Technical Report

UCAM-CL-TR-242
ISSN 1476-2986

Number 242

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

An object oriented approach to virtual memory management

Glenford Ezra Mapp

January 1992

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 1991 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Summary

Advances in computer technology are being pooled together to form a new computing environment which is characterised by powerful workstations with vast amounts of memory connected to high speed networks. This environment will provide a large number of diverse services such as multimedia communications, expert systems and object-oriented databases. In order to develop these complex applications in an efficient manner, new interfaces are required which are simple, fast and flexible and which allow the programmer to use an object-oriented approach throughout the design and implementation of an application. Virtual memory techniques are increasingly being used to build these new facilities.

In addition, since CPU speeds continue to increase faster than disk speeds, an I/O bottleneck may develop in which the CPU may be idle for long periods waiting for paging requests to be satisfied. To overcome this problem, it is necessary to develop new paging algorithms that better reflect how different objects are used. Thus a facility to page objects on a per-object basis is required and a testbed is also needed to obtain experimental data on the paging activity of different objects.

Virtual memory techniques, previously only used in mainframe and minicomputer architectures, are being employed in the memory management units of modern microprocessors. With very large address spaces becoming a standard feature of most systems, the use of memory mapping is seen as an effective way of providing greater flexibility as well as improved system efficiency.

This thesis presents an object-oriented interface for memory-mapped objects. Each object has a designated object type. Handles are associated with different object types and the interface allows users to define and manage new object types. Moving data between the object and its backing store is done by user-level processes called object managers. Object managers interact with the kernel via a specified interface thus allowing users to build their own object managers. A framework to compare different algorithms was also developed and an experimental testbed was designed to gather and analyse data on the paging activity of various programs. Using the testbed, conventional paging algorithms were applied to different types of objects and the results were compared. New paging algorithms were designed and implemented for objects that are accessed in a highly sequential manner.

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of any work done in collaboration.

Furthermore, this dissertation is not substantially the same as any that I have submitted for a degree, diploma or any other qualification at any other university.

No part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or any other qualification.

To Salome, my wonderful wife

Acknowledgments

I wish to express my deep appreciation to my supervisor, Dr. Ian Leslie, for his encouragement, guidance and support during the course of my research. I would also like to thank members of the Wanda Group at the Computer Laboratory, in particular, Joe Dixon, Sai Lai Lo, Tim Wilson, Cormac Sreenan and Paul Jardetzky for many useful discussions.

I am also indebted to the following people who have read and commented upon this dissertation: Ian Leslie, Sue Thompson, Matthew Doar, David Greaves, Andy Harter and Leslie French.

Special thanks to Salome, my wife, for her help in the typing and presentation of this work and more importantly giving her love and support to this effort.

Many thanks to Dr. Malcolm Wilkinson from Anamartic Limited for allowing me to include information on one of their products in this dissertation. Thanks must also go to Martyn Johnson and Chris Hadley, system administrators at the Lab, for all their practical assistance.

Thanks to my graduate colleagues for their encouragement and support. In this regard, special thanks to Peter Newman, Bhaskar Harita and David Greaves with whom I shared some memorable times as a member of the "Back-Room Boys" in the Old Music School.

During my research I received financial support, for which I am most grateful, from the Cambridge Commonwealth Trust, my wife, Salome, Clare Hall, the Computer Laboratory and the Committee of Vice Chancellors.

Finally, my deepest appreciation and praise goes to the LORD whose unfailing love was demonstrated many times during this research. In expression of this, and in the spirit of this effort, I quote the following:

> A good man may have many troubles,
> but the Lord delivers him from them all.
>
> Psalm 34:19

Contents

List of Figures

List of Tables

Glossary of Terms

The number of the page on which the term is introduced appears in parenthesis.

ATC Address Translation Cache (14)

ATM Asynchronous Transfer Mode (2)

BCPL Basic Combined Programming Language (35)

BSPA BufStream Paging Algorithm (133)

CPL Current Privilege Level (12)

CPU Central Processing Ubit (3)

DEC Digital Equipment Corporation (23)

DOS Disk Operating System (31)

DPL Descriptor Privilege Level (13)

DST Descriptor Segment Table (22)

DWS Dampled Working Set (54)

GDT Global Descriptor Table (11)

GE General Electric (21)

GLRU Global Least Recently Used (52)

IBM International Business Machines (4)

INRIA Institut National de Recherche en Informatique et en Automatique (42)

I/O Input/Output (3)

ip Interpagefault Time (101)

IPC Interprocess Communication (36)

IRIM Inter-Reference Interval Model (62)

IRM Independent Reference Model (56)

KST Known Segment Table (22)

LAN Local Area Network (1)

LDT	Local Descriptor Table (11)
LRU	Least Recently Used (14)
LRUSM	LRU Stack Model (56)
MIT	Massachusetts Institute of Technology (21)
MLRU	Modified LRU (104)
MMU	Memory Managemeny Unit (2)
OS/2	Operating System/2 (31)
PFF	Page Fault Frequency (54)
PMMU	Paged Memory Management Unit (13)
PTE	Page Table Entry (6)
QoS	Quality-of-Service (79)
RAM	Random Access Memory (1)
RISC	Reduced Instruction Set Computer (1)
ROM	Read-only Memory (16)
RPC	Remote Procedure Call (43)
rr	Reclaim ratio (100)
SCSI	Small Computer System Interface (51)
SG	Shared Globally (14)
SINMAN	Systems Internal Names Manager (23)
SSPA	SimpleStream Paging Algorithm (133)
SWS	Sampled Working Set (53)
TLB	Translation Lookaside Buffer (6)
VLSI	Very Large Scale Integrated (1)
VSWS	Variable-Interval Working Set (54)
WAN	Wide Area Network (2)
WS	Working Set (53)

WSI Wafer Scale Integration (51)

WSPA WriteStream Paging Algorithm (133)

Trademarks

Unix is a registered trademark of AT&T.
SunOs is a registered trademark of Sun MicroSystems.
Chorus is a registered trademark of Chorus Systems.

Chapter 1

Introduction

At present, computer technology is evolving at a rapid rate. The processing power of modern microprocessors is greater than the mainframes of the Sixties, and with the emergence of RISC technology, processors are likely to continue getting faster. In addition, continuing improvements in VLSI techniques will ensure that the cost of producing these units continues to fall. This phenomenon has already made possible the widespread use of workstations and personal computers.

With cheaper processors becoming widely available, multiprocessor architectures are being built using different interconnection networks. These configurations attempt to exploit parallel computing to achieve improved performance which could be sustained as the number of processors is increased. These systems are providing a serious challenge to mainframe architectures in the area of large-scale scientific computing.

Another dramatic development has been the declining cost of **random access memory** or RAM. New techniques will soon make four Megabyte (MB) RAM chips a commonplace item. Hence, in the near future, most machines will have large amounts (i.e. 64–128 MB) of main memory. Non-volatile memory, though more expensive than RAM, also continues to fall in cost and provides an effective alternative to disk storage in certain computer environments.

Computer networks are also increasing in speed. The standard 10 Megabit-per-second (Mbps) Ethernet local area network or LAN of the Eighties will soon be superseded by 100 Mbps Token Ring systems. In addition, with the appearance of fast packet switching,

wide area networks or WANs are being built to run at speeds of 500 Mbps to one Gigabit per second (Gbps) using Asynchronous Transfer Mode (ATM) techniques. These networks will carry new services, e.g. voice and video, and provide a large number of high bandwidth channels.

Based on these developments, the new computing environment will comprise a multitude of powerful machines, having large amounts of memory connected to fast networks. The software requirements for this new environment are radically different from the computing models of the last three decades. This is especially true in the area of **operating systems**. For example, Unix, one of the most popular operating systems for the last two decades, was developed for the time-sharing environment of the early Seventies. At that time networking was minimal, large amounts of primary memory was a rare occurrence, and disk storage was also limited.

Within the operating systems environment, one of the areas that will be affected is the **memory management system**. This is because the memory management algorithms for current operating systems are based on the assumption that memory is a scarce resource. With large amounts of cheap RAM becoming readily available, it is possible to adopt a more flexible approach. Perhaps, the most significant influence will come from the faster, more sophisticated **memory management units** or (MMUs) of modern microprocessors. These MMUs now use sophisticated virtual memory management techniques, which were previously only found on mainframe and minicomputer systems.

Virtual memory techniques will therefore play a major role in delivering the benefits of the new environment to its users. New virtual memory management systems must be designed to give users greater flexibility by providing simple yet powerful user interfaces. At present, users on most systems use one interface when accessing conventional program segments which is defined by the programming language and another interface when accessing objects residing in secondary storage which is defined by the operating system. If these two interfaces can be harmonised then the programmer can access objects in a unified manner, without worrying about the need to move objects to and from secondary storage. These new interfaces should also be designed with the intention of supporting persistent storage for object-oriented languages, such as C++, and object-oriented databases systems since they are becoming important facilities in many computing environments.

The use of the memory-mapped technique, in which objects are mapped into the user's address space on demand, is a key requirement since it hides the existence of the memory hierarchy and makes the system, not the programmer, responsible for the movement of data between main memory and secondary storage. Though memory mapped interfaces have been implemented on several operating systems, they do not give added functionality to the

user as they do not provide the facilities for users to easily implement logical abstractions.

The microkernel approach to operating system design, in which most of the services previously done by a monolithic operating system are now implemented using user-level processes, is also becoming very popular as it enhances the flexibility, portability and modularity of the system. This approach has been adopted in the design of a number of operating systems including Mach and Chorus. It should also be used in this effort since it emphasizes minimal changes to the microkernel with most of the work being done by other user-level processes.

Paging, the movement of the pages of a process between main memory and secondary storage as the process executes, is an important service of virtual memory management systems. Since program sizes are increasing, support for paging will remain an essential function of these systems for some time to come. However, as CPU speeds are increasing much faster than disk speeds, an I/O bottleneck may develop where the CPU spends a significant amount of time waiting for page requests to be satisfied.

To address this problem, it is necessary to review traditional paging techniques including conventional paging algorithms with a view to developing better analytical models of program behaviour. Previously, the development of paging models has been hampered by a poor framework with which to compare different paging algorithms and a lack of data on the paging activity of programs running on modern operating systems. This requires the development of a testbed from which experimental data can be obtained.

To improve overall efficiency of the system, an examination of the use of conventional paging algorithms on different types of objects is needed. It is essential to develop new paging algorithms that better reflect the different access patterns of various objects. Thus, it is necessary to implement paging on a **per-object** basis in which different objects may be paged using different paging algorithms. This will also facilitate the development of new paging algorithms for objects whose access patterns are well understood.

This thesis explores the issues discussed above. A new user interface is proposed and implemented on Wanda, an experimental operating system developed at the University of Cambridge. A new framework for comparing different paging algorithms is described and is used to analyse the performance of traditional paging algorithms using the GCC Compiler Suite. New paging algorithms are designed and implemented for objects that are accessed in a highly sequential manner.

Outline

An outline of this dissertation is presented below.

Chapter 2 reviews various memory management units highlighting features that affect operating system design.

Chapter 3 outlines the virtual memory management system of several operating systems, ranging from Multics, the ancestor of most present-day operating systems, to OS/2, the new operating system for Intel microprocessors proposed by IBM.

Chapter 4 examines the microkernel approach to operating system design using two prominent microkernels, Mach and Chorus.

Chapter 5 explores the issues in the design, implementation and analysis of paging algorithms and shows the need for a new framework.

Chapter 6 outlines the design of the new user interface while **Chapter 7** details its implementation.

In **Chapter 8**, a new framework for the analysis of paging algorithms is developed and results for different paging algorithms on the GCC Compiler Suite are presented and compared.

Chapter 9 explores the design and implementation of new paging algorithms to support objects that are accessed in a highly sequential manner.

Finally, **Chapter 10** details the conclusions and future work that can be derived from this effort.

Chapter 2

Memory Management Units

2.1 Introduction

Different approaches to virtual memory management are clearly reflected in the design of various memory management units. This chapter investigates the features of several MMUs, and in particular, aspects of their design that influence the virtual memory management system.

2.2 Virtual Memory Techniques

2.2.1 Virtual to Physical Translations

Virtual memory is based on a simple concept. The programmer is given a large, linear set of addresses, commonly called an **address space**. These addresses, known as virtual addresses, are generated irrespective of the physical addresses at which the information is actually stored. The translation from a virtual to a physical address is done as the program is being executed.

Both physical and virtual addresses are divided in small partitions called **pages**, typically from 256 bytes to 8 Kilobytes (KB) in size. A page is the unit of translation between physical and virtual addresses. Each individual page in an address space is represented

by a virtual page number, **vpn**, while each physical page is represented by a physical page number, **ppn**. Translation, therefore, involves taking the virtual address obtaining the vpn and finding the corresponding ppn. **Pagetables** are structures that contain the mapping between the vpn and the ppn and are usually stored in main memory. However, since access to main memory is slower than accessing other parts on the CPU chip, the most recent virtual to physical translations are stored in a fast **Translation Lookaside Buffer** or TLB usually located on or near the CPU chip. When a virtual address is generated, the TLB is first searched. If the translation is found in the TLB, called a TLB hit, the corresponding physical address is put out on the address bus.

If the translation is not found in the TLB, called a TLB miss, the pagetables in main memory are consulted to obtain the corresponding ppn. Translations in the pagetable are valid if a **valid** bit is set in the corresponding **pagetable entry** or PTE. If a physical page has not been assigned to the vpn, this condition is known as a **pagefault** and the operating system assigns a new page and transfers the data from secondary storage to the page. This activity is called **paging** and forms an integral part of virtual memory management.

The proportion of TLB hits for a given process is called the **hit ratio**. A very high ratio means that most of the virtual to physical translations are found in the TLB. Thus improved performance can be obtained by increasing the hit ratio of the TLB. In addition, since virtual to physical translations take place while the process is executing, it is possible that a process with a very large address space can be executed on machines with small amounts of physical memory. This is because most processes execute in relatively small regions of their address spaces for relatively long intervals.

This behaviour is referred to as **locality of reference**. A program spends long periods of time within a locale and shorter periods moving from one locale to the next. The time in a given locale may be represented as a **phase** in the execution cycle of the program. Thus the amount of physical memory that the process requires to continue execution is small and can be assigned as it executes rather than satisfying its entire requirements beforehand. The **working set** of each phase comprises the set of pages that the program accesses while in that phase. The **resident set** is defined as the set of pages in main memory that can be accessed by the program. The same principle can be extended to support a multi-tasking environment where different processes are concurrently executing in different address spaces.

2.2.2 Control Mechanisms

Multi-tasking also requires protection mechanisms which prevent processes from accessing or changing information that would damage or corrupt the system. The need for these mechanisms was realised in the early development of operating systems to protect the supervisor or executive program against unauthorised access by user programs. So, in addition to a physical page number, pagetable entries usually contain bits that govern access to the page. These include the **supervisor/user** bit, which, when set, indicates that the page could only be accessed in supervisor mode. There are bits to indicate read, write and sometimes execute access modes. Control mechanisms have been extended to cover several aspects of operating system design, including the global sharing of pages by all processes, represented by a **global** bit, as well as a bit to indicate whether data from the page may be cached by external caches. The **reference** bit indicates that an address on a page has been accessed while the **modified** bit indicates that an address on the page has been modified. These bits provide the hardware support for many paging algorithms used in modern operating systems.

2.3 Implementation Strategies

The different approaches to implementing hardware support for virtual memory are primarily related to the partitioning of the address space and how the virtual to physical translation is performed. The simplest approach is the **paged** approach in which the address space is divided into pages. There is only one pagetable per process and translation is done by using the vpn as an index into the table. This technique is simple and fast. However, the size of the pagetable is proportional to the size of the address space. In addition, the entire pagetable must be kept in main memory thus limiting the amount of multi-tasking that is possible due to a large amount of memory being used as pagetables.

Another method, which has been employed, is the **segmented** architecture. Here the address space is divided into large sections called **segments**. The main advantage in using this approach is that since segments are very large and only a few are needed in memory at the same time, the virtual to physical address translations for active segments can be held in special registers on the CPU. There is no need for a TLB. In addition, an entire segment can be paged in a single operation. However, since segments are associated with large amounts of physical memory, processes that do not use most of the segment will cause the corresponding physical memory to be under-utilised. This phenomenon is known as **fragmentation** and can severely reduce the performance of the overall system.

The **segmented-paged** architecture combines the two above approaches. In this scheme, segments are divided into pages so that each active segment has an associated pagetable. This architecture has several advantages. Firstly, paging reduces the fragmentation problem of the segmented architecture as there is no need to reserve physical memory for an entire segment. If a segment is accessed and the corresponding physical page is not there, a pagefault will be generated. In addition, only segments that are currently active in memory, must have pagetables assigned to them. Thus pagetables may be built as required. Finally, since it is possible to provide access control mechanisms for segments as well as pages, a greater degree of sharing can be obtained. Using different access control settings in the segment descriptors, it is possible for different users to share the same object (i.e. same pagetable) with different access rights.

The next approach to be considered is the **Inverted Page Table** (IPT) technique. With this method, shown in Figure 2.1, a hash function is performed on a virtual address to yield an index into a hash table. The corresponding value in the table is a physical page number referred to as a **pagetable index** or PTI. The PTI is used to index into an inverted pagetable to yield a virtual address. If the vpn in the IPT matches the vpn of the virtual address, the PTI is the ppn for that translation. This appears to be the reverse of normal translation – hence the term *inverted pagetable.* The main advantage of this technique is that the size of the IPT is proportional to the amount of *physical* memory the system supports.

However, different virtual addresses may hash to the same PTI. This is known as a **collision.** One way to resolve this is to keep the relevant virtual addresses in a linked list in the IPT so that the system follows the chain until it finds the correct virtual address or the chain ends which indicates a pagefault.

The main disadvantages of the Inverted Page Table approach are that the hash table and the IPT must be resident in main memory and two accesses are required for one translation. These effects can be reduced by using a sizeable TLB. In addition, only a limited amount of global sharing, in which an object is located at the same virtual address for all address spaces, can be easily supported. This is because the hash function makes it very difficult to map the same physical address onto different virtual addresses.

The methods described above are typically executed by the hardware once the relevant tables are set up and the corresponding registers are loaded. Another option, which is becoming increasingly popular, is to allow TLB misses to be managed entirely in software. Instructions are provided to obtain the faulted address and load the virtual to physical translation into the TLB, but exactly how the translation is done is a matter for the operating system designer not the hardware engineer. There are obvious advantages to

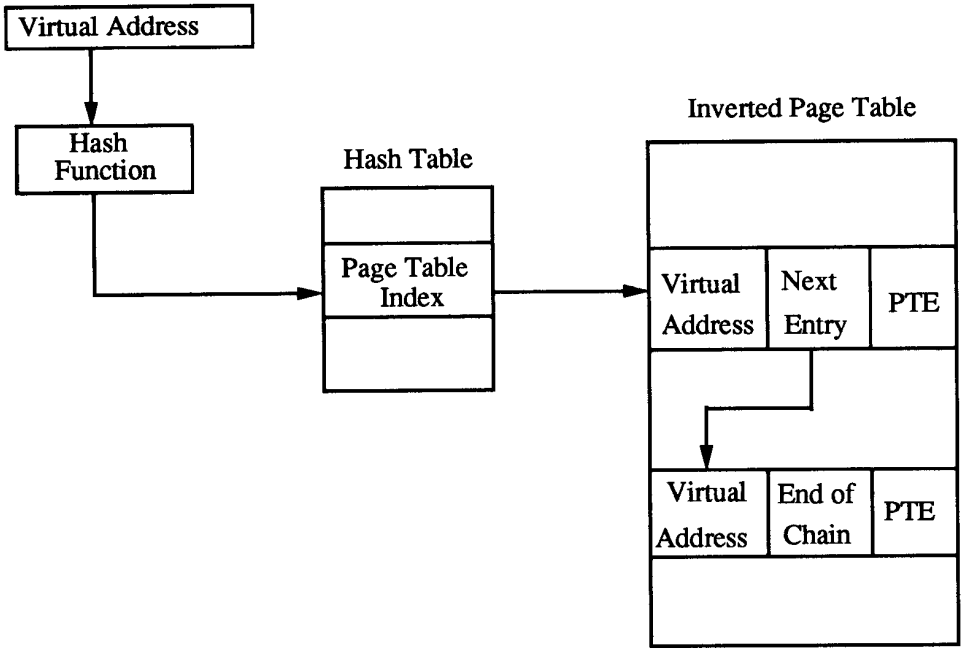

Figure 2.1: The Inverted Page Table Technique

this approach. Namely, the operating system designer is free to implement unique and specialised solutions to meet the specifications of his system while the amount of hardware design and silicon needed for the MMU may be severely reduced. This scheme may perform worse than a solution implemented in hardware if a significant number of TLB misses occur, thus a high TLB ratio is essential for good performance.

2.4 Specific Architectures

In this subsection, we examine several memory management units that use techniques described above giving a detailed description of a design based on each of the above approaches.

2.4.1 The Vax Architecture

The Vax is an example [**Dig86**] of a paged virtual memory architecture with some unique characteristics. The system uses a large pagetable as the **system pagetable** or spt. The physical address and length of the spt are kept in special registers known as the **System**

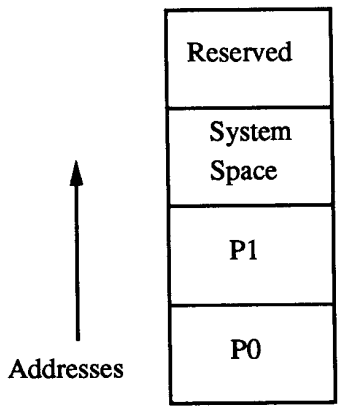

Figure 2.2: The Vax Address Space

Branch Register (SBR) and the **System Length Register** (SLR) respectively. All virtual address translations are done via the spt.

The address space of a process, shown in Figure 2.2, is divided in four fixed-size regions. P0 and P1 are regions that are specific to the individual process while system space is shared by all processes. The P0 region grows linearly upwards and is used to map the text and data segments of a process. The P1 region grows downwards. The stack as well as other control information are mapped in this region. The Vax supports a four Gigabyte address space in which the P0 and P1 regions occur in the first two Gigabytes with one Gigabyte used for system space and the other Gigabyte reserved for future use. The architecture employs a page size of 512 bytes.

P0 and P1 regions are represented by pagetables which are assigned from the spt. The start of these regions in the spt are indicated by the **p0br** and **p1br** registers while their lengths are placed in the **p0lr** and **p1lr** registers respectively. To translate a virtual address in the P0 region, the virtual page number is added to the p0lr register. The result is used to index into the spt to get the pagetable entry which contains the physical page number. Thus two memory accesses are required to do a translation in the P0 or P1 region. For an address in system space, the spt is used directly. This means that all processes see the same data in system space.

This scheme has a number of drawbacks. Firstly, to support a large virtual address space, the system pagetable must also be large. Secondly, to have small pagetables for the P0 and P1 regions, logically independent segments must be placed very close to each other, making it difficult to dynamically grow segments. Finally, though system space allows processes to share common utilities, it is difficult to provide some form of *limited* access

control where a small number of processes are required to have access to data, possibly with different access rights. The data has to be mapped into the address space of each of the processes involved using different pagetables. This introduces problems in keeping track of modification to pages since a number of pagetables must be simultaneously consulted, resulting in poor performance.

The Vax supports four different modes of privilege, namely, **kernel, executive, supervisor** and **user**. The kernel mode is the most privileged followed by the executive and then supervisor modes respectively. The user mode is the least privileged. By combining these privilege modes with read and write controls, it is possible to support a wide range of access control levels. The system works via the following rules:

1. The access modes supported by the system are read/write, read-only and none.
2. Whichever level has read access, all more privileged levels also have read access.
3. Whichever level has write access, all more privileged levels also have write access.

For example, the access mode, supervisor-read-only and kernel-write, means that in kernel mode the page may be read from or written to, in executive or supervisor modes the page is read-only, while in user mode, it is not accessible. The access control modes are represented by four bits in each pagetable entry. There is also a valid bit as well as a modified bit. The Vax 11/780, the first implementation of this architecture, employed a TLB consisting of 128 entries, with a reported hit ratio 98% [**Hennessy90**, pages 441–444].

2.4.2 The Intel 80286

The segmented approach is seen in several architectures including the Burroughs B5000, PDP series and more recently the 80286 microprocessor [**Ciminiera87**], [**Hennessy90**, pages 445–449]. The 80286 supports a one Gigabyte address space and offers a maximum physical size of 16 MB. Segments are 64KB long. A virtual address is comprised of two components, a **segment offset**, which is 16 bits long, and a **segment selector**. The address resolution scheme is shown in Fig. 2.3.

Each process has two segment descriptor tables, the **local descriptor** table or (LDT) and the **global descriptor** table or (GDT). The T1 bit in the segment selector indicates which table must be accessed to obtain the relevant segment descriptor. Like the Vax, global segments are seen by all processes. The privilege level of a segment is represented by two bits in the segment selector called the RPL field. Segment selectors are loaded into four

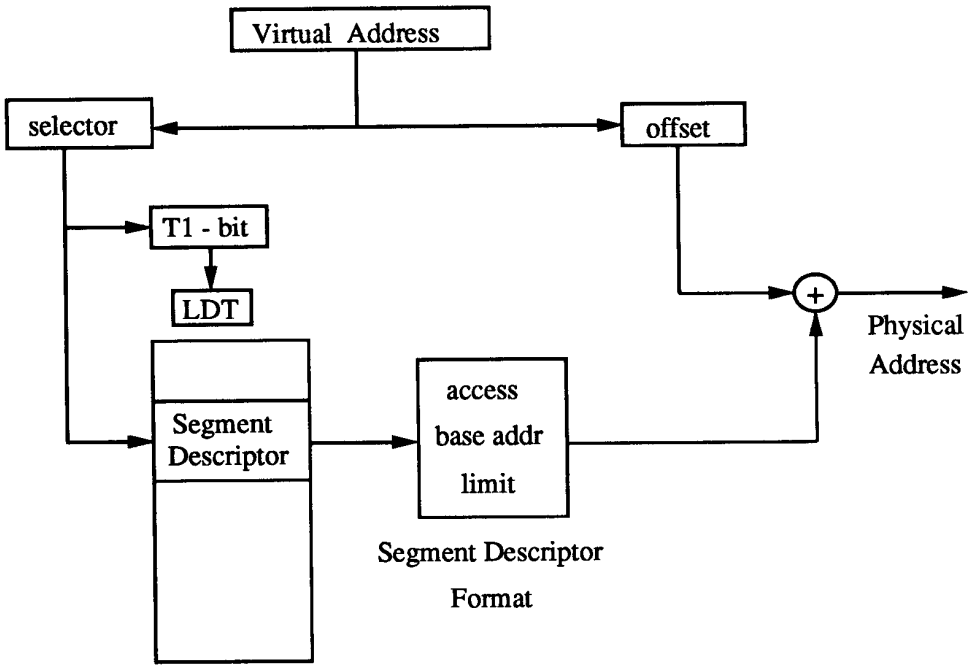

Figure 2.3: Address Resolution on the 80286

registers in the CPU. These include the code, data and stack segment registers. The fourth segment register can be used to access another segment. When a context switch occurs, these registers are loaded with the code, data and stack selectors for the next task.

An interesting feature of the 80286 MMU design is its access control mechanism. There are four levels of privilege with level 0 being the highest and level 3 being the lowest. The **current privilege level** or CPL of a process is the privilege level contained in its stack and code segment selectors. The access rules may be summarised as follows:

1. Data segments are accessible only by tasks at the same or higher level of privilege.
2. Subroutines can only be called by tasks at the same or lower levels of privilege than the called routine.

Each segment descriptor contains an access control byte which includes a valid or present bit, a read/write bit as well as an executable bit, which is set for code segments. There are bits which indicate the direction segments are expected to grow and an access or reference bit which is set by the hardware when the segment descriptor is accessed. This can be reset periodically by the operating system and hence be used as an indication of how frequently a particular segment is referenced.

In addition, there is a two-bit field in the access control byte that indicates the **descriptor privilege level** or DPL for that segment. When a segment register is loaded with a new value, access control checking is invoked on the DPL field to ensure that unauthorised access does not occur. When a data segment is loaded, the hardware checks that the CPL is the same or less than (i.e. *greater privilege*) the DPL of the segment.

Transfer of control to other segments is carefully checked to ensure that the transfer address is accessible by the task, the constraints of the privilege level are met, and the transfer destination address is the correct entry point for the routine. This is done using a **gate mechanism** which handles the transfer of control to another routine via a **call-gate**. A **task gate** handles the transfer of control to another task. In addition, an **interrupt gate** handles interrupts while exceptions are fielded via a **trap gate** mechanism.

2.4.3 The MC68851

The complexity of the segmented-paged architecture is reflected in the design of the MMUs adopting this scheme. These MMUs therefore require a larger amount of silicon and depending on the number of features that the designers want to support, an off-chip MMU is sometimes built. We examine one such MMU: the paged memory management unit chip of the 68020, the MC68851, also known as the PMMU [**Mot86**], [**Milenkovic90**, pages 81–82] . This chip supports a total address space of 32 Gigabytes which is subdivided into eight 4-Gigabyte regions. Each region is associated with a task number or *alias* and contains the address space of an individual task. A complete virtual address comprises a task alias and the virtual address generated within the address space of that task.

Translations are done using a multi-level translation tree starting from a designated root pointer. The tree may be up to five levels in depth, each level containing a pointer to the next table until the pagetable is reached. At each level of the tree, access control mechanisms can be applied. This allows different tasks to share the same segment with different access control privileges. There are bits in each descriptor table at each level to indicate supervisor/user access as well as read-only access which signifies that the segment or page is protected against writing in both supervisor and user privilege modes.

An interesting feature of the MC68851 is that it supports both **long** and **short** segment and page descriptor formats. Essentially, the short descriptors are 2 bytes long and access control is governed by the mechanisms already mentioned above. The long descriptors are 4 bytes long and contain two additional 3-bit fields for various read and write access privileges. This allows the operating system designer as well as application engineers to build very sophisticated access control mechanisms.

The page descriptor also has several control bits including a lock bit, which locks the translation in the TLB, a modified bit and a reference bit. There is also a cache inhibit (CI) bit which, if set, specifies that addresses on the corresponding page may not be stored by external caches. A global bit, which indicates that the translation is shared by all running tasks, is also present.

Another distinguishing feature of the MC68851 is that a page descriptor in one pagetable can point to another descriptor in a different pagetable. This indirection allows the same translation to be shared by all tasks using the page and thus the state of the modified and used bits reflect the use of the page by all the tasks. The PMMU also supports several page sizes, from 256 bytes increasing in powers of 2, to 32 KB. This allows the operating system designer to choose an appropriate page size.

The **Address Translation Cache** or (ATC), another name for a TLB, contains 64 entries and is fully associative. The logical address part of a TLB entry contains the tag, the logical address and a bit which indicates if the entry is shared globally (SG) by all tasks. If the global bit is set in a page descriptor, then the SG bit is also set in the ATC. When the SG bit is not set, a match occurs if both the task alias and the virtual address match the logical address in the ATC.

However, when the SG bit is set, only the virtual address field is examined and comparison of the task alias field is suppressed. Thus the match will be found by every task and only one entry in the ATC is sufficient. Moreover, since entries in the ATC are replaced using a modified **Least Recently Used** or (LRU) policy, globally shared entries that are frequently accessed will remain in the ATC for long periods, substantially improving performance.

Transfer of control to other routines with different access privileges is achieved using the **CALLM** instruction. This instruction is executed using special module descriptors or gates. The address of the module descriptor is passed during the CALLM call and is checked to ensure that the corresponding page descriptor has its gate bit set. This bit indicates that the module is valid and prevents a user from maliciously or erroneously using an invalid module descriptor. The instruction may also specify that a stack change must occur. This forces the arguments to be copied onto a separate stack.

2.4.4 The HP Precision Architecture

The inverted pagetable approach is gaining popularity because of its simplicity and its ability to efficiently support very large sparse address spaces. The **Precision Architec-**

ture (PA) is a RISC architecture announced by Hewlett-Packard (HP) in 1986 [**Hew87**], [**Mahon86**]. The MMU supports three different configurations. Level 0, which uses absolute addressing while Level 1 and Level 2 provide virtual addressing.

The virtual memory system is structured as a set of different address spaces, each comprising 4 Gigabytes. Level 1 contains 2^{16} address spaces while level 2 supports 2^{32} address spaces. The individual address spaces are specified using space registers and are divided into 2 KB pages. There are eight space registers labelled 0 through 7. There are also two TLBs, an instruction TLB (ITLB) and a data TLB (DTLB) – though it is possible to combine the two to form a single TLB. Each TLB entry has a 15-bit **access ID** field as well as an access type field which forms part of the protection mechanism of the system.

The system uses an inverted pagetable which may have negative offset values. These values are used to indicate the mapping of virtual I/0 devices. The PA also has an elaborate access control mechanism. There are four levels of privilege ranging from level 0 to level 3, with level 0 being the most privileged. Access bits for read, write and execute operations are also provided. There are four control registers that contain protection identifiers associated with the current process. These contain 15-bit fields and one of them must match the access ID field in the TLB before access to the page is granted.

The access type field is subdivided into three fields, a type subfield, privilege level 1 (PL1) and privilege level 2 (PL2). The type subfield contains the access bits mentioned above, but the PL1 and PL2 subfields *qualify* access. For read access, the current privilege level (CPL) must be at least as privileged as PL1. For write access, the CPL must be at least as privileged as PL2. For execution access, the CPL must be as least as privileged as PL1 but no more privileged than PL2.

Other features of the MMU include reference, valid, and modified bits. There is also a **page reference trap** bit which, when set, causes a page reference trap interrupt when a data reference is made to the page. This is used for debugging. In addition, there is a break bit which also causes a data memory break trap interrupt when instructions that modify data use this translation.

2.4.5 The MIPS Architecture

As previously mentioned, many operating system designers are beginning to favour the software TLB approach as it allows for greater flexibility. The MIPS R2000/R3000 architecture employs such an approach [**Kane88**].

The architecture supports a 4-Gigabyte address space of which two Gigabytes can be

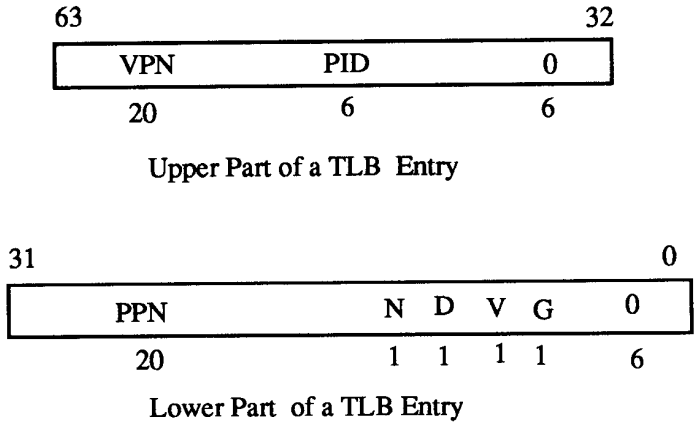

Figure 2.4: A TLB Entry for the MIPS Architecture

accessed in user mode. When a process is executing in kernel mode, three regions can be accessed. Firstly, there is a 512 MB cached, unmapped segment. This is used to map the first 512 MB of physical memory into the kernel address space. References to this region can be cached but no entries are stored in the TLB. This is normally used for the operating system. Secondly, there is another 512 MB segment which is both uncached and unmapped. This is typically used for I/0 registers, read-only memory (ROM) and disk buffers. Finally, there is a 1-Gigabyte virtual address space that is mapped and is used for setting up pagetables as well as allocating memory for stacks and dynamic data structures. This region allows mapping on a per-page basis.

The system supports a page size of 4KB and a TLB containing 64 entries. As shown in Figure 2.4, each entry includes a vpn and a six-bit process identifier (PI) field, which allows multiple processes to share a TLB. The other part of the entry comprises the ppn and bit fields to indicate constraints on the use of the page. The N bit specifies whether data on the page can be cached and the dirty bit indicates whether data has been modified. The valid bit indicates whether the entry in the TLB is valid and the global bit indicates whether the page is globally shared. There are two registers that are used to read, write and probe the TLB. These registers are referred to as the **Entry Hi** and **Entry Lo** registers and correspond to the two parts of each TLB entry. When an address translation exception occurs, these registers are loaded with relevant information about the address that caused the exception.

Exception Handling Using the Software Approach

Two additional registers are used in handling TLB misses. They are the **index** and **random** registers. The architecture provides the exception handler with four instructions to manipulate the TLB. These are:

Translation Lookaside Buffer Probe (tlbp) – this probes the TLB to see if an entry matches the Entry Hi register contents. If a match occurs, the index of the entry is loaded into the index register.

Translation Lookaside Buffer Read (tlbr) – this instruction loads the Entry Hi and Entry Lo registers with the contents of the TLB entry specified by the contents of the index register.

Translation Lookaside Buffer Write (tlbw) – this instruction loads the TLB entry which is specified by the index register, with the contents of the Entry Hi and Entry Lo registers.

Translation Lookaside Buffer Writer Random (tlbwr) – this instruction loads the TLB entry which is specified by the random register with the contents of the Entry Hi and Entry Lo registers.

When a TLB miss occurs, the exception handler first invokes the tlbp instruction to find out if there is a TLB entry for the faulted address. If a TLB entry exists, it checks the state of the bit fields to determine why the translation was not executed. If an entry does not exist, it must replace an entry in the TLB if the TLB is full. To do this, it loads the index register with the TLB entry it is going to replace and then invokes the tlbr instruction. If the TLB entry is being replaced randomly, then the value of the random register is used.

The exception handler stores the contents of this entry in its software data structures. It then puts the contents of the correct translation into the Entry Hi and Entry Lo registers respectively and performs a write index or write random instruction to load the translation into the TLB.

2.5 Summary and Conclusions

In this chapter different approaches to the design of MMUs were surveyed and implementations based on each approach were examined. The paged approach has several deficiencies

including large pagetables, its inability to manage logically different entities efficiently as well as inadequate mechanisms for sharing objects with different access rights. The segmented scheme has the advantage of not requiring TLB support. However, the problem of fragmentation associated with this scheme represents a serious drawback.

The segmented-paged approach combines the benefits of both the paged and segmented architectures. The added complexities associated with this design are outweighed by the advantages of smaller pagetables, no segment fragmentation problems, and the ability to share objects with different access rights. Improved VLSI techniques now allow these MMUs to be on the CPU chip as seen in the 68030's and 68040's. It is interesting to note that Intel, the manufacturer of the 80286 microprocessor, has switched to the segmented paged-architecture for the 80386's and 80486's.

The inverted pagetable technique allows the total pagetable size to be proportional to the size of physical memory and not the size of the virtual address space. Thus it manages large sparse address spaces with great efficiency and since the size of an address space continues to increase (e.g. 64 address bits for the Precision Architecture), this approach will be increasingly adopted. However, the inability of this scheme to easily support an object that is mapped into several address spaces at different virtual addresses with different access rights is a severe hindrance for systems using the memory-mapped approach.

The software-managed TLB approach is the most flexible of all the schemes. It also requires minimal hardware and allows the system designer to implement software data structures to support new features for building advanced virtual memory management systems. As CPU speeds increase, the cost of handling TLB misses in software will fall and thus this approach will be more frequently used. At present, however, the cost of a TLB miss when using this approach is still about 10 times slower on some architectures [**Edenfield90**] when compared with the other techniques that use hardware mechanisms to find the correct translation.

In addition to providing sophisticated access control mechanisms, most MMUs have features which aid the implementation of virtual memory management systems. These include reference and modified bits for paging algorithms, TLB support for multi-tasking and the cache inhibit line for cache coherency. There is also support for the global sharing of objects as well as debugging as seen in the Precision Architecture. Given the continuous improvements in VLSI techniques, more features will continue to be introduced.

Using these features, virtual memory management systems can be designed to provide efficient and flexible interfaces that contribute significantly to the design of modern operating systems. The virtual memory management systems of a diverse set of operating

systems are examined in the next chapter.

Chapter 3

Virtual Memory Management in Operating System Design: An Historical Perspective

3.1 Introduction

From its introduction on the Atlas computer system in 1961 [**Kilburn62**], virtual memory has been employed in the design of the memory management of various operating systems. Initially, the cost of hardware support for virtual memory limited its use to large mainframe environments. However, with the continuous improvements in VLSI design, it has been introduced in minicomputers and, more recently, on microprocessor architectures. This chapter examines virtual memory management in seven operating systems and the facilities provided by these systems using virtual memory techniques.

3.2 Multics

Multics was developed as a joint project between MIT, Bell Labs and General Electric (GE) and was conceived of as a computer utility providing support for hundreds of time-sharing users [**Organick72**]. It was developed in the Sixties on the segmented-paged architectures of the GE635 and GE645.

This hardware supported a virtual address space of 2^{36} 36-bit words using a page size of 1024 words. The address space was divided into segments, which had a maximum size of 64 Kilowords (KW). Each process had a Descriptor Segment Table (DST), which was indexed by a segment number. The segment descriptor of an active segment also contained the address of the associated pagetable [**Bensoussan72**]. With such a large address space, the approach adopted was that each segment would represent a logical entity such as a file, procedure or an array. These segments were dynamically mapped into the address space of the process as the process was being executed.

Each segment contained a number of attributes including an **access control list**, which indicated the users authorised to access the segment. Supervisor calls were used to create a segment, delete a segment, change its access rights or its entry name as well as list its attributes. The name of each segment and its attributes were placed in a **catalogue** or filing system which was implemented in a hierarchical tree structure, beginning with a root directory. The **path name** of a segment was specified relative to this directory, while its entry name, the name by which a segment was usually referenced, was the last name in the path name. The **current directory** was first searched when a process accessed a segment and the **path mechanism** governed search rules when trying to locate a segment in the catalogue.

When a program first referenced a segment, a dynamic linker routine was invoked to find its path name. The segment was then assigned a unique number in the segment descriptor table. In addition, each known segment was kept in a per-process structure called the **Known Segment Table** (KST). When the path name and the segment number were obtained, a (path name, segment number) entry was made in the KST. Further references to the segment were resolved using its segment number. The pagetable associated with a segment was actually mapped in by the segment fault handler after a missing segment fault was generated when the segment was first accessed.

The Multics system used ring protection mechanisms to support access control [**Saltzer74**]. This system comprised a set of concentric rings representing different levels of privilege. Ring 0 was the most privileged and contained essential kernel code, like the interrupt handlers, scheduling routines, etc. Processes executing segments in this ring were non-interruptible until they were finished. Access privilege decreased as one moved further away from the centre. Sixty four rings were supported on the GE645 hardware with user processes usually executing from ring 33 onwards. Each segment descriptor also contained the ring or more commonly a *group* of rings in which the corresponding segment must be executed. If it was necessary to execute a segment in a more privileged ring, an **inter-ring access fault** was generated and the operating system invoked access control routines

before the faulting process resumed execution.

Multics introduced many seminal ideas in operating system design which are still adhered to today. It was used at MIT and at approximately 100 other sites. However, because it depended heavily on the GE645 architecture, which was expensive and proprietary technology, it was outside the financial reach of most other computing research environments and universities. In addition, the concept of a general computing utility was increasingly being challenged by the arrival of minicomputers. These were less powerful but also much less expensive than the mainframe systems. They also had good interfacing facilities to numerous devices. Perhaps the most famous minicomputer series is the PDP series developed by Digital Equipment Corporation (DEC) .

3.3 The Cap Computer System

The Cap Computer System was developed at the University of Cambridge in the Seventies to explore memory protection using hardware mechanisms [**Wilkes79**]. The design used a segmented architecture with a process having a list of active segments. Unlike conventional systems, where the loading of the base segment registers must be done when the machine is in a privileged mode, with the Cap system, loading of the segment registers was done in any mode, but the hardware carefully restricted the entries that could be loaded. Valid entries were associated with **capabilities** and access to a segment required a capability for that segment.

The creation and the checking of capabilities was done by a **Capability Unit** which was controlled by a section of the microprogram [**Needham77b**]. Capabilities were divided into two types. The **D-type** or data-type capability was used to access data with read, write and execute. A **C-type** or capability-type capability was used to access a segment containing a list of capabilities and the access privileges were **read capability** and/or **write capability**.

Like Multics, segments and files shared a close relationship and the virtual memory system interfaced directly with the Cap filing system to manage segments being brought into or leaving main memory [**Needham77a**]. The virtual memory management system comprised a number of system processes namely, the **real store manager**, the **virtual store manager** and the **System Internal Names Manager**, known as SINMAN.

The real store manager was responsible for bringing segments into memory when needed and deciding which ones should be swapped out when memory was required by other segments. When a segment was not in memory the associated capability had a special

form known as an **outform capability**. If a process attempted to access the segment, a trap would occur. The system would then check the capability for the segment and would discover that the segment was not in memory. It would then contact the real store manager to bring it into memory. On accomplishing this task, the real store manager would convert the outform presentation of the capability into a normal one. The real store manager handled single segments no larger than 32 KW. Accessing larger segments or files was done using windowing techniques. It also provided facilities like flushing updates on a segment to disk and allowing users to swap out segments that were no longer in use.

The virtual store manager was involved in detecting whether the segment in memory was no longer in use. The number of processes having a capability for a given segment was reference-counted. This count was incremented when a process was issued with a new capability for the segment and was decremented whenever the virtual store manager received a message from a process indicating that it was no longer interested in the segment.

SINMAN was responsible for permanent storage and was the primary interface between the virtual memory system and the Cap filing system. An integer was associated with every object in the system and was referred to as the **system internal name** for that object. Cap supported three types of virtual objects:

Segments: treated as a linear sequence of bytes.

Directory Segments: related text names to System Internal Names.

Procedure Description Blocks (PDB): templates to construct protected procedures.

SINMAN had the responsibility of keeping track of all the objects in the system that should be kept in existence and marking all objects that should be deleted from the disk. Objects in the former category were either kept in directory segments or procedure description blocks or were being used by current processes. SINMAN deleted a segment once its reference count fell to zero. Directory segments and procedure descriptor blocks were only deleted if the objects they contained were also not referenced. Another duty of SINMAN was to issue capabilities for particular objects to authorised programs.

3.4 UNIX

Unix, developed at Bell Labs by Ken Thompson and Dennis Ritchie, has been one of the most popular operating systems of the last two decades [**Bach86**]. It was first built for

the PDP series but has since been ported to many other architectures. The success of Unix can be traced to many factors including the fact that it was written in C, making it very portable. In addition, it ran on the PDP series which most computer departments could afford and Unix was, for a long time, freely distributed to various university sites along with its source code.

Another design factor that made Unix easily portable was that it had a relatively simple virtual memory management system. A Unix process is composed of three regions: the text region which is fixed, the data region which grows upwards and the stack region which grows downwards. There are system calls to extend the data region and the operating system will extend the stack as required.

Unlike Multics, the original Unix system does not support the idea of memory-mapped files or the dynamic linking of segments. The file system, though it employs the same hierarchical scheme as Multics, is implemented using a buffer cache and disk block structures and is independent of the memory management system. In fact, some physical memory must be allocated exclusively for the buffer cache when the system is booted.

Unix provides an integrated I/O interface in which operations on files as well as devices use the same system calls. Devices such as the monitor and keyboard, are regarded as special files by the file system. This is an elegant abstraction that makes I/O operations easy to perform from the point of view of the end-user since there is no need to be familiar with the actual physical device.

This I/O interface supports two calls that involve the movement of data, namely:

read (fd, buffer, n)

write (fd, buffer, n)

where:

fd is a file descriptor.
buffer is the address of a buffer in the caller's address space.
n is the number of bytes to be transferred.

The **read** call involves locating the data on disk, copying it to a buffer in the kernel, then copying from the kernel into the user's address space starting at **buffer**. This scheme has several disadvantages.

Firstly, two copies are required to move data from the disk into user space, which is expensive. Secondly, since each user has a private copy of the data, it is difficult to share

files concurrently because there is nothing to stop users from simultaneously updating private copies of the same part of the file. In addition, changes made by a user would have to be flushed back to the disk immediately to allow other users to see them and they can only do so by invoking the read operation again!

The file interface in Unix is character-oriented, where a file is basically viewed as a linear sequence of bytes. There are calls to read from and write to a file as well as to move to a given position in a file, thus providing limited random access. Since most users require sequential input, and output data sequentially as well, this abstraction will suffice for most applications. However, for users needing a different data abstraction, (e.g. viewing a file as a set of records), it is necessary to build this abstraction on top of the Unix abstraction or to use the I/O interface directly, which are both inconvenient.

Another feature of Unix is the **fork** and **exec** system calls. The fork call creates another process called the **child**, which is an exact replica of the process invoking the call, called the **parent**. Early implementations of fork involved copying the data and stack regions of the parent into those allocated for the child while the text segment, which is read-only, is shared. Such copying is time-consuming and later versions of Unix used the **copy-on-write** (cow) technique for managing the stack and data regions. These regions are now remapped read-only with the operating system setting a bit in each PTE to indicate copy-on-write sharing. When a child or parent attempts to write to a given page, an access violation fault occurs. The fault handler notices that the region is mapped copy-on-write and the faulted process receives a private copy of the page before continuing. This reduces the amount of copying since only pages that are modified in each region are copied.

The exec call replaces the segments of the calling process with the segments of the process whose name is passed as an argument to the call. This allows the programmer to implement a multi-tasking environment by first forking off the child process and then doing an exec call to start another program. If the parent has opened files before invoking the fork call, these descriptors will also be available to the child. Thus, before the exec call, it is possible to close files which will not be used by the new process as well as redirect its input and output.

3.5 BSD Unix

The most influential group outside Bell Labs that has contributed new ideas to the Unix evolution has been the Computer Science Department at the University of California, Berkeley. Contributions include the first Unix implementations of virtual memory, demand

paging and page replacement as well as the IPC socket mechanism and the TCP/IP networking protocol suite [**Leffler89**]. The BSD Unix memory management system was influenced by Tenex [**Bobraw72**] as well as Multics and an interface was specified to support memory-mapped files in the 4.2 BSD reference manual.

The original interface, known as the **mmap** interface, allows a user to map a file or portions thereof into the address space of a process. This interface supports options of read, write and execute permissions, while sharing may be specified as private or shared. If sharing is private, indicated by the MAP_PRIVATE option, changes made to the file by the user are seen only by that user. However, if the MAP_ SHARED option is used, all changes will be seen by other processes sharing the file. There is also a **getpagesize** call which returns the system pagesize. A process can move pages within its own memory using the **mremap** call. In addition, protection of a region can be controlled using the **mprotect** call.

The **madvise** call is used to advise the system on how a process intends to utilise a region of its address space. The options include:-

MADV-NORMAL: no special treatment.

MADV-RANDOM: expect random references.

MADV-SEQUENTIAL: expect sequential references.

MADV-WILLNEED: will shortly require the pages in this region.

MADV-WILLNOTNEED: pages associated with this region are no longer needed and may be removed.

A process can obtain information about whether the pages of a given region are in memory using the **mincore** call. This returns the current core residency of the pages in terms of a character array with a value of 1 indicating that the page is in core. Finally, the **munmap** call will unmap a region of pages from a given address space.

The mmap interface was revised and implemented on SunOS, the operating system created by Sun MicroSystems (SUN) [**Gingell87**]. The protection options have been increased to include MAP_NONE which indicates that pages of a given region cannot be accessed. The MAP_FIXED option is used to tell the virtual memory management system that the new region must start at the specified address. If MAP_FIXED is not set, this address is used as hint to the operating system, but the system may return a different starting address. If MAP_FIXED is set, and the system cannot map the region at the specified address, then the call fails.

MAP_RENAME causes the pages currently mapped into a region to be assigned to the pages of a file at a given offset and length. The relevant pages must be mapped privately using the MAP_PRIVATE option. MAP_INHERIT indicates that a mapping will be passed onto a child process and is present after the child does an exec call to start another program. MAP_NORESERVE tells the system not to reserve any swap space for a given region.

Another call was added by the SunOS team. The **msync** call causes all modified pages within a specified region to be flushed to backing store and may invalidate these pages in the local cache causing further references to be obtained from the backing store. The flags options are MS_ASYNC, which allows the caller to return immediately, and MS_INVALIDATE which tells the system to flush caches. The mremap call was not implemented.

A new interface has been proposed [**McKusick87**]. One of the reasons for this attempt was the desire to use memory-mapped files for high speed interprocess communication. The motivation is to improve the performance of synchronisation mechanisms by avoiding the overhead associated with using system calls to achieve synchronisation. This is done by mapping the semaphore that controls access to the shared region as part of the shared region itself.

New options for the mmap call include MAP_FILE, which specifies that the object is a regular file or character-special device memory. MAP_ANON is used to indicate that the object concerned is a private, zero-filled region which cannot be flushed to backing store and will be discarded after it is used. The MAP_HASSEMAPHORE option is used to indicate that a region may contain semaphores. Another option for the madvise call was the MADV_SPACEAVAIL flag which forces the system to allocate physical pages for that region.

New calls have also been added to deal with synchronisation. The **mset** call will test and set a given semaphore. Another argument specifies whether the process will wait if the semaphore is not free. If it is non-zero, the process relinquishes the processor until notified to retry the call. The **mclear** call releases the lock. If processes are waiting on the semaphore, they are advised to retry for the lock.

3.6 VMS

VMS is an operating system developed by DEC to run on the Vax hardware [**Kenah84**]. This VAX/VMS arrangement forms quite a popular computer environment. The system

supports a sophisticated memory management interface and gives users greater control over the memory usage of their programs compared with other operating systems.

Since VMS was designed for the Vax, its virtual address space for a process is identical to that of the VAX architecture. The P0 region is called the **program** region while the P1 region is called the **control** region. Pages can be added to the program or control regions via the **$EXPREG** call which returns the range of virtual addresses of the new pages. The **$CRETVA** system service creates a separate segment in the P0 or P1 region. Arguments to this call include whether the region should be read/write or read-only, and the level of privilege, i.e. kernel, executive, supervisor or user, associated with the region. Regions created by the $CRETVA and $EXPREG may be deleted using **$DELVA** call. This call deletes pages beginning with a specified address. However, the caller must have the same or higher privilege than that associated with the region.

The **$CRMPSC** call is used by a process to associate or map a section of its address space with a specified region of a file or another region in memory. The **$MGBLSC** call is used to map a global section that has been created by another process. A process can map a part of a section into a particular address space and subsequently remap a different part of the section into the same virtual address space giving a windowing effect. A process unmaps an object from its address space via the **$DELTVA** system call. A temporary global section is deleted when all processes sharing the section have unmapped it from their address spaces. Permanent global sections must be explicitly deleted using the **$DGBLSC** system service. Modifications are written back to disk by the **$UPDSEC** call.

The VMS memory management system also provides facilities to control both the paging and swapping operations on certain regions. The interface includes:

$ADJWSL: which increases or decreases the maximum number of pages the process can have in its working set.

$PRGWS: removes one or more pages from the working set.

$LKWSET: will lock the current working set into physical memory.

$ULWSET: will unlock the pages of the working set which has been locked $LKWSET.

Process swapping, the removal of the resident set of a process to secondary storage in one operation, is usually triggered by low activity or large delay (e.g. waiting for I/O), and is normally applied to all processes in VMS. A privileged process, however, can prevent swapping using a **$SETSWM** call. This mode can also be set when a process creates

another process provided that the creating process has the required level of privilege. These privileges also allow a process to lock pages into memory using the **$LCKPAG** system service. A page that is locked in memory, remains in memory even when the rest of the working set has been swapped out. These pages may be unlocked using the **$ULKPAG** service.

Finally, the memory management system also allows protection variables to be set on individual pages. This is done using the **$SETPRT** system call which changes the protection of a page or a group of pages. The arguments to the call specify the type of access (none, read-only or read/write) for which each of the four access modes (kernel, executive, supervisor and user). Only the owner or a more privileged process can change the protection on a page.

3.7 Pilot

The operating systems described above support multi-user time-sharing computer environments. However, with the emergence of less expensive hardware, the personal computer and workstation environments have evolved in the last decade to replace time-sharing systems as the dominant computing models of the future. Pilot [**Redell80**] is an operating system developed for a workstation environment and is a successor to the Alto operating system [**Thacker82**]. Like Multics, Pilot supports an intimate coupling between its file system and its virtual memory management. Virtual memory is the only way to access files and files are the only backing store for virtual memory.

The virtual memory management system supports a total linear address space of 2^{36} 16 bit words. Pilot divides this homogeneous address space into contiguous sets of pages called **spaces**, which are managed via an interface called SPACE. Spaces can be used in three primary functions:

Allocation Entity: allocates a region of virtual memory which has been requested by a client.

Mapping Entity: associates a region of virtual memory with part of a file.

Swapping Entity: transfers pages between primary memory and the backing store. Thus the system allocates a new space to move data from the disk into memory.

Spaces are managed using a nested approach. So a new space is always a subset of a previously existing space. The resulting tree structure has as its root a predefined space

consisting of all the virtual memory in the system.

The calls provided by the space interface are outlined below. The **Space.Create** call allocates a space of a given size which may be further subdivided into smaller regions. There is also a **Space.Map** call which associates the space with the contents of a file. Swapping involves removing an entire space to backing store. A client program can thus optimise its mapped spaces by dividing them into subspaces, each containing items whose access patterns are known to be strongly correlated.

In addition, the **Space.Activate** call is used by the client to advise Pilot that a space will soon be needed and should be mapped in as soon as possible. Conversely, the **Space.Deactivate** call informs Pilot that a space is no longer needed in primary memory. The **Space.Kill** indicates that the contents of the space is no longer of interest and will be overwritten shortly. This prevents useless swapping back to memory. Finally, there is a **Space.ForceOut** call which causes the contents of a space to be written to its backing store and does not return until the operation is completed. This call was added to support crash recovery algorithms that clients may want to implement.

3.8 Operating System/2

Operating Systems/2 or OS/2 was developed by IBM as the successor to the widely popular Disk Operating System or (DOS) which was introduced for the IBM PC. OS/2 was specified to run on the 80286 and 80386 Intel microprocessors [**Cook88a**]. While the 80286 is a purely segmented architecture as discussed in Chapter 2, the 80386 supports both segmentation and paging. However, OS/2 utilises a segmented approach to remain compatible with the 80286 architecture. An additional constraint involves the need to support the large amount of DOS software that is in existence.

OS/2, therefore, can run in both a **real** mode, in which the virtual memory is turned off and memory access is constrained to 640 KB, that of the DOS environment, or **protected** mode which uses the virtual memory hardware. The system also supports a number of new facilities including physical memory access up to 16 MB, multiple concurrent applications, multi-tasking, system and user-level semaphores, dynamic linking of processes and the demand loading of code and data segments. Good support also exists for writing device drivers [**Mizell88**].

The memory management system draws heavily, some [**Shammas88**] would argue too heavily, on the architectural design of the 80286. All memory objects are stored in a single system-wide table called a **Handle Table** which is used to map objects to segments. As

indicated previously, segments that are shared by all processors are stored in the global descriptor table (GDT) while local process segments are stored in the local descriptor table (LDT). However, to permit a more controlled form of sharing, the LDT in OS/2 is divided into public and private sectors. If a segment is designated as sharable in the LDT, then it is placed in the public sector and that slot is reserved in every other LDT in the system. This allows programs to pass pointers to shared data since it is mapped in the same position of every address space.

Segments in OS/2 are directly visible to application programs and system calls are used to perform operations on them [**Cook88b**]:

DosAllogSeg: allocates a memory segment ranging from 1byte to 64 KB.

DosGiveSeg: indicates the segment is sharable.

DosGetSeg: flags a segment to indicate that it can be discarded.

DosReallocSeg: this call changes the size of a segment. Sharable segments are only allowed to increase in size.

DosFreeSeg: frees an unshared memory segment. If the segment is shared then the usage count is decremented and the segment is freed if the count is zero.

DosMemAvail:- returns the largest available region of free storage.

DosLockSeg: this call prevents a segment that can be discarded from being removed by the system.

DosUnlockSeg: reverses the action of DosLockSeg.

DosAllocShrSeg or **DosGetShrSeg**: these calls are used to allocate a shared segment.

DosAllocHuge: this is used to allocate a number of 64 KB segments.

DosGetHugeShift: this is used to address the individual selectors obtained from the DosAllocHuge system call when a large object is mapped into the address space using a number of segment selectors.

DosReallocHuge: changes the size of the segments.

Routines are also provided to deal with the sub-allocation of segments:

DosSubSet: initialises a segment for sub-allocation.

DosSubAllocate: allocates intra-segment memory.

DosSubFree: frees a region of intra-segment memory.

3.9 Summary and Conclusions

In this chapter, the virtual memory management features of several operating systems were examined. Multics used memory-mapped files as part of its virtual memory management system. Cap was an interesting system, but the success of the Vax architecture and Unix resulted in the dominance of paged virtual memory systems over segmented architectures. In addition, interest in the use of capabilities as an integral part of a computer architecture has steadily declined [**Hennessy90**, pages 465–466].

The designers of Unix did not use an integrated memory-mapped approach. Instead, program segments are managed by the virtual memory management system while the file system uses a buffer cache located in the kernel. Thus a separate interface, based on the Unix file data structure, is needed to move data between secondary storage and main memory. This results in a degradation in performance since data must be copied from the buffer cache into user-space. It also complicates programming at the user level since the user must explicitly invoke operations to move the data.

The mmap interface was designed to overcome the shortcomings of the original Unix system and, like Pilot, uses advisory calls to inform the system about future access patterns. However, this may not be an easy exercise since many program modules may be concurrently accessing an object. Thus it is sometimes difficult, if not impossible, to be aware of the future access patterns on an object.

In addition, it is debatable whether present operating systems can fully make use of this information. For whereas it is relatively easy to lock and unlock pages in memory as well as swap segments to disk, in order to use the information on new access patterns to maximise system performance, it may be necessary to change the paging algorithm being used on a mapped entity. Since the paging algorithms are usually fixed by the operating system designer and cannot be easily changed, the corresponding gain in performance is not normally realised. In addition, some systems like BSD Unix employ a **global** algorithm which does not directly take the access patterns of individual processes into account. To maximise the use of information on future access patterns, a paging system in which objects are paged on a *per-object* basis is required.

While the VMS/VAX arrangement as well as OS/2 provide powerful memory mapping facilities they are difficult to use because the interfaces provided are concerned with the management of *pages*, in the case of VMS, and *segments* in the case of OS/2. These are architectural entities which most users would not like to manage **directly** and therefore puts a strain on the programming of complex applications.

For the interfaces described above, user processes have to manage the information about all the objects they have mapped into their address spaces. This also discourages the use of memory-mapped interfaces. In addition, it should be pointed out that memory mapping is a *low-level* abstraction in which the logical entity envisaged by the user is represented as just a linear sequence of bytes in his address space. There is no *inherent* mechanism – like the file structure in Unix – with which to *navigate* the data. Certain types of applications provide their own abstractions and routines for accessing the data, e.g. a database system. However, the ordinary user is left to build these facilities himself. This hampers the effectiveness of the memory mapping mechanism since the user is forced to think of the data in a more *physical* manner than the logical abstraction he would like to use.

The cumulative effect of these observations is that the ordinary user sees memory-mapping as a complex technique, best left to the experts, e.g. application programmers.

The development of modern operating systems, e.g. Mach and Chorus, as well as the abundance of cheap but sophisticated MMUs have resulted in renewed interest in memory-mapping techniques. However, the lack of simple, powerful user interfaces severely hinders its widespread use. This highlights the need for a more *object-oriented approach* to the design of virtual memory interfaces, where, like object-oriented languages, the user is allowed to think on a more abstract level while the programming environment handles the details of actual implementation.

These facilities must be provided without too much cost in system performance and without the need to rewrite entire operating systems. Fortunately, in the Eighties, a new approach to the design of operating systems has evolved which adopts a more modular approach. This is the subject of the following chapter.

Chapter 4

Virtual Memory Management In MicroKernel Operating Systems

4.1 Introduction

In the Sixties, operating systems were usually designed based on specific hardware architectures. This was acceptable since a large proportion of the operating system was written in the assembly language of the processor. However, with the development of system programming languages such as BCPL and C, the porting of operating systems to various architectures has become an easier proposition. Portability, therefore, is now an important issue in operating system design. In addition, with the explosion in network communications in the Seventies, the design of a network operating system with the primary goal of supporting heterogeneous machines in a loosely coupled environment is increasingly seen as a normal undertaking.

Modern operating system designs also seek to exploit parallel or concurrent applications running on multiprocessors or other tightly-coupled arrangements. A number of multiprocessor operating systems have been developed [**Ousterhout88**], and it remains a very active area of research [**McJones87**]. However, today it is undesirable to design systems without taking into account the need for compatibility with existing operating systems – notably, Unix. Without such support, one is faced with the task of writing numerous applications to encourage end-users to use the system [**Tanenbaum90**].

The need to provide the same functionality to the end-user over a wide range of diverse architectures has led to a move away from the monolithic systems of the Sixties and Seventies. Instead, many services, traditionally provided by the kernel of the operating systems, are now implemented by a set of user-level servers running on a very small kernel. The approach, called the **microkernel** approach, is becoming widely used because possible losses in performance are easily outweighed by benefits of clarity, modularity and portability.

Since most of the functionality of the system is implemented in user-space, in order to achieve good performance, it is necessary to have powerful user interfaces. Increasingly, the application of virtual memory techniques is seen as a way of achieving these goals. This chapter examines two microkernel operating systems, exploring in detail their virtual memory management systems.

4.2 MACH

Mach [**Tevanian Jr.87b**] is a portable, multiprocessor operating system developed at Carnegie-Mellon University (CMU) in the late Eighties. It is the successor to Accent [**Fitzgerald85**], a communications-oriented operating system whose main goal was the integration of a paged virtual memory management system with interprocess communication (IPC). Other goals were simple access to data using memory mapped techniques, allowing greater sharing between processes and the transfer of large objects into different address spaces. Accent was implemented as part of the CMU SPICE project and in 1985, it was supported on two hundred personal computers and commercially installed on a thousand systems. A commercially-available version of Unix system V has been built on top of the Accent kernel.

Mach incorporates the facilities developed in Accent and also provides new features. There are six basic abstractions supported by the Mach kernel:

task: an execution environment and the basic unit of system resource allocation. This includes a paged virtual address space and access to the system resources such as processors, ports, capabilities and virtual memory.

thread: the basic unit of execution. A thread executes in the virtual memory of a single task.

port: a simplex communication channel implemented as a message queue managed by the kernel.

port set: a group of ports that are treated as a logical unit. Messages for individual ports are placed on a single queue.

message: a collection of data objects used in communication between threads.

memory object: an object usually residing in secondary storage that is mapped into the address space of a task.

Message passing is the *essential* service of the system, as communication between different tasks as well as with the kernel take place by sending messages. Operations on objects are requested by sending messages to the ports which represent them. In order to use a port, it is necessary to have the proper access rights. The send access right indicates that a thread may send messages to a port while the receive access right allows a thread to dequeue messages from the port. Only one task may have receive access rights to a given port at any time but any thread in that task may use the port to receive messages.

4.2.1 IPC Mechanism

Like Accent, Mach uses virtual memory techniques to transmit large amounts of data being sent as messages. When a task sends a long message, the data is remapped into a special address space used by the kernel called the **ipc_map**. The data is also remapped copy-on-write in the sender's address space so if the sender now attempts to write to it, new pages will be generated, leaving the original message intact. When a task performs a receive operation, the data is mapped into the receiver's address space.

4.2.2 The Virtual Memory Management System

The virtual memory management system in Mach provides several facilities. A virtual address space is divided into fixed-size pages and a region in an address space is a set of contiguous pages. Some regions in an address space may be associated with memory objects which are used as its backing store. These objects are used to satisfy page-in and page-out requests on behalf of the region. Regions may also be inherited by child tasks by specifying the **inheritance** property of a region. This may be set to shared, copied or absent.

A task may protect regions in its address space by specifying a current protection level required to access a region. In addition, there is a maximum protection level associated with a given page. The interface used by Mach [**Walmer89**] is shown in Table 4.1.

vm_ allocate	Allocates a region of memory in the address space of a specified task.
vm_ deallocate	Deallocates a region of a virtual address space.
vm_read	Allows a region of the virtual memory of the specified task to be read by the calling task.
vm_write	Allows a region of the virtual memory of the specified task to be written to by the calling task.
vm_copy	Causes a region of memory in the specified task to be copied to a region in the calling task.
vm_map	Maps a region into the caller's address space specifying the memory object which will be used to satisfy pagefaults on the region.
vm_region	Obtains the characteristics of a region including, its starting address, its size, the current protection level, its maximum protection level, its inheritance attribute and the name of the memory object associated with the region.
vm_protect	Changes the protection attribute of a region.
vm_inherit	Specifies how a region in a task's address space is to be shared with a child task.
vm_statistics	Returns the kernel usage of virtual memory since the kernel was booted. This includes the number of free, active and inactive pages, the number of page-in and page-out operations and copy-on-write faults.

Table 4.1: The Memory-Mapping Inferface in Mach

memory_object_init	The kernel is requesting that the memory manager accept requests for data associated with a given object.
memory_object_data_request	Requests data for a region with the desired level of access. The memory manager should return with the specified data or inform the kernel to supply zero-filled pages for this region if required.
memory_object_data_write	Provides the memory manager with data that has been modified in the cache.
memory_object_data_unlock	Requests the memory manager to permit at least the desired access on an object.
memory_object_copy	Notifies the memory manager that a new copy of a region of an original object has been made.
memory_object_terminate	Indicates that the memory manager can deallocate its resources for that object.

Table 4.2: Kernel–Memory Manager Interface in Mach

4.2.3 External Pagers

An important feature of Mach is the interface to support **memory managers** which are user-level processes. The system has a default memory manager that is part of the kernel and is used to manage files as well as temporary objects. However, the system provides an interface between the kernel and external memory managers to carry out page-in and page-out operations as well as cache coherency mechanisms. The kernel calls that are made to the object manager are shown in Table 4.2 while calls by the memory manager to the kernel are shown in Table 4.3.

memory_object_set_attributes	Sets the attributes on an object.
memory_object_get_attributes	Retrieves the current attributes associated with the memory object.
memory_object_lock_request	Allows the memory manager to make requests concerning the management of the object cache.
memory_object_data_provided	Supplies the kernel with data requested for a specified object and an indication of the kind of access that is prohibited from using the cache. Data is copied from the memory manager into the object.
memory_object_data_unavailable	Indicates that the memory manager does not have data for a given region and that the kernel should provide the required memory.
memory_object_data_error	Indicates that the memory manager cannot return the data requested.
memory_object_destroy	Tells the kernel to shut down a memory object.
vm_set_default_memory_manager	Sets the default memory manager.

Table 4.3: The Memory Manager–Kernel Interface in Mach

4.2.4 Copy-On-Write Sharing in Mach

When objects are shared using copy-on-write attributes, special management routines are used to keep track of pages modified by individual tasks [**Tevanian Jr.87a**]. Mach creates special memory objects to hold these modified pages called **shadow objects**. The shadow object points to the original memory object where the unchanged data is held. A shadow object may itself be shadowed as a result of a subsequent copy-on-write operation creating a list or chain as shown in Figure 4.1. When the system attempts to find a page in the shadow object, it journeys up this chain until it is successful.

In Mach, if a memory object is mapped into a task and another task needs to share this memory object with copy-on-write access, e.g. by invoking the fork system call, two new memory objects are created and the original memory object is mapped read-only. These shadow objects will contain the pages of the memory object which were modified by each task while the unmodified pages remain in the original mapping. The mapping in the

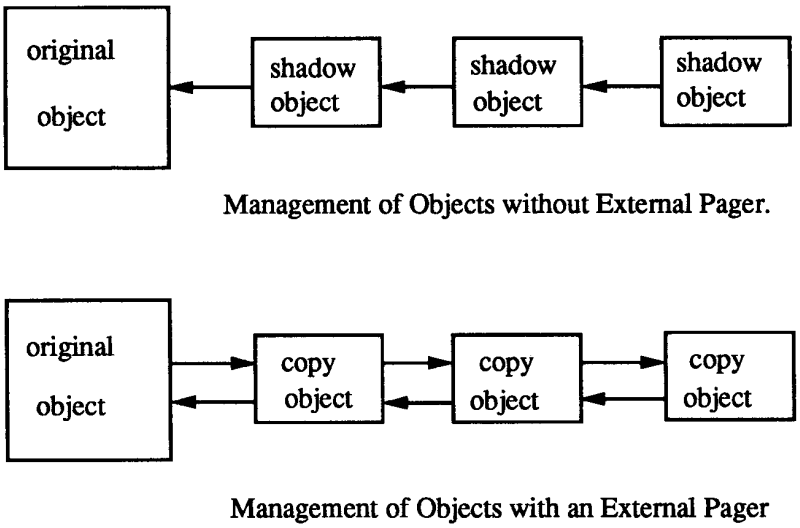

Figure 4.1: Management of Copy-On-Write Sharing in Mach

first task must therefore be changed to point to one of the shadow objects instead of the original mapping.

Since shadowing is transparent to user tasks, it is also transparent to an external pager. Thus when a page is modified in a shadow object, the external page has no knowledge of this and cannot write it to the backing store.

To solve this problem, it is necessary to create another type of object called the **copy object**. This object contains all the original pages which are managed by the external pager. When a copy-on-write fault occurs on an object with a copy object the original page is moved to the copy object. A new page is allocated and data from the original page is copied into it. Thus the external pager can access the original object as well as associated copy objects to get modified pages.

The original object may now hold modified pages since *unmodified* pages are now stored in the copy object. Thus, there is a bidirectional link between a copy object and its original. A copy object, like shadow objects, must consult the original object for pages it does not have. These pages would have been read but not yet written to; so that they would not be in the copy object.

The complexity of the Mach virtual memory management system lies in the need to prevent the buildup of large chains of shadow objects and copy objects. This can be caused by a Unix process that repeatedly forks. Mach runs an automatic garbage collection routine that collects and deletes intermediate shadow objects which it detects are no longer needed.

4.2.5 Unix File Interface

The Unix file abstraction is supported by Mach [**Tevanian Jr.87c**] using memory-mapped files. The **map_fd** call takes as its arguments: the Unix file descriptor of the open file, the offset within the file at which the mapping occurs, a pointer to the address in the address space at which the mapped file should start, a boolean variable that informs the kernel to select an unused region, and the number of bytes to be mapped in.

Mach also implements the standard Unix I/O package for the C environment using memory mapped techniques. When a file is opened, it is mapped into the caller's address space. The stdio buffer, in effect, has been enlarged to the size of the file. The copying of data using fread and fwrite routines is replaced by the **fmap** system call. So after opening a file, the fmap routine is called with arguments of the open file descriptor and the size of the data wishing to be accessed. This call returns the start of a region in the virtual memory which corresponds to the location of the buffered file contents. The results show improvements of over 20% in speed was achieved on the VAX 8650 while other systems were of 10-15% faster compared with the BSD implementation of the UNIX stdio interface [**Tevanian Jr.87c**].

4.3 Chorus

Chorus was started as a research project at INRIA in France in 1979. It was conceived as a small communications-oriented kernel or **Nucleus** with both system and application services exchanging messages via relevant ports. Presently, four versions of Chorus have been produced and key ideas from other distributed systems have been incorporated into these designs. An implementation of Unix System V, called Chorus/MIX, has been built on top of the Chorus architecture.

In Chorus, an address space is referred to as an **actor**. The kernel distinguishes between two types of actors, **system** actors and **user** actors. Some kernel services can only be executed by threads belonging to system actors and system actors may be created so that they use the kernel address space instead of a separate address space. These actors are referred to as **supervisor** actors.

A **thread** is the unit of execution and Chorus supports multi-threaded actors. A **port** is a region associated with an actor where messages for that actor are received. Any thread having knowledge of a port can send messages to it and ports may migrate from one actor to another. Ports may also be grouped together dynamically to form port groups. Actors, ports and port groups all have unique identifiers (UIs), which are global,

location-independent and unique in time and space.

The use of resources is controlled by using capabilities which are issued by different servers while actors and ports have protection identifiers which are used to authenticate incoming messages. A host is referred to as a **site** and a **node** comprises a number of sites connected together by a physical medium.

The Chorus Nucleus is divided into four major components namely:

The **Chorus Supervisor**: dispatches interrupts, traps and exceptions delivered by the hardware.

The **Chorus Real Time Executive**: controls processor allocation and provides synchronisation as well as scheduling.

The **Chorus Interprocess Communication Manager**: provides asynchronous message exchanges and remote procedure call (RPC) facilities.

The **Chorus Virtual Memory Manager**: is responsible for managing memory requirements of the system.

4.3.1 The IPC Mechanism

The IPC Mechanism is used to transport messages between different users as well as from the system and is decoupled from the Chorus memory management. The scheme uses a special segment known as the **IPC buffer** segment which is managed as a pool of fixed-size 64 KB slots. A message is therefore limited to 64 KB in length. When a message is sent, the kernel copies its contents to one of these slots and when the message is read, it is transferred from the IPC buffer to the context of the receiving thread.

4.3.2 The Virtual Memory Management System

The virtual memory management system [**Abrossimov89b**] includes a number of system calls that manipulate different regions of an actor. The interface is is shown in Table 4.4.

Chorus also has an interface that manages segments which are represented using capabilities. This is shown in Table 4.5.

Like the external pager in Mach, segments are managed outside the kernel by external servers called **mappers**. Mappers are responsible for synchronising access to, and maintaining the consistency of, segments which may be mapped into different actors on several

rgnAllocate	Allocates a new region in the actor.
rgnFree	Deallocates a region in an actor.
rgnInit	Initialises the contents of a region using part of a segment.
rgnInitFromActor	Initialises the region of an actor using a region from another actor which is specified by its capability.
rgnMap	Allocates a new region in an actor mapping parts of a segment starting from a given offset into this region.
rgnMapFromActor	Similar to rgnMap, but instead of using a segment, a memory region from a different actor is used.
vmGetPhysAddr	Returns the physical address of the virtual page.
vmLock	Locks part of an address space in memory.
vmPageSize	Returns the system pagesize.
vmPhysCap	Builds an I/O segment and returns its capability.
vmStat	Returns information about the state of the memory management system such as the amount of physical memory available.
vmUnLock	Unlocks part of the address space of an actor, allowing its physical pages to be swapped out.

Table 4.4: The Memory-Mapping Interface in Chorus

sgCopy	Copies data from one segment to another.
sgRead	Copies data from a segment to a region.
sgWrite	Copies data from a segment to a region.
sgLockInmemory	Locks a part of the segment in physical memory.
sgUnLock	Permits a fragment of a segment to be swapped out.

Table 4.5: The Segment Interface in Chorus

mpUsed	Associates a local cache with a given segment.
mpPullIn	Reads a fragment of a segment.
mpGetWriteAccess	Requests write access to a segment or region.
mpPushOut	Writes back data to a segment.
mpCreate	Creates a segment and associates with a local cache.
mpRelease	Terminates access to a given segment via the cache.

Table 4.6: The Mapper Interface in Chorus

chSync	Forces all modified blocks to be written back to secondary storage.
chInvalidate	Destroys a local cache with no write back operations on modified blocks.
chFlush	Destroys the cache but writes to modify pages back to secondary stages beforehand.
chLockInMemory	Fixes the local cache in memory and ignores flushing requests.
chUnlock	Permits flushing.
chRelease	Releases and destroys an unmapped cache.

Table 4.7: The Local Cache Control Interface in Chorus

sites. There are routines to allocate and initialise segments including copying data from another address space or another segment. The mapper calls are shown in Table 4.6 while the Local Cache Control Interface is shown in Table 4.7.

4.3.3 Copy-on-Write Mechanisms in Chorus

Like Mach, Chorus implements copy-on-write sharing techniques [**Abrossimov89a**]. However, Chorus employs a different scheme in which **history objects** are used to contain pages of the unmodified source. If an actor is required to share a given segment, called the **src**, which is currently mapped into another actor, a new segment, **cpy1**, is created. cpy1 is the descendent of src and also its history object. All the pages in the src are made read-only.

When a user attempts to write to the copy object, a write violation occurs and a new page frame is allocated for the copy. If a page has already been allocated for the src, then its contents are copied to the new page. If the page is not in the src, a pagefault occurs and the page is fetched from secondary storage and placed in the src. Its contents are then

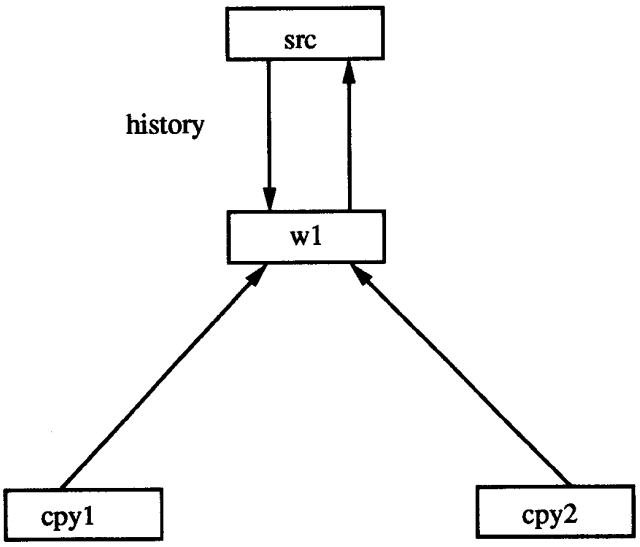

Figure 4.2: Copy-On-Write Sharing in Chorus

copied to the page allocated for the copy object.

When a write violation occurs in the src, if the copy already has allocated its own page, then it will suffice to make the page in the src cache writable by changing its page descriptor. Otherwise, a new page is allocated in the copy and the contents of the src are copied to it after which the page in the src cache is made writable.

If another actor is now required to share the source copy-on-write, two new history objects are created; another cache, which we call **cpy2**, and an intermediate or working cache, **w1**. w1 is inserted between the src and both cpy1 and cpy2 as shown in Figure 4.2. The src cache is again remapped read only. When data is modified in src, cpy1 or cpy2, the original pages are placed in w1, where it can be obtained by the other actors. A pagefault in cpy1 or cpy2 causes the w1 cache to be searched and then the src cache to be examined. If a third copy-on-write sharing call is requested of the src then another working cache **w2**, will be created and inserted between the src cache and the new copy object **cpy3**. Like Mach, a large number of inactive history objects can exist.

4.4 Summary and Conclusions

In this chapter, the features of two microkernel operating systems were examined. Both these systems support multiprocessor and multi-threaded environments and use advanced virtual management techniques. The microkernel approach is becoming the *preferred*

approach among operating system designers and a number of operating systems are being developed using this approach.

The external memory manager interface allows users to implement their own page-in and page-out mechanisms. In addition, paging algorithms and cache coherency protocols may be selected based on the type of data being accessed. This fits in well with the microkernel philosophy. However, data must be passed between the address spaces of the object manager and that of the user to service pagefaults. The efficiency of these operations will affect overall system performance. The need here is to prevent excessive copying, as seen in Mach. However, if memory objects are also mapped into the address space of the memory manager, then its address space may soon be overloaded.

Both interfaces also provide good support for copy-on-write sharing. As was shown, these mechanisms can complicate the virtual memory management. The copy-on-write mechanisms in Chorus appear to be simpler but also more efficient than those used in Mach [**Abrossimov89b**].

There is no doubt that the success of Mach and its predecessor, Accent, as well as Chorus has generated renewed interest in the memory-mapped approach since they clearly have a better overall system performance when compared with traditional Unix systems.

While the memory mapping interfaces of Mach and Chorus provide good features, they are not easy to use. Users must keep track of where different objects are mapped into their address spaces and the relevant access rights. Conceptually, the user must translate between a region in an address space, which is represented by a starting address and a length – the level at which most of the interface calls are invoked – and the logical abstraction that is required. Thus, it is not easy for users to build and manage their own abstractions. A simpler mechanism is needed.

Chapter 5

Paging Algorithms: The Need for A New Framework

5.1 A Historical Perspective

The study of paging increased with the development of multiprogramming and the time-sharing computer environment where several programs are competing for memory simultaneously. The main goal of these studies was to prevent **thrashing**. This occurs when too many programs are executing concurrently resulting in an extremely high pagefault rate. Thus the CPU spends most of its time waiting for page requests to be satisfied. Most studies sought the **optimal multiprogramming level** [**Denning75a**], of a system which kept CPU utilisation high while preventing thrashing.

The part of the disk that is reserved for moving data to and from secondary storage during paging is referred to as **swap space**. Since the movement of data between primary and secondary memory is slow, i.e. tens of milliseconds, the CPU is switched to another process while waiting for the pagefaults of other processes to be serviced. The slower the disk, the larger swap space requirement for a given level of multiprogramming. However, since a large number of users are supported on a time-sharing system, the CPU will always have jobs to run while waiting for the pagefaults of other jobs to be serviced. The optimal multiprogramming level determines the maximum number of users that could be serviced simultaneously before thrashing occurred. It is important to realise that since time-sharing

systems are dedicated to supporting a large number of users, **throughput**, measured in terms of the number of interactions completed per second, is the dominant criterion of performance.

There exists a large body of literature on paging in a time-sharing environment and a number of related issues [**Smith78a**] that affect the memory management of the system have been examined.

5.1.1 New Motivation

Time-sharing is no longer the dominant model of computing having been replaced by the workstation/personal computer environment. The approach to paging in this environment differs from the time-sharing model in several ways.

Firstly, in a workstation environment, response time is the dominant criterion of performance. The user *expects* a fast response time. This is usually translated into higher user productivity. Thadani [**Thadani81**] showed that user productivity in an interactive system is greatly affected by the response time of the system. For example, he found that user productivity, measured in interactions per user per hour, more than doubled when going from a response time of 3 seconds to a response time of 0.5 seconds on a system supporting engineers and programmers involved in manufacturing operations. He also reported a 67% increase for a more powerful system used by program developers. Other studies support these findings [**IBM82**].

Secondly, the level of multi-tasking is lower, so that there are fewer runnable tasks while the CPU is waiting for a pagefault to be serviced. Thus as CPU speeds continue to increase much faster than disk speeds, an I/O bottleneck may develop with the CPU being idle during paging requests. Hence, the corresponding increase in the response time of the system will be less and user productivity also will not increase accordingly.

One solution to the problem is to banish paging and/or virtual memory altogether. With physical memory getting cheaper, it may seem logical to assume that a machine will always have enough physical memory to run any program. However, while it is true that the cost of memory continues to fall, the average program size is also increasing. There are several reasons for this. Firstly, programs are becoming more sophisticated and contain more features and functionality than ever before. Secondly, user interfaces have also become sophisticated with graphical user interfaces (GUIs) becoming the norm. The interfaces have large memory requirements.

In addition, object-oriented languages tend to produce larger programs than those of tra-

ditional languages [**Wu**]. So with the trend towards this kind of programming increasing, program sizes will also increase. All these factors suggest that program size will continue to exceed physical memory. This highlights the need to re-examine the use of virtual memory techniques, especially paging algorithms, to provide greater system efficiency.

Another development is the use of faster auxiliary memory for swap-space. Since the level of multi-tasking is less in the workstation environment, the amount of swap space need not be very large. Thus an **intermediate storage device** which has less storage capacity than disks but a faster access time is required. A Cambridge company called Anamartic has developed an intermediate storage device based on the Wafer Scale Integration (WSI) of memory chips. Instead of making discrete chips from a wafer containing several memory chips, the chips are connected together on the wafer with the bad ones being noted and ignored. The system is interfaced to a SCSI controller and thus can be accessed by the CPU via the SCSI interface. This arrangement has been found to be about 10 times faster than the fastest disks presently on the market and would make an appropriate swap device.

An interesting solution is to use main memory on idle machines as swap space [**Felten91**]. Since network speeds are increasing, accessing main memory on a remote machine will be faster than getting data from the local disk. This involves a radical change in the management of swap space since swap space in traditional systems refers to some form of secondary storage and not primary memory located on another machine. Preliminary studies show that significant performance gains may be possible. An extension of this concept is the use of **memory servers** which primarily consist of large volumes of volatile and non-volatile memory. These servers have powerful network interfaces and are used by clients to store and retrieve pages of data.

Observations on virtual memory interfaces which support the memory mapping of individual objects indicate that access patterns of some objects may differ considerably from those of conventional program segments. Conventional paging algorithms may not be appropriate for such objects. A key issue is whether different objects exhibit the **locality of reference** phenomenon seen with program segments. In fact, some objects, e.g. files and databases [**Smith78b**] are known to have highly sequential rather than localised access patterns. This requires a facility to implement and test various paging algorithms on a **per-object** basis where different paging algorithms may be used on different objects to achieve better overall performance. A main goal is to provide a platform for the analysis of paging activity without having to modify large parts of the kernel when new algorithms are introduced.

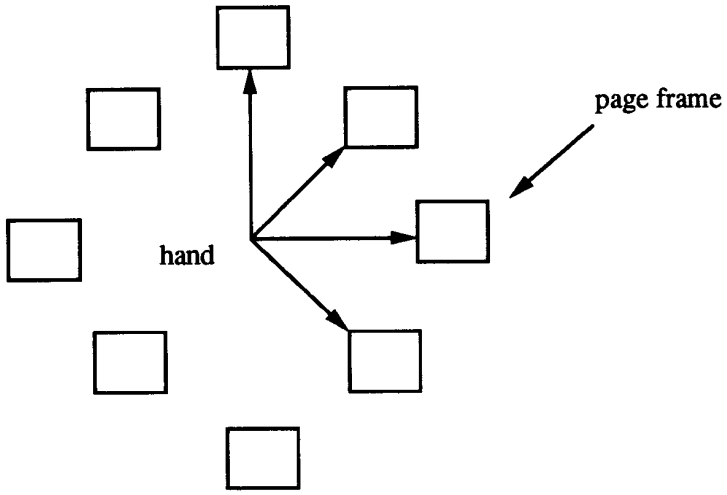

Figure 5.1: Clock Algorithm

5.2 Traditional Paging Algorithms

Paging algorithms are used to manage the allocation of physical pages to different tasks. These algorithms may be divided into two categories. **Demand paging** only allocates pages as they are required (i.e. when a pagefault occurs). Other paging systems implement some kind of **pre-paging** or **prefetching**. When a pagefault occurs the system will not only get the faulted paged but other pages of the object as well.

Demand paging algorithms concentrate on a **page replacement policy**. While fetching the faulted page, it may remove pages from the resident set to make room for this new page. The replacement policy decides which page(s) should be replaced. These algorithms can be divided into two types: **global** and **local**. Global algorithms manage all the primary memory as a single entity, while local algorithms manage the resident set of the individual tasks.

Global LRU (GLRU) is a global algorithm in which all the page descriptors in the system are kept on a linked list with the least recently used pages at the head of the queue and the most recently used at the tail. When a pagefault occurs, the system checks each page descriptor to see if the page has been referenced. If so, it is moved to the back of the queue. Global LRU has been found to be a good replacement policy but is normally very expensive to implement as the LRU list must be frequently updated.

An approximation to GLRU is the **CLOCK** algorithm [**Carr81**]. For this algorithm, all the page descriptors are linked together to form circle as shown in Figure 5.1. A designated

frame pointer moves around the circle like the hands of a clock. When a pagefault occurs the **hand** moves to the next page descriptor. The paging routine examines the used bit to determine if the page has been referenced. If so, it clears the bit and moves on to the next page descriptor until it finds a page whose used bit is not set. This page is replaceable. However, if the modified bit has been set, the page is queued to be written back to secondary storage. The routine continues until it finds a page that has not been referenced and also not modified. It uses this page to satisfy the pagefault. The CLOCK algorithm approximates to a LRU policy with significantly lower overhead.

Local paging algorithms are used to manage the resident set of a process in a more direct manner. For example, local LRU (LLRU) is similar to global LRU but is used on a fixed number of pages that have been assigned to a process. Another local algorithm is the **Working Set** (WS) algorithm [**Denning80**] in which the working set of a program is defined as the pages that have been referenced in the last θ time units of program execution. At the end of each reference, all the pages in the resident set are examined. Pages that have been not referenced within θ units are removed. If all the pages have been referenced within this time, then another page is added to the resident set. Thus the resident set size changes dynamically and sometimes drastically as the program executes. WS is an expensive policy since it must keep track of the time of the last reference for every page in the resident set and to perform such operations for every reference is impractical.

The **Sampled Working Set** (SWS) [**Ferrari83**] policy samples the resident set at fixed intervals in the virtual time of the process. At the beginning of a sampling interval, the used bit of all frames in the resident set are reset. At the end of the interval, pages that have not been referenced are removed from the resident set. If a pagefault occurs during the interval, the page is added to the resident set. The SWS policy is less expensive to implement than the WS policy since there is no need to keep information on the last time a page was referenced.

Another algorithm using the working set technique, but in a different manner, is the **VMIN** algorithm [**Prieve76**]. This is a **look-ahead algorithm** and thus cannot be realized in practice. It can however be used on program traces. On every reference, the algorithm looks ahead to the next time the corresponding page will be referenced. If this is greater than θ, the page is swapped out. In this situation, θ is usually represented by a fixed number of instructions. The comparison of WS with VMIN reveals some interesting results during transitions from one phase of the program to another. Whereas VMIN can determine that a page in the former phase is not needed and can be removed before the transition occurs, WS has no such ability. As a result, during a transition with WS, the working set expands to accommodate the new pages, while the old pages are only swapped

out θ units after the start of the transition.

The **Damped Working Set** (DWS) policy [**Smith76**] was developed to improve the WS policy during abrupt transitions. DWS is a algorithm that reduces the size of the resident set by removing the least recently used page when the pagefault rate gets larger than a given threshold. This ensures that during a transition from one phase to the next, pages of the old phase that are not referenced in the new phase will be swapped out quicker than in WS. Though it has proved effective in reducing peak resident set size, it is as computationally expensive as WS itself.

Another variable-size policy which is easier to implement than WS is the **Page Fault Frequency** (PFF) policy [**Chu76**]. When a pagefault occurs, pages may only be removed if the last pagefault occurred at θ or more time units before the present fault. If the time is less than θ, the frame is added to the resident set. If the time is greater than θ, then pages whose used bits are not set, are removed. $1/\theta$ therefore serves as a **threshold** of the pagefault fault rate below which pages may be removed from the resident set.

While it does detect transitions and adjusts well, PFF depends on pagefaults occurring frequently to work in a proper manner. For example, consider a program moving from a phase with many pages to a phase with few pages. The PFF will detect the new phase and expand its resident set, but because the transition is short and the next pagefault may occur a long time after the transition, old pages may remain in the resident set for a prolonged period. To overcome this deficiency, a time limit using interrupts has been proposed on the interpagefault interval [**Sadeh75**]. If a pagefault has not occurred and the time limit is reached, then the algorithm is invoked.

A variation of the SWS policy, known as the **Variable-Interval Sampled Working Set Policy** (VSWS) [**Ferrari83**] , has also been implemented. This algorithm is described below. The parameters L and M are the minimum and maximum durations of the sampling interval. Q is the number of pagefaults after which the used bits are scanned.

- If the virtual time since scanning the use bits $\geq$ L, then suspend the process and scan the used bits.
- At the Qth pagefault, implement the following action:-

  If the virtual time since the process was last scanned $<$ M, then the process is resumed until M.

  If the elapsed time is greater than or equal to M, then scan the used bits while processing the Qth pagefault.

Compared with SWS, it was found that VSWS was more responsive to the dynamic situations. In addition, the number of suspensions of the process in VSWS was normally less than SWS. However, obtaining a desirable value of Q for a given program can be difficult and a way of adjusting the value of Q as the program executes has been investigated [**Pizzarello89**].

Both PFF and Working Set algorithms have been shown to exhibit anomalous behaviour in a number of cases [**Franklin78**], specifically:

The pagefault rate increases even though the mean resident set size increases.

The pagefault rate increases when the control parameter of the algorithm is also increased.

The mean resident set size decreases even though the control parameter of the paging algorithm is increased.

These anomalies are noted more frequently for PFF than WS [**Ferrari83**]. PFF is also said to be more sensitive to variations in its control parameter than WS [**Denning80**].

5.2.1 Combination of Local and Global Policies

Overall, global paging policies are easier to implement. However, they are more prone to thrashing and are difficult to analyse. Local paging policies tend to be more expensive, but are also more robust. An algorithm known as the **WSClock** algorithm [**Carr84**] tries to obtain the benefits of both types of policies.

This algorithm uses the general Clock arrangement described earlier. However, each frame is associated with an owning task. When a pagefault occurs the frame pointer is advanced to point to the next frame. If the used bit is set, then the time of last reference, (LR), for the page, p, is set to the task's virtual time (VT). If the used bit is not set and $VT - LR(p) \geq \theta$, then the page is removed from the resident set. A page is replaceable if it is not part of the resident set or the task concerned is not active. If the replaceable page is dirty, it is queued to be written back to disk and the algorithm continues until a clean, replaceable page is found. It is claimed that WSClock performs as good as pure WS with much less overhead.

5.3 Analytical Paging Models

Paging models can generally be divided into two broad categories. There are models which try to model program behaviour, i.e. how an individual program behaves, while other models attempt to model entire computer systems. We will concentrate on the former, since it is more relevant and is not tied to a given model of computing. It should be stressed however, that to accomplish the latter, one must also model program behaviour.

5.3.1 Reference String Models

Since program traces are expensive to generate, both in terms of the CPU time as well as requiring a large amount of storage, another way of representing the behaviour of real programs was sought. The **reference string model** is a sequence of numbers derived from the program trace which gives the number of the page being referenced at time t.

A number of models have been proposed using reference strings. The simplest is the **Independent Reference Model** (IRM) which regards the reference string as a strict-sense stationary process, i.e. the distribution of the variable does not vary with time. Thus for all t, the probability that the address referenced at time t, $r(t)$, is on page i is a_i, as shown in Equation 5.1.

$$P_r[r(t) = i] = a_i \tag{5.1}$$

A more sophisticated model [**Spirn76**] is the **LRU Stack Model** (LRUSM) as shown in Figure 5.2. The LRU stack, s_t, is an ordering of all the pages in a program by how recently they have been accessed. The pages are arranged in a stack with the most recently used page residing at the top of the stack. When a page is referenced at time t, it is taken from its position and placed at the top of the stack. The distance of a page from the top of the stack serves as an indication of how recently a page has been referenced. The distance probability, a_i, is the probability that the next reference will come from the page located at the i^{th} position in the stack. The occurrence of locality leads us to expect that distance probabilities would decrease as i increases, thus:

$$a_1 \geq a_2 \geq a_3 ..... \geq a_n \tag{5.2}$$

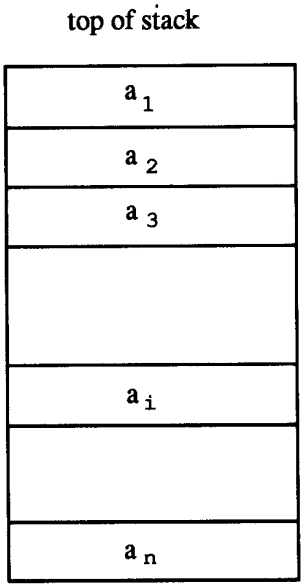

Figure 5.2: The LRU Stack Model

While this model holds true when the program is in a phase, it is not valid when the program is in transition between two successive phases. An additional component is required to handle phase transitions. A number of Markovian models have been put forward to address this issue [**Shedler72**].

Another problem is that this model assumes that the values of the different a_i's are independent of each other. However, for real programs this is not so since locality *implies* clustering. Using LRUSM, Spirn [**Spirn77**] asserted that the pagefault rate under LRU is lower than that for a dynamic algorithm having the same average memory size. He claimed that the problem with the performance of LRU was that while the size of the favoured locality varies with time, in LRU this is assumed to be fixed. Dynamic algorithms (such as WS) perform better than LRU since they are able to track changes in program locality.

5.3.2 Phase Transition Models

Phase transition models attempt to divide program behaviour into distinct states. These states form a macromodel with IRM or LRUSM being used as the micromodel. Kahn [**Kahn76**] explored a two-state macromodel which contained a phase state and a transition state. When the program was in the phase state, locality of reference was assumed to be dominant while in the transition state it was assumed to be less influential.

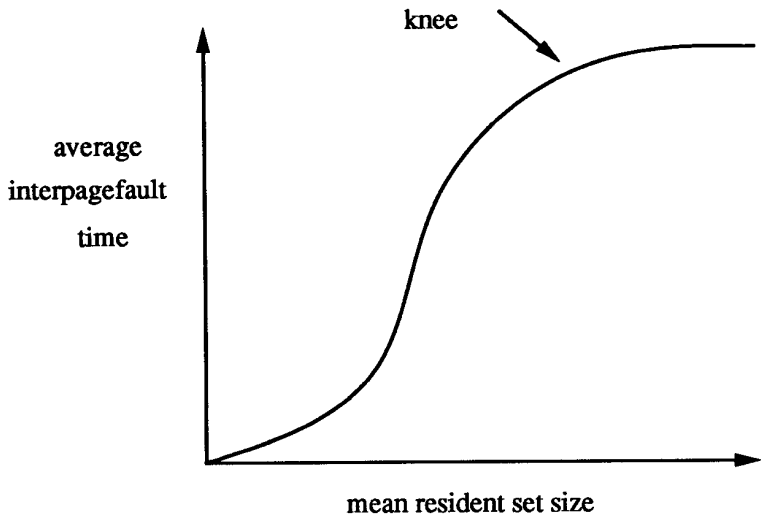

Figure 5.3: The lifetime curve

The most important issue with phase transition models is which of the two models dominates the overall model. Kahn's work indicates that the macromodel dominates [**Denning80**, pages 71–72]. His results showed that variable-size algorithms, e.g. WS, were better than fixed-size algorithms such as LRU.

5.3.3 Markov Models

A few Markov models of program behaviour have been proposed. Sekino [**Sekino72**] employed a first-order Markov chain to model program behaviour in a multiprogramming system. Each page in the program was represented as a state in the Markov chain. The transition matrix contained elements p_{ij} which represented the probability of the next reference being on page j given that the current reference was found on page i.

Franklin and Gupta [**Franklin74**] also used a first order Markov chain to calculate the pagefault probability for a general system using different memory sizes and different paging algorithms.

5.3.4 Lifetime Curves

Perhaps the most frequently used model of program behaviour is the **lifetime curve** as shown in Figure 5.3. This is a graph that gives the mean time between pagefaults $L(m)$

against the mean resident set size m, and a constant value denoted by a. One feature of these curves is the existence of a primary knee [**Denning75b**].

Various formulae have been put forward to explain the knee of the curve, including:

$$L(m) = am^k \tag{5.3}$$

where k is dependent on the locality of the program and is normally in the range of 1.5 to 3 [**Belady69**].

$$L(m) = a2^{km} \tag{5.4}$$

used by Alderson et al. [**Alderson72**].

$$L(m) = a/[1 + (b/m)^2] \tag{5.5}$$

where b is the number of page frames that provides the process with half of its largest possible lifetime. This curve gives the desired convex-concave shape and was used by Chamberlain et al. [**Chamberlain73**].

From the concept of locality of reference, the lifetime curve has a logical shape. When the mean size of the resident set is small, numerous pagefaults occur so that the interpagefault time is small. As the number of allocated pages increases, the interpagefault time also increases. However at a critical point, the resident set size is large enough to hold the entire working sets of *most* phases of the program. Thus further increases in the size of the resident set do not result in significant increases in the interpagefault time, hence the knee.

The knee of the curve is also associated with the optimal programming level of the program since it represents the point at which the program is able to make maximum use of the pages that are assigned to it. Attempts to keep programs operating at the knee of the curve and thus achieve maximum utilisation of the overall memory have been reported [**Denning76**].

5.3.5 Criticisms of the Lifetime Curve

There have been a number of criticisms about the lifetime curve model [**Carr84**, pages 47-48]. First, some of the curves have been produced using page sizes that are *not* representative of page sizes used in real operating systems. For example, models by Kahn [**Kahn76**] and Simon [**Simon79**] depend on lifetime measurements which use a pagesize of 64 words (i.e. 128 bytes).

Another issue is the extent to which the position of the knee is dependent on the execution time of the program. The same programs, if run for longer or shorter intervals, would produce different lifetime curves. Perhaps the most important issue is the inclusion of initial pagefaults (i.e the first time a page is referenced) in lifetime curves. Most of these faults occur when the program is in its initial phases. For a demand paging policy, the number of these faults are fixed for each program and are *independent* of the replacement algorithm that is being used. These faults should therefore be eliminated from results for page replacement policies. Carr [**Carr84**, pages 47-48] claims that if this were done, then lifetime curves would be kneeless.

In addition, the application of particular lifetime curves to a general program model is questionable. This is because the parameters of the lifetime curve (i.e the mean inter-pagefault time and the mean resident set size) are very program specific. It is difficult to see the paging activity of small interactive programs as well as large computational ones having similar lifetime curves. Since lifetime curves are used both in simulation and analytical models, it is necessary to gauge how well they represent the behaviour of different programs. The obvious attraction of lifetime curves is that they can be easily generated from a program trace and thus are derived from real data.

General Criticisms of Paging Models

The usefulness of the results obtained from paging models to the operating system designer has also been an issue of great debate within the computer research community. As outlined by Saltzer [**Saltzer76**], paging models sometimes contain assumptions about operating systems that are not valid. Also, issues that are of interest to the operating system designer may be ignored in paging models. For example, while designers of new paging algorithms report results which clearly show improvements in the overall system efficiency, they do not measure the computational cost of implementing a given algorithm. To the operating system designer, this is a key issue that is often neglected.

In the context of memory management, traditional paging models have not provided suf-

ficient data on the demands of paging activity on the primary and secondary storage systems. For example, transitions from one phase to the next play an important part in the overall system performance. These transitions, though short, can generate an extremely high pagefault rate. Paging algorithms, in an attempt to detect and remove old pages during or just after a transition, will increase the demands on the memory hierarchy at these critical moments which may result in the swapping device being saturated. Though the benefits of paging algorithms are given in terms of the mean interpagefault time and mean resident set size, it is also necessary to give the amount of deviation or **jitter** of the resident set size that different paging algorithms generate to get a better estimation of the demands being placed on the system.

In any virtual memory system, a large amount of effort is spent moving data between primary and secondary storage as pages are moved in and out of resident sets. For good overall performance, it is essential that a high throughput is maintained. An attempt to improve this throughput lies behind increases in the pagesize of modern operating systems. Today 4 KB and 8 KB pagesizes are quickly becoming the norm. Paging algorithms also play an important part since they determine the *rate* at which paging traffic is generated and thus must be considered.

5.3.6 Simulation Models

A number of simulation models of paging behaviour have been attempted. The main problem with the use of simulation is the large amount of computational time required to get a reasonable result. The other problem is deciding the parameters of the model that will drive the simulation. Some simulation models use the lifetime curve to generate pagefault events for objects having a certain resident size. A number of efforts in the simulation of program behaviour are briefly described below.

Alderson et al [**Alderson72**] used the lifetime curve in Equation 5.4 to compare paging algorithms as well as load strategies. Grit [**Grit77**] used a simple two-program model with global LRU. He used a stochastic stack distance model to generate memory references. Masuda [**Masuda77**] modelled an entire computer system. He employed a phase-transition model to represent overall program behaviour. The program behaviour in each phase was simulated using a stack distance model. Gomaa [**Gomaa79**] modelled the IBM VM/370 system using different models of program behaviour. First he used the lifetime curve suggested by Belady in Equation 5.3, then another lifetime curve and finally he assumed that paging rates of the program are precisely known. He concluded that lifetimes curves are very crude models of program behaviour.

A detailed simulation of paging models was carried out by Carr [**Carr84**, pages 65–100]. In this model, Carr attempted to simulate in detail various aspects of a computer system including main memory, the I/O subsystem and the task models. The task model dealt with program execution and Carr used an **Inter-Reference Interval Model** or IRIM [**Carr84**, pages 77–83] to represent program behaviour. The IRIM model was used to determine the resident set of a program at any point in the simulation. To simulate locality of reference, IRIM divides all the pages into **busy** and **idle** subsets, the busy set being those pages referenced at least every ω references where ω is a control parameter, similar to θ for WS.

Thus, if this technique is applied to a program trace, the resulting IRIM string can be used to explicitly represent the behaviour of the program. At the operating system level, Carr simulated the scheduler, virtual memory management, including different page replacement policies, and the swapping to and from disk. Carr used the simulation to examine different page replacement policies for various programs including:

LKED: the IBM 370 Linkage Editor.

SYNCSORT: a sorting routine.

FORC: the IBM Fortran H compiler. The compiler was compiling 210 statements using the highest optimisation level.

WATF: the University of Waterloo Fortran IV compiler. The compiler was measured compiling 1800 statements.

ASMC: the IBM Assembler H. It is measured while translating a 575 statement program into 370 Assembler Language.

SCRP: the University of Waterloo Script text processor.

PASC: the Stanford Pascal compiler compiling a 1100 line section of its source code.

DRAW: a graphics utility written in Fortran.

Carr obtained lifetime curves based on different paging algorithms including LRU, WS and VMIN algorithms. His results suggest that traditional lifetime curves do not adequately represent program behaviour.

5.3.7 Empirical Results

Perhaps the most disturbing thing about research into paging activity and virtual memory systems is the *lack* of data on the behaviour of real programs running on common operating

systems. This lack of real data brings into question the validity of many analytical models that have been developed and has, in turn, led to operating system designers ignoring the results of these studies.

Coffman et al. [**Coffman68**] examined a number of programs running on the IBM System/360 Model 50 computer including a **WATFOR** Fortran compiler and a program for computing Fourier transforms. These programs were used to study the performance of two algorithms: local LRU and the **Belady Optimum Replacement** or BOR algorithm. The BOR algorithm is based on a knowledge of the program execution cycle and finds the optimal resident set that minimises the degree of swapping during program execution. Data on the performance of these algorithms using pagesizes of 64, 256 and 1024 words were obtained.

Rodriguez-Rosell [**Rodriguez-Rosell73**] published experimental data on the Working Set algorithm employed on the IBM System/360. Results were presented for the University of Waterloo's G-level assembler as it assembled a program of about 300 instructions long. The mean working set size and its standard deviation as a function of θ were published. The **reentry rate**, the number of pages entering the resident set per unit time, was also examined. These results were published for page sizes from 1 to 8K. It is interesting to quote part of the conclusion of this paper:

> It is hoped that experimental computer scientists will provide similar measurements to compare and ascertain patterns in program behaviour.

Boyse [**Boyse74**] presented data for the IBM Fortran IV compiler as well as a line editor program. He noted the lack of empirical data and its effect on the modelling on program behaviour. Saltzer [**Saltzer76**] also highlighted the need for a more experimental approach to the modelling of program behaviour.

5.4 Summary: New Framework and New Tools

The workstation/personal computer environment, rather than the time-sharing environment, is now the dominant model of computing. In this environment, the response time is taken as the most important criterion of performance. Reduced multi-tasking and faster CPUs are making this environment more susceptible to thrashing.

While this issue may be partially addressed by advances in hardware, including the development of sophisticated MMU's as discussed in Chapter 2, or the use of faster devices as

swap space, a study of paging activity with the aim of devising new techniques and new paging algorithms that will improve system performance is required.

In order to achieve this goal, it is necessary to develop paging models that adequately reflect all aspects of program behaviour based on experimental data from modern operating systems. This in turn requires a proper framework in which paging activity may be analysed that isolates the essential characteristics of paging behaviour and their effects on overall system performance. A testbed must therefore be developed which can monitor the paging activity of objects which are accessed as programs are executed. For greater efficiency the use of paging algorithms on a per-object basis should be investigated. Thus different objects may be paged using different paging algorithms.

An object-oriented virtual memory system that supports memory-mapped objects is a basic requirement of this plan. The design of such a system is the subject of the next chapter.

Chapter 6

The Design of an Object-Oriented Virtual Memory Interface

6.1 Motivation

As stated in the previous chapters, there is a need for a more flexible approach to virtual memory management which provides users with the ability to use large amounts of data in an **object-oriented** manner. The motivation behind this approach is similar to that behind the use of object-oriented languages in which the user is allowed to define entities called **objects** and specify the operations that may be performed on them. This allows the programmer to think about the object using abstractions which reflect how the object is used while reducing the need to worry about the details of the implementation.

In this chapter, the design issues are first discussed. Next the user interface is specified. The concept of an object manager is then introduced and its interface is outlined.

6.2 The Design Issues

In discussing the design of any system it is necessary to define the terminology that will be used. An **object** is defined as a logical entity that can be mapped into an address space. There may be certain operations that are associated with different objects which we call

object routines or methods. The data abstraction that the user invokes to access the data within an object is called a **handle**.

6.2.1 Typed Objects

The introduction of the concept of an **object type** seems to be a *natural* way to embody the concept of objects as logical entities. Instances of different logical abstractions may be viewed as objects of different types. A process manipulates objects by mapping them into its address space and invoking operations on them by using the relevant handles. A handle may be associated with an object of a given type. For example, if an object mapped into the a user's address space is of type *file*, then the associated handle may be called a **file handle**.

In order to use memory mapping effectively, it is necessary for a process to know about objects in its address space including how and where they are mapped. In most systems, this information is kept in the **process map** for that process which resides in the kernel and is normally inaccessible to the user. Such information should also be readily available to the process. However, a key issue involved here is how to protect such a facility from erroneous or even malicious use by user-level processes.

An interesting solution to this problem is to **protect** the process map using virtual memory techniques. Thus the relevant process map is mapped **read-only** in the address space of the process. This allows the process to easily read its process map, if required, but not to modify it except with the authorisation of the system, thus protecting it against malicious use. Each entry in the table, known as a **map_entry**, contains the name of the object, the type of object, the starting address and length of the object, the access rights of the user, the index or **virid** of the map_entry in the process map and the number of threads accessing the object in the same address space. The structure is shown in Figure 6.1. This is a simple mechanism since the process map is protected by the virtual memory system.

6.3 The User Interface

The interface must provide at least a minimal set of facilities: the ability to map and unmap objects in an address space, to create and delete objects, and to flush updates to disk. The ability to grow or shrink objects is also needed. However, the issue of synchronisation as well as the ability to lock objects into memory for better performance are also addressed.

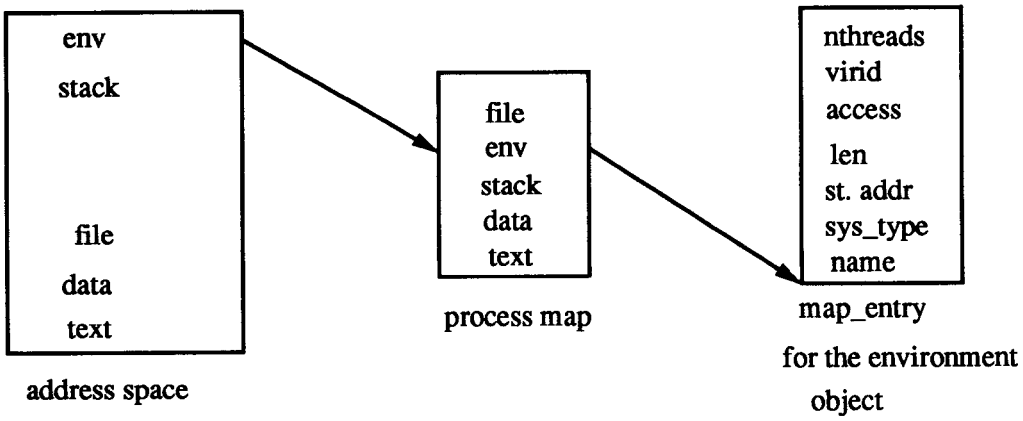

Figure 6.1: The process map

We now define a prototype interface. The following calls are provided:

- **location = GetMyProcTable**: gets the process map for that process and maps it read-only into the address space returning its *location.*
- **vir-id = MapObject(type, name, access)**: maps the object specified by the *name* and *type* into the caller's address space. The *vir-id* represents an index into the process table which contains the map_entry information. The *access* argument specifies the access rights desired by the user. The *type* argument is used by the system to contact another user process that will manage the object. This is expanded on later in the chapter.
- **result = UnmapObject(vir-id, flen)**: unmaps the object specified by the *vir-id* from the caller's address space. *Flen* specifies the final size of the object while *result* is set to 0 if successful and -1 if failure occurs.
- **vir-id = Create(type, name, access, size)**: similar to the MapObject call but this call specifies a given *size* with which the object should be created. In addition, the address space that created the object is designated the **owner** of the object.
- **result = Destroy(vir-id)**: destroys an object that has been created. The invoker of this call must be part of the address space that owns the object. The object is reference-counted and it is destroyed once its reference-count is zero. The object cannot be reused.
- **result = Flush(vir-id, flen)**: flushes the object to backing store. *Flen* again denotes the final size of the object. This system checks for all the modified pages, then marks them read-only and queues them to be written out. If however, a user

attempts to write to one of these pages before the write-out is finished, he is blocked until it is completed.

- **result = Discard(vir-id)**: prevents all modifications to the object made on the local machine from being written to backing store.

- **new_size = Extend(vir-id, new_size, reloc)**: increases the size of an object to *new_size*. If new_size is less than the current size, the size of the object is shrunk if no other thread is sharing the object. The *reloc* variable indicates whether the kernel can relocate the object in the user's address space if it cannot be extended at its present location.

- **result = LockInMem(vir-id)**: disables paging on an object. This call was included to support time-critical situations where system resources may be in use for long intervals. For example, when copying a large amount of data from an object to network buffers, locking the object in memory before proceeding will improve overall system performance since these buffers will be freed in a shorter time interval.

- **result = UnlockInMem(vir-id)**: causes the paging algorithm for that object to be invoked on future pagefaults.

- **result = AcquireLock(vir-id, offset, len, which)**: it was decided to support **many-readers-one-writer** synchronisation with one write lock being associated with the entire object. The call attempts to acquire a **read** lock or the **write** for an object. This is indicated by the value of the *which* argument. It is acknowledged that objects such as databases will require locking mechanisms of much finer granularity, so it was decided to include the offset and the length of the region for which locking is sought. By extending the design, it is possible to pass this information to a **lock manager**.

- **result = ReleaseLock(vir-id, which)**: releases the lock specified by *which* on an object given by *vir-id*.

- **sys_type = GetObjType(name)**: allows users to find out if an object type given by name is supported by the system. If the object type is supported, then the system associates a number with this object type. The value of this number is returned in *sys_type*. If the object type is not supported, sys_type will have a value of -1.

In the MapObject and Create calls, the access desired by the caller must be specified. The access variable in these calls contains numerous bit fields that instruct the system to perform certain operations on objects:

MAP_TYPETEXT or MAP_TYPEDATA – specifies the type of information being mapped whether text or data. This indication is useful for debugging purposes.

MAP_GROWUP, MAP_GROWNOT or MAP_GROWDOWN – indicates the direction in which the object is expected to grow.

MAP_READONLY, MAP_READWRITE or MAP_COPYONWRITE - specifies the type of access desired, whether read/write, read-only or copy-on-write.

MAP_ZEROFILL – indicates that an object must be supplied with zero-filled pages from the system when a pagefault occurs.

MAP_SAVE – indicates that the object should be saved when it is no longer in use. This will ensure fast restarts if a process must be executed again without any modification.

MAP_DISCARD – indicates that changes made to this object cannot be flushed to disk.

MAP_MEMLOCK – this bit field informs the system that the caller wants the object initially locked in memory.

MAP_RECORD – the record indicator tells the system to record information about the pagefaults associated with the object.

MAP_INPLACE – indicates that an object must be mapped into an address space at a specified virtual address. If this is not possible the call fails.

6.3.1 Other Issues

The MapObject call maps an entire object into the user's address space. However, it would also be a useful feature to map parts of an object as well. For example, a user may desire to examine certain parts of a file. In addition, sometimes objects must be mapped at specific addresses in an address space, for example, when starting a new process. It should also be possible to load data into an object from an external source, e.g. a list of physical pages.

To solve these problems, a structure that associates a physical segment with an object and indicates the part of the object to which the mapping is related was introduced. So when a pagefault occurs on an object, the exception routine checks the physical segment to see if it contains the page.

These requirements detailed above also require an additional call to be added to the interface:-

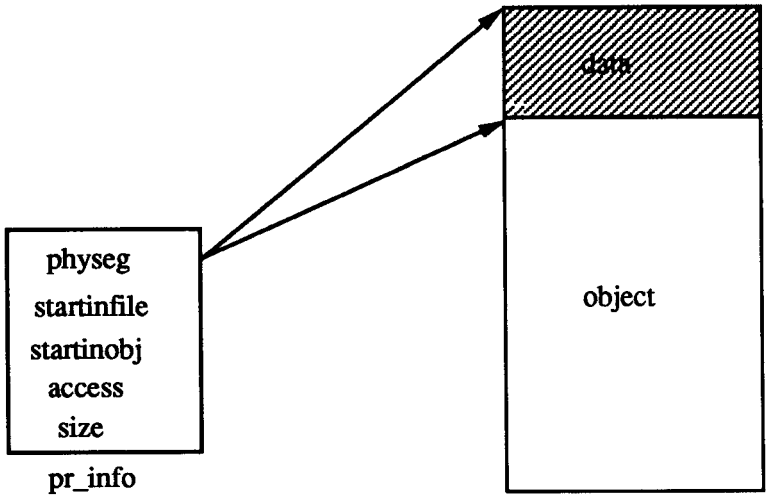

Figure 6.2: The CreateProcObject Call

vir-id = CreateProcObject(asid, type, name, pr_info)

where: `asid` is the address space into which the object should be mapped.
`type` is the type of object being mapped.
`name` is the name of the object.
`pr_info` is a structure containing the relevant information.

The pr_info structure contains the following fields:-

size: indicates the size of the object being created or mapped.

access: the access required.

startinobj: indicates where the object should start in the address space.

startinfile: the starting address in the remote file associated with the mapping.

pid: the physical segment associated with this object.

In addition, another bit field was added to the access variable. The **MAP_PRELOAD** field indicates whether the caller wants the data in the physical segment to be loaded into the object before it is accessed as shown in Figure 6.2.

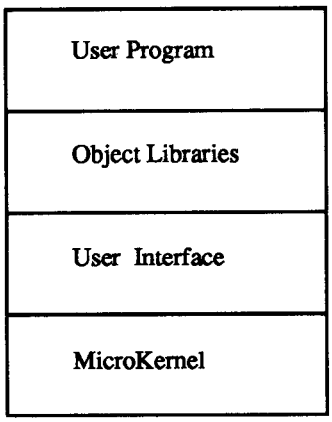

Figure 6.3: The Architecture

6.4 The Architecture

The user interface specified above would normally be implemented as system calls from user-space or as messages exchanged between the user's address space and the address space of the kernel.

It is also important to note that the kernel does not have any knowledge of the data structures used to implement the handle associated with a given object. The user is therefore free to implement any handle that is appropriate. However, in a working environment it is likely that there will be a number of *installed* handles which manage objects that are frequently used, e.g. files.

The routines associated with these handles may be kept in a number of object libraries which are linked with the user's program. These libraries are similar to the class libraries associated with object-oriented languages, for example, C++. Like class libraries, these routines may be written by other programmers and are merely invoked by the user. These routines will in turn invoke calls in the user interface to map and manage objects. The setup is shown in Figure 6.3.

6.5 Building An Object Library : A Simple Example

In this section, routines associated with the management of files are implemented and may form part of an object library for managing file objects.

Let us propose a simple file data structure as follows:-

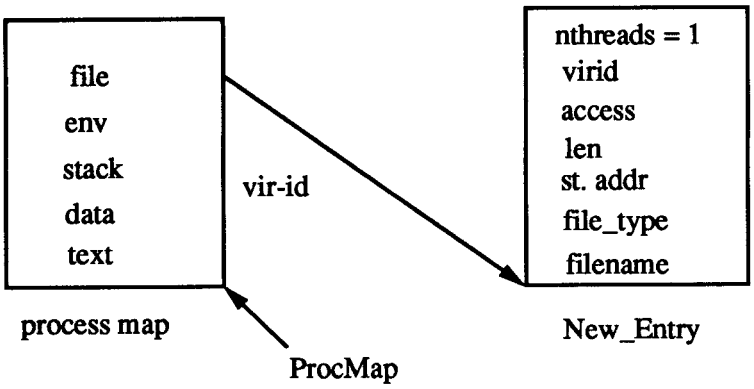

Figure 6.4: ProcMap after the MapObject call

```
typedef struct file {
        int _cnt;           - the number of bytes to be accessed.
        char *_ptr;         - next character to be accessed.
        char *_base;        - the base address of the file.
        short _flag;        - the access control indicator.
        short _file;        - the file number.
} FILE;
```

When a process begins execution, it invokes the GetMyProcTable call to map the process table into its address space. The pointer to this table is given by **Proc_Map**.

We now detail the implementation of the **fopen** Unix call
as part of an object library.

fopen(filename, mode);

From the mode we get the access variable, whether read/write or read-only, and can invoke the MapObject call in the user interface to map the object into the user's address space.

vir-id = MapObject("file", filename, access);

This mapping will be indicated by a new map_entry in the process map.

New_Entry is a pointer to a map_entry and is set to point at the file that

has just been mapped in as shown in Figure 6.4.

New_Entry = &Proc_Map[vir-id];

We now initialise the file data structure.

fp->_cnt = New_Entry->len;

fp->_base = fp->_ptr = New_Entry->st.addr;

fp->_flag = New_Entry->access;

Finally, we set the file number to point to the index of the new entry in the process map.

fp->_file = New_Entry->virid;

Routines similar to those used for the Unix file system can be developed based on this file data structure.

We can also include routines for the file object library that are not found in the Unix File Interfa

LockFileInMem(fp) = LockInMem(fp->_file);

UnlockFileInMem(fp) = UnlockInMem(fp->_file);

GetReadLock(fp) = AcquireLock(fp->_file,0,0,READ_LOCK);

GetWriteLock(fp) = AcquireLock(fp->_file,0,0,WRITE_LOCK);

ReleaseReadLock(fp) = ReleaseLock(fp->_file,READ_LOCK);

ReleaseWriteLock(fp) = ReleaseLock(fp->_file,WRITE_LOCK);

6.5.1 Benefits

There are several benefits to using this interface as compared with others presented so far. First, the interface refers to objects as logical entities and does not deal in terms of pages and segments or other architectural features. This helps the user to realise his

logical abstraction without having to be aware of the architecture of the system. Secondly, it uses a simple protection mechanism, i.e. the virtual memory system, to protect against changes in the process map.

The interface also frees the programmer of having to manage the information on memory mapped objects. This information is readily available by reading the process map. In addition, since the user must use the map_entries in the process map to invoke calls on the interface, the system can easily check his access capabilities.

6.6 Using Different Handles

Sometimes it may be necessary to use a different handle on an object other than the one normally associated with its type. This may be as a result of an object being composed of smaller objects each with a different handle, or there may be a better handle to navigate the data at a certain point. Thus, it is necessary to build a **type** interface that can be used to specify the relationship between handles associated with the different object types.

Four relationships may be declared:

SYSTYPE: a handle is of this type if its object type is known to the system. This means that the *GetObjType* call will return a valid *sys_type* number.

ALIAS: indicates that a handle **x** can be substituted for a handle **y**.

SUBTYPE: indicates that a handle **x** is a subtype of handle **y**.

UNIVERSAL: indicates that a handle **x** is a subtype of every other handle. However, to limit the amount of book-keeping required, it was decided not to allow subtypes of a universal handle.

6.6.1 The Type Interface

The interface supports two interface calls, namely:

- **Declare (a,b,rel)**

  This call declares the relationship between handles of types **a** and **b** where **rel** is one of the relationships above. For relationships of types SYSTYPE and UNIVERSAL, **a** = **b**.

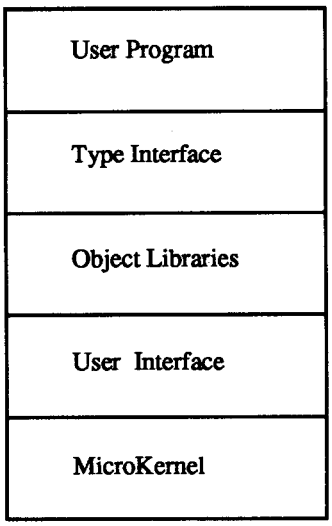

Figure 6.5: The Modified Architecture

- **pseudo_entry = GetPrimHandle(vir-id, start_addr, len, p)**

  A pseudo_entry is a pointer to a map_entry structure that is not part of the process map. This call is used to associate a pseudo_entry with an object of type p, starting at *start_addr* and of length *len* in another object which is represented by the map_entry specified by the index *vir-id*. The type-checking routine checks to see if there is a declared relationship between *p* and the object. It then checks that the starting and ending addresses of the new object are within the larger object.

6.6.2 A New Layer

The type interface forms part of the user's environment and should be implemented as a library which is linked along with the user's program. It may be represented by a layer between the user and the object libraries as shown in Figure 6.5.

6.6.3 A Brief Example

Let us imagine we would like to read our mail which is located on a remote fileserver. A mail message has both a header and data which is the actual message. The size of the data is given in the mail header. We would like to invoke routines to read the mail on our machine which has an installed mail handle.

We can define the handle as follows:

```
typedef struct message_reader {
        int n;              - the number of lines read.
        char *newline;      - a pointer to the first character in the new line.
        int totalcount;     - the total number of bytes in the message.
        control_chars;      - control characters.
        char mh;            - to use the process map.
} MessReader;
```

There is also a routine to set up the MessReader from a map_entry.

We define this routine as follows:

```
MessReader *SetMessReadHandle(New_Entry){
        mh = (MessReader *) malloc(sizeof(MessReader));
        mh->n = 0;
        mh->newline = New_Entry->st.addr;
        mh->totalcount = New_Entry->len;
        mh->mh = New_Entry->virid;
};
```

We first declare the MessReader type to be a subset of the type file.

```
Declare(MessReader,file, SUBTYPE)
```

We now open the file with the mail message get back the file descriptor. Using the file descriptor, we read the mail header to obtain offset in the file at which the data starts and length of the message. We then invoke the GetPrimHandle call to get a pseudo_entry representing the message.

```
pseudo_entry = GetPrimHandle(fp->_file,(fp->_base + offset),len,MessReader)
```

We now call the SetMessageReadHandler routine with pseudo_entry as the argument. The **mh** returned can be used to read the message.

There are several advantages to this scheme. First, a handle is implemented as though the system directly supports objects of its type. It is the user's responsibility to use the type

interface to obtain the relevant handle at any point in time. Secondly, there is a small amount of code required to integrate a new handle into the system. All that is required is for the user to specify how to set up the handle from a map_entry. Once this is done, it is easy to use the handle either as a subtype of another handle or an alias to another handle. However, it is important to note that each handle data structure has a variable which is set to the index (i.e. New_Entry->vir-id) of the map_entry in the process table to which it is related. This allows the user to invoke the user interface directly.

There are limitations: the most severe being that the type checking to ensure that handles are properly related is done at run time. It would be much better to do this at *compile time* so the user can quickly correct any errors without having to run the program. Secondly, this interface would greatly benefit from facilities of an object-oriented language like C++. This would make it easier to define new interfaces using the inheritance facilities of the language.

6.7 Object Managers

To manage the paging of objects, additional support is needed. In keeping with our microkernel philosophy, it was decided that these functions should be done by user-level processes which we call **object managers**. An object manager is responsible for one or more object types. Its main task is to move data between an object and its backing store.

One of the major issues in the design of object managers is the need to minimise overhead with regard to moving data from the object manager to the relevant process. For example, it is essential to eliminate the need to copy data involved in servicing a pagefault from a buffer in the object manager to the page where the data should go since this limits the throughput of the system. One way of achieving this is to map the object into the address space of the object manager as well. This, however, places a critical load on the address space of the object manager and complicates the virtual memory management system.

In this design, a simpler mechanism is employed. All the physical pages that are on the free-list after the kernel has been initialised are mapped into the address space of an object manager. Each free page has an index relative to the first free page in the system. These pages are initially mapped *non-accessible*. The page data structure for every page contains an index relative to the first free page. When a pagefault occurs a free page is taken off the list and made accessible to the object manager. Given the index of the page, the object manager locates the page in its address space and writes the required data directly into it. When the object manager is finished, the page is again made non-accessible to the object

manager as it is mapped into the virtual address space of the object.

Object managers *register* with the kernel, stating their willingness to handle events on objects of a given type. When events, such as pagefaults occur, they are directed to the respective object manager.

6.7.1 Object Manager Interface

The interface for the object manager must support a number of functions. These include the ability to map the free pagelist into its address space, to register with the kernel about managing an object of a given type, to get the relevant characteristics of the object from the remote fileserver as the object is being mapped into an address space and to get the relevant data from the system to service pagefaults and return the results. The interface calls are given below:

- **location = MapFreePgTable**: maps all the free page data structures into the address space of the object manager returning the virtual address.
- **sys_type = RegObjHandler(objname)**: informs the kernel that the caller is willing to be the object manager of objects of the type given by *objname*. The kernel returns a number which is then associated with objects of that type.
- **result = ConstructPageTable (obj_num, size)**: invokes this call to construct the pagetables for the object specified by the *obj_num* whose size is given by *size*. This is usually done after the object manager has obtained the size of the object from a remote server.
- **result = Investigate (obj_num, os)**: the object manager uses this call to obtain information about the object specified by obj_num. The *os* argument points to a structure that contains the required information for the object manager to handle the pagefault from its own address space.
- **result = ReturnResult (obj-num, os)**: returns the result of an operation that the object manager has initiated in response to an event on an object.

6.7.2 Using Default Object Managers

In a distributed environment, the system would support a number of default object managers. For a Unix environment, these object managers would manage file, text, data, bss,

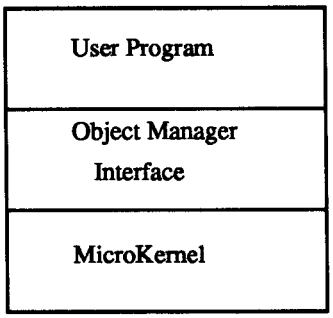

Figure 6.6: The Architectural Layers for the Object Manager

stack and environment objects. A user can use these object managers directly by specifying the corresponding type in the MapObject or Create call. However, users are encouraged to build their own object managers for objects (e.g. voice or video) that may have totally different characteristics to the types supported by the default object managers. The layers associated with an object manager are shown in Figure 6.6.

6.7.3 Benefits

The key advantages of this interface is that it is small and fairly easy to use. In addition, object managers are free to implement their own mechanisms for moving data to and from secondary storage. This is vital when factors such as the **Quality-of-Service** or QoS associated with the object must be taken into account. Finally, the interface uses virtual memory techniques to avoid excessive copying across address spaces.

6.8 Summary

This chapter has detailed the design of an object-oriented virtual memory interface highlighting different design issues. This interface is built on the idea of the *protected process* map which is protected by the virtual memory system and maintains information about objects that are mapped into the user's address space. It also allows the programmer to use logical abstractions without worrying about how data is moved to and from secondary storage.

Object managers are user-level processes that manage the movement of data from an object in main memory to and from its backing store. A standard interface is provided allowing users to build their own object managers. Virtual memory techniques are used

to avoid copying from the object manager to the object pagetables without overloading the object manager's address space. Examples of how the proposed interface may be used have been demonstrated, showing its flexibility and its simplicity. An implementation of the interface is discussed in the next chapter.

Chapter 7

Implementation

7.1 Introduction

This chapter details the implementation of the virtual memory management interface outlined in the previous chapter. The hardware and relevant aspects of Wanda, an experimental operating system being developed at the University of Cambridge, are first described. Subsequently, new algorithms and data structures required to implement the interface are investigated.

7.2 Hardware

A single board computer system containing a 68020 CPU with a paged memory management unit (PMMU) and 1 MB of RAM running at 16MHz was used. This arrangement is rated at 2 MIPS [**Ratzer87**]. Other cards included an 8 MB RAM card, an Ethernet controller VME card and two other cards each containing 1 MB of non-volatile memory. The cards were placed in a crate.

7.3 Wanda

Wanda is a micro-kernel operating system under development at the University of Cambridge since 1988. It is being used as a vehicle for research into a number of different areas including fast packet switching [**Leslie91**]. Wanda supports a multiprocessor environment and was first implemented on the Firefly, a prototype multiprocessor developed at DEC SRC [**Thacker87**]. In the summer of 1989, Wanda was ported to the 68020 machines with PMMU's and subsequently to the 68030's. Versions of Wanda for 68020's with no memory management facilities have also been implemented. The kernel has also been recently ported to the ACORN RISC Machine (ARM) and 68000's systems. Components of Wanda that are relevant to this implementation including scheduling, the event mechanism and the virtual memory management system are *briefly* described below.

Wanda has a preemptive scheduler which supports scheduling in a multiprocessor environment. Threads are scheduled based on their priorities which include a **kernel** priority, a **user** priority and an **idler** priority for threads that are run when no user or kernel threads are running. The system also provides both kernel and user-level semaphores. Threads waiting on semaphores are allowed to spin on idle processors waiting to be unblocked, providing low latency.

Low-level events, including hardware exceptions, may be communicated to the relevant process via an **event mechanism**. When each address space is started, an **event-waiter** thread is created. Using the **Wait-for-Event system** call, this thread is notified of different events. When an event occurs, the event type and event-specific data structures are placed on the event queue of the process and its event-waiter thread is signalled.

Wanda supports a pagesize of 1024 bytes, a maximum segment size of 4 MB and a virtual address space of 128 MBytes on the 680x0's. The virtual memory interface provides calls which allow user processes to obtain physical memory as a list of pages and map it into their address spaces. A privileged user process called the **Process Server** or ProcSvr uses this interface to start other user processes. All the physical memory is allocated before the process is started and the system does not support the paging of objects.

7.3.1 The ANSA Testbench

In order to build a distributed system, the **ANSA Testbench** [**Arc90**] has been ported to Wanda. The Testbench allows servers to export interfaces via a matching service known as the **Trader**. The interfaces are imported by client programs and a client obtains service by remotely invoking operations on the interface. The invocation and the associated data

structures are made into a message which is transmitted to the server. The results are communicated back to the client.

7.4 New VM Structures

Virtual memory structures in Wanda were designed to manage pagetables and physical pages allowing them to be mapped in and out of address spaces. Access control protection was implemented using only the access control bits in the page descriptors. For segmented-paged architectures like the 68020, no protection was enforced using segment descriptors.

My Changes

To allow users to share the same object with different access rights, it was necessary to implement more protection using segment descriptors. The other issue concerned the direction in which segments are grown. This is also specified in the segment descriptor to minimise the size of the associated pagetables. It was decided that two objects which are growing in different directions would not be placed in the same segment since this would result in large pagetables. This meant that a more sophisticated access control checking mechanism had to be implemented which took these factors into account.

It was also necessary to build another facility to associate a region of an address space with an object reference so that it would be possible to translate an address in the user's address space to an object number and a page number. This information was placed in a data structure and queued on a linked list associated with the address space. It included the object number, the page in the address space at which the object starts and the length of the object in pages.

Each object has its own pagetable. When an object is mapped into the address space, the pagetable allocated for the object in that address space uses *indirect* page descriptors, as described in Section 2.4.3, to point to the corresponding page descriptors in the pagetable of the object. Thus the page descriptors in the object pagetable reflect the access patterns of all the address spaces that are sharing the object.

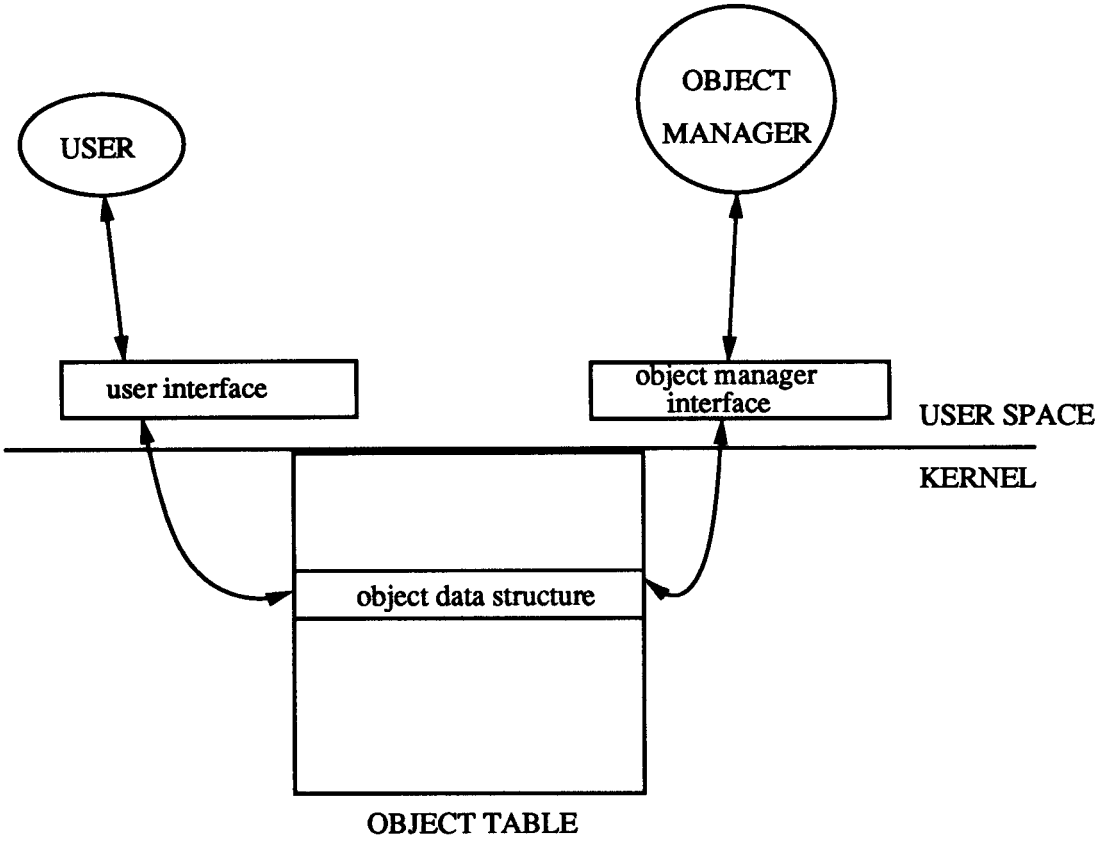

Figure 7.1: The Overall System

7.5 Object Management

Object managers interact with users via an **Object Table**. The object table resides in the kernel but may be mapped into the address spaces of object managers. The overall system is shown in Figure 7.1.

Active objects of a given type are also linked together in a **type** list. The data structure used to represent an object contains a number of characteristics:

name: the name of the object.

obj_type: its sys_type number.

handler: the object manager responsible for the object.

pgtable: the pagetables of the object.

PageBit_Map: this is an array containing page-specific data.

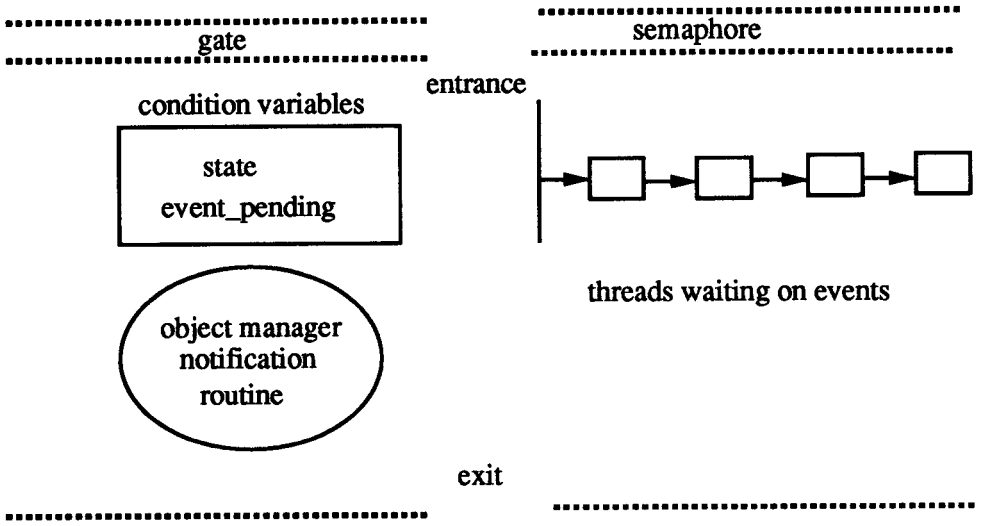

Figure 7.2: The Object Monitor

obj_locktype: the type of locking permitted on the object.

owner: the address space that owns the object if the object was created locally.

addrlist: the list of address spaces into which the object is mapped.

page_info: information on the paging algorithms to be used on this object.

monitor data structures: a monitor structure used to synchronise access to the object.

object routines: routines that are associated with the management of the object.

7.5.1 The Monitor Structure

Each object has data structures to implement a monitor, including a gate semaphore and condition variables as shown in Figure 7.2. The variables associated with the monitor are the **state** variable, which indicates the state of the object, and the **event_pending** variable. The event_pending variable is used to indicate **actions** or **events** which must occur before particular threads can proceed. Threads waiting on events are queued on a single queue within the monitor. Threads waiting on a given event are unblocked when this event has occurred.

The Wanda event mechanism is used to notify an object manager of events occurring on an object which require its attention. The object manager does an *Investigate* call to examine the state of the object and get the necessary information to deal with the event.

It then returns to user space. After servicing the event, it does a *ReturnResult* call to return the results of the call and unblocks threads waiting on the event.

7.6 The Interaction Protocol

The interaction between a user thread and the object manager is based on a simple protocol. The rules are:

1. User threads obtain service by setting event_pending indicators.
2. Once an object is created, only the object_manager can change the state of an object. This simplifies the protocol and allows the object manager to refine it. For example, errors detected by the object manager are indicated by setting the object to an ERROR state.

A state machine of the protocol is shown in Figure 7.3. To demonstrate the protocol, we suppose that a user has just invoked the MapObject call on the interface. The type list containing objects of that type is first searched to see if the name matches the name of any object in the type list. If not, a new object data structure is created with its initial state set to **NASCENT**. The object manager is then signalled to get the pagetables for the object. To do this, the user thread sets the event_pending variable to **PHY_PENDING** and sends a message to the object manager. The object manager must first obtain the size of the object. This is done by querying the remote storage server using an ANSA RPC. It then issues a *ConstructPageTable* call.

When this is completed, the object manager sets the state of the object to **PHY_ALLOCATED** and unblocks all the threads waiting on the PHY_PENDING event. The user thread may now map the object into its address space. After this operation, the object manager is again signalled by the user thread using a **MEM_PENDING** event notification. The object manager looks at the object to see if any further service is needed before the object is accessed. For example, if data from a physical segment needs to be preloaded into the object, it may be done at this stage. The object manager changes the state of the object to **MEM_ALLOCATED** and unblocks all the threads waiting on the MEM_PENDING event. The user thread now returns to user-space and can begin to access the object.

Once in the MEM_ALLOCATED state, certain events do not cause a change in state: for example, when a pagefault occurs or when a flush operation is invoked on an object.

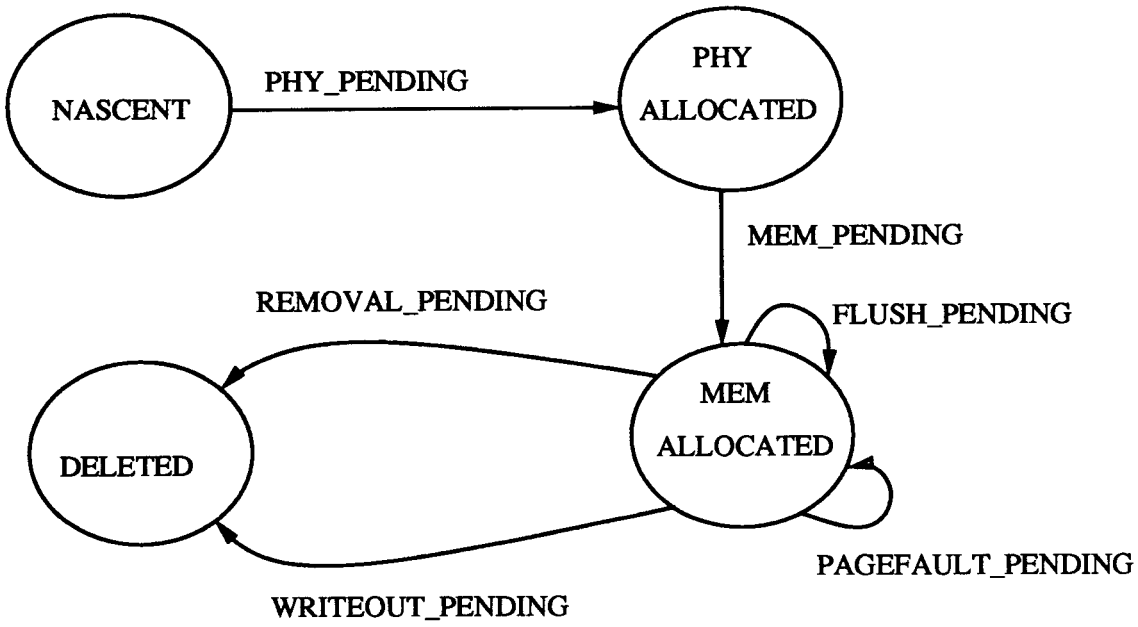

Figure 7.3: The Interaction Protocol

When the user has finished using an object, the UnmapObject routine is called. If no other thread in the user's address space has opened the object, it is unmapped from the user's address space. If no other address space is using the object, and it is mapped read/write with no discard indicator being set, the system attempts to write the modified pages back to memory. It queues the list of modified pages onto the object data structure and signals the object manager using the WRITEOUT_PENDING event indication.

The object manager then proceeds to write out the modified pages. It will then check to see if another user has attempted to reuse the object. This is indicated by setting the REUSE_PENDING event indication. If this event is not pending, the object manager then checks to see if the save indicator is set. This is set in response to the MAP_SAVE option which tells the system to keep a local copy around as long as possible. If this is not set, then the object manager proceeds to deallocate the resources associated with the object.

If the discard bit on an object is set, or the object is mapped read-only then there is no need to set the WRITEOUT_PENDING event indication, instead the user thread sets the REMOVAL_PENDING event indication which causes the object manager to remove the object. If, however, the save option is set, then this action is not initiated and the call terminates.

7.6.1 Created Objects

The entire sequence outlined above need not be followed by objects that are created using the Create call. This is because the Create call specifies the size of the object to be created. This means that the system can create the pagetables and map the object into memory without the assistance of the object manager so these objects are normally put in the initial state of MEM_ALLOCATED. Created objects also have their MAP_ZEROFILL indicator set so that when a pagefault occurs, only a zero-filled page is ever required. There is no need to contact an object manager.

Temporal objects which are created for the lifetime of a process, for example, the bss region of a process, are created with the MAP_DISCARD option being set in the Create call. Flushing is disabled and the object is deleted after it is used. No backing store is ever allocated for such an object.

7.6.2 Benefits

One of the major benefits of this design is the *clear separation* of the interaction protocol and the monitor/event concept. This permits us to change the interaction protocol, the *policy*, without having to change the *mechanism*, the monitor/event technique. Hence if the interaction protocol is changed, the monitor code will not be affected.

7.6.3 Pagefault Handling

Extra support is needed to handle pagefaults. This is because there may be any number of pagefaults occurring on an object at any one time, either on the same page or on different pages as objects may be shared among different address spaces. When a pagefault occurs it is necessary to know which page of the object is involved. This is done by the low-level virtual memory management system which translates the faulted address in user's address space to an object number and a page number.

Given these parameters, the exception routine then jumps into the monitor of the object. The PageBit_Map is an array that is indexed using the page number of the faulted region of the address space. Each element of the PageBit_Map contains a variable that indicates the **state** of the corresponding page.

Page states are:

PAGE_NOT_PRESENT: the page is not present.

PAGE_REQUESTED: the page has been requested but not yet brought into memory.

PAGE_IN_MEMORY: the page is in memory. This is set once the page has been mapped into the object.

PAGE_ON_SWAPQUEUE: the page is waiting to be paged out.

PAGE_ON_RECLAIM_QUEUE: the page is waiting to be reclaimed by the system.

PAGE_SWAPPED_OUT: the page has been paged out.

PAGE_ON_QUEUE: the page is about to be written out.

PAGE_WRITEOUT_PENDING: the page is being written out.

If the page is not present, the faulted thread first gets a page from the free list. This page is then made accessible to the object manager. The thread then sets the event_pending variable to PAGEFAULT_PENDING and then signals the object manager. Threads waiting on pagefaults are not placed on the monitor queue because it will be necessary to kick not only the threads waiting on a particular pagefault to be serviced. So the page number of the fault must also be known and this information is not part of the monitor and thus the PageBit_Map is used. A **threadlist**, another structure in each element of the PageBit_Map, is used to queue threads waiting on pagefaults for that particular page to be serviced.

When an object manager gets the data from the remote server to service a pagefault, it issues the *ReturnResult* call. This call first maps in the assigned page into the resident set of the process and then unblocks all the threads waiting in the threadlist for the pagefault to be serviced. The page is now mapped into the pagetables for the object. It is also made inaccessible to the object manager.

7.6.4 Copy-On-Write Mechanisms

It was decided to use a scheme similar to the copy-on-write mechanism used in Chorus as this appeared simpler to implement than the Mach copy-on-write mechanisms. Unmodified pages are placed in a structure called an **archive**. An object primarily has two archives associated with it. The first archive is called its **source** archive. When a pagefault occurs in an object, the source archive is first searched. An object cannot modify the pages in its source archive. Hence, if other users wish to share the original object, this archive is used.

The other archive is called its **copy** archive. All the pages that are changed by the user when the object is mapped copy-on-write are placed in this archive. Thus if other users wish to share the object, incorporating the most recent changes, a new object is created such that its source archive is the copy archive of the previous object.

Copy and source archives may be joined together so that a chain develops and when a pagefault occurs lower down in the chain, archives are recursively searched, each step going to a higher archive in the chain until the page is found or a pagefault is signalled. When the pagefault is satisfied the page is also placed in the uppermost archive (called the root archive) so that other archives in different parts of the overall structure will also have access to it.

7.6.5 Paging

Swap Device

The swap device was 2 MB of non-volatile RAM. It was decided to use non-volatile RAM rather than disk to experiment with faster swap times and compare them with disk access times. When the system is booted, the kernel counts the number of non-volatile pages that are available and puts them on a free list. The swap area is managed on a per-object basis with each swap object having a list of pages that have been paged out. When a list of pages from an object must be paged out, the contents of these pages are copied to swap pages which are then queued on its swap object.

It should also be pointed out that, for this implementation, there is no inherent advantage in using non-volatile RAM as compared with RAM. When the system was being configured a few Megabytes of non-volatile RAM was available so this was used. RAM was scarce because it was needed for other Wanda machines. The use of non-volatile RAM may be relevant in the context of integrating the management of swap space with that of a general filing system but this issue is not addressed here.

The Pager

Pages that are removed from the resident set are placed on a special queue in the object data structure call the **swap** queue. The page state for the page is changed to PAGE_ON_SWAPQUEUE. The pages are paged out by a kernel thread designated as the **Pager**.

The Pager operates on a list of requests using a **PageSemaphore**. When pages from an object need to be paged out, a request is sent to the Pager and the PageSemaphore is signalled. The Pager waits on the semaphore and when unblocked, dequeues the request, finds the respective object and moves the pages to the swap device.

An interesting operational issue is the priority at which the Pager should execute. If it is set at kernel priority, the Pager would be invoked every time pages are removed from the resident set since the Wanda scheduler is preemptive and the priority of the Pager would be higher that than of the object manager servicing the fault!

It was therefore decided to run the Pager at user-priority to allow several pages to accumulate before the Pager pages them out in a single operation. This increases the throughput at which pages are moved to secondary storage. In addition, if a page has been previously paged out and is needed very shortly afterwards, then it can be easily reclaimed from the swap queue. This would avoid the need to consult the swap space of the object, thus increasing the throughput of the user process. The effect of this policy was tested and experimental results were obtained.

The PageStealer

Rather than place pages that have been paged out back on the system free pagelist, the Pager places these pages on another queue in the object data structure. This queue is called the **page-reclaim queue** and the page state is altered to PAGE_ON_RECLAIM_QUEUE. The system runs a kernel thread called the **PageStealer** which is responsible for, among other things, reclaiming pages from objects.

When memory on the freelist reaches a lower threshold, about 15% of the total, the PageStealer is invoked. It examines each type list in turn. For each object in the list, it checks if any address spaces are referencing the object. If not, the object has been saved at the request of a user. It frees all the physical pages associated with the object placing them back on the free pagelist. If the object is in use, the PageStealer deallocates pages that are on the reclaim list. It continues this operation until it has finished going through the type list.

The motivation behind this approach is to keep pages in the vicinity of the object for as long as possible. If a page fault occurs on a page and its page state indicates that is on the reclaim queue, then the page is retrieved and brought back into the resident set, the copy of the page on the swap space is invalidated and the associated page on the swap device is freed. This saves the extra copy from swap device back into memory.

7.6.6 Object Routines

The mechanisms detailed above are implemented by routines that form part of the definition of an object. This allows the virtual memory management system to be implemented in an object-oriented manner. The routines are:

- **ObjSatisfyFault**: This routine is invoked when a pagefault must be serviced and the object manager must be contacted to obtain the page from the remote fileserver.
- **ObjZeroFill**: This is called when a zero-filled page is required to satisfy a pagefault. This occurs when the object has been created or is being extended on the local site.
- **ObjPageAlgol**: This is the paging algorithm associated with the object.
- **ObjReclaim**: This routine is invoked when it is necessary to retrieve a page that is on the reclaim queue.
- **ObjSwapOut**: This is called by Pager to move the pages on the swap queue of an object to its backing store.
- **ObjSwapIn**: This routine is invoked when a page on the swap device is required in main memory.
- **ObjPrefetch**: This is invoked as part of the implementation of prefetching paging algorithms which are discussed in Chapter 9.

7.6.7 Synchronisation

To implement the many-readers-one-writer synchronisation defined by the interface specification, another monitor structure was used. There are four event_pending indications associated with this monitor. They are:

REQUEST_READLOCK_PENDING: indicates that there are users waiting to read the object.

REQUEST_WRITELOCK_PENDING: indicates that there are writers waiting to write.

RELEASE_READLOCK_PENDING: indicates that there are users presently reading the object.

RELEASE_WRITELOCK_PENDING: indicates that a user is presently writing to the object.

There are also variables which show the number of threads associated with each event. These form part of the **lock** data structure which also contains a gate semaphore and a single event queue. Let us suppose that the synchronisation policy is as follows:

1. Writers are favoured over readers.

When a process wants to read an object, it invokes the AcquireLock call with the lock indication set to Read_Lock, enters the monitor and checks the event_pending variable. If no event is pending, it sets that variable to RELEASE_READLOCK_PENDING, increments the corresponding variable and returns to user-space. In addition, if the RELEASE_READLOCK_PENDING indication is the only indication that is set, then it is safe to read the object as other users are currently reading. If, however, the RELEASE_WRITELOCK_PENDING or REQUEST_WRITELOCK_PENDING flag is set, then the user process increments the REQUEST_READLOCK_PENDING variable and is then queued on the event queue.

When a process wants to write to the object, it invokes the AcquireLock call with the lock indication set to Write_Lock. If no event is pending, then the RELEASE_WRITELOCK_PENDING indication is set and the corresponding variable incremented. The user then returns to user-space. However, if any event_pending indication is set, the writer sets the REQUEST_WRITELOCK_PENDING, increments the number of writers waiting and queues on the event queue.

When a user process has finished reading, it decrements the variable which indicates the number of other processes currently reading the object. If this is non-zero, it indicates that some processes are still reading and the call terminates. However, if the process is the last reader, then it first checks to see if there are any writers waiting and if so, decrements the number of writers waiting and unblocks the first one in the queue before returning to user-space.

When a writer has finished a write operation, it first checks to see if there are any others writers are present. If so, it decrements the number of writers waiting and unblocks the first writer. If there are no other writers, it then checks to see there are any users waiting to read the page. If so, they are all unblocked.

The synchronisation policy here will always ensure that readers are accessing the most current version of the object. Other policies can also be implemented using this model.

For example, there could be serialisation, in which reads and writes occur in the same order as the interface calls are invoked.

It is very easy to build an interface for a **lock manager** using the same techniques used to build the interface for object managers. Thus, where necessary, an event is generated and sent to the lock manager. The lock manager takes the necessary steps, giving access to portions of the object as required. Such a scheme would be applicable in database environments where fine-grain locking is essential.

7.6.8 Unix File Handling Routines

Part of the Unix file interface was ported using the interface. The following library routines were supported:-

fopen (filename, mode) – Uses the MapObject call to map the file into the user's address space. Presently read-only, read-write and copy-on-write modes are supported.

fcreate (file, mode) – This call will Create a file using the create call with a default size.

fclose (fp) – closes the file whose structure is given by fp. This uses the UnmapObject call to unmap the file from the user's address space.

fflush (fp) – causes a file to be flushed. The FlushObject call is invoked to cause the object to be flushed to the permanent storage space.

fseek (fp, offset, whence) – positions a file at a given location.

freopen(filename, type, stream).

Calls based on the Unix I/O interface are also supported for memory mapped files:

fd = open (name, flag, mode)

bytes_read = read (fd, buffer, n)

bytes_written = write (fd, buffer, n)

position = fseek (fd, offset, whence)

Reading characters from a file is similar to the Unix library call, however writing is a bit different, since if the file goes beyond its allocated size, it must be extended using the *Extend* call.

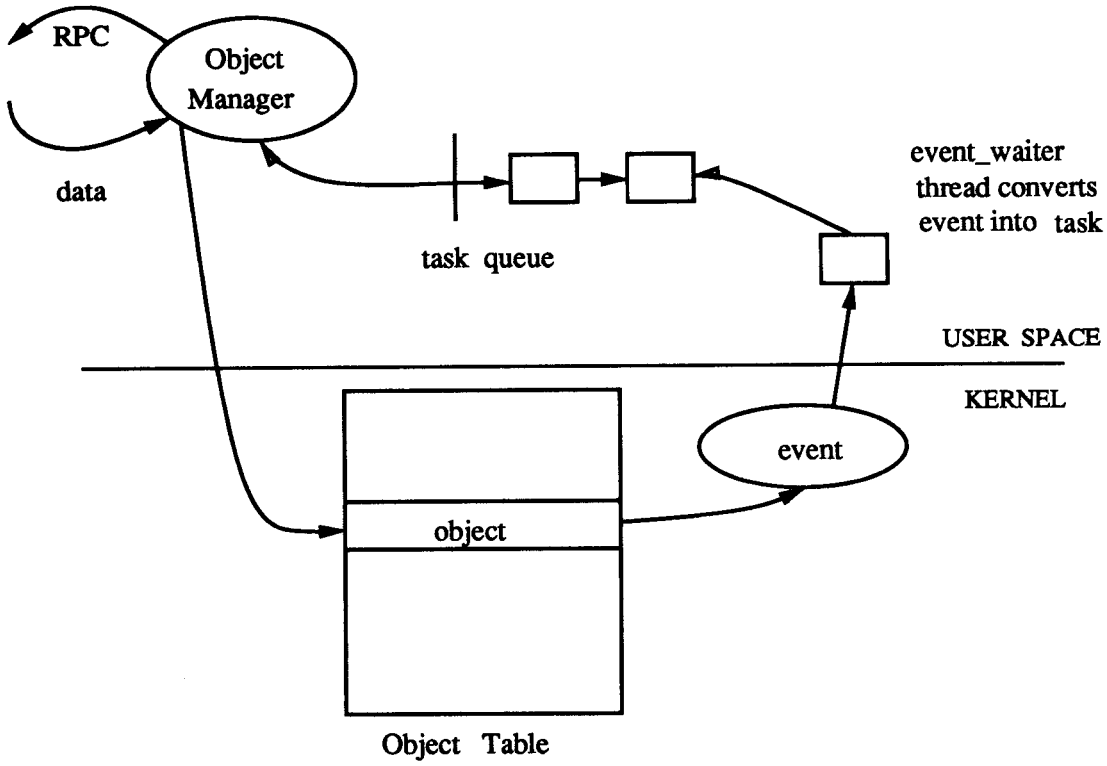

Figure 7.4: The Object Manager

7.7 Object Managers

As previously indicated, object managers service requests on objects and once an object is created only its object manager can change its state. The system has a default object manager called the **PageSvr**. The Wanda event mechanism is used to signal to the object manager that an object requires service. The event_waiter thread of the object manager uses this event notification to compose a task for which the attention of the task manager is required. The task is then queued onto a task queue as shown in Figure 7.4. The event_waiter then signals a **Work** semaphore. When the object manager has no work to perform, it is blocked on the Work semaphore. When it is unblocked it dequeues the respective task. It does an *Investigate* call to determine the true state of the object and to obtain more information to deal with the request. When the object manager receives a REMOVAL_PENDING or WRITEOUT_PENDING event indication, it flushes the task queue, removing other tasks related to that object.

Parameter	Value
Time between a pagefault and waiting on the object manager	0.8842 (msecs)
Time to service a zerofill fault	1.250 (msecs)
Time for the ProcSvr to create all the objects for a new process	81.450 (msecs)
Maximum sustained throughput of the PageSvr	38.6 (pages/sec)

Table 7.1: Preliminary Performance Results

7.7.1 Logger

One of the functions assigned to the default object manager is to log information about the creation and deletion of objects. A thread known as the **logger** is spawned off when the PageSvr is started up. This thread invokes a **GetLog** call that blocks in the kernel waiting on events to occur. When an object is created the information is placed in a **log** data structure. The Log Table is mapped into the address space of the PageSvr.

The logger writes the information about objects that are created or deleted into pre-allocated buffer. An ANSA RPC is then used to append this information to an NFS file. Information stored by the logger when an object is created comprise the name of the object, its type, the paging algorithm to be used on the object, its object manager and its size. When an object is deleted, additional information on the paging activity of the object may also be recorded.

7.8 Preliminary Performance Measurements

Though the interface has not been tuned for optimum performance, some results are now presented to show how well the system performs. These results were obtained from five runs of the GCC Compiler Suite, and represent average readings. The results are shown in Table 7.1.

The first reading measures from the time a pagefault occurs to when the faulted thread is blocked on the threadlist waiting for the object manager to service the fault. This involves entering the object monitor, checking the page state, sending a message to the object manager and then blocking on the threadlist. The second measurement involves the time it takes to service a zero-fill pagefault.

To create a new process, the ProcSvr first gets a new address space and then maps each of the objects needed into that address space using the CreateProcObject call. Each process

contains one object of the following types; text, data, bss, stack and environment. The environment object is used to map the environment variables for the process. Arguments for the stack as well as environment variables are contained in the corresponding physical data segments. The arguments for the stack are preloaded when it is mapped into the address space. It then creates the initial thread and starts the address space. The average time for these routines is shown in the above table. This result is satisfactory because the ProcSvr is operating in user-space.

Finally, the rate at which the PageSvr gets pages using an ANSA RPC to another process called the ReqSvr was also measured. Since this operation uses the network, the efficiency would vary according to the instantaneous traffic on the network. These measurements represent the maximum sustained throughput measured over a period of twenty seconds.

The results show the PageSvr operation to be relatively slow. There are several reasons for this, the most significant being the relative slowness of the ANSA RPC mechanism [**Nicolaou90**, pages 148–149]. It should also be noted that the ReqSvr has not been tuned for optimal performance. In RPC mechanisms, the marshalling and unmarshalling of arguments in the remote invocation are time consuming. While such facilities are needed for sophisticated applications with complex interfaces, the entire RPC mechanism is a heavyweight operation when used for such a low-level operation as the servicing of page-faults. A less complicated transfer mechanism was suggested and shown to significantly improve the throughput of low-level services such as paging and file transfers [**Wilson91**].

7.9 Summary: The New World

In this chapter, the implementation of the interface specified in Chapter 6 on a 68020 configuration running the Wanda operating system was described. Changes to the Wanda memory management were detailed and new data structures were introduced.

Co-ordination between object managers and users wanting service is achieved via an Object Table with a monitor being used to synchronise access to object data structures. An interaction protocol specifies the different states and events associated with object management. This arrangement results in the clear separation between the policies and the mechanisms used in the management of objects. Another monitor is used to synchronise user access and a many-readers-one-writer synchronisation mechanism has been implemented. Some features of the Unix file interface have also been ported to the system.

The boot image of the new system comprises the kernel image and the images of the ProcSvr and the PageSvr. The PageSvr manages objects of types **file**, **text**, **data**, **bss**,

stack and **env**. However, users are free to build their own object managers to manage different types of objects.

Pagefaults that occur in address spaces are handled by the PageSvr. It uses the ANSA RPC mechanism to get the data. The PageSvr also spawns off the logger thread when it starts up which logs the creation and deletion of objects. Preliminary results indicate that the system performs well though it has not been tuned to achieve optimal performance.

As shown in this chapter, it is relatively easy to extend the interface to provide support for additional servers, e.g. lock managers for database systems. One such extension is discussed in the next chapter.

Chapter 8

Performance of Traditional Paging Algorithms

8.1 Introduction

In this chapter, the performance of three local paging algorithms, the local LRU algorithm, the Working Set algorithm and the Page Fault Frequency algorithm are examined. These algorithms were chosen because they, or some of their variants, have been implemented in several operating systems.

A new framework for comparing algorithms as well as describing program behaviour is first developed. Its main objective is to highlight the essential nature of replacement algorithms [**Carr84**]. Secondly, an experimental testbed is set up to investigate the performance of different paging algorithms operating on different types of objects. The testbed is an extension of the interface described earlier and incorporates the necessary data structures to record the parameters of paging activity specified by the new framework.

The GCC Compiler Suite was examined using the testbed and experimental results for different paging algorithms are presented.

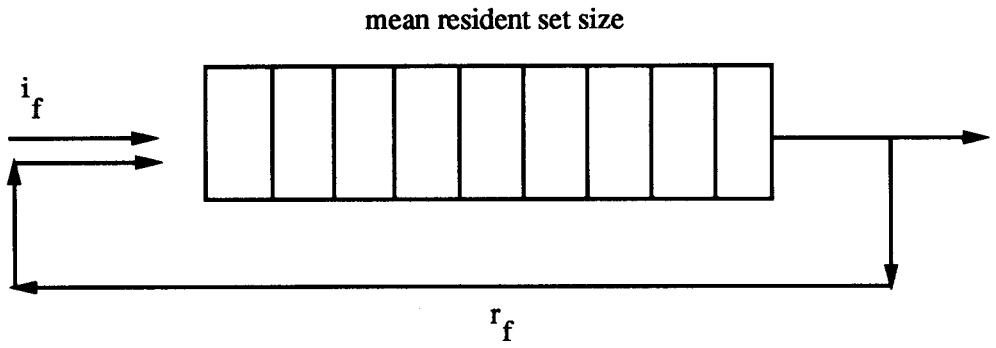

Figure 8.1: Paging Activity

8.2 A Framework For Paging

The following view of paging, as shown in Figure 8.1, is used to develop a new framework. Pagefaults on an object are divided into two types: **initial faults** or i_f and **reclaim faults** or r_f. An initial fault is generated when a page is first referenced while reclaim faults are generated on pages that were previously in the resident set but have been paged out by the paging algorithm. Initial faults are therefore *independent* of the paging algorithm being employed while reclaim faults are *entirely* caused by the paging algorithm.

Let the ratio called the **reclaim ratio**, rr, be defined as follows:

$$rr = \frac{r_f}{i_f} \tag{8.1}$$

This ratio serves as a basis of comparison between two algorithms:

> A paging algorithm X is better than a paging algorithm Y for a given object if for the same mean resident size, m, the reclaim ratio of X is lower than that of Y.

The lifetime curve can also be related to the reclaim ratio. The average interpagefault time, ip, can be expressed as a function of the mean resident size, m:

$$ip(m) = \frac{T_{ex}}{i_f + r_f(m)} \tag{8.2}$$

where: T_{ex} is the total execution time of the program and $r_f(m)$ is the relationship between the number of reclaim faults and mean resident set size.

By dividing the numerator and the denominator by i_f, ip can be expressed in terms of rr:

$$ip(m) = \frac{\frac{T_{ex}}{i_f}}{1 + rr(m)} \tag{8.3}$$

The numerator of Equation 8.3 represents the average interpagefault time when no paging algorithm is used. This can be represented by a constant for a given object, giving the equation:

$$\frac{T_{ex}}{i_f} = C_0 \tag{8.4}$$

Thus we can represent $ip(m)$ as:

$$ip(m) = \frac{C_0}{1 + rr(m)} \tag{8.5}$$

where $rr(m)$ is the reclaim ratio as a function of m.

From Equation 8.5, it is apparent that $ip(m)$ is dependent on $rr(m)$. Thus $rr(m)$ can be used as a reflection of program behaviour. This is good for two reasons. Firstly, it is possible to measure rr with greater accuracy than it is possible to measure ip since measuring a large number of very small time differences is disruptive and is likely to be less accurate. Secondly, there is a simple relationship between rr and ip since ip for a given object is directly proportional to $1/(1 + rr)$.

It is easy to differentiate between reclaim faults and initial faults using the interface developed in the previous chapters since, as indicated in Section 7.6.3, each page of an object has an associated state. The PAGE_NOT_PRESENT state indicates that the corresponding page has never been in memory and thus was never part of the resident set. A page that has been removed from the resident set may be in one of three states depending on the stage it has reached in attempting to get onto the swap device.

8.3 Design and Implementation of the TestBed

8.3.1 PageData Structures

The main goal of the testbed is to be able to specify different paging algorithms to be used on various objects and to record and analyse the subsequent paging activity. A special data structure called the **pagedata** structure is used to specify information on paging algorithms. It comprises:

pag_algol: the paging algorithm being employed.

limit: specifies the maximum number of pages that the object can have in its resident set.

window: specifies a time factor beyond which pages are removed from the resident set. This is used by working set algorithms.

n_pages: specifies the number of pages that should be brought in when a pagefault occurs. For demand-paging algorithms this is set to one.

8.3.2 Paging Interface

Information on the paging algorithms for different objects may be passed to the kernel using two different mechanisms. First, the object manager, when it registers with the kernel to handle events associated with a given object type, may also specify the paging algorithm to be used when an object of this type is paged. The other mechanism involves the building and management of a **paging database** which contains information on how *individual* objects are to be paged. The database consists of a set of **pdata** structures, each containing the object name and its object type as well as a pagedata structure with the paging information. The interface to manipulate the entries in the database comprises:

- **AddPg_Base (pdata)**: adds an entry to the database.
- **CheckPg_Base (pdata)**: returns the paging information about the object specified by the object name and object type contained in the pdata structure.
- **ChangePg_Base (pdata)**: changes the paging algorithm associated with an object specified by its object name and object type.
- **DeletePg_Base (pobject, ptype)**: deletes the entry associated with the object specified by the object name, *pobject*, and the object type denoted by *ptype*.

When an object is being created, the database is checked to see if it contains any information about the object. To speed up this operation, the database is partitioned into subsections based on the number of object types supported by the system. If a match is found, the corresponding algorithm is used. If not, the paging algorithm specified by the corresponding object manager is employed.

8.3.3 Remote Paging Specification

To simplify the testing of paging algorithms, a **remote paging specification** was developed. Paging information about objects of interest is placed in a remote file. When the system is started, the logger maps the file into the address space of the PageSvr and reads its contents. It then adds this information to the paging database using the *AddPg_Base* call. Thus, if these objects are created, they are paged in the manner specified by the remote file. This mechanism was used during the testing of paging algorithms. To change the paging algorithm associated with an object, the remote specification for this object was changed. This was much quicker than embedding paging information in individual programs or libraries since they would have to be recompiled and relinked every time a change was made to the paging algorithms.

8.3.4 Recording Information

When paging information about an object is required, the MAP_RECORD bit field is set in the access variable as the object is being mapped in. Information on the pagefaults of all the objects that are mapped into an address space is placed in a single file.

Pagefault data structures are used to record the following information about the object when a pagefault occurs:

object-number: the object number of the faulted object.

block or **page** number: the number of the page within the object which contains the faulted address. Thus it is possible to examine the order in which an object was accessed.

virtual process time: the process time at which the pagefault occurred. This can be used to calculate the rate at which pagefaults are occurring in different phases of the program as well as the interpagefault time.

service fault time: the time taken to service the pagefault. This serves as a clear basis for comparing the overhead involved in using different paging algorithms.

initial faults: the number of initial pagefaults.

reclaim faults: the number of pages that have been re-introduced into the resident set.

swapped-out pages: the total number of pages that have been removed from the resident set.

resident set size: the number of pages in the resident set at the time of the pagefault. This allows us to calculate both the mean and standard deviation of the resident set size.

8.3.5 The Recorder

Information on the paging activity of a process is kept in pagefault structures and as an object faults into the address space, a list of pagefault structures is generated. After the process has finished executing, a message is sent to the process server. The process server checks to see if any pagefault structures are associated with the address space. If so, it sends a message to the **recorder**. The recorder is a special thread that transfers the information from the pagedata structures to a remote file so that it can be analysed in the Unix environment.

The recorder is part of the address space of the PageSvr as shown in Figure 8.2. The pagefault data structures are mapped into its address space, so the recorder can access them directly. It then creates an output file of the appropriate size to which it writes the data contained in the pagefault data structures.

8.4 The Page Fault Algorithms

Three local algorithms were selected for investigation. Firstly, the **Modified LRU** or MLRU algorithm was chosen. This algorithm is similar to pure LRU but uses a counter for each page to indicate how recently a page has been used, rather than manipulating the linked list of resident pages. When the algorithm is invoked, the routine examines the pagetable entry for each page in the resident set. If the used bit is set, it is cleared and the page counter is reset to zero. If the used bit is not set, the counter is incremented. Hence, the page with the highest count will be the least recently used page and will be replaced if necessary.

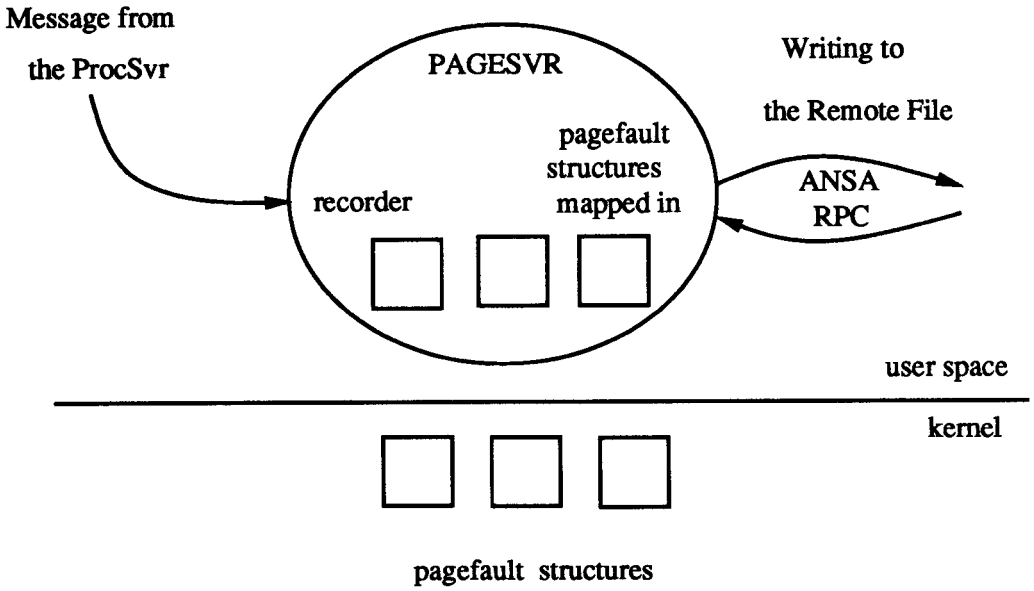

Figure 8.2: The Recorder

Secondly, the **Working Set** or (WS) algorithm was chosen. In this arrangement, the algorithm is invoked after every pagefault. An integer associated with each page records the virtual time at which the page was last referenced. The algorithm examines each page descriptor in the resident set. If the used bit is set, the time variable for the page is set to the virtual process time of the present pagefault. If the bit is not set, then the virtual process time of the present fault is subtracted from the last recorded time that the page was accessed. If this result is greater than θ, the page is removed from the resident set.

Finally, the **Pagefault Fault Frequency** or (PFF) algorithm was examined. This is the *classic* implementation as described in Section 5.2. When a pagefault occurs, if the time of the last pagefault is greater than θ, then all the unused pages are removed otherwise the used bit of the pages that have been referenced are reset and a new page is added to the resident set.

8.5 The Program Suite

To test paging algorithms, it is necessary to decide the set of programs to which these algorithms may be applied. The GCC Compiler Suite (Version 1.37) was chosen as the main benchmark. The suite is composed of four programs.

At the top level there is **gcc68k**. This is the program that is invoked by the user to start

the compilation. gcc68k checks the argument list associated with the call. It then invokes the preprocessor, **gcc68k-cpp** or *cpp*. This program checks the source and include files and produces a file with a **.cpp** extension. When the preprocessor is finished, gcc68k invokes the main compiler **gcc68k-cc1** or *cc1*. This program takes the .cpp file and produces an assembly language file with a **.s** extension. Next the assembler **gcc68k-as** or *as* is then invoked. The assembler translates the .s file to produce a corresponding object file with a **.o** extension.

The GCC Compiler Suite was chosen because the characteristics of the four programs are very different. gcc68k is the front end of the system and acts very much like an interactive job, i.e. it issues commands and waits for responses before proceeding. The preprocessor is I/O intensive especially if several include files are involved since each file must be parsed and the .cpp file must be assembled. The main compiler, cc1, is the most computationally intensive of all the programs. It is also the largest. The assembler is also a mixture of computation and I/O as it translates the assembly language file into the object file.

It was difficult to decide which input file to use in testing the compiler since there is no such thing as a *typical* compilation. The criteria for choosing were: firstly, the file had to include a substantial number of header files to make the preprocessor behave in a very I/O intensive manner. Secondly, it had to be large enough to make the main compiler, cc1, run for several seconds so that paging activity could be measured over a substantial period.

The file **user_type.c** was chosen. It is the code for the type interface mentioned in Chapter 6. It contains 14 header files, ranging in size from 600 to 21,350 bytes, and has 461 lines of C code. The main compiler, cc1, used approximately 19.36 seconds of CPU time in the compilation effort, while the assembler took approximately 5.35 seconds and produced a **user_types.o** file that was 7,055 bytes in length.

8.6 The Analysis of the Results

When the file containing all the pagefaults of a process was obtained, it was processed by several **awk** routines [**Aho84**]. Some routines extracted data that could be plotted immediately, while other routines were embedded in programs that performed detailed calculations. When calculating the mean and standard deviation of the resident set of a program, pagefault calculations were only started *after* a page was first removed from the resident set by the paging algorithm. Thus the resident set size was calculated without including initial faults that occurred before any page was actually paging out.

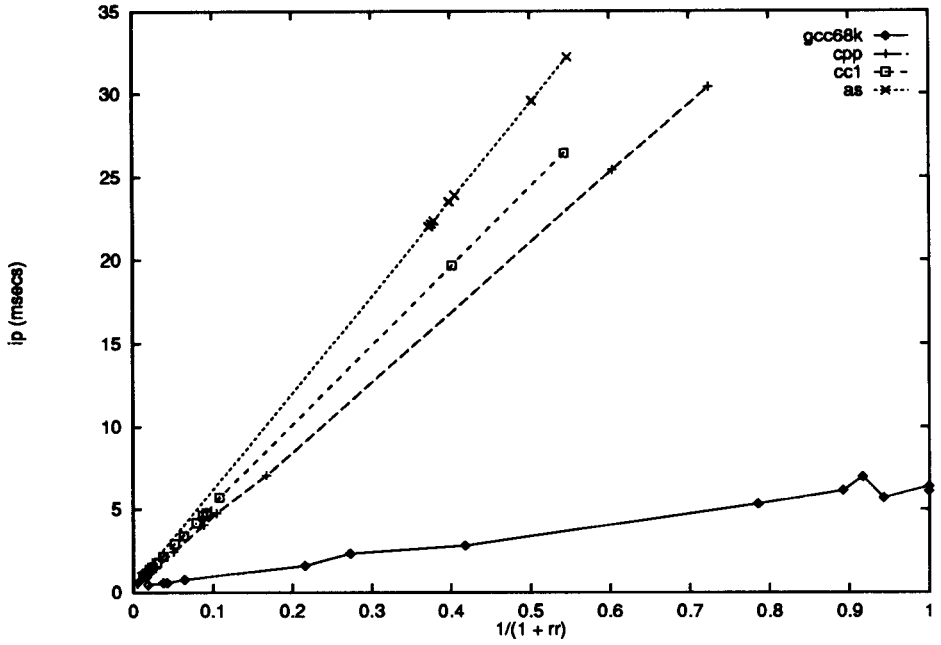

Figure 8.3: The Interpagefault Time vs $1/(1 + rr)$ for the text segments of the Compiler Suite using the WS algorithm

Program	i_f	C_0 (msecs/fault)
gcc68k	33	8.59
cpp	58	41.06
cc1	401	48.37
as	92	58.68

Table 8.1: Values of C_0 for the GCC Compiler Suite

8.7 Results

Equation 8.5 indicates that ip, the interpagefault time, is directly proportional to $1/(1 + rr)$ with C_0, the average interpagefault time when no page algorithm is used, as the gradient of the line. Figure 8.3 shows the graph for the text segments of different programs operating under the WS algorithm. It shows straight lines for the assembler, compiler and preprocessor, while an approximate straight line was obtained for the gcc68k program. The small interpagefault times and the small number of pagefaults for gcc68k accentuate the errors in measuring the interpagefault times. The values of C_0 for the text segment of the individual programs are shown in Table 8.1.

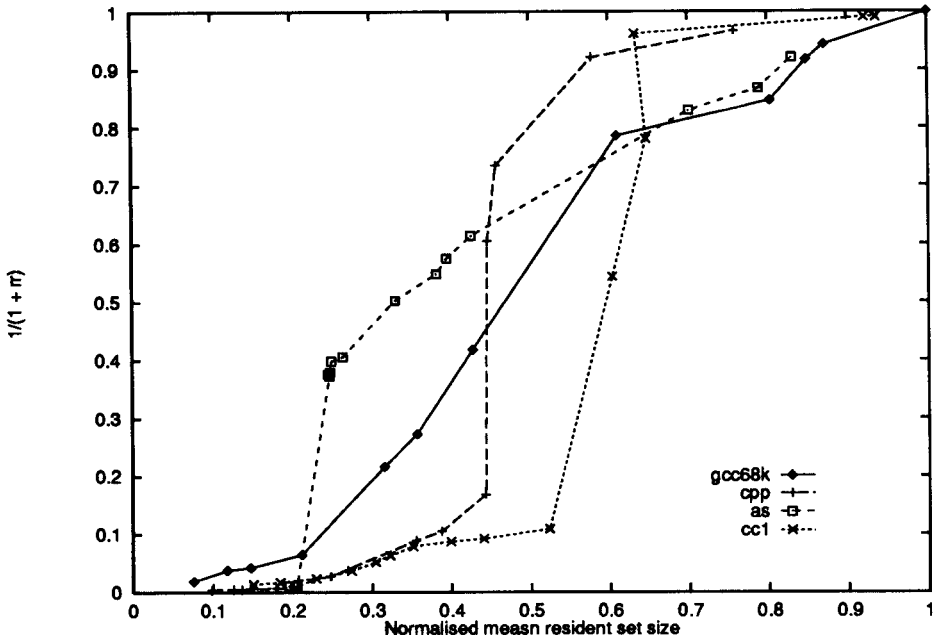

Figure 8.4: $1/(1+rr)$ vs Normalised mean resident set size for gcc68k and cpp

However, whereas the equation predicts a straight line passing through the origin, the measurements show a very small intercept on the y-axis of approximately 372 μsecs on average. This is probably due to the time taken to calculate the virtual time at which the pagefault occurred and as this was included in each reading. An easy way to obtain the value C_0 is to measure the values of ip and rr for any points and find the gradient between them.

8.7.1 A New Criterion for Lifetime Curves

The results obtained have demonstrated that the mean interpagefault time, ip, is directly proportional to $1/(1+rr)$. Thus by plotting $1/(1+rr)$ versus the mean resident size, m, it is possible to obtain a curve that is more representative of program behaviour since it is independent of the particular value of C_0. Objects with the same curves can be said to have similar program behaviour. In the subsequent discussion these curves are referred to as lifetime curves.

The results presented in Figure 8.4 show the lifetime curves for the text segments of each program of the Compiler Suite operating under the WS algorithm. For proper comparison, the mean resident set size is divided by the total number of initial faults, i_f, for each program. Curves for cpp and cc1 have the shape of the traditional lifetime curve but they

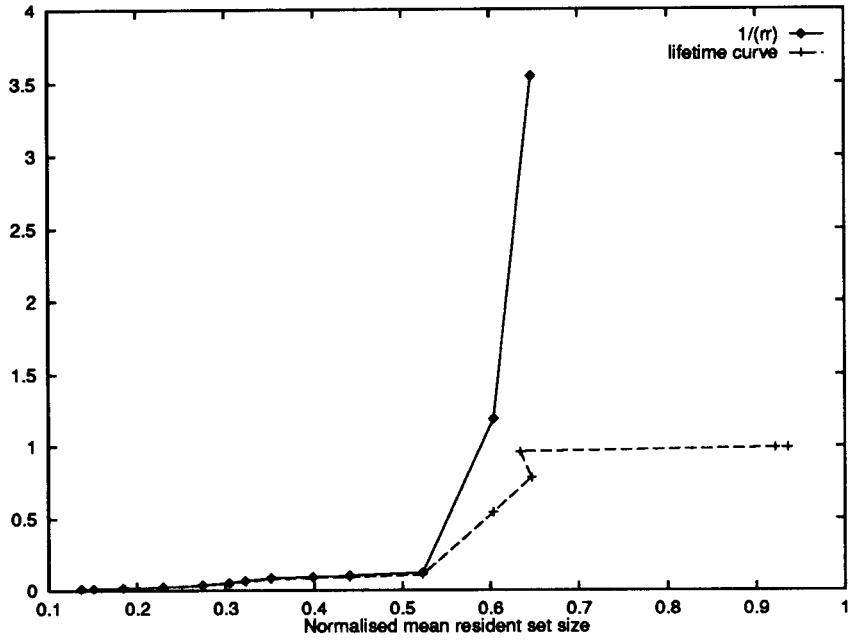

Figure 8.5: Comparing $1/(1+rr)$ and $1/(rr)$

are very different from each other while those for as and gcc68k are not at all similar to the traditional lifetime curve. This result suggests no two lifetime curves are identical and while some are similar there are also large differences.

8.7.2 An explanation

Figure 8.5 shows the comparison between the lifetime curve and $1/(rr)$ for cc1. It can be seen that for small values of the mean set size, i.e. when rr is large, the corresponding values are almost identical since in this region, $1/(1+rr) \approx 1/(rr)$. As the mean resident set size increases, rr begins to fall. However, when rr is small, a sudden drop in rr may produce a large change in $1/(rr)$ causing a steep rise in the lifetime curve so $1/(1+rr) \rightarrow 1$.

Both the *change* and *value* of rr at which the change occurs are necessary to produce the traditional lifetime curve. This can be seen by looking at the lifetime curve for the preprocessor as well as plotting the values of rr divided by a factor of one hundred as shown in Figure 8.6. The most significant drop in rr occurs between the normalised mean resident sizes of 0.2 and 0.3. But at the end of that change rr is sufficiently large to dominate the lifetime curve and thus only a small change is seen. However, at a lower value of rr, a smaller change produces a much sharper rise in the lifetime curve.

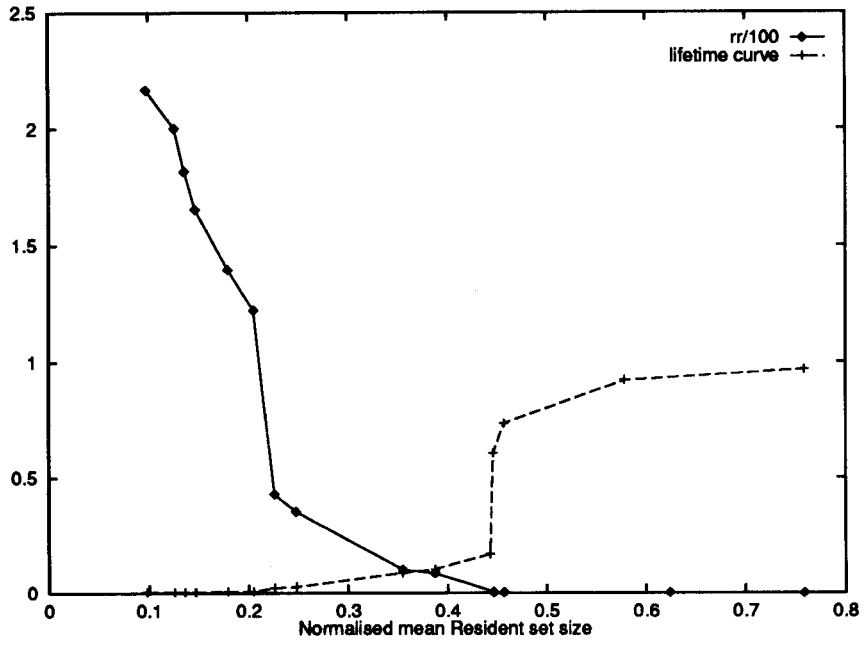

Figure 8.6: Comparing $rr/100$ and $1/(1+rr)$

8.7.3 Comparison of Different Paging Algorithms

This section compares the results for different paging algorithms operating on the text segments of the programs in the GCC Compiler Suite using the criterion of Section 8.2. The results for gcc68k are displayed in Figure 8.7 and show WS and PFF algorithms performing better than the MLRU algorithm. At the knee of the curve, PFF performs better than the WS algorithm, however, above the knee, both algorithms are identical.

The results for the preprocessor as displayed in Figure 8.8 again show that PFF and WS give better performance than MLRU, however, the knees of these curves are fairly deep with PFF performing significantly better than WS in this region. Unlike the previous graph, WS and PFF cross each other at a high value of rr. This suggests than the WS algorithm may perform better than PFF at very low values of the mean resident set size for some programs.

Figure 8.9 shows the results for the main compiler, cc1. Like the previous graph, WS and PFF cross at a high value of rr. PFF does better than WS above and below the knee of the curve. There is a sharp drop in reclaim ratio for WS between 200–250 pages. This accounts for the steep rise of its lifetime curve in this region.

The results for the assembler are shown in Figure 8.10. This graph is different from the

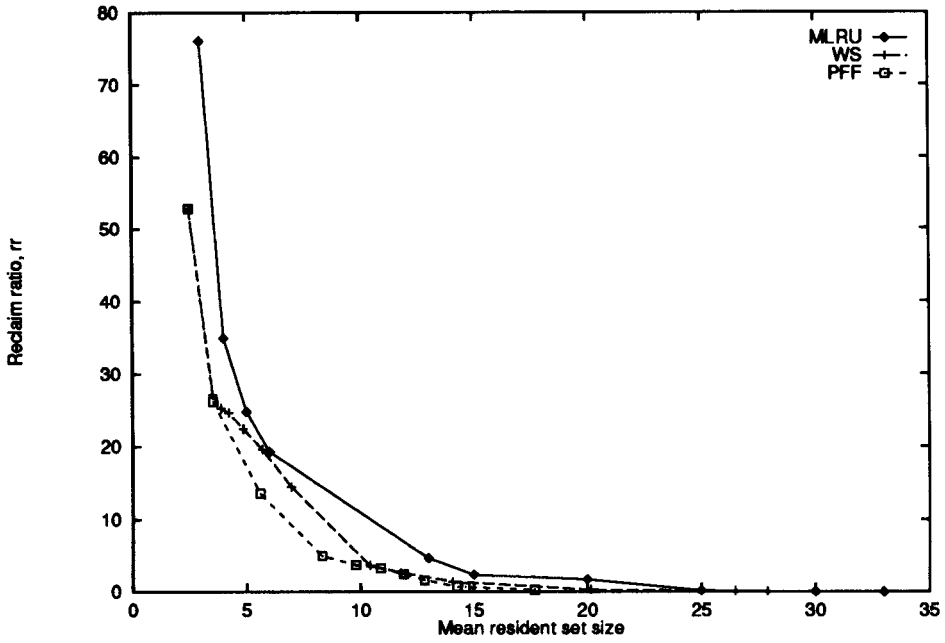

Figure 8.7: Results for the text segment of gcc68k

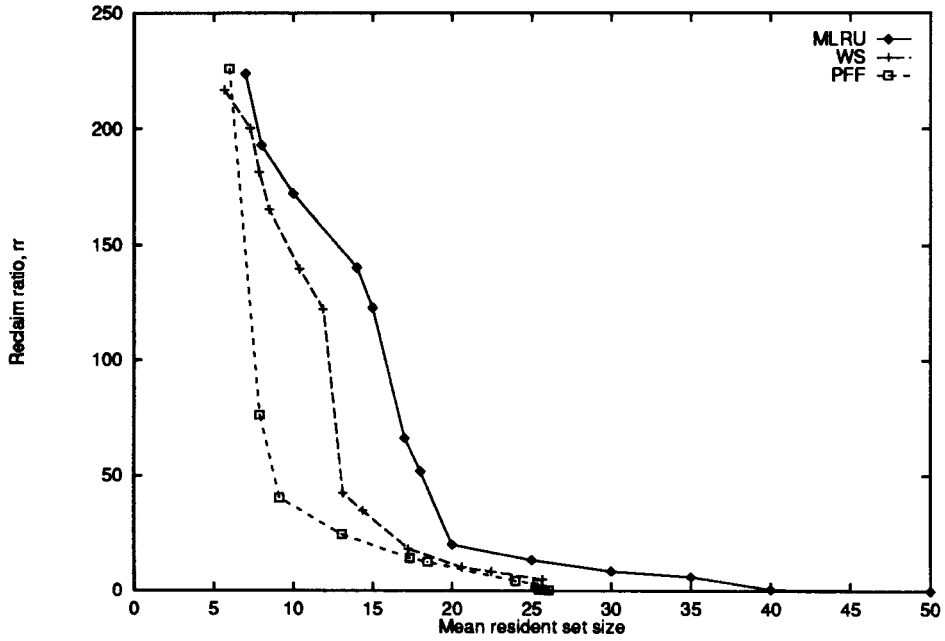

Figure 8.8: Results for the text segment of cpp

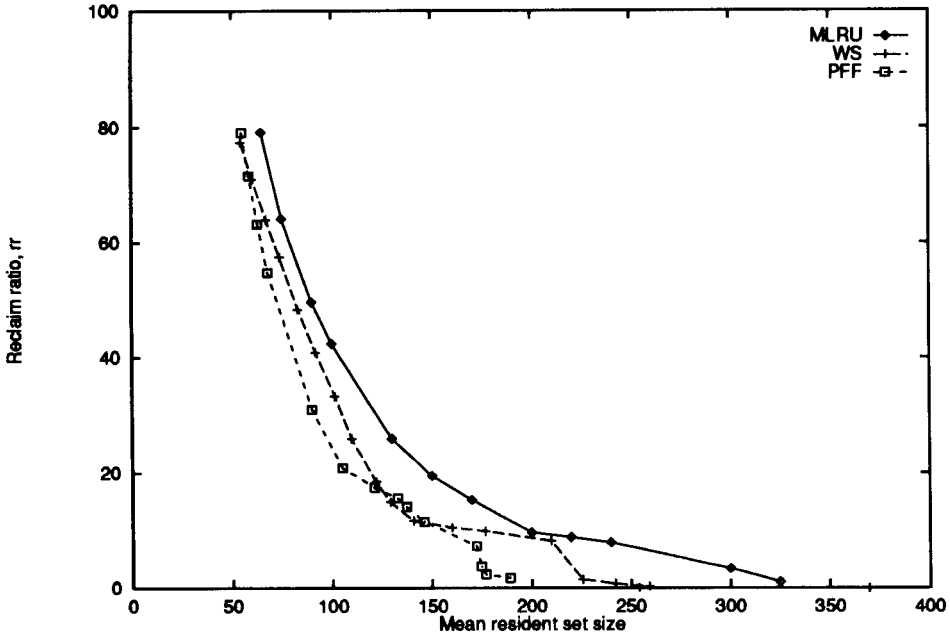

Figure 8.9: Results for the text segment of cc1

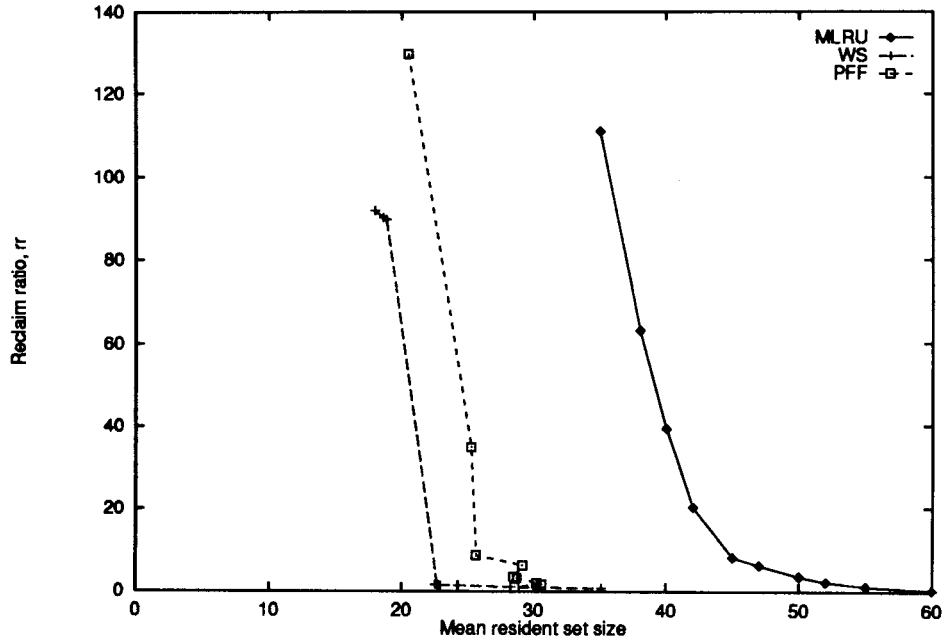

Figure 8.10: Results for the text segment of as

others. Firstly, the curves are further apart. Secondly, all the curves have fairly high gradients and finally, WS appears to perform better than PFF. The curves for WS and PFF have no "knee": very abrupt changes in the value of rr occur between the mean resident set sizes of 20–27 pages. This suggests that the assembler spends most of its time executing in a small number of phases. If the number of pages allocated is not sufficient to contain these phases, a high reclaim ratio rr is obtained. If however, the phases are contained within the resident set, then a much lower reclaim ratio is obtained. An examination of the curve for PFF reveals the anomalous behaviour of the algorithm at lower values of rr. At these points, a higher value of rr and thus a higher pagefault rate is observed for a greater mean resident set size as discussed in Section 5.2.

8.7.4 Conclusions

In the light of these results, several inferences can be made. Firstly, there is no universal model for traditional lifetime curves. A more useful lifetime curve can be obtained by plotting the $1/(1+rr)$ against m. For some programs the shape of this curve may resemble traditional lifetime curves but the exact shape is determined by the relationship between the reclaim ratio, rr, and the mean resident set size, m. Traditional lifetime curve shapes are associated with sharp falls in rr with respect to the mean resident set size when the value of rr is small.

Few general trends can be observed. In all the situations explored, WS and PFF perform better than MLRU. In some situations, WS performs better than PFF and in others the reverse is true. Overall, the results highlight the need for paging models and descriptions of paging behaviour to be based on the *observed* behaviour of the programs to which they will be applied. This in turn points to the need for the paging information about programs running on different architectures and different operating systems to be widely available to those wanting to model program behaviour.

8.8 Other Issues

When choosing a paging algorithm, other aspects of paging activity should be investigated. One of these is how different paging algorithms affect the overall memory management and in particular the paging traffic as pages are moved between swap space and main memory. In this regard, it is beneficial to examine the relationship between the standard deviation and the mean of the resident set size for different paging algorithms. Figure 8.11 shows the standard deviation of the resident set size for the the text segment of cc1. The graphs

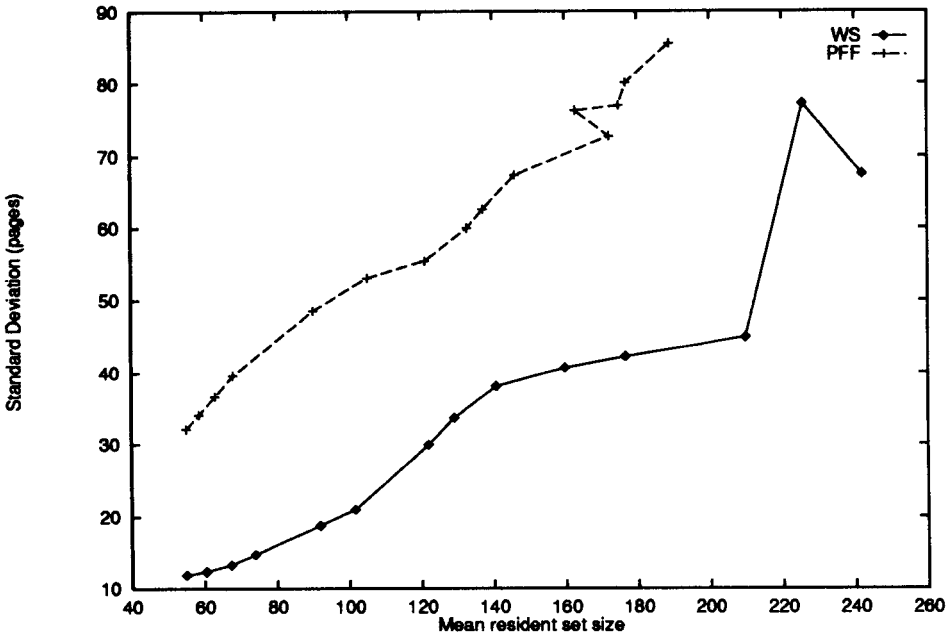

Figure 8.11: Standard Deviation vs Mean resident set size for cc1

show that the PFF algorithm has a higher standard deviation for the same mean resident set size than the WS algorithm. MLRU has a standard deviation of zero throughout. This suggests that PFF generates more paging traffic than WS or MLRU as the mean resident set size is changing in a more dynamic manner. Curves for gcc68k and cpp also show the same trend.

However, as shown in Figure 8.12, the curves for the assembler do not follow this pattern. The curve for WS drops sharply as the mean resident set size decreases from 24 to 19 pages. This corresponds to the sudden rise in the reclaim ratio when it was plotted against the mean resident set size for the assembler as shown in Figure 8.10. The low values of the standard deviation for small values of the mean resident set size indicate that the assembler operates in phases with high locality. This result suggests that the standard deviation may be affected in a complementary way to the reclaim ratio for PFF and WS.

8.8.1 Service Time Issues

A major issue in choosing a paging algorithm is the time required to service a pagefault. In this implementation, this is the time taken to execute the paging algorithm, queue the pages that have been paged out on the swap queue, signal the Pager, get the required page

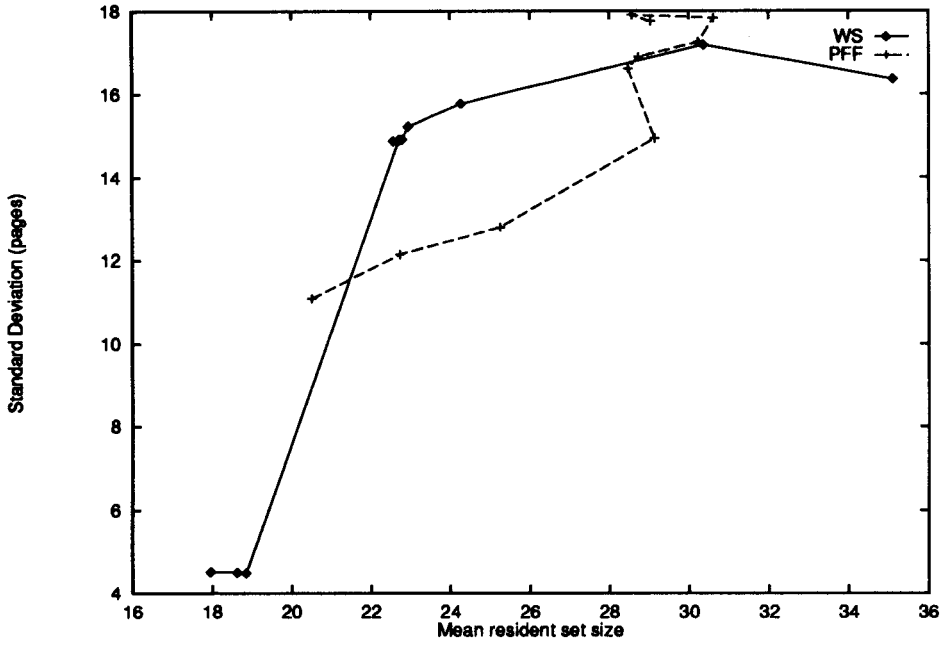

Figure 8.12: Standard Deviation vs Mean resident set size for as

and add it to the resident set. If it is an initial fault, then the PageSvr must fetch the page. However, for a reclaim fault, the page is retrieved by the user process. This involves taking the page from the reclaim queue, mapping it into the pagetables, and adding it to the resident set. The service time for reclaim faults are of particular interest since these faults are caused by the paging algorithm. This serves as a good basis for comparison between the cost of different paging algorithms.

Figure 8.13 shows the service time for reclaim faults for the text segment of cc1. A straight line for MLRU and the curves for PFF and WS show a straight line for a large region of the mean resident set size. Similar results were obtained for gcc68k, cpp and as. The reason for these results is that as the working set gets larger, more time is spent in the paging algorithm since the list of resident pages is longer and each page descriptor is examined. The parameters of the service time are presented in Table 8.2.

MLRU has the highest gradient which suggests that the algorithm is expensive to implement. PFF was found to be the least expensive, performing better than the WS algorithm. The fact that the service time is found to be proportional to the size of the working set is significant since paging models have not shown any relationship between the service time and the mean resident set size.

The result is also significant for choosing the optimal operating point on the lifetime curve.

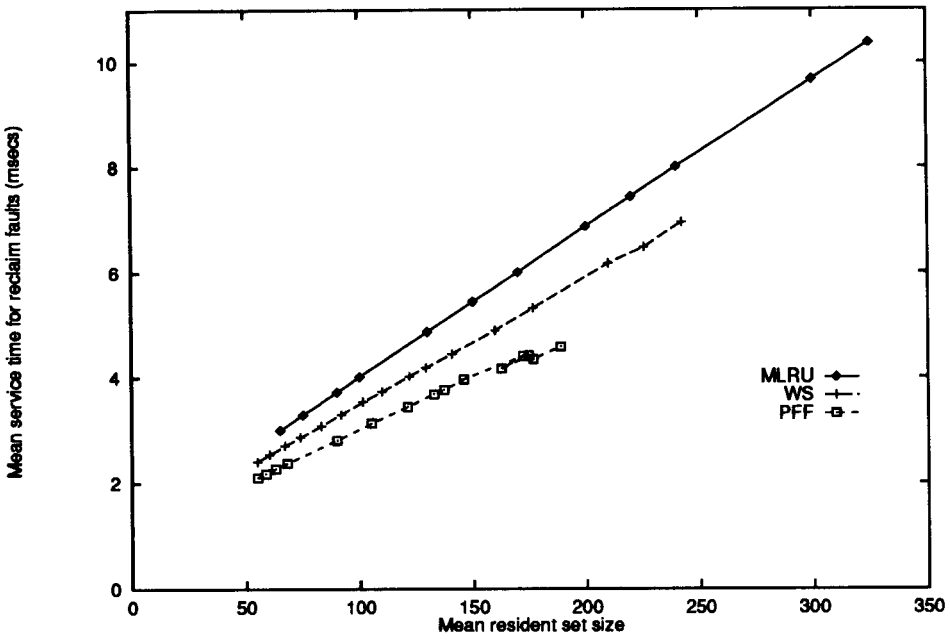

Figure 8.13: Service Time Distribution for reclaim faults for cc1

Algorithm	Cost per page (μsecs)	Constant Overhead (μsecs)
MLRU	28.54	1161
WS	23.79	1100
PFF	20.48	976

Table 8.2: Service Time Parameters for Different Paging Algorithms

Since the optimal resident set size is usually chosen using a point on the knee of the curve, the mean resident set size is likely to be large. Thus as the graph shows the service time is likely to be high, e.g. a few milliseconds. Traditionally, this did not matter since the cost of moving data to and from the disk, e.g. 25–40 msecs, far outweighed the cost of any paging algorithm. However, if the movement to and from swap space is now much faster, then the cost of the paging algorithm at a given point of operation becomes a key parameter that must be considered since it may be possible to operate at points where the reclaim ratio is high if the service time is also significantly smaller and get better performance than operating at the knee of the curve. Hence the service time must be taken into account in determining the optimal resident set size.

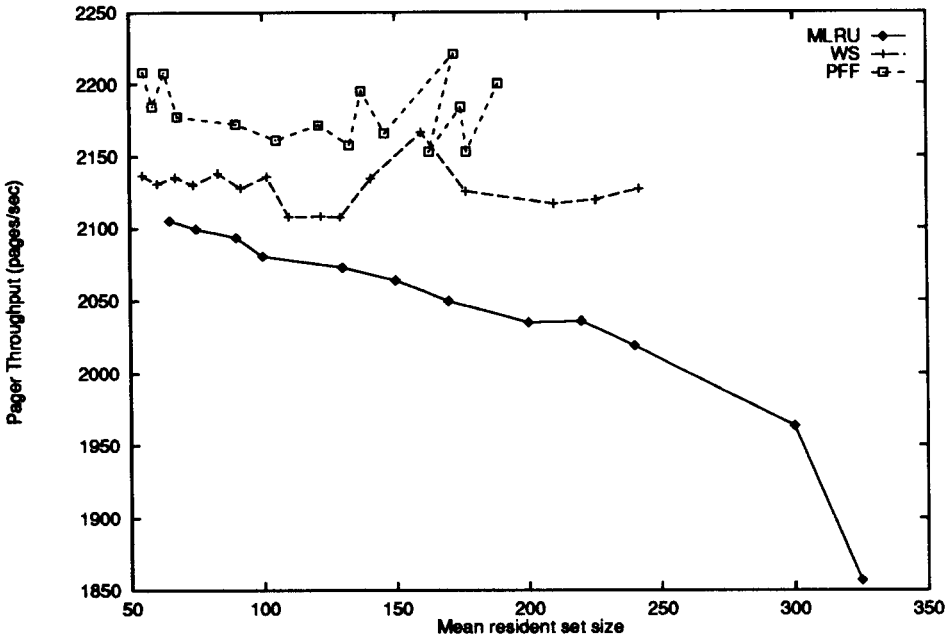

Figure 8.14: Pager Throughput vs Mean Resident Set Size

8.8.2 Performance of the Pager

The above discussion indicates the importance of accurately determining the throughput of the paging operation. Information on the paging operation was gathered on a per object basis and comprised the total time that the Pager spent moving pages to swap space, the total number of pages actually moved to swap space and the total number of visits the Pager made to the object.

From these parameters, the throughput of the Pager on a given object and the average number of pages that the Pager paged out per visit were derived. Figure 8.14 shows the throughput maintained by the Pager for all three paging algorithms for the text segment of cc1. It shows that the PFF algorithm gives the highest throughput, with an average of about 2180 pages/sec. This represents an average time of 458.7 μsecs to move a page to the non-volatile swap area and compares favourably with swap times of 25–40 msecs per page for disk. Thus using RAM (volatile or non-volatile) as swap space can result in a speedup of between 55–88 times compared to using disk storage.

WS, though having a slightly lower throughput than PFF, maintained an average of 2128 pages/sec. The MLRU curve is different since it drops sharply as the mean resident set size increases. This is because the Pager is running at user priority. Thus the user program invokes the paging algorithm several times before the Pager is allowed to execute.

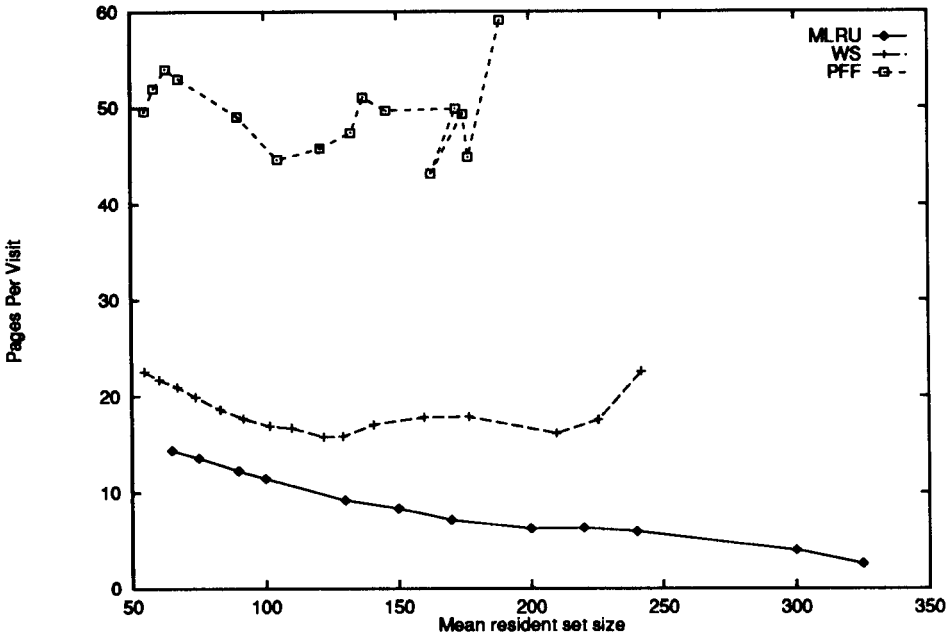

Figure 8.15: Page Per Visit vs Mean Resident Set Size

MLRU benefits most from running the Pager at user priority since only one page is removed every time the algorithm is invoked. However, when its paging limit is small, the algorithm is invoked more frequently and thus the Pager finds more pages on the swap queue, resulting in an increase in throughput. This can be seen in Figure 8.15, where the number of pages per visit as seen by the Pager increases for MLRU as the limit on the number of pages in the resident set is lowered. This result validates the idea that running the Pager at user priority can improve the throughput of the paging operation.

8.9 Other Objects

The results presented so far are concerned with the text segments of the programs in the Compiler Suite. Results presented in this section concern the data, bss and file object types. Text segments are usually the biggest segments of most programs, this is also true for the GCC Compiler Suite, and bss, and data segments are usually smaller in comparison, so results also reflect the paging of smaller objects.

8.9.1 Data Objects

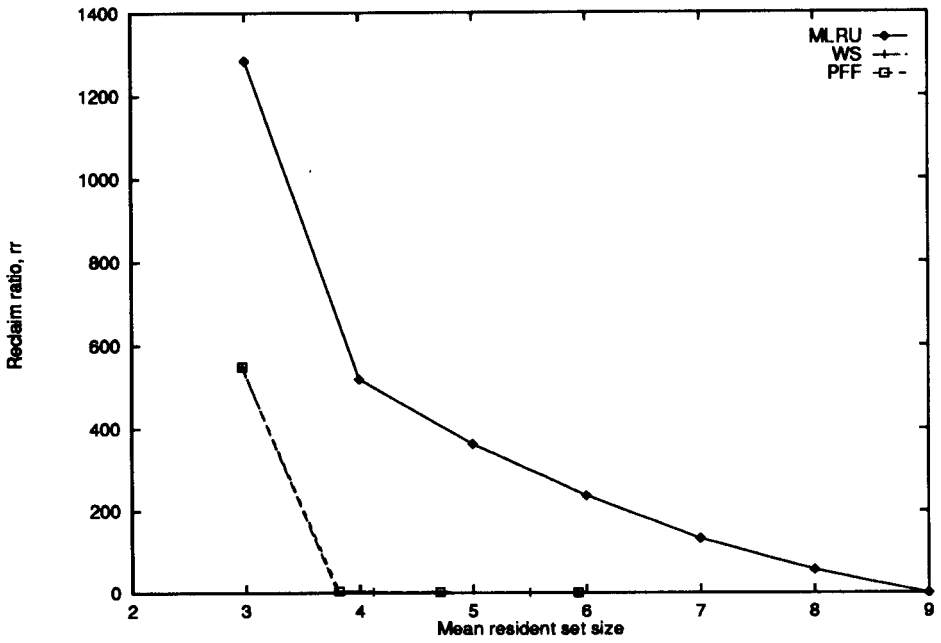

Figure 8.16: Results for the data segment of cc1

Figure 8.16 shows the results of paging algorithms on the data segment of cc1. Firstly, extremely high values of rr are involved. This is because data segments, though smaller than text segments, are referenced just as frequently. There is no difference between the results of the WS and PFF algorithms. This suggests the small sizes of the data objects and their access patterns reduced the performance differences between WS and PFF.

The results for cpp are displayed in Figure 8.17. It shows higher values of the reclaim ratio for MLRU when compared with the results for the text segment. Both PFF and WS are represented by the same co-ordinate on the graph, i.e. (3,0).

Whereas it is right to assume that these pages were always in use when the algorithm was invoked, it should be noted that the algorithms were only invoked on *three* occasions because the size of the segment was 3 KB. As a result, it is possible that the pages in the resident set will always have their used bits set when pagefaults occur. This is particularly true for very small objects that are frequently accessed, like data segments. Since PFF and WS only remove pages when they are not in use, if the number of pagefaults is small, it is likely that all the pages will remain in memory without any page ever being swapped out, hence the value of the co-ordinate. This represents a *limitation* in the use of paging algorithms, like PFF and WS, on very small objects. In these circumstances therefore it is best not to implement any paging algorithm on these objects since the effect will be negligible for algorithms like PFF and WS, or bad, e.g. MLRU.

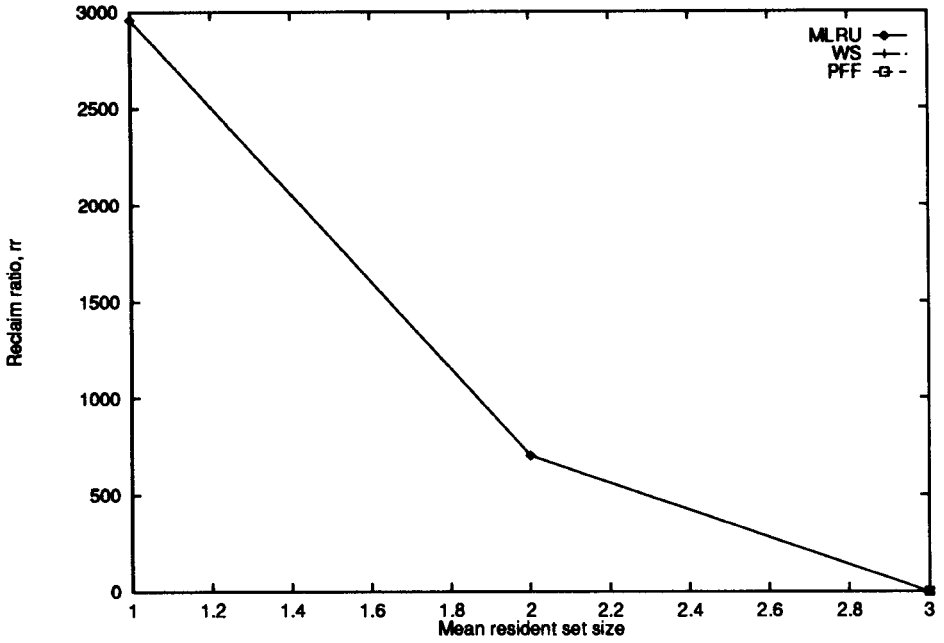

Figure 8.17: Results for the data segment of cpp

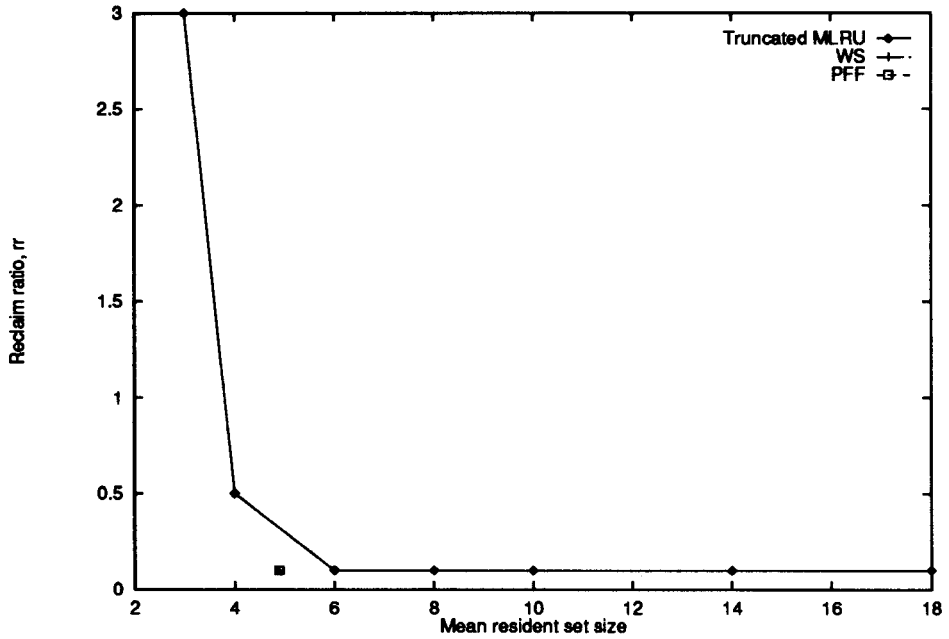

Figure 8.18: Truncated Results for the data segment of as

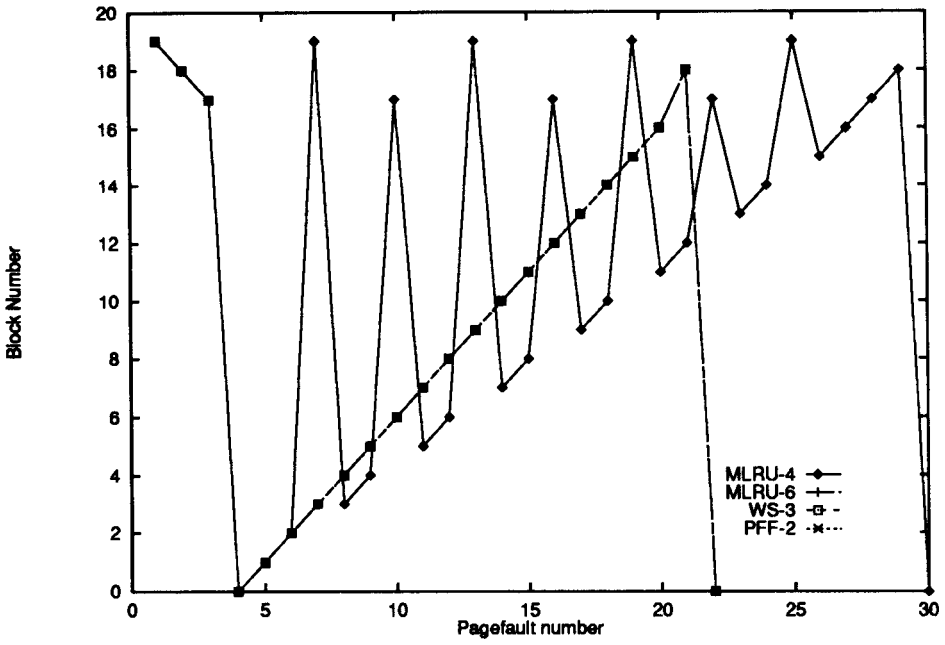

Figure 8.19: Block Distribution Curves for the data segment of as

Figure 8.18 shows the results for the data segments for the assembler. The values of reclaim ratio, rr, for MLRU at lower values of the mean resident set are not shown in order to highlight the value obtained for WS and PFF. The reclaim ratio for MLRU remains constant for most values of the resident set size and only begins to increase as the mean resident set size falls. The results for PFF and WS are represented by a single point which is obtained over a wide range of the control parameters.

To investigate this result further, the **block distribution** or bd curve of the data segment of *as* at various points was obtained. The bd curve indicates the order in which the pages of an object are accessed and is obtained from the data acquired using the testbed. The block distribution curve for MLRU with page limits of 4 and 6 as well as WS with $\theta = 3msecs$ and PFF with $\theta = 2msecs$ are shown in Figure 8.19. MLRU-6, WS-3 and PFF-2 are represented by the same curve. This curve indicates that only two blocks are reclaimed. These are block 18 and block 0.

The degree of *sequential access* displayed by the curve is surprising. This curve may be compared with that obtained for MRLU-4. In the latter, blocks 19 and 17 are continuously reclaimed as the program executes, which suggests that most accesses to the segment are made to these blocks. Under PFF and WS, these two blocks would not be paged out because they are continuously used. Blocks 18 and 0 are referenced again after a long time, so would have been paged out and must be brought into memory. This explains why

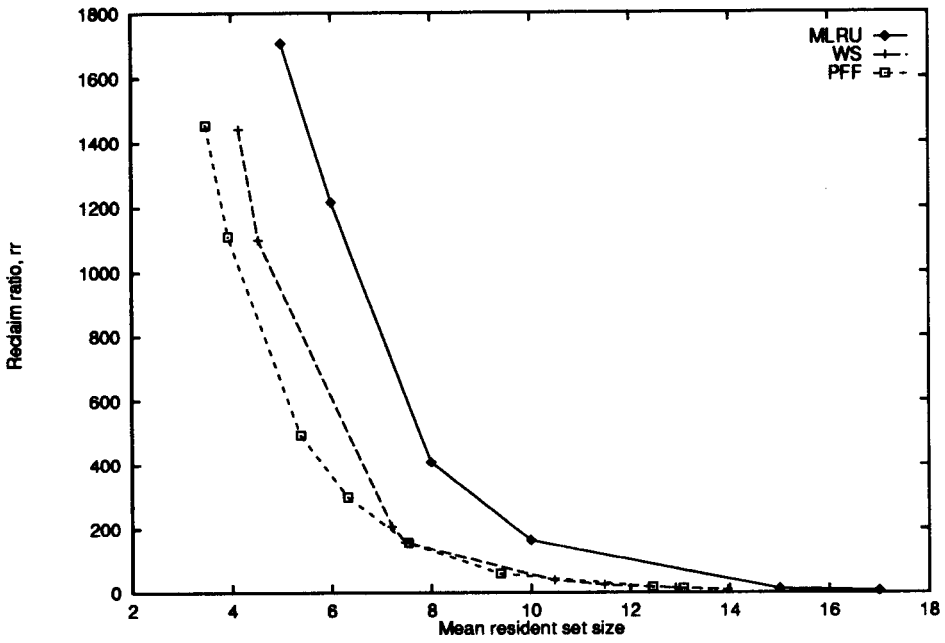

Figure 8.20: Results for the bss segment of cpp

WS and PFF are represented by the same point in Figure 8.19 and why the curve is also constant for MLRU over a large region of the mean resident set size. The reclaim ratio is 0.1 since only two reclaim faults occur for an object of 20 pages.

8.9.2 Bss Objects

The size of bss objects can be smaller or greater than the size of data objects thus bss segments may display the same characteristics of data segments. Firstly, the results for the bss segment of cpp, the preprocessor are shown in Figure 8.20. The reclaim ratios are extremely high and are comparable to those obtained for the data segment of cpp in Figure 8.17. The results are for bss segment of cc1 are shown in Figure 8.21. The reclaim ratios for the bss segment of cc1 are much lower than those of the data segment as shown in Figure 8.16. This suggests that the access patterns of the two objects are very different.

8.9.3 File Objects

It was decided to monitor the paging activity of several files that were accessed during compilation: the source file, user_types.c, the intermediate files test.cpp and test.s and the

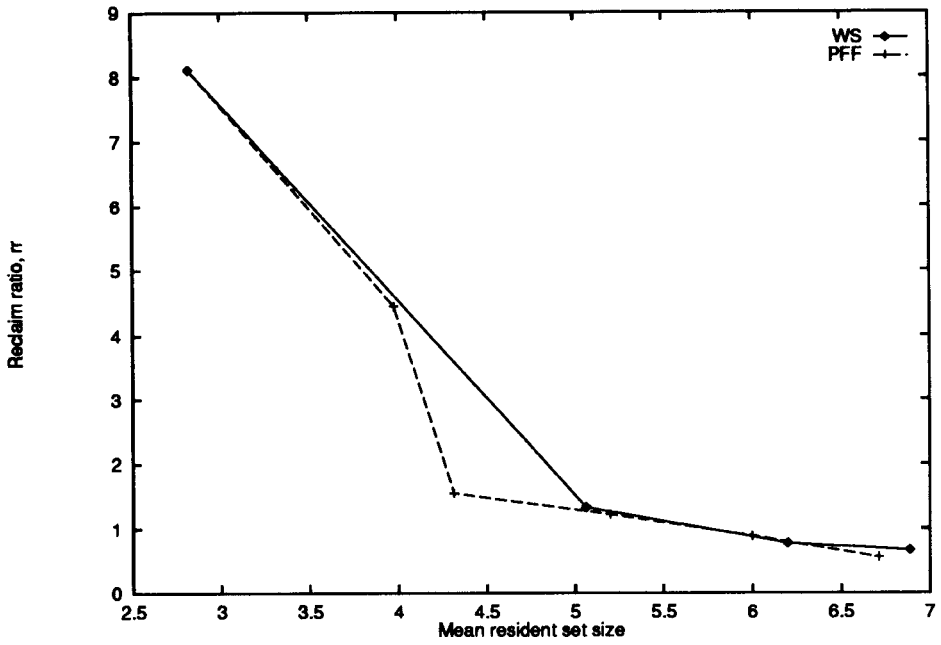

Figure 8.21: Results for the bss segment of cc1

object file, user_types.o. Figure 8.22 shows the results for the source file user_types.c. It shows that all algorithms and all control parameter values yield a reclaim ratio of zero. Similar results are obtained for the output file, user_types.o. The shape of the graph may be explained by plotting block distribution curves for the two files. These are shown in Figure 8.23. The graph shows that both the source and output files are accessed serially. So once pages are referenced they are not referenced again, hence the rr is always zero. This is *independent* of the paging algorithm being used.

The results for the intermediate file, test.cpp, are shown in Figure 8.24. The graph shows a reclaim ratio of 1 for MLRU, PFF and WS and for all the mean resident set sizes. Similar results are also obtained for test.s. This is because the intermediate files are accessed by two programs. The file test.cpp is created by the preprocessor and then accessed by the main compiler, cc1; while test.s is created by the cc1 and passed to the assembler. Each file is kept in memory until it is requested by the second program.

Both programs have the same sequential access pattern as shown in Figure 8.25. Hence it can be inferred that the preprocessor wrote to the file sequentially. The paging algorithms deallocated pages of lower block numbers as the file grew longer. When the compiler started to read the file, the pages with the lower block numbers were reclaimed while the higher block numbers were removed only to be reclaimed again as the compiler reached to the end of the file. This led to all the blocks in the file being reclaimed once hence the

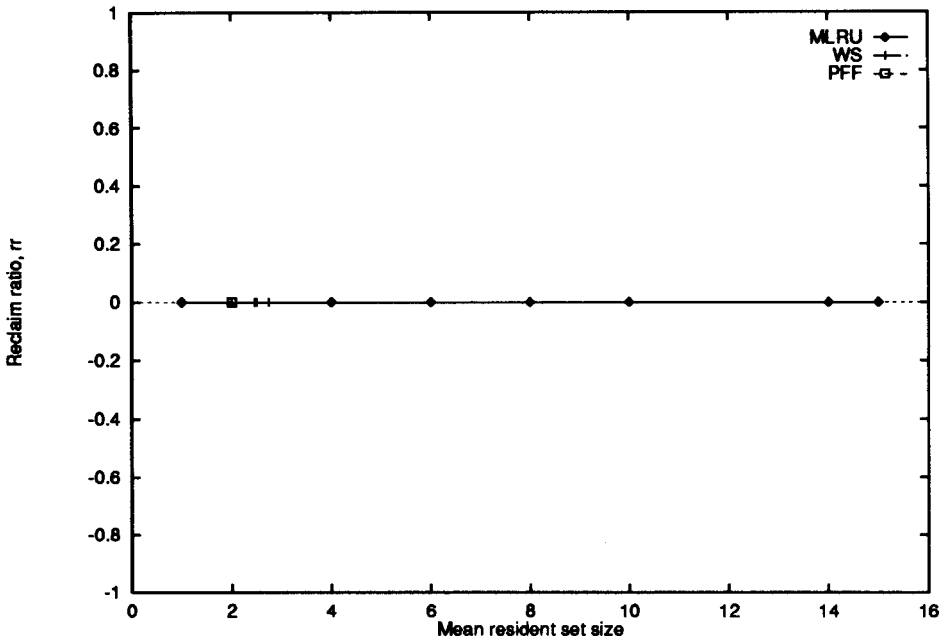

Figure 8.22: Results for user_types.c

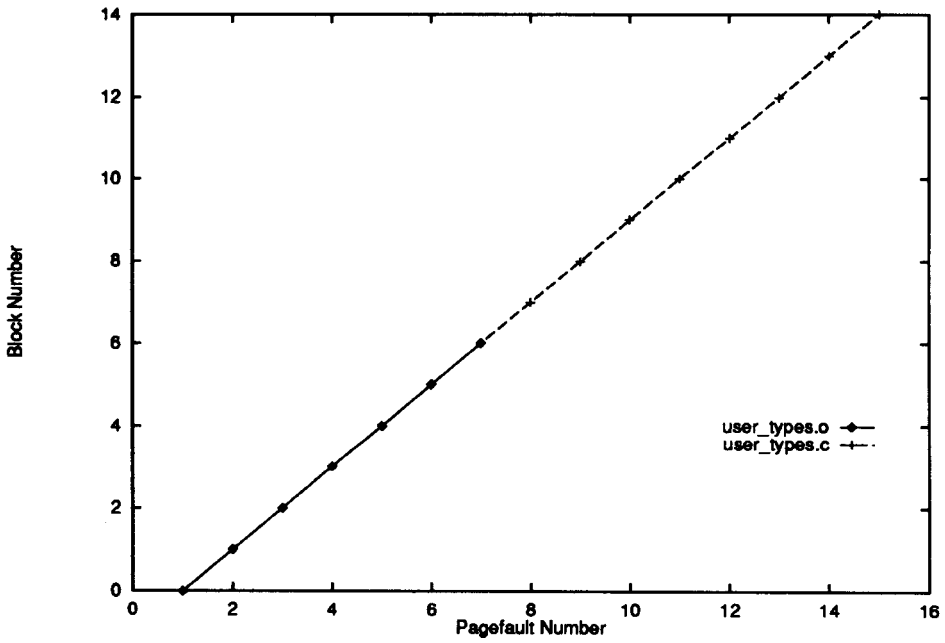

Figure 8.23: Block Distribution Curves for user_types.o and user_types.c

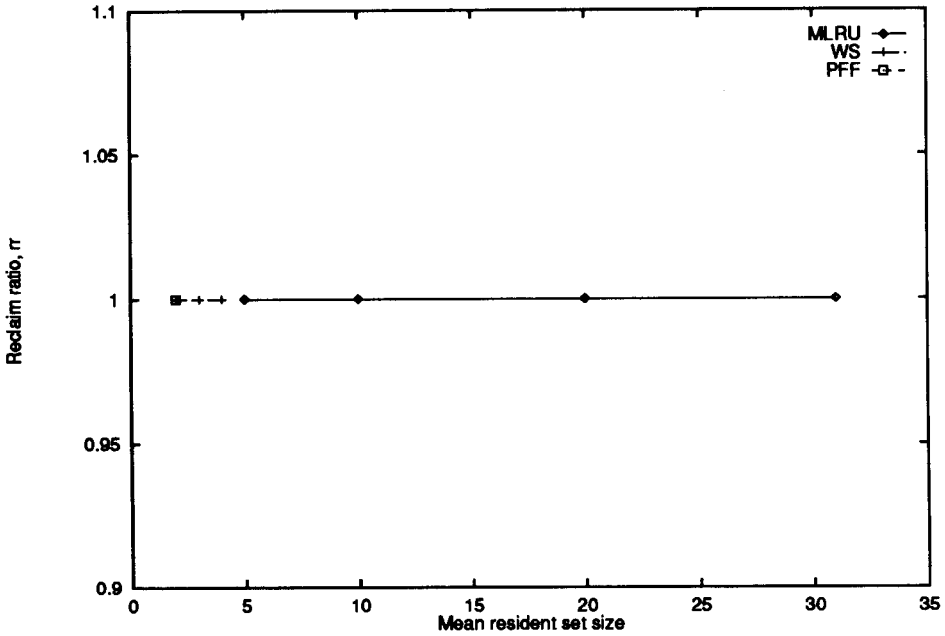

Figure 8.24: Results for the intermediate file, test.cpp

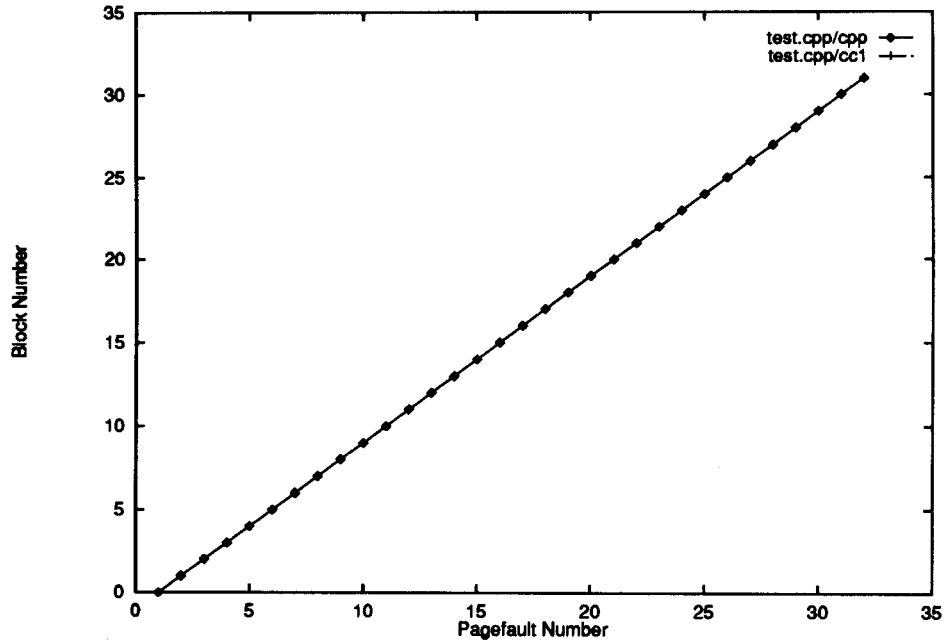

Figure 8.25: Block Distribution Curves for test.cpp

value of the reclaim ratio, rr.

8.10 Summary and Conclusions

In this chapter, several aspects of the implementation and evaluation of conventional paging algorithms were examined. It was shown that more useful lifetime curves could be obtained using $1/(1 + rr)$. This approach is superior to traditional lifetime curves. The reclaim ratio, rr, can be measured more accurately and the result eliminates the need to measure the interpagefault times.

The paging interface is an extension of the interface described in Chapter 6. It allows users to specify the paging algorithm that can be used on an object before that object is created. This allows users to experiment with different paging algorithms for a given object until they find one that they think is most suitable.

As mentioned previously, the remote paging facility was extremely helpful in the testing of various paging algorithm on objects. It is also relevant in the microkernel context where many services are implemented by user-level processes. It may be possible to page several of these servers using paging algorithms which are specified in a remote file that is read as the system is started. An object manager may also be paged if its operation is not dependent on object(s) for which it is responsible. For example, it is not possible to page the PageSvr since it manages text, data and other objects which are needed to start any process including PageSvr itself. However, it may be possible to page a video object manager since it is unlikely to use video objects as part of its operation.

The testbed developed in this chapter provides a number of features which make it a powerful tool in the analysis of paging algorithms. For example, it can accurately measure the service time for pagefaults and thus gives us a direct method for comparing the cost of different algorithms. Features like the block distribution curve can be used to investigate the order in which objects were accessed and thus provide more detailed information on how different paging algorithms behave.

The GCC Compiler Suite was used to investigate the validity of paging models and to obtain data on the performance of MLRU, WS and PFF. The results invalidated the claim of a universal paging model. They also indicated that the standard deviation, the service time and the paging throughput should be also be considered when evaluating the performance of a paging algorithm. Faster swapping times would affect the operating parameters of paging algorithms and have a beneficial effect on overall system performance.

Of the algorithms examined, WS and PFF performed better than MLRU in all cases. PFF was also better than WS in most, but not all, cases involving the paging of text segments. It also showed anomalous behaviour in some of the curves that were presented. Data segments and bss segments generally have higher reclaim ratios than text segments. MLRU performed particularly badly compared to WS and PFF for data and bss segments. For very small segments, WS and PFF had very little impact since pages were not swapped out. However, since these segments were frequently accessed, no paging algorithms should be used on small data and bss objects.

All the algorithms examined had very little impact on file objects because the files were being accessed sequentially. This result could be expected since these algorithms were developed for program segments. New algorithms are therefore required to optimise the memory use for other types of objects. The design of new paging algorithms for objects with highly sequential access patterns is the subject of the next chapter.

Chapter 9

New Paging Algorithms

9.1 Introduction

As shown in the previous chapter, conventional paging algorithms, while performing well on traditional program segments, have little impact on objects whose access patterns are radically different from program segments. Some files, for example, have access patterns which show a very high degree of sequential access. Access patterns of database systems are also characterised by high sequential access and weak locality [**Rodriguez-Rosell76**], [**Smith78b**]. Some common devices may only be accessed in a sequential manner, for example, reading input from a keyboard or writing output to a display. Voice and video devices also produce large amounts of data that are usually processed in a sequential manner.

In Unix System V, [**Bach86**, pages 344–354] devices may be accessed using a **stream** interface. A stream [**Ritchie84**] is a full duplex connection between a device driver and the file descriptor which represents the device to the user. It consists of two unidirectional queues which move data in opposite directions. When the user wishes to read data from the device, the Unix I/O read call is invoked since Unix treats physical devices as special files. This causes data to be moved from the device into the input queue. When the program writes to the device, data is placed in the output queue where it eventually reaches the head of the stream and is written onto the device. This arrangement is shown in Figure 9.1.

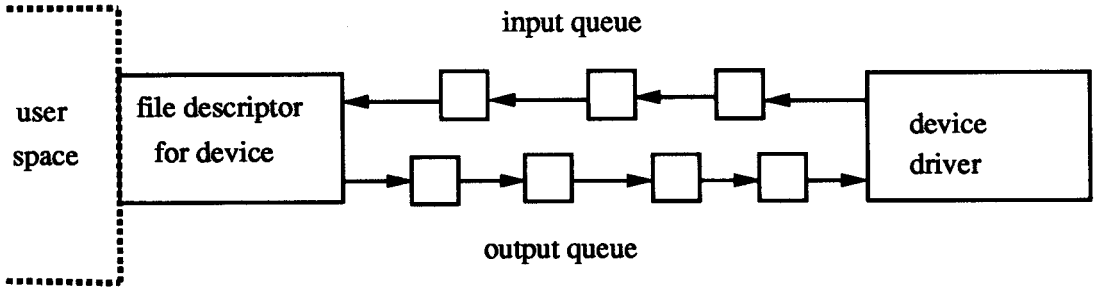

Figure 9.1: A Unix Stream

Streams have several advantages over other access mechanisms including the fact that it can easily be applied to support a large number of diverse devices. They are also useful since they buffer data between the file interface and the device driver. Thus, more data than that requested by the user may be placed in the input queue so that the user would not block until the stream is empty. Another important advantage of streams is that a number of modules may be placed in the path of the stream to implement features like line disciplines for terminal management, or protocol processing. These modules therefore can be readily tailored to specific computing environments.

From a memory-management context, one of the main disadvantages of the stream interface is that, like the Unix file interface, it is not integrated into the virtual memory management of the system. This integration is desirable for objects that are usually larger than the system pagesize, e.g. files. Such integration would require designing paging algorithms which differ from conventional ones.

With traditional paging schemes, when a pagefault occurs the user is blocked until the page or page(s) are brought into memory. Modified pages are written back to secondary storage when all users are finished using the object or a user invokes *flush* operations on the object.

In this chapter, new paging algorithms are designed that incorporate some of the ideas behind the stream concept. These algorithms are tested and results are presented and compared with traditional paging systems.

9.2 Design Issues

Prefetching

Since the objects concerned are accessed in a highly sequential manner, prefetching several pages when a pagefault occurs would increase the overall system performance [**Lau82**]. This gives an even greater performance improvement in a distributed context since only one RPC is required to fetch several pages rather than one RPC per page. Prefetching is also beneficial where a large quantity of data is involved and the unit of access is much greater than the system pagesize so several pages must be mapped in before any useful work can be accomplished.

Buffering

Buffering may be readily applied to highly sequential access. The main advantage of buffering is to allow the user to continue execution while new pages are being fetched. To implement this using object-oriented virtual memory, it is necessary to associate a queue of pages called a **buff** queue with the object data structure.

When an object is opened, the buff queue is filled with the first few pages of the object. When a pagefault occurs as the object is first accessed, pages in the buff queue are mapped into the resident set. The user then invokes a routine that causes the object manager to refill the buff queue and returns to user-space.

Two main side-effects of this scheme must now be considered. The first occurs when the user consumes pages faster than the object manager is able to bring them into memory. Thus it is possible for the user to find that the buff queue is empty. The solution adopted here is that the user blocks by invoking a **WaitOnStream** routine. This routine queues the user on the threadlist associated with the page that caused the pagefault. The user is unblocked by the object manager after the buff queue is filled.

The other problem occurs when the user jumps to another place in the object. For example, if a file is rewound. Here, pages in the buff queue are immediately deallocated and the user invokes a routine that forces the buff queue to be filled starting from the page that caused the pagefault. When this is done, these pages are mapped into memory and the user then calls the object manager to refill the buff queue before returning to user space.

Quick Deallocation

For pages that are being accessed sequentially, once they have been referenced, they may not be referenced again for some time or not at all. These pages can therefore be immediately removed from the resident set. Since the paging algorithm is invoked when a pagefault occurs or when prefetched pages are added to the resident set, it is necessary to differentiate between the need to deallocate all the pages and to take no action since pages representing another part of the object are being loaded.

To distinguish between these two situations it is necessary to check the page descriptors of the resident set. If any of the used bits are set, it indicates that some pages have been referenced and thus the entire resident set is deallocated. If no used bits are set this indicates that the resident set is being filled with new pages.

Immediate Writeback

Once a process has written to a set of pages associated with sequentially accessed objects, they are rarely written to again. Thus the changes can be immediately written back to secondary storage in addition to being put on local swap space. This eliminates the need for most applications to continuously use *flush* operations to force data to the disk since if the site crashes only changes to the pages in the resident set will be lost.

A key issue here is the need to stop the PageStealer from removing the pages from the reclaim queue before pages are written out. Thus a **streamind** variable was added to the object data structure which, when set, indicates that a stream write-out is taking place so the PageStealer would not remove pages from the reclaim queue until the respective pages are written out.

We can therefore represent this sequence of events graphically as shown in Figure 9.2. Pages are read from secondary storage into a buffer queue where they are held until loaded into the resident set when a pagefault occurs. Once the pages have been deallocated they are immediately written back to secondary storage. It is easy to link the writeout queue of one process with the buff queue of another to produce a cascading effect, thus allowing other modules to be inserted into the stream.

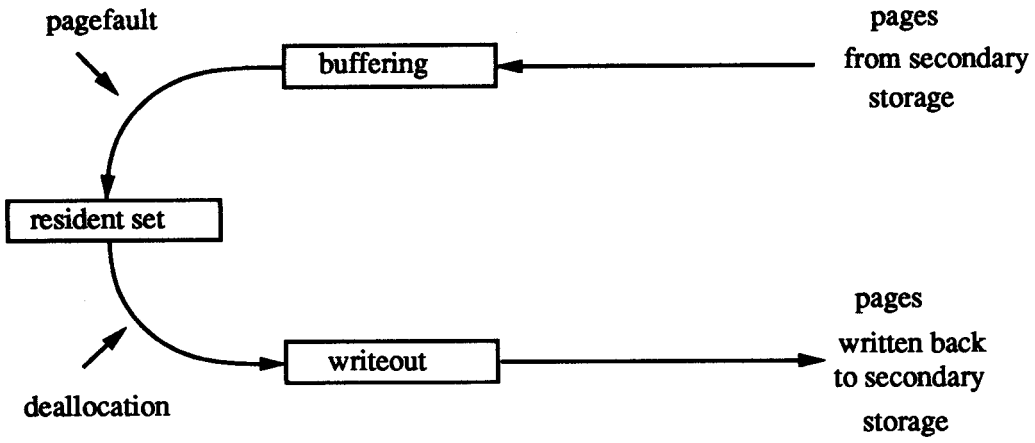

Figure 9.2: Paging for objects with highly sequential access patterns

9.3 Classification and Testing

In order to test the benefits of the individual components of this scheme, it is necessary to groups different aspects into different paging algorithms. Also, some aspects may be relevant to some objects and not to others. For example, a read-only object will not have any modified pages to write out. The proposed algorithms are:

SimpleStream Paging Algorithm or SSPA – this includes prefetching and deallocation.

BufStream Paging Algorithm or BSPA – this comprises prefetching, deallocation and buffering.

WriteStream Paging Algorithm or WSPA – contains all four features.

Testing

A database program was used to test these algorithms. SSPA, BSPA and WSPA are compared with the conventional algorithms whose operations have been enhanced by allowing them to prefetch a given number of pages when a page is first referenced, i.e. when an initial fault occurs. When a reclaim fault occurs, these algorithms operate in their traditional manner. The control parameters of these algorithms are set to ensure a quick deallocation of pages. For example, consider MLRU: if the total number of pages to be acquired when a pagefault occurs is set to n and the maximum number of pages allowed is also set to

n, this means that all the pages that were previously accessed will be deallocated as the prefetched pages are added to the resident set. For PFF and WS, it suffices to set the control parameters to very small values, e.g. 5 μsecs, forcing pages that were recently accessed to be deallocated.

The database accessed during testing is 189,812 bytes long and contains 10,799 records. It was built by one of the analytical routines for the analysis of paging results obtained from the experiments in the previous chapter and is used to calculate the average interpagefault time for all the pagefaults on an object as well as the average interpagefault times for initial faults and the reclaim faults separately. Hence calculations are performed using all the records in the database.

The test program first opened the database and copied it to a temporary file. This generated initial faults on both the database and the temporary file. When this phase was completed, the database file was closed, the temporary file was rewound and the calculation of the required parameters was performed. In this operation, only reclaim faults were generated since the entire file had been referenced during copying. So it is possible to analyse the performance of the prefetching component of the algorithm by analysing the service time of the initial faults for a different number of prefetched pages and to compare the normal replacement policies with the stream algorithms by examining the reclaim faults generated as calculations are being performed. All other objects in the address space had no paging algorithm assigned to them making the paging algorithm for the files the only variable quantity. The results for the temporary file were analysed.

9.4 Results

Figure 9.3 shows the average service time for a pagefault against the number of pages fetched when an initial fault occurs. The service time increases for all the curves as the number of pages fetched increases since more pages must be taken off the free-list and placed in the resident set. When an initial fault occurs and the paging algorithm is invoked, new pages are obtained. The paging algorithm is also invoked as each prefetched page is added to the working set.

These results show that SSPA, BSPA and WSPA perform better than the traditional paging algorithms for initial faults. The BSPA algorithm performed best since it benefits from having pages assigned initially as the object was mapped into the address space. The difference between the WSPA and BSPA algorithms reflects the cost of queueing the pages to be written out as they were removed from the resident set. However, WPSA still

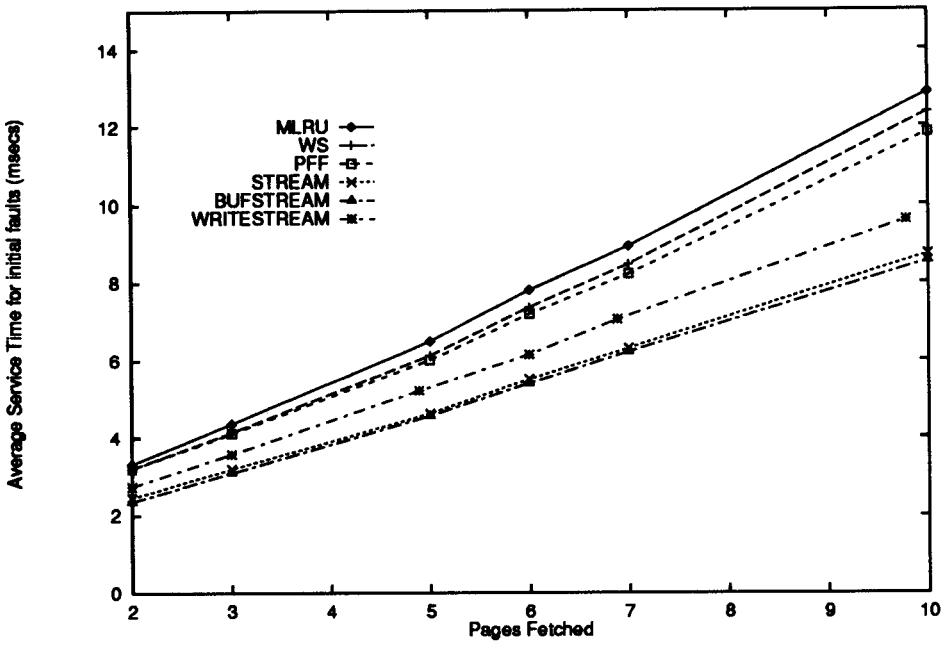

Figure 9.3: Service time for initial faults

performs much better than the traditional paging algorithms.

Figure 9.4 shows the average service time to service a reclaim fault against the number of pages that are fetched. Since it is a reclaim fault, no prefetching is done in the traditional algorithms. In the case of the SSPA, BSPA and WSPA, the entire resident set is discarded and are placed on the swap queue. Pages from the next part of the stream are swapped back into memory. This accounts for the significant reduction in the service rate for these faults for the SSPA, BPSA and WSPA algorithms while WS, PFF and MLRU all increase.

Finally, Figure 9.5 shows the throughput achieved by the Pager for different paging algorithms. Again the stream algorithms perform better. This was primary due to their quick deallocation policy.

9.5 Summary and Conclusion

These results in this chapter support the theory that new paging algorithms can be designed which perform better than conventional ones in certain circumstances because they better reflect the way in which the objects are being used.

The development of the these algorithms was motivated by the results obtained in the

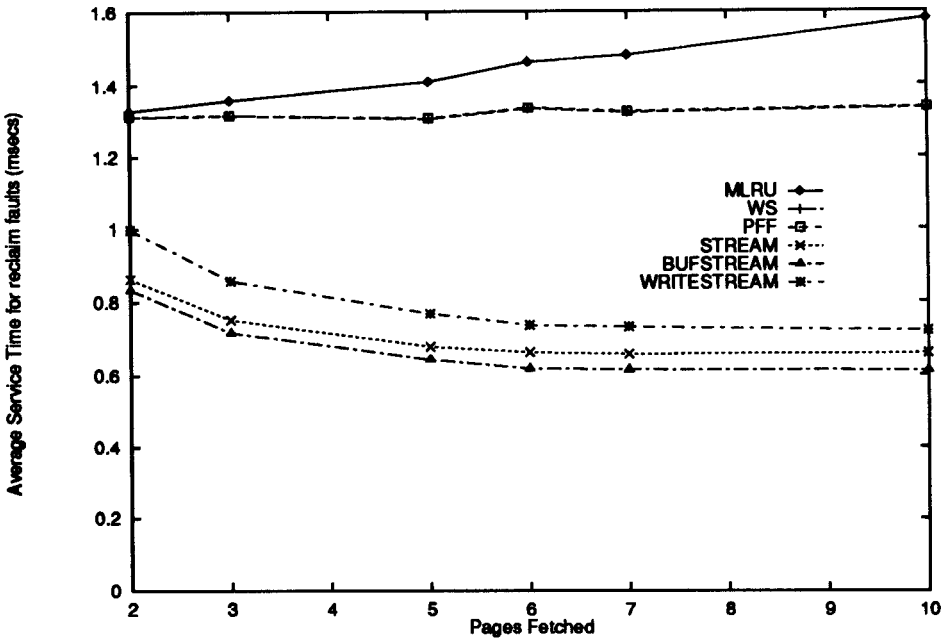

Figure 9.4: Average Service Time for **reclaim** faults

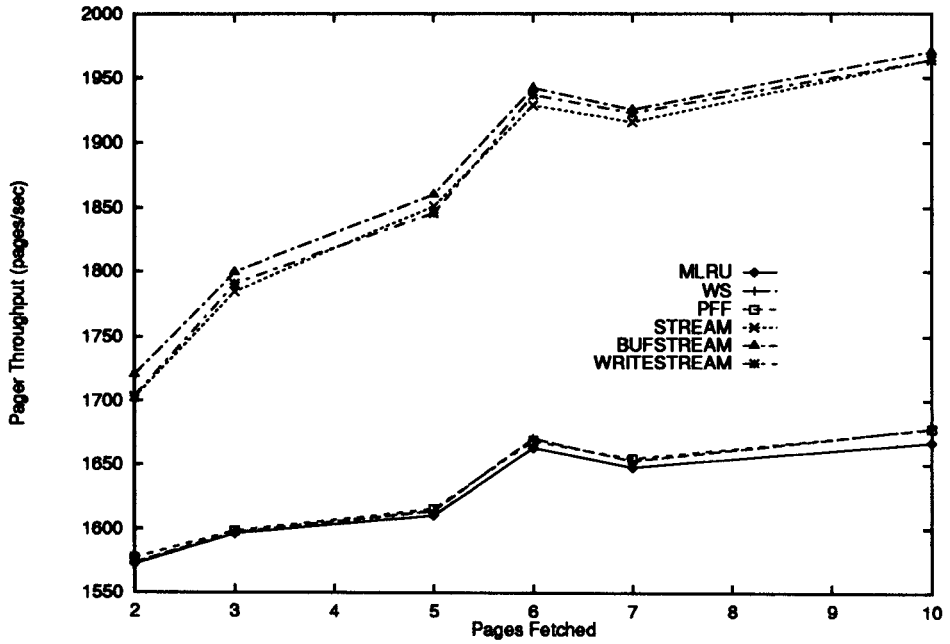

Figure 9.5: Pager Throughput

previous chapter which showed that conventional paging algorithms have little impact on files or objects that are accessed in a highly sequential manner.

These algorithms are especially applicable in the new computing environments since they will directly benefit from multiprocessor configurations where buffering, execution and writeout may be performed on separate processors thus ensuring big real-time improvements and thus they can be used to support new services such as voice and video applications.

Chapter 10

Conclusions and Further Work

10.1 Conclusions

This dissertation has explored an object-oriented approach to virtual memory management. This was done by first building an interface based on the protected process map. This novel approach allowed users to build or specify appropriate handles by which objects should be accessed. It provides the user with greater flexibility and the capability of programming in a more object-oriented manner. A type interface was also built, allowing users to decompose complex objects into their various subtypes and access these objects using their appropriate handles.

Using this interface, different paging algorithms were investigated and traditional ways of modelling program behaviour were critically examined. The results showed that the reclaim ratio, rr, had a significant impact on program behaviour and that a more relevant lifetime curve can be obtained plotting $1/(1+rr)$ versus the mean resident set size. It was also shown that there was no universal lifetime curve. Other aspects of paging activity were also shown to be significant when assessing the behaviour of paging algorithms. These factors are the standard distribution of the resident set size, the service time and the throughput at which paging operations can be achieved.

This investigation also highlighted some practical issues in the memory management area of operating system design including the use of faster swap space to improve overall system performance. Devices to meet these requirements would be faster than disk but may also

have a smaller capacity. New technologies including Wafer Scale Integration (WSI) and flash memory systems [**Pashley89**] are being targeted to fill this niche. Thus as CPU speeds increase, there will be a greater need to improve the memory hierarchy of the system to obtain better overall performance.

Paging on a per-object basis showed that while conventional localised paging algorithms like the Working Set and Page Fault Frequency algorithms perform well on text objects, their impact on data and bss segments was minimal because of the small size of these objects. With file objects, these algorithms had little impact because of the highly sequential access pattern associated with files. New algorithms were proposed and were shown to perform better than conventional paging algorithms for objects with highly sequential access patterns.

A virtual memory management system based on the support for objects of different types that can be mapped into different address spaces is an efficient system that can be easily extended. There are a number of reasons for making this assertion.

First, this approach allows paging to be done on a per-object basis. Second, most of the work of managing objects is done by object managers, freeing the kernel from having to know everything about the characteristics of different objects. A standard interface is provided to allow users to implement their own object managers. Object managers are free to implement mechanisms that would meet the Quality-of-Service or QoS requirements associated with particular objects.

The study of traditional paging algorithms clearly showed the need for providing a facility for acquiring data on real systems in order to develop a model that reflects how objects are actually used. A testbed was developed and can be used to generate data on the paging activity of various types of programs.

10.2 Further Work

10.2.1 Paging Issues

This dissertation shows that there is scope for work to be done in the area of the design and implementation of paging algorithms. It is also desirable to study in detail programs that are commonly used to optimise their performance since a lot of time is usually spent running these programs. These programs include compilers, linkers, text editors, mail handlers. Improvements in the way these programs are paged will have a proportionally greater impact on the overall productivity of end-users.

The archiving of information on how particular objects should be paged is also a possible extension to this area of research. This specification will allow the system to use paging algorithms for programs whose paging behaviour have been previously analysed to give optimal performance.

Another area for further work is an investigation into **adaptive paging algorithms** based on values of the reclaim ratio, *rr*. A program may be allowed to operate within certain values of *rr* denoted by two thresholds. If *rr* gets above the upper threshold, the resident set size is increased, while if *rr* moves below the lower threshold, the mean resident set size is reduced. This will reduce the need to obtain the parameters that will give optimal paging performance *before* the program is executed.

Finally, another way in which this work could be extended would be to investigate the need for other characteristics that can be defined on a per-object basis. For example, the ability to define the QoS parameters is particularly relevant in the multimedia environment where different streams, managed by different object managers, must be synchronised and managed in a time-related manner. This may result in specified actions by object managers, e.g. using a low-level RPC, or by the operating system, e.g. applying a fast paging algorithm, in order to meet the QoS requirements associated with a given object.

This issue is also relevant in a distributed storage system where the user should be able to specify parameters that indicate the QoS related to the storage and availability of a given object. For example, it should be possible for the user to specify a 99% availability on an object or that a given object should be replicated, etc. These issues are currently being addressed [**Lo89**], [**Wilkes89**].

10.2.2 Extending the User Interface

Perhaps the most interesting and obvious extension of this work is to provide support for persistent objects in object-oriented languages such as C++ as well as object-oriented databases.

Interfacing with Object-Oriented languages

The goal here is to allow object-oriented languages such as C++, to use the interface outlined in the previous chapters to manage objects. Thus the programmer can use a *single interface* as specified by the resultant system to access any type of object.

Objects that require persistent support may include the user interface calls as part of the definition of the object. These objects are therefore created by calling into the interface and the object may be initialised using the corresponding map_entry. Other interface calls may be invoked as part of the virtual methods associated with the object.

Support for Databases

As indicated previously, the interface presented in this dissertation can easily be extended to provide support for a lock manager which is able to provide a finer granularity of access than the synchronisation mechanism specified in the interface. To implement a database manager, it is also necessary to provide disk management routines. This can be easily achieved since Wanda already has a user interface to the disk driver.

Thus, the database manager also acts as the object manager for objects of type *database*. When a request to open a database is received, the kernel passes the request to the database manager which causes the relevant database to be mapped into the address space of the caller.

When a process wants to write to the database, the call *AcquireLock* is invoked, specifying the offset and length of the region for which access is required. If another process is using this region then the process is blocked. If not, it receives the write lock and returns to complete the transaction. Pagefaults on the database will result in requests to the database manager to satisfy faults on relevant pages accordingly. The database manager can then check to see whether the access that generated the pagefault is in accordance with its synchronisation policy and will take corrective measures if it is not.

When the user commits the transaction, the relevant pages are flushed to disk by the database manager and the user releases the write lock. For added security, it will be necessary to also unmap the relevant pages of the object from the user's address space when the write lock is released, preventing from further access to the region by the user.

10.2.3 Evolution to a Distributed System

Finally, we look at the issues involved in using the interface described in this dissertation to build a distributed system. An important issue in this regard is the need to implement a distributed cache coherency protocol between object managers of the same type that reside on different machines. In addition, if the system intends to support process migration and since a process is made up of a number of objects, it must also support object migration

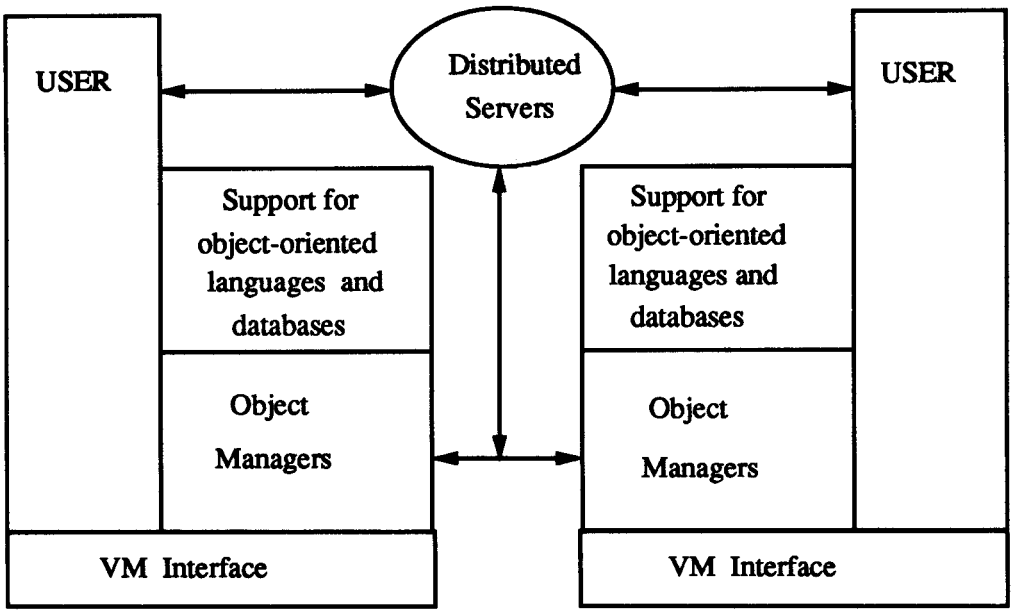

Figure 10.1: A Distributed System

as well as global naming and location transparency.

Figure 10.1 shows how the interface might evolve to provide a distributed environment. Object managers manage access to passive objects stored on various storage systems, including file, voice and video servers. They also implement cache coherency and synchronisation algorithms allowing passive objects to be shared among processes on various sites. Support will exist to integrate persistent support for object-oriented languages and databases. In addition users will also communicate with other distributed servers for example, printers, plotters, CPU servers, etc.

10.3 Final Word

In this dissertation an object-oriented virtual memory management system has been implemented and shown to be efficient, flexible and easily extensible. The modelling of program behaviour was also examined and a new framework for accessing various algorithms has been suggested. A powerful testbed was developed to gather data on the paging activity of various programs. Experimental results have been presented. New paging algorithms to handle objects that are accessed in a highly sequential manner have been designed and implemented.

References

[Abrossimov89a] V. Abrossimov, M. Rozier, and M. Gien. Virtual Memory Management in Chorus. In *Proceedings of the European Workshop*, Lecture Notes In Computer Science, Berlin, FRG, April 1989. Springer-Verlag.

[Abrossimov89b] V. Abrossimov, M. Rozier, and M. Shapiro. Generic Virtual Memory Management on Operating System Kernels. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Litchfield Park, Arizona, December 1989.

[Aho84] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. Awk – A Pattern Scanning and Processing Language. In *ULTRIX-32 Supplementary Documents: Volume II Programmer*, chapter 3, pages 5–10. DIGITAL, 1984.

[Alderson72] A. Alderson, W.C. Lynch, and B. Randell. Thrashing in a Multiprogrammed System. In *Operating Systems Techniques*. Academic Press, 1972.

[Arc90] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 ORD, UK. *ANSA Testbench Implementation Manual*, 3rd Edition, August 1990.

[Bach86] M. Bach. *The Design of the Unix Operating System*. Prentice-Hall International, 1986.

[Belady69] L.A. Belady and C.J. Kuener. Dynamic Space Sharing in Computer Systems. *CACM*, 12(5):282–288, May 1969.

[Bensoussan72] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics Virtual Memory: Concepts and Design. *CACM*, 15(5):308–318, May 1972.

[Bobraw72] J.D. Bobraw, J.D. Burchfiel, D.L. Murphy, and R. S. Tomlinson. TENEX, a Paged Time Sharing System for the PDP-10. *CACM*, 15(3):135–143, March 1972.

[Boyse74] J.W. Boyse. Execution Characteristics of Programs in a Page-on-Demand System. *CACM*, 17(4):192–196, April 1974.

[Carr81] R.W. Carr and J.L. Hennessy. WSClock – A Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 87–95, December 1981.

[Carr84] R.W. Carr. *Virtual Memory Management.* Computer Science: Systems Programming. UMI Press, 1984.

[Chamberlain73] D.D. Chamberlain, S.H. Fuller, and L.Y. Liu. An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory. *IBM Journal of Research and Development*, pages 404–412, 1973.

[Chu76] W.W. Chu and H. Opderbeck. Program Behaviour and the Page-Fault-Frequency Replacement Algorithm. *Computer*, pages 29–38, November 1976.

[Ciminiera87] L. Ciminiera and A. Valenzano. *Advanced Microprocessor Architectures*, chapter 8, pages 351–374. Electronic System Engineering Series. Addison-Wesley, 1987.

[Coffman68] E.G. Coffman and L.C. Varian. Further Experimental Data on the Behaviour of Programs in a Paging Environment. *CACM*, 11(7):471–474, July 1968.

[Cook88a] R.L. Cook and F.L. Rawson III. The Design of Operating System/2. *IBM Systems Journal*, 27(2):90–104, 1988.

[Cook88b] R.L. Cook, F.L. Rawson III, J.A. Tunkel, and R.L. Williams. Writing an Operating System/2 application. *IBM Systems Journal*, 27(2):134–157, 1988.

[Denning75a] P.J. Denning and G.S. Graham. Multiprogrammed Memory Management. *Proceedings of The IEEE*, 63(6):924–939, June 1975.

[Denning75b] P.J. Denning and K.C. Kahn. A Study of Program Locality and Lifetime Functions. In *Proceedings of the Fifth Symposium on Operating System Principles*, pages 207–216, November 1975.

[Denning76] P.J. Denning, K.C. Kahn, J. Leroudier, D. Potier, and R. Suri. Optimal Multiprogramming. *Acta Informatica*, 7(2):197–216, 1976.

[Denning80] P.J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–83, January 1980.

[Dig86] Digital Equipment Corporation. *The Vax Architecture Handbook*, 1986. Chapter 7.

[Edenfield90] R.W. Edenfield, M.G. Gallup, W.B. Ledbetter Jr, R.C. McGarity, E.E. Quintana, and R.A. Reininger. The 68040 Processor: Part 2, Memory Design and Chip Verification. *IEEE Micro*, pages 22–35, June 1990.

[Felten91] E.W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical report, Department of Computer Science, University of Washington, 1991.

[Ferrari83] D. Ferrari and Y. Yih. VSWS: The Variable-Interval Sampled Working Set Policy. *IEEE Transactions on Software Engineering*, SE-9(3):299–305, May 1983.

[Fitzgerald85] R. Fitzgerald and R.F. Rashid. The Integration of Virtual Memory and Interprocess Communication in Accent. In *Proceedings of the Tenth Symposium on Operating System Principles*, Orcas Island, Washington, December 1985.

[Franklin74] M.A. Franklin and R.K. Gupta. Computation of Fault Probabilities from Program Transition Diagram. *CACM*, 17(4):186–191, April 1974.

[Franklin78] M.A. Franklin, G.S. Graham, and R.K. Gupta. Anomalies with Variable Partition Algorithms. *CACM*, 21(3), March 1978.

[Gingell87] R.A. Gingell, J.P. Moran, and W.A. Shannon. Virtual Memory Architecture in SunOS. In *USENIX Association Proceedings*, pages 81–94, 1987.

[Gomaa79] H. Gomaa. A Simulation Based Model of a Virtual Storage System. In *12th Proceedings Annual Simulation Symposium*, 1979.

[Grit77] D.H. Grit. Global LRU Page Replacement in a Multiprogrammed Environment. In *Proceedings of Sigmetrics 1977 / CMG VIII Conf.*, December 1977.

[Hennessy90] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc, 1990.

[Hew87] Hewlett-Packard. *HP Precision Architecture and Instruction Manual*, 1987. Chapter 3.

[IBM82] IBM, White Plains, N.Y. *The Economic Value of Rapid Response Time*, 1982.

[Kahn76] K.C. Kahn. *Program Behaviour and Load Dependent System Performance.* PhD Thesis, Purdue University, 1976.

[Kane88] G. Kane. *mips RISC Architecture*, chapter 4. Prentice-Hall International, 1988.

[Kenah84] L.J. Kenah and S.F. Bate. *VAX/VMS and Data Structures.* DIGITAL PRESS, 1984.

[Kilburn62] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11:223–234, April 1962.

[Lau82] E.J. Lau. *Performance Improvement of Virtual Memory Systems*, chapter 4, pages 87–133. Computer Studies: System Programming. UMI Press, 1982.

[Leffler89] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3 BSD Unix Operating System*, pages 26–29. Addison-Wesley, 1989.

[Leslie91] I. Leslie and D. McAuley. Fairisle: An ATM Network for the Local Area. In *Proceedings of ACM SIGCOMM*, September 1991.

[Lo89] S. Lo. Data Availability in a Distributed System. Thesis Proposal, July 1989.

[Mahon86] M.J. Mahon, R.B. Lee, T.C. Miller, J.C. Huck, and W.R. Brgy. Hewlett-Packard Precision Architecture:- The Processor. *Hewlett-Packard Journal*, 37(8):4–22, August 1986.

[Masuda77] T. Masuda. Effect of Program Localities on Memory Management Strategies. In *Proceedings of Sixth Symposium on Operating System Principles*, pages 117–124, November 1977.

[McJones87] P.R. McJones and G.F. Swart. Evolving the Unix System Interface to Support Multithreaded Programs. Technical report, Systems Research Center, Digital Research Corporation, September 1987.

[McKusick87] M.K. McKusick and Karels M.J. A New Virtual Memory Implementation for Berkeley UNIX. Technical report, Department of Engineering and Computer Science, University of California, Berkeley, 1987.

[Milenkovic90] M. Milenkovic. Microprocessor Memory Management Units. *IEEE Micro*, pages 70–85, April 1990.

[Mizell88] A.M. Mizell. Understanding device drivers in Operating System/2. *IBM Systems Journal*, 27(2):170–184, 1988.

[Mot86] Motorola. *MC68851 Paged Memory Management Unit User's Manual*, 1986. Chapter 5.

[Needham77a] R.M. Needham and A.D. Birrell. The CAP Filing System. In *Proceedings of Sixth ACM Symposium on Operating Systems Principles*, pages 11–16, November 1977.

[Needham77b] R.M. Needham and R.D.H. Walker. The Cambridge CAP Computer and its Protection System. In *Proceedings of Sixth ACM Symposium on Operating Systems Principles*, pages 1–10, November 1977.

[Nicolaou90] C.A. Nicolaou. *A Distributed Architecture for Multimedia Communication Systems.* PhD Thesis, University of Cambridge, December 1990.

[Organick72] E.I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, 1972.

[Ousterhout88] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite Network Operating System. *Computer*, February 1988.

[Pashley89] R.D. Pashley and S.K. Lai. Flash memories: the best of two worlds. *IEEE Spectrum*, pages 30–33, December 1989.

[Pizzarello89] A. Pizzarello and W.J. Brophy. Memory Management for a Large Operating System. *Performance Evaluation Review*, 17(1):232–232, May 1989.

[Prieve76] B.G. Prieve and R.S. Fabry. VMIN – An Optimal Variable-Space Replacement Algorithm. *CACM*, 19(5):295–308, May 1976.

[Ratzer87] G. Ratzer. *Micros to SUPERMICROS: An Overview*, pages 226–232. Prentice-Hall International, 1987.

[Redell80] D.D. Redell, Y.K. Dalal, T.R. Horseley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell. Pilot: An Operating System for a Personal Computer. *CACM*, 23:81–92, February 1980.

[Ritchie84] D.M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, pages 1897–1910, October 1984.

[Rodriguez-Rosell73] J. Rodriguez-Rosell. Empirical Working Set Behaviour. *CACM*, 16(9), September 1973.

[Rodriguez-Rosell76] J. Rodriguez-Rosell. Empirical Data References in Data Base Systems. *Computer*, pages 9–13, November 1976.

[Sadeh75] E. Sadeh. An Analysis of The Performance of the Page Fault Frequency (PFF) Replacement Algorithm. In *Proceedings of the Fifth Symposium on Operating System Principles*, pages 6–13, November 1975.

[Saltzer74] J.H. Saltzer. Protection and Control of Information Sharing in Multics. *CACM*, 17(7):388–402, July 1974.

[Saltzer76] J.H. Saltzer. On the Modelling of Paging Algorithms. *CACM*, pages 307–308, May 1976.

[Sekino72] A. Sekino. *Performance Evaluation of Multiprogrammed Time-shared Computer Systems.* PhD Thesis, MIT, 1972.

[Shammas88] N.C. Shammas. Life After DOS. *BYTE IBM Special Edition*, pages 143–150, 1988.

[Shedler72] G.S. Shedler and C. Tung. Locality in Page Reference Strings. *SIAM Journal of Computing*, 1(3):218–241, September 1972.

[Simon79] R. Simon. *The Modeling of Virtual Memory Systems.* PhD Thesis, Computer Science Department, Purdue University, 1979.

[Smith76] A.J. Smith. A Modified Working Set Paging Algorithm. *IEEE Transactions on Computers*, C-25(9):907–914, September 1976.

[Smith78a] A.J. Smith. Bibliography on Paging and Related Topics, September 1978.

[Smith78b] A.J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[Spirn76] J. Spirn. Distance Strings Models for Program Behaviour. *IEEE Computer*, pages 14–20, November 1976.

[Spirn77] J.R. Spirn. *Program Behaviour: Models and Measurements.* Elsevier-North Holland, 1977.

[Tanenbaum90] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *CACM*, 33(12):46–63, December 1990.

[**Tevanian Jr.87a**] A. Tevanian Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* PhD Thesis, Carnegie-Mellon University, December 1987.

[**Tevanian Jr.87b**] A. Tevanian Jr., R.F. Rashid, M. Young, D. Golub, R. Baron, D. Black, W.D. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceeding of the Second Symposium on Architectural Support for Programming and Operating Systems*, pages 31–39, October 1987.

[**Tevanian Jr.87c**] A. Tevanian Jr., R.F. Rashid, M.W. Young, D.B. Golub, M.R. Thompson, D. Bolosky, and R. Sanzi. A Unix Interface for Shared Memory and Memory Mapped Files Under Mach. Technical report, Department of Computer Science, Pittsburgh, PA 15213, 1987.

[**Thacker82**] C.P. Thacker, E.M. McCreight, E.M. Lampson, R.F. Sproull, and D.R. Boggs. Alto: A personal computer. In *Computer Structures: Principles and Examples*, pages 549–572. McGraw-Hill, 1982.

[**Thacker87**] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite Jr. Firefly: A Multiprocessor Workstation. Technical report, Digital System Research Center, Palo Alto, California, December 1987.

[**Thadani81**] A.J. Thadani. Interactive user productivity. *IBM Systems Journal*, 20(4):407–423, 1981.

[**Walmer89**] L.R. Walmer and M.R. Thompson. A Programmer's Guide to Mach System Calls. Technical report, Department of Computer Science, Carnegie-Mellon University, December 1989.

[**Wilkes79**] M.V. Wilkes and R.M. Needham. *The Cambridge CAP Computer and Its Operating System.* Operating and Programming Systems Series. Elsevier North Holland, 1979.

[**Wilkes89**] J. Wilkes. DataMesh – scope and objectives: a commentary. Technical report, Hewlett-Packard, July 1989.

[**Wilson91**] T.D. Wilson. A Reliable Stream Protocol, April 1991. A talk at the Wanda Fun Day.

[**Wu**] Z. Wu. Personal Demo: Comparing 'Hello World' programs for C and C++.