A Novel Multi-Layer Virtual Memory System to Solve Memory Leak Problem

Prepared by

Ahmed Ali Otoom

Supervisor

Dr. Mohammad I. Malkawi

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science.

Graduate College of Computer Studies

Amman Arab University for Graduate Studies

July, 2007

www.manaraa.com

AUTHORIZATION OF DISSEMINATION

I, the undersigned "Ahmed Ali Otoom", authorize hereby Amman Arab University for Graduate Studies to provide copies of this dissertation to libraries, institutions, agencies, and any other parties upon their request.

Name: Ahmed Ali Otoom

Signature:

Date: 8 – July - 2007

www.manaraa.com

c

RESOLUTION OF THE EXAMINING COMMITTEE

This dissertation titled " A Novel Multi-Layer Virtual Memory System to Solve Memory Leak Problem ", has been defended and approved on 7 /7/ 2007

Examining Committee	Title	Signature
Dr. Mohammad Al-Haj Hassan	Chair	
Dr. Mohammad Malkawi	Member and Supervisor	
Dr. Naim Ajlouni	Member	
Dr. Ghassan Al-Utaibi	Member	

للاستشارات المنارة
www.manaraa.com

Acknowledgment

"All praises and thanks to ALLAH"

I would like to thank my supervisor Dr. Mohammad Malkawi who provided me full support, encouragement, and guidance in order to get this dissertation ready. Without his help and support, this work would not have been possible. He was always available any time I needed help.

My sincere thanks go to Dr. Naim Ajlouni; the Dean of Graduate College of Computer Studies, Dr. Hilal Al-Bayatti; the Head of Computer Science Department, all of the lecturers, administration, and the staff of Amman Arab University for Graduate Studies.

I also thank my parents, my wife, my daughters: Moneera, Mona, Roqayya, and Eman, my sons: Abedallah, and Abedalrahman, my brothers: Abu Walid, Mohammed, Sultan, Essa, Dr. Sameer, and Samer, my sisters: Mona, Manwa, Ebtisam, Amal, Samer, Sawsan, and Wafa, my relatives, my friends, all of my colleagues, and Dr. Mohammed Alrefai for their encouragement.

www.manaraa.com

Dedication

I dedicate this work to

My Parents

www.manaraa.com

Table of Contents

Acknowledgment v
Dedication vi
Table of Contents vii
List of Figures xi
List of Tables xii
Abstract xiii
Abstract in Arabic xv
Arabic Summary xv
CHAPTER ONE INTRODUCTION 2
1.0 Introduction 2
1.1 The Statement of the Problem 4
1.1.1 Problem Definition 4
1.1.2 Dissertation Questions 5
1.1.3 Dissertation Scope 6
1.1.4 Dissertation Contributions 6
1.2 Dissertation Methodology 8
1.3 Organization of Dissertation 8
1.4 Conclusion 10
Chapter Two Dynamic Memory Allocation and Current Approaches for Solving Memory Leak Problem 11
2.0 Introduction 11
2.1.0 Dynamic Memory Allocation 11
2.1.1 Dynamic Memory Allocation Strategies 14
2.1.2.0 Dynamic Memory Allocation Algorithms 14
2.1.2.1 Fixed-Size-Blocks Allocation 15
2.1.2.2 Buddy Blocks Allocation 15
2.1.2.3 Heap-based Memory Allocation 15
2.1.3 Dynamic Memory Allocation Functions 15
2.1.4 The Access State of Referencing a Dynamically Allocated Object 17
2.1.5 Dynamic Memory Allocation and Memory Leak 17
2.2 Approaches for memory leak detection 19
2.2.1 Assertions and Static Analysis 19
2.2.2 Dynamic checkers 21
2.2.3 Hardware-assisted solutions 23

www.manaraa.com

2.2.4 Garbage collectors and memory leak 27
2.3 Shortcomings of current approaches 31
2.4 Conclusion 32
Chapter Three A New Approach for Memory Leak Detection (MLD) Using Aging in Physical Memory Space 33
3.0 Introduction 33
3.1 Memory Leak Detection (MLD) Using Aging in Physical Memory 35
3.2 MLD Algorithm Pseudo Code 39
3.3 MLD Algorithm Explanation 40
3.3.1 MLD Main Function 42
3.3.2 Initialization 43
3.3.3 Bookkeeping 44
3.3.4 Memory Allocation and Deallocation Functions 46
3.3.4.1 Malloc () 47
3.3.4.2 free () 48
3.3.5 Memory Leak Detection and Sweeping 49
3.3.5.1 Memory Leak Detection 49
3.3.5.2 Memory Leak Sweeping 50
3.4 Crash Delay 51
3.5 False Positives 52
3.6 False Negatives 53
3.7 Memory Leaks and Telemetry 54
3.8 Conclusion 55
Chapter Four Multi-Layer Virtual Memory System (ML-VMS) 56
4.0 Introduction 56
4.1 Multi-Layer Virtual Memory System (ML-VMS) 59
4.2 Address Resolution 61
4.3 The MLDR Algorithm Based on the ML-VMS 62
4.3.1 Memory Allocation 62
4.3.2 Memory Deallocation 63
4.3.3 ML-VMS and Aging 63
4.3.4 Object Recovery 64
4.4 MLDR algorithm block diagram based on ML-VMS 64
4.5 MLDR Explanation 67
4.5.1 Memory Allocation Module Explanation 67
4.5.2 Memory Deallocation Module Explanation 69
4.5.3 MLDR with Aging Module Explanation 71
4.5.4 Object Recovery Module Explanation 74

www.manaraa.com

4.6 Crash Preventing 76
4.7 False Positives 78
4.8 False Negatives 79
4.9 Memory Leaks and Telemetry 80
4.10 Conclusion 81
Chapter Five Performance Evaluation and Simulation 83
5.1 Performance Analysis 83
5.1.1 The MLD Complexity 84
5.1.2 The MLD Parallelized 86
5.1.3 Demand Paging Access Time 88
5.1.4 ML-VMS Access Time 89
5.1.5 ML-VMS Overhead Cost 90
5.2 Trace-Driven Simulation of Memory Leak Detection Algorithm 91
5.2.1 Data Trace Collection and Leak Injection (Stage One) 92
5.2.1.1 MLD Bench Mark 92
5.2.1.2 Data Collection 93
5.2.1.3 Leak Injection 97
5.2.2 Trace-Driven Simulation (Stage Two) 98
5.2.3 Simulation Assumptions 101
5.2.4 Input and Output Parameters 101
5.2.5 Simulation Model 101
5.2.5.1 Simulation Model (Demand Paging) 102
5.2.5.2 Simulation Model of Demand Paging with MLD Algorithm 104
5.2.6 Simulation Results 105
5.2.6.1 Time versus Heap Size 105
5.2.6.2 Page Age Threshold (Page_Age_Theshold) Vs False Negatives and Overhead Cost 108
5.2.6.3 Page Size Vs False Negatives 112
5.3 Simulation Results of the MLDR 114
5.3.1 Time versus Heap Size 114
5.3.2 Time versus Heap Size for both of the MLD and MLDR 117
5.3.3Time versus Heap Size and Disk Space Used 118
5.4 Comparing MLD and MLDR to Current Solutions 122
5.4.1 MLD and MLDR versus SWAT 123
5.4.2 MLD and MLDR versus Garbage Collectors 127
5.5 Conclusion 128
Chapter Six Conclusions and Future Works 130

www.manaraa.com

6.0 Introduction 130
6.1 Results 130
6.1.1 Performance 131
6.1.2 Crash Preventing 132
6.1.3 False Negatives 135
6.1.4 False Positives 135
6.1.5 Run-time solution 136
6.2 Simulation Results 137
6.3 Implementation Guidelines 137
6.3 MLD versus MLDR 137
6.4 Future work 138
References 139
Appendices 144

المنارة للاستشارات
www.manaraa.com

LIST OF FIGURES

FIGURE 1: THE PROCESS MAIN PARTS: DATA AND CODE 12
FIGURE 2: OVERALL MECHANISM OF THE HEAPMON(SHETTY ET AL, 2004) 26
FIGURE 3: A REPRESENTATIVE, THOUGH SMALL, STATE OF COMPUTATION (ABDULLAHI AND RINGWOOD, 1998) 29
FIGURE 4: GARBAGE COLLECTION TAXONOMY (ABDULLAHI AND RINGWOOD, 1998) ... 30
FIGURE 5: THE PSEUDO CODE FOR MLD ALGORITHM 39
FIGURE 6: FLOW DIAGRAM FOR MLD ALGORITHM 41
FIGURE 7: MLD ALGORITHM - MAIN FUNCTION 42
FIGURE 8: MLD ALGORITHM-INITIALIZE() FUNCTION 43
FIGURE 9: MLD ALGORITHM-BOOKKEEPING() FUNCTION 44
FIGURE 10: MLD ALGORITHM – MALLOC() FUNCTION 47
FIGURE 11: MLD ALGORITHM- FREE() FUNCTION 48
FIGURE 12: MLD ALGORITHM- ISLEAKYPAGE() - LEAK DETECTION FUNCTION 49
FIGURE 13: MLD ALGORITHM- SWEEPER() – LEAK SWEEPING FUNCTION 50
FIGURE 14: ADDRESS RESOLUTION IN ML-VMS - MAPPING VHTR INTO PHYSICAL ADDRESSES 61
FIGURE 15 : THE BLOCK DIAGRAM OF MLDR ALGORITHM 67
FIGURE 16: MEMORY ALLOCATION FOR MLDR 67
FIGURE 17: MALLOC() FUNCTION FOR THE MLDR 68
FIGURE 18: MEMORY DEALLOCATION OF MLDR 69
FIGURE 19: MLDR WITH AGING 71
FIGURE 20: A BLOCK OF CODE FOR THE MLDR EXAMPLE 72
FIGURE 21: OBJECT RECOVERY FOR MLDR 75
FIGURE 22: MLD PARALLELIZED 86
FIGURE 23: TRACE-DRIVEN SIMULATION PROGRAM - ABSTRACT BLOCK DIAGRAM 91
FIGURE 24: A SNAPSHOT OF ONE OF THE TRACE FILES 96
FIGURE 25: VIRTUAL MEMORY SYSTEM IMPLEMENTED BY DEMAND PAGING 103
FIGURE 26: VIRTUAL MEMORY SYSTEM IMPLEMENTED BY DEMAND PAGING WITH MLD 104
FIGURE 27: EXPERIMENT ONE: TIME VS HEAP SIZE 107
FIGURE 28: PAGE_AGE_THRESHOLD VS NO OF FALSE NEGATIVES AND OVERHEAD 110
FIGURE 29: PAGE SIZE VERSUS NUMBER OF FALSE NEGATIVE OBJECTS 112
FIGURE 30: EXPERIMENT ONE: TIME VS HEAP SIZE 117
FIGURE 31: TIME VERSUS HEAP SIZE COMPARISON BETWEEN MLD AND MLDR 117
FIGURE 32: TIME VERSUS DISK SPACE USED 119
FIGURE 33: PAGE_AGE_THRESHOLD VS NO OF FALSE NEGATIVES AND OVERHEAD 121

www.manaraa.com

LIST OF TABLES

TABLE 1: AUGMENTED PAGE TABLE 45
TABLE 2: MALLOCTABLE, MEMORY ALLOCATION TABLE 48
TABLE 3: VIRTUAL HEAP TABLE(VHT)....60
TABLE 4: A HORIZONTAL SNAPSHOT SLICE OF THE AUGMENTED PAGE TABLE AT TIME (T6) 73
TABLE 5: BENCHMARK USED PARAMETERS 95
TABLE 6: AN EXAMPLE OF ACCUMULATED AMOUNT OF INJECTED LEAK IN TWO TRACE FILES....98
TABLE 7: HEAP SIZE VERSUS TIME....106
TABLE 8: FALSE NEGATIVES VS DIFFERENT VALUES OF PAGE_AGE_THRESHOLD 109
TABLE 9: PAGE SIZE VERSUS FALSE NEGATIVE OBJECTS....112
TABLE 10: HEAP SIZE VERSUS TIME....115
TABLE 11: TIME VERSUS HEAP SIZE AND USED DISK SPACE 119
TABLE 12: PAGE AGE THRESHOLD VERSUS FALSE NEGATIVES, FALSE POSITIVES AND OVERHEAD....120
TABLE 13: THE MLD AND MLDR VERSUS SWAT 126
TABLE 14: MLD VERSUS MLDR 138

www.manaraa.com

A Novel Multi-Layer Virtual Memory System to Solve Memory Leak Problem

Prepared by
Ahmed Ali Otoom

Supervisor
Dr. Mohammad I. Malkawi

Abstract

Memory leak problem is one of the major causes of software failures. Current approaches for solving memory leak problem are not thorough; they either detect memory leak in development environments using source code, re-linking, or recompilation or they only remove unreachable objects in run-time garbage-collected environments. These approaches do not provide a complete run-time solution.

This dissertation provides two new approaches for memory leak detection and recovery in addition to a novel approach for dynamic memory management. The first is memory leak detection (MLD) algorithm. This algorithm reflects both of the physical and virtual behavior of memory allocation and benefits from the hardware support available for tracking physical pages in real memory in order to detect leak in virtual address space. The latter is a memory leak detection and recovery (MLDR) algorithm based on a novel approach for dynamic memory management, a multi-layer virtual memory system (ML-VMS). The ML-VMS reorganizes the currently used dynamic memory management and dynamic memory allocation mechanisms in order to solve or overcome the problem of memory leak

The MLD uses a conservative approach to remove unreachable objects and save address space. It delays possible application crashes due to the lack of virtual memory, but it can not solve the problem of stale objects. The MLDR provides a thorough run-time solution. It handles the problem of false positives, false negatives, and prevents target applications from crash due to the lack of virtual

www.manaraa.com

memory given a well-tuned parameters and that a target application can tolerate an additional overhead. Both approaches are trace-driven simulated in order to provide a proof of concept and show the algorithms' validity. Our approach is compared to some current approaches for memory leak detection and recovery and is shown how it outperforms these approaches in providing a complete run-time solution. The MLDR is recommended for mission critical applications that have to live for a long time and can tolerate a controllable overhead cost.

Key words: multi-layer virtual memory system, memory leak detection, memory leak recovery, virtual memory, memory aging, and dynamic memory allocation.

www.manaraa.com

نظام مبتكر متعدد الطبقات للذاكرة الافتراضية لحل مشكلة تسرب الذاكرة

إعداد

احمد علي عتوم

اشــراف الـدكتور

محمد عصام ملكاوي

Arabic Summary

تعتبر مشكلة تسرب الذاكرة أحد الأسباب الرئيسة في فشل واخفاق البرمجيات. ان الاساليب المتبعة حاليا لحل هذه المشـكلة تعاني من انها غير كامله فهي اما تكتشـف المشكلة خلال مرحلة تطوير البرمجيات باســتخدام البرنامج المصــدري وتســتخدم اعادة الربط والترجمة واما تقوم بحذف الكينونات التي لا يمكن الوصــول اليها اثناء التنفيذ وكما هو متبع في بيئة البرمجيات الجامعة للمهملات وبالتالي فان الحلول الحالية لا تقدم حلا تنفيذيا شاملا للمشكلة.

تقدم هذه الاطروحة طريقتين جديدتين لاكتشـاف المشـكلة ومعالجتها بالاضـافه الى اسلوب مبتكر لنظـام ادارة الـذاكرة الـديناميكي. الطريقة الاولى هي خوارزمية اكتشـــاف الذاكرة المتسرـــبه. تعكس هذه الطريقـة اســـلوب عمل كل من الذاكرة الفيزيائية والذاكرة الافتراضـــية في حجز الكينونات وتســـتفيد من المعدات المتوفرة لمتابعة الصـــفحات الفيزيائية في الذاكرة الحقيقية من اجل اكتشـــاف تسرـب الذاكرة في الذاكرة الافتراضية. والطريقة الثانية هي خوارزمية اكتشاف الذاكرة المتسربه ومعالجتها. تبنى هذه الطريقة على الاسلوب المبتكر المتعدد الطبقات لادارة الذاكرة. يقوم هذا الاسلوب باضافة طبقة جديدة لنظام ادارة الذاكرة الافتراضية بحيث يسمح بحل شامل للمشكلة.

تســـتخدم الطريقة الاولى اســـلوب تحفظي في حذف الكينونات التي لا يمكن الوصـــول اليها من البرنامج من اجل توفير مســـاحة اضـــافية تســـمح للبرنامج باســـتمرار التنفيذ. تطيل هذه الطريقة في عمر البرنامج وتؤخر الفشـل المحتمل نتيجة لعدم توفر المســاحة المطلوبة من الذاكرة لكنها في الوقت نفســة لا تحل من مشكلة

www.manaraa.com

الكينونـات المتعفنـه وهي الكينونـات التي حجزهـا البرنـامج منـذ زمن بعيـد ويحتمـل ان يقوم باستخدامها مرة اخرى. تقدم الطريقة الثانية حلا اشمل للمشكلة فهي تحل مشكلة الكينونات التي لا يمكن الوصـول اليها بالاضـافة الى الكينونات المتعفنه. كما انها تسـتطيع ان تمنع فشـل البرمجيات لانها تضـمن ان المســاحة المطلوبه للحجز متوفرة دائما على اعتبار ان لدينا مســاحة غير متناهية على الاقراص الصــلبة. تم اثبات الطريقتين باسـتخدام التحليل وبرامج المحاكاة وتم مقارنه الاسـلوب الجديد مع بعض الحلول الحالية وبيان مدى نجاح الاسـلوب الجديد في ايجاد حل شـامل للمشـكلة. ينصـح باسـتخدام الاسـلوب الجديد في البرمجيات التي تتطلب ان تبقى في حالة تنفيذ في ولفترة زمنية طويلة.

www.manaraa.com

CHAPTER ONE
INTRODUCTION

This chapter introduces the problem to be addressed, gives justification and purpose of this work, and lists the major questions, contributions and the basic structure of the dissertation.

1.0 Introduction

Due to time-to-market pressures, limited availability of resources, cost reduction, increased demand of software, and software's inherent complexity, software producers often release their products without enough testing and without having their products undergo enough necessary quality assurance constraints. Memory leak is one of the famous notorious memory bugs that dominate the US-CERT and CERT/CC vulnerability reports (US-CERT and CERT/CC, 2007). According to Marcus and Stern (Marcus and Stern, 2000), Software bugs in deployed codes account for as much as 40% of computer system failures. The NIST (National Institute of Standards and Technology, Department of Commerce, 2002) estimated that the software bugs cost the U.S. economy $59.9 billion annually, or 0.6 % of GDP. In this dissertation, we concentrate on one of the major software bugs called memory leak.

In most general terms, memory leak occurs because of 1) unreachable objects: objects exist because the program either unintentionally or maliciously neglects to free heap-allocated objects and therefore these objects are lost and 2) useless objects: a program keeps references to objects that are never used again.

www.manaraa.com

Memory leak is one of the major causes of software failures. Memory leak is mainly serious in long live applications. Memory leak slowly consumes available memory, causing performance degradation and crashing the system. Memory leak is hard to detect since it has few symptoms other than slow and steady increase in memory consumption. Memory leak occurs because some imperative languages place the responsibility of memory allocation and reclamation on the programmers. Programmers must use reclamation methods such as dispose, delete, or free system calls. This explicit store management leads to two common bugs: memory leak and dangling reference problem. Memory leak may lead to running out of virtual address space which results in computation failure. Another consequence of memory leak is thrashing. Memory leak can cause extreme nonlocality of reference as live cells are dispersed over a large virtual address space.

Current approaches for solving memory leak problem are not thorough; they either detect memory leak in development environments as performed by static analysis tools which requires the existence of source code or they garbage collect unreachable objects as performed by garbage collectors. These collectors provide partial solution only in the languages that was designed with garbage collection in mind. There is no complete run-time solution available.

In this dissertation, we explore the concept of aging in the paged physical space for the detection of potential leaks in the virtual space. We also present a

www.manaraa.com

novel approach for dynamic memory management, a multi-layer virtual memory system. The approach reorganizes the dynamic memory allocation space in a manner that enables a complete run-time resolution of memory leak.

1.1 The Statement of the Problem

1.1.1 Problem Definition

Memory leak often results in failures especially in long live processes. A software application may hang or result in an overall system crash or global system performance degradation due to memory leaks. Different businesses can deal with software failures in different ways. In some cases, system administrators simply restart the system whenever the memory leaks to a point where a crash is eminent or performance degrades beyond an acceptable level. Systems with critical applications can not tolerate the cost of frequent shutdowns or performance degradation. The consequences of unresolved memory leaks in real-time systems can have a direct impact on human safety, security and business sustainability.

Currently, there are some solutions to memory leak problem, but these solutions are not thorough, suffer from performance degradation, and there is no complete run-time solution. The current dynamic memory management mechanisms allow these problems to exist and add constraints to potential solutions. Reorganizing the current dynamic memory management system into a multi-layer virtual memory system provides a basis for a fundamental solution to memory leak and other problems.

www.manaraa.com

The purpose of this study is to present a novel approach for dynamic memory management in a multi-layer virtual memory system. This approach reorganizes the currently used dynamic memory management and dynamic memory allocation mechanisms in order to solve or overcome the problem of memory leak. It also explores the concept of aging in the paged physical space as a method of detection of potential leaks in the virtual space.

1.1.2 Dissertation Questions

The major dissertation questions are:

1. Why does the currently used dynamic memory management system permit the memory leak issue to exist?
2. What are the available solutions to memory leak problem and what are their shortcomings?
3. How can we use the concept of page aging in physical space as a means for detecting memory leaks?
4. Why aging in physical space is preferred over aging in the virtual space?
5. What is Multi-Layer Virtual Memory System?
6. What are the guidelines that facilitate the implementation of the Multi-Layer Virtual Memory System?
7. How does the Multi-Layer Virtual Memory System help in resolving memory leak problems?

www.manaraa.com

1.1.3 Dissertation Scope

This dissertation will focus on solving the problem of memory leak. The Multi-Layer Virtual Memory System and aging will be investigated in the extent of its relation to the resolution of the memory leak problem.

1.1.4 Dissertation Contributions

Our dissertation adds the following contributions:

1. Present and develop a new approach for memory leak detection using the concept of aging in the physical memory system to identify and solve memory leaks in the virtual memory system, thus allowing the algorithm to utilize the hardware available for virtual memory organization as shown in chapter 3.
2. Present and develop a novel approach for dynamic memory management, a multi-layer virtual memory system as shown in chapter 4.
3. Present a new approach for memory leak detection and recovery based on the Multi-Layer Virtual Memory System as shown in chapter 4.
4. Provide guidelines that facilitate the implementation of both approaches: memory leak detection with aging and memory leak detection and recovery based on the new structure of the Multi-Layer Virtual Memory System. These guidelines are explained in chapters 3 and 4 respectively.

The memory leak detection and recovery based on the Multi-Layer Virtual Memory System approach that we present in this dissertation provides the following contributions:

www.manaraa.com

a) Solves the problem of both of the unreachable and useless objects.

b) Handles the problem of false positives.

c) Provides run-time solution for memory leak detection and recovery whereas most of previous approaches either detect memory leak in development environment or remove only unreachable objects in run-time environment as performed by garbage collectors.

d) Prevents programs from crashing by guaranteeing that the requested space on the virtual space is always available by moving potential leaky objects to disk that presumably has an unlimited space.

e) Although there is a performance penalty cost, discussed in chapter 5, that will be paid by the algorithm, this penalty is kept to the minimum by:

1) The performance penalty cost will never be paid (i.e deferred) until either the program exceeds a certain threshold in virtual address space or there is no available memory to be allocated and the application is about to crash.

2) The algorithm consists of modular parts allowing optimal future implementations to these modules which reduce the overall performance penalty cost, as a result.

3) The algorithm utilizes tunable parameters that are designed in order to reduce the cost.

4) Performance evaluation of the algorithms and methods is presented in the dissertation.

www.manaraa.com

5) Parallel programming and data partition is suggested in chapter 5 and left to be explored as a future work.

1.2 Dissertation Methodology

Our dissertation passes through the following phases:

- Identifying the shortcomings of the current solutions of memory leak problem and how it is related to the current dynamic memory management system.
- Developing two descriptive algorithms. The first is for the new approach of memory leak detection using aging in physical memory space. The latter is a memory leak detection and recovery that utilizes the novel dynamic memory management, multi-layer virtual memory system to solve memory leak problems.
- Evaluating the performance of both approaches using analysis and a trace-driven simulation program. The simulation program helps in algorithms' validation and verification and provides a proof of concept.
- Analyzing the results and concluding.

1.3 Organization of Dissertation

This dissertation consists of six chapters that build on each other. Chapters three and four along with the analysis in chapter five are the core material being used to publish the following papers:

a. Memory Leak Detection Using Aging in physical Memory Space
b. A Novel Multi-Layer Virtual Memory System for Solving Memory Leak Problem.

www.manaraa.com

The following is a detailed outline for each chapter:

- Chapter I: Introduction. This chapter introduces the problem to be addressed, gives a justification and purpose of this work, and lists the basic structure of the dissertation.
- Chapter II. Dynamic Memory Allocation and Current Approaches for Solving Memory Leak Problem. This chapter overviews the dynamic memory allocation process, discusses and surveys the existing mechanisms for detecting and solving memory leak problem and lists the shortcomings of these available solutions. Readers who are familiar with dynamic memory allocation and current approaches to solving memory leak and their shortcomings can skip this chapter.
- Chapter III. A New Approach for Memory Leak Detection Using Aging in Physical Memory Space. In this chapter, we exploit the concept of aging in physical memory space to develop a new approach for memory leak detection in virtual address space using aging in physical memory space.
- Chapter IV. A Multi-Layer Virtual Memory System. In this chapter, we describe the Multi-Layer Virtual Memory System. We show how the Multi-Layer Virtual Memory System along with a new memory leak detection and recovery algorithm allow for efficient resolution of memory leak problem. We also provide some guidelines that facilitate the implementation of this new structure.

www.manaraa.com

- Chapter V. Performance Evaluation and Simulation. This chapter analyzes the performance of the presented algorithms in terms of access time and complexity. It shows, through analysis and a trace-driven simulation program, how some of the performance measures can be enhanced. A separate simulation model is provided for each algorithm. Along with performance analysis, simulation programs validate the new approaches and provide a proof of concept. This chapter also compares both algorithms to current memory leak solutions and shows how the new approach outperforms the current approaches in providing a complete run-time solution.
- Chapter VI. Conclusions and Future Works. This chapter provides basic conclusions as well as directions for future research.

1.4 Conclusion

This chapter introduces the problem which is summarized in presenting a novel approach for dynamic memory management that will reorganize the currently used dynamic memory management and dynamic memory allocation mechanisms in order to solve or overcome the problem of memory leak. It also gives justifications and purpose of this work, and lists the major dissertation questions, contributions and the basic structure of the dissertation

www.manaraa.com

Chapter Two
Dynamic Memory Allocation and Current Approaches for Solving Memory Leak Problem

This chapter overviews the dynamic memory allocation process, discusses the existing mechanisms for detecting and solving memory leak problem and lists the shortcomings of the available memory leak solutions.

2.0 Introduction

Memory leak is particularly serious in long live applications. It slowly consumes available memory, causing performance degradation and eventually crashing the system. Memory leak is among the hardest bugs to detect since it has few symptoms other than slow and steady increase in memory consumption (Xie and Aiken, 2005).

In the next sections, we review the dynamic memory allocation process. We show how the current dynamic memory allocation process allows the problem of memory leak to occur. Then, we review various approaches used for memory leak detection and recovery. After that, we list the shortcomings of these approaches and finally, we summarize the chapter.

2.1.0 Dynamic Memory Allocation

Dynamic memory allocation is the allocation of memory storage for use during the runtime of the program. A dynamically allocated object remains allocated until it is freed explicitly by a programmer or by a garbage collector

www.manaraa.com

(Wikipedia, 2007). So, dynamic memory allocation is the allocation of memory for the use of a computer program during runtime. Figure 1 shows how memory is distributed among many pieces of data and code.

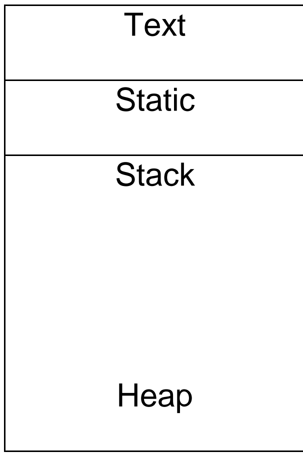


FIGURE 1: THE PROCESS MAIN PARTS: DATA AND CODE

Each program can be divided into text and data. The text is the actual program code and the data is the information that the text (code) operates on. Data can be further subdivided into static, stack, and heap.

Where:

Static: storage space is compiled into the program. Static variables are allocated during program loading and deallocated when program exits. For more information about pointers and memory, refer to "Pointer and Memory" by Parlante (Parlante, 2000).

```
/* global variables are static data */
int x[10]; /* x is stored in static area */
main () {
```

www.manaraa.com

stack: local variables and parameters of function calls are managed by the system in the stack space during runtime. Stack variables are all allocated when entering the declaring block and deallocated when exitting.

```
void dosomthing() {
float y; /* y is stored in the stack *
```

heap: dynamic allocations via new or malloc are allocated also at runtime and stored in the heap space. Heap objects are allocated with malloc() or new() and deallocated with free() or delete().

```
main()
char * str; /* the address of str is stored on the stack */
/* Allocate a string of 5 bytes on the heap. */
str = (char *) malloc(5);
...
/* de-allocate the heap memory*/
(void) free(string);
```

A dynamically allocated object remains allocated until it is deallocated explicitly, either by the programmer or by a garbage collector. Both stack and heap management complicate memory management because these areas are dynamic. Of the two, stack management is much simpler than heap management because the stack space is allocated by function call frames in a regular Last In First Out (LIFO) pattern. Two factors make up heap management (Kline, 2007):

1. how to manage the allocation of variable-sized chunks of contiguous memory
2. how to manage the random order of allocations and deallocations

www.manaraa.com

2.1.1 Dynamic Memory Allocation Strategies

Several allocation strategies are used to find an available hole (free area) for a new allocation. Some of these strategies are cited in (Kline, 2007):

1. First Fit: look for the first hole sufficiently large, starting from the beginning
2. Next Fit: look for the first hole sufficiently large, but start from where you left off previously.
3. Best Fit: look for a hole of smallest possible size
4. Worst Fit: look for a hole of largest possible size

The selection of a specific strategy has direct impact on performance as well as on external and internal fragmentation. Wilson and others (Wilson et al, 1995) discuss these issues in their survey “Dynamic Storage Allocation: A Survey and Critical Review”.

2.1.2.0 Dynamic Memory Allocation Algorithms

The main problem for most algorithms is to avoid both internal and external fragmentation while keeping allocation and deallocation efficient. Various approaches are being used by memory allocation algorithms such as fixed-size-blocks allocation, buddy blocks allocation, and heap-based memory allocation (WikiPedia, 2007):

www.manaraa.com

2.1.2.1 Fixed-Size-Blocks Allocation

This solution uses a LIFO linked list of fixed size blocks of memory.

2.1.2.2 Buddy Blocks Allocation

This solution uses a binary buddy block allocator. Memory is allocated from a large block of memory that is a power of two in size. If a block is more than twice as large as desired, it is broken in two. One is selected and the process is repeated recursively until the block is large enough. All the buddies of a particular size are kept in a sorted linked list or tree. When a block is freed, it is compared to its buddy. If they are both free, they are combined and placed in the next-largest size buddy-block list. The allocator starts with the smallest sufficiently large block in order to avoid breaking blocks.

2.1.2.3 Heap-based Memory Allocation

Memory is allocated from a large pool of unused memory area called the heap. The size of memory allocation can be determined at runtime. Allocated regions are accessed via a reference. A free list is a linked list that connects unallocated regions of the heap together. A deallocated region is added to the free list, an allocated region is removed from the free list.

2.1.3 Dynamic Memory Allocation Functions

Gary Watson (Gary, 2007) provides a detailed list for memory allocation commands and how they can be used in C language. The most frequently used commands are described below:

www.manaraa.com

1. malloc fucntion: has the general form

void *malloc (unsigned int size)

This function allocates a specified amount of memory in bytes. It will return a pointer to the beginning of the allocated space.

2. calloc function: has the general form

void *calloc (unsigned int number, unsigned intsize)

The calloc routine allocates a certain number of items, each of size bytes, and returns a pointer to the space.

3. realloc function: has the general form

void *realloc (void *old_pnt, unsigned int new_size)

The realloc function expands or shrinks the memory allocation in old_pnt to new_size number of bytes. Realloc copies as much of the information from old_pnt as it can into the new_pnt space it returns, up to new_size bytes. If there is a problem allocating this memory, a 0L value will be returned. If the old_pnt is 0L then realloc will do the equivalent of a malloc (new_size). If new_size is 0 and old_pnt is not 0L, then it will do the equivalent of free (old_pnt) and will return 0L.

4. free fucntion: has the general form

void free (void *pnt)

www.manaraa.com

The free routine releases allocation in pnt which was returned by malloc, calloc, or realloc back to the heap. This allows other parts of the program to re-use memory that is not needed anymore. It guarantees that the process does not grow too big and swallow a large portion of the system resources.

2.1.4 The Access State of Referencing a Dynamically Allocated Object

When a pointer is used to reference a dynamically allocated object, the access operation will result in one of the following states (success, illegal access, and corruption):

1) *success*, if the pointer points to a reachable live object.

2) *illegal access*, if the object has been deallocated and the pointer contains the address that is not allocated to some other object.

3) *corruption*, if the object has been deallocated and the pointer contains the address that is allocated to some other object. The pointer will misguidedly access an object that is not supposed to access.

2.1.5 Dynamic Memory Allocation and Memory Leak

Memory leak is one of the major causes of software failures. Memory leak occurs because some imperative languages (e.g C, C++, Pascal, etc) place the responsibility of memory allocation and reclamation on the programmers. Programmers must use reclamation methods such as dispose, delete, or free system calls.

www.manaraa.com

This explicit store management leads to two common bugs: i) incompleteness (also called memory leaks) and ii) unsoundness (also called the dangling reference problem). Memory leak may lead to running out of virtual

address space which results in computation failure (Abdullahi and Ringwood, 1998). On systems where all memory is in RAM, memory leak will result in an immediate failure (Wikipedia, 2007). Another consequence of memory leak is thrashing. Following Denning (Denning, 1968), a computation with page faults every few instructions is said to thrash. Memory leaks can cause extreme nonlocality of reference as live cells are dispersed over a large virtual address space.

Explicit dynamic memory allocation in some languages like C and C++ leads to memory leak problem. Memory leak occurs when a program fails to deallocate the unwanted previously allocated memory chunks. In some garbage collected languages, memory leak occurs when a program keeps reference to a memory chunk that will never be accessed in the future.

www.manaraa.com

2.2 Approaches for memory leak detection

In order to detect software bugs such as memory leak, many approaches have been proposed for dynamic code monitoring. The most commonly used approaches are assertions and static analysis, dynamic checkers, hardware-assisted solutions, and garbage collectors. We review these approaches in the following sub-sections.

2.2.1 Assertions and Static Analysis

Assertions are inserted by programmers to perform required checks at certain places. The program aborts if assertions are violated (Zhou et al, 2005). Adding annotation (Evans, 1996) to the source code is used to make assumptions

about memory management explicit at interface points. Examples of this approach include explicit model checking (Musuvathi et al. 2002; Stern and Dill 1995) and program analysis (Choi

et al. 2002; Engler and Ashcraft 2003; Hallem et al. 2002). Most static tools require significant involvement of the programmer to write specifications or annotate programs. Annotation is used by a static checker to make fixing memory management problems in a more systematic and goal-directed manner. An efficient use of the static checker should detect a broad class of errors that includes misuse of pointers,

www.manaraa.com

use of dead storage, memory leaks, and dangerous aliasing. This approach does not eliminate the need for run-time testing nor does it detect all errors.

Static analysis tools can find leaks before running the program by analyzing source code and thus do not cause any runtime overhead. An example of these tools is PREfix(Bush et al, 2000). This tool simulates the execution of individual functions. It derives the information directly from the source code rather than acquired through user annotations. Another static analysis tool, called Clouseau, is presented in (Heine and Lam, 2003). This tool implements a practical ownership model of memory management. In this model, every object is pointed to by one and only one owning pointer which has the exclusive right to delete the object or to pass the right of ownership to another pointer. Static analysis tools lack dynamic information which leads to conservative results including false positives and they do not find all leaks.

www.manaraa.com

2.2.2 Dynamic checkers

Dynamic checkers are automated tools that detect common bugs at run time, with instrumentation inserted in the code that monitors invariants and reports violations as errors. The analysis of this approach is based on actual execution paths and accurate values of variables and aliasing information. Some examples of these tools are DIDUCE (Hangal and Lam, 2002), Purify (Hastings and Joyce, 1992), Valgrind (Nethercote and Seward 2003), StackGuard (Cowan et al, 1998), Insure++ (Parasoft Insure++, 2007), and Eraser (Savage et al, 1997). Some of these tools can detect memory leaks, memory corruption, buffer overflow, and data races. These tools often use compilers and code rewriting tools. While this approach is promising it suffers from: i) dynamic aliasing especially in C and C++, ii) high run-time overhead, iii) hard-coded bug detection functionality, iv) language specificity, and v) difficulty to work with low-level code(Zhou et al, 2005).

One of the low-overhead memory leak detection tools is SWAT (Chilimbi and Hauswirth, 2004). SWAT reports 'stale' heap objects that have not been accessed for a user definable, long time as leaks. SWAT has been used by several product groups at Microsoft and has proved effective at detecting leaks with a false positive rate less than 10%. According to Chilimbi and Hauswirth, SWAT has three advantages over Purify tool from Rational.

www.manaraa.com

First, it uses unique strategy of identifying leaks based on object staleness. Second, since it uses sampling, the overhead is significantly lower than Purify (5% Vs 3-5X). Third, SWAT provides an indication of which program instruction last accessed by the leaked object.

JRocket, JVM, is a real-time memory leak detection tool that utilizes a trend analysis of memory growth to detect memory leak. Each time there is a garbage collection in the JVM, JRocket sends trend data to the memory leak detector. The trend analysis shows the common object types in the heap and the rate at which memory of these objects are growing. The longer the trend analysis is, the more reliable the trend is. The JRocket can be used to detect memory leak, find out what is leaking, and then drill down to what is causing the leak in the code. One of the limitations of this tool is that it uses Java-based communication protocol which means that JRocket will create new objects to send information over to the management console. This may not be ideal, as the system probably low on memory because of memory leak. Another limitation is that when there are large amounts of information to send over, there is a risk of losing the connection to the management console because of timeout problem (Ostlund, 2005).

JProbe from Quest Software (Quest Software, 2007) is another tool that tracks memory growth to detect memory leak in Java applications. It views memory usage for specific classes, methods and/or instances. It's up to the users to determine the impact of memory leak or code change (JProbe, 2007).

www.manaraa.com

The debug memory allocation or *dmalloc* library has been designed to provide powerful debugging facilities such as memory-leak tracking, fence-post write detection, file/line number reporting, and general logging of statistics. The dmalloc library replaces the heap library calls normally found in the system libraries with its own versions. When a call is made to malloc (for example), the dmalloc's

version of the memory allocation function is called. The dmalloc library keeps track of a number of pieces of debugging information about pointer including: where it was allocated, exactly how much memory was requested, when the call was made, etc. This information can then be verified when the pointer is freed or reallocated and the details can be logged on any errors (Watson, 2004).

2.2.3 Hardware-assisted solutions

In this subsection, we review two approaches that use software/hardware solutions: hardware-assisted watch points and HeapMon.

In hardware-assisted watch points, a simple hardware is used to support watching user selected memory locations. When a watched location is accessed, an exception is generated and handled by the interactive debugger. In general, dynamic monitoring is classified into two categories: Code-Controlled monitoring (CCM) and Location-Controlled Monitoring (LCM).

www.manaraa.com

In CCM, monitoring is performed at special points in programs as done by assertion and most of dynamic checkers. In LCM, monitoring is associated with memory locations as in hardware-assisted watch points and Intelligent Watcher (iWatcher). LCM has two advantages over CCM. First, LCM monitors all access to memory locations using all variable and pointer names, whereas CCM may miss some accesses due to aliasing problems. Latter, LCM monitors only those instructions that truly access the watched location, whereas CCM monitors many unnecessary points. The main advantage of CCM is that it does not need hardware support, while the LCM needs

it. IWatcher is demonstrated, by simulation, to detect buffer overflow, memory leaks, accessing freed locations, stack smashing, and invariant violations (Zhou et al, 2005).

HeapMon (Shetty et al, 2004) is another novel software/hardware approach to detecting memory bugs such as reads from initialized or unallocated memory locations. Memory leak is detected if, at the end of program execution, there are words in the heap region that are still in one of the allocated states. HeapMon relies on a helper thread that runs on a separate processor in a Chip Multi-Processor (CMP) system. The Thread associates a state bit with each word on the heap. The state bit indicates whether the word is unallocated, allocated but uninitialized, or allocated and initialized.

www.manaraa.com

The state bits are updated when the word is allocated, initialized, or deallocated. Bugs are detected as illegal operations, such as writes to unallocated regions and reads from unallocated or uninitialized memory regions. These bugs are logged to enable developers to pinpoint the bug's nature and location. The hardware support consists of an extra state bit for each cached word, communication queues between the application thread and the helper thread, and a small private cache for the helper thread. The main advantages of HeapMon are: i) no human intervention is needed, either to insert breakpoints or watch points ii)the bug detector is written in software, iii) no compiler is needed beyond relinking the application with a static library or running it with dynamically-linked library, and iv) the overhead is low. The storage overhead of this approach is 3.1% of the cache size and 6.2% of the allocated heap memory size. The execution overhead is 8% on average and 26% on the worst case.

www.manaraa.com

Figure 2 shows the general mechanism of HeapMon checking. Each heap memory request proceeds in three steps. First, the request from the main processor is forwarded to the main memory (step 1a) and to the HeapMon thread (step 1b).

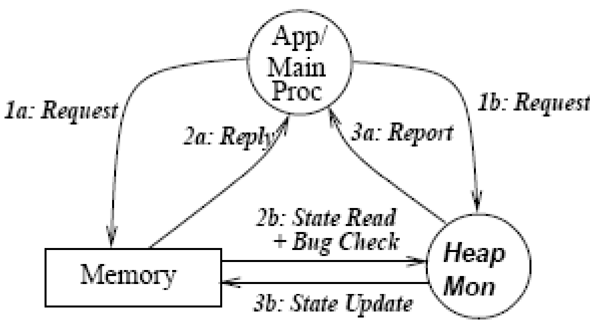


FIGURE 2: OVERALL MECHANISM OF THE HEAPMON(SHETTY ET AL, 2004)

Requests are events of interest: memory allocation, memory deallocation, and heap memory access. Extra information, such as process id, is piggybacked to the HeapMon to perform the necessary checks. On a read request, the memory replies with data (step 2a) and the tag processor reads the state for the request word (step 2b) and performs a bug check whether the request type is allowed for the current state of the word. The result of the bug check is reported to the main processor (step 3a) and the state is updated if necessary (step 3b).

المنارة للاستشارات
www.manaraa.com

2.2.4 Garbage collectors and memory leak

Garbage Collection has been an integral part of many programming languages such as Java, Lisp, Smalltalk, Eiffel, Haskell, ML, Scheme, and Modula-3, and has been in use since the early 1960s. There are many indisputable benefits of garbage collectors including increased reliability, decoupling memory

management from class interface design, and less development time. Dangling pointers and memory leaks do not occur in Java. However, garbage collection has some performance impacts, pauses, configuration complexity, and nondeterministic finalization (Goetz, 2003). Languages that use garbage collectors are not immune to memory leaks. Although the garbage collector can recover memory that has become unreachable and therefore logically useless, it cannot free memory that is still reachable and therefore potentially still useful (Wikipedia,2007). A leak detector that is based on a type accurate garbage collection finds more memory leaks than a leak detector based on conservative garbage collection.(Hirzel and Diwan, 2000)

www.manaraa.com

Garbage collection is an inevitable consequence of programming languages that use dynamic data structures. With dynamic structures, the state of computation can be considered as many-rooted, directed graph called the computation graph (figure 3). The roots are the entry points to the graph. The internal vertices are realized as cells, contiguous segments of memory. A cell is a base address from which offsets can be accessed. In object-oriented languages, cells are objects. If cells are not of a fixed size they often have a terminator or an indicator of length in the cell's header. The indicator is the number of bytes in the cell or a pointer to the last byte in the cell. Edges are realized by store address fields within cells. Cells referenced directly or indirectly from the root are called reachable, accessible, or live. As computation processes, addition and deletion of

www.manaraa.com

roots, vertices, and edges modifies the graph. As a result, some portions of the graph become unreachable, inaccessible, or dead. These disconnected subgraphs make no contribution to the computation and known as garbage. In figure 3, the garbage cell is denoted by a filled circle and the accessible cell by unfilled circle. Without reuse, the finite store for allocating new vertices diminishes to zero. The garbage collector is a process by which the area occupied by garbage is reused (Abdullahi and Ringwood, 1998).

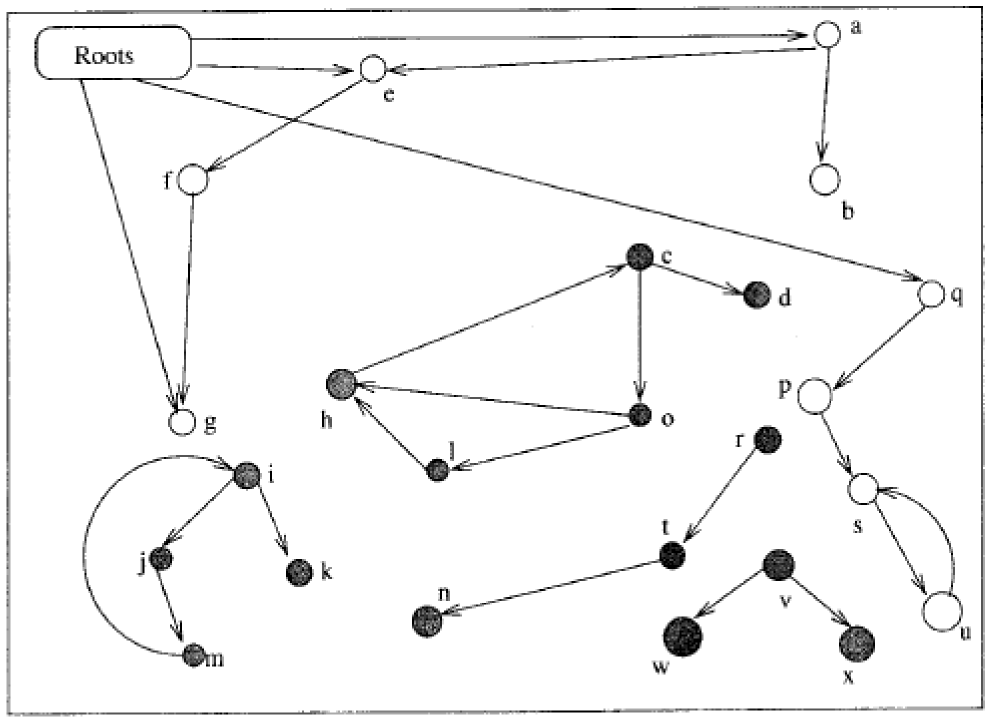


FIGURE 3: A REPRESENTATIVE, THOUGH SMALL, STATE OF COMPUTATION (ABDULLAHI AND RINGWOOD, 1998)

Dijkstra et al introduce two useful abstractions to the study of garbage collection. The mutator and the collector. The mutator abstracts the process that performs the computation and allocation. The collector abstracts the process that

www.manaraa.com

reclaims garbage (Dijkstra et al, 1978). This abstraction, along with an excellent survey about garbage collection techniques in uniprocessor environment, is also mentioned in literature as a two-phase abstraction: garbage detection phase and garbage reclamation phase (Wilson, 1992).

In his review, Abdullahi, lists the taxonomy of garbage collection (figure 4). The collector process is divided into two subprocesses: Identification, I, and Reclamation, R.

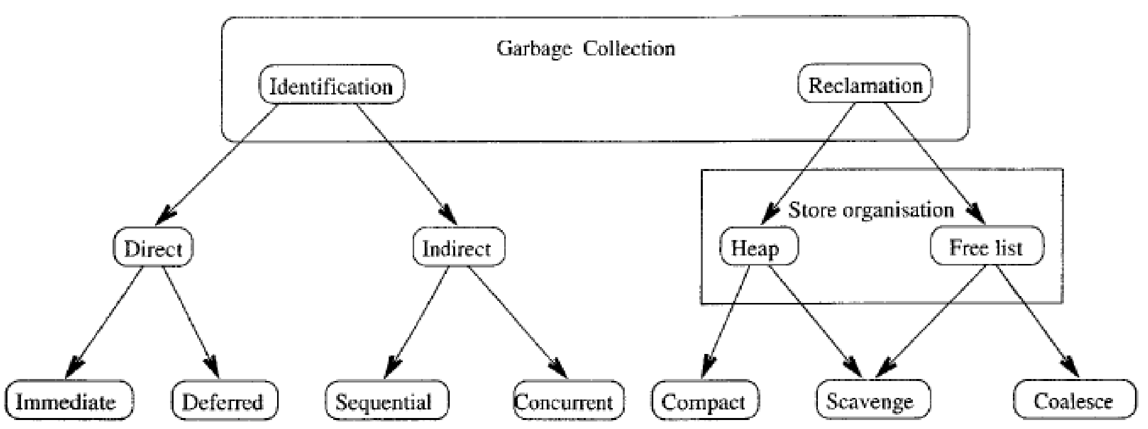


FIGURE 4: GARBAGE COLLECTION TAXONOMY (ABDULLAHI AND RINGWOOD, 1998)

Two classes of identification are identified: direct and indirect. Direct identification (also called reference counting) identifies cells that have no reference to them. Indirect identification identifies live cells by tracing them from the roots - what remains must be unallocated or garbage. Reclamation is classified depending on how free store is managed; If it can be managed as a free-list or a heap. If managed as a free-list contiguous garbage can be coalesced to form larger cells. If managed as a heap, a single reference, the top of the heap, indicates the

www.manaraa.com

division between allocated and unallocated store (Abdullahi and Ringwood, 1998). More references about garabage collection and garbage collection techniques can be found on the "Garbage collection bibliography" (Jones, 2003).

2.3 Shortcomings of current approaches

Static analysis tools can find leaks before running the program which adds no running time overhead, but this limits them to be used in development environment where source code is available. This approach does not eliminate the need for run-time testing nor does it detect all errors. Static tools have no value when source code is not available. More over, static analysis tools lack dynamic information which leads to conservative results including false positives and they do not find all leaks.

Dynamic checkers often use compilers and code rewriting tools. According to Zhou et al (Zhou et al, 2005) this approach suffers from: i) dynamic aliasing especially in C and C++, ii) high run-time overhead, iii) hard-coded bug detection functionality, iv) language specificity, and v) difficulty to work with low-level code.

Despite the clear benefits of garbage collection such as increased reliability, decoupling memory management from class interface design, and less development time, garbage collectors work only with languages designed with garbage collection in mind. More over, languages that use garbage collectors are not immune to memory leaks. Garbage collector can recover memory that has become unreachable and therefore logically useless but, it cannot free memory that is still reachable and therefore potentially still useful.

www.manaraa.com

2.4 Conclusion

Memory leak occurs as a result of using dynamic memory allocation where some imperative languages (e.g C, C++, Pascal, etc) place the responsibility of memory allocation and reclamation on the programmers. In garbage collected languages, memory leak occurs when the programs keep a reference to an object that will never be used in the future.

The most commonly used approaches for memory leak detection and recovery are assertions and static analysis, dynamic checkers, hardware-assisted solutions, and garbage collectors. These approaches suffer from several shortcomings. Static analysis tools lack dynamic information which leads to conservative results including false positives and they do not find all leaks. Dynamic checkers often use compilers and code rewriting tools. They suffer from: i) dynamic aliasing, ii) high run-time overhead, iii) hard-coded bug detection functionality, iv) language specificity, and v) difficulty to work with low-level code. Hardware-assisted solutions are costly to implement and are used to enable developers to pinpoint the bug's nature and location with certain overhead cost. Garbage collectors are limited to languages designed with garbage collection in mind and can only remove unreachable objects in run-time environment.

Most of the available solutions are not thorough, suffer from performance degradation, and do not provide a complete run-time solution. We alleviate some of these problems in the next chapter, a new approach for memory leak detection using aging in physical memory space. In chapter 4, we provide a complete run-time solution by introducing another new approach for memory leak detection and recovery based on the introduced novel structure, the ML-VMS

www.manaraa.com

Chapter Three
A New Approach for Memory Leak Detection (MLD) Using Aging in Physical Memory Space

3.0 Introduction

Software aging refers to resource contention issues that can cause performance degradation or can cause systems to hang, panic, or crash. Software aging can include memory leaks, unreleased file locks, accumulation of unterminated threads, data corruption/round-off accrual, file space fragmentation, and others (Gross et al, 2002).

Aging, in this dissertation, is related to the time a piece of memory object remains untouched. A leaky object by definition will begin to age since it will no longer be accessed by any application program. Aging, in this context, can then be used to detect memory leakage. Memory leaks and aging refer to objects in the heap virtual space. Detecting the age and the leaky status of an object in the virtual space is time consuming. In this dissertation, we exploit the status of a memory object in reference to the physical memory space in order to detect the age of an object and hence the leak in virtual address space. A new approach for memory leak detection is presented based on the aging of an object in the physical space. This approach for memory leak detection provides the following contributions:

1. Memory leak in the virtual space is detected based on aging in the physical space, thus allowing the algorithm to utilize the hardware available for virtual memory organization.

www.manaraa.com

2. MLD provides a conservative run-time solution for memory leak. This is similar to conservative garbage collectors in the sense that it deals with unreachable objects but it detects memory leak based on physical memory aging.
3. The performance penalty cost (PPC) that will be paid by the algorithm is kept to minimum using the following techiques:
   a. The PPC will never be paid (i.e deferred) until the program exceeds a certain threshold in virtual address space, i.e., the heap size grows beyond a certain limit.
   b. The algorithm consists of modular parts allowing optimal future implementations to these modules which reduce the overall PPC, as a result.
   c. Parallel threads may be utilized as a tool for further performance enhancement (See Chapter 5)

In the next sub-sections, we present our approach for memory leak detection. In this chapter, we present various performance metrics such as program crash delay, false positives, false negatives, and telemetry. The chapter will end with some conclusions.

www.manaraa.com

3.1 Memory Leak Detection (MLD) Using Aging in Physical Memory

Once a chunk of memory is leaked, it will no longer be accessed by the application program. Hence, a leaky memory will begin to age. Aging, in this context, is related to the time a piece of memory remains untouched. Memory

aging can then be used to detect leakage. In classic computer systems, memory allocation is done both at virtual and physical levels. A memory leakage in the virtual space will render the corresponding mapped physical memory in a "page-out status". A page-out status makes a page in the physical memory a target for the replacement policy. Going backward, a paged out page can correspond, but not necessarily, to a leaked chunk in the virtual space. The age of a physical page is the time elapsed since the page is swapped out of the physical to the virtual space. More precisely, the age begins to accumulate from the time a page is marked by the replacement policy as a target for replacement. Note that a page may remain in the physical space for a long time after it has been marked as a replaceable page.

www.manaraa.com

The aging in the virtual address space for an application is correspondent to a similar aging in the physical address space. Memory aging is measured as a time elapsed since i) the last dereference to the memory chunk in the heap by one of its pointers, ii) a page was marked for replacement in the physical space, or iii) a page was last swapped to virtual space. Thus, the memory detection mechanism can rely on either the virtual space or physical space. A memory chunk will be considered a candidate leak if the age of its corresponding memory page exceeds a certain limit (threshold). This threshold can be either user defined or tuned by a telemetry tool.

The memory leak detection (MLD) algorithm can rely on memory aging either in the virtual or physical space. Aging in the virtual space is more accurate

www.manaraa.com

than aging in the physical space. The physical space is much smaller than the virtual address space, and pages can age more quickly, and become candidates for replacement. Also, aging in the physical space depends on the replacement policy used by the OS. Different replacement policies (LRU, FIFO, OPT, LFU(Silberschatz et al,2005)) select different pages for replacement at any given time. Furthermore, pages in physical memory may become candidates for replacement because of the behavior of other processes, given that global replacement policies are used.

On the other hand, aging in the virtual space is process dependent. A chunk of memory continues to age as long as no reference is made to this chunk. The main problem of tracing the age of memory pieces in the virtual space is the overhead associated with keeping track of the age and scanning for older chunks in memory. This is particularly true when the list of allocated memory is relatively large. Using the physical memory allocation for leak detection has the advantage of hardware support in most virtual memory systems, and thus the detection time overhead can be negligible.

Another problem with using physical memory is that a page in physical memory may correspond to several chunks in the virtual space. Several chunks, whose size is smaller than a page, are mapped into one physical page. Hence, if only one chunk in the virtual space is active, while the other chunks have leaked, then the corresponding physical page remains active and the leaks in that page will not be detected. Using a smaller page size can relax this problem,

www.manaraa.com

but does not resolve it. Further optimization can be applied to relax this problem. However, the necessity of applying optimizations will be dictated by the experimental measurement of how serious this problem is.

In order to have a more accurate detection algorithm, we propose a new algorithm that reflects both the physical and virtual behavior of memory allocation. We will benefit from the hardware support available for tracking physical pages in real memory. We list the MLD algorithm pseudo code next. After that, we explain it block by block.

www.manaraa.com

3.2 MLD Algorithm Pseudo Code

Figure 5 shows the pseudo code for MLD algorithm

```
For every process that wants to exploit MLD algorithm //this process becomes a monitored process
Begin
          Initialize()
           Perform the following functions concurrently
              Bookkeeping()
              Run the Sweeper() if it is started

End
Initialize () {
          Set Heap_Size_Threshold to input value
          Set Sweeper_Sleep_Time to input value
          Set Page_Age_Threshold  to initial input value
}
Bookkeeping(){
          Reset time stamp field for any new entry added to page table.
          For every victim page selected from the mapped physical space
                    Time stamp the corresponding page in the virtual address space by setting the time
                    -stamp  field in the page table to the current time.
          For every paged-in page
                    Reset the Time stamp field.
}
Sweeper(){
          //use conservative approach for garbage collection
          while(true){
          For each page in the page table of the monitored process{
                    If isLeakyPage(page_number, current_time, Page_Age_Threshold (τ)){
                              Mark all reachable chunks from (static, stack, and registers) as live objects
                              //use conservative approach//consider all a like pointers as pointers
                                 Garbage Collect unreachable chunks(dead)
                                 Remove Garbage Collected chunks entries from mallocTable

                    }
          }
         sleep(Sweeper_Sleep_Time)
         }
}
bool isleakyPage (page_number, current_time, Page_Age_Threshold (τ)) {
  if (time stamp of page_number>0){
               Age = current time – time stamp of page
               If Age > Page_Age_Threshold(τ) then return true
  }
  Return false
}
//Sweeper() process is started by the memory allocation function
//MallocTable , a table that tracks allocations and deallocations, is maintained by memory allocation
function
```

FIGURE 5: THE PSEUDO CODE FOR MLD ALGORITHM

www.manaraa.com

3.3 MLD Algorithm Explanation

In this sub section, we explain the MLD algorithm and provide guidelines that facilitate its implementation.

Figure 6 represents the block diagram of MLD algorithm. The diagram shows the major components required to implement the algorithm. Next, we describe these components and explain how the algorithm works.

www.manaraa.com

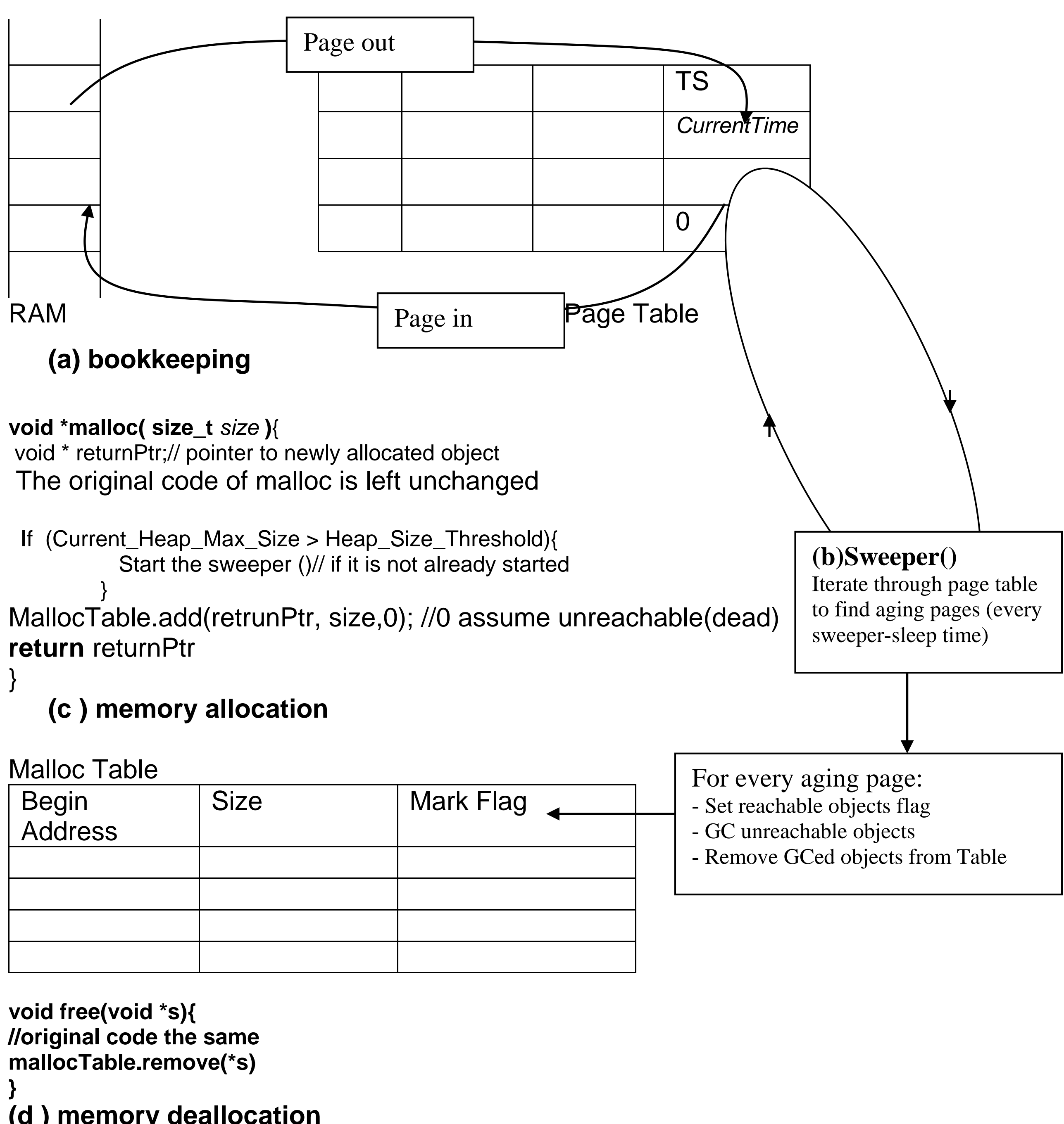


FIGURE 6: FLOW DIAGRAM FOR MLD ALGORITHM

المنارة للاستشارات
www.manaraa.com

3.3.1 MLD Main Function

```
For every process that wants to exploit MLD algorithm //this process becomes
monitored process
Begin
        Initialize()
          Perform the following functions concurrently
           Bookkeeping()
           Run the Sweeper() if it is started

End
```

FIGURE 7: MLD ALGORITHM - MAIN FUNCTION

The “main function” of the algorithm, figure 7, keeps iterating over monitored processes. The monitored processes are the processes chosen by the operating system or the system administrator to exploit MLD. A telemetry tool will be very helpful in controlling and monitoring such processes. Choosing a subset of all processes to be monitored reduces the performance cost of MLD to minimum. As a rule of thumb, there are some types of processes that can benefit from MLD such as: long live processes and critical applications that can not tolerate crashes. In theory, MLD can be used to solve memory leak in any process. However, there is no meaning to pay the cost of MLD in a short term application, an uncritical application, or applications that are proven not to have memory leak related problems.

The “main function” of MLD algorithm initiates the initialization function which we will explain next. After initialization, the “main function” starts the bookkeeping() function and the sweeper() function and continues these functions for ever. Note that the sweeper must have been started aleady before it can be used by the “main function”.

www.manaraa.com

3.3.2 Initialization

```
Initialize () {
        Set Heap_Size_Threshold to input value
        Set Sweeper_Sleep_Time to input value
        Set Page_Age_Threshold  to initial input value
}
```

FIGURE 8: MLD ALGORITHM-INITIALIZE() FUNCTION

The initialize() function, figure 8, initializes some important parameters that affect the behavior of the MLD algorithm. These parameters are:

Heap_Size_Threshold: The sweeper function will start sweeping once the size of the heap exceeds this threshold value. For example, Heap_Size_Threshold can be set to 80% or 90% of the maximum possible heap size. This parameter is useful to keep the cost of memory lead detection and recovery as low as possible. Note that memory leak is not a problem in its own. It becomes a threat to the running application only when a malloc function fails to allocate memory due to memory unavailability. Hence, we propose to run the sweeping part of the algorithm, which is responsible, for recovering leaky objects, only when the heap size has approached its maximum limit.

Sweeper_Sleep_Time: the time in milliseconds a sweeper will wait between any two successive scans. The smaller the value of Sweeper_Sleep_time the more overhead the sweeper will generate on operating system and vice versa.

Page_Age_Threshold: the age value above which a page will be considered to have a potential leak. Page_Age_Threshold is a tunable parameter. One simple way to dynamically calculate the average age of the Page_Age_Threshold:

www.manaraa.com

$$Page-Age-Threshold=\frac{\sum_{i=1}^{n} page_Age(i)}{n}$$

Where n is the number of pages available in the virtual address space. A telemetry tool will be very handy in initializing and tuning such parameters.

3.3.3 Bookkeeping

```
Bookkeeping(){
        Reset time stamp field (TS) for any new entry added to page table.
        For every victim page selected from the mapped physical space
                Time stamp the corresponding page in the virtual address space
                by setting the time - stamp field in the page table to the current
                time.
        For every paged-in page
                Reset the Time stamp field.
}
```

FIGURE 9: MLD ALGORITHM-BOOKKEEPING() FUNCTION

The major task of bookkeeping() function shown in figure 9 and figure 6(a) is to timestamp a page whenever it is selected as a victim by the page out replacement algorithm. This time will be used to calculate the time a page remains out of physical memory, i.e., the age of a page.

The page table is the data structure that stores the mapping between the virtual addresses and the physical addresses. In a computer architecture where the word size is 32 bits, we are able to address 2^{32} different virtual locations. If the page size is 1 KB, then the page table has 2^{22} entries.

To facilitate the implementation of the aging algorithm for leak detection and the bookkeeping() functionality, we augment the page table in the virtual memory system with a new attribute that we call Time Stamp (TS) as shown in table 1. TS Attribute is marked with "*" to indicate that it is a new entry in the page

www.manaraa.com

table. The TS field is initialized to zero each time a new page entry is added to a page table. When a page is swapped back into the physical space, TS is set to zero also.

Presence Bit(PB)	Frame no.(FN)	Secondary Storage Address (SSA)	Dirty Bit (Dbit)	* Time Stamp (TS)

Table 1: Augmented Page Table

Where:

Presence bit (sometimes called valid-invalid bit)(**PB**): PB indicates whether the physical page is in main memory or must be fetched from secondary storage (a page fault). When this bit is set to “valid”, it indicates that the associated page is both legal (in the process’s logical address space) and in memory. If the bit is set to “invalid”, it indicates that the page is either not valid (not in the process’s address space), or is valid but is currently on disk. Illegal addresses are trapped by using the valid-invalid bit.

Frame number(**FN**): FN indicates the physical base address of a frame.

Secondary storage address (**SSA**): SSA is used to locate the data on disk.

www.manaraa.com

Dirty (modify) **bit** (**Dbit**): Dbit is set whenever any word or byte in the page is written into. When a page is selected for replacement, the dirty bit is examined. If it is set, it must be written back to disk. If it is not set, then the page has not been modified and it can be overwritten without writing it to disk.

Time Stamp* (**TS**): Time stamp attribute added to page table to facilitate bookkeeping functionality of the MLD.

The best place to implement the bookkeeping functionality is in the page replacement policy. Pages become candidate for replacement in the physical memory according to the replacement policy used(LRU, FIFO, LFU,OPT (Silberschatz et al,2005)). Irrespective of the replacement policy used, once a page is selected as a victim page, the TS of the corresponding page in the virtual address space is set to the current time. For any paged-in page the TS field is reset. The TS of zero value means the page is new and no longer considered an aged page.

3.3.4 Memory Allocation and Deallocation Functions

In order to implement the MLD algorithm, we suggest changes to memory allocation and deallocation functions such as malloc(), new() , free(), dispose() and delete() functions. We show the changes to malloc() and free() in the next two sub sections. Changes to other allocation/deallocation functions are the same, and hence we do not provide a description for all funcitons. The interface to the malloc() and free() functions is left intact. This means we do not need to

www.manaraa.com

make significant changes to available user applications. Changes will be only implemented in memory allocation and deallocation functions. Next, we present these changes in malloc() and free() as an example.

3.3.4.1 Malloc ()

Figure 10 shows malloc() after being modified to accommodate the MLD algorithm. The original code of malloc() is left as is. However, the malloc() function execution incorporates two major changes just before it returns a pointer to the newly allocated memory chunk. The first change is made to defer paying the performance penalty cost, The heap size is checked by malloc(). If it has reached a predetermined threshold value, then the Sweeper() is started.. Once the threshold limit is reached this means the application is about to reach the maximum size of the heap and the application state becomes critical. Applications with heaps that do not reach the Heap_Size_Threshold will never pay the cost of sweeping and therefore the cost incurred is kept to minimum.

```
void *malloc( size_t size ){
 void * returnPtr;// pointer to newly allocated object
 The original code of malloc is left unchanged
// This piece of code is intended to monitor the execution of the Sweeper().
  If  (Current_Heap_Max_Size > Heap_Size_Threshold){
                Start The sweeper()// if it is not already started
            }
//This piece of code is intended to maintain a table structure for
//allocations/deallocations
MallocTable.add(retrunPtr, size,0); //0 assume unreachable(dead)
return returnPtr
}
```

FIGURE 10: MLD ALGORITHM – MALLOC() FUNCTION

www.manaraa.com

The second change made to the malloc() function execution is intended to maintain a new table data structure. We call this table MallocTable and it is shown in Table 2.

Begin Address	Size	Mark Flag

Table 2: MallocTable, Memory allocation table

For any memory chunk created by malloc(), a corresponding entry will show up in the MallocTable. This entry will contain the starting address of the chunk in the heap, the chunk size, and a mark flag. Reachable objects are identified by scanning the static stack, and registers. The mark flag is set by the sweeper if a chunk is reachable and remains zero if it is not. All entries with mark flag reset are unreachable objects and have to be garbage collected.

3.3.4.2 free ()

The free() function is shown in figure 11 and figure 6(b). The original code of free() function remains the same. A single change is made to the free() function execution. Just before the free() function exits, it removes the freed object from the MallocTable.

```
void free(void *s){
//original code the same
mallocTable.remove(*s)
}
```

FIGURE 11: MLD ALGORITHM- FREE() FUNCTION

www.manaraa.com

3.3.5 Memory Leak Detection and Sweeping

The main strength of the detection algorithm is that it utilizes the information provided by the virtual memory manager to identify the age of a memory chunk in the virtual memory using physical memory allocation information.

There is a certain category of memory allocations, which can be considered by the compiler and/or the user as non-leaky no matter how long they remain in memory, i.e., independent of age. Typical examples are dictionaries, trap and exception handling objects, and other libraries. Such objects can be locked permanently until the program exits. All other leaks are treated according to the MLD algorithm. Next, we explain memory leak detection and sweeping.

3.3.5.1 Memory Leak Detection

```
bool isleakyPage (page_number, current_time, Page_Age_Threshold (τ)) {
  if (time stamp of page_number>0){
    Age = current time – time stamp of page
    If Age > Page_Age_Threshold(τ) then return true
  }
  Return false }
}
```

FIGURE 12: MLD ALGORITHM- ISLEAKYPAGE() - LEAK DETECTION FUNCTION

According to the leak detection function, isLeakyPage(), shown in figure 12, a page is considered leaky if its page age is greater than the Page_Age_Threshold, where the page age is equal to the value of current time minus page time stamp. The time stamp is entered into the corresponding page table entry (TS) when a page is selected for replacement by the page replacement policy. A leaky page, in this context, may or may not contain a real

www.manaraa.com

leaky object; thus it is considered as a candidate for leak. Since a leaky page has not been used in memory for a relatively long time, it is likely that it has some unreachable objects (real leak), but this is not necessary. If the time stamp is zero this means that the page is active and available in physical memory.

3.3.5.2 Memory Leak Sweeping

```
Sweeper(){
            //use conservative approach for garbage collection
            while(true){
          For each page in the page table for the monitored process{
                              If isLeakyPage(page_number, current_time,
Page_Age_Threshold (τ)){
                              Mark all reachable chunks from (static, stack, and
registers) as live objects
                              //use conservative approach//consider all a like pointers
as pointers
                                        GC unreachable chunks(dead)
                                        Remove GCed chunks entries from mallocTable

                              }
          }
           sleep(Sweeper_Sleep_Time)
```

FIGURE 13: MLD ALGORITHM- SWEEPER() – LEAK SWEEPING FUNCTION

The Sweeper(), figures 13 and 6(b), is a process started by the malloc() function when the heap size grows to a point close to its maximum size. The main function of the Sweeper() is to remove unreachable objects from aged pages, given that the aged page is found to include a leak. The Sweeper() starts sweeping for the application that exceeds the Heap_Size_Threshold. The Sweeper() iterates through the page table to find out potential leaky pages of the target application every Sweeper_Sleep_Time. The pages identified by the Sweepr() at this level are the ones with potential leaks. In order to determine

www.manaraa.com

which page actually contains a real leak, the Sweeper() performs another function. For each page in the set of potential leaky pages, the sweeper starts scanning from the roots (static, stack , and registers) to find out if this page has unreachable objects. Unreachable chunks in every identified leaky page are deleted (garbage collected). Note that the scanner does not have to scan across all the heap objects which may be very large. This has always been the main disadvantage of classic garbage collection tools. In our approach, the Sweeper() first identifies the potential leak locations, and then performs the scanning against these locations only. This way, the cost of finding the leaky objects is drastically reduced.

3.4 Crash Delay

By removing unreachable objects from the heap, the sweeper() will save additional new room in the heap for future allocations. This additional room will make the target application live longer and delay possible crash due to lack of memory. However, for several reasons, the Sweeper() may not prevent program crashing due to lack of memory. Among these reasons are: 1) setting Heap_Size_Threshold to a relatively large value which delays the startup of the Sweeper(), 2) setting the Sweeper_Sleep_Time to a large value that makes the sweeper() not able to cope with the speed of the allocation operations being made by target application. Allocation operations are process dependent, and 3) Setting the Page_Age_Threshold to a relatively large value which makes it more difficult for the Sweeper() to identify enough leaky pages. In fact, the Sweeper()

www.manaraa.com

will fail to identify any single leaky page if the Page_Age_Threshold is extremely large. In case the Sweeper() fails to ensure that the required space is available on the heap to satisfy allocation requests, the target application will crash.

Crash prevention is one of the major advantages of MLD algorithm if the sweeper() is able to remove unreachable objects and make enough room for new allocations. In the worst case, the MLD algorithm delays the crash if it is imminent and makes application live longer. The ML-VMS (Chapter 4) along with a memory leak detection and recovery algorithm will be able to completely prevent crashing as the requested size for allocation will always be available.

3.5 False Positives

One of the main problems in memory leak detection tools is the potential error known as "false positives". In other words, a detected leak is not a real leak. The object identified as a potential leak gets dereferenced after the system has given up on it! Referencing an object after it has been removed from memory, i.e., deallocated, causes incorrect results or the program to crash altogether. False positives can not be tolerated in critical mission applications. Hence, we have to be careful when dealing with false positives.

MLD algorithm is a conservative algorithm. It produces zero false positives. The MLD starts scanning from the roots (static, stack, and registers) searching for reachable objects. Once reachable objects are identified, the remaining are hundred percent unreachable and can not be false positive.

www.manaraa.com

Without the scanning part of the algorithm, the MLD produces false positives. However, the rate of false positives can be controlled using the Page_Age_Threshold. Increasing this threshold value will reduce the rate of false positives. For some noncritical applications, where the program crash and restart does not cause a serious penalty to the users, the MLD can very well be used without the scanning part.

3.6 False Negatives

False negatives are leaky chunks that go undetected. Note that a page allocated to physical memory may consist of one or more memory objects in the heap virtual space. If at least one of these objects remains active, then the corresponding page will never age beyond the age threshold. As a result, the other objects allocated to the same page will go undetected if they become leaky. This phenomenon will result in false negatives, i.e., undetected leaky objects.

MLD algorithm does not totally remove false negatives, but it can minimize the number of false negatives by decreasing the Page_Age_Threshold. Decreasing Page_Age_Threshold will make MLD identify more leaky objects. However, this operation will increase the cost of the sweeper() that will sweep a relatively large number of potential leaky pages. So there is a trade off. After all, several numbers of false negatives can be tolerated since the MLD will help the application to keep going.

Another way is to choose a smaller page size. In this case, the number of independent objects allocated to the same page will be reduced. However, there

www.manaraa.com

are some disadvantages for choosing smaller page sizes. One way of getting around this problem is to use the concept of dynamic page size setting, where pages will have different sizes, and each page size is determined based on the memory objects sizes. Smaller objects will be allocated to smaller page sizes, while larger objects will be allocated to larger page sizes. This concept will be the subject of future research.

3.7 Memory Leaks and Telemetry

The main concept behind memory leak detection is the aging of objects. The telemetry subsystem may be used to monitor the aging of objects. The age monitor reports the objects whose age exceeds a certain threshold value. This process allows the user more control over memory leaks. The report includes the memory object, the owner, the age, and recommendation on how to deal with the object.

Some benefits that can be gained by a telemetry tool:

- Ability to select which processes will exploit the MLD algorithm and which will not.
- Initialize and tune MLD parameters such as: Heap_Size_Threshold, Sweeper_Sleep_Time, and Page_Age_Threshold for each monitored proecess.
- Create and monitor a sweeper for each monitored process.
- Monitor heap limits, false positives, false negatives, and other performance measures

www.manaraa.com

- Allow the user to discover potential leaky objects and make appropriate corrections in the source code.

3.8 Conclusion

This chapter provides a full description of a new approach for memory leak detection (MLD) algorithm using aging in physical memory. This algorithm reflects both the physical and virtual behavior of memory allocation and benefits from the hardware support available for tracking physical pages in real memory.

The MLD is shown to follow a conservative approach in removing unreachable objects. The MLD is not able to deal with stale objects at run time because there is no way to tell if these objects will be referenced in the future by a running program. The current structure of virtual memory system and dynamic allocation prevents the availability of such a complete run-time solution. Applications that exploit this algorithm are able to live longer than the applications without it. The false negatives and overhead depend on some input parameters like Page_Age_Threshold and system parameters like Page_Size which suggest a need for a telemetry tool. In the next chapter, we provide a complete run-time solution. In chapter 5, we show the effect of tuning the input parameters on performance and we provide performance evaluation results

www.manaraa.com

Chapter Four
Multi-Layer Virtual Memory System (ML-VMS)

4.0 Introduction

In the previous chapter, we showed that the current structure of virtual memory system and dynamic allocation prevents the availability of a complete run-time solution for memory leak problem.

In this chapter, we propose to reorganize the virtual memory system into a Multi-Layer Virtual Memory System (ML-VMS). The ML-VMS is a novel structure that reorganizes the virtual memory system and dynamic memory management. A new algorithm for memory leak detection and recovery (MLDR) is presented based on this new structure. The MLDR still uses the physical memory aging as done by the MLD, however, it will utilize the proposed new ML-VMS. We show how the ML-VMS along with the MLDR allow for efficient resolution of memory leak problem.

The idea of the ML-VMS emerged out of the original MLD algorithm. Initially, the MLD algorithm, based on aging, was supposed to deallocate all objects in a page whose age exceeds the given threshold. However, to account for false positives, it was suggested that objects can be kept on the hard disk, while removed from the heap, in case they get dereferenced (false positives). The main challenge was to be able to recover the false positives from the hard disk. Several approaches were discussed and addressed, including the use of hardware traps, the modifications of addresses of freed objects, and others

www.manaraa.com

. However, careful investigation of this challenge reveals that the main issue is to establish a mapping between the heap storage and the global disk storage where potential leaky obects can be stored.

Moreover, the real problem of leaks is that the heap storage has a physical limit imposed by the size of the address words (32 bits in 32-bit machines and 64 bits in 64-bit machines). Memory leaks may exhaust the heap size, although the application may not in reality need that much space. As such, the ability to move objects from the heap and store them in the much larger disk space (unlimited space) and recover them when needed will allow a greedy MLD algorithm to dispose of potential leaks without the fear of a crash whenever these objects are derferenced. Also, an application program which requires very large memory space (larger than the 32-bit address space) will be able to run without facing "out of memory" failure mode. Of course, with 64-bit machines this problem may not be as serious. However, our experience with application development is that users will tend to exhaust computer resources as soon as they become available. In this chaper, we will present the ML-VMS approach which provides a solution to this challenge.

While the MLD is not able to deal with stale objects the MLDR provides a complete run-time solution for memory leak problem. A simulation model will be used, in chapter 5, to validate both of the ML-VMS and MLDR and provide a proof of concept. We also provide some guidelines that facilitate the

www.manaraa.com

implementation of this new structure (ML_VMS) and MLDR.

This ML-VMS along with MLDR algorithm provides the following contributions:

1. Deals with both unreachable and useless (stale) objects.
2. Handles the problem of false positives. If a false positive object is deleted from the heap and referenced later by the application the deleted object is recovered.
3. Provides run-time solution for memory leak detection and recovery whereas most of previous approaches either detect memory leak in development environment or remove only unreachable objects in run-time environment as performed by garbage collectors.
4. Prevents programs from crashing by guaranteeing that space on the virtual memory is always available by moving potential leaky objects to disk that presumably has an unlimited space.
5. Although there is a performance penalty cost (PPC) that will be paid by the algorithm, this penalty is kept to minimum by using the following techniques:
   a. The PPC will never be paid (i.e deferred) until either the program exceeds a certain threshold in virtual address space or there is no available memory to be allocated and the application is about to crash.

www.manaraa.com

b. The algorithm consists of modular parts allowing optimal future implementations to these modules which reduce the overall PPC, as a result.

c. Aging in the virtual space is detected based on aging in the physical space, thus allowing the algorithm to utilize the hardware available for virtual memory organization.

d. The algorithm utilizes tunable parameters that reduce the cost to the minimum.

e. Parallel programming is suggested, as shown in chapter 5, to provide more enhancements in performance.

We describe the structure of the new ML-VMS and show how the ML-VMS allows for efficient resolution of memory leak problem through the memory leak detection and recovery (MLDR) algorithm.

4.1 Multi-Layer Virtual Memory System (ML-VMS)

The ML-VMS is constructed by adding an additional layer to the virtual memory system. This layer introduces a new data structure that we call a virtual heap table (VHT). The VHT is shown in Table 3. The VHT contains an entry for each allocated memory chunk. The memory chunk can be located on the heap and then VHT contains its virtual address or it can be located on disk and the VHT contains its disk address.

www.manaraa.com

Virtual Heap Table Index (VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
0				
1				

Table 3: Virtual Heap Table(VHT)

Where:

Virtual Heap Table Index (VHTI): is an index to the virtual heap table. This entry does not have to be stored in the table. Each entry in the table is identified by this index.

Presence Bit (PB): PB tells whether the object is on the virtual address space or on the disk. If the PB is set, it indicates the object is allocated in the given virtual address space entry. Otherwise; the object was deallocated from the virtual address space by the MLDR algorithm and it (the object) is currently available on disk in the given disk address entry. The default value of PB is one. i.e the object is available on the virtual address space.

Virtual Address (VA): is an entry for the base virtual address of an object. This entry has meaning only when the PB is set.

Size: is an entry that contains the size of the memory chunk (object).

Disk address (DA): contains the address at which the object was backed up before being deallocated by the MLDR algorithm. This entry has meaning only when the PB is zero.

www.manaraa.com

4.2 Address Resolution

Address resolution is the process of address translation from a virtual address to a physical address. In the ML-VMS organization, there are two levels of address translation. One level is required to find the virtual address of an object. The second level is to find the physical address in the physical memory. Address resolution in the new ML-VMS is illustrated in figure 14. The figure shows how a VHT reference (VHTR) is translated into a physical address.

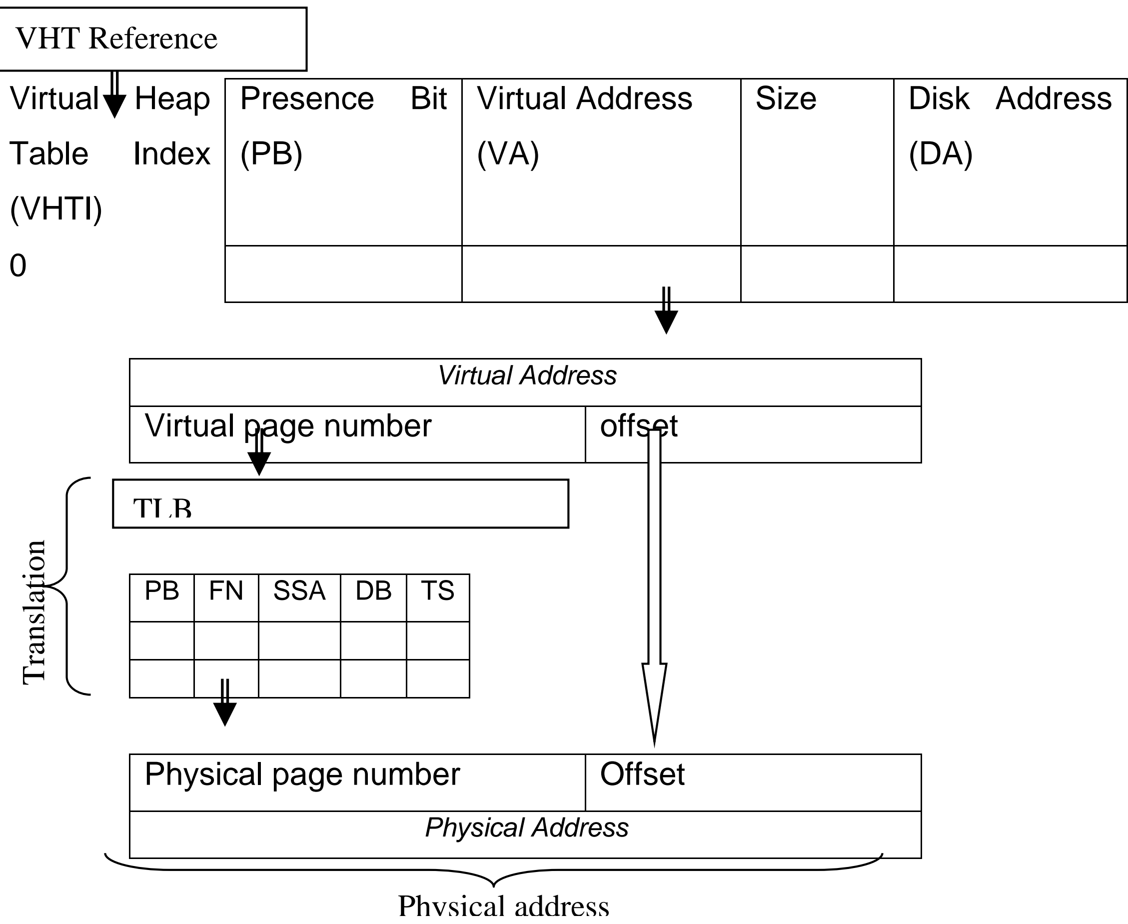


FIGURE 14: ADDRESS RESOLUTION IN ML-VMS - MAPPING VHTR INTO PHYSICAL ADDRESSES

المنارة للاستشارات
www.manaraa.com

According to the ML-VMS, programs now use virtual heap table references (VHTR) instead of virtual addresses. Address resolution is started with a valid VHTR. This reference is an index to the VHT that we call virtual heap table index (VHTI). If the PB of the VHTI is set, address resolution proceeds normally with the VA that is associated with the VHTI. Otherwise; the object is on disk at a given DA and it has to be recovered (4.3.4) and the address resolution must be restarted. Next, we list the memory leak detection and recovery algorithm (MLDR) based on this ML-VMS.

4.3 The MLDR Algorithm Based on the ML-VMS

The MLDR algorithm contains the following modules: memory allocation, memory deallocation, ML-VMS with aging, and object recovery. We define and present these modules respectively.

4.3.1 Memory Allocation

Memory allocation proceeds as follows:

a. Find a free memory chunk and return its virtual address (*VA*).
b. Place the *VA* in a new entry in the VHT.
c. Set the PB
d. Return the corresponding virtual heap table index (VHTI) as a virtual heap table reference (VHTR) to the calling program.

www.manaraa.com

e.

4.3.2 Memory Deallocation

Memory deallocation proceeds as follows:

a. If the PB in the VHT is reset then the VHTR belongs to an object that is freed earlier. Exit memory Deallocation.
b. Else translate the given VHTR of a memory chunk into a *VA*
c. Proceed normally to deallocate the given *VA* in the usual deallocatin process.
d. Reset the PB

4.3.3 ML-VMS and Aging

The ML-VMS facilitates the implementation of the aging algorithm and hides the memory leak problem.

If the aging algorithm decides that a given page is aging and must be freed, it performs the following steps:

a) Use the number of the aging page as a search key to look up all of page's corresponding aging memory chunks in the VHT.
b) For each aging chunk CHi
   1) Backup CHi to disk if it is reachable (useless or stale)
   2) Set the disk address (DA) of CHi in the VHT and reset the corresponding PB if it is reachable and remove the VHT entry if it is not reachable.

www.manaraa.com

3)

4) Deallocate CHi from the Heap whether it is reachable or not.

4.3.4 Object Recovery

a. Find a free memory chunk and return its virtual address (*VA*)

b. Copy the object from disk to the new located object

c. set the PB

d. set the virtual address entry in the VHT to the new *VA*

e. Remove the recovered chunk from disk.

4.4 MLDR algorithm block diagram based on ML-VMS

The MLDR algorithm block diagram is shown in figure 15.

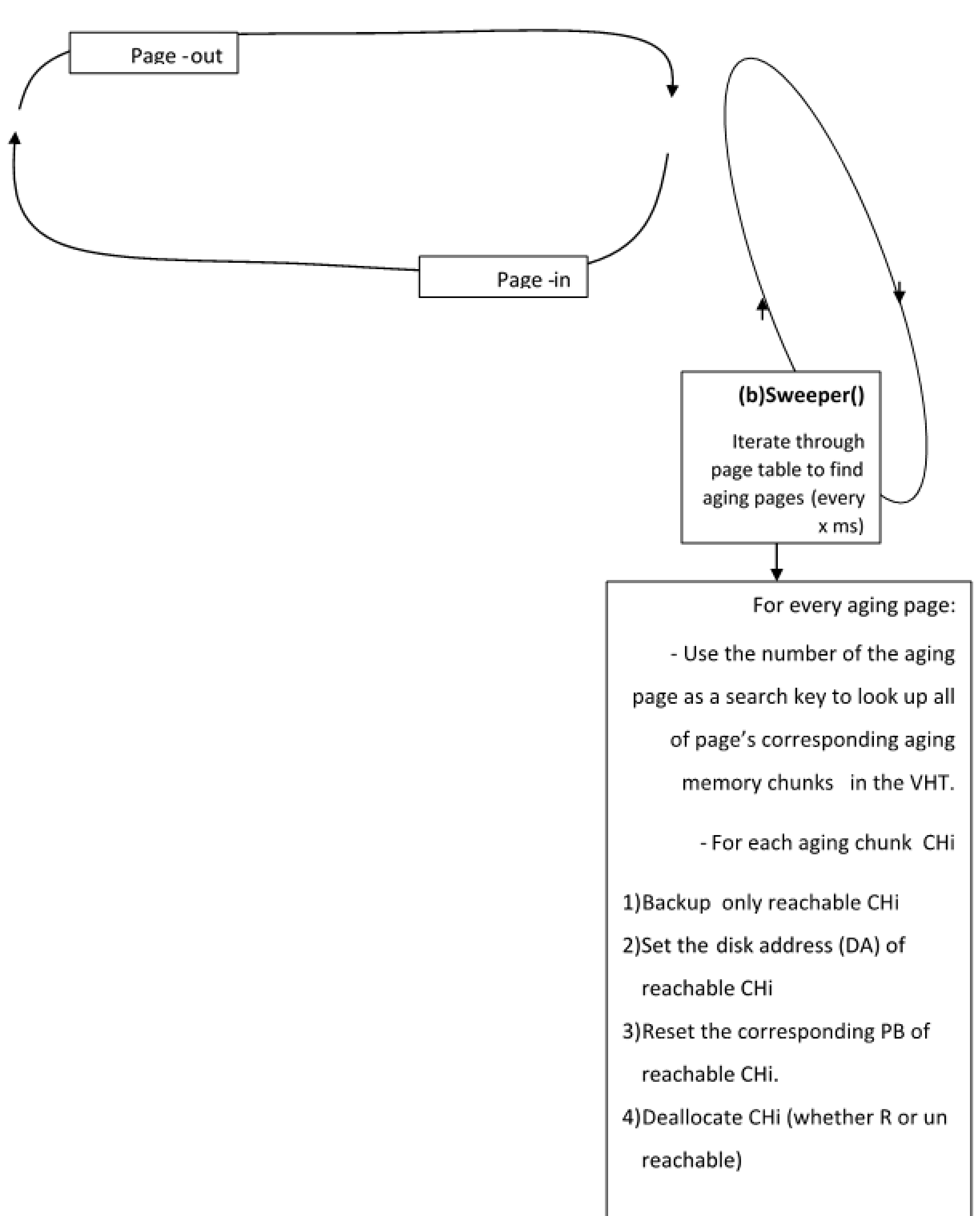
RAM

			TS
			CurrentTime
			0

Page Table

(a) bookkeeping

المنارة للاستشارات
www.manaraa.com



المنارة للاستشارات
www.manaraa.com

```
void *malloc( size_t size ){
 void * returnPtr;// pointer to newly allocated object
 long int VHTI;//unique identifier
 The original code of malloc is left unchanged

  If  (Current_Heap_Max_Size > Heap_Size_Threshold){
                Start The sweeper() //if it is not already started
             }
VHTI=getIndex()
HeapTable.add(VHTI,1,retrunPtr, size,0);

returnVHTI
}
```

(c) memory allocation

Heap Table

Virtual Heap Table Index (VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
0				
1				

المنارة للاستشارات
www.manaraa.com

FIGURE 15 : THE BLOCK DIAGRAM OF MLDR ALGORITHM

4.5 MLDR Explanation

In this section, we explain the main modules of the algorithm along with enough necessary examples. The aging concept is used in both of the MLD (chapter 3) and the MLDR (chapter 4). Readers should refer to chapter 3 on details about aging in physical memory space if necessary. The following modules, that are explained next, are related to the MLDR.

4.5.1 Memory Allocation Module Explanation

Memory allocation module proceeds as shown in figure 16.

a) Find a free memory chunk and return its virtual address (*VA*).
b) Place the *VA* in a new entry in the VHT.
c) Set the PB
d) Return the corresponding virtual heap table index (VHTI) as a virtual heap table reference (VHTR) to the calling program.

FIGURE 16: MEMORY ALLOCATION FOR MLDR

The memory allocation process starts by finding a free chunk in the heap and returning its virtual address (VA). This VA is placed as a new entry in the VHT. The PB is set to indicate that this chunk is available in the virtual address space. The VHTI is returned to the calling program. For example, after an application executes the statement:

```
x = malloc(50);
```

www.manaraa.com

the VHT will have a new entry as follows:

(VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
0	1	0x8012a67	50	0

We assume this allocation was the first allocation in the program. PB is set. VA=0x8012a67 which is the address of the chunk in the heap. The value of x in the program now contains the VHTI=0 not 0x8012a67 as usually done by current virtual memory systems.

Figure 17 shows the suggested changes made to the malloc() function in order to implement the memory allocation module of the MLDR.

```
void *malloc( size_t size ){
 void * returnPtr;// pointer to newly allocated object
 long int VHTI;//unique identifier
 The original code of malloc is left unchanged
  If  (Current_Heap_Max_Size > Heap_Size_Threshold){
                Start The sweeper() //if it is not already started
             }
VHTI=getIndex()
HeapTable.add(VHTI,1,retrunPtr, size,0);
returnVHTI
}
```

FIGURE 17: MALLOC() FUNCTION FOR THE MLDR

www.manaraa.com

Most of the original code for malloc() including the malloc() interface is left intact so there will be no need to make any changes in the user programs. There are two issues to mention. The first is that, as we discussed in the MLD, the malloc() starts the sweeper() when the heap size exceeds a certain threshold. The latter is that the malloc() adds the VA to the heap table and returns the VHTI of that entry. Each added item will have the PB set by default, the VA, the size and null value for the DA.

4.5.2 Memory Deallocation Module Explanation

Memory deallocation proceeds as follows:

a) If the PB in the VHT is reset then the VHTR belongs to an object that is freed earlier. Exit memory Deallocation.

b) Else translate the given VHTR of a memory chunk into a *VA*

c) Proceed normally to deallocate the given *VA* in the usual deallocatin process.

d) Reset the PB

FIGURE 18: MEMORY DEALLOCATION OF MLDR

Memory deallocation module, figure 18, starts by passing a VHTR. If the PB of that reference is reset then this is an attempt to free an already freed chunk. In this case, the deallocation module exits. Otherwise, the VHTR is translated into a VA. Then, this VA is freed from the heap. And the PB is reset. For example, after an application executes the statement:

www.manaraa.com

free(x); //and x has the value of VHTR= 0.

the VHT will be as follows:

(VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
0	0	0x8012a67	50	0

The free() function translates the VHTR of x =0 to the VA=0x8012a67. This VA is deallocated from the heap and the PB is reset to indicate that the VA in this entry no longer exists. Any other future attempt to free this object again by calling free(); say free(y) where y was set to x. The module will find that the PB is reset and exits. This result is important and shows how the ML-VMS solves the problem of aliasing and dangling pointers. This is outside the scope of our Dissertation but is mentioned for future research.

المنارة للاستشارات
www.manaraa.com

4.5.3 MLDR with Aging Module Explanation

If the aging algorithm decides that a given page is aging and must be freed, it performs the following steps:

a) Use the number of the aging page as a search key to look up all of page's corresponding aging memory chunks in the VHT.

b) For each aging chunk CHi

1) Backup CHi to disk if it is reachable (useless or stale)

2) Set the disk address (DA) of CHi in the VHT and reset the corresponding PB if it is reachable and remove the VHT entry if it is not reachable.

3) Deallocate CHi from the Heap whether it is reachable or not.

FIGURE 19: MLDR WITH AGING

المنارة للاستشارات
www.manaraa.com

The aging in physical memory space is used to detect aging in virtual address space. This concept was discussed thoroughly in chapter 3. Here, we explain how it is related to the MLDR and the ML-VMS, figure 19, using the following four-step example.

Step one:

We start the example by presenting a block of code, figure 20, shown next. In this example, xPtr, yPtr, and zPtr point to chunks of sizes 40, 60, and 20 respectively. Assume that all of these pointers are created and mapped to page 1 on the page table.

```
char* xPtr, *yPtr, *zPtr;
xPtr = (char *) malloc(40);/* allocate memory */
yPtr = (char *) malloc(60); /* allocate memory */
zPtr = (char *) malloc(20); /* allocate memory */
```

FIGURE 20: A BLOCK OF CODE FOR THE MLDR EXAMPLE

Step two:

After allocating xPtr, yPtr, and zPtr according to memory allocation module of the MLDR, we get the following snapshot of the VHT:

(VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
…	....	…..	…	….
120	1	0x8012c00	40	0
121	1	0x8013d00	60	0
122	1	0x8018e11	20	0
…	…	…	…	….

Later on, suppose at time T6 we take a horizontal slice of the page table that we show next.

www.manaraa.com

	Presence Bit(PB)	Frame no.(FN)	Dirty Bit (DB)	* Time Stamp (TS)
Page 0	....	...	...	....
Page 1	1	12	0	T1

Table 4: A horizontal snapshot slice of the augmented page table at time (t6)

The bookkeeping() function of the aging algorithm has already time-stamped page 1 with T1. T1 represents the last time page 1 was used in the physical memory.

Step three:

The aging algorithm will decide that page 1 is aging because its age=T6-T1 is greater than a threshold value. The aging algorithm proceeds in looking up all of the pages's corresponding chunks in the VHT. These chunks in the aging page are those pointed by VHTI= 120, 121 and 122.

Step four:

Now, we perform the steps 1 through 3 of the aging model on each element: 120, 121, and 122. Assume that entry 120 is still reachable by the application program and entries 121,122 are not. Then after executing the aging module we will have the following snap shot of the VHT.

www.manaraa.com

(VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
…	....	…..	…	….
120	0	0x8012c00	40	0xdab678ca
…	…	…	…	….

All entries were deallocated from the heap whether they are reachable or not. The chunk with VHTI=120 is backed-up to disk. It's PB is reset to indicate that the chunk is no longer available on the virtual address space and it is on the disk at the disk address indicated by the DA entry=0xdab678ca.

4.5.4 Object Recovery Module Explanation

Object recovery module, shown in figure 21, is executed once an application makes a reference to an object that was already deallocated and backed up to disk. This is called a reference to a false positive object. A false positive object was identified earlier as a potential leak and moved to disk. It turns out, later on, that this object is being accessed by the program. The MLDR object recovery module a long with the ML-VMS is capable of recovering such an object.

www.manaraa.com

a. Find a free memory chunk and return its virtual address (*VA*)

b. Copy the object from disk to the new located object

c. set the PB

d. set the virtual address entry in the VHT to the new *VA*

e. Remove the recovered chunk from disk.

FIGURE 21: OBJECT RECOVERY FOR MLDR

We explain how the object recovery module works using an example. We take a snapshot of the VHT before and after executing the object recovery module. Assume the snapshot of the VHT before executing the recovery module as follows:

(VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
…	....	…..	…	….
120	0	0x8012b00	40	0xdab678ca
…	…	…	…	….

www.manaraa.com

If the program has to access entry 120 it finds out that the PB is reset. This means the chunk is available on disk. The module allocates a new room for chunk stored in DA=0xdab678ca in the heap and return it's VA, say 0x8003c01. The PB is set to indicate the chunk is now available in the virtual address space and can be accessed normally and the VA is changed in the VHT to point to the new returned VA. The recovered chunk with DA=0xdab678ca is removed from the disk to save space. The new snapshot of the VHT after performing the object recovery looks as follows:

(VHTI)	Presence Bit (PB)	Virtual Address (VA)	Size	Disk Address (DA)
…	....	…..	…	….
120	1	0x8003c01	40	0xdab678ca
…	…	…	…	….

4.6 Crash Preventing

We have seen that the MLD can delay a possible application crash for an application dependent period of time. In case the crash is imminent, the MLD will

www.manaraa.com

not prevent it. One big enhancement of the MLDR over the MLD is that the MLDR can prevent the target application from crashing if the input parameters are well-tuned. Among these parameters are Heap_Size_Threshold, Page_Age_Threshold, and Sweeper_Sleep_Time. The MLDR removes both of the unreachable objects and stale or useless objects, in an aging page, in order to make enough room for new allocations. The requested size for allocation is guaranteed to be always available assuming a large disk is used.

Crash preventing performed by the sweeper(), however, is not always guaranteed for several reasons. These are the same reasons that apply to the crash delay for the MLD. Among these reason are: 1) setting Heap_Size_Threshold to a relatively large value which delays the startup of the Sweeper(), 2) setting the Sweeper_Sleep_Time to a large value that makes the sweeper() not able to cope with the speed of the allocation operations being made by the target application. We have to keep in mind that allocation operations are process dependent, and 3) Setting the Page_Age_Threshold to a relatively large value which makes it more difficult for the Sweeper() to identify enough leaky pages. In fact, the Sweeper() fails to identify any single leaky page if the Page_Age_Threshold is extremely large. In case the Sweeper() fails to ensure that the required space is available on the heap to satisfy allocation requests, the target application will crash.

www.manaraa.com

Actually, we can make use of ML-VMS such that the program will never crash. Here is how. Assume that the sweeper has slept for a very long time. Many leaks are there and have not been discovered. The heap size reached its limit. In this case, the malloc() should be able to free objects in the heap based on FIFO or LRU or Random; and allocate new chunks. This is one of the powerful features of the ML-VMS. We will not address this issue further, and will defer it to future research and investigation.

If our system can tolerate performance overhead cost paid by the sweeper(), the general rule of thumb is to minimize all of the mentioned input parameters. Minmizing Heap_Size_Threshold makes the sweeper() start early and provide enough space before it is too late. Minimizing Page_Age_Theshold makes the MLDR identify more aging pages and provide more enough room. Minimizing Sweeper_Sleep_Time makes the MLDR run the sweeper more frequently and, as a result, identify more aging pages.

4.7 False Positives

"False positives" problem is one of the potential errors in memory leak detection tools. False positive means the detected leak is not a real leak. If the system has given up on a false positive object and got dereferenced later on then the program crashes altogether. False positives can not be tolerated in critical mission applications.

www.manaraa.com

We have seen that the MLD produces zero false positives because it implements a conservative approach that considers every value similar to a pointer as a pointer. The new structure of the ML-VMS allows the MLDR to remove all aging chunks if they are reachable or unreachable. The problem of false positives

occurs when a reachable chunk that has not been used for a relatively long period of time is aged. In that case, the MLDR will remove these aged chunks and falls in the false positive problem in case any of them get dereferenced. The MLDR provides a solution to false positives problem based on the ML-VMS by using the object recovery module of the algorithm.

4.8 False Negatives

A false negative is another potential error in memory leak detection tools. It means the failure to detect real memory leak. In case, there is at least one active chunk in the virtual space page, while the other chunks have leaked, then the corresponding physical page remains active and the leaks in that page will not be detected.

www.manaraa.com

As in the MLD algorithm, the MLDR does not totally remove false negatives. The MLDR is also similar to the MLD in terms that it can minimize the number of false negatives by decreasing the Page_Age_Threshold. Decreasing the Page_Age_Threshold makes the MLDR identify more leaky objects. However, this operation increases the cost paid by the sweeper() that sweeps, as a result, a relatively large number of potential leaky pages. So, there is a trade off. The cost of sweeping in the case of the MLDR is much higher than the cost of sweeping in the MLD because the MLDR sweeping process requires an additional work. The MLDR backs up the removed chunks to the disk in case they might be used again. Writing to a disk is a costly operation in the virtual memory system. Several numbers of false negatives can be tolerated since the MLD will help to keep the application running.

4.9 Memory Leaks and Telemetry

Some parameters affect the performance of the MLDR such as Heap_Size_Threshold, Page_Age_Theshold, and Sweeper_Sleep_Time. We suggest implementing a telemetry tool that tunes these parameters in order to generate the minimum false positive rate given the maximum tolerable overhead. Some other benefits that can be gained by a telemetry tool are the same benefits discussed earlier in chapter 3.

www.manaraa.com

A telemetry tool is outside the scope of this dissertation and is left for future work.

4.10 Conclusion

This chapter presents a new approach for virtual memory system. It reorganizes the virtual memory system into a Multi-Layer Virtual Memroy System (ML-VMS). The ML-VMS adds a new layer to the current virtual memory system and dynamic memory management.

A new algorithm for memory leak detection and recovery (MLDR) is presented based on this new structure. The MLDR still uses the physical memory aging as done by the MLD (chapter 3) but it builds upon the proposed new ML-VMS. We show how the ML-VMS along with the MLDR allows for a complete run-time resolution of memory leak problem.

The MLDR resolves the problem of both of the reachable and unreachable objects. It handles the problem of false positives. If a false positive object is deleted from the heap and referenced later by the application the deleted object is recovered. It provides a run-time solution for memory leak detection and recovery

www.manaraa.com

whereas most of previous approaches either detect memory leak in development environments or remove only unreachable objects in run-time environment. It prevents programs from crashing. The MLDR guarantees that space on the virtual memory is always available by moving potential leaky objects to disk that presumably has an unlimited space.

www.manaraa.com

Chapter Five
Performance Evaluation and Simulation

This chapter analyzes the performance of the MLD and the MLDR based on the ML-VMS in terms of access time and complexity. It shows, through analysis and a trace-driven simulation program, how some of the performance measures can be enhanced. We compare both of the MLD and the MLDR to current memory leak solutions and show how the new approach outperforms the current approaches in providing a complete run-time solution.

We start the performance analysis in section (5.1). Then, we discuss the simulation model of the the MLD and its simulation results in section (5.2). After that, we discuss the simulation results of the the MLDR (5.3) and finally, we compare both of the MLD and MLDR to some available solutions (5.4).

5.1 Performance Analysis

In the next sub sections, we analyze the performance of the MLD, the ML-VMS and the MLDR in terms of access time and complexity and show how this cost can be minimized.

www.manaraa.com

5.1.1 The MLD Complexity

The MLD algorithm (3.1.1) incurs the cost of initialization, bookkeeping and sweeping these cost are given according to equation (1).

MLD cost = Initialization cost + bookkeeping cost + sweeping cost …. (1)

Initialization cost:

This cost is paid once when the application that exploits MLD starts up. It includes initializing three input parameters so this cost is of the O(3).

Bookkeeping cost:

This cost is paid for every paged-out or paged-in page. By setting or resetting the time stamp of the corresponding page in the page table. Suppose, on the worst case, the application swaps in/out n pages then the overall cost is of the O(n).

Sweeping cost:

The sweeper is the bottle neck for the MLD algorithm. It performs the following operations:

1. Iterates over n pages to determine if they are leaky. This operation has the complexity O(n). In practice, the number of pages are process dependent and has an upper constant value which makes this operation have the cost O(number of pages). Where number of pages = process size/page size.
2. For each leaky page the set of unreachable objects have to be found by scanning from static, stack, and registers. Therefore, this operation has the complexity of O(n) assuming linear search is used.

www.manaraa.com

3.

4. The sweeper itself has to be re-executed in every sweeper_sleep_time. This operation has a complexity of O(n).

Since all of these operations are nested then the overall complexity of the sweeper is $O(n^2)$. Sweeping cost dominates the MLD other costs, initialization and bookkeeping. Therefore the overall complexity of the MLD is O (n^2).

Complexity minimization:

The cost of the MLD is kept to minimum by:

1. This cost will never be paid until the heap size threshold limit is reached.

www.manaraa.com

2. The sweeper will be in a sleeping state before being re-executed for a sweeper-sleep-time period.
3. The sweeper can be parallelized as shown in the next section

5.1.2 The MLD Parallelized

In computer environments where parallelism can be used, the MLD performance can be enhanced. We suggest using data partition in order to achieve an enhancement in performance. Figure 22 shows how the MLD can use data partition to parallelize the problem of sweeping.

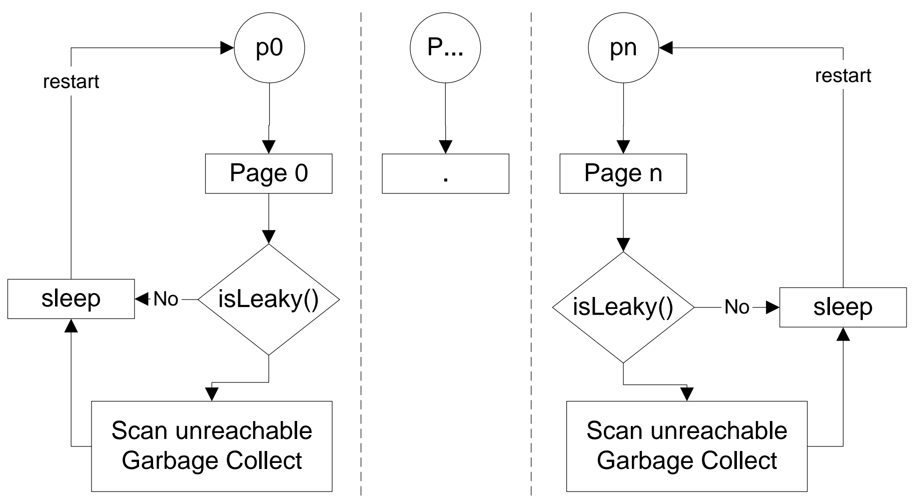


Figure 22: MLD Parallelized

www.manaraa.com

Let

Page[]={page 0, page 1,…,page n} represent the set of page indexes in a page table for a given application where n>=0

process[]={p0, p1, .., pn} represent the set of parallel processes. Where n>=0

Then for every $page_i$ in page[] assign the process p_i from process[].

If the page is aging and therefore might contain a potential leak the process continues; otherwise it sleeps a sweeper-sleep-time period and restarted. The activity of scanning unreachable objects and garbage collection also can be further parallelized for extra performance enhancement. Using parallelism to enhance performance of the MLD is suggested to be investigated in a future work.

www.manaraa.com

5.1.3 Demand Paging Access Time

Demand paging access time is governed by memory access time and the page-fault time.

Let p be the probability of a page fault ($0 <= p <= 1$).

ma be memory access time

pgTime page fault time

dpTime demand paging access time

Then

Demand paging access time (dpTime) is calculated according to the following formula

$$dpTime = (1-p) * ma + p \times pgTime \qquad (2)$$

in order to minimize the dpTime, the pgTime should be as low as possible since ma is usually given in nanoseconds whereas pgTime is in milliseconds. Since this parameter is physically set by the hardware, the only remaining parameter that is SW and architecture dependent is the probability of page faults. Thus minimizing (p) is the target of system architecture and optimization.

www.manaraa.com

5.1.4 ML-VMS Access Time

The ML-VMS has significant effect on the performance of the computer system. The ML-VMS adds additional layer on the demand paging system. In addition to the cost paid by the demand paging memory system, the ML-VMS may incur the cost of accessing a disk either to backup a chunk or to recover another chunk in case of false positives.

The ML-VMS incurs additional time over that of demand paging in case an application is referencing a chunk that has already moved to disk. i.e false positive.

Let pFP be the probability of a false positive access (0<= pFP <=1).

fpTime false positive time

dpTime demand paging access time

Then the ML-VMS access time is given according to the following equation:

ML-VMS access time = (1-pFP) * dpTime + pFP * fpTime (3)

Where fpTime is the service time incurred to perform the following operations

1. Service the false positive interrupt.
2. Recover the chunk from disk including making a new room for the recovered chunk and updating the VHT.
3. Restart the process.

Since most of the above operations are disk operations, we expect the service time, fpTime, to be costly. We minimize this cost by:

www.manaraa.com

a. reducing the probability of false positives (pFP) to the minimum. We show in the next two section how increasing the page age threshold value reduces the false positive rate and, as a result, reduces the performance cost.

b. storing the backed up objects on the swap space instead of the regular disk. Swap space is usually faster than that of the file system.

By substituting equation 2 for dpTime in equation 3 the ML-VMS time is given according to equation 4.

$$\text{ML-VMS access time} = (1-pFP)\,((1-p) * ma + p \times pgTime) + pFP * fpTime \quad (4)$$

5.1.5 ML-VMS Overhead Cost

The overhead of the ML-VMS can be derived from equation 2 and 4 according to the following equation:

$$OH = \text{ML-VMS access time} / dpTime \quad (5)$$

The ML-VMS will be OH times more expensive than that of the demand paging access time. Since false positive rate is the dominant factor of the ML-VMS then from equation 5 the ML-VMS overhead can be minimized by just minimizing the false positive rate as we show in the simulation results of the MLDR.

www.manaraa.com

5.2 Trace-Driven Simulation of Memory Leak Detection Algorithm

In order to validate, verify, and provide a proof of concept to the MLD algorithm, we built and ran a trace-driven simulation program. Figure 23 shows an abstract block diagram for the top-level of the simulation program.

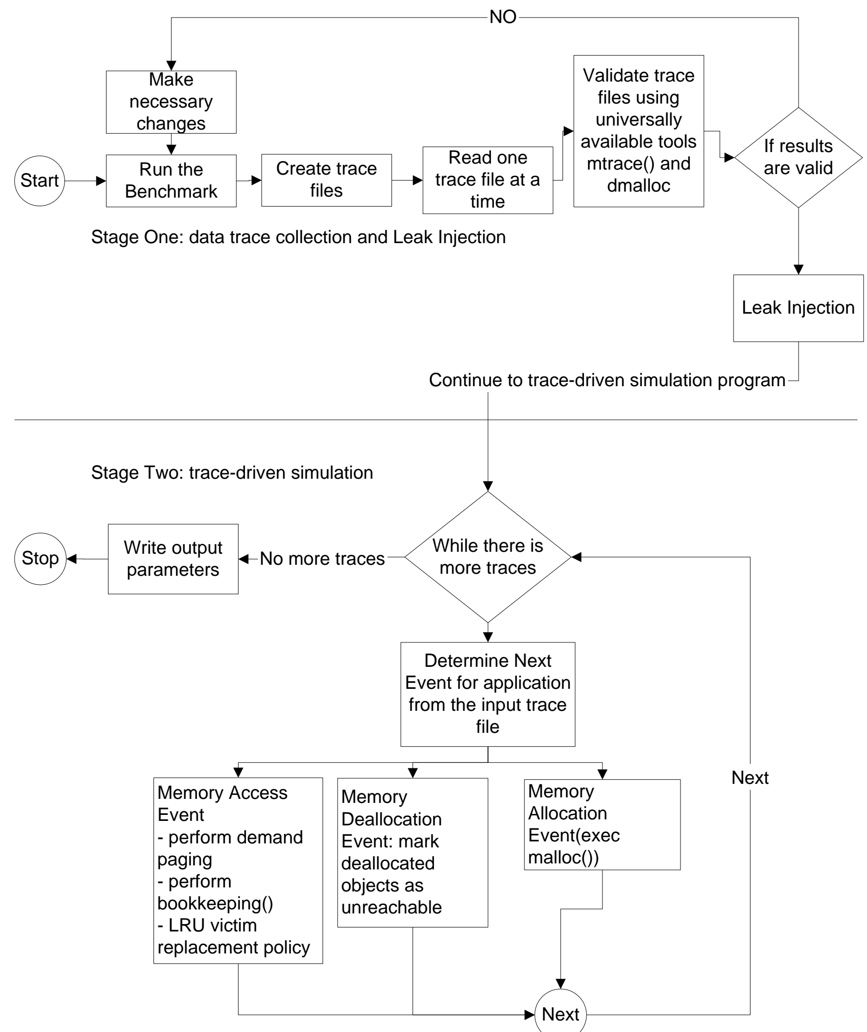


Figure 23: trace-driven Simulation program - abstract block diagram

The trace-driven simulation program consists of two stages. These stages

www.manaraa.com

are shown in figure 23 separated with a horizontal line. The first stage is data trace collection and leak injection and the latter is the trace-driven simulation for the memory leak detection algorithm. We illustrate these stages next.

5.2.1 Data Trace Collection and Leak Injection (Stage One)

In this section, we describe the benchmark used, the data collection technique, and the process of leak injection.

5.2.1.1 MLD Bench Mark

In order to collect data traces and use it as an input to test our trace-driven simulation program, we built our own benchmark. The benchmark generates random allocations and deallocations and records it to a trace file. A survey of trace-driven memory simulation can be found in (Uhlig and Mudge, 1997; Toomula,2004) and the accurecy of trace-driven simulations in (Goldschmidt and Hennessy,1993). We could use the trace provided by a synthetic model (Zorn and Grunwald, 1994) or a trace generated by program execution (Bhansali et al, 2006). Our benchmark provides the following advantages:

1. It is designed to run similar to real-world programs. The runtime period of this benchmark is controlled by a tunable parameter (MAX_TRANS); MAX_TRANS is described in the next section (data collection.)
2. It has a known behavior and its generated statistics matches 100% to the statistics collected by universally well-known tools like memory trace (mtrace) and debug memory allocation library (dmalloc).

www.manaraa.com

3. It has no internal memory leak. It generates a balanced allocation and deallocation transactions. Before it terminates, it deallocates all undeallocated objects. Thus, the only memory leak in the trace files is our previously known injected leak. This allows a more deterministic analysis of the memory leak detection algorithm.

Our benchmark consists of two files *mldbench.h* and *mldbench.c*. The source code for these files is provided in appendix B.

5.2.1.2 Data Collection

In order to generate and validate trace files, we perform the following steps:

1. Tune parameters in mldbench.h file. The mldbench.h contains several parameters that control the output nature of the trace files. Not all real-world applications allocate and deallocate memory in the same manner. Some applications allocate small-sized objects, while others allocate medium or large-sized objects. Some applications hold allocated objects for a long time; others hold objects for a short time and so on. Tuning the benchmark makes its output similar to a target application. We list the parameters used in mldbench.h file:

www.manaraa.com

a) MAX_TRANS: the maximum number of allocations and deallocations to create. The larger the value the lengthier the trace file will be.
b) SEED: the seed value to the random function.
c) P_MALLOC: the probability that malloc() will be called.
d) P_FREE: the probability that free() will be called. P_MALLOC and P_FREE sum to one. Setting up P_MALLOC to a value higher than P_FREE makes the benchmark allocate more often than it deallocates, i.e., creates a leaky environment.
e) P_MIN_SIZE: probability of allocating objects of small size. Small sized objects belong to the closed interval [MIN_SMALL_CHUNCK_SIZE, MAX_SMALL_CHUNCK_SIZE]
f) P_MEDIUM_SIZE: probability of allocating medium sized objects. Medium sized objects belong to the closed interval [MIN_MEDIUM_CHUNCK_SIZE, MAX_MEDIUM_CHUNCK_SIZE]
g) P_LARGE_SIZE: probability of allocating large sized objects. Large sized objects belong to the closed interval[MIN_LARGE_CHUNCK_SIZE, MAX_LARGE_CHUNCK_SIZE]

www.manaraa.com

h)

Table X shows the initial setup we have used for these initial parameters

Parameter Name	Value
MAX_TRANS	10000, 100000
SEED	12755765
P_MALLOC	0.50
P-FREE	0.50
P_MIN_SIZE	0.85
P_MEDIUM_SIZE	0.15
P_LARGE_SIZE	0.05
MIN_SMALL_CHUNCK_SIZE	1 byte
MAX_SMALL_CHUNCK_SIZE	256 byte
MIN_MEDIUM_CHUNCK_SIZE	257 byte
MAX_MEDIUM_CHUNCK_SIZE	4 Kbyte
MIN_LARGE_CHUNCK_SIZE	4 Kbyte +1 byte
MAX_LARGE_CHUNCK_SIZE	10 Kbyte

Table 5: Benchmark used parameters

The user is encouraged to tune up the mldbench.h to the parameters that he/she thinks are more close to the target application being simulated.

2. Compile and run. We compiled and ran the benchmark using gcc compiler under Suse Linux 9.0. For each different value of MAX_TRANS parameter we get a different trace file.

www.manaraa.com

3. Create a balanced trace. The benchmark is supposed to generate a balanced trace file for every single run. A balanced trace file contains a deallocation transaction for every allocation transaction. A balanced trace file has no memory leak. Before the benchmark terminates, it deallocates all the remaining undeallocated objects. Later on, we show how we introduce a known leak in the trace files (leak injection) and show how much of that leak is detected by our MLD algorithm.
4. Validate with mtrace and dmalloc. We validate the benchmark and trace files using two well-known debugging tools: memory trace tool (mtrace) and debug memory allocation library (dmalloc). Both tools reported that the benchmark generates no memory leak in any created trace file. Trace files can vary in size from small to large depending on the input parameters.

For clarity purposes, we provide figure 24 to show a snapshot of one of these trace files:

```
MAX TRANS: 100000
Trans   ra          size
+       0x804a008   5338
+       0x804b4e8   118
+       0x804b568   7010
+       0x804d0d0   200
-       0x804b568   7010
+       0x804b568   222
-       0x804a008   5338
-       0x804b4e8   118
+       0x804a008   67
-       0x804a008   67
-       0x804d0d0   200
+       0x804a008   243
+       0x804b650   8564
-       0x804b568   222
```

FIGURE 24: A SNAPSHOT OF ONE OF THE TRACE FILES

www.manaraa.com

The plus (+) sign means allocation and the minus (-) sign means deallocation. The remaining values represent the return address, in hexadecimal format, and the size of each allocated or deallocated chunk in bytes.

We show how these traces are used in the trace driven simulation model.

5.2.1.3 Leak Injection

The benchmark, as already discussed, generates a balanced trace files. A balanced trace file contains a deallocation transaction for every allocated object. So, the trace file has no memory leak. Before the benchmark terminates, it deallocates all the remaining undeallocated objects.

In order to inject leak in the trace files, we simply mark the deallocation transactions that represent the free() function, in the trace files, as if they were deleted or commented. These transactions are marked and not deleted. This mark is used by the simulation program to know the exact point where a live object becomes unreachable. Unreachable objects will age because they are no longer accessed by the application and they will be detected by the MLD aging algorithm. For example, table 6 shows the accumulated number and size of the leak that we inject in two trace files generated by the benchmark. We will find out how much of the injected leak is recovered by the MLD algorithm.

www.manaraa.com

Trace File for benchmark	Injected leak in the trace file	
	No of objects	Size (byte)
	(a)	(b)
Trace0	4976	3401804
Trace1	49855	34197871

Table 6: An example of accumulated amount of injected leak in two trace files

5.2.2 Trace-Driven Simulation (Stage Two)

The trace-driven simulation program, stage two, starts by reading a trace file and the simulation input parameters (Input parameters are listed in appendix A; table A.1). The program runs until the trace file has no more transactions. At the end of simulation execution, the program writes output parameters to one or more output files. Output parameters are listed in appendix A; table A.2.

In each single run on a trace file, the simulation program determines the next event the target process (*Pi*) is going to go through. The process will enter one of the following events: Memory allocation event, free event, or memory access event. Although, in reality, a process might be in some other events, these are the only events of most interest in order to monitor virtual address space. A process might be allocating memory and, as a result, increasing the heap size and consuming virtual address space. A process might be freeing memory and, therefore, saving virtual address space. Or, a process might be accessing memory.

www.manaraa.com

If a memory reference is made to a page that is available in memory, then the reference is completed and no further action is required. Otherwise; its corresponding page is paged-in and its time stamp (TS) is reset according to the bookkeeping part of the MLD algorithm. If there is no space in RAM for the requested page, a victim page is selected for replacement and the current time is written into its corresponding page table entry (TS).

To simulate memory allocation and deallocation events, we have used our own versions of malloc() and free() functions and implemented the changes that we proposed in section 3.1. For memory access event, we developed a function

www.manaraa.com

that simulates demand paging. Bookkeeping part of the algorithm was implemented in the page replacement policy. The page replacement policy used, in this simulation, is implemented using least recently used (LRU) strategy. The source code of MLD simulation program is provided in appendix C.

Determination of the next event is governed by the data available in the trace files. Each transaction in the trace files contains a sign, return address, and a size. Plus sign means allocation, and minus sign means deallocation. The remaining values represent the return address in hexadecimal format and the size of each allocated or deallocated chunk. The probability of performing memory access is an input parameter. Processes that suffer from memory leak problem usually call malloc() more often than free(). Trace files govern how and when malloc and free will be called. In reality, calling malloc() and free() are process dependent. On one hand, we may find a process with no single memory allocation or deallocation statement. On the other hand, the majority of the statements in some other processes might be memory allocation and deallocation.

In the next sub-sections, we introduce the simulation assumptions, the input and output parameters, the simulation model, and discuss several experiments that illustrate the MLD performance measures.

www.manaraa.com

5.2.3 Simulation Assumptions

The following assumptions are taken under consideration:

- The demand paging is used.
- A global replacement strategy is used. Victim pages are selected according to LRU strategy. The impact of the replacement policy on the MLD is not investigated in this thesis. This could be the subject of a future research.
- Trace files are considered valid representations to real-world application.
- A process under simulation is highly influenced by the input parameters such as RAM size, Max heap size, heap size threshold, page_age_threshold, whether it is running with other processors or running alone, and page file size.

5.2.4 Input and Output Parameters

Appendix A lists the input and output parameters used in the simulation program respectively.

5.2.5 Simulation Model

According to one of the input parameters, AgingFlag (Appendix A), the simulation program can run a simulated process according to the following two models:

www.manaraa.com

a. Demand paging model: if the input AgingFlag is reset the simulation program runs a process according to the normal demand paging model and does not perform any action of the MLD algorithm.
b. Demand paging model with the MLD: if the input AgingFlag is set the simulation program runs a process according to the modified demand paging model that implements the MLD algorithm.

AgingFlag is used so that the same simulation program can compare the behavior of a process under the normal situation with the behavior of the same process under the MLD algorithm.

5.2.5.1 Simulation Model (Demand Paging)

This simulation model models the virtual memory system. Virtual memory is a technique that allows processes to execute while not being completely available in physical memory. We have chosen to implement the model by demand paging. This is the most commonly used approach in real-life operating systems.

Process 1 Process 2 Process 3 Process n

www.manaraa.com

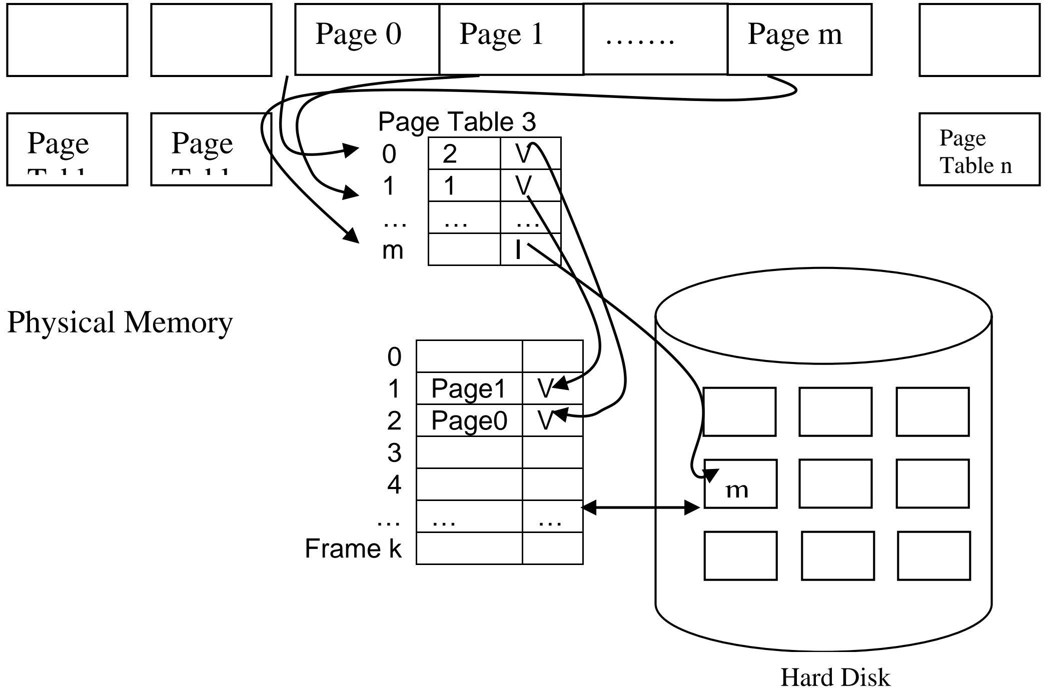


Figure 25: Virtual memory system implemented by demand paging

As shown in figure 25, the simulation model can run (n) number of processes where n is an input parameter. Each process has its own page table

and competes with other processes on the shared physical memory. The figure shows how pages one and two of process three are both valid and available in memory. It also shows that page (m) is invalid and available on disk and it has to be swapped-in once needed.

www.manaraa.com

5.2.5.2 Simulation Model of Demand Paging with MLD Algorithm

The simulation model is updated to reflect the MLD major parts listed in the block diagram (figure 6). We show the model with the MLD algorithm in figure 26. In addition to demand paging, the model now implements bookkeeping() and the sweeper()

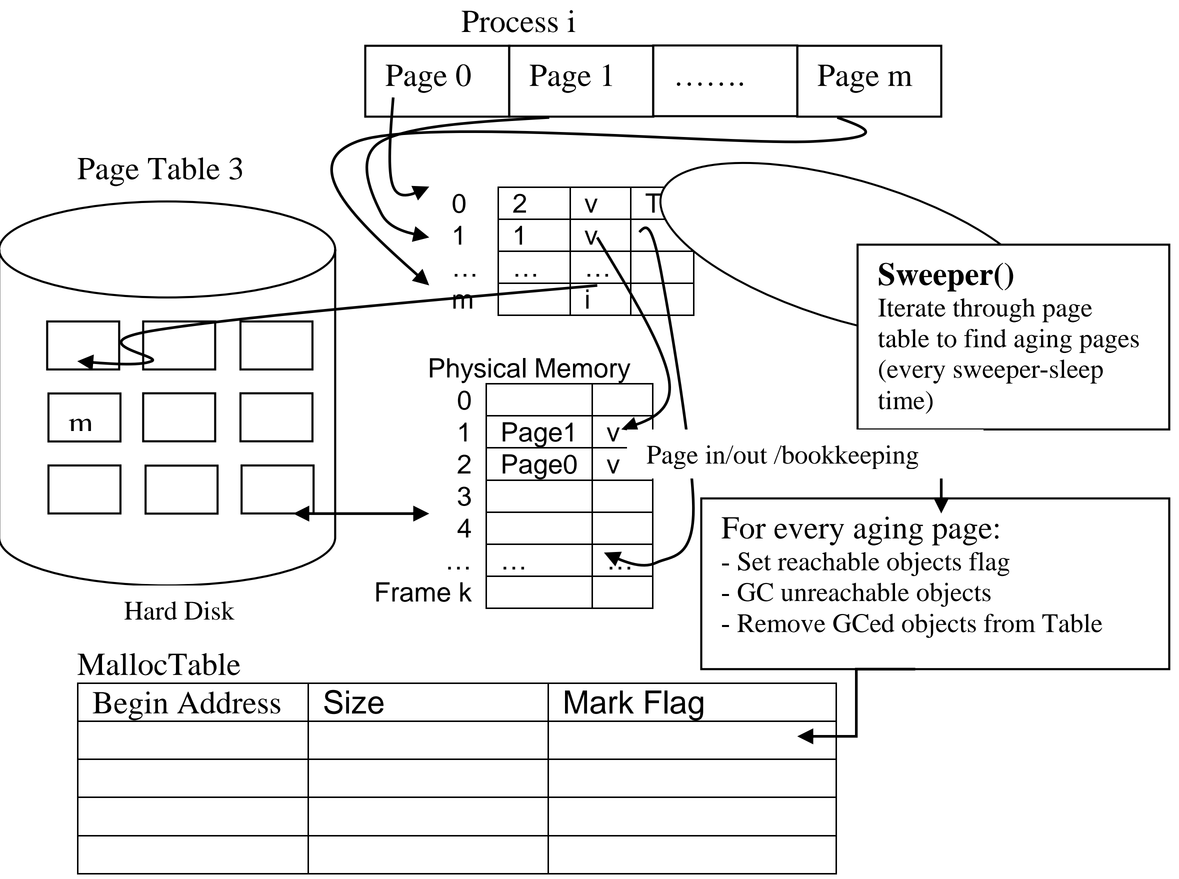


FIGURE 26: VIRTUAL MEMORY SYSTEM IMPLEMENTED BY DEMAND PAGING WITH MLD

المنارة للاستشارات
www.manaraa.com

5.2.6 Simulation Results

5.2.6.1 Time versus Heap Size

In this experiment, shown in figure 27, we run two processes; process zero (p0) and process one (p1). We set the input parameters to make both processes read from the same trace file. The only difference between p0 and p1 is that p1 implements the MLD algorithm and p0 does not. The Heap_Size_Threshold for p1 is set to 80% of the maximum heap size. We run the experiment and record the following statistics versus time as shown in table 7.

Time	MaxHeapSize	80%Theshold	P0_HeapMaxSize	P1_HeapMaxSizeW_MLD
1000	2097152	1677721.6	169441	169441
2000	2097152	1677721.6	341528	341528
3000	2097152	1677721.6	509593	509593
4000	2097152	1677721.6	619021	619021
5000	2097152	1677721.6	761058	761058
6000	2097152	1677721.6	900351	900351
7000	2097152	1677721.6	1049641	1049641
8000	2097152	1677721.6	1234014	1234014
9000	2097152	1677721.6	1405741	1405741
10000	2097152	1677721.6	1557247	1557247
11000	2097152	1677721.6	1745974	1682868
12000	2097152	1677721.6	1864535	1682868
13000	2097152	1677721.6	2050102	1692461
14000	2097152	1677721.6	2195087	1711977
15000	2097152	1677721.6	2380028	1731262
16000	2097152	1677721.6	2600321	1791244
17000	2097152	1677721.6	2748867	1825067
18000	2097152	1677721.6	2946119	1888295
19000	2097152	1677721.6	3114711	1967552
20000	2097152	1677721.6	3287400	2028340

www.manaraa.com

21000	2097152	1677721.6	3509536	2123939
22000	2097152	1677721.6	3688190	2189165
23000	2097152	1677721.6	3713226	2206496
24000	2097152	1677721.6	3713226	2206496

Table 7: Heap Size versus Time

The Heap_Size_Threshold for p1 is set to 80% of the maximum heap size as shown in the triangle-line in figure 27. The maximum heap size is an input parameter for this experiment and is shown as a squared-line. Theoretically, maximum heap size in some platforms is equal to 2^{32} bytes.

The experiment results show that heap size is growing with time for both processes which is expected since they allocate and never deallocate memory. P0

crosses the maximum heap size border at time (t1 = 20000). p1 keeps running for a longer period than p0. p1 activates the sweeper just after the threshold of 80% is reached as shown in point (S = 11000). The sweeper makes extra room for new allocations. However, p1 also crosses the line of the maximum heap size but at time (t2). In this simulation, we allow processes to go beyond the maximum size of the heap in order to collect statistics.

www.manaraa.com

In reality, p0 will crash at t1 whereas p1 will crash at t2. t1 and t2 are the points in time where heap space is exhausted for processes p0 and p1 respectively. We conclude from this experiment that p1 with the MLD algorithm lives longer than p0.

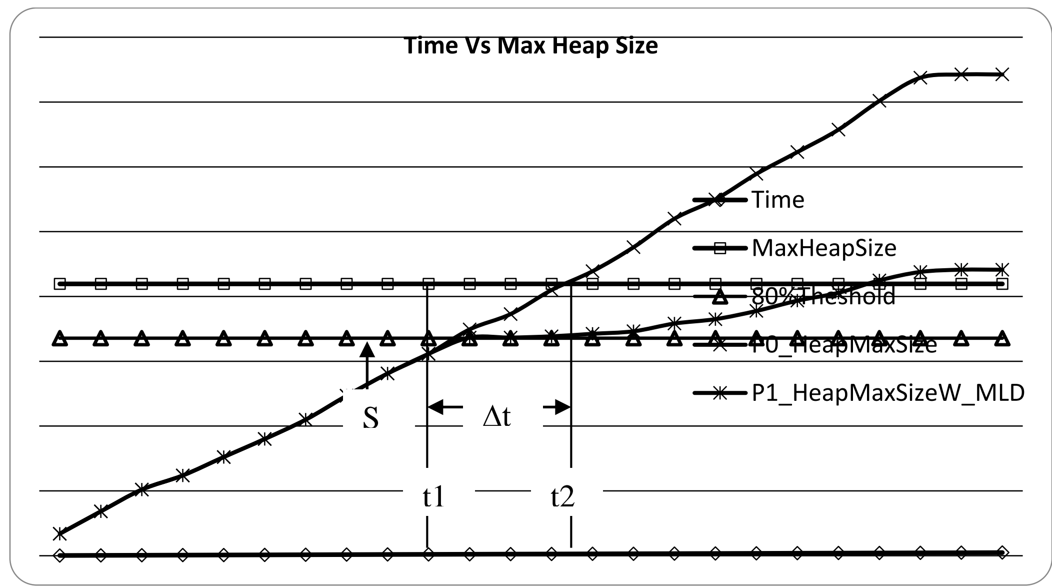


Figure 27: Experiment One: Time Vs Heap Size

The scenario of this experiment was one of the worst possible scenarios and yet, p1 with MLD proved to live longer for a delta time period equal to t2-t1 as follows:

$$\Delta t = t2 - t1$$

Where:

www.manaraa.com

t1 is the crash time of process (p0) that does not implement the MLD and

t2 is the crash time of process (p1) that implements the MLD

In more relaxed experiments (scenarios), the delta time period may be big enough to satisfy the customer of the process p1 or makes p1 just survive until critical time in mission-critical applications is passed. After all, removing unreachable objects and saving the corresponding space on the heap for future allocations is better than just doing nothing.

5.2.6.2 Page Age Threshold (Page_Age_Theshold) Vs False Negatives and Overhead Cost

In this experiment, we test the effect of choosing different values for Page_Age_Threshold, an input parameter to the MLD, on the number of false negative objects and overhead cost. False negative objects are unreachable objects that were not identified by the MLD algorithm. The simulation is set to run the same trace file ten different times as shown in table 8. In each single run, we record the number of false negative objects and the overhead. The overhead is a counter that represents how many times the sweeper() is called. Calling the sweeper() more often means incurring more cost and vice versa. The Page_Age_Threshold is computed dynamically to be equal to a constant (K), an input parameter, times the average page age(AvgAge) as follows:

$$Page_Age_Threshold = K(AvgAge)$$

Where: K is a constant parameter,

www.manaraa.com

AvgAge is the average page age that is computed accumulatively during run time and adjusted to include every aging page, and OverHead is the number of times the sweeper() is called.

Trace file, trace0, is used for this particular experiment. From table 6, we already know that this file has 4976 injected leaky objects with total size of 3401804 bytes. Trace file; trace1, is a lengthier file and it produces a close results.

# of run	Constant (K)	#Injected leaky objects (a)	#Recovered objects (b)	#False NEGs (a-b)	Percentage of false NEGs to injected (a-b)/a *100	OverHead counter
1	0.05	4976	2627	2349	47%	6676
2	0.1	4976	2603	2373	48%	6561
3	0.15	4976	2626	2350	47%	6176
4	0.25	4976	2577	2399	48%	5896
5	0.5	4976	2475	2501	50%	5026
6	0.75	4976	2394	2582	52%	4267
7	1	4976	2384	2592	52%	3750
8	1.25	4976	2257	2719	55%	3165
9	1.5	4976	2125	2851	57%	2820
10	1.75	4976	2041	2935	59%	2399

Table 8: False Negatives Vs different values of Page_Age_Threshold

On one hand, as shown in figure 28 in the Xed-line, the number of false negatives is proportionally increasing with the increase in the Page_Age_Theshold. This result is expected since the algorithm will identify less leaky pages as the Page_Age_Threshold increases.

www.manaraa.com

In fact, if the Page_Age_Threshold is very small then most of the pages in the virtual space will be aging pages and the MLD will look into them for leak. In this case, a few

unreachable objects will go undetected which decreases the number of false negatives. If the Page_Age_Threshold value is extremely high the MLD will rarely find an aging page which increases false negatives.

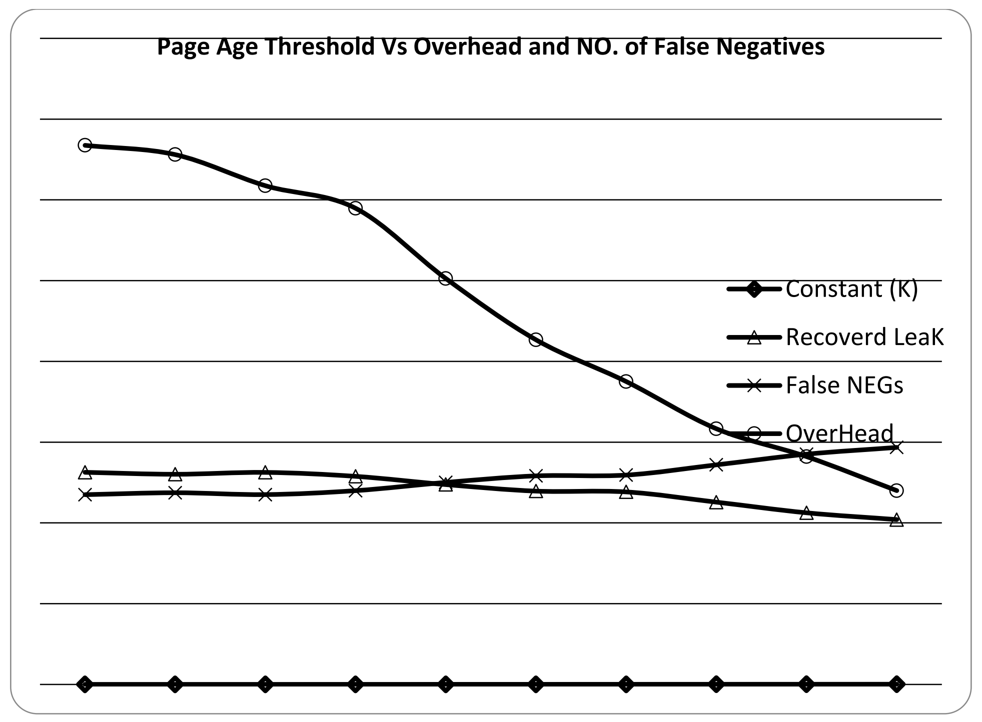


FIGURE 28: PAGE_AGE_THRESHOLD VS NO OF FALSE NEGATIVES AND OVERHEAD

المنارة للاستشارات
www.manaraa.com

On the other hand, the overhead value associated with calling the sweeper(), shown in figure 28, the circled-line, decreases as the page age threshold increases. This result is also expected. The MLD calls the sweeper() less often as the Page_Age_Theshold increases.

The triangle-line goes exactly the reverse of the Xed-line. We provide it for clarity purposes. As the number of false negative objects increases the number of the recovered objects decreases and vice versa.

This experiment concludes that small Page_Age_Threshold values lead to small number of false negatives and high overhead and vice versa. In this case our recommendation is system dependent. If the system can tolerate high overhead cost, then use a small value for Page_Age_Threshold and minimize the number of false negatives; otherwise increase the Page_Age_Threshold as much as the system can tolerate the overhead cost. Some systems can tolerate overhead cost by using parallel programming and multiple processors. In this case, the sweeper() can be assigned to a set of processors having the system not worry about the overhead. As mentioned earlier, a telemetry tool can provide a great help in tuning Page_Age_Threshold parameter and make the administrator visualize the effect of tuning this parameter on the overhead cost.

www.manaraa.com

5.2.6.3 Page Size Vs False Negatives

This experiment shows the relation of page size to the number of false negative objects. We run this experiment on trace file, trace0, by varying page size from 1Kb to 16KB and record the statistics shown in table 9.

Page Size(KB)	#Recovered Objects	#FalseNegs Objects
1	2314	2662
2	2531	2445
4	2698	2278
8	2829	2147
16	3021	1955

Table 9: page size versus false negative objects

The data is plotted in figure 29 below

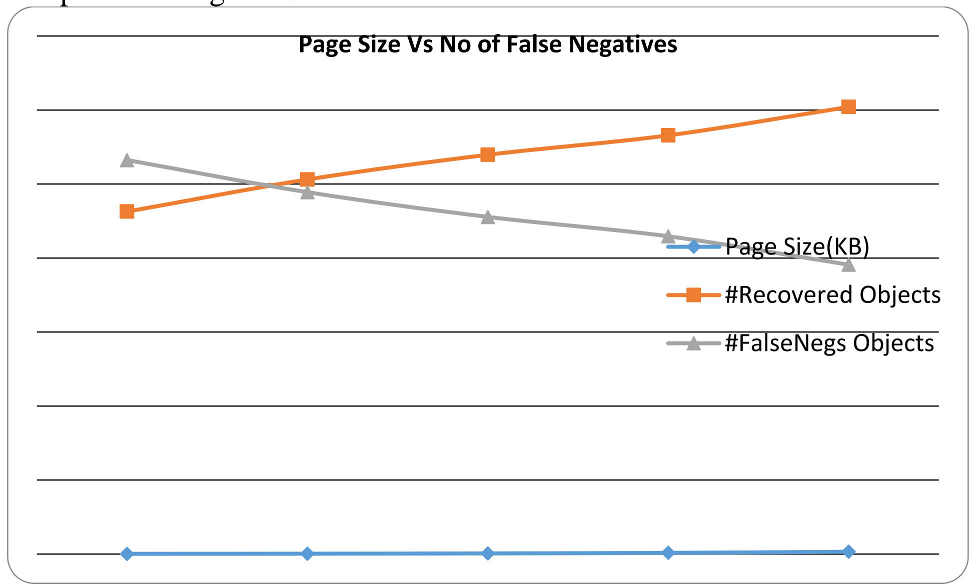


Figure 29: Page Size Versus Number of False Negative objects

www.manaraa.com

The figure shows that as the page size increases the number of false negatives decreases. This occurs because once a page is aged all of the unreachable chunks it contains will be removed altogether. The higher the page in size the higher the number of unreachable chunks it contains. Once these chunks are removed the number of recovered objects will be increased and therefore the number of false negatives will be decreased.

This finding is contrary to the intuitive argument, that if only one chunk in the virtual space is active, while the other chunks have leaked, then the corresponding physical page remains active and the leaks in that page will not be detected. This argument is true to a certain degree, but the global replacement strategy of LRU algorithm will force some pages to age by choosing them as victim pages even though they contain live chunks. The sweeper() will detect such aging pages if it is set to work on a small page_age_threshold. The sweeper() will remove

www.manaraa.com

all of the unreachable chunks in these pages even if it has a relatively smaller age. The higher the size of the page the higher the number of the unreachable chunks it contains, and therefore the smaller the number of false negatives will be.

5.3 Simulation Results of the MLDR

A trace-driven simulation program is built in order to validate, verify, and provide a proof of concept to the MLDR algorithm. In the next sub sections, we discuss the major results of this simulation program.

5.3.1 Time versus Heap Size

Time is shown versus heap size in figure 30. In this experiment, we run two processes; process zero (p0) and process one (p1). We set the input parameters to make both processes read from the same trace file. The only difference between p0 and p1 is that p1 implements the MLDR algorithm and p0 does not. We run the experiment and record the following statistics versus time as shown in table 10. The Maximum Heap Size is system dependent. For this experiment, the maximum heap size is an input parameter.

www.manaraa.com

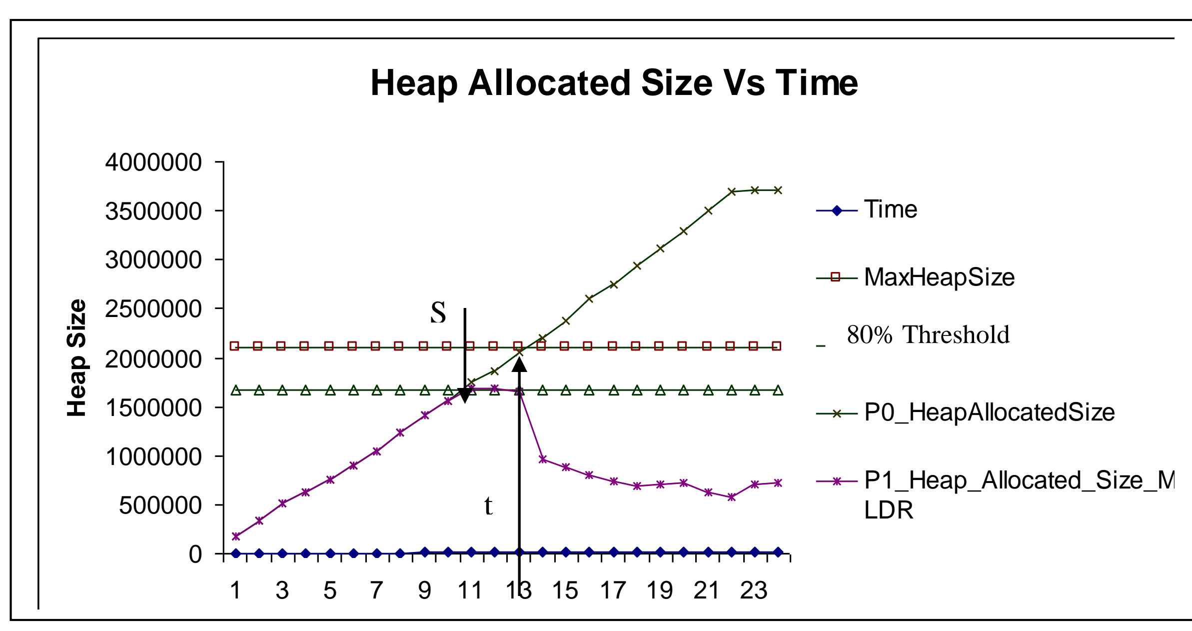
Time	MaxHeapSize	80%Theshold	P0_HeapAllocatedSize	P1_Heap_Allocated_Size_MLDR
1000	2097152	1677722	169441	169441
2000	2097152	1677722	341528	341528
3000	2097152	1677722	509593	509593
4000	2097152	1677722	619021	619021
5000	2097152	1677722	761058	761058
6000	2097152	1677722	900351	900351
7000	2097152	1677722	1049641	1049641
8000	2097152	1677722	1234014	1234014
9000	2097152	1677722	1405741	1405741
10000	2097152	1677722	1557247	1557247
11000	2097152	1677722	1745974	1682868
12000	2097152	1677722	1864535	1682868
13000	2097152	1677722	2050102	1658285
14000	2097152	1677722	2195087	965517
15000	2097152	1677722	2380028	889703
16000	2097152	1677722	2600321	804142
17000	2097152	1677722	2748867	743908
18000	2097152	1677722	2946119	698117
19000	2097152	1677722	3114711	710462
20000	2097152	1677722	3287400	719379
21000	2097152	1677722	3509536	630856
22000	2097152	1677722	3688190	581057
23000	2097152	1677722	3713226	710462
24000	2097152	1677722	3713226	719379

Table 10: Heap Size versus Time

www.manaraa.com

The Heap_Size_Threshold for p1 is set to 80% of the maximum heap size as shown in the triangled-line in figure 30. The maximum heap size is shown as a squared-line. Theoretically, the maximum heap size in some platforms is equal to 2^{32} bytes.

The experiment results show that heap size is growing with time for both processes. This result is expected since both processes allocate and never deallocate memory. P1 does not start deallocating until heap size threshold is reached. P0 crosses the maximum heap size border at time (t). The simulation allows the program to run beyond the maximum heap size in order to record statistics. In reality, time (t) is the time at which process p0 crashes due to failure in allocating additional space for the application to keep going. p1 starts the backup and aging module of the MLDR at point (S) when the 80% threshold is reached. The backup and aging module of the MLDR makes extra room for new allocations. By removing unreachable chunks and backing up aging reachable chunks for p1,



www.manaraa.com

we can save enough room for p1 to keep going.

Figure 30: Experiment One: Time Vs Heap Size

5.3.2 Time versus Heap Size for both of the MLD and MLDR

For the sake of comparison among a regular process, a process with the MLD and a process with the MLDR, we repeat the previous experiment by showing these three types of processes.

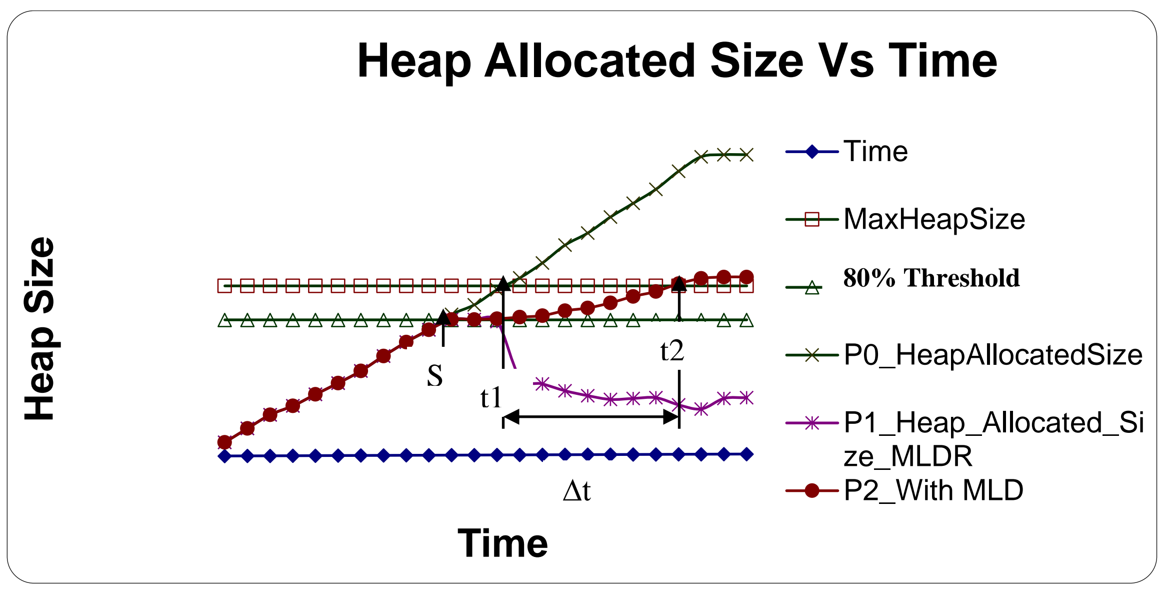


Figure 31: Time versus heap size comparison between MLD and MLDR

Figure 31 shows the same processes listed in Figure 30. However, this time we include an additional process(p2). P2 is a process that implements the MLD as explained in chapter 3 of this dissertation. P2 is shown as a filled-circled line. The figure shows the following results: a regular process crashes once the maximum heap size limit is reached. A process with the MLD starts sweeping and saves extra room for new allocations and makes the application live longer for a delta time period. In some scenarios, a process with the MLD will crash if it fails to satisfy allocation requests. A process with the MLDR, however, provides much more room

www.manaraa.com

once a heap size threshold is reached and prevents crashing. This result is not always true because it is dependent on some input parameters that we have already discussed in crash preventing in chapters 3 and 4.

5.3.3Time versus Heap Size and Disk Space Used

The backup and recovery mechanism of the MLDR works by saving space on the virtual address space. The MLDR moves potentially leaky objects to disk. Table 11 lists the results of this experiment; time versus heap allocated size and disk space used given 80% threshold value is used.

Time	80%Threshold	Heap Allocated Size	Disk Space Used
1000	1677722	57359	0
2000	1677722	195695	0
3000	1677722	265621	0
4000	1677722	358133	0
5000	1677722	422128	0
6000	1677722	513125	0
7000	1677722	580893	0
8000	1677722	628770	0
9000	1677722	723789	0
10000	1677722	824466	0
11000	1677722	955641	0
12000	1677722	1061927	0
13000	1677722	1195961	0
14000	1677722	1333416	0
15000	1677722	1448488	0
16000	1677722	1598918	0
17000	1677722	1677722	0
18000	1677722	980570	697254
19000	1677722	918643	759181
20000	1677722	839608	838216
21000	1677722	747474	930350
22000	1677722	703162	974662

www.manaraa.com

23000	1677722	679870	1017864
24000	1677722	699729	1008085
25000	1677722	634412	1083417
26000	1677722	573913	1153619
27000	1677722	548118	1187049
28000	1677722	564793	1215291
29000	1677722	613900	1231088
30000	1677722	631011	1283338
31000	1677722	635194	1308675

Table 11: Time versus heap size and used disk space

The table results are shown in Figure 32. The figure shows that when a process reaches the threshold line it starts the backup and recovery module. It backs up the potentially leaky chunks to disk.

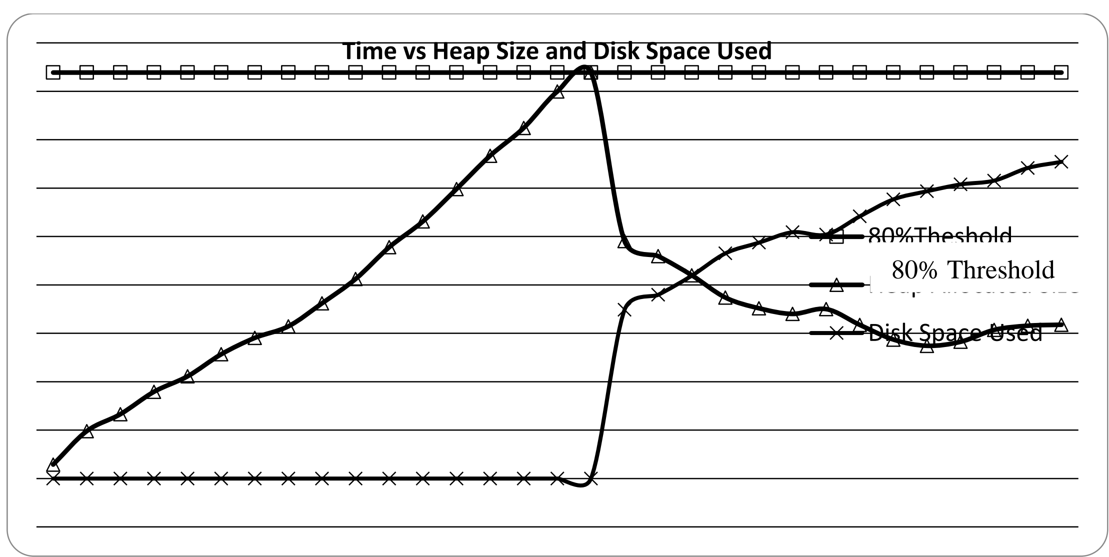


Figure 32: Time versus disk space used

The disk space used remains zero as long as the heap size threshold is not reached. Once the threshold limit is reached, the recovery and backup of MLDR starts working and, as a result, the disk space used starts increasing and the heap allocated size starts decreasing.

www.manaraa.com

5.3.4 Page Age Threshold Vs False Negatives, False Positives and Overhead Cost

In this experiment, we test the effect of choosing different values for Page_Age_Threshold, an input parameter to MLD, on the number of false negative objects, false positive objects, and overhead cost. False negative objects are unreachable objects that have not been identified by the MLD algorithm. False positive objects are objects that are identified as leaky and have been dereferenced after the system has moved them to disk. Overhead cost is associated with the number of disk backup and recovery operations. Increasing the number of backing up objects or recovering them has a performance cost. The simulation is set to run the same trace file ten different times as shown in table 12. In each single run, we record the number of false negative objects, false positive objects, and the overhead. The experiment is conducted on trace file (trace0) and the results are shown in figure 33.

# of run	Constant (K)	#Injected leaky objects (a)	#Recovered objects (b)	#False NEGs (a-b)	#False Positives	Overhead Cost
1	0.05	4976	2596	2618	11024	24213
2	0.1	4976	2609	2603.7	10472	23092
3	0.15	4976	2596	2618	10479	23117
4	0.25	4976	2520	2701.6	10081	22236
5	0.5	4976	2479	2746.7	8332	18553
6	0.75	4976	2347	2891.9	7109	16007
7	1	4976	2318	2923.8	5804	13380
8	1.25	4976	2207	3045.9	4225	9992
9	1.5	4976	2185	3070.1	2868	7170
10	1.75	4976	2107	3155.9	1603	4571

Table 12: Page age threshold versus False Negatives, false positives and overhead

www.manaraa.com

The results are shown in figure 33.

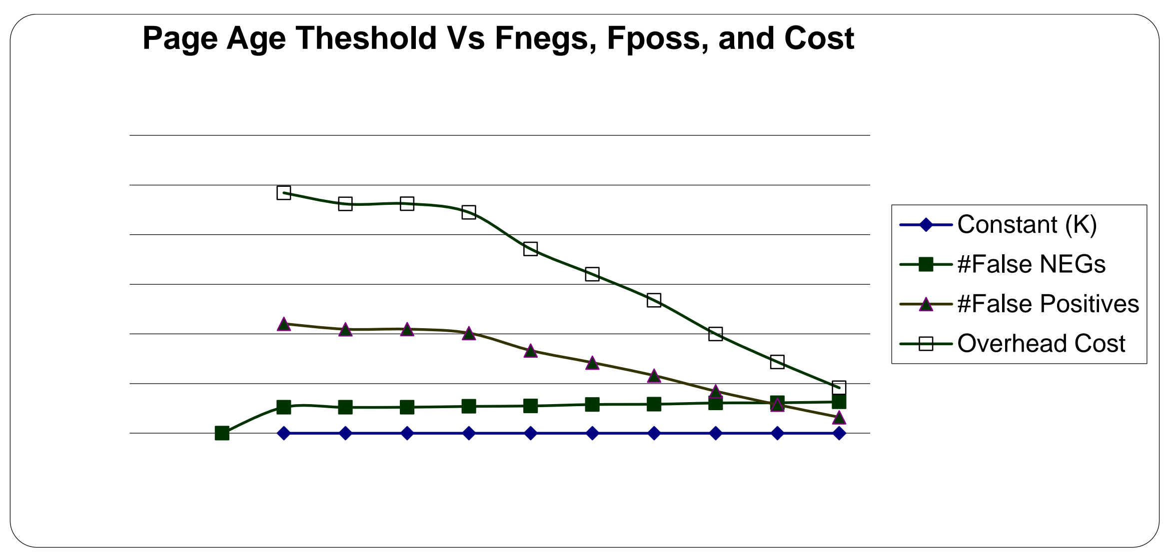


FIGURE 33: PAGE_AGE_THRESHOLD VS NO OF FALSE NEGATIVES AND OVERHEAD

On the one hand, the number of false negatives as represented by a filled-squared-line is proportionally increasing as the Page_Age_Theshold increases. This result is expected since the algorithm will identify less leaky pages as the Page_Age_Threshold increases. In fact, if the Page_Age_Threshold is very small then most of the pages in the virtual space will be aging pages and the MLDR will look into them for leak. In this case, a few unreachable objects will go undetected which decreases the number of false negatives. If the Page_Age_Threshold value is extremely high the MLDR will rarely find an aging page which increases false negatives.

On the other hand, both of the number of false positives and the overhead cost value associated with the backup and recovery module decrease as the page age threshold increases. This result is also expected. The MLDR will call the backup and recovery module less often as the Page_Age_Theshold increases.

www.manaraa.com

This experiment concludes that small Page_Age_Threshold values are associated with small number of false negatives and a relatively large number of false positives and an increase in the overhead cost and vice versa. In this case, our recommendation is system dependent. If the system can tolerate high overhead cost, then use a small value for Page_Age_Threshold and minimize the number of false negatives; otherwise increase the Page_Age_Threshold as much as the system can tolerate the overhead cost. Some systems can tolerate overhead cost by using parallel programming and multiple processors. In this case, the backup and recovery module can be assigned to a set of processors and use a fast accessed disks. As mentioned in chapter 3, a telemetry tool can provide a great help in tuning the MLD. A telemetry tool can also be very handy in tuning up the MLDR and it's parameters in order to reduce false negatives under a given tolerable cost.

5.4 Comparing MLD and MLDR to Current Solutions

In this section, we compare the MLD and the MLDR to current memory leak solutions. We list both of similarities and differences. First, we compare the MLD and the MLDR to SWAT, a well-known debugging tool used by Microsoft group. Then, we compare the MLD and the MLDR to Garbage collectors.

www.manaraa.com

5.4.1 MLD and MLDR versus SWAT

Table 13 compares SWAT to the MLD and MLDR according to the following criteria: i) goal of the tool, ii) memory leak detection, iii) age tracking, iv) memory leak reporting, v) false negatives , vi) false positives, vii) time used, viii) stale objects , ix) staleness predicate, and x) overhead.

Goal of the tool	SWAT: - Provide run-time checking (debugging) tool. - Can ship with production code in order to detect errors during real use. - Can not delay or prevent crashing. It does not provide a run-time solution. It can not decide by its own to delete unreachable objects. MLD and MLDR: - MLD provides a partial run-time solution - MLDR provides a complete run-time solution - Can delay or prevent crash by making enough room to new allocations via removing unreachable objects.
Memory leak detection	SWAT: If a heap object has not been accessed for a long time then it is a memory leak. MLD and MLDR: If a page in the page table has not been accessed for a long time then it is a potential leaky page that may contain leaky objects.

المنارة للاستشارات
www.manaraa.com

Age tracking	SWAT: - Age is tracked per every single object in the heap (*the term age is not used explicitly in SWAT*) - Since these objects in the heap are very large a sampling approach is used to build a heap model for all allocated objects from a statistical sampling trace of access to heap virtual address space. MLD and MLDR: - Age is tracked per pages - Use hardware available in physical memory. - Time stamp the corresponding page in page table of the paged-out page.
Memory Leak Reporting	SWAT: At the end of an application run. (*The normal behavior of a checking/debugging tool*) MLD and MLDR: Found leaks are removed if they are unreachable in MLD. The MLDR removes during run-time both of stale and unreachable chunks.
False negatives	SWAT: False negatives are not generated since tracking is made per each object in the heap. MLD and MLDR: False negatives might be generated. Tracking liveness per page may make some leaky chunks in some pages go undetected because some other chunks are alive.

www.manaraa.com

False positives	SWAT: Some of leak detected is false positive. It is the responsibility of the developer to decide whether these leaks are real or NOT. MLD and MLDR: - MLD produces zero false positives because it uses a conservative approach. - MLDR handles the problem of false positives by providing a recovery module.
Time used	SWAT: Use number of accesses to represent time. Does not use wall clock time to measure staleness, this avoids labeling objects of an interactive application that left idle overnight as stale. MLD and MLDR: Use clock time to time stamp last use of a page. The time threshold can be well-tuned to avoid misidentifying idle objects in interactive applications as stale.
Stale objects	SWAT: Can detect stale objects and report them to the developer. MLD and MLDR: The MLD can only handle unreachable objects. The MLD can theoretically detect stale objects but it does not because it can not decide whether to remove these objects or not and contribute to the overall run-time solution by providing extra room for new allocations. The MLDR can detect and handle both of the stale and unreachable objects and can contribute to the overall run-time solution by removing these objects and recover in case they were misidentified.

www.manaraa.com

Staleness predicate	SWAT: Staleness predicate decides whether an object is leaked or not according to: a. (never accessed) if the object is never accessed b. (constant time) if the idle time is greater than a threshold c. (Active time) if the idle time is greater than (N * active time). If the object has been active for a long time it is allowed to be inactive for a long time. MLD and MLDR: If the page age is greater than a tunable threshold. This tunable threshold can replace all of the three suggested measures used in SWAT.
Overhead	SWAT: Minimize overhead by sampling the accesses made to the heap. MLD and MLDR: - MLD and MLDR track pages and SWAT track individual chunks. - Time stamping is more efficient in MLD and MLDR since we exploit the physical available hardware. - SWAT spends considerable time in searching the heap model and keeping track of time.

Table 13: The MLD and MLDR versus SWAT

www.manaraa.com

5.4.2 MLD and MLDR versus Garbage Collectors

All of the MLD, MLDR and garbage collectors provide some sort of a run-time solution to memory leak problem. Garbage collectors are limited to languages designed with garbage collection in mind. Garbage collectors can only handle unreachable objects. They can not recover stale objects because the virtual memory system does not allow for such a solution. The MLD and MLDR are not limited to specific languages. Both of the MLD and MLDR utilize the aging in the physical memory (a new approach) to detect memory leak in the virtual address space. The MLD is similar to garbage collectors in terms in can handle unreachable objects by using a conservative approach; whereas the MLDR can handle both of the unreachable and stale objects. The MLDR along with the ML-VMS is the only algorithm, to our knowledge, that provides a complete run-time solution. The MLDR can decide during run-time to remove stale objects to save extra room in the virtual address space. If these stale objects are turned out to be false positives then the MLDR is able to recover these objects.

www.manaraa.com

5.5 Conclusion

This chapter analyzes the performance of the MLD, MLDR, and the ML-VMS. The MLD have the complexity of $O(n^2)$. The ML-VMS has significant effect on the performance of the computer system. ML-VMS adds additional layer on the demand paging system. In addition to the cost paid by the demand paging memory system, ML-VMS may incur the cost of accessing a disk either to backup a chunk or to recover another chunk in case of false positives.

The performance cost can be reduced by 1) reducing the probability of incurring false positives to the minimum. Increasing the page age threshold value reduces the false positive rate and, as a result, reduces the performance cost, 2) storing the backed up objects on the swap space instead of the regular disk. Swap space is usually faster than that of the file system and 3) suggesting parallel programming to enhance the performance.

The MLD is validated, verified and proved to be sound using a trace-driven simulation program. A benchmark was designed and built to provide the simulation program with allocation and deallocation transactions that simulate a target application. The simulation results have shown that the MLD is capable of removing unreachable objects and provide more room for new allocations. Applications that exploit this algorithm are shown to live longer than the applications without it. The false negative rates and overhead were shown to be highly dependent on some input parameters like Page_Age_Threshold and system parameters like Page_Size.

www.manaraa.com

The ML-VMS and the MLDR are also validated, verified and proved to be sound using a trace-driven simulation program. The simulation results show how the problem of false positives and false negatives can be reduced under given tolerable cost. They also show how the MLDR can prevent an application from crashing once a certain heap size threshold is reached

www.manaraa.com

Chapter Six
Conclusions and Future Works

This chapter provides basic conclusions as well as directions for future research.

6.0 Introduction

This dissertation provides a novel approach for dynamic memory management, a multi-layer virtual memory system (ML-VMS). The ML-VMS reorganizes the currently used dynamic memory management and dynamic memory allocation mechanisms in order to solve or overcome the problem of memory leak.

In addition to the ML-VMS, this dissertation provides two new approaches for memory leak detection and recovery. The first is memory leak detection (MLD) using aging in physical memory. This algorithm reflects both the physical and virtual behavior of memory allocation and benefits from the hardware support available for tracking physical pages in real memory. The latter is memory leak detection and recovery (MLDR) based on the ML-VMS. The MLDR uses the physical memory aging as a mechanism of leak detection and builds on the ML-VMS to provide a complete run-time solution to the memory leak problem.

6.1 Results

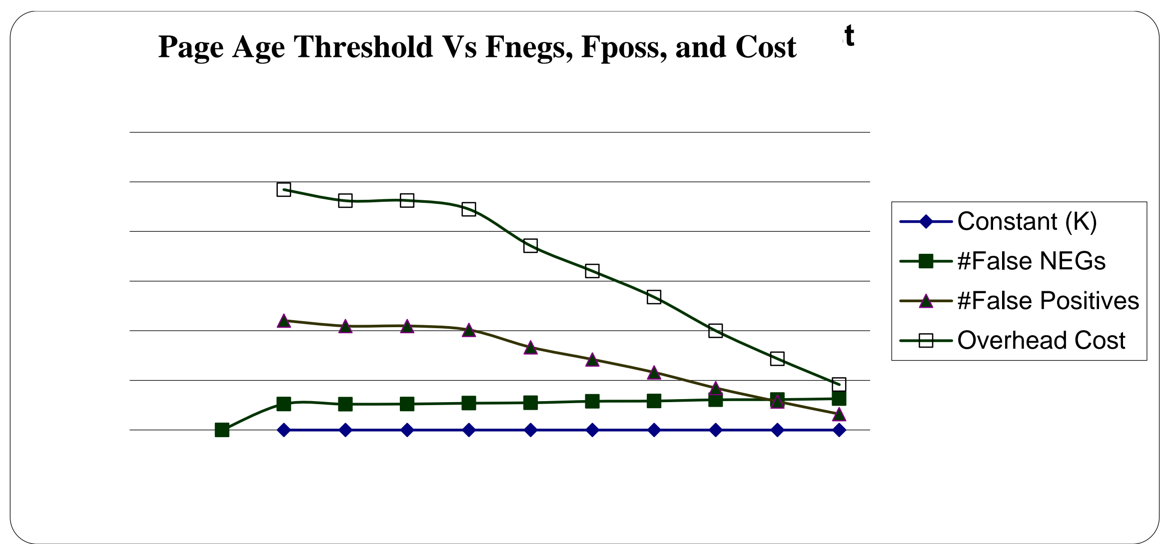
Results are discussed in terms of the factors that generally affect memory leak detection and recovery tools such as: performance, crash preventing or crash delay, false negatives, false positives, and run-time solution.

www.manaraa.com

6.1.1 Performance

The performance analysis conducted for the MLD, MLDR and the ML-VMS has shown that the MLD has the complexity of $O(n^2)$ and that the ML-VMS has significant effect on the performance of the computer system. The ML-VMS adds additional layer on the demand paging system. In addition to the cost paid by the demand paging memory system, the ML-VMS may incur the cost of accessing a disk either to backup a chunk or to recover another chunk in case of false positives as shown in equations 1 through 4 in chapter 5.

The performance cost is reduced by 1) reducing the probability of incurring false positives to the minimum. Increasing the page age threshold value reduces the false positive rate and, as a result, reduces the performance cost, 2) suggesting that the backed up objects are stored on the swap space instead of the regular disk. Swap space is usually faster than that of the file system and 3) suggesting parallel programming to enhance the performance. Figure 33 is repeated below to illustrate how increasing the page age threshold; an input parameter, can reduce the overhead cost.



المنارة للاستشارات
www.manaraa.com

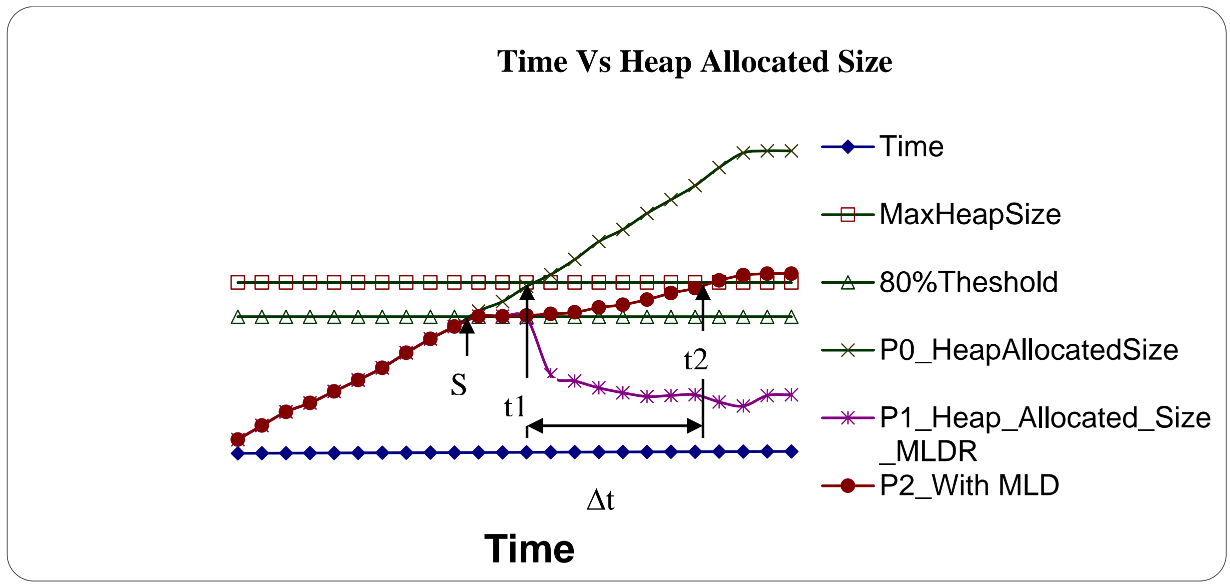
Decreasing the page age threshold in order to minimize the number of false negatives adds a cost on performance by calling the sweeper() more often. This additional sweeping increases the cost. So, there is a trade off. Decreasing false positives enhances performance; whereas decreasing false negatives adds to the cost. The cost of sweeping in the case of the MLDR is much higher than the cost of sweeping in the MLD because the MLDR sweeping process requires an additional work. The MLDR backs up removed chunks to the disk in case they might be used again. Writing to a disk is a costly operation in the virtual memory system. If the system can tolerate the cost, then use a small value for Page_Age_Threshold and minimize the number of false negatives; otherwise increase the Page_Age_Threshold as much as the system can tolerate. Some systems can tolerate overhead cost by using parallel programming and multiple processors. In this case, the sweeper() can be assigned to a set of processors having the system not worry about the overhead. A telemetry tool can provide a great help in tuning the input parameters and make the administrator visualize the effect of tuning these parameters on the overhead cost.

6.1.2 Crash Preventing

The MLD can delay a possible application crash for an application dependent period of time. In case the crash is imminent, the MLD will not prevent it. One big enhancement of the MLDR over the MLD is that the MLDR can prevent the target application from crashing if the input parameters are well-tuned. Among

www.manaraa.com

these parameters are Heap_Size_Threshold, Page_Age_Threshold, and Sweeper_Sleep_Time. The MLDR removes both of the unreachable objects and stale or useless objects, in an aging page, in order to make enough room for new allocations. The requested size for allocation is guaranteed to be always available assuming a large disk is used. We repeat figure 31 below to illustrate our conclusion on crash preventing



The figure shows the following results: a regular process (p0) crashes once the maximum heap size limit is reached. A process with the MLD (p2) starts sweeping and saves extra room for new allocations and makes the application live longer and delay a possible future crash for a delta time (Δt) period. In some scenarios, a process with the MLD will crash if it fails to satisfy allocation requests. A process with the MLDR (p1), however, provides much more room once a heap size threshold is reached and prevents crashing.

المنارة للاستشارات
www.manaraa.com

Crash preventing performed by the MLDR sweeper(), however, is not always guaranteed for several reasons. These are the same reasons that apply to the crash delay for the MLD. Among these reason are: 1) setting Heap_Size_Threshold to a relatively large value which delays the startup of the Sweeper(), 2) setting the Sweeper_Sleep_Time to a large value that makes the sweeper() not able to cope with the speed of the allocation operations being made by the target application. We have to keep in mind that allocation operations are process dependent, and 3) Setting the Page_Age_Threshold to a relatively large value which makes it more difficult for the Sweeper() to identify enough leaky pages. In fact, the Sweeper() fails to identify any single leaky page if the Page_Age_Threshold is extremely large. In case the Sweeper() fails to ensure that the required space is available on the heap to satisfy allocation requests, the target application will crash.

If our system can tolerate performance overhead cost paid by the sweeper(), the general rule of thumb is to minimize all of the already mentioned three input parameters. Minimizing Heap_Size_Threshold makes the sweeper() start early and provides enough space before it is too late. Minimizing Page_Age_Theshold makes the MLDR identify more aging pages and provide more enough room. Minimizing Sweeper_Sleep_Time makes the MLDR run the sweeper more frequently and, as a result, identify more aging pages.

www.manaraa.com

6.1.3 False Negatives

False negatives are leaky chunks that go undetected. Both of the MLD and the MLDR do not totally remove false negatives. The MLDR is similar to the MLD in terms that it can minimize the number of false negatives by decreasing the Page_Age_Threshold. Decreasing Page_Age_Threshold makes both approaches identify more leaky objects and, as a result, decrease the number of leaky chunks that will go undetected. After all, several numbers of false negatives can be tolerated since the the MLD and the MLDR will help to keep the application running.

6.1.4 False Positives

"False positives" means a detected leak is not a real leak. The object identified as a potential leak gets dereferenced after the system has given up on it! Referencing an object after it has been removed from memory, i.e., deallocated, causes incorrect results or the program to crash altogether. False positives can not be tolerated in critical mission applications.

We have seen that the MLD produces zero false positives because it implements a conservative approach that considers every a like pointer a pointer. The new structure of the ML-VMS allows the MLDR to remove all aging chunks if they are reachable or unreachable. The problem of false positives occurs when a reachable chunk that has not been used for a relatively long period of time is aged. In that case, the MLDR will remove these aged chunks to disk and falls in the false positive problem in case any of them get dereferenced. The MLDR provides a

www.manaraa.com

solution to false positives problem based on the ML-VMS using the object recovery module of the algorithm. If a false positive object is deleted from the heap and get dereferenced later by the application the deleted object is recovered.

6.1.5 Run-time solution

Current approaches for solving memory leak problem are not thorough; they either detect memory leak in development environments as performed by static analysis tools which requires the existence of source code or they garbage collect unreachable objects as performed by garbage collectors. These collectors provide partial solution only in the languages that were designed with garbage collection in mind. There is no complete run-time solution available.

The MLD uses a conservative approach to remove unreachable objects and save address space. It is similar to garbage collectors in terms of removing unreachable objects. However, it uses memory aging as a method of leak detection and it has the advantage of being suitable to applications that do not have a built-in garbage collection.

The MLDR provides a thorough run-time solution. It handles the problem of false positives, false negatives, and prevents target applications from crash due to the lack of virtual memory given a well-tuned parameters and that a target application can tolerate an additional overhead cost. The MLDR is recommended for mission critical applications that have to live for a long time and can tolerate a controllable overhead cost.

www.manaraa.com

6.2 Simulation Results

All of the MLD, the MLDR and the ML-VMS are simulated using a trace-driven simulation program that utilizes the trace generated by a benchmark that we develop for this purpose. The simulation results, as shown in chapter 5, are used to illustrate the new approaches' validity and to provide a proof of concept along with the performance analysis.

6.3 Implementation Guidelines

This dissertation provides guidelines that facilitate an implementation of the MLD, the ML-VMS, and the MLDR. It shows the necessary structures needed (VHT, MallocTable), input and out parameters, adjustments necessary to the page table, adjustments necessary to the memory allocations and deallocations functions, and suggests implementing the bookkeeping functionality in the memory page replacement algorithm such as LRU.

6.3 MLD versus MLDR

Table 14 shows a quick reference for comparison between the MLD and the MLDR provided in this dissertation. This table is provided as a quick reference. Details about comparison criteria are already provided in this dissertation.

Approach / Comparison criteria	MLD	MLDR
Produce false negatives	Yes	Yes
Can handle false positives	NO	Yes
How to deal with possible crash	Can delay crashes	Can prevent crashes

www.manaraa.com

Can handle stale objects	NO	Yes
Provide run-time solution	Yes/partial	Yes/complete
False positive recovery	Irrelevant	Yes
Overhead	Low	Relatively high

Table 14: MLD versus MLDR

6.4 Future work

This dissertation opens different areas for future research as follows:

1. Although the MLD algorithm was tested in a simulation environment, it would be better to test it on a real operating system.
2. A telemetry tool is suggested to monitor and tune the performance parameters.
3. Repeat the experiments provided in this dissertation with actual trace from real-world applications.
4. Find how MLD and MLDR can be parallelized and show the effect on performance.
5. We believe that the ML-VMS can provide additional benefits other than facilitating the solution of memory leak problem such as: solving the problem of dangling pointer and memory corruption. Another research in this area would reveal more results.

www.manaraa.com

References

1. **Abdullahi, Saleh E. and Ringwood, Graem A.**, *Garbage Collecting the Internet: A survey of Distributed Garbage Collection*, ACM, 1998
2. **Bhansali, S., Chen, W., S., Edwards, A., Murray, R., Drinic, M., Mihocka, D. and Chau, J.,** *Framework for Instruction-level Tracing and Analysis of Program Executions*, Microsoft Corporation, 2006
3. **Bush, W. R., Pincus , J. D., and Sielaff D. J.,** *A Static Analyzer for Finding Dynamic Programming Errors,* Software Practice and Experience, 2000
4. **CERT/CC**. CERT/CC Advisories. http://www.cert.org/advisories, 2007
5. **Chilimbi, T. M., and Hauswirth M,** *Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling,* ACM, 2004
6. **Choi, J.-D., Lee, K., Loginov, A., O'callahan, R., Sarkar, V., And Sridharan, M.** *Efficient and precise datarace detection for multithreaded object-oriented programs*. In Proceedings of the ACMSIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2002.
7. **Cowan, C. et al**, *StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks*. In Proceedings of the 7th USENIX Security Symposium. 63–78, 1998
8. **Denning,** P. J., *Thrashing: Its Causes and Preventions, In Proceedings of the AFIPS National Conference*, 1968

www.manaraa.com

9. **Dijkstra, E. W., Lamport, L., Martin, A.J, Scholton, C.S., and Steffens, E.F.M,** *On-the-Fly Garbage Collection: An Exercise in Cooperation,* ACM, 1978
10. **Evans**, D., *Static Detection of Dynamic Memory Errors*, ACM,1996
11. **Engler, D. And Ashcraft, K.** *RacerX: Effective, static detection of race conditions and deadlocks*. In Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003
12. **Silberscatz, A. ,Galvin, P. and Gagne, G.,** Operating System Concepts, 7th edition, John Wiley and Sons, 2005
13. **Goldschmidt, S.R. and Hennessy, J.L.** *The Accurecy of Trace-Driven Simulations of Multiprocessors*, ACM, 1993
14. **Goetz B.,** *Java Theory and Practice: A brief history of garbage collection, http://www-128.ibm.com/developerworks/java/library/j-jtp10283/,* 2003
15. **Gross K. C., Bhardwaj V., and Bickford R.,** *Proactive Detection of Software Aging Mechanisms in Performance Critical Computers*, Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE, 2002
16. ***Hallem, S., Chelf, B., Xie, Y., And Engler, D.*** *A system and language for building system specific, static analyses.* In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), 2002.

www.manaraa.com

17. **Hangal, S. and Lam, M.S.** *Tracking down software bugs using automatic anomaly detection.* In Proceedings of the 22nd International Conference on Software Engineering (ICSE) 291–301, 2002.

18. **Hastings, R. and Joyce, B.,** *Purify: Fast detection of memory leaks and access errors.* In Proceedings of the 1999 USENIX Winter Technical Conference. 125–136.

19. **Heine, D. L. and Lam, M. S. ,** *A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector,* ACM, 2003

20. **Hirzel, M. and Diwan A.,** *On the Type Accuracy of Garbage Collection*, ACM, 2000

21. **Jones, Richard,** *the Garbage Collection Bibliography***,** Computing Laboratory, 2003

22. **Kline, Robert M.,** Memory Management Basices**,** **http://www.cs.wcupa.edu/~rkline/OS/MemManage.html, 2007**

23. **Marcus E**. and **Stern**, *Blueprints for High Availability*, J Willey, New York, 2000

24. **Musuvathi, M., Park, D., Chou, A., Engler, D. R., AND Dill, D. L.** *CMC: A pragmatic approach to model checking real code***.** In Proceedings of the 5th Symposium on Operating System Design and Implementation (SOSP), 2002.

25. **National Institute of Standards and Technology, Department of Commerce,** *Software errors cost US economy $59.9 billion annually.* NIST NEWS Release 2002

www.manaraa.com

26. **Nethercote, N., and Seward J.** ***Valgrind:*** *A program supervision framework*. In Proceedings of the 3rd International Workshop on Runtime Verification (RV), 2003

27. **Ostlund, H.** A., *Memory Leak Detection with the JRocket JVM: Unique Capabilities with the right tools,* SYS-CON publications INC, 2005

28. **Parasoft Insure++,** http://www.parasoft.com/jsp/products/home.jsp?product=Insure&, 2007

29. **Parlante, Nick,** *Pointers and Memory*, **http://cslibrary.stanford.edu, 2000**

30. **Quest Software,** *JProbe Memory Debugger: Eliminate Memory Leaks and Execessive Garbage Collection, http://www.quest.com/jprobe/debugger.aspx, Quest Software, 2007*

31. **Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., AND Anderson, T.,** *Eraser***:** *A dynamic data race detector for multithreaded programs*. ACM, 1997.

32. **Shetty, R., Kharbutli, M., Solihin, Y., and Prvulovic, M.,** *HeapMon: A Low-leve, Automatic, and Programmable Memory Bug Detector,* In the proceedings of the first IBM PAC2 conference, 2004

33. **Stern, U. And Dill, D. L.** *Automatic verification of the SCI cache coherence protocol*. In Conference on Correct Hardware Design and Verification Methods (CHARME), 1995

34. **Toomula, A. and Subhlok, J.,** *Replicating Memory Behaviour for Performance Prediction,* ACM, 2004

www.manaraa.com

35. **Uhlig, R. and Mudge, T.** *Trace-driven Memory Simulation: A survey*, ACM, 1997

36. US-CERT. US-CERT Vulnerability Notes Database. http://www.kb.cert.org/vuls/, 2007

37. **Watson, Gray.** *Debug Malloc Library v5.5.0,* 2007

38. **Wikipedia** Encyclopedia, http://wikipedia.com, 2007

39. **Wilson, Paul R., Jhonstone Mark S., Neely, Michael, and Boles David,** Dynamic Storage Allocation: A Survey and Critical Review, ACM, 1995

40. **Wilson, Paul R.,** *Uniprocessor Garbage Collection Techniques,* ACM**,** 1992

41. **Xie, Y. and Aiken A.,** *Context – and Path – Sensitive Memory Leak Detection,* ACM, 2005

42. **Zhou, Y. , Zhou, P. , Qin, F. , Liu, W. , and Torrellas, J.,** *Efficient and Flexible architectural Support for Dynamic Monitoring,* ACM, 2005

43. **Zorn, B. and Grunwald, D.,** *Evaluating Models of Memory Allocation, ACM, 1994*

www.manaraa.com

Appendices

Appendix A: Input and Output Parameters of Simulation Program

In this appendix, we list the simulation input parameters in Table A.1 and the simulation output parameters in Table A.2.

Parameter	Description
SeedIntValue	The seed value for the C++ random function, rand(), used in the simulation program. With different seed values, due to the nature of randomness, we may get different outputs.
NumberOfProcesses	Number of processes used in the simulation. In order to study the behavior of processes with different characteristics, the simulation program can simulate running (n) processes at the same time.
PageFileSize(KB)	In demand paging, all processes use the same page file size.
MaxRAMSize(KB)	RAM size in KB; We can decrease size to get quick results; small sized RAM causes pages to age quickly
LocalityOfRef	Locality of reference(percentage)
ProbOfReferencingMem	Probability of referencing a memory location
GenerateDumpFile	If this flag is set the simulation generate a dump file on exit for page table(s), heap(s), malloctable(s), and RAM in order to provide full details about these structures.
GerateStatVsTime	If this flag is set the simulation records statistics vs time
initHeapSize	Initial heap size.
pageAgeThreshold	Page age threshold value
pageAgeMul(K)	A constant K explained in the dissertation.
AgingFlag	1: means invoke aging algorithm for this process 0 Not. This flag is useful to compare two processes that have same input parameters but one invokes aging alg. And the other NOT.

Table A.1: Simulation input parameters

Parameter	Description
#mallocs	The number of calling malloc()
#MemRefs	Number of times a virtual memory is referenced
#Frees	Number of times free() is called
H_MaxSize	Heap maximum size
H_FreeSize	Heap free size
H_LeakedSize	Heap leaked size
#H_Live_obj	Number of live objects (reachable)

www.manaraa.com

#False NEG	Number of false negatives; Real leaks that were NOT identified by the aging algorithm
#FalsePOS	Number of times a reference is made to an object previously identified as a potential leak.

Table A.2: Simulation output parameters

www.manaraa.com

Appendix B: Benchmark Source code mldbench. H and mldbench.c

mldBench.h

```
//maximum allocations and deallocations made
#define MAX_TRANS 600000
//seed value to srand() function
#define SEED  12755765

#define ALLOCATED 1
#define FREE      0

//propability of allocation and deallocation
//eventually all allocated chunks will be deallocated before
//the bench exit this speed makes allocation faster the deallocation
#define P_MALLOC 0.50
#define P_FREE  0.50

//probability of the next size allocation
#define P_MIN_SIZE  0.85
#define P_MEDIUM_SIZE  0.10
#define P_LARGE_SIZE  0.05

/* 1-256 BYTE*/
#define MIN_SMALL_CHUNCK_SIZE 1
#define MAX_SMALL_CHUNCK_SIZE 256
/* 256 BYTE -64k BYTE*/
#define MIN_MEDIUM_CHUNCK_SIZE 257
#define MAX_MEDIUM_CHUNCK_SIZE 4*1024
/* 64k BYTE -65000KB*/
#define MIN_LARGE_CHUNCK_SIZE 4*1024 +1
#define MAX_LARGE_CHUNCK_SIZE 10*1024
```

mldBench.c

```
#include <stdlib.h>
#include <mcheck.h>
#include <assert.h>
#include "mldBench.h"

#ifdef DMALLOC
#include "dmalloc.h"
#endif
```

www.manaraa.com

```
long  s=0;//cnt of small size chunks
long  sSize=0;//accumulated size for small objects created
long  m=0;//cnt of medium size chunks
long  mSize=0;//accumulated size for medium objects created
long  l=0;//cnt of large size chunks
long  lSize;//accumulated size for large objects created

long  allocated=0;//cnt of allocated chunks
long  allocatedSize=0;//accumulated size for allocated objects
long  hcnt=0;//cnt of location in array

long  allocatedNotFreed=0;//cnt of allocated and not freed objects
long  allocatedNotFreedSize=0;//Size of allocated and not freed objects
long  freed=0;//cnt of freed objects
long  freedSize=0;//size of freed objects

void printHeapHeader (char *allocationRef[], long allocationSize[], int
allocationStatus []);
void printHeapLine   (long index, char trans,char *allocationRef[], long
allocationSize[]);
void printAllocedNotFreed (char *allocationRef[], long allocationSize[],int
allocationStatus[]);
void freeRemaingLiveObjects(char *allocationRef[], long allocationSize[],int
allocationStatus[]);
void printTrailer();

long getTheIndexOftheXthNotFreedElement(long x, int allocationStatus[]);
long getSize();

int main () {
  mtrace();
  srand(SEED);//set seed for random function
  /*define an array of all allocations*/
  char * allocationRef[MAX_TRANS];
  /* store allocated size for each allocated chunck*/
  long   allocationSize[MAX_TRANS];
  /* allocation status 1: allocated 0: free */
  int   allocationStatus[MAX_TRANS];

  printHeapHeader(allocationRef,allocationSize,allocationStatus);

  int trans;
    for (trans=0;   trans<MAX_TRANS; trans++){
    if ((rand()%10000/10000.0) < P_MALLOC)
    {
```

www.manaraa.com

```
    //perform allocation
    //generate random size [MIN_CHUNK_SIZE, MAX_CHUNCK_SIZE]
ACCORDING TO GIVEN PROP.
    long size=getSize();
    allocationRef[hcnt]= (char *) malloc(size);
    assert(allocationRef!=NULL);

    allocationSize[hcnt]=size;
    allocationStatus[hcnt]=ALLOCATED; //allocated
    allocated++; allocatedNotFreed++;
    allocatedSize+=size; allocatedNotFreedSize+=size;//accumulate allocated
size
    //printf("A:%d: ",hcnt);
    printHeapLine(hcnt,'+',allocationRef,allocationSize);
    hcnt++;//next allocation cnt
    } else {
    /*perform deallocation*/
    //getRandom object to free
    long x,index=-1;
    if (allocatedNotFreed>0) {
       x =rand()%allocatedNotFreed +1; //select random object x to free
       //iterate to find x
       index=getTheIndexOftheXthNotFreedElement(x,allocationStatus);
         }//if
    if (index>-1) {
      //free object
      allocatedNotFreed--;
      allocatedNotFreedSize-=allocationSize[index];
        freed++;
        freedSize+=allocationSize[index];
        //printf("F:%d: ",index);
        printHeapLine(index,'-',allocationRef,allocationSize);
        allocationStatus[index]=FREE;//mark as freed
        free(allocationRef[index]);//remove object from heap
        allocationRef[index]=NULL;//null the pointer so as not to remain dangling
    }else {
      // printf("No available object to free...\n");//index=-1

        }
    }//if
  }//for
  freeRemaingLiveObjects(allocationRef, allocationSize, allocationStatus);
 printTrailer();
 //printAllocedNotFreed (allocationRef,allocationSize,allocationStatus);
 return 0;
}//main
```

www.manaraa.com

```
void printTrailer() {
  //list allocations

  printf ("#Small  Objects: %d Size: %d \n",s,sSize);
  printf ("#Meduim Objects: %d Size: %d \n",m,mSize);
  printf ("#Large  Objects: %d Size: %d \n",l,lSize);
  printf ("Alloc'd Objects: %d Size: %d Byte \n",allocated,allocatedSize);
  printf ("Freed   objects: %d size: %d Byte \n\n",freed,freedSize);
  printf ("Stale:Alloc'd NotFreed: %d size: %d
Byte\n",allocatedNotFreed,allocatedNotFreedSize);
  }//printTrailer
long getTheIndexOftheXthNotFreedElement(long x, int allocationStatus[] ){
  long index =-1;
  long i=0;
  long cnt=0;
  for (i=0;i<hcnt;i++) {
    if (allocationStatus[i]!=0) {//this object is allocated and Not Freed
      cnt++;
      if (cnt==x){
        //this is the intended object to be freed
           index=i;
           break;
      }//if
    }//if
  }//for
  return index;
}
void  freeRemaingLiveObjects(char *allocationRef[], long allocationSize[],int
allocationStatus[]){
  long i=0;
  for (i=0;i<hcnt;i++) {
    if (allocationStatus[i]!=0) {//this object is still allocated and Not Freed
           //free it
           allocatedNotFreed--;
        allocatedNotFreedSize-=allocationSize[i];
           freed++;
           freedSize+=allocationSize[i];
           //printf("F:%d: ",index);
           printHeapLine(i,'-',allocationRef,allocationSize);
           allocationStatus[i]=FREE;//mark as freed
           free(allocationRef[i]);//remove object from heap
           allocationRef[i]=NULL;//null the pointer so as not to remain dangling
      }//if
  }//for
```

www.manaraa.com

```
void printHeapHeader (char * allocationRef[], long allocationSize[], int
allocationStatus[]){
        printf("MAX TRANS: %d\n",MAX_TRANS);
        int i;
        printf("location\tra\tsize\n");

}
void printAllocedNotFreed (char *allocationRef[], long allocationSize[],int
allocationStatus[]){
      long i=0;
        for (i=0;   i<hcnt; i++){
              if (allocationStatus[i]!=0) {//allocated and not freed
                  printf("%p\t%d\n", allocationRef[i],allocationSize[i]);
                }
        }//for
}
void printHeapLine (long index,char trans, char * allocationRef[], long
allocationSize[]){
                printf("%c\t%p\t%d\n",trans,
allocationRef[index],allocationSize[index]);
}
long getSize() {
    long size=1;//default value
    float propNextSize =rand()%100000/100000.00;
    if (propNextSize <P_MIN_SIZE ) {
         //small size chunck
       size=rand()%(MAX_SMALL_CHUNCK_SIZE + 1 -
MIN_SMALL_CHUNCK_SIZE) + MIN_SMALL_CHUNCK_SIZE;
         s++; sSize+=size;
         }else if ((propNextSize-P_MIN_SIZE)<P_MEDIUM_SIZE){
         //MEDIUM size chunk
          size=rand()%(MAX_MEDIUM_CHUNCK_SIZE + 1 -
MIN_MEDIUM_CHUNCK_SIZE) + MIN_MEDIUM_CHUNCK_SIZE;
          m++; mSize+=size;
         }else {
         //large chunks are created
         size=rand()%(MAX_LARGE_CHUNCK_SIZE + 1 -
MIN_LARGE_CHUNCK_SIZE) + MIN_LARGE_CHUNCK_SIZE;
         l++; lSize+=size;
         }
    return size;
```

www.manaraa.com

APPENDIX C: SOURCE CODE FOR SIMULATION PROGRAM

Heap.h and Heap.cpp

```
#include <fstream.h>

#ifndef HEAP_H
#define HEAP_H

struct memChunk;
typedef memChunk * chunkPtrType;

class Heap {

        public: Heap();//constructor
                ~Heap();//destructor
                long getHeapMaxSize() {return HeapMaxSize;}
      long getHeapAllocatedSize() {return heapAllocatedSize;}
                long getHeapAllocatedObjects () { return heapAllocatedObjects;}
                long getHeapLeakedSize() { return heapLeakedSize;}
                long getHeapLeakedObjects() {return heapLeakedObjects;}
                long myMalloc(long ra,long inputSize,bool leaky);
                // allocate inputSize and return memRef ( represent a pointer to a
chunk)
                bool myFree(long ra);//Free a chunk pointed to by memRef

                long getMemRefToFree();//returns an allocated and NOT leaky
chunk;
                long getMemRefToAcess(float locality, long prevRef);//returns an
allocated and NOT leaky chunk;
                bool markUnAccessable(long ra);

                void dumpHeap(long processID, ofstream &outputFile);
        private:
                void coalesce (chunkPtrType cur);//merge adjacent free chunks
                long HeapMaxSize; // max size of the Heap
                long heapAllocatedSize; // the allocated size from the Heap;
                long heapAllocatedObjects;
                long heapLeakedSize;
                long heapLeakedObjects;
                double getRandomProp();
                chunkPtrType Head;

};//end class
#endif
```

www.manaraa.com

Heap.cpp

```
#include "Heap.h"
#include "List.h"

#include <iostream.h>
#include <iomanip.h>
#include <assert.h>
#include <stdlib.h>

const int PRECISION=10000;//Four digits precision //used in getRandomProb()

struct memChunk {
        chunkPtrType previous;
        long ra;
        long memRef;
        long chunkSize;
        bool chunkIsAllocated;// true: allocated ; false: free
        bool leaky;//true leaky chunk; false NOT leaky
        chunkPtrType next;
};

List *list=new List();

Heap:: Heap()
{                       //Heap initialization
                        HeapMaxSize =0;
                        heapAllocatedSize=0;
                        heapAllocatedObjects=0;
                        heapLeakedSize=0;
                    heapLeakedObjects=0;
         Head=NULL;
Heap:: ~Heap() {}//Should be implemented to free the heap and return it to
memory
// otherwise a leak will occur
long Heap::myMalloc (long ra,long inputSize, bool leakyflag) {
         long memr=-1;
         heapAllocatedSize += inputSize;
         heapAllocatedObjects++;
         heapLeakedObjects++;
         heapLeakedSize += inputSize;//allocation not freed is assumed leak
   //the first allocation in an empty Heap
   if (Head==NULL) {
                              Head = new memChunk; // first allocation
                              assert (Head!=NULL);
                              Head->ra=ra;
```

www.manaraa.com

```
                              Head->memRef =0; // 0 is the  first memRef in the
Heap
                              Head->chunkSize = inputSize;//intially all heap is free
                              HeapMaxSize+=inputSize;
                              Head->chunkIsAllocated = true;//not allocated (free);
                              Head->leaky=leakyflag;
                              Head->next = NULL;
                              Head->previous = NULL;
                              return 0;//allocated in memory reference 0
        }

        // first fit allocation strategy
                chunkPtrType temp,prev, cur=Head;
                while (cur!=NULL) {
                        if ((cur->chunkSize > inputSize) && (!cur->chunkIsAllocated))
{
                              //this is a free chunk with first fit
                              //split into two 1) Active (inputSize)// 2) Free
(chunkSize-inputSize)
                              temp = new memChunk; //temp point to remaining
free chunck
                              assert (temp!=NULL);
                              temp->memRef = cur->memRef +inputSize;
                              temp->chunkSize = cur->chunkSize -inputSize;
                              temp->ra=0;
                              temp->chunkIsAllocated =false; //remaining free
chunk
                              temp->leaky =false;
                              temp->next = cur->next;
            temp->previous =cur;

                              cur->chunkSize =inputSize;
                              cur->ra=ra;
                              cur->chunkIsAllocated =true;//allocated
                              cur->leaky =leakyflag;
                              if (cur->next !=NULL) { cur->next->previous=temp; }
```

www.manaraa.com

```
                    cur->next = temp;

                    return cur->memRef;//return a pointer to newly
allocatd chunck
                }
                if ((cur->chunkSize == inputSize) && (!cur-
>chunkIsAllocated)) {
                    cur->ra=ra;
                    cur->chunkIsAllocated =true;//allocated
                    cur->leaky = leakyflag;
                    return cur->memRef;//return a pointer to this chunk
                }
            prev=cur;
            cur=cur->next;
            }//

            if (cur==NULL) {
                //Heap is Full; extend the Heap from the OS
                    temp = new memChunk;
                    //temp point to a chunk to be added to end of the
Heap
                    assert (temp!=NULL);
                    temp->memRef = HeapMaxSize;
                    temp->chunkSize = inputSize;
                    temp->ra =ra;
                    temp->chunkIsAllocated =true;
                    temp->leaky =leakyflag;
                    temp->next = NULL;//Chunk at end of the heap;
                    HeapMaxSize+=inputSize;//update the max size to
reflect the new value
                    prev->next  =temp;
                    temp->previous = prev;
                    return temp->memRef;

            }
        return memr;//should not reach this statement
```

www.manaraa.com

```
}

bool Heap::myFree(long ra){
        //True means freeing an allocated chunk
        //False means either freeing already free chunk or chunk is not available;

    chunkPtrType prev, cur=Head;
         while (cur!=NULL) {
                 if (cur->ra ==ra) {
                         if (cur->chunkIsAllocated) {
                                 //freeing allocatd chunk
                                 cur->chunkIsAllocated=false;
                                 cur->leaky=false; //free chunks are not leak
                                 cur->ra=0;//will remove rturn address
                                 heapLeakedObjects--;
                                 heapLeakedSize -= cur->chunkSize;
                                 coalesce(cur);//merge adjacent free chunks
                                 return true;//success;
                         }else {
```

www.manaraa.com

```
                                //Freeing already free chunck
                                 return false;//error;
                          }


                  }
           prev=cur;
           cur=cur->next;
           }//while
          return false;//ra is NOT available;
}

bool Heap::markUnAccessable(long ra){
          //set leaky of a chunk to true. this chunk will not be accessed
      chunkPtrType cur=Head;
           while (cur!=NULL) {
                  if (cur->ra ==ra) {
                    //passed ra must be to already allocated chunk
          cur->leaky=true;
                    return true;
                  }
           cur=cur->next;
           }//while
          return false;//ra is NOT available;
long Heap::getMemRefToFree() {
          //returns a reference to an allocated NOT leaky chunk;
           chunkPtrType cur=Head;
           long r,x=0;
           while (cur!=NULL) {//x counts the no of not leaky and allocated
                                                    // can be accessed
                    if ((!cur->leaky) && (cur->chunkIsAllocated))  {
                            // allocated and NOT leaky can be choosed
                            ++x;
                    }//if
           cur=cur->next;
           }//while

           if (x>0) {
             r= rand () % x +1;
             // selecet random object allocated and Not Leaked

             cur=Head;

             long i=0;
             while (cur!=NULL) {
```

www.manaraa.com

```
                    if ((!cur->leaky) && (cur->chunkIsAllocated))
        // allocated and NOT leaky can be choosed
                                ++i;
                                if (i==r) { //choosed object
                                        //return ref to any part of object
                                        return cur->memRef + rand()%cur-
>chunkSize;
                                }//if
                        }//if
                  cur=cur->next;
                }//while
         }//if x>0 there is not leaky and allocated chunk can be returnd

        return -1;//no MemRef available to be freed

}//getMemRefToFree

long Heap::getMemRefToAcess(float locality,long prevRef) {
        double r;
        r=getRandomProp();
        if ((r<=locality) && (prevRef !=-1)) {
                //look up the next closest, to prevRef, allocated chunk Not Leaky
and return it
                chunkPtrType cur=Head;
                while ((cur!=NULL)&&(cur->memRef <=prevRef)) {//move to
location of prevRef
                        cur= cur->next;
                }//while
                while (cur!=NULL){
                  if ((!cur->leaky) && (cur->chunkIsAllocated))  {
                        // allocated and NOT leaky can be choosed
                        //return a memRef to any part of the object
                        return cur->memRef + rand()%cur->chunkSize;
                  }//if
                  cur=cur->next;
                }//while
        }//if

        //The next line is reached if r > locality
        //return random memRef from Heap
   //The next block gurantee to return a random memRef to allocated not leaky
chunk

  return getMemRefToFree();
```

www.manaraa.com

```
}//getMemRefToAccess

void Heap::coalesce (chunkPtrType cur) {
  chunkPtrType pr, nx;
  pr=NULL;//assume initially no previous
  nx=NULL;//assume initially no next

  if ((cur->previous  != NULL) && (!cur->previous->chunkIsAllocated)) {
              pr = cur->previous;// there is a previous free chunk
        }
  if ((cur->next != NULL) && (!cur->next->chunkIsAllocated)) {
              nx = cur->next;// there is a next free chunk
        }

  if ((pr == NULL ) && (nx ==NULL)) return;//This is a stand alone free chunk

  if  ((pr !=NULL ) && (nx!=NULL)){//Preceeded by free chunk and followed by
free chunk
           pr->chunkSize += cur->chunkSize + nx->chunkSize;
              pr->next=nx->next;
              if (nx->next !=NULL) { nx->next->previous =pr;}
              nx->next = NULL;
              nx->previous=NULL;
              delete nx;
              nx=NULL;

              }
  else if ((pr !=NULL ) && (nx==NULL)) {//preceeded by Free chunk
                   nx=cur->next;
                   pr->chunkSize += cur->chunkSize;
                   pr->next=nx;
                   if (nx!=NULL) { cur->next->previous =pr;}

  }else if ((pr ==NULL ) && (nx!=NULL )){//Followed by free chunk
                        pr=cur;
                        cur=pr->next;
                        nx=cur->next;
                        pr->chunkSize += cur->chunkSize;
                        pr->next=nx;
                        if (nx!=NULL) { cur->next->previous =pr;}
              }
```

www.manaraa.com

```
        //delete node pointed to by cur
        cur->next=NULL;
        cur->previous=NULL;
        delete cur;
        cur=NULL;
        return;

}//merge adjacent free chunks

void Heap::dumpHeap (long processID, ofstream &dumpFile) {
        dumpFile <<"\nHeap DUMP for processID: " <<processID<<endl;
        dumpFile <<"Heap Max Size(Byte): " <<getHeapMaxSize() <<endl;
        dumpFile <<"Heap Allocated size(Byte): " <<getHeapAllocatedSize()
<<endl;
        dumpFile <<"# Heap Allocated Objects: "<< getHeapAllocatedObjects ()
<<endl;
        dumpFile <<"Leaked Size(Byte): " <<getHeapLeakedSize() <<endl;
        dumpFile <<"#Leaked Objects: " << getHeapLeakedObjects()<<endl;

        dumpFile<<setw(12)<<"MemRef"<<setw(12)<<"ra"<<setw(12)<<"Size"<<
setw(8)<<"Status"
                <<setw(12)<< "Lk_Sts"
                <<setw(12)<<"Prev"<< setw(12)<<"Next"<<endl;
        int i=0;
        for (chunkPtrType cur=Head; cur!=NULL; cur =cur->next) {
                dumpFile<<setw(12)<<cur->memRef
                        <<setw(12)<<cur->ra
                        <<setw(12)<<cur->chunkSize <<setw(8);
                        if (cur->chunkIsAllocated ) {
                                dumpFile <<"Active";
                        }else { dumpFile <<"Free"; }
                dumpFile <<setw(12);
                if (cur->leaky) {
                        dumpFile<<"lky";
                }else { dumpFile <<"NotLky";}
       dumpFile <<setw(12);
;
```

www.manaraa.com

```
			if (cur->previous ==NULL ) {
				dumpFile<<"NULL";
			}else { dumpFile <<cur->previous->memRef;}
			 dumpFile <<setw(12);
			if (cur->next  ==NULL ) {
				dumpFile<<"NULL\n";
			}else { dumpFile <<cur->next->memRef <<endl;}
		i++;
		//if (i%5==0){break;}//donot perform complete dump
		}//for
		dumpFile <<"-------------------------------------------------------------------";
double Heap::getRandomProp() {
   double r;
		r= rand() % PRECISION;
		r/=PRECISION;
		return r;
}
```

PTable.h

```
#include <fstream.h>
#include <assert.h>

#ifndef PTABLE_H
#define PTABLE_H

class PTABLE {

		public: PTABLE(int);//constructor
			~PTABLE();//destructor
			long getNoOfPages () { return noOfPages; }
			long getPageSize  () { return pageSize;}
			long getProcessSize(){ return processSize;}
			long getPtableTimeStamp(long pRef) { return
myPtableTimeStamp[pRef];}
			bool incrementSize (int chunkSize);
			bool pageRefInRAM (long pageRef);
			bool isDirty      (long pRef) {return myPtableDB[pRef];}
			bool isPotLeak    (long pRef) {return myPtablePL[pRef];}
			void setDiryBit   (long pRef,bool status) {myPtableDB[pRef]=status;}
			void dumpPTABLE(long processID, ofstream &);
			void setPtableFrameRef (long pRef,long frameIndex) {
myPtableFrameRef[pRef]=frameIndex;}
			void setPtablePB(long pRef, bool status)
{myPtablePB[pRef]=status;}
```

www.manaraa.com

```
                void setPtablePL(long pRef, bool status)
{myPtablePL[pRef]=status;}
                void setPtableTimeStamp(long pRef, long timeStamp)
{myPtableTimeStamp[pRef]=timeStamp;}

                void setPtableLRUtimeStamp(long pRef,long
timeStamp){myPtableLRUtimeStamp[pRef]=timeStamp;}
                long getPtableLRUtimeStamp(long pRef){return
myPtableLRUtimeStamp[pRef];}

                void setPtableOnSwapSpc(long pRef, bool status)
{myPtableOnSwapSpc[pRef]=status;}
```

www.manaraa.com

```
        private:

                long noOfPages;
                int  pageSize;
                long processSize;
                long * myPtableFrameRef;
                bool * myPtablePB;
                bool * myPtableOnSwapSpc; // 1 page on swap space; 0 NOT
                bool * myPtableDB;
                bool * myPtablePL;//potential Leak flag
                long * myPtableTimeStamp;
                long * myPtableLRUtimeStamp;
                bool firstVisit;//first visit to increment pagetable
 };//end class
#endif
```

PTable.cpp

```
#include "PTABLE.h"
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
#include <stdlib.h>

PTABLE::PTABLE(int pSize) :pageSize(pSize)
{
        noOfPages=0;//no pages in current page table yet
        processSize =0;//initialy the size of the heap of process is zero
        cout <<"pagetab ins: "<<noOfPages<<endl;
        myPtableFrameRef = NULL;
        myPtablePB       = NULL;
        myPtableOnSwapSpc= NULL;
        myPtableDB            = NULL;
        myPtablePL            = NULL;
        myPtableTimeStamp= NULL;
        myPtableLRUtimeStamp=NULL;
   firstVisit=true;//mark to indicate first visit to increment function
PTABLE:: ~PTABLE() {}
bool PTABLE::pageRefInRAM (long pageRef){
        return (myPtablePB[pageRef]);
bool PTABLE::incrementSize (int chunkSize){
        if (firstVisit) {//create first entry
                firstVisit=false;
                myPtableFrameRef = new long [noOfPages+1];
                assert(myPtableFrameRef !=NULL);
                myPtablePB       = new bool [noOfPages+1];
                assert(myPtablePB       !=NULL);
```

www.manaraa.com

```
		myPtableOnSwapSpc= new bool [noOfPages+1];
		assert(myPtableOnSwapSpc!=NULL);
		myPtableDB	= new bool [noOfPages+1];
		assert(myPtableDB	!=NULL);
		myPtablePL	= new bool [noOfPages+1];
		assert(myPtablePL	!=NULL);
		myPtableTimeStamp= new long [noOfPages+1];
		assert(myPtableTimeStamp!=NULL);
		myPtableLRUtimeStamp=new long[noOfPages+1];
		assert(myPtableLRUtimeStamp!=NULL);

		myPtableFrameRef [noOfPages]= 0;
		myPtablePB	[noOfPages]= false;
		myPtableOnSwapSpc[noOfPages]= false;
		myPtableDB	[noOfPages]= false;//initially not dirty
		myPtablePL	[noOfPages]= false;
		myPtableTimeStamp[noOfPages]= 0;
	myPtableLRUtimeStamp[noOfPages]=0;

		noOfPages++;//first page is added
	}//if

	//increment processSize
	processSize += chunkSize;

	long newSizeInPages = (long)
ceil(static_cast<double>((double)processSize/(pageSize*1024)));
	if (newSizeInPages == noOfPages) {//no need to exted ptable. Space is
already available
		return true;
	}//if

	//extend page table by (newSizeInPages - noOfPages ) pages
	//save pointers

	long * mPFR   = myPtableFrameRef;
	bool * mPPB   = myPtablePB;
	bool * mOSwS  = myPtableOnSwapSpc;
	bool * mDB    = myPtableDB;
	bool * mPL    = myPtablePL;
	long * mTimeStamp=myPtableTimeStamp;
	long * mPtableLRUtimeStamp= myPtableLRUtimeStamp;
```

www.manaraa.com

```
//allocate the required new size
	myPtableFrameRef = new long [newSizeInPages];
	assert(myPtableFrameRef !=NULL);
	myPtablePB	= new bool [newSizeInPages];
	assert(myPtablePB	!=NULL);
	myPtableOnSwapSpc= new bool [newSizeInPages];
	assert(myPtableOnSwapSpc!=NULL);
	myPtableDB	= new bool [newSizeInPages];
	assert(myPtableDB	!=NULL);
	myPtablePL	= new bool [newSizeInPages];
	assert(myPtablePL	!=NULL);
	myPtableTimeStamp= new long [newSizeInPages];
	assert(myPtableTimeStamp!=NULL);
	myPtableLRUtimeStamp=new long[newSizeInPages];
	assert(myPtableLRUtimeStamp!=NULL);

	for (long i=0; i<noOfPages; i++ ) {//copy old to new
		myPtableFrameRef [i]= mPFR[i];
		myPtablePB	[i]= mPPB[i];
		myPtableOnSwapSpc[i]= mOSwS[i];
		myPtableDB	[i]= mDB[i];
		myPtablePL	[i]= mPL[i];
		myPtableTimeStamp[i]= mTimeStamp[i];
		myPtableLRUtimeStamp[i]= mPtableLRUtimeStamp[i];
	}
	//initialize extended pages
	for (long x=noOfPages; x< newSizeInPages; x++) {
		myPtableFrameRef [x]= 0;
		myPtablePB	[x]= false;
		myPtableOnSwapSpc[x]= false;
		myPtableDB	[x]= false;//extended area not dirty yet
		myPtablePL	[x]= false;
		myPtableTimeStamp[x]= 0;
		myPtableLRUtimeStamp[x]=0;
	}

	//delete old copy to save space in Vir. Add. Space
	delete [] mPFR;
	delete [] mPPB;
	delete [] mOSwS;
	delete [] mDB;
	delete [] mPL;
	delete [] mTimeStamp;
```

www.manaraa.com

```
	// set # of pages to new value
	noOfPages = newSizeInPages;
	return true;
}void PTABLE::dumpPTABLE (long processID, ofstream & dumpFile) {
	dumpFile <<endl <<"....Page Table DUMP.... for process: " <<processID
<<endl;
	dumpFile <<setw(20) <<"# of pages   : " <<setw(15)<<noOfPages <<endl;
	dumpFile <<setw(20) <<"Page Size(KB): " <<setw(15)<<pageSize
<<endl;
	dumpFile <<setw(20) <<"Process Size(Byte): "
<<setw(15)<<processSize<<endl;

	dumpFile <<setw(15) <<"Page Index"
		<<setw(15)<<"FrameRef"
		<<setw(15)<<"Status"
		<<setw(10)<<"OnSwapSpc"
		<<setw(10)<<"DirtyBit"
		<<setw(10)<<"PotLeak"
		<<setw(10)<<"TimeStamp"
		<<setw(10)<<"LRUtime"
		<<endl;

	for (int i=0;i<noOfPages;i++) {
		dumpFile <<setw(15)
		<<i
		<<setw(15)<<myPtableFrameRef[i]
		<<setw(15)<<myPtablePB[i]
		<<setw(10)<<myPtableOnSwapSpc[i]
		<<setw(10)<<((myPtableDB[i]==1)?"W":"R")
		<<setw(10)<<myPtablePL[i]
		<<setw(10)<<myPtableTimeStamp[i]
		<<setw(10)<<myPtableLRUtimeStamp[i]
		<<endl;
		//if (i%5==0){break;}//donot perform complete dump
	}//for
}
```

MallocTable.h

```
#include <fstream.h>

#ifndef MALLOCTABLE_H
#define MALLOCTABLE_H

struct memChunk;
typedef memChunk * chunkPtrType;
```

www.manaraa.com

```
class MallocTable {

          public: MallocTable();//constructor
                    ~MallocTable();//destructor
                    bool removeFromMallocTable(long ra);//remove a  chunk from
MallocTable
                    bool addToMallocTable(long ra,long inputMemRef, long
inputSize,bool leakyflag);
                    void dumpMallocTable(long processID, ofstream &outputFile);
                    long countLeaky(long &size);
                    long countUnLeaky(long &size);
                    void getUnreachableMemRefs(long page, long pageSize, long &
noOfRefs,long unReachable[]);
                    bool markUnReachable(long ra);
          private:

                    chunkPtrType Head, Tail;
       long objectsCreated, sizeObjectsCreated;
                    long objectsRemoved, sizeObjectsRemoved;

};//end class
#endif
```

MallocTable.cpp

```
#include "MallocTable.h"

#include <iostream.h>
#include <iomanip.h>
#include <assert.h>
#include <stdlib.h>

const int PRECISION=10000;//Four digits precision //used in getRandomProb()

struct memChunk {
          chunkPtrType previous;
          long ra;
          long memRef;
          long chunkSize;
          bool leaky;//true leaky chunk; false NOT leaky
          chunkPtrType next;
};

MallocTable:: MallocTable() {
          Head=Tail=NULL;
          objectsCreated    = 0;
          sizeObjectsCreated= 0;
```

www.manaraa.com

```
        objectsRemoved    = 0;
        sizeObjectsRemoved= 0;
}
MallocTable::~MallocTable() { }

void MallocTable::dumpMallocTable (long processID, ofstream &dumpFile) {
        long size1=0, size2=0;
        dumpFile <<"\nMallocTable DUMP for processID: " <<processID<<endl;
        dumpFile <<"Leak objects  : " <<countLeaky(size1) << " Unleaky     : "
<<countUnLeaky(size2) <<endl;
        dumpFile <<"Leak objs Size: " <<size1           << " Unleaky size: "
<<size2             <<endl;
        dumpFile <<"objectsCreated: " <<objectsCreated << "
sizeObjectsCreated: " <<sizeObjectsCreated <<endl;
        dumpFile <<"objectsRemoved: " <<objectsRemoved << "
sizeObjectsRemoved: " <<sizeObjectsRemoved <<endl;

        dumpFile<<setw(12)<<"MemRef"
                <<setw(12)<<"ra"
                <<setw(12)<<"Size"
                <<setw(12)<< "Lk_Sts"
                <<setw(12)<<"Prev"<< setw(12)<<"Next"<<endl;
        int i=0;
        for (chunkPtrType cur=Head; cur!=NULL; cur =cur->next) {
                dumpFile<<setw(12)<<cur->memRef
                        <<setw(12)<<cur->ra
                        <<setw(12)<<cur->chunkSize;
                dumpFile <<setw(12);
                if (cur->leaky) {
                        dumpFile<<"lky";
                }else { dumpFile <<"NotLky";}

                dumpFile <<setw(12);
                if (cur->previous ==NULL ) {
                        dumpFile<<"NULL";
                }else { dumpFile <<cur->previous->memRef;}
                 dumpFile <<setw(12);
                if (cur->next  ==NULL ) {
                        dumpFile<<"NULL\n";
                }else { dumpFile <<cur->next->memRef <<endl;}
                i++;
                if (i%5==0){break;}//donot perform complete dump
        }//for
        dumpFile <<"-----------------------------------------------------------------";
```

www.manaraa.com

```
long MallocTable::countLeaky(long & size) {
	long cnt=0;
	chunkPtrType  cur=Head;
		while (cur!=NULL) {

		if (cur->leaky ) {cnt++; size += cur->chunkSize;}
		cur = cur->next;


		}//while
	return cnt;
bool MallocTable::markUnReachable(long ra){
	//mark a chunk as un reachable it can be freed by MLD
	//pass an ra to available chunck
	chunkPtrType  cur=Head;
		while (cur!=NULL) {

			if ( cur->ra == ra ) {
			  cur->leaky=true;
			  return true;
			}
		cur = cur->next;
		}//while
	return false;//must not be reached
void MallocTable::getUnreachableMemRefs(long page, long pageSize, long &
noOfRefs,long unReachable[]) {
	long lowRef = page * pageSize * 1024; // low memref to look at
	long highRef= lowRef + pageSize *1024 -1; // high ref
   noOfRefs=0;

	chunkPtrType  cur=Head;
	while (cur!=NULL) {
		if (cur->memRef < lowRef) { cur= cur->next; continue;}
		if (cur->memRef > highRef){ break;}//finish searching
		// the following memory refrences are in page
		if (cur->leaky ) {
			//this chunk is unreachable and in aged page
			unReachable[noOfRefs]= cur->ra;
			noOfRefs++;
		}
	cur = cur->next;
	}//while
long MallocTable::countUnLeaky(long & size) {
	long cnt=0;
	chunkPtrType  cur=Head;
```

www.manaraa.com

```
            while (cur!=NULL) {

            if (!(cur->leaky )) {cnt++; size +=cur->chunkSize;}

            cur = cur->next;

            }//while
        return cnt;
}
bool MallocTable::addToMallocTable(long ra,long inputMemRef, long inputSize,
bool leakyflag) {
         objectsCreated++; sizeObjectsCreated +=inputSize;

        chunkPtrType temp,prev, cur;

        cur=Head;
        prev=NULL;

        if (Head == NULL) {
                    //first item in mallocTable
                    temp = new memChunk;
                    assert(temp!=NULL);
                    temp->previous = NULL;
                    temp->ra       = ra;
                    temp->memRef   = inputMemRef;
                    temp->chunkSize= inputSize;
                    temp->leaky    = leakyflag;
                    temp->next     = NULL;
                    Head           = temp;
                    Tail           = temp;
                    return true;

        }
        while (cur!=NULL) {
            //find a place to insert chunk sorted
            if (cur->memRef >= inputMemRef) {break;}

                    prev= cur;
                    cur = cur->next;


        }//while
        if (cur==Head) {
              // add as the first item to mallocTable
                    temp = new memChunk;
                    assert (temp!=NULL);
```

www.manaraa.com

```
                        temp->previous = NULL;
                        temp->next     = cur;
                        temp->memRef   = inputMemRef;
                        temp->ra       = ra;
                        temp->chunkSize= inputSize;
                        temp->leaky    = leakyflag;
                        cur->previous = temp;
                        Head=temp;
                        return true;
                }
        if (cur ==NULL) {
                // add as the last  item to mallocTable
                        temp = new memChunk;
                        assert (temp!=NULL);
                        temp->previous = prev;
                        temp->next     = NULL;
                        temp->ra       = ra;
                        temp->memRef   = inputMemRef;
                        temp->chunkSize= inputSize;
                        temp->leaky    = leakyflag;
                        prev->next     = temp;
                        Tail=temp;
                        return true;
                }
        // add between prev and cur
                        temp = new memChunk;
                        assert (temp!=NULL);
                        temp->previous = prev;
                        temp->next     = cur;
                        prev->next     = temp;
                        cur ->previous = temp;
                        temp->memRef   = inputMemRef;
                        temp->ra       = ra;
                        temp->chunkSize= inputSize;
                        temp->leaky    = leakyflag;
                        return true;
}

bool MallocTable::removeFromMallocTable(long ra){
    objectsRemoved++;
    chunkPtrType prev, cur=Head;
        while (cur!=NULL) {
                if (cur->ra ==ra) {
                        if (Head == Tail){
                                //removing the only available element
```

www.manaraa.com

```
			Head->next = NULL;
			Head->previous=NULL;
			sizeObjectsRemoved +=Head->chunkSize;
			delete Head;
			Head=NULL;
			Tail=NULL;
			return true;
		}//
		if (cur==Head) {
			//removing first element
			Head = Head->next;
			Head->previous = NULL;
cur->next = NULL;
			cur->previous=NULL;
			sizeObjectsRemoved +=cur->chunkSize;
			delete cur;
			cur=NULL;
			return true;

		} else if (cur==Tail) {
			//removing last element
			sizeObjectsRemoved +=Tail->chunkSize;
			Tail = Tail->previous;
			Tail->next = NULL;
cur->next = NULL;
			cur->previous=NULL;
			delete cur;
			cur=NULL;
			return true;
		} else {
			// removing item in the middle
			sizeObjectsRemoved +=cur->chunkSize;
			prev->next = cur->next;
			cur->next->previous  = prev;
			cur->previous=NULL;
			cur->next    =NULL;
			delete cur;
			cur=NULL;
			return true;

		}
```


المنارة للاستشارات
www.manaraa.com

```
            }//if
        prev=cur;
        cur=cur->next;
        }//while
      return false;//MemRef is NOT available;
}
```

RAM.h

```
#include <fstream.h>

#ifndef RAM_H
#define RAM_H

class RAM {

      public: RAM(long, int);//constructor
            ~RAM();//destructor
            long getMaxSize () { return RAMMaxSize;}
            long getNoOfFrames() {return noOfFrames;}
            long getPageSize()   {return pageSize;}
            long getRAMProcess(long frameIndex) {return
myRAMProcesses[frameIndex];}
            bool getRAMStatus (long frameIndex) {return
myRAMStatus[frameIndex];}
            long getRAMPageRefs(long frameIndex){return
myRAMPageRefs[frameIndex];}
            bool existsFreeFrame (long & frameIndex);
            void setRAMPageRefs (long frameIndex, long pRef) {
myRAMPageRefs[frameIndex]=pRef;}
            void setRAMProcesses(long frameIndex, long proc) {
myRAMProcesses[frameIndex]=proc;}
            void setRAMStatus   (long frameIndex, bool status){
myRAMStatus[frameIndex]=status;}
     void dumpRAM(ofstream &);
      private:

            long RAMMaxSize; // in KB, max size of the Heap
            long noOfFrames;
            int  pageSize;//in KB
     long * myRAMPageRefs;
            long * myRAMProcesses;
            bool * myRAMStatus;
};//end class
#endif
```

RAM.cpp

```
#include "RAM.h"
```

www.manaraa.com

```
#include <iostream.h>
#include <iomanip.h>

const int RAM_ATT=3;//#columns in the Ram


RAM::RAM(long maxSize, int pSize) :pageSize(pSize)
{		RAMMaxSize = maxSize;
		noOfFrames = RAMMaxSize/pageSize;
	myRAMPageRefs = new long [noOfFrames];
		myRAMProcesses= new long [noOfFrames];
		myRAMStatus   = new bool [noOfFrames];


		//initialize RAM
		for (int i=0;i<noOfFrames;i++) {
			myRAMPageRefs [i]=0;
			myRAMProcesses[i]=0;
			myRAMStatus   [i]=false;
		}
}

RAM:: ~RAM() {}//Should be implemented to free the RAM and return it to
memory
				// otherwise a leak will occur
bool RAM::existsFreeFrame (long & frameIndex) {
	bool res=false;
	for (int i=0;i<noOfFrames;i++)
		if	(!myRAMStatus[i]) {
			// frame is free
			res=true;
			frameIndex=i;//returns the index of free frame
			break;
		}
	return res;
}

void RAM::dumpRAM (ofstream & dumpFile) {
   dumpFile <<endl <<"....RAM DUMP...."<<endl;
	dumpFile <<setw(20) <<"Max Size (KB): " <<setw(15)<<RAMMaxSize
<<endl;
	dumpFile <<setw(20) <<"# of Frames  : " <<setw(15)<<noOfFrames
<<endl;
	dumpFile <<setw(20) <<"Page Size(KB): " <<setw(15)<<pageSize
<<endl;
```

www.manaraa.com

```
	dumpFile <<setw(15) <<"Frame Index" <<setw(15) <<"PageRef"
<<setw(15)<<"Process"<<setw(15)<<"Status"<<endl;

	for (int i=0;i<noOfFrames;i++) {
		dumpFile <<setw(15)
		<<i
		<<setw(15)
		<<myRAMPageRefs [i]
		<<setw(15)
		<<myRAMProcesses[i]
		<<setw(15)
		<<myRAMStatus   [i]
		<<endl;
		if (i%5==0){break;}//donot perform complete dump
	}//for
}//dumpRAM
```

mainSimProg.cpp

```
#include "Heap.h"
#include "RAM.h"
#include "PTABLE.h"
#include "MallocTable.h"

#include <iostream.h>

#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>
#include <time.h>
#include <math.h>

const int MAXPROCESSES=20; //Maximum number of processes
const int PR_ATT=12; //Process attributes
const int PRECISION=10000;//Four digits precision //used in getRandomProb()

long simTime=0;//Counter of simulation steps; represents time
int pageSize;//page size in KB
int nOfProcesses;//number of processes
long maxRAMSize;//Maximum RAM Size
int nOfFrames;// number of frames =maxRAMSize/pageSize
long maxSimSteps; //maximum number of memory refrences at which simulation
unsigned int seed; //seed value for random function
					// will stop this counter is esentially represents time
float localityOfRef;//locality of reference. How much likely a process is going to
						// to reference the same page next.
```

www.manaraa.com

```
float propOfReferencingMem;
int   generateDumpFileFlag;// 1 generate dump file 0: DO NOT generate a file

long  memRefrenceToAccess;
long  TimePassBeforeWriteToStatVsTimeFile;
long  max_Heap_Size;

char trans; //+ allocate - deallocate
long ra;    //return address

enum {HEAPTHRESHOLD, NO_MALLOCS, NO_REFS,
NO_MARKED_UNACCESSABLE, LAST_ACCESSED_REF,
        NO_HITS, NO_FAULTS, FALSEPOSITIVES, PAGEOUTS, AGECNT,
AGEAVG,
        USE_AGING_FLAG, PAGEAGEMUL, OVERHEAD};

double processes[MAXPROCESSES][PR_ATT];
//Array of Processes
//        index HEAPTHRESHOLD HEAPTHRESHOLD, if max heap size
>HEAPTHRESHOLD page is aged
//   INDEX NO_MALLOCS    How many times malloc is called
//   index NO_REFS       how many times memory refrences are made
//   index NO_MARKED_UNACCESSABLE      how many chunks marked
unAccessable
                                                            //will not be
selected to be accessed
//   index LAST_ACCESSED_REF  Last reference accessed by a process
//   index NO_HITS      number of page hits; page found in RAM When requested
//   index NO_FAULTS    number of page faults
//   index FALSEPOSITIVES  // number of false positives for a process
//   index PAGEOUTS       // number of page outs;//if DB is set we need to page-
out the page
//   index AGECNT         // number of pages of Pot. Leak included in computing
the average
//        index AGEAVG                // the average age for pages marked with
Pot Leak flag
//   index USE_AGING_FLAG  // if set the process will use aging alg. Else NOT
//   index PAGEAGEMUL      // PAGE AGE MULTIPLIER
//   index OVERHEAD       //Overhead associated with calling Sweeper()
Heap *Heaps[MAXPROCESSES];//Heaps[0] the heap for process zero
                                       //Heaps[1] the heap for process one
and so on.
PTABLE* pageTable[MAXPROCESSES];// pageTable[0] the page table for
process 0
                                                       //  pageTable[1] the
```

www.manaraa.com

```
 page table for process 1 and so on
MallocTable* mallocTable[MAXPROCESSES];//mallocTable[0] mallocTable for process 0
                                                                        //mallocTable[1] mallocTable for process 1
RAM  *ram;


void initialize();
void readInputFile(ofstream&);
void recordStatVsTime(int p,long SimTime,ofstream &StatVsTime);
void recordStatVsTimeHeader(ofstream &StatVsTime);
void dumpProcessesStat();

double getRandomProp();
void createDumpFile(int);
void performMemAccess( int );
long getMemRef(int p, float localityOfRef, long prevRef);
long getRandomVictimPage(long &victimPageProcess,long &frame);
void removeUnReachableObjectsForAgingPages(int pCnt,long simTime);

long getLRUvictimPage(long &,long &);

void setDirtyBit(int pCnt,long pageRef);
void swapPages( long victimPageProcess,long victimPage,
                    long frame, int pCnt,long pageRef, long simTime) ;

long AgedPages=0;

int main () {
        initialize();
        srand(seed);
   long memAllocRef;
        bool proceed=true;
        int  chunkSize;
        double r;

        ofstream StatVsTime ("StatVsTime.dat",ios::out);//Store Stat Vs Time
        if (!StatVsTime){
                cerr << "File StatVsTime.dat could not be created";
                exit(1);
        }
        ifstream traceFile ("tracefilesequal/trace10000Pm50_50_85_10_5",ios::in);
        if (!traceFile) {cerr<<"Can not open trace file\n";exit(3);}
```

www.manaraa.com

```
          recordStatVsTimeHeader(StatVsTime);
          //simulation loops until there are no more trace data

          while(proceed) {
                    simTime++;
                    for (int p=0; p<nOfProcesses; p++) {
                              r = getRandomProp();
                              if (r<= propOfReferencingMem ) {
                                        performMemAccess(p);
                              } else {//call malloc function
                                        proceed=false;
                                        if (traceFile>>trans>>ra>>chunkSize) {
                                                  proceed=true;
                                        }
               if (proceed==false) {break;}
                                        if (trans=='+') {//perform allocation
                                          processes[p][NO_MALLOCS]++;
                                          //created chuncks assumed not leaky until they
reach the place of free
                                          memAllocRef= Heaps[p]->myMalloc
(ra,chunkSize,false);
                                          //record this allocation in mallocTable ; on
malloctable also they are not leaky
                                          mallocTable[p]-
>addToMallocTable(ra,memAllocRef,chunkSize,false);
                                          //increase the size of page table by chunkSize if
needed
                                          pageTable[p]->incrementSize (chunkSize);
                                        } else {// trans='-' call free() function
                                          //here set leaky to true on Heap and
malloctable.actual freeing is left
                                          //for the aging algorithm
                                          processes[p][NO_MARKED_UNACCESSABLE]++;
                                        //  Heaps[p]->myFree (ra);//
                                          bool y= Heaps[p]->markUnAccessable(ra);
                                          //The object freed from heap is removed from
MallocTable
                                          // mallocTable[p]->removeFromMallocTable(ra);
                                          y=mallocTable[p]->markUnReachable(ra);

                                        }//if

                              }//if r<propMemRef
                    //here, we may record statistics from heap statistics VS simTime
```

www.manaraa.com

```
                    if (simTime%TimePassBeforeWriteToStatVsTimeFile  ==0) {
                      recordStatVsTime(p, simTime,StatVsTime);
                       cout <<simTime <<endl;
                            }//if
                    }//for
                   // if (simTime==10) {break;}
         }//while
   cout <<"\nAged Pages: "<< AgedPages<<endl;
         dumpProcessesStat();

         createDumpFile(generateDumpFileFlag);

return 0;//program ends normally
}//main

void performMemAccess (int p) {

         long memRefToAccess, pRef,victimPageProcess,victimPage,frame;
         processes[p][NO_REFS]++;
         //generate a reference to a Live memory location to be accessed
         processes[p][LAST_ACCESSED_REF]= memRefToAccess=
                    getMemRef(p, localityOfRef,(long)
processes[p][LAST_ACCESSED_REF]);
         //Address resolution. Convert memRef into virtual page index (pRef)
         if (memRefToAccess>0) {
                   pRef = (long)
ceil(static_cast<double>((double)memRefToAccess/(pageSize*1024)))-1;}
if (pRef >0) {//A valid page refernce
         setDirtyBit(p,pRef);//update pagetable to reflect the randomly
                                                   //generated dirtyFlag.
                                                   //we need to indicate whether this is a
Read or Write access

         if (pageTable[p]->pageRefInRAM (pRef)) {
                   //page is in RAM use it // a page hit occured
                   processes[p][NO_HITS]++;
                   //time stamp page being accessed for LRU algorithm
                   pageTable[p]->setPtableLRUtimeStamp(pRef,simTime);
         } else {
                   //page fault occured
                   processes[p][NO_FAULTS]++;
                   //add outgoing page to swap space of proces if it is not already
there //call by ref
                   //rese time stamp..................
                   long frameIndex;
                   if (ram->existsFreeFrame (frameIndex)) {
```

www.manaraa.com

```
                        //use it
                        //cout <<"empty frame: "<<frameIndex <<endl;
                        ram->setRAMPageRefs(frameIndex,pRef);
                        ram->setRAMProcesses(frameIndex,p);
                        ram->setRAMStatus(frameIndex,true);//occupied by pRef of
process p
                        //place frameIndex in page table
                        pageTable[p]->setPtableFrameRef(pRef,frameIndex);
                        //Page is now available in page table*/
                        pageTable[p]->setPtablePB(pRef,true);
              //time stamp page being accessed for LRU algorithm
                   pageTable[p]->setPtableLRUtimeStamp(pRef,simTime);
                }else {
                           //Choose a victim page using LRU selection algorithm
                                //Global replacement strategy is used
                        //victimPage
=getRandomVictimPage(victimPageProcess,frame);
                                victimPage
=getLRUvictimPage(victimPageProcess,frame);
            //cout <<"vicPage: "<<victimPage<<" Frame: "<<frame <<endl;
                                if (pageTable[victimPageProcess]-
>isDirty(victimPage)) {//dirty page
                                        //increment page outs per process

        ++processes[victimPageProcess][PAGEOUTS];
                                        //write to disk if dirty
                                }//if

        swapPages(victimPageProcess,victimPage,frame,p,pRef,simTime);

   if ((processes[p][USE_AGING_FLAG]==1) &&
                (Heaps[p]->getHeapMaxSize() >
(processes[p][HEAPTHRESHOLD]*max_Heap_Size))) {
                //remove unreachable objects
                removeUnReachableObjectsForAgingPages(p,simTime);
        }//if
}//if (pRef>0)

}//performMemAccess
 void removeUnReachableObjectsForAgingPages(int pCnt,long simTime) {
        // this function identifies leaky objects  and
        // remove them from the Heap and MallocTable
        const long Max =4096;//the max possible references in a page of 4Kb
        long noOfRefs=0;
        long unReachable[Max];
        long pages;double pageAge;
```

www.manaraa.com

```
        pages =pageTable[pCnt]->getNoOfPages();
        for (int page=0; page<pages; page++ ){
                if (pageTable[pCnt]->isPotLeak (page)){
                                pageAge =simTime - pageTable[pCnt]-
>getPtableTimeStamp(page);
                                //cout <<"simTime:"<<simTime<<" pTs:"
<<pageTable[pCnt]->getPtableTimeStamp(page)<<" pAge:"<< pageAge<<endl;
                                if ((pageAge >(processes[pCnt][PAGEAGEMUL] *
processes[pCnt][AGEAVG])) &&
                                        (processes[pCnt][AGEAVG]>0) )
                                {
                                   ++AgedPages;
                                        //find chunks in this page and mark them as
potential garbage
               processes[pCnt][OVERHEAD]++;// inc
               pageTable[pCnt]->setPtablePB  (page,false);
                                        pageTable[pCnt]->setPtablePL  (page,false);
                                        pageTable[pCnt]->setDiryBit(page,false);
                                   pageTable[pCnt]-
>setPtableOnSwapSpc(page,false);
                                        //UNREACHABLE OBJECTS MUST BE
FREED FROM
                                        //1)HEAP AND 2)MALLOCTABLE
                                        //fllowing lines are the Sweeper()
                                        noOfRefs=0;
                                        mallocTable[pCnt]-
>getUnreachableMemRefs(page,pageSize,noOfRefs,unReachable);
                                        for (int i=0;i<noOfRefs;i++) {
                                                //remove from malloctable
                                                mallocTable[pCnt]-
>removeFromMallocTable(unReachable[i]);
                                                //remove from Heap
                                                Heaps[pCnt]->myFree(unReachable[i]);
                                        }//for
                                }//if
                        }//if
        }//for int page

}//removeUnReachableObjectsForAgingPages

void swapPages( long victimPageProcess,long victimPage,
                           long frame, int pCnt,long pageRef, long simTime) {
        //clear presence bit from old page//
        pageTable[victimPageProcess]->setPtablePB(victimPage,false);
```

www.manaraa.com

```
        //mark swap-out page as potential leak
        pageTable[victimPageProcess]->setPtablePL(victimPage,true);
        pageTable[victimPageProcess]-
>setPtableTimeStamp(victimPage,simTime);

        //load to ram
        ram->setRAMPageRefs (frame,pageRef);//store page in RAM
        ram->setRAMProcesses(frame,pCnt); //store to which process
        ram->setRAMStatus   (frame,true);//mark page in RAM as occupied
        //update page table to point to new page
        pageTable[pCnt]->setPtableFrameRef (pageRef,frame);//load new
pageRef to PT
        pageTable[pCnt]->setPtablePB(pageRef,true);//Page is now present
        //time stamp page being accessed for LRU algorithm
        pageTable[pCnt]->setPtableLRUtimeStamp(pageRef,simTime);

        if (pageTable[pCnt]->isPotLeak (pageRef)) {
                //include this pageRef in the accumulated Page_Age_Threshold

        processes[pCnt][AGEAVG]=((processes[pCnt][AGECNT]*processes[pCnt]
[AGEAVG])
                                        + (simTime-pageTable[pCnt]-
>getPtableTimeStamp (pageRef)))

/++processes[pCnt][AGECNT];

        }
        //clear potential leak flag if page is swapped-in
        //since we are accessing a pRef; clear PL bit and time stamp
        pageTable[pCnt]->setPtablePL(pageRef,false);
        pageTable[pCnt]->setPtableTimeStamp(pageRef,0);//if PL is false no
meaning of timeStamp;stm. can be removed

}//swapPages

long getMemRef(int p, float localityOfRef, long prevRef) {
         //This function gets a memory reference to be acccessed

  return Heaps[p]->getMemRefToAcess(localityOfRef,(long)
processes[p][LAST_ACCESSED_REF]);

}//getMemRef
```

www.manaraa.com

```
long getLRUvictimPage(long &victimPageProcess,long &frame) {
	//Global replacement strategy is used
	//select the page with minimum LRU time
	//The implementation is not fast/ uses sequential search
	//but it will NOT affect the result of our simulation
	long min = 99999999; //initialize to a large number
	long victim=0;
	victimPageProcess=ram->getRAMProcess(0);//choose page 0 process 0
as default

	for (int i=0;i< ram->getNoOfFrames();i++) {
		if (ram->getRAMStatus(i)) {// this is an occupied frame
			if ( pageTable[ram->getRAMProcess(i)]-
>getPtableLRUtimeStamp (ram->getRAMPageRefs(i))<min) {
				min= pageTable[ram->getRAMProcess(i)]-
>getPtableLRUtimeStamp  (ram->getRAMPageRefs(i));
				victim=ram->getRAMPageRefs(i);
				victimPageProcess=ram->getRAMProcess(i);
				frame=i;
			}
		}//if
	}//

	return victim;

}//getLRUvictimPage
long getRandomVictimPage(long &victimPageProcess,long &frame) {

	// a frame is selected randomly from the RAM// This implementation is fast
for our simulation
	// we could use another replacement strategy like LRU,or LFU
	long victim=0;
	long r;
	r = rand() % ram->getNoOfFrames();// r is a frame index of a choosed
victim frame

	victim=ram->getRAMPageRefs(r);
	victimPageProcess=ram->getRAMProcess(r);
	frame=r;

	return victim;

double getRandomProp() {
   double r;
	r= rand() % PRECISION;
	r/=PRECISION;
```

www.manaraa.com

```
        return r;
}void recordStatVsTimeHeader(ofstream &StatVsTime) {
        StatVsTime<<endl;
        StatVsTime
                <<setw(10)<<"Process"
                <<setw(10)<<"Time"
                <<setw(15)<<"HEAPTHRESHOLD"
                <<setw(15)<<"PAGEAGEMUL"
                <<setw(15)<<"UseAgAlg?"

                <<setw(15)<<"#Mallocs"
           <<setw(15)<<"#Refs"

                <<setw(15)<<"#MrkUNacc"

                <<setw(15)<<"H_MaxSize"

                <<setw(15)<<"#H_leakedObjs"
                <<setw(15)<<"#page Hits"
                <<setw(15)<<"#page Faults"
                <<setw(15)<<"#FalsePos"
                <<setw(15)<<"#PageOuts"
                <<setw(15)<<"#PgsINcInAVG"
                <<setw(15)<<"Page AVG"
                <<setw(15)<<"OverHead"
           <<endl;
}void recordStatVsTime(int p, long time,ofstream &StatVsTime){

                StatVsTime
                <<setw(10)<<p
                <<setw(10)<<time
                <<setw(15)<<processes[p][HEAPTHRESHOLD]
                <<setw(15)<<processes[p][PAGEAGEMUL]
                <<setw(15)<<processes[p][USE_AGING_FLAG]
                <<setw(15)<<processes[p][NO_MALLOCS]
                <<setw(15)<<processes[p][NO_REFS]
                <<setw(15)<<processes[p][NO_MARKED_UNACCESSABLE]
                <<setw(15)<<Heaps[p]->getHeapMaxSize () //maxSize

                <<setw(15)<<Heaps[p]->getHeapLeakedObjects ()

                <<setw(15)<<processes[p][NO_HITS]
                <<setw(15)<<processes[p][NO_FAULTS]
                <<setw(15)<<processes[p][FALSEPOSITIVES]
                <<setw(15)<<processes[p][PAGEOUTS]
```

www.manaraa.com

```
            <<setw(15)<<processes[p][AGECNT]
            <<setw(15)<<processes[p][AGEAVG]
            <<setw(15)<<processes[p][OVERHEAD]

            <<endl;
void initialize() {
        cerr <<"Simulation may take time depending on the input
parameters.\nPlease wait....\n";

        ofstream outputFile ("OutputFile.dat",ios::out);//Store Page refrences to a
file
        if (!outputFile){
                cerr << "File outputFile.dat could not be opened";
                exit(1);
        }

        readInputFile(outputFile);//Reads Simulation parameters
        srand(seed);//seed random function with time in milliseconds
void readInputFile(ofstream &outputFile) {
        ifstream inputFile ("inputFile.dat",ios::in);
        if (!inputFile){
                cerr << "File inputFile.dat could not be opened";
                exit(1);
        }
  char filler0[25], filler [25],filler1[25],filler3[25],filler4[55],filler5[55],filler7[30];
  inputFile >>filler>>seed;
  outputFile
<<setprecision(2)<<setiosflags(ios::left|ios::fixed|ios::showpoint)<<setw(20)<<"Se
ed val="<<setw(20)<<seed<<endl;
  inputFile >>filler>>pageSize;
  outputFile <<setw(20)<<filler<<setw(20)<<pageSize<<endl;
  inputFile>>filler>>nOfProcesses;
  outputFile <<setw(20)<<filler<<setw(20)<<nOfProcesses<<endl;
  inputFile>>filler>>maxRAMSize;
  outputFile <<setw(20)<<filler<<setw(20)<<maxRAMSize<<endl;
  nOfFrames =maxRAMSize/pageSize;
   outputFile<<setw(20)<<"No of Frames="<<setw(20)<<nOfFrames<<endl;

  inputFile>>filler>>localityOfRef;
  outputFile <<setw(20)<<filler<<setw(20)<<localityOfRef<<endl;

  inputFile>>filler5>>propOfReferencingMem;
```

www.manaraa.com

```
  propOfReferencingMem /=100;
  outputFile <<setw(30)<<filler5<<setw(12)<<propOfReferencingMem<<endl;


  inputFile>>filler7>>generateDumpFileFlag;
  outputFile <<setw(20)<<filler7<<setw(20)<<generateDumpFileFlag<<endl;
  inputFile>>filler5>>TimePassBeforeWriteToStatVsTimeFile;

outputFile<<setw(36)<<filler5<<setw(7)<<TimePassBeforeWriteToStatVsTimeFil
e<<endl;
  inputFile>>filler5>>max_Heap_Size;
  outputFile<<setw(36)<<filler5<<setw(7)<<max_Heap_Size <<endl;

  //convert from MB to byte
  max_Heap_Size = max_Heap_Size *1024 * 1024;
  inputFile>>filler0>>filler1>>filler3>>filler4;


  long x1;
  while (inputFile>>x1){
          inputFile>>processes[x1][HEAPTHRESHOLD]
                  >>processes[x1][PAGEAGEMUL]
                  >>processes[x1][USE_AGING_FLAG];

          //intialize a heap for eah process.
          Heaps[x1]= new Heap();
          //initialize a page table for each Process
          pageTable[x1] = new PTABLE (pageSize);
     mallocTable[x1]= new MallocTable();

          //intialize attributes
          processes[x1][NO_MALLOCS]=0;
          processes[x1][NO_REFS]=0;
          processes[x1][NO_MARKED_UNACCESSABLE]=0;
          processes[x1][LAST_ACCESSED_REF]=0;
          processes[x1][NO_HITS]=0;
          processes[x1][NO_FAULTS]=0;
          processes[x1][FALSEPOSITIVES]=0;
          processes[x1][PAGEOUTS]=0;
          processes[x1][AGECNT]=0;
          processes[x1][AGEAVG]=0;
          processes[x1][OVERHEAD]=0;
  //intialize RAM;
  ram = new RAM(maxRAMSize,pageSize);
```

www.manaraa.com

```
}//ReadInputFile
void dumpProcessesStat(){
	 ofstream outputFile ("OutputFile.dat",ios::app);//Store Page refrences to a
file
	if (!outputFile){
		cerr << "File OutputFile.dat could not be created";
		exit(1);
	}
	outputFile<<endl;
	outputFile
		<<setw(10)<<"Process"
		<<setw(15)<<"HEAPTHRESHOLD"
		<<setw(15)<<"PAGEAGEMUL"
		<<setw(15)<<"UseAgAlg?"
		<<setw(15)<<"#Mallocs"
		<<setw(15)<<"#MemRefs"
		<<setw(15)<<"#MrkUnacc"
		<<setw(15)<<"H_MaxSize"

		<<setw(15)<<"#H_Live_obj"
		<<setw(15)<<"#page Hits"
		<<setw(15)<<"#page Faults"
		<<setw(15)<<"#FalsePos"
		<<setw(15)<<"#PageOuts"
		<<setw(15)<<"#PgsINcInAVG"
		<<setw(15)<<"Page AVG"
		<<setw(15)<<"OverHead"
	   <<endl;

	for (int i=0 ; i<nOfProcesses;i++)  {
		outputFile
		<< setw(10)<<i
		<<setw(15)<<processes[i][HEAPTHRESHOLD]
		<<setw(15)<<processes[i][PAGEAGEMUL]
		<<setw(15)<<processes[i][USE_AGING_FLAG]
		<<setw(15)<<processes[i][NO_MALLOCS]
		<<setw(15)<<processes[i][NO_REFS]
		<<setw(15)<<processes[i][NO_MARKED_UNACCESSABLE]
		<<setw(15)<<Heaps[i]->getHeapMaxSize () //maxSize

		<<setw(15)<<Heaps[i]->getHeapLeakedObjects ()

		<<setw(15)<<processes[i][NO_HITS]
		<<setw(15)<<processes[i][NO_FAULTS]
		<<setw(15)<<processes[i][FALSEPOSITIVES]
		<<setw(15)<<processes[i][PAGEOUTS]
```

www.manaraa.com

```
                <<setw(15)<<processes[i][AGECNT]
                <<setw(15)<<processes[i][AGEAVG]
                <<setw(15)<<processes[i][OVERHEAD]
                <<endl;
        }
        outputFile<<endl;
}

void createDumpFile(int flag) {
        //clear old file
        ofstream dumpFile ("dumpFile.dat",ios::out);//dump page tables, heaps,
and ram to file
        if (!dumpFile){
                cerr << "File dumpFile.dat could not be created";
                exit(1);
        }
        if (flag==1) {//
                for (int x=0;x<nOfProcesses;x++) {

                        //dumpFile <<"\n...."<<endl;
                        pageTable[x]->dumpPTABLE (x,dumpFile);
                        //dumpFile <<"\n...."<<endl;
                        Heaps[x]->dumpHeap (x,dumpFile);
                        mallocTable[x]->dumpMallocTable(x,dumpFile);
                        //dumpFile <<"\n...."<<endl;
                }
                ram->dumpRAM(dumpFile);

void setDirtyBit(int pCnt,long pageRef){
        //propability of read =50% of write 50%
        double r = getRandomProp();

        if (r <= 0.5) {
                pageTable[pCnt]->setDiryBit(pageRef,false);//Read
        }else{
                pageTable[pCnt]->setDiryBit(pageRef,true);//Write
```

List.h

```
#include<iostream.h>
#ifndef LIST_H
#define LIST_H

struct  listChunk;
typedef listChunk * listPtrType;
```

www.manaraa.com

```
class List {

public: List(){ listHead=NULL;};//constructor
		~List();//destructor
		bool isEmpty() {return listHead==NULL;}
	void add(long ref);
		long getItem();//get the first item and remove from list
		void dumpList();
	private:
		listPtrType listHead;

};//end class
#endif
```

List.cpp

```
#include "List.h"
#include <stdlib.h>
#include <iostream.h>

struct  listChunk {
	long memRef;
	listPtrType next;
};

void List::dumpList () {
	listPtrType cur =listHead;
	while (cur!=NULL) {
		cout <<cur->memRef <<endl;
		cur=cur->next;
	}
}//dumpList

void List::add (long ref) {
	if (listHead==NULL) {
		//first item to add
		listHead= new listChunk;
		listHead->memRef =ref;
		listHead->next=NULL;
	}else {
		listPtrType temp;
		temp=new listChunk;
		temp->memRef =ref;
		temp->next =listHead;
		listHead=temp;
	}
```

www.manaraa.com

```
long List::getItem(){
        long first=-1;
        listPtrType cur=listHead;

        if (!isEmpty()){
                first = cur->memRef;
                listHead=listHead->next;
                delete cur;
                cur=NULL;
        }
   return first;
```

www.manaraa.com