Capability-Based Addressing

R.S. Fabry
University of California

Various addressing schemes making use of segment tables are examined. The inadequacies of these schemes when dealing with shared addresses are explained. These inadequacies are traced to the lack of an efficient absolute address for objects in these systems. The direct use of a capability as an address is shown to overcome these difficulties because it provides the needed absolute address. Implementation of capability-based addressing is discussed. It is predicted that the use of tags to identify capabilities will dominate. A hardware address translation scheme which never requires the modification of the representation of capabilities is suggested. The scheme uses a main memory hash table for obtaining a segment's location in main memory given its unique code. The hash table is avoided for recently accessed segments by means of a set of associative registers. A computer using capability-based addressing may be substantially superior to present systems on the basis of protection, simplicity of programming conventions, and efficient implementation.

Key Words and Phrases: addressing, capabilities, addressing hardware, protection, protection hardware, shared addresses, information sharing, operating systems, computer utility, segmentation, tagged architecture

CR Categories: 4.30, 4.32, 4.34, 6.21

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973, under the title "The Case for Capability-Based Computers."

Author's address: Computer Science Division, Evans Hall, University of California, Berkeley, CA 94720.

Introduction

The idea of a capability which acts like a ticket authorizing the use of some object was developed by Dennis and Van Horn [15] as a generalization of addressing and protection schemes such as the codewords of the Rice computer [28], the descriptors of the Burroughs machines [6, 7], and the segment and page tables in computers such as the GE-645 and IBM 360/67 [1, 14]. Dennis and Van Horn extended the earlier schemes to include not just memory but all systems objects—memory, processes, input/output devices, and so on—and to allow the explicit manipulation of access control by non-system programs. The idea is that a capability is a special kind of address for an object, that these addresses can be created only by the system, and that, in order to use any object, one must address it via one of these addresses.

The use of capabilities as a protection mechanism has been the subject of considerable interest [24, 29, 32, 43]. It is assumed that the reader is familiar with the use of capabilities for protection; a different aspect of capabilities is developed here.

It is argued below that there is an advantage in using capabilities as a basic component of the address of every object (except for objects associated with the processor such as its registers). In order to accomplish this, user programs must be able to store capabilities freely into various permanent user data structures (subject, of course, to some scheme for preserving the integrity of the representation of capabilities). Not all schemes which use capabilities actually allow capabilities to be used as permanent addresses in this way. For example, the original Dennis and Van Horn scheme did not because it insisted that capabilities be stored only in C-lists associated with computations.

Context-Independent Addresses

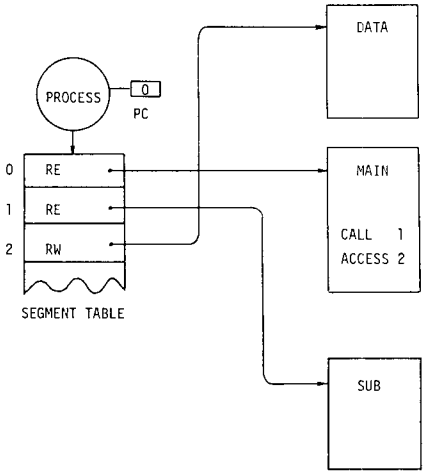
The advantage of a capability used as an address is that its interpretation is context independent. It provides an absolute address for an object. This fact is more important than it may at first appear.

Before the use of address relocation—such as base and limit registers, paging, and segmentation—jobs were allocated fixed areas of physical memory. Addresses within the jobs were relocated at load time, and a job was not moved once it had started running. The lack of the ability to dynamically relocate resulted in underutilized computers. To avoid this underutilization, address relocation was introduced. But in doing so, a new problem was also introduced. Consider two jobs which need to interact with each other. In a system without relocation, jobs share an address space and can be allowed to interact freely, sharing data structures and addresses as easily as if they were a single job. With address relocation, however, the meaning of an address becomes context dependent; each job has its own address space,

or perhaps several. This fact is generally interpreted as an advantage: base and limit registers, paging, and segmentation, by virtue of their address relocation, allow users to be isolated from each other, thus providing protection of one job from another. On the other hand, the sharing of addresses becomes more difficult, and this side effect is generally ignored. This effect is particulaly ironic for those systems which stress their usefulness for cooperating users who want to work together, sharing programs and data.

Although a capability functions as an absolute address, the use of capabilities does not prevent a system from using address relocation. A capability is an absolute address for a virtual object; the system is free to relocate the virtual object so long as it maintains the correspondence between the object and its capabilities.

Fig. 1. Segment addresses.



The Problem of Shared Segment References

Various addressing schemes making use of segment tables have been implemented in present day systems. Although the inadequacies of these schemes when dealing with shared segment addresses may be apparent to users of the systems, the inadequacies have not (with rare exception) been explained in system descriptions, nor does there exist a systematic comparison of the problems which arise with each of the schemes.

As an example of the problems encountered in these systems, consider a particular structure, a set of interacting subprograms. Figure 1 shows a process which has three segments: a data segment, a main program, and a subroutine. The segment table translates integer segment addresses into references to the appropriate segments and specifies the permitted types of access. R means reading is allowed, W means writing is allowed, and E means executing is allowed. The entries in the segment table can be thought of as descriptors, capabilities, codewords, or pointers; for our purposes these are the same. The main program in Figure 1 contains two segment references, a call on the subroutine, coded as CALL 1, and an access of the data segment, coded as ACCESS 2. The PC register associated with the process contains the segment number of the segment which the program counter addresses. In an actual system, the location of a word within a segment is also important. For simplicity, the word number components of addresses are omitted.

Figure 2 shows the case in which the program and the segment references which it contains are shared. Assume for the moment that the correspondence between integers and segments is constructed independently for each process. Then, as in the figure, a segment may be referred to by different integers in different processes. How can the segment references in the shared main program be coded? For process 1, the references should be coded CALL 1 and ACCESS 2. For process 2, however, they should be coded CALL 0 and ACCESS 1. Four different solutions to this problem of shared segment references are presented below.

Fig. 2. Shared segment addresses.

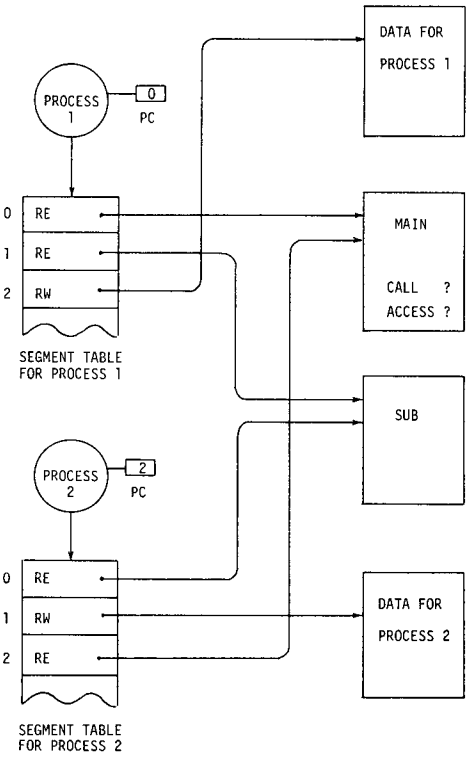


Fig. 3. Uniform address solution.

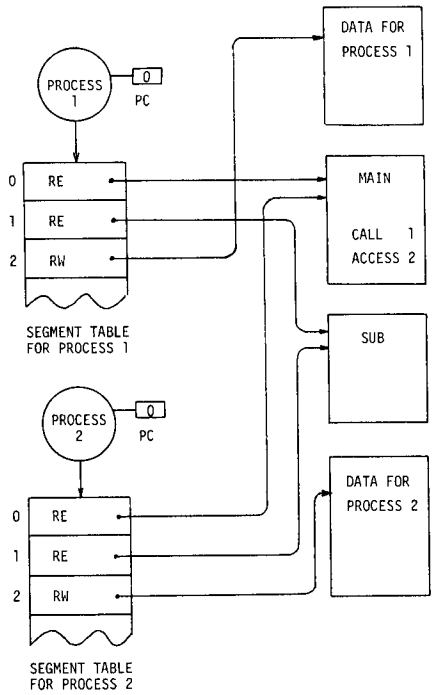
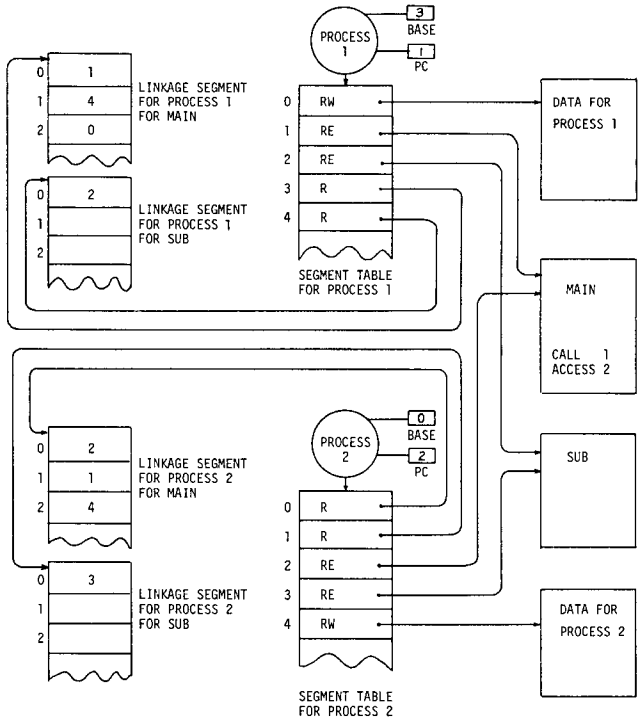


Fig. 4. Indirect evaluation solution.



Uniform Address Solution

The uniform address solution is shown in Figure 3. This solution requires that a shared integer segment address be interpreted in a *functionally equivalent* manner for all processes sharing the address. A shared address is said to receive a functionally equivalent interpretation for a set of processes if the objects referred to by the address are used in the same way by each process. For example, in Figure 3, the segment address 2 refers to the data segment used by the process making the reference, while the segment address 1 refers to the segment containing the subroutine called by the main program in a certain instruction. Note that a functionally equivalent interpretation of a shared address sometimes causes the same object to be referenced by all processes and sometimes causes a different object to be referenced by each process.

The uniform address solution requires that the functions of the various shared integer segment addresses be defined centrally so that there will be no conflicts. This requirement rules out the possibility of a single process executing several independently constructed, shared subprograms. This is ruled out because each independent constructor would be free to choose a function for a particular index and the chosen functions would usually conflict. This is a rather serious drawback if one desires a programming environment in which a user is able to build on the work of others in a general way [13].

Generality notwithstanding, the uniform address solution is used successfully by Burroughs. The Burroughs systems require a user to compile all his program at once (except for certain standard system-wide subroutines). Thus the compiler can allocate the integer segment addresses at compile time and embed them in the code. The Program Reference Table of the B5700 functions exactly as the segment table in Figure 3 does [6, 7, 35].

Indirect Evaluation Solution

The indirect evaluation solution is shown in Figure 4. A shared integer segment address is treated as an index of a position within a linkage segment and the linkage segment contains segment table indexes. One linkage segment per independently-created subprogram per process is assumed; the linkage segment is created the first time a subprogram is executed by a process. (A slightly different scheme can be obtained if a new linkage segment is created each time the subprogram is activated.) Thus, when process 1 executes the code for ACCESS 2, word 2 of the linkage segment for process 1 for the main program is fetched. This word contains 0, which is then taken to be the segment table index of the segment to be accessed, in this case the data segment for process 1. Some processor register must be used to remember the address of the linkage segment. Base registers are indicated for this purpose in Figure 4. It is assumed that both processes are executing the main program, and thus each base register contains the segment table index of the linkage segment for the main program.

Calling independently created subprograms is more complicated with indirect evaluation of segment refer-

ences since the base register contents must be changed. Figure 4 assumes that call instructions contain the address of the linkage segment of the subroutine to be called and that word 0 of this linkage segment contains the segment table index of the segment containing the code for the subprogram. Thus, when process 1 executes CALL 1, Word 1 of the linkage segment for process 1 for the main program is fetched. This word contains 4, which is then placed in the base register. Word 0 of the linkage segment indicated by the new contents of the base register is then fetched. This word contains 2, which is then placed in PC.

The point of linkage segments is to create independently allocated sets of integer segment addresses in order to overcome the main drawback of the uniform address solution. Thus there must be at least one linkage segment per independent allocation of addresses.

When the indirect evaluation solution is used, segment addresses passed from one subprogram to another as parameters are treated differently than addresses embedded in shared programs. Segment table indexes are passed rather than linkage segment indexes; this is because the segment table is process-wide, whereas the linkage segments are not.

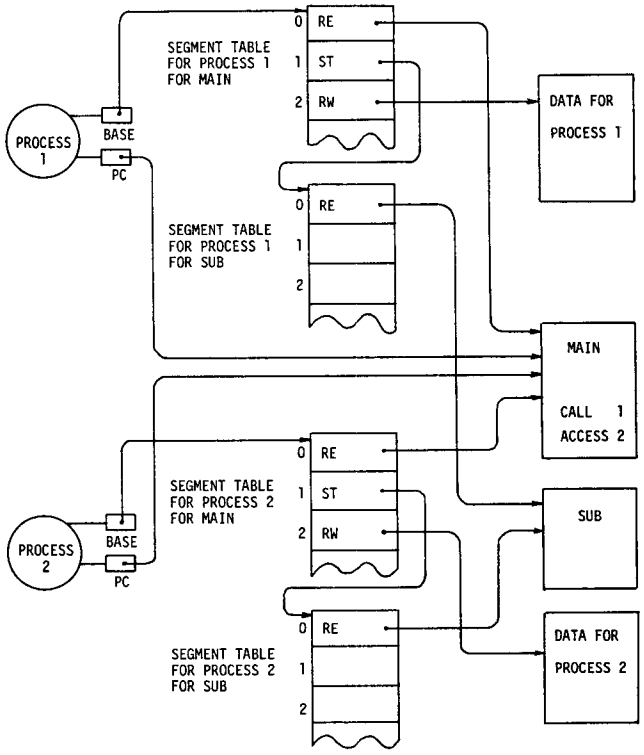
The indirect evaluation solution has several disadvantages. It requires extra space to hold the indirection information, extra overhead to set up the indirection information, and extra memory references to obtain the indirection information. Most important, however, the solution is inadequate. It provides one kind of address space for addresses which are to be used by many programs but one process; it provides another kind of address space for addresses which are to be used by many processes but one program. It makes no provision for an address space for addresses which are to be used by many processes and many programs. Such addresses might be needed, for example, in a multisegment data structure which existed independently of any program or process.

Nevertheless, the indirect evaluation solution has been used successfully for Multics [3, 11, 14, 36]. The actual Multics scheme differs in details from what has been described, but it is the same in concept.

Multiple Segment Table Solution

The multiple segment table solution is shown in Figure 5. This solution can be viewed as a modification of the indirect evaluation solution in which segment table indexes in the linkage segments are replaced by capabilities and the linkage segments are renamed segment tables. The base register and the program counter which contained segment table indexes are modified to contain capabilities also. Thus, when process 1 executes the code for ACCESS 2, the evaluation of the integer segment address works in the same way as for Figure 3; the difference is that the segment table is now private to a particular program as well as to a particular process. Figure 5 assumes that the subroutine instruction con-

Fig. 5. Multiple segment table solution.

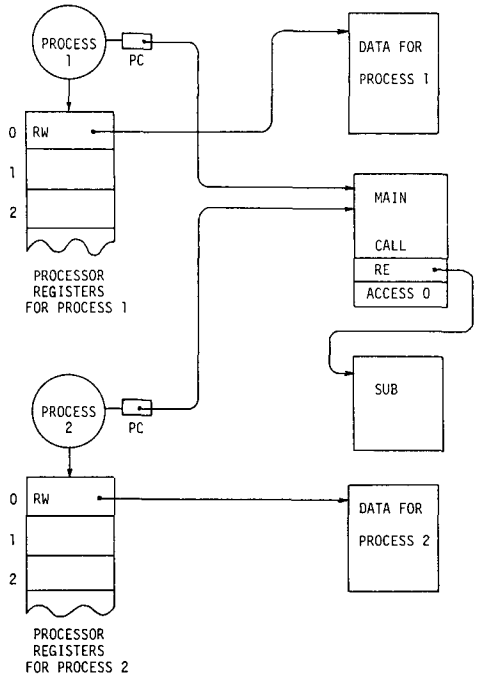


tains the address of the segment table for the called program, and that word 0 in the segment table points to the segment containing the called program. Thus, when process 1 executes the code for CALL 1, word 1 of the segment table for process 1 for the main program is loaded into the base register. Word 0 of this new segment table is then fetched and placed in PC.

The main disadvantage of the multiple segment table solution is that it does away with the per-process segment table and thus with the only addresses which could be shared among several programs being executed by the same process. Thus this solution compounds the problem of shared segment references.

The difference becomes apparent if one considers parameter passing during a subroutine call. For example, in the scheme of Evans and LeClerc [18, 33], which uses a multiple-segment-table-type solution, entries are made in the segment table for a subroutine each time the subroutine is called; these entries are capabilities for the various parameters passed to the subroutine. Such a scheme either disallows recursive subroutines or else requires a new version of the segment table for each level of recursion. Another solution is to store the capabilities for the parameters in a stack, much as is already done on the Burroughs machines. (The Evans and LeClerc scheme also allows a segment table to be associated with a data structure. If one reads into their scheme a mechanism for varying the contents of the segment tables associated with data structures dynamically

Fig. 6. Capability addressing solution.



and under program control, then one would classify it as using a capability addressing solution.)

One can consider the display registers of the B6700 to define segment tables and view the B6700 as using the multiple segment table solution [8, 35].

The protection system suggested by Needham [34] uses multiple segment table addressing in which there are four simultaneously available segment tables: one for capabilities which are global to the process; one for capabilities associated with the current program and shared by all processes using the program; one for capabilities associated with the current program and private to the process; and one for the arguments for this activation of the program. As Needham points out in his paper, there are still difficulties with pointers from one segment to another which appear in some shared data structures.

Capability Addressing Solution

The capability addressing solution [21] is shown in Figure 6. In this scheme capabilities may be used wherever the integer addresses were used previously. In particular, capabilities may be stored in segments and in the registers of the processor. (There must, of course, be some scheme for preserving the integrity of the representation of capabilities. Two schemes are discussed later.) This scheme does away with segment tables and with the mandatory indirect evaluation of shared addresses. In Figure 6, when ACCESS 0 is executed, the 0 means the segment indicated by register 0 and is thus evaluated indirectly. The subroutine call instruction is assumed to be followed by a capability for the segment containing the subroutine to be called. When the CALL is executed, this capability is placed in PC.

In comparing Figure 6 with Figure 3, a distinction is made between processor registers and the segment tables. This distinction is not related to implementation technology but rather to allocation. The allocation of processor registers is under control of the person or compiler generating even the smallest section of code; one is always free to redefine the use of these registers by saving the contents and later restoring them. Thus there is no requirement for a central mechanism to define the use of the registers, and the main problem with the uniform address solution is avoided.

Figure 6 illustrates both types of functionally equivalent interpretations for segment addresses. The access to the data segment must refer to a different segment for each process and thus specifies indirect evaluation through a processor register. The reference to the subroutine refers to the same segment for each process and is thus embedded directly in the program. (Note that this is not meant to imply that references to called programs must always be bound in advance, but that for cases in which advance binding is appropriate, it can be handled that way.)

Other Solutions

The solutions which have been compared are not the only solutions to the problem of shared segment references. They are the ones which have been most thoroughly developed and which appear to have the most promise.

Another solution is to address each segment with a unique integer which is assigned at the time the segment is created, never changed, and not reused even after the segment has disappeared from the system. Call this the unique integer solution. As is explained below in the section on implementation, a similar unique integer is the major component in a capability. In fact, aside from the access control bits which determine whether or not reading, etc., is allowed, the only difference is that in the capability addressing solution, the integers are known to refer to segments which may be accessed, while in the unique integer solution, accessing rights must be determined separately.

Comparison of Relative and Abolute Addresses

The rather lengthy example just completed is a comparatively easy one for the addressing schemes which are based on segment tables. Should we have attempted to construct a shared time-varying multisegment data structure containing internal cross-references and having an existence independent of any particular program or process, we could have done so only by using absolute addresses.

The reason for this is best understood in terms of an example. In Multics, two users can set up private indi-

rection tables to translate from segment numbers contained in a shared data structure to segment table indexes for arbitrary segments. Linkage segments are, of course, an instance of such an indirection table for programs. If these two users want the segment numbers contained in the data structure to have an absolute interpretation, they need only arrange the indirection table properly. But the problem remains as to how the shared data structure can specify how the indirection table should be arranged. This specification requires some way to refer to a segment in a context-independent manner; i.e. it requires absolute addresses.

Multics, of course, provides what is, in effect, a second way of addressing all segments in order to handle this case, namely the full path names of the file system. (A different system might not provide this second way of addressing. There is no inherent reason to insist that every segment is named by the file system or that protection be provided on a per-segment basis in terms of read, write, execute, and append. The reader who doubts this is referred to systems which allow users to define new types of objects, perhaps consisting of many segments and perhaps with very different modes of access being relevant [23, 29, 32, 42, 46].)

In Multics, the use of the file system's full path name as an absolute address may be quite awkward because of its variable length. Furthermore, if the name is embedded in a data structure rather than a program, it will be necessary either to convert the name into a segment number each time it is used or else to use some ad hoc indirect evaluation. One wonders two things: What fraction of the time that a file system name is used would a simple absolute address have sufficed? and How much programmer time is spent minimizing the occurrence of absolute references in order to create programs which run efficiently? If one could measure both the direct cost of the linkage mechanisms and the indirect cost of creating programs which utilize these mechanisms in a reasonable way, it might turn out that one substantial source of inefficiency in the modern multiprogramming systems which rely on shared objects is that they have eliminated the old-fashioned idea of an absolute address for such objects.

Hardware Implementation

The problems of implementing capability addressing are now examined. There are several computers in which every explicit memory access uses an address in the form of a segment capability and word number pair and which allow capabilities to be directly manipulated by user programs in the traditional ways that addresses are used. One is the Chicago Magic Number Computer developed by the Institute for Computer Research at the University of Chicago [19, 20, 21, 41, 47]. This system was never completed. A second computer is the System 250 built by the Plessey Company in England [9, 10, 16, 17, 25, 26]. The Plessey system is available commercially.

A number of systems use capabilities as a protection mechanism at the operating system level but run on conventional machines, including the CAL-TSS system [30, 42], the BCC system [31], the SUE system [40], and the HYDRA system [46]. Since these systems interpret capabilities as addresses in software, they are somewhat less relevant to the present discussion.

Tagged machines such as the Burroughs B6700, the Rice computers [22, 23, 28], and Iliffe's Basic Machine [27] have the potential of implementing what we have described above as capability addressing. However, there appear to be no operating systems yet for these machines which allow capabilities to appear as addresses in arbitrary ways within retained data structures. Thus these systems are also somewhat less relevant to the present discussion.

Based on experience with these various implementations a number of implementation considerations have been clarified.

Integrity of Capabilities

Because of the access control properties of capabilities, it is important that no ordinary program can manufacture or modify the bit pattern with which a capability is represented. Two ways are known for maintaining the integrity of the representation of capabilities: the *tagged* approach and the *partition* approach.

The tagged approach used on the Burroughs B6700, the Rice computers, and the Basic Machine adds one or more tag bits to each word in a segment and to each processor register. This tag is used to specify whether the contents of the word or register are a capability or not. We refer to a piece of information which is not a capability as being *data;* in this sense, data includes programs. The testing and setting of the tag bits is done by the processor each time an access is made, and uses certain simple rules: when a word is copied, the copy is given the same tag as the original; arithmetic and logical instructions must be applied to words tagged as data and always produce a data tag on the result; addressing always checks that the segment address is tagged as a capability; and so on.

The partition approach is used on the Chicago Magic Number Machine and on the Plessey System 250. In the partition approach, each segment is designated at creation as containing either capabilities or data. In addition, there is one set of processor registers for data and one for capabilities. The processor instruction set satisfies rules analogous to those above: data can be copied only into data segments and registers; capabilities can be copied only into capability segments and registers; and so on.

The tagged approach and the partition approach are equivalent in the sense that a structure represented with one approach can be translated into an equivalent structure in the other approach. Which approach is better? The partition approach has several advantages. It is

simple for capabilities and data words to be different lengths. This may be important, since capabilities tend to be big—at least 64 bits long and perhaps longer. (In a bit-oriented machine like the B1700 [44, 45] this argument for the partition approach may vanish, however.) The partition approach allows capabilities to be located by the operating system more easily since they are in known places. This is important for both the Chicago Magic Number Computer and the Plessey System 250 since, in both, the operating system modifies the representation of capabilities under certain conditions. An implementation is described below, however, in which such modifications are not required. Another advantage of the partition approach is that tag bits are not required in memory. A disadvantage of the partition approach is due to the fact that most objects require both data and capabilities and thus require two segments with the partition approach instead of the one required by the tagged approach. Extra capabilities are required to pair the segments as is usually desired. Various operations must then deal with two segments and become more complex. Extra secondary storage accesses may be required to move the pair of segments in and out of memory. To use Saltzer's distinction [37], the advantages of the partition approach are all technological, while some of its disadvantages are intrinsic. Thus one might expect the tagged approach to dominate in the long run.

Address Translation

From the user's point of view, a capability is simply an address for a virtual object and is specified whenever the object is to be accessed. From the implementation point of view, a capability is a bit pattern which specifies to the address translation logic where the physical object which currently represents the virtual object is located. This discussion is restricted to capabilities for segments, although analogous statements apply to other objects. Access type checking, such as checking whether or not a store operation is allowed, is well understood and will be ignored here. Furthermore, it is assumed that segments are not paged; paging may be introduced in an obvious way. Thus the situation is as follows. A user wishes to access some word in some segment. He writes an instruction which specifies a capability for the segment to be accessed and an integer which identifies the word within the segment. What does the hardware do when such an instruction is executed?

In the scheme used on the Chicago Magic Number Computer, there are two representations of segment capabilities, known as *in-form* and *out-form*. These two forms are distinguished by a bit in the capability representation. An in-form capability is used only for segments which are in primary memory. It contains the absolute address of the origin of the segment in primary memory and the length of the segment. In-form capabilities are never allowed to exist on secondary storage; a capability is converted to out-form before being moved to secondary storage. Out-form capabilities contain the secondary storage address of the first record of the segment and a unique sequence number which serves to invalidate capabilities for segments which no longer exist. An attempt to access a segment using an in-form capability causes the hardware to compare the requested word's offset with the length of the segment and, if there is no conflict, to calculate the address of the word of primary memory to be accessed by adding the offset to the address of the origin of the segment. An attempt to access a segment using an out-form capability results in a trap to the system.

The disadvantage of the approach taken on the Chicago Magic Number Computer is that the operating system must frequently convert back and forth between in-form and out-form representations, and must occasionally update the length and address fields in all of the in-form capabilities for some segment. Various schemes are used to minimize this overhead. In retrospect, it appears that the overhead is still substantial.

The Plessey System 250 also uses a scheme of in-form and out-form capabilities. The scheme has several improvements over the Chicago Magic Number Computer, especially in the representation of in-form capabilities. In-form capabilities are evaluated indirectly. There is an indirection table stored in primary memory at a fixed location. Each segment for which capabilities are presently in primary memory has an entry in the indirection table. The in-form capability for an object contains the index of the segment's entry in the indirection table. The entry in the indirection table contains a bit which says whether or not the segment is in primary memory, and contains the segment's secondary storage address and length. If the segment is in primary memory, the entry also contains the segment's primary memory address. The indirection table entry is not fetched on every access to a segment; it is instead fetched whenever a capability is loaded into a processor register.

Using this scheme, the length and primary memory address fields for a segment appear only in one place—the segment's entry in the indirection table (assuming no process which uses the segment is running). This substantially simplifies updating this information. Furthermore, in-form capabilities appear only in primary memory, and out-form capabilities appear only in secondary storage. This convention makes it simple for the system to decide when to convert back and forth between representations. There may still be a substantial overhead in such conversion, however.

The following hardware implementation for address translation is suggested for future implementers of capability-based addressing. It would have been beyond the scope of the hardware available for the Chicago Magic Number Computer but should be reasonable for a computer being designed today. As suggested by Dennis and Van Horn, there is a *unique code* associated with each segment. The unique code is assigned at the time the segment is created and does not change during the life of the segment. It is not reused, even after the segment

disappears from the system. There is only one representation of a capability, and it contains the segment's unique code.

The hardware must be able to find the base address and size for a segment in primary memory once it knows the unique code for the segment. It does this by consulting a hash table maintained in primary memory by the operating system which contains an entry for every segment residing in primary memory. There is a single hash table for all users. For each unique code entered in the hash table there is a presence bit which tells whether or not the segment is in primary memory; additional fields indicate the segment's size and the secondary storage address of its origin. Once an entry has been put in the hash table, the entry remains, even if the segment is written back to secondary storage. Entries age out of the hash table slowly, much as the active segment table entries are handled in Multics. The reason for keeping entries in the table after the segment has left primary memory is to speed up bringing the segment in again, should it be needed.

When a segment is accessed and the hardware looks up its unique code, there are three possible results. The segment may be in primary memory, in which case the appropriate word is accessed. The segment may be in the hash table but not present in primary memory, in which case the hardware causes a type A exception and reports the address of the hash table entry. Finally, the segment may not be in the hash table, in which case the hardware causes a type B exception and reports the unique code.

In the case of a type A exception, the operating system initiates a read using the secondary storage address and size obtained from the hash table entry and blocks the process which was making the access. When the segment has been read in, the hash table entry is updated and the process is allowed to continue. In the case of a type B exception, the operating system first obtains the segment's size and secondary storage address and places them in a newly allocated hash table entry and then proceeds as with a type A exception. Obtaining a segment's size and secondary storage address, given its unique code, is done by consulting a data structure on secondary storage which provides the mapping between the unique code and the size and secondary storage address for all segments. Such a data structure is organized as a modified hash table so as to minimize the expected number of secondary storage accesses required to find an entry.

The final feature of the suggested implementation is a small associative memory which remembers the sizes and primary memory addresses for the unique codes of the most recently accessed segments. Experience with Multics indicates that even a small associative memory can be quite effective [38].

Paging

Experience with Burroughs machines indicates that when segments are allocated in terms of "natural" units for the problem being solved (and the compiler automatically breaks up large arrays), segment sizes are on the average smaller than typical present-day page sizes [2]. On Multics, where the cost of an additional segment is high in terms of additional linkage operations and additional system bookkeeping information, a typical user makes segments larger by combining several different objects in a single segment, thus making his program run more efficiently. This practice should be discouraged, however, since neither the protection mechanism nor the memory management mechanism allows objects thus combined to be treated individually.

Experience with the Plessey System 250 indicates segment sizes more like those of Burroughs' machines than like those of Multics. Thus a paged address translation scheme may perform worse than a nonpaged scheme. M. O'Halloran, one of the designers of the Plessey System 250, suggests that an inverse concept of paging—i.e. many segments per page rather than many pages per segment—is needed to cope with so many very small segments.

Instruction Sets

For capability-based addressing, capabilities must be able to be copied around freely. The capability functions as a basic component in addresses for every object beyond the walls of the processor. The user must be able to do anything with a capability that he would do with an ordinary address on an ordinary machine. Addresses containing capabilities may be used for parameter passing, subroutine returns, elaborate data structures, and so on. Furthermore, every instruction which addresses a word, input/output device, etc., must implicitly or explicitly specify a capability for the object to be accessed.

An *enter instruction* is needed to call a subroutine and simultaneously change the protection domain. The Chicago Magic Number Computer demonstrated that an enter instruction need be no less efficient than an ordinary call instruction. A new type of access for segments is added, called *enter access*. Enter access is weaker than read, write, or execute access and allows only the transfer of control to a fixed entry point, say word zero, using the enter instruction. The enter instruction works like the call used with Figure 6, except that the enter changes the access bits in the capability which is placed in PC to allow reading and executing the program segment. By giving the calling program a capability for the called program's segment which specifies only enter access, the called program, but not the calling program, can obtain the capabilities embedded in the called program's segment.

The Stack

When a program is organized into subroutines, each subroutine may need a temporary storage area for parameters, returns, and local storage. Such storage is often implemented as a stack frame allocated on a common stack each time there is a subroutine call. If the

subroutines run in different protection domains, the stack frames cannot be allocated in a single segment. This is because a subroutine might keep a capability for its stack frame even after it returns control to its caller. It could then use this capability later to interfere with other subroutines to which the stack frame is allocated. The problem could be avoided if there was an efficient mechanism for revoking capabilities.

Assuming revocation is not possible, one solution is to allocate each stack frame in a newly-created segment which will be discarded when the stack frame is no longer needed. Such a scheme adds a substantial overhead to subroutine calling and returning. A better solution is a hardware-managed stack which is not treated as a segment for which capabilities exist, but as a stack of processor registers. The Burroughs B6700 has such a stack, although the implementation relies partly on system compilers and is complex because of its ability to cope with Algol naming. Schroeder's thesis is also relevant [39]. The design of the Chicago Magic Number Computer is quite weak in this respect.

If the stack is arranged so that the temporary storage of the calling routine is unavailable to the called routine, so that the called program cannot alter the return location, and so that parameters can be passed in an orderly way, then the simple enter instruction described above can be used for passing control to a more privileged program, to a less privileged program, or between mutually suspicious programs.

The Own Variable Problem

In addition to temporary storage allocated for a subroutine each time the subroutine is called, a routine may need storage which is allocated when a process first executes the routine and which is retained from call to call of the subroutine by that process. In Algol, such storage is provided by *own* variables. For example, a pseudo random number generator needs an own variable to remember where it is in its pseudo random sequence of numbers. Such information could be retained by the caller and passed as a parameter, but such a solution violates programming generality [13].

Linkage segments, in addition to providing for the indirect evaluation of segment addresses, provide a simple implementation for own variables. It would be unsatisfactory to remove the need for the indirection information in a linkage segment only to find that linkage segments remain so as to implement own variables.

The Algol concept of own variables is not fully general, however. It is likely that languages which provide more control over retention, such as Berry's Oregano [4], will prevail in the long run. Should this be the case, the implementation of own variables based on linkage segments will be too specialized, and one would expect to provide a stack mechanism which allows for retention of stack frames such as the scheme suggested by Bobrow and Wegbreit [5].

Conclusion

Capability-based addressing provides an efficient type of absolute address for an object. The use of such absolute addresses can simplify programming conventions when a general-purpose scheme for shared addresses is required. Recent advances eliminate the need for the modification of the representation of capabilities by the operating system and suggest how to solve the own variable problem in a general way. These advances eliminate the major implementation problems of previously designed systems. A computer using capability-based addressing may now be substantially superior to present systems on the basis of protection, simplicity of programming conventions, and efficient implementation.

References

1. Arden, B.W., Galler, B.A., O'Brien, T.C., and Westervelt, F.H. Program and addressing structure in a time-sharing environment. *J. ACM 13*, 1 (Jan. 1966), 1–16.
2. Batson, A., et al. Measurements of segment size. Proc. 3rd Symp. on Operating Systems Principles. Stanford U., Oct. 1971, 25–29.
3. Bensoussan, A., Clingen, C.T., and Daley, R.C. The MULTICS virtual memory: concepts and design. *Comm. ACM 15*, 5 (May 1972), 308–318.
4. Berry, D.M. Introduction to Oregano. In J. Tou and P. Wegner (Eds.). Sigplan Notices—Proc. Symposium on Data Structures in Programming Languages, Vol. 6., No. 2, Feb. 1971, pp. 171–190.
5. Bobrow, D.G., and Wegbreit, B. A model and stack implementation of multiple environments. *Comm. ACM 16*, 10 (Oct. 1973), 591–603.
6. Burroughs Corporation. Burroughs B5500 Information processing systems reference manual. Detroit, Mich., 1964.
7. Burroughs Corporation. The descriptor—a definition of the B5000 information processing system. Detroit, Mich., 1961.
8. Cleary, J.G. Process handling on Burroughs B6500. Proc. Fourth Australian Comp. Conf., Adelaide, South Australia, 1969, pp. 231–239.
9. Cosserat, D.C. A capability oriented multi-processor system for real-time applications. Presented at the I.C.C. Conf., Washington, D.C., Oct. 1972, 8 pp.
10. Cotton, J.M. The operational requirements for future communications control processors. Presented at Internat. Switching Symp., Cambridge, Mass., June 6–9, 1972, 5 pp.
11. Daley, R.C., and Dennis, J.B. Virtual memory, processes, and sharing in MULTICS. *Comm. ACM 11*, 5 (May 1968), 306–313.
12. Daley, R.C., and Neumann, P.G. A general purpose file system for secondary storage. Proc. AFIPS 1965 FJCC, Vol. 27, Pt. I., AFIPS Press, Montvale, N.J., pp. 213–230.
13. Dennis, J.B. Programming generality, parallelism and computer architecture. Proc. IFIP 1968, North Holland, Amsterdam, pp. C1–7.
14. Dennis, J.B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12*, 4 (Oct. 1965), 589–602.
15. Dennis, J.B., and Van Horn, E.C. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (Mar. 1966), 143–155.
16. England, D.M. Architectural features of System 250. In *Infotech State of the Art Report on Operating Systems*, 1972, 12 pp.
17. England, D.M. Operating System of System 250. Presented at Internat. Switching Symp., Cambridge, Mass., June 6–9, 1972, 5 pp.
18. Evans, D.C., and LeClerc, J.Y. Address mapping and the control of access in an interactive computer. Proc. AFIPS 1967 SJCC, Vol. 30, AFIPS Press, Montvale, N.J., pp. 23–32.

19. Fabry, R.S. A user's view of capabilities. *ICR Quart. Rep. 15* (Nov. 1967), ICR, U. of Chicago, Sec. 1C.
20. Fabry, R.S. Preliminary description of a supervisor for a machine oriented around capabilities. *ICR Quart. Rep. 18* (Aug. 1968), ICR, U. of Chicago, Sec. 1B.
21. Fabry, R.S. List-structured addressing. Ph.D. Th., U. of Chicago, 1971.
22. Feustal, E.A. The Rice research computer—a tagged architecture. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J. pp. 369–377.
23. Feustal, E.A. On the advantages of tagged architecture. *IEEE Trans. on Computers C-22*, 7 (July 1973), 644–656.
24. Graham, G.S., and Denning, P.J. Protection—principles and practice. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 417–429.
25. Halton, D. Hardware of the System 250 for communication control. Presented at the Internat. Switching Symp., Cambridge, Mass., June 6–9, 1972, 7 pp.
26. Hamer-Hodges, K.J. Fault resistance and recovery within System 250. Presented at I.C.C. Conf., Washington, D.C., Oct. 1972, 6 pp.
27. Iliffe, J.K. *Basic machine principles.* American Elsevier, New York, 1968.
28. Iliffe, J.K., and Jodeit, J.G. A dynamic storage allocation scheme. *Comput. J. 5* (Oct. 1962), 200–209.
29. Jones, A.K. Protection structures. Ph.D. Th., Carnegie-Mellon U., 1973.
30. Lampson, B.W. On reliable and extendable operating systems. In Techniques in Software Engineering, NATO Science Committee Workshop Material, Vol. II, Sept. 1969.
31. Lampson, B.W. Dynamic protection structures. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 27–38.
32. Lampson, B.W. Protection. Proc. 5th Ann. Princeton Conf., Princeton U., Mar. 1971, pp. 437–443.
33. LeClerc, J.Y. Memory structures for interactive computers. Project GENIE document No. 40.10.110, U. of California, Berkeley, 1966.
34. Needham, R.M. Protection systems and protection implementations. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 571–578.
35. Organick, E.I. *Computer System Organization—the B5700 B6700 Series.* Academic Press, New York, 1973.
36. Organick, E.I. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, Mass., 1972.
37. Saltzer, J.H. Traffic control in a multiplexed computer system. MAC-TR-30, Proj. MAC, MIT, Cambridge, Mass., 1966.
38. Schroeder, M.D. Performance of the GE-645 associative memory while Multics is in operation. Proc. Workshop on System Performance Evaluation, Cambridge, Mass., 1971, pp. 227–245.
39. Schroeder, M.D. Cooperation of mutually suspicious subsystems in a computer utility. Ph.D. Th., MIT, 1972.
40. Sevick, K.C., et al. Project SUE as a learning experience. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N. J., pp. 331–339.
41. Shepherd, J. Principal design features of the multi-computer. (The Chicago Magic Number Computer). *ICR Quart. Rep. 19* (Nov. 1968), ICR, U. of Chicago, Sec. 1-C.
42. Sturgis, H.E. A postmortem of a time sharing system. Ph.D. Th., U. of California, Berkeley, 1973.
43. Wilkes, M.V. *Time Sharing Computer Systems.* 2nd ed., American Elsevier, New York, 1972.
44. Wilner, W.T. Design of the Burroughs B1700. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 489–497.
45. Wilner, W.T. Burroughs B1700 memory utilization. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 579–586.
46. Wulf, W.A., et al. HYDRA: The kernel of a multiprocessor operating system. Carnegie Mellon U., Comput. Sci. Dep. rep., June 1973.
47. Yngve, V.H. The Chicago Magic Number Computer. *ICR Quart. Rep. 18* (Nov. 1968), ICR, U. of Chicago, Sec. 1-B.

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Key Words and Phrases: operating system, third generation architecture, sensitive instruction, formal requirements, abstract model, proof, virtual machine, virtual memory, hypervisor, virtual machine monitor
CR Categories: 4.32, 4.35, 5.21, 5.22

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15–17, 1973.

This research was supported in part by the U.S. Atomic Energy Commission, Contract No. AT(11-1) Gen 10, Project 14 and in part by the Electronic Systems Division, U.S. Air Force, Hanscom Field, Bedford, Massachusetts under Contract Number F19628-70-0217.

Authors' addresses: Gerald J. Popek, Computer Science Department, University of California, Los Angeles CA 90024; Robert P. Goldberg, Honeywell Information Systems, Waltham, MA 02154.