Using Type-Extension to Organize Virtual-Memory Mechanisms [†]

P.A. Janson

IBM Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Much effort is currently being devoted to producing computer systems that are easy to understand, to verify and to develop. The general methodology for designing such a system consists of decomposing it into a structured set of modules so that the modules can be understood, verified and developed individually, and so that the understanding/verification of

[†] This research was performed in the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science. It was sponsored in part by Honeywell Information Systems Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defence under ARPA Order No. 2641, which was monitored by ISTAO under contract No. F19628-74-C-0193.

Support from the IBM Zurich Research Laboratory during the preparation of this paper is gratefully acknowledged.

June 25, 1981

the system can be derived from the understanding/verification of its modules. While many of the mechanisms in a computer system have been decomposed successfully into a structured set of modules, no technique has been proposed to organize the virtual memory mechanism of a system in such a way.

The present paper proposes using type extension for that purpose. The virtaul memory mechanism consists of a set of type manager modules implementing abstract information containers. The structure of the mechanism reflects the structure of the containers that are implemented. While using type extension to organize a virtual memory mechanism is conceptually simple, it is hard to achieve in practice. All existing or proposed uses of type extension assume the existence of information containers that are uniformly accessible, can always be grown and are protected. Using type extension inside a virtual memory mechanism raises implementation problems since such containers are not implemented. Their implementation is precisely the objective of the virtual memory mechanism. In addition to explaining how type extension can be supported inside a virtual memory mechanism, the paper briefly discusses some aspects of its application to the reorganization of the kernal of a commercial, general-purpose, time-sharing system. It concludes by presenting some results of that case study concerning the organization of operating systems.

1 Introduction

This paper is concerned with a technique for organizing the virtual memory mechanism of a computer system. The technique is not aimed at improving the functionality of virtual memory mechanisms but rather concentrates on simplifying the procedures and the data structures necessary to support such mechanisms. Although the technique was elaborated in the context of and applied to the design of the virtual memory mechanism of a real, general-purpose, time-sharing system, we believe it is applicable to any kind of computer facility, real-time system, distributed filing system or conventional time-sharing system.

Our interest in techniques for organizing virtual memory mechanisms is motivated by the size and the complexity of existing operating systems for large computing facilities. Operating systems and the various subsystems that compose them are extremely hard to understand. They are still harder to maintain through several development stages. Their correctness cannot be verified. There is a definite need to simplify such systems: understanding their operation should be easy to derive from reading their code; minor system changes should be easy to implement locally without requiring a global understanding and without altering the overall integrity of the system; correctness should be easy to verify by a methodical inspection of the system programs.

Many researchers have attacked the problem of organizing operating systems. The general approach for doing so consists of decomposing the system

concerned into a set of subsystems so that it becomes possible to analyze the properties of each subsystem independently and to relate them in some way to derive the properties of the whole system. In short, the approach consists of breaking-up a large problem into related smaller ones. This approach has proved very successful in general. It has been used to decompose the THE system [1] into levels of abstraction. It has been used to decompose the Venus system [2] into abstract machines. More recently, it was used by the Mitre Corporation to organize security kernels for a PDP-11/45 system [3] and for a Multics system [4]. It was also used in four systems that will be mentioned again later: a virtual memory system designed at Carnegie-Mellon University [5], a hierarchical system designed at Stanford University [6], the CAL/TSS system [7] and the SRI system [8]

Unfortunately, and in spite of what the reader might expect from the abundant literature cited above, no satisfactory solution has ever been proposed to organize a virtual memory mechanism. In all the systems mentioned, the virtual memory mechanism has been isolated from the rest of the system. In some cases, it has even been itself decomposed into several modules. However, the decomposition process has never gone far enough and has not been performed according to any specific method. Consequently, the resulting virtual memory mechanisms remain hard to understand, to maintain and to verify. In the CMU system [5] and in the Stanford system [6], the virtual memory mechanism is implemented by two modules, called levels of abstraction. The existence of two modules clarifies the overall organi-

zation of the system but it does not simplify the internal organization of the virtual memory mechanism: instead of each module being half the size of what a single module would be, the modules duplicate one another almost entirely. In the CMU system, they implement fixed size and variable size virtual memory segments. In the Stanford system, they implement virtual memory for a fixed number of system processes and for a variable number of user processes. In the Cal and SRI systems, which also implement their virtual memory mechanism with two modules, called abstract machines, the two modules do not duplicate each other. Even so, however, the mechanism implemented by any individual module is too large and too complex to be easily understood, implemented or verified.

The objective of the research reported here was precisely to remedy this situation by proposing a technique capable of helping the system designer in decomposing his virtual memory mechanism into modules that are sufficiently small and properly chosen to yield a simple system organization. The technique to be described is based on the concept of type extension. This concept was first introduced in programming languages (e.g., SIMULA) where it has become very popular [9]. It was then envisioned in the operating system area by the designers of Cal [10] and Hydra [11]. It was actually used, more or less formally, in Cal [7], in Hydra [12] and in the SRI system [8]. However, in the operating system area, it was used formally and successfully only for high-level primitives, not inside primitives as low as virtual memory mechanisms. The reasons for the ini-

tial failure of the concept at the lowest level of an operating system will become clear in Section 3 of this paper.

In the following section, the paper will review some background concepts about the organization of software systems in general and about the concepts of type extension in particular. In the third section, the paper will show how type extension can be used to organize a system. It will also explain the difficulties of using the concept at the lower levels of an operating system. It will finally describe a solution to overcome these difficulties, and apply the concept to organize a virtual memory mechanism. In the fourth section, the paper will briefly discuss how the concept was used to develop a prototype system by reorganizing the kernel of a real time-sharing operating system, namely Multics, which is commercially available from Honeywell Information Systems Inc. The benefits of type extension will be assessed. In the last section, the paper will conclude by a tentative evaluation of the applicability of the concept to other operating system mechanisms.

2. Background

2.1 The Organization of Operating Systems

As briefly stated in the Introduction, the basic means for organizing a software system consists of decomposing it into subsystems, which we shall further call *modules*, in such a way that it is possible to analyze

modules independently of one another while also allowing the derivation of properties of the whole system from the properties of its modules. Thus, it is desirable that a module be clearly distinguished from others, and be sufficiently small to allow one person to understand, develop or verify it separately. It is also desirable that modules be connected to one another by some well-defined *structuring relation* that allows inferring the properties of the system from those of its modules. We now consider somewhat in detail what is meant by a module. We shall then consider what is meant by a structuring relation.

Two concepts of a module can be found in the literature. The first one, which we call a *strict module*, is discussed in Parnas [13] and Liskov [14]. A strict module is a collection of procedures and data structures that is totally isolated from other modules, i.e., every procedure or data structure belongs statically in one and only one module. Notice that strict modularity does not preclude the temporary sharing of arguments to inter-module calls. The second type of module, which we call a *weak module*, is discussed and used in Parnas [15] and Habermann [16]. A weak module is a collection of procedures and data structures of which the data structures may be statically part of (shared by) several modules at a time. In our research, we have used the concept of a strict module as it is much cleaner and simpler to deal with. Indeed, since data structures are never shared by two or more modules, they are never part of any module interface. Thus, module interfaces never contain any information about the implementation

of data structures. The designers of two different modules never have to agree on the format or the management of a data structure (argument list excepted), in particular, the synchronization of concurrent activity involving a common data structure. The information hiding principle [17] is respected, with all the resulting benefits.

Note that a module is always defined by the procedures it contains and not by the process(es) executing it. One or more concurrent processes might be executing the code of a given module at a given time.

Two structuring relations have been proposed to characterize the organization of modules in a system: the usage relation and the dependency relation. The *usage relation* was proposed by Parnas [15]. In fact, weak modules have been proposed in conjunction with the usage relation. A module A is said to use a module B if B performs a service for A, i.e., if the execution of B can be triggered by actions (calls or messages) of A and A expects results from the computation performed by B.

It is claimed that the usage relation suffices to infer the properties of a system from those of its modules if that system can be represented as a set of modules hierarchically structured by the usage relation. To derive the properties of the system, it is sufficient to infer the properties of the top-level modules of the usage hierarchy from the properties of the modules they use. While this would be true for strict modules, it is not so for weak ones. For instance, if A and B do not "use" each other but share a data structure, one cannot verify the correctness/understanding the

behavior of A unless the management of the shared data structure is fully understood, which requires verifying the correctness / understanding the behavior of B.

The *dependency relation*, suggested by Feiertag in unpublished work he did at the M.I.T., is more complete. It states that a module A depends on a module B if the correctness of A cannot be verified without verifying the correctness of B, i.e., if A makes any assumption about the operation of B. Module A can be said to make assumptions about module B in three cases: if it transfers control to B and expects B to return control and potential results after it has completed its computation; if it sends a message to (a process executing) B and expects to receive a reply message with potential results; if it shares a data structure with B and expects B not to affect the integrity of that data structure.[†] The dependency relation is more complete in that it takes into account interactions between weak modules over shared data bases. While we prefer the dependency to the usage relation in general, we can consider them interchangeably in this paper since we have chosen to work with strict modules, which rules out dependencies owing to shared data bases in the first place.

[†]Note that if A transfers control or sends a message to B and neither expects to regain control, nor counts on observing consequences or results from its interation with B, then A does not depend on B. It is said simply to notify B of an event (without caring about what B does once it is notified).

In summary, our goal is to propose a technique for decomposing any virtual memory mechanism into a set of strict modules that is hierarchically structured by the dependency relation. This should enable the designers of a system to study modules independently of one another while allowing them to derive the properties of the system by using its hierarchical dependency-based structure.

2.2 The Concept of Type Extension

In the foregoing, we have summarized background concepts concerning the organization of systems. In the following, we shall review the concept of type extension, which will enable us, in the next section, to see how it can be used to organize a system as proposed above.

Type extension is a concept borrowed from the programming language field. It is based on the idea of abstract data types, e.g., integers, reals, arrays, stacks, etc. An abstract data type is defined by a set of operations that can be applied to any object of that type by a program called a type manager for that type. The first property of abstract data types is that their users need not know about their implementation. In other words, if a user wants to manipulate an array, he does not need to know how arrays are stored and managed. He needs only to call the array manager and request it to manipulate the array in the desired fashion. The second property of abstract data types is that their users cannot access their implementation. Even if a user knows how arrays are imple-

mented, he cannot access them: only the array manager is allowed to access them. Arrays are stored in internal data structures of their manager and are hidden and protected from outside programs.

The concept of type extension, then, comes from the idea of building abstract data types on top of one another by defining the operations on a higher-level type in terms of operations on lower-level types. For instance, a complex number can be defined as a pair of reals and all operations on a complex number are defined in terms of operations on the two reals (REAL and IMAG or MAGNITUDE and PHASE) that implement it.

This is formalized in Figure 1. The abstract type objects, in terms of which a high-level type object 0 is defined, are called the *components*

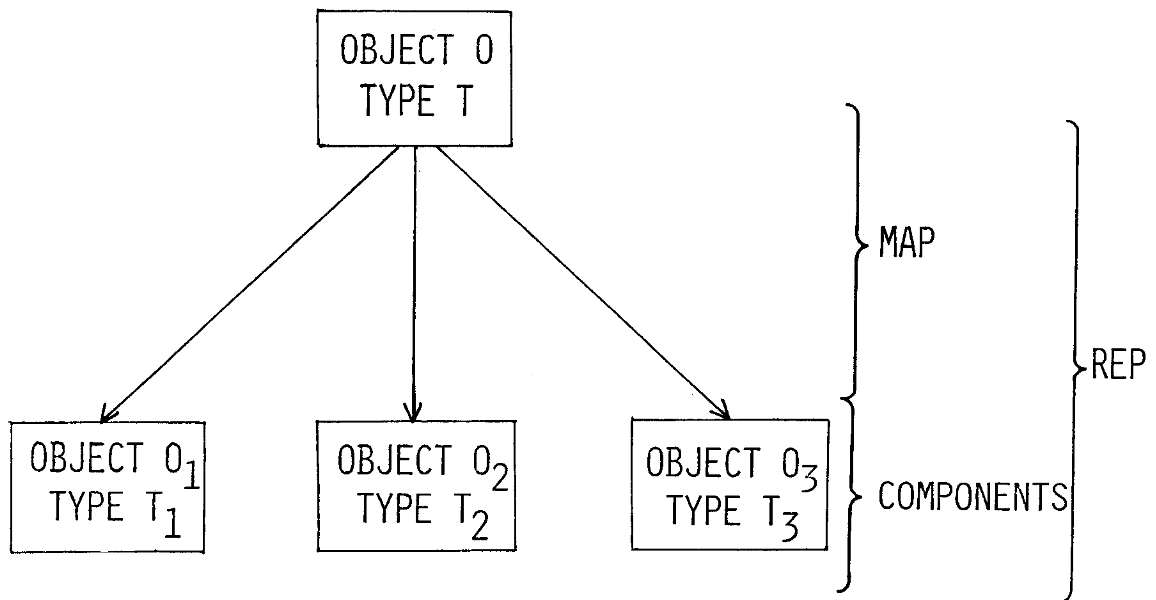


Fig. 1. Definition of an abstract type.

of 0. The relation that binds the identity of 0 to the identities of its components is called the *map* of 0. The map of 0 together with its components is called the *representation* (rep.) of 0.

In the programming language area and later in the operating system field, type extension has been used to relieve the users of an abstract type object (e.g., an array of integers in a programming language, or an abstract data structure like a file directory in an operating system) from coping with the intricacies of the implementation of that object, while, at the same time, protecting the representation of the object from accidental damage by careless or curious users. In a virtual memory mechanism, type extension will be used not so much to provide simple interfaces to deal with complex abstractions, but rather to force the designers to write simple and well-organized software to support the complex mechanisms embedded in the virtual memory mechanism.

3 Type Extension as an Organizational Technique

The previous section has summarized what it means to organize a system and what it means to use type extension, at least in the programming language field. The purpose of this section is to explain what it means to use type extension to organize a virtual memory mechanism.

The key idea consists of applying the concept of type extension not to abstract data types (integers, reals, etc.) but rather to abstract informa-

tion containers such as one might encounter in a virtual memory mechanism: disk records, core blocks, pages, segments, files, catalogs, volumes, etc. One must conceive and design the virtual memory mechanism as a set of type managers implementing abstract information containers in terms of more primitive information containers.

If a virtual memory mechanism can be thought of as suggested above, it is guaranteed to be strictly modular. Indeed, every type manager with its procedures and data structures constitutes a module. And, since its data structures are pertinent to the representation of objects it hides and protects internally, a type manager is a strict module.

Furthermore, type managers are related by dependencies. Indeed, if segments, for instance, are implemented as collections of pages, operations on segments are implemented in terms of operations on pages. Thus, if the segment manager is invoked to operate on a segment, it will turn around and invoke the page manager to operate on the page components of the seg-

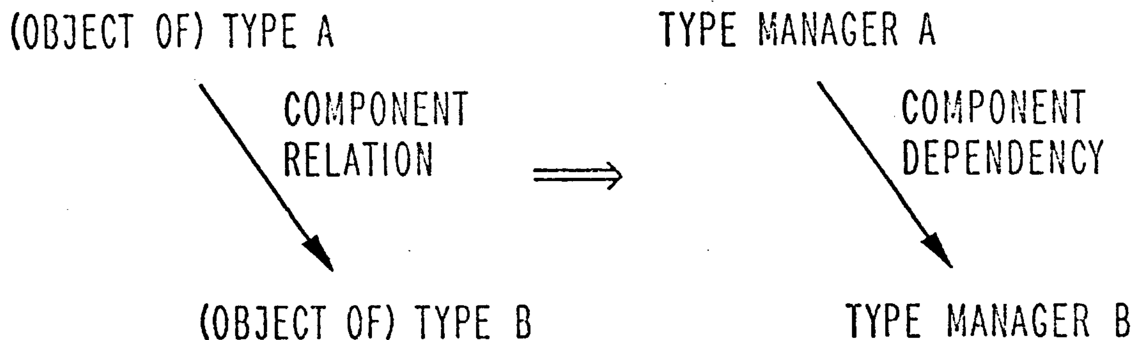


Fig. 2. Derivation of component dependencies.

ment. This means that the segment manager depends on the services provided by the page manager. If the structure of every abstract information container is drawn as in Figure 1, and the resulting figures are all connected together, one obtains the *structure graph* of the virtual memory mechanism. For every arc in that graph, there is a corresponding dependency between the type manager implementing the origin type of the arc and the type manager implementing the target type of the arc. This dependency is called a *component dependency* (Fig. 2).

The graph of all component dependencies is hierarchically structured by the dependency relation. This fact is the basic argument for trying to use type extension as an organizational technique in the operating system field. It was exploited by the designers of the Cal, Hydra and SRI systems to cast some structure into their respective systems. However, in all cases the technique was not used below the level of the interface of the virtual memory mechanism. The concept of a directory, a catalog or a name space is implemented as a true abstract type in all three systems. However, lower level concepts (segments, files, pages, records, blocks, etc.) are not.

The question is: why not? The answer is: because using type extension at and below the virtual memory level is not trivial, as explained below.

All component dependencies are said to be *explicit* dependencies in that they can be explicitly derived from the structure graph of the system.

However, component dependencies are not the only ones in a system. They result from the implementation of the objects *supported* by the virtual memory mechanism, namely, the abstract information containers. They do not take into account the implementation of the objects *supporting* the virtual memory mechanism itself, namely, the type managers. The implementation of type managers gives rise to dependencies that are said to be *implicit*. Explicit and implicit dependencies constitute the *dependency graph* of the system. Implicit dependencies exist above and below the virtual memory level of a system. However, it will become clear, after we discuss their nature, that they do not pose any structural problem above the virtual memory level. They can eventually be ignored above that level. The designer of the Cal, Hydra and SRI systems did not even know about their existence. Yet, they are the major source of trouble in implementing type extension below the virtual memory level.

Implicit dependencies fall into four categories: program storage dependencies, data dependencies, memory dependencies and processor dependencies. At least one dependency of each kind is necessary to implement any type manager. The procedures of a type manager must be stored in some type of information container. By extracting executable code from such a container, a type manager makes itself dependent on the supplier of the container: [†]

[†] Indeed, executing code and writing/reading information in a container are implicit (hardware) forms of calling the type manager for that container: if a segment manager does not properly allocate space for its segments, attempts to access segments that are improperly supported will result in errors.

this is a *program storage* dependency. The data structures of a type manager, in particular the maps of the objects it implements, must also be stored in some type of information container. By writing into/reading from such a container, a type manager makes its correct operation dependent on that of the supplier of the container: this is a *data* dependency. A type manager must be able to name its procedures and data structures to reference them. In other words, the addresses of its procedures and data structures must be described in some memory space seen by the manager. This memory space, i.e., the list of addresses, capabilities or descriptors for the type manager must also be stored in some container. Thus, the type manager is also dependent on the supplier of that container: this is a *memory* dependency. Finally, the code of a type manager must be interpreted by some kind of processor. If that processor does not properly carry on execution, alters sequencing of instructions, mishandles branching, stops running or somehow misinterprets the control structure of a program, the type manager will not operate correctly. Thus, the type manager depends on the processor. In general, a type manager cannot have a physical processor all to itself. What it sees is some sort of a virtual processor or process resulting from the multiplexing of a physical processor among several type managers. The physical processor together with the mechanism that multiplexes it implements a collection of abstract processors. An abstract processor is an abstract object composed of at least a processor state and sometimes a processor that causes the state to evolve. Thus, every type manager depends on a module

supplying abstract processors: this is a *processor* dependency.

In a user environment, implicit dependencies pose no problem. A catalog, directory or name space manager is implicitly dependent on several type managers supplying information containers (e.g., segments, files, pages, capability lists) and abstract processors. However, these type managers are at a level below the catalog, directory or name space managers, inside the virtual memory mechanism. Thus, the resulting implicit dependencies are certainly "downward" in that they do not violate the partial order of type managers defined by the explicit dependencies.

Inside the virtual memory mechanism (and the virtual processor mechanism, for that matter), the situation is not so clear cut because the purpose of these mechanisms is precisely to implement abstract information containers and code interpreters. In order to implement the procedures, data structures, memory and processor required to support his type manager, the designer must be careful to utilize only abstractions defined at lower levels as dictated by the graph of explicit dependencies. If no lower-level abstraction is suitable, a special-purpose abstraction must be designed. If a designer were careless, he could accidentally implement the programs of a type manager with containers defined in terms of objects supported by a higher-level type manager. This would result in an "upward" dependency, which would violate the partial order of the desired hierarchically structured dependency graph. Back to the earlier example of segments composed of pages, the procedures and data structures of the page manager may not be implemented by segments. (If they were, it would be

necessary to verify that the segment manager never damages those segments implementing the page manager and that the page manager never damages segment pages, in particular, pages of its own segments: a tricky fixed-point problem!).

In summary, exploitation of type extension at or below the virtual memory mechanism of a system demands a lot of care. Special attention must be given to identifying every single implicit dependency and verifying that it does not cause any structural problem.

4 Demonstration of the Technique

The previous section has explained how type extension could be used as an organizational technique. The question that must be answered next is obviously: can type extension be used as suggested, in practice? In other words, is it possible to envision a realistic virtual memory system as implementing a hierarchy of abstract information containers and code interpreters? It is of course impossible to answer this question positively for any system because it is not feasible to prove in any formal way that every system could be envisioned from the type extension viewpoint described.

Rather than proposing vague and hazardous statements about the type-extension technique, the research project reported here concentrated on actually applying the technique to the organization — rather the reor-

ganization — of a real, commercial, general-purpose, time-sharing system. The Multics system [18, 19] was chosen for this experiment for several reasons.

First, redesigning an existing and viable system was deemed more interesting than designing some imaginary system to avoid the pitfalls of generating an unrealistic toy system and diverting the center of the research from type extension to operating system concepts.

Second, Multics is a large, powerful and sophisticated system of which the virtual memory mechanism is indeed a complex maze of code and data. Hence, Multics is as good a candidate as any other system, for, if testing type extension succeeds on Multics, it is likely to succeed anywhere else.

Finally, the present research fitted perfectly within a large project [20] aimed at producing a more understandable version of the whole Multics system.

The redesign of the Multics virtual memory system involved two major steps. In the first, the existing system was studied. Its modules were listed and its structure was analyzed by identifying every inter-module dependency as defined in Section 2. This permitted identification of all the sources of complexity in the system, namely, the weak modules and the violations of the hierarchical structure.

In a second step, a new design was proposed, which was based on type extension, eliminated the modularity and structure problems discovered earlier and provided exactly the same functionality as the original system, at

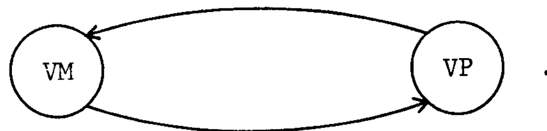
the user interface. The design proposed has not been implemented although an implementation would have been interesting. However, it would have required many more man-years of work than were available. Short of an implementation, the feasibility and practicality of the proposed design were judged by comparing it to the original one. This comparison was made possible by the way in which the new design was produced. Neither the original design, nor the new one can be described here for lack of space. Very detailed information on both designs can be found in [21]. However, this paper will try to show why the two designs could be compared easily and what some of their differences are. This will provide some insight into the use of the type extension technique in a real system.

Type extension is not a mechanical tool for producing a modular and structured system design in a systematic way. Instead, it is an evaluation and guidance tool as well as a design discipline that allows a designer to iteratively rate and tune his design towards a modular and structured one. Applying type extension to the design of a system is a matter of experience, "good taste", and feeling for the concept. There is nothing methodical about type extension.

In the case study involving Multics, the new design was evolved from the original one in several design iteration steps, just as a well-organized design should be derived from a first draft in the general case. After each design iteration, the system was inspected from the perspective of type extension. Every inter-module dependency that appeared to violate the hier-

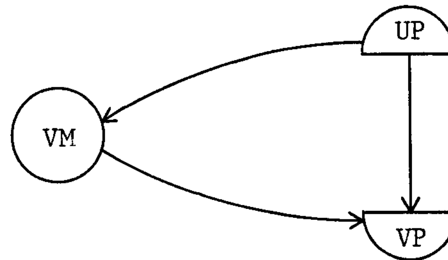
archical structure of the system was identified to a specific (explicit or implicit) dependency in the type extension sense. This always permitted the expression, in terms of data abstractions, of the exact nature of the problem behind what looked like a "nasty" upward dependency in terms of system design. Then, it has always proved possible to add, delete, combine or separate abstractions so as to remove undesired dependencies. However feasible, this task was never trivial by any measure, as suggested by the following examples.

Perhaps the most remarkable example of organizational problems involves both the virtual memory and the virtual processor mechanisms. In the original design, the virtual memory mechanism depends on the virtual processor mechanism to switch a physical processor from a process P to a process P' when P must wait for a virtual I/O operation and P' is ready to run. On the other hand, the virtual processor mechanism multiplexes physical processors among a very large number of processes. There are too many processes to keep an image of the state of each of them in primary memory at all times. Thus, the virtual processor mechanism depends on the virtual memory mechanism to move processor state images in and out of primary memory. This situation yields a dependency loop:



To eliminate the loop, the process abstraction was split into two: virtual

processor and user processor. A virtual processor is a processor state image that resides in primary memory. A user processor is a processor state image that may reside outside primary memory. A virtual processor is a primary memory resident image (component) of a user processor. The reader may verify for himself that the following dependency graph results:



Similar designs were arrived at without type extension [22, 6]. However, type extension made clear what the problem was. The virtual memory mechanism is a primary memory resident (set of) type manager(s). Its code must therefore be interpreted (processor dependency) by a type of abstract processor of which the state is always in primary memory. This type cannot be a user processor; it must be something different. The virtual processor concept is created for that purpose.

Another example of structural problems involves the mechanism for control, allocation and accounting of storage resource usage (disk space). In the original design, this mechanism was implemented by a collection of unrelated procedures scattered across three modules of the original system. The three modules depended on one another to implement resource management.

The type extension view suggested that a common abstraction, called a quota cell, was implicitly assumed by the scattered resource management procedures. This abstraction was isolated in a stand-alone, dedicated type manager, whence all structural problems disappeared.

Several problems involving the handling of page faults, segment faults, the distribution of segments on disk packs, and the implementation of address spaces for low-level type managers required modifications to existing abstractions or to the hardware.

A huge set of problems dealing with minor modularity and structure violations was easily corrected by a few changes in certain algorithms.

Over all, organization of the new design is very different from that of the original one. However, tracing the similarities between the designs through the various stages of the evolution of the new one was sufficiently easy to conclude that a straightforward implementation of the new design should be feasible and yield a system as practical and efficient as the original one.

5 Conclusions

Type extension appears to be a very powerful technique that can be used to organize virtual memory mechanisms. In addition, we have reasons to believe that it could be applied to other problems, as suggested below.

An interesting research topic could be to analyze the applicability of the technique to future systems: distributed data-base systems, networks

file systems, large transaction systems, etc.

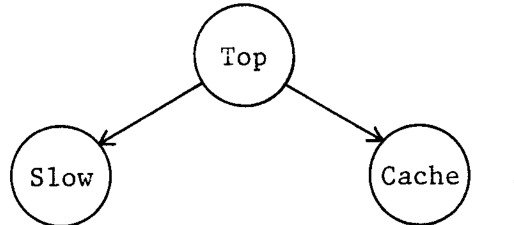
Another possible application area for the technique might be hardware design. In the case study involving Multics, no regard has been given to whether a type manager was implemented in hardware, software or both. Some of the modifications recommended for certain abstractions resulted in hardware modifications. This suggests that the technique may well be applicable in this area.

There is also reason to believe that the technique is applicable to I/O management mechanisms. In applying it to the virtual memory mechanism of Multics, we applied it to the virtual I/O mechanism of Multics, by regarding external devices as information containers. There is no reason why the same approach would not work for user I/O devices.

Finally, while the technique was conceived in the framework of virtual memory mechanisms; we have clearly demonstrated its applicability to virtual processor mechanisms. The very same concept of type extension was used and was in fact necessary to organize both mechanisms.

Actually, the use of type extension in the virtual memory mechanism revealed the existence of a structural pattern that appears to be crucial wherever resources are multiplexed. This same pattern also appears in the virtual processor mechanism of Multics. It is called the software cache pattern.

The pattern involves triple abstractions related as pictured below:

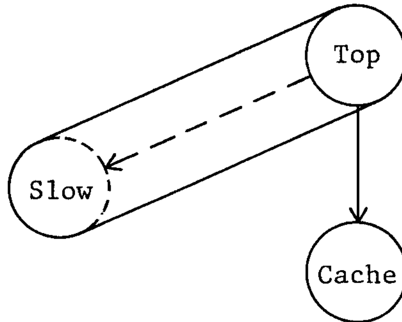


A top abstraction is implemented in terms (composed) of a slow and a cache abstraction designed to contain the same information. However, accessing the information in a container of the slow type is slow, while accessing the same information in a container of the cache type is fast. This models the situation represented by a slow core memory and a fast cache memory, and justifies the name of the structural pattern. The top abstraction is designed to hide the exact location of a piece of information from the users. If a user requests that an operation be performed on an object of the top abstract type, the top abstraction manager will see whether the object desired has a cache component, i.e., whether it has an image implemented by an object of the cache type. If not, the slow-type component is copied into a cache-type component upon which the operation requested is performed. When this cache component is no longer used, it is first copied back to the slow component to update the (slow) image of the object and then deleted. Thus, the top abstraction manager in effect multiplexes fast and presumably scarce and expensive resources (cache type) among slow and presumably cheaper and more abundant resources.

This pattern is found on several occasions in the redesigned Multics

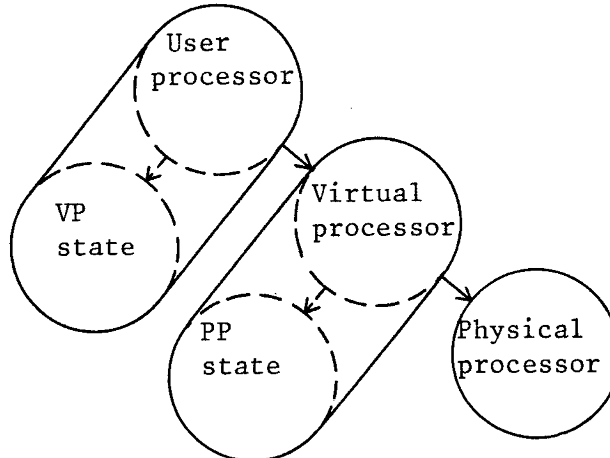
system. The concept of a page (top) is implemented in terms of the concepts of a disk record (slow) and a core block (cache). A page is permanently composed of a disk record. On occasions, when it is used, it is moved into core, meaning that the disk record is copied in one core block of a pool. The page manager, or paging system, multiplexes core blocks among disk records. The pattern is found again for segments, which are implemented by passive and active segments. The latter are more readily accessible as they have their page maps in core. The segment manager multiplexes page map slots in core among page map slots residing on disk.

The pattern is found in several more places in a degenerate form. When the management of the slow abstraction is trivial, the slow abstraction is sometimes merged with the top one, yielding a two-level pattern:



This pattern is found in various places of the virtual memory mechanism and is also found in the virtual processor mechanism. In fact, the concept of a virtual processor is designed to multiplex one or more physical processors among several computations represented by processor states residing in core. Similarly, the concept of a user processor is designed to multi-

plex virtual processors, i.e., in core slots for processor states, among computations represented by processor states residing out of core. The structural pattern of the virtual processor mechanism is pictured below:



A final question that could be raised concerns the efficiency of a system based on type extension. The hierarchical structure of the set of modules suggests that the invocation of an operation of a high-level type manager will result in a cascade of invocations of low-level primitives. Such a cascade would indeed be time consuming considering the cost of inter-procedure calls or inter-process messages in high-level languages. However, while type extension suggests the existence of a cascade of invocations in the high-level language description of the system, it implies nothing at the level of the machine language. The modularity and structure of the system as expressed in the high-level language description may bear little or no relation to modules and calls found in the machine language implementation of the system. Indeed, the compiler used to translate the high-level language should include macro-expansion and global optimization features. This makes it possible to translate high-level calls by doing in-line substitu-

tion of machine code, thus avoiding cascades of calls: hence the claim that using type extension to design a system should not affect the efficiency of its implementation.

References

1. Dijkstra, E.W. (May 1968), The Structure of the THE Multiprogramming System. *Comm. of the ACM, vol. 11, no. 12,* pp. 341-346.

2. Liskov, B.H. (March 1972), The Design of the Venus Operating System. *Comm. of the ACM, vol. 15, no. 3,* pp. 144-149.

3. Schiller, W.L. (republished May 1975), The Design and Specification of a Security Kernel for the PDP-11/45. ESD-TR-75-69 and MTR-2934, Mitre Corporation.

4. Ames, S.R. (April 1975), The Design of a Security Kernel. M75-212 Mitre Corporation.

5. Price, W.R. (June 1973), Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems. Ph.D.Th., Dept. of Computer Science, CMU.

6. Saxena, A.R. (Jan. 1976), A Verified Specification of a Hierarchical Operating System. TR-107, Stanford Electronics Labs.

7. Lampson, B.W. and Sturgis, H.E. (May 1976), Reflections on an Operating System Design. *Comm. of the ACM, vol.19, no. 5.* pp. 251-265.

8. Neumann, P.G. et al. (June 1975, partly modified Dec. 1975), A Provably Secure Operating System, SRI Final Rep.

9. Liskov, B.H. (Feb. 1976), A Note on CLU. CSG Memo 136, Laboratory for Computer Science, M.I.T.

10. Redell, D.D. (Nov. 1974), Naming and Protection in Extensible Operating Systems. Ph.D.Th., U.C. Berkeley & MAC-TR-140, Laboratory for Computer Science, M.I.T.

11. Jones, A.K. (June 1973), Protection in Programmed Systems. Ph.D.Th., Dept. of Computer Science, CMU.

12. Wulf, W. et al. (June 1974), Hydra, the Kernel of a Multiprocessor Operating System. *Comm. of the ACM, vol. 17, no. 6,* pp. 337-334.

13. Parnas, D.L. (Dec. 1972), On the Criteria to be Used in Decomposing Systems into Modules. *Comm. of the ACM, vol. 15, no. 12,* pp. 1053-1058.

14. Liskov, B.H. (1972), A Design Methodology for Reliable Software Systems. Proc. AFIPS FJCC, vol. 41, pp. 191-199.

15. Parnas, D.L. (March 1976), Some Hypotheses about the "Uses" Hierarchy for Operating Systems. Res. BS I 76/1, Tech. Hochschule Darmstadt, Fachbereich Informatik.

16. Habermann, A.N., Flon, L., and Cooprider, L. (May 1976), Modularization and Hierarchy in a Family of Operating Systems. *Comm. of the ACM, vol. 19, no. 5.* pp. 266-272.

17. Parnas, D.L. (Aug. 1971), Information Distribution Aspects of Design Methodology. Proc. IFIP Congress, pp. 340-344.

18. Organick, E.I. (1972), *The Multics System: An Examination of its Structure.* M.I.T. Press.

19. Introduction to Multics. (Feb. 1974), MAC-TR-123, Laboratory for Computer Science, M.I.T.

20. Schroeder, M.D. et al. (Nov. 1977), The Multics Kernel Design Project. Proc. Sixth ACM Symposium on Operating Systems Principles.

21. Janson, P.A. (Sept. 1976), Using Type Extension to Organize Virtual Memory Mechanisms. MIT-LCS-TR-167, Laboratory for Computer Science, M.I.T.

22. Reed, D.P. (1976), Processor Multiplexing in a Layered Operating System. MIT-LCS-TR-164, Laboratory for Computer Science, M.I.T.