PROJECT MAC

19 December 1975

Computer Systems Research Division

Request for Comments No. 99

A METHODOLOGY FOR THE DESIGN OF STRUCTURED VIRTUAL MEMORY SYSTEMS

Thesis Proposal by P. A. Janson

Thesis Supervisor: M. D. Schroeder

Massachusetts Institute of Technology

Project MAC

Computer Systems Research Division

Cambridge - Massachusetts

Proposal for Thesis Research in
Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Title: A methodology for the design of structured virtual memory mechanisms

Submitted by: Philippe A. Janson Author's Signature:

Date of Submission: 19 December 1975

Expected Date of Completion: May 1976

Brief Statement of the Problem:

Much effort is currently being devoted to producing certifiably secure computer systems. The general methodology to reach that goal consists of decomposing a system into a structured set of modules. Each module should be sufficiently small so it can be proved correct with existing verification techniques. A partial ordering based on functional dependency should exist among the modules so that a structured verification of the entire system can be generated from the verification of the individual modules. No methodology published today has succeeded in decomposing the virtual memory mechanism of a computer system into modules that are sufficiently small to be verified and interact with one another in a way that allows a structured verification of the entire system. A methodology is proposed to achieve that objective. The essence of the methodology consists of generalizing the concept of type extension to exploit it in an area of a system where it has never been applied before. It is suggested that, by using the systematic approach of type extension, the virtual memory mechanism of a system can be regarded as being implemented in terms of abstract objects and can be organized into a structure reflecting that of the abstract objects. The usefulness of the method in designing efficient and well structured virtual memory mechanisms will be illustrated by a case study involving a model of a commercially available, general purpose computer system.

Introduction.

A computing utility provides a community of users with the ability to share access to its resources. If access to the resources is uncontrolled, unauthorized access to the information stored in the computer system can result. The advent of large computing utilities has thus fostered the need for computer systems that are capable of protecting the information they contain.

Much effort is currently being devoted to producing secure computer systems [see for instance Neumann75, Schroeder75]. A system is secure if its implementation implies its formal specifications and if its formal specifications match the security model it is claimed to implement [see for instance Bell73, Walter74].

Proving the correctness of an entire system as a single unit, by verifying that its implementation implies its formal specifications, is impossible because existing formal program verification techniques are not powerful enough. An alternative method to prove the correctness of a system consists of partitioning the system into regions, verifying each region separately (in fact, correctness needs to be proved only for the security kernel, which contains regions critical to security), and then proving that the correctness of the regions implies the correctness of the system. This method requires that each region be sufficiently small to be verified by human auditing or even by automatic techniques. It also requires that the regions be partially ordered by a functional dependency relation, so that a structured proof of correctness of the system can be generated from the individual proofs of correctness of each region.

In many systems, the security critical procedures and data bases are too

large to be core resident at all times: they must use the virtual memory mechanism of the system. In addition, the virtual memory mechanism supports the user objects that the system claims to protect. Therefore, the virtual memory mechanism is critical to security. Its correctness must be proved to produce a secure system. Unfortunately, no design methodology exists that has succeeded in breaking the virtual memory mechanism of a large system into regions that are sufficiently small and well organized to allow verification to take place.

The proposed thesis will develop a methodology for organizing a virtual memory (VM) mechanism into a set of small and partially ordered regions. The methodology is based on the idea of a world of objects that is structured by a concept of type extension. This concept resembles the type extension concept encountered in protection mechanisms. However, it is designed to be used in the specific area of virtual memory design, and not in other areas of system design or in user environments. This constitutes the originality of the proposed design methodology. All objects involved in the VM mechanism are classified into several abstract types. The relation existing between an abstract type and the types implementing it will naturally lead to the design of type managers (regions) that are partially ordered by a dependency relation. The proposed thesis will demonstrate the use of the methodology by applying it to the design of an <u>efficient</u> VM mechanism for an existing general purpose computing utility.

<u>Related work on partitioning systems into regions.</u>

Partitioning a system into regions includes two aspects. The regions must be sufficiently small so they can be verified, and they must relate to one another in such a way that their individual correctness can be used to

prove the correctness of the whole system. This means that some ordering based on functional dependency must exist among the regions so that the correctness of a region can be derived from the correctness of the regions it depends on and the correctness of the system can be implied by that of the user visible regions.

One can roughly distinguish two different -- although not unrelated -- approaches to the problem of partitioning systems into regions: the level of abstraction (LA) approach and the type extension (TE) approach. The LA approach assumes a totally ordered set of regions that are usually based on functional abstractions of the mechanisms encountered in the system. The TE approach is based on data abstractions of the mechanisms encountered in the system. It presumes partially ordered regions but does not preclude a total ordering. The LA and TE approaches are in fact compatible. They can be and usually are used complementarily within one system.

The LA approach was first encountered in the THE system [Dijkstra68], then in the Venus system [Liskov72]. It was also used in an operating system for a PDP-11/45 [Schiller73], in a VM system [Price73], in the CAL system [Lampson75] and more recently in three structured system designs [Saxena75, Neumann75, Biba75]. Each system is organized into a set of regions called LAs that are totally ordered by a linear dependency relation. Processor management, memory management and device management are examples of mechanisms that usually constitute LAs. The LA approach has been used successfully in decomposing the high level primitives of a system into regions. However, the current literature does not propose any design that decomposes a VM mechanism into LAs corresponding to regions of sufficiently small size to allow verification. In all systems mentioned above, the VM mechanism (or memory

management function) invariably constitutes one or at most two LAs. When the VM is just a paged memory [Saxena75], or a segmented memory [Schiller73], the corresponding LA may be amenable to verification. But if the VM is composed of paged segments [Neumann75, Biba75], verifying segment, page, core and device management within one or even two LAs becomes an impossible task. In fact, in the SRI system [Neumann75] and in Price's system [Price73], the use of two LAs instead of one does not help at all because many functions (e.g. paging) are duplicated as part of each LA. Thus, neither LA is substantially smaller than an equivalent single LA would be. In CAL [Lampson75], the use of two LAs helps towards reducing the size of each LA but is not yet sufficient to make their verification possible and their understanding trivial, as each LA still seems too large.

A formal TE approach was first suggested for the Hydra system [Jones73]. Somewhat different approaches are embodied in the CLU language [Liskov74], in the CAL system [Lampson75] and in the SRI system [Neumann75]. Under the TE approach, data items such as directories or messages are regarded as abstract type objects and are implemented in terms of more primitive type objects like segments or pages. All objects of a certain type (e.g., directories) are maintained by the manager for that type (e.g., the directory manager). The type managers are the regions of the system. Since each abstract type object is implemented in terms of objects of more primitive types (therefore the term "type extension"), a partial ordering of the regions follows from the dependency between the type managers. This structure among the regions will be called an object based structure as it results from the underlying existence of abstract objects.

The TE approach to partitioning has two advantages over the LA approach.

First, in the TE approach, all the attributes of an object are defined and manipulated within the appropriate type manager, whereas in the LA approach, different levels may be responsible for different attributes of the same entity [Habermann75] or, more generally, it may not be clear what the entities are. Thus, the object based interface of a type manager tends to define abstractions more precisely and more completely than the functional interface of a LA. The user of an abstraction will tend to write programs that are cleaner and simpler if the abstraction is defined by an object based interface than if it is defined by a functional interface. Second, LAs assume a total ordering of regions that is unnecessarily constraining, whereas the TE approach allows a more natural partial ordering of regions.

However, the TE approach is more difficult to apply to systems design for two reasons. First, not every concept in a system can be cast into an object based structure. Some of the mechanisms composing a system have an intrinsic operational aspect, which may not be modeled properly by an object based structure. For instance, a dynamic linking primitive does not "manage" (completely and exclusively control) linkage nor any other kind of abstract objects; neither do code conversion, character comparison or bit string manipulation routines. Second, TE itself is a sophisticated feature that, under its current form, requires the support of a VM mechanism, so that the VM mechanism and lower level mechanisms cannot use TE.

The Hydra kernel [Wulf74] has been designed to support operating systems based on the TE approach. The TE approach has also been used in the CAL system [Lampson75] and in the SRI system [Neumann75] in conjunction with the LA approach, as mentioned earlier. In the Hydra and SRI systems, the higher level primitives of the system fit well into an object based structure.

However, the formal notions of abstract objects and extended types are all but absent from the implementation of the lower level primitives of the systems, where a functional approach similar to LAs was taken. In both systems, the most primitive type of object is precisely the type of information container maintained by the VM mechanism (pages or segments). However, the implementation of these primitive types of objects depends on functional abstractions like resource allocation and paging, rather than on more primitive data abstractions like core blocks and disk records for instance. Thus, even though the VM mechanism implements data abstractions, the advantages of TE are lost in the mechanism itself because those data abstractions are not themselves implemented in terms of more primitive data abstractions. (The CAL system is somewhat more successful in this respect as it follows more closely the formalism of type extension.) In all three of the Hydra, CAL and SRI systems, the partitioning techniques used for the VM mechanism have not succeeded in breaking it into regions of a size sufficiently small to permit verification. The Hydra kernel [Wulf74], which supports pages as a primitive type is not partitioned at all. In the CAL system, the VM mechanism is implemented by two LAs each of which supports a different type of file. Unfortunately, each LA supports many other types of data abstraction in addition to files. As a result, the size of each LA seems beyond the capabilities of existing verification techniques. In the SRI system [Neumann74, Robinson75, Neumann75], the software for the VM mechanism is composed of two Parnas-like abstract machines [Parnas72a, 72b]. The higher level machine provides user segments and the lower level machine provides a fixed number of fixed size, permanently active system segments. The abstract machines cannot properly be called type managers as they implement much more

than just segments. Their internal operation, which is precisely the complex part of any VM mechanism, is an agglomerate of resource and I/O management functions. These functions do not correspond to operations on any formal abstract object. The design of the two VM abstract machines has failed to recognize paging as a mechanism they share, which could have been isolated in a separate LA. Instead, the paging mechanism is duplicated as part of each abstract machine, as mentioned earlier. Consequently, the two abstract machines are implemented by programs that seem too large and too complex to be verified with existing techniques.

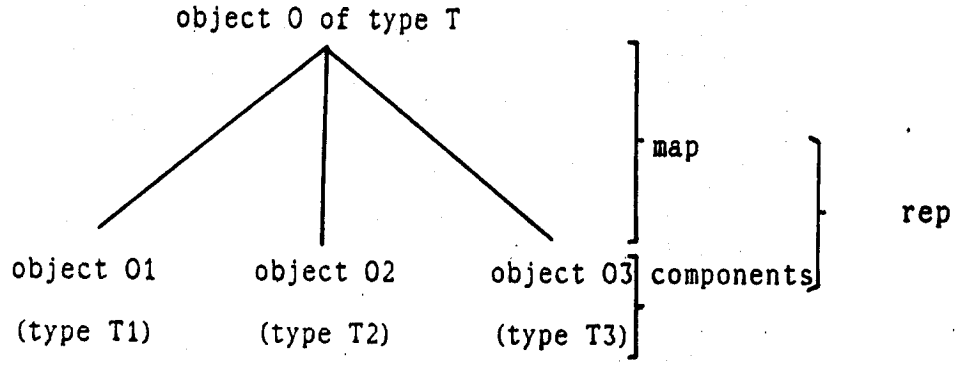
The methodology to be proposed for designing an efficient and well structured VM mechanism is based on a generalized TE concept that preserves the advantages of the traditional concept used in Hydra, in CLU, in CAL and in the SRI system, but makes it applicable to VM mechanisms.

A generalized type extension concept.

To understand how the generalized TE concept and the traditional one differ and how they may complement each other in a system, we will define a model of an abstract object and point out the differences and the connections between the two concepts on the elements of the model.

Every abstract type is defined by a set of operations that can be performed on all objects of that type by the manager for that type. Every abstract object is named by a unique identifier (uid). The uid is unique over all objects of all types and over all times: the uid of a destroyed object is never reused. The uid is the concatenation of a type identifier (tid) and an object identifier (oid). The tid is unique over all types and the oid is unique within each type to allow type dependant interpretation. Every abstract type is implemented in terms of more primitive types. Thus, an

object O of type T is implemented by a set of components, which are objects O1, ..., On of more primitive types T1, ..., Tn respectively. The correspondence between an object and its components is called the map of the object. The map together with the components of an object compose the representation (rep) of the object.



With the above model of an abstract object, we are in a position to point out the differences between our generalized TE concept and the traditional one.

The first difference has to do with the primitive types of objects we will be manipulating. The traditional TE concept has always implicitly assumed that the most primitive types, which we will further call base level types, are the virtual information containers supported by the VM mechanism (pages or segments). However primitive these types may be, they are still "extended". Pages and segments are logical entities but have no intrinsic hardware connotation. Instead, the most primitive types we will be considering, which we call bottom level types, are core blocks and disk records, which are information containers with a direct hardware representation. Segments and pages are implemented in terms of in-core and out-of-core copies of information. Core blocks and disk records are the most primitive objects one could conceive: they are implemented by themselves. The

fundamental difference between base level and bottom level objects is their life time. Base level objects are created and deleted [Liskov74]: we further refer to them as C/D objects. Most of the objects manipulated by a VM mechanism (those of types between bottom and base levels) are never created or deleted (except for occasions when hardware modules are reconfigured). They have a permanent existence and are only allocated and freed: therefore their name of A/F objects.

The second difference between the two TE concepts is a direct consequence of the distinction between C/D objects and A/F objects. In the traditional TE concept, the uid of an object is meaningful to a user of the object between creation and destruction of the object. This time span corresponds to the life time of the object. The uid loses all meaning after the object is destroyed. In the generalized TE concept, the uid of an A/F object retains its meaning for a user of the object between allocation and deallocation of the object. But this time span does not correspond to the life time of the object. After the object is deallocated, the uid looses its meaning for the users but keeps it for the type manager. The id of a core block is its absolute address, which may cease to be used temporarily, while the object is free, but continues nevertheless to exist.

The third and most fundamental feature of the generalized TE concept deals with protection. The generalized TE concept does not afford to A/F objects the protection that the traditional TE concept affords to C/D objects. Protection of C/D objects is based, for instance, on sealing their uids inside capabilities. The use of capabilities implies the maintenance of a system-wide table of currently valid capabilities. Since there is in general a very large number of objects for which valid capabilities exist, such a

table is too large to be core resident at all times: it requires the support of the VM mechanism. But the purpose of A/F objects is precisely to implement the VM mechanism. Thus, they cannot at the same time depend on it to implement the system-wide table to hold capabilities for them. Consequently, the use of capabilities for A/F objects must be rejected as impractical. However, this does not mean that A/F objects are unprotected. The protection of A/F objects is based on two constraints that are enforced outside of the formalism of the TE concept. First, only those regions of the system implementing the VM mechanism are allowed to use A/F objects. Second, those regions must be proved to "conserve" the uids of the A/F objects they use: they must be proved to use only uids that were passed to them by higher level callers or returned to them as the result of downward calls (requests for allocation for instance), and they must be proved to destroy or cause destruction of all instances of uids (except the originals owned by type managers) of objects they return to free pools.

The security of an A/F object is thus dependent on the trustworthiness of the regions using it, while the correctness of these regions is dependent on that of the A/F type manager. This situation creates a dependency loop between the A/F type manager and its users. In fact, this is a "benign" dependency loop. The true reason for avoiding mutual dependency between two regions is to make the design of the system more understandable and its correctness easier to prove. In the present case, the correctness of the region using the A/F object <u>does</u> depend on the correctness of the A/F type manager but the correctness of the type manager depends only on the proper conservation of the uid of the object. The proposed thesis will show that the conservation property does <u>not</u> depend recursively on the semantic correctness

of the type manager. Therefore the loop is benign in that it can easily be broken conceptually and complicates neither the understanding nor the verification of the system. Thus, while the protection property of the traditional TE concept is not preserved for A/F objects, a different kind of protection is provided by the certification of the VM mechanism. More importantly, the structural property and the resulting advantages in designing an object based system are preserved by the generalized TE concept.

The fourth generalization of the TE concept results from the essence of operations on A/F objects. The essence of an operation on a C/D object consists of manipulating its components. Manipulating its map is a trivial task. In fact, in most implementations, a C/D type manager does not even bother to implement the maps for objects of its type. A central mechanism creates and protects the maps for all abstract type objects under the form of many capability lists [Jones73] or of a huge segment containing all the maps [Neumann75]. On the contrary, the essence of an operation on an A/F object consists of manipulating the map and not so much of manipulating the components. The reps of A/F objects consist essentially of maps that allow the hardware to go, for instance, from a segment to its page table, from the page table to a page and from a page to a core block or a disk record to ultimately perform an efficient "read" or "write" on a bottom level component. Not only is the map the crucial part of the rep of an A/F object, but A/F type managers cannot depend on any central mechanism to implement maps: the traditional TE concept uses abstract information containers to implement maps, which is just what the VM mechanism is trying to implement. Instead, each A/F type manager must implement the maps for the objects it manages out of whatever primitive information containers are available to it.

The proposed design methodology.

Given the TE concept defined above, the methodology consists of defining some suitable set of abstract types to ultimately implement base level objects in terms of bottom level objects. The choice of the abstract types is a tradeoff between efficiency of operation and ease of verification. If many abstract types are defined between bottom and base levels, the regions implementing the type managers are likely to be small and thus easy to verify but the implementation is likely to be inefficient because of all the maps that will be established between a segment and the core blocks that ultimately implement it. Once the abstract types are defined, the dependency graph between the regions of the VM mechanism can easily be drawn. For a given abstract type, the type manager will depend on all type managers implementing the components of the objects of the given type, and on the type manager implementing the information containers used to store the maps of the objects of the given type.

A case study.

In order to demonstrate the use of the methodology, the proposed thesis will apply it to a case study. This study involves a model of the Multics system [Organick72, Multics74]. The model exhibits the general functionality of Multics as perceived by its users: users see a segmented VM of which each word can be addressed by segment number and offset; thanks to a demand paging mechanism, users are unaware of the pages of a segment; user segments are catalogued in a hierarchical file system, protected by access control lists and periodically saved by a backup mechanism. However, the model is an abstraction of Multics in that none of the internal interfaces of Multics are preserved. The functionality of most mechanisms not directly accessible to

the users is different from the original Multics and is abstracted as much as is possible without oversimplifying the problem of designing the VM mechanism. The Multics system has been chosen as the basis for the model for three reasons.

First, it is a commercially available and viable system. Thus, by applying the methodology to the redesign of its VM mechanism, we will avoid the pitfall of triviality that would result from using a toy system as a case study.

Second, it is a large, powerful and sophisticated system of which the VM mechanism is efficient but unamenable to any kind of proof due to its size and complexity. While applying our design methodology to the case study, we will preserve the functionality of the Multics VM mechanism as perceived by a user, but we will completely reorganize its internal hardware and software structure to simplify it.

Finally, we have chosen Multics because it is available, because we are familiar with it and because the proposed thesis fits in the framework of an ongoing research project aimed at restructuring the security kernel of Multics [Schroeder75].

The design to be proposed will consist of the specifications for all the abstract machines [Parnas72a, 72b] of the system kernel (in the Hydra sense of the word as opposed to the security kernel). The system kernel will comprise the VM mechanism and a virtual processor management mechanism that the VM mechanism depends on. The virtual processor management mechanism is not an original part of the proposed thesis. Such a mechanism is currently being designed at the Computer Systems Research Division of Project MAC along the lines of an object based structure. Its specifications will simply be

included in the proposed thesis for completeness. The VM mechanism will not include any policy for user resource control but it will contain mechanisms for physical resource control (to avoid overflowing physically available space). It will assume it is supporting Multics-like access control, user resource control and backup functions but these functions will not be included as parts of the system kernel. They will only have some impact on its design.

The efficiency of the new design of the VM mechanism will be rated by showing that the number and the nature of the abstract object maps encountered on the access path from a segment to its bottom level components match the number and the nature of the data structures encountered on the equivalent path in the current Multics implementation.

Conclusion.

The main purpose of the proposed thesis is to present a design methodology for producing a structured VM mechanism for a certifiably secure system, by extending the systematic approach of type extension to an area of system design to which it was not originally thought to be applicable. A second goal of the thesis is to demonstrate the applicability of the methodology on a case study. One result of the proposed thesis will be to demonstrate that a VM mechanism can be designed in a structured fashion and still be efficient.

The design methodology is based on a generalized TE mechanism. It consists of casting the concepts manipulated by the VM mechanism into an object based structure and then deriving the dependency graph of the type managers involved in the operation of the VM mechanism from the object based structure defined previously.

The methodology will be demonstrated by redesigning the VM mechanism for

Multics, a commercially available, general purpose computer system. Specifications will be given for the different type managers involved in the VM mechanism. The dependency graph will be shown to be loop-free. The efficiency of the new design will be shown to be comparable to that of the current design.

References.

[Bell73] D. E. Bell, L. J. LaPadula, "Secure computer systems", ESD-TR-73-278, Mitre Corp. (Nov. 1973).

[Biba75] K. J. Biba, "Kernel levels of abstraction", Multics Design Note 13, Mitre Corp. (Feb. 1975).

[Dijkstra68] E. W. Dijkstra, "The structure of the THE multiprogramming system", CACM 11, 5, p 341-346 (May 1968).

[Habermann75] A. N. Habermann, L. Flon, L. Cooprider, "Modularization and hierarchy in a family of operating systems", to appear in CACM 19 3 (Mar. 1976).

[Jones73] A. K. Jones, "Protection in programmed systems", Ph. D. Th., Dept of Comp. Sc., CMU (Jun. 1973).

[Lampson75] B. W. Lampson, H. E. Sturgis, "Reflections on an operating system design", to appear in CACM 19 3 (Mar. 1976).

[Liskov72] B. H. Liskov, "The design of the Venus operating system", CACM 15, 3, p 144-149 (Mar. 1972).

[Liskov74] B. H. Liskov, "A note on CLU", CSG Memo 112, MIT Project MAC (Nov. 1974).

[Neumann74] P. G. Neumann, et al., "On the design of a provably secure operating system", Proc. Workshop on Protection in Operating Systems, IRIA, p 161-175 (Aug. 1974).

[Neumann75] P. G. Neumann, et al., "A provably secure operating system", SRI Final Report (Jun. 1975).

[Organick72] E. I. Organick, "The Multics system: An examination of its structure", MIT Press (1972).

[Parnas72a] D. L. Parnas, "A technique for software module specification with examples", CACM 15, 5, p 330-336 (May 1972).

[Parnas72b] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", CACM 15, 12, p 1053-1058 (Dec. 1972).

[Price73] W.R.Price, "Implications of a virtual memory mechanism for implementing protection in a family of operating systems", Ph.D.Th., Dept. of Comp. Sc., CMU (Jun.1973).

[Robinson75] L.Robinson, et al., "On attaining reliable software for a secure operating system", Proc. Intl. Conf. on Reliable Software (Apr.1975).

[Saxena75] A.R.Saxena, T.H.Bredt, "A structured specification of a hierarchical operating system", Proc. Intl. Conf. on Reliable Software, p 310-318 (Apr.1975).

[Schiller73] W.L.Schiller, "Design of a security kernel for the PDP-11/45", ESD-TR-73-294, Mitre Corp. (Dec.1973).

[Schroeder75] M.D.Schroeder, "Engineering a security kernel for Multics", ACM Fifth Symp. on Operating Systems Principles (Nov.1975).

[Walter74] K.G.Walter, et al., "Primitive models for computer security", Dept. of Comp. and Info. Sc., Case Western Reserve Univ. (Jan.1974).

[Wulf74] W.Wulf, et al., "Hydra: the kernel of a multiprocessor operating system", CACM 17, 6, p 337-345 (Jun.1974).

[Multics74] ---, "Introduction to Multics", MAC-TR-123, MIT Project MAC (Feb.1974).