FORMAL PROPERTIES OF RECURSIVE VIRTUAL MACHINE ARCHITECTURES.

Gerald Belpaire* New York University

Nai-Ting Hsu** University of Wisconsin

A formal model of hardware/software architectures is developed and applied to Virtual Machine Systems. Results are derived on the sufficient conditions that a machine architecture must verify in order to support VM systems. The model deals explicitly with resource mappings (protection) and with I/O devices. Some already published results are retrieved and other ones, more general, are obtained.

Key Words and Phrases: virtual machine, virtual machine monitor, architecture, operating systems, formal requirements.

CR Categories: 4.32, 4.35, 5.21, 5.22

INTRODUCTION.

The principles and functions of Virtual Machine Systems (VM) have been quite extensively described in the literature. In this study, the interest we have in them resides less in the ability to implement hierarchical structures of identical machines than in the theoretical and practical fall-outs of such implementations. In particular, the study of VM systems has uncovered interesting relations between the software structure and the host machine architecture. Based upon these findings, several new machine architectures were proposed for the design of VM systems [1, 3, 4, 5], and theoretical results were obtained on the sufficient conditions that the hardware should verify in order to support a VM system [6].

In this paper, we further investigate the properties of the hardware/software interface of the VM systems. We adopt a classical definition of VM systems and then proceed to examining (with the appropriate formalism) what machine architectures are needed to implement them. Some already known results are retrieved [6] and some new ones are obtained. Examples of architectures are discussed.

Our aim is to reach a better theoreti- understanding of the Virtual Machine "phenomenon", but also to provide practical guidelines for machine architects and system designers.

DEFINITION OF A VIRTUAL MACHINE SYSTEM.

A (non-recursive) VM system simply consists of a perfect multiplexing of the real machine, as on Fig. 1.

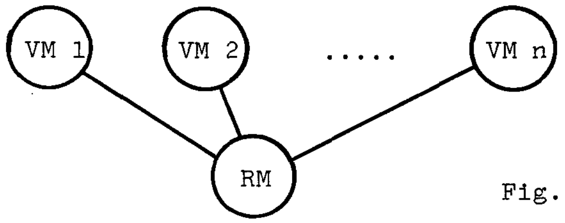

Fig. 1

The time-sharing aspect of the multiplexing is secondary; what is important is the concurrent co-existence of several virtual machines. This multiplexing includes the scheduling of the CPU, but also the sharing of the memory and of the other resources of the VM. Classically, this will be implemented by a software nucleus called Virtual Machine Monitor (VMM). The VMM also implements some facilititіes that are the equivalent, for the Virtual Machines, of the hardware starting and halting functions of the Real Machine.

Three essential characteristics must be possessed by such a VM system:
1. The execution environment of each VM is, from a logical standpoint, equivalent to the one of the Real Machine. Said otherwise, what runs on the RM can run on the VM.

* Department of Computer Science, Courant Institute of Mathematical Sciences. This work is supported by ERDA under contract E(11-1)-3077.

** Computer Sciences Department and Mathematics Research Center. This work is supported in part by the US Army contract DAHC04-75-C-0024.

2. The performance of a VM with respect to the one of the RM, is degraded only by the need for resource sharing.
3. The different VMs are protected by impassable walls, i.e. they cannot interfere with each other.

If the VMM can run on a Virtual Machine to generate another level of VMs, the system is a Recursive Virtual Machine System. Its structure is a potentially infinite tree of VMs. Each node of the tree must verify the same three characteristics of VM systems.

These definitions have been given, with minor differences only,in a number of paper on the subject. However the second characteristic as stated here is rather stronger than usual. In particular, it prohibits any interpretive execution of the instructions by the VMM. Interpretation can be rapidly catastrophic when the system has several levels of recursion, as its detrimental effect is spread throughout all the levels. Early implementations of VM systems have compromised on this point for practical reasons: no suitable hardware was available. Our present aim is in part to characterize the type of hardware needed to avoid these drawbacks. This is not to mean that no compromises should be made but that we want to evaluate at what price they can be avoided.

In order to examine such aspects more formally, we shall define in the next section an ideal real machine that will perfectly verify the three criteria of a VM system. This architecture is hypothetical, but, by imposing restrictions on its structure, we shall be able to determine under which conditions its properties are invariant, i.e. under which conditions the system remains a VM system.

THE INFINITE RECURSIVE ARCHITECTURE.

We first consider a simple machine architecture with one processing unit, one central memory, and one I/O peripheral. The primitiveness of this machine is taken for simplicity; it however presents the minimum set of features necessary to illustrate our point. It consists of (Fig. 2):
1. One CPU (Central Processing Unit) that we denote by π.
2. One I/O unit denoted by ε.
3. One central memory denoted by μ.

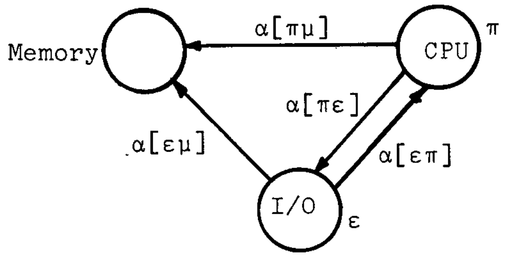

Fig. 2

Basic communication links exist between these devices. They are:
1.The CPU can perform memory accesses. This ability to address memory and to fetch data is denoted by $\alpha[\pi\mu]$.
2.In the same way the I/O device can communicate with the memory in some manner, be it through a channel or by means of an intermediary communication processor. This is symbolized by $\alpha[\varepsilon\mu]$.
3. The CPU has the ability to issue I/O commands either by specific I/O instructions or through reserved addresses of the memory. This is symbolized by $\alpha[\pi\varepsilon]$.
4. The I/O device can communicate directly with the CPU by means of interrupts : $\alpha[\varepsilon\pi]$.

The purpose of this notation is to represent the existence of such communications rather than the mechanism of their functions. For example, the fact that I/O commands can be done by memery accesses is irrelevant at this point, it will be detailed at a later stage, The different execution modes (kernel supervisor,..) will also be defined later.

We can now define an hypothetical machine (Fig. 3) made of an arbitrary (potentially infinite) tree whose nodes are the elementary machines defined hereabove.

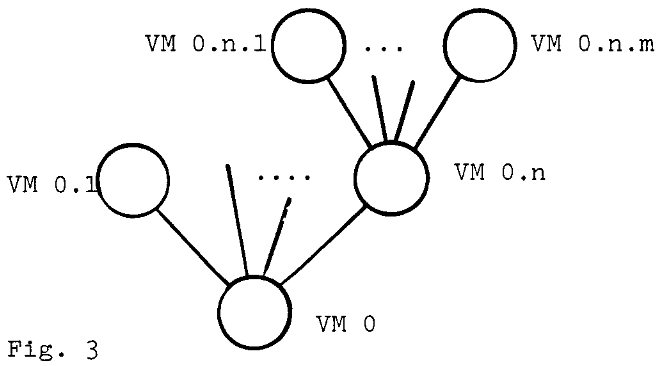

Fig. 3

A parent machine (i.e. a lower level node) has the ability to start and to halt its child machines. This provides for the only communication (in this architecture) between nodes belonging to different levels. It is easy to show that this architecture possesses the three characteristics of a VM system. Indeed:
1. The execution environment of all nodes are equivalent. This is fairly obvious since all nodes are identical and have the same abilities to start child machines and to stop their own execution.
2. All nodes have identical performances.
3. Each node is completely isolated from all nodes having the same parent. By isolation, it is meant that no interference of any kind is possible (not even a start or stop command).

We call this tree of machines the Infinite Recursive Architecture (IRA). Of course its properties were obtained by using an infinite number of resources and therefore the IRA does not deserve the Virtual Machine label. However, there is no way for a program to determine whether it runs on the IRA or on a perfect VM system. In that sense the IRA captures the notions common to all VM systems. Our task now is to examine how these notions and properties remain invariant while the number of resources is reduced.

In the IRA the only restrictions on the intra-node architectures is that they must be identical for each node. Beyond this, the architecture is completely independent of the VM system structure. In particular, no restrictions are made upon the memory mappings or on the processor instructions, provided that they are identical for each node. It therefore seems reasonable to state that only the reduction of the number of resources, which is effectively a mapping of all the resources of the VM system into one node, will be the cause of further imposed limitations on the intra-node architectures. These limitations will be necessary to enforce a proper working of the Virtual Machine Monitor, i.e., they give the VMM the power to control the resources of the system. The processes, other than the VMM, running on a VM can in principle use any form of protection, independently of the VMM form of protectiom. But this would impose that the VMM has different power than these processes, a fact not likely to be accepted because it would mean that virtualizable chines have a specific feature to support the Virtual Machine Monitor, different from the ones supporting the other processes. In conclusion, although the intra-node architecture is in principle not affected by the virtualization, the need to consider the VMM as any other process will limit the architecture*.

VIRTUALIZATION OF THE IRA.

Typically, a VM system with the structure of the IRA has the property that all the resources are mapped into the resources of the node VM 0 (the Real Machine). By mapping it is meant that each resource is available to a Virtual Machine only when it is mapped into a real resource: the mapping describes the allocation of real resources to virtual resources. When this allocation holds, the virtual resource is said to be realized.

Formally this will be defined by a time dependent mapping between the set of virtual resources and the set of real resources. This mapping has the following meaning:
1.For the memory, it is the usual virtual memory address translation.
2. For the CPU, it abstracts the status registers and the allocation of these registers to the VM.
3. For the I/O device, it specifies the device status words and the device buffers.

* An example of a VM design that may give in certain realizations different powers to the VMM than to other processes is the Hardware Virtualizer [4].

Whenever a given VM attempts to use a virtual resource while the latter is not realized, a VM fault occurs and is routed to the VMM of the parent machine. The VMM has the functions of properly handling VM faults and the allocation of VM resources. In some VM systems it also has the function of interpreting some instructions. For reasons explained earlier we momentarily assume that the latter function is non existent.

The proper handling of the allocation of VM resources is a problem of local program correctness of the VMM. We need only to guarantee that the architecture provides the appropriate features for the VMM to implement its functions. This is to say that the different resource mappings must be represented in the machine and accessible to the VMM only. The problem of VM protection cannot be solved by assumption on the correct behavior of the system as we must consider possible occurrences of errors and of malicious attitudes. The architecture must therefore include features that will render these "errors", even in the worst case, harmless.

We now proceed to the formal treatment of the virtualization and consider first the case where the CPU has only one execution mode.

<u>First case: One execution mode.</u>

We treat first an arbitrary pair of levels of Virtual Machines (Fig. 4). On this figure, $\underline{k} = 0.k_1.k_2....$ is an arbitrary node label.

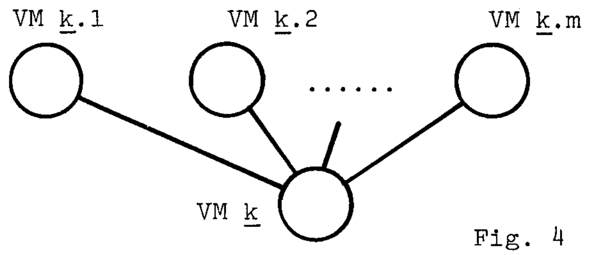

Fig. 4

Let f_t be the notation for the realization map (also called resource map [4]). It represents the mappings as they hold at time t.** To define f_t precisely, we have three cases to consider:

1. The memory.

$$f_t[\mu_{\underline{k}.i}] : \mu_{\underline{k}.i} \rightarrow \mu_{\underline{k}} \cup \{\omega\}$$

i.e. it is a mapping from the memory space of VM $\underline{k}$.i to the memory space of VM $\underline{k}$ and to the element ω.

** The time appears here as a parameter of the mapping, i.e. for different instants, there are different mappings. This representation is equivalent to a representation in terms of states and it was chosen for simplicity. For more details on the state vs. time duality, cf. [7].

To a virtual address x corresponds the location $f_t[\mu_{\underline{k}.i}](x)$ in $\mu_{\underline{k}}$ if x is realized. If $f_t[\mu_{\underline{k}.i}](x) = \bar{\omega}$ then the location x is not realized and a VM fault will occur when an access to x is attempted.

2. The CPU.

$$f_t[\pi_{\underline{k}.i}] : \{\pi_{\underline{k}.i}\} \rightarrow \{\pi_{\underline{k}}, \omega\}$$

If $f_t[\pi_{\underline{k}.i}](\pi_{\underline{k}.i}) = \omega$ then $\pi_{\underline{k}.i}$ is not realized and the machine VM $\underline{k}.i$ is not currently running.

3. The I/O device.

The definition is analogous to the CPU map :

$$f_t[\varepsilon_{\underline{k}.i}] : \{\varepsilon_{\underline{k}.i}\} \rightarrow \{\varepsilon_{\underline{k}}, \omega\}$$

We also define $\bar{\mu}_{\underline{k}}$ as the memory area of the machine VM $\underline{k}$, that is used by the VMM running on VM $\underline{k}$.

To define the recursive aspects of the VM system, we need only to take the functional composition of the mappings. For instance, if x is a virtual address for the machine VM $0.k_1.k_2....k_n$, the real location realizing x is:

$$f_t[\mu_{0.k1}] \circ f_t[\mu_{0.k_1.k_2}] \circ \ldots\ldots \circ f_t[\mu_{0.k_1.k_2....k_n}](x)$$

if this composition is defined, i.e. if no resulting value, intermediary or final, is ω. The mapping must be taken at the same instant t. This does not mean that it must be the same "clock" instant, but the same "system" instant, i.e. that between the first and the last effective mapping calculations the state of the system, from the viewpoint of the machine VM $0.k_1...k_n$ does not change.

At this state of our developments, two remarks can be made :
1. The virtual machine at one level is "virtual" for the lower level and "real" for the higher level. Mappings are nested as well as VM faults [4]. Similar observations were made for hierarchies of processes [2].
2. The CPU is preemptible and reusable, and for these reasons it can be shared on a time dependent basis. The core memory is also preemptible and reusable, but this is not true for the memory taken globally (core + secondary memory). Indeed to preempt secondary memory is inevitably to loose information. Therefore the memory is sharable only on a space dependent basis and it is impossible that each VM can be allocated the total (core + secondary) memory. In this sense, the virtualization considered hereabove is not acceptable since a VM cannot share the entire Real Machine. This is a rather theoretical point however, as in practice the secondary memory is considered as a memory sink rather than as a part of the Real Machine.

We now make the assumption that all accesses within a particular VM are made through memory locations. This is a quite realistic hypothesis as some actual implementations are working on this principle. It was also taken as hypothesis in other papers [4, 6, 1]. In this case it is easy to define formally the different accesses characterized earlier :

1. Let $m_{\underline{k}.i}$ $\mu_{\underline{k}.i}$ be the set of virtual memory locations used for the regular memory accesses by the CPU and the I/O device. They correspond to the accesses:

$$\alpha[\pi_{\underline{k}.i}\mu_{\underline{k}.i}] \text{ and } \alpha[\varepsilon_{\underline{k}.i}\mu_{\underline{k}.i}]$$

2. Let $s_{\underline{k}.i}$ be the set of virtual addresses by which the CPU controls the I/O device (by the access $\alpha[\pi_{\underline{k}.i}\varepsilon_{\underline{k}.i}]$).

3. Let $v_{\underline{k}.i}$ be the set of (virtual) interrupt vectors. (access $\alpha[\varepsilon_{\underline{k}.i}\pi_{\underline{k}.i}]$).

The protection requirements (third characteristic of a VM system) imply:

1. For all pairs i, j, at any time t, and for each node VM $\underline{k}$,

if $r_{\underline{k}.i} = m_{\underline{k}.i} \cup s_{\underline{k}.i} \cup v_{\underline{k}.i}$,

$$f_t[\mu_{\underline{k}.i}](r_{\underline{k}.i}) \cap f_t[\mu_{\underline{k}.j}](r_{\underline{k}.j}) = \emptyset$$

This means that the address spaces of two VMs having the same parent are mapped into disjoint address spaces at any time t .
This property is enforced by the VMM program, i.e. it is a condition of correctness of the VMM.

2. The mapping f_t must be represented in the machine in order to be accessible to the VMM and to the VMM only. A sufficient condition is that it resides in the virtual address space of the VMM.

Let $\rho(f_t)$ be the notation for the representation of f_t , i.e. $\rho(f_t[\mu_{\underline{k}.i}])$ is a set of memory locations.
We can formulate the following theorem to summarize the preceding conditions :

Theorem 1.

A sufficient condition for a proper working of a VM system of the type defined hereabove is :
1. The representation of the realization map of each node resides in the address space of its VMM : $\rho(f_t[\mu_{\underline{k}.i}]) \subseteq \bar{\mu}_{\underline{k}}$ at all time t
2. If the mapping is not defined when used at time t , it generates a VM fault in the VMM address space.
3. In the sense defined hereabove, the VMM is correct.

The situation is schematized on Fig. 5. The large circles are the VM address spaces, with the realization maps represented by the arrows between them. The double circles are for the representation of the realization maps; they must reside in the VMM virtual address space of the lower level. To be complete the figure should be extended to a tree of virtual machines.

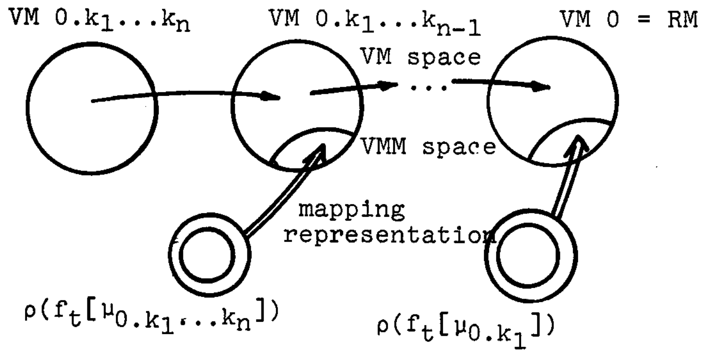

Fig. 5

Because the representation of the realization maps are residing in virtual memory, an address calculation by means of the mapping can cause a VM fault if the representation is not realized at a lower level. It is evident that such a design, although secure in principle, will cause a tremendous overhead as it forces the address calculation to be done in virtual memory. The conditions given earlier are thus minimal theoretical conditions. "Accomodations" must be made to render them practical.

One solution that was taken as basis for a number of proposed solutions is to force the representation of the realization maps to reside in fixed physical locations and to make it time independent. This means that the mappings are not in virtual memory anymore and that they remain constantly in the same location. Because only one node of the tree of VMs can be active at a time (since there is only one CPU),only one path of mappings must be available at that time. The inactive mappings can therefore reside anywhere in the virtual memory of the VMM.

The simplest solution using these principles is the Hardware Virtualizer (HV)[4] . In this scheme, the mappings representations are in virtual memory and a hardware register (the Virtual Machine Identifier Register, VMID) indicates how to access them. This design however generates a problem: how to protect the VMID. Goldberg describes in his paper a way to remedy this problem.

In Lauer's proposal [5], the mapping representations are also in virtual memory, and are accessed through a stack of registers (the Display). A register containing the currently realized level of VM ensures that only the appropriate registers of the Display are accessed by the VMM. Only the active mappings are pointed at by the Display.

A more recent proposal [1] generalized the preceding ones by using a stack of registers representing the mappings (multiple copies of RMRs, Resource Management Registers). Here too a level register ensures proper accesses to the RMRs.

At this point, the solution to adopt is determined only by efficiency considerations.

Before considering the case with two execution modes, several remarks can be made one the preceding results :
1. The virtualization does not require any interpretive execution that would not be needed on the Real Machine, if one excepts the resource mapping calculations.
2, No particular paging or segmentation mechanism is suggested for the memory management.
3. The program counter is considered to be an intra-VM feature, working only in virtual memory.
4. All I/O accesses must be made by virtual addresses. In particular no instruction such as the RESET on the DEC PDP 11 is tolerated.
5. When an interrupt occurs, it will be routed to the appropriate VM by the memory mapping, provided that the VM node label is memorized in a location associated with the I/O device (i.e. the interrupt vector is one or several virtual addresses + a VM node where they are accounted for). If any interrupt mask is set (also done through some virtual memory address), the interrupt will be appropriately inhibited for the corresponding machine. However, interrupts are accesses of a particular kind as they must be acknowledged (if not masked). This means that a VM currently being realized on the CPU might be interrupted to serve another VM. This situation is abnormal with respect to the VM system characteristics (stricto sensu, a VM interferes with another one: a violation of the protection), and it led to suggestions that other mechanisms than interrupts should be devised [5].

Second case: Two execution modes.

We assume that the CPU can be in two execution modes : privileged (P) and non-privileged (NP) mode. Typically in the P-mode the CPU can execute special instructions not executable in the NP-mode. We represent this by defining in the P-mode accesses of the machine that are different from the ones in the NP-mode. Indeed the privileged instructions are usually used for specific accesses to the processor status, the memory management registers, or the I/O devices. The effects of these accesses is not as important as the ability of performing them, at least for our present purpose.

We keep in this section all the previously defined concepts and objects and we assume that they denote the non-privileged mode features. To represent the privileged mode features, we simply use the non-privileged mode notation with a star. For example:

$\pi_{\underline{k}}$ is a virtual CPU in the NP-mode

$\pi^*_{\underline{k}}$ is the same CPU in the P-mode

We also define the accesses by which the CPU cam modify its own execution mode:(e.g. by loading new virtual status words):

$\alpha[\pi_{\underline{k}}\pi_{\underline{k}}]$ and $\alpha^*[\pi_{\underline{k}}\pi_{\underline{k}}]$ for the VM $\underline{k}$

The access $\alpha[\varepsilon\pi]$, which was to represent the effects of the interrupts, had, in the case with one execution mode, the effect of branching the processor to a new instruction in the virtual memory. With two execution modes, the interrupts might have the effect of changing the execution mode. This can be "automatic": e.g. after an interrupt the CPU is always in the P-mode, or it can be done by fetching a new processor status word specifying the new mode. In this paper, the latter situation is assumed to hold.

We also assume that all accesses are made through memory locations as for the case with one execution mode. This gives the following virtual memory areas for the virtual machine VM $\underline{k}.i$:

1. The virtual addresses $m_{\underline{k}.i}$ and $m^*_{\underline{k}.i}$ for the accesses $\alpha[\pi_{\underline{k}.i}\mu_{\underline{k}.i}]$, $\alpha[\varepsilon_{\underline{k}.i}\mu_{\underline{k}.i}]$, $\alpha^*[\pi_{\underline{k}.i}\mu_{\underline{k}.i}]$, $\alpha^*[\varepsilon_{\underline{k}.i}\mu_{\underline{k}.i}]$, all accesses which are traditional read and write of memory data.
2. The virtual addresses by which the CPU controls the I/O devices: $s_{\underline{k}.i}$ and $s^*_{\underline{k}.i}$
3. The interrupts vectors: $v_{\underline{k}.i}$ and $v^*_{\underline{k}.i}$. The possible change of execution mode is done by fetching a new processor status word from one of the locations $v_{\underline{k}.i}$ or $v^*_{\underline{k}.i}$
4. The virtual addresses of the processor status word (accesses $\alpha[\pi_{\underline{k}.i}\pi_{\underline{k}.i}]$ and $\alpha^*[\pi_{\underline{k}.i}\pi_{\underline{k}.i}]$): $p_{\underline{k}.i}$ and $p^*_{\underline{k}.i}$.
5. The definitions of $r_{\underline{k}.i}$, $\bar{r}_{\underline{k}.i}$, and of $\bar{\mu}_{\underline{k}}$ and $\bar{\mu}^*_{\underline{k}}$ are similar to the ones given for one execution mode.

In most machines, simplifications will be made, e.g. $m_{\underline{k}.i} = m^*_{\underline{k}.i}$ or $v_{k=i} = v^*_{k.i}$ These are not needed for our present purpose.

In the P-mode the f^*_t mappings will be used, and in the NP-mode f_t will be used. This fact is familiar to the users of the DEC PDP 11/45 where different sets of memory management registers are used for different execution modes.

We make a last assumption: A P-mode VM $\underline{k}.i$ is mapped into a P-mode VM $\underline{k}$, and a NP-mode VM $\underline{k}.i$ is mapped into a NP-mode VM $\underline{k}$.

With these definitions, it is easy to generalize the Theorem 1 to the case with two execution modes:

Theorem 2.

A sufficient condition for the proper working of a VM system with two execution modes is:
1. The representation of the realization maps for each mode resides in the VMM address space :

$\rho(f_t[\mu_{\underline{k}.i}]) \subseteq \bar{\mu}_{\underline{k}}$ and $\rho(f^*_t[\mu_{\underline{k}.i}]) \subseteq \mu^*_{\underline{k}}$

This must be true at all time t.

2.If the mapping is not defined when used at time t, it generates a VM fault in the VMM address space, resp. $\bar{\mu}_{\underline{k}}$ and $\bar{\mu}^*_{\underline{k}}$.

3. The VMM is correct.

This result suggests that the two execution modes are completely separated. One mode is not used to "control" the other one. The only function of the P-mode is to implement different instructions than the NP-mode instructions. What these instructions are is independent of the virtualization of the machine.

We can conclude that execution modes are virtual execution modes in the case where all accesses are made through virtual memory locations. No interpretive execution of the privileged instructions is needed. This is true for the particular implementations described earlier (HV, Display, Multiple Copies of RMRs), and shows formally their superiority on the classical third generation architectures.

<u>Third case: Two execution modes with sensitive instructions.</u>

We assume now that the accesses $\alpha[\pi\pi]$ and $\alpha^*[\pi\pi]$ (which change the execution mode) are made through a status register accessed by a special indtruction (similar to the LPSW on the IBM S/360). The address of the register is absolute and a VM using the instruction will modify its contents. In this case the access is not made through the virtual memory mappings. The scheme of operation is schematized on Fig. 6.

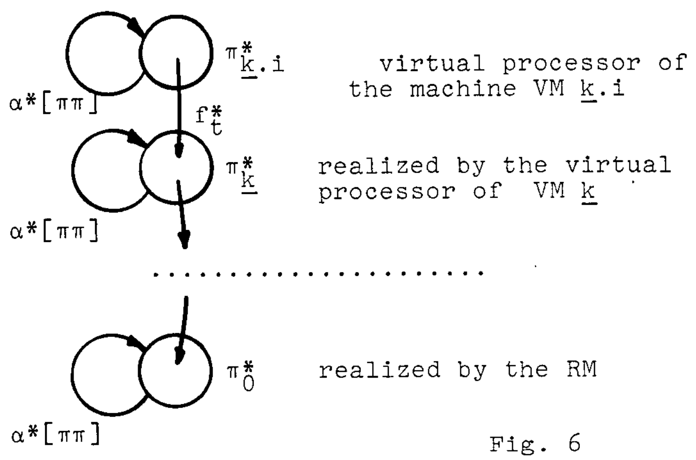
$\pi^*_{\underline{k}.i}$
virtual processor of the machine VM $\underline{k}.i$
$\alpha^*[\pi\pi]$
f^*_t
$\pi^*_{\underline{k}}$
realized by the virtual processor of VM $\underline{k}$
$\alpha^*[\pi\pi]$
π^*_0
realized by the RM
$\alpha^*[\pi\pi]$

Fig. 6

Whenever $\pi^*_{\underline{k}.i}$ executes the special instruction accessing the status register, the access $\alpha^*[\pi\pi]$ is performed in the machine VM $\underline{k}$.i. But since the register is common to all Virtual Machines, this will cause the execution of an access $\alpha^*[\pi\pi]$ at all levels or, said otherwise, this will cause a change of the execution mode of all the Virtual Machines, including the Real Machine VM 0. Such a situation does not verify the third characteristic of a VM system (namely that the VM environments be separated) and must therefore be prevented.

The classical solution to this problem proceeds as follows :
1. All the VMs are always in the NP-mode, including the Real Machine.
2. When an access to the status register is made, it generates a VM fault at level 0, and the VMM at that level can interpret the fault and take the appropriate actions. This is possible only if :
  a. The special instruction is a privileged instruction, i.e. it generates a VM fault when executed in NP mode.
  b. The VM fault vector of the privileged instruction provokes a change of mode to P-mode, and branches to a location in the VMM space of level 0 (i.e. in $\overline{\mu}^*_0$).

The "special" instruction considered hereabove is a particular case of a class of instructions called sensitive instructions*. A sensitive instruction is any instruction that bypasses the accesses made through the realization maps. It includes the following examples:
1. Access to an absolute processor status word.
2. Access to physical absolute address (bypass of the memory map) or access to memory by another mapping (Move from Previous Instruction/Data Space, DEC PDP 11).
3. Instructions whose effect depends on the physical address (the address is virtual but the effect of the instruction depends on the physical address into which it is mapped).
4. Access to I/O devices by absolute address of status and command words.
5. Access to memory made in absolute address by I/O devices.
6. Interrupt vectors in absolute addresses.
7. Interrupts provoking automatically a change of an absolute processor status word.
8. Access to absolute memory management registers.

The list is not exhaustive, Actually, in this work, contrary to what is done in [6], a sensitive instruction is not defined "positively": it is defined as anything that does not behave in a "virtual" manner.

If one generalizes the solution given hereabove to the virtualization of machines with sensitive instructions, one can state the following theorem (due to Popek and Goldberg)[6]:

Theorem 3.

A sufficient condition for the proper working of a VM system with sensitive instructions is:
1. All sensitive instructions must be privileged instructions (they must generate a VM fault when used in NP-mode).
2. The VM fault provokes a change to P-mode of the Real Machine and branches to a location in $\overline{\mu}^*_0$, the address space of the VMM at level 0.

If is evident that in spite of its simplicity, this theorem defines a virtualization that will involve interpretive execution of the privileged instructions. This has two n gative consequences;
1. It decreases the performances of the system.
2. It adds to the complexity of the Virtual Machine Monitor (i.e. an increase in programming and maintenance costs).

CONCLUSIONS.

We have set forth a method of analysis of VM systems that proceeds in two steps :
1. The structure of the system is specified without concern for the sharing of resources, i.e. the number of resources is assumed to be unlimited.
2. We examine the restrictions needed to preserve the structure of the system while reducing the number of resources and establishing resource sharing. This leads to the formulation of sufficient conditions that the machine architecture should verify in order to support the VM system.

When considering a generalized resource mapping mechanism, one can observe that a VM system can be supported with a minimum of overhead and with a relative simplicity of the Virtual Machine Monitor functions. The problem of the resource management overhead can be resolved by suitable implementation of multiple copies of Resource Management Registers. If one uses classical machine architectures with a lot of complex and not easily defined sensitive instructions, one obtains both a significant inefficiency of the system and a definite complexity of the design.

From this a lesson can be drawn: simple and sound hardware enhances simple software structures, "wild" architectures generate "wild" programming problems.

REFERENCES.

1. Belpaire, G. and Hsu N.-T. Hardware Architecture for Recursive Virtual Machines. Proc. of the ACM '75 Annual Conference, Minneapolis, Minn., October 1975.

* Defined by Popek and Goldberg [6]. Their concepts do not include completely ours but we keep the same name.

2. Belpaire, G. and Wilmotte, J.P. Correctness of Realizations of Levels of Abstraction in Operating Systems. in Operating Systems, Proc. of an Intl. Symposium, Rocquencourt, April 1974. Gelenbe, E. and Kaiser, C. (Eds.), Lecture Notes in Computer Science 16, 1-14, Springer-Verlag, Berlin-Heidelberg, 1974.

3. Goldberg, R.P. Hardware Requirements for Virtual Machine Systens. Fourth Hawaii International Conference on System Sciences, Honolulu, Hawaii, Jan. 1971, pp. 449-451.

4. Goldberg, R.P. Architecture of Virtual Machines. Proc. NCC 1973, AFIPS Press, Montvale, N.J.,pp. 309-318.

5. Lauer, H.C. and Wyeth, D. A Recursive Virtual Machine Architecture. Technical Report #54, Computing Laboratory, University of Newcastle upon Tyne, G.B., 1973.

6. Popek, G.J., and Goldberg R.P. Formal Requirements for Virtualizable Third Generation Architectures. Comm. ACM 17, 7, July 1974, pp. 412-421.

7. Mesarovic, M.D., and Takahara, Y. General Systems Theory: Mathematical Foundations. Academic Press, New York, 1975.

8. Goldberg, R.P. Architectural Principles for Virtual Computer Systems. Technical Report #24-72, Center for Research in Computing Technology, Harvard University.