Protection systems and protection implementations

by R. M. NEEDHAM

University of Cambridge
Cambridge, England

INTRODUCTION

The paper discusses the nature of systems for protection of information in the central memory of a computer, describing the potentialities and limitations of a variety of approaches. It is based upon work done in the course of a current project on protection systems at the Computer Laboratory, Cambridge, and outlines a system which is being developed to the point of hardware implementation in the Laboratory.

PROTECTION SYSTEMS AND PROTECTION IMPLEMENTATIONS

For the purpose of this paper Protection is understood to refer to logical and physical mechanisms for controlling access to data in the central memory of the computer. The purpose of protection systems is to insure that at any point in the execution of a job by means of the computer, only those data objects which require to be accessible are accessible, and that this access is only of the mode, for example reading only permitted, which is required for performance of the task in hand. The object of work on protection systems is to devise mechanisms which will afford protection to the greatest extent possible, and do so without excessive expense in hardware, runtime, or program size. The hope is that if such mechanisms can be devised, then it will be very much easier to contain and to localize the consequences of hardware or software failure, and to know much more precisely than is the case at present which of the activities in which a computer is engaged must be suspected of having been spoiled by the failure, and must therefore be re-initiated.

In order to get any rationale for a protection implementation, we must set up some defined concepts in terms of which protection systems can be discussed. The first of these is that of the segment, the unit of information to which protection applies. A segment is a set of words whose addresses are contiguous in a virtual address space, and whose protection status is at all times the same. Protection is thus intimately bound up with addressing, since our very definition of the unit of protection is in terms of an addressing mechanism. This approach allows us to specify a protection regime by giving a list of those segments accessible to a process at a particular time, together with notes as to the kind of access which is permitted. A somewhat minimal protection regime could then be described by saying that segment A contains data to which read-write access is permitted, while the words of segment B may only be executed as instructions. A major object of research in protection implementations is to propose mechanisms whereby any desired protection regime can be implemented, with as few limitations as possible imposed by the engineering approach adopted.

Protection regimes are not constant during the life of a process. They may change as the work proceeds, and in a fully general discussion they should be allowed to change arbitrarily. Statements would be allowed, for example, to the effect that certain segments were only accessible if the value standing in a system micro-second clock were prime. In practice, one departs from full generality, and limits those circumstances which may give rise to a change of protection regime. A reasonable approximation is to say that changes of protection regime are associated with changes of the segment from which instructions are currently being extracted; this is not to say that such segment changes must necessarily give rise to changes of protection regime, but only that no change of protection regime may occur without a change in the program segment.

The first proposals for the physical design of a processor which took these ideas seriously were by Yngve and Fabry. A summary of their ideas will be found in Wilkes. The essential aspect of these proposals was that there was no restriction on them imposed by any of the implementation techniques. It was thus possible to arrange, in principle, that a process's capability list

571
always contained exactly and only what it should. Yngve and Fabry adopted the same approach to change of protection regimes as we have, namely that it only occurred when there was, additionally, a change of the program segment. A special instruction, called ENTER, caused a complete replacement of the process's capability list, and could thus change the protection regime of the process in an arbitrary way.

In a capability system of the type just described there are two problems calling for further discussion. First, if a capability indicates the absolute store address of the segment to which it refers, there is the problem of updating all copies of a capability when the segment is shifted in memory, and in deleting all copies when, and only when, the segment is destroyed. An obvious solution is to centralize the lists of absolute capabilities, and replace the capability lists associated with running process by lists of pointers to the central list. This is more than a simple technological device because it conceptually replaces the current capability lists of a program by a mechanism which selects from a larger list. This selection function has come to seem more and more important to us. Secondly, the original proposals dealt rather clumsily with pieces of data which were the property of a process, in the sense that if the process were deleted the data would go too, but which were only accessible when the right pieces of code were being executed. On the other hand, the proposals dealt very elegantly with bundles of capabilities which invariably became accessible when a certain piece of code was used, regardless of the process using it. The idea that will be developed is that the capability list of a process is to be regarded as that which defines a selection from all the absolute capabilities that exist; at any time in the history of the process some other mechanisms make further selections from the capabilities of the process, the selected capabilities being physically accessible in virtue of the current protection environment. Thus we have the idea of multiple levels of selection.

We may now focus on the implementation of protection as the implementation of selection functions among capabilities, where by a capability we mean that which defines the physical position and size of a segment and the access mode allowed. Immediately there are two ways to proceed, which depend on the extent to which addressing is brought further into the protection implementation. One way is to proceed by means of lock and key systems. A lock and key system is one in which any segment, including here a segment containing capabilities, has associated with it a lock. At any stage in the history of a process there is associated with the process a key. Access to a segment is permitted if, and only if, the current key fits the lock of the segment. A lock and key system tends to separate the notions of addressing and of protection. In such an approach, a process may address any segment whatsoever; only those in which the key fits the lock will do other than give rise to violations. There is no relationship between the mechanisms for addressing a given word and the mechanisms for addressing a given word and the mechanisms for validating access other than that which is implicit in the segment being the unit of protection. Accordingly, it becomes feasible to arrange that a segment has the same name, that is to say it is addressed in the same manner, throughout the lifetime of the process or even to go further and to say that all segments are uniquely identified in the computer. This approach has much merit in that it avoids any renaming problems when communication is involved. Unfortunately, it proves extremely difficult to set up lock and key systems which are of sufficient generality to achieve the desired results. Because of the great potential advantages of lock and key systems, the reasons why this is so merit some examination.

LOCKS AND KEYS

Consider a situation in which all distinct protection regimes which can ever occur are identified by name or number. One could then imagine a lock and key system in which the key consists simply of the name of the current protection regime and the lock associated with the segment consisted of a list of the names or numbers of all protection regimes in which the segment was accessible, together with the nature of the access permitted. It is clear that this places no restrictions at all on the variety of accessibility patterns which can be implemented. It is, however, a very expensive thing to consider doing; there is no convenient limitation which can be set on the length of the lock, and the process of consulting it to see whether a particular key matched would be extremely slow. All practical lock and key systems which have been proposed work by means of some sort of encoding scheme, the purpose of which is to reduce the locks and keys to a fixed and convenient size. Any such encoding scheme regards the lock and key as being bit patterns between which a certain relation is sought. For example the lock and key may be two parts of a single valid message unit in an error correcting code. If we take this as an example, we see that every lock has to have the right relationship to each key which is supposed to fit it. If we now take a particular lock, it is possible in virtue of the structure of the relationship we have constructed to list in principle, all of the keys which will open it; equally every key can be accompanied by a list of those locks which it will open. We can think of listing out the possible locks and
drawing lines from each pointing to the appropriate keys, and also putting in lines in the inverse sense from the keys to their locks. We shall be able to express the total variety of protection regimes we are interested in if, and only if, we can make an assignment of locks and keys to the segments in the regimes in such a manner that invalid access is never allowed. This poses an extremely difficult combinatorial problem in all non-trivial cases. It is at the least an extensive task to find allocations which satisfy all the constraints and, even if one can succeed in doing so, a small change in the protection regimes to be implemented may result in a total upset to the lock and key allocation. It appears that one either has to put up with the necessity for computing allocations of locks and keys, or alternatively to accept a lock and key system which will not implement all the protection regimes which might be required. One can sum up by saying that sufficiently powerful lock and key systems are too difficult in practice because of the allocation problem, and that lock and key systems in which one can face the allocation problem are not powerful enough. A good example is the plain hierarchical protection system afforded by representing the locks and keys by small integers, and saying that access is permitted if, for example, the key is less than or equal to, the lock. This is easy to think about and easy to implement; unfortunately, it places extreme restrictions on the protection regimes which can be described. If in protection regime A, something has to be accessible which was not accessible in protection regime B, then necessarily everything which is accessible in B must be accessible in A too. It is just not possible to deal with some situations which occur commonly in practice, such as the following. Suppose there is an input program \( P_1 \) which has to have access to an input buffer \( B_1 \); suppose further that there is an output program \( P_o \) which has to have access to an output buffer \( B_o \). It is not possible to arrange that each of these programs has access to its buffer but not to either of the others. These are simple consequences of the linear arrangement of privilege.

An additional difficulty about simple lock and key systems is that they do not deal satisfactorily with the non-static and unpredictable nature of protection regimes. Arguments which have been passed to a program which runs in a particular protection regime may carry with them the requirement that during running certain segments are accessible because they contain the data passed and not because they are permanently associated with the called program. A simple hierarchic system is in no difficulty if the new protection regime is further up the hierarchy than the previous one, but it is in very serious difficulty if the new protection regime is lower down than the previous one. The more one elaborates lock and key systems, the more this problem becomes a troublesome addition to the allocation problem mentioned before. For these reasons, after a great deal of investigation, we did for the time being abandon the use of lock and key systems as a means of implementing the selection we desire.

**SELECTION BY INDIRECTION**

As foreshadowed above, the obvious alternative means of selection for accessible segments is by the use of indirection tables. If all segments are accessed via an indirection table or via one of a set of indirection tables, then it is possible to constrain the selection of available segments in quite arbitrary ways by suitable construction of the indirection tables. A consequence of the use of indirection tables is that addressing has become much more bound up with the protection implementation. This can be seen by looking at the complete specification for getting at a word of core. In order to specify a work, we must give three pieces of information:

1. which indirection table must be used,
2. which entry in that indirection table indicates the required segment,
3. which word in that segment is wanted.

The first two of these will be called the *segment specifier*, and the three collectively an *address*.

The segment specifier of a segment depends on the protection regime, and so in turn does the address of a word. If the protection regime changes, a new set of indirection tables will be brought into use, and the addresses of words will in general change too. What changes is not the segment itself, nor is it the capability for the segment; the change is to the means of finding the capability.

This point is of the greatest importance, and it is worth recapitulating it in a sharp form. On one side of the divide we have systems such as those which rely totally on locks and keys, where if a program attempts to load the accumulator with the contents of a certain word, then the actions it undertakes are in all circumstances the same, regardless of protection regime, although in some protection regimes they may cause a violation. On the other side of the divide, we have systems in which protection is so bound up with addressing that the bit pattern to be presented in order to load a certain word into the accumulator differs according to the current protection regime. The latter approach gives the flexibility which we have been unable to achieve in the former. However, if the mode of
addressing words or segments is influenced by the protection environment in force, then there are complications in the compilation process that do not arise in a system with permanent segment addressing. Secondly, one gets into some difficulties with pointers from one segment to another. If we have a data structure which exists in more than one segment, some of the pointers in one segment will point to places in another segment. If the specifier of the segment changes, we are in difficulty. Although this does not happen very often, a solution must be found. The non-uniformity of treatment of pointers is something which compiler writers dislike since the existence of the non-uniformity may not be evident at a convenient time in the compilation process.

Bearing in mind the above points, we now look at methods of implementation of systems which rely upon indirection to perform selection. The principal choice we have considered is between a system with explicitly named capability registers, and one without. A system with explicitly named capability registers works in the following way. A number of registers are provided, usually about eight, each of which is able to contain a capability for a segment in absolute form. Typically this consists of a base, a limit, and an access code. A process is at any time equipped with one or more capability segments, which contain either absolute capabilities, or information from which absolute capabilities may be found or constructed. The system has an instruction called 'load capability register' which has two arguments. The first argument is the number of a capability register to be loaded, and the second is an indication of which capability is to be loaded there. It must indicate which capability segment to use if there is more than one, and which entry in the selected capability segment should be used. A store reference instruction will then be interpreted via a capability register. A subsidiary point is whether or not the selection of which capability register to use is part of the address field of the instruction or part of the function field. The significance of this point is whether or not the capability register selection can be changed by index modification. Take first the case where the capability register selection cannot be changed by index modification. In this case a particular instruction in the program has it fixed for all time which register is going to be used. This approach imposes a rather considerable lack of flexibility. Some of this lack of flexibility is associated with any explicit capability register scheme, and will be mentioned in a moment. One aspect, however, is unique to this approach, namely that it is impossible to have a pointer from one segment into another. There is no uniform way of writing a program which will follow a chain searching for something, if that chain is likely to pass through words of more than one segment. It was remarked above that there are difficulties in this area anyway, and possibly the solution to the problem is to decide that intersegment pointers should be disallowed.

Turning now to the alternative case where the capability register selector can be altered by index modification, we see that the particular difficulty just referred to does not arise. Provided that the capabilities for the segments in which the data structure resides are loaded, and known to be loaded, into the correct registers (where 'correct' means the ones which were assumed when the points were set up), then inter-segment pointers are perfectly possible. This proviso, however, indicates the lack of flexibility which remains. A great deal of pre-allocation of capability registers has to be done in any system which refers to them explicitly. Furthermore, an instruction will only be correctly executed if the right capability register has been loaded. Unless there are sufficient capability registers, which may be rather a lot, there is a good deal of keeping track to be done to insure that at all times the correct capabilities are where they should be as the flow of control proceeds round the program. For example, it may be desirable to pass the address of a word around in a program at a time when a capability for the segment containing it is not necessarily loaded. There is of course no need for the capability to be loaded until the address is actually used. We find that we need, in effect, two sorts of address which can be described as a particular address and a general address. A particular address consists of a capability register number and an offset. It is valid in all circumstances in which the capability register has been properly loaded. A general address consists of a complete segment specifier and offset; the segment specifier is just the second argument of a 'load capability register' instruction. If a piece of program, say a sub-routine, receives a general address, it is in a position to load the indicated capability into whichever capability register it thinks fit. However, in this case also we have difficulties of compilation, because the compiler cannot know when to use general addresses and when to use particular addresses. Furthermore, considerations of economy would suggest that we do not need two forms of address; of the two it is clear that the general one should be retained.

THE SYSTEM PROPOSED

This final remark leads to the outline of the system we have eventually proposed. There are no explicitly named and explicitly loaded capability registers; instead the general address as defined in the last paragraph is interpreted directly by the hardware. The hardware
must internally have registers in which absolute capabilities are to be found, and what it does, when presented with a general address, is to test whether the absolute capability corresponding to the segment specifier part of the general address has already been loaded into one of the internal registers. There are a variety of ways of doing this at hardware level. We are now in a position where programs only use addresses in the form 'segment specifier, offset', and the runtime interpretation of the segment specifier is buried beneath the hardware-software interface. We must remember, however, that the interpretation of a segment specifier will still depend on the protection regime, because it makes use of indirection tables as a means of selection.

It is now time to return to a question implied about, namely how many indirection tables there should be and what they should be used for. The structure we are talking about is sketched in outline in Figure 1.

In this structure, a change of protection regime will be implemented either by changing the contents of the indirection tables, or by bringing into use new indirection tables and putting out of use old ones. Some things are most naturally done by amending the contents of indirection tables. For example, a system call to give the process a brand new segment results in a change to the protection environment which is most easily made by extending a presently existing indirection table. The call has said something like 'get me a new segment of size n and call it Jack' where Jack is a segment specifier. The consequence will be that the appropriate indirection table entry will be set. On the other hand, when protection regimes change not by giving the process new resources but by changing the accessibility of the resources already given, it is expedient to bring new (but pre-existent) tables into use and similarly to dispose (temporarily) of old ones. We have chosen to classify the segments available for a process at any time into four classes, implying that there are four current indirection tables:

1. Segments which are available to the process regardless of which program is currently being executed; these are known as G for global.
2. Segments which contain the code, or alternatively read only data, for a current program; these are called P for program.
3. Segments which, although the property of the process, are only accessible within the current program; these are called type CP.
4. Segments which are accessible because they have been passed to the current program from the program which called it; these are called type A for argument.

For example, consider a program package whose duty it is to perform an input/output operation, such as taking a string of characters away from the calling program, despositing it in a buffer, and subsequently disposing of it. The code of the package may read from or write to the calling program's data area, it will require to be passed capabilities which will be A type. If in the course of executing this package it is necessary to make calls to the generally available operating system facilities, the ENTER capabilities for these facilities will probably be capabilities of type G. If, however, the system calls may only be made from within the input/output package we are describing, those ENTER capabilities could be either of P type or of CP type. The action of an ENTER instruction will thus be to change three of the four indirection tables. The table G will not be changed, because it is always available. The P indirection table will be replaced by one which is the defining characteristic of the called package; everything referred to in the P table will be shareable between all users of this procedure. The existing CP table will be replaced on ENTER by one set up to have the required properties at the time when the procedure was made available to the process. Making a procedure available to the process thus consists of equipping the process with the required ENTER capability and with the required indirection tables. The A indirection table will be replaced by one which is characteristic of this particular call. It is convenient to place A indirection tables on a special stack of standard format and distinguished from any stack that the running program may create for its own purposes. The special stack can also be used to store the links associated with ENTER instructions. Specifically, if an A type indirection table is constructed before a call, it will be the top few words of the stack. One or two special instructions are provided for moving pointers to capabilities from one indirection table to another, and...
one of these is specifically used for establishing entries in what will be a new argument type indirection table. It is worth noting that in the system proposed material other than that in global segments will only be available to called programs if appropriate capabilities are explicitly passed. There is inevitably a slight overhead on calls, but this is unavoidable in any system which does not have hierarchical protection. In hierarchical systems, it is usually assumed that when a call is made to a more privileged regime (and most calls are like this) everything which was previously available is still available.

We are now in a position to give some account of the protection system as it appears when a process is running without any reference to problems of inter-process communication or of coordination. At any time the protection regime is represented by the current settings of the four indirection tables. Some of the capabilities referred to in these indirection tables will be ENTER capabilities; these delineate those changes of protection regime which are immediately possible. When one of the ENTER capabilities is exercised by means of the ENTER instruction, the protection regime changes and the P, CP, A indirection tables are all replaced. We thus see that an ENTER capability must specify, directly or indirectly, the capabilities for the two new indirection tables of the P and CP types, the A type being part of a stack as previously described. What an ENTER capability actually looks like in a process capability segment is an implementation decision.

We can now look at the same questions from another angle, and consider how to construct a protected procedure—that is, a procedure which will be entered with an ENTER instruction and which will run in its own protection regime.

A protected procedure is characterized by its P- and CP-indirection tables. Accordingly, to construct one we must construct these tables, and insure that there are in the process's capability segment the correct capabilities for the indirection tables to select. A specimen prescription for such a procedure could look something like this:

"There are 4 entries in the P-table. The first must select a segment of program whose text-name is Peter and the only access needed is 'execute.' The second selects a translation table called Bill, and 'read' access is required. The third and fourth must select ENTER capabilities for two standard system functions.

"There are 2 entries in the CP-table. One is for local workspace of the procedure, and should be a copy of named segment Alfred, which contains initial data values. It must be readable and writeable. The second must be a workspace segment to use as a buffer, readable and writable, and 1000 words long."

A routine that interprets this prescription and sets up an ENTER capability for the procedure in question then takes the following actions. First it procures suitable segments in which to build the indirection tables, and then it sets about filling them in. In the case of a workspace segment, whose initial contents do not matter, all that is necessary is to ask the core management routine for a segment of a suitable size and set the appropriate pointer in the indirection table. In the case of a segment whose initial contents must be set from a file, then the file system must be consulted in order to discover the segment size and disc address. There is a third possibility, namely that the prescription is for a segment already known to the process, and in this case the insertion of a new pointer is all that is needed. The two entries for standard system functions mentioned above would very likely fall under this case. Since the purpose of the routine is to equip a process with a new ENTER capability, it may be convenient to write it so that it can act recursively when the prescription itself calls for ENTER capabilities. The final action of the routine is to construct the ENTER capability which was originally requested, and leave a pointer to it in a suitable place.

In this approach the protection procedure is regarded as a totally encapsulated entity which can be incorporated into the environment of a process without any presuppositions as to what was there already. If parts of the (read- or execute-only) environment were present already, then they will be re-used. It is open to take a slightly different approach and to construct protected procedures on the assumption that, for all processes, certain standard functions are available through the G-indirection table, this being always accessible. Doing this makes P-indirection tables shorter, but requires more conventions as to the way processes are set up.

THE PROBLEM OF INVALID ARGUMENTS

It is common for one or more of the arguments of a call to a protected procedure (or indeed any procedure) to be the address of a piece of store to which the procedure will write. There is no protection problem if the store so addressed is accessible to the calling program; potential difficulties arise if it is not so accessible but would be accessible to the called program. As a concrete example, suppose that in a traditional computer where the supervisor runs in a privileged mode, all memory being accessible, there is a system call to read n words from an input document to store starting at address a.
If a user program executes this call, giving as argument an address in store available to itself, there is no problem; what, however, if the address is that of store inaccessible to the user, but accessible to the supervisor? Unless precautions are taken, the supervisor may, when presented with an invalid argument, over-write its own program or important data. This problem is not new; there are explicit counter-measures to it in the hardware of the Atlas. However, the more generalized one's approach, the more difficult it is likely to be to deal with this class of difficulty.

In a system with explicitly named capability registers, and in which the capability register number is in the function part of an instruction (i.e., it cannot be altered by address modification) the problem cannot arise. This is because any address passed as an argument will only be interpreted by the called program as referring to an authorized segment, and no possible action can mislead it. As soon as we move to a system in which the capability register number or the segment specifier are parts of the address passed, then there is the possibility of trouble. Difficulties of this sort arise in any system in which indirect references to segment names or numbers are possible.

In order to guard against the danger just referred to a check must be made which depends on a number of different pieces of information being available at the same time. We must know:

1. The protection regime in which the address was constructed;
2. The protection regime in which the word referred to by the address is accessible;
3. Whether it is allowable to construct an address in the former regime which refers to a word in the latter.

In the structure outlined above, where there are four indirection tables, a simple rule results as follows: an address residing in a segment of type G or A may not specify a word in a segment of type P or I. The difficulty comes in knowing when the rule is being broken. As an example of a common sequence in which the relevant information is not all at hand at once, consider:

- Load index register from store
- Access store via index register

Item 1 above is available in the first instruction, but it is not then known that the word will be used as an address. Item 2 is known on the second instruction, but not where the contents of the index register came from. Any approach, for example the use of indirect instructions, which has both pieces of information available at the same time will enable the problem to be solved, e.g.,

'Load accumulator indirectly from store'

because both addresses and hence both segment types are known in the course of the same instruction, or

'Validate stored address'

which is like 'Load accumulator indirectly' except that it does not load the accumulator, but only checks that the address in store obeys the rules. A really satisfactory solution to the problem of invalid argument addresses would not place on programming style the constraints which are imposed by the compulsory use of indirect or validation instructions. Such a solution is not yet obviously available.

The body of this paper has been concerned with protection systems within a process. Nothing has been said about how the process obtains its resources and from where. There follows a brief view on how this aspect of a system may be organized.

The time available to a process is administered by a superior process called its coordinator. The coordinator is responsible for allocating time to its junior processes, and for synchronizing their execution where necessary by managing their halting and freeing. In addition to being the source of time allocation, the coordinator has responsibility for space allocation. Finally, any process may act as coordinator for processes junior to itself.

This view has consequences for protection. The within-process protection architecture discussed above aids the orderly use of the process's resources, and all privileges conferred on particular procedures are relative privileges within the general facilities available to a process. Since all facilities available to a process are mediated by the coordinator, the last statement implies that privileges are valid within the universe set up by a coordinator for its junior processes, this universe being a subset of that available to the coordinator itself.

It is a consequence of these remarks that privileges enjoyed by a coordinator in virtue of its relationship to its superior may not be passed on to the coordinator's juniors. They exist in the wrong world.

The result then is that a coordinator may pass to its junior processes, when setting them up or later, access to core segments or subsegments available to it, with or without further access restrictions. It may not pass an 'ENTER' capability at all, though it may be able to pass the use of pieces of code from which an ENTER capability can be constructed for the junior. Since the coordinator has complete control over the actions of its
junior processes, including interfering with register settings during halts or after interrupts, passing an ENTER capability could allow the coordinator to perform, via a subordinate, action which would ordinarily be forbidden. The ENTER refers not merely to a piece of code but to package whose existence implements privileges granted by the coordinator's superior.

In the above approach, there is nothing unique about the status of a coordinator. Any program may create subprocesses for which it carries out coordinator functions according to any queuing logic or discipline it may choose. Two instructions are to be provided in our experimental system to assist in this operation; 'ENTER SUBPROCESS' which effects the complete change of protection context required by making current a new process capability segment and new indirection tables—the new capability segment being defined by reference to the old—and 'ENTER COORDINATOR' which reverses this action.

CONCLUSION

The foregoing discussion has attempted to describe the requirements upon a protection system for information in central memory, and to bring out the problems which arise from various approaches. The upshot is an abandonment for the most general protection systems of lock-and-key methods, and the use instead of methods which rely on selection by indirection. It should not be forgotten, however, that if the requirements of a protection system are modest, then a lock-and-key method may well be feasible. An outline was given of a practicable indirection technique for use in more general cases; again it should not be forgotten that others can be devised which may be more suitable in particular cases.

ACKNOWLEDGMENTS

I am much indebted to Professor M. V. Wilkes who has devoted a great deal of effort to improving the text of this paper, as well as many discussions of its content. The ideas have benefited from interaction with numerous colleagues, in particular three research students, J. R. Horton, C. C. Kirkham, and R. D. H. Walker. The work is part of a project supported by the Science Research Council.

REFERENCES

1 M V Wilkes
   Time sharing computer systems
2 R S Fabry
   Preliminary description of a supervisor organised around capabilities
   Quart Prog Report No 18 Section IIA Inst Comp Res
   Univ Chicago 1968