CMU-CS-78-102

An Extensible File System for HYDRA

Guy Almes and George Robertson

Department of Computer Science
Carnegie-Mellon University

February 9, 1978

Abstract: An extensible file system has been designed and implemented for Hydra, an advanced capability-based operating system. This system demonstrates three notable advances to subsystem design:

1. It provides a protected and efficient implementation via user-level code of functions ordinarily implemented as part of a conventional system's monolithic privileged section,

2. It provides practical solutions to two protection problems, the Modification Problem and the Confinement Problem, for users of the file system, and

3. It provides separation of mechanisms for data representation from mechanisms for protection and synchronization, thus allowing an extensible family of subfile systems to evolve.

This paper treats the design and implementation of the Hydra File System and reflects on its implications for subsystem design and implementation.

Keywords: protection, capability, files, confinement, data abstraction.

This work was supported by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research. This paper also appears in the Proceedings of the Third International Conference on Software Engineering;

1. Introduction

Among the more important concepts in systems, languages, and programming methodology during the last several years are those of data type [Hoare 72], clean control structure [Dijkstra 72, Hoare 74], and capability-based addressing [Fabry 74]. These concepts are contributing to an increasingly coherent object-oriented view of programming, manifested in the language developments of the Alphard and CLU groups [Jones/Liskov 76], in the systems work of Hydra (at Carnegie-Mellon [Wulf 74, Wulf 75]) and similar systems (e.g., at the University of California [Lampson/Sturgis 76], Cambridge [Needham 72], IRIA/LABORIA [Ferrie 76], Plessey Telecommunications [England 74], SRI [Robinson 75], and others at Carnegie-Mellon University [Habermann 76, Jones 77]), and in the continuing work on the Multics system [Schroeder 77]. This paper explores the success of the Hydra system in realizing an often-cited claim of such systems [Cohen/Jefferson 75]: the ability to provide an adequately protected and efficient implementation via user-level code of functions ordinarily implemented as part of a conventional system's monolithic privileged section. Specifically, it explores the design and implementation of an extensible file system using only the protection mechanisms available to the ordinary users of Hydra.

Section 2 describes the goals for a file system for Hydra. Section 3 provides a detailed description of the design and implementation of this file system, emphasizing its use of the Hydra protection mechanisms. Section 4 closes the paper with a critical evaluation of the system in light of the claims made in [Cohen/Jefferson 75] and the goals cited in Section 2.

2. Goals for a Hydra File System

The primary goals for the file system were (1) implementation without special privileges, (2) practical solutions to two protection problems, and (3) separation between representation and protection to allow extensibility. Each of these goals is discussed in the following sections.

2.1. Implementation without Privilege

The principal goal of the project was to produce a usable file system without using privileges denied to ordinary system users. The practicality of building subsystems in this way was recognized by Fabry in the context of a Directory System; he viewed a capability mechanism as a crucial tool:

> Computer systems which do not allow processor-independent interpretations for references to the set of objects to which a directory name might correspond do not allow virtual-computer processing devices to represent a directory as a data structure. Such systems, if directories are to be available at all, must represent the directory as a data structure hidden away within the operating system...
>
> With list-structured addressing using capabilities, a directory can be represented very simply as a table of names and a table of capabilities which correspond to the names. [Fabry 71, p. 105]

The basic technique for accomplishing this in Hydra is outlined in [Cohen/Jefferson 75] and is reviewed in Section 3.1. This ability is shared by several other capability-based systems, notably CAL [Lampson/Sturgis 76] and the Plessey System 250 [England 74].

2.2. The Modification and Confinement Problems

A second project goal was to allow practical solution of two protection problems described in [Cohen/Jefferson 75]:

1. The Modification Problem. Suppose a user wants to access a file in a read-only fashion. Suppose further that he wants to restrict the file system from modifying the file in any way. Hydra allows a capability for such a file to be passed to the file system Procedures with this constraint satisfied. [Cohen/Jefferson 75, Section 3.2]

2. The Confinement Problem. Conversely, suppose a user has a sensitive file and wants to insure that, while accessing the file through the file system, no

> information from the file will be transmitted to the outside world. Hydra allows the file system to be called in a confined mode, in which only the parameters explicitly passed and the objects local to the Procedure invocation can be modified. [Cohen/Jefferson 75, Section 3.5]

The Confinement Problem was first posed in [Lampson 73]; very few systems even attempt solutions to it. The only such systems known to the authors assign a sensitivity-level, from a partially ordered set of levels, to each datum and insure that information flows only in ways determined by this partial ordering [Lipner 76, Denning 76]. This scheme models the military security system and thus represents a special case of the general problem stated by Lampson.

2.3. Separation of Representation and Protection

Another basic goal was to separate the responsibility for uniform and correct synchronization and protection from that for efficient representation and transput. This separation allows extensibility of representation. We recognized several possible file representations, each with a different intended set of uses. One file representation might use stored line numbers to implement text files, while another might use a simple byte stream to implement binary files. In these cases, there is a large common set of operations that users will perform. In addition, file protection and synchronization (e.g., mutual exclusion on file write) and file access record maintenance are common across these different representations. It should be possible to form a file system that supports protection, synchronization, access record maintenance, and the common operations, and leave the actual representation to sub-file systems. The problem of extending the file system to support new file representations is then reduced to a problem of supplying one of these sub-file systems, which should be a much easier task.

3. Design and Implementation

This section describes the design of the File System along with some details of its implementation. Before discussing the design, we will provide a brief review of Hydra's protection mechanisms.

3.1. Review of Hydra Protection Mechanisms

The reader is urged to refer to the broad summary of Hydra in [Wulf 74] and the more detailed paper on Hydra's protection mechanisms [Cohen/Jefferson 75]. As noted in these papers, a central notion in Hydra is that all resources and data structures are objects. These objects are addressed only through capabilities, which provide an addressing mechanism that is absolute and context-independent in the sense of [Fabry 74]. Objects can contain both data and capabilities, thus allowing complex data structures to be built. Also, each object has a unique 64-bit Name [1] and a 64-bit Type. Each capability contains two fields: the Name of the object and a set of Access Rights. These Access Rights are in three groups: (1) the Generic Rights (e.g., $GetDataRts to read data from an object) control the operations applicable to objects of any Type; (2) Un-Amplifiable Access Rights (e.g., $UnconfineRts and $ModifyRts, discussed in section 4.2) solve specific protection problems; and (3) the Auxiliary Rights (e.g., $FSCopyRts to read or copy a File object) control Type-specific operations. Each distinct object Type is supported by a different Subsystem, comprised of a description of the Type's data format and a set of Procedures which implement Type-specific operations. Hydra itself implements several basic Types (e.g., Type, Procedure, Process, Page, Semaphore, and Port); it can thus be regarded as a uniform capability mechanism together with a few built-in Subsystems.

A Hydra Subsystem is implemented in two steps: the description of the format of the objects of the new Type and the provision of Procedures that implement the Type-specific operations.

The description of format is accomplished by the creation of a new object whose Type is Type, and whose Name is "File" (or "Directory" or "Connection"). The static attributes of File objects (e.g., the number of data words and capabilities they can hold) are stored in this Type object. This Type object can then be used to create objects whose Type is File, but whose Name is "MyTextFile" (or "SortSource" or "SortObject"). This method of object creation clearly identifies each object with a specific Type and data structure.

---

[1] Within this paper, alphabetic Names will be used to denote objects. It should be remembered, however, that the actual Name is the unique 64-bit value provided by Hydra.

The implementation of Type-specific operations is accomplished by the construction of a Procedure for each such operation. Each Procedure (an object of Type Procedure) in Hydra has two important constituents: (1) a set of inherited capabilities for objects that belong to the Subsystem, including Pages that contain the Procedure's code and (2) a set of formal parameters, denoted by special capabilities, called Parameter Templates. When a Procedure is called, a new protection domain (an object of Type LNS, for Local Name Space) is formed. The inherited capabilities are copied directly from the Procedure and the code Pages are made addressable. The formal Parameter Templates are then merged with the capabilities passed by the caller to result in actual parameter capabilities in the LNS. The nature of this merging is critical to the implementation of Subsystems and should be well understood. Any Parameter Template can specify a Type and a minimal set of Access Rights that the caller's parameter must satisfy. Should any of the caller's parameters be of the wrong Type or have insufficient Access Rights, the Procedure Call will fail. A Subsystem will normally make such Parameter Templates available to the entire community, for they simply accomplish the checking of actual parameters and, of themselves, grant no access.

A special kind of Parameter Template, the Amplification Template, is derived from the Type object and kept private to the Subsystem. In addition to Type and Access Rights checking, an Amplification Template can turn on any of the Generic Access Rights (it cannot, however, turn on the Un-Amplifiable Access Rights). Thus, although users of the Subsystem will lack these Generic Rights and cannot manipulate the representation of the object directly, they can manipulate them by invocation of Subsystem Procedures which gain these Generic Rights over the duration of the Call. To explain this Amplification differently, we may say that the Amplification Template signifies the Subsystem's right to control the representation of all objects of its Type in general, but does not grant access to any particular object. On the other hand, the user's capability for a particular object signifies his right to address it, to share it in a flexible way, to use it as a building block for other data structures, and to manipulate it by invocation of the Type-specific operations implemented by Subsystem Procedures.

A Procedure can be invoked via the Call operation if the caller has a capability for the Procedure to be called. Alternatively, the Typecall mechanism can be used, which eliminates the need for the user to obtain an actual Procedure capability. The Typecall is identical to Call, except that the Procedure is specified, not by a Procedure capability, but by (1) a capability for an object, called a Type Representative, of the Subsystem's Type and (2) an integer Index. When the Typecall is invoked, Hydra locates the Type object that corresponds to the Type Representative and uses the Index to locate a capability in this Type object, which must be a Procedure capability. This Procedure is then Called.

Finally, Hydra provides a Port System for inter-Process communication. Using this Subsystem, a Process can send and receive messages from Port to Port along established connections. Like objects, these messages can contain both data and capabilities. The Port System itself handles the necessary synchronization and queueing associated with sending and receiving the messages.

3.2. File System Design

A basic goal of the File System is the separation of responsibility for synchronization and protection from responsibility for efficient representation and transput. The former responsibilities are handled by one fixed File System; the latter by several SubFile Systems [2]. The File System defines and supports the new Type called File. Each SubFile System defines and supports a new SubFile Type. A user's file is an object of Type File, and all accesses to it are initiated through Typecalls to the File System. The File object has a capability for a SubFile object, which contains the actual data stored in the file. The File System handles activities common to all SubFile Systems, and passes representation-specific requests on to the appropriate SubFile System via Typecalls on the SubFile object.

There is a fundamental convention that only the File System and the supporting SubFile System ever have capabilities for SubFile objects. The File System and all SubFile Systems respect this convention; it allows the SubFile Systems to know they are called only by the File System, which will have satisfied protection and synchronization requirements.

The File System makes effective use of both the Typecall and Port mechanisms of Hydra to accomplish transput. Typecalls, using a File object as a Type Representative, are used to Create new Files, to Open or Close them, and to Copy, Query, Compare, or Edit them. These Calls will be described in more detail later. In normal transput, however, the only necessary Calls are Open and Close. Each SubFile System provides a Monitor Process with which the user communicates via messages sent through the Port System. The Open Call to the File System evokes an Open Call to the SubFile System, which establishes the SubFile as an opened SubFile and connects the user's Port to the SubFile System Monitor's Port. All transput is then accomplished through messages to this Port. Using these messages in a manner similar to the classical technique of "double buffering", processing concurrency between the user process and the SubFile Monitor is achieved. When the transput is completed, the Close Call disconnects the user's Port from the SubFile Monitor's Port.

---

[2] Although there may be many different SubFile Systems, each supporting its own Type, any particular File object will always use a particular SubFile System specified upon File creation. Thus, in the rest of the paper, we will often refer to the SubFile System, meaning the SubFile System that supports the subfile of the File under discussion.

3.2.1 File System: Protection and Synchronization

A File is an object of Type File and is effectively manipulable only through the File System Procedures via Typecalls. Its representation contains both capabilities and data. One capability addresses a Semaphore used for exclusive access to the File object. Another addresses the current SubFile (i.e., the SubFile object that represents the current version of the File). The data contain record access maintenance information such as the print name for the File, creation date, last access date, and last modification date.

All user manipulations of files are initiated via one of the following Typecalls to the File System: Create, Open, Close, Copy, Query, Compare, and Edit. We will briefly describe each.

The Create Call takes a File as a parameter. It creates a new object of Type File and initializes the File's maintenance information (e.g., print name and creation date). Using the current SubFile of the File parameter, a SubFile Create Typecall is made to create a new SubFile object. Capabilities for that new SubFile and for a new Semaphore are then placed in the new File. Finally, the new File is returned to the user with all Auxiliary Rights, all Un-Amplifiable Access Rights, but no Generic Rights.

The Open Call takes a File, a Port, and a Job object (used to access the system Scheduler) as parameters. If the file is being 'opened for writing', then appropriate synchronization is done and a new SubFile is created and opened; otherwise the current SubFile is opened. This scheme allows many readers and one writer simultaneous access to the File. The appropriate SubFile System is notified, via SubFile Open Typecall, that the SubFile is being opened. The Job object is passed to the SubFile System for its use. The SubFile Open returns its Monitor's Port, which is then connected to the user's Port. Finally, an object of Type OpenFile (containing information needed by Close) is created and returned to the user.

The Close Call takes an OpenFile object as a parameter. It performs a SubFile Close Typecall, which notifies the SubFile Monitor. If the file had been 'opened for writing', then SubFile Close returns the new SubFile which is then made the current SubFile. The user's Port is then disconnected from the SubFile Monitor's Port. Finally, some maintenance information is updated in the File (e.g., latest access date).

The Copy Call takes a File and produces a physical copy of it (data and capabilities), which is returned to the caller. However, the SubFile is not physically copied since the SubFile of a File can be read and replaced, but never modified.

The Query Call provides a means for the user to interrogate the maintenance information stored in the File. It also provides an open-ended way of obtaining information about the SubFile. The user specifies which field he wishes to see; any field unknown to the File System will yield a request to the SubFile System (via SubFile Query Typecall).

The Compare Call takes two File objects and returns information resulting from comparing their respective SubFiles. No SubFile Compare operation is needed. The information returned indicates whether the SubFiles are identical and whether they are of the same SubFile Type.

Finally, the Edit Call provides an entry into the editor most appropriate for the SubFile (via SubFile Edit Typecall). The SubFile Editor may replace the SubFile in the File by performing a NewSub Typecall on the File. The NewSub Typecall requires as a parameter a capability for the original SubFile to insure that only the SubFile System can make such a modification.

In each of these Calls, the File System does appropriate rights checking using the Parameter Template mechanism provided by the Hydra Call mechanism. It also provides the common kinds of synchronization found in most file systems. For example, a File 'opened for writing' by one user cannot be simultaneously edited by another. In sum, the File System is responsible for all things that SubFile Systems have in common, leaving only representational issues to the SubFile Systems.

3.2.2 SubFile System: Transput and Representation

A SubFile System is the only system which ever has direct access to the data stored in a SubFile object. It has sole responsibility for data representation and access. Each SubFile System is built around a particular SubFile Type and provides Procedures (called from the File System via Typecalls) for Create, Open, Close, Query, and Edit. It also provides a Monitor Process, which cooperates with the SubFile Open and SubFile Close to establish Port System connections with users for transput.

The SubFile Create Procedure creates an empty SubFile in whatever form is suitable for its representation. The SubFile Query Procedure responds to information requests that are representation-specific (e.g., what is the length of the file?). The SubFile Edit Procedure invokes the editor most appropriate for its representation (e.g., line-oriented editor or character-oriented editor).

The SubFile Open, SubFile Close, and SubFile Monitor Procedures cooperate with each other to establish Port System connections with users for transput. Once a File is opened, all transput is handled by messages between the user and the SubFile Monitor. The formats used for these messages are very general and open-ended. The first word of the message contains a pointer to the message header, which may be in the message text or in a Page object passed along with the message. The message header has the transput opcode and a word for error return from the SubFile Monitor. The most common message format has two descriptors following the message header: one for the text being transput, and one for an optional key (e.g., line number or byte index into file). Each descriptor contains a pointer to the text (key), which may be either in the message or in the Page object. It also contains requested and actual lengths for the text (key). The semantics of a key are SubFile Type specific, while the syntax is fixed for all SubFile Types. The set of transput opcodes is also open-ended, with some SubFile Monitors implementing different subsets of the available opcodes.

The most common opcodes are FileRead, FileWrite, FileRewind, FileToEnd, FileReadGivingKey, FileSeek, and FileSeekRead. The last three deal with keys (e.g., FileReadGivingKey in a line oriented SubFile returns the next line with its line number).

It is important to note that in the above discussion no restrictions were placed on the representation of a SubFile object. Although the most obvious implementation represents the SubFile as a list of capabilities for Page objects, there is no requirement to do so. In fact, three SubFile Systems have been considered that use other representations. The first, which has already been implemented, is a line printer Spooler SubFile System. Another is a Terminal SubFile System which provides access to a user's terminal, allowing uniform access to files or terminals. The third example is an ARPA Network SubFile System, which makes file transfer between different Network Hosts very straightforward. The point all these examples is that the representational issues are left entirely to the various SubFile Systems, while the protection and synchronization issues are handled by the File System.

3.3. File System Implementation

The File System and two SubFile Systems (one line-oriented, the other a line printer spooler) have been operational since early fall of 1976. A third SubFile System (simple byte stream) has been implemented to explore the Confinement Problem. The design of these systems took place during May and June of 1976 and involved six people for a total of about fourteen man-days. The implementation of each of these systems took less that one man-month. Each was written in BLISS11, a PDP11 dialect of BLISS [Wulf 71], and has approximately three-thousand machine instructions. Coding, compiling, and most of the testing of the File System and various SubFile Systems were done independently. The only testing that required coordination between two implementors occurred when communication between the File System and a SubFile System was in question. This amounted to only a few man-days in each case.

In order to simplify maintenance, each SubFile System produces crash dumps whenever unexpected failures occur. This has made most of the errors very easy to detect and correct. A few obscure errors have required installing special versions of a SubFile System with a debugging system. Once an error has been corrected in a SubFile System Procedure, the Typecall mechanism has made it extremely easy to install new versions. The SubFile System maintainer merely has to replace capabilities for the changed Procedures in the SubFile Type object. All such maintenance activities are done without special privileges.

4. Evaluation

We have described a set of goals for an extensible file system for Hydra and details of a design and implementation of a file system to meet those goals. The design critically depends on the capability-based protection mechanisms of Hydra and makes heavy use of Hydra's Type, Subsystem, and Port mechanisms. This section provides a more explicit evaluation of our design with respect to the goals of [Cohen/Jefferson 75] and Section 2.

4.1. Implementation without Privileges

The most important goal of our work was the implementation of a practical File System without special privileges. The Hydra protection mechanisms allowed us to insure the integrity of the data structure and Type-specific operations of the File System. Furthermore, these same mechanisms encouraged a modular decomposition of the system; this decomposition, together with the use of the BLISS11 language greatly eased the design, implementation, and maintenance efforts.

While the Hydra file system design is functionally very nice, it is slower than it would be were it part of the Hydra Kernel. This is a natural result of using mechanisms as general and powerful as the Hydra Protection Mechanisms as extensively as we did. One dominant example of this is the implementation of the Procedure Call mechanism. During a series of traced calls to the File System, the average amount of computation within the File System was 49 msec and the average cost of Call overhead was 51 msec (i.e., about half the total time). Thus, while the domain crossings did make our protection and software engineering results possible, we find that they are very expensive on Hydra, since it implements the capability mechanisms via software.

4.2. The Modification and Confinement Problems

A second result was the implementation of a useful Subsystem able to solve the two chief protection problems discussed in [Cohen/Jefferson 75].

The Modification Problem was the simpler of the two. When a user passes a File parameter lacking $ModifyRts, Hydra prevents the File System from modifying the File or any object accessed through capabilities in the File. Whenever a File System operation has no intrinsic need to modify the File, it detects when the passed File parameter lacks $ModifyRts and functions correctly without modifying the File in any way. Thus in the case of the 'open for reading' operation, the implementing Procedure refrains from modifying the date of last access field in the File object when the capability lacks $ModifyRts. A more subtle constraint involves the Semaphore that controls exclusive access to the File object during Open and Close operations. An

analysis of the 'open for reading' Procedure reveals that the only indivisible operation on the File object necessary during the Open is the copying of a capability for the current SubFile object. Due to the capability mechanism of Hydra, this operation is indivisible, so no explicit locking is necessary and $ModifyRts is not needed. Were it necessary to determine the current SubFile by means of access to a multi-word data structure in the File object, on the other hand, some locking, and thus $ModifyRts, would be needed.

The second, more difficult, and more interesting of the two problems is that of Confinement. A confined Procedure Call is performed whenever the capability for the Procedure object of a Call (or Type Representative of a Typecall) lacks $UnconfineRts. In this case, Hydra removes $UnconfineRts and $ModifyRts from each capability inherited from the Procedure. This very simple mechanism insures that the only objects that may be modified in the Call are (1) the actual parameters passed explicitly by the caller and (2) objects local to the Procedure invocation. Furthermore, this mechanism is transitive, since only Procedure capabilities or Type Representatives passed as actual parameters can possibly have $UnconfineRts. Although it was not difficult to implement the File System under this constraint, it was difficult to construct confinable SubFile Systems. In the case of the Spooler SubFile System, there is the intrinsic need to modify a particular inherited object, the line printer device. In other cases, the need to modify is not intrinsic, but technological, and stems from the multiplexing of a single monitor Process to handle the transput for all its open SubFiles. Given this efficiency-oriented shared monitor concept, confinement is impossible.

One particular SubFile System was constructed, however, to explore the feasibility of a Confinable SubFile System. The Open Procedure of the Confinable SubFile System spins off a special monitor Process for each open SubFile. Due to the confinement constraint, the initial LNS of this monitor Process will be confined and may only modify the parameters of the Open Call. One detail of this should be pointed out, however, to be perfectly clear. As noted above, the propagation of confinement may be broken by the explicit passing of a Type Representative, having $UnconfineRts, by the caller. If the caller trusts the Subsystem identified by this Type Representative, then this is an appropriate way for him to limit the Confinement intended and is faithful to Lampson's first confinement criterion: "Transitivity: If a confined program calls another program which is not trusted, the called program must also be confined" [Lampson 73, p. 614, emphasis added]. In the case of the Confinable SubFile System, an explicit Type Representative for a Job object is needed to allow the Calls on the Process Scheduler (called the Policy Module in Hydra [Levin 75]) that are necessary to spin off the confined monitor; the Scheduler, for intrinsic reasons, cannot operate confined. Note, however, that this relaxation of confinement for the Scheduler is the only exception. It is done above-board, for the Type Representative is passed explicitly, and allows unconfined Calls only on the Scheduler (which would be within the privileged portion of a conventional system). Apart from the Scheduler, moreover, Hydra itself enforces the confinement of the File System, the SubFile System, and any other Subsystems they may invoke through inherited capabilities.

A point can now be made about the importance of the Modification Problem. The most obvious motivation, that given in [Cohen/Jefferson 75], is that a user should be

able to attempt a read-only access to a File without any risk of its corruption, as in the case when a user suspects a File System bug. This is not a forceful motivation, however, for such a situation would occur very rarely in such a critical Subsystem. A more convincing motivation stems from the desire to make a solution to the Confinement Problem a practical reality. A user can call a Subsystem confined, but that Call will fail unless the Subsystem can effectively get its work done without $ModifyRts and $UnconfineRts for its inherited capabilities. If, for example, a Subsystem needs to read a File (or look up a read-only item in a Directory) and if the File (or Directory) System did not solve the Modification Problem (it might fail by insisting on being able to update a date of last access field), then it would be impractical or impossible for the Subsystem to function confined. Thus, any Subsystem that intends to solve the Confinement Problem in a practical way must also solve the Modification Problem.

4.3. Separation of Representation and Protection

Another interesting result came from the separation of representation and protection issues. The distinction between the File System and SubFile Systems has made the File System highly extensible. This kind of extensibility is very important in experimental computing environments, like Hydra, where representational issues are open-ended.

In sum, an extensible file system has been constructed for Hydra. It is usable and was implemented at low cost without special privileges. Hydra's capability-based protection, Type, Subsystem, and Port mechanisms were all critical to the success of this project.

Acknowledgements. We wish to recognize the important contributions of David Lamb, Joseph Newcomer, Samuel Harbison, and Philip Karlton to the design of the File System. We also wish to thank our patient colleagues who commented on drafts of this paper, with special thanks to Roy Levin and Lee Schiller.

5. References

Cohen/Jefferson 75: E. Cohen and D. Jefferson, "Protection in the Hydra Operating System", Proc. 5th Symposium on Operating Systems Principles, Austin, November 1975.

Denning 76: D. Denning, "A Lattice Model of Secure Information Flow", Comm. ACM, (May 1976) 19 5.

Dijkstra 72: E. Dijkstra, "Notes on Structured Programming", in Dahl, Dijkstra, and Hoare, Structured Programming, 1972, Academic Press, New York.

England 74: D. M. England, "Capability Concept Mechanisms and Structure in System 250", Proc. IRIA Workshop on Protection in Operating Systems, Rocquencourt, August 1974.

Fabry 71: R. Fabry, List-Structured Addressing, Ph.D. Thesis, University of Chicago, March 1971.

Fabry 74: R. Fabry, "Capability-Based Addressing", Comm. ACM, (July 1974) 17 7.

Ferrie 76: J. Ferrie, C. Kaiser, D. Lauciaux, and B. Martin, "An Extensible Structure for Protected Systems' Design", Computer Journal, (November 1976) 19 4.

Habermann 76: A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems", Comm. ACM, (May 1976) 19 5.

Hoare 72: C.A.R. Hoare, "Notes on Data Structuring", in Dahl, Dijkstra, and Hoare, Structured Programming, 1972, Academic Press, New York.

Hoare 74: C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Comm. ACM, (October 1974) 17 10.

Jones/Liskov 76: A. Jones and B. Liskov, "A Language Extension for Controlling Access to Shared Data", IEEE Transactions on Software Engineering, (December 1976) SE-2 4.

Jones 77: A. Jones, R. Chansler, I. Durham, P. Feiler, and K. Schwans, "Software Management of CM*, a Distributed Multiprocessor", Proc. National Computer Conference, Dallas, June 1977.

Lampson 73: B. Lampson, "A Note of the Confinement Problem", Comm. ACM, (October 1973) 16 10.

Lampson/Sturgis 76: B. Lampson and H. Sturgis, "Reflections on an Operating System Design", Comm. ACM, (May 1976) 19 5.

References

Levin 75: R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/Mechanism Separation in Hydra", Proc. 5th Symposium on Operating Systems Principles, Austin, November 1975.

Lipner 75: S. Lipner, "A Comment on the Confinement Problem", Proc. 5th Symposium on Operating Systems Principles, Austin, November 1975.

Needham 72: R. Needham, "Protection Systems and Protection Implementation", Proc. Fall Joint Computer Conference, Anaheim, December 1972.

Robinson 75: L. Robinson, K. Levitt, P. Neumann, and A. Saxena, "On Attaining Reliable Software for a Secure Operating System", Proc. International Conference on Reliable Software, Los Angeles, April 1975.

Schroeder 77: M. Schroeder, D. Clark, and J. Saltzer, "The Multics Kernel Design Project", Proc. 6th Symposium on Operating Systems Principles, West Lafayette, November 1977.

Wulf 71: W. Wulf, "BLISS: A Language for Systems Programming", Comm. ACM, (December 1971) 14 12.

Wulf 74: W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System", Comm. ACM, (June 1974) 17 6.

Wulf 75: W. Wulf, R. Levin, and C. Pierson, "Overview of the Hydra Operating System Development", Proc. 5th Symposium on Operating Systems Principles, Austin, November 1975.