10

Issues in Capability-Based Architectures

10.1 Introduction

Previous chapters have followed the transition from early descriptor-based computer architectures to the latest in commercially available capability systems. The examination began with the Burroughs B5000 and the Rice University computer. Both of these machines used descriptors, or user-addressable base/limit registers, to define a program's addressing environment. Capability systems extended this idea in several significant ways:

1. Capabilities are *protected addresses*. They can be freely copied, passed as parameters, and transmitted from domain to domain, but cannot be forged or modified by users.
2. Capabilities are *context-independent*. They address the same object independent of the domain or process in which they are used.
3. Capabilities contain access rights as well as addressing information.
4. The address or identifier in a capability is independent of the physical base and limit information used for memory mapping. This identifier is used to locate a single physical descriptor for the addressed object.
5. Capabilities and the objects they address can be saved in long-term storage. They have a lifetime longer than the existence of the process that created them.
6. Capabilities provide a uniform mechanism for *naming* all types of objects in the environment, both hardware and software supported. This enables users to extend the facilities provided by the hardware and vendor-supplied operating system software. Moreover, they provide run-time support for abstraction and object-based programming.

Of course, these capability concepts did not appear at once but evolved over time. Each new system was able to benefit from experience gained in previous systems—even those that were short-lived.

This chapter discusses some of the design issues in capability-based systems. Although each topic could be a chapter in itself, the discussions here are relatively brief. Where possible, tradeoffs are examined in the light of the various systems described.

10.2 Segmentation

This book began by examining the objectives of early systems in diverging from the conventional linear address space. Because each of the systems examined includes a segmented memory space, it is fitting to begin the discussion with a review of segmentation. Segments are the fundamental objects in capability systems; they provide the units of addressing and sharing.

The reasons for segmentation are much the same today as they were in 1960:

1. Segments correspond to logical program entities. They can be used to decompose programs and data structures into units that are meaningful to the programmer.
2. Segmentation allows the logical entities to grow or shrink.
3. Segmentation supports memory relocation and virtual storage.
4. Segments provide logical units of separation, protection, and sharing, both between programs and processes and within a single program or process. Moreover, segments allow for a dynamically changing memory environment.

On early machines, a segment was addressed through a descriptor—usually contained in a descriptor segment. Iliffe's Basic Language Machine included a type for each segment to indicate the kind of information contained there. The type was stored in the descriptor for a segment; it allowed automatic conversion and tagging when data elements were moved from memory into registers.

On current object-based machines, abstract objects are composed of one or more segments. For multisegment objects, a capability for a base segment serves to address the object as a whole. This base segment contains pointers to the other segments forming the object. Segments are thus the basic units addressed by capabilities.

Although segments are the fundamental units of storage allocation, paging can be provided along with segmentation, as in the IBM System/38. Each segment is divided into fixed-sized pages that can be independently located. Paging adds additional storage overhead for the system data structures that maintain information about the memory state. However, the division of physical and virtual storage into fixed-sized units simplifies memory management by removing the memory shuffling and compacting problems.

10.3 Storage of Capabilities

Capability systems have no privileged mode of operation. All privileges, including those permitted to the operating system, are derived from the possession of capabilities. The integrity of the entire system depends on the fact that users cannot forge capabilities or modify them directly. For this reason, the hardware must be able to detect and prohibit any attempt to modify a capability with data instructions. Two different schemes have been used to provide this capability protection: C-lists and tagging.

Most systems have chosen to implement C-lists—often implemented as capability segments—to protect capabilities. Using this protection mechanism, capabilities are stored only in capability segments where they are segregated from user-modifiable data. Separation of capabilities can complicate the construction of record-oriented data structures in which it is natural to mix data and pointers (capabilities). However, a compiler can mask this problem by implementing the structure in two parts or by storing a specifier for the capability, such as the C-list index, instead of the capability itself.

The implementation of C-lists is technologically simpler than tagging; it requires no special hardware on a per-information-unit basis. A single bit in the physical mapping information for each segment indicates whether the segment contains capabilities or data. Or, as is often the case, the distinction is maintained in the access rights of capabilities used to address a segment. Each segment capability indicates whether capability or data access is allowed to that segment. The operating system is privileged because it possesses capabilities that allow data access to user's capability segments.

In addition to implementation advantages, C-lists can provide added efficiency in capability addressing. For example, capabilities can be specified by their index in a C-list. If multiple C-lists are allowed, then multiple indices may be needed. Or, if the number of directly addressable capability segments can be restricted (e.g., the Intel 432's 4 environments or CAP's 16 capability segments), a small number of bits appended to the index can specify which C-list to use. Short forms of addressing can be provided for cases where the most frequently used capabilities are stored at small displacements from the start of the C-list. Thus, C-list schemes often result in a reduction of the number of bits needed to refer to a capability, as compared to the number of bits needed for a general memory address.

The second method of capability protection requires the use of tag bits. Tagging allows capabilities to be stored with user data. The ability to combine capabilities and data can simplify data structuring for the user. Tagging probably has not been used much in the past because of the added memory cost and implementation complexity. Still, several early descriptor systems used tagging when memory was scarce. Certainly memory cost should not be an issue today.

The storage cost of tagging depends on the size of the tagged information units: the smaller the tagged unit the greater the overhead. Most modern systems are byte addressable, but tagging on a byte basis is probably overly expensive. On the System/38 there is one tag bit for each 32-bit word. In a case where tags are not provided on the smallest addressable information unit, capabilities must be aligned on the boundary of a tagged unit, such as a 32-bit word. If capabilities are larger than the tagged unit, as they are with 16-byte System/38 capabilities, alignment must be on larger units.

The System/38 requires that capabilities be aligned on 16-byte boundaries and that the tag bits associated with the four consecutive words be set. The alignment requirement prohibits a user from addressing four consecutive tagged words that do not form a valid capability. For example, two consecutively stored capabilities will cause eight tag bits to be set. A user could address four consecutive words consisting of the last two words of the first capability and the first two words of the second capability. This four-word item is not a valid capability even though the associated tags are set. The alignment requirement could be eliminated at the cost of a second tag bit with each 32-bit word. The second tag bit would indicate whether or not the associated word is the first word of a multi-word capability.

Tag bits can be either part of a data element, which reduces the number of bits in the element, or part of a special storage area associated with each element. The System/38 chose to store the tags outside of the data element in an area accessible only to microcode. When a segment is written to disk, the hardware extracts the tags and stores them in a compact form along with the segment. They are later reinserted when the segment is read back into memory.

StarOS and the Intel 432 have chosen a scheme combining advantages of both tagging and C-lists. These systems support two-part segments that contain a data portion and a capability portion. The descriptor for the segment indicates the size of each portion and the position of the dividing line. Addressing occurs as with separate segments; the type of an operand determines in which portion it is contained. This design reduces the number of segments and mapping descriptors. Since most objects require both a data part and a capability part, the two-part segment scheme halves the number of segments needed to hold an object's representation.

The tagged memory approach is appealing in terms of generality; it allows capabilities to be freely mixed with data, just as pointers or addresses are freely mixed in virtual memory systems. A single stack can serve for local storage of both data and capabilities. The actual implementation of a tagging scheme has a number of complexities. The C-list approach is appealing in its simple implementation and in the addressing efficiencies that can be gained. C-lists can reduce the number of bits needed to address capabilities. Another advantage of C-lists (which will become apparent in later sections) is that they reduce the time required to search for capabilities.

In his comparison of the two techniques, Fabry claims that:

> ...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 [Fabry 74].

It may be too soon to tell, but so far, the partition (C-list) approach has dominated. Credit is probably due to current high-level languages, whose use masks the intrinsic disadvantages of C-lists.

10.4 Capability Representation

A fundamental decision in capability system design is the physical representation of capabilities. A capability contains two parts:

1. an identifier or name for an object, and
2. some access rights or privileges to that object.

The implementation of these fields influences the generality with which the capability can be applied, the work required to manage capabilities in both hardware and software, and the lifetime of objects and capabilities. In evaluating the evolution of the DEC PDP-11 minicomputer, Bell and Strecker state that:

> There is only one mistake that can be made in a computer design that is difficult to recover from—not providing enough address bits for memory addressing and memory management [Bell 76].

This applies to capability systems as well as conventional computers such as the PDP-11. The capability identifier corresponds directly to the address on conventional machines.

Early descriptor and codeword machines used single-word descriptors to address segments. Each descriptor contained all of the mapping information for the segment. Copying of a descriptor caused duplication of the mapping information. This duplication of memory base and limit values for a single segment added complexity to the task of relocation, which the descriptor was meant to simplify.

Two characteristics of these machines simplified the implementation of descriptors. First, the machines had large words and were word addressable. Second, they had relatively small memory spaces. Therefore, the base and limit information could be easily packed into a single word of the word-addressable machine. This removed the need for special alignment of descriptors.

New capability systems must contend with smaller word sizes, larger address spaces, byte addressability, and the greater volume of information needed to manage the system efficiently (e.g., usage and garbage collection bits). An additional problem is the long lifetime of objects on capability systems, in contrast to conventional machines where an object only exists for the lifetime of a program. The longer the object lifetime, the more bits needed for an object's address. These issues have forced an important distinction between the capability itself and the physical mapping information for the object. Thus, we see a separation between the capability, which contains an identifier, and the mapping descriptor, which is generally located in a centralized system table. This distinction is exemplified in the separation of information between Intel 432 access descriptors and object descriptors.

An important component of capability operation is the structure of the identifier. Each object or segment is given an ID at the time of its creation. This ID is often generated by a sequential counter, a clock, a disk address, or the values of indices used to locate the object's descriptor. The number of bits in the ID partly determines the number of objects that can exist at one time. Depending on the number of bits used, the ID can be unique for all time, unique for the life of the object, or unique during the object's residency in primary memory. Each of the possibilities has potential problems.

On most capability systems an object's ID is a direct index into a system mapping table. The mapping table contains descriptors for the object, giving its physical location, size, and so on. For example, capabilities on the Plessey 250 contain a 16-bit index into the System Capability Table. The use of this index as an object's ID places two restrictions on the system. First, the maximum number of addressable segments (at least in primary memory) at any one time is 2^{16} or 64K. Second, the System Capability Table must always be resident in physical memory. On the Plessey 250, the mapping table for 64K descriptors occupies about 589K bytes of storage.

The Intel 432 uses a two-level indexing structure, where the ID is 24 bits, allowing for 16 million objects. The 24-bit ID is divided into two 12-bit table indices. The first selects a descriptor in the central Object Table Directory. This descriptor addresses an object table in which the second index locates the descriptor for the object (this structure was shown in Figure 9-5). The two-level scheme allows the second-level object tables to be swapped out, reducing the amount of required storage overhead. Only the Object Table Directory, which has a maximum size of 64K bytes, need always be memory resident.

Both the Plessey and Intel mechanisms provide for a limited number of objects relative to the lifetime of the system. Therefore, object IDs must be reused when objects are destroyed. One problem with reuse of IDs is knowing what IDs are available to be reused. Since an object's ID is an index in the mapping table, a linked list of free table slots can be kept and used to assign new IDs and descriptors. When a new object is created, a free descriptor is taken and its index becomes the object's ID.

A second problem with reusable IDs is dangling references. When an object is deleted, outstanding capabilities for that object will still reference the mapping table descriptor slot for the object. If a new object is assigned to that descriptor slot, the old object's capability could be used to gain access to the new object. This implies that (1) an object cannot be deleted (or its descriptor reallocated) while capabilities exist for the object, or (2) all capabilities for an object to be deleted must be found and disabled. This problem is discussed further in Section 10.7 on object lifetimes and garbage collection.

Several capability systems have tried to alleviate the problems inherent in indexing schemes by implementing a unique-for-all-time ID space. On such systems, the ID is sufficiently large that the IDs are never used up. For example, object IDs on Hydra are 64 bits, allowing for over 10^{19} objects (it is left as an exercise for the reader to determine how long this address space would last if the system creates, for example, 100 new objects every millisecond). The IBM System/38 architecture also provides a large address space. A 40-bit ID, or segment number, provides for over one trillion segments. This number of segments is not likely to be consumed in the lifetime of most systems. Another unusual feature of the System/38 is that capabilities contain a virtual address that can reference a specific byte. In contrast, on most systems the capability identifies a segment, and a separate byte offset must be supplied independently. This feature is reminiscent of the earlier descriptor machines.

Of course, with a large address space, locating an object's descriptor from its unique ID is more complex than with direct indexing. The Hydra system hashes the unique ID to select one of 128 lists of active object entries in the Active Global Symbol Table. If the object is not found, a search of the Passive Global Symbol Table is made. Because the IBM System/38 uses paging, mapping information is associated with each page of a segment. A Page Directory Table contains the unique virtual page number of each page of primary memory. A hashed lookup is made in the Page Directory Table. If the lookup fails, a page fault occurs and the page must be read in from disk. System/38 capabilities retain the same form whether or not the segment is in primary memory.

All of these schemes require a one- or two-level table lookup to translate a capability identifier into a memory address. This overhead is comparable to the overhead involved in any segmented or virtual memory system. However, access via capabilities may incur additional overheads in order to validate type, access rights, and offset. Also, schemes that allow indirection in capabilities may require additional lookups. On the IBM System/38, some references require a user profile search to validate access rights to the object. References on the Intel 432 may require access to an object selector in memory that specifies the location of a capability operand. Those systems that do not have explicit or implicit capability registers always require an extra memory reference to fetch the capability from memory.

With the use of caches, translation buffers, and other processor-internal registers, there are probably no inherent performance disadvantages of capability system addressing relative to conventional virtual memory systems. All sophisticated modern systems require several levels of addressing indirection and rely on specialized high-speed memory to reduce the apparent overhead.

10.5 Objects

One of the more interesting developments in computer architecture is the relationship between capability hardware and object-based software. Capabilities provide a uniform naming mechanism for all types of objects. In addition to simple segments, capabilities are used to address abstract objects whose representations are stored in segments. This ability to uniformly address complex objects allows the programmer to extend the architectural interface in order to support high-level operating system or application functions.

All object-based systems supply a basic set of system objects. These objects usually provide for low-level resource management and interprocess communication. For example, message ports and processes are common system-supported objects. The IBM System/38 also includes a number of system objects that aid in the construction of database systems. Hardware support of object operations increases performance and hides object implementation.

One possible disadvantage of supporting many objects at the hardware level is the added complexity of the machine. The Intel 432 and IBM System/38 architectures are surely among the most complex in existence. The chances for error in hardware or microcode design and implementation are great. In addition, any high-level mechanism that is moved into hardware must be carefully considered. Because the mechanism and its interface cannot easily be changed, an ill-designed mechanism will simply waste valuable resources. The tradeoff of whether or not to support a particular type in hardware is one of performance and integrity versus machine complexity.

10.6 Protected Procedures and Type Extension

One of the strengths of capability systems is that they allow operating systems and users to extend the hardware interface in a uniform way. This facility is available because capabilities can address operating system and user-implemented objects, as well as hardware supported objects. The only difference is that software-implemented operations are obtained through a CALL or ENTER instruction, while hardware-implemented operations are obtained through hardware instructions.

There are several requirements for a system that allows users to construct their own type managers: that is, protected subsystems that create and manipulate protected objects.

1. A user must be able to construct a type manager: an execution environment consisting of type management procedures and private data segments and objects. This private environment is usually called a domain. The domain is the static representation of the type management system.
2. The type manager must be able to distribute controlled access for its execution environment to its clients. Access is passed through a capability that permits invocation of public procedures but gives no access to any of the private objects in the domain.
3. The hardware must supply a mechanism to invoke the environment. Using the capability for the domain, a client must be able to cause execution of one of the public procedures in the domain. The invocation creates the dynamic type management environment in which the executing procedure has access to domain-local procedures and objects not available to its caller.
4. A type management procedure executing within the domain must be able to create new objects. It must be able to allocate segments in which the representation for new objects can be stored.
5. A type management procedure must be able to return to a client a *sealed capability* for an object. The sealing mechanism must prohibit the client from directly accessing the object's representation. Thus, the client holds the capability as proof of ownership and can pass it on to other users. Any operations on the object are performed by passing the capability as a calling parameter to a type management procedure. The type manager must retain the privilege to *unseal* capabilities of its type, thus gaining access to their representations.

Capability systems have implemented the addressing of programmer-defined type managers in several ways. One common mechanism is to provide a new instance of the type manager for each new object. When an object is created, the type manager returns an enter capability for a new instance of itself. This capability addresses a domain that includes capabilities for type management procedures along with a capability for the representation of the new object instance. The object is manipulated by calling type management procedures through the returned domain capability. The Plessey System 250 Central Capability Block and CAP Enter PRL scheme are examples of this mechanism.

A second mechanism is the use of restriction and amplification of capabilities. The type manager returns restricted capabilities for new object instances to its clients. These restricted capabilities cannot be used to access an object's representation, although they contain type-specific rights. The type manager retains a private capability that permits it to amplify all capabilities of its type. Clients of such a type manager must either have a separate capability for the type management domain or be able to access the domain indirectly through the object capability. The Hydra and Intel 432 systems use restriction and amplification. The Hydra TYPECALL mechanism allows the possessor of the capability for an object to call the object's type manager. The Intel 432 RETRIEVE TYPE DEFINITION instruction returns to the caller a capability for the type management domain of a specified object capability.

Whatever the mechanism, a system must be able to (1) define a procedure execution environment that is distinct from the environment in which the procedure was called and (2) protect the representation of an object so that only its type manager can directly modify its storage. A system that permits users to create such environments simplifies the construction and extension of operating systems by eliminating the notion of privilege that exists in conventional systems. Thus, modules traditionally constructed as part of a monolithic privileged kernel can be implemented and debugged independently as user programs.

10.7 Object Lifetimes and Garbage Collection

The object concept has dramatically changed the conventional concept of secondary storage. Traditional systems have stream-, record-, or block-oriented file systems that preserve information. Program-addressable entities are by default not long-lived; preservation of short-lived entities requires that they be converted to a format acceptable to the file system. On object-based systems, it is natural to wish to preserve objects on secondary storage—that is, to provide a virtual object storage system.

Many capability systems distinguish temporary and permanent objects. The CAL-TSS system became overly complex to some extent because of the decision not to handle secondary memory in the kernel and the inability to name temporary objects in the same way as permanent objects. Plessey 250 provided a virtual segment interface to the user and handled storage of capability segments on disk. Hydra presented a large, flat, object address space. Object storage is provided by both the System/38 and the Intel 432. Both of these systems also have temporary objects that have special treatment. The System/38 reserves part of its address space for temporary objects; these objects do not receive normal protection and are deleted when the system is booted. The Intel 432 gives temporary status to objects allocated out of local stack storage; these objects are implicitly deleted when the procedure in which they are allocated returns.

Object destruction is a difficult problem in capability systems. On the System/38, each object has an owner and the owner can delete the object explicitly. However, on most capability systems there is no concept of an object's owner. An object has some number of users, and each user possesses a capability for the object. Since capabilities can be easily deleted or passed from user to user, the set of users for an object can change dynamically.

It is often difficult to tell when an object is no longer needed. Garbage objects must be deleted or the system's disk or memory will eventually overflow with useless objects. The solution to this problem is garbage collection. A garbage collection process (or processes) is responsible for finding and deleting garbage objects. An object is garbage when it can no longer be accessed by any user. In the simplest case, if all capabilities for the object have been deleted, the object can never be referenced and can safely be destroyed.

One method of garbage detection is to maintain a reference count with each object. The reference count indicates the number of capabilities for the object and must be updated whenever a capability for the object is copied or deleted. When a reference count is decremented to zero, the object can be deleted.

There are at least two problems with reference counts that make them insufficient to solve the garbage collection problem completely. First, circularities can exist in the object structure. For example, if object A contains a capability for object B, while object B contains a capability for object A, then both will have reference counts of at least one. However, if no other capabilities exist for either object, then A and B are not accessible and should be deleted. Second, it is difficult to maintain the integrity of reference counts over system crashes. It would be costly to update a reference count on secondary storage for each capability copy or delete operation. If reference counts are only updated periodically on disk, a system crash can introduce inconsistencies.

Object-based systems must, therefore, resort to garbage collection. A simplified garbage collector would operate as follows. The garbage collection process starts with a set of root objects. In general, each user of the system has a principal C-list or directory that is the root of all objects the user can access; these lists or directories form the roots. The garbage collector first marks every object in the system as being unreachable (there must be some way of locating all objects through a master directory). The garbage collector then marks all objects in the root directories as being reachable. The C-lists of these objects must then be scanned to see if they refer to other objects to be marked as reachable, and so on recursively. Eventually all objects will be marked as reachable or unreachable, and a pass can be made to delete the unreachable objects.

Garbage collection is complex because it must operate concurrently with normal system processing. That is, a garbage collector must operate while objects and capabilities are being created, copied, and deleted. On some systems, such as StarOS, the garbage collector must be concerned with partitioning of the system. It must be able to operate while some nodes are unreachable and still guarantee that it will not delete an accessible object (worse than not deleting a garbage object). Similar problems exist on any system with multiple secondary storage devices in which one or more devices can be off-line at a given time. The garbage collector must be capable of finding objects that are not referenced at all, as well as objects that are members of unreachable cycles. Studies of garbage collection systems and algorithms can be found in [Bishop 77], [Dijkstra 78], and [Almes 80].

A related problem is garbage collection of the address space; that is, the reuse of descriptor slots in object mapping tables, such as the Plessey 250 System Capability Table and the Intel

432 object tables. These table slots must be reallocated because the table, which must be resident in physical memory, cannot map all objects known to the system. Therefore, on most systems, the mapping tables are used only to hold descriptors for objects resident in primary memory. This implies that an object can have different IDs during its lifetime if it is repeatedly moved between primary and secondary storage.

Systems such as Plessey and Intel solve this problem by using two formats for capabilities, an active form and a passive form (sometimes called an inform and outform). A simplified model of the use of active and passive capabilities follows. When each object is created, it is assigned an ID that is guaranteed to be unique for at least the life of the object (although not for all time). This ID might be generated by the physical disk address of the secondary storage for the object. All capabilities for the object, when stored on secondary memory, are kept in passive form. Passive capabilities contain this *long-term* ID.

When an object is brought into primary memory, it is allocated a mapping table descriptor. The mapping table index provides the *short-term* ID for that period of primary memory residency. When a capability is used as a reference, the hardware or software must be able to detect whether the capability is active or passive. An active capability will contain a short-term ID and can be used to directly access an object. A capability in passive form will cause a trap. The software can then examine the long-term ID in the passive capability and convert it to the short-term ID for the object in memory. Or, if the object is not currently in memory, it is swapped in and a descriptor and short-term ID are assigned.

When an object is removed from primary memory, its capabilities are converted to passive form for storage on disk. However, the system must ensure that no active capabilities exist for the object before its mapping table descriptor can be reallocated. Any remaining active capabilities must be in primary memory since they cannot be stored on disk. Therefore, the system can either maintain a reference count for active capabilities or search the C-lists of all resident processes to passivate any active capabilities for the object.

Another design decision to be made in managing secondary object storage is determining how and when an object's secondary storage copy will be updated. The operating system can manage virtual object storage, automatically moving objects between primary and secondary memory. This corresponds to swapping in conventional systems. However, this transparent storage mechanism does not ensure that an object's secondary memory copy is always up to date. Some applications need to guarantee that certain modifications will not be lost by a crash. Another scheme, then, is for the system to provide explicit checkpointing operations for type managers. A type manager performs temporary object modifications in memory and atomically outputs the object to permanent storage by requesting a checkpoint.

An additional problem with object storage is the use of transportable media. Object IDs may be unique for a single system, but are typically not unique for all systems. Moving an object from one computer system to another creates problems because the object's ID may be duplicated on the other system. Backup of objects provides a similar problem. Maintaining capability integrity on transportable media or over networks is an additional concern.

10.8 Object Locking

One advantage of capability systems is the ease with which objects can be shared among several users. This sharing poses problems when users of a shared object must perform multistep atomic transactions. That is, a user may need to execute several object operations with the assurance that no other user can access the same object until the transaction is complete. Exclusion is also required to prohibit a process from operating on inconsistent data when an I/O device is transmitting to object storage. Thus, locking facilities are provided in many capability systems.

The Intel 432 provides instructions to lock and unlock objects. A lock is simply a 16-bit field stored within the data part of a segment; the lock contains a 14-bit process ID and a 2-bit lock type. Objects can be locked either by hardware or software. Some system objects have locks in the processor—defined object data part. Hardware manipulates these locks to obtain exclusion when performing certain operations. Software uses the LOCK OBJECT and UNLOCK OBJECT instructions to obtain mutual exclusion to an object. Execution of a LOCK OBJECT instruction checks if the lock is free; if it is free, the process ID of the current process is stored in the lock and it is marked busy. The instruction returns a boolean result to indicate whether or not the instruction succeeded in obtaining the lock.

The IBM System/38 has a set of higher level lock operations to allow increased concurrency for database operations. Objects can be locked in one of five modes:

1. shared read—user can read, other users can read or write
2. shared read only—user can read, other users can read
3. shared update—user can read or write, other users can read or write
4. exclusive allow read—user can read or write, other users can only read
5. exclusive no read—user can read or write, other users cannot access

The LOCK OBJECT instruction requests one or more locks on one or more objects. The instruction will either succeed in obtaining all locks specified or no locks will be held; that is, if a lock cannot be obtained, all previous locks obtained by the instruction are released. The instruction can specify that the program either wait for locks that are currently unavailable or return immediately. There is also a time-out parameter that indicates the maximum time that the instruction should contend for a lock.

The horizontal microcode on the System/38 maintains a data structure that indicates (for each object for which a lock is held) the type of lock being held and the ID of the requester. Several locks may be held for a single object; this will be indicated in the data structure. The System/38 provides instructions to examine all locks held by a process or an object.

There are, thus, several basic types of locking facilities, including implicit and explicit locks. Implicit locks occur as the result of hardware manipulation of an object; this operation usually requires mutual exclusion. Software may request mutual exclusion or with more sophisticated mechanisms may request only certain types of exclusion to allow maximum concurrency.

10.9 Revocation

One strength of capability systems is the ability to copy and transmit object access rights freely between processes. This strength can also be a weakness when a user needs to restrict access to an object for which capabilities have previously been distributed. In this case, a *revocation* mechanism is needed to retract or cancel the outstanding capabilities. A good examination of such mechanisms is provided by Redell [Redell 74a]. With the exception of the System/38, none of the systems examined have attempted to support revocation.

The System/38 provides revocation through user profiles. Some System/38 capabilities (unauthorized capabilities) do not contain access rights. An object access that specifies such a capability requires a process-local profile table lookup to check the permitted access. The owner of an object can later revoke the object's access rights stored in another process's profile. This scheme combines the concept of access list with capability addressing. However, it adds some complexity to the use of capabilities because unauthorized capabilities require a profile search while authorized capabilities do not. Unauthorized capabilities are not context-independent and, therefore, cannot always be shared with other processes.

A program may wish to restrict capability access in other ways. For example, a calling procedure might want to ensure that a called program does not retain or pass on a capability parameter. The Hydra system provides access rights bits in the capability that specify whether a capability can be stored in a C-list with longer life than the procedure invocation.

Restriction of capability copying can be handled by access rights, but revocation is a more difficult problem. Only the System/38 has considered revocation an important facility to provide. Perhaps other systems have not been willing to pay the cost of the additional overhead. Or, more likely, they were not as concerned with the security and protection problems brought on by the easy propagation of capabilities. These problems will become more important to solve as capability systems find more acceptance in commercial applications.

10.10 Conclusions

This book has followed the history of capability systems from early descriptor machines and Iliffe's codewords, through the first designs by Dennis and Van Horn at MIT and Fabry at Chicago, to the most recent commercial systems by IBM and Intel. Capability systems are of great interest today because of the object approach that is affecting the design of languages, operating systems, and hardware. The object approach promises to influence to a large extent the way in which software is produced in the future.

There are a number of benefits to be gained from capability systems. Although many of these benefits have been described previously, some of the most important ones are restated here.

1. Capability systems permit great flexibility in dynamic sharing of information. This flexibility is due to the global, context-independent interpretation of capabilities, and the ability of users to copy and transmit capabilities freely. Sharing

of data structures does not require operating system intervention for mapping shared structures or for buffering information between processes.

2. Capabilities provide a single uniform mechanism for naming objects of all types. Most traditional systems require many different naming schemes for operating system objects as well as hardware objects.
3. Capability systems provide a good basis for protection and isolation of software components. A procedure's domain can be restricted to include only those objects absolutely required for operation. Different procedures, even in the same subsystem, can execute in disjoint, overlapping, or identical domains. This protection mechanism aids in software reliability.
4. There is nothing "privileged" about protection on a capability system; that is, there is generally no privileged mode of operation. The ability to access objects is defined by the execution domain. Traditionally privileged software systems can thus be implemented as standard user programs. Users can add functions to the operating system base in a uniform way without requiring special privilege.
5. Capability systems support a long-term, single-level object storage system that removes the concept of secondary storage file systems.
6. Capability systems make an explicit distinction between addresses and data. This distinction makes garbage collection of objects possible.

In addition to these advantages, there are a number of associated problems.

1. Capabilities and their associated mapping information can consume additional storage space. For example, System/38 capabilities require 16 bytes of storage. Intel 432 capabilities are only 32 bits in size, but the mapping tables require 16 bytes per object.
2. Garbage collection of the object space may be required to locate objects that are no longer accessible. Garbage collection is a complex and resource consuming task.
3. Garbage collection of the name space may be required to avoid dangling references whenever an object is destroyed. The required capability search is particularly difficult on a system that uses tagging of capabilities, because all memory segments can potentially contain capabilities. On a system using C-lists, only the capability segments need to be searched, but this can still be a costly operation.
4. The advantages of protection and isolation are gained through the use of a protected procedure mechanism. The call or enter mechanism used to invoke a protected procedure can be expensive, since a new addressing environment

must be constructed. (A call on a capability system is analogous in many ways to a context switch on a conventional system.) This cost can force a programming style contrary to that intended. Although these mechanisms provide excellent support for small domains, they may prove expensive for subsystems that need to pass large, complex information structures.

5. Capability systems can be costly in the number of memory references needed to access an operand. Every operand reference requires access to a capability and to several mapping tables (although this overhead exists on any segmented or paged system). Systems with explicit capability registers seem better in this respect, and caches can help as well.
6. Whether or not capabilities can be used to build a secure system is still an open issue. Capability systems typically support unrestricted passing of information, while secure systems require controls on information passing. It is difficult in most capability systems (with the exception of System/38) to determine who has access to an object.

These lists indicate that capability mechanisms may increase programming generality and protection at the possible cost of performance. Although capability systems may simplify the construction of complex systems, they add new complexities to the hardware and operating system implementation. Still, the performance problems suffered by many early capability systems were often due to peculiarities of the individual designs or to hardware poorly matched to the task. There is probably no inherent reason why a capability-based system cannot perform as well as a conventional architecture machine.

It is the success or failure of the object-based programming approach that will eventually determine the success or failure of capability architectures. Although object-based programming can be supported by specialized languages on conventional machines, capability addressing provides run-time protection and error detection. Capabilities can support an environment with a mix of different object-based and conventional languages on the same machine. Whether or not the object approach allows programmers to handle the complexity inherent in sophisticated applications better remains to be demonstrated. We have surely seen only the first generation of object-based and capability-based systems to appear in the commercial marketplace.