PROJECT MAC 2 December 1975

Computer Systems Research Division Request for Comments No. 97

A NOTE ON THE FINAL REPORT FROM SRI

by Philippe A. Janson

My interests have been drifting lately towards the study of virtual memory systems and storage systems, and methodologies for designing them. A relevant document on these topics is the final report recently issued by SRI on the design of their provably secure operating system. I have investigated the design of their virtual memory system with particular care and I have extracted from the report and summarized here for interested people a description of it. Less interested readers will probably find all they want to know about the system. More interested readers will be able to decide whether they want to read the report itself.

Foreword

The following document will attempt to describe the structure of the virtual memory (VM) mechanism contained in the provably secure operating system designed by SRI. It is hoped that our description will match as closely as possible the formal specifications (FS) presented in the SRI report. Any inaccuracy is of course to be blamed on us although it should be said that the report itself is unclear on a few occasions.

The report presents a very promising methodology for describing and implementing structured operating systems. To demonstrate the use of the methodology, SRI has applied it to the design of one operating system. Unfortunately, SRI has not tried to implement the so designed operating system. The design suffers from this lack in that it overlooks several feasibility and efficiency issues. After reading the report, we believe that the virtual memory implementation suggested in appendix B would indeed be impractical.

The VM system

The SRI system comprises 11 levels of AMs as summarized below. The following names try to capture what I feel is the actual function of each level.

level	manages	provides
0	interrupts, capabilities	address interpretation interrupt dispatching
1	addresses	address calculation
2	virtual processors	process switching
3	fixed-VM segments (permanently active)	paging, device control
4	user segments revocable capabilities	segmentation, revocation (paging?)
5	mapping of ETOs to REPs	extendibility
6	directories	name-uid mapping access control
7	user objects	lost-object problem handling
8	linkage sections	(link_man, etc...)
9	links	dynamic linking
10	processes	process scheduling

In this document, we will be concerned about levels 0, 3 and 4 which implement the VM. Levels 1 and 2 are of interest only as they are used by levels 3 and 4. Level 1 basically takes care of indexing and indirection in addressing. We can ignore it if we assume that all addresses uttered by levels 3 and 4 refer directly to the appropriate operands. Level 2 basically allows switching processors between scheduled processes, much like in Reed's proposed design, when waiting for a page to arrive in core.

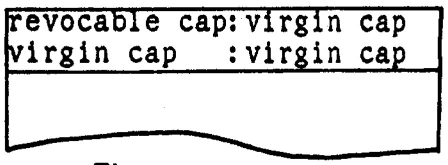
While the system seems well structured above level 4 thanks to the type extension mechanism provided at level 5, the lower levels of the system are not as well organized because they are not based on data abstractions. Each level manages more than one type of object (provided interrupts, capabilities, addresses, fixed-VM segments, devices and user segments are viewed as types) and many types are managed by more than one level. The hierarchical structure exists only through the functional structure provided by the AMs (see mapping functions in appendix B) but does not correspond to any hierarchy of extended types. This functional hierarchy leaves several mechanisms undefined because the FS are interface oriented and not enough inside-mechanism oriented. They fail to convey information about the internal operation of the system. For instance, a page is not recognized as an object per se and therefore, there is no AM dedicated to the management of pages. Thus, concepts like paging and resource control, which are attached to the concept of a page, are regarded as mechanisms internal to various levels and are not specified at any interface of any specific level. It is impossible to understand or to prove anything about these mechanisms as they are not described in the FS but rather left to the discretion of the implementer.

Level 4 is claimed to contain a backup mechanism but this does not show through the FS either and it is not described in the report.

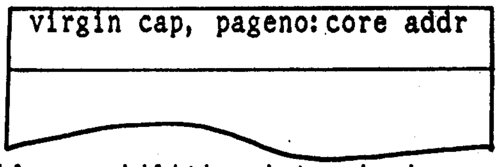
<u>LEVEL 0</u>

--Data bases

cap_map



addr_map



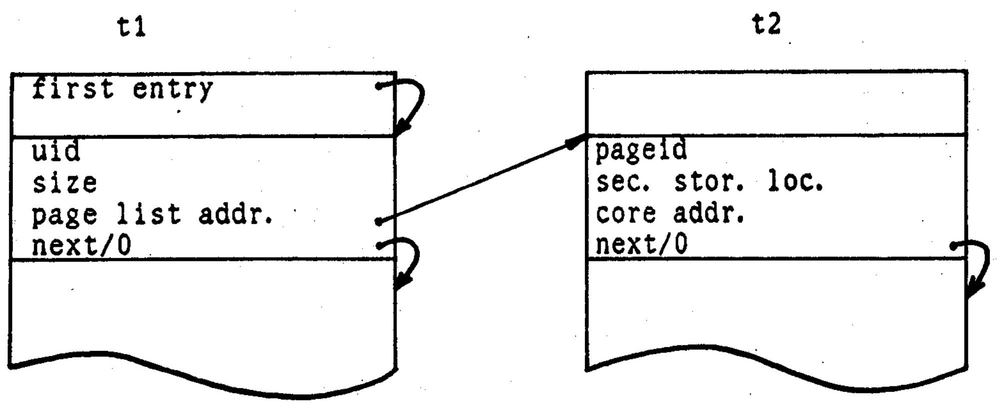
The cap_map maps revocable capabilities into virgin capabilities that directly point to the object they denote (see Redell's revocation scheme). Virgin capabilities are mapped into themselves. The addr_map maps any in-core page to its absolute address. The two maps are core resident. If a particular capability is not in the cap_map, a cap_map fault occurs. It is our guess that cap_map faults are directed to level 4 as level 4 is the repository for all capabilities, even though level 0 is the official capability manager. If a page is not in core, an addr_map fault occurs that is directed to level 3 (or level 4 for user segments?). It is not immediately clear which AM handles I/O for page faults because the fault/interrupt side of an abstract machine (AM) is never specified in the FS of the AM. Such interfaces are only loosely mentioned in the accompanying text but never described. Some paragraphs suggest that level 3 handles paging I/O (pp 0-4, A.3-1). Other paragraphs suggest that both levels 3 and 4 are responsible for it (pp 5-18, 7-5). Also, the FS do not embody concepts like a wired page. Thus, it seems unclear how one would prove that page control programs, wherever they are, do not take page faults. The text does state that certain pages must remain in core at all times but this does not show through the FS.

--Primitives

The primitives comprise hardware functions to add and delete entries in the two maps, extract information from the various fields of an entry, generate a new capability, and move information from one core word to another (processor registers are part of core). Level 0 primitives are actually accessible only to level 1. But level 1 has primitives that directly map into level 0 primitives after address calculation. Thus, we may regard level 0 primitives as accessible to higher levels.

LEVEL 3

--Data bases



T1 contains one entry per fixed-VM segment. T2 contains one entry per fixed-VM segment page. T1 and t2 are core resident. Thus, fixed-VM segment are always "active" in the Multics sense but their pages are not wired down. T1 and t2 basically corresponds to a wired down AST with linked entries for all the segments that have to be permanently active but can be paged in/out of core. Four such segments will be used by level 4 to contain the system map and the tree of revocable capabilities.

N.B.: pageids are not used anywhere in the FS.

--Primitives

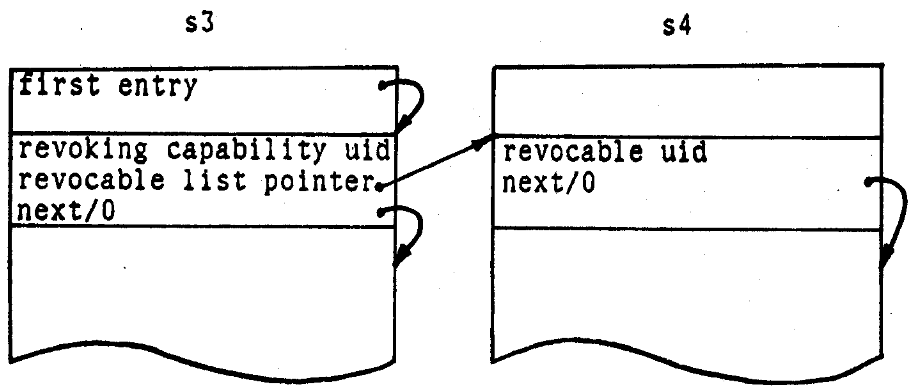
There exist primitives to read and write fixed-VM segments, and to set/get their size. There also exist primitives to create and delete fixed-VM segments (p A.3-3). This sounds reasonable but contradicts the text of the report (p 5-17), which says that there is only a fixed number of fixed-VM segments.

LEVEL 4

--Data bases

Level 4 manages two data bases, s1 and s2 of formats identical to those of t1 and t2 respectively. S1 is the system map. It describes all user segments. S2 is an AST that compounds page tables and file maps of user segments. As s1 and s2 are fixed-VM segments, their pages are not core resident.

Two more data bases are used to store the tree of revocable capabilities.



--Primitives

Level 4 primitives comprise functions to read, write, create and delete segments, to change their size (explicit call), to chase down a virgin uid given a revocable capability, and to create and revoke revocable capabilities.

Conclusion: problems

To conclude our discussion, we presume that SRI realizes that more work is required to produce a design that would lead to an efficient and practical implementation. The above design is oversimplified and not very realistic as illustrated by the examples in the next two paragraphs. SRI has demonstrated the use of the design methodology on a "paper" system but the effectiveness of the method remains to be shown on a real system design.

1) Assume that the 256th page of a segment s is not in core. An addr_map fault will occur when it is first referenced. Since the file map for the segment is in s2, a fault handler must be invoked presumably at level 4. It will look in s1 to find the address of the file map of s. This may cause a second addr_map fault, on s1 this time. Then, the linked file map will be followed down 255 links (potentially causing a flury of addr_map faults) before the original fault will be resolved. Finally, execution can proceed ... until the next fault, which is bound to happen soon since the previous flury of faults has substantially increased the working set of the user program, which, in a system with limited core resources, would cause most of that working set to be thrown out of core.

2) It seems as though addr_map faults on level 4 segments must be directed to level 4 and faults on level 3 segments must be directed to level 3. Thus, there must exist local paging device management functions at each level since the interface of level 3 does not seem to provide such functions to level 4. Unless the paging device cannot be shared across levels 3 and 4, there has to be one paging device map accessible to both levels. This defeats the whole idea of hierarchical dependancy.