Introduction to Time-Sharing Concepts

R.W. Watson

Technical Progress Report No. 249-68
Project No. 78140

SHELL DEVELOPMENT COMPANY
A DIVISION OF SHELL OIL COMPANY
Emeryville, California

249-68

INTRODUCTION TO TIME-SHARING CONCEPTS

Project No. 78140

TIME-SHARING SYSTEM EVALUATION

Technical Progress Report No. 249-68

This report is based on work through June 1968

Issued: January 1969

Written and Reviewed: R.W. Watson, A.E. Sassus, C.H. Holm, M.D. Kudlick, F.G. Stockton

Approval: R.S. Miller

CONTENTS

CONTENTS (Contd)

ABSTRACT

The development of computer operating systems has been motivated by a desire to achieve more effective utilization of total system resources. Resource sharing is the fundamental concept. Different classes of operating systems can be distinguished by looking at the relative importance given to basic computing resources, including that of the human user.

Operating systems share resources among several processes, although most resources can only be used by one process at a time. It is for this reason that such a wide variety of systems have been called "time-sharing" in papers and advertising. In the most common concept of "time-sharing", the user is a prime system resource and is effectively made a part of the system by being able to interact on-line from a terminal with a wide range of hardware - software resources; receiving in turn rapid response from the system.

This paper explores some of the meanings of the phrase "time-sharing" in current usage, introduces in detail basic hardware - software concepts found implemented in varying forms in "time-sharing" systems, and discusses some of the difficulties and trends which have developed and are developing in the field. Most of the examples in the discussion are taken from the SDS-940 or the Multics (GE-645) time-sharing systems. This survey differs from other introductions to time-sharing in its greater level of detail and its attempt to make clear the motivations for various developments in the field.

INTRODUCTION TO TIME-SHARING CONCEPTS

Introduction

The development of computer operating systems has been motivated by a desire to achieve more effective utilization of total system resources. Resource sharing is the fundamental concept and different classes of operating systems can be distinguished by looking at the relative importance given to basic computing system resources such as the

arithmetic-logic unit,

main memory,

auxiliary memory (drums, disks etc.),

peripheral devices (card readers, printers etc.),

communications equipment (communication lines, remote terminals),

compilers,

data files,

service and application routines, and

programming and debugging time.

Because operating systems share or multiplex resources, most of which can only be used by a single process at a time, all operating systems could be called "time-sharing" systems. This sharing of resources sequentially in time is the reason such a wide variety of systems have been called "time-sharing", the user is a prime system resource and is effectively made a part of the system by being able to interact on-line from a terminal with a wide range of hardware-software resources; receiving in turn rapid response from the system.9)10)26)31)47)

The purpose of this paper is to explore some of the meanings of the phrase "time-sharing" in current usage, to introduce important basic concepts found implemented in varying forms in "time-sharing" systems and to discuss some of the difficulties and trends which have developed and are developing in the field. This paper is written to be self contained, although some knowledge of computing is assumed. The paper is divided into two main parts, one introducing hardware concepts, and the other introducing software concepts. Because hardware and software are so intimately interrelated, some software concepts are introduced during the hardware discussion and vice versa. For example, software concepts are introduced during the discussion of segmentation and hardware concepts are introduced during the discussion of terminal input/output. The reader approaching this material for the first time should pass by the more detailed examples.

A wide variety of systems can be classified. A classification having some utility is the following:

9) See Bibliography.

Class 1) General purpose time-sharing systems
(Examples being the GE-645, IBM 360/67, RCA Spectra 46, SDS 940, DEC PDP-10, SDC-Q32, MIT CTSS)

Class 2) On-line file maintenance and retrieval systems
(Examples being the American Airlines Sabre System, Banking and Stock Information Systems, Shell's STADAC II System, On-line Inventory Control Systems)

Class 3) Multiprogramming batch systems allowing remote access
(Examples being IBM's OS/360, UNIVAC's Exec 8, GE's GECOS, CDC's SCOPE)

Class 4) Special purpose time-sharing systems
(Examples being RAND's JOSS, GE's BASIC, IBM's QUICKTRAN)

The examples listed above are by no means exhaustive. We are primarily interested in class 1 systems in this report, although the other classes are discussed briefly to help clarify the differences among them. All four classes share many fundamental building blocks in their implementation. To understand the detailed design concepts of any one class provides solid foundation for the study of the others. A detailed discussion is given of requirements for, and concepts of, class 1, the general purpose time-sharing systems. After the discussion of general purpose time-sharing systems the other classes of system listed above are considered.

The discussion takes most of its examples from two systems, the SDS-940 and the Multics (GE-645) systems. These two systems were chosen because one, the SDS-940 was designed with modest goals in mind, is easy to understand, has been commercially successful and illustrates well the major concepts in the design of a time-sharing system. The second system, the Multics system, was designed with the ambitious goal of making a quantum jump in the conceptual state of the art, is more difficult to understand, is still in the research and development phase, and illustrates at a high level generality the major concepts in the design of a time-sharing system. The SDS-940 was developed at the University of California, Berkeley and is a 24 bit medium scale system giving good response to around 24 on-line users. The Multics system is being developed at MIT's project MAC in a joint effort between GE, MIT and the Bell Telephone Laboratories and is a 36 bit large scale system expected when operational to give service to over 100 on-line users.[a] Although Multics is run on the GE-645, we prefer to use the

a) The number of users a system will support with good response is difficult to estimate. This number is a function of the type of computations each user has placed on the system. For example, a system might be able to support several hundred users (assuming enough teletype connections were available) performing small engineering calculations, but might be able to support less than fity users with large linear programming or general data manipulation type jobs.

name Multics to indicate the developmental nature of the project and thus the fact there is not yet a commercial version of the system called the GE-645. Besides drawing examples from the above two systems other systems will be used in examples where appropriate.

General Purpose Time-Sharing Systems

One of the distinguishing features of a general purpose time-sharing system, besides rapid response to on-line users, is the capability for concurrent utilization of a large variety of software facilities. Such facilities include text editors, debugging aids, assemblers, a variety of compilers, a library of special application routines, file management facilities and an open ended ability to add additional facilities. No limitations are placed on the type of files created or the users ability to create programs to manipulate these files. Further, such systems are designed to allow upwards of two dozen or more users to interact with this generality and to expect rapid system response to their terminal actions.

Some General Conclusions

General purpose time-sharing systems are one class of resource sharing system. It is not yet clear that one system can be designed to meet economically the requirements of the four classes of system described above. However, many important functional capabilities are required in common by these various classes of system. The detailed implementation of the functions vary among systems, because each class gives greater emphasis to a different set of resources. The prime resource to be shared is main memory. The hardware design must take into account all aspects of memory design; memory addressing, memory protection, memory allocation, flow of information within memory hierarchies, and memory bus organization and design. No current machine has a memory system design fully adequate for a large scale general purpose time-sharing system. Such systems as the IBM 360/67, and GE 645, while containing many important memory system ideas, should still be considered research and development efforts and not commercial products. Resource sharing computers, because of the intimate interrelation of hardware-software, must be designed by men with experience in both areas. Such men are only now being trained by universities and industry. Promising developments in various aspects of memory system design exist on several machines or are in the research and development phase. Experience with these systems coupled with decreasing hardware costs should lead to economical large scale time-sharing systems in the mid 1970's.

The problem of "swapping", which results because not all programs can reside in main memory and therefore must be moved (swapped) between main and auxiliary storage, must be taken into account from the earliest system design phases. Many current difficulties have resulted because it was assumed resource allocation algorithms could be designed independently of hardware performance characteristics.

Simulation and mathematical analysis as design tools can be expected to receive increasing attention. Early comments in the time-sharing field that this type of system was too complicated to simulate or analyze resulted from the designers lack of experience in this area and unwillingness to devert the effort required to develop the appropriate techniques. These same tools should also prove useful to computer users in configuring systems and choosing resource allocation algorithms appropriate to their installation.

System protection is another very important area in the design of a resource sharing system. At present most of the protection mechanisms exist as software. As experience is gained, more protection mechanisms will appear in the hardware, thus leading to system efficiencies.

Until the past two years, time-sharing development was primarily found in the universities. The rapid rise of time-sharing service bureaus and the use of these services and inhouse time-sharing systems by industrial and research firms has broadened the area of time-sharing knowledge,and research and development. The difficulties and opportunities in implementing and utilizing such systems, now being more widely recognized,should lead, in the next few years, to systems being designed from the ground up for resource sharing, rather than having features just tacked on to facilitate resourch sharing as has occurred on second and third generation systems.

General Concepts

The most general term which can be used to cover the four classes of systems listed above is multiprogrammed. Multiprogramming is the concurrent execution of two or more processes in one computer system. By concurrent we mean that two or more processes are in partial states of completion and that resources are being allocated among these processes to balance system loads and to meet response or throughput requirements. In a system which is not multiprogrammed,one job is completed before the next one begins. The term multiprogramming is not to be confused with that of multiprocessing which is used to designate a system having more than one central processing unit (arithmetic logical unit).

Present technology and the technology of the forseeable future places constraints on the size of main storage available. This fact has two main effects: 1) not all processes[a)] in a time-sharing system can be in main storage at once and 2) the size of the physical address space available to a process is restricted. Because all processes can not be simultaneously in main storage,some must reside on auxiliary storage devices such as drums or disks between times in which they are executing instructions. This movement of processes between main and auxiliary storage is commonly called "swapping". Another way to look at the problem is to think of auxiliary storage as containing the totality of information required for the complete execution of all computations; it is the task of the system to maintain in main memory a portion of the totality that is relevant to some subset of the active computations. Another way to look at the problem is to consider each resource as having certain properties relating to its ease of sharing or multiplexing. For example, processors are easy to multiplex among processes because only a few fast registers need to be changed. Memory is more difficult to multiplex,

a) The terms process, program, user, and job are defined later in this section.

possibly requiring a swap operation. This swap operation limits the speed with which memory may be multiplexed. The multiplexing properties of memory create major problems for both the hardware and software designers. These problems are discussed in this paper.

In order that processes can be started and stopped in such a way that this swapping is invisible to a process, certain information must be saved. This collection of information is often called a "state vector" because it defines the state of the machine at the time the process is stopped.[15]

The state vector of the total machine includes:

1) the contents of the program counter

2) the contents of the central registers of the machine. These registers include programmer accessible registers and, depending on the machine addressing structure, may include some non-programmer accessible registers.

3) the address space of the machine and the contents of every address in it.

4) the state of all input/output (I/O) devices attached to the machine.

The state vector for a given process requires less information than listed above because the individual process does not need to know the state of all the I/O devices nor the contents of all the addresses. Further, code techniques exist which give indirect information about a process' address space which allow a process' state vector to be stored in a few machine words. In effect, switching from one process controlling the machine to another is accomplished by switching the state vectors controlling the system.

The concept of address space needs clarification because it is related both to the swapping problem and the addressing problem as seen by a programmer. Addressing is the means by which a process distinguishes among the storage locations in its address space. Two address spaces exist: 1) the physical address space which consists of the actual main and auxiliary storage locations physically and directly addressable and 2) an abstract or logical address space (often called the name space[15]) which consists of the contents of abstract or logical locations addressed by processes. If the logical address space is larger than the physical address space of main memory, the term virtual memory is commonly used. The term virtual memory has also been used in place of logical memory. In this paper the term virtual memory will denote a logical space larger than the physical space of main memory. The mapping from an abstract address space to physical addresses is handled by both hardware and software techniques. Some of the more common are discussed in appropriate sections of this report.

A processor is an entity which performs transformations of information. More simply put, a processor executes instructions stored in main memory. A processor can be implemented in software such as an operating system or compiler or else can be implemented in hardware such as an arithmetic-logic unit, or an input-output unit. In this paper the term "processor" denotes a hardware device. A process is an entity which can control a processor: it is a set of commands or instructions and data. For the purposes of this paper a process is an entity which has a state vector and is thus capable of being swapped.[a] The time-sharing system may be designed so that sub-processes are allowed, each of which can have a state vector which is created by a process higher in the hierarchy of processes. This hierarchy of processes can have its members executing independently as seen by the system. Processes are made up of programs and subprograms or, equivalently, procedures.

Every process in the system is part of a job which can be thought of as a tree of processes. The basic feature of a job is its complete independence from other jobs as far as the system is concerned. The root of the tree of processes is created by the system and is responsible for recognizing commands to the system, handling illegal actions by lower order processes, and initiating and destroying subprocesses.

A job is a convenient unit for accounting purposes, although accounting could in principle be handled for individual processes. The term user has a wide variability of usage, being a synonym for process or job or for the human sitting at the console. A user can create multiple jobs. A user has the following attributes:

1) a name
2) authority to expend certain resources
3) a collection of permanent and temporary files holding data, and possibly authority to access other files
4) a collection of 0 or more jobs.

Associated with the user are certain protection and access rights. A user is a named accounting entity and thus could be one or more people using this authority to access the system. Generally throughout this paper we use the term user to refer to a person sitting at a terminal, but the properties of a user given above should be borne in mind.

Hardware Concepts

The type of hardware features required for general purpose time-sharing systems are for the most part also required for the other classes above as well.[18][19][24][30][31][32][36][45][54] The essential characteristic of such systems is the dynamic allocation of system resources. The major hardware features required by such systems are: 1) protection mechanisms to safeguard one process from another and the system from itself and user

a) In IBM's terminology,the term task is used in a manner similar to the usage here of the term process. Another synonym for process is computation and for variety we use the two terms interchangeably in this paper.

processes, and 2) mechanisms which allow efficient dynamic allocation of resources. Both requirements above are interrelated. An additional requirement of such systems is high reliability.

Resource Allocation

The central resource in current systems is main memory. It is main memory which holds instructions for the arithmetic-logic processors (CPU's) and for the input-output processors (IOP's), is used as a buffer for information passing over communication lines and moving between various I/O and secondary storage devices, and holds the code for the resident operating system. The proper design of the memory system is critical to the success of a large scale time-sharing system. Up until the past few years the standard computer system model used by most people was that of Figure 1. In this model the arithmetic-logic processor is shown in the center and has been called the central processing unit.

There are five major problem areas in the design of a total memory system:

1) development of memory addressing techniques which allow processes to address a logical address space, possibly larger than the physical space of main memory,
2) development of physical memory allocation techniques and techniques for mapping the logical address space into the physical address space,
3) development of memory protection techniques,
4) development of a memory organization and bus structure which permits all processors and devices utilizing main memory to operate at full speed with minimal interference,
5) development of techniques to utilize a hierarachy of storage devices in a device independent manner.

The latter problem is briefly discussed in a later section. Problems 1, 2 and 3 are usually solved with interrelated techniques and have been given considerable attention. Problem 4, while being recognized, has not until very recently been given the attention required.

Memory Addressing and Allocation Techniques

The multiplexing properties of main memory devices must be fully considered, in order to specify an appropriate addressing and allocation scheme for a time-shared computer. In principle, any fraction of main memory may be allocated to a process. This is not true of processors, which may only be allocated as a unit. Processors can be multiplexed rapidly while main memory cannot be as easily multiplexed because of the swapping requirement. The design of an addressing scheme for a time-shared computer must emphasize the allocation advantages of main memory and make it possible to minimize the multiplexing disadvantages. Movement of information between

Figures 1 and 2 follow

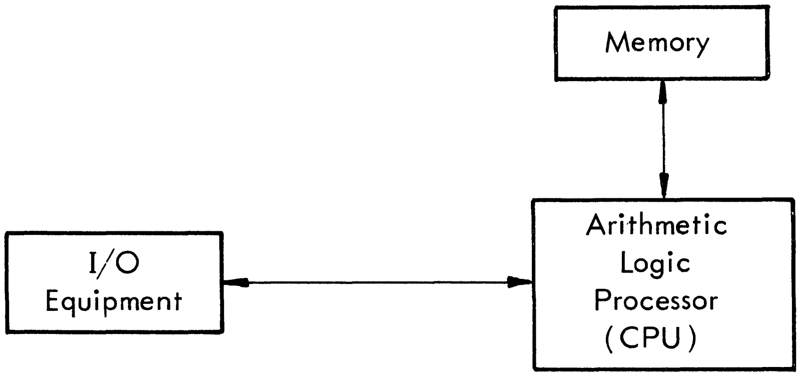


Figure 1. CPU CENTERED MODEL OF A COMPUTER SYSTEM

Now another model is required which shows the memory as the central resource as in Figure 2.

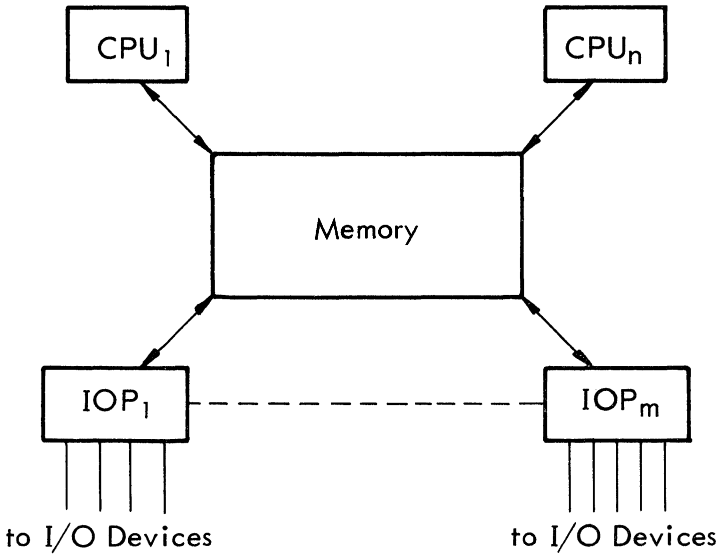


Figure 2. MEMORY CENTERED MODEL OF A COMPUTER SYSTEM

249-68
64692

main memory and auxiliary storage in one of the major sources of machine overhead and idle time.

In order to minimize this movement, one wants to utilize main memory as effectively as possible.

For example, one would prefer to have only one copy of a particular procedure, say a compiler, in core used by several processes rather than requiring each process to obtain a separate copy. Programs which are designed to be shared by several processes are often called reentrant programs or pure procedures. A reentrant program has two characteristics, 1) none of its instructions or addresses are modified during its execution and 2) temporary storage and data areas are maintained outside the procedure itself, usually in the memory space of the calling programs. Although programs with the above capability can be written for machines with a wide variety of addressing techniques, some addressing techniques make the writing and protection of such programs simpler.

Another way to utilize memory more effectively is to be able to have flexibility with respect to where in physical memory process may be placed. This ability to dynamically relocate processes in physical memory can be achieved with a variety of addressing and allocation techniques.

The effect of the addressing and allocation scheme on the user must also be considered. The cost of designing and implementing application systems and even the possibility to attack classes of problems is likely to be affected by the properties of the addressing and allocation scheme. The various trade offs possible in the design of an addressing and allocation system must take into account user needs as well as system considerations.

The first decision a designer must make is the size of the logical address space, in particular, is it going to be smaller than, equal to, or larger than the physical address space. The structure of the logical address space must then be determined. Many structures are possible such as the large linear array commonly used, or a set of linkable linear arrays as found in Multics, or a tree structure, and so forth. The decision must be made as to how much of this structuring to perform in hardware and how much to perform in software. After this decision is made, the technique of translating or mapping the logical addresses to physical addresses must be determined. There are three points at which this mapping can take place.32)44)

1) When the procedure is prepared as an operable computer program. The result is an absolute program which, in effect, is assigned the same resources each time it is run.

2) When the program is loaded. This is known as static relocation.

3) When the program is in execution. This is called dynamic relocation.

Memory protection schemes are easily developed for any of the above approaches and a discussion of this topic is given in a separate section. In this section, some of the factors influencing the choice of a structure and mapping technique are considered. We consider only the linear array or set of linear arrays, because more specialized structures such as trees or rings are usually left for implementation by software processors.

The translation of data references to physical addresses is easily accomplished during program preparation, but suffers from the severe problems which arise when one attempts to share or change programs. Further, translation at that time restricts the logical address space to that of the physical address space.

The process of static relocation is not trivial and involves a fair amount of computation. With static relocation,a user can be initially loaded anywhere in memory. However, when the process is removed to auxiliary storage and then returned during swapping, it must be placed in the same locations as before, to avoid the loading process. (Even to go through the loading process again implies that the procedure and data must meet special conditions.) The major gain of static relocation is that during the loading process, independently written programs and data can be combined into a computation with proper linking of parts. The proper mapping to the physical address space is performed by the loader. Each program can be written in a logical space of its own, but no duplication of symbolic location names is allowed, although programming techniques can be developed to resolve such duplication.

The ability to load programs anywhere in physical memory is useful in the linking process above, but of little value in achieving effective memory utilization in a time-shared system. For example, when a new program is to be started, the system can attempt to find a computation which would fit in an available block of cells. If such a computation can be found and it can remain in main memory until completion, static relocation is sufficient to enable several computations to be share main memory. (The assumption of some sort of memory protection scheme is implicit. This topic is covered in a separate discussion.) A more usual situation will be that the total number of free cells available is equal to that number required by a new computation, but that these cells are not in a contiguous block. If swapping is required, then even if a contiguous block was available on initial loading, the same contiguous block is not likely to be available each time the computation is run, without moving some information to another spot in main memory or moving it to secondary storage. Because of the above reasons, systems without dynamic relocation hardware, when used for time-sharing, generally have allowed only one computation to reside in memory at a given time.9) Thus, during the swapping process the system must remain idle. It is the above situation which motivated the development of dynamic relocation methods.

Dynamic Relocation

Base Registers. The simplest and most common dynamic relocation technique uses base registers.32) A base register is a register that can have its contents added to the address of each memory operation. By adding the contents of a base register to all addresses, one can load a computation anywhere in memory in a block of contiguous cells and then set the appropriate base address of the program into the base register. Using base registers, programs are initially loaded using static relocation techniques, but can be dynamically relocated as a unit later, without going through the loading process. This flexibility results because the loading is to logical space not physical space. The base registers in effect form a hardware map which maps logical space to physical space. Further flexibility is gained if there is more than one base register, which facilitates sharing of programs and gives the possibility of splitting a program for loading into non-contiguous storage areas.

Program sharing is performed in a system using base registers by writing the reentrant programs to make memory references to themselves through one base register and to make memory references to data in the calling process through a second base register. The size of the logical address space using static relocation or dynamic relocation using base registers is usually equal to or less than the size of the physical address space. A larger physical space can be simulated by the user by explicitly overwriting a portion of his computation not immediately required with another part brought in from auxiliary storage. This process is called overlaying. Overlaying is closely related to the concept of swapping, except that overlaying is a user responsibility whereas swapping is a system responsibility. Overlaying requires careful organization by the programmer of the physical memory requirements of his computation. Careful planning is required to assure that no two procedures or data structures which are to be used concurrently occupy the same positions in logical or physical space. Whether or not overlay planning should be looked upon as a chore or as an opportunity for programming discipline is a question open for discussion. Certainly one can provide the programmer with system aids to facilitate overlay planning and implementation.

One of the problems uncovered with static relocation was the fact that once loaded a computation's address references were bound to a certain contiguous area of memory and that during swapping the computation had to be returned to the same area of main memory each time it was to be given control of the processor. Using base registers, this restriction no longer holds. When the processor is to be switched to a computation not in main memory, a free contiguous block of main memory must be found for it to reside in. If such a block exists, then no information needs to be transferred to auxiliary memory. The more usual situation which results is that while enough free cells may be available in main memory for the computation, they are not in a large enough contiguous block. In this case, a system designed to use base-registers can do three things: 1) search for a process which will fit into one of the available contiguous blocks, 2) swap out part of some process presently in main memory bordering on a free area inorder to make a large enough contiguous area, or 3) perform a compacting operation on main memory. Systems giving good user response can be designed using one or more of the above approaches.1)

One possible way out of the problem mentioned above of finding a large enough contiguous area would be to use multiple base registers so that smaller pieces of the process could be loaded into existing free spaces. This approach would be impractical because the instructions of a given piece must refer to the correct base register. Thus, the programmer or compiler must decide how to split up the process and which base registers to assign which pieces. Binding of base register addresses at load time constitutes a binding of the process to a portion of logical space. The system could not perform this function dynamically because it would be very time consuming and complicated. When a process is started up, the proper numbers must be placed in the base registers, which implies that system conventions must be established so that the system knows which base register a given program piece is using. Conventions would also be required so that shared programs could use different base registers from those of the calling process or other shared processes being used concurrently. Even with all the above complexity, the problem would not be solved because pieces of free space smaller than the program pieces would result after the system had been running. To get around this problem one could break up processes and physical memory into uniform sized pieces. Such a step leads us to the concept of paging introduced below.

The difficulties with the base register approach are:

1) Software difficulties arise if more than two base registers are used. This limitation to two base registers implies that memory cannot be fully utilized because contiguous free areas smaller than needed by many computations will develop during operation. The inability to achieve full memory utilization may lead to compaciting or swapping which theoretically could be avoided if the free areas could be fully utilized.

2) Logical address space is limited in size by the size of physical address space. This statement is not a theoretical limitation because one can devise ways of creating virtual memory using base registers; but a few minutes reflection demonstrates that pratical difficulties make such schemes uneconomic.

The question has been asked whether the added hardware and software complexity and expense, to overcome the problems listed above, introduced by more sophisticated dynamic relocation techniques really results in a corresponding increase in system efficiency, improved response and programming ease. We discuss some of the arguments pro and con after introducing further dynamic relocation concepts.

Paging. The characteristic of dynamic relocation using base registers, which requires programs to be located in contiguous areas of main memory, leads to difficulties, as pointed out above, in fully utilizing main memory, because free areas develop which are not large enough to be used. If, however, programs and main memory could be broken into small units and the program pieces could be located in corresponding sized blocks anywhere in main memory, then the possibility exists of more effectively utilizing main memory. "Paging" is the name given to a set of techniques which enable such memory fragmentation to be implemented. Paging techniques can also allow economic implementation of a logical memory space larger than the physical memory space.[15)]

In a paged system, physical memory is considered to be broken up into blocks of a fixed size, usually 512 words, 1024 words or 2048 words. In recent systems the page size can be changed dynamically by the system. The memory can be more fully utilized by the resident monitor system if small page sizes are available (64,256 words). The use of multiple page sizes does lead to increased time spent by the system in swapping, as discussed later, and therefore, this use should be restricted. The programs are also considered to be split into pages of a size equal to the block size of physical memory. Thus, the address in such a system is considered to be represented by two numbers: 1) a page address or number and 2) a line-within-page address. For a machine with an n-bit address field, the top p bits are considered the page address and the remaining n-p bits are the line address. The addressing scheme on the SDS-940 is paged and illustrates the concepts involved.

A process in the SDS-940 can be as large as 16K[a)] words. Physical memory in the SDS-940 can be as large as 64K and thus the logical address space is smaller than the physical address space. The more general case of a paged system yielding a virtual memory is discussed later. A process in the SDS-940 is broken up into 2K word pages and memory is similarly broken into 2K word blocks. There are 14 bits in the address field of a 940 instruction word broken into two parts, a 3 bit page number and an 11 bit line-within-page number. The relocation mechanism, illustrated in Figure 3, consists of eight 6 bit registers called a memory map. These registers are considered numbered 0 to 7 and correspond to logical pages. Within the map register is a number for the actual physical block containing the code for the logical page. For example, in Figure 3 logical page 0 is in physical block 32, logical page 1 is in physical block 3 and so forth.

The logical address is converted to a physical address as shown in Figure 4.

The 3 bit page number indicates which map register contains the physical block number where the page actually resides. The map register is 6 bits long and is shown in Figure 5.

a) K = 1024.

Figures 3 and 4 follow

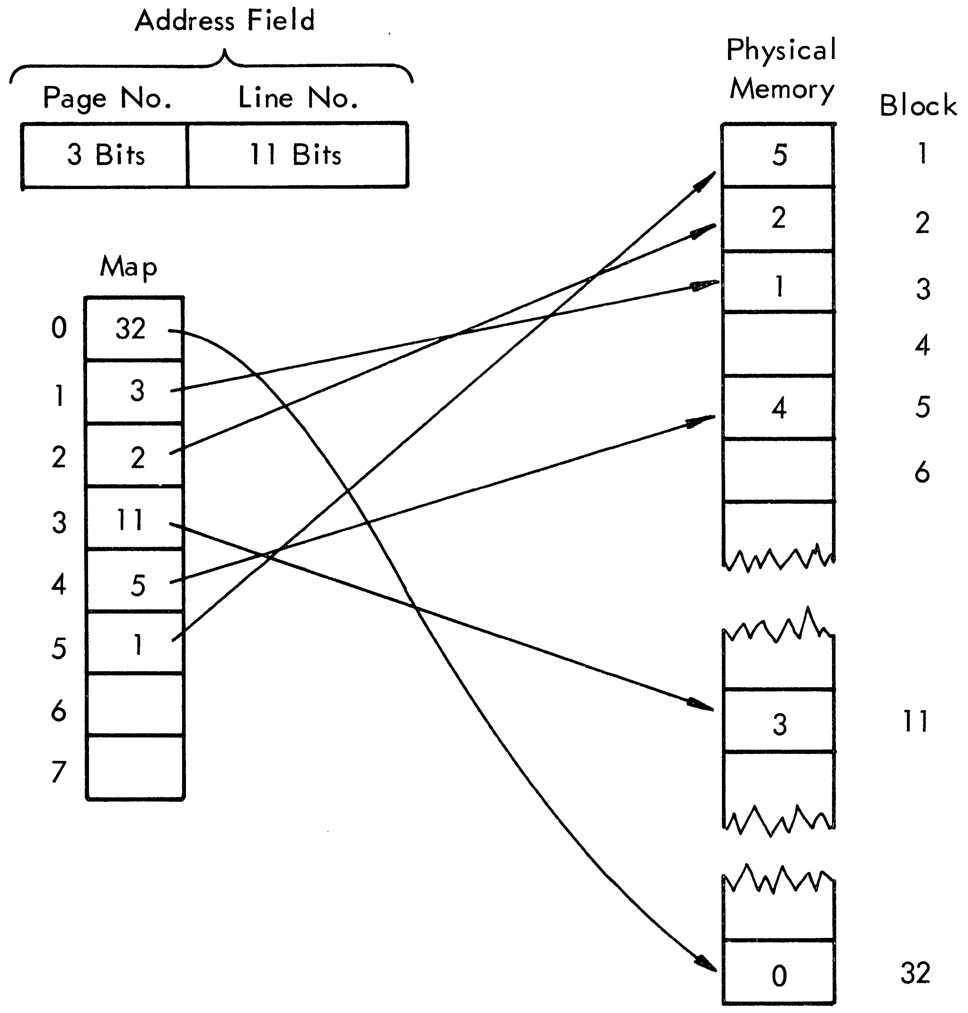


Figure 3. PAGING IN THE SDS-940

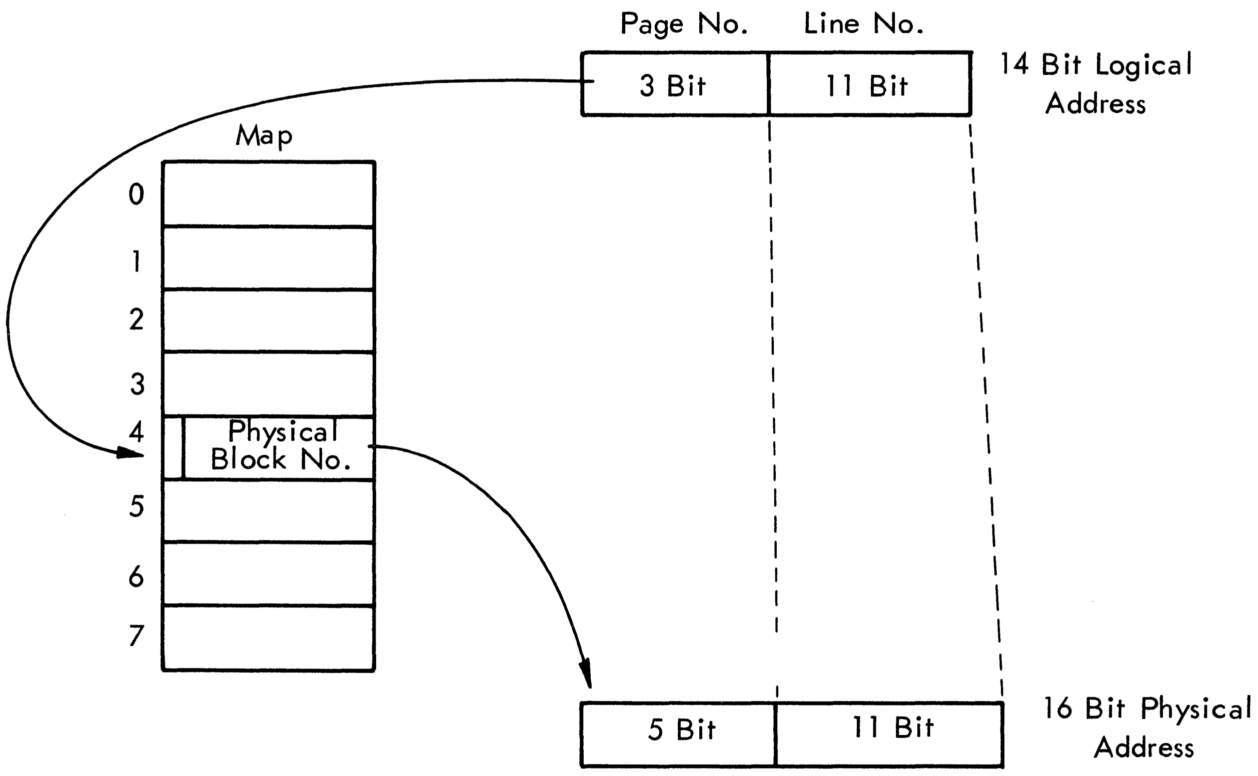


Figure 4. LOGICAL TO PHYSICAL ADDRESS MAPPING IN THE SDS-940

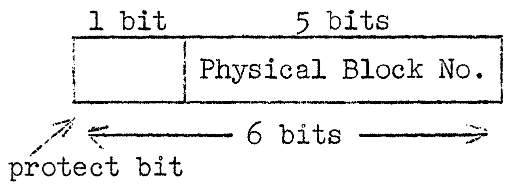


Figure 5. MAP REGISTER IN THE SDS-940

Five bits contain the physical block number and 1 bit is for memory protection to be discussed later. The physical address is simply formed by concatenating the physical block number with the line number to form a 16 bit address. With 16 bits,64K of memory can be addressed.

The above hardware mechanism is quite simple, but to work as part of the total system requires additional software tables to be discussed later in this report. The basic idea is that when a user procedure is to be brought into main storage, the software monitor examines the state of main storage and swaps out only as many pages as are required in conjuction with free pages to meet the needs of the incoming process. The monitor then assigns the available physical blocks to the logical pages of the incoming process and swaps its pages into these blocks. The monitor then changes the memory map, which is structured as two 24 bit words. Then after restoring the active registers and program counter to the values which they had when the process was executing when it was terminated, the process is restarted.

The most important general concept introduced above is that of a memory mapping. A map translates the logical address space into the physical address space. In the dynamic relocation techniques, the map is a set of tables in memory or a set of hardware registers. In the static relocation technique the map is a program. In the dynamic relocation method using base registers, the base registers are the map. The page map can be looked at as a way of efficiently implementing multiple base registers. The paging process is completely invisible to the user and compilers, which function as if they were working with one contiguous logical block. The memory fragmentation ability made possible with paging gives greater flexibility in allocating memory and allows the possibility of decreasing the time lost through swapping by being able to maintain more processes or process fragments in memory at a given time. Thus, memory can be more effectively utilized. The concept of a memory map which dynamically converts logical addresses to physical addresses has been an important innovation in machine organization.

The SDS-940 paging mechanism does not allow a straightforward implementation of a virtual memory space. A more general approach to paging is shown in Figure 6. Here the map is a table (page table) in main memory. One page table exists for each job. The physical block number corresponding to a given page is found by a table lookup in this page table. The control bits can be used to indicate whether or not the page resides in memory or on an auxiliary storage device. The page table base register points to the base of the page table for the process currently in control of the machine. The page number from the logical address when added to the contents of the

Figure 6 follows

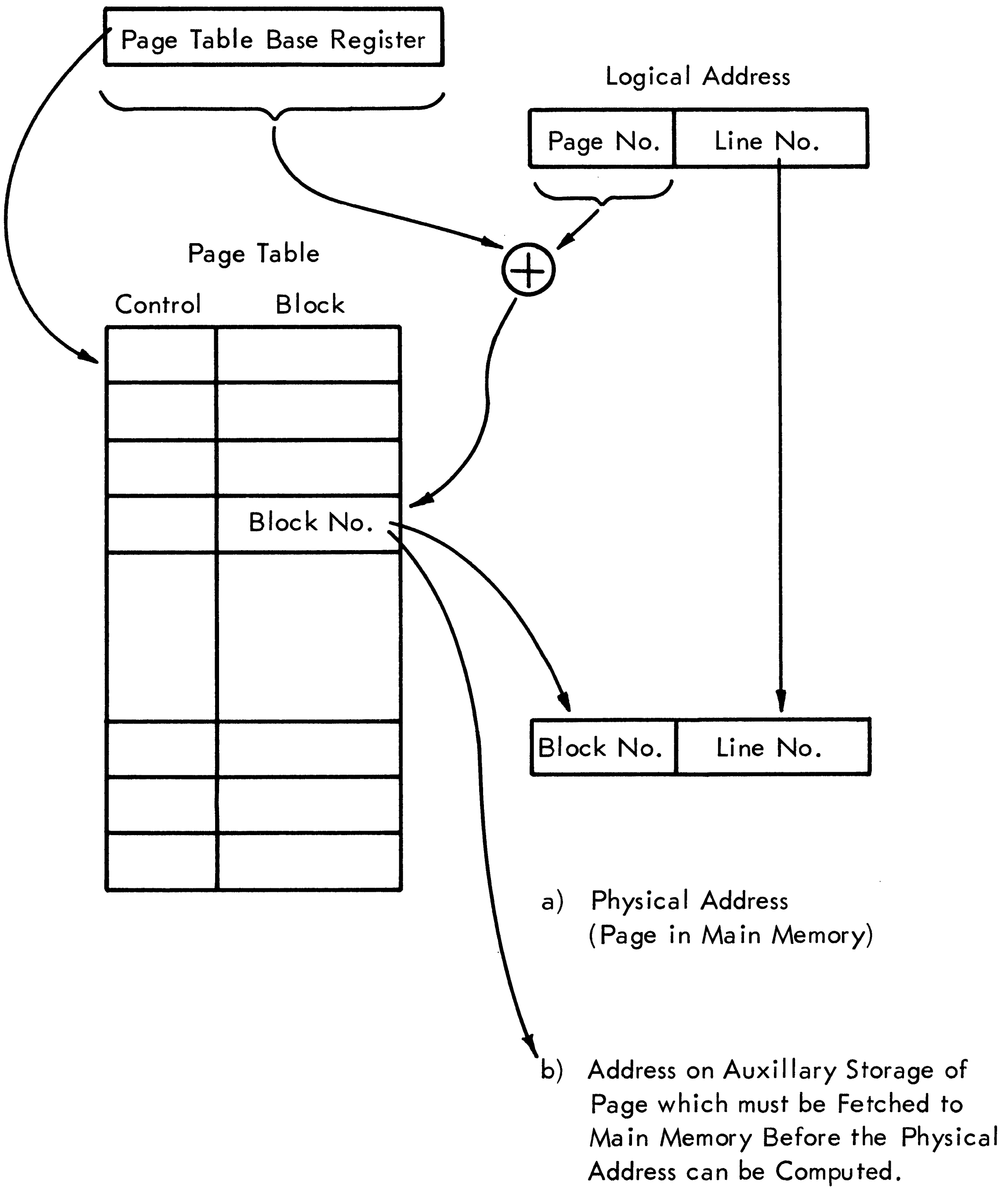

Page Table Base Register
Logical Address
Page No.
Line No.
Page Table
Control
Block
Block No.
Block No.
Line No.
a) Physical Address (Page in Main Memory)
b) Address on Auxillary Storage of Page which must be Fetched to Main Memory Before the Physical Address can be Computed.

Figure 6. PAGING IN A MORE GENERAL FORM ALLOWING A VIRTUAL MEMORY SPACE

249-68
64692

page table base register indicates which word in the page table contains the block number in which the page resides. The number in the "block" portion of the table can indicate an actual starting address for the page in main memory or a location in auxiliary storage at which the page can be found. If the control bits indicate that the latter case holds, then a call to the system can be generated to fetch the page to memory before resuming computation. Using this approach, the logical address space can be larger than the physical address space. The factors limiting the size of the virtual memory are 1) the number of available address field bits which can be generated, and 2) an economical limit for a page table size. The difficulty with the approach just outlined is that all memory references require an additional access time to get the block number from the page table. It would be too expensive to implement the large page table required for a virtual memory of any size in hardware, as well as requiring too much time to change this hardware table each time processes controlling the machine are switched. Therefore, to eliminate the extra memory reference most of the time, a special hardware map using an associative lookup can be implemented.24) Such a map is shown in Figure 7. The map is called an associative map because it is addressed by association or content rather than explicit address. Using the associative map, the page number of the logical address is simultaneously compared against all the page numbers in the map. If the page number is found, the block number is output and the physical address is formed by concatinating the line number with the block number. If no match is found between the page number of the logical address and the page numbers of the map, reference must be made to the page table in main memory. The new page number - block number pair is inserted into the map replacing one of the entries there. The optimum associative map size and the optimum strategy for replacing entries already in the map are design problems of such a paged system.

Using a paging scheme such as outlined above, the entire process would not have to be loaded into main memory at the time computation began. Only those pages initially required could be loaded and as reference was made to pages not in main memory, the page table would indicate this fact by generating a call to the supervisor. The supervisor would then bring in the page. The pros and cons of this "demand paging" approach are discussed in the section on swapping.

The sharing of programs and data is an important requirement for a time-sharing system, as mentioned earlier. There are many ways that sharing can be implemented. One way would require each job to obtain a separate copy of the procedure or data structure to be shared. Obtaining separate copies increases the memory requirement and leads to increased swapping activity. Therefore, it is desirable to share frequently used procedures and data in such a way that only one copy is required in core. Figure 8 illustrates how such sharing is accomplished in a paged system. In Figure 8a, two source programs A and B share a compiler C. The compiler's pages are labeled C_1, C_2 and C_3. Note that entries for the compiler's pages must be placed in the map of each job which uses it, although only one copy of the compiler exists in physical memory. Note further, that the compiler's entries must be in the same relative positions in each map. Similarly the source programs SA and SB

Figures 7 and 8 follow

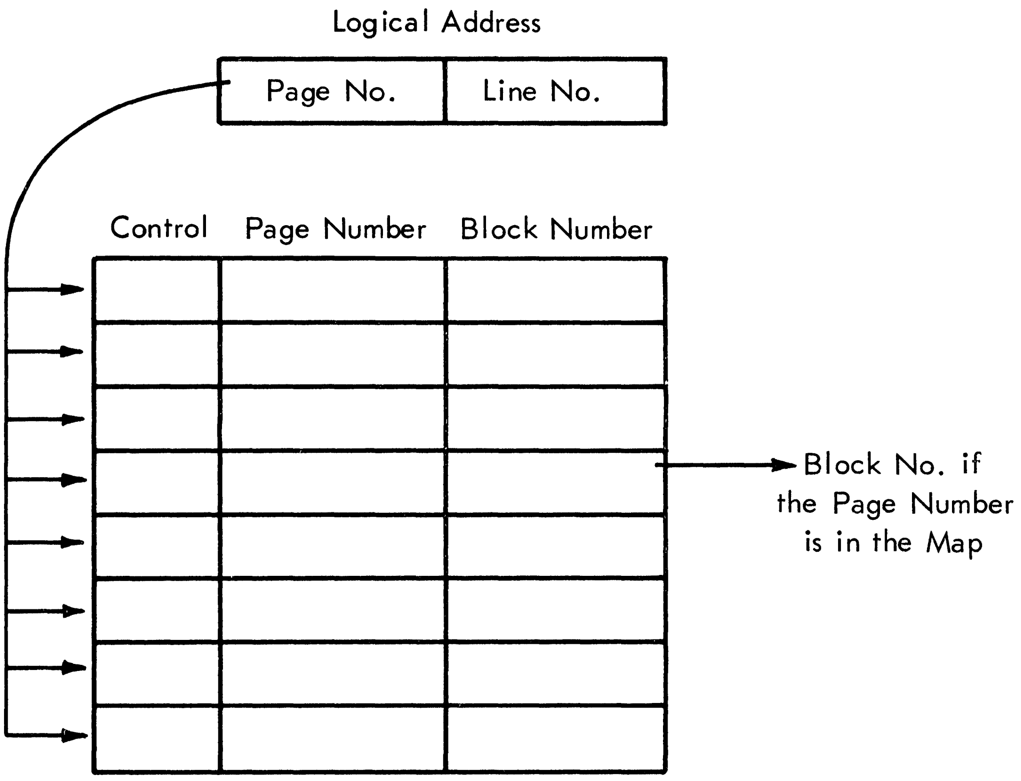


Figure 7. AN ASSOCIATIVE MAP

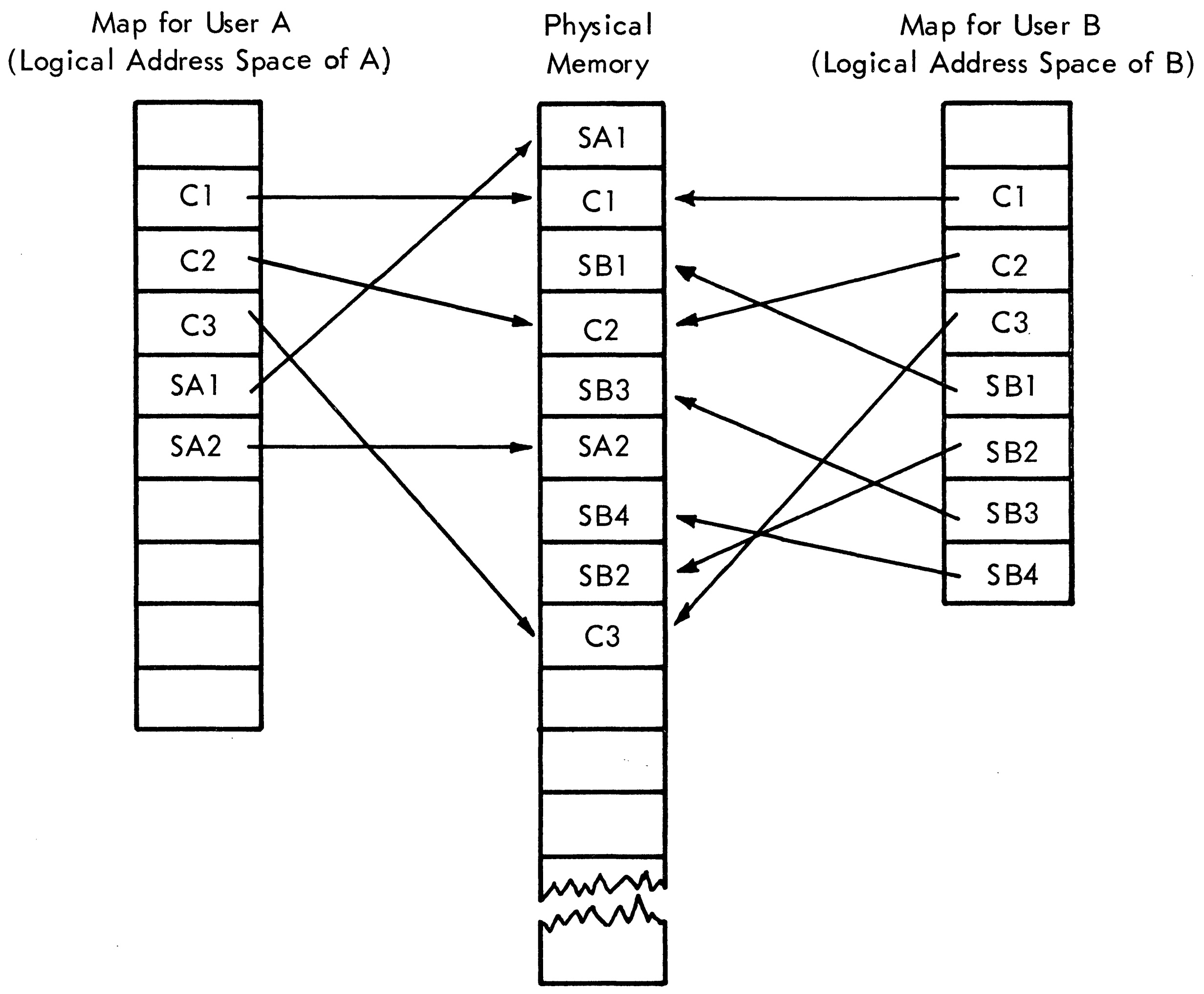


Figure 8a. A COMPILER BEING SHARED IN A PAGED MEMORY

249-68
64692

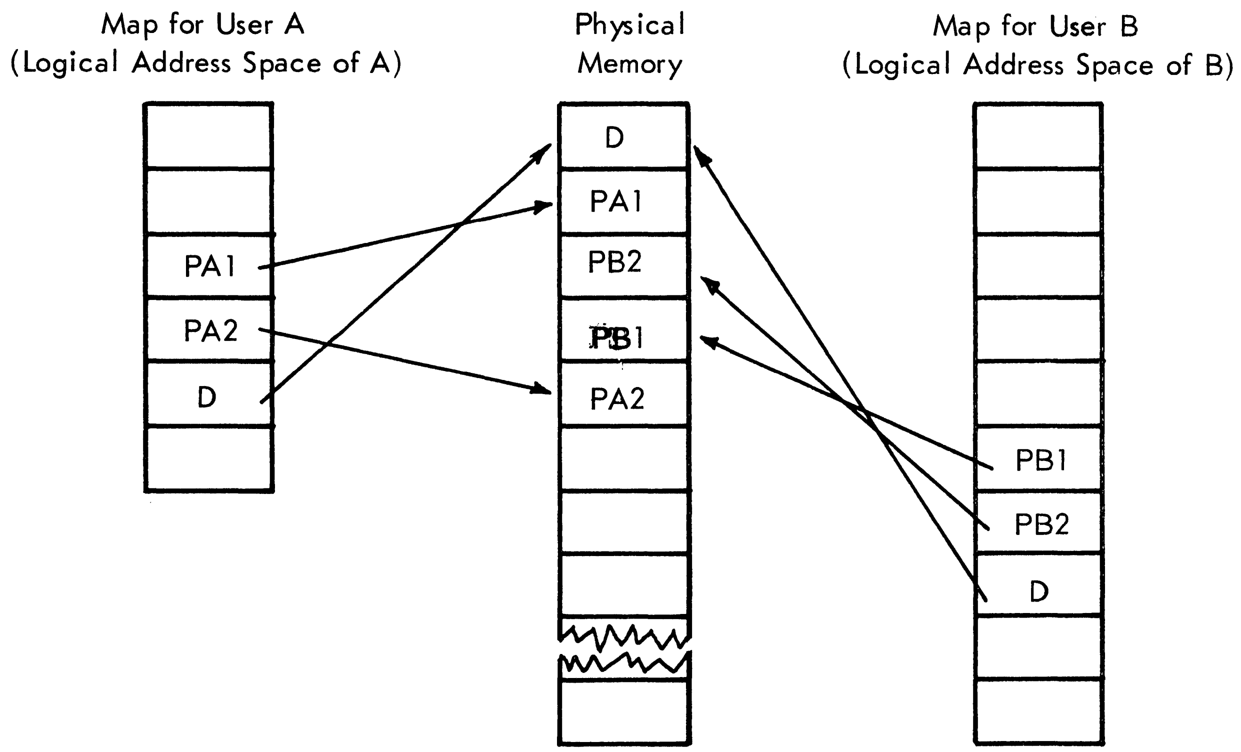


Figure 8b. SHARING OF DATA IN A PAGED MEMORY

must have consecutive entries starting at the same relative position in the two maps. The above requirement results because at the time the compiler is initially loaded into the system its memory references are bound to fixed areas of the logical address space. When the compiler is loaded it is in effect loaded into logical space and the address field contains a page number and a line within the page number. The page number is the index to the entry in the map which contains the base location of the block of memory containing the page. Every time the compiler is executing it is referencing the same relative location in a map. Therefore, every process using the compiler must contain entries in its map for the compiler and in the correct positions which were determined at the time the compiler was loaded into the system. The initial loading could place the compiler anywhere in logical space. This concept of loading as a binding process, to positions in physical space in the case of static relocation, and to positions in logical space in the case of systems using base registers or paging, is a very important concept which should be grasped before proceeding.

An alternative to placing the compilers pages in the same relative location in each users logical space is to have a separate map for the compiler. This map would be invoked when the compiler was called. If the compiler is shared, the difficulty arises as to how to handle the logical placement of the users source code in the compiler's map. Because each source code program being compiled must be stored starting in the same position in the compiler's logical space, the compiler's map would have to be modified each time control is switched to a different job. Another alternative is to have some way of indicating that certain memory references are to be made through the compiler's map and others (those referring to the source code) are to be made through the user's map. The above alternatives introduce extra complexity and are of little value.

Let us now consider the problem of two procedures P_A and P_B sharing data. This case is illustrated in Figure 8b. To share data the logical page "D", representing the data, can be in different locations in each map. The above situation holds true as long as the data itself contains no addresses. For example, if indirect addressing through D back to P_A and P_B could take place, then P_A and P_B would have to reside in the same relative locations in logical space. This location would be determined at the time the data was created. If indirect references through D to itself were to be allowed, then P_A and P_B could occupy different relative locations, but D would have to be placed in the same relative location in each map.

The above discussion shows how programs and data can be shared if certain conventions are followed. What are the implications of these conventions? The implications are:

1) space must be taken in the page table of each job using a shared data structure or a shared program
2) no job can simultaneously use two or more shared routines or data structures which occupy the same position in logical space.

The latter problem can be avoided if the logical space is large enough and the bookkeeping of the installation careful enough inorder to insure that all shared programs and their data which could be used concurrently are loaded into the system in different positions in logical space. This solution would require large page tables (maps). Another way around the problem is to use hardware base registers as part of the address formation process before going through the paging schemes above. This latter approach is impractical because it would require multiple base registers, very careful bookkeeping and the establishment of conventions on base register usage. Use of base registers to perform relocation in logical space is a step toward segmentation. Therefore, rather than mix hardware mapping techniques, a better design procedure is to step back and examine the problems to be solved and then to produce if possible a integrated approach to their solution. The introduction of segmentation, described in the next section, is an attempt to produce such an integrated approach.

Besides restricting the full generality of program and data sharing, paging techniques do not aid the programmer in dealing with memory allocation for data bases such as lists, symbol tables and push down stacks which can grow and contract during execution. The programmer must explicitly plan for such situations so that the map can be set properly. In the SDS-940 system this problem is reduced somewhat through system calls, which allow the process the ability to interact with the system to change its map. Thus, the process has considerable power to control the contents of its logical address space. This latter ability, however, is more in the line of facilitating overlays than in fundamentally altering the structure of the logical space one can obtain with paging.

<u>The structure of the logical space that one obtains with paging is identical to that obtained using base registers</u>. The structure obtained in either case is that of a large contiguous array which is dynamically relocatable. To the user there is no difference in the way he would program using either approach. What is gained in using paging over using base registers is a method which possibly may yield more effective memory utilization through memory fragmentation. One also obtains a practical method for the implemention of a large virtual memory. More flexibility may also be given the system programmer in his design of program and data sharing conventions. The apparent disadvantages of paging over the use of base registers are: 1) paging requires a hardware memory map to be efficient, 2) the software implementation may be more complex if the full advantage of paging is to be utilized, and 3) the resident software system will probably be larger, thus requiring more main memory. Given the trend toward decreasing hardware costs, the extra cost of a hardware map is a very small part of that of the total system. The software complexity and system memory required to utilize paging is not great as seen in the discussion of the SDS-940 system in the second half of this paper. Better utilization of memory enables more processes to reside concurrently in main memory, which increases the possibility of performing useful computation in one process while the swapping of another takes place. Faster auxiliary storage devices can minimize this advantage, however. The small extra cost of paging hardware

would seem worthwhile considering the extra flexibility offered the system programmer to experiment with CPU, memory allocation, and swapping algorethms.

Segmentation. The concept of segmentation, inspite of the many words written about it, is not yet widely understood. The reasons for the confusion about the concept are several: 1) the motivations for its development have not been made clear, 2) varying versions of the concept exist in different machines, for example, segmentation as it exists in the IBM 360/67 system has important differences from segmentation as it exists in the GE-645 system; 3) the distinctions between a logical address space and a physical address space are not widely understood; and 4) because many of the implementation mechanisms are similar between paging systems and segmentation systems these two distinctly different concepts tend to be blurred together. In this section, we try to give a clear explanation of segmentation as we understand its most general form, indicate some of the differences between various versions of the concept, emphasize again the difference between logical and physical address space, make clear the distinction between paging and segmentation, review the motivation for the development of segmentation, and finally question some of the basic assumptions underlying present implementations. Segmentation is looked at from a slightly different point of view in the discussion on file systems later in this paper.

The discussion of dynamic relocation to this point showed that there were certain problems with physical memory allocation inherent in systems using base registers; solutions to these physical memory allocation problems were offered by the introduction of paging. Paged systems inturn were shown to have some problems with logical memory allocation which exist if programs and data are to be shared with full generality and if data structures are to be allowed to grow and contract at will without explicit allocation planning by the programmer. Segmentation offers solutions to these problems. The above problems could be solved in a paged system if a very large logical memory space could be created. Then all shared procedures could occupy a unique position in this space and data structures could occupy positions far enough apart from each other and procedures so that they could grow at will. The above solution could be achieved in a paged system by having enough address bits and a large logical to physical memory map (page table) for each job. Such a solution is impractical because the map for each job, to assure that there would be no conflicts of positioning in logical space,[a] would have to be very large. This map would contain large gaps if one wanted to allow data structures to grow at will. Gaps would also arise because concurrently shared procedures could occupy widely separated positions in logical space. Further, careful bookkeeping would be required to assure that no two shared procedures which might be used concurrently occupied the same position in logical space. It is imperative for the reader to understand the above argument if the motivation for segmentation is to be fully grasped. If at this point it is not clear we suggest a review of the

a) Conflicts of position in logical space are sometimes referred to as name conflicts because the position in space of an element is also its name. We prefer the use of the term "position conflict" because it illustrates more graphically the nature of the problem.

discussion on paging where examples are given of the importance of the position in logical space of shared procedures and data.

In summary then, the difficulty of using paging for sharing procedures and data in full generality and allowing for data structure growth results 1) because of the large, possibly sparsely filled, map required and 2) because careful bookkeeping by the installation and system would have to be maintained a) to be certain procedures being used concurrently do not occupy the same position in logical space (i.e., have the same name) and b) to position shared data properly which have address references. Again many of the papers in the literature may be clearer if "name" is translated to "position in logical space".[15][54]

The technique of segmentation was developed as a way to more easily manage a very large logical space. A large logical space is required to solve the problem discussed above. Segmentation is a technique for dealing with logical space allocation and should be kept separate in ones mind from paging which is a concept for dealing with physical space allocation. The fact that some implementations of the segmentation concept also use paging for physical memory allocation should not be allowed to confuse the distinction between the concepts.

A segment is an ordered set of data elements having a name. A particular data element is referenced by a symbolic segment name, symbolic data item name within the segment, $<S>/[\alpha]$.

Segmentation is often referred to as a two dimensional logical address space. A paging system is not considered two dimensional, even though the address has a page number and a line number pair because these conventions are invisible to the user. To be general one could consider base-register and paged systems as segmented systems allowing one segment and thus the segment name is implicit. In a general segmented system, the user programs his addresses using a pair notation, $<S>/[\alpha]$. The notation $<S>$ indicates a symbolic segment name S, and the notation $[\alpha]$ indicates a symbolic element within segment name α. A segment is a self contained logical entity of related information such as a procedure, data array, symbol table or push-down stack. There is no logical restriction on the length of a segment, although in any given implementation there will be an upper bound on segment length.[18][19][36][52] Segments can grow and contract as needed. A segmented system provides a logical space of non homogeneous units called segments. A base register or paged system provides a logical space of one homogeneous unit. Each segment is a separate logical entity. The problems to be solved and the parameters to be determined in developing an implementation of the segmentation concept are 1) developing a method for mapping symbolic address pairs $<S>/[\alpha]$ to physical locations, 2) determining the number of segments to be allowed and their size, 3) developing a method of linking segments together, 4) developing a method for sharing of segments, and 5) developing a method for protection of segments. A variety of solutions to these problems have been implemented in hardware or suggested in the literature.[2][11][15][18][19][20][29][32][36][44][54][22] We discuss in some detail the implementation of one segmentation system, the Multics (GE-645) system and then briefly discuss some variations found in other proposals or implemented systems.

The logical address space or virtual memory of the GE-645 system can contain up to 2^{14} segments, each of which can be up to 2^{18} 36-bit words in length. The GE-645 CPU has an accumlater register, multiplier/quotient register, eight index registers and a program counter which perform the normal function of such registers.

Additional registers are required to implement the segmentation concept. These registers are, a descriptor base register, a procedure base register, and four base pair registers. The purpose of these extra registers is briefly introduced here and their function expanded on in more detailed discussion to follow. The descriptor base register points to the location in memory of the segment descriptor table, discussed below. The procedure base register contains the segment number (name) of the procedure being executed. Each of the four base pair registers constains a segment name/item name pair and has a specific function as described below. The most general form of an address is a symbolic segment name/item name pair. Eventually this pair of symbolic names must be converted to a physical memory location. The discussion to follow indicates how this mapping takes place for the more common types of memory references.

Associated with each process there is a segment descriptor table (often called a descriptor segment) which is itself a segment. This table contains one word called a segment descriptor for each segment known to the computation. Each segment descriptor contains the base address in main memory for the segment (we defer the introduction of how paging is used with segmentation until the main concepts have been introduced) or its location in auxiliary memory. The segment descriptor also contains control bits used for memory protection and other purposes not of interest here. The segment descriptor table can be thought of as an array of base registers. Before going on to look at implementation details let us see how segmentation solves the problem of handling a large logical address space and provides for dynamic relocation within this space. To see how segmentation solves the problem of sharing of procedures and data in full generality, we introduce later in this section one additional major concept, that of the linkage segment.

A large logical space within which procedures and data can be relocated is obtained by breaking up a computation into self contained procedure and data units called segments. The use of the segment descriptor table allows these units to be relocated in logical space. Another way of stating it is that the solution is obtained through the use of two dimensional addressing $<S>/[\alpha]$. All addresses in a program or data are relative to the start of a segment. All segments can be positioned in logical space so that there is no conflict of position by proper relocation through the use of the segment descriptor table. The segment descriptor table is in effect an array of base registers as shown in Figure 9a. The segment name $<S>$, or as we shall see a transformation of $<S>$, indicates the proper segment descriptor (base register) to combine with the location within segment address $[\alpha]$ to yield an address in a one dimensional logical space as shown in Figure 10b. This logical address is then converted directly to a physical address or is converted through a map.

The segment descriptor table as an array of base registers is a satisfactory solution to the relocation problem in logical space whereas a set of hardware base registers was not a satisfactory solution because 1) the same hardware base register had to be used for a procedure or data structure base by every computation sharing that procedure or data structure 2) the same hardware base register had to be used for a procedure or data structure base each time the procedure or data structure is used by the same computation. These restricutions resulted because the address of the base register was bound into the instruction word at the time of loading. A segment descriptor for a given segment <S> can occupy different positions at different times in the same computations segment table. The mechanisms which make this possible are outlined below. The above discussion is not to imply that it is theoretically impossible to perform equivalent relocation in logical space with physical base registers, but to imply that as intricate as some of the segmentation implementation details become, the implementation of an equivalent user invisible function using physical base registers would be much more involved and thus practically infeasible. Segmentation in effect offers a technique for dynamic allocation of base registers (segment descriptors); the set of base registers required for a given computation is called the segment descriptor table.

When a segment is loaded into the system it is not bound in logical space because all addresses are relative to the beginning of some segment. Only at the time a segment is made known to a process by creating a segment descriptor does binding of the segments location take place. This binding is caused by the transformation of the segment name into an index in the segment descriptor table of the created segment descriptor. Different processes using the same segment and even the same process on different occassions can have the segment descriptor in different places in a segment descriptor table. One can think of the position in the segment descriptor table as a position in logical space. Because this same segment can have a descriptor at different positions in the descriptor tables of different jobs or in different positions in the descriptor table of the same job on different occassions, one has a dynamically relocatable logical space. Base registers and paging gave a dynamically relocatable physical space.

To enable a segmentation system to work, the segments must be properly linked together. This linkage must be implemented in such a way as to allow the realization of sharing of segments with full generality. This linkage problem is handled by associating with each segment, making external references, another segment called the linkage segment. All intersegment references are handled by indirect addressing through a linkage segment. The details of how the linkage segment is set up and used are given later in this section.

Segmentation has been developed to achieve full generality in the allocation of logical space. If segmentation is to be usefully criticized this must be performed at two levels: 1) at the level of the fundamental assumption; namely, full generality in manipulating logical space is truly required, and 2) at the level of a particular implementation; namely, is the implementation being examined the most effective way to achieve the goal. As any system designer knows, there are many cost-generality trade offs in the implementation of a set of concepts. One can question a particular segmentation implementation as to whether the appropriate trade offs were made. The payoff to achieve a given level of generality measured in numbers of applications or value of applications requiring this level of generality must be weighed against the costs as measured in hardware expense, software development expense and machine running expense associated with achieving this level of generality.

For example, the decision made in the development of the Multics system, was to achieve full generality without cost being considered seriously. As a research and development decision, this is clearly a valid one and one which will yield much information about how and at what cost full generality of allocation of logical space can be achieved. If GE decides to implement a version of the system as a commercial product, then careful questioning is required of the cost-generality trade offs made against an installations expected usage of the levels of generality offered with the system.

Examples of the type of question one could ask are:

1) How frequently are multiple users going to concurrently share <u>two</u> or more procedures in a given computation, thus introducing the possibility of position conflicts of these procedures in logical space?

2) Is this frequency of use great enough and the memory requirements of the procedure to be shared large enough so that the price associated with obtaining full generality of sharing gives an adequate return on the investment; or would it not be better to make a trade off and share one copy of very frequently used procedures such as standard compilers, but allow sharing through the use of multiple copies of other classes of procedure? That is, one can allow sharing without trying to avoid multiple copies for all such procedures or data structures.

3) Is requiring the use of overlays for large programs really a serious problem, which cannot be more effectively handled through machine aids other than that of providing a large virtual memory?

We do not believe that the above type of questions have been adequately dealt with by proponents of segmentation. If the costs associated with implementing segmentation in full generality were low, then such questions would be irrelevent. However, as we see below where details of an implementation are discussed, the present state of the art requires that a substantial price must be paid for a segmentation implementation both in development, hardware, and system overhead during operation. Our conclusion at this point in time is that a segmentation implementation in full generality makes a valuable research project, but that the results must be carefully analyzed before the concept is generally accepted as a requirement for large scale time sharing.

We now outline some of the details of the segmentation implementation in the Multics system. Software and hardware are so closely interlinked that concepts in both areas are introduced. For the initial discussion, we assume all segments required by a computation are "known" and thus have entries in the segment descriptor table. We outline later the steps required to make a segment known. The relative location within the segment table of the descriptor for a segment with symbolic name <S> is called the segment number, S#, of segment <S>. The segment <S> may be shared by several computations and have a different segment number in each; that is, the descriptors for segment <S> may appear in different relative locations within the segment descriptor table for each computation. Figure 9a shows a process in memory with segments A, B, C, D and T. Note that the contents of the descriptor base register points to the base of the segment descriptor table (also called the descriptor segment). Figure 9b shows two processes resident in memory sharing segments A and T. Each process has a different location in its segment descriptor table for the segment descriptors for A and T. Therefore, each process refers to segments A and T with different segment numbers. The descriptor base register's contents are changed each time a different process is given control of the machine so that it points to the beginning of the correct segment descriptor table.

To make a memory reference, two numbers are necessary: 1) a segment number is required to make reference to the correct segment descriptor in the segment descriptor table, 2) a location within the segment. The pair segment number/location number within the segment is called a generalized address. The notation S#/α is used, where S# is the segment number and α is the location within <S> of the symbolic element name [α]. We now outline some of the ways in which a generalized address is formed. First consider instruction fetches. The segment number of the executing segment is contained in the procedure base register. The location within the segment is contained in the program counter. If the procedure segment is executing a sequence of instructions that lie entirely within the segment, the contents of the procedure base register remain unchanged.

There are two types of GE-645 instruction words, type 0 and type 1. Type 0 instructions obtain their operands from within the executing segment. The segment number is obtained from the procedure base register and the location within the segment is obtained through the normal process of address modification, possibly involving indexing and indirect addressing within the executing segment.

The type 1 instructions are used to reference data in a segment outside the one executing and form a generalized address as shown in Figure 10a. The segment tag is used to point to one of the four base pair registers. Part of this register contains the segment number (how the proper segment number got there is outlined later) and the remainder contains a location base within the segment which when combined with the address field and possible indexing yields the location number within the segment. Figure 10b indicates how the actual operand address in logical space is obtained from the generalized address. The segment number is added to the number in the descriptor base register to locate the correct segment

Figures 9 and 10 follow

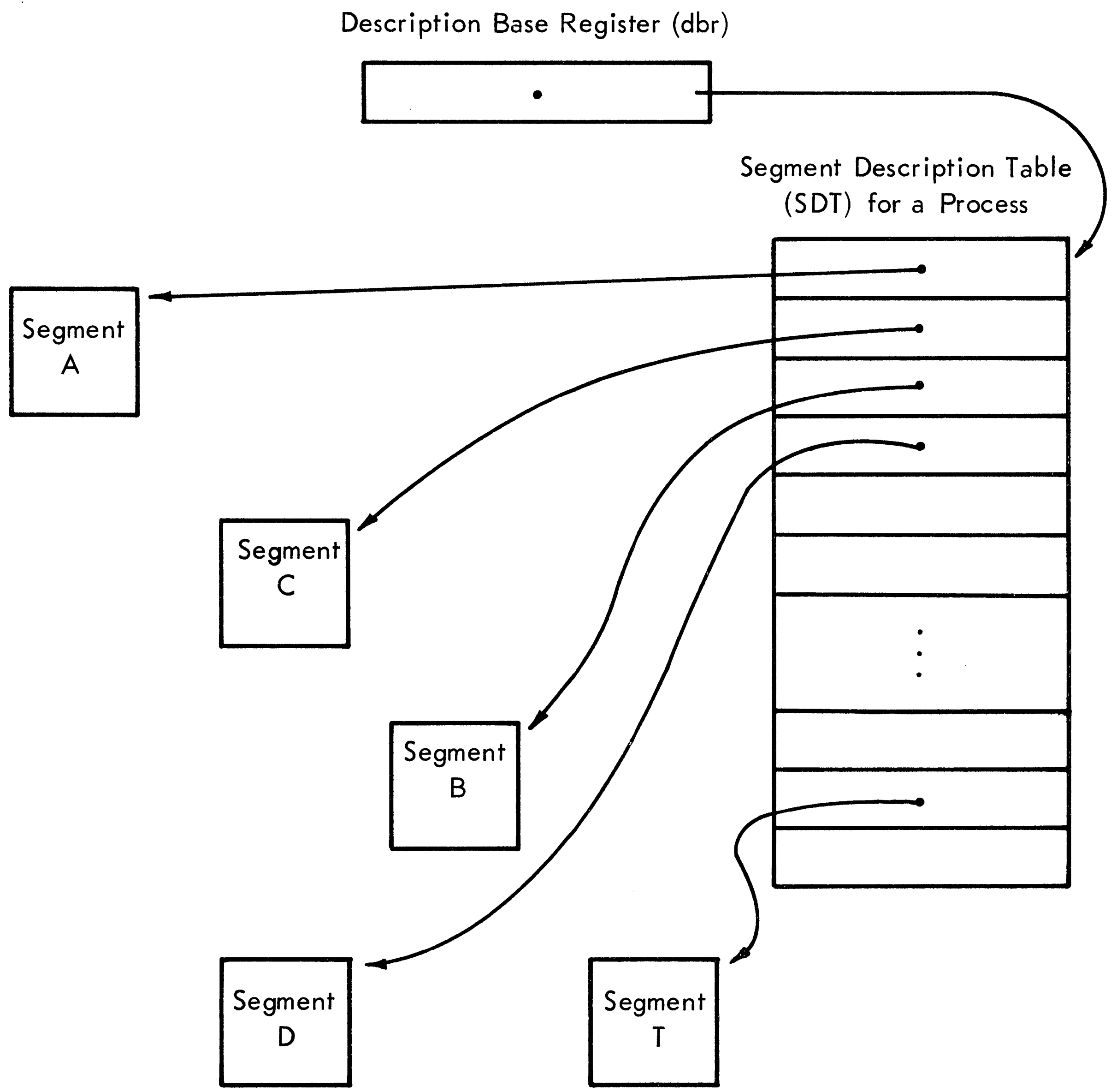

Description Base Register (dbr)
Segment Description Table (SDT) for a Process
Segment A
Segment C
Segment B
Segment D
Segment T

Figure 9a. A SEGMENTED PROCESS RESIDENT IN MEMORY

249-68
64692

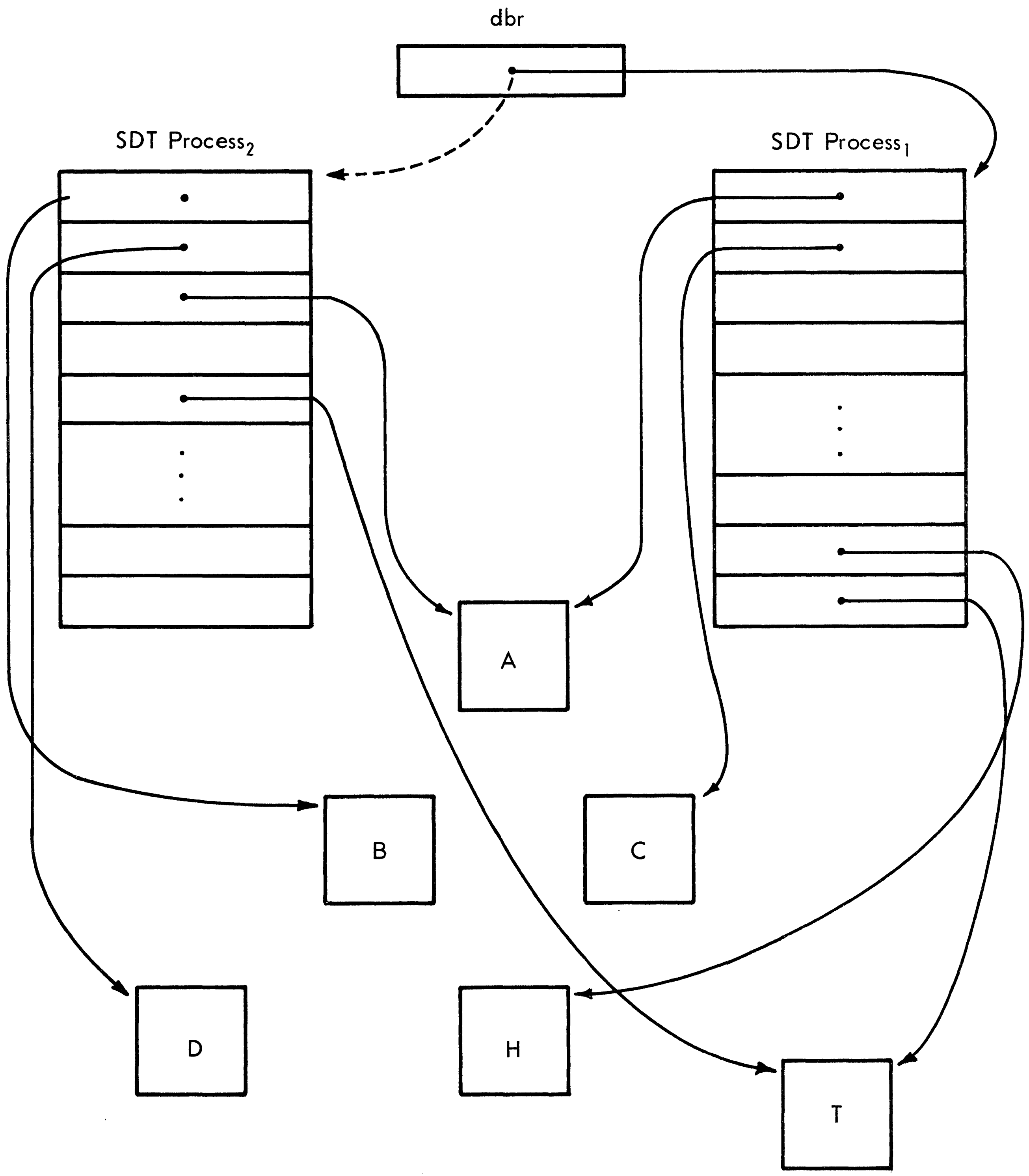
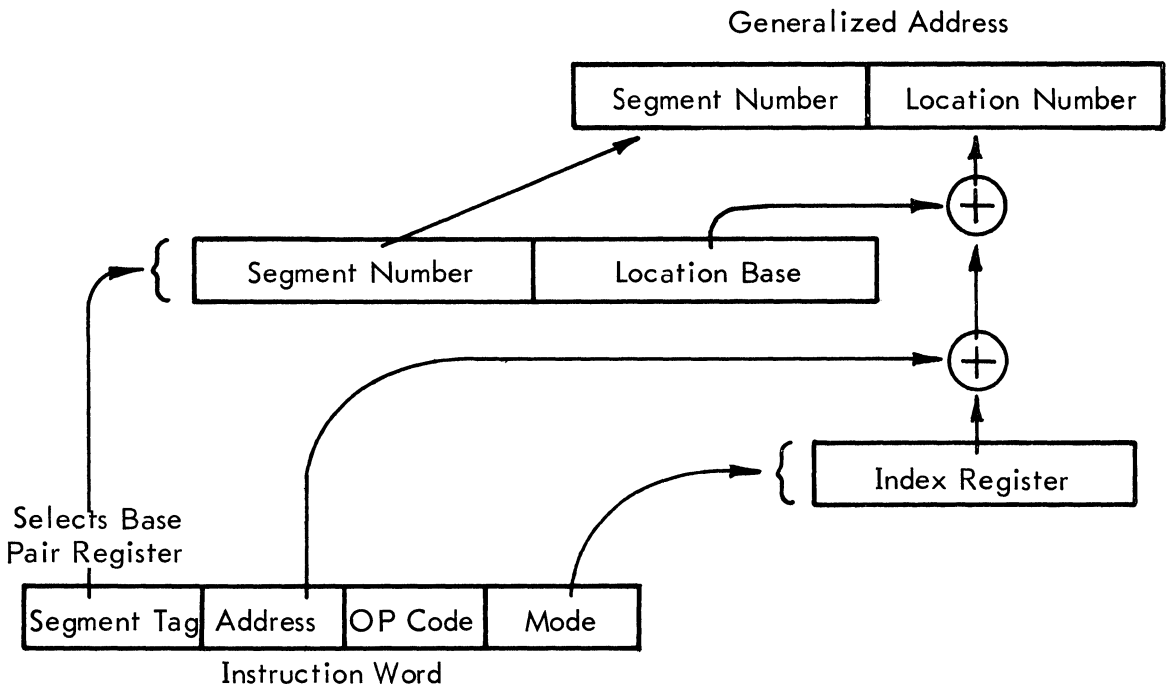
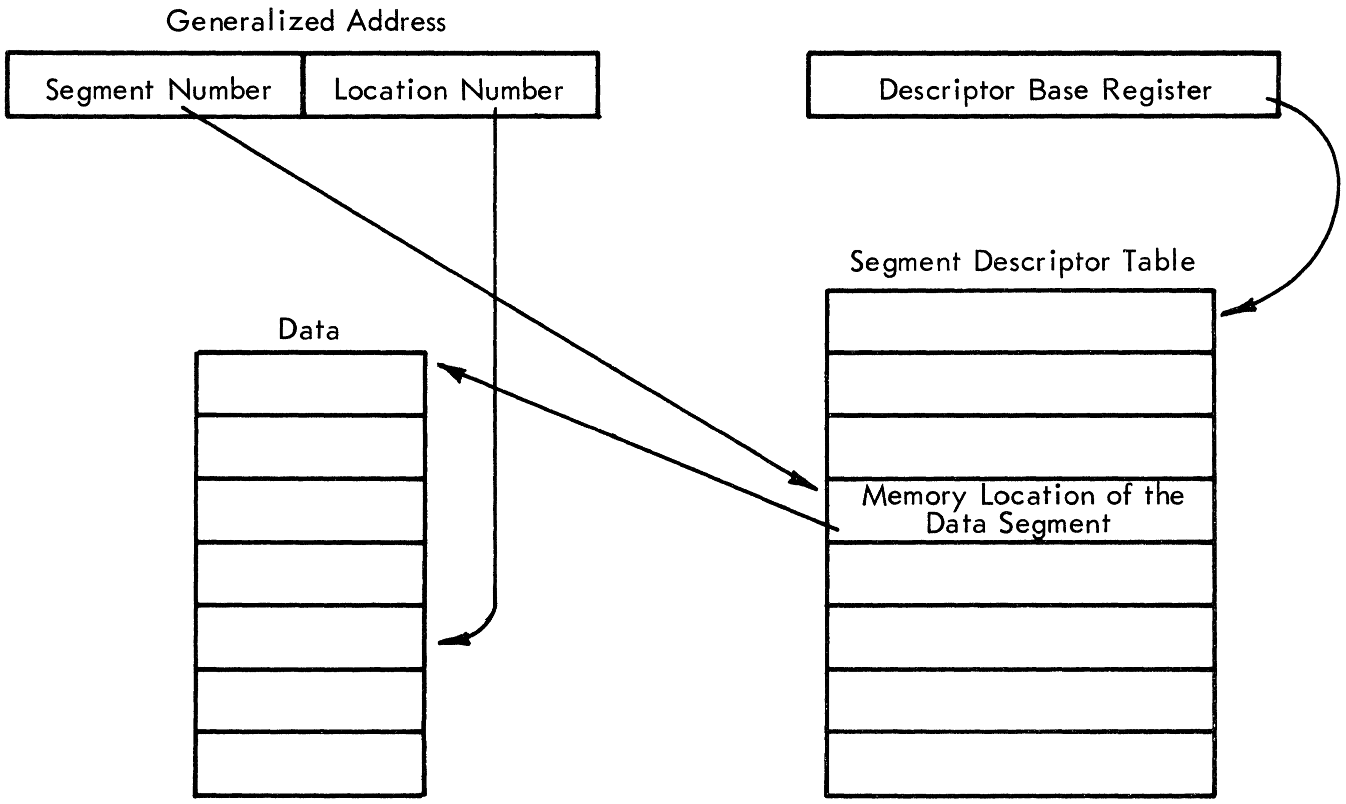


Figure 9b. TWO PROCESSES RESIDENT IN MEMORY SHARING SEGMENTS

249-68
64692



a. Computation of the Generalized Address in the GE 645



b. Calculation of Physical Location

Figure 10.

249-68
64692

descriptor in the segment descriptor table which in turn points to the base of the segment. The location number is added to the address in the segment descriptor to form the location required.

The above mechanism provides the solution to the problem of allowing data structures to grow and contract at will without the programmer having to worry about memory allocation. If the data structure is a segment, it can grow to any size up to the maximum segment size imposed by the implementation or shrink at will. Only as much physical memory is used as presently required by the data structure. In the actual implementation, the segments are paged and the pointer in the segment descriptor really points to the base of a page table containing page descriptors for the pages of the segment. A page descriptor contains the address in main or auxiliary storage where the given page is stored. The location number is broken into two parts, a page number and a line within the page. The actual physical address is obtained via a double table look up as shown in Figure 11.

If the scheme were left as above it would be unsatisfactory because of the time required to compute each address. To get around this problem a small associative memory[a)] is incorporated in the processor as a memory map.[19)] The key used to address this memory is the combined page and segment number. If the double indexing described above had recently been performed, the page and segment number along with the actual page address would have been inserted in the associative memory. Then when this page-segment number is used again, the page address is retrieved directly and the lookup in the segment and page descriptor tables is avoided.

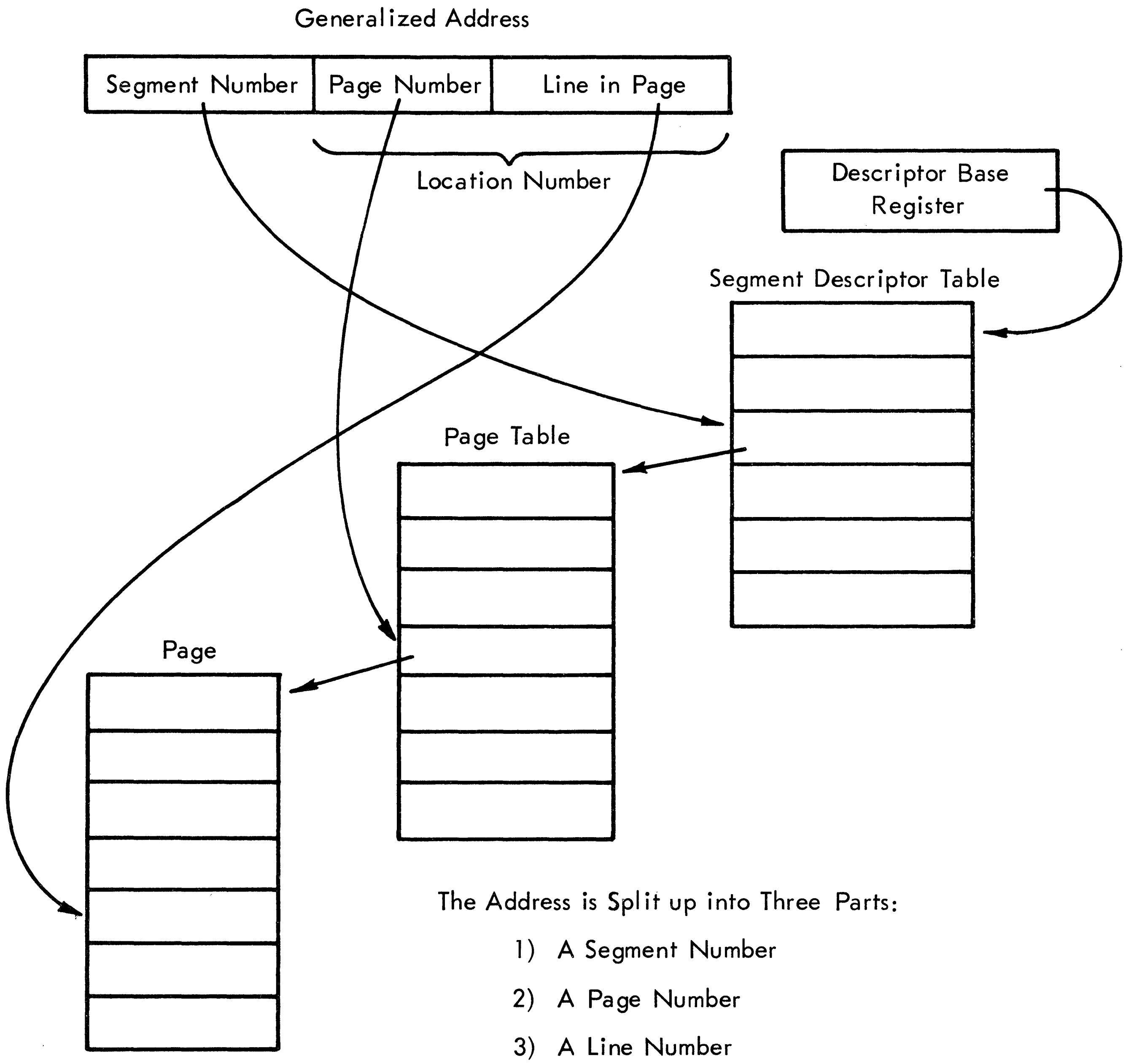
The associative memory is restricted in size and therefore hardware algorethms have to be implemented which determine which page-segment numbers to replace when it is full and pages not in the associative memory are addressed.[18)19)]

Let us now back track and discuss indirect addressing through segments outside the one executing. Indirect addressing is heavily used for the segment linking scheme to be discussed below. Whenever an indirect word address is even,the GE-645 fetches a pair of words. The location of the indirect pair is obtained as in Figures 10 and 11. This pair, based on the setting of certain bits, can be interpreted in three ways: 1) fetch the operand from the same segment as the indirect word, 2) fetch the operand from the segment which has the segment number contained in the first word of the pair at the location within the segment given by the second word, or 3) fetch the operand from the segment given by the base-pair register indicated in the first word and location within the segment indicated in the second word.

Indirect addressing and indexing can proceed to any level and thus pass through several segments. Indirect addressing may go several levels in one segment before reaching the operand or before passing to another segment.

a) An associative memory was described earlier for a paged machine in which the key was the page number. The concept is identical here except that a longer key is required containing the segment number and the page number.

Figure 11 follows



The Address is Split up into Three Parts:

1) A Segment Number
2) A Page Number
3) A Line Number

Figure 11. CALCULATION OF PHYSICAL LOCATION USING PAGING

249-68
64692

In effect indirect addressing on the 645 is identical conceptually with indirect addressing on conventional machines except that one obtains a segment number as well as location number at each level. The latter can be modified by indexing.

With the above outline of addressing on the GE-645, we can now discuss the techniques used to link segments. In a system using static relocation, dynamic relocation using base-registers or paging, all procedures and data are linked together at the time of loading into physical or logical space or at the time the map is set up when the process is initialized. The philosophy of the Multics system is that linking is only to take place at the time a process reaches the point where a given segment is actually required. Prior to this point, the required segment is simply a file stored somewhere within the auxiliary memory hierarchy. The supervisor segment is invoked when the segment is required. The supervisor fetches the segment from auxiliary memory, records the segment as known in the proper tables of the calling process and then returns control to the calling process. The above, in the most general terms, is the <u>linking</u> process. The linking process must satisfy three goals: 1) enable segments to be swapped, 2) enable segments to be shared with full generality, and 3) enable segments to be loaded as needed. Let us consider the latter case first.

The motivation for the load as needed requirements is the desire to minimize the memory required by a computation and thus to minimize swapping. A segment is needed when the first reference is made to it. For example, consider the code

```
                    <A>
                     .
                     .
                     .
                     .
     load accumulater <S>/[place 1]
     store accumulater <S>/[place 2]
                     .
                     .
                     .
                     .
```

which is executing in a segment <A>. The two dimensional symbolic addresses refer to a segment different from <A>. The outline of the process for linking segments <A> and <S> is the following. Associated with every segment is another segment called the linkage segment which, for example, for segment <A> we call <link A>. This segment is created by the assembler or compiler. There is clearly not enough room in the address field of an instruction for a two dimensional symbolic name and so what the compiler creates is a pair of indirect words in the linkage segment associated with each out-of-segment reference. During the indirect addressing operation a certain code in the indirect word causes a transfer to take place to the supervisor if the referenced segment address is in symbolic form. The indirect pair actually contain pointers to the symbol strings.

When a symbolic segment name is ecnountered, the supervisor searches the known segment table for a symbolic segment name the same as the one in the instruction. If the name is found, the associated segment number is placed in the first indirect word. The symbolic location within segment name, for example, [place 1] above, must now be converted to a relative location within the segment. Each segment carries with it a symbol table of all symbols which may be referenced by external segments. Thus once the segment number is known, the segment can be located and the segment symbol table searched to find the location of [place 1], for example. This number replaces the contents of the second indirect word associated with the given instruction in the linkage segment. The code bits of the indirect pair are modified to indicate that now a generalized address exists there and the memory reference can be allowed to proceed as discussed earlier for indirect addresses.

If the search of the known segment table fails to find a segment name equal to the one given in the instruction, then supervisor routines called the "file system" are invoked to find the segment. When the segment is found, using other tables called the file directories, it is loaded into main memory. A segment descriptor for the segment is set up.

The relative location of this descriptor in the segment descriptor table becomes the segment number. A symbolic segment name/segment number pair are added to the known segment table and then addressing proceeds as just described above.

The above scheme is completely general. The segment number can vary from computing session to computing session or be different for different processes sharing the segment. Just because a segment is known by one process does not mean it will be known by another process. Part of the job of the file system is to determine if a segment had already been loaded previously because another computation was using it. The segment descriptor of the second process is set up to point to the base of the present location of the segment, thus requiring only one copy to be in memory.

The price paid for the above generality is:

1) The memory space for the known segment table, linkage segments, and the symbol table carried with each segment.

2) The time spent in the supervisor changing two dimensional symbolic addresses into two dimensional generalized addresses the first time instructions referring to external segments are referenced.

3) The time spent referencing external segments with indirect addresses. The alert reader probably still has the question, how does a segment know where its linkage segment is located? The answer is that one of the base-pair registers is assigned by convention to this task.

We are now ready to see what happens when a segment is swapped. The segment may be returned to a different place in memory each time. This problem is simply handled by modifying the contents of the segment descriptor. All the previous linking steps do not need to be repeated because the segment descriptor location in the segment descriptor table is not altered. Thus the segment number remains the same. At a further level of detail one should remember that when we refer to a segment's location, we really mean the location of the page table for the segment. The page table in turn indicates where in physical memory the segment actually resides. We see another price paid for the generality, the time required to change the segment descriptor and page descriptors on each swap.

We are now ready to see how segmentation allows sharing to take place with full generality. The interesting question is, how does the shared procedure, which must be a pure procedure, access the correct data segments for the process which is presently in control? This problem is solved by associating with the shared procedure a linkage segment for each process using it. Or from another point of view each process using a shared segment has a linkage segment for the shared segment. When the shared procedure is initially created a linkage segment is created. A copy of this linkage segment is obtained by the system for each process using the shared segment. These linkage segments are "filled in" as described above to provide the proper coupling. The correct linkage segment is referenced at a given time because the base-pair registers are saved and restored as part of the machine state.

The previous discussion showed how multiple shared procedures can be used simultaneously without conflicts of location in logical space through relocation by use of the segment descriptor table. The solution to the problem of sharing data containing addresses now needs to be pointed out. If the addresses are internal to the data segment, they appear relative to the start of the segment. The starting point of the segment is given by the segment descriptor and thus no conflict of positioning in logical space can occur for this case. If the addresses refer back to a procedure segment then a linkage segment, is set up for each process using the shared data. The linkage segment assures that no conflict can occur in positioning of the data segment or procedure segments using the data.

The linkage segment is thus a key concept in the implementation of a segmented system. The reader may still wonder why use the linkage segment for intersegment references when the addressing mechanism allows intersegment references to take place more directly with the base-pair registers? There are only four base-pair registers. Thus, we would have conflicts in addressing these registers if very great care was not taken to be sure shared procedures being used concurrently did not refer to the same base register. In other words we would have one of the problems we were trying to avoid by using segmentation.

In the Multics system, procedures and data will, in general, occupy different segments. Requiring indirect address references in the inner loop of a matrix multiply routine, for example, would be very costly. This is one of the prices paid to obtain full generality. By use of assembly level coding one can get around this extra memory reference by using a base-pair register, but again this must be done with care or conflicts can arise and is not likely to be undertaken by most users.

The above discussion of the segmentation implementation in the Multics system has left out many details which the reader can obtain from reference 37. Hopefully, however, the general outlines of the scheme are clear and in particular the reader has some feeling for the price paid to achieve full generality.

Let us now examine briefly some other implementations to see a few possible variations of the segmentation concept. The earliest implementation of segmentation, known to us, occurred on the Burroughs B-5000. A process in the B-5000 could contain 1024 segments. Procedure segments may be any length and data segments are limited to 1024 words. Data structures requiring greater space use multiple segments. The segments are not paged and are thus also the unit of physical memory allocation. The table equivalent to the segment descriptor table, in the B-5000 system, is called the "program reference table". No attempt is made to use segmentation in its full generality and therfore a more efficient system is made possible.52)

In a segmentation scheme such as that of Multics where any segment can reference in principle the entire logical address space (all segments), many bits of address information are required in the implementation. Thus, requiring pair-base registers, paired indirect words and so forth. In practice any given segment only needs to communicate with a limited number of segments. A segmentation scheme of Evans and Le Clerc29) allows any given segment to address a small number of other segments; each of these in turn can address other segments so that the total address space is very large, but the addressing and protection hardware required may be simplified.

The system other than Multics which has received considerable attention within the computer profession is the IBM 360/67.18) This system is based on a scheme developed at the University of Michigan.2) The segmentation system of the RCA Spectra 70/46 is very similar. The first major difference between the GE-645 and the IBM 360/67 is the way a generalized address (segment number/location within segment) is formed.

All generalized addresses in the IBM 360/67 are formed using conventional indexing and indirect addressing. The address formed after full modification is broken into two parts, a segment number and a location within segment number (the location within segment number is paged as in the GE-645, but we ignore that level of detail). In the GE-645 the segment number could not be modified by indexing. The two types of segmentation have been called linear and symbolic segmentation respectively.44) With symbolic segmentation (GE-645), the segments are logically independent in the sense that there are no contiguous addresses across segment boundaries. That is, there is no relationship between the last address of segment n and the first address of segment n+1. There is no way using address modification through indexing to obtain an address in segment n+1 from an address in segment n. With linear segmentation (IBM 360/67) addresses are contiguous across segment boundaries. The last address of segment n is contiguous with the first address of segment n+1. That is, one can modify an address in segment n to produce an address in segment n+1.

The above difference has many implications, one of the more important is that, because the segment number is the relative position of a segment descriptor in the segment descriptor table, the position in the segment descriptor table of a descriptor is very important in a linear segmented system. A linear segmented space is not two dimensional, but is a large one dimensional logical space. Another way to look at the difference is that linear segmentation blurs segmentation as a logical concept with the concept of physical memory allocation.

The 24 bit address version of the IBM 360/67 can only address 16 segments, which makes it necessary to pack several independent programs in the same segment. With such a usage one can reduce the number of page tables required over what a strictly paged system would require because the page tables of shared programs can be shared, but the full logical generality of segmentation is lost. That is, used in this way segmentation facilitates physical memory utilization, but ceases to be strictly a logical concept, thus, losing much of its justification for existence.

As with the GE-645, the IBM 360/67 uses an associative map to try and reduce the number of times the double table lookup in segment and page descriptor tables is required. The IBM 360/67, however, has the quirk that the instruction fetches are not made through the full map, but through a separate associative register which acts like a base register for the program counter. The situation of a loop executing across page boundaries requires references to the maps in main memory. This situation would not occur in the GE-645, because both pages of the loop would have entries in the associative map. Another quirk of the 360/67 is that memory protection is based on blocks of physical memory and not on logical units like segments, as on the GE-645. It seems reasonable to assume that the designers of the IBM 360/67 did not fully understand the motivation for segmentation, because their implementation does not show carefully thought out, valid cost-generality trade offs. The implementation instead is a rather inhomogeneous mixture of blurred concepts.

What, then, can be said about the present state of the art with regard to segmentation? The 360/67 has failed to materialize as an econonomically justifiable commercial product in spite of the thousands of man hours and millions of dollars invested in the system by IBM. The failure of this system is not entirely to be blamed on the segmentation implementation, however. A major portion of the blame is to be given to the swapping philosophy chosen and to the inadequate attention given to the total memory and I/O system design. The swapping philosophy is based on demand paging, which tries to maintain in memory only those pages required by a process and to swap in a page only on demand. The auxiliary storage units chosen for swapping mediums were poorly matched to such a philosophy.[34,35] Swapping is discussed in some detail in the next section.

The Multics system is behind schedule. Although the entire design of the segmentation implementation is logically consistent, the costs associated with achieving this level of generality and consistency are high. The swapping philosophy of the Multics system is also based on demand paging. Although the GE auxiliary swapping storage is of higher performance than that of the 360/67, it is still not well suited to be used with demand paging.

There are no reasons to believe that initial versions of Multics will be any more economical than the 360/67. Again one must remember that Multics is a research project trying to determine the techniques required and the costs associated with achieving full generality of segmentation. Based on experience with early versions of the system, the developers should be able to experiment with improved swapping and resource allocation algorithms and determine what extra or modified hardware could ease bottlenecks. The designers can then examine the various cost-generality trade offs available and redesign the system in a modular fashion such that in a commercial version installations can pay for the generality required.

The Burroughs B-5000 computer and its successors are successful commercial products, but segmentation was not implemented with the same degree of generality attempted with the IBM 360/67 or Multics systems. Burroughs is presently designing a large scale system, the Burroughs B-8500 for general purpose time-sharing. The first version of this machine had an inadequate memory bus and I/O system design for handling efficient swapping. A new version is being designed. Hopefully, this system can capitalize on Burroughs experience with segmented systems and the experience of the IBM and the Multics groups.

We can sum up in conclusion the following points: 1) large scale systems attempting to achieve a large logical memory space and full generality of procedure and data sharing are still very much in a research and development stage, 2) segmentation is a sound logical concept if one accepts the basic assumption that a very large logical space and full generality of procedure and data sharing are required in a large scale time-sharing system, 3) if one does not accept this basic assumption, then one can still achieve large logical and physical memory spaces economically in more conventional base-register or paged systems without the full generality in sharing single copies of procedures and data. Procedures and data can still be shared, however, 4) the rapidly changing technology of large scale circuit integration may alter significantly the design philosophy of future segmentation implementations, 5) decreasing hardware costs and experience with present systems should make more economical segmentation systems possible by the mid 1970's. The same decrease in hardware costs and increase in understanding may still favor more straight forward systems, because effective utilization of memory may not be such an important criteria. Full generality in sharing procedures and data can be looked at as a way to conserve memory. However, as hardware costs drop, the same effect, as seen by the programmer, can be achieved by allowing multiple copies of some procedures and data to exist.

Miscellaneous Memory Allocations and Addressing Problems

An aspect of addressing which needs mentioning is that related to I/O devices. When a user requests certain information to be written into or out of main storage in a system utilizing paging or segmentation, he uses logical addresses, and thus the I/O processor ideally should also access memory through the same type of map used by the central processing unit.[50)] This requirement is particularly important for block transfers where part of the block is in one page and the remainder is in one or more other pages. In a conventional I/O processor the starting address of a transfer and the number of words to be transferred is given to the I/O processor by the CPU and then the specified block of information is transferred. In a paged system without mapping in the I/O processor, a new address needs to be sent to the I/O processor by the CPU when a page boundary is passed. This process slows down the I/O operation and, on transfers to devices such as high speed drums, could result in lost information.

On the SDS-940, the direct access communication (DACC) channels are sent two page numbers before a transfer is started so that when a page boundary is crossed, the second page number is automatically used to control the transfer. Then a signal is sent to the CPU to send the next page number and the CPU has the time corresponding to a page transfer to respond. Another approach is being taken with a cathode ray tube display process to be attached directly to the 940 memory.[53] In this device a full memory map is used identical to that in the CPU.

The remaining topic which requires some discussion is the question of how system routines access memory. In the SDS-940 system there are, in fact, two hardware memory maps, one for user processes and the other for the system. The system would not require mapping if system routines occuppied fixed core positions. Some system routines are not used frequently enough or required to give such a fast response that they must permanently reside in main memory. One would like to be able to dynamically allocate memory to these routines as they are needed. The SDS-940 system achieves this ability by using a system map. The machine uses one or the other of the two maps depending on the mode in which it is operating, either system mode or user mode. These modes are discussed in the section on protection. Use of two maps enables the system to perform its functions without requiring changes to the users map.

The Multics system uses a different approach. System segments are assigned to each job's segment descriptor table and logically become part of each user's computation. These system routines use the same mapping mechanism as the user's routines. The potential difficulty with this approach is that system interrupt and allocation routines and user calls in the system which occur during the execution of a user process will cause changes to the associative map, thus necessitating frequent double table look ups to restore the map when control returns to the user process. We have seen no comment on the above problem in the literature. Whether or not it is a potentially serious problem depends on the frequency of interrupts and transfers to the system. There are no readily available statistics on the above problem. This lack of statistics is one of the problems facing system designers. There is an increasing interest in the computer profession in collecting statistics on system parameters, which when they become available should be valuable to the designers of the next generation of machines.

Swapping

Let us now look in more detail at the problems encountered with demand paging in particular and swapping in general. If there were enough main memory, swapping would be unnecessary, or at least might have to be performed less frequently. The prime limitation on the present capacity of main memories is cost, although signal propagation time is also a factor. Hardware technology is changing rapidly, however, and the cost and size of main memory can be expected to decrease. As an example of the rapidly changing hardware technology, large scale integrated circuit memories costing 1-2 cents per bit will be commercially available in the early 1970's. These memories will have read-write cycle times of 100 nanoseconds or less and capacities in the millions of bits. By comparison core memories with 500 nanoseconds cycle times now cost around 5 cents per bit. The availability of the low cost, highly reliable large scale integrated circuits will make possible hardware solutions to many of the problems now solved with software. Thus simpler more efficient implementations of segmentation-paging concepts will be possible in the next generation of machines.

Even with the type of main memory cost reduction illustrated above, main memory is still an order of magnitude more expensive than auxiliary storage such as drums and, therefore, designers will try to take advantage of this cost differential by designing systems which utilize swapping. There are two important parameters of an auxiliary storage device used for swapping, 1) the average length of time required to access the required block of information, and 2) the time required to transfer the block to or from main memory. The former is called the <u>average access time</u> and the latter is inversely proportional to the <u>transfer rate</u>. The average access time for most drums used for swapping, to date, is between 15-40 milliseconds, although UNIVAC has a small 1.5×10^6 character drum with an average access time of 4.25 milliseconds. Transfer rates can vary on drums from a few hundred thousand six-bit characters per second to several million six-bit characters per second. Besides the times mentioned above associated with the hardware characteristics, there is the additional time required to locate the required information. It has been stated, by MIT's Professor Corbato at a course given on problems in time-sharing during the summer of 1968, that when a Multics page fault (a missing page) occurred, the routines handling the condition required as long as 75 milliseconds to perform their processing when two page sizes (64 words and 1000 words) were used. When one page size (1000 words) was used, this time was reduced to 4 milliseconds. Even at the lower figure, this time is significant.

Randell and Kuehner[44)] have suggested a measure of memory utilization called the <u>space-time-product</u>; space being the amount of main memory a process is occupying, and the time being the length of time the space is occupied. The measure is useful because even if the computation for one process can be overlapped with the swapping of another, the process requiring the swapping is taking up space in memory for some length of time. Thus, this idle process is using a valuable resource. The choice of a swapping algorithm must be chosen to minimize the total space-time-product of all processes run in the system and still meet rapid response criteria to the users. The choice of a swapping algorithm and the success or failure of a total paged system is related to, 1) the page demand characteristics of programs, 2) the software time required to locate pages on auxiliary memory, and 3) the accessing and transfer characteristics of auxiliary storage devices.

Fine et al.,[17)] and Varian and Coffman[5)51)] have investigated the dynamic behavior to be expected in a paged computer and have produced some pessimistic results. In the Fine study five rather large programs were studied. Program size varied from 14K to 44K words and averaged 30K words. Each program was considered divided into 1K word pages. None of the programs were in any way designed for a paged machine. This is not an important limitation because Coffman[5] argued that programs designed for a paged machine will yield similar results. A plot of data presented by Fine is shown in Figure 12. This plot shows that very early in a time slice (at 30 milliseconds) 18 to 20 pages are required. In fact they found that if a program required 20 pages, 25% of the time the program required these pages within 7 milliseconds. This high page demand means that very frequently, using a demand paging swapping algorithm, the process is going to be halted waiting for a page. If one plots the space-time-product based on such a curve as above one gets the result shown in Figure 13. This figure shows the active periods followed by periods of inactivity. The attempt to mask these periods of inactivity by running other processes is only going to be successful if the ratio of inactive to active time is low. Using Fine's data, the first 10 active periods have a total active time of 0.8 milliseconds (on the SDC-Q32). Using a demand paging strategy and a drum with 17 millisecond average access and an infinite transfer rate, the inactive time during the period would be 170 milliseconds. Thus the ratio is higher than 200 to 1. Because all other processes in the system are going to behave with similar statistics as well as be halted for other I/O services, demand paging from even a fast drum is rapidly going to result in an I/O bound system. Nielsen's simulation of the IBM 360/67[34,35] showed similar results.

Several approaches have been suggested to improve this situation. The more important ones are: 1) user or compiler optimization of program structure to decrease the page demand rate, 2) starting each process with a "working set" of pages (affinity paging), and 3) minimize the average access time for the swapping medium. No practical proposals for accomplishing suggestion 1) above have appeared and Coffman[5] and Fine[17] are not optimistic that this approach can yield significant improvements. The second suggestion is quite workable and will undoubtedly be used. Denning[14] has suggested a method for determining a working set to be used in an affinity paging algorithm. He defines the working set of information $W(t, \tau)$ of a process at time t to be the collection of data items referenced by the process during the process time interval $(t-\tau, t)$. The parameter τ must be selected to reflect other parameters such as main memory size, auxiliary storage transfer rates and access time. Another design problem is determining $W(t, \tau)$. Denning makes a proposal for a hardware device which resides in main memory and collects the necessary statistics. The reader interested in the details is referred to his paper. He further makes the important point that the scheduling process and memory management are intimately interrelated for a system using a swapping algorithm based on a working set. The important point here is that main memory and auxiliary memory parameters will indicate what the nature of the basic swapping philosophy will be. This latter choice will inturn influence the other important resource allocation algorithms. The net result is that resource allocation and swapping algorithms for an efficient economical system are intimately related to hardware performance characteristics. This statement seems obvious, but its truth does not seem to be reflected in the stated objective of the Multics design, which is to implement a hardware independent system.

Figures 12 and 13 follow

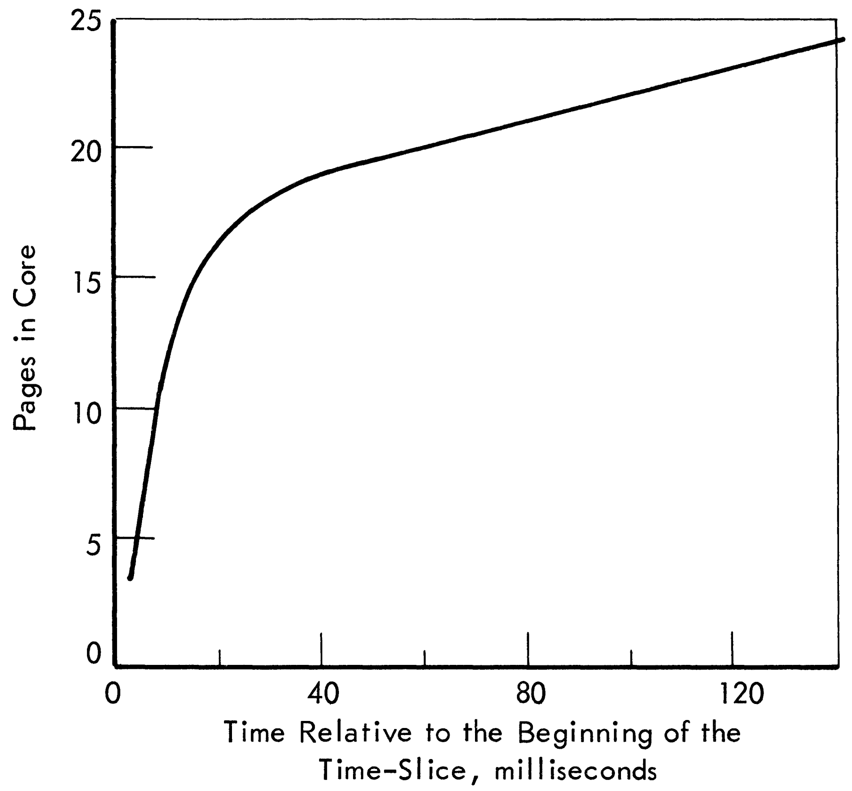
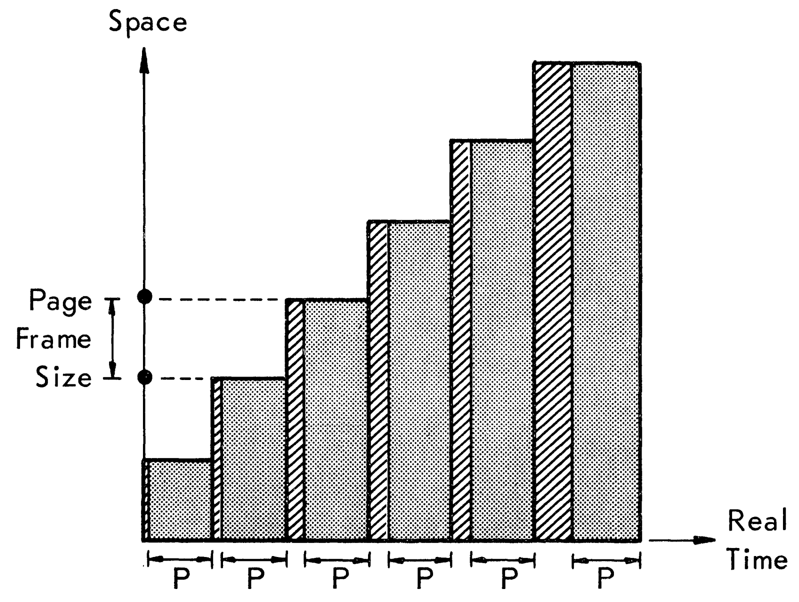


Figure 12. REPRODUCTION OF DATA PRESENTED BY FINE



▩	Program Awaiting Page
▨	Program Active
P	Average Time for Page Fetch

Figure 13. STORAGE UTILIZATION WITH DEMAND PAGING

Faster swapping mediums such as bulk core are being experimented with. Bulk core is slower speed random access storage (access time being 3-8 microseconds) then main storage and less expensive by about a factor of 2-4. Decreasing the average access time of rotating devices below the UNIVAC drums 4 milliseconds will require making drums even smaller, which will require breakthroughs in increasing information packing densities. Otherwise large numbers of such drums will be required, thus decreasing the cost advantage of drums over bulk core. Some improvements can be expected in this area, but a factor of two seems optimistic. Bulk core is still expensive relative to rotating devices, but total system performance may make it economical. Affinity paging will be used with rotating devices because a working set of pages can be brought into main memory very rapidly compared to the average access time. Disk storage with its longer average access time and slower transfer rates is a poor choice for a swapping medium.

Let us consider the use of bulk core briefly as its use is expected to be more common, particularly as its cost drops. There are two basic types of bulk core presently on the market. One type is marketed by IBM and called large capacity storage (LCS). The other type is marketed by CDC called extended core storage (ECS). These two types of storage have fundamentally different properties as backing stores and would lead to the implementation of very different swapping systems. At Carnegie-Mellon Institute they are experimenting with LCS as a swapping medium for their IBM 360/67.[28)] LCS has an 8 μsec cycle time and a 4 μsec access time and can be directly accessed by the CPU. In addition, transfers can be made in blocks via a storage channel making the LCS appear as a drum with an 8 μsec average access time and a transfer rate of 400,000 32-bit words per second. In an LCS system there are in effect two different page sizes, a 1024 word page size which is swapped via the channel and an LCS double word which can be directly accessed. If the monitor "knew" that only a few words in a page were to be referenced, it would be cheaper to access the words directly in LCS rather than performing the swap. How the monitor is to know such a fact is an unsolved problem, but the opportunity is there. It takes 2.5 milliseconds to swap a 1K word page from LCS and assuming some time to process the page demand this means latency for LCS is about 3-4 milliseconds. This is a significant improvement over a drum and if the cost (about 3 cents per bit presently) can be reduced by a factor of five or so, LCS will be economically competitive with drums as a swapping medium. Experience at Carnegie-Mellon using LCS has shown a significant performance improvement, as would be expected. The use of bulk core storage still does not solve the swapping problem completely because without placing restrictions on program size it may not be economically possible to keep all users programs and data in bulk storage and thus the "swapping problem" is pushed back one level in the storage hierarchy between bulk core and drums or disks.

CDC's extended core storage cannot be directly accessed by the CPU and requires 3 μsecs to access the first 60 bit word. Once the first word is located a transfer rate of 10 million 60 bit words per second can be maintained. Thus a 1K word block can be transferred in 100 μsecs. This high speed has two implications: 1) swapping and computation cannot be overlapped because the transfer requires all the memory cycles and 2) processing necessary to locate the required block in ECS constitutes the major delay in accessing the block.

The main unit of information transfer in an ECS system is probably more suited to be the program or segment rather than the page and thus a machine designed around ECS can probably be designed with a simpler storage management system than one designed around a drum or LCS.

Memory Bus and Control Structure

It has been stated that the central resource in resource sharing systems is main memory. In such systems high transfer rate secondary storage devices in addition to numerous input-output devices share access to the memory with the central processing unit(s). The processors which control the secondary storage and I/O devices and communication with memory are usually referred to as channels, I/O controllers, or I/O processors. Sometimes the function of several processors is combined into a device called a generalized I/O controller (GIOC)[19]. The design problem is to provide the various processors with an adequate data transfer capability to and from main memory. Ideally each processor should be able to transfer a datum to or from main memory at its convenience without regard to the ability of the memory to accept or supply the datum at that particular moment, or the ability of the processor-to-memory transfer-addressing path (memory bus) to effect the transfer. Unfortunately economic and technical considerations relegate the above ideal to a standard against which practical systems can be compared.

In practical systems the rate at which data can be transferred between processors and main memory is limited by the transfer capabilities of the memory itself and the memory busses. The rate at which the memory can transfer information is often referred to as the memory bandwidth, usually measured in words per second. Bandwidth limitations also exist for the memory busses. Because the memory system is shared by several processors, care must be taken in the design to keep performance from being seriously degraded due to interference caused by simultaneous attempts on the part of the several processors to utilize a facility such as a memory bus or portion of memory itself. Figure 14 shows the usual method for organizing the memory structure in a resource sharing system.

The memory, instead of being considered one monolithic unit with one set of addressing circuitry and one set of read-write circuits, is broken into some number n of smaller boxes each with its own set of addressing and read-write circuits. In the SDS-940 for example a 64K memory is broken into four 16K boxes. Accessing each box are up to m data-address busses. Each data-address bus has cables for data to and from memory and the required address. The quantity m in present systems usually varies from 2 to 5. A schematic of one representative memory box is shown in Figure 15. Any combination of busses, a, b, c, or d may request access to the memory box simultaneously. The priority access control must decide which bus is to be given access on this particular memory cycle and the requests on the other busses must remain hanging until the next cycle or be cancelled and renewed

Figures 14 and 15 follow

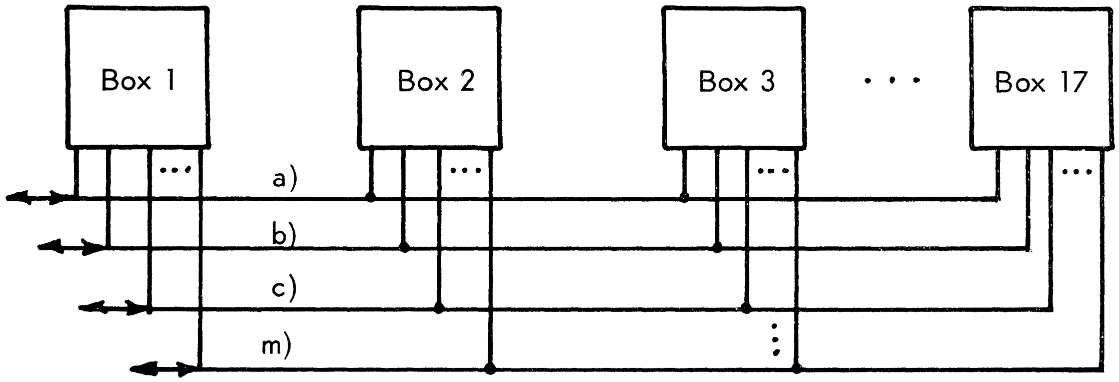


Figure 14. MEMORY ORGANIZATION IN A RESOURCE SHARING SYSTEM

249-68
64692

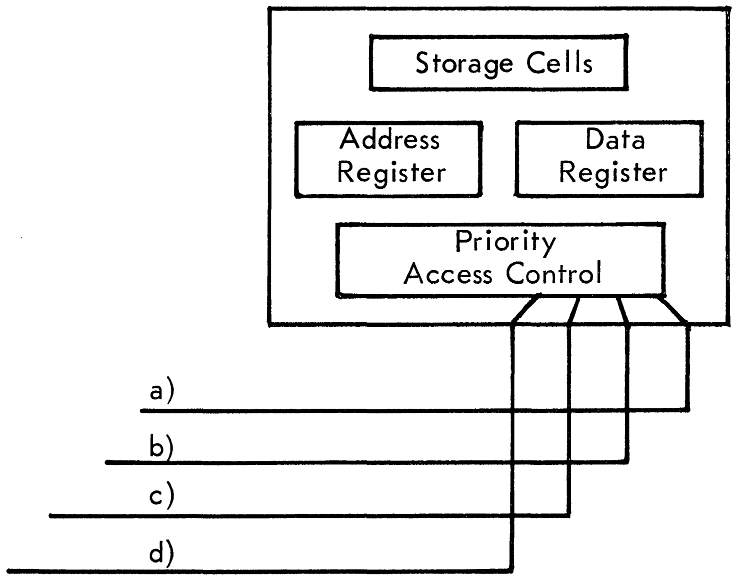


Figure 15. DETAIL OF A MEMORY BOX

on the next cycle. The term bus structure is frequently used to refer to the bus-priority control complex.

The scheme shown in Figure 14 cuts interference by allowing simultaneous access to more than one box. That is, if bus (a) requests access to box 2 at the same time bus (b) requests access to box 3 both accesses are granted because each box has its own addressing and read-write circuitry. Even given the scheme shown in Figure 14 serious interference could result if memory addresses were contiguous in the boxes, for example, box 1 having addresses 1 to 16K and box 2 addresses 16K + 1 to 32K etc. If this were the case, transfer from a high speed drum might tie up a box during the period of transfer and cause interference with the arithmetic-logical unit. To get around this problem designers have developed the technique called interleaving. In an interleaved memory, consecutive addresses are in different memory boxes. For example, in a two-memory-box system all the even addresses would be in one box and all the odd addresses in the other. With an interleaved memory probability of one processor tying up the memory for a significant time is greatly decreased.

To make a memory reference in one of these multi-bus configurations, a processor first makes a request for use of its memory bus; and when this request is granted, uses this bus to transmit its memory reference request to the appropriate memory module. If either of these requests is rejected the processor repeats the procedure or leaves its request hanging.

To resolve conflicting requests for either a memory bus or memory box some type of priority mechanism must be employed. Priorities typically are assigned to processors and to memory busses; the former are used to resolve conflicting for a memory module. The design of the total memory structure including the priority scheme is critical if bottlenecks in this important area are to be avoided and expansion capability is to be provided for the addition of new I/O devices and processors.

A discussion of some of the present systems is instructive as it shows the variety of approaches which exist. Two basic types of memory systems exist, synchronous and asynchronous. In synchronous systems, requests for memory can occur only at discrete time points and thus the concept of "simultaneous requests" can be used. The point of request is usually just prior to the start of a memory cycle and at this time all processors requesting a memory reference place their requests on the lines and the priority mechanism arbitrates between them. In asynchronous systems, a processor can request a memory reference at any point in time and the concept of simultaneous requests loses meaning. In systems such as the GE 645, asynchronous memory access is used, while the SDS-940 is an example of a synchronous system. The argument for using asynchronous memory references and asynchronous communication among major system modules is that it allows replacement of modules with others having different performance characteristics without requiring modifications to the unreplaced system modules. Further, asynchronous communication can take place at maximum rates not restricted by the times fixed for synchronous systems. These advantages are real, but have an associated cost. In asynchronous systems the concept of priority has little meaning because the requests for access generally are handled on a first come first served basis. Therefore, physical and statistical properties of the various processes cannot be adequately taken advantage of to improve the performance at a given cost point by the sharing of memory busses and allowing the use of variable priority requests.

A typical asynchronous system is shown in Figure 16. In this figure each central processing unit has its own bus to memory. In a system with faster CPU's than the GE 645 using instruction look ahead [fetching of the next instruction or n instructions while the current instruction (s) are being executed], each CPU might have more than one memory bus to handle the overlap of instruction and operand fetch. Note that the fast secondary storage device used for swapping (the drum) has its own bus to each memory module. Finally a processor called the GIOC (general I/O controller) has a separate bus to each module. The GIOC is a complex piece of hardware combining the functions of dedicated channels and multiplexor channels along with buffering, assembly[a] and device control functions.[19] One of its major functions is to schedule access to its one bus to each memory module. Because the system is asynchronous, access to a memory module is on a first come first serve basis. The memory modules are interleaved. This asynchronous approach does provide ease of replacement of CPU's or memories with faster units with little or no hardware and software modifications, but is expensive of hardware and possibly somewhat restrictive on the approaches one can take to adding fast I/O devices other than the drum.

An example of a synchronous system which can inexpensively use simple priority schemes to increase memory and bus bandwidth is the SDS-940. A schematic of the SDS-940 is shown in Figure 17.

There are two busses to 940 memory. The memory is interleaved four ways. In the SDS-930, which served as the basis for the 940, the bus called B_2 (second path) in the figure always had priority over bus B_1 (first path). This arrangement of priority is common in second generation machines and many third generation machines and is based on the idea that high speed devices such as drums and discs must have access to memory when required or data can be lost, because these devices are constantly rotating. In the Direct Access Communications Channel (DACC) of the 930 there is a one register buffer as shown in Figure 18.

---

a) Assembly is the operation of accepting characters from the attached devices and assembling them into computer words. A dedicated channel allows only one high speed device to be attached at a time for communication with main memory. A multiplexor channel allows many slow speed devices to be attached at one time for communication with main memory. The multiplexor channel in effect communicates between attached devices in such a way that each device functions as though it is attached to a dedicated channel.

Figures 16 and 17 follow

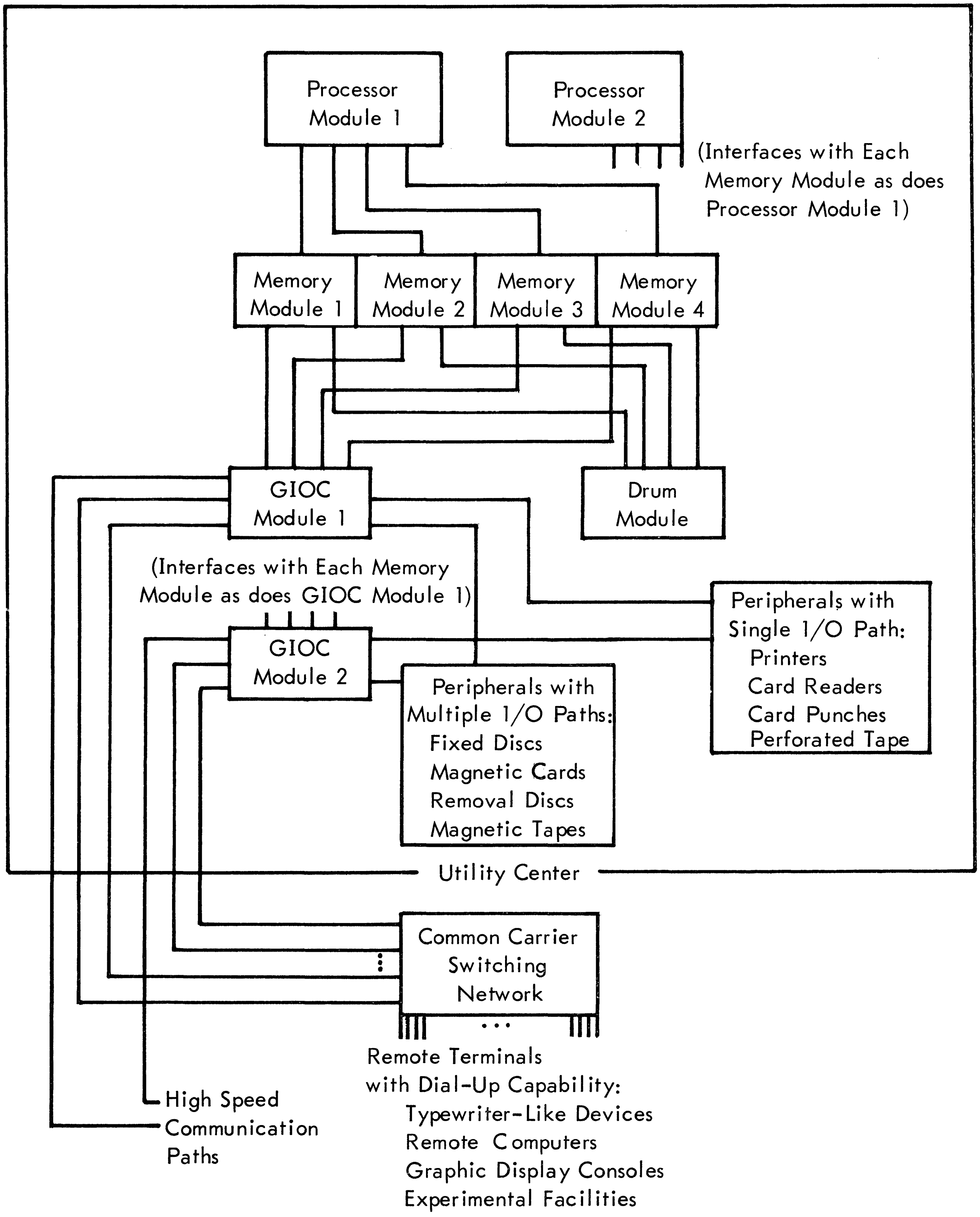

Processor Module 1
Processor Module 2
(Interfaces with Each Memory Module as does Processor Module 1)
Memory Module 1
Memory Module 2
Memory Module 3
Memory Module 4
GIOC Module 1
Drum Module
(Interfaces with Each Memory Module as does GIOC Module 1)
GIOC Module 2
Peripherals with Single I/O Path: Printers Card Readers Card Punches Perforated Tape
Peripherals with Multiple I/O Paths: Fixed Discs Magnetic Cards Removal Discs Magnetic Tapes
Utility Center
Common Carrier Switching Network
High Speed Communication Paths
Remote Terminals with Dial-Up Capability: Typewriter-Like Devices Remote Computers Graphic Display Consoles Experimental Facilities

Figure 16. EXAMPLE OF A MULTICS SYSTEM CONFIGURATION

249-68
64692

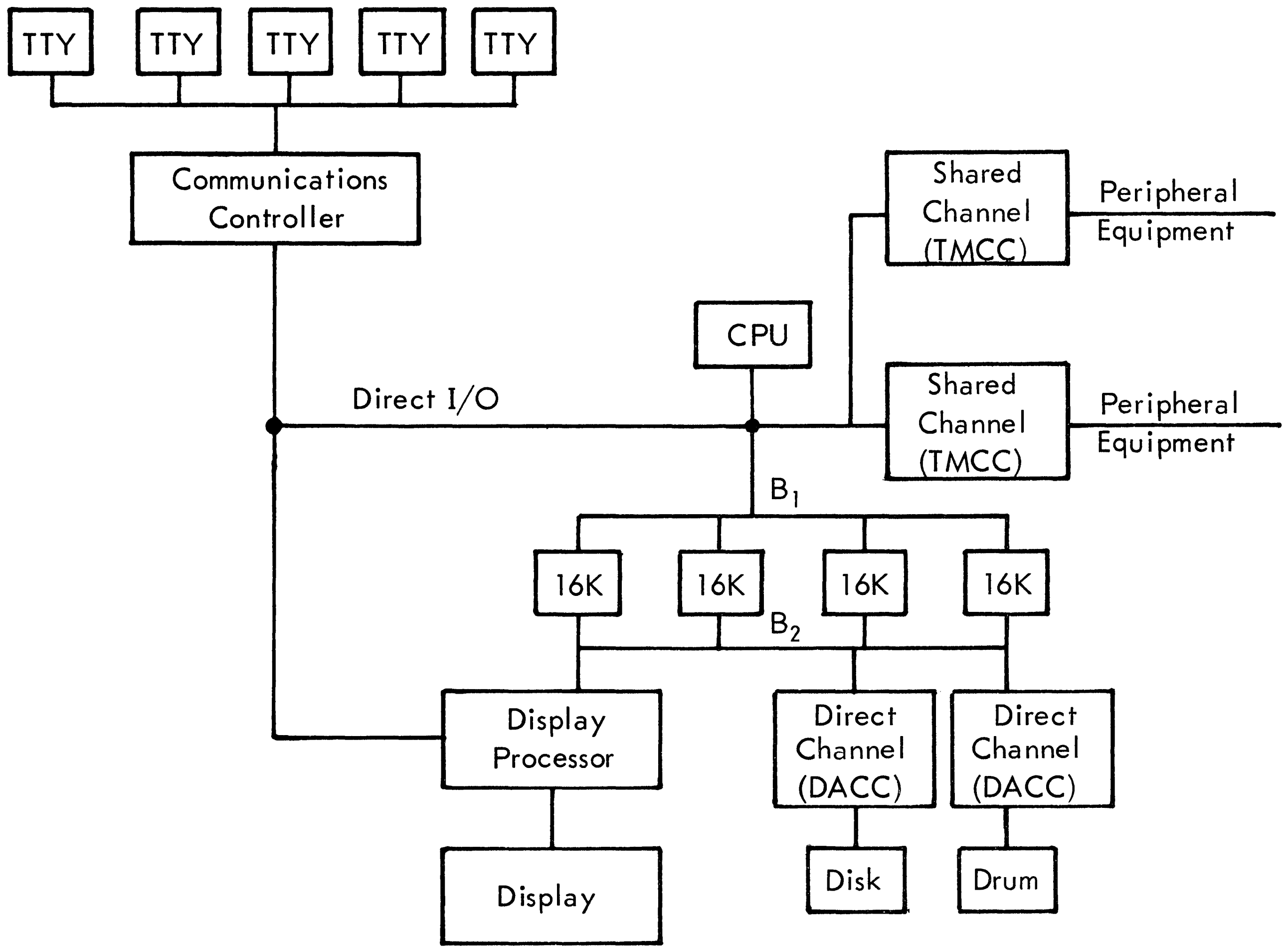


Figure 17. SDS 940 CONFIGURATION

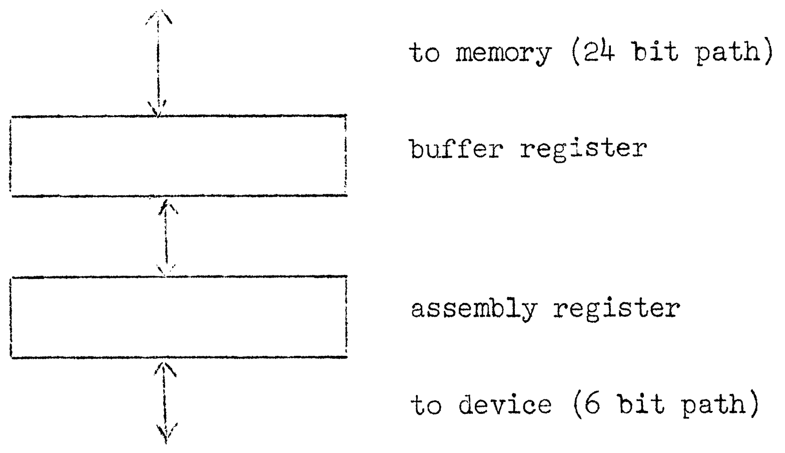


Figure 18. REGISTERS IN THE DIRECT CHANNEL

The assembly register packs four 6 bit characters in a 24 bit word and then transfers the word to the buffer register on input from a device. On output to a device an analogous operation results in the opposite direction. At this point the 930 DACC requests a memory cycle to store the contents of the buffer register. Because the second path has priority over the first path this request is granted and the potential value of the buffer register is not obtained. To see the effect of this arrangement on memory interference consider the following example. If a high speed drum were transferring characters at the memory rate (i.e., one word every four memory cycles) and the memory consisted of two boxes not interleaved an interference rate of 25% could result. If the two boxes were interleaved 12% interference could result. Even 12% is a high interference factor and is unnecessary if the full buffering capability of the buffer register is used.

When the assembly register transfers a word to the buffer register, the buffer register has at least three memory cycles and possibly four in which to transfer its contents to memory before the assembly register transfers its next word to it. Therefore, during any one of these three to four cycles the buffer register could accomplish its transfer task. It is the purpose of buffering to gain time. Therefore, what the developers of the 940 did was to take advantage of this potential buffering action by allowing the second path to request memory access with either a higher or lower priority than the first path. For example, after the assembly register transfers its contents to the buffer register, the DACC makes its first request for memory with low priority. If this request is granted then one knows that no interference resulted. If this request is not granted, then a second low priority request is made. If this second request is not granted then the third request is a high priority one. Simulations have shown that with this system memory interference with the CPU is less than 1%.40)41)

Making use of this priority feature, a high performance general purpose display processor has been designed for attachment to ERC's 940. In a display the picture on the screen must be repainted on the screen 30 - 60 times per second to give the illusion of a continuous picture. The display commands must be stored in a memory. With a computer not having the 940's priority scheme, attachment directly to main memory would cause serious interference with the CPU and thus the display would have to be maintained from the memory of a separate small computer or separate memory. With the 940's priority scheme and with the display accessing memory on an average of every 11 μs, less than 2% interference with the CPU is expected.53)

There is one additional priority problem and that is for the second bus itself if several devices are attached to it. In the 930, devices are strung on the second bus daisy chain fashion as shown in Figure 19.

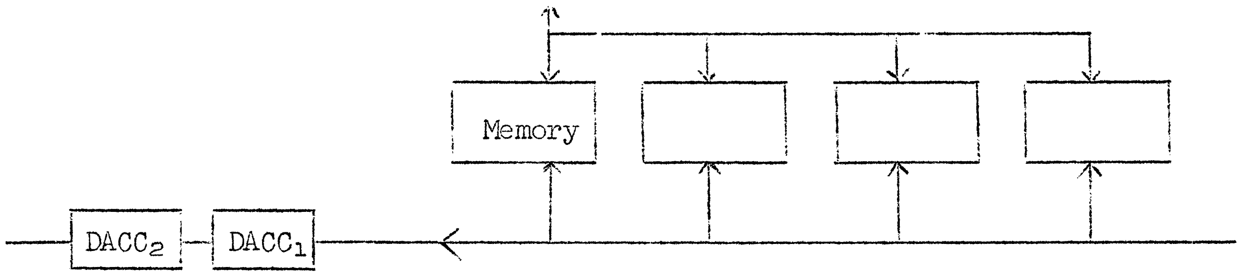


Figure 19. 930 SECOND BUS STRUCTURE

Priority for the second path is determined by position. That is, the further from the memory the device is attached, the higher its priority. For example, DACC$_2$ has a higher priority than DACC$_1$. Such a scheme is fine for the case where the second path always has priority over the first path, but inefficient when devices on the second path can request access with variable priority. For in this case DACC$_2$ may be requesting with a low priority and DACC$_1$ with a high priority, but DACC$_2$ is tying up the bus. Therefore, simple changes to the logic were made so that devices on the second path request access to the bus with an A, B or C priority, with A being higher priority than B. The device requesting with the highest priority gains control of the bus. In case two or more devices request access with the same priority, physical position is used to distinguish among them. Priority with respect to the CPU is determined as follows: A equals high and B or C equal low.

In contrast to the machine discussed above let us consider a machine with a memory organization we consider inadequate to meet the demands which may be placed on it in a time-sharing system, the IBM 360/50. A schematic of the 360/50's memory access path is shown in Figure 20.

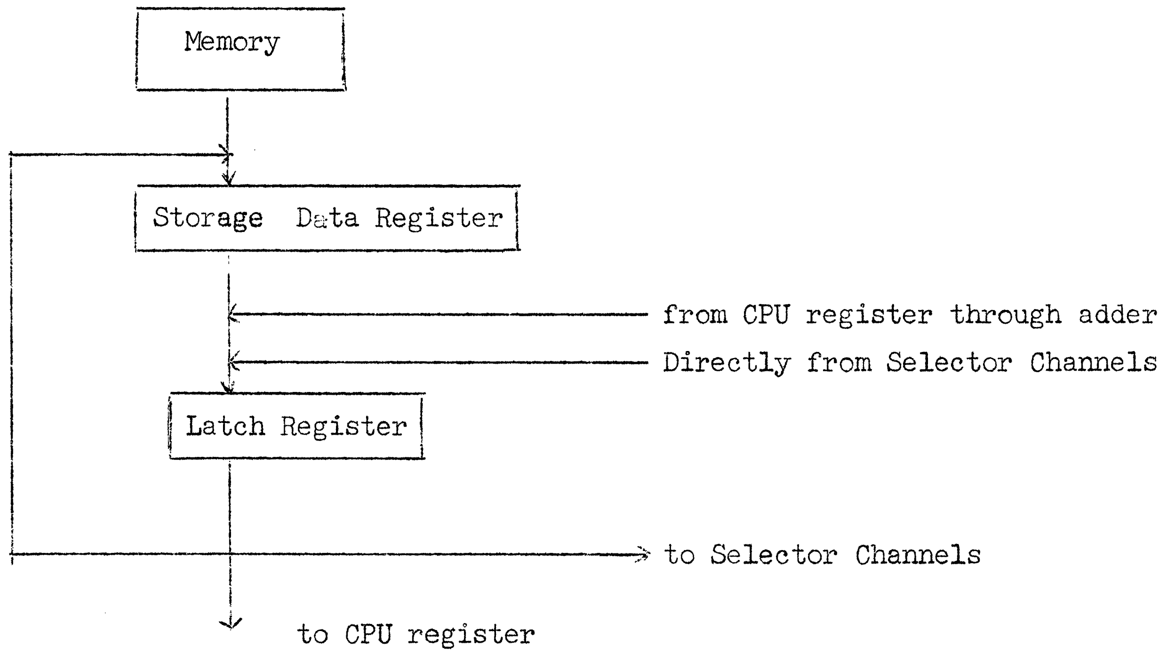


Figure 20. IBM 360/50 MEMORY ACCESS PATH

In the 360/50, memory is not interleaved and only one path to memory exists. Further, this one path shares access and addressing registers with the CPU. When the Selector Channels (Selector Channels are processors which can connect any one high speed I/O device to the bus) require to transfer a word to or from memory, the CPU is halted because there are common registers involved. There is another type of channel available on the 360/50 called a Multiplexor Channel (a multiplexor channel allows multiple slower speed I/O devices to be connected to a data path simultaneously). The hardware for assembly of characters from the slower speed devices into words uses CPU registers. Further, temporary storage for addresses associated with the slower speed devices and the partially assembled words is part of main storage. Thus, during character transfers from the slower speed devices CPU interference results.

Recent Developments

As mentioned earlier there are two major problems in designing a hardware system for resource sharing: 1) achieving the memory addressing capacity and speeds required, and 2) achieving the rate of information flow required between the memory and the processors. Large memory addressing capacity and high speed are incompatible requirements because of the physical distances over which the signals must travel. Thus even if cost were not an important factor, design of the storage system would require careful attention.

In present systems main store is made up of core or thin film memory. In systems like the IBM 360/67 and GE 645 mechanisms have been devised as discussed earlier in order to make the distinction between the levels of the storage hierarchy invisible to the user. The user thinks he is dealing with a large "virtual" store all of which operates at the same speed. This is an important concept, but has yet to be economically demonstrated in its full generality because of the problems brought out earlier. The problem results primarily because of the rotational, accessing and transfer delays inherent in the rotational stores. The substitution of slower bulk core in place of drums as in the Carnegie system results in a substantial improvement in performance, but at a considerable cost. Costs of hardware are decreasing and this approach will be used more frequently in the future. However, use of bulk core in place of drums as the first level of backup storage still does not come to grips with the speed differential of the logic of the processors versus that of core and thin film memory. Further, the problem of supplying adequate memory bandwidth so that all processors can operate at their full rate is still not solved.

Two machines, the IBM 360/85[8] recently announced and the Scientific Control Corporation (SCC) 6700, under development at the University of California, Berkeley, offer related but different approaches to the above two problems. Both these systems use a programmer invisible level of high speed solid state buffer storage between core storage and the CPU, in the case of the 360/85, and between all processors in the case of the SCC 6700. The goal is to use this high speed buffer storage in such a way that the processor "thinks" that all the storage is constructed of this high speed solid state memory. Before making a comparison between the systems a brief description of each is given. The discussion of the SCC 6700 is based on unpublished discussions with researchers at the University of California, Berkeley.

A schematic of the Model 85 memory system is given below.

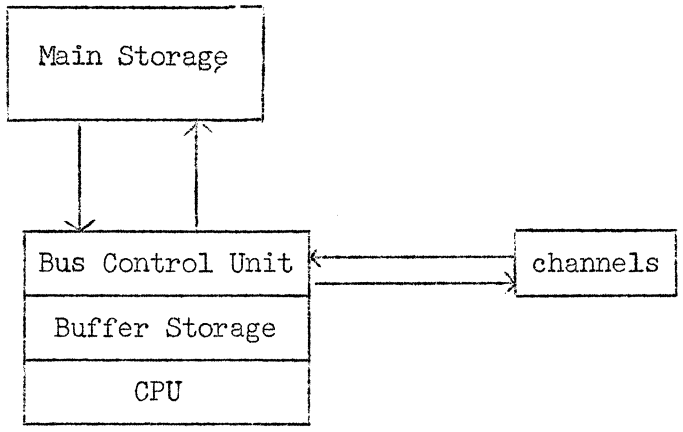


Figure 21. IBM 360/85 MEMORY SYSTEM

Main storage in this system has a cycle time of about 1 microsecond. For storage configurations of 500K and 1000K words (32 bit), storage is interleaved four ways. For smaller storage configurations, storage is interleaved two ways. Note that the buffer storage is only available to the CPU and not to the I/O or other processors. The buffer storage has a cycle time of 80 nanoseconds. The buffer storage is either 4K, 6K or 8K words. The goal in designing this system was oriented toward increasing the effective speed of memory as seen from the CPU. The importance of high data transfer rate between all processors and memory has not been highly developed in this machine because it is still seen as primarily a batch processing machine for CPU-bound scientific applications. The memory bus is 4 words wide in order to achieve the bandwidth required for the main applications envisioned. For business applications which are primarily I/O bound the organization of the 360/85 offers little advantage.

Main memory and the buffer storage are organized into sectors of 256 words. During operation, a correspondence is set up between buffer storage sectors and main storage sectors in which each buffer storage sector is assigned to a single different main storage sector. Because of the limited number of buffer storage sectors, most main storage sectors do not have any buffer storage sectors assigned to them. Each of the buffer storage sectors has a 14-bit sector address register, which holds the address of the main storage sector to which it is assigned.

The assignment of buffer storage sectors is dynamically adjusted during operation, so that they are assigned to the main storage sectors that are currently being used by programs. If the program causes a fetch from a main storage sector that does not have a buffer storage sector assigned to it, one of the buffer storage sectors is then reassigned to that main storage sector. To make a good selection of a buffer storage sector to reassign, enough information is maintained to order the buffer storage sectors into an activity list.

When a buffer storage sector is assigned to a different main storage sector, the entire 256 words located in that main storage sector is not loaded into the buffer at once. Rather each sector is divided into 16 blocks of 4 words each, which are located on demand.

Storage operations always cause main storage to be updated. If the main storage sector being changed has a buffer storage sector assigned to it, the buffer is also updated, otherwise, no activity related to the buffer takes place. Since all the data in the buffer are also in main storage, it is not necessary on a buffer storage sector reassignment to move any data from the buffer to main storage.

Two 80 nanosecond cycles are required to fetch data that are in the buffer. The first cycle is used to examine the sector address and the validity bits to determine if the data are in the buffer. The second cycle is then used to read the data out of the buffer. If the data are not present in the buffer, additional cycles are required while the block is loaded into the buffer from main storage.

Simulation was used extensively during the design of this memory system as well as that for the SCC-6700. There are many important parameters such as choice of a replacement algorithm, buffer size, sector and block sizes which must be determined.

With the simulation running a representative scientific oriented job mix, it was found that mean performance of this system as compared to an ideal system consisting of only 80 ns memory was 81%.[8]) That is, 81% of the time on average the CPU obtained information from the Buffer Storage. Although the simulation technique used by IBM is sound, there has been controversy over the advertising claims based on the simulations in the trade journal "Computerworld", (see May 22, 1968 issue).

The Scientific Control Corporation (SCC) 6700 machine is being developed jointly by SCC and researchers at the University of California, Berkeley. The group at the University of California is the same one which developed the SDS-940. The orginal goal was to develop a hardware organization for resource sharing which would incorporate as many hardware features as possible which research and development centered around the 940 showed would be desirable (several PhD theses and MS papers resulted from work on the 940 and generalized developments on that machine) and at the same time would allow much of the software developed for the SDS-940 to run with minor changes. As development proceeded the constraint on being compatible with 940 software has relaxed and the SCC-6700 will allow larger user processes than were allowed on the 940 and will require extensive software changes in the system. User programs written for the 940 will require fewer changes to run on the SCC-6700 however.

A few changes from the user point of view are mentioned here before going into the memory bus and memory organization because of the wide spread knowledge of the 940 within Shell. The addressing structure has been modified so that the users program sees a virtual memory of 512K. New instructions for byte manipulation and 48-bit floating point arithmetic have been added; modifications of the branch, skip, shift and I/O instruction groups have also been made.

The designers of this system recognized that memory organization, both main and secondary, was the central problem and all the other areas of the machine design should pivot around this point. Designers of the IBM 360/67, 360/85 and GE 645 were stuck with modifying existing machines and therefore were not in our view able to shift their point of view sufficiently to a memory centered one, although many valuable ideas have been introduced into these machines. A schematic of the 6700 memory organization is shown in <u>Figure 22</u>.

The basic machine has 8 memory boxes containing 16K of 1 μsec core memory and 6 fast 200 nanosecond register sets to be described. Additional memory boxes can be added. The memory is interleaved 8 ways and a 48 bit double word (instructions and fixed point data are 24 bits) is always fetched. It is beyond the scope of this report to go into all the details of the addressing and priority scheme, but some highlights can be mentioned. The 6 fast register sets in each box consist of storage for a double word, an address, and various status and priority bits.

Figure 22 follows

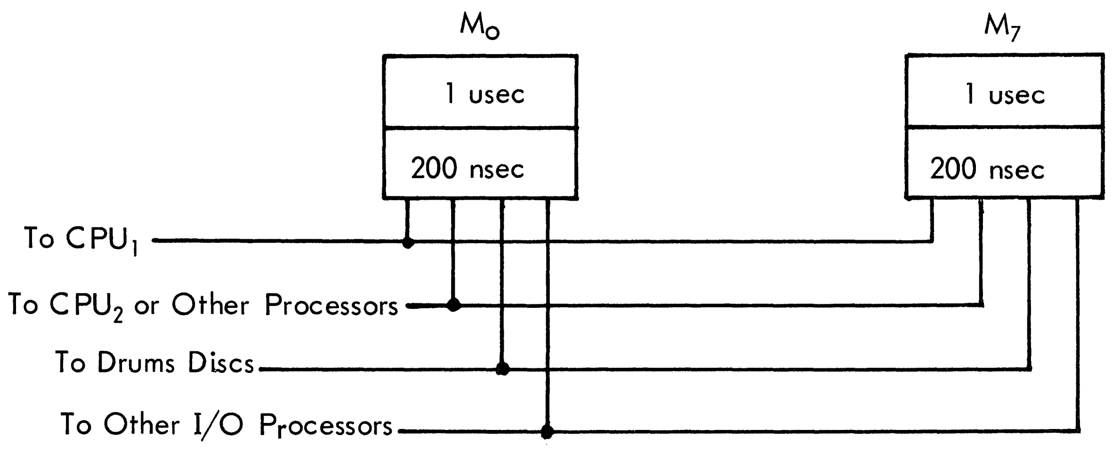


Figure 22. MEMORY ORGANIZATION OF THE SCC-6700

249-68
64692

It was mentioned earlier that sophisticated priority schemes could not be used with an asynchronous system because the concept of simultanerty cannot be made precise. Yet synchronous systems which have as a basic unit of time a memory cycle cannot run as fast as asynchronous systems. The designers of this system have chosen to make it synchronous, but in order to give the processors greater freedom to proceed at their own pace and to effectively use the 200 nsec memory, the machine uses 100 nsec intervals as the basic timing unit and the points at which requests can be made to a memory box.

Two main priorities are involved in this system, priority for access to a memory box, and priority for access to the core part of each box. The first priority problem is handled in two ways: 1) each request for memory is made either with a high or low priority request, 2) if two or more processors request access to the same box at the same time with the same priority a hard wired priority among the busses resolves the conflict. However, if a bus with a low hardwired priority requests a cycle indicating it wants it with high priority it gets access even if a bus with a higher hardwired priority is requesting a cycle with low priority. Once access to a box is granted the address is compared to the addresses of the fast memory and if it is found on a fetch and the status bit indicates that the data portion is valid, the data are delivered to the processor after 100 nanoseconds. Otherwise a hardwired algorithm releases a register set to this request. Additional core-access priority information comes with the requests to indicate whether a low, warning, medium or high priority is required for access to core. This priority is used to determine which request in the fast registers to service on the next core cycle. The warning level is used to reserve a fast register set by processors which know ahead of time in which box their next request will occur. When a second warning request comes along all previous warning requests are reset to a high priority.

Because the CPU's are designed around instruction and operand fetch look ahead they can make requests far enough in advance so that the required information will be in fast memory at the time it is actually required. Similarly the drum and discs on block transfers know ahead of time which boxes are going to be required. On stores, the information is left in fast memory by a processor and this information is later used to update core. If a fetch to the same location is then made at a slightly later time the information can be obtained from fast storage. For tight loops all information may reside in the fast storage. The result of this memory organization is that simulations have shown that more than 15 million words per second can flow to and from memory to keep all processors running at close to full capacity. The decision to use 6 fast register sets was arrived at through the simulation, which showed this number as optimum from a cost-performance point of view.

Besides the memory organization given above, the SCC-6700 has another unique design approach. The various I/O processors are microprogrammed to allow them considerable independence of the CPU to perform tasks previously requiring software. An example is the drum processor which has been given capabilities to handle swapping functions previously involving the CPU. A useful way to think of the above processors is to view the processors as forming a hierarchy much as the various storage media form a hierarchy.

The main advantage of the memory buffer organization of the 6700 over the 360/85 is that fast memory is available to all processors and not just the CPU. Further, stores in the 6700 are made to fast memory and not to the slower core memory. In a time-sharing system, transfers between main storage and secondary storage are one of the critical bottlenecks and thus this system approach of the 6700 is particularly important. The scheme of the 6700 should require less hardware and therefore be less expensive. The synchronous approach of the 6700 also allows development of a priority scheme which uses all available information to decrease interference for core requests.

System Protection

There are many levels of protection required in a resource sharing system, both hardware and software.[20)27)] Even though resource sharing systems have been in operation for a decade, they are still in early developmental stages. Therefore, the full range of protection mechanisms and the costs of the various protection mechanisms which are implemented or could be implemented is not fully known. Computer system design practice is to put in hardware only the most general purpose mechanisms or to solve clearly seen system bottlenecks with hardware. Thus, most of the protective mechanisms exist as software routines rather than as hardware. As experience is gained with present systems and designers gather statistics on the cost of the various protection mechanisms and see how they can be generalized and related to one another, more hardware protection will evolve. In this section we examine the main protection methods which exist in hardware and leave until later discussions of protection mechanisms existing in the software. The following varieties of protection have been distinguished.

1) Protection of the system from user processes. It is imperative that users not be able to take actions which would stop the system from running or destroy information essential to the system.

2) Protection of users from each other. A user must not be allowed to take any action which would harm the operation of another users process.

3) Protection of users from themselves. A user may have several processes in execution or one process with several subprocesses all of which are in intercommunication. A user may want to protect these processes from each other in terms of memory access, execution times or interaction paths.

4) Protection of the system from itself. A time-sharing system is a very complex entity which is constantly evolving. It is very desirable to have protection mechanisms which limit the damage which a malfunctioning module can perform.

For hardware design purposes, the above protection requirements have been reduced to two main areas:

a) protection against accessing, changing or transferring control to certain words in the physical memory of the machine (memory protection)

b) protection against executing certain instructions (control protection).

Given a set of hardware protection features, the software designer can use these features in many ways to develop a total protection scheme for the system. In this section we present various hardware protection features found on current or proposed machines and later in the section on software protection, the subject is treated again from the software point of view.

Memory Protection

In memory protection schemes, areas of memory can be given different classifications such as

1) inaccessible

2) read-write

3) read only

4) execute only.

Examples of uses for the various categories of protection above are the following:

Many programs may be residing in main memory at a given time as well as the monitor system and therefore some areas of store must be made inaccessible. Full read-write privilege is required by most programs. Some subsystems such as compilers, test editors and debugging aids as well as library routines may be shared by many users and thus by making them read only they can be properly protected. In a computer utility, renting time for use of special programs will be a significant business. In order to protect proprietary programming or other techniques one wants to prevent users from reading the code, but one does want the user to be able to execute the code, thus illustrating a situation requiring execute only protection.

Memory protection can be applied to the physical address space or to the logical address space. To protect the physical address space, the physical memory is broken up into blocks of a fixed size and a protection key is assigned to each block as a hardware register. This key indicates the type of access allowed to its corresponding block. The difficulty with this approach is that it does not distinguish among processes. Any process can write in a block with a write attribute key or execute a block with an execute attribute key. In a system using shared information some people need write privileges to certain codes and other persons are only to be allowed read access to the same code. To use such a system the software monitor would have to run around memory changing the protection keys each time the system switched control to a different process. Such an approach would greatly increase system overhead and result in an awkward design.

In an attempt to improve the situation somewhat, the IBM 360 uses two keys. One set of keys is assigned to memory blocks as described above and another key is kept as a status word with each process. Access of a particular degree is granted to a block if the key of the status word bears a specified logical relation to the key of the block. This scheme is still very inadequate because any process which has a key which might inadvertently bear a write relation to a block can write in a block. To guard against this would require very large keys. The IBM 360 uses a four bit key which gives only 16 combinations. Adequate protection using such a system still requires extensive software intervention.

A much better place to apply protection, which gets around the type of problem discussed above, is to protect the logical address space rather than the physical address space. In systems which use relocation or base registers to map the logical space to the physical, protection is provided with "bounds registers". The bounds register splits the memory into two parts, a contiguous region between the area pointed to by the base register and bound by the bounds register and the remainder of memory. Access is granted a process in the former area and denied the process in the latter area. This approach is an improvement over the previous memory protection scheme because protection is more easily changed as processes controlling the machine charge but still suffers from a serious defect. The type of protection offered by the bounds register approach is all or nothing. In a time-sharing system finer distinctions of protection are essential. Because protection is limited to contiguous areas, the sharing of procedures will require frequent changes of base-bounds register pairs.

The best approach to protection, we believe, is found on those systems having paging and/or segmentation hardware. The IBM 360/67 has the poor design feature that protection is maintained on physical blocks of memory, even though the system has elaborate segmentation hardware. In a protection scheme based on the paging or segmentation hardware, a procedure can more easily be given access only to those procedures and data segments necessary to accomplish its task and then only the type of access required to do its task.

The SDS-940, for example, has a simple protection scheme utilizing the map registers described above.30) It will be recalled that each map register contained 6 bits, but only five were used for the block address and the remaining bit was used to define a protection classification. If the protect bit is a One then the page is "read only", that is information can be retrieved from the page, but no information can be written into the page. If an attempt is made to write into a read only page, a fault interrupt is generated which transfers control to the proper system routine. If the protect bit is a Zero then the user can read from, or write on the page if it has been assigned to the user. Map registers corresponding to pages not assigned to the user have the protect bit set to a 1 and the other bits set to 0. This means that the user could never be assigned real block 0 which is always used by the system. The map registers are set to the type of access to be granted from information maintained in other tables.

A more general system of protection is employed on the GE-645 where protection is at the segment level.[20)] Part of the information stored in the segment descriptor word is the type of access to be granted the segment, write only, read only, read-write, execute only, no access, access only by system level processes. This protection information is set into the segment descriptor at the time a process requests access to the segment. The segment is located in the file system through a directory which not only indicates where on auxiliary storage the segment is stored, but which users can access the segment and with what type of access. Based on this information a segment descriptor word is set up. Another level of protection exists on the GE-645 and a similar mechanism exists on the IBM 360/67. The descriptor base register, and segment descriptors have associated with them a bounds field indicating the maximum defined segment number and segment length respectively. These bounds are compared against the two components of a generalized address and a failure to be within these bounds generates memory access violation.

There is one difficulty with the above approach; protection is essentially defined on the segment global to all other segments in the same computation; the computation being that entity which uses all the segments in the segment descriptor table. That is, any segment in the computation has the same access rights to a given segment as defined in that segment's descriptor. The difficulty can be seen using the example of a debugging program. The debugging segment requires read-write access to itself and to the program being debugged. The user program segment should have no access to the debugging segment and full access to itself. Both debugging segment and user program segment form a computation having one segment descriptor table. The question then is what access rights should be placed in the segment descriptor of the debugging segment? In the GE-645 system the issue is sidestepped through the design chosen for the total software protection scheme. The difficulty arose because protection was placed on the segment rather than the access path to the segment. The directed segmentation scheme of LeClerc[29)] places protection on the path by use of a protection connection matrix. A similar result could be achieved in a system such as the 645. A computation can have n possible segments. Each segment can have n possible access paths (including reference to itself). Thus, each segment may be assigned a protection vector consisting of n sets of capabilities. A segment will be given access to the jth segment in the computation's segment table in a manner defined by the jth entry in its protection vector. The protection vector can be stored in memory with the segment descriptor and implemented in the hardware map. It should be pointed out that on the present GE-645 the protection information in the segment descriptor for a segment is also entered in the hardware map when one of its pages is set up in the map.[19)]

Even with the flexible protection scheme existing in hardware on the GE-645, considerable software is required to implement the total protection system as we shall see in a later section. Some small additions to the hardware could reduce the software required as discussed later.

Control Protection

Certain instructions in the machine must not be generally available for any process to execute.18)19)30) For example, processes must be prevented from directly performing I/O instructions because other processes may be using a particular device. All attempts to perform I/O operations should go through the system so that these requests for service can be scheduled. Further, user processes must be prevented from halting the machine or executing any other instructions which might interfere with the system or other users.

To handle the above problem, machines are designed to operate in two modes, one usually called system mode and the other called user mode. In system mode, all instructions in the machine are executable. In user mode, certain classes of instructions, called privileged instructions are prohibited. If an attempt is made to execute a privileged instruction, a fault interrupt is generated which transfers control to the system. For example, in the SDS-940, the following instructions are privileged; all I/O instructions all instructions to control interrupts, all instructions to sense conditions of I/O devices or console switches, all unused operation codes, the halt instruction. This ability to interact with the system gives greater flexibility to the user in developing his own systems to run under the time-sharing system. Further, depending on the addressing scheme used, addressing may be different in the two modes. For example, in the SDS-940 there are two memory maps, one for the system routines and one for the user routines. A simple scheme exists in the 940 to allow the system to use the user map also, which facilitates communication between modes and simplifies the writing of reentrant routines.

In the above approach the mode classification is associated with a process. Other schemes have been suggested such as associating mode classification with areas of memory, thus tying the problem of privileged instructions to the memory protection system. Such a suggestion has the advantage of increasing flexibility in the software design.

This problem of communication between modes is one requiring careful consideration, because if the design of the mode switching scheme has not been carefully considered, programming can be awkward and time can be wasted in extra checking on the validity of the calls between modes.

The cleanest mode switching scheme known to us (from a programming and protection viewpoint) is that implemented on the SDS-940.26)30) This scheme utilizes programmed operators. The programmed operator concept is in effect a method for making subroutine calls logically appear to the user as machine instructions. When a bit in the instruction word signifying a programmed operator is detected, the bits which are normally interpreted as the operation code are then interpreted as an address to which control is transferred. Two types of programmed operator exist on the SDS-940: 1) a system programmed operator in which the transfer is made to locations in the system code and the mode is switched to system mode (the previous mode is saved and 2) a user programmed operator in which transfer is made to locations in the user's process. All calls to the system for assistance are thus made with system programmed operators. To return from system mode to user mode, the system executes a jump instruction addressed through the user map rather than the system map. While in system mode the system can access locations in user processes by use of the user map. Besides giving a simple interface between modes, the programmed operator concept allows the user to think he is programming a machine very different and more powerful than is provided by the bare hardware.

The mode switching mechanism used in the Multics system requires considerable overhead to check the validity of system calls. In Multics, shared system segments are assigned to each process and have entries in each users segment descriptor table. Transfers to the system are handled in the same way as normal intersegment references. The control bits of the system segment descriptors indicate that system segments operate in system mode.

The extra overhead involved in switching to system segments results because of the software implemented ring structure protection method implemented in Multics. The details are discussed in the section on software protection. As mentioned earlier, because the system segments use the same hardware map as the users segments, extra overhead involved in modification to this map result on each system call. The Multics mode switching mechanism is general and integral in the sense that system calls are treated uniformly with all intersegment references, but a price has been paid.

Interrupt System

Another important feature required by a resource sharing system is a priority interrupt system.19)30) The basic concept of the interrupt is very simple, but yet did not exist in developed form on most machines until third generation equipment. The CPU starts and stops various I/O processors and for some devices effects detailed control over the device. Without an interrupt system,the CPU spends much time testing sense bits associated with the devices to determine when they require service. With an interrupt system the CPU ignores the devices until an explicit signal is received from the device.

The simplest thing that happens when a device signals on an interrupt line is that the CPU branches to a fixed location and executes the instruction there,which in turn is usually a branch to a routine to handle the interrupt. This routine stores in temporary locations the contents of any CPU registers which it may need to use and then restores these registers when it is finished. A transfer is then made back to the program which was interrupted.

Most present systems have several interrupt lines with priorities associated with each, such that if an interrupt on level four occurs and before the routine associated with this level is finished an interrupt occurs on level two, then control is transferred to the routine associated with this higher priority level. When the level two routine finishes, control passes back to the level four routine which when finished returns control to the program originally interrupted.

The process of handling interrupts can be considered as the driving point of resource sharing systems. All the algorithms for scheduling I/O and resource allocation are started, stopped and conditioned by the various interrupts. All interrupts are explicit calls to the system for action or assistance.

A user program can be considered to be in one of three states: 1) ready to execute, 2) blocked waiting for some I/O or other process to finish or 3) running. When the system receives an interrupt, the effect is to change the state of one or more user programs and the system. There are two major classes of interrupts: 1) interrupts generated external to the CPU (I/O interrupts, clock interrupts are examples) and 2) interrupts generated within the CPU, examples being arithmetic overflow, or illegal instruction executions. Because these two classes require different handling in a multiprocess system a clear distinction is made between them in the GE-645 system, although this is not the case in most other systems.

Let us examine briefly four types of interrupts and see the type of effect they have on the system and programs.

I/O interrupts: These interrupts are generated by the successful or unsuccessful completion of an input/output operation. Such an interrupt is generated when a channel has completed transmission of a block of data or a device has completed some operations such as a tape rewind or disk arm positioning. In most systems, the interrupt level identifies the device requesting attention, but the exact meaning of the signal can only be determined by obtaining additional status information contained with the device itself.

Upon receipt of the signal and determination of the meaning of the signal several things can happen, some typical examples being:

1) a device may be ready to transmit information to or from an I/O buffer in system memory.

2) a device may be finished with a transmission which causes the device to be freed for other users and changes the status of a user from blocked to ready.

3) An error condition may occur requiring system attention.

Timer interrupt: Time sharing systems have attached to them one or more hardware clocks which generate an interrupt at fixed intervals. (The clock on the 940 generates an interrupt every 16 ms.) These timing marks are used for accounting purposes, as system protection, and as major input into the scheduling routines. The system protection uses are varied, but usually involve periodic checks of I/O operations to be sure that they are completed within expected times and that some hardware or software malfunction has not occurred which could tie up an important piece of equipment.

User program interrupts: Two subclasses of interrupts can be generated, one from arithmetic operations resulting in divide check, or arithmetic overflow or underflow, and the other from attempts to execute illegal instructions or reference illegal memory locations.

Hardware Failure interrupts: These interrupts result from hardware failures such as errors detected on data transmission, or power failure.

Accounting during interrupt processing is handled in two ways. In the SDS-940 system, the user whose program is running during an interrupt is charged for the interrupt processing time even if the interrupt is serving another users program. The justification for this approach is that over a given period of time the interrupt time required for each program will average out and thus the hardware-software expense required to sort out interrupt processing charges is not worth attempting.

In the GE-645 system on the other hand, interrupt processing time will be charged to the user whose program is being serviced.

An example of a machine with a weak interrupt system from a time-sharing point of view is the IBM 360/67. On the average, to isolate the source of an interrupt and transfer control to the appropriate routine requires over 300 microseconds.[34][35] Given the frequency of interrupts in a time-sharing system, particularly one with demand paging, such an overhead was one of the factors leading to the problems of the 360/67.

Reliability and Maintenance

To say that a time-sharing system should be reliable and easy to maintain is clearly to state the obvious. It is useful however, to briefly review some of the problems and approaches encountered in achieving these goals. When a system used for batch processing fails, little damage is done other than that of causing a delay in returning a job to a user. In a time-sharing system on the other hand, a user may have several hours of work invested at a console which can be destroyed if a system failure occurs. Failures can be of two types, hardware failures and software failures. The term crash is often used to refer to a failure.

Time-sharing systems (particularly the larger ones such as the 645) have considerably more hardware than comparable batch systems and thus high reliability is even harder to achieve. The very close coupling of hardware and software systems frequently make it difficult to determine where responsibility for trouble is to be found.

All machines are usually delivered with a set of diagnostic programs which exercise the hardware and can be used by maintenance engineers to locate trouble spots. These routines usually have required full control of the machine. In a time-sharing system, the goal is to have continuous availability of the system. In a single processor system the failure of the processor clearly takes the system out of usage until the proper repairs are made. However, there are many components of even a single processor system which if they fail do not require that the system be removed from usage from all users. The failure of a memory box or noncritical I/O gear are examples of such noncatastrophic failures In such a situation one desires diagnostic programs which can be run as special user programs and which enable the failure to be located and repaired without stopping the system. There are many classes of failures which are not permanent, but are intermittent or data dependent. These failures are difficult to isolate and may not require that the system be stopped if diagnostic routines can be run as special user programs. It is further desirable that hardware facilities be provided that allow maintenance engineers to place nonprogrammer accessible circuitry in test states in a way which does not interfere with the general operation as seen by other console users.

The most common conceptual approach to increasing reliability has been to provide a duplicated system as shown in Figure 16. In this system, duplicate information paths exist from I/O equipment to the two processors. Either processor can handle the system functions. The increase in performance of such dual systems is not well known as yet, but is not estimated to be over about 20%. This low performance increase is caused by the hardware and software interlocks which must be set to avoid the multiple processors interfering with each other. Performance increase is not usually the motivation for such systems, however. The idea is to enable at least a crippled system to continue to provide service in case of a failure.

Another type of duplication designed into single processor versions of IBM 360/67 and the GE-645 is multiple data paths to auxiliary store and peripheral devices. For example, one could expand Figure 16 to show a path from each GIOC to each device. Thus, if one GIOC goes down another path to the device exists. Similarly each device could have two controllers. When a system module fails, the operator can reconfigure the system by altering appropriate system tables and manual switches and the device can be repaired. This idea of removing failed units also applys to memory banks as well.

The use of error detection and correction codes for data transfers between main memory and auxiliary storage and peripherals is commonly used. The use of these codes to protect the memory operations is increasing used as well to protect data transfers within the processors. Use of codes which are preserved under arithmetic operations is also becoming a common practice.

Practical versions of these codes can usually detect an error in one or more bit positions of a unit of information such as a character or full word. Correction of one bit in error is also practical.

As systems have increased in complexity, increasing attention is being given to designing features for increased reliability and ease of maintenance into the system from the earliest design period.

The hardware concepts introduced in the sections above should enable the reader to understand the more detailed literature in the resource sharing system design area. The very important area of communications technology has not been covered. An introduction to this area is contained in references.38)39) Some additional hardware concepts are introduced in the discussion of software concepts where it seemed more appropriate for clarity to link the two areas more closely together.

Software Concepts

There are three major parts of a time-sharing software system: 1) the routines which must permanently reside in main memory because they are used frequently or must react quickly to the changing demands for system resources, 2) the routines which are swapped in and out of main storage as part of the system, or are assigned to user processes which perform communication between the system and the user, and perform infrequently required system functions and 3) the subsystems such as compilers, text editors, debugging aids, mathematical routines and other user oriented routines. The resident routines handle requests for service of I/O devices, schedule processes, allocate memory and I/O devices, handle communications with user terminals and all other tasks which occur at a great enough frequency or require sufficiently fast response. Associated with these resident routines are many tables containing status information about user processes and system resources.

The swapped system routines are associated with less frequently required services such as allocation of subsystems, logging users in and out of the system, accounting, and opening and closing of files.

The subsystems are swapped and, along with other system routines, are written as reentrant routines.

In many time-shared systems, methods for allowing users to logically incorporate system facilities within their own programs are available and therefore there may be very fuzzy lines between the categories given above.

In this report little is said about the subsystems themselves except to discuss briefly how they are linked to the rest of the system. Primary attention is given to a discussion of software concepts usually found in the resident routines. Because of the close interaction of software and hardware in a time-sharing system, many software concepts were introduced during earlier sections.

The Resident Routines

The central problem is the following. Only one process can be executing at a time in a single processor system, although I/O for other processes in or out of various buffers may be going on concurrently (the problem of intercommunication in a multi-processor system is beyond the scope of this report). A scheduling process decides when programs are to be blocked or started. When the system is to begin execution of a different process, allocation of main memory must take place. In a demand paging scheme such as that of the IBM 360/67 or Multics systems, memory allocation may take place more frequently and in smaller pieces, but the problems are similar to those encountered in a system such as the SDS-940 where an entire process and data reside in main memory at the time execution begins.

Processes in the system are created by setting up a state vector for the process. The point of creation may be at the point a user logs into the system from a terminal or may take place invisible to the user. A user job may consist of several processes which in effect operate in parallel with one another, each of which has its own state vector and may in turn create other processes. One of the problems in the design of the software system is to develop methods of communication among related processes.

Scheduling

In the terminology of the MIT time-sharing group, a process can be in one of three states, running, ready to run, or blocked. A blocked process is one which cannot run until some signal arrives to unblock it. These unblocking signals are referred to as wakeup signals and change the status of a process from blocked to ready to run. A signal which causes one process to be unblocked may also cause another to be blocked. These signals can come from many sources, such as the terminal communications equipment, a timer, completion of a disk seek, or execution of an instruction with an illegal memory address. In general these signals arrive through the hardware interrupt system, but may also be generated by the system or other processes which generate such a signal on the basis of a value stored in a register or memory location. The latter source of signals is the main one used for interprocess communications.

These wakeup signals are sent along with related status information to a part of the system often called the scheduler.[27)] In the Multics system the term traffic controller is used.[46)] One of the defining differences between classes of multiprogramming systems, besides the range of available facilities, is the design of the scheduler. In an on-line file management system or other I/O oriented system, the prime wakeup signals are associated with the I/O devices, while in a general purpose time-sharing system the signals from clock sources are given increased importance. From the point of view of the system, the life history of a process is an alternation of running and blocked periods. Each running period is terminated by a signal to block this process. This blocking signal may be made by the process itself, a system process or some other user process. A simple example is a request by a process for a character from an I/O device. If the character is not available, then the process may be blocked until the arrival of the character. The request for the character may wakeup a system routine to refill a buffer from the required I/O device. During the discussion of the handling of various specific resources, additional examples of signals for blocking and wakeup are given.

There are two main approaches to the communication of blocking and wakeup signals. In one approach the process generating the signal must be explicitly aware of all the processes affected. In the other approach the process generating the signal need not be explicitly aware of which processes are affected, but only need perform simple actions such as incrementing a certain memory cell. The latter approach is taken with the SDS-940 system.[27)] For example, when a process is blocked a word in its state vector is set indicating the reason for it being blocked. This word also has an address to be checked by the scheduler and bits indicating what condition the scheduler should find which would awaken the process. The I/O or other routine which is to awaken the process at the completion of some action does not know explicitly which process it is servicing, but does know to set a particular location when it is finished.

The former approach is more natural and the one primarily used in hardware communication between modules, but requires more code when implemented in software. The latter approach results in code which may be easier to modify because of the decoupling possible between system modules, but requires extra time to determine which process to run next, because the appropriate scheduling routine must check one or more cells designated by each process in some order until it finds a process to run.

In the discussion on scheduling up to this point,we have ignored the fact that the number of processors in a system is generally going to be much smaller than the number of unblocked processes. Therefore, it is necessary to have some algorithm for deciding which, of the several processes that are ready to run, should be run next. This algorithm is often called the scheduling algorithm. The scheduling algorithm is an important part of the scheduler, but as we have seen above is by no means the whole of the scheduler.

The criteria for the design of a scheduling algorithm are:[27)]

1) to minimize the effort expended by the system in switching processes,

2) to be able to react rapidly to the changing collection of processes ready to run so that the most important process is being run at every instant.

There is much discussion in the literature about what features a scheduling algorithm should have to meet the above criteria.[5)] However, the simplified mathematical models used as a basis for many of these studies are quite removed from the features of real systems. The use of system simulation as a means for adjusting scheduling algorithms to particular machine configurations and user environments is slowly growing and does lead to useful results.

Many of the algorithms used on current machines were designed based on the designers intuitions and are far from optimal. The goals of designing a scheduling algorithm which is both responsive and inexpensive of system resources are contradictory and therefore, tradeoffs in both areas are required for practical algorithms.

Basic to any scheduling algorithm is some scheme for keeping track of the processes which are ready to run. A list of such processes is called a ready list. The ready list has a structure which can be quite complex or very simple depending on the algorithm design. The simplest list is one in which a process ready to run is placed on the end of the ready list and the processes are run on a first come-first served basis.

Once a process is started running, it cannot be allowed to run as long as it might require, if responsiveness to all users is to be maintained. Therefore, one of the important sources of blocking signals is a clock source. A clock source is a device for producing an interrupt at some fixed frequency which can be used to increment registers for recording elapsed time. The length of time which a process is allowed to run before it is blocked is called a quantum. A quantum size is one of the important parameters of a scheduling algorithm. The quantum size may be fixed or variable in size depending on some other parameters such as process size, process priority, or the length of time the process ran last time.

A more complex ready list structure than mentioned above splits the ready list into sections such that the various sections have different priorities. Thus, at the point where a process must be chosen to run, the scheduling algorithm always starts looking at the highest priority section first. The processes on the lower priority levels only get run if there are no processes ready to run on the higher priority levels. There is always the possibility, unless additional features are added to the scheme, that some processes on the lower priority levels may not get run or may run so infrequently that the response is not acceptable.

One scheme to avoid this problem has been incorporated in the SDS-940 system. The ready list is broken up into four parts. The highest priority level is assigned to processes which were blocked waiting for teletype input/output, the second priority level is assigned to processes blocked for other I/O, the third level is assigned to processes blocked at the end of a short quantum and the fourth level is assigned to processes blocked at the end of a long quantum. (The scheduling algorithm uses two quantum numbers. One quantum number is called a short quantum and the other a long quantum.) A process is always allowed to run for a short quantum and if at the end of this time no other process is ready to run it can continue. During the time a process is running the number stored in the machine associated with this process' long quantum is decremented on each clock pulse. When the process is blocked the present value of the long quantum is stored and the decrementing continued next time the process is run. Thus, eventually the long quantum runs out and the process is moved to the lowest priority queue. This method insures that all processes will run with reasonable respone to each. This scheme is just one of many which could be devised to limit the number of times a process can appear on the high priority levels of the ready list. Additional complexity can be added to a scheduling algorithm to take into account maximum utilization of memory or any other resource. In the previous discussion of swapping we indicated that scheduling algorithms could not be designed independently of the algorithms chosen for swapping and memory allocation. All the algorithms must be chosen to reflect the performance characteristics of the hardware and to some extent the characteristics of the user community at a particular installation.

The above discussion of scheduling has applied primarily to user processes. The scheduler must also control system processes. The system processes are blocked and unblocked by interrupts, by requests for service by user processes, and by status information from I/O devices. Most of the system activity is performed by resident routines which do not require swapping. When an interrupt or user request arrives, control is transferred to the appropriate routine, the required action is performed, and control is returned to the user process. The system may have ready lists of its own to handle requests for I/O and other services on devices which can be used by only one process at a time.

When a system routine is required which does not normally reside in main memory, a state vector for this process may be set up and this process placed on the ready list. In the SDS-940 system, for example, the major part of the nonresident system is considered a process which is assigned to all users. It is written as reentrant code so that only one copy is required in main memory which is shared by all the users. In the Multics system, system segments are assigned as part of each users computation.

A page of memory invisible to the user is also assigned to each user job in the 940 system and is swapped as part of the user's processes. This page of memory is used for constants, tables, I/O buffers and temporary storage required by the system during the running of the user's processes. The nature of some of the tables in the linkage page is discussed in the sections below.

Besides the priorities assigned to various processes, the scheduler must superimpose priorities associated with a balanced utilization of system resources. For example, when swapping is involved and considerable system resources are expended in bringing a process into main memory, the scheduler must maintain some deferral power over incoming blocking signals to avoid situations where a process is brought into main memory only to be immediately blocked. To complicate the discussion further, the operator and user processors in some systems (the SDS-940 being an example) can interact with the system to alter scheduling priorities or in other ways directly affect the scheduling process. This effect is accomplished by creation of sub-processes or by giving commands to the scheduler.

The scheduling problem is highly complex with many possible approaches available and parameters to be determined. A scheduling algorithm which may be appropriate for one configuration of equipment or for one installation and set of users may not perform satisfactorily for another.

Hopefully manufacturers of time-sharing systems can design their systems and develop accurate simulators which can aid in the choice of some of the parameters to match a given installation.

Memory Allocation

The memory allocation problem can be stated quite simply. To be able to perform memory allocation the system must know which processes are in main memory and where, and the system must know which areas of main memory are free. Assuming more than one process is residing in memory at a given time and that a process residing on auxiliary storate is to be brought into main memory, then the system must decide which process(es) or parts of processes are to be moved to auxiliary storage to make room for the new process about to be started. The system must also know where on auxiliary storage to place the processes to be removed.

The exact details of how the above problems are solved naturally varies from system to system, but there are common features in the various implementation approaches, because the main concepts involved are in the nature of the tables required. At this point we discuss the memory allocation scheme used in the SDS-940 because, if the reader understands the basic approach used there, then he has a good basis for understanding other systems. Also many of the memory allocation concepts of the Multics and IBM 360/67 systems were discussed earlier. To understand the following discussion the reader should be familiar with the SDS-940 paging mechanism and relabeling (map) registers described earlier.

When a process is running, the actual physical locations in main memory in which the logical pages of the program are stored are indicated by the real relabeling registers as described earlier. To determine the physical blocks in main memory to be assigned to an incoming process, the memory allocation routines work with two tables. One table is called the real memory table (RMT) which contains information about which logical pages of which processes are using the various physical blocks of main memory. The other table is called the real memory counter (RMC) which contains information about the number of processes using the code stored in the various physical blocks. For example, the text editing system, which is reentrant, may be being used by several users and thus efficiencies are gained if it is not swapped if possible. Additional counters are used to indicate frequency of physical block usage to assist in determining which blocks should be selected for assignment to the incoming process. The actual pages associated with a job, possibly containing several processes, are kept track of in a table called the program memory table (PMT). There is a word of storage in the PMT for each of the users logical pages. This word indicates where that page is located in physical storage (main memory or auxiliary storage). When the page is in main memory this word indicates the starting physical address in which it is stored; when it is on auxiliary storage the physical address is given here. The RMT does not actually contain explicit information, but pointers back into the PMT. When the memory allocation routines determine that certain physical blocks are to be assigned to an incoming process, then any information of value in them must be removed to auxiliary storage. By use of the RMT the system can reset the address in the proper PMT entry to indicate the auxiliary storage location to which the page has been removed.

Within the PMT there is no necessary ordering; that is, the position of the entries in the PMT does not necessarily correspond to the logical ordering of pages in a process, although all pages belonging to one user are in contiguous PMT cells. The logical ordering is stored in two words of the state vector. These words have the same format as the real relabeling registers and are referred to as pseudo relabeling registers. The difference between them is that the pseudo relabeling registers point not to real memory, but to entries in the PMT. The order of bytes in the pseudo relabeling registers is the logical order of the program's pages. That is, byte 1 points to an entry in the PMT where information about logical page 1 is stored, and so on. The use of the above mentioned tables is shown in Figure 23. Through calls to the system, processes can change the contents of the pseudo relabeling registers and in this way perform overlays.

In capsule form the following events take place when a process is blocked and another process is to be run.

1) A check is made using the PMT to determine if the process to run is already in main memory and if not how many physical blocks are required. If the new process resides in memory already, transfer of control takes place immediately.

2) Using the RMT, RMC and usage routines, the memory allocation routines determine which blocks are to be freed.

3) Using the pseudo relabeling registers and the PMT, the location and logical order of the pages to be brought into main memory are determined.

4) I/O commands for transferring pages into and out of memory are prepared and the swap takes place.

5) The real relabeling registers are set up.

6) The machine registers are restored from the values stored in the state vector and the process is started.

The methods for allocating space on the auxiliary storage device (a drum in the case of the SDS-940) are important concepts and are discussed below.

To make the swapping process more efficient, use can be made of the fact that some pages of processes are read only and of the fact that not all pages contain locations which are modified when a process is run. Read only and unmodified pages do not need to be returned to auxiliary storage during a swap because copies of these pages already exist there. On the SDS-940 the above concept is implemented by use of software and the hardware memory protection system. When the real relabeling registers for a process are set up all pages have the read only bit set. A software code exists in the PMT

Figure 23 follows

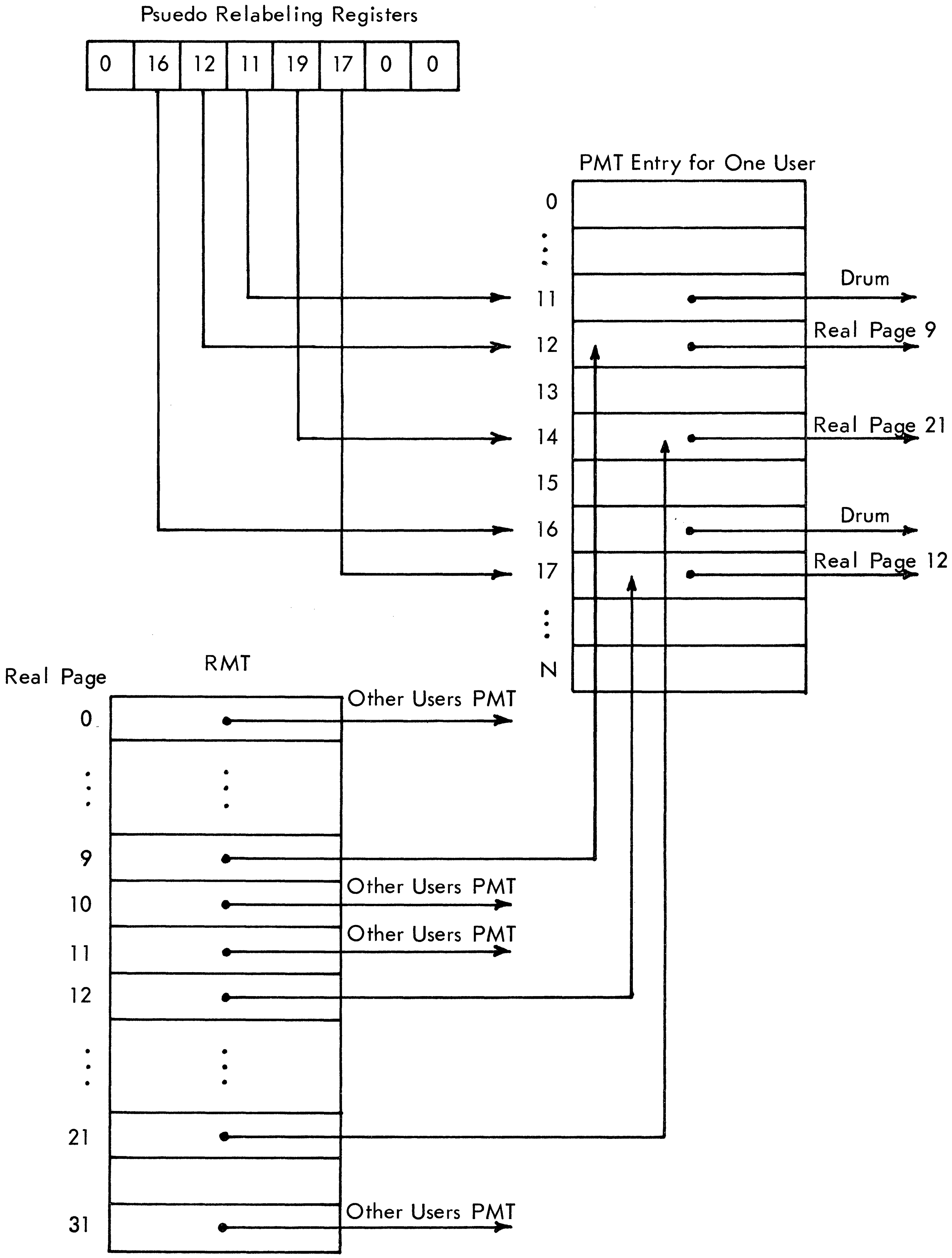


Figure 23. RELATION OF MEMORY ALLOCATION TABLES

249-68
64692

indicating whether or not the page is really read only or not. Then when the process attempts to write into the page a memory fault interrupt is generated. The system checks to see whether or not the page is truly read only. If the page is not, a bit in the PMT is set to indicate that this page has been written into, the read only bit in the relabeling register for this page is cleared, and the write instruction allowed to proceed.

In the 940 system the user can interact directly with the memory allocation routines to change the PMT entries or the pseudo relabeling registers. This interaction ability aids in the design of overlays and can lead to increased system efficiency.

The Multics and IBM 360/67 system have mechanisms for memory allocation similar to, but more general than those mentioned above. These systems maintain tables equivalent to the real memory table of the SDS-940 system. Mechanisms for determining which pages to replace also exist. The role played in the SDS-940 by the pseudo relabeling registers and PMT in defining a users logical space and inventory of currently loaded pages is played by the segment descriptor and paging tables in a more general way in Multics and the IBM 360/67, as was discussed in some detail earlier.

Teletype and Other Terminals

In this report we only consider character oriented terminals. In a separate report (53) a detailed discussion is given of a general purpose graphic terminal. There are two main approaches to terminal communication: one approach transmits on a single character basis and the other approach transmits a block of characters at a time. The former approach gives the user greater flexibility than the latter in designing interactive systems, but requires more system time in character handling. The SDS-940 system uses the single character approach for teletype input-output. Many of the character oriented graphic terminals require block transfers. The IBM Quiktran system transmits on a line at a time basis. The software handling of either approach has much in common and only the single character approach is discussed here.

On the SDS-940, the communications interface is connected to the direct I/O lines and each character input or output requires about 300 microseconds of processing by the system. An alternative approach commonly used, and the one used on the GE-645, is to input characters directly into memory through a multiplex channel. For a system such as the 940, which usually has 16-32 teletypes attached, the overhead involved in handling the terminal I/O is not excessive. For large systems using 50-200 terminals this approach is to inefficient. Designers of such systems are beginning to use separate small processors to handle the terminal communications through their own direct access to main memory. Efficient instructions for packing and unpacking characters from machine words are also important in decreasing the system effort expended in character handling. The SDS-940 does not have such instructions.

The wide range of available character oriented terminal devices and range of data rates associated with these devices make it essential that the terminal interface hardware and software be modular in design. Devices are available with varying character sets which must be recognized by software routines. Each device may have special control codes for its operation. Data rates of available devices range between 10-240 characters per second. Systems such as the SDS-940, which were designed for teletypes as the main terminal, have required extensive modifications to the software routines to add other terminals and modifications to the hardware interface equipment to accomodate the higher data rates of character oriented graphics terminals. It is to be expected that future systems will be more modular in design so that a variety of terminals can be used with few if any system modifications.

The important concept to understand, about I/O in general as well as terminal I/O, is that the user program never communicates directly with the I/O device, but rather communicates with a buffer (a buffer is a storage area in main core usually maintained by the system). Large block transfers to or from secondary storage using a tape or disc may take place from user memory and this case is discussed in the section on general I/O. In the case of terminal I/O, when the user program issues an instruction which is to send a character to a terminal, the system interpretively executes this instruction (a system call is actually generated by use of a system programmed operator as discussed earlier and the system performs the required function) and places the character to be sent to the terminal in an output buffer associated with this process. Later, other system routines which are "aware" of the I/O activity to all terminals can remove the characters from the buffers and send them to the terminals. If a process has a great deal of output to the terminal it could overflow the buffer. The system watches the load level of the output buffer and, if it is about to be filled, the process is blocked until the buffer has been emptied.

On input, the characters from the terminal are automatically loaded by the system into an input buffer. When a process is executing and issues an input character instruction, it is interpretively executed by the system which takes a character from the input buffer and places it in a register assigned by convention or in a user designated core location. If no characters are in the buffer, the process is blocked until a sufficient number of characters arrive in the buffer.

The user may have written his process to accept single character commands. In this case setting some limit on the number of characters which must be in the buffer before the process is unblocked would be unworkable. The SDS-940 and other systems use a concept of <u>break characters</u> to get around this problem. Certain characters or possibly all characters (the number and extent of the break character set is set by commands which the user can give the system) have the significance that when they are received, the process is unblocked so that when its turn for execution comes up it can obtain the character from the buffer. All characters are entered into the buffer and when the input buffer reaches a particular limit the users program is unblocked so that within a short time it will be placed in execution and can empty the buffer.

Another concept associated with terminal I/O is called echoing in the 940 system and is found on other systems as well. This mechanism makes use of the fact that terminal send and receive units are usually logically separate from one another, although they may exist in the same physical frame. In the echoing approach, when the user presses a key the character code goes into the central computer and is not printed on the terminal by the user's action. The system accepts the character and then puts it in a buffer separate from the other output (so as not to mix it with the process generated output). The system then outputs the echo buffer. Teletypes have a switch which could directly connect the input and output mechanisms together so that when a key is pressed the character is printed. The question then is why go through the more expensive echo procedure? There are two main advantages:

1) the echoing process gives an error detection ability. If the correct characters come back, then the user knows that the characters were correctly received. The usefulness of this type of assurance for data input is clear.

2) the echoing process gives a code conversion or translation capability by which a user can send in one character and receive the same, a different, or no character back.

Another feature available on many systems allows terminals to be linked together on input or output or both. There are many uses for such a feature when instruction is taking place or two or more people are working together on a problem.

System response can be significantly speeded up if the teletype routines handle simple editing chores rather than require the various subsystems to have separate editing routines. Thus, when a user hits the wrong key and then sends an editing character to cancel the last character or entire line, this frequently required editing task can be performed immediately without requiring the expense of swapping in a subsystem.

General I/O

In this section, we discuss briefly the scheduling of access to I/O devices. In the next section, a discussion is given of some of the problems associated with allocation of storage space on direct access devices such as disks or drums. Most I/O devices are not easily shared among users. These resources are more effectively dedicated to a specific user for a given time period, despite the fact that the user may not make maximum use of the resource. Devices in this category are tape drives and reels, line or other printers, card equipment and optical character scanning equipment. Devices such as disks and drums, while only allowing access by one process at a time, can rapidly be switched from process to process and thus are not considered dedicated devices.

Two major functions must be performed in handling I/O devices.

1) reserving and allocating these resources[13)]

2) protection of the resource dedicated to one user from interference by another user.[38]

During transfer operations, proper handling of the main memory buffer areas is also required.

A user can explicitly ask for access to a device from a terminal or his process may issue an I/O request. An example of the former is a request to transfer a character file from storage on the disk to a line printer. In order to efficiently utilize the I/O devices, the system maintains buffer areas which can be allocated to the different devices. On input, for example, the system attempts to maintain a buffer full of information so that when a process requests information from a device, the system has anticipated the request and the information is in main storage. Thus the process receives the required information and can keep executing. Without the buffering, the process would have had to be blocked and the system might have to swap in another process while the required information is brought into main storage. On output the system fills a buffer and automatically outputs it to the required device. When the buffer is full the process is blocked until it is emptied. There are two types of records maintained by the system 1) logical records and 2) physical records. A file is made up of one or more logical records. Logical records may be of arbitrary length and structure and are process dependent. Physical records are of fixed length appropriate to the particular device. Logical records can be made up of one or more physical records. If the logical records are smaller than the physical records, more than one logical record will be packed in a physical record to increase I/O transfer and storage efficiency. Further discussion of this area is given in the section on the file system. The buffer size is equal to the size of a physical record and is variable depending on the device.

A user may want to transmit blocks of information larger than the standard physical records and thus define his own physical records. In this case the buffering is handled directly within the user memory. Earlier it was mentioned that a page of memory not directly accessible to the user was swapped with the user process in the SDS-940. The system-maintained buffers are kept in this area. If a user requests a word of data from an I/O device which is not available in a buffer, the process is blocked, but the page of memory containing the buffer is locked in memory until the buffer has been filled. Similarly, if the process attempts to output to a buffer which is full the process is blocked until the buffer is emptied. The page containing the buffer is kept in main memory even though the rest of the process is swapped out. Identical actions are taken if the buffer is maintained by the user in his accessible memory.

The above discussion assumed that the I/O device to or from which information was being transferred had been allocated to the process. This allocation takes place through a process called <u>opening</u> a device. When a process is finished with the device, the device is <u>closed</u>. These functions are handled by the system routines invisible to the user processes. When the system attempts to open a device for a process it checks a table containing information about the particular device. The name device control blocks is often given to these tables. If the device is presently in use, the table will indicate this fact. When the device is in use two things can happen depending on the system design, 1) the process may be dismissed and required to replace its request when its turn in the ready list arrives, or 2) a request for this device can be placed on a queue and the process blocked until the device can be assigned to this process.

Assuming that the device is free it is assigned to the requesting process and it is opened for the process. The tables associated with the device are consulted to determine the size of buffer required as well as other device dependent information. A file control block is set up associated with this transfer which contains all device and process dependent information required for the generalized I/O and interrupt handling routines to actually perform the required I/O. Associated with the I/O transfer is a file name which is the users handle on the transfer. This file name is translated to a number by the system which is used to gain access to the correct file control block for the transfer. Protection of the device is gained because all processes must request service from the system, which can then check to see if a device is free.

Allocation of Space on Direct Access Devices

As mentioned above, the user works with logical records and the system works with physical records. On a magnetic tape system, the physical records are arranged sequentially on the tape. Tape files have the disadvantage that they are difficult to change by addition or deletion without rewriting the entire file. The characteristics of direct access devices make it easy to modify parts of a file without changing the remainder of the file. Besides this flexibility available for use in file updating, direct access devices can also give rapid access to the required information. The organization of the storage space on these devices to yield these advantages can be handled in many ways. We discuss one set of techniques here to give a feeling for the concepts involved. Our discussion applies to disk units and to drums. A disk unit is made up of a stack of plates coated with a magnetic material. These plates are constantly rotating. The plates are organized into concentric circles called tracks on which information is stored. The tracks are also commonly broken up into sections called sectors. In most systems the minimum quantity of information which can be transferred is a sector. The organization is shown in Figure 24. There are two types of disk unit. One type has a read/write head associated with each track so that access to a particular track can be gained at electronic speeds. The other type has one or more head/arm assemblies which must be mechanically positioned to the required track. This positioning operation is called a seek. Time required to perform a seek ranges upwards from about 75 milliseconds depending on the number of head/arm assemblies, and on disk size. To address information on the disk, the unit must be sent a plate, track and sector address. Files of information are organized on the disk in physical records equal in size to a sector. The physical records can be stored anywhere on the disk, although for transfer and seek efficiency many systems try to store the physical records in contiguous sectors. There are two problems that have to be solved: one is to keep track of the locations and proper sequencing of physical records on

Figure 24 follows

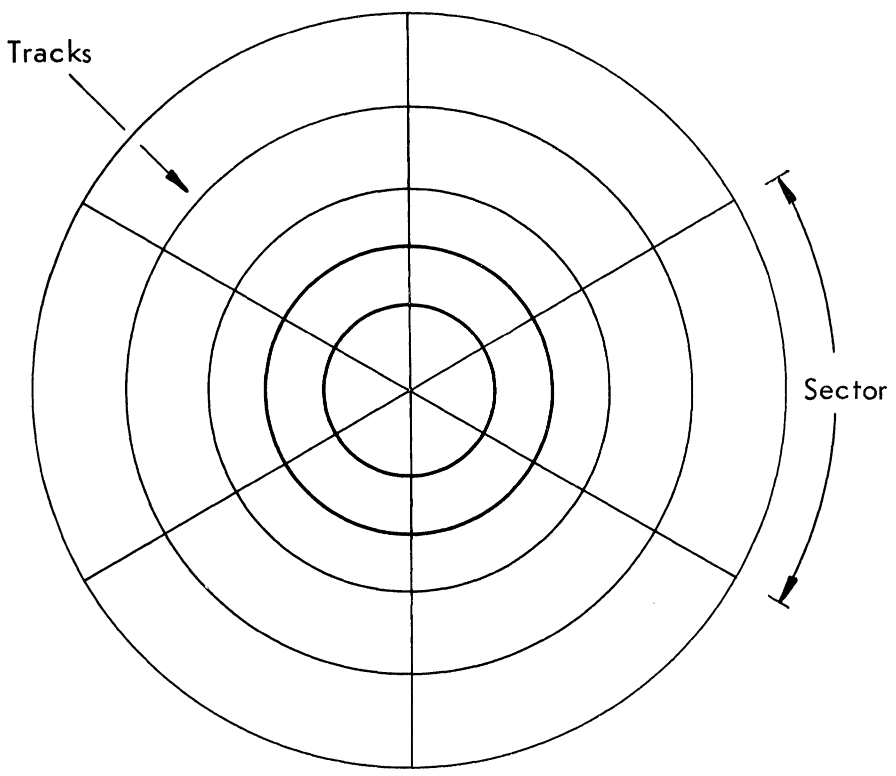


Figure 24. ORGANIZATION OF A DISK SURFACE

the disk and the other is to keep track of free space which can be used for new files and the expansion of old files. In order to prevent a few users from using all available space, restrictions on the space individual users can take are also required.

There are many trade-offs available with regard to storage space used to keep track of where files are stored, available free space, and the time (measured in number of disk accesses) required to access a file. The technique used on the SDS-940 for keeping track of the physical records of a file uses what are called index blocks. An index block is nothing more than a table of contiguous words, the i^{th} of which has the address of the physical record for the i^{th} logical record in the file sequence. A file may be of such size that several index blocks are required. These blocks can be chained together by using one word in the index block to contain the address of the next index block in sequence. The index blocks are kept on the disk and are usually a sector in length. The address of the first index block is maintained in a table called a file directory to be discussed later.

To keep track of free space many methods could be used depending on the size of the disk file, acceptable time to find a free sector and so forth. One method is to chain free sectors together which is simply done and takes little main storage space, but does require access time if the system is to obtain more than one sector. Another method is to use a bit map. In the bit map a sequence of words large enough so that there is one bit available for each sector is set aside. The bits are set if the corresponding sectors are free and cleared if they are in use. Given the bit position in the map, simple calculations can determine the corresponding physical address of the sector on the disk. This procedure works well for devices with limited capacity, but requires excessive storage for the map if used with large files.

This entire subject of direct access device organization is very important and a detailed presentation of concepts in this area is contained in reference (25).

The File System[a)]

What is a file? A file is a collection of related information with a name. Files are stored in a computer system as a string of elements, bits, characters, computer words and so forth. Examples of files are procedures, strings of text, arrays of numbers. A file has all the characteristics given in the definition of a segment. And in fact, the words "file" and "segment" are interchangeable in the Multics system. The Multics system is designed to treat files in a very general and uniform manner. We have seen some of the costs associated with achieving this goal. The SDS-940 system does not treat all files in a uniform manner, but does in the end offer the user the main functions he desires in a file system, although at a cost to the user of more explicit interaction with the system.

a) The author wishes to acknowledge his debt to M. D. Kudlick for discussions on the relationship between the Multics and SDS-940 file systems.

Because the characteristics of the file system are of such importance to the user, a separate report on time-sharing file systems is in preparation (21). The discussion given here introduces many of the basic concepts and ties together the concepts related to files with those introduced previously for addressing and memory allocation. The need for a file system arises because of the limitations on the size of main memory. Because of these limitations auxiliary storate is required in the form of disks, drums, tape strip, paper tape, magnetic tape, cards, photo stores and so forth.

The file system is that portion of the total memory management system dealing primarily with auxiliary storage. The file system must interface smoothly with the schemes used for addressing and allocation of main memory. In systems such as Multics where the logical address space is very large, the total memory management system is the file system. That is, Multics is designed so that the programmer thinks that he is using a very large main memory (his logical address space) and all movements of information between auxiliary storage devices and main memory are invisible to him. In the SDS-940 system, on the other hand, the user must deal explicitly with the file system. We see in the discussion to follow some of the advantages and disadvantages of the two approaches and the type of cost generality trade offs available.

A general purpose time-sharing file system should satisfy the following requirements.

1) The user should be able to create, change and delete files.

2) The users should be able to access each others files, inorder to build on each others work.

3) The user should be able to control who has access to his files and the type of access allowed such as read, write, and execute.

4) The user should be able to structure his files in a form appropriate to his problem.

5) The user needs to communicate information between files.

Let us examine how the above abilities are achieved in the SDS-940 and Multics systems. The two systems approach the problem of sharing files and allowing users to access each others files in very different ways. Multics, as we have seen, treats all files as segments and all segments are treated in a uniform manner. All segments can be relocated in logical space and thus can be shared with complete generality, with only one copy of each residing in the system. In the SDS-940 system, the designers realized that, based on frequency and type of usage, there are different classes of files and that system economies can be achieved if these classes are treated differently.

One of the Multics design goals was to remove all concern for storage management from the user. There are in fact several classes of users, which might be classified as application programmers, system programmers developing compilers and other tools which are application oriented, and system programmers expanding and maintaining the supervisor portion of the system. Multics has succeeded in removing many storage management burdens from the application programmers, but has placed very many additional burdens on the last two classes of users. One of the original Multics design goals had been to remove many of the storage management conventions and concerns from system programmers as well as the application programmers, but as implementation proceeded, it was apparently found impossible to meet this goal. This failure is unfortunate because one of the uses of a large scale time-sharing system is for groups of users such as chemical or other engineers to develop large machine aided design systems which will need special languages. Further, inorder to run efficiently these systems will need to use special systems programming to get around the intersegment linking costs described earlier. To implement large machine aided design systems on a time-sharing system with a design philosophy similar to that of the SDS-940 would also require system programming. The important point to be made is that generality has an associated cost. Unfortunately the cost does not seem to be a linear function of generality. The SDS-940 system requires some awareness of storage management on the part of the application programmer, but much less knowledge is required of the other classes of users.

In the Multics system a file is a segment and therefore any file in the system can be directly addressed by name and location. The system locates the file by use of tables called file directories to be discussed late later. If the user addressing the segment is allowed access, it is made known to him as was discussed earlier. No explicit file instructions are required to access the file. All the instructions of the machine which address operands can access any file (segment) in the system. The user of Multics can create a file just by performing output to a new segment which he names. The user deletes a file by explicit command to the system. All procedures in Multics are written as reentrant and therefore can be fully shared, with only one copy existing in the system. All compilers or other "subsystems" are segments in Multics and treated in a uniform manner with all other system and user segments. Data segments can be fully shared even if they contain address pointers for structuring purposes.[25)] Only one copy of such common data bases are required in the system and no position conflicts result in the logical space of the users.

The design goals of the Multics file system are very desirable if they could be economically achieved. We have spent considerable space in this paper pointing out the costs associated with achieving these goals because they have not been fully reported in the literature. The design of Multics can be looked upon as an attempt to achieve a certain level of theoretical purity and as such many valuable insights have resulted from this effort.

The designers of the SDS-940 took a more pragmatic engineering approach and sacrificed generality wherever frequency of use and associated cost did not seem to merit it. The fact that the SDS-940 is a much less powerful computer than the GE-645 should not obscure the point we would like to make here, namely that a more progmatic philosophy is also possible on systems larger than the SDS-940.

There are three classes of files in the SDS-940 system:

1) Files which can be shared by multiple users with only one copy existing in the system. These files are all procedures such as compilers, text editors, debugging routines and assemblers and are called "subsystems". At present there is no provision for sharing single copies of data files and later we indicate how such sharing would take place when the need develops.

2) Files which all users can access by acquiring a separate copy. These files are called "system library files".

3) Files which are semi-private to each user.

Each of the above class of file is accessed in a slightly different manner.

Subsystems are accessed by command from the terminal to the SDS-940 system. The system fetches the subsystem, assigns it space in the users program memory table, sets up pseudo relabeling registers for the subsystem, and finally transfers control to the subsystem. The subsystem may then ask for the names of additional files it may require. An example is a compiler requesting the name of the file containing symbolic source code input and the name of the file to output the compiled binary code. Only one subsystem at a time can be running for a given user. This latter restriction on generality is not serious considering the nature of the procedures running as subsystems. This restriction is one of the cost generality trade offs available.

The library file mechanism allows users to build on each others work and concurrently use multiple procedures developed by others. The price paid with this mechanism is possibly that of introducing multiple copies of a given procedure into main memory. The designers felt that the frequency was low with which a given library file would be used concurrently by several processes and therefore, introducing the mechanisms to allow sharing of one copy with full generality was not worth the associated costs. If because of the nature of the work at a particular installation a given set of library routines should frequently be used concurrently by several processes, then these routines could be made into a subsystem. The amount of effort required to make such a conversion depends on whether or not the procedures were original written with such a possibility in mind. No conflicts in positioning in logical space result through use of multiple library routines by a single process, because since each process using a given library routine has its own copy, the library routine can be loaded anywhere in the processes logical space.

A user can access another users semi-private files, if he has permission, by commanding the system to copy the desired file to his file space. The result is that two copies of the file exist in the system. Again ease of system development was traded for slightly extra effort on the user's part and some waste of secondary storage space. However, to make a more general mechanism as in Multics requires a large resident system, over 50×10^3 words, and an even larger non resident system. So there is a trade off between user file and memory requirements and system file and memory requirements. When a user develops a file which is to be used frequently by others, then it can be easily made a library file or with more effort converted to a subsystem.

The case of sharing one copy of data files needs to be discussed. At present the SDS-940 system contains no mechanism for this purpose. Data files can be accessed by multiple users only through the library or semi-private file mechanism. Paging as we saw earlier, placed no restrictions on sharing of single copies of data if the data did not contain addresses. If the data contains address pointers for structuring purposes, then such shared data would have to be placed in the same position in each users map which accesses it. Such data structures require special programs to interput the structure, however, and there seem few reasons why the subsystem mechanism could not be generalized at the cost of system development effort to handle such cases if the need should develop. Users wishing to develop special programs to transform such data would have to get the subsystem to create an intermediate file which their special programs could access. Another approach would be to use many of the linkage techniques of Multics, such as linkage segments, for special subsystems. The SDS-940 and Multics system represent two poles in design philosophy and there are many trade offs possible between them.

We have discussed problems related to sharing of files and now indicate briefly how a user communicates between files. The user of Multics can communicate directly between files just by addressing data files with the normal machine load and store instructions or by transferring between procedure files by normal machine transfer instructions. The linkage process was discussed earlier. The user of the SDS-940 system must explicitly "open" the required data files with system calls as discussed earlier and then input and output to the files with special system programmed operators, which to the programmer are, however, as easy to use as load and store instructions. The system handles the buffering and the actual transfers between main memory and secondary storage. To communicate between procedure files the user must create intermediate data files, transfer control to the system and then explicitly request that control be given to a new procedure file by command from the terminal. Thus, a cost paid in the SDS-940 system is that the user must have more knowledge of what is going on and may have to interact more frequently with the system from his terminal. We feel that future economical large scale time-sharing file systems will be based on a compromise between the philosophies represented by the SDS-940 and the Multics system. For a very large number of applications, the SDS-940 file system approach is quite adequate; however, there are many applications requiring structured data files which require a more general file system.

Let us now examine briefly the mechanisms by which the time-sharing system keeps track of files on secondary storage. The basic concept is that of the file directory. Each user has a file directory[a] which is maintained

a) In Multics a user can have multiple directories structured as a tree. This ability to structure groups of files may be of aid in developing large systems.[12][21]

permanently on disc or other storage media between sessions. During a session, a given user's directory can be brought into main memory by the system. The system must maintain a master directory containing the location of each users directory.

The directory contains the following types of information for each file, usually stored as a table of contiguous computer words.

1) the file name as a character string or a pointer to such a character string

2) type of file (file type means binary, character or other type as defined in a particular system)

3) access protection (possibly a list of other users who have access to the file and the kind of access which they are allowed such as read only, write only, read-write, execute only)

4) information indicating directly where the file is stored or a pointer to further tables containing such information (i.e., a pointer to the index block discussed earlier)

5) other information such as date of definition, frequency of use or additional facts felt to be useful by an installation or system designer.

In large systems the number of files which must be stored may exceed the capacity of a single disk or other single storage device and therefore, a hierarchy of storage media such as tape strip, photo store and magnetic tape may also be required. One wants the most frequently used information on the small capacity, higher speed devices and the infrequently used information on the slower, larger capacity, devices. The system must keep usage statistics to enable it to move the information in the hierarchy to satisfy the above goals. The problems of handling a storage hierarchy in a user invisible manner are new and little experience has been gained with working systems, although systems such as Multics and the IBM 360/67 are attempting to deal with this problem.

Other very important problems exist in the design of file backup systems to give file protection. For systems with a small volume of files, backup protection can be easily obtained by periodically dumping the files from the disk to magnetic tape. However, most installations will eventually have file volumes in which dumping could take hours just to move the tapes and therefore more sophisticated backup procedures must be devised. The subject of file dumping is discussed further in references (12) and (21).

System Protection

The design of the protection system is one of the more critical aspects of a time-sharing system. System protection exists at many levels, both in hardware and software.[20)][27)] Two ways of looking at protection are 1) protection of resources (software and hardware) and 2) protection of access paths of processes with respect to each other. A well designed system will have an integrated approach to both points of view, uniformly and clearly implemented throughout the system. Also, a well designed system will offer many levels and types of access rights to those facilities it requires with the minimum level of control required. Such an approach will increase system reliability by limiting the area of damage caused by hardware and software bugs.

The protection scheme of the GE-645 is expensive of system time, but is conceptually useful because protection throughout the system has been treated in a very general and uniform manner. On the 940 system, in contrast, protection exists at many points, but is handled in a non integrated way. The GE-645 ring approach (to be described below) offers many levels of protection whereas the 940 offers only two. The ability to have many levels is a desirable feature as seen from experience on the SDS-940. The two basic levels of protection available in the SDS-940 system are executive status and non-executive status. A user with executive status has access to all system routines and great power to modify various tables and other parameters. For this reason users are not given this status and yet some users could usefully employ some of the system routines. The risks involved, however, are very great because the executive status user then has uncontrolled access and this risk prevents giving any user executive status.

Within the software system, protection for specific resources is maintained in various tables associated with these resources. For example, as was discussed above, the files are protected with information stored in the table called a file directory. In the case of the GE-645, protection information for a segment (in the 645 the terms file and segment are identical) is transferred to the segment descriptor table when a segment is activated. Access to subsystems can be similarly restricted with information stored in users account tables and the tables used to initialize the subsystems as is accomplished on the SDS-940. Users of the SDS-940 have a subsystem status which defines the subsystems they have access to. I/O devices are protected by information stored in device and file control blocks. For example, a user's account table could indicate whether the user has access to magnetic tape drives or graphical displays, or similar information could be kept in tables associated with the devices. The most general problem is defining access privileges of processes with respect to each other. The most general, integrated and clearly defined approach to this problem is found on the Multics system.20) The protection scheme is based on the idea of concentric rings of protection with the inner rings having the highest protection. Segments are assigned to these rings. The most critical system segments (those interfacing to the hardware) are in ring 0 and less critical system segments are in ring 1 and so forth. User segments are assigned to rings further out in the series. In outline the system works as follows. A segment can directly access any other segment in the same ring. If a ring boundary is crossed a call to the system is made. Access is freely granted to segments in higher ring numbers, but restricted for segments in lower ring numbers. To call an inner ring only certain entry points are allowed. The system checks the target address against the entries on a valid entry list for the target segment. If the user is allowed access and the target entry point is valid, then further checking takes place. Because addresses can be passed as arguments they must be checked to see that the calling segment has access to the segments they refer to. This is necessary because the inner ring has greater access privileges and these adresses might not be checked when used. The addresses passed as arguments, if improper, could cause the inner segment to perform erroneous action and thus cause considerable damage to itself or other system segments.

In the earlier discussion on switching between system and user modes, we indicated that considerable overhead was involved in this process in the Multics system because a call to the system is a transfer to another segment and uses standard intersegment transfer conventions. Some of the protection conventions have been outlined above and should show some of the costs associated with calls to system segments which occupy positions in the inner rings.

Other problems with the ring system exist because two processes can share segments and after validation one process may be blocked and another process started which can use the segment in the inner ring without validation. To avoid difficulties in this area, careful handling of interrupts must be performed during the validation process. The ring system uses the protection available in the hardware fully, but still requires much software manipulation for validation and for remembering which ring is presently in control and in which ring to return control after a routine in a particular ring is completed. Simple additions to the hardware could decrease significantly the bookkeeping required for the latter. The simple additions would be to add additional bits to the procedure base register described earlier to indicate the ring number of the segment in execution and to add a ring number to the information stored as a segment descriptor.[20/] Presently this information is maintained in the file directory. This is an example of the type of improvement one can expect when further experience is gained with the total system.

The Non-resident Portion of the System

All routines which are not required to respond within micro-seconds or a few milliseconds to requests for service are candidates for being made non-resident. The other criterion to be applied is frequency of use. If the routines are infrequently used, the resources expended in bringing them into main memory are justified by the savings in memory requirements. In a large system such as Multics, with hundreds of thousands of words of code, it is one of the important design problems to clearly define which routines can be made non-resident.

There are a large number of routines associated with serving the interface between the user at a terminal and the system which are generally non-resident. Many routines associated with defining and searching for files are also commonly non-resident, as are routines used with I/O devices which are not heavily used. The interface with the user and other non-resident routines are lumped together in the SDS-940 system in a reentrant procedure called the executive. The Multics system consists of segments, some of which are locked in core, resident segments, and others of which are available on call-non-resident segments. The particular segment interfacing with the user is called the command system.

Time-sharing commands are available:

1) to log in and out of the system,

2) to create, destroy, and manipulate files,

3) to gain access to subsystems (subsystems may also have a command language of their own),

4) to determine system status,

5) privileged commands are usually available to the operator to define new users, delete users, change user privileges, access accounting information and access more detailed system status information then is available to the general user.

When a user logs-in to the system, various tables are set up for the user and the command system is made available to him. The command system is written with reentrant code and logically each user thinks he has a private copy of the command system. The user can gain access to the command system at any time, even though he may be communicating with a subsystem or one of his own programs, by pressing a character conventionally assigned by the system for this task. When the teletype handling routines "see" this character, transfer is made to the routines required to bring the command system into memory if it is not already there.

The command system collects a string of characters forming a command, decodes the command and transfers to the routine for handling the decoded command. Any arguments for the command are passed to the routine also.

If the command is a call for a subsystem, the subsystem is assigned to the user by setting up the appropriate memory tables of the calling user and, if the subsystem is not in core, it is then read into core by calls to the appropriate I/O routines. Information required to set up a subsystem is stored in special tables.

The handling of the file directory, and the checking of user status to determine the nature of access allowed to a given file is often handled by non-resident routines, as defining files does not happen frequently enough to warrant permanent residency.

The Concept of Overhead

The term "overhead" is heard frequently in discussions on time-sharing. The meaning of the term is very fuzzy and is generally used to denote the per cent time the machine is occupied performing system functions such as swapping, protection, scheduling and so forth. The general implication is that overhead is inherently bad. Looked at in this way system functions are not seen in proper perspective. One certainly wants to minimize, relative to a given level of service performed by the system, the time the machine spends on system functions, but one must recognize the usefulness of system functions in the total work of the user. In a system in which there is much use of shared information, the time spent by the system in protection functions may have to be increased, but hopefully the use of shared information increases significantly the users productivity. Similarly, the use of large virtual memory spaces will increase the time spent in system routines, but possibly to the total advantage of designers of large user systems. The determination of tolerable overhead is only possible with a sound weighing of all the costs associated with a computer operation, which must include the user. Varying figures will be acceptable by different installations depending on the nature of their work load.

A more meaningful measure of system utilization then overhead is system idle time. The amount of idle time is a direct indication of the total design balance of hardware component performance characteristics with each other and the resource allocation algorithms. Nielsen's[34)35)] simulations of the IBM 360/67 system showed clearly the results on idle time of a poorly balanced system. Measurement of system component utilization (hardware-software) is a long overdue engineering discipline which has been started in the past two years. It is only with sound data that show clearly the information flow bottlenecks that more economical time-sharing systems can be developed.

Some Concluding Remarks on Time-Sharing

This paper has tried to set forth the major concepts involved in the design of present time-shared computer systems. Based on an understanding of these concepts and their underlying assumptions and motivations, it should be possible to set up criteria for evaluating such systems. The most important point to be emphasized is that there are many cost-generality trade offs which must be considered if efficient economical systems are to be made available at a given point in the developing state of the art. Not only must the designers of such systems consider these factors, but each installation planning or acquiring such a system must consider them in light of their own projected needs.

The central problem around which time-sharing development has pivoted to date is the limitation imposed by the cost of main memory. The attempt to develop schemes such as paging to enable main memory to be fully utilized, to develop schemes such as segmentation to limit the requirement for multiple copies of shared data and procedures, to develop elaborate swapping schemes to cut costs through use of less expensive auxiliary storage devices such as drums, develop elaborate memory organizations and bus structures to allow several processors to concurrently use a given block of memory, all result because of the present cost of main memory. Looking at trends in the development of large scale circuit integration, there seems to be a high probability that major cost-performance improvements can be expected within the next five years. Such improvements will, in our opinion, make possible the design of more straight forward systems. This trend will result because one will then be able to trade ease of system design and implementation for a less efficient utilization of main memory.

What the field desparately needs are statistics on system utilization so that bottlenecks and costs can be seen more clearly. Simulation and analytical tools are also required so that various trade offs can be carefully studied. Some researchers in the time-sharing field have tried to indicate that their systems are too complicated to study with simulation and that the first version of the system is in fact the simulation. Work on simulation at UC Berkeley,[40] SDC,[17] Stanford[34][35] and MIT[49] shows that the above view should not be readily accepted. The pessimistic view indicates more the fact that early system developers did not want to stop and develop these tools before moving on to implementation than the fact that useful simulation is impossible.

We can expect that the experience of the growing number of time-sharing service bureaus is going to have a large impact on future developments in this field. These firms more so than industrial research laboratories or universities, where most time-shared computer systems are installed, are going to 1) be very sensitive to cost-generality trade offs and 2) have a collective experience with a very wide range of user requirements. To date universities have been the major source of time-sharing developments. Computer manufacturers have been building competence in the time-sharing area and can be expected to take a more decisive hand in the mid 1970's.

The time-sharing field has been subject to bursts of enghusiasm for concepts such as segmentation without carefully examining the underlying assumptions, costs or alternative ways to achieve the same goals. Concepts have been very easy to generate to date, but many have proven quite difficult to implement in practice. Many firms have ordered systems on the basis of a manufacturers promise or on the basis of a working prototyke only to be very disappointed by the manufacturers failure to deliver or by the length of time required to turn the prototyke into a system meeting commercial standards of reliability. All of the above factors coupled with increasing knowledge of the concepts involved in time-shared systems are helping to produce a more mature questioning attitude on the part of potential time-sharing system customers.

With the above we close our introduction to concepts involved in the design of a general purpose time-sharing system and now move to a brief discussion of on-line file maintenance and retrieval systems, multiprogramming batch systems allowing remote access, and special purpose time-sharing systems to examine some of the characteristics which differentiate them.

On-line File Maintenance and Retrieval Systems

Examples of this class of systems are airline reservation systems, computer assisted instruction systems, stock transaction systems, and ordering and inventory control systems, a well known specific example being the American Airline SABRE System.[16] Inputs to these systems are queries from user terminals. Their outputs are answers to the queries obtained by analyzing the contents of a large data base. Most of these systems are special purpose and do not allow the users to generate procedures, or generate queries outside a fixed set. There are systems developing in this class though which do allow users to generate simple procedures to manipulate the information retrieved from the data base. Interrupts from the I/O devices are the major signals driving the scheduling algorithms.[45]) Each message has a priority and security level associated with it. These systems are primarily I/O limited. Although the users view the system as providing a fixed set of processing capabilities oriented toward the particular application, different users may be using different capabilities at the same time with different levels of priorities. Because the arithmetic logical operations on the data are usually limited, once a process is started it usually remains in main memory until completed unless space is required for a higher priority process. More than one process occupies main memory at any given time to allow processing to take place during the frequent I/O requests of the processes in this class of system.

This class of system requires capabilities for protection, for rapid transmission of information between main and secondary storage, routines for handling terminal devices, and priority scheduling and allocation algorithms, as does the general purpose time-sharing system. In summary one can list the following characteristics of this class of system:

1) the entire system is I/O bound,

2) the system performance times are critically affected by the order in which various I/O tasks are performed,

3) it is important to have many different jobs in main memory, each placing requests into an I/O queue so that the monitor can best schedule the order of peripheral storage accesses,

4) swapping should be kept to a minimum to avoid aggravating an already I/O bound system,

5) each user should be capable of making logically independent requests for service,

6) each user expects responses to his query to be returned rapidly because his task usually will require a sequence of queries to be presented.

7) The state of main storage is likely to be one of the prime factors influencing the scheduling process. The following types of questions would be looked at by the scheduler:

   a) Is enough free storage available to satisfy the request?

   b) If the request is granted will the job free storage?

   c) Will the job be making additional requests for storage?

In a system one of the major delays results from positioning of access mechanisms.[12] Therefore, the I/O scheduler should make some attempt to order requests to minimize positioning time. Other factors such as job priority, elapsed time since the request was entered, and main storage availability will affect the I/O scheduler also.

Multiprogramming Batch Systems With Remote Access

A separate study is being undertaken at ERC defining characteristics of this class of system and therefore only a very brief discussion is given here. Systems in this class include as examples IBM's OS/360, UNIVAC's Exec 8, and GE's GECOS. These systems have all the same general features as the general purpose time-sharing system for scheduling and resource allocation. The main difference between these two classes lies in the specific emphasis given in the design details of the various modules. These systems have been designed primarily for batch throughput and, while adaptable to conversational computing, tend to be expensive when used in this manner. This type of on-line usage, in our opinion, needs to be reflected in the design details of the entire system.

In practive these systems usually can allow a limited number of on-line terminals access to a reentrant subsystem such as a text editor[a] for the preparation of source code to be used with a compiler for a language such as FORTRAN. When the code is prepared, it is entered on the batch queue usually with a high priority in order to achieve reasonable response at the terminal. Although features which enable swapping may be available, present systems in this class were not designed to efficiently handle this type of usage and usually restrict program size to keep them in main memory. The conversational features of present systems are more often second thoughts patched into the original batch oriented structure, than original design goals reflected throughout the system.

The design goals of OS/360 and the MULTICS operating system on the GE-645 have many features in common in their attempts to be all things to all users and have suffered as a result. They differ though in the prime emphasis, one on batch throughput, the other on conversational computing and manipulation of resources. It is not clear that any one system can ever be designed to perform economically and efficiently on both classes of service, although either can handle some of the type of work primarily designed for the other. The computer field is really only in the second generation of operating system design and the question as to whether or not one system can be designed for both classes of usage will be more clearly answered when the lessons learned on this generation have been absorbed and reflected in changes in hardware design and software design on the next generation. Our observations of the present evolution of these two classes of system are that they each are beginning to incorporate features found initially in the other.

a) At present OS/360 has no such editor, although one is planned for UNIVAC's Exec 8. Various OS users have developed text editors. The discussion here is general to this class of system to indicate the type of approach usually followed.

One difference is particularly noticeable at present between these two classes of system and that is the hardware organization. Both classes require the same basic features and yet, due to the fact that universities are providing the prime source of general purpose time-sharing system ideas, the hardware organization of systems oriented toward general purpose time-sharing is generally richer in aids to effective sharing of resources. Another important difference between these two classes of systems is that the multiprogrammed batch-oriented systems are often designed to run on a family of machines. Therefore, the designers of these systems have tried to avoid producing algorithms which utilize time dependencies of any machine or device in the line. This approach decreases the cost of producing operating systems for a family of machines and offers the user expansion capability at minimal cost, but does not lead to the design and implementation of the most efficient systems. The smaller models of such a family tend to dictate many features of the design of compatible software and larger models tend not to be effectively utilized. The designers of many of the available time-sharing systems, on the other hand, have tried to utilize as much knowledge as possible of the timing characteristics of their hardware to achieve greater efficiency and faster response.

With the above general introduction let us briefly examine the structure of the most general version of IBM's OS/360 called MVT (Multiprogramming with a Variable number of Tasks) which is representative of this class of system.[23)] In OS terminology the word task is used to represent the type of entity called a process throughout this report. A job is an accounting entity logically separate from all other jobs and consists of one or more tasks. A task is a program and its data, for example, a compiler and the source program to be compiled. A task can create subtasks.

More than one task is resident in main memory at one time. There are two main schedulers in the system, one to schedule jobs and the other to schedule tasks. When the user enters his job into the system he indicates a priority and main storage requirement. Jobs are scheduled into the system based on these two parameters.

Main memory is allocated by what IBM calls partitions. A partition is a contiguous area of memory equal to the requirement input with the job. Once a job is entered into main memory it remains there until it is finished. There are several jobs in different partitions in main memory simultaneously. The tasks of a job are either defined explicitly with control information entered with the job or implicitly by tasks setting up further subtasks. There may be several tasks in a partition. Each task has associated with it a Task Control Block (TCB) which contains state information for the task and is used by the task scheduler. To review, a job scheduler determines which jobs are to be brought into main memory. A task scheduler then determines which task is to be run at a given time. The task scheduler gains control on each interrupt and examines the task queue to determine the highest priority task ready to run to place in execution next. There is no time slicing among tasks in present versions of MVT, but clock interrupts can be put in the system to perform this function as this type of interrupt is the same in principle as any other. Without the time interrupt, a task once placed in execution is allowed to run until it generates a call to the system or an I/O operation for some other task is completed.

A feature called rollout is being implemented which allows a high priority task to gain more memory than initially assigned by explicitly asking the system for more memory. The additional memory is obtained by forcing a lower priority task to be moved to the disk. When the higher priority job is completed the unfinished task can be moved from the disk back to main memory. There is no provision for swapping as understood in general purpose time sharing systems.

To implement a "time-sharing" like ability with OS there are several possible approaches, two of which are mentioned here to give the reader a feeling for what is involved. Under OS, software support can be placed in a partition which is given the highest priority to provide limited conversational abilities.

One way to provide this ability directly within the structure of OS is to allow each remote terminal to create tasks which are like any other OS task.22) By the use of a clock interrupt and the variation of task priority numbers the system can commutate its resources among the tasks in the partition.

Another approach is to set up a very high priority task in partition which is actually a submonitor. It then performs its own scheduling of user tasks and could perform rather inefficient swapping by standard calls to the appropriate OS I/O routines.

From the above discussion and that of general purpose time-sharing given earlier, one can see that there are significant differences at the implementation level in scheduling, memory allocation and resource handling (software and hardware) between general purpose time-sharing systems and on-line usage of batch multiprogramming systems with remote access. Besides the differences in resource allocation which exist between the two classes of system, there are important differences in the protection features which are available. Sharing of information in the sense occurring in a general purpose time-sharing system does not take place in a batch system and therefore a batch oriented system will have a weaker protection system.

Special Purpose Time-Sharing Systems

The main defining characteristic of special purpose time-sharing systems is that these systems allow programs to be written in a very limited number of languages (usually one). Examples are the JOSS system at Rand4) or the IBM Quiktran system. These systems require very limited file manipulation and simple file structures. Because the size of programs generated are usually not large, several programs can be maintained in main storage simultaneously and overlapped computing and swapping are easily achieved. These systems and their language are designed for a specific application area and are noted more for the careful attention given to ease of use and learning than to features of their system design.

RWW:lc

BIBLIOGRAPHY

1. Amdahl, G., Unpublished simulation study, IBM Corporation 1965.

2. Arden, B. W. et al., "Program and Addressing Structure in a Time-Sharing Environment", JACM 13, 1 (Jun 1966), pp. 1-16.

3. Bell, G., Pirtle, M. W. "Time-Sharing Bibliography", Proceedings of the IEEE, Vol 54 No. 12, December 1966,p 1764.

4. Bryan, G. E.,"JOSS 20,000 Hours at the Console: A Statistical Survey", AFIPS Conference Proceedings, Vol. 31,FJCC 1967, pp 769-778.

5. Coffman, E. G. Varian, ° C., "Further Experimental Data on the Behavicr of Programs in a Paging Environment", Communications of the ACM, Vol. 11, No. 7,July 1968, p 471.

6. Coffman, E. G., Kleinrock, "Computer Scheduling Methods and their Counter Measures", AFIPS Conference Proceedings, Vol. 32, Spring 1968, p 11.

7. Comfort, W. T., "A Computing System Design for User Service", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington D. C., pp 619-626.

8. Conti, C. J. and others, IBM System/360, Model 85 System Design, IBM Technical Report TR 00 1688, January 3, 1968.

9. Corbato, F. J., et al., "An Experimental Time-Sharing System", AFIPS Conference Proceedings 21 (1962 SJCC), National Press, Palo Alto, 1962, pp 335-344.

10. Corbato, F. J., and Vyssotsky, V. A., "Introduction and Overview of the Multics System", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965 pp 185-196.

11. Daley, R. C., Dennis, J. B., "Virtual Memory Processes, and Sharing in MULTICS",ACM Symposium on Operating System Principles Oct 1967, published in Communications of ACM, Vol. 11, No. 5, May 1968, p 306.

12. Daley, R. C. and Neuman, P. G., "A General Purpose File System for Secondary Storage", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books Washington, D. C., 1965 pp 213-229.

13. Denning, Peter J., "Effects of Scheduling on File Memory Operations", AFIPS Conference Proceedings, Vol. 30, Spring 1967, pp 9-21.

14. Denning, Peter J., "The Working Set Model for Program Behavior", ACM Symposium on Operating System Principles, Oct. 1967, CACM, Vol. II, No. 5, May 1968.

15. Dennis, J. R., "Segmentation and the Design of Multiprogrammed Computer Systems", JACM 12, 4 (Oct 1965), pp 589-602.

16. Evans, G. J., Jr, "Experience Gained from the American Airline SABRE System Control Program", Proceedings 22nd National Conference ACM, 1967 pp 77-84.

17. Fine, G., and others, "Dynamic Program Behavior Under Paging", Proceedings National ACM Meeting, 1966, p 223.

BIBLIOGRAPHY (Contd)

18. Gibson, C. T., "Time-sharing in the IBM System/360: Model 67", AFIPS Conference Proceedings 1966, Spring, Vol. 28, p 61.

19. Glaser, E. L. et al., "System Design of a Computer for Time-Sharing Applications", AFIPS Conference Proceedings 27 (1965 FJCC) pp 197-202.

20. Graham, R. M., "Protection in an Information Processing Utility",ACM Symposium in Operating System Principles,Oct 1967, Published in Communications of the ACM, Vol. 11, No. 5, May 1968, p 365.

21. Ha, E. P. L., Data Management in Time Sharing Systems, Shell Technical Progress Report in preparation.

22. Hobbs, W., and others, "The Baylor Medical School Teleprocessing System", AFIPS Conference Proceedings, Vol. 32, Spring 1968, p 31.

23. IBM Systems Journal, Vol.Five, Number One, 1966 (devoted to a descriptor of OS/360).

24. Kilburn, T., et al., "One-level Storage System, IEEE Trans. EC 11, 2, 1962,p 223.

25. Kudlick, M. D., File Structures for Random Access Devices, Shell Technical Progress Report in preparation.

26. Lampson, B. W., et al., "A User Machine in a Time-Sharing System", Proc. IEEE (Dec 1966).

27. Lampson, B. W., "Scheduling and Protection on Interactive Multi-Processor Systems", Project Genie, University of California, Berkeley Document No. 40.10.150, January 20, 1967.

28. Lauer, H. C., "Bulk Core in a 360/67 Time-Sharing System", AFIPS Conference Proceedings, Vol. 31, 1967 Fall, pp 601-611.

29. Le Clerc, Jean-Yves, "Memory Structures for Interactive Computers", Project Genie Document 40.10.110, University of California Berkeley (May 1966).

30. Lichtenberger, W. W., and Pirtle, M. W., "A Facility for Experimentation in Man-Machine Interaction", AFIPS Conference Proceedings 27 (1965 FJCC), pp 589-598.

31. McCullough, J. D., et al., "A Design for a Multiple User Multiprocessing System", AFIPS Conference Proceedings, Vol. 27, 1965, Fall p 611.

32. McGee, W. C., "On Dynamic Program Relocation", IBM Systems Journal, Vol. 4, No. 3, 1965, p 184.

33. Morris, D., Summer F. H., "An Appraisal of the Altas Supervisor", Proceedings 22nd National Conference ACM, 1967,pp 59-76.

34. Nielson, N. R., "Computer Simulation of Computer System Performance", Proceedings 22nd National Conference ACM, 1967,pp 581-590.

35. Nielson, N. R., "The Simulation of Time-Sharing Systems", Communication of the ACM, Vol. 10, No. 7, July 1967, pp 397-412.

BIBLIOGRAPHY (Contd)

36. Oppenheimer, G., Weiger, N., "Resource Management for a Medium Scale Time Sharing Operating System", ACM Symposium on Operating System Principles, Oct 1967, Published in Communications of the ACM, Vol. 11 No. 5, May 1968, p 313.

37. Organick, E. I., "A Guide to Multics for Subsystem Writers", a Project MAC document March 1967.

38. Csesana J. F., et al., "Communication and Input/Output Switching in a Multiplex Computing System", AFIPS Conference Proceedings 27 (1965 FJCC).

39. O'Sullivan, T. C., "Exploting the Time-Sharing Environment", Proceedings 22nd National Conference, ACM, 1967, pp 169-176.

40. Pirtle, Mel, "Intercommunication of Processors and Memory", AFIPS Conference Proceedings, Vol. 31, 1967 Fall, pp 621-633.

41. Pirtle, M. W., "Intercommunication of Digital Computer Processors and Memories", Ph.D Thesis University of California, Berkeley, 1967.

42. Pirtle, M. W., "Modifications of the SDS-930 for the Implementation of Time-Sharing", Project Genie, University of California, Berkeley, Document No. M-1, April 4, 1967.

43. Pyke, T. N., Jr., "Time-Shared Computer Systems", in Advances in Computers Vol. 8, Academic Press, N. Y., N. Y., 1967.

44. Randell, B. Kuehner, C. J., "Dynamic Storage Allocation Systems", ACM Symposium on Operating System Principles, Oct 1967, Published in Communications of the ACM, Vol. 11 No. 5, May 1968, p 297.

45. Reiter, Allen, "A Resource-Allocation Scheme for Multi-user On-line Operation of a Small Computer", AFIPS Conference Proceedings, Vol. 30, 1967 (SJCC), pp 1-7.

46. Saltzer, J. H., "Traffic Control in a Multiplexed Computer System", MAC-TR-30 (Thesis) M.I.T., Cambridge Mass, July 1966.

47. Schwartz, J., "A General Purpose Time-Sharing System", AFIPS Conference Proceedings 25, (1964 SJCC), pp 397-411.

48. Sherr, A. L., "Analysis of Storage Performance and Dynamic Relocation Techniques", IBM document Tr-00-1494.

49. Sherr, A. L., "An Analysis of Time-Shared Computer Systems", Ph.D Thesis June 1965, MIT, Project MAC document MAC-TR-18.

50. Smith, Arthur A., "Input/Output in Time-Shared, Segmented, Multi-processor Systems", MAC-TR-28 (Thesis) MIT, Cambridge, Mass, June 1966.

51. Varian, L. C., Coffman, E. G., "An Empirical Study of the Behavior of Programs in a Paging Environment", ACM Symposium on Operation System Principles, Oct 1967.

52. Wald, B., "The Descriptor - a definition of the B5000 information processing system", Burroughs Corp., Detroit Mich., 1961.

BIBLIOGRAPHY (Contd)

53. Watson, R. W., Design of a General Purpose Graphic Terminal for a Time-Sharing System, Emeryville Technical Progress Report 138-68.

54. Wegner, P., "Machine Organization for Multi-programming", Proceedings 22nd National Conference ACM, 1967,pp 135-150.

55. Wood, T. C., "A Generalized Supervisor for a Time-Shared Operating System", AFIPS Conference Proceedings, Vol. 31, FJCC 1967, pp 209-214.