Dynamic Storage Allocation Systems

B. Randell and C. J. Kuehner

IBM Thomas J. Watson Research Center, Yorktown Heights, New York

In many recent computer system designs, hardware facilities have been provided for easing the problems of storage allocation. A method of characterizing dynamic storage allocation systems—according to the functional capabilities provided and the underlying techniques used—is presented. The basic purpose of the paper is to provide a useful perspective from which the utility of various hardware facilities may be assessed. A brief survey of storage allocation facilities in several representative computer systems is included as an appendix.

KEY WORDS AND PHRASES: segmentation, paging, multiprogramming, storage allocation, storage management, virtual memories, storage fragmentation, storage hierarchies, addressing mechanisms

CR CATEGORIES: 4.30, 6.20

Introduction

As both computer systems and computer applications have become more complex, the problems of storage allocation and their various solutions have undergone considerable evolution.

In early computer systems, programs were run one at a time, each program during its execution having access to the entire facilities of the machine. In simple cases a programmer would have sufficient working storage (e.g. core storage) to contain all the program code and data needed to run his program, and the problem of storage allocation was minimal. Assembly programs could be used to permit a programmer to refer to storage locations symbolically. The actual assignment of specific addresses for execution would then be performed during the assembly process or while the assembled program was being loaded into storage.

For cases of insufficient working storage, the programmer had to devise a strategy for segmenting his program and/or its data, and for controlling the "overlaying" of segments. Thus during the execution of the program, the demand for and the supply of working storage would be matched by keeping temporarily unneeded segments in backing storage. The simplest strategies involved preplanned allocation and overlaying on the basis of worst case estimates of storage requirements. Where these "static" procedures were judged unsatisfactory, algorithms that achieved the desired allocation strategies had to be incorporated in the program and had to be applied "dynamically" as program execution progressed. In fact, unless a compiler was sufficiently sophisticated to automatically provide an allocational strategy (whether static or dynamic), even programs written in high level languages had to contain explicit provisions for storage management.

In many current computer systems, programs are made to coexist in working storage so that multiprogramming techniques can be used to improve system throughput by increased resource utilization. Similarly, such coexistence is desirable if time-sharing techniques are to be used to improve response times to individual users. Thus the operating system has to perform a storage allocation function, allocating storage among the various coexisting programs. Since the arrival and duration of these programs will in general be unpredictable, this allocation has to be performed dynamically. Clearly if such operating system controlled dynamic allocation is to be really effective, the storage resources provided for an individual program must vary from run to run, and even during the duration of a run. In such circumstances even the allocation of storage for an individual program cannot be performed statically.

In fact storage allocation has come to be regarded as one of the basic responsibilities of the computing system itself. Furthermore the addressing facilities used by a program to access data have been made independent, to a greater or lesser extent, of the physical locations at which the data is stored. As a result programs, and even major sections of the operating system, can be written without having to include procedures implementing storage allocation strategies into their coding.

This assumption of responsibility for storage allocation has lead to the provision of special hardware facilities in several recent computer systems. The facilities vary considerably in function and in design. In this paper an attempt is made to identify and evaluate the basic characteristics of storage allocation systems and the storage addressing facilities that they provide. A brief survey of several computer systems which contain special provisions for storage allocation is included as an appendix. Thus it is intended to provide a perspective for a comparative assessment of the various hardware facilities, and the storage management systems that have been built up around them.

Storage Addressing

The information stored in a computer is in general accessed using numerical addresses. This leads to problems such as relocatability and the need to allocate groups of consecutively addressed (i.e. contiguous) locations.

The ability to relocate (i.e. move) information requires knowledge of the whereabouts of any actual physical

Presented at an ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1–4, 1967.

storage addresses (i.e. absolute addresses) included in the body of a program, or stored in registers or working storage, since these will have to be updated. The most convenient solution is to insure that there are no such stored absolute addresses, because all access to information is via, for example, base registers or an address mapping device. Techniques for dealing with the problem when stored absolute addresses are permitted are often very complex—extended discussions of this topic are given by Corbato [3] and by McGee [19].

The provision of one or more groups of consecutively addressed locations is needed so that address arithmetic can be used to access information within a group of locations. (Instruction fetching on a 1-address computer is a special case of this.) The extent to which contiguity (either actual or apparent) is provided is one of the distinguishing characteristics of dynamic storage allocation systems.

Other aspects of storage addressing, such as facilities for interprogram communication, storage protection, and access control, are somewhat beyond the scope of this paper, although some mention of them is made in a later section. Papers dealing with these subjects include those of Dennis [6], Evans and Leclerc [7], and McGee [19].

Basic Characteristics of Dynamic Storage Allocation Systems. The four characteristics believed to be the most useful for revealing the functional capability and underlying mechanisms of current hardware-assisted dynamic storage allocation systems are related to the concepts of:

1. Name space.
2. Predictive information.
3. Artificial contiguity.
4. Uniformity of units of storage allocation.

The discussions of these characteristics given below incorporate a set of definitions (sometimes implicit) for terms such as "segment" and "page." Where possible these conform to earlier usage, although not all the terms have been uniquely defined in earlier papers.

Name Space. Name space has come into usage as a term for the set of names which can be used by a program to refer to informational items. Variations in the name space structure provided are of immediate importance to the user of a system, because of its direct influence on the way programs are written.

By far the most common type is the linear name space, that is one in which permissible names are the integers $0, 1, \cdots, n$. In many basic computer systems such a name space is in fact the set of absolute addresses used to access working storage. The extent of the name space is directly related to the number of bits used to represent absolute addresses. The next level in sophistication is obtained in many systems by providing a relocation register, limit register pair. All name representations are checked against the contents of the limit register and then have the contents of the relocation register added to them, in order to produce an absolute address. Thus a linear name space, whose size can be smaller than that provided by the absolute address representation, can be used to access items starting at an arbitrary address in storage. However, as described below, a linear name space need not have a direct relationship to the set of actual physical storage addresses, nor be limited in size by it.

The second common type of name space is the *segmented name space*. This is essentially a name space composed of a set of separate linear name spaces. Each linear name space is used to access an ordered set of information items which have been declared to constitute a *segment*. A segmented name space is often described as a *two-dimensional name space* since it is necessary to specify two names (name of segment, name of item within segment) in order to access an item. In the most general system the various segments can have different extents. Moreover, the extent of each segment can be varied during execution by special program directives. Furthermore, segments can be caused to come into existence, or to cease to exist, by program directives. Segments possessing these attributes will be referred to as dynamic segments.

Examples of computer systems providing a linear name space include the IBM 7094 and the Ferranti ATLAS, while examples of systems providing a segmented name space include the Burroughs B5000 and MULTICS (GE 645).

The basic disadvantage of a segmented name space over a linear name space is the added complexity of the addressing mechanism needed when the actual physical storage is in fact addressed linearly. The complexity is not too detrimental in itself, but it can possibly cause a significant increase in the time taken to address storage. However this increase can be considerably reduced by the use of sophisticated hardware mechanisms (as in the Burroughs B8500 system, Appendix A.4).

Most of the advantages claimed for segmentation derive from the fact that the segment represents a convenient high level notation for creating a meaningful structuring of the information used by a program. This holds whether the segmenting is specified by a programmer or automatically by a compiler. Thus the advantages include:

(i) The information conveyed by the fact of segmentation can be used by the system in making decisions as to the allocation of storage space and the movement of information between levels of a storage hierarchy.

(ii) Segments form a very convenient unit for purposes of information protection and sharing, between programs.

(iii) The checking of illegal subscripting can be performed automatically. Each array used by a program can be specified to be a separate segment in order that attempted violations of the array bounds can be intercepted.

(iv) The use of a segmented name space alleviates the programmer's task of name allocation. For example, separate segments can be used for each set of items whose size is going to vary dynamically.

It is possible to further classify segmented name spaces in accordance with the characteristics of the space provided for the names of segments, i.e. the first term of the ordered pair (name of segment, name of item within seg-

ment). In particular one can distinguish the alternatives of a linear or a symbolic name space, for segment names. The terms *linearly segmented name space*, and *symbolically segmented name space* describe these two cases. It should be noted that the distinction between linear and segmented name spaces is based purely on the method used to specify the information which is to be accessed—it is independent of any underlying storage allocation mechanism.

The difference between linearly segmented and symbolically segmented name spaces is somewhat more subtle. The basic difference is that in the latter the segments are in no sense ordered, since users are not provided with any means of manipulating a segment name to produce another name. This lack of ordering means that there is no name contiguity to cause the sort of problems that are present in the task of allocating and reallocating addresses (see sections on *Uniformity of Unit of Storage Allocation* and *Storage Allocation Strategies*). Thus one does not need to search a dictionary for a group of available contiguous segment names, and more importantly, one does not have to reallocate names when the dictionary has become fragmented or, if dynamic name reallocation is not possible, tolerate the fragmentation. A symbolically segmented name space consequently involves far less bookkeeping than a linearly segmented name space.

A possible advantage which would be claimed on behalf of linearly segmented name spaces is that they permit indexing across segment names. This can be of utility if the maximum permitted extent of segments is less than the full address representation. This is the case for example when the segment name, item name pair have been compressed into the standard name representation. However if segments are unordered there is little point in usurping part of the address representation for segment names, and in fact there is no a priori need to limit the maximum extent of segments to less than that permitted by the full address representation.

The Burroughs B5000 is an example of a system which provides a symbolically segmented name space, whereas the IBM 360/67 provides a linearly segmented name space. The MULTICS system falls somewhat between these two, since by convention programmers are dissuaded from manipulating segment names, although the segment name space is actually linear. In fact in both the IBM 360/67 and the MULTICS systems a sequence of bits at the most significant end of the address representation is considered to be the segment name. In the B5000 the segment name is part of an instruction and cannot be manipulated.

Predictive Information. A second basic characteristic of dynamic storage allocation systems relates to the inclusion in programs of directives predicting the probable uses of storage over the next short time interval. A system may or may not permit such predictions, which is not the same as having the programs incorporate an explicit storage allocation strategy. The consequences of predictions will be related to the overall situation as regards storage utilization, and the directives are essentially advisory. The source of predictive information can be either the programmer or compiler.

It should be noted that the possible use of predictive information must be carefully distinguished from segmentation. Confusion sometimes arises because segments merely by their existence implicitly contain system exploitable information about future use of storage. For example, if the program has started using information from a particular segment, it is likely, in a short time, to need to use other information in that segment. In fact, segmentation *does* provide a convenient unit of prediction, although there are systems which, while not providing a segmented name space, do allow explicit predictive information, as for example, the IBM M44/44X system (Appendix A.2).

The authors' opinion is that the general level of performance of the system should not be dependent on the extent and accuracy of predictive information supplied by users. The system should in general achieve acceptable performance without such user-supplied information. Provision and debugging of predictive information should be regarded as an attempt to "tune" the system for special cases. The situation is different when the information is provided by a compiler, but *only* if it is known that all programs written for the computer system will use such compilers. (This can be achieved by legislation, or by the use of an authoritarian operating system.)

Pioneering work on the concepts of segmentation and the use of predictive information to control storage allocation was done in connection with Project ACSI-MATIC [10]. In this system programs were accompanied by "program descriptions," which could be varied dynamically, and which specified, for example, (i) which storage medium a particular segment was to be in when it was used, and (ii) permissions and restrictions on the overlaying of groups of segments. Storage allocations strategies were then based on the analysis of these descriptions.

Artificial Contiguity. In earlier sections care has been taken to distinguish between the name used by a program to specify a particular informational item and the address used by the computer system to access the location in which the item is stored. This has been done despite the fact that on many computers, particularly the early ones, such names and addresses appeared to be identical. This distinction is particularly important in discussing the subject of artificial contiguity.

The subject of address contiguity, or more properly, name contiguity, was introduced in the section on *Storage Addressing*. Until recently, name contiguity necessitated an underlying address contiguity. However in several recent computer systems, mechanisms (usually part-hardware, part-software) have been provided to give name contiguity without the necessity for a complete address

contiguity. This is done by providing a mapping function in the path between the specification of a name by a program and the accessing by absolute address of the corresponding location. The mapping is usually based on the use of a group of the most significant bits of the name. A set of separate blocks of locations, whose absolute addresses are contiguous, can then be made to correspond to a single set of contiguous names, as shown in Figures 1 and 2. The first example of such a system was the Ferranti ATLAS computer.

This third characteristic of storage allocation systems need not be apparent to the users of the system, since its only immediate effect is on the inner workings of a system. However the fact that a mechanism is available for providing artificial contiguity is almost invariably used for disguising the actual extent of physical working storage. This point is sometimes stressed by calling such systems "virtual storage systems." Thus in the IBM M44/44X the extent of the linear name space which is provided for each user is approximately two million words, ten times the actual extent of physical working storage. Similarly, in the MULTICS system each segment can be larger than actual physical working storage.

Such systems can be contrasted with, for example the Burroughs B5000, in which the maximum size of segment is 1024 words, although a typical size for working storage is 24,000 words. This limitation is reflected, for example, in the fact that the maximum size vector that an ALGOL programmer can declare is 1024 words. However by virtue of the way the compiler implements multidimensional arrays, the programmer can declare, for instance a 1024 × 1024 word matrix. In other words, the limitation is on contiguous naming and not on apparently accessible information.

There are many cases when it is desirable to use address arithmetic to specify access to a large set of informational items. Thus the large linear name space that is provided using artificial contiguity can be very useful. However it is much more convenient to have a set of such large linear name spaces as separate segments, rather than having just a single large linear name space. This is because, in the latter case, groups of information items, which could well have been placed in separate segments, will have to coexist in a single name space. Therefore problems of name allocation which need not have concerned the user will remain to be solved. In fact even if the segmented name space has severe limitations on the size of segments, the convenience of use will often outweigh the advantage of being able to index over a single large linear name space. This point is reinforced by the fact that if a large linear name space is sparsely used the computing system has very little information on which to base rational judgments as to appropriate storage allocation strategies.

As regards the actual mapping mechanism which provides artificial contiguity, its main disadvantages are likely to be cost and reduced speed of addressing. However it can be of considerable use in aiding the underlying implementation of the storage allocation system, no matter what type of name space is provided for the user.

Uniformity of Unit of Storage Allocation. All dynamic storage allocation systems use blocks of contiguous working storage locations as their units of allocation. The size of such blocks can be either apparent to the user or hidden by an address mapping system. The final classification of storage allocation strategies relates to the question of whether or not these blocks of contiguous storage locations are all of the same size.

If the size of the unit of allocation is varied in order to suit the needs of the information to be stored, the problem of storage fragmentation becomes directly apparent. In other words, the storage space available for further allocation becomes fragmented into numerous little sets of contiguous locations. The two main alternative courses of action in such circumstances are either (i) to accept the decreased storage utilization, or (ii) to move information around in storage so as to remove any unused spaces between the sets of contiguous locations. When the average allocation request involves an amount of storage that is quite small compared with the extent of physical storage, the former course is often quite reasonable, since analysis or

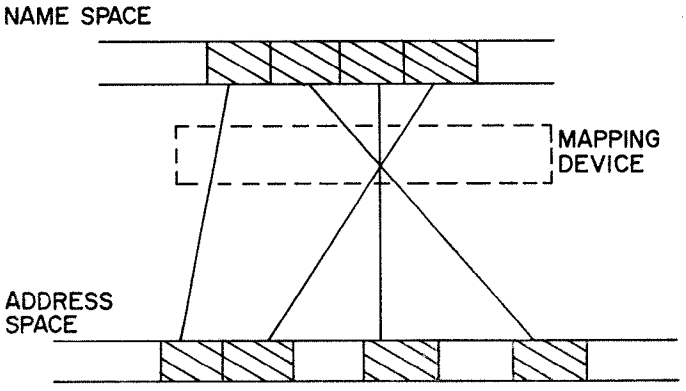

FIG. 1. Artificial name contiguity

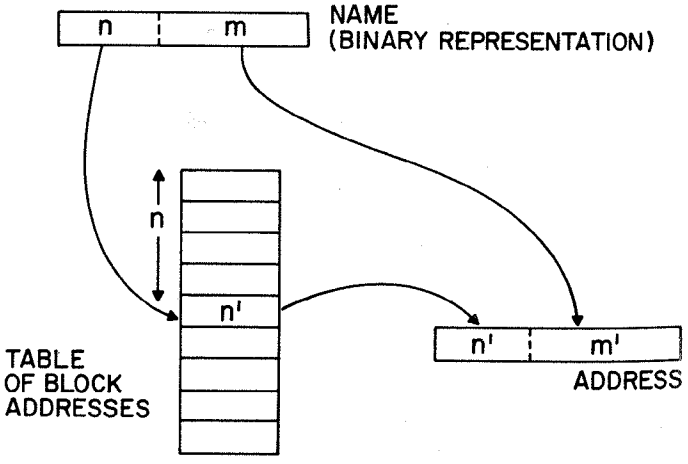

FIG. 2. A simple mapping scheme

experimentation can often be used to show that the storage utilization will remain at an acceptable level (see for example Wald [18]). The other course of action has led to the development of sophisticated strategies for minimizing both fragmentation and the corrective data movement (see section on *Placement Strategies*).

On the other hand, storage can be allocated in blocks of equal size, which we call "page frames," a "page" being the set of informational items that can fit within a page frame. Systems in which this is the case and which use a mapping device to make the addresses of items in pages independent of the particular page frame in which the page currently resides are often referred to as "paging systems."

One of the great virtues of such systems is their simplicity, since a page can be placed in any available page frame. However, it is sometimes claimed that a further advantage of a paging system is that it entirely eliminates the problem of storage fragmentation. Rather, what is true is that paging just obscures the problem, since the fragmentation occurs within pages. It is only rarely that an allocation request will correspond exactly to the capacity of an integral number of page frames, and many page frames will be only partly used. This will be the case whether the allocation request is made explicitly or implicitly by virtue of the way that the user has allocated names in a linear name space. In fact one of the problems of designing a system based on a uniform unit of allocation is choosing the size of the unit. If it is too small, there will be an unacceptable amount of overhead. If it is too large, too much space will be wasted. However a paging system, if properly used, can be very effective. The difficulty is that if this is not the case, and the utilization of space within pages is found to be unacceptably low; program recoding and data reorganization will probably be necessary in order to improve the situation.

The earliest paging system was the Ferranti ATLAS which had 512-word pages. Several later systems, though still commonly referred to as paging systems, do not have a uniform unit of allocation. An example is MULTICS, which has two page frame sizes—1024 and 64 words. Thus, at least in theory, this system has to contain provisions for dealing with the storage fragmentation problem. At the other end of the scale the unit of allocation in the Burroughs B5000 directly reflects the allocation request.

Basic Characteristics—Summary. The above discussions have been intended to show that each of the four basic characteristics is of considerable utility in describing a storage allocation system, and that collectively they have the advantage of being, to a large degree, mutually independent. They draw attention to the fact that, despite the many varied types of storage allocation systems in existence, not all of the more promising choices of a set of characteristics have been tried. In fact in the above discussions it may have been apparent that the authors tend to favor, from the point of view of user convenience and system efficiency, such a choice, namely:

(i) a symbolically segmented name space;

(ii) provisions for accepting predictions about future use of segments;

(iii) artificial contiguity used if it is essential, to provide large segments, but with use of the mapping device avoided in accessing small segments; and

(iv) nonuniform units of allocation, corresponding closely to the size of small segments, but with large segments if allowed, allocated using a set of separate blocks.

Storage Allocation Strategies

The selection of a particular combination of the four basic characteristics of storage allocation systems provides a preliminary system specification. No detailed specification of a storage allocation system would however be complete without a description of the basic strategies it incorporates. These strategies, which for example control where information is to be placed in storage, when information is to be fetched, etc., although closely integrated, are most easily described as if they were each separately concerned with a different problem area. Three such problem areas are identified and discussed below.

It cannot be stressed too strongly that the strategies of storage allocation must be fully integrated with the overall strategies for allocating and scheduling the computer system resources. For example, a system in which entirely independent decisions are taken as to processor scheduling and storage allocation is unlikely to perform acceptably in any but the most undemanding of environments. Detailed discussions of the interactions between the storage allocation system and the operating system are however beyond the scope of this paper. Similarly, the additional problems associated with the allocation of storage to the various parts of the storage allocation system itself, though considerable, will not be covered.

The choice of suitable strategies will depend highly upon the environment in which they are to be used and in particular the characteristics of the various storage levels and their interconnections. A simple illustration of this is given below in the discussion of fetch strategies.

Fetch Strategies. There exist many strategies governing when to fetch information that is required by a program. For instance, information can be fetched before it is needed, at the moment it is needed (e.g. "demand paging"), or even later at the convenience of the system. The latter two cases make most sense when there is something else to be done while awaiting arrival of the information.

Demand paging uses the address mapping device to detect reference to a page which is not currently in one of the page frames. A page fetch will then be initiated. Demand paging thus tends to minimize the amount of working storage allocated to each program, since only pages which are referenced are loaded. However a more significant measure of a strategy's effectiveness is the

space-time product. A program which is awaiting arrival of a further page will, unless extra page transmission is introduced, continue to occupy working storage. Thus the space-time product will be affected by the time taken to fetch pages, which will depend on the performance of the storage medium on which pages that cannot be held in working storage are kept. If page fetching is a slow process, a large part of the space-time product for a program may well be due to space occupied while the program is inactive awaiting further pages. This is represented graphically in Figure 3.

A large space-time product will not overly affect the performance (as opposed to utilization) of a system if the time spent on fetching pages can normally be overlapped with the execution of other programs. This will certainly be the case when there is sufficient working storage space for each program so that further pages are not demanded too frequently, and so that a sufficient reserve of programs can be kept in working storage ready for execution. Demand paging however can be quite effective, without requiring an excessive amount of working storage, when the time taken to fetch a page is very small.

An additional complexity in fetch strategies arises when there are several levels of working storage, all directly accessible to the processor. In such circumstances there is the problem of whether a given item should be fetched to a higher storage level, since this will be worthwhile only if the item is going to be used frequently.

Placement Strategies. Once it is decided that some information is to be fetched, then some strategy is needed for deciding where to put the information, assuming that a choice of available spaces exists. The question arises only for systems which have a nonuniform unit of storage allocation. On such systems, careful placement can considerably reduce storage fragmentation. A common and frequently satisfactory strategy is to place the information in the smallest space which is sufficient to contain it. An alternative strategy, which involves less bookkeeping, is to place large blocks of information starting at one end of storage and small blocks starting at the other end. A further alternative is given in Appendix A.4.

The choice of a placement strategy should be influenced by several factors. These include the relative importance of minimizing storage fragmentation, the frequency of storage allocation requests, the average size of allocation unit, and the number of different allocation units.

Replacement Strategies. When it is necessary to make room in working storage for some new information, a replacement strategy is used to determine which informational units should be overlayed. The strategy should seek to avoid the overlaying of information which may be required again in the near future. Program and information structure, conveyed perhaps by segmenting, or recent history of usage of information may guide the allocator toward this ideal.

A detailed evaluation of several replacement strategies for the case of uniform units of allocation has been given by Belady [1]. The case of variable units of allocation is in general more complex because of the additional possibility of moving information within working storage in order to compact vacant spaces.

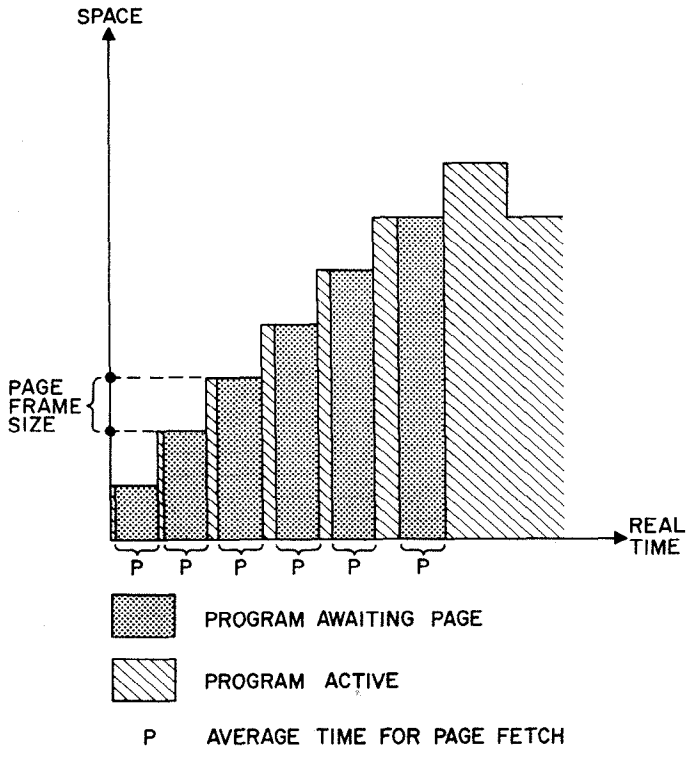

FIG. 3. Storage utilization with demand paging

Special Hardware Facilities

In order to facilitate the task of storage allocation, various special hardware facilities have been incorporated in several recent computer systems. A list of some of the functions performed in part or in whole by these facilities is given below.

(i) *Address mapping.* For example, indirect addressing through a special mapping memory, or through an associative memory, is used to give some measure of independence between the names of informational items and the addresses of their storage locations and can be used to provide artificial name contiguity.

(ii) *Address bound violation detection.* The automatic checking of an address against base and limit values as it is being used to access information can be used for example to ensure that a name falls within the extent of a segment.

(iii) *Storage packing.* The need to speed up the process of storage packing to reduce fragmentation is sometimes catered for by fast autonomous storage to storage channel operations.

(iv) *Information gathering.* Typical examples of special hardware for information gathering are sensors which record the fact of usage or of modifications of the information constituting a page or a segment. Such sensors can then

be interrogated in order to guide the actions of a replacement strategy.

(v) *Trapping invalid accesses.* The automatic trapping of attempts to access information not currently in working storage is usually provided as one of the functions of a mapping mechanism. It is at the heart of the demand paging strategy and is also of use in many other situations.

(vi) *Reduction of addressing overhead.* Many computers have special hardware for the purpose of reducing the average time taken to determine the current location of an item of information. The most obvious example of such a device is a small associative memory in which recently-used segment and/or page locations are kept. If it were not for such mechanisms, the cost in extra addressing time caused by the provision of, say, segmentation and artificial name contiguity, would often be unacceptable.

Conclusions

In conclusion, it is sufficient to reiterate various points.

(i) Storage allocation strategies must be fully integrated with the overall strategies for allocating and scheduling the use of computer system resources.

(ii) The choice of a suitable storage allocation system is strongly dependent on the characteristics of the various storage levels, and their interconnections, provided by the computer system on which it is implemented.

(iii) The four basic characteristics of storage allocation systems identified in section on *Basic Characteristics of Dynamic Storage Allocation System* (name space provided, acceptance of predictive information, artificial contiguity, and uniformity of unit of allocation) are largely independent.

(iv) Two rather different types of segmented name space, namely symbolically and linearly segmented name spaces, are in common use.

(v) Storage fragmentation is not prevented, but just obscured, by paging techniques. In fact such techniques are of no assistance in handling the problem of fragmentation within pages.

(vi) An examination of the "space-time" product for a program illustrates the dangers of demand paging in unsuitable environments.

Acknowledgments. The writing of this paper has been aided considerably by the numerous discussions the authors have had with various colleagues and, in particular, L. A. Belady, M. Lehman, H. P. Schlaeppi, and F. W. Zurcher. Thanks are also due to Mrs. J. Galto for her iterative preparation of the manuscript. However, responsibility for errors or inadequacies and for the views expressed must remain with the authors.

APPENDIX. Specific Computer Systems

This brief survey of relevant aspects of several computer systems is intended to illustrate the many combinations of functional capability, underlying strategies, and special hardware facilities that have been chosen by system designers.

A.1. Ferranti ATLAS

The Ferranti ATLAS computer [8, 14, 15] was the first to incorporate mapping mechanisms which allowed a heterogeneous physical storage system to be accessed using a large linear address space. The physical storage consisted of 16,384 words of core storage and a 98,304 word drum, while the programmer could use a full 24-bit address representation.

This was also the first use of demand paging as a fetch strategy, storage being allocated in units of 512 words. The replacement strategy, which is used to ensure that one page frame is kept vacant, ready for the next page demand, is based on a "learning program." The learning program makes use of information which records the length of time since the page in each page frame has been accessed and the previous duration of inactivity for that page. It attempts to find a page which appears to be no longer in use. If all the pages are in current use it tries to choose the one which, if the recent pattern of use is maintained, will be the last to be required. This replacement strategy and its performance were originally described by Kilburn et al. [14] and are also discussed by Belady [1].

The limited amount of core storage on Atlas makes full multiprogramming infeasible. In fact paging is presented as a technique for storage management within the confines of a single program. However, core storage is partitioned dynamically so that the execution of one program can proceed while output from previously executed programs and input for programs awaiting execution are being performed. Thus at least some of the time spent awaiting the arrival of pages can be overlapped.

A.2. IBM M44/44X

The IBM M44/44X is an experimental computer system designed and installed at the Thomas J. Watson Research Center [20]. The basic hardware of this system (called the M44) is a 7044 computer, which has been extensively modified, in particular by the addition of approximately 200,000 words of directly addressable 8 microsecond core memory. Each of the online users, communicating with the system by means of a terminal, is given the impression that he is using a separate computer (called a 44X) with a 2 million word linear name space, a full instruction complement, and a skeleton operating system. These "virtual machines" are in fact provided by a Modular Operating System (MOS) running on the M44.

The 8 microsecond core store is used as working storage for 44X programs, with a 9 million word IBM 1301 disk file being used as backing storage. Storage allocation is performed by MOS, using a demand paging technique. The page size may be varied at system start-up for experimentation purposes. Various replacement algorithms have been used. One of particular interest selects at random from a set of equally acceptable candidates determined on the basis of frequency of usage and whether or not a page has been modified (see Belady [1]).

As a supplement to the normal technique of demand paging in the M44/44X system, it is possible for programs

to convey predictive information about future storage needs. This is done by two special instructions: one indicates that a page will shortly be needed; the other indicates that it will not be needed for some time. However, as yet very little use has been made of these facilities, and thus it is not known how effective they might be.

Address mapping is performed on this system by indirect addressing through a special mapping store. This store is also used to contain information about page usage which is gathered automatically by special hardware.

Demand paging appears particularly effective on this system, despite the large speed differential between core and disk, because of the large amount of real core storage available. This allows a significant portion of each user's programs to remain in core storage during execution of a round robin scheduling algorithm. Thus the number of page transfers is kept within bounds, and those that occur can in general be overlapped by switching the M44 to another 44X program.

A.3 Burroughs B5000

The B5000 [17, 23] was one of the first systems to provide programmers with a segmented name space (in fact a symbolically segmented name space). Segments are dynamic but have a maximum size of 1024 words.

Programs in the B5000 are segmented by compilers at the level of Algol blocks, or Cobol paragraphs. The segment is used directly as the unit of allocation. Each segment is fetched when reference is first made to information in the segment.

Each program in the system has associated with it a Program Reference Table (PRT). This table is allocated as the program's initial segment, and a special register is set to point at the starting address of the table. Every segment of the program is represented by an entry in this table.

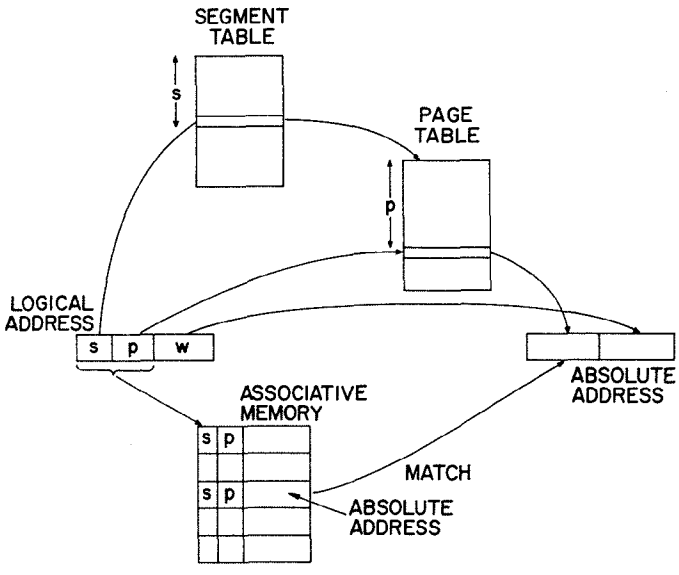

Fig. 4. Two-level mapping scheme

This entry gives the base address and extent of the segment, and an indication of whether the segment is currently in working storage.

Several different placement and replacement strategies have been employed during the development of this system. Among those found to be effective were a placement strategy of choosing the smallest available block of sufficient size and a replacement strategy which was essentially cyclical.

A.4. Rice University Computer

Another early storage allocation system providing a segmented name space was that implemented on the Rice University Computer. The system is the subject of a paper by Iliffe and Jodeit [13] who describe it as being based on the use of "codewords." In fact codewords are used to provide a compact characterization of individual program or data segments, and are thus approximately analogous to the descriptors, or PRT elements, used in the B5000 system. Probably the major difference between codewords and descriptors is that codewords contain an index register address. When the codeword is used to access a segment, the contents of the specified index register are automatically added to the segment base address given in the codewords. The equivalent operation on the B5000 would have to be programmed explicitly.

The basic system described by Iliffe and Jodeit is concerned with storage allocation solely for the working storage, since the only backing storage available on the computer was magnetic tape. However the paper includes proposals for extending the system to deal with a more suitable backing store such as a drum.

The unit of allocation is the segment, which is therefore limited to the size of physical working storage. The fetch strategy is in general to fetch segments when the first attempt is made to access them, though explicit requests to fetch or store segments are permitted.

The placement strategy is as follows. Segments are initially placed sequentially in storage in a block of contiguous locations, the first of which is a "back reference" to the codeword of the segment. When a segment loses its significance the block in which it was stored is designated as "inactive," and its first word set up with the size of the block and the location of the next inactive block in storage. When space is required for a segment, the chain of inactive blocks is searched sequentially for one of sufficient size. If one is found, the requested amount of space is allocated, and if any unused space is left over it replaces the original inactive block in the chain. If an inactive block of sufficient size cannot be found, an attempt is made to make one by finding groups of adjacent inactive blocks which can be combined. If this fails a replacement algorithm, which takes into account whether a copy of a segment exists in backing storage and whether or not a segment has been used since it was last considered for replacement, is applied iteratively until a block of sufficient size is released.

A.5. BURROUGHS B8500

The storage allocation system provided in the B8500 [18] is very similar to that of the B5000. Detailed descriptions are not yet available, but a description of some of the novel hardware facilities of the B8500 have been published [11].

The most notable of these is a 44 word thin film associative memory. This is used for instruction and data fetch lookahead (16 words), temporary storage of program reference table elements and index words (24 words) and a 4 word storage queue. In the B8500 any word in storage can be used as an index register. Recently used registers are automatically retained in this memory together with PRT elements corresponding to recently accessed program and data segments.

A.6. MULTICS

The *Multi*plexed *I*nformation and *C*omputing *S*ervice (MULTICS) [4, 5, 10, 21] is a joint project of Project MAC at MIT, Bell Telephone Laboratories and the General Electric Company. The MULTICS system is to provide computing utility service to a large number of users. The initial system is being implemented on a GE 645 computer. A "small but useful" GE 645 configuration is described [21] as including two processors, 128K words of core storage, 4 million words of drum storage, and 16 million words of disk storage.

As mentioned earlier, the system provides each user with a linearly segmented name space, which by convention is used as a symbolically segmented name space. Segments are dynamic and have a maximum extent of 256K words. Each user can have access to a maximum of 256K different segments.

Unlike the B5000 system, the segment is not the unit of allocation. Instead allocation is performed by a variant of the standard paging technique, since in fact two different page sizes (64 and 1024 words) are used. Thus, at the cost of somewhat added complexity to the placement and replacement strategies, the loss in storage utilization caused by fragmentation occurring within pages can be reduced.

Name contiguity within segments is provided by a mapping mechanism using two levels of indirect addressing, through a segment table and a set of page tables (shown diagramatically in Figure 4). Each entry in the segment table indicates the location of the page table corresponding to that segment. A small associative memory is used to contain the locations of recently accessed pages in order to reduce the overhead caused by the mapping process. This contrasts with the associative memory incorporated in ATLAS, which performs the mapping directly.

The basic fetch strategy is demand paging, but provisions are also made to allow a programmer to specify:

(i) that certain procedures or data be kept permanently in working storage;

(ii) that certain information will be accessed shortly, and should be brought into working storage if possible;

(iii) that certain information will not be accessed again, and may be removed from working storage.

A.7. IBM SYSTEM/360 MODEL 67

The Model 67 [2] is also intended as a large multiple-access computer system, and incorporates several extensions to the standard System/360 architecture. A typical system is described [7] as having two processors, three memory modules, each of 256K 8-bit bytes, a drum capacity of 4 million bytes, and close to 500 million bytes of disk storage.

Two versions are planned, one with a 24-bit address representation, the other with a 32-bit representation. Users are provided with a segmented name space, in which segments have a maximum size of one million bytes. The maximum number of segments is 16 with 24-bit addressing, or 4096 with 32-bit addressing.

The name space is linearly segmented, and is used as such. Moreover the limited number of segments, at least in the 24-bit addressing version, makes it necessary to pack, for example, several independent programs into the same segment. Therefore the segmentation is intended to reduce the number of page table entries that have to be stored and not normally to convey structural information about programs and their data.

The address mapping mechanism, which is of the basic form shown in Figure 4, incorporates an eight word associative memory which is used to contain page table entries corresponding to recently used pages. A ninth associative register is used to speed up the mapping of the instruction counter into an actual physical address. Automatic recording of the fact or of modification of the contents of each page frame is provided.

REFERENCES

1. BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J. 5*, 2 (1966), 78–101.
2. COMFORT, W. T. A computing system design for user service. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 619–628.
3. CORBATO F. J. System requirements for multiple access, time-shared computers. MIT Rep. MAC-TR-3, Cambridge, Mass., 1964.
4. ——, AND VYSSOTSKY, V. A. Introduction and overview of the MULTICS System. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 185–196.
5. DALEY, R. C., AND NEUMANN, P. G. A general-purpose file system for secondary storage. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 213–230.
6. DENNIS, J. B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12*, 4 (1965), 589–602.
7. EVANS, D. C., AND LECLERC, J. Y. Address mapping and the control of access in an interactive computer. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30. Thompson Books, Washington, D. C., pp. 23–30.
8. FOTHERINGHAM, J. Dynamic storage allocation in the ATLAS computer, including an automatic use of a backing store. *Comm. ACM 4*, 10 (Oct. 1961), 435–436.
9. GIBSON, C. T. Time-sharing in the IBM System/360: Model

(*Please turn the page*)

67. Proc. AFIPS 1966 Spring Joint Comput. Conf., Vol. 28. Spartan Books, New York, pp. 61–78.
10. GLASER, E. L., COULEUR, J. F., AND OLIVER, G. A. System design of a computer for time-sharing applications. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 197–202.
11. GLUCK, S. E. Impact of scratchpad memories in design: multifunctional scratchpad memories in the Burroughs B8500. Proc. AFIPS, 1965 Fall Joint Comput. Conf. Vol. 27, Part 1. Spartan Books, New York, pp. 661–666.
12. HOLT, A. W Program organization and record keeping for dynamic storage allocation. *Comm. ACM 4*, 10 (Oct. 1961), 422–431.
13. ILIFFE, J. K., AND JODEIT, J. G. A dynamic storage allocation scheme. *Comput. J. 5*, 3 (1962) 200–209.
14. KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., AND SUMNER, F. H. One-level storage system. *IEEE Trans. EC 11*, 2 (1962) 223–235.
15. ——, HOWARTH, D. J., PAYNE, R. B., AND SUMNER, F. H. The Manchester University ATLAS Operating System Part 1: The Internal Organization. *Comput. J. 4*, 3 (1961) 222–225.
16. LONERGAN, W., AND KING, P. Design of the B5000 system. *Datamation 7*, 5 (1961) 28–32.
17. MACKENZIE, F. B. Automated secondary storage management. *Datamation 11*, 11 (1965) 24–28.
18. MCCULLOUGH, J. D., SPEIERMAN, K. H., AND ZURCHER, F. W. A design for a multiple user multiprocessing system. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 611–617.
19. MCGEE, W. C. On dynamic program relocation. *IBM Syst. J. 4*, 3 (1965) 184–199.
20. O'NEILL, R. W. Experience using a time-sharing multiprogramming system with dynamic address relocation hardware. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30. Thompson Books, Washington, D. C. pp. 611–621.
21. VYSSOTSKY, V. A., CORBATO, F. J., AND GRAHAM, R. M. Structure of the MULTICS supervisor. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 203–212.
22. WALD, B. Utilization of a multiprocessor in command and control. *Proc. IEEE 54*, 12 (1966) 1885–1888.
23. ——. The descriptor—a definition of the B5000 information processing system. Burroughs Corp., Detroit, Mich., 1961.

Virtual Memory, Processes, and Sharing in MULTICS

Robert C. Daley and Jack B. Dennis
Massachusetts Institute of Technology, Cambridge, Massachusetts

Some basic concepts involved in the design of the MULTICS operating system are introduced. MULTICS concepts of processes, address space, and virtual memory are defined and the use of paging and segmentation is explained. The means by which users may share procedures and data is discussed and the mechanism by which symbolic references are dynamically transformed into virtual machine addresses is described in detail.

KEY WORDS AND PHRASES: virtual memory, information sharing, shared procedures, data sharing, dynamic linking, segmentation, paging, multiprogramming, storage management, storage hierarchies, file maintenance
CR CATEGORIES: 3.73, 4.32

Presented at an ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1–4, 1967; revised December, 1967.

This paper is based on notes prepared by J. Dennis for the University of Michigan Summer Conference on Computer and Program Organization, June 1966.

The work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction of this report, in whole or in part, is permitted for any purpose of the United States Government.

Introduction

In MULTICS [1] (*Multi*plexed *I*nformation and *C*omputing *S*ervice), fundamental design decisions were made so the system would effectively serve the computing needs of a large community of users with diverse interests, operating principally from remote terminals. Among the objectives were these three:

(1) To provide the user with a large machine-independent virtual memory, thus placing the responsibility for the management of physical storage with the system software. By this means the user is provided with an address space large enough to eliminate the need for complicated buffering and overlay techniques. Users, therefore, are relieved of the burden of preplanning the transfer of information between storage levels, and user programs become independent of the nature of the various storage devices in the system.

(2) To permit a degree of programming generality not previously practical. This includes the ability of one procedure to use another procedure knowing only its name, and without knowledge of its requirements for storage, or the additional procedures upon which it may in turn call. For example, a user should be able to initiate a computa-