Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology
Kirtipur, Kathmandu

DETERMINING OPTIMAL PAGE SIZE FOR MULTIPROGRAMMING OPERATING SYSTEM

Dissertation

Submitted to
Central Department of Computer Science and Information Technology
(Tribhuvan University)
For the fulfillment of the
Master of Science in Computer Science and Information Technology

Thesis Supervisor
Prof. Dr. Onkar P. Sharma
Marist College, Poughkeepsie
New York, USA

Submitted by:
Tiwari, Paras Babu

May, 2007

DETERMINING OPTIMAL PAGE SIZE FOR MULTIPROGRAMMING OPERATING SYSTEM

By
Tiwari, Paras Babu

DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF MASTER OF

COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

Supervisor : Prof. Dr. Onkar P. Sharma

Marist College, Poughkeepsie

New York, USA

Previous Degree: Bachelor in Computer Science

Tribhuvan University, 2004

Central Department of Computer Science and Information

Technology (CDCSIT)

Tribhuvan University

Kirtipur, Kathmandu

Nepal

May, 2007

LETTER OF RECOMENDATION

Mr. Paras Babu Tiwari has carried out this dissertation work entitled “Determining Optimal Page Size for Multiprogramming Operating System” under my supervision and guidance. To the best of my knowledge this is an original work in computer science. I, therefore recommend for further evaluation.

Prof. Dr. Onkar P. Sharma

Marist College, Poughkeepsie

New York, USA

(Supervisor)

We certify that we read this dissertation and in our opinion it is satisfactory in the scope and quality as a dissertation in the partial fulfillment for the requirement of Master of Science in Computer Science and Information Technology.

Evaluation Committee

---	---
Mr. Min Bahadur Khati	Prof. Dr. Onkar P. Sharama
Act. Head of Department	Thesis Supervisor
Central Department of Computer	Marist College, Poughkeepsie
And Information Technology	New York, USA
Tribhuvan University	

---	---
(External Examiner)	(Internal Examiner)

Date: _____

ABSTRACT

Determining optimal page size is one of the challenging tasks for designer of the operating systems. If page size is large, it creates larger fragmentation. On the other hand, if it is small, page table requires a huge memory space. The page size also affects the miss ratio. Present study analyzes these issues. For analysis purpose, a Multiprogramming operating system (MOS) is undertaken. Parameters are selected which affect the page size, and by varying them, performance analysis on MOS is experimentally conducted. The result achieved from the experiment is that memory fragmentation increases and storage requirement for the page table decreases in proportion to the page size. But the page fault does not decrease in proportion to page size. After the experiment, the optimal page size for the MOS system is found to be 40 bytes. This thesis tries to further study the existing theory on page size.

ACKNOWLEDGEMENTS

Many people have contributed to complete this thesis work.

I would like to express heartfelt regards to my supervisor Prof. Dr. Onkar P Sharma, Professor, and Graduate director, Marist College, USA for his continuous guideline, support, and inspiration throughout the thesis work.

Prof. Dr. Devi Dutta Poudyal, former head of the Central Department of Computer Science and Information Technology, Kirtipur deserves my special thanks as he has played vital role in bringing my supervisor and me together.

I would like to express my deep sense of appreciation to Prof. Shashidhar Ram Joshi, Professor of Institute of Engineering, Tribhuvan University, Pulchowk, Lalitpur who has supported uniquely in every way while preparing the thesis.

I convey my special affectionate credit to Mr. Min Bahadur Khati, Acting head of Department of Central Department of Computer Science and Information Technology, Kirtipur and all faculties of the department for their suggestion and comments for the improvement of the thesis.

I am highly grateful to Mr. Bhim Kumar Shreshta, for his valueable cooperation by going through this thesis and correcting the grammatical mistakes in it.

ABBREVIATIONS

TLB	Translation Look Aside Buffer
GD	Get Data
PD	Put Data
LR	Load a virtual memory locations contents into R.
R	Four Byte General Purpose Register
CR	Comparer R to contents of virtual memory location
H	Halt user program
SR	Store contents of R into virtual memory location
BT	Branch on True
SI	Supervisor interrupts
PI	Program Interrupt
TI	Timer Interrupt
I/O	Input Output
IOI	I/O Interrupt
MOS	Multiprogramming Operating System

eb	Empty Buffer
ifb	Input Full Buffer
ofb	Output Full Buffer
ebq	Empty Buffer Queue
ifbq	Input Full Buffer Queue
ofbq	Output Full Buffer Queue
RQ	Ready Queue
TQ	Terminate Queue
LQ	Load Queue
TSC	Time Slice Counter
TTC	Total Time Counter
TS	Time Slice
TTL	Total Time Limit
TLC	Total Line Count
TLL	Total Line Limit
IS	Input Spooling
OS	Output Spooling
RA	Real Address
VA	Virtual Address
IR	Instruction Register

Table of Contents

LIST OF FIGURES

LIST OF TABLES

1 CHAPTER 1: INTRODUCTION

1.1 Memory Management

Ideally, every programmer wants infinitely large, fast, and inexpensive memory. Unfortunately, technology does not provide such memories. Consequently, most computers have a memory hierarchy, with a small amount of very fast, expensive volatile cache memory, tens of megabytes of medium speed, medium-price volatile main memory(RAM) and tens or hundreds of gigabytes of slow, cheap, nonvolatile storage. It is the job of the operating system to coordinate how these memories are used.

The part of the operating system that manages the memory hierarchy is called memory manager. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.

In general, memory management should able to handle following things.

Relocation

Programs in memory must be able to reside in different parts of the memory at different times. This is because when the program is swapped back

into memory after being swapped out for a while it can not always be placed in the same location. Memory management in the operating system should therefore be able to relocate programs in memory and handle memory references in the code of the program so that they always point to the right location in memory.

Protection

Processes should not be able to reference the memory for another process without permission.

Sharing

Even though the memory for different processes is protected from each other, different processes should be able to share information and therefore access the same part of memory.

Logical organization

Programs are often organized in modules. Some of these modules could be shared between different programs, some are "read" only and some contain data that can be modified. The memory management is responsible for handling this logical organization that is different from the physical linear address space. One way to arrange this organization is segmentation.

Physical organization

Memory is divided into main memory and secondary memory. Memory management in the operating system handles moving information between these two levels of memory.

1.2 Memory Management Unit (MMU)

Memory Management Unit is a class of computer hardware components responsible for handling memory accesses requested by the CPU. Among the functions of such devices are the translations of virtual addresses to physical, memory protection, cache control etc.

Modern MMUs typically divide the virtual address space (the range of addresses used by the processor) into pages, whose size is 2^n, usually a few kilobytes. The bottom m bits of the address (the offset within a page) are determined by the page size. The upper address bits, n-m, are the (virtual) page number. The MMU normally translates virtual page numbers to physical page numbers via an associative cache called a Translation Lookaside Buffer (TLB). When the TLB lacks a translation, a slower mechanism involving hardware-specific data structures or software assistance is used. The data items found in such data structures are typically called page table entries (PTEs), and the data structure itself is typically called a page table. The physical page number is combined with the page offset to give the complete physical address.

A PTE or TLB entry may also include information about whether the page has been written to (the dirty bit), when it was last used, what kind of processes (user mode, supervisor mode) may read and write it, and whether it should be cached.

It is possible that a TLB entry or PTE prohibits access to a virtual page, perhaps because no physical memory (RAM) has been allocated to that virtual page. In this case the MMU will signal a page fault to the CPU. The operating

system will then handle the situation appropriately, perhaps by trying to find a spare page of RAM and set up a new PTE to map it to the requested virtual address. If no RAM is free it may be necessary to choose an existing page, using some replacement algorithm, save it to disk and then load the requested/referenced page in that space.

In some cases a "page fault" may indicate a software bug. A key benefit of an MMU is memory protection. An operating system can use it to protect against errant programs, by disallowing access to memory that a particular program should not have access to. Typically, an operating system assigns each program its own virtual address space.

1.3 Fragmentation

Fragmentation causes wastage in computer storage. There are different kinds of fragmentation. Two important fragmentations are:

1) Internal Fragmentation
2) External Fragmentation

1.3.1 Internal Fragmentation

Internal fragmentation occurs when storage is allocated without ever intending to use it. This space is wasted. It is often accepted in return for increased efficiency or simplicity in allocation. The term "internal" refers to the fact that the unusable storage is inside the allocated region but is not being used.

For example, in many file systems, files always start at the beginning of a sector, because this simplifies organization and makes it easier to grow files. Any

space left over between the last byte of the file and the first byte of the next sector is internal fragmentation. Similarly, a program which allocates a single byte of data is often allocated many additional bytes for metadata and alignment. Likewise the last page of a process almost always has some un-utilized space left.This extra space is also internal fragmentation.

1.3.2 External Fragmentation

External fragmentation is the phenomenon in which free storage becomes divided into many small pieces over time. It is a weakness of certain storage allocation algorithms, occurring when an application allocates and deallocates regions of storage of varying sizes, and the allocation algorithm responds by leaving the allocated and deallocated regions interspersed. The result is that, although free storage is available, it is effectively unusable because it is divided into pieces that are too small to satisfy the demands of the application. The term "external" refers to the fact that the unusable storage is outside the allocated regions.

For example, in dynamic memory allocation, a block of 2000 bytes might be requested, but the largest contiguous block of free space, has only 800 bytes. Even if there are ten blocks of 800 bytes of free space, separated by allocated regions, one still cannot allocate the requested block of 2000 bytes, and the allocation request will fail.

1.4 Virtual Memory

Virtual memory is an addressing scheme that allows non-contiguous memory to be addressed as if it is contiguous. The technique used by all current implementations provides two major capabilities to the system:

Memory can be addressed that does not currently reside in main memory and the hardware and operating system will load the required memory from auxiliary storage automatically, without any knowledge of the program addressing the memory, thus allowing a program to reference more (RAM) memory than actually exists in the computer.

In multi tasking systems, total memory isolation, referred as a discrete address space, can be provided to every task except the lowest level operating system. This greatly increases reliability by isolating program problems within a specific task and allowing unrelated tasks to continue to process.

Virtual memory has been a feature of most of the today's operating system. In x86 32 bit processor the maximum memory that can be addressed is 2^32 = 4GB. This 4GB set of addressable addresses is called the address space and the addresses are called virtual addresses.

In Linux system, this 4GB address space is divided into the 3 GB user space and 1GB kernel space. In the normal, default Windows OS configuration, 2

GB of this address space are allocated to the process's private use and the other 2 GB are allocated to shared and operating system use.

1.5 Demand Paging

As there is much less physical memory than virtual memory, the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to only load virtual pages that are currently being used by the executing program. This process is known as the demand paging. When a process attempts to access a virtual address that is not currently in memory the processor cannot find a page table entry for the virtual page referenced. At this point, the processor notifies the operating system that a page fault has occurred.

Linux uses demand paging to load executable images into a processes virtual memory. Whenever a command is executed, the file containing it is opened and its contents are mapped into the processes virtual memory. This is done by modifying the data structures describing this processes memory map and is known as memory mapping. However, only the first part of the image is actually brought into physical memory. The rest of the image is left on disk. As the image executes, it generates page faults and Linux uses the processes memory map in order to determine the parts of the image to bring into memory for execution.

1.6 Page Table

1.6.1 Definition

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are those unique to the accessing process. Physical addresses are those unique to the CPU.

1.6.2 Role of the page table

During a virtual to physical address translation, the virtual address is broken up into a virtual page number and an offset. With hardware support for virtual memory, the address is looked up within the Translation Lookaside Buffer(TLB). The TLB is specifically designed to perform this lookup in parallel, so this process is extremely fast. If there is a match for a page within the TLB (a TLB hit), the physical frame number is retrieved, the offset replaced, and the memory access can continue. However, if there is no match (called a TLB miss), the second part-of-call is the page table.

When the hardware is unable to find a physical frame for a virtual page, it will generate a processor interrupt called a page fault. Hardware architectures offer the chance for an interrupt handler to be installed by the operating system to deal with such page faults. The handler can look up the address mapping in

the page table, and can see whether a mapping exists in the page table. If one exists, it is written back to the TLB, and the faulting instruction is restarted, with the consequence that the hardware will look in the TLB again, find the mapping, and the translation will succeed.

However, the page table lookup may not be successful for two reasons:

- There is no translation available for that address - the memory access to that virtual address is thus bad or invalid, or
- The page is not resident in physical memory.

In the first case, the memory access is invalid, and the operating system must take some action to deal with the problem. On modern operating systems, it will send a segmentation fault to the offending program. In the second case, the page is normally stored elsewhere, such as on a disk. To handle this case, the page needs to be taken from disk and put into physical memory. When physical memory is not full, this is quite simple, one simply needs to write the page into physical memory, modify the entry in the page table to say that it is present in physical memory, write the mapping into the TLB and restart the instruction.

However, when physical memory is full, and there are no free frames available, pages in physical memory may need to be swapped with the page that needs to be written to physical memory. The page table needs to be updated to mark that the pages that were previously in physical memory are no longer so, and to mark that the page that was on disk is no longer so also. Then write the mapping into the TLB and restart the instruction. This process however is

extremely slow in comparison to memory access via the TLB or even the page table, which lies in physical memory. Which page to swap is the subject of page replacement algorithms.

A very simple example of how the mapping works is shown in figure 1-1. In this example, computer can generate 16-bit addresses, from 0 up to 64KB. These are the virtual addresses. This computer, however has only 32KB of physical memory, so although 64KB programs can be written, they cannot be loaded into memory in their entirety and run. A complete copy of a program's core image, up to 64 KB, must be present on the disk, so that pieces can be brought while necessary.

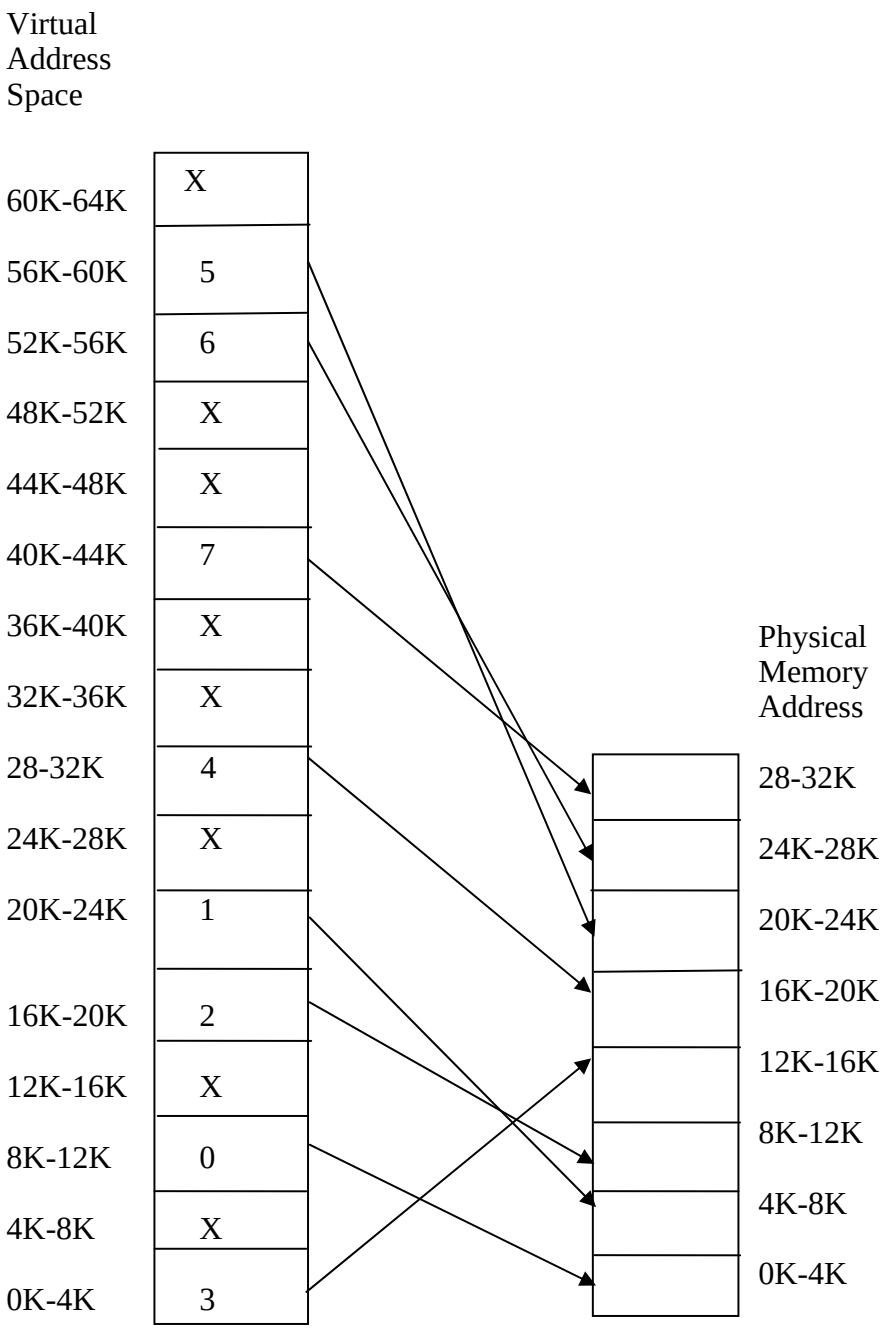

Figure 1-1 Mapping between virtual address and physical address

1.6.3 Page Table Data

The simplest page table systems often maintain a frame table and a page table.

The frame table, in the most basic system, holds information about which frames are mapped. In more advanced systems, the frame table can also hold information to which address space a page belongs, or statistics information, or other background information.

The page table has different fields. The most important field is the page frame number. There is also auxiliary information about the page such as a present bit, protection bits, a dirty or modified bit and Referenced bit.

The page frame number provides the mapping between a virtual address of a page and the address of a physical frame. The present/absent bit can indicate about pages that are currently present in physical memory. The protection bits tell what kind of access is permitted.

The dirty bit allows us a performance optimization. It keeps track of page

usage. When a page is written to, the hardware automatically sets the modified bit. During the swapping, this page is used to determine whether a page is modified or not. If the page was not modified, we don't need to write this page back to disk since the page hasn't changed. However, if the page was modified, we would need to write the page back so if we reload the page, we get the correct information back.

The Referenced bit also keeps track of page usage**.** This bit is set whenever a page is referenced, for either reading or writing. Its value is to help the operating system, choose a page to evict when a page faults occur.

1.6.4 Multilevel page table

Modern computers use virtual addresses of at least 32 bits. With, say, a 4-KB page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than 1 million pages. With 1 million pages in the virtual address space, the page table must have 1 million entries. This takes large amount of memory.

To get around this problem, many computers use a multilevel page table. In 32 bit virtual address system that bit is divided into 3 fields, 10-bit for PT1 field, 10-bit for PT2 field, and a 12-bit offset field. Since offsets are 12 bits, pages are 4KB, and there are a total of 2^{20} of them. The secret of the multilevel page table is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around.

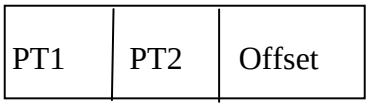

Figure 1-2 A 32-bit address with two page table fields

Linux uses three levels of page tables. Each Page Table accessed contains the page frame number of the next level of Page Table. To translate a virtual address into a physical one, the processor must take the contents of each level field, convert it into an offset into the physical page containing the Page Table and read the page frame number of the next level of Page Table. This is repeated three times until the page frame number of the physical page containing the virtual address is found. Now the final field in the virtual address, the byte offset, is used to find the data inside the page.

1.6.5 Page Size

An important parameter in the design of the paged computing system is the page size i.e. number of bytes of information transferred from one level of hierarchy to another level of hierarchy. Determining best page size requires balance of several factors. As a result, there is no overall optimum.

One of the factor that is influenced by the page size is the space needed to store the page table. In mono-programming system, only one process runs at a time. In such system, virtual address corresponds to the real physical address, so no page table is required to map virtual address to physical one. But in multiprogramming system, several processes may run simultaneously. In such system, virtual address and physical address are not usually the same. So each process needs a page table, to map the virtual address into physical address. In addition, if we have small page size, maintaining page table for each running process takes large space. This reasoning argues for a large page size.

A randomly chosen text, data, or stack segment will not fill an integral number of pages. On the average, half of the final page will be empty. The extra space in that page is wasted. This wastage of space is called internal fragmentation. This reasoning argues for a small page size

Transfers to and from the disk are generally a page at a time. When a transfer from the disk occurs, most of the time is consumed for seek and rotational delay. The seek and rotational delay are independent of the page size. Therefore transferring a small page takes as much time as transferring a large page. This reason strongly favors the need of large page size.

1.7 Literature Survey

Prof. Andrew S. Tanenbaum in his book Modern Operating Systems [1] discusses two major factors for determining the page size. This includes the fragmentation of memory and access and transfer time from secondary storage device. If large page size is used huge amount of the memory is wasted in the fragmentation. Transfer time to and from the disk is high in the use of small page size.

Besides these two parameters, other factors are also influenced by the page size. One of the factor that is influenced by page size is the miss ratio. In Chu and Opderbeck's paper [2], they found miss ratio heavily dependent on the page size. Their miss ratio curves asymptotically approach a value, which is simply the number of initial loading misses divided by length of observed page reference string.

But Bennet[3] examined a page reference trace from the IBM Advanced Administrative System, a large internal IBM data management system. Bennet found no concise relationship between page size and miss ratio. Rather he noticed, the size of main memory affects miss ratio more than page size did. The cache multiprogramming trace of Kalpan and Winder[4] and the program address traces of Lewis and Shelder[5] and of Anacker and Wang[6] gave similar results.

The reason behind the difference between these two experimental results is discussed by Fagin, R. A, [7] in his paper “The independence of miss ratio on page size” published in the journal of ACM. According to him, Chu and Opderbeck's made no distinction between initial loading miss and “long-term” or “transient- free” misses. On the other hand, others distinguished between these two misses. In case of fixed number of pages loaded in memory, the “transient-free” or “long- term” miss means; the miss ratio is measured starting at a time after the main memory has filled; in the working set case, “ transient-free” means that the miss ratio is measured starting at a time grater than T, where T is the window size.

In a computer, system minimization of virtual to physical address translation time has a great importance. To minimize this time most of the today's microprocessors store their recently accessed translations in a buffer on the chip. This buffer is called the translation look aside buffer (TLB). If a translation is not found in the TLB, the processor takes a TLB miss.

The number of entries in the TLB multiplied by the page size is defined as TLB reach. If the page size is small, then the page table is large. As the page table is large, TLB will have only few numbers of entries out of the large page table. On the other hand, if the page size is large, then the page table is small. As the page table is small, TLB will have most of the entries of the page table. Hence, the TLB reach is directly proportional to the page size. Large page size

has better TLB reach than smaller page size.

The TLB reach is critical to the performance of an application. If the TLB reach is not enough to cover the working set of the process, the process may spend significant portion of it's time satisfying TLB misses. Hence, if a system has to perform well it should have good TLB reach. If we have large page size, then TLB reach is high and hence an application, which has large working set, can run smoothly. However, having large page size degrades the performance for the application, which has small working set because large page size creates fragmentation for such process.

The solution of this problem has been discussed by Narayanan Ganapathy and C. Schimmel in their paper [8] “General purpose Operating System for Multiple page Sizes”. According to them, the solution of the problem is to have operating system that can support multiple page sizes. The implementation of the multiple page size is a challenging task and lots of work is going on this area.

Thus choosing optimal page size requires taking into consideration of several conflicting goals, and detailed study about how the system behaves on various page sizes.

1.8 Objective and Outline

The main objective of this thesis is to study the behavior of a Multiprogramming Operating System (MOS) on various page sizes and find out the optimal page size for the MOS system.

This thesis consists of eight chapters. Chapter 1 deals with the introduction to the basic concept of operating system. Chapter 2 discusses about specification of the MOS project. Similarly, chapter 3 provides the detailed design of the MOS system. Experimental data used in the study has been presented in chapter 4. Chapter 5 contains the result found in the study and the implication of the result. Chapter 6 gives the conclusion and Chapter 7 states the limitation of the study and future work on the page size.

2 MOS SYSTEM SPECIFICATION

2.1 Project Specification

MOS is a multiprogramming operating system for a hypothetical computer. According to the specification given by Prof. Dr. Onkar P. Sharma in his paper[9] “Enhancing Operating System Course Using a Comprehensive Project: Decades of Experience Outlined” the MOS system consists of CPU, card reader, 3 channels, printer, magnetic drum, main memory and supervisor memory. The supervisor memory has unlimited storage. Secondary storage device has 100 40-byte tracks. Similarly, the main memory has 300 words and each word is 4 byte long. The channel1 performs input from card reader to supervisor memory. The channel2 performs output from supervisor memory to printer, channel3 transfers information between the drum and memory, as well as supervisor memory and the drum.Channel3 is faster than other two channels, and channel1 and channel2 operate at the same rate.

The machine has a simplified user view. The user view of the machine consists of CPU, card reader, virtual memory and printer. The virtual memory is 100 four-byte words organized into 10 word pages. There are three CPU registers: a general purpose four byte register R, a two byte instruction counter

(IC) and a one byte toggle register C. The register C holds result of compare R and memory location (CR) mnemonic. The CPU can accept following seven instructions.

LR: Load a virtual memory locations contents into R,

SR: Store contents of R into virtual memory location,

CR: Comparer R to contents of virtual memory location,

BT: Branch on True,

GD: Get Data,

PD: Put Data,

H: Halt user program,

The machine switches between slave and master mode. MOS runs in the master mode and User program runs in slave mode. It is assumed that the MOS takes no time in the master mode and each instruction takes one time unit to simulate in the user mode. Each instruction occupies one word of memory. The first two bytes of the instruction contains the mnemonic such as GD/ PD and last two bytes contains the virtual address. The virtual address can range from 00 to 99. The first instruction is loaded in the memory location 00.

MOS is interrupt driven. There are four types of interrupt. These are Supervisor interrupt (SI), Program Interrupt (PI), Timer Interrupt (TI) and input output(I/O) Interrupt (IOI) .SI interrupt occurs when GD, PD or H instructions are

encountered in user program and behaves like system call instructions . PI interrupt occurs when an error condition such as memory protection exception or page fault occurs. Timer interrupt (TI) is generated when a user program exceeds its time quantum or has exceeded its total allowed run time. IOI interrupt is generated by a channel when its operation has completed. The CPU is not interruptible in master mode, however appropriate interrupt register is set when the interrupt causing event occur.

MOS employs Input spooling to transfer job from card reader to drum before loading and executing the program. Channel 1 and channel 3 participate in this transfer. At the time of execution, real I/O occurs from and to the drum to shorten the I/O delay. Loading of program cards as well as I/O operations are handled by channel 3. The output of the job is written into the drum. When job terminates, channel2 and channel3 coordinate to send the output from drum to printer. The task of transferring information from memory to drum is called output spooling and the task of transferring information from drum to memory is called input spooling. Buffering is employed to provide synchronization.

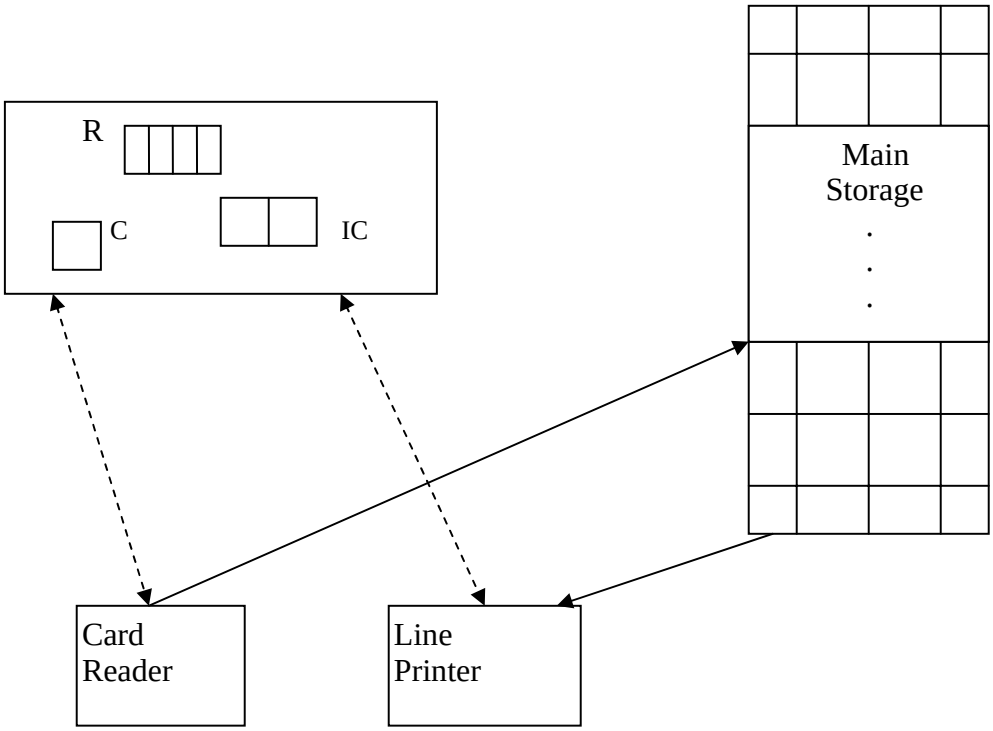

Figure 2-1 Virtual User Machine

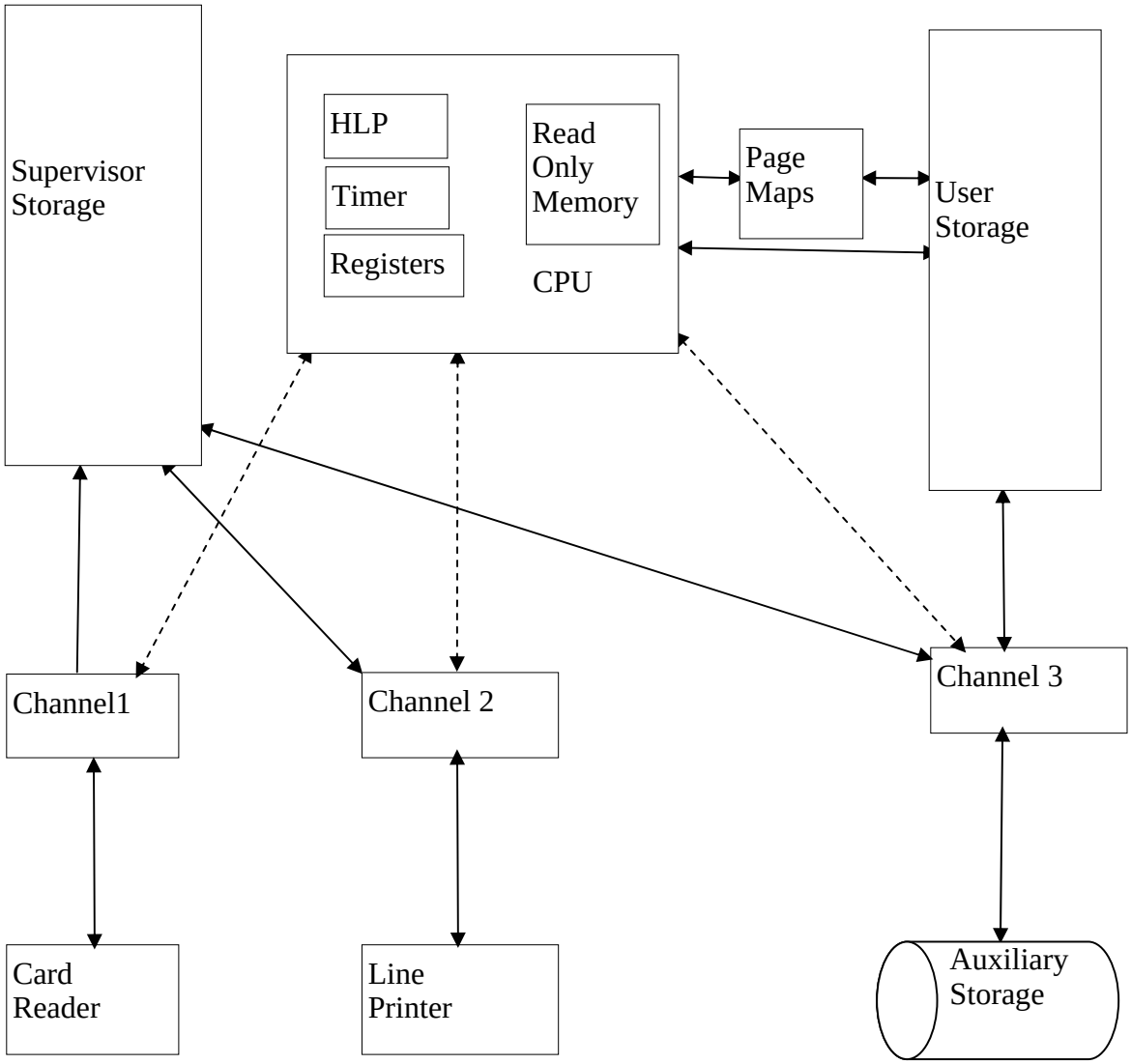

Figure 2-2 The Real Machine

3 DESIGN OF THE MOS SYSTEM

3.1) Flow of the System

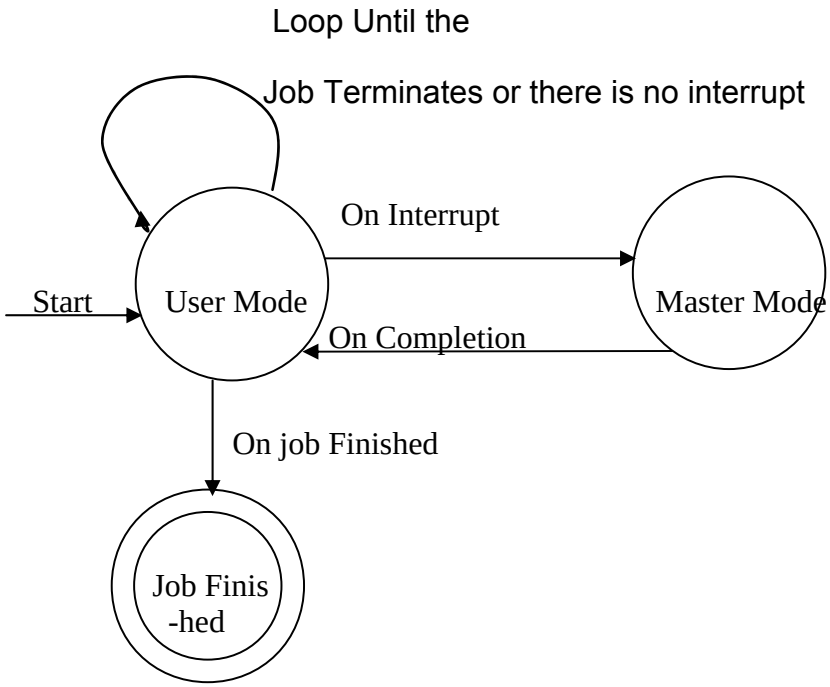

Figure 3-1 Overall State Diagram of MOS Machine

Description

- The MOS machine loops in user mode until there is no interrupt.
- When interrupt occurs, the machine switches into master mode. In master mode, it serves the interrupt and after completion of the operation, machine switches into the user mode.
- Repeat step 1 and 2 until all jobs are finished.

3.2) Definition of the Constants

3.2.1) Error Message Coding

S.No.	EM	Error
1	0	No Error
2	1	Out of Data
3	2	Line Limit Exceeded
4	3	Time Limit Exceeded
5	4	Operation Code Error
6	5	Operand Code Error
7	6	Invalid Page Fault

Table 3-1 Error Message Coding

3.2.2) Interrupt Values

S.No.	Flag	Value	Description
1	SI	1	On GD
2	SI	2	On PD
3	SI	3	On H
4	TI	1	On Time Slice Out
5	TI	2	On Time Limit Exceeded
6	PI	1	Operation Error
7	PI	2	Operand Code Error
8	PI	3	Invalid Page Fault
9	IOI	1	Channel 1 done

S.No.	Flag	Value	Description
10	IOI	2	Channel 2 done
11	IOI	4	Channel 3 done

Table 3-2 Interrupt Values

3.3) Data Structure used in the MOS system

3.3.1) PCB structure

Process Control Block (PCB) is an array data structure, maintained for each process. It holds the state and all information related to a process which MOS needs to monitor its execution. Specifically, it should have at least following fields

PCB Structure {

Job id of the process

Total Time Limit of the job

Total Line Limit of the job

Variable to hold the content of Page Table Register (PTR).

Number of program card

Number of data card

Location of the Program card in disk

Location of the Data card in disk

Total Time Counter of the job

Total Line Counter of the job

Location of the output track in the disk

Flag for error message

Array to hold the content of Register

Array to track whether page is modified or not

Array to track whether page is Referenced or not

}

3.3.2) Queues

The MOS system uses two different types of Queues.

1. Queues that hold the job during processing ,
2. Queues used for input and output spooling.

First types of queues are Load Queue (LQ), Ready Queue (RQ), Swap Queue (SQ), Terminate Queue (TQ), I/O Queue (IOQ) and PCB Queue (PCBQ). They all have same structure. The major element of these queues is the PCB structure.

The second types of queues are Empty Buffer Queue (ebq), Input full Buffer Queue (ifbq) and Output full Buffer queue (ofbq). All these queues have buffer as the major element, which is a character array of 40 bytes. These buffers are initially fixed to some number, all of these are declared as empty buffer and during the operation of the MOS, these buffers are moved into different queues. When the buffer is in input full queue, we call it as input full buffer. Similarly, when the buffer is in output full queue we call it as output full buffer.

3.4) Algorithm

Function Name: kernel ()

Outline:

This function simulates the master mode operation of the MOS

Algorithm

- ✓ Check the interrupt. There are four types of interrupt: Supervisor Interrupts (SI), Program Interrupts (PI), Timer interrupt (TI) and I/O interrupt (IOI).For the various values and combination of these interrupts, MOS should take different action as indicated in the following tables.

TI	*SI*	*Action*
1	0	Time Slice exceeded, so move PCB to the end of Ready Queue.
0 or 1	1	Move PCB, RQ---->IOQ(Read)
0 or 1	2	Move PCB, RQ---->IOQ(Write)
0 or 1	3	Move PCB, RQ---->TQ(No Error)
2	0	Move PCB,RQ---->TQ(Time Limit Exceeded)
2	1	Move PCB, RQ---->TQ(Time Limit Exceeded)
2	2	Move PCB, RQ---->IOQ(Write) then TQ(Time Limit Exceeded)
2	3	Move PCB, RQ---->TQ(No Error)

Table 3-3 Action of MOS

TI	***PI***	***Action***
0 or 1	1	Move PCB, RQ---->TQ(Operation Code Error)
0 or 1	2	Move PCB, RQ---->TQ(Operand Code Error)

0 or 1	3	Page Fault:
		If Valid
		If Frame Available
		Allocate
		Update Page Table
		Adjust IC if necessary
		if the page fault is due to not having program card
		Move PCB, RQ---->LQ
		else
		Move PCB, RQ---->SQ
		else
		Move PCB, RQ---->TQ(Invalid Page Fault)

TI	*PI*	*Action*
2	1	Move PCB, RQ---->TQ(Time Limit Exceeded,Operation Code Error)
2	2	Move PCB, RQ---->TQ(Time Limit Exceeded,Operand Code Error)
2	3	Move PCB, RQ---->TQ(No Error)

Table 3-4 Action of MOS

IOI	*Action*
0	No Action
1	Interrupt Service Routine1(IR1)
2	Interrupt Service Routine1(IR2)
3	IR2 ,IR1
4	Interrupt Service Routine3(IR3)
5	IR1, IR3
6	IR3, IR2
7	IR2, IR1 IR3

Note: IR_i is interrupt service routine for channel i for i=1,2,3.

Table 3-5 Action of MOS

- ✓ Examine input full buffer(ifb)
    - o if ifb is $AMJ:
        - Create and Initialize PCB.
        - Allocate frame for page table.
        - Initialize page table and Page Table Register(PTR)
        - Set F<---P(Program cards to follow)
        - Change status from ifb to Empty Buffer(eb)
        - Return buffer to ebq.
- ✓ if ifb is $DTA:
    - o Set F<---P(Data cards to follow)
    - o Change Status from ifb to empty buffer(eb)
    - o Return buffer to empty buffer queue(ebq).
- ✓ if ifb is $END:
    - o Place PCB on Ready queue and change status from ifb to eb,
    - o Return buffer to ebq.
- ✓ Otherwise
    - o Place ifb to ifbq, save F information.
- ✓ Assigning new task in priority queue

- if a PCB on TQ(output spool first)
  - if last line count in PCB
    - Get two empty buffer and fill it with blanks, change status from eb to Output Full Buffer(ofb) and place the buffer into Output Full Buffer Queue(ofbq).
    - Prepare two lines of messages, move them into ebq(if available),change status from eb to ofb,and place these buffer into ofbq.
    - Release PCB, all remaining drum tracks and all memory blocks.
  - Else if ebq not empty and channel3 is not busy
    - Get Next buffer from ebq
    - Find track number of next output line
    - Task <---- Output Spooling (OS)
    - Start Channel3
- if ifbq not empty and channel3 is not busy
  - Get Next buffer from ifbq
  - Get a drum track.
  - Task <---- Input Spooling(IS)
  - Start Channel3

- if a PCB on LQ(Load Next) and Channel3 is not busy
    - Find the track number of next program card
    - Allocate a frame
    - Update page table
    - Task <---- Load (LD)
    - Start Channel3
- if a PCB on IOQ(Now I/O) and Channel3 is not busy
    - If Read(GD)
        - if no more data card
            - Move PCB,IOQ---->TQ(Terminate[3])
        - Else
            - Find track number of next data card
            - Get memory Real Address(RA)
            - Task <---- GD
            - Start Channel3
    - else if Write(PD)
        - if Total Line Counter(TLC)> Total Line Limit(TLL)
            - Move PCB,IOQ---->TQ(Terminate[2])
        - Else

        - Get a drum track, if available
        - Update PCB
        - Find memory RA
        - Task <---- PD
        - Start Channel3
- if a PCB on SQ and Channel3 is not busy
    - if memory frame now available
        - Allocate
        - Update page Table
        - Adjust IC, if necessary
        - Move PCB, SQ---->RQ with Time Slice Counter(TSC) <--- 0
    - Else
        - Run page Replacement Algorithm
        - Find a victim frame
        - Allocate and Reallocate this frame by updating both page tables
        - if victim frame not written into,
            - locate drum track for faulted page
            - Task <---- Swap Queue Read(SQR)

        - Start Channel3
    - Else
        - Task <----- Swap Queue Write (SQW)
        - Start Channel3

- ✓ End of Assigning Task
- ✓ Set mode flag equals to zero
- ✓ Switch into the Slave mode
- ✓ End of Function Kernel

Function Name: Create_Pcb(input full buffer)

Outline:

This function creates the PCB

Algorithm

- ✓ Allocate space for PCB node
- ✓ Extract job id, Total Time Limit, Total Line Limit and place it into the PCB.
- ✓ Insert newly created PCB into PCB queue.

Function Name:

Start_Channeli();

Outline

This function starts the Channeli(i=1,2,3)

Algorithm:

- ✓ Adjust IOI(Subtract 1,2,4).
- ✓ Reset Channel timer to zero.
- ✓ Set Channel flag to busy.

Function Name

IR1();

Outline

This function clears the channel1 and start it again.

Algorithm:

- ✓ Read next card in given eb, change status to ifb,place on ifbq.
- ✓ if not e-o-f and ebq not empty
    - o Get next eb.
    - o Start Channel1()

Function Name:

IR2();

Outline

This function clears the channel2 and starts it again.

Algorithm:

- ✓ Print given ofb, change status from ofb to eb.
- ✓ Return buffer to ebq
- ✓ if ofbq not empty
    - o Get next ofb.
    - o Start Channel2().

Function Name:

IR3();

Outline

This function simulates the behavior of Channel3.

Algorithm:

- ✓ if Task is IS
    - o Write given ifb on given track
    - o Place track number in P or D part of PCB
    - o Change status from ifb to eb.
    - o Return buffers to ebq.
- ✓ if Task is OS
    - o Read information from given track into given eb

- Change status from eb to ofb
- Release track
- Decrement line count in PCB

✓ if Task is LD

- Load program card from given track into indicated memory block.
- Decrement count in PCB
- If zero, place PCB on RQ after all the initializations.

✓ if Task is GD

- Read data card from given track into indicated memory block.
- Decrement count in PCB
- Move PCB to RQ after setting TSC<---0

✓ if Task is PD

- Write information from the indicated memory block to the given track.
- Increment Total Line Count(TLC) in PCB
- if(TI=2 or 3),
  - Move PCB on TQ
- else

        - Move PCB to RQ after setting TSC<---0

- ✓ if Task is SQW
    - o Write the information from the victim frame to the given track.
    - o Locate drum track with faulted page
    - o Task<---SQR
    - o Start Channel3
- ✓ if Task is SQR
    - o Read drum track with faulted page in newly allocated frame.
    - o Move PCB, SQ--->RQ after setting TSC<---0.

Function Name:

EXECUTEUSERPROGRAM ();

Outline:

This procedure simulates the user mode operation of the MOS system.

Algorithm:

LOOP

- if(there is job in ready queue)
    - o MAP IC to RA

        - if PI != 0
        - {
            - Page Fault: It may be due to no program card is not loaded or illegal virtual address.

- END LOOP
        - }
        - Instruction Register(IR) <--- memory[RA]
        - IC <--- IC+1
        - Map the IR[3,4] to RA
        - if PI != 0
        - {
            - Page Fault: It may be due to illegal user address or virtual page is not mapped to physical frame.

- END LOOP
        - }
        - Examine IR[1,2]
            - LR: R <--- memory[RA]
            - SR: R ----> memory[RA]
            - CR: Compare R and memory[RA]
                - if equal C<---True

- else
  - C<---False
- BT: if C = T then IC<--- IR[3,4]
- GD: SI=1(Input Request)
- PD: SI=2(Output Request)
- H: SI=3(Terminate Request)
- Otherwise PI<---1(Operand Error)

- End-Examine

- Call the SIMULATION procedure, which simulates the behavior of timer hardware.
- If SI,PI,TI or IOI not equal to zero then
  - Exit from Loop
  - Switch into Master Mode
- else
  - Loop in Slave Mode

End-Loop

Function Name:

SIMULATION ();

Outline:

This procedure simulates the behavior of the Timer Hardware.

Algorithm:

- ✓ Increment Total Time Counter (TTC)
- ✓ if TTC= Total Time Limit(TTL) then TI<--2
- ✓ Increment TSC
- ✓ if TSC = Time Slice(TS),then TI<--1
- ✓ For all Channeli,i=1,2,3
    - o if(Channeli flag is busy)
        - Increment Channeli timer
        - if(Channeli Timer = Channeli Total time)
            - Increment IOI accordingly
            - Set Channel Completion Interrupt
- ✓ End-For

Function Name:

ADDRESS_MAP(Virtual Address(VA));

Outline:

This procedure map virtual address to physical address.

Algorithm:

- ✓ Let four bytes of PTR is denoted by: a_0,a_1,a_2,a_3
- ✓ Let Virtual Address is denoted by x_1(Most Significant Bit),x_2(Least Significant Bit).

- ✓ If(x1>a1)
    - o PI=3
    - o return
- ✓ Map the virtual address to physical address by using the formula $10*[10*(10a_2+a_3)+x_1]+x_2$, where $10a_2+a_3$, gives the number of the user storage block in which the page table resides. Multiplication of this number by 10 gives the base address of memory. Address of the page is obtained by adding x1 to it. Its content gives the memory block of the required information. Address of the actual memory location can be obtained by multiplying this number by 10 and adding offset to it.
- ✓ If page is not mapped to memory frame then
    - o Set PI=3
    - o return
- ✓ else
- ✓ Return Real address.

4 ANALYSIS

4.1 Modification on the MOS

One hundred and ninety-two different jobs are used to study the behavior of the MOS system on different page sizes. In addition, following changes are made on the original specification of the MOS.

4.1.1 Page Size

In original specification of a MOS system, only one page size was used, and its size was 40 bytes. In the present study, different page sizes are used. The smallest page size used for the study is 40 bytes and successive page sizes are calculated by using following formula:

PAGE_SIZE = $40*2^i$ for i=0,1, 2, 3…, n-1,

where n is total number of the different pages in the study. In this study, the value of n is fixed to be 18 because of the capacity of the physical memory of the experimental computer.

For example, the lowest page size used in the experiment is 40 *2^0 = 40 bytes. Similarly other page sizes are 80 bytes, 160 bytes,320 bytes, 640 bytes etc.

4.1.2 Virtual Memory

MOS in original specification had 400 bytes virtual memory and this memory was divided into ten ten-word blocks for paging purposes. The virtual memory is the LCM of all the page sizes for present study. The reason behind taking LCM is that virtual memory must be divisible by the page size to form the blocks and the lowest number divisible by all the page sizes is obviously the LCM of all these sizes. The number of the blocks in the virtual memory equals the size of the virtual memory divided by page size.

For example, the LCM of all the page sizes used in this experiment is found to be 5242880. So the virtual memory is 5242880 bytes for present study. Suppose we are studying MOS system with page size 80, then number of the blocks in the virtual memory equals 5242880/80 = 65536. Similar calculation can be applied for other page sizes too.

4.1.3 Disk Size

The disk in original specification of the MOS system contained 100 tracks, with 10 four-byte words per track. But this disk size is too small for the present study. So a large value is chosen as the disk size. To compute the disk size, we first evaluate the LCM of all the page sizes used in the study and this LCM is multiplied by 100 to get the number of tracks in the disk. The size of each track is equal to the page size.

For example, the disk size for current study is 524288000 bytes which is calculated by multiplying LCM of all the page sizes with 100. Let us consider we are analyzing the MOS system with page size of 40 bytes, then the numbers of the tracks in the disk are 13107200 and size of each track equals 40 bytes. Similarly, if the page size is 80 bytes, then the numbers of tracks in the disk are 6553600 and size of each track equals 80 bytes. In each case the total disk size remains constant.

4.1.4 Memory Size

In original specification of the MOS system, user storage had 300 four-byte words. It was divided into 30 ten-word blocks for paging purposes. The user storage is obtained by multiplying LCM of the page size by 30 for this study. The number of blocks in memory is obtained by dividing the memory size by page size. Thus now the user storage contains (LCM*30)/PAGE_SIZE PAGE_SIZE/WORD_SIZE blocks.

For example, multiplication of LCM of all the page sizes with 30 gives the user storage and its value equals 157286400 bytes for current system. If we are going to study the MOS system with page size 40, then this user storage is divided into 3932160(157286400/40) ten (40/4) - word blocks. Similarly, when we study MOS system with page size 640, the user storage is divided into 245760(157286400/640) 160(640/4) - word blocks. In each case, total user storage remains constant.

4.1.5 Virtual to Physical mapping

Previously in original specification of MOS system, a two digit operand address, x_1x_2 in virtual space was mapped by the relocation hardware into the real user storage address by using the mapping $10[10(10a_2+a_3)+x_1]+x_2$, where $10a_2+a_3$ is the number of the user storage block in which page table resides and [x] refers to the content at the memory location x. Now the mapping is modified as Block_Size *[Block_Size $(10a_2+a_3)$ $+x_1]$ $+x_2$,where, Block_Size = PAGE_SIZE/WORD_SIZE. Other symbols have their usual meaning.

For example, consider a two digit virtual address 80 is going to map into the real address. If the page size under study is 40 bytes, and page table of the job resides at 5^{th} block of memory, then value of $10a_2+a_3$ equals five. Further assume that the word size for the MOS machine is fixed to be 4. Under these assumptions, we will have

$x_1 = 8$ and $x_2 = 0$

Block_Size = 40/4 = 10

Real Address = 10 *[10*5+8] + 0

= 10 * [58] + 0

Now suppose content at the memory location 58 is 9, then

Real Address = 10 *9+0 = 90.

If we take the PAGE_SIZE = 80, then mapping of virtual address 80 proceeds as follows:

Block_Size = 80/4 = 20

Real Address = 20 *[20*5+8] + 0

= 20 * [108] + 0

Suppose content at the memory location 108 is 9, then

Real Address = 20 *9 + 0

= 180

4.2 Parameters under Study

Following are the different parameters under study:

- ✓ Page Fault,
- ✓ Memory Fragmentation,
- ✓ Storage requirement for the page table,

4.2.1 Page Fault

One global variable is used to calculate the total number of page fault. Its value is initially set to zero and when the valid page fault occurs, the value is incremented by one. When program terminates, the value of this variable gives total number of page fault.

4.2.2 Memory Fragmentation

Memory fragmentation is calculated by finding the empty space in the final page. Therefore, to calculate fragmentation for a job, at the time of releasing the memory, we subtract the total space occupied by the content in the final page from the page size. The memory fragmentation calculated, by this way, for each job, is summed together, and stored in a variable. After completion of all jobs, the

content of the variable gives the total memory fragmentation for a run.

For example, suppose page size under study is 40 bytes. Assume content at the final page is 25 bytes. The fragmentation therefore equals to 40 – 25 = 15 bytes. This procedure of calculating fragmentation is repeated for all the jobs, and the fragmentation for each job is summed together. After completion of all jobs, we get the total memory fragmentation for a run.

4.2.3 Storage requirement for the page table

If we know the number of entries in the page table and each entry in page table takes 4 bytes, then the following formula can be used for calculating the storage requirement:

Storage Requirement for page table = memory occupied by each page table entry (4 bytes) * Total number of entries

4.3 Experimental Data

S.No	*Page Size*	*Virtual Memory Size(in byte)*	*Disk Size (in byte)*	*Memory Size (in byte)*	*Word Size(byte)*
1	40	5242880	524288000	157286400	4
2	80	5242880	524288000	157286400	4
3	160	5242880	524288000	157286400	4
4	320	5242880	524288000	157286400	4
5	640	5242880	524288000	157286400	4
6	1280	5242880	524288000	157286400	4

S.No	Page Size	Virtual Memory Size(in byte)	Disk Size (in byte)	Memory Size (in byte)	Word Size(byte)
7	2560	5242880	524288000	157286400	4
8	5120	5242880	524288000	157286400	4
9	10240	5242880	524288000	157286400	4
10	20480	5242880	524288000	157286400	4
11	40960	5242880	524288000	157286400	4
12	81920	5242880	524288000	157286400	4
13	163840	5242880	524288000	157286400	4

S.No	*Page Size*	*Virtual Memory Size(in byte)*	*Disk Size (in byte)*	*Memory Size (in byte)*	*Word Size(byte)*
14	327680	5242880	524288000	157286400	4
15	655360	5242880	524288000	157286400	4
16	1310720	5242880	524288000	157286400	4
17	2621440	5242880	524288000	157286400	4
18	5242880	5242880	524288000	157286400	4

Table 4-1 Experimental Data

5 RESULTS AND DISCUSSION

5.1 Presentation of the finding

S.No	***Page Size***	***Page Fault***	***Memory Fragmentation (in byte)***	***Storage Requirement for the page table (in byte)***
1	40	1416	3888	524288
2	80	804	7855	262144
3	160	492	15042	131072
4	320	300	30964	65536
5	640	192	62923	32768

S.No	Page Size	Page Fault	Memory Fragmentation (in byte)	Storage Requirement for the page table (in byte)
6	1280	192	126486	16384
7	2560	192	256533	8192
8	5120	192	514592	4096
9	10240	192	1030692	2048
10	20480	192	2062887	1024
11	40960	192	4027272	512
12	81920	192	7959680	256

S.No	*Page Size*	*Page Fault*	*Memory Fragmentation (in byte)*	*Storage Requirement for the page table (in byte)*
13	163840	192	16413577	128
14	327680	192	32028649	64
15	655360	192	65058793	32
16	1310720	192	132119082	16
17	2621440	192	264239658	8
18	5242880	192	518480810	4

Table 5-1 Experimental Result

5.2 Discussion of the finding

5.2.1 Relationship between Page Size and Memory Fragmentation

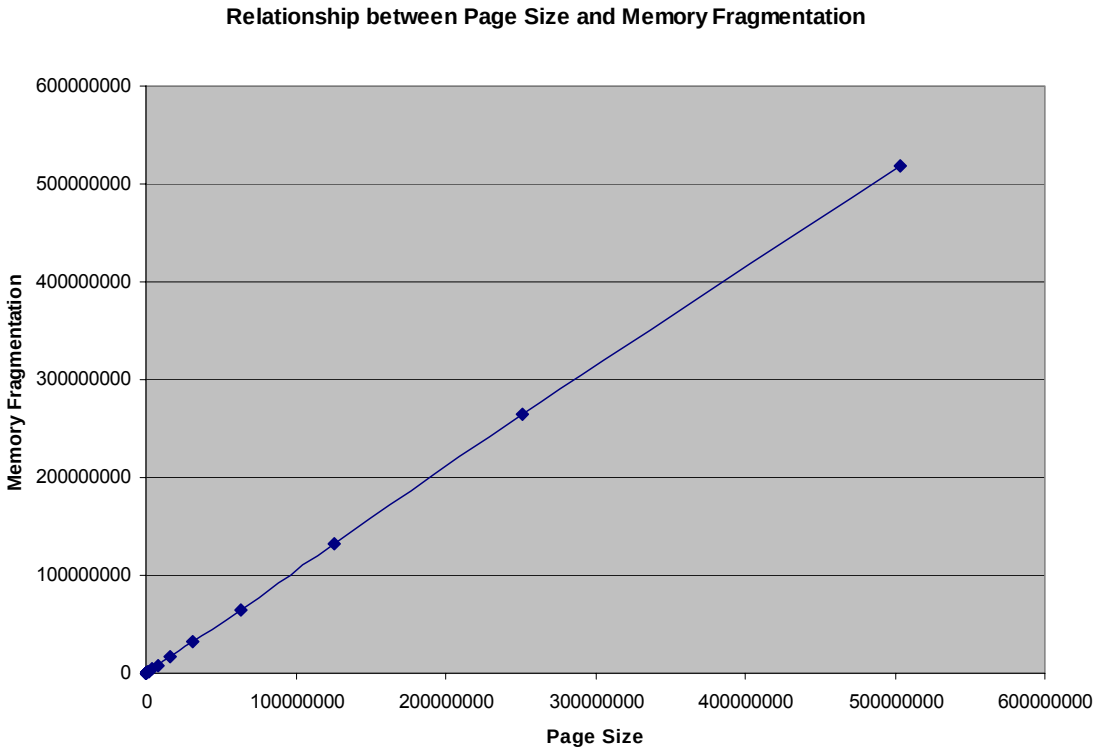

Figure 5-1 Relationship between Page Size and Memory Fragmentation

Figure 5-1 shows the relationship between the page size and memory fragmentation. Initially the page size is small, the memory fragmentation is also small, as the page size increases the fragmentation also increases. The reason behind increasing the memory fragmentation with page size is that a randomly chosen data or an instruction segment of a user job may not fill an integral number of pages. The portion of the page, in which no data or instruction is placed is wasted as the internal fragmentation. This wastage in the page

increases with the increase in the page size. Hence, the memory fragmentation increases with the page size.

The result found from this experiment is in accordance with the Prof. Andrew S. Tanenbaum argument about relationship between page size and memory fragmentation. According to Tanenbaum, with n segments in memory and a page size of p bytes, np/2 bytes will be wasted on internal fragmentation. Table 5-2 shows a difference between the calculated values of memory fragmentation using Tanenbaum formula and memory fragmentation found during experiment. The experimental value of the memory fragmentation is obtained from the table 5-1.

S.No	Page Size	Memory Fragmentation obtained using Tanenbaum formula for each job(A)	Memory fragmentation obtained through experiment (B)	Percentage of difference between two values = X= (|A-B|)*100/A
1	40	192*40/2 = 3840	3888	1.234568
2	80	192*80/2 = 7680	7855	2.22788
3	160	192*160/2 = 15360	15042	2.114081
4	320	192*320/2=30720	30964	0.788012
5	640	192*640/2=61440	62923	2.356849
6	1280	192*1280/2 = 122880	126486	2.850908
7	2560	192* 2560/2 = 245760	256533	4.19946
8	5120	192* 5120/2 = 491520	514592	4.483552
9	10240	192* 10240/2 = 983040	1030692	4.623302
10	20480	192* 20480/2 = 1966080	2062887	4.692792

11	40960	192* 40960/2 = 3932160	4027272	2.361698
12	81920	192* 81920/2 = 7864320	7959680	1.198038
13	163840	192 * 163840/2 = 15728640	16413577	4.17299
14	327680	192 * 327680/2 = 31457280	32028649	1.783931
15	655360	192 * 655360/2 = 62914560	65058793	3.295839
16	1310720	192 * 1310720/2 = 125829120	132119082	4.760828
17	2621440	192 * 2621440/2 = 251658240	264239658	4.761366
18	5242880	192 * 5242880/2 = 503316480	518480810	2.924762

Table 5-2 Difference Between Experimental and Theoritical Values of Memory Fragmentation

Sum of Percentage of difference between two values (∑X) = 54.83086

Arithmetic Mean = ∑X / N = 54.83086/18 = 3%

From the above statistical calculation it is clear that, on average, difference between the theoretical value of the memory fragmentation and experimentally found value is very low i.e. 3%. This difference in value can be justified from the fact that formula used to calculate the theoretical value of memory fragmentation is not exact but it is an average. Hence, this study strongly favors the Tanenbaum argument on the memory fragmentation.

5.2.2 Relationship between Page Size and Page Fault

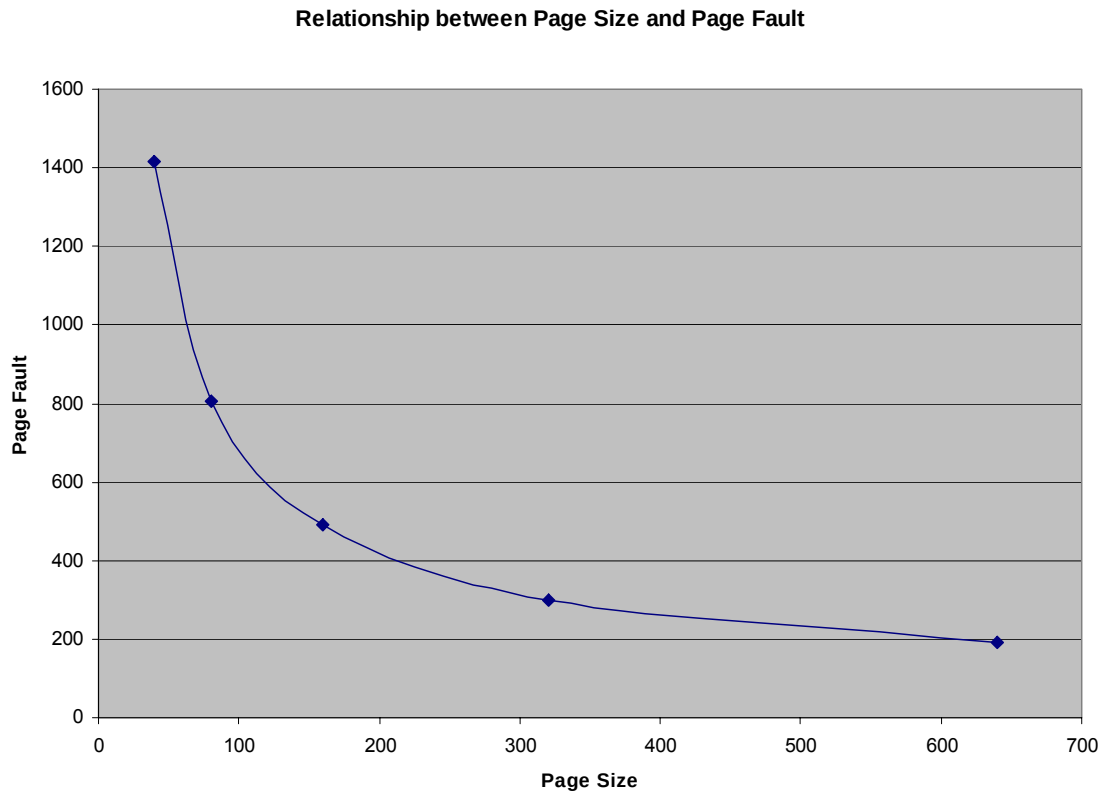

Figure 5-2 Relationship between Page Size and Page Fault

Figure 5-2 shows the relationship between the page size and page fault. The page fault decreases as the page size increases. This happens because increase in page size decreases the number of entries in the page table. As a result, more addresses map to the same page table. If the page is not evicted by swapping, there will be page fault for the first time for an access to that page

table entry. For the subsequent access to the same page table entry, there is no page fault, since that page table entry has already been mapped to the physical memory block. Because of this reason, the page fault decreases with the increase in page size.

Chu and Opderbeck's [2], and Bennet[3] found different result on the relationship between page size and page fault. In Chu and Opderbeck's [2] study, they found that page fault is heavily dependent on the page size. But Bennet found no concise relationship between these two. Later on Fagin, R. A[7] explained the factor behind this difference .

According to Fagin, R. A. [7], this difference was caused by taking consideration of the initial loading miss. According to him, if we take the consideration of initial loading miss, then page fault heavily depends on page size. On the other hand, if we do not take consideration of initial loading miss, then effect of page size on page fault is minimum. Chu and Opderbeck's [2] took consideration of initial loading miss while Bennet didn't.

Table 5-3 shows the page fault for first few jobs. Dividing the page fault of a run (Given in Table 5-1) by total number of job gives the page fault for each job.

S.No	Page Size	Page Fault for each Job (X)
1	40	1416/192 = 7.375
2	80	804/192 = 4.1875
3	160	492/192 = 2.5625
4	320	300/192 = 1.5625
5	640	192/192 = 1
6	1280	192/192 = 1

Table 5-3 Page fault for first few job

The input program used for this study references virtual memory location 20 to 90 more, than other memory location. This implies that about seven pages of the page table are referenced heavily by a job. Therefore, on average when half of pages get loaded, the program can work smoothly. Thus, the initial loading miss approximately equals 3 for this study.

Total number of page fault $\sum X = 17.6875$

Average number of page fault = $\sum X/N = 17.6875/6 = 2.947917 =$ 3(Approximate)

From the above statistical calculation, it is clear to us that the total number of the page fault found from experiment is nearly equal to the initial loading misses. Therefore, from the experiment we can conclude that the page size affects page fault more when we take consideration into initial loading misses, and if we do not, the effect of the page size on page fault is minimum. This is exactly the conclusion drawn by Fagin, R. A[7].

Hence, this study supports the argument given by Fagin, R. A. to explain the difference of the study made by Chu and Opderbeck's [2] and Bennet[3].

5.2.3 Relationship between Page Size and Storage Requirement for page table

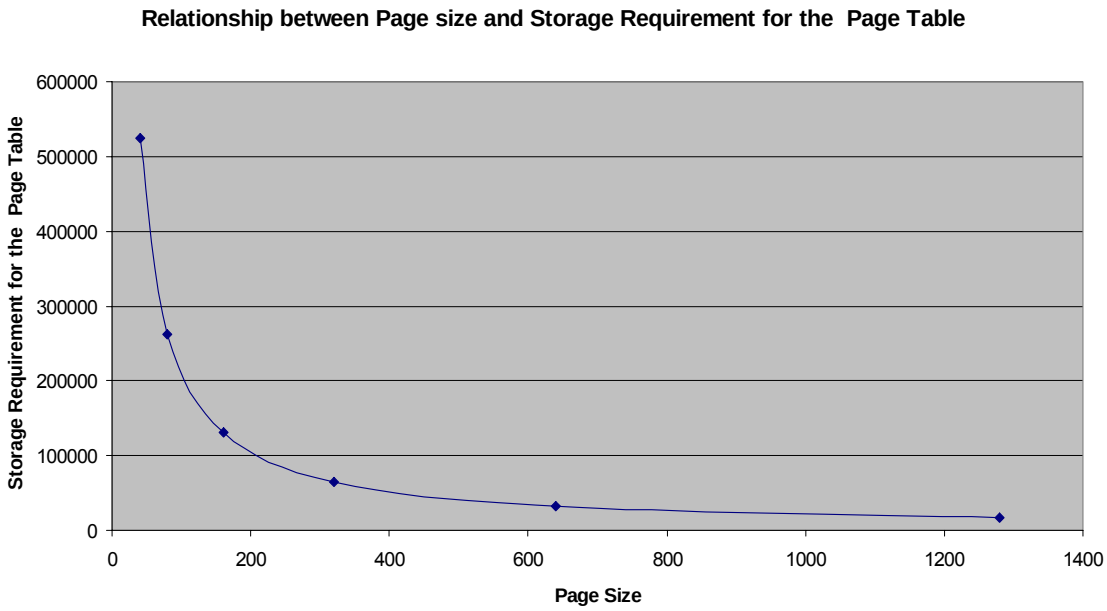

Figure 5-3 Relationship between Page size and Storage Requirement for the Page Table

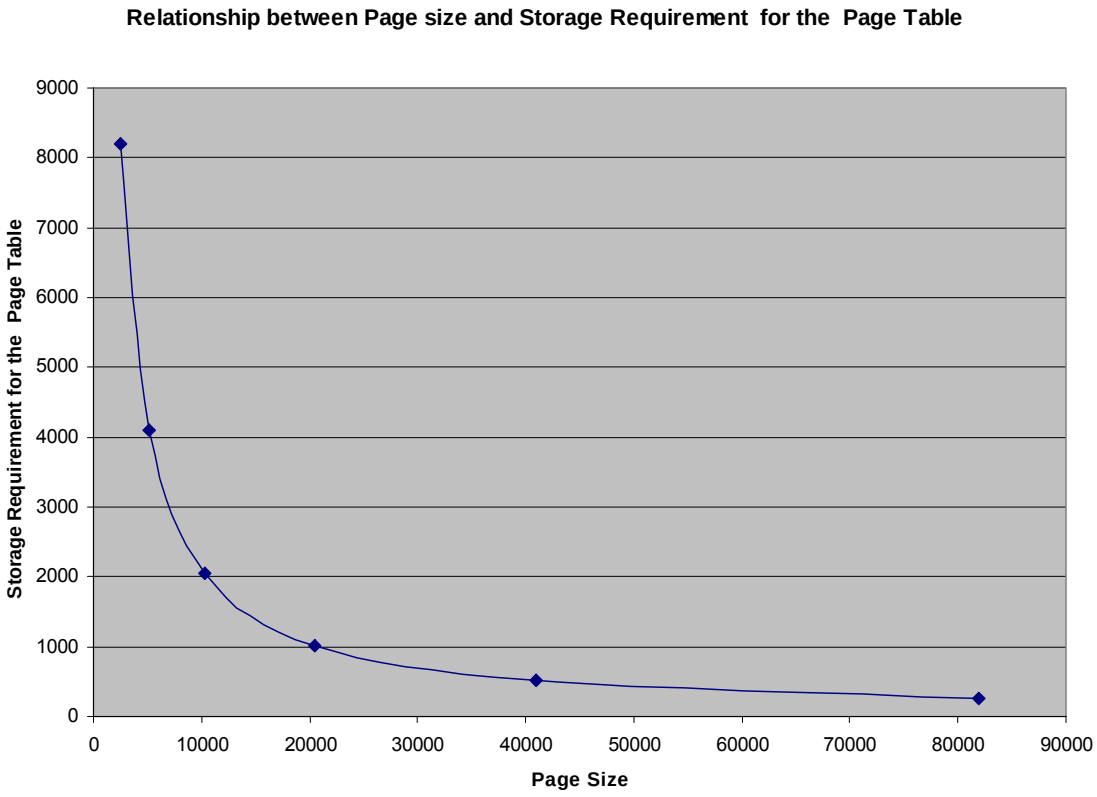

Figure 5-4 Relationship between Page size and Storage Requirement for the Page Table

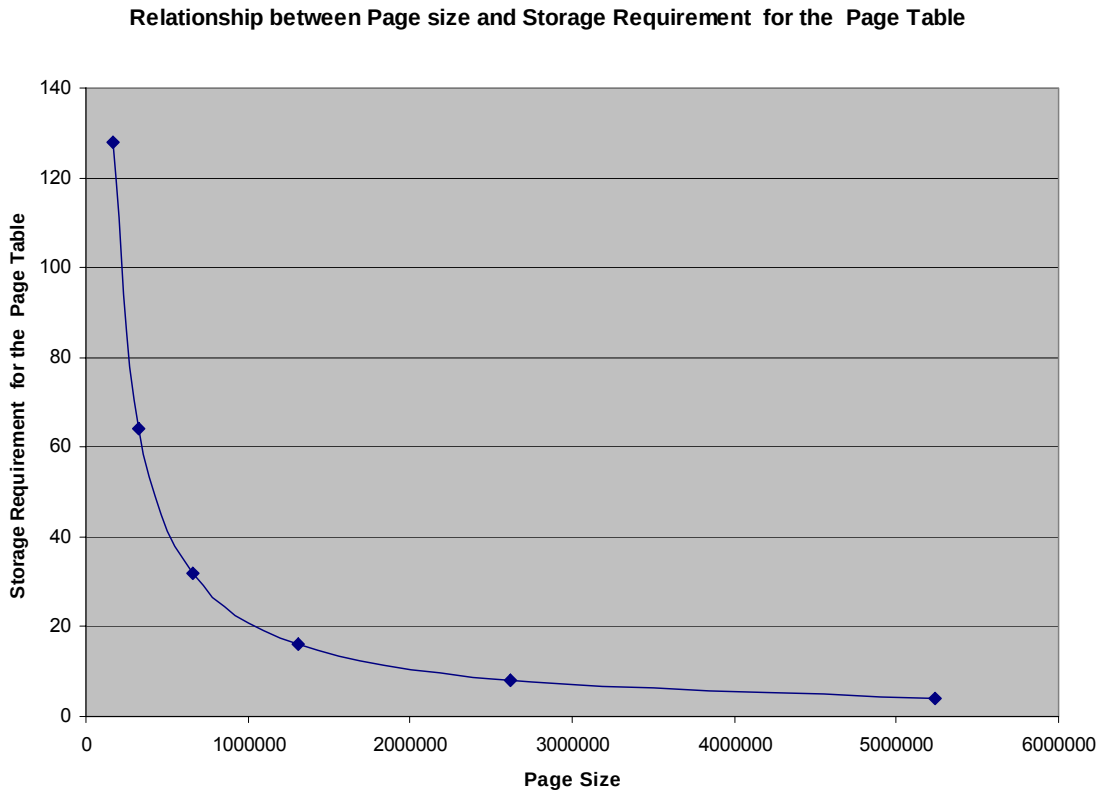

Figure 5-5 Relationship between Page size and Storage Requirement for the Page Table

Figure 5-3, Figure 5-4 and Figure 5-5 show the relationship between the page size and storage requirement for the page table. The observation from these figures indicates that as the page size increases, storage requirement for the page table decreases. This behaviour is due to the fact that as the page size is large, there will be fewer number of entries in the page table.The storage requirement for the page table is diretly proportional to the number of entries in the page table.So fewer the number of page table entries, lesser the storage required for the page table. On the other hand, as the page size decreases, the storage requirement for the page table increases.The reasoning behind this is

similar to the above argument. Decrease in the page size increases the page table entries and when the number of page table entry increases; the storage requirement for the page table also increases.

Prof. Andrew S. Tanenbaum[1] in his book Modern operating systems analyzes the relationship between page size and storage requirement of the page table mathematically. His analysis shows that if process size is s byte, and the page size is p bytes, and each page entry requires e bytes, then the approximate number of pages needed per process is s/p bytes occupying se/p bytes.

Size of the process is dependent on the underlying machine architecture. For example in 32 bit machine architecture, the size of user program could be 2^32 = 4GB.Further, the storage requirement for each page table entry is equal to the lowest addressable unit of the underlying machine. The process size and the storage requirement of each page table entry equals the size of the virtual memory and the word size of the MOS machine for this study. The values of these parameters are 5242880 bytes and 4 bytes, respectively, for the present study. Thus for a particular machine architecture, the process size and the storage requirement for each page table entry are constant. Hence, storage requirement for the page table is inversely proportional to the page size.

Mathematically we can write,

$$\text{Storage Requirement for the page table} \propto 1/\text{Page\_SIZE}$$

The experimental result further shed light on the above statement of

relationship between storage requirement for the page table and page size. The experiment shows that storage requirement for the page table decreases with the increase in the page size increases with decrease in page size. Therefore, storage requirement of the page table is inversely proportional to the page size.

Thus, present study experimentally verifies Tanenbaum mathematical argument of the relationship between storage requirement of the page table and page size.

5.3 Optimal Page Size

This study shows that the memory fragmentation increases and storage requirement decreases in proportion to the page size. However, the page fault does not decrease in proportion to the page size. If we do not take the consideration of the initial loading misses, then page size has minimum effect on the page fault.

The storage requirement for the page table can be lowered sufficiently by using the multilevel paging. Only one memory block is sufficient to store the page table when we use the multilevel paging technique. The only demerit of the multilevel page table is that mapping of virtual address to physical one takes longer time than with single level page table. If we have only 2 or 3 level of page table then storage requirement for the page table outage the time needed to map an address. Therefore, modern operating system such as Linux [10] uses the multilevel page table.

For the MOS system, the page size has minimum effect on the page fault after the working set of a process has been loaded into the memory. Further, the storage requirement for the page table can be minimized using multilevel paging technique. Therefore, the important criterion for choosing the page for MOS system is the fragmentation of memory.

Since the memory fragmentation is lowest for the page size 40 bytes, the optimal page size for the MOS system is 40 bytes.

The page size in the original MOS system was specified to be 40 bytes by the author Prof. Dr. Onkar P. Sharma [9]. He must have taken the consideration of optimal page size while specifying the MOS machine. This experiment supports his decision of choosing page size of 40 bytes for MOS machine.

6 CONCLUSION

While studying the behavior of MOS system on different page sizes, the influence of the page size on following parameters are studied:

- Memory Fragmentation
- Page Fault
- Storage requirement for the page table

From this study, following four conclusions are drawn:

I) The memory fragmentation and page size are directly proportional to each other. The experimentally found result supports the Tannebaum [1] view of memory fragmentation i.e. on the average; half of the final page will be empty.

II) The page fault depends on page size heavily if we take into consideration the initial loading miss. The effect of the page size on page fault is minimum after the working set of a process gets loaded. This result supports the conclusion drawn by Fagin, R. A. [7], who explained the reason behind two contradictory results found by Chu and Opderbeck's [2] and Bennet[3] study.

III) The storage requirement for the page table and page size is

inversely proportional to each other. This result from the experiment verify the mathematical relationship given by the Tanenbaum[1] between storage requirement for the page table and page size.

IV) Finally, the optimal page size for the MOS system is found to be 40 bytes. The experimentally found optimal value of the page size is in accordance with the original specification of the MOS system given by Prof. Dr. Onkar P. Sharma[9].The author must have chosen the page size by taking consideration of the optimal value. Present study favors the decision made by the author for choosing page size.

7 LIMITATION AND FUTURE WORK

The effect of page size on the transfer time from the disk has not been touched in present study. The transfer time from the disk is one of the critical parameter for not choosing small page sizes.

In future, the effect of page size on transfer time from the disk will be studied. The MOS will be modified to support multiple page sizes simultaneously, and the behavior of the MOS system will be studied.

REFERENCES

1) Tanenbaum, Andrew S. Modern Operating Systems, Prentice Hall of India, 2004

2) CHU,W.W., And Opderbeck, H, Performance of the replacement algorithm with different page sizes, Computer 7, 11(Nov 1974).

3) Bennet B. T. Private Communication

4) Kalpan K.R., and Winder, R.D. Cache Based Computer Systems, Computer 6, 3 (March 1973), 30-36

5) Lewis,P.A.W. and Shelder G.S. Emperically,derived micromodels of for sequences of page exception IBM J. Res. Develop. 17, 2(March 1973),86-100

6) Anacker, W, and Wang, C.P. Performance evaluation of computing systems with memory hierarchies IEEE Trans. Electronic Computers EC-16(1967) 765-773

7) Fagin, R. A counterintuitive example of computer paging. Res. Rep. RC 5031, IBM Thomas J. Watson Research Center, Yorktown Heights, N Y , Aug. 1974; Comm. ACM.

8) N. Ganapathy and C. Schimmel. General purpose Operating

System for Multiple page Sizes. Proceedings of the USENIX 1998 Annual Technical Conference, Berkeley, CA, June 1998.

9) Sharma, Onkar P., Enhancing Operating System Course Using a Comprehensive Project: Decades of Experience Outlined, Consortium for Computing Sciences in Colleges,2007

10) www.linux.com

BIBILOGRAPHY

1. Sharma, Onkar P., An Operating System project, it's accompanying problems and their object-oriented design solution, The Journal of Computing for small Colleges, Vol. 8, No. 2, Nov 1992.
2. Shaw, Alan C. The Logical Design of Operating Systems, Prentice Hall, 2003.
3. Peter J. Denning , Virtual Memory, Computing Surveys, September, 1970
4. Douglas W. Clark and Joel S. Emer. Performance of the VAX 11/780 Translation Buffer: Simulation and Measurement. ACM Transactions on Computer Systems (February 1970).
5. King, W.F. III Analysis of Paging Algorithms IFIP Conf Proc , Ljubljana, Yugoslavia, Aug 1971.
6. Madhusudhan Talluri, Mark D. Hill and Yousef A. Khalidi. A new page table for 64-bit address spaces. In Proc. Of Symposium of Operating System Principles(SOSP), Dec 1995.
7. Comer, D. and Fossum T. Operating System Design,The Xinu Approach, Prentice Hall, Inc 1998 .
8. Silberschatz, A. and P.B. Galvin, Operating System Concepts, Fifth Edition. John-Wiley 1998

9. Anacker, W, and Wang, C.P. Performance evaluation of computing systems with memory hierarchies IEEE Trans. Electronic Computers EC-16(1967) 765-773

10. Lewis,P.A.W. and Shelder G.S. Emperically,derived micromodels of for sequences of page exception IBM J. Res. Develop. 17, 2(March 1973),86-100

11. John K. Ousterhout . Why Operating Systems Aren't Getting Faster As Fast
Hardware. Proceedings of the Summer 1991 USENIX Conference, June, 1991.

APPENDIX A SAMPLE INPUT PROGRAM

$AMJ001110001000

GD20GD30GD40GD50PD20PD30PD40LR50SR31PD30

GD60PD60H

$DTA

This is

uhhhhhhh

ADS

mmmm

Advanced Data Structures

$END0011

$AMJ001210001000

GD50PD50GD60PD60GD70LR70SR50PD50PD60LR71

SR50PD50PD60LR72SR50PD50PD60LR73SR50PD50

PD60LR74SR50PD50PD60LR75SR50PD50PD60LR76

SR50PD50PD60LR77SR50PD50PD60LR78SR50GD80

PD80PD90H

$DTA

9 Bottles of Beer on the wall

Take one down,pass it around

8 7 6 5 4 3 2 1 0

NO BOTTLES OF BEER ON THE WALL!!

$END0012

$AMJ002100221000

GD50PD50GD60PD60GD70PD70GD80LR80SR50PD50

PD50SR60PD60PD70GD80LR80SR50PD50PD50SR60

PD60PD70GD80LR80SR50PD50PD50SR60PD60PD70

GD80LR80SR50PD50PD50SR60PD60PD70GD80LR80

SR50PD50H

$DTA

5 bottles of beer on the wall

5 bottles of beer,

Take 1 down, pass it around

4 bo

3 bo

2 bo

1 bo

0 bo

$END0021

$AMJ002200140014

GD20PD20GD30PD30GD40PD40GD50PD50PD20GD20

PD20PD30H

$DTA

KNOCK KNOCK!

WHO IS THERE?

HAHA

WHO

HEHE

$END0022

$AMJ003110001000

GD40PD40GD50PD50GD60LR60SR40PD40PD50GD60

LR60SR40PD40PD50GD60LR60SR40PD40PD50GD60

LR60SR40PD40GD70PD70GD80PD80GD60LR60SR40

PD40PD50GD60LR60SR40PD40PD50GD90PD90H

$DTA

6 Seconds till the world is over

We're all gonna die!

5

4

3

1

1, what happened to 2??

Just kidding

2

1

KABOOOOOOOOOMMMMM!!!!!!

$END0031

$AMJ003210001000

GD30GD40LR40SR33GD50LR50SR34GD60LR60SR35

GD70LR70SR36PD30GD80PD80GD90PD90GD90PD90

GD90PD90GD90PD90GD90PD90H

$DTA

Operating

Syst

ems

is

fun

I really love this class!!

There is nothing else like it

The final project is so fun!!

I can't wait to take the final exam.

$END0032

$AMJ004110001000

GD20PD20GD30LR30SR20PD20GD40LR40SR20PD20

GD50LR50SR20PD20GD60PD60H

$DTA

One fish

Two

Red

Blue

By Dr. Seuss

$END0041

$AMJ004200061000

GD30PD30GD40PD90GD50PD50GD60PD60GD70LR70

SR30PD30PD40PD50PD60GD80LR80SR30PD30PD40

PD50GD90PD90H

$DTA

3 os students jumping on the bed

One fell off and bumped her head

Mama called Dr.Sharma & Dr. Sharma said

No more os students jumping on the bed

2 os

1 os

"Send those os students back to class!"

$END0042

$AMJ005100280004

GD40PD40LR40SR64LR41SR63LR42SR62LR43SR61

LR44SR60PD60LR40CR40BT12SR80PD80LR41SR80

PD80LR42SR80PD80LR43SR80PD80LR44SR80H

$DTA

P I Z Z A

$END0051

$AMJ005200800020

GD40LR41SR50SR51LR40SR60SR61SR62LR42SR70

LR44SR71LR45SR72LR42SR80LR42SR81LR45SR82

LR44SR90LR42SR91LR43SR92PD50PD70PD50PD60

PD50PD80PD50PD60PD50PD90PD50H

$DTA

---- | X | X O | O

$END0052

$AMJ006109000900

GD40PD40GD50LR50SR40PD40GD50LR50SR40PD40

GD50LR50SR40PD40GD50LR50SR40PD40GD50LR50

SR40LR51SR42LR52SR43LX53SR44LR54SR45PD40

H

$DTA

0 SECONDS

1 SE

2 SE

3 SE

4 SE

5 SES INTO THE NEW YEAR

$END0061

$AMJ006209000900

GD40PD40GD50LR50SR40PD40GD50LR50SR40PD40

GD50LR50SR40PD40GD50LR50SR40PD40GD50LR50

SR40LR51SR42LR52SR43LR53SR44LR54SR45PD40

H

$DTA

5 SECONDS

4 SE

3 SE

2 SE

1 SE

0 SES HAPPY NEW YEAR

$END0062

$AMJ007110001000

GD40GD50LR50SR45LR51SR46PD40GD60PD60LR52

SR45LR53SR46PD40GD70PD70LR54SR45LR55SR46

PD40GD80PD80GD90PD90LR50SR45LR51SR46PD40

H

$DTA

Oompa Loompa doompad

ee doo ah dee ee dah

I've got another puzzle for you

If you are wise you'll listen to me

If you're not greedy you will go far

You will live in happiness too

$END0071

$AMJ007210001000

GD30PD30GD40PD40GD50PD50GD60PD60GD70LRX0

SR60PD60LR71SR60PD60LR72SR60PD60LR73SR60

PD60LR74SR60PD60GD80PD80H

$DTA

WELCOME TO SHARMA'S SODA MACHINE

PLEASE INSERT 1.25 FOR A SODA.

OUR MACHINE ONLY ACCEPTS QUARTERS

0.00 HAS BEEN INSERTED.

0.250.500.751.001.25

PLEASE TAKE YOUR SODA!

$END0072

$AMJ008100400009

GD50LR50SR90PD90GD60LR60SR91PD90GD70LR70

SR92PD90GD80LR80SR93PD90GD40LR40SR94PD90

GD40LR40SR94PD90LR40SR93PD90LR40SR92PD90

LR40SR91PD90H

$DTA

1

2

3

4

5

$END0081

$AMJ008200530020

GD50PD50GD60LR60SR50PD50GD60LR60SR50GD70

LR70SR51PD50GD60LR60SR50GD70LR70SR51PD50

GD60LR60SR50PD50GD60LR60SR50GD70LR70SR51

PD80H

$DTA

One sheep

Two

Thre

e

Red

Blue sheep

gree

n

$END0082

APPENDIX B SAMPLE OUTPUT

```
This is
uhhhhhhh
ADS
uhhhmmmm
Advanced Data Structures


Job Id:0011
No Error


9   Bottles of Beer on the wall
Take one down,pass it around
8   Bottles of Beer on the wall
Take one down,pass it around
7   Bottles of Beer on the wall
Take one down,pass it around
6   Bottles of Beer on the wall
Take one down,pass it around
5   Bottles of Beer on the wall
Take one down,pass it around
4   Bottles of Beer on the wall
Take one down,pass it around
3   Bottles of Beer on the wall
Take one down,pass it around
2   Bottles of Beer on the wall
Take one down,pass it around
1   Bottles of Beer on the wall
Take one down,pass it around
NO BOTTLES OF BEER ON THE WALL!!


Job Id:0012
Invalid Page Fault


5 bottles of beer on the wall
5 bottles of beer,
Take 1 down, pass it around
```

4 bottles of beer on the wall
4 bottles of beer on the wall
4 bottles of beer,
Take 1 down, pass it around
3 bottles of beer on the wall
3 bottles of beer on the wall
3 bottles of beer,

Job Id:0021
Time Limit Exceeded

KNOCK KNOCK!
WHO IS THERE?
HAHA
WHO
KNOCK KNOCK!
HEHE
WHO IS THERE?

Job Id:0022
No Error

6 Seconds till the world is over
We're all gonna die!
5 Seconds till the world is over
We're all gonna die!
4 Seconds till the world is over
We're all gonna die!
3 Seconds till the world is over
We're all gonna die!
1 Seconds till the world is over
1, what happened to 2??
Just kidding
2 Seconds till the world is over
We're all gonna die!
1 Seconds till the world is over
We're all gonna die!
KABOOOOOOOOOOMMMMM!!!!!!

Job Id:0031
No Error

Operating Systems is fun
I really love this class!!
There is nothing else like it
The final project is so fun!!
I can't wait to take the final exam.

Job Id:0032
Out Of Data

One fish
Two fish
Red fish
Blue fish
By Dr. Seuss

Job Id:0041
No Error

3 os students jumping on the bed

Job Id:0042
Invalid Page Fault

P I Z Z A
A Z Z I P
A Z Z I P
A Z Z I P

Job Id:0051
Line Limit Exceeded

```
   |   |
 X | O | O
   |   |
-----------
   |   |
 X| X | O
   |   |
-----------
   |   |
```

O| X | X
   | |

Job Id:0052
No Error

0 SECONDS
1 SECONDS
2 SECONDS
3 SECONDS
4 SECONDS

Job Id:0061
Operation Code Error

5 SECONDS
4 SECONDS
3 SECONDS
2 SECONDS
1 SECONDS
0 SECONDS HAPPY NEW YEAR

Job Id:0062
No Error

Oompa Loompa doompadee doo
I've got another puzzle for you
Oompa Loompa doompadah dee
If you are wise you'll listen to me
Oompa Loompa doompadee dah
If you're not greedy you will go far
You will live in happiness too
Oompa Loompa doompadee doo

Job Id:0071
No Error

WELCOME TO SHARMA'S SODA MACHINE
PLEASE INSERT 1.25 FOR A SODA.
OUR MACHINE ONLY ACCEPTS QUARTERS

0.00 HAS BEEN INSERTED.

Job Id:0072
Operand Error

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

Job Id:0081
No Error

One sheep
Two sheep
Three sheep
Red sheep
Blue sheep

Job Id:0082
Invalid Page Fault