# 6

# The Hydra System

## 6.1 Introduction

This chapter marks the transition from capability-based to object-based computer systems. Although somewhat subtle, the distinction is one of philosophy. The systems previously described are primarily concerned with capabilities for memory addressing and protection, although they support abstraction and extension of operating system resources as well. The principal concern of the systems discussed in the remaining chapters is the use of data abstraction in the design and construction of complex systems. In these systems, *abstract objects* are the fundamental units of computing. Each system is viewed as a collection of logical and physical resource objects. Users can uniformly extend the system by adding new types of resources and procedures that manipulate those resources.

The subject of this chapter is Hydra, an object-based operating system built at Carnegie-Mellon University. Hydra runs on C.mmp (Computer.multi-mini-processor), a multiprocessor hardware system developed at Carnegie. Hydra is significant because of its design philosophy and the flexibility it provides for users to extend the base system. This flexibility is supported by capability-based addressing.

## 6.2 Hydra Overview

In the early 1970s, a project began at Carnegie-Mellon University to investigate computer structures for artificial intelligence applications. These applications required substantial processing power available only on costly high-performance

processors. At that time, however, relatively inexpensive minicomputers were becoming available. Therefore, the project sought to demonstrate the cost performance advantages of multiprocessors based on inexpensive minicomputers.

The C.mmp hardware was designed to explore one point in the multiprocessor space [Fuller 78]. Its hardware structure differs from conventional multiprocessing systems in the use of minicomputers, the large number of processors involved, and the use of a crossbar switch for interconnecting processors to main memory. C.mmp consists of up to 16 DEC PDP-11 minicomputers connected to up to 32 megabytes of shared memory. The memory is organized in 16 memory banks connected to the processing units by a 16 x 16 crossbar switch.

Hydra [Wulf 74a, Wulf 81] is the operating system kernel for the C.mmp computer system. Hydra is not a complete operating system in the sense of Multics, Tops-20, or Unix™; rather, it is a base on which different operating system facilities can be implemented. For example, Hydra allows users to build multiple file systems, command languages, and schedulers. Hydra was designed to allow operating system experimentation: flexibility and ease of extension were important goals. Experimentation is often difficult with traditional operating systems because new subsystems require change to a privileged kernel. Any error in privileged code can cause a system failure. To avoid this problem, the designers of Hydra built a kernel on which traditional operating system components could be implemented as user programs. This facility has strong implications for the protection system because user programs must be able to protect their objects from unauthorized access.

Two fundamental design decisions that permit experimentation on the Hydra system are:

- the separation of policy and mechanism in the kernel [Levin 75], and
- the use of an object-based model of computation with capability protection.

The separation of policy and mechanism allows experimentation with policy decisions such as scheduling and memory management. Basic mechanisms, such as low-level dispatching, are implemented in the kernel, while scheduling policy for user processes can be set by (possibly multiple) higher-level procedures. Because this part of the Hydra design is not related to the object system, it will not be described here.

Hydra's object model and its implementation are the subject of the following sections.

### 6.3 Hydra Objects and Types

The philosophy that "everything is an object" is key to the Hydra design. All physical and logical resources available to Hydra programs are viewed as objects. Examples of objects are procedures, procedure invocations (called local name spaces), processes, disks, files, message ports, and directories. Objects are the basic unit of addressing and protection in Hydra and are addressed through capabilities. The Hydra kernel's main responsibility is to support the creation and manipulation of (1) new object types, (2) instances of those types, and (3) capabilities.

Each Hydra object is described by three components:

- A *name* that uniquely identifies the object from all other objects ever created. The name is a 64-bit number constructed from an ever-increasing system clock value and a 4-bit number identifying the processor on which the object was created.
- A *type* that determines what operations can be performed on the object. The type is actually the 64-bit name of another object in the system that implements these operations.
- A *representation* that contains the information that makes up the current state of the object. The representation consists of two parts: a *data-part* and a *C-list*. The data-part contains memory words that can be read or written; the C-list contains capabilities for other objects and can only be modified through kernel operations.

Figure 6-1 shows an example of a Hydra object. Although shown as strings, the object's name and type are actually 64-bit binary numbers. The object's type is the name of another object in the system—a type object. Hydra objects include capabilities as part of their representation. By storing capabilities for other objects in its C-list, an object can be built as a collection of several Hydra objects.

Each Hydra type represents a kind of resource. A *type object* is the representative for all instances of a given resource. It contains:

- information about the creation of new instances of the type (for example, the initial C-list size and data-part size), and
- capabilities for procedures to operate on instances of the type.

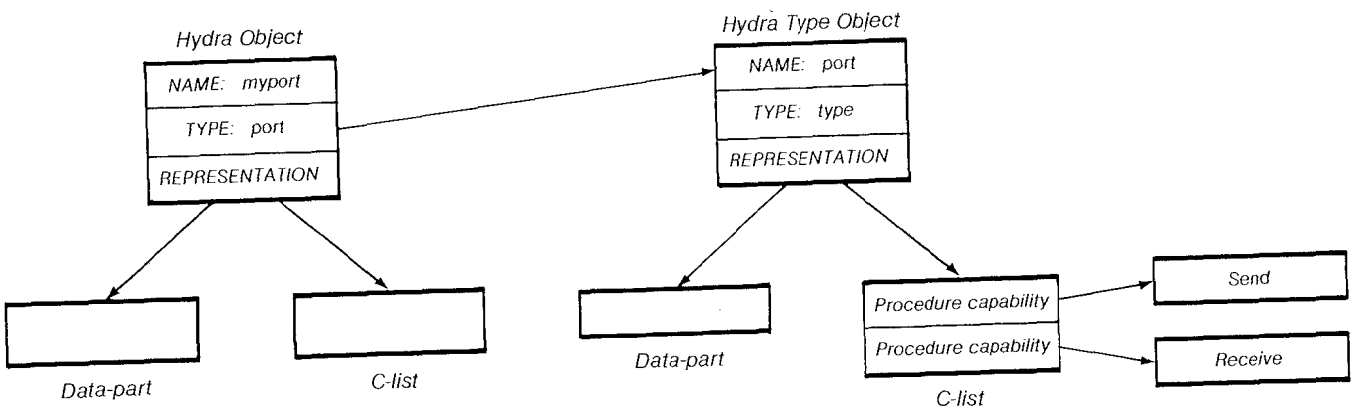



*Figure 6-1:* Hydra Object and Type Object

| | |
|---|---|
| PROCESS | The basic unit of scheduling and execution. |
| PROCEDURE | The static description of an executable procedure. |
| LOCAL NAME SPACE (LNS) | The dynamic representation of an executing procedure. |
| PAGE | A virtual page of C.mmp memory that can be directly accessed. |
| SEMAPHORE | A synchronization primitive. |
| PORT | A message transmission and reception facility. |
| DEVICE | A physical I/O device. |
| POLICY | A module that can make high-level scheduling policy decisions. |
| DATA | An object with a data-part only. |
| UNIVERSAL | A basic object with both a C-list and data-part. |
| TYPE | The representative for all objects of a given type. |

*Table 6-1:* Hydra Kernel-Implemented Types

Thus, the type object is generally responsible for creating new objects of its type and performing all operations on those objects. For example, to create a new message port, the user issues a $CREATE call to the port type object. The port type object creates a new port object, initializes its data-part and C-list appropriately, and returns a capability for the object to the caller. Table 6-1 lists the types directly supported by the Hydra kernel for performance reasons.

To extend the Hydra operating system, users create new type objects that support new kinds of resources. A user creates a new type object by calling the type manager for type objects. Figure 6-2 shows the three-level Hydra type hierarchy. Note that all objects are represented by a type object, including the type objects themselves. The special type object at the root of the hierarchy is called type "type"; that is, both its name and type are "type." This specially designated object is used to create and manipulate type objects.

## 6.4 Processes, Procedures, and Local Name Spaces

A *process* is the basic unit of scheduling in the Hydra system. There is no explicit process hierarchy; any process can create other processes and pass capabilities for those processes to others. The access rights in a process capability determine what operations can be performed on that process—for exam-

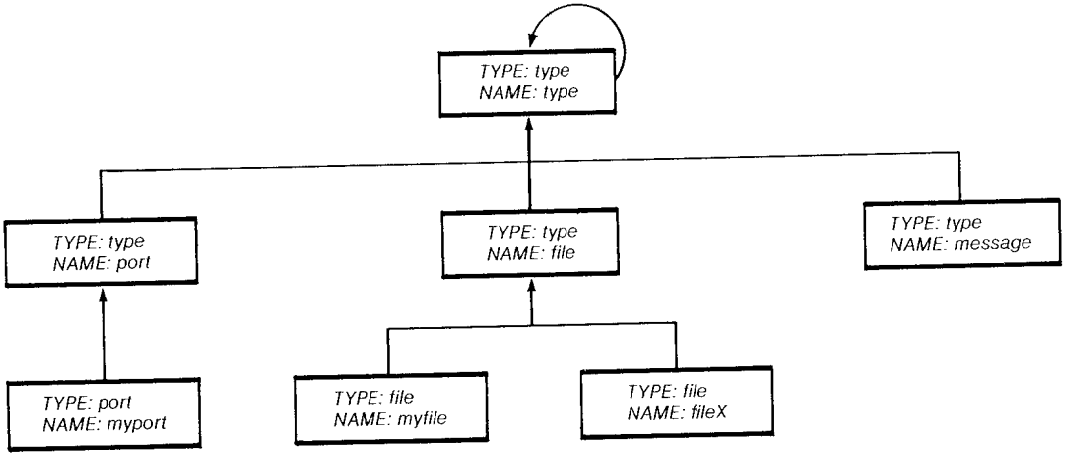



*Figure 6-2:* Hydra Type Hierarchy

ple, whether it can be stopped and started. A process with suitably privileged capabilities can, therefore, schedule the execution of other processes.

The Hydra protection system is procedure-based rather than process-based. All Hydra procedures are protected procedures that carry their own execution domains. The current domain of a process depends on the procedure that it is executing. The process is the entity in which the procedure is scheduled, and it maintains the chain of procedure calls that have occurred within the process.

To differentiate a procedure from its executing invocations, Hydra supports two object types: the procedure object and the *local name space* object. A Hydra *procedure object* is the *static* representation of a procedure. The procedure object contains instructions, constant values, and capabilities that are needed by the procedure for its execution. The capabilities are maintained in the C-list of the procedure object.

The procedure object is actually a template from which an activation is built when the procedure is called. A procedure is called through a procedure capability. When a procedure call occurs, the Hydra kernel creates a local name space object and initializes it from information contained in the associated procedure object. The LNS is the activation record for the executing procedure; it represents the *dynamic* state of the procedure's execution. Since procedures can be shared, several LNS objects can exist to represent different activations of a single procedure. Hydra allows both recursive and re-entrant procedures.

The LNS defines the dynamic addressing environment for a procedure. All of the objects that can be directly addressed by the procedure must be reachable through capabilities in the C-list of the LNS. The capabilities are initially obtained from two places:

- the called procedure object (these are known as inherited capabilities), and
- capability actual parameters passed by the caller.

Within the executing procedure, capabilities are addressed by their index in the LNS C-list. As the procedure executes, the LNS changes as capabilities are acquired, copied, and deleted.

### 6.5 Hydra Operations

C.mmp is constructed from PDP-11 minicomputers, which do not support capabilities or virtual memory addressing.

Therefore, all Hydra object operations are performed through calls to the Hydra kernel. A procedure cannot manipulate the data-part of an object with processor instructions. Instead, the procedure performs a kernel operation to copy data from the data-part into its local memory for examination or modification. Another call to the kernel moves data from local memory to the object's data-part. No direct copying is allowed to the C-list.

Since a number of operations are common to objects of all types, the kernel provides a set of *generic operations* that can be applied to any object, assuming the caller has a sufficiently privileged capability. Table 6-2 lists some of these object operations, as well as some of the standard capability operations.

A typical kernel call might specify several parameters that are capabilities. In general, any parameter requiring a capability will also allow a *path* to a capability. The path allows a user to specify several levels of indirection to the target object. The path is specified as a list of C-list indices, leading from a capa-

| | |
|---|---|
| $GETDATA | Copy data from the data-part of a specified object to local memory. |
| $PUTDATA | Copy data from local memory to the data-part of a specified object. |
| $APPENDDATA | Append data from local memory to the data-part of a specified object, extending the size of the data-part. |
| $MAKEDATA | Create a new data object (data-part only) initialized with N words from a local segment, and return a capability for the new object. |
| $MAKEUNIVERSAL | Create a new universal object (data-part and C-list) and return a capability for the new object. |
| $GETCAPA | Copy a specified target capability (e.g., in a specified object's C-list) to the current LNS (local addressing environment). |
| $PUTCAPA | Copy a capability from the current LNS to a specified object C-list slot. |
| $APPENDCAPA | Append a capability from the current LNS to a specified object's C-list, extending the C-list size. |
| $COMPARE | Compare two capabilities. |
| $RESTRICT | Reduce the rights in a specified capability. |
| $DELETE | Delete a specified capability. |
| $CREATE | Create a new object with the same type and representation as another object. |

*Table 6-2:* Generic Object and Capability Operations

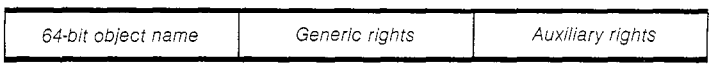



*Figure 6-3:* Hydra Capability

bility in the current LNS C-list, through a capability in the C-list of the object selected, and so on.

## 6.6 Capabilities and Rights

Hydra capabilities contain an object's name and access rights. The access rights are divided into two fields: a 16-bit *generic rights* field and an 8-bit *auxiliary rights* field, as illustrated in Figure 6-3. (This figure is somewhat simplified; capabilities have different formats which are shown in detail in Section 6.9.) The generic rights, listed in Table 6-3, can be applied to any Hydra object. In general, they control permission to execute the generic operations listed in Table 6-2. The auxiliary rights field is type specific; its interpretation is made by the procedures that operate on the specific object type.

The rights are single-bit encoded, and the presence of a bit always indicates the granting of a privilege. This convention simplifies rights restriction and rights checking and allows the

| | |
|---|---|
| GetDataRts, PutDataRts, AppendDataRts | Required to get, put, or append data to an object's data-part. |
| GetCapaRts, PutCapaRts, AppendCapaRts | Required to get, put, or append to an object's data-part. |
| DeleteRts | Allows this capability to be deleted from a C-list. |
| KillRts | Allows deletion of capabilities from the C-list of the named object. The capability to be deleted in that C-list must have DeleteRts. |
| ModifyRts | Required for any modification to an object's representation. |
| EnvRts | Environment rights allows a capability to be stored outside of the current LNS. |
| UncfRts | Unconfined rights allows an object addressed through a specified object to be modified. |
| CopyRts | Required to execute the $COPY operation. |

*Table 6-3:* Capability and Generic Object Access Rights

kernel to verify that a capability has sufficient generic and auxiliary rights for a specific operation.

A type manager typically has the power, through possession of a special capability, to gain additional privileges to an object of its type passed by a caller. This facility, known as rights amplification, will be described in Section 6.7. In some cases a caller may wish to restrict a subsystem's use of capability parameters and the objects they address. In particular, the user may wish to ensure that a called procedure *does not*:

- modify the representation of an object,
- retain the capability for an object following return of the call, or
- copy information from an object into any memory that could be shared with other programs.

These restrictions can be guaranteed through the use of three special rights listed in Table 6-3: modify rights (ModifyRts), environment rights (EnvRts), and unconfined rights (UncfRts) [Cohen 75, Almes 80].

ModifyRts is required in any capability that is used to modify the representation of an object. For example, in order to write to an object's data-part, the executing procedure must have a capability containing both PutDataRts and ModifyRts. By removing ModifyRts from a capability parameter, a program can guarantee that a called procedure will not modify that object because, unlike the other generic rights, ModifyRts *cannot* be gained through amplification.

EnvRts is required for a procedure to remove a capability from its local name space. When a program removes EnvRts from a capability that is passed as a parameter, it guarantees that no copies of the capability can be retained by the called domain following its return. Without EnvRts, it is impossible for a called procedure to save a capability in a local object's C-list to be used later. Although a capability without EnvRts can be passed to another procedure as a parameter, that procedure will once again find a capability in its LNS without EnvRts and will not be able to save it. EnvRts also cannot be gained through amplification.

Although EnvRts prohibits a procedure from saving a capability, it does not prohibit the procedure from copying all of the possibly confidential information from that object into a new object. UncfRts, when removed from a procedure capability, restricts the storage of information by the called proce-

dure. If a procedure is called using a capability lacking UncfRts, all capabilities copied from the procedure object into the LNS will have UncfRts and ModifyRts removed. That is, the procedure will be forced to execute in an environment in which it cannot modify any of its own objects or any objects reachable through its own capabilities. Therefore, it will not be able to maintain any permanent state following its return. The only objects that can be modified by the call are those passed by capability parameters that contain ModifyRts.

## 6.7 Supporting Protected Subsystems

A major goal of the Hydra system is the support of the object-based programming methodology. That is, facilities are added to the operating system by creating new object types. A *type manager*, represented by a Hydra type object, is a module that creates new instances of its type and performs operations on those instances. The objective of this methodology is to localize knowledge of the representation and implementation of each type to its type manager. Users can call type manager procedures to create and manipulate objects, but cannot directly access an object's representation.

To support this programming style, a type manager must be able to:

- create new object instances of its type,
- return a capability for a new instance to the caller requesting its creation (this capability identifies the object but must not allow its owner to access the object's representation directly), and
- retain the ability to access the object's representation when passed a capability for an object it created.

The type manager must, therefore, be able to *restrict* the rights in a capability that it returns to a caller and later *amplify* those rights when the capability is returned. The amplified rights permit the type manager to examine and modify the object's representation. Amplification occurs during procedure calls through a special type of capability owned by the type manager called a template.

### 6.7.1 Templates

There are two common operations that the kernel performs during Hydra procedure calls. First, the kernel verifies that parameter capabilities have the expected type and required

rights for the operation implemented by the procedure. Second, the kernel can, under controlled circumstances, amplify the rights passed in a capability parameter. This facility is required to allow subsystems to perform operations on an object that are not permitted to the user of the object.

Both the type checking and amplification facilities are provided through a mechanism called capability *templates*. A template is a kind of capability used by type managers to implement type systems. Templates do not address objects, but give the possessor special privileges over objects or capabilities of a specified type. As the name implies, the template capability is a form used to verify the structure of a capability or to construct a capability. Templates are stored in procedure C-lists and can be manipulated with capability operations. There are three types of templates: parameter templates, amplification templates, and creation templates.

*Parameter templates* are used to verify the capability parameters passed to a procedure. The procedure object's C-list contains parameter templates as well as capabilities for procedure-local objects. When a procedure call occurs, the kernel builds the LNS C-list from the procedure object's C-list. The procedure's C-list contains its own capabilities that are copied directly to the LNS C-list and parameter templates that represent slots to be filled in with capabilities passed as parameters. The parameter template contains a type field and a required rights field. When copying a capability parameter to the LNS, the kernel verifies that the type matches the template's type field and that the rights in the capability are *at least as privileged* as those required by the template. Special templates can also be provided that will match any type or rights.

The procedure C-list can also contain *amplification templates*. An amplification template contains a type field and two rights fields: required rights and new rights. The type and required rights fields of the amplification template are used to validate the capability parameter in the same way that a parameter template is used. However, once validated, the kernel stores the capability in the LNS with the *new rights* field specified in the amplification template. These rights can either amplify or restrict rights, as specified by the template.

Amplification templates can only be created by the owner of a capability for a type object. In general, only a type object will own the amplification templates for its own type. However, it is possible for a subsystem to own amplification templates for objects of several types. Figure 6-4 illustrates a Hydra proce-

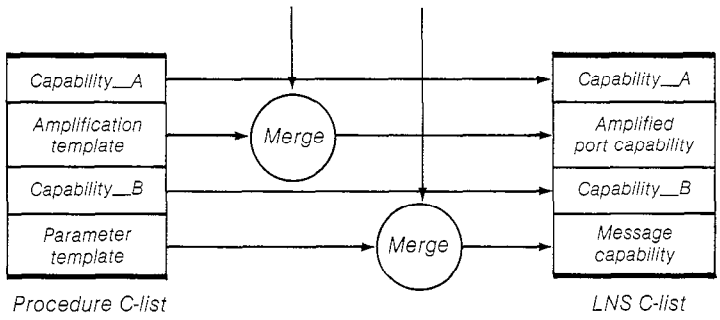



*Figure 6-4*: Hydra Procedure Call

dure call that uses both parameter and amplification templates. The call sends a message, identified by a message object capability, to a port identified by a port object capability. The call is made to the port type manager that must manipulate the representation of the port object to indicate that a message has arrived. In this example, the C-list of the procedure object contains two inherited capabilities that are copied directly to the new LNS. The procedure C-list has an amplification template that is merged with the port capability actual parameter. The merge operation verifies the type and rights of the capability and stores a capability in the LNS with *amplified* rights. The procedure C-list also has a parameter template that is merged with the message capability parameter. In this case, the merge operation simply verifies the type and access rights of that capability and then copies the capability actual parameter into the LNS.

The third template type, the *creation template*, is not used in the procedure call mechanism, but can be used to create a new instance of a specific type. A creation template contains an object type and rights. Using the $CREATE kernel operation, an object with the specified type and rights can be created. In general, subsystems do not provide creation templates; they require that a user call the subsystem in order to create a new instance. The subsystem then uses its private creation template to create the new instance, which the subsystem initializes appropriately. The subsystem might then restrict some of the rights in the capability returned for the new object and pass that restricted capability to the user.

### 6.7.2 *Typecalls*

A Hydra type manager can be thought of as a collection of procedures that has the ability, usually through possession of templates, to manipulate the representation of a particular object type. A program calls these type management procedures using procedure capabilities in the current LNS.

In fact, the concept of type manager is formalized by the Hydra TYPECALL mechanism. A TYPECALL is a call to an object's type manager that is made through the capability for the object itself. Thus, a procedure capability is not needed for a TYPECALL; only a capability for an object is needed. The procedure capability is located in the C-list of the object's type manager, which can be found indirectly through the object.

Figure 6-5 shows an example of the TYPECALL mechanism. The TYPECALL invokes the second procedure in the type object for the specified port object. Two parameters are passed to the TYPECALL, the capability for the port object and the capability for a message object. The capability for the port object is listed twice: once as the object through which the TYPECALL is made and once as a parameter to the TYPECALL.

The TYPECALL mechanism supports abstraction in several ways. First, the owner of an object does not need to possess capabilities for its type object or for procedure objects to manipulate that object. In effect, a TYPECALL requests that the object perform an operation on itself. Second, if all objects support a common set of generic operations at identical type indices, a user can find information without knowing an object's type. For example, if all type objects implement a "tell me your type name" operation as the first procedure and "display yourself" as the second, then a user can apply those operations equally on all objects.

## 6.8 Hydra Object Storage System

A Hydra object, once created, has a lifetime independent of the process that created it. As long as a capability exists for an object, that object will be retained by Hydra and made available when referenced. Hydra stores most long-lived objects on disk when they are not in use and brings them into primary memory when a reference is made. Given a capability for an object, a user can perform any legitimate operation without concern for whether or not the object is currently in primary memory.

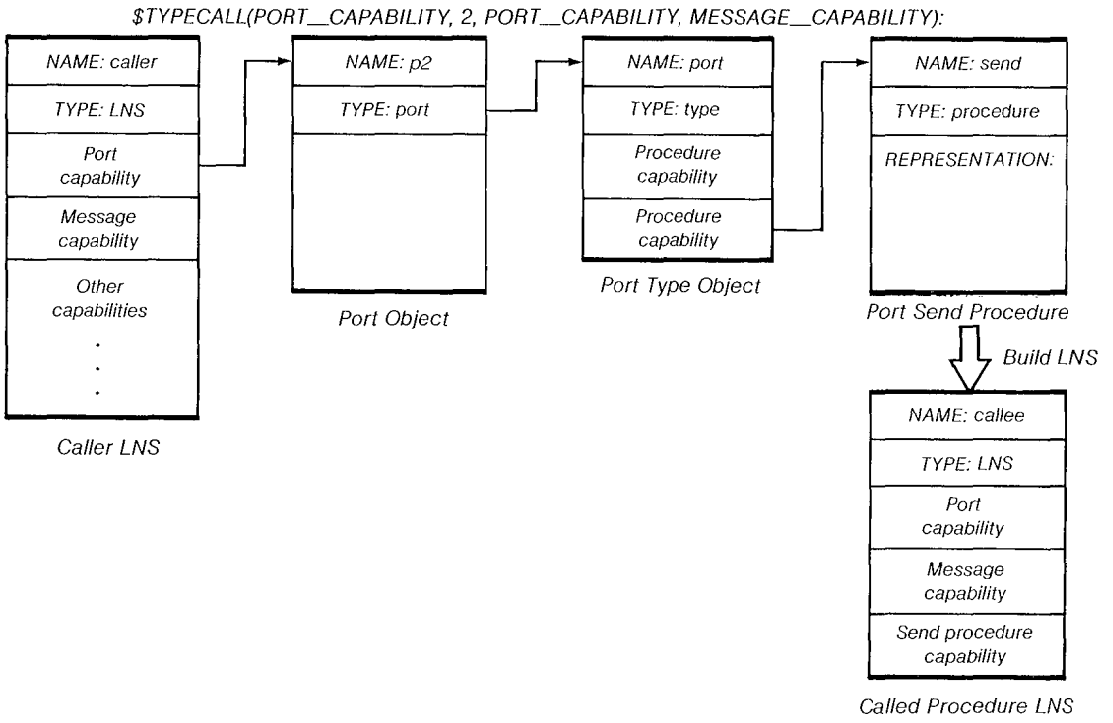



*Figure 6-5*: Hydra TypeCall

Hydra, thus, provides a uniform single-level object addressing environment to its users. Although objects can be stored in primary or secondary memory, the location of an object is invisible to the Hydra user. The Hydra kernel must, therefore, manage the movement of objects between primary and secondary storage. The mechanism for storing and locating objects is the Hydra *Global Symbol Table.*

The Global Symbol Table (GST) contains information about every object reachable by the kernel. The GST is divided into two parts: the Active GST and the Passive GST. The *Active GST* maintains information about objects stored in primary memory, while the *Passive GST* maintains information about objects stored in secondary memory. An object is said to be active or passive depending on whether it is in primary or secondary memory.

As previously stated, the representation of a Hydra object consists of its C-list and data-part. In addition, the kernel constructs data structures called the *active fixed part* and *passive fixed part* that contain state information for active and passive objects, respectively. Table 6-4 shows the formats of the two fixed parts. As their names imply, the fixed parts have a fixed size for easy storage and access. Many object operations can be satisfied by reference to the fixed part alone, and it is possible for the active fixed part to be in primary memory while the representation is still in secondary memory. In this case, the object's fixed part is said to be active while the representation is passive.

When a new object is created, Hydra stores its representa-

| *Passive Fixed Part* | *Active Fixed Part* |
|---|---|
| Global Object Name | Global Object Name |
| Object Flags | Object Flags |
| Current Version Disk Address | Current Version Disk Address |
| Previous Version Disk Address | Previous Version Disk Address |
| Type Name | Total Reference Count |
| Color (for garbage collection) | Active Reference Count |
| | Type Object Index |
| | Checksum of Fixed Part |
| | State |
| | C-List Primary Memory Address |
| | Data-Part Primary Memory Address |
| | Mutex Semaphore (object lock) |
| | Time Stamp (of last access) |
| | Color (for garbage collection) |

*Table 6-4.* Hydra Active and Passive Fixed Parts

tion in primary memory and allocates and initializes an active fixed part. The kernel stores the object's active fixed part in a data structure called the Active GST directory. The Active GST directory is organized as an array of 128 headers of linked lists, as shown in Figure 6-6. Each linked list contains active fixed parts, and the appropriate list for an object's fixed part is determined by a hashing function on the object's 64-bit name.

The division of the Active GST directory into 128 lists serves two purposes. First, it speeds up the GST search, since the linked lists can be kept relatively short. Second, it allows parallelism in the access of the active fixed parts. Only one processor can search a linked list and access a specific active fixed part at a time. By dividing the Active GST into 128 lists, a lock can be maintained separately for each list, allowing simultaneous searches of different lists.

There are two events that cause a Hydra object to be copied to secondary memory. First, *passivation* can be triggered by a kernel process that removes objects from primary memory according to their last reference times. This is analogous to swapping in traditional systems. Second, a program can perform an explicit UPDATE function, requesting that an object's representation be written to disk. In this case, the object remains active with the guarantee that the active and passive copies are identical. The UPDATE operation is used to ensure consistency over system crashes, because any active representation will be lost following a crash. UPDATE is used primarily by type managers.

Two versions of each object are kept on secondary stor-

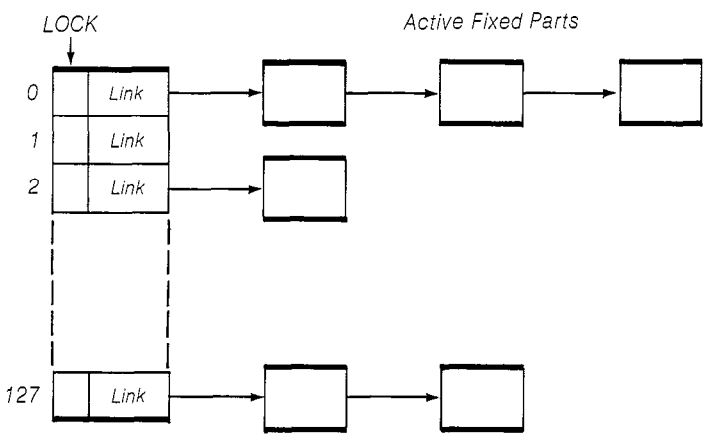



*Figure 6-6*: Active Fixed Part Directory

age—a current version and a previous version. When an object is passivated, its representation is written to secondary storage, destroying the older of the two versions. If any failure occurs during the write operation, the newer version on disk is left intact. Following successful completion of the UPDATE, the newly passivated image becomes the current version, with the former current version becoming the previous version.

Passive objects are stored in the Passive GST. A passive object is stored as a contiguous array of disk blocks containing the passive fixed part, data-part, and C-list. To locate a passive object, a search of the Passive GST directory is made. The Passive GST directory is stored on a high-speed, fixed-head disk and consists of copies of all of the passive fixed parts. The passive fixed parts are organized in 256 blocks for the purpose of synchronization and parallel search. The global object name is used as a key in the search for the correct block.

Object *activation* occurs when the kernel fails to locate a referenced object in the Active GST. The kernel must then search the Passive GST directory. Activation can occur in two phases. First, the object's fixed part is activated. The active fixed part is constructed from information in the passive fixed part. Many operations can be completed with activation of the fixed part alone. Then, if the object's representation must be activated, the C-list and data-part are read into memory.

### 6.9 Capability Representation

Just as Hydra objects can be active or passive, so Hydra capabilities have both active and passive forms. These forms are shown in Figure 6-7. Active and passive capabilities differ in the format of the object address. An active capability contains the primary memory address of the object's active fixed part, while a passive capability contains the object's 64-bit name. An object reference using an active capability is obviously more efficient, as it does not require a GST search.

An active capability cannot be stored on secondary memory because it contains the primary memory address of the active fixed part, which can be swapped out. When Hydra writes an object to secondary storage, it converts all the capabilities in its C-list to passive form. When Hydra activates an object, it leaves capabilities in passive form until they are used. When a program addresses an object through a passive capability, the kernel searches the GST and converts that capability to active form.

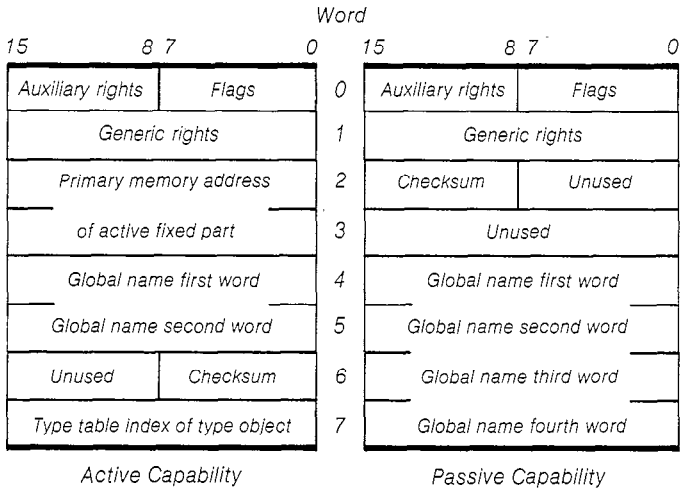



*Figure 6-7:* Hydra Capability Formats

Because an active capability contains the primary memory address of the active fixed part, an active fixed part cannot be removed from memory as long as active capabilities exist for the object. For this reason, an *active reference count* is maintained in the active fixed part. The active reference count indicates the number of physical addresses that exist for the fixed part. When this count is decremented to zero, the active fixed part (and the object's representation) can be passivated.

## 6.10 Reference Counts and Garbage Collection

On systems such as Hydra, with long-term object storage, it is difficult to know when an object can be deleted. An object can have many users since capabilities can be freely passed between processes. Users can also delete capabilities, and when no capabilities exist for an object, the object should be deleted. Objects that are no longer reachable are known as *garbage objects* and the general problem of finding them is known as *garbage collection.*

Reference counts can help in the garbage collection problem, and Hydra maintains both an active reference count and a total reference count in an object's active fixed part. The total reference count indicates the total number of capabilities for an object, including passive capabilities in the Passive GST. If the total and active reference counts in an active fixed part become zero, the kernel deletes the object because it can no longer be referenced.

Reference counts in themselves are insufficient to stop the accumulation of garbage objects for several reasons. First, reference counts cannot catch object reference cycles. For example, if objects X and Y have capabilities for each other in their C-lists but no other capabilities for X and Y exist, then both objects are garbage and should be deleted. However, both objects will have reference counts of one. Second, because the active and passive fixed parts for Hydra objects are not always consistent, any total reference count maintained in the passive fixed part can be in error following a crash. This inconsistency occurs because it is not feasible to modify the passive fixed part reference count on every capability copy operation.

Because of the insufficiency of reference counts, Hydra includes a parallel garbage collector [Almes 80]. The parallel garbage collector consists of a collection of processes that execute concurrently with normal system operation. The garbage collector scans all objects, marking those that are reachable. The *color* field in the active and passive fixed part is provided for this purpose. Following the marking of objects, another scan is made to locate objects that were not marked—those that are unreachable and therefore are garbage. These objects are deleted.

It is important to note that while the garbage collector is running, capabilities can be freely copied and deleted. The Hydra garbage collector must also cope with the dual residency of objects in the Passive and Active GSTs.

### 6.11 Discussion

Perhaps the best indication of Hydra's success is that much of its philosophy now seems obvious. The object model and the large single-level object address space have found their way into contemporary products. These ideas did not completely originate with Hydra, nor was their implementation on Hydra totally successful (reflections on the Hydra/C.mmp system by its designers can be found in [Wulf 78 and Wulf 81]). However, the basic philosophy has proven to be a valuable model for system design.

Although previous capability systems provided primitive objects, user-defined objects, and capability addressing, Hydra is the first to present its users with a uniform model of the abstract object as the fundamental system resource. All resources are represented as objects, and objects can be protected and passed from domain to domain. Users can create new re-

sources, represented by type objects, and can control instances of these resources through type-specific procedures.

As the designers point out, the system probably went too far with the flexibility allowed for object protection [Wulf 81]. For example, although direct operations on an object's representation can be restricted to the object's type manager, the protection system allows any user with a sufficiently privileged capability to access the object. To support this generality in a controlled fashion, Hydra defines a large set of generic object rights. In the usual case, however, only the type manager is allowed to access the object, and it must amplify the needed rights through an amplification template. In general, it would be simpler to restrict representation access to type managers who are implicitly given all rights to their objects' representations.

Hydra also attempts to solve some complex protection problems with special rights bits. A caller can prevent a called procedure from modifying an object or "leaking" information from the object. However, it is not always possible for a procedure to operate correctly without some of the special rights (for example, modify rights). Some subsystems may not be able to operate in a confined environment. In addition, it is often difficult for the caller to know what effect the removal of special rights will have on a called subroutine, although good documentation practices can help alleviate this problem.

Many of Hydra's shortcomings are a result of the hardware base, including the small address space and lack of hardware capability support in the PDP-11s. All capability and object operations are executed by operating system software, and even a type manager must copy data from the representation of its objects to local memory for modification. A domain change on Hydra, which requires creation of a new local name space object, type and rights checking of capabilities, and so forth, takes over 35 milliseconds. This severe penalty for a domain change forces a programming style that is contrary to that which is intended. That is, if domain changes are expensive, programmers will tend to use them infrequently and programs will not be written to execute in the small constrained protection domains originally envisioned.

In general, Hydra's objects are too expensive (in terms of space overhead, time for creation, etc.) for their actual usage. Measurements of Hydra show that over 98 percent of all objects are created and destroyed without ever being passivated [Almes 80]. Hydra objects are, therefore, relatively short-

lived. The same measurements show that the median object size is 46 bytes for the data-part and 6 capabilities for the C-list. The GST active fixed part overhead for such an object is rather large, as is the cost of each capability.

An important feature of Hydra is the use of large object names—its unique-for-all-time object identifiers. By using a 64-bit value for an object's name, the kernel avoids searching for dangling references when an object is deleted. Although an object's name never changes, capabilities are modified when moved between primary and secondary storage. The change of capability format is simply a performance optimization used to reduce the overhead of Hydra's software-implemented capability support. An operation on an object's capabilities, such as the change from active to passive format, is simplified by the fact that all capabilities are stored in a single C-list.

The Hydra GST is the mechanism for implementing a single-level uniform address space. The single-level address space greatly simplifies a number of problems for both users and the operating system. Most programs do not need to know about the existence of secondary storage. For type managers that must ensure that an object's representation is preserved on secondary memory, Hydra provides the UPDATE operation.

The Hydra developers succeeded in constructing a large, functioning operating system (details of the development can be found in [Wulf 75]). In addition, they were able to implement several useful subsystems outside of the kernel, as intended. These included directory systems, file systems, text editors, and command languages. Perhaps the greatest shortcoming of Hydra, however, was that it did not become a system of choice among programmers at Carnegie-Mellon. Lampson and Sturgis, in their retrospective on CAL-TSS, state the common problem of many operating system research projects:

> ...we failed to realize that a kernel is not the same thing as an operating system, and hence drastically underestimated the work required to make the system usable by ordinary programmers. The developers of Hydra appear to have followed us down this garden path [Lampson 76].

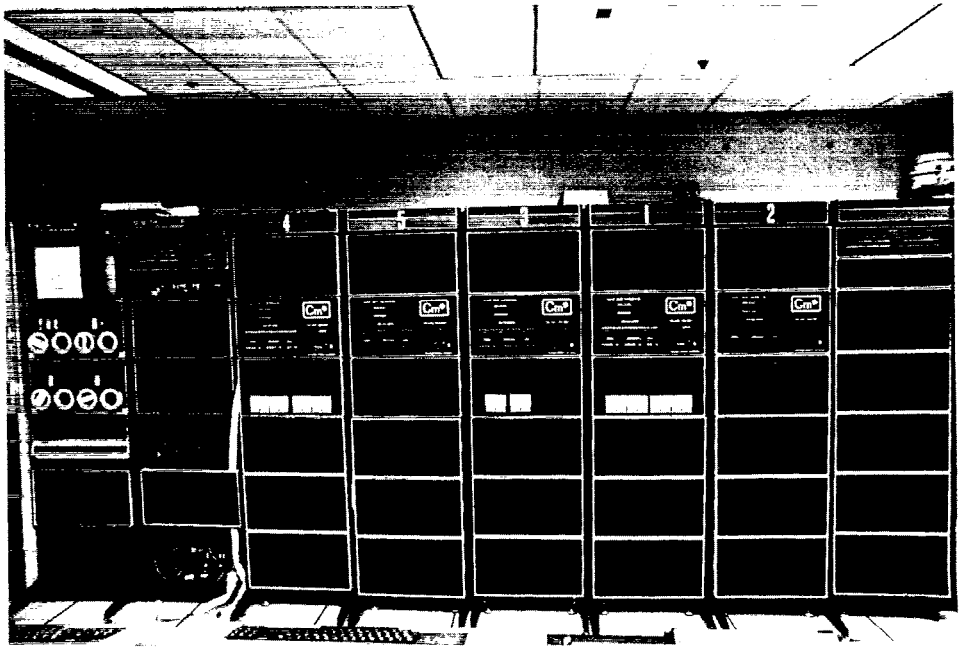
Even so, a tremendous experience was gained from Hydra that has passed to many follow-on systems. The C.mmp hardware was finally dismantled in March 1980; however, still operating at Carnegie-Mellon was a direct descendant of Hydra/C.mmp, which is discussed in the next chapter.

## 6.12 For Further Reading

The Hydra philosophy was first presented in the original *CACM* paper on Hydra [Wulf 74a]. More recently, Wulf, Levin, and Harbison have written an excellent book on the Hydra system that describes both the kernel and some of its subsystems [Wulf 81]. The book also includes performance measurements of Hydra and the C.mmp hardware. The paper by Wulf and Harbison is a retrospective on the Hydra/C.mmp experience [Wulf 78].

Three papers on Hydra appeared in the *Proceedings of the 5th ACM Symposium on Operating Systems Principles* in 1975. These well-known papers describe the separation of policy and mechanism in Hydra [Levin 75], the Hydra protection system [Cohen 75], and the Hydra software development effort [Wulf 75].

Almes' thesis describes the Hydra garbage collector and also presents measurements of the GST mechanism showing object size and lifetime distributions [Almes 80]. The paper by Almes and Robertson describes the construction of one of several Hydra file systems [Almes 78]. Low-level details of the Hydra kernel and its operations are documented in the *Hydra Kernel Reference Manual* [Cohen 76].



The Cm* computer. (Courtesy Dr. Zary Segall.)