The Design and Implementation of a Multi-Programming Virtual Memory Operating System for a Mini-Computer

by Lee Parks

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the
Massachusettes Institute of Technology
May, 1979

Signature redacted

Signature of Author_____
Department of Physics
May, 1979

Signature redacted

Certified by_____
Thesis Supervisor

Signature redacted

Accepted by_____
Chairmen, Departmental Committee On Theses

The Design and Implementation of a Multi-Programming Virtual Memory Operating System for a Mini-Computer

by Lee Parks

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
in Physics
at the
Massachusettes Institute of Technology
May, 1979

Abstract

This paper describes the implementation, design methodology and design tools of the MagicSix operating system developed at the Architecture Machine Group of the MIT Department of Architecture. The system was implemented on an Interdata 7/32 32-bit mini-computer and heavily relies the use of a segmented virtual address space. Both the goals of MagicSix and the way in which people participated in the project are discussed.

Thesis Advisor: Nicholas Negroponte
Associate Professor of Computer Graphics
Department of Architecture

Acknowledgements

I feel I should start by acknowledging the MIT Undergraduate Research Opportunities Program which helped sponsor most of the work that led to MagicSix. Next, I would like to thank all the members of the Architecture Machine Group for their help, assistance and friendship during my years at the lab. The ArchMach has provided a unique and stimulating environment in which to work. I would like to extend special thanks to Professor Nicholas Negroponte who let MagicSix follow its course for three years. To Mike Kazar, the father of MagicSix, Ted Anderson, and Rick Kovalcik, my fellow cohorts, I wish to say thank you for your friendship and assistance. I wish to thank Seth Steinberg, my boss at ArchMach, for all his help over the years and particularly for reading the draft of my thesis. I perhaps owe the deepest intellectual debt to Bernie Greenberg who guided me into the secrets of Multics. I also wish to thank the systems programmers of the MIT Artificial Intelligence Lab and Dave Moon in particular for his especially useful insight. I can't forget SIPB, the people who got me in to this mess. Last, but certainly not least, I wish to thank Pandora Berman for turning my thesis into English.

CONTENTS

FIGURES

1. Introduction

One might ask why systems programming, and in particular operating systems programming, has such a bad reputation. Operating systems are clearly necessary if programmers are to get down to the real work: solving problems instead of fighting the computer. It became clear in the early days of computing that a bare computer was not be sufficient. The very earliest computer users spent much of their time duplicating other's subroutine libraries. The world waited for someone to create a computer system that allowed the programmer to concentrate on the problem at hand rather than on a hundred side issues. If computers were ever to become everyday tools then they would have to become more adapted to people.

Enter the operating system-- the program that would provide all the services needed to the programmer not intrinsic to the hardware. It seemed that those who wrote operating systems had a golden opportunity to become the saviors of the computer industry and perhaps they had, however, much blood has been spilled along a road paved with good intentions. A complete analysis of why "operating system" has become a dirty word is beyond the present scope but a summary is in order. It will provide the context in which the Architecture Machine Group undertook MagicSix.

The seeds of failure lie in the very problem an operating systems designer begins with: Designing a program which will make

this machine a hospitable environment. Often this broad statement is narrowed by adding a taxonomy of specific tasks for the system to accomplish, for example, "This system shall make it easy to perform real-time processing." Nonetheless, what the designer is presented with is an open-ended problem with a myriad of options and conflicting requirements. No matter what the system manages to achieve, it usually cannot satisfy every design goal. Moreover as new applications are continually being planned, the system must evolve abreast of rising user expectations.

To add to these difficulties, system programmers, those who will write the system, and applications programmers, those who will use it, function in different worlds, use different jargon, and have disparate goals and philosophy. (It might be ventured that the two types of programmers have fundamentally dissimilar psychologies.) Neither group has a more correct view of the world, they just do not share the same one and this leads to tensions between them. Of course the people who will use the finished product seldom have any idea of the complexity of an operating system and have unrealizable expectations of what a real system will support. The operating system authors tend to encourage this by promising what they cannot deliver. The difficulty of the task, the inability to effectively communicate between implementor and specifier, the different expectations of the two groups all point towards failure. These problems have been recognized for years, yet there remains the impression that

spite of them all, things should be better.

The burden here falls squarely on the backs of systems programmers. In short, they have failed to produce even marginally acceptable systems for reasonable cost and in a reasonable time. The list of failures, in whole or in part, is long. A few are TSS/360, OS/360, SCOPE, EXEC8, TOPS-10, MULTICS. Some of these systems have survived: Multics has developed into the best time-sharing system available. But even the eventual successes were not completed on time. So operating systems programming does not have a fine reputation, but through these failures, much knowledge has been gained which can used to ease the task of creating new systems today. The problems that caused operating system to be created remain with us today and will have to be solved over and over again in the foreseeable future as new computer architectures develop. Thus, systems design must continue, and it can succeed in the right environment.

The Architecture Machine Group of the MIT Department of Architecture has grown into a lab encompassing fifty people--professors, research staff, graduate and undergraduate students. The business of the lab today most broadly considered is man-machine interaction: Interaction in ways that are far beyond the present day computer terminal-computer programmer interface, reaching to the day when computers will become tools for non-computer trained (even non-technical) personnel. The current

approach to the problem is reflected in the lab's use of color raster graphics, touch sensitive displays, optical video disks, and an interface that has been termed "Captain Kirk's Chair." [TREK] Thus any operating system used here must be able to support a unique and changing set of peripherals, primarily in real-time situations. The nature of the problems attacked by the lab has led to the development of home grown software systems and hardware devices.

The lab began ten years ago using 16-bit minicomputers produced by the Interdata Corporation. The machines had a 64K byte address space and an instruction set derived from the IBM System/360 series architecture. The lab decided from the outset that the vendor supplied operating system was not sufficient and proceeded to write their own. The system, called M.A.G.I.C. (Mockapetris And Gregory Interactive Computer, named after the system's authors) was developed with the aid of the IBM 360/67 at the MIT Information Processing Center, by doing cross-assemblies and emulation there. Many of the ideas used in M.A.G.I.C. were borrowed from CP-67 and the vendor's operating system. M.A.G.I.C. underwent five major versions and was transported through five hardware changes (upgrading to new Interdata processors with upward compatible instruction sets) of processors and three disk systems. By late 1975 the system was running on Interdata 7/32 computers in 16-bit compatibility mode with Calcomp 2314 disk drives. The system clearly showed its age and origins in many

ways. The lab had also outgrown a 64K address space: in fact, many kludges were created to allow programs to use the extra address space of the 7/32. During this time the lab underwent a major change in programming languages. In the beginning, systems programming was almost exclusively in assembly language with application programming done in vendor supplied FORTRAN IV. By 1976, the lab had an in-house developed PL/1 subset compiler which was used for most applications programs and some systems programs. [SAS] By early 1976 the time had come for a new operating system to be developed which took advantage of the 20-bit address space of the 7/32 and its 32-bit instructions.

In 1976 a project was begun to write a new operating system for the Interdata 7/32 to be written as much as possible in PL/1. It began with one novice systems programmer working on an assembler and one reworking the PL/1 compiler for the expanded instruction set. The project gradually grew until there were four undergraduates writing the system and one full time research staff member maintaining the compiler and coordinating the project. Three years after it was begun MagicSix became the standard operating system in the lab and, though all the results are not in, the system is a success. The lab now has more people simultaneously at work, producing more programs faster. It is estimated that from 7 to 10 man years were spend in total to produce the operating system, assemblers, a text editor, a lisp interpreter, a 32 bit version of the PL/1 compiler and everything

necessary for the use and maintenance of the system. Except for the PL/1 compiler, everything was written from scratch. The lesson of MagicSix is that successful operating systems can be written at a reasonable cost and in limited time.

2. The History of Operating Systems

An operating system provides a model computational world in which a programmer writes his program. The model consists of an environment and the specific objects of the world such as pieces of the operating system, different people's programs, etc. In order to understand the MagicSix environment a little must be understood of the development of operating systems and of the computer architectures that supported them. The traditional computer, the "von Neumann" machine is the computer we are all familiar with today and knowledge of it is presumed. The discussion below presumes a basic hardware system consisting of at least one processing unit-- the device that analyzes computer instructions in a program and performs the specified action (e.g. fixed or floating point arithmetic, or branching)-- and a hierarchical memory system. This memory system consists of a main memory, which is the only place the processor can look to find instruction or data, and a secondary store which can be used to hold programs not currently running. The subtle effects of the environment of an operating system extend to the way every program using that system is written, thus an operating system must be understood as a philosophy under which programs are written. The following will quickly trace the history and philosophical development of operating systems that led to MagicSix.

2.1 History of Computer Operating Systems

The first computers in the modern sense of the word were little more than glorified digital adding machines with one major difference: the ability to store programs in memory rather than as wires. With that change came the ability to quickly write and modify an algorithm, thus opening up the computer to new worlds of possibilities. In the beginning, programmers had to work in the lowest level machine code. [PSL:History]

Soon, the first "system programs" were developed. These were the assemblers and were followed shortly by linkers and loaders. Assemblers allowed the programmer to symbolically specify machine operation codes and to define labels for reference in branch instructions. Assemblers were an obvious and great advance that for the first time brought programming to the non-expert. They were systems programs in that everyone used them, they were treated as general utilities, and they became integral to the environment the programmer saw. With the advent of assemblers came the ability to share programs that others had written.

Without exception, the results of a program had to be output, usually on some sort of printer, and often there was also input to be read. Every input/output device required some method of control by the program, from simply requesting a card be read or a line printed, to intimate control of device internals. In addition, there often had to be conversions from one type of data

representation to another-- binary to BCD, floating point to BCD, etc. Naturally if someone else had gone to the trouble of writing some program to do I/O, or some data conversion, others wanted to use it instead of writing their own. This led to the development of such simple ideas as appending some special deck which contained common shared routines at the end of a user's program and assembling them together. These special decks became the first subroutine libraries.

Of course, there were many problems with this naive solution. First of all, how did a user make sure that the labels in his assembly program didn't conflict with those of a subroutine he used? What did the user do if he used only parts "a" and "c" of the library-- did he still have to waste time and memory by using the whole library? Despite the problems the standard routines were often better than writing your own from scratch. These subroutine libraries became systems software, common programs that were used by many others for differing applications. Their proper operation was relied on by many programmers.

Next came the linkage editors which took various separately assembled programs and put them together so that they could call each other almost as if they had been assembled together. Subroutine libraries now were combined with the user program after assembly, greatly reducing the problem of name conflicts. Using linkers to combine program modules was the second step in

the development of program sharing. The use of program libraries became easier and more widespread.

Higher level languages such as FORTRAN were developed. These generally depended on linkage editors to combine modules like user's FORTRAN programs and the FORTRAN runtime library of I/O drivers and mathematical subroutines. Machines rapidly grew bigger, more expensive and more powerful. The subroutine libraries became standard components associated with the computer. As they grew larger they became highly shared resources. Bureaucracies grew up to administer the computer and charge for its use. Programs were developed to account for the amount of computer time a program used and to calculate charging. It became necessary and desirable to find a better way to run the machine than turning the whole computer over to a programmer who might not be all that adept at handling the machinery. The programmer who walked up to the card reader, placed his deck in the hopper and pressed magic buttons disappeared. In short, most computer centers found it more desirable to take another step away from the bare machine.

In the very beginning, the interface between the system level and the user level was the machine's hardware. When assemblers became available, the border moved to encompass the assembler into the system. Subroutine libraries, linkers, accounting programs began to be looked upon as fitting to together to create a system.

Programmers began not to submit programs to the computer but "jobs" which compiled, linked, and ran a program providing both data and accounting information. Programs to sequence jobs by reading them into the machine, processing the account information and reading the next job were written. Pieces of these programs and of the subroutine libraries became permanently resident in the computer's main memory. The idea of the "operating system" was born. An operating system was an integrated set of programs that made a computer more civilized and easy to use. It joined common subroutines and elementary routines to control computer operations together in a single system. It must be stressed that an operating system is more than the sum of its component parts. To become a system, the various pieces must be made consistent with and coherent to one another-- they must be fused so that the programmer views the result as a system rather than many individual fragments. [PSL:SYSTEMS]

During this period computers became faster than the I/O devices they were connected to and it was soon realized that more benefit could be gained by multi-programming the processing unit. Multi-programming is a technique whereby many programs could be running at the same time, sharing the system's main memory. The processor runs one program for a certain period of time, then switches to another. This allows the computer to take advantage of the I/O latency time. The computer was still run in a "batch" mode, where the programmer submitted the program on some storage

media such as Hollerith cards or paper tape to the computer operator and then waited for his printed output. It took from minutes to days for a job to finally get run and the output delivered. One early example of a complete batch system--subroutine libraries, scheduling, accounting and job control--was the IBM IBSYS/IBJOB system. [PSL:History] [PSL:D825]

The story of operating systems splits here along two distinct lines. The batch systems continued to grow in size and complexity but not in new fundamental ideas. As programmers became more removed from the machine what they gained in programmability they lost in interaction. They not only became detached from the machine, but also from their programs. In the beginning when a program had an error the programmer was right there, ready to stop the processor and probe into the problem. [IBM:ASD] Whether this was by using the computer's control panel or some more sophisticated device (such as a computer terminal) there was immediate interaction. The programmer, if he found his problem, could go so far as to patch the running program and then restart it. With the advent of batch systems the programmer had to settle with a post-mortem core dump and at worst he got nothing but wrong answers. The economics of running a large, fast computer and the bureaucracy that grew up around them made it impossible to retain the old "hands on" approach. The need to interact with a program, coupled with the desire to have more sophisticated ways of preparing a program than a key-punch

machine, gave rise to "time-sharing systems."

Time-sharing was a new way to divide up a computer among many people by giving each one the appearance of having the machine to himself. [PSL:MULTICS] [PSL:EXPERIMENT] Users sitting at computer terminal devices could submit programs to be run and debug them interactively with the advantages of special debugging programs. This was achieved by techniques of multiprogramming similar to those used in batch systems but the impetus was on program development. Systems were developed which allowed the programmer to keep his programs on tape or disk and to edit them from his computer terminal. The editing programs could be much more sophisticated than a mechanical card punch and more flexible in program format. Debugging tools similarly benefited from the time-sharing environment; they could be highly interactive while providing more features than a computer's control console.

2.2 The Arguments For and Against Time-sharing

Arguments for and against time-sharing have raged for years. It is true that for merely running the same debugged programs over and over batch processing would suffice-- that is, if these programs relied on no interaction. Time-sharing opened up new areas of computer applications from more sophisticated program development tools to "intelligent typewriters." As such, time-sharing was clearly a leap beyond batch systems. The most

ambitious time-sharing systems provide both the interactive environment and the ability to run background batch jobs. Batch systems continued to be used and developed but the notion of time-sharing grew in importance until today, when its influence can be seen grafted onto traditional batch-oriented systems. (Witness CDC's KRONOS and NOS time-sharing option, IBM's TSO, MVS, and VSPC.) Time-sharing, it is true did not live up to its initial expectations. For many years it did not become the clear choice over batch because it was accused of being more complex and of wasting hardware resources. However these are not its true faults.

While initial time-sharing systems were indeed more complex than batch systems, this situation did not continue for long. Batch systems developed for third generation computers were as large and complex as time-sharing systems, often more so. Time-sharing was in these days on a steep learning curve and each system tried to go great leaps beyond the previous systems. Under such conditions it is no wonder that there was ample ammunition to use against time-sharing. But relatively simple, unsophisticated systems were developed cheaply and quickly. The PDP 1 time sharing system was written even as batch computing was emerging and the MIT ITS system emerged as OS/360 was just being released. Neither of these system was overly complex. [ITS]

It has been charged that time-sharing is too expensive. The real

question, however, is too expensive for what. Time-sharing makes possible applications and complex systems that are impossible in a batch environment. The current need to develop programs as fast as possible, which arises from considerations of labor cost and need, can only be satisfied by some means of interactive computing. That time-sharing may slow down background batch jobs is not due to the expense of the time-sharing system but the lack of necessary computer resources to support both. Time-sharing systems are not appreciably larger than batch systems and any additional operating overhead is directly due the need to interact with many users.

Perhaps the most damaging charge against time-sharing is that it is very slow and doesn't really provide the needed interaction fast enough. This can be analyzed in two ways. First, time-sharing provides a level of service not possible at all in a batch system where there is no possible conception of interaction. Second, the speed of a time-sharing system is highly dependent on the number of people being served by it. Despite all efforts to minimize this effect, in the end a computer will support just so much use. The more advanced the level of service, the more it must cost in machine terms, but the less in human terms. It can be said that if a system is too slow then there are just too many users and you should buy more computer: this was always true of batch systems.

2.3 Beyond Time-sharing

The story of operating system does not end with time-sharing. Today there is a movement back to one user per computer. This development is being driven by three separate forces, two of them human factors and one technological. These so-called "personal computers" bear little resemblance to the bare machines of yesteryear. It is fully presumed that to support these machines will take as much systems software as time-sharing, perhaps more.

Time-sharing was the first attempt to return to interactive computing. The emphasis was on fast response and reasonably supportive environment within the confines of sharing a single processor many ways. To improve the "quality" of computing, to make computers available to a wider group of people with less fuss and bother, to make more computing available and have it cover more areas-- these are also the aims of personal computing. At first it was the human operator who had to be flexible to use the machine: he had to learn baroque languages, strange conventions, and mystical incantations to get the machine to solve his problem. Later he had to learn less to run his program and much more to debug it. Time-sharing attempted to bridge the gap between computer and programmer-- to bring the machine more intimately in contact with the user. Finally now, personal computers are attempting to discover better ways to make computers interface to humans with little concern for how much

that will take in terms of fancy technology. In the long run, the decision to usea computer will be determined by human factors, thus today's efforts are directed towards making it as painless as possible.

In our society, people like to have a sense of ownership, possession and the ability to customize their property. Computers have never seemed suitable to individual ownership until recently. Personal computers move in the direction of giving one machine to one person for that person to control. This removes an entire layer of bureaucracy that previous computer users have had to go through to get access to the machine. There is no longer any need to deal with computer operators or dial the phone to see if the machine is up again. The computer becomes like the telephone or the typewriter, just another appliance, albeit a complicated one.

All of this is possible because of the emerging computer hardware. The cost of the electronic parts of computer systems has consistently fallen. The development of large scale integrated circuitry has brought the cost of some CPUs below the price of a suitable power supply to make them work. Processing elements, memories and device controller prices will continue their rapid fall at least into the late 1980's making it possible to get more computer for less, enough more so that a large percentage of the machine's resources can be spent on supporting

a highly user-oriented, or personalized system.

The trend from batch to time-sharing to personal computers shows a growing emphasis on enlarging the domain of problems that can be solved with the aid of computers, by making computers more usable through human engineering.

2.4 Some Major Hardware Developments

The development of operating systems has led to the modification computer architectures. Most of the changes have not been in the central processing unit, which has retained the basic von Neumann design, but in computer memory systems. These changes were initially needed to protect one user's programs from another's in a multi-programming environment. [PSL:ATLAS]

In a multi-programming environment the operating system and several user programs coexist in the computer's main memory simultaneuosly. The operating system remains in overall control and can, in general, preempt a user program currently running in order to perform supervisory tasks or switch to another user program. Some of the earliest systems did not share memory between tasks; instead one user's program was rolled into memory, run for a time and then was rolled out to make room for someone else. As main memory sizes increased it became possible for there to be more than one program extant in the memory and for multi-programming to take advantage of this. There could also be

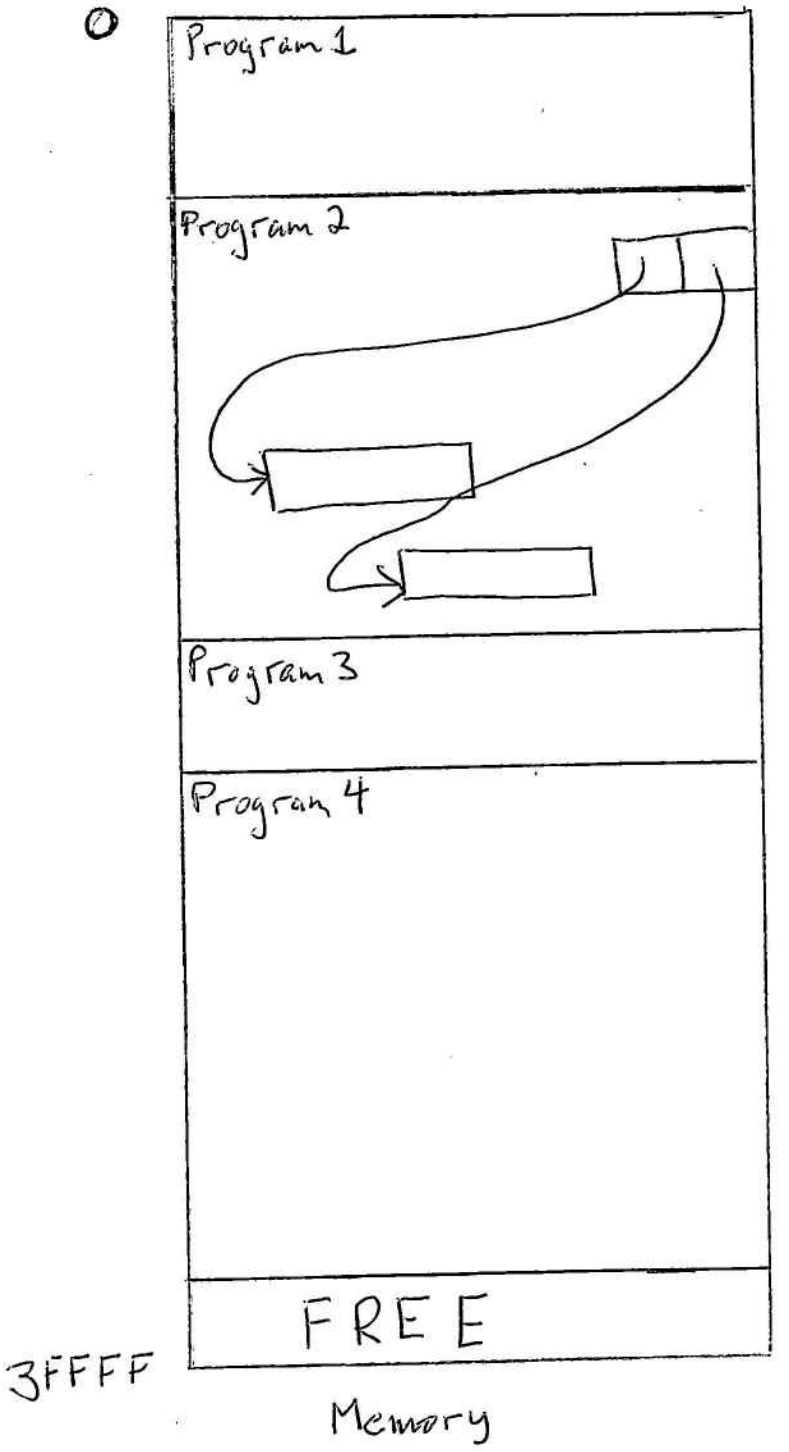
storage allocated for a permanent operating system kernel which provided scheduling and I/O control functions. It was unfortunately possible for one program, either accidently or with malicious intent, to destroy another, including the operating system.

Some vocabulary is necessary to understand the development of memory mapping techniques. The "name space" of an object is the totality of entities to which the object can refer. In our context only entities external to the object (in the sense of PL/1 external) are considered part of its name space. For example, a program's name space is all the external subroutines and external data items it uses. The name space of a processor includes its registers and its address space. In a traditional system, when a program is link-edited, the character string name of an external subroutine is converted to the address of the subroutine in the core load being created. In a dynamic linking system the conversion from the character string name of a subroutine to an address occurs the first time the external subroutine is called.

When a program is loaded into memory the names by which it refers to a part of itself or to external data and subroutines must be turned into names which will be recognized by the processor. While the representation of names has many forms before a program is loaded, depending on the particulars of the computer and the

operating system, after loading the standard form for names are memory addresses. The set of names used by a program before it is loaded is termed the "program name space", while the names used in the computer form the "machine name space." Traditionally, a program was assigned a fixed place in memory for the duration of its run and once thus resolved, its machine name space of addresses was static. In a multi-programming environment this led to several difficulties. Say there are four programs, numbered one through four and loaded into memory in that order at the same time, and further that they are all of different sizes. If programs two and four terminate first, their storage would become free for some new program to be loaded; however since one and three are fixed in memory, no programs larger than either two or four could be started. Now, it might be possible to move program three but this task would involve remapping the name space of the program, which is currently the same as physical memory addresses. This process would involve searching through the program for each name (a memory address) and changing it. (It might also be impossible if some external agency such as an I/O handler knew where program three was.) (fig. 1) If a new mapping level were inserted between the machine name space and the physical location in memory, it would be possible to avoid having to do any more than moving program three in physical memory and changing the mapping. At the same time, this map could specify which addresses were legal, i.e. belonged to a

Fig. 1.



program, and made all other addresses illegal to use. Thus, with one mechanism multi-programming could be made more efficient and the protection of one program from another was solved. This method is traditionally called "base and bounds," whereby two hardware registers are used to map the machine name space to the physical memory. (fig. 2) [PSL:SEGMENTS]

This memory mapping technique can be further modified to avoid the necessity of shuffling programs in physical memory. Let the physical memory be divided in partitions of equal size, called pages and let the machine space be similarly divided. For each user instead of a base and bounds register, there is an array which specifies a mapping from a machine page to a physical memory page. (fig. 3) There is no longer any requirement to copy programs between places in physical memory, in fact, there is no need for a page in machine space to have a physical memory page associated with it except when it is actually be used by the processor. This is called a "virtual memory system." There are various different paging algorithms for determining when and what pages are to moved from main memory to secondary memory and vice-versa.

To understand the memory management of MagicSix one final refinement is necessary which remains controversial even today, fifteen years after it was developed-- that is segmentation. Segmentation facilitates the controlled sharing without copying

Fig. 2.

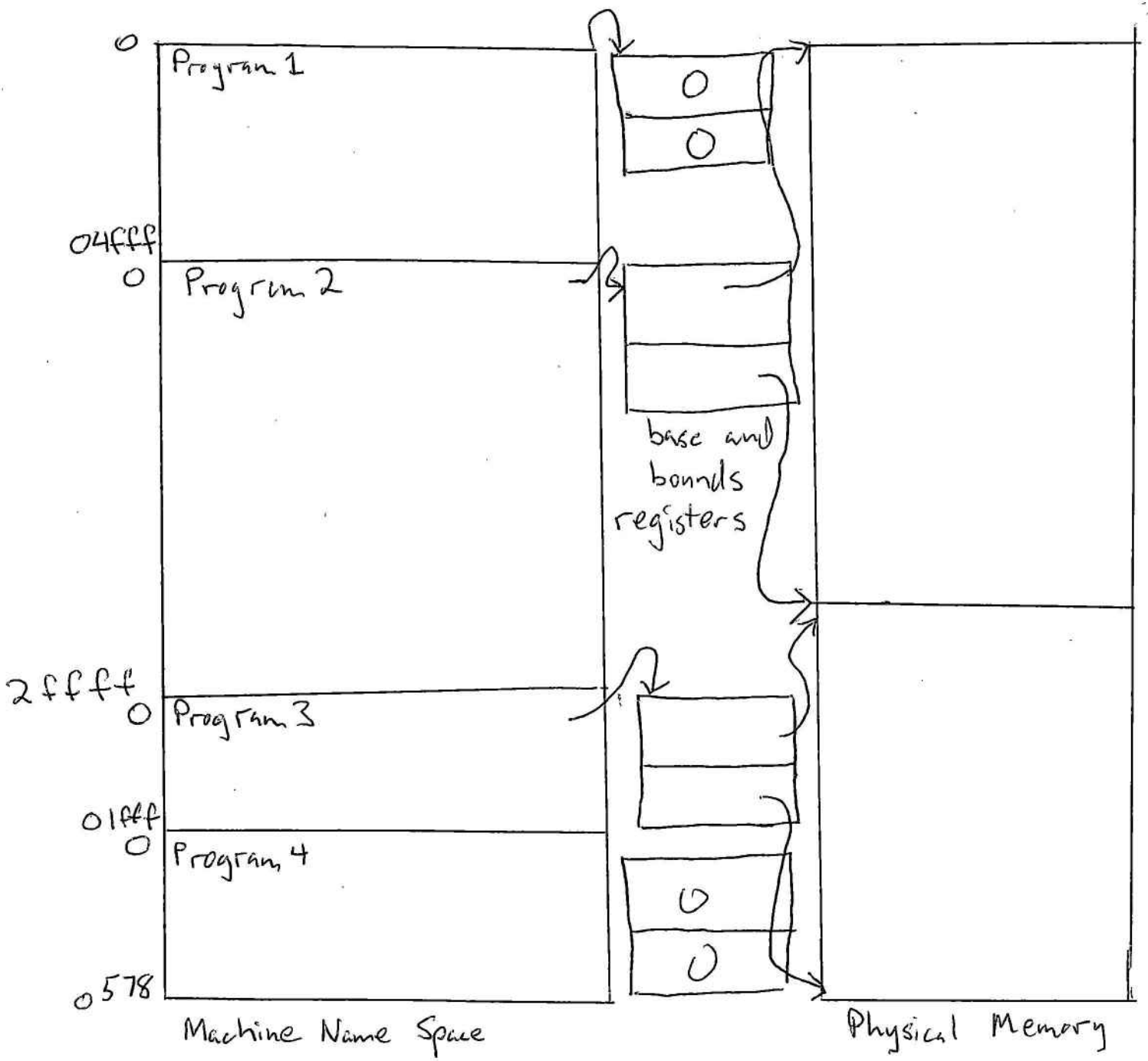
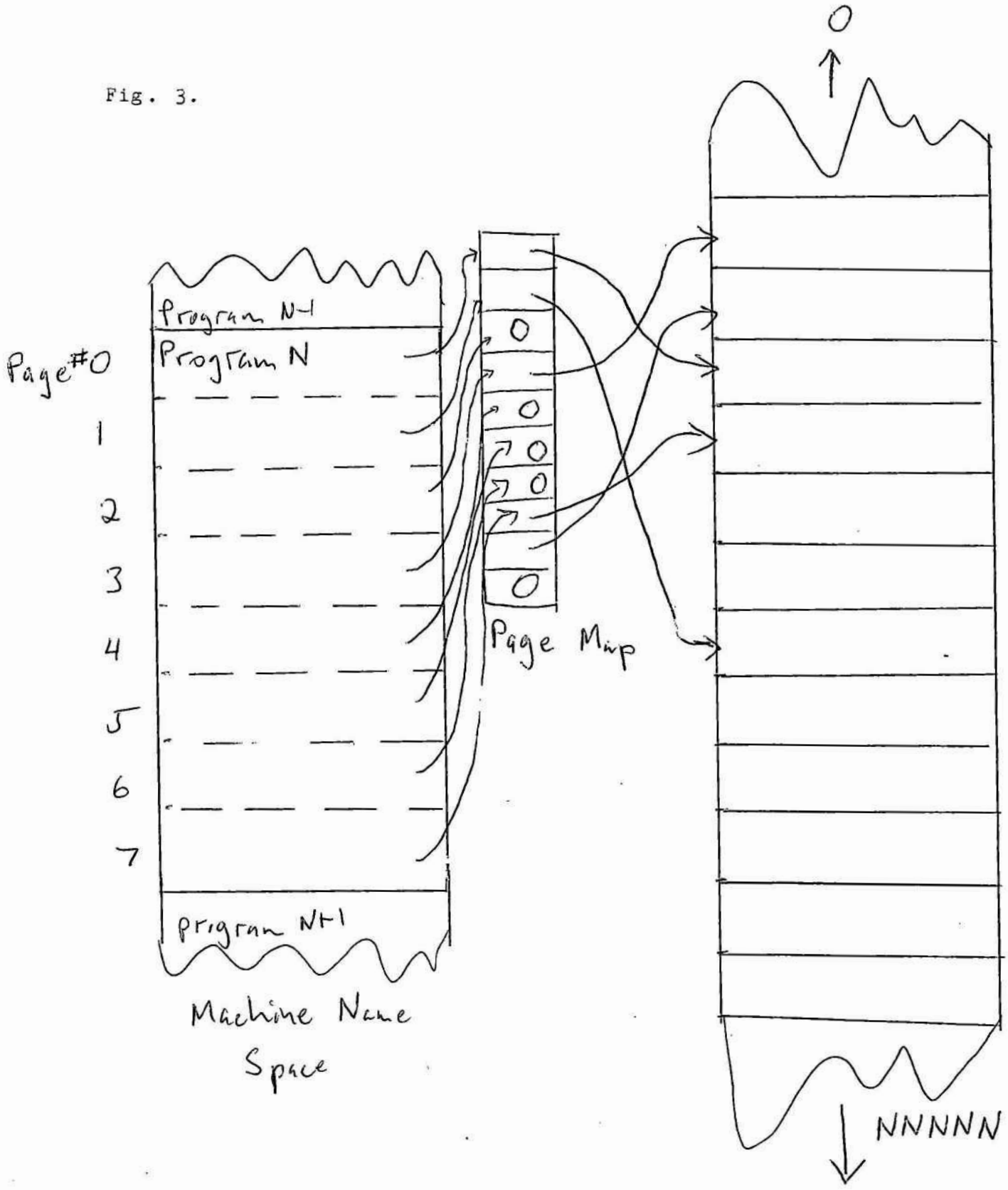
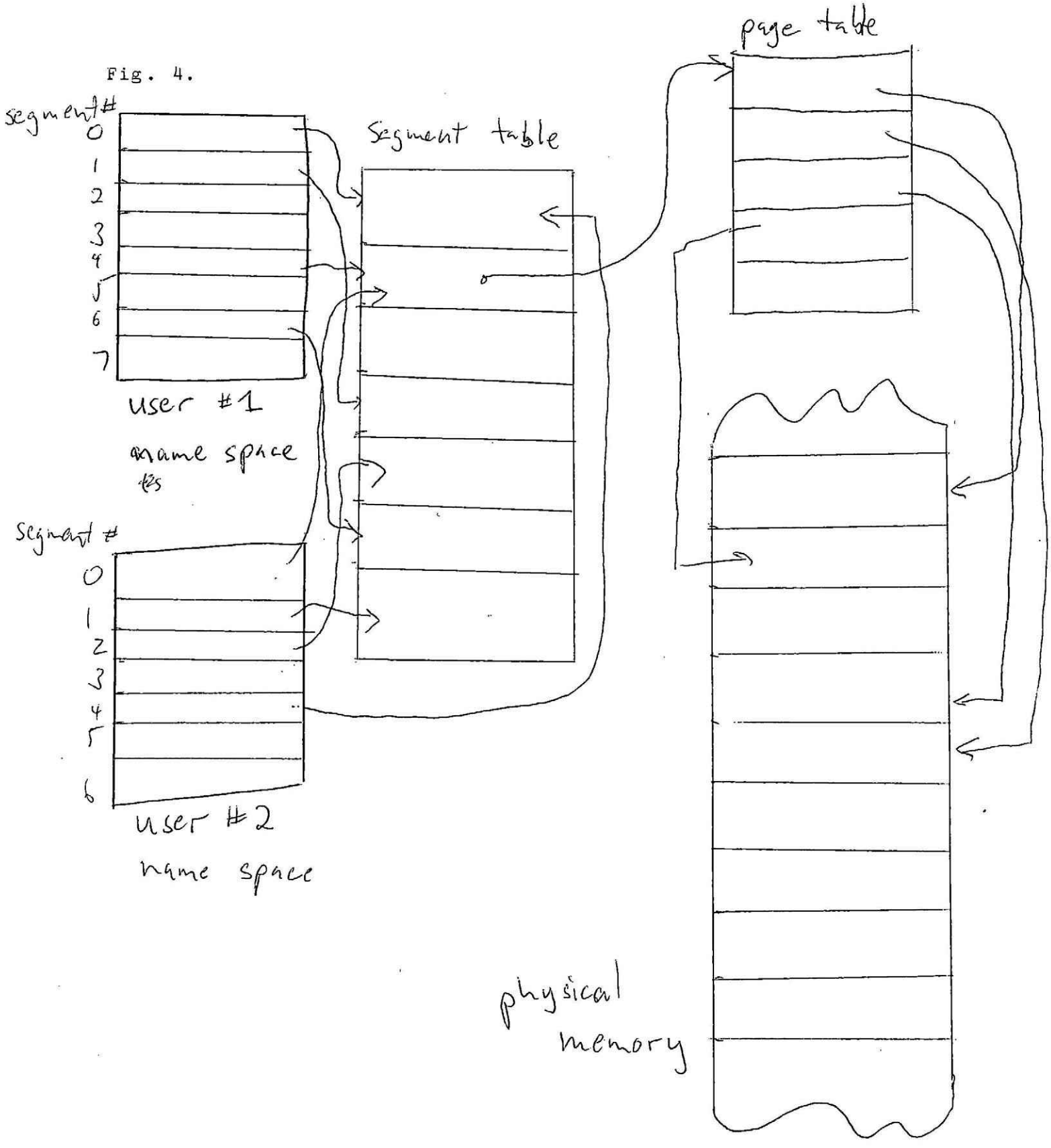

0
Program 1
04fff
0
Program 2
base and
bounds
registers
2ffff
0
Program 3
01fff
0
Program 4
0578
Machine Name Space
Physical Memory

Fig. 3.


0
Program N-1
Program N
Page#0
1
2
3
4
5
6
7
Program N+1
Machine Name Space
Page Map
NNNNN

between different users of common subroutines, and provides for letting objects, such as data spaces and stacks, grow. The machine space is reformulated to consist not of a traditional single linear address space, but of many segments each of which is a linear address space. Before segmentation, a name in machine space was a single component, the address; in segmentation a name has two components, a segment name and an offset into that segment's linear address space. The form of the segment name in machine space is an integer whose range determines the maximum number of objects that can be simultaneously addressed. Names in program space, which are object names, also consist of two pieces; when a program is loaded these are converted (mapped) to the machine space format by determining which segment contains the desired object. A machine address can be converted to the physical memory address of an object by keeping an array of base and bounds registers indexed by segment number. If paging is added, each base and bounds register in the array is replaced by a pointer to a page table in physical memory just as the single base and bounds register was above. (fig. 4) The physical memory is filled with objects, and two programs referring to the same subroutine will refer to the same object in memory. The term segment is often used confusingly to mean both the object as it exists in machine space and in some external world, such as a file system. Initiating a segment is the process whereby an external object is associated with a segment in machine space. A

Fig. 4.


page table
segment #
0
1
2
3
4
5
6
7
Segment table
user #1
name space
Segment #
0
1
2
3
4
5
6
user #2
name space
physical
memory

segment can grow and shrink as the object which it represents does, thus making it possible to replace the traditional conception of a data file accessed with I/O operations with that of a segment accessed directly as if it were always in memory. (e.g. as program variables) This allows the programmer to view his program and other programs which he may call as subroutines, and his data areas all as subtypes of a common abstraction, the segment. [PSL:SEGMENTS]

3. The Philosophy and Goals of MagicSix

3.1 Philosophy and Goals

The basic goal of MagicSix was to create an environment where programmers could quickly and simply develop everything from small, trivial programs to large, complex, inter-related computer systems. It was felt that large systems would be developed and debugged in pieces by different programmers perhaps using different languages, and that the environment should support this incremental development. Furthermore, many of the systems would be extensible in nature and some means of adding new features without re-compiling and re-linking was felt necessary. Because most of the programs of the lab were written in PL/1 a good environment for the execution of these programs was essential. Here good environment implies support of recursive procedures, the signaling mechanism and other language features. The system would have to support multi-programming and for reasons of economics time-sharing rather than personal computing. The system should also strive for maximum flexibility to accommodate unforeseen tasks that it might be called on to perform. This would preferably include the ability to add new system modules without having to significantly alter the system. As a final requirement from a program development point of view, the operating system should support the sharing of procedures and data files among users. The traditional modes of sharing were

considered limited because actual copies of the shared procedures had to be made, thus multiplying storage requirements and making it difficult to distribute new versions. MagicSix sought not to require copying shared programs.

From an implementation point of view, it was taken for granted that most of the system would be written in a higher level language, in this case PL/1. This would make the system more readable and more maintainable, and more importantly it would greatly speed development. Some desirable system features would have been just too difficult to write in assembly language had we chosen that path. It was hoped that the system would be made up of relatively self-contained modules which could for the most part be written independently, allowing different people to code each module. Ideally, the system would also lend itself to easy extension so that new features could be added without severe disruptions to existing forms. A uniform calling sequence for PL/1 and assembler language was needed to accomplish the design goals mentioned above, as well as to allow the system to freely call between its two implementation languages. This would also permit system procedures to be rewritten in different languages for reasons of efficiency or readability.

At the most practical level, the system had to be reasonably efficient to be able to support the desired load. We also had to realize that the system was being implemented on a mini-computer

over whose architecture we had no control. The system was also going to be implemented by undergraduate students working part time during school and full time during summers and long vacations. The lab could not wait forever for the system and an undergraduate could expect to work no more than three years on its development.

3.2 Role Models for MagicSix

All of the systems programmers who worked on MagicSix development were heavily influenced by two computer systems on the MIT campus, the Multics systems developed originally by MIT, General Electric (later Honeywell Information Systems), and Bell Labs, and the ITS time-sharing system developed on the MIT campus by members of the MIT Artificial Intelligence Laboratory. The experience of these two systems helped to form a firm if not verbalized philosophy for the system environment, methods of implementation, and user interface of MagicSix. [MagicSix]

Multics is the foremost time-sharing system available today from a major computer manufacturer. It provides the prime example of system designers attempting to frame a consistent, extensible, and flexible user-to-system interface. MagicSix borrows much of the overall Multics system design while differing in implementation details. In particular Multics provided the model of a process with a segmented address space containing the

operating system. The goals of MagicSix were derived by beginning with Multics concepts subject to the limitations of a mini-computer and a desire to orient around high speed display terminals.

The principal weakness of Multics was in the area of terminal support, which was modeled after 13 characters per second (CPS) half-duplex printing terminals (IBM Model 2741 and 1050 terminals). MagicSix was to be designed around display terminals which would never operate below 120 characters per second with the majority at 960 CPS. The ITS system provided a ready example of good display terminal support and interactive software oriented around fast terminals. The debt to these two systems, especially Multics, is obvious in almost every aspect of MagicSix.

3.3 The MagicSix Environment

3.3.1 The Basic Environment

The fundamental idea of the Multics and MagicSix environments is a "process," consisting of a name space of segments referring to particular objects-- procedures and data-- from a global world of object and an execution state which includes the name of the procedure objects currently executing and the associated machine state (accumulators, index and pointer registers, point of execution within the procedure, etc). The process name space

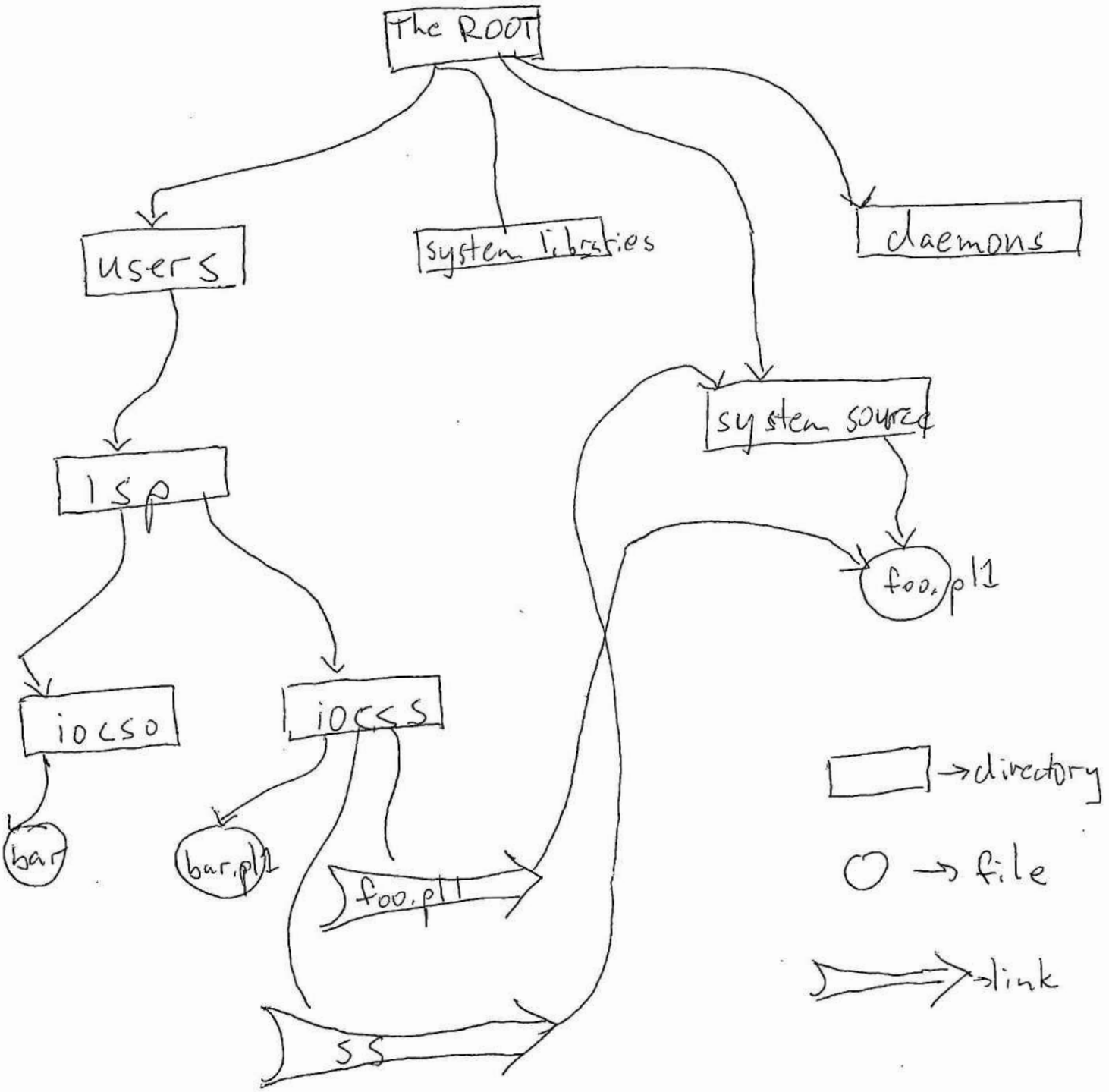
consists of segments which are mapped to objects from the file hierarchy of the system. (see Chapter 2) An object is "made known" in a process by associating it with a segment in the process name space and by specifying a reference name (or names) that can be used to refer to the segment. Thus an initiated object has both its segment number and its reference names. Reference names are primarily used in the dynamic linking process. Procedure objects are with few exceptions read-only, re-entrant and recursive. A procedure object is not quite the same as the traditional program, which is generally not designed to be called by another program but is to be "run," produce output, and go away. Procedures bear more resemblances to subroutines which are just executed and then return. For example, when a process is created, a process overseer is called in the new process which initializes certain per-process databases and then calls the command listener procedure. The command listener reads and parses a command line and calls the desired command procedure as a subroutine with arguments. When an error, such as fixed point overflow occurs, a PL/1 condition is signalled in the process. If the user has supplied no handler for that condition, the standard default handler prints an error message and invokes the command listener recursively. This allows for multiple levels of the command processing loop. Corrective action can be taken and the procedure restarted.

The execution environment is designed to be hospitable for PL/1

programs-- saved registers and automatic variables are allocated on a stack segment, static variables in a common static segment. Extensive use is made of the PL/1 condition mechanism both by the system itself and in system-user communication. Since different procedures are generally different objects it is neither necessary nor desirable to linkage-edit a procedure together with all of its subroutines. References from one procedure to another are resolved at run time when the actual reference occurs. This linkage technique is called "dynamic linking."

The MagicSix file system is tree-structured and consists of three types of branches: files, directories, and links. (fig. 5) Directories are merely segments with a specific structure that is known about by the system. The top node in the file system is a directory called the root. A directory contains a list of branches contained at this level in the tree and, excepting links, their index in the volume table of contents (VTOC) used for actually locating the branch. Files are often referred to as segments because of their close relationship to the segmented address space. Links are ways to refer to other locations in the tree, either directories or segments. The complete specification of a branch is called its absolute pathname and consists of the names of all the directory nodes between the root and the name of the branch itself separated by ">" characters. For example: ">users>lsp>foo.pl1" is the branch named "foo.pl1" in the directory "lsp" which is a directory contained in "users" which

Fig. 5.


The ROOT
users
system libraries
daemons
system source
lsp
foo.pl1
iocso
iocss
bar
bar.pl1
foo.pl1
ss
→ directory
→ file
→ link

is a directory contained in the root. A process may also have a "working directory" which is the name of a directory whose branches are currently of prime interest. Branches may be named relative to the working directory by a "relative pathanme". Relative pathnames do not begin with a ">" to distinguish then from absolute pathnames. (e.g. "print >u>lsp>iocss>foo.pl1" would print the segment foo.pl1, specified by an absolute pathame. "print foo.pl1" would print the same segment specified by relative pathame provided the working directory was ">u>lsp>iocss".) An absolute pathname can be formed from a relative pathname by a simple algorithm: Starting with the working directory, go up one level in the tree for each "<" at the beginning of a relative pathname, then starting from here in the tree find the branch named in the remaining pathname. File and directory branches also contain the "access control list" or ACL which specifies what the access to the file will be when it is initiated. The modes of access that can be specified are read-only, read-write, read-execute and read-execute-write. Currently access is global with no attempt to provide different access for different users. Because of the close nature of the Architecture Machine Group, personal access was thought an unnecessary mis-feature which would be possible to add later if desired. Global access was included to support the system and insure that procedures were read-execute only.

The segments in a process are divided into two classes, called

ring 0 and ring 1 which reflect a protection hierarchy. The ring number of a segment is the ring number of its associated branch in the file system. The access mode and ring bracket, both taken from the branch, are combined to form the "effective access" which determines the access to the segment by objects in the process based on their ring number. In MagicSix effective access is always read-execute-write for ring 0 and the access term from the branch for ring 1. (The development of access is more complicated in Multics.) An inward call from ring 1 to ring 0 is allowed only if the gate bit is set for the segment. Ring 0 segments are part of the protected operating system while ring 1 segments are user segments or non-protected pieces of the operating system. By this means the system can safely be embedded within a process's address space and still be protected from non-system programs (and non-protected parts of the system). Ring 1 is generally termed the user ring and ring 0, the system ring. [MPM Reference]

Most system programs are inner ring (ring 0) segments and therefore have more access privileges than user programs. Programs of both rings exist in a process in the same name space; thus system routines are no different from any other procedures except in their access privileges. The system is part of the process name space and consists primarily of ring 0 segments. Calls between procedures of the same ring are handled as in more traditional systems while inter-ring calls involve additional

overhead. A call from ring 1 into ring 0 is only allowed to occur at designated entries called gates in the ring 0 segment. At the time of the call, the arguments of the caller have their access validated, so it is impossible for a ring 1 program to destroy a segment it does not have access to by fooling a ring 0 program. To ensure protection, outward (ring 0 to ring 1) calls are not permitted; however, ring 0 procedures can return to their ring 1 callers. Since both rings share the same name space they can freely pass pointers to segments back and forth (subject to access checking). The ring mechanism together with segmentation and dynamic linking allow the MagicSix system to be completely extensible-- a new procedure becomes a protected entity in the system by being made a ring 0 segment. MagicSix, due to hardware limitations, can only partially ensure that proper access is maintained in cross ring calls, while Multics can support the ring mechanism in its entirety. While because of hardware, MagicSix cannot be secured from malicious users, there have been few occasions of a ring 0 procedure damaging a segment because of being passed a bad argument from ring 1. Almost all such occurences are trapped by the gate keeper, which is a system module that checks access when gating into ring 0.

The I/O interface of the system consists of three pieces, device allocation, device control, and I/O streams. Device allocation is a system module which grants and relinquishes ownership of a device by a process. Processes are only allowed to do I/O on

devices which they currently own. Device control modules are system programs which have control over an I/O device and perform buffering and some data translation. These are the only procedures which contain actual I/O instructions. I/O streams provide an abstraction of the input/output methods of traditional systems and encompasses general process communication. Indeed, some I/O streams do not deal with real I/O devices at all but rather provide for interprocess communication or traditional I/O access methods for data segments. The stream concept attempts to remove concerns about the vagaries of specific I/O devices and concentrate attention on the idea of moving data from one place to another.

A stream is a set of associated procedures-- called an "IOSIM" or I/O System Interface Module-- which implements two stream functions, attaching and detaching, and four types of I/O operations, input, output, order, and mode. Attaching a stream performs all the initialization necessary for this IOSIM including requesting ownership of I/O devices. Detaching performs cleanup operations for the IOSIM. Input and output operations allow input and output of data in multiples of bytes. Order operations provide information about the state of an I/O stream and operations to change that state. The specific order operations are a function of the particular type of IOSIM. For example the "tty_io" IOSIM supports the order operation of "status" which returns how many bytes of input and output data

are currently buffered. The mode operation provides a way to discover and set specific device characteristics, such as terminal line length or tape drive density. [MPM Reference]

3.3.2 The System Model of the User

Every system has a model of its users; MagicSix is no exception. From the text editor to the "help" system, MagicSix presumes that it will be communicating with users on high speed display terminals of at least 24 lines by 80 columns. The help system, for example, does not pause every four lines to see if the user wants more information, which might be desirable if he had a slow terminal, and the text editor does not even work from printing terminals! Because of the high speed of output, the system can provide online documentation faster than a user could possibly find it in a manual. Consequently there is little about MagicSix that is designed for paper form.

MagicSix provides the ability for a user to customize his environment. Many programs can have their default actions affected by the "profile" the user sets up in his user information file. He may also provide his own substitute for all non-protected system procedures, including his command listener and command processor. His information file also contains such information as his office number, full name, home phone, etc. which can be accessed online by other users. Every user of the

system maintains some information in this file, even if it is only his full name. The system also provides mail and message systems to sent, edit and selectively read both mail and messages from other users. Messages are short notes send between concurrently logged in processes. Mail is exactly that, some text sent from one user to another's mailbox. Thus messages are analogous to telephones and mail to postal mail.

The text editor of MagicSix, called "TVmacs", is a user extensible real-time display oriented editor. It is modeled after the EMACS editor of ITS (There is now a similar EMACS on Multics) and is largely compatible with it. This editor is used for all document and program text editing. The usefulness of a real-time editor to all the users of a system cannot be underestimated; without it MagicSix would have taken years longer to develop. The extensibility allows users to create editor commands especially designed for specific tasks such as document editing or editing programs in specific programming languages. It also provides the ability for individuals to customize the environment to their own personal styles. [DLW] [OTA] [BSG] [RMS] [ECC]

MagicSix also tries to provide for naive users. The "help" system has a list of various names to key a single help file so that users do not need to know exactly what they are looking for. (There are currently some experiments trying to greatly expand on this concept using TVmacs and its implementation language

"SINE"). The error messages are not cryptic numbers to be looked up in an obscure manual but English language descriptions, often with the line number of the program where the error occurred. The PL/1 compiler has a debugging mode which allows a user to flow-trace his procedure as it executes and to set break points. [RK]

4. The MagicSix Development Effort

4.1. The MagicSix Hardware

The MagicSix system was designed to run on Interdata 7/32 32-bit minicomputers having at least 256K bytes of main memory and equipped with either hardware or firmware floating point. The average instruction rate is about 0.3 million instructions per second (MIPS). The machine is byte addressed and has a 20-bit address space which is divided into 16 segments of 64K bytes each. The system currently supports both 2314 and CDC storage module disks and presumes at least one disk drive. A 9-track industry standard tape drive is useful for system backup and program interchange but is not required. Asynchronous communication lines are used to connect terminals to the system and modem support is provided. The largest configuration in use at this writing has 512KB memory, a 300 MB CDC Storage Module Disk, an 800 BPI 9 track tape drive, 6 9600-baud hardwired display terminals and 2 1200/300-baud modems, a Summagraphics data tablet, an MCA VideoDisc interface, a VOTRAX voice synthesis unit, and a RAMTEK 9300 Color Raster Graphics System. That configuration can comfortably support four to five users.

The design of the 7/32 memory system has been the single largest hindrance to system development. The memory mapping hardware was not designed for virtual memory and supports only 16 64KB segments. Interdata apparently believed that the processor should

only be able to have as much address space as the maximum amount of physical memory. (Recently announced Interdata 32-bit processors continue this policy.) There are no inherent reasons why the processor could not have an address space of 32-bits. Sixteen segments are not enough to support the MagicSix object oriented environment. To overcome the lack of segments the system has the concept of an "address spaces" which is a named object consisting of a set of 16 segments. Six segments of every address space contain the same objects-- the system kernel, the PL/1 run-time operators, user ring system utilities, the ring 0 stack, the ring 1 stack, and the ring 1 linkage segment. [MLK] Processes have multiple address spaces which combine to determine the totality of known segments in the process. Procedures in one address space cannot directly address any segments of another space; however, procedure calls between address spaces are supported. Objects in the file system may have an address space name associated with them. When an object is initiated it can be placed in the address space the process is currently running in, the address space specified in the file system, or any other named address space, by giving options to the initiation procedure. The dynamic linker, when initiating a segment, always uses the file system address space if present. Otherwise it uses the process's current address space. When both caller and called procedure lie in the same address space (an "intra-address space call"), the link will be converted to a branch instruction

directly to the called procedure. If they lie in different address space (an "inter-address space call") the link is converted to a call to a system procedure called the "switcher." The switcher attempts to make sure that all the arguments of the call will be addressable in the new address space, including segments pointed to by PL/1 pointer arguments. If one of the arguments is a PL/1 structure which contains pointers, the switcher is unable to ensure the pointers will be valid in the new space. Thus, the system attempts to make inter-address space calls transparent to the programmer. It has been found that calls involving the switcher are ten times slower than normal calls; however, the frequency of these calls is much lower. With this modification MagicSix supports the object-oriented segmented system environment mentioned earlier, with no upper bound on the total number of initiated segments.

The other difficulty with the 7/32 memory system is memory mapping hardware which was not designed to support virtual memory. For example the standard Interdata mapper does not return the fault address in a procedure which takes a segment-swapped-out fault, making it difficult to determine which instruction to restart after the segment is swapped back in. (Note: The 7/32 has variable length instructions.) MagicSix attempts to back up the procedure address register by a combination of instruction simulation and instruction format knowledge. The standard Interdata memory system will also

complete an instruction that references swapped-out data, thereby destroying the general registers. This was solved by modifying the processor. [IDATA]

4.2 System Implementation

The bulk of MagicSix was implemented by four undergraduate students over a period of three years. Most major system modules were designed and originally coded by Michael Kazar, with various pieces written or redesigned by others later. Much of the original kernel of the system was coded initially in assembler language, as it was written before PL/1 was operational on the system. The PL/1 environment is not supported at interrupt level so all interrupt code is in assembly language. Major system modules have been written and sometimes rewritten without impacting other parts of the system. The system itself consists of the unswapped kernel, which resides in the low 64 KB of main memory, and all other ring 0 programs in the file system. In the following paragraphs each of the major system components in the kernel are briefly considered.

The system is supported by two kernel processes which manage memory and disk I/O requests. The memory manager or swapper is called the Reality Interface Module (RIM) and is responsible for all swapping of segments to and from disk as well as within main memory. The RIM receives messages from normal processes

requesting it to perform a memory management function, such as swapping in a segment. The RIM implements a demand swapping algorithm for segments similar to demand paging on other systems. When the RIM gets a segment swap-in message, it will find or make a slot in memory for it and issue a request to have it swapped in. If there is enough total empty memory for a segment, the RIM will shuffle all in-core segments to make a single area large enough to hold the requested segment. If there is not sufficient free memory the RIM will swap out the least recently used segments until there is. It is not possible to keep an accurate track of the least recently used segment but several approximating methods are used. The RIM also has the ability to freeze segments in specific areas of physical memory during direct memory access I/O transfers.

The disk server is a kernel process which handles requests to transfer segments to and from main memory. With few exceptions all disk server requests originate as messages from the RIM consisting of a VTOC address and physical memory address. (see section on file system below) The disk server then either writes the addressed segment back to disk, or reads it in from disk. The disk server tries to optimize the I/O transfers by doing multiple disk sector reads and writes but does not overlap seeks if more than one disk drive is present. In retrospect, it appears that the functions of the disk server could have been subsumed into the RIM with an attendant increase in efficiency.

The scheduler is invoked at every system clock interrupt (0.25 seconds) to determine which process should be run for the next clock period, or "quantum." It may also be invoked by processes voluntarily giving up the processor before the end of their quantum. The scheduler implements a round robin system modified by an optimization to allow CPU-bound jobs to run for multiple quanta. If no process is ready to run the scheduler puts the processor in the wait state until another interrupt of some kind occurs. The current scheduler is a very old system module which is in need of revision to use a better algorithm than the current round robin scheme.

The signaller is the heart of the PL/1 condition mechanism. Given a condition to signal, the signaller walks down the process's stack looking for the corresponding handler. If it traverses the entire stack without finding a handler, the system standard default handler is called, which outputs a message to the processes standard output stream. The signaller also knows about certain special handlers such as "anyother" which catch all conditions.

The ability for one procedure to call another is implemented by the dynamic linker. References by one procedure to another, called "links," are stored as a text string of the name of the procedure to call and a data block of a processor instruction to invoke the linker and an offset to the text string. When a

procedure is first executed, its static storage and every link's data block are copied to a segment called the combined linkage segment which contains the linkage sections of all segments. In a call the procedure uses the copy of the link that is in the linkage section. The text string of a link is <reference name>$<entry point name>, where <reference name> is a name specified when the segment is initiated (which is by default the segment's name) and <entry point name> is the name of a procedure in <reference name>. The first call through a link invokes the linker, which uses a set of rules, called the "search rules," to find a segment with reference name <reference name> that contains a procedure named <entry point name>. The search rules specify that initiated segments are always searched first, followed by a list of directories to be searched in order for a segment named <reference name>. Subject to address space issues that have already been considered, the link is turned into a branch instruction to the called procedure; the linker is never invoked again for this call.

MagicSix has supported three different file systems beginning with one compatible with the old M.A.G.I.C. 5.0 system. This file system was written in assembly language. After booting the system to the point that the PL/1 compiler would run under MagicSix, the first version of the native file system, a tree-structured directory system with up to 16 levels, was written in PL/1. Internally the file system is oriented about a

Volume Table of Contents or VTOC having at least one VTOC entry (VTOCE) per file. A complete file name, called an absolute pathname, is a character string of up to 168 ASCII characters in length made up of up to 16 components separated by the ">" character. A sample absolute pathname would be: ">u>lsp>iocss>attach.pl1" which refers to a file named "attach.pl1" in the directory "iocss" which is in the directory "lsp", etc. Given an absolute pathname a file is located on disk by recursively locating its containing directory. The uppermost directory, the "root," has a location that is always known. The MagicSix file system supports branches called links (not to be confused with the "links" for dynamic linking) which are pointers to other branches in the tree. When initiating a link branch, the branch that is the object of the link is initiated.

After every system crash, the file system is made consistent by a program known as the salvager which repairs all damage caused by the crash. Salvages of a large file system can take a long time. After many months of experience it was decided that the file hierarchy should be divided into volumes which have separate VTOCs for the files they contain. A volume is connected to the file system through a directory branch which specifies the volume name and VTOC index of the volume's root directory. The physical disk drive that a volume is located on is specified at the time the volume is mounted. At system boot time certain volumes are pre-mounted and connected to the root volume by system

initialization. A file can only be accessed if all its containing directories are on mounted volumes. It is possible to specify that certain volumes are to be write-protected by the disk server. Every volume can be salavged separately.

In the design of the MagicSix file system, care was taken to provide a robust structure that would isolate errors and prevent them from spreading. To this end it is next to impossible to create two files which accidently share a page of disk; if one should somehow be created, the error will disappear if one of the files is deleted. Certainly the file system could have been made more reliable, but all simple, reasonable steps were taken.

4.3 System Summary

The MagicSix system described above is held together by the "procedure call" mechanism. The system can be viewed as an integrated set of procedures and conventions embedded in a philosophy of system design. The system can be extended by adding new modules without modifying the system kernel, and therefore can easily meet new requirements, because of dynamic linking.

5. Comparison with Other Systems

MagicSix is a unique system for a mini-computer in many ways. Few systems at any price support dynamic linking, as sophisticated a run time environment, and a tree structured file system. However, it is still useful to compare MagicSix to some of its competitors. Three operating systems and their related hardware will be considered below in relation to MagicSix and its hardware.

5.1 Unix on the PDP 11 and the VAX

The Unix operating system was developed by Bell Labs in 1971 to provide a reasonable program development system. The System has evolved in a number of ways over the years and two versions are considered here, Version 6 for the PDP 11/34 and Version 7 for the VAX 11/780. [UNIX]

The PDP 11 is a 16-bit minicomputer and lacks the address space and 32-bit arithmetic of a 7/32. The entire PDP 11 address space fits into a single segment on the 7/32. As such it is difficult to compare MagicSix and this Unix version. Unix does support a tree-structured file system with file system volumes similar to MagicSix, however the implementation is not as resistant to errors. After a system crash a system programmer has to boot the system and fix the file system, while on MagicSix this has generally not been found to be necessary. It is also not

possible to link directories together-- a feature which allows users on MagicSix to move whole subtrees transparently. Unix does support global and owner access while MagicSix currently only has global access by explicit design decision. Unix is not object oriented and maintains the traditional model of creating a core image to run a program. It does not support dynamic linking. Files in the hierarchy of Unix are accessed with traditional I/O methods as opposed to the MagicSix paradigm of segmentation. These comments are not intended to denigrate Unix as a system for a 16-bit computer; it is just not commensurate with the goals of MagicSix. Byte for byte Unix supports more users than MagicSix but this may be attributed to the simple unsophisticated environment and the much smaller PDP 11 address space. Unix does manage to implement almost the entire system in a higher level language which is a direction MagicSix should follow.

In contrast to the PDP 11 the VAX 11/780 has more address space than the 7/32. The VAX has hardware to support demand paging, but it is not oriented towards segmentation. Unix on the VAX is not significantly different from on the PDP 11 except that there is more address space to use in a core load. Unix is too oriented to the small PDP 11 to take advantage of all the features of the VAX. Except for being able to run larger programs, VAX Unix has no advantages over PDP 11 Unix, in fact it seems a waste on a machine like the VAX. VAX Unix does not even currently support demand paging.

5.2 VAX/VMS on the VAX 11/780

The VAX 11/780 is an extension, with significant changes, of the PDP 11 architecture to 32-bits. VAX/VMS is the vendor supplied operating system for the VAX 11/780 system. The system is almost entirely written in BLISS, which is not a true high level language. [VAX]

VAX/VMS is a large scale operating system designed to support the full capabilities of the VAX 11/780. With the address space of the VAX, VMS processes can directly address much more memory than MagicSix. Unfortunately, VMS continues the idea of the core load. Programs on the VAX must be completely linkage edited prior to execution and thus it does not meet the sharabilty criteria. VMS supports demand paging which cannot be done with the 7/32 hardware-- however VMS has each user page against himself rather than against all processes. This paging algorithm causes every VMS user to have to carefully tune the paging parameters for each program. This violates the general rule of system design which states that the operating system should try to maximize overall performance rather than forcing each process to explicitly attempt to maximize its own performance. It also makes more work for the naive programmer. VAX/VMS does not have a real-time display editor and does not have device-independent display support. VAX/VMS has a tree-structured file system but doesn't allow links between directories. Considering that VAX/VMS was

recently developed, it is a very conservative operating system and does not use many advanced concepts which would make it a more flexible, sophisticated programming environment.

Overall MagicSix tries to support a richer environment with a greater emphasis on programming development than VAX/VMS. It tries to do more with less. The VAX 11/780 hardware is considerably more advanced, both architecturally and in implementation, than the 7/32 and it presents a great opportunity for new system development.

5.3 VM/370 on the IBM 4341 or 4331

There are by far more IBM System/370 architecture computers in the world than any other. (excluding micro-processors) IBM has not been traditionally strong on providing advanced time-sharing system from its computers. Of all the various System/370 operating systems, only VM/370 is in any way comparable to MagicSix in that it was designed for time-sharing. [VM1] [VM2]

The 43xx mainframes seem to place these System/370-compatible processors squarely in the mini to midi computer range. (Software support prices from IBM tend to work against this however.) It seems appropriate to compare MagicSix to IBM's time-sharing operating system VM/370. On the hardware level there is again little comparison between a 43xx processor and a 7/32. Both 43xx machines support a 16MB paged virtual address space. The

System/370 instruction set with its limited offsets from base registers tends to complicate compiler writing, particularly in producing position independent code such as that used in MagicSix. VM/370 embodies a completely different approach to systems design from Unix, VAX/VMS and MagicSix. VM provides each process with a "virtual machine," that is the image of a System/370 processor down to the interrupt vectors and virtual devices. While this is a fine system to develop new operating systems under, it is not suitable for non system program development as it stands. To provide a user environment there is the CMS (Conversational Monitor System), a complete single user operating system that runs in a process's virtual machine. VM/CMS does not support a sophisticated file system, editing system, or terminal support. Standard VM cannot provide remote echo for full-duplex terminals or character-at-a-time input which is vital to highly interactive programs. Furthermore the file system is oriented around virtual disk drives: every VM virtual machine is allocated a fixed amount of storage in disk drive tracks. Unlike all the previously discussed systems there is no system-wide pool from which storage is used and returned allowing usage to be leveled over the entire user community. By its very nature and design VM tends to isolate processes and prevent sharing and interprocess communication. VM/CMS is a fine test bed for debugging operating system but as an environment for program development it seems barely adequate. It might be possible to

write a very sophisticated time-sharing system to run in the VM environment, however this has not yet been done by IBM. (Outside vendors such as NCSS and IDC have moved in this direction.) VM meets few of the design goals for MagicSix.

5.4 Comparison Summary

All of the above comments must be taken as being written by an unabashed enthusiast for the MagicSix environment; biases in the estimates of what these other systems offer must be assumed. It does seem that the MagicSix environment stretches the 7/32 to the limit. Perhaps more comprimises would have lead to a significant improvment in system performance. On the other hand, the system does manage to support itself and provide a flexible extensible interface for non-systems programming. As was mentioned much earlier, the emphasis on the system should be to provide a rich environment for the designing and running of programs from the complex to the simple, rather than on the last ounce of mistaken "efficiency."

6. Successes, Failures and Extensions

6.1 Successes and Failures

Thus far MagicSix has been a success at the Archiecture Machine Group. The system has achieved a reliability of up to four days between software crashes. The have been however, some conspicuous successes and failures.

The greatest failure in the system is not having the ability to more closely control the scheduler. The nature of the lab's highly interactive tasks make differential scheduling priorities important if the machine is to multi-program real-time tasks. It would be useful to support real-time processes with low priority backround auxiliary processes. An improved scheduler would also provide overall better performance of the system. Also, real-time interupt handling would be improved if the hardware supported multiple interupt priority levels.

None of the systems programmers had ever written an operating system prior to working on MagicSix. The system code shows in many places the naivete and lack of certain skills that were slowly learned as the system grew. Much of the code has been rewritten at least once but things could be much better. By the end of this year, only one of the original systems programers will still be at the Architecture Machine Group, so system maintainability will become a serious issue. There is very little

written documentation on system internals. There is little doubt that given a second chance, the system would be much cleaner in design and implementation, but there will be no second chance.

On the other hand, the system is fairly reliable. The mean time between hardware failures barely exceeds the mean time between software failures. With little training, non-system programmers have been able to boot the system and salvage the file system. Once someone has learned a few "tricks of the trade" almost any file system problem can be overcome. To the greatest extent possible, the system boot and file system salvaging procedures have been automated.

MagicSix has made more computer time available to the lab than when the computers were run as single-user M.A.G.I.C. systems. Despite the fact that MagicSix is time-sharing, response time is faster than with the old system. MagicSix provides regular backup of the file system to prevent users from losing their work. MagicSix can backup files 10 to 20 times faster than M.A.G.I.C. and can under time-sharing.

Lastly, MagicSix has made writing programs easier. Real-time display editing and dynamic linking have been the most important improvements. Real-time editing makes learning to edit programs and text easy for people without technical backgrounds. It has allowed systems programmers to write and debug major portions of the system without having any program listing facility available.

Dynamic linking, at the least has removed the burden of learning to use traditional static linking programs. Moreover it has facilitated sharing of programs between users in ways not possible in traditional system. For example, a user may correct a bug in a shared procedure and every user of that procedure will get the bug fixed without changing his own programs.

One other successful feature of MagicSix deserves to be singled out. The ability to map data files into the address space as segments has freed users from needing to learn about file I/O. There is no longer any reason to treat external data differently from a procedure's own local variables. A stored table in a segment can be treated directly as an array without the need to open a file and read its contents into a local array. Programmers are allowed to structure data in a logical manner unconstrained by issues of I/O access methods or internal file formats. The real proof that this concept works lies with the many programmers who have said that it simplifies their programs.

6.2 Extensions

There are many areas of extension open to MagicSix. Currently outstanding are projects to complete device-independant display support, to extend file system volumes to make them more flexible to use, and to write a PL/1 display-oriented debugger. There are also many other near term projects such as re-writing the command

processor to be more flexible, bringing up computer network support and general system cleansing. The lab will also soon investigate methods of using MagicSix in a distributed processing environment. No one knows where MagicSix will head in the long term.

7. A Guide for the Future

It is hoped that MagicSix will have some impact on the development of future operating systems. The system has demonstrated three things, each of which can aid in future systems work. No matter what will happen to the system in the future, it has provided a unique learning experience for everyone who has been involved in its development.

The MagicSix development effort has shown that it does not have to be expensive or take forever to develop an advanced operating system. If undergraduate students working part time can create this system from scratch there is no reason why it cannot be done commercially. It should be noted that essential to the development effort was the small number of people involved--after a certain point, the more people working on a project the slower it goes. Furthermore, the systems programming group was not hindered in any way from experimenting with many different ideas for the system and its implementation and no results were expected for over two years.

A segmented virtual address space used to create an object-oriented environment is a powerful tool for system design and user programming. Combined with rings it provides a way to integrate the system and user programs together by mean of a single mechanism-- the procedure call. With dynamic linking it allows for natural system extensibility and a flexible user

environment which can be tailored to specific individuals or projects. For user programming it removes the need to learn about file I/O and data access methods. Data files can be easily accessed as program variables. Dynamic linking provides the same flexibility to user ring sub-systems as it does to the operating system.

Finally MagicSix shows that reasonably sophisticated operating systems can be constructed on mini-computers provided with the barest necessities. It indicates that hardware designers should more carefully consider how their decisions impact the writing of operating systems especially in the area of memory management and overall address space. Minor changes in computer architecture can make or break an operating system.

7.1 Conclusion

The main concern of and influence on the design of MagicSix was that the system provide a powerful environment for program development. The structures needed to meet that goal were developed from ideas borrowed from two systems, Multics and ITS, and adapted to the particular situation. The question remains why more systems don't begin with the user firmly in mind. If the computer is ever to fulfill its potential more systems will have to be written with the goal of better supporting their users than have been to date.

Appendix I - Maintaining MagicSix

MagicSix has essentially reached a stable state. It will be possible for the Architecture Machine Group to use the current MagicSix with few changes for at least a year. In the long term, however, the lab will need to train new system programmers to provide the inevitable changes required by future projects. Considering the labs dwindling systems programming staff, in the next year, only essential maintenance and reliability problems should be addressed.

It has become all to evident that the Interdata 7/32 can never provide suitable service under MagicSix. The hardware is not only limited to 512K bytes of main memory and improperly designed memory mapping but it is extremely unreliable. The 7/32 I/O architecture can only marginally support the fast CDC storage module disks that the lab wishes to use. In view of all the hardware difficulties MagicSix must be modified to be more fault resistant. It should be noted that any changes will be limited to more gracefull crashes and providing more hardware diagnostic information.

In the area of crashes, it is possible to recover almost all the segments that were modified in core but not yet written to disk. Thus, an emergency shutdown routine could be developed to save those segments after a system crash, greatly reducing the amount

of work lost. To aid in restarting non-fatal crashes, debugging crashed systems, and bringing the system up, a system debugger should be developed. This debugger need only have the capabilities to examine and set memory (and the registers) and restart the processor at an arbitrary location.

The system should also log all disk errors, including the disk address of the error and type of error. Ideally this log should be kept online in a file for future analysis. On a hard disk error, the system should return to the debugger for analysis by a systems programmer who can determine if the system can be safely restarted. Furthermore, since the Interdata disk controller cannot detect errors when writing disk blocks, a special read-after-write mode should be implemented to be used when disk failure is suspected. Ideally all these methods should be supplemented by a complete log for every processor of every hardware change, hardware failure and system failure to be saved for correlation analysis.

Finally is should be noted that operating systems place a much greater burden on a computer system than any conceivable diagnostic. Diagnostic programs however, provide a very controlled environment useful in locating specific hardware error while operating systems are not in the least limited and controlled. Thus while the operating system can detect errors it will be difficult to produce the repeatable situations needed to

isolate a problem. Even such innocuous changes as disk read-after-write will completely change the overall timing in the system. As has been seen repeatably in the development of MagicSix, timing and system load are key determiners of hardware errors.

Perhaps the best course the Architecture Machine can take is to replace the 7/32 with compatible new hardware that is more reliable. Interdata has recently announced a new series of 32-bit 7/32 compatible processors which will hopefull prove to be better than the 7/32. These machines might provide the needed growth path for MagicSix. The lab must also try to recruit undergraduates to become systems programmers. Unfortunately it is difficult to find people who are competent and have the desire for systems programming.

Appendix II - References

[BSG] Greenberg, Bernard S., EMACS Documentation, Online Documentation on the MIT Multics Comouter System, (in file >udd>Multics>Greenberg>emacs>editor.info), MIT Multics, Cambridge, Mass. 1979

[DLW] Weinreb, Daniel L., A Real-Time Display-Oriented Editor for the Lisp Machine, S. B. Thesis, MIT Electrical Engineering and Computer Science Department, Cambridge, Mass., January 1979

[ECC] Ciccarelli, Eugene, An Introduction to the EMACS Editor, MIT Artificial Intelligence Laboratory Memo 447, MIT AI Lab, Cambridge, Mass., January 1978

[IBM:ASD] Technical Newsletter No. 11, IBM Applied Science Division, International Business Machines Corporation, March, 1956

[IDATA] Interdata 32 Bit Series Reference Manual, Publication Number 29-36R01, Interdata Inc., 1973

[ITS] Eastlake, Donald, Greenblatt, Richard, Holloway, Jack, Knight, Thomas, and Nelson, S., ITS 1.5 Reference Manual, MIT Artificial Intelligence Laboratory Memo 161, MIT AI Lab, Cambridge, Mass., July 1969

[MLK] Kazar, Michael L., Dynamic Linking in a Small Address Space, S. B. Thesis, MIT Electrical Engineering and Computer Science Department, Cambridge, Mass., May, 1978

[MagicSix] MagicSix Commands Manual, Internal Publication, MIT Architecture Machine Group, Cambridge, Mass., March 1978

[MPM Reference] Multics Programmer's Manual, Reference Guide, Order Number AG91, Honeywell Information Systems, Waltham Massachusettes, 1978

[OTA] Anderson, Owen Theodore, The Design and Implementation of a Display-Oriented Editor Writing System, S. B. Thesis, MIT Physics Department, Cambridge, Mass., January, 1979

[PSL] Rosen, Saul ed., Programming Systems and Languages, McGraw-Hill Inc., New York, New York, 1967

[PSL:ATLAS] Kilburn, T., Payne, R. B., Howarth, D. J., "The Atlas Supervisor," (Contained in PSL, pg. 661) Proceedings of the 1961 Eastern Joint Computer Conference, AFIPS, vol. 20, pg. 279-294

[PSL:D825] Thompson, R. N. and Wilkinson, J. A., "The D825 Automatic Operating and Scheduling System," (Contained in PSL, pg. 647) Proceedings of the 1963 Spring Joint Computer Conference, AFIPS, vol. 23, pg. 41-49

[PSL:EXPERIMENT] Corbató, F. J., Daggett, M. M. and Daley R. C., "An Experimental TIme-Sharing System," (Contained in PSL, pg. 683) Proceeding of the 1962 Spring Join Computer Conference, AFIPS, vol. 21, pg. 335-344

[PSL:History] Rosen, Saul, "Programming Systems and Languages-- A Historical Survey," (Contained in PSL, pg. 3) Proceedings of the Spring Joint Computer Conference, April, 1964

[PSL:LINKAGE] McCarthy, J., Corbató, F. J., and Daggett, M. M., "The Linking Segment Subprogram and Linking Loader," (Contained in PSL pg. 572) Communication of the ACM, Vol. 6, pg. 391-395

[PSL:MULTICS] Corbató, F. J. and Vyssotsky, V. A., "Introduction to the Multics System," (Contained in PSL, pg. 714) Proceeding of the 1965 Fall Joint Computer Conference, AFIPS, vol. 27, pg. 185-196

[PSL:OPERATING] Mealy, G. H., "Operating Systems" (Contained in PSL, pg.516) Rand Corporation Report P-2584, May, 1962

[PSL:SEGMENTS] Dennis, J. B., "Segmentation and the Design of Multiprogrammed Computer Systems," (Contained in PSL, pg. 699) IEEE International Convention Record, Institue of Electrical and Electronic Engineers, New York, 1965, Part 3, pg. 214-225.

[RK] Kovalcik, Richard Jr., A Display-Oriented Debugger for PL/1, S. B. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, Mass, May, 1979

[RMS] Stallman, Richard M., ITS EMACS Order, Online Documenation on the MIT AI Lab PDP 10 (in file AI:INFO;EMACS ORDER), MIT AI Lab, Cambridge, Mass.

[SAS] Steinberg, Seth A., A PL/1 Subset for a Minicomputer, S.B. Thesis, MIT Electrical Engineering and Computer Science Department, Cambridge, Mass., May, 1974

[TREK] Reminiscent of the television show Star Trek created by Gene Roddenberry.

[VAX] Vax 11/780 Software Handbook, vol. 3, Digital Equipment Corporation, 1977

[VM1] VM/370 CMS Users Guide, Order Number GC20-1819-1, International Business Machines Corporation, Poughkeepsie, New York, August, 1977

[VM2] Vm/370 CP Commands Manual, Order Number GC20-1820-2, International Business Machines Corporation, Poughkeepsie, New York, August, 1977

[UNIX] Ritchie, Dennis M. and Thompson, Kenneth, "The Unix Time-Sharing System," Communications of the ACM, vol. 17, number 7, July 1974