Proceedings of Sixth ACM Symposium on Operating Systems Principles (November 1977) 1-10.

# The Cambridge CAP Computer and its protection system

R.M.Needham and R.D.H.Walker
Computer Laboratory, University of Cambridge

This paper gives an outline of the architecture of the CAP computer as it concerns capability-based protection and then gives an account of how protected procedures are used in the construction of an operating system.

## Outline of architecture

The architecture of the CAP, implemented partly by hard logic and partly by microprogram, is designed to support a very fine-grained system of memory protection. The intention is that each module of program which is executed on the CAP shall have access to exactly and only that data which are required for correct functioning of the program. Access to a particular area of memory should never imply access to any other. This requirement has led to the design of a non-hierarchic protection architecture, in which emphasis has been placed on the representation of very detailed protection environments, and on the possibility of rapid switching from one environment to another. The CAP also supports a hierarchical structure of processes, of such a nature that the position of a process in the hierarchy determines the resources available to it.

### Capabilities and segments

In this paper the term *segment* will be used to refer to a segment of memory consisting of a contiguous set of memory locations defined by a base and a limit. Access to the contents of a segment may only be obtained by the use of a *capability* for that segment. A capability may be evaluated to obtain the base and limit of the segment to which it refers together with an *access status*. The access status in a capability consists of five bits of which three refer to data access, namely 'read data', 'write data', and 'execute' and two refer to capability access, namely 'read capability' and 'write capability'. A *data-type* capability has one or more of the data access bits, and a *capability-type* capability has one or both of the capability access bits. No capability may have both data and capability access bits, though it is possible for both types of capability to exist for the same physical segment.

Capabilities are stored in segments with capability-type access and are evaluated by the microprogram as and when required. At any moment in the life of a process, the resources accessible to it are defined by the capabilities which it is able to use. We shall now describe the way in which addresses as issued by a program select capabilities, the way in which accessible sets of capabilities change as a process moves from one domain of protection to another, and then the structure, strongly related to the hierarchic structure of processes, of the information used during the evaluation of capabilities.

#### *Addressing*

The hardware allows a process to have immediate access to the capabilities contained in up to 16 capability segments, though in the case of the CAP operating system 6 have been found sufficient. One could imagine that there were 16 registers in CAP which contained capabilities for the 16 capability segments; this is the effect although there are no such physical registers. The address of a word in memory must thus specify the identity of the capability segment containing the capability for the segment concerned, the offset of the slot there containing it, and finally the offset in the segment of the word required. Thus an instruction to add into the accumulator word 26 of a segment defined by a capability standing in capability segment 4 at offset 3 could be written

XADDS 4/3/26.

The number of the capability segment and the offset in it - 4/3 in this instance - are referred to as the *capability specifier*. Evaluation of the absolute address is performed at run-time by the capability unit (not by the microprogram) in a manner described later.

#### *The capabilities of a single process*

Associated with each process there is a fundamental segment called the Process Resource List (PRL), which

specifies at any time the total resources which could be available to any protected procedure in which the process could run. The manner of this specification will be described later. The capabilities in capability segments each contain a pointer to an entry in the PRL, so that the collection of capability segments constituting a protected procedure specify a selection from the total resources of the process. Provided that the access statuses involved are suitable, a capability may be copied from one capability segment to another belonging to the same process; this is done particularly when capabilities are passed as arguments from one protected procedure to another. Notice that copying a capability from a capability segment belonging to one process into a capability segment belonging to another cannot be allowed, since the pointer in it would be interpreted in the wrong context.

In addition to the PRL pointer, segment capabilities contain a relative base, a limit, and an access status. These permit capabilities to be refined as they are copied, so that it is possible to pass a capability for a part only of an accessible segment, or with a reduced access status, for example from RW to R only. These facilities are provided by the REFINE instruction, which takes as its arguments a source capability, a destination, a relative base, a refined limit, and an access reduction. (Fig. 1)

A process is fundamentally represented by its Process Resource List. Two of the entries in it (Fig. 2) must refer to segments which are used by the microprogram in the management of the process and in handling records of nested protection environments:

a) the process base, which is used to preserve the values of the processor registers and the description of the current protection environment when the process is not in execution, and which also is used to keep various other pieces of state information.

b) The C-stack, the function of which will be described shortly.

The PRL also contains an entry corresponding to itself; the architecture does not require this but the operating system does. Any PRL entry corresponding to a segment capability contains information which permits the calculation of the appropriate base, limit, and access status for adjustment using the values found in the capability segment, as above.

*Changes to the protection environment*

As indicated above, the protection environment of a process at any time is represented by the set of capability segments which are available for use. Accordingly the environment is changed by changing that set, which is done by the ENTER and RETURN instructions, making use of *enter capabilities*. The execution of an ENTER causes the values of five of the sixteen capability segments to change. Three of them (Nos. 4, 5, and 6) change so as to select new capability segments specified by the enter capability itself; operating system conventions have been established for their use and will be described later. It is these three capability segments which define the *protected procedure* specified by the enter capability, containing capabilities for its code, static data, and workspace. Capability segments nos. 2 and 3 are used in connection with the passage of capabilities as arguments and their delivery as results; in the course of an ENTER instruction the old capability segment 2 becomes inaccessible, the old capability segment 3 becomes the new segment 2, and the new capability segment 3 is initially undefined. An instruction is provided for creating a new capability segment no. 3 if required. Capabilities are passed as arguments by copying them to segment 3 with MOVECAP or REFINE, and capability results are found in segment 3 after the *status quo* is restored by a RETURN instruction. The RETURN instruction makes use of information preserved by ENTER on the Cstack mentioned earlier; that segment is also used to provide space for capability segments 2 & 3 themselves, by a mechanism which will be descibed below. The PRL entry corresponding to an enter capability specifies the offsets in the PRL of the entries corresponding to capability segments 4, 5, and 6 in the protected procedure associated with the enter capability. (Figure 3) An enter capability also has access bits, which appear in a machine register for inspection if desired in the new environment.

*The process hierarchy*

It is now possible to describe the process hierarchy and at last to complete the interpretation of PRL entries.

At the root of the hierarchy, called level 1, there is a single process. Its Process Resource List, called the Master Resource List, (MRL), is interpreted physically. Segment entries in it include an absolute base, and an over-riding limit and access status. The program in which this process runs is responsible for the creation and management of level 2 processes, and is called the Master Coordinator, (MC).

The PRL of an level 2 process is an ordinary data segment of the MC. A special instruction ENTER SUB-PROCESS (ESP) takes as argument a capability for a segment to be used as the process resource list of a junior process to be entered forthwith; the appropriate parts of segment-type entries in it will be interpreted as capability specifiers in the addressing environment of the coordinator at the time of execution of the ESP instruction. Other parts of segment-type entries will be interpreted as relative bases, limits, and access statuses. The microprogram, when loading a capability in

the junior process will, as indicated above, read the capability in the capability segment, interpret it relative to the process resource list, and then continue the cycle of relative interpretation via the coordinator's capability segments and the coordinator's resource list until absolute or physical information is reached. (Fig 4) Since all process resource list entries for a junior process are interpreted relative to the coordinator's address space at the time of execution of the ESP instruction, no breach of security is possible as a result of using this instruction. It is accordingly not in any way privileged, and a level 2 process may use it to create and enter a level 3 process, and so on until physical limitations of the equipment are reached.

In order that the coordinator may in due course be resumed, the ESP instruction first preserves in the coordinator's process base the current register values and a description of its current protection environment. The corresponding values and description are then picked up from the junior's process base and loaded into the appropriate machine registers.

Resumption of execution of the coordinator occurs as a result of any of:

1. the execution in the junior process of an ENTER COORDINATOR (EC) instruction
2. the occurrence of a trap detected by the microprogram during the execution of the junior process
3. the occurrence of an external interrupt, which always causes resumption of the Master Coordinator.

An additional argument to the ESP instruction specifies a register which, on resumption of the coordinator at the instruction next following the ESP, contains an indication of the cause of resumption, as above.

The structure for a particular process, consisting of the process resource list and capability segments as described above, is iterated in a manner corresponding to the depth in the process hierarchy to form the complete data structure employed in the evaluation of a segment capability. It should be emphasised that, at any particular time, the structure only exists for *current* processes, i.e. the process in execution and any others which are coordinators of current processes. It is not possible to write a program to exhibit the 'complete capability data structure for the system', since there is, at the level of the architecture, no such thing. Only those parts are identifiable which have been activated by an ESP instruction and not subsequently deactivated by resumption of the relevant coordinator. The microprogram could thus by no means load a capability belonging to a non-current process.

**The Capability Unit**

The capability unit contains the bases, limits, and access statuses of segments, computed by the microprogram from capabilities and ready for use. It includes a slave memory which can retain up to 64 such *evaluated capabilities*, in order to avoid intolerable repetition of the evaluation cycle. The slave memory works not on a pure associative principle, but by a combination of four-way parallel search and hardware hashing. The capability unit as a whole works autonomously of the microprogram processor, which is only responsible for setting up its contents.

The primary key to the search of the capability memory is the capability specifier part of a program-supplied address. The capability unit has logic additional to that associated with the search operations, the purpose of which is to allow an optimisation connected with the ENTER and RETURN instructions. When one of these instructions is executed, there is a switch to a new set of current capability segments. However it is arranged that any evaluated capabilities for the former set of capability segments, and for any other segments defined by means of them, may be left in the slave memory in such a manner that they will never be found by the associative search. Should any such capabilities still be present in the slave memory when the domain of protection to which they belong is reactivated, they are automatically re-enabled. The techniques used to effect this optimisation also permit evaluated capabilities belonging to several processes to be present in the slave memory at the same time.

When capabilities in capability segments are overwritten in the course of the REFINE instruction, or when a whole capability segment is destroyed, as in the case of the old capability segment no.3 in a RETURN instruction, the microprogram takes care that any obsolete evaluated capabilities are disabled. This extends also to any capabilities which are, because of the process hierarchy, descendants of that which was destroyed. The slave capability memory contains some information in relation to the immediate genesis of an evaluated capability in order to make this a practicable task for the microprogram. The aggregate of this information is, in fact, the physical representation of the current state of the process hierarchy. As was mentioned earlier, it is possible for there to exist data-type capabilities for the same memory area as there are capability type capabilities. This facility is only made available to a very small number of intimate system procedures, which are relied upon to use a FLUSH instruction when necessary to keep the slave store up to date. This instruction would also be used if programs altered the content of segment-type entries in the PRL, but no procedures in fact have occasion to do so. FLUSH is selective and if properly used does not clear out more than necessary.

As part of the CAP's initial load procedure, capabilities for the Master Resource List, its associated process base and capability segments, are evaluated and loaded into the slave memory. The overwriting rules for the slave memory are implemented by microprogram and ensure that the evaluated capabilities for the following are never overwritten:

the Master Resource List and the associated process base;

the process resource list and process base for the current process; and

capability segments from which evaluated segment capabilities are currently in the slave memory.

Otherwise overwriting is optimally arranged with regard to the hardware hash algorithm, except that capabilities for segments in the current domain of protection are preserved as long as possible.

**Input and output**

The remaining principal area of the architecture concerns input and output. All input and output memory accesses make use of the capability unit in the ordinary way, so that there is no necessity for special action, for example to ensure that computed absolute addresses remain valid throughout a transfer. Peripheral transfers are all mediated by a small peripheral computer, to the memory of which the CAP microprogram has direct access. The CAP process responsible for a certain device is activated when material is available in the peripheral computer's memory, and the actual transfer from one memory to the other takes place at maximum speed in the addressing environment of the device process. There is no autonomous channel (which would either need to work with absolute addresses, or would need to have a capability unit of its own), and there is no need for one since if the transfers happen at all they take place at maximum speed, fully occupying all equipment.

**Interface between the architecture and the operating system**

The commitment to protection facilities and the way the process hierarchy is defined do not commit the designer to any particular operating system structure. Great care has been taken in the implementation to ensure that the correct action of the microprogram does not depend on the correctness of any system data structures. Operating system errors naturally cause the machine to behave in a manner other than that desired, but they cannot, for example, cause the microprogram to go into a loop or behave unpredictably. This clearly desirable requirement has in fact had considerable effects on the design of the microprogram, making it both larger and slower that it would have been if system data structures had been relied upon; it has also affected the way in which some parts of the operating system are actuated. This section describes the way in which the interface between architecture and operating system has been set up and used.

*Processes*

Only two levels of process are used. At level 1 runs a single process, the Master Coordinator mentioned earlier. It is responsible for scheduling and dispatching level 2 processes. The level 2 processes include some which perform major functions of the operating system and also some belonging to users. A user could write a sub-system consisting of level 3 processes, his level 2 process functioning as their coordinator. It is a consequence, and a merit, of the particular capability-based system of protection used, that exactly the same facilities for safeguarding the integrity of his system are available to the writer of the sub-operating system working at levels 2 and 3 as were available to the designers of the main operating system working at levels 1 and 2. The main operating system takes no note, however, of the possibility of proliferation of levels since there is no way it can know that a level 2 - 3 system even exists.

The Master Coordinator has various functions additional to scheduling and dispatching. It is responsible for noting external interrupts, as described earlier, and for certain operations on capabilities which are required by the swapping function of the virtual memory. It is also responsible for the provision of synchronisation primitives, for some aspects of the creation of message channels between level 2 processes, and for some aspects of fault handling.

*Virtual Memory Interface*

To discuss the way in which the virtual memory system is connected with the architecture, we first recall the chain of references which is used when evaluating a capability on behalf of a level 2 process. (Fig.4) The microprogram reads from, in order, the level 2 capability segment; the level 2 PRL; a level 1 capability segment and finally the MRL or level 1 PRL. (See fig. 5) It is arranged that all level 2 PRL entries for a particular segment are defined in term of the same level 1 capability. It is called the leading capability for the segment. The leading capability selects a refined version of the memory area specified by an MRL entry, in the usual manner. A small number of MRL entries span the whole of the memory available for non-resident material. If a segment is to be swapped out or overwritten then the coordinator is caused to damage the leading capability so that the capability evaluation cycle will fail. The microprogram takes care of any evaluated capabilities which may need expunging from the slave store. If an attempt is made to evaluate a capability for a segment which is not in memory the microprogram finds that the evaluation cannot be completed and causes the coordinator to be entered. As

it does so it passes three words of information describing what has happened. It should be noted that the microprogram does not know that what has happened was an attempt to touch a segment which was not in memory; it is up to the operating system designer to choose exactly which type of loading failure he will use to indicate that contingency. There is a considerable latitude for the designer in his choice and he has a similar freedom about any other system function which is to be activated as the result of a trap. The coordinator does not know how to interpret the information passed to it by the microprogram; on receipt of this kind of entry its action is to cause a particular protected procedure, which is always resident, to be entered in in the level 2 process. This procedure is called FAULTPROC; and its task is to analyse the information words, and to make use of other information it can glean, to decide whether what has happened is to be interpreted as an attempt to touch a segment which is not in memory, or as indicating a need for some other system action, or as a plain blunder by the user, such as attempting to use an enter capability as a store capability. In the first case a message is sent to the Real Store Manager to do the needful; further details do not concern the architectural interface. It may be remarked that FAULTPROC is an example of a procedure in a level 2 process which is provided with a capability for the process's PRL - in this case a data-type capability with read-only access.

Another use of FAULTPROC is to cause new instances of protected procedures to be made when required. The mechanism is outlined in a later section.

## Use of Protected Procedures in the Operating System

The CAP operating system is implemented entirely in terms of sets of protected procedures, which are used by user processes or by dedicated processes of the system. In this section we discuss with examples the different ways in which the facilities provided by protected procedures are used. First of all we indicate the conventions which have been set up for the use of the various capabilityCs.

*Conventions for capability segments*

It was explained earlier how capability segments no.2 to no.6 are treated specially in the execution of ENTER and RETURN instructions, and how nos 4, 5, and 6 constitute the representation of a protected procedure They are used in system practice in the following ways:

No.4 is called the *P* (Program) capability segment. Here reside all capabilities which are present by virtue of the identity of the protected procedure, irrespective of by which process the procedure is used. Capabilities for the program segments of the procedure are P-capabilities.

No.5 is called the *I* (Interface) capability segment. It contains all capabilities which are present by virtue of the identities both of the process and of the procedure. I-capabilities represent the workspace of the protected procedure associated with the process that called it.

No.6 is called the *R* (Resource) capability segment. This is used in the implementation of protected objects. R-capabilities are specific to the object implemented by an enter capability and constitute its protected representation.

No.2 is called the *A* (Argument) capability segment. A-capabilities are those passed as arguments to a protected procedure from the protected procedure that called it.

No.3 is called the *N* (New argument) capability segment. N-capabilities are those which have been prepared for passage as arguments to a protected procedure about to be called, or which contain capability results as a product of that call.

**Types of service provided by system protected procedures**

The services provided by operating system protected procedures may be roughly classified into four varieties, which will be discussed in turn: gate-keeping, operating system intervention, protected objects, and trivial services. Since the mechanism is the same in all cases, the several uses of it differ in emphasis and purpose rather than in kind.

*Gatekeeping*

The majority of operating system calls fall into this category. It frequently happens that the criteria for accessibility or use of some service is more complex that that which could be encoded in a simple access status of a few bits. The approach adopted in the CAP operating system in cases of this sort is to place the capabilities required for the service in a protected procedure, to which the user has an enter capability. The protected procedure may validate the call in an arbitrarily complicated way. In the course of this checking, the procedure may take into account software capabilities presented by the caller as evidence of his right to some service. Accordingly, provision has been made for the reading of parts of capabilities as data - this does not constitute a security breach.

The foregoing indicates one kind of gatekeeping; there is another which has a very similar implementation and is not always distinguished. This is where there is no doubt as to the caller's right to request some service, but there is doubt about its trustworthiness to go through the detail required. Here the code of the protected procedure acting as gatekeeper is not validation code, it is detailed execution code. (Compare the office machine so ill-designed that only trained secretaries can be let use it.) We now give some examples of calls to protected procedures as gatekeepers.

1. Coordinator calls

Critical sections of code which must not be interrupted, such as process scheduling, are executed in the master coordinator and not in any subordinate process of the system. The ENTER COORDINATOR (EC) instruction is not privileged, i.e. one does not need any particular capability to use it, and may be executed by a user program at any time. The EC takes an integer argument to describe the service required. It turned out that only one of the coordinator services was straightforwardly available to any piece of code - 'wait event' and that all the others were privileged in some way, either because of the delicacy of the operation or because of its drastic character. Privilege is frequently indicated by the possession of a suitable software capability. To place the validation code in the coordinator itself would be very bad, since it not merely requires that interrupts be held off, which ought to be minimised, but also is made awkward because the checks should be interpreted in the addressing environment of the junior.

Accordingly the validation and sequencing routines were all moved into a protected procedure called ECPROC running within the caller's process. All requests for privileged coordinator services are programmed as calls to ECPROC, using standard argument passing mechanisms. All actual calls on the coordinator for critical sections of code are made by ECPROC, and the only check the coordinator has to do is to verify that the call really is from ECPROC. This is readily done by checking the identity of the P-capability segment, which is a constant for the process. The technique described has resulted in the removal of a great deal of complexity from a very sensitive program (the coordinator) at the cost of slightly greater overhead in some cases.

The interprocess message system shows the usage of ECPROC. The message system is implemented in terms of *channels*, and users are issued with software capabilities which represent the 'send' and 'receive' ends of the channels. Message sending and reception are carried out by calls to ECPROC which take as arguments the (software) capabilities for the message channels. ECPROC performs any transfer of data or capabilities which may be required, and then makes a call to the coordinator if any scheduling action is required.

2. Calls to other processes

Some operating system services are provided in dedicated system processes rather than as protected procedures in the user's process. This is done sometimes to give a simple serialisation of the operations, and sometimes to let the user's work proceed in parallel. Demands for system services provided in other processes are transmitted by the message system, but the message channel capabilities are held by a protected procedure rather than being directly available to the user. There are three reasons for this:
- The same Enter interface may be used for these calls as for other system calls;
- entry validation may be done in the calling process; this is especially important in cases
  where the system process supplies more services than every customer is entitled to;
- the format of messages does not have to be published, with advantages to the systems people and to the simplicity-seeking user.

An example of this kind of call is furnished by the *Interactive Stream Protected Procedure (ISPP)* which provides input and output access to serial peripherals. Each peripheral has a dedicated process which looks after, in a manner suited to the quirks of the device, such things as multiple buffering, lookahead, code translation, or local line editing. Other processes wishing to use serial peripherals are given enter capabilities for ISPP's which validate calls and do the message handling in a reasonably optimal manner. They also provide the same programming interface as the *Spooled Stream Protected Procedure (SSPP)* which does stream transfers to and from files. It is thus made easy for programs to be unaware of the difference between the two types of stream.

3. Operations on system data structures

Some system-wide data structures are shared between all processes. The integrity of each such structure is entrusted to a specially-written protected procedure which has exclusive custody of the capabilities for the data structure. All operations on the data structure are handled by calling on the procedure responsible for it. The continued integrity of the data structure is thus a matter of ensuring that the single procedure which operates on it is right. It is worth remarking that we distinguish this case form the somewhat similar one of protected objects because the system-wide data structures are usually unique and specially generated, not exemplars of a class which may have many members.

The case of the message system and ECPROC has been

discussed already; ECPROC is the procedure, available to all processes, responsible for the *message segment*, a system-wide structure of channels and messages in transit. Another case is the *System Internal Name Directory (SINDIRECTORY)* which, amongst other things, maintains reference counts for capabilities preserved in the filing system. The SINDIRECTORY is only operated on by the SINMAN procedure.

*Operating System Intervention*

There are times when a process must enter the operating system involuntarily, particularly after a fault or a non-deterministic event such as a virtual memory trap. In order to preserve the principle that the operating system should have as little access to user capabilities as possible, the entry takes the form of simulating the effect of an ENTER at the point of the fault or trap into a special protected procedure which inspects the fault and decides what is to be done. The only example of this type of procedure is FAULTPROC, into which a process is diverted whenever the hardware or microprogram detects a trap. The coordinator preserves the state of the process and causes the FAULTPROC entry by editing the dumped copy of the protection state of the process. The use made of this facility was described earlier.

*Protected Objects*

A file directory is an example of a protected object. A user is interested in a file directory only in so far as he is able to use it to preserve capabilities in and to retrieve capabilties from. The internal organisation of the directory is of no interest to him. He does however require to be able to pass a capability for the directory to other protected procedures in order that these procedures may use his preserved capabilities by name.

In the CAP operating system, protected objects are implemented by protected procedures. Possession of an enter capability for the procedure implementing a protected object is the basic qualification for access to the object itself; the degree of access available may be refined by the use of access bits in the enter capability itself or by restrictions encoded in the procedure. For instance, if one has a capability for a directory with access status sufficient to preserve, retrieve, and remove capabilities, the REFINE instruction may be used to mask out some access bits, so producing a new capability for the directory which may only be used for, for example, retrieve operations. The new capability may then be passed as argument to some protected procedure in the confidence that no damage can be done to the contents of the directory.

A protected object has two components:
- the program embedded in the protected procedure
- the representation of the object.

The program is the same for all instances of protected objects of the same type, whereas the representation is unique to an instance. As indicated earlier, the R-capability segment is used to contain capabilities which belong to the representation of a particular object.

For example, all directories in the system are protected procedures with identical P-capability segments (containing the capabilities for the DIRMAN program), with R-capabilities describing the representation of a particular directory. It has been arranged that all directory procedures of a particular process have the same I-capability segment, containing capabilities for short-term workspace. If capabilities for directories are themselves preserved in other directories, only the representational part needs to be retained, together of course with a notation that the type of the retained object is 'directory'. On retrieval of such an object, a new protected procedure is made with R-capabilities for the representation part and standard P and I capability segments. The resulting enter capability is reatuned to the caller as the directory capability requested. It is usually the case that, as here, only the R-capability segment needs to be created when creating a protected object.

*Trivial Services*

It was planned that the operating system would be built up out of protected procedures, with protection needs as the basis for procedural subdivision; in practice the protected procedure has become the natural unit of modularity in programming for the CAP *whether or not* there is a protection need in any particular case. This modularity is encouraged by the use of high-level language systems which have become geared up to the production of protected procedures.

Some examples of trivial services, available as protected procedures although there is really nothing to protect are:
- a program for performing elementary syntax analysis and parameter decoding on the string provided as argument to many user programs;
- a program which, when given an operating system fault number, returns a string describing the fault, suitable for exhibiting to a user.

**Creation of Enter Capabilities**

A process's stock of enter capabilities varies from time to time according to its requirements. Some protected objects, in particular, have short lifetimes and enter capabilities for them are created and destroyed frquently. A process may obtain an enter capability either by *dynamic creation* of the capability or by *retrieval* of the capability

from the filing system. In both cases, the creation of a new PRL entry is involved.

*Dynamic creation of enter capabilities*

Any user in the system may request that an enter capability be created if he is able to supply capabilities for two (P & I) or three (P, I, & R) capability segments. A system protected procedure called MAKEENTER is supplied for this purpose. It takes as arguments the two or three capabilities for capability segments and returns as a result an enter capability selecting a newly made PRL entry containing pointer fields for the two (P,I) or three (P,I,R) capability segments supplied by the user.

Internally, MAKEENTER has a software capability permitting it to make capability interrogation and creation calls to ECPROC. MAKEENTER calls ECPROC once for each argument capability in order to ascertain the PRL offsets of the relevant PRL entries. MAKEENTER is then able to put together the two words of the PRL entry required, and pass them to ECPROC to allocate and fill in the PRL. ECPROC returns an enter capability, referring to the newly created PRL entry, which is simply passed back to the user.

Inside ECPROC, the capability and interrogation calls are implemented by means of the privilege of having data-type capabilities for current capability segments and the PRL. ECPROC is thus very highly privileged indeed.

Assuming that MAKEENTER and ECPROC are correct, the process has in no way extended its privileges by encapsulating previously owned capabilities in a new protected procedure. The most frequent application of MAKEENTER is in the manufacture of protected objects.

*Enter capabilities retrieved from the filing system*

A protected procedure is represented in the filing system by a segment containing a prescription for the construction of an enter capability. Such a segment is known as a *Procedure Control Block (PCB).* The information in a PCB specifies the required size and content of the capability segments, either two or three in number, which have to be constructed to produce an appropriate PRL entry for the enter capability (Fig. 3). Retrieval of enter capabilities takes place in two stages. When a directory manager is caused to retrieve an enter capability, the immediate result is a capability of enter type, selecting a PRL entry of *segment* type, which specifies the PCB segment. This configuration is called an *unlinked enter.* An ENTER instruction causes a trap when an unlinked enter is used, since an enter-type capability in a capability segment is not allowed to select a segment-type PRL entry. The effect of the trap is to cause FAULTPROC to be entered in just the same way as for virtual memory traps. When FAULTPROC has determined the cause of the trap, it makes use of special capabilities to construct a proper segment-type capability for the PCB. This it passes in the ordinary way to a comparatively unprivileged protected procedure called LINKER, whose task is to interpret the PCB and return as results capabilities for the capability segments needed to complete the enter capability. FAULTPROC completes the PRL entry in the manner of MAKEENTER, except that it is altering an existing PRL entry rather than making a new one. Finally FAULTPROC executes a RETURN instruction, and the original ENTER is retried. Any other unlinked enter capabilities held by the process which refer to the same PRL entry will have been completed by this operation.

Some of the capabilities specified in a PCB may themselves be enter capabilities. The two-stage method of retrieval of such capabilities avoids long and complex recursion in LINKER, and also ensures that enter capabilities are only completed if they are actually used, which avoids potential waste of effort.

PCB's are made by an operation converse to that of the LINKER and constitute a system type, known to the filing system in the same way as are file directories. A system protected procedure called MAKEPACK is given the privilege of handling PCB's as segments (achieved by giving it a particular kind of enter capability for the System Internal Name Manager). Anyone may obtain an empty PCB and pass it to MAKEPACK; MAKEPACK obtains access to it as a segment, and then accepts capabilities passed to it with statements of what is to be done with them. For example, "First argument capability to be used for capability P0 of the new procedure". The effect of this is to ensure that until the PCB is destroyed the object specified by the offered capability cannot be deleted, and will be retrieved when the PCB is linked. Facilities are also provided for specifying an object by textual file name; doing so does not guarantee the continued existence of the object. Software capabilities may be passed to MAKEPACK like any others; it is also possible, by making a request such as "Capability 13 workspace 1k read/write", to specify that a particular capability should be for a scratch segment to be generated at link time.

MAKEPACK is a powerful procedure since it is responsible for the integrity of many system procedures with a variety of privileges. However it has no other privileges itself than the one mentioned, and cannot perform any other non-public system operations.

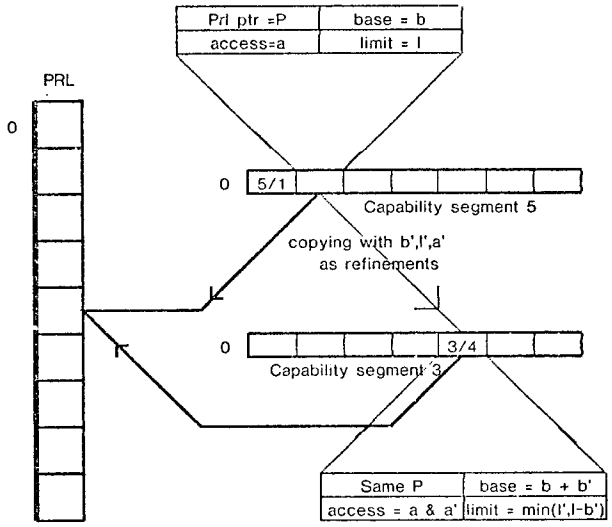
An enter capability for MAKEPACK is globally available, since the input data are thoroughly checked and in no way sensitive. A second protected procedure called MAKEPCB is provided to furnish a convenient interface to

a terminal user for the creation and editing of PCBs. MAKEPCB is entirely unprivileged and exists because of a desire to separate programs responsible for the user interface, which one may wish to alter as tastes change, from privileged programs which should only be altered if found to be wrong.


## Acknowledgements

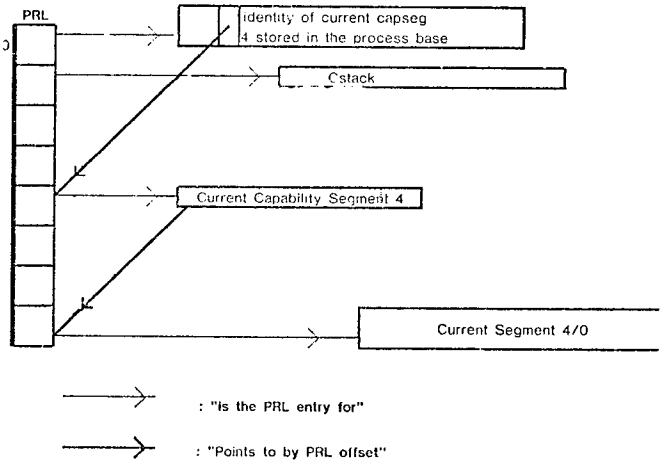
The CAP project is supported by the Science Research Council. We have received constant encouragement and advice from M.V.Wilkes; D.J.Wheeler carried out the engineering design with great energy and skill; the staff of the Laboratory's workshops, in particular N.W.Unwin, V.Claydon, and D.B.Prince, have constructed a very reliable and attractive piece of equipment. We are obliged to Butler Lampson for many helpful comments on the exposition.


Fig. 1 Two capability segments and the PRL



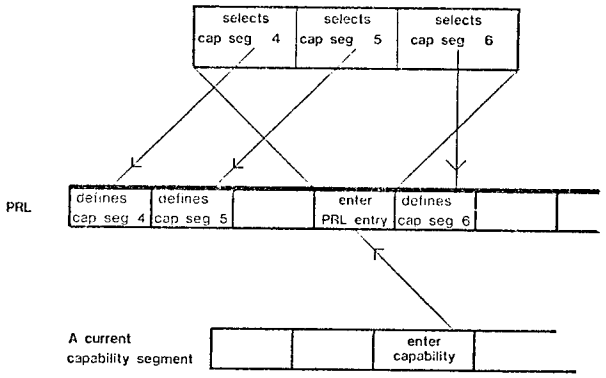

Capability 5/1 is refined to capability 3/4, using the same PRL entry although size and/or access may be reduced

Fig. 2 Part of the structure of a process



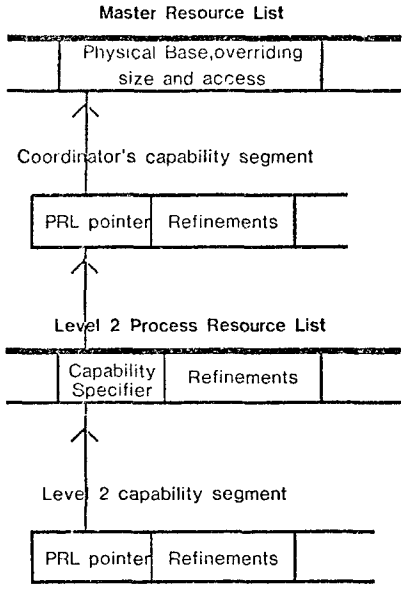

Other capability segments are pointed at similarly

Fig 3



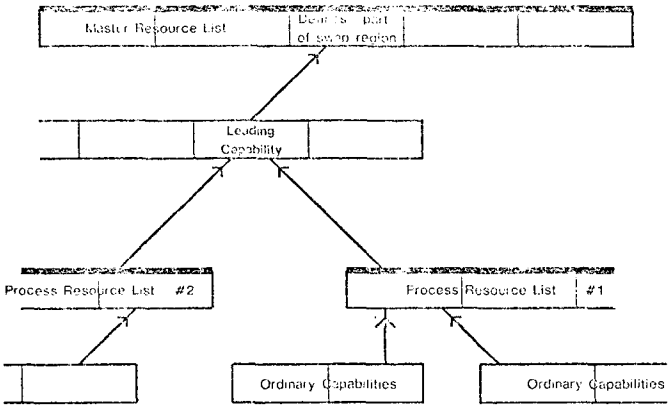

In the example, the third entry of the current capability segment is an enter capability The PRL entry, expanded for clarity, selects the three capability segments of the protected procedure defined by the enter capability .

Fig 4



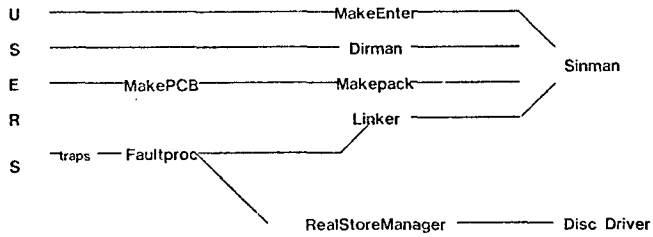

The chain of references for loading a capability for a level 2 process

Fig 5




The leading capability is on the evaluation route for all instances of capabilities for a particular segment. The MRL entry simply selects a core region out of which the segment was allocated.

Fig 6




The relationships between the main programs used as examples in the paper; all procedures call ECPROC but with varying privileges conveyed by software capabilities