Proceedings of Sixth ACM Symposium on Operating Systems Principles (November 1977) 11-16.

The Cap Filing System

R.M.Needham and A.D.Birrell
Computer Laboratory, University of Cambridge

The filing system for the CAP is based on the idea of preservation of capabilities: if a program has been able to obtain some capability then it has an absolute right to preserve it for subsequent use. The pursuit of this principle, using capability-oriented mechanisms in preference to access control lists, has led to a filing system in which a preserved capability may be retrieved from different directories to achieve different access statuses, in which the significance of a text name depends on the directory to which it is presented, and in which filing system 'privilege' is expressed by possession of directory capabilities.

Preservation of capabilities

An executing program (protected procedure) in the CAP computer has access to a set of capabilities specified by its capability segments. These capabilities fall into various classes: store capabilities for segments of code or data, ENTER capabilities permitting transfer of control to other protected procedures, and software capabilities which confer such privileges as using a message channel, receiving peripheral interrupts, or stopping the system. These capabilities all represent to a program the right to have or to do something; the preservation of information from one run of a program to another (a universal operating system requirement) is thus seen by the CAP programmer as the preservation of a capability rather than of an object itself. It is the responsibility of the system to preserve an object *because* the user has preserved a capability for it. Ritchie and Thompson make a somewhat similar point in [1] in a non-capability context.

Capabilities are preserved in structures of two types, *directories* and *procedure control blocks* or PCBs. The primary purpose of a directory is to record a correspondence between a text names and capabilities; it will be explained later how directories perform other functions related to access control. The purpose of a PCB is to preserve the capabilities required for the construction of an instance of a protected procedure (see Needham and Walker, and also below). We do not discuss PCBs much in this paper; one remark which should be made is that since directories are not the only way in which capabilities are preserved, it follows that there is no requirement that a preserved capability have a text name.

The capabilities most commonly preserved are those for store segments. The CAP operating system implements a virtual memory for segments in user processes and in most system processes. Associated with each segment is a *system internal name* (SIN) and a disk address. In order to preserve a given store capability, it is necessary to determine the SIN of the specified segment and store it in a directory. A system data structure is accordingly maintained in which the SIN of any segment capability in current use may be looked up. In order to keep track of which segments have capabilities preserved in directories, a use count for segments is maintained in a system data structure called the SIN-directory. The count is incremented whan a capability for the segment is preserved, and is decremented when a directory entry is removed. The count is also incremented when a current capability is created for the segment, that is, when the segment becomes active in the virtual memory. When a user retrieves a preserved store capability from a directory, a system process (the *virtual store manager* or VSM) is told that the object corresponding to a particular SIN is wanted in the virtual memory, determines its disk address, and constructs a store capability appropriately. This capability is then passed to the user.

As mentioned, the use count in a SIN determines whether the system has a duty to maintain the corresponding segment. Since, as will be seem, directory structures can be cyclic, there is no guarantee that inaccessible disk space will be relinquished at the earliest possible moment. The disk is garbage-collected at each system restart; the filing system is not designed to run for ever.

Preservation of ENTER capabilities is achieved by storing the SIN of a segment containing a 'recipe' for construction of the protected procedure belonging to the ENTER capability. The segment containing the recipe is the procedure control block.

Preservation of software capabilities is much simpler. They do not refer to anything outside their own representations, so thay can simply be copied into the directory.

Capabilities for Directories

We have described how the primary aim of directories is achieved - to facilitate the preservation of capabilities in relation to particular text names. We now describe other aspect of the filing system, which are consequences of the policy of choosing the mechanisms of capabilities rather than of access lists.

The preservation of capabilities could be achieved by having a single data structure called 'the directory' in which any further structure was absent or expressed solely by conventions about the textual structure of names (cf. OS/360). The SIN-directory is a single structure, but it is quite unconcerned with text-names and thus with the operations usually associated with file directories. It is usually considered convenient to have more than one directory, and for the structure of directories to have some system significance. The original intention on the CAP was to construct a straight hierarchical structure of directories and subdirectories and to control access to files by means of access lists; there is nothing in the underlying capability implementation which forces the use of a filing system which follows the capability analogy in its structure. In the course of designing such a system we realised that to do so would be a substantial simplification. Accordingly we introduced the concept of a *directory capability*. Analogously to the way in which a store capability allows access to a set of words of store, a directory capability allows access to a set of preserved capabilities. Consider for example two users 'ADB' and RMN'; each is likely to be provided with a directory capability for his own *user file directory* (UFD). Each UFD would have a number of preserved capabilities. For example, ADB's UFD might have entries .STOP, a software capability for stopping the system, .TEXT, a store capability for the source of a compiler, and .BIN, a store capability for the executable code of the compiler. RMN's UFD might contain other preserved capabilities named .P, .Q.

The operation of retrieving a capability now has two arguments - it requires a text name and a directory capability. Any retrieval must provide these two things; there is no implied, default, or 'working' directory. By allowing the user to perform a wide range of operations on directory capabilities just as he can on store capabilities, many facilities become available at once. Suppose ADB had a large number of files, and wanted to be able to group them. He could obtain from the system a capability for a new empty directory just as he could obtain a capability for a new empty store segment. He could then preserve capabilities in it and later retrieve them. It is likely that he would wish to preserve a capability for the new directory in his existing UFD, but it is not essential that he do so. The significant thing about a directory is its *type*, not its presence in some larger structure. It is possible for a program to construct a complicated structure of directories which, like work segments, will go away when the program finishes. If ADB does preserve a capability for his new directory in his UFD, we might have a situation where the UFD contains the software capability .STOP, and a directory capability, say .A68C, which is the newly preserved directory. This in turn contains .TEXT and .BIN which are preserved store capabilities. Before considering further ramifications we outline the implementation of directory capabilities.

Internal Structure of Directories

The data content of a directory (text names, SINs and so on) is stored in a segment which naturally has its own SIN. Preservation of a directory capability is effected by storing its SIN, just as for store capabilities. The information in a directory, however, is privileged because of the presence of SINs. The SINs are just integers, and they refer to all the objects known to the filing system or virtual memory. It is therefore highly desirable that a user should not be able to assert falsely that a preserved segment capability is a directory capability, or to write arbitrarily into a directory segment.

The former safeguard is achieved by having the SIN-directory record type information about each retained segment; three basic types are recorded - store, directory, and PCB. The programs which actually construct capabilities for issue to a user make sure that the correct variety is made on each occasion. The latter safeguard is easily achieved by making use of the CAP's protected procedure mechanism. A directory capability in the hands of the user is an ENTER capability for a protected procedure (see Needham and Walker) which has the directory segment bound to its R-capability segment. All such directory capabilities share the same code, that of a procedure called the *directory manager* which provides the only interface between the user and the directory segment. It is the responsibility of the SIN-directory manager to construct the ENTER capability when a directory capability is retrieved. The user cannot read the directory segment so he cannot come to rely in any way on its format. The user does not thus see himself as presenting a directory capability and a text-name to a retrieval engine; he sees himself as presenting a text-name to a procedure. This contrasts with CAL-TSS, where two capabilities and a text-name were presented to a retriever, an approach which seems to leave more management reponsibility to the user, for example in remembering which access capability goes with which text-name, unless the generality be limited by convention.

Sharing capabilities

It is customary for filing systems to provide facilities for users to allow access to some of their files to be available to other users. This is commonly achieved by an insertion in a directory of the general form 'allow access to .BIN if user = RMN', 'allow access to .BIN if program = LOADER',or sometimes 'allow access to .BIN if the requestor produces such-and-such a magic number'. We have chosen to provide a capability-oriented system which does not have such access lists at all. The right of access to an object is signified by possession of a capability for it. A program can pass this right to another by passing the capability (by mechanisms which do not concern us in detail here). For example, suppose that ADB wished to give RMN some access to the executable code of his compiler, the operations would proceed as follows:

1. ADB retrieves a capability for .BIN from his subdirectory

2. ABD passes the capability to RMN

3. RMN can now preserve this capability in his UFD.

After this sequence there would be two preserved capabilities for the compiler, .BIN in ADB's sub-directory, and .COMPBIN, say, in RMN's UFD. The use-count in the SIN-directory will have been incremented in the course of the new preservation, and even if ADB were to remove .BIN from his subdirectory RMN's entry would still be valid.

Since directory capabilities are handled uniformly, a similar sequence of events could have been used to allow RMN to preserve a capability for the sub-directory, thus allowing RMN some access to each of the capabilities preserved therein:

1. ADB retrieves .A68C from his UFD
2. ADB passes the capability to RMN
3. RMN preserves this capability in his UFD as .COMP.

In this case RMN's UFD would now contain an entry, .COMP, which is a preserved directory capability for the same directory as that retained as .A68C in ADB's UFD. Sub-directory capabilities may be used in several ways. RMN could retrieve his capability for .COMP and then present to it simple names such as .BIN, or he could present to his UFD a two-part name .COMP.BIN. The dots in a file name separate it out into a series of components each of which except the last must name a directory capability in the directory to which it is presented. The directory capabilities are retrieved and the residue of the names presented. If the last component of the name selects a directory capability then the directory is retrieved for the caller. The implications of this naming structure for access status are considered later.

Commentary

The system as described so far has features in common with the filing systems both of UNIX (Ritchie and Thompson) and CAL-TSS (Lampson and Sturgis). It shares with UNIX the lack of interest within the directory structure in physical information about files, and the possibility of having multiple names for the same thing. The CAP directories, however, are not pure name-manipulators as are the UNIX directories. This is a non-trivial difference, because the use of directories to contain access information means that multiple entries for the same file need not have the same access status. We take the view that the only things about the file which must really be unique are its position and size (discovered via the SIN-directory) and its type (kept in the SIN-directory). UNIX has a single access status for a file, held in a manner which would be analogous to our keeping the access status in the SIN-directory. Also UNIX insists that directories form a strict hierarchy, in order that a reference-count system can work without losing objects. This point is perhaps better discussed in relation to CAL-TSS. That system is the closest to ours in structure, since it has a general naming network as a consequence of being able arbitrarily to retain directory capabilities. CAL however took precautions to prevent the formation of 'lost' objects or substructures which we have not considered necessary. We have been prepared to take the view that loss of substructures is unusual and that the space can and should be recovered on system start. This is largely a matter of taste, and the distinguished entry method used in CAL could be implemented in CAP without significant system change. We would probably refuse to allow the distinguished entry for an object to be deleted while others existed, rather than allowing it to vanish and *ipso facto* invalidating all others. (The implementation would be easy because the first occasion on which a capability for an object is preserved is apparent from the SIN-directory, whose manager already has code specific to this case. The appropriate directory entry would then be made the distinguished one). There is a substantial difference in the use made of the directory structure in the two systems. The concept of a link is entirely absent in the CAP, which provides for further stages of name lookup rather than for lookup of a different name. The CAP directory manager does not need to know whether a particular directory entry is for a file or for another directory; if the series of directories runs out before the series of name-parts then the user is in error and if the name runs out before a file is found then a directory is retrieved.

In practice little use is made of retrieval using extremely long names directly. Programs retrieve the directories they will require and use them with, usually, one-part names. The visible effects of this are rather similar to those of systems which have a system notion of 'current directory' or which have methods of 'presuming' or 'defaulting' the first so many components of a file name. In our case it is entirely up to the programmer to make what arrangements he sees fit, perhaps retrieving and holding on to several directories and always knowing by context which of them to use. Another characteristic of the CAP system is the access control techniques used, and to these we now turn.

Access Controls

In general, capabilities do not give unrestricted access to the object to which they refer. A filing system must be able to retrieve capabilities with the status intended; it us usual to return an error if that status is null since the appropriate capabilities would be of little use. We consider first store capabilities and then directory capabilities.

For most store capabilities the rights are a combination of one or more of 'read', 'write', and 'execute'. If a program has access to a capability, it may pass to another program some more restricted access. An assembler, for example, might very well have RWE access for the segment into which it places a core-image, but only give RE to the user when assembly is complete. The author of a character file would have RW access to it, but would pass only R accss to his readers. This refinement of access is available in the CAP as a machine instruction implemented by microprogram. The directory manager when preserving a capability stores with the name and SIN the access status of the capability handed in for preservation, and will not retrieve the capability with any higher access. For example, if ADB manufactured executable code of his compiler using RWE access, and so preserved the capability, he could refine it to RE before passing it to RMN who could then preserve it with at most that access.

Analogous considerations apply to directories, for example ADB's new subdirectory. If he passes a capability for it to RMN he may wish to restrict the operations that RMN may perform using the directory. For example, he may wish to prevent RMN from creating new entries, or deleting certain entries, or overwriting others. The way of achieving this in most filing systems has been to store in or with the directory a list of users or of categories of users or of access keys, together with the rights that being the user or possessing the key conferred. In the CAP system we have avoided such lists while producing a system of similar practical function by including access control information in the directory capability. Each directory capability specifies some access status for the directory; this access status consists of five bits called C,V,X,Y and Z. The 'C' bit allows the creation of entries in the directory and the other four bits indicate the access rights obtainable for each preserved capability, as follows.

Stored in the directory with each preserved capability is an *access matrix*; this has one row for each of the bits V,X,Y,Z. Each row contains bits indicating access rights to the entry. (See the examples below.) If the entry is for a preserved store capability, the bits in each row of the entry are a selection from:

D - indicates permission to *delete* the entry

U - indicates permission to *update* the entry (i.e., make it contain a different preserved capability)

A - indicates permission to *alter* the access matrix for the entry

R,W,E - allow the retrieved form of the capability to include bits R,W,E respectively

If a program has some access status for a directory (as specified by a directory-capability), then it can use that capability to obtain some access to an entry; that access is restricted to the inclusive 'OR' of each row of the entry's access matrix for which the corresponding bit (V,X,Y,Z) appears in the program's access status for the directory. For example, if the entry .BIN in ADB's new sub-directory has an access matrix:

	D	U	A	R	W	E
V:	1	.	.	.	.	.
X:	.	1	.	.	.	.
Y:	.	.	.	.	1	.
Z:	.	.	.	1	.	1

and if ADB has a capability giving him status CXYZ for the sub-directory then ADB could create new entries, and would have access URWE to .BIN; he could thus retrieve a capability for the binary with up to RWE access, or could make the entry .BIN refer to some different store segment.

Given a directory capability a user may produce a capability for the same directory but with reduced access by means of the REFINE instruction, which works upon enter capabilities just as it does on segment capabilities. ADB, with his CXYZ capability for the sub-directory could use this facility to obtain a YZ capability for it; he could then pass this to RMN. RMN could preserve this capability (if he wishes), but could in no way obtain a status greater than YZ for the sub-directory. Such a status prevents RMN from creating entries in the directory, or from altering the entry .BIN, but he can obtain up to RWE access to the segment named by .BIN. Similarly, someone with Z-access to the sub-directory could obtain at most RE access to .BIN.

If an entry is for a preserved directory capability, exactly the same rules apply. For example, when ADB created the sub-directory (by entering a privileged system procedure, publicly available), he would obtain a capability giving status CVXYZ for the sub-directory. Suppose he preserved this in his UFD as .A68C, with access matrix:

	D	U	A	C	V	X	Y	Z
V:	.	.	1	.	.	.	.	.
X:	.	.	.	1	.	1	.	.
Y:	.	.	.	.	.	.	1	.
Z:	.	.	.	.	.	.	.	1

Then, since ADB has status CVXYZ for his UFD, he has access ACXYZ to the entry .A68C allowing him to retrieve a capability with up to CXYZ status for the sub-directory. The 'A' bit has been used here for two purposes. Firstly, ADB has prevented himself accidentally deleting the entry .A68C by not including the 'D' or 'U' bits in the access matrix -- he could subsequently delete or update the entry by *first* altering the access matrix to include D or U, *then* deleting or updating the entry. Secondly, although by altering the access matrix ADB could give himself V access to the sub-directory, he has arranged that this access is not at present available. This is an example of the extremely useful facility that although ADB has privileges with regard to the sub-directory, he can temporarily prevent himself from exerting those privileges.

A text name presented to a directory capability is a sequence of components, each of which except the last specifies a directory in which the next component is to be looked up. For example, ADB could enter his UFD to retrieve .A68C.BIN; this means looking up .A68C in the UFD, to find a directory from which .BIN is retrieved. If the set of preserved capabilities accessible from a directory by means of such multiple look-ups is considered as a directed graph, then a text name specifies a path through this graph. At each step in the path, the access status is multiplied by an access matrix as described above, ending with some access to the desired entry in the desired directory. If at some stage on this path the access status is found to be zero, the request is, of course, rejected by the directory manager.

The above is a fairly complete description of the mechanisms and facilities made available by the directory manager. Certain conventions are adopted as managerial policy in using the CAP filing system, but as with most conventions, they are not uniformly applied. Capabilities for the UFD's are preserved in one directory, the *Master File Directory* (MFD). The MFD is available (with status 'Y') to the program when logging in; when this program is satisfied of a user's identity, it retrieves his UFD from the MFD with as much status as it can, and passes it to the user's command program. The access matrix in the MFD for most users' UFD's is:

	D	U	A	C	V	X	Y	Z
V:	.	.	1	.	.	.	.	.
X:	.	.	.	.	.	.	.	.
Y:	.	.	.	1	1	1	1	1
Z:	.	.	.	.	.	.	.	1

and most UFD's have a preserved directory-capability called .* giving Z-access to the MFD. Thus, in general, a user can obtain Z-access to other users' UFD's. For example, RMN could retrieve .*.ADB.A68C.BIN to obtain RE access to the executable code of the compiler. We have already seen that RMN has another, more privileged path which he could use to obtain RWE access to the same store segment. Similarly, while ADB could use the path .*.ADB.A68C to obtain Z-access to the sub-directory, he could use .A68C to obtain CXYZ, or could alter the access matrix to allow himself up to CVXYZ. The MFD thus permits users to name most files by a means which gives them minimum access. Some files he may not be able to get to because the lookup stops if it encounters a directory to which no access is available; also there are files with no names at all, as mentioned earlier.

Summary

The CAP filing system has been designed for a specific capability-based machine, but although the implementation was made much easier by the peculiarities of the CAP protection system it would be practicable on any capability-based architecture of sufficient generality. The basic step was to equate the concept of 'file' with that of a preserved capability; thereafter we attempted to continue the capability analogy in the structure of the directory system. We have produced a filing system of considerable generality and flexibility; indeed the facilities are so general that they are restricted by convention to a manageable subset.

By handling directory capabilities in the same way as store capabilities we allow not only hierarchical directories but also shared directories - the structure can form an arbitrary directed graph, which may even be cyclic. The access controls permit giving to others capabilities with restricted or privileged access to segments or to directories, and allow privileged users to avoid exerting their privilege.

By allowing dynamic creation of directories, we allow the manufacture of directories or of complex structures of directories separate from the main filing system, accessible only to certain programs (this is used for the password file, for example).

A feature of the system which is sufficiently unusual that it may confuse the user with experience elsewhere is that the access to an object my differ depending on the naming path used. In particular to present name *n* to directory *d* is not necessarily the same as presenting *d.n* to some 'root' directory - the object named is the same, but the access may be different.

The filing system has been satisfactory in use. Since all aspects of its security are enforced by a single system protected procedure, protected by an ENTER capability from outside interference, we hace a high degree of confidence in it.

Acknowledgements

The CAP Project is supported by the Science Research Council.

References

1. Ritchie, D.M. and Thompson, K. , The UNIX Time-Sharing System, Comm.ACM, July 1974

2. Lampson.B.W. and Sturgis, H.E. , Reflections on an Operating System Design, Comm.ACM, May 1976

3. Needham, R.M. and Walker, R.D.H. , The CAP Computer and its Protection System, SOSP6, 1977.