MIT/LCS/TR-167

USING TYPE EXTENSION TO ORGANIZE VIRTUAL MEMORY MECHANISMS

Philippe Arnaud Janson

September 1976

This research was supported in part by Honeywell Information Systems Inc., and in part by the United States Air Force Information Systems Technology Applications Office (ISTAO) and the Advanced Research Projects Agency (ARPA) of the Department of Defense of the United States under ARPA Order No. 2641, which was monitored by ISTAO under Contract No. F 19628 - 74 - C - 0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(Formerly Project Mac)

CAMBRIDGE MASSACHUSETTS 02139

This empty page was substituted for a blank page in the original document.

USING TYPE EXTENSION
TO ORGANIZE VIRTUAL MEMORY MECHANISMS

by

Philippe A. Janson

Abstract.
=========

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 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 thesis proposes to use type extension for that purpose. The virtual 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 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 thesis demonstrates its use in a case study involving a commercial, general-purpose, time-sharing system. It concludes by providing some insights into the organization of virtual memory mechanisms for time-sharing systems.

This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology, on 9 August 1976, in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

Thesis Supervisor: Michael D. Schroeder.
Title: Assistant Professor of Electrical Engineering and Computer Science.

Acknowledgements.

These lines implement a fundamental component of an abstract object of type "doctoral thesis". The type manager for a doctoral thesis is realized by two interacting processes, called "committee" and "student". These lines serve as an information container used by the student process to express his gratitude to the committee process as well as to all lower level type managers who provided the moral and material ingredients necessary to compose the thesis.

"A tout seigneur, tout honneur", I would like to thank first my thesis committee for its cooperation in managing this thesis. While an ideal committee looks as a single process to the student process, a real committee is realized by three parallel processes, who fortunately agreed and operated in reasonable synchrony in this case, and I thank them for that. I thank my supervisor process, Professor Michael Schroeder, for his interesting suggestions, his accurate criticisms, his enthusiastic encouragements and the many hours of dedicated work he devoted to this thesis, particularly during the past nine months. I thank my reader processes, Professors David Redell and Barbara Liskov, for their highly appreciated comments on the many drafts of this thesis and for the enriching discussions we had about the research. I would also like to thank a virtual reader process, Professor Jack Dennis, for his comments on early drafts of the first chapters of the thesis, and a remotely cooperating process, Professor Jerome Saltzer, who, through occasional short interrupts, provided us with some crisp and insightful remarks on our research.

Next, I would like to extend my thanks to several people associated with Honeywell Information Systems Inc.: to André Bensoussan for discussing with me the high level issues that presided to the design of NSS; to Bernard Greenberg for helping me discover the marvels and the surprises hidden in NSS; to Jerry Stern for his comments on the implementation of military security controls in Multics; and to Tom VanVleck for his remarks on the quota mechanism of Multics. I would also like to thank some of my fellow students at M.I.T.: Richard Feiertag and Douglas Hunt for the random discussions we had, which generated many ideas and much thinking; Andrew Mason for answering my many questions about the mechanics of NSS; and particularly David Reed for feeding me information on processor management in Multics, for discussing the applicability of type extension to it, and for commenting on the final draft of the thesis.

Perhaps I should thank the Xerox Corporation for having hired 66.7% of my thesis committee on the West coast starting in August 1976, and the Commonwealth Fund of New York -- after all -- for having required that I be out of the country by September 1976. While cooperation of processes in physically distributed systems is a fashionable research topic, the prospect of experimenting with it on a doctoral thesis was not particularly thrilling and has certainly been a strong incentive to all of us to wrap up the work within two years.

Thanks are also due to Ellen Lewis, who typed portions of the thesis, and to Multics, which did not crash too often on me while I was typing the rest.

Last but not least, I express my wholehearted gratitude to those who supplied the bottom level components that made this whole thesis possible: to my sweet wife, Catherine, for her patience and her courage throughout these long years away from the old continent she loves so much; to my little Perrine, without whose distracting interrupts this thesis might have been completed months later (or earlier?); to her expected sibling for not deciding to be born before the thesis was completed, which would have messed up the sequence of events quite a bit; and to my parents, who not only helped fuel my wallet but also, through letters and cassettes and transatlantic calls, provided me with heaps of encouragements.

P.J.
Cambridge, 21 July 1976.

Table of contents.

Chapter I. Introduction. 8

1. Thesis motivations and objectives. 12
2. Background and related work. 16
   - Parallel processes to implement sequential algorithms. 17
   - Partitioning, decomposition and modularity. 18
   - Total ordering vs. partial ordering. 21
   - Functional abstractions vs. data abstractions. 25
3. Approach and thesis plan. 30

Chapter II. Type extension in a virtual memory mechanism. 33

1. A data abstraction model. 34
2. The nature of abstract types. 36
3. The nature of maps. 41
4. Sharing components. 46
5. Protecting internal type objects. 49
6. The nature of type managers. 57
7. Conclusion. 59

Chapter III. Use of the type extension technique to design a virtual memory mechanism. 60

1. Use of the type extension concept. 60
2. Type extension and modularity. 65
3. Type extension and structure. 68
   - Causes of dependency. 68
   - Construction of the dependency graph. 76
   - Conclusion. 80
4. Type extension and deadlock prevention. 81
5. Conclusion. 85

Chapter IV. Case study. 86

1. Introduction. 86
2. The Multics virtual memory mechanism. 89
   - Abbreviations used in chapter IV. 90
   - Processes. 90
   - Pages. 91
   - Segments. 91
   - Address spaces. 92
   - Directories. 95
   - Quota mechanism. 97
   - Removable disk packs. 99
3. Study of NSS. 102
   - Mechanical operation of NSS. 103
   - Organization of NSS. 113
   - Modularity violations in NSS. 116
   - Structure violations in NSS. 117
4. Design of MSS. 125
   - Development of MSS type structure. 127
   - MSS dependency structure. 139
   - Solutions to problem areas. 157
5. Structural patterns. 176
   - Software caches. 176
   - Merging hierarchies. 180
   - Notifications. 182
6. Conclusion. 183
   - Performance. 183
   - Modules as data abstractions. 185
   - Horizontal protection of internal objects. 186
   - Conclusion. 186

Chapter V. Conclusion. 188

1. Summary. 188
2. Results. 190
3. Future research. 191

Bibliography. 196

Appendix. Military security controls in Multics. 200

Biographical note. 210

Table of figures.

4.1. NSS data bases and interconnections. 112

4.2. Best fit model for the organization of NSS. 114

4.3. Fictitious type structure for NSS. 134

4.4. MSS type structure graph. 140

4.5. MSS component dependencies. 141

4.6. MSS dependency graph. 156

I. Introduction.

Research reported in this thesis is concerned with the organization of virtual memory mechanisms for computing utilities. The thesis does not explore new algorithms for managing virtual memories but presents a technique for organizing any virtual memory mechanism given the specifications of its user interface and the available hardware technology. The objective of this technique is to organize the virtual memory mechanism into a structured set of modules so as to make the resulting implementation easier to understand, verify and maintain. We have reason to believe that the technique that is presented can be used for organizing the virtual memory mechanism of almost any kind of computing utility, be it a distributed system, a real-time system or a conventional time-sharing system. In this thesis, the technique has been elaborated within the context of and applied to the design of the virtual memory mechanism of a general purpose commercial time-sharing system of the kind one might use on a university campus or in a research laboratory, for information storage, text editing and interactive program development and scientific calculation. The interest of the thesis is twofold. On one hand, it presents a technique that we believe is applicable to any sort of computing utility. On the other hand, it provides insights into the organization of structured and modular virtual memory mechanisms for conventional time-sharing systems.

A time-sharing computing utility provides a community of users with the ability to share access to a set of hardware resources such as processors, memory and I/O devices, to a library of programs and services, and to the information stored in the system. If access to these facilities were not controlled, abuses might be committed by users of the system. For instance, a

user could take over the system and cause it to deny any service to other users. Or a user could covertly take advantage of programs and services written by others and avoid the charges for the usage of such facilities. Much worse than the above two examples would be the uncontrolled access to the information stored in the system. Such uncontrolled access could result in unauthorized release and modification of that information [Saltzer75] and violation of the privacy of certain people. While the usefulness of computing utilities has been recognized, the need to control access to the stored information has also been realized. The advent of computing utilities has thus fostered the desire for computer systems that are accredited secure, i.e. certified to protect the information they contain.

Much effort is currently being devoted to producing <u>certified secure</u> time-sharing systems [see for instance Neumann75, Schroeder75]. Producing such a system involves several operations [Schroeder75] as shown in the following figure. First, it is necessary to verify that the programs of the system implement the specifications, i.e. to verify <u>correctness</u>. Second, it is necessary to verify that the specifications embody the model of information security that the system must support, i.e. to verify <u>security</u>. Finally, it is necessary to evaluate the verification of correctness, the verification of security and the model of security to <u>certify</u> that the implemented system will meet the security standards demanded by the community of users it serves. Evaluating the verifications of correctness and security means subjectively rating the quality and the effectiveness of the proof techniques or the automatic verifiers that may have been used for the first two steps. Evaluating the model of security means subjectively rating the formal description that this model proposes for the security standards that are only informally defined by the user community. Of the certification process, two

operations -- the verification of correctness of the implementation and the evaluation of that verification -- are of particular concern in this thesis.

implementation (formal)

⇓ verification of correctness is the validation of the formal correspondance between the implementation and the specifications

specifications (formal)

⇓ verification of security is the validation of the formal correspondance between the specifications and the model

security model (formal)

⇓ certification is a subjective evaluation of the two verifications and of the correspondance between the model and the standards of security

security standards

Verifying the correctness of a system and later evaluating that verification are impossible tasks if one attempts to verify the whole system as a single unit. First, existing verification techniques are not sufficiently powerful to establish the correctness of a program of the size of current time-sharing operating systems. (For certification purposes, it is actually sufficient to verify the correctness of the <u>security kernel</u> of the system, that is of the code critical to security. However, even this simpler task is beyond the capabilities of present verification techniques.) Second, even if adequate verification techniques were available, the resulting verification would probably be too long and too complex for any human auditor to read it, understand it, evaluate it and accept the responsibility for

subsequent certification. The purpose of the evaluation is to convince the community of users that the verification of correctness is adequate. For this evaluation to be significant and convincing requires that the human auditor who will certify the system be able to convince himself of the adequacy of the verification. Thus, the verification of correctness should be short, simple and systematic or the task of evaluating it would be hopeless. The use of higher level system programming languages may improve the situation somewhat by making programs more amenable to verification and the resulting verification easier to evaluate. Yet, new system programming languages will not suffice to overcome the verification problem.

The alternative to verifying a whole system as a single unit consists of dividing the system into elements called modules to divide the task of verifying the system into a set of subtasks of manageable size, each of which consists of verifying an individual module. This method presumes that each module is clearly distinct from others and sufficiently small to be verified individually by existing techniques. It also presumes that all modules are organized in a structured way that allows the correctness of the system to be logically inferred from the correctness of its modules. The problem with this method is thus to organize the system into a structured set of modules that are distinct and small.

In addition to ease of verification, two significant advantages are gained from dividing a system into a structured set of modules: understandability and maintainability. Even if one were not going to undertake the verification of a system, dividing it into a set of modules allows analyzing and/or modifying modules individually. Furthermore, the structure of the system may be used to learn about the system in a systematic fashion (e.g., bottom-up or top-down).

1. Thesis motivations and objectives.

The present thesis is concerned with a technique for organizing virtual memory mechanisms for computing utilities in a structured and modular way. Such a technique is desirable because the virtual memory mechanism of a system is, in general, part of its security kernel and therefore should be structured and modular so it can be verified easily.

Some authors [Popek74] claim it is possible to design a system such that the virtual memory mechanism is not part of the security kernel and therefore need not be verified to verify the correctness of the security kernel. We disagree with this claim for two reasons. First, although we believe that Popek's design is interesting and has removed a piece of the virtual memory mechanism from the security kernel, we disagree with the statement that the whole virtual memory mechanism is outside the security kernel. Second, although we believe that Popek's design can be applied with great benefit to simple systems like the PDP-11/45 he has been considering, we believe the benefit of exploiting it for more sophisticated systems is relatively small. Let us justify our position. First, according to Popek's design, the policy decision to bring a piece of data into primary memory -- even security critical data -- is left to the users and cannot affect the security kernel but all the mechanisms (space allocation, disk I/O, etc...) necessary to implement that decision are provided by the security kernel. Thus, only the policy for driving the virtual memory operation is outside the security kernel. The mechanisms, which are secondary in nature but probably larger in terms of code, must be verified as part of the security kernel. Second, the design proposed by Popek is fairly interesting for simple systems in which the functions of multiplexing primary memory and protecting logical information

can be supported together. However, in systems where these functions are decoupled, the relative amount of virtual memory code that can reside outside the security kernel is very small. Consider the following example. In the Multics system [Organick72], protection of information involves mapping segments into user address spaces while multiplexing of primary memory involves mapping pages into primary memory. A direct application of Popek's design to Multics would suggest that, when a user wants to reference a segment, the entire segment be mapped into the user address space and into the system primary memory. This is impractical because, in a system supporting information containers the size of Multics segments, it is imperative that mapping a segment into a user address space be decoupled from mapping its pages into the primary memory. The mechanism for paging segments must be part of the security kernel because it manipulates physical information containers (pages) that implement the logical information containers (segments) protected by the security kernel. If paging were not in the security kernel, it could affect the integrity of the correspondance between logical and physical information containers, thereby defeating the protection mechanism that the security kernel is supposedly implementing. Thus, in relation to Popek's design, the only element of the virtual memory mechanism that should still be under the responsibility of users is the mapping of segments into user address spaces. This design has indeed been recommended for Multics [Bratt75]. However, the amount of code that is removed from the security kernel as a result is very small relative to what remains in the security kernel. One could not claim that this design has removed the virtual memory mechanism from the security kernel.

In summary, we consider that the virtual memory mechanism of a computer system is a part of the security kernel of the system because security depends

on that mechanism in two ways. First, all information secured by the protection mechanism of the system is stored in containers (pages and/or segments) that are handled by the virtual memory mechanism. Thus, if the virtual memory mechanism did not operate correctly, it could lose, damage or interchange information containers, thereby violating security. Second, security critical programs and data bases themselves are too large to be core resident at all times. Thus, the security kernel relies on the virtual memory mechanism to move them in and out of core when necessary. Consequently, the virtual memory mechanism of a system must be verified in order to produce a certified secure system.

The lack of a suitable technique to organize the virtual memory mechanism of a system motivated this thesis. Our objective was thus to develop such a technique to make virtual memory mechanisms easier to understand, verify and maintain. The technique to be presented is based on a concept of type extension to be defined in chapter II. The virtual memory mechanism is regarded as implementing a collection of abstract objects (repositories for data). A different module of the virtual memory mechanism is designed to support every different type of abstract object. The formalism of type extension is used to organize the modules of the virtual memory mechanism into a structure reflecting that of the abstract objects supported by the virtual memory mechanism. That formalism also contributes to simplifying the specifications of the interfaces of the modules that are defined. The task of the designer consists of choosing abstract data types such that they can be implemented by modules of a size amenable to human understanding and hopefully manageable by known verification techniques. The use of a type extension formalism makes the virtual memory mechanism modular and structured in a way that simplifies understanding it, verifying it and maintaining it.

One original aspect of the thesis resides in the exploitation of the systematic approach of type extension in an area of system design to which its applicability has never been demonstrated before. So far, true, formal type extension mechanisms have been used only in programming languages and in areas of system design that are at levels higher than the virtual memory mechanism. To exploit type extension inside a virtual memory mechanism, one must face the issues that uniformly accessible and "growable" information containers are not available.

The type extension technique proposed in the thesis is more than just a means to evaluate and enforce the organization of a virtual memory mechanism. With most software organization techniques, all the hard design decisions are left to the designer of a system in that he must choose how to modularize his system. The organization techniques provide the designer only with rules to enforce modularity and structure after he has defined the role of each module. The type extension technique presented in this thesis provides the designer not only with rules to enforce modularity and structure but also with hints about the possible modularization of his system. In this respect, the type extension technique is closer to a design methodology than most existing organization techniques.

In order to show the use and the effectiveness of the type extension technique, the thesis will exploit it to reorganize the virtual memory mechanism of a commercially available, general purpose time-sharing computing utility, the Multics system [Multics74]. Emphasis will be placed upon the conceptual aspects and the engineering aspects of the technique. Some attention will be given to the performance of the resulting system but an accurate performance evaluation cannot be given since the reorganized system has not been implemented. The modularity and structure of the new design and

their impact on the understandability of the new system will also be discussed. However, we will not attempt to verify the correctness the new system, again because it has not been implemented.

In the remainder of this introductory chapter, we will first review some background notions on the organization of structured and modular systems. While doing so, we will survey the existing literature on related topics. We will then present our method of approach to the problem of organizing virtual memory mechanisms. As we do so, we will outline the plan of the thesis.

2. Background and related work.

The literature is not very abundant in the area of the organization of structured and modular virtual memory mechanisms. However, information on that subject can be found in numerous papers covering a wider topic: the design of structured security kernels and techniques for the organization of structured operating systems. Therefore, we will survey various techniques used in connection with this wider problem and discuss their applicability to the design of virtual memory mechanisms. We emphasize the fact that most of the techniques to be described are compatible with one another and can be used complementarily within one system. It will be seen that several systems indeed exploit more than one technique at a time.

Techniques for organizing a system should achieve two goals. First, the various modules of the system should have the property that they are distinct and sufficiently small to be amenable to human understanding and hopefully to existing verification techniques. Second, the organization of the modules and their interactions must have the property that they allow a verification of correctness of the entire system to be derived from the verification of its individual modules, i.e. that they allow a structured verification of

correctness of the system to be carried out.

A recent paper [Schroeder75] that surveyed the techniques used in the design of a security kernel for the Multics system mentions two techniques that are of particular interest for virtual memory mechanisms. The first technique is based on the use of parallel processes and the second one on partitioning the security kernel into separate protected subsystems.

Parallel processes to implement sequential algorithms.

This first technique consists of implementing a mechanism with programs that can execute in separate parallel processes. A paper [Hoare73] on a structured paging system used the technique to design a mechanism for moving pages of information between the paging device and the primary memory of a computer system. This design is extremely clear and understandable. Individual parallel processes, called monitors, are used for each separate function: allocating primary memory space, allocating paging device space, moving one page into primary memory, moving one page out of primary memory, etc... Each monitor represents a very small amount of code and the interactions between the monitors (i.e. the structure of the system) are well defined and would make a structured verification of the system possible. However, the design is unrealistic in that it requires two monitors for each page of information. It seems impossible to keep the status information for all those monitors in primary memory at all times.

Another design [Huber76], which is very realistic and has been implemented in the Multics system, uses only one process per level of memory to handle the outward movement of pages. Each such process is responsible for continually maintaining free space at the memory level it is managing by purging selected pages to the next outer level of memory. Demand paging is implemented by each user process for itself. This very elegant technique

constitutes a definite step towards the simplification of the design of a paging system: the inward movement of pages can be analysed and verified independently from the outward movement; and the outward movement can be analysed and verified separately for each level of memory. We will retain this technique as a useful and efficient one. It will be used in conjunction with the technique we will present to further simplify the task of understanding and verifying the virtual memory mechanism of a computer system.

The elegance of the above technique and the simplification resulting from its use are not accidental. They are the result of considering the true nature of the mechanism that is being implemented. It is often thought that by implementing all the system functions a user can trigger within the process representing that user, a simpler system will result since the user process is strictly sequential. Although often correct, this attitude should not be adopted systematically. Using user processes to implement some function may actually cause more parallelism to occur. As a result of such a design, all kinds of locks may be necessary to synchronize all user processes. Several functions, like the removal of pages to secondary storage, should be considered from a system point of view even though they are performed as a result of the interactions of individual users. Such functions are better implemented by dedicated system processes than within each user process. While looking parallel to a user, a dedicated system process casts a sequential aspect in the implementation of the function it performs. The resulting implementation is cleaner and more understandable.

Partitioning, decomposition and modularity.

According to Schroeder [Schroeder75], "partitioning is really the same problem as dividing [a system] into separate procedures and data bases, with the extra property that the modularity is enforced by the system's protection

mechanisms". Feiertag has used the term partitioning more loosely to mean dividing a system into modules with or without the protection connotation suggested by Schroeder. To clarify the situation, we will use the term partitioning in the sense proposed by Schroeder and we will use the term decomposition to mean the division of a system into modules without any protection connotation. We will use simply modularizing to mean either decomposition or partitioning. Decomposition is a conceptual technique for defining modules. Partitioning also includes a practical technique for enforcing that definition.

Partitioning is more desirable than decomposition from a verification point of view because the enforced protection allows the designer of a module to depend on certain security properties of his module that are guaranteed to be true if the protection mechanism operates correctly. Those security properties can be helpful in the verification of the module. However, there may be programming environments -- particularily in a virtual memory mechanism -- where the protection mechanisms required to support partitioning simply are not available. In such environments, the designer must satisfy himself with decomposition.

While partitioning is more helpful than decomposition in verifying a system, both techniques are of equivalent help in specifying or understanding the modules of a system. Parnas's specification technique for software modules [Parnas72a, Parnas72b] can be used as well with partitioning as with decomposition. The difference between the two techniques will be felt only at the level of the security properties of a module. With partitioning, those properties are guaranteed to hold because of the protection mechanism. With decomposition, they must always be proved to hold.

Both techniques for modularizing a system consist of dividing what would

be a complex mechanism into a set of simpler mechanisms, and implementing each simple mechanism with one module that can be hardware, software or a combination of both. In either case, the module is a collection of algorithms and data bases. Two concepts of modularity can be found in the literature. Modules in the strict sense [Liskov72b, Parnas72] are isolated collections of procedures and data bases where no data base can be shared by several modules: the data bases in one module can be referenced only by the procedures in that module. Modules in the weak sense [Habermann76, Parnas76] are collections of procedures and data bases that are not isolated. They intersect over data bases that can be shared and managed by several modules. The possibility of shared data bases belonging in several modules presents two disadvantages. First, the boundaries of any individual module are not clearly defined and modules are not clearly distinct. Thus, it is extremely hard to identify and specify simply, precisely, completely and correctly the interface of a module, i.e. the set of channels (including potential shared data bases) over which it interacts with other modules. Second, the formal specification of the interface of a module sharing a data base must contain statements about the behavior of the module with respect to the shared data base. Hence, implementation level information about a shared data base propagates into the design level specifications of the modules that share the data base, thereby making those specifications harder to understand, harder to use in verifying the modules and harder to maintain when changing the implementation of the shared data base. Instead, the interface of a strict module is much easier to identify and specify because strict modules respect Parnas's information hiding principle [Parnas71]. Since there are no shared data bases, modules cannot interact via data bases. Thus, identification of interfaces is easier because they never include data bases. And specifications of interfaces never

contain any information about the implementation of any data base. Based on these observations, we have chosen to work with strict modules in this thesis. (More details on the advantages of strict modules will be given in chapter III.) Unless otherwise stated, the word "module" will implicitly mean "strict module".

Partitioning and decomposition are only names for denoting two similar approaches to replacing what would be a large module by a set of small modules. There are two problems hidden behind those names: how does one choose modules to implement a system and how does one organize them in a way that makes a structured verification of the system possible? We will consider various answers to these questions in the next two sections.

Total ordering vs. partial ordering.

One objective of any module organization in a system is the feasibility of a structured verification of the system. This objective implies that there be an ordering between the modules of the system such that the verification or the understanding of a set of modules can be inferred systematically from the verification or the understanding of subsets of that set. Parnas [Parnas76] has suggested that an ordering based on a relation he calls "uses" is adequate to infer the correctness of a system from the correctness of its parts. A module B is above a module A with respect to the "uses" relation if A provides a service used by B. In this thesis, we will use an ordering based on a relation of dependency. A module B depends on a module A if the correct operation of B depends in any way on the operation of A, i.e. if verifying the correctness of B requires making any assumption about the operation of A. (More details on the dependency relation will be given in chapter III.) A module B that "uses" a module A depends on A because it assumes the correctness of the service provided by A. However, a module B that depends on

A does not necessarily "use" A. For instance, if A and B share a data base, B might not "use" A in the sense that it might not need the service provided by A but B depends on A for preserving the integrity of the shared data base. Two modules that interact via a shared data base might not "use" one another but are automatically mutually dependent on one another. And mutual dependence means a dependency loop in the structure of the system, which makes systematic understanding and verification harder. Thus, the dependency relation is preferred to the "uses" relation because it discourages the use of weak modules, which were deemed undesirable earlier, as they encourage cyclic dependencies. On the other hand, Parnas argues [Parnas76] that considering the "uses" relation and allowing weak modules enhances system efficiency. As will be seen in chapter IV, we do not believe that the loss of efficiency due to eliminating weak modules is substantial. In addition, we believe it is improper to trade clarity for efficiency of code, particularily in products of the size and complexity of current virtual memory mechanisms. (1)

Whatever the ordering relation, two particular ways exist to organize modules into a structured system: the total ordering technique and the partial ordering technique.

The <u>total ordering</u> technique is based on a total ordering of the modules, i.e. each module is ordered with respect to every other module. A module can depend on (use) only primitives provided by lower level modules and is unaware of higher level modules. In many systems, the expression "level of abstraction" has been used to mean modules that are totally ordered. Except

(1) In the absence of weak modules, the dependency relation is equivalent to the "uses" relation. Since we have decided to not use weak modules, unless otherwise stated, the phrase "depends on" will mean essentially the same as "uses". However, the reader must bear in mind the fact that the meanings are similar only because we restrict ourselves to strict modules.

when talking about such specific systems, we will avoid the expression because it is confusing. It has been used by too many authors to mean too many different concepts.

The total ordering technique has been used for the design of numerous systems. Its use has been very successful in structuring the higher levels of a security kernel or an operating system. However, it seems to have been less successful in structuring the lower levels, and particularily the virtual memory mechanism, of a computer system. For some reason, every published system design based on the total ordering technique proposes to implement the virtual memory mechanism of the system with too few modules that are too large to be verified in most cases.

In the THE system [Dijkstra68] and in a PDP-11/45 security kernel [Schiller73], the entire virtual memory mechanism is contained in only one module, called a level of abstraction. Given the relative simplicity of these systems, a one-level virtual memory mechanism may be sufficiently small to be understandable and verifiable by existing techniques. But such an approach cannot be taken in computer systems with more sophisticated virtual memory mechanisms.

In a virtual memory mechanism designed at Carnegie Mellon [Price73, Parnas74] and in an operating system designed at Stanford [Saxena75, Saxena76], the virtual memory mechanism is implemented by two levels of abstraction. The existence of two levels of abstraction perhaps clarifies the structure of the system but it does not necessarily simplify its verification. Instead of each level of abstraction being half the size that a single level of abstraction would be, the levels of abstraction largely duplicate each other. In the Stanford system, the virtual memory mechanism is divided into two levels to eliminate a cyclic dependency between that mechanism and the

virtual processor mechanism, but not to simplify the virtual memory mechanism itself. The lower level of abstraction implements the virtual memory mechanism for a fixed small number of system processes while the higher level of abstraction implements the virtual memory mechanism for a large number of user processes. This design is very interesting to eliminate the cyclic dependency but in terms of size and complexity, each level of abstraction is comparable to what a single combined level of abstraction would be. Both levels of abstraction have to implement resource control and paging for the kind of processes they serve. The CMU system is based on a distinction between fixed size segments existing in fixed number and variable size segments existing in variable quantity. Clearly, the level of abstraction supporting the former kind of segments is somewhat simpler than the level of abstraction supporting the latter kind. However, both levels duplicate one another in implementing paging for the kind of segments they support.

In the Cal system [Lampson69, Sturgis74, Lampson75] and in the SRI system [Neumann74, Robinson75, Neumann75], the virtual memory mechanism is also implemented by two levels of abstraction. The levels of abstraction were chosen so that they hardly duplicate each other. Even so, each level of abstraction seems too large to be amenable to human understanding and to existing verification techniques. In particular, the levels of abstraction really implement more than one abstraction. This tends to confuse understanding and verification as the concepts implemented by the virtual memory mechanism are not clearly separated.

The Mitre Corporation has proposed a redesign of the Multics system [Ames75] that includes a virtual memory mechanism composed of three levels of abstraction. The two lower levels of abstraction (core management and "other storage" management) look like they are sufficiently

small to be amenable to human understanding and verification. However, the higher level of abstraction (segment management) is cluttered with directory control, access control and information backup mechanisms. These mechanisms are not part of the segment manager and ideally should be implemented at higher levels. Unfortunately, the original functionality of these mechanisms, which Mitre has tried to respect as much as possible, suggests that they all share with the segment management mechanism the access to the directories of the file system.

The partial ordering technique has not been exploited as often as the total ordering technique. As its name suggests, it is based on a partial ordering, as opposed to a total ordering, of the modules of the system. It has been exploited in the Venus system [Liskov72a]. Feiertag has demonstrated the advantages of partial ordering in a case study involving the Multics system. The partial ordering technique is more appealing than the total ordering technique because it is more flexible and more natural. The designer is not forced to cast every module into an arbitrary, unrealistic and too constraining, totally ordered structure.

Functional abstractions vs. data abstractions.

The present section will discuss various ways to define the role of the modules composing a system. One may distinguish two kinds of modules: functional abstractions and data abstractions.

A module defined as a functional abstraction appears as a primitive (or a set of primitives) implementing a function that can be applied to data supplied by the caller to transform it in some specified way. A functional abstraction module may have internal state information but usually does not.

A module defined as a data abstraction appears as a collection of objects. The primitives supported by the module appear as channels to

store/retrieve information into/from the objects. A data abstraction module always has internal state information: the representation of the objects it maintains. Any individual primitive (or subset of primitives) of a data abstraction module constitutes a functional abstraction. It is only all together that the primitives constitute the data abstraction.

The data abstraction technique has two advantages over the functional abstraction technique. First, it is the basis of the concept of type extension. This concept was first defined in the context of programming languages (e.g., SIMULA 67). It was then used by various people in the system design area. Different ways for implementing type extension in this latter area were proposed by Jones [Jones73] and by Redell [Redell74]. Type extension mechanisms were first designed for the Cal system [Lampson69, Sturgis74, Lampson76], the Hydra system [Cohen75, Levin75, Wulf75] and the SRI system [Neumann74, Robinson75, Neumann75]. Very recently, the type extension concept was used again in the programming language area for CLU [Liskov76], a language that will be mentioned several times in this thesis.

The formalism of type extension requires that each type of object, that is each kind of data abstraction, be managed completely and exclusively by one module, called a type manager. All the attributes of an object are defined and maintained by the type manager for that object. And any type manager defines and maintains the attributes of only one type of object. It is always clear which module is responsible for which type of object. Transactions with a type manager are restricted to operations on objects, i.e. to invocations of the primitives of the type manager. Thus, a type manager both hides and protects the implementation of the objects it supports. Since the primitives defining a data abstraction are grouped so as to hide the implementation of the data abstraction, the specification of the interface of a type manager

contains no knowledge about the implementation of the objects it supports and is thereby simplified. Since a type manager protects its objects, it naturally tends to not share its internal data bases with other modules. Thus, data abstractions lead to modules in the strict sense. This point will be developed further in chapter III. On the other hand, with functional abstractions, different modules may be responsible for different attributes of the same entity or, more generally, it may not be clear what the entities are. Nothing in the nature of a functional abstraction suggests that it should not share a data base with another functional abstraction. Consequently, functional abstractions may lead to modules in the weak sense, with all the inconvenience that can result [Habermann76, Parnas76]. In summary, the first advantage of data abstractions over functional abstractions is that one can concentrate exclusively on one module when establishing or verifying the properties of one type of objects, and the interface of a type manager tends to define an abstraction more simply, more precisely and more completely than the interface of a (potentially weak) module based on a functional abstraction.

The second advantage of data abstractions is the ease with which the partially ordered structure of the system can be derived from the structure of the abstract objects it manipulates. In a system based on type extension, each data abstraction is defined in terms of more primitive data abstractions and each abstract object is implemented in terms of more primitive objects. Thus, if objects of type T are implemented in terms of objects of types T1 to Tn, the type manager for T will directly depend on (use) the type managers for T1 to Tn. The organization of the type managers directly reflects the structure of the data abstractions they stand for. This organization will be called an <u>object based dependency structure</u> because of the underlying

existence of abstract objects. In general, it corresponds to a partial ordering but it may be cast into totally ordered modules [Neumann75, Lampson75].

However, implementations of type extension, under their current form, present two disadvantages that functional abstractions do not have. First, not every mechanism in a system can be cast easily into a data abstraction. Many system programs have an intrinsic functional aspect that is hard to model properly with data abstractions. For instance, a loader is best regarded as a functional abstraction than as the manager of any type of data abstraction. Second, all existing type extension mechanisms depend on (use) a memory management and information protection mechanism, which may itself be complex. Such a type extension mechanism cannot be used to structure the underlying memory management and protection mechanism because this would create a dependency loop between the two mechanisms, which violates the partial ordering requirement of any module organization. All systems providing a type extension mechanism to their users do not exploit the mechanism within their virtual memory mechanism because the virtual memory mechanism is the memory management and protection mechanism used by the type extension mechanism. At best, one can distinguish the shadow of a type extension concept in the virtual memory mechanism of certain systems. But this shadow is not supported by any actual type extension formalism, much less enforced by any type extension mechanism.

For example, any operating system built around the Hydra kernel can use the type extension mechanism provided by the kernel. However, the kernel itself cannot and does not use the type extension mechanism because it implements the memory management and protection mechanisms that are precisely required to support type extension. In fact, the abstract objects (pages)

protected by the kernel are manipulated by primitives that evoke functional abstractions more than data abstractions [Levin75].

In the SRI system [Neumann75], most modules above the two levels of abstraction that implement the virtual memory mechanism strictly respect type extension. However, modules at and below the virtual memory levels do not. First, some of the abstractions supported are not data abstractions but rather functional abstractions (e.g., interrupt handling, masking, access revocation). Second, even abstractions that look like data abstractions are not implemented as data abstractions. Segments and pages are implemented in terms of lower level functional abstractions rather than being implemented in terms of more primitive data abstractions like disk records and core blocks, for instance. Finally, each level of abstraction violates type extension in that it really implements more than one abstraction, as mentioned earlier. For instance, concepts such as disk I/O, resource control and information backup are not recognized as separate abstractions. They are implemented within the two virtual memory levels of abstraction together with segments and pages thereby making the two levels of abstraction larger and more complex.

In the Cal system [Lampson75], the two levels of abstraction implementing the virtual memory mechanism support two different kinds of file: core files and disk files. As in the SRI system, the formalism of type extension is violated and some of its advantages are lost because the lower level of abstraction implements half a dozen data abstractions instead of just one. Not only does this make that lower level too large for understanding and verification by existing techniques, but it also increases its complexity because some of the primitives it provides involve more than one abstract object.

3. Approach and thesis plan.

From the previous discussion, it appears that the partial ordering technique is more desirable than the total ordering technique to organize the modules of a system into a structured program. The partial ordering technique is more flexible and less restrictive than the total ordering technique. Therefore, it is easier to use and yields a more understandable system.

It also appears that using type extension and designing modules after data abstractions rather than functional abstractions simplifies the resulting system. Not only does type extension naturally lead to a system that has an object based structure, which is -- in general -- a partially ordered structure but, in addition, type extension protects and hides all the details of each abstraction inside a type manager module that embodies that abstraction. Thus, understanding and verification of any abstraction boils down to understanding and verifying the single module of the system that embodies the abstraction (assuming that the abstractions it depends on (uses) are correct).

Unfortunately, data abstractions are not always easy to use because not every mechanism in a system can be cast into a data abstraction and because type extension mechanisms traditionally require the support of a memory management and protection mechanism that provides uniformly accessible, "growable" and protected information containers.

The use of a type extension mechanism to structure a virtual memory mechanism is particularly awkward. First, a virtual memory mechanism may include mechanisms such as paging and resource control, which have inherent functional aspects that are hard to model with data abstractions. Second, a virtual memory mechanism implements the memory management and protection

mechanisms required by existing type extension mechanisms. Therefore, a virtual memory mechanism cannot use a type extension mechanism without creating dependency loops that violate the partial ordering of the system structure.

The purpose of this thesis is to present a technique for organizing virtual memory mechanisms that exhibit an object based structure, that is, virtual memory mechanisms that are implemented by a set of partially ordered modules designed on the basis of type extension.

The originality of this technique resides in the use of a new type extension concept that preserves the advantages of data abstractions over functional abstractions while overcoming the two problems encountered in trying to stucture a virtual memory mechanism with a traditional type extension mechanism. Chapter II will examine in detail the differences between the traditional and the new type extension concepts.

Chapter III will explain how the type extension concept we propose can be exploited to organize the virtual memory mechanism of a system. It will first be shown that type extension suggests and greatly helps building modules in the strict sense. The relation between type extension and dependency will then be studied. It will be demonstrated that the type extension concept helps choosing a set of data abstractions to implement the virtual memory mechanism and deriving the dependency structure that ties the data abstractions together. Finally, the impact of the type extension concept on deadlock prevention will be examined.

In chapter IV, the usefulness and the applicability of the type extension concept will be demonstrated by a case study. The type extension concept will be used to reorganize the virtual memory mechanism of a commercial time-sharing computing utility, the Multics system. The functionality of the

user interface to the virtual memory mechanism will first be described for the readers who might not be familiar with Multics. The present implementation of the Multics virtual memory mechanism will then be examined. The problems associated with this implementation will be pointed out. Finally, a new design that respects the functionality of the Multics virtual memory mechanism and is based on type extension will be presented. Particular care will be given to showing how the type extension concept helps avoid the problems that were pointed out in the current implementation of the Multics virtual memory mechanism. To conclude the chapter, a few observations about the organization of the new design and the benefits of type extension will be made.

While chapters II and III are crucial to the understanding of the type extension concept and its impact on the organization of a system, chapter IV is by far the most important part of the thesis. It represents a demonstration of the use of the type extension technique and reports on the bulk of the research that was performed to support this thesis.

II. Type extension in a virtual memory mechanism.

This chapter will describe the details of a type extension concept designed to be used in organizing the virtual memory mechanism of a computing utility. When we will need to distinguish this concept from all type extension concepts proposed earlier, we will refer to it simply as the new type extension concept or our type extension concept. We will refer to earlier concepts as classical or traditional concepts. These names are used solely to distinguish various kinds of type extension. We do not imply in any way that all classical concepts are identical and indistinguishable among themselves. They are classical or traditional only insofar as they cannot be used inside a virtual memory mechanism, which distinguishes them from our concept of type extension. The advantages of using our concept inside a virtual memory mechanism will be discussed. In order to make those advantages clear to the reader, the differences between our type extension concept and classical type extension concepts, such as those used in Cal, Hydra and the SRI system, for instance, will be pointed out in terms of a data abstraction model. This model will be used as a reference to talk about the features of various concepts of type extension and their ability or inability to solve the problems that are encountered in organizing a structured and modular virtual memory mechanism. Classical type extension concepts and our concept will be examined from five different viewpoints: the nature of abstract types, the nature of abstract objects, the possibility of sharing components among objects, the protection of abstract objects and the implementation of abstractions.

1. A data abstraction model.

The basis for any data abstraction lies in the concepts of abstract objects and abstract types. Our purpose here is to give semi-formal definitions of these concepts.

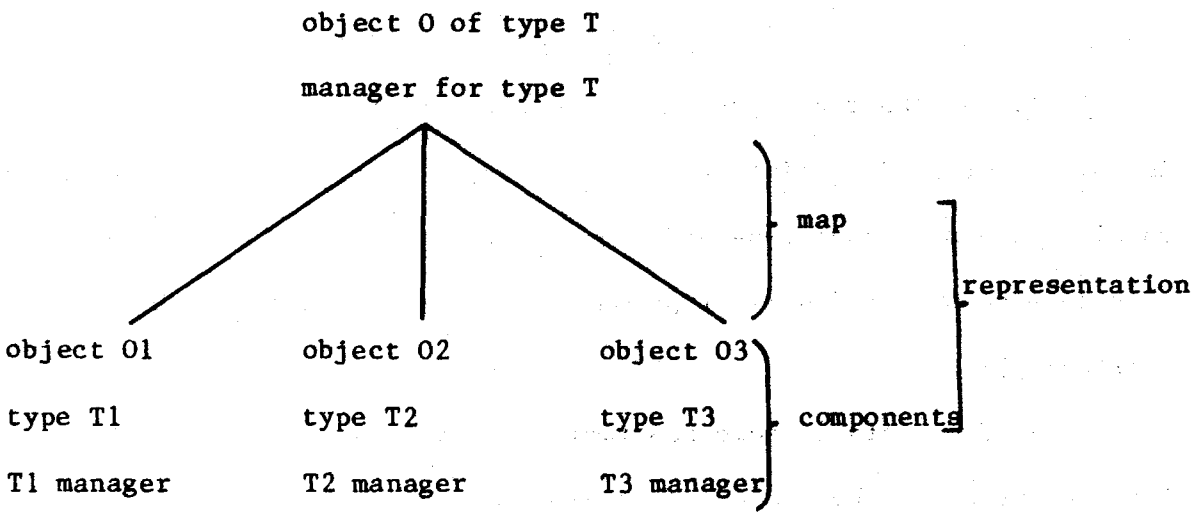
The set of all objects in a system is partitioned into subsets called abstract types. An abstract type has three properties [Liskov76]:

1. An abstract type is defined by a set of operations that can be performed on all objects of that type and are supported by a program designed for that purpose and called the type manager (or the cluster in CLU [Liskov76]).
2. The users of objects of some abstract type need not be aware of the implementation of the objects to manipulate them.
3. In fact, it would do the users no good to be aware of the implementation of the objects, because they are allowed to manipulate them only by invoking the abstract type operations supported by the type manager and not by directly accessing their implementation.

Abstract objects are repositories for structured data. Each abstract object is named by a unique identifier (uid). This uid is the result of the concatenation of a type identifier defining the type of the object and an object identifier. An object identifier is unique over all times but only within each type. Thus, its interpretation is type dependent. A uid is unique over all objects and all times.

All abstract types (except the most primitive ones) are defined in terms of more primitive types (therefore the term "type extension"). An abstract object O of type T is implemented in terms of abstract objects O1, ...,On of more primitive types T1, ...,Tn respectively. Objects O1, ...,On are called

the <u>components</u> of O. The information establishing the correspondance between O and its components is called the <u>map</u> of O. The map together with the components of O is called the <u>representation</u> of O.



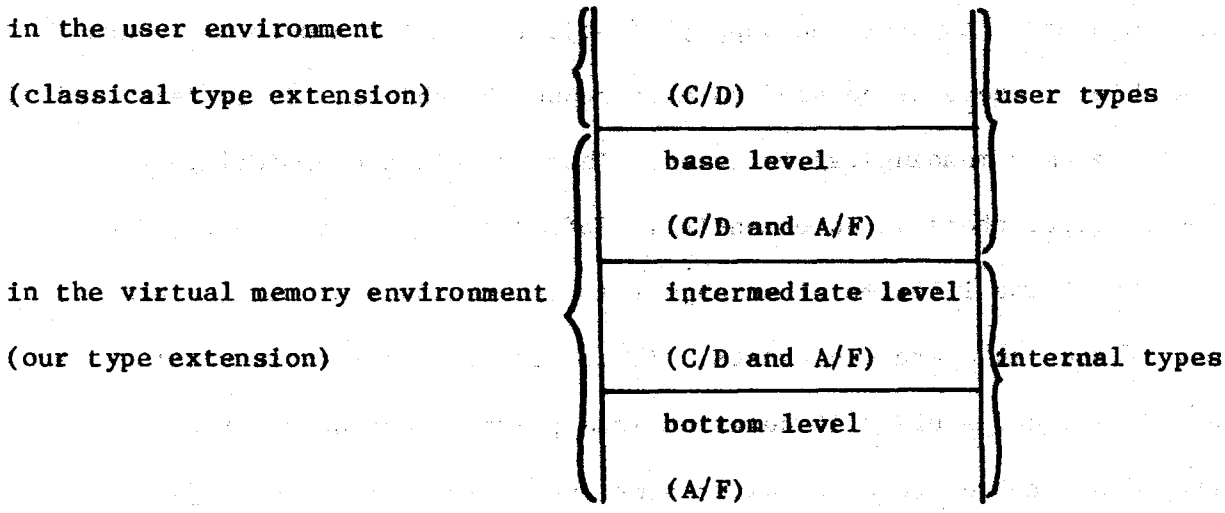
Thus, the complete structure of an object could be pictured as a (possibly multi-level) tree rooted at the object and branching out through the levels of components and subcomponents of the object. Only one level of this structure is visible by any type manager since each type manager knows about the structure of and can access the representation of only the objects it manages. The structure is nonetheless existant as defined by all type managers together. The structure of O reflects a structure embedded in the definition of T. Any object of type T has a structure similar to that of O because type T is defined in terms of types T1, T2 and T3. Similarily, the structure of O reflects a dependency structure that exists between the type manager for T and the type managers for T1, T2 and T3. Since T is defined in terms of T1, T2 and T3, any operation supported by the type manager for T is expressed in terms of operations supported by the type managers for T1, T2 and T3. Thus, the type manager for T depends on (uses) the type managers for T1, T2 and T3

(therefore the term "object based dependency structure").

With the above model of an abstract object, we are in a position to point out the connections and the differences between the classical type extension concept and the one we propose to use for virtual memory mechanisms.

2. The nature of abstract types.

In the rest of the thesis, we will be considering various kinds of abstract types. In order to help the reader visualize what the various kinds are, we have tried to capture their relation in the following figure.



User types are those visible to the users. Users may invoke user type managers to operate on user type objects. The most primitive user types will be called the base level types. Such types are precisely the types of abstract, virtual information containers (e.g., segments or pages) implemented by the virtual memory mechanism for the users. The virtual memory mechanism implements base level types in terms of more primitive types of information containers called internal types, which are not visible to the users. The most primitive internal types will be called bottom level types. Such types of information containers have a direct hardware representation (e.g., disk

records or core blocks). Through several layers of intermediate types, base level objects are ultimately implemented in terms of bottom level objects.

The objective of this section is to discuss the first of five differences between classical type extension concepts and our type extension concept. This difference deals with the supply and lifetime, and with the possible reconfigurability of the objects considered under each concept.

In all existing designs and implementations of the classical type extension concept, the supply of abstract objects of any type is apparently unlimited. Users can always create as many objects as they can afford to pay for. They can never exhaust the supply of objects. After a user has ceased to use an object, there is no need to ever reuse the object. It is destroyed and its uid becomes meaningless for ever. Thus, with the classical type extension concept, there is an essentially infinite supply of objects. And one can say that the lifetime of an object is bounded by its creation and its deletion, if such an operation exists. The object does not exist until it is created and assigned a uid. If deletion is a possible operation, the object is destroyed and ceases to exist after some user deletes it. The object uid is discarded for ever. Such objects will be called create/delete (C/D) objects. The abstract types of information containers provided by the virtual memory mechanism of a system to its users, i.e. the base level types, are in general C/D types.

With our type extension concept, it is possible to define C/D types but it is desirable to define other types as well. The purpose of a virtual memory mechanism is to implement (simulate) an apparently unlimited supply of base level objects using a definitely limited supply of information containers like, for instance, core blocks and disk records. Such information containers are the most primitive types of objects one can conceive. They have a direct

hardware representation. They stand for themselves, have no maps and no components. They are the bottom level objects. A bottom level type provides a supply of objects that is limited because it directly corresponds to a limited amount of physical space. Bottom level objects are of course never destroyed. They are only deallocated and are kept around for later reallocation. They have a physical existence and stay around as long as the system operates (except for reconfiguration circumstances to be discussed soon). Thus, the bottom level objects exist in limited supply and need to be reused time after time. They are never created and deleted as are C/D objects. Instead, they are allocated and freed (A/F). While the lifetime of a C/D object is finite in the sense of being bounded by the user requests to create and delete it, the lifetime of an A/F object is infinite in the sense that a user can only request its allocation and its deallocation. Deallocation does not mean destruction of the object. Nor does it mean invalidation of the uid of the object. The uid is temporarily out of service but not meaningless.

Other types envisioned under our type extension concept (intermediate and base level) may be either C/D or A/F depending on the needs and the possibilities. For instance, since the base level provides users with a large supply of abstract information containers that can be created and deleted, some base level objects are obviously C/D. However, there may also be additional A/F types at the base level, as will be seen in the case study of chapter IV. It will also be seen there that most intermediate types tend to be A/F types though this is not an absolute rule. Some may be C/D types.

Certain types defined under our type extension concept may differ from types defined under a classical type extension concept in terms of the supply and the lifetime of the objects they provide, as we have just seen. Certain

types considered under our type extension concept may also differ from the types considered under the classical type extension concept in that the objects they provide may be dynamically reconfigured. In general, this only requires that the type managers for such types provide two primitives, "configure" and "deconfigure", of which the purpose is to change appropriately the state of the objects they are invoked to operate on.

At this point, it is desirable to discuss the relation between the configure/deconfigure primitives and the allocate/free primitives. To do this, we will relate our concept of type extension to the model of dynamic resource reconfiguration proposed by Schell [Schell71].

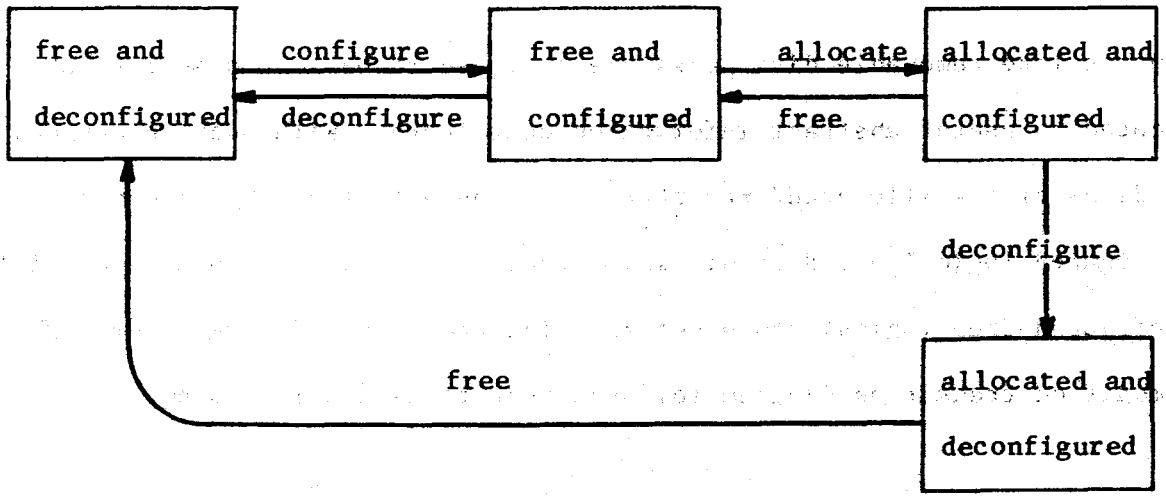
Schell's algorithm for reconfiguring resources first makes a distinction between physical resources and logical resources. An example of a physical resource is a memory box. A logical resource is a physical resource that is accessible. The logical resource corresponding to a memory box is a memory box to which the system is connected. To understand how a physical resource can be configured into a logical resource, we refer the reader to Schell's thesis [Schell71].

We are interested primarily in the configuration of logical resources. Schell's logical resources correspond to pools of our A/F objects. (1) In terms of Schell's model, logical resources evolve between one of four states as explained by the following figure.

---

(1) For readers familiar with Schell's terminology, the following correspondances exist with our terminology:

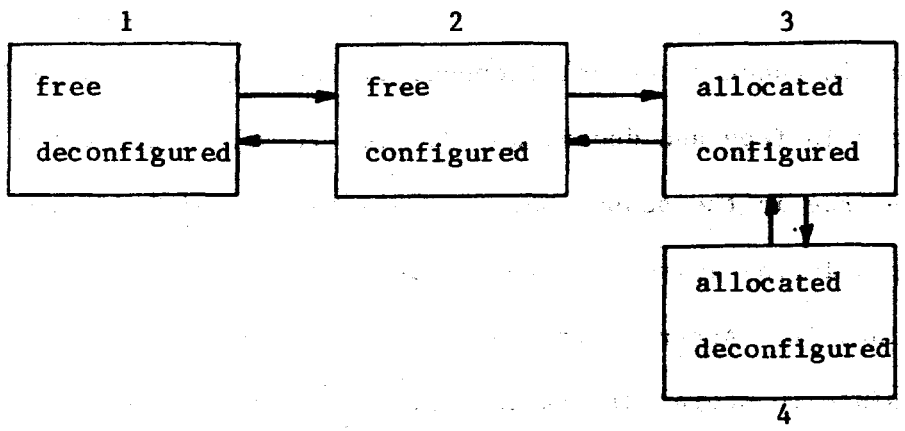
Schell's	ours
available	configured
unavailable	deconfigured
free	free
bound	allocated



Consider for instance a logical (connected) core block. Initially, a logical core block is free and deconfigured, i.e. it may be connected to the system but it is free in the sense that it contains no useful information and it is deconfigured in the sense that the system is unaware of it and cannot store anything into it. After it is configured, the core block becomes logically accessible to store information into it. When it is allocated (for instance, to an address space, a segment or a page), it does contain potentially useful information. When it is deconfigured and if it was free, it goes back directly to the initial state. If it was allocated, it is first marked deconfigured even though it may still contain information and then it is freed and returned to the initial state.

In connection with our type extension concept, the transition diagram deserves a comment. It differs from the one suggested by Schell for certain types of objects. According to Schell, a resource that is deconfigured logically must be freed (emptied of all useful information) before it can be deconfigured physically. This is because Schell has envisioned the reconfiguration of objects like core blocks that cannot retain information while they are disconnected. However, we want to be more general and consider removable disk packs, for instance, as in the case study of chapter IV.

Resources like removable disk packs are designed to be kept off-line but allocated to useful abstract information containers. With such resources, regardless of the allocated/free state of an object, that object can be deconfigured logically and physically without having to return to the initial deconfigured-free logical state before being deconfigured physically. Thus, the modified transition diagram for such objects is the following.



In essence, the resources considered by Schell always had to be deconfigured logically and returned to the free state (1) before they could be deconfigured physically. In this thesis, we also consider resources that need not return to the free state as they are deconfigured logically before they can be deconfigured physically. They can be deconfigured physically from either state (1) or (4).

3. The nature of maps.

The second difference between the classical type extension concept and our type extension concept deals with the nature of the maps of objects supported by the virtual memory mechanism.

Consider the type extension concept in the system design area [Cohen75, Neumann75]. Outside the virtual memory mechanism, the role of object maps is

secondary. The essence of any operation on an abstract object consists of operating on the components of the object. The map itself is affected only when components are added or deleted.

Within a virtual memory mechanism, we will observe just the opposite situation. Most of the code of a type manager will be devoted to manipulating the map of the objects it supports. Very little will be concerned with storing/retrieving information into/from the components of the objects. Such operations are usually implemented in hardware. This contrast with the classical type extension concept (in the system design field) is justified by the fact that our type extension concept is designed to organize virtual memory mechanisms. The essence of a virtual memory mechanism is to perform a mapping function from base level objects to bottom level objects. Thus, most of the information stored in the representation of an object is information mapping that object into its components. Most objects implemented by the virtual memory mechanism consist essentially of their map. (Of course, bottom level objects have no map and consist only of user information that was stored into them.)

With the classical type extension concept, the implementation of object maps is fairly straightforward. Maps are implemented by base level objects. For instance, in Hydra [Jones73], each map is implemented by an individual list of capabilities (c-list). In the SRI system [Neumann75], all maps are collected in a centrally managed segment. Thus, there is a dependency ("uses" relation) between any type manager and the type manager that implements maps. This dependency does not reflect the structure of any abstract object. It is not a component dependency. It is a dependency resulting solely from the need to implement maps with some type of information container.

With our type extension concept, the implementation of maps is no trivial

problem. It is out of the question to envision using base level objects to implement the maps for objects implemented by the virtual memory mechanism since base level objects are themselves implemented by such objects and we want to avoid dependency loops in the structure of the system. The problem with the new type extension concept is that each type manager must reinvent its own type extension mechanism, that is, it must decide what information containers it will use to implement the map of the objects it supports. In order to avoid violating the partial ordering of the abstract types and of the dependency structure of the virtual memory mechanism, a given type manager must implement the maps for its objects in terms of information containers of an abstract type defined at a lower level. As will be seen in chapter IV, if none of the abstract types defined at lower levels are suitable for implementing the maps as desired, an abstract type may be created for that special purpose. (1)

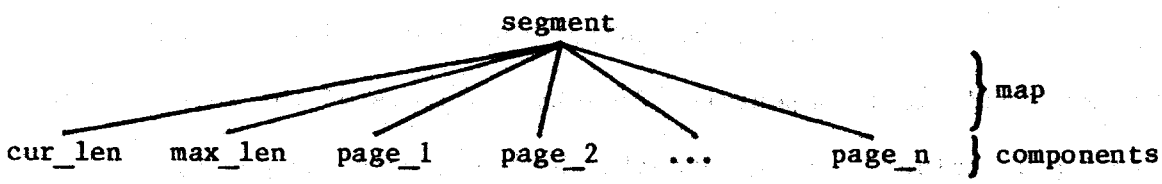
A solution that will be exploited in chapter IV for implementing the maps of the objects of some abstract type consists of collecting the maps of many objects of that type into one information container that constitutes an internal data base of the type manager for that type. Maps are in general very small in terms of storage requirements, often too small to efficiently utilize the amount of physical storage provided by an abstract container of any available type. Therefore, it is desirable to gather many maps into one

---

(1) This observation supports our earlier claim that the SRI system does not really use type extension to organize its virtual memory mechanism. It fails to specify means to implement the maps of the abstract objects (pages and segments) its virtual memory supports. No data abstractions are defined for that purpose. Maps are implemented by some unspecified internal mechanism that is left to the creativity of the system programmer and to the imagination of the user. If a true type extension approach had been taken, the problem of the implementation of maps would have been raised explicitly and would not be left undefined as it is.

abstract container to achieve more efficient utilization of the physical storage. The implementation of maps is a manifestation of a problem often referred to as the "small object problem".

This problem is also encountered in practice (see chapter IV) for the implementation of small components of abstract objects. Its solution in this case is similar to the one suggested for the implementation of maps. Several small components are collected in one information container. Consider a hypothetical abstract type called segment. The structure of a segment is defined below.



The components called page_1, ..., page_n are repositories for user information. They are implemented by abstract objects of type "page". The cur_len and max_len components are repositories for information that is maintained by the segment manager about the current and the maximum number of pages in a segment. From the point of view of programming languages, these components are implemented by objects of type "integer". However, from the point of view of a virtual memory mechanism, we are not interested in the abstract information (integers) that represents these components but rather in the abstract containers that implement them. (1) If there existed abstract

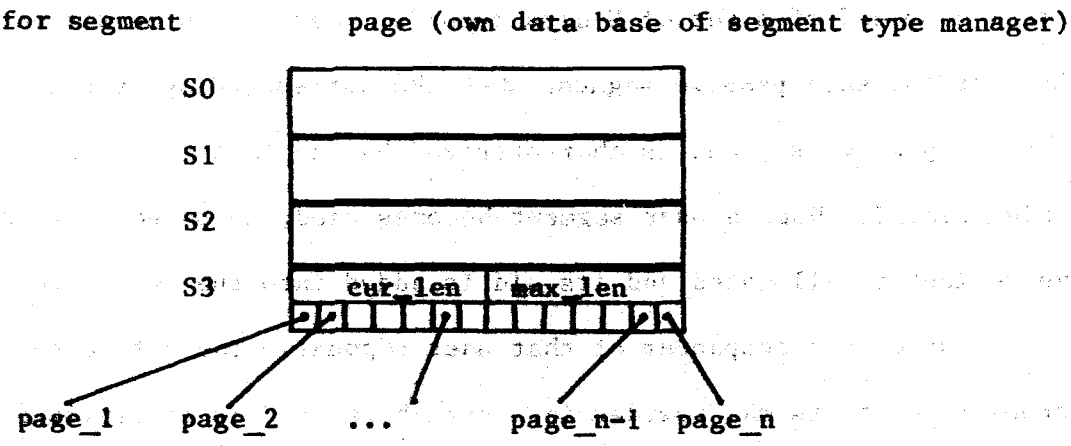
(1) To relate the programming language view of the small components as integers and the virtual memory view of the small components as information containers, one can regard information containers as type generators. The concept of a type generator has been defined in CLU [Liskov76]. Arrays and stacks, for instance, are type generators, i.e. they do not define abstract

containers that could contain efficiently small amounts of information, they would be suitable for our purpose. Unfortunately, existing virtual memory mechanisms do not support small information containers; much to the contrary, they tend to support containers that are as large as possible. Thus, we are faced with the problem of implementing small components with large information containers. To solve this problem, we propose storing the small components of an object in one container.

In fact, we can solve the problem of implementing maps and small components even more efficiently by combining these small objects together. Assuming that small components are not shared by several objects, we can store the small components and the map of an object together; and we can even collect the maps and small components of several objects of the same type in one information container. In our example about segments, we could, for instance, use one page to implement the maps of four segments, together with their small components. In fact, since the small components of an object are implemented together with its map, the map need not explicitly contain the uids (addresses) of the small components. It may directly contain the components themselves as illustrated below. To this extent, the small components are better regarded as <u>attributes</u> than as components of the segment. They are implemented by (stored in) an information container defined below the level of segments. They are interpreted and used by the segment manager. They are analogous to what is called the "data-part" of the object in Hydra [Cohen75].

---

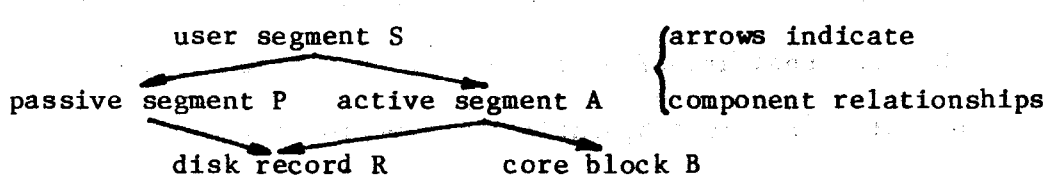
types per se but they can be used in conjunction with abstract types to generate other abstract types, e.g., arrays of integers or stacks of reals.



4. Sharing components.

The third difference between the classical and the new type extension concepts deals with sharing a component among several objects. Such sharing is generally regarded as dangerous because of the side-effects it may have. Yet, use of the new type extension concept for implementing a virtual memory mechanism has shown that such sharing may be desirable precisely for its side-effects. A few comments are in order about shared components.

First, consider how sharing can be caused to occur. The following figure illustrates a sharing situation in a hypothetical virtual memory mechanism.



Five abstract types are involved in the example: disk records and core blocks at the bottom level, active and passive segments at the intermediate level and user segments at the base level. In the quiescent state, i.e. while it is not used, a user segment is composed of only a passive segment. A passive segment

is itself composed of a set of N disk records, where N can be set explicitly by the user for each passive segment when the corresponding user segment is created. A passive segment is characterized by the fact that its map resides in a disk record. When a user segment becomes used, an object called an active segment is allocated and its uid is added into the map of the user segment to make it a component of that user segment. The active segment has the property that its map resides in a core block and is initialized to denote the same N disk records as the map of the passive segment denotes. Thus, when a user segment S is used, its components P and A share N disk records as their components. When the kth page of S is referenced, a demand paging mechanism implemented inside the active segment manager requests the allocation of a core block B, makes B a component of A and copies into B the information contained in the kth disk record (R) of A. Sharing occurs not as a result of the active segment manager having independently requested the allocation of the N disk records but as a result of the user segment manager passing from the passive segment manager to the active segment manager the list of disk records composing P. Sharing cannot be the result of two type managers requesting the allocation of the same components because objects can be allocated only once. Instead, sharing is the result of the uids of the shared components having propagated from the map of one object to the map of another as the result of an explicit operation of some module (the user segment manager in our example) that necessarily executes at a level higher than that of the type managers for the objects sharing components or is one of these type managers themselves. This observation will be used in the next section as we discuss the protection of internal type objects.

A second comment is in order on the use of shared components. The second property of abstract types states that the module using an object of some

abstract type need not be aware of the implementation of the object. This may seem to contradict the fact that the user segment manager has caused P and A to share R and therefore is aware of R. In fact, there is no contradiction if one considers the original intent behind the second property of abstract types. The intent is to relieve the module using an object from worrying about the details of implementation of the object and to hide from it any internal feature of the object that it does not need externally to use the object. That the user segment manager may see the uid for R does not mean that it must know about the implementation of P and A. It does not have to know how R is used by the passive segment manager and the active segment manager. Nor does it have to know how R fits together with other objects to compose P and A. The active segment manager and the passive segment manager hide from the user segment manager internal features of P and A that are of no interest to it. However, if P is a passive segment and A an active segment, there must be a way for modules using such abstractions to talk about the disk records in the segments. Such features are in no way internal. They are explicitly specified by the various abstract type definitions as externally accessible, logical attributes of these abstract types. To compare this to the programming language area, the disk records in a segment should be regarded as the integers in an array. Integers in an array are, by definition of an array of integers, logical attributes that are accessible to the user of the array via the array indexing operations.

5. Protecting internal type objects.

The fourth difference between the new type extension concept and the classical one deals with the protection they afford to the abstract objects they consider.

In order to ensure the third property of abstract types, i.e. the exclusive privileges of a type manager to manipulate the objects it supports, any type extension mechanism requires the help of a protection mechanism. Most existing and proposed type extension mechanisms rely upon a protection mechanism based on the use of capabilities. The uids of abstract objects are locked into capabilities. The first purpose of capabilities is horizontal protection, i.e. the protection of users against each other. A user can request a type manager to operate on an object only if he has a capability for the object. The second purpose of capabilities is vertical protection, i.e. the protection of type managers from their users. A user cannot directly manipulate the representation of an object because the privilege required to turn (amplify [Jones73] or unseal [Redell74]) the capability for the object into a capability for its representation belongs exclusively to the type manager for the object. Unfortunately, a capability mechanism (or any other run-time protection mechanism) appears to be unsuitable and uneconomical to protect internal type (bottom and intermediate level) objects.

Consider bottom level objects in particular. If capabilities were used, they would have to be revocable. With bottom level objects, there must exist a mechanism to guarantee that the user of an object cannot access that object after it has been deallocated. There exist two classical ways to implement revocation in a capability mechanism. With the first, every time a capability is given to a user, its location is recorded in a table. When a privilege

must be revoked, the system can chase and destroy all the capabilities that embody it. This scheme is used in the Multics system [Organick72]. However, it would be very impractical to use it in a system where capabilities can be freely copied. A better scheme [Redell74] consists essentially of maintaining a list of currently valid capabilities and matching against that list any capability that is presented to access an object. With either revocation scheme, a revocation table is required (either to record where capabilities are or to record currently valid ones). In general, there are many objects in a system. Even though there is a limited supply of bottom level objects, that supply may be large (e.g., disk records). Therefore, either kind of revocation table may be very large, possibly too large to be core resident at all times. Since bottom level type managers are below the base level of the virtual memory mechanism, they cannot use base level objects to implement revocation tables. Thus, they cannot take advantage of the I/O mechanism embedded in the virtual memory mechanism to manage revocation tables. Instead, special purpose mechanisms for I/O and revocation table manipulation would have to be provided at the bottom level of the virtual memory mechanism to enable all bottom level type managers to maintain revocation tables. This would not only increase the size and the complexity of the virtual memory mechanism but it might add a sustantial overhead in I/O time to the system due to the management of revocation tables. Thus, using revocable capabilities for bottom level objects would be inefficient and inconvenient. The same argument holds about the inconvenience of any run-time protection mechanism at the bottom level: some potentially large data base is always necessary to decide what accesses should be authorized.

Furthermore, capabilities are too sophisticated and expensive for the protection of internal type objects in general. They are not really

necessary. As far as the users of the system are concerned, they see only base level objects. They do not see internal type objects and never own capabilities for them. The only "users" of internal type objects are the system programs implementing the virtual memory mechanism. Such programs are very different from user programs in that it can be demonstrated through compilation and verification that they do not violate the protection of internal type objects at run-time. Thus, the protection of internal type objects from system programs does not require a run-time protection mechanism. The use of such a mechanism to keep system programs from hurting one another seems an expensive solution to solve a problem that has a much cheaper solution, as is explained below.

In the above discussion, we have tried to suggest that to carry the philosophy of type extension all the way through the lowest levels of a system, it may be desirable to have a protection mechanism different from capabilities and other run-time mechanisms. First, it seems difficult to support revocation if a run-time protection mechanism is used and second, it seems uneconomical to use a run-time mechanism at those levels. Yet, in no way are we suggesting that the use of a run-time protection mechanism must be precluded for all internal types. Furthermore, with new technology, the use of capabilities might become more practical. For instance, the machine described in [Radin76] contains a built-in capability mechanism for doing type extension from the bottom level up. Although one may argue about the cost of manufacturing such hardware today, the proposed machine would greatly help support the type extension concept proposed in this thesis. The protection mechanism we recommend does away completely with run-time protection for internal type objects. Vertical protection is provided by a compile-time mechanism and horizontal protection partly by the compile-time mechanism and

partly by a verification-time mechanism, as will be seen. These mechanisms provide the same protection as a run-time mechanism would provide but at a much lower cost.

Let us first consider vertical protection. We recommend a protection mechanism similar to the one used in the CLU language [Liskov76]. Instead of depending on the capability mechanism provided by the system to the users at run-time, the CLU type extension mechanism depends on a compile-time protection mechanism based on <u>type checking</u>. The CLU system consists of a set of description units (interface specifications) for clusters (type managers) and procedures (functional abstractions). Every time a new CLU program is compiled into the CLU system, the type checking mechanism enforces three properties. First, it verifies that all clusters and procedures invoked (used) by the program being compiled already belong in the CLU system, i.e. that there exist description units for them. Second, it verifies that the type of every argument of every call to every cluster or procedure invoked by the program being compiled is declared to be the type expected by the invoked cluster or procedure, as specified in the CLU system. Third, the type checking mechanism ensures that the program being compiled does not (try to) access the representation of any of the objects it manipulates except for the objects it supports if it is a cluster.

We recommend using such a type checking mechanism to enforce the vertical protection of internal type objects. Thus, modules of the virtual memory mechanism must be formally described to the compiler before compilation takes place so that type checking can be performed at compile-time. The implementation of vertical protection for internal type objects points out one difference between user and internal type objects. The vertical protection of an internal type is not guaranteed by its type manager because the type

manager does not provide capabilities or any other mechanism to protect the objects it implements at run-time. (1) Instead, vertical protection is guaranteed by the compile-time type checking mechanism.

Let us now consider the horizontal protection of internal type objects. Insuring that a module cannot access objects owned by other modules for which it has received no privilege can also be guaranteed by the compiler in most cases. For instance, the compiler should prevent modules from generating random references that would cause them to access local variables in the activation records of other modules. The compiler should also prevent modules from referencing arrays beyond their bounds. This sort of horizontal protection issues do not pose any problem. However, the following issue poses a problem. The problem consists of guaranteeing that no module of the virtual memory mechanism ever uses the uid of an object after that object has been deallocated. In CLU, objects cannot be deleted. Therefore, there is no need to implement revocation of deleted uids. In fact, we suspect that objects are never deleted precisely because there is no obvious way to revoke deleted uids in a user environment like CLU. However, the problem is quite different in a system environment like a virtual memory mechanism.

In a user environment, one must expect a user to delete an object but forget to delete all the uids he has for it or forget to tell other users sharing the object to delete their uids for it. And one must expect a left over uid to be used after the object that it once denoted has been deleted. Thus, a protection mechanism is necessary to prevent the usage of uids of

(1) An elementary run-time type checking mechanism may be built into the programming language for union types [Liskov76], which stand for one of several types, the actual type being determined only at run-time. Notice that we have not felt the slightest need for union types in the particular case study of chapter IV but we do not mean to rule them out in general.

deleted objects. Thanks to a capability revocation mechanism, the user can be prevented from using uids of deleted objects. If he did, he would soon realize his mistake and could take some appropriate corrective action.

In the system environment of a virtual memory mechanism, the above situation can and must be prevented from ever occurring. It must be prevented in any case simply because it would give rise to errors. The users of the virtual memory mechanism do not see internal type objects. They do not have any knowledge about internal type objects. Therefore, they could never take an appropriate corrective action if a module of the virtual memory mechanism used the uid of a deallocated or deleted object. Thus, virtual memory modules should simply never put themselves in a situation in which they could use the uid of a deallocated or deleted object. Using the uid of any deallocated internal type object is a blatant violation of the protection of some base level object because the internal type object may have been reallocated in the meantime. If a core block had been deallocated from one segment and reallocated to another, the user of the first segment could access a portion of the other segment if the segment manager had not properly deleted the uid of the deallocated/reallocated core block from the map of the first segment.

One can prevent the usage of uids of deallocated or deleted virtual memory objects precisely because they are never used or seen in user modules. They are confined to virtual memory modules. And unlike user modules, the correctness of virtual memory modules must be verified prior to their execution. Part of their verification consists of showing that they are coded so as to destroy any copy of the uid of any object when it is deallocated or deleted. We will refer to this property as the <u>conservation of uids</u>: all copies of a uid must disappear from the modules that used it when the corresponding object is deallocated or deleted. Thus, the horizontal

protection of internal type objects is guaranteed not by any run-time protection mechanism like capabilities but by the uid conservation that is checked at verification-time.

The above paragraph points out another difference between internal type objects and user objects. For internal type objects, the type manager does not implement capabilities or any other run-time protection mechanism. Thus, it is not responsible for the horizontal protection any more than for the vertical protection of the objects it maintains. Horizontal protection is guaranteed by the fact that the virtual memory modules are verified to conserve the uids of internal type objects. A type manager for an internal type only guarantees the correct manipulation of internal type objects. It will never allocate/create an object that is already allocated/created. But it does not prevent modules having used an object from referencing it after it is deallocated or deleted. Modules doing so would hurt only themselves or other modules using the same objects. In practice, they hurt neither themselves nor other modules because all modules are verified to properly conserve all uids together. Thus, mutual dependencies that would exist between two modules if each one needed to assume that the other conserved uids properly are eliminated at run-time because uid conservation was checked at verification-time, globally for all modules.

The protection of internal type objects deserves a last comment. We know what a type checking mechanism consists of and we know it can be built. But we do not have similar experience for uid conservation. We do not know if we can verify it systematically, if we can verify it at all. Rather than trying to produce potentially complex rules to write systematically assertions that would guarantee the conservation of uids in all possible cases, we propose a pragmatic approach to formulating the assertions. Experience with the type

extension concept in the case study of chapter IV has taught us that formulating the right set of assertions to guarantee uid conservation in every particular case was easy while trying to produce rules to formulate assertions systematically in all possible cases was extremely hard. In most cases, the uid of a internal type object was found to be confined to the module that requested the allocation of the object. In such cases, one can formulate easily assertions that guarantee the uid conservation within that module. It is simple to require that all copies of the uid made within the single module be destroyed when the object is returned to the free pool. In the few cases where a uid was found to propagate through several modules, we noticed that its propagation was controlled by the module that is highest in the dependency structure of all modules through which the uid propagated. Consequently, it is easy to produce assertions on that one module to guarantee that it controls the destruction of copies of the uid as well as it controls their propagation. Verifying these assertions will automatically suggest to the designer the assertions that may have to be verified about the lower level modules through which the uid propagates.

For an example of such a situation, consider the hypothetical system used to illustrate sharing in the previous section. The uids of disk records propagate through the passive segment manager, the user segment manager and the active segment manager precisely to cause sharing. Yet, their propagation is controlled by the user segment manager because the disk records are ultimately the bottom level components of user segments. The conservation of those uids is guaranteed as follows. Disk records are deallocated only when the passive segment they compose is deallocated. This passive segment is deallocated only when the user segment it is a component of is deleted. At that time, the semantics of the user segment manager require that the

corresponding active segment also be deallocated. In other words, uid conservation does not even require verifying any special purpose assertion. It is guaranteed by the fact that the user segment manager always deactivates a user segment before it deletes it. Of course, one can verify uid conservation only if it can be shown that the active segment manager and the passive segment manager clear (erase) the map of the objects they maintain when they are requested to deallocate these objects. Also, copies stored in working storage must be destroyed. Such assertions may not be part of the original definition of the two type managers but they are easy to formulate and certainly seem reasonable (not too constraining). As was mentioned in the section on shared components, sharing is always controlled explicitly by some module that depends on (uses) or is one of the type managers managing the objects that share components. That module also controls the propagation of the shared uids. Thus, at least in all situations where uids propagate because of sharing, we suspect it should be as easy as in our example to formulate the assertions guaranteeing uid conservation. In all sharing situations of the case study of chapter IV, verifying uid conservation is not a problem.

6. The nature of type managers.

A type manager is an isolated collection of procedures and data bases that constitute a module. In general, an instance of a module is available in every user process in the form of a callable program. If modules are isolated by a partitioning technique based on the protection mechanism of the system, then they are realized by <u>domains</u> that can be invoked only at certain entry points called gates. If modules are defined by a simple decomposition of the system, with no enforcement of isolation by any protection mechanism, then

they are realized in every user process by what Feiertag has called a region that is an unprotected collection of callable procedures. However, there is nothing intrinsic about type managers that implies the above implementation. Nothing precludes realizing a type manager module with a dedicated process and its operations with inter-process messages. This kind of implementation is possible because of the common nature of processes and domains/regions. They are all made up of procedures and data bases. A domain/region is a set of procedures that can be executed sequentially with respect to a user computation. A process is a set of procedures that can be executed in parallel with a user computation. In practice, modules are often realized with a domain/region. However, there may be cases where using processes or a combination of processes and domains/regions is necessary or desirable. The advantage of using a process is that it can take the initiative of certain actions and carry them out in parallel to user processes, which would be impossible with a domain/region that can start operating only upon a synchronous user request. For instance, to implement the asynchronous but sequential page removal algorithm described in chapter I, the page manager would have to be realized by one region that is synchronous to the execution of user processes and one process that runs in parallel to them. Both the region and the process are part of the implementation of the page manager module for the very reason that they manage the page maps that tell whether any page is in or out of core. Thus, clarity is gained because the region and the process perform distinct operations on pages in their own way. However, the interaction between the region and the process cannot be ignored in the verification of the type manager. As will be seen in chapter III, the particular realization of a type manager as a region, a domain, a process or a combination of these does not matter from the point of view of the modularity

of the system because modularity considers only modules and is not concerned about their nature.

7. Conclusion.

This chapter has first defined a data abstraction model. Then, based on this model, the features of conventional type extension concepts were compared to those of the type extension concept we propose to use for virtual memory mechanisms. In particular, we have examined the lifetime and the supply, and the reconfiguration of objects. We have also discussed the problems of implementing object maps, sharing components and protecting objects in the specific environment of the virtual memory mechanism.

III. Use of the type extension technique to design a virtual memory mechanism.

The present chapter will discuss the impact of using the type extension concept defined in chapter II on several aspects of the design of a virtual memory mechanism. First, we will discuss the use of type extension as a technique to organize virtual memory mechanisms. Second, we will discuss the impact of type extension on modularity and explain in what sense the type extension concept fosters strict modules. Third, we will discuss the impact of type extension on structure, we will examine what situations could cause violations of the partially ordered structure of a system, and we will see which of those situations the type extension concept helps avoid or simply eliminates. Finally, we will discuss the advantages of type extension with respect to a locking strategy for avoiding deadly embraces on system data bases.

1. Use of the type extension concept.

It should be kept in mind that the topic of this thesis is not the presentation of a tool for automating the design of virtual memory mechanisms. We are not trying to systematically produce "off-the-shelf" virtual memory mechanisms. (Not because this would be undesirable but just because it is too hard.) We are concerned primarily with recognizing a well-organized design once we see it. Thus, the type extension technique is not a tool for mechanically generating the design. It is mainly a means for evaluating the organization of an existing system design. The type extension concept is a context in which the organization of a system can be evaluated and progressively tuned towards a strictly modular and partially ordered design.

The whole point of the type extension concept is regarding the virtual

memory mechanism as a set of modules, most of which are likely to be type managers, that provide abstract types of information containers. The problem that must be solved is the choice of an "adequate" set of abstract information container types. How successful a designer may be at making a good choice of abstractions depends on his experience and -- shall we say -- his good taste. There is no single best way to organize a system. There are probably many good ways to organize it. How fast a designer can arrive at such a good design depends on his ability to adopt the type extension view, i.e. to visualize data abstractions providing the functionality he has in mind. This may require several design iterations in the course of which the designer may notice modularity and structure problems and should adjust his design accordingly.

While the design of a system and the choice of abstractions may be very hard problems and are left entirely to the designer, the type extension technique more than any other technique provides a few handles on these problems. On the one hand, it provides feedback such that the organization of the system may be evaluated and problems may be detected. On the other hand, it may suggest possible corrective actions for certain problems. In the remainder of this section on the use of type extension, we review only briefly the overall process of exploiting type extension to design a system. More details and specific examples about this process will be given in later sections of chapters III and IV.

Let us first consider the feedback provided by type extension on the design of a system. When the designer believes he has worked out a possible decomposition of his system into modules and before he implements any module, he can construct two graphs that will provide him with useful feedback to evaluate the organization of his system. Such graphs can always be drawn,

even if the system is not well-organized, which the graphs would precisely indicate. Parts of such graphs are shown below for a hypothetical well-organized system.

structure graph	dependency graph
file	file manager
↓	↓
segment	segment manager
↓	↓
page	page manager
↓	↓
core block	core block manager

The structure graph of a virtual memory mechanism is a graph that shows all the abstract types implemented within the mechanism and their component relationships. The graph contains one node per abstract type and one directed arc per component relationship, such that the target of an arc denotes a type that is a component of the type denoted by the origin of the arc. The structure graph of a system is constructed by examining the mechanical operation of the system, envisioning the data bases it should contain, considering the connections between these data bases, and by attempting to view the data bases as abstract objects and the links that connect them as component relationships. The process of constructing the structure graph of a system will become clearer in chapter IV. For the time being, it would be presumptuous to be more explicit or more formal about it because the type extension technique would look like an automatic design tool, which it really is not, and the construction of a structure graph would seem to be a mechanical task, which it is not in reality.

The dependency graph of a virtual memory mechanism shows the complete dependency structure of the system. Every node in the dependency graph stands for one of the modules of the system and every arc indicates that the module

at its origin depends on the module at its tip. The dependency graph of a system is derived from its structure graph by considering that each type in the structure graph corresponds to a type manager module in the dependency graph and each component relationship in the structure graph corresponds to a dependency relation in the dependency graph (because if a type T is composed partly of a type T′, the type manager for T depends on the type manager for T′ to manipulate the representation of objects of type T). The arcs of the dependency graph, however, are a superset of those in the structure graph as they indicate module dependencies generated by many different causes, not just by component relationships. The various sorts of dependency relationships represented by dependency graph arcs and the construction of the complete dependency graph will be examined in detail in section 3 of this chapter.

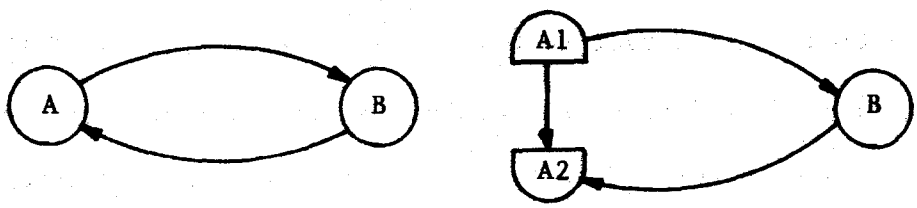
The feedback provided by the two graphs we have just described is the existence of directed loops they might contain. A directed loop in the structure graph violates the type extension rule stating that abstract types may be defined only in terms of more primitive types, i.e. that the structure graph must be partially ordered. A directed loop in the dependency graph should also be eliminated because it means a violation of the partial ordering of the dependency structure of the system.

Let us now consider the possible corrective actions that are provided by the type extension technique to eliminate potential loops in the two graphs. It would be nice to have a complete set of rules for eliminating loops systematically in all cases. Unfortunately, our experience with the type extension technique is limited to the design of only one system. Thus, we do not dare claim that the rules that have proved to be sufficient for eliminating loops in that particular system would be sufficient to eliminate all loops in any system. Consequently, the following are three interesting

but only informal examples of how one may look at a loop and its environment to try to analyze its cause and perceive a way to eliminate it either by adding nodes to or by subtracting nodes from the graphs.

For instance, a loop may exist between two nodes A and B because A and B stand for abstractions that are so tightly coupled that they should be regarded as a single abstraction. Replacing A and B by a single node X obviously eliminates the loop. The cost of this change is that the abstractions represented by A and B are no longer distinct and will not be specified by individual internal interfaces in the final design.

As another example, a loop may be due to a node standing for a complex abstraction that could be split into two simpler abstractions in such a way that the loop is broken as follows:



This type of action is discussed by Parnas [Parnas76]. It was used by Reed [Reed76] to split the Multics virtual processor mechanism (A) into two levels and avoid a dependency loop with the virtual memory mechanism (B). Splitting abstractions in such a way presents the advantage of separating ideas into modules that will be specified by their own internal interfaces in the final design. Of course, it costs the price of maintaining an extra level of mapping between abstractions. In addition, it may not always be possible to find a clean split between two abstractions.

As a last example, loops may be due to a failure to recognize several ideas as parts of the same abstraction (such is the case with the quota problem discussed in chapter IV) or to provide an adequate abstraction where

it is needed (such is the case with the address space problem discussed in chapter IV). Adding appropriate abstractions to the graphs solves such problems.

2. Type extension and modularity.

The objective of the designer of a virtual memory mechanism is to organize his system into a structured set of small and distinct modules. The next section will discuss the impact of type extension on structure. The purpose of this section is to discuss the relation between modularity and type extension.

With the type extension concept, each type manager is a module of the system. As mentioned earlier, each module is implemented by a set of callable procedures (domain or region) and/or by a set of dedicated system processes. While this distinction is important to build the dependency graph of the system, it is largely irrelevant to the evaluation of modularity because modularity is defined with respect to the programs that compose a type manager and not with respect to what process, domain or region type managers are executed in nor how they are invoked.

As suggested in chapter I, a type manager is said to be strictly modular if none of its internal data bases are ever shared with other type managers. Hence, modularity implies that type managers can interact with one another strictly via inter-module calls and/or inter-process signals, depending on how the module is executed. In fact, most of the data kept by a type manager T is data pertinent to the representation of the objects it maintains. By virtue of the definition of type extension, such data may never be accessed by other type managers. In order to access it, other type managers must invoke the operations defined for the type implemented by the type manager T. Thus, the

rule that a type manager should not share any internal data base with other type managers does not add much of a constraint on type managers since most data bases cannot be shared anyway by definition of type extension. Only data bases not directly pertinent to the representation of objects (e.g., metering and performance monitoring information) could be shared but should not according to the modularity constraint. (1)

The main reason for ruling out interactions via shared data bases is, as stated earlier, to keep the interfaces of modules as simple and easy to define as possible, according to Parnas's information hiding principle [Parnas71]. Strict modules tend to define abstractions that are more meaningful, more precise, more complete.

A second reason for ruling out shared data bases is to keep a tighter control on the dependencies between modules. As long as type managers can interact only by calling one another or by sending signals to one another, it is (relatively) easy to keep track of every interaction and to decide whether it raises a dependency problem, as will be seen in the next section. Interactions via shared internal data bases would make the identification of dependencies extremely hard because every shared data base would have to be studied from the point of view of every module sharing it. This study would require a huge (and otherwise practically useless) cross-reference table that, for each data item, indicates which module reads from it or writes into it.

A third reason for eliminating shared data bases is to ease explicit recognition of parallel activity when it is present. If data bases can be shared, the designer of one module may not know whether or not a data base his

(1) One might conceive a situation where several type managers share access to a common read-only data base. Such a situation is acceptable as reading from a non-writeable data base cannot mean interacting.

module is using may be used at the same time by other modules being executed in parallel processes. Failing to recognize such parallelism would almost certainly lead to a disaster. By insisting that data bases never be shared among modules, we make it easier to recognize parallelism explicitly. Knowing that his module is the only one to reference the data bases it uses, a designer is in full control of parallel activity over these data bases. He can code his module so that user and/or system processes that may execute portions of its code in parallel will appropriately synchronize their activity over the data bases they share as parallel instances or portions of the same module.

A fourth reason for disallowing shared data bases is because disallowance helps provide a strategy for avoiding deadly embraces when locking system data bases, as will be seen in section 4 of this chapter.

Eliminating the possibility of sharing internal data bases among modules may sound like a drastic measure likely to impair the performance of the system and complicate the task of the designer. Based on our experience with the redesign of the Multics virtual memory mechanism, we do not think this measure poses any problem in practice. We do not believe that the performance of the Multics virtual memory mechanism would be any better if data bases could be shared. And we have not encountered a single circumstance in the virtual memory mechanism of Multics where a shared data base would have been desirable. In general, we suspect that the desire to share a data base among several modules will not occur to the designers of those modules if they strictly respect the formalism of type extension. Indeed, each type manager has a natural tendency to be a strict module because its main function is to manage an isolated collection of objects, a task which does not require sharing internal data bases.

Eliminating shared internal data bases is a sound decision and should become common practice. If anything, it fosters system flexibility because it makes modules independent of one another in the sense that they do not have to agree on the format and semantics of any shared data.

3. Type extension and structure.

The structure graph of a system indicates what data abstractions are supported by the system and how they are implemented in terms of one another. The objective of this section is to discuss the relation between the structure graph and the dependency graph of a system. First, we will give a definition of the concept of dependency and we will analyze all possible causes of dependency. While doing so, we will see how helpful the type extension concept is to eliminate or point out dependencies that would violate the partial ordering of the dependency graph. Second, we will indicate how to construct the dependency graph of a system, by using the type structure graph.

Causes of dependency.

First of all, it is time to define more carefully the dependency relation. The fact that a type manager A can influence the operation of a type manager B does not mean that B depends on A's actions. In particular, if B implements a service for A, B's operation will obviously be driven by A's commands to B. Yet, B may be declared independent of A. There is no connection between the concept of dependency we are interested in and the fact that the operation of B is influenced by the actions of A. There is a dependency, in our sense of the word, only if the correct operation of B depends on the operation of A, i.e. if some action of A could cause B to operate in a way different from that stated by its interface specification. In other words, B depends on A if verifying B's correct operation requires

using assertions defined in the interface specification of A.

With respect to the partially ordered dependency structure of a system, we define an upward dependency to be one that would violate the partial ordering. Conceptually, all upward dependencies are of course forbidden as they make understanding and verifying the system more complicated. Practically, however, the designer of a system may encounter situations that could result in upward dependencies. We will see in this section that one can distinguish three sets of upward dependencies with respect to the type extension concept. Upward dependencies of the first set, which could potentially occur in a system not based on type extension, are totally eliminated by type extension. Upward dependencies of the second set are not eliminated but are easy to find thanks to type extension and can be corrected. And type extension is of no help with respect to upward dependencies of the third set.

It is interesting to compare in some detail Parnas's "uses" relation, which was mentioned in chapter I [Parnas76], to the dependency relation to justify our use the latter relation in this thesis. A module that "uses" another module depends on it because the correctness of the used module must be assumed to verify the correctness of the first module. However, a module that does not "use" another module may still depend on it, as was suggested in chapter I. For instance, assume that two modules share a common data base. They may not "use" one another in the sense that neither of them ever causes the other to perform any service on its behalf. Yet, they mutually depend on one another because verifying the correctness of either one implies verifying the correctness of the data base they share, which depends on the assumption that the other module always leaves the data base in a consistent state.

One advantage of the dependency relation over the "uses" relation was

mentioned in chapter II: it discourages the use of weak modules (shared data bases) because they cause dependency loops in the structure of the system.

A second advantage of the dependency relation over the "uses" relation deals with the maintenance of the system. With the dependency relation, when a single module is modified, it is sufficient to reverify the correctness of that module and the modules that depend on it to guarantee the correctness of the entire system. With the "uses" relation, it may be necessary to reverify any module, whether it uses, is used by or bears no relation to the modified module, because any module may share a data base with the modified module and its correctness may be affected by changes in the modified module. In other words, a change in the modified module may affect the semantics of a data base, which may implicitly affect the specification of the interface of any module sharing that data base with the modified module, and require that such a module be reverified even though its code may not have changed.

Given the definition of dependency, we can analyze the possible causes of dependency between two modules. The first cause of dependency deals with the way modules interact. We first consider what module interactions are possible. We have just seen that modules may not interact via data bases. We have also seen earlier that modules may be implemented by callable domains/regions and/or by dedicated system processes. Thus, a module may interact with a callable module only by giving control to it. And a module can interact with a process module only by exchanging messages with it.

We now discuss when module interactions may give rise to dependencies. A module A that initiates a transaction with a module B becomes dependent on B if its own continued correct operation depends on an assumed response of B within a finite amount of time. If A initiates a transaction with B and abandons control but does not expect any response from B in any amount of

time, A will be said in the quiescent state (independent of B). In practice, the quiescent state of a module implies that all its internal data bases are consistent, that none of them is locked and that there is no pending procedure invocation in the module.

We finally discuss what forms of interactions yield dependencies, given the definition of the quiescent state of a module. First, if a module A simply sends a message to a module B, does not abandon control and does not expect any response from B, then it is not dependent on B, as explained above. This is a trivial case of module interaction where a module informs another module, as it executes, of some situation it has encountered (e.g., some event to be metered, recorded or accounted for by the recipient of the message). This form of module interaction will not further retain our attention. Second, if a module A sends a message to a module B and then abandons control, a form of transaction we will further call signalling, A depends on B only if it abandoned control in a non-quiescent state, thereby implying that it expects some response from B to resume an unfinished task. Third, if a module A transfers control ("go to") to a module B, it depends on B only if it transferred control in a non-quiescent state, thereby implying that it expects to regain control and complete some pending work. Finally, if a module A calls a module B, it is always dependent on B because a call expects a matching return. To summarize important situations described in this paragraphs, we will use the terms "invoking" and "notifying" as defined below throughout the rest of the thesis. A module A that transfers control or signals an event to a module B in a non-quiescent state, or calls B will be said to invoke (1) the service of B. Invoking a service is synonymous to

(1) Invocations to certain type managers may take the form of an implicit hardware operation if the type manager supports an abstraction that is

using a service in Parnas's sense [Parnas76]. Invoking a module implies depending on it. This is the first cause of dependency. On the other hand, a module A that transfers control or signals an event to a module B and remains in the quiescent state (independent of B) will be said to notify B of some event. Notifications are important forms of interactions. In particular, processor exceptions in a system are notifications by a hardware module to some software module that an exceptional situation requiring software support was encountered. In fact, most instances of notifications may be regarded as error returns from implicit invocations. For instance, an implicit invocation of an operation to access a word of a page that is not in primary memory results in a processor exception that is often called a page fault.

The type extension concept does not prevent upward dependencies due to upward invocations. Indeed, a designer might conceive two modules that invoke one another. The type checking compiler can eliminate upward calls because it knows that they all correspond to upward dependencies. But it cannot eliminate upward invocations corresponding to upward transfers of control or signals because it does not have enough knowledge about modules to distinguish such upward invocations from upward notifications, which are allowed, i.e. it does not know how to decide whether a module abandons control in a quiescent state or not. However, by restricting module interactions to calls, transfers of control and signals, type extension makes the detection of upward dependencies easier to the designer. For every module interaction, the state of the module that initiates the transaction should be inspected by the designer, who has sufficient knowledge about his module to decide whether it is in a quiescent state or not. If it is not quiescent at the time of the

---

implemented partly in hardware. For instance, reading or writing an in core segment are primitives invoked implicitly by hardware operations.

dependencies because it totally ignores types and ranges of values. Argument validation is a very hard problem that has to be carefully dealt with by every individual module designer. The use of programming languages with built-in language type checking features (e.g., CLU) might provide systematic type validation for values but still provides no help with respect to range validation.

The case of references is different. By accepting a reference to an object, a type manager might become dependent on the supplier of the reference (the invoker) and on the supplier of the object denoted by the reference (the type manager implementing the object). It becomes dependent on the type manager of the object if it uses the object (because using the object implies invoking the type manager to manipulate the object). It becomes dependent on the supplier of the reference if it assumes that the reference is meaningful, i.e. that it denotes an accessible object. It would also become dependent on the supplier of the reference if it assumed anything about the type of the object denoted by the reference.

Because of the type checking mechanism built into the compiler, a type manager can never invoke a higher level type manager to operate on a higher level object. In other words, a type manager can never interpret a reference to a higher level object. We will call this the <u>information level</u> rule: high level references cannot be interpreted in low level modules. Thus, the type extension concept systematically prevents the designer from producing modules that accept as arguments references to higher level objects and use them to access the objects. (Notice that the type checking compiler does not prevent a module from accepting and storing references to high level objects as long as it does not use them to access the objects. References that are never used are simply regarded as values and are not type checked as denoting information

containers.)

Because of the type checking feature also, any assumption a type manager may make about the type of the objects for which it receives references as arguments is validated at compile-time. Thus, the type extension concept systematically prevents the designer from producing a module that would be dependent on its invoker because it failed to validate the type of references it has received.

Notice that in most virtual memory mechanisms, associated with references to certain abstract types are certain system events. For instance, a segment fault might be an event associated with referencing a segment of which the map is not in primary memory. A page fault might be an event associated with referencing a page that is not in primary memory. An access violation is an event associated with referencing an object to which access has been revoked or which does not exist. If a type manager A depends on a type manager B, it must be prepared for the events associated with the objects implemented by B to occur when it references them. On the other hand, if it cannot tolerate these events, it should not use objects of type B to avoid being dependent on type manager B. Consider, for instance, the case of access violations in a virtual memory mechanism. For internal type objects (i.e. objects not visible to users, below the base level), the uid conservation property that is checked at verification-time guarantees that while an object is deallocated, no copy of its uid exists outside the type manager that manages it. Thus, if a type manager receives a uid of an internal type object as argument, it can rely upon the fact that that uid denotes an allocated object because that fact was checked when the system was verified. However, for a base level object, a uid or a capability may still exist in some module after the possibility or the right to access the corresponding object has been removed. Thus, if a type

manager depends on a base level type manager and accepts a capability or a uid for a base level object as argument, it must expect running into access violations and be prepared to handle them in some appropriate way.

One comment is in order about a situation where the two causes of dependency (invocation and lack of argument validation) could frequently cause dependency loops. By virtue of the first cause of dependency, a type manager is always dependent on a type manager it invokes to have a service performed. This implies that the invoked type manager may never depend on its invoker because a dependency loop violating the desired partially ordered structure would occur. Thus, instances of dependency due to the second cause (argument passing) must be eliminated in all cases of module interactions corresponding to invocations. An invoked type manager should never assume anything about the input arguments it receives, or if it does, it must validate its assumptions before using the arguments, otherwise it would be dependent on its invoker, thereby causing an upward dependency. Therefore, we impose that the designer of a module performing a service always define in the specifications of the service interface of his module the assumptions it makes about the arguments it expects and code his module to dynamically validate those assumptions before performing the service. (If it did not validate those assumptions, it would be incorrect with respect to its specifications.)

Construction of the dependency graph.

Having analyzed what actions cause dependency, we will now examine the relation between the dependency graph of a virtual memory mechanism and its structure graph by examining all the possible reasons why a type manager might become dependent on another type manager.

The basic relation between the dependency graph and the structure graph of a system is the one to one correspondance between nodes. But there are

many more arcs in the dependency graph than in the structure graph. Every node in the structure graph corresponds to an abstract type. Since every abstract type requires its own type manager, there must be one node in the dependency graph for every node in the structure graph. That node stands for the corresponding type manager. In addition, all arcs in the structure graph have their equivalent in the dependency graph. Every directed arc in the structure graph corresponds to a component relationship. Every operation on an abstract object is translated by the type manager for that object into operations on one or more components of the object. Thus, the type manager translates invocations to itself into invocations to the type managers of the components of the objects it manages. Since the type structure graph should correspond to a partially ordered structure of abstract types, the dependency structure indicates type managers that are partially ordered by the dependency relation. The instances of dependency identified above are called <u>component dependencies</u>.

Component dependencies are not the only arcs in the dependency graph. A second category of dependencies are <u>map dependencies</u>. They were briefly mentioned in chapter II. Any type manager must maintain a map for each object it implements. The maps themselves must be implemented out of some type of information container. Thus, the type manager depends on the type manager for the maps because it invokes it to manipulate the maps.

A third category of dependencies is called <u>program dependencies</u>. Every type manager (except type managers supported by hardware operations) is a program, i.e. a set of procedures and data bases (not including the objects it manages and their maps). The procedures and data bases must be stored in some type of information container. The type manager will be dependent on the type manager for these information containers because reading, writing and

executing its own programs is equivalent to invoking (usually implicitly through the hardware) the type manager implementing the program containers.

Like map dependencies, program dependencies are not a problem in a user environment. With classical implementations of type extension, maps and programs can both be implemented out of base level objects (e.g., segments or pages). However, within the context of the virtual memory mechanism, base level objects are not defined. Thus, other types must be found. This task is not trivial. As in the case study of chapter IV, special purpose types may have to be created to implement maps and programs because none of the types available at a given level may be adequate for this purpose. This is a first instance of a situation where the construction of the dependency graph may point out deficiencies in the structure graph and require a design iteration to modify the structure graph.

The fourth category of dependencies is called <u>address space dependencies</u>. The procedures, data bases and object maps that a type manager uses are objects. The set of all such objects defines the address space of the type manager. In most systems, the address space in which a callable subsystem (domain or region) or a process executes is implemented by some sort of information container or set of registers that contains the complete collection of all uids and capabilities that the subsystem or the process can reference as part of its address space. Thus, every type manager depends on some other type manager to implement the information container materializing the address space it executes in. Again, in a user environment, some base level container can be used to collect the capabilities in the address space of a type manager. (For instance, in Multics, a descriptor segment is used for that purpose. In Hydra, a local name space (LNS) defines the address space of each procedure.) However, in a virtual memory environment, base

level containers cannot be used for that purpose because they are not defined. Special purpose information containers may also have to be defined to implement address spaces. This is a second instance of a situation where the construction of the dependency graph may suggest a design iteration.

The fifth category of dependencies is called <u>interpreter dependencies</u>. Whether a type manager is implemented as a callable subsystem or as a dedicated process, it needs a processor to interpret its code and thus run it, and to control its rate of execution. If each type manager could have a hardware processor all to itself, it would depend on only the correct design of that processor. However, hardware processors are too scarce and expensive resources to be dedicated to executing any single type manager. In practice, type managers, as well as user modules, run on abstract types of processors, sometimes called virtual processors, among which the physical processors are multiplexed. Thus, they depend on the correctness of those virtual processors and on the correctness of the mechanism that multiplexes the physical processors among them. One can envision using a type extension concept for multiplexing (bottom level) processors among abstract (base level) processes that is similar to the concept we use to multiplex bottom level core blocks among higher level abstract pages. A parallel research project [Reed76] that was concerned with reorganizing the virtual processor management mechanism of the Multics system has in fact implicitly used a concept of type extension. The type extension concepts used in the virtual memory and virtual processor contexts are very similar. In particular, they are identical with respect to the dependency relation. The same problems of implementing components, maps, programs, address spaces and interpreters arise with virtual memory type managers as with virtual processor type managers. Virtual memory type managers depend on other virtual memory type managers to implement everything

but their interpreters. For their interpreter, they depend on some virtual processor type managers. Virtual processor type managers depend on other virtual processor type managers to implement components and interpreters but they depend on some virtual memory type managers to implement maps, programs and address spaces. Thus, the virtual memory and virtual processor dependency structures are intimately interleaved. This can be observed in the case study of chapter IV. One must be particularily careful not to cause mutual dependencies of the sort discussed by Saxena [Saxena76] between the virtual processor type managers and the virtual memory type managers. If the dependency graph indicates dependency loops, a design iteration should be performed to modify the structure graph. In fact, such a dependency loop is exactly what prompted Reed [Reed76] to redesign the processor multiplexing mechanism of Multics.

Conclusion.

The concept of type extension does not automatically eliminate all upward dependencies and guarantee a partially ordered dependency structure. However, it does eliminate some upward dependencies which are among the most frequently encountered in existing systems. Accepting reference arguments of an unexpected type is impossible because the compile-time type checking mechanism validates the type of all references passed to a type manager. Using objects defined at a higher level is impossible because of the information level rule that is also guaranteed by the type checking mechanism. A second set of upward dependencies are not eliminated by the type extension mechanism but they are easy to spot and remove thanks to it. Such is the case for upward signals and upwards transfers of control before which a type manager should return to a quiescent state. Finally, certain kinds of upward dependencies are not considered at all by our type extension concept. Such is the case for

the validation of the range of arguments in general. Such dependencies must be watched for explicitly by the designer of every module.

We have seen how the dependency graph of a system can be derived from its type structure graph. By analyzing the implementation of a type manager, one can identify all the type managers it depends on to interpret its code, to implement its address space, to store its programs and to store the components and the maps of the objects it manages.

We have tried to suggest how to iteratively design and evaluate a virtual memory mechanism by inspecting its type structure graph and its dependency graph. We have seen how helpful type extension can be in displaying the modularity and the structure of the system without requiring much more knowledge about its design than is embedded in the dependency graph. After the modularity and the structure of the system are apparent, the designer may produce the formal specifications and the code for the modules.

4. Type extension and deadlock prevention.

In this section, we will study the impact of type extension on a third aspect of the design of a system: deadlock prevention. Deadlocks (otherwise known as deadly embraces) are situations that can occur when several parallel processes compete for resources that can be used by only one of them at a time. For instance, if two processes P and Q decide simultaneously to acquire exclusive usage of two resources A and B, they will need to lock them to avoid conflicts. However, if P locks A first and Q locks B first, neither P nor Q will able to proceed because each will see one of the data bases locked (by the other) and thus temporarily out of service as far as it is concerned. This is a deadlock situation.

Several algorithms have been designed to help designers make sure that

their system is deadlock-free, i.e. that it can never put itself in a deadlock situation [Bensoussan68, Havender68, Habermann69]. These algorithms are undoubtedly helpful. But verifying that a system respects them and is deadlock-free may be hard. In general, it requires the designer of every module of the system to be aware of the deadlock prevention algorithm on a system-wide basis and thus to be aware in some measure of what designers of other modules have done with respect to deadlock prevention.

In this section, we are concerned with a specific kind of deadlocks, namely those that may occur when several parallel processes compete for locking of internal system data bases to preserve the integrity of these data bases. We will show that the type extension concept defined in chapter II, together with the restriction that modules may not share data bases, guarantees the prevention of deadlocks due to competition for locking of internal system data bases. The type extension concept provides a strategy for orderly locking of internal system data bases that guarantees a system-wide respect of Havender's deadlock prevention algorithm, while not requiring any particular attention to or awareness of the algorithm on the part of the designers of the system.

Havender's algorithm is summarized here. With every data base that can be accessed by several parallel processes but by only one of these at a time, one associates a lock (semaphore). In order to be allowed to access the data base, any process must test and set the lock in one atomic (hardware) operation. If it sees the lock already set, it must wait on it. Havender's algorithm requires that all the locks in the system be ordered by associating a number with them. Furthermore, a process may neither test and set a lock X nor wait on it if the relation X>Y holds, where Y is the lowest numbered lock

that that same process has currently set. (1)

This algorithm guarantees the absence of deadlocks due to competition over internal system data bases because any process that wants to lock a set of data bases must lock them in a fixed decreasing order that is the same for all processes. Thus, in our earlier example, if we assume that the fixed lock ordering is B>A, a deadlock cannot occur because P cannot lock A first and try to lock B then. Both P and Q must try to lock B first and whichever succeeds may go ahead, lock A and do its task while the other is waiting on B but not deadlocked.

The modularity of type managers implies that any data base is local to one type manager. Thus, if it is shared by parallel processes, these parallel processes must be executing parallel instances or portions of the same type manager. Since locks are attached to data bases, they are also local to type managers. In other words, locks, as well as the data bases they protect, are never shared across type managers. Let us temporarily assume that there is only one lock per type manager.

The dependency structure of the system imposes a partial ordering on the type managers. Thus, that partial ordering is also imposed on the locks of these type managers. It is true that two locks corresponding to two type managers that are not ordered with respect to one another may, as a result, bear the same number. However, if two type managers are not ordered with respect to one another, they do not depend on one another and in particular, control cannot leave one in a non-quiescent state and go to the other. In other words, there is no computation path that could cause a process to

(1) This is stronger than necessary. Bensoussan's algorithm states that the minimal condition for deadlock prevention is only that a process may not wait on a lock if X>Y but it may always test and set the lock if it can.

execute the code of one type manager while an invocation of the other is pending in that same process. Pending invocations of the two type managers can never exist at the same time in any process. At least one of the type managers must be in the quiescent state as far as that process is concerned, thereby implying that its associated lock cannot be set by that process. Thus, the fact that two locks can bear the same number does not matter. This only means that they are on different computational paths and that no process will ever be able to set them both in any order. What does matter is that on any single computational (dependency) path, all the locks that can potentially be set are ordered.

By virtue of the definition of dependency, if a process is currently executing in a type manager at a level L, all type managers at lower levels are in the quiescent state as far as that process is concerned, and in particular, none of their locks can be set by that process. If execution moves down one level, then a lock at level (L-1) may be set or waited on. This is safe since the current lowest lock set by the process is numbered L. If execution backs out of level L into (L+1), then lock L must be released since the type manager at level L, by virtue of the dependency graph, returns to the quiescent state.

Hence, provided the designer of each type manager is careful to always return to the quiescent state before transferring (or returning) control upward, which we assumed he was to remain independent of higher level type managers, respect of Havender's algorithm is always guaranteed on a system-wide basis.

Now, let us release the original assumption that there is only one lock per type manager. If there may be several locks within one type manager and if several parallel processes executing parallel instances or portions of that

type manager may set them in random order, we again have a deadlock possibility. What is necessary is a local locking strategy in every type manager that prevents parallel processes executing within that type manager to deadlock one another. Type extension does nothing to enforce Havender's algorithm at the level of an individual type manager but since the size of a type manager ideally should be manageable by one person, that person can easily keep track of all the locks used by the type manager and code the type manager so that processes executing its code always attempt to set locks in the same order. In essence, type extension suggests that, from a global point of view, all the locks in a type manager should be regarded as one lock and must be released to leave the type manager in the quiescent state.

5. Conclusion.

In this chapter, we have first examined how and when the type extension technique can be exploited to organize the design of system. We have then examined the relations between the type extension concept and three aspects of the design of a system: modularity, structure and deadlock prevention. We have seen that type extension was helpful in guaranteeing that the system has these three properties.

IV. Case Study.

1. Introduction.

The objective of this chapter is to demonstrate the usefulness and to illustrate the exploitation of the type extension technique described in chapters II and III by using it to reorganize a real virtual memory mechanism.

Our intent is not to design a new virtual memory mechanism, but to show how our type extension technique can be exploited to organize a real virtual memory mechanism. On this basis, there is no point trying to be creative about the functionality of the virtual memory mechanism to be designed. Such creativity would present two dangers. First, it would divert the attention of the reader from the real purpose of the case study. And second, it could result in an unconvincing paper design of an unrealistic, too ambitious or too simplified system.

To avoid these two pitfalls, we have chosen to reorganize an existing and viable virtual memory mechanism. The specific example we have chosen for our case study is the virtual memory mechanism of the Multics system that is offered commercially by Honeywell Information Systems Inc. [Bensoussan72, Multics74, Organick72].

Our choice of the Multics virtual memory mechanism was motivated by two reasons. First, Multics is a large, powerful and sophisticated system. Its virtual memory mechanism is useful and practical but is currently a maze of programs that is certainly unamenable to verification and hard to understand. Because of this complexity, the Multics virtual memory mechanism is as good as any other for our case study. If our type extension technique succeeds in modularizing and restructuring the Multics virtual memory mechanism, the chances are high that it could be used successfully for any other system.

Second, the Multics system was available to us, we were familiar with it and moreover, our case study fitted within the framework of an ongoing research project aimed at producing a more understandable version of Multics that will be a prototype for a future certifiably secure system [Schroeder]. Thus, the design that will result from the case study is of direct interest to the project concerned with the overall reorganization of Multics.

Throughout the rest of this chapter, we will be concerned with several different designs of the same virtual memory mechanism. It may be appropriate to distinguish them here. The original design of the Multics virtual memory mechanism described in [Bensoussan72] has been superceded recently by a new one. The functionality of the original design has been preserved by the new design. However the new design has added several features to the original virtual memory mechanism, notably the possibility to add and remove disk packs from the on-line virtual memory of the system. To distinguish the new design from the original one, the new virtual memory mechanism is called New Storage System (NSS). Our case study will not be concerned with the original Multics virtual memory mechanism. Instead, we will produce a system design with the functionality of NSS. Documentation on NSS is not generally available yet. All we can say is NSS has preserved the functionality of the original design and is similar to IBM's OS/360 as far as removable disk packs are concerned. For lack of adequate documentation, we will devote section 2 of this chapter to the description of the functionality of the user interface of NSS. This section may be skipped without loss of continuity by readers familiar with the functionality of the Multics virtual memory mechanism. Preserving the reverse alphabetical order after OS/360 and NSS, we will refer to the design presented in this chapter as My Storage System (MSS) to distinguish it from the other designs.

Section 3 of this chapter will be devoted to an analyzis of NSS. Analyzing NSS is interesting for two reasons. First, in designing MSS, we will use the design of NSS as a starting point and evolve NSS towards a well-organized system based on type extension. Second, we will describe some instances of violations of modularity and we will explain how these instances contribute to the complexity of NSS. We will also list the most important violations of the dependency structure and we will explain how they are a source of difficulty in understanding NSS. Isolating these problems will allow us to study their nature and will tell us where to concentrate our attention to later evolve NSS towards a well-organized system.

Section 4 will present the design of MSS. Evolving NSS towards MSS was a very long and involved process that required many design iterations. It is impossible and probably uninteresting to describe all these design iterations here. However, we will briefly describe the evolution from NSS to MSS for two reasons. First, we hope that showing the connections and the similarities between NSS and MSS will enhance the credibility of the design of MSS based on the observation that it is so close to the design of an existing system. Second, showing a sketch of the evolution of MSS will illustrate the thought processes involved in designing a system based on type extension.

Section 5 will conclude the chapter by evaluating the impact of type extension on certain aspects of the design of a virtual memory mechanism like MSS. We will summarize observations on particular structural patterns that are encountered several times in MSS and appear to be fundamental to the use of type extension at the low levels of an operating system. This summary is particularly interesting for readers who might read only quickly and superficially the rest of chapter IV and for readers who are interested only in the general conclusions of our case study.

2. The Multics virtual memory mechanism.

Our purpose here is to outline the functionality of the user interface of the Multics virtual memory mechanism we are going to design. This functionality is that of NSS. We intend to preserve it in MSS. Someone familiar with NSS may ignore this section without loss of continuity.

The level of detail we have adopted and the particular aspects we mention here to describe the virtual memory mechanism were selected only on the basis of what we felt -- ex post facto -- posed organization problems and deserved attention in this thesis. Thus, the following description is by no means a complete specification of the Multics virtual memory interface: many aspects of this interface present no design or implementation challenge and are therefore not worth mentioning here. (1) On the other hand, some features of the virtual memory mechanism are a constant source of difficulty to the designer: this explains why certain aspects of the user interface are discussed at great length. As much as possible, we have eliminated from this section the details of the implementation of the Multics virtual memory mechanism. Before diving into the description of Multics, it may be useful to inform the reader that the following table provides a brief and easy to look up summary of all the abbreviations that are used throughout the rest of this chapter.

(1) One of the aspects of the Multics virtual memory mechanism that is not mentioned here is the non-discretionary military protection mechanism that it implements. This aspect presents no new design challenge related to the use of type extension and is omitted from chapter IV. Interested readers will find a discussion of military security controls in an appendix.

Abbreviations used in chapter IV.

APT	active process table (describes states of processes in NSS)
AST	active segment table (describes all active segments)
ASTEP	AST entry pointer (denotes an active segment)
CDSG	core descriptor segment (contains SDWs for system segments in MSS)
CMAP	core map (contains bit map of core)
CRU	current record usage (of a segment)
CSL	current segment length
CU	change usage (operation to update U in a quota cell)
DBR	descriptor base register (denotes the base of a user address space)
DSG	descriptor segment (contains SDWs for connected segments)
DT	disk table (describes all disk packs)
DTAM	date and time access last modified (in directory and VTOC)
DTEM	date and time entry last modified (in directory and KST)
DTM	date and time last modified (a segment)
DTU	date and time last used (a segment)
FSDCT	file system device configuration table (contains bit maps of disks)
KST	known segment table (describes all segments known to a process)
LVID	logical volume identifier (denotes a set of disk packs)
LVRD	logical volume registration data (describes a set of disk packs)
MQ	move quota (operation to move Q between two quota cells)
MSL	maximum segment length
MSS	Multics Storage System (new mechanism presented in this thesis)
NSS	New Storage System (current Multics virtual memory mechanism)
PFT	page frame table (contains all PTWs in MSS)
PLID	principal identifier (denotes a user in the system)
PTA	page table address (denotes the first page of a segment)
PTW	page table word (describes one page of a segment)
PVID	physical volume identifier (denotes a disk pack)
PVT	physical volume table (describes all mounted packs)
PVTX	PVT index (denotes a mounted pack)
Q	quota (upper bound of U in a quota cell)
QCT	quota cell table (contains all (active) quota cells in MSS)
QCTEP	QCT entry pointer (denotes a quota cell in MSS)
RU	reset usage (operation to extract and reset the TRP of a quota cell)
SDW	segment descriptor word (describes a connected segment)
TRP	time-record product (integral of U in a quota cell)
U	usage (in a quota cell)
UHT	UID hash table (hashes UIDs into ASTEPs)
UID	unique identifier (denotes a segment)
UPT	user process table (describes states of user processes in MSS)
VPT	virtual processor table (describes virtual processors in MSS)
VTOC	volume table of contents (describes passive segments)
VTOCX	VTOC index (denotes a passive segment)

Processes.

Every user authorized to use a Multics system at a given site is named inside that system by a principal identifier (PLID). This PLID is used to

control the user's access to the information stored in the system. While a user is actually using the system, he is represented internally by and interacts with a process bearing his PLID. What a process can reference is determined by the PLID that is tagged on it. The physical processors are multiplexed among all processes.

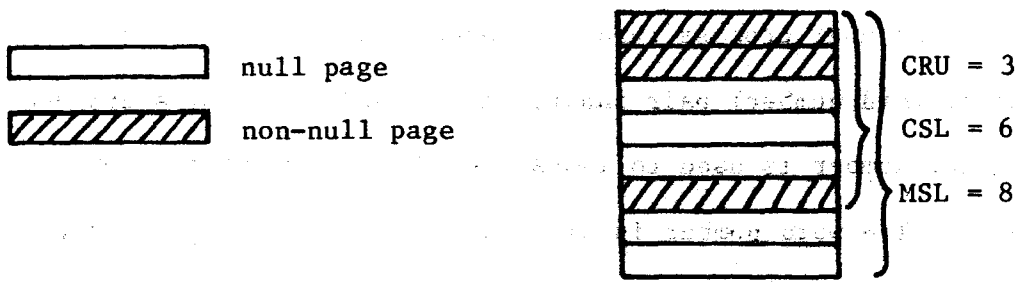
Pages.

Multics supports a demand paged virtual memory. A user never has to worry about bringing the information he is going to reference into primary memory or copying an area of primary memory out to secondary storage to make room for other information in primary memory. When a user references a piece of information, it is brought into primary memory automatically if it is not already there. Then, if it is not used for a while, the piece of information is removed automatically by the system and flushed out to secondary storage. Primary memory and secondary storage are divided into logical blocks/records of a fixed number of sequentially numbered words. The information represented by any such block/record is called a page. Information is moved automatically from primary memory to secondary storage or vice versa by entire pages. If a user references a page that is not in primary memory, a processor exception known as a page fault occurs. It causes the user computation to be suspended temporarily and directs the system to bring in the desired page. A user can observe page faults insofar as he can see the delays they cause. He is otherwise unaware of pages.

Segments.

A user never references pages directly. A user references segments. A segment is a logical collection of pages that are sequentially numbered. The location of the pages of a segment are denoted by the words of its page table. Segments have many attributes that a user can get at or set. Most of these

pose no design challenge. Two kinds of attributes deserve some attention here as they will be referred to later. First, are the date-time-used (DTU) and date-time-modified (DTM) attributes. They indicate to the users the date and the time a segment was respectively last used and last modified. Second are the length attributes of a segment. The maximum-segment-length (MSL) is a user definable attribute defining the maximum number of pages that a segment may contain. The current-segment-length (CSL) is the number of pages a segment currently contains; it is determined on the basis of the highest numbered page of the segment that is not full of zeroes (null page), as shown in the following figure.



Finally, the current-record-usage (CRU) of a segment indicates how many pages are not null. The user is charged for the space occupied by his segment on the basis of this quantity. He pays only for non-null pages because null pages are not actually allocated. If a page is null, the page table word (PTW) that describes it does not denote anything. The page has no image and occupies no physical space. Physical space is allocated and the user is charged for it only if and when the page contains some non-zero bit.

Address Spaces.

Now comes the question of how a user references a segment. Every user process has its own address space. This address space is implemented by a descriptor segment (DSG). A descriptor segment has a fixed, small MSL. Every entry in a descriptor segment is called a segment descriptor word (SDW) and is

indexed by segment number. A SDW is represented schematically below.

page table address	fault tag
current segment length	access modes

Every SDW describes one segment. It tells the user about the absolute page table address (PTA) and the CSL of the segment. It also indicates the access modes (read mode, write mode, execute mode, etc...) that constrain what the user can do with the segment. The meaning of the fault tag is examined later. While the process of a user is running on a physical processor, the PTA of the process DSG is loaded into the distinguished descriptor base register (DBR) of the processor. The user process then references a piece of information by a (segment number, word number) pair indirectly through the DBR, a SDW and a PTW. The segment number is used to index the DSG and get at the SDW of the desired segment. The word number is divided by the page length to obtain the number of the desired PTW within the page table of the desired segment. And the remainder of the division is used as an index for the desired word within the desired page. Retrieving the SDW and the PTW on every reference is speeded up by the use of hardware associative memories that hold the most recently used SDWs and PTWs [Schroeder71].

If a user references a segment beyond its CSL, a processor exception known as a bound fault occurs. As a result of it, the CSL attribute of the segment is increased unless it is already equal to the MSL. In the latter case, a bound violation is signalled to the user and the computation is aborted.

Since there may be very many segments in the system, it is impossible to keep the page table of each of them at a fixed address in primary memory all the time. When a segment is not being used, its page table is removed from

primary memory and stored on secondary storage, where it is called the file map. This is called deactivating the segment (the inverse operation is called activation).

It is now time to explain the first role of the fault tag in a SDW. The first role is to signal changes in the physical attributes of a segment across processes. If a bound fault or a deactivation occurs, all processes that have a SDW for the segment must be notified so they can update this SDW if they care to. To signal bound faults and deactivations, the fault tag is turned on in all the SDWs that denote the segment that is affected, to cause processors to trap if they try to access the segment. This is called disconnection (the reverse operation is called connection). If any of the disconnected processes later references the segment, it causes a processor exception known as a segment fault. As a result, that process is made aware that something has changed. Either it has to update the bound of the segment it referenced in the SDW it trapped on or it has to retrieve and perhaps reactivate the segment page table.

The obvious question to ask now is: how does a process reconnect itself to a segment? It must be able to retrieve the page table of the segment and to recompute its CSL. For this purpose, the system maintains system-wide data bases (to be described soon) where it records the CSL and the current PTA of every segment. On a segment fault, the faulting process must extract from these data bases the information it needs to perform the connection. This requires that the process be able to translate the faulting, hardware interpretable segment number, which identifies the segment within the process address space, into some sort of unique name that identifies the segment within the entire system and permits the process to address the system-wide data bases. For this purpose, every process owns a table called a known

segment table (KST) that maps every segment number into a unique name. In fact the KST maps each segment number into two unique names that are synonymous in denoting the same segment but are meaningful at different levels of the system, as will be seen. (The implementation may confer more functions to the KST, but these have no effect on the user interface.)

Both the DSG and the KST exist only as long as the user process exists. Every time a process is created , a new KST and DSG are provided. The connection operation described earlier fabricates a SDW in the DSG from information that is in the KST entry with the corresponding segment number. The next question to answer is: how does that information get into the KST entry in the first place or, in other words, how is a segment added to the address space of a process?

Directories.

The file system is composed of a set of system-wide data bases called directories. Although directories appear different from segments to the user, they are stored like segments by the system. Directories are used for four functions: naming, addressing, protection, and resource accounting.

The naming function consists of providing system-wide unique names for denoting segments. Each directory entry describes either another directory or a segment. Thus, the file system appears to the user as a hierarchical tree of directories (nodes) and segments (leaves). (1) The tree is rooted at a distinguished directory called "root" that is not described in any directory entry. Every entry in a directory bears two names: a symbolic name (character string) that is unique within the directory and the (numerical) unique

(1) The Multics terminology is often unfortunate, particularly about segments. Most of the time, a segment means a logical collection of pages. But in the context of the file system, a segment means a leaf of the tree. We will try to make clear what kind of segment we mean when we use the word.

identifier (UID) of the segment or directory described by the directory entry. UIDs are unique over the entire system. Thus, a directory entry is uniquely defined by its path name in the tree and uniquely recognized by the UID it contains. The path name is user oriented; the UID is system oriented. UIDs are easy to store and use within the system but path names are provided to users as they are easier to use to name directory entries. Adding a segment to an address space is called initiating the segment (the inverse operation is called terminating). (Initiating a segment is somewhat synonymous to opening a file on other systems.) When a user requests that the segment described by a given directory entry be initiated, a KST entry is allocated, the symbolic name of the directory entry and the UID of the segment it describes are copied into the KST entry, and the corresponding segment number is returned to the user. Notice that path names and UIDs are entirely decoupled ways to identify a directory entry in the sense that the path name may change while the UID remains the same. If a user has initiated a segment with a given UID, he can still use that UID to refer to the segment even if the name of some directory entry along the tree path for reaching the segment has changed.

The addressing function of the file system consists of providing a way for the user to get at the physical attributes of a segment (PTA, CSL). This function is required to activate a segment or to recompute its CSL, as mentioned in the description of the connection operation. The directories are the system-wide data bases mentioned there.

The protection function consists of providing an access control list for every segment and for every directory. An access control list determines what PLIDs can acquire access to the associated segment or directory and how. The exact functionality of the access control list mechanism is of no interest here [see Saltzer's paper in Multics74]. The only point that deserves

attention here is the revocation of access. A user A can cause the access of another user B to some segment to be revoked by changing what the access control list says. The functionality of Multics requires that this change be effective immediately even if user B has initiated the segment and is currently connected to it. Access revocation is the second purpose of the fault tag in a SDW. It is turned on if access is revoked. When trying to reconnect a disconnected segment, the system recomputes the access of the faulting process and denies reconnection, signalling an access violation if access has been revoked.

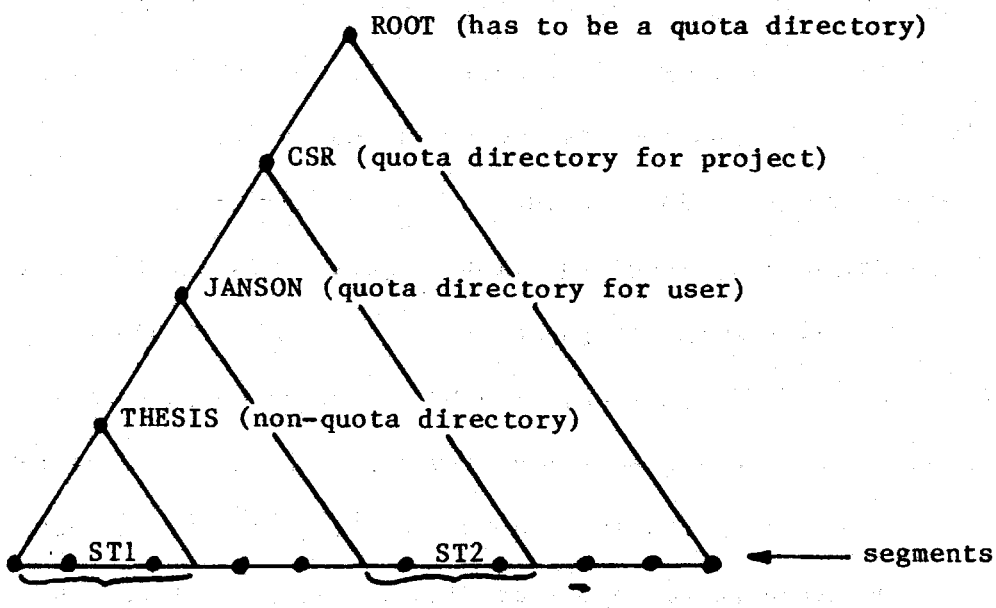
Finally, the resource accounting function consists of providing system administrators with a means to bill users for the space used by their segments. It also provides a means to project administrators or individual users to control their own resource usage. The resource accounting function, otherwise known as quota mechanism, poses many design problems and is extremely complex to describe. The following is a simplified description of the mechanism that preserves the design challenges of the original mechanism while trying to eliminate some of the complexity of its functionality.

Quota Mechanism.

Resource accounting is done on a per directory basis. Every directory has associated with it a quota cell. The resource unit is one page. (Notice that users do not pay for directory pages in the current implementation of Multics.)

There are two kinds of directories: quota directories and non-quota directories. Although the following restriction does not quite hold in theory in Multics, let us assume for simplicity that a directory can be a quota directory only if its parent is a quota directory. This restriction appears to hold in practice, as far as users are concerned. Thus, quota directories

are aggregated at the higher levels of the directory hierarchy as in the following example.



The quota cell of a non-quota directory contains only a running total U of the number of pages used by all the segments (sum of their CRUs) described in that and inferior directories. For example, the quota cell of THESIS indicates the number of pages used by segments in subtree ST1. The quota cell of a quota directory contains resource usage, resource control and resource accounting information. The resource usage information, further referred to as usage (U), consists of the running total of the number of pages used by segments in that and inferior non-quota directories. Thus, for CSR, U is the number of pages used by segments in subtree ST2. The resource control information, also known as quota (Q), is an upper bound for U. The resource accounting information, also known as the time-record-product (TRP), is the integral over time of U.

The quota mechanism supports three functions: reset usage product (RU), move quota (MQ) and change usage (CU).

The RU function performs accounting. It is clear from the above

discussion that the sum of the TRPs of all quota directories is equal to the integral of resource usage for the whole system. Thus, by periodically extracting the TRP of all quota directories and resetting them to zero, system administrators can bill the owners of those quota directories for the resources they have used and recover the total cost of pages used during the elapsed accounting period. This is the purpose of the RU operation.

The MQ operation is intended to allow project administrators and users to put upper bounds on the amount of pages they are willing to pay for in any directory or subtree of directories. The MQ operation moves portions of Q from a parent directory to its son or vice versa. The parent directory must be a quota directory. As a result of a MQ operation, the son directory may change status from quota to non-quota or vice versa. In either case, and in order to preserve the meaning of Q relative to U, the MQ primitive must update the U value of the parent directory. If Q is delegated to a son that becomes a quota directory, then U (son) must be subtracted from U (parent) as it is now charged against Q (son). If all Q is removed from a son that becomes a non-quota directory, then U (son) must be added to U (parent) as it is now charged against Q (parent) and Q (son) is zero.

Finally, the purpose of the CU operation is to update the U of quota cells when segments are grown or shrunk. Again, to preserve the meaning of a Q with respect to the associated U, when a segment is grown/shrunk, a page must be added to/subtracted from the U of the quota cell of every directory above the affected segment up to and including the lowest quota directory. If U = Q in that directory, then the segment may not be grown and a <u>quota overflow</u> condition is signalled to the user.

<u>Removable disk packs.</u>

We are now almost done with our description of the functionality of the

Multics virtual memory mechanism. We must yet describe the removable disk pack feature. This description will be extremely simplified for two reasons: as we are writing this thesis, the exact functionality of removing disk packs is not defined because the implementation of NSS is not completed, and the functionality as we perceive it does not seem to pose any major design challenge.

The purpose of having removable disk packs is to be able to not have all segments on-line all the time, i.e. to extend the available virtual memory to off-line packs.

The organization of disk packs is governed by two concepts: physical and logical volumes. Physical volumes are physical packs. Logical volumes are collections of physical volumes. Two rules apply to the organization of information on volumes. First, all the pages of a segment must reside on one physical volume. Every segment has as a tag the physical volume its pages belong in. Second, all the segments in one directory must reside in one logical volume. Every directory is tagged with a logical volume attribute defining the logical volume its son segments belong in. When a segment is created, its home logical volume is determined from the logical volume attribute of its parent directory. The home physical volume is selected as the least full of the physical volumes in the home logical volume. If the system lacks physical space to grow a segment on its home physical volume (out-of-physical-volume (OOPV) condition), it may move the entire segment to the -- then -- least full physical volume of the home logical volume.

The removability of disk packs is governed by two rules. First, they must be mounted/demounted by entire logical volumes so that all the segments in one directory are either on-line or off-line at the same time. Second, one logical volume, which is called the root logical volume and contains all

directories and some segments critical to system operation, can never be demounted. Thus, the skeleton of the file system hierarchy, i.e. the directories, is permanently accessible on-line. As a consequence, it is always possible to find out about the unique names, the secondary storage address and the access control list of a segment, and to access a quota cell. However, it is not always possible to access or find out about the attributes (DTU, DTM, CSL, MSL, CRU) of any segment. This is possible only for segments stored in logical volumes that are on-line. If a user tries to access the content or the attributes of a segment that is off-line, he will receive an error message indicating that the desired logical volume is not on-line. In order to reference an off-line segment or its attributes, it is necessary to first request explicitly that the right logical volume be mounted.

For each logical volume, there exists registration data (LVRD). The LVRD specify which physical volumes are in the logical volume. These physical volumes cannot be mounted individually. The LVRD also specify what directories of the file system are master directories for the logical volume. The segments in a directory D are normally stored in the same logical volume as the segments in the parent directory of D. However, if D is a master directory for some logical volume, the segments in D are stored in that logical volume. In other words, a master directory is recognized by the fact that its LVID attribute was not inherited from its parent but is determined by some LVRD.

There is an obvious connection between the space that is physically available in a logical volume and the fact that that space will be shared by segments described only in the master directories of the logical volume and in inferior non-master directories. In view of this connection, quota on a master directory cannot be adjusted by moving it to/from its parent directory

since the parent directory, by definition, controls resource allocation on a different logical volume. Thus, a master directory is a boundary for the MQ operation: quota cannot move between a master directory and its parent. In reality, it will be possible to adjust quota on a master directory by moving it to/from the quota pools of a space partition defined in the LVRD of the corresponding logical volume. However, the functionality of this mechanism is not yet fully defined and the mechanism is not yet implemented. Since the mechanism does not seem to pose any design challenge in a system based on type extension, we will ignore its details and and assume for simplicity in this thesis that quota on the master directories of a logical volume can be adjusted externally, in an unspecified way, by the executives (see below) of the logical volume.

For every logical volume, there is also an access control list. This access control list specifies the authorized "users" and "executives" of the logical volume. The "users" can mount (use) and demount the logical volume. The "executives" can move quota between the master directories of the logical volume and quota pools defined in the LVRD of the logical volume. Only the person known as the "owner" of the logical volume can set the access control list of the logical volume.

3. Study of NSS.

The purpose of this section is to examine how the virtual memory mechanism that was described in the previous section is implemented in NSS. While studying NSS, which is not well organized and not based on type extension, may sound irrelevant to the topic of this thesis, there are two reasons for doing so. First, MSS will be evolved directly from NSS with amazingly few changes. Thus, it is interesting to describe NSS as it is the

basis for MSS and is very similar to it in many respects. Second, in studying NSS, we will point out the problem areas in its organization. This will allow us to concentrate later on these areas to analyze the nature of the problems and solve them in MSS.

To characterize NSS, we will first describe its mechanical operation by surveying the data bases it contains and their management. We will then step back from the detailed mechanical description of NSS to give a somewhat abstract description of its organization. This will allow us, in a third step, to point out examples of problems associated with the modularity of NSS. And finally, we will isolate problems associated with its structure.

Mechanical operation of NSS.

Our objective here is to describe the mechanical operation of NSS in a way that will prepare the reader to later visualize data abstractions in NSS, which was not designed with type extension in mind, and to evolve it towards MSS, which is explicitly based on type extension. From chapter II, we know that an abstract object is a repository for data. Thus, in more common terms, an abstract object is a data base or a data item. Therefore, to arrive at a data abstraction view of NSS, it may be helpful to describe its mechanical operation from a data base point of view. Adopting such a data base view is exactly what helped us to produce MSS from NSS in reality. Thus, the following discussion will gravitate around NSS data bases and their interconnections. This discussion does not pretend to give a full description of all data bases. First, it provides only information necessary to gain an overall understanding of the mechanical operation of the system. And second, it omits the description of specific operations that cause organizational problems as these operations are described later in more detail.

a. Logical volumes. (1)

When a user wants to operate on a logical volume, he can name it by a path name. The path name denotes a segment in the file system hierarchy that contains the LVRD for the logical volume to be operated on. This LVRD contains, first of all, a logical volume identifier (LVID) for the logical volume under concern. The LVID of a logical volume is a system oriented name for talking about a logical volume. While users never see LVIDs, the system never uses path names to denote logical volumes. The LVRD of a logical volume also contains a list of physical volume identifiers (PVID) that specifies which physical volumes compose the logical volume, and a list of path names that specifies which directories of the file system are master directories for the logical volumes and how much quota these master directories may have. Every time a user wants to mount/demount a logical volume, the list of PVIDs is used to mount/demount the physical volumes of the logical volume. Every time a user wants to create a master directory for some logical volume, the list of master directories is used to determine if the directory to be created may be a master directory for the given logical volume and if so, to set the quota on it. The LVRD finally specifies if the logical volume is demountable.

b. Physical volumes.

Thanks to a hash table associated with a data base called the Disk Table (DT), every PVID is mapped into an entry of that DT. This entry describes the corresponding physical volume (device type, size, configuration, etc...). In particular, the entry indicates if the physical volume is demountable and if

(1) The information in this section may not reflect the true implementation of NSS because that implementation does not yet exist as we write these lines. The implementation that is described here is believed to be a close approximation to what will eventually happen to be a real implementation. At any rate, the functionality of the mechanism is respected.

it is currently mounted. If the physical volume is currently mounted, the entry specifies which drive it is mounted on by giving the index (PVTX) of the entry corresponding to that drive in a table called the physical volume table (PVT).

c. Configured volumes.

The PVT is a system table that contains one entry for every disk drive. Thus, a PVTX identifies one drive. A PVT entry indicates if the corresponding drive is currently used. If it is, the entry indicates whether the configured volume (the physical volume mounted on it) may be demounted. (This information duplicates what is in the DT and the LVRD.) The entry also indicates the PVID of the configured volume and the LVID of the logical volume of which the configured volume is a component. The PVT contains two more pieces of information. First, it contains a hash table that allows searching the PVT to find out what drive (PVTX) a physical volume with a given PVID is mounted on. While it is possible to find what drive (PVTX) a PVID is bound to by looking up the description of the given physical volume in the DT, it is also possible to find out about this binding by searching for the drive description in the PVT. (Notice that while a PVID is given and a PVTX is expected in return, this operation is not an operation on a physical volume. It does not affect any physical volume in any way. It concerns a configured volume. This seemingly irrelevant comment will have a tremendous importance later.) Second, when a logical volume is mounted, all PVT entries that are allocated to mount its physical volumes are threaded into a circular list. Thus, given a PVTX for one configured volume of a logical volume, it is possible to get the PVTXs for all others. The above two pieces of information are used to create and move (OOPV) segments, and to activate them, as will be seen soon.

d. Directories.

A user may refer to a directory by its path name. However, as soon as he has initiated the directory in his process, he will address it by segment number, as does the system internally. A directory contains three data areas: an entry name hash table, a UID hash table and directory entries. The entry name hash table is used to find an entry in the directory, given its symbolic name. The UID hash table is used to find an entry in the directory, given its UID.

As mentioned in section 2 of this chapter, every directory is also tagged with the LVID of a logical volume. That LVID indicates the logical volume on which the son segments of the directory should be stored. (Notice that the LVID is never used to operate on (mount/demount, etc...) the logical volume. It is only used as a common tag for all the segments in the directory. This observation will have a fundamental impact on the design of MSS.)

Every entry in a directory describes a file. (The name file is used to mean either a directory or a segment.) A directory entry associates its own symbolic name with the UID, the secondary storage location and the access control list of the file it describes. The access control list is a list associating PLIDs with the access they should be granted to the file denoted by the entry. The secondary storage location of a file is defined as follows. Every physical volume is divided into two zones: the paging zone and the cataloging zone called the <u>volume table of content</u> (VTOC). The paging zone is used to store file pages and the cataloging zone is used to store file maps. A VTOC entry contains one file map. The secondary storage location of a file is defined by the PVID of the physical volume where the file is stored and an <u>index</u> (VTOCX) to the entry where the map of the file is stored in the VTOC.

To create an entry in a directory, i.e. to create a file, the LVID tag of

the directory is used to search the PVT until a configured volume with a matching LVID is found. (If none is found, a logical-volume-not-mounted error is returned.) The circular list of configured volumes with that LVID tag is then searched to find the least full configured volume. A VTOC entry is then allocated on that volume. A new UID is created. And the (UID, PVID, VTOCX) for the new file are stored in the new directory entry.

e. Processes.

While we are interested in the virtual memory mechanism of Multics, we need to say a few words about the virtual processor mechanism insofar as it interacts with the virtual memory mechanism. (More information on the virtual processor mechanism can be found in [Reed76].)

Every user attached to the system is represented by a process. The system maintains a table called Active Process Table (APT) of which each entry describes the state of one process. An APT entry bears as a name the PLID of the user owning the process. It binds the PLID to a DSG and a KST for the user process. The DSG and the KST are permanently connected segments, i.e. there always is a SDW to address them in the DSG. If they were not permanently connected, the system could never connect them because connecting a segment requires addressing a DSG and a KST, as will be seen.

f. Initiated segments.

In order to address a file by hardware, it is necessary to acquire a segment number for it. This is the initiation operation mentioned earlier. It is impossible to initiate a file in a process unless its parent directory is already initiated in that process because initiating a file requires addressing the directory entry that describes it. To initiate a file, the user must supply the segment number of its parent directory and the symbolic name of the entry describing the file in the parent directory. An entry is

then allocated in the KST, and the segment number of the parent directory, the UID of the directory entry and the symbolic name of the directory entry are stored into it. The parent directory segment number in every KST entry binds that entry to the KST entry for the parent and the (parent directory segment number, UID, entry name) triple binds the KST entry to the directory entry both by name and by UID. From then on, the file is said to be known with a segment number equal to the sequence number of the KST entry allocated to it.

While it is known, it cannot be addressed yet because it is not connected and it may be inactive. When the user first uses the segment number just allocated to the file, he takes a segment fault. As a result, the system looks up the faulting segment number in the KST. It follows the (directory segment number, UID) binding from the KST entry to the directory entry to find the access control list and the (PVID, VTOCX) of the faulting segment. It uses these information items to activate the segment if necessary and to connect it in this process, as explained below. (Notice that the system follows the UID binding rather than the entry name binding from the KST entry to the directory entry. Thus, it is not affected by potential changes in the name of the directory entry.)

g. Segments.

NSS is ambiguous about the definition of segments. In this paragraph, we mean to talk about stored segments, i.e. logical collections of pages, as opposed to file system segments, i.e. leaves in the hierarchical file system. In the quiescent state, a segment is materialized by its VTOC entry. The VTOC entry contains its UID, its file map and its storage attributes (DTU, DTM, CSL, MSL, CRU).

To activate a segment, the system uses the (PVID, VTOCX) stored in the directory entry describing the segment. It first searches the PVT to find the

PVTX corresponding to the given PVID. It then uses the resulting (PVTX, VTOCX) to access the VTOC entry of the segment. It proceeds to find a free entry in a core resident table called the Active Segment Table (AST). (If there is no free entry, it deactivates another segment, as explained later.) It finally copies the VTOC entry into the AST entry, thereby bringing the page table of the activated segment in core. It also adds an entry to the UID hash table (UHT) of the AST to map the UID of the activated segment into a relative pointer (ASTEP) to its AST entry.

In addition to the page table and the storage attributes of the active segment it describes, an AST entry also contains the (PVTX, VTOCX) denoting the secondary storage home of the segment. If the segment page table or its storage attributes change while the segment is active, they are copied back to the home VTOC entry when the segment is deactivated. Finally, an AST entry contains a trailer list. A trailer is an (ASTEP, segment number) pair that identifies a SDW (denoted by its segment number) inside a DSG (denoted by the ASTEP to its AST entry). The trailer list of an AST entry denotes all SDWs in any DSG that are connected to the active segment. It is used to disconnect these SDWs if access to the segment is revoked, if its bound changes or if it is deactivated.

h. Connected segments.

For every KST entry, there is one SDW. (1) Having initiated a segment is sufficient to address it by segment number. However, a segment fault occurs when a disconnected SDW is used. On a segment fault, the system follows the

(1) Notice that the inverse is not true. Some SDWs that are permanently connected permit addressing of certain system segments that are not described in any KST entry. Segment faults and bound faults on such segments cannot be taken because they require the existence of a KST entry to be handled properly.

binding from the KST entry with the faulting segment number to the directory entry describing the segment that must now be connected. In a way to be explained later, it inspects the access control list to determine the access mode of the faulting process to the faulting segment and it copies the computed mode into the SDW. It then uses the (UID, PVID, VTOCX) of the segment to retrieve its CSL and page table. If the segment is active, the UID hashes directly into an ASTEP. Otherwise, the segment is first activated, by using the (PVID, VTOCX), to get an ASTEP for it. In either case, the system then finds the CSL and the page table in the AST entry denoted by the ASTEP, stores the PTA and the CSL in the SDW, and turns off the fault tag in the SDW.

(When a bound fault occurs, the system proceeds as for a segment fault to retrieve the CSL of the segment to be grown.)

i. Pages.

The paging mechanism is based on three data bases: PTWs, the Core Map (CMAP) and the File System Device Configuration Table (FSDCT). For every configured volume, the FSDCT contains a bitmap indicating the allocated/free status of every disk record on that volume. The CMAP contains one entry per core block. If a core block is used, its associated CMAP entry points to the PTW that denotes the core block and to the disk record that is the home of the page currently in the core block. A PTW contains the disk or core address of the page it stands for and a "used" flag and a "modified" flag that describe the state of the page. These flags are maintained by the hardware as the PTW is indirected through.

The paging mechanism operates as follows. On a page fault, if the PTW faulted on is null and if the quota mechanism (see later) allows creation of a page, the system looks in the AST entry containing the faulting PTW for the PVTX of the home volume of the segment being grown. It searches a free disk

record in the bitmap associated with this volume in the FSDCT. If there is no free disk record, it signals an OOPV condition (see later). If there is a free record, it allocates it and copies its address in the PTW.

If, on a page fault, the PTW faulted on denotes a disk record, that record must be copied into a core block. The system searches the CMAP for a free entry. If it finds one, it copies the address of the disk record into the free entry; it copies the absolute address of the faulting PTW into the free entry; it copies the address of the associated core block into the PTW; and it threads the CMAP entry to a circular list of used CMAP entries.

If no CMAP entry could be found, a page must be thrown out of core to free a core block. The system walks around the list of used CMAP entries. From each CMAP entry, it follows the binding to the PTW that uses the associated core block. If the "used" flag is on in that PTW, the system turns it off. If it is off, meaning that the page has not been used since the system last revolved around the circular list of used CMAP entries, the system frees the CMAP entry. To do so, it looks at the "modified" flag in the PTW. If it is off, the system restores the PTW to its quiescent state by copying into it the disk address that was earlier saved in the CMAP entry associated with the core block denoted by the PTW. It then frees the CMAP entry. If the "modified" flag is on, the system first copies the core block it is freeing back into the disk record home of the page it is removing from core, and it restores the PTW to its quiescent state. It then frees the CMAP entry.

j. Summary.

Figure 4.1 summarizes the data bases involved in the operation of NSS and their interconnections. Boxes stand for data bases or parts of data bases. Since these boxes will later be identified with abstract types, we might as well think of them now as denoting various kinds of objects. Every arc

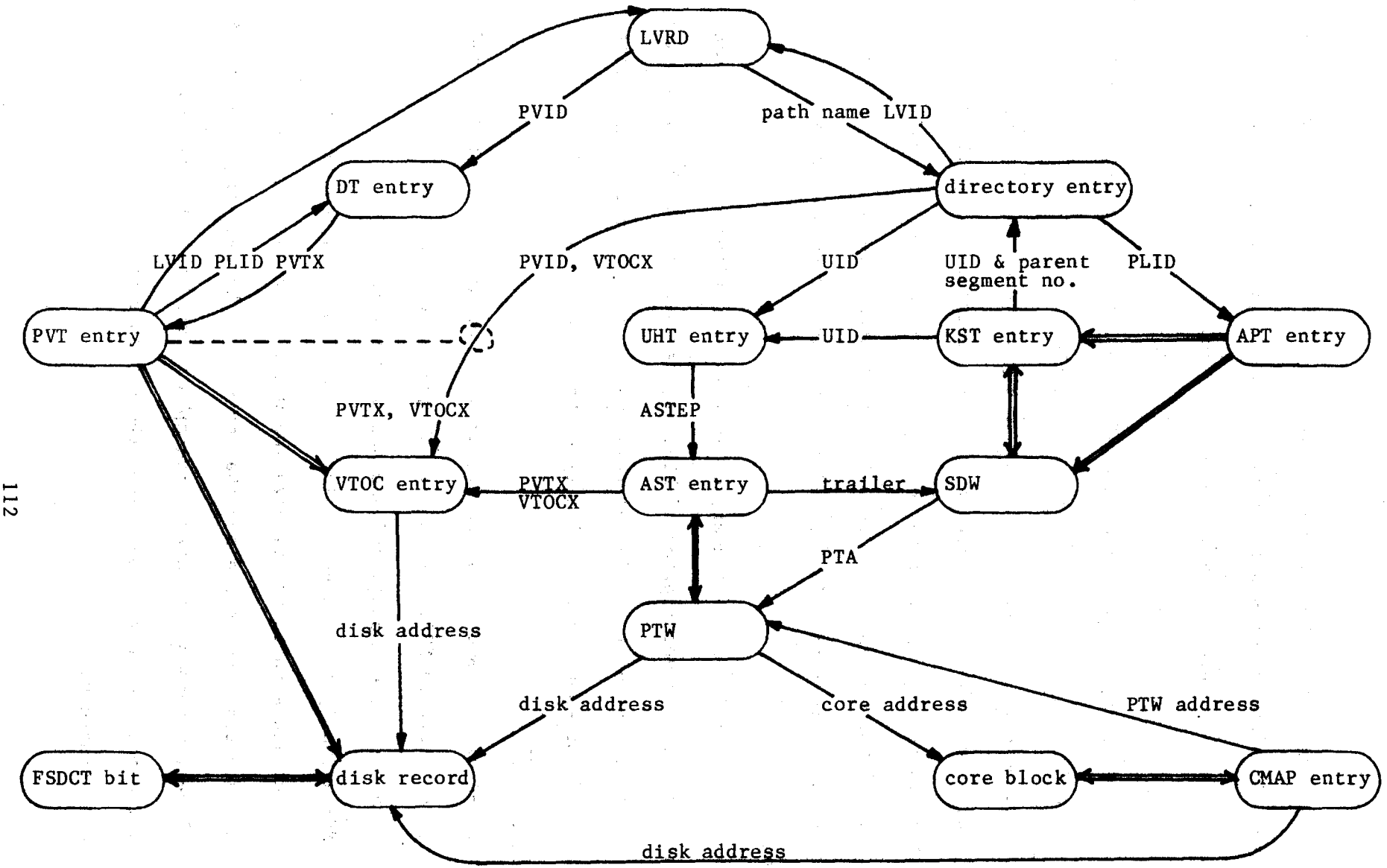


Figure 4.1: NSS data bases and interconnections.

indicates a binding between two objects of different kinds. Single lines indicate explicit bindings or stored denotations. For instance, a SDW denotes a page table. Double lines indicate implicit bindings or correspondances. For instance, a PVT entry corresponds to a collection of disk records and VTOC entries in the sense that it describes a configured volume that is composed of these disk records and VTOC entries. The distinction between single and double lines is purely informative. It has no significance for the later use of the figure. The dashed line deserves a special comment. A (PVID, VTOCX) in a directory entry binds that entry to a VTOC entry only on the condition that the physical volume denoted by the PVID is mounted, i.e. only if the PVT maps the PVID into some PVTX. The dashed line stands for this condition on the binding.

While figure 4.1 is only a summary of what was described in this section, adds no knowledge to the description of NSS, and may seem of little importance now, we will see in the next section of this chapter the interest it presents for deriving a first approximation of a type structure graph for MSS.

Organization of NSS.

The purpose of this section is to try to see a modular decomposition and a dependency structure in NSS, i.e. to try to group the procedures of NSS around the data bases they manage and see how they interact.

Since NSS actually is neither modular nor structured, we cannot see a perfectly modular and structured organization where there is none. The organization discussed here is the one that best fits the real organization of NSS. It will be the purpose of later paragraphs to show how the real NSS differs from the model we try to see here, thereby causing violations of modularity and structure.

volume control	LVRD, DT
directory control	DIRECTORIES
address space control	KSTS
segment control	PVT, AST HASH TABLE VTOCS, AST, DSGS
page control	PTWS, FSDCT, CMAP
traffic control	APT

Figure 4.2: best fit model of the organization of NSS.

The model that best fits the organization of NSS is given in figure 4.2. Every line describes one module. For every module, we have indicated a name and the list of data bases it manages. Every module depends on the modules below it. The partially ordered dependency structure is thus a total ordering in the case of NSS.

Volume control manages the LVRD of all logical volumes and the DT. It implements the operations for creating, deleting, mounting and demounting logical and physical volumes. It also supports the operations for controlling the use of master directories, although the management of these directories is obviously done by directory control. Volume control depends on directory control because LVRD files and the DT are segments stored in the file system and the concept of a master directory requires the existence of the file system.

Directory control manages all directories. It implements the operations for creating and deleting directory entries, and for managing their names and their access control lists. It depends on address space control because directories must be initiated by every user in his own process in order for the system to be able to address them on the user's behalf.

Address space control manages KSTs. It implements the operations for initiating and terminating segments and for handling segment faults and bound faults. It depends on segment control because handling segment and bound faults requires activating and connecting segments.

Segment control manages the AST and the VTOCs, the PVT, and the DSGs. It implements the operations for creating, deleting, activating, deactivating, connecting and disconnecting segments. It also controls the allocation of disk drives by managing the PVT. It depends on page control because most of

its code resides in paged segments.

Page control manages page tables and individual PTWs, the FSDCT and the CMAP. It implements the page allocation, the page fault and the page removal algorithms. It depends on traffic control because one cannot afford to keep a processor idle, waiting in a loop for an I/O operation to complete on a page. Thus, page control depends on traffic control to multiplex processors among ready processes while other processes wait for I/O on a page to complete.

Finally, traffic control manages the APT. It supports the creation, deletion and scheduling of processes and the multiplexing of processors.

Modularity violations in NSS.

Modularity violations occur when a procedure in one module references a data base in another module. Such violations are innumerable in NSS. It is impossible to mention them all here and in addition, it would be uninteresting to mention them all as most of these violations are benign and could easily be fixed.

Our purpose here is to pick two illustrative examples of violations of modularity and to discuss them with respect to the understandability of the system.

Consider the case of the AST. The AST as a whole is managed by the segment control module. Yet, page tables are stored in the AST and are managed by page control. Thus, we have a case of two modules managing one data base. The problem would not be too bad if each module managed only a distinct area of the AST. In essence, they would be managing distinct data bases. Unfortunately, this is not even true. Segment control never touches PTWs, but page control does touch AST information that belongs in the segment control module. In particular, for every segment, page control maintains a "file-modified" switch, a "file-map-modified" switch and a running total of

the number of pages that are currently in core. These pieces of information are used by segment control to deactivate segments. The "modified" switches are used by segment control to decide whether to update the DTM and the file map of a segment in its VTOC entry when it deactivates it. The number of pages in core is used by segment control as an approximate measure of the segment usage rate. This measure is used to select for deactivation the segment with the smallest number of pages in core so as to minimize the paging activity that is required to deactivate a segment. The sharing of the AST complicates the understanding and the verification of NSS because there is an implicit interaction between page control and segment control over data items they share access to. Also, the primary function of page control is to manage PTWs but it cannot ignore totally the fact that pages are logically grouped into entities called segments at a higher level. As a result, the interface between segment control and page control is very hard to define completely and correctly.

Second, we would like to mention the case of one procedure whose function is to translate almost any kind of segment identifier into any other kind and to return any segment attribute given any kind of segment identifier. Such a procedure really cuts across all modules of NSS. It knows about the formats of directories, KSTs, DSGs and the AST. It is not a functional abstraction that is usable at different levels. It is a cluster of functions belonging in different levels and glued together in one procedure. Understanding such a procedure is extremely awkward as it requires a good deal of knowledge about several levels of NSS.

Structure violations in NSS.

In the rest of this section, we will examine violations of the partial ordering of the dependency structure in NSS. There are only a few instances

of such violations. Therefore we will study them all. We study them all because they all constitute major problem areas in NSS and we want to analyze them to focus on their solution in MSS.

a. Process loading.

As mentioned earlier, page control depends on traffic control to multiplex processors among processes while waiting for I/O operations to complete on pages. Unfortunately, traffic control also depends on page control to load processes. The address space of a process, as materialized by its DSG, contains SDWs not only for segments the user has initiated and referenced, but in addition for all system programs, which are thus in every user address space. (Such programs are of course not accessible in the user protection environment.) In order for a process to run at all, that page of the DSG which contains the SDWs for system programs must be wired in core. Otherwise, the process could not handle its page faults and bring in core any page it references, including the DSG page, since the programs required to handle page faults are system programs described by SDWs in the DSG page itself. Thus, prior to switching a processor to a new process, the traffic control module (executed in the old process) must invoke the page control module to "load" (read in and wire the DSG page of) the new process. This causes a first upward dependency violating the structure of figure 4.2. It is due to an upward call. It is a manifestation of a dependency loop encountered in many systems between the virtual memory mechanism and the processor multiplexing mechanism, and analyzed in [Saxena76] and in [Reed76].

b. Page fault handling.

When a process references a page of a segment that is not in core, a flag in the PTW causes a page fault. As a result, the state of the processor is saved to record what caused the fault. The processor abandons the user

computation and it traps -- within the user process -- to the page control module of which the function is to bring in the page. Unfortunately, to do so, page control is dependent upon address space control (and segment control by transitivity).

Two processes cannot be in page control at the same time. This is to preserve the consistency of PTWs as a whole. Synchronization is obtained by using a global page control lock. Every process must test and set that lock before entering page control code. Unfortunately, there is always a time window between the moment a process takes a page fault and the moment it sets the page control lock. Even if the page control lock is not set when a process P takes a page fault, it may be set by some other process Q by the time P gets a chance to test it. In some rare but possible cases, the activity of Q in page control may precisely be pertinent to the PTW that P faulted on. In particular, Q may have entered page control to deactivate or simply delete the whole page table containing the PTW that P faulted on. As soon as Q leaves page control, P may set the page control lock and enter page control. However, to determine what PTW originally caused the fault, P cannot use the absolute address of the PTW that may have been saved in the processor state. That absolute address may now denote a PTW in the page table of another segment or simply a free PTW. Thus, to make sure it finds the correct PTW, page control must repeat the whole address translation operation that was originally performed by the processor before it took the fault. This includes interpreting the segment number to get to the SDW and from there to the page table. (Of course, if the segment has indeed been deleted or deactivated, the SDW will be disconnected. At that point, page control must quit and restore the processor state, which will cause P to retry the operation and this time take a segment fault.) The upward dependency of page control on address space

control resides in the fact that page control relies on the meaning of segment numbers and SDWs to get at a faulting PTW. Yet, page control operates at a very low level, should not know about segment numbers and should not use SDWs as they are stored in the paged DSG, which makes possible the accumulation of page faults. This upward dependency corresponds to a violation of the information level rule mentioned earlier. Debugging has of course shown that NSS works, but understanding and verifying the system are more complex due to the above violation.

c. OOPV conditions.

The next upward dependency is due to OOPV conditions mentioned earlier. When a segment needs to be grown but the physical volume it is on is full, the segment must be moved to another physical volume prior to being grown. The overall mechanism for doing so follows. When a user process touches a zero (null) page of a segment, it causes a page fault. Page control is invoked to grow the segment. If it discovers an OOPV condition, it posts it in the AST entry of the segment (this is another violation of modularity) and it invokes segment control to disconnect the segment (this constitutes a first upward dependency). Page control then restores the processor state. The process retries referencing the segment and this time takes a segment fault since it is disconnected. Address space control is entered to process the segment fault and invokes segment control to reconnect the SDW. Segment control eventually discovers that an OOPV condition was posted by page control. It follows the circular list of threaded PVT entries from the original home configured volume to the currently least full configured volume within the same logical volume and proceeds to move the segment from the original home to the new one. Finally, the move must be reflected at directory control level where the (PVID, VTOCX) of the segment must be changed. To do so, segment

control reaches up and writes into the directory entry describing the moved segment, violating both modularity and dependency. Dependency is violated in the following way. In order to write into the directory, segment control obtains the segment number of the directory and the UID of the appropriate directory entry from the KST. In doing so, it relies on address space control for having stored the right parent directory segment number and the right entry UID at the right place. Furthermore, a (directory segment number, UID) pair denotes a directory entry and should be meaningful only at directory control level. It can be stored at lower levels, but it should remain unused because it denotes a directory control data base that should, from a type extension point of view, remain unknown to lower modules according to the information level rule. By interpreting the pair, segment control becomes dependent on directory control as it does not know (or should not know) anything about a data base like a directory that belongs in a higher level module. Notice that a low level module touching a high level data base results in a violation of modularity on top of which there is a violation of dependency structure due to violating the information level rule. This was the case both in the handling of page faults and of OOPV conditions.

d. Access recalculation on segment faults.

The main role of the KST is to bind segment numbers to directory entries so that, on a segment fault or a bound fault, one can go to the directory to recalculate access or reactivate the segment, or to the AST/VTOC to access its CSL. On a segment fault, address space control uses the (parent segment number, UID) pair in the KST entry to access the directory entry, obtain the (PVID,VTOCX) and compute the access mode from the access control list. Since reading an access control list and extracting from it the desired information for one process is an expensive operation, the following mechanism is used.

The KST is used as a sort of software cache for the access information residing in directory entries. The access information for one process to one segment resides in the access control list of the directory entry describing the segment and is stored in a more accessible and readable form in the corresponding KST entry. In addition, both the directory entry and the KST entry contain a date-and-time-entry-was-last-modified (DTEM). On a segment fault, address space control uses the (directory segment number, UID) pair to access the directory entry and read its DTEM. If this DTEM is more recent than the DTEM in the KST entry, access must effectively be recomputed from the access control list as it may have changed. (The DTEM is also updated in the KST entry). Otherwise, the access information in the KST entry is still up to date and can readily be used without recomputation. Address space control depends on directory control again because of a violation of the information level rule on top of a violation of modularity. Address space control should not touch directories and should moreover never use (interpret) a (parent segment number, UID) pair because, even though it knows about segment numbers, it should in principle not know what directories are and should not assume anything about their usage or their management (format).

e. The quota problem.

The implementation of the quota mechanism in NSS is the biggest source of modularity and dependency violations. The quota mechanism involves every module from page control to directory control. In short, the cause of the problem is the following. Quota cells are organized into a hierarchy that is defined at directory control level; they are stored in the VTOCs and the AST at segment control level; they are managed at page control level; and processing page faults that require growing some segment requires address space control level information (SDWs) as explained earlier. Thus, page

control depends on address space control to interpret SDWs (see earlier). Page control also depends on segment control because that is where the quota cells it manages are stored. And segment control depends on directory control because it cannot ignore the quota cell hierarchy, which is in fact defined at directory control level.

To understand the above implementation, one must consider the CU operation. The CU operation is invoked in two cases. First, when a segment is deactivated or truncated (on a user request), the pages that (may) have become null are deleted. Second, when a user references a null PTW, a page is created. In the above two situations the CRU of the segment changes and therefore the CU operation must be applied to the quota cells of all superior directories up to and including the lowest quota directory. Unfortunately, the hardware that supports Multics does not distinguish normal page faults from page faults on null PTWs (further referred to as quota faults). Thus, page control is entered on a quota fault and page control is responsible for the CU operation. Since two processes must not update a quota cell at the same time, quota cells must be protected by some lock. Since the CU operation is performed under the protection of the page control lock, all other quota operations must be performed under the protection of the page control lock as well. This is the argument that was used to justify why page control actually manages quota cells.

The next question concerns the storage of quota cells. In the original Multics implementation, quota cells were stored in directories themselves since each directory has its own quota cell. However, this implementation was deemed inefficient because updating a potentially large set of quota cells on a CU operation was likely to cause a flurry of page faults on directories, which is undesirable while a process is already processing a page fault. In

NSS, it was decided to store a quota cell in the VTOC entry of its directory rather than in the directory itself. And in order to avoid delays due to I/O on VTOC entries, it was decided to keep active any directory that has at least one son active and to copy the quota cell of an active directory into the AST entry for the directory. (For this and other reasons, any directory with active sons was also kept active in the original Multics design so that NSS did not change that aspect of the implementation of the Multics virtual memory mechanism.) Thus, when a quota fault is taken on some segment, the quota cells of all superior directories are guaranteed to be in core and readily accessible. Since segment control must not arbitrarily deactivate directories, every AST entry describing a directory must contain a count of the number of segments below that directory that are active. Also, to allow page control to walk up the quota cell hierarchy on a CU operation, the AST entry of any segment/directory is threaded to the AST entry for its parent. Consequently, page control depends on segment control because it uses AST entry threads that are supplied by segment control and manages quota information stored in segment control data bases.

Finally, segment control depends on directory control because, for various reasons that will not be described here as they lack interest, it may at times operate on inactive quota cells in VTOC entries. Since these inactive quota cells are not threaded into a hierarchy as are the active quota cells in the AST, segment control invokes directory to find out about the structure of the directory hierarchy.

In addition to being hard to understand, the above design is inefficient in that it requires many directories to remain active, even though they are not used, when all that needs to be accessible is their quota cell, which represents only a few words out of each AST entry.

f. Program, address space and interpreter dependencies.

Although NSS is not based on type extension, the programs, the address space and the interpreter of a module should still be implemented in terms of mechanisms defined at or below the level of the module. Yet, NSS contains innumerable instances of dependency structure violations resulting from program, address space and interpreter dependencies.

To name only one such instance, consider the case of the page control module. The page control module contains certain programs stored into segments that for obvious reasons need to be permanently active and have all their pages wired in core to avoid infinitely recursive page faults. Unfortunately, there does not exist an abstraction equivalent to a permanently active, wired down segment. Thus, page control uses regular segments and assumes that segment control will not deactivate or otherwise damage the PTWs of these segments. Also, the page control module runs in the user address space as materialized by a DSG. That DSG should also be a permanently active, wired down segment. Short of having such objects, the DSG is implemented by a paged segment. Thus, again, page control trusts segment control not to deactivate or otherwise damage the DSG it uses to define its address space. Finally, the page control module depends on processes implemented by the traffic control module to interpret its code, while the traffic control module depends on the page control module for loading and unloading (paging in and out the state of) processes it schedules.

4. Design of MSS.

The objective of this section is to discuss the design of MSS.

In a first step, we will try to visualize data abstractions in NSS and after briefly discussing the problems associated with these abstractions, we

will slightly modify them to yield the type structure graph of MSS. The transition from NSS to MSS was a painfully long process in reality, which we cannot afford to retrace completely here. Thus, we will not say in detail how and why we chose to modify the fictitious type structure of NSS the way we did to arrive at the real type structure graph of MSS. However, by giving a brief summary of the transition, we hope to communicate to the reader the basic operations involved in designing a well-organized system that is based on type extension and we hope to convince the reader that MSS is close enough to NSS that it could be implemented in practice.

In a second step, we will build the complete dependency graph of MSS. To do so, as recommended in chapter III, we will take the type structure graph of MSS and add to it dependency arcs corresponding to map, program, address space and interpreter dependencies for each type manager. We will see that the graph so constructed is partially ordered.

In a last step, we will look back at the problem areas encountered in the organization of NSS and see how they are no longer problem areas in MSS. Considering the problem areas last departs from the way we constructed MSS in reality. In designing MSS, we considered the problems first and in trying to solve them, we progressively arrived at the type structure graph and the dependency graph to be presented at the beginning of this section. We have decided to present the design of MSS first and to discuss the problems it solves then for two reasons. First, some readers might not be interested in the specific system design problems that MSS solves, in which case they may skip the end of this section without loss of continuity. Second, talking about the specific problems first and then discussing the organization would present the risk of distracting the reader from the importance of the partially ordered structure of modules.

Development of MSS type structure.

In order to develop the type structure graph for MSS, we want to look at NSS from a type extension point of view to see if we can distinguish abstract types in it. Of course, such types cannot be formally stuctured and are probably not well-defined. But they may provide us with a first cut idea of a type structure graph for MSS. It will then be possible to formalize those types and to improve their organization to derive a well formed type structure graph for MSS.

The basic process for evolving NSS towards a system based on type extension consists of regarding its data bases as abstract objects and their interconnections as component relationships. We will first associate every kind of data base in NSS with an abstract type and we will define informally the semantics of every abstract type. We will then examine the interconnections between the various kinds of data bases in NSS to see how the different abstract types are related, i.e. what the type structure graph of NSS is. We will finally discuss the structure violations found in that type structure graph and indicate what can be done about them to arrive at a well structured graph for MSS.

a. Choice of abstractions.

In this section, we associate every kind of data base with an abstract type and we define informally the semantics and the "raison d'être" of that type. Unless otherwise stated, these semantics will be preserved in MSS. We are not concerned here about the relation between the various abstract types.

Logical volumes represent what the user sees as single (logical) disk packs and associate these atomic (logical) packs with sets of (physical) disk packs and with master directories controlling the use of space on these disk packs. A logical volume is associated with a LVRD file. A logical volume is

named by the path name of its LVRD. It is also named by a LVID tag. However, this tag only identifies the logical volume. It is insufficient to retrieve its LVRD and operate on it. Logical volumes are base level, C/D objects. Logical volumes can be created, deleted, mounted and demounted. In addition, it is possible to define master directories for logical volumes.

Physical volumes represent the physical disk packs that compose the logical volume objects defined above. A physical volume is associated with an entry of the DT. A physical volume is named by its PVID, which, thanks to the DT hash table, denotes the associated entry in the DT. Physical volumes are C/D objects. (There can be very many entries in the DT.) They can be created, deleted, mounted and demounted. The operations defined on physical volumes are the operations used to implement the operations defined on logical volumes for each of the physical volumes that compose them.

Configured volumes represent disk drives. The allocation of configured volumes to physical volumes is intended to model the allocation of disk drives to mount disk packs. A configured volume is associated with an entry of the PVT. A configured volume is named by its PVTX. Configured volumes are A/F objects. A configured volume can be allocated to mount a physical volume and deallocated when the corresponding physical volume is demounted. In addition, it is possible to search all allocated configured volumes, i.e. all mounted disk packs, for one that bears a given PVID. This operation corresponds to searching the PVT, as explained earlier.

Directories are the places where users catalog files. Directories are the nodes of the file system. A directory is named by its path name. Directories are base level C/D objects. The operations defined on directories are those defined at the interface of the directory control module in NSS (e.g., create, delete, create entry, delete entry, rename entry, set access

control list, move quota, etc...). If a directory is initiated in a process, that process can also name the directory by segment number.

User process objects are the internal representation of users and carry out inside the system the intentions of the users. The user process concept is associated with an entry of the APT (later renamed the User Process Table (UPT) in MSS). A user process is named by the PLID of its APT entry. User processes are base level C/D objects. (The system supports a limited number of processes at a time.) User processes are subject to synchronization operations of which one purpose is to multiplex the processors of the system.

A known segment is a slot in the address space of a user process. Every user process has a finite set of known segments available in his address space. The user can request the allocation and the deallocation of known segments but he cannot read or write known segments. Such operations are not defined on this type of abstraction. The known segment concept is associated with a KST entry. A known segment is named by the segment number denoting its KST entry. (Note that a segment number must be interpreted with respect to the KST of a specific process. That specific process is implicitly defined as being the one interpreting the segment number.) Known segments are base level A/F objects. Known segments are allocated/freed in response to a user's request to initiate/terminate files. Segment faults and bound faults are events defined on known segments. (So are quota faults in MSS, as will be seen later.)

The segment abstraction stands for a logical collection of fixed size physical information containers. A segment is associated with a UHT entry. A segment is named by a UID. Segments are C/D objects. Segments can be created, deleted, activated and deactivated. Segments implement the segments and directories defined in the file system.

A connected segment is also a slot in the address space of a user process. Every user process has as many connected segments as he has known segments and they are in a one-to-one correspondance. The allocation of connected segments is implicitly controlled by the allocation of known segments because of this one-to-one correspondance. Thus, the user cannot explicitly request the allocation or the deallocation of a connected segment. However, he can read or write connected segments. The connected segment concept is associated with a SDW. A connected segment is named by the segment number of its SDW. (Note that a segment number for a connected segment must be interpreted with respect to the DSG of a specific process. This specific process is defined as being the one interpreting the segment number.) Connected segments are base level A/F objects. In addition to read, write and execute, two operations defined on connected segments are connection and disconnection. They can be invoked only by the system.

The passive segment abstraction stands for the disk image of a segment. A passive segment is associated with a VTOC entry. A passive segment is named by a (UID, PVTX, VTOCX) triple denoting its VTOC entry. Notice that the (PVTX, VTOCX) pair is sufficient to retrieve the VTOC entry but not to identify it as corresponding to the right passive segment. The importance of this remark will become clear soon. Passive segments are A/F objects. One can read or write their VTOC entry.

The active segment abstraction stands for the image of a segment of which the page table is in core. An active segment is associated with an AST entry. An active segment is named by its ASTEP. Active segments are A/F objects. One can read or write their associated AST entry and cause their CSL, MSL, CRU, DTM and DTU attributes to be updated. Notice that in MSS some active segments are not the image of any segment. They are stand-alone objects used

by the system. Such active segments are not components or subcomponents of any segment, known segment or directory. Segment, bound and quota faults are not defined on them. They cannot be deactivated (deallocated) because the segment manager, which multiplexes active segments among passive segment and implements the segment deactivation algorithm, is unaware of them and has no ASTEP to name them.

The page frame abstraction stands for a block of contiguous words of storage. A page frame is associated with a PTW. A page frame is named by the absolute address of its PTW. Page frames are A/F objects. One can read or write the PTW of a page frame. Page faults are events defined on page frames. We will see in MSS that PTWs are no longer stored in the AST. Instead, they are stored in a separate table called the Page Frame Table (PFT). The PFT is an array of PTWs. The concept of a segment as a logical collection of page frames is totally lost at this level. All the page frame manager knows about is a page table, i.e. a collection of page frames with sequential names (the absolute addresses of their PTWs). It can allocate and free page frames with sequential names. It can inspect several page frames at a time given the name of the first one (PTA) and the number of page frames to be inspected (CSL) but it never knows what segment a page frame is part of.

The concept of a disk record is associated with a block of storage on disk and one bit in the FSDCT. A disk record is named by its disk address. Disk records are bottom level A/F objects. They can be allocated, freed, read and written.

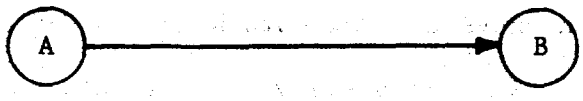
The concept of a core block is associated with a block of storage in core and one entry of the CMAP. A core block is denoted by its core address. Core blocks are bottom level A/F objects. They can be allocated, freed, read and written.

b. Type structure of NSS.

We now turn our attention to the relation between abstractions in NSS. In other words, we want to find out what a type structure graph for NSS would be.

We will say that a data base A is bound to a data base B if A contains a name denoting B. The basic technique for constructing the type structure graph of NSS consists of saying that if a data base A is bound to a data base B then a procedure operating on A may follow the binding from A to B and request that some operation be performed on B. Thus, the abstraction associated with B should be regarded as a component of the abstraction associated with A since operations on A may eventually affect B when the binding is followed to it.

In fact, not all bindings should be viewed as component relationships. Consider a data base A that is bound to a data base B.



The binding between A and B can fulfill one of two mutually exclusive roles. First, it may be there so that the programs managing A may find B and invoke the programs managing B to perform some operation on B. In type extension terms, this means that the management of A may, on occasions, require operating on B. The binding exist to record the name of the specific object (B) that should be operated on as part of operating on A. The type manager of A may use this binding to request the type manager of B to operate on B. In other words, B is a component of A. A binding of this first kind will be called an <u>interpreted</u> binding. Second, the binding may be stored in A by the programs managing A only to record some association between A and B (not a component relationship) but not to ever operate on B. In type extension

terms, either the type manager for A does not care at all about this binding and the binding is stored there on behalf of some other module, or the type manager for A cares about the binding only insofar as it establishes a relation between A and B but not a component relation. This latter case would be indicated by the fact that the type manager for A never needs to invoke the type manager for B to operate on B because it cares about B only as an object somehow related to A but not as a component of A. A binding that does not correspond to a component relationship will be called an <u>uninterpreted</u> binding. Examples of interpreted and uninterpreted bindings in a real system will be discussed soon.

Thus, when considering a binding between two data bases in NSS, we must wonder whether the binding is interpreted or not by the programs managing the type of data base at the origin of the binding.

In order to develop a fictitious type structure graph for NSS, we consider figure 4.1, which shows all the data bases in NSS and summarizes all the bindings between them. We regard every kind of data base represented in that figure as corresponding to one of the abstract data types defined above. And we must carefully review every binding between any two data bases to decide whether it is interpreted or not. The result of this process is shown in figure 4.3. Every box stands for an abstract type and every (solid) arc for a component relationship. We cannot afford to justify here every component relationship in figure 4.3. From the description of the mechanical operation of NSS, the reader may, if he cares, verify that every component relationship indeed corresponds to an interpreted binding. However, we will discuss here the uninterpreted bindings of figure 4.1 that are represented by dashed lines in figure 4.3 and are not component relationships.

First of all, a comment is necessary to interpret the abstraction marked

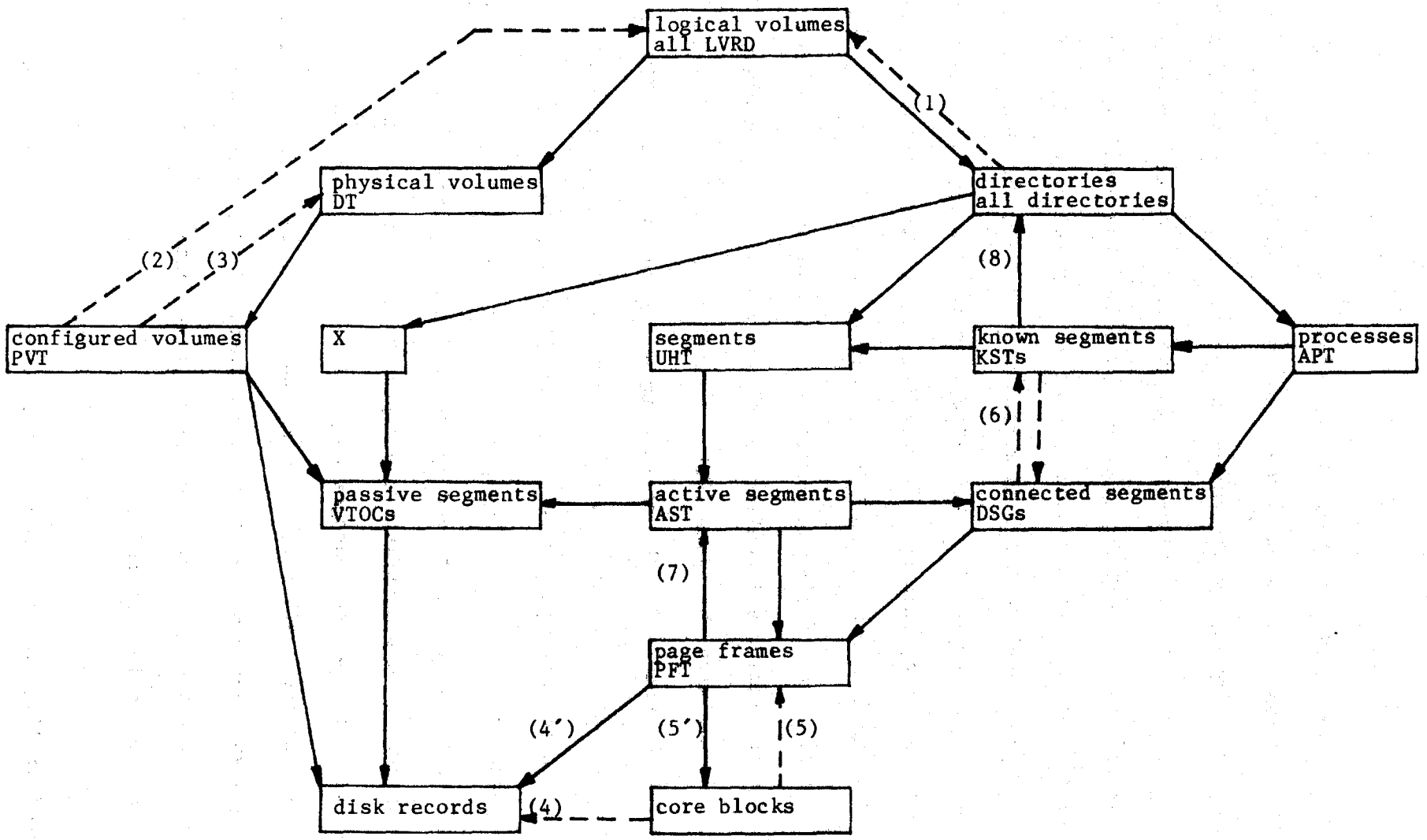


Figure 4.3: fictitious type structure for NSS.

X. This abstraction maps the (PVID, VTOCX) stored in a directory entry to the (PVTX, VTOCX) necessary to access a VTOC entry. In reality, it maps a PVID into a PVTX. This mapping is determined by the DT and duplicated in the PVT. As expressed in figure 4.1, this mapping is conditional on the physical volume with the given PVID being mounted, i.e. on the PVID being bound to some PVTX. Thus, abstraction X implements a mapping that already exists in the PVT. As will be seen in MSS, abstraction X is merged with the segment abstraction. A segment will be named by a (UID, PVID, VTOCX) triple. Every time the segment manager is invoked to operate on a segment denoted by such a triple, it first uses the UID alone in the hope that the segment is active and will be found in the UHT. If this search fails, it invokes a primitive of the configured volume manager to examine all configured volumes and see if any corresponds to the physical volume with the given PVID. If such a configured volume is found, its PVTX is returned. The segment manager then uses the (PVTX, VTOCX) it now has to request operations on the corresponding passive segment. In other words, the segment manager is dependent on the configured volume manager to map a segment into its passive component because the mapping relation is stored in the PVT that is maintained by the configured volume manager. This will later appear as an appropriate map dependency. It is not the responsibility of the segment manager to guarantee against the loss of its objects. Its only responsibility is to guarantee that the integrity of its objects is always preserved even if they are lost. Notice that losses can be recovered from if the higher level mechanisms that cause them care to correct them and retrieve the lost objects.

Now, consider arcs (1), (2) and (3) of figure 4.3. They are all uninterpreted bindings for the same reason; they are never used by any

program to operate on the target objects they denote. Arc (1) is the LVID tag of a directory. As mentioned earlier, it is never used to operate on the logical volume it denotes. It is used as a sort of color tag that the directory manager wants to be common to all segments in a directory. The directory manager does not care about what logical volume it denotes. In fact, it does not care about what logical volumes are. All it knows is that every directory has a color tag and that to create a segment in a directory, it is necessary to match that tag with a similar tag in the PVT entry of the configured volume on which the segment is created. In the PVT, arc (2) stands for this matching tag. Again, it is an uninterpreted binding. In fact, it is the inverse of a component relationship. The configured segment manager does not care about what logical volumes are. It only knows that every configured volume bears a LVID color tag and that all configured volumes with the same tag are threaded in a circular list. The same sort of argument could be made for arc (3), which stands for the PVID tag of a configured volume, yet another sort of color tag used to retrieve passive segments as expressed in the previous paragraph.

One might believe that, in MSS, the directory manager and the segment manager would depend on the meaning of color tags that are really determined at the physical volume and logical volume levels. This is not the case. The directory manager and the segment manager use the tags to group segments into directories and pages into segments respectively. Thus, tags provide a strategy for physically organizing, storing and retrieving information. However, the directory and segment managers do not depend on them to guarantee the integrity of the logical objects they manage because the ultimate item that associates directory entries to segments and to VTOC entries is a UID. UIDs are not sufficient to retrieve segments but they are sufficent to

identify them. Thus, if tags got mixed up somehow by the logical or the physical volume manager, the segment manager would not be able to retrieve its segments. However, it would never damage them accidentally thanks to the ultimate integrity check provided by UIDs. Consequently, the segment manager could signal discrepancies in the associations of color tags it sees with respect to associations it previously saw. But as soon as these discrepancies would be fixed, for instance, by salvaging some LVRD or the DT, the segment manager would recover its full capacity and no segment would have been damaged or lost in the meantime. UIDs essentially constitute a protection mechanism that prevents the segment manager from suffering from malfunctions in the logical or physical volume managers. The semantics of the segment manager guarantee the integrity of all segments but do not guarantee that all segments can always be accessed. This latter property may be imposed and verified only at the level of the logical volume manager.

Now consider arcs (4) and (5). These bindings are used by the page removal algorithm, as explained earlier. Since the purpose of this algorithm is to throw pages out of core when necessary, which requires manipulating the PTWs that represent the pages, the algorithm is part of what should be called the page frame manager. (4) and (5) are stored in CMAP entries by the core block manager only on behalf of the page removal algorithm. The core block manager never uses these bindings to operate on PTWs or on disk records. (4) is only a way to save the address of the home of a page (4′) while the page is in core. (5) is only the inverse of the component relationship (5′).

While arcs (6) clearly establish an identity between segment numbers of KST entries and segment numbers of SDWs, they are not component relationships. While operating on a KST entry with segment number N, the KST management programs never invoke the DSG management programs to operate on the SDW with

number N. Conversely, while operating on an SDW with number N, the DSG management programs never depend on the KST management programs to operate on the KST entry with number N.

We now have a fictitious type structure graph for NSS. It is fictitious in the sense that NSS does not really respect any type extension formalism and the type structure graph is not well structured. The partial ordering of abstract types is violated in two places. First, an AST entry is composed of a collection of PTWs but its is also possible to get to an AST entry (7) from any of its PTWs, to operate on it. This is indeed what the paging mechanism does to maintain for every segment a "file-modified" switch, a "file-map-modified" switch and a number of pages in core, as described earlier. Second, there is a loop of component relationships between directories, processes and known segments.

c. Type structure of MSS.

To construct the type structure graph of MSS by evolving that of NSS, we consider figure 4.3.

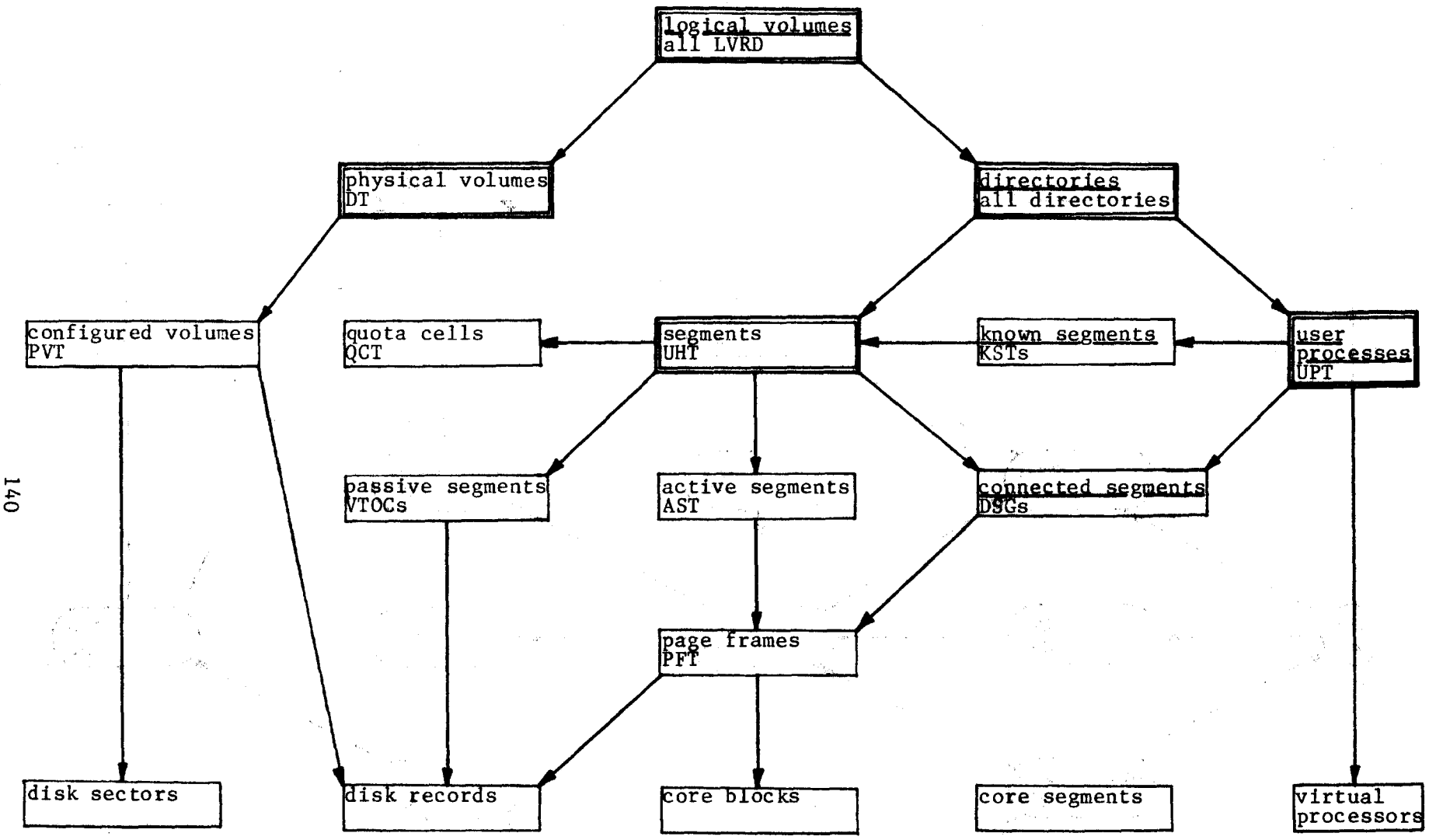
We first consider the dashed lines. Lines (1), (2), (3) and (6) can be discarded as no module ever interprets them as component relationships. Line (5) can be discarded as it is interpreted by the page frame manager, as the inverse of a component relationship represented by line (5′). Line (4) can be discarded as it is interpreted by the page frame manager, as a component relationship already represented by line (4′). We then remove line (7) as it violates partial ordering. This is to say that it is now impossible to go from a PTW to the containing AST entry. In other words, PTWs are now stored in a separate table, the PFT, as announced earlier. We also remove line (8) for the same reason. This is to say that it is now impossible to go from a KST entry to the corresponding directory entry. In other words, the known

segment manager will not be allowed to invoke the directory manager to operate on a directory entry or to extract information from it. The impact of the blunt deletion of the two lines mentioned above will be studied later. Finally, we merge abstraction X with the segment abstraction, as explained earlier. The result of this evolution process is shown in figure 4.4, which shows the complete type structure graph of MSS. The names of the data bases under certain abstract types denote the data base(s) managed within the type managers for the corresponding types. As we proceed, we will see that these data bases really implement the maps of the objects of the corresponding abstract types.

The informal semantics of abstract types defined in NSS are preserved in MSS. Some new abstract types are defined in figure 4.4. Their purpose will be justified and their semantics will be given later, as we examine the problem areas in NSS and take care of them in MSS. The details of the mechanical sequence of operations for handling such events as page faults, quota faults, OOPV conditions, bound faults and segment faults will also be given in the third part of this section on MSS, as we discuss how MSS handles situations that caused problems in NSS. Notice that two component relationships have been slightly moved in figure 4.4. A UHT entry rather than the AST entry it points at is designed to contain the trailer list of an active segment and the (PVTX, VTOCX) denoting the home of the segment. These bindings have been moved from the AST to the UHT because they will yield component dependencies that would conflict with the partial ordering of the dependency graph of MSS if they were not moved. (Specifically, they would conflict with map dependencies to be described later.)

<u>MSS dependency structure.</u>

A basic dependency graph for MSS is given in figure 4.5. This graph



underlined: base level
: C/D
: A/F

Figure 4.4: MSS type structure graph.

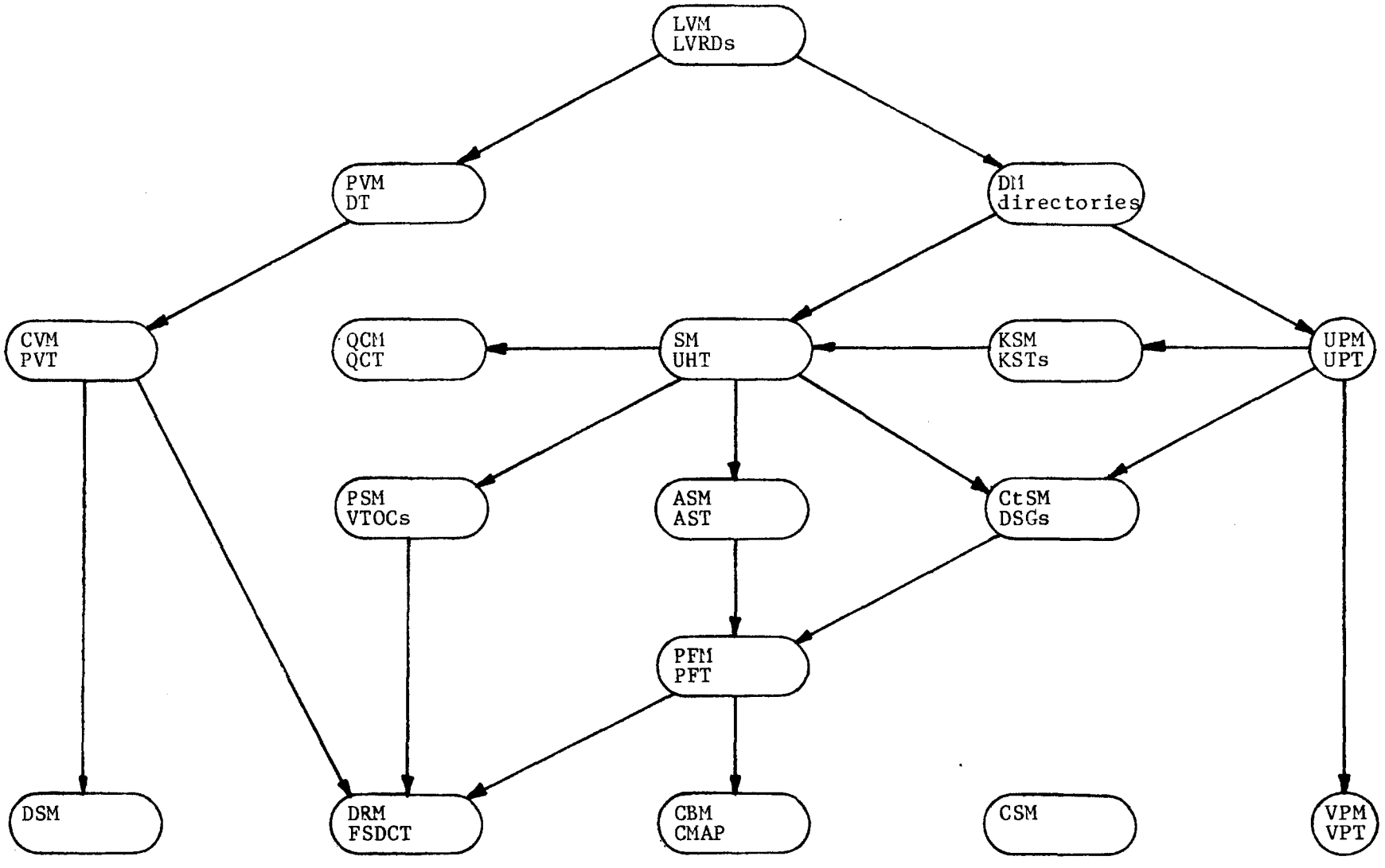


Figure 4.5: MSS component dependencies.

contains only the component dependencies that are derived directly from the type structure of figure 4.4 by drawing one type manager for each abstract type and one component dependency for each component relationship. Each node gives the name of the internal data base(s) of the type manager.

a. Map dependencies.

The dependency graph of figure 4.5 shows only component dependencies. We are now concerned with adding to that graph dependencies due to the implementation of the maps of objects of each abstract type. We must design the system so that none of the map dependencies to be introduced cause a dependency loop with any of the component dependencies that we already have drawn.

The map of a logical volume is in its LVRD, which binds the logical volume to a collection of physical volumes and of master directories. The LVRD is implemented by a segment that is cataloged in some directory. Thus, to implement maps of logical volumes, the logical volume manager is dependent on the directory manager.

The map of a physical volume is stored in a DT entry, which binds the physical volume to a configured volume if it is mounted. The DT is also implemented by a segment that is cataloged in some directory so it can be backed up like all files of the file system by a mechanism (not described here) operating above the virtual memory mechanism, which periodically orders copies of every recently modified file to be made on a tape. Thus, to implement maps of physical volumes, the physical volume manager depends on the directory manager.

Of course, LVRD segments and the DT must be stored on the root logical volume to be on-line at all times. The root logical volume is guaranteed to

be on-line at all times by the configured volume manager. Each configured volume corresponding to a physical volume of the root logical volume is marked with a special "root" tag that is set at system initialization and prevents that volume from being demounted even if the logical volume manager or the physical volume manager accidentally requested that it be demounted.

The map of a configured volume is in an entry of the PVT, which specifies what portion of the configured volume constitutes the disk record pool and what portion constitutes the disk sector pool (see later). The PVT is implemented by a core segment. Thus, the configured volume manager depends on the core segment manager for implementing maps of configured volumes.

A core segment is a new type of segment that is introduced now for the first time but will often be needed later at places where NSS used permanently active, wired down segments. A core segment is an unpaged collection of contiguous words in core. Core segments are core resident information containers. They reside in a portion of core that is determined as the system is initialized and is distinct from the portion of core used by core blocks. Core segments are used exclusively by system modules, as will be seen. They are defined and initialized when the system is brought up. They are never allocated or freed thereafter. They are fixed size information containers. The only operations defined on core segments are READ, WRITE, and EXECUTE. The core segment manager is implemented entirely by hardware operations. Core segments are named by their absolute core address. They can be addressed in hardware by SDWs of which a dedicated bit indicates that the field normally containing a PTA denotes the base of the segment itself rather than the base of its page table.

The map of a directory is the data base that binds the directory to the set of (UID, PVID, VTOCX) triples of the segments described in the directory

and to the set of PLIDs of the users allowed to access these segments. This data base is the segment containing the directory entries. Thus, the directory manager depends on the segment manager for implementing directory maps.

The map of a user process is its UPT entry. A UPT entry occasionally binds a user process to a virtual processor (see later). It also binds the user process to a collection of known segments and to a collection of connected segments. (In fact, it binds the user process to the image of a DBR, i.e. to a DSG. Thus, it indirectly binds the user process to a set of SDWs and to a set of KST entries. A user process is indirectly bound to a KST because every DSG contains a SDW with a conventional, fixed segment number that denotes the KST of the corresponding process. There is no reason to regard DSGs and KSTs as abstractions maintained separately from connected segments and known segments because the management of such tables is concerned solely with management of SDWs and KST entries respectively. In consequence, figure 4.5, which maps user processes directly into known segments and connected segments is a simplified view of the reality but it is absolutely correct: user processes are bound (indirectly and implicitly) to sets of known segments and connected segments.) The UPT is implemented by an active segment. Thus, the user process manager depends on the active segment manager for implementing maps of user processes. An active segment is not cataloged in the file system, which means that it has no secondary storage home and vanishes when the system is brought down. This poses no problem because, when the system is shut down, all user processes are deleted and the UPT is empty anyway.

The map of a known segment is contained in a KST entry, which binds the known segment to a segment UID. (In fact, we will see later that a KST entry

binds the known segment to the (UID, PVID, VTOCX) triple of a segment.) A KST is a connected segment that should never be disconnected because the operation of connecting a segment requires the presence of a connected KST. Thus, the known segment manager depends on the connected segment manager for implementing the maps of known segments.

The map of a connected segment is its SDW. The DSG containing the SDW is also a connected segment that may never be disconnected for the same reasons as the KST. Thus, the connected segment manager depends on itself to implement the maps for connected segments. Although this unquestionably constitutes a dependency loop, this loop does not pose a conceptual problem, as explained below.

In reality, a permanently connected segment should be regarded as an abstract type different from other connected segments on which segment faults can be taken. Then, there would be no dependency of the connected segment manager on itself. Furthermore, the type checking mechanism could distinguish permanently connected segments from other connected segments, which is important since the system will operate correctly only if it can be verified that KSTs and DSGs are never disconnected. While regarding permanently connected segments as an abstract type different from connected segments is the only rigorously correct approach to type extension, we have taken the liberty to depart from such strict formalism for two reasons. First, permanently connected segments are very much like connected segments and by slightly modifying the connected segment manager, it can also support permanently connected segments. Second, there need be only three permanently connected segments per process. Therefore, it is easy to declare and build into the type checking mechanism that three SDWs with identical, fixed segment numbers in each DSG are dedicated to implementing permanently connected

segments. The connected segment manager may be coded to never disconnect these segments even if it were accidentally requested to do so. And the type checking mechanism may recognize the three fixed segment numbers as denoting the equivalent of permanently connected segments. In essence, when a user process is created, the user process manager invokes an initialization primitive of the connected segment manager. This primitive first invokes the active segment manager to allocate and build three active segments with given lengths (CSL). It then manufactures three SDWs for these segments. It stores the three SDWs in one of the three active segments, to become the DSG, at a location corresponding to the three reserved segment numbers. It finally returns an image of the SDW for the DSG to the user process manager. That image binds the process to its DSG and implicitly to its KST and will serve to load the DBR of a processor every time the process is scheduled.

The map of a segment is constituted partly by a UHT entry, which binds the segment to its active component, if any, and partly by information embedded in the PVT, which binds the PVID tag of the segment to a PVTX tag. The UHT is implemented as an active segment. Thus, the segment manager depends on the active segment manager and on the configured volume manager for mapping segments into their components.

The map of a passive segment is its file map in its VTOC entry. (In fact, it is the whole VTOC entry if one considers that the storage attributes of the segment (DTU, DTM, etc...) that reside in the VTOC entry are what we called in chapter II the small components of the segment and are stored together with the map.) A VTOC is implemented by disk sectors. Thus, the passive segment manager depends on the disk sector manager to implement the maps of passive segments.

Although the concept of a disk sector already existed in NSS, it was not

recognized as an independent data type. Disk sectors were managed by the page control module. Every disk is divided into a paging zone and a cataloging zone, as explained earlier. The paging zone is the disk record pool. The cataloging zone, i.e. the VTOC, is the disk sector pool. Disk sectors are containers smaller than disk records. A VTOC entry is implemented by three disk sectors that are contiguous. Since disk sectors are used solely to implement VTOC entries, there is no need to keep track of their A/F status. It is implicitly determined by the A/F status of the corresponding VTOC entry. All the disk sector manager does is READ and WRITE disk sectors and configure them when told to by the configured volume manager. Configuring disk sectors consists only in remembering how many sectors are associated with a given PVTX tag.

The map of an active segment is its AST entry, which contains the small components of the segment and binds it to a collection of page frames (PTA, CSL). The AST must be a core segment because the active segment manager would depend on itself if the AST were an active segment that had to be maintained permanently active.

The map of a page frame is its PTW. PTWs are collected in the PFT, which must be a core segment because the page frame manager may not take a page fault on the table it maintains to handle page faults.

Disk sectors, disk records, core blocks and core segments are bottom level objects and have no map. The <u>quota cell table</u> (QCT) and the <u>virtual processor table</u> (VPT), which serve to store the maps of quota cells and virtual processors respectively, will be discussed later. Let us say for now that the QCT is an active segment and the VPT a core segment.

b. Program dependencies.

Program dependencies are those that make any type manager dependent on

one or more other type managers for storing the procedures and data bases that implement it. By examining the implementation of maps, we have answered the question of the implementation of every data base in figure 4.5 except the CMAP and the FSDCT. Since these tables are used below the level of the paging mechanism, they must be core resident. Thus, they are core segments.

We must now discuss the implementation of the procedures and the working storage for each type manager. In Multics, the working storage is constituted by an Algol-like push-down stack for storing the local variables and the argument lists of pending procedure invocations.

For the implementation of procedures in MSS, one must consider two environments of execution, the core resident environment and the paged, permanently active (paged for short) environment. The core resident environment is implemented entirely by core segments and the paged environment is implemented entirely by active segments. These active segments are in practice permanently active because the segment deactivation algorithm, which operates in the segment manager, can only deactivate (i.e. request the deallocation of) active segments that are components of segments. It does not see and cannot operate on any other active segment because it does not have any ASTEPs to name them.

The core segment manager is implemented entirely by hardware operations. Thus, it is in neither the resident nor the paged environment. It is in a hardware environment below both. The core resident environment comprises the disk sector manager, the disk record manager, the active segment manager, the page frame manager, the core block manager and the virtual processor manager. All other type managers are in the paged environment.

The active segment manager must execute in the paged environment because it would depend on itself if it executed in the other. The page frame

manager, the core block manager and the disk record manager must execute in the resident environment because they implement the mechanism for supporting non-core resident information containers and cannot depend on this mechanism. The disk sector manager executes in the resident environment for efficiency only to avoid taking page faults when READing or WRITing a disk sector. The virtual processor manager executes in the resident environment by construction (see later). All other type managers can safely operate in the paged environment.

Now consider the implementation of stacks. The type managers executing in the core resident environment obviously require a core resident stack. The implication of this statement is that any procedure in the paged environment that calls a procedure in the resident environment must pass its arguments to the called procedure on the core resident stack. This is a practical consequence of the information level rule. Since the resident environment knows only about core segments, it cannot accept arguments that are not in a core segment. This constraint is enforced by the type checking mechanism, which will refuse to compile a type manager that passes non-resident arguments to a resident type manager.

A priori, one resident stack per user process can be used to pile the pending invocations of all resident type managers in that process. This stack must be stored in a core segment that is of a size sufficient to hold the deepest pile of pending invocations that can possibly occur along any computation path through the resident type managers. NSS embodies a similar design constraint in that the resident part of the system (traffic control and page control) must use a special stack that is stored in a core resident (actually permanently active and non-pageable) segment. In fact, in NSS, there need not be one such stack for each process that can execute resident

code because, by design, only one process at a time can execute in traffic control and one in page control. In practice, there is one such stack for each physical processor. Even though only two processors may be executing resident code (one in traffic control and one in page control) at the same time, every processor needs its own stack because, while two processors are executing resident code, all others might be waiting (e.g. on the traffic control lock) in an idle loop to start executing that resident code and while doing so need to store the arguments that will be passed to the resident code as soon as it is entered. Thus, we recommend an analogous design for MSS, where traffic control is to be read as the virtual processor manager (see further) and page control is to be read as the other resident type managers.

The implementation of a stack for the paged environment is somewhat more tricky. Whatever the type of that stack is, every argument passed from the user environment into the paged environment must be stored into the stack for the paged environment. A type manager (e.g., the connected segment manager) of the paged environment cannot access an argument stored in a random segment because that segment may be of a type defined at a higher level (e.g. known segment). This is again an information level problem. But this time, the user who passes the argument is not subject to the compile-time type checking mechanism that the system programs are subject to. Thus, it may in effect pass an argument stored in a random segment. Consequently, at the user interface but inside the paged (type checked) environment, there must exist conversion routines that accept user supplied arguments stored in any type of segment and copy them into the paged stack. The need for such routines was not felt in the early days of Multics and many system problems (including crashes) resulted from the absence of such routines.

A priori, one would be tempted to implement one paged stack per user

process with an active segment having a number of pages (CSL) sufficient to hold the deepest possible pile of invocations in that process along any computation path through the paged environment. In fact, it is necessary that the per process stack be a (permanently) active segment. However, this is not sufficient. The stack under concern is used, among other cases, while handling segment faults. Thus, a process must never take a segment fault on its paged stack to avoid infinitely recursive segment faults. In other words, the paged stack must be implemented by a permanently connected segment (that is itself implemented by a permanently active segment). With the DSG and the KST, the paged stack of a process is the third and last permanently connected segment needed in every process. The obvious question to ask now is: why do we explicitly state that the stack is a permanently connected segment while we do not explicitly state that the procedures of the known segment manager, which handles segment faults, are permanently connected? The answer to this question will become clear in the next paragraphs. In short, two kinds of address spaces are defined: per process address spaces and per processor address spaces. Since all user processes share the same stored procedures to implement the paged type managers, SDWs for such procedures may be stored in per processor address spaces, which contain only permanently connected SDWs, as will be seen. Thus, the procedures of all paged type managers are de facto permanently connected. However, the working storage of the paged environment cannot be shared by all user processes like the procedures are. Every process may be doing its own computation in the paged environment at a given time. Thus, the SDWs for the paged stacks may not be stored in per processor address spaces because a per processor address space is shared by all the processes that at one time or another get to execute on the corresponding processor. SDWs for paged stacks must therefore be stored in per process address spaces,

i.e. in what we have called so far DSGs. We explicitly insist that SDWs for paged stacks be permanently connected because SDWs in user DSGs normally are not.

c. Address space dependencies.

This fourth kind of dependency is due to the implementation of the address space of a type manager. In NSS, the address space is implemented on a per process basis. The address space of a user process is defined by a DSG. Because DSGs are permanently connected segments in MSS, at least all type managers below the connected segment manager cannot use a DSG as the realization of their address space. This is example of a situation where the dependency graph suggests that the NSS design is inadequate and another abstraction (namely core segments) must be used to implement the address space that part of the virtual memory mechanism will be executing in.

To achieve a proper design for MSS, we recommend the following hardware architecture. Rather than using only one DBR and consequently one DSG while it executes, a processor should use two DBRs, DBR0 and DBR1, defining essentially two domains of execution. Any segment number below a fixed boundary is interpreted by the hardware as an index denoting a SDW in the DSG denoted by DBR0. The value of the fixed boundary is subtracted from any segment number superior or equal to it and the result is used as an index denoting a SDW in the DSG denoted by DBR1. The DBR1 is similar to the DBR in NSS. It denotes a DSG that is a permanently connected segment and contains the map of all connected segments. It is used to address all segments that may be connected, the paged stack, the KST and the DSG itself within the running process. The DBR0 is not a per process DBR but a per processor DBR that denotes a <u>core descriptor segment</u> (CDSG) implemented by a core segment. The CDSG is a repository for the SDWs of all the active and core segments

implementing all system procedures, data bases and maps, except the paged stack, the KST and the DSG of every process, precisely because there is one of each per process and their SDWs cannot be stored in per processor CDSGs. One CDSG is defined and initialized for each processor. It contains the SDW for the core stack of the corresponding processor. The use of a per processor CDSG eliminates the need to load and unload processes in MSS. Indeed, SDWs for the core segments implementing the active segment manager and lower level type managers are in the CDSG that is a core resident segment and never needs to be loaded and unloaded.

d. Interpreter dependencies.

Every type manager of the virtual memory mechanism has to depend on an abstract code interpreter to execute its procedures. On the other hand, modules implementing abstract code interpreters (virtual processor type managers) must depend on some virtual memory type managers to implement their maps, programs and address spaces. One must therefore pay attention not to introduce any dependency loop in the system by making a type manager P implementing a abstract code interpreter depend on a type manager M implementing an abstract information container and at the same time making M dependent on P to implement its code interpreter.

This dependency loop between the virtual memory and virtual processor mechanisms is present in most existing systems in one form or another. In NSS, it is manifested among other instances by the loop between traffic control and page control, where page control depends on traffic control to do processor multiplexing while traffic control depends on page control to load and unload processes.

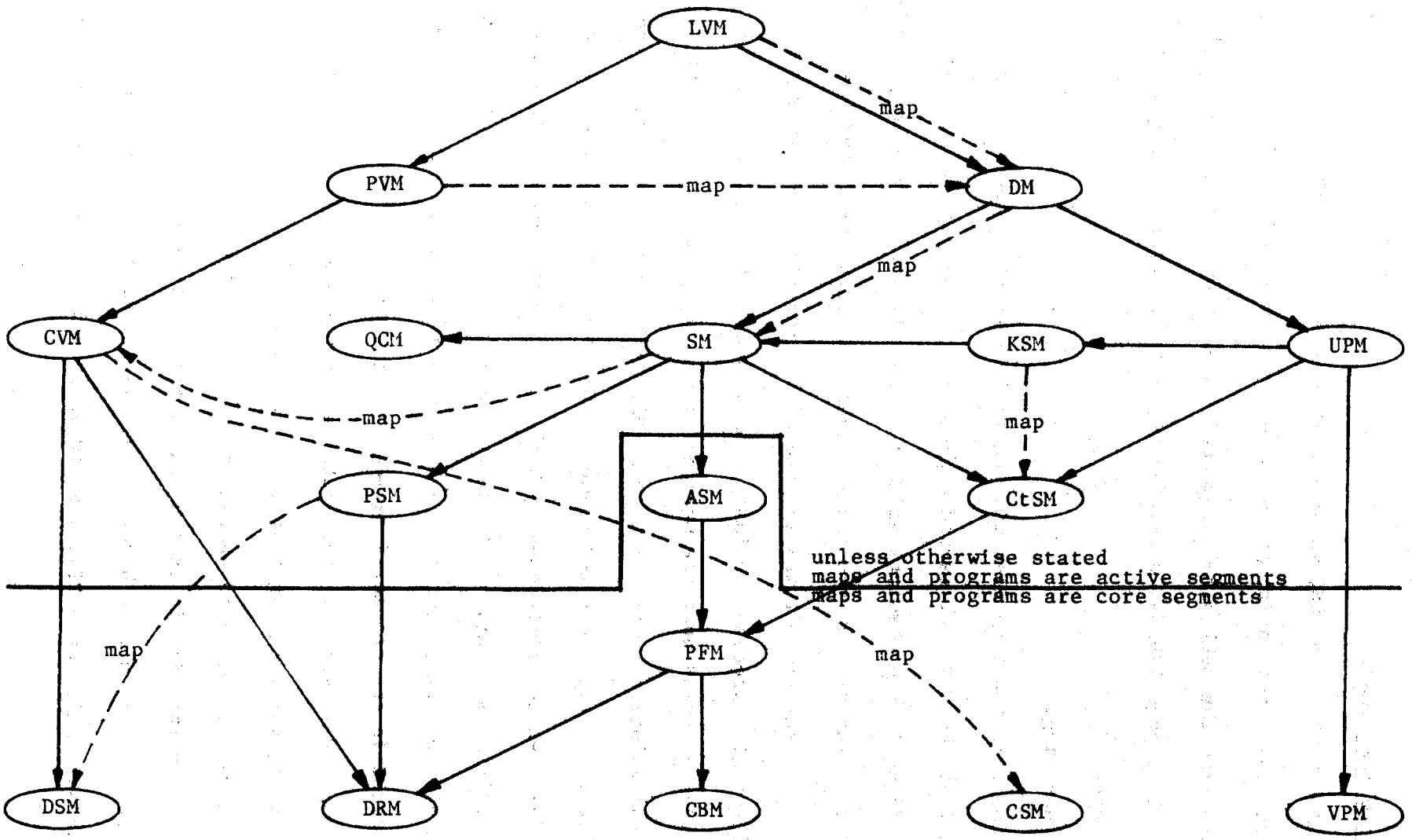
Solutions are proposed to solve the problem of mutual dependency between the virtual memory and virtual processor mechanisms both in [Saxena76] and in

[Reed76]. The two solutions are very similar in nature. They consist of interleaving layers of the virtual memory and virtual processor mechanisms. Reed's solution is interesting because it implicitly uses a type extension concept very similar to ours to implement abstract code interpreters in two layers (see figure 4.5). The top layer implements user processes as we have defined them in the description of NSS. The bottom layer implements virtual processors. User processes are implemented in terms of virtual processors and virtual processors are implemented in terms of physical processors (not represented in figure 4.5 because they are implemented in hardware, entirely below the virtual memory mechanism described by the figure). The virtual processor manager multiplexes physical processors among virtual processors. The user process manager multiplexes virtual processors among user processes. User processes are bound to virtual processors at the request of the scheduler, which is a piece of the user process manager that executes in a dedicated (permanently allocated) virtual processor. A user process is unbound from its virtual processor component whenever that component "finishes its current task", i.e. when it goes blocked, waiting for a user generated event.

The dependency loop between the virtual memory and virtual processor mechanisms is avoided in the following way. The user process manager freely takes advantage of virtual memory facilities. However, no virtual memory type manager depends on it to interpret its code. Virtual memory type managers can be entered only as a result of user initiated calls or as a result of taking a processor exception that always leads to either handling the exception and restoring the processor state or aborting the computation and signalling an error to the user. Thus, while executing in the virtual memory mechanism, a user process may temporarily lose its physical processor subcomponent (e.g.,

while waiting for paging I/O). However, it will never loose its virtual processor component as all computation paths in the virtual memory mechanism are finite and ultimately return control to user modules without ever putting themselves in a situation of waiting for a user generated event. In addition, the user process manager could not accidentally unbind a user process from its virtual processor component while the latter executes system code because the virtual processor manager refuses to deallocate (unbind) any virtual processor of which the instruction register currently points to a SDW in the CDSG, thereby indicating that it is executing system code. Thus, virtual memory type managers depend on the virtual processor manager to interpret their code and not on the user process manager. In turn, the virtual processor manager depends only on the core segment manager to implement its maps, programs and address space, i.e. it is entirely core resident. (As explained earlier, the core segment manager does not depend on the virtual processor manager to interpret its code because it is implemented entirely in hardware. Thus, the core segment manager depends only on hardware processors, which in turn do not depend on anything because they are bottom level objects.)

With these last dependencies, we have a complete list of the dependencies in MSS. Figure 4.6 represents the dependency graph of MSS. The line cutting across the figure separates the paged environment from the resident environment. Every box represents a type manager. Solid lines represent component dependencies. Dashed lines represent map dependencies. Certain map and all program, address space and interpreter dependencies are omitted from the figure to avoid confusion. The dependencies that are omitted are trivial anyway. The graph summarizes component dependencies and some non-trivial dependencies for maps.



interpreter dependencies: every module, except the CSM, depends on the VPM
address space dependencies: every module, except the CSM, depends on the CSM

Figure 4.6: MSS dependency graph.

Solutions to problem areas.

In this last section on MSS, we will discuss how MSS handles the situations that caused modularity and structure problems in NSS. While doing so, we will describe the mechanical sequence of operations for handling all system events, i.e. page faults, quota faults, OOPV conditions, bound faults and segment faults.

a. Page fault handling.

In NSS, the page fault handler was required to simulate the addressing hardware to get at the PTW that caused a page fault. The page fault handler could not use directly the absolute address of the PTW left in the copy of the processor state at the time of the fault because that PTW might have been deallocated between the time it was faulted on and the time the page fault handler could actually process the fault. As a result, the fault handler in page control had to depend upon the meaning of SDWs in address space control.

An apparent solution to the problem in MSS would be for the hardware to signal page faults to the connected segment manager so that it could interpret the SDW and then pass the fault down with the absolute address of the faulting PTW to the page frame manager. In fact, this does not solve the problem. It is still possible for the page frame to be deallocated between the time the SDW is interpreted in the connected segment manager and the time control is given to the page frame manager, for instance, if the segment containing the page frame is deactivated during that time window.

Instead, a way is needed to guarantee that a page frame cannot be deallocated between the time it is faulted on and the time its disk record component is copied into a core block. Thus, we recommend that when a physical processor faults on a PTW, the PTW be locked so as to indicate that

the corresponding page frame cannot be deallocated until the I/O operation it is subject to completes.

This solution is adequate but raises one problem. By setting a lock by hardware in a PTW, a virtual processor can prevent the page frame from being deallocated. But it will not prevent other virtual processors executing on other physical processors from faulting on the page frame. What strategy must be used in this case? If a virtual processor faults on a PTW that is locked, it cannot afford to wait in an idle loop for the lock to be reset because this lock will be set for the entire duration of the I/O operation on the page frame. Thus, if a faulting virtual processor sees a lock set in a PTW, it must invoke the virtual processor manager to put itself on a waiting list for the occurrence of the I/O completion event corresponding to the page frame it faulted on. When the event occurs, the PTW is unlocked and all virtual processors waiting for the event are be notified. At this point, these virtual processors should not lock the PTW but should simply restore the processor state that existed before the fault they took to retry referencing the page frame. Chances are high that the page frame will be in core. If so there is no need to lock the PTW. Or the page frame may have been deallocated between the time it was unlocked and the time the waiting virtual processors were notified, which is why these virtual processors should not lock it anyway and must restart the faulting computation.

Another problem is now raised. This is a traffic control problem examined by Saltzer [Saltzer66] and called the wakeup waiting problem. Assume a virtual processor A faults on and locks a PTW. Towards the end of the I/O operation on the affected page frame, a virtual processor B faults on the PTW and finds the lock set. Assume further that the I/O operation completes then and is notified to all waiting virtual processors before B has had a chance to

put itself on the waiting list for the event. If so, B will go blocked and wait for ever for an event that has already occurred. Saltzer has described the following solution to this problem.

A so called wakeup waiting switch for the I/O event is associated with B. The virtual processor manager first notifies all waiting virtual processors of the I/O completion event and then sets the switch of B. On its part, B first resets its switch and then calls the virtual processor manager to put itself on the waiting list. The virtual processor manager will do so only if the wakeup waiting switch of B is still off. If it were on, it would mean that the event B was going to be waiting for has just occurred and it is not necessary for B to wait any longer.

The question is: how does the virtual processor manager know what virtual processors (like B) are interested in an I/O completion event it is signalling? The answer is simpler than it may seem. Since virtual processors cannot be preempted while executing system code, as explained earlier, a virtual processor handling a page fault either is currently running on a physical processor or is already waiting for some I/O completion event. Thus, virtual processors interested in a specific I/O completion event and not yet on the waiting list for it can only be among the virtual processors that are currently running on a physical processor. Thus, the wakeup waiting problem for PTW locks in MSS can be solved with two hardware registers in each physical processor. When a physical processor takes a page fault and finds the PTW locked, it posts the absolute address of the faulting PTW in the first register and turns off the wakeup waiting switch in the other register, all in one atomic operation, to prepare itself to invoke the wait primitive of the virtual processor manager. On its part, the notify primitive of the virtual processor manager first notifies waiting virtual processors and then

broadcasts to all physical processors the identity of the PTW event it just notified. If this name matches the one in the first register of a physical processor, that processor turns on its switch in the second register. This will indicate to the wait primitive of the virtual processor manager that the event the virtual processor wanted to wait on has already occurred.

In fact, the lock in a PTW serves a more general purpose. It means that some (any) I/O operation is going on on the PTW, as a result of a page fault or of a decision of the page removal algorithm. The reader can verify for himself that the operation of the wakeup waiting switch is not affected by the more general meaning of the PTW lock. The page removal algorithm operates, as recommended by Huber [Huber76], in a virtual processor that is dedicated to executing just that algorithm. The algorithm is implemented as part of the page frame manager even though the circular list of core blocks that is the basis of the algorithm is maintained by threading CMAP entries together in the core block manager. The core block manager acts as a subroutine to the page frame manager both for allocating and deallocating core blocks. However, the driving procedures both for the page fault and the page removal algorithms are part of the page frame manager. This explains why the core block to page frame and core block to disk record bindings that are kept in CMAP entries remain uninterpreted at the core block level. They are kept only on behalf of the page frame manager.

In the implementation of NSS (and MSS by inference), the mechanism for deactivating a segment is triggered only after the segment is disconnected in all processes. Since a virtual processor needs a SDW to get at a PTW, a virtual processor could never take a page fault on a segment being deactivated. Thus, it is not necessary that the page frame deallocation algorithm, which is invoked to "deactivate" the page table of a segment, set

the PTW locks while it cleans up the page table. However, not setting the locks is assuming something about the segment disconnection algorithm and thus depending on the correctness of higher type managers. Hence, we recommend setting PTW locks while deallocating a collection of page frames as desirable for clarity although not necessary for proper operation.

b. The quota mechanism.

The manipulation of quota cells in NSS is a major source of problems because it occurs at page control level while quota cells are stored at segment control level and related by a hierarchy defined at directory control level.

There are two reasons why quota cells are manipulated by page control in NSS. First, quota faults are not distinguished from page faults by the hardware. Thus, they are passed to the page fault handler in page control. Second, in response to a quota fault, a disk record must be allocated and made a component of a page frame. Since this part of quota fault handling occurs in page control anyway, NSS implements all of quota handling in page control so that the manipulation of quota cells is protected by the page control lock. The above reasons are plausible excuses for manipulating quota cells in page control but they are not sufficient to warrant doing it that way in a well-organized system.

In MSS, the hardware is modified so that quota faults are distinguished from page faults. (The hardware is also modified so as to avoid taking a quota fault when only reading a zero (null) page. A word of zeroes is simply returned.) Quota faults should not be regarded as page related events since no page exists when they occur. Quota faults should instead be regarded as segment related events. Thus, quota faults do not involve locking a PTW. They are directed into the known segment manager and appear as an error return

from an operation to write into a connected segment. The known segment manager translates the segment number of the segment to be grown into a UID and then invokes the segment manager, which will handle the fault. If the segment to be grown is deactivated by then, the segment manager simply returns and the reference that caused the quota fault must be retried.

If the segment to be grown is active, the segment manager will drive the operations to allocate the page frame and manipulate the quota cell. However, the operations themselves are carried out by the page frame manager on one hand and by a new type manager, called the quota cell manager, on the other hand (see figure 4.6).

The quota cell manager manages the QCT, which contains one entry per quota cell. A quota cell is named by the <u>relative pointer</u> (QCTEP) to the entry that implements it in the QCT. The quota cell manager supports the CU, MQ and RU operations. The quota cell manager is designed to fulfill the lack of a type manager dedicated to implementing concepts (quota faults and quota cells) that are logically related but are scattered across several modules in NSS. Under its quiescent state, the quota information associated with a directory is not called a quota cell and is stored as uninterpreted bits in the VTOC entry of the directory. The quota information is copied into a quota cell of the quota cell manager only while it might be used, i.e. while a segment below the directory is active. Thus, the quota cell manager really implements the concept of an active quota cell. The segment manager maintains the UHT to map UIDs of segments into corresponding ASTEPs and (PVTX, VTOCX) pairs. It also maintains, in a similar way, a hash table associated with the UHT, which maps UIDs of segments representing directories into corresponding QCTEPs and (PVTX, VTOCX) pairs. In essence, this second UID hash table maps UIDs of quota cells into passive quota cells and active quota cells. The

activation of quota cells is done as follows.

When the segment manager is invoked to activate a segment, it must be given the (UID, PVID, VTOCX) for that segment and the list of similar triples for all superior directories, in order. (We will see soon that it is always possible to generate this list because the only type manager to ever request that a segment be activated is the known segment manager and every KST entry contains the triple for the corresponding segment as well as the segment number of the KST entry describing the parent of that segment.) After the segment is activated, the segment manager searches the table associated with the UHT for the UIDs of all superior directories to find out what superior quota cells are already active. It activates those that are not, using the list of (PVID, VTOCX) pairs. The procedure for deactivating quota cells will be explained soon.

At this point, the following has been gained over NSS. First, thanks to a separate quota fault mechanism and the creation of a dedicated quota cell manager, the actual allocation of a page frame occurs in the page frame manager while the manipulation and the storage of (active) quota cells is under the responsibility of the quota cell manager. Second, since (active) quota cells are stored in the QCT of the quota cell manager as opposed to the AST, segments and directories can be deactivated uniformly, without any consideration of hierarchy at the segment manager level. However, three problems remain.

First, while the segment manager and the passive segment manager see quota information, the quota cell manager is responsible for handling quota cells. Thus, when a request to access a quota cell is passed to the segment manager, the segment manager may not directly reference the quota information. A strict type extension view says that the segment manager contains the

mechanism to activate segments and quota cells but not to manipulate them. Thus, the segment manager may never directly update quota information. It must always activate the quota cell first and then operate on it via requests to the quota cell manager. The quota cell manager guarantees the correct management of a quota cell given the information in that cell. But the segment manager is responsible for the integrity of that information. (To parallel this situation, the active segment manager guarantees the semantics of the CSL, MSL and CRU of an active segment while the segment manager is responsible for the integrity of those items between activations.)

Second, in order to support the CU operation to grow and shrink segments, every segment that is active (susceptible to grow or shrink) must now be bound to the quota cell of its parent directory. Every UHT entry (1) could contain the QCTEP of its parent directory quota cell. On a quota fault, the segment manager would then extract this QCTEP from the UHT entry and present it to the quota cell manager to perform the CU operation.

Finally, there is the problem of the quota cell hierarchy. This hierarchy is fundamental to the CU operation and it must be used to decide how quota cells may be deactivated. Indeed, directories and segments can be deactivated uniformly only because quota cells have been removed from the AST. However, the concept of a hierarchy must now exist among quota cells in the QCT to allow a CU operation to walk up the hierarchy of quota cells, updating all cells up to and including the quota cell of the lowest quota directory, and to prevent quota cells from being deactivated as long as any segment that is lower in the hierarchy is active. Thus, quota cells in the QCT must be

(1) This is true only for UHT entries corresponding to segments that are charged to a quota cell. Directory segments and even some user segments are never charged to any quota cell. The UHT entries for such segments point to no quota cell.

threaded to their parent just as AST entries were in NSS and they must contain a count of inferior active segments. This defines a hierarchy of quota cells.

The quota cell deactivation algorithm may be part of the segment deactivation algorithm. Every time a segment is successfully deactivated, the segment manager calls the quota cell manager to find out whether the quota cell associated with the parent directory of the deactivated segment has any other segment charged to it currently active. If not, the segment manager requests the deactivation of the quota cell and repeatedly calls the quota cell manager to walk up the hierarchy of quota cells until a cell is found that cannot be deactivated because it has a segment charged to it currently active.

While this design is plausible, it has several disadvantages. First, the ability to deactivate segments and directories uniformly is an advantage but the price we have paid for this advantage is dear. A list of (UID, PVID, VTOCX) triples must be given to activate any segment so that the hierarchy of quota cells above that segment can be reconstructed. Second, the sheer idea of maintaining a hierarchy of quota cells is a source of complexity for the quota cell manager. Of course, the hierarchy is necessary to support the MQ operation but it would appear redundant to keep a duplicate copy of that hierarchy in the quota cell manager while it also exists (in a less accessible form) in the directory manager. Would it not be possible to have a quota cell manager that can move quota between any two quota cells and let the directory manager constrain which two quota cells can be subject to a specific MQ operation? The following paragraphs explain how this is possible.

The hierarchy of quota cells is kept in the quota cell manager so that when a CU operation is performed, the U field of the chain of affected quota cells may be updated. However, the CU operation updates the U in the chain of

indirect quota cells not for itself but to support the MQ operation. When a directory changes status from quota to non-quota or vice-versa as a result of a MQ operation, the U of its associated quota cell has to be added to/subtracted from the U of the quota cell of its parent. If the CU operation did not maintain the U for all quota cells, changing the quota status of a directory on a MQ operation would require computing its U at the time of the operation. This means adding the CRU of all the segments in the subtree below the affected directory, which may be a very expensive task. In fact, it is an impossible task because it would require "freezing" the subtree and the CRU of every segment in the subtree so that the computation would be consistent and look atomic. "Freezing" would require the use of a lock keeping every process temporarily out of both the segment manager and the directory manager since the computation requires adding quantities that are defined at the segment level but are related by a hierarchy defined at the directory level. Such a lock would violate modularity and is thus ruled out.

Based on the observation that all quota cell threads and the associated complexity in the quota cell manager result from the need to support the MQ operation, we suggest a slight change to the definition of the MQ operation that will eliminate these problems. When the directory manager is invoked by a user to move quota between a parent directory P and one of its sons S, it passes the call down through the segment manager to the quota cell manager and tells the quota cell manager whether S contains any entry. If as a result of the MQ operation defined in NSS, S would not change quota status (i.e. from quota directory to non-quota or vice-versa), the new MQ is defined to perform like the original one. But if S would change quota status according to the original functionality of MQ, the new MQ will let the change occur as expected only if the directory manager indicated that S is empty (contains no entry).

If S is not empty, the new MQ operation is defined to abort and return an error code indicating that the move is illegal. Thus, the new MQ operation is identical to the original one except for the fact that one may not move any quota to a non-quota directory or remove all quota from a quota directory if that directory contains one or more entries. This modification has two consequences.

First, when the quota cell manager moves quota down from P to S, which is assumed empty and non-quota, it need not read the U of S to subtract it from the U of P because the U of S is guaranteed to be null since S is empty. In other words, it is not necessary to ever maintain the quota cell of a non-quota directory because the only useful item it used to contain was the U field and the new MQ operation never looks at it anyway.

Second, in addition to not maintaining the quota cell of a non-quota directory, it is not necessary to thread quota cells together. Since a quota directory can become a non-quota directory only if it is empty, it is guaranteed to not change status as long as there is any segment charged to it. Thus, a segment can be charged <u>directly</u> to the <u>same</u> quota cell during its whole life because the existence of the segment guarantees that all directories above it are not empty and therefore cannot change status.

In consequence, when a directory entry is created, it should be tagged with an integer N indicating how many levels higher in the file system hierarchy the lowest quota directory is. This integer is incremented by one at each level of the hierarchy unless a new quota directory is created, in which case the integer is reset to 0. When a segment is initiated, the KST entry should be tagged with the integer N of the corresponding directory entry. Finally, when the segment is activated, the known segment manager, which always requests the activation, as will be seen, need not construct a

whole list of (UID, PVID, VTOCX) triples but need only retrieve the triple for the lowest quota directory above the segment being activated. Retrieving this triple is easy since every KST entry is bound by segment number to the KST entry for the parent directory of the segment it describes, the KST entry of the segment to be activated contains the integer N telling how many KST entries "above" the current one must be followed to reach the lowest quota directory, and the KST entry for that quota directory contains the desired triple, as will be seen soon. To control the deactivation of quota cells, it is sufficient to have a count in each quota cell of the number of segments charged to it that are currently active.

While the functionality change we recommend for the MQ operation may sound drastic, experience with Multics shows that it would not be in practice. A data collection experiment on Multics has shown that almost all quota directories are so-called project directories or user directories. (Each project on Multics has its own directory and each user has its own directory that is a son of the directory of the project the user is working on.) Such directories are quota directories from their creation to their destruction, i.e. since a time when they are obviously empty to a time when they have to be emptied anyway. Only on the order of a dozen directories were made quota directories by individual users. For such users, it is still possible to simulate NSS with MSS if they want to move quota to a non-empty non-quota directory or to remove all quota from a non-empty quota directory. They can create an empty directory named X, move quota to it, then move the subtree below the original directory into X, delete the original directory and rename X after the original directory. This is an involved operation that requires extra quota for the transition state with two directories but it should satisfy the need of the few sophisticated users who would ever want to use it.

As we have designed it, the MSS quota mechanism is perfectly clear, structured and modular. The policy for the MQ and RU operations, which are based on the existence of a hierarchy and are conditional on certain directory attributes, is defined at the directory level. However, the directory manager does not know anything about the storage and the manipulation of quota cells. This mechanism is implemented by the quota cell manager, which in turn ignores everything about the hierarchy. All the quota cell manager needs is a count, in each quota cell, of how many inferior segments charged to it are active. This count is updated by the segment manager every time a segment is activated or deactivated. The page frame manager is not at all involved in the management of quota cells. It only allocates page frames upon requests originating from the segment manager, after the quota cell manager has authorized such allocation.

c. OOPV conditions and access recalculation.

OOPV conditions and access recalculation cause modularity and structure violations in NSS because they require accessing information that resides at directory control level ((PVID, VTOCX) and access control list) but they are handled at lower levels (segment control and address space control).

These two problems admit of similar solutions in MSS. It is tempting to say that OOPV conditions and segment faults should be reflected as high as the directory manager since that is where the necessary information is stored to handle those events. However, OOPV conditions are events detected on only a very small subset of quota faults. Since the occurrence of an OOPV condition can be detected only after some processing has been done, the proposed design would imply that all quota faults be reflected as high as the directory manager level. This design is not plausible because quota faults are error returns from operations on segments and have nothing to do with directories.

As to segment faults, it would be inappropriate to reflect them to the directory manager. Segment faults are faults involving the address space of a process and not the whole file system. Thus, segment faults as well as quota faults should be reflected no higher than the known segment manager.

In NSS, the address space control module uses the (parent segment number, UID) pair in the KST entry of a faulting segment to update the (PVID, VTOCX) of the segment on an OOPV condition and to get the (PVID, VTOCX) and the access control list on a segment fault. Such a design is ruled out in MSS since we declared earlier that the known segment manager may never interpret the binding between a KST entry and a directory entry to access or operate on the directory. This would violate the partially ordered structure of the dependency graph.

In MSS, not only the UID and access information but also the (PVID, VTOCX) of a segment are stored in its KST entry. Thus, as long as these attributes do not change, segment fault (and bound fault) handling is more efficient than in NSS as it does not require getting at the directory entry at all for either the (PVID, VTOCX) or the access control list. Activation and connection of a segment can always be requested directly by the known segment manager without requiring accessing directories.

Access must be recalculated in NSS when the DTEM in the directory entry is more recent than the DTEM in the KST entry. In MSS, the comparison between two such quantities cannot be made by the known segment manager since it cannot read the directory or invoke the directory manager to read it. Thus, instead of using DTEMs to decide whether access must be recalculated, we propose using a dedicated attribute, called <u>date-time-access-modified</u> (DTAM), that is stored in the VTOC entry of every segment below the known segment manager level rather than in the directory entry above the known segment

manager level. Every time the access control list of a segment is modified, the DTEM and the DTAM must be modified. On a segment fault, the known segment manager invokes the connected segment manager through the segment manager to reconnect the faulting segment. It passes to the segment manager the UID, the (PVID, VTOCX), the access information and the DTAM that are stored in the KST entry. If that DTAM happens to be less recent than the DTAM in the VTOC entry, indicating that the access control list has changed, the segment manager simply returns an access fault code to the known segment manager. If the UID that is found in the VTOC entry denoted by the (PVID, VTOCX) is not the UID that was given by the known segment manager to the segment manager, indicating that the segment with the given UID has been moved, the segment manager returns a move fault code to the segment manager. In either case, when the known segment manager receives a fault code, it deduces that the KST entry is no more up to date. It thus returns to the quiescent state and notifies the directory manager that the KST entry needs to be updated if it cares to proceed.

One problem remains to be solved. Changing an access control list is the result of a user call to the directory manager. The directory manager is in a position to request updating the DTAM in the VTOC entry and the access control list in the directory entry without causing violations of organization. However, changing a (PVID, VTOCX) is the result of an OOPV condition, which is detected by the disk record manager when it cannot find a disk record to satisfy a quota fault. Thus, the disk record manager must return an OOPV code to the page frame manager, the page frame manager returns the code to the active segment manager, the active segment manager to the segment manager and the segment manager to the known segment manager that originally intercepted the quota fault. Unfortunately, the known segment manager is not in a

position to move the affected segment, much less to update the (PVID, VTOCX) in the directory entry. Our design specifies that the known segment manager must again return to a quiescent state and notify the directory manager to let the move happen and the directory be updated.

The MSS answer to both access recalculation and OOPV handling is based on the comparison of DTAMs or UIDs in KST and VTOC entries and on notifications from the known segment manager to the directory manager. The known segment manager transfers to the directory manager on quota faults it cannot handle because of OOPV conditions and on segment faults it cannot handle because of an out of date KST entry. In both cases, when it receives control, the directory manager can move the segment or update the KST entry as appropriate by invoking respectively the segment manager or the known segment manager with information it keeps. After doing so, it can invoke the known segment manager to retry processing the quota/segment fault it was handling and restore the processor state that existed before the fault. Notice that the known segment manager does not depend on the directory manager because it cannot be hurt if the directory manager fails to call it to finish up its task. The known segment manager only performs a service for the directory manager. It notifies OOPV conditions and out-of-date KST entries and is willing to support quota and segment fault if the directory manager cares to invoke it for that purpose. The known segment manager is not dependent on the directory manager any more than the hardware notifying quota and segment faults is dependent on the known segment manager.

d. Segment deactivation.

So far, we have examined how dependency stucture violations that existed in NSS were avoided in MSS. All violations of modularity that existed in NSS are fixed in MSS in the sense that every module in MSS manages its own data

bases. However, the radical decision not to share data bases may have an impact on the design in certain cases. It does in the case of the AST. In NSS, segment control and page control share access to the AST. For every segment in the AST, page control maintains a running total of the number of pages in core, a "file-modified" switch and a "file-map-modified" switch that are used to deactivate segments. Such items cannot be maintained by the page frame manager in MSS because it is totally unaware of the grouping of pages into segments. This is indicated by our earlier declaration that there is no binding from a PTW to the segment that contains it.

In MSS, the segment deactivation algorithm is executed as part of the segment manager but in a dedicated virtual processor that runs in parallel to virtual processors implementing user processes. Thus, in a way similar to the page removal activity, the segment deactivation activity is a task that is performed "on the side" and is asynchronous with the activity of user processes. The segment deactivation algorithm consists of sequentially inspecting all active segments that are components of segments to find those that have been used the least recently. This is a sort of LRU algorithm for segments. In order to record the identity of all active segments that are components of segments, the active segment manager threads them together in a circular list on behalf of the segment manager as the core block manager threads together the core blocks on behalf of the page frame manager. Thus, the segment manager will never accidentally deactivate an active segment that is not a component of a segment (i.e., an active segment used to store the maps, procedures, or working storage of some type manager) because it simply does not have names for them and they are not threaded in the list of candidates for deactivation. The selection and deactivation of a segment are performed as follows.

First, consider the problem of evaluating the usage of a segment, which was possible in NSS thanks to the count of in-core pages maintained by page control for each segment. MSS answer to this problem is based on the observation that it is desirable that the paging activity be heavier than the segment activation activity, i.e. it is desirable that there always be many more AST entries than there are core blocks. Otherwise, deactivating a segment would often require throwing out of core several pages, which makes deactivation more expensive. Page removal would be driven mainly by segment deactivation and not so much by the page removal algorithm. The hypothesis that there should be many more AST entries than there are core blocks in a system is verified for all existing Multics installations. Under, this hypothesis, there will always be at least one segment that has no page in core. Thus, the segment deactivation algorithm that we propose for MSS, which looks for a segment with no page in core, performs in practice like the NSS segment deactivation algorithm, which looks for the segment with the least number of pages in core. The advantage of the algorithm we propose for MSS is that it does not require keeping a running total of in-core pages for each segment. Instead, to determine if a segment has any page in core, the page frame manager can be invoked with the name of the first page frame (PTA) and a number of page frames (CSL). Determining if any of these page frames is in core is extremely fast as it only requires "ORing" a collection of sequential PTWs together and checking if the result of the "OR" operation has the in-core flag on. To find a segment that has no page in core, the segment deactivation algorithm must repeatedly invoke the page frame manager until it finds a collection of page frames of which none is in core. Of course, the "ORing" operation may still be somewhat less efficient than directly looking up a pages-in-core count. However, experience with Multics indicates that the

average CSL of a segment is between one and two pages. Thus, only one or two PTWs must be looked at on the average.

Now consider the problem of deactivating a selected segment. In MSS, like in NSS, segment deactivation is the time to update the DTM, DTU and file map of a segment in its VTOC entry if necessary. DTU is always updated (because the segment would not have been activated if it were not to be used). The DTM is updated only if the segment was written into. In NSS, this situation was detected by looking at the "file-modified" switch kept by page control in the AST entry. For the purpose of detecting this situation in MSS, each PTW contains not one but two "modified" bits. They are both set by hardware. When the page removal algorithm copies out a page that has been modified, it looks at and resets only one bit. The other serves as a reminder to tell the segment manager at segment deactivation time that the segment containing that PTW was modified. In NSS, the file map of the segment must be updated in its VTOC entry if the "file-map-modified" switch is on in the AST. In MSS, there can be no such switch. Thus, the segment manager proceeds as follows. The file map of a segment must be updated if the segment has grown or if it has shrunk while it was active. A segment grows as the result of a quota fault. Since the segment manager is involved in handling quota faults, every time it handles a quota fault on some segment, it can record that fact in the UHT entry for the segment to remind itself of later updating the file map when it deactivates the segment. A segment shrinks when the page frame manager decides to throw out one of its pages and discovers that the page has become null while it was in core, in which case the page frame manager requests the deallocation of the home disk record of the page and zeroes out the disk address field of its PTW. When the time comes to deactivate a segment, the page frames of that segment that have become null during the

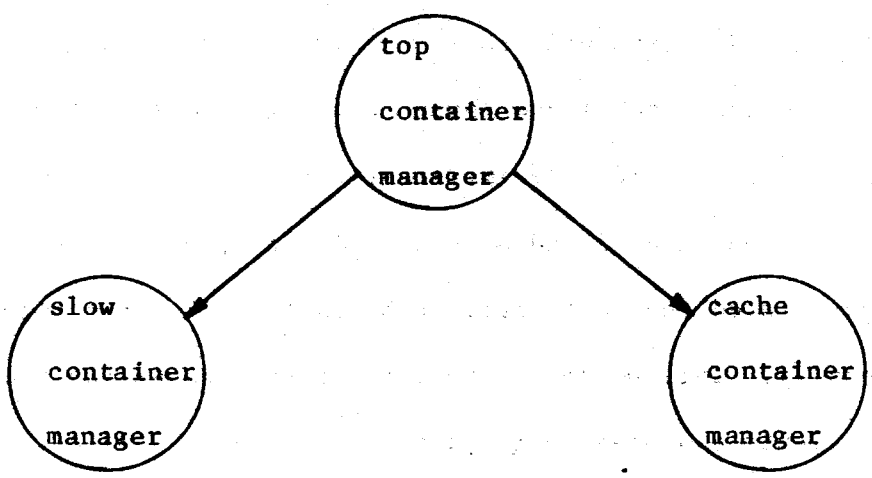
latest activity period are recognized to the fact that their PTW contains a null disk address but has the second "modified" bit on. Thus, when the page frame manager is invoked to deallocate the collection of page frames of a segment being deactivated, it returns the number of the highest page frame that is not null and the number of page frames that have become null over the latest activity period. The former number becomes the new CSL of the segment. If the latter number is not zero, it is subtracted from the CRU of the segment and the U field of the quota cell to which the segment is charged. The file map is then updated in the VTOC entry. This overwrites the address of any disk record that was deallocated by the page frame manager, thereby guaranteeing horizontal protection of these disk records.

5. Structural patterns.

In producing the design of MSS, we have come across three interesting structural patterns that appear to be related to the nature of certain problems they solve. The objective of the present section is to review the nature of these three patterns and to discuss the problems they solve.

Software caches.

The basic pattern is the following.

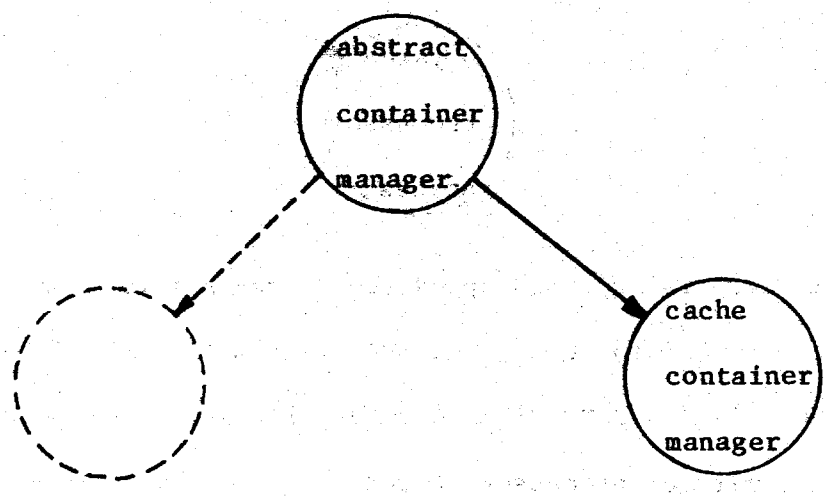


On several occasions, two kinds of data abstractions are defined to contain the same sort of information. For one of them, called the slow container, only two operations are defined. They are READ and WRITE, of which the purpose is obviously to extract information from or to pour down information into a slow container. Many operations may be defined on the other, called the cache container. A third abstraction is defined on top of both, of which the purpose is to move information from the slow container to the cache container and vice-versa to allow operating on that information while it resides in the cache container. The top abstraction may be viewed as a functional abstraction if it implements only the movement of information between the two lower types of containers. On the other hand, it may be viewed as a data abstraction (as we did in MSS) if it supports for the top level information containers all the operations that are supported for the cache information containers and implements them by moving the information to be operated on from a slow container to a cache container and then invoking the operations on the cache container, thereby making the encaching/decaching function transparent to the user.

This basic pattern manifests itself in MSS with disk records (slow), core blocks (cache) and page frames (top), with passive segments (slow), active segments (cache) and segments (top). It is also found with passive segments (slow), quota cells (cache) and segments (top) from the point of view of quota cells. In fact, we could have implemented the concepts of a passive quota cell, an active quota cell and a quota cell independently from segments, which would have isolated the quota cell cache pattern from the segment cache pattern totally. Yet, we decided not to do so because the slow abstractions (passive segments and passive quota cells) are very similar as far as their implementation and maintenance are concerned, and the top abstractions

(segment and quota cell) are intimately related by the synchrony of their activations and deactivations. Separating the abstractions would have increased the complexity of the organization of the system while not reducing the size of the passive segment manager or the segment manager substantially. Thus, we estimated that it would be better to merge the abstractions at those levels.

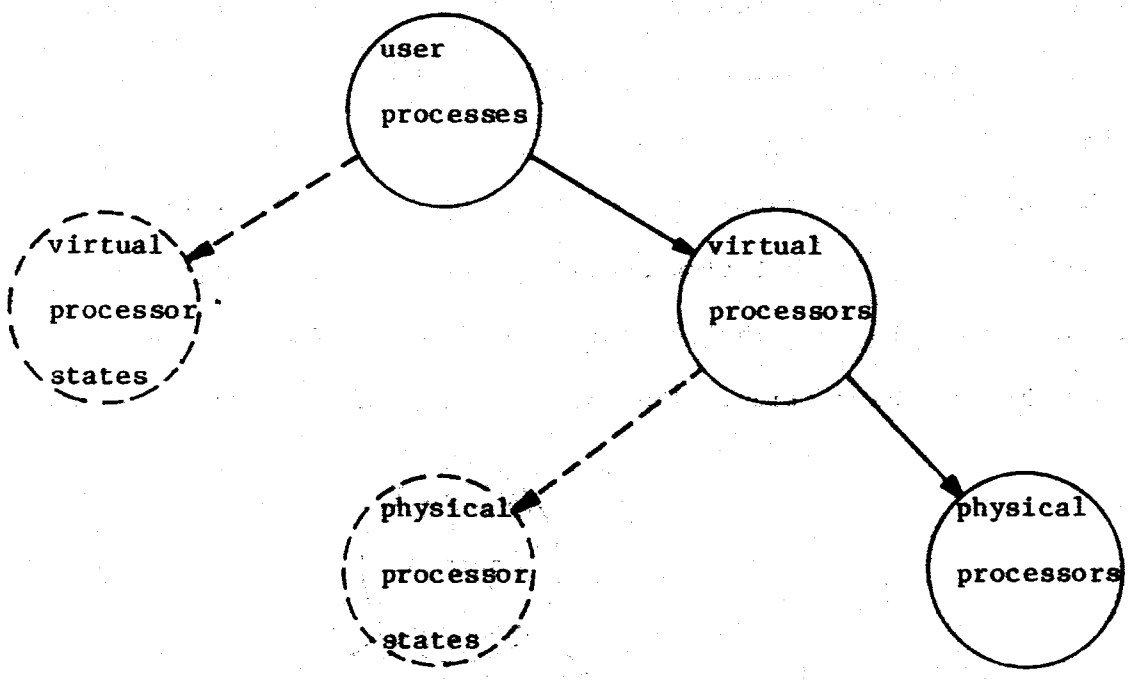
On occasions, the management of the slow containers requires so little data and is so simple that slow containers need not be managed explicitly by their own type manager. The concept of a slow container is collapsed into that of the top abstraction. The following dependency graph results.



An example of this pattern is the configured volume manager (cache) and the physical volume manager (top and slow) case where a demounted physical volume is such a trivial abstraction that it is merged with the physical volume abstraction, i.e. all DT entries are kept together, whether they correspond to mounted physical volumes or not. Another example of such collapsing can be found below.

The software cache pattern is not limited to the virtual memory mechanism dependency structure. It also shows up in the virtual processor mechanism dependency structure, as noted in [Reed76]. Here, the pattern involves

abstract code interpreters instead of abstract information containers. The virtual processor structure is in fact a triple repetition with collapsing of the software cache pattern.



Physical processors are high speed code interpreters. Physical processor states are loaded into real processors when possible. This represents an abstraction called a virtual processor. Virtual processors are capable of interpreting virtual processor states. Thus, virtual processor states are loaded into virtual processors as these are available, thereby implementing a user process abstraction. In fact, the management of the user process and virtual processor state abstractions is so simple that these abstractions are collapsed into the user process concept in MSS.

Interestingly, one notices that the "slow" type manager tends to depend on the "cache" type manager in a software cache pattern. This is because it takes advantage of the software cache abstraction to implement its own programs, maps or interpreters. For instance, the passive segment manager depends on the active segment manager to implement its programs and the user

process manager depends on the virtual processor manager to interpret its code.

The existence of the software cache pattern is an independent confirmation of the results observed by Parnas [Parnas76] who noted that a series of functions is often repeated at two levels within a system in such a way that the higher occurrence has more abstract resources available to it but the lower occurrence contains functions that are more efficient and are used more frequently.

Merging hierarchies.

The second structural pattern that governs the organization of several modules of MSS is the merging hierarchies pattern. One can distinguish two hierarchies of abstractions in MSS: physical resources and logical resources. The physical resources hierarchy is composed of logical volumes, physical volumes, configured volumes and disk records. A logical volume is a collection of physical volumes; a physical volume is the top abstraction of a cache pattern of which the cache abstraction is a configured volume; and a configured volume is a set of disk records. The logical resources hierarchy is composed of directories, segments, passive segments and disk records. A directory describes a collection of segments; a segment is the top abstraction of a cache pattern of which the slow abstraction is a passive segment; and a passive segment is a set of disk records. These hierarchies merge in the sense that they share, at the bottom level, the disk records, which can be viewed either as physical resources or as logical resources.

The interesting fact about these merging hierarchies is that they are not unrelated. As might be expected in almost any system, there exists a policy for grouping logical information containers into physical information containers. Logical resources are grouped in a way dictated by physical

resources. In MSS, all the disk records of one segment must reside on the same physical volume and all the segments in one directory must reside in the same logical volume. One might suspect, as we did until we saw the final design of MSS, that two situations would result from the tight coupling of the hierarchies. First, directories and segments would be composed of (dependent on) logical volumes and physical volumes since these physical containers dictate the grouping of those logical containers. (This would not be outrageous but disappointing since physical containers conceptually contain several logical containers rather than the other way around.) Second, the coupling between the two hierarchies might in fact be so tight that it would generate some unavoidable dependency loops.

It is interesting to see that directories and segments are not components of volumes and it is satisfying to observe that there are no upward dependencies in MSS. This design was achieved by the simultaneous use of UIDs and of tags, as we called the LVIDs, PVIDs and PVTXs earlier. The tags serve as a means to retrieve and associate containers while the UIDs serve for naming them and ultimately validating their identity. Tags are viewed as mere bit strings on which to associate objects. They are not used for what they denote but only for what they are. If they get mixed up by their managers, the integrity of information will not be hurt because it is protected by the UIDs. In other words the directory manager, the segment manager and the passive segment manager are not dependent on the logical and physical volume manager because the latter can never cause the former to operate in a way different from that stated in their formal specifications. While these specifications do not guarantee that a piece of information can always be accessed, they do guarantee that if it can be accessed, its integrity is unaltered.

Notifications.

The last interesting pattern that is found in MSS is trivial but very useful. It consists of a low level module notifying a higher level module and abandoning control without making itself dependent on the higher level module.

This pattern is of course very frequent in any processor in the form of hardware processor exceptions. A processor exception is a notification from the bottom level code interpreter to some higher level type manager indicating that interpretation cannot proceed because some element necessary to proceed is not accessible to the code interpreter. If executing a hardware instruction is regarded, as it should be, as invoking an operation of the bottom level code interpreter, a processor exception may be viewed as an error return from an operation that could not be performed.

The pattern is also used in software in MSS to notify OOPV conditions and out-of-date KST entries. In either case, the known segment manager notifies the directory manager that it cannot proceed with what it is doing. In the OOPV case, the hardware first notifies the known segment manager that it cannot write into some page of some segment because the page does not exist. This is an indication that a reference to a connected segment could not be interpreted. The known segment manager knows how to cause changes in the connected segment that will make the reference succeed. In a few cases though, it is told that the connected segment cannot be grown for lack of physical space. This implies that the segment should be moved and hence that the directory entry must be updated. Since the known segment manager does not know what directories are, it translates the original quota fault on a connected segment into a move fault on a known segment, which the directory manager intercepts and handles appropriately. The same sort of reasoning can be proposed for the case of access recalculation. The known segment manager

can handle segment faults on connected segments for which access has not changed. But it cannot handle other segment faults and therefore maps them into access faults on known segments that the directory manager intercepts and handles.

6. Conclusion.

At this stage, it is probably unnecessary to review the modularity and the structure of MSS. It is clear from the preceding sections that none of the data bases are shared and that the dependency structure contains no directed loop. It is also clear that every type manager is substantially simpler and smaller than the modules defined in NSS. The purpose of this concluding section is to discuss a few consequences of having used type extension to organize MSS.

Performance.

The efficiency aspects of MSS deserve some comments. It is impossible to rate the efficiency on an absolute scale as we have no unit of efficiency. It is even inappropriate to dare evaluate the efficiency of MSS versus that of NSS. MSS has not been implemented. Because it is so similar to NSS, we are convinced that MSS is a logically plausible system but it would be ambitious to make any firm statement about its performance versus that of NSS. Not only is it difficult to distinguish in MSS the areas that might be more or less efficient than their equivalent in NSS but it is impossible to predict the impact of MSS on the performance of the whole operating system. Regardless of the particular strong or weak points in the performance of MSS itself, certain design features that are individually more efficient may cause bottle necks in the overall system because their logical conception causes them to be executed more frequently. Conversely, certain functions that might be individually

less efficient in MSS may in fact not have a negative impact on the overall design because they are performed less often than in NSS.

However, we would like to risk two comments on the performance of MSS. First, we believe that it would be comparable to that of NSS. To justify this, we observe several facts. The overall structure of MSS can be mapped into the overall structure of NSS so that the higher level type managers of MSS correspond to the higher level modules of NSS. MSS maintains the same data bases as NSS, except for the fact that the AST, the QCT and the PFT are separated in MSS. Those MSS data bases that have an equivalent in NSS fulfill the same function as that equivalent except for a few instances (e.g., the DTEM in a directory entry is no longer used for access recalculation, and the number of in-core pages is no longer kept for every active segment). The handling of certain situations in MSS is slower than in NSS (e.g., OOPV conditions and access recalculations) because more computation goes on before they are detected. On the other hand, those situations are rare and the handling of the corresponding situations in NSS (page fault and segment fault) is more efficient because it is assumed a priori that the rare situations do not occur normally. (Page faults do not require sorting out quota faults and OOPV conditions and segment faults do not require fetching the (PVID, VTOCX) from the directory entry on every occurrence.) In addition, the handling of the quota problem in MSS has released the constraint that all the directories above an active segment or directory must be active. This represents a non-negligible saving of space in the AST, which is bound to be translated into an improvement in the performance of the system (less paging activity if the core block pool is increased or less AST activity if the original AST size is preserved.) The point we are driving at is that the structure and the modularity are primarily -- though not exclusively -- abstract views of the

design. In essence, the resulting implementation is very close -- though not identical -- to that of NSS. Therefore, we suspect that the performances of MSS and NSS should be in the same vicinity.

Second, readers might object that the performance of MSS may suffer from its modularity. Since data bases are never shared, a module can find out about some information residing in the data bases of some other module only by asking the other module about it. It cannot directly reference the data base. Calling the other module would of course be less efficient than directly referencing the data base. In reality, we do not believe that modularity at the level of the language used to code the system can affect performance at the level of the machine language. Indeed, the type checking compiler may and should include macro expansion and global optimization features. Thus, it is possible to have strict modules at the level of the coding language, which is all that counts for understanding, maintaining and verifying the system. And at the same time, the compiler may compile a module by substituting code in-line. The resulting MSS object code would no longer be a strict module, but this does not matter, and its performance could be as good as the equivalent code in NSS.

Modules as data abstractions.

The nature of all the modules in MSS deserves one comment. All modules happen to stand for data abstractions (type managers) only because we have strictly respected the type extension view at all times and have regarded every concept from an object based point of view. However, the type extension technique does not intrinsically rule out functional abstractions. We are convinced that there may exist systems for which a certain aspect cannot be properly designed if it is viewed from a data abstraction point of view.

Horizontal protection of internal objects.

In chapter II, we made the point that the horizontal protection of internal type objects, i.e. the objects implemented by the virtual memory mechanism below the base level, was not guaranteed by their type manager. Instead, assertions must be verified about the modules using the internal type objects to show that they never use the uid of a deallocated/deleted object. We suggested then that, while the task of producing the assertions might seem hard, it was not in practice. In most cases, only one module ever sees the uid of an object provided by some type manager. Thus, assertions can easily be imposed on that module to guarantee that it destroys all copies of the uid of an object it releases. One can verify in this way the protection of all internal types except disk records and page frames, which are used by several modules in the sense that their uids may at times be seen by several modules. For the latter internal types, we suggested in chapter II that it nonetheless would be sufficient, in general, to produce assertions on only one module because, usually, only one module has caused the propagation of copies of the uids through several modules, is aware of the distribution of the uids and can cause their destruction. This is indeed the case of disk records and page frames. Disk records are shared by page frames and passive segments. The segment manager controls the propagation of their uids. Page frames are shared by active segments and connected segments. The segment manager also controls the propagation of their uids.

Conclusion.

This chapter has demonstrated the applicability of type extension as a technique for organizing the virtual memory mechanism of a real general purpose time-sharing system. In a first section of the chapter, we have presented the functionality of the mechanism as the user sees it. In a second

section, we have described the implementation of the mechanism as it exist in reality. We have pointed out where and why modularity and structure were violated in this real implementation. In a third section, we have proposed a new design based on type extension that preserves the original functionality of the mechanism. We have briefly characterized each module of the mechanism and we have systematically inspected the structure of the mechanism to conclude that the design was strictly modular and partially ordered by the dependency relation. We have then explained how the organization problems encountered in the real implementation of the mechanism are eliminated in the proposed design. In a fourth section, we have discussed three structural patterns that seem to be fundamental in the design of a well-organized virtual memory mechanism for a general purpose time-sharing system. The software cache pattern appears every time a scarce resource is multiplexed among more abundant abstractions. The merging hierarchies pattern appears where the organization of logical containers is governed by the organization of physical containers. The notification pattern appears every time a module notifies an event upward to implement an error return from an operation that was, sometimes implicitly, invoked.

V. Conclusion.

1. Summary.

This thesis has presented a technique for organizing the virtual memory mechanism of a computing utility. The technique is based on the concept of type extension. The virtual memory mechanism of a system is regarded as a set of type managers supporting abstract information containers. These abstract containers are implemented in terms of more primitive containers such as core blocks and disk records. Strictly applied to a virtual memory mechanism, the formalism of type extension helps organize the design of the mechanism into a structured set of modules. The modules are the type managers for the abstract containers utilized and supported by the virtual memory mechanism. The structure of the mechanism reflects the structure of the information containers it implements.

In addition to reviewing some background notions and existing literature on organization techniques for computing systems, chapter I justified why a technique for organizing virtual memory mechanisms is desirable. Such mechanisms are often complex and require a fairly large amount of code. Thus, it is desirable that they be well-organized to facilitate understanding and maintaining them. In addition, the rise in interest for certifiably secure systems has fostered the need for systems of which the security kernel can be verified correct. A verification of correctness cannot be carried out unless the security kernel is well-organized. Since the virtual memory mechanism of a system is, in general, part of the security kernel, it is necessary to organize it so it can be verified correct.

Chapter II has defined the meaning of type extension in the environment of a virtual memory mechanism. It has examined the nature of the abstractions

one may encounter in such an environment. In particular, it has introduced the idea of abstract types providing a limited supply of objects that have an essentially unbounded lifetime. Such abstractions are necessary to model the behavior of storage resources that are scarce and need be multiplexed among all users over time. It has also pointed out the difficulty of providing a mechanism (e.g., capabilities) for protecting abstract information containers at run-time inside a virtual memory mechanism. Implementing run-time protection inside a virtual memory mechanism is difficult with today's technology because run-time protection depends on addressing potentially large access control data bases while addressing of large data bases depends on precisely having a virtual memory mechanism or a special purpose but then cumbersome I/O mechanism.

Chapter III has explained how the type extension concept can be exploited as an organization technique in a virtual memory mechanism, and what the advantages of this technique are. The usefulness of the technique for selecting abstractions to modularize a virtual memory mechanism was pointed out. The property of type extension to foster modularity and structure in a virtual memory mechanism was discussed. In particular, it was explained how the environment of execution of a type manager (maps, programs, address space, code interpreter) should be set up to avoid violations of the partially ordered, object based dependency structure of the system. Finally, the advantages of type extension towards providing a strategy for avoiding deadly embraces when locking system data bases were stressed.

In chapter IV, we have demonstrated the applicability and the usefulness of the type extension technique by exploiting it to (re)organize the virtual memory mechanism of a real computing utility, the Multics system. This case study is interesting because it showed the ability of the type extension

technique to cope with the complex functionality of a real system. Most existing techniques have been illustrated by examples of applications involving "paper" systems. Such examples demonstrate only the conceptual aspects of the technique that is employed but fail to demonstrate its applicability to the organization of a real system. The functionality of the "paper" systems is in some sense built to fit what their organization permits. In this thesis, we wanted to examine both the conceptual and the engineering aspects of the type extension technique. Thus, we took the unconstrained functionality of a viable system as granted and worked from there towards a clean organization.

2. Results.

The first contribution of the thesis is a technique for organizing the virtual memory mechanism of a computing utility. This technique has proved convenient to use and capable of yielding modular and structured designs for virtual memory mechanisms. It was demonstrated useful to organize even real systems with all the complexity and hardware constraints associated with them.

The second contribution of the thesis is in the area of the design of general purpose, time-sharing systems. By illustrating the use of the type extension technique in chapter IV, we have at the same time produced a well-organized design for the virtual memory mechanism of such a system. To the best of our knowledge, such a clean design has never before been proposed for virtual memory mechanisms of the size and complexity of the Multics storage system. It is satisfying to know that well-organized designs for such mechanisms exist and could lead to reasonably efficient implementations.

A third result of the thesis, which was not expected as an initial objective but came as an interesting conclusion, is the demonstration of the

analogy between the type extension concept we have used to organize MSS and the type extension concept that was used implicitly to organize the virtual processor mechanism of Multics [Reed76]. The two concepts serve somewhat different purposes. Reed used type extension to break a dependency loop between the virtual memory and the virtual processor mechanisms of Multics. We used type extension to break the virtual memory mechanism of Multics into a set of smaller and simpler mechanisms. However, for all practical purposes, the two concepts are equivalent. They deal with the same problem, the organization of a mechanism in the security kernel. They face the same issues, designing partially ordered modules in the constrained execution environment of the security kernel. They are equivalent with respect to the dependency relation in that the dependency structures that result from using them are interleaved. And last but not least, the software cache structural pattern appears to be fundamental to the exploitation of type extension for multiplexing resources, be they storage or processing resources.

3. Future research.

A first research topic that remains open is the practicality of virtual memory mechanisms based on type extension. We believe that a system like MSS is conceptually plausible and could achieve a reasonable performance if it were implemented. However, we cannot assert this until someone will have built such a system clear through the implementation, so that the system can be tested. Even with the type extension technique as it was described in this thesis, we have a long way to go to implement a system based on type extension. We lack the most elementary tools to achieve a correct and efficient implementation, namely a compiler that can perform type checking, verify uid conservation, do in-line substitution of macros and optimize code

across type managers to break the barriers of strict modules at the level of the object code.

The thesis has developed a technique based on type extension for organizing the virtual memory mechanism of a system. This technique appears very similar in nature if not in origin to the technique used elsewhere [Reed76] to organize the virtual processor mechanism of a system. A question that naturally comes to the mind is: is the type extension technique used for virtual memory and virtual processor mechanisms also applicable to I/O mechanisms? Could it be exploited to organize the external communication interfaces of a system, including the eventual network interfaces? In our own mind, we suspect type extension is applicable to this area. Since it proved applicable to the organization of the virtual memory I/O of a system, it is probably applicable to the non-virtual memory (external) I/O of a system. However, we have given no further thoughts to the idea and we do not have any demonstration of its validity.

As technology evolves, we observe a trend towards distributed computing and very large distributed data bases. Also, we may forecast for the not too distant future the introduction in the market of personal computers that can tap into a cable or be hooked to a network to communicate with distant hosts to access enhanced storage and processing facilities. Because of these observations, there is little doubt that what is called today the external network interface of a system will be looked at tomorrow as an internal interface in some distributed storage system. What looks today like a transfer of information between physically and logically separate storage systems might look tomorrow like a transfer of information between two components of a higher level abstract file in a physically distributed but logically integrated storage system. One may wonder whether the type

extension technique or some variation of it will be useable for organizing such distributed systems. In particular, will it be capable of coping with a storage system of which some parts may be inaccessible at times due to local failures while other parts must remain accessible and consistent? Will it be capable of collecting different local storage system architectures into a single global formalism of type extension? Many more questions could be formulated along these lines.

Leaving the concept of type extension and returning to present systems, further research is necessary to produce certifiably secure systems. While the type extension technique, among others, represents a step in the direction of more understandable and reliable systems, it does in no way constitute the final answer to all problems. Further research is needed in four directions: programming languages, specification techniques, security standards and verification techniques.

As far as programming languages are concerned, it is desirable to have high-level languages with type extension features and a system programming orientation. CLU [Liskov76] includes the desirable type extension features. However, under its current implementation, it is not adequate for system programming purposes. The CLU system assumes the existence of a "heap" for storing all its variables. One of the main problems of doing type extension inside the kernel of a system is that such a heap is not available and variables must be stored in different kinds of containers with different addressing characteristics.

While module specification techniques such as the one proposed by Parnas [Parnas72a] or the one used by SRI [Neumann75] seem appealing, they lack several features. First, they lack adequate semantics to express certain facts dealing with, for instance, parallelism and what Parnas calls the

"global properties" of a module (properties involving several of the functions supported by the module). Second, they lack an adequate language to formulate specifications. Neither Parnas nor SRI claim the language they use is adequate. But the fact that it is not is an indication that we lack a language that is altogether powerful and readable, yet formal.

Only one concept of security -- the U.S. Department of Defense security controls -- has been analyzed to the point where formal models could be formulated for it [Bell73]. While this concept is interesting for military purposes, it is not for most civilian applications. Protection mechanisms such as access control lists and capabilities have been built into various systems that are designed mainly for civilian use. However, we have no idea of what formal security models these mechanisms might support. Before we embark on certifying a system, we should know what we are trying to certify about it.

Finally, verification techniques will be necessary to prove the correctness and the security of systems. Some good work has already been produced in the area. However, we are not convinced that a system like MSS, for instance, could be verified with existing techniques, even if complete specifications existed for it. The same problem is encountered with verification techniques as with specification techniques. There is a lack of ability to handle problems like parallelism and "global properties".

This thesis has attempted to present a technique for organizing virtual memory mechanisms to make them understandable and verifiable. Considering the crucial role played by the virtual memory mechanism of a system with respect to security, this is not a negligible result. However, it falls far short of achieving the objectives contemplated in our quest for certifiably secure

systems, as indicated by the previous paragraphs. All the tasks evoked earlier -- choice of a security model, formulation of specifications, verification of correctness and security, and certification -- are necessary steps towards a secure system but we do not know how to take most of them. Yet, it is worthwhile attacking these problems because there will be a need to solve them for as long as there will be computers. Because of the ever expanding applications of computers in private and public sectors, and because of the ever increasing costs of developing and maintaining software, computer systems that are easy to understand, to maintain and to verify are becoming a necessity. The times of perhaps efficient but "quick and dirty computer hacking" are over. Considering what evil computers can do to us, it is time that we consider what good we could do to them so they will be cheaper to maintain, they will be safer to use, they will be harder to break and they will serve us better.

Bibliography.

[Ames75] S.R.Ames, "The design of a security kernel", M75-212, Mitre Corp. (Apr.1975).

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

[Bensoussan68] A.Bensoussan, "Overview of the locking strategy in the file system", Multics System Programmers' Manual, section BG.19.00 (Dec.1968).

[Bensoussan72] A.Bensoussan, C.T.Clingen, R.C.Daley, "The Multics virtual memory: concepts and design", CACM 15 5, pp 308-318 (May 1972).

[Bratt75] R.G.Bratt, "Minimizing the naming facilities requiring protection in a computing utility", S.M.Th., MIT & MAC-TR-156, MIT Lab. for Comp. Sc. (Sep.1975).

[Cohen75] E.Cohen, D.Jefferson, "Protection in the Hydra operating system", Proc. ACM 5 Symp. on Oper. Syst. Princ., pp 141-160 (Nov.1975).

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

[Habermann69] A.N.Habermann, "Prevention of system deadlocks", CACM 12 7, pp 373-377 (Jul.1969).

[Habermann76] A.N.Habermann, L.Flon, L.Cooprider, "Modularization and hierarchy in a family of operating systems", CACM 19 5, pp 266-272 (May 1976).

[Havender68] J.W.Havender, "Avoiding deadlocks in multi-tasking systems", IBM SJ 7 2, pp 74-84 (1968).

[Hoare73] C.A.R.Hoare, "A structured paging system", Computer Journal 16 3, pp 209-215 (Aug.1973).

[Huber76] A.R.Huber, "A multi-process design of a paging system", S.M.Th., MIT & to appear as TR, MIT Lab. for Comp. Sc. (1976).

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

[Lampson69] B.W.Lampson, "On reliable and extendable operating systems", 2nd NATO Conf. on Softw. Eng. (Oct.1969) & Infotech State of the Art Report (1971).

[Lampson76] B.W.Lampson, H.E.Sturgis, "Reflections on an operating system design", CACM 19 5, pp 251-265 (May 1976).

[Levin75] R.Levin, et al., "Policy/mechanism separation in Hydra", Proc. ACM 5 Symp. on Oper.Syst. Princ., pp 132-140 (Nov.1975).

[Liskov72a] B.H.Liskov, "The design of the Venus operating system", CACM 15 3, pp 144-149 (Mar.1972).

[Liskov72b] B.H.Liskov, "A design methodology for reliable software systems", Proc. AFIPS FJCC 41, pp 191-199 (1972).

[Liskov76] B.H.Liskov, "A note on CLU", CSG Memo 136, MIT Lab. for Comp. Sc. (Feb.1976).

[Multics74] --- "Introduction to Multics", MAC-TR-123, MIT Lab. for Comp. Sc. (Feb.1974).

[Neumann74] P.G.Neumann, et al., "On the design of a provably secure operating system", Proc. Intl. Workshop on Prot. in Oper. Syst., IRIA, pp 161-170 (Aug.1975).

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

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

[Parnas71] D.L.Parnas, "Information distribution aspects of design methodology", Proc. IFIP Cong., pp 340-344 (Aug.1971).

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

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

[Parnas74] D.L.Parnas, W.R.Price, "Using memory access control as the only protection mechanism", Proc. Intl. Workshop on Prot. in Oper. Syst., IRIA, pp 177-182 (Aug.1974).

[Parnas76] D.L.Parnas, "Some hypotheses about the "uses" hierarchy for operating systems", Research Report BS I 76/1, Technishe Hochschule Darmstadt, Fachbereich Informatik (Mar.1976).

[Popek74] G.J.Pokek, C.S.Kline, "The design of a verified protection system", Proc. Intl. Workshop on Prot. in Oper. Syst., IRIA, pp 183-196 (Aug.1974).

[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).

[Radin76] G.Radin, P.R.Schneider, "An architecture for an extended machine with protected addressing", IBM, TR 00.2757 (May 1976).

[Redell74] D.D.Redell, "Naming and protection in extensible operating systems", Ph.D.Th., U.C.Berkeley & MAC-TR-140, MIT Lab. for Comp. Sc. (Nov.1974).

[Reed76] D.P.Reed, "Processor multiplexing in a layered operating system", facility", S.M.Th., MIT & MIT/LCS/TR-164, MIT Lab. for Comp. Sc. (Jun.1976).

[Robinson75] L.Robinson, et al., "On attaining reliable software for a secure operating system", Proc. Intl. Conf. on Reliable Software, pp 267-284 (Apr.1975).

[Saltzer66] J.H.Saltzer, "Traffic control in a multiplexed computer system", Sc.D.Th., MIT & MAC-TR-30, MIT Lab. for Comp. Sc. (Jul.1966).

[Saltzer75] J.H.Saltzer, M.D.Schroeder, "The protection of information in computer systems", Proc. IEEE 63 9, pp 1278-1308 (Sep.1975).

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

[Saxena76] A.R.Saxena, "A verified specification of a hierarchical operating system", TR-107, Stanford Electronics Laboratories (Jan.1976).

[Schell71] R.R.Schell, "Dynamic reconfiguration in a modular computer system", Ph.D.Th., MIT & TR-86, MIT Lab. for Comp. Sc. (Jun.1971).

[Schiller73] W.L.Schiller, "The design and specification of a security kernel for the PDP-11/45", ESD-TR-75-69 & MTR-2934, Mitre Corp. (republished May 1975).

[Schroeder71] M.D.Schroeder, "Performance of the GE-645 associative memory while Multics in operation", Proc. ACM Workshop on Syst. Perf. Eval., pp 227-245 (Apr.1971).

[Schroeder75] M.D.Schroeder, "Engineering a security kernel for Multics", Proc. ACM 5 Symp. on Oper. Syst. Princ., pp 25-32 (Nov.1975).

[Sturgis74] H.E.Sturgis, "A postmortem for a time-sharing system", Ph.D.Th., U.C.Berkeley & Report CSL 74-1, XEROX PARC (Jan.1974).

[Wulf75] W.A.Wulf, R.Levin, C.Pierson, "Overview of the Hydra operating system development", Proc. ACM 5 Symp. on Oper. Syst. Princ., pp 122-131 (Nov.1975).

Appendix. Military security controls in Multics.

Among the various aspects of the functionality of Multics that we have ignored in the design of MSS is the Access Isolation Mechanism (AIM), which is a protection mechanism capable of supporting isolated compartments for storing information. The AIM was initially designed to enforce the security controls defined by the U.S. Department of Defense. This mechanism was purposely ignored because type extension does neither help nor hinder its design. No new modularity or structural issues are raised by the AIM with respect to type extension. There is no interest in discussing the AIM from a type extension point of view.

However, there are two reasons for discussing type extension from the AIM point of view in an appendix. First, military security controls are the only concept of security for which there exists a formal model [Bell73]. Therefore, chances are that the first system to be certified secure will be certified with respect to that model. We wanted to demonstrate that the type extension technique, which was developed, among other reasons, to make virtual memory mechanisms easier to certify, can be used to organize the virtual memory mechanism of the sort of system that will precisely be certified first. Second, this thesis is one result of a project aimed at producing a prototype security kernel for Multics [Schroeder75]. This project, in turn, is one piece of the U.S.A.F. security program that sponsored the development of the military security controls model [Bell73].

Military security controls.

For more information on the functionality of these controls, we refer the reader to the formal model [Bell73]. Here, we will only summarize the main concepts involved in the AIM. Four levels of security are defined in a total

ordering: unclassified, confidential, secret and top secret (four more may be defined in Multics). Several categories may be defined: e.g., AEC, NATO, US, etc.. Sets of categories are partially ordered by the set containment relation. The cross-product of a level and a category set defines a compartment. Compartments are thus also partially ordered by a relation that is the product of the set containment relation and the total ordering relation of levels. Every piece of information is classified into some compartment. Every agent (user) is cleared for a given compartment. The security controls state that an agent can read a piece of information only if his clearance is greater than or equal to the classification of that information. This is called the no-read-up rule. To confine a piece of information to its compartment and higher compartments, the no-read-up rule is not sufficient. It is necessary to prevent an agent authorized to read the information from copying it down into lower compartments where unauthorized agents would be able to read it. This is called the no-write-down rule or more generally the *-property. These security controls are called non-discretionary, which means that they not only prevent unauthorized access but they also prevent authorized agents from leaking information to unauthorized ones at their discretion.

Computer environment.

In a computer environment, the enforcement of the security controls may be implemented as follows. For the no-read-up rule, it is sufficient to never give a user READ access to information stored in higher compartments. For the no-write-down rule, things are more complex. Of course, it is necessary to never give a user WRITE access to information stored in lower compartments. However, this is not sufficient. This blocks only overt information channels between compartments.

There also exist covert channels. These channels are divided into storage channels and time channels. Storage channels are those involving stored data as the support for transmitting information. For instance, if a low clearance user and a high clearance user share the resources of a disk, they both depend on the disk manager for using the disk. By varying his usage of space on the disk, the high clearance user may transmit information to the low clearance user. The information is transmitted via the stored data kept by the disk manager to manage space on the disks. Time channels are those involving elapsed time between observable events as the support for transmitting information. For instance, if a high clearance user and a low clearance user share core, by varying heavily his page fault rate with time, the high clearance user may transmit information to the low clearance user. The information is transmitted via the variations with time of the page fault rate, which might not be stored anywhere but causes delays observable by any user.

In general, it is demanded that storage channels be blocked because they tend to have a relatively large bandwidth. There are two ways to block them. First, shared storage resources can be preallocated by compartment so that the resource usage of one compartment is invisible to and cannot affect other compartments. Second, shared storage resources can be multiplexed over time in such a way that two compartments are never aware that they compete for the resources.On the other hand, it is not required, in general, that storage channels be blocked because they tend to be relatively low bandwidth channels and because blocking them would imply preallocating time. This would be logically equivalent to having an independent computer for each compartment so that time dependent events and signals generated by a high clearance user could not be observed by low clearance users. This design would be grossly

inefficient as it ignores the fundamental principles of multiprogramming. As long as time channels are identified, they present a limited risk and are usually tolerated. Notice that this paragraph has only described current techniques for implementing the AIM. Since we are only interested in showing that type extension does neither help nor hinder implementing the AIM as it exists today, it is not our intention to formulate any judgement about the current implementation of the AIM nor to try do do better than current technology does.

Application to MSS.

Since the incarnation of an agent in MSS is a user process, the concept of clearance is attached to user processes. And since the mode of access (READ or WRITE) to information is controlled on a per segment basis, the concept of classification is attached to segments. In terms of MSS abstractions, we have the choice of attaching it to the directory entry that describes a segment or to the segment itself. If we are concerned about only non-discretionary controls, i.e. if we care to certify the system against only Bell's model and do not care about the correctness of the discretionary access control list mechanism, it is better to attach classifications to segments. By doing so, the directory manager, which is a substantial module, and higher level modules are essentially outside the AIM security kernel. What they can access will be constrained entirely by the AIM built into the segment manager and lower level type managers. This is a vivid example of the interest of a partial ordering based on the dependency relation. Within the same system, one can consider two (or more) different security kernels. The discretionary security kernel is built on top of and includes the AIM security kernel. The interface of the discretionary security kernel is the interface presented to the users. The interface of the AIM security kernel is the interface

presented to the discretionary security kernel. It is defined by the specifications of the configured volume manager, the segment manager, the known segment manager, the connected segment manager and the user process manager. Thus, thanks to the partial ordering of modules, we can define layered security kernel interfaces. Every layer of a security kernel is responsible for the enforcement of a distinguished set of security properties. Each security kernel embodies more security properties than the lower one. Let us now consider the impact of the AIM on the design of MSS.

Let us first consider the abstractions outside the AIM security kernel. The classification of a logical volume, a physical volume or a directory is defined by the classification of the segment that directly or indirectly implements its maps. (The DT is assumed to be split into as many Physical Volume Registration Data (PVRD) segments as there are physical volumes. Thus, every physical volume needs a path name denoting its PVRD segment in the file system.) A priori, a logical volume might contain physical volumes (1) and master directories in different compartments, a directory may describe sons in different compartments and a physical volume may contain disk records belonging to segments in different compartments. There is however one fundamental restriction on the possible classification of the components of such abstractions. The classification of the components of an object O must be higher than or equal to the classification of (the map of) O. If this were

(1) We do not see any use for having physical volumes in different compartments inside the same logical volume. Since all physical volumes in a logical volume are logically equivalent as far as the user is concerned, we fail to see any situation that would justify classifying the physical volumes in a logical volume into different compartments. In addition, this would complicate the mounting/demounting operations as will soon become clear. Thus, even though it may be conceivable to have physical volumes in different compartments, we will assume that this is never desired in practice in the remainder of this appendix.

not the case, it would be impossible to access the components because of the AIM. If a user had the clearance for O, he would not be able to operate on the components by virtue of the *-property. And, if he had the clearance for the components, he would not be able to find out what they are because he would not be allowed to read the map of O by virtue of the no-read-up rule. The conclusion that the structure of an object must correspond to a non-decreasing hierarchy of compartments was arrived at independently in NSS, where the file system hierarchy of directories and segments must be non-decreasing in classification.

In MSS, like in NSS, we have the concept of a transition object, which denotes an object of which the classification is higher than the classification of the object it is a component of. In MSS, like in NSS, the manipulation of transition objects is extremely delicate. Indeed, a user who is cleared to operate on a transition object T may read the map of the object O of which T is a component but he may not write into that map by virtue of the *-property. Thus, he cannot perform any operation on T that would require changing the map of O. (In particular, he cannot delete/deallocate T unless he lowers his clearance or requests the assistance of a so-called <u>trusted</u> user, i.e. a user who has the privilege to write-down but is trusted not to abuse it to transmit sensitive information.) We will say that transition objects are <u>frozen</u>. Practical consequences of this statement for MSS as well as for NSS follow. A user cleared to manipulate a transition master directory cannot operate on it (e.g., set quota on it) in any way that would require modifying the LVRD of the logical volume composed of the master directory because this LVRD has a lower classification. A user cleared to use a transition segment (directory or file system segment) cannot operate on it in any fashion that would require modifying the directory entry that describes

the segment because this entry is part of a directory with a lower classification. In more specific terms, a transition segment implementing a directory can never be subject to access control list changes or to moves resulting from OOPV conditions because this would require modifying the access control list or the PVID,VTOCX in the directory entry describing the transition directory, which is ruled out by the *-property. A transition segment implementing a file system segment may never be subject to access control list changes or to OOPV moves and in addition may never be subject to a quota fault because this would require updating the quota cell of some superior quota directory that has, by definition, a lower classification. Furthermore, every transition directory should be a quota directory to allow the inferior segments to be grown/shrunk. If this were not the case, the CU operation would (because it should be coded so) refuse to grow/shrink a segment if it discovered that the quota cell which it should update has a lower classification than the segment charged to it.

Until now, we have examined the impact of the AIM primarily on the type managers outside the AIM security kernel. As far as this security kernel is concerned, we have only concluded that it should never grant a user READ access to segments in higher compartments and that it should refuse to handle OOPV conditions and quota faults on transition segments.

Let us now examine in more details what the AIM security kernel should do to enforce the AIM. In particular, let us study what is necessary to block the covert storage channels. As explained earlier, it is necessary to guarantee that every type of shared storage resource be preallocated by compartment. Segments are not a problem. They are C/D resources, which means that every UID is given out only once. It is never reused by (shared with) other users. Thus, whether it is used or not cannot be exploited as a storage

channel. A burst in the creation of segments would use up many UIDs at once, which might be exploited as a time channel. However, if UIDs were generated by reading a microsecond clock and if we assume that it is impossible to create more than one segment every microsecond, it would be impossible to tell whether a given UID is in use or not. Known and connected segments are not a problem as they are never shared by several user processes. Every process has its own. Thus, a process could never use its KST or DSG as a covert channel to another process. Core segments are not a problem either because they are used strictly inside the security kernel, which will obviously not try to exploit them to leak information. Active segments, quota cells, page frames and core blocks are not a problem because, even though they are shared by all processes, processes do not directly control their deallocation. Deallocation is under the control of the segment deactivation and page removal algorithms. The above abstractions might be exploited as time channels (e.g., varying page fault and segment activation rates) but such channels are very slow because of the noise introduced by the deallocation algorithms and by all processes competing for the same resources at once.

Disk records could be a problem because processes are in full control of their allocation and deallocation. For instance, a high classification process could use up all the records on a physical volume, thereby causing a low classification process to observe OOPV conditions on any segment it might try to grow. In fact, this sort of channel can be blocked in practice if the security officers of the system are careful not to overallocate quota. Indeed, if the sum of all quota allocated to the compartments sharing a physical volume is no greater than the total amount of disk records available on the physical volume, then the disk records are essentially preallocated by compartment and OOPV conditions will never be observed.

Passive segments and disk sectors (i.e. VTOC entries) are a problem (in MSS as well as in NSS!) because there is no mechanism like quota for controlling their preallocation. Thus, by varying its usage of VTOC entries, a high classification process can transmit information to a low classification process. The only obvious way to block this channel short of introducing quota on VTOC entries is to require that all transition directories be master directories and that all master directories for a logical volume be in the same compartment, or, in short, that each logical volume belong in a single compartment. In other words, this says that all the segments of which the passive image is on one logical volume must be in the same compartments so that the VTOC entries (and the disk records too, by the way) on that logical volume are not, in practice, shared by several compartments. The latter statement does not require that the logical volume, physical volume and directory entry managers be in the AIM security kernel. The compartment associated with a given logical volume and its component physical volumes may be recorded in the label of every physical volume. This label is secure because it can be accessed only by the configured volume manager, which is part of the security kernel. Thus, every time a user process requests the creation of a segment with a given classification on a given logical volume, the given classification is compared with the classification of the given logical volume. If they are not equal, creation is denied.

Finally, configured volumes are not a problem. While it is true that they are shared and not preallocated, it would be hard to use them as channels to communicate information. Allocating and releasing configured volumes implies mounting and demounting physical volumes, which is a slow and easy to monitor information channel.

In this appendix, we have not tried to propose any new solution to the

implementation of the AIM. We have not tried to design a new mechanism for better implementing the military security controls. We have simply attempted to show that the restrictions imposed by the AIM in a state of the art system, the problems of implementing the AIM and the solutions to these problems can be handled by the type extension concept. Whatever can be done in NSS can be done in MSS. And MSS is not particularily helpful to implement the AIM. Type extension neither helps nor hinders sealing off storage channels. Type extension and the military security controls are orthogonal issues.

Biographical note.

Philippe Janson was born in Brussels, Belgium on 7 December 1949. He attended high school there, graduating from the Athenée Robert Catteau in June 1967. He entered the Université Libre de Bruxelles in September 1967, receiving the degrees of Candidat Ingénieur Civil (June 1969), Candidat en Sciences Mathématiques (October 1970) and Ingénieur Civil Mécanicien-Electricien (June 1972). His major field of interest then was electronics. He received a Harkness Fellowship from the Commonwealth Fund of New York in September 1972, which permitted him to study Computer Science at the Massachusetts Institute of Technology, where he received the degrees of S.M. (June 1974) and E.E. (February 1975).

During the summer of 1974, he was a staff member at Project MAC (now Laboratory for Computer Science), working on the development of the Multics system. In September of 1974, he became a research assistant at Project MAC, working in the area of system design, which is the subject of his doctoral thesis.

He is a member of the Association for Computing Machinery and of the Association des Ingénieurs de l'Université de Bruxelles.

In December 1971, he married the former Catherine Rolin. The Jansons have 1.9 children, Perrine and X.

Publications.

"Study and discrete simulation of the scheduler of the CDC SCOPE 3.4 operating system", (French), Engineering Dissertation, U.L.Brussels (June 1972).

"Removing the dynamic linker from the security kernel of a computing utility", S.M. and E.E. thesis, M.I.T. & MAC-TR-132, MIT Lab. for Comp. Sc. (June 1974).

"Dynamic linking and environment initialization in a multi-domain process", Proc. ACM 5th Symp. on Oper. Syst. Princ., ACM Oper. Syst. Review 9 5, pp 43-50 (Nov. 1975).

"Validating the protection mechanism of a computer system", IRIA Workshop on Protection and Security in Data Networks, Paris (28-30 June 1976).