5

The Cambridge CAP Computer

5.1 Introduction

In 1970, Roger Needham and Maurice Wilkes at Cambridge University began a research project to construct a capability-based machine. In contrast to the Chicago and Plessey designs, which included program-loadable capability registers, Needham and Wilkes' design made registers invisible to the programmer. That is, the machine contained a set of internal registers that the hardware would automatically load when a program specified a capability. Fortunately, the construction of this machine was simplified by several events that had occurred in the years since Wilkes' trip to observe the development of the Chicago Magic Number machine. First, it was possible to build reliable hardware from off-the-shelf TTL components. Second, and more important, it was possible for the computer to contain a reasonably large micro-control storage. The micro-control storage was used to implement the implicit loading of capabilities.

The result of the project, the CAP computer, has been operational at Cambridge since 1976. CAP (not an acronym) is a fully functional computer with an operating system, file system, compilers, and so on. The CAP system is the subject of many papers and a book [Wilkes 79], and the design decisions are the topic of Robin Walker's thesis [Walker 73].

5.2 Hardware Overview

The basic CAP CPU consists of a microprogramming control unit, 4K 16-bit words of micro-control storage, and an

arithmetic unit. The CPU contains a 64-entry capability unit that holds *evaluated* capabilities, that is, capabilities and the primary memory locations of the segments they address. These 64 capability unit entries are the registers implicitly loaded by the microprogram. The CAP CPU also contains a 2 x 256-entry cache and a 32-entry write buffer for performance enhancement. All CAP I/O, with the exception of a single control terminal and paper tape, is performed by an associated minicomputer.

CAP's memory is organized into segments up to 64K 32-bit words in size. A segment can contain data or capabilities, but not both. Although a process can address up to 4096 segments, an executing procedure can access a maximum of 16 capability segments at any time. A protected procedure mechanism allows different procedures to access different capability segments. The CAP system provides 16 general-purpose 32-bit registers, B0 through B15, for arithmetic and addressing; these registers cannot be used to hold capabilities. Register B15 contains the current instruction address; B0 is a read-only register that always contains zero. A single accumulator, capable of holding an 8-bit exponent and 64-bit mantissa, is available for floating point computation. In general, arithmetic functions operate on 32-bit integer or floating point values.

CAP's instruction set includes over 200 instructions. Both binary and floating point arithmetic are supported, as well as a variety of logical and control instructions, and a small set of capability manipulation instructions.

5.3 CAP Process Structure

A process is the basic execution and protection entity in the CAP system. A process is defined by a set of data structures that describe a collection of accessible segments and other resources. CAP objects are addressed through capabilities contained within a process's capability segments. Each executing procedure in the CAP system operates within the context of a process.

Like previous capability-based designs, the CAP system provides a process tree structure, as shown in Figure 5-1. The process structure is supported by an instruction that creates subprocesses and an instruction that requests service from a parent process. At the root of the tree is a process called the *Master Coordinator*. The Master Coordinator controls all system hardware resources, which it allocates among level-2 user

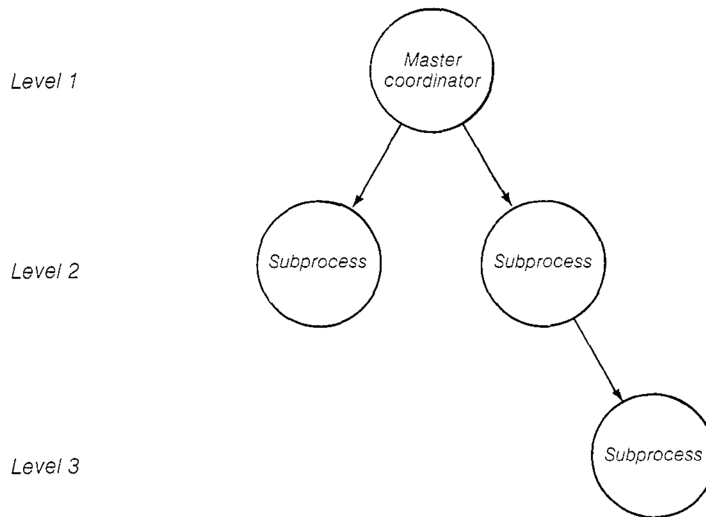


Figure 5-1: CAP Process Hierarchy

processes. Each level-2 user process can, in turn, create further subprocesses, acting as a coordinator for them.

CAP's designers chose to use the process tree mechanism to eliminate the need for a privileged mode of operation. Each CAP process can control the addressing environment and execution of its subprocesses without special privilege or operating system intervention. The desire to provide a very general process tree structure led to a design that closely linked addressing to process structure. This facility was probably overemphasized in the design and only two levels are actually used: the Master Coordinator at level 1 and the user processes at level 2.

5.4 CAP Addressing Overview

A high-level view of CAP addressing is useful before delving into the detailed mechanism. As mentioned, addressing and process structure are intimately related on the CAP system. Figure 5-2 shows the addressing relationship between a process and its subprocess. Two objects of interest are pictured for each process: a capability segment and a data structure called the *Process Resource List* (PRL).

On CAP, a process must possess a capability for any object to be accessed. Capabilities are stored in capability segments. In contrast to the Plessey and CAL-TSS designs, in which capabilities refer to entries in a system-wide table, capabilities

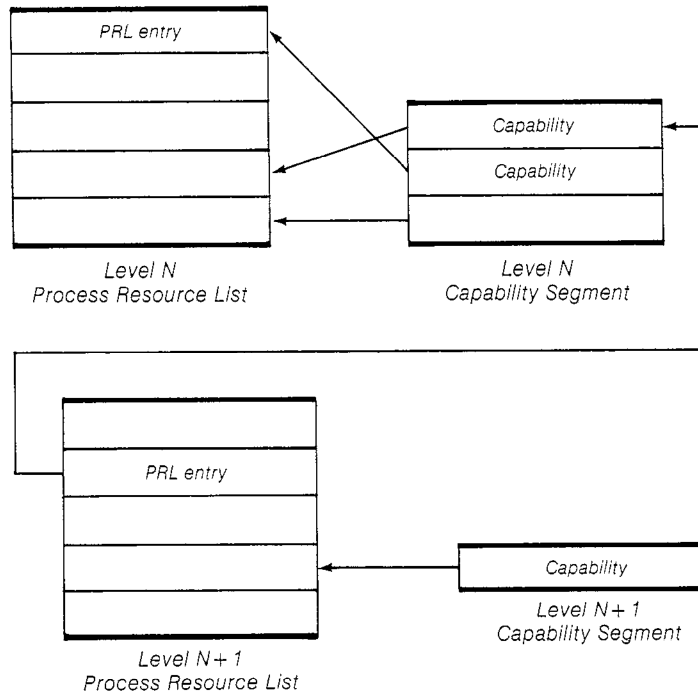


Figure 5-2: CAP Process Addressing

on CAP refer to entries in a process-local table, the Process Resource List. The Process Resource List differs from previous schemes in another important way. PRL entries do not contain primary memory addresses, but instead refer to capabilities *in capability segments of the parent process*. This upward indirection is shown in Figure 5-2 by the arrow leading from the level N+1 PRL entry to the level N capability segment. Indirection continues from there to the level N Process Resource List, and so on, until the Master Coordinator is reached at the top of the tree. The Master Coordinator's Process Resource List contains the primary memory address for each segment.

The following sections describe this addressing structure in more detail, but the reason for the extra indirection is worth noting here: it provides a process with the freedom to control its subprocesses. In the CAP system, a process can directly write the PRL and capability segments of its subprocesses. In this way, a process can dynamically control the addressing environment of its inferiors without operating system interven-

tion. Permitting a process data access to its subprocesses' capability segments does not violate the protection system because of the indirection in addressing. Ultimately, all capabilities and PRL entries in a subprocess must refer to valid capabilities held by its parent process. Therefore, although a parent process can create capabilities for its offspring, these capabilities can only address objects that are accessible to the parent.

5.5 Capabilities and Virtual Addresses

Within a CAP process, an executing procedure addresses segments through capabilities stored in its capability segments. Capabilities can be specified by CAP instructions and manipulated in controlled ways by user programs. Figure 5-3 shows the CAP capability format. As described above, each capability refers to one entry in the Process Resource List. Each capability also contains a type field in the two high-order bits that differentiates segment capabilities, enter capabilities, and so on. The bits marked *W* and *U* are set by hardware to indicate that a segment has been written or accessed, respectively.

The encoding of the access field is shown also in Figure 5-3. CAP permits read and/or write access to a capability segment, or read, write, and/or execute access to a data segment. Write capability access permits a process to execute instructions to move capabilities to a segment; it does not allow data operations on the segment. The base, size, and access fields in a capability can be used to *refine* access to a segment defined by a PRL entry. For example, a program can create a new capability with read-only access to a segment for which the PRL permits read/write access. Or, using base and size, a capability can be refined to address only a contiguous subset of a segment. The REFINE instruction performs these operations.

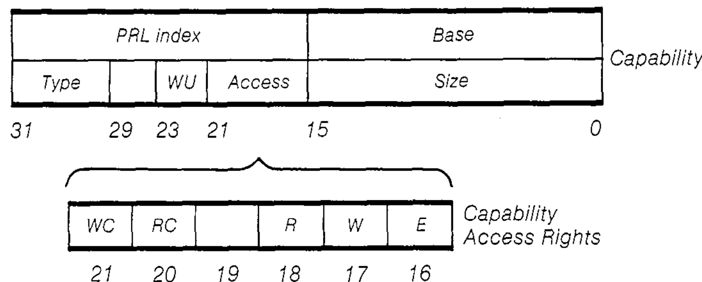


Figure 5-3: CAP Capability and Access Rights Formats

To reference a word in memory, the CAP programmer must specify a capability for a segment and the offset of the word within that segment. The capability is specified by an index in one of the 16 capability segments. A complete CAP virtual address, then, consists of three parts: a capability segment number, a capability index, and an offset into the selected segment.

Figure 5-4 shows the format of a CAP virtual address when stored in memory or a general register. The upper 16 bits of the address are known as the *segment specifier* because they select a capability for the addressed segment. The segment specifier consists of two values: I, the number of one of the 16 capability segments, and F, the index of a capability within that segment. The capability selected in Figure 5-4 contains the index of PRL entry M, which points to a data segment (although the addressing is indirect). The value K in the virtual address is the offset of the target word in this data segment.

Note that each capability segment can hold a maximum of 256 capabilities because the capability index field in Figure 5-4

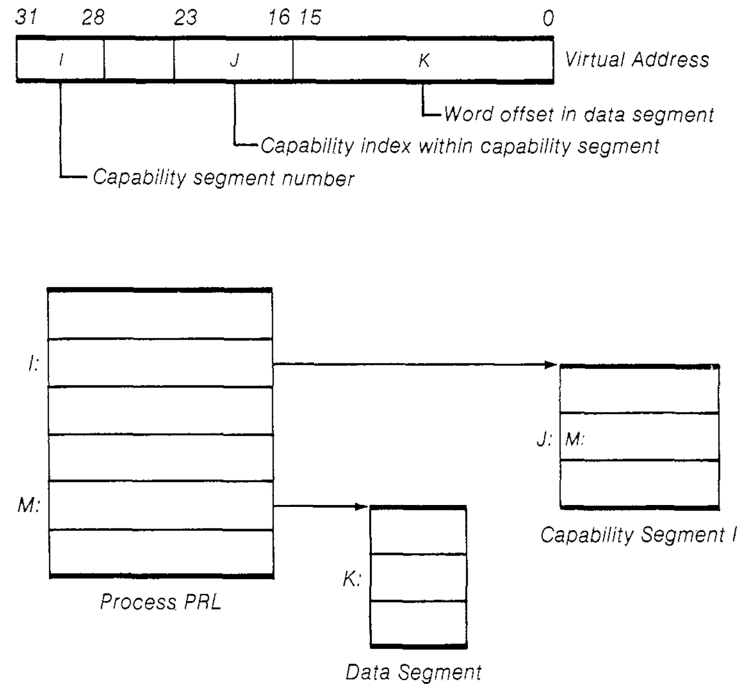


Figure 5-4: CAP Virtual Address

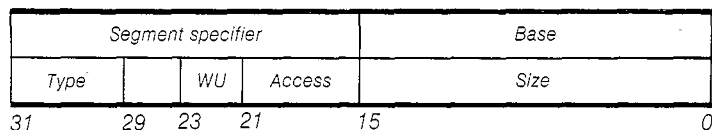
is 8 bits long. There are 16 capability segments, so the process can address a maximum of 4096 capabilities at a given time.

5.6 Process Data Structures

A CAP Process Resource List defines all of the resources available to a CAP process. Figure 5-5 shows the structure of entries in a PRL. A PRL entry is identical in format to a capability, except that the PRL index of the capability is replaced by the segment specifier field. The segment specifier selects a capability in one of the capability segments of the parent process. Just as the base, size, and access fields in a capability can be used to refine the access permitted by a PRL entry, these fields in the PRL entry can be used to refine the access permitted by the parent's capability.

PRL entries resemble capabilities in structure; however, the PRL is not a C-list and differs from a C-list in two important ways. First, PRL entries cannot be manipulated by programs executing within the process. Second, the PRL must contain entries for objects needed by *all* procedures that the process executes. In contrast, most capability systems allow procedures to access private objects not available to the C-list of their caller. Different procedures executing within a CAP process can be restricted to different capability segments and, hence, to different objects; but all of the objects that they collectively address must have entries in the PRL.

In addition to the PRL, each process has a data structure called the *Process Base*, which contains the state of the process. By convention, the first entry in the PRL addresses the Process Base. The first 16 words of the Process Base define the 16 process capability segments by indicating the offset of the PRL entry for each segment, as shown in Figure 5-6. The V bit in each word specifies whether or not that capability segment exists, and the 8-bit offset field indicates which PRL entry



Segment specifier<31:28> = Parent capability segment
Segment specifier<23:16> = Index of capability within specified parent segment

Figure 5-5: CAP PRL Entry

0	*V*	*0*	*Offset*
. . .	*Capability segment pointers*		
15	*V*	*0*	*Offset*
16	*B0*		
. . .	*General registers*		
31	*B15 (PC)*		
32	*Microprogram storage/state*		
33	*Accumulator high half*		
34	*Accumulator low half*		
35	*Count-down timer*		
36	*EC instruction information*		
37	*Microprogram register*		
38	*Unused*		
39	*Wakeup waiting switch*		
40	*C-stack next entry*		
41	*C-stack current frame*		
	Used by software		

Figure 5-6: CAP Process Base

addresses the corresponding capability segment. All capability segments accessible to a process must, therefore, be addressed through the first 256 PRL entries. The remaining words in the Process Base contain copies of the general registers, a count-down timer, and pointers to the C-stack—a data structure used to save capabilities during procedure invocation.

5.7 Memory Address Evaluation

This section reviews the translation process from virtual address to primary memory location. Because each process

owns all segments available to its children, the Master Coordinator at the root of the tree must have capabilities for all segments in the system. In fact, the Master Coordinator is the only process that addresses memory directly. In the PRL of the Master Coordinator, called the *Master Resource List* (MRL), are capabilities similar in format to that shown in Figure 5-5; however, word 0 of these MRL entries contains a memory address in the low-order 20 bits. All capabilities ultimately refer to these MRL entries.

The steps to translating an address are as follows:

1. Locate the specified capability segment in the process, and select the capability in the index contained in the virtual address.
2. Follow the capability link to the entry in the process PRL. Minimize access rights through a logical AND operation, and compute new base and length if required.
3. From the PRL entry, locate a capability in the parent process's capability segment. Once again, apply rights, base, and length minimization.
4. Follow this capability back to the entry in the parent's PRL.
5. Continue this process until the MRL is reached, at which time the physical address can be calculated. Check the offset supplied in the original general address for legality and make the requested reference.

Certain facts are apparent about this mechanism. First, several levels of indirection, and hence, several memory references, are required before an actual operand can be accessed. This problem can be handled with the special hardware that the CAP provides. Second, because capabilities refer to a process-local structure, the PRL, they cannot easily be transferred between processes even at the same level of the hierarchy. Capabilities cannot be copied between processes unless both processes have identical PRLs. Third, capabilities cannot be copied directly from parent to child, but must be passed by constructing PRL entries and corresponding capabilities in the child that refer to the parent capability. Fourth, because of the indirection in both capabilities and PRL entries, a process is totally free to create capability segments and PRL entries for its subprocesses.

5.8 Subprocess Creation

Any CAP process is capable of creating subprocesses to which it can pass access rights to various objects. The creation

of a subprocess is accomplished by the ENTER SUBPROCESS (ESP) instruction. One operand of the ESP instruction is a segment that will become the PRL of the new subprocess. Another operand is the index of the PRL entry in that segment for the subprocess's Process Base.

A parent process creates a subprocess PRL by allocating a data segment and constructing PRL entries that refer to the parent's capabilities. Because of the way PRL addressing is implemented, the construction of subprocess PRL entries requires no special privilege. It is impossible for the parent to construct a PRL capability for its offspring that allows it to address an object not addressable by the parent. Since the access rights are minimized at each level during the address evaluation, it is also impossible to increase access rights to an addressable object.

The ESP instruction allows any process to create a subprocess, to define the resources of the subprocess, and to protect itself from the subprocess. Each parent can also service requests from its subprocesses. The subprocess issues an ENTER COORDINATOR (EC) instruction, specifying a code for the operation to be performed. Execution of the EC instruction causes resumption of the parent process at the instruction following the ESP that initiated the subprocess. The code is placed in a general register specified by the original ESP.

Multiprogramming on the CAP system is implemented by using the countdown timer stored in each Process Base. When an ESP instruction is executed, control passes to the subprocess. The subprocess continues execution until either its timer expires or it executes an EC instruction, causing return of control to the parent. The parent process can service the EC or timer expiration, resuming the interrupted process or another subprocess if it likes. The parent might also request service from its own parent via an EC instruction. Before resuming a subprocess by ESP, the parent resets the countdown timer in the process base of the subprocess.

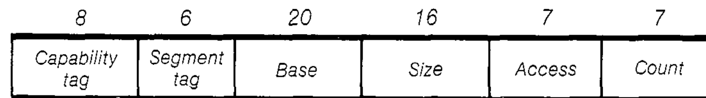
Thus, any process can coordinate the execution of its subprocesses, relinquishing its own allotted processor time for each subprocess to run. In fact, the current process is allowed to run because a set of processes, rooted in the Master Coordinator and terminating with the current process, have each relinquished processor time via ESP. Each process in the list is at a different level of the process tree, and each executes under a time limit specified by its parent. The CAP hardware must, therefore, maintain timers for each level of the process tree

because a timer could expire at any level, thereby returning control to the parent of the expiring process.

5.9 The Capability Unit

The CAP capability unit contains storage elements used by the microprogram to enhance system performance. The storage elements include 64 capability registers and 16 tag memory registers, whose use will be described in this section. The principal function of the capability unit is to reduce the effect of CAP's multiple levels of indirection. The capability unit acts as a cache memory (or what is commonly called a translation buffer) for storing recently used segment virtual addresses and their corresponding segment physical addressing information.

Figure 5-7 shows the structure of a capability unit capability register. Each capability register contains information about a segment capability. The base, size, and access fields are used to compute the primary memory address and to validate the attempted memory access. Two tag fields uniquely identify the capability within the capability unit; the segment tag identifies the capability segment that holds the capability, and the capability tag contains the capability's index within that segment. The segment tag is the number of another capability register in the capability unit. Each capability is contained in one of 16 capability segments, and to load a capability into a register, the capability for its capability segment must also be loaded in a register. The number of that register is used as the segment tag field.



Capability tag	*Contains the index of this capability within its capability segment.*
Segment tag	*Identifies the segment containing the capability.*
Base	*Contains the primary memory address of the segment.*
Size	*Contains the size of the segment in words.*
Access	*Indicates the permitted segment access rights.*
Count	*Contains a count of the number of references to the capability from within the capability unit.*

Figure 5-7: Capability Unit Register Format

When a program attempts to access a virtual address, the microprogram loads that address into the virtual address register of the capability unit, as shown in Figure 5-8. The capability unit then autonomously attempts to locate the capability register containing the physical attributes of the segment addressed. If the capability is found, the capability unit validates the requested access and performs the primary memory request. If the capability is not found, the capability unit notifies the microprogram, which must then load the needed information into a capability register.

The capability register search uses one of the 16-tag memory registers shown in Figure 5-8. Each of the 16-tag memory registers corresponds to one of the 16-process capability segments. Whenever the microprogram loads a capability for capability segment I into a register, it also loads the number of that register into the corresponding tag memory register. Therefore, tag memory register I specifies the location of the capability for capability segment I in the capability unit. A valid bit in each tag memory register indicates whether or not that register has been loaded.

From the virtual address presented to the capability unit, the unit selects one tag memory register based on the capability segment specifier (the upper 4 bits). The capability unit then uses the tag memory register in an associative search. The capability unit searches for a capability register whose segment tag field matches the contents of the tag memory register. If the tag fields match, then the register contains a capability that is stored in the correct capability segment. The unit must then check the capability index field in the virtual address, shown as J is Figure 5-8, with the capability tag field in the register. If these fields match, the correct segment register has been found. If the J fields do not match, the search continues. The capability unit is able to examine four capability registers at a time during the search.

5.10 Protected Procedures

The protected procedure is the principal CAP protection mechanism. Although other capability systems execute protected procedures in a new process, all procedures called from within a CAP process execute within that same process. However, different procedures may have access to different capability segments and, hence, to different objects. The protected procedure mechanism causes switching of capability segments and, therefore, changes the access domain of a procedure.

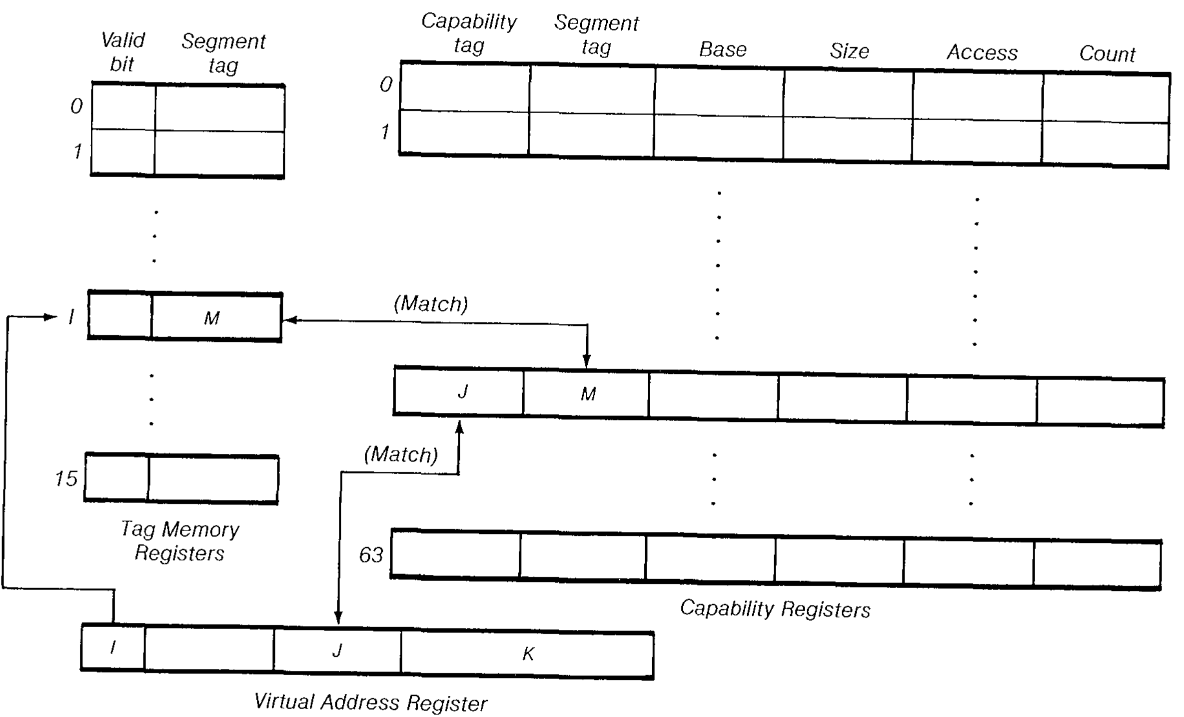


Figure 5-8: CAP Capability Unit

Protected procedures are used extensively both within the CAP operating system and by user programs. All operating system services are programmed as protected procedures, and all compilers output protected procedures. The use of protected procedures to perform system functions is particularly important within the CAP system. Although services could be provided through ENTER COORDINATOR instructions to the Master Coordinator, such instructions would cause a serialization of service. That is, once the Master Coordinator is entered, the service routine would have to complete before another process could execute. By placing operating system services within protected procedures available to every process, several processes can execute service routines simultaneously.

A protected procedure can be called only through an *enter capability* which the caller must possess. Figure 5-9 shows an enter capability and the PRL entry to which it refers. The execution of a protected procedure call causes 5 of the 16 capability segments to be changed. These new capability segments form part of the new domain in which the protected procedure executes. The enter PRL entry shown in Figure 5-9 contains fields that define three of the new capability segments. The creator of a protected procedure is free to use these segments in any way; however, the conventional name and use of the new capability segments are as follows:

A The *argument* capability segment contains capabilities passed as parameters to the currently executing procedure.
N The *new argument* segment is used to construct an argument list for a procedure to be called. This segment becomes the A segment of the called procedure.

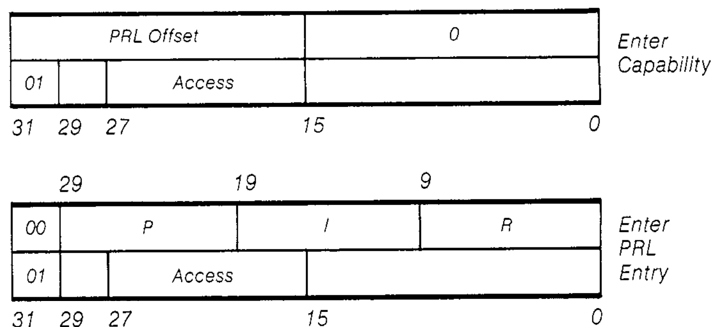


Figure 5-9: CAP Enter Capability and Enter PRL Formats

P The *procedure* segment contains capabilities for code and data segments that are shared by all processes executing a protected procedure.

I The *interface* segment contains capabilities that are used by the procedure but are specific to the executing process, for example, a process-local workspace.

R The *resource* segment contains capabilities specific to one instance of the protected procedure. For example, the R segment might be used to address the representation of an object managed by a protected type manager. The representation would be accessible only to the protected procedure.

A program executes an ENTER instruction to call the protected procedure. The single operand to the ENTER instruction is the location of the enter capability. Parameters are passed in the N segment. The ENTER instruction then changes the execution environment, using a data structure called the *C-stack* to save information about the current procedure. The C-stack is a segment in which the invocation stack (the procedure-calling record) is maintained. Each procedure call causes the hardware to place a new invocation frame on the C-stack by updating the C-stack pointers in the Process Base. The RETURN instruction restores information placed on the C-stack, removing the current frame and returning control to the caller.

In more detail, the ENTER instruction causes the following events to occur:

- A new C-stack frame is allocated. This 6-word frame is loaded with procedure state information, including the PRL indices for the current P, I, and R segments.
- The PRL indices for the new P, I, and R segments, stored in the enter PRL entry, are used to modify the three words in the Process Base that address these three capability segments.
- The PRL index for the current A segment is saved on the C-stack. The A segment slot in the Process Base is loaded with the PRL index of the current N segment. The Process Base slot for the N segment is invalidated.
- The current program counter (B15) is saved on the C-stack.
- The access rights specified by the enter capability and the enter PRL entry are ANDed and placed in B14, for examination by the procedure.
- The program counter is loaded with the address of the first word of the segment addressed by the new P capability.

The protected procedure begins execution at the first word of the P segment. It executes in the new domain created by the

ENTER instruction and has access to new A, P, I, and R segments. When the procedure is entered, no N segment exists. Should the procedure wish to create a new argument segment for a further procedure call, it executes a MAKEIND instruction to specify the length of the new N segment. The N segment is also allocated on the C-stack. Execution of a RETURN instruction destroys the N segment and replaces the previous P, I, R, and A segments.

Each CAP user program is, in fact, a protected procedure, and is restricted to a subset of the objects addressed by its process' PRL. This subset is defined by the P, I, and R capability segments made available to the program by its enter capability. Other procedures callable by the program can have access to different segments. The enter PRL entry for a protected procedure *seals* three capabilities, making them available to the protected procedure when it is called.

The protected procedure mechanism supports the creation of protected objects and object type managers. For example, Figure 5-10 shows the implementation of a subsystem supporting protected objects of type *message port*. Each instance of a port object is represented by a new instance of the port pro-

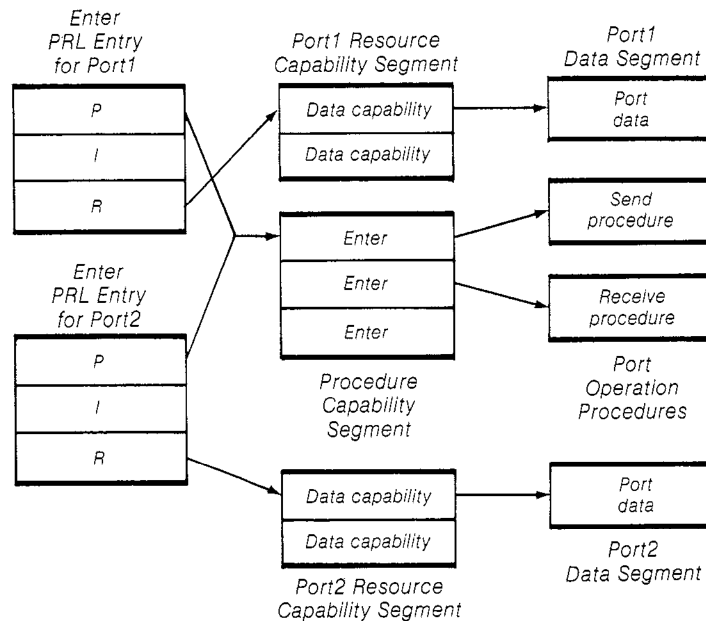


Figure 5-10: CAP Protected Object Implementation

tected procedures. Each instance of the port system contains a pointer to the port protected procedures and a pointer to the segments containing the data structures for one port instance. Figure 5-10 shows enter PRL entries for two ports. Both PRL entries address the same P segment and share the procedures that operate on the ports, but every object has a different R segment that contains the representation of that object instance.

To create a new object, then, the type manager creates an instance of itself with a new R segment. All PRL entries for objects of the same type share a P capability but have different R capabilities for the segments containing the object's representation. Processes are given enter capabilities that address these PRL entries. The type manager defines and interprets the access rights in its enter capabilities. The ENTER instruction makes those access rights easily accessible by placing them in a register.

5.11 Long-Term Storage and Long-Term Names

Like the Plessey 250, the CAP operating system provides for long-term storage of objects. Three types of objects can be preserved on secondary storage: segments, directories, and Procedure Description Blocks. A Procedure Description Block is a segment created by the operating system that defines how a protected procedure should be constructed, including its segments and the capabilities in those segments.

CAP capabilities, like Plessey 250 capabilities, contain the index of a data structure in memory (the PRL). This index is a short-term identifier for an object and is meaningful on the CAP system only during the lifetime of a single process. Therefore, in order to preserve and name objects with a long lifetime, each object must have a unique long-term name. When object names are saved on secondary memory, they must be stored as long-term names.

Each CAP object's long-term name is unique for the life of that object. The long-term name is called the *system internal name* of the object. An object's system internal name is constructed from the disk block address where the object is stored. The CAP operating system maintains a list of all long-term objects that includes the number of references to each object on secondary storage. In addition, the operating system maintains a list for each CAP process that contains the system internal names for all objects addressed by that process's PRL.

Every CAP user has one or more directories in which to store text names of long-term objects and their associated system internal names. Directories are managed by a protected procedure known as the directory manager.

The operating system maintains the storage for an object as long as a reference to that object exists in a directory, in a Procedure Description Block, or in the PRL of an executing process. When a process requests an object from a directory, the system first checks the process-local system internal name list to see if that object is currently in memory. If so, the process will already have a PRL entry addressing the object and a capability can be constructed. Otherwise, the system's long-term system internal name list must be consulted and the object fetched from secondary storage. This operation will cause a PRL entry to be allocated, a capability to be constructed, and a notation to be made in the process-local system internal name list.

Protected procedures are stored on secondary memory as Procedure Description Blocks. A protected procedure, as previously described, consists of three capability segments (procedure, interface, and resource) that are made available as the result of an ENTER instruction. These segments contain capabilities that are used by the protected procedure but may be hidden from other process procedures.

When a protected procedure is created, the operating system constructs a Procedure Description Block containing system internal names of the objects accessible to the protected procedure. The operating system returns an enter capability and places an enter PRL entry in the Process Resource List of the creating process. The PRL entry is constructed so that a trap will occur if an ENTER instruction attempts to use that entry. If a trap occurs, the operating system builds the P, I, and R capability segments from the system internal names in the Procedure Description Block. In this way, such segments do not need to be allocated unless the procedure is actually called.

5.12 Discussion

The Cambridge CAP computer is the first successful university-built hardware and software capability system. Unlike previous university efforts, the CAP implementors completed a system that serves both as a research tool and as a useful service facility. The CAP system is interesting because of sev-

eral design aspects, including the addressing structure and the use of the microprogram and capability unit for implicit capability loading.

The most influential decision made in CAP's design was the choice of a capability protection system based on a process hierarchy. The goal was to allow any process complete freedom to supervise the activities of its subprocesses. The CAP system permits a process to control the processor scheduling as well as the memory resources of its offspring. The ENTER SUBPROCESS and ENTER COORDINATOR instructions operate at any level of the tree, allowing any process to act as a complete coordinator.

CAP's addressing structure permits direct control of subprocess addressing domains by a parent process. In contrast, a parent process on other capability systems must call a supervisor service to place a capability in a subprocess's C-list. On CAP, however, a process can have data access to its subprocesses' capability segments. No protection violation occurs because of the indirection in subprocess capabilities, although this indirection reduces the efficiency of capability addressing.

An additional problem is caused by the local nature of the Process Resource List. Because all capabilities address the PRL, a process-local structure, they cannot be passed easily between processes. CAP capabilities are different from capabilities on previous systems because they do not contain a *global context-independent* identifier. Although each CAP object has a system-wide unique name, a CAP capability contains a PRL index which is a process-local object name.

Following their initial experience, CAP's designers felt that the process tree had been much overemphasized in the design. The generality of a multi-level process structure, while providing conceptual advantages, led to performance and implementation difficulties. Therefore, only two levels of process structure are actually used in the CAP—the Master Coordinator and the level-2 processes. However, the effect of the process tree design on addressing remains.

A more essential CAP mechanism is the protected procedure. Protected procedures are widely used, both within the operating system and by user programs. Most of CAP's operating system is implemented as protected procedures that execute within the domain of each process; this alleviates the bottleneck that would be caused if all services were performed directly by the Master Coordinator.

Protected procedures are also useful for implementing type managers and protected objects. The procedure (P) segment

for the protected procedure specifies the protected object management routines, while the resource (R) segment can be used to specify the representation of a single object instance managed by those routines. When a new object instance is created, the type manager creates a new instance of its protected procedure system. This new instance is represented by a new enter capability and enter PRL entry that have access to a new R segment.

Although the protected procedure mechanism supports the creation of protected objects, it is not extensively used for that purpose within the operating system due to the cost of protected procedures. Using this mechanism for protected objects, a new instance of the type manager (that is, a new protected procedure with its enter PRL entry) must be created for every new object. Creation of a new instance of a protected procedure also causes creation of a new Procedure Description Block, which involves both space and time overhead to the system.

A less expensive mechanism is provided by *software capabilities* (not described in the chapter). The operating system uses software capabilities for addressing operating system objects. Software capabilities can be placed in process capability segments and are protected in the same way that segment capabilities are protected. The type field in the capability indicates whether it is a software capability or another type of capability. A protected procedure can return a software capability to a process as proof of object ownership. The bits in a software capability can be defined by the protected procedure and used in any way. However, software capabilities can only be used by operating system protected procedures because they rely on convention to distinguish the type of object addressed by the software capability.

CAP's capability unit serves to reduce the overhead references required for address translation. A memory reference in a level-2 user process requires four overhead references before the word is accessed, because two capabilities and two PRL entries must be read to compute the primary memory address. The capability unit reduces this overhead by caching frequently used segment capabilities and their segment primary memory addresses.

Additionally, the use of tag memory registers and the structure of the capability register tags permit registers to remain loaded over domain changes. That is, when a context switch or protected procedure call occurs, only the tag memory registers

need to be changed. A call to a short protected procedure will not cause a turnover of registers in the capability store. However, the capability unit requires that a large number of evaluated capabilities be loaded in registers before it can operate. For example, for each process capability in the capability unit, the unit must also hold evaluated capabilities for the segment containing the capability, for the process PRL and Process Base of the current process, and for the PRL and Process Base of the parent process. The overhead is significant, and the 64-register size of the store would make large process trees impractical.

Additional overhead always exists in capability management, and this can be seen in light of the CAP addressing structure. Because capabilities are defined indirectly, a parent has the ability to modify or invalidate a capability to which a junior process refers. Using this mechanism, it is possible to revoke authority to an object previously allowed a subprocess (and potentially, its juniors). Since the capability unit maintains translated copies of capabilities, however, it is possible for a change at a higher level in the process tree to be made while a lower level capability exists in the capability unit along with its physical address. Therefore, each time a capability in memory is modified, the capability unit must ensure that no junior process capabilities are left in the unit that refer indirectly to the modified capability. Although this is analogous to the operation required on a virtual memory translation buffer in any virtual memory system, the operation is more frequent with capabilities because, while users can modify capabilities, only the operating system can modify process page registers.

The CAP project has been successful for reasons related both to the structure of the hardware and the amount of useful software available to its users. Since it became operational, the CAP system has continued to be a useful research and computation facility at Cambridge University, and the base hardware has proven flexible enough to allow further experimentation with capability architecture [Herbert 78a].

5.13 For Further Reading

Much literature is available on the CAP system and its software. A general discussion of capability addressing and the CAP approach can be found in [Needham 72 and Needham 74]. The best overview of the CAP system is provided in the paper by Needham and Walker [Needham 77a], the book by

Wilkes and Needham [Wilkes 79], and the thesis by Walker [Walker 73]. The book describes the operating and filing systems as well as the hardware. The filing system is described also in [Needham 77b, Birrell 78]. Performance evaluations of the CAP system can be found in the papers by Cook [Cook 78, Cook 78b].

Since the original CAP design, Herbert has experimented with a new CAP capability architecture implemented by a microprogrammed kernel running on the CAP hardware [Herbert 78a, Herbert 78b, Herbert 79]. A version of [Herbert 79] is reprinted in [Wilkes 79]. Herbert's kernel corrects some of CAP's problems and supports global naming and a form of sealing as described by Redell [Redell 74a].

The Hydra/C.mmp computer. (Courtesy William Wulf.)