Capability-Based Computer Systems

Capability- and Object-Based System Concepts

Although the complexity of computer applications increases yearly, the underlying hardware architecture for applications has remained unchanged for decades. It is, therefore, not surprising that the demands of modern applications have exposed limitations in conventional architectures. For example, many conventional systems lack support in:

1. *Information sharing and communications.* An essential system function is the dynamic sharing and exchange of information, whether on a timesharing system or across a network. Fundamental to the sharing of storage is the addressing or naming of objects. Sharing is difficult on conventional systems because addressing is local to a single process. Sharing would be simplified if addresses could be transmitted between processes and used to access the shared data.
2. *Protection and security.* As information sharing becomes easier, users require access controls on their private data. It must also be possible to share information with, or run programs written by, other users without compromising confidential data. On conventional systems, all of a user's objects are accessible to any program which the user runs. Protection would be enhanced if a user could restrict access to only those objects a program requires for its execution.
3. *Reliable construction and maintenance of complex systems.* Conventional architectures support a single privileged mode of operation. This structure leads to monolithic design; any module needing protection must be part of the single operating system kernel. If, instead, any module could execute within a protected domain, systems could be built as a collection of independent modules extensible by any user.

Over the last several decades, computer industry and university scientists have been searching for alternative architectures that better support these essential functions. One alternative architectural structure is *capability-based* addressing. Capability-based systems support the *object-based* approach to computing.

This book explains the capability/object-based approach and its implications, and examines the features, advantages, and disadvantages of many existing designs. Each chapter presents details of one or more capability-based systems. Table 1-1 lists the systems described, where they were developed, and when they were designed or introduced.

System	Developer	Year	Attributes
Rice University Computer	Rice University	1959	segmented memory with "codeword" addressing
Burroughs B5000	Burroughs Corp.	1961	stack machine with descriptor addressing
Basic Language Machine	International Computers Ltd., U.K.	1964	high-level machine with codeword addressing
Dennis and Van Horn Supervisor	MIT	1966	conceptual design for capability supervisor
PDP-1 Time-sharing System	MIT	1967	capability supervisor
Multicomputer/ Magic Number Machine	University of Chicago Institute for Computer Research	1967	first capability hardware system design
CAL-TSS	U.C. Berkeley Computer Center	1968	capability operating system for CDC 6400
System 250	Plessey Corp., U.K.	1969	first industrial capability hardware and software system
CAP Computer	University of Cambridge, U.K.	1970	capability hardware with microcode support
Hydra	Carnegie-Mellon University	1971	object-based multiprocessor O.S.
StarOS	Carnegie-Mellon University	1975	object-based multiprocessor O.S.
System/38	IBM, Rochester, MN.	1978	first major commercial capability system, tagged capabilities
iAPX 432	Intel, Aloha, OR.	1981	highly-integrated object-based microprocessor system

Table 1-1: Major Descriptor and Capability Systems

Before surveying these systems at a detailed architectural level, it is useful to introduce the concepts of capabilities and object-based systems. This chapter defines the concept of capability, describes the use of capabilities in memory addressing and protection, introduces the object-based programming approach, and relates object-based systems to capability-based addressing.

Simplified examples of capability-based and conventional computer systems are presented throughout this chapter. These examples are meant to introduce the capability model by contrasting it with more traditional addressing mechanisms. In fact, many design choices are possible in both domains, and many conventional systems exhibit some of the properties of capability systems. No one of the following models is representative of all capability or conventional systems.

1.1 Capability-Based Systems

Capability-based systems differ significantly from conventional computer systems. Capabilities provide (1) a single mechanism to address both primary and secondary memory, and (2) a single mechanism to address both hardware and software resources. While solving many difficult problems in complex system design, capability systems introduce new challenges of their own.

Conceptually, a capability is a token, ticket, or key that gives the possessor permission to access an entity or object in a computer system. A capability is implemented as a data structure that contains two items of information: a *unique object identifier* and *access rights*, as shown in Figure 1-1.

The identifier *addresses* or *names* a single object in the computer system. An object, in this context, can be any logical or physical entity, such as a segment of memory, an array, a file, a

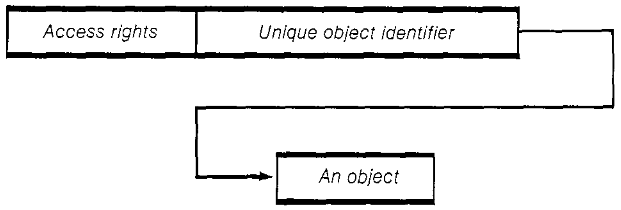

Figure 1-1: A Capability

line printer, or a message port. The access rights define the *operations* that can be performed on that object. For example, the access rights can permit read-only access to a memory segment or send-and-receive access to a message port.

Each user, program, or procedure in a capability system has access to a *list of capabilities*. These capabilities identify all of the objects which that user, program, or procedure is permitted to access. To specify an object, the user provides the *index* of a capability in the list. For example, to output a record to a file, the user might call the file system as follows:

```
PUT( file_capability , "this is a record" );
```

The capability specified in the call serves two purposes. First, it identifies the file to be written. Second, it indicates whether the operation to be performed (PUT in this case) is permitted.

A capability thus provides addressing and access rights to an object. Capabilities are the basis for object protection; a program cannot access an object unless its capability list contains a suitably privileged capability for the object. Therefore, the system must prohibit a program from directly modifying the bits in a capability. If a program could modify the bits in a capability, it could forge access to any object in the system by changing the identifier and access rights fields.

Capability system integrity is usually maintained by prohibiting direct program modification of the capability list. The capability list is modified only by the operating system or the hardware. However, programs can obtain new capabilities by executing operating system or hardware operations. For example, when a program calls an operating system routine to create a new file, the operating system stores a capability for that file in the program's capability list. A capability system also provides other capability operations. Examples include operations to:

1. Move capabilities to different locations in a capability list.
2. Delete a capability.
3. Restrict the rights in a capability, producing a less-privileged version.
4. Pass a capability as a parameter to a procedure.
5. Transmit a capability to another user in the system.

Thus, a program can execute direct control over the movement of capabilities and can share capabilities, and therefore, objects, with other programs and users.

It is possible for a user to have several capability lists. One list will generally be the master capability list containing capabilities for secondary lists, and so on. This structure is similar to a multi-level directory system, but, while directories address only files, capabilities address objects of many types.

1.1.1 Memory Addressing in Computer Systems

This section presents simplified models for both conventional and capability-based memory addressing systems. Although capabilities can control access to many object types, early capability-based systems concentrated on using capabilities for primary memory addressing. The first use of capabilities for memory protection was in the Chicago Magic Number Machine [Fabry 67, Yngve 68], and an early description of capability-based memory protection appeared in Wilkes' book on timesharing systems [Wilkes 68]. Later, [Fabry 74] described the advantages of capabilities for generalized addressing and sharing.

For purposes of a simplified model, consider a conventional computer supporting a multiprogramming system in which each program executes within a single process. A program is divided into a collection of segments, where a segment is a contiguous section of memory that represents some logical entity, such as a procedure or array. A process defines a program's address space: that is, the memory segments it can access. The process also contains data structures that describe the user, and a directory that contains the names of a set of files. These files represent the user's long-term storage.

When a program is run, the operating system creates a process-local segment table that defines the memory segments available to the program. The segment table is a list of *descriptors* that contain physical information about each segment. Figure 1-2 shows example formats for a process virtual address and segment table descriptor. The operating system loads various segments needed by the program into primary memory, and loads the segment table descriptors with the physical address and length of each segment. A process can then access segments by reading from or writing to virtual addresses.

Each virtual address contains two fields: the segment number and the offset of a memory element within that specified segment. On each virtual address reference the hardware uses the segment number field as an index to locate an entry in the

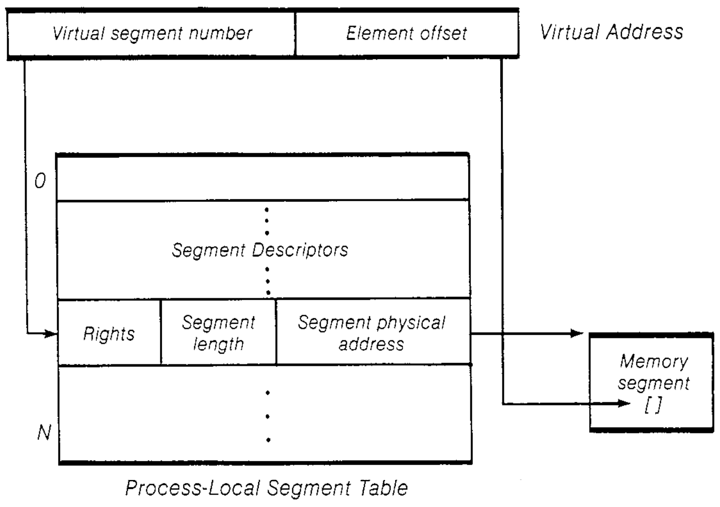

Figure 1-2: Conventional Segment Address Translation

process segment table. This descriptor contains the physical location of the segment. The length field in the descriptor is used to check that the offset in the virtual address is within the segment bounds. The rights field in the segment table entry indicates the type of access permitted to that segment (for example, read or write).

The model shown in Figure 1-2 has the following properties:

1. The system supports a segmented process virtual address space. A virtual address is local to the process and is translated through the process-local segment table.
2. A program can construct any virtual address and can attempt to read or write that address. On each reference, the hardware ensures that (a) the segment exists, (b) the offset is valid, and (c) the attempted operation is permitted. Otherwise, an error is signaled.
3. Loading of segment table entries is a privileged operation and can be accomplished only by the operating system. In general, a segment table is created at the time a program is loaded. The program then executes in a static addressing environment.
4. Sharing of segments between processes requires that the operating system arrange for both process-local segment tables to address the shared segments. If two processes wish to use the same virtual address to access a shared segment,

the segment descriptors must be in the same locations in both segment tables.

5. Any dynamic sharing of segments requires operating system intervention to load segment descriptors.

A *capability-based system* also supports the concept of a process that defines a program's execution environment. In the capability system, each process has a capability list that defines the segments it can access. Instead of the segment table descriptors available to the conventional system hardware, the capability addressing system consists of a set of *capability registers*. The *program* can execute hardware instructions to transfer capabilities between the capability list and the capability registers. The number of capability registers is generally small compared to the size of the capability list. Thus, at any time, the capability registers define a subset of the potentially accessible segments that can be physically addressed by the hardware. A simplified hardware model for this system is shown in Figure 1-3.

The model shown in Figure 1-3 has the following properties:

1. The system has a segmented virtual address space. A segment of memory can only be addressed by an instruction if a capability for that segment has been loaded into a capability register.

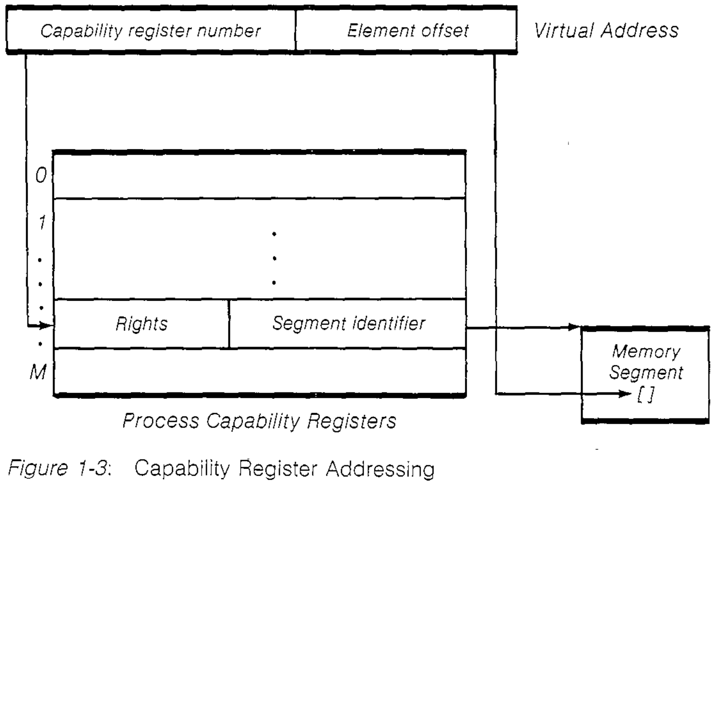

Figure 1-3: Capability Register Addressing

2. While loading of a segment descriptor in the conventional system is privileged, loading of a capability register is not. Instead of controlling the loading of the register, the capability system controls *the pattern of bits* that can be loaded. Only a valid capability can be loaded into a capability register.
3. The capability system provides a dynamically changing address space. The address space changes whenever the program changes one of the capability registers.
4. A virtual address identifies a process-local capability register. In this sense, a virtual address has similar properties to a virtual address in the conventional system. Sharing a virtual address does not in itself give access to the same segment.
5. A capability, however, is *not* process-local. Capabilities are *context independent*; that is, the segment addressed by a capability is independent of the process using that capability. A process can share a segment by copying or sending a capability from its capability list to the capability list of a cooperating process. Each of the processes can then access the segment.

One important difference between the conventional and capability approaches involves the ability of a program to affect system-wide or process-local objects. In the conventional system, a program executes within a virtual address space defined by a process. Every procedure called by that program has access to the process address space, including segments and files. Every procedure executes within an identical protection environment.

In the capability system, a procedure can only affect objects for which capability registers have been loaded. It is possible, therefore, for different procedures called by the same program to have access to different segments. Although all procedures may have the potential to load capability registers from the capability list, some procedures may choose to execute within a very small addressing sphere.

The ability to restrict the execution or addressing environment of a procedure has several benefits. First, if a procedure is allowed access only to those segments absolutely needed, the hardware can detect any erroneous references. For example, a reference past the end of an array might be caught before it destroys another variable. Second, if a procedure is found to be in error, it is easy to determine what segments might have been affected. If the segments that could have been modified were local to the procedure, recovery might be substantially easier.

Most capability systems go a step further by allowing each procedure to have a *private* capability list. A procedure can thus protect its objects from accidental or malicious access by its callers, and a program can protect its objects from access by called procedures. Every procedure can have, in effect, its own address space. To permit a procedure access to a local object, a program can pass a capability for the object as a parameter when the procedure is called. Therefore, in a capability system, every procedure can be protected from every other procedure because each has a private capability list. When one procedure calls another, it knows that the called procedure can access only local objects for which capabilities are passed.

1.1.2 The Context of an Address

Each object in a capability system has a unique identifier. Conceptually, each object's identifier is unique for all time. That is, an identifier is assigned when an object is created and that identifier is never reused, even after the object is deleted. During the object's lifetime, its unique identifier is used within capabilities to specify the object. An attempt to use a capability with an identifier for a deleted object causes an error.

In practice, the object identifier field of a capability must be used by hardware to locate the object. From the hardware viewpoint, the identifier is an address—either the address of a segment or perhaps the address of a central descriptor that contains physical information about the segment. The need to handle addresses efficiently in hardware typically causes addresses to be small—16 or 32 bits, for example. For this reason, identifiers tend to have too few bits to be unique for all time. However, the choice of the number of bits in an identifier is an important system design decision that dictates the way in which capabilities can be used.

In conventional systems, an address is meaningful only within a single process. In a capability system, addresses (capabilities and their identifiers) are context-independent. That is, the interpretation of a capability is independent of the process using it. The unique identifier within a capability must have a system-wide interpretation. Unique identifiers must be large enough to address all of the segments likely to be in use by all executing processes at any time. This allows capabilities to be freely passed between processes and used to access shared data.

Addressing on most conventional systems is restricted in terms of time as well as context. An address is meaningful only within the lifetime of a single process. Therefore, addresses cannot be used to name objects whose lifetimes are greater than the process creating the objects. If a process wishes to create a long-term storage object, such as a file, it must interface to the file system. Files typically require different naming, protection, and storage mechanisms than memory segments.

A significant advance made possible by capabilities is the naming and protection of both long-term and short-term objects with a single mechanism. If the identifier field is very large, it may be possible to implement identifiers unique for all time. Each object is addressed by capabilities containing its unique identifier, independent of whether it is stored in primary or secondary memory. The operating system or hardware can maintain data structures that indicate the location of each object. If a program attempts to access an object in secondary memory, the hardware or operating system can bring the object into primary memory so that the operation can be completed. From the program's point of view, however, there is a single-level address space. Capabilities, as well as data, can be saved for long periods of time and stored in secondary memory.

There are, therefore, several contexts in which an address can have meaning. For example, for:

1. Primary memory segments of a single process.
2. Primary memory segments of all existing processes.
3. All existing segments in both primary and secondary memory.

Most conventional systems support only type 1, while capabilities allow for any of the listed addressing types. More importantly, while conventional systems are concerned only with the protection of *data*, capability systems are concerned also with the protection of *addresses*. A process on a capability system cannot fabricate new addresses. As systems become more general in their addressing structure as in types 2 and 3, the protection of addresses becomes crucial to the integrity of the system.

1.1.3 Protection in Computer Systems

Lampson contrasts the capability approach with the traditional approach by showing the structure of protection information needed in a traditional operating system [Lampson 71]. Figure 1-4 depicts an access matrix showing the privileges that each system user is permitted with respect to each system object. For example, user Fred has read and write privileges to File1 and no privileges to File2, while user Sandy is allowed to read both files.

System Objects

System Users		File1	File2	File3	ProcessJ	Mailbox10	...
	Fred	Read Write		Read	Delete Suspend Wakeup	Send	
	Sandy	Read	Read			Send Receive	
	Molly			Read Write		Send	
	⋮						

Figure 1-4: System Object Access Matrix

One conventional approach to the maintenance of protection information is *access control lists*, in which the operating system keeps an *access list* for each object in the system. Each object's list contains the names of users permitted access to the object and the privileges they may exercise. When a user attempts to access an object, the operating system checks the access list associated with that object to see if the operation is authorized. Each of the columns of Figure 1-4 represents an access control list.

The capability system offers an alternative structure in which the operating system arranges protection information by user instead of by object. A *capability list* is associated with each user in the system. Each capability contains the name of an object in the system and the user's permitted privileges for accessing the object. To access an object, the user specifies a capability in the local capability list. Each of the rows of Figure 1-4 represents a capability list. Figure 1-5 shows an access list

Access List for Mailbox10	Capability list for Fred
Fred(send)	File1(read,write)
Sandy(send,receive)	File3(read)
Molly(send)	ProcessJ(delete,suspend,wakeup)
⋮	Mailbox10(send)
	⋮

Figure 1-5: Access Control and Capability Lists

and a capability list derived from the protection matrix in Figure 1-4.

One important difference between the capability list and access list is the user's ability to *name* objects. In the access list approach, a user can attempt to name any object in the system as the target of an operation. The system then checks that object's access list. In the capability system, however, a user can only name those objects for which a capability is held: that is, to which some access is permitted.

In either case, the integrity of the system is only as good as the integrity of the data structures used to maintain the protection information. Both access control list and capability list mechanisms must be carefully controlled so that users cannot gain unauthorized access to an object.

Similar protection options exist outside the computer world. A useful analogy is the control of a safe deposit box. Suppose, for example, that Carla wishes to keep all of her valuables in a safe deposit box in the bank. On occasion, she would like one or more trustworthy friends to make deposits or withdrawals. There are basically two ways that the bank can control access to the box. First, the bank can maintain a list of people authorized to access the box. To make a transaction, Carla or any of her friends must prove their identity to the bank's satisfaction. The bank checks the (access control) list for Carla's safe deposit box and allows the transaction if the person is authorized. Or, instead of maintaining a list, the bank can issue Carla one or more keys to her safe deposit box. If Carla needs to have a friend access the box, she simply gives a key to the friend.

A number of observations can be made about these two alternative protection systems. The properties of the access list scheme are:

1. The bank must maintain a list for each safe deposit box.
2. The bank must ensure the validity of the list at all times (e.g., it cannot allow the night watchman to add a name).
3. The bank must be able to verify the identity of those asking to use a box.
4. To allow a new person to use the box, the owner must visit the bank, verify that he or she is the owner of the box, and have the new name added to the list.
5. A friend cannot extend his or her privilege to someone else.
6. If a friend becomes untrustworthy, the owner can visit the bank and have that person's name removed from the list.

The alternative scheme involving keys has the following properties:

1. The bank need not be involved in any transactions once the keys are given, except to allow a valid keyholder into the vault.
2. The physical lock and key system must be relatively secure; that is, it must be extremely difficult to forge a key or to pick the lock on a safe deposit box.
3. The owner of a box can simply pass a key to anyone who needs to access the box.
4. Once a key has been passed to a friend, it is difficult to keep them from giving the key to someone else.
5. Once a friend has made a transaction, the owner can ask for the key back, although it may not be possible to know whether or not the friend has made a copy.

The advantage of the key-based system is ease of use for both the bank and customer. However, if today's friends are likely to become tomorrow's enemies, the access list has the advantage of simple guaranteed access removal. Of course, the access control list and the key (or capability) systems are not mutually exclusive, and can be combined in either the computer or banking world to provide the advantages of both systems for increased protection.

1.2 The Object-Based Approach

Over the last few decades, several areas of computer science have converged on a single approach to system design. This approach, known as *object-based computing*, seeks to raise the level of abstraction in system design. The events that have encouraged object-based design include:

1. Advances in computer architecture, including capability systems and hardware support for operating systems concepts.
2. Advances in programming languages, as demonstrated in Simula [Dahl 66], Pascal [Jensen 75], Smalltalk [Ingalls 78], CLU [Liskov 77], and Ada [DOD 80].
3. Advances in programming methodology, including modularization and information hiding [Parnas 72] and monitors [Hoare 74].

This section introduces the object approach and discusses its relationship to capability-based computer systems.

What is object-based computing? Simply stated, the object approach is a method of structuring systems that supports *ab-*

straction. It is a philosophy of system design that decomposes a problem into (1) a set of *abstract object types*, or resources in the system, and (2) a set of *operations* that manipulate *instances* of each object type.

To make this idea more concrete, consider the following simplified example. Imagine that we are programming a traffic simulation for a city. First, define a set of objects that represent, abstractly, the fundamental entities that make up the traffic system. Some of the object *types* for the traffic simulation might be:

- passenger
- bus
- bus stop
- taxi
- car

Then, for each object type, define the operations that can be performed. Bus objects, for example, might support the operations:

- PUT_BUS_INTO_SERVICE(bus_number)
- MOVE_BUS(bus_number, bus_stop)
- LOAD_PASSENGERS(bus_number, passenger_list)
- UNLOAD_PASSENGERS(bus_number, passenger_list)
- GET_PASSENGER_COUNT(bus_number)
- GET_POSITION(bus_number)
- REMOVE_BUS_FROM_SERVICE(bus_number)

Each bus operation accepts a bus number as a parameter. At any time there may be many bus objects in the system, and we identify each bus by a unique number. Each of these bus objects is an *instance* of the *type* bus. The *type* of an object identifies it as a member of a class of objects that share some behavioral properties, such as the set of operations that can be performed on them.

What has been gained by defining the system in this way? First, there now exist a fundamental set of objects and operations for the simulation. We can now implement the procedures to perform the operations on each type of object. Since only a limited number of procedures operate on each object type, access to the internal data structures used to maintain the state of each type can be restricted. This isolation of the knowledge of those data structures should simplify any future changes to one of the object abstractions because only a limited set of procedures is affected.

Second, and more importantly, we have raised the *level of abstraction* in the simulation program. That is, we can now program the simulation using buses, passengers, and bus stops as the fundamental objects, instead of bits, bytes, and words, which are normally provided by the underlying hardware. The buses and passengers are our data types just as bits and bytes are the data types supported in hardware. The simulation program will consist mainly of control structures plus procedure calls to perform operations on instances of our fundamental objects.

Of course, in this example, the procedures implementing the operations are programmed using lower-level objects, such as bytes, words, and so on. Or, they may be further decomposed into simpler abstract objects that are then implemented at a low level. Object-based systems provide a fundamental set of objects that can be used for computing. From this basis, the programmer constructs new higher-level object types using combinations of the fundamental objects. In this way the system is extended to provide new features by creating more sophisticated abstractions.

This methodology aims to increase productivity, improve reliability, and ease system modification. Through the use of well-defined and well-controlled object interfaces, systems designers hope to simplify the construction of complex computer systems.

1.2.1 Capabilities and Object-Based Systems

In the simulation example, each object is identified by a unique number. To move a bus from one stop to another, we call the MOVE_BUS operation with the unique number of the bus to move. For purposes of the simple simulation, a small set of integers suffices to identify the buses or other objects. No protection is needed because these objects are implemented and used by a single program.

The use of the object approach to build operating system facilities presents different requirements. For example, suppose we wish to build a calendar system to keep track of scheduled meetings, deadlines, reminders, and so on. The fundamental object of the calendar system, from the user's point of view, is a calendar object. Our calendar management system provides routines that create a new calendar, and modify, query, or display an existing calendar. Many users in the system will, of course, want to use this facility.

Several familiar issues now arise: (1) how does a user name a calendar object, (2) how is that calendar protected from access by other users, and (3) how can calendars be shared under controlled circumstances? Only the owner of a calendar should be able to make changes, and the annotations in each calendar must be protected from other users, since they might contain confidential information. However, a user might permit selected other users to check if he or she is busy during a certain time, in order to automate the scheduling of meetings.

Capabilities provide a solution to these problems. When a user creates a new calendar, the calendar creation routine allocates a segment of memory for which it receives a capability. This segment is used to store data structures that will hold the calendar's state. The create routine uses this capability to initialize the data structures, and then returns it to the caller as proof of ownership of the calendar. In order to later modify or query the calendar, the user specifies the returned capability; the capability identifies the calendar and allows the modify or query procedure to gain access to the data structures. Only a user with a valid capability can access a calendar.

A weakness with this scenario is that the calendar system cannot prevent the calendar owner from using its capability to access the data structures directly. The calendar system would like to protect its data structures both to ensure consistency and to guarantee that future changes in data format are invisible outside of the subsystem. In addition, if a user passes a calendar capability to another user, the second user can then modify the data structures or read confidential information.

These problems exist because the calendar system returns a fully-privileged calendar capability to the user. Instead, what is needed is a capability that identifies a specific calendar and is proof of ownership, but does not allow direct access to the underlying data structures. In other words, the calendar system would like to return only *restricted* capabilities to its clients. However, the calendar system must retain the ability to later *amplify* the privileges in one of its restricted capabilities so that it can access the data structures for a calendar.

There are several ways of providing type managers with this special ability. (These mechanisms are examined in detail throughout the book.) However, given this power over capabilities for its objects, a type manager can ensure that its clients operate only through the well-defined object operation interface. A client can pass a capability parameter to the type manager when requesting a service, but cannot otherwise use the capability to read or write the object it addresses. This facility is fundamental to any system that allows creation and protection of new system types.

1.3 Summary

The capability concept can be applied in hardware and software to many problems in computer system design. Capabilities provide a different way of thinking about addressing, protection, and sharing of objects. Some of the properties of capabilities illustrated in this chapter include their use in:

1. Addressing primary memory in a computer system.
2. Sharing objects.
3. Providing a uniform means of addressing short- and long-term storage.
4. Support for a dynamic addressing environment.
5. Support for data abstraction and information hiding.

These, of course, are advantages of capability-based systems. The most important advantage is support for object-based programming. Object-based programming methodology seeks to simplify the design, implementation, debugging, and maintenance of sophisticated applications. While capabilities solve a number of system problems, their use raises a whole new set of concerns. And, as is often the case in computer system design, the concept is much simpler than the implementation.

The remainder of this book is devoted to examining many different capability-based and object-based designs. The characteristics of each system are described with emphasis on addressing, protection, and object management. Each system represents a different set of tradeoffs and presents different advantages and disadvantages. When comparing the systems, consider the differences in goals, technologies, and resources available to the system developers.

The final chapter of this book considers issues in capability system design common to all of the systems described. A few of the questions to be considered follow. It may be useful to remember these questions when examining each system design.

1. What is the structure of an address?
2. How is a capability represented? How is a capability used to locate an object?

3. How are capabilities protected?
4. What is the lifetime of a capability?
5. What types of objects are supported by the hardware and software?
6. What is the lifetime of an object?
7. How can users extend the primitive set of objects provided by the base hardware and software?

1.4 For Further Reading

The concept of capability is formally defined in the 1966 paper by Dennis and Van Horn [Dennis 66]. Chapter 3 examines this paper in some detail. The paper by Fabry [Fabry 74] compares capability addressing and conventional segmented addressing of primary memory, while Redell [Redell 74a] describes issues in capability systems and the use of sealing mechanisms that support the addition of new object types to a system. These papers are a fundamental part of capability literature.

Capability systems have been discussed in various contexts. Two papers by Lampson [Lampson 69 and Lampson 71] describe the requirements for protection in operating systems and the capability protection model. The surveys by Linden [Linden 76] and Denning [Denning 76], which appeared in a special issue of *ACM Computing Surveys*, describe capability systems and their relationship to security and fault tolerance in operating systems.

The architecture books by Myers [Myers 82] and Iliffe [Iliffe 82] also discuss some of the systems described in this book. Myers' book contains details of Sward [Myers 80], a capability-based research system built at IBM that is omitted here. A capability system model, as well as discussion of some existing capability systems, appears in the book by Gehringer [Gehringer 82]. Jones [Jones 78a] provides a good introduction to the concepts of object-based programming.

The Burroughs B5000 computer. (Courtesy Burroughs Corporation.)