Machine organization for multiprogramming

by PETER WEGNER
Cornell University
Ithaca, New York

INTRODUCTION

This paper is intended as an introduction to some of the basic concepts of multiprogramming for readers who wish to study the more specialized literature in this field. It attempts to develop a framework for the discussion of multiprogramming which motivates the forms of machine organization used in current multiprogramming systems. The key requirement in multiprogramming systems is that information structures be represented in a hardware-independent form until the moment of execution, rather than being converted to a hardware-dependent form at load time. This requirement leads directly to the concept of hardware-independent virtual address spaces, and to the concept of virtual processors which are linked to physical computer resources through address mapping tables. The structure of the class of hardware-independent virtual processors in the IBM 360 model 67 and GE 645 systems (1), (2), (3), (4), is developed in some detail. Questions of efficiency of throughput in the resulting class of computer systems are considered.

Resource allocation in multiprogramming systems

Multiprogramming, multiprocessing and multiaccessing

Computer systems in which a number of user programs may be simultaneously competing for physical computer resources such as memory registers or processing units are referred to as *multiprogrammed* computer systems. The set of techniques for realizing multiprogrammed computer systems is referred to as *multiprogramming*. Multiprogramming may be performed either on a computer with a single processor or on a computer with multiple processors. The set of techniques for realizing computer systems with more than one processing unit is referred to as *multiprocessing*. A subfield of multiprogramming is concerned with the problems of computer system organization which arise specifically because of the multiplicity of input-out devices which interface with the system. The problems in this area are referred to as problems of *multiaccessing*.

Both multiprocessing and multiaccessing involve the allocation of scarce computer resources such as the main memory and the processing units among competing user-initiated programs, and therefore are subfields of the general area of multiprogramming. However, multiprogramming may occur even on computer systems with only a single input channel and only a single processor.

Efficiency Versus flexibility

A large computer system may be thought of as a utility which is intended to serve a variety of users both flexibly and efficiently. Access to the system by the user should be simple, rapid, and sufficiently flexible to allow the user to suit the mode of access to his needs. For example, an application which requires the computer to make real time responses to an on-line process requires a different mode of operation from that for a batch processing problem whose results are not so urgently required. The mode of operation required to service a user who is debugging a program at a typewriter console and requires small bursts of computation to be performed within a short period of elapsed time must also be catered for.

Program execution in each of the permitted modes of operation should be efficient both in terms of resource utilization and in terms of user requirements. Saltzer (1) has classified the problems of computer system organization into *technological* ones concerned with efficient resource utilization (throughput), and *intrinsic* ones concerned with the convenience of the user. An alternative is to consider the user as one of the resources of the computer system, whose efficiency of utilization is determined by the user facilities and the response pattern of the computer system to run requests by that user. Intrinsic problems may in this way be modelled into technological problems; i.e., the intrinsic problems of providing adequate service to an on-line process

or to a user at a console can be modelled by the technological problem of providing certain user facilities and computer response patterns for classes of peripheral devices.

Time slicing

The hardware of a computer system consists of a collection of physical resources each of which has certain operating characteristics. When considered statically all computer resources are information storage devices which at different times are occupied by different items of information. Every resource has a one-dimensional existence through time referred to as its *time line*. The time line of each resource can be subdivided into segments called *time slices* corresponding to periods for which the resource is allocated to a particular information item. The computer resources form a hierarchy such that some are more in demand than others, although the total storage capacity in the computer (including auxiliary memory) is sufficient for information items of all computations.*

At any given point of time the computer hardware is occupied by a group of "loosely interconnected" information structures each of which represents a process at some stage of execution. The term *process* or *computation* will be used to denote the sequence of information structures representing the program and data of a given "user" during successive stages of execution.** It is convenient to introduce the notion of a time line for computations. The time line of computation measures progress within the computation in terms of the number of executed instructions since the beginning of the computation, and has no direct correspondence with real time.

The information structure associated with a computation undergoes transformations as it progresses along its time line. A snapshot of the information structure at a given point of the computation will be referred to as an *instantaneous description.* A computation may be completely characterized by the sequence of instantaneous descriptions to which it gives rise. Individual instructions or executable program segments may be characterized by the effect which they have in transforming instantaneous descriptions.

Allocation of information to resources

The physical storage registers in which instructions and data reside while they are actually being transformed are referred to as *processor registers.* Processor registers are in very great demand during a computation, and the time slice of processor registers allocated to an information item is restricted to the time that the information item is required in transforming the instantaneous description. When an information item in a processing register is no longer required it is moved to a register that is less in demand by an *information moving instruction.*

The speed at which information moving instructions operate is determined by the accessing characteristics of the information storage media between which the information is moved. It is important that instructions which move information in and out of processing registers can be rapidly executed, since the moving of information in and out of processing registers constitutes a greater computational bottleneck than the processing time. Information which has been moved out of the processing registers and is no longer required can be moved to an information medium with slower accessing characteristics by more slowly executed information moving instructions which do not tie up the processing unit while they are being executed.

The information storage medium which serves as the direct source and destination of processing-unit information is called the main memory. Information storage media to which information is moved when it is not directly required by the processing unit is called the auxiliary memory. There may, in general, be several levels of auxiliary memory with different accessing speeds, some communicating directly with the main memory and others communicating with the main memory through one or more intermediate levels of auxiliary memory.

A computer system normally contains only a small number of processing units in which processing can be performed, and a hierarchy of different memory devices with different accessing speeds. Information which is not currently in use is normally stored in a

*We adopt the point of view that a coTputation has just as real an existence in time as a physical resource, and use the terms "time line" and "time slice" to denote time segments for both kinds of objects.

**The term "user" does not necessarily have human connotations and should be thought of as a group of programs for performing a certain function, or a "front" for purposes of accounting, rather than as something of flesh and blood. Thus system programs for performing specific system functions may be thought of as users. "Users" not under the control of the problem programmer are sometimes referred to as "Daemon users" (1) It is usually possible to dynamically partition executed instructions so that each is associated with precisely one user. However, there are some fuzzy boundaries in such partitioning for which arbitrary decisions must be made. Partitioning of a set of static information structures among users for purposes of "space accounting" present problems because structures may be shared by more than one user.

low-speed memory device. Information which is currently being used in processing is stored in processing registers. Information which is about to be used must be stored in the main memory if the computer is to access it directly. If an information item accessed by a processing unit is not in the main memory, the processing unit cannot proceed with the computation until the slow information transfer from auxiliary to main memory has been accomplished. The time for an information transfer from auxiliary to main memory is typically at least one thousand times as long as the transfer time from the main memory to the processing unit, so that the real time required to execute an instruction whose information is not in main memory is several orders of magnitude greater than that required for an instruction whose information is in main memory.

If a processing unit is to execute a sequence of instructions at its normal processing speed, then all components of the instantaneous description accessed during execution of this sequence of instructions must be in the main computer memory. It is one of the principal tasks of a programming system to organize information transfers between various levels of auxiliary memory so that information is in the main memory when it is required by the processor. The programming system must allocate time slices of blocks of physical main memory registers so that information required for processing is usually, though not always, in the main memory before it is used. At the same time, the memory time slices allocated to an information item should not greatly exceed the time period during which it is used, so that it can be freed for use by other information items. The efficiency of decisions regarding allocation of physical memory to information items is determined by the time pattern of accesses to the information item. In considering this time pattern, it is important to distinguish the time pattern of access in the internal time scale of a given program and the time pattern in real time when interrupts are to be taken into account.

Time patterns of accessing

Time patterns in which access to a given information block occurs in bursts separated by long intervals with no accesses allow very much greater efficiency of physical memory allocation than real time patterns in which accesses to a large number of blocks are interspersed with each other in a relatively uniform manner.

The real time pattern of information accesses in a multiprogramming system is inevitably more diffuse than in a batch-processing system because different processes are interleaved with each other on a given processor. Such interleaving of processes not only requires information items of a number of interleaved processes to occupy concurrent time slices of the main memory, but also requires information items of each of the processes to occupy its time slice for a longer period of time.

The index of main memory utilization by a given computation or set of computations is clearly the product of the number of physical main memory registers used and the time for which they were used. This index will be referred to as the *memory slice* of the computation or set of computations. An example will be given to show that the memory slice occupied by a set of processes rises sharply as the number of processes being simultaneously executed increases.

Example: Assume that there are n tasks with similar time and space requirements to be executed on a single processor of the multiprogramming system. Assume also that each process requires m fixed size blocks (pages) of main memory to operate efficiently and that the internal process time during which the process is required is k seconds for each process. Then the memory slice required for processing the set of n tasks in sequence is kmn units. If, however, the n tasks are interleaved, then each task occupies mn blocks for kn seconds, so that the memory slice required to execute the set of tasks is kmn^2 units.

Multiprogramming leads to greater technological efficiency by allowing processor idle time in a given process to be used by another process which is ready for execution. It greatly facilitates more efficient servicing of multiple users requiring real time responses and short elapsed time responses. However, it leads to a greater strain on memory resources even in the case when program characteristics are assumed known and memory allocation problems are assumed to have been solved.

Matching software to resource allocation

In a multiprogramming system with given facilities for allocation of information structures to resources the system software must be specifically designed to work efficiently under the given allocation scheme. Efficient design of system software can improve overall system efficiency at two levels.

1. If frequently used system programs are constructed to make efficient use of computer resources during their execution, than all programs that utilize scarce computer resources during the execution of these system programs will operate more efficiently, resulting in an overall improvement of system efficiency.

2. Compilers and other programs that determine the run-time representation of user programs should cause programs to have a run-time representation that makes efficient use of allocation facilities during execution.

If, as has happened in a number of instances, the performance of a given multiprogramming system has been found to be poor, then it is difficult to judge whether the poor performance is due to inherently unworkable allocation procedures, or to software design which made poor use of the given allocation procedures. A complex system is as weak as its weakest link, and it is not always possible to identify the weakest link in a complex system. Indeed, since components of a system strongly interact, there are usually a number of alternative ways of improving the overall performance of a system, such as expanding hardware capacity of critical hardware components, placing restrictions on multiprogramming within the system, providing poorer elapsed time service to certain classes of users, redesigning software system modules, redesigning the run-time representation strategy for programs, redesigning the basic hardware allocation scheme and many other strategies. In order to determine which of these factors is the critical one, some means of measuring system performance must be devised, and the behavior of the system under changes in system design parameters must be measured. The measurement of system performance will be further discussed in a later section.

Virtual processors

Resource-independent information structures

One of the principal differences between batch-processing programming systems lies in the degree to grammed programming systems lies in the degree to which a user program has control over physical computer resources during the execution of his program. In batch-processing programming systems, machine language programs are permitted in which the user decides for himself how physical resources are to be allocated during program execution, and has complete control over the real time sequence of events within the computer during execution of his program except in exceptional circumstances which cause interrupts. In a multiprogrammed computer system, the programmer has control over the time sequence of events in his own program, but has little explicit control over the allocation of computer resources among different programs in the programming systems.

A multiprogrammed system allocates scare computer resources to programs during execution. Since the physical resources allocated to a program may be different on different instances of execution, it is essential that a multiprogrammed computer system provide facilities for the run-time representation of programs in a manner that is independent of the physical resources they will occupy during execution.

It will be assumed that the physical computer resources are approximately as follows:

1. Several hundred thousand main memory registers addressable by a linear sequence of integer addresses.
2. One or more processing units having access to a common main memory.
3. Several hundred million registers of fast auxiliary memory with a block access time of a few miliseconds.
4. Data channels to a wide unpredictable variety of input-output devices such as tapes, printers, card readers, typewriter consoles, direct data channels to on-line equipment, scopes, etc.
5. A number of meters and clocks for measuring resource usage.

In a batch-processing programming system the above resources can be directly addressed at the machine language level. In a multiprogrammed system the allocation of resources to information structures associated with a particular user is performed dynamically by the programming system. It is therefore convenient to store the information structures in a hardware-independent manner during execution.

Virtual machine language

The hardware-independent run-time representation of instructions will be referred to as *virtual machine language* to emphasize that it is a hardware-independent representation. The computer system is designed to execute programs specified in virtual machine language rather than programs in a more hardware oriented language. The virtual machine language programs may be thought of as being executed *interpretively* by the programming system. Like every interpretive system, a penalty is paid in that there is an interpretive overhead in the execution of individual instructions. In the multiprogrammed systems considered below only the address field is interpreted, and indirect addressing hardware is used to reduce the interpretive overhead.

The principal reason for choosing a run-time representation which must be interpreted arises from the requirement that the run-time representation be hardware independent. However, once the decision for an interpretive run-time representation has been made, other benefits associated with interpretive languages can be exploited. The run-time representation

can be chosen so that it is a clean and logical language. Additional flexibility of control sequencing, diagnostics, and control over access, can be provided by interpretive control bits encountered during indirect addressing.

Interpretation is normally restricted to indirect addressing, but may in certain cases require system programs to be executed. Such system programs are referred to as *hardware management routines.* Hardware management routines of a computer system are equivalent in their effect to microprograms which modify the primitive hardware structure of the computer and give the user the illusion of a more civilized environment. However, hardware management routines are implemented by software, and may require a considerable programmed overhead to achieve their effect during execution.

The term "virtual memory" will be used to distinguish the memory seen by each user from the physical memory of the actual computer. The concept of a hardware independent "virtual address" will be defined and distinguished from that of a physical register address. The concept of a "virtual processor" or "virtual computer" is defined as the computer configuration which each user sees when he writes his program, and distinguished from the physical computer that is actually available. It will be assumed that a multiprogrammed physical computer can cope with an indefinite number of identical hardware-independent virtual computers. A virtual computer has hardware-independent *virtual registers* and a *virtual processing unit.* Each programmer programs his virtual computer as though it were a physical computer all of whose resources are dedicated to execution of the program specified by the programmer. The programming system allocates physical facilities of the physical computer to virtual facilities of each virtual computer as they are required.

Although the virtual machine language cannot refer to physical storage registers, some form of addressing must be available within the virtual machine language. The set of all addresses available to the user will be called the *virtual address space*, and individual addresses in the virtual address space will be called *virtual addresses.* All information items accessible in a given program are referred to by virtual addresses. Information that is placed in a given virtual address is assumed to remain in that virtual address unless it is modified or moved, just as information in a conventional computer. However, the programmer has no control over the physical storage medium in which virtual addresses are stored. The correspondence between physical and virtual addresses is completely under the control of the computer system. It is the responsibility of the computer system to move locks of information about in the physical memory hierarchy so that information appears in the main memory when it is required for processing, and is retired to auxiliary memory when no longer required, to make room for other blocks of information.

The programming system must provide facilities not only for moving blocks of information in the physical storage hierarchy, but also for accessing the physical register corresponding to a given virtual address when such access is required during execution. The correspondence between virtual addresses and physical addresses is stored for each program in a set of *address mapping tables,* which are updated whenever a block of information is moved within the physical storage hierarchy, and used for table look-up whenever access to information specified by a virtual address is required during execution. The structure of the address mapping table depends on the relation between the virtual address space and physical address space and also on the hardware facilities available for performing address mapping. The structure of address mapping tables will be further discussed below.

Virtual address space organization and two-component addressing

Since the virtual address space is hardware independent, the system designer has considerable free dom in designing the virtual address space. In designing a virtual address space the following factors must be considered.

1. The virtual address space must be related to the physical address space in such a manner that mapping virtual addresses to physical addresses through the address mapping table can be performed reasonably rapidly.

2. Within the constraints imposed by 1, the virtual address space should be designed for the convenience of the programmer.

Programmers find it convenient to subdivide the information structures of a computation into program and data segments which correspond to logical subdivisions of the problem. The virtual address organization described below structures the address space into a set of *segments* which can be independently named, so that logical segments of a computation can conveniently be mapped into segments of the virtual address space. Information structures within a segment are referred to by a *two-component* virtual address (i, j) where i specifies the segment address (segment name), and j specifies a word-within-segment address. In the discussion below some of

the design considerations which determine the form of a two-component address space are given.

The simplest form of virtual address space is a one-dimensional sequence of virtual addresses running from 0 through $2^n - 1$ for some n. In choosing the size of the virtual address space we are not restricted to the size of any specific physical storage medium. Techniques are discussed below which permit the size of the virtual address space to be independent of the number of bits in virtual-machine-language instructions.

The number of address bits in an instruction can be reduced if the convention is adopted that the address field contains merely a *displacement* relative to an origin specified in a special register. If the maximum displacement permitted is 2_ℓ then a main address field of ℓ bits is sufficient, independent of the size of the address space. Special registers which specify the origin with respect to which displacements are measured are referred to as *relocation registers or base registers.*

The number of bits required in a base register to specify the origin for purposes of relocation can be reduced by p bits if the convention is adopted that origins can occur only at registers which are multiples of 2^p. If the maximum displacement is 2_ℓ then it is convenient to choose $p = \ell$, so that an increment of 1 in a base register is associated with an increment of 2_ℓ in the address space. When this convention is adopted, then addresses in an address space with 2^n addresses can be represented by a k-bit base register address on an ℓ-bit main memory address where $k + \ell = n$.

The above organization structures an address space of 2^n addresses into 2^k blocks each of which contains 2_ℓ words, where $K + \ell = n$. The resulting blocks will be referred to as *segments**. The contents of the k-bit base register will be referred to as a *segment address* and the ℓ-bit address in the address field will be referred to as a *word-within-segment* address.

Addressing by means of a segment address and a word-within-segment address is referred to as *two-component addressing*. Two-component addressing allows a very large address space to be defined without unduly increasing the number of bits in the address field. For example, in the IBM 360 Model 67, an address space of 2^{32} words is defined by 12-bit segment addresses and 20-bit work-within-segment addresses, while in the GE 645 machine an address space of 2^{36} words is defined by 18-bit segment addresses and 18-bit word-within-segment addresses.

In the scheme described above the segments may be thought of as being laid end to end in the address space so that the last address on one segment is a neighbor of the first address of the next segment. However, the address spaces associated with different segment addresses may be made truly independent of each other by suppressing carries from the most significant bit of a word address into a segment address, and causing either an end around carry or an error interrupt whenever such a carry occurs. When this is done the two address coordinates become truly independent, and the address space becomes a set of independent segment address spaces.

When the address space being considered is a virtual address space, then physical addresses need be assigned only to those elements of the address space in which information is actually being stored. This allows extravagant provisions to be made for the possible growth of segments stored in the address space without committing physical resources to unused portions of the segment. In a virtual address space of this kind the problem of dynamic storage allocation is solved by very sparse use of the address space so that there is almost always room for structures to expand. Physical storage for structures in the virtual address space of a given program is provided by a "hidden" allocator whose characteristics are further discussed in a later section.

Each user programs as though he has his own virtual processor with a private virtual address space. There is a set of address mapping tables which are consulted during execution to determine the physical address. During the course of the computation a given virtual address may at different times correspond to a number of different physical registers of the memory hierarchy. The system keeps track of blocks of information by updating the address mapping tables of the associated virtual computer whenever a block of information is moved.

The use of address mapping tables not only permits the same virtual address to be represented by different physical addresses at different points of the computation, but also permits addresses of two different virtual memories to denote the same physical address and thereby to have access to the same common information. In particular, the address space of every virtual processor permits access to a common set of system routines.

The virtual memory of every virtual processor may be thought of as being *initialized* so that it has a standard set of initial system facilities resident in its virtual memory.

*The term "segment" is used in different ways by different computer system designers. This definition does not allow segments to be truly independent because of carry from the ℓth to the $\ell + 1$th position. Truly independent segment naming requires suppression of the carry as indicated below.

Auxiliary memory and user communication

Although the virtual memory of each user is very large, there may still be programs for which the virtual memory is not large enough and for which auxiliary memory is therefore required. The requirement of hardware independence applies to auxiliary memory as well as to the main memory. The auxiliary memory accessible to a user will be called the virtual auxiliary memory.

Two alternative approaches can be adopted to auxiliary memory space.

1. Each user has a private virtual auxiliary memory space.
2. There is a system wide virtual auxiliary memory space.

The second approach is the one adopted in the IBM and Multics systems and will be illustrated below.

The system wide virtual auxiliary memory will be referred to as the *file system.* The file system may be described by a directed graph with an initial vertex called the *root vertex* and a number of terminal vertices. The terminal vertices correspond to information blocks and the non-terminal vertices consist of sets of pointers to lower level vertices. The sets of pointers associated with non-terminal vertices are referred to as *catalogs* (IBM) or directories (Multics). The directory (catalog) associated with the root vertex of the file system tree structure is referred to as the *root directory.**

The information structures associated with vertices in the file system will be referred to as files. Access to all files in the file system must pass through the root directory. Each file in the file system has a *tree name* which consists of a sequence of pointers through successive directories terminating in a pointer to the file itself. The tree name is the address of the file in the virtual auxiliary memory. A given file in the file system may in general have more than one tree name, corresponding to different paths through the graph structure from the root directory to the file. However, the convention is usually adopted that there are no loops in the graph which represents the file structure; i.e., the vertices of the file system constitute a partial ordering.

The set of physical storage registers in which files of the file system are stored may vary during execution. The correspondence between addresses in the virtual auxiliary memory and physical registers is determined by a system wide *file system address mapping table.*

*Specific auxiliary memory designs along these lines are given in (2), and by Daley and Neuman in (3).

The system-wide virtual auxiliary memory fulfills the following functions:

a) It stores information structures private to individual computations which it is inconvenient to store in the main virtual memory.

b) It serves as a common information base which stores individual program and data segments that are generally available to all computations or selected classes of computations.

c) It can be used for purposes of communicating between computations.

In order to ensure privacy of information in categories a) and c), and freedom from unauthorized modification of information in all categories, there are means of restricting the form of access to information stored in the virtual auxiliary memory. The modes of permitted access may be a combination of the following:

X The segment may be executed as a program.
R Reading from the segment is permitted.
W Writing of information into the segment is permitted.
A Changing the size of the segment is permitted.

The mode of access permitted to a given segment in the auxiliary memory is not determined solely by the segment being accessed but by the relation between the accessing process and the accessed segment. This effect can be achieved by encoding the mode of access in the sequence of pointers that constitute its tree name. For example the mode of access determined by a sequence of pointers can be taken to be the mode of access associated with the last of the pointers.

The above logical attributes of segments in an information structure are represented at the physical level by bit patterns in address mapping tables which are interpretively interrogated during execution. When a segment is "moved" from auxiliary memory to a given virtual address space, the accessing bit patterns which determine the mode of access are initialized in the address mapping tables of the virtual processor.

When a user is given permission to use the system he is allocated a standard *initialized virtual processor,* with access to a standard set of system programs in his main addressing space and access to a standard set of files in the file system in standard accessing modes. During execution he may build up information structures both in his virtual address space and in the file system. However, he may also wish to request access to information in the file system that is not made available on an automatic basis. Two categories of information in this class may be distinguished.

1. System files for which access requests are channeled through the computer operator and are made available by an action of the computer op-

erator, possibly after consultation with the system administrator of the computation center.
2. Private files for which access requests must be made directly to the user having control over these files.

The system must contain facilities for granting of access to privileged files both by system administrators and by private system users. A set of primitive system operations for allowing such access is discussed in (5).

Mapping of an object from the virtual auxiliary memory to the virtual main memory of a virtual computer does not require moving of the information itself but merely updating of the address mapping tables of the virtual computer to establish the correspondence between the physical registers of the information item and the virtual main memory address with which it has become associated. However, in performing a mapping between the virtual auxiliary memory and the virtual main memory, information regarding the mode of access to the information must be preserved. The encoding of accessing information in the main memory address mapping tables is further discussed in a later section.

Virtual computers

Each user of a multiprogrammed computer system has at his disposal a virtual computer with a virtual address space which is initialized to have access to a standard set of system facilities. During the lifetime of a given computation the user may introduce his own information structures into his virtual address space, and introduce information structures from the virtual auxiliary memory into his virtual address space.

A virtual computer has an associated *stateword* which contains the information that resides in the processing unit when the process is being executed. However, the stateword has an existence as an information structure independently of whether it is loaded into a physical processing unit. When the stateword occupies a processing unit, the computation associated with that virtual computer is said to be *active* or *running*. When the stateword is stored in the main or auxiliary memory, the computation is said to be *passive* or *blocked*.

The stateword of a virtual computer C contains information stored in processing unit registers such as the accumulator and instruction location register. It also contains a pointer to the address mapping tables which determine the correspondence between virtual and physical addresses for the given computation. The pointer to the address mapping tables links the stateword to all information structures of the virtual computer associated with the stateword. The term *computation* will be used to denote the sequence of instantaneous descriptions associated with a given virtual computer.

The transition from a computation C_1 to a computation C_2 on a given processing unit is accomplished by storing the stateword associated with C_1 and loading the stateword associated with C_2 into the processing unit.

Loading of the new stateword automatically causes a new set of address mapping tables to be used in interpreting address fields. The address mapping tables are used both in the instruction fetch phase and in the instruction execution phase as indicated below.

Moving of information in the physical memory during a computation causes changes in the address mapping tables of the associated virtual computer but not in the virtual addresses of the moved information. Location independent virtual addresses and location independent pure procedure segments are made possible by the expedient of interposing an interpretive address mapping phase into the computation. The address mapping tables rather than the address itself are modified whenever the physical address changes during execution.

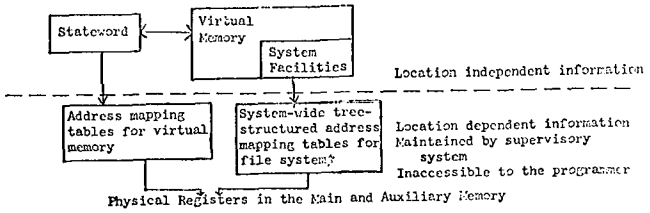

Figure 1.—A virtual computer with address mapping tables that determine the physical location of accessible information structures

Figure 1 illustrates the relation between the information structure which constitutes a virtual computer and the physical registers in which the information structure is stored.

Measures of system efficiency

When a virtual computer is in execution, it is occupying a time slice of a physical processor. It is also occupying time slices of a number of other resources. It is the job of the computer system to allocate time slices of resources to virtual processors so that computations specified by users can be executed both efficiently and flexibly.

One of the measures of efficiency is the proportion of the time that a physical processor spends doing computations specified by virtual processors of users. This proportion is less than one because a processor may be idle or perform administrative system func-

tions such as process interchange or memory allocation. System functions such as input-output are provided as a service to the user and charged to the user. System functions such as accounting or interrupt routines that are not explicitly requested by the user but form part of the system overhead for a computation may also legitimately be charged to the user.* However, certain system wide computations and certain kinds of excessive administrative overhead due to system inefficiency cannot legitimately be charged to the user.

One of the features of a multiprogramming system is that the user is never charged for idle time on a physical processor. If a given physical processor becomes idle because the information required to execute an instruction is not in the physical main memory, then the current computation is interrupted while the information is brought into core, and execution of some other process is initiated on this processor.

More precise data on the factors which affect the efficiency of through put in a computer system can be obtained by breaking down the time spent in various system functions into categories such as interrupt servicing, resource allocation, resource accounting, etc., and measuring the time spent in each of these activities, the proportion of idle time due to each of these activities, and the change in these times brought about by the change of certain design parameters.

Efficiency is measured above in terms of the efficiency of processor utilization. We shall consider next the effect of the problem mix on the efficiency of processor utilization.

Foreground and background processes

Computations which have elapsed time deadlines for their completion are called *foreground processes.* Real time computations, interactive computations in which a user at a console expects an "immediate" response, and debugging runs are examples of foreground processes.

Computations for which there is no pressing real time deadline are referred to as *background processes.* Long production runs and batch-processing runs for which an elapsed time of more than a few minutes is acceptable are examples of background processes. Background processes usually make less stringent demands on input-output resources than foreground processes. The running of programs as background processes can be encouraged both by lower charging rates and by system rules.**

*The user of electricity has to pay for the electricity used in running his electric meter.

**The classification of processes into foreground and background processors is a special case of a classification into priority classes, each having its own scheduling and accounting algorithm.

Foreground processes tend to make heavy use of input-output facilities while background processes tend to make heavier use of processor time. In order to avoid situations in which a processor is idle because there are no processes waiting to be processed, it is desirable to include in the problem mix a number of background processes which make heavy use of the processor and relatively light use of other resources. Multiprogramming systems are specifically designed to take advantage of variation in the resource requirements of different computations for the purpose of improving the average overall efficiency of resource utilization.

Processing unit organization for two-component addressing

Representations of identifiers

One of the most important problems in programming at all language levels is the representation of identifiers. There are at least four levels of representation of names that must be considered in a multiprogramming system.

1. The source language level
   Names at the source language level are symbolic, e.g., X, X(I) etc. The association of names with objects they denote is determined partially by context and partially by *declarations* which specify *attributes* associated with the given name.
2. The instruction address level
   The contents of the address and special register fields of an instruction determine a rule for computing an address. The address is determined both by the contents of address and special register fields and by the contents of explicit and implicit registers used in the address computation.
3. The virtual address level
   The modified address obtained from an instruction address by indexing, indirect addressing and other forms of address computation results in a virtual address. In the multiprogrammed computers considered here the virtual address is a two-component address with a segment component and word within segment component.
4. Physical register address level.
   A physical register address is the address of a physical register in the main memory.

The translation from a source language name to an instruction address is accomplished by a compiler which translates source language into object language. The translation from an instruction address to a virtual address and from a virtual address to a register

address is accomplished by computer hardware during execution.

In the present section the translation from instruction addresses to virtual address will be considered. The translation from virtual addresses to physical addresses will be considered in the following section.

It will be assumed that each processing unit contains a two-component *address register* comprised of a *segment register* and a *word register.* During execution the address register contains the two component address of the next instruction to be executed. The basic instruction execution cycle is as follows:

1. Fetch an instruction from the virtual address specified in the address register.
2. If a data access is required fetch the data from the virtual data address specified by the instruction.
3. Execute the instruction.

Two-component instruction fetch

The instruction fetch phase makes use of a register referred to as the *temporary address register* comprised of a *temporary segment register* and a *temporary* word register. During the instruction fetch phase the address register is moved to the temporary address register, the word register is incremented by 1, and the content of the temporary register is used to compute the physical register containing the instruction as illustrated in Figure 2.

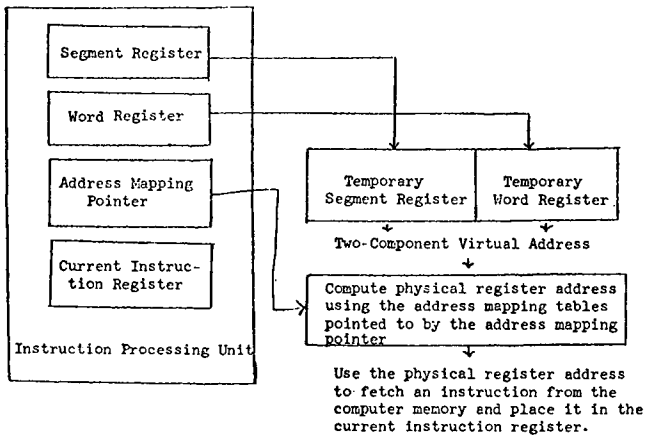

Figure 2. – The instruction fetch phase

An address mapping pointer stored in the instruction processing unit is used to determine the physical origin of the address mapping tables to be used in the physical instruction address.

The determination of the virtual address in the instruction fetch phase is trivial since the two-component virtual address is explicitly stored in the processing unit. However, the effective address computation in the data fetch phase involves converting the address fields of a one-address instruction into a two-component physical address and is considerably more complex. Assume that the one address instruction has the following format.

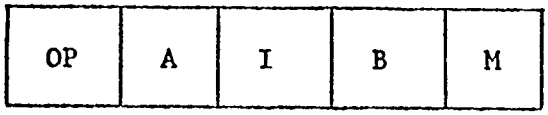

Figure 3. – Two-component instruction address format

OP is the *operation code field,* A is the *address field,* I is the *index register field,* B is the *base register field* and is used to point to one of a number of *base registers* containing a two-component base address, and M is a *modifier field.*

Two-component data fetch

The simplest form of address computation occurs when the I, B and M fields specify no modification. In this case A is assumed to be an absolute word address in the segment being executed, so that the segment address is automatically taken to be that of the segment register in the instruction processing unit.

Indexing using an index register pointed to by the I field, and indirect addressing using a bit in the M field are assumed to operate on the word address A just as though it were a one-component address.

If no base register is specified, then the segment address is always assumed to be that of the currently executed segment. However, if a base register is specified, then the segment component of the base register becomes the segment component of the effective address, and the word component of the base register is used to increment the word component of the effective address, just as though it were an exta index register.

The effective address computed during the data fetch phase is stored in the temporary segment register and temporary word register just as in the instruction fetch phase. If indirect addressing is specified, the register address corresponding to this effective address is used to replace the A, I, and M fields by the A, I, B and M fields of the fetched instruction, and to initiate a further effective address computation. Otherwise the content of the register address is used as the data item for the current operation.

The address computation during the data fetch phase is illustrated in Figure 4.

The above machine language instruction format is basically a one-address instruction format in which the second component is specified by a pointer to a two component base register. This requires the segment number to be set by special base register loading instructions prior to use of a given segment. The word component of the base register may be thought of as a relocation register which determines as a relative initial address within the segment. Because of this

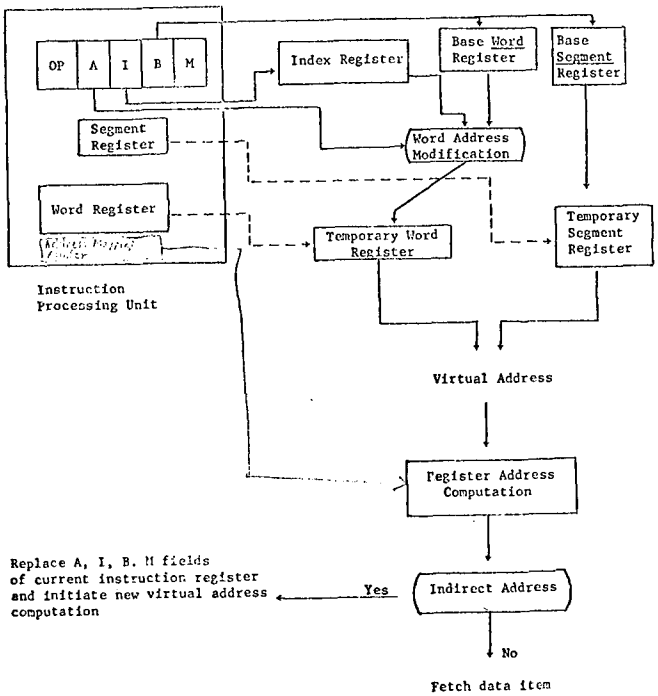

Figure 4. – Effective address computation during the data fetch phase

relocation facility the word address can be regarded as a relative rather than absolute word-within-segment address, and the number of address bits in an instruction word need not be the full segment size.

Machine language programming for a computer with two-component addresses requires the programmer to keep track of both index registers and base registers addresses and is therefore more complex than on a machine with fewer address modifiers. However, if standard conventions are adopted for inter-segment communication the burden of the machine language programmer can be eased. Clearly programming in machine language is the exception rather than the rule.

Physical register computation under paging and segmentation

Mapping of segments without paging

In the previous section it was shown that conversion from virtual to physical addresses was required both during the instruction fetch phase and during the data fetch phase. The overall features of this address computation are similar both in the IBM and GE address computation, and will be described in greater detail below.

Two-component addressing suggests that each segment of a virtual address space have an independently specified physical origin. As a first approximation to an address mapping scheme we shall assume that each segment of a process must occupy a contiguous block of registers in the memory hierarchy. In this case the initial segment address would completely specify all storage allocation information about the segment. The address mapping tables would consist of a *segment table* with one entry per segment specifying the initial address of the segment if it is in main memory and a *segment not in core* marker if the segment is not in main memory.

The stateword of a processor contains a word called the *address mapping* pointer which points to the first address of the segment table. The segment table entry for a particular segment is obtained by relative addressing relative to the address mapping word.

The above scheme is clearly impracticable when the segment size is of the same order of magnitude as the number of registers in the main memory. Since information structures stored in segments normally occupy only a small initial portion of the segment, the above scheme would also be very wasteful. One modification which would make the above scheme more practicable would be to allocate memory only to the initial portion of the segment that actually contains information. This would require a specification of both the initial address and the segment length in the address mapping table. On access to the segment the system would first check the segment not in core marker. If the segment were in core it would check that the word address was less than the segment length. Access to the segment would be performed only if these checks were satisfactory. Otherwise an interrupt would be initiated causing the system to take some action.

The above checks illustrate some of the advantages of interposing an interpretive address mapping between the virtual and physical addresses. Multiprogrammed systems take advantage of this intermediate stage of interpretation in other ways too.

Segment attributes

The entry for each segment in the address mapping table may be thought of as a "description list" which specifies segment wide accessing attributes. The accessing attributes so far introduced are location, length and the property of being in core. Other attributes which may conveniently be specified in this de- ent modes of access may conveniently be distinguished.

1. Access which involves *reading* words of the segment
2. Access which involves *writing* words of the segment
3. Access which involves *executing* words of the segment
4. Access which involves *adding* or *deleting* words of the segment.

These four modes of access are referred to respectively as the *read mode, write mode, execute mode* and *append mode,* and will be denoted respectively by R, W, X and A. Each mode of access may be controlled by a single bit in the segment description list. If during execution a form of access which is not permitted is attempted, then an interrupt occurs resulting in a system action.

A restriction on the mode of access to a segment may be thought of as a *mode of protection* of that segment from interference by other segments. Since mode of access and mode of protection are reverse sides of the same coin, the terms access and protection will be used interchangeably.

Although allocation of only the used portion of a segment is a great improvement over allocation of the whole segment, it may still lead to difficulties.

1. Variable size segments make the problem of storage allocation when a new segment is introduced into the memory very complex.
2. One or two very large segments may use up the whole of physical memory and therefore present efficient multiprogramming.

Pages

In order to avoid both of the above problems it is convenient to choose a fixed size unit for purposes of storage allocation which is independent of segment size and sufficiently small so that a large number (say 1,000) of these units may simultaneously reside in the main memory. This unit will be called the *page.*

The number of words in a page will be chosen to be a power of 2, say 2^m. If the number of words in a segment is 2^n, $n>m$, then each segment will be subdivided for purposes of storage allocation into 2^k pages where $k + m = n$. Since each page of a segment can be mapped independently into a block of storage, an initial address and storage not in core indicator is required for each of the pages of a given segment. This information is stored in a *page table.*

When pages are used as the unit of storage allocation, then address mapping consists of two stages of indirect addressing through the segment table and the page table associated with the segment. Each stage of indirect addressing may have associated with it certain interpretive tests triggered by indicators stored along with pointer information in the address mapping tables. Attributes that are associated with the segment as a whole are stored in the segment table. These attributes include the location of the segment page table, the access mode of the segment, the length (number of pages) of the segment, etc. Attributes of individual pages include their location, whether or not they are in core, etc. Thus the two stage interpretation process permits testing for run time segment attributes to be separated from attributes associated purely with the storage allocation process.

The physical memory of the computer is subdivided into pages for purposes of storage allocation. The virtual memory of each virtual processor is also subdivided into pages. When a virtual processor is initiated by placing its stateword into the physical processor the majority of its pages normally reside in the auxiliary memory. If, during execution access to a page which is not currently in core in required, the absence of the page will be discovered during address mapping and a missing page fault will occur. The missing page fault will cause a system program for memory allocation to allocate a page in core for the required page, possibly retiring an existing page to the auxiliary memory to make room for the new page. The virtual processor will become blocked while the memory allocation mechanism brings in the required page, giving up the physical processor to some other virtual processor that can proceed with its computation. The given processor will reactivated when the page has been read into the main memory. When it regains possession of a physical processor it will again access the required page through the address computation mechanism and will this time succeed.

It is assumed that segments stored in the auxiliary memory occupy a contiguous set of physical registers in the auxiliary memory. In order to retrieve a missing page from the auxiliary memory, a table must be available which specifies for each segment of a process the tree name or physical auxiliary memory address for that segment. This table is referred to as the *segment name table.* Thus the relation between virtual and physical addresses is in fact determined by two tables. The segment table specifies physical addresses for segments which are in core and the segment name table specifies physical addresses for segments that are not in core.

It was assumed above that the page table of the segment containing the page being accessed was in core. The page table of a segment will itself occupy a page of main memory and need be created only when at least one of the pages of the segment are in core. If the page table is not in core then a *missing segment fault* will occur at the segment-table stage of indirect addressing. A missing segment fault will cause the system to allocate a page for the page table, create a page table for the segment with missing page faults in all its entries, and return control to the interrupted program. Note that no information from auxiliary memory is actually required when setting up a page table, so that the page table can be set up by the system by merely borrowing the processor that requires

the page table. However, time would be required to retire a page if no pages were available. In order to reduce storage allocation waiting time associated with pages that are being retired, four or five vacant pages are normally available in the main memory, and a page is retired whenever the threshold of vacant pages falls below this level.

When the number of segments in the virtual address space is very large, it is no longer possible to have the complete segment table of the virtual processor in the main memory. This can be avoided by allowing the segment table itself to be paged. The segment table can be paged without any extra machinery by allowing it to be a segment of the virtual processor.

When the segment table is paged then the address mapping word of the stateword of the virtual processor points not to the segment table but to the page table of the segment table. Access to a physical register now requires three stages of indirect addressing through the page table of the segment table, the segment table itself, and the page table of the segment.

Address mapping under paging and segmentation

When both the segment table and the segment address are paged a two component virtual address (i,j) effectively becomes a four component address (k,m;l,m) where the segment table page table contains 2^k entries, the page tables of individual segments contain 2_ℓ entries, and pages contain 2^m entries. Since page tables themselves occupy pages of the computer memory, k and ℓ must not exceed m, and should be chosen to be m for maximum memory utilization. Example: If the address space permits 2^{18}, then a segments each having a maximum length of 2^{18} then a page size of 2^9 would result in page tables with 2^9 entries. In this case $k=\ell=m=9$.

The segment address together with the page component of the word within segment address is sometimes called a *virtual page address* since it is the address of a page of the virtual memory.

The three stage indirect address computation which results when both individual segments and the segment table are paged is illustrated in Figure 5.

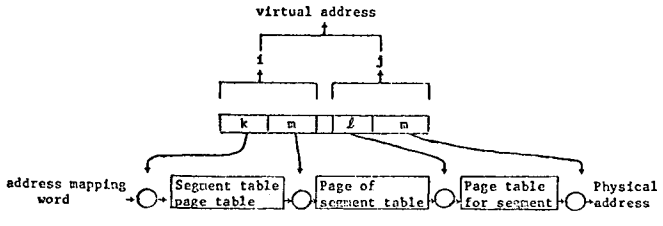

Figure 5. – Address computation under paging and segmentation

The address mapping word of the processor is modified by the first (k bit) component to determine a page of the segment table. The initial address of this page is modified by the second component to determine the segment table entry. The segment table entry is incremented by the third component to determine the register which specifies the initial page address. Finally the initial page address is incremented by the fourth component to determine the physical register address.

Each of the above stages may result either in a missing information fault or in an accessing gault due to accessing attributes associated with the information access not being met.

Associative registers

The effectiveness of a system in which memory allocation is performed by paging depends in part on the characteristics of the information structure on which the computation is being performed. Paged storage allocation is most effective for computations in which there are long sequences of instructions whose information requirements are restricted to a small number of pages. If the number of pages to which access is required in a computational sequence is large then the computation will require a large number of in core pages to run without interruption. In this case a choice must be made between allowing the computation to occupy a disproportionate amount of main memory thereby impairing the efficiency of other processes, or executing the process in a highly inefficient manner, constantly retiring pages that will again be required at a later point of the process.

The overall efficiency of a computer system under paged storage allocation depends in large measures on the information accessing characteristics of the "average" process in the system. If the average process has long computational sequences requiring only a small number of pages then a small number of "memory-eating" processes with large storage requirements can be tolerated. However, if the typical process accesses large numbers of pages intermittently during most of its computational life then paging may not provide a sufficient economy of storage allocation to justify the time and space overhead that it introduces.

Paging introduces a time overhead by requiring extra indirect addressing during execution of individual instructions and by requiring system actions during allocation and retiring of pages. It introduces space overhead by requiring extra space for address mapping tables and by requiring space for system programs and their address mapping tables. One of the factors which determines whether paged storage al-

location can succeed is the degree to which the time and space penalty of paging can be reduced.

The time and space penalties introduced by paging can be reduced in part by hardware and in part by efficient system organization. For example, there is hardware for automatic (non-programmed) indirect addressing from the processor address mapping word through page and segment tables to the physical address. This reduces the time penalty to three memory cycles per memory access. This time penalty can be further reduced by means of a special set of hardware registers known as *associative register.*

An associative register is a register that is addressed by its content rather than by an address. The associative registers used to speed up the address computation contain direct correspondence, between a small number of virtual page addresses and corresponding physical addresses. The content of an associative register is illustrated in Figure 6.

Virtual page address	Physical register address	Statistical Usage information

Figure 6. – Associative register format

A processor typically has eight or possibly sixteen very rapid access associative registers in which recently accessed virtual page addresses and corresponding physical addresses are stored.* Whenever access to a given virtual address is required, the associative memory is scanned for the virtual page address. If the virtual page address is found, the physical address is given in the physical register field of the associative register, and can immediately be used for accessing purposes without performing multistage indirect addressing. If the virtual page address is not present in the auxiliary memory, multistage indirect addressing is performed in the normal manner. The resulting physical address is used not only to access the physical memory but also to establish a new entry in the associative memory for the accessed page, retiring a current entry in the associative memory. The statistical usage information is used to determine which of the current entries the new entry is to replace.

The effectiveness of this scheme depends on the proportion of the time that memory accesses can be accomplished through the associative memory. This in turn depends on the size of the associative memory, the rule for replacing segments of the associative memory, and the type of process mix for which computations are being performed.

*Note that the virtual page address is a process-dependent quantity, and that virtual-actual address correspondences are valid only in the lifetime of the process in which they were loaded. It is usual to clear the associative registers when replacing one process by another. However, an alternative scheme is further discussed below.

The technique of allowing rapid access to information on the basis of recency and frequency of use is sometimes referred to as *look behind.* This is to be contrasted with *look ahead* techniques which try to predict the information which will be required by looking ahead in the instruction sequence.

Factors which determine the efficiency of paging schemes

Simulation has shown that a small number of associative registers will in a typical computation require the address computation to be performed less than 20% of the time. Thus the time factor for address mapping during accessing can be considerably reduced. However, it has been found that the bottlenecks introduced by paging do not lie in the time penalty during address computation but rather in space problems in the following categories:

1. The system facilities for paging eat up a large amount of space for page tables and other purposes.
2. Problems tend to require a large number of pages for their execution. Accessing does not tend to be localized to a small number of pages over short time sequences (say 10,000 instruction times) but tends to range quite widely requiring frequent interchange of pages.
3. The time required by a process to build up a sufficient number of pages in the main memory so that it can run for an appreciable length of time without missing page faults tends to be quite long, particularly since missing page faults cause the process to lose control of the processor, and since successive pages can never be read in parallel. Thus building up of a process in main memory to the point that it can run efficiently represents a considerable real time investment. The space constraints may well be such that more time is spent building up the memory investment of processes to the point at which they can run efficiently than is spent in the efficient execution of processes.

The memory utilization of a group of programs in a multiprogramming system can conveniently be measured by a *memory utilization chart.* A memory utilization chart measures space along its horizontal dimension and time along its vertical dimension. The total memory space is represented by a fixed horizontal span and the amount of space occupied by each program is represented by a portion of this span. As time moves in the vertical direction, the space utilization of each program is represented by a vertical band. Figure 7 indicates a memory containing three programs. During the time span indicated in the graph,

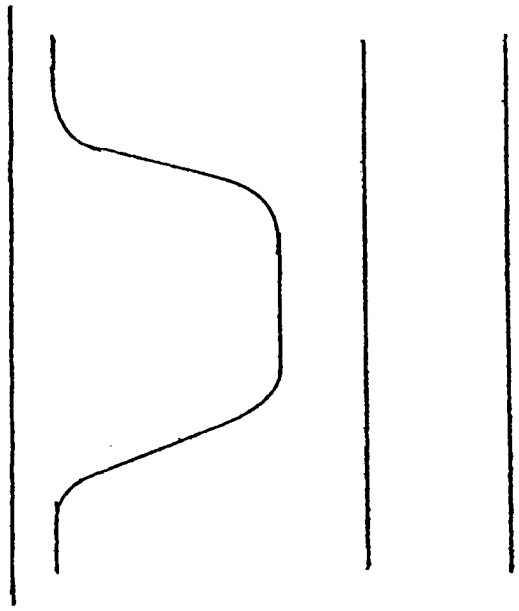

Figure 7. – Time profile of memory utilization

the leftmost program expands and takes up memory space at the expense of the second program, retains a fixed, large portion of the memory for a given time interval, and then relinquishes the space to the second program. During this time, the third program, possibly a background program, retains a fixed amount of space in the memory throughout the time period.

A profile such as that above implies that the leftmost program built up pages to a level at which it was able to run with relatively few interruptions, ran at this level for a while, and was then phased out, allowing the middle program to resume operation. This profile is essentially a healthy one and page allocation techniques must allow programs to quickly build up their page requirements to the level at which they can run in an uninterrupted fashion, and to maintain their complement of pages at this level for a sufficient time so that the real time investment required to build up this complement of pages pays off.

In an actual multiprogramming system the number of programs that can simultaneously share portions of the main memory is considerably greater than three. One significant parameter is the ratio n/k of number of pages n in the main memory and the number of processes k which may simultaneously share the main memory. It has been found that, for a page size of 2^{10} words, a ratio $n/k = 10$ allows sufficient freedom for programs to expand their pages at the expense of others, while a ratio $n/k \leqq 5$ leads to overcrowding of the memory with competing programs.

The dynamic behavior of programs under paging has been simulated in a number of experimental studies such as (6) (7), and the overall conclusion appears to be that "demand paging" for individual pages leads to highly inefficient computer utilization. It is likely that multiprogramming systems of the future will adopt some form of grouped page storage allocation, where the group of pages allocated during a single storage allocation interrupt is determined either by the structure of processes or by the storage requirements during the previous activation of the process.

The efficiency of paged memory allocation would be greatly increased if groups of pages having a high incidence of internal cross referencing and a low incidence of external referencing could be isolated by the supervisory system and moved in and out of memory as a single unit. Groups of pages which are treated as a single unit for purposes of allocation are sometimes referred to as *hyperpages*.

The problem of efficiently partitioning a problem into hyperpages may be thought of as a *clustering problem* in which individual pages are represented by vertices into clusters having high density of traffic within clusters and low density of traffic between clusters. However, it is not clear that significant clustering patterns could be established at a level at which clusters were significantly smaller than complete programs. Moreover, clustering patterns within programs are likely to vary with time, and it is likely that a look-behind technique for paging of individual processes would be more effective than a "static" clustering technique.

Clustering techniques are said to be static because they determine groups of pages that remain fixed throughout execution. When comparing look-behind techniques with clustering techniques, it is convenient to think of the set of most recently used pages singled out by the look-behind process as a single cluster which changes in composition through time.

The look-behind techniques discussed above assume a single system-wide set of associative registers which is cleared on every process interchange. Thus a newly activated process has no initial look-behind information, builds up this information as it goes along, and has its look-behind information destroyed as soon as it loses control of the processor. Since the correspondence between logical and physical addresses might change while the process is not in control, this information would not be of any use when the process regains control, unless provision were made for updating it. However, the information specifying the "cluster" of most recently used pages is an important piece of information and could be used for page control if it were available. It is felt that storage of the set of virtual page addresses on termination of a process as part of an extended stateword, might be a worthwhile hardware extension. For example, if these vir-

tual page addresses were available, then the loaded process could check for the presence of these pages and bring in any missing ones in parallel with the initial part of its computation.

At any instant of time the associative registers contain solely information pertaining to a single process. When the process is terminated, the part of this information that is dependent on physical resources becomes outdated as the system reallocates its resources. However, part of the information tells us about recent page-usage of the process and is as relevant if the process is restarted in a year on a different machine as when it is restarted within a millisecond. Careful use of this information could lead to considerably improved paging algorithms.

Problems of paging are caused essentially because main memory space is a critical resource in current computers. In the long run it may well prove cheaper to expand the main memory to a point where it ceases to be a critical resource rather than to play costly games whose objective is efficient memory allocation. However, even if the memory allocation problem disappears, there will be other critical resources whose allocation will require similar techniques and solutions as those required for efficient memory allocation.

REFERENCES

1 J H SALTZER
Traffic control in a multiplexed computer system
Ph.D. Thesis MIT July 1966
Available from MIT as MACTR-30

2 IBM Systems Research Library, 1966
Basic concepts and facilities

3 Proceedings of the Gall Joint Computer Conference 1965
Six papers on various aspects of the Multics programming system

4 ARDEN et al.
Program and addressing structure in a time sharing environment
JACM January 1966

5 J B DENNIS and E VAN HORN
Programmed semantics for multiprogrammed computations
Comm ACM March 1966

6 G H FINE et al.
Dynamic program behavior under paging
Proc 21st ACM Conference August 1966

7 R W O'NEILL
Experience using a time-shared multiprogramming system with dynamic address relocation hardware
Proceedings Spring Joint Computer Conference April 1967