**NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:**
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

# HYDRA: THE KERNEL OF A MULTIPROCESSOR OPERATING SYSTEM*

Wulf, Cohen, Corwin, Jones, Levin, Pierson, Pollack

JUN 1973

## Abstract

This paper describes the design philosophy of HYDRA - the kernel of an operating system for C.mmp, the Carnegie-Mellon Multi-Mini-Processor. This philosophy is realized through the introduction of a generalized notion of 'resource', both physical and virtual, called an 'object'. Mechanisms are presented for dealing with objects, including the creation of new types, specification of new operations applicable to a given type, sharing, and protection of any reference to a given object against improper application of any of the operations defined with respect to that type of object. The mechanisms provide a coherent basis for extension of the system in two directions: the introduction of new facilities, and the creation of highly secure systems.

*This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

## Introduction

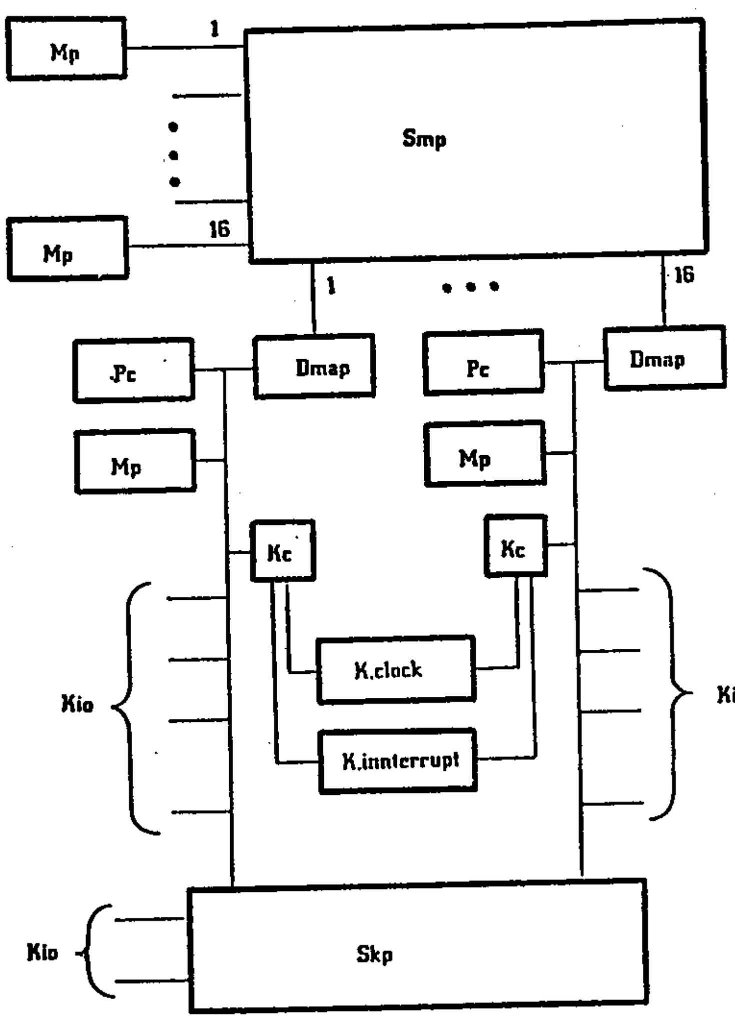
The Hydra system is the 'kernel' base for a collection of operating systems designed to exploit and explore the potential inherent in a multiprocessor computer system. Since the field of parallel processing in general, and multiprocessing in particular, is not current art, the design of Hydra has a dual goal imposed upon it: (1) to provide, as any operating system must, an environment for effective utilization of the hardware resources, and (2) to facilitate the construction of such environments. In the latter case the goal is to provide a meta-environment which can serve as the host for exploration of the space of user-visable operating environments.

The particular hardware on which Hydra has been implemented is C.mmp, a multiprocessor constructed at Carnegie-Mellon University. Although the details of the design of C.mmp are not essential to an understanding of the material which follows, the following brief description has been included to help set the context (a more detailed description may be found in [WB72]). Figure 1 illustrates that C.mmp permits the connection of (up to) 16 processors 32 million bytes of shared primary memory through a cross-bar switch (Smp). The processors are any of the various models of PDP-11* minicomputers. Each processor is actually an independent computer system with a small amount of private memory, secondary memories, i/o devices, etc.. Processors may interrupt each other at any of four priority levels; a central clock serves both for unique-name generation (see below) and broadcasts a central time base to all processors. Relocation hardware (D.map) on each processor's bus provides mapping of virtual addresses on that bus to physical addresses in shared primary memory.

-------------------

*Manufactured by Digital Equipment Corporation.

FIGURE 1: CMU MULTI-MINI-PROCESSOR



WHERE:

Mp = PRIMARY MEMORY; Pc = PROCESSOR;

Smp = PROCESSOR-TO-MEMORY SWITCH;

Skp = PROCESSOR-TO-IO SWITCH;

Dmap = ADDRESS RELOCATION MAP;

Kio = IO DEVICE CONTROLLERS

Kc = SPECIAL HARDWARE CONTROL

K.clock = 60-BIT CLOCK

K.Innterrupt = INTER-PROCESSOR INTERRUPT

## Design Philosophy

The design philosophy of Hydra evolved from both the environment in which the system was to function and a set of principles held by its designers. The central goals of the system together with the attitudes expressed below suggest that, at the heart of the system, one should build a collection of facilities of 'universal applicability' and 'absolute reliability' -- a set of mechanisms from which an arbitrary set of operating system facilities and policies can be conveniently, flexibly, efficiently, and reliably constructed. Moreover, lest the flexibility be constrained at any instant, it should be possible for an arbitrary number of systems created from these facilities to co-exist simultaneously. The collection of such facilities has been called the kernel or nucleus [Br70] of an operating system. The more specific considerations are listed below:

1. Multiprocessor environment: Although multiprocessors have been discussed for well over a decade and a few have been built, both the potentials and problems of those systems are dimly perceived. The design of Hydra was constrained to be sufficiently conservative to insure its construction and utility in a reasonable time frame, yet flexible enough to permit experimental exploration within the design space bounded by its hardware capabilities.

2. Separation of mechanism and policy: Among the major causes of our inability to experiment with, and adapt, existing operating systems is their failure to properly separate mechanisms from policies. (Hansen [Br70] has presented that policy. This systematic exclusion from the kernel of policy-making code markedly contributes to the flexibility of the system, for it leaves the complex decisions in the hands of the person who should make them - the higher-level system designer. When necessary, the kernel instantiates policy decisions by invoking various "policy modules" in a higher level system associated with the program for which the decision must be made. Thus, the designer of an operating environment for C.mmp is obligated to consider precisely those units of code which are his exclusive responsibility. We believe this represents a major step forward from conventional systems.

3. Integration of the design with implementation methodology: It has been observed that the structure of extant operating systems bear a remarkable resemblance to that of the organization which created them. This observation is one of a set which asserts the (practical) impossibility of separating the design from the methodology to be used in implementing the design. The authors' predisposition for

implementation methodology is a cross between structured programming as advocated by Dijkstra and others [DDH72] and the modularization philosophy of Parnas [Pa71].

4. Rejection of strict hierarchical layering: The notion of a strict hierarchically layered system has become popular since first described by Dijkstra for the THE system [Dij68a]. While we believe that the system as viewed by any single user should be hierarchically structured, we reject the notion as a global design criterion. We believe that if the entire system is so structured, the design will severely limit the flexibility available to the high-level user and will strangle experimentation; in particular, there is no reason to believe that the same hierarchical relation should exist for control as for resource allocation, or as for protection, etc.

5. Protection: Flexibility and protection are closely related, but not inversely proportional. We believe that protection is not merely a restrictive device imposed by "the system" to insure the integrity of user operations, but is a key tool in the proper design of operating systems. It is essential for protection to exist in a uniform manner throughout the system, and not to be applied only to specific entities (e.g., files). The idea of capabilities (in the sense of Dennis [DVH66]) is most important in the Hydra design; the kernel provides a protection facility for all entities in the system. This protection includes not only the traditional read, write, execute capabilities, but arbitrary protection conditions whose meaning is determined by higher-level software.

6. Reliability: The existence of multiple copies of critical hardware resources suggests the possibility of highly reliable operation. Our desire is to provide commensurate reliability in the software. We include under the title reliability - data integrity, both system and user; crash rate; percentage of uptime, etc. In conventional multiprogramming and timesharing systems reliability is heavily weighted by the hardware reliability and, in general, if the (central) processor goes down, the system goes down - often with a considerable loss of information. In a uni-processor configuration the alternatives are few, while common sense suggests that a symmetric multiprocessor arrangement should be able to do better. A primary consideration in the design of Hydra is the desire for intelligent detection of, and recovery from, hardware failures. (By "intelligent" we do not mean "take a core dump and reload".) In a multiprocessor

configuration it should be possible to isolate and disable a malfunctioning processor and transfer its load to the remaining ones with a minimal loss of continuity. Thus, symmetry considerations in the design of the hardware and software are most important.

Data consistency is as important as execution continuity, particularly if the protection facility available at the kernel level is to be reliable. It is essential to store vital data in such a fashion that the chances of recovering from a random memory failure are maximized. As a natural result of the protection mechanism, "vital data" includes not only kernel tables but also data structures in the higher-level software whose content is uninterpreted by the kernel, but whose form is precisely known. Thus, we can provide the same consistency checks for higher-level systems that we use for the kernel.

Another point should be made in connection with reliability. Integrity of form may be sufficient to allow the lowest levels of the system to function, but integrity of content is much more important to higher-level software. When a hardware malfunction occurs, the kernel must ascertain and correct any resulting errors in its own operations (e.g., verifying a table), but this is not sufficient in itself. Higher-level programs must be notified that a malfunction has occurred which may have altered some program or data entity [Pa72] (this is another example of the non-hierarchical organization of Hydra). Reliability requires intelligent error handling at all software levels in which this type of recovery behavior is possible.

Defining a kernel with the attributes given above places an awesome responsibility upon its designers. It is, nevertheless, the approach taken in the Hydra system. Although we make no claim that the set of facilities provided by the Hydra kernel is either minimal (the most primitive 'adequate' set) or maximally desirable, we do believe the set provides primitives which are both necessary and adequate for the construction of a large and interesting class of operating environments. It is our view that the carefully chosen, well-integrated set of functions provided by Hydra will enable the user of C.mmp to create his own operating environment without being confined to pre-determined command and file systems, execution scenarios, resource allocation policies, etc.

Given the general decision to adopt the 'kernel system' approach, the question remains as to what belongs in a kernel and, perhaps more importantly, what does not. Non-specific answers to this question are implicit in the attitudes enumerated earlier, e.g., a set of mechanisms may be appropriate in a kernel, but policy decisions certainly are not. For other, more specific, answers we must step outside these attitudes alone and consider the nature of the entity to be built using the facilities of a kernel.

If a kernel is to provide facilities for building an operating system, and we wish to know what these facilities should be,then it is relevant to ask what an operating system _is_ or _does_. Two views are commonly held: (1) An operating system defines a 'virtual machine' by providing facilities, or resources, which are more convenient than those provided by the 'bare' hardware, and (2) An operating system allocates (hardware) resources in such a way as to most effectively utilize them. Of course these views are, respectively, the bird's-eye and worm's-eye views of what is a single entity with multiple goals. Nevertheless, the important observation for our purposes is the emphasis placed, in both views, on the central role of _resources_ -- both physical and virtual.

The mechanisms provided by the Hydra kernel are all intended to support the abstracted notion of a resource (incarnations of a resource are called _objects_). These mechanisms provide for the creation and representation of new _types_ of resources, as well as operations defined on them, protected access to instances of one or more resources within controlled execution domains, and controlled passing of both control and resources between execution domains. The key aspects of these facilities are the generalized notion of resource, the definition of an execution domain, and the protection mechanism which allows or prevents access to resources within a domain.

## _Overview of the Hydra Environment_

Before proceeding to a detailed description of the mechanisms it will be convenient to present a somewhat incomplete and simplistic view of the execution environment created by the Hydra kernel. The material presented in this section will be elaborated further in the following sections; however, the overview will attempt to provide the context necessary to understand the more detailed information.

In order to understand the execution environment which the kernel provides, one must clearly understand the interrelationships of three object types: _procedure_, _LNS_, and _process_. These primitive objects are provided by the kernel specifically for the purpose of creating and manipulating an execution environment.

The procedure object is simply an abstraction of the intuitive notion of procedure or subroutine; that is, a procedure has some "code" and some "data" associated with it, it may accept parameters, and it may return values. Hydra procedures go beyond this simple model by including protection facilities, as we shall see shortly. The act of creating a procedure object is analagous to the task of writing an ALGOL procedure; one produces a body of code, associates the code with a name, declares the data which the code requires, and specifies the nature of the parameters and return values which are involved. In more abstract terms, one creates a sequence of instructions and describes the environment in which they will ultimately execute; in Hydra this abstraction is made precise. Let us consider the environment description first.

A procedure object contains a list of references to other objects which must be accessed during the execution of the procedure's code. This is, in fact, a list of capabilities [La69] and therefore defines not only which objects the procedure may reference, but also what actions it may perform on those objects. The capabilities which a procedure requires may be divided into two groups: those which are caller independent and those which are caller dependent. These groups naturally correspond to those objects which the procedure always accesses (at least potentially), and those objects which are considered parameters. Obviously, the former of these groups can be precisely specified at the time the procedure is created, while the latter can only be characterized, since the actual objects passed as parameters are unknown until execution time. Thus, the environment defined by a procedure object contains some "holes" or "parameter positions" which are only partially specified at creation time. These holes are filled in for each execution of the procedure, using capabilities provided by the caller. We will return to a discussion of the mechanism by which a procedure characterizes its parameters, but first we must examine the LNS.

A procedure is a static entity; an LNS (local name space) is the record of the execution environment of a procedure when that procedure is invoked (called). There is a unique LNS for each invocation, which disappears after the procedure terminates. The LNS for a particular invocation is the result of combining the caller-independent capabilites (listed in the procedure object) with caller-dependent actual parameters (only characterized in the procedure object) to form a single list of capabilities. The LNS defines the totality of capabilities available to a procedure during the execution resulting from a particular invocation. The elements of an LNS are called items and consist of a reference to an object together with the capabilities just described. Note that the LNS, while heavily dependent upon the corresponding procedure for its initialization, is a wholly independent object thereafter, and alterations of the LNS do not affect the procedure object; this implies, among other things, that procedures are reentrant and potentially recursive.

A procedure object may contain templates for items in addition to the usual collection of caller-independent capabilities. Templates characterize the actual parameters expected by the procedure. When the procedure is called, the slots in the LNS which correspond to parameter templates in the procedure object are filled with "normal" items derived from the actual parameters supplied by the caller. This "derivation" is, in fact, the heart of the protection-checking mechanism, and the template defines the checking to be performed. If the caller's rights are adequate, an item is constructed in the (new) LNS which references the object passed by the caller and which contains rights formed by merging the caller's rights with the rights specified in the template. This implies that a callee may have greater freedom to operate on an object than the caller who passed it as a parameter, but the caller can in no way obtain that freedom for himself. We shall see that this potential expansion of rights across environment domains is a key factor in achieving the flexibility goals of the kernel and in allowing us to reject hierarchical structures without introducing chaos.

Before proceeding let us review the major actions of the CALL mechanism. An executing body of code first notifies the kernel that it wishes to call a procedure. The kernel examines the actual parameter items specified by the caller and determines whether all protection requirements are met. If so, the kernel creates a new LNS which defines the new environment; the caller's LNS is superceded by this new LNS for the duration of the called procedure's execution. The body of code associated with the callee receives control from the kernel and begins executing. When it completes its function, it will return control to its caller by way of the kernel. The kernel will delete the callee's LNS and restore that of the caller, thus returning to the previous environment.

## Processes

Up to this point, nothing we have described suggests any exploitation of the parallel processing capabilities of C.mmp. The actions involved in calling and returning from procedures are strictly sequential in nature, being in essence formalizations of traditional subroutine behavior. We come now to the unit of asynchronous processing in Hydra - the process. A process in the technical sense defined by Hydra corresponds closely to one's intuitive notion of a process. Viewed from the outisde, it is the smallest entity which can be independently scheduled for execution. Viewed from the inside, it constitutes a precise record of the changes of environment induced by a sequence of calls. In other words, a process is a stack of LNS's which represents the cumulative state of a single sequential task. Hydra implements inter-process communication and synchronization by providing elementary message buffering primitives and Dijkstra-style semaphore operations. These facilities are well-understood [Dij68b] and will not be discussed here.

Scheduling a process for execution involves two fundamentally different tasks: (1) making policy decisions which determine the process's priority, and (2) performing the mechanical job of starting the process when the proper time arrives. The former of these tasks is associated with the so-called scheduling algorithm and is implemented via policy modules which reside outside the kernel. The kernel handles the second task by providing a scheduling mechanism which performs process multiplexing according to parameters set by the higher-level policy-maker. These issues wil be discussed more fully later when the appropriate context is available.

## The Protection Mechanism

The protection mechanism is at the heart of the Hydra design. In describing the mechanism it is important at the outset to distinguish between protection and security and to determine what is to be protected and against what.

Protection is, in our view, a mechanism. A system utilizing that mechanism may be more or less secure depending upon policies governing the use of the mechanism (for example, passwords and the like are policy issues) and upon the reliability of the programs which manipulate the protected entities. Thus the design goal of the Hydra protection mechanism is to provide a set of concepts and facilities on which a system with an arbitrarily high degree of security may be built, but not to inherently provide that security. A particular consequence of this philosophy has been to discard the notion of 'ownership'. While ownership is a useful, perhaps even important, concept for certain 'security' strategies, to include the concept at the most primitive levels would be to exclude the construction of certain other classes of truly secure systems.

Our rejection of hierarchical system structures, and especially ones which employ a single hierarchical relation for all aspects of system interaction, is also, in part, a consequence of the distinction between protection and security. A failure to distinguish these issues coupled with a strict hierarchical structure leads inevitably to a succession of increasingly privileged system components, and ultimately to a "most privileged" one, which gain their privilege exclusively by virtue of their position in the hierarchy. Such structures are inherently wrong, and are at the heart of society's concern with computer security. Technologists like hierarchical structures; they are elegant, but experience from the real world shows they are not viable security structures. The problem, then, which Hydra attempts to face squarely, is to maintain order in a non-hierarchical environment.

The obvious candidate for (the unit of) protection is the object since this is the abstracted notion of an arbitrary resource. Similarly, the Hydra procedure is considered to be the abstraction of an operation. Thus Hydra provides a protection mechanism for the application of operations (_procedures_) to instances of resources (_objects_). All of the familiar security for files (e.g., read, write, delete), memory (e.g., read, write, execute), etc., can be conveniently modeled in this way. In addition a large additional class of secure systems can be built.

Everything of interest in the Hydra view is the abstracted notion of a resource, called an _object_, or a reference to an object, called an _item_. Each object has a _unique name_, a _type part_, and a _representation_ (consisting of an _item part_, and a _data part_).

The unique name of an object distinguishes the object not only from all other extant objects, but from all objects which have existed or will exist. Knowledge of the unique name of an object does _not_ grant access to the object since objects may only be referenced through items (which are not manipulable except by the kernel).

The type part of an object serves to identify the object with that class of objects whose type parts have an identical value. The type part contains, in fact, the unique name of a distinguished object which serves as the representative of such a class. Since there is a potentially infinite supply of unique names, there is a potentially infinite number of object types as well. A new class of objects may be created simply by creating a single object to serve as its distinguished representative.

Objects become inaccessible only when there are no references to them. It is possible to generate self-referential structures, and although a general garbage-collection deletion mechanism is required, these structures are rare. Hence a reference count is maintained in each object and objects are deleted* when this count becomes zero.

The representation portion of an object contains whatever information is relevant to the representation of the resource which the object denotes. This information may be of two types: data (which is uninterpreted by the kernel) and references to other objects. These two kinds of information are stored in the _data_ and _item_ parts of the object respectively. Given the appropriate access rights a program may manipulate the data part of an object freely. Even with the most liberal access rights, however, the item part of an object may be manipulated only by invocation of kernel functions.

------------------

*The deletion strategy is not essential to an understanding of Hydra, therefore this attribute of objects will not be discussed further.

On purely formal grounds, the object-universe described above may be derived by starting with a single distinguished object whose unique name** is TYPE and whose type is TYPE -- that is, it names itself as the representative of the class with only a single member. Subsequent objects are created by naming an extant object as their representative. Although it is possible to allow any object to serve as the representative, nothing is gained by doing so; therefore, we restrict the representatives to being objects whose type is TYPE. Consider, for example, the following collection of objects:

| NAME | TYPE |
|---|---|
| TYPE | TYPE |
| PAGE | TYPE |
| FILE | TYPE |
| DISK | TYPE |
| P1 | PAGE |
| P2 | PAGE |
| F1 | FILE |
| D1 | DISK |
| D2 | DISK |

In this case we have four distinct classes of objects: TYPE, PAGE, FILE, and DISK. The objects with these names and type TYPE are the representatives of the classes. The members of the classes are, respectively, {TYPE, PAGE, FILE, DISK}, {P1, P2}, {F1}, {D1, D2}. Thus, for example, P1 and P2 are actual page objects; presumably the data part of these two objects would contain the actual bit configurations one would expect to find in our usual intuitive concept of a page. Notice that we have purposely used objects in the example to represent both virtual resources (files) and physical resources (disks); this is precisely how they are used in Hydra.

On practical grounds, it is both inconvenient and inefficient to begin with a single distinguished object. The following object types are some of those present in the base system:

------------------

**We have tried to use suggestive names for objects in the example. In practice, of course, the unique names of objects have no visible representation; they are 64-bit values obtained from K.clock (Figure 1).

| | |
|---|---|
| TYPE | as described above |
| NULL | the empty object |
| LNS | the 'current' LNS defines the execution |
| PROCEDURE | combined code and partial environment (complete environment definition when procedure is 'called') |
| PROCESS | the smallest separately schedulable |
| SEMAPHORE | Dijkstra style [Dij68b] synchronization |
| PAGE | the counterpart of pages in TSS, etc. |
| {various device-type objects, e.g., disk} | |

As mentioned briefly several times previously, in addition to objects Hydra supports references to objects (called items). Items are, however, more than simple pointers. Together with the CALL mechanism (for crossing execution domains), items are the key to the protection mechanism. As such, items may only exist in the item part of an object and may be directly manipulated only by the kernel.

Each item includes information detailing the operations which may be performed on the object referenced by the item. This information is in the form of a capability list [La69], called the rights list; hence an item consists of two parts: the name of an object and a rights list. Whenever an operation is attempted on an object, the requestor supplies an item referencing that object. The kernel examines the rights list and prevents the operation when a protection failure occurs (i.e., when the requestor does not have the appropriate capabilities). It is important to understand that the rights checking operation does not require interpretation of the list; the kernel can determine when a protection failure has occurred without assigning meaning to the individual rights bits.

Not all rights are type-dependent; there exist operations worthy of protection which are well-defined for any object. These are precisely the operations which the kernel provides for controlled manipulation of objects. Accordingly, we partition the rights list of an item into two mutually exclusive sets -- the type-independent rights (called kernel rights), and the type-dependent rights (called auxilliary rights). A 'right' in either set grants permission to pass the item as a parameter to any procedure in a particular class. The kernel defines these classes for type-independent rights, and the creator of the type defines them for auxilliary rights.

The notions of object and item now permit us to be more specific about one of the object types mentioned earlier - the LNS. At any instant of time,the execution environment of a program is defined by an LNS object associated with it. The item part of the LNS contains references to objects wich may be accessed by that program at that instant. (In addition, of course, the program may be able to access objects referenced by the objects referenced in its LNS, and so on.) The LNS provides a mapping function between local names in a program (i.e., small integers naming 'slots' in the LNS) and globally unique objects. More than this, however, the rights lists in each item define the permissible access rights of this program at this instant.

Thus far we have described two essential elements of the protection mechanism, objects and items. The third and final element is the rule governing the passing from one execution domain to another and how protection changes at this interface. The execution domain is, at any instant, defined by the current LNS and an LNS is uniquely associated with an invocation of a procedure; thus, execution domains change precisely when a procedure is entered or exited. The kernel provides two primitive functions, CALL and RETURN which allow a procedure to invoke a new procedure or to return to the current procedure's caller.

The essential function of the CALL mechanism is to instantiate a procedure: to create an LNS for its execution domain, and to transfer control to the code body. The essential aspect of the CALL mechanism for the present, however, is its parameter passing/checking mechanism. Since a procedure is an object it has an item part; this item part serves as the prototype for the procedure's LNS when it is instantiated (i.e., the procedure is CALLed). Thus the item part of a procedure contains items which reference the caller-independent capabilities of any invocation of the procedure.

In addition, the item part of a procedure may contain parameter templates for items which will be passed as actual parameters when the procedure is CALLed. The template contains a type attribute, which specifies the required type of the corresponding actual parameter. (One can also have a template which accepts any type.) If a type mismatch occurs, the call is not permitted and an error code is returned to the caller. If the types agree, the rights are then checked, using a special field present only in templates called the "check-rights field". The rights contained in the actual parameter item must include the rights specified in the check-rights field of the template; otherwise, a protection failure occurs and the call is not permitted. If the caller's rights are adequate, an item is constructed in the (new) LNS which references the object passed by the caller, but which contains a rights list specified by the template. (A template has a "regular" rights field distinct from the check-rights which specifies the rights which the callee (i.e., the procedure) will need to operate on the actual parameter.) This implies that a callee may have greater freedom to operate on an object than the caller who passed it as a parameter, but the caller can in no way obtain that freedom for himself (since the additional rights are present only in the callee's LNS - the caller's LNS is unchanged). Also, by appropriate use of the check-rights and type fields in a template, the creator of a procedure can implement arbitrary type-dependent protection checking. (This follows from the observation that the check-rights field is not interpreted by the kernel; the interpretation of the auxilliary rights is up to the user.)

## An Example

In this section we present an example which demonstrates the power of the protection mechanism described above to accomplish in a natural way a kind of protection wwich can be achieved in existing systems, if at all, only by rather artificial devices. While the specific example is itself somewhat artificial, it has *very* realistic counterparts.

Consider the case of a research worker who, being a diligent fellow, wishes to keep himself abreast of the relevant literature in his field. Also having access to a computer, this researcher has written some programs to maintain an annotated bibliography on that computer. The programs permit him to update the bibliography either by inserting new entries or changing existing ones; he may also print the bibliography in total, or selectively on any one of several criteria; he may also wish to completely erase an entire bibliography occasionally. In addition, our hero has organized both the programs and the bibliography structure to be very efficient. It's a nice system of programs, indeed!

Now, in the fullness of time, the researcher decides that it would be to his advantage to allow his colleagues, and perhaps his students, to use his programs and his bibliographies. In addition to creating their own independent bibliographies, the colleagues may be able to add new entries to the researcher's own or to add annotations which (may) provide the researcher with additional insights. He is concerned, however, about several aspects of the protection of both his programs and data:

1. No one, except himself, should be able to erase his bibliographies.

2. He worked hard on his system of programs and he would not like anyone else to copy or modify them. In any case his supervisor has informed him that since the programs were developed on the employer's time, the employer is considering selling them as a proprietary package.

3. Some of the items cited in his own bibliography were written by his colleagues. His annotations are occasionally cryptic, and he would prefer that they were not read by everyone. He would like to choose selectively who may read the annotations.

4. The data structures used to contain the bibliography items are highly optimized and 'delicate'. He would like to insure that when an update is done, the data structures are correctly manipulated.

5. From time to time he changes the programs; either to correct errors or to add new features. For a period of time after the changes are made he would like to allow only a small, sympathetic subset of his (growing) user community to use the new versions of his programs.

6. He suspects that after he has allowed others to use his programs and build their own bibliographies they will share some of his concerns, e.g., items (1), (3) and (5). In particular, they will not want _him_ to be able to erase or examine their bibliographies, or to force a new version of the programs upon them.

Several of our researchers' concerns can, of course, be handled by most 'reasonable' protection systems; others, however, cannot. The most straightforward implementation of the bibliography would be to store the bibliographic information in a single file*; therefore let us frame the discussion in that context.

Since most file systems only protect read access to an entire file there is no way to enforce selective printing, i.e., to distinguish between the accessors who may print the entire file and those which may not print the annotations. Similarly, undifferentiated read access may permit an unscrupulous user to dump an entire bibliography file, determine its structure, and thus compromise the proprietary nature of the programs.

Undifferentiated write access implies analogous problems. Clearly the operation of updating the file implies writing on it; in fact it may conceivably imply a massive reorganization in order to maintain the 'optimal' data structure.

The concept of 'ownership', and its corollary privileges, present in many

-----------------

*Some, but not all, of the problems raised can be solved by an esoteric multi-file structure for the bibliography; however, these solutions violate the 'naturalness' criterion so will not be mentioned.

extant systems may imply that the user of this system cannot protect himself (unless he takes special, explicit precautions) from examination of his bibliographies by the author of the system and/or from unexpected alteration of the system. In particular he may not be able to protect himself from alteration in ways which penetrate the security of his bibliographies.

Now let us consider the 'natural' implementation of the bibliography system in Hydra and how this implementation overcomes the problems mentioned above.

Clearly a bibliography is a new type of virtual resource. Therefore we would create a new object type; call it BIBLIO. In fact, of course, we will want to use existing file mechanisms to represent bibliographies. In all likelihood an instance of a bibliography object will have an empty data part and its item part will merely consist of a single item which references a file object.

Even though the representation of a bibliography is a file, file operations are not applicable to bibliography ojects; they are applicable only to file objects. we can create new operatons (procedures), however, which are applicable to bibliography objects, for example:

| | |
|---|---|
| $U(\beta,p_1,\ldots,p_n)$ | Update |
| $P(\beta,p_1,\ldots,p_m)$ | Print |
| $PWOA(\beta,p_1,\ldots,p_m)$ | Print WithOut Annotations |
| $E(\beta)$ | Erase |

In each of these $\beta$ must be an item which references a bibliography object and the $p_i$'s further specify the nature of the update, print, etc., to be done. (Notice that even though P and PWOA are distinct procedures, in the sense of being distinct objects, they need not necessarily have distinct code bodies; that is, the item part of each of these procedure objects, their prototype LNS's, may reference some or all of the same page objects.)

For simplicity, let us assume that each of the procedures above is uniquely associated with a single bit in the 'auxiliary rights' field of an item which references a bibliography object; denote these bits by the lower case version of the procedure name, i.e., u, p, etc. Thus in order to validly execute 'CALL $U(\beta,\ldots)$' it is first necessary that $\beta$ be a reference to a bibliography object, and second that the 'u' bit of $\beta$ be set.

Figure 2 illustrates (incompletely) a situation involving several 'users' and several bibliographies which might exist at some instant of time. Rectangular boxes denote objects. Directed arrows illustrate item references, and the lower case letters along these arrows signify which of the auxilliary rights bits are set in these items.

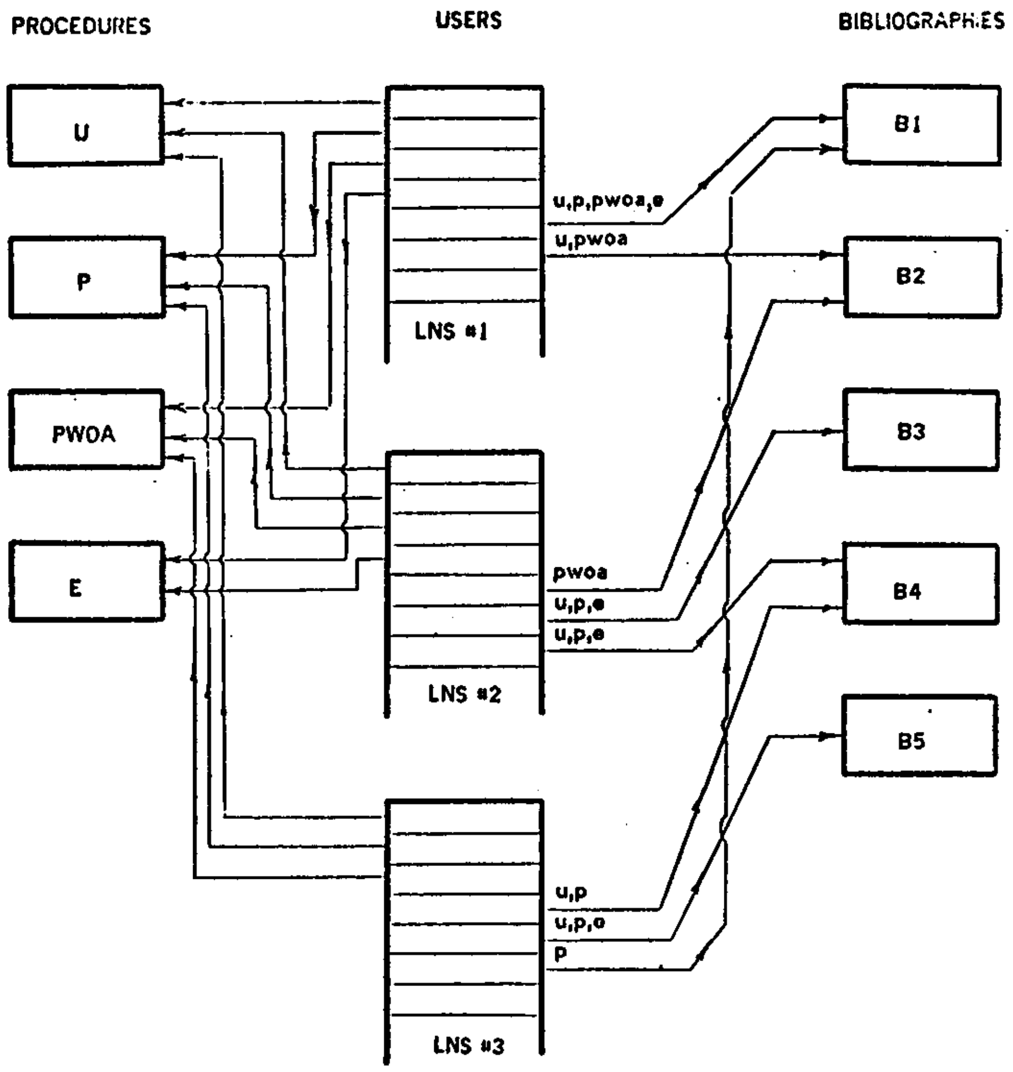
The following kinds of information may be gleaned by inspection of Figure 2:

1. User 1 may access all of the procedures U, P, PWOA, and E. He may also access bibliography objects B1 and B2. He may perform any of the operations U, P, PWOA, and E on B1, but he may only perform U and PWOA on B2.

2. User 2 may also access all of the procedures and, in addition, may access three bibliography objects: B2, B3, and B4. He may only perform PWOA on B2, but may perform U, P, or E on B3 and B4.

3. User 3 may only access three of the procedures; he does not have a reference to E. He may access three bibliography objects - B1, B4, and B5 and may, in principle, perform U or P on B1, and U, P, or E on B4, and P on B5. Notice, however, that the right, in principle, to perform E on B4 is useless to him since he does not have a reference to E.

FIGURE 2: BIBLIOGRAPHY EXAMPLE



It should now be clear that each of the protection concerns expressed by our friend the researcher is neatly handled by this scheme. For example,

1. Since the operation of printing is the protected 'right' in the system rather than the act of reading, it is possible to distinguish between printing the entire bibliography and printing it without annotations. Moreover, since the concept of 'read' is not defined with respect to bibliographies at all, it is simply impossible for someone to examine the representation of a bibliography object and determine its structure; the proprietary nature of the system is therefore insured.

2. Similarly, since the operation of updating a bibliography is distinct from that of writing the file which represents it, the internal integrity of the data structure is guaranteed (at least if the procedure U works correctly).

We would like to make one more point with respect to the example before leaving it. The conventional view with respect to sharing resources is that there are precisely two cases: (1) the shared resource is passed to another 'user' - in which case the 'rights' which may be passed must be a subset of those of the passer, or (2) the shared resource is passed to the 'operating system' - in which case the set of 'rights' expands drastically. We reject both of these cases as inadequate to serve as the basis of a truly secure system.

As discussed in the previous section, the rights acquired by a procedure to an item passed to it as an actual parameter are obtained from the template in that procedure's prototype LNS. These rights may be a subset, superset, or totally disjoint from those of the caller. The point is that a procedure is invested with those rights, and only those rights, which it needs to do its job. (By way of analogy, I am not permitted to repair my telephone. I am permitted, however, to invoke an operation, namely a telephone repairman, that can repair it. The repairman inherently has the right to repair telephones; he does not, however, have access to my particular telephone until I grant him access to it.)

## Systems and Subsystems

The previous sections describe how the kernel supports the notion of an object, operations on objects, and protection. It is now time to question to what extent these mechanisms permit and facilitate the construction of operating systems; part of the response is implicit in what has already been described, and part is not.

An 'operating system', in the sense of a monolithic entity which provides various facilities to the user, is not an appropriate image of a user environment as it would exist in the Hydra context. Rather, a user environment consists of a collection of resources (objects) of various types and procedures which operate on them. The environment in which one user operates may or may not be the same as that for another user; it may be totally different, or may partially overlap.

It is entirely possible, for example, that at some point in the evolution of the Hydra environment several different 'file systems' will have been devised. Each such 'system' will consist of a distinct object type to denote the style of file supported by that system, and a collection of operations (procedures) for dealing with that style of file. The various styles of files, for example, may correspond to different access methods, different queueing strategies for dealing with disk, or different security policies. An individual user may use any one of the systems, or because the various systems are optimized along distinct dimensions, he may use more than one. Similar comments, of course, apply to every type of facility provided by an operating system, e.g., command interpreters, synchronization mechanisms, etc.

It should be clear from the discussion above that the Hydra mechanisms provide a consistent framework within which the 'virtual machine' visble to the individual user may be easily defined and modified. The second goal of an operating system, however, is to optimize the utilization of the underlying hardware. This optimization involves policy decisions which must be made in 'real time'. (Two prime examples of these decisions are those involving scheduling processes and processors, and those involving paging operations.) The primary goal of this section is the description of how these policy decisions have been factored out of the kernel and framed in the <u>object</u> <u>procedure</u> context.

While the goal of separating mechanism and policy is a laudable one, it is impossible to achieve a complete separation in practice. Any mechanism limits the set of policies to those which are feasible with respect to that mechanism. Therefore the original goal transmutes into one which attempts, by careful choice of the mechanism, to limit the class of feasible policies as little as possible. Further, we suspect a universal law which states that flexibiliiy, in this case with respect to the class of feasible policies, has a cost. Therefore the practical goal is to minimize the constraints on feasible policies subject to the real time constraints imposed by the need for a policy decision. The result has been to limit both the nature of feasible policies and the times at which they can be invoked.

Notice, too, that the kernel need not have any explicit knowledge of those object types and procedures which implement the user-visible aspects of an operating system. The opposite is true for those which implement policy; that is, the kernel, and only the kernel, may be aware that a policy decision must be made. Thus the kernel must either make the decision itself or know how to invoke someone to make it.

The uniform reaction of a Hydraphile when presented a problem is, "Well, create an object which....". That is precisely how we shall introduce policy decision features.

A 'subsystem' object is another object type recognized by the kernel. A subsystem object contains references to procedures which implement those policy decisions required by the kernel. Each process object contains a reference to such a subsystem object; hence, whenever a policy decision relative to a given process is required the kernel executes a call on the appropriate policy procedure.

It is worthwhile to make two observations relative to this scheme: first, since subsystem objects are associated with a process, it is possible that several subsystems are coextant -- each making policy decisions appropriate to distinct classes of jobs; second, this is another example of the non-hierarchical nature of Hydra.

Rather than detail the various policy/mechanism issues, which are the subject of another report [Wu73], we shall simply illustrate the nature of these issues with a single example; that of scheduling.

Scheduling a process for execution involves three separable tasks: (1) determining the process's importance relative to others, (2) preparing the process for execution, e.g., making sure that the relevant pages are 'in-core', and (3) performing the mechanical jobs involved in actually starting the process when the appropriate time arrives. The first two of these are performed by policy procedures in a subsystem. Only the last is done by the kernel. More specifically, the subsystem specifies to the kernel a set of parameters which control a simplistic multiplexing algorithm among processes which the subsystem has specified are eligible for execution. Short term, 'time critical' decisions are made by the kernel on the basis of these parameters. Longer-term decisions are made by the subsystem. In particular, one parameter allows the subsystem to specify when it should be consulted again; thus, in the limiting case, the subsystem may intercede on each scheduling decision.

## Conclusion

An operating system, even the kernel of one, is a large undertaking. It involves many interrelated decisions. Indeed we believe that the consistency and cleanliness of this interrelation is more important to the ultimate utility of the system than any of the individual decisions. It is this aspect of the Hydra design that we feel is most important.

Acknowledgments

It is difficult to give proper credit to the sources of all the ideas presented above. Although we have felt free to change terminology, the work of Dennis [DvH66], Dijkstra [Dij68b], Hansen [Br70], and especially Lampson [La69] and Jones [Jo73] have had a significant impact. The ideas of these individuals will clearly show through to those who are familiar with them. The remaining ideas and the cement which holds the design together emerged in discussion between the authors.

## References

[Br70] Brinch-Hansen, P., "The Nucleus of a Multiprogramming System," CACM 13 (4/70), 238.

[Dij68a] Dijkstra, E.W., "The Structure of THE Multiprogramming System," CACM 11,5 (4/68), 341-6.

[DvH66] Dennis, J. B. and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations," CACM 9,3 (3/66), 143-55.

[Dij68b] Dijkstra, E. W., "Cooperating Sequential Processes," Programming Languages (ed. F. Genuys), Academic Press (1968), 43-112.

[La69] Lampson, B. W., "Dynamic Protection Structures," Proc. AFIPS Conf. 35 (FJCC 1969).

[Jo73] Jones, A. K., Protection Structures, Ph.D. Thesis, Carnegie-Mellon University, 1973.

[Pa72] Parnas, D. L., Response to Detected Errors in Well-Structured Programs, Carnegie-Mellon University, Computer Science Department report, July, 1972.

[Pa71] Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules, Carnegie-Mellon University, Computer Science Department report, August, 1971.

[Wb72] Wulf, W. A. and Bell, C. G., "C.mmp - A Multi-Mini-Processor," Proc. AFIPS FJCC 1972, 765-77.

[DDH72] Dahl, O. J., Dijkstra, E. W., and Hoare, C.A.R., Structured Programming, Academic Press, 1972.

[Wu73] Wulf, W. et al., Policy/Mechanism Separation in the HYDRA System, Carnegie-Mellon University, Computer Science Department report, to be published.