Proceedings of Sixth ACM Symposium on Operating Systems Principles (November 1977) 17-22.

The CAP project - an interim evaluation

R.M.Needham
Computer Laboratory, University of Cambridge

The CAP project has included the design and construction of a computer with an unusual and very detailed structure of memory protection, and subsequently the development of an operating system which fully exploits the protection facilities. The present paper passes the work in review and draws conclusions about good and bad aspects of the system. The basic architecture of the CAP machine is described in [1] and a largely prospective description of the protection system is given in [2].

The project was started as an experiment in hardware memory protection. A computer was to be designed in which operating system development was easy, in which ruggedness was produced by a much more fine-grained network of firewalls than was (or is) usual, and in which the full range of protection facilities was available to the writers of subsystems. Simplicity of mechanism was a very important goal, although some emphasis was placed on flexibility of protection policy.

The intention may be summed up as: to provide a system which, without unreasonable overheads, applied in the most rigorous manner attainable the principle of minimum privilege as it relates to access to memory. This intention has been to a very large extent achieved. In the computer and operating system as constructed, no compromise has been necessary on fine-grainedness of the protection, and all of the facilities are available to the ordinary user. With fine-grained protection goes a need for very frequent changes of protection environment. It had originally been hoped that such changes of protection environment would be sufficiently cheap that, for example, were it desired one could have a change of protection environment for every character read from an input file. Whether or not it was sensible to wish to do that, in the result it has proved unreasonably costly to do so, and protection environment changes only occur once per line.

Outline of Protection Support

The basic unit of protection in the CAP is the segment which is a contiguous set of words of memory from 0 to 65535 in length in incremenets of one word. The next higher unit of protection is the protected procedure. Each protected procedure has its own fully encapsulated address space, that is, the memory location referred by an address depends upon the protected procedure in which that address is used. The interpretation of capabilities for segments is done by hard logic in the capability unit, and the interpretation of Enter capabilities for protected procedures by microprogram. The mciroprogram is also responsible for loading the associatively selected capability register in the capability unit.

The division of responsibility between capability unit and microprogram has proved satisfactory. Support in hard logic for segment capabilities is essential if performance is to be maintained, whereas it would be too complicated for protected procedure call and return. The microprogram support for these latter functions, and for associated operations to do with moving capabilities, gives the speed and ruggedness which are necessary if frequent changes of protection environment are to be tolerable. There is no kernel in the software sense, although the parts of the microprogram which are concerned with capability manipulation (as against input/output or just doing instructions) may perhaps be regarded as corresponding to a kernel. They amount to some 1500 micro-orders out of a total of some 3200. It is characteristic of the capability-handling microprogram that it regards all in-core data structures with complete suspicion and does not rely for its consistent and specified performance on the integrity of any data supplied by the operating system. Since all the system data structures are constructed out of regular segments, a substantial amount of the checking is performed by the capability unit, but some has to be done by sequential microprogram. It would have been possible to make some of the capability handling operations faster by omitting checks, which would amount to embodying in the microprogrma the assumption that operating system conventions had been adhered to, and accepting the risk that if they had not then the microprogram could behave in an unspecified way. The decision not to make assumptions about system data structures was deliberately taken, and is regarded as justified on grounds of good design. It has sometimes led to apparent inefficiencies, as the following example shows.

It is described in a companion paper [1] how, when the microprogram detects something which should lead to a trap in a running process, entry is forced to the coordinator. The coordinator then causes entry to a particular protected procedure called FAULTPROC in the offending process. This somewhat roundabout operation is a consequence of the principle of suspicion between the microprogram and in-core data structures. For the microprogram to force entry to FAULTPROC directly there would have to be available, in a known place, an enter capability for it. Furthermore, since the traps handled by FAULTPROC include that one which is interpreted by the operating system as indicating that an attempt has been made to load a capability for a segment which is not in memory, it would have to be known that the segments of FAULTPROC itself were all in memory. The presence of the enter capability for FAULTPROC in the right place, and the presence of its segments in memory, are matters of operating system conventions rather than of architectural design, and the microprogram would have to behave in a reasonable manner if the convention were not followed. The error handling required for this purpose in the microprogram would be impracticably complicated and quite different from the regular error handling, particularly bearing in mind that some traps may validly occur in FAULTPROC itself. The only way in which this complexity would be avoided would be to verify on each entry to a process from the coordinator that the requisite apparatus was present and in good order. To do this would make the process entry sequence intolerably long. Some verification of the same general class is done in the present system -- for example, it is verified that the register dump area is not merely readable so that register values can be restored for the process being entered but writable so that they may later be preserved -- and any substantial addition would be too much. Accordingly, the microprogram causes entry to the coordinator whenever a trap occurs because this is an action which can be guaranteed to be possible without giving rise to further traps. It is still a major system error if the enter capability for FAULTPROC is misplaced or if its segments are not in memory, but the effects of such an error are clearly ascribable to the operating system software rather than to the microprogram, and changes in operating system convention do not lead to a requirement for consequential changes to the microprogram.

An Extended Kind of Capability

Capabilities for segments are presented to the addressing hardware, and ENTER capabilities to the microprogram. The idea has been extended to include capabilities which cannot validly be presented to either, but which may be presented to protected procedures as a sign of authority to request some action. These capabilities are known as *software capabilities*, they are kept in capability segments just like segment capabilities and ENTER capabilities, and may be retrieved from the filing system in just the same way. They furnish a powerful way of unifying the treatment of various kinds of privilege or permission with the general capability structure in various circumstances where direct use of the general mechanism would appear wasteful. Two classes of use of software capabilities may be distinguished.

a) Software capabilities are sometimes used to specify the objects on which a protected procedure is to act. For example, a procedure called SETUP creates message channels between processes. A software capability contains the identifier of a particular channel and is a sign of authority to create, as the case may be, the sending or receiving end of it. Notice that the software capability here is *not* a form of representation of an abstract object, but a permission to make one.

b) Software capabilities are sometimes used to authorize use of specific options in the use of a single protected procedure. It sometimes happens that a number of actions require almost the same program, but it is nevertheless desirable to separate the privileges to call for them. The operations "create capability" and "update capability" form an example.

In both of the above instances the use of software capabilities could have been avoided by having more protected procedures. However, protected procedures require capability segments, and the use of software capabilities saves several tens of them per process. The existence of large numbers of capability segments leads to administrative overheads which are well worth avoiding if it is possible to do so without loss of protection. The importance of this point was not sufficiently appreciated in the early design stages.

Effectiveness of Protected Procedures and Objects

Consider a simple protected procedure called SETUP. This exists to set up message communication routes between processes. Its list of capabilities is as follows:

3 code segments, of which two are parts of the ALGOL68C library shared with all other ALGOL68C procedures, and one is the code of SETUP itself (under 500 compiled instructions)

1 workspace segment -- the stack

1 segment shared among all instances of SETUP in different processes; it contains a global table of existing message channels

2 permission capabilities, of which one is to create entries in the process resource list, and the other is to create new message channels

SETUP may be called by anyone. It accepts as arguments software capabilities relating to particular logical

channels, and its task is to bring the current process into connection with a logical channel, if necessary creating the channel itself. It verifies that the capability presented to it is of the correct type, consults its global table to see whether the channel exists, and uses its permissions to call ECPROC, a more privileged procedure, to do what setting up is needed. No enter capability for ECPROC is required in the above list, since such a capability is globally available, all actions of ECPROC being protected by a requirement for software capabilities.

From this description of the action of SETUP, we may draw a number of observations about the mechanism.

1. *Small domains*

SETUP is a very short program, and the control paths through it are also short, but it is practical and sensible to treat it as a separate protected procedure given the microprogram support for moving between domains of protection. Similar remarks apply to ECPROC.

2. *Minimum privilege*

SETUP can only do the job intended. It has no other privilege and no bug in it could cause it, for example, to alter an existing capability.

3. *Explicitness of privilege*

It is immediately clear from inspection of the capabilities available to SETUP just what it may and may not do. It is a great deal quicker, in cases of doubt, to look at the capability list than at the code itself. It was noticed that for debugging purposes, a software capability had been made available to SETUP which gave the drastic permission to cause the whole system to stop, and that this permission had not been removed. Once noticed, it was removed without difficulty. It may be remarked, that although tidiness required the removal of the code which exercised the departed privilege, it was not necessary to remove the code at once to get rid of the protection error, since the code would have failed in execution.

4. *Ease of testing*

SETUP provides a small number of simple services. It is thus easy to test because one can proceed exhaustively through the significantly different types of call. This is a further consequence of the practicability of small domains for which the hardware encapsulation is known to be complete.

Protected Objects

The implementation of protected objects in CAP is by means of protected procedures. A typical example of a protected object is a file directory; the ony way in which anyone may use it (other than the file system restart procedure) is by means of an instance of a protected procedure known as a *directory manager* which has the directory bound into it. A user program may have a number of directories available to it; they appear as distinct ENTER capabilities for distinct instances of the directory manager. The instances differ in the content of their R-capability segments. See [1].

A theoretical deficiency of this protected object mechanism is that there is no general way to disassemble a protected object. For example, there is no method by which a procedure can take an instance of the directory manager and extract the whole or a part of the directory itself, or by which it can cause the directory manager to deliver the directory segment as a result. It would be awkward for this reason to provide operations which intrinsically require the use of two or more directory segments and which need, for protection reasons, to be performed by the directory manager or a similar protected procedure. This theoretical deficiency has not caused any difficulty in the construction of our operating system. It is believed that the reason is the fine grain of protection, which has the consequence that protected objects are all very simple. The procedures which perform the permitted operations upon the objects perform all of them rather than some of them and the simplicity of the object avoids any need for partial disassembly as provided in the Hydra system, where the protected objects can be altogether more complex and high-level structures.

Other Aspects of Protected Procedures

A recent addition to ENTER capabilities is the use of access bits in them. The field which contains access bits in a segment capability may contain in an ENTER capability a bit-pattern which is notified to the called procedure in a register. The bit-pattern may be amended by the REFINE instruction in just the same way as the access bits for a segment capability. This mechanism enables the services which a procedure will give to vary between different copies of its ENTER capability, accepting or rejecting requests on the basis of the access bits used.

The operating system designer has a number of possibilities open to him if he wishes to provide several services using similar programs which should not all have to be available at the same time. Firstly, he may set up quite separate protected procedures, and arrange that their ENTER capabilities are distributed appropriately. Secondly, he may provide a single ENTER capability and protect the individual functions by software capabilities. Thirdly, he may provide a single ENTER capability and protect the individual functions by its access bits. The designer chooses the mechanism to use on the basis of economy of apparatus. Software capabilities are appropriate where the number of options is large or where the capabiltiies themselves contain some data - as in the case of SETUP's argument capability. Access bits are appropriate

where the options are few, and distinct enter capabilities where the sections of program or data unique to particular options are substantial.

An aspect of the protected procedure system which we now consider to be unfortunate is the stack implementation underlying the call of protected procedures. The stack implementation has latent problems of overflow which are tiresome to deal with properly, and there is no good reason for it. A non-stack implementation would probably be better since there is no occasion for recursive invocation of the *same* protection domain, as distinct from a different instance of a domain -- e.g. the directory manager for another directory. In the design phase we thought that instances would arise where protected procedure A, for example, called B which called A, but the system has just not worked out that way, so we now consider the flexibility unjustified in relation to the complication caused.

Programming Considerations

With the exception of a few hundred words of assembly code, the entire system has been written in ALGOL68C. The special operations connected with the protection system have been provided as library routines, as have some operating system primitives such as "send message". It had originally been intended to extend the language so that such instructions as ENTER, MOVE CAPABILITY, REFINE CAPABILITY, RETURN, etc., would be compiled in-line. This was not done for lack of effort, and the approach adopted is now considered to be advantageous, since it enables us to make use of the extensive machine-independent parts of the compiler, which lessens our software maintenance commitment.

Every protected procedure in the operating system is an ALGOL68 *complete program*, not an ALGOL68C procedure. This is important, because calling conventions between protected procedures can be decided on system grounds, not language grounds. It also helps to avoid the system becoming a single-language one; BCPL is now fully available, though the original intention to write parts of the operating system in it has not been pursued. BCPL has been used quite extensively for user programs.

Each protected procedure has its own workspace (stack, and, where necessary, heap). This local storage persists from one call of the protected procedure to the next, unless the procedure explicitly creates and destroys it. This was taken as the default case because almost all operating system procedures have storage which should persist in the way indicated -- indeed, if they did not, there is no obvious reason why they should need to be protected at all. It has been found convenient to take advantage of the persistence of local storage, and of the fact that each protected procedur is a complete program, by implementing a kind of coroutine mechanism. In the ALGOL68C library there is a subroutine called *return* which includes a RETURN instruction. The library subroutine is so arranged that when the protected procedure is next ENTERed, control resumes at the instruction after the RETURN instruction. This leads to exit from the *return* subroutine, and resumption of the main program at the point immediately following the call. The effect is that when a protected procedure is first called it is started at the beginning, but subsequent calls pick up where the last one left off. As a result, a typical protected procedure has the structure:

```
BEGIN
   initialization code
END
DO # to infinity
   CASE first argument IN
      services offered
   ESAC
   return (result)
OD;
```

The initialization code is only executed once. If constraints are to be imposed on the succession of calls to the procedure they may be achieved by putting calls of *return* elsewhere. A very similar structure is found in the main programs for services provided as separate processes activated by message:

```
BEGIN
   initialization code
END;
DO # to infinity #
   WHILE messages(input)=0 DO waitevent OD;
   receive message with reply (a,b,c,d);
   CASE a IN
      services offered
   ESAC;
   return reply (p,q,r,s);
OD;
```

The similarity of structure makes it relatively easy to give effect to policy changes as to whether particular services should be provided as protected procedures or in separate processes.

We have been extremely satisfied with ALGOL68C. The combination of its powerful compile-time checks with the run-time protection of the capability system has had the result that a great many system procedures have worked correctly once they would do anything at all.

Virtual Memory System

1. *Swapping Aspects*

The CAP was not designed as a paged machine. This

decision was consistent with the emphasis on protection of small units of information, which, as has been said, may be only a few words long. The consequence, however, has been that there have been severe problems of real store management. The store allocator has to handle requests for sequential memory ranging from half-a-dozen words to 32K, and the number of very small segments is large. Capability segments of from six to twelve words are common and a user process may easily have forty of them. There are two undesirable consequences: firstly, there is fragmentation on the disc, and secondly, the time taken to clear a store region by swapping out is increased by the multiplicity of separate transfers and latencies. Work is in hand to mitigate these problems by amalgamating all capability segments of a process into one or two reasonably sized units for swapping purposes. The architecture is helpful here, since a number of distinct entries in the Process Resource List may all be defined in terms of the same senior capability which, as explained elsewhere, is the unit of swapping.

Although careful programming is improving store management, it must be admitted that the organizational problems of a multiplicity of small segments are a drawback of the general approach.

2. *Structural Aspects*

The CAP virtual memory [1,3] is composed of segments of three types: regular segments, directory segments, and procedure control blocks. It is structured by means of system internal names which function as pointers. A user program may retrieve capabilities for virtual memory objects explicitly by means of a directory manager or implicitly, by retrieving a procedure control block (PCB) which is turned into an instance of protected procedure by the linker.

The procedure control block is the most characteristic part of the system. It contains the specification of a protected procedure in which existing virtual memory objects are referred to either by system internal name, or by file title. The existence of a reference to an object by system internal name in a PCB implies that the object cannot be destroyed, just as does a reference to such an object in a file directory. The possibility of retention by internal name in a PCB requires that the filing system restart procedure look at PCBs as well as directories when performing its consistency check. It is slightly unfortunate that this was not realized sooner, since the internal formats of directories and PCBs could have been made much more similar, with consequent simplification.

It would not be difficult in principle to include in the structure other types of segments which contain system internal names; however, the complexities which would be produced in the restart procedure have deterred us from doing so, and new data structures are instead built out of the existing types of object. Ideally, there would be a single format for segments in which system internal names were found, and such segments would contain nothing else. Directories and PCBs would each consist of pairs of segments one of which contained nothing but system internal names, the other containing pointers to specific system internal names together with, in the case of directories textnames and access information, and in the case of PCB's the data specifying the size and other contents of the PCB segment. The functional analogy between system internal name segments and capability segments would be closer. We were deterred from this approach by the consequent intensification of the difficulties mentioned earlier in the section on swapping, since the number of small segments would be much increased.

The user has to be aware of the existence of system internal names for two reasons. Firstly, when he is, by means of the appropriate protected procedure, making a new PCB, he needs to say whether he would like a particular constituent object to be linked by system internal name or by file title. Secondly, if he has two preserved capabilities there is no general way for him to discover whether or not they are capabilities for the same object other than to obtain and to compare their system internal names. This is not entirely satisfactory; there has been debate as to whether the present system whereby the internal name is accessible for this purpose should be retained or whether to change to a less efficient but purer system whereby a user has to present a pair of capabilities to a protected procedure which will discover and report to him whether they are identical or not. There is no loss of protection in allowing the user to see system internal names, but their values should be of no interest to him.

Filing System

Some structural aspects of the filing system have been covered by implication in the foregoing, and a general description is given in [3]. The original intention had been to provide a more conventional filing system in which a directory served to regulate, by access lists or otherwise, the issue of capabilities for filed objects. The only objects which could be filed would be new ones into which some information had been recorded. Such filing systems have mechanisms for renaming and some kinds of sharing by means of links or other devices. To have constructed such a filing system would have been in no way incompatible with the objectives of the project as given at the outset of this paper. We chose instead to implement the system described, as being interestingly different and generally compatible with capability ideas in the extreme sense; any capability which comes into a user's possession may be preserved permanently if desired. A system internal name in a directory or PCB may be thought of simply, and accurately, as the preserved form of a capability; its integrity is protected by the procedures which

encapsulate the two types of segment.

Several consequences follow for the user as unusual and interesting features.

1) Segments can exist whose capabilities are not preserved in any directory but only in a PCB. The access controls on such segments are uncomfortable for system programmers, which is probably a good thing. There is *no* way of accessing them except via the appropriate protected procedure, and nobody who can circumvent the restriction. No accident such as the leakage of a password can enable a user to have direct access.

2) A user can bind an existing version of a segment to his program with the assurance that it will continue to be available. This may be regarded as beneficial, since new versions of compilers, translators, libraries, and other utilities are commonly unreliable. This facility is the obverse of the lack of means for revocation of access.

3) It is possible to create directories, or structures of directories, which are not themselves retained anywhere, and which will disappear when they become inaccessible via any current capability. The existence of an object of type directory does not imply any connection with a master or root directory.

Conclusions and Further Work

The principal dangers foreseen at the outset of the CAP project were that an attempt to incorporate the desired degree of protection into a practical system might fail under a weight of complexity and mechanism, and that the basic protection design might prove inadequate for the requirements of a real, rather than a toy, operating system. It is now clear that these dangers have been avoided. Much remains to be found out, under various heads.

1) *Relationship of ordinary users to the system*

It has been demonstrated that the system does not impede ordinary users when developing ordinary programs. It remains to be demonstrated whether or not the protection features are of value to users developing elaborate program subsystems.

2) *Quantitative evaluations*

It is only when a substantial amount of regular computation, as against system development computation, is done that it becomes meaningful to attempt empirical studies of the costs and effectiveness of the protection features.

3) *Restructuring*

A claimed benefit from the explicitness and precision of the protection in CAP is that system restructuring is easier than it would otherwise be. Experience to date suggests that this is so, but valuable insight will be gained from such substantial rearrangements are are inevitably suggested in hindsight.

4) *Clean-up problems*

The deliberate lack of distinction between system procedures and ordinary procedures accentuates the need for thorough understanding of the techniques related to premature termination of computations, failure to complete sequences of operations, etc. It is no longer appropriate, for example, to hold off the effect of a console 'quit' signal until the affected process is no longer running in the operating system, since the moment of leaving the system is not defined.

Acknowledgements

The CAP project is supported by the Science Research Council. The operating system and associated programs are the outcome of design and implementation work by numerous people, notably A.D. Birrell, J.S. Fenton, D.W. Payne, C.J. Slinn, and R.J.B. Taylor, with many contributions from later research students and from others interested in building systems.

References

[1] "The Cambridge CAP computer and its protection system," R.M. Needham and R.D.H. Walker (*SOSP6*, 1977).

[2] "Protection Systems and Protection Implementations", R.M. Needham (*FJCC*, 1972).

[3] "The CAP filing system," R.M. Needham and A.D. Birrell (*SOSP6*, 1977).