Operating Systems — C. Weissman, Editor

HYDRA: The Kernel of a Multiprocessor Operating System

W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack
Carnegie-Mellon University

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.

Key Words and Phrases: operating system, kernel, nucleus, protection, security
CR Categories: 4.3, 6.2

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

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

Authors' address: Department of Computer Science, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213.

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-visible 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 [9]). C.mmp permits the connection of 16 processors to 32 million bytes of shared primary memory through a cross-bar switch. The processors are any of the various models of PDP-11[1] 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 for unique-name generation (see below) and also broadcasts a central time base to all processors. Relocation hardware on each processor's bus provides mapping of virtual addresses on that bus to physical addresses in shared primary memory.

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 basic facilities has been called the *kernel* or *nucleus* [1] 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

[1] Manufactured by Digital Equipment Corporation.

few have been built, both the potentials and problems of these 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 [1] has presented cogent arguments for this separation.) Such separation 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.

3. *Integration of the design with implementation methodology.* It has been observed that the structure of extant operating systems bears 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 hybrid of structured programming as advocated by Dijkstra and others [2] and the modularization philosophy of Parnas [8].

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 [3]. 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 through the system, and not to be applied only to specific entities (e.g. files). The idea of capabilities (in the sense of Dennis [5]) 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 most critical hardware resources in C.mmp suggests the possibility of highly reliable operation. Our desire is to provide commensurate reliability in the software. Reliability not only requires that the system be correct, but that it be able to detect and recover from errors that do exist—as the result of hardware malfunction, for example.

Defining a kernel with all the attributes given above is difficult, and perhaps impractical at the current state of the art. It is, nevertheless, the approach taken in the HYDRA system. Although we make no claim either that the set of facilities provided by the HYDRA kernel is minimal (the most primitive "adequate" set) or that it is 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 set of functions provided by HYDRA will enable the user of C.mmp to create his own operating environment without being confined to predetermined 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 important, what does not. Nonspecific 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 an "abstract 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 abstract.

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. The remainder of this paper focuses on these issues, thus deemphasizing several of the other issues raised earlier.

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;[2] 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 analogous 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 [7] 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 capabilities (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. 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.

Up to this point the term "capability" has been used in a somewhat loose and intuitive sense; subsequently it will be used in a technical sense. A *capability* consists of a reference to an object together with a collection of "access rights" to that object. Possession of a capability is taken as prima facie evidence that the possessor may access the object in the ways, and in *only* the ways, described by the capability. Capabilities themselves are manipulated only by the kernel; hence it is impossible to "forge" a capability.

A procedure object may contain *templates* 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" capabilities 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, a capability 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 enforced 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 capabilities supplied by the caller and determines whether all pro-

[2] It should be noted however, that the cost of entering a HYDRA procedure is considerably greater than, say, a Fortran subroutine. We do not expect that simple subroutines, such as SIN, would use the HYDRA mechanism until better hardware is provided. The reader should visualize a procedure as replacing relatively large code units and/or at points where protection environments must change.

tection 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.

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 outside, 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 interprocess communication and synchronization by providing elementary message buffering primitives and Dijkstra-style semaphore operations. These facilities have been treated in [4] and elsewhere and will not be discussed here.

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.

In our view, protection is a mechanism; security is a policy. A system utilizing such a 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 highly secure system 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 an 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. 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 nonhierarchical 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 a *capability*. Each object has a *unique name*, a *type part*, and a *representation* (consisting of a *capability 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 capabilities (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. (By convention the type of these representatives is the name of a special distinguished object whose name is TYPE and which names itself in its type field.) 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[3] 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 normally un-

[3] The deletion mechanism is not essential to an understanding of HYDRA, therefore this function of the kernel will not be discussed further.

interpreted by the kernel), and references to other objects. These two kinds of information are stored in the *data* and *capability* 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 capability part of an object may be manipulated only by invocation of kernel functions.

The fact that every object in HYDRA may contain capabilities referencing other objects is a significant departure from other capability-based systems [5, 7], and is a major factor in satisfying the flexibility goal of the design. It permits new object types to be represented wholly or partially in terms of existing types while preserving the protection philosophy through each level of representation. An understanding of the use of the capability-part of an object to represent new objects is essential to an understanding of the HYDRA philosophy; we will return to it after discussing capabilities, procedures, and the CALL mechanism further.

As mentioned several times previously, in addition to supporting objects, HYDRA supports *references to objects* (called *capabilities*). Capabilities are, however, more than simple pointers. Together with the actual/formal parameter "derivation" of the CALL mechanism (for crossing execution domains), capabilities are the key to the protection mechanism. As such, capabilities may only exist in the capability part of an object and may be directly manipulated only by the kernel.

Each capability includes information detailing the operations which may be performed on the object referenced by the capability. Whenever an operation is attempted on an object, the requestor supplies a capability 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 rights). 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 and capabilities. Accordingly, we partition the rights list of a capability into two mutually exclusive sets—the type-independent rights (called *kernel rights*), and the type-dependent rights (called *auxiliary rights*). A "right" in either set grants permission to pass the capability 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 auxiliary rights.

The notions of *object* and *capability* now permit us to be more specific about one of the object types mentioned earlier—the LNS. At any instant, the execution environment (domain) of a program is defined by an LNS object associated with it. The capability part of the LNS contains references to objects which 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 capability 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 *capabilities*. 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, or "derivation" alluded to previously. Since a procedure is an object, it has a capability part; this capability part serves as the prototype for the procedure's LNS when it is instantiated (i.e. the procedure is CALLed). Thus the capability part of a procedure contains capabilities which reference the caller-independent capabilities of any invocation of the procedure.

In addition, the capability part of a procedure may contain parameter *templates* for capabilities 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 capability 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, a capability 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 rather general type-dependent protection checking. (This follows from the observation that the check-rights field is not interpreted by the kernel; the interpretation of the auxiliary rights is up to the user.)

Path Names and the Walk Right

We have avoided enumerating the specific primitives provided by HYDRA; this is intentional since we are more concerned with philosophy than with implementation. However, a few primitives must be discussed in order to fairly present the philosophy—one is *walk*. The *walk* primitive is a one-level coercion which, given a capability and a nonnegative integer, produces the capability which occupies the specified position in the capability part of the object named by the parameter capability. The *walk* primitive, like all kernel primitives, is an access right protected by the "kernel rights" bits in a capability.

Because of the *walk* primitive, the environment of a procedure does not consist of the objects named by capabilities in its LNS alone. Rather, it is the closure of the set of objects reachable along a path (originating in the LNS) such that every capability along the path (except possibly the last) grant the *walk* right. Recognizing this, all of the kernel primitives accept path names as parameters and the *walk* right is checked at each step along the path. The use of path names and walk rights effect a significant reduction in the number of capabilities needed in an LNS. Far more important, however, is that the *walk* right (or rather the lack of it) is used to prevent access to the representation of an object. The amplification *walk* rights is one of the most common when entering a procedure which implements operations on a particular object type.

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 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.

The flexibility necessary to obtain this form of extension results from both the representation of objects as both data and capabilities, and from the nature of the protection mechanism. Let us return, for a moment, to the issue raised earlier concerning the use of the capability-part of an object in the representation of new object types. First, an instance of an object may be created by invoking a kernel primitive *create* and passing to it a capability referencing the representative of type of object one wishes to create. In particular, then, invoking *create* with a capability referencing the distinguished object named TYPE will create the representative of a new type class. Subsequent calls on *create* passing capabilities referencing this new type representative will create instances of the new class of objects.

Suppose, for example, that at some instant at least the types[4] FILE, SEMAPHORE, and PROCEDURE exist. We wish to introduce the notion of a general directory object, which defines a mapping from print names to capabilities. One idea is to list the print names in the data part, and associate each such name with a capability in the capability part. (Clearly, a more sophisticated structure could be devised.) To support the mutual exclusion needed to permit shared access to directories, we also include a capability for a SEMAPHORE in the DIRECTORY's capability part. We may now create PROCEDURES which "understand" the representation just described and which provide typical directory functions. Notice that we have not restricted the capabilities contained in a DIRECTORY to be of the same type; indeed, the PROCEDURES which maintain directories may specifically allow directories of FILES, or PROCEDURES, or of arbitrarily mixed types. In fact, a directory may contain capabilities for other directories, which suggests the potential for general path names in a directory structure, and inherent search rules. Many variations are possible; the important idea is that the nature of the directory "subsystem" is completely user-specified.

[4] We have tried to use suggestive names for object and type names in the examples. In practice, of course, the unique names of objects have no visible representation; they are 64-bit values.

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 which 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 references 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 references 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), above. 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 the concerns of our researchers 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;[5] 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 who 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 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 capability part will merely consist of a single capability which references an ordinary file object.

Even though the representation of a bibliography is a file, file operations are not applicable to bibliography objects; they are applicable only to file objects; we can create new operations (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, β must be a capability which references a bibliography object and the p_i's further specify the nature of the update, print, etc., to be done. (Notice

[5] Some, but not all, of the problems raised can be solved by an esoteric multi-file structure for the bibliography; however, since these solutions violate the "naturalness" criterion, they will not be mentioned.

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 capability 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 a capability which references a bibliography object; denote these bits by the lowercase version of the procedure name, i.e. u, p, etc. Thus in order to validly execute "CALL U(β,...)" it is first necessary that β be a reference to a bibliography object, and second that the "u" bit of β be set.

Figure 1 illustrates (incompletely) a situation involving several "users" and several bibliographies which might exist at some instant. Rectangular boxes denote objects. Directed arrows illustrate capability references, and the lowercase letters along these arrows signify which of the auxiliary rights bits are set in these capabilities.

The following kinds of information may be gleaned by inspection of Figure 1.

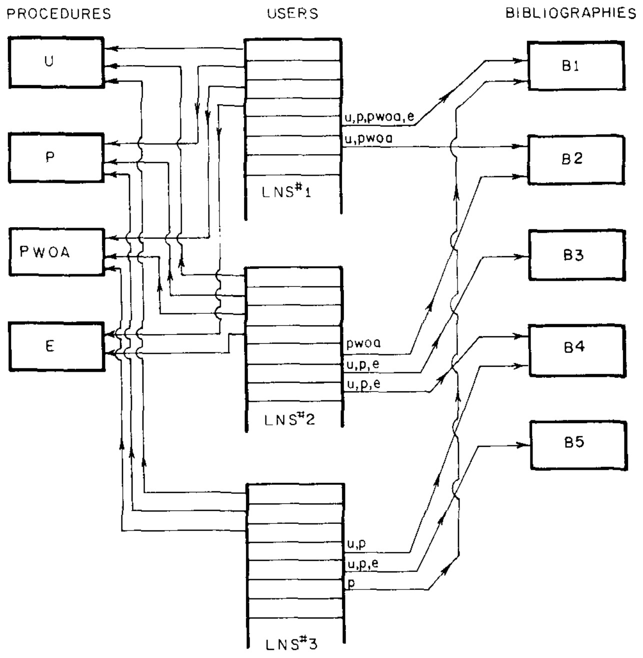
1. User #1 may access all of the procedures U, P, PWOA, and E. He may also access bibliography objects B1 and B2. He may perform any of the operations U, P, PWOA, and E on B1, but he may only perform U and PWOA on B2.
2. User #2 may also access all of the procedures and, in addition, may access three 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 B4, and U, P, or E on B5, and P on B1. Notice, however, that the right, in principle, to perform E on B5 is useless to him since he does not have a reference to capability E.

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 two more points with respect to the example before leaving it. The conventional

Fig. 1. Bibliography example.



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 owned by the "operating system"—in which case the set of "rights" expands drastically when an operating system function is invoked. We reject both of these cases as inadequate to serve as the basis of a truly secure system.

On several occasions we have referred to our rejection of "ownership" as a design concept. Consider the consequences of this statement in the context of the example; specifically, who "owns" object B1? We cannot tell from the static picture who originally created the object: indeed the creator may not be able to reference it any longer. We do know that user #1 has more rights to B1 than does user #2; however, even user #1 can only invoke BIBLIO-specific operations on B1—he cannot, for example, manipulate the representation of B1.

Nor does the BIBLIO subsystem "own" B1. As discussed in the previous section, the rights acquired by a procedure to a capability 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. Note that the procedures comprised within the BIBLIO system do not have access to *any* BIBLIO objects except as parameters. (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.)

Conclusion

An operating system, even the kernel of one, is a large undertaking which involves many interrelated decisions. Indeed we believe that the consistency and cleanliness of this interrelation are 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.

A word or two concerning the status of the system and our experience with it seem appropriate. At the time of this writing the kernel has been operational for about nine months. The existence of a kernel, however, does not make a system usable, nor does it relieve the implementers from the responsibility of developing at least one version of the subsystems, e.g. a file system, which users require. We are currently developing these subsystems. The extent of our experience to date is that the facilities provided by the kernel do indeed significantly simplify the construction of these subsystems.

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 works of Dennis [5], Dijkstra [4], Hansen [1], and especially Lampson [7] and Jones [6] have had 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.

Received June 1973; revised December 1973

References

1. Brinch-Hansen, P. The nucleus of a multiprogramming system. *Comm. ACM 13,* 4 (Apr. 1970), 238–241.
2. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. *Structured Programming,* Academic Press, New York, 1972.
3. Dijkstra, E.W. The structure of THE multiprogramming system. *Comm. ACM 11,* 5 (May 1968), 341–346.
4. Dijkstra, E.W., Cooperating sequential processes. In *Programming Languages,* F. Genuys, (Ed.), Academic Press, New York, 1968, pp. 43–112.
5. Dennis, J.B., and Van Horn, E.C. Programming semantics for multiprogrammed computations. *Comm. ACM 9,* 3 (Mar. 1966), 143–155.
6. Jones, A.K. Protection in programming systems. Ph.D. Th. Carnegie-Mellon U., 1973.
7. Lampson, B.W. Dynamic Protection Structures, Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J. pp. 27–38.
8. Parnas, D.L. On the criteria to be used in decomposing systems into modules. Comput. Sci. Dep. Rep., Carnegie-Mellon U., 1971.
9. Wulf, W.A., and Bell, C.G. C.mmp—a multi-mini-processor. Proc. AFIPS 1972, FJCC. Vol. 41, AFIPS Press, Montvale, N.J. pp. 765–777.

Information Retrieval and Language Processing — C.A. Montgomery, Editor

An Information-Theoretic Approach to Text Searching in Direct Access Systems

Ian J. Barton, Susan E. Creasey, Michael F. Lynch, and Michael J. Snell
University of Sheffield

Using direct access computer files of bibliographic information, an attempt is made to overcome one of the problems often associated with information retrieval, namely, the maintenance and use of large dictionaries, the greater part of which is used only infrequently. A novel method is presented, which maps the hyperbolic frequency distribution of text characteristics onto a rectangular distribution. This is more suited to implementation on storage devices.

This method treats text as a string of characters rather than words bounded by spaces, and chooses subsets of strings such that their frequencies of occurrence are more even than those of word types. The members of this subset are then used as index keys for retrieval. The rectangular distribution of key frequencies results in a much simplified file organization and promises considerable cost advantages.

Key Words and Phrases: text searching, information theory, file organization, direct access, information retrieval, character string, bit vector

CR Categories: 3.42, 3.70, 3.73, 3.74, 5.6

Introduction

Information dissemination services providing computer based searches of bibliographic data bases have until recently depended largely on serial searches of magnetic tape files [1, 2]. However, the need to process large

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This work was supported by the Office of Scientific and Technical Information of the British Government. Authors' address: Postgraduate School of Librarianship and Information Science, University of Sheffield, Western Bank, Sheffield, S10 2TN, U.K.