3

Early Capability Architectures

3.1 Introduction

Although the Burroughs, Rice, and BLM systems included capability-like addressing structures, the word "capability" was not introduced until 1966, by Dennis and Van Horn of MIT [Dennis 66]. Dennis and Van Horn defined a hypothetical operating system supervisor for a multiprogramming system. Multiprogramming systems were already in use at that time; however, many difficult problems had yet to be solved. The MIT design used the concept of capability addressing to provide a uniform solution to several issues in multiprogramming systems, including sharing and cooperation between processes, protection of processes, debugging, and naming of objects.

The concept of capability addressing presented by Dennis and Van Horn quickly found its way into several hardware and software systems. This chapter first describes the Dennis and Van Horn supervisor and its use of capabilities and then examines some of the early systems influenced by its design.

3.2 Dennis and Van Horn's Supervisor

Dennis and Van Horn's operating system supervisor is defined by a set of objects and a set of operations for each type of object. The operations, implemented by the supervisor, are called *meta-instructions*. To describe this system and its meta-instructions involves the introduction of the following terms:

- *segment*—an addressable collection of consecutive memory words,
- *process*—a thread of control through an instruction stream, and
- *computation*—one or more processes that share an addressing environment and cooperate to solve a task.

A process is the basic execution entity. A process executes within an environment called a *sphere of protection* or *domain*. The sphere of protection for a process defines the segments that it can address, the I/O operations that it can perform, and other objects, such as directories, that it can manipulate.

As part of its state, a process in the Dennis and Van Horn system contains a pointer to a list of *capabilities*, called a *C-list* for short. Each capability in the C-list names an object in the system and specifies the access rights permitted to that object. The name is a pointer that the supervisor can use to locate the object; however, the authors suggest that systems avoid the use of physical attributes such as addresses for pointers. The name is a unique bit string assigned to an object when it is created. The naming of objects in an address-independent manner simplifies relocation and management of memory.

The access rights in a capability are specific to the type of object named. For example, the rights bits allow execute, read, read/execute, read/write, or read/write/execute access for segments. Each capability also contains a single bit indicating whether or not its possessor is the owner of the object. An object's owner has special rights with respect to the object, such as the ability to delete it.

Each process in the system, then, has a pointer to a single C-list containing capabilities naming all of the objects it can access. When executing a supervisor meta-instruction, the process specifies capabilities by their index in the C-list. A computation consists of several potentially cooperating processes that share a single sphere of protection. That is, the processes in a computation share the same C-list. Figure 3-1 shows three processes that make up two distinct computations.

The supervisor allows the creation of tree-structured processes. Using a FORK operation, a process can create a parallel process executing within its sphere of protection. In addition, a process can create and control subprocesses, called *inferior spheres*, that execute in separate subordinate domains. To create an inferior sphere, a process executes a CREATE SPHERE meta-instruction. As a parameter to the meta-instruction, the

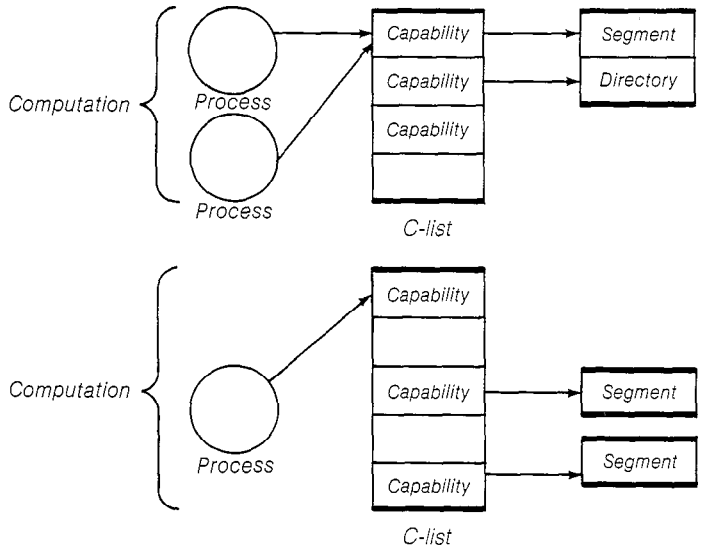

Figure 3-1: Processes, Computations, and C-Lists

process specifies an entry in its C-list, in which the supervisor places a capability for the inferior. This capability can then be used to control the inferior process.

When a process executes a CREATE SPHERE meta-instruction, the supervisor creates the inferior with an empty C-list. Using its capability for the inferior, the parent process can execute meta-instructions to:

- move capabilities from its C-list to the inferior's C-list,
- start and stop the inferior,
- examine or change the inferior's state, and
- remove capabilities from the inferior's C-list.

The creating process can construct any sphere of protection desired for the inferior, with the restriction that the superior's C-list must contain any capabilities to be copied to the inferior's C-list. Table 3-1 lists the Dennis and Van Horn meta-instructions that operate on inferior spheres, capabilities, and directories (which are described in Section 3.6).

Inferior spheres are useful for debugging. When testing a new procedure, a user might like to constrain the environment in which the procedure can execute so that an error will not accidentally destroy the user's objects. When a process creates an inferior sphere, it specifies the address of a procedure to handle any special conditions. If an error or exception is de-

CREATE SPHERE	create an inferior sphere and return a process capability to the creator
GRANT	copy a capability to an inferior's C-list with specified access rights
EXAMINE	copy inferior's capability into superior's C-list
UNGRANT	delete capability from inferior's C-list
ENTER	call protected procedure with one capability parameter
RELEASE	remove capability from C-list
CREATE	create a new segment, entry, or directory
PLACE	insert capability and text name into directory
ACQUIRE	search directory for text name and copy associated capability into C-list
REMOVE	remove named item and associated capability from directory
DELETE	delete object specified by name
LINK	obtain capability for another user's root directory and insert in C-list

Table 3-1: Dennis and Van Horn Supervisor Capability Operations

tected in the inferior, the supervisor creates a new process within the sphere of the parent process to execute the error-handling procedure. Or, the inferior can explicitly signal the parent through special meta-instructions. This feature allows a superior to build a supervisory environment for its inferior which is equivalent to that provided by the superior's parent (or by the supervisor).

Although C-lists provide for object addressing, they do not satisfy the need for object naming. Users in a multiprogramming system must be able to identify objects (particularly long-term objects such as files) using mnemonic character string names. They must also be able to share objects with other users in the system. In order to allow users to name objects and retain them indefinitely, the supervisor provides primitives for the creation and manipulation of capability *directories*.

A directory contains a list of directory entries. Each entry consists of a text name, an associated capability, and a single bit specifying whether the entry is private or free. The private/free bit allows a user to share a directory without permitting access to all of the directory entries. Directory entries

are accessed by text name, and meta-instructions are provided to copy a directory capability to the user's C-list, place a C-list capability in a directory along with an associated name, or remove a directory entry. The directory meta-instructions—PLACE, ACQUIRE, REMOVE, DELETE, and LINK—are listed among the capability operations in Table 3-1.

Each user has a single root directory that contains capabilities for the user's permanent objects. When a user initiates a session (that is, when the user logs into the system), the supervisor creates a new process and places a capability for the root directory in the process's C-list, giving the process access to these objects. A process can then load capabilities from the root directory into the C-list by executing an ACQUIRE meta-instruction. The ACQUIRE specifies three parameters: the capability for the root directory, the text name of the object to be loaded, and the C-list location in which to place the associated capability.

New directories can be created and capabilities for directories can be stored in other directories. Thus, a user can build graph-structured directory mechanisms and share directories or subdirectories. To facilitate object sharing, the supervisor allows a process to obtain a capability for another process's root directory. In turn, the root directory can be traversed to locate subdirectories, and so on. However, when examining another user's directory structure, only those entries marked as free can be accessed.

The Dennis and Van Horn supervisor does not support a separate concept of files. Any segment or directory is potentially long-lived and can be used to store information from session to session or over system restarts. An object is maintained by the system as long as a capability exists for that object. Therefore, to make a segment or directory long-lived, a user simply stores a capability for that object in the root directory or any long-lived directory reachable through the root. The supervisor automatically deletes an object when the last capability for that object is deleted. Deleting any single capability for an object does not necessarily cause the object to be deleted because other capabilities for the object may still exist. The supervisor does support an explicit DELETE meta-instruction that can be used by a process with owner privileges to an object.

One of the most important aspects of the Dennis and Van Horn supervisor is its support for protected procedures. Within a multiprogramming system, it should be possible for a

user to create a procedure that provides service to many different users. However, this procedure must be able to protect local objects from its callers, and the callers may wish to guarantee that the procedure does not destroy or compromise any of their local objects. The protected procedure meets both of these needs.

A process creates a protected procedure by obtaining an *entry capability* through a supervisor meta-instruction. The entry capability contains a pointer to the C-list of the process that created it. It also contains an index, i, and a range, n, for a set of sequential procedure capabilities within the C-list of the creating process. The entry capability can then be passed to any process (through the directory mechanism, for example) and used to call any of the n procedures. To call a protected procedure, a process executes an ENTER meta-instruction specifying:

- an entry capability,
- the index of one of the n procedures to be called, and
- a capability parameter to be passed to the protected procedure.

The entry capability and capability parameter must be in the caller's C-list. As a result of the ENTER instruction, the supervisor creates a new process to execute the protected procedure. This new process executes in the sphere of protection *specified by the C-list pointer contained in the entry capability.* Figure 3-2 shows this change from the sphere of the caller to the sphere of the protected procedure. The entry capability in Figure 3-2 allows its owner to call one of two procedures defined by capabilities in the protected C-list.

A protected procedure, then, executes in the domain de-

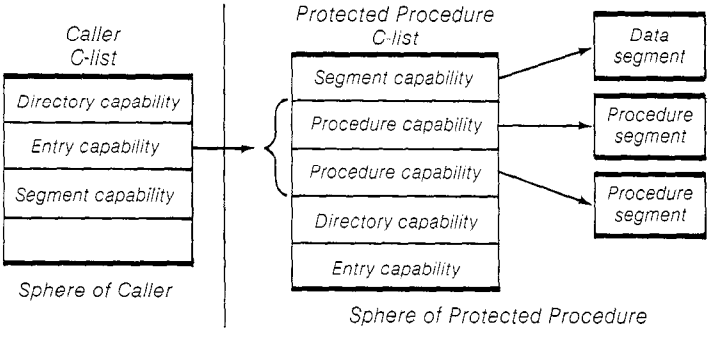

Figure 3-2. Protected Procedure Protection Spheres

fined by the procedure's creator, not in the domain of the caller. In this way, the caller and the protected procedure are mutually isolated. The caller has no access to the protected procedure's objects, and the procedure has no access to the caller's objects, with the exception of those objects passed explicitly through the capability parameter. Because this parameter can be a directory capability as well as a segment capability, a caller can pass a list of capabilities or an arbitrary data structure. A process possessing an entry capability can only use that capability to call one of a sequence of procedures. Once that procedure begins execution, it has access to all of the objects available in its private C-list.

The Dennis and Van Horn conceptual design became very influential on later systems. However, there are many ways to apply the concepts and many problems inherent in doing so. The first system to incorporate the concept of capability was a timesharing system at MIT, which is examined in the following section.

3.3 The MIT PDP-1 Timesharing System

The first computer system to include Dennis and Van Horn's capability operations was a timeshared operating system constructed at MIT from Dennis' design [Ackerman 67, MIT 71]. The system ran on a modified 12K-word Digital Equipment Corporation PDP-1 computer, the first minicomputer. The timesharing system supported five "typewriters" and used capabilities only to reference a few relatively high-level system resources, such as terminals, tapes, and drums. However, the operating system allowed users to extend this set of resources by creating new protected subsystems. It is the protected subsystem mechanism that is briefly examined here.

Each process running on the PDP-1 timesharing system has a C-list (also called the program reference list, after the Burroughs B5000), in which capabilities are held. The C-list is actually maintained in locations 0-77 of process address space. These locations are protected against program examination or modification and can only be manipulated by the operating system. Each capability is addressed by its index in the list.

Capabilities are created by special supervisor instructions. Each capability represents a resource object owned by the process. The supervisor supports a small number of resource types: I/O device, inferior process, file, directory, queue, and entry. When the process wishes to perform an operation on a

resource object, it *invokes* the object's capability through an INVOKE instruction. The INVOKE instruction specifies: (1) the C-list index of the capability to be invoked and (2) an operation to perform on the object represented by the capability. The INVOKE is similar to the ENTER instruction in the Dennis and Van Horn design.

Dennis and Van Horn's supervisor allows a process to create protected procedures that execute in private spheres of protection to protect local data from access by their callers. The PDP-1 system goes a step further. It allows creation of controlled subsystems that maintain different protected data objects on behalf of different processes, just as the operating system maintains files, for example, on behalf of different processes. To do this, the subsystem must be able to verify that a process is permitted access to an invoked object.

A subsystem is accessed through entry capabilities in the same way that protected procedures are accessed in the Dennis and Van Horn supervisor. To identify different subsystem resource objects, however, the PDP-1 system allows a subsystem to create different versions of its entry capabilities. The entry capabilities for a given subsystem are equivalent except for a *transmitted word* field that can be specified by the subsystem when the entry is created. In this way, the subsystem can maintain protected data structures on behalf of many processes. When a process calls the subsystem to create a new resource, the subsystem returns an entry capability with a transmitted word uniquely identifying that resource. Subsequently, when the user invokes an operation on that resource through the entry capability, the subsystem interrogates the transmitted word to determine which data structures to access. The transmitted word field is 6 bits in size, allowing a subsystem to support only 64 different objects; however, the PDP-1 supports a small user community.

The system was in operation for student use until the mid-1970s. It was distinguished not only by its capability supervisor but also by its space war game that ran on the PDP-1 video display. Following the MIT PDP-1 system, a major step in capability systems design took place at the University of Chicago. This work was significant because it used capabilities as a hardware protection mechanism.

3.4 The Chicago Magic Number Machine

In 1967 a group at the University of Chicago Institute for Computer Research began work on the Multicomputer, later

called the Chicago Magic Number Machine [Fabry 67, Shepherd 68, Yngve 68]. The goals of the project were ambitious: to provide a general-purpose computing resource for the Institute, to allow computer science research, and to interface to new peripheral devices. The project was perhaps too ambitious; in fact, the system was never completed. Nevertheless, the Chicago effort was the first attempt to build an integrated hardware/software capability system [Fabry 68]. The implementation of capability-based primary memory protection in this machine was to serve as a model for several early capability designs.

The Chicago machine provides a general register architecture and a segmented memory space. Memory is addressed through capabilities, and a process must possess a capability for any segment it addresses. Capabilities can be stored in registers or in memory; however, they cannot be mixed with data. Therefore, the machine supports two sets of registers—*data registers* and *capability registers,* and two types of segments—*data segments* and *capability segments.*

There are sixteen 16-bit, general-purpose data registers, three of which can be used as index registers. Capabilities are stored in six capability registers, each holding multiple 16-bit fields because capabilities are longer than the machine's 16-bit words. Several bits in each segment capability indicate whether the addressed segment contains data or capabilities. Hardware LOAD and STORE instructions allow programs to move capabilities between capability registers and capability segments, but programs are prohibited from performing data operations on capabilities. A process can have many capability segments, and capabilities can be copied freely between them.

For a program to access an element in a memory segment, the program must first load a capability for that segment into a capability register. The capability registers therefore act as a hardware C-list. A capability for a memory segment describes:

- the segment base *address,*
- the segment *length,*
- the *type* of the segment (data or capability),
- an *activity code,* indicating whether the segment is in primary memory or secondary store, and
- an *access code,* indicating how the segment may be used.

The access codes for data segments are read, read/execute, read/write, and read/write/execute; the access codes for capa-

bility segments are enter, enter/read, and enter/read/write. A program with capability read and capability write access to a capability segment can execute capability load and store operations on that segment, but cannot perform data operations on the capabilities. A user is never given data access to a capability segment, because that would allow the user to fabricate capabilities. However, the operating system supervisor may keep capabilities permitting data access to a user's capability segments. The supervisor uses these capabilities to perform meta-instructions that create a new capability or modify a capability.

To access an operand in primary memory, an instruction specifies a memory address using three components:

- a capability register containing a segment capability,
- a data register or literal value specifying the relative offset of a data element in the segment, and
- an optional index register containing an index that can be added to the supplied offset.

This allows, for example, addressing of an array that is located within a data segment. The hardware computes the sum of the two offsets and the base address contained in the capability to generate the primary memory address. It also verifies that the address lies within the segment, that the type of access is legal, and that the segment is in primary memory.

Segments can be created, extended, and destroyed by execution of supervisor meta-instructions, as shown in Table 3-2. A meta-instruction is also available to copy (snapshot) a segment onto secondary storage. The snapshot operation requires as a parameter the number of days the copy should be maintained. The current state of a segment and all backup copies are identified by the same capability, but the backups are differentiated by the time and date the copy was made. When a program retrieves a snapshot, the supervisor allocates a memory segment, copies the snapshot to that segment, and returns a new capability for that new segment to the user.

The Magic Number Machine is a multiprogramming system in which each process has as part of its state:

- a name,
- a capability for an account to be charged for its resource usage,

CREATE SEGMENT	create a new segment of given size and type and return a capability for it
CHANGE SEGMENT SIZE	increase or decrease segment size
DESTROY SEGMENT	delete segment
SNAPSHOT	copy current segment state to backing storage, marked with current time and date
RETRIEVE	copy specified snapshot from backing store into a new segment
CHANGE ACCESS CODE	produce a new version of a capability with reduced access rights
EXAMINE CAPABILITY	several meta-instructions to allow inspection of segment size, type, ID, access code, and activity code
CREATE PROCESS	create a subordinate process and return a process capability
MAIL	send capability and associated text name to specified user

Table 3-2: Chicago Magic Number Supervisor Capability Operations

- a capability for a base capability segment addressing the user's objects, and
- a capability for a mailbox.

Interprocess communication takes place between process mailboxes. A mailbox consists of a capability segment and an associated data segment. Using the MAIL meta-instruction, a process can send a capability and an associated informational text name to another process that can read, copy, or delete the information.

In addition to the hardware registers and the information listed above, each process has two segments associated with its context: a *process data segment* and *process capability segment*. Each of these segments has a fixed-sized storage region followed by a stack for data or capabilities. Two capability registers are reserved to address these segments, and two data registers act as stack pointers, although there are no explicit stack instructions (i.e., the registers must be manually updated).

A protected procedure mechanism in the Chicago Magic Number Machine allows for efficient one-way protection; that is, the procedure is protected from its caller but the caller is not protected from the procedure. Each protected procedure consists of at least one program segment and one capability segment, called the *linkage segment*, as shown in Figure 3-3. An

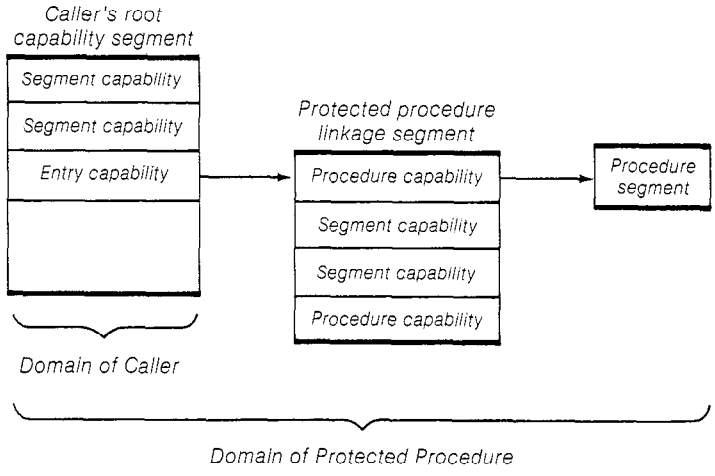

Figure 3-3: Chicago Magic Number Machine Linkage Segment

entry capability for the procedure points to the linkage segment, which contains capabilities for all objects needed by the procedure such as instruction segments, data segments, I/O operations, and so on. The first capability in the linkage segment points to the procedure entry point. Possession of an enter-only capability for the linkage segment allows the possessor to call the procedure using this first capability, but permits no other linkage segment access. Thus, the protected procedure can execute in a richer environment than its caller because it can access the entire linkage segment. Parameters can be passed either on the stack or in the registers.

Work on the Chicago Multicomputer/Magic Number Machine was eventually abandoned due to lack of funding. Although the project was never completed, the design was passed on to others including a group at Berkeley who incorporated some of its features into a new operating system, which is described next.

3.5 The CAL-TSS System

Started in the summer of 1968 at the University of California at Berkeley's computer center, the CAL-TSS project was an attempt to implement a general-purpose, capability-based operating system on conventional hardware. CAL-TSS was designed to supply timesharing services to several hundred users of a CDC 6400 computer system, thereby replacing

CDC's SCOPE operating system. Work on design and implementation continued until the fall of 1971, when it became clear that the system could not meet its goals in terms of service and performance. Funding was stopped and the project abandoned. Since then, its designers have published several appraisals of the project's successes and failures [Sturgis 74, Lampson 76].

The CAL-TSS operating system is a layered design in which each layer provides a virtual machine to the next higher layer. Each layer is specified as a set of objects and operations on those objects. This section examines the innermost layer of the supervisor which handles capabilities and object addressing.

The basic unit of protection in the CAL-TSS system is a *domain*, an environment containing hardware registers, primary memory, and a C-list. (A domain corresponds to the sphere of protection in the Dennis and Van Horn supervisor.) Access to objects outside the domain can occur only through *invocation* of a C-list capability; the possessor of a capability *invokes* an operation on the object it addresses by specifying the capability, the operation to be performed, and other optional parameters. A *process* is the execution entity of a domain, and its C-list may contain capabilities for other subordinate processes over which it exercises control.

Capabilities in the CAL-TSS system have three components:

- a *type* field that specifies the nature of the object addressed,
- an *option bits* field that indicates operations which can be performed by the possessor of the capability, and
- a *value* field that identifies the object and contains a pointer to the object.

Each capability occupies two 60-bit words in a C-list. A process has a root C-list and can create new second-level C-lists. When a process invokes a supervisor operation, it can specify capabilities stored in either the root C-list or any second-level C-list as parameters. A capability specification can therefore consist of two indices: one to locate a C-list capability in the root C-list and another for the target capability in a second-level C-list.

The CAL-TSS supervisor implements eight types of objects. A process can call supervisor operations to create and manipulate the following object types:

- kernel files (simple sequential byte streams),
- C-lists,

- event channels (interprocess communication channels),
- processes,
- allocation blocks (for accounting and resource control),
- labels (for naming short-lived objects and domains),
- capability-creating authorizations (user subsystems), and
- operations.

The last two supervisor-implemented types listed, capability-creating authorizations and operations, will be discussed later.

One important advance of CAL-TSS over its predecessors is in its physical object addressing. When the CAL-TSS supervisor creates a new object, it assigns that object a unique identifier. The identifier for that object is never reused, even after the object is destroyed. The use of unique identifiers solves a difficult system problem. If, for example, an object identifier could be reused after object deletion, the supervisor would have to guarantee that all capabilities for an object are destroyed before the object is destroyed. Otherwise, the remaining capabilities would be *dangling references*, that is, pointers to an object that does not exist. Were the supervisor to reuse the identifier later for a newly created object, such dangling references could be used inadvertently to modify the new object.

The CAL-TSS kernel provides a second level of indirection in addressing to greatly simplify relocation. Primary memory addressing of objects occurs through a single system table: the Master Object Table (MOT). The MOT is a kernel data structure that contains entries for every object in the system. Each MOT entry holds the unique object identifier and the primary memory address of one object's data. CAL-TSS capabilities do not contain primary memory addresses. Instead, a capability contains the unique identifier for the object it addresses and an index into the Master Object Table.

Figure 3-4 illustrates a C-list capability and the Master Object Table. The capability addresses a file object, as indicated by the type field shown symbolically as *"File."* The capability's value field contains the index of the MOT entry, M, which in turn contains the primary memory address of the file. All capabilities for the same file will contain the same MOT index. If the supervisor needs to relocate the file's primary memory segment, only a single MOT entry will have to be changed.

Both the capability and the MOT entry shown in Figure 3-4 contain the file object's unique identifier, IDx. The supervisor verifies that the identifiers in the capability and the MOT entry

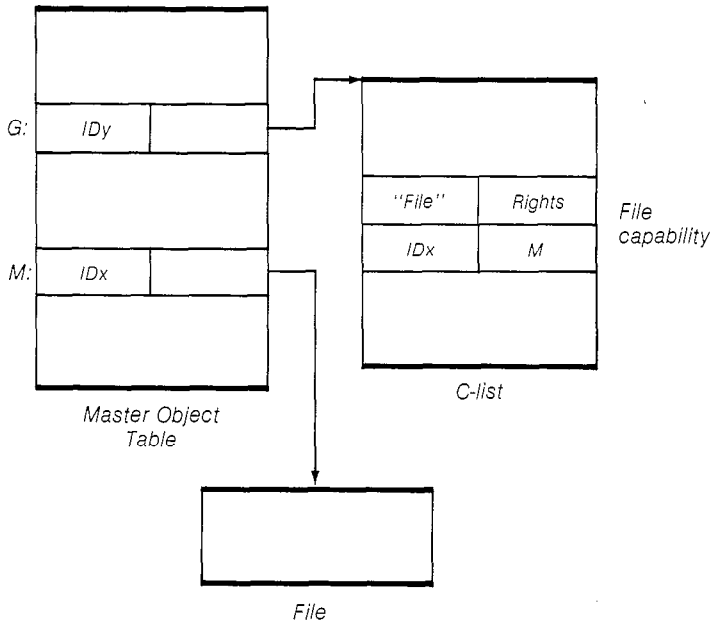

Figure 3-4: CAL-TSS Object Addressing

are identical for every operation invoked on the capability. When an object is deleted, the supervisor increments the identifier field of the object's MOT entry. Any subsequent attempt to use a capability for the deleted object (a dangling reference) would fail because the identifiers would not match.

Note that the C-list in Figure 3-4 is also a supervisor object and is addressed by the MOT entry at index *G*. The unique identifier for the C-list is *IDy*, an identifier that would be stored in any capabilities addressing the C-list.

The CAL-TSS system supports two object types that allow users to extend the small set of supervisor-implemented objects. A *capability-creating authorization* is an object permitting its possessor to create private capabilities for a private user-defined subsystem. Each user subsystem implements a single new type. To use this facility, a subsystem executes a supervisor meta-instruction to receive a capability for a capability-creating authorization object. The authorization object contains a new system-wide, *unique* type field. The subsystem can then present this capability to the supervisor, along with a 60-bit value, and obtain a new capability containing the subsystem's type and the specified value. The value inserted in the

capability corresponds to the transmitted word field that a subsystem can insert into capabilities on the MIT PDP-1 supervisor; it uniquely identifies an object implemented by the subsystem.

Such private capabilities receive the same protection as system capabilities, and can only be stored in C-lists and manipulated by kernel meta-instructions. Thus, a private capability can be passed to another domain to indicate ownership and rights to an object protected by the subsystem. For example, a user could implement a protected mail subsystem with the operations CREATE MAILBOX, DESTROY MAILBOX, READ MAIL, and WRITE MAIL. The subsystem would first obtain a capability-creating authorization containing a unique type field. Another domain calling the create mailbox operation would receive a capability containing the mailbox subsystem's type field and a unique value field to identify the newly created mailbox. The possessor of the capability could later present it to the mail system in order to read, write, or delete that mailbox, but could not modify the capability or directly access the mailbox representation. In this way, users can build subsystems that extend facilities provided by the base operating system.

A CAL-TSS *operation* is a supervisor-implemented object that allows the possessor to request a kernel or private meta-instruction; that is, to invoke a service. The operation object is a list describing the service to be performed, followed by specification of how the parameters are to be obtained. If the operation is for a private domain, that domain must be named along with an indication of the service requested. The parameter list specifies whether each parameter is: (1) data in the caller's memory, (2) a capability in the caller's C-list, (3) immediate data in the operation list itself, or (4) a fixed capability stored in the operation list.

The ability to contain immediate capabilities in the parameter list of an operation object is a powerful feature. It allows the called domain to receive a capability not available to the caller and thus is similar to the Chicago machine linkage segment. However, because the designers did not realize this advantage of operation objects until sometime after the system was constructed, the feature was never used.

When the CAL-TSS project was finally terminated in 1971, it had become clear that the system would never live up to expectations for either performance or functionality. There were many reasons for this, some being crucial design flaws.

One of the major design difficulties was the hardware base: a CDC 6400 with 32K 60-bit words of primary memory and 300K words of extended core storage (ECS). ECS is a memory device used as high-speed secondary storage. It is not used for execution, but data can be block-transferred between ECS and main storage at rates of several megawords per second. Management of ECS was one of the principal design problems. Equally troublesome was the 6400 memory management support, consisting of only a single base and limit register pair. Nevertheless, much was learned from the CAL-TSS project about the design choices available to capability system implementors.

3.6 Discussion

This chapter has examined early attempts to define and implement capability-based hardware and software systems. All of the systems described were designed in the late 1960s. These systems show one obvious relationship to the machines examined in the previous chapter: capabilities are descriptors used to address memory segments and other system objects. In a sense, the difference is merely one of terminology. The concept of capabilities and the C-list, as Dennis and Van Horn state, follows from the B5000's descriptors and Program Reference Table. However, there are some significant conceptual differences in the general way capability addressing is applied, in the lifetimes of capabilities and the objects addressed, and in the protected procedure mechanism that allows users to extend the functions of the operating system supervisor.

Capabilities are protected addresses; that is, a process can create new capabilities in its C-list only by calling a supervisor meta-instruction. Once a process receives a capability, it cannot directly modify the bits in the capability. The capabilities accessible to a process at any time define its sphere of protection or domain. All of the addresses (that is, capabilities) which a process can specify must either be contained in its domain at the time the process is created or be obtained through interaction with the kernel or other domains.

Because capabilities must be protected from user modification, these systems chose to isolate them within C-lists. C-lists are implemented as one or more segments that user processes cannot directly write with data instructions. Capabilities cannot be embedded in user data. This requirement is somewhat

restrictive because complex data structures that include pointers cannot always be naturally represented. The problem often can be circumvented by storing a C-list index in the data rather than the capability itself. However, storing a C-list index in place of a capability makes sharing data structures difficult if the processes do not share the same C-list. Another problem caused by the segregation of capabilities and data is the need for separate stacks and registers. Machines that support capabilities must have both data and capability stacks and data and capability registers. An alternative would be to support tagging, as in the BLM.

While the Dennis and Van Horn supervisor allows each process to have only one C-list, users of the Chicago Magic Number Machine and the CAL-TSS can store capabilities in multiple capability segments, chaining them together as desired to form complex tree or graph structures. The ability to construct additional C-lists allows fine-grained sharing of capabilities. Small C-lists can be created for sharing small collections of objects. The C-list addressing mechanism has a significant affect on the sharing of capabilities and the protection of objects. For example, if a procedure addresses its objects by C-list index, the procedure cannot be shared unless the sharing processes store the procedure's objects in the same locations in their respective C-lists. However, if a procedure executes with its own C-list, in which it places capabilities passed as parameters by its callers, this problem does not arise.

To compensate for the single C-list, Dennis and Van Horn allow capability directories for storage of capabilities and associated text names. The capability directory concept is a powerful extension of the directories provided by most operating systems. Even on most contemporary computers, directories can only be used to name files. In contrast, a capability directory allows the user to name and store many different object types. Directories can be shared between domains, and the Dennis and Van Horn system allows any user to obtain a capability for another user's root directory. A user can protect directory entries from external examination by setting a private bit associated with each entry. However, this mechanism in itself is insufficient for selective sharing among several users, because it is impossible to grant privileges to one user that are denied to another.

An additional method for exchanging capabilities between domains is the mail facility of the Chicago machine. Each domain has a local mailbox consisting of a capability and data

segment pair used to receive capabilities and symbolic capability names. Mailing a capability is equivalent to transferring a single directory entry between domains. It is unclear whether any additional information is placed in the mailbox, but some authentication information for the sender, either with the message or added by the mail system, probably should be required.

All of the systems examined support subordinate processes and process tree structures. A superior process is given complete control of an inferior that it creates. The superior defines the domain of the inferior by granting capabilities. It has the power to start, stop, modify the state of, generate simulated interrupts to, and service faults for the inferior. Mechanisms such as this allow users to build and test complex subsystems and to debug inferior processes. It may also be possible to simulate the kernel or hardware environment and, depending on the completeness of the mechanism, to debug kernel procedures.

Protected procedure mechanisms are available on all of these early systems. Dennis and Van Horn provide protected procedures through entry capabilities. The creator of the protected procedure obtains an entry and makes it public for users of the service. The protected procedure executes in a separate process in its creator's domain and receives a single capability parameter from its caller. The caller and callee are isolated from each other. In the CAL-TSS system, protected procedures also execute in a separate domain, with an operation object serving as the entry. The operation object specifies some number of data and/or capability parameters and methods to obtain them. The Chicago machine sacrifices two-way isolation for the improved performance of a one-way mechanism. A protected procedure on the Chicago machine executes in the domain of its caller and has access to its caller's objects. The protected procedure also has access to private capabilities contained in its linkage segment. Parameters are passed on the stack or in registers.

In addition to protected procedures, the MIT PDP-1 and CAL-TSS systems allow user processes to manufacture private capabilities. This type-extension mechanism allows user programs to extend kernel facilities in a uniform manner by creating new object types. User-created operations are invoked in the same way that supervisor meta-instructions are invoked.

The CAL-TSS capability-creating authorization and the MIT PDP-1 transmitted word facilities are *sealing* mecha-

nisms. A value is sealed in the capability that is not directly usable by the possessor of that capability. When passed back to the implementing subsystem, the subsystem—using a special capability it maintains—can *unseal* the value to determine which object the capability addresses. Sealing mechanisms are also provided by the Chicago machine's linkage segments and by CAL-TSS operations. In these systems, capabilities are sealed inside of special linkage segments. An entry capability for the linkage segment only allows its possessor to call procedures through specific entries in the segment. As a result of the CALL or ENTER instruction, the linkage segment is unsealed and its capabilities made available to the called procedure.

Perhaps the most important generalization of addressing provided by capabilities is support for long-lived objects. Capabilities allow uniform addressing of both short-term and long-term objects. Traditional computer systems require different addressing mechanisms for primary memory, secondary memory files, and supervisor-implemented objects. A capability can be used to address abstract objects of any type and any lifetime, implemented by either hardware or software. This advantage of capability systems raises a number of issues: how large must capabilities be to address the longer lifetime of objects, how can capabilities and objects be saved on secondary storage, what happens if capabilities or objects are deleted, how does the system know when an object can be deleted, and so on?

The Dennis and Van Horn supervisor allows objects to live an arbitrary length of time. An object exists until it is explicitly deleted or until all capabilities pointing to the object are removed. Thus, all objects are potentially long-lived, and the system must be capable of determining when the last capability for an object is deleted, or secondary storage will eventually become filled with garbage objects. Directories are used to keep track of long-term objects and their capabilities and to allow user reference to these objects by symbolic names. In the Chicago Magic Number Machine, snapshots are made of objects to force them to secondary storage. The objects can be retrieved later, although the issue of storing capabilities was not addressed by the design. When an object is retrieved from disk in the Chicago system, it is not retrieved as the same object but is placed in a new segment for which a new capability is generated.

One of the critical shortcomings of the CAL-TSS system was its failure to provide uniform addressing for permanent storage. The CAL-TSS system differentiated between user

objects, which could be saved on secondary storage, and kernel objects, which could not be saved on secondary storage. Moreover, because user objects were stored merely as byte streams, the CAL-TSS system could not save C-lists on disk while maintaining protection system integrity. The decision to support different object lifetimes, based on the belief that kernel objects were short-lived and would not require permanent storage, led to many quirks in the operating system.

Finally, one of the most important features in these systems was the physical implementation of addressing. Like earlier descriptor systems, the Chicago Magic Number Machine maintained hardware location information in the capability itself. This led to the relocation problems of descriptor systems; that is, relocation of a segment required a search for all capabilities addressing that segment. CAL-TSS took an important step by separating the capability from the addressing information, as recommended by Dennis and Van Horn. The physical relocation information is held in a central Master Object Table, and the capability contains a MOT index and a unique object identifier. Thus, relocation does not require a search for an object's capabilities. Deletion of an object also requires no search, because an attempt to use the capability for a deleted object will fail when the kernel checks the unique identifier in the MOT entry.

The Dennis and Van Horn supervisor defined the formal concepts of capability addressing. The MIT PDP-1 system, the Chicago Magic Number Machine, and the CAL-TSS system were the first trial implementations. The MIT timesharing system was in operation for several years, providing service to a small number of users, although capabilities were not a central part of the system's design. The Chicago and CAL-TSS systems were much more ambitious in terms of design, implementation, and goals. Perhaps the problem with these systems was the expectation that they would provide service to a large user community. In this sense, both systems failed, because neither was completed. However, when viewed as research projects, these early systems explored the crucial design issues and demonstrated both the advantages and difficulties of using an important new addressing technique.

3.7 For Further Reading

Dennis and Van Horn's publication paved the way for research in capability- and object-based systems [Dennis 66]. It provided the step from descriptors to more generalized ad-

dressing. It is difficult to tell how radical the fundamental concepts were when compared to systems like the Basic Language Machine, which was never completely described in the literature. Is it just a matter of terminology? This issue is discussed in Iliffe's letter to the Surveyors' Forum in the September 1977 issue of *ACM Computing Surveys* (Volume 9, Number 3) and in Dennis' response.

The Chicago and CAL-TSS efforts, while not resulting in finished products, did provide much insight about the design of capability systems. Fabry's paper [Fabry 74], based on his thesis [Fabry 68], is a detailed discussion of the advantages of capability addressing over traditional segmented addressing of primary memory. The paper by Lampson and Sturgis [Lampson 76], in addition to its technical description of CAL-TSS, provides an excellent discussion of the pitfalls of ambitious research projects.

The Plessey 250 computer. (Courtesy Plessey Telecommunications Ltd.)