M. I. T. Laboratory for Computer Science

Computer Systems Research Division

1 March 1976

Request for Comments No. 108

Reflections and suggestions about storage systems design.

by Philippe Janson.

RFC 99, which was my thesis proposal, outlined some ideas about using type extension to structure the virtual memory mechanism of a shared computer utility. My present task consists of demonstrating the interest of this method by applying it to the design of a real life storage system (SS). I have chosen to design a system that will have a functionality very similar to that of the new Multics storage system (NSS). The first step towards this goal consisted of performing an study of the functionality and the current implementation (28.0) of NSS, a task to which I have devoted the past two weeks.

This RFC deals altogether with three objectives that are summarized here:

1. I report on features and details of NSS that I specifically do or do not like from the point of view of modularity and structure;
2. Stepping back from the specifics of NSS, I discuss a few facts about the design of a SS that seem to make it more or less understandable.
3. I occasionally mention some ideas about the redesign of NSS.

---

0. Foreword.

Before I proceed to discuss and criticize NSS, I would like to express here my enthusiastic and encouraging feelings to the NSS crowd in general and to Andre Bensoussan and Bernie Greenberg in particular. Regardless of all the "nasty" comments I may make in this report, I think NSS is a great system, a major step forward in the achievement of a more modular and better structured Multics. Most of the "bad" features about NSS are leftovers from the old SS or dead ends the NSS gang was forced into by hardware constraints. Good features about NSS are its structure and its reliability as compared to the old SS. While the structure is not yet flawless, much progress has been made: in particular, the separation between directory attributes and vtoc attributes has removed the main reason why segment control depended on directory control. This separation has also enhanced the reliability by making it possible to back up and salvage one disk pack at a time.

1. Problem statement.

The problem I am trying to solve consists of making the tasks of maintaining, verifying and in general understanding a SS much simpler than they are today.

The solution consists of subdividing the system into several modules implementing different abstractions so that understanding any of the abstractions does not require understanding the whole system. This subdivision is effective only if:

1. the modules resulting from it are significantly simpler than the original system (If they were not, very little would be gained by subdividing the system.);
2. the modules are easy to identify and actually distinct (If their boundaries were fuzzy or if their interactions were not explicitly defined, it would

be hard to deal with any module without having to concern oneself with its neighbors.);

3. the interactions between the modules suggest a structured organization based on a partial ordering dependency relation that can be analyzed systematically from the bottom up (If this were not the case, it would be hard to decide where to start a complete and methodical analysis of the whole system.).

Thus, one must organize the SS into a set of modules in which each module is sufficiently small and self-contained to be understood individually and the set of all modules is partially ordered to allow a global understanding of the system to be derived systematically from the understanding of its parts.

2. Causes of complexity.

A system may deviate from the ideal partially ordered organization both in structure and in modularity.

The most frequently mentioned deviations are violations of the system structure by dependency loops or upward dependencies. Yet, in practice, I did not find this to be the worst problem. There are not many instances of it in NSS and most of these instances do not pose any major understanding problem.

On the other hand, NSS contains innumerable violations of modularity by shared data bases or hidden module interactions. Most of them indeed make understanding the system much harder. Some of them result in upward dependencies. However, the main disadvantage associated with these is not the upward dependency as much as the violation of modularity.

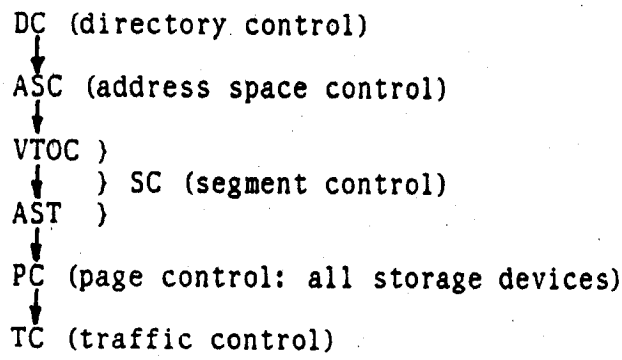
As a guideline for the remainder of this RFC, I define a module to be a collection of procedures and data bases that are related to one another in that they implement together one specific abstraction, and that are isolated from other procedures and data bases in that no procedure of the module

references any data base outside the module and no data base in the module is ever referenced by any procedure outside the module. As an isolated collection of procedures and data bases, a module realizes a protected subsystem. This subsystem can be run in an isolated domain in every user process or or in isolated domains of certain dedicated system processes.

Thus, interactions between modules in an ideal system should be limited to inter-module (inter-domain) calls and inter-process communication (ipc) messages.

With the above definition of modularity, a module depends on another module only if it calls the other module or if it sends the other module an ipc message and then goes blocked, waiting for an answer in a non-quiescent state (e.g., with a lock set or with some data base in an inconsistent state).

Figure 1 represents the perfectly structured and modular organization that I found to be closest to the actual organization of NSS in the sense that it points out the fewest violations of modularity and structure in NSS.



N.B.: volume management and backup control are not yet implemented in in the system version that was examined (28.0).

figure 1

3. Upward dependencies.

a. Old problems:

NSS has not eliminated two major problems causing violations of the structure

of the Multics SS:

-- the TC-PC loop due to loading and unloading processes. Thanks to Dave Reed, we now know how to solve this problem. The TC module has been split into two modules: the higher level module depends on PC to load and unload processes and PC depends on the lower level of TC to multiplex the processors. This lower level TC module manages only loaded processes and therefore never calls upon PC (see RFC 66).

-- PC-SC-DC dependency due to the management of quota. When it creates a new page, PC increases the number of used records in the appropriate quota cells. However, these quota cells are kept in the ASTEs of the superior directories of the segment being grown. This creates a dependency on segment control. Furthermore, SC manages information about the directory hierarchy because it is needed for the proper management of quota.

NSS also has failed to eliminate two minor problems of upward dependency in the Multics SS:

-- the ASC-DC dependency due to the use by SC of the directory entry pointer stored in the KSTE of every known segment to recalculate access information on a seg_fault. However, these problems would be easy to fix by reflecting seg_faults as high as DC.

-- the SC-ASC dependency due to disconnection of SDWs. This problem could be fixed by regarding SDWs as objects belonging in SC (after all, they are capabilities for segments) rather than in ASC. and managing them at SC level rather than at ASC level as is currently done.

b. New problems:

-- While NSS has tried to remove the concept of the directory hierarchy from the physical addressing and the backup of segments, it has not totally succeeded with physical addressing. When a page fault is taken on a null page

and the page cannot be created for lack of room on the physical volume (PV) where the segment to be grown resides, PC invokes ASC to disconnect the faulting user's SDW, and restores the control unit. This causes SC to be invoked to move the segment from one PV to another. SC must update the appropriate DIRE to reflect the move because the pvid and vtocx are kept there. For this purpose, it uses again the directory entry pointer that is stored in the KSTE describing the segment to be moved in the process on behalf of which it is moved. This is called the "out of PV" (oopv) problem.

-- Unless the system programmers are careful as to how they use the information stored in the directories and logical volume registration records, they might introduce a dependency loop between DC and logical volume (LV) control when they implement the concept of a master directory. Directories use LV information (lvids) to force their inferior segments to be grouped in identical LVs. On the other hand LV control uses directory information (pathnames) to restrict what directories can store their inferior segments in a given LV.

As a minor problem, NSS has not produced as clean a SC module as possible:

-- The boundary between AST management and VTOC management could and should have been drawn more clearly. I believe a lot of clarity could be gained by splitting SC into two modules of half the size.

-- The VTOC part of SC calls upon ASC and DC to walk its way up the hierarchy when updating quota information as a result of truncating an inactive segment.

4. Violations of modularity.

NSS contains many violations of modularity. Since most of them are benign and uninteresting, it would be of little use and otherwise impossible to list them all here. Most of these violations may be easy to fix. However,

they make understanding the system a non-trivial task. I will list here only a few instances of such violations for illustrative purposes.

Having carefully studied NSS, I claim with confidence if not conviction that the adoption of a systematic object based approach for the design of a SS would help avoid most of the violations of modularity because they are more often than not the result of using functional abstractions (FA) instead of data abstractions (DA). FAs make accidental violations of modularity easy in two different ways analysed below.

a. Hidden module interactions via shared data bases:

By coding programs as functional abstractions, one fails to look at data bases as objects that should be managed by dedicated modules. As a result, hidden module interactions violating modularity appear through the sharing of such data bases by programs belonging in different functional modules. Hidden module interactions are undesirable because they make the identification of individual modules and the understanding of the system more difficult.

-- Example 1: several modules touch one data base:

A blatant example of this situation is the sharing of the AST by SC and PC. Not only does the AST contain SC (ASTEs) and PC (page tables) information but PC even references the SC information to provide SC with usage information for segment deactivation purposes. Another example of this situation is the sharing of DIREs by DC and SC. DC programs and the segment_mover (oopv condition handler) are both concerned with the pvid and vtocx of a segment. All these programs touch the directory where the information is kept.

-- Example 2: one module manages data bases that could really belong in different modules:

PC manages the core map, the PD map and the FSDCT, which could really be managed separately for clarity. Core blocks, PD records and secondary storage

should and can be regarded as distinct resources that must be managed by distinct modules.

-- Example 3: some programs touch so many data bases that it is impossible to decide which module they belong in:

The home of fs_get is as vaguely defined as the semantics of the English word "to get". This programs "gets" almost anything from almost anywhere (e.g., it gets KST information given a segment number, directory information given a pathname, segment numbers given reference names, etc...). Notice that it cannot be regarded as a functional abstraction usable at various levels. It has internal state information in the sense that all the functions it supports know about the format of the data base they were designed to manipulate. The program is really a cluster of functions any one of which is usable at only one level. It should be split into several distinct programs every one of which belongs in a different level.

b. Interactions over locks:

The concept of locking implies the existence of a data base shared by parallel processes. According to the earlier definition of modularity, in all processes that are synchronized by a given lock, the specific domains that access the lock must be regarded as parts (instances) of the same module. These domains do indeed share access to the lock, which is a special case data base, and they interact with each other's course of execution. Such interaction is not legal at module boundaries. It can happen only within a module.

Miraculously, NSS contains no instance of two modules interacting over a lock. However, this miracle is not the result of a design aimed at respecting modularity. It is a side-effect of the overall locking strategy that is used to avoid deadly embraces.

The Multics locking strategy is based on Bensoussan's algorithm. All the locks a process may want to wait on at any time are ordered. The implementation of the system must verify the deadlock prevention rule: a process is allowed to wait on a lock x only if the relation x<y holds where y is the lowest of all the locks currently set by the process. In particular, the locks used by NSS are ordered as follows:

user_dir_l>proj_dir_l>udd_l>AST_l>page_tables_l>traffic_control_l

DC SC PC TC

As the reader may notice, the ordering of the locks is in parallel with the ordering of the modules that manage them, as I suggested in figure 1. Thus, it is not by accident and not for the sake of modularity that modules do not interact over locks. It is only for the sake of the deadlock prevention rule. If a module did interact with another over a lock, not only would modularity be violated but the chances are high that Bensoussan's rule would be violated. At the very least, it would be necessary and extremely difficult to verify that it is not.

In fact, this fortunate accident about NSS points out an interesting conclusion about the relation between type extension and locking stategy. In Multics, the locking strategy has to be consciously respected and enforced by every individual programmer. This requires the programmer of a given module to know exactly what locks are set by modules calling his module and by modules his module calls, a very undesirable requirement indeed. With type extension, the whole problem goes away. Since type extension implies modularity, there can never be any lock shared by two modules. Furthermore, since type extension guarantees a partially ordered dependency structure (1) ,

---

(1) Notice that a module A depends on a module B if A somehow transfers control to B and expects control to return after a transaction or leaves a

it imposes a partial ordering on the locks that each module can wait on or set. In other words, type extension guarantees an automatic enforcement of Bensoussan's algorithm (1) at the level of the system. Of course, within an individual module (e.g., DC), it is the responsibility of the designer of that module to enforce Bensoussan's rule among the locks used by that module (e.g., directory locks).

Using type extension also has side-effects on the use of locks. In NSS, the paging function and a piece of the quota function are implemented by PC. Thus, when taking a page_fault that requires accounting for quota, by locking the page_tables_1, one automatically locks everybody out of PC and any SC function that involves touching page tables. In a system based on type extension, it may be tempting to regard quota cells as a separate type of objects. Under this conception, a "quota_fault" would be managed by the quota manager. The quota manager would not have access to the page tables lock maintained by PC and therefore, could not lock everybody out of SC functions that involve touching -- in particular deleting -- the page table of a segment. Thus, the programmer of the quota module should expect to encounter changes in the world between the moment it is asked to process a quota fault and the moment it actually asks PC to create a new page. In fact, this is not a real problem. Similar situations already exist in NSS. For instance, when a boundfault is taken, SC must first verify that it still is a boundfault, i.e. that someone has not already moved the segment page table or increased

---

lock set in A while control is in B, which is synonymous with expecting to regain control later to unlock the lock.

(1) In fact, type extension even guarantees the enforcement of Haverty's algorithm. This algorithm imposes a constraint on the locking strategy that is stronger than that imposed by Bensoussan's algorithm: a process cannot wait and cannot set a lock that is higher than the lowest lock it has currently locked.

its maximum length by the time the fault must be processed. Similarily, when an oopv situation is handled, the segment_mover first verifies that there still is an oopv situation, i.e. that nobody has released any record in the meantime.

5. Use of parallel processes.

In NSS, all parallel instances of a module are subsystems in user processes. All these instances implement the same task. However, this is not a rule. It only happens to be so for the moment. Andy Huber has proposed a design of PC in which the page removal algorithm is implemented in a dedicated system process while every user process handles page faults for itself. Applying Andy Huber's design of PC results in an implementation where most of the parallel instances of PC are subsystems that handle page_faults in user processes and some distinguished instances are dedicated to implementing the page removal algorithm sequentially in a system process.

In addition to serializing some tasks (e.g., page removal), the use of dedicated system processes to implement special instances of some modules has another advantage. It makes possible the use of ipc messages between modules. The advantage of ipc messages over inter-module calls resides in the possibility for a low level module to take the initiative of a transaction with a high level module. With the closed call-return protocol of inter-domain communication, this is impossible because a low level module can never expect a high level module to return, so that calling it would violate the dependency structure. However, if a low level module wants to transmit information to a high level module and does not expect an answer, it can use the bottom level ipc module to send a message to a system process dedicated to running a parallel instance of the high level module. This mechanism is similar to the use of upward transfers (open calls) by the transfer vector

that intercepts processor exceptions in Multics. In fact, upward transfers expecting no matching returns could be simulated in user processes.

For instance, in Dave Reed's implementation of TC, the low level module can send messages (interrupts) to the high level TC module whenever virtual processors are stopped and can be unbound. In NSS, every instance of SC in a user process will be able to send messages to the backup daemon whenever the queue of segments to be backed up will have grown beyond some threshold. The use of such a queue together with ipc messages is what allows NSS to eliminate one upward dependency that existed in the old Multics SS, namely the fact that SC depended on DC to propagate the date-time-modified of a file up the directory hierarchy. To implement quota control at a level higher than PC, one could conceive having the PC module send messages to the quota control module in a dedicated process, and then simply unlock the page_tables_1 and go blocked. It would then be up to the quota control module to make its resource allocation policy decision and eventually call PC to actually allocate or free pages. In general, messages can be used to signal low level events to high level modules in user processes.

6. Information levels.

An interesting question about the design of a SS is: can information relevant to one level be stored at another level? Much of the information treated in a SS is mapping information. (e.g., pathnames, uids, segnos, etc...). In general, a given piece of information is (or at least should be) meaningful only to the one module that manages it. The question is: can the piece of information be stored at a higher or a lower level?

The answer is yes, although there are certain rules to be respected to preserve the structure of the system.

a. Storing information at a lower level.

Any module that accepts requests from higher level modules to store information into a data base or an object it maintains may never interpret or depend in any way on this information. This would cause an upward dependency. If this rule is obeyed, it is possible to safely store high level information at a low level. In fact, this is what happens in a system supporting demand paging. Users can safely store their information into pages because it can be proved that PC does not depend on this information for its own operation. (That PC deallocates null pages means that its operation is influenced by user information, which is normal since PC provides a service to store user information. However, it does not mean that its correct operation is dependant on user information. PC regards a null page as an order to do something but how it does it is not influenced by the content of the page.) Similarily, in NSS, it is safe to store pathnames in VTOCEs because the vtoc_man never uses these names. They are uninterpreted strings at that level. They are stored there by DC for the sake of the directory salvager, which, when implemented, will extract them to reconstruct the directory hierarchy if it is damaged. They are used as "backwards bindings" for reliability but they are not necessary in the normal course of operations.

b. Storing information at a higher level.

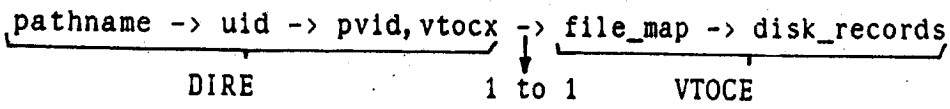
Any module that releases mapping information it interprets to higher level modules must be aware of the potential difficulty to reverse the binding it so creates. When the SC module of NSS releases the pvid and vtocx of a segment to DC, it creates a hard to reverse binding. The dependency between SC and DC results only from the fact that the segment_mover reaches up in the appropriate DIRE to reverse a binding it has created earlier. Mapping information for low level objects may be released to high level modules only if:

-- there exists a dependable mechanism for the low level interpreting the information to invalidate (revoke) or catch (inhibit) an obsolete binding and -- it is possible to reverse the binding without causing upward dependencies, i.e. when the user of a binding is notified that the binding is obsolete, it can cause it to be reversed by invoking the lower level module.

7. Mapping and "caching" functions.

When analysing a SS, one reaches the conclusion that it performs two main functions: mapping and "caching". The mapping function consists of maintaining bindings between various levels of identifiers. What I have named the caching function consists of establishing in parallel to mapping bindings equivalent bindings that permit faster access to the object they denote. The word "caching" comes from the analogy with the hardware cache technique, which binds the address of a word to a location in the memory and a location in the cache.

Let us examine NSS to understand more about mapping and caching. In a hypothetical quiescent state, NSS establishes the following mapping bindings:



This sequence of mappings binds a file as denoted by its pathname to the set of disk records that compose it. In an active state, NSS adds several cache bindings in parallel to the above mapping bindings to speed up references to the file by by-passing bindings that are kept in a hard to use format or on a

slow device:

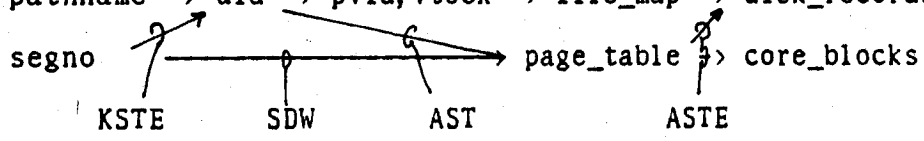


figure 2

Thus, an ASTE appears as a cache for a VTOCE and an SDW is a cache for a KSTE, which is itself a cache for a DIRE. In order not to confuse the above figure, I have omitted some more cache bindings like the SDWAM, the PTWAM, and the PVT (cache for the disk_table).

The identification of each mapping and caching binding appears to be fundamental to the modularization and the structure of a clean SS design as will be seen in the next section.

8. Modularization and structuring of a SS.

The identification of every different kind of binding in a SS can be used as a guideline to modularize the system. In an object based system, every binding is established by some kind of identifier that can be regarded as denoting an abstract object. Identifiers of identical format and meaning denote abstract objects of the same data type. This is not to say that one must regard every different kind of binding as corresponding to a different data type. However, unless the management of some kind of binding is trivial, I believe there is a good reason to want a different data type for every different kind of binding so as to manage every binding separately.

The identification of bindings is even more important to structure a system than it is to modularize it. I had reached this conclusion earlier but it was confirmed by my study of NSS. Once it is decided what module is

responsible for what binding, a graph such as that of figure 2 provides an excellent handle to organize the system into a partially ordered structure, i.e. to establish the dependency graph between the various type managers. The graph of figure 2 suggests a partial ordering based on the binding relation of the various identifiers manipulated by the system. This partial ordering is a parallel of the partial ordering, based on the dependency relation, that exists between the various modules of the system.

An interesting remark can be made about the relation between the manager of a mapping binding and the manager of a parallel caching binding. Although it is not a rule, one notices in practice that if a dependency relation exists for some functional reason between the two managers, it is usually the mapping binding manager that depends on the cache binding manager because, in general, the mapping binding manager wants to take advantage of the cache function to access its own data bases faster. In NSS, DC depends on ASC (because directories themselves must be in the cache to be accessible). Similarily, VTOC control depends on AST control (because VTOC control manages quota cells, which are accessible only in their encached version, in the AST). One could extend the cache analogy to Dave Reed's implementation of TC. Virtual processors can be regarded as caches for user processes to access (schedule and run) them faster. Again, the process manager depends on the virtual processor manager because it is itself implemented as a dedicated virtual processor.

Yet, the cache analogy must be used with care and cannot be stretched too far. What I have named "caching" is not strictly identical to the phenomenon observed with hardware caches.

-- With hardware caches, the same address can be used to access both the slow memory and the cache because one can afford to access the cache by association

on the slow memory address. With software caches, different identifiers are used to access the slow memory and the cache because an associative cache would be too expensive. For instance, the AST is accessed by ASTE pointer while the VTOCs are accessed by VTOC index as it would be too expensive to maintain a complete association between uids, and ASTE pointers and VTOC indices. Directory entries are denoted by pathnames while known segments are denoted by segment numbers as it would be too expensive to maintain a complete association between segment numbers and pathnames.

-- With hardware caches, it is desirable to have a fast and simple algorithm to decide what information can be stored or is worth storing in the cache. Thus, the encacheability of a piece of information is decided globally on the basis of the information container that contains it (e.g., the segment). With software caches, one can afford more sophisticated algorithms to decide whether some piece of information can and shoud be encached. The encacheability can be decided selectively on the basis of the information itself rather than on the basis of whole information containers. For instance, knowing that the time-record-product of a quota directory is recomputed only once for each phase of activity of the directory, it would be a waste to store it in the AST. Thus, it is made an "unenASTable" (due to B. Greenberg) VTOC attribute.