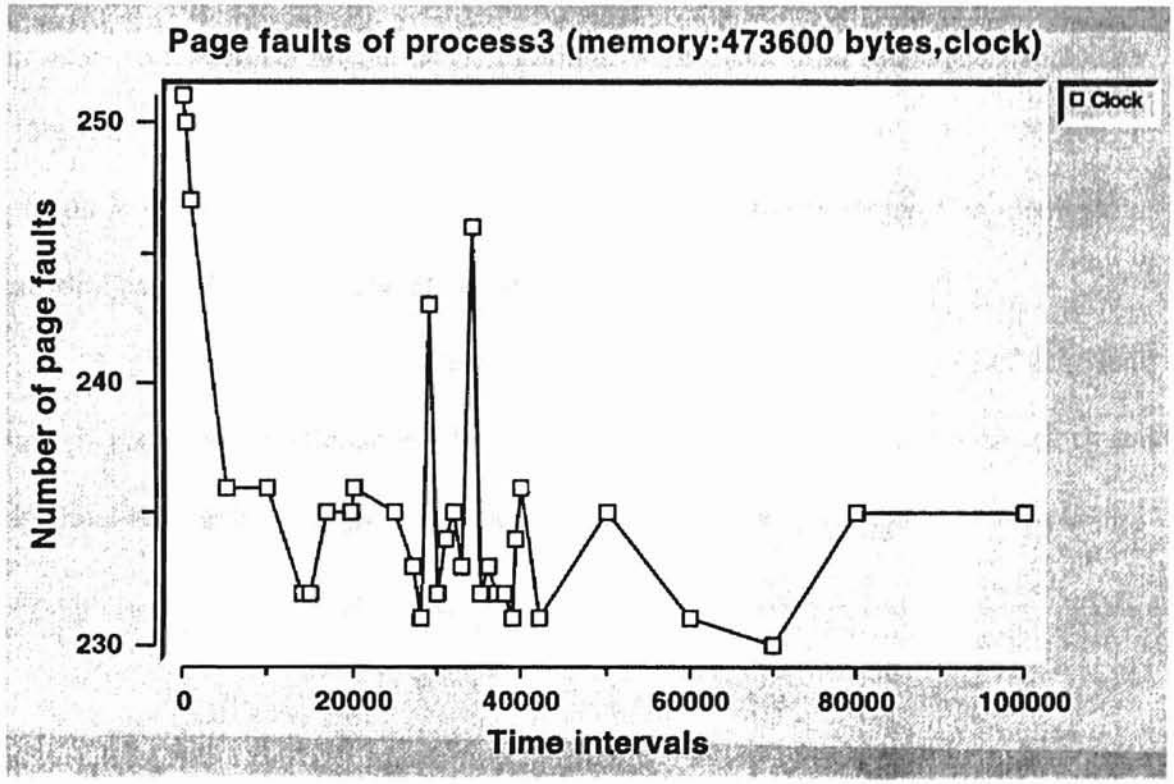

Figure 14. Expansion of Figure 13 from 10 to 100,000

Figure 15 depicts the graph illustrating the change of page fault numbers according to different time intervals using the additional-reference-bits algorithm with 473,600 bytes as the memory size. Figure 16 is the enlarged graph of Figure 15 in the interval from 10 to 1,000,000 for finding the best range for the time intervals. The range from 60,000 to 140,000 was roughly found as the best interval range. Time interval 140,000 was taken as the best interval for the additional-reference-bits algorithm for the simulation.

We can see the points where there are abrupt fluctuations in these Figures (i.e., 13, 14, 15, and 16). These are caused by the behavior of the input trace. These aberrant points were ignored when the best time interval was considered. Although the graphs are drawn using straight lines connecting each point to the next point, we could surely observe vibrations in a single straight line. The best interval point in the best range was decided after the range was taken. After taking the interval of 28,000 clocks for the clock algorithm and the interval of 140,000 clocks for the additional-reference-bits algorithm, the other two intervals for the clock algorithm were taken from -40% (i.e., 16,800) to +40% (i.e., 39,200) of 28,000, and from -50% (i.e., 70,000) to +50% (i.e., 210,000) of 140,000 in the additional-reference-bits algorithm. The percentages have no empirical or statistical basis, they can be considered arbitrary.

Appendix C contains the experimental results showing the difference of page fault numbers according to different intervals with the two given memory sizes in the clock and additional-reference-bits algorithms. Figures 13 through 16 are based on Appendix C.

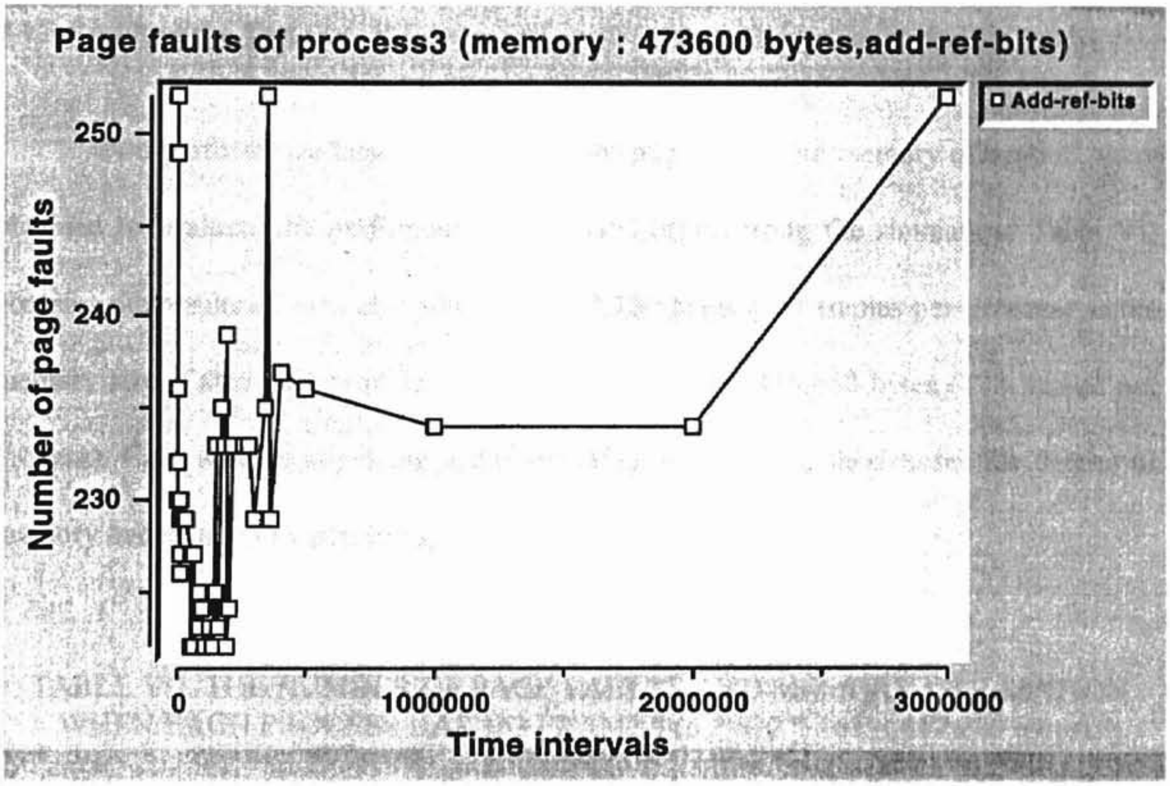

Figure 15. The number of page faults generated as affected by the change of regular time intervals in the additional-reference-bits algorithm

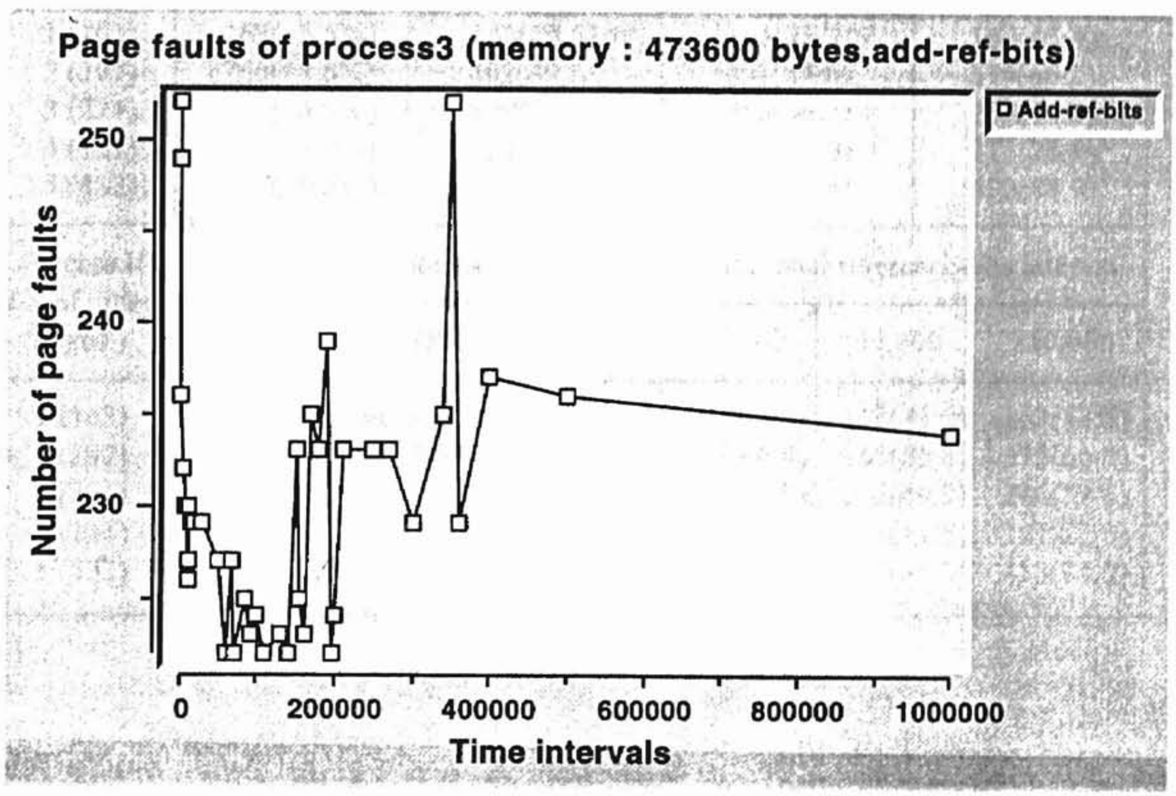

Figure 16. Expansion of Figure 15 from 10 to 1,000,000

4.1.3 Result of the Test

Two performance factors, the number of page faults and memory utilization, were obtained to evaluate the performance of the algorithms using the simulation. Table VII contains the results of each algorithm with 417,280 bytes (163 frames per process) as the memory size. Table VIII provides analogous results with 455,680 bytes (178 frames per process). Each number appearing inside parentheses in each table denotes the degree of memory occupancy as a percentage.

TABLE VII. THE NUMBER OF PAGE FAULTS AND MEMORY UTILIZATION WHEN EACH PROCESS HAS 163 FRAMES (5 * 512 * 163 = 417,280 bytes)

Process ID (# of unique pages)	New implementation			
	Leftmost	Rightmost	Highest	LRU leaf
1 (163)	163(3.5%)	163(28.4)	163(43.3)	163(43.4)
2 (297)	8760(98.6%)	402(49.8)	301(64.6)	314(66.2)
3 (221)	305(4.9%)	820(88.2)	258(54.2)	259(54.1)
4 (121)	121(1.8%)	121(14.5)	121(22.0)	121(21.9)
5 (432)	697(9.8%)	649(74.5)	432(84.9)	432(84.9)

Process ID (# of unique pages)	Clock interval			Additional-reference-bits interval		
	16,800	28,000	39,200	70,000	140,000	210,000
1 (163)	163(43.1)	163(43.3)	163(43.6)	163(43.2)	163(43.0)	163(43.3)
2 (297)	317(66.4)	317(66.8)	317(67.3)	315(66.4)	315(66.3)	315(66.5)
3 (221)	287(57.5)	251(52.9)	275(56.7)	237(50.8)	233(49.9)	246(52.6)
4 (121)	121(21.8)	121(21.9)	121(22.1)	121(21.9)	121(21.8)	121(21.8)
5 (432)	432(85.0)	432(84.9)	432(84.8)	432(84.9)	432(85.0)	433(84.9)

TABLE VIII. THE NUMBER OF PAGE FAULTS AND MEMORY UTILIZATION WHEN EACH PROCESS HAS 178 FRAMES (5 * 512 * 178 = 455,680 bytes)

Process ID (# of unique pages)	New implementation			
	Leftmost	Rightmost	Highest	LRU leaf
1 (163)	163(3.2)	163(3.4)	163(39.8)	163(39.7)
2 (297)	8738(98.4)	8262(98.3)	301(62.7)	309(63.0)
3 (221)	275(4.4)	587(8.4)	247(49.4)	244(48.6)
4 (121)	121(1.6)	121(1.7)	121(20.1)	121(20.1)
5 (432)	682(9.5)	634(9.3)	432(82.9)	432(82.9)

Process ID (# of unique pages)	Clock interval			Additional-reference-bits interval		
	16,800	28,000	39,200	70,000	140,000	210,000
1 (163)	163(39.6)	163(39.7)	163(39.5)	163(39.7)	163(39.6)	163(39.9)
2 (297)	316(64.2)	316(64.2)	316(64.3)	313(64.6)	313(63.8)	315(64.7)
3 (221)	239(48.0)	236(47.6)	234(47.4)	224(46.0)	228(46.6)	244(48.9)
4 (121)	121(20.0)	121(20.1)	121(20.0)	121(20.1)	121(20.0)	121(20.2)
5 (432)	432(83.0)	432(82.9)	432(83.0)	432(82.9)	432(83.0)	433(82.8)

4.2 Analysis

4.2.1 Graphs

The graphs were plotted using BLT which is an extension of Tk [Ousterhout 94]. There are two kinds of graphs. One is for depicting page fault numbers and the other is for displaying memory utilization. The graphs showing the page fault numbers have page fault numbers on the y_axis vs. process ID on the x_axis. On the other hand, the graphs representing memory utilization take the percentage of memory occupancy on the y_axis vs. process ID on the x_axis. Each graph consists of four different types of plots according to the subject of discussion. Firstly, the graph comparing each algorithm with the other algorithms was represented (Figures 17 and 18). Secondly, the graph comparing one implementation method with the other methods within the new implementation using splay tree was given (Figures 19 and 20). Thirdly, the graph comparing the clock algorithm with itself using different time intervals was obtained (Figures 21 and 22). Finally, the graph comparing the additional-reference-bits algorithm with itself using other time intervals was presented (Figures 23 and 24).

4.2.2 Observations

From the below graphs presented in this subsection, the page fault numbers and memory utilization of different algorithms or, as the case might be, each method of a given algorithm can be obtained. These graphs are based on a memory size of 417,280 bytes calculated using method 1 as described in Subsection 3.3.3, a page size of 512 bytes, and 10,000 clocks as page fault handling time. The algorithm (or the method) that has a lower page fault rate and a higher memory occupancy than the others, is favored and recommended for improving system performance.

Figure 17 represents the page fault numbers of each process when using three different page replacement algorithms. The highest leaf method in the new implementation, interval 28,000 in the clock algorithm, and interval 140,000 in the additional-reference-bits algorithm were considered for the graph. Figure 18 gives the memory utilization of each process when using the same methods and algorithms which were used for Figure 17.

In Figure 17, the number of page faults of process 2 is 301 when using the highest leaf method in the new implementation. This value is less than 317 and 315, which are generated by the clock and additional-reference-bits algorithms. The number of page faults for process 3 is 258 which is more than 251 and 233 as generated using the other two algorithms. The other three processes had the same values for the number of page faults generated when for all three algorithms. These were the result of the fact that the allocated memory to each process was big enough to execute the program (see Table III).

Figure 18 shows that the memory occupancy of each process is almost the same regardless of the algorithm used. Process 4 had the lowest occupancy, one reason being that the start point of memory after which process 4 has minimum page fault numbers is less than the other processes.

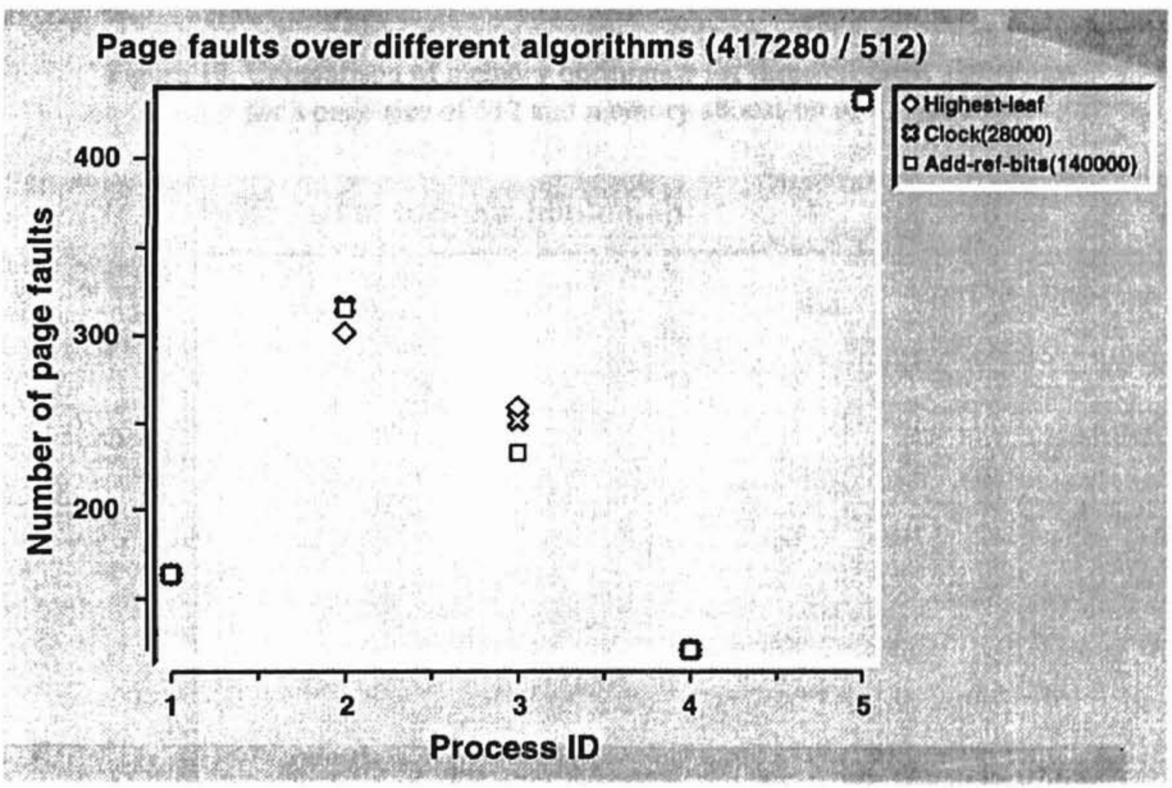

Figure 17. Comparison of page fault numbers for three different algorithms for a page size of 512 and memory allocation of 417,280 bytes

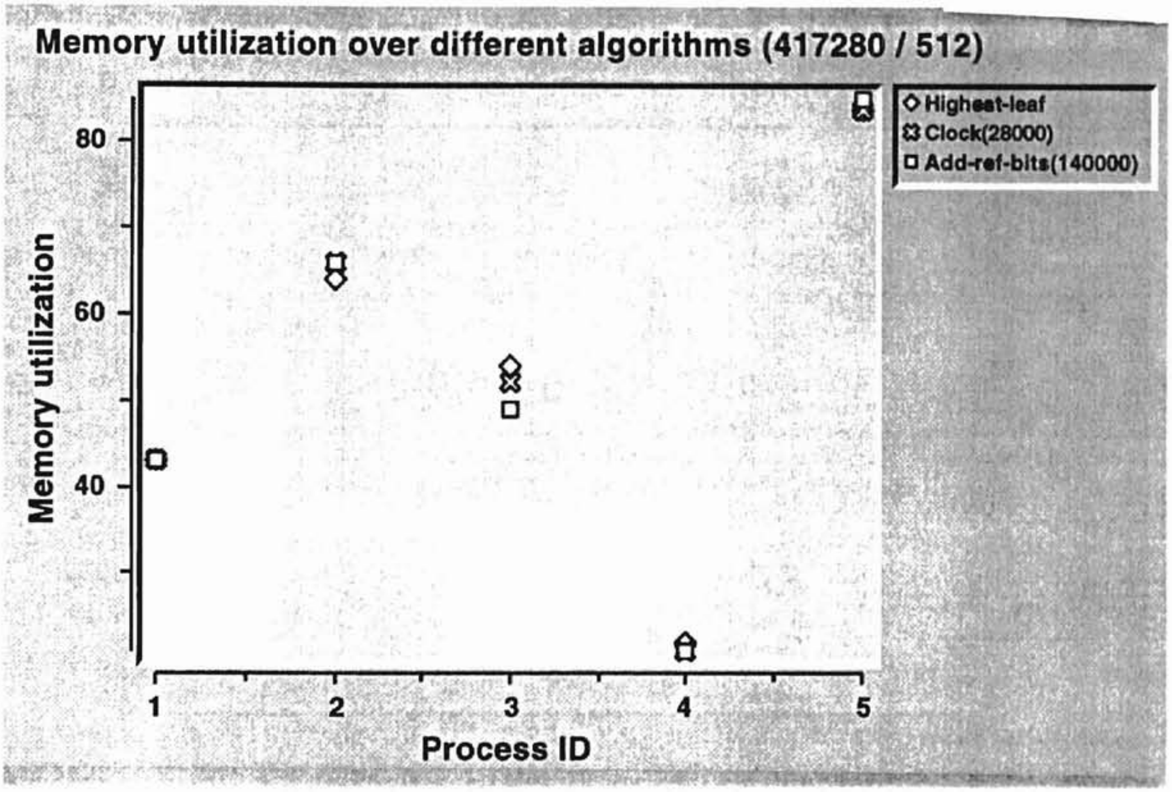

Figure 18. Comparison of memory occupancy for three different algorithms for a page size of 512 and memory allocation of 417,280 bytes

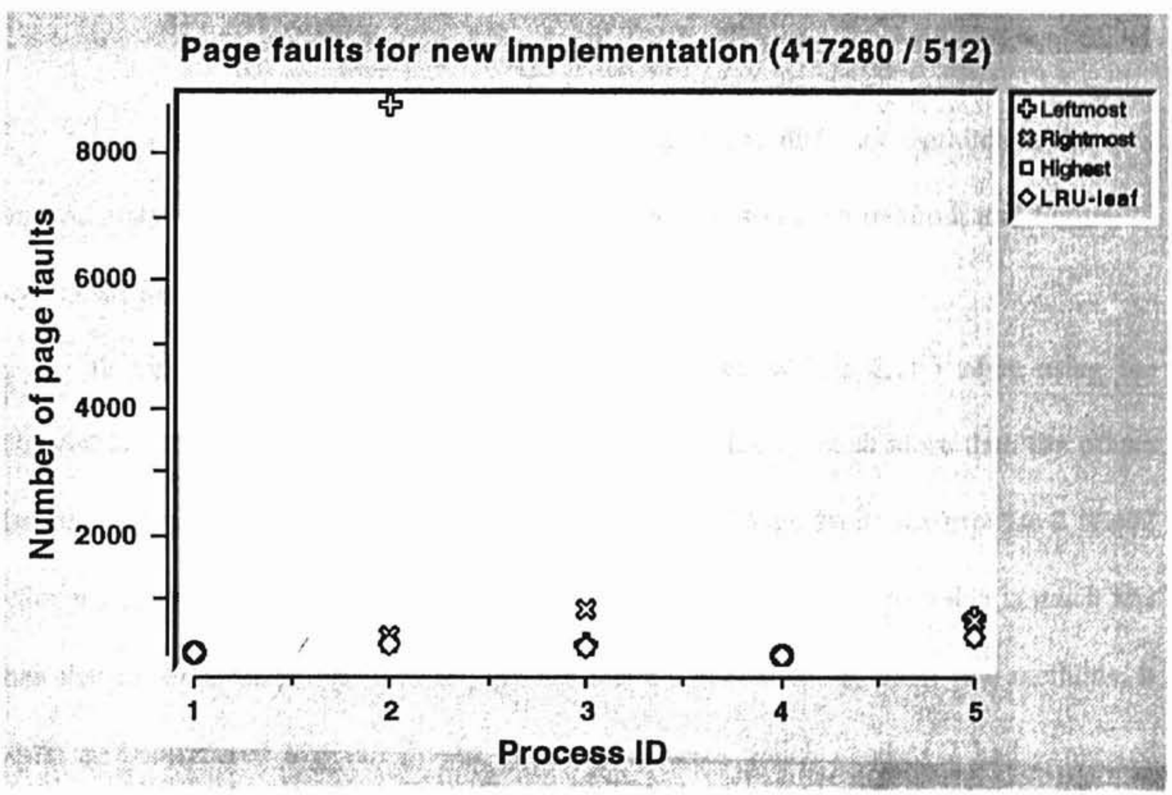

Figure 19. Comparison of page fault numbers in the four different methods used in the new implementation for a page size of 512 and for memory allocation of 417,280 bytes

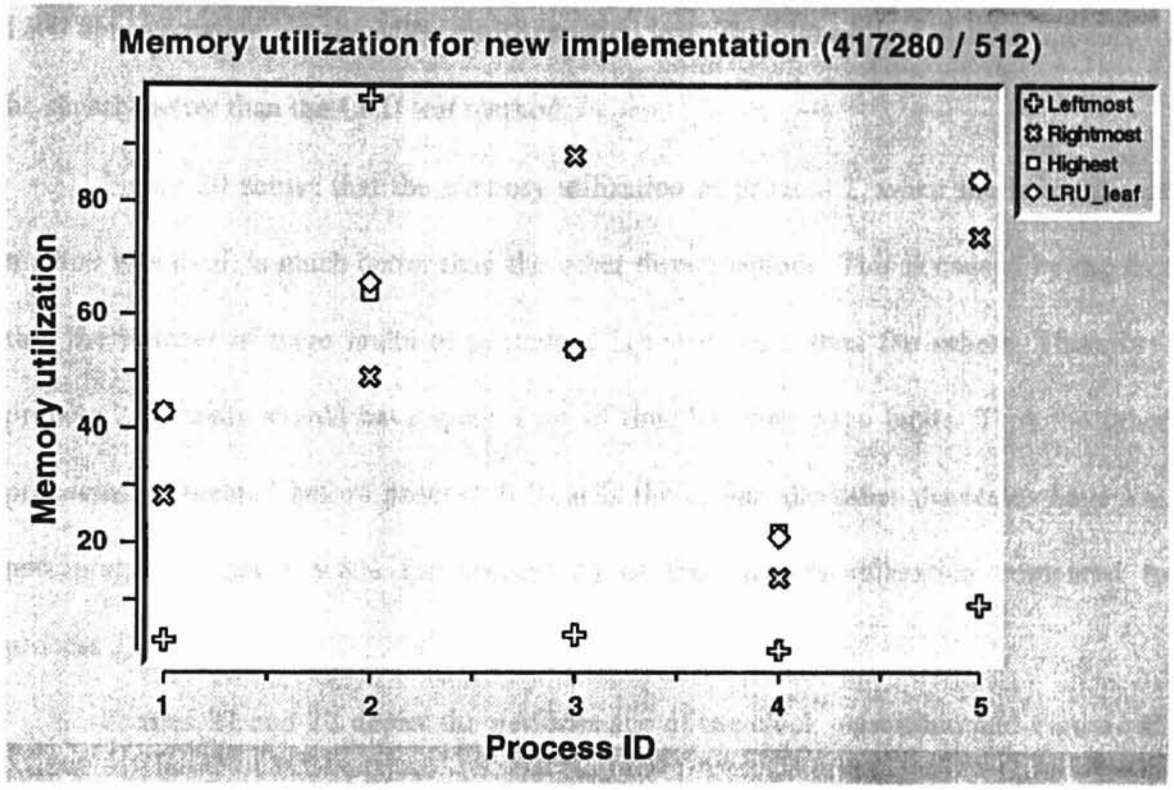

Figure 20. Comparison of memory occupancy in the four different methods used in the new implementation for a page size of 512 and for memory allocation of 417,280 bytes

Figures 19 and 20 contain graphs that compare the different methods in the new implementation. Figure 19 gives the number of page faults in each method, and Figure 20 depicts the memory utilization.

In Figure 19, the number of page faults for process 2 is 8,760 when using the leftmost leaf method in the new implementation. This value is much more than the others that are generated by the other methods. The number of page faults for process 2 is 402 when using the rightmost leaf method in the new implementation. This value is much less than that of the leftmost leaf method, but not much higher than the other two methods. It could be conjectured that the shapes of the splay trees which process 2 had generated were mostly right heavy. Therefore, as expected the leftmost leaf method became not the

LRU approximation but the MRU approximation leaf. The highest leaf method appears to be slightly better than the LRU leaf method.

Figure 20 shows that the memory utilization of process 2, when the leftmost leaf method was used, is much better than the other three methods. This is caused by the fact that the number of page faults of process 2 is much more than the others. Therefore, process 2 certainly should have spent a lot of time handling page faults. Thus the other processes terminated before process 2. That is the reason the other processes have less percentages (at most 9.8% for process 5) of the memory utilization compared to process 2.

Figures 21 and 22 depict the performance of the clock algorithm, and Figures 23 and 24 depict the performance of the additional-reference-bits algorithm. There are three regular time intervals (i.e., 16,800, 28,000, and 39,200) in the clock algorithm, and three (i.e., 70,000, 140,000, and 210,000) in the additional-reference-bits algorithm. The best time interval among these three can be taken for comparison against the new implementation using splay tree.

There were no major differences in the page fault rate or the memory utilization when each interval was used in either the clock or the additional-reference-bits algorithm. This is caused by the fact that the two randomly selected intervals (i.e., +40%, -40% of the best interval for clock and +50%, -50% for additional-reference-bits) in each algorithm roughly belonged to the best time interval range.

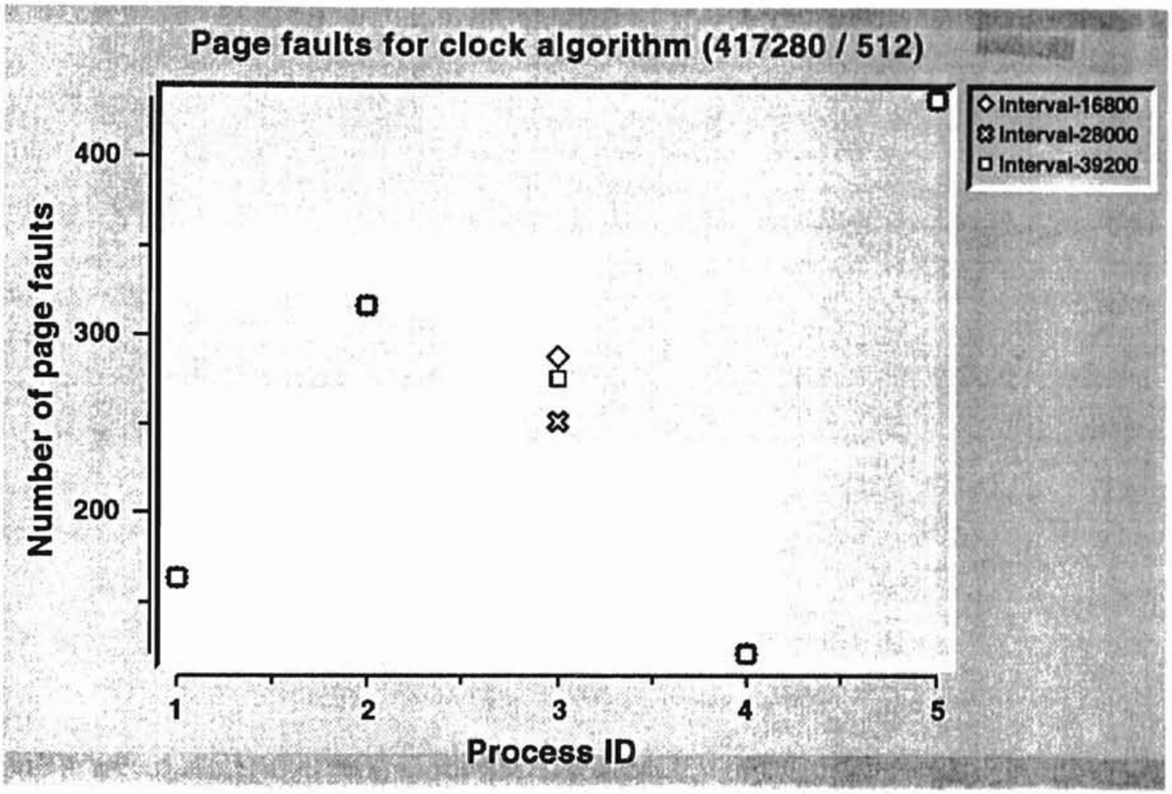

Figure 21. Comparison of page fault numbers for three different intervals used in the clock algorithm with a page size of 512 and memory allocation of 417,280 bytes

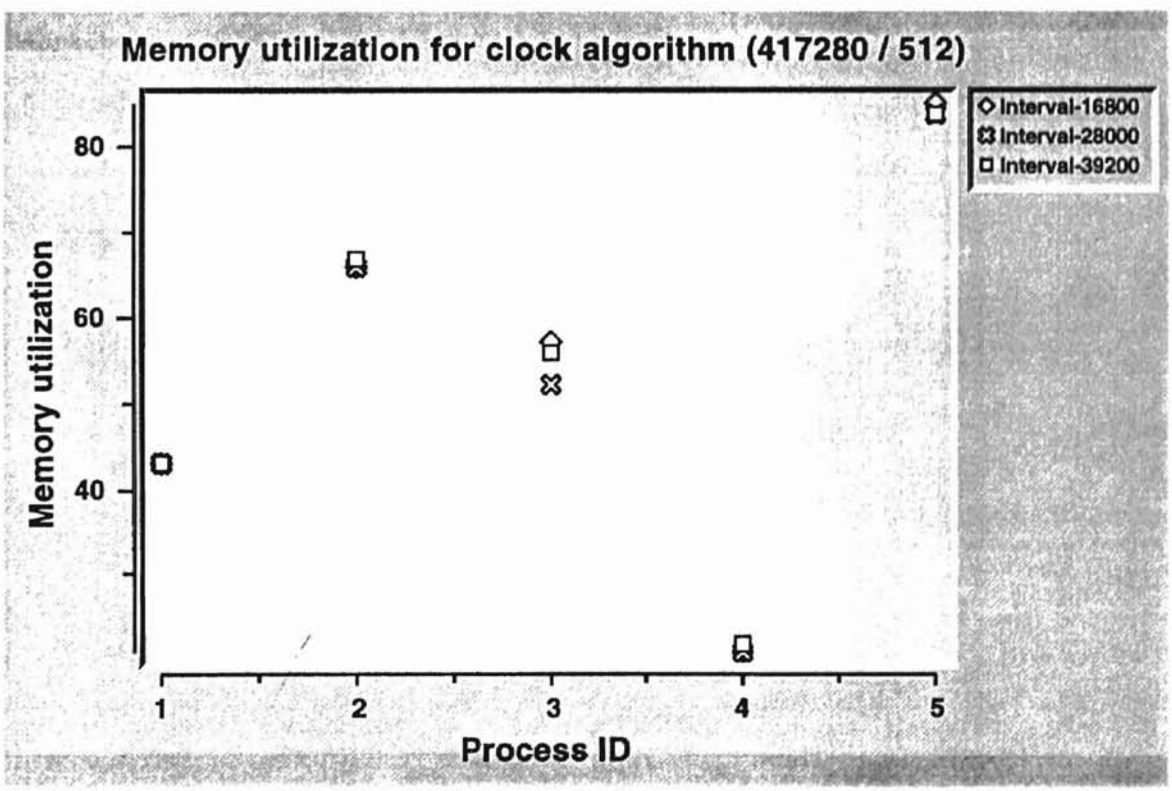

Figure 22. Comparison of memory occupancy for three different intervals used in the clock algorithm with a page size of 512 and memory allocation of 417,280 bytes

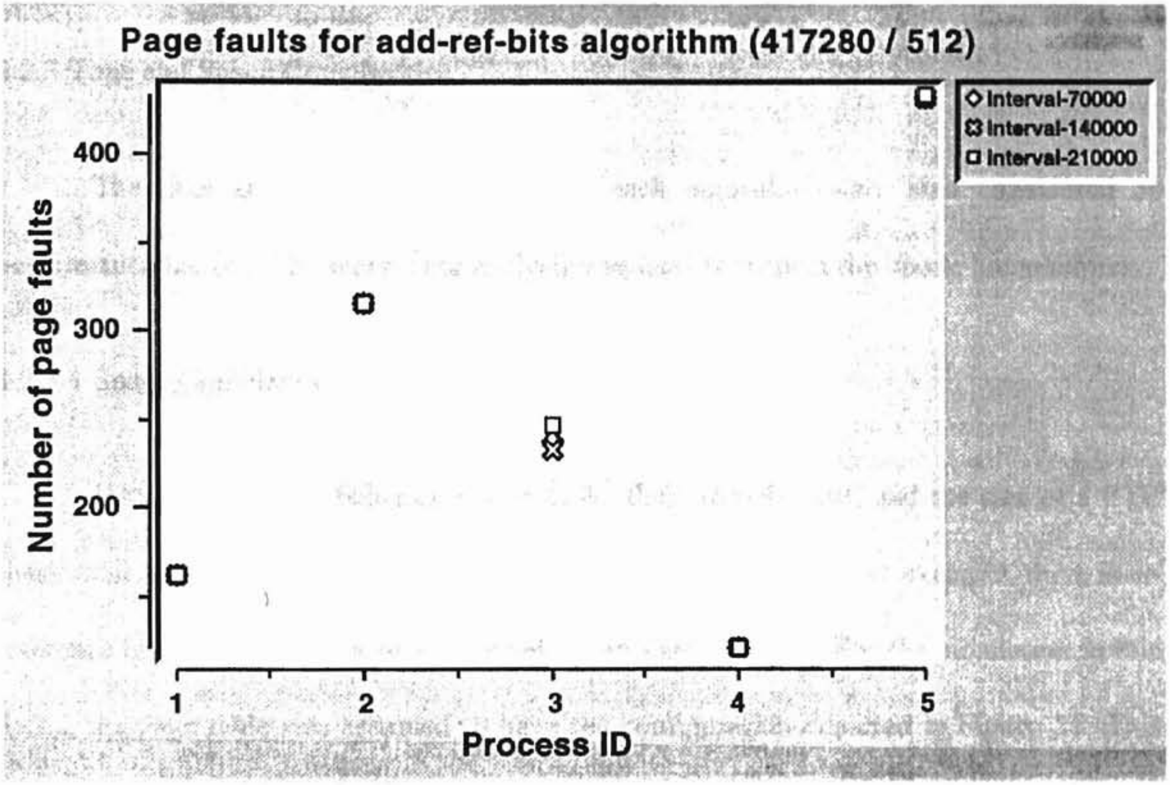

Figure 23. Comparison of page fault numbers for three different intervals used in the additional-reference-bits algorithm for a page size of 512 and memory allocation of 417,280 bytes

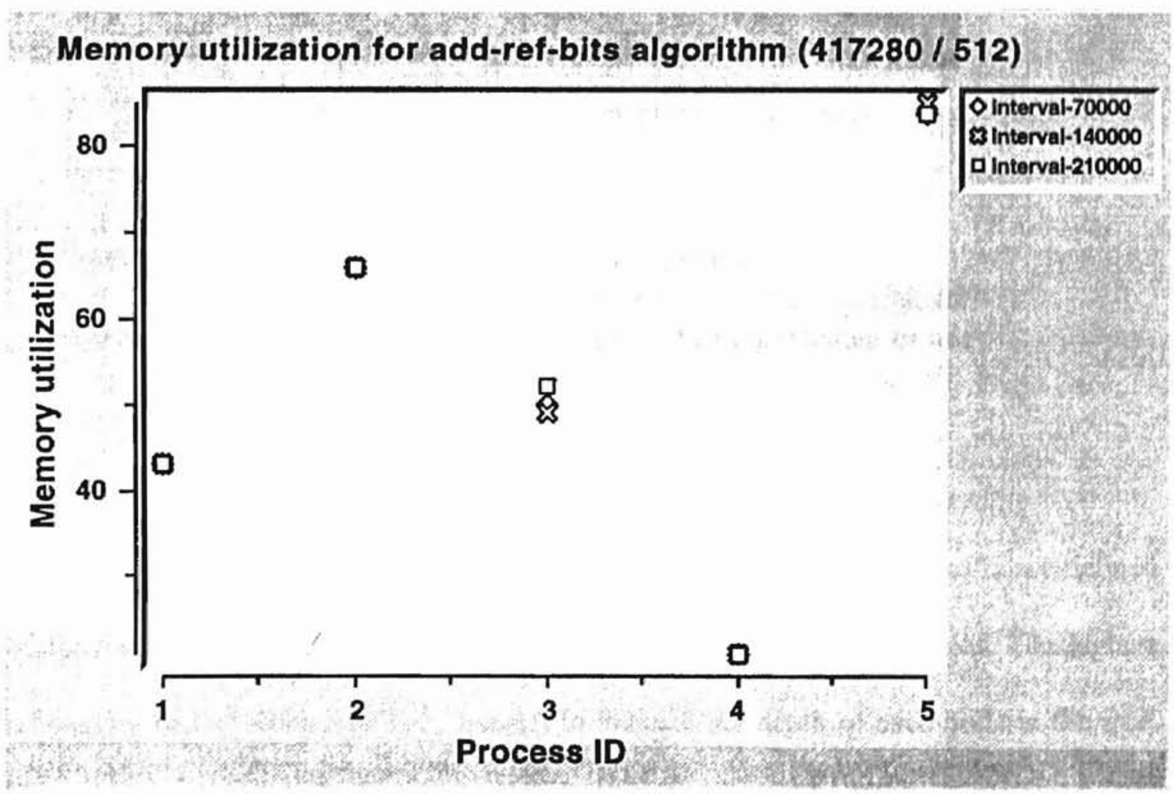

Figure 24. Comparison of memory occupancy for three different intervals used in the additional-reference-bits algorithm for a page size of 512 and memory allocation of 417,280 bytes

4.2.3 Time and Space Complexities

The time and space complexities of each algorithm were also considered as performance factors. The worst case analysis was used to inspect the above complexities.

4.2.3.1 Space Complexity

Usually, the exact fields of a page table, their arrangement, and the size of a PTE (page table entry) are highly machine dependent [Tanenbaum 92]. For example, there is no reference bit in the VAX machine [Hennessy and Patterson 90]. For the simulation in this thesis, the page table was assumed to have the configuration depicted in Figure 25. This figure gives a typical PTE. Each entry of a page table is basically an architecture-defined field except for the page frame number [Hennessy and Patterson 90].

	. . .	X	M	P	V	page frame number

X : the reference bit
M : the modify bit indicating whether or not the page is dirty
P : the protection bit(s) indicating what kinds of access is permitted
V : the valid bit (or the present/absent bit) indicating whether or not the PTE has a valid address

Figure 25. A typical page table entry

To implement the new algorithm that uses splay trees, three software-defined pointer fields (i.e., right, left, and parent) were added as page table entries. The highest leaf method had an extra field (i.e., height) to indicate the depth of each node in the tree. Except the LRU leaf method, the other three methods did not require any extra software table. To implement the LRU leaf method, one software table implemented as a linked list was used. This linked list contained the leaves in the tree. In the worst case, there are $\lceil n/2 \rceil$ (n being the number of nodes in the tree) leaves in a binary tree [Weiss 92]. Each node of the linked list had three fields (Figure 6). So the extra space complexity of the LRU leaf method is $O(3\lceil n/2 \rceil)$ which equals O(n).

To implement the clock algorithm, one extra pointer field, which indicates the next entry, was used because a circular queue was implemented using a linked list. So the space complexity of the clock algrithm is O(n).

An 8-bit shift register was added as an entry of the page table to implement the additional-reference-bits algorithm for the simulation. The shift register is provided by the hardware. This algorithm also used a linked list. Therefore an extra pointer field to indicate next node was used. Table IX shows the space complexity of each algorithm.

TABLE IX. SPACE COMPLEXITY OF EACH ALGORITHM IN THE WORST CASE

Left(right)most leaf	Highest leaf	LRU leaf	Clock	Additional-reference-bits
O(n)	O(n)	O(n)	O(n)	O(n)

(n: the number of frames allocated ≈ the number of nodes in the tree)

There is no significant difference in the space complexity of the algorithms because the space complexity of each method in the new implementation equals O(n) by the property of the big-oh notation.

4.2.3.2 Time Complexity

Time complexity is mainly considered when searching for a page, choose a victim page, and rebuilding the page table for the simulation.

4.2.3.2.1 Searching

The page to be referenced must be searched for regardless of whether it is main memory or not. In the case of the new implementation, the single search operation needs O(n) time in the worst case. The reason being that the time complexity of the search operation for the binary search tree is O(n) in the worst case. Although the O(log n) bound on any single operation cannot be guaranteed in the splay tree, the operations of splay tree have O(log n) amortized time. It is more reasonable that the amortized time be considered when long sequences of operations are processed, as is the case in this trace-driven simulation.

In the clock and additional-reference-bits algorithms, the search takes O(n) in the worst case because these algorithms are implemented using linked lists.

4.2.3.2.2 Selecting a victim page

Tree traversal must be done to get the leaves in the tree and to get the height of each leaf when selecting a victim page in the highest leaf and LRU leaf methods. Therefore, it takes O(n) to selecting a victim page in the above two methods. In the case of the LRU leaf method, the leaf queue must be checked to see whether each leaf node in the current state was a leaf in the previous state during the traversal. The size of the leaf queue is at most $\lceil n/2 \rceil$ (n being the number of nodes in the tree). Therefore $O(n\lceil n/2 \rceil)$ equals $O(n^2)$ taken as the complexity of the LRU leaf method. In the leftmost (rightmost) leaf method, O(log n) must be taken because only finding the leftmost (rightmost) leaf is needed.

In the clock algorithm, O(n) is taken in the worst case because all nodes must be traversed when the FIFO emulation occurs.

In the additional-reference-bits algorithm, all nodes in the linked list must be traversed to get the page which has the smallest value of shift register. So it also takes O(n) time in the worst case.

4.2.3.2.3 Rebuilding

In the new implementation, whenever a page is referenced, the page table is rebuilt using splaying. The time complexity of splaying is O(log n) in amortized bound [Sleator and Tarjan 85]. In the clock algorithm, the reference bits are cleared after each regular interval. When the time interval is 1 in the worst case, O(n) time is taken whenever a page is referenced. In the additional-reference-bits algorithm, one bit is shifted right at each time interval. When the time interval is 1 in the worst case, O(n) time is taken whenever a page is referenced.

TABLE X. TIME COMPLEXITY OF EACH ALGORITHM

	Left(right)most	Highest	LRU leaf	Clock	Add-ref-bits
Search	O(log n)	O(log n)	O(log n)	O(n)	O(n)
Choose	O(log n)	O(n)	$O(n^2)$	O(n)	O(n)
Restruct	O(log n)	O(log n)	O(log n)	O(n)	O(n)

(n: the number of frames allocated ≈ the number of nodes in the tree)

Table X shows that the time complexity of the new implementation except the LRU leaf method is better than the other two traditional algorithms when the amortized bound was considered in the new implementation.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

In Chapter I, the significance of memory management, virtual memory, splay tree, trace-driven simulation, and the main objective of the thesis were stated. Chapter II contained a review of the virtual memory management schemes and splay tree operations. The topics covered in this chapter were paging, page replacement algorithms, splay tree, and performance evaluation factors. Chapter III presented the implementation platform and environment, and discussed the input parameters, the fundamental data structures used, and the implementation details to implement each algorithm. Chapter IV addressed the test programs (i.e., the test traces) used as input and the graphs obtained. This chapter also analyzed the results of the simulation using performance graphs as well as time and space complexities.

The main goal of the thesis was to develop a trace-driven simulation to apply a splay tree to implement a page replacement algorithm. To drive the simulation, five traces consisting of virtual addresses, obtained from New Mexico State University, were used as input. The new implementation was compared to two traditional LRU approximations (i.e., clock and additional-reference-bits). The evaluation factors for performance (i.e., page faults rate and memory utilization), were analyzed using graphs obtained from the results of the simulation. The time and space complexities of the algorithms were also compared. Four methods were used to select a victim page in the new implementation: the leftmost leaf, the rightmost leaf, the highest leaf, and the LRU leaf methods. The highest leaf method, which does not need any hardware support, had the most reasonable result over the performance factors considered. Therefore, the highest leaf method could be recommended as a page replacement algorithm.

5.2 Future Work

The simulation (implemented as part of this thesis) handles the case where the memory is equally divided among processes. Equal allocation would not be an applicable approach when processes need to allocate memory according to their dynamic behaviors. If the memory is divided among processes according to the estimated memory amount which each program needs, higher memory utilization and more tolerable page fault rate would be expected.

Parallel processes were not used in this simulation. Using two parallel processes for the new implementation (i.e., one for searching and the other for splaying) would be an attractive approach to decrease the execution time.

REFERENCES

[Aho et al. 74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1974.

[Aho et al. 87] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1987.

[Belady 66] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer", *IBM Systems Journal*, Vol. 5, No. 2, pp. 78-101, 1966.

[Belady et al. 69] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine", *Communications of the ACM*, Vol. 12, No. 6, pp. 349-353, June 1969.

[Belady et al. 81] L. A. Belady, R. P. Parmlee, and C. A. Scalzi, "The IBM History of Memory Management Technology", *IBM Journal of Research and Development*, Vol. 25, No. 5, pp. 491-503, September 1981.

[Carr 84] R. W. Carr, *Virtual Memory Management*, UMI Research Press, Ann Arbor, MI, 1984.

[Coffman and Denning 73] E. G. Coffman, Jr. and P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[Deitel 90] H. M. Deitel, *An Introduction to Operating Systems*, Second Edition, Addison-Wesley Publishing Company, Inc., Reading, MA, February 1990.

[Denning 70] P. J. Denning, "Virtual Memory", *ACM Computing Surveys*, Vol. 2, No. 3, pp. 153-189, September 1970.

[Dijkstra 76] E. W. Dijstra, *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[Hennessy and Patterson 90] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Lister and Eager 93] A. M. Listser and R. D. Eager, *Fundamentals of Operating Systems*, Fifth Edition, Springer-Verlag, Inc., London, UK, 1993.

[Nutt 92] G. J. Nutt, *Centralized and Distributed Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[Ousterhout 94] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Welsey Publishing Company, Reading, MA, 1994.

[Poursepanj 94] Ali Poursepanj, "The PowerPC Performance Modeling Methodology", *Communications of the ACM*, Vol. 37, No. 6, pp. 47-55, June 1994.

[Sequent 90] *Symmetry Multiprocessor Architecture Overview*, Sequent Computer Systems, Inc., 1990.

[Silberschatz and Galvin 94] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, Fourth Edition, Addison-Welsey Publishing Company, Reading, MA, 1994.

[Sleator and Tarjan 85] D. D. Sleator and R.E. Tarjan, "Self-Adjusting Binary Search Tree", *Journal of the ACM*, Vol. 32, No. 3, pp. 652-686, July 1985.

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

[Tanenbaum 92] A. S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[Tarjan 83] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Udi 89] M. Udi, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley Publishing Company, Reading, MA, 1989.

[Weiss 92] M. A. Weiss, *Data Structures and Algorithm Analysis*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1992.

APPENDICES

APPENDIX A:

GLOSSARY

Amortized Time: The average time of an operation over a worst-case sequence of operations.

ANSI: American National Standards Institute.

Belady's Anomaly: The phenomenon that more page faults occur when the number of frames allocated to a process is increased.

Demand Paging: A simple technique that transports only the pages that are referenced from secondary memory to main memory.

Dinero+: A common format used for capturing and representing traces defined at /pub/tracebase4/r3000/README of the ftp site tracebase@nmsu.edu as follows: "in addition to the usual type and address fields, a third field is present that lists the instruction word for instruction fetches".

Dirty: When the information of a page in the memory differs from that on the disk, the page is called dirty.

FIFO: First In First Out.

Hit Time: The time to access the upper level of the memory hierarchy.

LRU: Least Recently Used.

Miss Rate: The fraction of memory accesses not found in the memory. This is sometimes represented as a percentage.

MMU: Memory Management Unit.

Multiprogramming: The existence of several programs on the same machine at the same time. Several programs are held simultaneously in memory. While a program is waiting for I/O, another program can use the CPU.

Preorder tree traversal: One of the tree traversal strategies. It processes the current node first and then the left subtree followed by the right subtree sequentially.

Process: A program in execution. A sequence of actions performed by a program.

Reference Bit: A reference bit is associated with each entry in a page table. It is set by hardware whenever a page is referenced, either for reading or for writing. Its value is used in several page replacement algorithms. Referred to sometimes as the X bit.

Reference String: A sequence of pages which are referenced by a program. It presents a program's dynamic behavior. If $A = \{ x \mid x$ is a page number of a given program $\}$, then $s = x_1 x_2 \dots x_n$, where $x_i \in A$, $1 \leq i \leq n$, is a reference string.

Space Complexity: The space needed by an algorithm expressed as a function of the size of a problem. Often it expresses the limiting or asymptotic behavior of an algorithm.

Time Complexity: The time needed by an algorithm expressed as a function of the size of a problem. Often it expresses the limiting or asymptotic behavior of an algorithm.

Time Sharing: A variant of multiprogramming which implies support for multiple on-line terminals, one for each active user of the system.

X Bit: Reference bit (see above).

APPENDIX B:

TRADEMARK INFORMATION

DYNIX, DYNIX/ptx:	Registered trademarks of the Sequent Computer Systems, Inc.
Sequent, Symmetry:	Registered trademarks of the Sequent Computer Systems, Inc.
UNIX:	A registered trademark of AT&T.

APPENDIX C:

EXPERIMENTAL RESULTS

C-1 The tables for representing the change of page fault numbers according to different memory sizes in the new implementation (leftmost leaf, rightmost leaf, and LRU leaf methods), clock algorithm (intervals 16,800 and 39,200), and additional-reference-bits algorithm (intervals 70,000 and 210,000).

C-2 The tables for representing the change of page fault numbers according to different intervals in the clock and additional-reference-bits algorithms. Two memory sizes (i.e., 473,600 and 504,320) were used to find the best range of intervals.

C-1-1 THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZES (LEFTMOST LEAF METHOD IN NEW IMPLEMENTATION)

Process	Number of frames allocated to each process						
ID	110	120	130	150	160	170	210
1	312	248	189	168	**163**	-	-
2	8839	8824	8809	8779	8766	8749	8377
3	2461	1785	1145	634	352	282	240
4	123	**121**	-	-	-	-	-
5	738	729	728	710	700	690	650
	220	240	290	300	310	420	430
1	-	-	-	-	-	-	-
2	7320	4513	379	**297**	-	-	-
3	**221**	-	-	-	-	-	-
4	-	-	-	-	-	-	-
5	620	-	570	560	550	440	**432**

C-1-2 THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZES (RIGHTMOST LEAF METHOD IN NEW IMLEMENTATION)

Process	Number of frames allocated to each process						
ID	100	110	120	160	170	210	220
1	225	192	185	164	**163**	-	-
2	11886	9337	6789	12853	10302	349	1941
3	7971	6742	4033	1048	674	236	**221**
4	128	**121**	-	-	-	-	-
5	726	716	701	652	642	602	592
	230	290	300	320	370	380	400
1	-	-	-	-	-	-	-
2	330	299	**297**	-	-	-	-
3	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-
5	582	522	512	492	442	**432**	-

C-1-3 THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZES (LRU LEAF METHOD IN NEW IMPLEMENTATION)

Process	Number of frames allocated to each process						
ID	50	60	100	110	120	130	140
1	265	233	178	175	170	168	**163**
2	496	474	434	395	361	338	325
3	1424	1114	389	338	318	291	274
4	124	**121**	-	-	-	-	-
5	528	485	434	433	-	**432**	-
	150	200	210	250	270	280	290
1	-	-	-	-	-	-	-
2	320	303	-	300	-	**297**	-
3	266	223	**221**	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

C-1-4 THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZES (CLOCK ALGORITHM WITH INTERVALS 16,800 AND 39,200)

Process	Number of frames allocated to each process (with 16,800)						
ID	105	110	115	120	135	140	145
1	175	177	175	176	173	165	164
2	324	324	-	-	-	323	324
3	365	340	343	329	320	317	287
4	122	**121**	-	-	-	-	-
5	434	-	-	**432**	-	-	-
	150	185	205	210	220	285	290
1	**163**	-	-	-	-	-	-
2	-	316	314	-	313	298	**297**
3	275	235	225	-	**221**	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

Process	Number of frames allocated to each process (with 39,200)						
ID	105	110	120	125	130	135	140
1	174	173	166	-	164	**163**	-
2	321	324	-	-	-	-	323
3	379	351	327	318	310	272	299
4	122	**121**	-	-	-	-	-
5	435	436	434	**432**	-	-	-
	185	197	210	215	220	285	290
1	-	-	-	-	-	-	-
2	316	-	314	313	-	298	**297**
3	234	229	225	226	**221**	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

C-1-5 THE NUMBER OF PAGE FAULTS ACCORDING TO MEMORY SIZES (ADDITIONAL-REFERENCE-BITS ALGORITHM WITH INTERVALS 70,000 AND 210,000)

Process	Number of frames allocated to each process (with 70,000)						
ID	55	60	130	135	140	145	150
1	289	281	178	-	169	**164**	-
2	369	349	315	-	-	-	-
3	1402	1171	262	272	250	249	252
4	122	**121**	-	-	-	-	-
5	547	499	433	-	-	-	**432**
	155	160	185	205	210	285	290
1	**163**	-	-	-	-	-	-
2	315	-	313	312	-	298	**297**
3	255	240	222	-	**221**	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

Process	Number of frames allocated to each process (with 210,000)						
ID	60	65	140	145	150	155	160
1	281	263	169	165	164	**163**	-
2	369	358	315	-	-	-	-
3	1120	1248	249	247	-	254	247
4	122	**121**	-	-	-	-	-
5	509	489	437	433	**432**	-	-
	185	195	200	205	285	290	300
1	-	-	-	-	-	-	-
2	-	309	307	-	298	**297**	-
3	233	225	**221**	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

C-2-1-1 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS (ADDITIONAL-REFERENCE-BITS ALGORITHM WITH MEMORY ALLOCATION 473,600 Bytes)

Process	Intervals						
ID	10	100	500	1000	5000	8000	10000
1	163	-	-	-	-	-	-
2	319	-	318	316	-	-	-
3	252	249	-	236	232	230	226
4	121	-	-	-	-	-	-
5	432	-	-	-	-	-	-
	11000	13000	15000	20000	30000	50000	60000
1	-	-	-	-	-	-	-
2	-	-	-	-	313	-	314
3	227	230	229	-	-	227	222
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
	66000	70000	84000	90000	100000	110000	130000
1	-	-	-	-	-	-	-
2	315	313	315	-	313	315	313
3	227	222	225	223	224	222	223
4	-	-	-	-	-	-	-
5	-	-	-	-	-	432	433
	140000	150000	154000	160000	170000	180000	190000
1	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-
3	222	233	225	223	235	233	239
4	-	-	-	-	-	-	-
5	432	-	-	-	-	-	-

C-2-1-2 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS (ADDITIONAL-REFERENCE-BITS ALGORITHM WITH MEMORY ALLOCATION 473,600 Bytes)

Process ID	Intervals						
	195000	200000	210000	250000	270000	300000	340000
1	163	-	-	-	-	-	-
2	313	-	315	319	313	317	-
3	222	224	233	-	-	229	235
4	121	-	-	-	-	-	-
5	433	432	433	435	432	-	-

Process ID	350000	360000	400000	500000	1000000
1	-	-	-	-	-
2	-	313	-	319	317
3	252	229	237	236	234
4	-	-	-	-	-
5	-	-	-	433	436

Process ID	2000000	3000000	4000000	10000000	80000000
1	-	-	-	-	-
2	319	-	-	-	-
3	-	252	-	-	-
4	-	-	-	-	-
5	437	-	-	-	-

C-2-2-1 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS (CLOCK ALGORITHM WITH MEMORY ALLOCATION 473,600 Bytes)

Process ID	Intervals						
	10	100	1000	5000	10000	14000	15000
1	163	-	-	-	-	-	-
2	319	318	317	316	-	-	-
3	251	250	247	236	-	232	-
4	121	-	-	-	-	-	-
5	435	433	432	-	-	-	-
	16800	19600	20000	25000	27000	28000	29000
1	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-
3	235	-	236	235	233	231	243
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
	30000	31000	32000	33000	34000	35000	36000
1	-	-	-	-	-	-	-
2	315	316	-	-	-	-	-
3	232	234	235	233	246	232	233
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
	37000	38000	39000	39200	40000	42000	50000
1	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-
3	232	-	231	234	236	231	235
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

C-2-2-2 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS (CLOCK ALGORITHM WITH MEMORY ALLOCATION 473,600 Bytes)

Process	Intervals						
ID	60000	70000	80000	100000	500000	600000	700000
1	163	-	-	-	-	-	-
2	315	316	-	-	-	-	-
3	231	230	235	-	-	250	251
4	121	-	-	-	-	-	-
5	432	-	-	-	-	-	-

	800000	1000000	1500000	2000000	3000000	80000000
1	-	-	-	-	-	-
2	-	-	-	-	-	-
3	-	235	251	-	-	-
4	-	-	-	-	-	-
5	-	-	-	-	-	-

C-2-3-1 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS (ADDITIONAL-REFERENCE-BITS ALGORITHM WITH MEMORY ALLOCATION 504,320 Bytes)

Process	Intervals						
ID	10	100	500	1000	8000	10000	11000
1	163	-	-	-	-	-	-
2	319	-	318	316	-	-	-
3	246	247	246	236	230	226	225
4	121	-	-	-	-	-	-
5	432	-	-	-	-	-	-
	15000	20000	30000	50000	60000	66000	70000
1	-	-	-	-	-	-	-
2	-	-	313	-	-	314	313
3	-	-	-	223	222	223	222
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
	84000	90000	100000	110000	130000	140000	150000
1	-	-	-	-	-	-	-
2	-	310	313	314	313	313	-
3	-	221	222	221	221	-	225
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
	154000	160000	170000	190000	196000	200000	210000
1	-	-	-	-	-	-	-
2	-	-	-	-	-	-	307
3	222	221	-	-	-	222	225
4	-	-	-	-	-	-	-
5	-	-	-	-	433	432	433

C-2-3-2 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS (ADDITIONAL-REFERENCE-BITS ALGORITHM WITH MEMORY ALLOCATION 504,320 Bytes)

Process	Intervals				
ID	250000	300000	400000	500000	1000000
1	163	-	-	-	-
2	319	317	313	317	319
3	225	-	227	226	229
4	121	-	-	-	-
5	432	-	-	436	437
	2000000	3000000	5000000	10000000	80000000
1	-	-	-	-	-
2	-	-	-	-	-
3	-	247	-	-	-
4	-	-	-	-	-
5	-	-	-	-	-

C-2-4-1 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS
(CLOCK ALGORITHM WITH MEMORY ALLOCATION 504,320 Bytes)

Process	Intervals						
ID	10	100	1000	5000	10000	14000	16800
1	163	-	-	-	-	-	-
2	319	318	317	-	-	316	-
3	247	246	243	233	-	228	233
4	121	-	-	-	-	-	-
5	435	433	432	-	-	-	-
	19600	20000	25000	27000	28000	29000	30000
1	-	-	-	-	-	-	-
2	-	317	-	316	-	-	-
3	-	-	231	229	227	237	228
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
	31000	32000	33000	34000	35000	36000	36400
1	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-
3	231	-	229	241	228	233	227
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
	37000	38000	39000	39200	40000	42000	60000
1	-	-	-	-	-	-	-
2	-	-	-	-	317	316	-
3	-	228	227	229	233	228	227
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-

C-2-4-2 THE NUMBER OF PAGE FAULTS ACCORDING TO TIME INTERVALS (CLOCK ALGORITHM WITH MEMORY ALLOCATION 504,320 Bytes)

Process ID	Intervals						
	70000	80000	100000	500000	550000	600000	700000
1	163	-	-	-	-	-	-
2	316	-	-	-	-	-	-
3	226	231	231	-	241	243	-
4	121	-	-	-	-	-	-
5	432	-	-	-	-	-	-
	1000000	1500000	5000000	7000000	10000000	80000000	
1	-	-	-	-	-	-	
2	-	-	-	-	-	-	
3	231	243	-	-	-	-	
4	-	-	-	-	-	-	
5	-	-	-	-	-	-	

APPENDIX D:

PROGRAM LISTING

```
/*/////////////////////////////////////////////////////////////////////////////////
//
//    LRU page replacement algorithm: A new approximation implementation.
//
// This program implements a simulation containing a new approach that applies splay tree
// as a data structure to implement the LRU page replacement algorithm. To evaluate the
// performance of the splay tree LRU replacement algorithm, two popular LRU approximation
// algorithms (i.e., clock algorithm and additional-reference-bits algorithm) are also
// implemented. The performance factors are the number of page faults and memory
// utilization. This simulation is implemented as a trace-driven model. The sampled traces
// developed at "Parallel Architecture Research Laboratory" of New Mexico State University
// are used as inputs to this simulation.
//
/////////////////////////////////////////////////////////////////////////////////*/

/*/////////////////////////////////////////////////////////////////////////////////
//
//                              Myhead.h
//
// This is the header file to implement the simulation.
/////////////////////////////////////////////////////////////////////////////////*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/times.h>

#define         YES     1
#define         NO      0
#define         ON      1
#define         OFF     0
#define         MAX_PROCESS     10              /* maximum degree of multiprogramming */

/* Structure that represents a node of a splay tree.
   - parent pointer is needed to do a bottom-up pass over an access path when splaying.
   - height field is needed to compute the height of each leaf. It is only used to
     implement the method that replace a highest leaf node. */

struct stree {
        int     page_num;                       /* a page is in the main memory */
        int     height;                         /* height of the node */
        struct stree    *right;                 /* indicates right child node */
        struct stree    *left;                  /* indicates left child node */
        struct stree    *parent;                /* indicates parent node */
        };
typedef struct stree PAGE_TABLE1;

PAGE_TABLE1     *root[MAX_PROCESS];             /* indicates each root of splay
                                                   tree for each process */
```

```
PAGE_TABLE1    *High_leaf[MAX_PROCESS];       /* indicates each highest leaf node
                                                 of splay tree for each process */

/* Structure used to link all leaves in a tree. This link is used to find the least
   recently used node among the leaves. To do this, it should be a queue. */

struct leaf_list {
       struct stree   *leaf;                  /* leaf node */
       int     e_flag;                        /* to know whether or not the leaf node
                                                 existed in the previous tree */
       struct leaf_list    *next;
       };
typedef struct leaf_list LEAF_L;

LEAF_L *lqhead[MAX_PROCESS];                  /* indicates head of leaf queue */
LEAF_L *lqtail[MAX_PROCESS];                  /* indicates tail of leaf queue */

/* Structure that represents an entry for the circular queue. Circular queue is used to
   implement clock algorithm. */

struct circular_que {
       int     rbit;                          /* reference bit */
       int     page_num;                      /* a page is in the main memory */
       struct circular_que    *next;
       };
typedef struct circular_que PAGE_TABLE2;

PAGE_TABLE2    *cqhead[MAX_PROCESS];          /* head of circular queue */
PAGE_TABLE2    *Hand[MAX_PROCESS];            /* indicates the entry that has
                                                 the oldest page */
PAGE_TABLE2    *before[MAX_PROCESS];          /* indicates the just previous entry of the
                                                 entry indicated by Hand pointer */

/* Structure that represents the 8-bit shift register. This contains ordering information
   of references for each page. Bit field is needed to shift 1 bit and to change leftmost
   bit. */

struct s_reg {
              unsigned int unused:7;
              unsigned int first:1;           /* leftmost bit */
            };
typedef struct s_reg SHIFT_REGISTER;

/* Structure that represents an entry page table to implement additional-reference bits
   algorithm. It has an 8-bit shift register. The union is used to know the value of shift
   register. */

struct add_ref {
       int     page_num;
       union shift {
              unsigned int   value:8;
              SHIFT_REGISTER reg;
              } shift_reg;                    /* 8-bit shift register */
       struct add_ref *next;
       };
typedef struct add_ref PAGE_TABLE3;

PAGE_TABLE3 *add_tail[MAX_PROCESS];           /* indicates tail of page table */

/* Structure that represents a header of page table */

struct add_table {
       int     num;                           /* number of pages in main memory */
       struct add_ref *next;
```

```
        };
typedef struct add_table HEAD_PTB3;

HEAD_PTB3 *Add_table[MAX_PROCESS];          /* each head of page table for each
                                               process */

/*  Structure that is used to implement the blocked queue */

struct  blocked_que {
        int     process_id;                 /* process that got blocked */
        int     enter_time;                 /* time when process enter blocked queue */
        struct blocked_que *next;           /* point next blocked process */
        };
typedef struct blocked_que BLOCK_Q;

BLOCK_Q *head;                              /* head of blocked queue */
BLOCK_Q *tail;                              /* tail of blocked queue */

/* Global Varables  */
int     CLOCK;                              /* virtual clock of system */
int     No_process;                         /* number of processes which execute at the
                                               same time */
int     Strategy;                           /* indicates algorithm to perform */
int     New_method;                         /* four different methods of the new
                                               implementation */
int     interval;                           /* time interval for the clock and additional
                                               -reference-bits algorithms */
int     Frame[MAX_PROCESS];                 /* number of frames provided to a process */
int     No_in_tree[MAX_PROCESS];            /* number of pages in splay tree */
int     No_in_cq[MAX_PROCESS];              /* number of pages in circular queue */


/*///////////////////////////////////////////////////////////////////////////////////////
//
//                              Perform.c
//
// This is the main file of the simulation. Input traces are available at the
// tracebase@nmsu.edu using anonymous ftp. These traces are in dinero+ format. These raw
// trace files were converted to pages. The files containing pages were used as input. The
// file names and the lengths of reference strings are stored in file "traces.dat".
///////////////////////////////////////////////////////////////////////////////////////*/

#include "myhead.h"

/* variables used in this file */
int     T_frame;                            /* total number of frames of main memory */
int     Mem_size;                           /* size of main memory (byte) */
int     M_mem_allo;                         /* method of memory allocation */
int     page_size;                          /* size of page (byte) */
int     Pfh_time;                           /* page fault handling time */
int     finish;                             /* to indicate how many jobs are finished */
int     No_blockedQ;                        /* number of processes in blocked queue */
int     Idle_time;                          /* CPU idle time */
int     len_util;                           /* index of array to store memory utilization
                                               at each virtual time interval */
FILE    *fptr[MAX_PROCESS];                 /* file descriptor to indicate each input
                                               trace file; one process executes one file
                                            */
int     F_blockQ[MAX_PROCESS];              /* flag to indicate whether or not each
                                               process gets blocked */
int     F_finish[MAX_PROCESS];              /* flag to indicate whether each process was
                                               finished */
int     cur_pos[MAX_PROCESS];               /* current position of file pointer */
int     how_much[MAX_PROCESS];              /* length of reference strings that should be
                                               processed */
int     amt_done[MAX_PROCESS];              /* amount of reference strings performed
                                               within one headway in reference strings
```

```
                                                    (how_much) */
int     No_pagefault[MAX_PROCESS];             /* number of page faults for each process */
char    trace_name[MAX_PROCESS][80];           /* name of input trace file */
int     len_refstr[MAX_PROCESS];               /* length of reference strings for each
                                                  process */
int     amt_performed[MAX_PROCESS];            /* total amount performed in each process */
float   M_util[MAX_PROCESS];                   /* memory utilization at each time interval
                                               */

/* functions used in this file */
void    GetPage(void);
void    Perform(void);
void    PrintMemutil(void);
void    ChooseMethod(void);
void    ChooseInterval(void);
void    Initialize(void);
void    CalMemutil(void);
void    GoToBlockedQ(int run_process,int amount);
int     ProcessHandling(int run_p,int howmuch);
BLOCK_Q *CheckBlockedQ(void);
void    ClearMem(int run_p);

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function: main()
//      Purpose : This is the main function of this program. It shows the main memu and
//                gets the selection from user. Upon a selection, it calls appropriate
/                 functions.
/////////////////////////////////////////////////////////////////////////////////////*/

void main(void)
{
   int          num;
   char         cnum[81];
   char         c;

   system("tput clear");
       printf("\t ==================================================\n");
   printf("\t =                                                =\n");
   printf("\t =         LRU PAGE REPLACEMENT ALGORITHM:        =\n");
   printf("\t =       A NEW APPROXIMATION IMPLEMENTATION       =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                     By:                        =\n");
   printf("\t =                 Jung, Eunjae                   =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                   Advisor:                     =\n");
   printf("\t =             Dr. M. H. Samadzadeh               =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t =                                                =\n");
   printf("\t ==================================================\n");
   printf(" Enter the any key to continue:");
   scanf("%c", &c);

   /* main memu for the simulation */
   for(;;)
   {
       printf("\t (Note: The simulation takes 1 to 2 hours.)\n");
        printf("\t ---------------------------------------------------\n");
```

```
        printf("\t -                          MENU                          - \n");
        printf("\t -                                                        - \n");
        printf("\t -  1. Convert virtual addresses to virtual pages.        - \n");
        printf("\t -  2. Perform the simulation.                            - \n");
        printf("\t -  3. Generate graph for page faults.                    - \n");
        printf("\t -  4. Generate graph for memory utilization.             - \n");
        printf("\t -  5. Exit the simulation.                               - \n");
        printf("\t ---------------------------------------------------------- \n");
        printf(" Select a number: ");
        scanf("%d",&num);
        switch(num)
        {

                case 1: GetPage();          /* convert virtual addresses to virtual pages
                                            */
                        break;

                case 2: Perform();          /* perform the simulation using three
                                               different algorithms */
                        break;

                case 3: PageFaultGraph();   /* generate the performance graph for page
                                               faults */
                        break;

                case 4: MemUtilGraph();     /* generate the performance graph for memory
                                               utilization */

                case 5: exit(0);            /* exit the simulation */
                default: printf("\n Invalid input, Try again. \n");
                        gets(cnum);
                        num=0;
                        break;

        }
    }
}

/*/////////////////////////////////////////////////////////////////////////////
//      Function : GetPage()
//      Purpose  : This function is used to convert virtual addresses to pages. It takes
//                 dinero+ format file as input and writes pages to output file. This
//                 output file is used as input to drive the simulation.
/////////////////////////////////////////////////////////////////////////////*/

void    GetPage(void)
{
        FILE    *fp1;                       /* for input file */
        FILE    *fp2;                       /* for output file */
        int     i,j,line;
        char    buf[200];
        char    inputfile[80];              /* name of input file */
        char    outfile[80];                /* name of output file */
        unsigned long  virtual;             /* virtual address */
        unsigned long  page;                /* virtual pages */

        printf("\n\t ====== Convert virtual addresses to pages ===== ");
        printf("\n\t Print the filename which has virtual addresses: ");
        scanf("%s",inputfile);              /* get name of input file */
        printf("\n\t Print the filename to keep the converted pages: ");
        scanf("%s",outfile);                /* get name of output file */
        printf("\n\t Select the page size (512, 1024, 2048, 4096 or 8192): ");
        scanf("%d", &page_size);            /* get page size */

        fp1=fopen(inputfile,"r");
        fp2=fopen(outfile,"w");
```

```
        line=0;                              /* to check the length of the reference
                                                string */
        while(!feof(fp1))
        {
                fgets(buf,200,fp1);
                sscanf(buf,"%d%x%x",&i,&virtual,&j);
                page=virtual/page_size;      /* get the page from virtual address */
                fprintf(fp2,"%d\n",page);
                line++;                      /* compute the length of reference string */

                memset(buf,80,'\0');
        }
        printf(" Length of Reference Strings:%d\n",line);
        fclose(fp1);
        fclose(fp2);
        free(inputfile);
        free(outfile);
        free(buf);
}

/*////////////////////////////////////////////////////////////////////////////////////
//      Function : Perform()
//      Purpose  : This function is used to perform the simulation.
//                 1. Get the input parameter from standard input(keyboard), and get the
//                    file names and the lengths of reference strings of input traces from
//                    file "traces.dat".
//                 2. CPU scheduling - to get a process to be executed, first check if
//                    there is a process that finished its I/O in blocked queue. If none
//                    exists, using random number generator, a process and the length of
//                    reference strings to be processed are selected. This is repeated
//                    until all processes finish.
//                 3. Print the performance parameters, number of page faults and memory
//                    utilization, for each process.
////////////////////////////////////////////////////////////////////////////////////*/

void    Perform(void)
{
        int             i,j;
        FILE            *fp;
        char            buf_str[100];
        int             No_ready;
        BLOCK_Q         *p;
        int             run_p;
        int             start_pos;
        int             nextlen;
        int             block;

                                             /* get input parameters */
        printf("\t ------------------ Put the Input -------------------- \n");

        valid=0;                             /* get number of processes */
        while(valid != 1 )
        {
                printf("\t * Number of processes (between 1 to 10): ");
                scanf("%d",&No_process);
                if ( No_process > MAX_PROCESS  )
                {                            /* # of processes larger than 10 (maximum
                                                degree of multiprogramming */
                        printf("\n\tError: Maximum # of processes is 10, try again. \n");
                        gets(buf_str);
                        valid=0;
                }
                else if( No_process == 0 )
                {                            /* input is invalid */
                        printf("\n\tError: Invalid input, try again. \n");
                        gets(buf_str);
                        valid=0;
```

```
        }
        else
                valid=1;

        if ( valid == 1 )
        {
                                        /* get the file names and the lengths of
                                           input traces */
                fp=fopen("traces.dat","r");
                j=0;
                memset(buf_str,80,'\0');
                while(!feof(fp))
                {
                    fgets(buf_str,80,fp);
                    memset(trace_name[j],80,'\0');
                    sscanf(buf_str,"%s%d",trace_name[j],&len_refstr[j]);
                    j++;                /* compute number of input traces */
                    memset(buf_str,80,'\0');
                }
                fclose(fp);
                if ( (j-1) < No_process)
                {                       /* number of processes more than number of
                                           input traces */
                    printf("\n\tError: # of processes is more than # of
                                         traces (5).\n");
                    gets(buf_str);
                    valid=0;
                }
        }

        if (valid == 0 )
                No_process=0;           /* initialize number of processes */
}

valid = 0;
while(valid != 1 )                      /* get the memory size */
{
        printf("\t * Memory Size (minimum size is 512 * # of processes): ");
        scanf("%d",&Mem_size);
        if ( Mem_size < (512*No_process))
        {
                if ( Mem_size == 0 )   /* input is invalid */
                        printf("\n\tError: Invalid input, try again.\n");
                else                    /*memory size is smaller than minimum size */
                        printf("\n\tError: Memory size is too small, try again.\n");
                valid=0;
                gets(buf_str);
                Mem_size=0;             /* initialize variable for memory size */
        }
        else
                valid =1;
}

valid=0;
while(valid != 1 )                      /* get the method of memory allocation */
{
        printf("\t * Method of memory allocation.\n");
        printf("\t      - 1. Propotional\n");
        printf("\t      - 2. Eqaul \n");
        printf("\t      - 3. Exit (return to menu) \n");
        printf("\t * Select a method: ");
        scanf("%d",&M_mem_allo);
        if ( (M_mem_allo == 1 ) || (M_mem_allo == 2)  ||
             (M_mem_allo == 3 ) )
        {
                valid=1;
                if (M_mem_allo == 3 )
```

```
                return;
        else
        {                               /* invalid input */
                printf("\n\tError: Invalid input, try again.\n");
                valid=0;
                gets(buf_str);
                M_mem_allo=0;           /* initialize variable */
        }
}
if ( M_mem_allo == 1)
        printf("\n\t Future work!  Memory will be allocated equally here.\n");

valid =0;
while(valid != 1)                       /* get page fault handling time */
{
        printf("\t * Page fault handling time (between 10000 and 600000):");
        scanf("%d",&Pfh_time);
        if ( (Pfh_time < 10000 ) || ( Pfh_time > 600000 ))
        {
                if (Pfh_time == 0)      /* input is invalid */
                        printf("\n\tError: Invalid input, try again.\n");
                else                    /* page fault handlig time is not the trivial
                                           range */
                        printf("\n\tError: Too small or too large, try again.\n");
                valid=0;
                gets(buf_str);
                Pfh_time=0;             /* initialize the variable */
        }
        else
                valid=1;
}

valid=0;
while(valid != 1)                       /* get the page size */
{
        printf("\t Select the page size (512, 1024, 2048, 4096, or 8192): ");
        scanf("%d", &page_size);
        if( (page_size == 512) || (page_size == 1024) ||
            (page_size == 2048) || (page_size == 4096) ||
            (page_size == 8192 ) )
                valid =1;
        else
        {
                if(page_size == 0 )     /* input is invalid */
                        printf("\n\tError: Invalid input, try again.\n");
                else
                        printf("\n\t Error: Choose one among 5 page sizes, try
                                again. \n");
                valid = 0;
                gets(buf_str);
                page_size=0;
        }
}

valid =0;
while(valid != 1)                       /* get the algorithm to perform */
{
        printf("\t * Page replacement algorithms");
        printf("\n\t    - 1. New implementation ");
        printf("\n\t    - 2. Clock algorithm ");
        printf("\n\t    - 3. Additional-reference-bits algorithm");
        printf("\n\t    - 4. Exit (return to menu)");
        printf("\t Select a algorithm: ");
        scanf("%d",&Strategy);
        switch(Strategy)
        {
            case 1: ChooseMethod();   /* in the new implementation */
```

```
                valid=1;
                break;
        case 2: ChooseInterval(); /* in the clock algorithm */
                valid=1;
                break;
        case 3: ChooseInterval(); /* in the add-ref-bits algorithm */
                valid=1;
                break;
        case 4: return;
        default: printf("\n\t Invalid input, try again. \n");
                 gets(buf_str);
                 valid=0;
        }
}

T_frame=Mem_size/page_size;          /* total frames needed */

if( (M_mem_allo == 1 ) || (M_mem_allo == 2 ) )
                                     /* local allocation: memory is equally
                                        partitioned to each process */
{
        for(i=0;i<No_process;i++)
                Frame[i]=T_frame/No_process;
}

Initialize();                        /* initialize the global variables */
finish=0;                            /* number of processes finished */

while(finish < No_process)
{
    p=CheckBlockedQ();              /* check blocked queue */
    if ( p == NULL )
                                     /* two cases: 1. No process in blocked queue.
                                                   2. No process which has done
                                                      I/O completely. */
    {
        No_ready=No_process-No_blockedQ;
        if ( No_process == (finish+No_blockedQ))
                block=YES;
        else
                block=NO;

        if ( ( No_ready == 0 ) || ( block == YES ) )
        {
                                     /* all processes were in the blocked queue or
                                        finished */
                Idle_time ++;        /* no process is running */
                CLOCK++;
                if(CLOCK%500 ==0)
                        CalMemutil();
        }
        else
        {
                run_p=0;

                                     /* get the process to be executed randomly */
                run_p=rand()%No_process;
                if ((F_blockQ[run_p] != ON ) && (F_finish[run_p] != ON))
                {
                                     /* get the length of headway in reference
                                        string randomly */
                        how_much[run_p]=rand()%len_refstr[run_p];
                        amt_done[run_p]=0;
                        nextlen=amt_performed[run_p]+how_much[run_p];
                        if ( nextlen > len_refstr[run_p])
                        {
                                how_much[run_p]=len_refstr[run_p]-
```

```
                                                    amt_performed[run_p];
                                    if (how_much[run_p]==0)
                                    {
                                            F_finish[run_p]=ON;
                                            /* compute the number of processes
                                               completed */
                                            finish++;
                                            /* release the memory used the process
                                               completed */
                                            ClearMem(run_p);

                                    }
                            }
                            else            /* run a process */
                                    ProcessHandling(run_p,how_much[run_p]);
                    }
            }
   }
   else
   {
                                            /* there was a process that finished its I/O
                                               in the blocked queue */
        run_p=p->process_id;
        ProcessHandling(run_p,how_much[run_p]);
   }
}

switch(Strategy)                            /* write information about input */
{
    case 1: printf("\n\t Algorithm: New Implementation ");
            printf(" Method: %d\n",New_method);
            switch(New_method)
            {
               case 1 : printf(" Method: leftmost leaf\n");
                        break;
               case 2 : printf(" Method: rightmost leaf\n");
                        break;
               case 3 : printf(" Method: highest leaf\n");
                        break;
               case 4 : printf(" Method: LRU leaf\n");
                        break;
             }
             break;
     case 2 : printf("\n\t Algorithm: Clock algorithm ");
             printf(" Interval -> %d\n",interval );
             break;
     case 3 : printf("\n\t Algorithm: Additional-reference-bits ");
             printf(" Interval -> %d\n",interval );
             break;
}
printf("\n\t Memory size: %d",Mem_size);
printf("\n\t Page size: %d",page_size);
printf("\n\t Page fault handling time: %d\n",Pfh_time);

                                            /* print the number of page faults for each
                                               process */
printf("\n\t ---------------- Page Fault Numbers -----------------\n ");
for(i=0;i<No_process;i++)
        printf("\t Process %d  -> %d\n",i,No_pagefault[i]);
printf("\t ---------------------------------------------------\n ");
printf("\t CLOCK is %d", CLOCK);
printf("\t IDLE TIME is %d", Idle_time);

                                            /* print the memory utilization */
PrintMemutil();
printf("\t ---------------------------------------------------\n ");
```

```
}

/*////////////////////////////////////////////////////////////////////////////////
//      Function : PrintMemutil()
//      Purpose  : This function is used to print average memory utilization for each
//                 process.
////////////////////////////////////////////////////////////////////////////////*/

void PrintMemutil(void)
{
        int i,j;
        float A_mutil[MAX_PROCESS];

        printf("\n\t ------ Average Memory Utilization -------- ");
        for(i=0;i<No_process;i++)
        {
                                                /* compute average memory utilization */
                A_mutil[i]=M_util[i]/(float)len_util;
                                                /* print the average memory utilization */
                printf("\n\t Process %d => %f",i,A_mutil[i]);
        }
        printf("\n");
}

/*////////////////////////////////////////////////////////////////////////////////
//      Function : ChooseMethod()
//       Purpose  : This function is used to choose the method which selects a victim page
//                  in the new implementation.
////////////////////////////////////////////////////////////////////////////////*/

void ChooseMethod(void)
{
        int     valid;
        char    buf[81];

        valid=0;
        while(valid != 1)
        {
                printf("\n\t ------------- New Implementation ----------------");
                printf("\n\t * Select the method");
                printf("\n\t      - 1. Replace the leftmost leaf");
                printf("\n\t      - 2. Replace the rightmost leaf");
                printf("\n\t      - 3. Replace the highest leaf ");
                printf("\n\t      - 4. Replace the LRU leaf: ");
                scanf("%d",&New_method);
                if ( (New_method <1 ) || ( New_method > 4 ) )
                {
                        gets(buf);
                        printf("\n\tError: Invalid input, try again.\n");
                        valid=0;
                }
                else
                        valid=1;
        }
}

/* ////////////////////////////////////////////////////////////////////////////////
//      Function : ChooseInterval()
//      Purpose  : This function is used to get the time interval. The time intervals are
//                  used in the clock algorithm and the additional-reference-bits
//                  algorithm.
////////////////////////////////////////////////////////////////////////////////*/

void ChooseInterval(void)
{
        int valid;
        char    buf[81];
```

```
        valid=0;
        while(valid != 1)
        {
                printf("\n\t * Put the interval(between 1 and 100000000):");
                scanf("%d",&interval);
                if (interval == 0 )
                {
                        printf("\n\tError:Invalid input, try again.");
                        gets(buf);
                        valid=0;
                }
                else
                        valid=1;
        }
}

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function : Initialize()
//      Purpose  : This function is used to initialize the variables.
/////////////////////////////////////////////////////////////////////////////////////*/

void Initialize(void)
{
        int i;

        CLOCK=0;                        /* set virtual clock to 0 */
        len_util=0;                     /* initialize variable for memory utilization
                                         */
        Idle_time=0;                    /* initialize cpu idle time */
        No_blockedQ=0;                  /* initialize number of processes blocked */
        head=tail=NULL;                 /* initialize header and tail of blocked
                                           queue */

        for(i=0;i<No_process;i++)
        {
                amt_performed[i]=0;     /* initialize the amount of reference string
                                           to be processed */
                root[i]=NULL;           /* initialize root of splay tree, page table
                                           for new implementation */
                High_leaf[i]=NULL;      /* initialize pointer indicates the highest
                                           leaf */
                lqhead[i]=NULL;         /* initialize header of leaf queue */
                lqtail[i]=NULL;         /* initialize tail of leaf queue */
                cqhead[i]=NULL;         /* initialize header of circular queue */
                Hand[i]=NULL;           /* initialize hand pointer */
                before[i]=NULL;
                                        /* initialize the header of linked list for
                                           additional-reference-bits algorithm */
                Add_table[i]=(PAGE_TABLE3 *)malloc(sizeof(PAGE_TABLE3));
                Add_table[i]->next=NULL;
                Add_table[i]->num=0;
                F_blockQ[i]=OFF;        /* clear flag for process blocked */
                F_finish[i]=OFF;        /* clear flag for process completed */
                No_in_cq[i]=0;          /* initialize number of pages in circular
                                           queue */
                No_in_tree[i]=0;        /* initialize number of pages in splay tree
                                           */
                No_pagefault[i]=0;      /* initialize number of page faults */
        }
}

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function: ProcessHandling()
//      Purpose : This is used to manage the running process. When the process executes a
//                page, first check whether or not this page is in memory. If the page
//                does not exist, the runnig process goes to the blocked queue. Else, the
```

```
//              running process continues to execute until it finishes the amount of
//              reference strings which the process should progress.
///////////////////////////////////////////////////////////////////////////////////*/

void    ProcessHandling(int run_p,int howmuch)
{
        int             len[MAX_PROCESS];
        char            buf[100];
        int             page;                   /* referenced page */
        int             page_fault;             /* indicates whether or not a page fault
                                                   occurs */
        int             free_frame;             /* number of frames available */
        PAGE_TABLE2     *tmp;

        fptr[run_p]=fopen(trace_name[run_p],"r");
        fseek(fptr[run_p],cur_pos[run_p],0); /* go to the current position of file */
        while( (!feof(fptr[run_p])) && (amt_done[run_p] < howmuch))
        {
                fgets(buf,100,fptr[run_p]);     /* gets a page to be executed */
                sscanf(buf,"%d",&page);
                amt_done[run_p]++;
                amt_performed[run_p]++;
                len[run_p]++;

                                                /* one clock cycle is used to access the
                                                   memory */
                CLOCK++;

                if ( (CLOCK % 500) == 0)        /* calculates memory utilization at every 500
                                                   clocks */
                        CalMemutil();

                                                /* checks if page is in main memory */
                page_fault=CheckPageTable(page,run_p);
                if ( page_fault == YES )
                {
                        No_pagefault[run_p]++;/* increases the number of page faults */

                                               /* running process will be blocked */
                        No_blockedQ++;
                        F_blockQ[run_p]=ON;
                        GoToBlockedQ(run_p);
                        cur_pos[run_p]=ftell(fptr[run_p]);
                        fclose(fptr[run_p]);
                        return;
                }
        }
        if (feof(fptr[run_p]))                /* one process is completed */
                finish++;                     /* increase the number of processes completed
                                              */

        else
        {                                     /* get the current position from the beginning
                                                 of the file */
                cur_pos[run_p]=ftell(fptr[run_p]);
                fclose(fptr[run_p]);
        }
 }

/*///////////////////////////////////////////////////////////////////////////////////
//      Function : CalMemutil()
//      Purpose  : This function is used to calculate the memory utilization at every 500
//              clocks.
///////////////////////////////////////////////////////////////////////////////////*/

void CalMemutil(void)
{
```

```
        int     i;
        float   mul;

        for(i=0;i<No_process;i++)
        {
                switch(Strategy)
                {
                                                /* calculate memory utilization as the
                                                   algorithm*/
                                                /* new implementation */
                        case 1: mul=(float)No_in_tree[i]/(float)Frame[i];
                                                /* add memory utilization calculated to
                                                   compute average value */
                                M_util[i]=mul+M_util[i];
                                break;

                                                /* clock algoritm */
                        case 2: mul=(float)No_in_cq[i]/(float)Frame[i];
                                M_util[i]=mul+M_util[i];
                                break;

                                                /* additional-reference-bits algorithm */
                        case 3: mul=(float)Add_table[i]->num/(float)Frame[i];
                                M_util[i]=mul+M_util[i];
                                break;
                }
        }
        len_util++;                             /* increase the number of memory utilizations
                                                   computed */
}

/*/////////////////////////////////////////////////////////////////////////////////////////
//      Function : CheckBlockedQ()
//      Purpose  : This function is used to check the blocked queue whether or not a
//                 blocked process which has been finished its I/O exists. If the process
//                 exists, the process is removed from the blocked queue.
/////////////////////////////////////////////////////////////////////////////////////////*/

BLOCK_Q *CheckBlockedQ(void)
{
        BLOCK_Q *tmp;

        if ( head == NULL )                     /* no process in the blocked queue */
                return(NULL);
        else
        {
                                                /* a blocked process which has been finished
                                                   its I/O exists */
                if ( (CLOCK - head->enter_time) >= Pfh_time)
                {
                                                /* removes the process from the blocked queue
                                                   */
                        tmp=head;
                        head=head->next;
                        No_blockedQ--;
                        F_blockQ[tmp->process_id]=OFF;
                        return(tmp);
                }
                else                            /* no process which has been finished */
                        return(NULL);
        }
}

/*/////////////////////////////////////////////////////////////////////////////////////////
//      Function : GoToBlockedQ
//      Purpose  : This function is used to send the process that has the page fault
//                 occurred to the blocked queue.
```

```
//////////////////////////////////////////////////////////////////////////////////*/

void GoToBlockedQ(int run_process)
{
        BLOCK_Q  *new;

                                                /* creates a new entry */
        new=(BLOCK_Q *)malloc(sizeof(BLOCK_Q));
        new->process_id=run_process;
        new->enter_time=CLOCK-1;
        new->next=NULL;

        if ( head == NULL )
        {                                       /* blocked queue is empty */
                head=new;
                tail=head;
        }
        else
        {
                                                /* add new entry to the tail of queue */
                tail->next=new;
                tail=tail->next;
        }
}

/*//////////////////////////////////////////////////////////////////////////////////
///     Function : CheckPageTable()
//      Purpose  : This function is used to check page table whether or not the page is in
//                 main memory. If a page fault occurs, return YES. Else, return NO.
//////////////////////////////////////////////////////////////////////////////////*/

int CheckPageTable(int page,int run_p)
{
        int     re;

        switch(Strategy)
        {
                case 1:  re = NewApproach(page,run_p);
                         break;
                case 2 : re = ClockAlg(page,run_p);
                         break;
                case 3 : re = AdditionalRefAlg(page,run_p);
                         break;
        }
        if (re == YES )
                return(YES);                    /* a page fault occurs */
        else
                return(NO);                     /* no page fault occurs */
}

/*//////////////////////////////////////////////////////////////////////////////////
//      Function : ClearMem()
//      Purpose  : This function is used to clear the memory which has been used by a
//                 process when the process finishes its execution.
//////////////////////////////////////////////////////////////////////////////////*/

void ClearMem(int run_p)
{
        switch(Strategy)
        {
                case 1: No_in_tree[run_p]=0;  /* release the memory used*/
                         break;
                case 2: No_in_cq[run_p]=0;    /* release the memory used */
                         break;
                case 3: Add_table[run_p]->num=0;
                         break;
        }
```

```
}

/*/////////////////////////////////////////////////////////////////////////////////////
//
//                              Newapp.c
//
// This file is to implement new implementation of LRU using splay tree as page table.
// This algorithm has 4 different methods to find the victim page which should be
// replaced.
//      1. Leftmost leaf:  Select the leftmost leaf as a victim page.
//      2. Rightmost leaf: Select the rightmost leaf as a victim page.
//      3. Highest leaf:   Select the leaf which has the highest height. It means the leaf
//                          is the farthest node from the root.
//     4. LRU leaf:       Select the leaf which is LRU among the leaves.
// Note : The height of root is 0. Each process has its own page table. Making the page
// table, checking the page fault and replacing a victim page when a page fault occurs are
// included in this file.
/////////////////////////////////////////////////////////////////////////////////////*/

#include "myhead.h"

int     s_exist;                            /* a flag to indicate whether or not page is
                                               in the tree */
int     leaf_num;                           /* the number of leaves in the tree */

                                            /* Functions used in this file */
PAGE_TABLE1 *Search(int page, PAGE_TABLE1 *rt, int run_p);
PAGE_TABLE1 *Splaying(PAGE_TABLE1 *cur,int run_p);
PAGE_TABLE1 *gp(PAGE_TABLE1 *x);
PAGE_TABLE1 *FindOldPage(PAGE_TABLE1 *top,int run_p);
void Insert(PAGE_TABLE1 *fa,int page,int run_p);
void RotateLeft(PAGE_TABLE1 *y,int run_p);
void RotateRight(PAGE_TABLE1 *y,int run_p);
void GetHeight(PAGE_TABLE1 *node,int run_p);
void RemovePage(int run_p);
void GetLeafQue(int run_p);

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function : NewApproach()
//      Purpose  : This is main function of Newapp.c file. Check whether or not the page
//                 needed immediately is in main memory. If the page is not in the tree,
//                 insert the page into the tree. When memory is full, remove the victim
//                 page from the tree. If a page fault is occurred, return YES, else,
//                 return NO.
/////////////////////////////////////////////////////////////////////////////////////*/

int NewApproach(int page,int run_p)
{
        PAGE_TABLE1 *newnode;
        PAGE_TABLE1 *top;
        PAGE_TABLE1 *father_node;           /* indicates the parent node of a node
                                               which will be inserted */
        int         free_frame;             /* the number of free frames */

        top=root[run_p];
        if(top == NULL)                     /* no page is in main memory */
        {
                                            /* make the root */
                newnode=(PAGE_TABLE1 *)malloc(sizeof(PAGE_TABLE1));
                newnode->page_num=page;
                newnode->right = NULL;
                newnode->left = NULL;
                newnode->parent = NULL;
                newnode->height = 0;
                top=newnode;
                No_in_tree[run_p]=1;
                root[run_p]=top;
```

```
            if ( New_method == 4 )          /* LRU leaf method */
            {
                    lqhead[run_p]=(LEAF_L *)malloc(sizeof(LEAF_L));
                    lqhead[run_p]->leaf=newnode;
                    lqhead[run_p]->next=NULL;
                    lqtail[run_p]=lqhead[run_p];
            }
            return(YES);                    /* a page fault occurs */
        }
        else
        {
            s_exist=NO;
            father_node=Search(page,top,run_p);
             if ( father_node == NULL )    /* the node exists in memory */
                    return(NO);             /* no page fault occurs */
            else
            {
                    Insert(father_node,page,run_p);
                    free_frame=Frame[run_p] - No_in_tree[run_p];
                    if (free_frame < 0 )    /* memory is full */
                                            /* remove the victim page */
                            RemovePage(run_p);
                    return(YES);            /* page fault occurs */
            }
        }
}

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function : Search()
//      Purpose  : This function is used to search the node containing the page will be
//                 executed immediately. If the node is in the tree, splay at the node,
//                 and NULL is returned. Else, the parent node of the node will contain
//                 the page is returned.
/////////////////////////////////////////////////////////////////////////////////////*/

PAGE_TABLE1 *Search(int page, PAGE_TABLE1 *rt, int run_p)
{
        PAGE_TABLE1 *top;
        PAGE_TABLE1 *ret_val;

        if ( rt == NULL )
                return(NULL);
        else
        {
                if (rt->page_num == page )
                {
                                            /* the node containing the immediately needed
                                               page is in tree */
                                            /* splay at the node */
                        top=Splaying(rt,run_p);
                        if  ( root[run_p] != top )
                        {                   /* root is changed */

                                root[run_p]=top;
                                            /* after slpaying, the height of each node
                                               and the leaf queue are changed */
                                if ( (New_method == 3) || ( New_method == 4) )
                                {
                                        top->height=0;
                                        High_leaf[run_p]=NULL;
                                        GetHeight(top,run_p);
                                }
                                if (New_method == 4 )
                                        GetLeafQue(run_p);
                        }
                        s_exist=YES;
                        return(NULL);
```

```
            }
            else if (rt->page_num < page )
            {
                                        /* search right subtree */
                    ret_val=Search(page,rt->right,run_p);
                    if ( s_exist == NO )
                    {
                            if( rt->right==NULL)
                                    return(rt);
                            else
                                    return(ret_val);
                    }
                    else
                            return(NULL);
            }
            else if (rt->page_num > page )
            {
                                        /* search left subtree */
                    ret_val=Search(page,rt->left,run_p);
                    if ( s_exist == NO )
                    {
                            if( rt->left == NULL)
                                    return(rt);
                            else
                                    return(ret_val);
                    }
                    else
                            return(NULL);
            }
    }
}

/*/////////////////////////////////////////////////////////////////////////////////////////
//      Function : Insert()
//      Purpose  : This function is used to insert the node into the tree. Parent node was
//                 already taken during the search operation. So it just links the node to
//                 the parent and then splay at the inserted node. Because tree is changed
//                 after splaying, the height and the leaf queue are reproduced.
/////////////////////////////////////////////////////////////////////////////////////////*/

void Insert(PAGE_TABLE1 *fa,int page,int run_p)
{
        PAGE_TABLE1 *newnode;
        int     i;

                                        /* creates a new node */
        newnode=(PAGE_TABLE1 *)malloc(sizeof(PAGE_TABLE1));
        newnode->page_num = page;
        newnode->left=NULL;
        newnode->right=NULL;
        newnode->parent=fa;
        newnode->height=(fa->height)+1;
        No_in_tree[run_p]++;

                                        /* links to the parent */
        if ( page > fa->page_num )
                fa->right=newnode;
        else if ( page < fa->page_num)
                fa->left=newnode;

                                        /* splay the tree at the inserted node */
        root[run_p]=Splaying(newnode,run_p);

                                        /* after slpaying, the height of each node
                                           and leaf queue are changed */
        if ( (New_method == 3 ) || ( New_method ==4 ) )
        {
```

```
			root[run_p]->height=0;
			High_leaf[run_p]=NULL;		/* highest leaf is changed */
			GetHeight(root[run_p],run_p);
		}
		if (New_method ==4 )
			GetLeafQue(run_p);
}

/*/////////////////////////////////////////////////////////////////////////////////////
//	Function : Splaying()
//	Purpose  : This function is used to implement splaying. It rebuilds the tree after
//		   each access that moves the accessed item to the root. To do this,
//		   zig,zig-zag,zig-zig steps are repeated bottom-up along the access path
//		   until the accessed item becomes the root of the tree.
/////////////////////////////////////////////////////////////////////////////////////*/

PAGE_TABLE1 *Splaying(PAGE_TABLE1 *cur,int run_p)
{
	PAGE_TABLE1 *grandfa;

	while(cur->parent != NULL )		/* until cur becomes the root */
	{
		grandfa=gp(cur);
		if ( cur==(cur->parent)->left)
		{
			if (grandfa == NULL )
						/* zig -> rotate the edge joining cur and the
						   root */
				RotateRight(cur->parent,run_p);

			else if ( cur->parent == grandfa->left)
			 {
						/* zig-zig -> rotate the edge joining parent
						   and grand parent and then rotate edge
						   joining cur and parent */
				RotateRight(grandfa,run_p);
				RotateRight(cur->parent,run_p);
			}

			else if ( cur->parent == grandfa->right)
			{
						/* zig-zag ->rotate the edge joining cur and
						   parent and then rotate edge joining cur
						   and grandparent */
				RotateRight(cur->parent,run_p);
				RotateLeft(cur->parent,run_p);
			}
		}
		else if ( cur==(cur->parent)->right)
		{
			if (grandfa == NULL )  /* zig */
				RotateLeft(cur->parent,run_p);

			else if ( cur->parent == grandfa->right)
			{			/* zig-zig */
				RotateLeft(grandfa,run_p);
				RotateLeft(cur->parent,run_p);
			}

			else if ( cur->parent == grandfa->left)
			{			/* zig-zag */
				RotateLeft(cur->parent,run_p);
				RotateRight(cur->parent,run_p);
			}
		}
	}
	return(cur);
```

```
}

/*///////////////////////////////////////////////////////////////////////////////////////
//      Function : gp()
//      Purpose  : This function is used to get the address of grandparent node.
///////////////////////////////////////////////////////////////////////////////////////*/

PAGE_TABLE1 *gp(PAGE_TABLE1 *x)
{
        return((x->parent)->parent);            /* returns grandparents of x */
}

/*///////////////////////////////////////////////////////////////////////////////////////
//      Function : RotateLeft()
//      Purpose  : This function is used to rotate the edge joining y and its
//                 right child.
///////////////////////////////////////////////////////////////////////////////////////*/

void RotateLeft(PAGE_TABLE1 *y,int run_p)
{
        PAGE_TABLE1 *x;                         /* right child of y */
        PAGE_TABLE1 *z;                         /* parent of y */

        x=y->right;
        z=y->parent;

        if ( z != NULL )
        {
                if (z->left == y )
                        z->left = x ;           /* x becomes left child of z */
                else if ( z->right == y )
                        z->right = x;           /* x becomes right child of z */
        }
        y->right=x->left;                       /* left child of x becomes right child of y
                                                */
        x->left=y;                              /* y becomes left child of x */
        x->parent = z;                          /* parent of x becomes z */
        y->parent = x;                          /* parent of y becomes x */

        if ( y->right != NULL )
                (y->right)->parent = y;         /* change the right of y to y */
}

/*///////////////////////////////////////////////////////////////////////////////////////
///     Function : RotateRight()
//      Purpose  : This function is used to rotate the edge joining y and its
//                 left child.
///////////////////////////////////////////////////////////////////////////////////////*/

void RotateRight(PAGE_TABLE1 *y,int run_p)
{
        PAGE_TABLE1 *x;                         /* left child of y */
        PAGE_TABLE1 *z;                         /* parent of y */

        x=y->left;
        z=y->parent;

        if ( z != NULL )
        {
                if (z->left == y )
                        z->left = x ;
                else if ( z->right == y )
                        z->right = x;
        }
        y->left=x->right;
        x->right=y;
        x->parent = z;
```

```
        y->parent = x;

        if ( y->left != NULL )
                (y->left)->parent =y;
}

/*/////////////////////////////////////////////////////////////////////////////////
//      Function : GetHeight()
//      Purpose  : This function is used to get the height of the each node and to get the
//                 list of all leaves in the tree. Prefix tree traversal is used because
//                 the height of the parent must be known to get the height of the node.
/////////////////////////////////////////////////////////////////////////////////*/

void GetHeight(PAGE_TABLE1 *node,int run_p)
{
        LEAF_L *tmp,*prev,*newlq,*nleaf;
        int exist;

        if(node == NULL )
                return;
        else
        {
                if (New_method == 3 )
                {
                        if (node == root[run_p] )
                                                /* height of root is 0 */
                                node->height=0;
                        else
                                                /* height of parent must already computed */
                                node->height=(node->parent)->height+1;
                }
                if ( (node->right == NULL) && ( node->left == NULL ))
                {                               /* node is leaf */
                        if ( New_method == 3 )
                        {
                                if ( High_leaf[run_p] == NULL )
                                        High_leaf[run_p]=node;
                                else
                                {
                                                /* get the leaf which has the fartest
                                                   height */
                                    if (High_leaf[run_p]->height < node->height)
                                        High_leaf[run_p]=node;
                                }
                        }
                        else if (New_method == 4 )
                        {
                                                /* check if the leaf was also a leaf
                                                   in the previous state */
                            tmp=lqhead[run_p];
                             exist=NO;
                            while(tmp != NULL)
                            {
                                if(node->page_num == (tmp->leaf)->page_num)
                                {
                                                /* this leaf was also a leaf in previous
                                                   state */
                                        exist=YES;
                                        tmp->e_flag=ON;
                                        break;
                                }
                                else
                                        tmp=tmp->next;
                            }
                            if(exist==NO)
                            {
                                                /* this new leaf is linked to the tail
```

```
                                                    of the leaf queue */
                                newlq=(LEAF_L *)malloc(sizeof(LEAF_L));
                                newlq->leaf=node;
                                newlq->e_flag=ON;
                                newlq->next=NULL;
                                lqtail[run_p]->next=newlq;
                                lqtail[run_p]=newlq;
                            }
                        }
                }
                GetHeight(node->left,run_p);  /* recursive */
                GetHeight(node->right,run_p);
        }
}

/*////////////////////////////////////////////////////////////////////////////////////////
//      Function : RemovePage()
//      Purpose  : This function is used to remove the victim page from the tree when
//                 memory is full and a page fault occurs.
////////////////////////////////////////////////////////////////////////////////////////*/

void RemovePage(int run_p)
{
        PAGE_TABLE1 *top;
        PAGE_TABLE1 *old;

        top=root[run_p];
        old=FindOldPage(top,run_p);             /* find the victim page */
        if(old == (old->parent)->right)         /* remove the victim page */
                (old->parent)->right=NULL;
        else if(old == (old->parent)->left)
                (old->parent)->left=NULL;
        No_in_tree[run_p]--;
        free(old);
}

/*////////////////////////////////////////////////////////////////////////////////////////
//      Function : FindOldPage()
//      Purpose  : This function is used to find the victim page.
////////////////////////////////////////////////////////////////////////////////////////*/

PAGE_TABLE1 *FindOldPage(PAGE_TABLE1 *top,int run_p)
{
        PAGE_TABLE1 *node;
        int     i,max_height;

        node=top;
        if ( New_method == 1 )                  /* leftmost leaf method */
        {
                while( (node->left != NULL) || (node->right != NULL) )
                {                               /* find the leftmost leaf in the tree */
                        if( node->left == NULL)
                                                /* go to right subtree */
                                node=node->right;
                        else
                                                /* go to left subtree */
                                node=node->left;
                }
                return(node);
        }
        else if (New_method ==2 )               /*rightmost leaf method */
        {
                while( (node->left != NULL) || (node->right != NULL) )
                {
                                                /* find the rightmost leaf */
                        if( node->right == NULL)
                                                /* go to the left subtree */
```

```
                        node=node->left;
                else                    /* go to the right subtree */
                        node=node->right;
            }
            return(node);

        }
        else if (New_method == 3 )              /* highest leaf method  */
                return(High_leaf[run_p]);
        else if (New_method == 4)               /* LRU leaf method */
        {
                                                /* the head of the queue is a victim page */
                node=lqhead[run_p]->leaf;
                lqhead[run_p]=lqhead[run_p]->next;
                return(node);
        }
}

/*//////////////////////////////////////////////////////////////////////////////////////////
//      Function : GetLeafQue()
//      Purpose  : This function is used to get the current leaves. Among those
//                 previous leaves, the leaves which are not the current leaves
//                 are removed from the leaf queue.
//////////////////////////////////////////////////////////////////////////////////////////*/

void GetLeafQue(int run_p)
{
        LEAF_L *tmp,*lqtmp,*prev;
        int exist;

        lqtmp=lqhead[run_p];
        while(lqtmp != NULL )
        {
                if (lqtmp->e_flag == OFF )      /* entry is not a leaf */
                {
                        if(lqtmp == lqhead[run_p])
                        {                       /* entry is the head of the queue */
                                tmp=lqhead[run_p];
                                lqhead[run_p]=lqhead[run_p]->next;
                                lqtmp=lqtmp->next;
                                free(tmp);
                        }
                        else
                        {
                                prev->next=lqtmp->next;
                                tmp=lqtmp;
                                lqtmp=lqtmp->next;
                                                /* entry is the tail of the queue */
                                if(tmp == lqtail[run_p])
                                        lqtail[run_p]=prev;
                                free(tmp);
                        }
                }
                else
                {
                        lqtmp->e_flag=OFF;
                        prev=lqtmp;
                        lqtmp=lqtmp->next;
                }
        }
}

/*//////////////////////////////////////////////////////////////////////////////////////////
//
//                              Ckago.c
//
// This file is to simulate a clock algorithm. A circular queue was used to contain the
```

```
// pages in main memory. Hand pointer indicates the oldest page which was referenced. The
// reference bit of each page is cleaned after a certain time interval. User can select
// the time interval.
/////////////////////////////////////////////////////////////////////////////////////*/

#include "myhead.h"

void AddNewPageCl(int page,int run_p);
void ReplacePageCl(int page,int run_p);
void SetRefBit(int run_p);
int  ClockAlg(int page,int run_p);
int  SearchCircularQ(int page,int run_p);

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function : ClockAlg()
//      Purpose  : This is main function of Ckago.c file. It checks if the page needed
//                 immediately is in main memory with searching the circular queue. If the
//                 page is not in main memory, and main memory is not full, then insert
//                 the page in the queue. If the main memory is full, replace the victim
//                 page with the page needed soon. If a page fault is occurred, return
//                 YES, else return NO.
/////////////////////////////////////////////////////////////////////////////////////*/

int ClockAlg(int page,int run_p)
{
        int     exist;

        if ( cqhead[run_p} != NULL )
                SetRefBit(run_p);
        exist=SearchCircularQ(page,run_p);
        if( exist == YES )
                return(NO);                     /* no page_fault occurs */
        else
        {
                if(Frame[run_p] == No_in_cq[run_p] )
                                                /* when memory is full, a victim page must
                                                   replaced */
                        ReplacePageCl(page,run_p);
                else                            /* add the new page to page table */
                        AddNewPageCl(page,run_p);
                return(YES);                    /* page fault occurred */
        }
}

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function : ReplacePageCl()
//      Purpose  : This is used to replace a victim page with the page needed immediately.
//                 When a victim page is chosen, if the reference bit of the page which
//                 Hand pointer indicates is OFF, the page is victim page. Else, Hand
//                 pointer advances until a reference bit of a page is OFF. The page is
//                 replaced with the new page.
/////////////////////////////////////////////////////////////////////////////////////*/

void ReplacePageCl(int page,int run_p)
{
        PAGE_TABLE2 *new,*tmp;

                                                /* search the victim page */
        while(Hand[run_p]->rbit != OFF )
        {
                Hand[run_p]->rbit=OFF;
                before[run_p]=Hand[run_p];
                Hand[run_p]=Hand[run_p]->next;
        }

                                                /* create an entry for a new page */
        new=(PAGE_TABLE2 *)malloc(sizeof(PAGE_TABLE2));
```

```
        new->rbit=ON;
        new->page_num=page;

                                        /* replace the victim page with a new page */
        new->next=Hand[run_p]->next;
        tmp=Hand[run_p];
        before[run_p]->next=new;
        if ( Hand[run_p] == cqhead[run_p] )
                cqhead[run_p]=new;
        Hand[run_p]=new->next;
        before[run_p]=new;
        free(tmp);
}

/*////////////////////////////////////////////////////////////////////////////////////////
//      Function : SearchCircularQ()
//      Purpose  : This is used to check if the page to be referenced is in the circular
//                 queue or not. If the page is in there, the reference bit of this page
//                 is set, and return YES. If not,return NO.
////////////////////////////////////////////////////////////////////////////////////////*/

int SearchCircularQ(int page,int run_p)
{
        PAGE_TABLE2 *tmp;
        int     exist;

        exist=NO;
        tmp=cqhead[run_p];
        if ( tmp == NULL )
                return(NO);                     /* no page exist in main memory */
        else
        {
                if(tmp->page_num == page )      /* check header of circular queue */
                 {
                        tmp->rbit=ON;           /* set the reference bit */
                        exist=YES;
                        return(YES);
                }
                tmp=tmp->next;
                while(tmp != cqhead[run_p] )
                {
                        if(tmp->page_num == page )
                        {                       /* referenced page is in the main memory */
                                tmp->rbit=ON;   /* set the reference bit */
                                exist=YES;
                                return(YES);
                        }
                        else
                                tmp=tmp->next; /* search the next entry */
                }
        }
        return(NO);                             /* page is not in curcular queue */
}

/*////////////////////////////////////////////////////////////////////////////////////////
//      Function : SetRefBit()
//      Purpose  : This is used to clear the reference bit of each page after a certain
//                 time interval.
////////////////////////////////////////////////////////////////////////////////////////*/

void SetRefBit(int run_p)
{
        PAGE_TABLE2     *temp;

        temp=cqhead[run_p];
        if ( interval == 1 )
        {
```

```
                temp->rbit=OFF;
                temp=temp->next;
                while(temp != cqhead[run_p] )
                {
                        temp->rbit=OFF;         /* clear reference bit */
                        temp=temp->next;
                }
        }
        else
        {
                        if( (CLOCK%interval) == 1 )
                        {                       /* clear the reference bit after a certain
                                                   time interval */
                                temp->rbit=OFF;
                                temp=temp->next;
                                while(temp != cqhead[run_p] )
                                {
                                        temp->rbit=OFF;
                                        temp=temp->next;
                                }
                        }
                }
}

/*///////////////////////////////////////////////////////////////////////////////////
//      Function : AddNewPageCl()
//      Purpose  : This is used to add the new entry containing the immediately needed
//                 page to the circular queue.
///////////////////////////////////////////////////////////////////////////////////*/

void AddNewPageCl(int page,int run_p)
{
        PAGE_TABLE2 *H,*new;

                                                /* create a new entry */
        new=(PAGE_TABLE2 *)malloc(sizeof(PAGE_TABLE2));
        new->page_num=page;
        new->rbit=ON;
        H=cqhead[run_p];
        if( H == NULL )                         /* circular queue is empty */
        {
                cqhead[run_p]=new;
                cqhead[run_p]->next=cqhead[run_p];
                Hand[run_p]=cqhead[run_p];
                before[run_p]=Hand[run_p];
                No_in_cq[run_p]=1;
        }
        else
        {
                                                /* insert to the tail of circular queue */
                while(H->next != cqhead[run_p])
                        H=H->next;
                new->next=H->next;
                H->next=new;
                No_in_cq[run_p] = No_in_cq[run_p]+1;
                if ( new->next == Hand[run_p])
                        before[run_p]=new;
        }
        H=cqhead[run_p];
}

/*///////////////////////////////////////////////////////////////////////////////////
//
//                              Addref.c
//
//
// This file is to implement additional-reference-bits algorithm. It can get the ordering
```

```
// of page references by keeping a 8-bit shift register in each entry of page table. The
// shift register records the reference bit for each page at each time interval. At each
// clock interrupt, the shift register is shifted right 1 bit and the current reference
// bits are inserted as the leftmost bit. The page with the lowest value in the shift
// register is least recently used page. When memory is full and a page fault is occurs,
// the page is replaced. If a number of pages have the same lowest value, this program
// chooses the page which is a frontier in the page table.
/////////////////////////////////////////////////////////////////////////////////////*/

#include "myhead.h"

                                        /* functions used in this file */
int AdditionalRefAlg(int page,int run_p);
void AddNewPageAdd(int page,int run_p);
PAGE_TABLE3 *GetSmallValue(int run_p);
void RemovePageAdd(struct add_ref *remove,int run_p);

/*///////////////////////////////////////////////////////////////////////////////////////
//      Function : AdditionalRefAlg()
//      Purpose  : This function is main function of Addref.c file. First, it checks if
//                 the page which to be referenced is in main memory or not. If the page
//                 doesn't exist, add to the page table. When memory is full, a LRU page
//                 which has a lowest value is replaced with a new page. If a page fault
//                 is occured, return YES, else return NO.
/////////////////////////////////////////////////////////////////////////////////////*/

int AdditionalRefAlg(int page,int run_p)
{
        PAGE_TABLE3 *tmp,*remove;
        int     f_exist;

        tmp=Add_table[run_p]->next;
        f_exist=NO;
        if (tmp == NULL )                       /* no page is in memory */
        {
                tmp=(struct add_ref *)malloc(sizeof(struct add_ref));
                tmp->page_num=page;
                tmp->shift_reg.value=0;
                tmp->shift_reg.reg.first=ON;
                tmp->next=NULL;
                Add_table[run_p]->next=tmp;
                add_tail[run_p]=tmp;
                Add_table[run_p]->num=1;
                return(YES);                    /* page fault occurs */
        }
        else
        {
                while(tmp!=NULL)
                {
                                                /* clear reference bit after the time
                                                   interval */
                        if (interval == 1 )
                        {                       /* time interval is 1 */
                                tmp->shift_reg.value=tmp->shift_reg.value>>1;
                                tmp->shift_reg.reg.first=OFF;
                        }
                        else
                        {
                                if( (CLOCK%interval)==1)
                                {
                                   tmp->shift_reg.value=tmp->shift_reg.value>>1;
                                   tmp->shift_reg.reg.first=OFF;
                                }
                        }
                        tmp=tmp->next;
                }
                tmp=Add_table[run_p]->next;
```

```
		while(tmp!=NULL)
		{
			if (tmp->page_num == page)
			{				/* page is in memory */
							/* set the leftmost bit of shift register */
				tmp->shift_reg.reg.first=ON;
				f_exist=YES;
				return(NO);
			}
			tmp=tmp->next;
		}
		if (f_exist != YES )
		{
							/* page fault occurs */
			if(Add_table[run_p]->num==Frame[run_p])
			{
				/* Memory is full */
				remove=GetSmallValue(run_p);
				RemovePageAdd(remove,run_p);
				AddNewPageAdd(page,run_p);
				return(YES);
			}
			else
			{
				AddNewPageAdd(page,run_p);
				return(YES);
			}
		}

	}
}

/*////////////////////////////////////////////////////////////////////////////////////
//	Function : AddNewPageAdd()
//	Purpose  : This function is used to insert a new page to the page table. A
//		   reference bit is inserted into the leftmost bit of shift register.
////////////////////////////////////////////////////////////////////////////////////*/

void AddNewPageAdd(int page,int run_p)
{
	PAGE_TABLE3 *tmp;

						/* create a new entry */
	tmp=(struct add_ref *)malloc(sizeof(struct add_ref));
	tmp->page_num=page;
	tmp->shift_reg.value=0;
	tmp->shift_reg.reg.first=ON;		/* register's leftmost bit is ON */
	tmp->next=NULL;

						/* insert to page table */
	add_tail[run_p]->next=tmp;
	add_tail[run_p]=tmp;
	Add_table[run_p]->num++;
}

/*////////////////////////////////////////////////////////////////////////////////////
//	Function : GetSmallValue()
//	Purpose  : This function is used to get the page which has the smallest value of
//		   the shift register. It will be a victim page.
////////////////////////////////////////////////////////////////////////////////////*/

PAGE_TABLE3 *GetSmallValue(int run_p)
{
	struct add_ref *tmp1,*smallest;

	smallest=Add_table[run_p]->next;
	tmp1=smallest->next;
```

```
        while(tmp1 != NULL )
        {                                       /* search has smallest value */
                if(smallest->shift_reg.value > tmp1->shift_reg.value)
                                                /* compare the value */
                        smallest=tmp1;
                tmp1=tmp1->next;                /* check the next entry */
        }
        return(smallest);                       /* return the smallest value */
}

/*/////////////////////////////////////////////////////////////////////////////////////////
//      Function : RemovePageAdd()
//      Purpose  : This function is used to remove the victim page from the page table.
/////////////////////////////////////////////////////////////////////////////////////////*/

void RemovePageAdd(struct add_ref *remove,int run_p)
{
        PAGE_TABLE3 *tmp,*before;

        tmp=Add_table[run_p]->next;
        if (remove->page_num == tmp->page_num )
        {
                                                /* victim page is first element of linked
                                                   list */
                                                /* remove the victim page */
                Add_table[run_p]->next=tmp->next;
                Add_table[run_p]->num--;
                free(tmp);
        }
        else
        {
                before=tmp;
                tmp=tmp->next;
                while(tmp != NULL )
                {                               /* search the node which has the samllest
                                                   value of 8 bit shift register */
                        if(remove->page_num == tmp->page_num)
                        {
                                                /* remove the victim page */
                                before->next=tmp->next;
                                free(tmp);
                                Add_table[run_p]->num--;
                                break;
                        }
                        else
                        {                       /* search the next entry */
                                before=tmp;
                                tmp=tmp->next;
                        }
                }
        }
}

/*/////////////////////////////////////////////////////////////////////////////////////////
//
//                              Graph.c
//
// This file is to generate and to show the graphs using the blt_graph command. The
// blt_graph is to create and to manipulate graph widgets. The blt_graph widget plots two
// variable data in a window. When we see the graphs, graphs have two push buttons, one is
// <print> (to make postscript file) and <quit> (to exit the graph display). The number of
// page faults and memory utilization of each process in each page replacement algorithm
// is viewed on the screen.
/////////////////////////////////////////////////////////////////////////////////////////*/

#include "myhead.h"
```

```
/*//////////////////////////////////////////////////////////////////////////////
//      Function : PageFaultGraph()
//      Purpose  : This function is used to choose a graph which will show the number of
//                 page faults according to one comparison basis.
//////////////////////////////////////////////////////////////////////////////*/

PageFaultGraph()
{
        int     re;
        int     valid;
        char    buf[81];

        valid=0;
        while(valid != 1)
        {
                printf("\n\t ------ Page fault graph ------------------------------ \n");
                printf("\t 1. Page faults of 3 different algorithms. \n");
                printf("\t        - New implementation (highest leaf method).\n");
                printf("\t        - Clock algorithm.\n");
                printf("\t        - Additional-reference-bits algorithm.\n");
                printf("\t 2. Page faults of 4 different methods in the new
                        implementation.\n");
                printf("\t        - Leftmost leaf method.\n");
                printf("\t        - Rightmost leaf method.\n");
                printf("\t        - Highest leaf method.\n");
                printf("\t        - LRU leaf method.\n");
                printf("\t 3. Page faults of 3 different intervals in the clock algorithm.
                        \n");
                printf("\t 4. Page faults of 3 different intervals in the additional\n");
                printf("\t    -reference-bits algorithm.\n");
                printf("\t 5. Page faults vs. Frames allocated.\n");
                printf("\t 6. Page faults vs. Regular time intervals in the clock
                        algorithm.\n");
                printf("\t 7. Page faults vs. Regular time intervals in the additional\n");
                printf("\t    -reference-bits algorithm.\n");
                printf("\t 8. Exit (return to menu). \n");
                printf("Select a number: ");
                scanf("%d",&re);

                switch(re)
                {
                        case 1: PageFaultGraph1();
                                        /* graph for page faults over
                                           different algorithm */
                                valid=1;
                                break;
                        case 2:  PageFaultGraph2();
                                        /* page faults for new implementation */
                                valid=1;
                                break;
                        case 3:  PageFaultGraph3(re);
                                        /* page faults for clock algorithm */
                                valid=1;
                                break;
                        case 4:  PageFaultGraph3(re);
                                        /* page faults for additaional-reference-bits
                                           algorithm */
                                valid=1;
                                break;
                        case 5:  PageFaultGraph5();
                                        /* page faults vs. # of frames allocated */
                                valid=1;
                                break;
                        case 6:  PageFaultGraph6();
                                        /* page faults vs. time interval for clock
                                           algorithm */
                                valid=1;
```

```
                        break;
                case 7:  PageFaultGraph7();
                                        /* page faults vs. time intervals for
                                           additional-reference-bits algorithm */
                        valid=1;
                        break;
                case 8: return;
                default: printf("\n\tError: Invalid Input, try again.\n");
                        gets(buf);
                        valid=0;
                        break;
                }
        }
}

/*/////////////////////////////////////////////////////////////////////////////////////
//      Function : MemUtilGraph()
//      Purpose  : This function is used to choose a graph which will show the memory
//                 occupancy according to one comparison basis.
/////////////////////////////////////////////////////////////////////////////////////*/

MemUtilGraph()
{
        int     re;
        int     valid;
        char    buf[81];

        valid=0;
        while(valid != 1)
        {
                printf("\n\t ------ Memory utilization graph ---------------------- \n");
                printf("\t 1. Memory utilization in the 3 different algorithms. \n");
                printf("\t       - New implementation (highest leaf method).\n");
                printf("\t       - Clock algorithm.\n");
                printf("\t       - Additional-reference-bits algorithm.\n");
                printf("\t 2. Memory utilization of 4 different methods in the new
                        implementation.\n");
                printf("\t       - Leftmost leaf method.\n");
                printf("\t       - Rightmost leaf method.\n");
                printf("\t       - Highest leaf method.\n");
                printf("\t       - LRU leaf method.\n");
                printf("\t 3. Memory utilization of 3 different intervals in the clock
                        algorithm.\n");
                printf("\t 4. Memory utilization of 3 different intervals in the
                        additional\n");
                printf("\t     -reference-bits algorithm.\n");
                printf("\t 5. Exit (return to menu). \n ");
                printf("Select a number: ");
                scanf("%d",&re);

                switch(re)
                {
                        case 1: MemUtilGraph1();
                                        /* memory utilization over different
                                           algorithms */
                                valid=1;
                                break;
                        case 2:  MemUtilGraph2();
                                        /* memory utilization for the new
                                           implementation */
                                valid=1;
                                break;
                        case 3:  MemUtilGraph3(re);
                                        /* memory utilization for the clock
                                           algorithm */
                                valid=1;
                                break;
```

```
                case 4:  MemUtilGraph3(re);
                                        /* memory utilization for the additional-
                                           reference-bits algorithm */
                        valid=1;
                        break;
                case 5: return;
                default: printf("\n\tError: Invalid input, try again. \n");
                        valid=0;
                        gets(buf);
                        break;
            }
        }
}

/*//////////////////////////////////////////////////////////////////////////////////////
//      Function : PageFaultGraph1()
//      Purpose  : This function is used to generate the page fault graph over 3 different
//                 algorithms. x-vaules are the process IDs and y-vaules are number of
//                 page faults of each process.
//////////////////////////////////////////////////////////////////////////////////////*/

PageFaultGraph1()
{
        FILE    *in;
        int     i;
        char    svalue[81];
        float   value_y1[MAX_PROCESS];          /* y values related to new implementation */
        float   value_y2[MAX_PROCESS];          /* y values related to clock algorithm */
        float   value_y3[MAX_PROCESS];          /* y values related to additional-reference-
                                                   bits algorithm */
        float   value_x[MAX_PROCESS];           /* x values */
        char    psfile[81];                     /* name of postscript file */
        char    s[81];
        char    sp_size[81];
        char    sm_size[81];
        int     p_size,mem_size;
        int     cinterval;                      /* time interval for clock algorithm */
        int     ainterval;                      /* time interval for additional-reference-
                                                   bits algorithm */
        int     valid;                          /* indicate that input parameters are valid
                                                */

        No_process=5;
        memset(psfile,81,NULL);
        in=fopen("Pageflt.grph","w");
        printf("\nPut the name of the postscript file: ");
        scanf("%s",psfile);                     /* get the name of postsrcipt file */

        valid=0;
        while(valid != 1 )
        {
                printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
                scanf("%s",sp_size);            /* get the page size */
                p_size=atoi(sp_size);
                if ( (p_size == 512 ) || (p_size == 1024) ||
                     (p_size == 2048) || (p_size == 4096) ||
                     (p_size == 8192) )         /* input is valid */
                        valid =1;
                else                            /* input is not valid */
                {
                        valid=0;
                        printf("\n\tError: Invalid input, try again\n");
                }
        }

        valid=0;
        while(valid != 1)
```

```
{
        printf("\nPut the memory size: ");
        scanf("%s",sm_size);            /* get the memory size */
        mem_size=atoi(sm_size);
        if (mem_size == 0)
        {
                printf("\n\tError: Invalid input, try again.\n");
                valid=0;
        }
        else
                valid=1;
}

valid=0;
while(valid != 1)
{
        printf("\nPut the interval for the clock algorithm: ");
        scanf("%s",svalue);             /* get the time interval for clock algorithm
                                        */
        cinterval=atoi(svalue);
        if (cinterval == 0)
        {
                printf("\n\tError: Invalid input, try again.\n");
                valid=0;
        }
        else
                valid=1;
}

valid=0;
while(valid != 1)
{
        printf("\nPut the interval for the additional-reference-bits algorithm: ");
        scanf("%s",svalue);             /* get the time interval for additional-
                                           reference-bits algorithm */
        ainterval=atoi(svalue);
        if ( ainterval ==0 )
        {
                printf("\n\tError: Invalid input, try again.\n");
                valid=0;
        }
        else
                valid=1;
}

                                        /* get x and y values */
for(i=0; i<No_process; i++)
        value_x[i]=i+1;                 /* x values are process IDs */
printf("\n\t x values are process IDs and y values are # of page faults\n");
printf("\t (Note: If you want exit, put <quit>.) \n");

for(i=0; i<No_process; i++)
{
        printf("\n Put the y1 value (New Implementation): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y1[i]=atof(svalue);
                                        /* y values are # of page faults for
                                           new implementation */
}

for(i=0; i<No_process; i++)
{
        printf("\n Put the y2 value (Clock Algorithm): ");
        scanf("%s",svalue);
```

```
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y2[i]=atof(svalue);
                                        /* y values are # of page faults for
                                           clock algorithm */
}

for(i=0; i<No_process; i++)
{
        printf("\n Put the y3 value (Additional-reference-bits Algorithm): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y3[i]=atof(svalue);
                                        /* y values are # of page faults for
                                           additional-reference-bits algorithm */
}

fprintf(in,"#!/contrib/bin/blt_wish -f\n");
fprintf(in,"\n");
fprintf(in,"if [file exists /contrib/library] {\n");
fprintf(in,"    set blt_library /contrib/library\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
                                        /* set the font and size */
fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
fprintf(in,"option add *graph.xTitle %cProcess ID %c \n",34,34);
                                        /* title of x axis */
fprintf(in,"option add *graph.yTitle %cNumber of page faults %c \n",34,34);
                                        /* title of y axis */
fprintf(in,"option add *graph.title %cPage faults over different algorithms (%d /
        %d) %c \n",34,mem_size,p_size,34);
                                        /* title of graph */
fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
strcpy(s,"Number of page faults over different algorithms. ");
RepeatBodyGraph1(in,psfile,s);

fprintf(in," set X {\n");               /* write x values to file for graph */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_x[i]);
fprintf(in,"\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in," set Y1 {\n");              /* write y values to file for graph */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y1[i]);
fprintf(in,"\n");
fprintf(in,"}\n");

fprintf(in,"set Y2 {\n");               /* write y values to file for graph */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y2[i]);
fprintf(in,"\n");
fprintf(in,"}\n");

fprintf(in,"set Y3 {\n");               /* write y values to file for graph */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y3[i]);
fprintf(in,"\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"\n");

fprintf(in,"$graph element create Highest-leaf -xdata $X -ydata $Y1 %c\n",92);
```

```
        fprintf(in,"      -symbol diamond -linewidth 0\n");
        fprintf(in,"$graph element create Clock(%d) -xdata $X -ydata $Y2
                %c\n",cinterval,92);
        fprintf(in,"      -symbol cross -linewidth 0\n");
        fprintf(in,"$graph element create Add-ref-bits(%d) -xdata $X -ydata $Y3
                %c\n",ainterval,92);
        fprintf(in,"      -symbol square -linewidth 0\n");
        RepeatBodyGraph2(in);
        fclose(in);

        system("chmod 777 Pageflt.grph");
        system("Pageflt.grph");                 /* show the graph */
        return;
}

/*////////////////////////////////////////////////////////////////////////////////////////
//      Function : PageFaultGraph2()
//      Purpose  : This function is used to generate the page fault graph over 4 different
//                 methods of the new implementation.
////////////////////////////////////////////////////////////////////////////////////////*/

PageFaultGraph2()
{
        FILE    *in;
        int     i;
        float   value_y1[MAX_PROCESS];          /* y values for leftmost leaf method */
        float   value_y2[MAX_PROCESS];          /* y values for rightmost leaf method */
        float   value_y3[MAX_PROCESS];          /* y values for highest leaf method */
        float   value_y4[MAX_PROCESS];          /* y values for LRU leaf method */
        float   value_x[MAX_PROCESS];           /* x values */
        char    psfile[81];                     /* name of postscript file */
        char    s[81];
        char    svalue[81];
        int     p_size,m_size;
        int     valid;                          /* indicates that input is valid */

        No_process=5;
        memset(psfile,81,NULL);
        in=fopen("Pageflt2.grph","w");
        printf("\nPut the name of the postscript file: ");
        scanf("%s",psfile);                     /* get the name of postscipt file */

        valid=0;
        while(valid != 1 )
        {
                printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
                scanf("%s",svalue);             /* get the page size */
                p_size=atoi(svalue);
                if ( (p_size == 512 ) || (p_size == 1024) ||
                     (p_size == 2048) || (p_size == 4096) ||
                     (p_size == 8192) )
                        valid =1;
                else
                {
                        valid=0;
                        printf("\n\tError: Invalid input, try again.\n");
                }
        }

        valid=0;
        while(valid != 1)
        {
                printf("\nPut the size of memory: ");
                scanf("%s",svalue);             /* get the memory size */
                m_size=atoi(svalue);
                if ( m_size == 0 )              /* input is not valid */
                {
```

```
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

                                        /* get x and y values */
for(i=0; i<No_process; i++)
        value_x[i]=i+1;                 /* x values are process IDs */
printf("\n\t x values are process IDs and y values are # of page faults\n");
printf("\t (Note: If you want exit, put <quit>.) \n");
for(i=0; i<No_process; i++)
{
        printf("\n Put the y1 value (Leftmost leaf) : ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y1[i]=atof(svalue);
                                        /* y values for leftmost leaf method */
}

for(i=0; i<No_process; i++)
{
        printf("\n Put the y2 value (Rightmost leaf) : ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y2[i]=atof(svalue);
                                        /* y values for rightmost leaf method */
}
for(i=0; i<No_process; i++)
{
        printf("\n Put the y3 value (Highest leaf) : ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y3[i]=atof(svalue);
                                        /* y values for highest leaf method */
}
for(i=0; i<No_process; i++)
{
        printf("\n Put the y4 value (LRU leaf) : ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y4[i]=atof(svalue);
                                        /* y values for LRU leaf method */
}

fprintf(in,"#!/contrib/bin/blt_wish -f\n");
fprintf(in,"\n");
fprintf(in,"if [file exists /contrib/library] {\n");
fprintf(in,"    set blt_library /contrib/library\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
                                        /* set the font and size */
fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
fprintf(in,"option add *graph.xTitle %cProcess ID %c \n",34,34);
                                        /* title of x axis */
fprintf(in,"option add *graph.yTitle %cNumber of page faults %c \n",34,34);
                                        /* title of y axis */
```

```
        fprintf(in,"option add *graph.title %cPage faults for new implementation (%d /
        %d)%c \n",34,m_size,p_size,34);         /* title of graph */

        fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
        strcpy(s,"Number of page faults for new implemantataion.");
        RepeatBodyGraph1(in,psfile,s);

        fprintf(in," set X {\n");
                for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_x[i]); /* write x values to the file */
        fprintf(in,"\n");
        fprintf(in,"}\n");
        fprintf(in,"\n");
        fprintf(in," set Y1 {\n");              /* write y values for leftmost leaf method to
                                                   file */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y1[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");

        fprintf(in,"set Y2 {\n");               /* write y values for rightmost leaf method
                                                   to file */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y2[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");

        fprintf(in,"set Y3 {\n");               /* write y values for highest leaf method */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y3[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");

        fprintf(in,"set Y4 {\n");               /* write y values for LRU leaf method */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y3[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");
        fprintf(in,"\n");
        fprintf(in,"\n");

        fprintf(in,"$graph element create Leftmost -xdata $X -ydata $Y1 %c\n",92);
        fprintf(in,"       -symbol plus -linewidth 0\n");
        fprintf(in,"$graph element create Rightmost -xdata $X -ydata $Y2 %c\n",92);
        fprintf(in,"       -symbol cross -linewidth 0\n");
        fprintf(in,"$graph element create Highest -xdata $X -ydata $Y3 %c\n",92);
        fprintf(in,"       -symbol square -linewidth 0\n");
        fprintf(in,"$graph element create LRU-leaf -xdata $X -ydata $Y4 %c\n",92);
        fprintf(in,"       -symbol diamond -linewidth 0\n");
        RepeatBodyGraph2(in);
        fclose(in);

        system("chmod 777 Pageflt2.grph");
        system("Pageflt2.grph");                /* show the graph */
        return;
}

/*//////////////////////////////////////////////////////////////////////////////////////
//      Function : PageFaultGraph3()
//      Purpose  : This function is used to generate the page fault graph over 3 different
//                 intervals in clock or additional-reference-bits algorithms.
//////////////////////////////////////////////////////////////////////////////////////*/

PageFaultGraph3(int sel)
{
        FILE    *in;                            /* file descriptor of file for graph */
        int     i;
```

```
float   value_y1[MAX_PROCESS];          /* y values */
float   value_y2[MAX_PROCESS];
float   value_y3[MAX_PROCESS];
float   value_x[MAX_PROCESS];           /* x values */
char    psfile[81];                     /* name of postscript file */
char    svalue[81];
char    s[81];
int     p_size,m_size;
int     valid;

No_process=5;
memset(psfile,81,NULL);
if (sel == 3 )
        in=fopen("Pageflt3.grph","w");
else if (sel == 4 )
        in=fopen("Pageflt4.grph","w");
printf("\nPut the name of the postscript file: ");
scanf("%s",psfile);                     /* get the name of postscript file */

valid=0;
while(valid != 1 )
{
        printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
        scanf("%s",svalue);             /* get the page size */
        p_size=atoi(svalue);
        if ( (p_size == 512 ) || (p_size == 1024) ||
             (p_size == 2048) || (p_size == 4096) ||
             (p_size == 8192) )
                valid =1;
        else
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
}

valid=0;
while(valid != 1)
{
        printf("\nPut the size of memory: ");
        scanf("%s",svalue);             /* get the memory size */
        m_size=atoi(svalue);
        if ( m_size == 0 )
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

for(i=0; i<No_process; i++)
        value_x[i]=i+1;                 /* get the x values */
printf("\n\t x values are process IDs and y values are # of page faults\n");
printf("\t (Note: If you want exit, put <quit>.) \n");
for(i=0; i<No_process; i++)
{
        if ( sel == 3 )
                printf("\n Put the y1 value (interval 16800) : ");
        if ( sel == 4 )
                printf("\n Put the y1 value (interval 70000) : ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y1[i]=atof(svalue);
                                        /* get the y values */
```

```
}

for(i=0; i<No_process; i++)
{
        if ( sel == 3 )
               printf("\n Put the y2 value (interval 28000) : ");
        if ( sel == 4 )
               printf("\n Put the y2 value (interval 140000) : ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
               return;
        else
               value_y2[i]=atof(svalue);
                                        /* get the y values */
}

for(i=0; i<No_process; i++)
{
        if (sel ==3 )
               printf("\n Put the y3 value (interval 39200) : ");
        if (sel ==4 )
               printf("\n Put the y3 value (interval 210000) : ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
               return;
        else
               value_y3[i]=atof(svalue);
                                        /* get the y values */
}

fprintf(in,"#!/contrib/bin/blt_wish -f\n");
fprintf(in,"\n");
fprintf(in,"if [file exists /contrib/library] {\n");
fprintf(in,"   set blt_library /contrib/library\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
                                        /* set the font and size */
fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
fprintf(in,"option add *graph.xTitle %cProcess ID %c \n",34,34);
                                        /* title of x axis */
fprintf(in,"option add *graph.yTitle %cNumber of page faults %c \n",34,34);
                                        /* title of y axis */
if ( sel == 3 )
        fprintf(in,"option add *graph.title %cPage faults for clock algorithm (%d /
        %d)%c \n",34,m_size,p_size,34);
                                        /* title of graph */
else if (sel ==4 )
        fprintf(in,"option add *graph.title %cPage faults for add-ref-bits
        algorithm (%d / %d)%c \n",34,m_size,p_size,34);
                                        /* title of graph */
fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
if ( sel == 3 )
        strcpy(s,"Number of page faults for clock algorithm");
else if (sel ==4 )
        strcpy(s,"Number of page faults for add-ref-bits algorithm");
RepeatBodyGraph1(in,psfile,s);
fprintf(in," set X {\n");               /* write x values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_x[i]);
fprintf(in,"\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in," set Y1 {\n");              /* write y values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y1[i]);
fprintf(in,"\n");
```

```
        fprintf(in,"}\n");

        fprintf(in,"set Y2 {\n");               /* write y values to the file */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y2[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");

        fprintf(in,"set Y3 {\n");               /* write y values to the file */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y3[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");
        fprintf(in,"\n");
        fprintf(in,"\n");

        if (sel ==3)
                fprintf(in,"$graph element create Interval-16800 -xdata $X -ydata $Y1
                %c\n",92);
        if (sel ==4)
                fprintf(in,"$graph element create Interval-70000 -xdata $X -ydata $Y1
                %c\n",92);
        fprintf(in,"      -symbol diamond -linewidth 0\n");
        if (sel == 3 )
                fprintf(in,"$graph element create Interval-28000 -xdata $X -ydata $Y2
                %c\n",92);
        if (sel ==4)
                fprintf(in,"$graph element create Interval-140000 -xdata $X -ydata $Y2
                %c\n",92);
        fprintf(in,"      -symbol cross -linewidth 0\n");
        if (sel == 3 )
                fprintf(in,"$graph element create Interval-39200 -xdata $X -ydata $Y3
                %c\n",92);
        if (sel == 4 )
                fprintf(in,"$graph element create Interval-210000 -xdata $X -ydata $Y3
                %c\n",92);
        fprintf(in,"      -symbol square -linewidth 0\n");
        RepeatBodyGraph2(in);
        fclose(in);

        if (sel == 3)                           /* for clock algorithm */
        {
                system("chmod 777 Pageflt3.grph");
                system("Pageflt3.grph");        /* generate the graph */
        }
        else if (sel == 4 )                     /* for additional-reference-bits algorithm */
        {
                system("chmod 777 Pageflt4.grph");
                system("Pageflt4.grph");        /* generate the graph */
        }
        return;
}

/*//////////////////////////////////////////////////////////////////////////////
//      Function : PageFaultGraph5()
//      Purpose  : This function is used to generate the page fault graph for a process
//                 when the number of frames allocated are increased.
//////////////////////////////////////////////////////////////////////////////*/

PageFaultGraph5()
{
        FILE    *in;
        int     i;
        float   value_x[100];                   /* x values are # of frames allocated */
        float   value_y1[100];                  /* y values are # of page faults */
        char    psfile[81];                     /* name of postscript file */
        char    s[81];
```

```
int     n,pnum;
char    methd[81];                  /* algorithm or method */
char    svalue[81];
int     p_size;
int     valid;

memset(psfile,80,NULL);
in=fopen("Pageflt5.grph","w");
printf("\nPut the name of the postscript file:");
scanf("%s",psfile);                 /* get the name of postscript file */

valid=0;
while(valid != 1 )
{
        printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
        scanf("%s",svalue);         /* get the page_size */
        p_size=atoi(svalue);
        if ( (p_size == 512 ) || (p_size == 1024) ||
             (p_size == 2048) || (p_size == 4096) ||
             (p_size == 8192) )
                valid =1;
        else
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
}

printf("\nPut the algorithm or the method:");
scanf("%s",methd);                  /* get the algorithm or method */

valid=0;
while( valid != 1)
{
        printf("\nPut the process number: ");
        scanf("%s",svalue);
        pnum=atoi(svalue);          /* get the process number */
        if (pnum == 0)
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

valid=0;
while( valid != 1 )
{
        printf("\nPut the number of points: ");
        scanf("%s",svalue);
        n=atoi(svalue);             /* get # of points */
        if (n == 0)
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

                                    /* get x and y values */
printf("\n\t x values are # of frames allocated and y values are # of page
        faults\n");
printf("\t (Note: If you want exit, put <quit>.) \n");
for(i=0; i<n; i++)
{
```

```
		printf("\n Put the x value: ");
		scanf("%s",svalue);
		if (strcmp(svalue,"quit") == 0 )
			return;
		else
			value_x[i]=atof(svalue);
					/* get x values (# of frames allocated) */
	}

	for(i=0; i<n; i++)
	{
		printf("\n Put the y value: ");
		scanf("%s",svalue);
		if (strcmp(svalue,"quit") == 0 )
			return;
		else
			value_y1[i]=atof(svalue);
					/* get y values (# of page faults) */
	}

	fprintf(in,"#!/contrib/bin/blt_wish -f\n");
	fprintf(in,"\n");
	fprintf(in,"if [file exists /contrib/library] {\n");
	fprintf(in,"   set blt_library /contrib/library\n");
	fprintf(in,"}\n");
	fprintf(in,"\n");
	fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
					/* set the font and size */
	fprintf(in,"option add *Blt_text.Font *Times-Bold-R*10* \n");
	fprintf(in,"option add *graph.xTitle %cNumber of frames allocated %c \n",34,34);
					/* title of x axis */
	fprintf(in,"option add *graph.yTitle %cNumber of page faults %c \n",34,34);
					/* title of y axis */
	fprintf(in,"option add *graph.title %cPage faults of process %d (page size:%d,
	%s)%c \n",34,pnum,p_size,methd,34);   /* title of graph */
	fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
	strcpy(s,"Number of page faults vs. number of frames");
	RepeatBodyGraph1(in,psfile,s);

	fprintf(in," set X {\n");		/* write x values to file */
	for(i=0; i<n; i++)
		fprintf(in,"%f ",value_x[i]);
	fprintf(in,"\n");
	fprintf(in,"}\n");
	fprintf(in,"\n");
	fprintf(in," set Y1 {\n");		/* write y values to file */
	for(i=0; i<n; i++)
		fprintf(in,"%f ",value_y1[i]);
	fprintf(in,"\n");
	fprintf(in,"}\n");
	fprintf(in,"\n");
	fprintf(in,"\n");

	fprintf(in,"$graph element create Highest -xdata $X -ydata $Y1 %c\n",92);
	fprintf(in,"       -symbol square -linewidth 1\n");
	RepeatBodyGraph2(in);
	fclose(in);

	system("chmod 777 Pageflt5.grph");
	system("Pageflt5.grph");		/* show the graph */
	return;
}

/*/////////////////////////////////////////////////////////////////////////////
//	Function : PageFaultGraph6()
//	Purpose  : This function is used to generate the page fault graph for
//		   a process when the time intervals are changed in the clock
```

```
//                algorithm.
///////////////////////////////////////////////////////////////////////////////*/

PageFaultGraph6()
{
        FILE    *in;
        int     i;
        float   value_x[100];               /* x values are time intervals for clock
                                               algorithm */
        float   value_y[100];               /* y values are # of page faults */
        char    psfile[81];                 /* name of postscript file */
        char    s[81];
        char    svalue[81];
        int     n,pnum,m_size,p_size;
        int     valid;

        memset(psfile,81,NULL);
        in=fopen("Pageflt6.grph","w");
        printf("\nPut the name of the postscript file: ");
        scanf("%s",psfile);                 /* get the name of postscipt file */

        valid=0;
        while(valid != 1 )
        {
                printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
                scanf("%s",svalue);         /* get the page size */
                p_size=atoi(svalue);
                if ( (p_size == 512 ) || (p_size == 1024) ||
                     (p_size == 2048) || (p_size == 4096) ||
                     (p_size == 8192) )
                        valid =1;
                else
                {
                        valid=0;
                        printf("\n\tError: Invalid input, try again.\n");
                }
        }

        valid=0;
        while(valid != 1)
        {
                printf("Put the process number: ");
                scanf("%s",svalue);         /* get the name of postscript file */
                pnum=atoi(svalue);
                if (pnum == 0)
                {
                        valid=0;
                        printf("\n\tError: Invalid input, try again.\n");
                }
                else
                        valid=1;
        }

        valid=0;
        while(valid != 1)
        {
                printf("Put the size of memory: ");
                scanf("%s",svalue);         /* get the memory */
                m_size=atoi(svalue);
                if (m_size == 0)
                {
                        valid=0;
                        printf("\n\tError: Invalid input, try again.\n");
                }
                else
                        valid=1;
        }
```

```
valid=0;
while(valid != 1)
{
        printf("Put the number of points: ");
        scanf("%s",svalue);
        n=atoi(svalue);                 /* get # of points */
        if (n == 0)
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

printf("\n\t x values are time intervals and y values are # of page faults\n");
printf("\t (Note: If you want exit, put <quit>.) \n");
for(i=0; i<n; i++)
{
        printf("\n Put the x value: ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_x[i]=atoi(svalue);
                                        /* get x values (time intervals) */
}

for(i=0; i<n; i++)
{
        printf("\n Put the y value: ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y[i]=atoi(svalue);
                                        /* get y values (# of page faults) */
}

fprintf(in,"#!/contrib/bin/blt_wish -f\n");
fprintf(in,"\n");
fprintf(in,"if [file exists /contrib/library] {\n");
fprintf(in,"    set blt_library /contrib/library\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
fprintf(in,"option add *graph.xTitle %cTime intervals %c \n",34,34);
                                        /* title of x axis */
fprintf(in,"option add *graph.yTitle %cNumber of page faults %c \n",34,34);
                                        /* title of y axis */
fprintf(in,"option add *graph.title %cPage faults of process%d (memory:%d
bytes,clock)%c \n",34,pnum,m_size,34);
                                        /* title of graph */
fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
strcpy(s,"Number of page faults vs. time intervals");
RepeatBodyGraph1(in,psfile,s);

                                        /* print x and y values to the file */
fprintf(in," set X {\n");               /* write x values to the file */
for(i=0; i<n; i++)
        fprintf(in,"%f ",value_x[i]);
fprintf(in,"\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in," set Y {\n");               /* write y values to file */
```

```
        for(i=0; i<n; i++)
                fprintf(in,"%f ",value_y[i]);
        fprintf(in,"\n");
        fprintf(in,")\n");
        fprintf(in,"\n");
        fprintf(in,"\n");

        fprintf(in,"$graph element create Clock -xdata $X -ydata $Y %c\n",92);
        fprintf(in,"       -symbol square -linewidth 1\n");
        RepeatBodyGraph2(in);
        fclose(in);

        system("chmod 777 Pageflt6.grph");
        system("Pageflt6.grph");               /* show the graph */
        return;
}

/*///////////////////////////////////////////////////////////////////////////////////////
//      Function : PageFaultGraph7()
//      Purpose  : This function is used to generate the page fault graph for a process
//                 when the time intervals are changed in the additional-reference-bits
//                 algorithm.
///////////////////////////////////////////////////////////////////////////////////////*/

PageFaultGraph7()
{
        FILE    *in;
        int     i;
        float   value_x[100];                  /* x values are time intervals for the
                                                  additional-reference-bits algorithm */
        float   value_y[100];                  /* y values are # of page faults */
        char    psfile[81];                    /* name of the postscript file */
        char    s[81];
        char    svalue[81];
        int     n,pnum,m_size,p_size;
        int     valid;

        memset(psfile,81,NULL);
        in=fopen("Pageflt7.grph","w");
        printf("\nPut the name of the postscript file: ");
        scanf("%s",psfile);                    /* get the name of postscript file */

        valid=0;
        while(valid != 1 )
        {
                printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
                scanf("%s",svalue);            /* get the page size */
                p_size=atoi(svalue);
                if ( (p_size == 512 ) || (p_size == 1024) ||
                     (p_size == 2048) || (p_size == 4096) ||
                     (p_size == 8192) )
                        valid =1;
                else
                {
                        valid=0;
                        printf("\n\tError: Invalid input, try again.\n");
                }
        }

        valid=0;
        while(valid != 1)
        {
                printf("Put the process number: ");
                scanf("%s",svalue);            /* get the # of processes */
                pnum=atoi(svalue);
                if (pnum == 0)
                {
```

```
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

valid=0;
while(valid != 1)
{
        printf("Put the size of memory: ");
        scanf("%s",svalue);             /* get the memory size */
        m_size=atoi(svalue);
        if (m_size == 0)
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

valid=0;
while(valid != 1)
{
        printf("Put the number of points: ");
        scanf("%s",svalue);             /* get the # of points */
        n=atoi(svalue);
        if (n == 0)
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

printf("\n\t x values are time intervals and y values are # of page faults\n");
printf("\t (Note: If you want exit, put <quit>.) \n");
for(i=0; i<n; i++)
{
        printf("\n Put the x value: ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_x[i]=atoi(svalue);
                                        /* get x values */
}

for(i=0; i<n; i++)
{
        printf("\n Put the y value: ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y[i]=atoi(svalue);
                                        /* get y values */
}

fprintf(in,"#!/contrib/bin/blt_wish -f\n");
fprintf(in,"\n");
fprintf(in,"if [file exists /contrib/library] {\n");
fprintf(in,"    set blt_library /contrib/library\n");
fprintf(in,"}\n");
fprintf(in,"\n");
```

```
        fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
        fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
        fprintf(in,"option add *graph.xTitle %cTime intervals %c \n",34,34);
                                            /* title of x axis */
        fprintf(in,"option add *graph.yTitle %cNumber of page faults %c \n",34,34);
                                            /* title of y values */
        fprintf(in,"option add *graph.title %cPage faults of process%d (memory : %d
        bytes,add-ref-bits)%c \n",34,pnum,m_size,34);
                                            /* title of graph */
        fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
        strcpy(s,"Number of page faults vs. time intervals");
        RepeatBodyGraph1(in,psfile,s);

                                            /* print x and y values to the file */
        fprintf(in," set X {\n");           /* write x values to file */
        for(i=0; i<n; i++)
                fprintf(in,"%f ",value_x[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");
        fprintf(in,"\n");
        fprintf(in," set Y {\n");           /* write y values to file */
        for(i=0; i<n; i++)
                fprintf(in,"%f ",value_y[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");
        fprintf(in,"\n");
        fprintf(in,"\n");

        fprintf(in,"$graph element create Add-ref-bits -xdata $X -ydata $Y %c\n",92);
        fprintf(in,"        -symbol square -linewidth 1\n");
        RepeatBodyGraph2(in);
        fclose(in);

        system("chmod 777 Pageflt7.grph");
        system("Pageflt7.grph");            /* show the graph */
        return;
}

/*///////////////////////////////////////////////////////////////////////////////
//      Function : MemUtilGraph1()
//      Purpose  : This function is used to generate the memory utilization
//                 graph over 3 different algorithms. The x values are process
//                 IDs and the y vaules are the memory utilization of each
//                 process.
///////////////////////////////////////////////////////////////////////////////*/

MemUtilGraph1()
{
        FILE    *in;
        int     i;
        float   value_y1[MAX_PROCESS];      /* y values for new implementation */
        float   value_y2[MAX_PROCESS];      /* y values for clock algorithm */
        float   value_y3[MAX_PROCESS];      /* y values for additional-reference-bits
                                               algorithm */
        float   value_x[MAX_PROCESS];       /* x values */
        char    psfile[81];                 /* name of postscript file */
        char    s[81];
        char    svalue[81];
        int     p_size,m_size;
        int     valid;

        No_process=5;
        memset(psfile,80,NULL);
        in=fopen("Memutil1.grph","w");
        printf("\nPut the name of the postscript file: ");
        scanf("%s",psfile);                 /* get the name of postscript file */
```

```
valid=0;
while(valid != 1 )
{
        printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
        scanf("%s",svalue);             /* get the page size */
        p_size=atoi(svalue);
        if ( (p_size == 512 ) || (p_size == 1024) ||
             (p_size == 2048) || (p_size == 4096) ||
             (p_size == 8192) )
                valid =1;
        else
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
}

valid=0;
while(valid != 1)
{
        printf("Put the size of memory: ");
        scanf("%s",svalue);             /* get the memory size */
        m_size=atoi(svalue);
        if (m_size == 0)
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

printf("\n\t x values are process IDs and y values are memory utilization\n");
printf("\t (Note: If you want exit, put <quit>.) \n");
                                        /* get x and y values */
for(i=0; i<No_process; i++)
        value_x[i]=i+1;                 /* x values are process IDs */
for(i=0; i<No_process; i++)
{
        printf("\n Put the y1 value (New implementation): ");
        scanf("%s",svalue);             /* get y values for the new implementation */
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y1[i]=atoi(svalue);
}
for(i=0; i<No_process; i++)
{
        printf("\n Put the y2 value (Clock algorithm): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y2[i]=atoi(svalue);
                                        /* get y values for the clock algorithm */
}
for(i=0; i<No_process; i++)
{
        printf("\n Put the y3 value (Additional-reference-bits algorithm): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y3[i]=atoi(svalue);
                                        /* get y values for the additional-reference-
                                           bits algorithm */
}
```

```
	fprintf(in,"#!/contrib/bin/blt_wish -f\n");
	fprintf(in,"\n");
	fprintf(in,"if [file exists /contrib/library] {\n");
	fprintf(in,"   set blt_library /contrib/library\n");
	fprintf(in,"}\n");
	fprintf(in,"\n");
	fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
	fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
	fprintf(in,"option add *graph.xTitle %cProcess ID %c \n",34,34);
	                                        /* title of x axis */
	fprintf(in,"option add *graph.yTitle %cMemory utilization %c \n",34,34);
	                                        /* title of y axis */
	fprintf(in,"option add *graph.title %cMemory utilization over different algorithms
	(%d / %d)%c \n",34,m_size,p_size,34); /* title of graph */
	fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
	strcpy(s,"Memory utilization over different algorithms");
	RepeatBodyGraph1(in,psfile);

	fprintf(in," set X {\n");            /* write x values to the file */
	for(i=0; i<No_process; i++)
		fprintf(in,"%f ",value_x[i]);
	fprintf(in,"\n");
	fprintf(in,"}\n");
	fprintf(in,"\n");
	fprintf(in," set Y1 {\n");           /* write y values to the file */
	for(i=0; i<No_process; i++)
		fprintf(in,"%f ",value_y1[i]);
	fprintf(in,"\n");
	fprintf(in,"}\n");

	fprintf(in,"set Y2 {\n");            /* write y values to the file */
	for(i=0; i<No_process; i++)
		fprintf(in,"%f ",value_y2[i]);
	fprintf(in,"\n");
	fprintf(in,"}\n");

	fprintf(in,"set Y3 {\n");            /* write y values to the file */
	for(i=0; i<No_process; i++)
		fprintf(in,"%f ",value_y3[i]);
	fprintf(in,"\n");
	fprintf(in,"}\n");
	fprintf(in,"\n");
	fprintf(in,"\n");

	fprintf(in,"$graph element create Highest-leaf -xdata $X -ydata $Y1 %c\n",92);
	fprintf(in,"      -symbol diamond -linewidth 0\n");
	fprintf(in,"$graph element create Clock(28000) -xdata $X -ydata $Y2 %c\n",92);
	fprintf(in,"      -symbol cross -linewidth 0\n");
	fprintf(in,"$graph element create Add-ref-bits(140000) -xdata $X -ydata $Y3
		%c\n",92);
	fprintf(in,"      -symbol square -linewidth 0\n");
	RepeatBodyGraph2(in);
	fclose(in);

	system("chmod 777 Memutil1.grph");
	system("Memutil1.grph");             /* show the graph */
	return;
}

/*////////////////////////////////////////////////////////////////////////////////////////
//	Function : MemUtilGraph2()
//	Purpose  : This function is used to generate the memory utilization graph over 4
//	           different methods in the new implemantation. The x values are the
//	           process IDs and the y values are the memory utilization of each process.
////////////////////////////////////////////////////////////////////////////////////////*/
```

```
MemUtilGraph2()
{
	FILE	*in;				/* file descriptor of graph file */
	int	i;
	float	value_y1[MAX_PROCESS];		/* y values for the leftmost leaf method */
	float	value_y2[MAX_PROCESS];		/* y values for the rightmost leaf method */
	float	value_y3[MAX_PROCESS];		/* y values for the highest leaf method */
	float	value_y4[MAX_PROCESS];		/* y values for the LRU leaf method */
	float	value_x[MAX_PROCESS];		/* x values */
	char	psfile[81];			/* name of the postscript file */
	char	s[81];
	char	svalue[81];
	int	p_size,m_size;
	int	valid;

	No_process=5;
	memset(psfile,80,NULL);
	in=fopen("Memutil2.grph","w");
	printf("\nPut the name of the postscript file: ");
	scanf("%s",psfile);			/* get the name of postscript file */

	valid=0;
	while(valid != 1 )
	{
		printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
		scanf("%s",svalue);		/* get the page size */
		p_size=atoi(svalue);
		if ( (p_size == 512 ) || (p_size == 1024) ||
		     (p_size == 2048) || (p_size == 4096) ||
		     (p_size == 8192) )
			valid =1;
		else
		{
			valid=0;
			printf("\n\tError: Invalid input, try again.\n");
		}
	}

	valid=0;
	while(valid != 1)
	{
		printf("Put the size of memory: ");
		scanf("%s",svalue);		/* get the memory size */
		m_size=atoi(svalue);
		if (m_size == 0)
		{
			valid=0;
			printf("\n\tError: Invalid input, try again.\n");
		}
		else
			valid=1;
	}

	printf("\n\t x values are process IDs and y values are memory utilization\n");
	printf("\t (Note: If you want exit, put <quit>.) \n");

						/* get the x and y values */
	for(i=0; i<No_process; i++)
		value_x[i]=i+1;			/* x values are process IDs */

	for(i=0; i<No_process; i++)
	{
		printf("\n Put the y1 value (Leftmost leaf method): ");
		scanf("%s",svalue);
		if (strcmp(svalue,"quit") == 0 )
			return;
		else
```

```
                value_y1[i]=atoi(svalue);
                                    /* get y values for the leftmost leaf
                                       method */
}
for(i=0; i<No_process; i++)
{
        printf("\n Put the y2 value (Rightmost leaf method): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y2[i]=atoi(svalue);
                                    /* get y values for the rightmost leaf
                                       method */
}
for(i=0; i<No_process; i++)
{
        printf("\n Put the y3 value (Highest leaf method): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y3[i]=atoi(svalue);
                                    /* get y values for the highest leaf
                                       method */
}
for(i=0; i<No_process; i++)
{
        printf("\n Put the y4 value (LRU leaf method): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y4[i]=atoi(svalue);
                                    /* get y values for the LRU leaf method */

}

fprintf(in,"#!/contrib/bin/blt_wish -f\n");
fprintf(in,"\n");
fprintf(in,"if [file exists /contrib/library] {\n");
fprintf(in,"    set blt_library /contrib/library\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
fprintf(in,"option add *graph.xTitle %cProcess ID %c \n",34,34);
                                    /* title of x axis */
fprintf(in,"option add *graph.yTitle %cMemory utilization %c \n",34,34);
                                    /* title of y axis */
fprintf(in,"option add *graph.title %c Memory utilization for new implementation
        (%d / %d)%c \n",34,m_size,p_size,34);
                                    /* title of the graph */
fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
strcpy(s,"Memory utilization of new implementation");
RepeatBodyGraph1(in,psfile,s);

fprintf(in," set X {\n");           /* write x values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_x[i]);
fprintf(in,"\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in," set Y1 {\n");          /* write y values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y1[i]);
fprintf(in,"\n");
```

```
        fprintf(in,"}\n");

        fprintf(in,"set Y2 {\n");              /* write y values to the file */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y2[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");

        fprintf(in,"set Y3 {\n");              /* write y values to the file */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y3[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");
        fprintf(in,"set Y4 {\n");              /* write y values to the file */
        for(i=0; i<No_process; i++)
                fprintf(in,"%f ",value_y4[i]);
        fprintf(in,"\n");
        fprintf(in,"}\n");
        fprintf(in,"\n");
        fprintf(in,"\n");

        fprintf(in,"$graph element create Leftmost -xdata $X -ydata $Y1 %c\n",92);
        fprintf(in,"       -symbol plus -linewidth 0\n");
        fprintf(in,"$graph element create Rightmost -xdata $X -ydata $Y2 %c\n",92);
        fprintf(in,"       -symbol cross -linewidth 0\n");
        fprintf(in,"$graph element create Highest -xdata $X -ydata $Y3 %c\n",92);
        fprintf(in,"       -symbol square -linewidth 0\n");
        fprintf(in,"$graph element create LRU_leaf -xdata $X -ydata $Y4 %c\n",92);
        fprintf(in,"       -symbol diamond -linewidth 0\n");
        RepeatBodyGraph2(in);
        fclose(in);

        system("chmod 777 Memutil2.grph");
        system("Memutil2.grph");               /* show the graph */
        return;
}

/*/////////////////////////////////////////////////////////////////////////////////////////
//      Function : MemUtilGraph3()
//      Purpose  : This function is used to generate the memory utilization graph for the
//                 3 different intervals in the clock or the additional-reference-bits
//                 algorithm.
/////////////////////////////////////////////////////////////////////////////////////////*/

MemUtilGraph3(int sel)
{
        FILE    *in;                           /* file descriptor of graph file */
        int     i;
        float   value_y1[MAX_PROCESS];         /* memory occupancy of each process when an
                                                  interval is given to either the clock or
                                                  additional-reference-bits algorithm */
        float   value_y2[MAX_PROCESS];
        float   value_y3[MAX_PROCESS];
        float   value_x[MAX_PROCESS];          /* x values are process IDs */
        char    psfile[81];                    /* name of the postscript file */
        char    s[81];
        char    svalue[81];
        int     p_size,m_size;
        int     valid;

        No_process=5;
        memset(psfile,80,NULL);
        if (sel == 3 )
                in=fopen("Memutil3.grph","w");
        else if (sel ==4 )
                in=fopen("Memutil4.grph","w");
        printf("\nPut the name of the postscript file: ");
```

```
scanf("%s",psfile);                    /* get the name of postscript file */

valid=0;
while(valid != 1 )
{
        printf("\nPut the page size (512, 1024, 2048, 4096 or 8192): ");
        scanf("%s",svalue);            /* get the page size */
        p_size=atoi(svalue);
        if ( (p_size == 512 ) || (p_size == 1024) ||
             (p_size == 2048) || (p_size == 4096) ||
             (p_size == 8192) )
               valid =1;
        else
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
}

valid=0;
while(valid != 1)
{
        printf("Put the size of memory: ");
        scanf("%s",svalue);            /* get the memory size */
        m_size=atoi(svalue);
        if (m_size == 0)
        {
                valid=0;
                printf("\n\tError: Invalid input, try again.\n");
        }
        else
                valid=1;
}

printf("\n\t x values are process IDs and y values are memory utilization\n");
printf("\t (Note: If you want exit, put <quit>.) \n");
for(i=0; i<No_process; i++)
        value_x[i]=i+1;                /* x values are process IDs */
for(i=0; i<No_process; i++)
{
        if ( sel == 3 )
                printf("\n Put the y1 value (interval 16800): ");
        if ( sel == 4 )
                printf("\n Put the y1 value (interval 70000): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y1[i]=atoi(svalue);
                                       /* get the y values */
}
for(i=0; i<No_process; i++)
{
        if ( sel == 3 )
                printf("\n Put the y2 value (interval 28000): ");
        if ( sel == 4 )
                printf("\n Put the y2 value (interval 140000): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
                return;
        else
                value_y2[i]=atoi(svalue);
                                       /* get the y values */
}
for(i=0; i<No_process; i++)
{
        if ( sel == 3 )
```

```
            printf("\n Put the y3 value (interval 39200): ");
        if ( sel == 4 )
            printf("\n Put the y3 value (interval 210000): ");
        scanf("%s",svalue);
        if (strcmp(svalue,"quit") == 0 )
            return;
        else
            value_y3[i]=atoi(svalue);
                                        /* get the y values */
}

fprintf(in,"#!/contrib/bin/blt_wish -f\n");
fprintf(in,"\n");
fprintf(in,"if [file exists /contrib/library] {\n");
fprintf(in,"    set blt_library /contrib/library\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"option add *Blt_htext.Font *Times-Bold-R*14* \n");
fprintf(in,"option add *Blt_text.Font *Times-Bold-R*12* \n");
fprintf(in,"option add *graph.xTitle %cProcess ID %c \n",34,34);
                                        /* title of x axis */
fprintf(in,"option add *graph.yTitle %cMemory utilization %c \n",34,34);
                                        /* title of y axis */
if ( sel == 3 )
        fprintf(in,"option add *graph.title %cMemory utilization for clock
          algorithm (%d / %d) %c \n",34,m_size,p_size,34);
                                        /* title of the graph */
else if (sel ==4 )
        fprintf(in,"option add *graph.title %cMemory utilization for add-ref-bits
         algorithm (%d / %d) %c \n",34,m_size,p_size,34);
                                        /* title of the graph */
fprintf(in,"option add *Blt_graph.legendFont *Times-*-*-8* \n");
if ( sel == 3 )
        strcpy(s,"Memory utilization of clock algorithm");
else if (sel ==4 )
        strcpy(s,"Memory utilization of add-ref-bits algorithm");
RepeatBodyGraph1(in,psfile,s);
fprintf(in," set X {\n");               /* write x values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_x[i]);
fprintf(in,"\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in," set Y1 {\n");              /* write y values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y1[i]);
fprintf(in,"\n");
fprintf(in,"}\n");

fprintf(in,"set Y2 {\n");               /* write y values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y2[i]);
fprintf(in,"\n");
fprintf(in,"}\n");

fprintf(in,"set Y3 {\n");               /* write y values to the file */
for(i=0; i<No_process; i++)
        fprintf(in,"%f ",value_y3[i]);
fprintf(in,"\n");
fprintf(in,"}\n");
fprintf(in,"\n");
fprintf(in,"\n");

if(sel ==3)
        fprintf(in,"$graph element create Interval-16800 -xdata $X -ydata $Y1
                %c\n",92);
if(sel ==4)
```

```
                fprintf(in,"$graph element create Interval-70000 -xdata $X -ydata $Y1
                        %c\n",92);

        fprintf(in,"      -symbol diamond -linewidth 0\n");

        if(sel ==3)
                fprintf(in,"$graph element create Interval-28000 -xdata $X -ydata $Y2
                        %c\n",92);
        if(sel ==4)
                fprintf(in,"$graph element create Interval-140000 -xdata $X -ydata $Y2
                        %c\n",92);
        fprintf(in,"      -symbol cross -linewidth 0\n");

        if(sel ==3)
                fprintf(in,"$graph element create Interval-39200 -xdata $X -ydata $Y3
                        %c\n",92);
        if(sel ==4)
                fprintf(in,"$graph element create Interval-210000 -xdata $X -ydata $Y3
                        %c\n",92);
        fprintf(in,"      -symbol square -linewidth 0\n");
        RepeatBodyGraph2(in);
        fclose(in);

        if (sel == 3 )
        {
                system("chmod 777 Memutil3.grph");
                system("Memutil3.grph");        /* show the graph */
        }
        else if (sel == 4 )
        {
                system("chmod 777 Memutil4.grph");
                system("Memutil4.grph");        /* show the graph */
        }
        return;
}

/*/////////////////////////////////////////////////////////////////////////////////////////
//      Function : RepeatBodyGraph1()
//      Purpose  : This function has the front part of the repeated codes to generate a
//                 graph.
/////////////////////////////////////////////////////////////////////////////////////////*/

RepeatBodyGraph1(FILE *in,char psfile[80],char s[80])
{

        fprintf(in,"\n");
        fprintf(in,"set visual [winfo screenvisual .] \n");
        fprintf(in,"if { $visual != %cstaticgray%c } { \n",34,34);
        fprintf(in,"     option add *print.background yellow \n");
                                        /* set the background color of the print
                                           button */
        fprintf(in,"     option add *quit.background white \n");
                                        /* set the background color of the quit
                                           button */
        fprintf(in,"}\n");
        fprintf(in,"\n");
        fprintf(in,"global graph\n");
        fprintf(in,"set graph .graph \n");
        fprintf(in,"blt_htext .header -text {%c%c\n",37,37);
                                        /* create the header part of the graph */
        fprintf(in,"%c%c %s \n",37,37,s);
        fprintf(in,"To create a postscript file, press the %c%c \n",37,37);
        fprintf(in,"button $blt_htext(widget).print -text print -command {\n");
        fprintf(in,"   .graph postscript %s -pagewidth 6i -pageheight 4i -landscape false
                \n", psfile);
        fprintf(in," }\n");
        fprintf(in,"\n");
```

```
        fprintf(in,"$blt_htext(widget) append $blt_htext(widget).print\n");
                                                /* create the print button */
        fprintf(in,"%c%c button. }\n",37,37);
        fprintf(in,"\n");
        fprintf(in,"blt_graph $graph\n");
        fprintf(in,"\n");
        fprintf(in,"blt_htext .footer -text {To finish, press the %c%c \n",37,37);
                                                /* create the footer of the graph */
        fprintf(in,"button $blt_htext(widget).quit -text quit -command {destroy
                .}\n");                         /* create the quit button */
        fprintf(in,"$blt_htext(widget) append $blt_htext(widget).quit\n");
        fprintf(in,"%c%c button.%c%c\n",37,37,37,37);
        fprintf(in,"$blt_htext(widget) -padx 20\n");
        fprintf(in,"%c%c}\n",37,37);
}

/*///////////////////////////////////////////////////////////////////////////////////////
//      Function : RepeatBodyGraph2()
//      Purpose  : This function is the behind part of the repeated code to generate a
//                 graph.
///////////////////////////////////////////////////////////////////////////////////////*/

RepeatBodyGraph2(FILE *in)
{
        fprintf(in,"# $graph crosshairs set on\n");
        fprintf(in,"\n");
        fprintf(in,"pack append . %c\n",92);
        fprintf(in," .header { padx 20 pady 10 } %c\n",92);
        fprintf(in,"        .graph { fill expand } %c\n",92);
        fprintf(in," .footer { padx 20 pady 10 }\n");
        fprintf(in,"\n");
        fprintf(in,"wm min . 0 0\n");
        fprintf(in,"\n");
        fprintf(in,"bind $graph <B1-ButtonRelease> { %cW crosshairs toggle  }\n",37);
        fprintf(in,"\n");
        fprintf(in,"proc TurnOnHairs { graph } {\n");
        fprintf(in,"    bind $graph <Any-Motion> {%cW crosshairs configure  -position
                @%c%cx, %c%cy}\n",37,37,37,37,37);
        fprintf(in," }\n");
        fprintf(in,"\n");
        fprintf(in,"proc TurnOffHairs { graph } {\n");
        fprintf(in,"    bind $graph <Any-Motion> {%cW crosshairs configure  -position
                @%c%cx, %c%cy}\n",37,37,37,37,37);
        fprintf(in," }\n");
        fprintf(in,"\n");
        fprintf(in,"bind $graph <Enter> { TurnOnHairs %c%cW }\n",37,37);
        fprintf(in,"bind $graph <Leave> { TurnOffHairs %c%cW }\n",37,37);
        fclose(in);
}

/*///////////////////////////////////////////////////////////////////////////////////////
//                              Makefile
// Makefile for new LRU approximation implementation. There are five files to make the
// execution file.
///////////////////////////////////////////////////////////////////////////////////////*/

CFLAGS = -O
the:    Perform.o Newapp.o Ckago.o Addref.o Graph.o
        cc $(CFLAGS) -o the Perform.o Newapp.o Ckago.o Addref.o Graph.o
```

VITA

Eunjae Jung

Candidate for the Degree of

Master of Science

Thesis: LRU PAGE REPLACEMENT ALGORITHM: A NEW APPROXIMATION IMPLEMENTATION

Major Field: Computer Science

Biographical:

Personal Data: Born in Wonju, Feburary 13, 1964, son of Changkun Jung, M.D., and Mrs. Sunja Jo Jung.

Education: Received Bachelor of Science in Mathematics from Myong Ji University, Seoul, Korea, in Feburary 1991; completed requirements for the Master of Science Degree at the Computer Science Department at Oklahoma State University in July 1996.

Professional Membership: Korean-American Scientists and Engineers Association.