A general statement of the problem of dynamic program relocation is presented as an aid in describing specific relocation principles.

The main purpose of the paper is to review a number of typical methods of meeting the expanding need for dynamic program relocation. Although no attempt is made at evaluation, the methods are discussed in the context of selected computer systems for tutorial concreteness.

On dynamic program relocation

by W. C. McGee

Recent developments in computer technology have suggested the feasibility of the time-shared mode of computer operation—that is, the simultaneous use of a single computer by more than one user, each of whom acts as if he were the exclusive user. While time sharing can often be justified on economic grounds alone, its biggest attraction seems to lie in providing capabilities beyond the reach of conventional operating modes, such as conversational programming, on-line problem solving, and interaction with other on-line users.

Time sharing poses some significant technical problems for the system designer. One of the most difficult, the proper allocation of computer resources, exists in other operating modes as well, but has special importance in time sharing because of the frequency and rapidity with which new resource allocations must be made. To adequately handle resource allocation under these conditions, it has been necessary to develop new hardware and programming techniques. One such hardware technique developed for this purpose is *dynamic program relocation.*

The purpose of this paper is to trace the evolution of dynamic program relocation, and to describe a few specific but characteristic dynamic program relocation techniques. In the interests of brevity, attention will be focused on the subject of dynamic relocation *per se*; the closely related system problems of storage allocation, sharing of procedures, replacement algorithms, and the like, will not be treated.

Introduction to relocation

From the user's viewpoint, a computer facility is a means of carrying out a procedure on a set of input data. In general, such a procedure contains references to elements in the input set, to intermediate results, and to instructions in the procedure itself. However, before the procedure can be executed on a computer, these references must be translated into references to specific parts of the machine. Thus, variables become identified as memory locations, files as tape drives, and the like. The translation is equivalent to allocating certain of the machine's resources to the procedure. In particular, a portion of the computer's main memory —that is, storage from which instructions are executed—must be allocated to hold the instructions, constants, and intermediate results of the procedure. The allocation of main memory is, in principle, no different from the allocation of other resources in the computer. However, because of main memory's central role in the computer process, and because main memory is expensive and hence should be utilized well, the problem of main memory allocation deserves special attention.

The translation of problem references into main memory addresses can be accomplished at three different times:

- When the procedure is prepared as an operable computer program. The result is an *absolute* program which is, in effect, assigned the same resources each time it is run.
- When the program is loaded. This is known as *static relocation.*
- During program execution. This is called *dynamic relocation.*

absolute programs

The translation of references into main memory addresses is easily accomplished during program preparation, but has a significant drawback: it makes it difficult to allocate memory concurrently to two or more independently written programs. A necessity for a joint allocation arises, for example, whenever a main program and several previously written subroutines are to be combined into a single program; the necessity also arises whenever independent programs are to be executed together in order to make better use of certain computer facilities, i.e., in multiprogramming.

In general, the allocation of memory to independently written programs can be achieved in two ways: (1) by assembling the programs together, or (2) by assembling the programs separately, taking special steps to provide interprogram communication and to prevent conflicting memory allocation. The first method requires some care to avoid use of the same label for different objects, but its biggest drawback is that it requires a separate assembly for each different combination of programs. The second method requires that the particular combination of programs to be used and the storage requirements of each program be known, information not always available when the programs are written. Thus, neither (1) nor (2) provides a satisfactory method of allocating memory to separate programs.

static relocation

Methods of static relocation were first introduced to overcome the difficulty of allocating memory concurrently to independently written programs. In static methods, references to main memory locations are left untranslated by the assembler/compiler and are translated into actual addresses only when the program is loaded for execution. The translation is carried out by the loader, often called a *relocating loader*.

Because memory allocation is performed by the loader, the particular combination of programs to be loaded together can be decided just prior to loading. Further, each program can be assembled independently of the others, provided only that suitable conventions are adopted for interprogram communication.

Implicit in static relocation is the assumption that the allocation of memory to a given program will remain fixed for the duration of the program execution. This is not always economical. For example, when a computer is being time-shared among a number of on-line users, control is typically passed from one program to the next in round-robin fashion, in order to provide each user with adequate response to his requests. In general, then, control will be taken away from a program before its completion. When the time comes to resume the interrupted program, the space previously occupied by the program may be occupied by some other program. Rather than displace the latter, it may be preferable to allocate a new part of memory to the interrupted program. Thus, the allocation of memory to a program may change many times in the course of the program's execution.

Under the following conditions, it is possible with a relocating loader to allocate memory to a program each time it is resumed.

- *Condition* 1: The program is separable into a data part and a procedure part; the procedure part is "pure procedure" (i.e., never modified during execution).
- *Condition* 2: The data part, including the contents of registers at the time of interrupt, contains no absolute memory addresses.
- *Condition* 3: When the program is interrupted, the data part must be dumped into auxiliary storage.

If these conditions are met, memory can be reallocated to an interrupted program by loading, with a relocating loader, the original copy of the procedure part of the program and the latest copy of the data part. Some time can be saved if the data part is further partitioned into read-only and read-write parts. Then only the read-write part need be dumped.

Conditions 1 and 3 are not too restrictive. In contemporary computer design, pure procedures are becoming more prevalent, and the dumping of an interrupted program is to be expected in any case. Condition 2, however, is harder to achieve. Among other things, it limits the use of coding techniques in which addresses are "computed" (e.g., key-to-address transformations), since the computed addresses must generally be stored, either in memory or registers, before they are used.

The preceding conventions may be relaxed somewhat by adopting the convention that a program will be interrupted only at certain "breakpoints" dictated by the program itself. For example, if interrupts are blocked during the execution of subroutines, and if subroutines are written so that they always fetch their parameters from, and store their results in, external locations, then subroutines have the attributes of pure procedures, even though they store intermediate results internally during execution. As pure procedures, they can be relocated simply by copying their original versions.

Nevertheless, the relocation of an interrupted program by methods of static relocation has the significant drawback that it requires the execution of a fairly elaborate procedure each time the interrupted program is resumed. If interruptions occur at a high rate, the time required for resumptions may become excessive.

dynamic relocation

Dynamic relocation methods evolve out of attempts to find a better solution to the problem of reallocating memory to uncompleted programs. In dynamic relocation, the translation of problem references to main memory addresses is delayed until the last possible moment, i.e., until access to memory is required in running the program. Because the program contains no absolute memory addresses, it is invariant to the actual memory allocation it receives; it can be interrupted at any point and reloaded into a different set of memory locations without modification.

While it solves the problem of reallocation, dynamic relocation requires special facilities for efficient implementation. Further, it may have significant implications for the instruction format, since instructions will, in general, hold untranslated addresses in a form appropriate to the relocation technique. For these reasons, the requirements for dynamic relocation should be considered early in the development of a computer.

storage protection

When a number of independent programs share main memory, it is typically required that programs not be allowed to access the memory allocated to other programs. In other words, each program's memory must be protected from the other programs. Frequently, the techniques used to achieve dynamic relocation can also be used to effect storage protection. For this reason, it is expedient to treat the two problems jointly.

Statement of the problem

Before considering some solutions to dynamic relocation, a more general statement of the problem may be helpful. Such a statement is best approached by first postulating the general structure of a program to which main memory is to be allocated. For this purpose, we draw upon definitions given by Dennis and Glaser.[1]

A program is composed of one or more *segments*. Formally, a segment is an ordered set of computer words in which each word is identified by a unique *word number* i. We will generally assume that the words of a segment are numbered sequentially, starting from 0. The number of words in a segment is called *segment length.*

The segment itself is given a *segment name* s. Except that each segment of a program must have a different name, the form of the segment name is not critical for our purposes.

By means of segment name and word number, one can refer unambiguously to any word in any segment. In particular, a reference to word i of segment s will have the form of the couple (s, i).

From the programmer's point of view, a segment is any portion of a program which may be written and compiled or assembled more or less independently of the rest of the program. A segment may consist entirely of instructions, entirely of data, or of both instructions and data. Examples of segments are: main programs, subroutines, lists of channel commands, and data arrays.

Before a program can be used, a correspondence must be established between the words of the program and certain locations in main memory. This correspondence is used to load the program into main memory and to translate program references into corresponding memory location values. In most computers, memory locations are identified by a single sequence of integers $y = 0, 1, \cdots$. Thus a correspondence must be established between a set of program words with identifiers of the form (s, i) and a set of memory locations with identifiers of the form y.

Given such a correspondence, it is the function of a dynamic relocation technique to translate each program reference encountered during program execution into a corresponding memory location value. Program references requiring translation may come from a number of sources. The principal source, of course, is the instruction sequence. Other sources include the instruction counter, which holds the identifier of the instruction currently being executed, and the input/output channels which refer, independently of the instruction sequence, to parts of the program being respecified by an input operation or copied by an output operation. If any of these sources is accessible to a program, the reference should be given to the program only in its untranslated form, thus avoiding any possible dependence on the current memory allocation. For example, a "link branch" type of instruction should store the "return address" in the form (s, i) rather than as the corresponding location value y, since the latter may change in the event the program is relocated. Sources which are not normally accessible to a program (e.g., channels, which are usually managed by the supervisor) may, if more convenient, maintain references in their absolute form.

To simplify the translation process, a number of dynamic relocation techniques assume that the various segments of a program have been bound into a single segment prior to execu-

The first type permits full access to the segment in question. The second type prohibits writing into a segment, and is typically used to allow two or more programs to share a common segment without the risk of inadvertent destruction. The third type, which allows writing but prohibits reading, can be useful in some program-checkout situations. The last type, which prohibits access, is generally used to provide complete isolation of independent programs.[2]

In summary, given a number of programs (each of one or more segments that have been partitioned into pages and stored in main memory), the function of a dynamic relocation technique may be stated as follows: For each reference (s, i) by program r to the ith word of segment s, determine if the reference is legitimate and, if so, translate the reference to the memory location of the referent. If the reference is not legitimate, signal a "protection exception."

Examples of dynamic relocation

In this section, we describe some existing and proposed dynamic relocation techniques, using the concepts and terminology developed in the preceding section. The descriptions will emphasize the logical and operational characteristics of the techniques, rather than their specific implementations. No attempt is made to assess the impact of these techniques on the performance of the computer in question, or to evaluate the techniques relative to one another.

case 1

One of the earliest examples of dynamic relocation is found in the optional "multiprogramming package" for the IBM 7090,[3] so named because it provides dynamic relocation and storage protection, both of which are useful in a multiprogramming mode of operation.

By our previous definitions, the 7090 dynamic relocation technique can only accommodate programs of a single segment or, equivalently, programs bound into a single segment. A program thus consists of a single ordered set of 36-bit words, numbered consecutively beginning at 0. The length of the program is limited by the available amount of main memory, not by the relocation technique.

In allocating main memory to a program, no provision is made for partitioning the program into pages. Instead, consecutive words of the program are simply loaded into consecutive memory locations. The correspondence between program word i and memory location y is given by

$$y = \beta + (i - \alpha),$$

where β is the origin of the program, that is, the lowest-numbered location allocated to the program, and α is the word number of the word stored at β. The word number α is usually chosen to be zero.

The translation of program references is effected through a

7-bit *relocation register* which holds the integer $\delta = (\beta - \alpha)/256$. The selection of an origin β is limited by the requirement that δ be an integer. A 15-bit reference i—which may come either from an instruction or the instruction counter—is translated to a 15-bit memory location y by adding the contents of the relocation register to the seven high-order bits of i (and ignoring any carry). Thus,

$$y = i + (\delta \times 2^8) = i + \left(\frac{\beta - \alpha}{2^8} \times 2^8\right) = \beta + (i - \alpha).$$

For example, if word 0 of a program is stored at location 2048, that is, $\alpha = 0$ and $\beta = 2048$, the relocation register would be loaded with

$$\delta = \frac{2048 - 0}{256} = 8.$$

In this case, a reference to word 10 would be translated to location value

$$y = 10 + (8 \times 256) = 2058.$$

The multiprogramming package provides storage protection through two 7-bit registers holding a lower bound x and an upper bound z. A translated location value y is considered valid only if $2^8 x \leq y \leq 2^8 z$.

If m programs are sharing main memory, the supervisor program must arrange for the time-sharing of the relocation register and bounds registers among the various programs. The effect of the storage-protect feature is to give each program read-write access to any word within itself, and no access to any word outside of itself. The access matrix $[t_{ij}]$ of our general problem statement thus becomes an $m \times m$ matrix in which

$$t_{ij} = \begin{cases} \text{"read-write" for } i = j \\ \text{"neither read nor write" for } i \neq j. \end{cases}$$

case 2

A salient feature of the Ferranti ATLAS computer, as described by Kilburn et al.,[4] is a large "main memory" whose locations appear to be randomly accessible, but which is, in fact, composed of a relatively small random access core store and a large amount of drum storage. References to words in core storage are effected in a manner described below; references to words in drum storage cause an interrupt to occur and a supervisor program to copy the desired word from drum to core storage, after which the reference is completed in a normal manner. A given word in the program may thus appear in many different core locations in the course of program execution. To avoid retranslating the program each time a group of words is placed in core, a dynamic relocation technique is used.

The ATLAS dynamic relocation technique treats a program as a single segment. A program may be of any length up to 2^{20} (approximately one million) 48-bit words. Words are numbered sequentially beginning with 0.

For core-storage allocation purposes, the program is partitioned into pages of 512 words each. The page number and line number of a given word are given by the eleven high-order and nine low-order bits, respectively, of the 20-bit word number.

Core storage is similarly partitioned into blocks of 512 locations each, and programs are allocated to core storage by placing pages in blocks. (In ATLAS terminology, blocks are "pages" and pages are "blocks"; here we follow the terminology of Reference 1.) Associated with each block k ($k = 0, 1, \cdots$) is an 11-bit *page register* which holds the page number $p(k)$ of the page currently assigned to that block. Thus, a core store of 2^{14} words is partitioned into $2^{14}/2^9 = 32$ blocks, and can accommodate up to thirty-two program pages at one time.

A reference to a word of a program is made by means of its 20-bit word number i. The reference is translated by first extracting the page number p and line number l from i, and then comparing p to the page number in each of the page registers. If a match is found in page register k, then k is concatenated with the line number l to form the desired location y. Thus

$$y = (k \times 2^9) + l,$$

where k is such that

$$p = p(k).$$

If no matching page number is found in the page registers, an interrupt occurs and the supervisor reads the required page from drum storage into an available block of core storage, possibly displacing another page, and places the new page number in the block's page register. Translation then proceeds as in the case of matching page numbers.

Although the technique makes no explicit provision for segmented programs, the effects of segmenting can be obtained through the judicious use of page numbers in writing/compiling programs. The range of available page numbers (0 to 2047) makes it feasible to assign sets of pages to different users or uses on a more or less permanent basis: system routines might use pages 0–500, User A pages 501–1500, etc.

case 3 Perhaps the first instance of dynamic relocation in which multiple-segment programs are accommodated is found in the Burroughs B5000 computer. As described by Lonergan and King,[5] a B5000 program may contain as many as 1024 segments. Each segment, composed of 48-bit words, may represent instructions, data, control information, and the like. The words of a segment are numbered serially beginning at 0.

The maximum length of a segment is determined by the facilities provided for referring to words within a segment. Essentially, two methods are provided in the B5000:

I. To refer to words in the control segment (that is, the segment in which control resides), a positive or negative displacement Δi relative to the current instruction word may be specified.

Such references are intended only for locating the next instruction to be executed, not for fetching or storing data. The maximum value of the displacement is 1024.

II. To refer generally to any word in the program, a 10-bit segment name s and a 10-bit word number i may be specified. The segment name is specified in the instruction; in fact, it occupies ten of the twelve bits of the "syllable," the B5000 equivalent of an instruction. The word number, on the other hand, is held in an implicitly designated register, where it must have been placed by a previous instruction.

The 10-bit word number limits to 1024 words the length of segments whose words may be individually addressed in Method II. In view of Method I, however, an instruction segment can be of any length, provided that it is always entered at one of its first 1024 words, and that it nowhere branches forward or backward more than 1024 words.

No provision is made for partitioning segments into pages. Each segment s is assigned a set of sequential memory locations starting at location $\beta(s)$.

For purposes of translating displacement references, the location c of the current instruction word is maintained in a *program counter*. The location of the referent is given simply by $c + \Delta i$, and is generally used to respecify c. In particular, the next instruction word sequence is located by specifying $\Delta i = 1$.

To translate references of the form (s, i), a *program reference table* (PRT) is maintained in main memory for each program r. The table, which contains a maximum of 1024 one-word entries, is stored starting at an arbitrary location $b(r)$. The entries are fully accessible to the program and may be used for a variety of purposes. In particular, any entry may be used to hold a segment *descriptor* which contains, among other things, the location of the first word $\beta(s)$ and the length $\lambda(s)$ of segment s.

Translation is effected by adding the segment name s to the base $b(r)$ of the PRT to locate the descriptor for segment s. The segment base $\beta(s)$ is then added to the word number i (held in the accumulator stack) to give the location of the referent. That is, $y = \beta(s) + i$.

Each time such a translation is made, the word number i is compared to the segment length $\lambda(s)$ in the segment descriptor. If $i \geq \lambda(s)$, a protection exception is signaled.

In most dynamic relocation schemes, once the location of the referent has been obtained and used to access memory, it is discarded. The B5000 instruction set, however, in some cases necessitates retention of y in the stack, so that it may be used directly by subsequent operations. For example, storing is accomplished by the "store" syllable, which itself has no provision for referring to program words; this instruction assumes that an appropriate translated location is already in the stack. Since translated locations are placed in the stack, where they are readily accessible to a program, care must be taken not to use them in such a way as to make the program location-dependent.

case 4 The main memory addressing technique of SYSTEM/360 permits a restricted form of dynamic program relocation without changes to the standard system. The method, which requires the observance of more programming conventions than necessary for the hardware-implemented relocation technique described in Case 5, may be of interest in its own right. The basic idea has already been discussed in the literature.[6]

In SYSTEM/360, references to main memory typically take the form of a triple (d, j, k), where d is a 12-bit "displacement," j is the 4-bit designator of a general register holding a 24-bit "index," and k is the 4-bit designator of a general register holding a 24-bit "base." The memory location to be accessed is determined by computing its effective address from the relation

$$y = d + (R_j) + (R_k),$$

where R_j and R_k are the general registers identified by j and k, respectively, and parentheses denote "content of." The means for performing this summation will be referred to as the *effective-address mechanism.*

This form of addressing is employed to minimize the number of instruction bits required to address main memory. In particular, the maximum memory (2^{24} bytes) can be addressed with only sixteen bits—twelve for the displacement and four for the base designator (assuming the latter have been previously placed in the general registers).

By viewing the role played by the base designator a little differently, the SYSTEM/360 addressing technique can be made to perform the function of dynamic relocation. To see this, let k designate segment s of a program, and the "indexed displacement" $d + (R_j)$ designate a byte within that segment. In SYSTEM/360, our generic term "word" becomes "byte." If we assume that general register k holds the segment base (the location of the first byte of the designated segment), then it is clear that the existing effective-address mechanism will perform the translation required for dynamic relocation:

$$y = \underbrace{d + (R_j)}_{\substack{\text{byte}\\ \text{number } i}} + \underbrace{(R_k)}_{\substack{\text{location of first}\\ \text{byte of segment } s}} .$$

The technique is subject to limitations in the structure of of programs that can be accommodated. First, although segments may in principle be of any length up to the maximum capacity of memory, the byte number in a segment must be expressed in the form $d + (R_j)$. Except for data arrays that can be conveniently addressed by indexing, this restriction will in practice tend to limit segments to the number of bytes addressable by d, viz., 4096. Of course, if a segment exceeds 4096 bytes, it may be broken into segments of a more convenient size.

Second, if a one-one correspondence were established between segments and general registers holding segment bases, the number of segments permitted in a program would be unacceptably low, especially in view of the fact that the general registers are also used for other purposes. It is clear, therefore, that at least some of the general registers need to hold different segment bases at different times. This can be accomplished by allowing programs to establish the base of any segment in any register during the execution of the program. For instance, the program could execute the following sequence (expressed in SYSTEM/360 assembler notation):

$$\begin{array}{ll} \text{SVC} & k_{LB} \\ \text{DC} & H\,'r_1, r_2' \end{array}$$

Here SVC is the supervisor call operation, k_{LB} is a code identifying this particular type of supervisor call, DC is the "define constant" assembler instruction, r_1 designates the general register into which a base is to be loaded, and r_2 designates the general register holding the "name" of a segment. The supervisor would respond by placing the segment base in the designated register and returning control to the program. For each active program, the supervisor would of course keep a list of the registers that contain segment bases, together with the corresponding segment names. Whenever a segment is relocated, the supervisor would scan these lists for a matching segment name and wherever a match is found, substitute the new base for the old base.

The method requires that a segment be stored in consecutive memory locations (i.e., consist of a single page). This shortcoming is offset to some extent by the freedom for selecting any byte location as the segment base. This permits tighter segment packing than if segments were constrained to start at certain fixed locations.

As noted above, the method uses the effective-address mechanism of the SYSTEM/360 to effect the translation required for dynamic relocation. Certain SYSTEM/360 instructions, however, do not use the effective-address mechanism in referring to memory. For example, in the BRANCH ON CONDITION instruction in the RR format (BCR), the 24-bit branch address is obtained directly from a specified general register. Such "computed" addresses may be generated in any manner. In particular, they may be generated by certain instructions that deposit translated addresses in the general registers, e.g. BRANCH AND LINK. In the method under discussion, such addresses cannot in general be used since they make a program dependent upon absolute memory locations; an effective address stored in memory at point A in the program for use at point B may be rendered useless if the program is interrupted and relocated between point A and point B. (By the same reasoning, segment bases deposited by the supervisor in the general register should not be stored away by the program.)

One solution to this problem is to require computed addresses to take the form (s, i) where s is, say, an 8-bit segment name, and i is the 24-bit byte number within the segment. The SYSTEM/360 instructions that generate and use computed addresses can then be replaced by supervisor calls, assuming that the supervisor carries out any necessary translation. For example, BCR r_1, r_2 would be replaced by

SVC k_{BCR}

DC $H\,'r_1, r_2'$

where r_2 now contains both a segment number and a byte number. The supervisor would compute the corresponding memory location and then branch to this location or to the point of call, according as the condition specified by r_1 were satisfied or not.

Finally, this method makes no provision for storage protection, which would presumably be accomplished through the SYSTEM/360 storage-protect feature. This feature requires that portions of a program to be protected in the same manner lie in one or more 2048-byte blocks with fixed (but not necessarily contiguous) locations. Because of this, the ability to designate arbitrary locations as segment bases would be limited to a certain extent. The feature provides three of the four access types defined above, viz., read-write, read only, and neither read nor write.

Read-write protection is achieved by giving each program a 4-bit "key" that matches the 4-bit "lock" on each block to which the program is to have read-write access. The latter two types of access are met by ensuring a mismatch between a program's key and the locks of blocks into which writing is to be inhibited. An additional "fetch protect" bit is added to the lock on each block; if this bit is 0, read-only access results, and if it is 1, access is prohibited. Within the constraints implied by this technique, access types may be assigned independently to each program-segment pair.

case 5

A hardware dynamic relocation technique has been provided in the SYSTEM/360 MODEL 67.[7] This technique will accommodate programs of as many as sixteen segments each with each segment containing a maximum of 2^{20} bytes. The bytes of a segment are numbered sequentially starting from 0.

Program references of the form (s, i) are contained in the 24-bit effective address normally used to address memory. The four high-order bits are interpreted as the segment number s, and the twenty low-order bits as the byte number i within the segment. This same interpretation holds regardless of the source of the effective address, be it the effective-address mechanism, the instruction counter, or whatever. It is clear that in addressing the program from an instruction, the reference (s, i) must be fabricated through the effective address mechanism. A convenient, but by no means unique, way of doing this is to place the integer $s \times 2^{20}$ in the general register (k) normally used for the base, and to use the remaining address components (d and j) to fabricate the word number i. Provided the latter does not exceed 2^{20}, the effective-address mechanism will then generate the reference in the required format.[8]

The technique permits segments to be partitioned into pages of at most 4096 bytes each. The first 4096 bytes of a segment comprise the first page, the next 4096 bytes comprise the next page, and so forth. As a result, the page and line numbers of a byte are given by the eight high-order and twelve low-order bits, respectively, of the 20-bit byte number.

Main memory is similarly partitioned into 4096 blocks each of 4096 consecutive bytes. Blocks are numbered serially from 0, so that the block number of location y is given by the twelve high-order bits of y.

A segment is stored in memory by placing each page in a different block. Pages of less than 4096 lines are stored from the beginning of the block, and any unused portions of the block are ignored. The blocks allotted to a given segment need not be contiguous.

The translation of program references into corresponding memory locations is accomplished with the aid of a *table register* and two types of tables stored in main memory—*segment tables* and *page tables.* A separate segment table is provided for each program; the table contains an entry for each segment of the program. A separate page table is in turn provided for each entry in a segment table.

The table register is loaded by the supervisor prior to relinquishing control to a given program r. In addition to control bits of no concern here, the table register contains

- an *origin* $\alpha_1(r)$
- a *length* $\lambda_1(r)$

In the context of program r, the origin is the location of the first byte of the segment table, and the length is the number of entries in the segment table.

Segment table entries are stored in such a way that the location of an entry can be simply computed from the segment table origin and the corresponding segment number. Each segment table entry contains

- the *origin* $\alpha_2(r, s)$ of the page table for segment s of program r
- the *length* $\lambda_2(r, s)$ of this page table
- a one-bit segment availability flag $\phi(r, s)$

The page table for segment s of the program r contains an entry for each page of this segment. The entries are so stored that the location of an entry can be computed directly from the page table origin and the corresponding page number. Each page table entry contains

- the *block number* $b(r, s, p)$ of the memory block assigned to this page

- a 1-bit page availability flag $f(r, s, p)$

The translation of a program reference (s, i) from program r proceeds as follows:

(1) Segment number s and segment table origin $\alpha_1(r)$ are combined to locate the proper segment table entry.
(2) If $\phi(r, s) = 1$, indicating that this segment is not available, the supervisor is called; otherwise, the translation proceeds.
(3) A page number p and a line number l are derived from byte number i. The page number and the page table origin $\alpha_2(r, s)$ are combined to locate the proper page table entry.
(4) If $f(r, s, p) = 1$, indicating that this page is not available, the supervisor is called; otherwise, the translation proceeds.
(5) The block number $b(r, s, p)$ is concatenated with the line number l to form the memory location of the referent.[9]

Memory protection is provided in two distinct ways. First, the translation implies that a legitimate block number be obtained on every memory reference. These block numbers come from the page table, which is set up by the supervisor to correspond to the current memory allocation. Thus, a program cannot gain access to any block not represented in its segment/page tables. In translating each reference, moreover, if segment or page number exceeds segment table length or page table length, respectively, a protection exception is signaled.

Second, the regular memory protection feature of SYSTEM/360 is available to guard individual blocks of memory as described previously. For example, to allow two programs to access a common segment of, say, a single page, the block number of this page would be entered in the tables for both programs. However, the block would be fitted with a lock different from the key of either program, thus assuring "read-only" access to the block.

Summary comment

In this survey we have limited ourselves to a general statement of the problem of dynamic relocation, and used this statement to describe a number of representative dynamic relocation techniques. This approach was used mainly to simplify the exposition. The problem statement is probably no more "general" than the composite of the techniques presented, and will undoubtedly require amending as experience with dynamic relocation is gained. For example, the two-level program structure may prove so useful in handling unwieldy arrays that it may be desirable to generalize it to n levels. The SYSTEM/360 relocation technique described earlier has an obvious generalization for handling such structures, viz., partition the effective address into n parts, provide n types of tables, etc.

All of the techniques described here share one important feature: they enable the user to work with an abstraction of the real computer which is operationally much simpler than the real computer. For example, the ATLAS main memory appears to the user to consist entirely of word-addressable, random-access storage, whereas in actuality it need consist of such storage only in part. Similarly, the main memory of the B5000 may be viewed as a "two-dimensional" memory whose addresses have two independent components, whereas in fact this memory is achieved with a conventional "one-dimensional" memory and an appropriate addressing technique. Such abstractions are coming to be known as *virtual machines.* The idea behind virtual machines is not new, being the essence of all programming systems which tend to mask the real computer from the programmer. As computer operation becomes more complex, the virtual machine concept will become increasingly important.

CITED REFERENCES AND FOOTNOTES

1. J. B. Dennis and E. L. Glaser, "The structure of on-line information processing systems," *Information System Sciences: Proceedings of the Second Congress,* (D. W. Walker, ed.), Spartan Books 1–11 (1965). Also, J. B. Dennis, "Segmentation and the design of multiprogrammed computer systems," *IEEE International Convention Record,* Part 3, 214–225 (1965).
2. A more general statement of the memory protection problem would recognize different types of access for each program-segment-processor triple, where "processor" denotes the facility requesting access: CPU, I/O channel, etc. This would accommodate schemes that protect memory on CPU accesses, but not on I/O accesses. In this paper, we assume that all I/O accesses are controlled by a supervisor program and are, therefore, irrelevant to "problem program" memory protection.
3. IBM Special Systems Feature Bulletin L22-6641-3, "IBM 7090 Data Processing System Multiprogramming Package," International Business Machines Corporation, White Plains, New York (June 1963).
4. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," *IRE Transactions on Electric Computers,* ECII, 223–235 (1962). Also C. H. Devonald and J. A. Fotheringham, "The ATLAS computer," *Datamation* 7, 5, 23–27 (May 1961).
5. W. Lonergan and P. King, "Design of the B5000 system," *Datamation* 7, No. 5, 28–32 (May 1961).
6. F. J. Corbató, "System requirements for multiple access time-shared computers," MAC-TR-3, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts.
7. This relocation technique is due to G. A. Blaauw and associates. A brief description of the Model 67 appears in "A new system for time-sharing," *IBM Computing Report for the Scientist and Engineer* 1, No. 1, 8–9 (May 1965), IBM Data Processing Division, White Plains, New York.
8. An alternate version of the 360 dynamic relocation technique provides for the generation of 32-bit effective addresses, of which the twelve high-order bits are interpreted as segment number, and the twenty low-order bits again serve as byte number. This version will thus accommodate programs of up to 4096 segments each.
9. In actual implementation, a small associative memory is used to hold the more frequently accessed relocation table entries. Except for the consequent increase in translation speed, the actual procedure is equivalent to that given in the text.