WIS-CS-75-242

COMPUTER SCIENCES DEPARTMENT
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Received February 27, 1975

HARDWARE ARCHITECTURE FOR RECURSIVE VIRTUAL MACHINES

Gerald Belpaire

and

Nai-Ting Hsu

Technical Report #242

February 1975

ABSTRACT

In order to support Virtual Machine (VM) Systems on most current computer systems, the Virtual Machine Monitor (VMM) is the only process allowed to reference directly a set of registers, called resource management registers (RMR). Any access operations to those registers executed by any other processes (i.e. processes running on VMs) must be detected and executed interpretively by the VMM. This way of implementing VMs is imposed by inadequate hardware mechanism for protection. In this paper, this problem is studied and a hardware architecture is proposed to solve this problem in an efficient and elegant way. The hardware architecture proposed is a stack of RMR, one for each level of VM. The processes running on a VM can only access the RMRs of their level. The hardware (Control Unit) will use the information stored in the RMR of the current level and in the RMRs of lower levels for resource mapping.

Key Words and Phrases: Virtual Machine (VM), Virtual Machine Monitor (VMM), Resource Management Registers (RMR).

"A good wife can never cook without food"

A chinese proverb

I. INTRODUCTION

Several papers have discussed the ideas and usefulness of virtual machine systems [1,6,11,12] as well as the hardware and/or software needed to support them [1,3,4,5,9,11,12,13,14]. But there is still a number of problems associated with the virtual machine systems that are either not solved or given unsatisfactory solutions. This paper presents a hardware architecture to solve in an elegant and efficient way some of the problems currently faced.

The Virtual Machine Monitor (VMM) is a software module used to implement a virtual machine system. It has three essential characteristics [13]. First, the VMM provides an environment for programs essentially equivalent to the host machine; second, program execution in this environment shows only minor decrease in performance; and third, the VMM is in complete control of the system resources. The execution environment supported by the VMM is a Virtual Machine (VM). On most current computer system the last characteristic forces the VMM to be the only process allowed to reference directly a set of registers, called resource management registers. Any access operations to those registers executed by any other processes (i.e. processes running on VMs) must be detected and executed interpretively by the VMM. For those machines with supervisor/user modes, this results in the restrictions that a set of instructions* must be a subset of the privileged instructions [13], and that the VMM must be the only process allowed to run in supervisor mode. Other processes must run in user mode. These restrictions are due to the improper supporting environment (i.e. hardware architecture).

In this paper, we first define the resource mapping relation between the virtual machine and real machine, discuss the problem in supporting virtual machine, then propose a hardware architecture which can support VM system efficiently and illustrate the hardware architecture by using an example.

---

*to be defined later

II. THE f-map AND THE ϕ-map :

Let $R = (r_1, r_2, \ldots, r_n)$ be the set of resources of the real machine, and $V = (v_1, v_2, \ldots, v_m)$ be the set of resources of a virtual machine. The function of the VMM is to dynamically build up a relation between V and R so that the processes running on VM can progress properly. More formally, the VMM maintains a resources mapping for each VM it supports. The mapping, called f-map , is from V to R and {t} (i.e. $f : V \to R \cup \{t\}$) . At any time if $v_i \in V$ and $r_j \in R$ then

$$f(v_i) = \begin{cases} r_j & \text{if } r_j \text{ is the resources of the real machine corresponding to } v_i \\ t & \text{if } v_i \text{ does not have a corresponding resources .} \end{cases}$$

When a virtual machine, for its own execution, uses a resources v_i , the real resource that will be used is the one obtained by the f-map : $f(v_i)$. If $f(v_i) = t$, when used, then it causes a trap to some trap handling routine of the VMM. This is called a VM-fault [5].

For most current computer systems, the operating system provides an execution environment, called extended machine, to the users. There is also a mapping relation between the set R and the set $E = (e_1, e_2, \ldots, e_\rho)$, the set of resources of an extended machine. This mapping, called ϕ-map , is a map from E to R and {t'} (i.e. $\phi : E \to R \cup \{t'\}$) such that if $e_k \in E$ and $r_j \in R$ then

$$\phi(e_k) = \begin{cases} r_j & \text{if } r_j \text{ is the resources of the real machine corresponding to } e_k \\ t' & \text{if } e_k \text{ does not have a corresponding resources .} \end{cases}$$

When an extended machine, for its own execution, uses a resources e_k , the real resources that will be used is the one obtained by the ϕ-map : $\phi(e_k)$. If $\phi(e_k) = t'$, when used, then it causes a trap to some trap handling routine of the operating system. This event is called an exception [5].

If the operating system runs on a VM, the set E will be mapped to the set V or {t'} by the ϕ-map and the set V will be mapped to the set R or {t} by the f-map . The total effect is the composition of two maps, i.e. $f \cdot \phi : E \to R \cup \{t\}$, if no exception happened for the ϕ-map .

III. HARDWARE FOR RECURSIVE VIRTUALIZATION:

If the f-map and the ϕ-map are implemented on the same machine, the resource mapping mechanism for the f-map and the ϕ-map will be similar. On current system the map (either the ϕ-map or the f-map) is represented by a data structure stored in the memory or in some registers. In order to have the map referenced directly by the hardware without the intervention of software, there should be some hardware mechanism that can access these data structures. Without loss of generality, we can consider that the map is stored distributively in two places, RMR and L. RMR is a set of special registers, called resource mapping registers. L is a set of memory locations which can be referenced by the hardware through the information stored in RMR. For example in a virtual memory system with segmentation, the segment descriptor base register belong to RMR and the segment table is stored in main memory. Let I be the set of access instructions to RMR. As defined in [3,4,13], the set I is called the sensitive instructions set. The instructions to access L will be the ordinary memory access instructions.

In order to keep the integrity of the system, the VMM or the operating system is the only process which should be allowed to reference or update the information stored in RMR and in L. This can be achieved in two ways: either the information stored in RMR and in L (i.e. the f-map or the ϕ-map) specifies an execution environment which does not include RMR and L, or the information stored in RMR and in L specifies an execution environment which does not include L and the ability to execute instructions in I. In either way, if other processes try to reference RMR or L either a trap or no operation will result (i.e. the map has the function of resource mapping and protection).

In virtual machine system the VMM uses RMR and L to build up an execution environment for a virtual machine. The operating system running on a VM also wants to use a copy of RMR and L to build up an execution environment for its extended machines. Unfortunately on current architectures there is only one instance of RMR in hardware. This forces

the VMM to simulate a copy of RMR for the VM and the access to RMR on the virtual machine should be detected and executed interpretively by the VMM. In computer systems with supervisor/user modes, this results to the restrictions that the set I should be a subset of the privilege instructions [13] and that the VMM should be the only process which can be executed in supervisor mode.

If 1. the virtual machine can have another copy of RMR, 2. the processes running on a VM can only reference to this copy of RMR, and 3. the control unit of CPU can link these two RMRs properly, then the restrictions mentioned above [13] disappear. This idea is analog to the idea of implementing recursive procedures by allocating one copy of local data area for each level of nesting.

The above idea can be generalized to support more than one level of virtual machines. Because the VMM supports a virtual machine with equivalent properties as the real machine, it is definitely possible to run the VMM on a virtual machine in order to support another level of virtual machine.

For convenient discussion, let the real machine be the level 0 and any VMM running on the VM of level i will support a new level of VM, level i+1. Suppose we want to build up a computer system to support n(>0) levels of virtual machines. The supplementary hardware needed consists of n+1 copies of RMR (one copy for the level 0 (real) machine and one copy for each level of virtual machine), a level indicator which indicates the current level of execution, and a Control Unit. The function of the Control Unit is

a. to link the proper copy of RMR to the current running level and prohibit any access action to those RMRs which belong to lower levels,
b. to perform resources mapping composition through different levels of RMRs,
c. to route the fault trap to the VMM at the appropriate level.

As mentioned hereabove, it is clear that RMR includes all registers of the CPU used for resource mapping and protection. For example on most third generation computer systems the RMR includes the execution

mode indicator, the memory management registers (e.g. relocation-bound register) and the interval timer.

IV. EXAMPLE:

This example is used to explain the idea more clearly and to show how the instructions of the Set I can be executed directly by the hardware and not interpreted by software. The Pascal programming language [15] is used in the following examples to represent the structure and the function of the hardware. The aim of these programs is to explain the ideas without indicating any particular hardware implementation.

Consider a computer with two modes of execution, i.e. supervisor/user modes, using one relocation-bound register for memory management*. Two kinds of faults are handled in this example. One is the memory trap and the other is halt. There are fixed words reserved in memory which served as fault vectors. In order to support n levels of virtual machine systems, the hardware needs to have a level indicator, n+1 copies of RMR which contains the mode indicator and the relocation-bound register, and a memory management unit which can map the address generated by a process to the address of real memory. In order to represent the program symmetrically, we assume that there is a fictitious level, level -1 , beneath level 0 , the real machine. The contents of relocation-bound register of level i-1 specify the memory space of the virtual machine of level i on the memory space at level i-1 .

1. The data structure

```
type   number of level = -1..n;
type   memory address = 0..memory size -1;
type   execution mode = (supervisor,user);
var    PC :     memory address;
var    level :  number of level;
var    memory : array [memory address] of store;
                {Assuming store is a primitive type compatible with
                other scalar type.}
var    fixl :   memory address;
                {starting location in memory to save old copies of
                RMR for memory trap}
```

*This idea can be extended to include any kind of memory management methods (e.g. paging and segmentation).

```
var    fix2 :   memory address;
                {Starting location in memory to fetch the new copy
                of RMR when a memory trap occurs}
var    fix3 :   memory address;
                {Starting location in memory to fetch the new copy
                of RMR when a halt interrupt occurs}
var    RMR :    array   [number of level] of
                record   mode :  execution mode;
                         base :  memory address;
                         bound : memory address;
                end
```

2. The memory management unit

```
function   real address (addr : memory address) : memory address;
var    i : number of level;
var    temp : memory address;
begin
       temp := addr;
       for  i := level downto -1 do
            with RMR [i] do
                if temp > bound  then  memory trap (i)
                                 else  temp := temp + base;
       real address := temp;
end
```

This function maps the address, *addr*, generated by a process running on a virtual machine to the real memory address.

3. The memory trap routine

```
procedure   memory trap  (i : number of level);
var    j,k : number of level;
var    t : memory address;
begin
       if   i = -1  then  halt(-1); {Reference to nonexisting physical
                                     location, it is an error of VMM
                                     on level 0}
```



```
       j := level;
       level := i-1;
       memory (real address (fix1)) := j-i;
       memory (real address (fix1 + 1)) := PC;
       t := fix1 + 2;
       for   k := i   to   j   do
            begin with   RMR [k]   do
                 begin   memory (real address (t)) := mode;
                         memory (real address (t+1)) := base;
                         memory (real address (t+2)) := bound;
                 end
                 t := t + 3;
            end
       PC := memory (real address (fix2));
       with   RMR [i]   do
            begin   mode  := memory (real address (fix2 + 1));
                    base  := memory (real address (fix2 + 2));
                    bound := memory (real address (fix2 + 3));
            end
       level := i;
end
```

This procedure specifies what should be done by the hardware when a memory trap occurs on the i^{th} level virtual machine. The value *level -i* will be saved with the value RMR [i] , RMR [i+1],..., RMR [level] . A routine running on level i is activated by the hardware to handle the memory trap. The processes below level i are not affected by the occurrence of the memory trap. The value *level -i* is saved instead of *level* for reason of symmetry. The VMM running on any *level* can assume that it is running on the real machine, (level 0). There is no way for the VMM to detect which level it is currently running on.

The contents of *base* and *bound* in RMR [i-1] specify the memory space of the virtual machine on level i. The value of *fix1* and *fix2* is fixed for every level. It is the responsibility of the memory management

unit to map them to the proper memory location.

4. The Load RMR instruction

```
procedure   Load RMR (loc : memory address);
var     temp1, temp2, temp3 : store;
var     j,k : number of level;
var     t : memory address;
begin
        k := memory (real address (loc));
        if   k = -1   then   halt (level -1);
        PC := memory (real address (loc + 1));
        t := loc + 2;
        temp1 := memory (real address (t));
        temp2 := memory (real address (t + 1));
        temp3 := memory (real address (t + 2));
        for   j := 1   to   k   do
              begin
                    t := t + 3;
                    with   RMR [level + j]   do
                         begin   mode  := memory (real address (t));
                                 base  := memory (real address (t + 1));
                                 bound := memory (real address (t + 2));
                         end
              end
        with   RMR [level]   do
              begin   mode  := temp1;
                      base  := temp2;
                      bound := temp3;
              end
        level := level + k;
end
```

This instruction is used to update the contents of *level*, PC and RMR. The value of the variable k specifies the number of level to be updated. If k=0 , only the RMR of the current level will be updated.

If k>0 , then RMR [level],..., RMR[level + k] will be updated and the next instruction to be executed will be on level *level + k* . If k = -1 , the current virtual machine is halted and the control returns to the VMM supporting the virtual machine. Load RMR can be used by the VMM to run a new virtual machine and the new PC may point to a bootstrap program.

5. The halt interrupt

```
procedure   halt (i : number of level);
begin
        if   i = -1   then   stop; {stop the real machine}
        level := i - 1;
        PC := memory (real address (fix3));
        with   RMR [i]   do;
             begin   mode  := memory (real address (fix3 + 1));
                     base  := memory (real address (fix3 + 2));
                     bound := memory (real address (fix3 + 3));
             end
        level := i;
end
```

In the machine described here the halt interrupt, the memory trap, and the Load RMR are the only operations which can update both RMR and *level*. All other instructions of the set I can only reference the RMR of the current level. Actually the interrupt and trap functions can not be properly represented by a procedure, because after completion, control will not return back to the caller but will start to execute a program in a new environment.

6. The JRST 1 (return to user mode) instruction

```
procedure   JRST 1;
begin   RMR [level].mode := user   end
```

This instruction is given only for illustration. On the PDP 10, such an instruction is not a privileged instruction. There is therefore

a Sensitive instruction (i.e. belonging to the Set I) that is not a privileged instruction. Thus, under the restrictions mentioned above [13], the PDP 10 cannot support a virtual machine system. On a machine with the new architecture, there are no such problems.

V. PROTECTION PROPERTIES:

As discussed above, the restrictions imposed on most current computer systems to support virtual machine systems are due to inadequate protection mechanism. The nesting of VMM and virtual machines form a process tree (Fig. 1). The nonterminal nodes correspond to the VMM

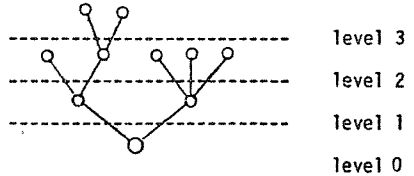

Fig. 1

and the leaves correspond to the processes running on virtual machines. The protection properties of this kind of system should have the following characteristics:

a. The execution environments of each node are equivalent.
b. The only processes known to a parent are its children processes. What kind of function its children processes perform is irrelevant to the parent, if they are working properly within their environments.
c. The execution environments of the children processes are contained in the execution environment of their parent process.
d. The same kind of protection mechanism are used in each node to build up protection walls among it as well as its children.
e. Process on each node can build up protection within itself. For examples, the VMM can build up protection within itself, the operating system running on a VM can build up protection for its user without going up one level of nesting (i.e. the operating system and the user's processes are still within one leave node).

Multiple copies of RMR provide an efficient way to support inter-node protection (i.e. the characteristics a,b,c, and d). In order to achieve property d , the structure of RMR needs to contain some kind of protection mechanism for intra-node protection. The kind of protection mechanism needed depends on the particular design, it can be ranged from the simplest priviled/user modes to complex protection mechanisms such

as capabilities [2,16].

VI. COMPARISON WITH OTHER ARCHITECTURES:

At least two other kinds of architecture were proposed to support VM systems. One is the Lauer's Recursive Virtual Machine Architecture [9] and the other one is the Goldberg's Hardware Virtualizer (HV) [5]. In Lauer's design, the f-maps are segment tables. The hardware maintains a display (actually a vector of addresses) which specifies where the f-maps, defining the currently used execution environment, of each level are stored in main memory. In Goldberg's HV, the f-maps are identified by names, stored in the identifier register, instead of addresses, and the machine can reference the f-maps by name. Both designs are special cases of the design proposed here. The entries of the display or the identifier register are the corresponding RMRs.

Besides their lesser generality, they have other disadvantages. The most serious one is that there is no intra-node protection at all. This forces the virtual machines to be the units of protection and modularity (i.e. no protection exists within the VMM itself). Indeed it is more expensive to support a system by nesting multiple levels than with only one level, because of the time consuming resource map composition. This will limit their applications to other fields such as operating systems, or secure systems.

The other disadvantage is that in Goldberg's design the f-map and the ϕ-map are implemented with different kinds of mechanism. The Hardware Virtualizer (HV) can only be used to construct the f-maps, it cannot be used to construct the ϕ-map. This means that it does not really support a VM system (i.e. the VMM execution environment is different from the real machine environment)

VII. CONCLUSION:

Multiple copies of RMR to support VM systems is a direct consequence of considering the way to implement multiple levels of f-map and ϕ-map. It provides an execution environment which is a combination of nested-address-space system (for inter-node protection) [10] and global-object-name system (for intra-node protection), e.g. capability.

Actually the multiple copies of RMR is equivalent to a stack of RMR. The way to implement it will depend on the design. It can be totally implemented in fast access registers. Or only the top entries of the stack will be in registers and the rest of them in core memory. The hardware can link and update them properly. In order to increase the performance in resources mapping, the composition map of current execution environment can be stored in some associative memory.

Not only this design can be used efficiently to support VM systems, but also it is possible to support other systems such as operating systems, or secure systems. One of the future research work is to explore and exploit its applications to other fields.

References

1. Buzen, J. P., and Gagliardi, U. O., The evaluation of virtual machine architecture, Proc. NCC 1973, AFIPS Press, Montvale, N.J., PP. 291-300.

2. Ferrie, J., Kaiser, C., Lanciaux, D., Martin, B., An extensible structure for protected systems design, Colloques IRIA, Protection in Operating Systems, France, Aug. 13-14, 1974, PP. 83-105.

3. Goldberg, R. P., Virtual machine systems, MIT Lincoln Laboratory Rept. No. MS-2686 (also 28L-0036), Lexington, Mass., 1969.

4. Goldberg, R. P., Hardware requirements for virtual machine systems, Fourth Hawaii International Conference on System Sciences, Honolulu, Hawaii, Jan., 1971, PP. 449-451.

5. Goldberg, R. P., Architecture of virtual machines, Proc. NCC 1973, AFIPS Press, Montvale, N.J., PP. 309-318.

6. Goldberg, R. P., Survey of Virtual Machine Research, COMPUTER, June 1974, PP. 34-45.

7. IBM Virtual Machine Facility/370 Planning Guide, IBM Corporation, Publication No. GC20-1801-0, 1972.

8. IBM Virtual Machine Facility/370: Release 2 Planning Guide, IBM Corporation, Publication No. GC20-1814-0, 1973.

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

10. Lauer, H. C., Protection and hierarchical addressing structures, Colloques IRIA, Protection in Operating Systems, France, Aug. 13-14, 1974, PP. 137-148.

11. Meyer, P. A., and Seawright, L. H., A Virtual Machine Time-Sharing System, IBM Systems Journal, Vol. 9, No. 3, 1970, PP. 199-218.

12. Parmelee, R. P., Peterson, T. I., Tillman, C. C., and Hatfield, D. J., Virtual storage and virtual machine concepts, IBM Systems Journal, Vol. 11, No. 2, 1972, PP. 99-130.

13. Popek, G. J., and Goldberg, R. P., Formal Requirements for Virtualizable Third Generation Architectures, CACM Vol. 17, No. 7, July 1974, PP. 412-421.

14. Srodawa, R. J., and Bates, L. A., An efficient virtual machine implementation, Proc. NCC 1973, AFIPS Press, Montvale, N.J., PP. 301-308.

15. Wirth, N., The Programming Language Pascal, Acta informatica 1, 1971, PP. 35-63.

16. Wulf, Cohen, Corwin, Jones, Levin, Pierson, and Pollack, HYDRA: The Kernel of a Multiprocessor Operating System, CACM Vol. 17, No. 6, June 1974, PP. 337-344.